Foundations of Sequence Analysis

Lecture notes for a course in the Wintersemester 2022/2023

Stefan Kurtz

October 16, 2022

Contents

Preface

This document represents the lecture notes of the master course *Foundation of Sequence Analysis* which will be held at the University of Hamburg between October 2022 and March 2023. Much effort was put into this document to ease understanding. For example, there are 72 examples, 9 tables, 54 figures, and 20 illustrations explaining the different notions and methods. The layout of the figures is often not optimal, which is due to the fact that they are also used in slides, which require a different format.

Besides the colleagues referenced, many more people made contributions. I would especially like to thank Sascha Steinbiss and Gordon Gremme for the many figures and examples they provided. Jörg Winkler did a very nice job in typesetting the figures in tikz-format. Stine Griep, Jan Oliver Fröhlich and Jörg Winkler pointed to several inconsistencies and errors in a previous version of this document. All this help is very much appreciated.

Hamburg, October 15, 2022

Stefan Kurtz

1 Overview

Figure 1: Sources of sequences

- molecular biology

DNA	...`aactacgt`...	4 nucleotides,	length: $\approx 10^3 - 10^9$
RNA	...`aacuacgu`...	4 nucleotides,	length: $\approx 10^2 - 10^3$
proteins	...`LISAISTLIEB`...	20 aminoacids,	length: $\approx 10^2 - 10^3$

`L` = Leucine, `I` = Isoleucine, `S` = Serine, `A` = Alanine, etc.

- text processing, like abstract of papers, laboratory books, medical reports
- 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.
- information transmission: sequence of bits, blockcodes
- phonetic spelling: e.g. english with 40 phonemes; japanese with 113 "morae" (syllables)
- spoken language: discretized measurements, multidimensional (specifying frequency and energy) on a dynamic time scale

1.1 Application areas

Sequences or equivalently texts, strings, or words are a natural way to represent information. Figure 1 contains a short list of areas where sequences to be analyzed come from. For more examples, see [KS83], a classical textbook on string processing problems.

There are thousands of different algorithms and data structures for handling biological sequence data, many of which have originally been developed in the bioinformatics and computational biology community. However, in many cases methods applied to biological sequences are adapted from methods developed for sequences from other sources. For example, many methods for fast index based search in biological sequence databases are adapted from methods originally developed for indexing web-pages. For this reason, it is often a good idea to look at methods developed in computer science before developing an own method. Often the best and most widely used methods in Bioinformatics are developed by an interdisciplinary interplay of computer scientists, bioinformaticians and wet-lab scientists.

When it comes to analyzing sequences one common feature is the following: Usually sequences encoding experimental or natural information are almost always inexact. Thus similar sequences have, in many cases, the same or similar meaning or effect. For this reason, a main part of this lecture will be devoted to notions of similarity of sequences, and we will show how to handle these notions algorithmically

Figure 2: Some problems relevant for sequences

1. sequence comparison: compare two sequences and show the similarities and differences. Example: how similar is gene A to gene B?
2. sequences matching: find all positions in a sequence where a pattern sequence occurs. Example: Where does the pattern `TATAA` occur in my genome?
3. regular expression matching: find all positions in a text where a regular expression matches. Example: Where do the codons `G[AG]T` occur in my gene?
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. Example: Map all my 10^8 short reads of length 100 bp to the mouse genome of $2.9 \cdot 10^9$ bp
6. dictionary matching: for a given word w find the word v in a given set of words with maximal similarity to w. Example: find the nearest neighbor of my oligomer of length 18 in the Protein Family Database.
7. data compression: find long duplicated substrings in a given text. Example: find a representation of my 1 000 genomes without any redundancies
8. data compression: sort suffixes of a given sequence lexicographically. Example: create an index structure of my genome which allows to quickly map my reads to the genome
9. structural pattern matching: find regularities in sequences, like repeats, tandems ww, palindromes, or unique subsequences. Example: find all maximal repeats of minimum length 100 with at least 97% identity in my genome.

1.2 Problems on strings

Figure 2 gives a short list of problems relevant for biological sequences.

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] and [Pev00] are also recommended. The textbook [Ohl13] covers some advanced topics in sequence analyses which will partly be considered in the genome informatics lecture.

2 Basic Notions and Definitions

Let S be a set, i.e. a collection of items of the same kind, without duplicates. We can have sets of numbers, sets of characters, sets of strings, sets of pairs of numbers, etc. $|S|$ denotes the number of elements in S.

$\mathbb{N}$ denotes the set of positive integers including 0. $\mathbb{R}^+$ denotes the set of positive real numbers including 0. The symbols $h, i, j, k, l, \ell, m, n, q, r$ 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. Sometimes we omit the multiplication operator and e.g. write something like $2n + 3m$ instead of $2 \cdot n + 3 \cdot m$.

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 for the sequence containing no characters: ε denotes the *empty sequence*. Concatenation of sequences u and v means to write u before v without any separator. So uv is the concatenation of u and v. 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

$$\mathcal{A}^* = \bigcup_{i \geq 0} \mathcal{A}^i = \mathcal{A}^0 \cup \mathcal{A}^1 \cup \mathcal{A}^2 \cup \ldots$$

where $\bigcup$ is the union-operator and

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

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 are composed by some character from $\mathcal{A}$ prepended to some sequence of length $i - 1$ from $\mathcal{A}^{i-1}$.

You all should have seen the notation for sets, in which the elements of a set are specified and enclosed in curly brackets, like in

$$\{1, 3, 5, 7, 9\}$$

specifying the set of all odd numbers smaller than 10.

In many cases, one wants to specify a set by properties satisfied by all elements in the set, for example:

$$\{i \mid i \in \mathbb{N}, i < 10, i \text{ is odd}\} \text{ or shorter } \{i \in \mathbb{N} \mid i < 10, i \text{ is odd}\}$$

This specifies the same set as above. The elements are specified by a variable i and properties referring to i given after the symbol $\mid$. Instead of this symbol, other authors often use a colon $:$ instead. The properties to be satisfied are separated by commas.

Formal definitions, like the one for $\mathcal{A}^i$, are very helpful to derive and prove properties of the defined items, as shown in the following lemma.

Lemma 1 For all $i \geq 0$, $|\mathcal{A}^i| = |\mathcal{A}|^i$.

Proof: For $i = 0$ we have $|\mathcal{A}^i| = |\mathcal{A}^0| = |\{\varepsilon\}| = 1 = |\mathcal{A}|^0$, that is, the lemma holds for $i = 0$. Suppose that $|\mathcal{A}^{i-1}| = |\mathcal{A}|^{i-1}$ holds for a fixed but arbitrary $i > 0$. Then, we can conclude

$$\begin{aligned} |\mathcal{A}^i| &= |\mathcal{A}| \cdot |\mathcal{A}^{i-1}| && \text{(by Definition of } \mathcal{A}^i = \{aw \mid a \in \mathcal{A}, w \in \mathcal{A}^{i-1}\}) \\ &= |\mathcal{A}| \cdot |\mathcal{A}|^{i-1} && \text{(by assumption)} \\ &= |\mathcal{A}|^i && \text{(by evaluation).} \end{aligned}$$

By the principle of induction this shows that the lemma holds for any $i > 0$ which completes the proof. □

$\mathcal{A}^+$ denotes $\mathcal{A}^* \setminus \{\varepsilon\}$, where $\setminus$ means subtraction of sets. 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. $\{\mathtt{A}, \ldots, \mathtt{Z}, \mathtt{a}, \ldots, \mathtt{z}, \mathtt{0}, \ldots, \mathtt{9}, \}$: alphanumeric subset of the ASCII-set
3. $\{\mathtt{A}, \ldots, \mathtt{Z}\} \setminus \{\mathtt{B}, \mathtt{J}, \mathtt{O}, \mathtt{U}, \mathtt{X}, \mathtt{Z}\}$: letter code for 20 amino acids
4. $\{\mathtt{a}, \mathtt{c}, \mathtt{g}, \mathtt{t}\}$: DNA alphabet (Adenine, Cytosine, Guanine, Thymine)
5. $\{\mathtt{R}, \mathtt{Y}\}$: purine $(\mathtt{a}, \mathtt{g})$/pyrimidine $(\mathtt{c}, \mathtt{t})$-alphabet
6. $\{\mathtt{I}, \mathtt{O}\}$: 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.

While the context usually allows to distinguish variables for characters and strings from concrete characters and strings, we use different fonts for these. In particular, we use typewriter fonts for concrete characters and sequences, like `a`, `c`, `ac`, `tataa`. Variables denoting a character or a string, like a and w below are written in italic fonts.

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 $\mathcal{A} = \{\mathtt{b}, \mathtt{c}\}$. 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 determine the set

$\mathcal{A}^2$ by applying the above definitions as follows:

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

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 $s = \mathtt{acca}$. The suffixes of s are $\mathtt{acca}$, $\mathtt{cca}$, $\mathtt{ca}$, $\mathtt{a}$, and ε. The prefixes of s are ε, $\mathtt{a}$, $\mathtt{ac}$, $\mathtt{acc}$ and $\mathtt{acca}$. The only substrings of s which are not prefixes and not suffixes are $\mathtt{c}$ and $\mathtt{cc}$.

$s[i]$ is the ith character of s. That is, if $|s| = n$, then $s = s[1]s[2]\ldots s[n]$ where $s[i] \in \mathcal{A}$. $s[n]s[n-1]\ldots s[1]$, denoted by s^{-1}, is the *reverse* of $s = s[1]s[2]\ldots s[n]$. If $i \leq j$, then $s[i\ldots j]$ is the substring of s beginning with the ith character and ending with the jth 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, 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 problem 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.

Example 4 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 is a mathematical notation used to describe the asymptotic behavior of functions. It allows us to indicate that we do not care for constants and lower-order terms. Its purpose is to characterize a function's behavior for very large inputs in a simple but rigorous way that simplifies the comparison of algorithms concerning their running time and space requirements.

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.

Definition 1 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

$$0 \leq f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0$$

Note that the n_0 is the minimum problem size for which f is dominated by g. c is a constant, i.e. it cannot depend on n.

Example 5 Consider the functions f and g defined as follows:

$$f(n) = 6n^4 - 2n^3 + 5$$
$$g(n) = n^4$$

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

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

With $c = 13$ and $n_0 = 1$ we can conclude $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$. That is, we have found the constant c and minimum input size n_0 as required in Definition 1. So f is $O(g)$.

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 dependency of the running time of the algorithm on the input size is $O(n^2)$. Table 1 shows some commonly used functions for specifying asymptotic upper bounds. Table 2, Table 3 and Figure 3 give more explanations, shows sample values and plots of most of these functions.

Table 1: Some commonly used functions for specifying asymptotic upper bounds.

notation	name	example
$O(1)$	constant	determining cost of an edit operation (see Section 3.3)
$O(\log n)$	logarithmic	finding an element in a sorted array of length n
$O(n)$	linear	determining the identity of two sequences both of length n (see Section 3.2)
$O(n \log n)$	quasilinear	determining an optimal chain of n matches, see Genome Informatics lecture
$O(n^2)$	quadratic	determining the edit distance of two sequences both of length n (see Section 3.3.3)
$O(n^c)$, $c > 1$	polynomial	find secondary structure of RNA-sequence s with minimum free energy ($n = \|s\|$, $c = 3$, see Genome Informatics lecture)
$O(c^n)$	exponential	recursive counting of all alignments
$O(n!)$	factorial	enumerate all subsets of set of size n

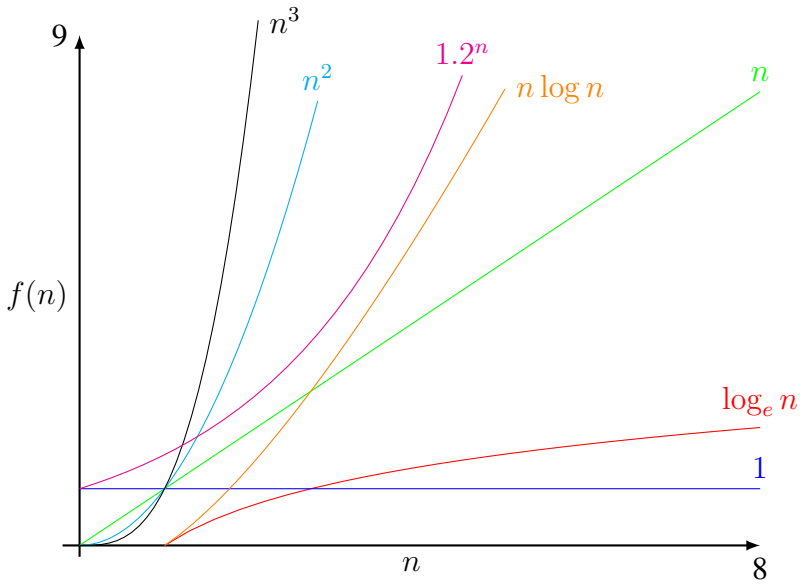
Table 2: Explanations of commonly used functions for O-Notation.

$O(1)$	All instructions of the program are executed at most only a few times, independent of the size of the input.
$O(\log n)$	The program gets slightly slower as n grows. Whenever n doubles, $\log n$ increases by a constant.
$O(n)$	Usually, a small amount of processing is done on each input element. Whenever n doubles, then so does the running time. This is the optimal situation for an algorithm that must process n inputs/produce n outputs.
$O(n \log n)$	This often arises in algorithms dividing the problem into smaller subproblems, solving them independently and then combining solutions. When n doubles the running time more than doubles (but not much more).
$O(n^2)$	This arises when algorithms process pairs of all data items (e.g. double-nested loop). Whenever n doubles, the running time increases fourfold.
$O(n^3)$	This arises when algorithms process triples of all data items (e.g. triple-nested loop). Whenever n doubles, the running time increases eightfold.
$O(2^n)$	Exponential run times arise naturally as solutions to problems using a "brute-force" approach. Whenever n doubles, the running time squares.

Table 3: Commonly used functions with sample values

$\log n$	$\sqrt{n}$	n	$n \log n$	n^2	n^3	2^n	$n!$
3	3	10	30	10^2	10^3	$1.02 \cdot 10^3$	$3.62 \cdot 10^6$
6	10	10^2	600	10^4	10^6	$1.27 \cdot 10^{30}$	$\approx 10^{158}$
9	31	10^3	$9 \cdot 10^3$	10^6	10^9	$1.07 \cdot 10^{301}$	$\approx 10^{2568}$
13	100	10^4	$1.3 \cdot 10^5$	10^8	10^{12}	very large	very large
16	316	10^5	$1.6 \cdot 10^6$	10^{10}	10^{15}	very large	very large
19	1 000	10^6	$1.9 \cdot 10^7$	10^{12}	10^{18}	very large	very large

Figure 3: Plots of some commonly used functions f for specifying asymptotic upper bounds shown for $n \leq 8$ and $f(n) \leq 9$.



3 Sequence comparisons

3.1 An overview of sequence comparison models

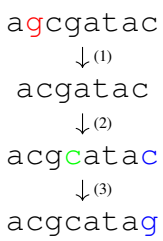
The comparison of sequences is an important operation applied in several fields, such as molecular biology, speech recognition, computer science, and coding theory. The most important model for sequence comparison is the model of edit distance. It measures the distance between sequences in terms of edit 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. Here is an example:

The sequence $u =$ `agcgatac` can be converted to $v =$ `acgcatag` as follows:

(1) delete the first g in u gives `acgatac`

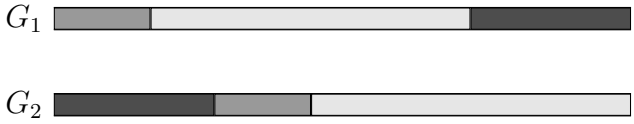
(2) insert c between g and a gives `acgcatac`

(3) replace the last c by g gives v



Such a sequence of edit operations (1,2,4) (also called alignment) can be computed in $O(mn)$ time (where m and n are the lengths of the two sequences), using the technique of dynamic programming (discussed later in full detail).

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 bacterial genome that has evolved from an ancestral genome by transposition of large sections of DNA, see the following illustration of two genomes G_1 and G_2, such that blocks of the same grey-scale are considered to be highly similar:



In such a case, the edit distance model should not be used since it gives a large edit distance, thus not revealing the similarity. There are many alternatives, so called alignment-free sequence comparison models, that are more appropriate for this case. In this lecture we consider three of these models: the maximal matches model, the q-gram model (see Section 3.9 and Section 3.10) and a model involving minHashes.

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 too expensive, while the order of the sequence characters is still important. Therefore heuristics have been developed, which approximate the edit distance model. These heuristics have been implemented in very popular programs like Fasta and Blast. 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 algorithms to compute the respective distances. For the rest of this section let u and v be sequences of length m and n, respectively. These are the sequences we compare.

3.2 The sequence comparison problem

The trivial method to compare two sequences is to compare them character by character:

$$u \text{ and } v \text{ are equal} \Leftrightarrow |u| = |v| \text{ and } u[i] = v[i] \text{ for all } i, 1 \leq i \leq n.$$

Thus, the equality can be determined in $O(n)$ time. However, this comparison model is too restrictive as it does not consider the requirements when comparing biological sequences:

- allowing for ambiguities in the genetic code, e.g. gcu, gcc, gca, and gcg all code for Alanin.
- tolerating errors in the sequencing technology
- variant calling: mapping reads which differ from the reference genome due to SNP or short indels
- homology search: searching for a protein with unknown function, a "similar" and not necessarily identical protein sequence, whose biological function is known.
- gene prediction: using the genes of a well-annotated genome to annotate a newly sequenced genome of a closely related species
- searching for a name of which the spelling is not exactly known
- finding inflected forms of a word
- accounting for typing errors

To generalize the trivial sequence comparison method, one defines a function $f : \mathcal{A}^* \times \mathcal{A}^* \rightarrow \mathbb{R}$, 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 and vice versa. Sequence comparison models based on distances minimize these. Sequence comparison models based on similarities maximize these. In the following, we define three distance models for sequences, the euclidean distance, the block distance and the hamming distance. The first two of these require integer alphabets.

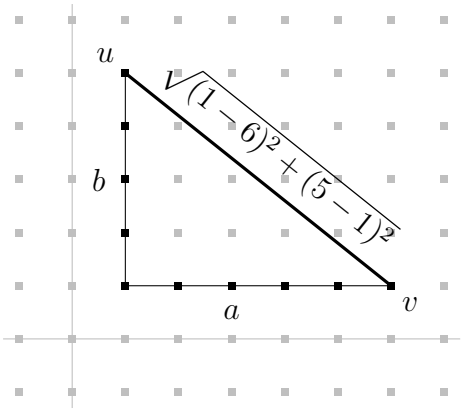
Definition 2 Let $\mathcal{A}$ be a finite subset of the integers. Suppose $n > 0$ and $M = \mathcal{A}^n$. Then we define the following distance notions:

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

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

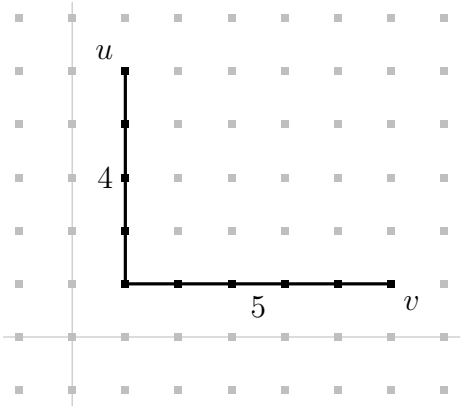
These distance notions only make sense for sequences of the same length and they require that the characters of the alphabet can be subtracted, which explains why we require that $\mathcal{A}$ consists of integers. The euclidean distance is the distance of the points (represented by fixed-length sequences of integers) in euclidean space of n dimensions.

Figure 4: Illustration of the euclidean distance for $n = 2$, $u = (1, 5)$ and $v = (6, 1)$. so $f_e(u, v) = \sqrt{(1-6)^2 + (5-1)^2} \approx 6.4$.



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

Figure 5: Illustration of the block distance for $n = 2$, $u = (1, 5)$ and $v = (6, 1)$. So $f_b(u, v) = |1 - 6| + |5 - 1| = 5 + 4 = 9$.



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

Example 6 Let $n = 2$, $a = u[1] - v[1]$, $b = u[2] - v[2]$, and c be the distance of u and v in $\mathbb{Z}^2$. Then, by the Pythagorean theorem, we have $a^2 + b^2 = c^2$ and

$$c = \sqrt{a^2 + b^2} = \sqrt{(u[1] - v[1])^2 + (u[2] - v[2])^2} = f_e(u, v)$$

Example 7 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 reach v when starting at u.

The euclidean distance and the block distance are rarely used for biological sequences, while the following is more common.

Definition 3 Let $\mathcal{A}$ be an alphabet, let $n > 0$ and $M = \mathcal{A}^n$. Then we define the hamming distance f_h as follows:

$$f_h(u, v) = |\{i \mid 1 \leq i \leq n, u[i] \neq v[i]\}|$$

The hamming distance counts the number of positions at which the sequences are different. It is only defined for sequences of the same length.

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

3.3 The edit distance model

The notion of edit operations, introduced in the early 1970's by Ulam [Ula72], is the key to the edit distance model.

Definition 4 An *edit operation* is a pair $(\alpha, \beta) \in (\mathcal{A}^1 \cup \{\varepsilon\}) \times (\mathcal{A}^1 \cup \{\varepsilon\}) \setminus \{(\varepsilon, \varepsilon)\}$. □

In other words, an edit operation is a pair (α, β) where $\alpha, \beta \in \mathcal{A}^1 \cup \{\varepsilon\}$ and $(\alpha, \beta) \neq (\varepsilon, \varepsilon)$. α and β denote *sequences* of length ≤ 1. However, each such string, whenever of length 1, is identified with the character it consists of.

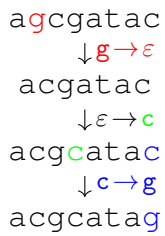
An edit operation (α, β) is usually written as $\alpha \to \beta$. This reflects the operational view which considers edit operations as rewrite rules transforming a source sequence into a target sequence, step by step.

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*.

Example 8 As in a previous example, we consider the transformation of $u =$ `agcgatac` into $v =$ `acgcatag`, this time using our notation for edit operations.



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 or similar sequences are. Often it is additionally of interest to represent the differences between two sequences as a collection of individual elementary differences [KS83]. The most important mode of such analyses is an alignment.

Definition 5 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

$$u = \alpha_1 \dots \alpha_h \text{ and}$$
$$v = \beta_1 \dots \beta_h.$$

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 9 The alignment $A = (\varepsilon \to \mathtt{t}, \mathtt{g} \to \mathtt{g}, \mathtt{c} \to \mathtt{a}, \varepsilon \to \mathtt{t}, \mathtt{a} \to \mathtt{a}, \mathtt{c} \to \varepsilon, \mathtt{t} \to \mathtt{t})$ of the sequences `gcact` and `tgatat` is written as follows:

$$\begin{pmatrix} \mathtt{-\,g\,c\,-\,a\,c\,t} \\ \mathtt{t\,g\,a\,t\,a\,-\,t} \end{pmatrix}$$

Example 10 Five alignments of $u = \mathtt{GATH}$ and $v = \mathtt{GCDHT}$:

$$A_1 = \begin{pmatrix} \mathtt{G-A\,T-H-} \\ \mathtt{G\,C--D\,H\,T} \end{pmatrix} \quad A_2 = \begin{pmatrix} \mathtt{G-A-T\,H-} \\ \mathtt{G\,C-D-H\,T} \end{pmatrix} \quad A_3 = \begin{pmatrix} \mathtt{G--A\,T\,H-} \\ \mathtt{G\,C\,D--H\,T} \end{pmatrix}$$
$$A_4 = \begin{pmatrix} \mathtt{G\,A--T\,H} \\ \mathtt{G\,C\,D\,H\,T-} \end{pmatrix} \quad A_5 = \begin{pmatrix} \mathtt{G\,A\,T\,H-} \\ \mathtt{G\,C\,D\,H\,T} \end{pmatrix}$$

The following table shows different key values of these alignments, where id is the number of indels and r is the number of replacements in the respective alignment.

	A_1	A_2	A_3	A_4	A_5
length	7	7	7	6	5
id	5	5	5	3	1
r	2	2	2	3	4
$id + 2 \cdot r$	9	9	9	9	9

As suggested by the previous example, the sum of the sequence length is directly related to the number of indels and replacements.

Lemma 2 Let A be an alignment of u and v. Let id be the number of insertions and deletions, and r be the number of replacements in A. Then $|u| + |v| = id + 2 \cdot r$.

Proof: Each of the $|u| + |v|$ characters in u and v must occur in A. Moreover, all characters in A are from the sequences u and v. Hence the number of characters in A and in u and v must be identical. In each indel one character occurs. In each replacement two of these characters occur. Hence the number of characters in A is $id + 2 \cdot r$. By the argumentation above, this number must be equal to $|u| + |v|$. So the equation holds. □

Here is another lemma characterizing the relation between alignment length and sequence lengths.

Lemma 3 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: 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 has maximal length among all alignments of u and v. Its length is $m + n$. Hence the length of A cannot be larger than $m + n$ and so $m + n \geq h$.

1. 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. That is, no alignment of u and v can be shorter than m. Hence $h \geq m = \max\{m, n\}$.

2. 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. That is, no alignment of u and v can be shorter than n. Hence $h \geq n = \max\{m, n\}$.

3.3.1 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.

Example 11 Here are all alignments of `ab` and `ca` (omitting the boundary brackets):

```
--ab   -a-b   a--b   a-b   -ab   -ab-   a-b-   ab-   ab--   ab-   -ab   a-b   ab
ca--   c-a-   -ca-   ca-   ca-   c--a   -c-a   c-a   --ca   -ca   c-a   -ca   ca
```

Hence $Aligns(2, 2) = 13$. Note that the number of alignments for two other sequences of the same lengths is also 13. This property holds in general, that is, the number of alignments only depends on the length of the sequences to be aligned, not on their content.

We will derive a recursive equation for $Aligns(m, n)$ by case distinction:

(1) Let $m = 0$ or $n = 0$. According to Lemma 3, an alignment of two sequences of length m and n has length h, s.t. $m + n \geq h \geq \max\{m, n\}$.

- If $m = n = 0$ we conclude $0 \geq h \geq 0$, i.e. $h = 0$. That is, an alignment of two empty sequences 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, all alignments of ε and v are of length n. They must consist of exactly n insertions. Of course, there is exactly one such alignment of ε and v.

Example 12 Let $v = \texttt{cd}$. The only alignment of ε and v is $\binom{\texttt{- -}}{\texttt{c d}}$

- If $m > 0$ and $n = 0$, then $m + n = m \geq h \geq m = \max\{m, n\}$, which implies $h = m$. That is, all alignments of u and ε are of length m. They must consist of exactly m deletions. Of course, there is exactly one such alignment.

Example 13 Let $u = \texttt{ab}$. The only alignment of u and ε is $\binom{\texttt{a b}}{\texttt{- -}}$

Thus in case (1) ($m = 0$ or $n = 0$) 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 of u and v 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 of u and v 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 of u and v 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 of u and v 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 of u and v 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.

So we have derived the following numbers of alignments in the 3 subcases:

- deletion case: $Aligns(m-1, n)$ alignments
- insertion case: $Aligns(m, n-1)$ alignments
- replacement case: $Aligns(m-1, n-1)$ alignments

As we exactly have three cases to obtain the total number we have to add them. So we conclude that the number $Aligns(m, n)$ of alignments of two sequences of length n and n , respectively, is

$$Aligns(m-1, n) + Aligns(m, n-1) + Aligns(m-1, n-1).$$

Altogether, we obtain the following recurrence:

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

Note that the notion recurrence refers to recursively defined equations.

Example 14

$$\begin{aligned} Aligns(2,2) &= Aligns(1,2) + Aligns(1,1) + Aligns(2,1) \\ &= (Aligns(0,2) + Aligns(0,1) + Aligns(1,1))+ \\ &\quad (Aligns(0,1) + Aligns(0,0) + Aligns(1,0))+ \\ &\quad (Aligns(1,1) + Aligns(1,0) + Aligns(2,0)) \\ &= (1 + 1 + Aligns(1,1))+ \\ &\quad (1 + 1 + 1)+ \\ &\quad (Aligns(1,1) + 1 + 1) \\ &= 7 + 2 \cdot Aligns(1,1) \\ &= 7 + 2 \cdot (Aligns(0,1) + Aligns(0,0) + Aligns(1,0)) \\ &= 7 + 2 \cdot (1 + 1 + 1) \\ &= 7 + 2 \cdot 3 \\ &= 13 \end{aligned}$$

According to [Laq81], $Aligns(n,n)$ can be approximated by the Stanton-Cowan-Numbers:

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

As $\sqrt{n} \geq 1$ and the expression $1 + \sqrt{2} \approx 2.4142$ is strictly greater than 1, one can state that the number of alignments grows exponentially with the length of the aligned sequences.

Example 15 For $n = 1\,000$ we have $Aligns(n,n) \approx \left(1+\sqrt{2}\right)^{2\,001} \cdot \sqrt{1\,000} = 10^{767.4}$.

3.3.2 The number of subsequences

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. So all alignments which differ only by the order of consecutive indels should be considered equivalent.

Example 16 The two alignments

$$\begin{pmatrix} \mathtt{a} & - \\ - & \mathtt{c} \end{pmatrix} \text{ and } \begin{pmatrix} - & \mathtt{a} \\ \mathtt{c} & - \end{pmatrix} \tag{3.1}$$

should be considered equivalent.

The three alignments

$$A_1 = \begin{pmatrix} \mathtt{G-ATT-H-} \\ \mathtt{GC--DHT} \end{pmatrix} A_2 = \begin{pmatrix} \mathtt{G-A-TH-} \\ \mathtt{GC-D-HT} \end{pmatrix} A_3 = \begin{pmatrix} \mathtt{G--ATH-} \\ \mathtt{GCD--HT} \end{pmatrix}$$

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

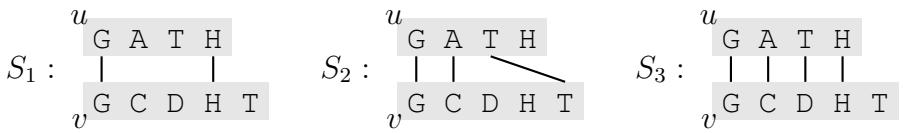
$$(i_1, j_1), \ldots, (i_r, j_r)$$

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 $u[i_h] \to v[j_h]$. All characters in u and v at positions not occurring in a given subsequence of u and v are considered to be deleted in u or v (without specifying the order in which the indels appear). So the empty subsequence stands for $\begin{pmatrix} a & - \\ - & c \end{pmatrix}$ and $\begin{pmatrix} - & a \\ c & - \end{pmatrix}$. In a graphical representation, the index pairs of the subsequence appear as lines connecting the characters in the subsequence.

Example 17 S_1=(1,1),(4,4), S_2=(1,1),(2,2),(3,5), S_3=(1,1),(2,2),(3,3),(4,4) are subsequences of $u = \mathtt{GATH}$ and $v = \mathtt{GCDHT}$, depicted as follows



They represent the alignments of Example 10. S_1 represents A_1, A_2, and A_3. S_2 represents A_4, and S_3 represents A_5.

Lemma 4 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 $i_1, \ldots, i_r$ at m positions there are $\binom{m}{r}$ possibilities; for the ordered selection of the

Table 4: Number of alignments and number of subsequences for two sequences of length n for n in the range from 0 to 10.

n	$Aligns(n,n)$	$Subseqs(n,n)$	$\frac{Aligns(n,n)}{Subseqs(n,n)}$
0	1	1	1.00
1	3	2	1.50
2	13	6	2.17
3	63	20	3.15
4	321	70	4.59
5	1 683	252	6.68
6	8 989	924	9.73
7	48 639	3 432	14.17
8	265 729	12 870	20.65
9	1 462 563	48 620	30.08
10	8 097 453	184 756	43.83

indices $j_1, \ldots, j_r$ at n positions 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).

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

$$Subseqs(1\,000, 1\,000) \approx 10^{600}$$

So we conclude that the number of alignments is much 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.

3.3.3 The edit distance problem

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

Definition 7 A *cost function* δ assigns non-negative costs to all edit operations such that $\delta(a \to b) \geq 0$ for all replacements $a \to b$ and $\delta(\alpha \to \beta) > 0$ for all indels $\alpha \to \beta$.

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 $\delta(\alpha \to \beta) = 0$ if and only if $\alpha = \beta$ for all edit operations $\alpha \to \beta$, then δ satisfies the *zero-property*. If $\delta(\alpha \to \beta) \leq \delta(\alpha \to \gamma) + \delta(\gamma \to \beta)$ for all edit operations $\alpha \to \beta$, $\alpha \to \gamma$, and $\gamma \to \beta$, then δ satisfies the *triangle inequality*.

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 18 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*.

Example 19 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*. If A is the only alignment of u and v without indels, then $\delta(A) = f_h(u, v)$. For an alignment A with indels, $\delta(A) = \infty$.

Example 20 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

representing

$\delta(a \to \varepsilon) = 3$ for $a \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$

$\delta(\varepsilon \to b) = 3$ for $b \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$

$\delta(a \to b) = 1$ for $(a, b) \in \{(\mathtt{A}, \mathtt{G}), (\mathtt{G}, \mathtt{A}), (\mathtt{T}, \mathtt{C}), (\mathtt{C}, \mathtt{T})\}$

$\delta(a \to b) = 2$ for $(a, b) \in \{(\mathtt{A}, \mathtt{T}), (\mathtt{T}, \mathtt{A}), (\mathtt{C}, \mathtt{G}), (\mathtt{G}, \mathtt{C})\}$

Then δ is the transversion/transition cost function. Bases A and G are called *purine*. 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 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.

Example 21 The following tables shows costs for all replacements of amino acids, as suggested by Willy Taylor. As it is symmetric only the lower triangle of the matrix is shown. Assuming positive indel costs, the cost function satisfies the zero-property.

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

Definition 8 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. □

Example 22 Let $u = ab$, $v = ca$ and δ be the unit cost function. Here are the alignments of u and v with the unit cost of each alignment shown below it:

--ab	-a-b	a--b	a-b	-ab	-ab-	a-b-	ab-	ab--	ab-	-ab	a-b	ab
ca--	c-a-	-ca-	ca-	ca-	c--a	-c-a	c-a	--ca	-ca	c-a	-ca	ca
4	4	4	3	2*	4	4	3	4	3	3	3	2*

As the smallest value is 2, we have $edist_\delta(u, v) = 2$. There are two alignments with cost 2, the optimal alignments (marked with $*$).

One important property of the edit distance is that it is a metric on $\mathcal{A}^*$.

Definition 9 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:

$f(x, y) = 0 \iff x = y$	zero property
$f(x, y) = f(y, x)$	symmetry
$f(x, y) \leq f(x, z) + f(z, y)$	triangle inequality

If the symmetry and triangle inequality holds for f, and also $x = y \Rightarrow f(x, y) = 0$, then f is a *pseudo-metric* on M.

If δ has the zero property, then so does $edist_\delta$. If δ is symmetric and satisfies the zero property as well as the triangle inequality, then $edist_\delta$ is a metric on $\mathcal{A}^*$, as shown in [WF74]. As seen in the example, there can be more than one optimal alignment. The unit edit distance is sometimes called Levenshtein distance. The following lemma states a simple property of the edit distance.

Lemma 5 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. Then $A^{-1} = (\alpha_h \to \beta_h, \ldots, \alpha_1 \to \beta_1)$ is an alignment of u^{-1} and v^{-1}. Suppose there is an alignment X of u^{-1} and v^{-1} s.t. $\delta(X) < \delta(A^{-1})$. That is, A^{-1} is not an 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, which is a contradiction. Hence our assumption above was wrong, i.e. there is no alignment X of u^{-1} and v^{-1} with $\delta(X) < \delta(A^{-1})$, so A^{-1} is optimal. Hence $edist_\delta(u, v) = \delta(A) = \delta(A^{-1}) = edist_\delta(u^{-1}, v^{-1})$.

Definition 10 The *edit distance problem* is to compute the edit distance and all optimal alignments. □

By specifying a concrete cost function, we obtain a special form of the edit distance:

Definition 11 If δ is the unit cost, then $edist_\delta$ is the *unit edit distance* or *Levenshtein distance*.

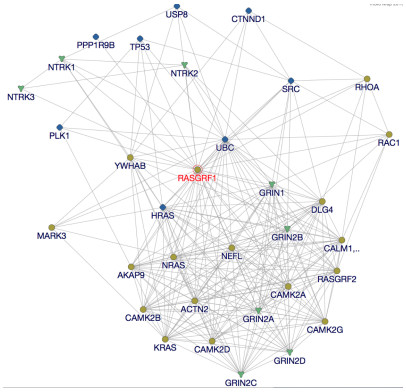
We will cast the edit distance problem as a graph theoretic problem and so we introduce some basic notions on graphs. A graph is a means of representing pairwise relations of any kind of items.

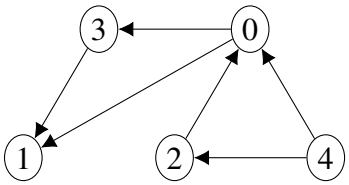
3.3.4 Basic definitions related to graphs

Definition 12 A graph consists of a set V of nodes (sometimes also called vertices) and a set $E \subseteq V \times V$ of edges. If for all $x, y \in V$, $(x, y) \in E$ implies $(y, x) \in E$, the graph is undirected.

Example 23 The graph (V, E) with $V = \{0, 1, 2, 3, 4\}$ and $E = \{(2, 0), (0, 1), (0, 3), (3, 1), (4, 0), (4, 2)\}$ is usually drawn as follows

Figure 6: An undirected graph created by the web server InBioMap (https://www.intomics.com) when searching the term RASGRF1. A protein is represented by a node. Different node symbols represent different subcellular locations. A protein-interaction is represented by an edge. An edge label (i.e. a confidence score for the corresponding interaction) becomes visible with a mouse-over event.





node labeled graph: nodes have labels (shown inside or besides node);
edge labeled graph: edges have labels (shown above or below edges);
graph is directed (the order of pairs in E matters); direction is expressed by arrows, i.e. $(x, y) \in E$ is written as $x \to y$

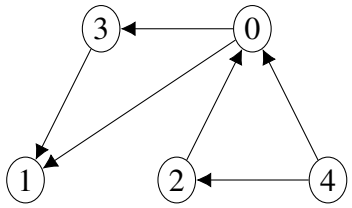
the placement of the nodes is arbitrary, i.e. it does not mean anything

An example of a graph representing biological data is given in Figure 6.

Definition 13 A *path* in a graph (V, E) is a sequence of nodes $v_0, v_1, \ldots, v_k$ for $k \geq 0$ such that for all i, $0 \leq i \leq k-1$ we have $(v_i, v_{i+1}) \in E$. An empty path satisfies $k = 0$, i.e. it has no edges and consists of a single node. A path starts with v_0 and ends with v_k and its length is k. So we state that it is a path from v_0 to v_k. Such a path is often written as

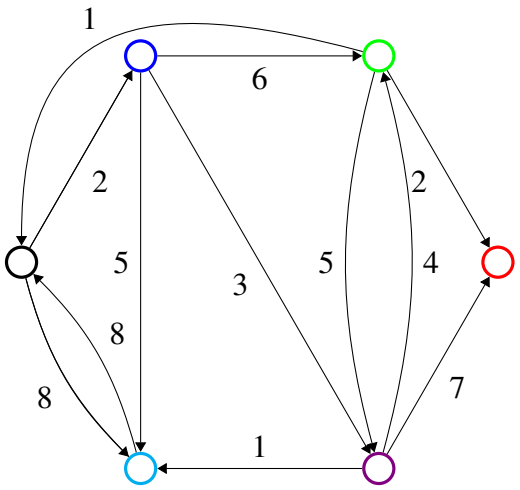
$$v_0 \to v_1 \to \cdots \to v_{k-1} \text{ (in directed graph)}$$
$$v_0 - v_1 - \cdots - v_{k-1} \text{ (in undirected graph)}$$

Example 24 Reconsider the directed graph (V, E) from Example 23.



$4 \to 2 \to 0 \to 1$
is a path of length 3 from 4 to 1
$4 \to 0 \to 3$
is a path of length 2 from 4 to 3
$3 \to 1$ is a path of length 1 from 3 to 1
$0 \to 1 \to 3$ is not a path, as $(1,3) \notin E$

Example 25 Here is a directed graph with unlabeled nodes. To better distinguish the nodes, they are shown in different colors. The edges are labeled and represent a weight function $w : E \to \mathbb{R}$.



such a graph can model:

nodes	edge labels
airports	flight times
social network users	$1 =$ has communicated
currencies	exchange rates
strings	costs of edit operations
genes	interaction level

- for an edge labeled graph it makes sense to define the weight of a path by adding up the weights of the edges it consists of

Definition 14 Consider a graph (V, E) with weight function $w : E \to \mathbb{R}$. For any path $p = v_0 \to v_1 \to \cdots \to v_k$, we define the weight $w(p)$ of p by

$$w(p) = \sum_{i=0}^{k-1} w(v_i \to v_{i+1})$$

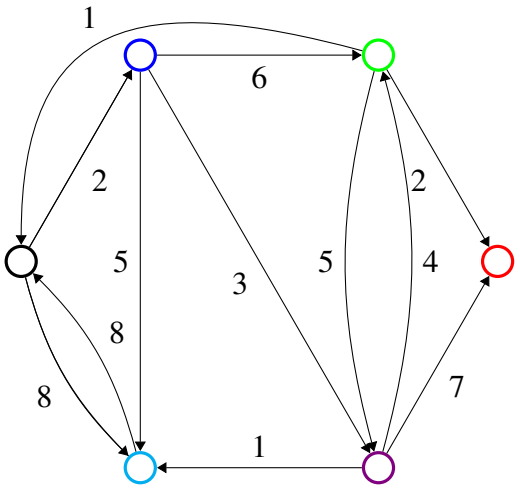
- a path of length 0 has weight 0
- when there is more than one path between two nodes, one is usually interested in the path of minimum weight (thus interpreting the weight as a distance)
- this leads to an important optimization problem:

Definition 15 Let $G = (V, E)$ be a graph with edge weights $w : E \to \mathbb{R}$. Let $s, x \in V$ and define

$$\Delta(s,x) = \begin{cases} \min\{w(p) \mid p \text{ is a path from } s \text{ to } x\} & \text{a path from } s \\ & \text{to } x \text{ exists} \\ \infty & \text{otherwise} \end{cases}$$

The *shortest-path problem* for s and x consists of computing $\Delta(s, x)$ and a path p from s to x such that $w(p) = \Delta(s, x)$. Such a path is called shortest or optimal path from s to x.

Example 26 Reconsider the graph from Example 25.

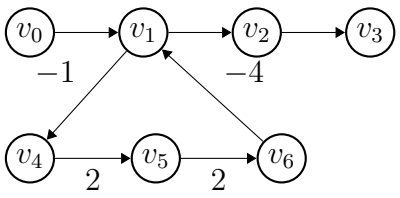


clockwise numbering of nodes from 0 to 5; start node s (0, black)

x	$\Delta(s,x)$	shortest path
0	0	0
1	2	$0 \to 1$
2	8	$0 \to 1 \to 2$
3	10	$0 \to 1 \to 2 \to 3$
4	5	$0 \to 1 \to 4$
5	6	$0 \to 1 \to 4 \to 5$

3.3.5 Existence of shortest paths

- suppose there is a path between s and u in graph G
- it may be surprising, that a shortest path does not always exist
- consider graph with a node x and non-empty path
  - from x to x (cycle),
  - with negative weight,
  - crossing the path from s to u
- then one can add additional cycles around x and decrease the weight
- any shortest path can be made shorter
- here is an example with $s = v_0$, $x = v_1$, and $u = v_3$



graph with negative weights appear e.g. in chemistry:

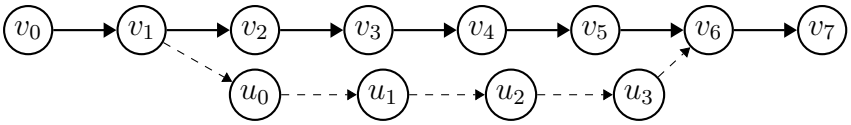
- compounds: nodes
- reaction: edge
- energy consumed ($\mathbb{R}_-$) or produced ($\mathbb{R}_+$): edge label

3.3.6 Non-negative weights and optimal subpaths

- in many cases, we can shift weights (by adding $-\min\{w(e) \mid e \in E\}$)
- so we restrict to the case of non-negative weights
- the key to many algorithms solving the shortest path problem is the fact that any subpath of a shortest path has minimum weight

Theorem 1 Consider a shortest path from s to u which includes a path from x to y. Then this path is a shortest path from x to y.

- we do not give a formal proof, but instead consider an example with a shortest path $p = v_0 \to v_1 \to v_2 \to \ldots \to v_5 \to v_6 \to v_7$:



- now look at the subpath $p' = v_1 \to v_2 \to \ldots \to v_5 \to v_6$
- assume that this is not the shortest path from v_1 to v_6 $(*)$
- so there exists a path p'' from v_1 to v_6, say along the dashed edges, such that $w(p'') < w(p')$
- but then we could replace the subpath p' in p by p'' and obtain a path from v_0 to v_7 with total weight smaller than $w(p)$
- as p is the shortest path, this is not possible

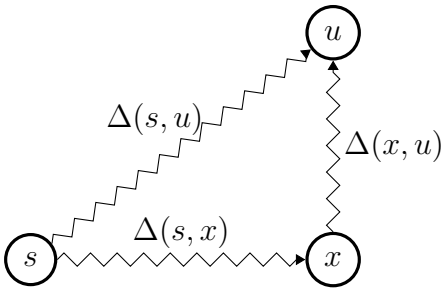
$\Rightarrow$ assumption $(*)$ was wrong, i.e. the subpath p' is a shortest path

3.3.7 Triangle inequality

- the *optimality of subpath*-property is based on a general property which holds for many mathematical structures: the triangle inequality

Theorem 2 For any nodes $s, u, x \in V$ we have $\Delta(s,u) \leq \Delta(s,x) + \Delta(x,u)$.

- again, we do not give a formal proof, but just consider the following picture, in which the zigzag-lines represent possibly long paths involving many nodes



- if we have a shortest path from s to u, any path from s to u involving some particular node x will not be shorter, even if the path from s to x and from x to u are both a shortest path
- so formally $\Delta(s,u) \leq \Delta(s,x) + \Delta(x,u)$

3.3.8 Single source shortest path problem

- recall that the shortest path problem we had considered involves two particular node s and u
- so lets denote it by $PathP(s, u)$, where $PathP$ stands for path problem
- a generalization of this is the single source shortest path problem:

  for a given node s (the source node) determine $\Delta(s, x)$ (and shortest path from s to x) for all $x \in V$
- let us denote this problem by $PathP(s, V)$
- with the most efficient algorithms known today, it is (in general) not easier to solve $PathP(s, u)$ for a particular $u \in V$ than to solve $PathP(s, V)$
- so consider how to solve $PathP(s, V)$
- if we have a solution for this problem, we can lookup the solution for a particular target node u

3.3.9 Directed acyclic graphs

Definition 16 A directed acyclic graph $G = (V, E)$ (often abbreviated as DAG) is a finite graph without non-empty cycles, i.e. for any node $s \in V$ there is no non-empty path from s to s.

A DAG can be used to model different kinds of data:

- dependencies of software consisting of many modules
- version history in version control systems
- logic blocks in electronic circuits design
- dependencies of events in Bayesian networks
- historical dependencies in family trees
- dependencies in a system of equations (recurrences)

We are in particular interested in

1. solving the shortest path problem for DAGs and
2. applying the solution to solve the edit distance problem

Let us first consider how to solve the first problem.

- we have seen that a shortest path from node s to x consists of subpaths each of which is a shortest path for the nodes it connects
- this of course holds for DAGs as a special form of graphs
- suppose $s \neq x$

- then the shortest path from s to x must contain a least one edge
- so consider the last edge of the shortest path, i.e. an edge $y \to x \in E$ for some node y
- by the subpath-optimality principle the subpath of our shortest path from s to x which connects s with y must be a shortest path, too.
- so we can conclude $\Delta(s,x) = \Delta(s,y) + w(y,x)$
- of course, we do not know in advance which incoming edge to x will be the last edge of a shortest path, but there will be at least one such edge
- so we just minimize over all incoming edges, i.e. we can compute

$$\Delta(s,x) = \min\{\Delta(s,y) + w(y,x) \mid y \to x \in E\}$$

- this is the central equation allowing to solve the shortest path problem
- to apply it we have to solve one subproblem:
- we can only evaluate

$$\Delta(s,x) = \min\{\Delta(s,y) + w(y,x) \mid y \to x \in E\}$$

  if we already know $\Delta(s,y)$
- so we have to compute the values $\Delta(s,\_)$ in an order which respects the edges, i.e. if there is an edge $y \to x \in E$ we have to compute $\Delta(s,y)$ before $\Delta(s,x)$
- this is an order of the nodes in the graph called *topological order*

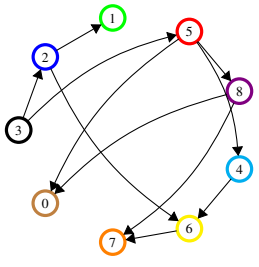
Definition 17 For a given directed graph $G = (V,E)$ a topological order $\prec$ of V satisfies:

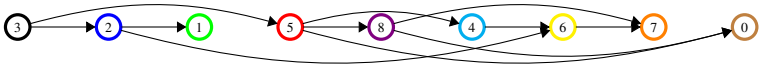
$$y \to x \in E \Rightarrow y \prec x$$

- note: there is at least one topological order for a DAG and usually more than one possible such orders

Example 27

- consider the graph $G = (V,E)$ with a layout of the nodes on a circle:
- here is a different layout of the graph in a linear form:





- as the edges only go from left to right, the layout implies the topological order

$$3 \prec 2 \prec 1 \prec 5 \prec 8 \prec 4 \prec 6 \prec 7 \prec 0$$

Conclusion: solving shortest path problem on DAG $G = (V, E)$

- compute a topological order $\prec$ on G (runtime: $O(|V| + |E|)$)
- for given start node s eliminate all nodes $x \in V$, s.t. $x \prec s$ and all nodes outgoing from x
- for all nodes $x \in V$, $s \prec x$ in topological order, compute

$$\Delta(s, x) = \min\{\Delta(s, y) + w(y, x) \mid y \to x \in E\}$$

  and store it in a table for looking it up later
- as $y \to x \in E$ implies $y \prec x$, $\Delta(s, y)$ is available (and can be looked up in constant time) when $\Delta(s, x)$ is evaluated
- overall runtime: $O(|V| + |E|)$
- to solve the edit distance problem using this method we introduce the notion of edit graphs, as a special form of graphs

3.3.10 Edit graphs

Definition 18 The *edit graph* $G(u, v)$ of u and v is an edge labeled directed graph. The nodes are the pairs (i, j), $0 \leq i \leq m$, $0 \leq j \leq n$. The edges are given as follows:

- For $1 \leq i \leq m, 0 \leq j \leq n$ there is a deletion edge

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

- For $0 \leq i \leq m, 1 \leq j \leq n$ there is an insertion edge

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

- For $1 \leq i \leq m, 1 \leq j \leq n$ there is a replacement edge

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

□

This is illustrated in Figure 7.

An edit graph for two sequences u and v, both of length 3 is shown in Figure 8.

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.

Figure 7: A part of the edit graph $G(u, v)$

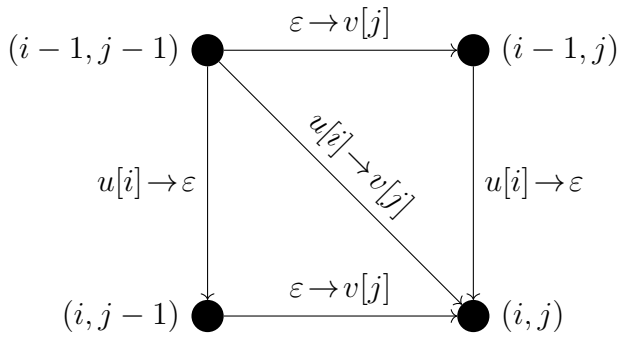
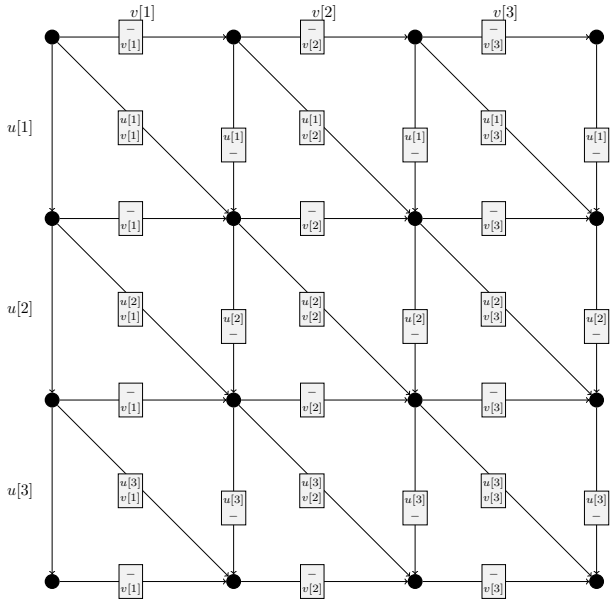


Figure 8: An edit graph $G(u, v)$ where $|u| = |v| = 3$

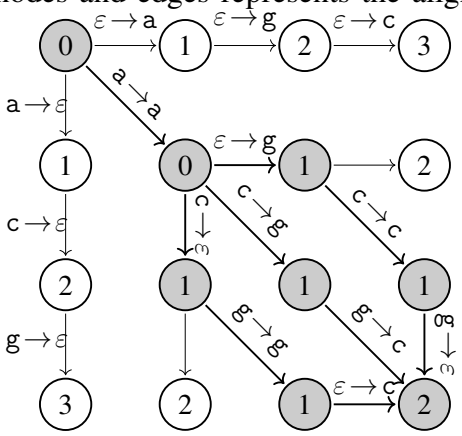


So there is a one-to-one correspondence of paths and alignments. When applying the cost function δ to the edges of $G(u, v)$ one obtains weights, i.e. $w((i', j') \xrightarrow{\alpha \mapsto \beta} (i, j)) = \delta(\alpha \to \beta)$. So each path represents an alignment and the weight of the path is the cost of the alignment. Thus, a shortest path from (i', j') to (i, j) is labeled by an optimal alignment of $u[i' + 1 \ldots i]$ and $v[j' + 1 \ldots j]$. In particular, a shortest path from $(0, 0)$ to (i, j) is labeled by an optimal alignment of $u[1 \ldots i]$ and $v[1 \ldots j]$.

Example 28 Edit graph for $(u, v) = (\mathtt{acg}, \mathtt{agc})$ restricted to the edges on shortest paths from $(0, 0)$. The paths from $(1, 1)$ to $(3, 3)$ of thick nodes and edges represents the alignments

$(\varepsilon \to \mathtt{g}, \mathtt{c} \to \mathtt{c}, \mathtt{g} \to \varepsilon)$, $(\mathtt{c} \to \mathtt{g}, \mathtt{g} \to \mathtt{c})$, $(\mathtt{c} \to \varepsilon, \mathtt{g} \to \mathtt{g}, \varepsilon \to \mathtt{c})$, of $u[2 \dots 3] = \mathtt{cg}$ and $v[2 \dots 3] = \mathtt{gc}$.

- node (i, j) labeled by the cost of an optimal alignment of $u[1 \dots i]$ and $v[1 \dots j]$, i.e. the edit distance of these sequences



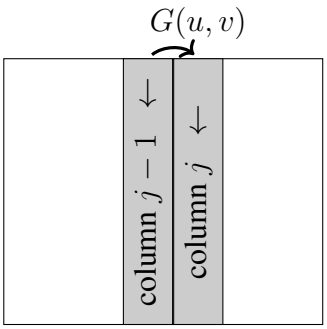
From the above, we deduce that a shortest path from $(0, 0)$ to (m, n) is labeled by an optimal alignment of $u = u[1 \dots m]$ and $v = v[1 \dots n]$. So computing optimal alignments means to enumerate shortest paths from $(0, 0)$ to (m, n) in $G(u, v)$.

Conclusion

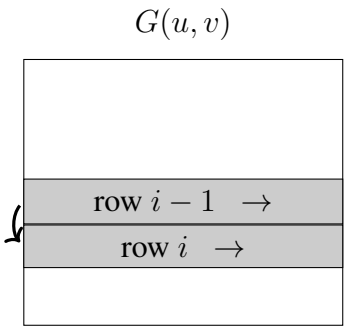
- We have formulated the edit distance problem for u and v as a shortest path problem for $G(u, v)$.
- we are in particular interested in the shortest path from $(0, 0)$ to (m, n).

As $G(u, v)$ only contains edges of the form $(i', j') \xrightarrow{\alpha \to \beta} (i, j)$ such that $i' < i$ or $j' < j$, there are no cycles. So $G(u, v)$ is a directed acyclic graph, a DAG. We already know how to solve shortest path problems on DAGs: We first have to determine a topological order of the nodes in $G(u, v)$. There are basically two main topological orders in $G(u, v)$:

columnwise topological order:
$(i', j') \prec (i, j) \iff$
$j' < j$ or $(j' = j$ and $i' < i)$



rowwise topological order:
$(i', j') \prec (i, j) \iff$
$i' < i$ or $(i' = i$ and $j' < j)$



- in both cases, we have $(i, j - 1) \prec (i, j)$, $(i - 1, j - 1) \prec (i, j)$, and $(i - 1, j) \prec (i, j)$.

3.3.11 From edit graphs to edit matrices

- recall the general recurrence from page 35: $\Delta(s,x) = \min\{\Delta(s,y) + w(y,x) \mid y \to x \in E\}$
- in our case the nodes are pairs (i,j) and the edges are labeled by edit operations, to which we apply δ to obtain weights

Conclusion: solving the edit distance problem

- compute $G(u,v)$
- for all nodes $(i,j) \in G(u,v)$ in topological order, compute

$$\Delta((0,0),(i,j)) = \min\{\Delta((0,0),(i',j')) + \delta(\alpha \to \beta) \mid (i',j') \xrightarrow{\alpha \to \beta} (i,j) \text{ is edge in } G(u,v)\}$$

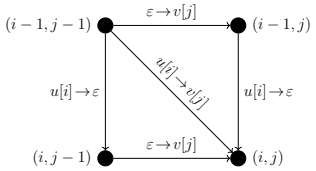
- to simplify the notation, we introduce the abbreviation

$$E_\delta(i,j) = \Delta((0,0),(i,j))$$

- and so the equation above simplifies to

$$E_\delta(i,j) = \min\{E_\delta(i',j') + \delta(\alpha \to \beta) \mid (i',j') \xrightarrow{\alpha \to \beta} (i,j) \text{ is edge in } G(u,v)\}$$

- recall the simple structure of the edit graph
- if $i > 0$ and $j > 0$, node (i,j) has three incoming edges, as already illustrated previously:



- so we can rewrite

$$E_\delta(i,j) = \min\{E_\delta(i',j') + \delta(\alpha \to \beta) \mid (i',j') \xrightarrow{\alpha \to \beta} (i,j) \text{ is edge in } G(u,v)\}$$

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

- for $i > 0$ and $j = 0$, node (i, j) only has one incoming edge $(i-1, j) \xrightarrow{u[i] \to \varepsilon} (i, j)$ and so we simplify

$$E_\delta(i,j) = \min\{E_\delta(i',j') + \delta(\alpha \to \beta) \mid (i',j') \xrightarrow{\alpha \to \beta} (i,j) \text{ is edge in } G(u,v)\}$$

$$\text{as } E_\delta(i,j) = E_\delta(i-1,j) + \delta(u[i] \to \varepsilon)$$

- for $i = 0$ and $j > 0$, node (i, j) only has one incoming edge $(i, j-1) \xrightarrow{\varepsilon \to v[j]} (i, j)$ and so

$$E_\delta(i,j) = E_\delta(i,j-1) + \delta(\varepsilon \to v[j])$$

- for $i = 0$ and $j = 0$, node (i, j) has no incoming edge and $E_\delta(i, j)$ is the weight of the shortest path from $(0, 0)$ to $(0, 0)$ which is 0. So $E_\delta(i, j) = 0$.

3.3.12 Solving the edit distance problem

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

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

Recall that $E_\delta(i, j)$ is

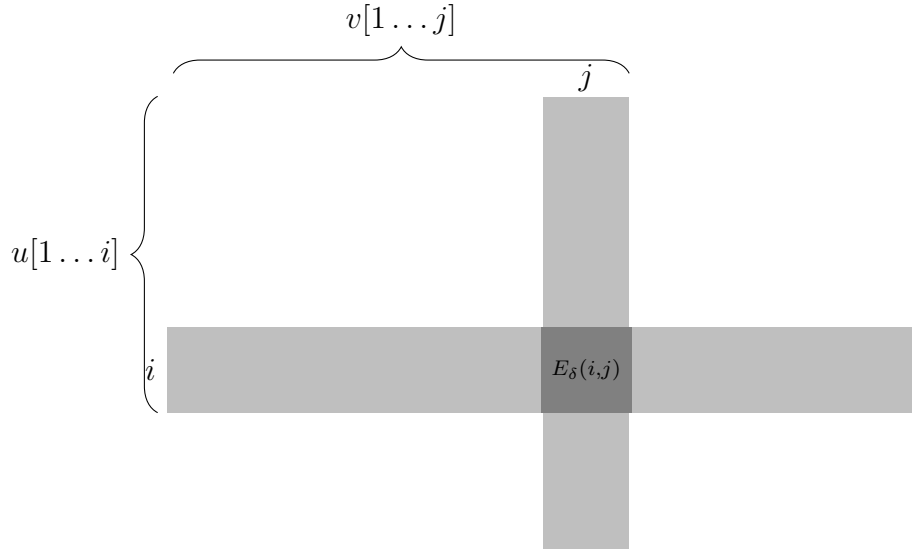
- the weight of the shortest path from $(0, 0)$ to (i, j) in $G(u, v)$ and equivalently
- also the cost of an optimal alignment of $u[1 \ldots i]$ and $v[1 \ldots j]$.

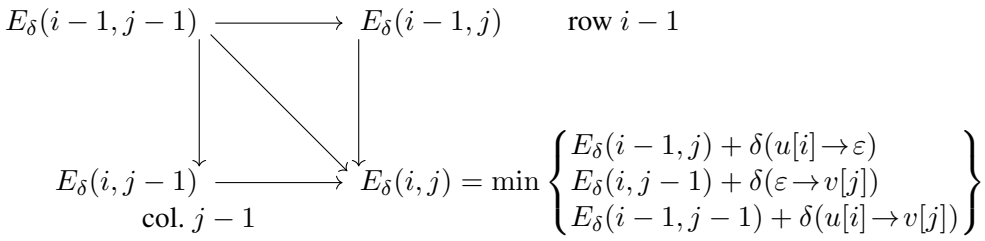
In the following we neglect $G(u, v)$ for a while and view E_δ as a matrix with $m + 1$ rows (indexed from 0 to m) and $n + 1$ columns (indexed from 0 to n) of numeric values. Such a matrix is called $(m + 1) \times (n + 1)$-matrix.

The index i and j are not only row and column numbers of the matrix, but also refer to prefixes of u and v, respectively, see Figure 9.

By definition, $E_\delta(m, n) = edist_\delta(u[1 \ldots m], v[1 \ldots n]) = edist_\delta(u, v)$, i.e. it gives the edit distance of u and v. According to the recurrence above, $E_\delta(i, j)$ depends on the previous value $E_\delta(i-1, j)$ in the current column j and on two values $E_\delta(i-1, j-1)$ and $E_\delta(i, j-1)$ in the previous column. See the following illustration, in which a directed edge $E_\delta(i', j') \to E_\delta(i, j)$ means that $E_\delta(i, j)$ depends on $E_\delta(i', j')$:

Figure 9: The curly brackets mark the substrings of u and v an entry in E_δ refers to. In particular, $E_\delta(i, j)$ refers to the pair of prefixes $u[1 \ldots i]$ and $v[1 \ldots j]$.





The values in E_δ need to be computed in topological order of the nodes in the underlying graph $G(u, v)$. We compute E_δ columnwise and each column from top to bottom.

Algorithm 1 (Computation of the edit distance)

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

1: $E_\delta(0,0) \leftarrow 0$
2: **for** $i \leftarrow 1$ **upto** m **do**
3: $\quad E_\delta(i,0) \leftarrow E_\delta(i-1,0) + \delta(u[i] \to \varepsilon)$
4: **end for**
5: **for** $j \leftarrow 1$ **upto** n **do**
6: $\quad E_\delta(0,j) \leftarrow E_\delta(0,j-1) + \delta(\varepsilon \to v[j])$
7: $\quad$ **for** $i \leftarrow 1$ **upto** m **do**
8: $$\quad\quad E_\delta(i,j) \leftarrow \min \left\{ \begin{array}{l} E_\delta(i-1,j) + \delta(u[i] \to \varepsilon) \\ E_\delta(i,j-1) + \delta(\varepsilon \to v[j]) \\ E_\delta(i-1,j-1) + \delta(u[i] \to v[j]) \end{array} \right\}$$
9: $\quad$ **end for**
10: **end for**
11: **print** $E_\delta(m,n)$

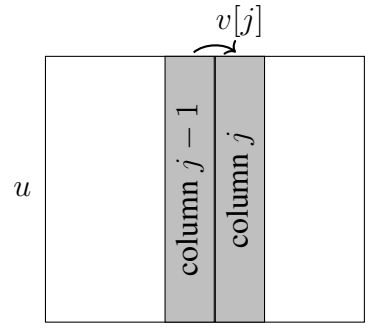
Example 29 Let $u =$ `bcacd`, $v =$ `dbadad`, and assume that δ is the unit cost function. Then E_δ is as follows, with the index range for i and j shown in the second column and second row.

	i		d	b	a	d	a	d
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	$\boxed{4}$

$$\boxed{4} = \min \left\{ \begin{array}{l} 4+1 \\ 4+1 \\ \mathbf{4}+0 \end{array} \right\}$$

Hence the edit distance of u and v is $E_\delta(5,6) = 4$. □

Each entry in E_δ is computed in constant time. As there are $(m+1) \cdot (n+1)$ entries, the running time is $O(mn)$. If we only want to compute the edit distance, we only need the last column and its last value. Obviously, the values in each column only depend on the values of the current and the previous column, see the following illustration.



Hence, it suffices to store only two columns, and thus the so called "distance-only algorithm" requires only $O(m)$ space. It is even 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).

In molecular biology, the above algorithm is often 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:

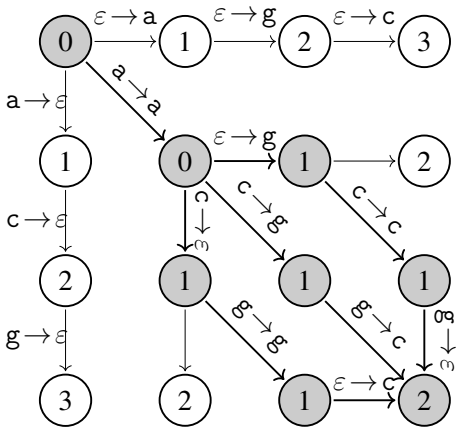
- optimal solutions to the problem can be derived from optimal solutions to subproblems.
- the optimal solutions can efficiently be determined, if a table of solutions for subproblems of increasing sizes are computed.

For the edit distance problem we derive the optimal solution $edist_\delta(u, v)$ from the optimal solution of subproblems, i.e. the edit distance of each pair of prefixes of u and v, respectively. And the optimal solutions to the subproblems are tabulated in E_δ, such that $E_\delta(i, j) = edist_\delta(u[1 \ldots i], v[1 \ldots j])$.

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 $O(mn)$ space. This traceback algorithm can best be explained by referring to the edit graph and the notion of minimizing edges.

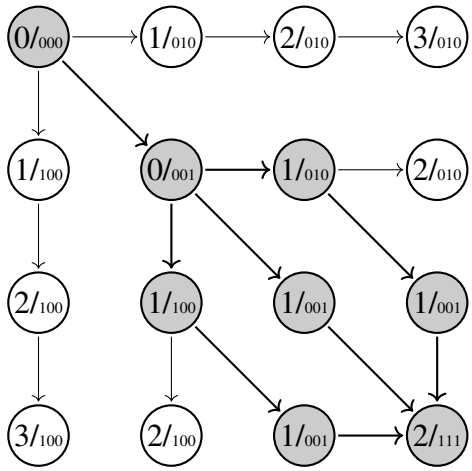
Definition 19 An edge $(i', j') \xrightarrow{\alpha \to \beta} (i, j)$ in $G(u, v)$ is *minimizing* if $E_\delta(i, j)$ equals $E_\delta(i', j') + \delta(\alpha \to \beta)$.

Example 30 The edit graph for $u = acg$ and $v = agc$ restricted to minimizing edges. The thick edges are those which are on the path from $(0, 0)$ to $(3, 3)$. Node (i, j) is labeled by $E_\delta(i, j)$.



The traceback starts at node (m, n) and traces the minimizing edges back to node $(0, 0)$. The traceback method 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 traceback (and implicitly storing the minimizing edges of the graph), we maintain with each entry $E_\delta(i, j)$ three bits, one for each type of edit operation. Each of these bits tells us whether the incoming edge into (i, j) is minimizing or not.

Example 31 Here we show the previous edit graph, in which each node (i, j) is labeled by $E_\delta(i, j)$ and the three bits representing the incoming minimizing edges.[1]



Thus we conclude:

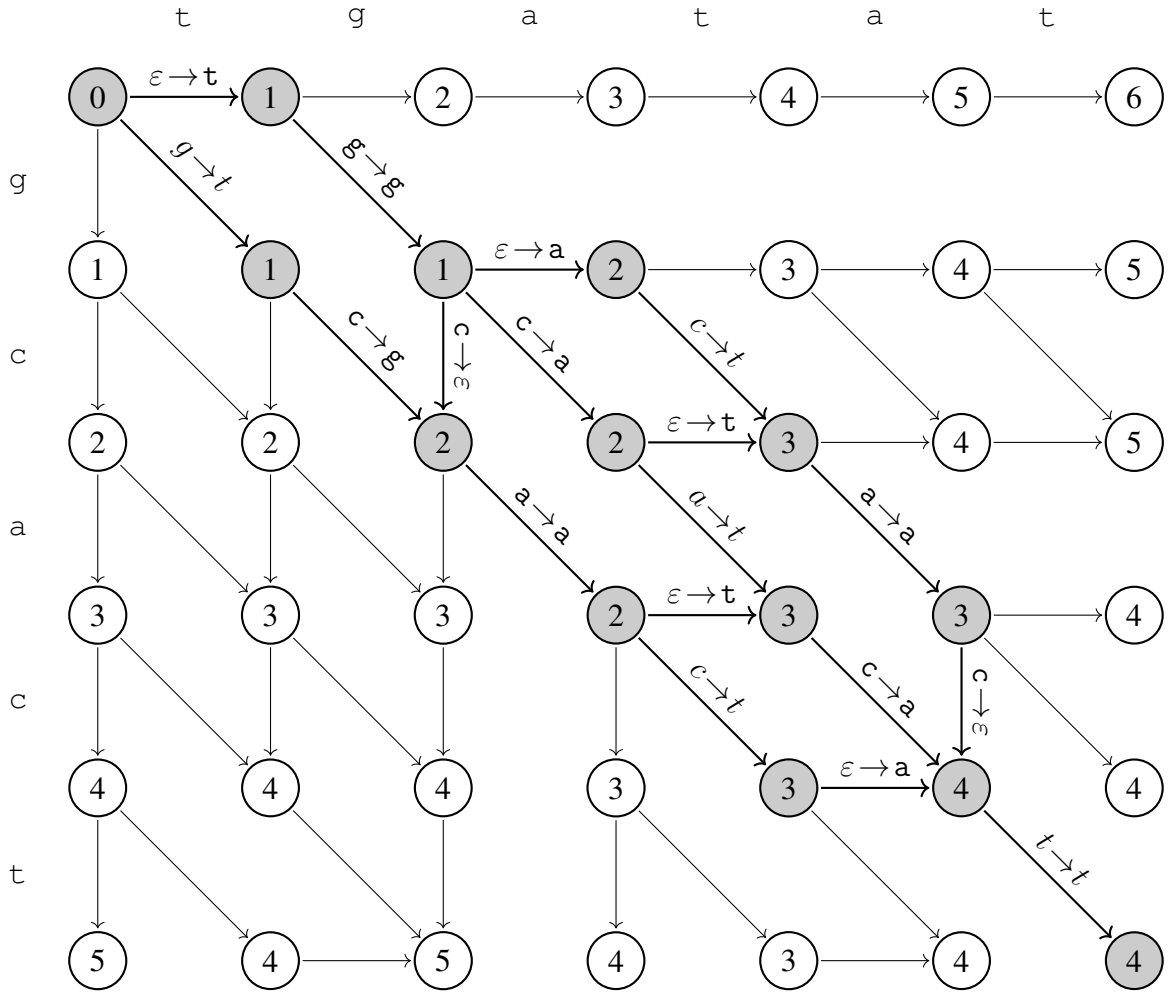
Theorem 3 The edit distance problem for two sequences u of length m and v of length n can be solved in $O(mn + z)$ time, where z is the total length of all alignments of u and v. □

Example 32 Let $u = \texttt{gcact}$ and $v = \texttt{tgatat}$. 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.

$$\begin{pmatrix} \texttt{g c a c - t} \\ \texttt{t g a t a t} \end{pmatrix} \begin{pmatrix} \texttt{- g c a c - t} \\ \texttt{t g - a t a t} \end{pmatrix} \begin{pmatrix} \texttt{g c a - c t} \\ \texttt{t g a t a t} \end{pmatrix} \begin{pmatrix} \texttt{- g c a - c t} \\ \texttt{t g - a t a t} \end{pmatrix}$$

[1] bits = $d\,i\,r$, d: deletion, i: insertion, r: replacement.

Figure 10: All nodes (marked with the corresponding distance values) and all minimizing edges in the edit graph $G(u, v)$ for $u = \texttt{gcact}$ and $v = \texttt{tgatat}$. For better readability, these sequences appear on the left and above the graph, respectively. The thick edges (marked with their edge labels) form the minimizing paths and all nodes on the minimizing paths are shown in gray.



$$\begin{pmatrix} - & \texttt{g} & \texttt{c} & \texttt{a} & \texttt{c} & \texttt{t} \\ \texttt{t} & \texttt{g} & \texttt{a} & \texttt{t} & \texttt{a} & \texttt{t} \end{pmatrix} \quad \begin{pmatrix} - & \texttt{g} & \texttt{c} & - & \texttt{a} & \texttt{c} & \texttt{t} \\ \texttt{t} & \texttt{g} & \texttt{a} & \texttt{t} & \texttt{a} & - & \texttt{t} \end{pmatrix} \quad \begin{pmatrix} - & \texttt{g} & - & \texttt{c} & \texttt{a} & \texttt{c} & \texttt{t} \\ \texttt{t} & \texttt{g} & \texttt{a} & \texttt{t} & \texttt{a} & - & \texttt{t} \end{pmatrix}$$

Figure 10 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 method is $O(mn)$. Using a distance-only algorithm in a recursive strategy, one obtains a divide and conquer algorithm that can determine each optimal

alignment in $O(\min\{m, n\})$ space and $O(mn)$ time (see Section 5.1). These algorithms are very important, since space, not time, is often the limiting factor when computing optimal alignments between large sequences.

3.4 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 DNA 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 the Example 33. In an analogy, protein sequences often contain several domains which they share with related sequences. But if only the domains are similar, then a global comparison would not reveal the similarity of the domains.

Example 33 The following global alignment shows 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.

```
CGGGGAGGCGCCCGCGCGGCCCAGTTAT-TCAGAATTAAAATCTTAGTAAACAT
                         ||||| | ||||
AATTAAAATCTTAGTAAACATCAGTT-TGTCAGCGGGGAGGCGCCCGCGCGGCC
```

Thus a global alignment does not make sense in this case and one should look for local similarities, displayed in a local alignment.

As a consequence, when comparing biological sequences it is sometimes important to perform local similarity comparisons: This means to find pairs of similar *substrings* of u and v. It does not make sense to look for pairs of substrings with some minimum distance, as ε is a substring of any sequence and the sequences ε and ε have distance 0. To clarify the notion of sequence similarity, we introduce score functions.

Definition 20 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 the sequences u' and v' is defined by

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

Table 5 shows the BLOSUM62 similarity matrix, which is widely used when comparing proteins.

Table 5: The BLOSUM62 similarity score matrix specifying a replacement score for each pair of amino acid. As the matrix is symmetric only the lower halve of the matrix is shown. With some additional scores for insertions and deletions we would obtain a score function.

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

Definition 21 Let σ be a score function. We define

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

   That is, $loc_\sigma(u, v)$ is the maximum score over all pairs of substrings of u and v.

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

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

   An alignment A of u' and v' satisfying $score_\sigma(u', v') = \sigma(A)$ is an *optimal local alignment* of u and v.

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

Example 34 Consider the sequences $u = \texttt{PQRAFADCSTVQ}$ and $v = \texttt{FYAFDACSLL}$ and the following similarity score function:

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

Then $loc_\sigma(u, v) = 8 = score_\sigma(u', v')$, where $u' = \texttt{AFADCS}$, $v' = \texttt{AFDACS}$ and

```
A F  -  A  D  C S
A F  D  A  -  C S
2 2 -1  2 -1  2 2
```

of score 8 is an optimal local alignment (with the score of each column in the last line).

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

> for each pair (u', v') of substrings u' of u and v' of v compute $score_\sigma(u', v')$ and determine the maximum over all such values

Computing $score_\sigma(u', v')$ is a global alignment problem where, instead of minimizing distances, similarities are maximized. Therefore, we can modify the DP Algorithm for computing the edit distance in such a way that a score function is used and maxima are computed instead of minima. Thus $score_\sigma(u', v')$ can be computed in $O(|u'||v'|) = O(mn)$ time. There are $O(m^2n^2)$ pairs (u', v') of substrings of u and v. Thus, this method would require $O(m^2n^2mn) = O(m^3n^3)$ time, which is, of course, too much.

Note that a substring of a string is a suffix of a prefix of that string:

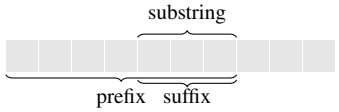
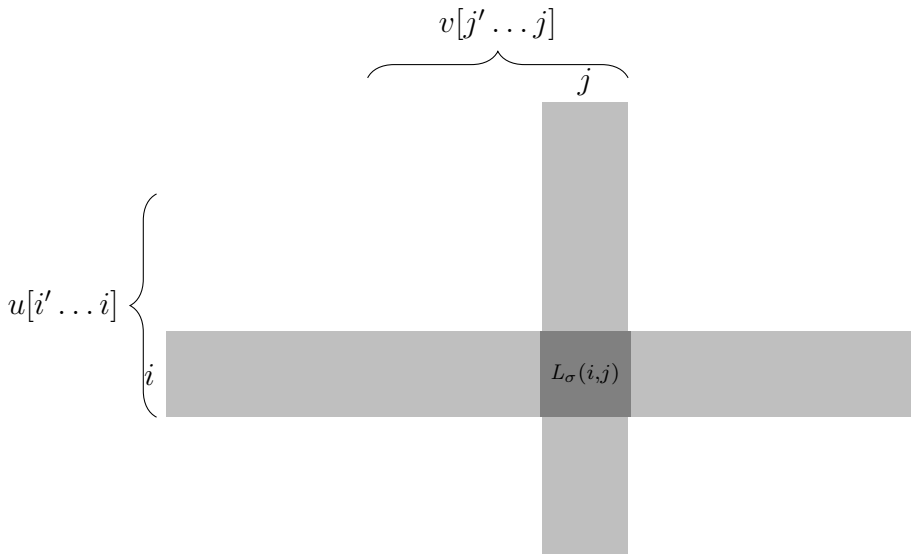


Figure 11: The curly brackets mark the substrings of u and v an entry in L_σ refers to. In particular, $L_\sigma(i,j)$ refers to pairs of suffixes $u[i' \ldots i]$ of $u[1 \ldots i]$ and suffixes $v[j' \ldots j]$ of $v[1 \ldots j]$.



So substring u' of u is a suffix of a prefix of u and substring v' of v is a suffix of a prefix of v. For example, if $u' = u[p \ldots q]$ (i.e. u' is a substring of u), then $u[1 \ldots q]$ is a prefix of u and u' is a suffix of this prefix. The idea is to compute a matrix where each entry (i,j) contains the maximal score for all pairs of suffixes of prefix $u[1 \ldots i]$ of u and prefix $v[1 \ldots j]$ of v. That is, we compute an $(m+1) \times (n+1)$-matrix L_σ defined by

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

See also Figure 11 for an illustration.

Now we can conclude,

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

So $loc_\sigma(u, v)$ can be computed by maximizing over all entries in 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 4 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: it means that 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):

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

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

Based on Theorem 4, we can derive Algorithm 2 which solves the local similarity search problem. The algorithm was first published in [SW81].

Algorithm 2 requires $O(mn)$ time and space.

Example 35 Consider the similarity score

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

and sequences $u =$ `FYAFDACSLL` and $v =$ `PQRAFADCSTVQ`. Then matrix L_σ is as follows:

Algorithm 2 (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 an optimal local alignment of u and v.

1. Compute matrix L_σ according to Theorem 4.
2. Compute a maximal entry, say $L_\sigma(i, j)$, in L_σ.
3. Compute optimal local alignments by a traceback from (i, j) on a maximizing path until some entry (i', j') satisfying $L_\sigma(i', j') = 0$ is found.

L_σ		F	Y	A	F	D	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
F	0	2	1	1	**4**	**3**	2	1	0	0	0
A	0	1	0	3	3	2	**5**	4	3	2	1
D	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

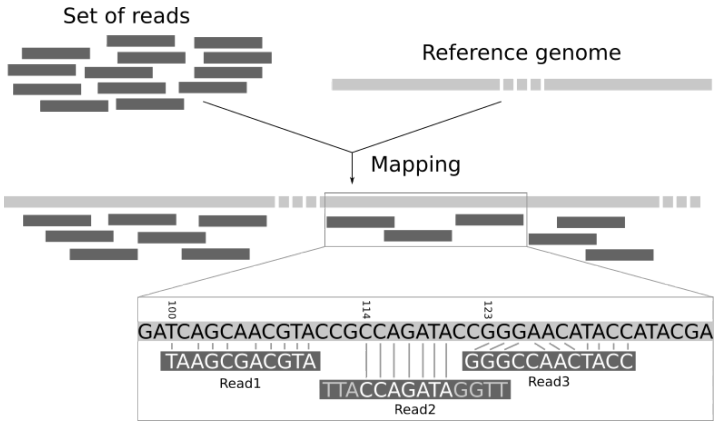
optimal local alignment:

A F – A D C S
A F D A – C S

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

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, implicitly all pairs of substrings of both sequences are aligned to obtain an optimal local alignment. The common technical concept underlying both methods, i.e. the computation of $(m + 1) \times (n + 1)$-matrices can adapted to other problems. Here we consider the approximate string matching problem as one such problem.

Figure 12: Schematic explanation of the readmapping problem as a special instance of the approximate string matching problem. Each of 10^5-10^8 reads, i.e. a DNA sequence of $10^2 - 10^5$bp. must be matched against a reference genome, allowing indels and mismatches. The assignment of the reads to the positions where they match is called read mapping.



3.5 The approximate string matching 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 matching problem* (APSMP, for short) consists of finding the positions in t where an approximate match ends. These positions are referred to as solutions of the approximate string matching problem. Let $k \in \mathbb{R}^+$ be a threshold value. An *approximate match* is a substring w of t such that $edist_\delta(p, w) \leq k$.

The approximate string matching problem is of special interest in biological sequence analysis. For instance, when searching a DNA database (the input string) for a pattern (e.g. a sequence read), few errors must be allowed, to take into account experimental inaccuracies as well as small differences in DNA among individuals of the same or related species. As the number of patterns has grown very fast, due to establishing novel sequencing technologies, the APSMP is one of the most well-studied problems (often named the read-mapping problem) in algorithmic bioinformatics.

https://training.galaxyproject.org/training-material/topics/sequence-analysis/ tutorials/mapping/tutorial.html

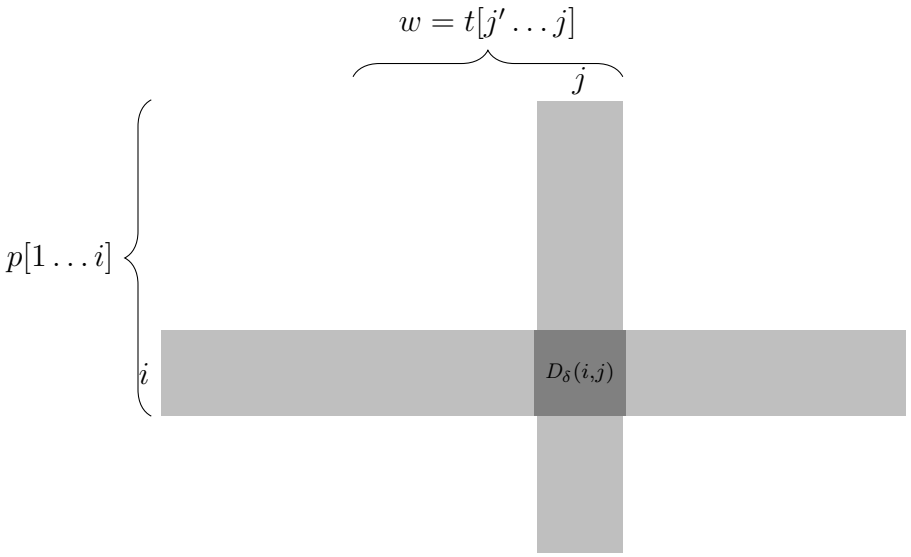
See [FRBM12] for a recent review of the available methods.

By a slight modification of the previous dynamic programming algorithms, one obtains a simple method (first published in [Sel80]) to solve the approximate string matching problem.

Algorithm Compute an $(m + 1) \times (n + 1)$-table D_δ such that for all (i, j), $0 \leq i \leq m$, $0 \leq j \leq n$, the following holds:

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

Figure 13: The curly brackets mark the substrings of p and t an entry $D_\delta(i,j)$ refers to.



Thus entry $D_\delta(i,j)$ states a property of the prefix of $p[1 \ldots i]$ of p (just like $E_\delta(i,j)$ states a property of the prefix $u[1 \ldots i]$ of u). Moreover, $D_\delta(i,j)$ states a property of the suffixes of $t[1 \ldots j]$ (just like $L_\sigma(i,j)$ states a property of the suffixes of $v[1 \ldots j]$). So table D_δ is a hybrid between E_δ and L_σ mixing global and local sequence alignment, see Figure 13 for an illustration. One often states that the APSMP is solved by semi-global alignment.

To understand the relation of D_δ with the APSMP recall the definition of the notion *approximate match*:

> there is an approximate match ending at position j, $0 \leq j \leq n$, 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

$$\underbrace{\min\{edist_\delta(p[1 \ldots m], w) \mid w \text{ is a suffix of } t[1 \ldots j]\}}_{D_\delta(m,j)} \leq k$$

which is equivalent to

$$D_\delta(m,j) \leq k$$

Thus, to solve the approximate string matching problem, one computes table D_δ and outputs all j satisfying $D_\delta(m,j) \leq k$. See Figure 14, for an example.

The approximate matches w ending at position j (and an optimal alignment of p and w, if necessary) can be output, by a traceback from $D_\delta(m,j)$ to an entry $D_\delta(0,j')$, $0 \leq j' \leq j-1$ in the first row of table D_δ.

Figure 14: Table D_δ for $p = \texttt{atggc}$, $t = \texttt{aggtatcgc}$ and the unit cost function δ. Let $k = 2$. Then in the last row we find the boxed values $\leq k$ for $j \in \{3, 4, 7, 8, 9\}$. Hence approximate matches end at these positions.

							j				
				a	g	g	t	a	t	c	g	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	t	2	2	1	1	2	1	1	0	1	2	2
	g	3	3	2	1	1	2	2	1	1	1	2
	g	4	4	3	2	1	2	3	2	2	1	2
	c	5	5	4	3	$\boxed{2}$	$\boxed{2}$	3	3	$\boxed{2}$	$\boxed{2}$	$\boxed{1}$

Recurrences for D_δ are easy to derive. For $i = 0$ we have

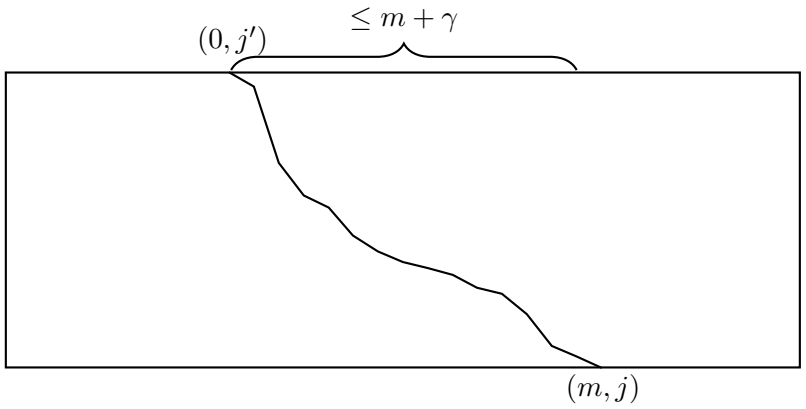
$$\begin{aligned} D_\delta(i,j) &= \min\{edist_\delta(p[1\ldots i], w) \mid w \text{ is a suffix of } t[1\ldots j]\} \\ &= \min\{edist_\delta(p[1\ldots 0], w) \mid w \text{ is a suffix of } t[1\ldots j]\} \\ &= \min\{edist_\delta(\varepsilon, w) \mid w \text{ is a suffix of } t[1\ldots j]\} \\ &= \min\{edist_\delta(\varepsilon, \varepsilon)\} \\ &= 0 \end{aligned}$$

That is, the values in the first row of D_δ are all 0. Intuitively, this means that any prefix of $t[1\ldots j]$ (i.e. $t[1\ldots j'-1]$ if w begins at position j'), can be inserted at no cost. The remaining values of D_δ (i.e. for $i > 0$) are computed in the same way as one computes E_δ:

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

Obviously, each entry in table D_δ can be evaluated in constant time. D_δ can be evaluated column by column from left to right, in the same way as E_δ, see the DP-Algorithm for computing the edit distance. Hence D_δ can be computed in $O(mn)$ time. If only the positions where an approximate match ends are to be enumerated, then $O(m)$ space suffices. To additionally compute approximate matches using a traceback of minimizing paths one does not have to store the entire table D_δ. This is because the threshold k determines an upper bound on the number of insertions and replacements allowed in an alignment of p with a suffix of $t[1\ldots j]$, see Figure 15.

Figure 15: A minimizing path from $D_\delta(m, j)$ to $D_\delta(0, j')$ has a maximum length of $m + \gamma$ where $\gamma = \left\lfloor \frac{k}{\min\{\delta(\varepsilon \to b) | b \in \mathcal{A}\}} \right\rfloor$.



Each replacement (diagonal edge) and insertion (horizontal edge) increases the length of the suffix of $t[1 \ldots j]$ matching p by 1. A deletion (vertical edge) does not affect the length of the matching suffix. So the larger the number of replacements and insertions, the larger the distance of j' and j.

Each replacement involves a different position of p and so the number of replacements is at most m. Each replacement has cost ≥ 0 and each insertion has cost $\geq \min\{\delta(\varepsilon \to b) \mid b \in \mathcal{A}\} > 0$. Thus γ insertions have at least total costs of

$$\gamma \cdot \min\{\delta(\varepsilon \to b) \mid b \in \mathcal{A}\}$$

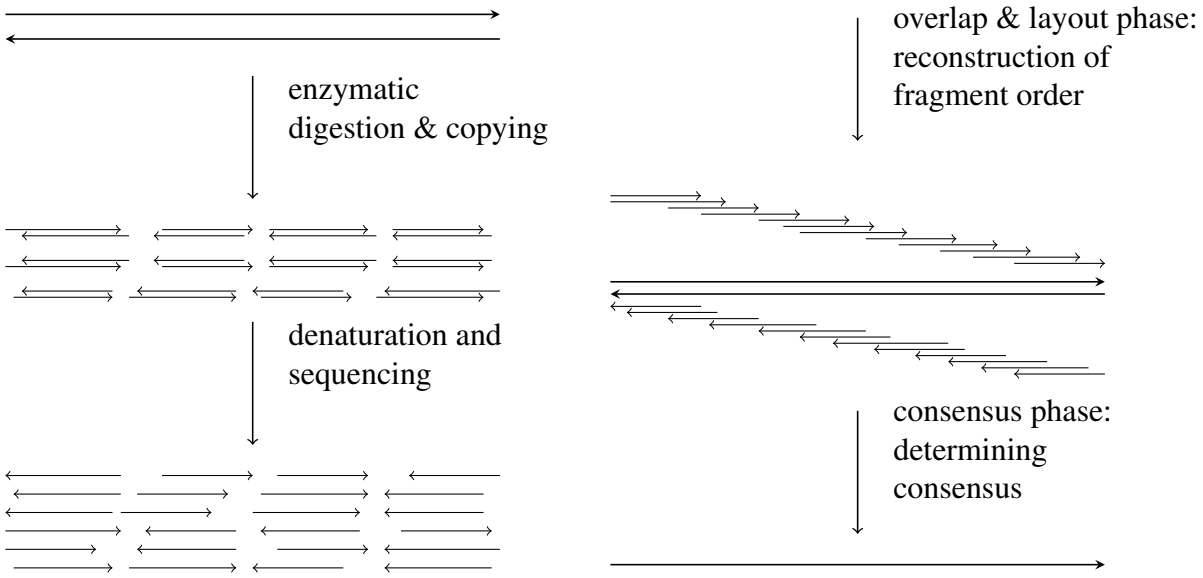
This must be $\leq k$, and from $\gamma \cdot \min\{\delta(\varepsilon \to b) \mid b \in \mathcal{A}\} \leq k$ we conclude

$$\gamma \leq \left\lfloor \frac{k}{\min\{\delta(\varepsilon \to b) \mid b \in \mathcal{A}\}} \right\rfloor.$$

So we have up to $m + \gamma$ replacements and insertions, and thus only have to store $\leq m + \gamma$ columns of D_δ at any time of the computation. So the space requirement is $O(m(m + \gamma))$.

If δ is the unit cost function, then $\min\{\delta(\varepsilon \to b) \mid b \in \mathcal{A}\} = 1$ and so $\gamma = k$. As a consequence, the space requirement is $O(m(m + k))$.

Figure 16: The phases of the shotgun read assembly process



3.6 The overlap problem in read assembly

Sequencing DNA means to determine the sequence of bases, the DNA consists of. This sequence is called *target sequence*. It typically has length between 30 000 and several million bases. The different sequencing technologies do not allow to determine the target sequence directly. Therefore, one applies the *shotgun sequencing technique*. This technique sequences parts of the DNA (called *reads*) as follows:

- the sequencing starts at a random position in one of the two complementary strands of the DNA.
- a continuous stretch of bases is sequenced in canonical direction, from $5'$ to $3'$.

The shotgun sequencing technique delivers many different reads. For each read we know the sequence of bases it consists of, but we do not know (a) the direction of the read (i.e. is it on the forward or reverse complementary strand?), and (b) the start position of the reads relative to the start of the corresponding strand.

The *read assembly problem* now consists of determining the entire target sequence, by assembling the given reads in the correct orientation and order. The different phases of the shotgun assembly process are depicted in Figure 16.

Example 36 Consider the read assembly problem for the following 4 reads:

```
ACCGT
CGTGC
TTAC
TACCGT
```

Assume we know that the target sequence has length 10. Then a possible assembly (shown as a multiple alignment) is as follows:

```
--ACCGT--
----CGTGC
TTAC-----
-TACCGT--
TTACCGTGC
```

The read assembly process can be divided into different phases:

- **Overlap phase:** Determine suffix/prefix overlaps for all pairs of reads. If suffix of one read is similar to a prefix of another read, the two reads are likely from the same regions of the DNA. In Example 36, there is an overlap of length 3 between sequences `ACCGT` and `CGTGC`.
- **Layout phase:** determine a multiple alignment, derived from the positioning of the different reads above each other, and from the introduction of gap symbols.
- **Consensus phase:** determine the target sequence by considering the different columns of the multiple alignment. Choose the base which is dominating a column (majority choice). In the example above, this is easy, since in each column, there is only one base.

In this section, we will focus on the overlap phase. The other phases will be considered in the Genome Informatics lecture.

Note that the reads delivered by the sequencers are not always exact. The amount of errors (i.e. the number inserted, deleted and replaced bases) in the reads varies with the sequencing technology, see the following table:

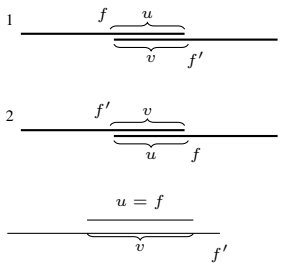
instrument	primary error	errors per bp
Illumina (all models)	substitution	≈ 0.001
Ion Torrent (all chips)	indel	≈ 0.01
Oxford Nanopore	deletions	≥ 0.04
Pacific Biosciences RS	CG deletions	≈ 0.15

source: `http://www.molecularecologist.com/next-gen-table-3c/`

To handle these errors in the reads we therefore have to compute approximate overlaps. Consider two reads f and f', which stem from overlapping parts of the sequenced DNA. Suppose a fixed error rate ϵ, $0 \leq \epsilon \leq 1$, i.e. ϵ is the number of errors per base of the read. By assumption, f can be derived from from the original by at most $\epsilon \cdot |f|$ errors. The corresponding holds for f'. As in the worst case, all errors may be in the overlapping parts of the reads, the overlapping part of f can be transformed with at most $k = \epsilon \cdot |f| + \epsilon \cdot |f'| = \epsilon \cdot (|f| + |f'|)$ edit operations into the overlapping part of f'. So for reads of the same length, say m, we allow $2m\epsilon$ errors, i.e. 2ϵ errors per base. Hence the following definitions make sense:

Definition 22 An alignment A is an *overlap of f and f'*, if there are substrings u of f, v of f', A is an alignment of u and v and at least one of the following properties holds:

1. u is a suffix of f and v is a prefix of f'
2. u is a prefix of f and v is a suffix of f'
3. $u = f$ and v is a substring of f'
4. u is a substring of f and $v = f'$



$len(A) = (|u| + |v|)/2$ is the length of the overlap A. Two reads f and f' *overlap with an error rate* ϵ if and only if there is an overlap A of f and f' such that $\delta(A) \leq k$, where $k = \epsilon \cdot (|f| + |f'|)$.

In general, the number of errors in an overlap will be much smaller than k. Only if all errors appear in the overlap, the number k of errors will be reached. The *overlap problem* for a pair f and f' of reads consists of deciding whether f and f' overlap, and if so, to compute an overlap A of f and f' satisfying $\delta(A) \leq k$.

The dynamic programming techniques developed earlier can be modified to solve the overlap problem. For a read f of length m and a read f' of length n, we compute an $(m+1) \times (n+1)$-table Ov_δ, such that for all (i, j), $0 \leq i \leq m$, $0 \leq j \leq n$ we have

$$Ov_\delta(i,j) = \min\{edist_\delta(u,v) \mid u \text{ is a suffix of } f[1 \ldots i] \text{ and } v = f'[1 \ldots j] \text{ or } u = f[1 \ldots i] \text{ and } v \text{ is a suffix of } f'[1 \ldots j]\}$$

One can show that there is an overlap of f and f', if and only if either

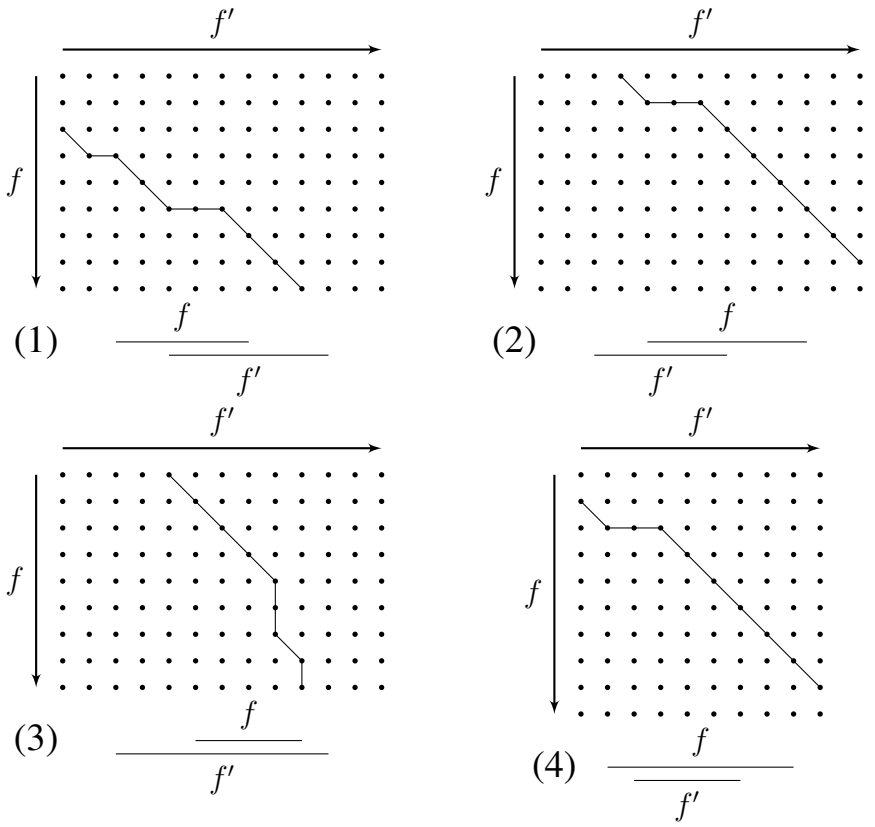
1. $Ov_\delta(i, n) \leq k$ for some i, $0 \leq i \leq n$ or
2. $Ov_\delta(m, j) \leq k$ for some j, $0 \leq j \leq m$.

To compute table Ov_δ, we have to consider that prefixes or suffixes of the reads can be deleted or inserted at no costs. Hence, one needs to set the first row and first column of Ov_δ to 0 and derives the following recurrences:

$$Ov_\delta(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \min \left\{ \begin{array}{ll} Ov_\delta(i-1,j) & + \delta(f[i] \to \varepsilon) \\ Ov_\delta(i,j-1) & + \delta(\varepsilon \to f'[j]) \\ Ov_\delta(i-1,j-1) & + \delta(f[i] \to f'[j]) \end{array} \right\} & \text{otherwise} \end{cases}$$

The four different forms of overlaps between f and f' correspond to four different kinds of paths in the edit graph $G(f, f')$. Each path begins in the first row or first column and it ends in the last row or last column. See the following table and Figure 17, to which the column with header *example* refers. Note that $m = |f|$ and $n = |f'|$.

Figure 17: The paths from a node in the first row or first column of $G(f, f')$ to a node in the last row or last column represents the overlaps between f and f'. Different forms of paths represent different forms of overlaps.

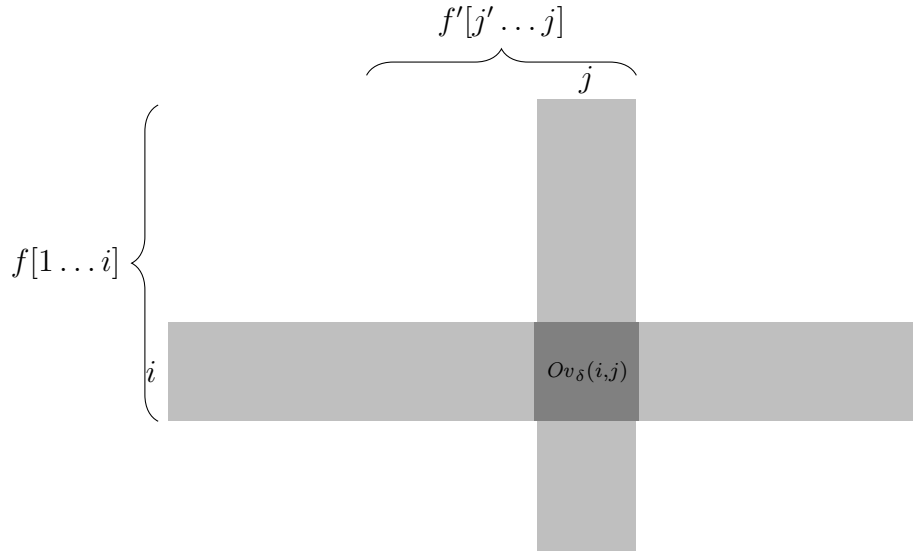


type	example	start position	end position
suffix of f with prefix of f'	(1)	$(i', 0)$	(m, j)
prefix of f with suffix of f'	(2)	$(0, j')$	(i, n)
f with substring of f'	(3)	$(0, j')$	(m, j)
substring of f with f'	(4)	$(i', 0)$	(i, n)

Besides the number of errors one also requires that the length $len(A)$ of the overlap A of f and f' is at least τ, where τ is some parameter. To determine $len(A)$ one maintains the following information:

- for each overlap beginning in the first column, one keeps track of the row from which it starts, minimizing the row number if in the edit graph there is more than one minimizing edge into the current node.

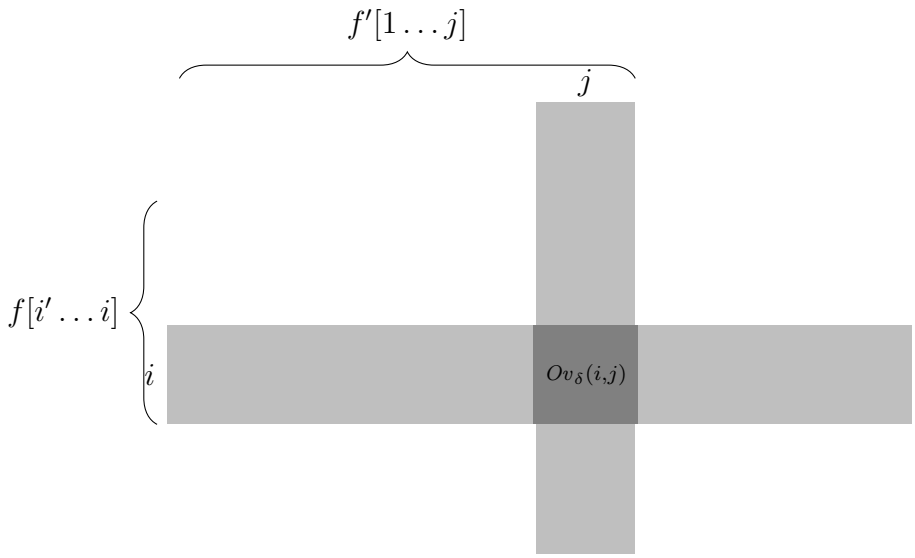
Figure 18: A prefix of f matches a substring of f'. So the minimizing path begins in the first row at column j'.



- for each overlap beginning in the first row, one keeps track of the column from which it starts, minimizing the column number if in the edit graph there is more than one minimizing edge into the current node.

The *minimizing choice* in these cases maximizes the overlap length. As we only consider values in the last row and last column of matrix Ov_δ we can determine the length of the start and end positions of the string involved in the overlap A. This allows to compute $len(A)$ as to verify that $len(A) \geq \tau$ holds.

Figure 19: A prefix of f' matches a substring of f. So the minimizing path begins in first column and row i'.



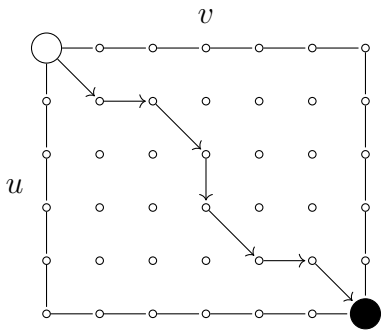
3.7 Overview of pairwise distance comparison methods based on dynamic programming

The sequence comparison methods described so far 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. We have introduced different tables in which the entry at (i, j) refers to different substrings of the aligned pairs of sequences.

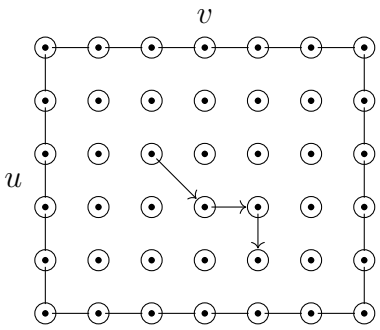
See Figure 18 and 19 for overlaps, as well as Figures 9, 11 and 13 for global alignment, local alignment and approximate string matching, respectively.

With respect to the traceback phase the differences of the methods can best be explained by the edit graph they imply. Each traceback reconstructs an alignment beginning with a start entry and ending with with a final entry, This is depicted in a schematic way in Figure 20.

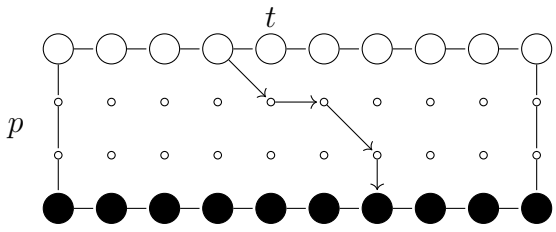
Figure 20: Traceback strategies for different alignment problems in an edit graph. Each traceback strategy is characterized by start entries (i.e. entries where the traceback ends or an optimal alignment starts) and by final entries (i.e. entries where the traceback starts or an alignment ends). Start entries are depicted as circles, while final entries are depicted as bullets.



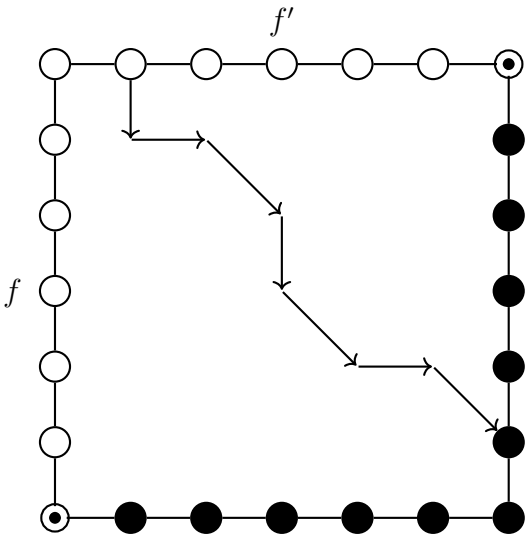
global alignment (table E_δ)



local alignment (table L_σ)

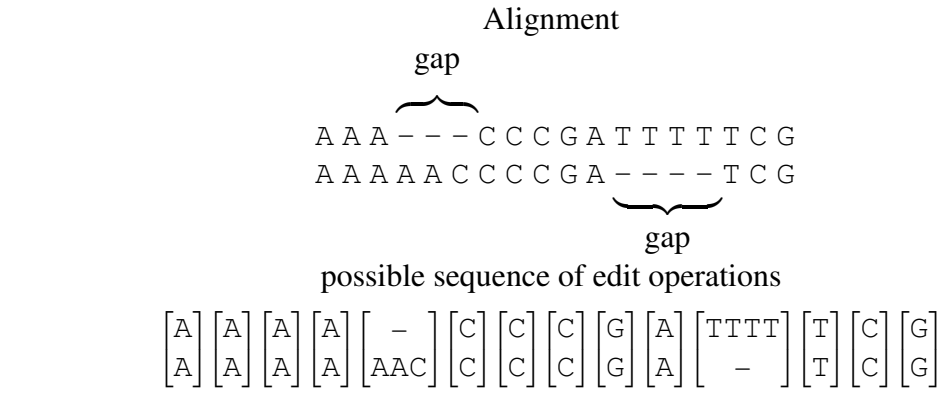


approximate string searching (table D_δ)



fragment overlap (table Ov_δ)

Figure 21: An alignment and the possible sequence of edit operations according to the extended alignment model.



3.8 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

$$(\alpha, \beta) \in (\mathcal{A} \times \mathcal{A}) \cup \underbrace{(\mathcal{A}^+ \times \{\varepsilon\})}_{|\text{gap}| \geq 1} \cup \underbrace{(\{\varepsilon\} \times \mathcal{A}^+)}_{|\text{gap}| \geq 1}$$

for each edit operation $\alpha \to \beta$.

Figure 21 shows an example of an extended alignment with gaps.

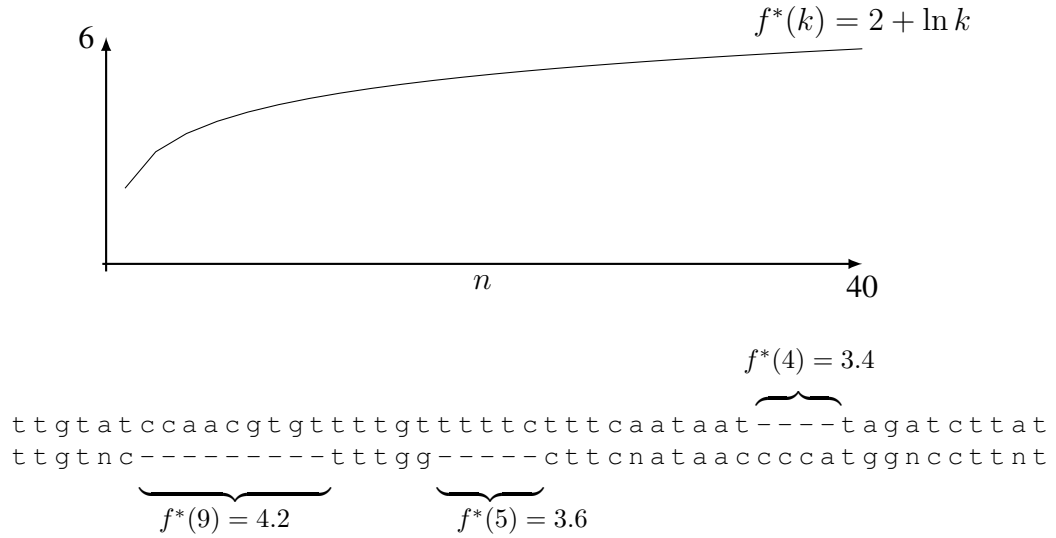
3.8.1 General gap costs

We also have to extend the cost function to the new kinds of edit operations. There are several choices and we start with the general gap cost model. This 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 general gap cost model is defined as the sum of the costs of the generalized edit operations, the alignment consists of. It is denoted by $\delta^*(A)$.

Figure 22: Example of gap cost function $f^*(k) = 2 + \ln k$ and alignment with corresponding gap costs shown above or below the gaps.



To handle general gap costs, one extends the edit graph $G(u, v)$ by additional edges, to accommodate all kinds of gaps. A gap $u[i'+1 \ldots i] \to \varepsilon$ is represented by an edge from (i', j) to (i, j) for some $0 \leq j \leq n$, $1 \leq i \leq m$ and $0 \leq i' \leq i-1$. A gap $\varepsilon \to v[j'+1 \ldots j]$ is represented by an edge from (i, j') to (i, j) for some $0 \leq i \leq m$, $1 \leq j \leq n$ and $0 \leq j' \leq j-1$. Thus, for each (i, j), $0 \leq i \leq m$, $0 \leq j \leq n$, the edit graph is extended by additional edges

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

for all i', $0 \leq i' \leq i-1$ and edges

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

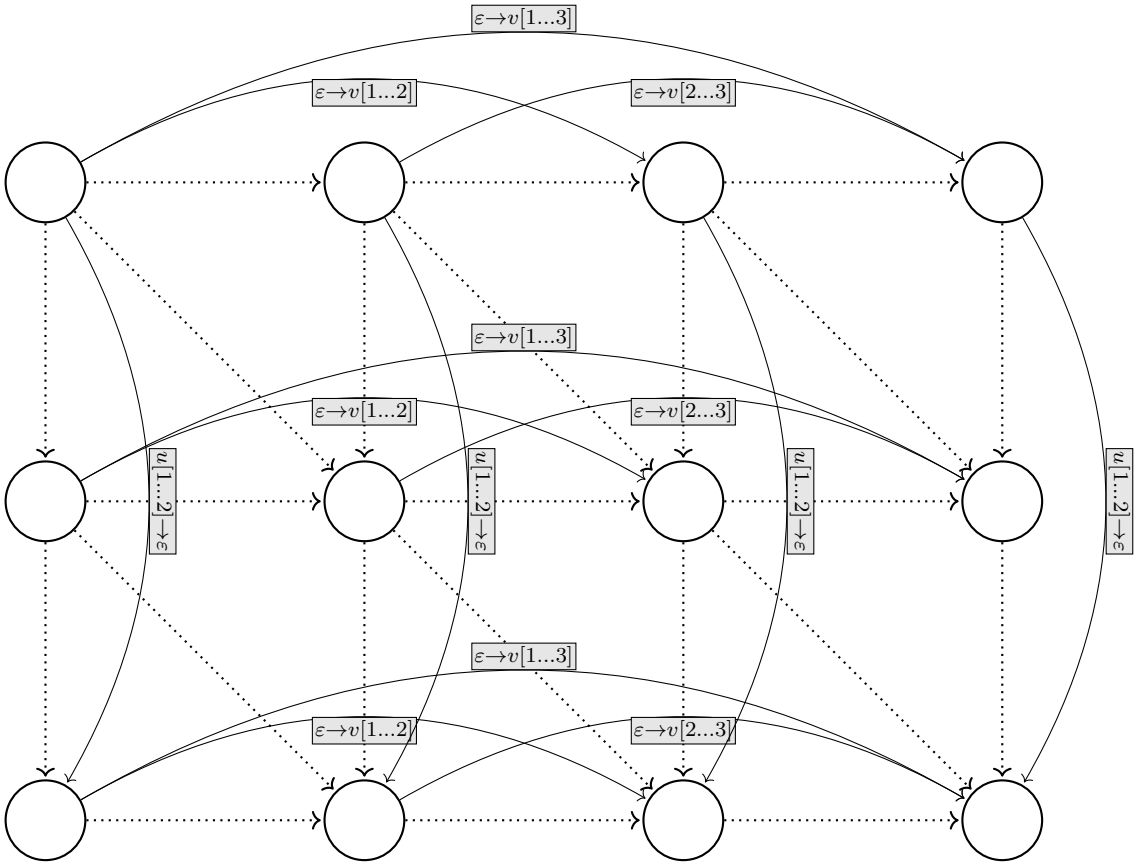
for all j', $0 \leq j' \leq j-1$.

Thus, each node in the edit graph has on average $(n/2-1) + (m/2-1)$ additional edges, see also Figure 23. That is, there are $O(m+n)$ edges into each node and the total number of edges becomes $O(mn\,(m+n))$.

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 derives the following recurrences:

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

Figure 23: The general-gap edit graph for $u[1 \ldots 2]$ and $v[1 \ldots 3]$. The dotted edges also occur in the standard edit graph. The additional thin edges are marked by deletions and insertions of substrings of length at least 2.



Note the compact description of the four different cases. The conditions involving i and j determine whether the value to the left is included in the minimization:

1. if $i > 0$ and $j = 0$, the overall minimum is the second minimum,
2. if $i = 0$ and $j > 0$, the overall minimum is the third minimum
3. if $i > 0$ and $j > 0$, the overall minimum is the minimum of the last three expressions.

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. The algorithm is the Needleman and Wunsch Algorithm [NW70]. Unfortunately, in the literature Algorithm 1 is often also called the Needleman and Wunsch Algorithm, which is misleading, as the latter involves general gaps costs while Algorithm 1 only handles linear gap costs.

3.8.2 Affine gap costs

A simplified cost model for gaps is the *affine gap model*. In this, 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

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

where

- the constant 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 $w[j]$.

The costs for an alignment A with respect to the affine gap cost model is defined as the sum of the costs of all edit operations, where gaps have costs according to function δ_{affine}. The affine gap cost is denoted by $\delta_{\mathsf{affine}}(A)$. The edit distance problem with affine gap costs consists of finding alignments A of u and v such that $\delta_{\mathsf{affine}}(A)$ is minimal.

Example 37 illustrates the effect of the affine gap cost model.

Example 37 This example was adapted from `http://homepage.usask.ca/~ctl271/857/affine_gap_penalties.shtml`.

Consider the following alignment whose cost is determined as follows:

		linear model	affine model	
match	mismatch	indel	gap init	gap ext.
0	1	3	2	1

```
       G A A T T C C G T T A
       |   | |   |   |     |
       G G A T - C - G - - A
linear 0 1 0 0 3 0 3 0 3 3 0 = 13
affine 0 1 0 0 3 0 3 0 3 1 0 = 11

       G A A T T C C G T T A
       |   | |     | |     |
       G G A T - - C G - - A
linear 0 1 0 0 3 3 0 0 3 3 0 = 13
affine 0 1 0 0 3 1 0 0 3 1 0 = 9
```

- make first gap larger, align second `C` in first sequence with first `C` in second sequence

$\Rightarrow$ linear cost remains constant, affine cost is reduced by 2

- second alignment with smaller affine gap cost is the preferred alignment

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

Affine gap costs are a special case of the general gap cost model and thus could be handled by the algorithm described in Section 3.8.1. We will however show that they can be handled more efficiently. The idea, first presented in [Got82], is to extend the edit graph to an affine-cost edit graph. This allows to recognize which edge begins a gap and which edge extends a gap and in turn 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 $(i-1, j-1) \xrightarrow{u[i] \twoheadrightarrow v[j]} (i, j)$ leads to three edges

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

  for all $x \in \{\mathtt{R}, \mathtt{D}, \mathtt{I}\}$, in the affine-cost edit graph.
- A deletion edge $(i-1, j) \xrightarrow{u[i] \twoheadrightarrow \varepsilon} (i, j)$ leads to three edges

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

  for all $x \in \{\mathtt{R}, \mathtt{D}, \mathtt{I}\}$, in the affine-cost edit graph.
- An insertion edge $(i, j-1) \xrightarrow{\varepsilon \twoheadrightarrow v[j]} (i, j)$ leads to three edges

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

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

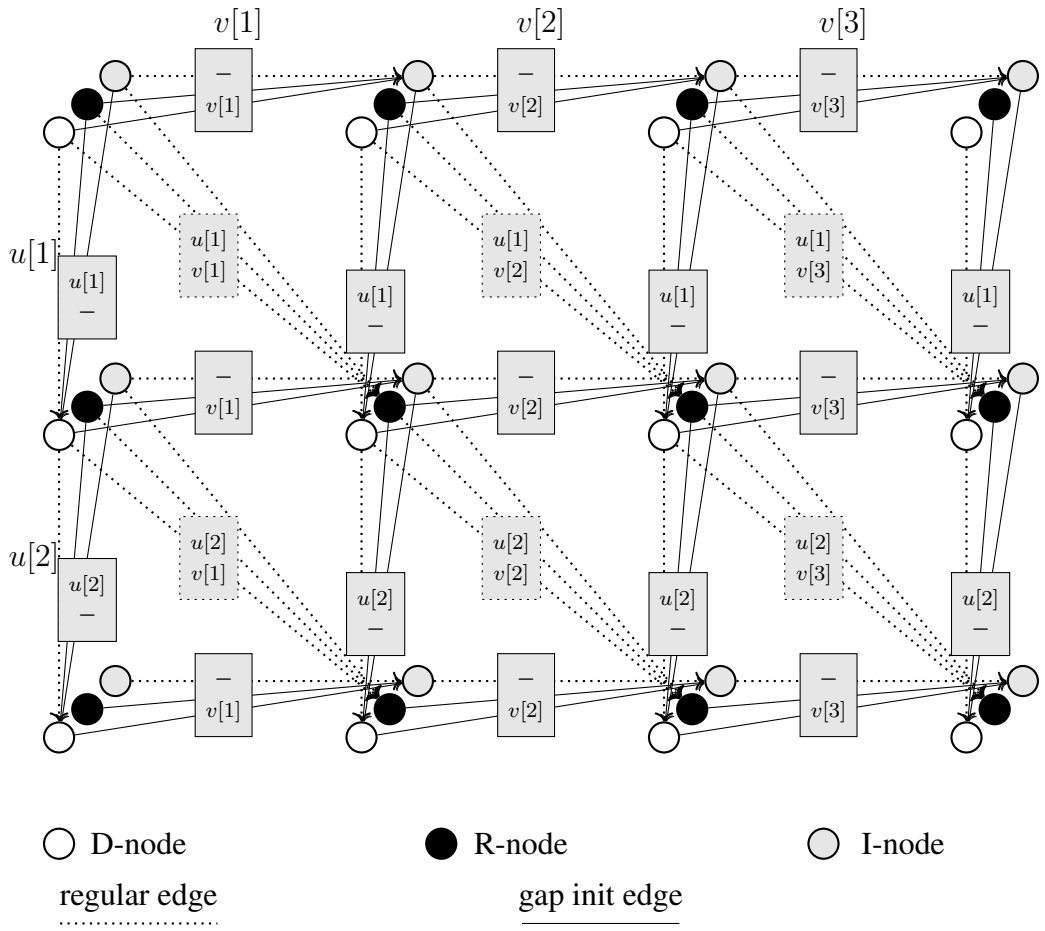
See Figure 24 for an affine-cost edit graph.

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

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

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 $(m, n, \mathtt{R})$,

Figure 24: The affine-cost edit graph for $u[1 \ldots 2]$ and $v[1 \ldots 3]$.



$(m, n, \mathtt{D})$, and $(m, n, \mathtt{I})$ as final entries. According to the considerations above, the edit graph has three times as many 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_{afn} such that the following holds:

- If $i = j = 0$, then $A_{\mathsf{afn}}(i, j, \mathtt{R}) = 0$ and $A_{\mathsf{afn}}(i, j, y) = g$ for $y \in \{\mathtt{D}, \mathtt{I}\}$. The latter handles the case for an alignment with an initial gap.
- If $i > 0$ or $j > 0$, then

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

One can derive the following recurrences for A_{afn}. As the boundary cases are already defined, we can restrict to the case that $i > 0$ or $j > 0$.

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

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

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

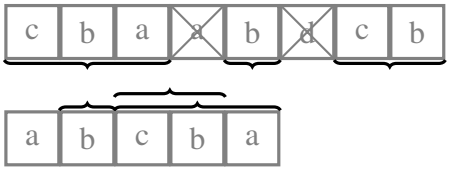
The cost of optimal affine gap cost alignment is

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

Obviously, each entry in table A_{afn} can be evaluated in constant time. Thus the algorithm runs in $O(mn)$ time and space. As there are approximately three times as many nodes and edges in the affine cost edit graph, compared to the standard edit graph, the practical running time with increase by a factor of about 3.

3.9 The maximal matches model for sequence comparison

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, as shown in the following illustration:



The key to the model is the notion of partition. Recall that u and v are strings of length m and n, respectively.

Definition 23 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.

The notion of partition is illustrated in Figure 25.

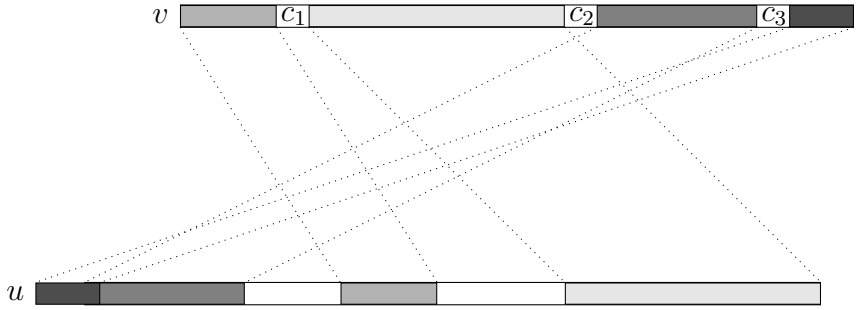
Example 38 Let $v =$ `cbaabdcb` and $u =$ `abcba`. $\Psi_1 = (\texttt{cba}, \texttt{a}, \texttt{b}, \texttt{d}, \texttt{cb})$ is a partition of v with respect to u, since `cba`, `b`, and `cb` are substrings of u. $\Psi_2 = (\texttt{cb}, \texttt{a}, \texttt{ab}, \texttt{d}, \texttt{cb})$ is a partition of v with respect to u, since `cb` and `ab` are substrings of u. Both partitions are of size 2. Suppose there is a partition of v with respect to u of size less than 2. This would be a partition with zero or one marked character, requiring that there is a submatch of size at least $\left\lceil \frac{|v|-1}{2} \right\rceil = \left\lceil \frac{8-1}{2} \right\rceil = 4$. But as the longest common substring of v and u is 3, such a partition does not exist. So Ψ_1 and Ψ_2 are of minimal size. Hence, *mmdist*$(v, u) = 2$.

There are two canonical partitions.

Definition 24 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, $1 \leq h \leq r$, $w_h c_h$ is not a substring of u, then Ψ is the *left-to-right partition* of v with respect to u. If for all h, $1 \leq h \leq r$, $c_h w_{h+1}$ 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 39 For the strings $v =$ `cbaabdcb` and $u =$ `abcba` of Example 38 we have:

Figure 25: Illustration of a partition of sequence v with respect to u. The blocks of different grey-scales are the submatches separated by marked characters. These blocks match substrings of the same grey-scale in sequence u. Additionally, the correspondence of submatches in v to substrings in u is shown by dotted lines connecting the left- and rightmost positions of involved strings. The substrings matched in u can overlap, as is the case for the first two substrings. The size of the partition is 3.



- $\Psi_{lr}(v, u) = \Psi_1 = (\texttt{cba}, \texttt{a}, \texttt{b}, \texttt{d}, \texttt{cb})$ as `cbaa` and `bd` are not substrings of u.
- $\Psi_{rl}(v, u) = \Psi_2 = (\texttt{cb}, \texttt{a}, \texttt{ab}, \texttt{d}, \texttt{cb})$ as `dcb` and `aab` are not substrings of u.

Example 40 For the strings $v' = \texttt{abcba}$ and $u' = \texttt{cbaabdcb}$. We have:

- $\Psi_{lr}(v', u') = (\texttt{ab}, \texttt{c}, \texttt{ba})$ as `abc` is not a substring of u'.
- $\Psi_{rl}(v', u') = (\texttt{a}, \texttt{b}, \texttt{cba})$ as `bcba` is not a substring of u.

One can show that $\Psi_{lr}(v, u)$ and $\Psi_{rl}(v, u)$ are of minimal size, see [EH88]. Hence, we can conclude $|\Psi_{lr}(v, u)| = \textit{mmdist}(v, u) = |\Psi_{rl}(v, u)|$. 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. Algorithm 3 gives pseudocode for the computation of the size of the left-to-right partition.

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 $ST(u)$.

Algorithm 3 (Computation of left-to-right partition)

Input: sequences $u = u[1 \ldots m]$ and $v = v[1 \ldots n]$
Output: size of left-to-right-partition

```
1:  function lrpartitionsize(u,v)
2:      s ← ε                                          ▷ s: substring of u
3:      marked ← 0                                     ▷ number of marked chars
4:      for idx = 1 to n do
5:          c ← v[idx]
6:          if sc is a substring of u then
7:              s ← sc
8:          else
9:              marked ← marked + 1                    ▷ marked char. c at pos idx
10:             s ← ε
11:         end if
12:     end for
13:     return marked
14: end function
```

Example 41 Let us reconsider the strings $v = \texttt{cbaabdcb}$ and $u = \texttt{abcba}$ from Example 39. We have seen that

- $\Psi_{lr}(v, u) = (\texttt{cba}, \texttt{a}, \texttt{b}, \texttt{d}, \texttt{cb})$ which implies $\mathit{mmdist}(v, u) = 2$ and
- $\Psi_{lr}(u, v) = (\texttt{ab}, \texttt{c}, \texttt{ba})$ which implies $\mathit{mmdist}(u, v) = 1$

As $\mathit{mmdist}(v, u) = 2 \neq 1 = \mathit{mmdist}(u, v) = 1$, *mmdist* is not symmetric. Hence, *mmdist* is not a metric on $\mathcal{A}^*$.

However, one can obtain a metric as follows:

Theorem 5 Let $mmm(u, v) = \log_2((\mathit{mmdist}(u, v) + 1) \cdot (\mathit{mmdist}(v, u) + 1))$. mmm is a metric on $\mathcal{A}^*$. □

From the above it is clear that $mmm(u, v)$ can be computed in $O(|\mathcal{A}| \cdot (m + n))$ time 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. The idea is that an alignment can be turned into a partition in which consecutive matches form the submatches and characters replaced and inserted in v are the marked characters.

Lemma 6 Let δ be the unit cost function. Consider an alignment A of v and u. Then there is an r, $0 \leq r \leq \delta(A)$, 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', $0 \leq r' \leq \delta(A')$ 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 a 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 6 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 Lemma 6 there is a partition Ψ of v with respect to u of size $r \leq \delta(A)$. *mmdist*$(v, u) \leq |\Psi| = r \leq \delta(A) = edist_\delta(v, u)$. □

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. For example, suppose we have a set S of sequences and want to find all pairs $(s, s') \in S \times S'$, $s \neq s'$ such that $edist_\delta(s, s') \leq k$. Instead of determining $edist_\delta(s, s')$ for each pair s, s' using $O(|s| \cdot |s'|)$ time, one first computes *mmdist*(s, s') in $O(|s| + |s'|)$ time. If *mmdist*$(s, s') > k$, then by Theorem 6 we have $edist_\delta(s, s') > k$, so that the pair (s, s') does not have to be considered further. That is, *mmdist* serves as a filter for the expensive computation of the unit edit distance. If *mmdist*$(s, s') \leq k$, then we have to compute $edist_\delta(s, s')$. In fact, there are algorithms for the approximate sequence searching problem using filtering techniques based on maximal matches.

3.10 The q-gram sequence comparison model

Like the maximal matches model, the q-gram model considers common substrings of the strings to be compared. Here $q > 0$ and a *q-gram* of a sequence s is a substring of s of length q. A q-gram is sometimes called q-mer or q-tuple. Technically, in this model, one counts the number of occurrences of different q-grams in the two sequences to be compared. 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 25 The *q-gram profile* of u is the function $P_{u,q} : \mathcal{A}^q \to \mathbb{N}$, such that for any $w \in \mathcal{A}^q$, $P_{u,q}(w)$ is the number of different positions in u where w occurs as substring. □

Example 42 Let $\mathcal{A} = \{\mathtt{a}, \mathtt{c}\}$ and $q = 2$. The q-gram profile of $u = \mathtt{aaca}$ is

$$\mathtt{aa} \mapsto 1, \mathtt{ac} \mapsto 1, \mathtt{ca} \mapsto 1, \mathtt{cc} \mapsto 0$$

The q-gram profile of $v = \mathtt{acacaacc}$ is

$$\mathtt{aa} \mapsto 1, \mathtt{ac} \mapsto 3, \mathtt{ca} \mapsto 2, \mathtt{cc} \mapsto 1$$

The size of the alphabet $\mathcal{A}$ and the choice of q determine the q-gram profile. 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-grams occur, i.e. all values in the profile are > 0. If $q = 4$ and $|\mathcal{A}| = 20$, then $|\mathcal{A}|^q = 160\,000$ and the string has to be very long to contain all q-grams. In general, one chooses $q \ll n$, e.g. $8 \leq q \leq 11$ when comparing, e.g. entire bacterial genomes.

As for given $\mathcal{A}$ and q the q-grams are uniquely determined, one often writes the q-gram profile as an ordered list

$$[P_{u,q}(w_0), P_{u,q}(w_1), \ldots, P_{u,q}(w_{r^q-1})]$$

with the q-grams w_0, w_1, $\ldots w_{r^q-1}$ in lexicographic order, where $r = |\mathcal{A}|$.

Example 43 Let $\mathcal{A} = \{\mathtt{a}, \mathtt{c}\}$ and $q = 2$. The two q-gram profile $\mathtt{aa} \mapsto 1, \mathtt{ac} \mapsto 1, \mathtt{ca} \mapsto 1, \mathtt{cc} \mapsto 0$ is written as $[1, 1, 1, 0]$. The q-gram profile $\mathtt{aa} \mapsto 1, \mathtt{ac} \mapsto 3, \mathtt{ca} \mapsto 2, \mathtt{cc} \mapsto 1$ is written as $[1, 3, 2, 1]$.

The q-gram distance is just the sum of the absolute differences of the profile-lists.

Definition 26 The *q-gram distance* $qgdist(u, v)$ of u and v is defined by

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

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.

Example 44 Let $\mathcal{A} = \{\mathtt{a}, \mathtt{c}\}$ and $q = 2$. Then $u = \mathtt{aaca}$ and $v = \mathtt{acaa}$ have the same q-gram profile

$$\mathtt{aa} \mapsto 1, \mathtt{ac} \mapsto 1, \mathtt{ca} \mapsto 1, \mathtt{cc} \mapsto 0$$

Hence, the q-gram distance of u and v is 0. As $u \neq v$, this contradicts the zero-property ($f(x, y) = 0 \iff x = y$). So $qgdist$ is not a metric. □

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 27 Let $\mathcal{A} = \{a_1, \ldots, a_r\}$ be an ordered alphabet such that $a_1 < a_2 < \cdots < a_r$. Then

$$\overline{a_\ell} = \ell - 1$$

is the code of a_ℓ and

$$\overline{w} = \sum_{i=1}^{q} \underbrace{\overline{w[i]}}_{\substack{\text{char}\\ \text{code}}} \cdot \underbrace{r^{q-i}}_{\text{weight}}$$

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

Example 45 Let $\mathcal{A} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ be the DNA-alphabet and define $\overline{\mathtt{A}} = 0$, $\overline{\mathtt{C}} = 1$, $\overline{\mathtt{G}} = 2$, and $\overline{\mathtt{T}} = 3$. For $q = 3$, there are $r^q = 4^3 = 64$ q-grams. Here are some examples of how the codes are computed:

$$\begin{aligned}
\overline{\mathtt{AAA}} &= \overline{\mathtt{A}} \cdot 4^2 + \overline{\mathtt{A}} \cdot 4^1 + \overline{\mathtt{A}} \cdot 4^0 = 0 \cdot 16 + 0 \cdot 4 + 0 \cdot 1 = 0 \\
\overline{\mathtt{ATA}} &= \overline{\mathtt{A}} \cdot 4^2 + \overline{\mathtt{T}} \cdot 4^1 + \overline{\mathtt{A}} \cdot 4^0 = 0 \cdot 16 + 3 \cdot 4 + 0 \cdot 1 = 12 \\
\overline{\mathtt{CGT}} &= \overline{\mathtt{C}} \cdot 4^2 + \overline{\mathtt{G}} \cdot 4^1 + \overline{\mathtt{T}} \cdot 4^0 = 1 \cdot 16 + 2 \cdot 4 + 3 \cdot 1 = 16 + 8 + 3 = 27 \\
\overline{\mathtt{TAA}} &= \overline{\mathtt{T}} \cdot 4^2 + \overline{\mathtt{A}} \cdot 4^1 + \overline{\mathtt{A}} \cdot 4^0 = 3 \cdot 16 + 0 \cdot 4 + 0 \cdot 1 = 48 \\
\overline{\mathtt{TTT}} &= \overline{\mathtt{T}} \cdot 4^2 + \overline{\mathtt{T}} \cdot 4^1 + \overline{\mathtt{T}} \cdot 4^0 = 3 \cdot 16 + 3 \cdot 4 + 3 \cdot 1 = 48 + 12 + 3 = 63
\end{aligned}$$

Let us generalize on the previous example. The first character in w is weighted by r^{q-1}, the second character by r^{q-2}, etc. If all characters in $w \in \mathcal{A}^q$ 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 $w \in \mathcal{A}^q$ 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, the maximum code of any $w \in \mathcal{A}^m$ is $r^q - 1$. Moreover, for any $u, w \in \mathcal{A}^q$, $\overline{u} = \overline{w}$ implies $u = v$ (proof will be an exercise). 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 $\overline{w} = i$. To put it into mathematical terms, the mapping from q-grams to integer codes is bijective.

Example 46 Let $\mathcal{A} = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ be the DNA-alphabet and define $\overline{\mathtt{A}} = 0$, $\overline{\mathtt{C}} = 1$, $\overline{\mathtt{G}} = 2$, and $\overline{\mathtt{T}} = 3$. For $q = 3$, there are $r^q = 4^3 = 64$ q-grams. The smallest q-gram AAA in the lexicographic order of all q-grams has integer code 0, the second smallest AAC has integer code 1, etc. In general, for any q and any ordered alphabet, the ith q-gram in the lexicographic order of all q-grams gets code i, see Figure 26 for an example.

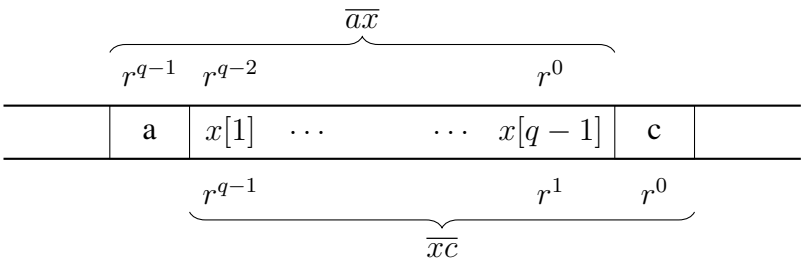
An important property is that the code of each q-gram in a sequence can be computed incrementally in constant time, due to the fact that

$$\overline{xc} = (\overline{ax} - \overline{a} \cdot r^{q-1}) \cdot r + \overline{c}$$

for any $x \in \mathcal{A}^{q-1}$ and any $a, c \in \mathcal{A}$, see the following illustration:

Figure 26: All 3-grams over the alphabet $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ with $\overline{\mathtt{A}} = 0$, $\overline{\mathtt{C}} = 1$, $\overline{\mathtt{G}} = 2$, and $\overline{\mathtt{T}} = 3$.

AAA 0	ACA 4	AGA 8	ATA 12
AAC 1	ACC 5	AGC 9	ATC 13
AAG 2	ACG 6	AGG 10	ATG 14
AAT 3	ACT 7	AGT 11	ATT 15
CAA 16	CCA 20	CGA 24	CTA 28
CAC 17	CCC 21	CGC 25	CTC 29
CAG 18	CCG 22	CGG 26	CTG 30
CAT 19	CCT 23	CGT 27	CTT 31
GAA 32	GCA 36	GGA 40	GTA 44
GAC 33	GCC 37	GGC 41	GTC 45
GAG 34	GCG 38	GGG 42	GTG 46
GAT 35	GCT 39	GGT 43	GTT 47
TAA 48	TCA 52	TGA 56	TTA 60
TAC 49	TCC 53	TGC 57	TTC 61
TAG 50	TCG 54	TGG 58	TTG 62
TAT 51	TCT 55	TGT 59	TTT 63



Here the weight for the window covering ax are shown above the characters they are multiplied with. Similarly, the weight for the window covering xc are shown below the characters they are multiplied with.

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

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

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

2. Compute the set $C = \{\overline{w} \mid w \text{ is } q\text{-gram of } u \text{ or } v\}$, i.e. the set of codes of all q-grams in u and w.

Algorithm 4 (Computation of q-gram distance)

Input: sequences $u = u[1 \ldots m]$, $v = v[1 \ldots n]$ over alphabet $\mathcal{A}$, $q > 0$
Output: $qgdist(u, v)$

1: $r \leftarrow |\mathcal{A}|$
2: **for** $c \leftarrow 0$ **upto** $r^q - 1$ **do**
3: $\quad (\tau_u[c], \tau_v[c]) \leftarrow (0, 0)$
4: **end for**
5: $c \leftarrow \sum_{i=1}^{q} \overline{u[i]} \cdot r^{q-i}$
6: $\tau_u[c] \leftarrow 1$
7: $C \leftarrow \{c\}$
8: **for** $i \leftarrow 1$ **upto** $m - q$ **do**
9: $\quad c \leftarrow (c - \overline{u[i]} \cdot r^{q-1}) \cdot r + \overline{u[i+q]}$
10: $\quad$ **if** $\tau_u[c] = 0$ **then**
11: $\quad\quad C \leftarrow C \cup \{c\}$
12: $\quad$ **end if**
13: $\quad \tau_u[c] \leftarrow \tau_u[c] + 1$
14: **end for**
15: $c \leftarrow \sum_{i=1}^{q} \overline{v[i]} \cdot r^{q-i}$
16: $\tau_v[c] \leftarrow 1$
17: **if** $\tau_u[c] = 0$ **then**
18: $\quad C \leftarrow C \cup \{c\}$
19: **end if**
20: **for** $i \leftarrow 1$ **upto** $n - q$ **do**
21: $\quad c \leftarrow (c - \overline{v[i]} \cdot r^{q-1}) \cdot r + \overline{v[i+q]}$
22: $\quad$ **if** $\tau_u[c] = 0$ and $\tau_v[c] = 0$ **then**
23: $\quad\quad C \leftarrow C \cup \{c\}$
24: $\quad$ **end if**
25: $\quad \tau_v[c] \leftarrow \tau_v[c] + 1$
26: **end for**
27: **return** $\sum_{c \in C} |\tau_u[c] - \tau_v[c]|$

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 $O(1)$ time. Hence the total running time 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 7 Let δ be the unit cost function. Then $qgdist(u, v)/(2 \cdot q) \leq edist_\delta(u, v)$.

Proof: See [JU91] or [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. See the remarks at the end of the section on the maximal matches model.

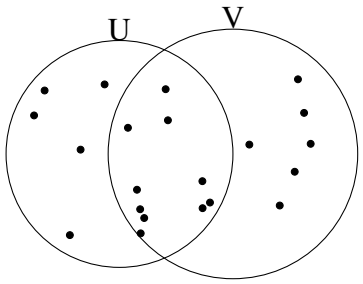
3.11 Mash: fast genome and metagenome distance estimation using MinHash

3.11.1 The Jaccard Index: general concept

- this section is based on[2] referred to by Ondov et al. in the following
- several frames are based on the presentation of the topic by Leo Förster, Genome Informatics Seminar, Wintersemester 2016/2017
- overall goal: compute a distance between two sequences
- distance can be determined by looking at what is common
- the more in common, the smaller the distance
- naive approach via Jaccard Index[3]

Jaccard Index

- is fraction of shared elements of two sets: $J(U,V) = \frac{|U \cap V|}{|U \cup V|}$



$$J(U,V) = \frac{|U \cap V|}{|U \cup V|} = \frac{10}{21} = 0.4762$$

Properties

- $0 \leq |U \cap V| \leq |U \cup V| \Rightarrow 0 \leq J(U,V) \leq 1$
- $U \cap V = \emptyset \Rightarrow J(U,V) = \frac{|U \cap V|}{|U \cup V|} = \frac{|\emptyset|}{|U \cup V|} = \frac{0}{|U \cup V|} = 0$
- $U = V \Rightarrow J(U,V) = \frac{|U \cap V|}{|U \cup V|} = \frac{|U|}{|U|} = 1$

3.11.2 The Jaccard Index: application to comparing sequences

- the comparison of sequences using the Jaccard index is based on the sets $G_q(u)$ and $G_q(v)$ of all q-grams of u and v, respectively.
- that is, one computes $J(U,V)$ for $U = G_q(u)$ and $V = G_q(v)$
- so the black dots in previous figure represent q-grams
- abbreviation: $J(u,v) = J(G_q(u), G_q(v))$

[2]Ondov, B. D., Treangen, T. J., Melsted, P., Mallonee, A. B., Bergman, N. H., Koren, S., and Phillippy, A. M. (2016).Mash: fast genome and metagenome distance estimation using MinHash. *Genome Biology*, 17(1):1–14.

[3]Jaccard, P (1901), Etude comparative de la distribution florale dans une portion des Alpes et des Jura, *Bullet. de la Société Vaudoise des Sciences Natur., 37: 547-579*

Efficiency

- common q-grams can be efficiently counted:
  - for u and v enumerate and encode q-grams as integers (as done before)
  - store integer codes in two lists and sort each list using radix sort
  - find common elements in two sorted integer lists by a merging approach (see following slides)
- for $m = |u|$, $n = |v|$: $O(\underbrace{m+n}_{\text{enumerate}} + \underbrace{m+n}_{\text{sort}} + \underbrace{m+n}_{\text{merge}})$ time

$\Rightarrow$ $O(m+n)$ space and time (optimal)

Algorithm 5 (Count common elements in two sorted lists of integers)

Input: $ulist$, $vlist$: sorted lists of integers and without duplicates
Output: number of common elements

```
1:  (i, j, common) ← (0, 0, 0)
2:  while i < |ulist| and j < |vlist| do
3:      if ulist[i] < vlist[j] then
4:          i ← i + 1
5:      else
6:          if ulist[i] > vlist[j] then
7:              j ← j + 1
8:          else
9:              (i, j, common) ← (i + 1, j + 1, common + 1)
10:         end if
11:     end if
12: end while
13: print common
```

3.11.3 Mash: the basic idea

- for the comparison of two single sequences u and v, the previous method is considerably faster than the algorithm computing the q-gram distance
- but if there are hundreds or thousands of sequences to be compared (all-against-all), the method is likely not efficient enough (especially in terms of space requirement)
- in such an application scenario, Mash (Ondov et. al.) considerably reduces the runtime and space requirement

Mash (Ondov et. al.)

1. computes reduced representation of q-grams of sequence (sketch)
2. *estimates* Jaccard Index from sketches of the sequences

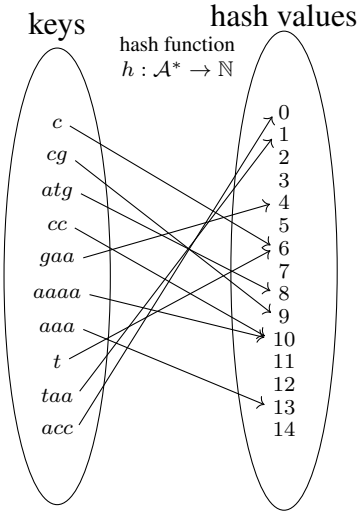
   (Jaccard estimate is often close to the Jaccard Index)
3. derives distance measure from Jaccard estimate

3.11.4 Excursion to hash functions

- one of the very basic tasks in computer science is to efficiently store values associated with keys
- simple solution:
  - store key/value pairs in list and use linear search to find value for given key in $O(n)$ time and space for n key value pairs

  - sort the keys and use binary search to find the value for a given key in $O(\log n)$ time and $O(n)$ space
- one usually wants to access the value for any key in $O(1)$ time
- a very common way to achieve this is to uniquely associate a key with an index of an array where to store the value
- this association is established by a hash function:

  a hash function maps any kind of (hashable) object to a unique integer
- see example for string-keys below



- collision when $h(w) = h(w')$ for $w \neq w'$, as in $h(cc) = 10 = h(aaaa)$ or $h(c) = 6 = h(t)$
- strategies to solve such conflicts: hashing with chaining, double hashing, open addressing, cuckoo hashing ...
- a hash function is used in a Python-dictionary or a Ruby-Hash or a map in the C++-standard template library
- it is hidden from the user
- Python: obtain hash-value via method `hash`, e.g. `hash('atcg')` $\Rightarrow$ 1 231 534 521 241 347 127
- can be applied to any hashable object (e.g. strings, numbers, functions)

3.11.5 Examples of hash functions for strings

$$\text{js}(s) = h_1(s, |s|) \text{ where}$$
$$h_1(s,i) = \begin{cases} 0 & \text{if } i = 0 \\ (ord(s[i]) + h_1(s, i-1) \cdot 2^5 + h_1(s, i-1)/4) & \\ \quad \hat{} \; h_1(s, i-1) & \text{otherwise} \end{cases}$$

$$\text{sdbm}(s) = h_2(s, |s|) \text{ where}$$
$$h_2(s,i) = \begin{cases} 0 & \text{if } i = 0 \\ ord(s[i]) + h_2(s, i-1) \cdot (2^6 + 2^{16} - 1) & \text{otherwise} \end{cases}$$

$$bp(s) = h_3(s, |s|) \text{ where}$$
$$h_3(s,i) = \begin{cases} 0 & \text{if } i = 0 \\ ord(s[i]) \; \hat{} \; (h_3(s, i-1) \cdot 2^7) & \text{otherwise} \end{cases}$$

- ord maps characters to integers

- ^ stands for exclusive or

3.11.6 Mash: sketch

Input

- input sequence u
- length q
- sketch size $s \geq 1$
- hash-function $h : \mathcal{A}^q \to \mathbb{N}$

$$u = \texttt{gtgatgtagtgagaagtag}$$
$$\Downarrow\ q = 5$$
$$G_q(u) = \left\{ \begin{matrix} \texttt{gtgat gtagt gagaa} \\ \texttt{tgatg tagtg agaag} \\ \texttt{gatgt agtga gaagt} \\ \texttt{atgta gtgag aagta} \\ \texttt{tgtag tgaga agtag} \end{matrix} \right\}$$

Algorithm

- enumerate set $G_q(u)$ of all q-grams of u
- compute $H_{q,h}(u) = \{h(w) \mid w \in G_q(u)\}$
- keep only the s smallest values of $H_{q,h}(u)$
- this is the sketch of u, denoted by $S_{q,h,s}(u)$
- fix q, h, s and abbreviate $S_{q,h,s}(u)$ by $S(u)$

$$\Downarrow\ h$$
$$H_{q,h}(u) = \left\{ \begin{matrix} 26\ 35\ 62 \\ 78\ 56\ 33 \\ 34\ \ 6\ 44 \\ \ 9\ 64\ 94 \\ 66\ 10\ 72 \end{matrix} \right\}$$
$$\Downarrow\ s = 5$$
$$S(u) = \{6, 9, 10, 26, 33\}$$

3.11.7 Mash: hash functions

- requirement for hash-function: avoid collisions, i.e. different q-grams w and t satisfying

$$h(w) = h(t)$$

- q-gram encoding, as used for q-gram distance has no collisions
- but when encoding A $\mapsto$ 0, C $\mapsto$ 1, G $\mapsto$ 2, T $\mapsto$ 3, poly-A sequences get smallest hash-values and more likely are represented in sketches
- there are other encodings, which avoid such effect, but require slightly more runtime to compute

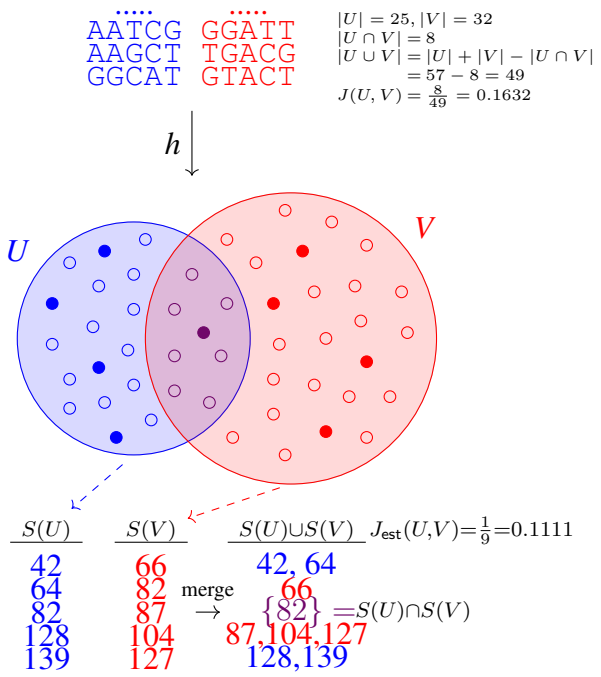
MinHash

- the fact that the smallest s hash-values form the sketch leads to the name of the technique: MinHash[4]
- for appropriate hash-functions $S(u)$ is a random sample of u

[4] Broder, 1997, where this technique was used for identifying duplicate documents on the WEB

- random samples allow to derive an estimate of the Jaccard Index

Idea
as $S(U)$ and $S(V)$ are random samples of the q-grams of u and v, the fraction of the shared items in $S(U)$ and $S(V)$ is expected to be similar as in U and V
Jaccard estimate $J_{\text{est}}(U,V) = \frac{|S(U) \cap S(V)|}{|S(U) \cup S(V)|}$



3.11.8 Computation of Jaccard estimate $J_{\text{est}}(U,V)$

Computation of $S(U)$ for $U = G_q(u)$ with $m = |u|$

- enumerate all q-grams of u: $O(m)$ runtime, $O(q)$ space
- apply hash function f to each q-gram: $O(mq)$ runtime, $O(1)$ space
- use min-heap to keep s smallest hash-values seen so far: $O(m \log s)$ runtime, $O(s)$ space

$\Rightarrow$ $O(mq + m \log s) = O(m(q + \log s))$ runtime, $O(s)$ space

Computation of $S(V)$ for $V = G_q(v)$ with $n = |v|$

$\Rightarrow$ $O(n(q + \log s))$ runtime, $O(s)$ space

Computation of $|S(U) \cap S(V)|$

- merge 2 sorted lists of integers of length s to count common elements: $O(s)$ runtime, $O(1)$ space

Computation of $|S(U) \cup S(V)|$

- compute $|S(U)| + |S(V)| - |S(U) \cap S(V)|$ in constant time

Total runtime

- $O(\underbrace{m(q+\log s)}_{\text{sketch of } u} + \underbrace{n(q+\log s)}_{\text{sketch of } v} + \underbrace{s}_{merge}) = O((m+n)(q+\log s)+s)$ runtime

Total Space requirement

- $O(s)$

- compare to computation of q-gram distance in $O(m+n+r^q)$ runtime and space
- MinHash-based method does not have improved running time, but uses less space

3.11.9 All-against-all comparison via Jaccard estimate

All-against-all

- given set $T = \{t_1, t_2, \ldots, t_k\}$ of k sequences
- all-against-all comparison: compare all pairs t_i and t_j, $1 \leq i < j \leq k$

Jaccard estimate for all pairs t_i, t_j

1. precompute sketches of all t_i and store each on file in sorted order
   $O(|t_1|(q+\log s)+|t_2|(q+\log s)+\cdots+|t_k|(q+\log s)) = O(n(q+\log s))$ time, where $n = \sum_{i=1}^{k} |t_i|$
2. compute Jaccard estimate for each pair t_i, t_j, $i < j$ in $O(s)$ runtime: $O(k^2 s)$ runtime for all pairs

$O(n(q+\log s)+k^2 s)$ total runtime $\Rightarrow$ very fast, as only step 1 depends on length of sequences

In the following we will show that the Jaccard estimate allows to derive an approximation of the mutation rate of the sequences to be compared

3.11.10 Excursion to Poisson Distributions

Definition 28 Poisson Distribution A Poisson Distribution P_λ is a discrete probability distribution with a parameter λ (describing the average number of events per interval) defined by

$$P_\lambda(k \text{ events in interval}) = \mathrm{e}^{-\lambda}\frac{\lambda^k}{k!}$$

with $k \in \mathbb{N}$ and $\mathrm{e} \approx 2.71828$ is the Eulerian Number.

A Poisson distributions is used to quantify the probability of events which occur in a period of time or a geographical location, if the following holds:

- k is the number of times an event occurs in an interval
- the events occur independently

- the rate at which the events occur is constant
- two events never occur at the same sub-interval or sub-location

Applications of Poisson Distributions

- Telecommunication: telephone calls arriving in a system
- Astronomy: photons arriving at a telescope
- Management: customers arriving at a counter
- Finance: number of losses occurring in a given period of time
- Land management: number of overfloods at a coast line
- Earthquake seismology: seismic risk for large earthquakes
- Radioactivity: number of decays in a given time interval in a radioactive sample
- Text editing: number of typos on a printed page
- Biology: number of mutations on a strand of DNA per unit length

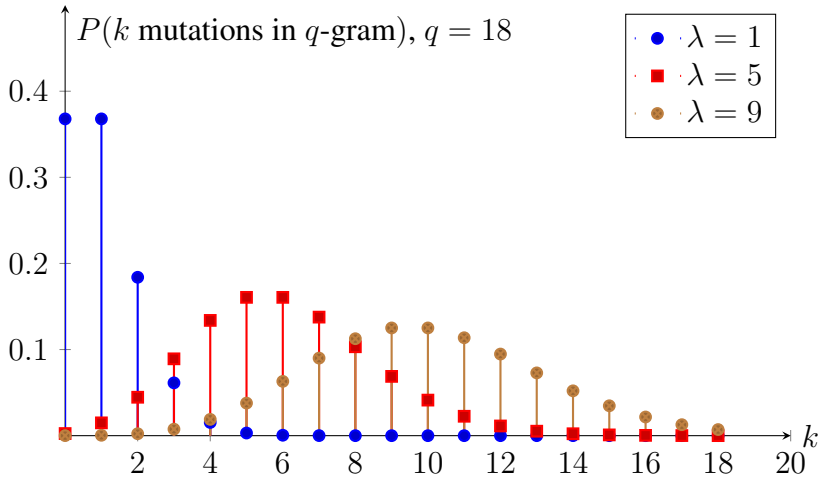
from https://en.wikipedia.org/wiki/Poisson_distribution

3.11.11 Poisson distributions for modelling DNA-mutations

- consider a DNA sequence in which we have on average 1 mutation per q-gram for $q = 18$
- so our interval size is 18 and $\lambda = 1$
- assuming a Poisson distribution we calculate

$$P_\lambda(k \text{ mutations in } q\text{-gram}) = \mathrm{e}^{-\lambda}\frac{\lambda^k}{k!} = \mathrm{e}^{-1}\frac{1^k}{1!}$$
$$P_\lambda(0 \text{ mutations in } q\text{-gram}) = \mathrm{e}^{-1}\frac{1^0}{0!} = \mathrm{e}^{-1} \approx 0.368$$
$$P_\lambda(1 \text{ mutations in } q\text{-gram}) = \mathrm{e}^{-1}\frac{1^1}{1!} = \mathrm{e}^{-1} \approx 0.368$$
$$P_\lambda(2 \text{ mutations in } q\text{-gram}) = \mathrm{e}^{-1}\frac{1^2}{2!} = \frac{\mathrm{e}^{-1}}{2} \approx 0.184$$

3.11.12 Poisson distribution for $\lambda \in \{1, 5, 9\}$



from https://tex.stackexchange.com/questions/282806/plot-the-poisson-function-correctly

3.11.13 Probability of matching q-grams

Assumption of evolutionary model of sequences

- sequences differ by mutations (indels and mismatches)
- mutations occur at single nucleotides randomly and independently with a constant rate d per base over the entire sequence
- so qd is average mutation rate per q-gram

$\Rightarrow$ probability of events (i.e. mutations) occurring in q-gram of the sequence follows a Poisson distribution with $\lambda = qd$, i.e.

$$
\begin{aligned}
P_\lambda(\overbrace{k \text{ mutations in } q\text{-gram}}^{\text{event}}) &= \mathrm{e}^{-qd}\frac{(qd)^k}{k!} \\
\Rightarrow P_\lambda(\text{matching } q\text{-gram}) &= P_\lambda(0 \text{ mutations in } q\text{-gram}) \\
&= \mathrm{e}^{-qd}\frac{(qd)^0}{0!} \\
&= \mathrm{e}^{-qd}\frac{1}{1} = \mathrm{e}^{-qd}
\end{aligned}
$$

3.11.14 Probability of q-matches and Jaccard Index

- let $n = \frac{|u|+|v|}{2}$ be the average length of u and v

- if e^{-qd} is the probability of a single q-gram match, then we expect

$$\gamma = n \cdot \mathrm{e}^{-qd}$$

  q-gram matches between the two sequences u and v
- Recall: with $U = G_q(u)$ and $V = G_q(v)$ we have

$$J(u,v) = J(U,V) = \frac{|U \cap V|}{|U \cup V|} \approx \frac{\gamma}{2n - \gamma} \tag{3.4}$$

- In (3.4) one ignores that q-grams may occur more than once $\Rightarrow \gamma \geq |U \cap V|$ and $2n - \gamma \geq |U \cup V|$
- but the ratio is very similar (so we use the symbol $\approx$ to express this)
- In $J(u,v) \approx \frac{\gamma}{2n-\gamma}$ we substitute γ by $n \cdot \mathrm{e}^{-qd}$:

$$\begin{aligned}
& J(u,v) \approx \frac{n \cdot \mathrm{e}^{-qd}}{2n - n \cdot \mathrm{e}^{-qd}} = \frac{n \cdot \mathrm{e}^{-qd}}{n \cdot (2 - \mathrm{e}^{-qd})} = \frac{\mathrm{e}^{-qd}}{2 - \mathrm{e}^{-qd}} \\
\iff & J(u,v) \cdot (2 - \mathrm{e}^{-qd}) \approx \mathrm{e}^{-qd} \\
\iff & 2 \cdot J(u,v) - J(u,v) \cdot \mathrm{e}^{-qd} \approx \mathrm{e}^{-qd} \\
\iff & 2 \cdot J(u,v) \approx J(u,v) \cdot \mathrm{e}^{-qd} + \mathrm{e}^{-qd} \\
\iff & 2 \cdot J(u,v) \approx (J(u,v) + 1) \cdot \mathrm{e}^{-qd} \\
\iff & \frac{2 \cdot J(u,v)}{J(u,v) + 1} \approx \mathrm{e}^{-qd}
\end{aligned}$$

- switching the left and right-hand size of this (approximate) equation, we obtain

$$\begin{aligned}
& \mathrm{e}^{-qd} \approx \frac{2 \cdot J(u,v)}{J(u,v) + 1} \\
\iff & \ln \mathrm{e}^{-qd} \approx \ln \frac{2 \cdot J(u,v)}{J(u,v) + 1} \\
\iff & -qd \approx \ln \frac{2 \cdot J(u,v)}{J(u,v) + 1} \\
\iff & d \approx -\frac{1}{q} \ln \frac{2 \cdot J(u,v)}{J(u,v) + 1}
\end{aligned}$$

- so we can estimate the mutation rate d (which is what we are interested in) in terms of $J(u,v)$ and q

3.11.15 Mutation rate and Jaccard Estimate

Now the Jaccard estimate comes back into play:

$$\text{as } J_{\text{est}}(u,v) = J_{\text{est}}(U,V) = \frac{|S(U) \cap S(V)|}{|S(U) \cup S(V)|} \approx J(u,v)$$

$$\text{we conclude } d \approx -\frac{1}{q} \ln \frac{2 \cdot J_{\text{est}}(u,v)}{J_{\text{est}}(u,v)+1}$$

- the mash distance $MD_{q,s}(u,v)$ derives the mutation rate of u and v from the Jaccard estimate of u and v:

$$MD_{q,s}(u,v) = \begin{cases} 1 & \text{if } J_{\text{est}}(u,v) = 0 \\ -\frac{1}{q} \ln \frac{2 \cdot J_{\text{est}}(u,v)}{J_{\text{est}}(u,v)+1} & \text{otherwise} \end{cases}$$

Just like $J_{\text{est}}(u,v)$, $MD_{q,s}(u,v)$ can be computed in $O(s)$ space and $O((|u|+|v|)(q+\log s))$ time

3.11.16 Gold standard for genome distance

- to determine whether the mash distance is a reliable distance estimator, one needs to compare it to a gold standard, i.e. a widely trusted measure of the distance of two (genome) sequences
- such a gold standard is based on the *average nucleotide identity* of u and v (abbreviation: $ANI(u,v)$)
- $ANI(u,v)$ is determined from a set $SLA(u,v)$ of significant local alignments of u and v:

$$ANI(u,v) = \frac{1}{|SLA(u,v)|} \sum_{A \in SLA(u,v)} \left(1 - \frac{2 \cdot \delta(A)}{|A.u| + |A.v|}\right) \tag{3.5}$$

where

- δ is the unit cost function,
- $A.u/A.v$ are the substrings of u/v involved in alignment A
- so ratio in (3.5) is the relative number of errors in alignment

Computing $SLA(u,v)$

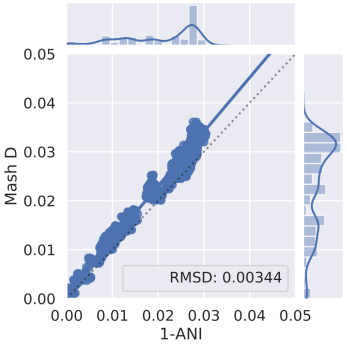
- $SLA(u, v)$ can be computed by any suitable program delivering local alignments
- as the DP-based Smith-Waterman algorithm is often too slow, one resorts to seed-extend methods (see section on Fasta and Blast) to compute local alignments
- the program `nucmer` and `delta-filter` from the MUMmer-software (Kurtz et al. 2004) are widely used in this context (and was applied in the results presented below)
- computations using `nucmer`/`delta-filter` still take much longer than methods not based on alignments

3.11.17 Results 1: verification against gold standard

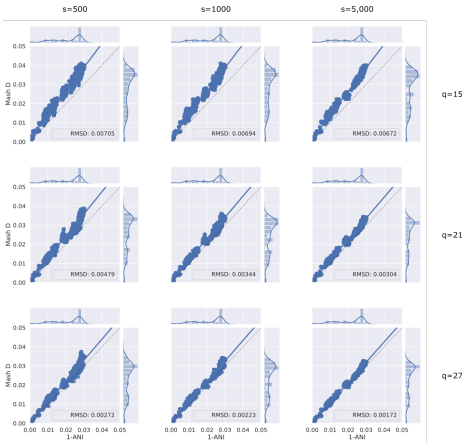
- input data: 500 *Escherichia* genomes $\Rightarrow 500 \cdot (500 - 1)/2 = 124\,750$ sequence pairs
- for each pair of sequences u and v plot a dot at coordinate $(1-ANI(u, v), MD_{q,s}(u, v))$, i.e. $1 - ANI(u, v)$ is plotted on X-axis and $MD_{q,s}(u, v)$ is plotted on Y-axis $\Rightarrow$ scatter plot

- ideally $\frac{MD_{q,s}()}{1-ANI()} = 1 \Rightarrow$ all values on dotted line
- blue lines: result of a linear regression of the dot-coordinates
- RMSD measures deviation of vectors of $MD_{q,s}()$- and $(1 - ANI()$-values
- histogram on top: distribution of $(1 - ANI())$-values
- rotated histogram on the right: distribution of $MD_{q,s}()$-values

result for $q = 21$, $s = 1\,000$



- rows: scatter plots for constant values of $q = 15$ (first row), $q = 21$ (second row) and $q = 27$ (third row)
- columns: scatter plots for constant values of $s = 500$ (1. col.), $s = 1\,000$ (2. col.) and $s = 5\,000$ (3. col.)



	RMSD from plots		
	s		
q	500	1 000	5 000
15	.00705	.00694	.00672
21	.00479	.00344	.00603
27	.00272	.00223	.00172

- $s \uparrow \Rightarrow$ RMSD $\downarrow$ for $q = 15, 27$
- $q \uparrow \Rightarrow$ RMSD $\downarrow$

$\Rightarrow$ $MD_{q,s}()$ is a very good estimator for distances of closely related genomes

3.11.18 Results 2: Clustering all RefSeq sequences (Ondov et al.)

- input: all genomes in NCBI RefSeq Release 70, i.e. 618 GB of genomic sequences from 54 118 organisms
- compute sketches for $q = 16$ and $s = 1\,000$

$\Rightarrow$ total size of all sketches: 93 MB (i.e. 0.091 GB)

- space reduction by factor $\geq 6\,800$
- runtime for computing sketches: 26.1 CPU hours
- $\frac{54\,118 \cdot (54\,118-1)}{2} \approx 1.46 \cdot 10^9$ pairwise comparisons in 6.9 CPU hours
- link all genomes u and v if $MD_{q,s}(u, v) \leq 0.05$ and p-value[5] of the distance is $\leq 1.0e - 10$
- resulting clustering, restricted to bacteria and viruses, is shown on Figure 27
- genomes belonging to same taxonomic group form (in most cases) own clusters $\Rightarrow$ consistency with taxonomy

3.11.19 Results 3: Clustering 17 primate genomes (Ondov et al.)

- input: 17 RefSeq primate genomes, each of length in the range from $2.6 \cdot 10^9$ to $3.2 \cdot 10^9$ bp
- compute mash distances in 2.5 CPU h for $s = 1\,000$ and $q = 21$

[5] method for computing this is described by Ondov et. al.

Figure 27: Result of clustering bacteria and viruses using `mash`

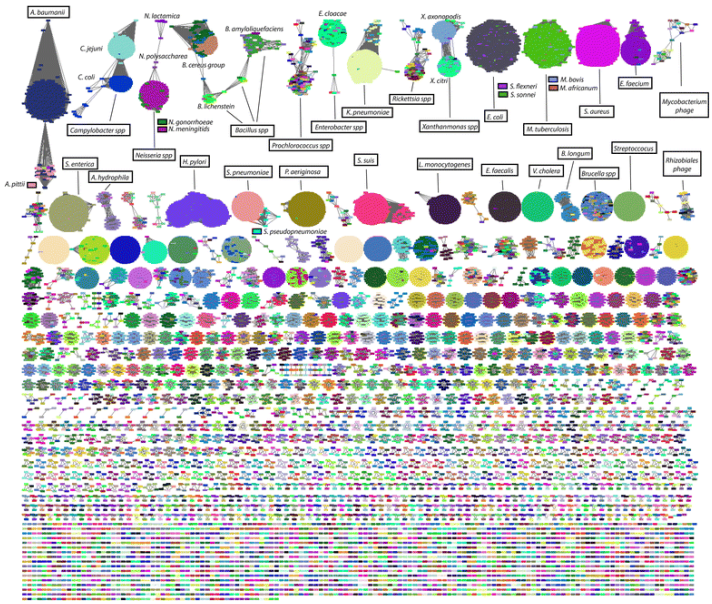
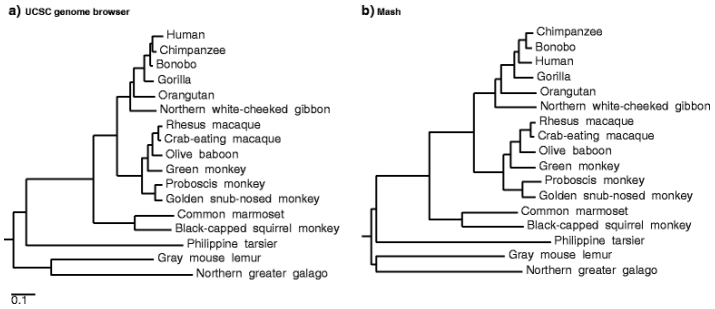


Figure 28: Phylogeny of 17 primate genomes: a) from UCSC genome browser and b) determined based on distances computed by `mash`



- from mash distances construct a phylogenetic tree using the Neighbor-Joining algorithm (discussed in the section on Phylogeny)
- compare it with alignment-based phylogenetic tree model downloaded from the UCSC genome browser, see Figure 28.

- topologically, both phylogenies are consistent, except for the split of Human versus Chimpanzee/Bonobo

- in this branch the mash topology is more similar to past phylogenetic studies and mitochondrial trees
- on average, branch lengths of the mash-based tree are slightly longer

3.11.20 Impact of the MinHash concept in genome informatics

- since 2015 the MinHash concept was used in several improved methods for solving important problems in large scale sequence comparison:
  - identification of overlaps in long reads for sequence assembly by MHAP [BKC+15],
  - mapping of long reads to reference sequences by MashMap [JDK+17],
  - computing homology-maps for complete genomes [JDK+17],
  - estimation of containments of long reads by Mash-Screen [OSS+19],
  - all-against-all comparison of 90 000 prokaryotic genomes [JRRP+18],
  - weighted minimizer sampling for mapping long reads in repetitive genomes [JRZ+20],
  - k-mer counting [DKGDG15, KDD17].
- the review [MSPK19, page 108-113] gives an overview of the theory and of applications of the MinHash concept

3.11.21 Conclusion

- sketches are constant size representations of samples of q-grams from a set of sequences
- for a sequence of length n, its sketch of size s can be computed in $O(n(q + \log s))$ time
- sketches are random samples and can be used to estimate the Jaccard Index of the sets of all q-grams of two sequences
- assuming the mutations are Poisson-distributed, one can show that the Jaccard Index is related to the mutation rate
- the mutation rate can thus be approximated by the Jaccard estimate to obtain the Mash distance $MD_{q,s}(u, v)$
- the evolutionary model based on Poisson processes is very simple and does not attempt to model more complex evolutionary processes
- there are some interesting applications of the Mash distance, such as large scale clustering of sequence sets or construction of phylogenies
- Ondov et. al. report more applications, like real-time genome identification from assemblies or reads and clustering massive metagenomic datasets

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 6 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.

Table 6: Overview of Database Search Methods as adapted from the textbook of Mount. Table 7 lists the URLs of the mentioned software tools.

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 7: The URLs of the software tools listed in Table 6

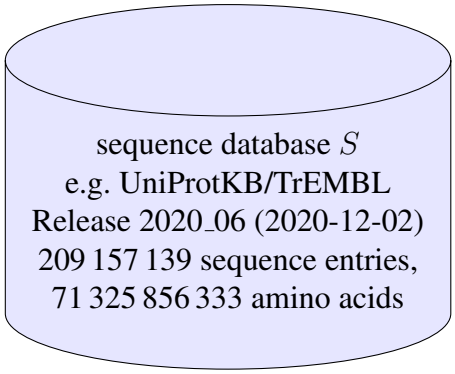
Tool	Web-Address
PROFILESEARCH	ftp.sdsc.edu/pub/sdsc/biology/
SSEARCH	http://fasta.bioch.virginia.edu/fasta/
BlastP	http://www.ncbi.nlm.nih.gov/BLAST/
WU-BLAST	http://blast.wustl.edu/
MAST	http://meme.sdsc.edu/meme/website/mast.html
PSI-BLAST	http://www.ncbi.nlm.nih.gov/BLAST
PROSITE	http://www.expasy.ch/prosite
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

Figure 29: Sample output of the program `ssearch36` (which implements the Smith-Waterman using SIMD-acceleration), when comparing a single protein sequence `mGSTM1` against a database of 13 143 protein sequences.

```
# ../bin/ssearch36 -q -w 80 ../seq/mgstm1.aa proteindatabase.fasta
1>>>mGSTM1 mouse glutathione transferase M1 - 218 aa
Library: PIR1 Annotated (rel. 66) 5121825 residues in 13143 sequences
Algorithm: Smith-Waterman (SSE2, Michael Farrar 2006) (7.2 Nov 2010)
Parameters: BL50 matrix (15:-5), open/ext: -10/-2
The best scores are:                                          s-w bits E(13143)
sp|P14942|GSTA4_RAT Glutathione S-transferase alpha-4; GST 8-8 (222) 179 49.9 6.1e-07
... (alignments deleted) ...
>>sp|P14942|GSTA4_RAT Glutathione S-transferase alpha-4; GST 8-8;            (222 aa)
Smith-Waterman score: 179; 25.6% identity (54.5% similar) in 75 aa overlap
                10        20        30        40        50        60        70
mGSTM    MPMILGYWNVRGLTHPIRMLLEYTDSSYDEKRYTMGDAPDFDRSQWLNEKF-KLG-LDFPNLPYL-IDGSHKITQSNA
             : :.. ::  . :: ::  .   ..:         .: ...   ::. : : : : ..: . :::   .::. :
sp|P14 MEVKPKLYYFQGRGRMESIRWLLATAGVEFEE---------EFLETREQYEKLQKDGCLLFGQVPLVEIDG-MLLTQTRA
            10        20        30                40        50        60         70
... (alignments deleted) ...
218 residues in 1 query   sequences
Total Scan time:  3.820 Total Display time:  0.130
```

4.1 Local Similarity Searches with Fasta

Fasta is a popular tool for comparing biological sequences. It was introduced in [LP85] and is further described in [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), illustrated as follows:



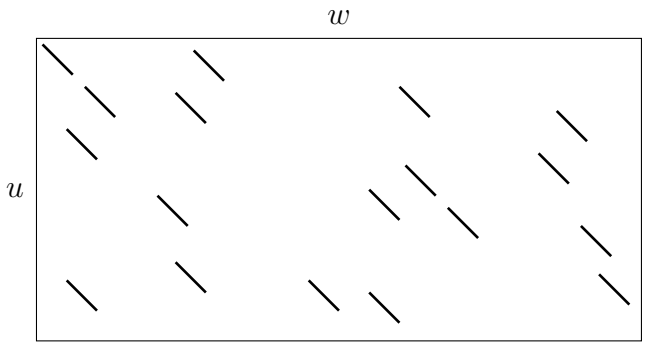

```
>query sequence w
MPMILGYWNVRGLTHPIRML
```

The problem is to find all sequences in S, which have local similarities with w and to display these similarities in form of alignments and their positions within the sequences, see Figure 29 for an example.

Applying the Smith-Waterman Algorithm to w and each sequence from S is too slow. The idea is to quickly eliminate many sequences from S, which likely do not contain any interesting local alignments. The remaining (hopefully few) sequences can be processed by expensive DP-based methods like the SW-Algorithm.

Such a filtering approach requires a similarity criterion and thresholds referring to the database sequences and the query sequence. The criterion must satisfy at least these three conditions:

Figure 30: Hot spots between the query sequence w and the database sequence u, as considered by the Fasta-Algorithm. Each Hot spot represents a q-gram occurring in u and w.



1. One must be able to quickly determine the database sequences satisfying/not satisfying the criterion.
2. The number of sequences satisfying the criterion must be very small compared to the entire sequence database
3. The sequences not satisfying the criterion do not have high local similarities to the query sequence.

4.1.1 Finding hot spots

Fasta provides such a filtering approach which is described here. There is no formally well-defined model of what Fasta computes, but a heuristic stepwise method defined next.

Consider an arbitrary but fixed $u \in S$. 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 common q-grams in u and w. In the context of Fasta these are called "hot-spots", see Figure 30 for an illustration. The number of hot spots on each diagonal gives a score, according to the definition following the figure.

Definition 29 Let $m = |u|$ and $n = |w|$. For d, $-m \leq d \leq n$ let

$$\mathit{hotsp}(u, w, d) = |\{(i, j) \mid \overbrace{1 \leq i \leq m - q + 1}^{\text{startpos in } u}, \overbrace{1 \leq j \leq n - q + 1}^{\text{startpos in } w}, \overbrace{j - i = d}^{\text{diag}}, \underbrace{u[i \ldots i + q - 1] = w[j \ldots j + q - 1]}_{\text{matching } q\text{-grams}}\}|$$

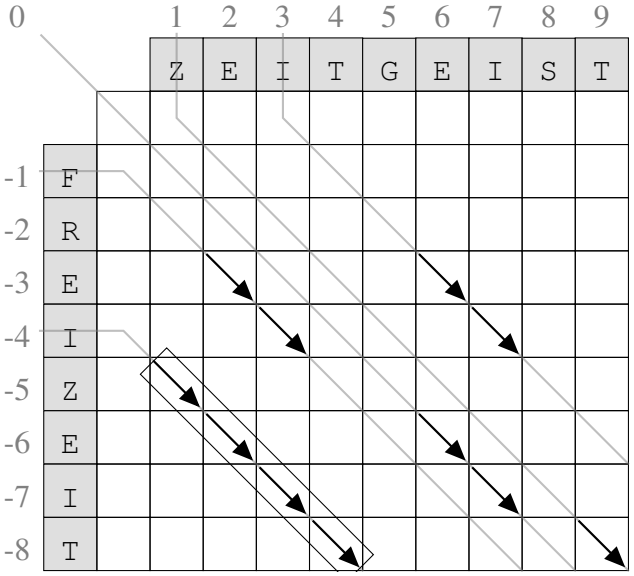
So $\mathit{hotsp}(u, w, d)$ is the number of matching q-grams in u and w whose start position pair (i, j) is on diagonal d. We are interested in the maximum *hotsp*-value for all diagonals.

Definition 30 The Fasta score is defined by $score_{\mathsf{fasta}}(u, w) = \max\{\mathit{hotsp}(u, w, d) \mid -m \leq d \leq n\}$. □

Example 47

Let $q = 2$, $u =$ `FREIZEIT` and $w =$ `ZEITGEIST`. In the table on the right, matching characters are represented by diagonal arcs. Diagonals and their numbers are shown in grey. We have

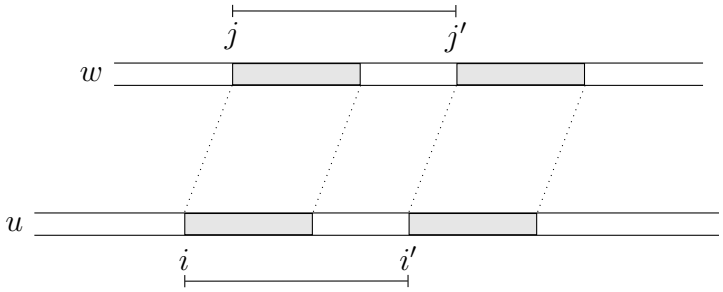
$\mathit{hotsp}(u, w, -4) = 3$,
$\mathit{hotsp}(u, w, -1) = 1$,
$\mathit{hotsp}(u, w, 0) = 1$,
$\mathit{hotsp}(u, w, 3) = 1$ and
$\mathit{hotsp}(u, w, d) = 0$ for
$d \notin \{-4, -1, 0, 3\}$



Note that only the q-grams on the same diagonal are counted. That is, if

$$u[i \ldots i+q-1] = w[j \ldots j+q-1] \text{ and}$$
$$u[i' \ldots i'+q-1] = w[j' \ldots j'+q-1]$$

are on the same diagonal d, we have $j - i = d = j' - i'$ which implies $j - i = j' - i'$ and therefore $i' - i = j' - j$, i.e. the start positions of the matching q-grams have the same distance in both u and w, see the following illustration:



The fact that the order of the q-grams is relevant is the main difference to the q-gram distance model, where the order of the q-grams is not important.

A crucial step of the Fasta-Algorithm is to first preprocess the query sequence w. This preprocessing step (which is independent of the database sequences) gathers information which allows us to efficiently determine the matching q-grams and thus the *hotsp*-values. Preprocessing makes sense, as we have to process w many times, namely for each database sequence. So

Algorithm 6 (Computing $score_{\mathsf{fasta}}(u, w)$)

1. Encode each q-gram as an integer c, $0 \leq c \leq r^q - 1$, where $r = |\mathcal{A}|$. The details of this encoding are described in Section 3.10.

2. The query sequence w is preprocessed into a table h_w such that for each c, $0 \leq c \leq r^q - 1$ we have
$$h_w(c) = \{i \mid \underbrace{1 \leq i \leq |w| - q + 1}_{\text{start pos of } q\text{-gram in } w}, c = \underbrace{\overline{w[i \ldots i + q - 1]}}_{\text{code of } q\text{-gram}}\}$$
That is, $h_w(c)$ holds the positions in w where the q-grams with integer code c occurs.

3. In the final phase, the database is processed as follows:

```
1:  n ← |w|
2:  for all u ∈ S do
3:      m ← |u|
4:      for d ← −m upto n do
5:          hotsp(u, w, d) ← 0
6:      end for
7:      for j ← 1 upto m − q + 1 do
8:          c ← u[j ... j + q − 1]
9:          for all i ∈ h_w(c) do
10:             hotsp(u, w, j − i) ← hotsp(u, w, j − i) + 1
11:         end for
12:     end for
13:     score_fasta(u, w) ← max{hotsp(u, w, d) | −m ≤ d ≤ n}
14: end for
```

the additional effort of the preprocessing likely pays off. Algorithm 6 provides details on how $score_{\mathsf{fasta}}(u, w)$ is computed. The preprocessing of w into table h_w can be done by scanning w twice: In the first scan, for each c, $0 \leq c \leq r^q - 1$, the size of $h_w(c)$ is determined. This is the same as computing the q-gram profile of w and takes $O(n + r^q)$ time, as we have seen in the section on the q-gram distance. Then one determines for each c, $0 \leq c \leq r^q - 1$, the partial sums $P(c) = \sum_{c' < c} |h_w(c')|$ in $O(r^q)$ time. Let H be an array of size $n - q + 1$. In a second scan over w one inserts in H the positions in w where a q-gram starts as follows:

```
1: for i ← 1 upto n − q + 1 do
2:     c ← w[i ... i + q − 1]                    ▷ O(1) time
3:     H[P(c)] ← i
4:     P(c) ← P(c) + 1
5: end for
```

Thus the preprocessing takes $O(n + r^q)$ time. H contains the start positions of all q-grams in w lexicographically ordered by the q-grams (i.e. their integer codes). For any c, $0 \leq$

$c \leq r^q - 1$, the subarray $H[\ell \ldots P(c) - 1]$ stores the elements in $h_w(c)$ where $\ell =$ if $c = 0$ then 0 else $P(c-1)$. So all elements in $h_w(c)$ can be enumerated in $O(|h_w(c)|)$ time.

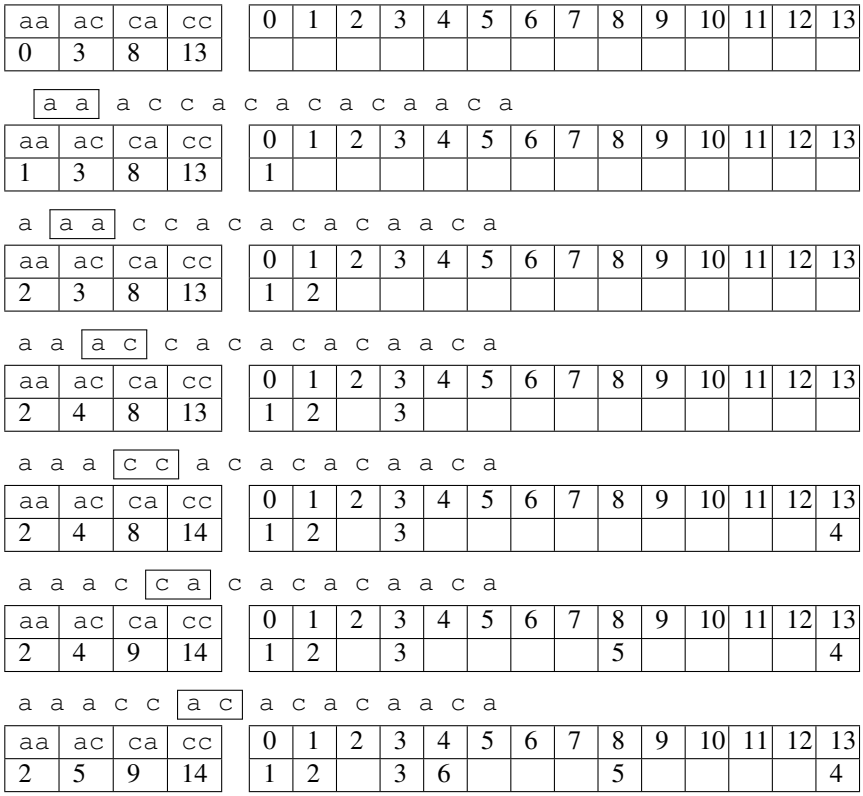
Example 48 Let $\mathcal{A} = \{\mathtt{a}, \mathtt{c}\}$, $q = 2$ and $w = \mathtt{aaaccacacacaaca}$. So $n = |w| = 15$. In the first step, a scan over w delivers the q-gram profile for w, see the first three columns in the following table.

q-gram	code c	$\lvert h_w(c)\rvert$	$P(c)$
aa	0	3	0
ac	1	5	3
ca	2	5	8
cc	3	1	13

From this the partial sums are computed, see column 4. The array H to insert the start positions of the q-grams into is of length

$$\begin{aligned} n - q + 1 = 15 - 2 + 1 = 14 = 13 + 1 = P(3) + |h_w(3)| \\ = P(r^q - 1) + |h_w(r^q - 1)| \end{aligned}$$

We again scan w from left to right. The following illustration shows how table P (left side, indexed by q-grams instead of integer codes) and table H (right side) are updated while w is scanned from left to right.



a a a c c a [c a] c a c a a c a

aa	ac	ca	cc
2	5	10	14

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2		3	6				5	7				4

a a a c c a c [a c] a c a a c a

aa	ac	ca	cc
2	6	10	14

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2		3	6	8			5	7				4

a a a c c a c a [c a] c a a c a

aa	ac	ca	cc
2	6	11	14

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2		3	6	8			5	7	9			4

a a a c c a c a c [a c] a a c a

aa	ac	ca	cc
2	7	11	14

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2		3	6	8	10		5	7	9			4

a a a c c a c a c a [c a] a c a

aa	ac	ca	cc
2	7	12	14

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2		3	6	8	10		5	7	9	11		4

a a a c c a c a c a c [a a] c a

aa	ac	ca	cc
3	7	12	14

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2	12	3	6	8	10		5	7	9	11		4

a a a c c a c a c a c a [a c] a

aa	ac	ca	cc
3	8	12	14

0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2	12	3	6	8	10	13	5	7	9	11		4

a a a c c a c a c a c a a [c a]

aa	ac	ca	cc
3	8	13	14

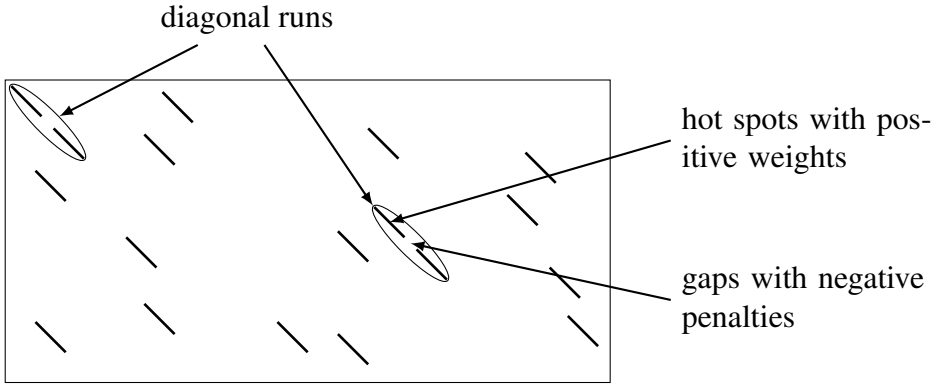
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2	12	3	6	8	10	13	5	7	9	11	14	4

From the final values in table P we can, for each integer code c, deduce the range in H where the elements in $h_w(c)$ are stored:

q-gram	code c	$h_w(c)$	subarray of H representing $h_w(c)$
aa	0	$\{1, 2, 12\}$	$H[0, \ldots, 2]$
ac	1	$\{3, 6, 8, 10, 13\}$	$H[3, \ldots, 7]$
ca	2	$\{5, 7, 9, 11, 14\}$	$H[8, \ldots, 12]$
cc	3	$\{13\}$	$H[13, \ldots, 13]$

The total number of start positions in $h_w(c)$ enumerated in line 9 of Algorithm 6, is the same as the number of common q-grams in u and w, which is $\sum_{d=-m}^{n} \mathit{hotsp}(u, w, d)$. Thus the running time of Algorithm 6 is clearly $O(r^q + m + n + \sum_{d=-m}^{n} \mathit{hotsp}(u, w, d))$ for one database sequence u. That is, the more common q-grams sequences w and u contain, the more time the algorithm requires. So the algorithm does not waste time on database sequences, which are filtered out, as they have a too small fasta-score.

Figure 31: Diagonal runs in the fasta algorithm



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-distance-matrix (without computing the matrix, of course). Diagonal runs are hot spots appearing on the same diagonal, with small gaps in between, see Figure 31 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 r by the upper left corner $(\ell_1(r), \ell_2(r))$ and the lower right corner $(h_1(r), h_2(r))$. The nodes for diagonal runs r and r' are connected if $h_1(r) < \ell_1(r')$ and $h_2(r) < \ell_2(r')$, see Figure 32 for an illustration. The edges get a negative weight. The graph is obviously acyclic and therefore we can efficiently compute a path of maximal total weight. In the lecture 'Genome Informatics' we will have a closer look at how these paths are efficiently computed.

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 33 for illustration. If the score of the optimal global alignment achieves some minimum threshold, then it is reported as a local alignment of the sequence pair in which it appears.

Figure 32: Diagonal runs (in the boxes) are connected by edges with negative weight.

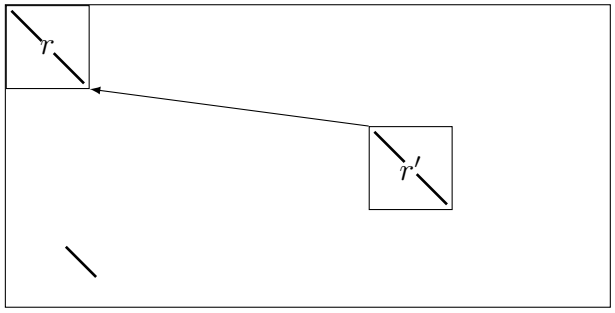


Figure 33: Optimal paths consisting of diagonal runs define a pair of substrings of w and u.

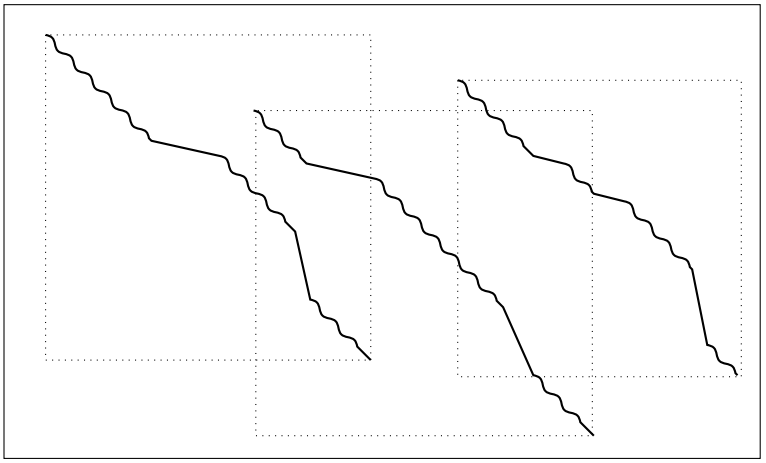


Table 8: Average running time of each stage of the BLAST algorithm according to the evaluation of [CWC04].

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%

4.2 The program BLAST

BLAST is the most popular program to perform sequence database searches. The name is an acronym for *Basic local alignment search tool*. As in [AGM+90], we will describe BLAST 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 sensitive when searching sequence databases for local alignments. The steps progressively reduce the search space, but each is more fine-grain and takes longer to process each sequence. Table 8 shows the average time spent performing each stage of the algorithm.

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 of blast hits does not allow for insertions and deletions. Again we suppose a query sequence w and an arbitrary but fixed database sequence u.

Definition 31 Let $q \in \mathbb{N}$ and $k \geq 0$ be a threshold. A blast-hit is a pair (i, j) of indices such that $score_\sigma(\underbrace{u[i \ldots i+q-1]}_{q\text{-gram in } u}, \underbrace{w[j \ldots j+q-1]}_{q\text{-gram in } w}) \geq k$. □

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

(1) As in Fasta, the query sequence is preprocessed. While the preprocessing in Fasta only collects information for determining exact matches, the BLAST preprocessing takes into consideration non-exact matches. This is achieved by constructing the k-environment $Env_k(w)$

for the query sequence w. This is defined as follows:

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

$Env_k(w)$ thus represents for each position in the query w the q-grams s achieving a score of k or greater when compared to the q-gram at that position. As s can be any q-gram, the environment does not depend on the database sequences.

Example 49 Consider the alphabet $\mathcal{A} = \{\mathtt{a}, \mathtt{c}\}$ and the unit score function σ which assigns score 2 to any match and -1 to any mismatch. Suppose $k = 3$, $q = 3$ and consider the q-gram $x = aca$ contained in some query sequence at say position j. Then one would enumerate all $|\mathcal{A}|^q = 2^3 = 8$ q-words $s \in \{\mathtt{aaa}, \mathtt{aac}, \mathtt{aca}, \mathtt{acc}, \mathtt{caa}, \mathtt{cac}, \mathtt{cca}, \mathtt{ccc}\}$ and verify $score_\sigma(s, x) \geq k$:

	a	c	a	$\sum$
aaa	$+2$	-1	$+2$	**3**
aac	$+2$	-1	-1	0
aca	$+2$	$+2$	$+2$	**6**
acc	$+2$	$+2$	-1	**3**
caa	-1	-1	$+2$	0
cac	-1	-1	-1	-3
cca	-1	$+2$	$+2$	**3**
ccc	-1	$+2$	-1	0

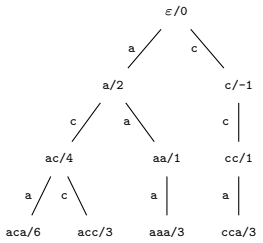
So the pairs $(\mathtt{aaa}, j)$, $(\mathtt{aca}, j)$, $(\mathtt{acc}, j)$, $(\mathtt{cca}, j)$ are elements in $Env_k(w)$.

An environment can be computed by first enumerating all q-grams in w. As seen in the previous example, common prefixes of these q-grams lead to the same positional scores. For example, for all q-grams beginning with a, we obtain a first score $+2$ when comparing them to aca. So it makes sense to only evaluate this score once and group the prefixes with a common first character in a subtree below some edge marked with that character. If we continue this further down, up to the last position, we obtain a trie, representing q-words. 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.

On each trie-node α representing the sequence y of length q' for some $q' \leq q$, one maintains $score_\sigma(y, x[1 \ldots q'])$ where x is the current q-gram of w. Each node of the trie at depth q with a score $\geq k$ represents a sequence from the k-environment of w, see Example 50 for an illustration. 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. Details of this approach are subject to an exercise.

Example 50 Consider the alphabet $\mathcal{A} = \{\mathtt{a}, \mathtt{c}\}$ and the unit score function σ which assigns score 2 to any match and -1 to any mismatch. Suppose $k = 3$, $q = 3$ and consider the q-gram

$x =$ `aca` contained in some query sequence, say at position j. The k-environment for `aca` is computed using this trie:



Each node is labeled by the string it represents and the score achieved when comparing the string to the corresponding prefix of `aca`; the leaves at depth q represent the strings in the environment. The following table shows the vector V and *Pos*(s) which will be explained later.

	aaa	aac	aca	acc	caa	cac	cca	ccc
$V =$ [	1	0	1	1	0	0	1	0]
Pos	j		j	j			j	

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

(3) $Env_k(w)$ is efficiently 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 *Pos*$(s) = \{j \mid (s, j) \in Env_k(w)\}$.

Example 50 shows V and *Pos*(s) for a q-gram $x = aca$. To represent the sets *Pos*(s) and to efficiently access them, we can apply techniques similar to those used for representing the sets $h_w(c)$ in the Fasta-algorithm.

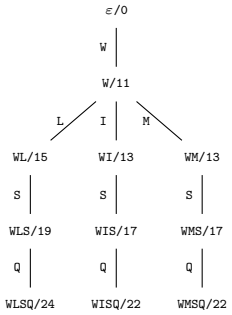
(4) Each database sequence u is then processed by shifting a window of length q over it. Suppose that $i + q$ characters have been processed and let s be the current substring of length q in the window, i.e. $s = u[i \dots i+q-1]$. Then we compute $\overline{s}$ in constant amortized time and look up if $V[\overline{s}]$ is 1. If this is the case, then by definition of V, there is some j, such that $(s, j) \in Env_k(w)$. These positions j can be found in the set $j \in$ *Pos*(s) which can efficiently be enumerated. For all $j \in$ *Pos*(s), (i, j) is a blast hit. This is processed further in the next steps.

Note that the search in the database sequence is entirely based on looking up values in tables V and Pos according to integer codes. No comparison of symbols according to a score function is necessary, as this has all been done in the preprocessing step. This is one of the main reasons why blast is so fast in determining the blast hits (but still takes a large share of the entire runtime of BLAST, see Table 8).

The first three steps (1)–(3) only depend on the query sequence w. Hence they only have to be performed once, before processing the database sequences in step (4).

Example 51 Suppose that σ is the BLOSUM62-score function, see Table 5. Let $q = 4$, $k = 22$ and

Figure 34: The trie implicitly generated by the algorithm constructing $Env_k(\texttt{WLSQ})$ for $k = 22$ and $q = 4$. Each node of depth $q' \leq q$ is labeled by the q'-gram it represents and the sum of the scores when comparing it with $\texttt{WLSQ}[1 \ldots q']$. The leaves are of depth q and in the environment they appear with the position 8, which is the start position of WLSQ in the query sequence w.



$$w = \underset{1\,2\,3\,4\,5\,6\,7\,8\,9\,0\,1}{\texttt{MGHLPLAWLSQ}}$$

Then we obtain the following k-environment $Env_k(w)$, in which the score of each element is given as a subscript of the pair. Note that LPLA at position 4 is the only q-gram in w not generating an element in $Env_k(w)$.

$$\begin{array}{llll}
(\texttt{MGHL}, 1)_{23} & (\texttt{GHLP}, 2)_{25} & (\texttt{GHVP}, 2)_{22} & (\texttt{GHIP}, 2)_{23} \\
(\texttt{GHMP}, 2)_{23} & (\texttt{HLPL}, 3)_{23} & (\texttt{PLVW}, 5)_{22} & (\texttt{PLAW}, 5)_{26} \\
(\texttt{PLGW}, 5)_{22} & (\texttt{PLSW}, 5)_{23} & (\texttt{PLTW}, 5)_{22} & (\texttt{PLCW}, 5)_{22} \\
(\texttt{PVAW}, 5)_{23} & (\texttt{PIAW}, 5)_{24} & (\texttt{PFAW}, 5)_{22} & (\texttt{PMAW}, 5)_{24} \\
(\texttt{LAWL}, 6)_{23} & (\texttt{AWLS}, 7)_{23} & (\texttt{WLSQ}, 8)_{24} & (\texttt{WISQ}, 8)_{22} \\
(\texttt{WMSQ}, 8)_{22} & & &
\end{array}$$

Fig. 34 shows the implicit trie generated when constructing $Env_k(\texttt{WLSQ})$.

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 blast 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. In the newer version of BLAST (BLAST 2.0 and later), an intermediate step was introduced, which is described next.

4.2.2 Stage 2: Identify pairs BLAST hits on the same diagonal

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 35. Consider two hits (i, j) and (i', j') on the same diagonal d, i.e. $j - i = d = j' - i'$ and suppose that $i < i'$. Hence $i + h = i'$ for some $h > 0$. As the blast hits are computed in increasing order of the positions in the database sequence, (i, j) is computed before (i', j'). From the assumption we can conclude $j' = i' + d = i + h + d = i + d + h = j + h$. Thus, with respect to the same diagonal, blast hits are computed in increasing order of the positions in both sequences and with the same distance h between i and i' as well as j and j'.

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 $i' - f(d) \leq \delta$, where δ is the *maximal distance on diagonal*-parameter. If $i' - f(d) \leq \delta$ holds, then the distance of the two blast hits

$$(\underbrace{f(d)}_{i}, \underbrace{d + f(d)}_{j=d+i}) \text{ and } (i', j') \text{ on diagonal } d$$

is at most δ and one has found a pair of relevant blast hits which is further processed. In any case (whether $f(d)$ is defined or not), one updates $f(d) \leftarrow i'$.

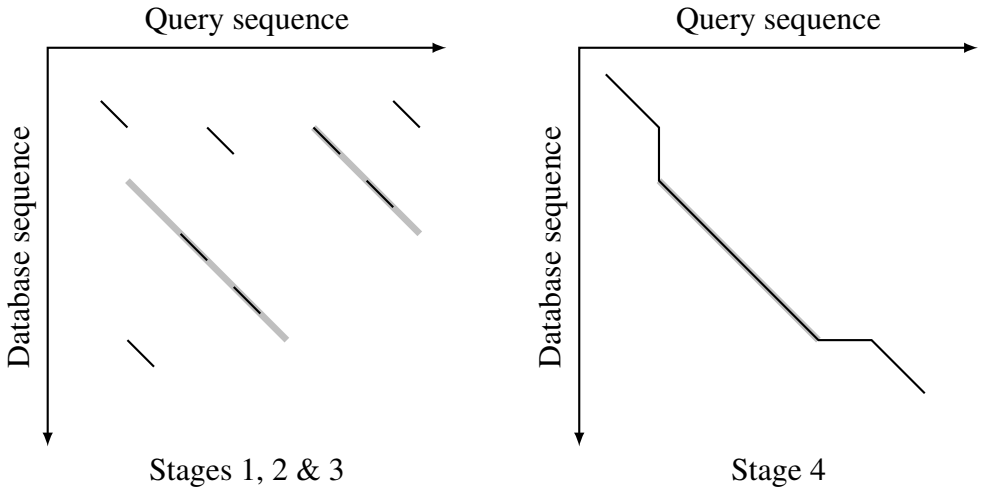
4.2.3 Stage 3: Ungapped extensions

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.

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 $(u[i - \ell], w[j - \ell])$, and $(u[i + r], w[j + r])$, for $1 \leq \ell \leq \min\{i - 1, j - 1\}$ and $0 \leq r \leq \min\{m - i, n - j\}$, 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 $X_{\mathsf{max}} - X_d$ is reached, the extension is stopped, see Algorithm 7 show the pseudocode of 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. 36). 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.

Figure 35: 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 grey 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 high-scoring gapped alignment.

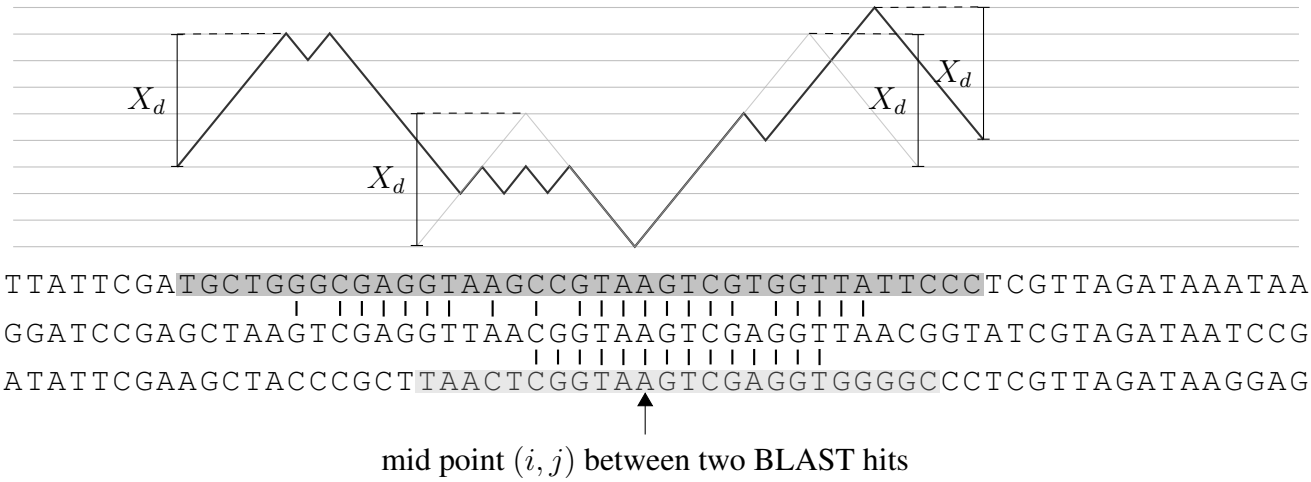


Algorithm 7 (BLAST hit extension)

A right-extension function starting at coordinates (i, j) on the middle between 2 blast hits on the same diagonal for the database seq. u and the query seq. w. σ is the score function and $X_d > 0$ is the drop-off param.

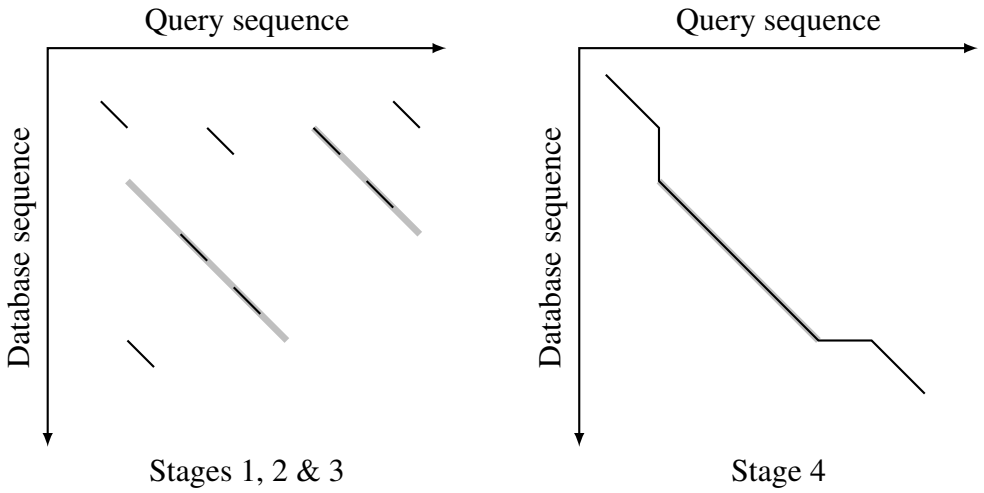
function $extendhitright(u, w, \sigma, X_d, i, j)$
$(X_{\mathsf{max}}, accum\_score, r) \leftarrow (0, 0, 0)$
while $i + r \leq |u|$ **and** $j + r \leq |w|$ **do**
    $accum\_score \leftarrow accum\_score + \sigma(u[i + r], w[j + r])$
    if $accum\_score < X_{\mathsf{max}} - X_d$ **then**
        break
    end if
    if $accum\_score > X_{\mathsf{max}}$ **then**
        $X_{\mathsf{max}} \leftarrow accum\_score$
    end if
    $r \leftarrow r + 1$
end while
return $r - 1$

Figure 36: Illustration of the blast hit extension strategy for a midpoint marked by the arrow. Match score is 1 and mismatch score is -1. At the bottom, two queries are shown above and below the database sequence in the middle, respectively. At the top, the accumulated score obtained when comparing the dark grey and light grey part of the query sequence against the corresponding part of the database sequence is shown as a thick and a thin line, respectively. The extension stops when the accumulated score drops below $X_{\mathsf{max}} - X_d$, which happens when the boundaries of the query substring in the grey boxes is reached.



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 following illustration (right side) illustrates an example of a gapped extension. The single, high-scoring ungapped extension identified in Stage 3 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, 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 3.8.2).

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 fifth 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.

4.2.5 Stage 5: Collect traceback information and display alignments

BLAST uses Karlin-Altschul statistics [AG96, KA90] to report the statistical significance of each gapped local alignment as an E-value. The E-value is the number of local alignments with at least the same score that one expects when comparing a random query sequence with typical amino-acid composition against a random database sequence. Of course, the length of the query sequence and the size of the database sequence are taken into consideration. There are three steps to determine the statistical significance of a gapped local alignment:

1. A nominal score S is determined for each local alignment based on a scoring matrix (BLOSUM [HH93] or PAM [DSO78]) and gap scores.

2. The nominal score is converted to a normalized score

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

   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 $Q \approx mn$ is the size of the search space, where m and n are the lengths of the query and the database sequence, respectively.

The resulting value of E is reported to the user. As the equations above are invertible, one can determine the minimum nominal 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

Functional biological sequences typically come in families, and many of the most powerful sequence analysis methods are based on identifying the relationship of an individual sequence to a sequence family. Sequences in a family usually diverged from each other in their primary sequence during evolution, while maintaining the same or a related function. As a consequence, identifying that a sequence belongs to the family, often allows inference about its function. Capturing all signals common to the members of a sequence family is best done by a statistical approach based on a multiple alignment (see next section) of the sequences in the family.

Example 52 Consider an example of a family of 5 sequences GTAT, GGCG, TTCG, GTCA, and GTTA which can be represented by the following multiple alignment:

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

For each of the four positions in the sequences we want to capture the conservation of specific symbols by determining their probability of occurrence. This leads to a PSSM:

A	0.0	0.0	0.2	0.4
C	0.0	0.0	0.6	0.0
G	0.8	0.2	0.0	0.4
T	0.2	0.8	0.2	0.2

Now each DNA-sequence of length 4 can be scored by summing up the scores of the bases at the corresponding positions, e.g. GTGA gets score $0.8 + 0.8 + 0.0 + 0.4 = 2.0$.

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 $M : \mathcal{A} \times \{1, \ldots, m\} \to \mathbb{R}$, where $\mathcal{A}$ is a finite alphabet and m is the length of M. Usually M is represented by an $|\mathcal{A}| \times m$-matrix, see Table 9 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 $score(w, M) = \sum_{h=1}^{m} M(w[h], h)$ 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, $1 \leq j \leq n-m+1$ in S such that $score(S[j \ldots j+m-1], M) \geq th$.

Example 53 Reconsider the PSSM of Example 52:

A	0.0	0.0	0.2	0.4
C	0.0	0.0	0.6	0.0
G	0.8	0.2	0.0	0.4
T	0.2	0.8	0.2	0.2

Now let $S =$ AGATCCTAACG and $th = 1.2$. Then we obtain two solutions to the PSSM-searching problems:

position	substring	score
3	ATCC	$0.0 + 0.8 + 0.6 + 0.0 = 1.4$
6	CTAA	$0.0 + 0.8 + 0.2 + 0.4 = 1.4$

A simple algorithm for the PSSM searching problem slides along the sequence and computes $score(w, M)$ for each $w = S[j \dots j+m-1]$, j, $1 \leq j \leq n-m+1$. The running time of this algorithm is $O(mn)$. This algorithm 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. A similar technique was used when computing the k-environment for Blast.

Example 54 Reconsider the PSSM of Example 52:

A	0.0	0.0	0.2	0.4
C	0.0	0.0	0.6	0.0
G	0.8	0.2	0.0	0.4
T	0.2	0.8	0.2	0.2

Let $S =$ AGATCCTAACG, $th = 1.2$ and consider the substrings CCTA and AACG at positions 5 and 8, respectively. For the first two characters in these substrings we obtain the score 0. The maximum score we can achieve for the last two characters is $0.6 + 0.4$. So once we have read the first two characters CC ad AA, we know that we can achieve a score of at most 1 for the entire string. This is below the threshold. So we can stop after the first two characters of these substrings.

To explain the lookahead scoring method, define

$$pfxscore_d(w, M) = \sum_{h=1}^{d} M(w[h], h),$$
$$\mathsf{max}_d = \max\{M(a, d) \mid a \in \mathcal{A}\},$$
$$\sigma_d = \sum_{h=d+1}^{m} \mathsf{max}_h$$

for any d, $1 \leq d \leq m$.

That is, $pfxscore_d(w, M)$ is the score for the prefix $w[1 \dots d]$ of length d. Moreover, 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, for $1 \leq d \leq m$.

The next lemma reveals an important property of prefix scores:

Lemma 7 The following statements are equivalent:

(1) $pfxscore_d(w, M) \geq th_d$ for all d, $1 \leq d \leq m$.

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

Proof: (1)⇒(2): Suppose that (1) holds. Consider the special case for $d = m$. Then $\sigma_m = \sum_{h=m+1}^{m} \mathsf{max}_h = 0$ and

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

(2)⇒(1): Suppose that (2) holds. Let d, $1 \leq d \leq m$ be fixed but arbitrary. Then

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

Hence $score(w, M) \geq th$ implies $pfxscore_d(w, M) + \sum_{h=d+1}^{m} M(w[h], h) \geq th$.
Since $M(w[h], h) \leq \mathsf{max}_h$ for all h, $1 \leq h \leq m$, we conclude

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

and hence

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

Lemma 7 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

Algorithm 8 (Lookahead scoring)

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

```
1: compute th_d for 1 ≤ d ≤ m
2: for all j ← 1 upto n − m + 1 do
3:     score ← 0
4:     for all d ← 1 upto m do
5:         score ← score + M(S[j + d − 1], d)
6:         if score < th_d then
7:             break
8:         end if
9:     end for
10:    if score ≥ th then
11:        print(match at position j with score score)
12:    end if
13: end for
```

intermediate threshold th_d is achieved. If not, the computation can be stopped. See Algorithm 8 for pseudo-code and Table 9 for an example applying the algorithm. The lookahead scoring algorithm runs in $O(kn)$ time, where k is the average number of PSSM-positions per sequence start position actually evaluated. In the worst case, k is 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.

Table 9: PSSM of length $m = 10$ of a zinc-finger motif. Let $th = 400$ be the score threshold. Then only substrings beginning with `C` or `V` can match the PSSM, since all other amino acids score below the intermediate threshold $th_1 = -7$. That is, lookahead scoring will skip over all substrings beginning with amino acids different from `C` and `V`, as $M(\texttt{V}, 1) = 16 \geq th_1 = -7$ and $M(\texttt{C}, 1) = 92 \geq th_1 = -7$ and $M(x, 1) < th_1 = -7$ for all other aminoacids x.

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

4.4 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 determine an E-value for a local alignment in which the number of mismatches is smaller than some threshold. We exclude alignments with indels i.e. focus on ungapped alignments. This is because gapped alignments are more difficult to handle statistically, using e.g. the Karlin-Altschul Statistics which we have considered as a black box at the end of the section on Blast. In this section, the cost of the local alignment is thus measured in terms of the hamming distance of the aligned pair of substrings. We describe all details of an approach to calculate E-values. For two strings x and y of equal length, define

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

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

Now consider two sequences u and v of length m and n, respectively. We want to determine pairs of substrings in u and v allowing for a maximum number $k \geq 0$ of mismatches in these substrings. So k is a *mismatch threshold*.

A *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.
- $1 \leq i \leq m - \ell + 1$ and $1 \leq j \leq n - \ell + 1$, 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, both of length ℓ.
- $\mathcal{H}(u[i \ldots i + \ell - 1], v[j \ldots j + \ell - 1]) \leq k$, 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 $u[i + \ell] \neq v[j + \ell]$. The following figure illustrates these notions:

$$u = \underbrace{\overset{i}{a}\underbrace{\qquad\qquad}_{u[i\ldots i+\ell-1]}\overset{i+\ell-1}{b}}_{=} \qquad \underbrace{\overset{j}{c}\underbrace{\qquad\qquad}_{v[j\ldots j+\ell-1]}\overset{j+\ell-1}{d}}_{=} = v$$

$$\text{left maximal} \iff u[i-1] = a \neq c = v[j-1]$$
$$\text{right maximal} \iff u[i+\ell] = b \neq d = v[j+\ell]$$

(ℓ, i, j) is *maximal* if it is left-maximal and right-maximal.

To assess the significance of a k-LMA, one determines its *E-value*, which is the size of the following set:

$$\{(\ell', i', j') \mid (\ell', i', j') \text{ is } k'\text{-LMA in } u' \text{ and } v', \ell \leq \ell', k' \leq k\}$$

where u' and v' are random sequences of length $|u'| = |u|$, $|v'| = |v|$.

That is, the *E*-value is the number of local alignments of the same length or longer and with the same or a fewer number of mismatches, that occur one would expect to find between two random sequences of the same length as u and v.

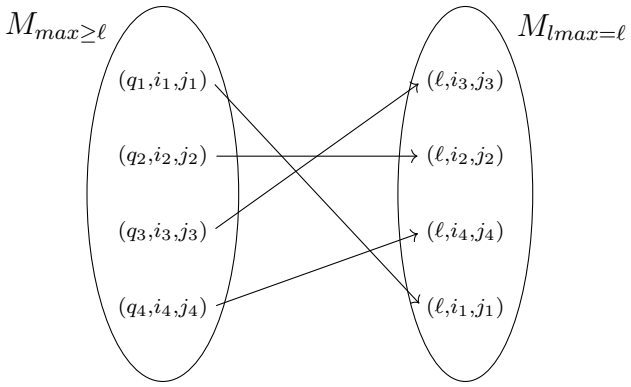
As a model of random sequences, we assume the uniform Bernoulli model. That is, we assume that at each position of a random sequence, each of the r characters in $\mathcal{A}$ occurs with the same probability $p = \frac{1}{r}$.

4.4.1 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 therefore use the term "maximal exact match" instead of k-LMA. We first show an important property of maximal exact matches of length $\geq \ell$.

Lemma 8 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, see the following illustration:



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 (pronounce as gamma). To show that γ is bijective, one first needs to show that 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'). In terms of the illustration above, this means that two different arcs never point to the same element in the set on the right. 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 $\gamma((q,i,j)) = (\ell,i,j) \neq (\ell,i',j') = \gamma((q',i',j'))$. Next we have to show that γ is surjective, i.e. for each left-maximal match (ℓ,i,j) there must be some maximal match (q,i,j) such that $\gamma((q,i,j)) = (\ell,i,j)$. In terms of the illustration above, this means that for any element in the set on the right, there is at least one arc pointing to it. But this is of course the case, 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.

We use the following notions and notations:

- $\mathbb{E}[\#\text{ events}]$ is the expected number of events
- in our case the events are maximal exact matches of length $\geq \ell$ or left-maximal matches of length $= \ell$.
- the expected number of events can in our case be determined as follows:
  - consider each possible position where an event can occur
  - determine the probability that the event occurs at the considered position
- the probability of an event occurring at position p is denoted by $Pr[\text{event occurs at position } p]$
- So $\mathbb{E}[\#\text{ events}] = \sum_{\text{position } p} Pr[\text{event occurs at position } p]$

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:

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

Note that the second equality is due to the fact that we can count the number of left-maximal exact 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

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

Example 55 shows some E-values of maximal exact matches of length $\geq \ell$.

4.4.2 Allowing for mismatches

E-values for k-LMAs, $k > 0$, 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 first have to compute the number of choices of k positions with mismatches from ℓ possible positions in the sequences. We can apply a standard formula from combinatorics:

Given a set X of size ℓ, there are exactly

$$\binom{\ell}{k} = \frac{\ell!}{k!(\ell-k)!} \tag{4.1}$$

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

$$\prod_{q=1}^{k}(\ell - (q-1)) = \ell \cdot (\ell - 1) \cdot \cdots \cdot (\ell - (k-1)) = \frac{\ell!}{(\ell - k)!} \tag{4.2}$$

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 (4.2) by $k!$ to obtain (4.1).

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, $p^{\ell-k}$ is the probability that there is a match in all remaining $\ell - k$ 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:

$$
\begin{aligned}
&\mathbb{E}[\#\text{ of maximal } \leq k\text{-LMAs of length } \geq \ell] \\
&= \sum_{k'=0}^{k} \sum_{\ell'=\ell}^{\min(m,n)} \sum_{i\in[1,m],j\in[1,n]} Pr[(\ell', i, j) \text{ is a maximal } k'\text{-LMA}] \\
&= \sum_{k'=0}^{k} \sum_{\ell'=\ell}^{\min(m,n)} \sum_{i\in[1,m],j\in[1,n]} Pr[\mathcal{H}(u[i \ldots i+\ell'-1], \\
&\qquad\qquad v[j \ldots j+\ell'-1]) = k' \text{ and} \\
&\qquad\qquad u[i-1] \neq v[j-1] \text{ and } u[i+\ell'] \neq v[j+\ell']] \\
&= \sum_{k'=0}^{k} \sum_{\ell'=\ell}^{\min(m,n)} \sum_{i\in[1,m],j\in[1,n]} Pr[\mathcal{H}(u[i \ldots i+\ell'-1], v[j \ldots j+\ell'-1]) = k'] \cdot \\
&\qquad\qquad Pr[u[i-1] \neq v[j-1]] \cdot \\
&\qquad\qquad Pr[u[i+\ell'] \neq v[j+\ell']] \\
&= \sum_{k'=0}^{k} \sum_{\ell'=\ell}^{\min(m,n)} \sum_{i\in[1,m],j\in[1,n]} \binom{\ell'}{k'} p^{\ell'-k'}(1-p)^{k'}(1-p)(1-p) \\
&= \sum_{i\in[1,m],j\in[1,n]} \sum_{k'=0}^{k} \sum_{\ell'=\ell}^{\min(m,n)} \binom{\ell'}{k'} p^{\ell'-k'}(1-p)^{k'+2} \\
&= m\, n \sum_{k'=0}^{k} \sum_{\ell'=\ell}^{\min(m,n)} \binom{\ell'}{k'} p^{\ell'-k'}(1-p)^{k'+2}.
\end{aligned}
$$

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 55 Consider two DNA 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	$1.3 \cdot 10^{2}$	$6.3 \cdot 10^{3}$	$1.4 \cdot 10^{5}$	$2.0 \cdot 10^{6}$
32	$3.0 \cdot 10^{-8}$	$2.9 \cdot 10^{-6}$	$1.4 \cdot 10^{-4}$	$4.1 \cdot 10^{-3}$
64	$1.7 \cdot 10^{-27}$	$3.2 \cdot 10^{-25}$	$3.0 \cdot 10^{-23}$	$1.9 \cdot 10^{-21}$
128	$4.9 \cdot 10^{-66}$	$1.9 \cdot 10^{-63}$	$3.6 \cdot 10^{-61}$	$4.5 \cdot 10^{-59}$

5 A linear space alignment algorithm

5.1 Motivation for reducing the space requirement

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 $(m+1) \times (n+1)$-table E_δ from which the alignments are computed by a traceback method. It is difficult to improve the running time of this algorithm, if the cost function is arbitrary, independent of whether linear or affine gap costs are used. In this section we will improve the space requirement by deriving an algorithm that only takes $O(m + n)$ space to compute a single optimal alignment of u and v. For ease of presentation we will restrict to linear gap costs, although the method can be generalized to affine gap costs. In most cases, it suffices to compute only a single alignment, and so the linear space algorithm is very important in practice.

5.2 Recalling the linear space distance only method

Recall that matrix E_δ satisfies

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

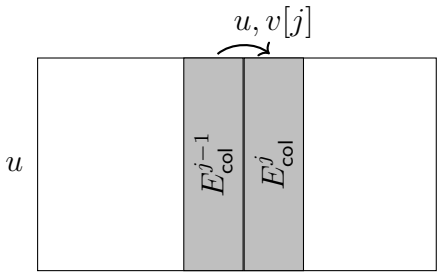
for all i, j, $0 \leq i \leq m$ and $0 \leq j \leq n$. It can be computed according to the following recurrence:

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

For each j, $0 \leq j \leq n$, let

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

denote the jth column of table E_δ. E^j_{col} is a vector of $m+1$ values. We will later refer to these 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 $v[j]$, as illustrated here:



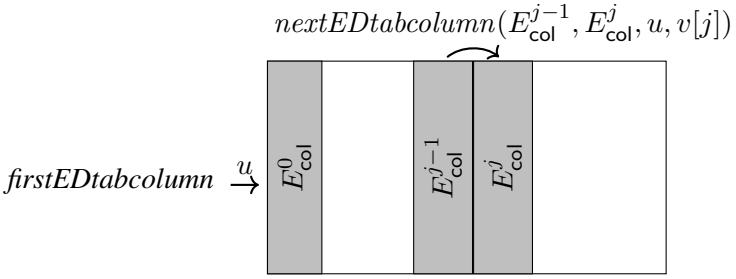
This follows from a simple analysis of the data dependencies of the fundamental recurrences for E_δ. Hence one can compute the sequence of column vectors

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

such that only two columns, the previous and the one being computed, need be stored at any time. Up until now in this section we have recalled what we already knew about the computation of matrix E_δ. We have done this more explicit by introducing a notation for the columns vectors of matrix E_δ. The computation of the sequence of column vectors can be described by a function *nextEDtabcolumn* which takes four parameters: the vectors storing the current and 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. Both functions can be used in a function *evaluateallEDtabcolumns* which takes an $(m+1)$-vector E and computes the last column E^m_{col} of table E_δ, see Algorithm 9 and the following illustration:

Algorithm 9 (Computation of first and next distance column)

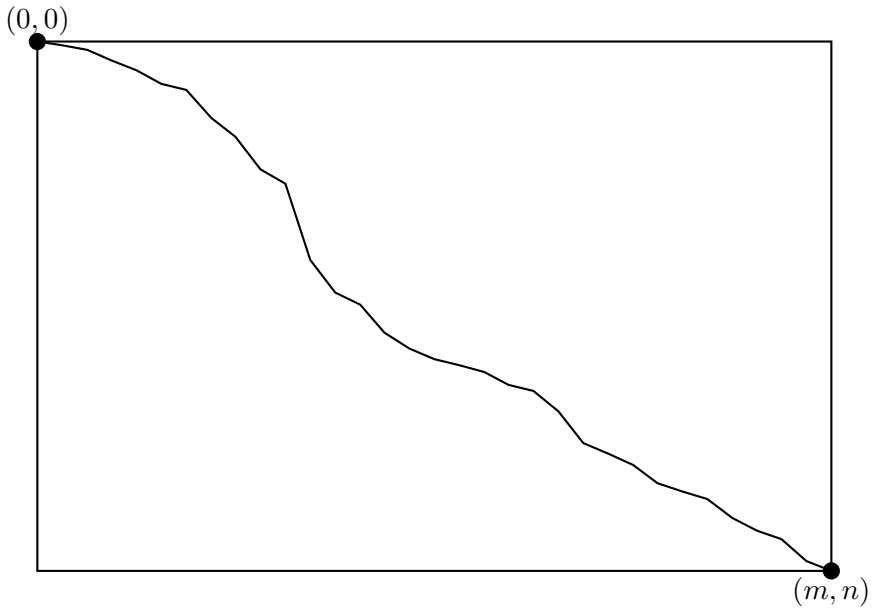
1: **function** *firstEDtabcolumn*(E, u)
2: $E(0) \leftarrow 0$
3: **for** $i \leftarrow 1$ **upto** $|u|$ **do** $E(i) \leftarrow E(i-1) + \delta(u[i] \rightarrow \varepsilon)$
4: **end for**
5: **end function**
6: **function** *nextEDtabcolumn*(E, E', u, b)
7: $E'(0) \leftarrow E(0) + \delta(\varepsilon \rightarrow b)$
8: **for** $i \leftarrow 1$ **upto** $|u|$ **do** $E'(i) \leftarrow \min \left\{ \begin{array}{l} E'(i-1) + \delta(u[i] \rightarrow \varepsilon) \\ E(i) + \delta(\varepsilon \rightarrow b) \\ E(i-1) + \delta(u[i] \rightarrow b) \end{array} \right\}$
9: **end for**
10: **end function**
11: **function** *evaluateallEDtabcolumns*(E, u, v)
12: *firstEDtabcolumn*(E, u);**allocate vector** E' of length $|u| + 1$
13: **for** $j \leftarrow 1$ **upto** $|v|$ **do** *nextEDtabcolumn*$(E, E', u, v[j])$; $E \leftarrow E'$;
14: **end for**
15: **end function**



This distance-only algorithm calls $evaluateallEDtabcolumns(E, u, v)$ and delivers E_{col}^{n} in vector E. The cost $E_{\delta}(m, n) = E_{\mathsf{col}}^{n}(m)$ of an optimal alignment is thus computed 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 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 column values. We leave it as an exercise to implement such a function. Such a function does not improve the asymptotic space requirement (it remains $O(\min\{m, n\})$), but simplifies the space management in practice.

Figure 37: An optimal alignment can be represented by a path from node $(0, 0)$ to node (m, n) in the edit-graph $G_\delta(u, v)$.



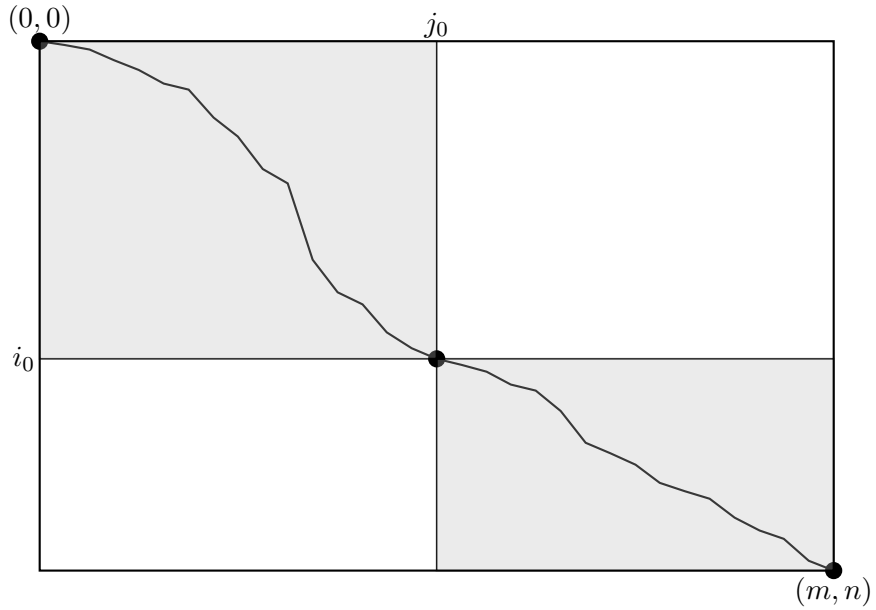
5.3 Applying the divide and concquer strategy

Now consider how to deliver of an optimal alignment using the algorithm of [PAD99]. Some parts of the presentation follow [Mye98]. First recall that the optimal alignment can be considered as a path in the edit-graph $G_\delta(u, v)$ which can be reconstructed by a traceback from node (m, n) to $(0, 0)$. See Figure 37 for an illustration.

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 of the linear-space optimal alignment algorithm is to keep some of the history and apply a divide-and-conquer strategy to obtain a representation of an optimal alignment from this history. The method works as follows: consider the column $j_0 = \lfloor n/2 \rfloor$ and find a node (i_0, j_0) that is on 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). Each recursion step delivers an additional crosspoint besides the two crosspoints $(0, 0)$ and (m, n). In particular, up to recursion level k, $k \leq \log_2 n$, the method has computed $cpc(k) = \sum_{i=1}^{k} 2^{i-1} = 2^k - 1$ crosspoints. E.g. $cpc(1) = 1$, $cpc(2) = 3$, $cpc(\log_2 n) = 2^{\log_2 n} - 1 = n - 1$.

The recursive strategy is illustrated in Figures 38-40.

Figure 38: The divide and conquer strategy begins by determining the middle column $j_0 = \lfloor n/2 \rfloor$ and finding a crosspoint (i_0, j_0) on a minimizing path from node $(0, 0)$ to node (m, n) in the edit-graph $G_\delta(u, v)$. Note that the minimizing path shown has not been computed yet. To find the other crosspoints representing the minimizing path, one needs to apply the same strategy to the two grey rectangles. This means that the sequence pairs $u[1 \ldots i_0]/v[1 \ldots j_0]$ and $u[i_0 + 1 \ldots m]/v[j_0 + 1 \ldots n]$ are both subject to the recursive strategy.

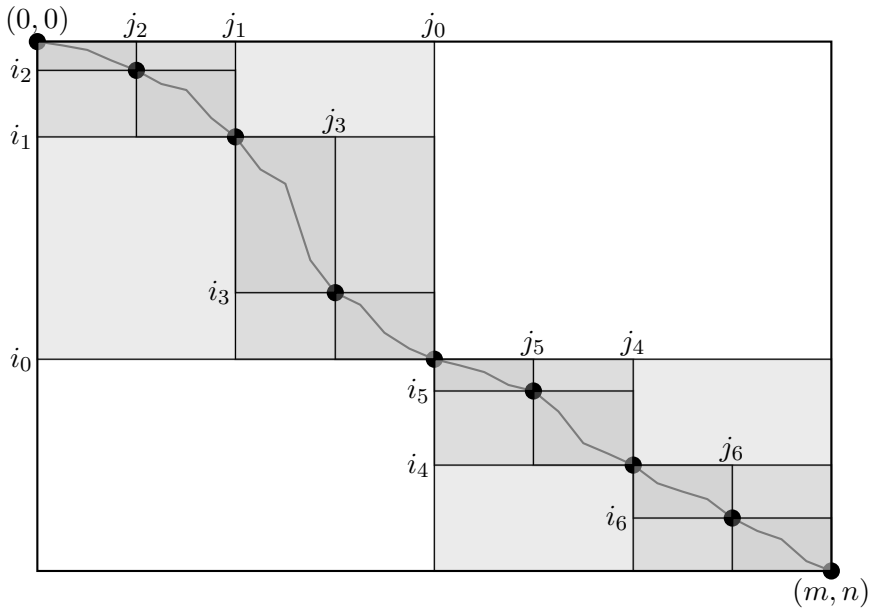


We say that a node $(\_, j_0)$ in the edit-graph $G_\delta(u, v)$ is on the middle of a path from $(0, 0)$ to (m, n) if and only if $j_0 = \lfloor n/2 \rfloor$. The divide-and-conquer strategy then needs to find an index i_0, $0 \leq i_0 \leq m$ such that (i_0, j_0) is on a minimizing path from $(0, 0)$ 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 *row index columns*. For each i, j, $0 \leq i \leq m$, $0 \leq j \leq j_0 - 1$, $R_{j_0}(i, j)$ is undefined. For each j, $j_0 \leq j \leq n$ and each i, $0 \leq i \leq m$, $R_{j_0}(i, j)$ 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). Note that $R_{j_0}(i, j)$ is not uniquely defined, as there can be more than one node in column j_0 from which the minimizing path ending in (i, j) starts. See Figure 41 for an example showing table R_{j_0} and E_δ superimposed. R_{j_0} can be computed using the recurrence of Figure 42. The computation of $R_{j_0}(i, j)$ for all j, $j_0 \leq j \leq n$ follows the same scheme as the computation of table E_δ. In particular, one computes the jth column

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

of R_{j_0} along with E^j_{col}. The corresponding recurrence is shown in Figure 42.

Figure 39: The $1 + 2 + 4 = 7$ crosspoints computed at recursion level 3 of the divide and conquer strategy are shown as bullets. Additionally, $(0, 0)$ and (m, n) are crosspoints. The dark grey rectangles represent the sequence pairs to be considered at the next recursion level.



With the same argumentation as above, one shows that the computation of the R_{j_0}-columns only requires one $(m + 1)$-element vector and two additional scalar values. It is not difficult to derive a function *nextEDtabRtabcolumn* (in analogy to *nextEDtabcolumn*) performing this computation in the right halve of the matrix. This 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. As $R^0_{j_0}$ is undefined in the left halve of the matrix, for $j < j_0$ one only needs to compute E^j_{col} using the previously explained functions *firstEDtabcolumn* and *nextEDtabcolumn*, see the following illustration.

Figure 40: The $1+2+4+8=15$ crosspoints computed at recursion level 4 of the divide and conquer strategy are shown as bullets. Additionally, $(0,0)$ and (m,n) are crosspoints. The column and and row indices are omitted. The dark grey rectangles represent the sequence pairs to be considered at the next recursion level.

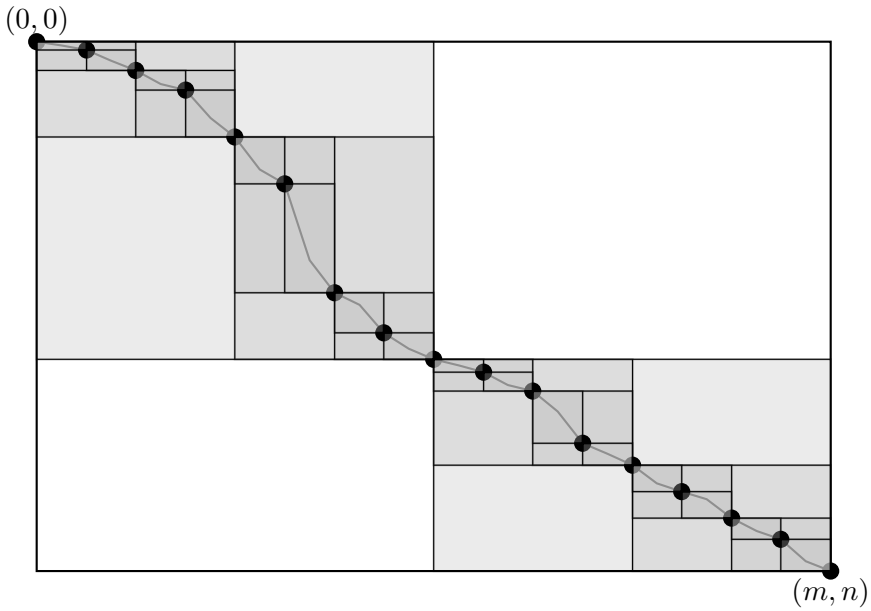


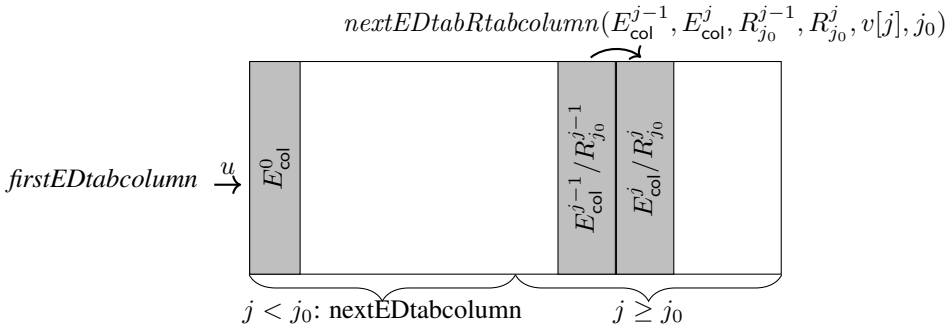
Figure 41: Tables E_δ and R_{j_0} for sequences $u =$ `bcacd`, $v =$ `dbadad`, unit cost function, and $j_0 = \left\lfloor \frac{|v|}{2} \right\rfloor = \lfloor \frac{6}{2} \rfloor = 3$. The two matrices are superimposed into one matrix of pairs $(e, \mathbf{r})$, where e is the distance value and $\mathbf{r}$ is the row index value. Undefined row index values are shown as $\perp$. The first row and column shows the sequence v and u, respectively. The second row and second column shows the column indices and row indices, respectively. As $i_0 = R_{j_0}(m,n) = 3$, $(i_0, j_0) = (3,3)$ (shown in blue) is a crosspoint.

					j_0			
			d	b	a	d	a	d
	i/j	0	1	2	3	4	5	6
	0	(0,⊥)	(1,⊥)	(2,⊥)	(3,**0**)	(4,**0**)	(5,**0**)	(6,**0**)
b	1	(1,⊥)	(1,⊥)	(1,⊥)	(2,**1**)	(3,**1**)	(4,**1**)	(5,**1**)
c	2	(2,⊥)	(2,⊥)	(2,⊥)	(2,**2**)	(3,**2**)	(4,**2**)	(5,**2**)
a	3	(3,⊥)	(3,⊥)	(3,⊥)	(2,**3**)	(3,**3**)	(3,**2**)	(4,**2**)
c	4	(4,⊥)	(4,⊥)	(4,⊥)	(3,**4**)	(3,**3**)	(4,**3**)	(4,**2**)
d	5	(5,⊥)	(4,⊥)	(5,⊥)	(4,**5**)	(3,**4**)	(4,**4**)	(4,**3**)

Figure 42: The recurrence for table R_{j_0}. The values depend on column E^{j-1}_{col} and E^{j}_{col}. Line 3 handles the case where the insertion edge is minimizing. Line 4 handles the case where the replacement edge is minimizing. Line 5 handles the case where the deletion edge is minimizing. If $j > j_0$, then at least one of the three cases applies, so the last *else if* could be replaced by *otherwise*. As usual, with a distance value $E^{j}_{\mathsf{col}}(i)$ we can store which of the incoming edges is minimizing, as to remove the dependency on other entries in E^{j-1}_{col} and E^{j}_{col}.

$$R_{j_0}(i,j) = \begin{cases} \mathit{undefined} & \text{if } 0 \leq j \leq j_0 - 1 \\ i & \text{else if } j = j_0 \\ R_{j_0}(i, j-1) & \text{else if } E^{j}_{\mathsf{col}}(i) = E^{j-1}_{\mathsf{col}}(i) + \delta(\varepsilon \to v[j]) \\ R_{j_0}(i-1, j-1) & \text{else if } E^{j}_{\mathsf{col}}(i) = E^{j-1}_{\mathsf{col}}(i-1) + \delta(u[i] \to v[j]) \\ R_{j_0}(i-1, j) & \text{else if } E^{j}_{\mathsf{col}}(i) = E^{j}_{\mathsf{col}}(i-1) + \delta(u[i] \to \varepsilon) \end{cases}$$

5.4 Combining the different steps



Combining these three functions one obtains a function $evaluateallcolumns$. This computes the final pair of columns of tables E_δ and R_{j_0}, given u, v, and a middle column index j_0.

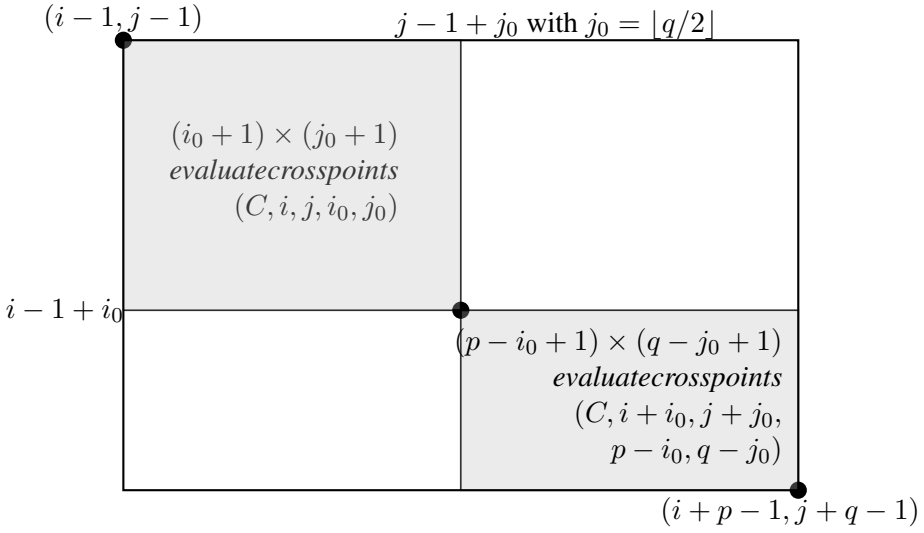
Suppose that we have determined E^n_{col} and $R^n_{j_0}$ as described above, where $j_0 = \lfloor n/2 \rfloor$. Then we know that $edist_\delta(u, v) = E^n_{\mathsf{col}}(m)$ and (i_0, j_0) with $i_0 = R^n_{j_0}(m)$, is on a minimizing path from $(0, 0)$ to (m, n). We store i_0 at index j_0 of a table 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, if $C(j) = i$, then (i, j) is on a minimizing path from $(0, 0)$ to (m, n). Note that $C(0) = 0$ and $C(n) = m$ as $(0, 0)$ and (m, n) are the initial and final crosspoints.

We now have to consider how to combine the computation of a single recursion step (as described above) into a function implementing the divide-step. For this apply the following generalization: suppose, that the two substrings of u and v to align, start at positions i, j, $1 \leq i \leq m$ and $1 \leq j \leq n$ and they are of length p and q, respectively. That is, the algorithm must compute a $(p + 1) \times (q + 1)$-table. Algorithm 10 gives a recursive function *evaluatecrosspoints*, which employs the divide-and-conquer strategy to compute the crosspoints. The initial call is *evaluatecrosspoints*$(C, 1, 1, m, n)$ as we want to compute the crosspoints for an alignment of $u = u[1 \ldots m]$ and $v = v[1 \ldots n]$.

In the algorithm, $evaluateallcolumns$ computes the distance and row index 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 $j - 1$, i.e. the implicit DP-table is divided into two halves at column $j - 1 + j_0$, see Figure 43. To store the columns, $evaluateallcolumns$ uses the vectors E and R, both of length $m + 1$.

Note that a crosspoint is not computed if $n < 2$. In this case, v is of length 0 or 1 and it is easy 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 index columns, and in turn the value i_0. This is correct, since once i_0 is determined, the values stored in vectors E and R are no longer needed. So they can be used in the two recursive calls of line 7 and 8 of Algorithm 10. This is essential to guarantee that the function only consumes $O(m)$ space.

Figure 43: To compute *evaluatecrosspoints*(C, i, j, p, q), the implicit DP-table is divided into two halves at column $j - 1 + j_0$ and row index $i - 1 + i_0$ is determined.



Algorithm 10 (Recursive function to compute crosspoints)

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

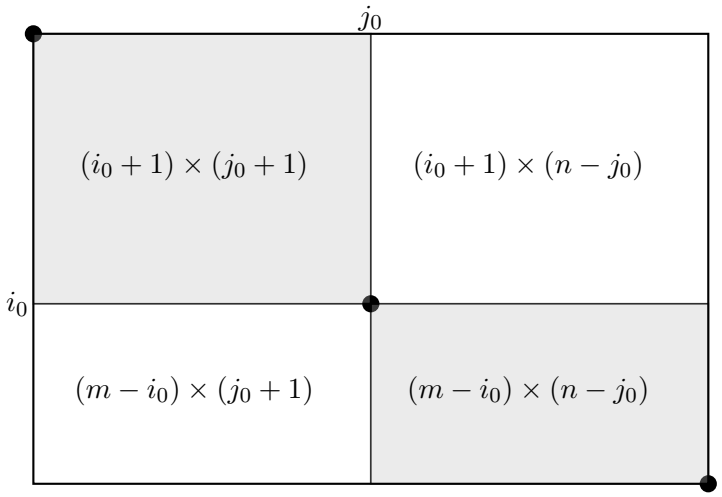
Output: filled table C of $n + 1$) crosspoints

```
1:  function evaluatecrosspoints(C, i, j, p, q)
2:      if q ≥ 2 then
3:          j_0 ← ⌊q/2⌋
4:          evaluateallcolumns(E, R, j_0, u[i ... i + p − 1], v[j ... j + q − 1])
5:          i_0 ← R[p]                          ▷ last value of last R_{j_0}-column
6:          C(j − 1 + j_0) ← i − 1 + i_0        ▷ i/j refer to strings ⇒ subtract 1
7:          evaluatecrosspoints(C, i, j, i_0, j_0)          ▷ eval. upper/left part
8:          evaluatecrosspoints(C, i + i_0, j + j_0, p − i_0, q − j_0)   ▷ low./right
9:      end if
10: end function
11: create table C of n + 1 entries and set C(0) ← 0 and C(n) ← m
12: evaluatecrosspoints(C, 1, 1, m, n)
```

5.5 Analysis of the space requirement

We now analyze the space requirement of the presented algorithm. We previously had pointed out that all calls to the function *evaluatecrosspoints* use the same two $(m+1)$-element vectors. Additionally, $O(n)$ space for 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 $\log_2 n$, the space requirement is $O(m + n + \log n) = O(m + n)$.

For sequences of length m and n, respectively, the function $evaluateallcolumns$ evaluates $(m + 1)(n + 1)$ values in matrix E_δ and $\frac{1}{2}(m + 1)(n + 1)$ values in R_{j_0}, summing up to $\frac{3}{2}(m + 1)(n + 1)$ values. After determining a crosspoint (i_0, j_0), two further matrices are evaluated: a $(i_0 + 1) \times (j_0 + 1)$-matrix (upper left) and a $(m - i_0) \times (n - j_0)$-matrix (lower right). As can be seen in the following illustration (consider the size of the areas reflecting the number of values), the sum of the number of values in these two matrices is about halve of the original size.



So the number of matrix values to compute is $\frac{1}{2}\frac{3}{2}(m + 1)(n + 1)$.

Let $p = \frac{3}{2}(m + 1)$ and $q = n + 1$. The algorithm computes the following number of values over the whole computation:

$$pq + \frac{1}{2}pq + \frac{1}{4}pq + \ldots\ldots = \sum_{i=0}^{\log_2 n} \frac{1}{2^i} pq$$

$$= \underbrace{\left(\sum_{i=0}^{\log_2 n} \frac{1}{2^i} \right)}_{\substack{\text{geometric} \\ \text{series} \leq 2}} pq \leq 2pq = 3(m + 1)(n + 1)$$

Thus the linear space algorithm does not compute more than three times as much values as the quadratic space algorithm. Since each value is computed in constant time, the running time

remains in $O(mn)$. Thus the improvement to linear space does not lead to an increase of the (asymptotic) running time. Empirical experience shows that an implementation of the linear space algorithm takes about twice as much time as the basic algorithm, thus even less than determined from the calculation above.

6 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, consider 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:

```
-FPIKWTAPEAALY---GRFTIKSDVWSFGILLTELTTKGRVPYPGMVNR-EVLDQVERG
-FPIKWTAPEAALY---GRFTIKSDVWSFGILLTELVTKGRVPYPGMVNR-EVLEQVERG
-FPIKWTAPESLAY---NKFSIKSDVWAFGVLLWEIATYGMSPYPGIDLS-QVYELLEKD
QVPVKWTAPEALNY---GRYSSESDVWSFGILLWETFSLGASPYPNLSNQ-QTREFVEKG
QIPVKWTAPEALNY---GWYSSESDVWSFGILLWEAFSLGAVPYANLSNQ-QTREAIEQG
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. Of course, we want to arrange the symbols in the columns in an optimal way to reveal as much as possible differences and similarities of the sequences. Therefore, we need to score multiple alignments.

In the following, we will formally define the notion of multiple alignment and describe two approaches for scoring multiple alignments. Moreover, we present several methods to 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, $1 \leq i \leq k$. 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

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

Definition 32 A *multiple sequence alignment* (*MSA*, for short) of S is a sequence of k strings $(s'_1, \ldots, s'_k)$ satisfying the following properties:

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

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

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

$\ell = |s'_k|$ is the length of the *MSA* $(s'_1, \ldots, s'_k)$ and it holds:

$$\max\{n_i \mid 1 \leq i \leq k\} \leq \ell \leq \sum_{i=1}^{k} n_i$$

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

6.1 Scoring MSAs by consensus costs

We consider two basic approaches to assign a quantitative measure to *MSA*s, i.e. a cost or a score. 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 cost 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, $1 \leq j \leq \ell$ and $a \in \mathcal{A}$ define $count(j, a) = |\{i \mid 1 \leq i \leq k, s'_i[j] = a\}|$, 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, $1 \leq j \leq \ell$, $count(j, cons[j]) \geq count(j, a)$ for all characters $a \in \mathcal{A}$. That is, a consensus sequence consists of characters occurring most often in the corresponding column of the MSA.

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

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

Example 56 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 $dist(cons, (s'_1, s'_2, s'_3)) = 1 + 2 + 1 + 1 + 0 + 2 = 7$.

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

Lemma 9 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, $1 \leq j \leq \ell$. First note that neither

$$count(j, cons_1[j]) > count(j, cons_2[j]) \text{ nor}$$
$$count(j, cons_1[j]) < count(j, cons_2[j]).$$

Hence $count(j, cons_1[j]) = count(j, cons_2[j])$, i.e. $cons_1[j]$ occurs equally often as $cons_2[j]$. As a consequence

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

Since this holds for each j, $1 \leq j \leq \ell$, the sum of the sizes over all j and thus the distance is 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 of an MSA $(s'_1, \ldots, s'_k)$ by

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

where $cons$ is a consensus of $(s'_1, \ldots, s'_k)$. The smaller the cost, the better the multiple alignment. If most columns contain a dominating character, the consensus cost is small. If the characters in the columns vary, the consensus cost is large.

The *MSA*-problem with consensus costs consists of 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).

6.2 Scoring MSAs by sums of pairwise scores

The second optimization criterion for *MSA*s is based on pairwise distances. Let δ be a function satisfying $\delta(\alpha \to \beta) \geq 0$ for each edit operation $\alpha \to \beta$; extend it by requiring $\delta(- \to -) = 0$.

The score is determined by computing the sum of scores (according to δ) for 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, $1 \leq j \leq \ell$ we define the SP-column score

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

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

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

Example 57 Consider the following *MSA*:

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

For all $\alpha, \beta \in \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, -\}$ define $\delta(\alpha \to \beta) = \begin{cases} 0 & \text{if } \alpha = \beta \\ 1 & \text{otherwise} \end{cases}$ We obtain the following SP-score for the alignment above:

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

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

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.

6.3 Computing optimal multiple sequence alignments

The dynamic programming techniques for pairwise sequence alignments employs the following recurrence:

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

This 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 $s_1[1 \ldots q_1], \ldots, s_k[1 \ldots q_k]$. Next we develop a recurrence for M. In analogy to the pairwise case, let $q_1, \ldots, q_k$ be arbitrary but fixed (in the range specified above) and consider all possible last columns of an *MSA* of $s_1[1 \ldots q_1], \ldots, s_k[1 \ldots q_k]$.

Example 58 Consider the 4 sequences $s_1 =$ `CATT`, $s_2 =$ `TATC`, $s_3 =$ `TCA` and $s_4 =$ `ATG`. Here are the 15 possible last columns of any multiple alignment of $S = \{s_1, \ldots, s_4\}$:

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

That is, each last column is a combination of the gap symbol or the last character of the corresponding sequence:

- `T` for row 1,
- `C` for row 2,
- `A` for row 3,
- `G` for row 4.

Note that there is no column consisting of gap-symbols only.

In general, a last column of the MSA consists of any combination of gaps and characters $s_i[q_i]$, $1 \leq i \leq k$. A column of gap-symbols only is not allowed. To describe these combinations one enumerates all k-tuples

$$(d_1, \ldots, d_k) \in \{0,1\} \times \cdots \times \{0,1\} \setminus \{(0, \ldots, 0)\}.$$

$d_i = 0$ means that the ith character in the last column is the gap-symbol. $d_i = 1$ means that the ith character in the last column is $s_i[q_i]$.

Example 59 For $s_1 = \texttt{CATT}$, $s_2 = \texttt{TATC}$, $s_3 = \texttt{TCA}$, $s_4 = \texttt{ATG}$ the 15 last columns

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

of any multiple alignment of $S = \{s_1, \ldots, s_4\}$ can be represented by:

1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	1	1	1	1	0	0	0
1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

If the ith row of the last column is character $s_i[q_i]$ (case $d_i = 1$), then $s_i[q_i]$ is aligned. So in dimension i, one has to align the prefix $s_i[1 \ldots q_i - 1] = s_i[1 \ldots q_i - d_i]$, i.e. one has to refer to the entries in $M(\ldots, q_i - 1, \ldots) = M(\ldots, q_i - d_i, \ldots)$. To obtain the character which appears in the last column, define a function φ by $\varphi(q_i, 1) = s_i[q_i]$. If the ith row of the last column is a gap (case $d_i = 0$), then $s_i[q_i] = s_i[q_i - d_i]$ is not aligned. So in dimension i, one has to align the prefix $s_i[1 \ldots q_i] = s_i[1 \ldots q_i - d_i]$, i.e. one has to refer to the entries in $M(\ldots, q_i - 0, \ldots) = M(\ldots, q_i - d_i, \ldots)$. To obtain the gap appearing in the last column define $\varphi(q_i, 0) = -$. So in both cases, one has to refer to the entry $M(\ldots, q_i - d_i, \ldots)$ and $\varphi(q_i, d_i)$ appears in row i of the last column of the MSA. Assuming that the minimum of an empty set is 0, we thus obtain the following recurrence for M:

$$
\begin{aligned}
&M(q_1, \ldots, q_k) \quad \overbrace{\text{previous entry}} \\
&= \min\{\overbrace{M(q_1 - d_1, \ldots, q_k - d_k)} + \\
&\qquad colcost(\underbrace{\varphi(q_1, d_1), \ldots, \varphi(q_k, d_k)}_{\text{last column}}) \mid (d_1, \ldots, d_k) \in \{0,1\} \times \cdots \times \{0,1\}, \\
&\qquad\qquad (d_1, \ldots, d_k) \neq (0, \ldots, 0), \\
&\qquad\qquad q_1 \geq d_1, \ldots, q_k \geq d_k\}
\end{aligned}
$$

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 consensus cost, then $colcost = conscost$. In the SP-scoring model, $colcost = \delta_{\mathsf{SP}}$.

Example 60 To exemplify the equation, let us partially evaluate it for $k = 2$. In the initial simplification step we obtain the following:

$$
\begin{aligned}
M(q_1, q_2) = \min\{&M(q_1 - d_1, q_2 - d_2) + colcost(\varphi(q_1, d_1), \varphi(q_2, d_2)) \mid (d_1, d_2) \in \{0,1\} \times \{0,1\}, \\
&(d_1, d_2) \neq (0, 0), \\
&q_1 \geq d_1, q_2 \geq d_2\}
\end{aligned}
$$

Now assume that $q_1 \geq 1$ and $q_2 \geq 1$. As $d_1 \leq 1$ and $d_2 \leq 1$, we can conclude $q_1 \geq d_1$ and $q_2 \geq d_2$. Then the right hand side simplifies to:

$$
\begin{aligned}
&= \min\{M(q_1 - d_1, q_2 - d_2) + colcost(\varphi(q_1, d_1), \varphi(q_2, d_2)) \mid (d_1, d_2) \in \{(1,0),(0,1),(1,1)\}\} \\
&= \min\{M(q_1 - d_1, q_2 - d_2) + colcost(\varphi(q_1, d_1), \varphi(q_2, d_2)) \mid (d_1, d_2) = (1,0)\} \cup \\
&\qquad \{M(q_1 - d_1, q_2 - d_2) + colcost(\varphi(q_1, d_1), \varphi(q_2, d_2)) \mid (d_1, d_2) = (0,1)\} \cup \\
&\qquad \{M(q_1 - d_1, q_2 - d_2) + colcost(\varphi(q_1, d_1), \varphi(q_2, d_2)) \mid (d_1, d_2) = (1,1)\} \\
&= \min\{M(q_1 - 1, q_2 - 0) + colcost(\varphi(q_1, 1), \varphi(q_2, 0))\} \cup \\
&\qquad \{M(q_1 - 0, q_2 - 1) + colcost(\varphi(q_1, 0), \varphi(q_2, 1))\} \cup \\
&\qquad \{M(q_1 - 1, q_2 - 1) + colcost(\varphi(q_1, 1), \varphi(q_2, 1))\} \\
&= \min\{M(q_1 - 1, q_2) + colcost(s_1[q_1], -)\} \cup \\
&\qquad \{M(q_1, q_2 - 1) + colcost(-, s_2[q_2])\} \cup \\
&\qquad \{M(q_1 - 1, q_2 - 1) + colcost(s_1[q_1], s_2[q_2])\}
\end{aligned}
$$

Now assume $q_1 \geq 1$ and $q_2 = 0$. Then $q_2 = 0 \geq d_2$ which implies $d_2 = 0$. Hence $d_1 = 1 \leq q_1$ and one obtains the equation

$$
\begin{aligned}
M(q_1, q_2) &= \min\{M(q_1 - d_1, q_2 - d_2) + colcost(s_1[q_1], -) \mid (d_1, d_2) \in \{(1,0)\}\} \\
&= M(q_1 - 1, q_2) + colcost(s_1[q_1], -)
\end{aligned}
$$

Now assume $q_1 = 0$ and $q_2 \geq 1$. In analogy to the previous case we get

$$M(q_1, q_2) = M(q_1, q_2 - 1) + colcost(-, s_2[q_2])$$

Finally, for $q_1 = 0$ and $q_2 = 0$ one obtains the equation $M(q_1, q_2) = 0$. Now put all four cases together, using the compact notation:

$$
M(q_1, q_2) = \min \left\{ \begin{array}{ll}
M(q_1 - 1, q_2) + colcost(s_1[q_1], -) & \text{if } q_1 \geq 1 \\
M(q_1, q_2 - 1) + colcost(-, s_2[q_2]) & \text{if } q_2 \geq 1 \\
M(q_1 - 1, q_2 - 1) + colcost(s_1[q_1], s_2[q_2]) & \text{if } q_1, q_2 \geq 1
\end{array} \right\}
$$

Let $colcost$ be a function that assigns positive costs to pairs of different characters and cost 0 to pairs of identical characters. This holds for example for the consensus cost function as well as for the sum-of-pairs score function. Hence the previous equation, after renaming

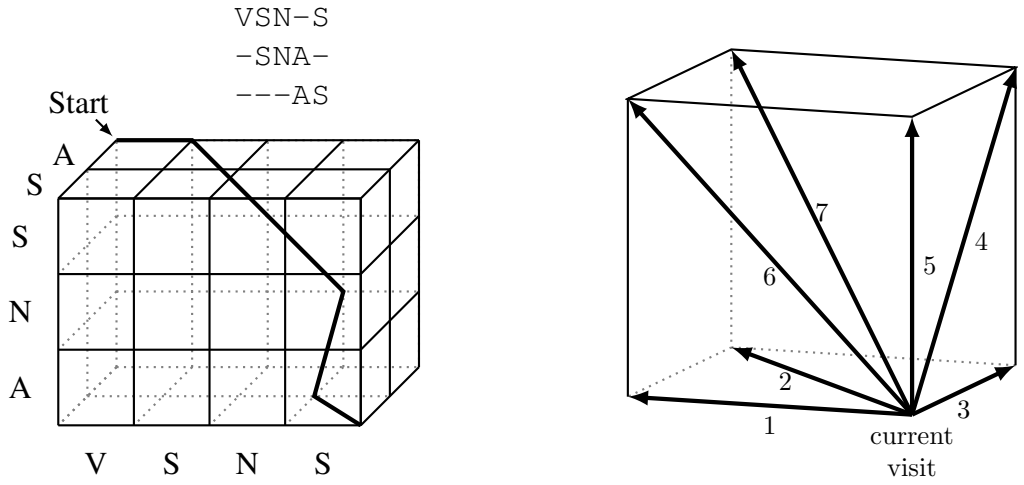
M	q_1	q_2	s_1	s_2	$colcost$
E_δ	i	j	u	v	δ

leads to the equation

$$
E_\delta(i, j) = \min \left\{ \begin{array}{ll}
E_\delta(i - 1, j) + \delta(u[i], -) & \text{if } i \geq 1 \\
E_\delta(i, j - 1) + \delta(-, v[j]) & \text{if } j \geq 1 \\
E_\delta(i - 1, j - 1) + \delta(u[i], v[j]) & \text{if } i, j \geq 1
\end{array} \right\}
$$

which is equivalent to the recurrence for E_δ shown at the beginning of this section. In an exercise you will be asked to partially evaluate the recurrence for $k = 3$.

Figure 44: The case for 3 sequences: alignment path (left) and 7 dependencies, each represented by an edge, for a given node (right) in the 3-dimensional table.



Now let us analyze the efficiency of an algorithm computing M by

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

Table M has $z = \prod_{i=1}^{k}(n_i + 1)$ entries. Each of the z entries in table M requires to compute the minimum of up to $2^k - 1$ values, each corresponding to a k-tuple $(d_1, \ldots, d_k) \in \{0,1\} \times \cdots \times \{0,1\} \setminus \{(0, \ldots, 0)\}$. Each of the up to $2^k - 1$ values is the sum of another entry in table M plus the cost for the corresponding last column of an MSA.

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 all $k(k-1)/2$ pairwise scores have to be added up. In a geometric view, each MSA can be considered as a path in a k-dimensional hypercube from one corner to the opposite corner. See, for example, Figure 44 (left), showing the path corresponding to an MSA of the sequence `VSNS`, `SNA`, `AS`. The up to $2^k - 1$ dependencies of a single entry in table M correspond to $2^k - 1$ backward edges in the k-dimensional hypercube, see Figure 44 (right) showing $2^k - 1 = 2^3 = 7$ backward edges for $k = 3$.

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 the bitvector in each entry (as described above), one can perform a traceback starting at entry $M(n_1, \ldots, n_k)$, computing each column of an optimal alignment in $O(k)$ steps. Hence the traceback phase requires $O(k \cdot \ell)$ 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.

Example 61 Consider the SP-scoring model and assume that each computation step requires 1 nanosecond (i.e. 10^{-9} seconds). The following table shows the approximate running time (in years) to compute MSAs for k sequences, each of length $\ell = 100$ or $\ell = 200$:

	running times (years)	
k	$\ell = 100$	$\ell = 200$
7	$2.0 \cdot 10^0$	$2.6 \cdot 10^2$
8	$8.1 \cdot 10^2$	$2.1 \cdot 10^5$
9	$1.8 \cdot 10^5$	$9.1 \cdot 10^7$
10	$2.6 \cdot 10^7$	$2.7 \cdot 10^{10}$

6.4 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. The resulting *MSA* of all sequences needs to incorporate the information from the pairwise alignments. This is expressed by the notion of compatibility, defined next.

Let $S = \{s_1, \ldots, s_k\}$ be a set of sequences, and S' be a subset of S. Consider 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 an alignment $Al_{S'}$ of S' if $Al_{S'} = proj(Al_S, S')$.

Example 62 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 (so Al_S is compatible with this alignment):

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

And $proj(Al_S, S'')$ is as follows (so Al_S is compatible with this alignment):

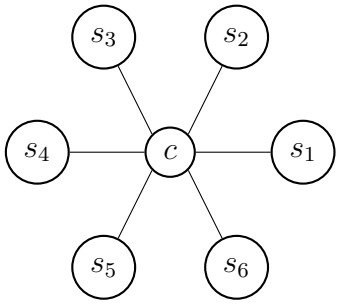
```
A - - T G
A T C G G
```

The relation between pairs of sequences can be represented by an alignment tree:

Definition 33 Let S be a set of sequences. An *alignment tree* for S is a labeled undirected 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 each of 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 (the root) and edges (c, s) for all $s \in S \setminus \{c\}$ (which are the leaves). In this case, one often uses the notion *star alignment*. See also Figure 45 for a

Figure 45: A star alignment tree of 7 sequences $c, s_1, \ldots, s_6$ with the center c and the remaining sequences $s_1, \ldots, s_6$ as leaves. Each edge from node s to node s' is implicitly labeled by an optimal pairwise alignment of s and s'.



star alignment tree. The star alignment method first identifies 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. 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 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 11. In this way, the resulting multiple alignment will be compatible with the pairwise alignments.

Example 63 Consider the sequences $c =$ ATGCATT, $s_1 =$ AGTCAAT, $s_2 =$ TCTCA, $s_3 =$ ACTGTAATT and the alignments

$$\begin{array}{ll} c' = & \texttt{A T G - C A T T} \\ s'_1 = & \texttt{A - G T C A A T} \\ s'_2 = & \texttt{- T C T C A - -} \end{array} \qquad \text{and} \qquad \begin{array}{ll} c'' = & \texttt{A - T G C - A T T} \\ s''_3 = & \texttt{A C T G T A A T T} \end{array}$$

The following table shows for which pairs of sequences the gaps are combined. The resulting *MSA* is shown in the last column.

combine		result
c'	c''	c''' = A-TG-C-ATT
s'_1	c''	s'''_1 = A--GTC-AAT
s'_2	c''	s'''_2 = --TCTC-A--
s''_3	c'	s'''_3 = ACTG-TAATT
		MSA

Note that the resulting *MSA* is not optimal, since it would probably decrease the distance, if we would replace s'''_3 by ACTGT-AATT.

Next we show that Algorithm 11 well approximates the optimal SP-alignment, if the pairwise scoring function satisfies some additional properties.

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

Algorithm 11 (Star alignment)

Input: A set S of sequences
Output: Star alignment Al of S
Part (1): Compute the center sequence c of the star alignment

for all $s \in S$ **do**
  for all $t \in S \setminus \{s\}$ **do**
    compute an opt. alignment of s and t with dist. $\delta_{\mathsf{opt}}(s,t)$
  end for
end for
$totalscore(t) \leftarrow \sum_{s \in S \setminus \{t\}} \delta_{\mathsf{opt}}(s,t)$
let $c \in S$ s.t. $totalscore(c) \leq totalscore(s)$ for all $s \in S$
let T be the star alignment tree with center c and leaves $S \setminus \{c\}$

Part (2): Determine a compatible multiple alignment

choose an arbitrary $s \in S$
let Al be an optimal alignment of c and s
$S' \leftarrow \{c, s\}$
while $S' \neq S$ **do**
  choose an arbitrary $s' \in S \setminus S'$
  $S' \leftarrow S' \cup \{s'\}$
  Determine an *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".
  ▷ assertion: Al' is compatible with Al and with the chosen optimal alignment of c and s'.
  $Al \leftarrow Al'$
end while

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

Lemma 10 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, $1 \leq i, j \leq k$:

$$\delta(s'_i, s'_j) \leq \delta(s'_i, c') + \delta(c', s'_j) = \delta_{\mathsf{opt}}(s_i, c) + \delta_{\mathsf{opt}}(c, s_j) \tag{6.1}$$

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

$$\begin{aligned}\delta(s'_i, s'_j) &= \sum_{1\le q\le \ell} \delta(s'_i[q], s'_j[q]) \\ &\le \sum_{1\le q\le \ell} (\delta(s'_i[q], c'[q]) + \delta(c'[q], s'_j[q]) \\ &= \sum_{1\le q\le \ell} \delta(s'_i[q], c'[q]) + \sum_{1\le q\le \ell} \delta(c'[q], s'_j[q]) \\ &= \delta(s'_i, c') + \delta(c', s'_j)\end{aligned}$$

It remains to show the equality in (6.1). 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, $\delta(s'_i, c') = \delta_{\mathsf{opt}}(s_i, c)$. With an analogous argumentation, one shows $\delta(c', s'_j) = \delta_{\mathsf{opt}}(c, s_j)$ which completes the proof. □

Now we can prove the central theorem of this section:

Theorem 8 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 11. Then the following property holds:

$$\underbrace{\delta_{\mathsf{SP}}(s'_1, \ldots, s'_k)}_{\substack{\text{star}\\ \text{alignment}}} \le \underbrace{\left(2 - \frac{2}{k}\right)}_{\substack{\text{approx}\\ \text{factor}}} \cdot \underbrace{\delta_{\mathsf{opt-SP}}(s_1, \ldots, s_k)}_{\substack{\text{optimal SP}\\ \text{alignment cost}}}$$

Proof: Consider an *MSA* $(s''_1, \ldots, s''_k)$ of S such that $\delta_{\mathsf{SP}}(s''_1, \ldots, s''_k) = \delta_{\mathsf{opt-SP}}(s_1, \ldots, s_k)$, 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 $c_{\mathsf{score}} = \sum_{i=1}^{k-1} \delta(s_i, c)$. Now first

observe, that

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

Moreover, we derive another inequality for sum'':

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

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

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

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

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

This result shows that the star alignment algorithm delivers a good approximation of the optimal SP alignment (by a factor $\leq \left(2 - \frac{2}{k}\right)$). By definition we also have $\delta_{\mathsf{opt-SP}}(s_1, \ldots, s_k) \leq \delta_{\mathsf{SP}}(s'_1, \ldots, s'_k)$ 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 $\delta_{\mathsf{SP}}(s'_1, \ldots, s'_k)$ using Algorithm 11. Now, if a value $d = M(q_1, \ldots, q_k)$ satisfies $d > \delta_{\mathsf{SP}}(s'_1, \ldots, s'_k)$ then $d > \delta_{\mathsf{opt-SP}}(s_1, \ldots, s_k)$ 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 entry in matrix M larger than

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

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

6.5 Multiple alignment in practice

6.5.1 Iterative multiple alignment

A more general approach than the star-alignment uses a rooted tree with the sequences $s_1, \ldots, s_k$ at the leaves. It then iteratively combines alignments along the paths of the tree. More specifically, such an iterative multiple alignment approach consists of the following steps (see Fig. 46):

1. For all pairs of sequences, compute an optimal pairwise alignment and the corresponding edit distance.
2. From these pairwise distances compute a tree whose leaves are the sequences.
3. Successively combine pairwise alignments to larger and larger alignments, guided by the branching order of the tree until one large multiple alignment remains.

The mentioned tree is the so-called *guide tree*.

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. This is exactly what the UPGMA algorithm does. It is explained in section 7.3.

The most striking advantages of the iterative multiple alignment method, in comparison to the optimal alignment computation via matrix M, are:

- 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 high quality MSAs.

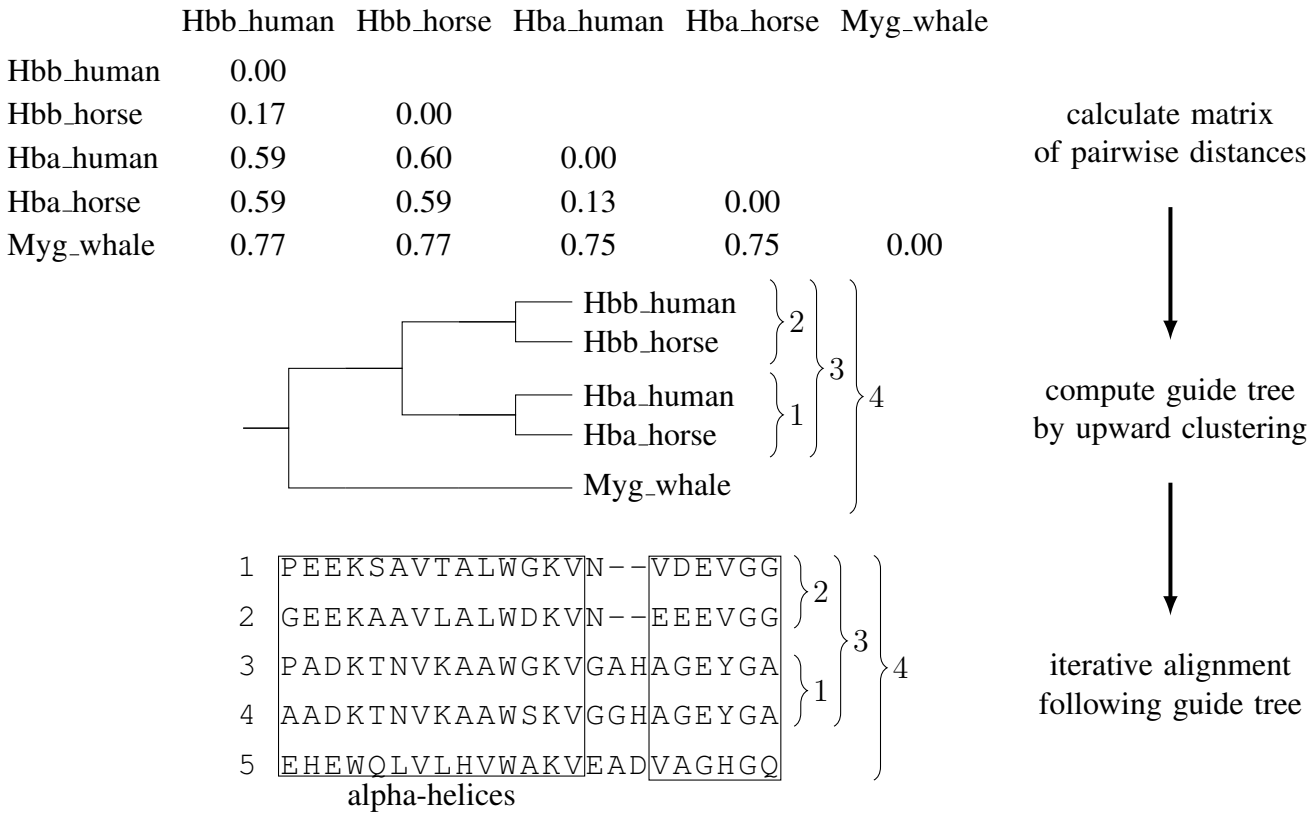
On the other hand there are a few disadvantages:

- Errors in an early step cannot be eliminated later, due to the applied principle "once a gap, always a gap".
- There is no well-defined objective function which must be optimized during the iterative alignment method. Thus the evaluation of the relative performance of this approach in comparison to other similar methods is difficult.
- A (possibly wrong) guide tree remains represented in the alignment and might bias the result.

One of the most widely used programs for iterative multiple alignment is the ClustalW program,[1] and its graphical user interface ClustalX.[2]

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

Figure 46: Overview of ClustalW. Figure adapted from [http://www.slideshare.net/jomcinerney/alignments]

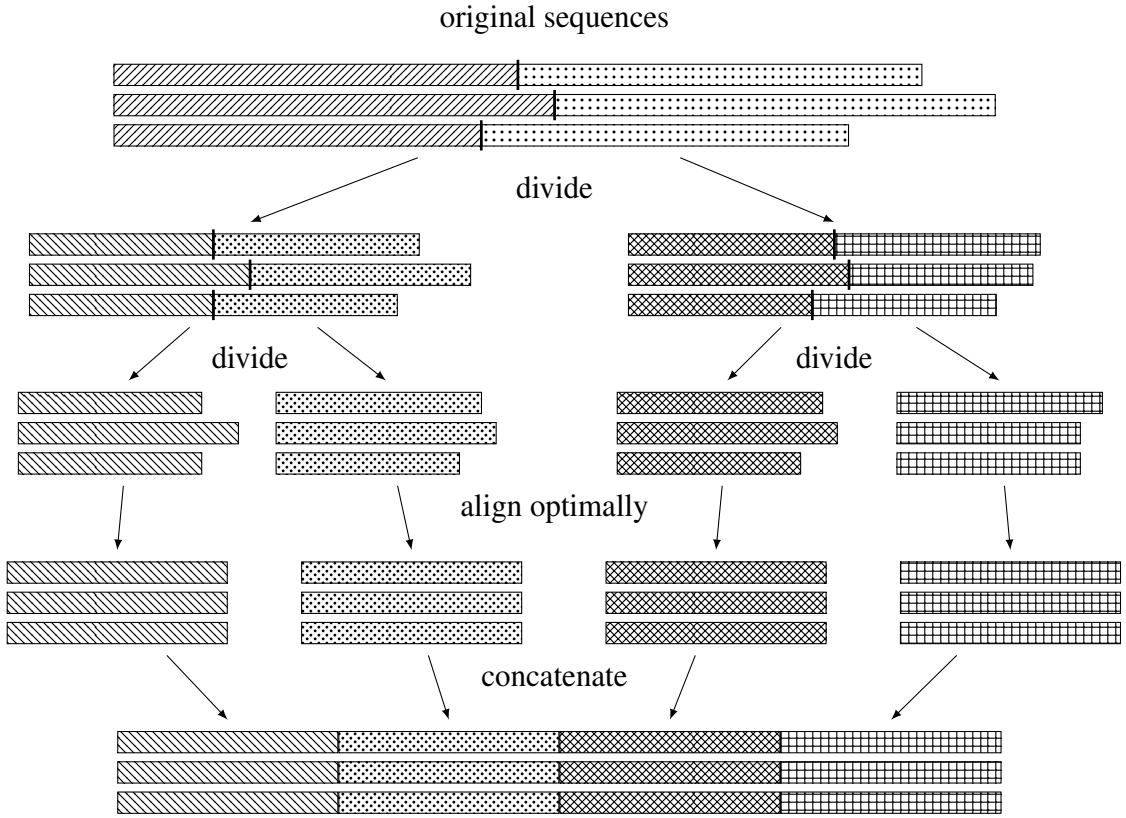


6.5.2 Divide-and-conquer alignment

The interdependence of tree and alignment is avoided in the divide-and-conquer multiple alignment algorithm (DCA, for short), described in [SMD97]. It works for the sum-of-pairs cost model.

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 method is re-iterated until the sequences are sufficiently short, so that they can be aligned optimally by the program MSA. Finally, the resulting short alignments are concatenated, yielding a multiple alignment of the original sequences, see Figure 47 for a graphical explanation.

Figure 47: The divide-and-conquer alignment method (DCA)



Computing Cut Positions

The main difficulty with this approach is 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 is used. This heuristic quantifies the compatibility of cut positions in distinct sequences.

Definition 34 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 $s[c+1 \ldots n]$.

Definition 35 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, define

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

Note that $\delta(s_p, s_q)$ is an abbreviated notation for the edit distance of s_p and s_q with respect to the cost function δ. $\delta(s_p(\leq c_p), s_q(\leq c_q))$ is the edit distance of the two prefix sequences

(ending at position c_p and c_q), respectively. $\delta(s_p(> c_p), s_q(> c_q))$ is the edit distance of the two suffix sequences (beginning at position $c_p + 1$ and $c_q + 1$), respectively. So $C_{s_p,s_q}[c_p, c_q]$ is the additional cost of forcing the pairwise alignment to not cross the cut positions c_p and c_q: If there is an optimal alignment which aligns $s_p(\leq c_p)$ with $s_q(\leq c_q)$ and $s_p(> c_p)$ with $s_q(> c_q)$, then $C_{s_p,s_q}[c_p, c_q] = 0$.

One uses standard methods for computing the global edit distance of two sequences to determine the required values above. Next one combines these values to quantify the effect of choosing the cut positions for all k sequences.

Definition 36 For given cut positions $c_1, c_2, \ldots, c_k$ define

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

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 ccp 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 of Carillo and Lipman to the sequences to obtain an optimal multiple sequence alignment. The DCA algorithm can be described by the following recursive-function:

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

The operator $\cdot$ between the results of the recursive calls of DCA is the concatenation of two MSAs.

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

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

7 Phylogenetic analysis

This section follows the lecture notes of Daniel Huson, University of Tuebingen.

Given a collection of species, the goal of phylogenetic analysis is to determine and describe the evolutionary relationship between the species. In particular, this involves determining the order of speciation events and their approximate timing.

It is generally assumed that speciation is a branching process: a population of organisms becomes separated into two sub-populations. Over time, these evolve into separate species that do not cross-breed.

Because of this assumption, a tree is often used to represent a proposed phylogeny for a set of species, showing how the species evolved from a common ancestor. We will study this in detail.

7.1 Phylogenetic trees

In the following, we will use X to denote a finite set of taxa. A taxon $x \in X$ is simply a representative of a group of individuals defined in some way. A phylogenetic tree (on X) is a triple $T = (V, E, \lambda)$ consisting of a connected graph (V, E) without cycles, together with a one-to-one labeling λ of the leaves by elements of X. There are two kinds of phylogenetic trees*:

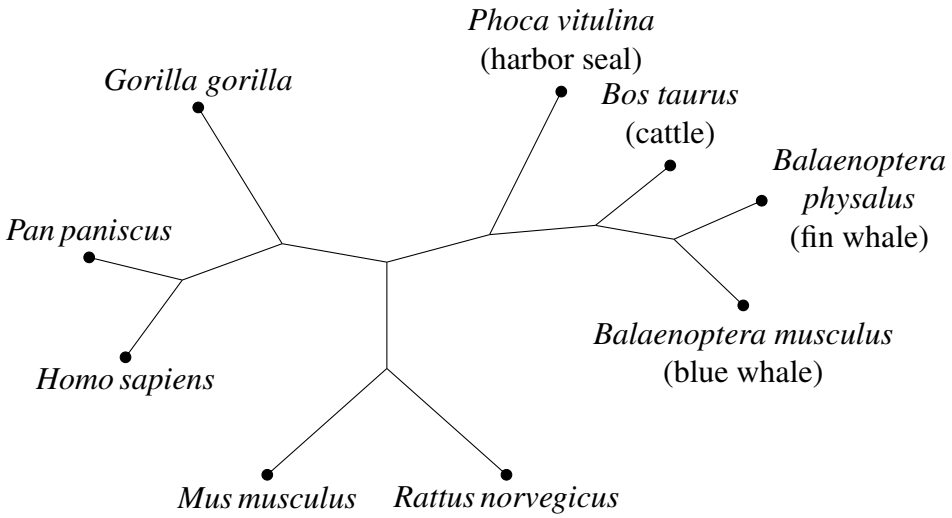
(1) Rooted trees reflect the most basal ancestor of the tree in question.

(2) Unrooted trees do not imply a known ancestral root.

https://www.ncbi.nlm.nih.gov/Class/NAWBIS/Modules/Phylogenetics/phylo9.html

A phylogenetic tree T is called *binary*, if every internal node has degree 3, except ρ, in case T is rooted with root ρ. Sometimes we will relax the definition to allow labels to be placed on internal nodes, or nodes to carry multiple labels. Occasionally we will allow an arbitrary node to be the root.

An unrooted phylogenetic tree for 9 taxa placed on the leaves. All internal nodes have degree 3. Such a tree is often displayed in a circular layout.

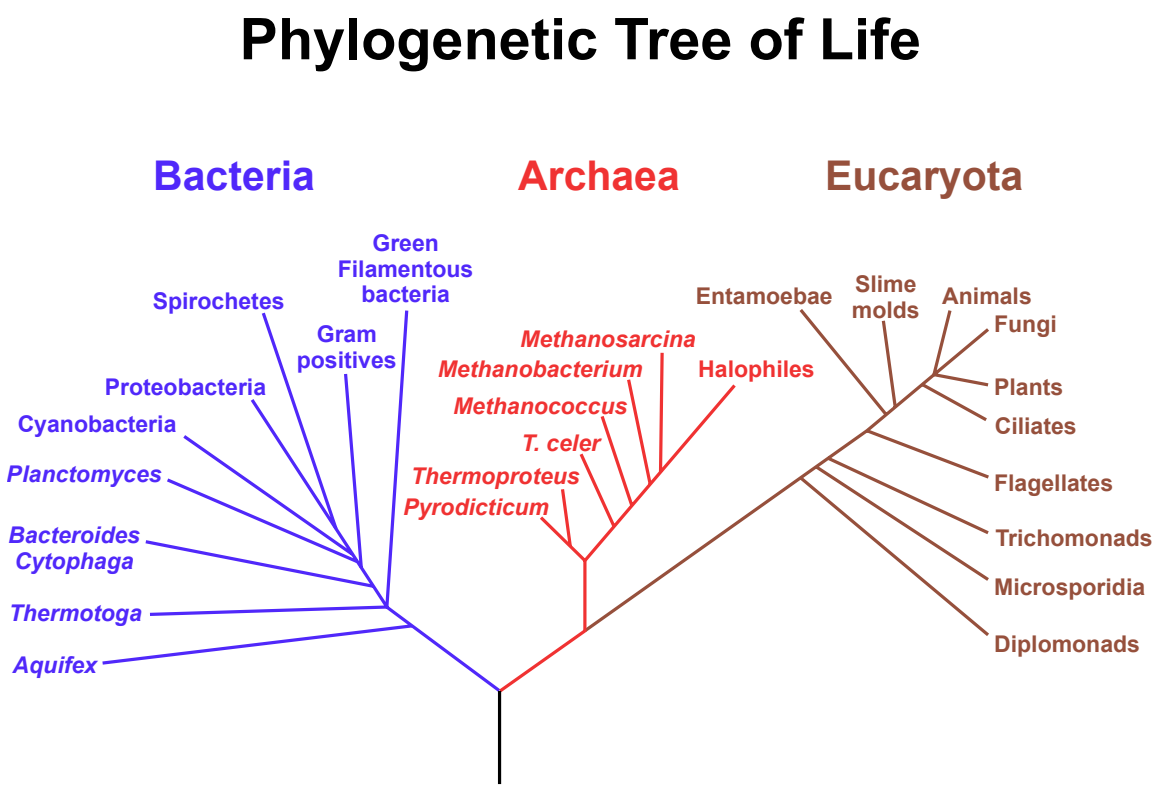


A rooted tree is usually drawn with the root placed at the bottom, top or left side, see Figure 50 and Figure 48.

7.1.1 Edge lengths

A phylogenetic tree describes the putative order of speciation events that gave rise to the considered taxa. Additionally, one assigns positive length to each edge of the tree. Ideally, these lengths should be proportional to the amount of time that lies between speciation events. However, in practice the edge lengths usually represent quantities obtained by some given computation and only correspond very indirectly to time.

Figure 48: A rooted phylogenetic tree of life, derived from the comparison of rRNA genes. It shows the three major kingdoms Bacteria, Archaea, and Eucaryota.



Source: `http://en.wikipedia.org/wiki/File:Phylogenetic_tree.svg`

7.1.2 Approaches to construct phylogenetic trees

There are three main approaches to constructing phylogenetic trees from molecular data.

1. *Distance methods* first compute a *distance function* from a given set of biological data and then determine a tree representing these distances as closely as possible.
2. *Maximum parsimony* takes as input a set of aligned sequences and attempts to find a tree and a labeling of its internal nodes by auxiliary sequences such that the number of mutations along the edges of the tree is a minimum.
3. Given a probabilistic model of evolution, *maximum likelihood approaches* aim at finding a phylogenetic tree that maximizes the likelihood of obtaining the given sequences.

Here we will focus on distance methods. For a set X of taxa we assume a distance function $d : X \times X \to \mathbb{R}^0_+$ that associates a distance $d(x, y)$ with every pair of taxa $x, y \in X$. We usually require that d is a metric, see definition 9.

Figure 49: Phylogenetic tree of novel coronavirus according to `https://nextstrain.org/ncov/global`. Clades 20H, 20I, 20J contain the 501Y mutation occurring in the SARS-CoV-2 virus B.1.1.7 spreading in Great Britain since late 2020.

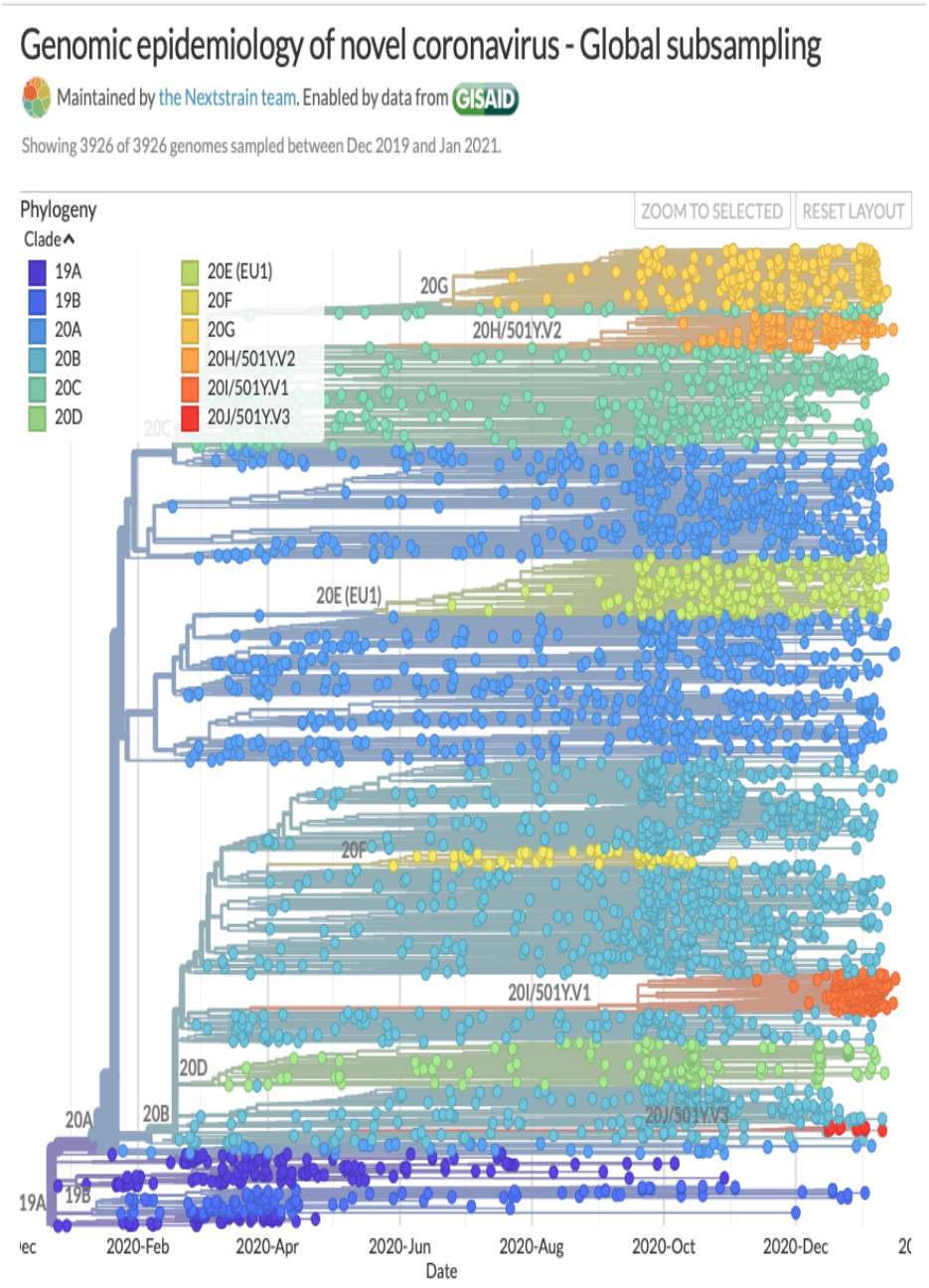
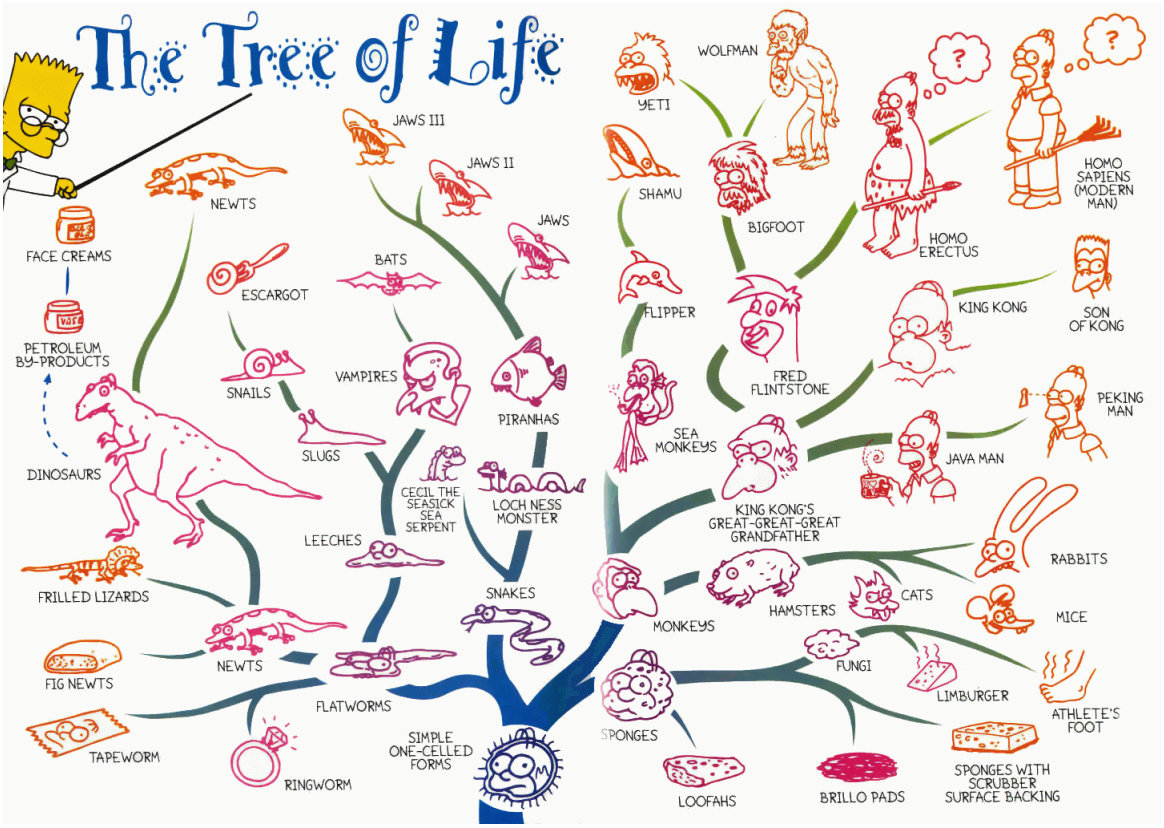


Figure 50: A modern version of the (rooted) tree of life, according to Homer Simpson, https://www.pinterest.com/pin/526639750148386179/



7.2 Additivity and the four-point condition

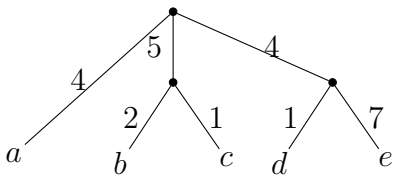
We next consider an important notion relating distance functions and phylogenetic trees.

Definition 37 Let X be a set of taxa and T_X be a phylogenetic tree over X. Let $d : X \times X \to \mathbb{R}^0_+$ be a distance function. d is *additive* with respect to T_X, if it was obtained by adding up edge lengths on paths between the leaves of T_X. That is, for all $x, y \in X$, $d(x, y)$ is the sum of the edge lengths of the unique path in T_X from x to y.

In the following we abbreviate 'with respect to' by 'w.r.t.'.

Example 64 Let $X = \{a, b, c, d, e\}$. Here is an example of a distance function $d : X \times X \to \mathbb{R}^0_+$ which is additive w.r.t. to the phylogenetic tree on the right:

	a	b	c	d	e
a	0	11	10	9	15
b	11	0	3	12	18
c	10	3	0	11	17
d	9	12	11	0	8
d	15	18	17	8	0



For example, $d(a, e) = 4 + 4 + 7 = 15$ and $d(a, c) = 4 + 5 + 1 = 10$.

Definition 38 A distance function $d : X \times X \to \mathbb{R}^0_+$ is *additive*, if there is a phylogenetic tree T_X such that d is additive with respect to T_X.

Interestingly, we can decide whether a distance function is additive without determining an appropriate phylogenetic tree. The method is based on a result by Buneman [Bun74].

Theorem 9 A distance function d on X is additive, if and only if for any four (not necessarily distinct) elements $w, x, y, z \in X$ the so-called four-point condition holds:

$$d(w, x) + d(y, z) \leq \max\{d(w, y) + d(x, z), d(w, z) + d(x, y)\} \tag{7.1}$$

Using this theorem, one can decide whether or not a distance function is additive. If $|X| = n$, then one has to enumerate all n^4 4-tuples $(w, x, y, z) \in X \times X \times X \times X$ of taxa and check whether condition (7.1) holds. This can be done in constant time for each 4-tuple. So the running time of such a decision method is $O(n^4)$.

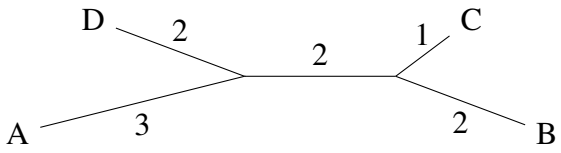
Example 65 Consider the distance function d for $X = \{A, B, C, D\}$, given by the following distance matrix:

	A	B	C	D
A	0	7	6	5
B	7	0	3	6
C	6	3	0	5
D	5	6	5	0

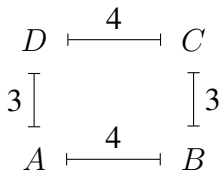
We evaluate
$d(A, B) + d(C, D) = 7 + 5 = 12$
$d(A, C) + d(B, D) = 6 + 6 = 12$
$d(A, D) + d(B, C) = 5 + 3 = 8$
and have to check $4^4 = 256$ 4-tuples.
For example, we verify

$$12 = d(A,B) + d(C,D) \leq \max\{d(A,C) + d(B,D), d(A,D) + d(B,C)\} = 12$$

Indeed, for all 4-tuples, the four-point condition holds. So d is additive. Here is a phylogenetic tree T_X such that d is additive w.r.t. T_X.



Example 66 Consider as distances the euclidean distances given in the plane:



The distance of D and B and of A and C is $\sqrt{4^2 + 3^2} = 5$. We evaluate

$$d(A, B) + d(C, D) = 4 + 4 = 8$$
$$d(A, C) + d(B, D) = 5 + 5 = 10$$
$$d(A, D) + d(B, C) = 3 + 3 = 6$$

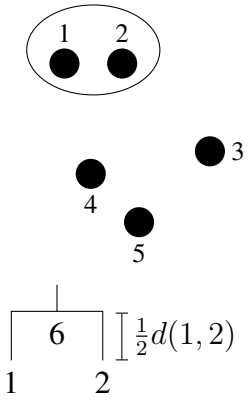
$$\text{Now } d(A, C) + d(B, D) = 10 > 8 = \max\{8, 6\} = \max\{d(A, B) + d(C, D), d(A, D) + d(B, C)\}$$

Hence the four-point condition does not hold. So d is not additive.

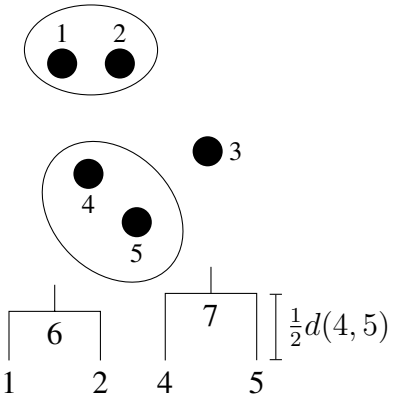
7.3 The UPGMA reconstruction method

We will now present a simple distance method called UPGMA which stands for *unweighted pair group method using arithmetic averages*. It was first described by Sokal & Michener [SM58]. Given a set of taxa X and a distance function d, UPGMA produces a rooted binary phylogenetic tree T_X with edge lengths. It operates by clustering the given taxa, at each stage merging two clusters, 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.

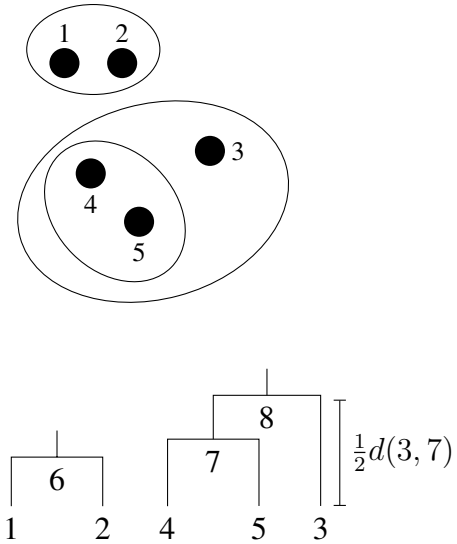
Example 67 Let $X = \{1, 2, 3, 4, 5\}$ and consider distances in the plane. The different steps of the UPGMA method are shown in the following illustration:



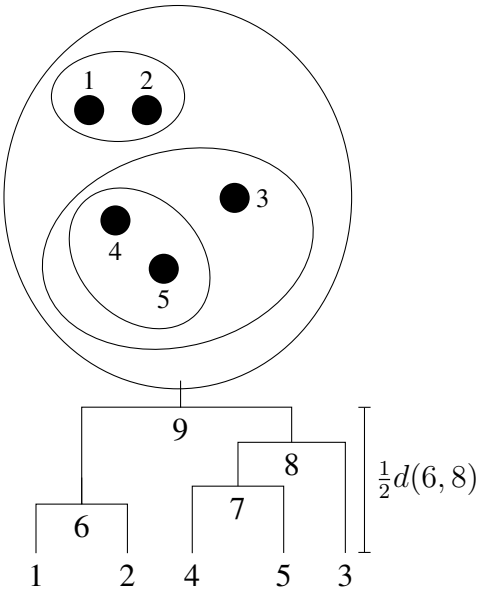
union of 1 and 2 ⇒ cluster 6



union of 4 and 5 ⇒ cluster 7



union of 7 and 3 ⇒ cluster 8



union of 6 and 8 ⇒ cluster 9

Initially, we are given a distance $d(x, y)$ between any two $x, y \in X$. We define the distance $d(C_i,C_j)$ between two clusters $C_i \subseteq X$ and $C_j \subseteq X$ with $C_i \cap C_j = \emptyset$ to be the average distance between all pairs of taxa from each cluster:

$$d(C_i,C_j) = \frac{1}{|C_i||C_j|} \sum_{x \in C_i} \sum_{y \in C_j} d(x, y)$$

Obviously, $|C_i||C_j|$ is the number of pairs we can construct from the clusters C_i and C_j. We have $d(\{x\}, \{y\}) = d(x, y)$ according to this definition. The following Lemma shows how to efficiently compute the distance between two clusters.

Lemma 11 If C_k is the union of two clusters C_i and C_j, and C_ℓ is any other cluster, then

$$d(C_k,C_\ell) = \frac{d(C_i,C_\ell)|C_i| + d(C_j,C_\ell)|C_j|}{|C_i| + |C_j|}$$

Proof:

$$\begin{aligned}
d(C_k,C_\ell) &= \frac{1}{|C_k||C_\ell|} \sum_{x \in C_k} \sum_{y \in C_\ell} d(x, y) \\
&= \frac{\frac{1}{|C_\ell|} \left(\sum_{x \in C_i \cup C_j} \sum_{y \in C_\ell} d(x, y) \right)}{|C_k|} \\
&= \frac{\frac{1}{|C_\ell|} \left(\sum_{x \in C_i} \sum_{y \in C_\ell} d(x, y) \right) + \frac{1}{|C_\ell|} \left(\sum_{x \in C_j} \sum_{y \in C_\ell} d(x, y) \right)}{|C_i|+|C_j|} \\
&= \frac{|C_i| \frac{1}{|C_i| \cdot |C_\ell|} \left(\sum_{x \in C_i} \sum_{y \in C_\ell} d(x, y) \right) + |C_j| \frac{1}{|C_j||C_\ell|} \left(\sum_{x \in C_j} \sum_{y \in C_\ell} d(x, y) \right)}{|C_i|+|C_j|} \\
&= \frac{|C_i| d(C_i,C_\ell) + |C_j| d(C_j,C_\ell)}{|C_i| + |C_j|} \\
&= \frac{d(C_i,C_\ell)|C_i| + d(C_j,C_\ell)|C_j|}{|C_i| + |C_j|}
\end{aligned}$$

The UPGMA reconstruction method is straightforward and described in Algorithm 12.

According to [Mur84], the running time of the UPGMA algorithm is $O(n^2)$.

Example 68 We consider a distance function over 5 sequences (Bsu, Bst, Lvi, Amo, and Mlu). The meaning of these abbreviations and a distance function (given as a matrix) is as follows:

Bsu *Bacillus subtilis*
Bst *Bacillus stearothermophilus*
Lvi *Lactobacillus viridescens*
Amo *Acholeplasma modicum*
Mlu *Micrococcus luteus*

	Bsu	*Bst*	*Lvi*	*Amo*	*Mlu*
Bsu	0.0	0.1715	0.2147	0.3091	0.2326
Bst		0.0	0.2991	0.3399	0.2058
Lvi			0.0	0.2795	0.3943
Amo				0.0	0.4289
Mlu					0.0

Algorithm 12 (UPGMA reconstruction method)

Input: A set of taxa X and a corresponding distance function d.
Output: A binary, rooted phylogenetic UPGMA tree T.

Initialization: Assign each taxon x_i to its own cluster C_i. Define one leaf of T for each taxon, placed at height zero

Iteration: Determine a pair (C_i, C_j) of distinct clusters for which $d(C_i,C_j)$ is minimal. Define a new cluster k by $C_k = C_i \cup C_j$. Determine $d(C_k,C_\ell)$ in constant time for all existing clusters C_ℓ using the update formula from Lemma 11. Define a node k with child nodes i and j, and place it at height $\frac{1}{2}d(C_i,C_j)$. Add C_k to the current set of clusters; remove C_i and C_j.

Termination: When only 2 clusters C_i and C_j remain, place the root at height $\frac{1}{2}d(C_i,C_j)$ and connect it to its children i and j

Each step of the UPGMA algorithm reduces the number of taxa by one. Hence the distance matrices become smaller and smaller. The first clustering step identifies Bsu and Bst to be the sequences with minimum distance 0.1715. These sequences are clustered, the new cluster is placed on a node at height $0.5 \cdot 0.1715 = 0.08575$ and the following distance matrix is produced:

	$\{Bsu, Bst\}$	Lvi	Amo	Mlu
$\{Bsu, Bst\}$	0.0	0.2569	0.3245	0.2192
Lvi		0.0	0.2795	0.3943
Amo			0.0	0.4289
Mlu				0.0

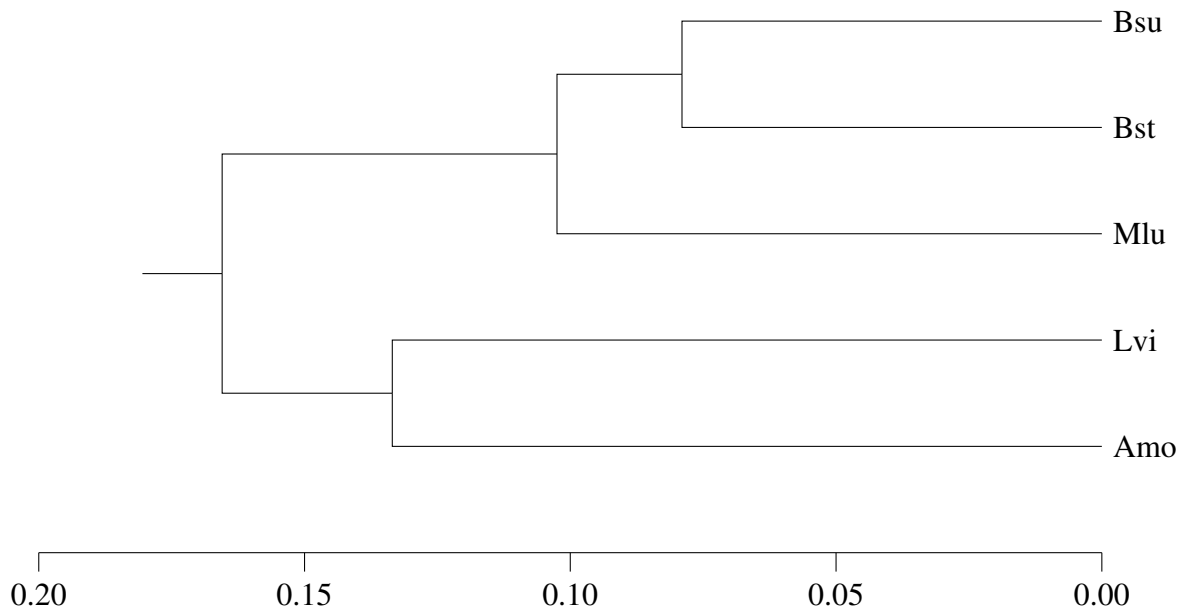
Now the algorithm clusters $\{Bsu, Bst\}$ and Mlu, since their distance 0.2192 is minimal. This step creates a new cluster at a node of height $0.5 \cdot 0.2192 = 0.1096$ and leads to the following distance function:

	$\{Bsu, Bst, Mlu\}$	Lvi	Amo
$\{Bsu, Bst, Mlu\}$	0.0	0.3027	0.3593
Lvi		0.0	0.2795
Amo			0.0

Now Lvi and Amo are clustered since their distance 0.2795 is minimal. This step creates a new cluster at a node of height $0.5 \cdot 0.2795 = 0.13975$ and the following distance matrix is produced:

	$\{Bsu, Bst, Mlu\}$	$\{Lvi, Amo\}$
$\{Bsu, Bst, Mlu\}$	0.0	0.3310
$\{Lvi, Amo\}$		0.0

The method produces the following tree with a root at height $0.5 \cdot 0.3310 = 0.1655$:



It is biologically incorrect, as we will see later.

The goal of phylogenetic analysis is usually to reconstruct a phylogenetic tree from real data that is consistent with some (unknown) tree describing the path of events that evolution actually took. The data can be e.g. sequences and possibly pairwise distances derived from the sequences. Unfortunately, we do not know the tree describing the evolution.

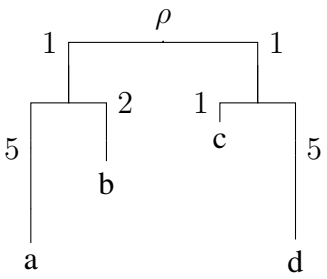
In contrast, in simulation studies, a known tree T^0 is used to generate artificial sequences and/or distances, under some specified model of evolution. A method to construct phylogenetic trees is then applied and its performance can be evaluated by comparing the resulting tree T with the true tree T^0.

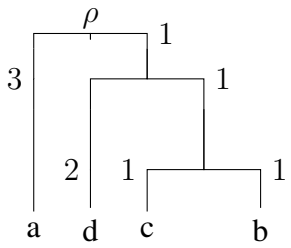
7.3.1 The molecular clock hypothesis

Given a distance function $d : X \times X \to \mathbb{R}_+^0$, the UPGMA method aims at building a rooted tree T_X with the property that all leaves have the same distance from the root ρ, see the following example:

Example 69 Here is a tree in which all leaves have the same distance 3 from the root.

Figure 51: A tree that cannot be reproduced by the UPGMA algorithm, as e.g. $d(\rho, b) = 3 \neq 6 = d(\rho, a)$.





So the UPGMA method is suitable for distance functions based on sequence data that has evolved with a mutation rate that is constant over time. The hypothesis that evolutionary events happen at a constant rate is called the *molecular clock hypothesis*.

If the distance function d on X is additive w.r.t. a binary rooted phylogenetic tree T_X that adheres to the molecular clock hypothesis, then the UPGMA method will construct this tree. Otherwise, if T_X does not adhere to the molecular clock hypothesis, then UPGMA may fail to reconstruct the tree correctly. For example, it cannot reconstruct the tree of Figure 51.

7.3.2 The ultrametric property

Before one applies the UPGMA algorithm to a given distance function, one needs to know if the distance function is additive w.r.t. a phylogenetic tree in which all leaves are at the same distance from the root. Here the ultrametric property is of particular relevance.

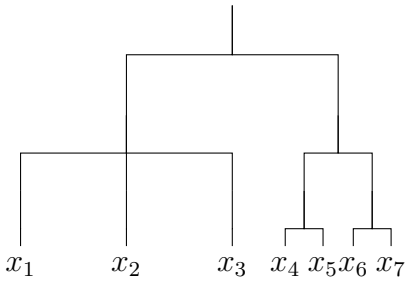
Definition 39 A distance function $d : X \times X \to \mathbb{R}_+^0$ is called an *ultrametric*, if for any triplet $(x, y, z) \in X \times X \times X$, the three distances $d(x, y)$, $d(x, z)$, and $d(y, z)$ have one of the following two properties:

(1) all three distances are equal

(2) two are equal and the remaining one is smaller

See also Figure 52, explaining the two ultrametric conditions (1) and (2).

Figure 52: A schematic phylogenetic tree, in which x_1, x_2, and x_3 satisfy condition (1) of Definition 39, and x_5, x_6, and x_7 satisfy condition (2) with $d(x_6, x_7) < d(x_5, x_6) = d(x_5, x_7)$.



Note that one can decide whether or not a distance function is ultrametric, according to Definition 39. If $|X| = n$, then one has to enumerate all n^3 triplets (x, y, z) of taxa and check for each of these whether condition (1) and (2) holds. This can be done in constant time. Hence the decision requires $O(n^3)$ time.

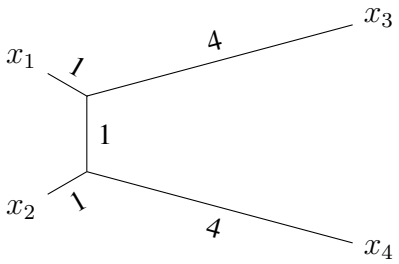
The next theorem relates the different notions introduced so far:

Theorem 10 Let $d : X \times X \to \mathbb{R}_+^0$ be a distance function that is additive w.r.t. tree T_X. Then d is ultrametric, if and only if every leaf in T_X has the same distance from the root. □

From the theorem we conclude that if d is additive w.r.t. some tree T_X and d is ultrametric, then every leaf in T_X has the same distance from the root. This implies that T_X can be reconstructed by the UPGMA-method. So it makes sense to check the additivity and the ultrametric property of the distance function, before applying the UPGMA-method. If both properties hold, the UPGMA will reconstruct the phylogenetic tree.

Figure 53: Suppose $X = \{x_1, x_2, x_3, x_4\}$ and distances derived from the tree below. Leaves x_1 and x_2 have minimum distance 3, but they are not neighbors in the tree. So the reconstructed tree based on first choosing taxa with minimum distance would be wrong.

	x_1	x_2	x_3	x_4
x_1	0	3	5	6
x_2		0	6	5
x_3			0	9
x_4				0



7.4 Neighbor-Joining

The most widely used distance based method to construct phylogenetic trees is the neighbor-joining method.It was introduced in Saitou & Nei [SN87]. Given an additive distance function d, neighbor-joining produces an unrooted phylogenetic tree T with edge lengths. It is especially suitable, when the rate of evolution of the separate subtree under consideration varies.

First, consider a tree T and let the distance function d be additive w.r.t. the leaves of T, i.e. d is a distance function defined on the leaves of T, obtained by adding edge lengths.

The neighbor-joining method is based on the fact that we can decide which nodes are neighboring without knowing the tree, but only using the distance function. However, it does not suffice simply to pick the two closest taxa, i.e. a pair (i, j) with $d(i, j)$ minimal, see Figure 53.
For the Neighbor-Joining method one defines

$$N_{i,j} = d(i, j) - (r_i + r_j)$$

where $r_i = \frac{1}{n-2} \sum_{x \in L} d(i, x)$, L is the current set of nodes in the tree, and $n = |L|$. r_i approximates the average distance to all other leaves: instead of dividing by n, one divides by $n - 2$.

Theorem 11 If d is additive w.r.t. some tree T, then the two leaves i and j for which $N_{i,j}$ is minimal are neighbors in T. □

This result ensures that the neighbor-joining algorithm will correctly reconstruct a tree from its additive distances. Unlike for the UPGMA-method, ultrametric properties (or something similar), are not required for the distance function.

Example 70 Let us illustrate this result using the tree of Figure 53.

$$
\begin{aligned}
r_{x_1} &= \frac{1}{2}(3+5+6) = 7 \\
r_{x_2} &= \frac{1}{2}(3+6+5) = 7 \\
r_{x_3} &= \frac{1}{2}(5+6+9) = 10 \\
r_{x_4} &= \frac{1}{2}(6+5+9) = 10
\end{aligned}
$$

$$
\begin{aligned}
\text{and } N_{x_1,x_2} &= d(x_1,x_2) - (r_{x_1} + r_{x_2}) = 3 - (7+7) = -11 \\
N_{x_1,x_3} &= d(x_1,x_3) - (r_{x_1} + r_{x_3}) = 5 - (7+10) = -12 \\
N_{x_1,x_4} &= d(x_1,x_4) - (r_{x_1} + r_{x_4}) = 6 - (7+10) = -11 \\
N_{x_2,x_3} &= d(x_2,x_3) - (r_{x_2} + r_{x_3}) = 6 - (7+10) = -11 \\
N_{x_2,x_4} &= d(x_2,x_4) - (r_{x_2} + r_{x_4}) = 5 - (7+10) = -12 \\
N_{x_3,x_4} &= d(x_3,x_4) - (r_{x_3} + r_{x_4}) = 9 - (10+10) = -11
\end{aligned}
$$

$$
\text{and so } N = \left\{ \begin{array}{c|cccc}
 & x_1 & x_2 & x_3 & x_4 \\
x_1 & 0 & -11 & -12 & -11 \\
x_2 & & 0 & -11 & -12 \\
x_3 & & & 0 & -11 \\
x_4 & & & & 0
\end{array} \right.
$$

The matrix N attains a minimum value for the pairs (x_1, x_3) and (x_2, x_4) and the corresponding leaves are indeed neighbors, as required.

The Neighbor-Joining Algorithm is specified in Algorithm 13.

Example 71 Let d_0 be given, as shown below. The following shows the different distance functions computed during the neighbor-joining algorithm, and the corresponding N-matrix

Algorithm 13 (Neighbor-Joining)

Input: Distance matrix d
Output: Phylogenetic tree T
Initialization: Let T consist of leaf nodes, one for each taxon. $L \leftarrow T$.
Iteration: Compute N from the current distance function. Pick a pair $i, j \in L$, s.t. $N_{i,j}$ is minimal. Define a new node k and set

$$d(k,m) = \frac{1}{2}(d(i,m) + d(j,m) - d(i,j)) \tag{7.2}$$

for all $m \in L \setminus \{i,j\}$. Add k to T and construct edges from k to i and to j. The edge length are:

$$\begin{aligned} d(i,k) &= \frac{1}{2}(d(i,j) + r_i - r_j) \\ d(j,k) &= d(i,j) - d(i,k) \end{aligned} \tag{7.3}$$

where $r_i = \frac{1}{n-2}\sum_{x \in L} d(i,x)$. Remove i and j from L and add k to L.
Termination: When L consists of only two elements i and j, add the remaining edge between i and j, with length $d(i,j)$.

computed from these.

$$d_0 = \begin{array}{c|cccc|c} & x_1 & x_2 & x_3 & x_4 & r_i \\ \hline x_1 & 0 & 3 & 5 & 6 & 7 \\ x_2 & & 0 & 6 & 5 & 7 \\ x_3 & & & 0 & 9 & 10 \\ x_4 & & & & 0 & 10 \end{array} \Rightarrow N^0 = \begin{array}{c|cccc} & x_1 & x_2 & x_3 & x_4 \\ \hline x_1 & & -11 & \underline{-12} & -11 \\ x_2 & & & -11 & -12 \\ x_3 & & & & -11 \\ x_4 & & & & \end{array}$$

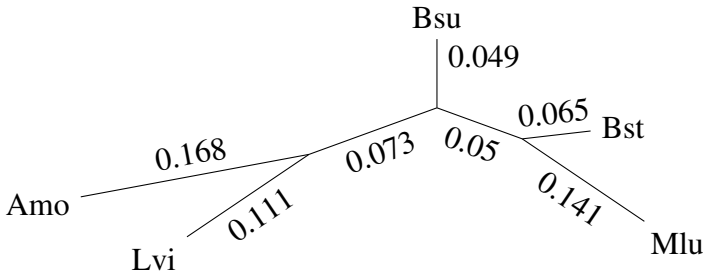
$$d_1 = \begin{array}{c|ccc|c} & x_1, x_3 & x_2 & x_4 & r_i \\ \hline \{x_1, x_3\} & 0 & 2 & 5 & 7 \\ x_2 & & 0 & 5 & 7 \\ x_4 & & & 0 & 10 \end{array} \Rightarrow N^1 = \begin{array}{c|ccc} & \{x_1, x_3\} & x_2 & x_4 \\ \hline \{x_1, x_3\} & & -12 & -12 \\ x_2 & & & \underline{-12} \\ x_4 & & & \end{array}$$

$$d_2 = \begin{array}{c|cc|c} & \{x_1, x_3\} & \{x_2, x_4\} & r_i \\ \hline \{x_1, x_3\} & 0 & 1 & 0 \\ \{x_2, x_4\} & & 0 & 0 \end{array}$$

Example 72 For the following distance function (which was already used in Example 68)

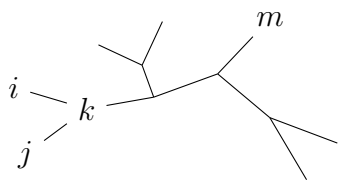
	Bsu	*Bst*	*Lvi*	*Amo*	*Mlu*
Bsu	0.0	0.1715	0.2147	0.3091	0.2326
Bst		0.0	0.2991	0.3399	0.2058
Lvi			0.0	0.2795	0.3943
Amo				0.0	0.4289
Mlu					0.0

the neighbor-joining algorithm delivers the following phylogenetic tree (with some edge length rounded):



Justification of update formula 7.2

Recall that k is the new parent for i and j and we need to compute the distance from k to any other leaf $m \notin \{i, j\}$, see the following illustration:



Given $d(i, m)$, $d(j, m)$, and $d(i, j)$, and due to the assumption that d is additive, we obtain

$$\begin{aligned} d(i,m) &= d(i,k) + d(k,m) \\ d(j,m) &= d(j,k) + d(k,m) \\ d(i,j) &= d(i,k) + d(j,k) \end{aligned}$$

$$\begin{aligned} \text{which implies } d(k,m) &= 0.5 \cdot 2 \cdot d(k,m) \\ &= 0.5 \cdot (d(i,m) - d(i,k) + d(j,m) - d(j,k)) \\ &= 0.5 \cdot (d(i,m) + d(j,m) - (d(i,k) + d(k,j)) \\ &= 0.5 \cdot (d(i,m) + d(j,m) - d(i,j)) \end{aligned}$$

This is how $d(k, m)$ is calculated in Equation (7.2) of Algorithm 13.

Justification of update formula 7.3

Let us now consider the update formula $d(i,k) = \frac{1}{2}(d(i,j) + r_i - r_j)$ in Algorithm 13. By definition,

$$\begin{aligned} r_i &= \frac{1}{n-2}\sum_{x\in L} d(i,x) \\ &= \frac{1}{n-2}\sum_{x\in L\setminus\{i,j\}} d(i,x) + \frac{1}{n-2}(d(i,j)+d(i,i)) \\ &= \underbrace{\frac{1}{n-2}\sum_{x\in L\setminus\{i,j\}} d(i,x)}_{q_i} + \frac{1}{n-2}d(i,j) \end{aligned}$$

In other words, r_i is the average distance q_i to all nodes (except for i and j) plus $\frac{1}{n-2}d(i,j)$.

$$\begin{aligned} \text{Now } r_i - r_j &= \frac{1}{n-2}\sum_{x\in L\setminus\{i,j\}} d(i,x) + \frac{1}{n-2}d(i,j) - \left(\frac{1}{n-2}\sum_{x\in L\setminus\{j,i\}} d(j,x) + \frac{1}{n-2}d(j,i)\right) \\ &= \frac{1}{n-2}\sum_{x\in L\setminus\{i,j\}} d(i,x) - \frac{1}{n-2}\sum_{x\in L\setminus\{j,i\}} d(j,x) + \frac{1}{n-2}d(i,j) - \frac{1}{n-2}d(i,j) \\ &= \frac{1}{n-2}\sum_{x\in L\setminus\{i,j\}} d(i,x) - \frac{1}{n-2}\sum_{x\in L\setminus\{j,i\}} d(j,x) \\ &= q_i - q_j \end{aligned}$$

Hence we obtain

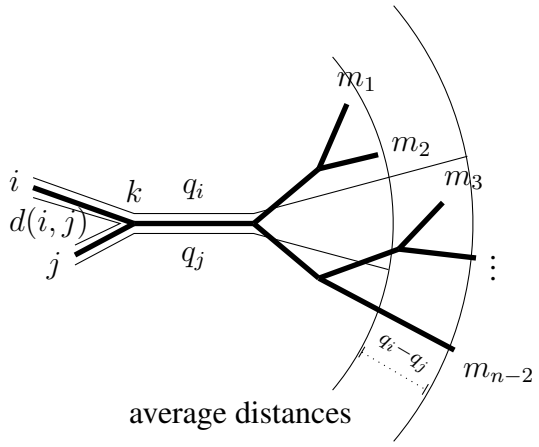
$$d(i,k) = \frac{1}{2}(d(i,j) + r_i - r_j) = \frac{1}{2}(d(i,j) + q_i - q_j)$$

That is, $d(i,k)$ is half of the distance from i to j plus the difference of the average distances to i and to j, see Figure 54 for a graphical explanation.

7.4.1 Application of Neighbor-Joining

Given an additive distance function $d : X \times X \to \mathbb{R}^0_+$, neighbor-joining is guaranteed to reconstruct a phylogenetic tree T_X, such that d is additive w.r.t. T_X. Unfortunately, in practice we are usually not given a distance function that is additive, but rather the distance function is usually obtained very indirectly by comparing sequence data generated along the tree. Such data is rarely additive. Nevertheless, the neighboring-joining method is often applied to such data and has proven to be a fast, useful and robust tree reconstruction method.

Figure 54: Determining $d(i,k) = \frac{1}{2}(d(i,j) + q_i - q_j)$.
The thick edges represent the tree consisting of the n nodes $\{i, j, m_1, \ldots, m_{n-2}\}$. The thin straight lines represent the distances involved in the computation of $d(i,k)$. As j is closer to k, the average distance q_j is smaller than q_i and so the thin straight line labeled by q_j ends on the inner circle, while the thin straight line labeled by q_i ends on the outer circle. The difference between q_i and q_j is shown as a dotted line between the circles.



Bibliography

[AG96] S.F. Altschul and W. Gish. Local alignment statistics. *Methods Enzymol*, 266:460–80, 1996.

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

[BKC+15] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James Drake, Jane M Landolin, and Adam M Phillippy. Assembling large genomes with single-molecule sequencing and locality sensitive hashing. *Nat. Biotech.*, 33(6), 2015.

[Bun74] P. Buneman. A Note on the Metric Properties of Trees. *Journal of Combinatorial Theory (B)*, 17:48–50, 1974.

[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.

[DKGDG15] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. KMC 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, MAY 15 2015.

[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.

[FRBM12] N.A. Fonseca, J. Rung, A. Brazma, and J.C. Marioni. Tools for mapping high-throughput sequencing data. *Bioinformatics*, 28(24):3169–3177, 2012.

[Got82] O. Gotoh. An Improved Algorithm for Matching Biological Sequences. *JMB*, 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.

[JDK+17] Chirag Jain, Alexander Dilthey, Sergey Koren, Srinivas Aluru, and Adam M Phillippy. A fast approximate algorithm for mapping long reads to large reference databases. In *International Conference on Research in Computational Molecular Biology*, pages 66–81. Springer, 2017.

[JRRP+18] Chirag Jain, Luis M Rodriguez-R, Adam M Phillippy, Konstantinos T Konstantinidis, and Srinivas Aluru. High throughput ANI analysis of 90K prokaryotic genomes reveals clear species boundaries. *Nature communications*, 9(1):5114, 2018.

[JRZ+20] Chirag Jain, Arang Rhie, Haowen Zhang, Claudia Chu, Brian P. Walenz, Sergey Koren, and Adam M. Phillippy. Weighted minimizer sampling improves long read mapping. *Bioinformatics (Oxford, England)*, 36(1):i111–i118, 2020.

[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. *PNAS*, 87:2264–2268, 1990.

[KDD17] Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. Kmc 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, 2017.

[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.

[Laq81] H.T. Laquer. Asymptotic limits for a two-dimensional recursion. *Stud. Appl. Math.*, 64:271–277, 1981.

[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.

[MSPK19] Guillaume Marçais, Brad Solomon, Rob Patro, and Carl Kingsford. Sketching and Sublinear Data Structures in Genomics. *Annual Review of Biomedical Data Science*, 2(1):93–118, 2019.

[Mur84] F. Murtagh. Complexities of Hierarchic Clustering Algorithms: the state of the art. *Computational Statistics Quarterly*, 1:101–113, 1984.

[Mye91] E.W. Myers. An Overview of Sequence Comparison Algorithms in Molecular Biology. Technical Report TR 91-29, TUCSON, 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. *JMB*, 48:443–453, 1970.

[Ohl13] E. Ohlebusch. *Bioinformatics Algorithms Sequence Analysis, Genome Rearrangements and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.

[OSS+19] Brian D. Ondov, Gabriel J. Starrett, Anna Sappington, Aleksandra Kostic, Sergey Koren, Christopher B. Buck, and Adam M. Phillippy. Mash screen: high-throughput sequence containment estimation for genome discovery. *Genome Biology*, 20(1):232, 2019.

[OTM+16] Brian D Ondov, Todd J Treangen, Páll Melsted, Adam B Mallonee, Nicholas H Bergman, Sergey Koren, and Adam M Phillippy. Mash: fast genome and metagenome distance estimation using minhash. *Genome Biology*, 17(1):1–14, 2016.

[PAD99] D.R. Powell, L. Allison, and T.I. Dix. A Versatile Divide and Conquer Technique for Optimal String Alignment. *IPL*, 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.

[Sel80] P.H. Sellers. The Theory and Computation of Evolutionary Distances: Pattern Recognition. *JALG*, 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.

[SM58] R. Sokal and C. Michener. A statistical method for evaluating systematic relationships. *University of Kansas Science Bulletin*, 38:1409–1438, 1958.

[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.

[SN87] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Mol. Biol. Evol.*, 4(4):406–425, Jul 1987.

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

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

[Ukk92] E. Ukkonen. Approximate String-Matching with q-Grams and Maximal Matches. *TCS*, 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. *JACM*, 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.