Sequence Analysis 1

Lecture notes

Faculty of Technology, Bielefeld University

Winter 2025/26

Preface

The lecture notes at hand are a re-structured, partially re-written and extended version of previous ones by Robert Giegerich, Stefan Kurtz, Enno Ohlebusch, Sven Rahmann and myself, over the years supported by several helping hands: Eyla Willing, Peter Husemann, Roland Wittler, Katharina Klerx, Linda Sundermann, Karsten Wüllems, Tizian Schulz, Michel T. Henrichs, Daniel Dörr, Marília D. V. Braga and Leonard Bohnenkämper.

This is the first of two parts, after the former "Sequence Analysis" class was split in the acacdemic year 2025/26 into two (maybe three) smaller study modules. It should be, to some degree, suitable for self study.

Jens Stoye, September 2025

Contents

1 Overview

At Bielefeld University, elements of sequence analysis are taught in several courses, starting with elementary pattern matching methods in "Algorithms and Data Structures" in the second semester. The present two-hour course "Sequence Analysis 1" is taught in the third semester and is continued by "Sequence Analysis 2" in the fourth semester. Occasionally, also a continuation "Sequence Analysis 3" is taught, covering extra fun and experimental material.

1.1 Prerequisites

It is assumed that the student has had some exposure to algorithms and mathematics, as well as to elementary facts of molecular biology. The following topics are useful, although not absolutely required, to understand the material here:

- exact string matching algorithms (e.g. the naive algorithm, Boyer-Moore, Boyer-Moore-Horspool, Knuth-Morris-Pratt),
- comparison-based sorting (e.g. insertion sort, mergesort, heapsort, quicksort),
- asymptotic complexity analysis (O notation).

The first two do not appear in these notes. Asymptotic complexity is briefly reviewed in Section 2.2.

1.2 Application Areas of Sequence Analysis

Sequences (or texts, strings, words, etc.) over a finite alphabet are a natural way to encode information. The following incomplete list presents some application areas of sequences of different kinds.

- Molecular biology (the focus of this course):

Molecule	Example	Alphabet	Length
DNA	...AACGACGT...	4 nucleotides	$\approx 10^3$–10^9
RNA	...AUCGGCUU...	4 nucleotides	$\approx 10^2$–10^3
Proteins	...LISAISTNETT...	20 amino acids	$\approx 10^2$–10^3

  The main idea behind biological sequence comparison is that an evolutionary relationship implies structural and functional similarity, which again implies sequence similarity.

- Phonetics:

  English: 40 phonemes
  Japanese: 113 "morae" (syllables)

- Audio files of spoken language, bird song, etc.: discrete multidimensional data (e.g. frequency, energy) over time
- Graphics: An image is a two-dimensional "sequence" of (r, g, b)-vectors with $r, g, b \in [0, 255]$ to encode color intensities for red, green and blue, of a screen pixel.
- Text processing: Texts are encoded (ASCII, Unicode) sequences of numbers.
- Information transmission: A sender sends binary digits (bits) over a (possibly noisy) channel to a receiver.
- Internet, web pages, any database of text or pseudo-textual information.

1.3 A Small Selection of Problems on Sequences

The "inform" in (bio-)informatics comes from information. Google states that their mission is "to organize the world's information and make it universally accessible and useful"[1]. Information is often stored and addressed sequentially. Therefore, to process information, we need to be able to process sequences. Here is a small selection of problems on sequences, although not all of them will be covered in this basic course.

1. Sequence comparison: Quantify the (dis-)similarity between two or more sequences and point out where they are particularly similar or different.
2. Exact string matching: Find all positions in a sequence (called the *text*) where another sequence (the *pattern*) occurs.
3. Approximate string matching: As exact string matching, but allow some *differences* between the pattern and its occurrence in the text.
4. Multiple (approximate) string matching: As above, but locate all positions where any pattern of a given *pattern set* occurs. Often more elegant and efficient methods are available than searching for each pattern separately.
5. Regular expression matching: Find all positions in a text that match an expression constructed according to specific rules.
6. Approximate dictionary search: Given a word w, find the most similar word to w in a given set of words (the dictionary). This is useful for correcting spelling errors.
7. Repeat discovery: Find long repeated parts of a sequence. This is useful for data compression, but also in genome analysis.
8. Data compression: Reduce the amount of memory used to store some text data set.

[1] `https://about.google/company-info`

9. Pattern discovery: Find all interesting parts of a sequence (this of course depends on your definition of *interesting*); this may concern surprisingly frequent subwords, tandem repeats, palindromes, unique subwords, etc.
10. Revision and change tracking: Compare different versions of a document, highlight their changes, produce a "patch" that succinctly encodes commands that transform one version into another. Version control systems like Subversion or Git are based on efficient algorithms for such tasks.
11. Error-correcting and re-synchronizing codes: During information transmission, the channel may be noisy, i.e., some of the bits may be changed, or sender and receiver may get out of sync. Therefore error-correcting and synchronizing codes for bit sequences have been developed. Problems are the development of new codes, and efficient encoding and decoding algorithms.

1.4 Selection of topics

The material selected for this two-hour course covers only parts of the above-mentioned topics. The expectation is that after this course the basic and most relevant problems in sequence analysis are understood, especially if the focus is on applications in bioinformatics. The continuation "Sequence Analysis 2" will cover additional topics, and deepen some of those that we can not cover here in full detail.

Some even more advanced material will still be left over, and might be discussed in a "Sequence Analysis 3" class, if there is interest. These include:

- index construction in linear time
- fast algorithms for approximate string matching (e.g. bit-shifting),
- advanced filtering methods for approximate string matching (e.g. gapped q-grams),
- efficient methods for advanced score functions in pairwise alignment,
- combinatorics on sequences.

1.5 Suggested Reading

Details about the recommended textbooks can be found in the bibliography. We suggest that students take note of the following ones.

- Gusfield (1997) published one of the first textbooks on sequence analysis. Nowadays, some of the algorithms described therein have been replaced by better and simpler ones. A revised edition would be very much appreciated, but it is still the fundamental reference for sequence analysis courses.
- Setubal and Meidanis (1997) give a good compilation of well explained sequence analysis algorithms in computational biology, among other topics.

- Another good sequence analysis book that places more emphasis on probabilistic models was written by Durbin et al. (1998).
- An even more mathematical style can be found in the book by Waterman et al. (2005).
- A more textual and less formal approach to sequence analysis is presented by Mount (2004). This book covers a lot of ground in bioinformatics and is a useful companion until the Master's degree.
- For the practically inclined who want to learn about the actual tools that implement some of the algorithms discussed in this course, the above book or the "Dummies" book by Claverie and Notredame (2007) is recommended.
- Finally, the classic algorithms textbook of Cormen et al. (2001) should be part of every computer science student's personal library. While it does not cover much of sequence analysis, it is a useful reference to look up elementary data structures, O notation, basic probability theory. It also contains a chapter on dynamic programming.

2 Basic Definitions

2.1 Sets and Basic Combinatorics

Sets of numbers. The following commonly known sets of numbers are of special interest for the topics taught in this course.

- $\mathbb{N} := \{1, 2, 3, \dots\}$ is the set of natural numbers.
- $\mathbb{N}_0 := \{0\} \cup \mathbb{N}$ additionally includes zero.
- $\mathbb{Z} := \{0, 1, -1, 2, -2, 3, -3, \dots\}$ is the set of integers.
- $\mathbb{R}$ ($\mathbb{R}_0^+$) is the set of (nonnegative) real numbers.

The **absolute value** or **modulus** of a number x is its distance from the origin and denoted by $|x|$, e.g. $|-5| = |5| = 5$.

An **interval** is a set of consecutive numbers and written as follows:

- $[a, b] := \{x \in \mathbb{R} : a \leq x \leq b\}$ (closed interval),
- $[a, b[:= \{x \in \mathbb{R} : a \leq x < b\}$ (half-open interval),
- $]a, b] := \{x \in \mathbb{R} : a < x \leq b\}$ (half-open interval),
- $]a, b[:= \{x \in \mathbb{R} : a < x < b\}$ (open interval).

Sometimes the interval notation is used for integers, too, especially when we talk about indices. So, for $a, b \in \mathbb{Z}$, $[a, b]$ may also mean the integer set $\{a, a + 1, \dots, b - 1, b\}$.

Elementary set combinatorics. Let S be any set. Then $|S|$ denotes the **cardinality** of S, i. e., the number of elements contained in S. We symbolically write $|S| := \infty$ if the number of elements is not a finite number.

With $\mathcal{P}(S)$ or 2^S we denote the **power set** of S, i. e., the set of all subsets of S. For each element of S, there are two choices if a subset is formed: it can be included or not. Thus the number of different subsets is $|\mathcal{P}(S)| = |2^S| = 2^{|S|}$.

To more specifically compute the number of k-element subsets of an n-element set, we introduce the following notation:

- $n^k := n \cdot \ \dots \ \cdot n$ (ordinary **power**, k factors),
- $n! := n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1$ (**factorial**).

When choosing k elements out of n, we have n choices for the first element, $n-1$ choices for the second one, and so on. To disregard the order among these k elements, we divide by the number of possible rearrangements or **permutations** of k elements; by the same argument as above these are $k!$ many. It follows that there are $n \cdot (n-1) \cdot \ldots \cdot (n-k+1)/k!$ different k-element subsets of an n-element set. This motivates the following definition of a **binomial coefficient**:

- $\binom{n}{k} := \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{k!} = \frac{n!}{(n-k)! \cdot k!}$ (read as "n choose k").

2.2 Asymptotics

We will analyze several algorithms during this course. In order to formalize statements such as "the running time increases quadratically with the sequence length", we review the asymptotic "**big-O notation**" here, also known as **Landau symbols**.

Let $f, g : \mathbb{N} \to \mathbb{R}_0^+$ be functions.

***O*(·):** We write $f(n) \in O(g(n))$ (or $f(n) = O(g(n))$, even though this is not an equality) if there exist $n_0 \in \mathbb{N}$ and $c > 0$ such that $f(n) \leq c \cdot g(n)$ for all $n \geq n_0$ (i. e., **for eventually all** n). In fact, $O(f(n))$ stands for the whole class of functions that grow at most as fast as $f(n)$, apart from constant factors and smaller order terms.

A function in $O(n^c)$ for some constant $c \in \mathbb{N}$ is said to be of **polynomial growth**; it does not need to be a polynomial itself, e.g. $n^{2.5} + \log \log n$. A function in $O(c^n)$ for constants $c > 1$ is said to be of **exponential growth**.

It is suggested to remember the following hierarchy of $O(\cdot)$-classes and extend it as needed.

$$O(1) \subset O(\log n) \subset O(\sqrt{n}) \subset O(n/\log n) \subset O(n) \subset O(n \log n) \subset O(n\sqrt{n}) \subset O(n^2) \subset O(n^3) \subset O(n^{\log n}) \subset O(2^n) \subset O(3^n) \subset O(n!) \subset O(n^n)$$

2.3 Graph Theory

Graphs and networks have an important role in many different areas, such as sequence analysis and bioinformatics in general. Metabolic networks and phylogenetic trees are just two examples and there are many more outside of this field, like electronic circuits and transport or communication networks. Some basics of graph theory will be introduced in this section.

A **graph** is a pair $G = (V, E)$ consisting of a set of **vertices** V and a set of **edges** E. The definitions of V and E vary and depend on the type of graph they build. In the following, the most basic graph types and some associated important definitions will be explained.

Undirected graph. Here V can be any set and $E \subseteq \binom{V}{2}$. The set $\binom{V}{2}$ refers to all subsets[1] of size two of V, i. e., all $\{u, v\}$, where $u, v \in V$ and $u \neq v$. If $e = \{u, v\} \in E$ is an edge connecting vertices u and v, then e is said to be **incident** to u and to v, and the vertices u and v are said to be **adjacent**. The **degree** of a vertex v is the number of edges that are incident to v. A **path** is a sequence of n vertices $(v_1, \ldots, v_n)$, where v_i and v_{i+1} are adjacent, for all $i = 1, \ldots, n-1$.

Representing edges as sets implies that there is no order applied – in contrast to tuples, which are used in the following paragraph.

Directed graph. Here V can be any set and $E \subseteq V \times V$. A directed graph is also called a **digraph**. In directed graphs, an edge (u, v) is interpreted as a link[2] from vertex u to vertex v. This relation is often written as $u \to v$. Furthermore, u can be referred to as the **source** and v can be referred to as the **target** of the edge (u, v). For a directed graph one distinguishes the **in-degree** and the **out-degree** of a vertex v, which determines the number of incoming and outgoing edges, i. e., edges that have v as target or as source, respectively. A vertex in a directed graph is **balanced** if its in-degree equals its out-degree. A **directed path** is a path $(v_1, \ldots, v_n)$, where v_i is linked to v_{i+1} by an edge $v_i \to v_{i+1}$, for all $i = 1, \ldots, n-1$.

More about paths and cycles. Given a path $p = (v_1, v_2, \ldots, v_n)$, its **length** is denoted by $\ell(p)$ and corresponds to the number of edges along its way. Clearly we have $\ell(p) = n-1$. The path p is **simple** if all vertices except possibly the first and the last one are distinct. If the first and the last vertices are the same, we call p a **cycle**. A graph that contains at least one cycle is called **cyclic**, otherwise it is called **acyclic**.

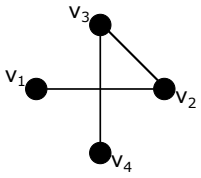
Connectivity. Two vertices u and v are **connected** if there exists a path p that starts in u and ends in v. An undirected graph $G = (V, E)$ is **connected** if every two vertices $u, v \in V$ are connected, otherwise it is **disconnected**. A directed graph $G = (V, E)$ is **weakly connected** or simply **connected** if its underlying undirected variant is connected, otherwise it is disconnected. And it is **strongly connected** if for every two vertices $\{u, v\}$ there exist a directed path from u to v and another directed path from v to u.

Examples of undirected and directed graphs are given in Figure 2.1.

Eulerian path. An **Eulerian path** in a graph $G = (V, E)$ is a path that uses every edge in E exactly once. Note that an Eulerian path is not necessarily simple and can visit each vertex multiple times. Its first vertex is called **initial vertex** or **source** and its last vertex is called **final vertex** or **sink**. If source and sink are the same vertex we have a so called **Eulerian cycle**.

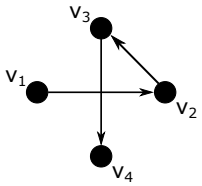
[1] This definition of E for an undirected graph G forbids *parallel* edges (connecting the same pair of vertices) and *loops* (connecting a vertex to itself), implying that G is a *simple* graph.

[2] This definition of E for a digraph forbids *parallel* edges (linking the same pair of vertices in the same order), but allows *loops* (linking a vertex to itself). If necessary, an additional restriction forbidding loops can be assumed.



node	degree
v_1	1
v_2	2
v_3	2
v_4	1

(a) An undirected graph.



node	in-degree	out-degree	balanced?
v_1	0	1	no
v_2	1	1	yes
v_3	1	1	yes
v_4	1	0	no

(b) A directed graph.

Figure 2.1: Examples of directed and undirected graphs.

A fundamental theorem of graph theory says that an Eulerian path exists in a connected graph G if and only if G is **balanced** as we describe below.

A connected *undirected* graph is balanced if one of the two following conditions is true:

C1. All except two vertices have even degree. In this case, we assign arbitrarily one of the two vertices of odd degree to be the source, and the other one to be the sink.

C2. All vertices have even degree. In this case, source and sink must be the same vertex, and it can be chosen arbitrarily.

For a connected *directed* graph, the two possible conditions for being balanced are:

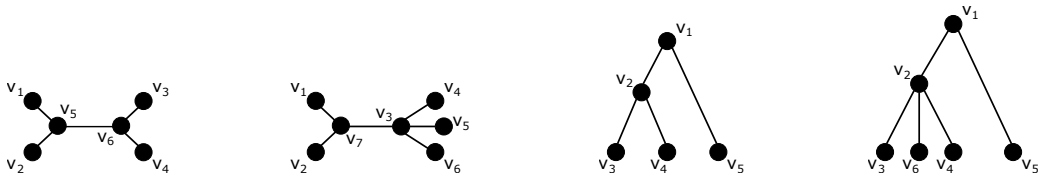
C1. All except two vertices are balanced. One of the unbalanced vertices must be the source and has exactly one more outgoing edge than incoming edges. The other unbalanced vertex must be the sink and has exactly one more incoming edge than outgoing edges.

C2. All vertices are balanced. In this case, source and sink must be the same vertex, and it can be chosen arbitrarily.

Hamiltonian path. A **Hamiltonian path** in a graph $G = (E, V)$ is a path that visits each vertex in V exactly once. If the start vertex and the end vertex of a Hamiltonian path are adjacent, then by adding that connecting edge one gets a *Hamiltonian cycle*, i.e. a cycle in G that visits each vertex. The problem of deciding whether a given graph contains a Hamiltonian cycle or not is a very prominent task in graph theory and computer science in general.

Trees. An **unrooted tree** is a connected, acyclic, undirected graph. Each vertex with a degree of one is called a **leaf** (terminal node). All other nodes are called **internal nodes**. An unrooted tree can be either **binary**, when its nodes have degree at most three, or **multifurcating** otherwise.

In a **rooted tree** one of the vertices is distinguished from the others and is called the **root**. Rooting a tree induces a hierarchical relationship of the nodes and creates a directed graph, since rooting implies a direction for each edge (by definition always pointing away from the root). The terms **parent**, **child**, **sibling**, **ancestor**, **descendant** are then defined in the obvious way. Rooting a tree also changes the notion of the degree of a node. First



(a) unrooted and binary (b) unrooted and multifurcating (c) rooted and binary (d) rooted and multifurcating

Figure 2.2: Trees of four different types.

note that the root has in-degree zero and every other node has in-degree one. Only the out-degree of the nodes may vary, therefore the **degree of a node in a rooted tree** refers to the **out-degree** of that node. Then, a leaf is defined as a node of (out-)degree zero. A rooted tree can be either **binary**, when its nodes have (out-)degree at most two, or **multifurcating** otherwise. The **depth** of node v in a rooted tree, denoted depth(v), is the length of the (unique) simple path from the root to v. The **depth of a rooted tree** T is the maximum depth of all of T's nodes. The **width** of a certain depth d of a rooted tree is the number of nodes in T whose depth is d. The **width of a rooted tree** T is the maximum width among all depths.

The removal of any edge divides (splits) a tree into two connected components. Given a node v other than the root in a rooted tree, the **subtree rooted** at v is the remaining tree after removing the edge that ends at v and the component containing the root. (The subtree rooted at the root is the complete, original tree.)

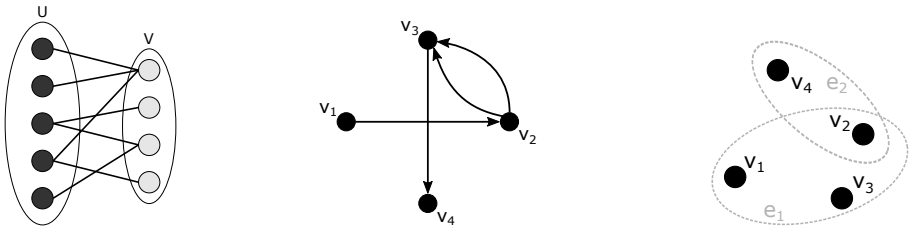
Examples of trees are given in Figure 2.2.

Bipartite graph. In this graph, that is often represented as $G = (U \cup V, E)$, the vertices are partitioned into two disjoint sets U and V, also called **partitions**. There exist only edges between vertices of different partitions, i. e., no edge connects a pair of vertices belonging to the same partition. More formally this means that, for all $e \in E$, it holds that $|e \cap U| = |e \cap V| = 1$. An example of a bipartite graph is shown in Figure 2.3 (a).

Multigraph. This type of graph has **parallel edges**, connecting the same pair of vertices. In directed multigraphs, parallel edges have the same direction (link the same source to the same target). An example of a multigraph is shown in Figure 2.3 (b).

Hypergraph. In this type of graph an edge can connect any number of vertices instead of just two. Such edges are called **hyperedges**. An example of a hypergraph is shown in Figure 2.3 (c).

Weights. Graphs can be enhanced with **weights**. They can be **vertex-weighted**, **edge-weighted** or both. Weights are specified by a weighting function $W_V : V \to \mathbb{R}$ or $W_E : E \to \mathbb{R}$, respectively. Weights can be used to symbolize many different numerical or even ordinal relationships. In case of vertices, this could be a minimum score to access



(a) A bipartite graph has its vertices partitioned into two sets U and V.

(b) A directed multigraph. Two parallel edges (pointing in the same direction) link vertices v_2 to v_3.

(c) A hypergraph whose hyperedges are e_1 and e_2. The hyperedge e_1 contains the vertices v_1, v_2 and v_3, while hyperedge e_2 contains v_2 and v_4.

Figure 2.3: Graphs of three different special types.

the corresponding vertex. For edges, it could stand for a cost or path length to travel from one vertex to another. For an edge-weighted graph, we denote by $w(p)$ the **weight** (or **length**) of a path $p = (v_1, v_2, \ldots, v_n)$, defined as the sum of its edge weights, i. e., $w(p) = \sum_{i=1}^{n-1} W_E\big(v_i, v_{i+1}\big)$. For example, the famous **Traveling Salesperson Problem** (TSP) asks in an edge-weighted graph for a shortest (lowest-weight) Hamiltonian cycle (if it exists).

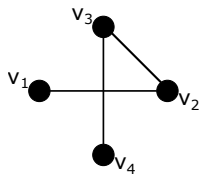
Labels. In a **labeled graph** vertices and/or edges are annotated with different kinds of labels. One example of a labeled graph is the *suffix tree*, which will be part of this lecture (see Chapter 10).

Representation. There are mainly three different forms to represent a graph $G = (V, E)$. The *edge list* is a list of all edges within G. The *adjacency matrix* is a $|V| \times |V|$ matrix with entries indicating whether an edge exists between any pair of vertices or not. For an unweighted graph, this matrix is binary. If the graph is weighted, the entries in the matrix usually represent the weights of the corresponding edges. *Adjacency lists* are used to store, for each vertex in V, a list of their adjacent vertices. See Figure 2.4 for examples.

2.4 Alphabets and Sequences

Alphabets. A finite **alphabet** is simply a finite set; we use Σ as the symbol for an alphabet. The elements of Σ are called **characters**, **letters**, or **symbols**. Here are some examples:

- the DNA alphabet {`A`,`C`,`G`,`T`} (adenine, cytosine, guanine, thymine);
- the puRine / pYrimidine alphabet for DNA {`R`, `Y`} (`R` = `A` or `G`; `Y` = `T` or `C`);



(a) Example of a graph G.

$[(v_1,v_2), (v_2,v_3), (v_3,v_4)]$

(b) Example of an edge list for G. The structure of G is represented as one list of all its edges.

	v_1	v_2	v_3	v_4
v_1	0	1	0	0
v_2	1	0	1	0
v_3	0	1	0	1
v_4	0	0	1	0

(c) Example of a binary adjacency matrix for G. As G has four vertices, the matrix size is 4×4. Each 1 represents an edge between the corresponding vertices in G.

v_1: v_2
v_2: v_1, v_3
v_3: v_2, v_4
v_4: v_3

(d) Example of adjacency lists for G. For each vertex in G there is a list that contains all vertices that are adjacent to it.

Figure 2.4: An undirected graph and three common forms of its representation.

- the amino acid one-letter code $\{\texttt{A}, \ldots, \texttt{Y}\} \setminus \{\texttt{B}, \texttt{J}, \texttt{O}, \texttt{U}, \texttt{X}\}$, see `http://en.wikipedia.org/wiki/List_of_standard_amino_acids` for more information about the individual amino acids;
- the HydrophObe / hydroPhIle alphabet $\{\texttt{H}, \texttt{P}\}$ or $\{\texttt{O}, \texttt{I}\}$;
- the positive / negative charge alphabet $\{\texttt{+}, \texttt{-}\}$;
- the IUPAC codes for DNA sequences ($\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}, \texttt{U}, \texttt{R}, \texttt{Y}, \texttt{M}, \texttt{K}, \texttt{W}, \texttt{S}, \texttt{B}, \texttt{D}, \texttt{H}, \texttt{V}, \texttt{N}\}$) and for protein sequences ($\{\texttt{A}, \ldots, \texttt{Z}\} \setminus \{\texttt{J}, \texttt{O}, \texttt{U}\}$), see `http://www.bioinformatics.org/sms/iupac.html`;
- the ASCII (American Standard Code for Information Interchange) alphabet, a 7-bit encoding (0-127) of commonly used characters in computer systems (see `http://en.wikipedia.org/wiki/ASCII`);
- the alphanumeric subset of the ASCII alphabet $\{\texttt{0}, \ldots, \texttt{9}, \texttt{A}, \ldots, \texttt{Z}, \texttt{a}, \ldots, \texttt{z}\}$; encoded by the numbers 48-57, 65-90 and 97-122, respectively.

These examples show that alphabets may have very different sizes. The **alphabet size** $\sigma := |\Sigma|$ is often an important parameter when we analyze the complexity of algorithms on sequences.

Sequences. A **sequence** (also called **string** or **word**) s over an alphabet Σ is represented as $s = s[1]s[2]\ldots s[n] = (s[1], s[2], \ldots, s[n])$, where, for $i = 1, \ldots, n$, each symbol $s[i] \in \Sigma$. The number of symbols in the sequence s, which here corresponds to n, is the **length** of s, and can be directly denoted by $|s|$. We denote by $\overleftarrow{s}$ the **reversal** of s, that is obtained by reverting the order of its symbols: $\overleftarrow{s} = s[n]s[n-1]\ldots s[1]$. The **empty sequence**, denoted by ε, consists of no symbols and has length 0. By the **concatenation**

(or **juxtaposition**) of two or more sequences we can obtain a longer sequence, whose length is the sum of the lengths of the concatenated sequences.

Let Σ^n be the set of all sequences whose length is n and consist of symbols from Σ. Note[3] that $\Sigma^0 = \{\varepsilon\}$ for any alphabet Σ. For $n \geq 1$ we can write $\Sigma^n = \{xa \mid x \in \Sigma^{n-1}, a \in \Sigma\}$. We further denote by Σ^* (resp. Σ^+) the set of all (resp. all nonempty) sequences over Σ:

$$\Sigma^* = \bigcup_{n \geq 0} \Sigma^n \qquad \text{and} \qquad \Sigma^+ = \bigcup_{n \geq 1} \Sigma^n.$$

Substrings. Let a sequence s be written as a concatenation $s = uvw$. Each of the (possibly empty) sequences u, v and w is a **substring** (or **subword**) of s. Furthermore, the substring u is a **prefix** of s and the substring w is a **suffix** of s. A substring of s that is distinct from s is said to be **proper**[4]. If we can write $s = uv$ and $s' = vw$ with $u, v, w \in \Sigma^*$, then we say that s and s' have an **overlap** of length $|v|$. Note that there always exists an overlap of length 0 between any pair of sequences.

The substring from position i to position j of s is denoted by $s[i \mathinner{..} j] = s[i]s[i+1] \dots s[j]$, assuming that $i \leq j$. If $i > j$, by convention we define $s[i \mathinner{..} j] = \varepsilon$. We say that a substring w **occurs** at position i in s if $s[i \mathinner{..} j] = w$ (in this case, obviously $j = i + |w| - 1$).

Given a sequence s and an integer k, with $0 \leq k \leq |s|$, a ***k*-mer** or ***k*-gram**[5] of s is a substring of s of length k. Note that at each of the positions from the set $\{1, 2, \dots, |s| - k + 1\}$ there is an occurrence of a k-mer in s.

Subsequences. While a sub*string* is a contiguous part of a sequence, a sub*sequence* does not need to be contiguous: Given a sequence s and m positions $i_1, i_2, \dots, i_m$, such that $1 \leq i_1 < i_2 < \dots < i_m \leq |s|$, the sequence $s[i_1, i_2, \dots, i_m] = s[i_1]s[i_2] \dots s[i_m]$ is called a **subsequence** of s.

Each substring is also a subsequence, but the converse is not generally true. For example, `ABB` is both a subsequence and a substring (even a prefix) of `ABBAB`, while `BBB` is a subsequence but not a substring.

Counting substrings and subsequences. Let s be a sequence with $|s| = n$. A nonempty substring of s is specified by its starting position i and ending position j with $1 \leq i \leq j \leq n$. This gives $n - k + 1$ substrings of length k occurring in s. In total the number of nonempty substrings occurring in s is $1 + 2 + \dots + n = \binom{n+1}{2} = (n+1) \cdot n/2$. We might further argue that the empty substring occurs $n + 1$ times in s (before the first character and after each of the n characters), so including the empty strings, there are $\binom{n+2}{2}$ ways of selecting a substring.

[3] Σ^0 is the unitary set consisting of the empty sequence (not the empty set, nor the empty sequence itself).

[4] A *proper substring* that is also a *prefix* (resp. *suffix*) is a *proper prefix* (resp. *proper suffix*).

[5] The term "gram" is probably related to the Greek word $\gamma\rho\acute{\alpha}\mu\mu\alpha$ (gramma), meaning letter. Usually the literature that adopts the term "gram" denotes its size by q (*q-gram*) instead of k (*k-mer*).

A (possibly empty) subsequence of s is specified by any selection of positions. Thus there are 2^n possibilities to specify a subsequence, which is exponential in n. There are $\binom{n}{k}$ possibilities to specify a subsequence of length k.

In both of the above cases, the substrings or subsequences obtained by different selections do not need to be different. For example if $s = \texttt{AAA}$, then $s[1 \ldots 2] = s[2 \ldots 3] = \texttt{AA}$. It is an interesting problem (for which this course will provide efficient methods) to compute the number of *distinct* substrings or subsequences of a given sequence s. In order to appreciate these efficient methods, the reader is invited to think about algorithms for solving these problems before continuing!

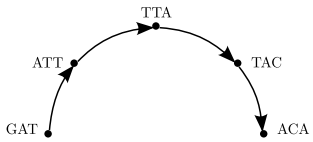
Words with the same q-gram profile and the de Bruijn subgraph. Given a sequence s and a positive integer $q \leq |s|$, we want to determine whether there are other sequences that are distinct but have the same q-gram profile as s. And, if there are, how many? For $q = 1$, the problem is trivial. Indeed, if a sequence t that is distinct from s corresponds to a permutation of the symbols of s, then s and t clearly share the same 1-gram profile. For $q \geq 2$, an elegant answer can be found with the help of the following graph.

Given a sequence s and a positive integer q, such that $2 \leq q \leq |s|$, the **de Bruijn subgraph** $B(s, q) = (V, E)$ is a directed multigraph. The set of vertices V is simply composed of all $(q-1)$-grams of s. The set of edges E is derived from the multiset of q-grams of s as follows. Let i be an integer iterated from 1 to $|s| - q + 1$. For each i, let the corresponding q-gram be denoted as $s_{q,i} = s[i \ldots i + q - 1] = aub$, where a and b are symbols and $|u| = q - 2$; we then add an edge to E, linking vertex au to vertex ub. Note that if the same q-gram occurs multiple times in s, say m times, we have m parallel edges occurring in the multigraph $B(s, q)$.

The construction of $B(s, q)$ forms an Eulerian path p_s, represented here by its sequence of edges $s_{q,1}, s_{q,2}, \ldots, s_{q,|s|-q+1}$. Consequently, $B(s, q)$ is *connected* and *balanced* and might contain even more than one Eulerian path. In fact, there is a one-to-one correspondence between Eulerian paths in $B(s, q)$ and sequences sharing the same q-gram profile. In other words, a sequence t that is distinct but has the same q-gram profile as s exists if and only if $B(s, q)$ has another Eulerian path p_t corresponding to the sequence of q-grams of t. Since p_t uses the same edges as p_s in a different order, the path p_t can only exist if $B(s, q)$ contains a cycle. Note, however, that the presence of a cycle is not a sufficient condition. Indeed, it is possible that $B(s, q)$ contains one or more cycles, but still admits only one Eulerian path. Some examples are given in Figures 2.5 and 2.6.

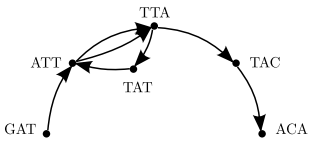
We summarize the observations above in the following theorem.

Theorem 1 Given a sequence s and an integer $q \geq 2$, the number of distinct sequences that have the same q-gram profile as s equals the number of distinct Eulerian paths in $B(s, q)$. If $B(s, q)$ is acyclic, it has a single Eulerian path and no sequence t exists that is distinct but has the same q-gram profile as s.



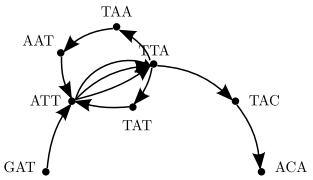
$s =$ `GATTACA` and $q = 4$

(a) Since this graph contains no cycle, it has only one Eulerian path. There can be no other word sharing the same 4-gram profile.



$s =$ `GATTATTACA` and $q = 4$

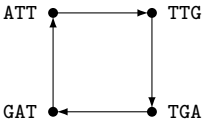
(b) Here the graph contains two cycles, but still only one Eulerian path. There is no other word sharing the same 4-gram profile.



$s =$ `GATTATTAATTACA` and $q = 4$

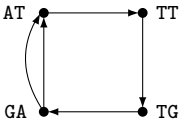
(c) This graph contains cycles and two distinct Eulerian paths. The other word sharing the same 4-gram profile is `GATTAATTATTACA`.

Figure 2.5: Examples of de Bruijn subgraphs for distinct sequences and $q = 4$.



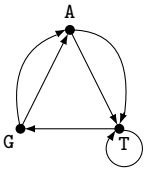
$s =$ `GATTGAT` and $q = 4$

(a) This graph has an Eulerian cycle starting at any of its four vertices. Therefore, three other sequences share the same 4-gram profile: `ATTGATT`, `TTGATTG` and `TGATTGA`.



$s =$ `GATTGAT` and $q = 3$

(b) This graph has two cycles, but only one Eulerian path. There is no other word sharing the same 3-gram profile.



$s =$ `GATTGAT` and $q = 2$

(c) This graph contains cycles and two distinct Eulerian paths. The other word sharing the same 2-gram profile is `GATGATT`.

Figure 2.6: De Bruijn subgraphs for the same sequence but with q varying from 4 to 2.

2.5 Maximality and minimality

For identifying a *maximum* pattern in some object, it is often useful to first find **maximal** candidates: these are the parts that fulfill the pattern and cannot be extended in any way. The **maximum** fulfilling parts are then the largest among the maximal ones. For example, consider the set of integers $S = \{1, 4, 5, 6, 8, 9, 11, 13, 14, 15\}$ and suppose we are searching for subsets of at least two elements composed of consecutive values only. Here the list of subsets fulfilling the search criteria is $\{4, 5\}$, $\{4, 5, 6\}$, $\{5, 6\}$, $\{8, 9\}$, $\{13, 14\}$, $\{13, 14, 15\}$, $\{14, 15\}$. If we restrict to *maximal* subsets, we have $\{4, 5, 6\}$, $\{8, 9\}$, $\{13, 14, 15\}$. And if we finally restrict to *maximum* subsets, we find the subsets $\{4, 5, 6\}$, $\{13, 14, 15\}$.

Analogously, for identifying a *minimum* pattern in some object, it is often useful to first find **minimal** candidates: these are the parts that fulfill the pattern and cannot be pruned in any way. The **minimum** fulfilling parts are then the smallest among the minimal ones. For example, consider the sequence $s =$ `ABABBA` and suppose we are searching for its *unique* substrings. These are the substrings of s that occur exactly once in s. Here the set of

unique substrings is $U = \{\texttt{ABABBA}, \texttt{ABABB}, \texttt{ABAB}, \texttt{ABA}, \texttt{BABBA}, \texttt{BABB}, \texttt{BAB}, \texttt{ABBA}, \texttt{ABB}, \texttt{BBA}, \texttt{BB}\}$. If we then restrict to *minimal* unique substrings, we have $U' = \{\texttt{ABA}, \texttt{BAB}, \texttt{BB}\}$ (for any other candidate x in the set U, at least one of the three elements of U' is a substring of x). And if we finally restrict to *minimum* unique substrings[6], we have only BB.

2.6 Review of Elementary Probability Theory

A little understanding about probabilities is required to follow some of the subsequent chapters. A **probability vector** is a (row) vector with nonnegative entries whose sum equals 1. A matrix of several probability vectors on top of each other is called a **stochastic matrix** (naturally, its rows sum to 1). Relative frequencies can often be interpreted as probabilities.

Often, we will associate a probability or frequency with every letter in an alphabet $\Sigma = \{a_1, \ldots, a_\sigma\}$ by specifying a probability vector $p = (p_1, \ldots, p_\sigma)$. If $p_1 = \cdots = p_\sigma = 1/\sigma$, we speak of the **uniform distribution** on Σ. Letter frequencies allow us to define a notion of **random text** or **independent and identically distributed (i.i.d.) text**, where each letter a_k appears according to its frequency p_k at any text position, independently of the other positions. Then, for a *fixed* length q, the probability that a random word X of length q equals a given $x = x_1 \ldots x_q$, is

$$\mathbb{P}(X = x) = \prod_{i=1}^{q} p_{k(x_i)},$$

where $k(c)$ is the index k of the letter c such that $c = a_k \in \Sigma$. In particular, for the uniform distribution, we have that $\mathbb{P}(X = x) = 1/\sigma^q$ for all words $x \in \Sigma^q$.

How many times do we see a given word x of length q as a substring of a random text T of length $n + q - 1$? That depends on T, of course; so we can only make a statement about expected values and probabilities. At each starting position $i = 1, \ldots, n$, the probability that the next q letters equal x is given by $p_x := \mathbb{P}(X = x)$ as computed above. Let us call this a *success*. The expected number of successes is then $n \cdot p_x$.

But what is precisely the probability to have exactly k successes? This is a surprisingly hard question, because successes at consecutive positions in the text are not independent: If x starts at position 17, it cannot also start at position 18 (unless it is the q-fold repetition of the same letter). However, if the word length is $q = 1$ (i. e., x is a single letter), we can make a stronger statement. Then the words starting at consecutive positions do not overlap and successes become independent of each other. Each text with k successes (each with probability p_x) also has $n - k$ non-successes (each with probability $1 - p_x$). Therefore each such text has probability $p_x^k \cdot (1 - p_x)^{n-k}$. There are $\binom{n}{k}$ possibilities to choose the k out of n positions where the successes occur. Therefore, if K denotes the random variable counting successes, we have

$$\mathbb{P}(K = k) = \binom{n}{k} \cdot p_x^k \cdot (1 - p_x)^{n-k}$$

[6] An efficient solution for this particular problem does not require the enumeration of all candidates as we did here and will be given in Chapter 11.

for $k \in \{0, 1, \ldots, n\}$, and $\mathbb{P}(K = k) = 0$ otherwise. This distribution is known as the **Binomial distribution** with parameters n and p_x. As said above, it specifies probabilities for the number k of successes in n *independent* trials, where each success has the same probability p_x.

When n is very large and p is very small, but their product (the expected number of successes) is $np = \lambda > 0$, the above probability can be approximated by

$$\mathbb{P}(K = k) \approx \frac{\mathrm{e}^{-\lambda} \cdot \lambda^k}{k!},$$

as a transition to the limit ($n \to \infty$, $p \to 0$, $np \to \lambda$) shows. (You may try to prove this as an exercise.) This distribution is called the **Poisson distribution** with parameter λ which is the expected value as well as the variance. It is often a good approximation when we count the (random) number of certain rare events.

Example 2 Let $s, t \in \Sigma^n$ and $|\Sigma| = \sigma$. In the i.i.d. model, what is the probability that s and t contain a particular character a the same number of times?

The probability that a is contained k times in s is $\binom{n}{k} \cdot \left(\frac{1}{\sigma}\right)^k \cdot \left(1 - \frac{1}{\sigma}\right)^{n-k}$. The same applies to t. Thus the probability that both s and t contain a exactly k times is the square of the above expression. Since k is arbitrary between 0 and n, we have to sum the probabilities over all k. Thus the answer is $\sum_{k=0}^{n} \left(\binom{n}{k} \cdot \frac{1}{\sigma^k} \cdot \left(1 - \frac{1}{\sigma}\right)^{n-k}\right)^2$. ◀

3 Distances Between Sequences

3.1 Problem Motivation

The trivial method to compare two sequences is to compare them character by character: x and y are **equal** if and only if $|x| = |y|$ and $x_i = y_i$ for all $i \in [1, |x|]$. However, many problems are more subtle than simply deciding whether two sequences are equal or not. Some examples are

- searching for a name of which the spelling is not exactly known,
- finding diffracted forms of a word,
- accounting for typing errors,
- tolerating error prone experimental measurements,
- allowing for redundancy in the genetic code, with 64 codons corresponding to 20 amino acids; for example, `GCA`, `GCC`, `GCG` and `GCU` (in short, `GCN`) all code for alanine,
- looking for a protein sequence with known biological function that is similar to a given protein sequence with unknown function.

Therefore, we would like to define a notion of distance between sequences that takes the value zero if and only if the sequences are equal and otherwise gives a quantification of their differences. This quantity may depend very much on the application! The first step is thus to compile some properties that *every* distance should satisfy.

3.2 Definition of a Metric

In mathematical terms, a distance function is often called a **metric**, but also simply **distance**. Given any set $\mathcal{X}$, a metric on $\mathcal{X}$ is a function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ that assigns a number (their distance) to any pair (x, y) of elements of $\mathcal{X}$ and satisfies the following properties:

$$d(x, y) = 0 \text{ if and only if } x = y \qquad \text{(identity of indiscernibles)} \qquad (3.1)$$

$$d(x, y) = d(y, x) \text{ for all } x \text{ and } y \qquad \text{(symmetry)} \qquad (3.2)$$

$$d(x, y) \leq d(x, z) + d(z, y) \text{ for all } x, y \text{ and } z \qquad \text{(triangle inequality)} \qquad (3.3)$$

From (3.1)–(3.3), it follows that:

$$d(x, y) \geq 0 \text{ for all } x \text{ and } y \qquad \text{(nonnegativity)} \qquad (3.4)$$

(a) Triangle inequality (b) Ultrametric triangle inequality

Figure 3.1: Illustrations of the triangle inequality and the ultrametric triangle inequality.

The pair $(\mathcal{X}, d)$ is called a **metric space**.

If only (3.2) and (3.3) hold and also $d(x, x) = 0$ for all x, but there are $x \neq y$ with $d(x, y) = 0$, we call d a **pseudo-metric**. A pseudo-metric on $\mathcal{X}$ can be turned into a true metric on a different set $\mathcal{X}'$, where each set of elements with distance zero from each other is contracted into a single element.

While a pseudo-metric is somewhat weaker (more general) than a metric, there are also more special variants of metrics. An interesting one is an **ultra-metric** which satisfies (3.1) and (3.2), and the following stronger version of the triangle inequality:

$$d(x, y) \leq \max\{d(x, z),\, d(z, y)\} \text{ for all } x,\, y \text{ and } z \quad \text{(ultrametric triangle inequality)}$$

It is particularly important in phylogenetics.

Figure 3.1 visually compares the triangle inequality (a) and the ultrametric triangle inequality (b).

3.3 Transformation Distances

A **transformation distance** $d(x, y)$ on a set $\mathcal{X}$ is defined as the minimum number of allowed operations needed to transform one element x into another y. The allowed operations characterize the distance.

The following can be said about the metric properties of an arbitrary transformation distance: The identity of indiscernibles is often easy to verify. Care has to be taken that the allowed operations imply the symmetry condition of the defined metric. The triangle inequality is always satisfied since the distance is defined as the *minimum* number of operations required. (Be sure to understand this!)

Depending on the set $\mathcal{X}$ and the allowed operations, a transformation distance may be easy or hard to compute.

3.4 A Very Simple Metric on Sequences of the Same Length

Take an alphabet Σ and consider two sequences $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$, both of length n, over Σ, i.e. $x, y \in \Sigma^n$. The following function is a metric distance:

$$d_H(x, y) := \sum_{i=1}^{n} \mathbb{1}_{\{x_i \neq y_i\}} \qquad \textbf{(Hamming distance)}$$

where the **indicator function** $\mathbb{1}_{\{\text{cond}\}}$ equals 1 if condition cond is true, and equals 0 otherwise.

Simply speaking, the Hamming distance counts in how many of their n positions the two sequences x and y differ:

Example 3 Let $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$,

$$\begin{aligned} x &= \mathtt{GATT\underline{T}ACA\underline{G}TAGGTC\underline{C}} \text{ and} \\ y &= \mathtt{GATT\underline{C}ACA\underline{C}TAGGTC\underline{A}}. \end{aligned}$$

The two sequences differ in the three positions 5, 9 and 16 (underlined). Therefore, their Hamming distance is $d_H(x, y) = 3$. ◀

Example 4 Given is an alphabet Σ of size σ. Consider a graph whose vertices are all strings in Σ^n, the so-called *sequence graph* of dimension n. Two vertices are connected by an edge if and only if the Hamming Distance between them is 1. How many edges does the graph contain?

There are exactly $n(\sigma - 1)$ strings at Hamming distance 1 of any given string (n positions times $(\sigma - 1)$ choices of differing characters). Thus there are $n(\sigma - 1)$ edges incident to each vertex. Obviously there are σ^n vertices. Since an edge is connected to two vertices, there are $\sigma^n \cdot n \cdot (\sigma - 1)/2$ edges altogether. ◀

The distance measure introduced in this section only make sense for sequences of the same length. The distances we consider next are also defined for sequences of different lengths.

3.5 Edit Distances for Sequences

Let Σ be a finite alphabet. Edit distances are a general class of transformation distances on Σ^*, defined by **edit operations**. The distance is defined as the minimum number of edit operations needed to re-write a source sequence $x \in \Sigma^*$ into a target sequence $y \in \Sigma^*$.

The edit operations that we consider are

- $\mathtt{C}$: *copy* the next character from x to y,
- $\mathtt{S}_c$ for each $c \in \Sigma$: *substitute* the next character from x by c in y,
- $\mathtt{I}_c$: *insert* c at the current position in y,

- `D`: *delete* the next character from x (i.e., skip over it),

The insert and delete operations are sometimes collectively called **indel** operations.

The following table lists some edit distances along with their operations and their associated **unit costs**. If a cost is infinite, the operation is not permitted.

Name		`C`	$\mathtt{S}_c$	$\mathtt{I}_c$	`D`
Hamming distance	d_H	0	1	∞	∞
LCS distance	d_{LCS}	0	∞	1	1
Edit distance[1]	d	0	1	1	1

Each distance definition must allow copying at zero cost to ensure $d(x, x) = 0$ for all x. All other operations receive a non-negative cost. Here are some subtle points to note:

- Previously, the Hamming distance $d_H(x, y)$ was undefined if $|x| \neq |y|$. With this definition, we could define it as ∞. This distinction is irrelevant in practice.
- The LCS distance (longest common subsequence distance) does not allow substitutions. However, since each substitution can be simulated by one deletion and one insertion, we don't need to assign infinite cost. Any cost ≥ 2 will lead to the same distance value. The name stems from the fact that $d_{LCS}(x, y) = |x| + |y| - 2 \cdot LCS(x, y)$, where $LCS(x, y)$ denotes the length of a longest common subsequence of x and y.
- The edit distance should be more precisely called *standard unit cost edit distance*. It is the most important one in practice and should be carefully remembered.

We can now explain a bit more formally how $x \in \Sigma^*$ is transformed into a sequence y by an **edit sequence**, i.e., a sequence over the **edit alphabet** $\mathcal{E}(\Sigma) := \{\mathtt{C}, \mathtt{S}_c, \mathtt{I}_c, \mathtt{D}\}$ (or an appropriate subset of it, depending on which of the above distances we use): Let $x \in \Sigma^*$ and $e \in \mathcal{E}(\Sigma)^*$. We define the **edit function** $E : \Sigma^* \times \mathcal{E}(\Sigma)^* \to \Sigma^*$ as follows. For $E(x, e)$, the elements of e are applied from left to right to the letters of x, producing a new target sequence y. At the end of the process, x must be exhausted, otherwise e does not fit to x.

Example 5 Let $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ and $x = \mathtt{GATTTTA}$. When applying the edit sequence $e = \mathtt{CCDCCS_AI_CC}$ to x, we get the new sequence $y = \mathtt{GATTACA}$, as can be verified easily. ◀

We define the **cost** of a sequence of edit operations as the sum of the costs of the individual operations according to the table above. With this at hand we can define the **edit distance** for a set of edit operations $\mathcal{E}$ as

$$d(x, y) := \min\{\text{cost}(e) : e \in \mathcal{E}^*, E(x, e) = y\}.$$

Theorem 6 Each of the three distance variations according to the table above (Hamming distance, LCS distance, Edit distance) defines a metric.

[1] The (unit cost) edit distance is also called *Levenshtein distance*, see (Levenshtein, 1966).

Proof. We need to verify that all conditions of a metric are satisfied. The identity of indiscernibles is easy. The triangle inequality is also trivial because the distance is a transformation distance (see Section 3.3). We need to make sure that symmetry holds, however. This can be seen by considering an optimal edit sequence e with $E(x, e) = y$. We shall show that there exists an edit sequence f with $\text{cost}(f) = \text{cost}(e)$ such that $E(y, f) = x$.

To obtain f with the desired properties, we replace in e each `D` by an appropriate $\texttt{I}_c$ and vice versa. Substitutions $\texttt{S}_c$ remain substitutions, but we need to replace the substituted character from y by the correct character from x. A copy `C` remains a copy. The edit sequence f constructed in this way obviously satisfies $\text{cost}(f) = \text{cost}(e)$ and $E(y, f) = x$. This shows that $d(y, x) \leq d(x, y)$.

To prove equality, we apply the argument twice and obtain $d(x, y) \leq d(y, x) \leq d(x, y)$. Since the first and last term are equal, all inequalities must in fact be equalities, proving the symmetry property. □

Invariance properties of the edit distance. We note that the edit distance is invariant under sequence reversal and bijective maps of the alphabet ("renaming the symbols"):

Theorem 7 Let d denote any variant of the above edit distances; let $\pi : \Sigma \to \Sigma'$ be a bijective map between alphabets (if $\Sigma' = \Sigma$, then π is a permutation of Σ). We extend π to strings over Σ by changing each symbol separately. Then, clearly $d(x, y) = d(\pi(x), \pi(y))$. Furthermore, $d(x, y) = d(\overleftarrow{x}, \overleftarrow{y})$.

Proof. Left as an exercise. Hint: Consider reversibility of edit operations. □

This theorem immediately implies that two DNA sequences have the same edit distance as their reverse complements.

4 Pairwise Sequence Alignment

4.1 Definition of Alignment

An edit sequence $e \in \mathcal{E}^*$ describes in detail how a sequence x can be transformed into another sequence y, but it is hard to visualize the exact relationship between single characters of x and y when looking at e. An equivalent description that is visually more useful is an alignment of x and y.

We first give an example: Take $x = \texttt{AAB}$ and $e = \texttt{I}_\texttt{B}\texttt{CS}_\texttt{B}\texttt{D}$; then $E(x, e) = \texttt{BAB} =: y$. Where does the first `B` in y come from? It has been inserted, so it is not related to the first `A` in x. The `A` in y is a copy of the first `A` from x. The last `B` in y has been created by substituting the second `A` in x by `B`. Finally, the last `B` in x has no corresponding character in y, since it was deleted. We write these relationships in the following way, called **alignment**.

$$\begin{pmatrix} - & \texttt{A} & \texttt{A} & \texttt{B} \\ \texttt{B} & \texttt{A} & \texttt{B} & - \end{pmatrix}$$

We see that an alignment consists of (vertical) pairs of symbols from Σ, representing copies or substitutions, or pairs of a symbol from Σ and a **gap character** (–), indicating an insertion or deletion. Note that a pair of gap characters is not possible. Formally, the **alignment alphabet** for Σ is defined as

$$\mathcal{A} \equiv \mathcal{A}(\Sigma) := (\Sigma \cup \{-\})^2 \setminus \left\{\binom{-}{-}\right\}$$

and an alignment is a finite sequence over $\mathcal{A}$.

More informally, an alignment can be seen as a matrix of characters from $\Sigma \cup \{-\}$ with two rows and an arbitrary number of columns, where the column $\binom{-}{-}$ is forbidden. In this matrix, for $i \in \{1, 2\}$, the ***i*-th row** equals the i-th sequence, if the included gaps are omitted. Each **column** of the matrix (alignment) corresponds to one of the edit operations. If one of the characters of a column is a gap character, it is called an **indel** column. A column of the form $\binom{a}{a}$ for $a \in \Sigma$ is called a **match** column (or copy column if we want to stick to the edit sequence terminology). Consequently, a column of the form $\binom{a}{b}$ for $a \neq b$ is called a **mismatch** column (substitution column).

There is an obvious one-to-one relationship between edit sequences that transform x into y and alignments of x and y. The edit operation `C` corresponds to $\binom{a}{a}$ for $a \in \Sigma$, $\texttt{S}_b$ when applied to a character a corresponds to $\binom{a}{b}$ for $a, b \in \Sigma$ with $a \neq b$, $\texttt{I}_a$ corresponds to $\binom{-}{a}$ for $a \in \Sigma$, and `D` corresponds to $\binom{a}{-}$ for $a \in \Sigma$.

Observation 8 Let $x \in \Sigma^m$, $y \in \Sigma^n$, and let A be an alignment of x and y. Let e be the edit sequence corresponding to A. Then

$$\max\{m, n\} \leq |A| = |e| \leq m + n.$$

Proof. The equality $|A| = |e|$ follows from the equivalence of edit sequences and alignments. Since each edit operation (alignment column) consumes at least one character from x or y and the column $\binom{-}{-}$ is forbidden, their number is bounded by $m+n$. The maximum is reached if only insertions and deletions (indel columns) are used. On the other hand, $m = |x| \leq |A|$ since the length of A is at least the number of non-gap characters in the first row of A. Similarly $n \leq |A|$, so that $|A| \geq \max\{m, n\}$. The boundary case is reached if the maximum number of copy and substitution operations (match/mismatch columns) and only the minimum number $\max\{m, n\} - \min\{m, n\}$ of indels is used. □

Definition 9 For the edit distance model, the **cost of an alignment column** is defined as the cost of the corresponding edit operation. More precisely: The match columns have a cost of 0 (like the copy operation), whereas the mismatch and indel columns have a cost of 1 (like the substitute and indel operations), when using unit costs. The **cost of an alignment** $A = (A_1, A_2, \ldots, A_n)$ is defined as the sum of its columns' cost, i.e., $\text{cost}(A) = \sum_{i=1}^{n} \text{cost}(A_i)$.

Definition 10 The **alignment distance** of two sequences $x, y \in \Sigma^*$ is defined as

$$d(x, y) := \min\{\text{cost}(A) : A \in \mathcal{A}^* \text{ is an alignment of } x \text{ and } y\}.$$

The **cost-minimizing alignments** are given by the set

$$A^{\text{opt}}(x, y) := \{A \in \mathcal{A}^* \text{ is an alignment of } x \text{ and } y \text{ and } \text{cost}(A) = d(x, y)\}.$$

When the cost function is clear from the context, a cost-minimizing alignment is often also called an **optimal alignment**.

Note the following subtle point: The alignment cost of two sequences should not be confused with the cost of a particular alignment of those sequences!

Problem 11 (Alignment Problem) For two given strings $x, y \in \Sigma^*$ and a given cost function, find the alignment distance of x and y and one or all optimal alignment(s).

4.2 An Efficient Algorithm to Compute Optimal Alignments

While it is easy to find *some* alignment $A \in \mathcal{A}^*$ of two given sequences $x \in \Sigma^*$ and $y \in \Sigma^*$, it may not be obvious how to find an *optimal* alignment of x and y. One simple way could be to enumerate all possible alignments of x and y, compute the cost of each of them, and then choose one of minimum cost. However, since the number of alignments grows exponentially (Waterman, 1995, Section 9.1), it is infeasible in practice to enumerate them all. Instead, the following **dynamic programming** approach appears promising.

We define a $(|x|+1) \times (|y|+1)$ matrix D, called the **alignment matrix** (or **edit matrix**), by

$$D(i, j) := d(x[1 \ldots i], y[1 \ldots j]) \qquad \text{for } 0 \leq i \leq |x|,\ 0 \leq j \leq |y|.$$

We are obviously looking for $D(|x|, |y|) = d(x, y)$. We shall point out how the distance of short prefixes can be used to compute the distance of longer prefixes; for concreteness we focus on standard unit costs.

If $i = 0$, we are transforming $x[1 \dots 0] = \varepsilon$ into $y[1 \dots j]$. This is only possible with j insertions, which together have cost j. Thus $D(0, j) = j$ for $0 \leq j \leq |y|$. Similarly $D(i, 0) = i$ for $0 \leq i \leq |x|$. The interesting question is how to obtain the values $D(i, j)$ for the remaining pairs of (i, j).

Theorem 12 For $1 \leq i \leq |x|$, $1 \leq j \leq |y|$,

$$D(i,j) = \min \begin{cases} D(i-1, j-1) + \mathbb{1}_{\{x[i] \neq y[j]\}}, \\ D(i-1, j) + 1, \\ D(i, j-1) + 1. \end{cases}$$

Proof. The proof is by induction on $i + j$. The basis is given by the initializations mentioned above. By induction hypothesis we may assume that the theorem correctly computes $D(i-1, j-1) = \min\{\text{cost}(A) \mid A \text{ is an optimal alignment of } x[1 \dots i-1] \text{ and } y[1 \dots j-1]\}$, so let $A_{\nwarrow}$ be an optimal alignment for this case. Similarly, let $A_{\uparrow}$ be optimal for $D(i-1, j) = \min\{\text{cost}(A) \mid A \text{ is an optimal alignment of } x[1 \dots i-1] \text{ and } y[1 \dots j]\}$ and $A_{\leftarrow}$ be optimal for $D(i, j-1) = \min\{\text{cost}(A) \mid A \text{ is an optimal alignment of } x[1 \dots i] \text{ and } y[1 \dots j-1]\}$.

We obtain three candidates for alignments to transform $x[1 \dots i]$ into $y[1 \dots j]$ as follows: (1) We extend $A_{\nwarrow}$ by a match or mismatch column $\binom{x[i]}{y[j]}$, depending whether $x[i]$ and $y[j]$ are equal or not; in the former case the cost remains unchanged, in the latter case it is increased by 1. (2) We extend $A_{\uparrow}$ by a delete column $\binom{x[i]}{-}$, increasing the cost by 1. (3) We extend $A_{\leftarrow}$ by an insert column $\binom{-}{y[j]}$, increasing the cost by 1.

Recall that $D(i, j)$ is defined as the minimum cost of any alignment of $x[1 \dots i]$ and $y[1 \dots j]$. Since we can pick the best of the three candidates, we have shown that inequality $\leq$ holds in the theorem.

It remains to be shown that there can be no other alignment with lower cost. Assume such an alignment A^* exists and consider its last column C and its prefix A' such that $A^* = A'C$. If C is a deletion column, then A' is an edit sequence transforming $x[1 \dots i-1]$ into $y[1 \dots j]$ with $\text{cost}(A^*) = \text{cost}(A') + 1 < D(i-1, j) + 1$ by assumption; thus $\text{cost}(A') < D(i-1, j)$, a contradiction, since $D(i-1, j)$ is the optimal cost by inductive assumption. The other options for o lead to similar contradictions: In each case, we would have seen a better value already at a previous element of D. Therefore, such an A^* cannot exist. □

It is important to understand that the above proof consists of two parts. The first part shows that, naturally, by appropriate extension of the corresponding alignments, $D(i, j)$ is *at most* the minimum of three values. The second part shows that the optimal alignment for $D(i, j)$ is always one of those three possibilities.

Together with the initialization, Theorem 12 can be translated immediately into an algorithm to compute the alignment distance. Some care must be taken, though:

- One might get the idea to implement a recursive function D that first checks for a boundary case and returns the appropriate value, or otherwise calls D recursively with smaller arguments. This is highly inefficient! Intermediate results would be computed again and again many times. For example, to compute $D(6, 6)$, we need $D(5, 5)$, $D(5, 6)$ and $D(6, 5)$, but to compute $D(5, 6)$, we also need $D(5, 5)$.

- It is thus much better to fill the matrix iteratively, either row by row, or column by column, or diagonal by diagonal where $i + j$ is constant. This is called a **pull-type** or **backward dynamic programming algorithm**. To obtain the value of $D(i, j)$, it pulls the information from previously computed cells.

Example 13 Let $x =$ `BCACD` and $y =$ `DBADAD`. We compute the edit matrix D as follows:

	y	ε	D	B	A	D	A	D
x		0	1	2	3	4	5	6
ε	0	0	1	2	3	4	5	6
B	1	1	1	1	2	3	4	5
C	2	2	2	2	2	3	4	5
A	3	3	3	3	2	3	3	4
C	4	4	4	4	3	3	4	4
D	5	5	4	5	4	3	4	4

Hence the edit distance of x and y is $d(x, y) = D(5, 6) = 4$. ◀

Let us analyze the time and space complexity of the pull-type algorithm. Assuming $|x| = m$ and $|y| = n$, we need to compute $(m + 1) \cdot (n + 1)$ values, and each computation takes constant time; thus the running time is $O(mn)$. The space complexity also appears to be $O(mn)$ since we apparently need to store the matrix D. However, it is sufficient to keep, for instance in a column-wise computation, just the previous column, or in a row-wise computation, just the previous row, in order to calculate the next one; thus the space complexity is $O(m + n)$.

Reconstructing an optimal alignment. The above statement about the space complexity is true if we are only interested in the distance value. Often, however, we also would like to know an optimal alignment. We essentially get it for free by the above algorithm, but we need to remember which one of the three cases was chosen in each cell of the matrix; therefore the space complexity increases to $O(mn)$.

More precisely, to find an optimal alignment let us define the **backtracing matrix** E of the same dimensions as D. We compute E while computing D, such that $E(i, j)$ contains the last edit operation of an optimal edit sequence that transforms $x[1 \dots i]$ into $y[1 \dots j]$. This information is available whenever we make a decision for one of the three predecessors to compute $D(i, j)$. In fact, there may be several optimal possibilities for $E(i, j)$. We can use an array of three bits to store any combination of the following seven possibilities: {{`M`}, {`D`}, {`I`}, {`M`, `D`}, {`M`, `I`}, {`D`, `I`}, {`M`, `D`, `I`}}, where `M` stands for *match*, `D` stands for *delete* and `I` stands for *insert* column..

From E, an optimal alignment A can be constructed backwards by **backtracing** (not to be confused with backtracking as explaind on the facing page) where we use the stored operations as a *trace* which we follow through the matrix E. Clearly A ends with one of the column types stored in $E(m, n)$, so we have determined the last column C of A. In case of `M`, a match $\binom{a}{a}$ or a mismatch column $\binom{a}{b}$ is generated, in case of a `D`, a delete column $\binom{a}{-}$ is generated, and in case of an `I` an insert column $\binom{-}{a}$ is generated. Then we continue as follows.

- If $C = \texttt{M}$, continue with $E(m-1, n-1)$.
- If $C = \texttt{D}$, continue with $E(m-1, n)$.
- if $C = \texttt{I}$, continue with $E(m, n-1)$.

We repeat this process until we arrive at $E(0,0)$. Note that in the first row we always move left, and in the first column we always move up.

Example 13 (cont'd) For the example edit matrix above ($x = \texttt{BCACD}$ and $y = \texttt{DBADAD}$), the backtracing matrix looks as follows, where the three bits "*pqr*" represent $p = \texttt{M}$ (match/diagonal), $q = \texttt{D}$ (delete/vertical) and $r = \texttt{I}$ (insert/horizontal).

	y	ε	D	B	A	D	A	D
x		0	1	2	3	4	5	6
ε	0	000	001	001	001	001	001	001
B	1	010	100	100	001	001	001	001
C	2	010	110	110	100	101	101	101
A	3	010	110	110	100	101	101	001
C	4	010	110	110	010	100	111	100
D	5	010	100	111	010	100	101	100

Backtracing starts at the bottom right corner, where the backtracing matrix has the entry $E(5,6) = 100$. Therefore all optimal alignments end with a match column $\binom{\texttt{D}}{\texttt{D}}$. The algorithm proceeds with entry $E(4,5) = 111$, which means that all three predecessors could have produced the maximum value $D(4,5) = 4$. Here we pick the horizontal one, corresponding to an insertion column $\binom{-}{\texttt{A}}$, and then move one field to the left where we find the backtracing matrix entry $E(4,4) = 010$. This means we go another diagonal step (mismatch column $\binom{\texttt{C}}{\texttt{D}}$). After that we can proceed with three more diagonal steps until we reach the top-left corner of the matrix. When we proceed this way (alternatives exist), then we arrive at the following optimal alignment:

$$\begin{pmatrix} \texttt{B} & \texttt{C} & \texttt{A} & \texttt{C} & - & \texttt{D} \\ \texttt{D} & \texttt{B} & \texttt{A} & \texttt{D} & \texttt{A} & \texttt{D} \end{pmatrix}$$

◀

If we want to obtain *all* optimal edit sequences, we systematically have to take different branches in the E-matrix whenever there are several possibilities. This can be done efficiently by a technique called **backtracking**, which finds systematically all solutions (not to be confused with backtracing mentioned on the preceding page): We push the coordinates of each branching E-entry (along with information about which branch we took and the length of the partial edit sequence) onto a stack during backtracing. When we reach $E(0,0)$, we have completed an optimal edit sequence and report it. Then we go back to the last branching point by popping its coordinates from the stack, remove the appropriate prefix from the edit sequence, and construct a new one by taking the next branch (pushing the new information back onto the stack) if there is still one that we have not taken. If not, we backtrack further. As soon as the stack is empty, the backtracking ends.

Problem 14 (All Optimal Alignments) Enumerate all optimal edit sequences (all optimal alignments) of two sequences $x \in \Sigma^m$ and $y \in \Sigma^n$.

Now it is easy to derive the details to prove the following:

Theorem 15 Problem All Optimal Alignments can be solved in $O(mn + z)$ time, where z is the total length of all optimal edit sequences of x and y.

Example 13 (cont'd) Our example $x =$ `BCACD` and $y =$ `DBADAD` contains six additional optimal alignments that can be obtained by choosing alternative routes during the backtracing procedure. The full set of solutions is the following:

$$A_1 = \begin{pmatrix} \mathtt{B} & \mathtt{C} & \mathtt{A} & \mathtt{C} & - & \mathtt{D} \\ \mathtt{D} & \mathtt{B} & \mathtt{A} & \mathtt{D} & \mathtt{A} & \mathtt{D} \end{pmatrix}$$

$$A_2 = \begin{pmatrix} - & \mathtt{B} & \mathtt{C} & \mathtt{A} & \mathtt{C} & - & \mathtt{D} \\ \mathtt{D} & \mathtt{B} & - & \mathtt{A} & \mathtt{D} & \mathtt{A} & \mathtt{D} \end{pmatrix}$$

$$A_3 = \begin{pmatrix} \mathtt{B} & \mathtt{C} & \mathtt{A} & - & \mathtt{C} & \mathtt{D} \\ \mathtt{D} & \mathtt{B} & \mathtt{A} & \mathtt{D} & \mathtt{A} & \mathtt{D} \end{pmatrix}$$

$$A_4 = \begin{pmatrix} - & \mathtt{B} & \mathtt{C} & \mathtt{A} & - & \mathtt{C} & \mathtt{D} \\ \mathtt{D} & \mathtt{B} & - & \mathtt{A} & \mathtt{D} & \mathtt{A} & \mathtt{D} \end{pmatrix}$$

$$A_5 = \begin{pmatrix} - & \mathtt{B} & \mathtt{C} & \mathtt{A} & \mathtt{C} & \mathtt{D} \\ \mathtt{D} & \mathtt{B} & \mathtt{A} & \mathtt{D} & \mathtt{A} & \mathtt{D} \end{pmatrix}$$

$$A_6 = \begin{pmatrix} - & \mathtt{B} & - & \mathtt{C} & \mathtt{A} & \mathtt{C} & \mathtt{D} \\ \mathtt{D} & \mathtt{B} & \mathtt{A} & \mathtt{D} & \mathtt{A} & - & \mathtt{D} \end{pmatrix}$$

$$A_7 = \begin{pmatrix} - & \mathtt{B} & \mathtt{C} & - & \mathtt{A} & \mathtt{C} & \mathtt{D} \\ \mathtt{D} & \mathtt{B} & \mathtt{A} & \mathtt{D} & \mathtt{A} & - & \mathtt{D} \end{pmatrix}$$

Not surprisingly, all of these alignments have cost $cost(A_1) = cost(A_2) = cost(A_3) = cost(A_4) = cost(A_5) = cost(A_6) = cost(A_7) = d(x, y) = 4$. ◀

4.3 The Alignment Score

So far, we only considered cost models, e.g. the unit cost, to measure the distance between two sequences. As long as we are interested in a **global** comparison, that makes sense. But when we are interested in a **local** comparison to learn about the least different parts of two sequences, we get a problem. A distance can only punish differences, not reward similarities, since it can never drop below zero. Note that the empty sequence ε is a (trivial) substring of every sequence and $d(\varepsilon, \varepsilon) = 0$; so this (probably completely uninteresting) common part is always among the best possible ones. Therefore the problem of finding the least different parts of two sequences is more easily formulated in terms of *similarity* than in terms of *dissimilarity* or *distance*.

That means that we assign a positive **score** to each match of two identical characters (corresponding to a copy operation) and a negative score to each mismatch (substitution operation), and also to each indel column (insertion or deletion). The indel score often has the same value for all characters $c \in \Sigma$. Its absolute value is also called **indel cost**.

Definition 16 The **score of an alignment** $A = (A_1, A_2, \ldots, A_n)$ is defined as the sum of its columns' scores, i.e., $\text{score}(A) = \sum_{i=1}^{n} \text{score}(A_i)$.

Definition 17 The **alignment similarity** of two sequences $x, y \in \Sigma^*$ is defined as

$$s(x, y) := \max\{\text{score}(A) \mid A \in \mathcal{A}^* \text{ is an alignment of } x \text{ and } y\}.$$

The **score-maximizing alignments** are given by the set

$$A^{\text{opt}}(x, y) := \{A \in \mathcal{A}^* \text{ is an alignment of } x \text{ and } y \text{ and } \text{score}(A) = s(x, y)\}.$$

When the score function is clear from the context, a score-maximizing alignment is often also called an **optimal alignment**.

5 Variations of Pairwise Sequence Alignment

In the previous chapter we have seen that edit sequences and sequence alignments are equivalent concepts to quantify and illustrate the differences (or similarities) of two given sequences. In this chapter we will use only the *alignment* model because this is the concept that is primarily used in bioinformatics. We will use the *similarity* scheme of maximizing scores, although in most cases an equivalent method minimizing costs could be used as well.

Let us look back at the alignment matrix of two input sequences x and y introduced in Section 4.2. Any path from the top-left corner to the bottom-right corner corresponds to an alignment of x and y, as can be easily seen from the way we reconstruct (from right to left) an optimal alignment using the backtracing matrix. By applying the dynamic programming scheme, i.e. filling the matrix in some way respecting the dependencies given by the alignment problem, optimal alignment scores are computed for each pair of prefixes from x and prefixes from y, and the final alignment is then found by backtracing.

In this chapter we show that by little modification of the "dependencies" mentioned in the previous sentence, slightly modified alignment problems can be solved in essentially the same way. To see this, consider the following general method:

Algorithm 18 (Universal Alignment Algorithm) In a $(|x|+1)\times(|y|+1)$ alignment matrix S define dependencies between the cells by an acyclic **predecessor** function $\mathrm{pred}(v)$ for any cell $v = (i, j)$, $0 \leq i \leq |x|$ and $0 \leq j \leq |y|$, in some way such that the top-left corner $v_S = (0, 0)$ has no predecessor and the bottom-right corner $v_E = (|x|, |y|)$ is not the predecessor of any cell. The score of the top-left corner is $S(v_S) = 0$. Then the score $S(v)$ of any other cell v, whose predecessors are already known, is computed as

$$S(v) = \max_{u \in \mathrm{pred}(v)} \{S(u) + \mathrm{score}(u \to v)\} \tag{5.1}$$

where $\mathrm{score}(u \to v)$ is the score of the alignment column corresponding to the step in the edit matrix from cell u to cell v.

Since pred is acyclic, S is well-defined, and we can arrange the computation in such an order that by the time we arrive at any cell v, we have already computed $S(u)$ for all predecessors needed for computing the maximum for $S(v)$. A possible order is to proceed row-wise through the rectangular matrix.

When computing $S(v)$, we take note which of the predecessors are maximizing. After arriving at v_E, we trace back a maximizing path to v_S in $O(|A|)$ time to reconstruct an alignment A. If we want to enumerate all optimal alignments we can use backtracking as described above.

5.1 Global Alignment

Global alignment is what we have discussed so far: Both sequences must be aligned from start to end. Often, the gap score is the same for all characters, say $-\gamma$ (with **gap cost** $\gamma > 0$).

It is a useful exercise to instantiate the Universal Alignment Algorithm shown in Equation (5.1) to this special case: As always, cell $v_S = (0, 0)$ has no predecessor and therefore score $S(0, 0) = 0$. Cells $(i, 0)$ and $(0, j)$ for $i, j \geq 1$ have one predecessor from $(i - 1, 0)$ and $(0, j - 1)$, respectively. Internal cells (i, j) for $i \geq 1$ and $j \geq 1$ have three predecessors. Therefore we obtain the following recurrence, known as the **global alignment algorithm** or **Needleman-Wunsch algorithm** (Needleman and Wunsch, 1970).

$$
\begin{array}{lcll}
S(0,0) & = & 0 & \\
S(i,0) & = & S(i-1,0) - \gamma & \text{for } 1 \leq i \leq |x| \\
S(0,j) & = & S(0,j-1) - \gamma & \text{for } 1 \leq j \leq |y| \\
S(i,j) & = & \max \left\{ \begin{array}{l} S(i-1,j-1) + \text{score}(x[i], y[j]) \\ S(i-1,j) - \gamma \\ S(i,j-1) - \gamma \end{array} \right\} & \text{for } 1 \leq i \leq |x|,\ 1 \leq j \leq |y|
\end{array}
$$

The global alignment similarity of x and y is then found in cell $(|x|, |y|)$:

$$s_{\text{global}}(x, y) = S(|x|, |y|).$$

Note that this algorithm is essentially the same algorithm as the one in Theorem 12, just phrased for scores instead of unit cost distances. It is easy to see that the time complexity is $O(|x| \cdot |y|)$.

5.2 Semi-global alignment

If one sequence, say x, is short and we are interested in the best match of x within y, we can modify global alignment in such a way that the whole of x is required to be aligned to any part of y. This **semi-global alignment** is global in the short sequence and local in the long sequence and also referred to as **approximate string matching**. In order to allow a prefix gap in the upper row of the alignment free of charge, we consider that the initial cell $v_S = (0, 0)$ is a predecessor of any cell $v = (0, j)$ in the first row of the alignment matrix, $1 \leq j \leq |y|$, with score zero, i.e. $\text{score}(v_S \to v) = 0$. In addition, to allow a suffix gap in the upper row of the alignment free of charge, we consider that any cell $v = (|x|, j)$ in the last row of the alignment matrix, $0 \leq j < |y|$, is a predecessor of cell $v_E = (|x|, |y|)$ with score zero, i.e. $\text{score}(v \to v_E) = 0$.

Applying the Universal Alignment Algorithm to the alignment matrix modified as just

described, we get the following recurrence.

$$\begin{array}{lcll} S(0,0) & = & 0 & \\ S(i,0) & = & S(i-1,0) - \gamma & \text{for } 1 \le i \le |x| \\ S(0,j) & = & S(0,0) & \text{for } 1 \le j \le |y| \\ S(i,j) & = & \max \left\{ \begin{array}{l} S(i-1,j-1) + \text{score}(x[i], y[j]) \\ S(i-1,j) - \gamma \\ S(i,j-1) - \gamma \end{array} \right\} & \text{for } 1 \le i \le |x|,\ 1 \le j \le |y| \end{array}$$

The semi-global alignment similarity is then the maximum entry in the last row:

$$s_{\text{semi-global}}(x,y) = \max\{S(|x|,j) \mid 0 \le j \le |y|\}.$$

Again, the time complexity is clearly $O(|x| \cdot |y|)$. A typical application of semi-global alignment in bioinformatics is **read mapping** where the location of a (relatively short) sequencing read is searched in a (large) genomic sequence in order to identify from which region in the genome (e.g., from which gene) the read might originate.

5.3 Free end gap alignment

If the two sequences are more or less of the same length and we expect that they have a long overlap (but may otherwise be rather unrelated), we have a similar situation as for semi-global alignment in the sense that gaps at the beginning or at the end of the alignment should be free of charge. Here, however, the gaps are usually not in the same row of the alignment, but may be one in the first and the other one in the second row, or *vice versa*. Therefore we extend the idea of semi-global alignment to both sequences. We consider the initial cell $v_S = (0,0)$ is a predecessor of any cell $v = (0,j)$ in the first row of the alignment matrix, $1 \le j \le |y|$, and of any cell $v = (i,0)$ in the first column of the alignment matrix, $1 \le i \le |x|$, all with score zero, i.e. $\text{score}(v_S \to v) = 0$. In addition, to allow suffix gaps free of charge, we consider that any cell $v = (|x|,j)$ in the last row of the alignment matrix, $0 \le j < |y|$, and any cell $v = (i,|y|)$ in the last column of the alignment matrix, $0 \le i < |x|$, is a predecessor of cell $v_E = (|x|,|y|)$ with score zero, i.e. $\text{score}(v \to v_E) = 0$.

Applying the Universal Alignment Algorithm to free end gap alignment, we get the following recurrence.

$$\begin{array}{lcll} S(0,0) & = & 0 & \\ S(i,0) & = & S(0,0) & \text{for } 1 \le i \le |x| \\ S(0,j) & = & S(0,0) & \text{for } 1 \le j \le |y| \\ S(i,j) & = & \max \left\{ \begin{array}{l} S(i-1,j-1) + \text{score}(x[i], y[j]) \\ S(i-1,j) - \gamma \\ S(i,j-1) - \gamma \end{array} \right\} & \text{for } 1 \le i \le |x|,\ 1 \le j \le |y| \end{array}$$

The free end gap alignment similarity is then the maximum entry in the last row or the last column:

$$s_{\text{free-end-gap}}(x,y) = \max\big(\max\{S(|x|,j) \mid 0 \le j \le |y|\}, \max\{S(i,|x|) \mid 0 \le i \le |x|\}\big).$$

As before, the time complexity is $O(|x| \cdot |y|)$. Free end gap alignment is very important for **genome assembly** (Chapter 9), i.e., when whole genomes are assembled from short DNA fragments: One has to determine how the fragments overlap, taking into account possible sequencing errors.

5.4 Local alignment

Often, two sequences x and y are not globally similar and also do not overlap at either end. Instead, they may share one highly similar region (e.g. a conserved protein domain) anywhere inside the sequences. In this case, it makes sense to look for the highest scoring pair of substrings of x and y. This is referred to as an optimal **local alignment** of x and y. Since a local alignment can start and end anywhere, the predecessor conditions in the alignment matrix must be relaxed even further: The initial cell $v_S = (0, 0)$ is a predecessor of any cell v in the whole matrix (except v_S itself) with score zero, and any cell v in the whole matrix (except v_E itself) is a predecessor of v_E with score zero.

Explicitly written, we obtain the local alignment algorithm, also known as the **Smith-Waterman algorithm** (Smith and Waterman, 1981). It is one of the most fundamental algorithms in computational biology.

$$
\begin{array}{lcll}
S(0,0) & = & 0 & \\
S(i,0) & = & S(0,0) & \text{for } 1 \le i \le |x| \\
S(0,j) & = & S(0,0) & \text{for } 1 \le j \le |y| \\
S(i,j) & = & \max \left\{ \begin{array}{l} S(0,0) \\ S(i-1,j-1) + \text{score}(x[i], y[j]) \\ S(i-1,j) - \gamma \\ S(i,j-1) - \gamma \end{array} \right\} & \text{for } 1 \le i \le |x|,\ 1 \le j \le |y|
\end{array}
$$

The local alignment similarity is then the maximum entry in the whole matrix:

$$s_{\text{local}}(x, y) = \max\{S(i, j) \mid 0 \le i \le |x|, 0 \le j \le |y|\}.$$

Although the search space has increased, analysis shows that this alignment variant takes only quadratic time $O(|x| \cdot |y|)$ as well, much better in fact than globally aligning each of the $O(|x|^2 \cdot |y|^2)$ pairs of substrings in $O(|x| \cdot |y|)$ time each, for a total time of $O(|x|^3 \cdot |y|^3)$. Because of its importance, this algorithm deserves a few remarks.

- Prior to the work of Smith and Waterman (1981), the notion of the "best" matching region of two sequences was not uniquely defined and often computed by heuristics whose output was taken as the definition of "best". Now we have a clear definition: Take any pair of substrings, compute an optimal global alignment for each pair, then take the pair with the highest score.
- Since the empty sequences are candidates for substrings and receive a global alignment score of zero, the local alignment score of any two sequences is always nonnegative.

- Two random sequences should have no similar region. Of course, even a single identical symbol will make a positive contribution to the score. Therefore small positive scores are meaningless. Also, the average score in a random model should be negative; otherwise we would obtain large positive scores with high probability for random sequences, which makes it hard to distinguish random similarities from true evolutionary relationships. We investigate these issues further in Section 6.3.
- Even though the $O(|x| \cdot |y|)$ running time appears reasonable at first sight, it can become a high bottleneck in large-scale sequence comparison. Therefore, often a **filter** or a **heuristic** is used. Several practical sequence comparison tools are presented in Chapter 6.
- The $O(|x| \cdot |y|)$ space requirement for the backtracing matrix E is an even more severe limitation than the time usage. In fact, linear-space methods exist (not explained here—they incur a small time penalty, approximately a factor of two) and are widely used in practice.
- The Smith-Waterman notion of local similarity has a serious flaw: It does not discard poorly conserved intermediate segments. The Smith-Waterman algorithm finds a local alignment with maximal score, but it is unable to find a local alignment with maximum degree of similarity (e.g. maximal percent of matches). As a result, local alignment sometimes produces a mosaic of well-conserved fragments, artificially connected by poorly-conserved or even unrelated fragments.

Once again we emphasize that local alignment makes sense only for measuring with a score function. If you use a cost function, the empty alignment would always be the best.

5.5 Gap Cost Variations for Alignments

General gap costs. So far, a gap (continuous stretch of indels in the same sequence) of length ℓ receives a score of $-\ell \cdot \gamma$, where $\gamma > 0$ is the gap cost (assuming that it is not character-specific or position-specific). Therefore it does not matter whether we interpret a run of ℓ consecutive indels as one long gap or as ℓ gaps of length 1. This model is called **linear gap costs**. (In general a function $g : \mathbb{R} \to \mathbb{R}$ is *linear* if $g(k + k') = g(k) + g(k')$ and $g(\lambda x) = \lambda g(x)$.)

When indels occur in biological sequences, they often concern more than a single character. Therefore we need more flexibility for specifying gap costs. The most general (not character- or position-specific) case allows a **general gap cost function** $g : \mathbb{N} \to \mathbb{R}_0^+$, where we pay $g(\ell)$ for a gap of length ℓ. We always set $g(0) = 0$ and frequently, in practice, we demand that gap costs are **subadditive**, i.e., $g(k + k') \leq g(k) + g(k')$ for all $k, k' \geq 0$. In this way, we penalize longer gaps relatively less than shorter gaps.

In protein coding regions, however, the following gap cost function may be reasonable; note that it is *not* subadditive:

$$g(\ell) := \begin{cases} \ell/3 & \text{if } \ell \bmod 3 = 0, \\ \ell + 2 & \text{otherwise.} \end{cases}$$

For the increased generality, we have to pay a price: a significant increase in the number of predecessors of a cell in the alignment matrix and therefore in the time complexity of the algorithm. The changes apply to all of the above variations (global, semi-global, free end gap, and local alignment). To the appropriate alignment matrix and predecessor function we add for each cell $v = (i, j)$, $0 \leq i \leq |x|$ and $0 \leq j \leq |y|$,

- the *vertical predecessors* $v' = (i', j)$ for all pairs $0 \leq i' < i \leq |x|$ with respective scores $\text{score}(v' \to v) = g(i - i')$ and
- the *horizontal predecessors* $v' = (i, j')$ for all pairs $0 \leq j' < j \leq |y|$ with respective scores $\text{score}(v' \to v) = g(j - j')$.

Since each cell has now $O(|x|+|y|)$ predecessors, the running time increases to $O(|x||y|(|x|+|y|))$ and is therefore cubic. In the following we show explicitly the global alignment algorithm with general gap costs $g : \mathbb{N} \to \mathbb{R}_0^+$.

$$
\begin{array}{lcll}
S(0,0) & = & 0 & \\
S(i,0) & = & \max_{0 \leq i' < i}\{S(i',0) - g(i-i')\} & \text{for } 1 \leq i \leq |x| \\
S(0,j) & = & \max_{0 \leq j' < j}\{S(0,j') - g(j-j')\} & \text{for } 1 \leq j \leq |y| \\
S(i,j) & = & \max \left\{ \begin{array}{l} S(i-1,j-1) + \text{score}(x[i], y[j]) \\ \max_{0 \leq i' < i}\{S(i',j) - g(i-i')\} \\ \max_{0 \leq j' < j}\{S(i,j') - g(j-j')\} \end{array} \right\} & \text{for } 1 \leq i \leq |x|,\ 1 \leq j \leq |y|
\end{array}
$$

The global alignment similarity with general gap costs g is then found as usual in cell $(|x|, |y|)$:

$$s^{(g)}_{\text{global}}(x, y) = S(|x|, |y|).$$

Although this starts to look intimidating, remember that this formula is still nothing else than Equation (5.1). General gap costs are useful, but the price in time complexity is usually too expensive to be paid. Fortunately, two special cases (but still more general than linear gap costs) allow more efficient algorithms: affine gap costs, to be discussed next, and concave gap costs, an advanced topic not considered in these notes.

Affine gap costs. Affine gap costs are important and widely used in practice. A gap cost function $g : \mathbb{N} \to \mathbb{R}_0^+$ is called an **affine gap cost function** if $g(l) = d + (l - 1) \cdot e$, where $d > 0$ is called the **gap open cost** and e, $0 < e \leq d$, is called the **gap extension cost**. The gap open cost is paid once for every consecutive run of gaps, namely at the first (opening) gap. Each additional gap character then costs only e. Of course, this case can be treated in the framework of general gap costs. We shall see, however, that a quadratic $O(|x||y|)$ time algorithm exists; the idea is due to Gotoh (1982). Again, we explain it for global alignment and the required modifications for the other alignment types are easy.

Recall that $S(i, j)$ is the alignment score for the two prefixes $x[1 \ldots i]$ and $y[1 \ldots j]$. In general, such a prefix alignment can end with a match/mismatch, a deletion, or an insertion. In the indel case, either the gap is of length $\ell = 1$, in which case its cost is $g(1) = d$, or its length is $\ell > 1$, in which case its cost can recursively be computed as $g(\ell) = g(\ell - 1) + e$.

The main idea is to additionally keep track of (i.e., to tabulate) the *state* of the last alignment column. In order to put this idea into an algorithm, we define the following additional two matrices:

$$V(i,j) = \max\left\{ score(A) \;\middle|\; \begin{array}{l} A \text{ is an alignment of the prefixes } x[1\ldots i] \text{ and } y[1\ldots j] \\ \text{that ends with a gap character in } y \end{array} \right\}$$

$$H(i,j) = \max\left\{ score(A) \;\middle|\; \begin{array}{l} A \text{ is an alignment of the prefixes } x[1\ldots i] \text{ and } y[1\ldots j] \\ \text{that ends with a gap character in } x \end{array} \right\}$$

Then

$$S(i,j) = \max\left\{S(i-1,j-1) + \text{score}(x[i],y[j]), V(i,j), H(i,j)\right\},$$

which gives us a method to compute the alignment matrix S, given the matrices V and H. It remains to explain how V and H can be computed efficiently. Consider the case of $V(i,j)$: A gap of length ℓ ending at position (i,j) is either a gap of length $\ell = 1$, in which case we can easily compute $V(i,j)$ as $V(i,j) = S(i-1,j) - d$. Or, it is a gap of length $\ell > 1$, in which case it is an extension of the best scoring vertical gap ending at position $(i-1,j)$, $V(i,j) = V(i-1,j) - e$. Together, we see that for $1 \le i \le |x|$ and $0 \le j \le |y|$,

$$V(i,j) = \max\left\{S(i-1,j) - d, V(i-1,j) - e\right\}.$$

Similarly, for horizontal gaps we obtain for $0 \le i \le |x|$ and $1 \le j \le |y|$,

$$H(i,j) = \max\left\{S(i,j-1) - d, H(i,j-1) - e\right\}.$$

The border elements of V and H are initialized in such a way that they do not contribute to the maximum in the first row or column:

$$V(0,j) = -\infty \text{ for } 0 \le j \le |y|, H(i,0) = -\infty \text{ for } 0 \le i \le |x|.$$

S is initialized as follows:

$$S(0,0) = 0, S(i,0) = V(i,0) \text{ for } 1 \le i \le |x|, S(0,j) = H(0,j) \text{ for } 1 \le j \le |y|.$$

An analysis of this algorithm reveals that it uses $O(|x||y|)$ time and space, although hidden in the O-notation is a larger constant factor compared to the case of linear gap costs. Note that, if just the optimal score is required, only two adjacent columns (or rows) from the matrices V, H, and S are needed. For backtracing, only the matrix E is needed; so V and H never need to be kept in memory entirely. Thus the memory requirement for pairwise alignment with affine gap costs is even in practice not much worse than for linear gap costs.

Example 19 Given amino acid strings `WW` and `WNDW`, compute the optimal global alignment score under affine gap costs with the following scoring scheme: Gap open cost of $d = 11$, gap extension cost of $e = 1$ and the substitution weights taken from the BLOSUM62 matrix (Henikoff and Henikoff, 1992), score(`W`, `W`) $= 11$, score(`W`, `N`) $= -4$, score(`W`, `D`) $= -4$.

Figure 5.1 shows the three matrices V, H and S used by the Gotoh algorithm to calculate a global alignment with affine gap costs. ◀

V	ϵ	W	N	D	W
ϵ	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
W	-11	-22	-23	-24	-25
W	-12	0	-11	-12	-13

H	ϵ	W	N	D	W
ϵ	$-\infty$	-11	-12	-13	-14
W	$-\infty$	-22	**0**	**-1**	-2
W	$-\infty$	-23	-11	-4	-5

S	ϵ	W	N	D	W
ϵ	**0**	-11	-12	-13	-14
W	-11	**11**	**0**	**-1**	-2
W	-12	0	7	-4	**10**

Figure 5.1: The three matrices V, H and S that are used to calculate a global alignment with affine gap costs with the Gotoh algorithm. The score-maximizing path is marked in bold face. If a value of the maximizing path of S originated from V or H, these cells are marked in bold face as well.

6 Pairwise Alignment in Practice

6.1 Alignment Visualization with Dot Plots

We begin with a visual method for (small scale) sequence comparison that is very popular with biologists. The most simple way of comparing two sequences $x \in \Sigma^m$ and $y \in \Sigma^n$ is the **dot plot**: The two sequences are drawn along the horizontal respectively vertical axis of a coordinate system, and positions (i, j) with identical characters $x[i] = y[j]$ are marked by a dot. Its time and memory requirements are $O(mn)$.

By visual inspection of such a dot plot, one can already observe a number of details about similar and dissimilar regions of x and y. For example, a diagonal stretch of dots refers to a common substring of the two strings, like SCENCE in Figure 6.1, upper part.

A disadvantage of dot plots is that they do not give a quantitative measure how similar the two sequences are. This is, of course, what the concept of sequence alignment provides. Dot plots are still important in practice since the human eye is not easily replaced by more abstract numeric quantities.

Another disadvantage of the basic dot plot, especially for DNA with its small alphabet size, is the cluttering because of scattered short "random" matches. Note that, even on random strings, the probability that a dot appears at any position, is $1/|\Sigma| = 1/4$ for DNA. So a quarter of all positions in a dot plot are black, which makes it hard to see the interesting similarities. Therefore the dotplot is usually filtered, e.g. by removing all dots that are not part of a consecutive match of length $\geq q$, where q is a user-adjustable parameter (see Figure 6.1, lower part).

6.2 Fundamentals of Rapid Database Search Methods

In practice, pairwise alignment algorithms are used for two related, but still conceptually different purposes, and it is important to keep the different goals in mind.

1. True pairwise alignment: Given **two sequences** $x, y \in \Sigma^*$ that we already know or suspect to be similar, report all similar regions and show the corresponding (even suboptimal) alignments.

2. Large-scale database search: Given a **query** $x \in \Sigma^*$ and a family (database) Y of **candidates**, find out (quickly) which $y \in Y$ share at least one sufficiently similar region with x and report those y along with the similarity score (e.g. the alignment score). The alignment itself is of little interest in this case; suboptimal alignments are of even less interest.

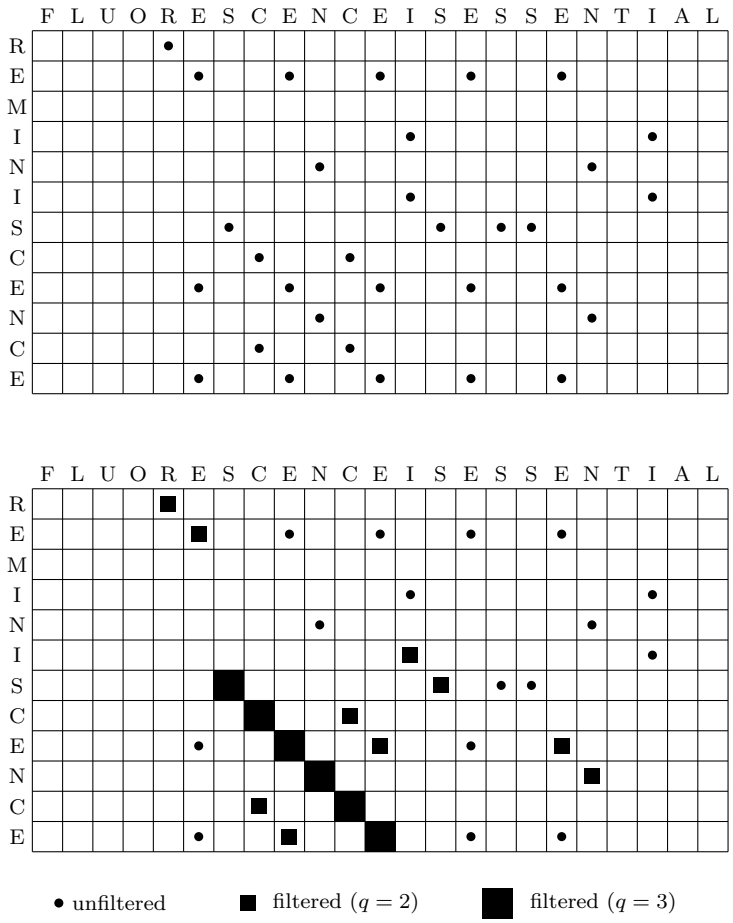


Figure 6.1: Upper part: Unfiltered dot plot. Lower part: Filtered dot plot with different filters applied. Here the filter keeps only those positions (i, j), that are part of a common substring of length $\geq q$.

The rest of this chapter is devoted to methods aiming at the second goal. Most database search methods in bioinformatics consider *local* and not *global* alignment. The simple reason is that entries in biological sequence databases often contain only parts of a gene or protein sequence and therefore global comparison may not be successful, while local overlaps will still be detected. Conversely, if two globally matching sequences are aligned with a local method, this will still work well and produce a nice global or near-global alignment.

An important idea in order to allow for fast database search is **filtering**. Such methods are often used in practice for large-scale database search. In the first (*filtration*) phase, accuracy is sacrificed for speed, in order to quickly identify several (but not all) sequences that are dissimilar from the query and do not need to be considered further. Then, in the second (*verification*) phase the quadratic-time algorithms of the previous chapter are considered to produce the exact result.

The following very simple observation, the so-called ***q*-gram lemma**, can be used to identify areas in two sequences where a strong local similarity *may* reside, and therefore build a good basis for methods implementing the filtration phase, if one sequence is the query and the other is a candidate database sequence:

Lemma 20 Given a local alignment of length ℓ with at most e errors (mismatches or indels), the aligned regions of the two strings contain at least $T(\ell, q, e) := \ell + 1 - q \cdot (e+1)$ common q-grams.

Proof. The number of common q-grams between two identical strings of length ℓ is $\ell - q + 1$. Each single-letter difference between the strings affects at most q of these q-grams and therefore reduces their number by at most q. Therefore the number of q-grams that are unaffected by e errors is at least $\ell - q + 1 - e \cdot q = \ell + 1 - q \cdot (e+1)$. □

Based on this lemma simple methods can be devised for finding database entries with strong local similarities to a query sequence (**seeds**) that in later steps then may be extended into longer matches. The methods have in common that the q-grams in the database need to be organized so that they can be accessed quickly. The ***q*-gram index** is a suitable datastructure for this task:

Definition 21 A ***q*-gram index** for $y \in \Sigma^n$ is a map $I : \Sigma^q \to \mathcal{P}(\{1, \ldots, n-q+1\})$ such that $I(z) = \{i_1(z), i_2(z), \ldots\}$, where $i_1(z) < i_2(z) < \ldots$ are the starting positions of the q-gram z in y. Clearly, $|I(z)|$ is the occurrence count of z in y, i.e. the number of occurrences of z in y.

A straightforward (but inefficient!) $O(|\Sigma|^q + qn)$ time method to create a q-gram index would be to use an (initially empty) dictionary with q-grams as keys and position arrays as values. One would slide a window of length q across y, and each position is added to the list of the appropriate q-gram.

A more low-level and efficient $O(|\Sigma|^q + n)$ time method is to use two integer arrays `first`$[0..|\Sigma|^q]$ and `pos`$[0..n-q+1]$ and a q-gram ranking function that assigns to each q-gram an integer between 0 and $|\Sigma|^q - 1$, illustrated in Figure 6.2. All starting positions of q-gram z with rank r are consecutively stored in `pos`, starting at position `first`$[r]$ and

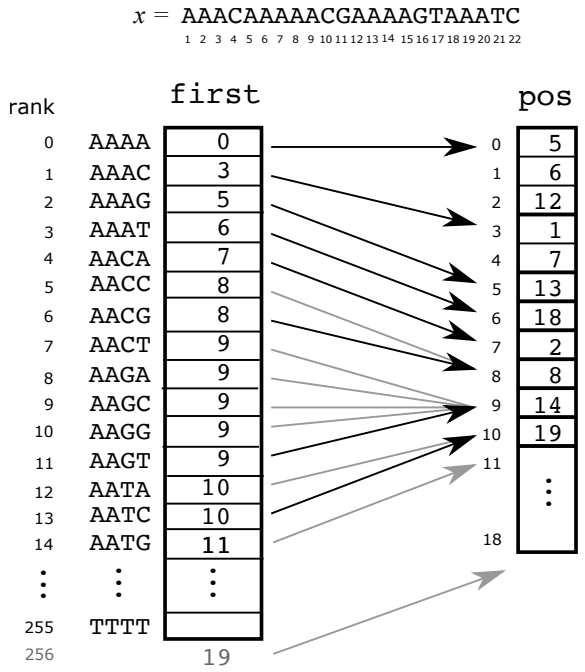


Figure 6.2: Illustration of an efficient q-gram index data structure for $q = 4$ and the input string $y =$ `AAACAAAAACGAAAAGTAAATC`. A q-gram receives the rank corresponding to its representation as a 4-ary number with rank(`A`) = 0, rank(`C`) = 1, rank(`G`) = 2 and rank(`T`) = 3.

ending at position `first`$[r+1]-1$ (assuming that `first`$[|\Sigma|^q] = n-q+2$). These can be constructed with a simple two-pass algorithm that scans y from left to right twice. In the first pass it counts the number of occurrences of each q-gram and creates `first`. In the second pass it inserts the q-gram starting positions at the appropriate spots in `pos`. Since the ranking function update takes constant time instead of $O(q)$ time, this version is more efficient (and also avoids object-oriented overhead).

6.3 Alignment Statistics

6.3.1 Preliminaries

In this section, we present some basic probabilistic calculations that are useful for choosing appropriate parameters of database search algorithms and to evaluate the quality of local alignments.

Often it is a problem to rank two alignments, especially if they have been created from different sequences under different scoring schemes (different score matrix and gap cost function): The absolute score value cannot be compared because the scores in one matrix might be scaled with a factor of 10, while the scores in the other matrix might be scaled with a factor of 100, so scores of 67 and 670 would *not* indicate that the second alignment is any better.

Statistical significance computations provide a universal way to evaluate and rank alignments (among other things). The central question in this context is: *How probable is it to observe the present event (or a more extremal one) in a null model?*

Definition 22 A **null model** in general is a random model for objects of interest that does not contain signal features. It specifies a probability distribution on the set of objects under consideration.

More specifically, a **null model for pairwise sequence alignment** (for given sequence lengths m and n) specifies a probability for each sequence pair $(x, y) \in \Sigma^m \times \Sigma^n$.

Definition 23 The most commonly used null model for pairwise sequence alignment is the **i.i.d. model**[1], where each sequence is created by randomly and independently drawing each character from a given alphabet with given character frequencies $f = (f_c)_{c\in\Sigma}$. The probability that a random sequence X of length m turns out to be exactly $x \in \Sigma^m$ is the product of its character frequencies: $\mathbb{P}(X = x) = \prod_{i=1}^{m} f_{x[i]}$. Similarly, $\mathbb{P}(Y = y) = \prod_{j=1}^{n} f_{y[j]}$. The probability that the pair (X, Y) is exactly (x, y) is the product of the individual probabilities: $\mathbb{P}((X, Y) = (x, y)) = \mathbb{P}(X = x) \cdot \mathbb{P}(Y = y)$.

When we observe an alignment score s for two given sequences, we can ask the following two questions: For random sequences from the null model of the same length as the given ones, what is the probability that two of these sequences have an alignment score of at least s and what is the expected number of pairwise alignments with an alignment score of at least s? The probability is called the ***p*-value** and the expected number is called the ***e*-value** associated to the event of observing score s.

Definition 24 The ***p*-value** of an event (with respect to a null model) is the probability to observe the event or a more extremal one in the null model.

Definition 25 The ***e*-value** of an event (with respect to a null model) is the expected number of events equal to or more extremal than the observed one in the null model.

Note that the null model ensures that the sequences have essentially no built-in similarity, since they are chosen independently. Any similarity measured by the alignment score is therefore due to chance. In other words, if a score $\geq s$ is quite probable in the null model, a score value of s is not an indicator of biological similarity or homology. The *smaller* the p-value and the e-value, the less likely it is that the observed similarity is simply due to chance and the *more significant* is the score. Good p-values are for example 10^{-10} or 10^{-20}.

A p-value can be converted into a universally interpretable score (a measure of surprise of the observed event), e.g. by $B := -\log_2(p)$, called the **bit score**. A bit score of $B \geq b$ always has probability 2^{-b} in the null model.

It is often a difficult problem to compute the p-value associated to an event. This is especially true for local sequence alignment. The remainder of this section provides an intuitive, but mathematically inexact approach.

[1] "i.i.d." is abbreviated for **i**ndependent and **i**dentically **d**istributed.

6.3.2 Statistics of q-gram Matches and FASTA Scores

Let us begin by computing the exact p-value of a q-gram match at position (i, j) in the alignment matrix. In the following, X and Y denote random sequences of length m and n, respectively, from the null model.

Look at two arbitrary characters $X[i]$ and $Y[j]$. What is the probability p that they are equal? The probability that both are equal to $c \in \Sigma$ is $f_c \cdot f_c = f_c^2$. Since c can be any character, we have

$$p = \mathbb{P}(X[i] = Y[j]) = \sum_{c \in \Sigma} f_c^2.$$

If the characters are **uniformly distributed**, i.e., if $f_c = 1/|\Sigma|$ for all $c \in \Sigma$, then $p = 1/|\Sigma|$.

Now let us look at two q-grams $X[i \dots i+q-1]$ and $Y[j \dots j+q-1]$. They are equal if and only if all q characters are equal; since these are independent in the i.i.d. model, the probability of an exact q-gram match (Hamming distance zero) is

$$\begin{aligned} p_0(q) &= \mathbb{P}(d_H(X[i \dots i+q-1], Y[j \dots j+q-1]) = 0) \\ &= \mathbb{P}(X[i \dots i+q-1] = Y[j \dots j+q-1]) = p^q. \end{aligned}$$

We can also compute the probability that the q-gram contains exactly one mismatch (probability $1-p$) and $q-1$ matches (probability p each): There are q positions where the mismatch can occur; therefore the total probability for Hamming distance 1 is

$$\mathbb{P}(d_H(X[i \dots i+q-1], Y[j \dots j+q-1]) = 1) = q \cdot (1-p) \cdot p^{q-1}.$$

Taken together, the probability of a q-gram match with *at most* one mismatch is

$$p_1(q) := \mathbb{P}(d_H(X[i \dots i+q-1], Y[j \dots j+q-1]) \leq 1) = [p + q(1-p)] \cdot p^{q-1}.$$

Similarly, we can compute the p-value $p_k(q)$ of a q-gram match with at most k mismatches.

Online database search. So far, we have considered fixed but arbitrary coordinates (i, j). If we run a q-gram based filter in a large-scale database search and consider each q-gram a hit that must be extended, we are interested in the e-value, the expected number $\mu(q)$ of exact hits, and the probability $p^*(q)$ of at least one hit. This is a so called **multiple testing problem**: At each position, there is a small chance of a random q-gram hit. When there are many positions, the probability of seeing at least one hit somewhere can become large, even though the individual probability is small.

Since there are $(m-q+1) \cdot (n-q+1)$ positions where a q-gram can start and each has the same probability, we have

$$\mu(q) = (m-q+1) \cdot (n-q+1) \cdot p^q.$$

Computing $p^*(q)$ is much more difficult. If all positions were independent and p^q was small, we could argue that the number of hits $N(q)$ has a Poisson distribution with expectation

$\mu(q)$. Since the probability of having zero hits is $\mathbb{P}(N(q) = 0) \approx \frac{e^{-\mu(q)} \cdot (\mu(q))^0}{0!} = e^{-\mu(q)}$, see Section 2.6, the probability of having at least one hit equals the probability of *not* having zero hits: $p^*(q) = \mathbb{P}(N(q) \geq 1) = 1 - \mathbb{P}(N(q) = 0)$ which results to

$$p^*(q) = 1 - \mathbb{P}(N(q) = 0) = 1 - e^{-\mu(q)}.$$

However, many potential q-grams in the alignment matrix overlap and therefore cannot be independent.

The longest match. We can ask for the p-value of the longest exact match between x and y being at least ℓ characters long, $\mathbb{P}(L(x, y) \geq \ell)$. This probability is equal to the probability of at least one match of length at least ℓ, which is $p^*(\ell) = 1 - e^{-\mu(\ell)} = 1 - e^{-(m-\ell+1)\cdot(n-\ell+1)\cdot p^\ell} \approx 1 - e^{-mnp^\ell} \approx mnp^\ell$ if $mnp^\ell \ll 1$.

The take-home message is: For relatively large ℓ, the probability that the longest exact match has length ℓ increases linearly with the search space size mn and decreases exponentially with the match length ℓ.

6.3.3 Statistics of Local Alignments

In the previous section we have argued that, if we measure the alignment score between random sequences X and Y of lengths m and n simply by the length $L(X, Y)$ of the longest exact match, we get

$$\mathbb{P}(L(X, Y) \geq \ell) \approx 1 - e^{-mnp^\ell} = 1 - e^{-Cmne^{-\lambda\ell}}$$

for constants $C > 0$ and $\lambda > 0$ such that $p = e^{-\lambda}$.

There is much theoretical and practical evidence that the same formula is still true if the length of the longest exact match is replaced by a more complex scoring function for local alignment that also allows mismatches and gaps. Restrictions are that the gap cost must be reasonably high (so the alignment is not dominated by gaps) and the average score of aligning two random characters must be negative (so the local alignment does not become global just by chance). In this case it can be argued (note that we haven't proved anything here and that our derivations are far from mathematically exact!) that the local alignment score $S(X, Y)$ also satisfies

$$\mathbb{P}(S(X, Y) \geq t) \approx 1 - e^{-Cmne^{-\lambda t}} \left[\approx Cmne^{-\lambda t} \text{ if } Cmne^{-\lambda t} \ll 1\right]$$

with constants $C > 0$ and $\lambda > 0$ that depend on the scoring scheme. This distribution is referred to as an **extreme value distribution** or **Gumbel distribution**.

6.4 BLAST: A fast Database Search Method

BLAST[2] (Altschul et al., 1990) is perhaps the most popular program to perform biological sequence database searches. Here we describe the program for protein sequences (BLASTP). We mainly consider an older version that was used until about 1998 (BLAST 1.4). The newer version (BLAST 2.0 and later) is discussed at the end of this section.

BLAST 1.4. The main idea of protein BLAST is to first find high-scoring q-gram hits, so-called BLAST hits, and then extend them.

Definition 26 For a given $q \in \mathbb{N}$ and $k \geq 0$, a **BLAST hit** of $x \in \Sigma^m$ and $y \in \Sigma^n$ is a pair (i, j) such that $\text{score}(x[i \dots i+q-1], y[j \dots j+q-1]) \geq k$.

To find BLAST hits, we proceed as follows: We create a list $N_k(x)$ of all q-grams in Σ^q that score highly if aligned without gaps to any q-gram in the query x and note the corresponding positions in x.

Definition 27 The k-neighborhood $N_k(x)$ of $x \in \Sigma^m$ is defined as

$$N_k(x) := \{(z, i) \mid z \in \Sigma^q, 1 \leq i \leq m-q+1, \text{score}(z, x[i \dots i+q-1]) \geq k\}.$$

The k-neighborhood can be represented similar to a q-gram index: We extend table `pos` such that for each $z \in \Sigma^q$, we store consecutively in `pos` the set $P_k(z)$ of positions i such that $(z, i) \in N_k(x)$, see Figure 6.3.

The size of this index grows exponentially with q and $1/k$, so these parameters should be selected carefully. For the PAM250 score matrix, $q = 4$ and $k = 17$ have been used successfully.

Once the index for x has been created, for each database sequence $y \in Y$ the following steps are executed.

1. Find BLAST hits: Scan y from left to right with a q-window, updating the current q-gram rank in constant time. For each position j, the hits are $\{(i, j) : (z, i) \in N_k(y[j \dots j+q-1])\}$. This takes $O(|y| + h)$ time, where h is the number of hits.
2. Each hit (i, j) is extended without gaps along the diagonal to the upper left and separately to the lower right, hoping to collect additional matching characters that increase the score. Each extension continues until too many mismatches accumulate and the score drops below $M - X$, where M is the maximal score reached so far and X is a user-specified drop-off parameter. This is the reason why this extension method is called ***X*-drop algorithm**. The maximally scoring part of both extensions is retained. The pair of sequences around the hit delivered by this extension is called a **maximum segment pair (MSP)**. For any such MSP, a significance score is computed. If this is better than a pre-defined significance threshold, then the MSP is kept for the next step.

[2] Basic Local Alignment Search Tool

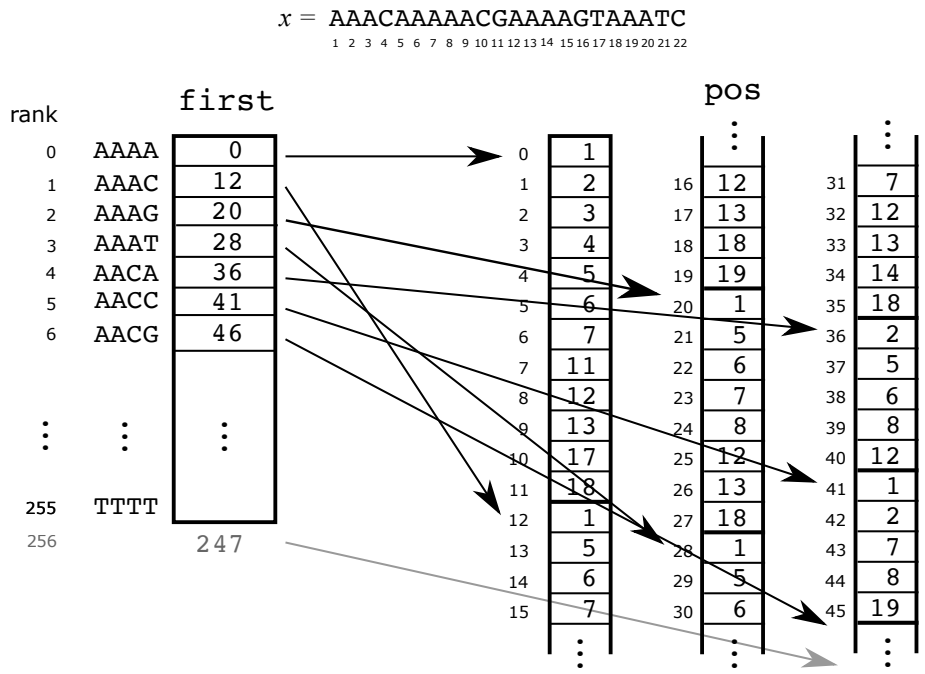


Figure 6.3: Illustration of the BLAST index storing the k-neighborhood of the input string $x =$ `AAACAAAAACGAAAAGTAAATC` for $q = 4$ and $k = 3$ (unit score).

3. MSPs on different diagonals are combined.
4. A combined significance threshold is computed.

BLAST 2.0. In later versions of BLAST (BLAST 2.0 or Gapped BLAST, see Altschul et al. (1997)), a different and much more time-efficient method is used. As before, hits are searched but with reduced values for k and q. This results in more hits. However, BLAST 2.0 looks for pairs of hits on the same diagonal within some maximal distance. The mid point between two hits on a diagonal is then used as the starting point for the X-drop algorithm described above. Usually only a few regions are extended, which makes the method much faster. Practice shows that the two-hit approach is similar w.r.t. sensitivity compared to the one-hit approach.

6.5 DIAMOND

A modern and much more efficient alternative software for aligning DNA or protein sequences against a protein database is DIAMOND (Buchfink et al., 2015). It combines the classical seed and extend paradigm with a clever double-indexing strategy for query and database. In addition, it is memory efficient, avoiding cache misses, speeding up the procedure even more. Therefore, in a setting where the same query is used in many searches, for example in a metagenomic study or an all-against-all protein sequence comparison, DIAMOND is several thousand times faster than BLAST.

7 Multiple Sequence Alignment

7.1 Basic Definitions

Multiple (sequence) alignment deals with the problem of aligning generally more than two sequences. While at first sight multiple alignment might look like a straightforward generalization of pairwise alignment, we will see that there are clear reasons why multiple alignments should be considered separately, reasons both from the application side, and from the computational side. In fact, the first applications of multiple alignments date back to the early days of phylogenetic studies on protein sequences in the 1960s. Back then, multiple alignments were possibly of higher interest than pairwise alignments. However, due to the computational hardness the progress on multiple alignment methods was slower than in the pairwise case, and the main results were not obtained before the 1980s and 1990s.

A multiple alignment is the simultaneous alignment of several sequences, usually displayed as a matrix with multiple rows, each row corresponding to one sequence.

The formal definition of a **multiple sequence alignment** is a straightforward generalization of that of a pairwise sequence alignment (see Section 4.1). We shall usually assume that there are $k \geq 2$ sequences $s_1, \ldots, s_k$ to be aligned.

Definition 28 Let Σ be the character alphabet, and $s_1, \ldots, s_k \in \Sigma^*$. Then $\mathcal{A}(k) := (\Sigma \cup \{-\})^k \setminus \{(-)^k\}$ is the **multiple alignment alphabet** for k sequences. In other words, the elements of this alphabet are columns of k symbols, at least one of which must be a character from Σ.

The **projection** of a multiple alignment A *to the i-th sequence* is a function $\pi_{\{i\}} : \mathcal{A}(k)^* \to \Sigma^*$ where

$$\pi_{\{i\}}(\binom{a_1}{\vdots}{a_k}) := \begin{cases} a_i & \text{if } a_i \neq - \\ \varepsilon & \text{if } a_i = - \end{cases}$$

is applied successively to all columns of the input alignment A. In other words, the projection reads the i-th row of A, omitting all gap characters.

A **global multiple alignment** of $s_1, \ldots, s_k$ is a $k \times l$-matrix A, where k and l denote the number of rows and columns of the alignment, respectively, with $\pi_{\{i\}}(A) = s_i$ for all $i = 1, \ldots, k$, such that A contains no column $\binom{-}{\vdots}{-}$.

We generalize the definition of the projection function to a set I of sequences (in particular to two sequences).

Definition 29 The **projection** of a multiple alignment A of k sequences *to an index set* $I = \{i_1, i_2, \ldots, i_q\}$, $q \leq k$ and $i_1 < i_2 < \ldots < i_q$, where each i_j corresponds to an index of one of the k sequences ($I \subseteq \{1, \ldots, k\}$), is defined as the function $\pi_I : \mathcal{A}(k)^* \to \mathcal{A}(q)^*$ that maps an alignment column $a = \binom{a_1}{\vdots}{a_k}$ as follows:

$$\pi_I(\tbinom{a_1}{\vdots}{a_k}) := \begin{cases} \varepsilon & \text{if } \binom{a_{i_1}}{\vdots}{a_{i_q}} = \binom{-}{\vdots}{-} \\ \binom{a_{i_1}}{\vdots}{a_{i_q}} & \text{otherwise} \end{cases}$$

In other words, the projection selects the rows with indices given by the index set I from the alignment and omits all columns that consist only of gaps. The definition is illustrated in the following example. The term *projection* is motivated in Figure 7.1.

Example 30 Let

$$A = \begin{pmatrix} \texttt{-} & \texttt{A} & \texttt{C} & \texttt{C} & \texttt{G} & \texttt{A} & \texttt{T} & \texttt{C} & \texttt{-} \\ \texttt{-} & \texttt{A} & \texttt{-} & \texttt{T} & \texttt{G} & \texttt{A} & \texttt{A} & \texttt{-} & \texttt{G} \\ \texttt{T} & \texttt{A} & \texttt{C} & \texttt{T} & \texttt{G} & \texttt{A} & \texttt{-} & \texttt{C} & \texttt{-} \\ \texttt{-} & \texttt{A} & \texttt{C} & \texttt{C} & \texttt{G} & \texttt{A} & \texttt{A} & \texttt{C} & \texttt{G} \end{pmatrix}.$$

Then

$$\begin{aligned} \pi_{\{1,2\}}(A) &= \begin{pmatrix} \texttt{A} & \texttt{C} & \texttt{C} & \texttt{G} & \texttt{A} & \texttt{T} & \texttt{C} & \texttt{-} \\ \texttt{A} & \texttt{-} & \texttt{T} & \texttt{G} & \texttt{A} & \texttt{A} & \texttt{-} & \texttt{G} \end{pmatrix} \\ \pi_{\{3,4\}}(A) &= \begin{pmatrix} \texttt{T} & \texttt{A} & \texttt{C} & \texttt{T} & \texttt{G} & \texttt{A} & \texttt{-} & \texttt{C} & \texttt{-} \\ \texttt{-} & \texttt{A} & \texttt{C} & \texttt{C} & \texttt{G} & \texttt{A} & \texttt{A} & \texttt{C} & \texttt{G} \end{pmatrix} \\ \pi_{\{1,2,4\}}(A) &= \begin{pmatrix} \texttt{A} & \texttt{C} & \texttt{C} & \texttt{G} & \texttt{A} & \texttt{T} & \texttt{C} & \texttt{-} \\ \texttt{A} & \texttt{-} & \texttt{T} & \texttt{G} & \texttt{A} & \texttt{A} & \texttt{-} & \texttt{G} \\ \texttt{A} & \texttt{C} & \texttt{C} & \texttt{G} & \texttt{A} & \texttt{A} & \texttt{C} & \texttt{G} \end{pmatrix} \end{aligned}$$

◀

7.2 Why multiple sequence comparison?

In the following we list a few justifications why multiple sequence alignment is more than just the repeated application of pairwise alignment.

- In a multiple sequence alignment, several sequences are compared at the same time. Properties that do not clearly become visible from the comparison of only two sequences will be highlighted if they appear in many sequences. For example, in the alignment shown in Figure 7.2 (top), if only s_1 and s_2 are compared, the two white areas might look most similar, inhibiting the alignment of the two gray regions. Only by studying all five sequences at the same time (Figure 7.2, bottom) it becomes clear that the gray region is a better characteristic feature of this sequence family than the white one.

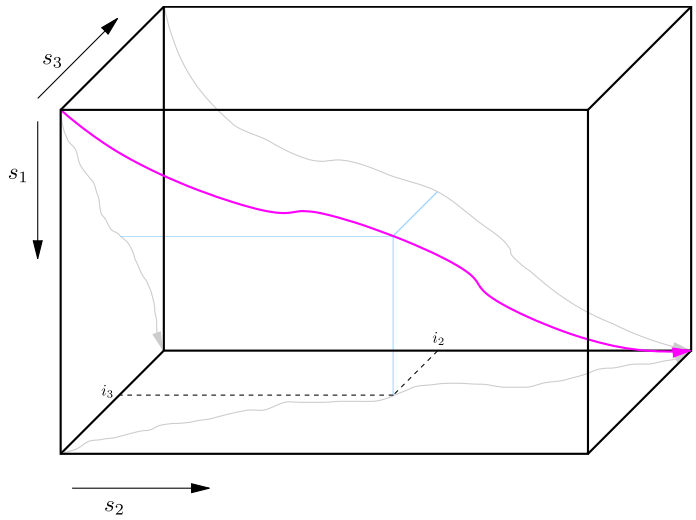


Figure 7.1: Projection of a multiple alignment of three sequences s_1, s_2, s_3 on the three planes (s_1, s_2), (s_2, s_3) and (s_3, s_1).

- Sometimes alignment ambiguities can be resolved due to the additional information given, as the following example shows: $s_1 =$ `VIEQLA` and $s_2 =$ `VINLA` may be aligned in the two different ways

$$A_1 = \begin{pmatrix} \texttt{V} & \texttt{I} & \texttt{E} & \texttt{Q} & \texttt{L} & \texttt{A} \\ \texttt{V} & \texttt{I} & \texttt{N} & \texttt{-} & \texttt{L} & \texttt{A} \end{pmatrix} \quad \text{and} \quad A_2 = \begin{pmatrix} \texttt{V} & \texttt{I} & \texttt{E} & \texttt{Q} & \texttt{L} & \texttt{A} \\ \texttt{V} & \texttt{I} & \texttt{-} & \texttt{N} & \texttt{L} & \texttt{A} \end{pmatrix}.$$

  Only the additional sequence $s_3 =$ `VINQLA` can show that the first alignment A_1 is probably the correct one.

- Dissimilar areas of related sequences become visible, showing regions where evolutionary changes happened.

The following two typical uses of multiple alignments can be identified:

1. Highlight *similarities* of the sequences in a family. Examples for applications of this kind of multiple alignments are:
   - sequence assembly,
   - molecular modeling, structure-function conclusions,
   - database search,
   - primer design.
2. Highlight *dissimilarities* between the sequences in a family. Here, the main application is the analysis of evolutionary relationships, like
   - reconstruction of phylogenetic trees,
   - analysis of single nucleotide polymorphisms (SNPs).

In conclusion: *One or two homologous sequences whisper ... a full multiple alignment shouts out loud.* (Hubbard et al., 1996)

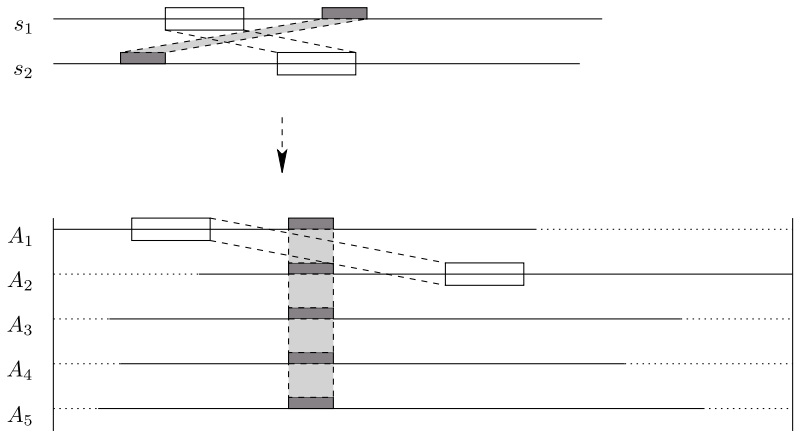


Figure 7.2: A small pattern present in many sequences may become better visible in a multiple sequence alignment.

7.3 Sum-of-Pairs Alignment

We have not discussed quantitative measures of multiple sequence alignment quality so far. In fact, several such measures exist. In this section we will define the sum-of-pairs cost as one example, and in Chapter 8 we will cover the tree cost as another one.

Like in pairwise alignment, the first choice one has to make is that between a distance (cost) and a similarity (score) function. Again, both are possible and being used, and one always has to make sure not to confuse the two. We will mostly use cost functions.

The multiple alignment cost functions discussed here are based on pairwise alignment cost functions, here denoted by $\text{cost}_2(\cdot)$ to highlight the fact that they apply to two sequences. (They may be weighted or unweighted, with homogeneous, affine, or general gap costs.)

Definition 31 The **sum-of-pairs cost** is just the sum of the costs of all pairwise projections:

$$\text{cost}_{\text{SP}}(A) = \sum_{1 \le p < q \le k} \text{cost}_2(\pi_{\{p,q\}}(A)).$$

(We similarly define the **sum-of-pairs score** $\text{score}_{\text{SP}}(A)$ of a multiple alignment A, and all definitions here and in the next section can be adapted in the same spirit.)

Example 32 Let $s_1 = \texttt{CGCTT}$, $s_2 = \texttt{ACGGT}$, $s_3 = \texttt{GCTGT}$ and

$$A = \begin{pmatrix} \texttt{C} & \texttt{G} & \texttt{C} & \texttt{T} & \texttt{-} & \texttt{T} \\ \texttt{-} & \texttt{A} & \texttt{C} & \texttt{G} & \texttt{G} & \texttt{T} \\ \texttt{-} & \texttt{G} & \texttt{C} & \texttt{T} & \texttt{G} & \texttt{T} \end{pmatrix}.$$

Assuming that $\text{cost}_2(\cdot)$ is the unit cost function, we get $\text{cost}_{\text{SP}}(A) = 4 + 2 + 2 = 8$. ◀

7.4 Multiple Sequence Alignment Problem

The multiple sequence alignment problem is formulated as a direct generalization of the pairwise case.

Problem 33 (Multiple Sequence Alignment Problem) Given k sequences $s_1, s_2, \ldots, s_k$ and a multiple alignment cost function cost, find an alignment A^{opt} of $s_1, s_2, \ldots, s_k$ such that $\text{cost}(A^{opt})$ is minimum among all possible alignments of $s_1, s_2, \ldots, s_k$.

Such an alignment A^{opt} is called an **optimal multiple alignment** of $s_1, s_2, \ldots, s_k$, and the value $d_{\text{SP}}(s_1, s_2, \ldots, s_k) := \text{cost}(A^{opt})$ is the **optimal multiple alignment cost** of $s_1, s_2, \ldots, s_k$.

Hardness. Sum-of-pairs alignment is an NP-hard[1] optimization problem with respect to the number of sequences k (Wang and Jiang, 1994). While we will not prove this here, we give an intuitive argument why the problem is difficult.

Let us have a look at the following example. It shows that sum-of-pairs-optimal multiple alignments can not be constructed "greedily" by combination of several optimal pairwise alignments:

Example 34 Let $s_1 = \texttt{CGCG}$, $s_2 = \texttt{ACGC}$ and $s_3 = \texttt{GCGA}$. In a unit cost scenario, the (only) optimal alignment of s_1 and s_2 is

$$A^{(1,2)} = \begin{pmatrix} \texttt{-} & \texttt{C} & \texttt{G} & \texttt{C} & \texttt{G} \\ \texttt{A} & \texttt{C} & \texttt{G} & \texttt{C} & \texttt{-} \end{pmatrix}$$

with $\text{cost}_2(A^{(1,2)}) = 2$, and the (only) optimal alignment of s_1 and s_3 is

$$A^{(1,3)} = \begin{pmatrix} \texttt{C} & \texttt{G} & \texttt{C} & \texttt{G} & \texttt{-} \\ \texttt{-} & \texttt{G} & \texttt{C} & \texttt{G} & \texttt{A} \end{pmatrix}$$

with $\text{cost}_2(A^{(1,3)}) = 2$.

Combining the two alignments into one multiple alignment, using the common sequence s_1 as seed, yields the multiple alignment

$$A^{((1,2),(1,3))} = \begin{pmatrix} \texttt{-} & \texttt{C} & \texttt{G} & \texttt{C} & \texttt{G} & \texttt{-} \\ \texttt{A} & \texttt{C} & \texttt{G} & \texttt{C} & \texttt{-} & \texttt{-} \\ \texttt{-} & \texttt{-} & \texttt{G} & \texttt{C} & \texttt{G} & \texttt{A} \end{pmatrix}$$

with $\text{cost}_{\text{SP}}(A^{((1,2),(1,3))}) = 2 + 2 + 4 = 8$. However, this is not a sum-of-pairs optimal alignment, which is

$$A^{opt} = \begin{pmatrix} \texttt{-} & \texttt{C} & \texttt{G} & \texttt{C} & \texttt{G} \\ \texttt{A} & \texttt{C} & \texttt{G} & \texttt{C} & \texttt{-} \\ \texttt{G} & \texttt{C} & \texttt{G} & \texttt{A} & \texttt{-} \end{pmatrix}$$

with $\text{cost}_{\text{SP}}(A^{opt}) = 2 + 3 + 2 = 7$. ◀

[1] If you don't know what exactly this means, just assume that any method solving this problem to optimality takes really, really long.

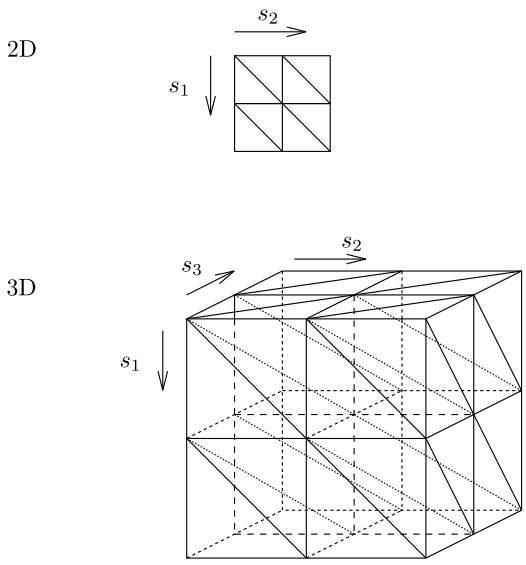


Figure 7.3: Top: Part of the 2-dimensional alignment matrix for two sequences. Bottom: Part of the 3-dimensional alignment matrix for three sequences.

7.5 An Exact Algorithm

The Needleman-Wunsch algorithm for pairwise global alignment can be generalized for the multiple alignment of k sequences $s_1, s_2, \ldots, s_k$ of lengths $n_1, n_2, \ldots, n_k$, respectively, in the obvious way (due to Waterman et al. (1976)). Here we give the formulation for distance minimization. Naturally, an equivalent algorithm exists for score maximization.

Instead of a two-dimensional alignment matrix, a k-dimensional alignment matrix is constructed, one dimension for each sequence. Each cell in the internal area of the matrix has $2^k - 1$ predecessors, and each transition $u \to v$ corresponds to a possible multiple alignment column c, weighted by its corresponding alignment score $\text{cost}(u \to v) = \text{cost}_{\text{SP}}(c)$. The optimal alignment problem translates to the problem of finding a path that minimizes the path weight, from the start cell $v_S = (0, 0, \ldots, 0)$ to the sink cell $v_E = (|s_1|, |s_2|, \ldots, |s_k|)$. The matrix is schematically illustrated in Figure 7.3 for $k = 3$ sequences.

Now we can return to the universal alignment algorithm from Chapter 5, here given in its dissimilarity (cost) version: For each cell v of the alignment matrix, compute in an appropriate order the dissimilarity value

$$D(v) = \min_{u \in \text{pred}(v)} \{D(u) + \text{cost}(u \to v)\}. \tag{7.1}$$

Applied to the k-dimensional alignment matrix, the algorithm allows to compute optimal multiple alignments.

Space and time complexity. The space complexity of this algorithm is given by the size of the k-dimensional alignment matrix. Obviously, this is $O(n_1 n_2 \cdots n_k)$ which is in $O(n^k)$ if n is the maximum sequence length.

The time complexity is even higher, since at each internal vertex where the sum-of-pairs cost is computed a minimization is taking place over $2^k - 1$ predecessor cells. Hence, the total time complexity (for homogeneous gap costs) is $O(n^k \cdot 2^k \cdot k^2)$. (For affine gap costs, the time complexity rises even more.)

While the time may be tolerable in some cases, the exponentially growing space complexity does not permit to run the algorithm on more than, say, six or seven typical protein sequences.

7.6 A Guide to Multiple Sequence Alignment Algorithms

As mentioned before, the sum-of-pairs multiple alignment problem is NP hard and all known exact algorithms have running times exponential in the number of sequences; thus aligning more than e.g. seven sequences of reasonable length is problematic. Even worse, from a biological point of view, tree alignment (see Chapter 8) should be the preferred choice, while its computational complexity is even higher than that of sum-of-pairs alignment.

Since in practice, one nevertheless needs to produce multiple alignments of (sometimes) hundreds of sequences, heuristics are required: **Progressive alignment** methods, discussed in Section 8, pick two similar sequences (or existing alignments) and align them to each other, and then proceed with the next pair, until only one alignment remains. These methods are by far the most widely used ones in practice. Another class of heuristics, called **segment-based** methods, first identify (unique) conserved segments and then chain them together to obtain multiple alignments.

There are, however, also some speedups possible when we stick to the original model of sum-of-pairs multiple alignment. One, according to an idea of Carrillo and Lipman (1988), is able to reduce the search space considerably in many practical cases, while still guaranteeing to find an optimal solution. Another one, the center-star method (Gusfield, 1993), is a simple 2-approximation, that means it is a method that no longer guarantees to find an optimal solution, but whose result is never more than a factor of two higher than the best possible solution. Finally, there is also a divide-and-conquer multiple alignment algorithm (Stoye, 1998) that does not come with any guaratees, but produces quite reasonable results in practice.

8 Tree Alignment and Progressive Alignment

From a biological point of view, sum-of-pairs scores are not very well motivated if the sequences under study originate from species that are related by a phylogenetic (evolutionary) tree. An alternative are models where the score reflects the tree-like relationship between the sequences. In this chapter we will discuss such a model, tree alignment, and a more pragmatic heuristic constructing multiple alignments in a tree-like fashion, progressive alignment.

8.1 Definition of Tree Alignment

To define the **tree score** of a multiple alignment, we assume that there additionally exists a given tree T (representing evolutionary relationships) with K nodes, whose k leaves represent the given sequences and whose $K-k$ internal nodes represent "deduced" sequences. The *extended* alignment A hence contains not only the given sequences $s_1, s_2, \ldots, s_k$, but also the (initially unknown) internal sequences $s_{k+1}, \ldots, s_K$ that must be found or guessed. A typical scenario is that the sequences at the leaves are orthologous proteins from different species whose phylogenetic relationship is well known, such as in Figure 8.1. Then the task is to find sequences at the inner nodes and an alignment of all sequences together, from which then the desired alignment of the leaf sequences is extracted.

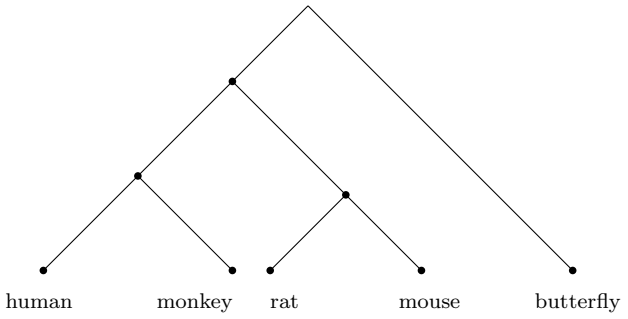


Figure 8.1: A phylogenetic tree

First we define the alignment cost for a given tree and alignment.

Definition 35 The **tree alignment cost** of an extended alignment A is the sum of the pairwise costs of all pairs of sequences that are connected by an edge in the tree:

$$\text{cost}^{(T)}_{\text{Tree}}(A) = \sum_{(p,q)\in E(T)} \text{cost}_2(\pi_{\{p,q\}}(A)),$$

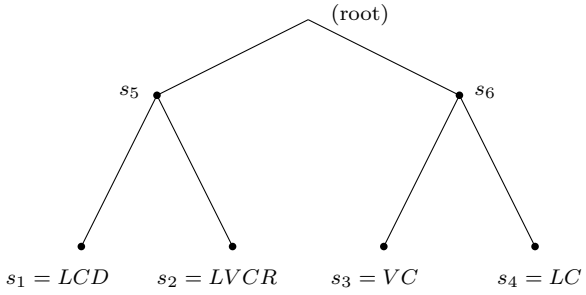
where $E(T)$ is the set of edges of T. (Clearly, as before, one can define the **tree alignment score** in a similar way.)

The tree alignment problem is now formally stated as follows.

Problem 36 (Tree Alignment Problem) Given a tree $T = (V, E)$ that has k sequences $s_1, \ldots, s_k$ attached to its leaves and a pairwise alignment cost function cost_2, find sequences $s_{k+1}, \ldots, s_K$ to be attached to the internal nodes of T and an alignment A of the sequences $s_1, \ldots, s_K$ such that $\text{cost}^{(T)}_{\text{Tree}}(A)$ is minimum.

The rationale behind tree alignment is that an alignment that minimizes the number of mutations along the branches of this tree probably best reflects the true evolutionary history and hence is most likely to be the correct one. This rationale is called the **parsimony principle**, and the sequences attached at internal nodes are called a **most parsimonious assignment**. (Obviously there is no guarantee that the most parsimonious assignment is the biologically correct one, and it can even be shown that in some special cases the most parsimonious assignment is likely to be wrong. In the field of phylogenetic tree studies that we are entering here, a long debate has been carried out about such topics, but that is another story.)

Example 37 Consider sequences $s_1 = LCD$, $s_2 = LVCR$, $s_3 = VC$, $s_4 = LC$ and the following tree with four leaves and two branching nodes:



If the sequences at the internal nodes are chosen as $s_5 = LVCD$ and $s_6 = LVC$ and as pairwise cost function cost_2 the unit cost function is used, then the following alignment

$$A = \left(\begin{array}{cccc} L & - & C & D \\ L & V & C & R \\ - & V & C & - \\ L & - & C & - \\ \hline L & V & C & D \\ L & V & C & - \end{array}\right)$$

has cost

$$\begin{aligned}
&\mathrm{cost}^{(T)}_{\mathrm{Tree}}(A) \\
&\quad = \mathrm{cost}_2(\pi_{\{1,5\}}(A)) + \mathrm{cost}_2(\pi_{\{2,5\}}(A)) + \mathrm{cost}_2(\pi_{\{3,6\}}(A)) + \mathrm{cost}_2(\pi_{\{4,6\}}(A)) \\
&\qquad + \mathrm{cost}_2(\pi_{\{5,6\}}(A)) \\
&\quad = 1 + 1 + 1 + 1 + 1 \\
&\quad = 5.
\end{aligned}$$

To find the sequences s_5 and s_6 achieving an overall optimal alignment is not so easy because of gaps. The next section explains a strategy. ◀

8.2 Solving the Tree Alignment Problem

The similarity to sum-of-pairs alignment already indicates that tree alignment may also be NP-hard, and this is indeed the case (Elias, 2006). However, for a single symbol the problem is easy, as we will see in Section 8.2.1. The general case is then handled in Section 8.2.2.

8.2.1 Fitch's Algorithm

Before we study the tree alignment problem itself, we first consider the following simpler problem:

Problem 38 (Minimum Mutation Problem) Given a phylogenetic tree T with characters attached to the leaves, find a labelling of the branching nodes of T with characters such that the overall number of character changes along the edges of T is minimized.

Here, the so-called **Fitch Algorithm** (Fitch, 1971) can be used, which we describe in its original version for the unit cost function and a rooted binary tree T. (If the tree is unrooted, a root can be arbitrarily chosen, the algorithm will always give the same result.)

1. **Bottom-up** phase: Assign to each internal node a set of potential labels.
   - For each leaf i set:
     $$R_i = \{x_i\} \quad (x_i = \text{character at leaf } i)$$
   - Bottom-up traversal of tree (from leaves to root)
     For internal node i with children j, k, set
     $$R_i = \begin{cases} R_j \cap R_k, & \text{if } R_j \cap R_k \neq \emptyset \\ R_j \cup R_k, & \text{otherwise.} \end{cases}$$
2. **Top-down** phase: Pick a label for each internal node.
   - Choose arbitrarily:

$$x_{root} = \text{ some } x \in R_{root}$$

- Top-down traversal of tree (from root to leaves)
  For internal node j with parent i, set

$$x_j = \begin{cases} x_i, & \text{if } x_i \in R_j \\ \text{some } x \in R_j, & \text{otherwise.} \end{cases}$$

See Figure 8.2 for an example of the Fitch Algorithm.

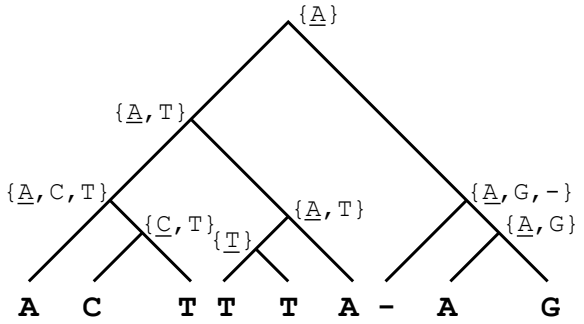


Figure 8.2: Illustration of the Fitch Algorithm for $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, \mathtt{-}\}$. The characters assigned during the top-down phase are underlined. The total number of character changes is 5.

The analysis of the Fitch Algorithm is easy: It takes $O(k|\Sigma|)$ time and space, because the number of internal nodes is bounded by $k-1$ and each step takes $O(|\Sigma|)$ time.

Note 1: A bottom-up traversal of a tree can be implemented by a *post-order traversal* that starts at the leaves of the tree and at each internal node recursively traverses the subtrees before visiting the node itself.

Note 2: A top-down traversal of a tree can be implemented by a *pre-order traversal* (also called *depth-first traversal*) that starts at the root of the tree, and for each internal node recursively visits the node first and then traverses the subtrees.

8.2.2 Sankoff's Algorithm

Now we return to the general case, the tree alignment problem (Problem 36). An exponential-time exact algorithm solving it is due to Sankoff (1975). It is conceptually similar to the algorithm described in Section 7.5 for the sum-of-pairs cost in that it also acts on a k-dimensional alignment matrix. The main difference is that in each step, when the cost of one of the $2^k - 1$ predecessor nodes is computed, this is not the SP alignment cost of the preceding letters in the k sequences, but it is the tree alignment cost. Since this cost includes letters from the (initially unknown) sequences $s_{k+1}, \ldots, s_K$, optimal choices for these have to be computed on the fly, and that is exactly where the Fitch algorithm comes in.

Technical details of Sankoff's algorithm go beyond the scope of this class. We just state that the time complexity is $O(n^k 2^k k|\Sigma|)$, which is even higher than for sum-of-pairs alignment because of the inner optimization generating the new sequences at the inner nodes of T. Knudsen (2003) shows how this algorithm can be generalized to affine gap costs.

8.3 Generalized Tree Alignment

Although Tree Alignment (Problem 36) is already NP-hard, there exists an even harder problem, the **generalized tree alignment problem**. Recall that in the tree alignment problem the tree T is given. In practice, however, the tree is often unknown and an additional optimization parameter. Thus the problem (in its distance version) is as follows.

Problem 39 (Generalized Tree Alignment Problem) Given sequences $s_1, \ldots, s_k$, find

1. a tree T,
2. sequences $s_{k+1}, \ldots, s_K$ to be assigned to the internal vertices of T, and
3. a multiple alignment A of the sequences $s_1, \ldots, s_K$,

such that the tree alignment cost $\text{cost}^{(T)}_{\text{Tree}}(A)$ is minimum among all such settings.

Various attempts have been made to address this problem, but only with very limited practical success.

The first way is conceptually simple, but very inefficient: Just apply Sankoff's algorithm to all tree topologies (of which there are exponentially many in k) and pick the best one. Clearly, this is feasible only for very small k.

An alternative approach is to solve the Steiner tree problem in sequence space. Therefore the generalized tree alignment problem is equivalently formulated as follows:

Problem 40 Given the complete weighted graph whose vertices V represent the infinite set Σ^* of all sequences over Σ and whose edges have as weights the edit distance between the connected vertices (the so-called **sequence space**) and a finite subset $V' \subseteq V$, find a tree T with vertices $V'' \subseteq V$ such that $V' \subseteq V''$ and the total weight of the edges in T is minimum. Such a tree is called a **Steiner tree** for V'.

For two-letter sequences over the alphabet $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, an example following this approach is visualized in Figure 8.3.

Again, we do not give the technical details of an algorithm solving the generalized tree alignment problem in this way, but we invite the reader to observe the elegance of the approach while also the infeasibility in practice.

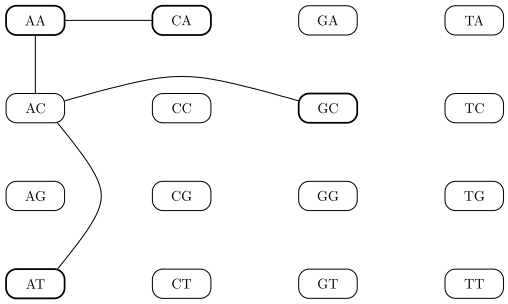


Figure 8.3: The generalized tree alignment problem for two-letter sequences as a Steiner tree problem in sequence space. Given are the sequences `AA`, `CA`, `AT`, and `GC`. The tree shown contains one additional internal "Steiner" node `AC` and has total cost 4.

8.4 Progressive Alignment

The **progressive alignment** method is a fast heuristic multiple alignment method. The most popular multiple alignment programs follow this strategy.

The basic idea of progressive alignment is that the multiple alignment is computed in a *progressive* fashion. In its simplest version, the given sequences $s_1, s_2, \ldots, s_k$ are added one after the other to the growing multiple alignment, i.e., first an alignment of s_1 and s_2 is computed, then s_3 is added, then s_4, and so on.

In order to proceed this way, a method is needed to align a sequence to an already given alignment. Obviously, this is just a special case of aligning two alignments to each other, and in Section 8.4.1 we will discuss a simple algorithm to do this.

In addition, the order in which the sequences are added to the growing alignment can be determined more freely than just following the input order. Often, the most similar sequences are aligned first in order to start with a well-supported, error-free alignment.

A more advanced version of progressive alignment that is motivated by the tree alignment approach described above is the progressive alignment along the branches of a rooted **alignment guide tree**. Like in tree alignment, a phylogenetic tree is given that carries the given sequences at its leaves. However, unlike in tree alignment, no global objective function is optimized, but instead multiple alignments are assigned to the internal nodes in a greedy fashion bottom up, from the leaves towards the root of the tree. Figure 8.4 illustrates this procedure.

Benefits of the progressive approach are:

- The method is more efficient than the exact tree alignment algorithm. Most algorithms following this strategy have a quadratic time complexity $O(n^2k^2)$.
- Since the sequences near each other in the guide tree are supposed to be similar, in the early stages of the algorithm alignments will be calculated where errors are unlikely. This will reduce the overall error rate.

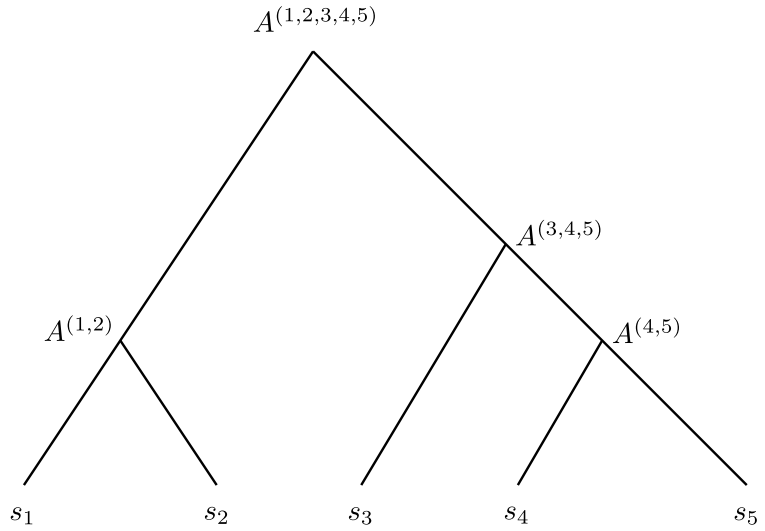


Figure 8.4: Progressive alignment along the branches of a phylogenetic tree. Sequences s_1 and s_2 are aligned, giving alignment $A^{(1,2)}$. Sequences s_4 and s_5 give $A^{(4,5)}$ which then is aligned with s_3 giving $A^{(3,4,5)}$. Finally, aligning $A^{(1,2)}$ and $A^{(3,4,5)}$ gives the multiple alignment of all sequences, $A^{(1,2,3,4,5)}$.

- Motifs that are family specific will be recognized early, and so they won't be superposed by errors of remote motifs.

However, there are also a few potential disadvantages:

- Early errors can not be revoked, even if further information becomes available in a later step in the overall procedure, see Figure 8.5. Feng and Doolittle (1987) coined the term "once a gap, always a gap" to describe this effect.

- Because of its procedural definition, the progressive alignment approach does not optimize a well-founded global objective function. This means that it is difficult to evaluate the quality of the result, since there is not a single value that is to be maximized or minimized and can be compared to the result of heuristic approaches.

- The method relies on the alignment guide tree. The tree must be known (or computed) before the method can start, and an error in the tree can make a large difference in the resulting alignment. Since multiple alignments are often used as a basis for construction of phylogenetic trees, here we have a typical "chicken and egg" problem, and in phylogenetic analyses one should be especially careful not to obtain trivial results with progressive alignments.

8.4.1 Aligning Two Alignments

An important subprocedure of the progressive alignment method is to align two existing multiple alignments.

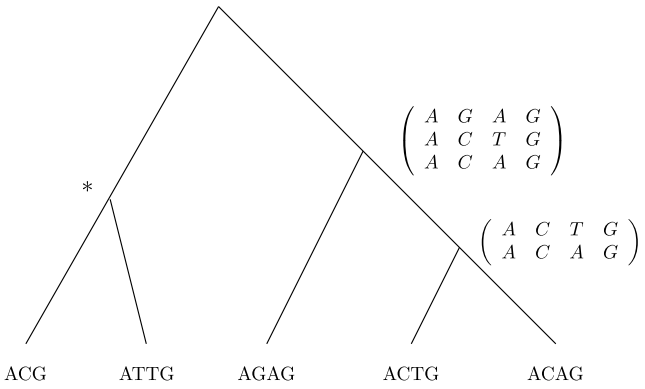


Figure 8.5: In a strict bottom-up progressive computation, it cannot be decided at the time of computing the alignment denoted with the asterisk ($*$) if it should be $\left(\begin{smallmatrix}\mathtt{A}&\mathtt{-}&\mathtt{C}&\mathtt{G}\\ \mathtt{A}&\mathtt{T}&\mathtt{T}&\mathtt{G}\end{smallmatrix}\right)$ or $\left(\begin{smallmatrix}\mathtt{A}&\mathtt{C}&\mathtt{-}&\mathtt{G}\\ \mathtt{A}&\mathtt{T}&\mathtt{T}&\mathtt{G}\end{smallmatrix}\right)$. Only the rest of the tree indicates that the second variant is probably the correct one.

Since the two alignments are fixed, this is a *pairwise* alignment procedure, with the only extension that the two entities to be aligned are not sequences of letters but sequences of alignment columns.

Therefore, it is necessary to provide a score for the alignment of two alignment columns. One way to define such an extended score is to add the scores of all pairwise (letter-letter) scores between the two columns.

For example, consider the two alignment columns

$$\begin{pmatrix}\mathtt{A}\\ \mathtt{G}\\ \mathtt{T}\\ \mathtt{-}\end{pmatrix} \quad \text{and} \quad \begin{pmatrix}\mathtt{A}\\ \mathtt{G}\end{pmatrix}.$$

In a unit cost scenario with matches of cost $\text{cost}(c,c)=0$ and mismatches and indels of cost $\text{cost}(c,c')=1$ for $c,c' \in \Sigma \cup \{-\}$, $c \neq c'$, an alignment of these two columns would be scored $\text{cost}(\mathtt{A},\mathtt{A})+\text{cost}(\mathtt{A},\mathtt{G})+\text{cost}(\mathtt{G},\mathtt{A})+\text{cost}(\mathtt{G},\mathtt{G})+\text{cost}(\mathtt{T},\mathtt{A})+\text{cost}(\mathtt{T},\mathtt{G})+\text{cost}(\mathtt{-},\mathtt{A})+\text{cost}(\mathtt{-},\mathtt{G}) = 0+1+1+0+1+1+1+1=6$. In general, the cost of aligning a column A_i of an alignment A with k_A rows and a column B_j of an alignment B with k_B rows is

$$\text{cost}(A_i, B_j) = \sum_{\substack{p \in \{1,\ldots,k_A\} \\ q \in \{1,\ldots,k_B\}}} \text{cost}(A_i[p], B_j[q]).$$

Based on such a column-column score, the dynamic programming algorithm can be performed as in the case of the alignment for two sequences.

The following example illustrates the procedure for the alignment of

$$A_1 = \begin{pmatrix}\mathtt{T}&\mathtt{A}&\mathtt{G}\\ \mathtt{G}&\mathtt{-}&\mathtt{C}\end{pmatrix} \quad \text{and} \quad A_2 = \begin{pmatrix}\mathtt{A}&\mathtt{T}&\mathtt{C}&\mathtt{A}&\mathtt{G}\\ \mathtt{A}&\mathtt{G}&\mathtt{C}&\mathtt{-}&\mathtt{G}\end{pmatrix}$$

using the unit cost model described above.

	ε	$\binom{\mathtt{A}}{\mathtt{A}}$	$\binom{\mathtt{T}}{\mathtt{G}}$	$\binom{\mathtt{C}}{\mathtt{C}}$	$\binom{\mathtt{A}}{-}$	$\binom{\mathtt{G}}{\mathtt{G}}$
ε	0	4	8	12	14	18
$\binom{\mathtt{T}}{\mathtt{G}}$	4	4	6	10	12	16
$\binom{\mathtt{A}}{-}$	6	6	8	...		
$\binom{\mathtt{G}}{\mathtt{C}}$	10					

The total time complexity of the algorithm is $O(n_A n_B k_A k_B)$ where n_A and n_B are the alignment lengths and, as above, k_A and k_B are the numbers of rows in the alignments A and B, respectively. However, it can be reduced to $O(n_A n_B)$ time if the alphabet $\Sigma = (c_1, c_2, \ldots, c_\sigma)$ is of constant size σ and an alignment column A_i is stored as a vector $(a_{c_1}, a_{c_2}, \ldots, a_{c_\sigma}, a_-)$ where a_c is the number of occurrences of character c in A_i, such that the cost of aligning two columns $A_i = (a_{c_1}, a_{c_2}, \ldots, a_{c_\sigma}, a_-)$ and $B_j = (b_{c_1}, b_{c_2}, \ldots, b_{c_\sigma}, b_-)$ can be written as

$$\text{cost}(A_i, B_j) = \sum_{c,c' \in \Sigma \cup \{-\}} a_c \cdot b_{c'} \cdot \text{cost}(c, c').$$

Adding affine gap costs is possible, but the exact treatment is rather complicated, in fact NP-hard (Kececioglu and Starrett, 2004). That is why usually heuristic approaches are used for affine gap costs in the alignment of two alignments.

8.5 Software for Progressive Alignment

8.5.1 The Family of Clustal Programs

The series of Clustal programs was developed by Julie Thompson and Des Higgins. The initial version was Clustal V (Higgins and Sharp, 1992), followed by Clustal W (Thompson et al., 1994). Later, an X windows interface was introduced, the package now being called Clustal X (Thompson et al., 1997). The newest and presumably last member of this family is Clustal Ω (Sievers et al., 2011).

In principle, the Clustal algorithm follows quite precisely the idea of progressive multiple alignment along a guide tree as described above, although the actual implementation is enhanced by several features regarding the optimized computation of a reasonable guide tree and the automatic selection of scoring schemes, which we do not discuss here.

The steps of the algorithm are the following:

1. By pairwise comparison of all input sequences $s_1, s_2, \ldots, s_k$, compute all pairwise optimal alignment similarity scores $s(s_i, s_j)$.
2. From these scores, compute an alignment guide tree using the Neighbor Joining algorithm (Saitou and Nei, 1987).
3. Finally, progressively compute the multiple alignment, guided by the branching order of the tree.

Clustal Ω includes much expert knowledge (optimized use of different protein scoring matrices, affine gap costs, etc.), so that the alignments are not only quickly computed but also of high quality, often better than those just optimized under some theoretical objective function. This and its easy-to-use interface are probably the main reasons for the great success of the program.

8.5.2 T-COFFEE

Another method to compute high-quality multiple alignments is the program T-COFFEE (Notredame et al., 2000). The procedure consists of three steps.

In the first step, a **primary library** of local and/or global alignments is computed. These alignments can be created in any way. Sequences can be contained in several alignments, and the different alignments do not need to be consistent with each other. The primary library can consist, for example, of optimal pairwise global alignments, optimal pairwise local alignments, suboptimal local alignments, heuristic multiple alignments, possibly several ones computed with different scoring schemes, etc. In the default setting, T-COFFEE uses Clustal W to create one global multiple alignment and also to compute global pairwise alignments for all pairs of sequences.

Then, in the second phase, the alignments from the primary library are combined to produce a **position-specific library** that tells for each pair of aligned sequence positions from the primary library the strength of its weight, i.e., the number of alignments of the primary library that support the pairing of these two positions. An advantage is that in this extension phase no substitution matrix is used, and so different scoring schemes from the construction of the primary library will not lose their influence on the final result.

In the third phase, ideally one would like to compute a maximum weight alignment (Kececioglu, 1993) from the (weighted) alignments of the extended library. Unfortunately this is a computationally hard problem. This is why a heuristic is used that is similar to the progressive alignment strategy described in the previous section.

8.5.3 MUSCLE

MUSCLE is public domain multiple alignment software for protein and nucleotide sequences. **MUSCLE** stands for **Mu**ltiple **S**equence **C**omparison by **L**og-**E**xpectation. It was designed by Robert C. Edgar (Edgar (2004a), Edgar (2004b)) and can be found at `http://www.drive5.com/muscle/`.

MUSCLE performs an iterated progressive alignment strategy and works in three stages. At the completion of each stage, a multiple alignment is available and the algorithm can be terminated.

Stage 1: Draft progressive The first stage builds a progressive alignment, similar to Clustal.

Similarity measure The similarity of each pair of input sequences is computed, either using k-mer counting or by constructing a global alignment of the pair and determining the fractional identity.

Distance estimate A triangular distance matrix D_1 is computed from the pairwise similarities.

Tree construction A tree T_1 is constructed from D_1 using UPGMA or neighbor-joining, and a root is identified.

Progressive alignment A progressive alignment is built by a post-order traversal of T_1, yielding a multiple alignment MSA_1 of all input sequences at the root.

Stage 2: Improved progressive The second stage attempts to improve the tree and builds a new progressive alignment according to this tree. This stage may be iterated.
The main source of error in the draft progressive stage is the approximate k-mer distance measure, which results in a suboptimal tree. MUSCLE therefore re-estimates the tree using the Kimura distance, which is more accurate but requires an alignment.

Similarity measure The similarity of each pair of sequences is computed using fractional identity computed from their mutual alignment in the current multiple alignment.

Tree construction A tree T_2 is constructed by computing a Kimura distance matrix D_2 (Kimura distance for each pair of input sequences from MSA_1) and applying a clustering method (UPGMA) to this matrix.

Tree comparison Compare T_1 and T_2, identifying the set of internal nodes for which the branching order has changed. If Stage 2 has executed more than once, and the number of changed nodes has not decreased, the process of improving the tree is considered to have converged and iteration terminates.

Progressive alignment A new progressive alignment is built. The existing alignment is retained of each subtree for which the branching order is unchanged; new alignments are created for the (possibly empty) set of changed nodes. When the alignment at the root (MSA_2) is completed, the algorithm may terminate, repeat this stage or go to Stage 3.

Stage 3: Refinement The third stage performs iterative refinement using a variant of tree-dependent restricted partitioning (Hirosawa et al., 1995).

Choice of bipartition An edge is deleted from T_2, dividing the sequences into two disjoint subsets (a bipartition). (Bottom-up traversal)

Profile extraction The multiple alignment of each subset is extracted from the current multiple alignment. Columns containing no characters (i.e., indels only) are discarded.

Re-alignment A new multiple alignment is produced by re-aligning the two multiple alignments to each other using the technique described in Section 8.4.1 on page 63 (Aligning two Alignments).

Accept/reject The SP-score of the multiple alignment implied by the new alignment is computed.

$$MSA_2 = \begin{cases} MSA_2 \text{ (accept)}, & \text{if } S_{SP}(MSA_2) > S_{SP}(MSA_1) \\ MSA_1 \text{ (discard)}, & \text{otherwise.} \end{cases}$$

Stage 3 is repeated until convergence or until a user-defined limit is reached. Visiting edges in order of decreasing distance from the root has the effect of first realigning individual sequences, then closely related groups

9 Genome Assembly

As long as there is no technology to sequence a chromosome from telomere to telomere in one piece, genome assembly will remain one of the most important sequence analysis tasks in bioinformatics. Informally, the genome assembly problem is easy to phrase:

Definition 41 (Genome Assembly, informal version) Given a (usually large) set of DNA sequencing reads, find a genome sequence (or several chromosomes in case of a multichromosomal species) that contains all the reads.

This problem formulation is clearly underspecified, since it does not explain how exactly the genome sequence relates to the reads and what other properties it should have. An extreme in the other direction is the **shortest common superstring (scs)** problem:

Definition 42 (Genome Assembly, scs version) Let S be a set of finite strings, find a shortest sequence g such that each $s \in S$ is a substring of g.

This version is unrealistic because in real data we can not expect that every sequenced read is an error-free extract from the genome. Moreover, there may be regions in the genome that are not covered by any reads and therefore may produce gaps in the final output sequence.

Moreover, the shortest common superstring problem is NP-hard (Maier and Storer, 1977), and more robust variants of it, that may be applicable here, are even more difficult to solve optimally. Therefore usually heuristic methods are applied, and in this chapter we will discuss the two most relevant of these.

9.1 Overlap, Layout, Consensus

The first practical genome assemblers all followed this scheme, which has been refined over the years. Here we explain the general ideas of the three steps: Overlap, Layout, Consensus (OLC). Several variants have been designed and implemented in many genome assembly software tools. Such details are not part of these lecture notes.

Overlap. Given the input set $S = \{s_1, \ldots, s_k\}$ of sequencing reads, compute overlaps between all pairs (s_i, s_j), and between all pairs $(s_i, \overleftarrow{s_j})$, $1 \leq i < j \leq k$. (The second pair is necessary because from a sequencing read it is not possible to see which DNA double-strand it originates from. Therefore both possible ways of overlap have to be tested.) The method of choice for these comparisons is free end gap alignment (Section 5.3), and usually also some filter is applied so that only substantial overlaps with a high score are recorded in a set of overlaps O.

Layout. A weighted graph data structure is computed, called the **overlap graph** $G(S, O)$, with the following components:

- for each string $s \in S$, $G(S, O)$ contains two vertices s^t (tail) and s^h (head), connected by a directed **read edge** from s^t to s^h of weight $l(s)$, where $l(s)$ is the length of read s;
- for each overlap $o \in O$, $G(S, O)$ contains an undirected **overlap edge** of weight $-\text{ovl}(o)$ connecting the two vertices representing the overlapping read extremities, where $\text{ovl}(o)$ is the overlap length of overlap o.

The overlap phase ends with some way to extract a set of disjoint long paths from this graph that will correspond to contiguous chromosome regions (**contigs**). A popular way is to use some kind of spanning tree heuristic for this step. The paths are collected in a set of contigs C.

Consensus. For each contig $c \in C$ compute a consensus sequence, usually by performing some multiple alignment procedure involving the read sequences along the path of the contig.

An important extension of the method arises from the fact that many sequencing techniques allow to obtain pairs of reads (*mate pairs* or *paired-end reads*), for which then their approximate distance and their relative orientation on the chromosome is known. This information can be included in the sequencing procedure, either in form of additional edges in the overlap graph, or as a heuristic for error correction during a postprocessing step. Details are omitted here.

A rough analysis of the OLC method is the following under the assumption that we have given k reads, each of length up to n: The overlap phase takes $O(k^2n^2)$ time and $O(k^2+n)$ space to compute $O(k^2)$ free end gap alignments, each in time $O(n^2)$ and space $O(n)$. The layout and consensus phases are not so easy to analyse, but it is clear that in the layout phase the overlap graph requires very much memory because it has to be stored in memory all at once. Therefore it is often a challenge to run an OLC assembler on very large, for example eukaryotic datasets.

9.2 Assembly Using de Bruijn Graphs

When in the early 2000s new "next generation" sequencing technologies were developed, producing much faster and cheaper than before huge datasets of hundreds of millions or even billions of (initially quite short) reads, it became impossible to assemble them by the OLC method, in particular because the overlap graph, that has two vertices for each read and potentially a quadratic number of edges, required too much memory. This led to the development of a new line of assembly algorithms whose memory consumption is independent of the input data size.

The idea is to construct a de Bruijn subgraph (see Section 2.4) of all k-mers present in the reads. Most methods also add the k-mers of the reverse complements of all reads, again because it is unknown from which DNA double-strand a read originates. The de

Bruijn graph is then usually cleaned by removing areas that are not well supported by the reads, i.e. have a low coverage. Also small "bubbles" that often originate from sequencing errors are removed ("flattened"). Finally, like in the layout phase of the OLC method, long non-branching paths are extracted and reported as contigs of the genome sequence.

9.3 Hybrid Assembly

With the advent of long read sequencing (Pacific Biosciences, Oxford Nanopore Technologies) in the late 2010s, the demand for new genome assembly techniques arose again. Genome projects now often produced two sets of reads, a large number of short reads with very high sequence correctness, and a smaller number of long reads to assist in the layout phase. To accommodate such datasets, so-called **hybrid assemblers** have been developed that can combine both types of reads. Different strategies can be followed:

- Gap filling and assembly upgrade: Assemble the short reads in the traditional way using de Bruijn graphs and then stitch together the contigs using the long reads.
- Long read only with read error correction: Map the short reads to the long reads in order to correct sequencing errors and then assemble the corrected long ones using variants of OLC assembly.
- Long read only with genome correction: Assemble the long reads by an OLC assembler and then map the short reads to the resulting genome in order to correct sequencing errors.

Note that these are only the general strategies. Details are omitted and differ widely between the individual implementations. The development of new genome assemblers is still an active area of research, and with each new sequencing technology also new assemblers have to be developed.

For example, due to their context information, very long reads are more and more used to directly assemble the two haplotypes of a diploid genome separately, which is very useful in population genomic studies and to associate diseases to certain genomic loci.

10 Suffix Trees

10.1 Motivation

The amount of sequence information in today's databases is growing rapidly. This is especially true for the domain of genomics: Past, current and future projects to sequence large genomes produce terabytes of sequence data, mainly DNA sequences and protein sequences. To make use of these sequences, larger and larger instances of string processing problems have to be solved.

While the sequence databases are growing rapidly, the sequence data they already contain does not change much over time. In this situation, indexing methods are applicable. These methods preprocess the sequences in order to extract relevant information and store this in form of an index.

There are two basic approaches to **index-based database searching**.

1. In the first (so-called **pattern-index**) approach, the database (of size N) is completely examined for every query (of size m). Each query, however, can be preprocessed as soon as it becomes known. This means that the running time of such a method is at least $\Theta(m+N)$ for each query, even if no similar sequences are found. This is in practice much faster than $O(mN)$ time for a full alignment, and it allows that the database changes after each query (e.g. new sequences might be added). **BLAST** (Section 6.4) is a well-known database search program that works in this way.

2. The second (**text-index**) approach preprocesses (indexes) the database before a number of queries are posed, assuming that the database changes only rarely because indexing takes time: Even if indexing time is only linear in the database size, the constant factor is usually quite high. On the other hand the index allows to immediately identify only those regions of the database that are potentially similar to the query. If these do not exist, the time spent for each query can become as small as $\Theta(m)$. Early methods using a text-index were based on the q-gram lemma (Section 6.2), and many more have been developed since, mostly based on the datastructures introduced in subsequent chapters.

The following table shows time complexities for pattern-index and text-index database searching if no similar regions are found. If there are such regions, they have to be examined, of course, which adds to the time complexities of the methods. However, the goal is to spend as little time as possible when there are no interesting similarities.

Method	Preprocessing	Querying	Total 1 query	Total k queries
pattern index	$O(m)$	$O(N)$	$O(m+N)$	$O(k(m+N))$
text index	$O(N)$	$O(m)$	$O(N+m)$	$O(N+km)$

It is clear that text indexing pays off as soon as the number of queries k on the same database becomes reasonably large.

The **suffix tree** is a basic index data structure for sequences that can be used both as a pattern index and as a text index. In this chapter we introduce the concept of suffix trees and show their most important properties, and in the next one (Chapter 11) we take a look at some applications of biological relevance.

Subsequently, in Chapter 12, we shall introduce the data structures **suffix array** and **enhanced suffix array**, which share many of the properties of suffix trees, but have a smaller memory footprint and perform better in practice. Even smaller is the memory footprint of the **Burrows-Wheeler Transformation** that follows in Chapters 13 and 14.

10.2 An Informal Introduction to Suffix Trees

Let us consider a constant alphabet Σ and a string $s \in \Sigma^n$. Think of s as a database of concatenated sequences, or a genome, etc.

In order to answer substring queries about s, e.g. to enumerate all substrings of s that satisfy certain properties, it is helpful to have an index of all substrings of s. We already know that there can be $O(n^2)$ distinct substrings of s, i.e., too many to represent them all *explicitly* with a reasonable amount of memory. We need an *implicit* representation.

Observation 43 Each substring of s is a prefix of a suffix of s, and s has only n suffixes, so we can consider an index consisting of all suffixes of s.

Now, let $\$ \notin \Sigma$ be a special character, sometimes called **sentinel**[1], appended to the end of s, so that no suffix is a prefix of another suffix. For example, if $s =$ `cabca`, the suffixes of `cabca$` are

$$\texttt{\$}, \texttt{a\$}, \texttt{ca\$}, \texttt{bca\$}, \texttt{abca\$}, \texttt{cabca\$}.$$

This list has $n+1 = \Theta(n)$ elements, but its total size is still $\Theta(n^2)$ because there are $n/2$ suffixes of length $\geq n/2$.

It is a central idea behind suffix trees to identify **common prefixes** of suffixes, which is achieved by *lexicographically sorting* the suffixes. This naturally leads to a rooted tree structure, as shown in Figure 10.1.

Note the effect of appending the sentinel \$: In the tree for cabca\$, every leaf corresponds to one suffix. Without the sentinel, some suffixes can end in the middle of an edge or at an internal node. To be precise, these would be exactly those suffixes that are a prefix of another suffix; we call them **nested suffixes**. Appending \$ ensures that there are no

[1] It guards the end of the string. Often we assume that it is lexicographically smaller than any character in the alphabet.

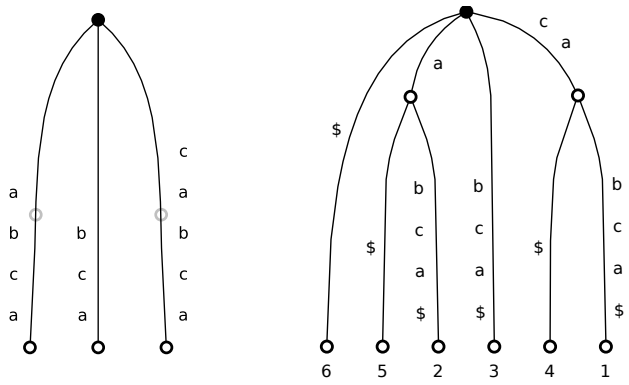


Figure 10.1: Suffix trees of `cabca` (left) and of `cabca$` (right). In the left tree, the hollow circles indicate suffixes that are also prefixes of other suffixes (**nested suffixes**). In the right tree, as a result of appending the sentinel character, this is no longer the case. Moreover, the leaves have been annotated with the starting positions of the suffixes.

nested suffixes, because \$ does not occur anywhere in s. Thus we obtain a one-to-one correspondence of suffixes and leaves in the suffix tree.

Note the following properties for the suffix tree T of $s\$ = \texttt{cabca\$}$, or the larger example shown in Figure 10.2.

- There is a bijection between suffixes of $s\$$ and leaves of T.
- Each internal node has at least two children.
- Each outgoing edge of an internal node begins with a different letter.
- Edges are annotated with substrings of $s\$$.
- Each substring s' of $s\$$ can be found by following a path from the root down the tree for $|s'|$ characters. Such a path may end in the middle of an edge, not necessarily at a leaf or at an internal node.

We now give more formal definitions and shall see that a "clever" representation of a suffix tree needs only linear, i.e., $O(n)$ space. Most importantly, we cannot store strings at the edges, because the sum of their lengths is still $O(n^2)$. Instead, we will store *references* to substrings of the input string $s\$$, which is always possible since each edge label is a substring of $s\$$, as noted above.

10.3 A Formal Introduction to Suffix Trees

Definitions. Let Σ be a constant alphabet. A **Σ-tree**, also called **trie**, is a rooted tree (see Section 2.3) whose edges are labeled with a *single character* from $\Sigma \cup \{\$\}$ in such a way that no node has two outgoing edges labeled with the same letter.

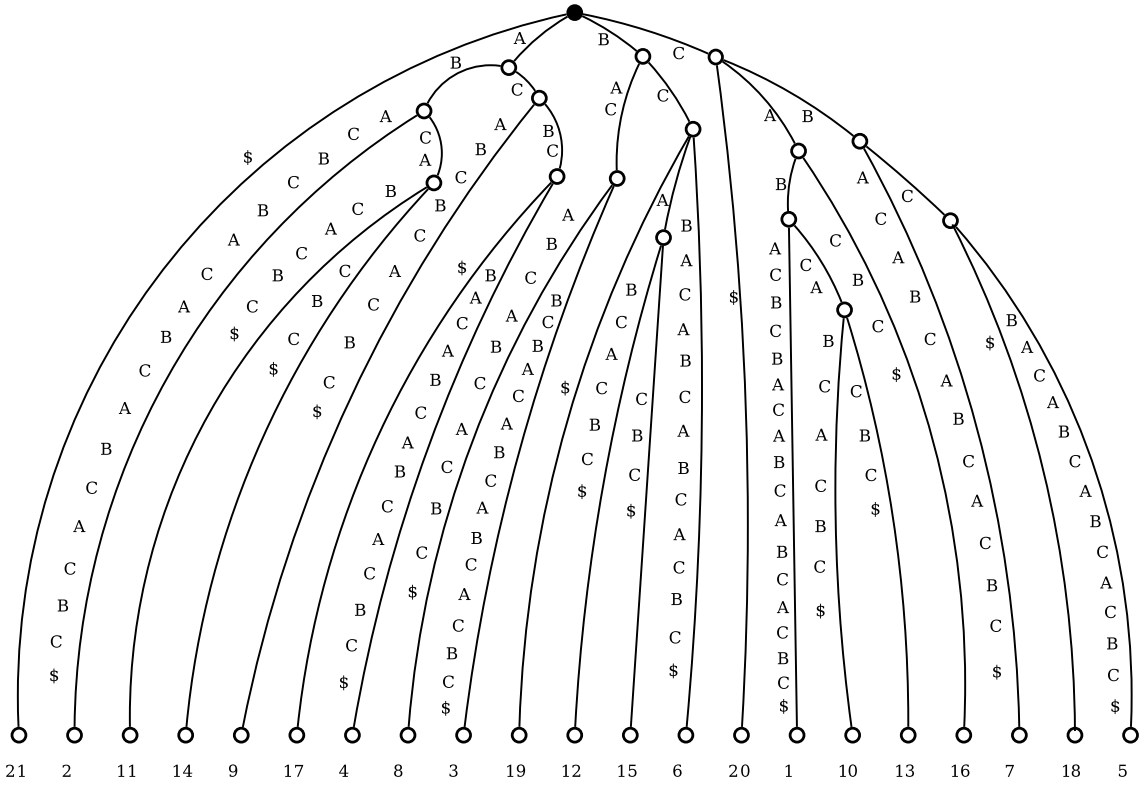


Figure 10.2: The suffix tree of `CABACBCBACABCABCACBC$`.

A **Σ^+-tree** is a rooted tree whose edges are labeled with *non-empty strings* over $\Sigma \cup \{\$\}$ in such a way that no node has two outgoing edges whose labels start with the same letter. A Σ^+-tree is **compact** if no node (except possibly the root) has exactly one child (i.e., all internal nodes have at least two children).

For a node v in a Σ- or Σ^+-tree T, we call **string(v)** the concatenation of the edge labels on the unique path from the root to v. We define the **string depth** stringdepth(v) := |string(v)|. In a Σ-tree, this is equal to depth(v), but in a Σ^+-tree, the depth of a node usually differs from its string depth because one edge can represent more than one character. We always have stringdepth(v) $\geq$ depth(v).

For a string x, if there exists a node v with string(v) = x, we write **node(x)** for v. Otherwise node(x) is undefined. Sometimes, one can also find $\overline{x}$ written for node(x) in the literature. Of course, node(ε) = $\overline{\varepsilon}$ is the root r.

We say that T **displays** a string $x \in \Sigma^*$ if x can be read along a path down the tree, starting from the root, i.e., if there exists a node v and a (possibly empty) string y such that xy = string(v). We finally define the **words** displayed by T by words(T) := $\{x \mid T$ displays $x\}$.

The **suffix tree** of s is the compact Σ^+-tree T with words(T) = $\{s' \mid s'$ is a substring of $s\}$. As mentioned above, we often consider the suffix tree of $s\$$ for some sentinel character

$\$ \notin \Sigma$, where each suffix ends in a leaf.

An edge leading to an internal node is an **internal edge**. An edge leading to a leaf is a **leaf edge**.

Generalized suffix trees. In many applications, we need a suffix tree built from more than one string (e. g. to compare two genomes). There is an important difference between the set of k suffix trees (one for each of k strings) and one (big) suffix tree for the concatenation of all k strings. The big suffix tree is very useful, the collection of small trees is generally useless!

When we concatenate several sequences into a long one, however, we need to make sure to separate the strings appropriately in such a way that we do not create artificial substrings that occur in none of the original strings.

For example, if we concatenate ab and ba without separating them, we would get $abba$, which contains bb as a substring, but bb does not occur in either of the original strings.

Therefore we use additional unique sentinel characters $\$_1, \$_2, \ldots$ that delimit each string.

Definition 44 Given strings $s_1, \ldots, s_k \in \Sigma^*$, the **generalized suffix tree** of $s_1, \ldots, s_k$ is the suffix tree of the string $s_1\$_1 s_2\$_2 \ldots s_k\$_k$, where $\$_1 < \$_2 < \cdots < \$_k$ are distinct sentinel characters that are not part of the underlying alphabet Σ.

In the case of only two strings, we usually use # to delimit the first string for convenience, thus the generalized suffix tree of s and t is the suffix tree of $s\#t\$$.

10.4 Space requirements of Suffix Trees

Note that, in general, the *suffix trie* τ of a string s of length n contains $O(n^2)$ nodes. We now show that the number of nodes in the *suffix tree* T of s is linear in n.

Lemma 45 The suffix tree T of a string of length n has at most $n - 1$ internal nodes.

Proof. Let L be the number of leaves, let I be the number of internal nodes, and let E be the number of edges in T. Each leaf and each internal node except the root has exactly one incoming edge; thus $E = L + I - 1$. On the other hand, each internal node is branching, i.e., has at least two outgoing edges; thus $E \geq 2I$. It follows that $L + I - 1 \geq 2I$, or $I \leq L - 1$. Since $L \leq n$ (there can not be more leaves than suffixes), we have $I \leq n - 1$ and the lemma follows. Also note that $E = L + I - 1 \leq 2n - 2$. □

By the above lemma, there are at most n leaves, $n - 1$ internal nodes and $2n - 2$ edges; all of these are linear in the string length n. The remaining problem are the edge labels, which are substrings of s and may each require $O(n)$ space for a total of $O(n^2)$. To avoid this, we do not store the edge labels explicitly, but only two numbers per edge: the start- and end-position of a substring of s that spells the edge label. The following theorem is now an easy consequence.

Theorem 46 The suffix tree of a string s of length n can be stored in $O(n)$ space.

Corollary 47 The generalized suffix tree of several strings $s_1, \ldots, s_k$ can be stored in $O(\sum_{i=1}^{k} |s_i|)$ space.

10.5 Suffix Tree Construction: The WOTD Algorithm

Suffix tree constructions have a long history and there are algorithms which construct suffix trees in linear time (Weiner, 1973; McCreight, 1976; Ukkonen, 1995; Farach, 1997).

Here we describe a simple suffix tree construction method that has quadratic worst-case time complexity, but is fast in practice and easy to explain: the **Write Only Top Down** (WOTD) suffix tree construction algorithm (Giegerich et al., 2003).

We assume that the input string $s\$$ ends with a sentinel character.

The WOTD algorithm adheres to the recursive structure of a suffix tree. Let u be a substring of $s\$$ such that node(u) is a branching node in the suffix tree T of $s\$$. The idea is that node(u) is **evaluated** recursively, creating the subtree below node(u) from the set of all suffixes of $s\$$ that have u as a prefix. To construct this subtree, we need the set

$$R(\text{node}(u)) := \{v \mid uv \text{ is a suffix of } s\$\}$$

of **remaining suffixes**. To store this set, we would not store the suffixes explicitly, but only their starting positions in $s\$$. More precisely, we proceed as follows.

1. At first $R(\text{node}(u))$ is divided into groups according to the first character of each suffix. For any character $c \in \Sigma \cup \{\$\}$, let $\text{group}(\text{node}(u), c) := \{w \mid cw \in R(\text{node}(u))\}$ be the ***c*-group** of $R(\text{node}(u))$.

2.a) If for a particular $c \in \Sigma \cup \{\$\}$, the set $\text{group}(\text{node}(u), c)$ contains only one string w, then there is a leaf edge labeled cw outgoing from node(u).

2.b) If $\text{group}(\text{node}(u), c)$ contains at least two strings, then there is an edge labeled cv leading to a branching node node(ucv) where v is the **longest common prefix (lcp)** of all strings in $\text{group}(\text{node}(u), c)$. This includes that v may be equal to ε. The child node(ucv) has the set of remaining suffixes $R(\text{node}(ucv)) = \{w \mid vw \in \text{group}(\text{node}(u), c)\}$.

The WOTD algorithm starts by evaluating the root from the set of all suffixes of $s\$$, setting $R(\text{node}(\varepsilon)) := \{v \mid v \text{ is a suffix of } s\$\}$. All internal nodes are then evaluated recursively in a top-down strategy.

Example 48 Consider the input string $s\$ =$ `abbabbab$`. The WOTD algorithm works as follows.

At first, the root is evaluated from the set of all non-empty suffixes of the string $s\$$, see Figure 10.3. The algorithm recognizes three groups of suffixes: the `$`-group, the `a`-group, and the `b`-group.

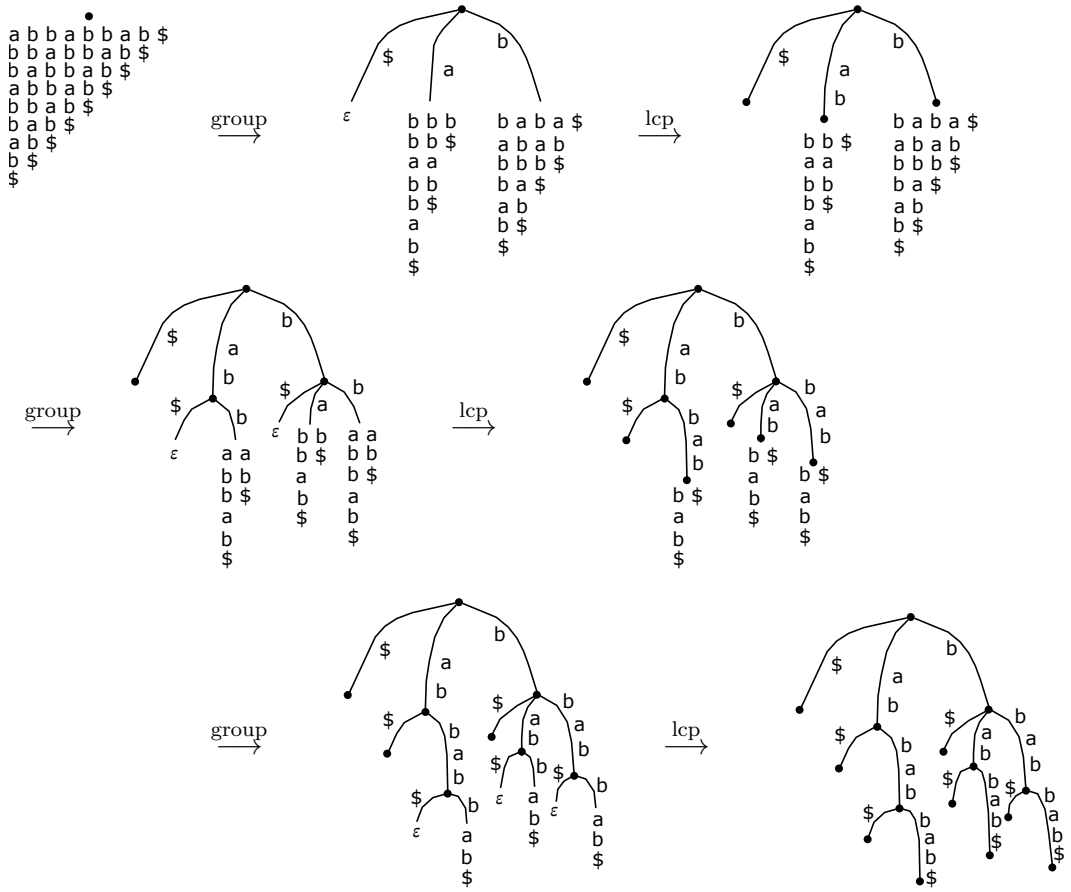


Figure 10.3: The write-only top-down construction of the suffix tree for `abbabbab$`

The `$`-group is singleton, so we obtain a leaf reached by an edge labeled `$`.

The `a`-group contains three suffixes, `bbabbab$`, `bbab$` and `b$`. (Recall that the preceding `a` is *not* part of the suffixes of the group.) We compute the longest common prefix of the strings in this group. This is `b` in our case. So the `a`-edge from the root is labeled by `ab`, and we obtain an unevaluated branching node with remaining suffixes `babbab$`, `bab$` and `$`, which is later evaluated recursively.

The `b`-group of the root contains five suffixes: `babbab$`, `abbab$`, `bab$`, `ab$` and `$`. The outgoing edge is labeled `b`, since there is no common prefix among the strings in the `b`-group, and the resulting branching node node(`b`) has five remaining suffixes `babbab$`, `abbab$`, `bab$`, `ab$` and `$`, which are recursively classified. ◀

Analysis. The worst case running time of WOTD is $O(n^2)$. Consider, for example, the string $s = \mathtt{a}^n$. The suffix tree for $s\$$ is a binary tree with exactly one branching node of depth i for each $i \in [0, n-1]$. To construct the branching node of depth i, exactly $n - i$ suffixes are considered. That is, the number of steps is $\sum_{i=0}^{n-1}(n-i) = \sum_{j=1}^{n} j = \binom{n+1}{2} \in O(n^2)$.

In the average case, the maximal depth of the branching nodes is much smaller than $n-1$, namely $O(\log_\sigma n)$, where $\sigma = |\Sigma|$. In other words, the length of the path to the deepest branching node in the suffix tree is $O(\log_\sigma n)$. The suffixes along the leaf edges are not read any more. Hence the expected running time of the WOTD construction is $O(n \log_\sigma n)$.

WOTD has several properties that make it interesting in practice:

- The subtrees of the suffix tree are constructed independently from each other. Hence the algorithm can easily be parallelized.
- The locality behavior is excellent: Due to the write-only-property, the construction of the subtrees depends only on the set of remaining suffixes. Thus the data required to construct the subtrees is very small. As a consequence, it often fits into the cache. This makes the algorithm fast in practice since a cache access is much faster than the access to the main memory. In many cases, WOTD is faster in practice than worst-case linear time suffix tree construction methods.
- The paths in the suffix tree are constructed in the order they are searched, namely top-down. Thus one can organize the algorithm such that a subtree is constructed only when it is traversed for the first time. This would result in a "lazy construction", which comes for free in a lazy functional programming language (such as Haskell), but could also be implemented in an eager imperative language (such as C). Experiments show that such a lazy construction is very fast in practice in many scenarios.

11 Suffix Tree Applications

11.1 Exact String Matching

Perhaps the most elementary problem in sequence analysis is the following, for which we have already seen a number of solutions:

Problem 49 (Exact String Matching Problem) Given a text $s \in \Sigma^*$ and a pattern $p \in \Sigma^*$, what can we say about occurrences of p in s? To be precise, we have to distinguish between different variants:

1. *decide* whether p occurs at least once in s (i. e., whether p is a substring of s),
2. *count* the number of occurrences of p in s,
3. *list* the starting positions of all occurrences of p in s.

We shall see that, given the suffix tree of $s\$$, the first problem can be decided in $O(|p|)$ time, which is independent of the text length. The second problem can be solved in the same time, using additional annotation in the suffix tree. The time to solve the third problem must obviously depend on the number of occurrences z of p in s. We show in three steps that it can be solved in optimal $O(|p| + z)$ time.

1. Since the suffix tree for $s\$$ contains all substrings of $s\$$, it is easy to verify whether p is a substring of s by following the path from the root directed by the characters of p. If at some point one cannot proceed with the next character in p, then p is not displayed by the suffix tree and hence it is not a substring of s. Otherwise, if p occurs in the suffix tree, then it is also a substring of s. Processing each character of p takes constant time, either by verifying that the next character of an edge label agrees with the next character of p, or by finding the appropriate outgoing edge of a branching node. The latter case assumes a constant alphabet size, i.e., $|\Sigma| = O(1)$. Therefore the total time is $O(|p|)$.

2. To count the number of occurrences, we could proceed as follows after solving the first problem. If p occurs at least once in s, we will have found a position in the tree (either in the middle of an edge or a node) that represents p. Now we only need to count the number of leaves below that position. However, this would take time proportional to the number of leaves. A better way is to preprocess the tree once in a bottom-up fashion and annotate each node with the number of leaves below. Then the answer can be found in the node immediately below or at p's position in the tree.

3. We first find the position in the tree that corresponds to p in $O(|p|)$ time according to step 1. Assuming that each leaf is annotated with the starting position of its associated suffix, we visit each of the z leaves below p and output its suffix starting position in $O(z)$ time.

Example 50 Let $s =$ `abbab`. The corresponding suffix tree of `abbab$` is shown in Figure 11.1.

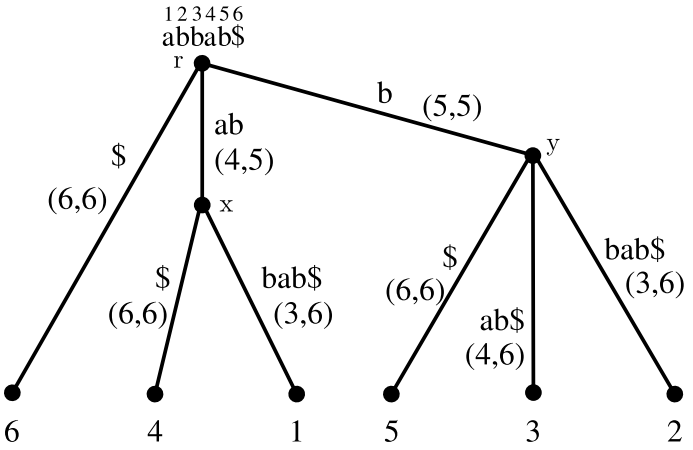


Figure 11.1: The suffix tree of `abbab$` with edge labels, substring pointers and leaf labels.

Suppose $p =$ `aba` is the pattern. Reading its first character `a`, we follow the `a`-edge from the root. Since the edge has length 2, we verify that the next character `b` agrees with the pattern. This is the case. We arrive at the branching node node(`ab`). Trying to continue, we see that there is no `a`-edge outgoing from node(`ab`), and we cannot proceed matching p against the suffix tree. In other words, p is not displayed by the tree, hence it is not a substring of s.

Now suppose $p =$ `b`. We follow the `b`-edge from the root, which brings us to the branching node node(`b`). Thus `b` is a substring of s. The leaf numbers in the subtree below node(`b`) are 2, 3 and 5. Indeed, `b` starts in $s\$ =$ `abbab$` at positions 2, 3 and 5. ◀

Note that the above analysis assumes that the alphabet size σ is a constant. If this is not the case, i.e. the alphabet size is considered an input variable, then the time it takes to find the correct outgoing edge at a branching vertex has also to be considered. Different data structures are possible, corresponding to different points in a typical space-time tradeoff: (1) A linked list of all outgoing edges requires space proportional to the degree of a vertex, but in the worst case also $O(\sigma)$ time. (2) A binary search tree takes the same space, but only $O(\log \sigma)$ time in the worst case. (3) An array with constant-time access to each outgoing edge, on the other hand, requires $\Theta(\sigma)$ space.

Longest matching prefix. One variation of this exact pattern matching algorithm is to search for the longest prefix p' of p that is a substring of s. This can clearly be done in $O(|p'|)$ time. This operation is useful in a number of more advanced string comparison models such as maximal matches distance or matching statistics.

11.2 Minimum Unique Substrings

Pattern discovery problems, in contrast to pattern matching problems, deal with the analysis of just one string, in which interesting regions are to be discovered. An example is given in the following.

Problem 51 (Minimum Unique Substrings Problem) Given a string $s \in \Sigma^*$, find all unique substrings of s that have minimum length.

Extensions of this problem have applications in DNA primer design, for example.

Example 52 Let $s =$ `ABABB`. Then the minimum unique substrings are `BA` and `BB`. ◀

We exploit two properties of the suffix tree of s\$.

- If a string w occurs at least twice in s, there are at least two suffixes in s\$, of which w is a proper prefix. Hence in the suffix tree of s\$, w corresponds to a path ending with an edge to a branching node.

- If a string w occurs only once in s, there is only one suffix in s\$ of which w is a prefix. Hence in the suffix tree of s\$, w corresponds to a path ending within a leaf-edge.

According to the second property, we can find the unique strings by looking at the paths ending on the edges to a leaf. So if we have reached a branching node, say node(w), then we only have to look at the leaf edges outgoing from node(w). Consider an edge node(w) $\to v$, where v is a leaf, and assume that au is the edge label with first character $a \in \Sigma$. Then wa occurs only once in s, but w occurs at least twice, since it is a branching node. Therefore wa is a candidate for being a minimum unique substring, although it is not necessarily of minimum length. It is an easy exercise, though, to keep only the shortest of these candidates. Note that the sentinel \$ is a trivial minimum unique substring by definition and therefore usually excluded.

Example 52 (cont'd) The suffix tree of s\$ = `ABABB` is shown in Figure 11.2. It has four leaf edges (ignoring the edge from the root labeled `$`), and therefore there are four candidates for minimum unique substrings, spelling from the root until the first character of any of these edges: `ABA`, `ABB`, `BA` and `BB`. The latter two are of minimum length and will therefore be reported as minimum unique substrings. ◀

The running time of this simple algorithm is linear in the number of nodes and edges in the suffix tree, since we have to visit each of these only once, and for each we do a constant amount of work. The overall algorithm thus runs in optimal linear time since the suffix tree can be constructed in linear time, there is a linear number of nodes and edges in the suffix tree and the simple length-filter at the end also takes linear time.

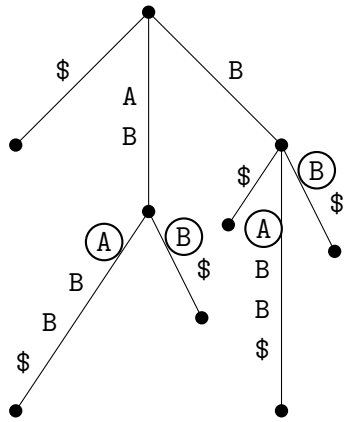


Figure 11.2: Suffix tree of $s\$ = \texttt{ABABB\$}$ illustrating the relationship between the suffix tree of $s\$$ and endpoints of candidates for minimum unique substrings of s.

11.3 Maximal Repeat-Pairs

Informally, a *repeat* in a string is a substring that occurs at least twice. However, care needs to be taken when formalizing the notion of repeat.

Definition 53 Given a string $s \in \Sigma^*$, a **repeat-pair** in s is a pair of substrings (i, j) and (i', j') such that $i < i'$ and $s[i, \ldots, j] = s[i', \ldots, j']$. Its **length** is $\ell = j - i + 1 = j' - i' + 1$.

The substring (i, j) is called the **left instance** of the repeat-pair, and (i', j') is called the **right instance** of the repeat-pair; see Figure 11.3. Note that the two instances may overlap.

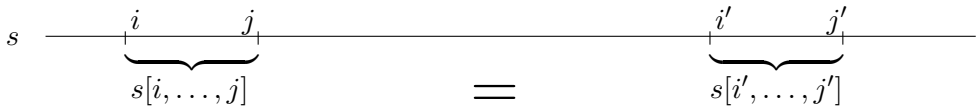


Figure 11.3: Illustration of a repeat-pair $((i, j), (i', j'))$ in sequence s.

Example 54 The string $s = \texttt{agagctcgagc}$, $|s| = 10$ contains the following repeat-pairs of length $\geq$ 2:

$((2,5),(8,11))$	gagc
$((2,4),(8,10))$	gag
$((2,3),(8,9))$	ga
$((3,5),(9,11))$	agc
$((1,2)(3,4))$ $((1,2)(10,11))$ $((3,4),(10,11))$	ag
$((4,5),(10,11))$	gc

◀

We see that shorter repeat-pairs are often contained in longer repeat-pairs. To remove redundancy, we introduce maximal repeat-pairs, illustrated in Figure 11.4. Essentially, maximality means that the repeated substring cannot be extended to the left or to the right.

Definition 55 A repeat-pair $((i, j), (i', j'))$ in a string s is **left-maximal** if and only if $i = 1$ or $s[i-1] \neq s[i'-1]$. It is **right-maximal** if and only if $j' = |s|$ or $s[j+1] \neq s[j'+1]$. A repeat-pair is **maximal** if it is both left-maximal and right-maximal.

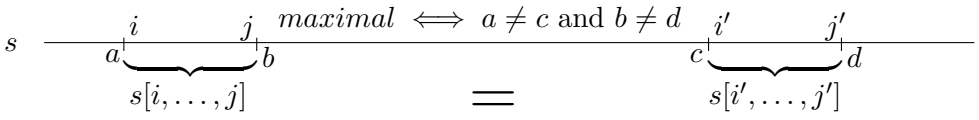


Figure 11.4: Illustration of maximality of repeat-pairs

From now on we restrict ourselves to maximal repeat-pairs. All non-maximal repeat-pairs can easily be obtained from the maximal repeat-pairs. In Example 54, most repeat-pairs can be extended to the left or to the right. Only the three repeat-pairs $((2, 5), (8, 11))$, $((1, 2), (3, 4))$ and $((1, 2), (10, 11))$ are maximal.

Problem 56 (Maximal Repeat-Pair Discovery Problem) Given a string $s \in \Sigma^*$, find all maximal repeat-pairs of s (possibly of a given minimal length ℓ).

An optimal algorithm. We shall present a linear-time algorithm to compute all maximal repeat-pairs. It works in two phases: In the first phase, the leaves of the suffix tree are annotated. In the second phase, the maximal repeat-pairs are reported while the branching nodes are annotated simultaneously in a bottom-up procedure.

In detail, suppose we have the suffix tree for some string $s\$$. We ignore leaf edges from the root, since the root corresponds to repeat-pairs of length zero and we are not interested in these. Figure 11.5 gives an example for the string $s = \texttt{ggcgctgcgcc}$.

In the **first phase**, the algorithm annotates each leaf of the suffix tree: leaf v with path-label string$(v) = s[i \ldots n]$ is annotated by the pair $(a; i)$, where i is the position at which the suffix ending at v starts and $a = s[i-1]$ is the character to the immediate left of that position. We also write $A(v, s[i-1]) = \{i\}$ to denote the annotation, and assume $A(v, c) = \emptyset$ for all characters $c \in \Sigma$ different from $s[i-1]$. The latter assumption holds in general (also for branching nodes) whenever there is no annotation for some character c. For the suffix tree of Figure 11.5, the leaf annotation is shown in Figure 11.6.

The leaf annotation gives us the character upon which we decide the left-maximality of a repeat-pair, plus a position where a repeated string occurs. We only have to combine this information at the branching nodes appropriately.

This is done in the **second phase** of the algorithm: In a bottom-up traversal, the repeat-pairs are reported and simultaneously the annotation for the branching nodes is computed.

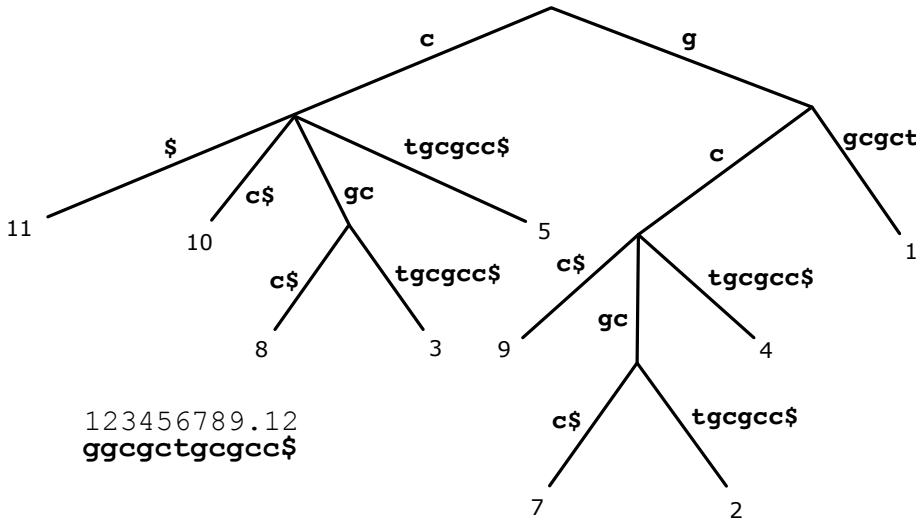


Figure 11.5: The suffix tree for `ggcgctgcgcc$`. Leaf edges from the root are not shown. These edges are not important for the algorithm.

A bottom-up traversal means that a branching node is visited only after all of its children have been visited.

Consider a vertex v. The first child copies its lists A to v. Then the other children of v are considered one after the other. Thereby a child, say w, is processed as follows:

1. Repeat-pairs (for the string ending at node v) are reported by combining the annotation already computed for node v with the complete annotation stored for w (this was already computed due to the bottom-up strategy). In particular, we consider all pairs $((i, i+q-1), (j, j+q-1))$, where

   - q is the string-depth of node v, i.e. $q = |\text{string}(v)|$,
   - $i \in A(v, \sigma)$ and $j \in A(w, \sigma')$ for some characters $\sigma \neq \sigma'$, where $A(v, \sigma)$ is the annotation already computed for v w.r.t. character σ and $A(w, \sigma')$ is the annotation stored for node w w.r.t. character σ'.

   Note that only those pairs are considered which have different characters to the left. Thus it guarantees **left-maximality** of the repeat-pairs.

2. Recall that we consider processing the edge $v \to w$ and let a be the first character of the label of this edge. The annotation already computed for v was inherited along edges outgoing from v, that are different from $v \to w$. Thus the first character of the label of such an edge, say b, is different from a. Now since string(v) is the repeated substring, b and a are characters to the right of string(v). As a consequence, only those positions are combined which have different characters to the right. In other words, the algorithm also guarantees **right-maximality** of the repeat-pairs.

As soon as for the current edge the repeat-pairs are reported, the algorithm computes the union $A(v, c) \cup A(w, c)$ for each character c, i.e. the annotation is inherited from node w to node v. In this way, after processing all edges outgoing from v, this node is annotated by

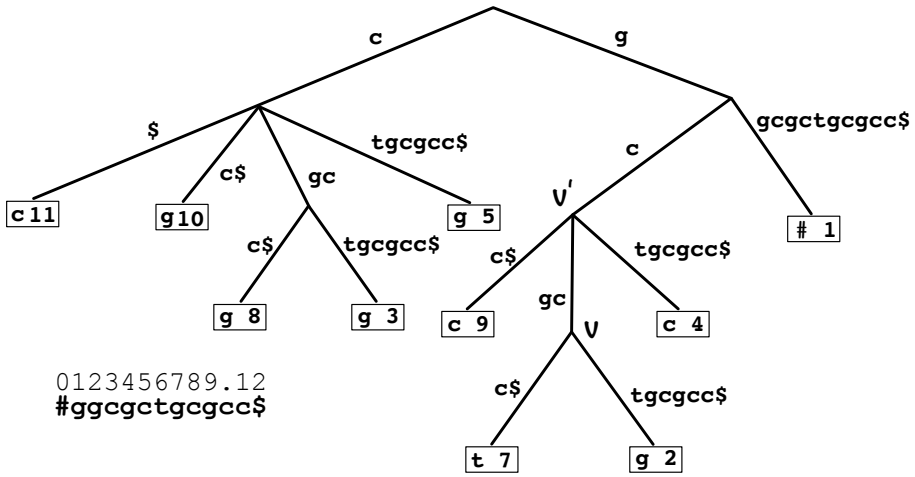


Figure 11.6: The suffix tree for ggcgctgcgcc$ with leaf annotation. For convenience, a second sentinel character # has been added to the left at index position 0.

the set of positions where string(v) occurs, and this set is divided into (possibly empty) disjoint subsets $A(v, c_1), \ldots, A(v, c_\sigma)$, where $\Sigma = \{c_1, \ldots, c_\sigma\}$.

Example 57 The suffix tree of the string $s\$ =$ ggcgctgcgcc$ is shown in Figure 11.5. We assume the leaf annotation of it (as shown in Figure 11.6) is already determined.

Proceeding from leaves 7 and 2, the bottom up traversal begins with node v whose path-label is gcgc, of string-depth $q = 4$. This node has two children. First, the leaf-list (t; 7) of the first child is copied to v. Then the list is compared to any leaf-list of all remaining children (here only one) that is not a t-list. Here this is only the list (g; 2). This pair is automatically left-maximal and therefore reported: $((2, 2 + 4 - 1), (7, 7 + 4 - 1)) = ((2, 5), (7, 10))$. Then the leaf-list of the second child, g, 2) is merged with the leaf-lists already present at noce v, as can be seen in Figure 11.7.

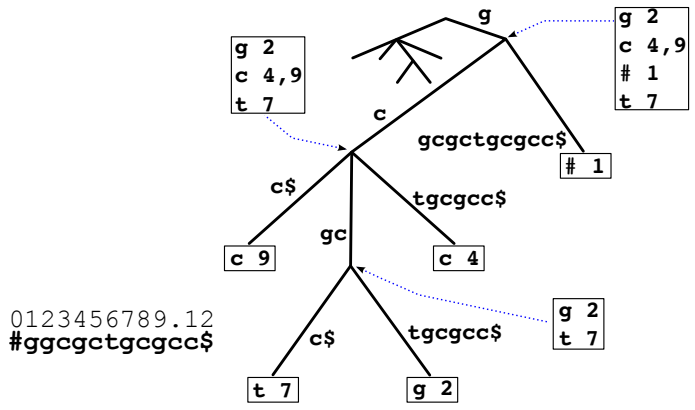


Figure 11.7: The annotation for a large part of the suffix tree of Figure 11.6 and some repeats.

Next comes the node v' with path-label gc of depth two. The algorithm starts by copying the leaf-list (c; 9) from the leftmost child of v'. Then the second child is processed, which

is the node v that has annotation $(\mathtt{g}, 2)$, $(\mathtt{t}, 7)$. Since both left-characters are different from $\mathtt{c}$, the repeat-pairs $((7,8),(9,10))$ and $((2,3),(9,10))$ are reported. Then the leaf-lists are merged (the new annotation for v' becomes $(\mathtt{c}, 9)$, $(\mathtt{t}, 7)$, $(\mathtt{g}, 2)$) and the procedure is continued with the third child, that has leaf-list $(\mathtt{c}, 4)$. The annotations $(\mathtt{c}, 9)$ and $(\mathtt{c}, 4)$ can not be combined because they have the same left-character. So only the repeat-pairs $((4,5),(7,8))$ and $((2,3),(4,5))$ are reported, resulting from the combination of $(\mathtt{t}; 7)$ and $(\mathtt{g}; 2)$ with $(\mathtt{c}; 4)$. The final annotation of v' is $(\mathtt{g}, 2)$, $(\mathtt{c}; 4, 9)$, $(\mathtt{t}; 7)$, which can also be written as $A(v', \mathtt{g}) = \{2\}$, $A(v', \mathtt{c}) = \{4, 9\}$ and $A(v', \mathtt{t}) = \{7\}$. We leave further processing up to the reader. ◀

Running Time. Let us now consider the running time of the algorithm. Traversing the suffix tree bottom-up can surely be done in time linear in the number of nodes, since each node is visited only once and we only have to follow the paths in the suffix tree. There are two operations performed during the traversal: Output of repeat-pairs and combination of annotations. If the annotation for each node is stored in linked lists, then the output operation can be implemented such that it runs in time linear in the number of repeat-pairs. Combining the annotations only involves linking lists together, and this can be done in time linear in the number of nodes visited during the traversal. Recall that the suffix tree can be constructed in $O(n)$ time. Hence the algorithm requires $O(n + z)$ time where n is the length of the input string and z is the number of maximal repeat-pairs. (Analysis shows that the number of maximal repeat-pairs of a string of length n is $z \in O(n^2)$ in the worst case.)

To analyze the space consumption of the algorithm, first note that we do not have to store the annotations for all nodes at once. As soon as a node and its parent has been processed, we no longer need the annotation. As a consequence, the annotation requires only $O(n)$ overall space. Hence the space consumption of the algorithm is $O(n)$.

Altogether the algorithm is optimal since its space and time requirements are linear in the size of the input plus the size of the output.

11.4 Maximal Unique Matches

The standard dynamic programming algorithm to compute an optimal alignment between two sequences of lengths m and n requires $O(mn)$ time. This is too slow if the sequences are on the order of millions or even billions of characters.

There are other methods which allow to align two genomes under the assumption that these are fairly similar. The basic idea is that the similarity often results in long identical substrings which occur in both genomes. These identities, called MUMs (for *maximal unique matches*) are almost surely part of any good alignment of the two genomes. So the first step is to find the MUMs. These are then taken as the fixed parts of an alignment, and the remaining parts of the genomes (those parts not included in a MUM) are aligned with traditional dynamic programming methods. In this section, we will show how to compute the MUMs in linear time. This is very important for the practical applicability of the method. We do not consider how to compute the final alignment. (The whole

procedure of genome alignment is discussed in Chapter 15 of these notes.) We first have to define the notion MUM precisely:

Definition 58 Given strings $s, t \in \Sigma^*$ and a minimal length $\ell \geq 1$, a MUM is a triple (i, j, L) representing a string $u = s[i, i + L - 1] = t[j, j + L - 1]$ of length L that satisfies the following conditions:

1. $L \geq \ell$ (length restriction),
2. u occurs exactly once in s and exactly once in t (uniqueness),
3. for any character a, neither au nor ua occur in both s and t (left- and right-maximality).

Problem 59 (Maximal Unique Matches Problem) Given $s, t \in \Sigma^*$ and a minimal length $\ell \geq 1$, find all MUMs of s and t.

Example 60 Let $s =$ `ccttcgt`, $t =$ `ctgtcgt`, and $\ell = 2$. Then there are two maximal unique matches, $(2, 1, 2)$ representing `ct` and $(4, 4, 4)$ representing `tcgt`. Now consider an optimal alignment of these two sequences (assuming unit costs for insertions, deletions, and replacements):

```
cct-tcgt
-ctgtcgt
```

The two MUMs `ct` and `tcgt` are part of this alignment. ◀

To compute the MUMs, we first construct the generalized suffix tree for s and t, i.e. the suffix tree of the concatenated string $x := s\#t\$$.

A MUM (i, j, L) representing u must occur exactly twice in x, once in s and once in t. Hence u corresponds to a path in the suffix tree ending with an edge to a branching node. Since a MUM must be right-maximal, u must even end in that branching node, and that node must have exactly two leaves as children, one in s and one in t. It remains to check the left-maximality in each case. We thus arrive at the following algorithm:

For each branching node v of the suffix tree of x,

1. check that its string-depth is at least ℓ (length restriction),
2. check that there are exactly two children, both of which are leaves (uniqueness, right-maximality),
3. check that the *suffix starting positions* i and j at those leaves correspond to positions one from s and one from t in x (match),
4. check that the characters $x[i - 1]$ and $x[j - 1]$ are different, or $i = 0$ or $j = 0$ (left-maximality).

If all checks are true, output string$v$ and/or its positions i and j.

Clearly, the algorithm runs in linear time $O(|x|)$ since each step (1. – 4.) can be organized to run in constant time, and there are a linear number of branching nodes in the suffix tree of x.

Example 60 (cont'd) Let $s =$ `ccttcgt`, $t =$ `ctgtcgt` and $\ell = 2$. Consider the suffix tree for s`#`t`$` shown in Figure 11.8.

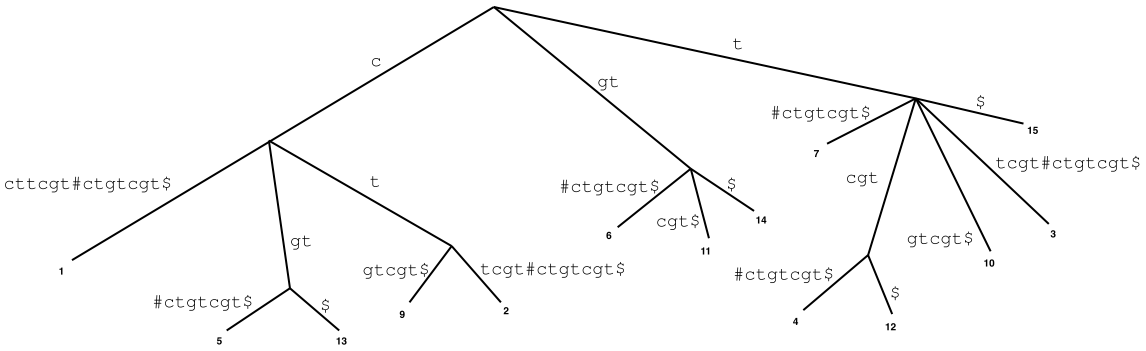


Figure 11.8: The suffix tree for `ccttcgt#ctgtcgt$` without the leaf edges from the root

The string `tcgt` occurs once in s (at index 4) and once in t (at index 4), since there are two corresponding leaf edges from branching node node(`tcgt`). Comparing the characters $s[3] =$ `t` and $t[3] =$ `g` immediately to the left of the occurrences of `tcgt` in s and t verifies left-maximality. Similarly for `ct`. Thus we have found the two MUMs $(4, 4, 4)$ and $(2, 1, 2)$. On the other hand, `cgt` (occurring at index 5 in s and at index 5 in t) is not left-maximal and therefore no MUM, because both occurrences have the left neighbor $s[4] = t[4] =$ `t`. ◀

12 Suffix Arrays

12.1 Motivation

We have already seen that suffix trees are a very useful data structure for a variety of string matching problems. In the early 1990s, it was believed that storing a suffix tree needs around 30–40 bytes per character. Given the smaller amounts of available memory at that time, this led to the invention of a "flat" data structure that is even more memory efficient but nevertheless captures the essence of the suffix tree: the **suffix array**.

Moreover, a suffix array can be complemented with additional information, called an **enhanced suffix array** or **extended suffix array**, and is then able to completely replace (because it is equivalent to) the suffix tree. Additionally, some problems have simpler algorithms on suffix arrays than on suffix trees (for other problems, the opposite is true).

A suffix array is easy to define: Imagine the suffix tree, and assume that at each internal node the edges toward the children are alphabetically ordered. If furthermore each leaf is annotated by the starting position of its corresponding suffix, we obtain the suffix array by reading the leaf labels from left to right. In fact, the suffix array is the array of the (starting positions of the) suffixes in ascending lexicographic order.

12.2 Basic Definitions

In this section, we start counting string positions at zero. Thus a string s of length n is written as $s = (s[0], \ldots, s[n-1])$. As before, we append a sentinel \$ to each string. In examples, we shall always assume a natural order on the alphabet Σ and define \$ to be lexicographically smaller than any character of Σ, i. e., $\$ < a < b < c < \ldots$.

Definition 61 For a string $s \in \Sigma^n$, the **suffix array** `pos` of $s\$$ is a permutation of the integers $\{0, \ldots, n\}$ such that `pos`$[r]$ is the starting position of the lexicographically r-th smallest suffix of $s\$$.

The **inverse suffix array** `rank` of $s\$$ is a permutation of the integers $\{0, \ldots, n\}$ such that `rank`$[p]$ is the lexicographic rank of the suffix starting at position p.

Clearly by definition `rank`[`pos`$[r]$] $= r$ for all $r \in \{0, \ldots, n\}$, and also `pos`[`rank`$[p]$] $= p$ for all $p \in \{0, \ldots, n\}$. Since we assume that \$ is the smallest character and occurs only at position n, we have `rank`$[n] = 0$ and `pos`$[0] = n$.

The suffix array by itself represents the order of the leaves of the suffix tree, but it does not contain information about the internal nodes. Recall that the string depth of an

internal node corresponds to the length of the longest common prefix of all suffixes ending at leaves below that node. We can therefore recover information about the internal nodes by making the following definition.

Definition 62 Given a string $s \in \Sigma^n$ and the suffix array `pos` of $t = s\$$, we define the **longest common prefix array** $\texttt{lcp} : \{1, \ldots, n\} \to \mathbb{N}_0$ by

$$\texttt{lcp}[r] := \max\{|x| \mid x \text{ is a prefix of both } t[\texttt{pos}[r-1]...n] \text{ and } t[\texttt{pos}[r]...n]\}.$$

In other words, $\texttt{lcp}[r]$ is the length of the longest common prefix of the suffixes starting at positions $\texttt{pos}[r-1]$ and $\texttt{pos}[r]$.

By convention, we additionally define $\texttt{lcp}[0] := -1$ and $\texttt{lcp}[n+1] := -1$; this avoids treating boundary cases specially.

Example 63 Table 12.1 shows the suffix array `pos` and the `lcp` array of the string `abbab$`. The suffix tree was given in Figure 11.1. Since there we started counting string positions at 1, the suffix array is obtained by subtracting 1 from the leaf labels, and reading them from left to right: $\texttt{pos} = (5, 3, 0, 4, 2, 1)$. ◀

r	0	1	2	3	4	5
suffix	$	ab$	abbab$	b$	bab$	bbab$
pos[r]	5	3	0	4	2	1

r	0	1	2	3	4	5	6
lcp[r]	- 1	0	2	0	1	1	- 1

Table 12.1: Suffix array `pos` of string $s\$ =$ `abbab$` and its longest common prefix array `lcp`. The inverse `rank` is clearly $(2, 5, 4, 1, 3, 0)$.

Intervals in the suffix array and lcp intervals. There is a one-to-one correspondence between internal nodes in the suffix tree and certain intervals in the suffix array. Indeed, we have already seen that all leaves below an internal node v in the suffix tree correspond to a particular interval of at least two elements in the suffix array. Here we denote this interval by $\texttt{pos}[j \ldots k]$, for positive integers $j < k$. If v has string depth d, it follows that:

- $\texttt{lcp}[i] < d$, since the suffix starting at $\texttt{pos}[i-1]$ is not below v and hence does not share a common prefix of length d with the suffix starting at pos$[i]$;
- $\texttt{lcp}[r] \geq d$ for all r with $i < r \leq j$, since all suffixes below v share a common prefix of at least d. On the other hand, at least for one r in this range, we have $\texttt{lcp}[r] = d$. Otherwise, if all $\texttt{lcp}[r] > d$ in this range, v would have a string depth larger than d, contradicting the original assumption.
- $\texttt{lcp}[j+1] < d$, since again the suffix starting at $\texttt{pos}[j+1]$ is not below v.

Conversely, each pair $([j, k], d)$ satisfying $j < k$ and the above conditions corresponds to an internal node in the suffix tree. Such an interval $[i, j]$ is called a d-interval, or generally, an **lcp interval**.

12.3 Suffix Array Construction Algorithms

12.3.1 Linear-Time Construction using a Suffix Tree

Given the suffix tree of $s\$$, it is straightforward to construct the suffix array of $s\$$ in linear time. We start with an empty list `pos` and then do a depth-first traversal of the suffix tree, visiting the children of each internal node in alphabetical order. Whenever we encounter a leaf, we append its annotation (the starting position of the associated suffix) to `pos`. This takes linear time, since we traverse each edge once down and later up again, and there is a linear number of edges.

If the suffix tree data structure stores the internal nodes in depth-first order, the suffix array is particularly simple to construct. One must merely walk through the memory locations from left to right, keep track of the current string depth, and collect all leaf pointers along the way. Leaf pointers can be recognized e.g. by a special flag. The leaf number is obtained by subtracting the current string depth from the leaf pointer.

This construction is simple and fast, but first requires the suffix tree. One of the major motivations of suffix arrays was to avoid constructing the tree in the first place. Therefore direct construction methods are more important in practice.

12.3.2 Direct Construction

The simplest direct construction method is to use any comparison-based sorting algorithm (e.g. Mergesort or Quicksort) and apply it to the suffixes of $s\$$. We let $n := |s|$. For the analysis we need to consider that a comparison of two suffixes does not take constant time, but $O(n)$ time, since up to n characters must be compared to decide which suffix is lexicographically smaller. Optimal comparison-based sorting algorithms need $O(n \log n)$ comparisons, so this approach constructs the suffix array in $O(n^2 \log n)$ time.

Especially Quicksort is an attractive choice for the basic sorting method, since it is fast in practice (but needs $O(n^2)$ comparisons in the worst case, so this would lead to an $O(n^3)$ worst-case suffix array construction algorithm), and sorting the suffix permutation can be done *in place*, so no additional memory besides the text and `pos` is required.

As already mentioned in the analysis of the WOTD algorithm (Section 10.5), for an average random text, two suffixes differ after $\log_\sigma n$ characters, so each comparison needs only $O(\log n)$ time on average. Combined with the average-case running time of QuickSort (or worst-case running time of MergeSort), we get an $O(n \log^2 n)$ average-case suffix array construction algorithm that is easy to implement and performs very well in practice on strings that do not contain long repeats.

Manber-Myers Algorithm. Manber and Myers (1993), who introduced the suffix array data structure, proposed an algorithm that runs in $O(n \log n)$ worst-case time on any string of length n. It uses an ingenious doubling technique due to Karp, Miller and Rosenberg (Karp et al., 1972).

The algorithm starts with an initial character-sorting phase (called phase $k = 0$) and then proceeds in up to $K := \lceil \log_2 n \rceil$ phases, numbered from $k = 1$ to $k = K$. The algorithm maintains the invariant that after phase $k \in \{0, \ldots, K\}$, all suffixes have been correctly sorted according to their first 2^k characters. Thus, in phase 0 it is indeed sufficient to group the suffixes according to their first character into σ buckets, one for each letter of the alphabet Σ.

Each of the following phases $k = 1, \ldots, K$ must then double the number of accounted characters and update the suffix order accordingly. This is achieved by refining the buckets of suffixes with equal first 2^{k-1} characters from the previous phase, using the order of their continuations starting after the common part, i.e. with offset 2^{k-1}. Interestingly, this order is given by the relative suffix order from the previous phase, resulting in a refinement by another 2^{k-1} characters in a single step. Therefore, after phase k, the suffixes are sorted according to their first $2^{k-1} + 2^{k-1} = 2^k$ characters.

After phase K, all suffixes are correctly sorted with respect to their first $2^K \geq n$ characters and therefore in correct lexicographic order. Clearly, the initial sorting in phase $k = 0$ is possible in linear time $O(n)$. In order to see that each doubling phase runs in $O(n)$ time as well, note that essentially a bucket sort is performed inside each group, using the bucket position from the previous phase as key. This takes in total $O(n)$ time for all buckets of one phase. Therefore, and considering that there are $O(\log n)$ phases, the $O(n \log n)$ running time of the overall algorithm is established.

The following example illustrates the algorithm.

Example 64 Given string $s\$ =$ `MAMMAMIAMMAMIA$`, the following arrays show the intermediate steps of the Manber-Myers algorithm.

```
MAMMAMIAMMAMIA$
0123456789.1234
```

σ	#	bucket start
$	1	0
A	5	1
I	2	6
M	7	8

after phase 0

$	A	I	M
14	1, 4, 7, 10, 13	6, 12	0, 2, 3, 5, 8, 9, 11

after phase 1

$	A$	AM	IA	MA	MI	MM
14	13	1, 4, 7, 10	6, 12	0, 3, 9	5, 11	2, 8

after phase 2

$	A$	AMIA	AMMA	IA$	IAMM	MAMI	MAMM	MIA$	MIAM	MMAM
14	13	4, 10	1, 7	12	6	3, 9	0	11	5	2, 8

after phase 3

$	A$	AMIA $	AMIA MMAM	AMMA MIA$	AMMA MIAM	IA$	IAMM AMIA	MAMI A$	MAMI AMMA	MAMM AMIA	MIA$	MIAM MAMI	MMAM IA$	MMAM IAMM
14	13	10	4	7	1	12	6	9	3	0	11	5	8	2

The algorithm takes at most $K = \lceil \log_2 15 \rceil = 4$ phases. More precisely, in this example it terminates already after only 3 phases. ◀

Skew Algorithm. There also exists a more advanced algorithm, called Skew (Kärkkäinen and Sanders, 2003) that directly constructs suffix arrays in $O(n)$ time. However, we will not give details here.

12.3.3 Construction of the `rank` and `lcp` Arrays

We show how to construct the `rank` and `lcp` arrays in linear time when the suffix array `pos` is already available.

The inverse suffix array `rank` can be easily computed from `pos` in linear time by the following one-liner, using the fact that `rank` and `pos` are inverse permutations of each other.

```
for(int r=0; r<=n; r++) rank[pos[r]]=r;
```

Computing `lcp` in linear time is a little bit more difficult. The naive algorithm, comparing the prefixes of all adjacent suffix pairs in `pos` until we find the first mismatch, can be described as follows:

```
lcp[0] = lcp[n+1] = -1;    // by convention
for(int r=1; r<=n; r++) {
  lcp[r] = LongestCommonPrefixLength(pos[r-1],pos[r]);
}
```

Here we assume that `LongestCommonPrefixLength(`p_1`,`p_2`)` is a function that compares the suffixes starting at positions p_1 and p_2 character by character until it finds the first mismatch, and returns the prefix length. Clearly the overall time complexity is $O(n^2)$.

The following algorithm, due to Kasai et al. (2001) achieves $O(n)$ time by comparing the suffix pairs in a different order. On a high level, it can be written as follows:

```
lcp[0] = lcp[n+1] = -1;     // by convention
for(int p=0; p<n; p++) {
  lcp[rank[p]] = LongestCommonPrefixLength(pos[rank[p]-1],p);
}
```

At first sight, we have gained nothing. Implemented in this way, the time complexity is still $O(n^2)$. The only difference is that the `lcp` array is not filled from left to right, but in apparently random order, depending on `rank`.

The key observation is that we do not need to evaluate `LongestCommonPrefixLength` from scratch for every position p. First of all, let us simplify the notation. Define `left`$[p] :=$ `pos`$[$`rank`$[p]-1]$; this is the number immediately left of the number p in the suffix array. Then the algorithm from above looks as follows:

```
lcp[0] = lcp[n+1] = -1;     // by convention
for(int p=0; p<n; p++) {
  lcp[rank[p]] = LongestCommonPrefixLength(left[p],p);
}
```

In the first iteration of the loop, we have $p = 0$ and compute the lcp-value at `rank`$[p]$, i.e., the length L of the longest common prefix of the suffixes starting at position p and at position `left`$[p]$ (which can be anywhere in the string).

In the next iteration, we look for the longest common prefix length of the suffixes starting at positions $p+1$ and at `left`$[p+1]$. If `left`$[p+1] =$ `left`$[p]+1$, we already know that the answer is $L-1$, one less than the previously computed value, since we are looking at the same string with the first character chopped off. If `left`$[p+1] \neq$ `left`$[p]+1$, we know at least that the current lcp value *cannot be smaller than* $L-1$, since (assuming $L > 0$ in the first place) the number `left`$[p]+1$ must still appear somewhere to the left of $p+1$ in the suffix array, but might not be directly adjacent. Still, the suffixes starting at `left`$[p]+1$ and $p+1$ share a prefix of length $L-1$. Everything in between must share a common prefix that is at least that long. Thus we do not need to check the first $L-1$ characters, we know that they are equal, and can immediately start comparing the L-th character.

The same idea applies to all p-iterations. We summarize the key idea in a lemma, which we have just proven by the above argument.

Lemma 65 Let $L :=$ `lcp`[`rank`$[p]$]. Then `lcp`[`rank`$[p+1]$] $\geq L-1$.

The algorithm then looks as follows.

```
lcp[0] = lcp[n+1] = -1;     // by convention
L = 0;
for(int p=0; p<n; p++) {
  L = lcp[rank[p]] = LongestCommonPrefixExtension(left[p],p,L);
  if (L>0) L--;
}
```

Here `LongestCommonPrefixExtension`(p_1, p_2, L) does essentially the same work as the function `LongestCommonPrefixLength` above, except that it starts comparing the suffixes at $p_1 + L$ and $p_2 + L$, effectively skipping the first L characters of the suffixes starting at p_1 and p_2, as they are known to be equal.

It remains to be proven that these savings lead to a linear-time algorithm. This is done by a technique called *amortized analysis*. We focus on the path of values that variable L takes. First, note that the maximal value that L can take at any time, including upon termination, is n. Initially L is zero. L is decreased at most n times. Thus in total, L can increase by at most $2n$ across the whole algorithm. Each increase is due to a successful character comparison. Each call of `LongestCommonPrefixExtension`, of which there are n, ends with a failed character comparison. Thus at most $3n = O(n)$ character comparisons are made during the course of the algorithm. Thus we have proven the following theorem.

Theorem 66 Given the string $s\$$ and its suffix array `pos`, the `lcp` array can be computed in linear time.

12.4 Applications of Suffix Arrays

Often, applications from suffix trees can be adapted to the suffix array data structure with little or no overhead. These include:

- exact string matching in $O(|p| \log n)$ time (or in $O(|p| + \log n)$ time with the `lcp` array)
- matching statistics
- Burrows-Wheeler transformation (Burrows and Wheeler, 1994), see Chapter 13
- FM index (Ferragina and Manzini, 2005), see Section 14.1
- ... and many more, see e.g. (Abouelhoda et al., 2002)

13 Burrows-Wheeler Transformation

13.1 Introduction

The Burrows-Wheeler transformation (BWT) is a technique to transform a text into a permutation of this text that is easy to compress and search. The central idea is to sort the cyclic rotations of the text and gaining an output where equal characters are grouped together. These grouped characters then are a favorable input for run-length encoding, where a sequence of numbers and characters is constructed, considerably reducing the length of the text (see Section 13.4.1).

Due to the fact that in sequence analysis mostly large data sets are processed, it is favorable to have a technique compressing the data to reduce memory requirement and at the same time enabling important algorithms to be executed on the converted data. The BWT provides a *transformation* of the text fulfilling both requirements in a useful and elegant way. Further, the transformation is bijective, so it is guaranteed that the original text can be reconstructed in an uncompression step, called *retransformation*.

13.2 Transformation and Retransformation

A simple definition of the Burrows-Wheeler transformation is the following:

Definition 67 Let $s \in \Sigma^*$ be an input string of length $n := |s|$ and $t = s\$$. Further, let M be the $(n+1) \times (n+1)$-matrix containing in its rows the lexicographically sorted cyclic rotations of t. The **Burrows-Wheeler transformation** $\text{bwt}(t)$ is the last column of M,

$$\text{bwt}(t)[i] := M(i, n) \text{ for all } i,\ 0 \leq i \leq n.$$

Note that the construction method implied by this definition is very inefficient. As described, it requires $O(n^2)$ space, and even if the lexicographic sorting is performed in an efficient way, it still takes at least quadratic time to generate the matrix M. A more efficient possibility to compute $\text{bwt}(t)$ is to construct the column L directly. This can be done by decrementing in the suffix array of t each entry of the array `pos`, modulo $n+1$ if necessary. Formally, we call $\texttt{left}[i] = \texttt{pos}[i]_{\text{dec}\, n+1}$ for all i, $1 \leq i \leq n$, where $x_{\text{dec}\, n+1} := (x-1) \mod (n+1)$. Then $\text{bwt}(t)[i] = s[\texttt{left}[i]]$.

Observation 68 Facing the output string $\text{bwt}(t)$ it can be seen that equal characters are often grouped together.

This phenomenon is called *left context*. It can be observed in every natural language and it is due to their structural properties, of course with differently distributed probabilities for every language.

Example 69 In an English text, there will be many occurrences of the word 'the' and also some occurrences of 'she' and 'he'. Sorting the cyclic rotations of the text will lead to a large group of 'h's in the first column and thus 't's, 's's and gaps will be grouped together in the last column. This can be explained by the probability for a 't' preceding 'he', which is obviously quite high in contrast to the probability of e.g. an 'h' preceding 'he'. ◀

Reconstruction. Besides $L = \text{bwt}(t)$, which is the last column of the matrix M, the first column F is available by lexicographically sorting $\text{bwt}(t)$. The reconstruction of the text is done in a back-to-front manner by a method called **Last-to-Front Mapping** (LF mapping) based on of the following observation on the Burrows-Wheeler transformation, which we also call the **central property of the BWT**:

Observation 70 The ith occurrence of a character x in L refers to the same character in the original text as the ith occurrence of x in F.

The LF mapping is accomplished by the following steps:

1. Examine the sentinel \$ in the last column.
2. Search the occurrence of the reconstructed character in F by exploiting Observation 70.
3. Examine the precursor of the reconstructed character. Due to the fact that the matrix contains the cyclic rotations, each character in the last column is the precursor of the character in the first column of the same row.

Repeat step 2 and step 3 until the sentinel \$ is reached again. The reconstruction phase ends and the original input string is obtained.

Efficient ways how to perform these searches are known, see e.g. Ferragina and Manzini (2005), Kärkkäinen (2007) or Adjeroh et al. (2008). The first of these approaches (FM-index) is discussed in Section 14.1.

Example 71 Let the text be $t = s\$ = \texttt{STETSTESTE\$}$.

Transformation. Construct the matrix M by building the cyclic rotations of t and sorting them (shown in Figure 13.1). The Burrows-Wheeler transformation $\text{bwt}(t)$ can be found in the last column $L = \text{bwt}(t) = \texttt{ETTTET\$SSSE}$.

F									L
$	S	T	E	T	S	T	E	S	T	E
E	$	S	T	E	T	S	T	E	S	T
E	S	T	E	$	S	T	E	T	S	T
E	T	S	T	E	S	T	E	$	S	T
S	T	E	$	S	T	E	T	S	T	E
S	T	E	S	T	E	$	S	T	E	T
S	T	E	T	S	T	E	S	T	E	$
T	E	$	S	T	E	T	S	T	E	S
T	E	S	T	E	$	S	T	E	T	S
T	E	T	S	T	E	S	T	E	$	S
T	S	T	E	S	T	E	$	S	T	E

Figure 13.1: This is the matrix M containing the cyclic rotations.

Retransformation. Reconstruct column F by lexicographically sorting the last column $L = \text{bwt}(t) =$ `ETTTET$SSSE`, giving $F =$ `$EEESSSTTTT` (this is sketched in Figure 13.2). Starting with the sentinel in L, the sentinel in F is searched. Because it is sorted, the sentinel of course is found at position 0. Thus the second reconstructed character is the `E` at $L[0]$. This is the first `E` in L, so the first `E` in F is searched, etc. In the end, the original input $t =$ `STETSTESTE$` is achieved again in right to left order.

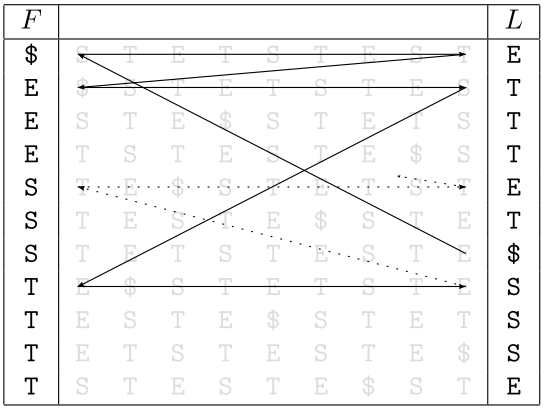


Figure 13.2: Illustration of the reconstruction path through the matrix.

◀

13.3 Exact String Matching

Consider again the task of searching a pattern p in a sequence s. Amazing about the BWT is the existence of an exact string matching algorithm that works directly on the transformed text, the so-called **backward search**. Similar to the reconstruction step of the BWT, this algorithm also works in a back-to-front manner, by first searching the last

character of the pattern in F. All occurring precursors in L are then compared to the next character in the pattern, matching ones are marked in F, and so on.

Eventually, this is a slightly modified application of the second and third step of the LF-mapping described before. But instead of always searching for one character in F we search for a range of characters and examine their precursors in L for matching.

This leads to the following iteration, which is started for $i := |p| - 1$ (index of last position in the pattern).

1. Determine the interval of all occurrences of $p[i]$ in F.
2. Continue with the same interval in L which corresponds to the precursors of the currently considered characters.
3. For all entries in the current interval in L which equal $p[i-1]$, determine their occurrence in F (LF-mapping). These define a new interval in F. Therefore it suffices to perform the LF-mapping only for the first and the last such element. (See Figure 13.3 for an example.)
   a) If this is empty, the algorithm ends and the pattern does not occur in the text.
   b) If the interval is not empty and $i = 0$, the pattern is found at the corresponding positions.
   c) Otherwise decrease i by 1 and continue with step 2

If the pattern was found we can examine two more properties. First, the number of precursors matching the first character of the pattern equals the number of occurences in the text. Second, if at the beginning the suffix array of the original text was stored and sorted along with the rotations, then after searching the first character of the pattern in F, the values at the corresponding indices in the suffix array will refer to the positions where the pattern is found the original text.

Example 72 Consider the text $t = s\$ =$ `RHABARBERBARBARA$` and the pattern $p =$ `BARBAR`. The Burrows-Wheeler Transform of t is $L = \text{bwt}(t) =$ `ARHBBBRRARBRAAEA$`.

Figure 13.3 illustrates the search: First, the last character of p, namely `R`, is searched in F and its first and last occurrences are highlighted in black. Then the next character of p, which is `A`, is searched in L. The first and last occurrences of `A` that are precursors of the already highlighted `R`s are also marked. The newly marked `A`s are searched in F and the precursors, which correspond to the next character of the pattern (`B`), are searched. Again, the first and last are marked in L and then searched in F. The next characters in p are `R` and `A`. Here there is only one matching precursor. Afterwards the last searched character, the `B`, is found as a precursor of the `A`.

In this case the pattern is found exactly once. To gain the index where the pattern starts in the text, the occurrence of the last found character (the first character of p) is searched in F. The corresponding value of the suffix array determines the wanted index. The suffix array in this example is [16,15,2,13,10,4,12,9,3,6,7,1,14,11,8,5,0]. Our search ended at the second occurring `B`, which is at index 7 in F. Looking up the value of the suffix array at

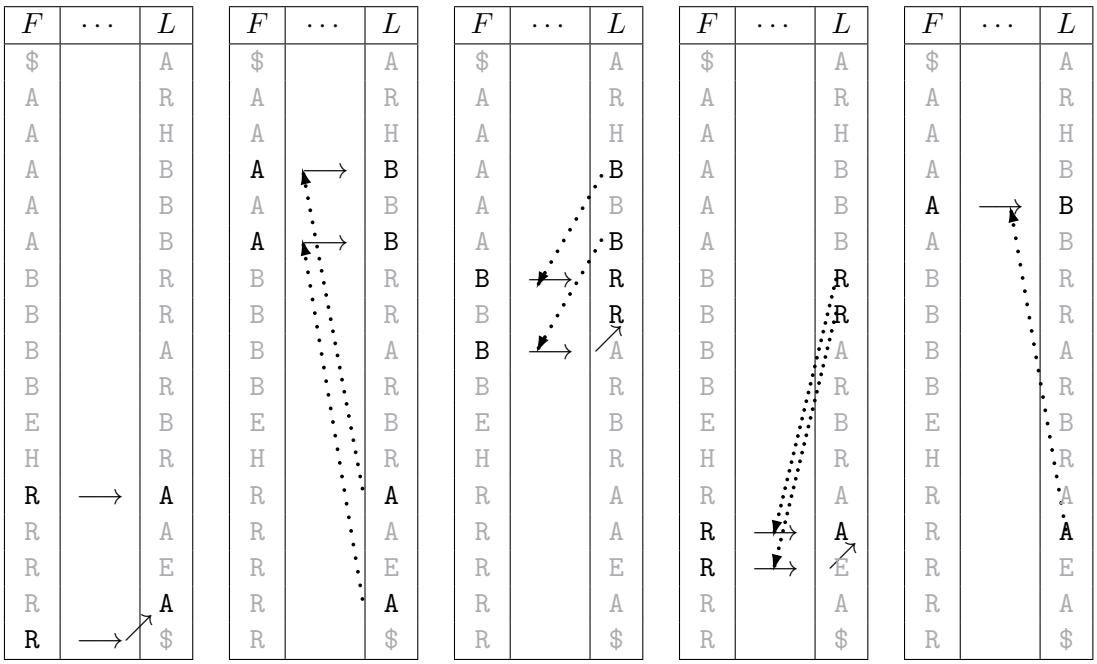


Figure 13.3: Searching the pattern BARBAR in RHABARBERBARBARA$.

index 7 gives a value of 9, therefore index 9 is the position where the pattern starts in s. ◀

13.4 Other Applications

Besides exact string matching, also other common string matching problems can be solved with the BWT. For example, read mappers have been developed that work on the BWT, e.g. BWA (Li and Durbin, 2009, 2010) or Bowtie (Langmead et al., 2009). They apply a semi-global alignment algorithm on short queries and contain a Smith-Waterman-like heuristic for longer queries, allowing higher error rates. For more information see `http://bio-bwa.sourceforge.net/bwa.shtml`.

The BWT of a text is further suitable for effective compression, e.g. with move-to-front or run-length encoding. The occurrence of grouped characters enables a significantly better compression than compressing the original text.

13.4.1 Compression with Run-Length Encoding

The run-length encoding (RLE) algorithm constructs a sequence of numbers and characters from the text. The text is searched for runs of equal characters, and these are replaced by the number of occurrences of this character in the run and the character itself, e.g. instead of EEEEE just 5E is saved. A threshold value determines how long a run at least must be to be compressed in this way. Default for this threshold is 3, so single and double characters are not changed, but instead of a run of three times the character c, the algorithm will save

3c. Texts with many and long runs thus will obviously be effectively compressed. This data structure is also called the **run-length compressed BWT (RLBWT)**. Clearly, it can be stored on $O(r)$ space where r is the number of runs of consecutive identical characters in bwt(t).

Example 73 Considering again the example $t = s\$ =$ `STETSTESTE$`, the compression with RLE will have no advantage. If instead the Burrows-Wheeler transform of t, bwt(t) = `ETTTET$SSSE`, is compressed, the space requirement is reduced by two characters, since the compressed string is rle(bwt(t)) = `E3TET$3SE`. As soon as the input string gets longer, it is likely that more grouped characters occur, and thus the space requirement reduction is even more significant. ◀

13.4.2 Matching Statistics

The **matching statistics** of a (long) sequence s of length n with respect to a (short) pattern p is an integer array M of length n such that $M[i]$ is the length of the longest prefix of the suffix $s[i, n]$ that is a substring of p. Like with suffix trees and suffix arrays, the matching statistics can also be computed efficiently using the BWT of p.

The main idea is to construct the BWT of $p\$$ and then perform a backward search with s, as long as an extension (to the left) is possible, i.e. the corresponding interval in F is not empty. This gives the M values for the suffixes of s. Once an extension is no longer possible, the "parent interval" (according to the *lcp interval* structure discussed in Section 12.2) of the last non-empty interval has to be chosen. This corresponds to removing non-branching suffixes of the previous entry in the matching statistic. Here the backward search is continued, yielding (unless again empty) the M values for the next position(s) of s. The procedure is continued until the beginning of s is reached.

Details can be worked out as an exercise.

14 Using the BWT Efficiently

Originally the Burrows-Wheeler transformation was developed with the application of better text compression and decompression in mind. This can essentially be implemented by iterated LF-mapping as described in the previous chapter. Later the method has also been used for string matching with the seed-and-extend approach. This is realized by backward search, which was also described in the previous chapter.

In this chapter we want to study how the same functionalities (LF-mapping, backward search) can be implemented in even less space, using the run length-encoded BWT. This can be very effective in practice, especially for very large texts.

Before we explain the main topic of this chapter, the MOVE datastructure, we quickly introduce two other datastructures that were important landmarks on the way.

14.1 The FM-Index

Ferragina and Manzini (2005) introduced the FM-index, which is the BWT enhanced by two access functions, **rank()** and **select()**, that allow a quick implementation of LF-mapping, defined as follows.

Definition 74 Given an input string s, let $t = s\$$, $L = bwt(t)$ and F be the alphabetically ordered list of characters in $s\$$. Then

$$\text{rank}(i) = \text{the number of occurrences of character } L[i] \text{ before index } i \text{ in } L$$

and

$$\text{select}(c, i) = \text{the } i\text{th occurrence of character } c \text{ in } F$$

Then $LF(i) = \text{select}(L[i], \text{rank}(i))$.

Efficient implementations of the two functions can be realized as follows:

select can easily be implemented in $O(\sigma)$ space and constant time: $\text{select}(c, i) = C[c] + i$, where $C[c]$ is the index just before the first occurrence of c in F, also called the **accumulative count** $C[c] = |\{i \mid F[i] < c\}|$.

rank can trivially be implemented in $O(n)$ space and $O(1)$ time by just storing for each index i its rank. A more succinct representation is not so easy, but with some effort it is possible to achieve with almost constant time access $O(\log \log_w \sigma)$ where w is the computer word size. Details are omitted here, see e.g. Belazzougui and Navarro (2015).

With these two little additions, several applications can be implemented very efficiently, for example:

Count: Return the number of occurrences of a pattern p in t, by iterated backward search, in $O(|p|)$ time.

Locate: Return the index in t of a given index in L, by counting the number of LF-mappings until the end of string marker \$ is reached, in $O(n)$ time. For long strings this is time-inefficient. The other extreme would be to store the rank for every index i as indicated above, which is space-inefficient. A good compromise is to store a few checkmarks for a subset of the indices in L, so that the series of LF-mappings will never be too long.

14.2 The r-Index

Gagie, Navarro and Prezza (2018) studied the problem of performing rank() and select() directly on the run length compressed BWT, which led to the r-index datastructure. In fact, it is quite a complicated construct consisting of several bitvectors, sparse bitvectors, wavelet trees and further auxiliary datastructures. The result, however, is quite impressive, as the r-index requires only $O(r)$ space and can perform the LF-mapping in *almost* constant time. (A variant requires *almost* $O(r)$ space and performs the LF-mapping in constant time.) Details go beyond the scope of this course.

14.3 The MOVE datastructure

A datastructure that achieves even better results than the r-index, while being conceptually much simpler, is MOVE (Nishimoto and Tabei, 2021; Zakeri et al., 2024). The central observation here is that – because of the central property of the BWT (Observation 70) – every run R in $L = \text{bwt}(t)$ is also a run in F (although the runs in F are usually longer than those in L).

Example 75 Let $t =$ `aacaabaaabaa$` ($n = 13$), then $L = \text{bwt}(t) =$ `aabbac$aaaaaa` ($r = 6$). The MOVE datastructure is shown in Figure 14.1. It is easy to see that the runs in L (denoted by circled numbers) can be found in F in permuted order. ◀

The MOVE datastructure stores for each run $\textcircled{i}$, $1 \leq i \leq r$, of the BWT four components. The first two components, c and ℓ, describe the run itself:

- $\textcircled{i}.c$ is the (repeated) character of run $\textcircled{i}$.
- $\textcircled{i}.\ell$ is the length of run $\textcircled{i}$.

The other two components, to-run and o_L, describe properties of the head (first element) of run $\textcircled{i}$:

- $\textcircled{i}$.to-run is the run inside which the head of $\textcircled{i}$ arrives in L after one LF-mappping.

	F		L	
(5)	$	1	a	(1)
(1)	a	2	a	
	a	3	b	(2)
(3)	a	4	b	
(6)	a	5	a	(3)
	a	6	c	(4)
	a	7	$	(5)
	a	8	a	(6)
	a	9	a	
	a	10	a	
(2)	b	11	a	
	b	12	a	
(4)	c	13	a	

Figure 14.1: MOVE datastructure of the string $s\$ =$ `aacaabaaabaa$`.

- $(i).o_L$ is the offset of the head of (i) inside (i).to-run after the LF-mapping.

Example 75 (cont'd) The MOVE datastructure for $t =$ `aacaabaaabaa$` looks as follows:

			head of run	
run	c	ℓ	**to-run**	o_L
(1)	a	2	(1)	1
(2)	b	2	(6)	3
(3)	a	1	(2)	1
(4)	c	1	(6)	5
(5)	$	1	(1)	0
(6)	a	6	(3)	0

◀

A position p in column L of the BWT is stored as a pair $p = ((i), o)$ where (i) is the position's run index in L and o, $0 \leq o < (i).\ell$ is the offset of the position inside the run. MOVE is the operation that maps a position p in L representing text index $t[j]$ to position p^* in L representing its left neighbor, $t[j-1]$. (Note the similarity to LF-mapping: LF-mapping starts at a position in F, then "horiziontally" maps to L and finally back to the corresponding run index in F; while MOVE starts at a position in L, then maps to the corresponding run index in F and finally "horizontally" back to L.) A MOVE can be performed efficiently without decompressing the BWT as follows:

$$MOVE((i), o) = \text{ff}((i).\text{to-run}, o - (i).o_L)$$

where

$$\text{ff}((i^*), o^*) = \begin{cases} ((i^*), o^*) & \text{if } o^* < (i^*).\ell \\ \text{ff}((i^*+1), o^* - (i^*).\ell) & \text{otherwise} \end{cases}$$

Note that the "fast forward" function ff is necessary to ensure that at the end the new position $p^* = (\textcircled{i^*}, o^*)$ lies inside its run $\textcircled{i^*}$. Unfortunately, this incremental approach is asymptotically not optimal. An alternative that guarantees a constant number of steps per MOVE-mapping (by splitting some runs so that the incremental search terminates quickly) and therefore is asymptotically optimal exists as well (Nishimoto and Tabei, 2021) but is slower in practice (Zakeri et al., 2024).

Example 75 (cont'd) Consider the string matching task of counting the number of occurrences of pattern `aaba` in text `aacaabaaabaa$`. As for the regular BWT we perform backward search and iterated MOVE operations.

We start with the last letter of the pattern `a`. The interval containing all occurrences of `a` in the text is delimited by the topmost position $p_1^\top = p^\top(\texttt{a}) = (\textcircled{1}, 0)$ of an `a` in L and the bottommost position $p_1^\bot = p^\bot(\texttt{a}) = (\textcircled{6}, 5)$ of an `a` in L. (The position pairs $(p^\top(c), p^\bot(c))$ can easily be preprocessed for each letter c of the alphabet.) After one MOVE operation we reach $MOVE(p_1^\top) = (\textcircled{1}, 1)$ and $MOVE(p_1^\bot) = (\textcircled{6}, 2)$. While $MOVE(p_1^\top) = MOVE(\textcircled{1}, 0) = \text{ff}(\textcircled{1}, 1) = (\textcircled{1}, 1)$ gives this result immediately, $MOVE(p_1^\bot) = MOVE(\textcircled{6}, 5)$ initially gives $\text{ff}(\textcircled{3}, 5)$, which then gives $\text{ff}(\textcircled{4}, 4)$, then $\text{ff}(\textcircled{5}, 3)$ and finally $\text{ff}(\textcircled{6}, 2) = (\textcircled{6}, 2)$.

Once we have performed the two $MOVE$ operations of the upper and lower bounds, we test if the letters corresponding to these positions represent the character `b` (the second last letter of our pattern). If not, we proceed down (with the upper position $p^\top$) and up (with the lower position $p_\bot$) until an occurrence of the letter `b` is reached. This can be done run-wise because all letters in a run are the same. We reach the positions $p_2^\top = (\textcircled{2}, 0)$ and $p_2^\bot = (\textcircled{2}, 1)$. Proceeding in the same way with the next pattern character `a`, we reach $p_3^\top = (\textcircled{6}, 3)$ and $p_3^\bot = (\textcircled{6}, 4)$ and finally with the first pattern character `a` we reach $p_4^\top = (\textcircled{6}, 0)$ and $p_4^\bot = (\textcircled{6}, 1)$. That means, the pattern `aaba` occurs two times in the text. ◀

It is also possible to efficiently track the current text positions during the LF-mapping so that the position of a matching pattern can be reported as well. This, however, goes beyond the scope of these lecture notes.

15 Whole Genome Alignment

Because more and more prokaryotic and eukaryotic genomes are sequenced, and their comparison reveals much information about their function and evolutionary history, the alignment of whole genomes is in general very valuable.

We start this chapter with a few general remarks on whole genome alignment:

1. The alignment of whole genomes makes sense only if there is a global similarity between the compared genomes. If the genomes do not have a common layout, other methods like genome rearrangement studies (Gascuel, 2005) should be applied.

2. In principle, like in the case of "normal" alignments, alignments of two or more whole genomes could be computed by dynamic programming. Since one deals with large amounts of data, though, the quadratic (or exponential in the multiple case) time complexity leads to extremely long computation times.

3. The previous point is why in practice one needs to apply faster methods that are specialized for similar DNA sequences, as they often appear in closely related genomes. Most of the commonly used methods for whole genome alignment perform two successive steps. First, identical (or highly similar) subregions (**seeds**) between the input genomes are identified that in a second step are then connected (**chained**) in the best possible way. This procedure may be applied recursively to smaller, not yet aligned regions.

In this chapter we will explain standard solutions for both steps in the following two subsections. Then we describe a few of the more popular tools for whole genome alignment, MUMmer (Delcher et al., 1999, 2002; Kurtz et al., 2004) and MAUVE (Darling et al., 2004).

15.1 Seed Detection

In whole genome alignment, two types of seeds are in popular use: Maximal unique matches (MUMs) and maximal exact matches (MEMs). Both may be defined for two or multiple sequences. We begin with MUMs that have already been studied in Section 11.4 for the case of $k = 2$ sequences.

Definition 76 A MUM (Maximal Unique Match) is a substring w that occurs exactly once in each sequence s_i, $1 \leq i \leq k$, and that can not simultaneously be extended to the left or to the right in every sequence. A MUM in more than two sequences is sometimes called a *multiMUM*.

The definition of MEMs is less restrictive as they may occur several times in the same sequence:

Definition 77 A MEM (Maximal Exact Match) is a substring w that occurs in all sequences $s_1, s_2, \ldots, s_k$ and that can not simultaneously be extended to the left or to the right in every sequence. A MEM in more than two sequences is sometimes called a *multi-MEM*.

Formally, a MEM is a tuple $(i_1, \ldots, i_k; L)$ such that $w = s_1[i_1, i_1+L-1] = s_2[i_2, i_2+L-1] = \ldots = s_k[i_k, i_k+L-1]$, $|\{s_1[i_1-1], s_2[i_2-1], \ldots, s_k[i_k-1]\}| \geq 2$ and $|\{s_1[i_1+L], s_2[i_2+L], \ldots, s_k[i_k+L]\}| \geq 2$. A MUM, in addition, has no other occurrence of w in any of the input sequences $s_1, s_2, \ldots, s_k$. Often, a minimum length ℓ is added to this definition, so that only MEMs (or MUMs) are of interest for which $L \geq \ell$.

Both MUMs and MEMs can be efficiently found using the generalized suffix tree T of the sequences $s_1, s_2, \ldots, s_k$ introduced in Section 10.3.

We begin with multiMEMs: It is easy to see that there is a correspondence between the internal nodes of T that have, in their subtree, at least one leaf for each input sequence, and the right-maximal exact matches. These nodes can be found by a bottom-up traversal of T, storing at each node the set of input sequences for which leaves exist in the corresponding subtree and their positions in the input sequences. In addition, to test for left-maximality, one has to test that there are no extensions possible to the left by looking up the character immediately to the left of their start positions in the input sequences. This simple algorithm takes $O(n+kr)$ time where $n = n_1+n_2+\cdots+n_k$ is the total length of all k genomes and r is the number of right-maximal exact matches. But also algorithms that run in $O(n)$ time are possible with some enhancement of the data structure.

MUMs have the additional restriction that in the subtree below the endpoint of the MUM, each sequence $s_1, s_2, \ldots, s_k$ must correspond to *exactly* one leaf. Also MUMs of k sequences can be found in $O(n)$ time.

15.2 Chaining

Once a set of seeds has been obtained, they have to be connected to form the genome alignment. This step is called **chaining**. It can be modeled as the following graph problem: Let $R = \{r_1, \ldots, r_z\}$ be the set of z seeds found in the first phase of the algorithm. Define a partial order $\prec$ on seeds where $r_i \prec r_j$ if and only if the end of seed r_i is smaller than the beginning of seed r_j in both s_1 and s_2. The directed, vertex-weighted graph $G = (V, E)$ contains the vertex set $V = R \cup \{\text{start}, \text{stop}\}$ and an edge $(r_i \to r_j) \in E$ if and only if $r_i \prec r_j$. Moreover, $(\text{start} \to r_i) \in E$ and $(r_i \to \text{stop}) \in E$ for all $1 \leq i \leq z$. The weight $w(v)$ of a vertex v is defined as the length of the seed represented by v, and $w(\text{start}) = w(\text{stop}) = 0$.

Problem 78 (Chaining Problem) Find a chain $c = (r_{i_0}, r_{i_1}, r_{i_2}, \ldots, r_{i_\ell}, r_{i_{\ell+1}})$, with $r_{i_0} = \text{start}$ and $r_{i_{\ell+1}} = \text{stop}$, where two neighboring vertices are connected by an edge $(r_{i_j} \to r_{i_{j+1}})$ for all $0 \leq j \leq \ell$, of heaviest weight $w(c) := \sum_{j=1}^{\ell} w(r_{i_j})$.

In principle, one could use Dijkstra's algorithm for finding shortest paths in a weighted graph for solving the chaining problem. This would take $O(|V|^2)$ time, or $O(|V| \log |V|)$ time using some additional tricks. It is well known, however, that in an acyclic graph a path of maximum weight can be found in $O(|V| + |E|)$ time by topologically sorting the vertices and then applying dynamic programming. Here, this easily yields an $O(z^2)$ (or $O(z \log z)$) time algorithm for the chaining. However, since the seeds can be linearly ordered, using a heaviest increasing subsequence algorithm the computation can be reduced further (Abouelhoda and Ohlebusch, 2005).

15.3 Collinear Multiple Genome Alignment (MUMmer)

A popular alignment program for two whole genomes is MUMmer whose first (Delcher et al., 1999) and second (Delcher et al., 2002) versions differ only slightly.

The general strategy of the overall algorithm is as follows:

1. Given k genomes $s_1, \ldots, s_k$, all MUMs are found using the generalized suffix tree as described in the previous section.
2. The chain of compatible MUMs that maximizes the weight along its path is selected, where a set of MUMs is *compatible* if the MUMs can be ordered linearly.
3. Short gaps (up to 5000 base pairs) are filled by ordinary alignment. Long gaps remain unaligned.

Overall, the first two phases take time $O(|s_1|+\cdots+|s_k|+z^2)$ or $O(|s_1|+\cdots+|s_k|+z \log z)$, depending on the time needed for chaining. The time used by the last phase depends on the size of the remaining gaps and the used algorithm. In the worst case the last phase dominates the whole procedure, for example when no single MUM was found. But in a typical case, where the genomes are of considerable global similarity, not too many and not too large gaps should remain such that the last phase does not require too much time.

One design decision in MUMmer 1 and 2 was to use MUMs as output of the filtration phase. The advantage is that this gives reliable anchors since the uniqueness is a strong hint that the regions in the two genomes indeed correspond to each other, i.e. are orthologous. However, if more than two genomes are compared, it is unlikely that there exist many MUMs that are present *and unique* in all considered sequences.

MUMmer in its third version (Kurtz et al., 2004) is therefore based on multi-MEMs, as the authors have noted that these are less restrictive and the chaining does not become more complicated.

15.4 Multiple Genome Alignment with Rearrangements (MAUVE)

Another genome alignment program described here is MAUVE (Darling et al., 2004). This program uses multi-MUMs as seeds, but can also deal with rearrangements, i.e., its

chaining algorithm is more general than in MUMmer, as it is not restricted to finding one chain of collinear seeds. Instead it looks for several locally collinear blocks of seeds. Among these blocks, by a greedy selection procedure the most reliable blocks are selected and assembled into a global "alignment". MAUVE is applied with less restrictive parameters recursively to the regions not aligned in the previous phase. It can be summarized as follows:

1. Find local alignments (multiMUMs).
2. Use the multiMUMs to calculate a phylogenetic guide tree.
3. Select a subset of the multiMUMs to use as anchors – these anchors are partitioned into locally collinear blocks (LCBs).
4. Perform recursive anchoring to identify additional alignment anchors within and outside each LCB.
5. Perform a progressive alignment of each LCB using the guide tree.

MAUVE can be found (but is no longer actively maintained) at `https://darlinglab.org/mauve/mauve.html`.

Bibliography

M. I. Abouelhoda and E. Ohlebusch. Chaining algorithms for multiple genome comparison. *J. Discr. Alg.*, 3:321–341, 2005.

M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *Proceedings of the Second International Workshop on Algorithms in Bioinformatics (WABI 2002)*, volume 2452 of *LNCS*, pages 449–463, 2002.

D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching.* Springer Verlag, 2008.

S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic Local Alignment Search Tool (BLAST). *J. Mol. Biol.*, 215:403–410, 1990.

S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Res.*, 25(17):3389–3402, Sep 1997.

D. Belazzougui and G. Navarro. Optimal lower and upper bounds for representing sequences. *ACM Trans. Alg.*, 11(4):1–21, 2015.

B. Buchfink, C. Xie, and D. Huson. Fast and sensitive protein alignment using DIAMOND. *Nat. Methods*, 12:59–60, 2015.

M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report TR 124, Digital Equipment Corporation, Palo Alto, CA, 1994.

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

J.-M. Claverie and C. Notredame. *BIoinformatics for Dummies.* John Wiley & Sons (For Dummies series), 2nd edition, 2007.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 2nd edition, 2001.

A. C. E. Darling, B. Mau, F. R. Blattner, and N. T. Perna. Mauve: Multiple alignment of conserved genomic sequence with rearrangements. *Genome Res.*, 14(7):1394–1403, 2004.

A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic Acids Res.*, 27(11):2369–2376, 1999.

A. L. Delcher, A. Phillippy, J. Carlton, and S. L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Res.*, 30(11):2478–2483, 2002.

R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge University Press, 1998.

R. C. Edgar. Muscle: a multiple sequence alignment method with reduced time and space complexity. *BMC Bioinformatics*, 5:113, 2004a.

R. C. Edgar. Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Res.*, 32(5):1792–1797, 2004b.

I. Elias. Settling the intractability of multiple alignment. *J. Comp. Biol.*, 13(r72):1323–1339, 2006.

M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Annu. Symp. Found. Comput. Sci., FOCS 1997*, pages 137–143, New York, NY, 1997. IEEE Press.

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

P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

W. M. Fitch. Toward defining the course of evolution: Minimum change for a specific tree topology. *Syst. Zool.*, 20(4):406–416, 1971.

T. Gagie, G. Navarro, and N. Prezza. Optimal-time text indexing in bwt-runs bounded space. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 1459–1477, 2018.

O. Gascuel, editor. *Mathematics of Evolution and Phylogeny*. Oxford University Press, 2005.

R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. *Softw. Pract. Exper.*, 33(11):1035–1049, 2003.

O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162: 705–708, 1982.

D. Gusfield. Efficient methods for multiple sequence alignment with guaranteed error bounds. *Bull. Math. Biol.*, 55(1):141–154, 1993.

D. Gusfield. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, 1997.

S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc. Natl. Acad. Sci. USA*, 89:10915–10919, 1992.

D. G. Higgins and P. M. Sharp. Clustal V: Improved software for multiple sequence alignment. *Comput. Appl. Biosci.*, 8:189–191, 1992.

M. Hirosawa, Y. Totoki, M. Hoshida, and M. Ishikawa. Comprehensive study on iterative algorithms of multiple sequence alignment. *Comput. Appl. Biosci.*, 11(1):13–18, 1995.

T. J. P. Hubbard, A. M. Lesk, and A. Tramontano. Gathering them into the fold. *Nat. Structural Biology*, 4:313, 1996.

J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theor. Comput. Sci.*, 387(3):249–257, 2007.

J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Proceedings of the 13th International Conference on Automata, Languages and Programming (ICALP)*, volume 2719 of *LNCS*, pages 943–955, 2003.

R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Conf. Proc. 4th Annu. ACM Symp. Theory Comput., STOC 1972*, pages 125–136, 1972.

T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 12th Symposium con Combinatorial Pattern Matching (CPM)*, volume 2089 of *LNCS*, pages 181–192, 2001.

J. Kececioglu. The maximum weight trace problem in multiple sequence alignment. In *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching, CPM 1993*, volume 684 of *LNCS*, pages 106–119, 1993.

J. Kececioglu and D. Starrett. Aligning alignments exactly. In *Proc. of the Eighth Annual International Conference on Computational Molecular Biology, RECOMB 2004*, pages 85–96, 2004.

B. Knudsen. Optimal multiple parsimony alignment with affine gap cost using a phylogenetic tree. In *Proceedings of the Third International Workshop on Algorithms in Bioinformatics, WABI 2003*, volume 2812 of *LNBI*, pages 433–446, 2003.

S. Kurtz, A. Phillippy, A. L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biol.*, 5:R 12, 2004.

B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biol.*, 10:R 25, 2009.

V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

H. Li and R. Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

H. Li and R. Durbin. Fast and accurate long-read alignment with burrows-wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

D. Maier and J. A. Storer. A note on the complexity of the superstring problem. Technical Report 233, Department of Electrical Engineering and Computer Science, Princeton University, 1977.

U. Manber and G. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Computing*, 22(5):935–948, 1993.

E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2): 262–272, 1976.

D. W. Mount. *Bioinformatics: Sequence and Genome Analysis.* Cold Spring Harbor Laboratory Press, 2nd edition, 2004.

S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Mol. Biol.*, 48(3):443–453, 1970.

T. Nishimoto and Y. Tabei. Optimal-time queries on BWT-runs compressed indexes. In *Proceedings of ICALP 2021*, 2021. Paper number 101.

C. Notredame, D. G. Higgins, and J. Heringa. T-Coffee: A novel method for fast and accurate multiple sequence alignment. *J. Mol. Biol.*, 302:205–217, 2000.

N. Saitou and M. Nei. The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Mol. Biol. Evol.*, 4(4):406–425, 1987.

D. Sankoff. Minimal mutation trees of sequences. *SIAM J. Appl. Math.*, 28(1):35–42, 1975.

J. C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology.* PWS Publishing Company, 1997.

F. Sievers, A. Wilm, D. Dineen, T. J. Gibson, K. Karplus, W. Li, R. Lopez, H. McWilliam, M. Remmert, J. Söding, et al. Fast, scalable generation of high-quality protein multiple sequence alignments using clustal omega. *Mol. Syst. Biol.*, 7(1), 2011.

T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147(1):195–197, 1981.

J. Stoye. Multiple sequence alignment with the divide-and-conquer method. *Gene COMBIS*, 211(2):GC45–GC56, 1998.

J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res.*, 22(22):4673–4680, 1994.

J. D. Thompson, T. J. Gibson, F. Plewniak, F. Jeanmougin, and D. G. Higgins. The ClustalX windows interface: Flexible strategies for multiple sequence alignment aided by quality analysis tools. *Nucleic Acids Res.*, 24:4876–4882, 1997.

E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14:249–260, 1995.

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

M. S. Waterman. *Introduction to Computational Biology.* Chapman and Hall, 1995.

M. S. Waterman, T. F. Smith, and W. A. Beyer. Some biological sequence metrics. *Adv. Math.*, 20:367–387, 1976.

M. S. Waterman, S. Tavaré, and R. C. Deonier. *Computational Genome Analysis: An Introduction*. Springer, 2nd edition, 2005.

P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11. IEEE, 1973.

M. Zakeri, N. K. Brown, O. Y. Ahmed, T. Gagie, and B. Langmead. Movi: A fast and cache-efficient full-text pangenome index. *iScience*, 27(111464), 2024.