Sequence Analysis

Lecture notes

Faculty of Technology, Bielefeld University

Winter 2015/16

Contents

1 Overview **1**
1.1 Application Areas of Sequence Analysis 1
1.2 A Small Selection of Problems on Sequences 2

2 Basic Definitions **3**
2.1 Sets and Basic Combinatorics 3
2.2 Review of Elementary Probability Theory 4
2.3 Asymptotics 6
2.4 Alphabets and Sequences 7

3 Metrics on Sequences **11**
3.1 Problem Motivation 11
3.2 Definition of a Metric 12
3.3 Transformation Distances 12
3.4 Metrics on Sequences of the Same Length 14
3.5 Edit Distances for Sequences 16
3.6 An Efficient Algorithm to Compute Edit Distances 18
3.7 The q-gram Distance 21
3.8 The Maximal Matches Distance 26
3.9 Filtering 28

4 Pairwise Sequence Alignment **29**
4.1 Definition of Alignment 29
4.2 The Alignment Score 31
4.3 The Alignment Graph 32
4.4 A Universal Alignment Algorithm 33
4.5 Alignment Types: Global, Free End Gaps, Local 34
4.6 Score and Cost Variations for Alignments 38

5 Advanced Topics in Pairwise Alignment **43**
5.1 Suboptimal Alignments . . . 43
5.2 Approximate String Matching . . . 44
5.3 The Forward-Backward Technique . . . 48
5.4 Pairwise Alignment in Linear Space . . . 50

6 Pairwise Alignment in Practice **55**
6.1 Alignment Visualization with Dot Plots . . . 55
6.2 Fundamentals of Rapid Database Search Methods . . . 56
6.3 BLAST: An On-line Database Search Method . . . 58

7 Suffix Trees **61**
7.1 Motivation . . . 61
7.2 An Informal Introduction to Suffix Trees . . . 62
7.3 A Formal Introduction to Suffix Trees . . . 64
7.4 Suffix Tree Construction: The WOTD Algorithm . . . 65
7.5 Suffix Tree Construction: Ukkonen's Linear-Time Online Algorithm . . . 67
7.6 Applications of Suffix Trees . . . 67
7.6.1 Exact String Matching . . . 67
7.6.2 The Shortest Unique Substring . . . 69
7.6.3 Maximal Repeats . . . 70
7.6.4 Maximal Unique Matches . . . 73

8 Suffix Arrays **77**
8.1 Motivation . . . 77
8.2 Basic Definitions . . . 78
8.3 Suffix Array Construction Algorithms . . . 79
8.3.1 Linear-Time Construction using a Suffix Tree . . . 79
8.3.2 Direct Construction . . . 80
8.3.3 Construction of the `rank` and `lcp` Arrays . . . 81
8.4 Applications of Suffix Arrays . . . 83

9 Burrows-Wheeler Transformation **85**
9.1 Introduction . . . 85
9.2 Transformation and Retransformation . . . 85
9.3 Exact String Matching . . . 88
9.4 Other Applications . . . 88
9.4.1 Compression with Run-Length Encoding . . . 89

10 Multiple Sequence Alignment **91**
10.1 Basic Definitions . . . 91
10.2 Why multiple sequence comparison? . . . 93
10.3 Multiple Alignment Scoring Functions . . . 94
10.4 Multiple Alignment Problems . . . 96
10.5 Digression: NP-completeness . . . 99
10.6 A Guide to Multiple Sequence Alignment Algorithms . . . 102

11 Algorithms for Sum-of-Pairs Multiple Alignment **103**
11.1 An Exact Algorithm . . . 103

11.1.1 The Basic Algorithm . . . 103
11.1.2 Variations of the Basic Algorithm . . . 105
11.2 Carrillo and Lipman's Search Space Reduction . . . 106
11.3 The Center-Star Approximation . . . 109
11.4 Divide-and-Conquer Alignment . . . 111

12 Tree Alignment Algorithms 117
12.1 Sankoff's Algorithm . . . 117
12.2 Generalized Tree Alignment . . . 119
12.2.1 Greedy Three-Way Tree Alignment Construction . . . 120
12.2.2 The Deferred Path Heuristic . . . 122

13 Whole Genome Alignment 123
13.1 Filter Algorithms . . . 124
13.2 Pairwise Genome Alignment (MUMmer 1 & 2) . . . 125
13.3 Multiple Genome Alignment (MGA and MUMmer 3) . . . 126
13.4 Multiple Genome Alignment with Rearrangements (MAUVE) . . . 126

A Distances versus Similarity Measures on Sequences 129
A.1 Biologically Inspired Distances . . . 129
A.2 From Distance to Similarity . . . 131
A.3 Log-Odds Score Matrices . . . 135

B Pairwise Sequence Alignment (Extended Material) 139
B.1 The Number of Global Alignments . . . 139

C Pairwise Alignment in Practice (Extended Material) 143
C.1 Fast Implementations of the Smith-Waterman Algorithm . . . 143
C.2 FASTA: An On-line Database Search Method . . . 143
C.3 Index-based Database Search Methods . . . 146
C.4 Software . . . 148

D Alignment Statistics 151
D.1 Preliminaries . . . 151
D.2 Statistics of q-gram Matches and FASTA Scores . . . 152
D.3 Statistics of Local Alignments . . . 154

E Basic RNA Secondary Structure Prediction 157
E.1 Introduction . . . 157
E.2 The Optimization Problem . . . 159
E.3 Context-Free Grammars . . . 160
E.4 The Nussinov Algorithm . . . 161
E.5 Further Remarks . . . 163

F Suffix Tree (Extended Material) 165
F.1 Memory Representations of Suffix Trees . . . 165

G Advanced Topics in Pairwise Alignment (Extended Material) 169
G.1 Length-Normalized Alignment . . . 169

G.1.1 A Problem with Alignment Scores . . . 169
G.1.2 A Length-Normalized Scoring Function for Local Alignment . . . 171
G.1.3 Finding an Optimal Normalized Alignment . . . 171
G.2 Parametric Alignment . . . 176
G.2.1 Introduction to Parametric Alignment . . . 176
G.2.2 Theory of Parametric Alignment . . . 177
G.2.3 Solving the Ray Search Problem . . . 178
G.2.4 Finding the Polygon Containing p . . . 181
G.2.5 Filling the Parameter Space . . . 182
G.2.6 Analysis and Improvements . . . 182
G.2.7 Parametric Alignment in Practice . . . 183

H Multiple Alignment in Practice: Mostly Progressive 185
H.1 Progressive Alignment . . . 185
H.1.1 Aligning Two Alignments . . . 187
H.2 Segment-Based Alignment . . . 188
H.2.1 Segment Identification . . . 189
H.2.2 Segment Selection and Assembly . . . 190
H.3 Software for Progressive and Segment-Based Alignment . . . 190
H.3.1 The Program Clustal W . . . 190
H.3.2 T-COFFEE . . . 191
H.3.3 DIALIGN . . . 192
H.3.4 MUSCLE . . . 192
H.3.5 QAlign . . . 194

Bibliography 195

Preface

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 first semester. The present three-hour course "Sequence Analysis" is taught in the third semester and is extended by a practical course in the fourth semester.

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 required to understand some of the material here:

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

These topic are assumed to have been covered before this course and the first two do not appear in these notes. In Section 2.3 the O notation is briefly reviewed, more advanced complexity topics are discussed in Section 10.5.

Advanced topics. Because the material is taught in one three-hour course, some advanced sequence analysis techniques are not covered. These include:

- 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 concave gap cost functions in pairwise alignment,

- in-depth discussion of methods used in practice. How to use existing software to solve a particular problem generally requires hands-on experience which can be conveyed in the "Sequence Analysis practical course". Some of the theory of such methods is covered in the Appendix of these notes.

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.
- 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 more recent 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.
- The classic algorithms textbook of Cormen et al. (2001) should be part of every 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. This is one of the best books available for computer scientists.

These lecture notes are an extended re-write of previous versions by Robert Giegerich, Stefan Kurtz (now in Hamburg), Enno Ohlebusch (now in Ulm), and Jens Stoye. This version contains more explanatory text and should be, to some degree, suitable for self study.

Sven Rahmann, July 2007

In the last years, these lecture notes were steadily improved. Besides minor corrections, many examples have been integrated and some parts were even completely rewritten. We want to thank Robert Giegerich for his comments and Eyla Willing for incorporating all the changes.

Peter Husemann, Jens Stoye and Roland Wittler, September 2010

We want to thank Katharina Westerholt for writing a new chapter on the *Burrows-Wheeler Transformation.*

Jens Stoye, September 2011

Until September 2012, "Sequence Analysis I" and "Sequence Analysis II" were taught as two-hour courses in the third and fourth semesters, accompanied by a two-hour practical course in the fourth semester. With the start of the winter semester in 2012/13 and the new study regulations, the lecture, now named "Sequence Analysis", is a three-hour course, only taught in the third semester, accompanied by an extended four-hours practical course in the fourth semester. Because there is now less time for the lecture and more time for the practical course, some topics covered by the lecture before are now shifted to the practical course. These lecture notes still include all topics, but the order is changed. Topics not discussed in the lecture are moved to the Appendix. In some chapters, just a few sections are moved to the Appendix, these chapters are marked as "Extended Material". In the beginning of the original chapters, the context of the extended material is summarized.

Linda Sundermann and Jens Stoye, October 2013

CHAPTER 1

Overview

1.1 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)

- Spoken language, bird song:
  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

1.2 A Small Selection of Problems on Sequences

The "inform" in (bio-)informatics comes from information. Google, Inc. states that its mission is "to organize the world's information and make it universally accessible and useful". 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.

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 than searching for each pattern separately are available.
5. Regular expression matching: Find all positions in a text that match an expression constructed according to specific rules.
6. Finding best approximations (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. Suffix sorting: Sort all suffixes of a given sequence lexicographically. This has many applications, including data compression and text indexing.
9. The "holy grail": 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 CVS and Subversion (`http://subversion.tigris.org`) 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.

CHAPTER 2

Basic Definitions

Contents of this chapter: Sets of numbers. Elementary set combinatorics. Alphabet. Sequence = string = word. Substring. Subsequence. Number of substrings and subsequences. Asymptotic growth of functions. Landau symbols. Logarithms. Probability vector. Uniform / Binomial / Poisson distribution.

2.1 Sets and Basic Combinatorics

Sets of numbers. The following sets of numbers are commonly known.

- $\mathbb{N} := \{1, 2, 3, \dots\}$ 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.
- $\mathbb{Q} := \{n/d : n \in \mathbb{Z},\, d \in \mathbb{N}\}$ is the set of rational numbers, i.e., fractions of a **numerator** n and a **denominator** d. $\mathbb{Q}_0^+$ is the set of nonnegative rational numbers.
- $\mathbb{R}$ ($\mathbb{R}_0^+$) is the set of (nonnegative) real numbers.
- $\mathbb{C} := \{x + \mathrm{i}y : x \in \mathbb{R},\, y \in \mathbb{R}\}$ is the set of complex numbers. Here $\mathrm{i} = \sqrt{-1}$ is the imaginary unit.

The **absolute value** or **modulus** of a number x is its distance from the origin and denoted by $|x|$, e.g. $|-5| = |5| = |3 + \mathrm{i}4| = 5$.

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 $\{a, a+1, \ldots, b-1, b\}$.

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:

- $n^k := n \cdot \cdots \cdot n$ (ordinary **power**, k factors),
- $n^{\underline{k}} := n \cdot (n-1) \cdot \cdots \cdot (n-k+1)$ (**falling power**, k factors),
- $n^{\overline{k}} := n \cdot (n+1) \cdot \cdots \cdot (n+k-1)$ (**rising power**, k factors),
- $n! := n^{\underline{n}} = 1^{\overline{n}}$ (**factorial**).

When choosing k elements out of n, we have n choices for the first element, $n-1$ for the second one, and so on, for a total of $n^{\underline{k}}$ choices. 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 $k^{\underline{k}} = k!$ many. It follows that there are $n^{\underline{k}}/k!$ different k-element subsets of an n-element set. This motivates the following definition of a **binomial coefficient**:

- $\binom{n}{k} := n^{\underline{k}}/k! = \frac{n!}{(n-k)! \cdot k!}$ (read as "n choose k").

2.2 Review of Elementary Probability Theory

A little understanding about probabilities is required to follow 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 **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. Here "i.i.d." stands for *independent*

and identically distributed. Then, for a *fixed* length q, the probability that a random word X of length q equals a given $x = x_1 \dots x_q$, 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 lenght 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 $i = 1, \dots, n$, 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 s 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 s successes (each with probability p_x) also has $n - s$ non-successes (each with probability $1 - p_x$). Therefore each such text has probability $p_x^s \cdot (1 - p_x)^{n-s}$. There are $\binom{n}{s}$ possibilities to choose the s out of n positions where the successes occur. Therefore, if S denotes the random variable counting successes, we have

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

for $s \in \{0, 1, \dots, n\}$, and $\mathbb{P}(S = s) = 0$ otherwise. This distribution is known as the **Binomial distribution** with parameters n and p_x. As said above, it specifies probabilities for the number s 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

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

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 2.1 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 $\sum_{k=0}^{n} \left(\binom{n}{k} \cdot \frac{1}{\sigma^k} \cdot \left(1 - \frac{1}{\sigma}\right)^{n-k}\right)^2$. ■

2.3 Asymptotics

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

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

$O(\cdot)$**:** 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.

$\Omega(\cdot)$**:** We write $f(n) \in \Omega(g(n))$ or $f(n) = \Omega(g(n))$ if and only if $g(n) \in O(f(n))$.

$\Theta(\cdot)$**:** If both $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$, then we write $f(n) = \Theta(g(n))$ and $g(n) = \Theta(f(n))$. In this case, f and g are **asymptotically of the same order**.

$\simeq$**:** We say that f and g are **asymptotically equivalent** and write $f(n) \simeq g(n)$ if $f(n)/g(n) \to 1$ as $n \to \infty$. This implies, and is stronger than, $f(n) = \Theta(g(n))$.

$o(\cdot)$**:** We write $f(n) \in o(g(n))$ if $f(n)$ is **asymptotically of smaller order** than $g(n)$, i.e., if $f(n)/g(n) \to 0$ as $n \to \infty$.

$\omega(\cdot)$**:** Conversely, $f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$.

A function in $o(n)$ is said to be **sublinear**. There is another weaker definition of sublinearity, according to which already $a \cdot n + o(n)$ is called (weakly) sublinear if $0 < a < 1$. 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 $\Omega(k^n) \cap O(c^n)$ for constants $c \geq k > 1$ is said to be of **exponential growth**. Functions such as $n^{\log n}$ that are no longer polynomial but not yet exponential are sometimes called of **superpolynomial growth**. Functions such as $n!$ or n^n are of **superexponential growth**.

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

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

A misconception sometimes intuitively guessed is that $\Theta(n \log n)$ is somehow the "middle road" between $\Theta(n)$ and $\Theta(n^2)$. This is absolutely not the case. Especially when it comes to running times of algorithms, an $O(n \log n)$ algorithm is essentially as good as a $\Theta(n)$ algorithm (and often much simpler, so the constant factors hidden in the O-notation are often smaller). To develop intuition, compare the running times $n^2 \in \Theta(n^2)$ with $10\, n \log n \in \Theta(n \log n)$ and $100\, n \in \Theta(n)$ of different hypothetical algorithms.

It is useful to remember that logarithms with respect to different bases $b, c > 1$ differ only by a constant factor: $\log_b(x) = \Theta(\log_c(x))$. More precisely, for any $b > 1$, $\log_b(x) = \ln(x)/\ln(b)$ where ln is the natural logarithm.

2.4 Alphabets and Sequences

Alphabets. A finite **alphabet** is simply a finite set; we mainly 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)
- The protein one-letter code {A,...,Z} \ {B,J,O,U,X,Z}; see `http://en.wikipedia.org/wiki/List_of_standard_amino_acids` for more information about the individual amino acids.
- The `HydrophObe` / hydro`PhIle` alphabet {`O`,`I`} or {`H`,`P`}
- The positive / negative charge alphabet {+, −}
- The IUPAC codes for DNA ({A, C, G, T, U, R, Y, M, K, W, S, B, D, H, V, N}) and protein ({A,...,Z} \ {J,O,U}) sequences (see `http://www.dna.affrc.go.jp/misc/MPsrch/InfoIUPAC.html`)
- The ASCII (American Standard Code for Information Interchange) alphabet, a 7-bit encoding (0–127) of commonly used characters in the American language (see `http://en.wikipedia.org/wiki/ASCII`).
- The alphanumeric subset of the ASCII alphabet {0,...,9,A,...,Z,a,...,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. By **concatenation** (also **juxtaposition**) of symbols from an alphabet, we create a **sequence** (also **string** or **word**). The **empty sequence** ε consists of no symbols. The **length** of a sequence s, written as $|s|$, is the number of symbols in it. We identify each symbol with a sequence of length 1.

We define $\Sigma^0 := \{\varepsilon\}$ (this is: the set consisting of the empty sequence; not the empty set $\emptyset$; nor the empty sequence ε!), $\Sigma^1 := \Sigma$ and $\Sigma^n := \{xa : x \in \Sigma^{n-1}, a \in \Sigma\}$ for $n \geq 2$. Thus Σ^n is the set of words of length n over Σ; such a word s is written as $s = (s_1, \ldots, s_n) = s[1]s[2]\ldots s[n]$.

We further define

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

as the set of all (resp. all nonempty) sequences over Σ.

With $\overleftarrow{s} := s[n]s[n-1]\ldots s[1]$ we denote the **reversal** of s.

We frequently use variables such as a, b, c, d for symbols from Σ, and $p, s, t, u, v, w, x, y, z$ for sequences from Σ^*.

Substrings. If $s = uvw$ for possibly empty sequences u,v,w from Σ^*, we call

- u a **prefix** of s,
- v a **substring** (**subword**) of s,
- w a **suffix** of s.

A prefix or suffix that is different from s is called **proper**. A substring v of s is called **right-branching** if there exist symbols $a \neq b$ such that both va and vb are substrings of s. Let $k \geq 0$. A ***k*-tuple**[1] (k-mer) of s is a length-k substring of s. Similarly, for $q \geq 0$, a ***q*-gram**[2] of s is a length-q substring of s.

We use the notation $t \blacktriangleleft s$ to indicate that t is a substring of s; equivalently we say that s is a **superstring** (**superword**) of t (written as $s \blacktriangleright t$).

We write $s[i \dots j] := s[i]s[i{+}1] \dots s[j]$ for the substring from position i to position j, assuming that $i \leq j$. For $i > j$, we set $s[i \dots j] := \varepsilon$. We say that $w \in \Sigma^m$ **occurs** at position i in $s \in \Sigma^n$ if $s[i + j] = w[1 + j]$ for all $0 \leq j < m$.

A set of sequences $S \subset \Sigma^*$ is called **prefix-closed** if each prefix of each $s \in S$ is also a member of S. We define **suffix-closed** analogously.

Subsequences. While a substring is a contiguous part of a sequence, a subsequence need not be contiguous. Thus if $s \in \Sigma^n$, $1 \leq k \leq n$ and $1 \leq i_1 < i_2 < \cdots < i_k \leq n$, then $s[i_1, i_2, \dots, i_k] := s_{i_1} s_{i_2} \dots s_{i_k}$ is called a **subsequence** of s. We write $t \lhd s$ to indicate that t is a subsequence of s. Equivalently, s is a **supersequence** of t, which is written as $s \rhd t$.

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, and BBB is a subsequence but not a substring.

Number of substrings and subsequences. Let $s \in \Sigma^n$.

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 and $1 + 2 + \cdots + n = \binom{n+1}{2} = (n+1) \cdot n/2$ nonempty substrings in total. 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. In any case, the number of substrings is in $\Theta(n^2)$.

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 need not be different. For example if $s = \text{AAA}$, then $s[1 \dots 2] = s[2 \dots 3] = \text{AA}$. It is an interesting problem (for which this course will provide efficient methods) to compute the

[1] The origin of the word "tuple" is easy to explain: pair, triple, quadruple, quintuple, ..., k-tuple, ... The popularity of the variable k for this purpose, however, is a mystery.

[2] "gram" is probably related to the Greek word $\gamma\rho\alpha\mu\mu\alpha$ (gramma), meaning letter. The origin of the variable q for this purpose is unknown to the authors of these lecture notes.

number of *distinct* substrings or subsequences of a given $s \in \Sigma^n$. In order to appreciate these efficient methods, the reader is invited to think about algorithms for solving these problems before continuing!

CHAPTER 3

Metrics on Sequences

Contents of this chapter: Metric. Transformation distances. p-distance. Hamming distance. Edit distances. Edit sequence. Invariance properties of edit distances. Efficient computation (dynamic programming) of edit distances. Optimal edit sequences. q-gram profile. q-gram distance. Ranking function. Unranking function. Efficient ranking of q-grams. De Bruijn graph. Maximal matches distance. Partition of a sequence. Left-to-right-partition. Filter.

3.1 Problem Motivation

The trivial method to compare two sequences is to compare them character by character: u and v are **equal** if and only if $|u| = |v|$ and $u_i = v_i$ for all $i \in [1, |u|]$. 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 ambiguities in the genetic code; for example, GCU, GCC, GCA, and GCG 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:

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

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

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

From (3.1)–(3.3), it follows that

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

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

If (3.2)–(3.4) hold and also $d(x,x) = 0$ for all x, we call d a **pseudo-metric**. It differs from a true metric by the fact that there may be $x \neq y$ with $d(x,y) = 0$. 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 something 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), (3.2), and the following stronger version of the triangle inequality:

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

It is particularly important in phylogenetics.

3.3 Transformation Distances

Different kinds of distances can be defined on different spaces. Before discussing several metrics on sequences, let us discuss other examples of metrics, the so-called transformation distances; we begin with examples.

1. On any set $\mathcal{X}$, we can define the **discrete metric**

$$d(x,y) := \begin{cases} 0 & \text{if } x = y, \\ 1 & \text{if } x \neq y. \end{cases}$$

   This is just the equal vs. unequal distinction discussed as not particularly useful above, but it satisfies all the properties that we demand from a metric.

2. Let Z be a set and $\mathcal{X} := \mathcal{P}(Z)$ its power set. Two ways to define a metric on $\mathcal{X}$ are

$$d_1(X,Y) := |X \setminus Y| + |Y \setminus X| = |(X \cup Y) \setminus (X \cap Y)| \qquad \textbf{(symmetric set distance)}$$
$$d_\infty(X,Y) := \max\{|X \setminus Y|, |Y \setminus X|\} \qquad \textbf{(set transformation distance)}$$

3. The numbers $\{1,\dots,n\}$ can be arranged in $n!$ different orders; such an arrangement π that uses each number exactly once is called a **permutation**. The set of all permutations of $\{1,\dots,n\}$ is called the **symmetric group** of order n and denoted by $\mathcal{S}_n$. Note that permutations can always be interpreted as particular sequences of length n over $\{1,\dots,n\}$. One possibility to define a metric on $\mathcal{S}_n$ that has applications in the study of genome rearrangements is the **inversion distance** or **reversal distance**. A reversal on a permutation π takes a subword of π and reverses it. Now $d(\pi,\tau)$ is defined as the minimal number of reversals required to transform π into τ.

   For example, if $\pi = (5,2,1,4,3)$ and $\tau = (1,2,3,4,5)$, we first reverse the first three numbers in π to obtain $(1,2,5,4,3)$ and then the last three numbers to obtain τ. Thus two reversals suffice. It is easy to see that no single reversal will transform π into τ; therefore $d(\pi,\tau) = 2$. Interestingly, the problem of computing this distance is known to be NP-hard.

All the above examples are particular cases of transformation distances. A **transformation distance** $d(x,y)$ on a set $\mathcal{X}$ is always defined as the minimum number of allowed operations needed to transform one element x into another y. The allowed operations characterize the distance. Of course, they must be defined in such a way that the metric properties are satisfied. To make this clear, we list the operations in each of the examples above.

discrete metric The only operation needed is a *change*, also called a *substitution*, of any element x to any other element $y \neq x$.

symmetric set distance The operations are *removal* and *addition* of an element from resp. to a set X.

set transformation distance The operations are *removal*, *addition* and *substitution* of an element.

inversion distance on permutations One might think that the only operation is a *reversal*, but in fact there are $1 + 2 + \cdots + (n-1) = \binom{n}{2}$ different types of reversals (one for each substring of length ≥ 2).

Care has to be taken that the allowed operations imply the symmetry condition of the defined metric. The triangle inequality is never a problem since the distance is defined as the *minimum* number of operations required (be sure to understand this!).

Increasing the number of allowed operations decreases the distance and vice versa. Therefore, the set transformation distance cannot be higher than the symmetric set distance.

As the above examples show, depending on the metric space $\mathcal{X}$ and the allowed operations, a transformation distance may be easy or hard to compute.

3.4 Metrics on Sequences of the Same Length

Take $\mathcal{X} = \mathbb{R}^n$ and consider two points $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$ (note that points in n-dimensional space are also sequences of length n over the infinite alphabet $\mathbb{R}$). The following functions are metrics.

$$d_1(x,y) := \sum_{i=1}^{n} |x_i - y_i| \qquad \textbf{(Manhattan distance)}$$

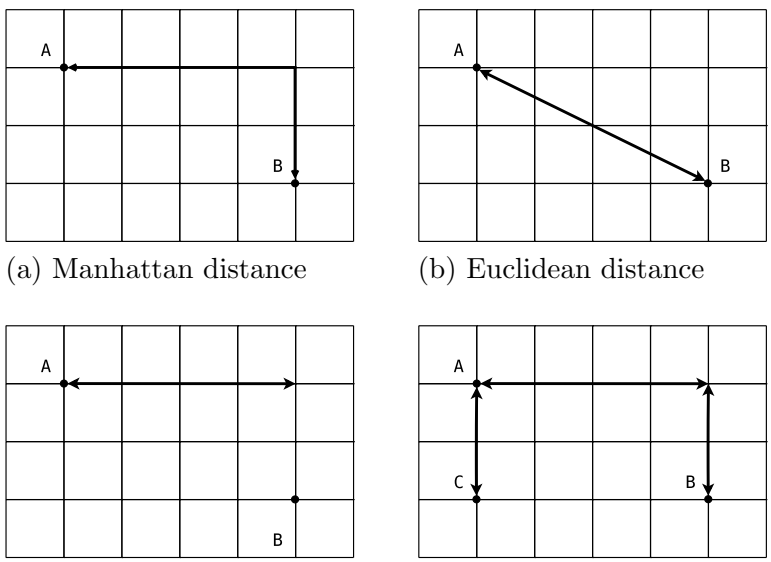
$$d_2(x,y) := \sqrt{\textstyle\sum_{i=1}^{n} |x_i - y_i|^2} \qquad \textbf{(Euclidean distance)}$$

$$d_\infty(x,y) := \max_{i=1,\ldots,n} |x_i - y_i| \qquad \textbf{(maximum metric)}$$

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

Example 3.1 Consider the location of the points A, B, and C in Figure 3.1. They have the following distances.

Manhattan distance:	$d_1(A,B) = 6$
Euclidean distance:	$d_2(A,B) = \sqrt{4^2 + 2^2} = \sqrt{20}$
Maximum metric:	$d_\infty(A,B) = \max\{4,2\} = 4$
Hamming distance:	$d_H(A,B) = 2$ because the coordinates differ in two dimensions.
	$d_H(A,C) = 1$; the coordinates differ only in one dimension.

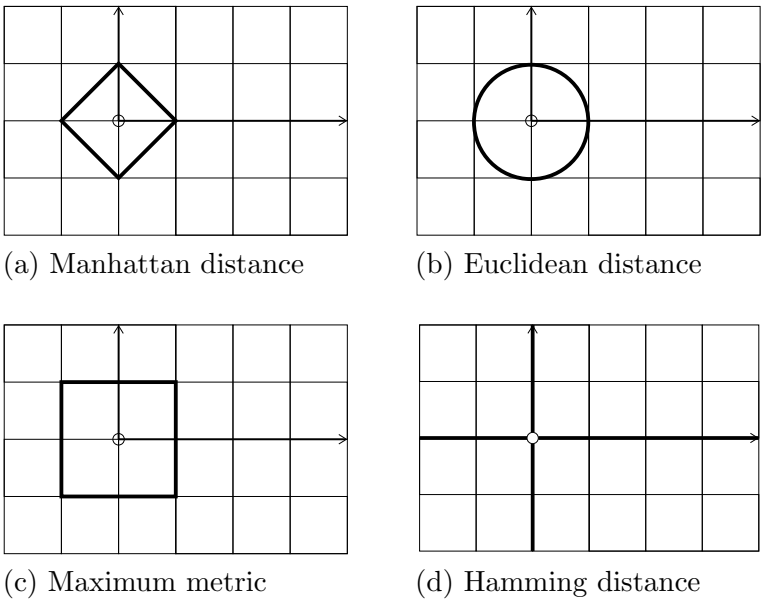


(a) Manhattan distance (b) Euclidean distance

(c) Maximum metric (d) Hamming distance

Figure 3.1: Examples of different metrics.

It is instructive to visualize (Figure 3.2) the set of points $x = (x_1, x_2) \in \mathbb{R}^2$ that have distance 1 from the origin for each of the above metrics.



(a) Manhattan distance (b) Euclidean distance

(c) Maximum metric (d) Hamming distance

Figure 3.2: Points with distance 1 for different metrics.

More generally, for every real number $p \geq 1$, the p-metric on $\mathbb{R}^n$ is defined by

$$d_p(x, y) = \Big(\sum_{i=1}^{n} |x_i - y_i|^p\Big)^{1/p}.$$

It is not trivial to verify the triangle inequality for the p-metric (it is known as the **Minkowski inequality** and verifying it is a popular exercise in first semester mathematics courses). The maximum metric is the limit case $p \to \infty$, as can be verified by some calculations. The Hamming distance can be interpreted as the p-metric for $p = 0$ (but no p-metric is defined for $0 < p < 1$).

For biological sequence data, the only distance in this section that makes general sense is the Hamming distance (there is no notion of difference or maximum between nucleotides).

Example 3.2 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 distances 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 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 $y \in \Sigma^*$. This is similar to the transformation distances considered above, with the important difference that the sequences here are processed from left to right.

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,
- $\mathtt{D}$: *delete* the next character from x (i.e., skip over it),
- $\mathtt{F}$: *flip* the next two characters in x while copying.

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 costs. If a cost is infinite, the operation is not permitted.

Name		$\mathtt{C}$	$\mathtt{S}_c$	$\mathtt{I}_c$	$\mathtt{D}$	$\mathtt{F}$
Hamming distance	d_H	0	1	∞	∞	∞
LCS distance	d_{LCS}	0	∞	1	1	∞
Edit distance	d	0	1	1	1	∞
Edit+Flip distance	d_F	0	1	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 unit-cost standard edit distance. It is the most important one in practice and should be carefully remembered. The flips are comparatively unimportant in practice.

We now formally define how $x \in \Sigma^*$ is transformed into a different sequence by an **edit sequence**, i.e., a sequence over the **edit alphabet** $\mathcal{E}(\Sigma) := \{\mathtt{C}, \mathtt{S}_c, \mathtt{I}_c, \mathtt{D}, \mathtt{F}\}$ (or the appropriate

subset of it, depending on which of the above distances we use). Thus we define the **edit function** $E : \Sigma^* \times \mathcal{E}^* \to \Sigma^*$ inductively by

$$\begin{aligned}
E(\varepsilon, \varepsilon) &:= \varepsilon \\
E(ax, \mathtt{C}\, e) &:= a\ E(x, e) && (a \in \Sigma, x \in \Sigma^*, e \in \mathcal{E}^*) \\
E(ax, \mathtt{S}_c\, e) &:= c\ E(x, e) && (a, c \in \Sigma, x \in \Sigma^*, e \in \mathcal{E}^*) \\
E(x, \mathtt{I}_c\, e) &:= c\ E(x, e) && (c \in \Sigma, x \in \Sigma^*, e \in \mathcal{E}^*) \\
E(ax, \mathtt{D}\, e) &:= E(x, e) && (a \in \Sigma, x \in \Sigma^*, e \in \mathcal{E}^*) \\
E(abx, \mathtt{F}\, e) &:= ba\ E(x, e) && (a, b \in \Sigma, x \in \Sigma^*, e \in \mathcal{E}^*)
\end{aligned}$$

For the remaining arguments, E is undefined, i.e., $E(x, \varepsilon)$ is undefined for $x \neq \varepsilon$ since there are no edit operations specified. Similarly, $E(\varepsilon, e)$ is undefined if $e \neq \varepsilon$ and $e_1 \notin \{\mathtt{I}_c : c \in \Sigma\}$, because there is no symbol that the first edit operation e_1 could operate on.

This definition can be translated immediately in a Haskell program. As an example, we compute

$$\begin{aligned}
E(\mathrm{AAB}, \mathtt{I}_{\mathrm{B}}\mathtt{CS}_{\mathrm{B}}\mathtt{D}) &= \mathrm{B}\ E(\mathrm{AAB}, \mathtt{CS}_{\mathrm{B}}\mathtt{D}) \\
&= \mathrm{BA}\ E(\mathrm{AB}, \mathtt{S}_{\mathrm{B}}\mathtt{D}) \\
&= \mathrm{BAB}\ E(\mathrm{B}, \mathtt{D}) \\
&= \mathrm{BAB}\ E(\varepsilon, \varepsilon) \\
&= \mathrm{BAB}.
\end{aligned}$$

We define the **cost** $cost(e)$ of an edit sequence $e \in \mathcal{E}^*$ as the sum of the costs of the individual operations according to the table above. We can now formally define the **edit distance** for a set of edit operations $\mathcal{E}$ as

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

Theorem 3.3 Each of the four distance variations according to the table above (Hamming distance, LCS distance, Edit distance, Edit+Flip distance) defines a metric.

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 defined as the minimum cost. 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 $cost(f) = cost(e) =: c$ such that $E(y, f) = x$.

To obtain f with the desired properties, we replace in e each $\mathtt{D}$ by an appropriate $\mathtt{I}_c$ and vice versa. Substitutions $\mathtt{S}_c$ remain substitutions, but we need to replace the substituted character from y by the correct character from x. Copies $\mathtt{C}$ remain copies and flips $\mathtt{F}$ remain flips (in the model where they are allowed). The edit sequence f constructed in this way obviously satisfies $cost(f) = cost(e)$ 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 3.4 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

1. $d(x, y) = d(\overleftarrow{x}, \overleftarrow{y})$, and
2. $d(x, y) = d(\pi(x), \pi(y))$.

Proof. Left as an exercise. Hint: Consider optimal edit sequences. □

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

3.6 An Efficient Algorithm to Compute Edit Distances

While it is easy to find some $e \in \mathcal{E}^*$ that transforms a given $x \in \Sigma^*$ into a given $y \in \Sigma^*$, it may not be obvious to find the minimum cost edit sequence. However, since x and y are processed from left to right, the following **dynamic programming** approach appears promising.

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

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

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 the standard unit cost edit distance (with substitution, insertion and deletion cost 1; flips are not permitted).

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

Theorem 3.5 For $1 \le i \le |x|$, $1 \le j \le |y|$,

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

Proof. The proof is by induction on $i + j$. The basis is given by the initialization. We may thus assume that the theorem correctly computes $D(i-1, j-1) = \min\{cost(e) : E(x[1 \dots i-1], e) = y[1 \dots j-1]\}$, so let $e_{\nwarrow}$ be an optimal edit sequence for this case. Similarly, let $e_{\uparrow}$ be optimal for $D(i-1, j) = \min\{cost(e) : E(x[1 \dots i-1], e) = y[1 \dots j]\}$ and $e_{\leftarrow}$ be optimal for $D(i, j-1) = \min\{cost(e) : E(x[1 \dots i], e) = y[1 \dots j-1]\}$.

We obtain three candidates for edit sequences to transform $x[1 \dots i]$ into $y[1 \dots j]$ as follows: (1) We extend $e_{\nwarrow}$ either by $\mathtt{C}$ if $x[i] = y[j]$ or by $\mathtt{S}_{y[j]}$ if $x[i] \neq y[j]$; the former leaves the cost unchanged, the latter increases the cost by 1. (2) We extend $e_{\uparrow}$ by $\mathtt{D}$, increasing the cost by 1. (3) We extend $e_{\leftarrow}$ by $\mathtt{I}_{y[j]}$, increasing the cost by 1.

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

It remains to be shown that there can be no other edit sequence with lower cost. Assume such an edit sequence e^* exists and consider its last operation o and its prefix e' such that $e^* = e'o$. If $o = \mathtt{D}$, then e' is an edit sequence transforming $x[1 \dots i-1]$ into $y[1 \dots j]$ with $cost(e^*) = cost(e') + 1 < D(i-1, j) + 1$ by assumption; thus $cost(e') < D(i-1, j)$, 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 e^* 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 edit sequences, $D(i, j)$ is *at most* the minimum of three values. The second part shows that the optimal edit sequence for $D(i, j)$ is always one of those three possibilities.

Together with the initialization, Theorem 3.5 can be translated immediately into an algorithm to compute the edit 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.

- An alternative is to push information forward as soon as we have obtained it. We initialize $D(0, 0) \leftarrow 0$ and $D(i, j) \leftarrow \infty$ for the remainder of the matrix. Again, we proceed row by row, or column by column or diagonal by diagonal, starting at $(0, 0)$, such that the following invariant holds. When we reach entry (i, j), it already contains the correct value. We then update the three successors as follows:

  - $D(i+1, j+1) \leftarrow \min\{D(i+1, j+1), D(i, j) + \mathbb{1}_{\{x[i+1] \neq y[j+1]\}}\}$
  - $D(i+1, j) \leftarrow \min\{D(i+1, j), D(i, j) + 1\}$
  - $D(i, j+1) \leftarrow \min\{D(i, j+1), D(i, j) + 1\}$

  Whenever we reach an entry, it has received its three updates and contains the correct value. This is called a **push-type** or **forward dynamic programming algorithm**. It seems less natural for this example, but there are other problems where this approach leads to much simpler code than the pull-type approach, so it is worth remembering.

Example 3.6 Let $x = \text{BCACD}$ and $y = \text{DBADAD}$. We compute the edit matrix D as follows:

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

Hence the edit distance of 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 last row or column, in order to calculate the next one; thus the space complexity is $O(m+n)$. The push-type algorithm has the same complexities.

Reconstructing the optimal edit sequence. 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 the optimal edit sequence. 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 $O(mn)$. Clearly, instead of storing the operations they can also be recomputed, if just the edit matrix D is stored.

To find an optimal edit sequence, let us define another 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 (for edit distance) to store any combination of the 7 possibilities.

From E, an optimal edit sequence e can be constructed backwards by **backtracing** (not to be confused with backtracking on the facing page) where we use the stored operations as a *trace* which we follow through the matrix E. Clearly e ends with one of the operations stored in $E(m, n)$, so we have determined the last character o of e. Depending on its value, we continue as follows.

- If $o = \texttt{C}$ or $o = \texttt{S}_c$ for some $c \in \Sigma$, continue with $E(m-1, n-1)$.
- If $o = \texttt{D}$, continue with $E(m-1, n)$.
- if $o = \texttt{I}_c$ for some $c \in \Sigma$, 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.

There are divide and conquer algorithms that can determine an optimal edit sequence in $O(m+n)$ space and $O(mn)$ time, using a distance-only algorithm as a sub-procedure. These algorithms (e.g. the **Hirschberg technique** discussed in Section 5.4), are very important, since space, not time, is often the limiting factor when comparing long sequences.

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** (finds systematically all solutions, not to be confused with backtracing 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 3.7 (Edit Sequence Problem) Enumerate all optimal edit sequences (with respect to standard unit cost edit distance) of two sequences $x \in \Sigma^m$ and $y \in \Sigma^n$.

Theorem 3.8 The edit sequence problem can be solved in $O(mn + z)$ time, where z is the total length of all optimal edit sequences of x and y.

Example 3.9 Let $x =$ BCACD and $y =$ DBADAD. We have seen that the standard unit cost edit distance between x and y is $d(x, y) = 4$. There are 7 edit sequences transforming x into y with this cost. Can you find them all? ■

Variations. The algorithms in this section have focused on the standard unit cost edit distance. However, the different edit distance variations from the previous section are easily handled as well; this is a good exercise.

3.7 The q-gram Distance

Edit distances emphasize the correct order of the characters in a string. If, however, a large block of a text is moved somewhere else (e.g. two paragraphs of a text are exchanged), the edit distance will be high, although from a certain standpoint, the texts are still quite similar. An appropriate notion of distance can be defined in several ways. Here we introduce the q-gram distance d_q, which is actually a pseudo-metric: There exist strings $x \neq y$ with $d_q(x, y) = 0$.

Definition 3.10 Let $x \in \Sigma^n$, let $\sigma := |\Sigma|$, and choose $q \in [1, n]$. The **occurrence count** of a q-gram $z \in \Sigma^q$ in x is the number

$$N(x, z) := |\{i \in [1, n - q + 1] : x[i \ldots (i + q - 1)] = z\}|.$$

The ***q*-gram profile** of x is a σ^q-element vector $p_q(x) := (p_q(x)_z)_{z \in \Sigma^q}$, indexed by all q-grams, defined by $p_q(x)_z := N(x, z)$.

The **q-gram distance** of $x \in \Sigma^n$ and $y \in \Sigma^m$ is defined for $1 \leq q \leq \min\{n, m\}$ as

$$d_q(x, y) := \sum_{z \in \Sigma^q} |p_q(x)_z - p_q(y)_z|.$$

Example 3.11 Consider $\Sigma = \{\text{a,b}\}$, $x = \text{abaa}$, $y = \text{abab}$, $z = \text{aaba}$, and $q = 2$. Using lexicographic order (aa, ab, ba, bb) among the q-grams, we have $p_2(x) = (1, 1, 1, 0)$, $p_2(y) = (0, 2, 1, 0)$, $p_2 = (1, 1, 1, 0)$, so $d_2(x, y) = 2$ and $d_2(x, z) = 0$ although $x \neq z$. ■

Theorem 3.12 The q-gram distance d_q is a pseudo-metric.

Proof. It fulfills nonnegativity, symmetry and the triangle inequality (exercise). □

Choice of q. In principle, we could use any $q \in [1, \min\{m, n\}]$ to compare two strings of length m and n. In practice, some choices are more reasonable than others.

For example, for $q = 1$, we merely add the differences of the single letter counts. If m and n are large, there are many (even very differently looking) strings with the same letter counts, which would all have distance zero from each other.

If on the other hand $q = n \leq m$, and $n \approx m$, the distance d_q between the strings is always close to $m - n$, which does not give us a good resolution either. Also, the q-gram profile of both strings is extremely sparse in these cases. We thus ask for a value of q such that each q-gram occurs about once (or $c \geq 1$ times) on average.

Assuming a uniform random distribution of letters, the probability that a fixed q-gram z starts at a fixed position $i \in [1, n-q+1]$ is $1/\sigma^q$. The expected number of occurrences of z in the text is thus $(n-q+1)/\sigma^q$. The condition that this number equals c leads to the condition $(n+1)/c = q/c + \sigma^q$, or approximately $n/c = \sigma^q$, since $1/c$ and q/c are comparatively small. Thus setting

$$q := \left\lfloor \frac{\log(n) - \log(c)}{\log(\sigma)} \right\rfloor$$

ensures an expected occurrence count $\geq c$ of each q-gram.

Relation to the edit distance. There is only a weak connection between unit-cost edit distance d and q-gram distance d_q. Recall that we may have $d_q(x, y) = 0$ even though the edit distance can be high. On the other hand, a single edit operation destroys up to q q-grams. Generally, one can show the following relationship.

Theorem 3.13 Let d denote the standard unit cost edit distance. Then

$$d_q(x, y)/(2q) \leq d(x, y).$$

Proof. Each edit operation locally affects up to q q-grams. Each changed q-gram can cause a change of up to 2 in d_q, as the occurrence count of one q-gram decreases, the other increases. Therefore each edit operation can lead to an increase of $2q$ of the q-gram distance in the worst case (e.g. consider AAAAAAAAAAA vs. AABAABAABAA with edit distance 3, $q = 3$, and, as can be verified, $d_3 = 18$). □

Practical computation. If the q-gram profile of a string $x \in \Sigma^n$ is dense (i.e., if it does not contain mostly zeros), it makes sense to implement it as an array $p[0 \ldots (\sigma^q - 1)]$, where $N[i]$ counts the number of occurrences of the i-th q-gram in some arbitrary but fixed (e. g. lexicographic) order.

Definition 3.14 For a finite set $\mathcal{X}$, a bijective function $r : \mathcal{X} \to \{0, \ldots, |\mathcal{X}| - 1\}$ is called **ranking function**, an algorithm that implements it is called **ranking algorithm**. The inverse of a ranking function is an **unranking function**, its implementation an **unranking algorithm**.

Of course, we are interested in an efficient (un-)ranking algorithm for the set of all q-grams over Σ. A classical one is to first define a ranking function r_Σ on the characters and then extend it to a ranking function r on the q-grams by interpreting them as base-σ numbers. There are two possibilities to number the positions of a q-gram: From q to 1 falling or from 1 to q rising, as shown in the Example 3.15.

Example 3.15 Two different possibilities to rank the q-gram $z = \text{ATACG}$:
(with $q{=}5$, $\Sigma{=}\{\text{A}, \text{C}, \text{G}, \text{T}\}$; $\sigma{=}4$, using $r_\Sigma(\text{A}) = 0$, $r_\Sigma(\text{C}){=}1$, $r_\Sigma(\text{G}){=}2$, and $r_\Sigma(\text{T}){=}3$)

$$\begin{aligned} r(\text{ATACG}) &= \underbrace{r_\Sigma(z[1]) \cdot \sigma^4}_{0 \cdot 4^4} + \underbrace{r_\Sigma(z[2]) \cdot \sigma^3}_{3 \cdot 4^3} + \underbrace{r_\Sigma(z[3]) \cdot \sigma^2}_{0 \cdot 4^2} + \underbrace{r_\Sigma(z[4]) \cdot \sigma^1}_{1 \cdot 4^1} + \underbrace{r_\Sigma(z[5]) \cdot \sigma^0}_{2 \cdot 4^0} \\ &= 0 + 192 + 0 + 4 + 2 = \mathbf{198} \end{aligned}$$

or respectively

$$\begin{aligned} r(\text{ATACG}) &= \underbrace{r_\Sigma(z[1]) \cdot \sigma^0}_{0 \cdot 4^0} + \underbrace{r_\Sigma(z[2]) \cdot \sigma^1}_{3 \cdot 4^1} + \underbrace{r_\Sigma(z[3]) \cdot \sigma^2}_{0 \cdot 4^2} + \underbrace{r_\Sigma(z[4]) \cdot \sigma^3}_{1 \cdot 4^3} + \underbrace{r_\Sigma(z[5]) \cdot \sigma^4}_{2 \cdot 4^4} \\ &= 0 + 12 + 0 + 64 + 512 = \mathbf{588} \end{aligned}$$

■

The first numbering of positions corresponds naturally to the ordering of positional number systems; in the decimal number 23, the first 2 has positional value (weight) $10^1 = 10$ whereas the 3 has the lower weight of $10^0 = 1$. For texts, however, the second numbering is more natural since we expect lower position numbers on the left side of higher position numbers.

In general, we see that $z \in \Sigma^q$ has rank

$$r(z) = \sum_{i=1}^{q} r_\Sigma\big(z[i]\big) \cdot \sigma^{q-i}, \qquad \text{resp.} \qquad r(z) = \sum_{i=1}^{q} r_\Sigma\big(z[i]\big) \cdot \sigma^{i-1}.$$

in the first resp. second position numbering system. Whichever system we use, we have to do it consistently.

The implementation efficiency of the two systems may differ slightly. Obviously, for $z \in \Sigma^q$, $r(z)$ can be computed in $O(q)$ time. Both ranking functions above have the additional advantage that when a window of length q is shifted along a text, the rank of the q-gram can be updated in $O(1)$ time: Assume that $t = azb$ with $a \in \Sigma$, $z \in \Sigma^{q-1}$, and $b \in \Sigma$. The first q-gram is az; let $j := r(az)$. Then in the first system, $r(zb) = (j \bmod \sigma^{q-1}) \cdot \sigma + r_\Sigma(b)$. In the second system, $r(zb) = \lfloor j/\sigma \rfloor + f(b)$, where $f(b) := r_\Sigma(b) \cdot \sigma^{q-1}$. We see that

the second way is possibly more efficient (as it avoids the modulo operation and only uses integer division) if f is implemented as a lookup table $f : \Sigma \to \mathbb{N}_0$. If $\sigma = 2^k$ for some $k \in \mathbb{N}$, multiplications and divisions can be implemented by bit shifting operators, e.g. in the second system, $r(zb) = (j \gg k) + f(b)$.

Now, to compute $d_q(x, y)$ for $x \in \Sigma^m$ and $y \in \Sigma^n$,

1. initialize $p_q[0 \dots (\sigma^q - 1)]$ with zeros,
2. for each $i \in [1, m - q + 1]$, increment $p_q[r(x[i..(i + q - 1)])]$,
3. for each $i \in [1, n - q + 1]$, decrement $p_q[r(y[i..(i + q - 1)])]$,
4. initialize $d := 0$,
5. for each $j \in [0, (\sigma^q - 1)]$, increment d by $|p_q[j]|$.

This procedure obviously takes $O(\sigma^q + m + n)$ time. If $m \leq n$ and q is as suggested above, this is $O(n)$ time overall.

If q is comparatively large, it does not make sense to maintain a full array. Instead, we maintain a data structure that contains only the nonzero entries of p_q. If $q = O(\log(m+n))$, we can still achieve $O(m + n)$ time using hashing.

Words with the same q-gram profile. Given $x \in \Sigma^m$ and $1 \leq q \leq m$, can we determine whether there are any $y \neq x$ with $d_q(x, y) = 0$, and if there are, how many? For $q = 1$, the problem is trivial; any permutation of the symbols gives such a y. So we shall assume that $q \geq 2$. An elegant answer can be found by constructing the Balanced De Bruijn subgraph $B(x, q)$ for x and q.

Definition 3.16 Given $x \in \Sigma^m$ and $2 \leq q \leq m$, the **Balanced De Bruijn Subgraph** $B(x, q)$ for x and q is a directed graph defined as follows:

- The vertices are all $(q - 1)$-grams of x.
- The edges correspond to the q-grams of x: For each q-gram $azb \blacktriangleleft x$ with $a \in \Sigma$, $z \in \Sigma^{q-2}$, $b \in \Sigma$, we draw an edge from the $(q - 1)$-gram vertex az to the $(q - 1)$-gram vertex zb. If the same q-gram occurs multiple times in x, the same edge also occurs multiple times in the graph.

Since $B(x, q)$ is constructed from overlapping q-grams, the graph is *connected* and *balanced* in the following sense.

Definition 3.17 A vertex in a directed graph is **balanced** if and only if its in-degree (number of incoming edges) equals its out-degree (number of outgoing edges).

A directed graph is **balanced** if either

1. all except two vertices are balanced. One of them, called the **initial vertex** or **source**, has one more outgoing edge than incoming edges. (It necessarily corresponds to the first $(q - 1)$-gram of x in $B(x, q)$.) The other one, called the **final vertex** or **sink**, has one more incoming edge than outgoing edges (and corresponds to the last $(q - 1)$-gram of x in $B(x, q)$). Or,

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

Definition 3.18 An **Eulerian path** in a graph $G = (V, E)$ is a path that uses every edge in E exactly once, according to its multiplicity.

A fundamental theorem of graph theory says that an Eulerian path exists if and only if the graph is connected and the above balance condition is satisfied. In the first case above, the path must necessarily start in the initial vertex and end in the final vertex. In the second case, the path can start anywhere but must end in the same vertex where it started.

Since $B(x, q)$ is balanced, it contains an Eulerian path, corresponding to the word x. However, it might contain more than one Eulerian path. The following observation, which follows directly from the construction of $B(x, q)$, is crucial.

Observation 3.19 There is a one-to-one correspondence between Eulerian paths in $B(x, q)$ and words with the same q-gram profile as x.

Thus other words $y \neq x$ with the same q-gram profile as x exist if and only if there is another Eulerian path through $B(x, q)$. Since such a path uses the same edges in a different order, a necessary condition is that $B(x, q)$ contains a cycle.

Theorem 3.20 The number of Eulerian paths in $B(x, q)$ equals the number of words y with $d_q(x, y) = 0$. If $B(x, q)$ contains no cycle, then there cannot exist a $y \neq x$ with $d_q(x, y) = 0$.

Note that $B(x, q)$ may contain one or more cycles, but still admit only one Eulerian path. We give some examples.

Example 3.21

1. Consider $x =$ GATTACA and $q = 4$. The graph $B(x, q)$ is shown in Figure 3.3 (left). We see that it contains no cycle, and therefore only one Eulerian path. There are no other words with the same q-gram profile.
2. Consider $y =$ GATTATTACA and $q = 4$. The graph $B(y, q)$ is shown in Figure 3.3 (center). We see that it contains a cycle, but still only one Eulerian path. There are no other words with the same q-gram profile.
3. Consider $z =$ GATTATTAATTACA and $q = 4$. The graph $B(z, q)$ is shown in Figure 3.3 (right). We see that it contains cycles and more than one Eulerian path. Another word with the same q-gram profile is $z' =$ GATTAATTATTACA.

■

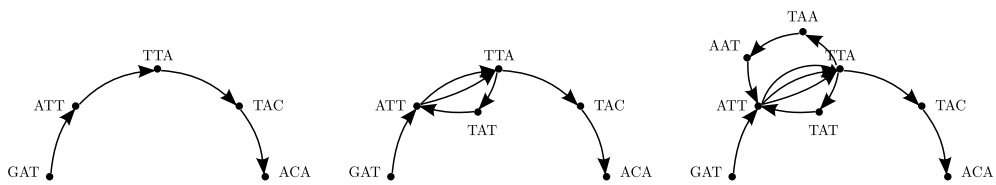


Figure 3.3: The three Balanced De Bruijn Subgraphs $B(x, 4), B(y, 4)$ and $B(z, 4)$.

One can write a computer program that constructs $B(x, q)$, identifies the initial and final vertices (if they exist), and systematically enumerates all Eulerian paths in $B(x, q)$ using backtracking.

3.8 The Maximal Matches Distance

Another notion of distance that admits large block movements is the maximal matches distance. It is based on the idea that we can cover a sequence x with substrings of another sequence (and vice versa) that are interspersed with single characters.

Definition 3.22 A **partition** of $x \in \Sigma^m$ with respect to $y \in \Sigma^n$ is a sequence $P = (z_1, b_1, \ldots, z_\ell, b_\ell, z_{\ell+1})$ for some $\ell \geq 0$ that satisfies the following conditions:

1. $x = z_1 b_1 z_2 b_2 \ldots z_\ell b_\ell z_{\ell+1}$.
2. Each z_l is a substring of y for $1 \leq l \leq \ell + 1$.
3. Each b_l is a single character for $1 \leq l \leq \ell$.

The **size of a partition** P as above is $|P| := \ell$. Note: If x is a substring of y, there exists the trivial partition $P = (x)$ of size 0.

Example 3.23 Let $x = \text{cbaabdcb}$ and $y = \text{abcba}$. The following sequences are partitions of x with respect to y, where we have marked the single characters in boldface: $P_1 = (\text{cba}, \mathbf{a}, \text{b}, \mathbf{d}, \text{cb})$ with $|P_1| = 2$, $P_2 = (\text{c}, \mathbf{b}, \text{a}, \mathbf{a}, \text{b}, \mathbf{d}, \text{cb})$ with $|P_2| = 3$ and $P_3 = (\text{cb}, \mathbf{a}, \text{ab}, \mathbf{d}, \text{cb})$ of size $|P_3| = 2$. ■

Definition 3.24 The **maximal matches distance** to x from y is defined as

$$\delta(x\|y) := \min\{|P| : P \text{ is a partition of } x \text{ with respect to } y\}.$$

Obviously δ is not symmetric: $\delta(x\|y) = 0$ is equivalent to x being a substring of y. Therefore the name *distance* is a misnomer. We can turn δ into a metric by using an appropriate symmetric function $f(\cdot, \cdot)$, i.e., defining $d_{||}(x, y) := f\big(\delta(x\|y), \delta(y\|x)\big)$. Care needs to be taken to satisfy the triangle inequality when choosing f.

Theorem 3.25 The following function is a metric on Σ^*.

$$d_{||}(x, y) := \log\big(\delta(x\|y) + 1\big) + \log\big(\delta(y\|x) + 1\big).$$

Proof. Symmetry and identity of indiscernibles are easily checked; we now check the triangle inequality. Let $x, y, u \in \Sigma^*$. The number of substrings from u needed to cover x is $\delta(x\|u)+1$. We further need $\delta(u\|y) + 1$ substrings from y to cover u. This means that we can cover x with at most $(\delta(x\|u) + 1) \cdot (\delta(u\|y) + 1)$ substrings from y, i.e.,

$$\delta(x\|y) + 1 \leq (\delta(x\|u) + 1) \cdot (\delta(u\|y) + 1) \quad \text{and (by the same argumentation)}$$
$$\delta(y\|x) + 1 \leq (\delta(y\|u) + 1) \cdot (\delta(u\|x) + 1).$$

Taking the logarithm of the product of these two inequalities, we obtain

$$\begin{aligned}&\log(\delta(x\|y)+1)+\log(\delta(y\|x)+1)\\ &\leq \log(\delta(x\|u)+1)+\log(\delta(u\|y)+1)+\log(\delta(y\|u)+1)+\log(\delta(u\|x)+1),\end{aligned}$$

which is the triangle inequality $d_{||}(x,y) \leq d_{||}(x,u) + d_{||}(u,y)$. □

Relation to the edit distance. An important property of the maximal matches distance is that it bounds the edit distance from below.

Theorem 3.26 Let d be the unit cost edit distance. Then

$$\max\{\delta(x\|y), \delta(y\|x)\} \leq d(x,y).$$

Proof. Consider a minimum-cost edit sequence e transforming x into y. Each run of copy operations in e corresponds to a substring of x that is exactly covered by a substring of y and can thus be used as part of a partition. It follows that there exists a partition (both of x with respect to y and of y with respect to x) whose size is bounded by the number of non-copy operations in e; this number is by definition equal to the edit distance. □

Efficient computation. The next question is how to find a minimum size partition of x with respect to y among all the partitions. Fortunately, this turns out to be easy: A **greedy strategy** does the job. We start covering x at the first position by a match of maximal length k such that $z_1 := x[1 \dots k]$ is a substring of y, but $z_1 b_1 = x[1 \dots (k+1)]$ is not a substring of y. We apply the same strategy to the remainder of x, $x[(k+2) \dots m]$.

Definition 3.27 The **left-to-right partition** $P_{\text{lr}}(x,y) = (z_1, b_1, \dots, z_\ell, b_\ell, z_{\ell+1})$ of x with respect to y is the partition defined by the condition that for all $l \in [1, \ell]$, $z_l b_l$ is not a substring of y. Similarly, we can define the **right-to-left partition**.

Theorem 3.28 The left-to-right partition is a partition of minimal length, i.e.,

$$|P_{\text{lr}}(x,y)| = \min\{|P| : P \text{ is a partition of } x \text{ with respect to } y\} = \delta(x\|y).$$

The same is true for the right-to-left partition.

Proof. If x is a substring of y, then the optimal partition is the left-to-right partition and has size zero. So assume that x is not a substring of y and any partition has size ≥ 1.

First, we prove the following (obvious) statement: If the optimal partition for a string x' (with respect to y) has size ℓ, and if $x = px'$ contains x' as a suffix, then the optimal partition for x cannot have size smaller than ℓ. If it did, we could restrict that smaller partition to the suffix x', and would thus obtain a smaller partition for x'.

Now we show that the above statement proves the theorem: A partition $P = (z_1, b_1, \dots)$ decomposes $x = z_1 b_1 x'$, where z_1 is a substring of y, b_1 is the first separating single character, and x' is the remaining suffix of x. The size of such a partition is at least one plus the size of an optimal partition of x'. Since the left-to-right partition induces the shortest suffix x',

by the above statement, its optimal partition is never larger than all partitions of longer suffixes.

The statement for the right-to-left partition follows since it is equal to $\overleftarrow{P_{\mathrm{lr}}}(\overleftarrow{x}, \overleftarrow{y})$. □

It remains to discuss how (and how fast) we can compute the size of a left-to-right partition. We will see later that the longest common substring z that starts at a given position i in x and occurs somewhere in y can be found in $O(|z|)$ time using an appropriate index data structure of y that ideally can be constructed in $O(|y|)$ time (e.g. a **suffix tree** of y, defined in Chapter 7). Since the substrings cover x, we obtain the following result.

Theorem 3.29 The maximal matches distance $\delta(x\|y)$ of a sequence $x \in \Sigma^m$ with respect to another sequence $y \in \Sigma^m$ can be computed in $O(m + n)$ time (using a suffix tree of y).

3.9 Filtering

The comparison of sequences is an important operation applied in several fields, such as molecular biology, speech recognition, computer science, and coding theory. The most important model for sequence comparison is the model of (standard unit cost) edit distance. However, when comparing biological sequences, the edit distance computation is often too time-consuming. A fortunate coincidence is that the maximal matches distance and the q-gram distance of two sequences can be computed in time proportional to the sum of the lengths of the two sequences ("linear time") and provide lower bounds for the edit distance. These results imply that we can use the q-gram distance or the maximal matches distance as a **filter**: Assume that our task is to check whether $d(x, y) \leq t$. Before we start the edit distance computation, we compute (for suitable q) the q-gram distance and the maximal matches distances. If we find that $d_q(x, y)/(2q) > t$ for any q, or $\delta(x\|y) > t$ or $\delta(y\|x) > t$, then we can decide quickly and correctly that $d(x, y) > t$ without actually computing $d(x, y)$, because these distances provide lower bounds. If none of the above values exceeds t, we compute the edit distance. The distinguishing feature of a filter is that it makes no errors.

On the other hand, one can develop **heuristics** that approximate the edit distance and are efficiently computable. Good heuristics will be close to the true result with high probability; hence the error made by using them is small most of the time.

CHAPTER 4

Pairwise Sequence Alignment

Contents of this chapter: Alignment. Alignment alphabet. Alignment score. Alignment graph. Global alignment. Local alignment. Linear gap costs. Affine gap costs.
Further contents in the appendix (Chapter B): Number of Alignments.

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 more visually useful is an **alignment** of x and y. We restrict our considerations to the operations of copying, substituting, inserting, and deleting characters.

We first give an example: Take $x = \text{AAB}$ and $e = \mathtt{I}_\text{B}\mathtt{CS}_\text{B}\mathtt{D}$; then $E(x, e) = \text{BAB} =: y$. 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 next 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 as follows.

$$\begin{matrix} - & \text{A} & \text{A} & \text{B} \\ \text{B} & \text{A} & \text{B} & - \end{matrix}$$

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

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

An **alignment** in general is a string over $\mathcal{A}$. A symbol $C \in \mathcal{A}$ is called a **column** of an alignment. If one of the characters of C 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. We define the bijective function $\alpha : \mathcal{E} \to \mathcal{A}$ by $\alpha(\mathtt{S}_{a,b}) := \binom{a}{b}$ for all $a, b \in \Sigma$, $\alpha(\mathtt{I}_a) := \binom{-}{a}$ and $\alpha(\mathtt{D}_a) := \binom{a}{-}$ for all $a \in \Sigma$.

Definition 4.1 Let $C = \binom{c_1}{c_2} \in \mathcal{A}$ be an alignment column. The **projection to the first row** $\pi_{\{1\}}$ and **projection to the second row** $\pi_{\{2\}}$ are defined as the function $\mathcal{A} \to \Sigma \cup \{\varepsilon\}$ with

$$\pi_{\{1\}}(C) := \begin{cases} c_1 & \text{if } c_1 \neq - \\ \varepsilon & \text{if } c_1 = - \end{cases} \qquad \pi_{\{2\}}(C) := \begin{cases} c_2 & \text{if } c_2 \neq - \\ \varepsilon & \text{if } c_2 = - \end{cases}$$

Further, let $A = (C_1 C_2 \cdots C_n)$ be an alignment. The **projection to the first row** $\pi_{\{1\}}$ and **projection to the second row** $\pi_{\{2\}}$ are then the following

$$\pi_{\{1\}}(A) := \pi_{\{1\}}(C_1)\pi_{\{1\}}(C_2)\cdots\pi_{\{1\}}(C_n) \qquad \pi_{\{2\}}(A) := \pi_{\{2\}}(C_1)\pi_{\{2\}}(C_2)\cdots\pi_{\{2\}}(C_n)$$

In other words, the first (second) projection simply reads the first (second) row of an alignment, omitting all gap characters.

Definition 4.2 Let $x, y \in \Sigma^*$. A **global alignment** of x and y is a sequence $A \in \mathcal{A}^*$ of alignment columns with $\pi_{\{1\}}(A) = x$ and $\pi_{\{2\}}(A) = y$.

Observation 4.3 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, their number is bounded by $m + n$. The maximum is reached if only insertions and deletions (indel columns) are used. On the other hand, $m = |x| = |\pi_1(A)| \leq |A|$ since the projection is never longer than the alignment. 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. □

The above observation applies to alignments based on match/mismatch and indel columns (copy/substitution and insertion/deletion operations). If additionally flips are allowed, we have to extend the alignment alphabet such that columns may also contain flipped pairs of letters ("flip columns"): $\mathcal{A} \supset \{\binom{ab}{ba} : a, b \in \Sigma, a \neq b\}$.

Definition 4.4 For the edit distance model, the **cost of an alignment column** $C \in \mathcal{A}$ is defined as the cost of the corresponding edit operation: $cost(C) := cost(\alpha^{-1}(C))$. The **cost of an alignment** $A \in \mathcal{A}^*$ is defined as the sum of its columns' cost, i.e., $cost(A) = \sum_{i=1}^{|A|} cost(A_i)$.

Definition 4.5 The **alignment cost** of two sequences $x, y \in \Sigma^*$ is defined as $d(x, y) := \min\{cost(A) : A \in \mathcal{A}^*, \pi_{\{1\}}(A) = x, \pi_{\{2\}}(A) = y\}$. The **cost-minimizing alignments** are $A^{\text{opt}}(x, y) := \{A \in \mathcal{A}^* : \pi_{\{1\}}(A) = x,\ \pi_{\{2\}}(A) = y,\ cost(A) = d(x, y)\}$.

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

Problem 4.6 (Alignment Problem) For two given strings $x, y \in \Sigma^*$ and a given cost function, find the alignment cost of x and y and one or all optimal alignment(s).

4.2 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.

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 similarity value (or **score**) to each pair that consists of a copy of the same letter and a negative score to each mismatched pair (corresponding to a substitution operation), and also to insertions and deletions.

Definition 4.7 Given $x, y \in \Sigma^*$ and an edit alphabet $\mathcal{E}$ with the score function $score(e) \in \mathbb{R}$ for each edit operation $e \in \mathcal{E}^*$ the **edit score** $s(x, y)$ is defined as

$$s(x, y) := \max\{score(e) : e \in \mathcal{E}^*,\ E(x, e) = y\}.$$

Similarly to the cost of an alignment, the score of an alignment can be defined.

Definition 4.8 Until we define more complex gap scoring models and give a sensible similarity function, the **score of an alignment column** $C \in \mathcal{A}$ is defined as the score of the corresponding edit operation: $score(C) := score(\alpha^{-1}(C))$, so the **gap score** $score(\binom{-}{a}) = score(\binom{a}{-})$ is negative and often has the same value for all $a \in \Sigma$. Its absolute value is also called **gap cost**. The **score of an alignment** $A \in \mathcal{A}^*$ is defined as the sum of its columns' scores, i.e., $score(A) = \sum_{i=1}^{|A|} score(A_i)$.

Definition 4.9 The **alignment score** of two sequences $x, y \in \Sigma^*$ is defined as $s(x, y) := \max\{score(A) : A \in \mathcal{A}^*, \pi_{\{1\}}(A) = x, \pi_{\{2\}}(A) = y\}$. The **score-maximizing alignments** are $A^{\text{opt}}(x, y) := \{A \in \mathcal{A}^* : \pi_{\{1\}}(A) = x,\ \pi_{\{2\}}(A) = y,\ score(A) = s(x, y)\}$.

4.3 The Alignment Graph

So far, we can describe the relation among two biological sequences with edit sequences and alignments. This section introduces a third way: as paths in a particular graph called the **alignment graph**.

Definition 4.10 The (global) **alignment graph** of two strings $x \in \Sigma^m$ and $y \in \Sigma^n$ is a directed acyclic edge-labeled and edge-weighted graph $G(x, y) := (V, E, \lambda, w)$ with

- vertex set $V := \{(i, j) \mid 0 \leq i \leq m,\ 0 \leq j \leq n\} \cup \{v_\circ, v_\bullet\}$,
- edge set $E \subseteq V \times V$ (edges are written in the form $u \to v$) with:
  - "horizontal" edges $(i, j) \to (i, j + 1)$ for all $0 \leq i \leq m$ and $0 \leq j < n$,
  - "vertical" edges $(i, j) \to (i + 1, j)$ for all $0 \leq i < m$ and $0 \leq j \leq n$,
  - "diagonal" edges $(i, j) \to (i + 1, j + 1)$ for all $0 \leq i < m$ and $0 \leq j < n$,
  - an "initialization" edge $v_\circ \to (0, 0)$ and a "finalization" edge $(m, n) \to v_\bullet$,
- edge labels $\lambda : E \to \mathcal{A}^*$ assigning zero, one, or several alignment columns to each edge as follows:
  - $\lambda((i, j) \to (i, j + 1)) := \binom{-}{y[j+1]}$,
  - $\lambda((i, j) \to (i + 1, j)) := \binom{x[i+1]}{-}$,
  - $\lambda((i, j) \to (i + 1, j + 1)) := \binom{x[i+1]}{y[j+1]}$,
  - $\lambda(v_\circ \to (0, 0)) := \varepsilon$, and $\lambda((m, n) \to v_\bullet) := \varepsilon$.

  Note that presently, each edge within the rectangular vertex matrix is labeled with exactly one alignment column.
- edge weight function $w : E \to \mathbb{R}$ assigning a score or cost to each edge. It is defined as the score or cost of the respective label (alignment column): $w(e) := score(\lambda(e))$ or $w(e) := cost(\lambda(e))$, where $score(\varepsilon) = cost(\varepsilon) := 0$.

Example 4.11 For unit cost edit distance, the cost weights are $w(e) = 1$ for vertical and horizontal edges, $w(e) = \mathbb{1}_{\{x[i+1] \neq y[j+1]\}}$ for the diagonal edge $e = ((i, j) \to (i + 1, j + 1))$, and $w(e) = 0$ for the edges from $v_\circ$ and to $v_\bullet$. ■

Definition 4.12 A **path** in a graph is a sequence of vertices $p = (v_0, \ldots, v_K)$ such that $(v_{k-1} \to v_k) \in E$ for all $k \in [1, K]$. The length of p is $|p| := K$. The **weight of a path** (score or cost, depending on the setting) is the sum of its edge weights, i.e., $w(p) := \sum_{k=1}^{|p|} w(v_{k-1} \to v_k)$.

Observation 4.13 Each global alignment A of x and y corresponds to a path $p(A)$ from $v_\circ$ to $v_\bullet$ in the global alignment graph $G(x, y)$. The alignment that corresponds to p is simply the concatenation of the labels on the path's edges. Furthermore, $score(A) = w(p(A))$ or $cost(A) = w(p(A))$, depending on the setting.

In particular, a maximum weight path from $v_\circ$ to $v_\bullet$ corresponds to a score-maximizing alignment of x and y.

In more detail, there is a one-to-one correspondence between alignments of $x[i' \dots i]$ with $y[j' \dots j]$ and paths from $(i'-1, j'-1)$ to (i, j). Also, there is a one-to-one correspondence between a vertex $v = (i, j)$ in an alignment graph and the position (i, j) in the corresponding alignment matrix (Section 3.6).

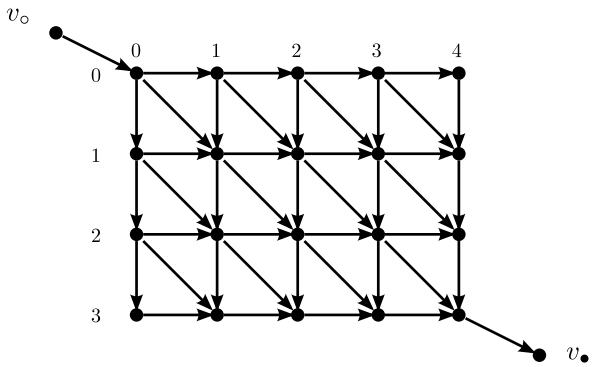


Figure 4.1: Global alignment graph for a sequence of length 3 and one of length 4. Edge weights and labels have been omitted. The initial vertex $v_\circ$ is shown at the top left corner; the final vertex $v_\bullet$ is shown at the bottom right corner.

4.4 A Universal Alignment Algorithm

The correspondence between paths in $G(x, y)$ and alignments implies that the **global alignment problem** (i.e., to find one or all optimal alignment/s) for $x, y \in \Sigma^*$ is equivalent to finding a maximum weight path from $v_\circ$ to $v_\bullet$ in $G(x, y)$.

Since the number of paths is equal to the number of alignments and therefore grows super-exponentially (see Appendix B), it is impossible to enumerate them all. We therefore use the same idea as in Theorem 3.5 and apply dynamic programming. The following presentation is given in a score context. In a cost context, we minimize costs instead of maximizing scores. The principle remains unchanged, of course.

Algorithm 4.14 (Universal Alignment Algorithm) In $G(x, y)$, we define vertex values $S : V \to \mathbb{R}$ as follows: We set

$$S(v) := \begin{cases} 0 & \text{if } v = v_\circ, \\ \max\limits_{(u \to v) \in E} \{S(u) + w(u \to v)\} & \text{if } v \neq v_\circ. \end{cases} \tag{4.1}$$

Since $G(x, y)$ is acyclic, S is well-defined, and we can arrange the computation in such an order that by the time we arrive at any vertex v, we have already computed $S(u)$ for all predecessors needed in the maximum for $S(v)$. A possible order is: $v_\circ$ first (the only vertex with no incoming edges), then row-wise through the rectangular vertex array, and finally $v_\bullet$ (the only vertex with no outgoing edges).

What is the interpretation of the value $S(v)$? The following theorem gives the answer.

Theorem 4.15 For the global alignment graph $G(x, y)$, $S(v)$ is the maximum weight of any path from $v_\circ$ to v. Therefore, if $v = (i, j)$, it is equal to the maximum alignment score of $x[1 \dots i]$ and $y[1 \dots j]$. It follows that the alignment score of x and y can be read off at $v_\bullet$, since its only predecessor is $(|x|, |y|)$:

$$s(x, y) = S\big((|x|, |y|)\big) = S(v_\bullet).$$

Proof. This is exactly the same argument as in the proof of Theorem 3.5, except that we are now maximizing scores instead of minimizing costs. □

Definition 4.16 For each edge $e = (u \to v) \in E$, by definition $S(v) \geq S(u) + w(e)$. We say that e is a **maximizing edge** if $S(v) = S(u) + w(e)$ and that a path p is a **maximizing path** if it consists only of maximizing edges. (In a cost context, we define a **minimizing edge** and a **minimizing path** similarly, thus $S(v) \leq S(u) + w(e)$.)

Backtracing. How do we find the optimal paths? As previously for the computation of the edit sequence, by **tracing back**.

When computing $S(v)$, we take note which of the incoming edges are maximizing edges. After arriving at $v_\bullet$, we trace back a maximizing path to $v_\circ$. If at some vertex we have to make a choice between several maximizing edges, we remember this branching point on a stack and first follow one choice. Later we use **backtracking** to come back and systematically explore the other choice(s). Thus, we can systematically enumerate all maximizing paths and hence construct all optimal alignments. (In a cost context, we only consider minimizing edges and trace back a minimizing path.) This procedure is entirely similar to the one described in Section 3.6. It takes $O(|A|)$ time to reconstruct an alignment A.

4.5 Alignment Types: Global, Free End Gaps, Local

A remarkable feature of our graph-based formulation of the global alignment problem is that we can now consider different variations of the alignment problem *without changing* Algorithm 4.14. In this sense, it is a Universal Alignment Algorithm. We do change the structure of the graph, though! For each variation, four things should be specified:

1. the structure of the alignment graph $G = G(x, y)$,
2. the interpretation of the vertex values $S(v)$,
3. an argument of proof that the given interpretation is correct with respect to the graph structure and the Universal Alignment Algorithm 4.14, and
4. an explicit representation of Equation (4.1), accounting for the graph structure.

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 $\gamma > 0$. We can write the universal algorithm explicitly for global alignment, and it is a good exercise to do it: Vertex $(0,0)$ has only one incoming edge from $v_\circ$, and vertices $(i,0)$ and $(0,j)$ for $i,j \geq 1$ have one incoming edge from $(i-1,0)$ and $(0,j-1)$, respectively. Vertices (i,j) for $i \geq 1$ and $j \geq 1$ have three incoming edges. Therefore we obtain the algorithm shown in Equation (4.2), known as the **global alignment algorithm** or **Needleman-Wunsch algorithm** (Needleman and Wunsch, 1970). It is important to understand that for our definition of $G(x,y)$, the recurrence in Equation (4.2) is exactly equivalent to Equation (4.1) in the Universal Alignment Algorithm (see 4.14). Also note that it is essentially the same algorithm as the one in Theorem 3.5.

$$S(v) = \begin{cases} 0 & \text{if } v = v_\circ \\ S(v_\circ) & \text{if } v = (0,0) \\ S\big((i-1,0)\big) - \gamma & \text{if } v = (i,0) \text{ for } 1 \leq i \leq |x|, \\ S\big((0,j-1)\big) - \gamma & \text{if } v = (0,j) \text{ for } 1 \leq j \leq |y|, \\ \max \left\{ \begin{array}{l} S\big((i-1,j-1)\big) + \mathit{score}(x[i],y[j]), \\ S\big((i-1,j)\big) - \gamma, \\ S\big((i,j-1)\big) - \gamma \end{array} \right\} & \text{if } v = (i,j) \text{ for } \left\{ \begin{array}{l} 1 \leq i \leq |x|, \\ 1 \leq j \leq |y| \end{array} \right\}, \\ S\big((|x|,|y|)\big) & \text{if } v = v_\bullet. \end{cases} \tag{4.2}$$

The initial case is trivial, from now on it will be omitted.

Free end gaps. If we expect that one sequence is (except for a few differences) essentially a substring of the other, or that $x = x'z$ and $y = zy'$, so that x and y have a long overlap z (but are otherwise unrelated), a global alignment makes little sense, since it would forcibly match equal symbols even in the regions where we do not expect any similarity. The culprit is the high cost to pay for gaps at either end of either sequence. We can remove those costs by changing the graph structure as follows (see Figure 4.2): In addition to the edges present in the global alignment graph, we add

- initialization edges $v_\circ \to (i,0)$ and $v_\circ \to (0,j)$ with weight zero and empty label for $1 \leq i \leq |x|$, $1 \leq j \leq |y|$,
- finalization edges $(i,|y|) \to v_\bullet$ and $(|x|,j) \to v_\bullet$ with weight zero and empty label for $1 \leq i \leq |x|$, $1 \leq j \leq |y|$.

This allows to start an alignment at the beginning of either sequence; it can start at any character in the other sequence with zero penalty. Similarly, the alignment ends at the end of either sequence. By this construction, $S\big((i,j)\big)$ can be interpreted as the maximum score of a global alignment of either $x[1 \ldots i]$ with $y[j' \ldots j]$ for any $j' \leq j$ or of $x[i' \ldots i]$ for any $i' \leq i$ with $y[1 \ldots j]$. Explicitly, the free end gap alignment algorithm looks as in Equation (4.3).

$$S(v) = \begin{cases} 0 & \text{if } v = v_\circ \\ S(v_\circ) & \text{if } v = (i,0) \text{ for } 0 \le i \le |x|, \\ S(v_\circ) & \text{if } v = (0,j) \text{ for } 0 \le j \le |y|, \\ \max \left\{ \begin{array}{l} S\big((i-1,j-1)\big) + \mathit{score}(x[i],y[j]), \\ S\big((i-1,j)\big) - \gamma, \\ S\big((i,j-1)\big) - \gamma \end{array} \right\} & \text{if } v = (i,j) \text{ for } \left\{ \begin{array}{l} 1 \le i \le |x|, \\ 1 \le j \le |y| \end{array} \right\}, \\ \max\limits_{\substack{0 \le i \le |x| \\ 0 \le j \le |y|}} \big\{ S\big((i,|y|)\big),\, S\big((|x|,j)\big) \big\} & \text{if } v = v_\bullet. \end{cases} \tag{4.3}$$

Again, it is important to understand that the recurrence is equivalent to Equation (4.1). Free end gap alignment is very important for **genome assembly**, i.e., when whole genomes are assembled from short sequenced DNA fragments: One has to determine how the fragments overlap, taking into account possible sequencing errors.

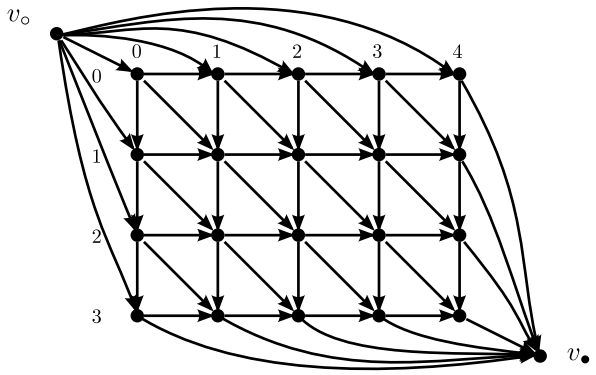


Figure 4.2: Free end gap alignment graph for a sequence of length 3 against one of length 4. The initial vertex $v_\circ$ is shown at the top left corner; the final vertex $v_\bullet$ is shown at the bottom right corner.

Approximate string matching. If one sequence, say x, is short and we are interested in the best match of x within y, we can modify global alignment or free end gap alignment in such a way that the whole of x is required to be aligned to any part of y. This is also referred to as **semi-global** or **glocal alignment** (global in the short sequence, local in the long sequence). We need the following edges in addition to those needed for a global alignment (they are a subset of those needed for free end gap alignment, see Figure 4.3):

- initialization edges $v_\circ \to (0,j)$ with weight zero and empty label for $1 \le j \le |y|$ (i.e., to the 0-th row),
- finalization edges $(|x|,j) \to v_\bullet$ with weight zero and empty label for $1 \le j \le |y|$ (i.e., from the last row).

We do not give an explicit formulation here, but come back to this problem in Section 5.2, where we present Sellers' algorithm and an improvement for this problem.

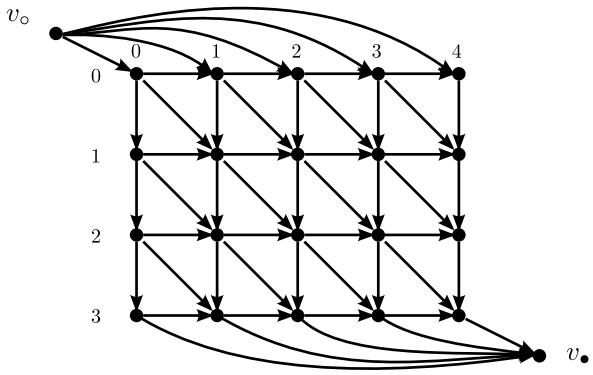


Figure 4.3: Alignment graph for the approximate string matching of a sequence of length 3 and one of length 4. The initial vertex $v_\circ$ is shown at the top left corner; the final vertex $v_\bullet$ is shown at the bottom right corner.

Local alignment. Often, two proteins are not globally similar and also do not overlap at either end. Instead, they may share one highly similar region (e.g. a conserved 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. As a local alignment can start and end anywhere, the following edges must be added to the global alignment graph (we do not show a figure, it would be too crowded):

- initialization edges $v_\circ \to (i,j)$ for all $0 \le i \le |x|$ and $0 \le j \le |y|$,
- finalization edges $(i,j) \to v_\bullet$ for all $0 \le i \le |x|$ and $0 \le j \le |y|$.

Now $S\big((i,j)\big)$ is the maximum score attainable by aligning substrings $x[i' \dots i]$ with $y[j' \dots j]$ for any $0 \le i' \le i$ and $0 \le j' \le j$. To get the best score of an alignment that ends anywhere, the finalization edges maximize over all ending positions. Explicitly written, we obtain the local alignment algorithm shown in Equation (4.4), also known as the **Smith-Waterman algorithm** (Smith and Waterman, 1981). It is one of the most fundamental algorithms in computational biology. Because of its importance, it deserves a few remarks.

$$S(v) = \begin{cases} 0 & \text{if } v = v_\circ \\ S(v_\circ) & \text{if } v = (i,0) \text{ for } 0 \le i \le |x|, \\ S(v_\circ) & \text{if } v = (0,j) \text{ for } 0 \le j \le |y|, \\ \max \left\{ \begin{array}{l} S(v_\circ), \\ S\big((i-1,j-1)\big) + \mathit{score}(x[i],y[j]), \\ S\big((i-1,j)\big) - \gamma, \\ S\big((i,j-1)\big) - \gamma \end{array} \right\} & \text{if } v = (i,j) \text{ for } \left\{ \begin{array}{l} 1 \le i \le |x|, \\ 1 \le j \le |y| \end{array} \right\}, \\ \max\limits_{\substack{0 \le i \le |x| \\ 0 \le j \le |y|}} \big\{ S\big((i,j)\big) \big\} & \text{if } v = v_\bullet. \end{cases} \tag{4.4}$$

- 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). Note that the time complexity of this algorithm is still $O(mn)$ for sequences of lengths m and n, much better than in fact globally aligning each of the $O(m^2n^2)$ pairs of substrings in $O(mn)$ time for a total time of $O(m^3n^3)$.

- 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 Appendix D.
- Even though the $O(mn)$ 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 $O(mn)$ space requirement for S and the backtracing pointers is an even more severe limitation. As mentioned previously, linear-space methods exist (they incur a small time penalty, approximately a factor of two, see Section 5.4 for details) 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 the local alignment with maximal score, but it is unable to find 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. Arslan et al. (2001) proposed an algorithm based on fractional programming with $O(mn \log m)$ running time on two sequences of lengths $m \geq n$ that reports the regions with maximum degree of similarity. In practice, this so-called **length-normalized local alignment** is only 3–5 times slower than the standard Smith-Waterman algorithm. We will discuss it in Appendix G.1.

The local alignment only makes sense for measuring with a score function. If you use a cost function the empty alignment would be always the best.

4.6 Score and Cost Variations for Alignments

Position-specific scores. The match/mismatch score in all of the above algorithms depends on a (sensible) similarity function *score* as defined in Definition A.2, often given by a log-odds score matrix. However, this does not need to be the case. The algorithm does not change in any way if indel and match/mismatch scores depend also on the *position* in the sequences. Therefore we can replace $score(x_i, y_j)$ by $score(i, j)$ and worry later about where to obtain reasonable score values.

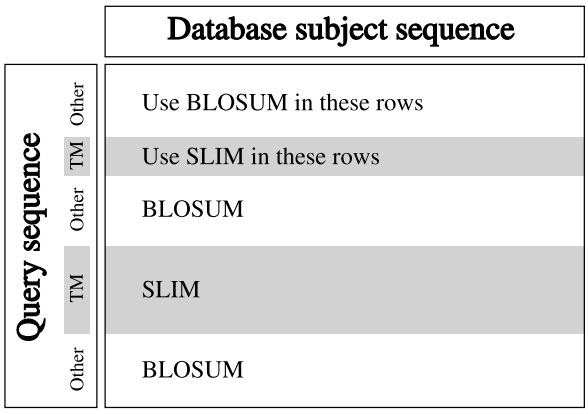


Figure 4.4: Bipartite scoring scheme for detection of homologous transmembrane proteins due to Müller et al. (2001). The figure represents the Smith-Waterman alignment matrix and indicates which scoring matrix is used for which query positions (rows): In transmembrane helices (TM), the transmembrane-specific scoring matrix SLIM is used, elsewhere the general-purpose matrix BLOSUM.

A useful application of this observation is as follows: Local alignment is often used in large-scale database searches to find remote homologs of a given protein. Transmembrane (TM) proteins have an unusual sequence composition (as pointed out in Section A.3). According to Müller et al. (2001), one can increase the sensitivity of the homology search, by using a bipartite scoring scheme with a (non-symmetric) transmembrane helix specific scoring matrix (such as SLIM) for the TM helices and a general-purpose scoring matrix (such as BLOSUM) for the remaining regions of the query protein; see Figure 4.4. As a consequence, the score of aligning x_i with y_j does not only depend on the amino acids x_i and y_j themselves, but also on the position i within the query sequence x.

General gap costs. So far, a gap of length d in any sequence receives a score of $-d \cdot \gamma$, 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 d consecutive gap characters as one long gap or as d gaps of length 1. This is the case of **linear gap costs**. In general a function $g : \mathbb{R} \to \mathbb{R}$ is *linear* if $g(k + l) = g(k) + g(l)$ and $g(\lambda x) = \lambda g(x)$.

When indels occur, 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 $g(l)$ 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., $g(k + l) \leq g(k) + g(l)$ for all $k, l \geq 0$. 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 edges in the alignment graph and in the time complexity of the algorithm. The changes

apply to all of the above variations (global, free end gap, glocal, and local alignment). To the appropriate $G(x, y)$ we add

- horizontal edges $(i, j') \to (i, j)$ for all $0 \leq i \leq |x|$ and all pairs $0 \leq j' < j \leq |y|$ with respective weights $g(j - j')$ and labels $\left(\frac{\cdots}{y[j'+1\ldots j]}\right)$,
- vertical edges $(i', j) \to (i, j)$ for all $0 \leq j \leq |y|$ and all pairs $0 \leq i' < i \leq |x|$ with respective weights $g(i - i')$ and labels $\left(\frac{x[i'+1\ldots i]}{\cdots}\right)$.

Since each vertex has now $O(m + n)$ predecessors, the running time of Algorithm 4.14 increases to cubic ($O(mn(m + n))$ for sequences of lengths m and n). In Equation (4.5) we show explicitly the Smith-Waterman algorithm with general gap costs $g : \mathbb{N} \to \mathbb{R}_0^+$.

$$S(v) = \begin{cases} 0 & \text{if } v = (i, 0) \text{ for } 0 \leq i \leq |x|, \\ 0 & \text{if } v = (0, j) \text{ for } 0 \leq j \leq |y|, \\ \max \left\{ \begin{array}{l} 0, \\ S\big((i-1, j-1)\big) + \mathit{score}(x[i], y[j]), \\ \max_{0 \leq i' < i} \left\{ S\big((i', j)\big) - g(i - i') \right\}, \\ \max_{0 \leq j' < j} \left\{ S\big((i, j')\big) - g(j - j') \right\} \end{array} \right\} & \text{if } v = (i, j) \text{ for } \left\{ \begin{array}{l} 1 \leq i \leq |x|, \\ 1 \leq j \leq |y| \end{array} \right\}, \\ \max\limits_{\substack{0 \leq i \leq |x| \\ 0 \leq j \leq |y|}} \left\{ S\big((i, j)\big) \right\} & \text{if } v = v_\bullet. \end{cases} \tag{4.5}$$

Although this starts to look intimidating, remember that this formula is still nothing else than Equation (4.1). 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 $0 < e \leq d$ is called the **gap extension cost**. The gap open cost is paid once for every consecutive run of gaps, including 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-time algorithm ($O(mn)$ time) exists; the idea is due to Gotoh (1982). We explain it for global alignment; the required modifications for the other alignment types are easy.

Recall that $S\big((i, j)\big)$ 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:

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

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

Then

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

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 $V\big((i,j)\big)$: 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 $V\big((i,j)\big)$ as $V\big((i,j)\big) = S\big((i-1,j)\big) - d$. 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 $(i-1,j)$, $V\big((i,j)\big) = V\big((i-1,j)\big) - e$. Together, we see that for $1 \le i \le m$ and $0 \le j \le n$,

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

Similarly, for horizontal gaps we obtain for $0 \le i \le m$ and $1 \le j \le n$,

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

The border elements are initialized in such a way that they do not contribute to the maximum in the first row or column, for example:

$$V\big((0,j)\big) = H\big((i,0)\big) = -\infty.$$

An analysis of this algorithm reveals that it uses $O(mn)$ time and space. Hidden in the O-notation is a larger constant factor compared to the case of linear gap costs. Note that, if only 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 S (or a matrix of back-pointers) 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.

The above affine gap cost formulas can also be derived as a special case of the Universal Alignment Algorithm 4.14. However, we need three copies of the rectangular vertex array, corresponding to S, V, and H.

Example 4.17 Given amino acid strings WW and $WNDW$ and the following scoring scheme, find the optimal global alignment. For affine gap costs: Gap open cost of $d = 11$, gap extension cost of $e = 1$ and the substitution scores are taken from the BLOSUM62 matrix ($score(\mathrm{W},\mathrm{W}) = 11$, $score(\mathrm{W},\mathrm{N}) = -4$, $score(\mathrm{W},\mathrm{D}) = -4$).

Figure 4.5 shows an alignment graph for global alignment with affine gap costs.

The initial ($v_\circ$) and final ($v_\bullet$) vertices are omitted. The thick edges mark the path to the optimal score. The alignment is $\binom{W-_d-_eW}{WNDW}$ where "$-_d$" is the start of a gap and "$-_e$" the extension. ■

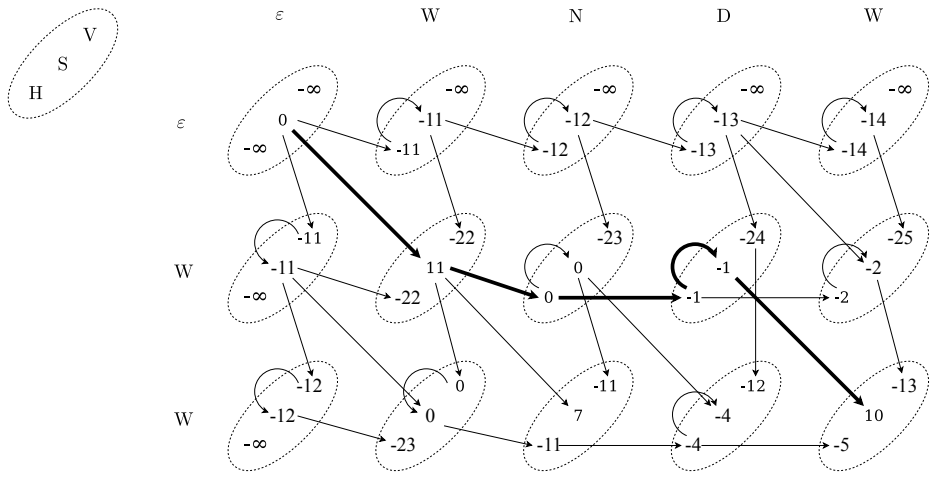


Figure 4.5: An alignment graph for a global alignment with affine gap costs. The score values for V, S and H are in the nodes as indicated on the left side. The edges pointing at each part of a vertex originate at the score-maximizing predecessor vertex or vertex part.

CHAPTER 5

Advanced Topics in Pairwise Alignment

Contents of this chapter: Suboptimal alignments, approximate string matching, Sellers' algorithm, Ukkonen's cutoff improvement, forward-backward technique, pairwise alignment in linear space, Hirschberg.
Further contents in the appendix (Chapter G): Length-normalized alignment, shadow effect, mosaic effect, optimal normalized alignment score, parametric alignment, ray search problem.

5.1 Suboptimal Alignments

Often it is desirable not only to find one or all *optimal* alignments of two sequences, but also to find **suboptimal alignment**s, i.e., alignments that score (slightly) below the optimal alignment score. This is especially relevant in a local alignment context where several similar regions might exist and be of biological interest, while the Smith-Waterman Algorithm reports only a single (optimal) one. Also, above we have already argued that in some cases a length-normalized alignment score may be more appropriate, giving us another reason to look at more than a single optimal alignment.

A simple way to find suboptimal local alignments could be to report first the alignment (obtained by backtracing) corresponding to the highest score in the local alignment graph, then the alignment corresponding to the second highest score, and so on. This approach has two disadvantages, though:

1. Redundancy: Many of the local alignments reported by this procedure will be largely overlapping, differing only by the precise placement of single gaps or the exact end characters of the alignments.

2. Shadowing effect: Some high scoring alignments will be invisible in this procedure if an even higher scoring alignment is very close by, for example if the two alignments cross each other in the edit graph, where always the higher scoring prefix will be chosen.

In order to circumvent these disadvantages, Waterman and Eggert (1987) gave a more sensible definition of suboptimal local alignments.

Definition 5.1 An alignment is a **nonoverlapping suboptimal local alignment** if it has no match or mismatch in common with any higher-scoring (sub)optimal local alignment. (Two alignments A and A' have a match or mismatch in common (they overlap) if there is at least one pair of positions (i, j) such that $x[i]$ is aligned with $y[j]$ both in A and in A'.)

Algorithmically, nonoverlapping alignments can be computed as follows: First the Smith-Waterman algorithm is run as usual, returning an optimal local alignment. Then, the matrix S is computed again, but any match or mismatch from the reported optimal alignment is forbidden, i.e., in the alignment graph the corresponding edges are removed. In terms of computing the alignment matrix, in the maximization for these cells only a subset of the four possible cases is considered.

This could be done by re-computing the whole matrix, but it is easy to see that it suffices to compute only that part of the matrix from the starting point of the first alignment to the bottom-right of the matrix. Moreover, whenever in a particular row or column the entries in the re-computation do not change compared to the entries in the original matrix, the computation along this row or column can be stopped. Waterman and Eggert suggest to alternate the computation of one row and one column at a time, each one until the new value coincides with the old value.

This procedure can be repeated for the third-best nonoverlapping local alignment, and so on.

If we assume the usual type of similarity function with expected negative score of a random symbol pair so that large parts of the alignment matrix are filled with zeros, then the part of the matrix that needs to be re-computed is not much larger than the alignment itself.

Then, to compute the K best nonoverlapping suboptimal local alignments, the algorithm takes an expected time of $O(mn + \sum_{k=1}^{K} L_k^2)$, where L_k is the length of the k-th reported alignment. In the worst case this is $O(mnK)$, but typically much less in practice.

Huang and Miller (1991) showed that the Waterman-Eggert algorithm can also be implemented for affine gap costs and in linear space.

5.2 Approximate String Matching

We have already discussed a variant of the Universal Alignment Algorithm that can be used to find a *best* (highest-scoring) approximate match of a (short) pattern $x \in \Sigma^m$ within a (long) text $y \in \Sigma^n$. Often, we are interested in *all* matches above a certain score (or below a certain cost) threshold. A simple modification makes this possible: We simply disregard the final vertex $v_\bullet$ and look at the last row of the S-matrix.

In this section, we come back to the cost-based formulation (and a D-matrix). We assume that a sensible cost function $cost : \mathcal{A} \to \mathbb{R}_0^+$ is given such that all indel costs have the same value $\gamma > 0$.

Definition 5.2 Given a pattern $x \in \Sigma^m$, a text $y \in \Sigma^n$, a sensible cost function $cost : \mathcal{A} \to \mathbb{R}_0^+$ with $cost(\binom{-}{a}) = cost(\binom{a}{-}) = \gamma > 0$ for all $a \in \Sigma$, and a threshold $k \geq 0$, then the **approximate string matching problem** consists of finding all text substrings $y' \blacktriangleleft y$ such that $d(x, y') \leq k$, where $d(\cdot, \cdot)$ denotes edit distance with respect to the given cost function.

In fact, it suffices to discover the ending positions j of the substrings y' because we can always reconstruct the whole substring and its starting position by backtracing.

For a change in style, this time we present an explicit formulation of an algorithm that solves the problem not as a recurrence, but in pseudo-code notation. Algorithm 5.1 is known as **Sellers' algorithm** (Sellers, 1980). Of course, it corresponds to the Universal DP variant discussed in the previous section for approximate string matching, except that we do not look at the final vertex $v_\bullet$ to find the best match, but look at all the vertices in the last row to identify all matches with $D(m, j) \leq k$ for $0 \leq j \leq n$.

Algorithm 5.1 Sellers' algorithm for a general cost function. $D(i, j)$ is the minimum cost to align $x[1 \ldots i]$ to $y[j' \ldots j]$ for any $j' \leq j$.

Input: pattern $x \in \Sigma^m$, text $y \in \Sigma^n$, threshold $k \in \mathbb{N}_0$,
cost function *cost* with indel cost $\gamma > 0$, defining an edit distance $d(\cdot, \cdot)$
Output: ending positions j such that $d(x, y') \leq k$,
where $y' := y[j' \ldots j]$ for some $j' \leq j$

```
▷ Initialize 0-th column of the edit matrix D:
for i ← 0 ... m do
    D(i, 0) ← i · γ
▷ Proceed column-by-column:
for j ← 1 ... n do
    D(0, j) ← 0
    for i ← 1 ... m do
        D(i, j) ← min{D(i − 1, j − 1) + cost(x[i], y[j]), D(i, j − 1) + γ, D(i − 1, j) + γ}
    if D(m, j) ≤ k then
        report (j, D(m, j)) ▷ report match ending at column j and its cost
```

Looking closely at Algorithm 5.1, we see that each iteration $j = 1 \ldots n$ transforms the previous column $D(\cdot, j-1)$ of the edit matrix into the current column $D(\cdot, j)$, based on character $y[j]$. We can formalize this as follows.

Definition 5.3 (The *NextColumn* Function) For a fixed pattern x of length $m \geq 0$, define function $NextColumn : \left(\mathbb{R}_0^+\right)^{m+1} \times \Sigma \to \left(\mathbb{R}_0^+\right)^{m+1}$, $(d, a) \mapsto d^+$, where $d = (d_0, \ldots, d_m)$ and $d^+ = (d_0^+, \ldots, d_m^+)$ are column vectors, as follows:

$$d_0^+ := 0;$$
$$d_i^+ := \min\left\{d_{i-1} + cost(x[i], a),\ d_i + \gamma,\ d_{i-1}^+ + \gamma\right\} \qquad \text{for } i \geq 1.$$

Let $D_j := (D(i,j))_{0 \le i \le m}$ denote the j-th column of D for $0 \le j \le n$. Then Sellers' algorithm effectively consists of repeatedly computing $D_j \leftarrow NextColumn(D_{j-1}, y_j)$ for each $j = 1 \dots n$.

Some remarks about Sellers' algorithm:

1. Since we only report end positions and costs, there is no need to store back pointers and the entire edit matrix D in Algorithm 5.1. The current and the previous column suffice. This decreases the memory requirement to $O(m)$, where m is the (short) pattern length.

2. If we have a very good match with costs $\ll k$ at position j, the neighboring positions may also match with cost $\le k$. To avoid this redundancy, it makes sense to post-process the output and only look for **runs of local minima**. Without formally defining it, if $k = 3$ and the respective costs at positions $j-2, \dots, j+2$ were $(2, 3, 1, 1, 2)$, all of these positions would be reported. We are only interested in the locally minimal value 1, which occurs as a run of length 2. So it would make sense to report the run interval and the minimum cost as $([j, j+1], 1)$.

3. Note that in fact we do not need to know the exact value of $D(m, j)$ as long as we can be sure that it exceeds k and therefore will not be reported.

The last observation leads to a considerable improvement of Sellers' algorithm: If we know that in column j all values after a certain row i_j^* exceed k, we do not need to compute them (e.g. we could assume that they are all equal to ∞ or $k+1$). The following definition captures this formally.

Definition 5.4 The **last essential index** i^* (for threshold k) of a vector $d \in \mathbb{R}^{m+1}$ is defined as $i^*(d) := \max\{i : d_i \le k\}$. Two vectors $d, d' \in \mathbb{R}^{m+1}$ are ***k*-equivalent** if and only if $i^*(d) = i^*(d')$ and $d_i = d'_i$ for all $i \le i^*(d)$. We write this as $d \equiv_k d'$.

Thus the last essential index i_j^* of column D_j is $i_j^* := i^*(D_j) = \max\{i : D(i,j) \le k\}$. From the definition it is clear that we cannot miss any k-approximate matches if we replace a column of the edit matrix by any k-equivalent vector (e.g. by not computing the cells below the last essential index and assuming a value of $k+1$).

The last essential index i_0^* of the 0-th column is easily found, because the scores increase monotonically from 0 to $m \cdot \gamma$. The other columns, however, are not necessarily monotone! Therefore, as we move column-by-column through the matrix, repeatedly invoking the $NextColumn$ function, we have to make sure that we can efficiently find the last essential index of the current column, given the previous column. The following lemma is the key.

Lemma 5.5 The property of k-equivalence is preserved by the $NextColumn$ function. In other words: Let $d, d' \in \mathbb{N}_0^{m+1}$. If $d \equiv_k d'$, then $NextColumn(d, a) \equiv_k NextColumn(d', a)$ for all $a \in \Sigma$.

Proof. Let $e := NextColumn(d, a)$ and $e' := NextColumn(d', a)$. By induction on i, we show that $e_i \le k$ or $e'_i \le k$ implies $e_i = e'_i$. Since $e_0 = e'_0 = 0$, this certainly holds for $i = 0$.

Now we show that the statement holds for each $i > 0$, assuming that it holds for $i - 1$. First, assume that $e_i \leq k$. We shall show that $e_i = e'_i$ from the recurrence $e_i = \min\{d_{i-1} + cost(x[i], a),\ d_i + \gamma, e_{i-1} + \gamma\}$.

Since both $cost(x[i], a) \geq 0$ and $\gamma \geq 0$, we know that the minimizing predecessor (be it d_{i-1}, d_i, or e_{i-1}) can be at most k. Therefore it must equal its value in the k-equivalent vector (non-equal entries are by definition larger than k). Therefore we obtain the same minimum when we compute $e'_i = \min\{d'_{i-1} + cost(x[i], a),\ d'_i + \gamma, e'_{i-1} + \gamma\}$, and hence $e_i = e'_i$.

Similarly, we show that $e'_i \leq k$ implies $e'_i = e_i$. □

How can we use the above fact to our advantage? Consider the effects of the d-entries below the last essential index on the next column. Since $d_i > k$ for all $i > i^*$ and costs are nonnegative, these cells provide candidate values $> k$ for e_i for $i > i^* + 1$. Therefore, the only chance that such e_i remain bounded by k is vertically, i.e., if $e_i = e_{i-1} + \gamma$. As soon as $e_i > k$ for some $i > i^* + 1$, we know that the last essential index of e occurs before that i. Algorithm 5.2 shows the details.

Algorithm 5.2 Improved cutoff-variant of Sellers' algorithm, maintaining the last essential index i^*_j of each column

Input: pattern $x \in \Sigma^m$, text $y \in \Sigma^n$, threshold $k \in \mathbb{N}_0$,
cost function *cost* with indel cost $\gamma > 0$, defining an edit distance $d(\cdot, \cdot)$
Output: ending positions j such that $d(x, y') \leq k$,
where $y' := y[j' \ldots j]$ for some $j' \leq j$

```
▷ Initialize 0-th column of the edit matrix D:
for i ← 0 ... m do
    D(i, 0) ← i · γ
i*_0 ← ⌊k/γ⌋
▷ Proceed column-by-column:
for j ← 1 ... n do
    D(0, j) ← 0
    i⁺ ← min{m, i*_{j−1} + 1}
    for i ← 1 ... i⁺ do
        D(i, j) ← min{D(i − 1, j − 1) + cost(x[i], y[j]), D(i, j − 1) + γ, D(i − 1, j) + γ}
    if D(i⁺, j) < k then
        i*_j ← min{m, i⁺ + ⌊(k − D(i⁺, j))/γ⌋}
        for i ← (i⁺ + 1) ... i*_j do
            D(i, j) ← D(i − 1, j) + γ
    else
        i*_j ← max{i ∈ [0, i⁺ − 1] : D(i, j) ≤ k}
    if i*_j = m then
        report (j, D(m, j)) ▷ report match ending at column j and its and cost
```

Example 5.6 Cutoff-variant of Sellers' Algorithm: Let $S = \text{AABB}$, $T = \text{BABAABABB}$ and $k = 1$. The following table shows which values are computed considering unit cost. The last essential index in each column is encircled. The k-approximate matches are underlined in the last row.

S \ T		B	A	B	A	A	B	A	B	B
	0	0	0	0	0	0	0	0	0	0
A	①	①	0	1	0	0	1	0	1	1
A		2	①	1	①	0	1	1	1	2
B				①	2	①	0	1	1	1
B					2		$\underline{①}$	$\underline{①}$	$\underline{①}$	$\underline{①}$

Four exemplary alignments, one ending at each position in T, are shown below.

I:		II:		III:		IV:	
S:	AABB	S:	AABB	S:	AAB–B	S:	AABB
T:	AAB–	T:	AABA	T:	AABAB	T:	BABB

■

Sometimes, because it was first published in Ukkonnen (1985), this algorithm is referred to as **Ukkonen's cutoff algorithm**, although it is more common to use this name if the cost function is standard unit cost. In that case, even further optimizations are possible, but we do not discuss the details here.

5.3 The Forward-Backward Technique

The following problem frequently arises as a sub-problem of more advanced alignment methods, that is why we discuss it beforehand in detail:

Problem 5.7 (Advanced Alignment Problem) Given two sequences s and t of lengths m and n, respectively, find their optimal global alignment under the condition that the alignment path passes through a given node (i_0, j_0) of the alignment graph.

This is not a new problem at all! In fact, we can decompose it into two standard global alignment problems: Aligning the prefixes $s[1 \dots i_0]$ and $t[1 \dots j_0]$, and aligning the suffixes $s[i_0 + 1 \dots m]$ and $t[j_0 + 1 \dots n]$.

A little more interesting is the case when we need to solve this problem for *every* point (i, j) *simultaneously.*

Fortunately, we can use the "forward-backward" technique. In addition to the usual alignment graph with alignment costs $D(i, j)$ (which record the minimally attainable cost for a prefix alignment), we additionally define the **reverse alignment graph**: We exchange initial and final vertex and reverse the direction of all edges. Effectively, we are thus globally aligning the reversed sequences. Hence, the associated alignment cost matrix $D^{rev} = (D^{rev}(i, j))$ records the maximally attainable scores for the suffix alignments. By defining $T(i, j) := D(i, j) + D^{rev}(i, j)$, we obtain the minimally attainable total cost of all paths that go through (i, j). Clearly matrices D, D^{rev}, and T thus can be computed in $O(mn)$ time.

For every (i, j), we can now obtain the **additional cost** $C(i, j)$ for passing through (i, j), compared to staying on an optimal alignment path. By definition, if (i, j) is on the path of an optimal alignment, this value is zero, otherwise it is positive; see Figure 5.1.

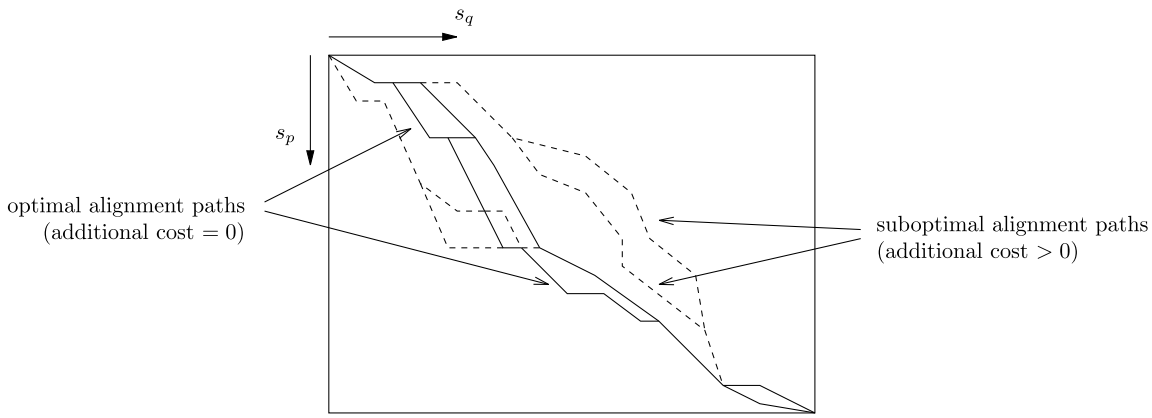


Figure 5.1: An additional cost matrix. Entry $C(i, j)$ contains the cost of the best possible alignment that passes through vertex (i, j) of the alignment graph.

Definition 5.8 Given two sequences s and t of lengths m and n, respectively, their **additional cost matrix** $C = (C(i, j))_{0 \le i \le m, 0 \le j \le n}$ is defined by

$$C(i,j) \quad = \quad \min \left\{ D(A +\!\!+ B) \;\middle|\; \begin{array}{l} A \text{ is an alignment of the prefixes} \\ \quad s[1 \ldots i] \text{ and } t[1 \ldots j] \\ B \text{ is an alignment of the suffixes} \\ \quad s[i+1 \ldots m] \text{ and } t[j+1 \ldots n] \end{array} \right\} - d(s,t)$$

where "$+\!\!+$" denotes concatenation of alignments. The **score loss matrix** is defined similarly in terms of maximal score.

For any additive alignment cost, where $cost(A +\!\!+ B) = cost(A) + cost(B)$, we have that $C(i, j) = T(i, j) - d(s, t)$.

Note that $T(0, 0) = T(m, n)$ is the optimal global alignment cost $d(s, t)$ (since every path, especially the optimal one, passes through $(0, 0)$), and so $T(i, j) \ge T(0, 0)$ for all (i, j). The quantity $T(i, j) - T(0, 0)$ thus is equal to the additional cost of vertex (i, j) in comparison to the optimal path. In similarity terms, the signs are reversed and $T(0, 0) - T(i, j)$ is the score loss of going through node (i, j) in comparison to the optimal path.

Thus the additional cost matrix (or score loss matrix) can be computed in $O(mn)$ time, too.

Example 5.9 Consider the sequences $s = CT$ and $t = AGT$. For unit cost, the edit matrix and reverse edit matrix are

D :

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

D^{rev} :

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

where D^{rev} is drawn in reverse direction), resulting in the following additional cost matrix:

$$C: \quad \begin{array}{c|cccc|} & & A & G & T \\ \hline & 0 & 0 & 1 & 3 \\ \text{C} & 1 & 0 & 0 & 2 \\ \text{T} & 3 & 2 & 1 & 0 \\ \hline \end{array}$$

We see that the paths of the two optimal alignments $A_1^{opt} = \begin{pmatrix} - & C & T \\ A & G & T \end{pmatrix}$ and $A_2^{opt} = \begin{pmatrix} C & - & T \\ A & G & T \end{pmatrix}$ correspond to the 0-entries in C. ■

Affine gap costs. For affine gap costs (see Section 4.6) the computation of the total or additional cost matrix is slightly more difficult: Here, the cost of a concatenated alignment is not always the sum of the costs of the individual alignments because gaps might be merged, so that the gap initiation penalty that is imposed twice in the two separate alignments must be counted only once in the concatenated alignment.

However, since the efficient computation of affine gap costs uses the two history matrices V and H (as defined in Section 4.6), and we know that by definition the costs stored in these matrices refer to those alignments that end with gaps, the additional cost matrix for affine gap costs can also be computed in quadratic time if the history matrices V and H and their reverse counterparts V^{rev} and H^{rev} are given:

$$T(i,j) \quad = \quad \min \left\{ \begin{array}{l} D(i,j) + D^{rev}(i,j) \\ V(i,j) + V^{rev}(i,j) - \text{gapinit} \\ H(i,j) + H^{rev}(i,j) - \text{gapinit} \end{array} \right\}.$$

Applications. The forward-backward technique is used, for example, in the following problems:

- linear-space alignment (Hirschberg technique, see also Section 5.4),
- discovering the regions of slightly sub-optimal alignments (those cells where the score gap, resp. additional cost remains below a small tolerance parameter),
- computing regions for the Carrillo-Lipman heuristic for multiple sequence alignment (see Section 11.2),
- finding cut-points for divide-and-conquer multiple alignment (see Section 11.4).

5.4 Pairwise Alignment in Linear Space

In Chapter 4 we have seen that the *optimal alignment score* of two sequences s and t of lengths m and n, respectively, as well as an *optimal alignment* can be computed in $O(mn)$ time and $O(mn)$ space.

Even though the $O(mn)$ time required to compute an optimal global or local alignment is sometimes frowned upon, it is not the main bottleneck in sequence analysis. The main

bottleneck so far is the $O(mn)$ space requirement. Think of a matrix of traceback-pointers for two bacterial genomes of 5 million nucleotides each. That matrix contains $25 \cdot 10^{12}$ entries and would thus need 25 Terabytes of memory, an impossibility in the year 2010. Fortunately, there is a solution (that is, a different one than waiting another 20 years for huge amounts of cheap memory).

We have already seen that the score by itself can easily be computed in $O(m + n)$ space, since we only need to store the sequences and one row or column of the edit matrix at a time. However, if we also want to output an optimal alignment, the whole edit matrix or a related matrix containing back-pointers is required for the backtracing phase.

In this section we present an algorithm that allows to compute an optimal *global* alignment in linear space. We discuss the case of *local* alignment below. Our presentation is for the simple case of homogeneous linear gap costs $g(\ell) = \ell \cdot \gamma$, where γ is the cost for a single gap character. However, it can be generalized for affine gap costs with some additional complications.

The algorithm works in a **divide-and-conquer** manner, where in each recursion the edit matrix $S = (S(i, j))$ is divided into an upper half and a lower half at its middle row $m' = \lceil m/2 \rceil$. The upper part of the edit matrix is computed as usual. The computation is continued in the lower part in the same way, but with the addition that for each cell $S(i, j)$ we additionally store a back-pointer matrix $M = (M(i, j))$, where $m' < i \leq m$, $0 \leq j \leq n$. Entry $M(i, j)$ points to a vertex of the middle row, where an optimal alignment ending in cell (i, j) passes through, i.e., $M(i, j) = n'$ means that an optimal global alignment ending at (i, j) passes through (m', n'). We only need one row of M at a time, since the pointer to row m' can be updated as we proceed through the edit matrix.

In the end, we find in $M(m, n)$ a pointer to a vertex of the middle row where an optimal global alignment of the two sequences s and t passes through. In other words, $(\lceil m/2 \rceil, M(m, n))$ is a vertex on an optimal alignment path.

The procedure is called recursively, once for the upper left "quarter", and once for the lower right "quarter" of the edit graph. The recursion is continued until only one row remains, in which case the problem can easily be solved directly. See Figure 5.2 for an illustration.

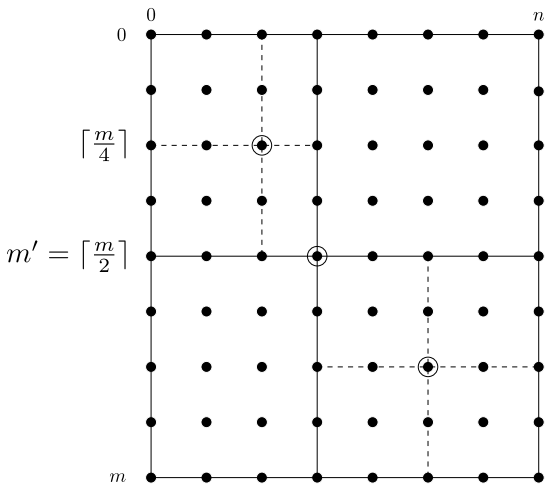


Figure 5.2: Illustration of linear-space alignment.

Remarks. While the linear-space algorithm presented here (which we have first seen in the book by Durbin et al. (1998)) is conceptually simple, the devil is in the details. An implementation probably requires a substantial amount of time for debugging.

An alternative to this algorithm does not only remember one point in row m' where the optimal alignment passes through, but the whole interval in row m', as well as the previous vertex in row $m' - 1$.

In yet another formulation, due to Hirschberg (1975), the lower part of the matrix S is computed in "backwards" direction. This is possible since the alignment score is symmetric. The middle row is then obtained as the sum of the two middle rows from the upper and the lower halves, and optimal alignments pass through all vertices where this sum is maximal.

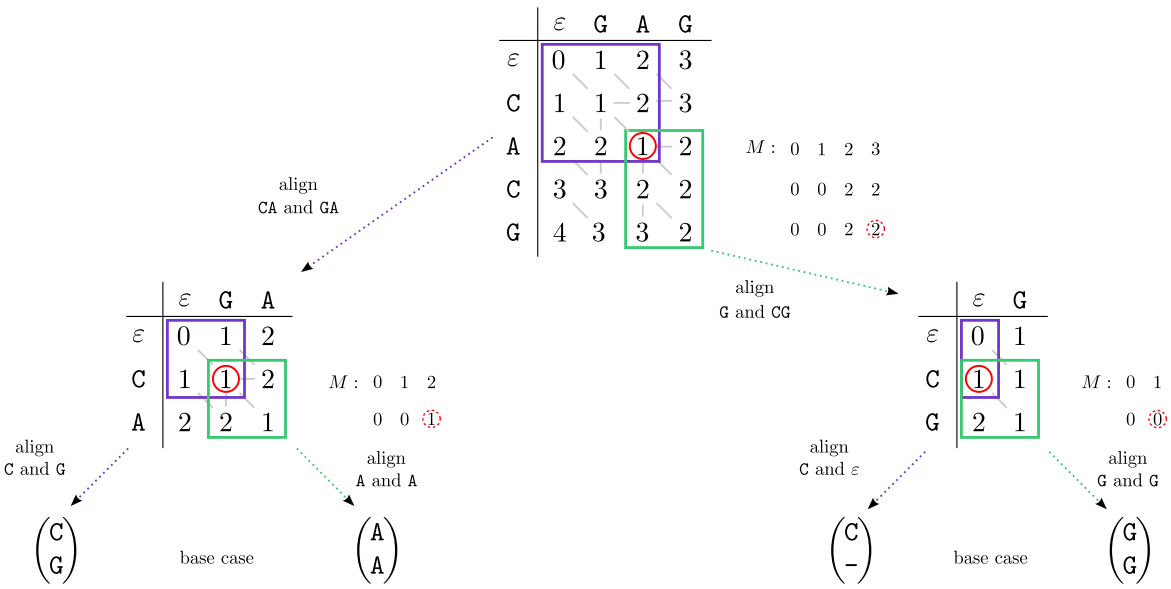
However, we believe that the formulation as given above is slightly simpler, although it uses a little bit more memory, namely one row of the matrix M.

Complexity analysis. The space complexity is in $O(m + n)$ since all matrix computations can be performed row-wise using memory for two adjacent rows at most, and the final alignment has at most $m + n$ columns.

A central question is, how much do we have to "pay" in terms of running time to obtain the linear space complexity? Fortunately, the answer is: Only approximately a constant factor of 2. In the original (quadratic-space) algorithm, each cell of the edit matrix S is computed once. In the linear-space version, some cells are computed several times during the recursions.

The amount of cells computed in the first pass is mn, in the second pass it is only half of the matrix, in the third pass it's a quarter of the matrix and so on, for a total of $mn \cdot (1 + 1/2 + 1/4 + \dots) \leq 2mn$ cells (geometric series). Thus, the asymptotic time complexity is still $O(mn)$.

Example 5.10 Given strings $s =$ `CACG` and $t =$ `GAG`, show how alignment in linear space works.



The recursion ends in four base cases resulting in the overall alignment $\binom{\texttt{CACG}}{\texttt{GA-G}}$. Note that if a cut position is found (indicated by a dotted circle in backpointer matrix M and by a circle in the edit matrix), the sequences are cut in two *after* the two characters, such that the left side of t' contains the circled character and the right side does not. ■

Affine and general gap costs. The presented technique integrates well with affine gap costs, but again, the devil is in the details. We need to remember whether a continuing horizontal or vertical gap passes through the optimal midpoint of the alignment and use this information during the recursive calls appropriately.

The Hirschberg technique (forward computation of the upper half, backward computation of the lower half) is slightly more complicated to combine with affine gap costs. The details were first given by Myers and Miller (1988).

General gap costs cannot be handled in linear space. We cannot even compute just the alignment score in linear space since we always need to refer back to *all* cells to the left and above of the current one.

Local alignment. We have described how to compute an optimal *global* alignment in linear space. What about *local* alignments?

Note that a local alignment is in fact a global alignment of the two best-aligning substrings of s and t. So once we know the start- and endpoints of these substrings, we can use the above global alignment procedure.

It is easy to find the endpoints: They are given by the entry with the highest score in the (forward) local alignment matrix. To find the start points, we have two options:

(a) We use the reversal technique: The start points are just the endpoints of the optimal local alignment of the reversed strings. This technique is simple, but may become problematic if there are several equally high-scoring local alignments.

(b) We use the back-pointer technique, similar to the M-matrix above: Whenever a new local alignment is started (by choosing the zero-option in the local alignment recurrence), we keep a back pointer to the cell containing the zero in all cells in which an optimal local alignment that starts in that zero cell ends.

Suboptimal alignments. In practice, we are often interested in several, say K good alignments instead of a single optimal one. Can this also be done in linear space? Essentially, we need to remember the K highest entries in the edit matrix and once we have found an optimal alignment we store a list or hash-table to remember its path and subsequently take care that the edges of this path cannot be used anymore when we re-compute the next (now optimal) alignment. This requires additional space proportional to the total lengths of all discovered alignments, which is linear for a constant number of alignments.

CHAPTER 6

Pairwise Alignment in Practice

Contents of this chapter: Dot Plots, rapid database search methods, sequence database, short exact matches, q-gram (index), BLAST: on-line database search method.
Further contents in the appendix (Chapter C): Fast Smith-Waterman, FASTA: on-line database search method, hot spots, diagonal runs, index-based database search methods (BLAT, SWIFT, QUASAR), software propositions.

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).

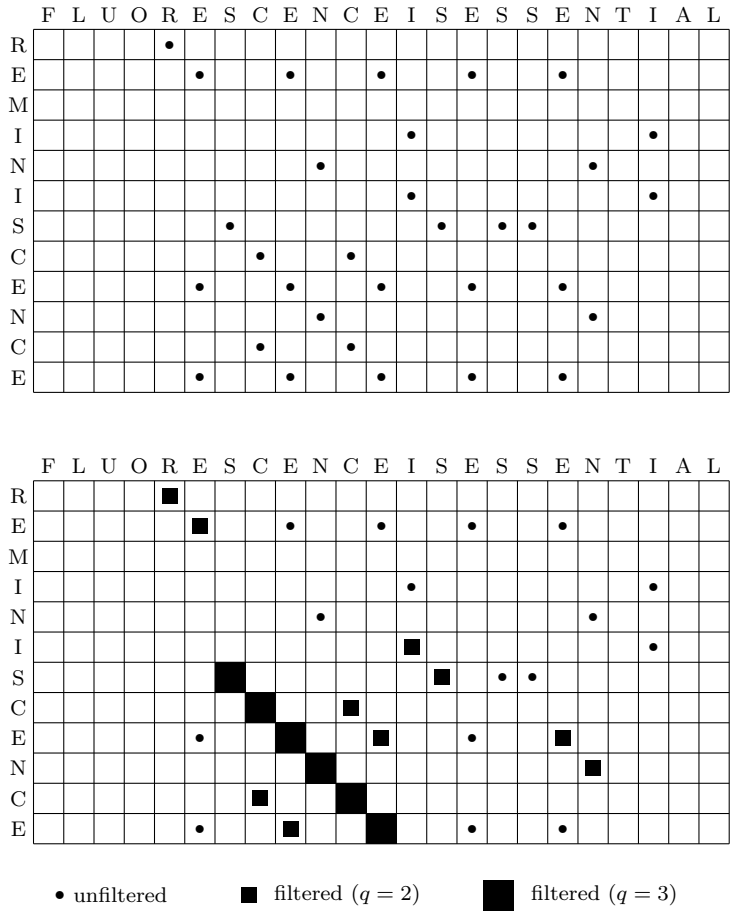


Figure 6.1: Upper part: Unfiltered dot plot. Lower part: Filtered dot plots. Here the filter keeps only those positions (i, j), that are part of a common substring of length $\geq q$.

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 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 searching: Given a **query** $x \in \Sigma^*$ and a family (database) Y of **subjects**, 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.

Definition 6.1 For our purposes, a **sequence database** $Y = (y_1, y_2, \ldots, y_L)$ is an ordered collection of sequences (sometimes we view Y simply as the set $\{y_1, y_2, \ldots, y_L\}$). We write $N := \sum_{y \in Y} |y|$ for the total length of the database.

The remainder of this chapter presents methods used in practice mainly for large-scale database search. Database search methods focus on speed and often sacrifice some accuracy to speed up similarity estimation. For the accurate alignment of a few sequences, the quadratic-time algorithms of the previous chapter are usually considered to produce the best-possible result (also called the gold standard), but recall the comment on length-normalized alignment at the end of Section 4.5.

Using short exact matches. Almost all methods make use of the following simple observation, the so-called ***q*-gram Lemma**:

Lemma 6.2 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.

In order to find exact matches of length q between query and database quickly, we first create a q-gram index I, either for the query x or for the database Y.

Definition 6.3 A ***q*-gram index** for $x \in \Sigma^m$ is a map $I : \Sigma^q \to \mathcal{P}(\{1, \ldots, m - q + 1\})$ 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 x and $|I(z)|$ is the occurrence count of z in x (cf. Definition 3.10).

A straightforward (but inefficient!) $O(|\Sigma|^q + qm)$ method to create a q-gram index (e.g. in Java) would be to use a `HashMap<String,ArrayList<Integer>>`, initialize each `ArrayList` to an empty list for each q-gram, slide a q-window across x and add each position to the appropriate list. The following code snippet assumes that string indexing starts at 1 and that `x.substring(i,j)` returns $x[i \ldots j-1]$.

```
for(i=1; i<=m-q+1; i++)
  I.get(x.substring(i,i+q)).add(i);
```

A more low-level and efficient $O(|\Sigma|^q + m)$ method is to use two integer arrays `first`$[0..|\Sigma|^q]$ and `pos`$[0..m-q+1]$ and the q-gram ranking function from Section 3.7 such that the starting positions of q-gram z with rank r are stored in `pos`, starting at position `first[r]` and ending at position `first[r+1]-1`. These can be constructed with a simple two-pass algorithm that scans x 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).

On-line and index-based searching. There are two basic approaches to rapid database searching.

1. In the first (so-called **on-line**) approach, the database (of size N) is completely examined for every query (of size m). Each query, however, can be pre-processed as soon as it becomes known. This means that the running time of such a method is at least $\Theta(N + m)$ 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). **FASTA** (Section C.2) and **BLAST** (Section 6.3) are well-known database search programs that work in this way.

2. The second (**index-based**) 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 contain potentially similar sequences. If these do not exist, the time spent for each query can become as small as $\Theta(m)$. Examples of index-based database search methods are BLAT and SWIFT, discussed in more detail in Section C.3.

The following table shows time complexities for on-line and index-based 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
On-line	——	$O(N + m)$	$O(N + m)$	$O(k(N + m))$
Index-based	$O(N)$	$O(m)$	$O(N + m)$	$O(N + k \cdot m)$

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

6.3 BLAST: An On-line Database Search Method

BLAST[1] (Altschul et al., 1990) is perhaps the most popular program to perform 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 exact q-gram hits, so-called BLAST hits, and then extend them.

Definition 6.4 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 $score(x[i \ldots i + q - 1], y[j \ldots j + q - 1]) \geq k$.

[1] Basic local alignment search tool

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 x and note the corresponding positions in x.

Definition 6.5 The k-neighborhood $N_k(x)$ of $x \in \Sigma^m$ is defined as

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

The k-neighborhood can be represented similar to a q-gram index: For each $z \in \Sigma^q$, we store the set $P_k(z)$ of positions i such that $(z, i) \in N_k(x)$.

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 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 $\{(i, j) : i \in P(y[j \ldots j + q - 1])\}$. This takes $O(|y| + z)$ time, where z is the number of hits.
2. Each hit (i, j) is scored (say, it has score X) and 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 $M - \Delta X$, where M is the maximal score reached so far and ΔX 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 reported.
3. MSPs on different diagonals are combined, and 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 left-extension and the right-extension method 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. to sensitivity than the one-hit approach.

CHAPTER 7

Suffix Trees

Contents of this chapter: Suffix trees, (enhanced) suffix arrays, nested suffixes, common prefixes, Σ-tree, Σ^+-tree, trie, generalized suffix trees, construction (WOTD, Ukkonen), applications (exact string matching, shortest unique substring, Maximal Unique Matches (MUM), Maximal Repeats).
Further contents in the appendix (Chapter F): Memory representation of suffix trees.

7.1 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 (e.g. human, mouse, rice) 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. Consequently, indexing methods are applicable. These methods preprocess the sequences in order to answer queries much faster than methods that work sequentially. Apart from the size of the string processing problems in genomics, their diversity is also remarkable. For example, to assemble the Genome of *Drosophila melanogaster*, a multitude of string processing problems involving exact and approximate pattern matching tasks had to be solved. These problems are often complicated by additional constraints about uniqueness, containment, or repetitiveness of sequences.

A **suffix tree** is a data structure suited to solve such problems. In this chapter, we introduce the concept of suffix trees, show their most important properties, and take a look at some applications of biological relevance.

Subsequently, in Chapter 8, 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.

7.2 An Informal Introduction to Suffix Trees

Let us consider a string $s \in \Sigma^n$. Think of s as a database of concatenated sequences (e.g. all sequences from UniProt), 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 up to $O(n^2)$ distinct substrings of s, i.e., too many to represent them all *explicitly* with a reasonable amount of memory. We need an *implicit* representation.

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.

We append a unique character $\$ \in \Sigma$ (called a **sentinel**[1]) 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

$$\$,\ a\$,\ ca\$,\ bca\$,\ abca\$,\ cabca\$.$$

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 7.1.

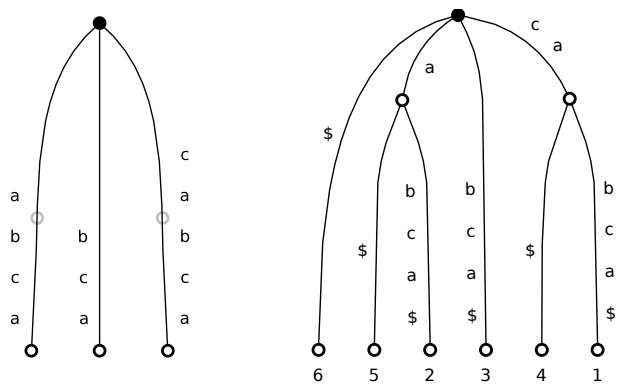


Figure 7.1: Suffix trees of cabca (left) and of cabca\$ (right). In the right tree, the leaves have been annotated with the starting positions of the suffixes.

Note the effect of appending the sentinal \$: 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 nested suffixes,

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

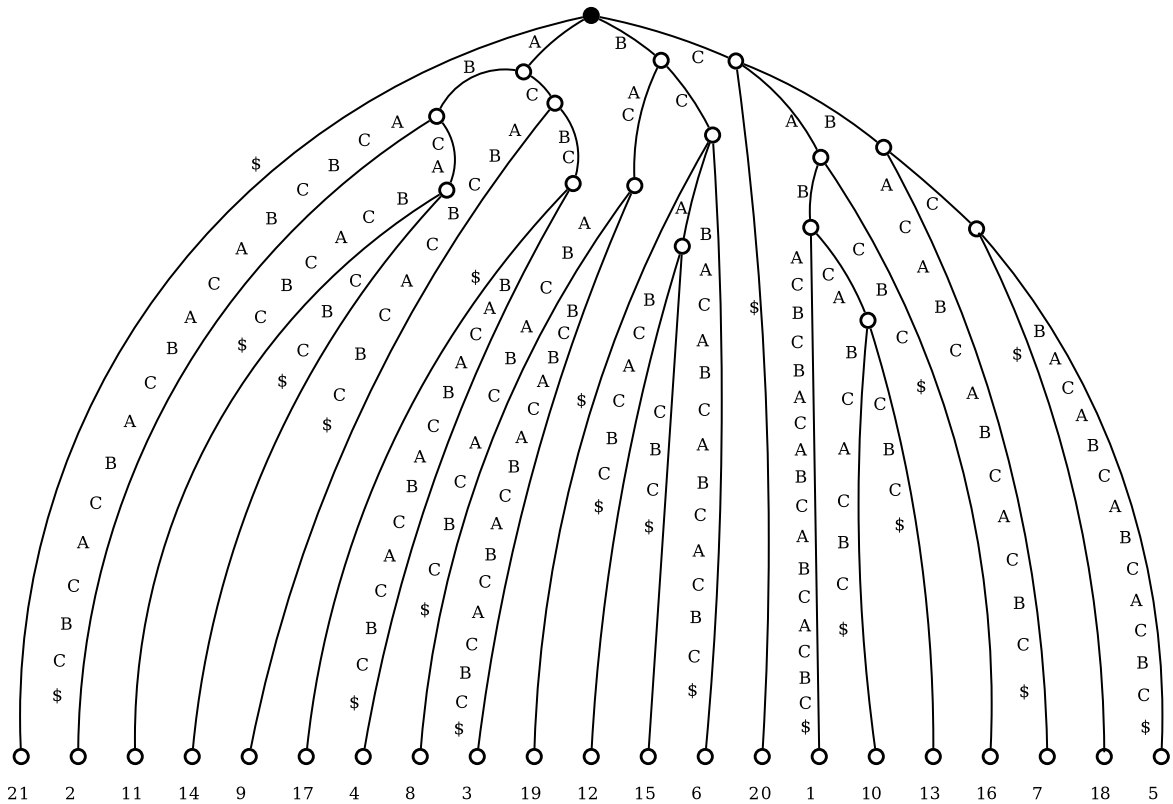


Figure 7.2: The suffix tree of CABACBCBACABCABCACBC$.

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 $s\$ = \text{cabca}\$$, or the larger example shown in Figure 7.2.

- 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 does not necessarily end at a leaf or at an internal node, but may end in the middle of an edge.

We now give more formal definitions and shall see that a "clever" representation of a suffix tree only needs linear, i.e., $O(n)$ space. Most importantly, we cannot store substrings at the edges, because the sum of their lengths is still $O(n^2)$. Instead, we will store *references* to the substrings.

7.3 A Formal Introduction to Suffix Trees

Definitions. A **rooted tree** is a connected directed acyclic graph with a special node r, called the **root**, such that all edges point away from it. The **depth** of a node v is its distance from the root; we have $\text{depth}(r) = 0$.

Let Σ be an alphabet. A **Σ-tree**, also called **trie**, is a rooted tree whose edges are labeled with a *single character* from Σ in such a way that no node has two outgoing edges labeled with the same letter.

A **Σ^+-tree** is a rooted tree whose edges are labeled with *non-empty strings* over Σ 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* $\text{stringdepth}(v) := |\text{string}(v)|$. In a Σ-tree, this is equal to $\text{depth}(v)$, 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 $\text{stringdepth}(v) \geq \text{depth}(v)$.

For a string x, if there exists a node v with $\text{string}(v) = x$, we write **node(x)** for v. Otherwise $\text{node}(x)$ is undefined. Sometimes, one can also find $\overline{x}$ written for $\text{node}(x)$ in the literature. Of course, $\text{node}(\varepsilon) = \overline{\varepsilon}$ 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 $xy = \text{string}(v)$. We finally define the *words* displayed by T by $\text{words}(T) := \{x : T \text{ displays } x\}$.

The **suffix trie** of $s \in \Sigma^*$ is the Σ-tree τ with $\text{words}(\tau) = \{s' : s' \text{ is a substring of } s\}$. The **suffix tree** of s is the compact Σ^+-tree T with $\text{words}(T) = \{s' : s' \text{ is a substring of } s\}$. As mentioned above, we often consider the suffix tree of $s\$$, where each suffix ends in a leaf.

In a suffix trie or suffix tree, an edge leading to an internal node is an **internal edge**. An edge leading to a leaf is a **leaf edge**.

Space requirements. 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 7.1 The suffix tree of a string of length n has at most $n - 1$ internal nodes.

Proof. Let L be the number of leaves and let I be the number of internal nodes. We do "double counting" of the edges; let E be their number. 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 suffixes than leaves), 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. 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 7.2 The suffix tree of a string s of length n can be stored in $O(n)$ space.

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 don't 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 original string.

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

Definition 7.3 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 $s\#t\$$.

7.4 Suffix Tree Construction: The WOTD Algorithm

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

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.

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

The WOTD algorithm adheres to the recursive structure of a suffix tree. The idea is that for each branching node node(u), the subtree below node(u) is determined by the set of all suffixes of $s\$$ that have u as a prefix. Starting with the root (where $u = \varepsilon$), we recursively construct the subtrees rooted in nodes corresponding to common prefixes.

To construct the subtree rooted in node(u), 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 $s\$$. To construct the subtree, we proceed as follows.

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

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

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

The WOTD algorithm starts by evaluating the root from the set of all suffixes of $s\$$. All internal nodes are evaluated recursively in a top-down strategy.

Example 7.4 Consider the input string $s\$:= abab\$$. 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 the first five columns in Figure 7.3 The algorithm recognizes three groups of suffixes: the \$-group, the a-group, and the b-group.

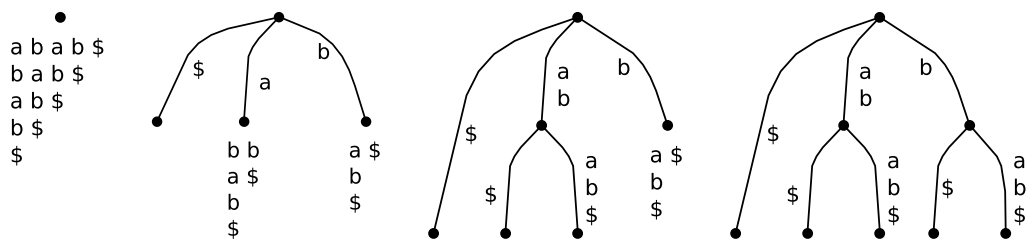


Figure 7.3: The write-only top-down construction of the suffix tree for $abab\$$

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

The a-group contains two suffixes, $bab\$$ 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 $ab\$$ and \$, which is evaluated recursively. Since all suffixes differ in the first character, we obtain two singleton groups of suffixes, and thus two leaf edges outgoing from $\text{node}(ab)$, labeled by \$ and $ab\$$.

The b-group of the root contains two suffixes, too: $ab\$$ 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 $\text{node}(b)$ has two remaining suffixes $ab\$$ and \$, which are recursively classified. ■

Analysis. The worst case running time of WOTD is $O(n^2)$. Consider, for example, the string $s = a^n$. 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 expected case, the maximal depth of the branching nodes is much smaller than $n-1$, namely $O(\log_{|\Sigma|}(n))$. 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 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 only depends 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 could construct a subtree only when it is traversed for the first time. This would result in a "lazy construction", which could also be implemented in an eager imperative language (such as C). Experiments show that such a lazy construction is very fast.

7.5 Suffix Tree Construction: Ukkonen's Linear-Time Online Algorithm

This algorithm constructs the suffix tree of s *online*, i.e., it generates a sequence of suffix trees for all prefixes of s, starting with the suffix tree of the empty sequence ε, followed by the suffix trees of $s[1]$, $s[1]s[2]$, $s[1]s[2]s[3]$, ..., s, $s\$$. The method is called *online* since in each step the suffix trees are constructed without knowing the remaining part of the input string. In other words, the algorithm may read the input string character by character from left to right.

Note that during the intermediate steps, the string has no sentinel character at its end. Therefore, not all suffixes correspond to leaves. Instead, some suffixes may end in the middle of an edge or at an internal node. This changes as soon as the last character \$ is added.

The suffix tree of the empty sequence consists just of the root and is easy to construct. Thus we only need to focus on the changes that happen when we append a character to s. If we can find a way to use only constant time per appended character (asymptotically), then we have a linear-time algorithm.

How this can be done is well explained in Gusfields's book, Chapter 6 (Gusfield, 1997).

7.6 Applications of Suffix Trees

7.6.1 Exact String Matching

Problem 7.5 (Exact String Matching Problem) Given a text $s \in \Sigma^*$ and a pattern $p \in \Sigma^*$, it can be defined in three variants:

1. decide whether p occurs at least once in s (i.e., whether $p \blacktriangleleft s$),

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 occurences z of p in s. We show that it can be solved in $O(|p| + z)$ 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 you 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 also is 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 pre-process 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 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 7.6 Let $s = abbab$. The corresponding suffix tree of abbab\$ is shown in Figure F.1 on page 166.

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. ■

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 required to find the left-to-right partition (which is optimal) to compute the maximal matches distance (see Section 3.8).

7.6.2 The Shortest Unique Substring

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 7.7 (Shortest Unique Substring Problem) Given a string $s \in \Sigma^*$, find all shortest strings u that occur once, but only once, in s.

Extensions of this problem have applications in DNA primer design, for example. The length restriction serves to exclude too trivial solutions.

Example 7.8 Let $s = abab$. Then the shortest unique substring is ba. ■

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) $\rightarrow v$, 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. Among all such strings, we pick the shortest one(s).

Note that we disregard edges whose label starts with \$, since the sentinel is always a unique substring by construction.

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 algorithm thus runs in linear time since the suffix tree can be constructed in linear time, and there is a linear number of nodes and edges in the suffix tree. This is optimal, since the running time is linear in the size of the input.

7.6.3 Maximal Repeats

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

Definition 7.9 Given a string $s \in \Sigma^*$, a **repeat** of s is a triple (i, i', ℓ) of two starting positions $i < i'$ in s and a length ℓ such that $s[i, \ldots, i + \ell - 1] = s[i', \ldots, i' + \ell - 1]$.

Equivalently, the same repeat can be described by two position pairs, containing starting and ending position of each instance, i.e., for (i, i', ℓ) we can also write $((i, j), (i', j'))$, where $j = i + \ell - 1$ and $j' = i' + \ell - 1$.

In that case (i, j) is called the **left instance** of the repeat and (i', j') is called the **right instance** of the repeat; see Figure 7.4. Note that the two instances may overlap.

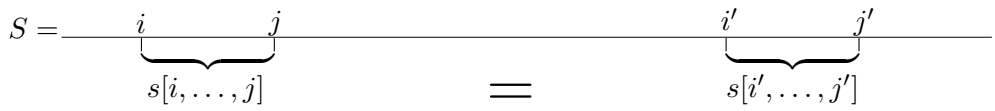


Figure 7.4: Illustration of a repeat

Example 7.10 The string $s = gagctcgagc$, $|s| = 10$ contains the following repeats of length ≥ 2:

$((1,4),(7,10))$	$gagc$
$((1,3),(7,9))$	gag
$((1,2),(7,8))$	ga
$((2,4),(8,10))$	agc
$((2,3),(9,10))$	ag
$((3,4),(9,10))$	gc

■

We see that shorter repeats are often contained in longer repeats. To remove redundancy, we introduce maximal repeats, illustrated in Figure 7.5. Essentially, maximality means that the repeated substring cannot be extended to the left or right.

Definition 7.11 A repeat $((i, j), (i', j'))$ 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 is **maximal** if it is both left-maximal and right-maximal.

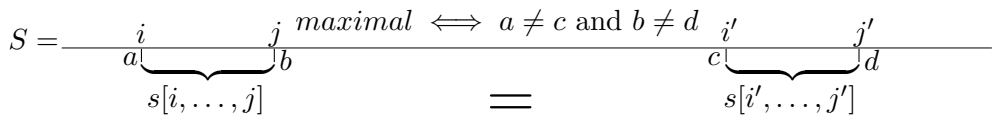


Figure 7.5: Illustration of maximality

From now on we restrict ourselves to maximal repeats. All non-maximal repeats can easily be obtained from the maximal repeats. In Example 7.10, the last four repeats can be extended to the left or to the right. Hence only the first repeat $\big((1,4),(7,10)\big)$ is maximal.

Problem 7.12 (Maximal Repeat Discovery Problem) Given a string $s \in \Sigma^*$, find all maximal repeats of s (possibly of a given minimal length ℓ).

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

In detail, suppose we have the suffix tree for some string s of length n over some alphabet Σ such that the first and the last character of s both occur exactly once in s. We ignore leaf edges from the root, since the root corresponds to repeats of length zero and we are not interested in these. Figure 7.6 gives an example for the string *ggcgctgcgcc*$.

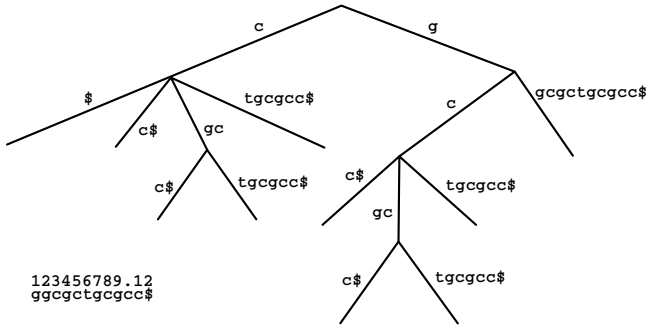


Figure 7.6: The suffix tree for *ggcgctgcgcc*$. Leaf edges from the root are not shown. These edges are not important for the algorithm.

In the **first phase**, the algorithm annotates each leaf of the suffix tree: if $\nu = s[i \dots n]$, then the leaf v with path-label ν is annotated by the pair (a, i), where i is the position at which the suffix ν 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 $A(v, \sigma) = \emptyset$ (the empty set) for all characters $\sigma \in \Sigma$ different from $s[i-1]$. The latter assumption holds in general (also for branching nodes) whenever there is no annotation (σ, j) for some j. For the suffix tree of Figure 7.6, the leaf annotation is shown in Figure 7.7.

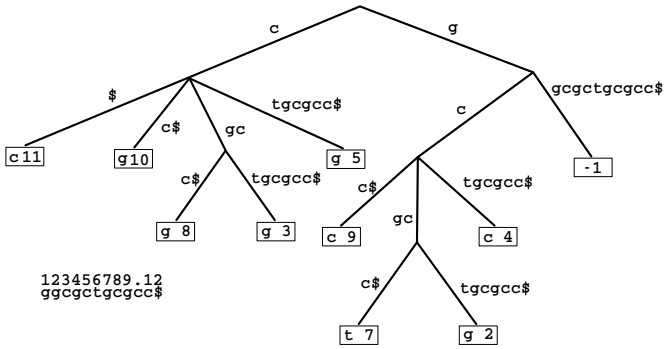


Figure 7.7: The suffix tree for *ggcgctgcgcc*$ with leaf annotation.

The leaf annotation gives us the character upon which we decide the left-maximality of a repeat, 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 repeats are reported and simultaneously the annotation for the branching nodes is computed. A bottom-up traversal means that a branching node is visited only after all of its children have been visited.

Each edge, say $v \to w$, is processed as follows:

1. Repeats (for the string ν 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 output all pairs $((i, i+q-1), (j, j+q-1))$, where
   - q is the depth of node v, i.e. $q = |\nu|$,
   - $i \in A(v, \sigma)$ and $j \in A(w, \sigma')$ for some characters $\sigma \neq \sigma'$,
   - $A(v, \sigma)$ 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'.
2. Only those positions are combined which have different characters to the left. Thus it guarantees left-maximality of the repeats. 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 x 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 σ, is different from a. Now since ν is the repeated substring, σ and a are characters to the right of ν. 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 repeats.

As soon as for the current edge the repeats are reported, the algorithm computes the union $A(v, \sigma) \cup A(w, \sigma)$ for all characters σ, 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 the set of positions where ν occurs, and this set is divided into (possibly empty) disjoint subsets $A(v, \sigma_1), \ldots, A(v, \sigma_r)$, where $\Sigma = \{\sigma_1, \ldots, \sigma_r\}$.

Example 7.13 The suffix tree of the string $s = ggcgctgcgcc\$$ is shown in Figure 7.6. We assume the leaf annotation of it (as shown in Figure 7.7) is already determined.

Proceeding from leaves 7 and 2, the bottom up traversal of the suffix tree for $ggcgctgcgcc\$$ begins with node v whose path-label is $gcgc$ of depth $q = 4$. This node has two children which do not have the same character to their left (t vs. g). The node is annotated with $(g, 2)$ and $(t, 7)$. Because our repeat is also left-maximal, it is reported: $((2, 2+4-1), (7, 7+4-1)) = ((2, 5), (7, 10))$, as can be seen in Figure 7.8, which shows the annotation for a large part of the suffix tree and some repeats.

Next comes the node v with path-label gc of depth two. The algorithm starts by processing the first edge outgoing from v. Since initially there is no annotation for v, no repeat is reported and v is annotated with the label of leaf 9: $(c, 9)$. Then the second edge is processed. This means that the annotation $(g, 2)$, $(t, 7)$ for w with path-label $gcgc$ is combined with the annotation $(c, 9)$. The new annotation for v becomes $(c, 9)$, $(t, 7)$, $(g, 2)$. This gives the repeats $((7, 8), (9, 10))$ and $((2, 3), (9, 10))$. Finally, the third edge is processed. $(c, 9)$ and $(c, 4)$ cannot be combined, see condition 2 above. So only the repeats $((4, 5), (7, 8))$ and

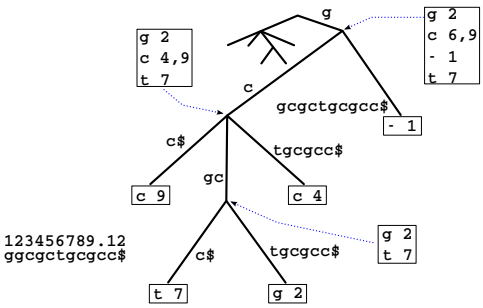


Figure 7.8: The annotation for a large part of the suffix tree of Figure 7.7 and some repeats.

$((2,3),(4,5))$ are reported, resulting from the combination of $(t,7)$ and $(g,2)$ with $(c,4)$. The final annotation for v is $(c,9)$, $(t,7)$, $(g,2)$, $(c,4)$, which can also be read as $A(v,g) = \{2\}$, $A(v,c) = \{4,9\}$ and $A(v,t) = \{7\}$. 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 repeats 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 repeats. 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 $O(n+k)$ time where n is the length of the input string and k is the number of repeats.

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

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

7.6.4 Maximal Unique Matches

The standard dynamic programming algorithm to compute the optimal alignment between two sequences of length m and n requires $O(mn)$ steps. This is too slow if the sequences are on the order of 100 000s or millions 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 part 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 will be discussed in Chapter 13 of these notes). We first have to define the notion MUM precisely:

Definition 7.14 Given strings $s, t \in \Sigma^*$ and a minimal length $\ell \geq 1$, a string u that satisfies

1. $|u| \geq \ell$,
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).

is called a **maximal unique match** or **MUM** of s and t.

Problem 7.15 (Maximal Unique Matches Problem) Given $s, t \in \Sigma^*$ and a minimal length $\ell \geq 1$, find all MUMs of s and t.

Example 7.16 Let $s = ccttcgt$, $t = ctgtcgt$, and $\ell = 2$. Then there are two maximal unique matches, ct and $tcgt$. 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 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, and no further children. It remains to check the left-maximality in each case. We thus arrive at the following algorithm:

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

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

If all checks are true, output string(q) and/or its positions i and j.
Clearly, the algorithm runs in linear time 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.

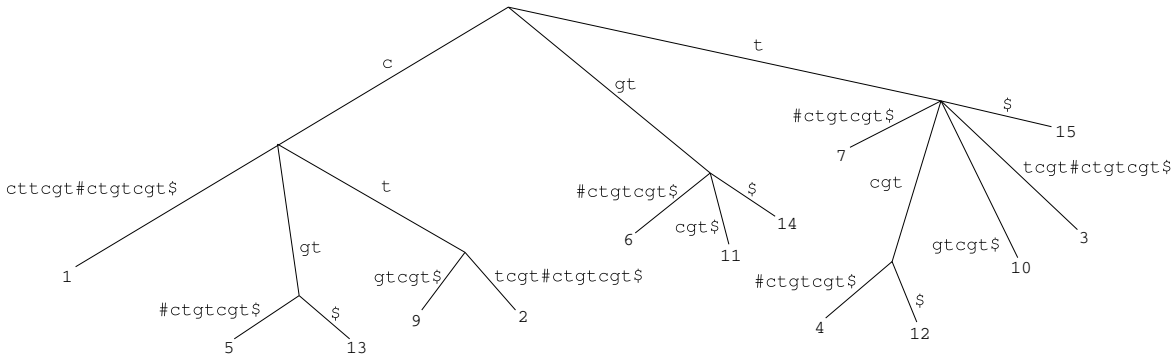


Figure 7.9: The suffix tree for *ccttcgt#ctgtcgt*\$ without the leaf edges from the root

Example 7.17 Let $s = ccttcgt$, $t = ctgtcgt$, $\ell = 2$. Consider the suffix tree for $s\#t\$$ shown in Figure 7.9.

The string *tcgt* occurs once in s and t, since there are two corresponding leaf edges from branching node node(*tcgt*). Comparing the characters g and t immediately to the left of the occurrences of *tcgt* in s and t verifies left-maximality. Similarly for *ct*. On the other hand, *cgt* is not left-maximal, because both occurrences have the left neighbor t. ■

CHAPTER 8

Suffix Arrays

Contents of this chapter: suffix array `pos`, inverse suffix array `rank`, longest common prefix array `lcp`, suffix array construction, quicksort, Manber-Myers algorithm, construction of `rank` and `pos`, applications.

8.1 Motivation

We have already seen that suffix trees are a very useful data structure for a variety of string matching problems. We have also discussed their memory requirements: Storing a suffix tree in the way outlined in Section F.1 is among the most efficient possible ways, but also limits the string length somewhat. Additionally, this way of storing the tree (instead of explicitly representing leaves and internal nodes) is rather new. 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 nevertheless captures the essence of the suffix tree: the **suffix array**.

More recently, it has become clear that a suffix array complemented with additional information, called an **enhanced suffix array** or **extended suffix array**, can 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: This gives us the (starting positions of the) suffixes in ascending lexicographic order.

8.2 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, and assume that \$ is lexicographically smaller than any character of the alphabet. In examples, we shall always assume a natural order on the alphabet, i.e., $a < b < c < \ldots$.

Definition 8.1 For a string $s \in \Sigma^n$, the **suffix array** $\texttt{pos}$ of $t = s\$$ is a permutation of the integers $\{0, \ldots, n\}$ such that $\texttt{pos}[r]$ is the starting position of the lexicographically r-th smallest suffix of t.

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

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

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 maximal common prefix of all suffixes below that node. We can therefore recover information about the internal nodes by making the following definition.

Definition 8.2 Given a string $s \in \Sigma^n$ and the suffix array $\texttt{pos}$ of $t = s\$$, we define the **longest common prefix array** $\texttt{lcp} : \{1, \ldots, n\} \to \mathbb{N}_0$ by

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

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 8.3 Table 8.1 shows the suffix array $\texttt{pos}$, its inverse $\texttt{rank}$ and the $\texttt{lcp}$ array of the string abbab\$. The suffix tree was given in Figure F.1. Since 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: $\texttt{pos} = (5, 3, 0, 4, 2, 1)$. ■

lcp intervals. It is important to realize the one-to-one correspondence between internal nodes in the suffix tree and certain intervals in the suffix array. We have already seen that all leaves below an internal node v in the suffix tree correspond to an interval $\texttt{pos}[r]$, $r \in [r^-, r^+]$, in the suffix array. If v has string depth d, it follows that

- $\texttt{lcp}[r^-] < d$, since the suffix starting at $\texttt{pos}[r^- - 1]$ is not below v and hence does not share a common prefix of length d with the suffix starting at $\text{pos}[r^-]$;

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

p	0	1	2	3	4	5
rank[p]	2	5	4	1	3	0

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

Table 8.1: Suffix array `pos` and its inverse `rank` of abbab$. The table also shows the longest common prefix array `lcp`.

- $\texttt{lcp}[r] \geq d$ for all $r^- < r \leq r^+$, 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 larger string depth than d, too.

- $\texttt{lcp}[r^+ + 1] < d$, since again the suffix starting at $\texttt{pos}[r^+ + 1]$ is not below v.

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

8.3 Suffix Array Construction Algorithms

8.3.1 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, and there is a linear number of edges.

If we use the memory layout described in Section F.1 for the suffix tree and store the internal nodes in depth-first order, this construction is particularly simple. We simply walk through the memory locations from left to right, keep track of the current string depth, and collect all leaf pointers along the way. We recognize a leaf pointer by the L-flag. The leaf number is obtained by subtracting the current string depth from the leaf pointer (see Section F.1 for details).

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.

8.3.2 Direct Construction

Quicksort: 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.

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 (e.g. using the C stdlib function `qsort`) and performs very well in practice on strings that do not contain long repeats.

Manber-Myers Algorithm: Manber and Myers (1990), 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 $K := \lceil \log_2(n) \rceil$ 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 classify all suffixes according to their first character. Each of the following phases must then double the number of accounted characters and update the suffix order accordingly. After phase K, all suffixes are correctly sorted with respect to their first $2^K \geq n$ characters and therefore in correct lexicographic order. We shall show that the initial sorting and each doubling phase runs in $O(n)$ time; thus establishing the $O(n \log n)$ running time of the algorithm.

Example 8.4 Given string $s\$ =$ `STETSTESTE$`, construct the suffix array using the Manber-Myers algorithm. The algorithm takes at most $k \in \{0, \ldots, K\}$ phases where $K = \lceil \log_2(11) \rceil = 4$.

```
0123456789.
STETSTESTE$
```

σ	#	bucket start
$	1	0
E	3	1
S	3	4
T	4	7

after phase $k = 0$ ($2^k = 1$)

$	E	E	E	S	S	S	T	T	T	T
10	2	6	9	0	4	7	1	3	5	8
/	3T	7S	10$	1T	5T	8T	2E	4S	6E	9E

after phase 1 ($2^k = 2$)	$ 10	E$ 9	ES 6	ET 2	ST 0	ST 4	ST 7	TE 1	TE 5	TE 8	TS 3
	/	/	8TE	4ST	2ET	6ES	9E$	3TS	7ST	10$	5TE
after phase 2 ($2^k = 4$)	$ 10	E$ 9	ES 6	ET 2	STE$ 7	STES 4	STET 0	TE$ 8	TEST 5	TETS 1	TS 3

The Manber-Myers algorithm has sorted the suffixes of $s\$$ at the latest after phase $K = 4$, more precisely in this example already after phase 2. In each phase, newly gained information is black and newly formed buckets are marked with | while old bucket borders are represented by ||. Below each character that has position i in $s\$$ the substring at position $i + 2^k$ of length 2^k is given. The substring is used for lexicographically sorting within the same bucket. ■

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.

8.3.3 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 Java 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 $\texttt{left}[p] := \texttt{pos}[\texttt{rank}[p] - 1]$; this is the number immediately left of the number p in the suffix array.

In the first iteration of the loop, we have $p = 0$ and compute the lcp-value at $\texttt{rank}[p]$, i.e., the length L of the longest common prefix of the suffixes starting at position p and at position $\texttt{left}[p]$ (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 $\texttt{left}[p + 1]$. If $\texttt{left}[p + 1] = \texttt{left}[p] + 1$, 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 $\texttt{left}[p + 1] \neq \texttt{left}[p] + 1$, 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 $\texttt{left}[p] + 1$ 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 $\texttt{left}[p] + 1$ and $p + 1$ share a prefix of length $L - 1$. Everything in between must share a common prefix that is at least as long. Thus we need not 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 8.5 Let $L := \texttt{lcp}[\texttt{rank}[p]]$. Then $\texttt{lcp}[\texttt{rank}[p + 1]] \geq L - 1$.

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 prove that these savings lead to a linear-time algorithm. 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 8.6 Given the string $s\$$ and its suffix array `pos`, the `lcp` array can be computed in linear time.

8.4 Applications of Suffix Arrays

- exact string matching ($O(|p| \log n)$ or $O(|p| + \log n)$ with `lcp` array)
- Quasar (q-gram based database searching using a suffix array)
- matching statistics
- Burrows-Wheeler Transformation, see Chapter 9
- ... and more

CHAPTER 9

Burrows-Wheeler Transformation

Contents of this chapter: Burrows-Wheeler transformation and retransformation, exact string matching with the BWT, compression with run-length encoding.

9.1 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, 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 9.4.1).

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.

9.2 Transformation and Retransformation

For the transformation of the input string, first all the cyclic rotations of the text are computed and written into a matrix, one below the other. Afterwards the rotations are lexicographically sorted. The last column L is the output of the transformation step, besides an index I that gives the row in which the original string is found.

Definition 9.1 Let s be the input string of length $|s| = n$. Let M be the $n \times n$-matrix containing in its rows the lexicographically sorted cyclic rotations of s. The *Burrows-Wheeler transform* $\mathrm{bwt}(s)$ is the last column of M,

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

The index I is the index of the row that contains the original string s, i.e. $s[j] = M(I, j)$ for all $0 \leq j < n$. Instead of this index, a sentinel character \$ can be appended, so no additional information has to be saved.

Another possibility to compute $\mathrm{bwt}(s)$ is to construct the suffix array and decrement each entry of the array `pos` (modulo n if necessary). Then $\mathrm{bwt}(s)[i] = s[\mathtt{pos}[i]_{\mathrm{dec}\, n}]$ for all $1 \leq i < n$, where $x_{\mathrm{dec}\, n} = (x - 1) \mod n$.

An important observation facing the output string $\mathrm{bwt}(s)$ is that equal characters are often grouped together due to the properties of natural speech. In an English text, for example, 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'. This phenomenon is called *left context* and can be observed in every natural language – of course with differently distributed probabilities.

The reconstruction of the text is done in a back-to-front manner. Besides $\mathrm{bwt}(s)$, which is the last column of the matrix M, the first column is available by lexicographically sorting $\mathrm{bwt}(s)$. After reconstructing the first column, the character at position I, respectively the row containing the sentinel \$ in the last column, is examined. (This is equivalent, because I indicates the row where the original string was found in M, thus in the last column at position I the last character of s is found. The sentinel, which is unique in the string, is appended to the end of the text. Thus, the row where the sentinel is found in the last column has to be the row where the original string is found in the matrix.)

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 in the same row. So the next step is to search the occurrence of the first reconstructed character in the first column and look up the precursor in the last. If this character is unique in the text, this is trivial. Otherwise an important property of the Burrows-Wheeler transform is used, which says that the ith occurrence of the character in L corresponds to the ith occurrence in F. So, in order to find the corresponding character from the last column in the first column, it is necessary to know how many instances of this character occur earlier in the last column. Next the corresponding occurrence of this character is searched in the first column. The index of this character thus gives the position where the precursor in the last column is found. (Efficient ways how to perform these searches are known, see e.g. Kärkkäinen (2007) or Adjeroh et al. (2008), but the details go beyond the scope of these lecture notes.) This procedure is repeated until the starting index I, respectively the sentinel \$, is reached again. Here, the reconstruction phase ends and the original input string is obtained.

Example 9.2 Let the text be $t = s\$ =$ STETSTESTE$.

Transformation: Construct the matrix M by building the cyclic rotations of t and sorting them (shown in Table 9.1). The Burrows-Wheeler transform bwt(t) can be found in the last column $L =$ bwt(t) = 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

Table 9.1: This is the matrix M containing the cyclic rotations.

Retransformation: Reconstruct column F by lexicographically sorting the last column $L =$ bwt(t) = ETTTET$SSSE, giving $F =$ $EEESSSTTTT (this is sketched in Figure 9.1). 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.

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 9.1: Illustration of the reconstruction path through the matrix.

■

9.3 Exact String Matching

Amazing about the BWT is the existence of an exact string matching algorithm that works directly on the transformed text. 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.

Example 9.3 Let the text again be $t = s\$ =$ `STETSTESTE$` and the pattern be $p =$ `TEST`. First, the last character of p, namely `T`, is searched in F and all occurrences are highlighted in black. Then the next character of p, which is `S`, is searched in L. All occurrences of `S` that are precursors of the already highlighted `T`s are also marked. The newly marked `S`s are searched in F and the precursors, which correspond to the next character of the pattern (`E`), are searched. Here there is only one matching precursor. Afterwards the last searched character, the `T`, is found as a precursor of the `E`. In Table 9.2 this example is sketched.

F	$\cdots$	L
$		E
E		T
E		T
E		T
S		E
S		T
S		$
T		S
T		S
T		S
T		E

F	$\cdots$	L
$		E
E		T
E		T
E		T
S		E
S		T
S		$
T	$\longrightarrow$	S
T	$\longrightarrow$	S
T	$\longrightarrow$	S
T		E

F	$\cdots$	L
$		E
E		T
E		T
E		T
S		E
S		T
S		$
T	$\longrightarrow$	S
T	$\longrightarrow$	S
T	$\longrightarrow$	S
T		E

F	$\cdots$	L
$		E
E		T
E		T
E		T
S	$\longrightarrow$	E
S	$\longrightarrow$	T
S	$\longrightarrow$	$
T	$\longrightarrow$	S
T	$\longrightarrow$	S
T	$\longrightarrow$	S
T		E

	F	$\cdots$	L
	$		E
	E		T
	E	$\longrightarrow$	T
	E		T
	S	$\longrightarrow$	E
	S		T
	S		$
	T	$\longrightarrow$	S
▷	T		S
	T		S
	T		E

Table 9.2: Searching the pattern `TEST` in `STETSTESTE$`. All observed characters are highlighted in `black`, the found pattern in addition is highlighted with a shaded background.

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 and the index is determined. Here, we thus have the second occurring `T` (marked with ▷ in Table 9.2), which is at index 8 in F. Looking up the start position of the rotation, the index 5 is gained, and this is exactly the position where the pattern starts in s. ■

9.4 Other Applications

Besides exact string matching, also other common string matching problems can be solved with the BWT. For example, alignment tools have been developed that work on the BWT, for example BWA (Li and Durbin, 2009, 2010) or Bowtie (Langmead et al., 2009). It applies a global alignment algorithm on short queries and contains 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.

9.4.1 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 x, the algorithm will save $3x$. Texts with many and long runs thus will obviously be effectively compressed.

Example 9.4 Considering the example $t = s\$ = \texttt{STETSTESTE\$}$, the compression with RLE will have no advantage. If instead the Burrows-Wheeler transform of t $\mathrm{bwt}(t) = \texttt{ETTTET\$SSSE}$ is compressed, the space requirement is reduced by two characters, since the compressed string is $\mathrm{rle}(\mathrm{bwt}(t)) = \texttt{E3TET\$3SE}$. As soon as the input string gets longer and thus more grouped characters occur, the space requirement reduction is even more significant. ■

CHAPTER 10

Multiple Sequence Alignment

Contents of this chapter: Multiple sequence comparison, multiple alignment scoring functions, (weighted) sum-of-pairs score, (weighted) sum-of-pairs cost, multiple sequence alignment problem, optimal alignment (score/cost), parsimony principle, generalized tree alignment.

10.1 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 at least two 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 10.1 Let Σ be the character alphabet, and $s_1, \ldots, s_k \in \Sigma^*$.

Let $\mathcal{A}(k) := (\Sigma \cup \{-\})^k \setminus \{(-)^k\}$ be 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 $\pi_{\{i\}}$ to the i-th sequence** is defined as the homomorphism $\mathcal{A}(k)^* \to \Sigma^*$ with

$$\pi_{\{i\}}(\left(\begin{smallmatrix} a_1 \\ \vdots \\ a_k \end{smallmatrix}\right)) := \begin{cases} a_i & \text{if } a_i \neq -, \\ \varepsilon & \text{if } a_i = -. \end{cases}$$

In other words, the projection reads the i-th row of a multiple alignment, omitting all gap characters.

A **global multiple alignment** of $s_1, \ldots, s_k$ is a sequence $A \in \mathcal{A}(k)^*$ with $\pi_{\{i\}}(A) = s_i$ for all $i = 1, \ldots, k$.

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

Definition 10.2 The projection of a multiple alignment A to an index set $I = \{i_1, i_2, \ldots, i_q\} \subseteq \{1, \ldots, k\}$ is defined as the unique homomorphism that maps an alignment column a as follows:

$$\pi_I(\left(\begin{smallmatrix} a_1 \\ \vdots \\ a_k \end{smallmatrix}\right)) := \begin{cases} \varepsilon & \text{if all } a_{i_l} = - \\ \left(\begin{smallmatrix} a_{i_1} \\ \vdots \\ a_{i_q} \end{smallmatrix}\right) & \text{otherwise.} \end{cases}$$

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 Figure 10.1 and in the following example.

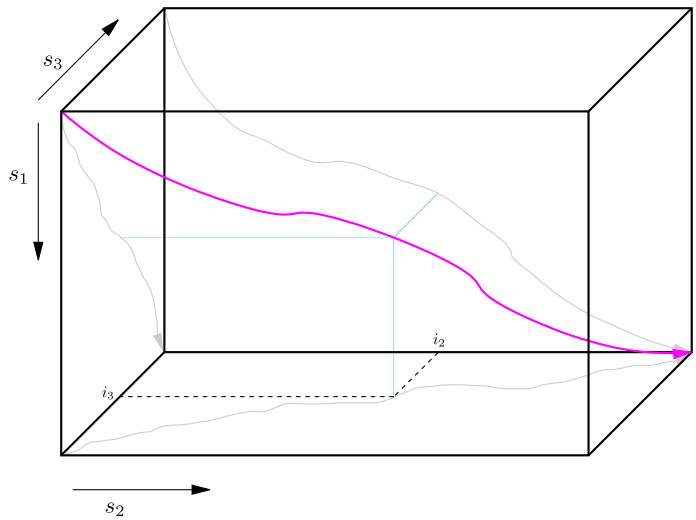


Figure 10.1: Projection of a multiple alignment of three sequences s_1, s_2, s_3 on the three planes (s_1, s_2), (s_2, s_3) and (s_3, s_1).

Example 10.3 Let

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

Then

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

■

10.2 Why multiple sequence comparison?

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

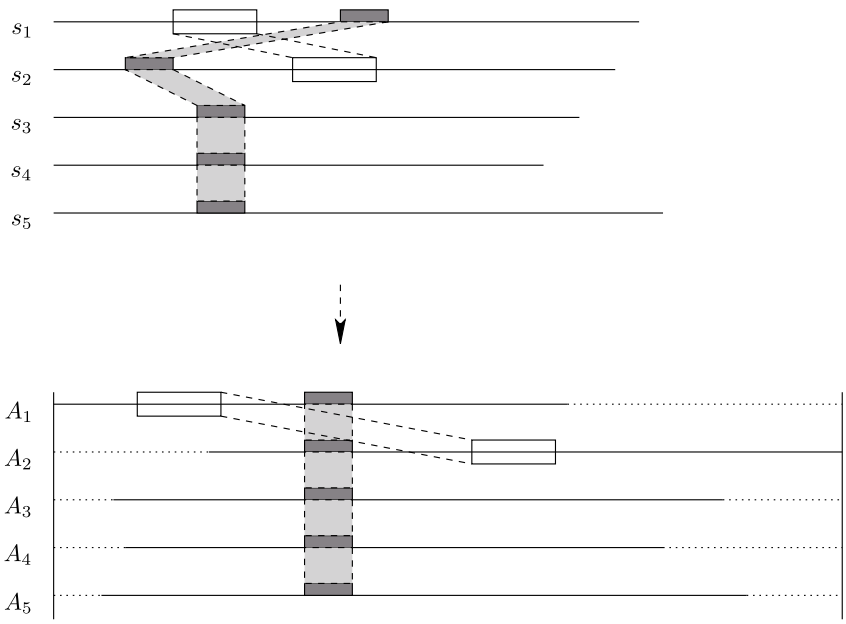


Figure 10.2: A small pattern present in many sequences may become better visible in a multiple sequence alignment.

- 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 10.2, 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 six sequences at the same time it becomes clear that the gray region is a better characteristic feature of this sequence family than the white one.

- Sometimes alignment ambiguities can be resolved due to the additional information given, as the following example shows: $s_1 = \texttt{VIEQLA}$ and $s_2 = \texttt{VINLA}$ 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 $s_3 = \texttt{VINQLA}$ 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 phylogenetic 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)

10.3 Multiple Alignment Scoring Functions

We have not discussed quantitative measures of alignment quality so far. In fact, several such measures exist, and in this section we will define a few of them.

Like in pairwise alignment, the first choice one has to make is that between a dissimilarity (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 encounter examples of both kinds.

All of the multiple alignment score functions discussed here are based on pairwise alignment score or cost functions, here denoted by $S_2(\cdot)$ for score functions and $D_2(\cdot)$ for distance or cost functions, to highlight the fact that they apply to two sequences. (They may be weighted or unweighted, with homogeneous, affine, or general gap costs.)

Definition 10.4 The following three scoring functions are commonly used.

- The **sum-of-pairs score** is just the sum of the scores of all pairwise projections:

$$S_{\mathrm{SP}}(A) = \sum_{1 \le p < q \le k} S_2(\pi_{\{p,q\}}(A)).$$

- The **weighted sum-of-pairs score** adds a weight factor $w_{i,j} \ge 0$ for each projection:

$$S_{\mathrm{WSP}}(A) = \sum_{1 \le p < q \le k} w_{p,q} \cdot S_2(\pi_{\{p,q\}}(A)).$$

- 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 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. The tree alignment score of such an extended alignment A is the sum of the scores of the projection of each pair of sequences that are connected by an edge in the tree:

$$S^{(T)}_{\mathrm{Tree}}(A) = \sum_{(p,q) \in E(T)} S_2(\pi_{\{p,q\}}(A)),$$

  where $E(T)$ is the set of edges of T.

From a biological point of view, tree scores make more sense than sum-of-pairs scores because they better reflect the fact that the evolution of species, and also the relationship of genes inside the genome of a single species, happens mostly tree-like. That is why often tree-alignment is considered producing better alignments than sum-of-pairs alignment.

We similarly define the **sum-of-pairs cost** $D_{\mathrm{SP}}(A)$, **weighted sum-of-pairs cost** $D_{\mathrm{WSP}}(A)$ and **tree cost** $D^{(T)}_{\mathrm{Tree}}(A)$ of a multiple alignment A. Instead of cost, we also say distance.

We give an example for the distance setting.

Example 10.5 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 $D_2(\cdot)$ is the unit cost function, we get $D_{\mathrm{SP}}(A) = 4 + 2 + 2 = 8$, $D_{\mathrm{WSP}}(A) = 4w_{1,2} + 2w_{1,3} + 2w_{2,3}$, and for an (unrooted) star tree T with the three sequence at its leaves and the sequence $s = \texttt{CGCTGT}$ at the center node, $D^{(T)}_{\mathrm{Tree}}(A) = 1 + 3 + 1 = 5$. ■

Observation 10.6 We point out a design flaw in the sum-of-pairs score. Assume that we have many ($k \approx 1000$) sequences, and all of them share a family-specific motif with a perfectly conserved alignment column (e.g. all `A`s). This is a very strong indication for functional relevance of this particular site.

If we obtain the 1001st sequence, we therefore should align it in the same way, especially we should also see an `A` in that alignment column. Everything else should be severely punished.

However, sum-of-pairs scoring does not really care: Assume an `A-A` match scores $+1$ and any mismatch scores -1. Having 1001 matching `A`s results in a score of $\binom{1001}{2} = 500500$, and 1000 matching `A`s and one mismatching character produce a score of $\binom{1000}{2} - 1000 = 498500$; hardly a big difference.

More precisely, the ratio for one non-matching character out of $k+1$ versus a perfectly conserved column is $\left(\binom{k}{2} - k\right) / \binom{k+1}{2} = (k-3)/(k+1) = 1 - 4/(k+1) \nearrow 1$, as $k \to \infty$.

This is an undesired feature: The more evidence of the column's conservation we collect, the harder we should punish a mismatch; however, the sum-of-pairs score ratio approaches 1 in the limit and therefore makes no distinction between a perfectly conserved column and columns with a constant number of mismatches.

This behavior is due to the quadratic number of terms in the sum-of-pairs score.

10.4 Multiple Alignment Problems

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

Problem 10.7 (Multiple Sequence Alignment Problem) Given k sequences $s_1, s_2, \ldots, s_k$ and an alignment score (cost) function S (D), find an alignment A^{opt} of $s_1, s_2, \ldots, s_k$ such that $S(A^{opt})$ is maximal ($D(A^{opt})$ is minimal) among all possible alignments of $s_1, s_2, \ldots, s_k$.

Such an alignment A^{opt} is called an **optimal alignment**, and $s(s_1, s_2, \ldots, s_k) := S(A^{opt})$ is the **optimal alignment score** ($d(s_1, s_2, \ldots, s_k) := D(A^{opt})$ is the **optimal alignment cost**) of $s_1, s_2, \ldots, s_k$.

Remark. In the case of tree alignment, every alignment involves additional sequences at the inner nodes, as discussed above. To find these sequences is part of the optimization problem! Therefore, the tree alignment problem is more precisely stated in the following.

In the first version of tree alignment that we consider, we assume that the tree structure, by which the given sequences are related, is known. A typical scenario is that the sequences are orthologous proteins from different species whose phylogenetic relationship is well known, as in Figure 10.3

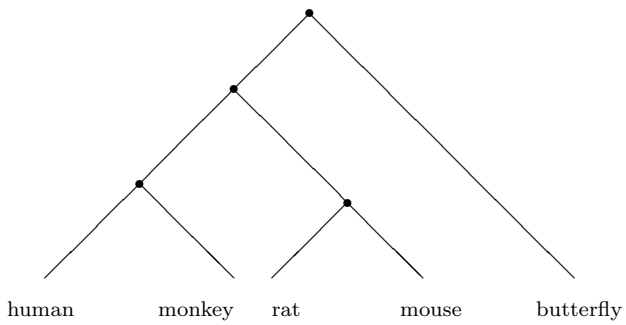
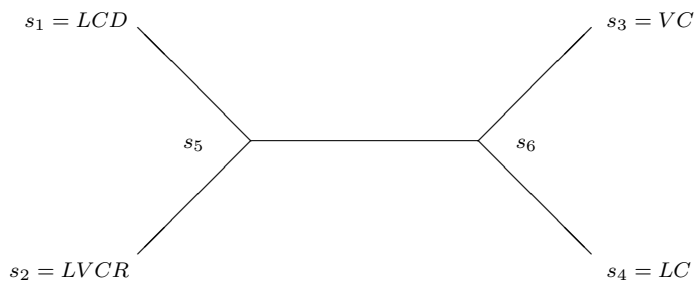


Figure 10.3: A phylogenetic tree

Problem 10.8 (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 score (cost) function S_2 (D_2), find sequences $s_{k+1}, \ldots, s_K$ to be attached to the internal vertices of T and an alignment A of the sequences $s_1, \ldots, s_K$ such that $S^{(T)}_{\text{Tree}}(A)$ is maximal ($D^{(T)}_{\text{Tree}}(A)$ is minimal).

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 attached sequences at internal vertices 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 10.9 Consider sequences $s_1 = LCD$, $s_2 = LVCR$, $s_3 = VC$, $s_4 = LC$ and the following (unrooted) tree with four leaves and two internal nodes:



If the sequences at the internal vertices are chosen as $s_5 = LVCD$ and $s_6 = LVC$ and as pairwise cost function D_2 the unit edit cost function is used, 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

$$\begin{aligned} D^{(T)}_{Tree}(A) &= D_2(A_1, A_5) + D_2(A_2, A_5) + D_2(A_3, A_6) + D_2(A_4, A_6) + D_2(A_5, A_6) \\ &= 1 + 1 + 1 + 1 + 1 \\ &= 5. \end{aligned}$$

■

Hardness. Both (weighted) sum-of-pairs alignment and tree alignment are NP-hard optimization problems (see Section 10.5) with respect to the number of sequences k (Wang and Jiang, 1994). While we will not prove this, 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 10.10 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 cost $D(A^{(1,2)}) = 2$, 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 cost $D(A^{(1,3)}) = 2$.

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 cost $D(A^{((1,2),(1,3))}) = 2 + 2 + 4 = 8$. However, this is not the 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 cost $D(A^{opt}) = 2 + 3 + 2 = 7$. ■

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

Problem 10.11 (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 $D^{(T)}_{\text{Tree}}(A)$ is minimal among all such settings.

It can be solved (theoretically) by solving the tree alignment problem (see Section 12.1) for all tree topologies (of which there are super-exponentially many in k) and picking the best one. An alternative approach is to solve the Steiner tree problem in sequence space, see Section 12.2.

10.5 Digression: NP-completeness

It was stated before that the (weighted) sum-of-pairs alignment and the tree alignment are NP-hard optimization problems with respect to the number of sequences k without further explaining what that means. This section shall now give a brief introduction to the field of NP-completeness about which every good bioinformatician should know. The contents in this section are based on the book of Cormen et al. (2001).

Speaking in an abstract way, a **problem** Q is a binary relation on a set I of **problem instances** and a set S of **problem solutions** (see Figure 10.4). A problem instance can point to no, one or several problem solutions and a problem solution can be one of no, one or several problem instances.

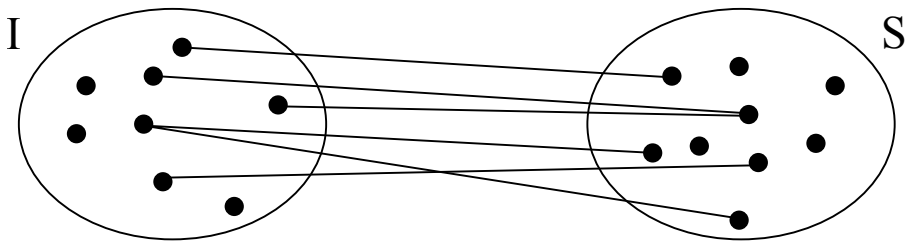


Figure 10.4: Binary relations between problem instances I and problem solutions S.

Example 10.12 A simple example of a problem is integer addition. Then the problem instances could be "$5+3$", "$4+4$" and "$1+4$" pointing to the problem solutions "8", "8" and "5", respectively. ■

An **optimization problem (OP)** or **search problem** addresses the minimization or maximization of a cost function. For example, finding the highest-scoring alignment of two or more sequences is an optimization problem.

A **decision problem (DP)** is a "yes/no question" with just two problem solutions, $S = \{0, 1\}$. The question whether for two given sequences there exists an alignment with a score higher than a constant k is a decision problem.

The discussion about NP-completeness mainly considers decision problems, but the results influence also optimization problems: Searching for a bound, every optimization problem can be stated as a series of decision problems. If a decision problem is difficult, the corresponding optimization problem is usually difficult as well.

Complexity classes. The set of all problems for which (decision) algorithms exist that have the same asymptotic behavior (e.g. running time) are contained in one complexity class. The class **P** contains all problems for which an algorithm exists that can *decide* in polynomial time whether a certain problem instance points to a valid problem solution. The class **NP** contains all problems for which an algorithm exists that can *verify* in polynomial time whether a given certificate (configuration of input variables) is able to solve an actual problem instance, i.e. it can decide in polynomial time whether the certificate is a correct solution to the problem or not. It follows directly that $P \subseteq NP$. The question is still open whether $P \stackrel{?}{=} NP$.

Reducibility. A problem Q is **reducible** to a problem Q' if every instance of Q can be formulated as an instance of Q' so that the solution of problem Q follows from the solution of problem Q'. If Q can be reduced to Q' in a simple way, it is not harder to solve Q than to solve Q'. Q is **polynomial-time reducible** to Q' (shortly: $Q \leq_p Q'$) if the reduction takes only polynomial time. $Q_1 \leq_p Q_2$ implies: $Q_2 \in P \Rightarrow Q_1 \in P$.

NP-hardness, NP-completeness. A problem Q is **NP-hard** if $Q' \leq_p Q$ for every $Q' \in NP$. A problem Q is in the class **NP-complete (NPC)** if it is NP-hard and $Q \in NP$. The following property holds for the class NPC: If some NP-complete problem is solvable in polynomial time, then $P = NP$. If some problem of NP is *not* solvable in polynomial time, then no single NP-complete problem is solvable in polynomial time.

For solving the question whether $P \stackrel{?}{=} NP$, NP-complete problems are intensely studied. The most common assumption is that $P \neq NP$. An alternative possibility is that $P = NP$ (see Figure 10.5 for schematic views of these two alternatives).

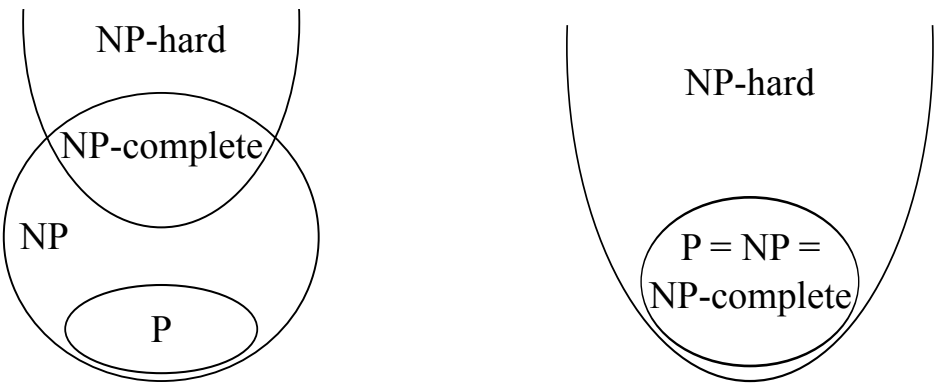


Figure 10.5: Left: $P \neq NP$. Right: $P = NP$.

The satisfiability problem SAT. The first known example of an NP-complete problem was the *satisfiability problem* (SAT). It is stated as follows: Given a boolean formula of length n, is it satisfiable, i.e. is there a configuration of variables so that the evaluation of the formula yields 1?

Example 10.13 Given the formula $(a \wedge b) \vee \bar{a}$, is there a configuration of variables so that the formula is true? Yes, there are even more solutions. One possible solution is $a = 1$ and $b = 1$. ■

To show that SAT is NP-complete, we prove two properties:

1. We show that SAT belongs to the class NP. This is the case, because given a certificate, it is easy to decide in polynomial time whether the evaluation of the formula with these variables indeed yields 1 or not.

2. We show that any problem Q' belonging to the class NP can be reduced to SAT ($Q' \leq_p SAT$) in polynomial time. This can be seen as follows:

- $Q' \in NP$ implies that a verification algorithm $A(x, y)$ exists for Q' that tells in polynomial time for any problem instance $x \in I$ and certificate $y \in S$ whether y is a correct solution of x ($A(x, y) = 1$) or not ($A(x, y) = 0$).

- Let $f(A, x)$ be a reduction function that partially evaluates the verification algorithm A according to its first argument, yielding the function $C(y) = f(A(x, y), x)$. Then we have:

  - Encoded as a boolean function, C is satisfiable ($C(y) = 1$) if and only if y is an optimal solution to x, otherwise $C(y) = 0$.

  - It can be shown that for any $Q' \in NP$ the corresponding boolean function $C(y)$ can be constructed in polynomial time, therefore if SAT can be solved in polynomial time, also Q' can be solved in polynomial time, i.e. $Q' \leq_p$ SAT.

Therefore SAT is an NP-complete problem.

Extension of the class NPC. For showing that some problem Q is in NPC, it is sufficient to show that $Q \in NP$ and to reduce a known NP-complete problem Q' (e.g. Q' = SAT) to Q:

1. Show that $Q \in NP$.

2. Choose a known NP-complete problem Q'.

3. Describe an algorithm that calculates a function f that projects every instance of Q' to an instance of Q.

4. Show that for f holds: $x \in Q'$ if and only if $f(x) \in Q$ for all $x \in \{0, 1\}^*$.

5. Show that f can be calculated in polynomial time.

Consequences of NP completeness. While the proof of NP-completeness of a problem implies that it is quite unlikely to find a deterministic polynomial-time algorithm to solve the problem to optimality in general, there are different ways to respond to such a result if one wants to solve the problem in feasible time:

1. Small problem instances might be solvable anyway.
2. Using adequate techniques to efficiently prune the search space, even larger instances may be solvable in reasonable time (**running time heuristics**).
3. The problem may not be hard for certain subclasses, so polynomial-time algorithms may exist for them, e.g. FPT (**Fixed Parameter Tractability**).
4. It may be sufficient to approximate the optimal solution to a certain degree. If closeness to the optimal solution can be guaranteed, one speaks of an **approximation algorithm**.
5. By no longer confining oneself to any optimality guarantee, one may get very fast heuristic algorithms that produce good but not always optimal or guaranteed close-to-optimal solutions (**correctness heuristics**).

10.6 A Guide to Multiple Sequence Alignment Algorithms

Sum-of-pairs alignment algorithms are discussed in Chapter 11. We first explain a multi-dimensional generalization of the pairwise dynamic programming (Needleman-Wunsch) algorithm. We next show how to reduce the search space considerably (according to an idea of Carrillo and Lipman). A simple 2-approximation is given by the center-star method. Finally, we discuss a divide-and-conquer heuristic.

Tree alignment and generalized tree alignment are discussed in Chapter 12. We discuss Sankoff's tree alignment algorithm and two heuristics for generalized tree alignment (Greedy Three-Way alignment and the Deferred Path Heuristic).

As mentioned above, the multiple alignment problem is NP hard and all known algorithms have running times exponential in the number of sequences; thus aligning more than 7 sequences of reasonable length is problematic. Since in practice, one needs to produce multiple alignments of (sometimes) hundreds of sequences, heuristics are required: **Progressive alignment** methods, discussed in Chapter H, 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.

CHAPTER 11

Algorithms for Sum-of-Pairs Multiple Alignment

Contents of this chapter: Basic algorithm, free end gaps, affine gap costs, Carrilo-Lipman, bounding pairwise alignment costs, pruning the alignment graph, Center-Star approximation, Divide-and-Conquer alignment, multiple additional cost, DCA algorithm.

Sum-of-pairs alignment has been extensively studied. However, be aware that from a biological point of view and because of Observation 10.6, tree alignment should be the preferred choice.

While we only discuss the unweighted sum-of-pairs alignment problem explicitly, everything in this chapter also applies to weighted sum-of-pairs.

11.1 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.

11.1.1 The Basic Algorithm

Instead of a two-dimensional edit graph, a k-dimensional weighted grid graph is constructed, one dimension for each sequence. Each edge e corresponds to a possible alignment column c, weighted by its corresponding alignment score $w(e) = D(c)$, and the optimal alignment problem translates to the problem of finding a shortest path from the source vertex $v_\circ$ to the sink vertex $v_\bullet$. The graph is illustrated in Figure 11.1.

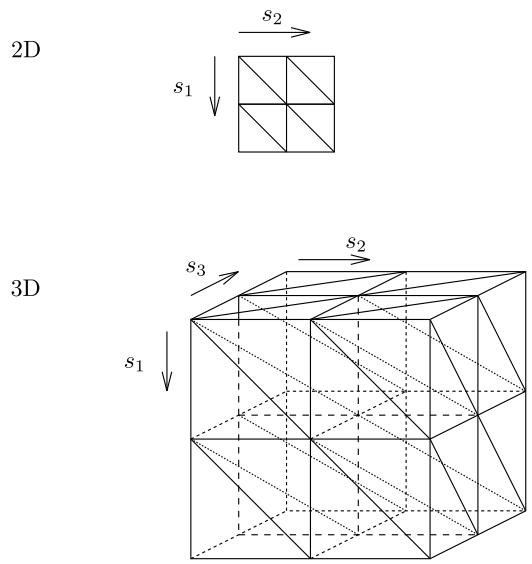


Figure 11.1: Top: Part of the 2-dimensional alignment graph for two sequences. Bottom: Part of the 3-dimensional alignment graph for three sequences. Not shown: Parts of the k-dimensional alignment graphs for $k \geq 4$.

Intuitively, for each vertex v in the edit graph we compute

$$D(v) = \min\{D(v') + w(v' \to v) \mid v' \text{ is a predecessor of } v\}. \tag{11.1}$$

More formally, this can be written as follows:

$$\begin{aligned} D(0,0,\ldots,0) &= 0 \\ D(\overbrace{i_1, i_2, \ldots, i_k}^{v}) &= \min_{\substack{\Delta_1,\ldots,\Delta_k \in \{0,1\} \\ \Delta_1+\ldots+\Delta_k \neq 0}} \left\{ D(\overbrace{i_1-\Delta_1, i_2-\Delta_2, \ldots, i_k-\Delta_k}^{\text{predecessor } v'}) + D_{\text{SP}} \overbrace{\begin{pmatrix} \Delta_1 s_1[i_1] \\ \Delta_2 s_2[i_2] \\ \ldots \\ \Delta_k s_k[i_k] \end{pmatrix}}^{\text{alignm. col.}} \right\} \end{aligned}$$

where for a character $c \in \Sigma$ the notation Δc denotes $\Delta c = c$ if $\Delta = 1$ and $\Delta c =$ '$-$' if $\Delta = 0$.

Space and time complexity. The space complexity of this algorithm is given by the size of the k-dimensional edit graph. 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 a maximization is taking place over $2^k - 1$ incoming edges. Hence, the total time complexity (for homogeneous gap costs) is $O(2^k \cdot n^k)$. For affine gap costs, the time complexity rises even more (see below).

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, five or six typical protein sequences. Even worse, while in the pairwise case the space usage can be reduced from quadratic to linear by only a little time overhead, the possible space reduction in the case of multiple alignment is rather small, as we now discuss.

11.1.2 Variations of the Basic Algorithm

Memory reduction. From the pairwise alignment, we remember that it is possible to reduce the space for optimal alignment computation from quadratic to linear using a divide-and-conquer approach (Section 5.4).

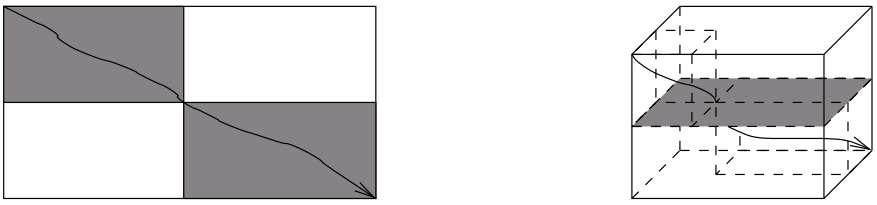


Figure 11.2: Divide-and-conquer space reduction technique for multiple alignment.

In priciple, the same idea can also be followed for multiple alignment, but the effect is much less dramatic. The space reduction is by one dimension, from $O(n^k)$ to $O(n^{k-1})$. That the space savings here are less impressive than for pairwise alignment becomes clear when one considers that for $k = 12$ sequences, the space usage is still $O(n^{11})$, using this technique.

Figure 11.2 sketches on the left side the first division step for the pairwise case, and on the right side the corresponding step in the multiple (here 3-dimensional) case.

Free end gaps. It is sometimes desired to penalize gaps at the beginning and the end of an alignment less than internal gaps (or not at all, also called free end-gap alignment), in order to avoid that shorter sequences are "spread" over longer ones, just to get a slightly higher score that is not counter-penalized by gap costs, because these have to be imposed anyway to account for the difference in sequence length.

This can be done in multiple alignment as in the pairwise case. It is just a different initialization of the base cases, i.e. the edges of the k-dimensional edit graph.

Affine gap costs. Affine gap costs can be defined in a straightforward manner as a simple generalization of the pairwise case, called **natural gap costs**: The cost of a multiple alignment is the sum of all costs of the pairwise projections, where the cost of a pairwise projection is computed with affine gap costs. In principle it is possible to perform the computation of such alignments similarly as it is done for pairwise alignments. However, the number of "history matrices" (V and H in the pairwise case) grows exponentially with the number of sequences, so that very much space is needed.

That is why Altschul (1989) suggested to use an approximation of the exact case, so-called **quasi-natural gap costs**. The idea is to look back by only one position in the edit graph and decide from the previous column in the alignment if a new gap is started in the column to be computed or not. The choices, compared to the "correct" choices in natural gap costs, are given in the following table whose first line depicts the different possible scenarios of pairwise alignment projections where an asterisk (`*`) refers to a character, a dash (`-`) refers to a blank, and a question mark (`?`) refers to either a character or a blank. The entries correspond to the cost (d is gap open cost, e is gap extension cost), where "no" means no new gap is opened (substitution cost):

	?* ?*	?- ?-	** *-	** --	-* *-	-* --
natural	no	0	d	e	d	d/e
quasi-natural	no	0	d	e	d	d

The last case in natural gap costs cannot be decided if only one previous column is given. It would be d for `*---*` / `*----` and e for `***--*` / `*-----` .

With this heuristic, the overall computation becomes slower by a factor of 2^k, yielding a time complexity for the complete multiple alignment computation of $O(n^k 2^{2k})$.

The space requirements do not change in the worst case, they remain $O(n^k)$.

11.2 Carrillo and Lipman's Search Space Reduction

Carrillo and Lipman (1988) suggested an effective way to prune the search space for multiple alignment. Their algorithm is still exponential, but in many cases the time of alignment computation is considerably reduced compared to the full dynamic programming algorithm from Section 11.1, making it applicable in practice for data sets of moderate size (7–10 protein sequences of length 300-400).

Let sequences $s_1, s_2, \ldots, s_k$ be given. The algorithm of Carrillo and Lipman is a branch-and-bound running time heuristic, based on a simple idea and observation.

Idea. As we have seen in Example 10.10, the pairwise projections of an optimal multiple alignment are not necessarily the optimal pairwise alignments (making the problem hard). However, intuitively, they can also not be particularly bad alignments. (If all pairwise projections were bad, the whole multiple alignment would also have a bad sum-of-pairs score.)

If we can quantify this idea, we can restrict the search space to those vertices that are part of close-to-optimal pairwise alignments for each pairwise projection.

Bounding pairwise Alignment costs. Here we derive the Carrillo-Lipman bound in the distance or cost setting. The same can be done (with reversed inequalities) in the score setting. This is left as an exercise.

We assume that we already know some (suboptimal) alignment A of the given sequences with cost $D_{\mathrm{SP}}(A) =: \delta \geq D_{\mathrm{SP}}(A^{opt})$, for example computed by the center-star heuristic (Section 11.3) or a progressive alignment method (Section H).

Remember the definition of the sum-of-pairs multiple alignment score:

$$D_{\mathrm{SP}}(A) = \sum_{p<q} D_2(\pi_{\{p,q\}}(A)).$$

We now pick a particular index pair (x, y), $x < y$ and remove it from the sum. In the remaining pairs, we use the fact that the pairwise projections of the optimal multiple

alignment cannot have a lower cost than the optimal corresponding pairwise alignments: $D_2(\pi_{\{p,q\}}(A^{opt})) \geq d(s_p, s_q)$. Thus

$$\begin{aligned}
\delta &\geq D_{\mathrm{SP}}(A^{opt}) \\
&= \sum_{p<q} D_2(\pi_{\{p,q\}}(A^{opt})) \\
&= D_2(\pi_{\{x,y\}}(A^{opt})) + \sum_{\substack{p<q \\ (p,q)\neq(x,y)}} D_2(\pi_{\{p,q\}}(A^{opt})) \\
&\geq D_2(\pi_{\{x,y\}}(A^{opt})) + \sum_{\substack{p<q \\ (p,q)\neq(x,y)}} d(s_p, s_q)
\end{aligned}$$

for any pair (x, y), $1 \leq x < y \leq k$.

This implies the following upper bound for the cost of the projection of an optimal multiple alignment on rows x and y:

$$D_2(\pi_{\{x,y\}}(A^{opt})) \;\leq\; \delta - \sum_{\substack{p<q \\ (p,q)\neq(x,y)}} d(s_p, s_q) \;=:\; U_{(x,y)}.$$

Note that in order to compute the upper bound $U_{(x,y)}$, only pairwise optimal alignments and a heuristic multiple alignment need to be calculated. This can be done efficiently.

Pruning the Alignment graph. We now keep only those vertices $(i_1, \ldots, i_k)$ of the alignment graph that satisfy the following condition for *all* pairs $(x, y), 1 \leq x < y \leq k$:

The best pairwise alignment of s_x and s_y through (i_x, i_y) has cost $\leq U_{(x,y)}$.

How do we know the cost of the best pairwise alignment through any given point (i_x, i_y)? This question was answered in Section 5.3: we use the forward-backward techniqe. In other words, for every pair (x, y), $x < y$, we compute a matrix $T_{(x,y)}$ such that $T_{(x,y)}(i_x, i_y)$ contains the cost of the best alignment of s_x and s_y that passes through the point (i_x, i_y).

We can define the set of vertices in the multi-dimensional alignment graph that satisfy the constraint imposed by the pair (x, y) as follows:

$$B_{(x,y)} := \{(i_1, \ldots, i_k) : T_{(x,y)}(i_x, i_y) \leq U_{(x,y)}\}$$

From the above considerations it is clear that the projection of an SP-optimal multiple alignment on sequences s_x and s_y can not have a cost higher than $U_{(x,y)}$, implying that in the picture of $T_{(x,y)}$ in Figure 11.3, the shaded regions with values $T_{(x,y)}(i, j) > U_{(x,y)}$ cannot contain such a projection.

Thus the optimal multiple alignment consists at most of vertices that are in $B_{(x,y)}$ for *all pairs* (x, y), i.e., in

$$B := \bigcap_{x<y} B_{(x,y)}.$$

This means: Instead of computing the distances in all $O(n^k)$ vertices in the alignment graph, we only need to compute the distances of $|B|$ vertices. The size of B depends on many things, e.g. on

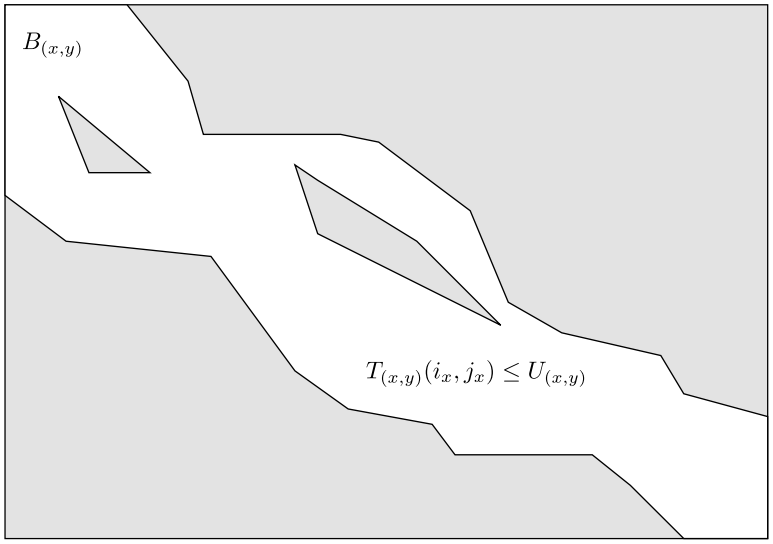


Figure 11.3: The Carrillo-Lipman bound states that the two-dimensional projection of an optimal multiple alignment cannot pass through the shaded region.

- the quality of the heuristic alignment with cost δ that serves as an upper bound of the optimal alignment cost. The closer δ is to the true optimal value, the smaller is B;
- the difference between the optimal pairwise alignment costs and the costs of the pairwise projections of the optimal multiple alignment.

Generally, if all k sequences are similar and the gap costs are small, then a good heuristic alignment (maybe even an optimal one) is easy to find, so B may contain almost no additional vertices besides the true optimal multiple alignment path. Thus, for a medium number of similar sequences, the simple Carrillo-Lipman heuristic may already perform very well (because the relevant regions become very small) while for even fewer but less closely related sequences even the best known heuristics will not finish within weeks of computation. Sequence similarity plays a much higher role than sequence length or the number of sequences.

Some implementations of the Carrillo-Lipman bound attempt to reduce the size of B further by "cheating": Remember that we need a heuristic alignment to obtain an upper bound δ on the optimal cost. If we have reason to suspect that we have found a bad alignment, and believe that a better one exists, we may replace δ by a smaller value (which could even be chosen differently for each pair (x, y)) $\delta - \epsilon_{(x,y)}$. Of course, since we cannot be sure that such a better alignment exists, we may reduce the size of B too much and in fact exclude the optimal multiple alignment from B. In practice, however, this kind of cheating works quite well if done carefully, and further reduces the running time.

Algorithm. We give a few remarks about implementing the above idea. We emphasize that the set B is conceptual and does not need to be constructed explicitly. Instead, it is sufficient to precompute all $T_{(x,y)}$ matrices, the optimal pairwise alignment scores $d(s_p, s_q)$ for all $1 \leq p < q \leq k$, and a heuristic alignment to obain δ.

It is advantageous not to implement the "pull-type" recurrence (11.1), but instead to use a "push-type" algorithm, as explained in Section 3.6.

Conceptually, we can imagine that every vertex has a distance of ∞ at the beginning of the algorithm, but we do not store this information explicitly. Instead, we maintain a list (e.g. a queue or priority queue) of "active" vertices, that initially contains only $(0, 0, \ldots, 0)$.

The algorithm then consists of "visiting" a vertex from the queue until the final vertex $(n_1, n_2, \ldots, n_k)$ is visited. When a vertex is visited, it already contains the correct alignment distance.

Visiting a vertex v consists of the following steps.

- We look at the $2^k - 1$ successors w of v in the alignment graph and check for each of them if it belongs to the set B. This is the case if w is already in the queue, or if the current distance value plus edge cost $v \to w$ satisfies the Carrillo-Lipman bound for every index pair (x, y). In the former case, the distance (and backpointer) information of w is updated if the new distance value coming from v is lower than the current one. In the latter case w is "activated" with the appropriate distance and backpointer information and added to the queue.

- We remove v from the queue. If we want to create an alignment in the end, we need to store v's information somewhere else to reconstruct the traceback information. If we only want the score, v can now be completely discarded.

- We pick the next vertex x to visit. The algorithm works correctly if x is

  - a vertex of minimum distance; then the algorithm is essentially **Dijkstra's algorithm** for single-source shortest paths in the (reduced) alignment graph, or

  - any vertex that has no active ancestors, so we can be sure that we do not need to update x's distance information after visiting x.

  Since often more than one vertex is available for visiting, a good target-driven choice may speed up the algorithm.

Implementations. An implementation of the Carrillo-Lipman heuristic is the program MSA (Gupta et al., 1995).

A newer implementation of many of the mentioned algorithms is the program QALIGN (Sammeth et al., 2003b), see `http://gi.cebitec.uni-bielefeld.de/QAlign`. This program not only runs much more stable than MSA, it also has a very nice and flexible graphical user interface.

11.3 The Center-Star Approximation

There exists a series of results on the approximability of the sum-of-pairs multiple sequence alignment problem.

Definition 11.1 For $c \geq 1$, an algorithm for a cost or distance minimization problem is called a ***c*-approximation** if its output solution has cost at most c times the optimal solution.

For a maximization problem, an algorithm is called a c-approximation if its output solution has score at least $1/c$ times the optimal solution.

A simple algorithm, the so-called *center star method* (Gusfield, 1991, 1993), is a 2-approximation for the sum-of-pairs multiple alignment problem if the underlying weighted edit distance satisfies the triangle inequality.

The algorithm is the following. First, for each sequence s_p, $1 \leq p \leq k$, its overall distance d_p to the other sequences is computed, i.e. the sum of the pairwise optimal alignment costs:

$$d_p = \sum_{1 \leq q \leq k} d(s_p, s_q).$$

Let s_c be the sequence that minimizes this overall distance, called the *center sequence*.

Secondly, a multiple alignment A_c is constructed from all pairwise optimal alignments where the center sequence is involved, i.e., all the optimal alignments of s_c and the other s_p, $p \neq c$, are combined into one multiple alignment A_c.

The claim is that this alignment is a 2-approximation for the optimal sum-of-pairs multiple alignment score of the sequences $s_1, s_2, \ldots, s_k$, i.e.,

$$D_{\mathrm{SP}}(A_c) \leq 2 \cdot d_{\mathrm{SP}}(s_1, s_2, \ldots, s_k).$$

This can be seen as follows:

$$\begin{aligned}
D_{\mathrm{SP}}(A_c) &= \sum_{p<q} D_2(\pi_{\{p,q\}}(A_c)) && \text{(definition)} \\
&= \frac{1}{2} \sum_{(p,q)} D_2(\pi_{\{p,q\}}(A_c)) && \text{(since } D_2(\pi_{\{p,p\}}) = 0\text{)} \\
&\leq \frac{1}{2} \sum_{(p,q)} \big(D_2(\pi_{\{p,c\}}(A_c)) + D_2(\pi_{\{c,q\}}(A_c))\big) && \text{(triangle inequality!)} \\
&= \frac{1}{2} \sum_{(p,q)} (d(s_p, s_c) + d(s_c, s_q)) && \text{(alignments to } c \text{ are optimal)} \\
&= k \cdot \sum_{q} d(s_c, s_q) && \text{(star property)} \\
&\leq \sum_{(p,q)} d(s_p, s_q) && \text{(minimal choice of } c\text{)} \\
&\leq \sum_{(p,q)} D_2(\pi_{\{p,q\}}(A^{opt})) \\
&= 2 \cdot \sum_{p<q} D_2(\pi_{\{p,q\}}(A^{opt})) \\
&= 2 \cdot D_{\mathrm{SP}}(A^{opt})
\end{aligned}$$

Hence A_c is a 2-approximation of the optimal alignment A^{opt}.

Time and space complexity. The running time of the algorithm can be estimated as follows, assuming all k sequences have the same length n:

In the first phase, $\binom{k}{2}$ pairwise alignments are computed, requiring overall $O(k^2n^2)$ time. In the second phase, the $k-1$ alignments are combined into one multiple alignment which can be done in time proportional to the size of the constructed alignment, i.e. $O(kn)$. This yields an overall running time of $O(k^2n^2)$.

The space complexity is $O(n+k)$ in the first phase (linear-space score computation plus storage for the k computed values d_p) and $O(kn)$ in the second phase (storage of k pairwise alignments, computed with Hirschberg's linear space algorithm, and the intermediate/final multiple alignment). Hence, the overall space complexity is $O(kn)$.

This idea can be generalized to a $2-\frac{\kappa}{k}$-approximation if instead of pairwise alignments, optimal multiple alignments of κ sequences are computed (see Bafna et al. (1994), Jiang et al. (1994) and Wang et al. (1996)).

11.4 Divide-and-Conquer Alignment

In this section we present a heuristic for sum-of-pairs multiple sequence alignment that no longer guarantees that an optimal alignment will be found.

While in the worst case the running time of the heuristic is still exponential in the number of sequences, the advantage is that on average the practical running time is much faster while the result is very often still optimal or close to optimal.

Idea. The basic idea of the heuristic is to cut all sequences at suitable cut points into left and right parts, and then to solve the two such generated alignment problems for the left parts and the right parts separately. This is done either by recursion or, if the sequences are short enough, by an exact algorithm. For a graphical sketch of the procedure, see Figure 11.4.

Finding cut positions. The main question about this procedure is how to choose good cut positions, since this choice is obviously critical for the quality of the final alignment. The ideal case is easy to formulate:

Definition 11.2 A family of cut positions $(c_1, c_2, \ldots, c_k)$ of the sequences $s_1, s_2, \ldots, s_k$ of lengths $n_1, n_2, \ldots, n_k$, respectively, is called an **optimal cut** if and only if there exists an alignment A of the prefixes $(s_1[1..c_1], s_2[1..c_2], \ldots, s_k[1..c_k])$ and an alignment B of the suffixes $(s_1[c_1+1..n_1], s_2[c_2+1..n_2], \ldots, s_k[c_k+1..n_k])$ such that the concatenation $A +\!\!+ B$ is an optimal alignment of $s_1, s_2, \ldots, s_k$.

There are many optimal cut positions, e.g. the trivial ones $(0, 0, \ldots, 0)$ and $(|s_1|, |s_2|, \ldots, |s_k|)$. In fact:

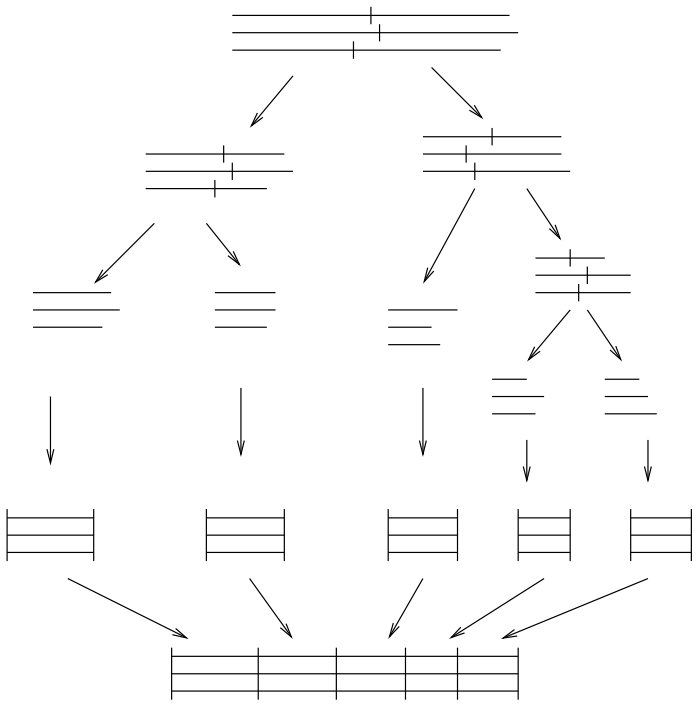


Figure 11.4: Overview of divide-and-conquer multiple sequence alignment.

Lemma 11.3 For any cut position $\hat{c}_1 \in \{0, \ldots, n_1\}$ of the first sequence s_1, there exist cut positions $c_2, \ldots, c_k$ of the remaining sequences $s_2, \ldots, s_k$ such that $(\hat{c}_1, c_2, \ldots, c_k)$ is an optimal cut of the sequences $s_1, s_2, \ldots, s_k$.

Proof. Assume that A^{opt} is an optimal alignment. Choose one of the points between the characters in the first row A_1^{opt} that correspond to the characters $s_1[\hat{c}_1]$ and $s_1[\hat{c}_1 + 1]$. The vertical "cut" at this point through the optimal alignment defines the cut positions $c_2, \ldots, c_k$ that, together with $\hat{c}_1$, are optimal by definition. See Figure 11.5 for an illustration. □

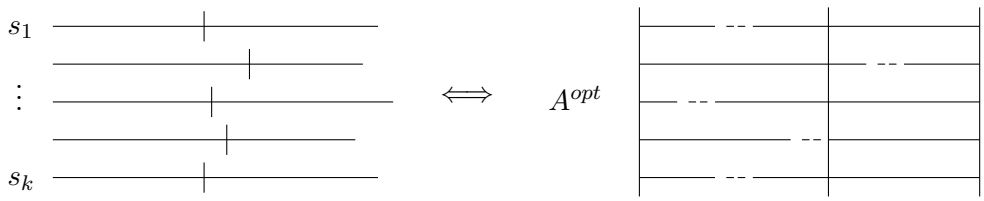


Figure 11.5: Connection between unaligned sequences $s_1, \ldots, s_k$ (left) and a multiple alignment A of them (right). If A is an optimal alignment A^{opt}, any vertical cut in the alignment corresponds to an optimal family of cut positions of the sequences.

The lemma tells us that a first cut position $\hat{c}_1$ of s_1 can be chosen arbitrarily (for symmetry reasons and to arrive at an efficient divide-and-conquer procedure, we will use $\lceil n_1/2 \rceil$ as the cut position), and then there exist cut positions of the other sequences that produce an optimal cut.

The NP-completeness of the SP-alignment problem implies, however, that it is unlikely that there exists a polynomial time algorithm for finding optimal cuts. Hence a heuristic is used to find good, though not always optimal cut positions.

This heuristic is based on pairwise sequence comparisons whose results are stored in an *additional cost matrix* that contains for each pair of cut positions (c_p, c_q) the penalty (additional cost) imposed by cutting the two sequences at these points, instead of cutting them at points compatible with an optimal pairwise alignment (see Definition 5.8). The efficient computation of additional cost matrices was discussed in Section 5.3.

The idea of using additional cost matrices to quantify the quality of a multiple-sequence cut is that the closer a potential cut point is to an optimal pairwise alignment path, the smaller will be the contribution of this pair to the overall sum-of-pairs multiple alignment cost. Since we are dealing with sum-of-pairs alignment, we define the additional cost of a family of cuts along the same rationale.

Definition 11.4 The **multiple additional cost** of a family of cut positions $(c_1, c_2, \ldots, c_k)$ is defined as

$$C(c_1, c_2, .., c_k) := \sum_{1 \le p < q \le k} C_{(p,q)}(c_p, c_q),$$

where $C_{(p,q)}$ is the additional cost matrix for pair $(p, q),\ p < q$.

A family of cut positions $(c_1, c_2, \ldots, c_k)$ is called ***C*-optimal** if it minimizes the multiple additional cost $C(c_1, c_2, \ldots, c_k)$.

Although maybe unintuitive, even a family of cut positions with zero multiple additional cost, i.e. one whose implied pairwise cuts are all compatible with the corresponding pairwise optimal alignments, is not necessarily optimal (see Example 11.5). The converse is also not true: neither has an optimal cut always multiple additional cost zero, nor is a cut with smallest possible additional cost necessarily optimal.

Example 11.5 Let $s_1 = CT$, $s_2 = AGT$, and $s_3 = G$. Using unit cost edit distance, the three additional cost matrices are the following:

$C_{(1,2)}$:

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

$C_{(1,3)}$:

	G	
C	0	1
T	0	0
	1	0

$C_{(2,3)}$:

	G	
A	0	2
G	0	1
T	1	0
	2	0

There are two C-optimal cuts: $(1, 1, 0)$ and $(1, 2, 1)$, both of which yield a multiple additional cost $C(1, 1, 0) = C(1, 2, 1) = 0$. Nevertheless, the implied multiple alignments are not necessarily optimal as can be seen for the cut $(1, 1, 0)$:

$$\begin{array}{c|c} C & T \\ A & GT \\ \varepsilon & G \end{array} \quad \rightarrow \quad \begin{pmatrix} C \\ A \\ - \end{pmatrix} +\!\!+ \begin{pmatrix} - & T \\ G & T \\ G & - \end{pmatrix} = \begin{pmatrix} C & - & T \\ A & G & T \\ - & G & - \end{pmatrix} \quad \text{(SP-cost 7, not optimal).}$$

But they might be optimal, as for the cut $(1, 2, 1)$:

$$\begin{array}{rcl} C & | & T \\ AG & | & T \\ G & | & \varepsilon \end{array} \quad \rightarrow \quad \begin{pmatrix} - & C \\ A & G \\ - & G \end{pmatrix} +\!\!+ \begin{pmatrix} T \\ T \\ - \end{pmatrix} = \begin{pmatrix} - & C & T \\ A & G & T \\ - & G & - \end{pmatrix} \quad \text{(SP-cost 6, optimal).}$$

■

Nevertheless, it is a good heuristic to choose a C-optimal cut.

DCA Algorithm. The divide-and-conquer alignment algorithm first chooses an arbitrary cut position $\hat{c}_1$ for the first sequence (usually the sequence is cut in half for symmetry reasons) and then searches for the remaining cut positions $c_2, \ldots, c_k$ such that the cut $(\hat{c}_1, c_2, \ldots, c_k)$ is C-optimal.

Then the sequences are cut in this way, and the procedure is repeated recursively until the maximal sequence length drops below a threshold value L, when the exact multiple sequence alignment algorithm MSA (Carrillo-Lipman bounds) is applied.

The pseudocode for this procedure is given in Algorithm 11.1.

Algorithm 11.1 $\boldsymbol{DCA(s_1, s_2, \ldots, s_k; L)}$

```
Let n_p ← |s_p| for all p, 1 ≤ p ≤ k
if max{n_1, n_2, ..., n_k} ≤ L then
    return MSA(s_1, s_2, ..., s_k)
else
    ĉ_1 ← ⌈n_1/2⌉
    compute (c_2, ..., c_k) such that (ĉ_1, c_2, ..., c_k) is a C-optimal cut
    return DCA(s_1[1..ĉ_1], s_2[1..c_2], ..., s_k[1..c_k]; L) ++ DCA(s_1[ĉ_1+1..n_1], s_2[c_2+1..n_2],
        ..., s_k[c_k+1..n_k]; L)
```

The search space in the computation of C-optimal cut positions is sketched in Figure 11.6. Since the cut position of the first sequence s_1 is fixed to $\hat{c}_1$ but the others can vary over their whole sequence length, the search space has size $O(n_2 \cdot n_3 \cdots n_k)$. This implies that the time for computing a C-optimal cut (i.e. finding the minimum in this search space) by exhaustive search takes $O(k^2 n^2 + n^{k-1})$ time and uses $O(k^2 n^2)$ space.

Overall complexity. Let L be the length bound below which the exact MSA algorithm is applied. Assuming that the maximum sequence length n is of the form $n = L \cdot 2^D$ and all cuts are symmetric (i.e. $n_{(d)} = n/2^d$ for all $d = 0, \ldots, D-1$), the whole divide-and-conquer alignment procedure has the overall time complexity

$$\begin{aligned} \left(\sum_{d=0}^{D-1} 2^d \left(k^2 n_{(d)}^2 + n_{(d)}^{k-1}\right)\right) + \frac{n}{L}(2^k L^k) &= \left(\sum_{d=0}^{D-1} 2^d \left(k^2 \frac{n^2}{2^{2d}} + \frac{n^{k-1}}{2^{d(k-1)}}\right)\right) + 2^k n L^{k-1} \\ &= \left(k^2 n^2 + \frac{n^{k-1}}{2^{k-2}}\right)\left(\sum_{d=0}^{D-1} \frac{1}{2^d}\right) + 2^k n L^{k-1} \\ &\in O(k^2 n^2 + n^{k-1} + 2^k n L^{k-1}) \end{aligned}$$

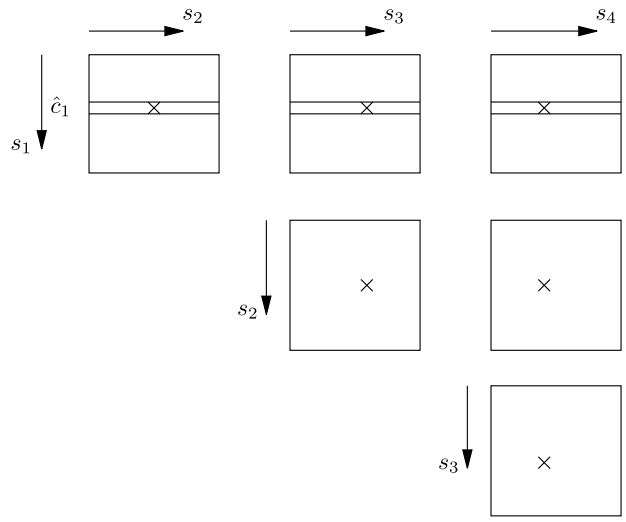


Figure 11.6: Calculation of C-optimal cuts.

and space complexity

$$\max_{0 \le d < D} \left\{ kn_{(d)} + \binom{k-1}{2} n_{(d)}^2 \right\} + L^k \quad \in \quad O(k^2 n^2 + L^k).$$

Compared to the standard dynamic programming for multiple alignment, the time is reduced only by one dimension, so it is not clear if the whole method is really an advantage. However:

- The space usage is polynomial, since only the space for the $\binom{k}{2}$ additional cost matrices is required.
- Additional cost matrices have a very nice structure, since the low values that are of interest here are often close to the main diagonal, and the values increase rapidly when one leaves this diagonal. Based on this observation, several branch-and-bound heuristics have been developed to speed-up the search for C-optimal cut positions, for more details see (Stoye, 1998).

An implementation of the divide-and-conquer alignment algorithm, plus further documentation, can be found at `http://bibiserv.techfak.uni-bielefeld.de/dca`; another implementation is part of the QAlign package.

CHAPTER 12

Tree Alignment Algorithms

Contents of this chapter: Sankoff, Minimum Mutation Problem, Fitch, generalized tree alignment, Steiner tree, three-way tree alignment construction, deferred path heuristic.

12.1 Sankoff's Algorithm

An exponential-time exact algorithm that solves the tree alignment problem (Problem 10.8) is due to Sankoff (1975).

Note that for a given alignment A the tree alignment cost with homogeneous gap costs can be equivalently written in the following column-wise form:

$$D^{(T)}_{\text{Tree}}(A) = \sum_{\{p,q\}\in E} D_2(\pi_{\{p,q\}}(A)) = \sum_{\text{column } j \text{ of } A} D^{(T)}(A_{*,j}),$$

where the cost of the jth alignment column of A is given by

$$D^{(T)}(A_{*,j}) := \sum_{\{p,q\}\in E} cost(A_{p,j}, A_{q,j})$$

and *cost* is a dissimilarity function on characters (and gaps) satisfying $cost(-,-) = 0$.

Then the optimal alignment cost for sequence prefixes $s_1[1..i_1], s_2[1..i_2], \ldots, s_k[1..i_k]$ can be computed by the recursive formula

$$D(i_1, i_2, \ldots, i_k)$$
$$= \min_{\substack{\Delta_1,\ldots,\Delta_k \in \{0,1\} \\ \Delta_1+\ldots+\Delta_k \neq 0}} \left\{ D(i_1 - \Delta_1, i_2 - \Delta_2, \ldots, i_k - \Delta_k) + \min_{c_{k+1},\ldots,c_K \in \Sigma \cup \{-\}} D^{(T)} \begin{pmatrix} \Delta_1 s_1[i_1] \\ \ldots \\ \Delta_k s_k[i_k] \\ c_{k+1} \\ \ldots \\ c_K \end{pmatrix} \right\}$$

where we use the notation $\Delta c := c$ if $\Delta = 1$ and $\Delta c := -$ if $\Delta = 0$ for a character $c \in \Sigma$ and, as in Definition 10.4, K is the number of vertices of the tree T.

The outer minimization alone is computationally intensive: Like for sum-of-pairs optimal multiple alignment, each entry in the k-dimensional search space of the indices $(i_1, i_2, \ldots, i_k)$ has $2^k - 1$ predecessors.

Moreover, there is the inner minimization where the optimal characters for the sequences at the $K - k$ internal nodes of the tree are to be selected:

Problem 12.1 (Minimum Mutation Problem) Given a phylogenetic tree T with characters attached to the leaves, find a labelling of the internal vertices 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
   $$x_j = \begin{cases} x_i, & \text{if } x_i \in R_j \\ \text{some } x \in R_j, & \text{otherwise.} \end{cases}$$

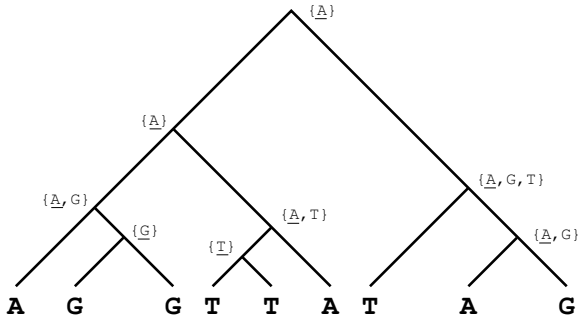


Figure 12.1: Illustration of the Fitch algorithm for $\Sigma = \{A, G, T\}$. The characters assigned during the backtracing phase are underlined.

See Figure 12.1 for an example of the Fitch algorithm.

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.

Since in the overall algorithm we have that for each of the $O(n^k)$ entries $(i_1, \ldots, i_k)$ in the alignment search space there are $2^k - 1$ choices for the $(\Delta_1, \Delta_2, \ldots, \Delta_k)$, and for each of them we run the Fitch algorithm, the overall time complexity of Sankoff's algorithm is $O(n^k 2^k k|\Sigma|)$ time.

Knudsen (2003) shows how this algorithm can be generalized to affine gap costs.

Note: 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 it recursively traverses the subtrees before visiting the node itself.

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 it recursively visits the node first and then traverses the subtrees.

12.2 Generalized Tree Alignment

The generalized tree alignment problem (Problem 10.11) can be equivalently formulated as a Steiner tree problem in sequence space:

Problem 12.2 Given the complete weighted graph whose vertices V represent the infinite set Σ^* of all sequences over Σ and whose edges have as weight the edit distance between the connected vertices (the so-called *Sequence Space*), and a 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 minimal. Such a tree is called a **Steiner tree** for V'.

For two-letter sequences over the alphabet $\Sigma = \{A, C, G, T\}$, this is visualized in Figure 12.2.

Generalized tree alignment is a very hard, but very attractive problem to solve. For example, the chicken-and-egg problem of alignment and tree construction is avoided; see also Section H.1. Therefore heuristic methods have been developed, two of which are presented here.

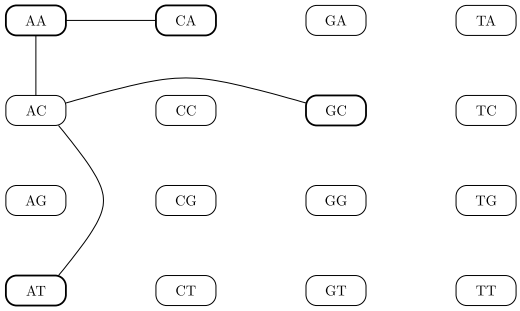


Figure 12.2: The generalized tree alignment problem for two-letter sequences as a Steiner tree problem in sequence space. Given are the sequences AA, CA, AT, and GC. The tree shown contains one additional internal "Steiner" node AC and has total cost 4.

12.2.1 Greedy Three-Way Tree Alignment Construction

This heuristic method was developed by Vingron and von Haeseler (1997).

The idea is to construct alignment and tree simultaneously by iterative insertion of leaves into a growing tree, resulting in a series of trees $T_2, T_3, \ldots, T_k$, and at the same time a series of alignments $A_1, A_2, \ldots, A_k$.

The starting point is the (trivial) tree T_2 with the two leaves representing s_1 and s_2, and the alignment A_2 that is just the optimal pairwise alignment of s_1 and s_2.

Then, iteratively the other sequences $s_3, \ldots, s_k$ are inserted in the tree as new leaves and joined into the growing alignment. In order to decide where a new leaf s_{i+1} is inserted in T_i, the cost of the new tree is computed for each possible insertion point, i.e., each internal edge of T_i, and compared to the cost of the corresponding alignment A_i.

In this procedure, the insertion of the new leaf s_{i+1} in an edge e (with the set of leaves L on the one side and the set of leaves R on the other side) of the tree is performed by breaking the edge e into two parts e_L and e_R and adding a new edge e_{new} leading to the new leaf (see Figure 12.3).

In order to compare tree cost and alignment cost, average distances between subtrees and subalignments are used. Thereby, the average distance between elements of two subtrees of T with leaves in L and R, respectively, is defined as follows:

$$\bar{D}_T(L,R) = \frac{1}{|L|}\frac{1}{|R|}\sum_{s\in L, t\in R} D_T(s,t)$$

where $D_T(s,t)$ is the distance (sum of edge lengths as defined below) between the two leaves s and t in the tree T.

The new alignment A_{i+1} is the result of an optimal three-way alignment of the projection of the previous alignment A_i to the leaves in L, the projection of the previous alignment A_i to the leaves in R, and the new sequence s_{i+1}. The average distance of the two sub-alignments is

$$\bar{D}_A(L,R) = \frac{1}{|L|}\frac{1}{|R|}\sum_{s\in L, t\in R} D_A(s,t)$$

where $D_A(s,t)$ is the cost of the projection of A on sequences s and t, that is, $D_A(s,t) = D_2(\pi_{\{s,t\}}(A))$.

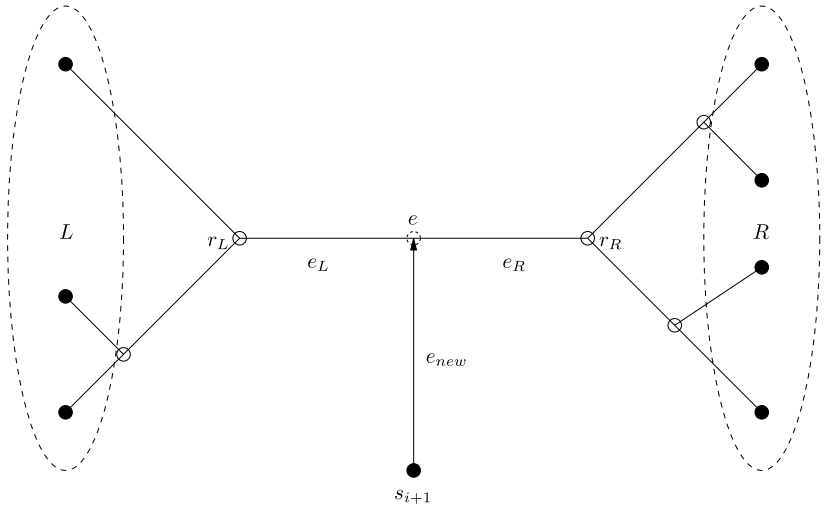


Figure 12.3: Inserting a new leaf s_{i+1} into edge e of tree T. Filled circles represent the given sequences, empty circles represent Steiner points (inferred sequences at internal points in the tree), and the dashed circle represents a possible insertion point for the edge leading to the newly inserted sequence s_{i+1}.

In order to avoid trivial results, during the optimization the average distances are required to be equal in the tree and in the alignment:

$$\begin{aligned} \bar{D}_{T_{i+1}}(\{s_{i+1}\}, L) &= \bar{D}_A(\{s_{i+1}\}, L) \\ \bar{D}_{T_{i+1}}(\{s_{i+1}\}, R) &= \bar{D}_A(\{s_{i+1}\}, R) \\ \bar{D}_{T_{i+1}}(L, R) &= \bar{D}_A(L, R). \end{aligned} \tag{12.1}$$

Now, we would like to derive the new edge lengths e_L, e_R, and e_{new}. Therefore, observe the following from Figure 12.3:

$$\begin{aligned} \bar{D}_{T_{i+1}}(\{s_{i+1}\}, L) &= e_{new} + e_L + \bar{D}_{T_i}(L, r_L) \\ \bar{D}_{T_{i+1}}(\{s_{i+1}\}, R) &= e_{new} + e_R + \bar{D}_{T_i}(R, r_R) \\ \bar{D}_{T_{i+1}}(L, R) &= e_L + e_R + \bar{D}_{T_i}(L, r_L) + \bar{D}_{T_i}(R, r_R). \end{aligned}$$

Using the equalities (12.1) this can be solved for the desired edge lengths:

$$\begin{aligned} e_L &= \frac{1}{2}\left(\bar{D}_A(s_{i+1}, L) + \bar{D}_A(L, R) - \bar{D}_A(s_{i+1}, R)\right) - \bar{D}_{T_i}(L, r_L) \\ e_R &= \frac{1}{2}\left(\bar{D}_A(L, R) + \bar{D}_A(s_{i+1}, R) - \bar{D}_A(s_{i+1}, L)\right) - \bar{D}_{T_i}(R, r_R) \\ e_{new} &= \frac{1}{2}\left(\bar{D}_A(s_{i+1}, R) + \bar{D}_A(s_{i+1}, L) - \bar{D}_A(L, R)\right). \end{aligned}$$

Note that by this equation, edge lengths can be negative. Therefore, among all insertion edges that result in no negative edge lengths, that one is selected for which the length of the new edge e_{new} in the tree is minimal. This defines the edge where the new leaf is inserted.

The insertion step is repeated until all leaves are inserted into the tree and thus T_k and A_k are obtained.

Finally, a refinement phase can be started, where one leaf at a time is removed from the tree and a new (possibly the same) insertion point is searched by the same procedure as above.

While there is no guarantee that the algorithm computes trees that come close to the optimal Steiner tree, in practice the method is not too bad, and quick as well, especially if a fast three-way alignment procedure is used.

12.2.2 The Deferred Path Heuristic

This heuristic method was developed by Schwikowski and Vingron (1997).

Similar to the greedy three-way alignment heuristic, the tree is iteratively constructed by inserting one sequence after the other. However, the tree is not directly constructed as a Steiner tree in sequence space, but instead first a directed acyclic *sequence graph* is computed representing all co-optimal and slightly sub-optimal pairwise alignments of the sequences. The algorithm is a kind of agglomerative clustering, where initially all singleton sequence graphs are constructed (as trivial nodes in the sequence space), and then in each iteration the two closest sequence graphs are merged into joint sequence graphs, where optimal and suboptimal paths between the two sequence graphs are considered (see Figure 12.4). In the end, the multiple alignment will be constructed from the final sequence graph for all k sequences.

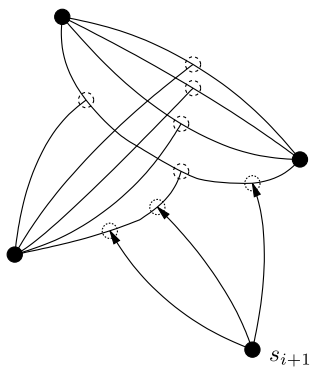


Figure 12.4: The deferred path heuristic for generalized tree alignment. Filled circles represent the given sequences, dashed circles represent possible Steiner points (inferred sequences at internal points in the tree). The dotted points are introduced during the insertion of sequence s_{i+1}.

This method yields an approximation algorithm with a guaranteed error bound of at most $\left(2 - \frac{2}{k}\right)$, with respect to the optimal generalized tree alignment.

CHAPTER 13

Whole Genome Alignment

Contents of this chapter: Filter algorithms, Maximal Unique Match (MUM), Maximal Exact Match (MEM), pairwise genome alignment (MUMer), Chaining Problem, multiple genome alignment (MGA, MUMer 3), multiple genome alignment with rearrangements (MAUVE).

Because many prokaryotic and eukaryotic genomes have been sequenced recently, and their comparison reveals much information about their function and evolutionary history, the alignment of whole genomes is currently very popular. A review of algorithms for whole genome alignment can be found in (Chain et al., 2003).

We start this chapter with a few general remarks on whole genome alignment:

1. The alignment of whole genomes only makes sense if there is a global similarity between the compared genomes. If the genomes do not have a common layout, other methods like 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 are filtration based, in the sense that in a first step highly similar (or identical) subregions are identified that in a second step are connected in the best possible way. This procedure may be repeated a few times.

In this chapter we will first explain the general approach of filtration with exact seeds in sequence analysis and how this can be efficiently performed using suffix trees. Then we describe a few of the more popular tools for whole genome alignment, MUMmer (Delcher et al., 1999, 2002; Kurtz et al., 2004), MGA (Höhl et al., 2002), and MAUVE (Darling et al., 2004).

13.1 Filter Algorithms

Filter algorithms (see also Section 3.9) are not only popular in whole genome alignment, but are also used in other methods for fast sequence comparison, the most popular examples being BLAST and FASTA, but they are also used in repeat finding, for instance.

Generally, a filter algorithm is a two-step procedure where first in a **filtration phase** small high-similarity regions, so-called seeds, are searched, and then in the second **verification phase** those regions that contain one or more seeds are postprocessed in various ways. As the particular way of postprocessing depends very much on the application that one is dealing with, in this section we will not go into further details. The kind of verification used in whole genome alignment (chaining) will be described in the following sections.

In the filtration phase, different types of seeds are considered by the various methods. Possibilities are

- exact or approximate matches,
- maximal (non-extendable) matches, and
- unique matches.

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. MUMs for two sequences were already discussed in Section 7.6.4, here we discuss the more general case. The precise definitions follow:

Definition 13.1 A MUM (Maximal Unique Match) is a substring 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.

The definition of MEMs is less restrictive as they may occur several times in the same sequence:

Definition 13.2 A MEM (Maximal Exact Match) is a substring that occurs in all sequences $s_1, s_2, \ldots, s_k$ and that cannot simultaneously be extended to the left or to the right in every sequence. A MEM in several sequences is sometimes called a *multiMEM*.

Both MUMs and MEMs can be efficiently found using the generalized suffix tree of the sequences $s_1, s_2, \ldots, s_k$.

We begin with multiMEMs: They can be found by a bottom-up traversal of the generalized suffix tree T of $s_1, s_2, \ldots, s_k$ (see Section 7.3). 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. In addition, to also test for left-maximality, one has to test that there are no extensions possible to the left. 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 pairs. 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. MUMs can be found in $O(kn)$ time.

Once the filtration has been performed, a verification step is necessary. In genome alignment this is done by *chaining*.

The task of finding a longest chain of compatible MUMs in step 2 (the **Chaining Problem**) can be modeled as the following graph problem: Let $R = \{r_1, \ldots, r_z\}$ be the MUMs found in the first phase of the algorithm. Define a partial order $\prec$ on MUMs where $r_i \prec r_j$ if and only if the end of MUM r_i is smaller than the beginning of MUM 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 MUM represented by v, and $w(\text{start}) = w(\text{stop}) = 0$.

Problem 13.3 (Chaining Problem) Find a chain $c = (\text{start} = r_{i_0}, r_{i_1}, r_{i_2}, \ldots, r_{i_\ell}, r_{i_{\ell+1}} = \text{stop})$, 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 $w(c) := \sum_{j=1}^{\ell} w(r_{i_j})$.

It is well known 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 $O(z^2)$ time algorithm for the chaining. However, since the MUMs can be linearly ordered, using a heaviest increasing subsequence algorithm the computation can even be reduced to $O(z \log z)$ time.

13.2 Pairwise Genome Alignment (MUMmer 1 & 2)

The first and most popular alignment program for two whole genomes was MUMmer whose first (Delcher et al., 1999) and second (Delcher et al., 2002) versions differ only slightly.

The overall algorithm is the following:

1. Given two genomes s_1 and s_2, all MUMs are found using the generalized suffix tree as described in the previous section.
2. The longest chain of compatible MUMs is selected, where a set of MUMs is *compatible* if the MUMs can be ordered linearly (See also Figure H.3 on page 190)
3. Short gaps (up to 5000 base pairs) are filled by ordinary pairwise alignment. Long gaps remain unaligned.

Overall, the first two phases take time $O(n_1 + n_2 + z^2)$ or $O(n_1 + n_2 + z \log z)$, respectively. 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.

13.3 Multiple Genome Alignment (MGA and MUMmer 3)

With only little changes, the above described whole genome alignment procedure can also be applied to more than two genomes.

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 sequences are compared, it is unlikely that there exist many MUMs that are present *and unique* in all considered sequences. That is why the authors of the first multiple genome alignment program MGA (Höhl et al., 2002) decided to search for multi-MEMs above a certain minimal length, instead of MUMs.

In addition to the use of multi-MEMs, MGA differs from MUMmer in the way how the gaps are treated that remain after the chaining phase (which is identical to the approach described for MUMmer, except that it is multi-dimensional). Since the computation of multiple alignments is often very time-consuming, gaps are filled only if they are rather short, and then by the progressive alignment method ClustalW. For all long gaps, the MGA procedure is applied recursively with less stringent parameters, i.e. a shorter minimal multi-MEM length. This way weak signals may be found in a second or third round.

The program MGA can be obtained from `http://bibiserv.techfak.uni-bielefeld.de/mga/`.

MUMmer in its third version (Kurtz et al., 2004) is now also based on multi-MEMs, as the authors have noted that these are less restrictive and the chaining does not become more complicated.

13.4 Multiple Genome Alignment with Rearrangements (MAUVE)

The last 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 and MGA, 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". Like MGA, 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 collinear groups called 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 at `http://gel.ahabs.wisc.edu/mauve`. There is also an advanced version of MAUVE that includes rearrangements between the aligned genomes.

APPENDIX A

Distances versus Similarity Measures on Sequences

Contents of this chapter: Biologically inspired distances between DNA and protein sequences. Dissimilarity and similarity measures. Equivalence of dissimilarity and similarity formulation. Log-odds scores. Non-symmetric score matrices.

So far we have only defined distances between sequences, focusing on their differences. Another viewpoint is to focus on their common features. This leads to the notion of similarity measures (or scores). The purpose of this chapter is to introduce biologically more meaningful measures than edit distance, and to make the transition from the cost viewpoint to the score viewpoint.

A.1 Biologically Inspired Distances

The unit cost edit distance simply counts certain differences between strings. There is no inherent biological meaning. If we want to compare DNA or protein sequences, for example, a biologically more meaningful distance should be used.

The following is called the **transversion/transition cost** on the DNA alphabet:

	A	G	C	T
A	0	1	2	2
G	1	0	2	2
C	2	2	0	1
T	2	2	1	0

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

Table A.1: Amino acid replacement costs as suggested by W. Taylor.

Bases A and G are called **purines**, and bases C and T are called **pyrimidines**. The transversion/transition cost function reflects the biological fact that a purine/purine and a pyrimidine/pyrimidine replacement is much more likely to occur than a purine/pyrimidine replacement. Often, an insertion/deletion cost of 3 is used with the above costs to take into account that a deletion or an insertion of a base is even more seldom. These costs define the **transition/transversion distance** between two DNA sequences.

To compare protein sequences and define a cost function on the amino acid alphabet, we may count how many DNA bases must change minimally in a codon to transform one amino acid into another.

A more sophisticated cost function for replacement of amino acids was calculated by W. Taylor according to the method described in Taylor and Jones (1993) and is shown in Table A.1. To completely define a metric, we would also have to define the costs for insertions and deletions.

To allow such general costs, we have to re-define the edit alphabet. The reason is that so far, we have considered an operation $\mathtt{S}_c$ that substitutes the next character by $c \in \Sigma$, but does not tell us which character is being substituted. Since the cost of $\mathtt{S}_c$ can vary depending on the substituted character, from now on, we use the following **edit alphabet**:

$$\mathcal{E} :\equiv \mathcal{E}(\Sigma) := \{\mathtt{S}_{a,c}, \mathtt{I}_c, \mathtt{D}_c : a \in \Sigma, c \in \Sigma\}.$$

The previous copy operation $\mathtt{C}$ that copied the next arbitrary character $a \in \Sigma$ is now the appropriate $\mathtt{S}_{a,a}$. (We do not consider flips at all for the time being.) Similarly, we have to modify the definition of the **edit function** $E : \Sigma^* \times \mathcal{E}^* \to \Sigma^*$ (cf. p. 17) accordingly.

We will not allow arbitrary cost functions on $\mathcal{E}$, but only those that satisfy certain (intuitive) properties.

Definition A.1 A **sensible cost function** or **sensible dissimilarity function** on $\mathcal{E}(\Sigma)$ is

a function $cost : \mathcal{E} \to \mathbb{R}_0^+$ such that

$$\begin{aligned}
0 \leq cost(\mathtt{S}_{a,a}) &\leq cost(\mathtt{D}_a) = cost(\mathtt{I}_a) && \text{for all } a \in \Sigma \\
cost(\mathtt{S}_{a,a}) &\leq cost(\mathtt{S}_{a,c}) = cost(\mathtt{S}_{c,a}) && \text{for all } a, c \in \Sigma \\
cost(\mathtt{S}_{a,c}) &\leq cost(\mathtt{S}_{a,b}) + cost(\mathtt{S}_{b,c}) && \text{for all } a, b, c \in \Sigma \\
cost(\mathtt{S}_{a,c}) &\leq cost(\mathtt{D}_a) + cost(\mathtt{I}_c) && \text{for all } a, c \in \Sigma
\end{aligned}$$

The first and second condition state that a copy operation on $a \in \Sigma$ should always have lower cost than any other operation involving a; they also impose symmetry on the cost function. If we want the resulting edit distance to be a metric, we will explicitly demand $cost(\mathtt{S}_{a,a}) = 0$ for all a. The third condition states that dissimilarity should be subadditive and is essential in proving the triangle inequality of the edit distance defined by this cost function. The fourth condition is essential for substitutions being used at all; if it were violated, it would always be cheaper to delete one character and insert another.

A.2 From Distance to Similarity

The concept of a metric is useful because it embodies the essential properties of our intuition about distances. For example, if the triangle inequality is violated, we could find a shorter way from x to y via a detour over z, which is not intuitive. However, distances also have disadvantages, which have not become apparent so far, because we have looked at whole sequences.

What would happen if we adopt a **local** instead of a **global** view and start looking for the least different parts of two sequences (the sequences could be, as a whole, very different from each other)? By definition, 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. More generally, short strings simply cannot accumulate as many differences as long strings, and would be preferred by a distance measure.

As another example of a problematic issue, look at the amino acid dissimilarities above. The dissimilarity between glutamine (Q) and tryptophan (W) is twice as large as the dissimilarity between methionine (M) and arginine (R): $d(\mathrm{Q}, \mathrm{W}) = 32$ vs. $d(\mathrm{M}, \mathrm{R}) = 16$. Which units are they measured in? What is the intuition behind the above values? How could we derive meaningful distances or dissimilarities between amino acids? The answers to these questions are not easily found.

It turns out that some problems are more easily formulated in terms of similarity than in terms of dissimilarity or distance. For example, when we want to find similar substrings of two sequences, we can assign a positive similarity value (or **score**) to each pair that consists of a copy of the same letter and a negative score to each mismatched pair (corresponding to a substitution operation), and also to insertions and deletions. The main difference is that we can explicitly reward desired features instead of only punishing undesired features. For some people, a similarity-centered view is also psychologically more attractive than a distance-centered view.

Again, a similarity function on an edit alphabet should satisfy certain intuitive properties.

Definition A.2 A **sensible similarity function** or **sensible score function** on $\mathcal{E}(\Sigma)$ is a function $score : \mathcal{E} \to \mathbb{R}$ such that

$$\begin{aligned}
&score(\mathtt{S}_{a,a}) \geq 0 \geq score(\mathtt{D}_a) = score(\mathtt{I}_a) && \text{for all } a \in \Sigma \\
&score(\mathtt{S}_{a,a}) \geq score(\mathtt{S}_{a,c}) && \text{for all } a, c \in \Sigma \\
&score(\mathtt{S}_{a,c}) = score(\mathtt{S}_{c,a}) && \text{for all } a, c \in \Sigma \\
&score(\mathtt{S}_{a,c}) \geq score(\mathtt{D}_a) + score(\mathtt{I}_c) && \text{for all } a, c \in \Sigma
\end{aligned}$$

The first and second condition state that the similarity between a and itself is always higher than between a and anything else. In particular, insertions and deletions never score positively. The first and third condition impose symmetry, and the last condition states that a direct substitution is preferable to an insertion and a deletion.

In contrast to costs, similarity scores can be both negative and positive. In fact, the zero plays an important role: it represents a neutral event. This is why a copy operation should always have nonnegative score, and an insertion/deletion operation should always have nonpositive score.

We now review the definition of edit distance and define edit similarity. Cost and score of an edit sequence are defined as the sum of their components' costs and scores, respectively:

$$cost(e) := \sum_{i=1}^{|e|} cost(e_i), \qquad \text{and} \qquad score(e) := \sum_{i=1}^{|e|} score(e_i) \qquad (e \in \mathcal{E}^*).$$

Definition A.3 Given $x, y \in \Sigma^*$, their **edit distance** $d(x, y)$ is defined as

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

The edit sequences (there can be more than one) that achieve the minimum are called the distance-**minimizing edit sequences** and written as the set

$$\begin{aligned}
e_d^{\text{opt}}(x, y) &:= \operatorname{argmin}\{cost(e) : e \in \mathcal{E}^*,\ E(x, e) = y\} \\
&:= \{e \in \mathcal{E}^* : E(x, e) = y,\ cost(e) = d(x, y)\}.
\end{aligned}$$

The **edit score** $s(x, y)$ is defined as

$$s(x, y) := \max\{score(e) : e \in \mathcal{E}^*,\ E(x, e) = y\}.$$

There is also the set of score-**maximizing edit sequences**

$$\begin{aligned}
e_s^{\text{opt}}(x, y) &:= \operatorname{argmax}\{score(e) : e \in \mathcal{E}^*,\ E(x, e) = y\} \\
&:= \{e \in \mathcal{E}^* : E(x, e) = y,\ score(e) = s(x, y)\}.
\end{aligned}$$

A general note on terminology: If $\mathcal{X}$ is some space and we maximize a function $f : \mathcal{X} \to \mathbb{R}$ in such a way that the maximum is achieved by at least one $x \in \mathcal{X}$, then $\max_{x \in \mathcal{X}} f(x)$ denotes the value of the maximum and $\operatorname{argmax}_{x \in \mathcal{X}} f(x)$ denotes the *set* of $x \in \mathcal{X}$ that achieve the maximum.

An equivalence question. Given a cost function, an important question is whether we can define a score function in such a way that an edit sequence has minimal cost if and only if it has maximal score, i.e., such that $e_d^{\text{opt}}(x, y) = e_s^{\text{opt}}(x, y)$ for all $x, y \in \Sigma^*$. The same question arises when the score function is given, and we look for an equivalent cost function.

We first present an equivalent score function for the unit cost edit distance. Then we give more general results.

Theorem A.4 The standard edit distance costs (0 for a copy, 1 for a substitution, insertion, and deletion) are equivalent to the following edit similarity scores: 1 for a copy, 0 for a substitution, $-1/2$ for an insertion and deletion. With this choice, $s(x, y) = (|x| + |y|)/2 - d(x, y)$ for all $x, y \in \Sigma^*$.

Proof. The idea is to find a monotonicity preserving transformation from costs to scores. We introduce variables $s_{\mathtt{C}}$, $s_{\mathtt{S}}$, $s_{\mathtt{I}}$, and $s_{\mathtt{D}}$ for the scores to be determined. We shall attempt to define them in such a way that there is a constant γ (that may depend on x and y, but not on e) such that $\mathit{score}(e) = \gamma - \mathit{cost}(e)$.

Let e be any edit sequence with $n_{\mathtt{C}}$ copy-type substitutions, $n_{\mathtt{S}}$ non-copy substitutions, $n_{\mathtt{I}}$ insertions and $n_{\mathtt{D}}$ deletions, so

$$\mathit{score}(e) = \sum_{o \in \{\mathtt{C},\mathtt{S},\mathtt{I},\mathtt{D}\}} s_o \, n_o.$$

Any edit sequence e that transforms some sequence $x \in \Sigma^*$ into $y \in \Sigma^*$ satisfies

$$|x| + |y| = 2\, n_{\mathtt{C}} + 2\, n_{\mathtt{S}} + n_{\mathtt{I}} + n_{\mathtt{D}},$$

or equivalently,

$$n_{\mathtt{C}} = \frac{|x| + |y|}{2} - n_{\mathtt{S}} - n_{\mathtt{I}}/2 - n_{\mathtt{D}}/2.$$

Therefore

$$\begin{aligned}
\mathit{score}(e) &= s_{\mathtt{C}} \cdot \left(\frac{|x| + |y|}{2} - n_{\mathtt{S}} - n_{\mathtt{I}}/2 - n_{\mathtt{D}}/2 \right) + s_{\mathtt{S}} \cdot n_{\mathtt{S}} + s_{\mathtt{I}} \cdot n_{\mathtt{I}} + s_{\mathtt{D}} \cdot n_{\mathtt{D}} \\
&= s_{\mathtt{C}} \cdot \frac{|x| + |y|}{2} + (s_{\mathtt{S}} - s_{\mathtt{C}}) \cdot n_{\mathtt{S}} + (s_{\mathtt{I}} - s_{\mathtt{C}}/2) \cdot n_{\mathtt{I}} + (s_{\mathtt{D}} - s_{\mathtt{C}}/2) \cdot n_{\mathtt{D}}.
\end{aligned}$$

We want to write this in such a way that

$$\mathit{score}(e) = \gamma - \mathit{cost}(e)$$

for some constant γ independent of the n-values to preserve monotonicity. Since

$$-\mathit{cost}(e) = -1 \cdot n_{\mathtt{S}} - 1 \cdot n_{\mathtt{I}} - 1 \cdot n_{\mathtt{D}},$$

we need to set the coefficients above correctly, i.e.,

$$\begin{aligned}
s_{\mathtt{C}} \cdot \frac{|x| + |y|}{2} &= \gamma, \\
s_{\mathtt{S}} - s_{\mathtt{C}} &= -1, \\
s_{\mathtt{I}} - s_{\mathtt{C}}/2 &= -1, \\
s_{\mathtt{D}} - s_{\mathtt{C}}/2 &= -1.
\end{aligned}$$

It follows that we can arbitrarily set $s_{\mathtt{C}} = 1$ to obtain $s_{\mathtt{S}} = 0$ and $s_{\mathtt{I}} = s_{\mathtt{D}} = -1/2$ and $\gamma = (|x| + |y|)/2$; the theorem is proved. □

Note that we could also have chosen $s_{\mathtt{C}} = 2$ to obtain $\gamma = |x| + |y|$, $s_{\mathtt{S}} = 1$, $s_{\mathtt{I}} = s_{\mathtt{D}} = 0$. This is just at the limit of being a sensible score function. Similarly, for $s_{\mathtt{C}} = 0$, we would get $\gamma = 0$ and $s_{\mathtt{S}} = s_{\mathtt{I}} = s_{\mathtt{D}} = -1$. Is there a best choice? This question becomes important when we discuss local similarity in Section 4.5.

A more general version of the above theorem is the following one.

Theorem A.5 Let *cost* be a sensible cost function. Let $M := \max_{a\in\Sigma} cost(\mathtt{S}_{a,a})$. For each edit operation $o \in \mathcal{E}$, define

$$score(o) := \begin{cases} M - cost(o) & \text{if } o \in \{\mathtt{S}_{a,c} : a, c \in \Sigma\}, \\ M/2 - cost(o) & \text{if } o \in \{\mathtt{I}_a, \mathtt{D}_a : a \in \Sigma\}. \end{cases}$$

Then

1. *score* is a sensible similarity function;
2. for the constant $\gamma := M \cdot (|x| + |y|)/2$, we have $s(x, y) = \gamma - d(x, y)$;
3. $e_s^{\text{opt}}(x, y) = e_d^{\text{opt}}(x, y)$.

Proof. 1. Score symmetry follows from cost symmetry. By definition of M as maximum copy cost, $score(\mathtt{S}_{a,a}) \geq 0$ for all $a \in \Sigma$. Also, $score(\mathtt{D}_a) = 1/2 \cdot (M - 2 \cdot cost(\mathtt{D}_a)) \leq 0$, since for sensible costs, $M \leq [cost(\mathtt{D}_a) + cost(\mathtt{I}_a)]$ for all $a \in \Sigma$. We have $score(\mathtt{S}_{a,a}) \geq score(\mathtt{S}_{a,c})$ for all $a \neq c$, since the reverse inequality holds for the costs by definition. Finally, we use $cost(\mathtt{S}_{a,c}) \leq cost(\mathtt{D}_a) + cost(\mathtt{I}_c)$ to see that $score(\mathtt{S}_{a,c}) = M - cost(\mathtt{S}_{a,c}) \geq M/2 - cost(\mathtt{D}_a) + M/2 - cost(\mathtt{I}_c) = score(\mathtt{D}_a) + score(\mathtt{I}_c)$.

2. The proof idea is the same as in the special case. Let e be any edit sequence transforming x into y, let $n_{a,b}$ be the number of $\mathtt{S}_{a,b}$-operations for $a, b \in \Sigma$, and let d_a and i_a be the numbers of $\mathtt{D}_a$ and $\mathtt{I}_a$ operations, respectively. Then $|x| + |y| = 2 \cdot \sum_{a,b} n_{a,b} + \sum_a [i_a + d_a]$. Thus

$$\begin{aligned} score(e) &= \sum_{a,b} n_{a,b} \cdot score(\mathtt{S}_{a,b}) + \sum_a [d_a \cdot score(\mathtt{D}_a) + i_a \cdot score(\mathtt{I}_a)] \\ &= \sum_{a,b} n_{a,b} \cdot (M - cost(\mathtt{S}_{a,b})) + \sum_a [d_a \cdot (M/2 - cost(\mathtt{D}_a)) + i_a \cdot (M/2 - cost(\mathtt{I}_a))] \\ &= M \cdot \Big(\sum_{a,b} n_{a,b} + \sum_a [d_a/2 + i_a/2]\Big) \\ &\qquad - \sum_{a,b} n_{a,b} \cdot cost(\mathtt{S}_{a,b}) - \sum_a [d_a \cdot cost(\mathtt{D}_a) + i_a \cdot cost(\mathtt{I}_a)] \\ &= M(|x| + |y|)/2 - cost(e). \end{aligned}$$

This shows that $score(e) = \gamma - cost(e)$ for all edit sequences that transform x into y. Let e^* be a cost-optimal edit sequence. By the above statement, it is also score-optimal. The converse argument holds for a score-optimal edit sequence. It follows that $s(x, y) = \max_{e:E(x,e)=y} score(e) = \gamma - \min_{e:E(x,e)=y} cost(e) = \gamma - d(x, y)$.

3. This statement has just been proven as part of 2. □

When we apply the above theorem to the unit cost edit distance, we obtain $M = 0$ and thus similarity scores $score(\mathtt{C}) \equiv score(\mathtt{S}_{a,a}) = 0$ for all $a \in \Sigma$, and negative unit scores for the other operations. Since the whole point of a similarity measure is to reward identities, this is probably not what we desire. In fact, we can often increase the constant M in the previous theorem and still obtain a sensible similarity function. We did this in Theorem A.4, where we arbitrarily chose $M := 1$.

Note that the conversion also works in the other direction when a similarity measure is given whose smallest identity score $\min_a score(\mathtt{S}_{a,a})$ is zero. In general, a conversion in the spirit of Theorem A.5 can always be attempted, but it may lead to a non-sensible cost function when starting with a similarity function.

A.3 Log-Odds Score Matrices

Now that we have established a similarity-based viewpoint, it is time to discuss how we can define similarity scores for biological sequences from an evolutionary point of view; we focus on proteins and define a similarity function on the alphabet of 20 amino acids.

The basic observation is that over evolutionary time-scales, in functional proteins, two similar amino acids are replaced more frequently by each other than dissimilar amino acids. The twist is now to *define* similarity by observing how often one amino acid is replaced by another one in a given amount of time.

Let $\pi = (\pi_a)_{a \in \Sigma}$ be the **frequency vector**, also **probability vector**, which means that $\pi_a \geq 0$ for all a and $\sum_a \pi_a = 1$, of the naturally occurring amino acids. If we look at two random positions in two randomly selected proteins, the probability that we see the ordered pair (a, b) is $\pi_a \cdot \pi_b$. The entries of π are also called **background frequencies** of the amino acids.

Now assume that we can track the fate of individual amino acids over a fixed evolutionary time period t. Let $M^t(a, b)$ be the observed frequency table of homologous amino acid pairs where we observed a at the beginning of the period and b at the end of the period, such that $\sum_{a,b} M^t(a, b) = 1$. Usually, we count substitutions and identities twice, i.e., once in each direction. This ensures the symmetry of $M^t : M^t(a, b) = M^t(b, a)$. The reason is that we can in fact not track amino acids during evolution; we can only observe the state of two present-day sequences and do not know their most recent common ancestor. The fields of **molecular evolution** and **phylogenetics** examine more of the resulting implications. For more examples of these fields see Chapters 10.4, 12 or H.

We can find the background frequencies as the marginals of M^t: $\pi_a = \sum_b M^t(a, b) = \sum_b M^t(b, a)$ because of symmetry.

There are two reasons why an entry $M^t(a, b)$ can be relatively large. The first reason is the one we are interested in: because a and b are similar. The second reason is that simply a or b could be frequent amino acids. To remove the second effect, we consider the so-called **likelihood ratio** $M^t(a, b)/(\pi_a \cdot \pi_b)$, which relates the probability of the pair (a, b) in an evolutionary model to its probability in a random model. We declare that a and b are similar if this ratio exceeds 1 and that they are dissimilar if this ratio is below 1. To obtain

an additive function, we take the logarithm and define the **log-odds score matrix** with respect to time period t by

$$S^t(a,b) := \log\left(\frac{M^t(a,b)}{\pi_a \cdot \pi_b}\right).$$

The time parameter t should be chosen in such a way that the score matrix is optimal for the problem at hand: If we want to compare two sequences whose most recent common ancestor existed, say, 530 million years ago, then we should ideally use a score matrix constructed from sequence pairs with distance $t \approx 1060$ million years.

Some remarks:

- For convenience, scores are often scaled by a constant factor and then rounded to integers. The score unit is called a **bit** if the score is obtained by a $\log_2(\cdot)$ operation. When multiplied by a factor of three, say, we get scores in **third-bits**.
- Since we cannot easily observe how DNA or proteins change over millions of years, constructing a score matrix is somewhat difficult. Since the pioneering work of Margaret Dayhoff and colleagues in the 1970s, Markov processes are used to model molecular evolution. Methods that allow to integrate information from sequences of different divergence times in a consistent fashion have been developed more recently.
- Important score matrix families for proteins are the **PAM(t)** and the **BLOSUM(s)** matrices. Here t is a divergence time parameter and s is a clustering similarity parameter inversely correlated to t. (The inventors of BLOSUM have chosen to index their matrices in a different way.) The BLOSUM matrices are the most widely used ones for protein sequence comparison.

Non-symmetric score matrices. We usually demand that similarity functions and hence score matrices are symmetric. In some cases, however, there are good reasons to choose a non-symmetric similarity function. Often then, the unsymmetry does not stem from an unsymmetry of M^t (for reasons explained above), but from different background frequencies. We mention an example.

Assume that we have a particular protein fragment (the "query") that forms a transmembrane helix. The amino acid composition in membrane regions differs strongly from that of the "average" protein: It is hydrophobically biased. Assume that we want to look for similar fragments in a large database of peptide sequences (of unknown or "average" type). It follows that the matrix M^t of pair frequencies should be one derived from an evolutionary process acting on transmembrane helices and that two different types of background frequencies should be used: The query background frequencies τ are those of the transmembrane model, different from the background frequencies π of the database. It follows that we should use

$$S^t(a,b) := \log\left(\frac{M^t(a,b)}{\tau_a \cdot \pi_b}\right),$$

or a rounded multiple thereof, so that now $S(a,b) \neq S(b,a)$ in general. Table A.2 shows the SLIM161 matrix for transmembrane helices (Müller et al., 2001). The 161 is the time parameter t referring to the expected number of evolutionary events per 100 positions.

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	**5**	-8	-5	-10	3	-7	-11	0	-5	1	1	-10	1	1	-5	2	1	-2	-3	2
R	-3	**10**	-4	-10	-2	-3	-11	-4	-4	-2	-1	-3	-1	-2	-7	-4	-3	-2	-3	-3
N	-1	-5	**8**	-2	0	-2	-7	-2	2	-1	-1	-6	0	1	-5	2	0	-2	2	-2
D	-3	-9	1	**9**	-4	-2	1	-2	-2	-2	-2	-6	-2	-1	-5	-3	-3	-3	-2	-2
C	2	-8	-5	-11	**11**	-7	-12	-3	-8	0	1	-12	1	3	-11	2	0	-1	0	1
Q	-1	-2	0	-3	0	**7**	-4	-2	0	0	0	-4	2	2	-4	0	-1	4	1	0
E	-3	-7	-2	3	-2	-1	**7**	-3	-1	-2	-2	-8	-1	0	-4	-1	-3	-1	-1	-2
G	1	-7	-4	-7	0	-5	-10	**7**	-7	-1	0	-8	0	1	-5	1	-1	-3	-2	-1
H	-1	-5	3	-5	-2	-1	-5	-4	**10**	-1	-1	-8	-1	2	-7	-1	-2	1	5	-2
I	-1	-9	-7	-11	-1	-7	-12	-4	-7	**6**	3	-11	4	2	-6	-3	-1	-2	-3	4
L	-1	-8	-6	-10	1	-7	-12	-4	-7	4	**5**	-11	4	3	-7	-3	-1	-1	-2	2
K	-1	2	-1	-4	-2	0	-7	-1	-3	0	-1	**6**	0	0	-1	-1	-1	-1	1	-2
M	-1	-7	-6	-10	1	-5	-11	-3	-7	4	4	-11	**7**	3	-7	-2	0	-1	-2	2
F	-1	-9	-5	-10	2	-6	-11	-3	-4	1	2	-11	2	**8**	-7	-2	-2	3	4	0
P	-3	-9	-7	-9	-7	-8	-10	-4	-9	-2	-3	-8	-3	-2	**11**	-3	-3	-3	-4	-2
S	2	-8	-1	-8	4	-5	-9	0	-4	0	0	-9	0	1	-5	**6**	1	-2	-1	0
T	2	-7	-3	-8	2	-6	-10	-2	-5	2	2	-9	3	2	-4	2	**4**	-4	-2	2
W	-3	-7	-6	-10	0	-2	-11	-5	-3	-1	0	-10	0	4	-6	-4	-5	**15**	2	-2
Y	-4	-8	-2	-9	1	-5	-10	-5	0	-2	-1	-8	-1	6	-7	-3	-3	2	**11**	-2
V	1	-9	-7	-10	1	-7	-11	-4	-7	5	3	-12	3	2	-6	-2	0	-2	-3	**5**

Table A.2: The non-symmetric SLIM 161 score matrix. Scores are given in third-bits, i.e., as $S(a,b) = \text{round}[3 \cdot \log_2(M(a,b)/(\tau_a \pi_b))]$.

APPENDIX B

Pairwise Sequence Alignment (Extended Material)

Contents of this chapter: Number of Alignments.

B.1 The Number of Global Alignments

How many ways are there to align two sequences of lengths m and n? Let $N(m, n)$ be that number. It is equal to the number of different edit sequences that transform a length-m sequence into a length-n sequence.

An algorithmic perspective. First, we derive a recurrence for $N(m, n)$. Obviously, there is just one possibility to align ε to any sequence; therefore $N(m, 0) = N(0, n) = 1$ for all $m \geq 0$ and $n \geq 0$.

Theorem B.1 For $m \geq 1$ and $n \geq 1$,

$$N(m, n) = N(m-1, n-1) + N(m-1, n) + N(m, n-1).$$

Proof. Each alignment must end with either a match/mismatch, an insertion, or a deletion. In the match/mismatch case, there are $N(m-1, n-1)$ ways to align the corresponding prefixes of length $m-1$ and $n-1$. The other cases are treated correspondingly. All these possibilities lead to different alignments; see also Figure B.1. This argument shows that $N(m, n)$ is at least as large as the stated sum. However, we have also covered all possibilities; therefore equality holds. □

Note that the proof of Theorem B.1 follows the same argumentation as the one for Theorem 3.5. This is because we are exploiting the same structural property of alignments (or edit sequences). A technique called **algebraic dynamic programming** developed in Robert

1	1	1	1	1	...
1	3	5	7	9	...
1	5	13	25	41	...
1	7	25	63	$\ddots$	$\ddots$
1	9	41	$\ddots$	$\ddots$	$\ddots$
1	$\vdots$	$\vdots$	$\ddots$	$\ddots$	$\ddots$

Figure B.1: The number $N(m,n)$ of global alignments of sequences of lengths m and n is equal to $N(m-1,n-1)+N(m,n-1)+N(m-1,n)$: This recurrence can be easily computed by dynamic programming, just like the edit distance.

Giegerich's group "Praktische Informatik" at Bielefeld University uses methods from algebra to make such connections more explicit.

Theorem B.1 provides us with an algorithm to compute $N(m,n)$ in $O(mn)$ arithmetic operations. As we will see below, the numbers $N(m,n)$ grow exponentially and thus have $O(\min\{m,n\})$ bits such that the total time is $O(mn\min\{m,n\})$, which by the way is exponential in the input size (the input just consists of $O(\log m+\log n)$ bits to encode the numbers m and n). When implementing a function to compute $N(m,n)$, it is advisable to use a library that can compute with integers of arbitrary size (e.g. the `BigInteger` class of the Java Runtime Environment).

A combinatorial perspective. Fortunately, it is also not difficult to find a closed (i.e., non-recursive) formula for $N(m,n)$. Because N is symmetric (which follows from the definition and formally from the symmetries of the initialization and of the recurrence), we will assume that $m \geq n \geq 0$.

Recall from Observation 4.3 that the number of match/mismatch columns k in any alignment satisfies $0 \leq k \leq n$. If there are k such columns, the alignment length is $m+n-k$. The number of indel columns is thus $m+n-2k$; there are then $m-k$ insertions in the longer sequence and $n-k$ insertions in the shorter sequence.

There are thus $\binom{m+n-k}{k}$ possibilities to choose the match/mismatch positions in the alignment, and, for each of these, further $\binom{m+n-2k}{n-k}$ possibilities to distribute the insertion positions of the shorter sequence (deletion positions of the longer sequence) among the indel positions.

For fixed k, we thus get $\binom{m+n-k}{k}\cdot\binom{m+n-2k}{n-k}=\binom{m+r}{n-r}\cdot\binom{m-n+2r}{r}$ possibilities, where we have set $r:=n-k$ (the number of indel positions in the shorter sequence). It follows that

$$N(m,n)=\sum_{r=0}^{n}\binom{m+r}{n-r}\cdot\binom{m-n+2r}{r}.$$

For $m=n$, we get the diagonal entries

$$N(n,n)=\sum_{r=0}^{n}\binom{n+r}{n-r}\cdot\binom{2r}{r}.$$

Laquer (1981) shows that

$$N(n,n) \approx \frac{1}{2^{5/4}\sqrt{\pi}} \cdot \frac{(3+2\sqrt{2})^{n+1/2}}{\sqrt{n}}.$$

For more information about this diagonal sequence, you may consult the extremely useful **on-line encyclopedia of integer sequences** maintained by Neil J.A. Sloane, sequence A001850, at `http://www.research.att.com/~njas/sequences/A001850`.

By looking up research papers on $N(m,n)$, particularly Gupta (1974), we can find out additional properties of $N(m,n)$, e.g.

$$N(m,n) = \sum_{j=0}^{n} \binom{n}{j} \cdot \binom{m+j}{n} = \sum_{j=0}^{n} \binom{m+j}{j} \cdot \binom{m}{n-j} = \sum_{j=0}^{n} 2^j \cdot \binom{n}{j} \cdot \binom{m}{j}.$$

Relating $N(m,n)$ to the theory of hypergeometric functions (in the same paper), we can compute a single row (or column) of the matrix. The important observation is that m is fixed in the next recurrence.

Theorem B.2 Recall that $N(m,0) = 1$ and $N(m,1) = 1 + 2m$ for $m \geq 1$. For $n \geq 2$,

$$N(m,n) = [(2m+1) \cdot N(m,n-1) + (n-1) \cdot N(m,n-2)]/n.$$

Proof. This goes far beyond the scope of these lecture notes. □

Disregarding the order of consecutive indels. One could argue that the above numbers are exaggerated because the order of insertions and deletions immediately following each other is not important. For example, the three alignments

A	X	–	–	C	A	–	X	–	C	A	–	–	X	C
B	–	Y	Z	D	B	Y	–	Z	D	B	Y	Z	–	D

should be considered equivalent. That is, only the choice of the aligned character pairs should matter. Let $N'(m,n)$ be the number of effective alignments, counted in this fashion; again we assume that $m \geq n$.

If $k \in [0,n]$ denotes the number of match/mismatch positions in each sequence, we can select k such positions in $\binom{n}{k}$ ways in the shorter sequence and in $\binom{m}{k}$ ways in the longer sequence. The total number of possibilities is thus

$$N'(m,n) = \sum_{k=0}^{n} \binom{m}{k} \cdot \binom{n}{k} = \binom{m+n}{n},$$

where the last identity is known as the **Vandermonde convolution** of Binomial coefficients.

For $n = m$ we get

$$N'(n,n) = \binom{2n}{n} = \frac{(2n)!}{(n!)^2} \simeq \frac{4^n}{\sqrt{\pi \cdot n}} \cdot \exp(-3/(24n));$$

this is sequence A000984 in the on-line encyclopedia of integer sequences. The asymptotic value follows from **Stirling's approximation** for factorials:

$$\ln(n!) = n \ln n - n + (\ln n)/2 + \ln(\sqrt{2\pi}) + 1/(12n) + o(1/n).$$

Finally, some numbers.

n	0	1	2	3	4	5	...	1000	Reference
$N(n,n)$	1	3	13	63	321	1683	...	$\approx 10^{767.4}$	A001850
$N'(n,n)$	1	2	6	20	70	252	...	$\approx 10^{600}$	A000984

As an exercise, draw all 13 alignments resp. all 6 distinct classes of alignments of $x = \text{AB}$ and $y = \text{CD}$.

APPENDIX C

Pairwise Alignment in Practice (Extended Material)

Contents of this chapter: Fast Smith-Waterman, FASTA: on-line database search method, hot spots, diagonal runs, index-based database search methods (BLAT, SWIFT, QUASAR), software propositions.

C.1 Fast Implementations of the Smith-Waterman Algorithm

Since alignment by dynamic programming in a large-scale context can be very time demanding, some approaches have been taken to accelerate the alignment comparison. One approach is the parallelization using special hardware or special commands of common hardware.

Remarkable is here an effort where by using special graphics commands from the Intel MMX architecture, it was possible to speed-up the computation of Smith-Waterman local alignments by a factor of six on a Pentium computer. Other projects attempt to exploit the power of modern graphics cards.

Another approach is the use of special hardware, either FPGAs or special processors.

Commercial products are the GeneMatcher and BlastMachine from Paracel.

C.2 FASTA: An On-line Database Search Method

FASTA[1] used to be a popular tool for comparing biological sequences; it seems to be used less these days. First consider the problem FASTA was designed for: Let x be a *query sequence* (e.g. a novel DNA-sequence or an unknown protein). Let Y be a set of sequences (the database). The problem is to find all sequences $y \in Y$ that are similar to x. We need

[1] FASTA, pronounced "fast-ay", stands for fast-all: It is fast on nucleotide as well as protein sequences.

to specify the similarity notion used by FASTA. There is no formal model of what FASTA computes; instead FASTA is a heuristic stepwise procedure of three phases that is executed for each $y \in Y$. On a high level, the algorithm proceeds as follows.

- Preprocessing: Create a q-gram index I of the query x
- For each $y \in Y$ do:
  1. Find hot spots and compute the FASTA score $C(x, y)$.
  2. Combine hot spots into diagonal runs.
  3. Find maximal paths in the graph of diagonal runs.

We explain the first step in detail and steps 2 and 3 on a higher level.

Finding hot spots.

Definition C.1 Given $x \in \Sigma^m$, $y \in \Sigma^n$, and $q \in \mathbb{N}$, a **hot spot** is a location (i, j) of starting positions of a common q-gram of x and y.

Hot spots are grouped according to the diagonal in the edit matrix where they occur: The coordinates (i, j) are said to lie on diagonal $d := j - i$. Diagonal indices d thus range from $-m$ to n.

Definition C.2 For $d \in \{-m \dots n\}$, let $c(d)$ be the number of hot spots on diagonal d, i.e.,

$$c(x, y;\, d) := \#\Big\{ i : x\big[i \dots (i + q - 1)\big] = y\big[(i + d) \dots (i + d + q - 1)\big] \Big\}.$$

Then the **FASTA score** of x and y is defined as

$$C(x, y) := \max_d c(x, y;\, d).$$

Example C.3 Take $x =$ `freizeit`, $y =$ `zeitgeist`, and $q = 2$. Then $c(-4) = 3$, $c(-1) = 1$, $c(0) = 1$, $c(3) = 1$ and $c(d) = 0$ for $d \notin \{-4, -1, 0, 3\}$; see Figure C.1. ■

		z	e	i	t	g	e	i	s	t	
f											
r											
e			↘				↘				
i				↘				↘			
z		↘									
e			↘				↘				
i				↘				↘			
t					↘					↘	

Figure C.1: Diagonal matches for `freizeit` and `zeitgeist`

To compute $c(x, y; d)$ for all diagonals d, we move a q-window across the subject y, and note which q-grams occur in the query x and where; so we can increment the correct diagonal counters. In essence, the following pseudo-code is executed (see Algorithm C.1). Note the use of the q-gram index I of the query x. The code hides the details of the implementation of the index I.

Algorithm C.1 Pseudo-code FASTA

```
for d = −m to n do
    c[d] = 0
for j = 1 to n − q + 1 do
    if j == 1 then
        r =rank(y[1]...y[q])
    else
        r =update_rank(r, y[j + q − 1])
    for each i in I[r] do
        c[j − i]++
```

Now it is easy to compute $C(x, y) = max_d\, c(x, y; d)$. The whole procedure takes $O(m + n + \sum_d c(x, y; d))$ time: The more common q-grams exist between x and y, the longer it takes. If in the end we find that $C(x, y)$ is small, i.e., smaller than a pre-defined threshold t, we decide that y is not sufficiently similar to x to warrant further investigation: We skip the next steps and immediately proceed with the next subject from the database. Of course, a good choice of q and t is crucial; we investigate these issues further in Chapter D.

If $C(x, y) \geq t$, one option is to directly compute a full Smith-Waterman alignment of x and y, and maybe even suboptimal alignments. In this sense one can use FASTA as a **heuristic filter** before running a full alignment.

Another option is to continue heuristically: We re-process the diagonals and note where on each diagonal we have a matching q-gram. Then these matches (including small mismatching regions in between) can be combined into diagonal runs.

Diagonal runs: Creation and optimal connection. Diagonal runs are hot spots on the same diagonal, possibly with small mismatching regions in between, see Figure C.2 for an illustration. To score diagonal runs, one assigns positive scores to the hot spots and negative scores to the mismatching regions. Note that not necessarily all hot spots on the same diagonal are put into a single diagonal run: One diagonal can contain more than one run.

In the next step, a directed graph is constructed. The nodes of the graph are the diagonal runs from the previous step with corresponding positive scores assigned. Let us represent a diagonal run r by its upper left corner $(top(r), left(r))$ and its lower right corner $(bottom(r), right(r))$. The nodes for runs r and r' are connected by an edge $r \rightarrow r'$ if and only if $bottom(r) \leq top(r')$ and $right(r) \leq left(r')$. Such an edge represents gaps and/or mismatches between r and r', and therefore edges receive a negative score.

The graph of runs is obviously acyclic and therefore we can compute all maximal paths. A path is maximal if extending it by another vertex (at either end or inside) lowers its score. The score of a path is the (positive) score plus the (negative) score of its edges.

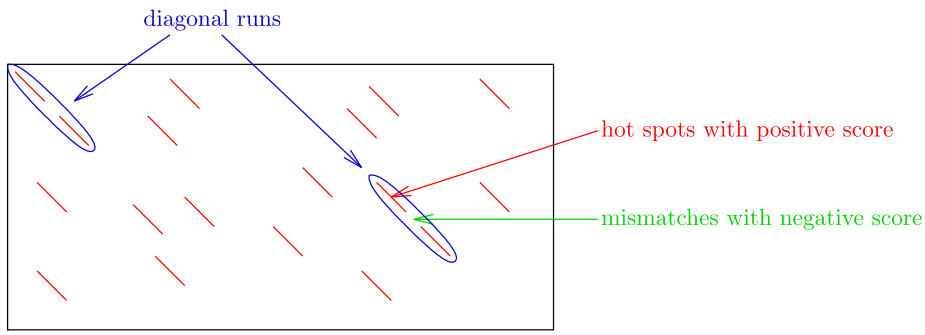


Figure C.2: Diagonal runs in the FASTA heuristic.

Suppose that all maximal paths are computed. From each path we pick the first and the last node. The upper left corner of the first node and the lower right corner of the last node define a pair of substrings of x and y. These are aligned using standard global alignment algorithms.

C.3 Index-based Database Search Methods

The previous methods do not pre-process the database in any way and therefore can react flexibly to continuing changes of the database. They do pre-process the query, but this is usually not referred to as indexing. As discussed above, database indexing can speed up searches tremendously. The general ideas are the same as for indexing the query: Often we simply generate position lists of q-grams of the database. If we do not want to specify q in advance, so-called suffix arrays provide an alternative. These are extremely useful data structures discussed in detail in Chapter 8. Here we give general ideas about three software tools that use slightly different indexing strategies.

Common strategies employed by these methods are:

- When building the index, partition the database into buckets to avoid keeping a hit counter for every database position or diagonal. Buckets usually overlap a little in order not to miss hits at bucket boundaries.
- Find high-scoring hits for each bucket. Discard a bucket when the number of hits is too low. This allows to discard a large fraction of the database very quickly.
- Run a more sensitive alignment algorithm on the remaining buckets.

BLAT. BLAT (Kent, 2002) was designed to be exceptionally fast for detecting highly similar sequences. It was developed for assembling the human genome, where hundreds of millions of DNA sequence reads have to be checked for possible overlap. Unlike BLAST, which allows to find somewhat more remote homologies, BLAT focuses on essentially identical sequence parts with very few errors.

During preprocessing, BLAT constructs an index of nonoverlapping (!) q-grams and their positions in the database. BLAT excludes q-grams from the index that occur too often.

For each query, BLAT looks up each overlapping q-gram of the query sequence in the index; it also has the option of looking up q-grams with one mismatch. In this way, BLAT builds a list of hits (each hit is a pair (i, j) of query and database position) where query and database share a q-gram. The hit list is split into buckets (of size 64K), based on the database position. The hits in each bucket are sorted according to their diagonal $i - j$. Hits within the gap limit are bundled together into proto-clumps. Hits within proto-clumps are then sorted along the database coordinate and joined into real clumps if they are within the window limit on the database coordinate. Clumps with less than the minimum number of hits are discarded; the remaining ones are used to define regions of the database which are homologous to the query sequence.

SWIFT. SWIFT (Rasmussen et al., 2006) was designed as an exact filter to find all ε-matches of a query against a database: Let n be the minimum length of an alignment and e the number of allowed errors. The error rate $\varepsilon := e/n$ is specified by the user: the smaller the allowed error rate, the faster the algorithm will run. After constructing a q-gram index for the database, the following filtration phase is executed for each query.

Let w be a window size whose value can be determined from n, q and e. SWIFT looks for parallelograms in the edit matrix of dimension $w \times e$ that may contain an ε-match. A threshold value t is determined such that only those parallelograms are of interest that contain t or more q-gram hits. In order to locate such parallelograms, a w-window is moved over the query, and q-gram hits are counted for each $e + 1$ adjacent diagonals. In practice, this step can be accelerated by using bins of size $e + 1 + \Delta$ for some $\Delta > 0$, e.g. $\Delta = 2^z$ for some $z \in \mathbb{N}$ with $2^z > e$. Each parallelogram whose hit counter reaches the threshold t is reported, possibly merging overlapping parallelograms.

The filtration phase guarantees that no ε-matches are lost. Now these matches can be extended to full local alignments. Therefore, dynamic programming is used to find an optimal local alignment starting in the parallelogram, possibly using BLAST's X-drop extension algorithm.

The time complexity of the method is $O(|\Sigma|^q + N)$ for the preprocessing (building the q-gram index) plus $O(mN|\Sigma|^{-q})$ expected time for filtration, which can be $O(m)$ if the user-defined choice of ε allows an appropriate choice of q. The time of the extension obviously depends on the similarity of the two sequences, and on the exact method that is used for the extension.

QUASAR. Each q-gram based approach required a careful choice of q (see also Chapter D). Depending on the application, a user may want to vary the value of q. However, with the q-gram index approach described here, this would require rebuilding the index. One way to leave the choice of q flexible is to use a data structure called a **suffix array**. The main difference between QUASAR[2] (Burkhardt et al., 1999) and other q-gram based approaches is precisely the use of suffix arrays.

[2] Q-gram Alignment based on Suffix ARrays

C.4 Software

This final section contains some pointers to implementations of the methods discussed in this chapter. This list is far from complete.

Dotter for creating dotplots with greyscale rendering

- E.L.L. Sonnhamer and R. Durbin
- `http://www.cgb.ki.se/cgb/groups/sonnhammer/Dotter.html`

DOTLET is a nice tool written in Java that generates dot plots for two given sequences.

- T. Junier and M. Pagni
- Web server: `http://myhits.isb-sib.ch/cgi-bin/dotlet`

DNADot nice interactive Web tool at `http://www.vivo.colostate.edu/molkit/dnadot/`

The FASTA package contains FASTA for protein/protein or DNA/DNA database search and SSEARCH for full Smith-Waterman alignment

- B. Pearson and D.J. Lipman
- `http://www.ebi.ac.uk/fasta/`
- `http://fasta.bioch.virginia.edu/`

BLAST Basic Local Alignment Search Tool

- NCBI BLAST website: `http://130.14.29.110/BLAST/`

BLAT is useful for mapping sequences to whole genomes

- J. Kent
- Web server: `http://genome.ucsc.edu`

SWIFT guarantees not to miss any ε-match

- K. Rasmussen, J. Stoye, and E.W. Myers
- `http://bibiserv.techfak.uni-bielefeld.de/swift/welcome.html`

STELLAR fast and exact pairwise local alignments using SWIFT

- B. Kehr, D. Weese, and K. Reinert
- `http://www.seqan.de/projects/stellar.html`

QUASAR q-gram based database searching using a suffix array

- Stefan Burkhardt and others
- provided as part of SeqAn at `http://www.seqan.de/`

EMBOSS Pairwise Alignment Algorithms. European Molecular Biology Open Source Software Suite `http://www.ebi.ac.uk/emboss/align/index.html`

- Global (*Needleman-Wunsch*) alignment: `needle`

- Local (*Smith-Waterman*) alignment: `water`

e2g is a web based-tool which efficiently maps large EST and cDNA data sets to genomic DNA.

- J. Krüger, A. Sczyrba, S. Kurtz, and R. Giegerich
- Webserver: `http://bibiserv.techfak.uni-bielefeld.de/e2g/`

APPENDIX D

Alignment Statistics

Contents of this chapter: Null model (for pairwise sequence alignment), statistics (q-gram matches, FASTA score), multiple testing problem, longest match, statistics of local alignments.

D.1 Preliminaries

In this chapter, 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 much 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 D.1 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 D.2 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 D.3 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 D.4 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 chapter provides an intuitive, but mathematically inexact approach.

D.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

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

[1] **i**ndependent and **i**dentically **d**istributed.

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 $\mathbb{P}(N(q) = 0) \approx \frac{\mathrm{e}^{-\mu(q)} \cdot [\mu(q)]^0}{0!} = \mathrm{e}^{-\mu(q)}$, see Section 2.2, 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

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

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. This probability is equal to the probability of at least one match of length at least ℓ, which is approximately $1 - \mathrm{e}^{-\mu(l)} = 1 - \mathrm{e}^{-(m-\ell+1)\cdot(n-\ell+1)\cdot p^\ell} \approx 1 - \mathrm{e}^{-mnp^\ell} \approx mnp^\ell$ 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.

The FASTA score. What is the probability to observe a FASTA score $C(X,Y) \geq t$ for a given q-gram length q and random sequences X and Y of lengths m and n, respectively? The most probable way to observe a FASTA score $\geq t$ is via the existence of t consecutively matching q-grams in the same diagonal for a total match length of $t + q - 1$. By the above paragraph, the probability is approximately

$$\mathbb{P}(C(X,Y) \geq t) \approx 1 - \mathrm{e}^{-mnp^{t+q-1}} \approx mnp^{t+q-1}.$$

This result can be used to guide the choice of the parameters q and score threshold t for the FASTA algorithm. For example, we could assume that average nucleotide queries have $m = n = 1000$ and $p = 1/4$. If we take $q = 6$, how must we choose t such that the above p-value is 0.01?

Of course, such a strict threshold may miss many weak similarities. The null model view is one-sided: We can make sure that we do not get too many hits by chance, but we do not know how many biologically interesting hits we might miss. To weigh these considerations against each other, we would have to specify a model for the biological signal, which is often close to impossible.

D.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 $C(X,Y)$ of the longest exact match, we get

$$\mathbb{P}(C(X,Y) \geq t) \approx 1 - \mathrm{e}^{-mnp^{t+q-1}} = 1 - \mathrm{e}^{-Cmne^{-\lambda t}}$$

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

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

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**.

APPENDIX E

Basic RNA Secondary Structure Prediction

Contents of this chapter: RNA structure, RNA structure elements, the Optimization Problem, Simple RNA Secondary Structure Prediction Problem, context-free grammars, the Nussinov Algorithm.

E.1 Introduction

RNA is in many ways similar to DNA, with a few important differences.

1. Thymine (`T`) is replaced by uracil (`U`), so the RNA alphabet is $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{U}\}$.
2. The additional base pair (`G-U`).
3. RNA is less stable than DNA and more easily degraded.
4. RNA mostly occurs as a single-stranded molecule that forms secondary structures by base-pairing.
5. While DNA is mainly the carrier of genetic information, RNA has a wide variety of tasks in the cell, e.g.
   - it acts as messenger (mRNA), transporting the transcribed genetic information,
   - it has regulatory functions,
   - it has structural tasks, e.g. in forming a part of the ribosome (rRNA),
   - it takes an active role in cellular processes, e.g. in the translation of mRNA to proteins, by transferring amino acids (tRNA).

As with proteins, the three-dimensional structure of an RNA molecule is crucial in determining its function. Since 3D-structure is hard to determine, an intermediate structural level, the **secondary structure**, is frequently considered. The secondary structure of an RNA molecule simply determines the intra-molecular basepairs. As with DNA, the Watson-Crick pairs $\{\mathtt{A},\mathtt{U}\}$ and $\{\mathtt{C},\mathtt{G}\}$ stabilize the molecule, but also the "wobble pair" $\{\mathtt{G},\mathtt{U}\}$ adds structural stability. We now define this formally.

Let $s \in \{\mathtt{A},\mathtt{C},\mathtt{G},\mathtt{U}\}^n$ be an RNA sequence of length n.

Definition E.1 An ***s*-basepair** is a set of two different numbers $\{i,j\} \subset \{1,\ldots,n\}$ such that $\{s[i],s[j]\} \in \{\{\mathtt{A},\mathtt{U}\},\{\mathtt{C},\mathtt{G}\},\{\mathtt{G},\mathtt{U}\}\}$. Basepairs must bridge at least $\delta \geq 0$ positions (e.g. $\delta = 3$); to account for this, we have the additional constraint $|i-j| > \delta$ for a basepair $\{i,j\}$.

Definition E.2 A **simple structure on *s*** is a set $S = \{b_1, b_2, \ldots, b_N\}$ of s-basepairs with $N \geq 0$ and the following properties:

- Either $N \leq 1$, or
- if $N \geq 2$ and $\{i,j\} \in S$ and $\{k,l\} \in S$ are two different basepairs such that $i < j$ and $k < l$, then $i < k < l < j$ or $k < i < j < l$ (nested basepairs) or $i < j < k < l$ or $k < l < i < j$ (separated basepairs).

In other words, basepairs in a simple structure may not cross.

The restriction that basepairs may not cross is actually unrealistic, but it helps to keep several computational problems regarding RNA analysis of manageable complexity, as we shall see shortly.

With the above definition of a simple structure, we cover the following structural elements of RNA molecules: **single-stranded RNA**, **stem** or **stacking region**, **hairpin loop**, **bulge loop**, **interior loop**, **junction** or **multi-loop**; see Figure E.1.

Several more complex RNA interactions are *not* covered, e.g. **pseudoknot**, **kissing hairpins**, **hairpin-bulge contact**. This is not such a severe restriction, because these elements seem to be comparatively rare and can be considered separately at a later stage.

We point out that all of the above structural elements can be rigorously defined in terms of sets of basepairs. For our purposes, an intuitive visual understanding of these elements is sufficient. There exist different ways to represent a simple structure in the computer. A popular way is the **dot-bracket** notation, where dots represent unpaired bases and corresponding parentheses represent paired bases:

```
5′ AACGCUCCCAUAGAUUACCGUACAAAGUAGGGCGC 3′
   ..(((.((..((...)).(.(...).)..))))).
```

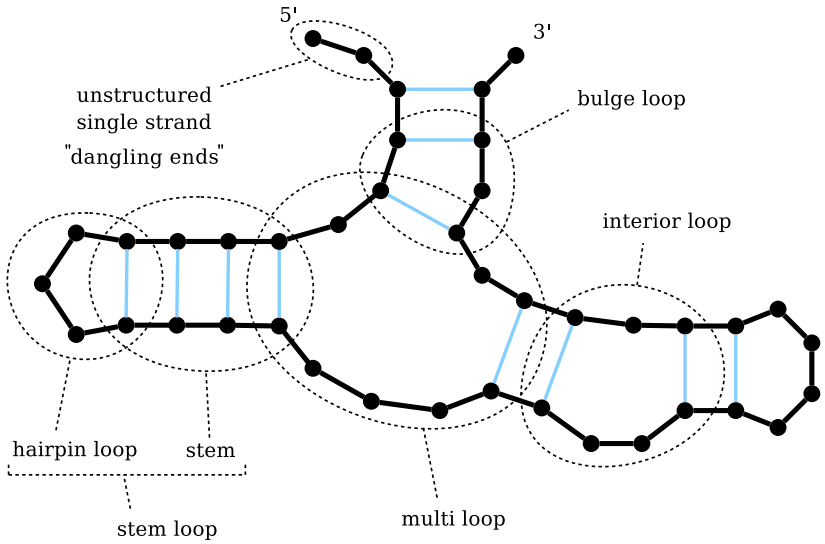


Figure E.1: RNA secondary structure elements covered by Definition E.2.

As an exercise, draw a picture of this structure, similar to that in Figure E.1, and create the dot-bracket representation of the structure in that figure.

Another popular way to visualize a structure is by means of a **mountain plot**: Starting with the dot-bracket representation, replace dots by `-` and brackets `()` by `/\`, and use a second dimension. Another alternative is a **circle plot**; see Figure E.2.

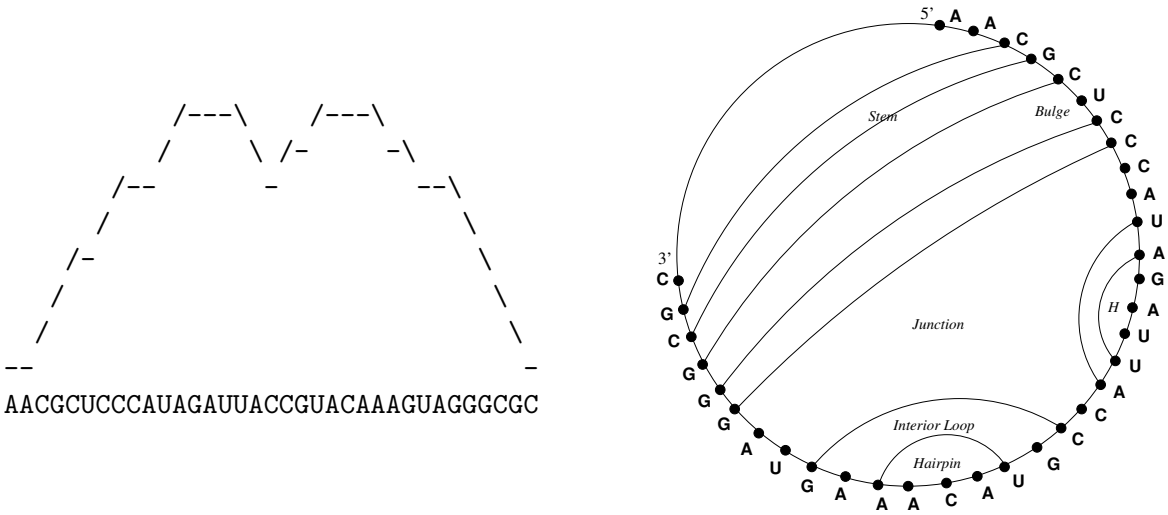


Figure E.2: Mountain plot (left) and circle plot (right) of an RNA secondary structure.

E.2 The Optimization Problem

Over time, an RNA molecule may assume several distinct structures. The more energetically stable a structure, the more likely the molecule will assume this structure. To make this

precise, we need to define an energy model for all secondary structure elements. This can become quite complicated (there is a specialized course on RNA structure prediction), so here we only consider a drastically simplified version of the problem: We assign a score (that can be interpreted as "stability" measure or negative free energy) to each basepair, depending on its type. We can simply count basepairs by assigning +1 to each pair, or we may take their relative stability into account by assigning $score(\{\mathtt{A},\mathtt{U}\}) := 2$, $score(\{\mathtt{C},\mathtt{G}\}) := 3$, $score(\{\mathtt{G},\mathtt{U})\} := 1$, for example. On a sequence s of length n, the score of a basepair $b = \{i, j\}$ is then defined as $score(b) := score(\{s[i], s[j]\})$. The computational problem becomes the following one.

Problem E.3 (Simple RNA Secondary Structure Prediction Problem) Given an RNA sequence s of length n, determine a simple structure S^* on s such that $score(S^*) = \max_S score(S)$ among all simple structures S on s, where $score(S) := \sum_{b \in S} score(b)$ is the sum of the base pair scores in the structure S.

Generally, the number of valid simple structures for a molecule of length n is exponential in n, so the most stable structure cannot be found in reasonable time by enumerating all possible structures. After a short digression to context free grammars, we present an efficient algorithm to solve the problem.

E.3 Context-Free Grammars

From the above discussion of the dot-bracket notation, we may say that a (simple secondary) structure is

- either the empty structure,
- or an unpaired base, followed by a structure,
- or a structure, followed by an unpaired base,
- or an outer basepair (with a sequence constraint), enclosing a structure (with a minimum length constraint),
- or two consecutive structures.

We recall the notion of a **context-free grammar** here to show that simple secondary structures allow a straightforward recursive description.

Definition E.4 A **context-free grammar** (CFG) is a quadruple $G = (N, T, P, S)$, where

- N is a finite set of **non-terminal symbols**,
- T is a finite set of **terminal symbols**, $N \cap T = \emptyset$,
- P is a finite subset of $N \times (N \cup T)^*$, the **productions**, which are rules for transforming a non-terminal symbol into a (possibly empty) sequence of terminal and non-terminal symbols,
- $S \in N$ is the **start symbol**.

Ignoring basepair sequence and distance constraints, in the RNA structure setting, the terminal symbols are $T = \{\texttt{(}, \texttt{.}, \texttt{)}\}$, and $N = \{S\}$, where S is also the start symbol, and represents a secondary structure. The productions are formalized versions of the above rules, i.e.,

- $S \to \varepsilon$, where ε denotes the empty sequence,
- $S \to \texttt{.}S$
- $S \to S\texttt{.}$
- $S \to \texttt{(}S\texttt{)}$
- $S \to SS$

Note that this context-free recursive description is only possible because we imposed the constraint that basepairs do not cross.

Also note that this grammar is ambiguous: There are generally many ways to produce a given dot-bracket string from the start symbol S. For the Nussinov Algorithm in the next section, this ambiguity is not a problem. However, for other tasks, e.g. counting the number of possible structures, we need to consider a grammar that produces each structure in a unique way. We can argue as follows: Either the structure is empty, or it contains at least one base. In each nonempty RNA structure, look at the last base. Either it is unpaired and preceded by a (shorter) prefix structure, or it is paired with some base at a previous position. We obtain the following non-ambiguous grammar:

$$S \to \varepsilon \mid S\texttt{.} \mid S\texttt{(}S\texttt{)}$$

E.4 The Nussinov Algorithm

To solve Problem E.3, we now give a dynamic programming algorithm based on the structural description of simple secondary structures from the previous section. This algorithm is due to Nussinov and Jacobson (1980). We also take into account the distance and sequence constraints for basepairs.

In the previous section, we have seen how to build longer secondary structures from shorter ones that can be arbitrary substrings of the longer ones. As before, let $s \in \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{U}\}^n$ be an RNA sequence of length n. For $i \leq j$, we define $M(i, j)$ as the solution to Problem E.3 (i.e., maximally attainable score) for the string $s[i \ldots j]$.

Since it is easy to find the optimal secondary structure of short sequences, let us start with those.

- If $j - i \leq \delta$ (in particular if $i = j$, and to avoid complications also if $i > j$), then $M(i, j) = 0$, since no basepair is possible due to the length or distance constraints.
- If $j - i > \delta$, then we can decompose the optimal structure according to one of the four cases. In each case, the resulting substructures must also be the optimal ones. Thus

$$M(i,j) = \max \left\{ \begin{array}{l} M(i+1,j), \\ M(i,j-1), \\ \mu(i,j) + M(i+1,j-1), \\ \max_{i+1<k<j} \left[M(i,k-1) + M(k,j) \right] \end{array} \right\} \begin{array}{r} (\text{case } S \to \texttt{.}S) \\ (\text{case } S \to S\texttt{.}) \\ (\text{case } S \to \texttt{(}S\texttt{)}) \\ (\text{all cases } S \to SS) \end{array}$$

where $\mu(i,j) := score(\{s[i], s[j]\}) > 0$ if $\{s[i], s[j]\}$ is a valid basepair, and $\mu(i,j) := -\infty$ otherwise, excluding the possibility of any forbidden basepair $\{i,j\}$.

This recurrence states: To find the optimal structure for $s[i \dots j]$, consider all possible decompositions according to the grammar and their respective scores, and evaluate which score for $s[i \dots j]$ would result from each decomposition. Then pick the best one.

This is possible because the composing elements of the optimal structure are themselves optimal. The proof is by contradiction: If they were not optimal and there existed better sub-structures, we could build a better overall structure from them, which would contradict its optimality.

The above recurrence is not yet an algorithm. We see that, to evaluate $M(i,j)$, we need access to all entries $M(x,y)$ for which $[x,y]$ is a (strictly shorter) sub-interval of $[i,j]$. This suggests to compute matrix M in order of increasing $j-i$ values. The initialization for $j-i \leq \delta$ is easily done. Then we proceed for increasing $d = j-i$, as in Algorithm E.1.

Algorithm E.1 Nussinov-Algorithm

Input: RNA sequence $s = s[1 \dots n]$;
a scoring function for basepairs;
minimum number δ of unpaired bases in a hairpin loop

Output: maximal score $M(i,j)$ for a simple secondary structure of each substring $s[i \dots j]$;
traceback indicators $T(i,j)$ indicating which case leads to the optimum.

```
for i ← 2, ..., n do
    M(i, i−1) ← 0
    T(i, i−1) ← "init"   // ε
for d ← 0, ..., δ do
    for i ← 1, ..., n−d do
        M(i, i+d) ← 0
        T(i, i+d) ← "init"   // .S and/or S.
for d ← (δ+1), ..., (n−1) do
    for i ← 1, ..., n−d do
        compute M(i, i+d) according to the recurrence
        store the maximizing case in T(i, i+d) (e.g. the maximizing k)
report score M(1, n)
start traceback with T(1, n)
```

To find the optimal secondary structure (in addition to the optimal score), we also store traceback indicators in a second matrix T that shows which of the production rules (and which value of k for the $S \to SS$ rule) was used at each step. By following the traceback pointers backwards, we can build the dot-bracket representation of the secondary structure. For this, we start in cell $(1,n)$ and look at the recursion of the algorithm: When we follow a $S \to SS$ production traceback, the process branches recursively into the two substructures, reconstructing each substructure independently. Diagonals mean $S \to (S)$ and horizontal or vertical movement uses $S \to S. \mid .S$. Note that when reaching the δ-diagonals only horizontal and vertical movement is allowed. Finally, when we end in the lowest diagonal (below the $i = j$ diagonal) we have $S \to \varepsilon$.

Complexity. The memory requirement for the whole procedure is obviously $O(n^2)$ and its time complexity is $O(n^3)$.

There exists a faster algorithm using the *Four-Russians* speedup yielding an $O(\frac{n^3}{\log n})$ time algorithm (Frid and Gusfield, 2009), but this goes far beyond the scope of these lecture notes.

Example E.5 The Nussinov matrix for the RNA sequence `GACUCGAGUU`. We use $\delta = 1$ and the scoring scheme: $(\mathtt{G}, \mathtt{C}) = 3, (\mathtt{A}, \mathtt{U}) = 2$ and $(\mathtt{G}, \mathtt{U}) = 1$. As can be seen in Figure E.3, there

		1	2	3	4	5	6	7	8	9	10
		G	A	C	U	C	G	A	G	U	U
1	G	0	0	3	3	5	5	5	6	7	8
2	A	0	0	0	2	2	3	3	5	7	7
3	C		0	0	0	0	3	3	5	5	5
4	U			0	0	0	1	2	3	3	3
5	C				0	0	0	0	3	3	3
6	G					0	0	0	0	2	3
7	A						0	0	0	2	2
8	G							0	0	0	1
9	U								0	0	0
10	U									0	0

δ

Figure E.3: The Nussinov matrix for the RNA sequence `GACUCGAGUU`.

exist two distinct variants (indicated in solid and dashed traces) in which the given RNA sequence can fold under this basepair scoring scheme. The resulting dot-bracket notations are `((.))((.))` and `((((..))))` for the solid and dashed variant, respectively. ■

When reaching the δ-diagonals we could move vertically or horizontally which is possible in the ambiguous grammar. However, since in a hairpin loop a sequence of $S \to .S \mid S.$ will eventually end with $S \to \varepsilon$ no matter whether the `.` is put left or right, this would lead to identical dot-bracket notations. Hence, we could also choose to allow *only* horizontal or *only* vertical movement within the initialized 0-diagonals as to simplify the backtracing of all optimal dot-bracket notations (only 1 backtrace instead of 4 for the dashed variant).

E.5 Further Remarks

As we have already pointed out, the single-basepair scoring model is much too simplistic. In practice, much more complicated energy functions are minimized. The basic idea, however, to exploit the recursive optimal substructure property, remains the same.

A specialized course on RNA structure prediction (by Robert Giegerich) also poses and answers the following problems, among others:

- How can we obtain suboptimal structures, and more importantly, classes of fundamentally different structures ("RNA shapes") and count their members?
- Given a description of a secondary structure, how can we efficiently find RNA sequences (e.g. in a genome) that match this structure?
- How can we align RNA sequences, taking structural considerations into account?
- How can be computationally consider complex structures such as kissing hairpins or pseudoknots?

As RNA is now considered to be one of the most fundamentally important molecules in evolution, answers to these questions have far-reaching implications.

A broad range of programs for analyzing, predicting and visualizing RNA secondary structures under different models and using different structure elements is available at:

`http://bibiserv.techfak.uni-bielefeld.de/bibi/Tools_RNA_Studio.html`.

APPENDIX F

Suffix Tree (Extended Material)

Contents of this chapter: Memory representation of suffix trees.

F.1 Memory Representations of Suffix Trees

Theorem 7.2 is important in both theory and practice: The index only requires a constant factor more space than the text. In practice, however, the exact constant factor is also important: Does the tree need 10, 100, or 1000 times the size of s? Here we discuss a memory-efficient way to store the tree.

We assume that the text length is bounded by the maximum integer size (often 2^{32} for unsigned integers, or 2^{31} for signed integers); thus storing a number or a reference to a suffix takes 4 bytes.

Above, we said that edge labels could be stored as a pair of numbers (8 bytes per edge). In fact, just one number (4 bytes per edge) is sufficient: Note that the labels of all leaf edges end at position n. Thus for leaf edges it is sufficient to store the starting position. Now we traverse the tree bottom-up: For edges e into internal nodes v, consider the leftmost outgoing edge of v and its already determined starting position j. It follows that we can label e by $(j-k, j-1)$ (if the length of the edge label is k) and only need to store the starting position $j-k$, since we can retrieve $j-1$ by looking up the starting position j of the first outgoing edge of the child.

We can store the edges to the children of each node consecutively in memory as follows.

- For an edge to an internal node, we store two integers: The pointer into the string as described above in order to reconstruct the edge label, and a reference to the child list of the target node.

- For an edge to a leaf, we only store one integer: The pointer into the string as described above in order to reconstruct the edge label.

In the field where we store the string pointer, we also use two flags (that can be coded as the two highest bits of the integer, effectively restricting the size of the string further):

1. a leaf-flag L to indicate a leaf edge,
2. a last-child flag $*$ to indicate the last child in a child list.

Using the two flags we can store 30 bit numbers in an 32 bit integer value:

bit index	31	30	29	...	2	1	0
content	L	*	30 bit number				

The order in which we store the nodes is flexible, but it is customary (and integrated well with the WOTD algorithm described below) to store them in depth-first pre-order. This means, we first store the children of the root, then the first child of the root and its children recursively, then the second child of the root, and so on.

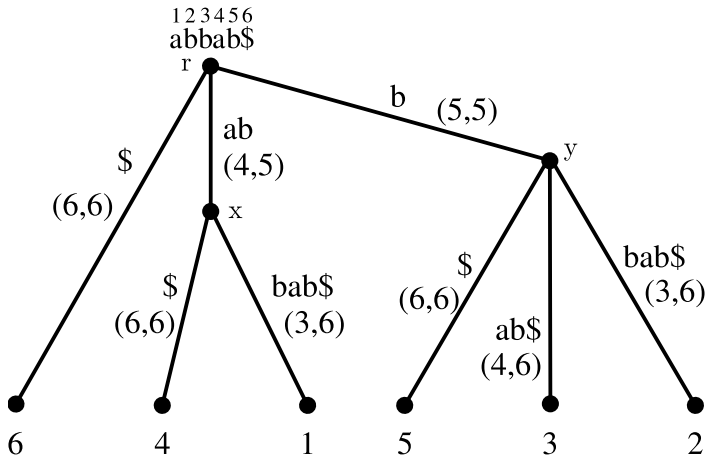


Figure F.1: The suffix tree of abbab$ with edge labels, substring pointers, and leaf labels.

For example, the suffix tree in Figure F.1 would be stored as follows (**bold** numbers indicate memory positions while string indices are in normal font).

parent node	r					x		y		
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
content	$\mathbf{6_L}$	**4**	**5**	**5***	**7**	$\mathbf{6_L}$	$\mathbf{3^*_L}$	$\mathbf{6_L}$	$\mathbf{4_L}$	$\mathbf{3^*_L}$
points to	6		(x)		(y)	4	1	5	3	2

Note how the edge to the child (x) of (r) is represented by two numbers. First, the 4 indicates the starting position of the substring (ab) to read along the edge. Next, the 5 directly points to the memory location where the children of x are stored. Since it begins with 6, we know that the above edge ends at string position 5 and hence has length 2.

It remains to answer the question how we can deduce toward which leaf a leaf edge points. While we move in the tree, we always keep track of the current string depth. This is straightforward, since we can deduce the length of each edge as described above. We just have to subtract the current string depth from the string pointer on a leaf edge to obtain the leaf label.

For example, the root has string depth zero and the edge into x has length 2, so we know that x has a string depth of 2. The second child of x is a leaf with substring pointer 3. Subtracting the string depth of x, we see that it points to leaf 1.

The memory requirements for storing the suffix tree in this way are 2 integers for each internal node (except the root) and 1 integer for each leaf. This assumes that the flags L and $*$ are stored as high-order bits inside the integers.

We point out that there are many other ways to organize the storage of a suffix tree in memory, especially if we want to store additional annotations for the internal nodes or leaves.

APPENDIX G

Advanced Topics in Pairwise Alignment (Extended Material)

Contents of this chapter: Length-normalized alignment, shadow effect, mosaic effect, optimal normalized alignment score, parametric alignment, ray search problem.

G.1 Length-Normalized Alignment

G.1.1 A Problem with Alignment Scores

A convenient feature of the usual score definition of (local or global) alignments is its additivity. This allows a simple computation of the score (by summing the scores of each column) and is also the basis for all alignment-related dynamic programming algorithms that we have seen so far.

However, for *local* alignment, additivity also leads to at least two undesired effects from a biologist's perspective (see also Figure G.1):

Shadow effect (long mediocre alignments mask short excellent alignments): Consider two candidate local alignments A_1 and A_2. Assume that A_1 achieves a score of 100 by aligning, say, 50 nucleotides of each sequence; further assume that there is no alignment with a higher score. We do not know how A_1 looks (i.e., if there are many or few gaps or mismatches inside); all we can say is that the average score per alignment column is 2. Now assume that A_2 achieves a score of 99 (and is therefore slightly suboptimal), but it only aligns 33 nucleotides of each sequence, thus is much shorter. On average, the score per alignment column is 3. We could argue that in fact this is a better alignment, but it will not be found by the Smith-Waterman algorithm because it scores slightly below the optimum. See Figure G.1, left-hand side.

Mosaic effect (inclusion of arbitrarily poor regions between two high-scoring ones): Consider an alignment that is well conserved at its borders, but has an extremely bad (thus low-scoring) middle part. For concreteness, let us assume that the score of the three parts is $(+100, -99, +100)$. This gives a total score of 101 and is thus better than each of the two separate border parts. However, a biologist might in fact be only interested in those, but has to read through a lot of "noise" because of additive scoring. See Figure G.1, right-hand side.

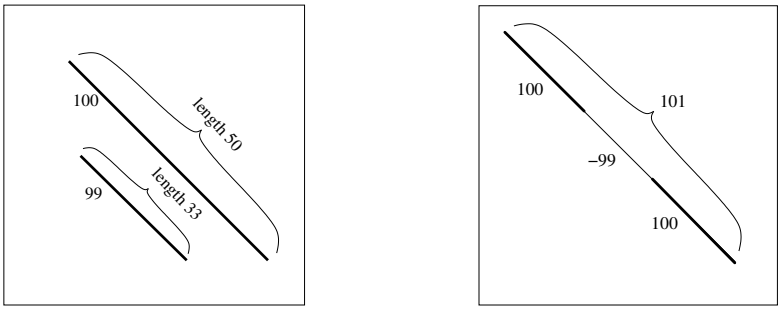


Figure G.1: Left: Shadow effect. Right: Mosaic effect.

Let us emphasize that the above problems do *not* indicate that the Smith-Waterman algorithm is incorrect. It correctly solves the local alignment problem, as specified previously, with an additive scoring function. What we are facing here is instead a **modeling problem**. There may be something wrong (or at least, not ideal) with the way we (or the whole computational biology community) have defined the quality (score) of a local alignment. It is important to keep modeling issues separate from algorithmic issues.

How can we deal with the above problems? Of course, the cleanest solution is to re-define the objective function. However, this leads to an entirely new computational problem, for which so far we may not have an algorithm. (No need to worry, in this section we will develop one!) In practice and under the usual time constraints, though, we may not want to develop new theories, although this is often the most interesting part of working as a computational biologist. The usual approach is to deal with such problems in an ad-hoc manner that works well in practice.

We already know how to compute suboptimal Smith-Waterman alignments, so we can compute some of them and re-evaluate them with different scoring functions. Such a heuristic assumes, of course, that good alignments under our new scoring function are also not too bad Smith-Waterman alignments. This cannot always be guaranteed, however. Other ad-hoc methods would be postprocessing Smith-Waterman alignments to cut out bad regions in the middle of an alignment, effectively splitting one alignment into two shorter but (relatively) better ones.

Interestingly, the popular BLAST heuristic (Altschul et al., 1990) proceeds quite cleverly, as we discussed in Section 6.3: Starting from well conserved seeds, it extends the alignment only until a once reached score drops too far and then again removes the bad border region. This effectively avoids the mosaic effect and leads to shorter, better conserved alignments. So although BLAST is sometimes accused of not finding the best Smith-Waterman alignment, this behavior in fact has a good reason.

G.1.2 A Length-Normalized Scoring Function for Local Alignment

The above problems suggest that it is better to consider a *score per alignment column* or a related quantity. In this section, we follow Arslan et al. (2001) and normalize the score by the sum of the lengths of the aligned substrings.

Recall that the standard optimal global and local alignment scores of s and t – let us call them $GAS(s,t)$ and $LAS(s,t)$, respectively – were defined as

$$GAS(s,t) := \max_{A \text{ Alignment of } s \text{ and } t} score(A),$$
$$LAS(s,t) := \max_{\text{substrings } s' \blacktriangleleft s,\ t' \blacktriangleleft t} GAS(s',t').$$

This relation is useful in many contexts: the local alignment score is in fact a global alignment score, once we know which substrings we have to align! The new idea is to normalize $GAS(s',t')$ with a length-related quantity.

The length of an alignment can be defined in different ways. For our purposes, the following convention is useful: The **length of an alignment** of (sub)strings s', t' is defined as $|s'|+|t'|$. (The obvious alternative, taking the number of columns of the alignment, is also possible.)

Thus we could look for substrings s', t' maximizing $GAS(s',t')/(|s'|+|t'|)$. This would give high preference to very short alignments (think about the two empty substrings!), so this idea still needs modifications. We could enforce the constraint $|s'|+|t'| \geq L_{\min}$, where $L_{\min} > 0$ is a minimum length parameter. Here we use a different, but related approach. Let $L > 0$ be a free parameter. We define the **optimal normalized alignment score** (with parameter L) as

$$NAS_L(s,t) := \max_{\text{substrings } s' \blacktriangleleft s,\ t' \blacktriangleleft t} GAS(s',t')/(|s'|+|t'|+L).$$

For $L \searrow 0$, very short alignments will be optimal. For $L \nearrow \infty$, the lengths of the substrings become less important, and we eventually obtain the same results as for standard local alignment.

G.1.3 Finding an Optimal Normalized Alignment

Now that we have defined the objective function, we need an efficient algorithm to compute $NAS_L(s,t)$.

The first (inefficient) possibility is to apply the definition: For each of the $O(m^2n^2)$ pairs of substrings $s' \blacktriangleleft s$ and $t' \blacktriangleleft t$, compute $GAS(s',t')/(|s'|+|t'|+L)$ in $O(mn)$ time and keep track of the maximum. This leads to a straightforward $O(m^3n^3)$, or, if $m = \Theta(n)$, an $O(n^6)$ time algorithm, which is very impractical. By re-using some information during the computation, it is easy to speed this up to an $O(m^2n^2)$ or $O(n^4)$ time algorithm, still not practical.

Fortunately, there is a better way. The main problem is that the target function to be maximized is not additive anymore (but this was the whole idea behind the change). We shall see that we can iteratively solve a series of standard alignment problems with changing scoring parameters to find the optimal length-normalized alignment.

Consider any alignment A. It consists of a certain number $n_{a,b}(A)$ of substitutions (or identities) $a \to b$, where $(a,b) \in \Sigma^2$, of a certain number $g(A)$ of contiguous gaps, and of a certain number $b(A)$ of blanks (gap characters). In this section it is useful to distinguish between the gap characters (which we call blanks) and the contiguous gaps, which consist of one or more blanks. For each gap's first blank, we pay an opening penalty of d, for each further blank, we pay an extension penalty of e.

The total score $score(A)$ of an alignment A is thus

$$score(A) = \Big(\sum_{(a,b)\in\Sigma^2} n_{a,b}(A) \cdot w(a,b) \Big) - g(A) \cdot d - (b(A) - g(A)) \cdot e.$$

The length $|s'| + |t'|$ of an alignment A of two substrings s' and t' is furthermore

$$length(A) = |s'| + |t'| = \Big(2 \cdot \sum_{(a,b)\in\Sigma^2} n_{a,b}(A) \Big) + b(A),$$

since each substitution column contains two characters and each column with a blank contains one character (of the other sequence).

Observation G.1 Both $score(A)$ and $length(A)$ are linear functions of the counts $(n_{a,b}(A))$, $g(A)$ and $b(A)$. In contrast, $score(A)/(length(A) + L)$ is a rational function of these counts.

The main idea is as follows: What we want to solve is the fractional problem

$$(P): \text{ Find } S^{max} = \max_A S(A), \text{ where } S(A) = \frac{score(A)}{length(A) + L}.$$

In order to solve this problem, we introduce a new parameter $\lambda \geq 0$ and define the parameterized problem

$$(P_\lambda): \text{ Find } S_\lambda^{max} = \max_A S_\lambda(A), \text{ where } S_\lambda(A) = score(A) - \lambda \cdot \big(length(A) + L\big).$$

First, in Lemma G.2, we show that for each fixed choice of λ, (P_λ) is a standard Smith-Waterman alignment problem with a scoring scheme that depends on λ; thus we know how to solve it. Next, in Lemma G.3, we show that we can solve (P) searching for the right λ by a particular strategy, which is fortunately easy to describe.

Lemma G.2 Problem (P_λ) can be solved by solving a standard Smith-Waterman alignment problem with a different scoring function. If $w(a,b)$, d, e are the substitution scores, gap open and gap extend costs of an instance of the parameterized alignment problem (P_λ), respectively, then define:

$$w_\lambda(a,b) = w(a,b) - 2\lambda, \qquad d_\lambda = d + \lambda, \qquad e_\lambda = e + \lambda.$$

The optimal score S_λ^{max} of (P_λ) is the solution $score_\lambda^{max}$ of the standard Smith-Waterman alignment problem using $w_\lambda(a,b)$, d_λ, e_λ, minus λL : $S_\lambda^{max} = score_\lambda^{max} - \lambda L$.

Proof. Let A be any valid alignment. Its score in (P_λ) is

$$\begin{aligned}
S_\lambda(A) &= score(A) - \lambda \cdot \big(length(A) + L\big) \\
&= \Big[\sum_{(a,b)\in\Sigma^2} n_{a,b}(A) \cdot w(a,b) - g(A) \cdot d - (b(A) - g(A)) \cdot e\Big] - \lambda\Big[2 \cdot \sum_{(a,b)\in\Sigma^2} n_{a,b}(A) + b(A) + L\Big] \\
&= \sum_{(a,b)\in\Sigma^2} n_{a,b}(A) \cdot \big(w(a,b) - 2\lambda\big) - g(A) \cdot (d - e) - b(A) \cdot (e + \lambda) - \lambda L. \\
&= \sum_{(a,b)\in\Sigma^2} n_{a,b}(A) \cdot w_\lambda(a,b) - g(A) \cdot d_\lambda - \big(b(A) - g(A)\big) \cdot e_\lambda - \lambda L \\
&= score_\lambda(A) - \lambda L
\end{aligned}$$

where $score_\lambda(A)$ is the SW-score of A under the modified scoring scheme $w_\lambda(a,b), d_\lambda$ and e_λ. Note that $-\lambda L$ is a constant, thus we need not consider it during maximization, therefore the statement of the lemma holds. □

Lemma G.3 For fixed λ, let S_λ^{max} be the solution of (P_λ), i.e.,

$$S_\lambda^{max} = \max_A score(A) - \lambda \cdot (length(A) + L).$$

Let S^{max} be the solution of (P), i.e., $S^{max} = \max_A \frac{score(A)}{length(A)+L}$. Then the following holds:

$$\begin{aligned}
S_\lambda^{max} = 0 &\iff S^{max} = \lambda, \\
S_\lambda^{max} < 0 &\iff S^{max} < \lambda, \\
S_\lambda^{max} > 0 &\iff S^{max} > \lambda.
\end{aligned}$$

Proof. First note that always $length(A) + L > 0$, since $L > 0$.

The definition of $S^{max} = \max_A \frac{score(A)}{length(A)+L}$ is equivalent to stating that for *all* alignments A $score(A) - S^{max} \cdot (length(A) + L) \leq 0$ and that there exists an alignment A^{max} for which equality holds:

$$score(A^{max}) - S^{max} \cdot (length(A^{max}) + L) = 0.$$

Thus for $S^{max} = \lambda$ we have $score(A^{max}) - \lambda(length(A^{max}) + L) = 0$, and at the same time $S_\lambda^{max} = score(A^{max}) - \lambda \cdot (length(A^{max}) + L)$, thus $S_\lambda^{max} = 0$.

Now consider that $S^{max} < \lambda$. Then for all alignments A, we have $S_\lambda = score(A) - \lambda \cdot (length(A) + L) < score(A) - S^{max} \cdot (length(A) + L) \leq 0$. Thus also $S_\lambda^{max} < 0$.

Taking $S^{max} > \lambda$ and considering A^{max} shows $S_\lambda^{max} > 0$.

This proves the three implications of the lemma in the $\Leftarrow$ direction. But since the three possibilities are exhaustive on each side, equivalence follows. □

The lemma tells us that if the *optimal* score S_λ^{max} of the parameterized problem (which is a Smith-Waterman alignment problem that we can solve) is zero, then we have found the optimal normalized score S^{max} of the normalized alignment problem, namely the parameter λ that led to the optimal score zero.

This suggests a simple bisection algorithm:

1. Identify an initial interval $[l, u]$ such that $l \leq S^{max} \leq u$.
2. Compute the midpoint $\lambda = (l + u)/2$.
3. Solve (P_λ). Let S_λ^{max} be the optimal score.
4. If $S_\lambda^{max} = 0$, we have found $S^{max} = \lambda$ and stop.
   If $S_\lambda^{max} < 0$, then $S^{max} < \lambda$, so we set $u := \lambda$ and continue searching in the lower half interval; otherwise, we set $l := \lambda$ and continue searching in the upper half interval: Go back to 2.

When the algorithm stops, the current value of λ is $S^{max} = NAS_L(s, t)$ and the last alignment computed in step 3 is the optimal normalized alignment.

It remains to find the initial interval in step 1: We claim that $0 \leq S^{max} \leq M/2$ is a safe choice, where $M > 0$ is the highest possible match score between characters. This is seen as follows. For $\lambda = 0$, we compute the ordinary Smith-Waterman alignment without subtracting a penalty of the length. This clearly has a nonnegative optimal score, thus $S_0^{max} \geq 0$, hence $S^{max} \geq 0$. If M is the highest positive match score, no alignment can score more than M per column, or $M/2$ per aligned character, thus $S^{max} \leq M/2$.

Example G.4 Consider two strings $s = GAGTT$ and $t = AGT$, let $L = 2$, and choose the following scores: $+2$ for matches, 0 for mismatches and -1 for indels (linear homogeneous). The task is to find the optimal normalized alignment score for s and t, in particular a value of λ such that the optimal score S_λ^{max} is zero.

First, observe that $M = +2$ is the highest match score. Moreover, $d_\lambda = e_\lambda$ because we use homogeneous gap costs.

Phase 1:

1. $[\ell, u] = [0, \frac{M}{2}] = [0, 1]$
2. $\lambda = \frac{1}{2}(\ell + u) = \frac{1}{2} \cdot 1 = \frac{1}{2}$
3. Modified scoring scheme:

$$\begin{cases} 2 - 2 \cdot \lambda, & w_\lambda(a, a) \\ 0 - 2 \cdot \lambda, & w_\lambda(a, b) \\ -(1 + \lambda), & d_\lambda, e_\lambda \end{cases} \Rightarrow \begin{cases} 2 - 2 \cdot \frac{1}{2} = 1, & w_{1/2}(a, a) \\ 0 - 2 \cdot \frac{1}{2} = -1, & w_{1/2}(a, b) \\ -(1 + \frac{1}{2}) = -\frac{3}{2}, & d_{1/2}, e_{1/2} \end{cases}$$

Smith-Waterman alignment:

	ε	G	A	G	T	T
ε	0	0	0	0	0	0
A	0	0	1	0	0	0
G	0	1	0	2	1/2	0
T	0	0	0	1/2	3	3/2

$\Rightarrow score_{1/2}^{max} = 3$

$S_\lambda^{max} = score_\lambda^{max} - \lambda \cdot L \quad \Rightarrow \quad S_{1/2}^{max} = score_{1/2}^{max} - \frac{1}{2} \cdot L = 3 - \frac{1}{2} \cdot 2 = 2$
$S_{1/2}^{max} > 0 \quad \Rightarrow \quad \ell = \lambda$ go to Phase 2

Phase 2:

1. $[\ell, u] = [\lambda, u] = [\frac{1}{2}, 1]$
2. $\lambda = \frac{1}{2}(\ell + u) = \frac{1}{2} \cdot \frac{3}{2} = \frac{3}{4}$
3. Modified scoring scheme:

$$\begin{cases} 2 - 2 \cdot \frac{3}{4} = \frac{1}{2}, & w_{3/4}(a,a) \\ 0 - 2 \cdot \frac{3}{4} = -\frac{3}{2}, & w_{3/4}(a,b) \\ -(1 + \frac{3}{4}) = -\frac{7}{4}, & d_{3/4}, e_{3/4} \end{cases}$$

Smith-Waterman alignment:

	ε	G	A	G	T	T
ε	0	0	0	0	0	0
A	0	0	1/2	0	0	0
G	0	1/2	0	1	0	0
T	0	0	0	0	3/2	1/2

$\Rightarrow score_{3/4}^{max} = \frac{3}{2}$

$S_{3/4}^{max} = score_{3/4}^{max} - \frac{3}{4} \cdot L = \frac{3}{2} - \frac{3}{4} \cdot 2 = 0$
$S_{3/4}^{max} = 0 \quad \Rightarrow$ Found $\lambda = \frac{3}{4}$.

Thus, we have found our $\lambda = \frac{3}{4}$ and our optimal normalized alignment score for s and t in only two recursions. ■

Note. There are cases in which the algorithm takes infinitely many steps if calculated precisely, for example for $\lambda = \frac{1}{2}$. In such cases, to reach numerical stability, after a certain number of steps or when a satisfying precision is reached, the procedure should be stopped.

Algorithm G.1 Dinkelbach's algorithm for normalized alignment; DANA(s, t, L): returns $NAS_L(s, t)$

```
λ ← 0, s ← −∞
while s ≠ 0 do
    Solve (P_λ) to obtain optimal score S_λ^max and alignment A
    λ ← score(A)/(length(A) + L)
return λ
```

Note that s refers to the score under the modified scoring scheme using parameter λ, while $score(A)$ refers to the unmodified scoring scheme.

Another way to find S^{max} (instead of the bisection approach) is to use a method distantly inspired by Newton's method. Once we have an alignment A from step 3 with nonzero S_λ^{max}, we ask how we have to modify λ so that this alignment obtains a score of zero. Of course, we expect that another alignment becomes optimal once we change λ. Nevertheless, choosing λ in this way brings us closer and closer to the solution.

This latter technique of solving a fractional optimization problem by solving a series of linear problems is known as **Dinkelbach's algorithm**; see Algorithm G.1. We do not give a proof of convergence, nor a worst-case analysis (which is not very good) of Dinkelbach's algorithm. In practice, Dinkelbach's algorithm is efficient (more so than the bisection method) and usually requires only 3–5 iterations and rarely more than 10. With a more complicated algorithm (which we shall not discuss), it can be shown that the length-normalized local sequence alignment problem can be solved in $O(nm \log n)$ time (i.e, in $O(\log n)$ iterations).

G.2 Parametric Alignment

G.2.1 Introduction to Parametric Alignment

Several parameters are involved in the computation of pairwise sequence alignments, and depending on their values, different alignments may be optimal. In fact, the choice of alignment parameters is crucial when we want to find biologically meaningful alignments. Section G.1 on normalized alignment has already given an indication of this: By changing the scoring scheme appropriately, we were able to solve the length-normalized local alignment problem. The present section contains an approach for the systematic study of the relationship between parameters and optimal alignments, the *parametric sequence alignment*.

The two most obvious parameter sets in pairwise alignment are

1. the substitution scores for every pair of symbols from the alphabet; these can often be derived empirically as log-odds scores, as outlined in Section A.3;
2. the gap cost function; for practical and efficiency reasons, we almost always use affine gap costs, where we have to specify the gap open penalty d and the gap extension penalty e.

The gap parameters are particularly difficult to choose from a theoretical basis. In the following, we assume that the substitution scores are fixed and that d and e are free parameters. Having only two parameters has the additional advantage that we can easily visualize the whole parameter space as a two-dimensional plane.

The study of this parameter space is useful in various ways:

1. Choice of alignment parameters: Assume that two sequences are given, together with their "correct" alignment. Then one can look at the above described plane and test if this alignment occurs, and if it does, for which gap parameters. This way values for the gap parameters can be found that (at least for the given pair of sequences) will lead to a meaningful alignment.
2. Robustness test: The alignments that are found in the vicinity of the point in the (d, e)-plane corresponding to the parameters used to obtain a certain optimal alignment A give an idea about the robustness of that alignment, i.e., whether A would also be optimal if the alignment parameters were chosen a bit differently.
3. Efficient computation of all co-optimal alignments: Knowledge of the shape of the parameter space allows to avoid redundant computations if all co-optimal alignments are seeked. Moreover, one can also restrict these computations to those alignments that have support by wider ranges than just a single (almost arbitrary) point in the parameter space (Gusfield, 1997).

But how is it possible to compute the vicinity of a certain point, or even to find out if and where a given alignment occurs in the plane?

The first approach would probably be to *sample* the plane, for example by systematically computing optimal alignments for many different parameter settings. This approach has its disadvantages:

- Even if many parameter values are tested, it is still possible that the parameters giving the "true" alignment are not among the tested ones. This could lead to wrong conclusions such as the one that a "biologically true" alignment can never be obtained as a "mathematically optimal" alignment, although only the parameters for obtaining the biologically correct one were not among those used in the sampling procedure.
- Many different parameter settings may yield the same optimal alignment, which is then computed several times. By a more clever strategy, these redundant computations can be avoided.

We will see in the following that there exists a more systematic way to compute the optimal alignments for *all* parameter settings.

G.2.2 Theory of Parametric Alignment

Our description follows Gusfield (1997, Section 13.1). We first repeat a key observation from Section G.1.

Consider any alignment A. It consists of a certain number $n_{a,b}(A)$ of substitutions (or identities) $a \to b$, where $(a, b) \in \Sigma^2$, of a certain number $g(A)$ of contiguous gaps, and of a certain number $b(A)$ of blanks (gap characters). Again, we distinguish between the gap characters (blanks) and the contiguous gaps, which consist of one or more blanks. For each gap's first blank, we pay an opening penalty of d, for each further blank, we pay an extension penalty of e.

$$S(A) = \Big(\sum_{(a,b)\in\Sigma^2} n_{a,b}(A) \cdot score(a, b) \Big) - g(A) \cdot d - \big(b(A) - g(A)\big) \cdot e.$$

Observation G.5 For a given alignment A, the score $S(A)$ is a linear function of the parameters d and e; let us write $S_A(d, e) := S(A)$.

For a given alignment, consider the following set of points: $P(A) := \{(d, e, S_A(d, e))\}$, where the reasonable values of d and e define the parameter space. Since $S_A(d, e)$ is linear, this set of points is a two-dimensional plane in three-dimensional space.

Lemma G.6 Let A and A' be two alignments (of the same sequences). If $P(A)$ and $P(A')$ intersect and are distinct, then there is a line L in the (d, e)-plane along which A and A' have equal score. If the planes do not intersect, then one of the two alignments has always a better score than the other.

Proof. In 3D-space, two planes either intersect in a line or are parallel. If the $P(A)$ and $P(A')$ intersect, then $S_A(d, e) = S_{A'}(d, e)$ by definition for each (d, e) along the intersection line. If the planes are parallel (and hence do not intersect) and distint, always $S_A > S_{A'}$ or vice versa in the whole parameter space. □

The line L divides $P(A)$ into two half-planes, inside one of which A has higher score than A', and in the other of which A' has higher score. The region where A is an optimal alignment is defined by the projection onto (d, e)-space of the intersection of many of such half-planes. The resulting area is necessarily a convex polygon. Thus we have:

Observation G.7 If A is optimal for at least one point p in the (d, e)-plane, then it is optimal either for only this point, or for a line through p, or for a convex polygon that contains p.

Putting things together, we obtain the following result, which is illustrated in Figure G.2.

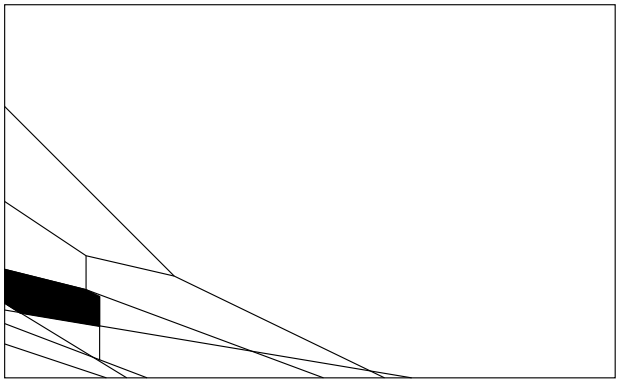


Figure G.2: Illustration of the (d, e)-plane and its decomposition into convex polygons. The picture was generated with XPARAL (see Section G.2.7 on page 183), with d at the x-axis and e at the y-axis.

Theorem G.8 Given two strings s and t, the (d, e)-plane decomposes into convex polygons, such that any alignment that is optimal for some point $p_0 = (d_0, e_0)$ in the interior of a polygon P_0, is optimal for all points in P_0 and nowhere else.

After these analytical and geometrical considerations, we now formally state the algorithmic problem we would like to solve:

Problem G.9 (Parametric Alignment Problem) Given two sequences s and t and fixed substitution scores, find the polygonal decomposition of the (d, e) parameter space, and for each polygon, find one (or all) of its optimal alignments.

We will solve this problem in several steps. Essential in all of these steps will be a technique called *ray search*, defined by the following problem statement:

Problem G.10 (Ray Search Problem) Given an alignment A, a point p where A is optimal, and a ray h in the (d, e)-plane starting at p, find the furthest point (call it r^*) from p on ray h where A remains optimal.

G.2.3 Solving the Ray Search Problem

Algorithm G.2 solves the Ray Search Problem. It also returns another alignment A' that has the same score as A in the boundary point r^* and hence is co-optimal at this point.

In step 4, the point r is found first solving the equation $S_A(d, e) = S_{A'}(d, e)$ for d and e, yielding a line in the (d, e)-plane. The intersection point between this line and the ray h is then r.

Algorithm G.2 Newton's ray search algorithm: For a given point p where the given alignment A is optimal and a direction (ray) h, it returns the furthest point r on h in (d, e)-space for which A is still optimal. It also returns an alignment A' that is co-optimal at point r.

```
set r to the point where h intersects the border of the parameter space
set A' = ⊥ (undefined)
while A is not an optimal alignment at point r do
    find an optimal Alignment A' at point r
    set r to the unique point on h where the value of A equals the value of A'
return (r, A')
```

Example G.11 Given $s = GAG$ and $t = AATTG$ with fixed scores $+3$ for matches and -3 for mismatches (use $-d - e \cdot (\ell - 1)$ for gaps of length ℓ), let $p := (4, 2)$ be a point in the (d, e)-parameter space. Let h be the ray defined by its origin p and the direction $(-2.5, -1)$. Calculate the optimal alignment A of s and t in p, as well as the farthest point on h, for which A is still optimal.
We denote two consecutive single gaps (d+d) as $\overline{d}\,\overline{d}$ and a long gap (d+e) as $\overline{d}\,\overline{e}$.

Given: Point $p = (4, 2)$, line $h(d) = \frac{2}{5}d + \frac{2}{5}$, see Figure G.3.

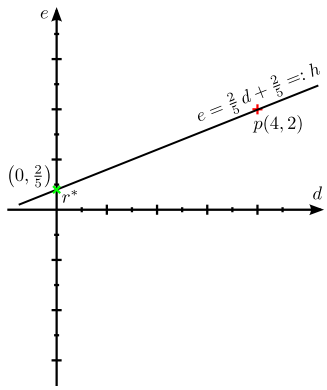


Figure G.3: Line $h(d)$ with its origin $p(4, 2)$ and its intersection with the y-axis $r^* = (0, \frac{2}{5})$

Step 1: Calculate $\mathcal{A}_1$ in $p(4, 2)$ and its score.

$$\begin{aligned} score_{4,2}(\mathcal{A}_1) &= score_{4,2}\begin{pmatrix}\texttt{AATTG}\\ \texttt{GA}\overline{d}\,\overline{e}\texttt{G}\end{pmatrix} = -3 \\ score(\mathcal{A}_1) &= 2 \cdot \text{match} + 1 \cdot \text{mismatch} - d - e \\ &= 6 - 3 - d - e \\ &= \boxed{3 - d - e} \end{aligned} \tag{i}$$

Set r^* to the point where h intersects the border of the parameter space: $r^* = (0, \frac{2}{5})$ and calculate optimal alignment $\mathcal{A}_2$ in r^*. Because $\mathcal{A}_1 \neq \mathcal{A}_2$, we need to calculate the score:

$$score_{0,\frac{2}{5}}(\mathcal{A}_2) = score_{0,\frac{2}{5}}\begin{pmatrix}\texttt{A}\overline{d}\texttt{ATTG}\\ \overline{d}\texttt{GA}\overline{d}\,\overline{d}\texttt{G}\end{pmatrix} = 6$$

$$\begin{aligned} score(\mathcal{A}_2) &= 2 \cdot \text{match} - 4 \cdot d \\ &= \boxed{6 - 4\,d} \end{aligned} \tag{ii}$$

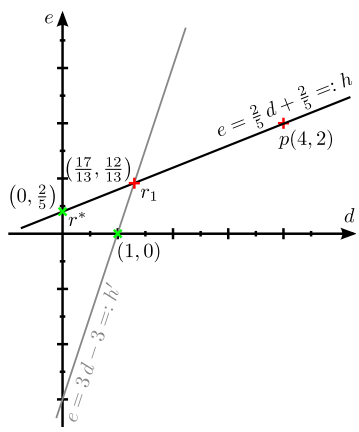
Set $Score(\mathcal{A}_1) = Score(\mathcal{A}_2) \quad \Rightarrow$ line h' (all points, where $\mathcal{A}_1$ and $\mathcal{A}_2$ have the same score)

$$\begin{aligned} score(\mathcal{A}_1) &= score(\mathcal{A}_2) \\ (i) &= (ii) \\ 3 - d - e &= 6 - 4\,d \\ \Leftrightarrow \quad e &= \boxed{3\,d - 3 := h'(d)} \end{aligned} \tag{iii}$$

Set $h(d) = h'(d) \quad \Rightarrow$ Intersection point r_1 on h with $score(\mathcal{A}_1) = score(\mathcal{A}_2)$

$$\begin{aligned} h(d) &= (iii) \\ \tfrac{2}{5}\,d + \tfrac{2}{5} &= 3\,d - 3 \\ \Leftrightarrow \quad \tfrac{3\cdot 5}{5}\,d - \tfrac{2}{5}\,d &= \tfrac{2}{5} + \tfrac{3\cdot 5}{5} \\ \Leftrightarrow \quad \tfrac{13}{5}\,d &= \tfrac{17}{5} \\ \Leftrightarrow \quad d &= \tfrac{17}{13} \quad \Rightarrow \quad e = 3\left(\tfrac{17}{13} - 1\right) = \tfrac{12}{13} \\ \Rightarrow r_1 &= \left(\frac{17}{13}, \frac{12}{13}\right) \end{aligned} \tag{iv}$$

Calculate optimal alignment in r_1. However, neither $\mathcal{A}_1$ nor $\mathcal{A}_2$ are optima in r_1!



Step 2: Continue searching for the correct A.
Let $\mathcal{A}_3$ be the calculated optimum at point r_1. Calculate its score:

$$score_{\frac{17}{13},\frac{12}{13}}(\mathcal{A}_3) = score_{\frac{17}{13},\frac{12}{13}}\begin{pmatrix} \overline{d}\texttt{AATTG} \\ \texttt{GA}\overline{d}\,\overline{e}\,\overline{e}\texttt{G} \end{pmatrix} = \frac{20}{13} = 1\frac{7}{13}$$

$$\begin{aligned} score(\mathcal{A}_3) &= 2 \cdot \text{match} - d - d - 2 \cdot e \\ &= \boxed{6 - 2\,d - 2\,e} \end{aligned} \tag{v}$$

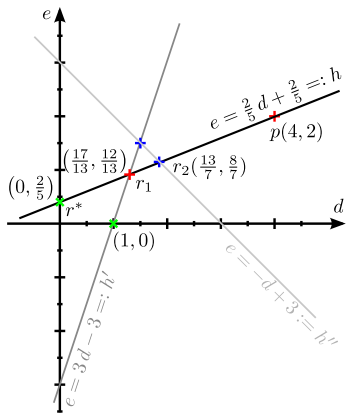
Set $score(\mathcal{A}_1) = score(\mathcal{A}_3) \quad \Rightarrow$ line $h''(d)$

$$\begin{aligned} score(\mathcal{A}_1) &= score(\mathcal{A}_3) \\ (i) &= (v) \\ 3 - d - e &= 6 - 2\,d - 2\,e \\ \Leftrightarrow \quad e &= \boxed{-d + 3 := h''(d)} \end{aligned} \tag{vi}$$

Set $h(d) = h''(d) \quad \Rightarrow \quad$ intersection point r_2 on h with $score(\mathcal{A}_1) = score(\mathcal{A}_3)$

$$\begin{aligned} h(d) &= (vi) \\ \tfrac{2}{5}\,d + \tfrac{2}{5} &= -d + 3 \\ \Leftrightarrow \quad \tfrac{2+5}{5}\,d &= \tfrac{3\cdot 5}{5} - \tfrac{2}{5} \\ \Leftrightarrow \quad \tfrac{7}{5}\,d &= \tfrac{13}{5} \\ \Leftrightarrow \quad d &= \tfrac{13}{7} \quad \Rightarrow \quad e = -\tfrac{13}{7} + 3 = \tfrac{8}{7} \\ \Rightarrow r_2 &= \left(\frac{13}{7}, \frac{8}{7}\right) \end{aligned} \tag{vii}$$

Calculate optimum in $r_2 \;\Rightarrow\; \mathcal{A}_1, \mathcal{A}_3$ are optima in r_2



We found our point $r_2 = \left(\frac{13}{7}, \frac{8}{7}\right)$, farthest away from p, laying on h, at which $\mathcal{A}_1$ is still optimal. ■

G.2.4 Finding the Polygon Containing p

The next step is to find the polygon that contains a given point p in (d, e)-space. This is done by Algorithm G.3 on page 182; see also Figure G.4.

The two degenerate cases mentioned in step 3 of Algorithm G.3 are treated as follows.

1. r^* is on a border of the parameter space: In this case the polygon Poly(A) touches the border of the parameter space and the line L^* will be that border. The two ray searches in step 4 of Algorithm G.3 move along this border.

Algorithm G.3 Finding the polygon containing p.

1: Start at p with optimal alignment A, whose polygon Poly(A) is to be determined.
2: Solve the ray search problem for an arbitrary ray h starting at p, yielding r^* and a co-optimal alignment A^* in r^*
3: Assume this is no degenerate case (r^* is not on a border of the parameter space and r^* is not a vertex of the decomposition); otherwise see text.
4: The alignments A and A^* define a line L^* that contains the dividing edge between the two polygons Poly(A) and Poly(A^*). The extent of the edge can be found by solving Newton's ray search problem two more times from the intersection point of h and L^*, once in each direction of L^*.
5: The other edges of Poly(A) are found by more rays pointing from p in directions not intersecting with edges determined in earlier steps. Endpoints of edges have to be identified. When the circle is closed, the procedure stops.

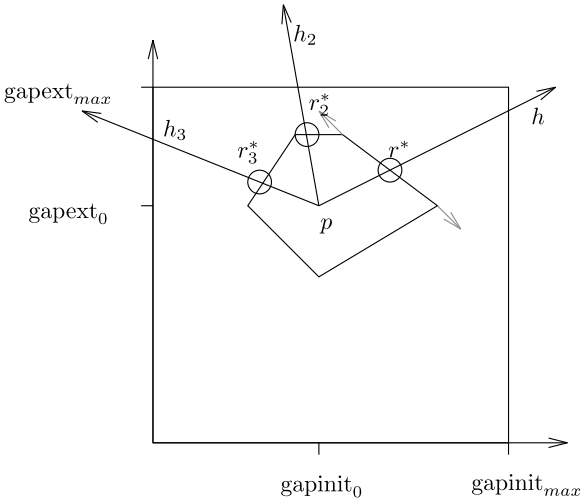


Figure G.4: Finding a whole polygon.

2. r^* is a vertex of the polygonal decomposition. This will be recognized by the fact that one or both of the ray searches in step 4 of the algorithm will not yield new points. One simply continues with a new ray search from p.

G.2.5 Filling the Parameter Space

Finally, the whole parameter space is filled with polygons as described in Algorithm G.4 on page 183.

G.2.6 Analysis and Improvements

The parametric alignment procedure described above is remarkably efficient, especially when a few little improvements are employed. (Details of the analysis can be found in (Gusfield, 1997), Section 13.1.)

Algorithm G.4 Filling the whole parameter space

1: Find an alignment A that is optimal in the interior of some (unknown) polygon (and not just on a point or line); this is easily possible with two ray searches.
2: Store in a list $\mathcal{L}$ all alignments that share polygon edges with A, where (by identification through the values ($n_{a,b}(A)$, $g(A)$ and $b(A)$) duplicates are identified and removed.
3: **while** there are unmarked elements in list $\mathcal{L}$ **do**
4: select an unmarked element A from $\mathcal{L}$
5: compute its polygon, possibly adding further alignments to $\mathcal{L}$
6: mark the element A

Let R be the number of polygons, E the number of edges, and V the number of vertices in the decomposition of the parameter space.

To find the edges of a polygon Poly(A) by Algorithm G.3, at most $6E_A$ ray searches are necessary, where E_A is the number of edges of Poly(A). (Up to $3E_A$ ray searches may be wasted by hitting vertices of the decomposition.) Since each of the E edges in the whole decomposition is shared by two polygons, the overall number of ray searches for all polygons is $12E$.

Further, note that each ray search requires at most R fixed-parameter alignment computations, and hence an upper bound for the overall time complexity is $O(ERmn)$, where m and n are, as usual, the sequence lengths.

This can be improved by applying two ideas:

1. Modify Newton's algorithm: Do not start with r^* at the border of the parameter space but at a closer point that can be determined from polygons computed in earlier steps.

2. Clever bookkeeping of already computed alignments, polygons, and edges.

This gives a time complexity of $O(R^2 + Rmn)$, which is $O(R + mn)$ per polygon.

Further considerations show that often the number of polygons R is of the order mn, so that the time to find each polygon is proportional to just the time for a single alignment computation.

For the (d, e) parameter space, this last assertion $R \in O(mn)$ can be seen as follows: Associate to alignment A the triple $(S(A), g(A), b(A))$ where, as before, $S(A)$ is the score of alignment A and $g(A)$ and $b(A)$ are the numbers of gaps and blanks in A, respectively. Now there exists another alignment A' with $g(A') = g(A)$ and $b(A') = b(A)$, but $S(A') > S(A)$ for all choices of (d, e). Then A can never be optimal at any point in (d, e)-space. Thus, among all triples with $g(A)$ and $b(A)$ as second and third components in their triple, only one is optimal at some point in the (d, e)-space. Moreover, there can be at most $m + n$ blanks and $\max\{m, n\} + 1$ gaps.

G.2.7 Parametric Alignment in Practice

Gusfield and Stelling (1996) developed a tool called XPARAL that implements the above mentioned parametric alignment approach. An example can be seen in Figure G.2 on

page 178. The program can be downloaded at `http://www.cs.ucdavis.edu/~gusfield/xparall/`.

APPENDIX H

Multiple Alignment in Practice: Mostly Progressive

Contents of this chapter: Progressive alignment, guide tree, aligning two alignments, segment-based alignment, segment identification, diagonal segments, selection and assembly, software proposition.

The **progressive alignment** method is a fast heuristic multiple alignment method that is motivated by the tree alignment approach. The most popular multiple alignment programs follow this strategy.

H.1 Progressive Alignment

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 H.1.1 we 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 in the previous chapter is the progressive alignment along the branches of an 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, from the leaves towards the root of the tree. Figure H.1 illustrates this procedure.

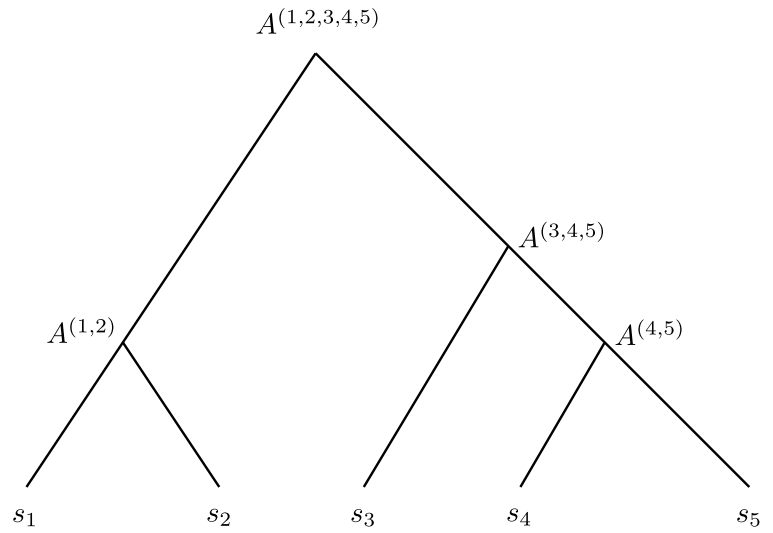


Figure H.1: Progressive alignment along the branches of a phylogenetic tree. Sequences s_1 and s_2 are aligned, giving alignment $A^{(1,2)}$. Sequences s_4 and s_5 give $A^{(4,5)}$ which then is aligned with s_3 giving $A^{(3,4,5)}$. Finally, aligning $A^{(1,2)}$ and $A^{(3,4,5)}$ gives the multiple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

Benefits of the progressive approach are:

- The method is more efficient than the sum-of-pairs multiple alignment. 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.
- 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 H.2. 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.

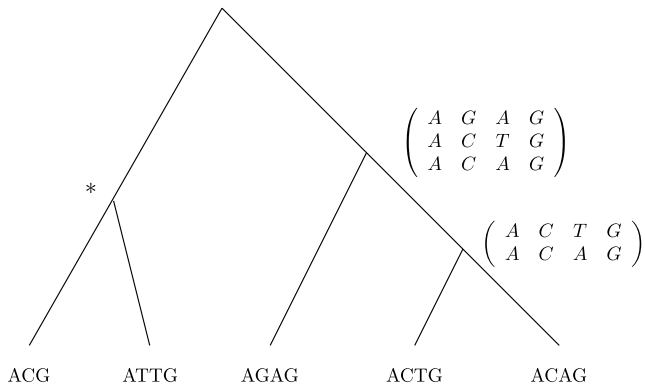


Figure H.2: In a strict bottom-up progressive computation, it cannot be decided at the time of computing the alignment denoted with the asterisk ($*$) if it should be $\begin{pmatrix} A & - & C & G \\ A & T & T & G \end{pmatrix}$ or $\begin{pmatrix} A & C & - & G \\ A & T & T & G \end{pmatrix}$. Only the rest of the tree indicates that the second variant is probably the correct one.

H.1.1 Aligning Two Alignments

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

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

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

In a unit cost scenario with matches of cost $cost(c, c) = 0$ and mismatches and indels of cost $cost(c, c') = 1$ for $c \neq c'$, $c, c' \in \Sigma \cup \{-\}$, an alignment of these two columns would be scored $cost(A, A) + cost(A, G) + cost(G, A) + cost(G, G) + cost(T, A) + cost(T, G) + cost(-, A) + cost(-, G) = 0 + 1 + 1 + 0 + 1 + 1 + 1 + 1 = 6$. 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

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

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

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

using the unit cost model described above.

	$\begin{matrix} - \\ - \end{matrix}$	$\begin{matrix} A \\ A \end{matrix}$	$\begin{matrix} T \\ G \end{matrix}$	$\begin{matrix} C \\ C \end{matrix}$	$\begin{matrix} A \\ - \end{matrix}$	$\begin{matrix} G \\ G \end{matrix}$
$\begin{matrix} - \\ - \end{matrix}$	0	4	8	12	14	18
$\begin{matrix} T \\ G \end{matrix}$	4	4	6	10	12	16
$\begin{matrix} A \\ - \end{matrix}$	6	6	8	...		
$\begin{matrix} G \\ C \end{matrix}$	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 $O(n_A n_B)$ if the alphabet $\Sigma = (c_1, c_2, \ldots, c_E)$ is of constant size E and an alignment column A_i is stored as a vector $(a_{c_1}, a_{c_2}, \ldots, a_{c_E}, a_-)$ where a_c is the number of occurrences of character c in A_i, such that the cost of aligning two columns $A_i = (a_{c_1}, a_{c_2}, \ldots, a_{c_E}, a_-)$ and $B_j = (b_{c_1}, b_{c_2}, \ldots, b_{c_E}, b_-)$ can be written as

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

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.

H.2 Segment-Based Alignment

The global multiple alignment methods mentioned so far, like the progressive methods or the simultaneous method DCA, have properties that are not necessarily desirable in all applications.

In particular, the alignment depends on many parameters, like the cost function or substitution matrix, gap penalties, and a scoring function (e.g. sum of pairs) for the multiple alignment. Apart from the problem that these parameters need to be chosen in advance, the global methods have the well-known weakness that local similarities may be aligned in a wrong way if they are not significant in a global context.

For example, the following alignment shows a potential misalignment that may be caused by the choice of the affine gap penalties:

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

If instead of individual positions, whole units of local similarities are compared and used to build a global multiple alignment, the result will depend much less on the particular alignment parameters, and instead reflect more the local similarities in the data, like in the following example:

$$A' = \begin{pmatrix} A & F & A & T & C & A & T & C & A \\ A & - & - & - & C & A & T & - & A \end{pmatrix}.$$

This is the idea of segment-based sequence alignment.

H.2.1 Segment Identification

The first step in a segment-based alignment approach is to identify the segments. There are different possibilities to obtain segments.

The possibly simplest strategy, followed by the program TWOALIGN Abdeddaïm (1997) is to use *local alignments* like they are computed by the Smith-Waterman algorithm. A disadvantage of this approach is that the local alignment computation also requires the usual alignment parameters like score function and gap penalties.

An alternative approach is to use gap-free local alignments. In this case, the segments correspond to *diagonal segments* of the edit matrix. Since gap-free alignments can be computed faster than gapped alignments, more time can be spent on the computation of their score. The approach followed by Morgenstern *et al.* (Morgenstern et al., 1996; Morgenstern, 1999) in the DIALIGN method is to use a statistical objective function that assigns the score $P_D(l_D, s_D)$ to a diagonal D of length l_D with s_D matches.

This score is the probability that a random segment of length l_D has at least s_D matches:

$$P_D(l_D, s_D) = \sum_{i=s_D}^{l_D} \binom{l_D}{i} p^i \, (1-p)^{l_D - i},$$

where $p = 0.25$ for nucleotides and $p = 0.05$ for amino acids. The *weight* w_D of a diagonal D is then defined as the negative logarithm of its score,

$$w_D = -\ln(P_D).$$

(In more statistical terms, the score would be called the p-value of the diagonal, and the weight would be called the score.)

The time to compute all diagonals is $O(n^3)$, but this can be reduced to $O(n^2)$ if the maximal length of a diagonal segment is bounded by a constant.

H.2.2 Segment Selection and Assembly

After a set of segments is identified, it is not necessarily possible to assemble them into a global alignment, since they might be *inconsistent*, i.e. the inclusion of some of the segments does not allow the inclusion of other segments into the alignment. In the case of two sequences, this is the case if and only if two segments "cross" as the two segments "ON" and "ANA" shown in Figure H.3.

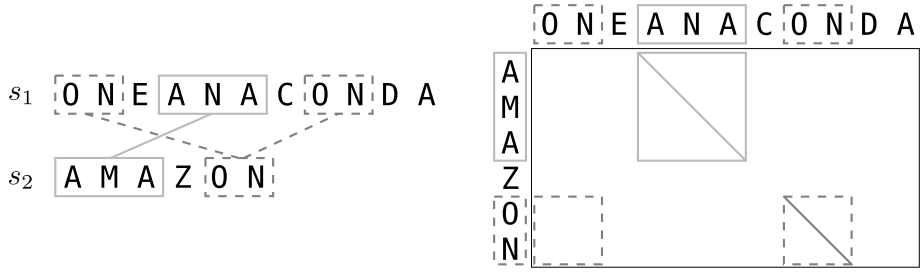


Figure H.3: Inconsistent pairwise alignment segments.

More generally, a set of segments of multiple sequences $s_1, s_2, \ldots, s_k$ is **inconsistent** if there exists no multiple alignment of these sequences that induces all these segments. Several methods exist to select from a set of segments a subset of consistent segments.

For example, the methods TWOALIGN and DIALIGN employ a greedy algorithm that considers one maximum scoring pairwise local alignment after the other (in order of decreasing alignment weight) and, if it is consistent with the previously assembled segments, adds it to the solution.

Given a set of consistent segments, in general there will still remain unaligned characters that are not contained in any of the consistent segments. In order to arrive at a global multiple alignment, it is desirable to also add these characters to the alignment. Different strategies can be followed.

The DIALIGN method simply fills the remaining regions with gaps, letting those characters unaligned that are not contained in any segment. Another possibility is to re-align the regions between the segments. This is not so easy, though, because the region "between the segments" is not clearly defined.

In any case, a global alignment is to be computed that respects the segments as fixed anchors, while for the remaining characters freedom is still given, as long as the resulting alignment is consistent with the segments. It has been suggested to align the sequences progressively (Myers, 1996) or simultaneously (Sammeth et al., 2003a).

H.3 Software for Progressive and Segment-Based Alignment

H.3.1 The Program Clustal W

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 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 W 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.

H.3.2 T-COFFEE

Another method to compute high-quality multiple alignments that is becoming more and more popular 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.

H.3.3 DIALIGN

DIALIGN is a practical implementation of segment-based alignment, enhanced in several ways.

By taking into account the information from various sequences during the selection of consistent segments, DIALIGN uses an optimized objective function in order to enhance the score of diagonals with a weak, but consistent signal over many sequences. The **overlap weight** of a diagonal D is defined as

$$olw(D) = w_D + \sum_{E \in \mathcal{D}} \tilde{w}(D, E),$$

where $\mathcal{D}$ is the set of all diagonals and $\tilde{w}(D_1, D_2) := w_{D_3}$ is the weight of the (implied) overlap D_3 of the two diagonals D_1 and D_2.

For coding DNA sequences, it is possible that the program automatically translates the codons into amino acids and then scores the diagonal with the amino acid scoring scheme.

In its second version, DIALIGN 2.0, the statistical score for diagonals was changed in order to upweight longer diagonals.

Also, the algorithm for consistency checking and greedy addition of diagonals into the growing multiple alignment has been improved several times.

H.3.4 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 H.1.1 on page 187 (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

We refer to the first two stages alone as MUSCLE-p, which produces MSA_2. MUSCLE-p has time complexity $O(N2L + NL2)$ and space complexity $O(N2 + NL + L2)$. Refinement adds an $O(N3L)$ term to the time complexity.

H.3.5 QAlign

A comfortable system that includes several alignment algorithms and a flexible alignment editor is QAlign (Sammeth et al., 2003b), respectively its latest version, QAlign2 (*panta rhei*). It can be found at `http://gi.cebitec.uni-bielefeld.de/QAlign`.

Bibliography

S. Abdeddaïm. On incremental computation of transitive closure and greedy alignment. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching, CPM 1997*, volume 1264 of *LNCS*, pages 167–179, 1997.

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). *Journal of Molecular Biology*, 215:403–410, 1990.

S. F. Altschul. Gap costs for multiple sequence alignment. *J. Theor. Biol.*, 138:297–309, 1989.

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.

A. N. Arslan, O. Egecioglu, and P. A. Pevzner. A new approach to sequence comparison: normalized sequence alignment. *Bioinformatics*, 17(4):327–337, 2001.

V. Bafna, E. L. Lawler, and P. A. Pevzner. Approximation algorithms for multiple sequence alignment. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching, CPM 1994*, volume 807 of *LNCS*, pages 43–53, 1994.

S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram based database searching using a suffix array (QUASAR). In *Proc. of the Third Annual International Conference on Computational Molecular Biology, RECOMB 1999*, pages 77–83, 1999.

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

P. Chain, S. Kurtz, E. Ohlebusch, and T. Slezak. An applications-focused review of comparative genomics tools: Capabilities, limitations and future challenges. *Briefings in Bioinformatics*, 4(2):105–123, 2003.

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, Aug 2004a.

R. C. Edgar. Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Res*, 32(5):1792–1797, 2004b.

D.-F. Feng and R. F. Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *J. Mol. Evol.*, 25:351–360, 1987.

W. M. Fitch. Toward defining the course of evolution: Minimum change for a specific tree topology. *Syst. Zool.*, 20(4):406–416, 1971.

Y. Frid and D. Gusfield. A simple, practical and complete $o(\frac{n^3}{\log n})$-time algorithm for RNA folding using the *Four-Russians* speedup. In *Proceedings of WABI 2009*, volume 5724 of *LNBI*, pages 97–107, 2009.

O. Gascuel, editor. *Mathematics of Evolution and Phylogeny.* Oxford University Press, 2005.

O. Gotoh. An improved algorithm for matching biological sequences. *J. Mol. Biol.*, 162: 705–708, 1982.

A. K. Gupta. On a "square" functional equation. *Pacific Journal of Mathematics*, 50(2): 449–454, 1974.

S. K. Gupta, J. D. Kececioglu, and A. A. Schäffer. Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment. *J. Comp. Biol.*, 2(3):459–472, 1995.

D. Gusfield. Efficient algorithms for inferring evolutionary trees. *Networks*, 21:19–28, 1991.

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.

D. Gusfield and P. Stelling. Parametric and inverse-parametric sequence alignment with xparal. *Methods Enzymol*, 266:481–494, 1996.

D. G. Higgins and P. M. Sharp. Clustal V: Improved software for multiple sequence alignment. *CABIOS*, 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, Feb 1995.

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

M. Höhl, S. Kurtz, and E. Ohlebusch. Efficient multiple genome alignment. *Bioinformatics*, 18(Suppl. 1):312–320, 2002. (Proceedings of ISMB 2002).

X. Huang and W. Miller. A time-efficient, linear-space local similarity algorithm. *Adv. Appl. Math.*, 12:337–357, 1991.

T. J. P. Hubbard, A. M. Lesk, and A. Tramontano. Gathering them into the fold. *Nature Structural Biology*, 4:313, 1996.

T. Jiang, E. L. Lawler, and L. Wang. Aligning sequences via an evolutionary tree: Complexity and approxiamtion. In *Conf. Proc. 26th Annu. ACM Symp. Theory Comput., STOC 1994*, pages 760–769, 1994.

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.

W. J. Kent. Blat–the blast-like alignment tool. *Genome Res*, 12(4):656–664, Apr 2002.

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.

J. Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theor. Comput. Sci.*, 387(3):249–257, 2007.

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.

H. T. Laquer. Asmyptotic limits for a two-dimensional recursion. *Stud. Appl. Math.*, 64: 271–277, 1981.

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.

U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the first annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327. SIAM, January 1990.

E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2): 262–272, 1976.

B. Morgenstern. DIALIGN 2: Improvement of the segment-to-segment approach to multiple sequence alignment. *Bioinformatics*, 15(3):211–218, 1999.

B. Morgenstern, A. W. M. Dress, and T. Werner. Multiple DNA and protein sequence alignment based on segment-to-segment comparison. *Proc. Natl. Acad. Sci. USA*, 93(22): 12098–12103, 1996.

D. W. Mount. *Bioinformatics: Sequence and Genome Analysis.* Cold Spring Harbor Laboratory Press, 2nd edition, 2004.

T. Müller, S. Rahmann, and M. Rehmsmeier. Non-symmetric score matrices and the detection of homologous transmembrane proteins. *Bioinformatics*, 17(Suppl 1):182–189, 2001.

E. W. Myers. Approximate matching of network expressions with spacers. *J. Comp. Biol.*, 3(1):33–51, 1996.

E. W. Myers and W. Miller. Optimal alignments in linear space. *Comput Appl Biosci*, 4(1): 11–17, Mar 1988.

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.

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.

R. Nussinov and A. B. Jacobson. Fast algorithm for predicting the secondary structure of single-stranded RNA. *Proc Natl Acad Sci USA*, 77:6903–6913, 1980.

K. Rasmussen, J. Stoye, and E. W. Myers. Efficient q-gram filters for finding all ϵ-matches over a given length. *J. Comp. Biol.*, 13(2):296–308, 2006.

N. Saitou and M. Nei. The neighbor-joining method: A new method for reconstructing phylogenetic trees. *Mol. Biol. Evol.*, 4(4):406–425, 1987.

M. Sammeth, B. Morgenstern, and J. Stoye. Divide-and-conquer multiple alignment with segment-based constraints. *Bioinformatics*, 19(Suppl. 2):ii189–ii195, 2003a. (Proceedings of ECCB 2003).

M. Sammeth, J. Rothgänger, W. Esser, J. Albert, J. Stoye, and D. Harmsen. QAlign: Quality-based multiple alignments with dynamic phylogenetic analysis. *Bioinformatics*, 19(12):1592–1593, 2003b.

D. Sankoff. Minimal mutation trees of sequences. *SIAM J. Appl. Math.*, 28(1):35–42, 1975.

B. Schwikowski and M. Vingron. The deferred path heuristic for the generalized tree alignment problem. *J. Comp. Biol.*, 4(3):415–431, 1997.

P. H. Sellers. Theory and computation of evolutionary distances: Pattern recognition. *J. Algorithms*, 1:359–373, 1980.

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, Mar 1981.

J. Stoye. Multiple sequence alignment with the divide-and-conquer method. *Gene*, 211(2): GC45–GC56, 1998.

W. R. Taylor and D. T. Jones. Deriving an amino acid distance matrix. *J Theor Biol*, 164 (1):65–83, Sep 1993.

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. Ukkonnen. Finding approximate patterns in strings. *J. Algorithms*, 6:132–137, 1985.

E. Ukkonnen. On-line construction of suffix trees. *Algorithmica*, 14:249–60, 1995.

M. Vingron and A. von Haeseler. Towards integration of multiple alignment and phylogenetic tree construction. *J. Comp. Biol.*, 4(1):23–34, 1997.

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

L. Wang, T. Jiang, and E. L. Lawler. Approximation algorithms for tree alignment with a given phylogeny. *Algorithmica*, 16:302–315, 1996.

M. S. Waterman and M. Eggert. A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons. *J. Mol. Biol.*, 197:723–728, 1987.

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.