

# SeqAn

## A Generic Software Library for Sequence Analysis

(SeqAn – eine generische Softwarebibliothek zur Sequenzanalyse)

Dissertation zur Erlangung des Grades  
eines Doktors der Naturwissenschaften

vorgelegt am

Fachbereich Mathematik und Informatik  
der Freien Universität Berlin

von

Andreas Gogol-Döring

2009

Betreuer: Prof. Dr. Knut Reinert  
Institut für Informatik  
Freie Universität Berlin

Gutachter:	Prof. Dr. Knut Reinert Institut für Informatik Freie Universität Berlin	Prof. Dr. Daniel Huson Wilhelm-Schickard-Institut für Informatik Universität Tübingen
------------	---	---

Termin der Disputation: 12. November 2009

für Fabienne



## Danksagung

Wenn nach jahrelanger Arbeit endlich einmal “alles gut” wird, so hat man das Vielen zu verdanken. SeqAn ist das Werk eines Teams, und darum möchte ich mich an dieser Stelle bei allen bedanken, die an der Entwicklung dieser Bibliothek beteiligt waren, allen voran meinem Doktorvater Knut Reinert. SeqAn war seine Idee, und nur durch seine motivierende Betreuung konnte aus dem Projekt überhaupt etwas werden.

Mein besonderer Dank gilt meinen Kollegen David Weese und Tobias Rausch, ohne deren ausgezeichnete Arbeit sich das Projekt niemals so vortrefflich entwickelt hätte. Bei euch wird SeqAn auch zukünftig in guten Händen sein.

Nicht vergessen werden dürfen auch jene unzähligen Studenten, die in Übungen, Praktika und Abschlussarbeiten einen wertvollen Teil zu SeqAn beigesteuert haben. Gerade in der Anfangszeit hatten sie es schwer, auf der Grundlage eines noch völlig unausgereiften Basisdesigns überhaupt Ergebnisse zu produzieren. Nur einige von ihnen werden in dieser Arbeit namentlich erwähnt, doch ihnen allen gebührt mein herzlicher Dank.

Zuletzt möchte ich noch das freundliche Umfeld an der Freien Universität Berlin dankend erwähnen, insbesondere die Kollegen aus der AGABI und der AGSE. Sechs Jahre sind wie im Flug vergangen.



## Zusammenfassung

SeqAn ist eine Programmbibliothek effizienter Algorithmen und Datenstrukturen zur Sequenzanalyse, d.h. zur Verarbeitung großer Mengen biomedizinischer Daten, insbesondere von Gen- und Proteinsequenzen. Die Entwicklung dieser Bibliothek zielt auf zwei Gruppen von Anwendern ab: Zum einen soll sie Programmierern bei der Entwicklung neuer Softwarewerkzeuge helfen. Derartige Softwarewerkzeuge sind unabdingbar für die biologische und medizinische Forschung. Zum anderen sollen Algorithmen designer die Bibliothek als eine Grundlage für Entwicklung, Test und Vergleich von Algorithmen verwenden können. Das Projekt versucht also, einen ingenieurwissenschaftlichen Beitrag zur Bioinformatik zu leisten, und will damit letztlich der naturwissenschaftlichen Forschung im Bereich der Lebenswissenschaften dienen.

Eine ausführlichere Zusammenfassung des SeqAn-Projekts und des Inhalts dieser Arbeit findet sich in Kapitel 3. Welchen Beitrag der Autor bei der Entwicklung der Bibliothek geleistet hat, wird in Abschnitt 3.3 erklärt.

## Abstract

SeqAn is a library of efficient algorithms and data structures for sequence analysis, which means processing large amounts of biomedical data like DNA or protein sequences. The library was developed for two groups of users: Software engineers can use it for the implementation of new software tools. Such tools are essential for biological and medical research. Algorithm designers may also use the library as a platform for the development, testing and comparison of algorithms. The project therefore contributes to bioinformatics engineering with the eventual purpose to promote the scientific research in life science.

A more detailed abstract of the SeqAn project and the contents of this thesis is provided in Chapter 3. The author's contribution to the development of the library is given in Section 3.3.





# Contents

<b>I</b>	<b>Background</b>	<b>1</b>
<b>1</b>	<b>Sequence Analysis</b>	<b>3</b>
1.1	Sequences in Bioinformatics . . . . .	3
1.2	Sequence Analysis Examples . . . . .	5
1.2.1	Sequence Assembly . . . . .	5
1.2.2	BLAST: Finding Similar Regions . . . . .	6
1.2.3	CLUSTAL W: Aligning Multiple Sequences . . . . .	7
<b>2</b>	<b>Software Libraries</b>	<b>9</b>
2.1	Software Libraries in General . . . . .	9
2.1.1	Benefits from Software Libraries . . . . .	10
2.1.2	Software Library Examples . . . . .	10
2.2	Bioinformatics Libraries . . . . .	11
2.2.1	Libraries for Using Existing Tools . . . . .	11
2.2.2	Libraries for Development of New Tools . . . . .	12
2.2.3	Conclusion . . . . .	14
<b>3</b>	<b>SeqAn</b>	<b>15</b>
3.1	Design of SeqAn . . . . .	16
3.2	Contents of SeqAn . . . . .	16
3.3	This Thesis . . . . .	16
<b>II</b>	<b>Design</b>	<b>19</b>
<b>4</b>	<b>Design Considerations</b>	<b>21</b>
4.1	Design Overview . . . . .	21
4.2	Design Goals . . . . .	22
4.2.1	Performance . . . . .	22
4.2.2	Simplicity . . . . .	23
4.2.3	Generality . . . . .	23
4.2.4	Refineability . . . . .	24
4.2.5	Extensibility . . . . .	24
4.2.6	Integration . . . . .	25

<b>5</b>	<b>Programming Techniques</b>	<b>27</b>
5.1	The C++ Programming Language . . . . .	27
5.2	Generic Programming . . . . .	28
5.3	Template Subclassing . . . . .	29
5.3.1	Template Subclassing Technique . . . . .	31
5.3.2	Comparison to Object Oriented Programming . . . . .	31
5.4	Global Function Interfaces . . . . .	33
5.4.1	Advantages of Global Interfaces . . . . .	33
5.4.2	Discussion . . . . .	34
5.5	Metafunctions . . . . .	36
5.6	Further Techniques . . . . .	38
5.6.1	Metaprogramming . . . . .	38
5.6.2	Tag Dispatching . . . . .	38
5.6.3	Defaults and Shortcuts . . . . .	39
<b>6</b>	<b>The Design In Examples</b>	<b>41</b>
6.1	Example 1: Value Counting . . . . .	41
6.1.1	The Metafunction <code>Value</code> . . . . .	42
6.1.2	The Metafunction <code>ValueSize</code> . . . . .	43
6.1.3	The Functions <code>length</code> . . . . .	43
6.1.4	The Functions <code>value</code> . . . . .	44
6.1.5	The Generic Algorithm <code>countValues</code> . . . . .	45
6.2	Example 2: Locality-Sensitive Hashing . . . . .	45
6.2.1	The Base Class <code>Shape</code> . . . . .	47
6.2.2	Generic Gapped Shapes . . . . .	47
6.2.3	Ungapped Shapes . . . . .	48
6.2.4	Hardwired Shapes . . . . .	49
6.2.5	Conclusion . . . . .	50
<b>III</b>	<b>Content</b>	<b>53</b>
<b>7</b>	<b>Basics</b>	<b>55</b>
7.1	Containers and Values . . . . .	55
7.2	Memory Allocation . . . . .	56
7.3	Move Operations . . . . .	58
7.4	Alphabets . . . . .	59
7.4.1	Simple Types . . . . .	59
7.4.2	Array Operations . . . . .	60
7.4.3	Alphabet Modifiers . . . . .	60
7.5	Iterators . . . . .	61
7.6	Conversions . . . . .	62
7.6.1	Sequence Conversions . . . . .	63

7.7	File Input/Output . . . . .	63
<b>8</b>	<b>Sequences</b>	<b>65</b>
8.1	Strings . . . . .	65
8.2	Overflow Strategies . . . . .	66
8.3	String Specializations . . . . .	67
8.3.1	Alloc Strings . . . . .	67
8.3.2	Array Strings . . . . .	69
8.3.3	Block Strings . . . . .	69
8.3.4	Packed Strings . . . . .	69
8.3.5	External Strings . . . . .	71
8.4	Sequence Adaptors . . . . .	72
8.5	Sequence Modifiers . . . . .	73
8.6	Segments . . . . .	74
8.7	Comparators . . . . .	75
8.8	String Sets . . . . .	77
<b>9</b>	<b>Alignments</b>	<b>79</b>
9.1	Gaps Data Structures . . . . .	79
9.1.1	SequenceGaps Specialization . . . . .	81
9.1.2	ArrayGaps Specialization . . . . .	81
9.1.3	SumlistGaps Specialization . . . . .	82
9.2	Alignment Data Structures . . . . .	83
9.3	Alignment Scoring . . . . .	84
9.3.1	Scoring Schemes . . . . .	84
9.3.2	Sequence Similarity and Sequence Distance . . . . .	85
9.4	Alignment Problems Overview . . . . .	86
9.5	Global Alignments . . . . .	86
9.5.1	Needleman-Wunsch Algorithm . . . . .	87
9.5.2	Gotoh's Algorithm . . . . .	88
9.5.3	Hirschberg's Algorithm . . . . .	90
9.5.4	Aligning with Free Start or End Gaps . . . . .	92
9.5.5	Progressive Alignment . . . . .	93
9.6	Chaining . . . . .	94
9.6.1	Seeds . . . . .	95
9.6.2	Generic Chaining . . . . .	95
9.6.3	Chaining Using Sparse Dynamic Programming . . . . .	97
9.6.4	Banded Alignment . . . . .	100
<b>10</b>	<b>Pattern Matching</b>	<b>101</b>
10.1	Exact Searching . . . . .	103
10.1.1	Brute-Force Exact Searching . . . . .	103
10.1.2	Horspool's Algorithm . . . . .	105

10.1.3	Shift-Or Algorithm . . . . .	105
10.1.4	Backward Factor Automaton Matching . . . . .	107
10.1.5	Backward Nondeterministic DAWG Matching . . . . .	108
10.1.6	Results . . . . .	109
10.2	Exact Searching of Multiple Needles . . . . .	111
10.2.1	Wu-Manber Algorithm . . . . .	111
10.2.2	Multiple BFAM Algorithm . . . . .	112
10.3	Approximate Searching . . . . .	115
10.3.1	Sellers' Algorithm . . . . .	117
10.3.2	Myers' Bitvector Algorithm . . . . .	118
10.3.3	Partition Filtering . . . . .	119
10.4	Other Pattern Matching Problems . . . . .	121
10.4.1	$k$ -Mismatch Searching . . . . .	121
10.4.2	Searching with Wildcards . . . . .	121
<b>11</b>	<b>Motif Discovery</b>	<b>125</b>
11.1	Local Alignments . . . . .	126
11.1.1	Smith-Waterman Algorithm . . . . .	126
11.1.2	Waterman-Eggert Algorithm . . . . .	128
11.2	Seed Based Motif Search . . . . .	128
11.2.1	Extending Seeds . . . . .	129
11.2.2	Combining Seeds . . . . .	132
11.3	Multiple Sequence Motifs . . . . .	135
11.3.1	The Randomized Heuristic <b>Projection</b> . . . . .	137
11.3.2	The Enumerating Algorithm <b>PMSP</b> . . . . .	139
<b>12</b>	<b>Indices</b>	<b>141</b>
12.1	$q$ -Gram Indices . . . . .	142
12.1.1	Shapes . . . . .	142
12.1.2	$q$ -Gram Index Construction . . . . .	143
12.2	Suffix Arrays . . . . .	145
12.2.1	Suffix Array Construction . . . . .	145
12.2.2	Searching in Suffix Arrays . . . . .	147
12.3	Enhanced Suffix Arrays . . . . .	150
12.3.1	LCP Table . . . . .	151
12.3.2	Suffix Trees Iterators . . . . .	151
<b>13</b>	<b>Graphs</b>	<b>157</b>
13.1	Automata . . . . .	160
13.1.1	Tries . . . . .	161
13.1.2	Factor Oracles . . . . .	162
13.2	Alignment Graphs . . . . .	163
13.2.1	Alignment Graph Data Structure . . . . .	163

13.2.2	Maximum Weight Trace Problem . . . . .	165
13.2.3	Segment Match Refinement . . . . .	167
<b>IV</b>	<b>Discussion</b>	<b>169</b>
<b>14</b>	<b>Library Quality and Applicability</b>	<b>171</b>
14.1	Design Quality . . . . .	171
14.1.1	Performance . . . . .	171
14.1.2	Simplicity . . . . .	172
14.1.3	Generality . . . . .	173
14.1.4	Refinebility . . . . .	173
14.1.5	Extensibility . . . . .	173
14.1.6	Integration . . . . .	173
14.2	Stability, Usability, Accessibility . . . . .	174
14.2.1	Testing . . . . .	174
14.2.2	Documentation . . . . .	176
14.2.3	Distribution . . . . .	177
<b>15</b>	<b>Example Application LAGAN</b>	<b>179</b>
15.1	The LAGAN Algorithm . . . . .	179
15.2	Implementation of LAGAN . . . . .	181
15.3	Results . . . . .	184
<b>16</b>	<b>Conclusion</b>	<b>187</b>
<b>A</b>	<b>Appendix</b>	<b>189</b>
A.1	Proof to Myers' Bitvector Algorithm . . . . .	189
A.2	Sum Lists . . . . .	192
A.3	LAGAN Sources . . . . .	194
	<b>Bibliography</b>	<b>197</b>
	<b>Index</b>	<b>209</b>



# Part I

## Background

*In this part, we will first discuss in Chapter 1 the role of sequence analysis in the life sciences. Chapter 2 explains how software libraries could facilitate the development of new software tools and algorithms for sequence analysis. A review of related work in Section 2.2 reveals that SeqAn is the only software library available that focus explicitly on the development of highly performant sequence analysis software by providing a comprehensive collection of the common algorithmic components and data structures. Chapter 3 gives a short overview of the SeqAn project.*





# Chapter 1

## Sequence Analysis

### 1.1 Sequences in Bioinformatics

Sequences play a major role in biology as a means of abstraction. For example *deoxyribonucleic acid* (DNA), the carrier of genetic information in the nucleus, as well as *proteins*, a main ingredient of the cell responsible for most biological activity, can be represented as sequences over an alphabet of four, respectively twenty characters. This is due to the fact that those molecules are *biopolymers*, large organic molecules assembled from small building blocks called ‘*monomers*’, which are all of the same kind and linked together to long chains. The monomers of nucleic acids like DNA or RNA (*ribonucleic acid*) are *nucleotides*, and each nucleotide contains one out of four possible *nucleobases*. The structure of a nucleic acid strand is therefore defined by the actual sequence of bases in its nucleotides. Proteins on the other hand are composed of *amino acids*. In natural proteins, twenty different kinds of amino acid occur. They all have a phosphate backbone and differ in their *residues*. In proteins, these amino acids may occur in any order and number. We call the information about the succession of the monomers in a nucleic acid and protein its *biological sequence*, and thus we consider these biopolymers a kind of ‘storage’ for this information. Many functions which are fulfilled by biopolymers like nucleic acids and proteins depend on their sequence composition. A DNA sequence for example encodes *genes*, which are construction plans for proteins. The cell first *transcribes* the genes into messenger RNA (*mRNA*), which is then, after some modifications, *translated* into a peptide, where every three nucleobases form a ‘*codon*’ that corresponds to one specific amino acid in the synthesized protein. The sequence of nucleotides in the DNA therefore defines the order of amino acids in the protein, which further specifies the three-dimensional shape the protein folds into. RNA may also fold into a structure that is crucial to fulfill its purposes in the cell. Moreover the degree of molecular binding between proteins and nucleic acids depends on their sequences; the protein synthesis for example involves certain proteins that can ‘dock’ only on specific patterns

in the DNA.

A deeper understanding of biological processes thus requires a broad knowledge of the biomolecule sequences, and in the last decades a lot of research was aimed to decode those sequences. The most prominent projects in this field were certainly the *Human Genome Project* (International Human Genome Sequencing Consortium 2001) and its counterpart by Celera Genomics (Venter et al. 2001) which both aimed to sequence the entire human genome. Decrypted biological sequences are deposited in public databases as *strings*, i.e. ordered sequences of characters from a finite alphabet  $\Sigma$ . The succession of bases in a DNA can for example be stored in a string of the alphabet  $\Sigma = \{A, C, G, T\}$ , where each letter stands for one nucleic base, e.g. ‘A’ for ‘adenine’. Figure 1 shows the rapid progression of the data volume deposited

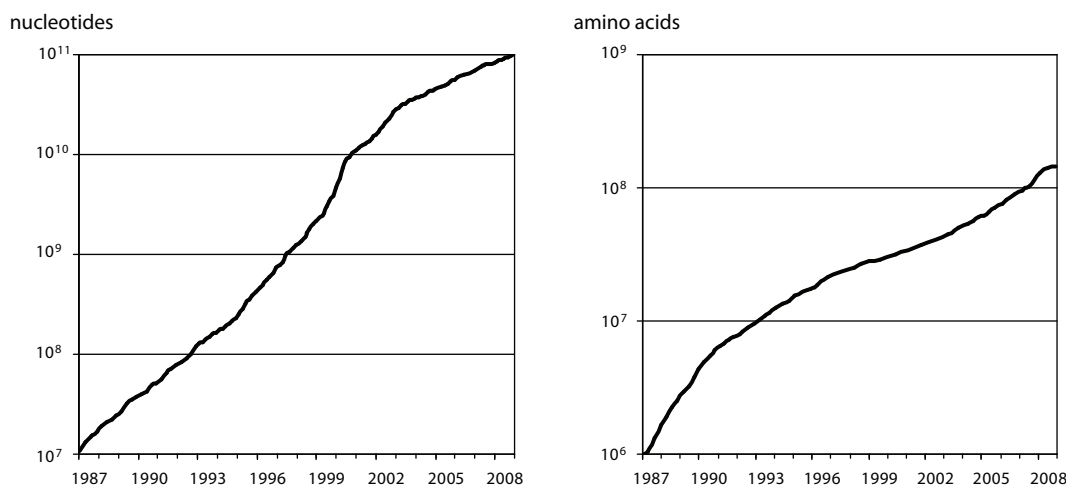


Figure 1: **Sequence Database Growth.** **Left:** Total number of bases stored in GenBank from the beginning of the year 1987 to the end of 2008. **Right:** Total number of million residues that were stored in UniProtKB/Swiss-Prot database during the same period. The data were taken from the release notes. Note the logarithmical scale.

in GenBank (Benson et al. 2008) and UniProtKB/Swiss-Prot (UniProt Consortium 2008). The number of nucleotides stored in GenBank has doubled approximately every 20 months and thus risen in two decades by four orders of magnitude. The protein database Swiss-Prot grew somewhat slower: From the beginning of the nineties, the amount of amino acids increased about 20% per year, i.e. it doubles every four years.

Lately, several new sequencing technologies like ‘pyrosequencing’ (also known as Roche/454 sequencing; Margulies et al. 2005) or ‘sequencing-by-synthesis’ (also known as Illumina/Solexa technology; Bentley 2006) were invented, and they allow a much higher throughput than previous approaches. Hence, the size of the databases is expected to grow even faster in the future, since the availability and decreasing cost of sequencing open the door to new applications in metagenomics or personalized medicine. The analysis of these data may help

to explain processes in the cell, like the regulation of gene expression, and this understanding offers the opportunity to develop new treatments of diseases or improved agricultural crops, and it will give us deeper insights into evolution.

## 1.2 Sequence Analysis Examples

*Sequence analysis* is the processing of biological sequences by means of bioinformatics algorithms and data structures. The typical objective of sequence analysis is to answer questions from biological or medical research. In this section, we will present several examples of sequence analysis tasks as well as some common tools for sequence analysis.

### 1.2.1 Sequence Assembly

All known methods for determining biological sequences are only capable to directly decrypt sequences of limited lengths. We need sequence analysis algorithms to produce longer sequences. For example, the first sequencing of the human genome (Venter et al. 2001; International Human Genome Sequencing Consortium 2001) was based on the chain-termination sequencing technique (Sanger et al. 1977) which is only capable to produce sequence reads shorter than a thousand nucleotides. For longer sequences a procedure called ‘*shotgun sequencing*’ proved to be viable, and it is able to determine even the sequences of whole eucaryotic genomes (‘whole-genome shotgun sequencing’, Staden 1997; see also Istrail et al. 2004). This method randomly samples and sequences fragments from the DNA such that on average any part is covered several times. The resulting sequence *reads* are then *assembled* by means of sequence analysis methods to get the complete sequence. This is not trivial, because errors occur during the sequencing of the reads, it is not clear which strand of the double-helix the read stems from, and because genomes are highly repetitive.

A first step in sequence assembly is usually to compute *overlap alignments* between the reads, for example by a *dynamic programming* alignment algorithm (see Section 9.5.4 on page 92). If two reads significantly overlap, then they putatively originate from the same location. For large numbers of reads, the computation of all needed overlap alignments could be accelerated by applying filtering. For example one could limit the search for overlapping candidates to those pairs of reads that share at least a given number of common *q*-grams (see Section 12.1.1 on page 142). The question is then how to conclude from the overlapping reads to the putative complete sequence. Usually several processing steps are necessary for computing a final consensus sequence (see e.g. Huson et al. 2001). Sequence assembly is especially hard for DNA that contains long repeats, since all reads that stem from repetitive regions cannot be

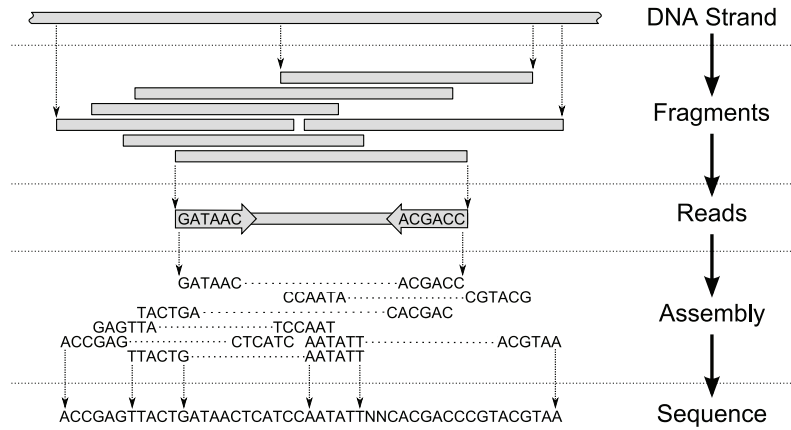


Figure 2: **Double-Barrel Shotgun Sequencing.** Fragments of a DNA strand are sequenced from both sides. The sequence assembly problem is to reconstruct the sequence of the DNA from the sequences of the reads.

definitely assigned to a single position. In this situation it could be helpful to apply ‘double-barrel’ shotgun sequencing, that is to sequence both ends of fragments that have a fixed length of several thousand nucleotides, see Figure 2. From that we get pairs of sequence reads with a certain distance in between, so if one of these reads falls into a repeat region, it could still be possible to determine its real position relative to its mate read. There are several de-novo sequence assemblers available; some of the more recent tools can even handle the rather short reads that are generated by next-generation sequencing technologies (e.g. Dohm et al. 2007; Zerbino and Birney 2008).

### 1.2.2 BLAST: Finding Similar Regions

Searching is probably the most basic operation in sequence analysis, and a program called ‘*Basic Local Alignment Search Tool*’ (BLAST, Altschul et al. 1990) has become the most widely used sequence analysis tool in bioscience. Comparable tools like for example FASTA (Pearson 1990) or BLAT (Kent 2002) are less popular. BLAST is a heuristic for finding optimal *local alignments* (Section 11.1) in two input sequences  $a$  and  $b$ . That means it searches similar substrings in  $a$  and  $b$ , where the *similarity* (Section 9.3.2 on page 85) between two strings is defined by the score of an optimal *alignment* between them. The longer the strings are and the less they differ, the higher is this score. BLAST does not only compute similar regions and scores, but it also estimates a statistical significance, i.e. it computes the probability for finding similar regions of a certain length in two *uncorrelated* sequences  $a$  and  $b$  simply by chance (Karlin and Altschul 1990). If this probability is very small, then we can conclude that the regions in  $a$  and  $b$  are probably *correlated* and there must be a reason for their similarity. Since BLAST runs very fast, it has turned out to be an extremely useful tool in practice, and therefore the paper

by Altschul, Gish, Miller, Myers, and Lipman (1990) became one of the most cited publications in science history. There are several variants (e.g. 'blastp' for Proteins or 'blastn' for DNA) and implementations (e.g. NCBI-BLAST or WU-BLAST) of the tool.

In short, the algorithm works as follows: BLAST first searches for *seeds* (Section 9.6.1), which are highly similar regions as for example exactly matching substrings of a certain length  $q$  (' $q$ -grams'). This means that BLAST finds only those local alignments that contain a seed, so it will find more alignments if the seed length  $q$  is reduced, although this will also slow down the search. The seeds may be found e.g. by an automaton (Section 13.1) or a  $q$ -gram hash index (Section 12.1). Each seed is extended in both directions by a *X-drop extension* (Section 11.2.1). The resulting local alignment is stored if it exceeds a certain level of quality. In the end, the best local alignments are printed out. SeqAn supports data structures and algorithms both for finding and extending seeds (Section 9.6.1 and 11.2), as well as functions for parsing the output of standard BLAST tool.

### 1.2.3 CLUSTAL W: Aligning Multiple Sequences

Among the most important tasks in sequence analysis is the *alignment* of sequences (Section 9.2): The sequences written one below the other form the rows of a matrix, and blank characters are inserted into these rows such that similar parts of the sequences are grouped together. The more matching characters stand in the same columns and the less blanks we needed to insert, the higher is usually the *score* of the alignment (Section 9.3.1). Alignments may explain a lot about sequences, since they reveal both the similarities between them but also the small differences within these similarities. If the sequences for example originate from different species, then the optimal alignment can be used to infer their phylogenetic relationship.

We will show in Section 9.5.1 how to compute an optimal alignment between two sequences by *dynamic programming* in quadratic time. Unfortunately, the runtime of grows exponentially for increasing numbers of sequences, and it was shown that the alignment problem is NP-hard (Wang and Jiang 1994), so practical tools for aligning multiple sequences are based on heuristics. One of the most common tools of this kind is CLUSTAL W, that applies a *progressive* approach, see Section 9.5.5. The tool works in three steps (see also Algorithm 5 on page 94):

- (1) The pairwise distances between the sequences are computed, either by counting common  $q$ -grams or by aligning them. The result is stored in a distance matrix.
- (2) From this distance matrix, a *hierarchical clustering* algorithm like UPGMA (e.g. Sneath and Sokal 1973) or *neighbor-joining* (Saitou and Nei

1987) computes a rooted binary ‘*guide tree*’. The leaves of the guide tree correspond to the sequences that are to be aligned.

- (3) The sequences are aligned following the guide tree from the leaves to the root. At each inner vertex of the tree, the multiple alignment between all leaves below this vertex is computed by *aligning the alignments* of the two child vertices, see Figure 3.

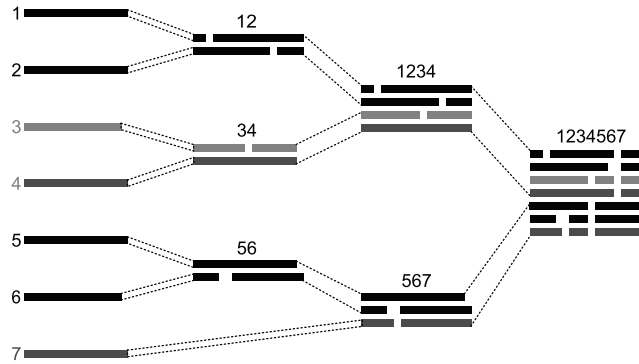


Figure 3: **Progressive Multiple Sequence Alignment.** The sequences on the left are aligned following a guide tree. Each vertex aligns the alignments from its child vertices, so the alignment on root vertex contains all input sequences.

This method is *greedy*, because once two sequences are aligned, then this alignment will be retained until the algorithm stops. A gap that is inserted will never be moved or removed again, and new gaps always affect the whole column of the alignment, thus any error that occurs in the early stages of the algorithm will be propagated to the end of the computation. As a remedy, CLUSTAL W applies a clustering algorithm to construct the guide tree, because this way similar sequences are joined earlier than distant sequences, and similar sequences are more likely to be aligned correctly.

SeqAn also offers progressive alignment algorithms that follow the improved T-Coffee tool (Notredame et al. 2000), see Section 9.5.5.

# Chapter 2

## Software Libraries

### 2.1 Software Libraries in General

One main goal of bioinformatics is to devise algorithms and develop software tools for biological and medical research. In this section, we will discuss how software libraries may improve the development of tools for sequence analysis. A *software library* is a set of reusable components, i.e. data structures and algorithms that use and manipulate these data structures. A component is *reusable*, if it can be used in different programs and by different programmers.

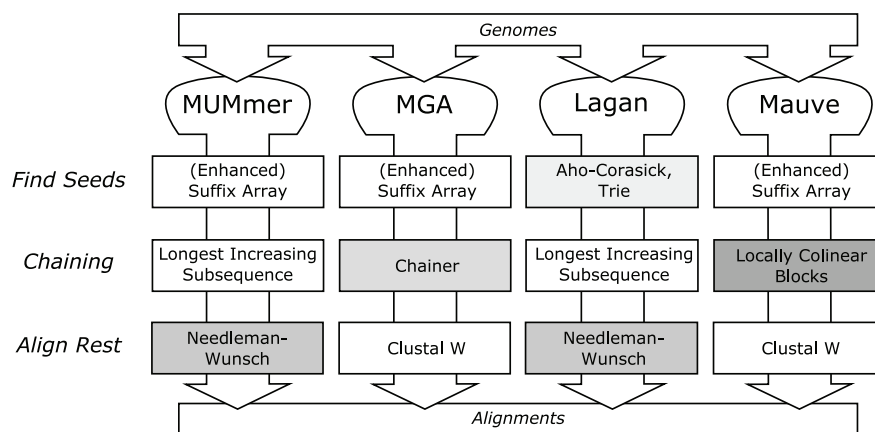


Figure 4: **Components of Genome Alignment Tools.**

This is illustrated in Figure 4 that shows the core components of four tools for genome alignment: MUMmer (Kurtz et al. 2004) MGA (Hohl et al. 2002) LAGAN (Brudno et al. 2003) Mauve (Darling et al. 2004). All these tools perform the following three steps: (1) search for seed fragments, (2) compute an optimal chain from these seeds (Section 9.6), and finally (3) close the gaps between the seeds. Obviously these tools apply similar building blocks, like

(enhanced) *suffix arrays* (Section 12.2) for seed finding, ‘longest increasing subsequence’ (Section 13.2.2) for chaining, or the Needleman-Wunsch algorithm (Section 9.5.1) for aligning the spaces between the chained seeds. In the example it is evident, that the developers of these tools would have profited from having a software library at hand that provides efficient implementations of the named algorithmic components.

### 2.1.1 Benefits from Software Libraries

*Software development* may benefit from software libraries in several ways: Their application *simplifies* the implementation of software, if a programmer can employ ready-to-use components from the library instead of re-implementing every part of the program, or to bother about the implementation details of the actually used algorithms or data structures. This *accelerates* the development process, which allows the program to be earlier on the market or – in the case of a bioinformatic tool – in the laboratories. These time savings *reduce the costs* of software development. Software libraries may also improve the *quality* and *robustness* of the resulting code, because the components of a library are widely used and therefore usually well tested. Moreover, the application of library components may also improve the program’s *performance*, since libraries are intended to be used many times, so they may offer advanced but fast algorithms or data structures the implementation of which would not pay out for a single project. This also shows that software libraries support *algorithm design* by providing benchmarks for well known problems and acting as test environments in which a new algorithm may prove its correctness and possibly its superiority to previous approaches. A well designed library invites the user to ‘play around’ with algorithms and data structures and thus promotes a fast testing of new *algorithmic ideas*. Finally, the best way for a new algorithm to arrive in software development is to publish it in a widely used software library, so libraries may help to close the gap between theory and practice in algorithmic research.

### 2.1.2 Software Library Examples

We will now shortly present two software libraries that are good examples for the usefulness of libraries in software development and that influenced the design of our library SeqAn.

#### LEDA

The ‘*Library of Efficient Data Types and Algorithms*’ (Mehlhorn and Näher 1999) is a C++ library for combinatorial and geometrical computing. Since it was proposed in 1989 by Mehlhorn and Näher, LEDA grew to an extremely



comprehensive software library. The range of its functions contains basic containers like arrays, lists, or sets, and it provides number types and graphs as well as algorithms and data structures for linear algebra, geometry, compression, and cryptography. LEDA was designed from the beginning to advance the transfer of theoretical algorithmic knowledge to practical tool programming, and in this respect it became a model for software libraries like SeqAn.

## STL

The ‘*Standard Template Library*’ (Stepanov and Lee 1995) is a C++ template library of basic containers and algorithms. With some changes and extensions, it became a part of the C++ standard library (ISO/IEC 1998; Josuttis 1999), so we will call this part of the C++ standard library the STL. The STL was one of the first C++ libraries that applied *generic programming* (e.g. Austern 1998, see Section 5.2), and it demonstrates that this programming paradigm is capable to implement flexible and performant libraries. The algorithms of the STL access and modify the contents of the containers by *iterator* objects; further key concepts of the STL are *functionals*, i.e. objects that implement the parenthesis operator (), and *type traits*, that resemble the *metafunctions* that we use in SeqAn (Section 5.5). The *library design* of SeqAn (Chapter II) can be regarded as an advancement of the techniques that were introduced into library design by the STL.

## 2.2 Bioinformatics Libraries

During the last decades, data structures like *suffix trees* (Section 12.3.2) or *suffix arrays* (Section 12.2) and algorithms like *Needleman-Wunsch* (Section 9.5.1) or *chaining by sparse dynamic programming* (Section 9.6.3) were successfully applied in a variety of bioinformatics tools. This suggests that software libraries could also be useful in the case of *sequence analysis*. In the following paragraphs we will review existing sequence analysis projects and point out the content and main purpose of the software libraries. There are several alternatives to design a library, mostly chosen with certain goals in mind: Some libraries want to glue together existing tools, possibly implemented in a variety of programming languages (Section 2.2.1), some others want to serve as a development platform (Section 2.2.2). We will end this comparison with an introductory description of SeqAn in Chapter 3.

### 2.2.1 Libraries for Using Existing Tools

One way to solve sequence analysis problems is to ‘stitch’ together already existing tools like BLAST (Altschul et al. 1990), CLUSTAL W (Thompson et al. 1994) or LAGAN (Brudno et al. 2003). Moreover, there are collections of

well matched tools like EMBOS (Rice et al. 2000) and Vmatch (Kurtz 2007). Vmatch for example is a set of versatile programs that can be combined to solve a variety of string matching tasks using a string index (see also Chapter 12). Joining tools usually requires ‘glue code’ for controlling the overall process, or for translating the output of one tool into a valid input for another tool. Since most of the time consuming tasks are done by the joint tools, the performance of the glue code is often less critical, so scripting languages like Perl (Wall 2000) are convenient for that purpose. Software libraries like *Bio-perl* (Stajich et al. 2002), *Biopython* (Chapman and Chang 2000), or *Bioruby* (Goto et al. 2003) may facilitate the development of this glue code. They offer comprehensive sets of wrappers to bioinformatics tools and input/output capabilities for various file formats and data bases. They also facilitate the access of sequence annotation data.

## 2.2.2 Libraries for Development of New Tools

Although the combination of already existing tools might be a good strategy for cope with many problems in bioinformatics, this does obviously not solve the problem of initially implementing the tools. For that reason a second kind of software library is needed to provide algorithmic components that can be used to implement new software tools. We will now shortly review some software libraries (in alphabetical order) that offer a fair amount of functionality usable for sequence analysis.

### BATS

The ‘*Basic Analysis Toolkit for Biological Sequences*’ (BATS; Giancarlo et al. 2007) is a collection of some functions for approximate string search, sequence alignment, sequence filtering and *z*-score computation. Parts of this library were included into SeqAn.

### Bio++

*Bio++* (Dutheil et al. 2006) is a comprehensive software library that provides reusable components for sequence analysis, phylogenetics, molecular evolution and population genetics. The sequence analysis part contains data types for storing and manipulating strings, string sets, alignments, and several convertible alphabet types, it supports the management of sequence annotations, the import and export of many file formats, and some basic sequence analysis tools including the Needleman-Wunsch algorithm (Needleman and Wunsch 1970). The strength of Bio++ is certainly its contribution to phylogenetics and molecular evolution. In contrast to SeqAn which applies the *generic programming* paradigm (Section 5.2), Bio++ is a purely object-oriented library that favors ease of development over performance and scalability.

### Bioconductor

*Bioconductor* (Gentleman et al. 2004) takes advantage of the programming language *R* (Ihaka and Gentleman 1996) that was designed to perform tasks in statistics. This software collection facilitates the (statistical) analysis of biological data by providing several hundred packages, many of them determined for very special applications. Some of these packages – e.g. the ‘Biostrings’ module – also offer common sequence analysis functionality like alignment and pattern matching.

### BioJava

*BioJava* (Holland et al. 2008) is a very extensive library that implements various data structures and algorithms for bioinformatics in the programming language *Java* (Arnold et al. 2005). Beside many tasks that are typical for ‘tool-stitching’ libraries as we described them in Section 2.2.1, BioJava also supports typical data structures and algorithms that are applied in common sequence analysis tool, like the Needleman-Wunsch (see Section 9.5.1) and Smith-Waterman (see Section 11.1.1) alignment algorithms, position specific weight matrices, hidden Markov models, and suffix trees.

### BTL

The ‘*Bioinformatic Template Library*’ (BTL; Pitt et al. 2001) emphasizes more on basic mathematical algorithms and data structures than on sequence analysis. It currently comprises some graph classes and linear algebra algorithms, but only a single sequence alignment algorithm: The Needleman-Wunsch algorithm for general gap costs with cubic running time (Needleman and Wunsch 1970).

### libcov

*Libcov* (Butt et al. 2005) has a focus on phylogenetics and clustering algorithms. It offers only basic data structures to handle sets of sequences.

### libsequence

The C++ class library *libsequence* (Thornton 2003) was designed to support the development of tools for evolutionary genetic analysis by providing basic sequence manipulation functions as well as algorithms for calculating sequence divergences and for *single nucleotide polymorphism* (SNP) analysis.

### NCBI C++ Toolkit

The NCBI C++ *Toolkit* (Vakatov et al. 2003) contains some functionality that was used to implement tools and services from the *National Center for Biotechnology Information*. The toolkit offers, beside other things, some sequence analysis functionality, like several very fast algorithms for sequence alignment. Moreover it contains an API for the NCBI implementation of BLAST (Altschul et al. 1990).

### SCL

The ‘*Sequence Class Library*’ (SCL; Vahrson et al. 1996) is a object-oriented C++ library that provides some basic sequence analysis components. To our knowledge it is not actively developed anymore.

## 2.2.3 Conclusion

The survey in the last section revealed that there was no software library for developing new software tools in C/C++ that covers all relevant areas of sequence analysis. The two most comprehensive libraries in the list – BioJava and BioConductor – support other programming languages (Java and R), but regarding the performance they cannot match up with C/C++; a comparison between the running times of alignment algorithms in different libraries will demonstrate this in Section 9.5, see the results in Table 30 on page 172.

Bio++ is yet the most elaborated library for C/C++, however it lacks basic sequence analysis functionality like string matching, motif searching and chaining, and it also provide no data structures for string indices or automata. Certainly Bio++ could be extended by these components; however we will argue in Section 5.3.2 that it is might be difficult to achieve very efficient code in an object-oriented approach as it is used in Bio++. The development of the BTL and SCL were stopped for years, and their contents have never reached the extent of SeqAn or even Bio++. The rest of the proposed C/C++ libraries (BATS, libcov, libsequence and the NCBI C++ Toolkit) were never designed to become comprehensive sequence analysis libraries, but just confine to provide a few useful tools or components. Moreover, the BTL, libcov and libsequence mainly focus on other applications than sequence analysis.

Hence we believe that there is no extensive software library for sequence analysis that facilitates the development of software tools with nearly optimal performance. The goal of the SeqAn project is to fill this gap.

# Chapter 3

## SeqAn

To conclude the introduction we summarize the goals and design of SeqAn, our generic C++ template library of efficient data types and algorithms for sequence analysis. The development of SeqAn has pursued two main goals, namely:

- (1) Enabling the rapid development of efficient *tools* for sequence analysis.
- (2) Promoting the design, comparison, and testing of *algorithms* for sequence analysis.

SeqAn accelerates the development process of tools and algorithms, and improves the quality and performance of sequence analysis software. In addition, it provides an experimental platform for algorithm engineering and closes the gap between state-of-the-art algorithmic techniques and the actual algorithmic components used in software tools (Section 2.1). SeqAn is the first software library with this ambition that was actually realized, see Section 2.2.



Figure 5: The SeqAn Logo.

### 3.1 Design of SeqAn

SeqAn was designed to promote (1) high *performance* of the provided components, (2) *simplicity* and usability of the library's handling, (3) *generality* of data types and algorithms such that they are widely applicable, (4) the definition of special *refinements* of generic classes or algorithms, (5) the *extensibility* of the library, and (6) easy *integration* with other libraries (Chapter 4).

We decided to implement the library in C++, since C++ provides language constructs that allow to achieve our design goals (see Section 5.1). The unique library design of SeqAn bases on (1) the *generic programming* paradigm, (2) a new technique for defining type hierarchies called *template subclassing*, (3) *global interfaces*, and (4) *metafunctions*, which provide constants and dependent types at compile time (Chapter 5). Our design differs from common programming practice, in particular SeqAn does *not* use object-oriented programming (Section 5.3.2). However, we will argue that in effect the library greatly benefits from our approach, and all design goals are met (Chapter 14).

### 3.2 Contents of SeqAn

SeqAn is a comprehensive library that was intended to cover a wide range of topics of sequence analysis. It offers a variety of practical state-of-the-art algorithmic components that provide a sound basis for the development of sequence analysis software. This includes: (1) data types for storing strings, segments of strings and string sets, as well as functions for all common string manipulation tasks including file input/output, (2) data types for storing gapped sequences and alignments, and also algorithms for computing optimal sequence alignments, (3) algorithms for exact and approximate pattern matching and for searching several patterns at once, (4) algorithms for finding common matches and motifs in sequences, (5) string index data structures, and (6) graph types for many purposes like automata and alignment graphs, as well as many algorithms that work on graphs (Part III).

SeqAn offers several alternative implementations for all core data types like strings, string sets, alignments, graphs, and indices. It also provides a variety of different algorithms for central tasks like pattern matching, motif finding, or the alignment of sequences. The user can therefore select the variant that fits best to the actual application.

### 3.3 This Thesis

The thesis at hand gives a detailed description of the software library SeqAn. The author contributed to this project mainly the following: He developed the main library design (Part II), i.e. the basic structure of the library

and the definition of the rules and methods used during the development of the library, and he was also responsible for a good part of the essential library functionality (see Chapters 7 to 11), both by means of personal contribution and supervision of students who supported the implementation in various projects and theses. Moreover, the author occupied coordinating functions within the project, and designed the main part of the project's working environment like the built system and the documentation system (see Section 14.2.2).

The following parts of the thesis describe the design and content of SeqAn:

**Part II** describes the general *library design*. We start in Section 4 by declaring the main goals we want to achieve with the library design. The means to reach these goals are proposed in Section 5, where the main programming techniques used in SeqAn are elaborated on. The application of these techniques is demonstrated using examples in Section 6.

**Part III** explicitly describes the components provided by SeqAn. After proposing some basic functionality in Section 7, we describe sequence data structures in Section 8, alignments in Section 9, pattern and motif searching in Sections 10 and 11, string indices in Section 12, and graphs in Section 13.

**Part IV** is the conclusion of the thesis. Section 14 explains our measures for quality assurance and the propagation of SeqAn, and argues why SeqAn is a viable and useful software library. In Section 15 we use SeqAn to re-implement the core functionality of the well-known software tool LAGAN (Brudno et al. 2003) for genome alignment. The resulting program takes only a hundred lines of code and is competitive to the original software.





# Part II

## Design

*SeqAn relies on a unique generic design. In this part, we explain the main goals that we pursued by the library (Chapter 4), as well as the programming techniques that we applied to achieve these goals (Chapter 5). These techniques are illustrated by some examples in Chapter 6.*



# Chapter 4

## Design Considerations

### 4.1 Design Overview

In this part, we discuss the *core library design* of SeqAn. We call it ‘core’ design, because it answers very basic questions like: What are the strategies for organizing the functionality in the library? What is the general form of classes and functions? What language features are applied, and how they are used? The core design does *not* specify what classes and functions should be implemented in the library. This ‘detailed’ design will be the topic of Part III, in which we will give a complete overview of the contents of SeqAn.

Although the core design is not directly connected with the actual contents of the library, it is influenced by the kind of functionality the library offers. For example we observe that sequence analysis relies on rather simple but *generic* data structures like sequences (Chapter 8), alignments (Chapter 9), string indices (Chapter 12), and graphs (Chapter 13) which makes it amenable to the generic programming paradigm, whereas libraries consisting of less generic but very complex data structures would probably be better implemented in a more *object oriented* way.

The decision for an appropriate core design also depends on the intended *application* of the library. As we stated in Chapter 3, SeqAn has the purpose to facilitate the development of new sequence analysis tools, and it is an algorithm engineering platform for comparing and developing efficient data structures and algorithms. Both applications require that the components of the library run as fast as possible, so *performance* is one of the most important objectives during the library design phase. Considerations like this lead us in Section 4.2 to six *main goals* for the core design of SeqAn. In the following Chapter 5, we will discuss by what programming techniques these goals can be achieved. It turns out that only a few powerful techniques suffice. The mechanics of the resulting core design is then demonstrated by examples in Chapter 6.

## 4.2 Design Goals

### 4.2.1 Performance

A first – and maybe most important – objective for SeqAn is *performance*:

*‘The library is designed to produce code  
that runs as fast as possible.’*

Since data structures usually must fit completely into main memory to be fast, we also aspire to offer data structures with minimal space consumption.

While performance is of course a desired feature of any software, it plays a critical role in the competition between software tools. For example, some applications in bioinformatics involve huge problem instances which may take running times of several hours or even days, so a tool’s speed can make the difference between a feasible and an infeasible experiment if compute costs are limited.

In sequence analysis the amount of data to be analyzed usually forbids the application of brute force algorithms even for very basic tasks like searching a pattern in a string or aligning two sequences. Hence, one has to resort to efficient data structures and algorithms that achieve the required speedup. A library can supply very complex algorithmic components, which are hard or costly to implement for tool designers. Nevertheless, a tool designer has always the option to solve the problem at hand by its own specialized code or to resort to ad-hoc solutions, instead of using the components in a library. Programming specifically for a given problem may even yield better results than using standard components, depending on the effort involved, hence it is crucial for the library components to be competitive in speed to specialized code.

Optimal performance is also crucial for algorithm engineering: No algorithm designer would be happy to sacrifice hard-earned speedups obtained from a clever algorithmic advance due to suboptimal implementations. A comparison between competing algorithms would always be influenced by varying implementation qualities, so the best way to make it fair is to compare implementations that are as effective as possible and are based on the same algorithmic components.

The need for performance is also our main reason to choose C++ as programming language (Stroustrup 2000) (ISO/IEC 1998), since carefully designed C++ code has best chances to outperform most alternative programming language (see Section 5.1).

Achieving a good performance affects the library in many respects. If there is a trade-off between speed or coding convenience, then performance is usually favored. For example, we omit time consuming parameter checks in the release build of the library.

### 4.2.2 Simplicity

The second main goal for the library design of SeqAn is *simplicity*. Software libraries should facilitate the development of software, and hence they need a clear organization of their functionality. Plain interfaces improve a library's usability and accessibility, make it easier for a potential user to evaluate the usefulness of the library, and reduce the training needed to use it. In addition the internal mechanisms of a library should never get too complex, since this would slow down the development process of the library, complicate its maintenance, and it could be a source of hidden errors. The more sophisticated a library was constructed and the more elaborated language features were applied, the higher is the risk for the user to become a victim of exotic compiler behaviour, unreadable error messages, or inconsistencies in the language standard.

Our goal is therefore:

*‘All parts of the library are constructed and applicable  
as simple as possible.’*

We feel confident that the application of SeqAn is in fact simple, although this is always in the eye of the beholder, and in this thesis we will demonstrate the ease of use in a multitude of short code examples.

### 4.2.3 Generality

The next goal of SeqAn is *generality*. Library designers cannot completely anticipate all applications a library will actually be used for, so it is advisable to keep it as general as possible. A library that is useful in many circumstances has better chances to be used. Also, the more probable it is that a library can be re-used in future occasions, the more it pays for a user to get accustomed with it. Hence our goal is:

*‘All parts of the library are applicable  
in as many circumstances as possible.’*

General components are more intuitive to describe and easier to understand than data structures and algorithms that are only usable for a few individual cases, so generality also supports the *simplicity* (Section 4.2.2) of the library. A good starting point for finding general components is to identify common elements in different tools, as we described it in Section 2.1.

Generality also means that we try to avoid redundancy in the library: If, for example, one algorithm can work on different types of classes, then it should not be re-implemented for each class, but only once for all classes, in a single piece of code. This makes the library more compact and easier to maintain.

We will explain in Section 5.2 how *generic programming* enables us to create data structures and algorithms that work on a variety of types, for example how to implement strings of arbitrary alphabets, or algorithms that work on any kind of string.

#### 4.2.4 Refineability

A good strategy for augmenting performance (Section 4.2.1) is to implement *specializations*: Sometimes the implementation of a function can be significantly improved, if we rely on a special context or the presence of some constraints. For example, searching an array is much faster if the values are sorted, thus it is advisable to define both a general but relatively slow linear search algorithm for unsorted arrays, and additionally a fast binary search algorithm for sorted arrays. A *specialization* overloads the *general* solutions (Section 4.2.3) for a special case, and the specialization can also be overloaded for an even more special case, so in the end we get a ‘*hierarchy of refinements*’.

The ideal library concept therefore fulfills the following rule:

‘*Whenever a specialization is reasonable,  
it is possible to integrate it easily into the library.*’

‘To integrate’ means, that the new specialization works seamlessly together with the rest of the library, and that it can be applied the same way as already existing alternatives. Our design therefore supports *polymorphism*, i.e. that the same interface may be realized by several implementations. This also promotes the simplicity (Section 4.2.2) of the library.

We will see in Section 5.3, how *template subclassing* enables us to implement specializations in a way, that the C++ compiler always uses the most appropriate – i.e. the most special – variant.

#### 4.2.5 Extensibility

A classical slogan of good programming is the so called ‘*open-closed principle*’, which states that a program should be open for extension but closed for modifications. We call this feature *extensibility*:

‘*The library can always be extended  
without changing already existing code.*’

‘Extending the library’ means to overwrite default behavior by defining new specializations (Section 4.2.4), or to add completely new functionality to the library. Extensibility is important both during the implementation of the library, because it simplifies its construction, and also for a user who wants to adapt the library to his needs.

### 4.2.6 Integration

It is often reasonable to use several libraries at once. This means that the libraries must be able to collaborate with each other:

*‘The library is able to work together with  
other libraries and built-in types.’*

This includes that SeqAn obeys some ‘rules of coexistence’, for example, to use its own namespaces `seqan` in order not to contaminate the global namespace, or not to define preprocessor macros that could conflict with code of other libraries. Moreover, we aim at providing means for a direct integration of external libraries: For example, string classes are provided not only by SeqAn (see Section 8.3) but also by many other libraries like the STL (Plauger, Lee, Musser, and Stepanov 2000) or LEDA (Mehlhorn and Näher 1999), and strings can also be stored in `char` arrays, so called ‘C-style strings’. It would be of great advantage, if we could implement algorithms that work on all these kinds of string.

We will explain in Section 5.4 how the SeqAn library design supports this kind of integration by using small global functions or metafunctions – so called ‘*shims*’ – to adapt external interfaces to the needs of SeqAn.





# Chapter 5

## Programming Techniques

In this chapter, we discuss the main techniques used in SeqAn to achieve the design goals that we described in Section 4.2, namely *generic programming* (Section 5.2), *template subclassing* (Section 5.3), *global interfaces* (Section 5.4), and *metafunctions* (Section 5.5). The combination of these four techniques forms the *core design* of SeqAn. In Section 5.6 we will propose further programming techniques that we apply in SeqAn. We start with discussing the reasons for using the programming language C++.

### 5.1 The C++ Programming Language

The programming language C++ was proposed by Bjarne Stroustrup in 1983 (see Stroustrup 2000) as an extension of the procedural and imperative programming language C (Kernighan and Ritchie 1988). SeqAn relies on ISO/IEC standard conform C++ (ISO/IEC 1998) that is supported by several compilers like the GNU C++ compiler (Griffith 2002) or the Visual C++ compiler (Visual C++ 2002).

The programming language C was designed as “a relatively ‘low level’ language” so that “the data types and control structures provided by C are supported directly by most computers”.<sup>1</sup> Although C was formulated to be independent from a particular architecture, it is in effect rather machine-oriented, which means that C programs matches the capabilities of present computer architectures and have therefore best chances to run fast. During the compilation of the C source code, the compiler may also apply optimizations to achieve further speed-ups. C++ enhances C by concepts like *object-oriented programming* and *generic programming* (Section 5.2), and though some of these new features (e.g. ‘virtual functions’) entail pitfalls to slow down the resulting programs, carefully employed C++ achieves in general the same performance as C. Due to the prevalence of C/C++ in the last decades, the co-evolution of

---

<sup>1</sup>see (Kernighan and Ritchie 1988), pages 5–6

computers and compilers made these language probably the best choice for high-performance applications. We decided to implement SeqAn in C++, because *performance* is among our main goals (Section 4.2.1) and the extended features of C++, namely *templates* (ISO/IEC 1998, 14), are well suited to attain an excellent *library design*.

There are prominent examples of C++ software libraries in the area of algorithm engineering like LEDA (Mehlhorn and Näher 1999) and CGAL (Fabri et al. 2000), many common software tools for sequence analysis like NCBI Blast (Altschul et al. 1990) are implemented in C++.

## 5.2 Generic Programming

SeqAn adopts *generic programming*, a paradigm that was proven to be an efficient design strategy in the C++ standard (ISO/IEC 1998). The standard template library (STL) (Plauger, Lee, Musser, and Stepanov 2000) as part of the C++ standard is a prototypical example for generic programming. Generic programming designs algorithms and data structures in a way that they work on all types that meet a minimal set of requirements. An example for a generic data structure in the STL is the class `vector`: It is a container for storing objects of a type `T` that are *assignable* (ISO/IEC 1998, 23.1), which means that we can assign one instance `s` of `T` to another instance `t` of `T`, i.e. the code `T t = s` is valid. This kind of requirement to the interface of a type `T` is called a *concept*, and we say that a type `T` *implements* a concept, if it fulfills all requirements stated by that concept; for example the concept ‘assignable’ is implemented by all built-in types and every class that has both a copy assignment operator and a copy constructor. Generic programming has two implications: (1) Data structures and algorithms work on *all* types `T` that implement the relevant concept, i.e. relevant is not the type `T` *itself* but its interface, and (2) this concept is *minimal* in the sense that it contains only those requirements that are essential for the data structure or algorithm to work on `T`. This way data structures and algorithms can be applied to as many types as possible, and hence generic programming promotes the *generality* of the library (see Section 4.2.3).

Generic data types and algorithms can be implemented in C++ using *templates* (ISO/IEC 1998, 14). A class template parametrizes a class with a list of types or constants. For example, a declaration for the class `vector` could be:

```
template <typename T> class vector;
```

where `T` stands for the *value type*, i.e. the type of the values that will be stored in `vector`. The template is generic, it can be applied to any type `T`. For example, a vector for storing `int` values is instantiated by:

```
vector<int> my_vector;
```

That is we use `int` as template argument for `T`, and the result of the instantiation is an object `my_vector` of the *complete type* `vector<int>`. The compiler employs the same template, i.e. the same piece of code, for different template argument types. The compilation succeeds if the applied template argument type supports all uses of the parameter `T` within the template code, so the C++ template instantiation process implies the *minimality* of the concepts.

Listing 1 shows an example for a generic algorithm. The function template `max` can be applied for two objects `a` and `b` of any type `T` that is *assignable* and can be compared using the `<` operator. The compiler may implicitly derive the type `T` from the given function arguments, for example `max(2, 7)` calls the instantiation of `max` for `T = int`.

```
template <typename T>
T max(T a, T b)
{
    if (a < b) return b;
    else return a;
}
```

Listing 1: **Example for a Generic Algorithm.** The function template `max` returns the maximum of two values `a` and `b`, where `a` and `b` could be from any suitable type `T`.

## 5.3 Template Subclassing

A generic algorithms that is applicable to a type `T` needs not to be optimal for that type. The algorithm `find` in the standard library (ISO/IEC 1998, 25.3.1.1) for example performs a sequential linear time search and is therefore capable of finding a given value in any standard compliant container. However, the container `map` was designed to support a faster logarithmic time search, so the algorithm `find` – though applicable – is not optimal for searching in `map`. This shows that sometimes a special algorithm could be faster than a generic algorithm. Hence, in order to achieve better performance (Section 4.2.1), we require our library (see Section 4.2.4) to support *refinements* of algorithms. A special version is only useful, if it really allows a speedup in some cases, and only in this case it will actually be implemented. Therefore we assume that for a given case always the *most special* applicable variant is the best, where we have to assure that there is always a definite ‘most special’ candidate according to the C++ function overload resolution rules (ISO/IEC 1998, 13.3 and 14.5.8).

<pre> template &lt;typename TValue,           typename TSpec&gt; class Container {     // generic container };  struct Map;  template &lt;typename TValue&gt; class Container&lt;TValue, Map&gt; {     // special map container }; </pre>	<pre> template &lt;typename T&gt; void find(T &amp;) {     // most general:     // for all types }  template &lt;typename TValue,           typename TSp&gt; void find(Container&lt;TValue, TSp&gt; &amp;) {     // more special:     // for all containers }  template &lt;typename TValue&gt; void find(Container&lt;TValue, Map&gt; &amp;) {     // even more special:     // only for maps } </pre>
---	---

Listing 2: **Template Subclassing Example.** Note that SeqAn does not implement a class `Container`.

Since one of our goals is *simplicity* (Section 4.2.2), and since it could be rather demanding for the user to find out the best algorithm out of various alternatives, we decide to apply *polymorphism*, that is all alternative implementations of an algorithm support the same interface. So we can write `find(obj)` for any container type `obj`, and this invokes the most suitable implementation of `find` depending on the type of `obj`. Listing 2 gives an example for this idea: The map is implemented in the specialization `Container<Map>` of the generic class `Container`. Since the subclass is specified by choosing a template argument, we call this approach *template subclassing*.

On the right side of Listing 2, we see different levels of specificity for `find` algorithms: The first is applicable for any type – therefore we call it the *most general function* –, the second for instances of `Container`, and the third only for `Container<Map>` objects. The rules of C++ *function overload resolution* (ISO/IEC 1998, 13.3) assures that the correct variant is called.

We need not to end the specialization on the level of `Container<Map>`. Suppose that we define `Map` as a class template with a template parameter `TSpec`, then we can also implement special variants of maps, for example `Container<Map<Hashing>>`. This way, we can define specialization hierarchies of unlimited ramification.

Note that we make no demands on template argument types like `Map` that are used for `TSpec`, in fact any type that is merely *declared* can be used as template argument, so its only important aspect is to act as a switch between different

specializations of `Container`. We call a class that is intended merely to serve as switch a ‘*tag class*’. Tag classes can also be used to switch between different modes of a function (see Section 5.6.2).

### 5.3.1 Template Subclassing Technique

The technique of *template subclassing* may be summarized as follows:

- The data types are realized as default implementation or specialization of class templates, e.g. `Class`, which have at least one template parameter `TSpec`.
- Refinements of `Class` are specified by using in `TSpec` a *tag class*, e.g. `Subclass`, that means they are implemented as class template specializations `Class<Subclass>`.
- Whenever further refinements may be possible, we declare the tag classes as class templates with at least one template parameter `TSpec`, in which more tag classes can be used. For example we may implement a class template specialization `Class<Subgroup<Subsubgroup<...>>>`. This way, we can reach arbitrary levels of specialization.
- Algorithms can be implemented for each level of specialization. If multiple implementations for different levels of specialization exist, then the C++ function overload resolution selects the most special from all applicable variants.

Note that we only need to define a class template specialization `Class<Subclass>` explicitly, if the *members* of the new refinement differ from the members of its ‘parent class’ `Class`. We will see in the Sections 5.4 and 5.5, that member functions and member types play a minor role in SeqAn, so in many cases an actual specialization of the class template is not needed.

### 5.3.2 Comparison to Object Oriented Programming

Template subclassing resembles class derivation in standard object-oriented programming. In the spirit of the object-oriented terminology we can say that the class `Container<Map>` was ‘derived’ from the general class `Container`, since all algorithms that are defined for `Container` also work for `Container<Map>` and are therefore ‘inherited’. Let us compare this to object oriented programming: The method `find` is inherited from the base class `Container` to all derived classes, but it was overloaded for the class `Map` which defines its own `find` method.

This approach has two drawbacks: The first disadvantage is that the method `find` is not a generic algorithm according to the definition in Section 5.2 but

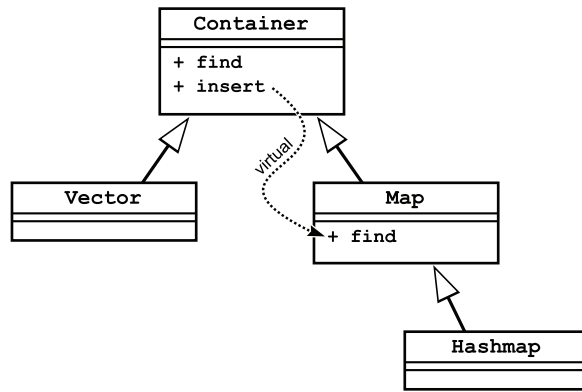


Figure 6: Object Oriented Example.

a member function. Its usage has the form `obj.find()`, which differs completely from the application of the generic algorithm that would be called by `find(obj)`. If we try to achieve the *refinement* goal (Section 4.2.4) this way, and if we therefore define both global functions and member functions, then the handling of the library gets more complicated, so we lose *simplicity*.

Object-oriented programming has another drawback, see Figure 6: Suppose that we want to define a function `insert` that adds a new value to a container, and that `insert` calls `find` in order to check whether the container already holds the given value. Since `insert` can be applied for all containers, we implement it as a member function of `Container`, so it is inherited also by `Map`. If we call `insert` to insert a value into a `Map` object, then `insert` should use the correct `find` function, i.e. the special ‘logarithmic search’ that was defined in `Map` rather than the general ‘linear search’ defined in `Container`. Therefore `find` has to be declared *virtual* (ISO/IEC 1998, 10.3), which means that each call of `find` costs an additional overhead: Virtual function calls are indirect via the lookup in a table of function pointers, and they are therefore more complicated than ‘ordinary’ function calls; in contrast non-virtual functions have the advantage that C++ compilers may use function inlining to completely save the overhead for calling them. The application of virtual functions in this case therefore reduces the *performance*.

We conclude that template subclassing better fits to our design goals than object-oriented programming: With template subclassing we can implement *polymorphic generic algorithms*, which means that the refinements of algorithms have the same interface as their generic counterparts. Template subclassing also needs no virtual functions but relies completely on *static function binding*, that is the compiler determines during compile time which function is called and can therefore apply optimizations to improve performance.

## 5.4 Global Function Interfaces

A *global function* in C++ is a function that was declared in namespace scope (ISO/IEC 1998, 3.3.5). In contrast to that, *member functions* are defined in the scope of a class. *Object-oriented programming* prefers member functions, because they work as *methods* which can be inherited and overloaded during the class derivation. *Generic programming* on the other hand also applies *global function templates* for implementing *generic* algorithms (see Section 5.2). In Section 5.3 we saw how we can use global function templates to implement algorithms for different levels of specialization. SeqAn relies on global functions anyway, and therefore the following design decision seems rather natural: SeqAn abstains from accessing objects via member functions as far as possible, that is, all functionality in SeqAn is accessed via global functions, with the exception of functions that must be members due to language restrictions, like constructors, destructors, assignment operators, bracket and parenthesis operators, and conversion functions. Since global functions substitute those member functions that would otherwise form the *interface* of a class, we call all functions that accept instances of a class as arguments the ‘*global function interface*’ of this class. For example, to determine the length of a string `str` in SeqAn, we call a global function `length(str)` instead of a member function `str.length()`. Using global interfaces is a main feature of SeqAn, and we will see in Section 5.5, that SeqAn also applies a global interface for accessing types.

Obviously the most direct way to achieve global function interfaces is simply to implement the functionality in global functions. This is also a prerequisite for using *template subclassing*, so most implementations in SeqAn actually reside in generic global functions, especially as far as this concerns the classes defined in the library. Alternatively, one can implement the functionality in a member function, and then call it via a ‘*shim*’ (Wilson (2004), Ch. 20), i.e. a small global function that act as a ‘wrapper’ for the member function. Shims are a good way to create new global interfaces for already existing data structures or for built-in types. For example, SeqAn contains several global functions `length` that work on the `basic_string` class from the C++ standard library or on zero terminated `char` arrays (so called *C-style strings*). So for determining the length of string `str` we can always call `length(str)` regardless whether `str` is an instance of the SeqAn class `String` (Section 8.1), or a `basic_string`, or an array of `char`. See Section 6.1 for more details about this example.

### 5.4.1 Advantages of Global Interfaces

Overall there are some good technical reasons to favor global functions over member functions (see also Czarnecki and Eisenecker 2000, 6.10.2):

- Global functions greatly support the *open-closed principle*, i.e. they fa-

vor the *extensibility* (Section 4.2.5): New functions can be added to the library at any time without changing the library’s code. This holds true for new specializations of already existing functions, as well as for completely new functionality. Moreover, it is possible to encapsulate the declarations of global functions in different header files and include them only if they are needed.

- The shim technique allows us to adapt arbitrary types to uniform interfaces, so using global functions is a good way to attain *integration* (Section 4.2.6) of the library with other libraries and built-in types.
- The difference between global ‘algorithms’ and non-global member functions, as they are used for example in the standard template library, can be somewhat confusing, especially if there exist algorithms and member functions with the same name. Therefore, abandoning global functions *simplifies* the library (Section 4.2.2). Moreover, global functions do not assume that one special function parameter acts as the ‘owner’ of the function, so they may sometimes be more intuitive e.g. when modeling symmetric operations like a matrix multiplication, which has no preference to be a member of the first or the second matrix.
- As we saw in Section 5.4, the obvious way to implement generic algorithms in C++ is to use global function templates, so global functions help to achieve a maximal *generality* (Section 4.2.3) of the library.

### 5.4.2 Discussion

Many programmers will probably at first be skeptical about our preference of global functions, since our approach contradicts common rules of object oriented programming, so we now discuss some possible objections.

#### Missing Protection

Global functions lack a protection model: They cannot be private nor protected, and they cannot access private and protected members of a class.

We addressed this problem by establishing rules of good coding practice. The main reason for a protection model is to prevent the programmer from accessing functions or data members that are intended for internal use only. A simple substitution for this feature is to establish clean naming conventions: We state that a ‘\_’-character within an identifier indicates that it is for internal use only. Global functions can access private members of a class  $\mathcal{C}$  if they are declare to be ‘friends’ of  $\mathcal{C}$ , but our experience showed that this approach is to inconvenient in practice, so we instead decided to declare data members to be public, but only functions that belong to the core implementation of  $\mathcal{C}$  are allowed to access them by convention.



### Possible Ambiguities

One could argue that we risk ambiguities when we define functions for several classes with the same name. Suppose that a class `String` and another class `Tree` should both support a function `length`. Then we simply implement two functions `length(String & str)` and `length(Tree & str)`, and this will work since both functions have different argument lists. A problem may only arise if multiple functions could be applied for the given arguments, in this case, we have to take care that there is always a ‘best’ alternative according to the C++ rules for function overload resolution (ISO/IEC 1998, 13.3 and 14.5.8).

### Handling of Namespaces

To avoid conflicts with other libraries, SeqAn defines all data types and functions for public use in the special namespace `seqan`. Nevertheless we need not to specify this namespace whenever we call a global function, because C++ specifies a rule for *argument-dependent name lookup*, also known as ‘*coenig lookup*’ (ISO/IEC 1998, 3.4.2), which means that if the compiler looks for the actual function `length` that is called by `length(str)`, then it also searches the namespace in which the type of the argument `str` was defined, so if `str` is an instance of the class `String`, then the matching function `length` is found, since both the function and the class are defined in the same namespace `seqan`, see Listing 3. A function that get arguments from different namespaces may cause ambiguities, so we decided not to use several namespaces in SeqAn.

```
namespace seqan
{
    class String { ... };
    size_t length(String &) { ... }
}

seqan::String str;
length(str);           //no namespace qualification needed
```

Listing 3: **Koenig Lookup Example.** We do not need to specify the namespace `seqan` when calling the function `length` from outside the namespace, because it is found by *argument-dependent name lookup*: The compiler searches in the namespace `seqan` for `length`, since the argument `str` was defined there.

### Inheritance and Dynamic Binding

One may think that global functions are not inherited during class derivation, but in fact they are. Supposed that we have two classes `Base` and `Derived` and

the second is derived from the first, then all functions that work for **Base** will also work for **Derived**. Nevertheless we do not make use of this observation in SeqAn, because we apply *template subclassing* instead, see Section 5.3.

Admittedly, global function cannot be virtual, but we will see that *template subclassing* (Section 5.3) can substitute object-oriented polymorphism in many cases, and since we use *static binding* instead of *dynamic binding*, our approach is much more efficient. If dynamic binding is indispensable, one can still use virtual functions and call them via global functions.

### Performance Overhead

In general, the overhead for calling global functions and (non-virtual) member functions is the same. Shims are very small functions and will usually be inlined, so we do not need to expect that shims affect the performance of a program if we apply an optimizing compiler.

## 5.5 Metafunctions

Generic algorithms usually have to know certain types that correspond to their arguments: An algorithm on containers may need to know which *type of values* are stored in the string, or what kind of iterator we need to access it. The usual way in the STL (Austern 1998) is to define the value type of a class like **vector** as a *member typedef* of this class, so it can be retrieved by **vector::value\_type**. Unfortunately member typedef declarations have the same disadvantages as any members: Since they are specified by the class definition, they cannot be changed or added to the class without changing the code of the class, and it is not possible in C++ to define members for built-in types. What we need therefore is a mechanism that returns an output type (e.g. the value type) given an input type (e.g. the string) and that thereby does not rely on members of the input type, but instead uses some kind of global interface. Such task can be performed by *metafunctions*, also known as type traits (Vandevoorde and Josuttis 2002, chapter 15). A *metafunction* is a construct to map some types or constants to other entities like types, constants, functions, or objects at compile time.

We use class templates to implement metafunctions in C++. Listing 4 shows an example for the definition and application of a metafunction **Value** for determining the value type of containers. The code defines **Value** for the **Container** class from Listing 2 and for C++ arrays. The returned type is defined as **Type**, so **Value<T>::Type** is the value type of a container class **T**. The generic algorithm **swapvalues** can be applied for both kinds of data type for swapping the first two values stored in a container.

The metafunctions we propose here constitute a global interface for accessing types, so they also share most of the advantages listed in Section 5.4.1.

Metafunctions can also be used to define additional *dependent types* that are not specified via template arguments. For example, SeqAn offers the metafunction `Size` which specifies the appropriate type for specifying memory amounts (e.g. for storing lengths of containers). This type is by default `size_t`, and it is hardly ever changed by the user, so it is not worth to specify it in another template argument. Nevertheless it is possible to overwrite the default with a new type, like a 64-bit integer (`__int64`) for those container classes that provide extra large storage by defining a new specialization of the metafunction `Size`.

```
template <typename T> class Value;

template <typename TValue, typename TSpec>
class Value < Container<TValue, TSpec> >
{
    typedef TValue Type;
};

template <typename T, size_t I>
class Value < T[I] >
{
    typedef T Type;
};

template <typename T>
void swapvalues(T & container)
{
    typedef typename Value<T>::Type TValue;
    TValue help = container[0];
    container[0] = container[1];
    container[1] = help;
}
```

Listing 4: **Meta Functions Example.** The example class `Container` was defined in Listing 2; it is not part of SeqAn.

Our naming convention states, that the return type of a metafunction is called `Value`. Another application of metafunctions is to define constants that depend on types. If a metafunction returns a constant, than this is called `VALUE`. For example, the metafunction `ValueSize` in SeqAn specifies for alphabet types the number of different values in the alphabet, so for `Dna` the metafunction call `ValueSize<Dna>::VALUE` returns 4.

## 5.6 Further Techniques

### 5.6.1 Metaprogramming

The name ‘metafunctions’ (Section 5.5) stems from the fact that one can consider them as functions of a ‘metaprogramming language’ that is interpreted by the compiler during the compilation process in order to produce the actual C++ code that is to be compiled afterwards. A metaprogram is processed during compile time and therefore does not burden the run time. One can do many things with metaprogramming (e.g. see Gurtovoy and Abrahams 2002), but since this technique is rather complicated and hard to maintain, we decided to use it only in limited circumstances. For example, SeqAn supports the metafunction `Log2` to calculate the integer logarithm of given constants, see Listing 5. This function is very helpful for example to compute the number of bits needed to store a value of a given alphabet type.

```
template < int numerus >
struct Log2
{
    enum { VALUE = Log2<(numerus+1)/2 >::VALUE + 1 };
};

template <> struct Log2<1> { enum { VALUE = 0 }; };
template <> struct Log2<0> { enum { VALUE = 0 }; };
```

Listing 5: **Metaprogram Example.** This metaprogram computes the rounded up logarithm to base 2. Call `Log2<c>::VALUE` to compute  $\lceil \log_2(c) \rceil$  for a constant value `c`.

### 5.6.2 Tag Dispatching

*Tag dispatching* is a programming technique that uses the types of additional function arguments, called ‘tag arguments’, for controlling the *overload resolution*, which is the process of determining the function that is actually executed for a given function call (ISO/IEC 1998, 13.3 and 14.5.8). Since only the types but not the actual instances of the function arguments are relevant for overload resolution, a tag argument need not to have any members. Those classes are called *tag classes*, and we showed in Section 5.3.1, how tag classes are used in *template subclassing* to select a data structure out of several alternatives.

Listing 6 shows how we can use tag classes to switch between different implementation alternatives of algorithms. The third argument of `globalAlignment` acts as tag argument that specifies the algorithm for computing a global alignment. In this example two algorithms are available: `NeedlemanWunsch` and

```
struct NeedlemanWunsch;
struct Hirschberg;

template <typename TAlignment, typename TScoring>
void globalAlignment(TAlignment & ali,
                    TScoring const & scoring,
                    NeedlemanWunsch)
{
    //Needleman-Wunsch algorithm
}

template <typename TAlignment, typename TScoring>
void globalAlignment(TAlignment & ali,
                    TScoring const & scoring,
                    Hirschberg)
{
    //Hirschberg's algorithm
}
```

Listing 6: Tag Dispatching Example.

**Hirschberg.** More tags of this kind supported by SeqAn are listed in Table 11 (in Section 9.5, page 87).

### 5.6.3 Defaults and Shortcuts

There are several ways for a further simplification of the data structures interface. One possibility is to define default arguments for template parameters. For example, one can write `String<char>` instead of `String<char, Alloc<void> >` in SeqAn, since the specialization `Alloc` is the default, see Section 8.1.

Moreover, SeqAn defines several *shortcuts* for frequently used classes. For example, we defined the type `DnaString` as a shortcut for `String<Dna>` and `DnaIterator` for the iterator `Iterator<DnaString>::Type` of `DnaString`. This way it is possible to program basic tasks in SeqAn even without explicitly defining any template arguments.



# Chapter 6

## The Design In Examples

The examples in this chapter will demonstrate the interplay of the programming techniques that we described in Chapter 5.

### 6.1 Example 1: Value Counting

In this example, we want to implement a generic algorithm that counts for each value in the alphabet how often it occurs in a given string. Algorithm 1 shows the general idea of this algorithm. The implementation should at least

	▷ COUNTVALUES ( $a_1 \dots a_m$ )
1	$counter[c] \leftarrow 0$ for each $c \in \Sigma$
2	<b>for</b> $i \leftarrow 1$ <b>to</b> $m$ <b>do</b>
3	$counter[a_i] \leftarrow counter[a_i] + 1$
4	report $counter$

Algorithm 1: **Algorithm for Counting String Values.** The algorithm counts for each value  $c$  of the alphabet  $\Sigma$  the number of occurrences of  $c$  in the string  $a_1 \dots a_m$ .

support the following kinds of string for arbitrary value types:

- Instances of SeqAn string classes `String`.
- C++ standard strings `basic_string`.
- Zero-terminated `char` arrays (C-style strings).

We will need the following functions and metafunctions: The metafunctions `Value` and `ValueSize` to determine the value type and the number of different values this type can get, the function `length` that returns the length of the string, and the function `value` for accessing the string at a given position. Note that all these functions and metafunctions are already defined in SeqAn; we will discuss this how.

### 6.1.1 The Metafunction Value

The metafunction `Value` determines the *value type* of a container. For `SeqAn` strings, the value type is the first template argument, so we define:

```
template <typename T> struct Value;

template <typename TValue, typename TSpec>
struct Value < String<TValue, TSpec> >
{
    typedef TValue Type;
};
```

The class `basic_string` of the C++ standard library has three template arguments, and it defines the member template `value_type`, so we define a *shim* for accessing its value type as follows:

```
template <typename TChar, typename TTraits, typename TAlloc>
struct Value < basic_string<TChar, TTraits, TAlloc> >
{
    typedef basic_string<TChar, TTraits, TAlloc> TString;
    typedef typename TString::value_type Type;
};
```

A metafunction `Value` for arrays was already describes in Listing 4 at page 37. We define specializations both for arrays and pointers:

```
template <typename T, size_t I>
struct Value < T [I] >
{
    typedef T Type;
};

template <typename T>
struct Value < T * >
{
    typedef T Type;
};
```

Moreover, to implement `Value` also for the `const` versions of these types, we specify the following rule, that delegates the metafunction call to the non-`const` version:

```
template <typename T>
struct Value < T const >
{
    typedef typename Value<T>::Type const Type;
};
```



### 6.1.2 The Metafunction ValueSize

The metafunction `ValueSize` returns the number of different values a variable of a given type `T` can get. The default implementation uses the number of bits that are needed to store a value of type `T`: A type that takes  $n$  bits may store at most  $2^n$  different values.

```
template <typename T>
struct ValueSize
{
    enum { VALUE = 1 << (sizeof(T) * 8) };
};
```

Here we use an `enum` declaration; alternatively we could also define a static member constant. Note that this implementation works on 32-bit machines only for types `T` with `sizeof(T) < 4`. However this is no serious restriction in our case, since the algorithm `COUNTVALUES` would not be appropriate anyway for larger alphabets.

For some alphabets which do not use all the bits for representing their values, `SeqAn` overloads `ValueSize` to define sharper bounds, e.g. for the nucleotide alphabet `Dna`:

```
template <>
struct ValueSize < Dna >
{
    enum { VALUE = 4 };
};
```

### 6.1.3 The Functions length

The implementation of `length` for `SeqAn` strings depends on the actual specialization of `String`, see Section 8.3. The length of the general purpose specialization `Alloc` for example results from the difference between the begin and the end of the string, which are both stored as data members in the object, so we may define:

```
template <typename TValue, typename TSpec> inline
typename Size< String< TValue, Alloc<TSpec> > const>::Type
length(String< TValue, Alloc<TSpec> > const & str)
{
    return end(str) - begin(str);
}
```

Note that the return value of `length` is determined by the metafunction `Size`. The default size type is `size_t`, which is sufficient for most applications. For

standard strings we need again a *shim* function that wraps the member function `length` of `basic_string`:

```
template <typename TChar, typename TTraits, typename TAlloc>
inline
typename Size< basic_string<TChar, TTraits, TAlloc> >::Type
length(basic_string<TChar, TTraits, TAlloc> const & str)
{
    return str.length();
}
```

The length of C-style string is determined by searching its zero-termination:

```
template <typename T>
inline typename Size<T *>::Type
length(T * str)
{
    if (!str) return 0;
    T * it = str;
    T zero = T();
    while ( *it != zero) ++it;
    return it - str;
}
```

A ‘zero’ is created by calling the default constructor of `T`. The length of a null pointer is defined to be 0.

#### 6.1.4 The Functions `value`

Since all kinds of string that we consider here support the subscript operator `[ ]` for accessing their values, we get by with a single default implementation of `value`:

```
template <typename TString, typename TPosition>
inline typename Value<TString>::Type
value(TString * str,
      TPosition pos)
{
    return str[pos];
}
```

Note that a class that supports the subscript operator always implements a member function ‘`operator [ ]`’, so in order to avoid the application of member functions (see Section 5.4), generic algorithms should always use the global function `value` instead of square brackets.

### 6.1.5 The Generic Algorithm `countValues`

Now we have all building blocks to implement `COUNTVALUES`. The result is the generic algorithm shown in Listing 7. Here we used the function `ordValue` that transforms a value of type `TValue` into `unsigned int` according to the `ord` function in Section 7.4, which maps the letters in the alphabet to numbers between 0 and the size of the alphabet  $-1$ .

```
template <typename TString>
void countValues(TString const & str)
{
    typedef typename Value<TString>::Type TValue;
    unsigned int const alphabet_size = ValueSize<TValue>::VALUE;
    unsigned int counter[alphabet_size];
    for (unsigned int i = 0; i < alphabet_size; ++i)
    {
        counter[i] = 0;
    }

    for (unsigned int i = 0; i < length(str); ++i)
    {
        TValue c = value(str, i);
        counter[ordValue(c)] += 1;
    }

    /* report counter */
}
```

Listing 7: Generic Algorithm for Counting String Values.

The function `countValues` can be used for all strings that support `Value`, `length`, and `value`, and for all value types that support `ValueSize`. These functions and metafunctions may be defined for all kinds of strings and all reasonable value types, so `COUNTVALUES` has potentially a very large area of application, and it is applicable to string types of different libraries, like `SeqAn` and the C++ standard library, as well as to built-in C-style strings. Thus we call this kind of programming ‘*library-spanning programming*’.

## 6.2 Example 2: Locality-Sensitive Hashing

In Section 12.1.1 we will propose the class `Shape` for storing a (*gapped*) *shape*, which is an ordered set  $s = \langle s_1, \dots, s_q \rangle$  of integers  $s_1 = 1 < s_2 < \dots < s_q$ . The subsequence  $a_{i+s_1}a_{i+s_2} \dots a_{i+s_q}$  of a string  $a = a_1 \dots a_n$  is called the (*gapped*) *q-gram* of  $a$  at position  $0 \leq i \leq n - s_q$ . For a *q-gram*  $b_1 \dots b_q$ , we define the

hash value

$$\text{hash}(b_1 \dots b_q) = \sum_{i=1}^q \text{ord}(b_i) |\Sigma|^{q-i}$$

(see Figure 7), where *ord* returns for each value of the alphabet  $\Sigma$  a unique integer  $\in \{0, \dots, |\Sigma| - 1\}$ , see Section 7.4.

A typical task in Bioinformatics is to compute the hash values for all  $q$ -grams of a given string, e.g. for building up a (gapped)  $q$ -gram index (Section 12.1), or to apply *locality-sensitive hashing* (Indyk and Motwani 1998) for motif finding (Section 11.3.1). Listing 8 shows a generic algorithm that iterates through `str` and computes at each position the hash value by calling the function `hash`.

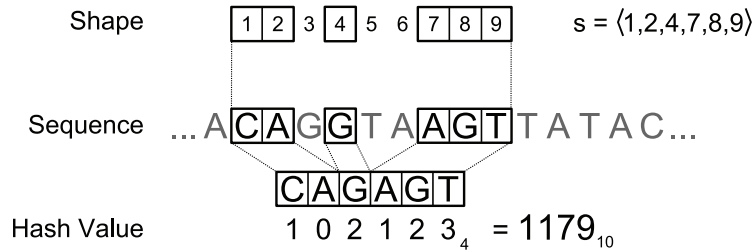


Figure 7: **Locality-Sensitive Hashing.** The example shows the application of the gapped shape  $s = \langle 1, 2, 4, 7, 8, 9 \rangle$ . The hash value of "CAGAGT" is 1179.

There are several ways for storing shapes of different kinds, see Table 22 on page 142. We will now discuss how these shape classes could be implemented in SeqAn.

```
template <typename TShape, typename TString>
void hashAll(TShape & shape,
             TString & str)
{
    typedef typename Iterator<TString>::Type TIterator;

    TIterator it = begin(str);
    TIterator it_end = end(str) - span(shape);

    while (it != it_end)
    {
        unsigned int hash_value = hash(shape, it);
        /* do some things with the hash value */
        ++it;
    }
}
```

Listing 8: **Generic Algorithm for Computing all  $q$ -Gram Hash Values.** The function `span` applied to the shape  $s = \langle s_1, \dots, s_q \rangle$  returns  $s_q - 1$ .

### 6.2.1 The Base Class Shape

We decide to implement all shapes in SeqAn as refinements of the class `Shape`. Each shape has to know the alphabet  $\Sigma$ , so we specify this *value type* in the first template parameter of `Shape`. The actual specialization is selected in the second template parameter `TSpec`:

```
template <typename TValue, typename TSpec = SimpleShape>
class Shape;
```

The default specialization is `SimpleShape`. We will define it in Section 6.2.3. Note that there is no default implementation of `Shape`, i.e. all shapes classes are defined as specializations.

### 6.2.2 Generic Gapped Shapes

The most straightforward implementation of a generic shape  $s = \langle s_1, \dots, s_q \rangle$  stores this sequence in a data member. We use the specialization `GappedShape` of `Shape` for this variant.

```
template <typename TSpec = void>
struct GappedShape;

template <typename TValue, typename TSpec>
class Shape< TValue, GappedShape<TSpec> >
{
public:
    unsigned span;
    String<unsigned int> diffs;
};
```

As a shortcut for this specialization we define:

```
typedef GappedShape<> GenericShape;
```

Since it always holds that  $s_1 = 1$ , we need to store only  $q - 1$  differences  $d_i = s_{i+1} - s_i$  in the container `diffs`. Moreover, we store  $s_q - 1$  in the member variable `span`, which can be retrieved by calling the function of the same name:

```
template <typename TValue, typename TSpec>
inline unsigned int
span(Shape< TValue, TSpec > const & shape)
{
    return shape.span;
};
```

Note that we define `span` in a way that it is also applicable for other specializations of `Shape`. The function `hash` can be implemented as follows:

```
template <typename TValue, typename TSpec, typename TIterator>
inline unsigned int
hash(Shape< TValue, GappedShape<TSpec> > const & shape,
     TIterator it)
{
    unsigned int val = *it;
    for (unsigned int i = 0; i < length(shape.diffs); ++i)
    {
        it += shape.diffs[i];
        val = val * ValueSize<TValue>::VALUE + *it;
    }
    return val;
};
```

### 6.2.3 Ungapped Shapes

The most frequently used shapes are *ungapped*, i.e. shapes  $s = \langle 1, 2, \dots, q \rangle$ . Ungapped shapes can be stored much simpler than gapped shapes:

```
template <typename TValue>
class Shape< TValue, SimpleShape >
{
public:
    unsigned int span;
};
```

That means we need not to store values  $s_i$  or  $d_i$  but only the ‘length’  $q$  of the shape. If we know  $q$  at compile time, then we can specify it in a template parameter and define `span` as a static member:

```
template <unsigned int q = 0>
struct UngappedShape<q>;

template <typename TValue, unsigned int q>
class Shape< TValue, UngappedShape<q> >
{
public:
    static unsigned int const span = q;
};
```

We call this a ‘fixed shape’, since for these shapes the span  $q$  cannot be changed at run time. Since both variants of ungapped shapes are very similar (and we do not need shapes for  $q = 0$ ) we define `SimpleShape` to be a sub-specialization of `UngappedShape` as follows:

```
typedef UngappedShape<0> SimpleShape;
```

This allows us to define functions for both kinds of ungapped shapes at once. Ungapped shapes have the advantage, that the hash value of the  $i$ -th  $q$ -gram can also be computed *incrementally* in constant time from the hash value of the  $i - 1$ -th  $q$ -gram according to the formula:

$$\text{hash}(a_{i+1} \dots a_{i+q}) = \text{hash}(a_i \dots a_{i+q-1})q - a_i|\Sigma|^q + a_{i+q}$$

So we define a function `hashNext` that computes the next hash value, given the previous hash value `prev`:<sup>1</sup>

```
template <typename TValue, unsigned int q, typename TIterator>
inline unsigned int
hashNext(Shape< TValue, UngappedShape<q> > const & shape,
         TIterator it,
         unsigned int prev)
{
    unsigned int val = prev * ValueSize<TValue>::VALUE
                      - *it * shape.fac
                      + *(it + shape.span);
    return val;
};
```

In the above code we store the value  $|\Sigma|^q$  in the member variable `fac`. In the case of fixed shapes this member variable could be a static member constant, so the compiler can apply additional optimizations which makes fixed shapes faster than shapes of variable length  $q$ .

Using `hashNext`, we can define a specialization of `hashAll` for ungapped shapes (Listing 9) that has a higher performance than the generic version in Listing 8.

### 6.2.4 Hardwired Shapes

We argued in the last section, that fixed shapes can be faster than variable shapes, because a shape that is already defined at compile time is better optimized. Therefore we define a specialization `HardwiredShape` of `GappedShape` which encodes a (gapped) shape within template parameters:

```
template <int d1, int d2, int d3, int d4, ...>
struct HardwiredShape;
```

---

<sup>1</sup>Note that `hashNext` in `SeqAn` does not get `prev` as function argument, since the previous hash value is stored in the shape.

```

template <typename TValue, unsigned int q, typename TString>
void hashAll(Shape< TValue, UngappedShape<q> > & shape,
             TString & str)
{
    typedef typename Iterator<TString>::Type TIterator;

    TIterator it = begin(str);
    TIterator it_end = end(str) - span(shape);

    unsigned int hash_value = hash(shape, it);
    /* do some things with the hash value */

    while (++it != it_end)
    {
        unsigned int hash_value = hashNext(shape, it, hash_value);
        /* do some things with the hash value */
    }
}

```

Listing 9: **Special Algorithm for Computing all Hash Values of Ungapped  $q$ -Grams.** The first hash value is computed by `hash`, and the rest incrementally from the previous value by `hashNext`.

For this shape type the function `hash` can be computed by recursive C++ templates, which in effect cause a loop unrolling during the compilation. In practice, a ‘hardwired shape’ achieves a much better performance than the generic gapped shape class from Section 6.2.2.

### 6.2.5 Conclusion

Figure 8 shows the hierarchy of specializations for `Shape`. The ‘left branch’ `Shape`  $\leftarrow$  `Shape<GappedShape>`  $\leftarrow$  `Shape<GappedShape<Hardwiredshape>`  $\leftarrow$  gives an example for the progressive specialization that we described in Section 5.3.1, where the ‘derived’ class determines the `TSpec` slot of its ‘base’ class. In the ‘right branch’, the derivation `Shape<UngappedShape>`  $\leftarrow$  `Shape<SimpleShape>` demonstrates that *template subclassing* is also capable of other kinds of class derivation: The shape class `Shape<SimpleShape>` is created by defining a template specialization of `Shape<UngappedShape<q>`  $\leftarrow$  for  $q = 0$ .

Each specialization of `Shape` has its own purpose: If we want to define the actual shape at run time, then we need `GenericShape` or `SimpleShape` instead of their faster ‘fixed’ variants `HardwiredShape` and `UngappedShape`, see Figure 9. For ungapped shapes, we better use the specializations `SimpleShape` or `UngappedShape` instead of the much slower alternatives `GenericShape` or `HardwiredShape`.



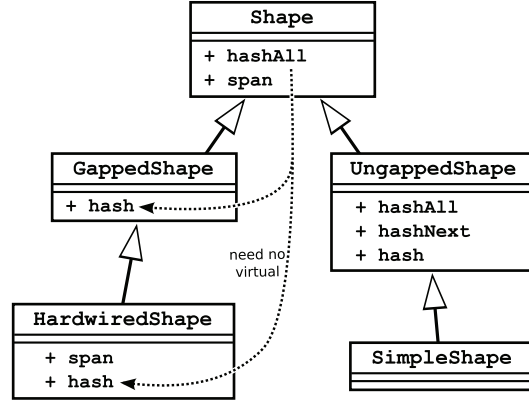


Figure 8: **Specialization Hierarchy of Shape.** The dotted pointer shows that `hashAll` calls the `hash` functions from descendant specializations. This is done without the need of ‘virtual’, since *template subclassing* relies on *static function binding*, i.e. it is known at compile time which function is actually called.

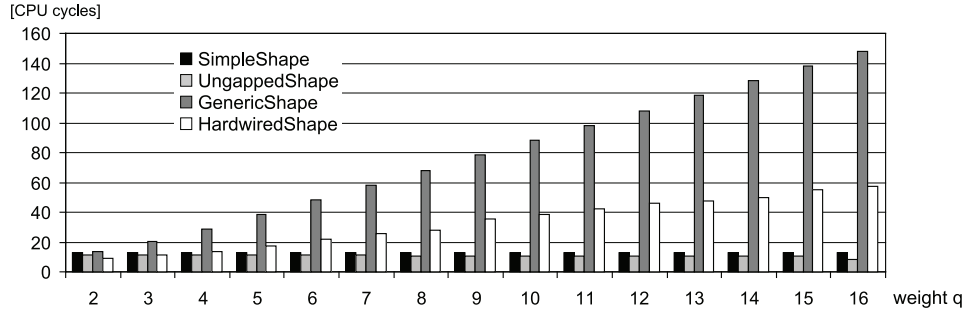


Figure 9: **Runtimes for  $q$ -gram Hashing.** Average runtimes for computing a hash value of an ungapped  $q$ -gram, where `SimpleShape` and `UngappedShape` use the function `hashNext`, and `GenericShape` and `HardwiredShape` the function `hash`. Alphabet size  $|\Sigma| = 4$ . The compiler optimized ‘fixed’ versions `UngappedShape` and `HardwiredShape` take only about 80% and 40% the time of their generic counterparts `SimpleShape` and `GenericShape`, respectively.



# Part III

## Content

*This part gives a detailed overview of the main contents of SeqAn from the algorithmic point of view. Chapter 7 explains basic functionality of the library. Sequence data structures like strings, segments or string sets are discussed in Chapter 8, gapped sequences and sequence alignments in Chapter 9, algorithms for searching patterns or finding motifs in sequences are proposed in the Chapters 10 and 11. The topic of Chapter 12 are string indices, and finally, Chapter 13 proposes the graph data structures and algorithms available in SeqAn. The string indices (Chapter 12) are in large part the work of David Weese, the graph library (Chapter 13) was implemented mainly by Tobias Rausch.*

*Note that this part is no programmers reference manual. The library documentation (see also Section 14.2.2) which is available from the project homepage [www.seqan.de](http://www.seqan.de) and part of the library releases contains all relevant information about that. The documentation also provides tutorials for the library's use.*



# Chapter 7

## Basics

In this chapter we describe basic functionality provided by SeqAn and we introduce some fundamental concepts that we will need in the following chapters. We start with the concept of *containers* of *values* in Section 7.1. The next Section 7.2 concerns memory allocation, and in Section 7.3 we explain the idea of *move* operations. The *alphabet* types provided by SeqAn are introduced in Section 7.4, and *iterators* in Section 7.5. Section 7.6 is about the *conversion* of types, and finally Section 7.7 describes the file input/output functionality in SeqAn.

### 7.1 Containers and Values

A *container* is a data structure that has the purpose to store *values*, i.e. objects of another data type. For example, a data structure `String` that stores the string "ACME" would contain the values 'A', 'C', 'M', and 'E'. Typically, all values stored in a container have the same type, we call it the *value type* of the container. The metafunction `Value` determines the value type for a given container type.

A *pseudo container* is a data structure that in fact does not store instances of the value type, but merely offers the same interface as a 'real' container. Saving memory is the main reason for using pseudo containers: For example a pseudo container `vector<bool>` could store the information about its content in a bit field instead of storing individual `bool` objects in a vector. By doing this, the container will take only one bit per value instead of one byte per value and thus save memory.

The interface of containers does not depend on the way they store the information about their values. This, however, raises questions concerning the value access. A very intuitive way of accessing the values within a container is a function that returns references. A *reference* behaves like the object it refers to but has a different type. This holds in particular for C++ reference types (C++ Standard 8.3.2), e.g. `int&` that is a data type for storing references to

`int` variables. It is also possible to design *proxy classes* that serve as references. Proxy classes are necessary if an access function is applied to a pseudo container, because pseudo containers do not store actual value objects, hence access functions cannot return C++ references, so the references must be emulated by a proxy class. Unfortunately, it is not possible in C++ to define proxy classes that behave correctly as references in all circumstances. For example, proxy objects usually do not fit into the same template subclassing hierarchy (Section 5.3) as the types they refer to, so different function overloads may be called if we use proxy objects instead of the values itself as function arguments. An alternative way to access values within a container are ‘get’-functions like `getValue` that either return a C++ reference to the value or, in case of a pseudo container, a temporary copy of the value. The type returned by a `getValue` can be determined by the metafunction `GetValue`, and the reference type by `Reference`. For example, `GetValue<vector<bool>>` returns `bool` and `Reference<vector<bool>>` a proxy class instead of `bool&` because `vector<bool>` is a pseudo container.

## 7.2 Memory Allocation

Controlling memory allocation is one of the big advantages of C++ compared to other programming languages as for example Java. Depending on the size of objects and the pattern they are allocated during the program execution, certain memory allocation strategies have advantages compared to others. SeqAn supports a variety of memory allocation strategies.

The two functions `allocate` and `deallocate` are used in SeqAn to allocate and deallocate dynamic memory (C++ Standard 3.7.3). Both functions take an allocator as an argument. An *allocator* is an object that is thought as to be ‘responsible’ for allocated memory. The default implementations of `allocate` and `deallocate` completely ignore the allocator but simply call the basic C++ operators `new` and `delete`. Although in principle every kind of object can be used as allocator, typically the object that stores the pointer to the allocated memory is used as allocator. For example, if memory is allocated for an alloc string (see 8.3.1), this string itself acts as allocator. A memory block should be deallocated using the same allocator object as it was allocated for.

The function `allocate` has an optional argument to specify the intended *allocator usage* for the requested memory. The user can thereby specialize `allocate` for different allocator applications. For example, the tag `TagAllocateTemp` specifies that the memory will only be used temporarily, whereas `TagAllocateStorage` indicates that the memory will be used in the long run for storing values of a container.

SeqAn also offers more complex allocators which support the function `clear`. This function deallocates at once all memory blocks that were previously al-

<code>SimpleAlloc</code>	General purpose allocator.
<code>SinglePool</code>	Allocator that pools memory blocks of specific size. Blocks of different sizes are not pooled.
<code>ClassPool</code>	Allocator that pools memory blocks for a specific class. The underlying functionality is the same as for <code>SinglePool</code> .
<code>MultiPool</code>	Allocator that pools memory blocks. Only blocks up to a certain size are pooled. The user can specify the size limit in a template argument.
<code>ChunkPool</code>	Allocator that pools one or more consecutive memory blocks of a specific size.

Table 1: **Allocators.** These specializations of `Allocator` support the `clear` function.

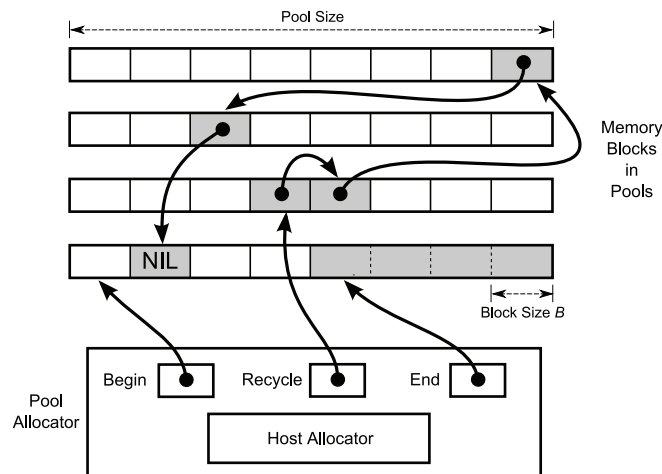


Figure 10: **Pool Allocator.** The `SinglePool` allocator optimizes the allocation of memory blocks of a certain size  $B$  that can be specified in a template argument. A host allocator – by default a `SimpleAlloc` allocator – is used to allocate the pools and requested memory blocks of size different than  $B$ . Not used memory blocks are displayed gray. Released blocks are stored in a linked list starting with the pointer `Recycle`. If a new memory block is requested from the pool, then it is taken from the beginning of this list or, if the list is empty, the block at the end position is used.

located. There are some allocator specializations for different uses predefined (see Table 1). Most of these allocators are pool allocators. A *pool allocator* implements its own memory management: It reserves storage for multiple memory blocks at a time and recycles deallocated blocks, see Figure 10. This reduces the number of expensive `new` and `delete` calls and speeds up the allocation and deallocation, see Figure 11 for timings.

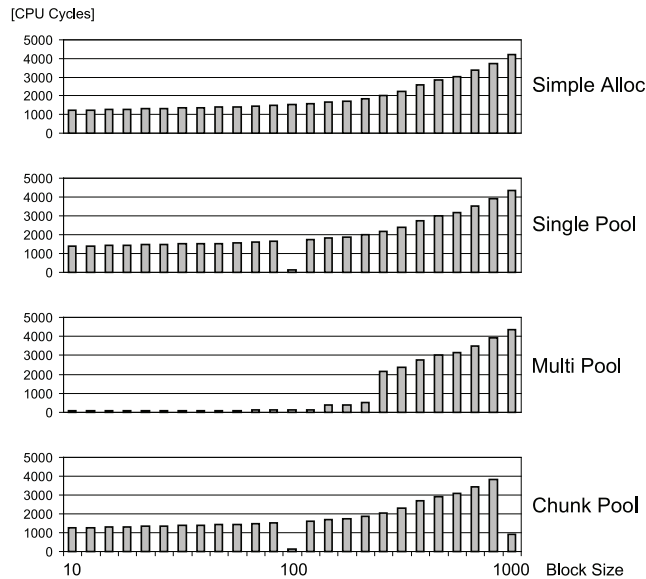


Figure 11: **Allocator Run Times.** The average time for allocating memory blocks of different sizes using (1) `SimpleAlloc` (2) `SinglePool<100>` (3) `MultiPool` and (4) `ChunkPool<100>`. The times for getting memory from `SimpleAlloc` reflects the expense for requesting it directly from the heap manager. Blocks of size 100 (in case of `SinglePool<100>`) or multiples of 100 (in case of `ChunkPool<100>`) are taken from the pool; `MultiPool` pools blocks of size  $\leq 256$ . All other blocks are requested from the heap manager. The figure shows that getting a memory block from a pool takes approximately 8% of the time needed to allocate the same amount of memory from the heap manager.

Note that the C++ standard concept ‘allocator’ (C++ Standard 20.1.5) differs from the SeqAn allocator concept. For example, the C++ standard requires that allocators implement several member functions, whereas the SeqAn library design avoids member functions as possible, see Section 5.4. SeqAn offers the adaptor class `ToStdAllocator` that fulfills the allocator requirements as defined in the C++ standard and wraps the member functions `allocate` and `deallocate` to their global counterparts. One purpose of `ToStdAllocator` is to make standard containers use the SeqAn allocators for retrieving memory.

## 7.3 Move Operations

There is often an opportunity to speed up copy operations, if the source is not needed any more after copying. Therefore, we introduce *move* operations,



i.e. assignments that allow the destruction of the source. For example, if a container stores its values in a dynamically allocated memory block, a move operation may simply pass the memory block of the source object to the target. The source container will be empty after the move operation. Move operations like `moveValue` are alternatives for regular assignment functions like `assignValue`.

```
String<char> str1 = "ACGT";
String<char> str2(str1, Move());
cout << str2;           //output: "ACGT"
cout << length(str1);   //output: 0
```

Listing 10: **Move Constructor Example.**

In many cases, SeqAn also offers special constructors that apply move operations. The *move constructor* differs from the regular copy constructor in an additional tag argument `Move` (see Listing 10).

## 7.4 Alphabets

A value type that can take only a limited number of values is called a (finite) *alphabet*  $\Sigma$ . We can retrieve the number of different values of an alphabet  $|\Sigma|$ , the *alphabet size*, by the metafunction `ValueSize`. Another useful metafunction called ‘`BitsPerValue`’ can be used to determine the number of bits needed to store a value of a given alphabet. Table 2 lists some alphabets predefined in SeqAn. Let  $\Sigma = \{\sigma_0, \dots, \sigma_{|\Sigma|-1}\}$  be an alphabet, then we denote  $ord(\sigma_i) = i$ . This number can be retrieved by calling the function `ordValue`. All predefined alphabets in SeqAn store their values in enumerated integers  $\{0, \dots, \text{ValueSize}-1\}$ , so `ordValue` is for those value types a trivial function.

### 7.4.1 Simple Types

Containers in SeqAn are usually designed as generic data structures that can be instantiated for arbitrary value types. The value type can therefore be any user defined class as well as a simple type. A *simple type* is a type that does not need a constructor to be created, a destructor to be destroyed, and neither a constructor nor an assignment operator to be copied.

Simple objects have the advantage that they can be moved within the computer’s main memory using fast memory manipulation functions. In many cases, containers that work on simple types can therefore be implemented

<b>Dna</b>	Alphabet for storing nucleotides of deoxyribonucleic acid, i.e. ‘A’, ‘C’, ‘G’, and ‘T’.
<b>Dna5</b>	Like <b>Dna</b> , but with an additional value ‘N’ for ‘unknown nucleotide’.
<b>Rna</b>	Alphabet for storing nucleotides of ribonucleic acid, i.e. ‘A’, ‘C’, ‘G’, and ‘U’.
<b>Rna5</b>	Like <b>Rna</b> , but with an additional value ‘N’ for ‘unknown nucleotide’.
<b>Iupac</b>	Iupac code for storing nucleotides of DNA/RNA. The Iupac codes are enumerated in this order: ‘U’= 0, ‘T’, ‘A’, ‘W’, ‘C’, ‘Y’, ‘M’, ‘H’, ‘G’, ‘K’, ‘R’, ‘D’, ‘S’, ‘B’, ‘V’, ‘N’= 15
<b>AminoAcid</b>	Alphabet for storing amino acids.

Table 2: **Alphabets in SeqAn.** The listed characters are the result when a value is converted into `char`.

much faster than generic containers that must copy values one after another using the correct assignment operator or copy constructor.

*POD* (‘plain old data’) types (C++ Standard 3.9) are simple, for example built-in types like `char` or `wchar_t`. A C++ class can also be simple even if it defines constructors, destructors or assignment operators, as long as these functions are not necessary for correctly creating, destroying, or copying instances of this class. All value types listed in Table 2 are simple.

The metafunction `IsSimple` can be used to distinguish between simple and non-simple types in metaprogramming.

## 7.4.2 Array Operations

In SeqAn a set of array operations serve as an abstraction layer to apply divergent handling between simple types and other kinds of types. For example, the general version of the function `arrayCopy` uses a loop to copy a range of objects into a target range, whereas a specialized version of `arrayCopy` for simple types applies the fast memory manipulation function `memmove` (C++ Standard 20.4.6).

## 7.4.3 Alphabet Modifiers

A *modifier* is a class that adapts types in a way that the adapted type is still of the same kind but shows some differences compared to the unmodified type. The alphabet expansion modifier `ModExpand` for example transform an alphabet into another alphabet that contains an additional character, i.e. the *value size* is increased by one. For example `ModifiedAlphabet<TValue, ModExpand<’-’> >` expands the alphabet

TValue by a gap character ‘-’. This alphabet is used in the context of gapped sequences and alignments, see Chapter 9. It is returned by the metafunction `GappedValueType` for value types that do not already contain a gap value. SeqAn also offers string modifiers, see Section 8.5.

## 7.5 Iterators

An *iterator* is an object that is used to browse through the values of a container. The metafunction `Iterator` can be used to determine an appropriate iterator type given a container. Figure 12 shows some examples. Some containers offer several kinds of iterators, which can be selected by an optional argument of `Iterator`. For example, the tag `Standard` can be used to get an iterator type that resembles the C++ standard random access iterator (see C++ Standard 24.1.5). The more elaborated *rooted iterator*, i.e. an iterator that ‘knows’ its container, can be selected by specifying the `Rooted` tag.

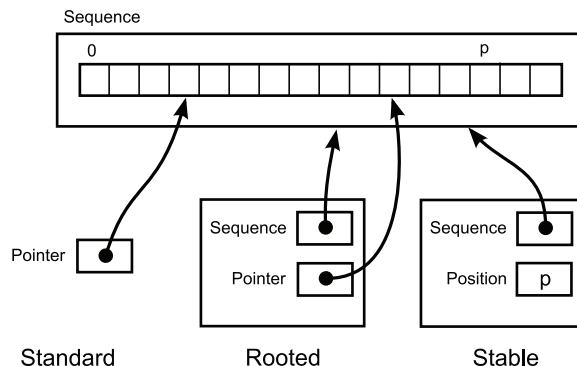


Figure 12: **Iterators for Alloc Strings.** See Section 8.3.1. The standard iterator is just a pointer to a value in the string. The rooted iterator also stores a pointer to the string itself. The stable iterator stored the position instead of a pointer to a value since pointers could be invalid when the alloc string is resized.

Rooted iterators offer some convenience for the user: They offer additional functions like `container` for determining the container on which the iterator works, and they simplify the interface for other functions like `atEnd`, see Listing 11. Moreover, rooted iterators may change the container’s length or capacity, which makes it possible to implement a more intuitive variant of a `remove` algorithm (see C++ Standard 25.2.7). On the other hand, standard iterators can often be implemented simply as pointers, and in practice they are faster than rooted iterators, which typically suffer from an *abstraction penalty*, see Section 5.2. Hence, the default iterator is set to `Standard` for most containers. This default is defined by the metafunction `DefaultIteratorSpec`. While rooted iterators can usually be converted into standard iterators, it is not always possible to convert standard iterators back into rooted iterators,

since standard iterators may lack the information about the container they work on. Therefore, many functions that return iterators like `begin` or `end` return rooted iterators instead of standard iterators; this way, they can be used to set both rooted and standard iterator variables. Alternatively it is possible to specify the returned iterator type explicitly by passing the iterator kind as a tag argument.

```
String<char> str = "ACME";
Iterator<String<char>, Rooted> it;      //a rooted iterator
for (it = begin(str); !atEnd(it); ++it)
{
    cout << *it;
}
```

Listing 11: **Rooted Iterator Example.** Since `it` is a rooted iterator, it supports the unary function `atEnd` that returns `true` if and only if the iterator points behind the end of its container. A standard iterator that does not ‘know’ its container could not support this function.

An iterator is *stable* if it stays valid even if its container is expanded, otherwise it is *unstable*. For example, the standard iterator of alloc strings (Section 8.3.1) – which is a simple pointer to a value in the string – is unstable, since during the expansion of an alloc string, all values are moved to new memory addresses. A typical implementation of stable iterators for strings store the position instead of a pointer to the current value. The `Iterator` metafunction called with the `Stable` tag returns a type for stable iterators.

## 7.6 Conversions

The function `convert` transforms objects from one type `TSource` into another type `TTarget` (see Listing 12). There are two possibilities for doing that: If the object can simply be reinterpreted as an object of type `TTarget`, `convert` returns a `TTarget&` referring to the original object. Otherwise, `convert` returns a temporary (C++ Standard 12.2) object of type `TTarget`. The actual return type can be determined by the metafunction `Convert`.

```
TSource obj;
Convert<TTarget, TSource>::Type obj2 = convert<TTarget>(obj);
```

Listing 12: **Value Conversion.** The program converts an object of type `TSource` into an object of type `TTarget`. `Convert<TTarget, TSource>::Type` is either `TTarget&` or `TTarget`, according as `obj` can be reinterpreted as `TTarget` object or not.

### 7.6.1 Sequence Conversions

A sequence of one value type can be converted into a sequence of another value type, if the two value types are convertible. SeqAn offers three different ways for sequence conversion:

- (1) **Copy conversion.** The source sequence is copied into the target sequence, e.g. during construction, by assignment (operator =), or using the function `assign`.
- (2) **Move conversion.** In some cases, the function `move` can perform an in-place conversion. For example, if source and target sequence are `Alloc` strings (see Section 8.3.1) and if the two value types have the same size, then `move` transfers the value storage of the source to the target string. After that, all values are converted to the new value type.
- (3) **Modifier conversion.** A modifier can ‘emulate’ a sequence with a different value type instead of creating an actual target sequence, see Section 8.5.

## 7.7 File Input/Output

SeqAn supports several ways for reading and writing sequences and alignments in different file formats. Table 3 shows some supported file formats. There are two ways for accessing a file in SeqAn:

- (1) **File Access Functions:** The function `read` loads data from a file and `write` saves data to a file. Both C-style `FILE` handles or C++ stream objects can be used as files. Many file formats like `Fasta` or `Embl` are designed to store multiple records, where each record contains a sequence and some *meta data* about this sequence. The meta data is loaded by `loadMeta` and saved by setting an optional argument of `write`. Function `goNext` skips the current record.

SeqAn also implements stream operators `>>` and `<<`, which are wrapped to `read` and `write` using the `Raw` file format.

- (2) **File Reader Strings:** The most simple way of reading a file that contains sequence data is to use a *file reader string* that emulates a constant string on a given file. It is implemented in the specialization `FileReader` of `String`. File reader strings support almost the complete string interface, including iteration. A file reader string should nevertheless be read sequentially, because random accesses can be very time consuming. Note that the contents of a file reader string cannot be changed.

<b>Raw</b>	The default file format. <b>Raw</b> applied to sequences means that the file content is directly interpreted as a sequence. <b>Raw</b> applied for writing an alignment generates a pretty print.
<b>Fasta</b>	A common file formats for storing sequences or alignments (Pearson and Lipman 1988). Each record consists of a single line starting with ‘>’ that contains meta data, followed by the sequence data.
<b>Embl</b>	The EMBL/Swissprot file format (Stoehr and Cameron 1991) for storing sequences and complex meta data. Each meta data entry starts with a two letter code (see EMBL User Manual 2008, 2008).
<b>Genbank</b>	The GenBank file format (Benson et al. 2008); an alternative notation of EMBL/Swissprot file format for sequence data.
<b>DotDrawing</b>	File format for graphs (see Chapter 13), write only.

Table 3: **Some File Formats.**

The functions **read** and **write** can also be used for loading and storing scoring matrices, i.e. **ScoreMatrix** specialization of class **Score**, see Section 9.3.1. Graphs (see Chapter 13) can be saved to files using the **DotDrawing** file format.

# Chapter 8

## Sequences

### 8.1 Strings

A *sequence* is a container that stores an ordered list of values. The number of these values is called the *length* of the sequence. The values in a sequence are ordered. We define  $i - 1$  to be the *position* of the  $i$ -th value in a sequence, i.e. the first value in the sequence stands at position 0 and the last at position ' $length - 1$ ', as it is standard in C. We call 0 the *begin position* and the length of the sequence the *end position*. Note that the end position is not the position of the last value in the sequence but the position after the last value.

In computer science, a *string* is usually defined as a sequence of characters taken from a finite alphabet, but since there is – concerning the design of data structures – no reason to distinguish between ‘characters’ and other kinds of values, we apply a divergent notation here: We call all sequences ‘*strings*’ that support constant time random access to their values. ‘*Random access*’ means that the string supports a function that returns the value stored at a specified position. Functions of this kind are `value` and `getValue`, as well as the subscript operator `[ ]`. In contrast to that, sequences that do not allow random access to their values, for example *streams*, are not called string. Note that we distinguish between ‘*simple*’ and ‘non-simple’ value types rather than between ‘characters’ and ‘non-characters’ (see Section 7.4.1), but this distinction has no impact on whether a sequence is called a ‘string’ or not. All string classes in SeqAn (with the exception of the external string, see Section 8.3.5) are designed as generic data structures that can be instantiated both for ‘simple’ and ‘non-simple’ value types.

SeqAn implements several string types as specializations of the class `String`. These specializations are described in Section 8.3. There is another specialized class `StringSet` that offers an implementation of ‘strings of strings’, i.e. strings that use again strings as value types. `StringSet` will be described in Section 8.8.

A sequence is *contiguous* if it stores its values consecutively in a single memory

block. Applied to a sequence type `T`, the metafunction `IsContiguous` returns `True` if `T` is contiguous, otherwise `False`. Examples for contiguous strings are the standard library `basic_string` (see C++ Standard 21.3) and simple `char` arrays.

## 8.2 Overflow Strategies

Some sequence types reserve space of storing values in advance. The number of values for which a sequence has reserved space is called the *capacity* of this sequence. The capacity is therefore an upper bound for the length of a sequence. A sequence is called ‘*expandable*’, if its capacity can be changed. All string classes in SeqAn– except of the *array string* (see Section 8.3.2) – are expandable. Changing the capacity can take much time, e.g. expanding an alloc string (see Section 8.3.1) necessitates to copy all values of this string into a new memory block.

<b>Exact</b>	Expand the sequence exactly as far as needed. The capacity is only changed if the current capacity is not large enough.
<b>Generous</b>	Whenever the capacity is exceeded, the new capacity is chosen somewhat large than currently needed. This way, the number of capacity changes is limited in a way that resizing the sequence only takes amortized constant time.
<b>Limit</b>	Instead of changing the capacity, the contents are limited to current capacity. All values that exceed the capacity are lost.
<b>Insist</b>	No capacity check is performed, so the user has to ensure that the container’s capacity is large enough.

Table 4: **Overflow Strategies.**

There are numerous functions in SeqAn that can change the length of a sequence. If the current capacity of a sequence is exceeded by changing the length, we say that the sequence ‘*overflows*’. The *overflow strategy* (see Table 4) determines the behaviour of a sequence in the case of an overflow. The user can specify the overflow strategy by applying a *switch argument*. Otherwise the overflow strategy is determined depending on the sequence class: For functions that are used to *explicitly* change a sequence’s length (like `resize` or `fill`) or capacity (`reserve`), the metafunction `DefaultOverflowExplicit` specifies the default overflow strategy. Functions like `appendValue` or `replace` that primarily serve other needs than changing lengths or capacities may also cause an overflow *implicitly*. For these functions, the metafunction `DefaultOverflowImplicit` is used to determine the default overflow strategy. For example, the alloc string uses **Exact** as explicit and **Generous** as



implicit default expansion strategy. Listing 13 gives an example for the effect of overflow strategies. The overflow strategy **Generous** is used to achieve

```
String<char> str;

//default expansion strategy Exact:
resize(str, 5);
//now the capacity of str is 5

//use expansion strategy Limit:
assign(str, "abcdefghi", Limit());
//only "abcde" was assigned to str.

//default expansion strategy Generous:
append(str, "ABCDEFGG");
//now str == "abcdeABCDEFGG"
```

Listing 13: **Overflow Strategies Example.**

amortized constant costs for appending single values to a string, e.g. the `alloc` string. When the string is expanded, the function `computeGenerousCapacity` is called to compute a new capacity for this string. The default implementation – which can be overwritten by the user – reserves 50% extra space for storing values. This additional memory is used to store values that are appended afterwards. One can easily show that the number of expansions of a string is logarithmic in the number of value appends, and that each value in the string is moved at most three times on average.

## 8.3 String Specializations

We will now describe the different specializations of the class **String** (see Table 5 for an overview).

### 8.3.1 Alloc Strings

`SeqAn` offers two contiguous string specializations: **Alloc** and **Array**. **Alloc** strings use dynamic memory (C++ Standard 3.7.3) for storing values. Expanding the string means therefore that we need to move all values into a new allocated larger memory block. That in turn makes most iterators unstable (see Section 7.5). The amortized costs for appending a value (e.g. using `appendValue`) is constant if the overflow strategy ‘**Generous**’ is used (see Section 8.2).

Since `alloc` strings are a good choice for most applications, **Alloc** is the default string specialization.

<b>Alloc</b>	The default string implementation that can be used for general purposes. The values are stored in a contiguous memory block on the heap. Changing the capacity can be very costly since all values must be copied into a new memory block.
<b>Array</b>	A fast but non-expandable string that stores its values in a member variable and that is therefore best suited for holding small temporary sequences.
<b>Block</b>	A string that stores its content in blocks, so that the capacity of the string can quickly be increased without copying existing values. Though iteration and random access to values are slightly slower than for alloc strings, block strings are a good choice for growing strings and stacks.
<b>Packed</b>	A string that stores as many values in one machine word as possible and that is therefore suitable for storing very large strings in memory. Since each value access takes some bit operations, packed strings are in general slower than other in-memory strings.
<b>External</b>	A string that stores the values in secondary memory (e.g. a hard disk). Only parts of the string are loaded into main memory whenever needed. This way, the total string length is not limited by the machine's physical memory.

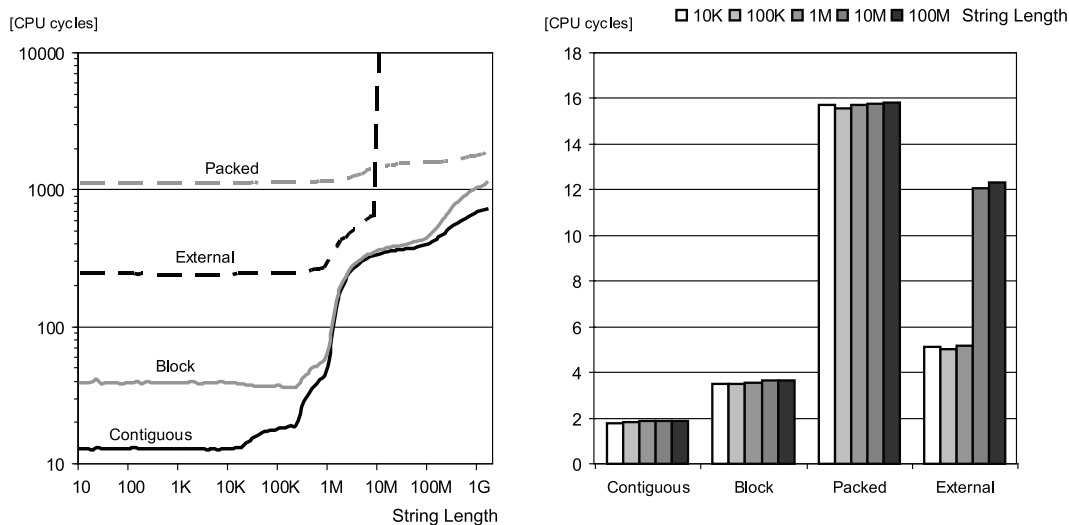
Table 5: **String Specializations.**

Figure 13: **String Value Accessing Run Times.** **Left:** Run times for copying a random value into a random place depending on the length of the strings. **Right:** Run times for iterating a string and moving the whole string one value further.

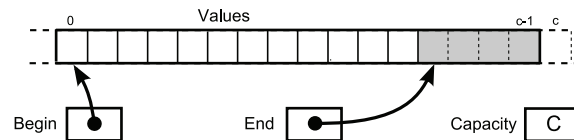


Figure 14: **Contiguous String.** The figure shows an alloc string. The values are stored in a single contiguous piece of memory. The string also stores the capacity of the storage and the begin and end of the currently used part.

### 8.3.2 Array Strings

**Array** strings store values in an array data member. This array has a fixed size, which is specified by a template argument. The advantage of array strings is that no expenses are incurred for allocating dynamic memory if the string is created with static or automatic storage duration (C++ Standard 3.7.2), i.e. the string is stored on the call stack at compile time. This can also speed up the value access, since the call stack is a frequently used part of the memory and has therefore a good chance to stay in the cache. On the other hand, the finite size of the call stack limits the capacity of the string. A typical application for the array strings is to provide quickly limited storage for sequences.

### 8.3.3 Block Strings

A **Block** string stores values in a set of fixed size memory blocks. The location of these blocks is stored in a directory. Block strings are expanded by adding new memory blocks. The advantage – compared to contiguous strings – is, that this can be done without moving values in memory. Block string iterators are therefore always stable (see Section 7.5), and it is uncritical to store pointers to values that are stored in a block string. Random access to a value at a given position in the block string is done in four steps: (1) Determine the number of the block the value is stored, (2) look up in the directory the location of the block, (3) determine the offset at which the value is stored within the block, (4) access the value. If the block size is set to a power of two, step (1) take only one **shift**- and step (3) one **and**-operation. Nevertheless, random accesses to values in block strings are up to three times slower than random accesses to values in contiguous strings, and iterating over a block string takes about two times longer than iterating over a contiguous string (see Figure 13).

The block string is optimized for appending and removing single values at the end of the string. It supports the functions **push** – a synonym for **appendValue** – and **pop**, and it is therefore best suited to be used as stack.

### 8.3.4 Packed Strings

Objects in C++ have at least a size of one *byte* (C++ Standard 1.7), hence each value takes at least eight *bits*. But this is more than actually needed

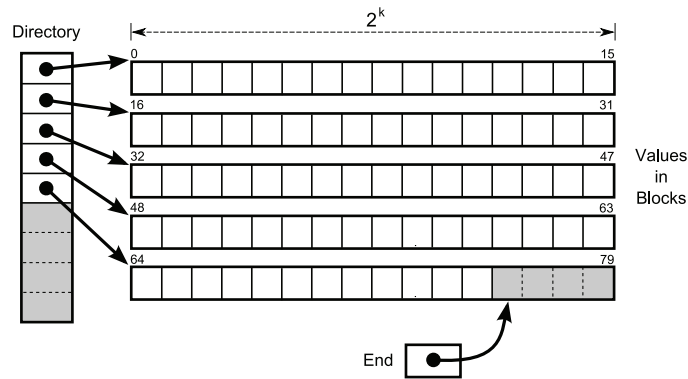


Figure 15: **Block String.** The values are stored in a set of blocks, each of the same size that is a power of two. A directory sequence stores pointers to these blocks. All blocks except of the last one are completely filled.

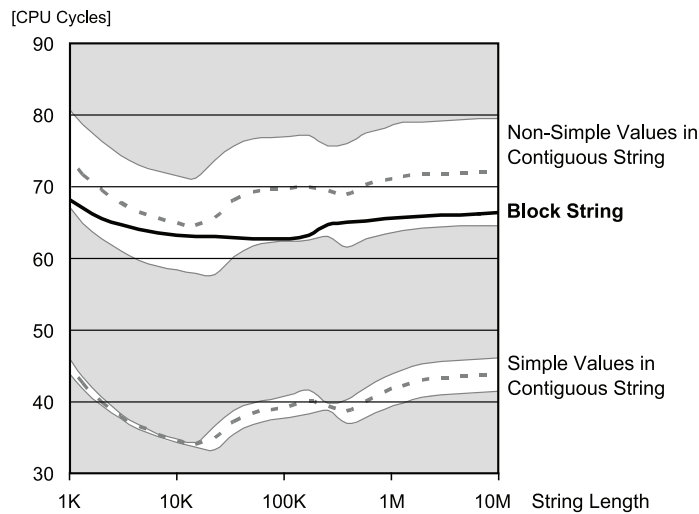


Figure 16: **Append Value Times.** The average time needed to append a single value to a string depending on the string length. Appending a single value to a contiguous string could be very expensive, if this causes an expansion of the string buffer, which means that the complete string must be copied. The two displayed corridors give the upper and lower bound for appending (1) a simple type value or (2) a non-simple type value to a contiguous string, where the `sizeof` of a single value is 1. The upper bound is reached, if the string was expanded during the last appending, and the lower bound, if the next appending would cause a buffer expansion. Block strings need no buffer expansions, so the times for appending a value is less dependent from the kind of object stored in it.

for some value types. For example, the alphabet `Dna` has only four letters ‘A’, ‘C’, ‘G’, and ‘T’, so that a `Dna` value can be encoded in only two bits. The metafunction `BitsPerValue` returns the number of bits needed to store a value.

The `Packed` string stores the values ‘packed’, i.e. each value takes only the minimal number of bytes. For example, the packed string compresses a `Dna` sequences down to a quarter of its ‘unpacked’ size.

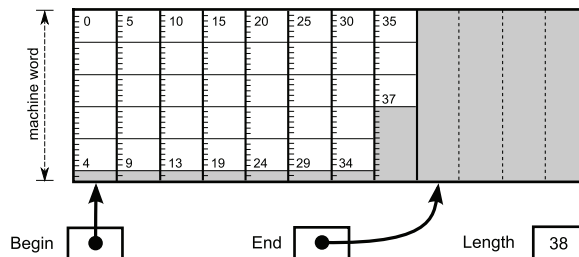


Figure 17: **Packed String.** The string stores as many values per machine word as possible.

In this example, each value takes six bits. The packed string stores five instead of four values per 32 bit machine word, only two bits per machine word are wasted.

However, the handling of packed strings is slower than for all other in-memory string types in `SeqAn`. In practice, random accesses in packed strings are up to two orders of magnitude slower than random accesses in contiguous strings (see Figure 13). This difference has three reasons: (1) Accessing a value in a packed string is much more complicated than accessing a value in a contiguous string, because each access takes multiple operations to filter out the relevant bits, (2) the high complexity of access operations obstruct an efficient optimization by the compiler, and (3) since packed strings are suitable to handle very long sequences, `SeqAn` packed string uses 64 bit position types, but this slows down random accesses on 32 bit machines by a factor of about two. The packed string’s iterator was optimized to speed up accesses, but an iteration still takes eight times longer than iterating an unpacked contiguous string (see Figure 13). For that reason, the application of packed strings is only advisable if the handled sequences are too long to be stored in main memory.

### 8.3.5 External Strings

The `External` string stores its values on external memory, i.e. in a file, with the effect that the main memory size does not limit the sequence length any more. In particular, external strings can be larger than 4 GB even on 32 bit machines, where we need then 64 bit words to store a positions of a value within the string. The file is organized into fixed length blocks, and only some of them are cached in main memory. Both block length and number of cached blocks can be specified in template arguments. When the user accesses a value of an uncached block, the block is loaded into memory, and in return, the

least recently used block in the cache is written back to the file. During an iteration, the external string's iterator prefetches asynchronously the next-inline memory block. This 'trick' speeds up the sequential iteration, but random accesses to values in external strings are very slow, see Figure 13.

## 8.4 Sequence Adaptors

SeqAn also implements complete string interface adaptors – both functions and metafunctions – for data types that are not part of SeqAn. This way, these data types can be accessed in the same way as SeqAn sequences, i.e. string algorithms in SeqAn can be applied to these data types. There are three adaptations:

- (1) For *zero terminated `char` arrays*, also known as 'C-style strings', the 'classical' way for storing strings in the programming language C. For example, the `length` function for C-style strings calls the standard library function `strlen`. The interface applies to arrays of `char` or `char_t` and to `char *` and `wchar_t *` pointers.

Unfortunately, it is not possible to distinguish between C-style strings and `char` pointers, which could also be iterators for C-style strings or other string classes, so if the user calls for example `append` to attach a `char *` to a string `str`, then we could either append a sequence of `char` or a single `char *`. Note that it is not possible to decide this just regarding the value type of `str`, since this could be any type into which either `char` or `char *` can be converted.

Another limitation of C-style strings is that we cannot define all common operators like `operator =` for built-in types like `char *`.

- (2) For the standard library class `basic_string`, that is widely used in C++ programs. For example, the `length` function for `basic_string` string calls the member function `length`.

However, there is a slight problem due to the iterators of `basic_string`. According to the standard, the iterator types are only known via member typedefs `basic_string::iterator` or `basic_string::const_iterator`, but it is not possible in C++ to specialize functions or metafunctions for member typedefs, because the compiler can not deduce the type defining class (in this case `basic_string`) given the defined type (the iterator). For example the following code to define the metafunction `Value` does *not* work, because the template arguments cannot be resolved given the actual iterator type:

```
template <typename TValue, typename TTrait, typename TAlloc>
struct Value<::std:: basic_string<TValue, TTrait, TAlloc>::iterator>
{
    typedef TValue Type;
};
```

We solved this problem by implementing a new iterator specialization `StdIteratorAdaptor` for `basic_string` that wraps the native iterator `basic_string::iterator`.

- (3) Finally, there is also a generic sequence interface that applies to any data type (if there is no further implementation) in a way, that an arbitrary object is regarded as a sequence of these objects of length 1.

## 8.5 Sequence Modifiers

SeqAn supports several *modifiers* (see Section 7.4.3) for strings that allow a different ‘view’ to a given string, see Table 6. Modifiers for strings are implemented in specializations of the class `ModifierString`. The required space of a modifier is constant, i.e. it does not depend on the length of the sequence. Each modifier exports a complete string interface, including an appropriate iterator and segment data types (Section 8.6).

<b>ModReverse</b>	The reverse $a_n \dots a_1$ of the string $a_1 \dots a_n$ .
<b>ModView</b>	<p>Transforms the values of a string <math>a_1 \dots a_n</math> using a custom functional. The type of the functional is specified as template argument of <code>ModView</code>. SeqAn offers the following predefined functionals:</p> <p><b>ModView&lt;FunctorConvert&gt;</b>: Converts the value type.</p> <p><b>ModView&lt;FunctorLowcase&gt;</b>: Converts to lower case characters, e.g. ‘A’ is converted to ‘a’.</p> <p><b>ModView&lt;FunctorUppcase&gt;</b>: Converts to upper case characters, e.g. ‘b’ is converted to ‘B’.</p> <p><b>ModView&lt;FunctorComplement&gt;</b>: Converts nucleotide value <code>Dna</code> or <code>Dna5</code> to their complement, e.g. ‘A’ is converted to ‘T’, ‘C’ to ‘G’, and vice versa.</p>

Table 6: **String Modifiers.**

Modifiers can also be nested, for example the following program shows how to get the reverse complement of a given `Dna` string by applying two modifiers on it:

```
String<Dna> myString = "attacgag";
typedef ModifiedString<String<Dna>, ModComplementDna> TMyComplement;
typedef ModifiedString<TMyComplement, ModReverse> TMyReverseComplement;
TMyReverseComplement myReverseComplement(myString);
std::cout << myReverseComplement << endl; //prints "CTCGTAAT"
```

Since accessing a string through a modifier causes a certain overhead, it could be advisable to convert the string itself – though this has the disadvantage that the original string gets lost. SeqAn therefore offers ‘in-place’ modifier functions `reverseInPlace` and `convertInPlace`. The following example program converts a `Dna` string to its reverse complement:

```
String<Dna> myString = "attacgag";
convertInPlace(myString, FunctorComplement<Dna>());
reverseInPlace(myString);
std::cout << myString << endl; //prints "CTCGTAAT"
```

## 8.6 Segments

A *segment* is a contiguous part of a sequence. The sequence is called the *host* of the segment. SeqAn implements segment data types for infixes, prefixes, and suffixes: A *prefix* is a segment that starts with the first value of the host, a *suffix* is a segment that ends with the last value of the host, and an *infix* is an arbitrary segment.

The segment data structures in SeqAn are pseudo containers: They do not store values itself but a link to their host and the begin and end borders of the segment, see Figure 18. These borders can be set either during the construction of the segment or by functions like `setBegin` or `setEnd`. Changing the content of a segment means to change the content of its host, see Listing 14.

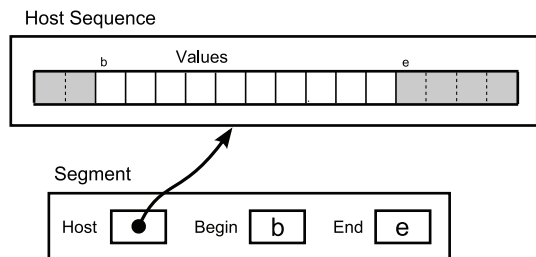


Figure 18: **Segment**. The infix segment stores a pointer to the host and the begin and end position of the subsequence.

The metafunctions `Infix`, `Prefix`, and `Suffix`, respectively, return for a given sequence an appropriate data type for storing the segment. The functions



```
String<char> str = "start_middle_end";
Infix<String<char> > inf = infix(str, 6, 12);
cout << inf;           //output: "middle"
inf = "overwrite";
cout << str;           //output: "start_overwrite_end";
prefix(str, 5) = "XYZ";
cout << str;           //output: "XYZ_overwrite_end";
```

Listing 14: **Segment Example.** This program demonstrates how the content of a string `str` can be changed by assigning new values to segments.

`infix`, `prefix`, and `suffix` create temporary segments that can directly be used to manipulate their host sequence. It is also possible to create segments of segments, but this does not introduce new types: A segment `A` of a segment `B` of a sequence `S` is again a segment of `S`. Note that changing the borders of `B` does not affect the borders of `A`.

## 8.7 Comparators

Let  $A = a_0a_1 \dots a_n$  and  $B = b_0b_1 \dots b_m$  be two sequences. If the value types of `A` and `B` support the ‘equal’ operator `==`, then we can derive an operator `==` for sequences as follows: `A == B` is **true** if  $n = m$  and  $a_i == b_i$  for all  $i$ ; otherwise `A == B` is **false**. If the value types also support the less operator `<`, we define the *lexicographic order* `<` for sequences as follows: `A < B` is **true** if either (1) `A` but not `B` is the empty sequence `""`, or (2)  $a_l < b_l$ , where  $l$  is the length of the longest common prefix of `A` and `B`; otherwise `A < B` is **false**. If `A` and `B` share a common prefix, then their *longest common prefix* is the longest sequence that is both a prefix of `A` and `B`, otherwise it is the empty sequence. If `<` for the value types is a strict total ordering – which is the case for built-in types like `int` or simple alphabets like `Dna` – the lexicographic order is also a strict total ordering. Based on `==` and `<`, we can also define the other common compare operations `!=`, `>`, `<=`, and `>=` as usual.

Each comparison involves a scan of the two sequences for searching the first mismatch between the strings. This could be expensive if the two sequences share a long common prefix. Suppose for example that we want to branch in a program depending on whether `A < B`, `A == B`, or `A > B`, for example:

```
if (A < B)          { /* code for case "A < B" */ }
else if (A > B) { /* code for case "A > B" */ }
else                { /* code for case "A == B" */ }
```

In this case, although only one scan would be enough to decide what case is to be applied, each operator `>` and `<` performs a new comparison. SeqAn offers

lexicals to avoid such unnecessary sequence scans. A *lexical* is an object that stores the result of a comparison. Applying a lexical to the example above leads to the following code:

```
Lexical<> comp(A, B);
if (isLess(comp))      { /* code for case "A < B" */ }
else if (isGreater(comp)) { /* code for case "A > B" */ }
else                   { /* code for case "A == B" */ }
```

The two sequences **A** and **B** are compared during the construction of the lexical `comp`. The result is stored in the lexical and is accessed via the functions `isLess` and `isGreater`.

<b>Owner</b>	The default specialization of <b>StringSet</b> . The sequences in this string set are stored in a string of string data structure. <code>concat</code> returns a special ‘concatenator’ object that simulates the concatenation of all these strings.
<b>Owner&lt;ConcatDirect&gt;</b>	The sequences are stored as parts of a long string. Since the sequences are already concatenated, <code>concat</code> just need to return this string. The string set also stores lengths and starting positions of the strings. Inserting new strings into the set or removing strings from the set is more expensive than for the default <b>Owner</b> specialization, since this involves moving all subsequent sequences in memory.
<b>Dependent&lt;Tight&gt;</b>	This specialization stores sequence pointers consecutively in an array. Another array stores an id value for each sequence. That means that accessing given an id needs a search through the id array.
<b>Dependent&lt;Generous&gt;</b>	The sequence pointers are stored in an array at the position of their ids. If a specific id is not present, the array stores a zero at this position. The advantage of this specialization is that accessing the sequence given its id is very fast. On the other hand, accessing a sequence given its position <i>i</i> can be expensive, since this means we have to find the <i>i</i> -th non-zero value in the array of sequence pointers. The space requirements of a string set object depends on the largest id rather than the number of sequences stored in the set. This could be inefficient for string sets that store a small subset out of a large number of sequences.

Table 7: **StringSet Specializations.**

## 8.8 String Sets

A set of sequences can either be stored in a sequence of sequences, for example in a `String<String<char>>`, or in `StringSet`. One advantage of using `StringSet` is that it supports the function `concat` that returns a *concatenator* of all sequences in the string set. A *concatenator* is an object that represents the concatenation of a set of strings. This way it is possible to build up index data structures for multiple sequences by using the same construction methods as for single sequences, (see Chapter 12). The specialization `Owner<ConcatDirect>` already stores the sequences in a concatenation. The concatenators for all other specializations of `StringSet` are ‘virtual’ sequences, that means their interface ‘simulates’ a concatenation of the sequences, but they do not literally concatenate the sequences into a single sequence. Hence in any case the sequences need not to be copied when a concatenator is created. There are two kinds of `StringSet` specializations in SeqAn: `Owner` and `Dependent`, see Table 7. `Owner` string sets actually store the sequences, whereas `Dependent` string sets just refer to sequences that are stored outside of the string set.

One string can be an element of several `Dependent` string sets. Typical tasks are therefore to find a specific string in a stringset, or to test whether the strings in two string sets are the same. Therefore a mechanism to identify the strings in the string set is needed, and, for performance reasons, this identification should not involve string comparisons. We solved this problem by introducing *ids*, which are by default `unsigned int` values. There are two ways for accessing the sequences in a string set: (1) the function `value` returns a reference to the sequence at a specific *position* within the sequence of sequences, and (2) `valueById` accesses a sequence given its *id*. In the case of `Owner` string sets, *id* and *position* of a string are always the same, but for `Dependent` string sets, the *ids* can differ from the *positions*. For example, if a `Dependent` string sets is used to represent subsets of strings that are stored in `Owner` string sets, one can use the *position* of the string within the `Owner` string set as *id* of the strings.



# Chapter 9

## Alignments

An alignment (see Section 1.2.3) is a compact notation for the similarities and the differences between two or more sequences. To get the similar regions together, the alignment process allows the insertion of *gaps* into the sequences, so we will first discuss in Section 9.1 which data structures for storing *gapped sequences* are provided in SeqAn, before we propose the alignment data structures in Section 9.2. Algorithms for computing (global) alignments are explained in Section 9.4 to 9.6.

### 9.1 Gaps Data Structures

In alignments obviously arises the need to store *gapped sequences*, i.e. sequences that contain gaps between the values. The simplest way to store a gapped sequence is to store it as a usual sequence using a value type that is extended by an extra blank value ‘-’. The gaps in the sequence are then represented by the regions in the sequences that contain blank values. On the other hand, it could be advantageous to store the ‘ungapped’ sequence and the position of the gaps separately for at least three reasons: (1) this way, we can extend arbitrary sequence data structures to gapped sequences without copying the sequence, (2) one sequence – or parts of it – can participate in several alignments, and (3) storing gaps as sequences of blank values could be very expensive for long gaps or for repeated manipulation of the gaps.

The class **Gaps** implements in SeqAn data structures for storing the positions of gaps or, more precisely spoken, ‘gap patterns’. A *gap pattern* of a sequence  $a = a_1a_2 \dots a_n$  is a strictly monotonically increasing function  $p$  that maps the values  $\{1, 2, \dots, n\}$  to values in  $\mathbb{N}$  (see Figure 19). The sequence  $a$  is in general a segment of a larger host sequence  $s$ , which is called the *source* of  $p$ . The *source position* of  $a_i$  is its position within the source sequence  $s$ , e.g. if  $a = s$ , then the source position of  $a_i$  is  $i - 1$ .

For  $p(i) = j$ , we call  $j - 1$  the *view position* of  $a_i$ . A  $j$  that is not a view position of any value in  $a$  is called a *blank*. A maximal run of blanks is called

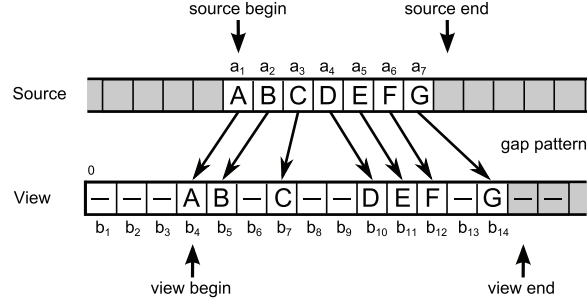


Figure 19: **Gaps Data Structure.** This is an example for a gap pattern function:  $p(1) = 4$ ,  $p(2) = 5$ ,  $p(3) = 7$ , ...

a *gap*. For example the  $L$  blanks  $j + 1, j + 2, \dots, j + L$  between  $p(i) = j$  and  $p(i + 1) = j + 1 + L$  are a gap of length  $L$ . Given a sequence  $a$  and a gap pattern  $p$ , the corresponding *gapped sequence*  $b = b_1 b_2 \dots b_m$  of length  $m = p(n)$  is defined by:

$$b_j = \begin{cases} a_i & \text{if there is an } i \text{ such that } p(i) = j \\ \text{'-'} & \text{(i.e. a blank character) otherwise} \end{cases}$$

The gapped sequence is also called the *view* of the gap pattern. If there are blanks  $0, 1, \dots, p(1) - 1$  at the beginning of  $b$ , they are called the *leading gap* of  $b$ . We call  $p(1) - 1$  the *begin view position* and  $p(n)$  the *end view position* of  $p$ , i.e. they are the begin and end position of the gapped sequence without the leading gap. The gapped sequence  $b$  can also be thought to be followed by an endless number of trailing blanks  $b_{m+1} = b_{m+2} = \dots = \text{'-'}'$ . We call these trailing blanks the '*trailing gap*'. A run of values without blanks between two gaps is called a *non-gap*.

<b>SequenceGaps</b>	A sequence of values including blank signs ' - '.
<b>ArrayGaps</b>	The lengths of gaps and non-gaps are stored in an array.
<b>SumlistGaps</b>	The gap pattern is stored in a two dimensional sum list.

Table 8: **Gaps Specializations Overview.**

SeqAn offers three different specializations for **Gaps**, each of which has certain advantages: (1) **SequenceGapsGaps**, (2) **ArrayGapsGaps**, and (3) **SumlistGapsGaps** (see Table 8). All implementations of gaps data structures offer functions for inserting and deleting gaps, for changing a gap's size, and for converting view positions to source positions and vice versa, which is necessary for example for random access of the values in the gapped sequence given a view position. The differences between the three specializations will be described now.

	SequenceGaps	ArrayGaps	SumlistGaps
inserting a new gap	$O(n)$	$O(g)$	$O(\log(g))$
removing a gap	$O(n)$	$O(g)$	$O(\log(g))$
changing a gap's size	$O(n)$	$O(1)$	$O(\log(g))$
conversion view to source	$O(n)$	$O(g)$	$O(\log(g))$
conversion source to view	$O(n)$	$O(g)$	$O(\log(g))$
accessing the value at a given view position	$O(1)$	$O(g)$	$O(\log(g))$
accessing the value of an iterator	$O(1)$	$O(1)$	$O(1)$
moving an iterator to the next position	$O(1)$	$O(1)$	$O(1)$
moving an iterator to a given view position	$O(1)$	$O(g)$	$O(\log(g))$

Table 9: **Time Consumption of Operations on Gapped Sequences.**  $n$  is length of the gapped sequence,  $g$  is the number of gaps in the gapped sequence.

### 9.1.1 SequenceGaps Specialization

The **SequenceGaps** specialization of **Gaps** is the most obvious implementation of a gaps data structure: It stores a gapped sequence simply as a sequence including ‘-’ blank values. Each gap of size  $L$  take therefore  $L$  blank values, except for the leading and the trailing gap that are known because **SequenceGaps** explicitly saves the begin view position and the length of the sequence in member variables. This special treatment of leading gaps makes sense because the leading and trailing gaps could be very long, especially if the **Gaps** data structure is used to store a line in a multiple sequence alignment as described in Section 9.2. Since **SequenceGaps** stores the source together with the gaps, it cannot refer to an external source sequence. If the ‘ungapped’ source is accessed by calling the function **source**, a temporary source string is created and returned by value.

The specialization **SequenceGaps** stores the view of the gapped sequence directly, so both iterating and random accessing its values are very fast. However, inserting and deleting blanks could be very expensive, because for each such operation the whole part of the view behind the modified position must be moved. A conversion between view and source position is also rather time consuming, because it involves a linear scan through the gapped sequence. Although **SequenceGaps** is rather space efficient for gapped sequences that only contain short gaps, **SequenceGaps** becomes wasteful for very large gaps, because the space needed to store a gap is linear to its size.

### 9.1.2 ArrayGaps Specialization

The **ArrayGaps** specialization of **Gaps** stores the sizes of gaps and gap free parts. For example, for the gapped sequence in Figure 19, it stores the array  $\{3, 2, 1, 1, 2, 3, 1, 1\}$ . Every second number in this array corresponds to a non-gap's size, the rest corresponds to the lengths of the gaps. The first number in the array is the size of the leading gap. The source sequence can either be

stored within the `Gaps` object or separately.

The advantage of `ArrayGaps` is to store the gaps more space efficient than `SequenceGaps` even for very large gaps.

### 9.1.3 SumlistGaps Specialization

The `SumlistGaps` specialization of `Gaps` stores a sequence of pairs, one pair for each non-gap in the gapped sequence. Each pair stores (1) the size of the non-gap and (2) the size of the non-gap plus the size of the preceding gap. For example, for the gapped sequence in Figure 19 on page 80, it stores the following sequence of pairs:  $\{(2, 5), (1, 2), (3, 5), (1, 2)\}$ . These pairs are saved in a two dimensional *sum list*, see Appendix A.2. Given a sum  $S$  and a dimension  $d$ , the sum list  $pair_1, pair_2, \dots, pair_t$  allows a fast search for the first pair  $pair_i$  in the list, for which holds: The  $d$ -th dimension of  $sum_i := pair_1 + pair_2 + \dots + pair_i$  is greater or equal to  $S$ . The search returns both  $pair_i$  and  $sum_i$ . Note that  $sum_i$  is a pair of source position  $sum_i[0]$  and view position  $sum_i[1]$  for the first blank behind the  $i$ -th non-gap. The conversion between view position and source position works as follows:

(1) **source  $\implies$  view:**

Given a source position  $S$ , search the first dimension of the sum list for  $S$  and find  $pair_i$ . The view position  $V$  that corresponds to  $S$  is given by:

$$V = sum_i[1] - (sum_i[0] - S)$$

(2) **view  $\implies$  source:**

Given a view position  $V$ , search the second dimension of the sum list for  $V$  and find  $pair_j$ . The source position  $S$  that corresponds to  $V$  is given by:

$$S = \begin{cases} sum_j[0] - pair_j[0] & \text{if } V \text{ is the position of a blank,} \\ sum_j[0] - (sum_j[1] - V) & \text{otherwise} \end{cases}$$

All operations on a sum list that are relevant for the implementation of `SumlistGaps` – like searching, inserting a pair, removing a pair, and changing a value of a pair – take  $O(\log t)$  time, where  $t$  is the number of pairs in the list. This affects the runtime of operations on `SumlistGaps` gapped sequences: Inserting or removing gaps, changing a gap’s size, conversion between view position and source position, and accessing the value at a given view position take time logarithmic to the number of gaps (see Table 9).



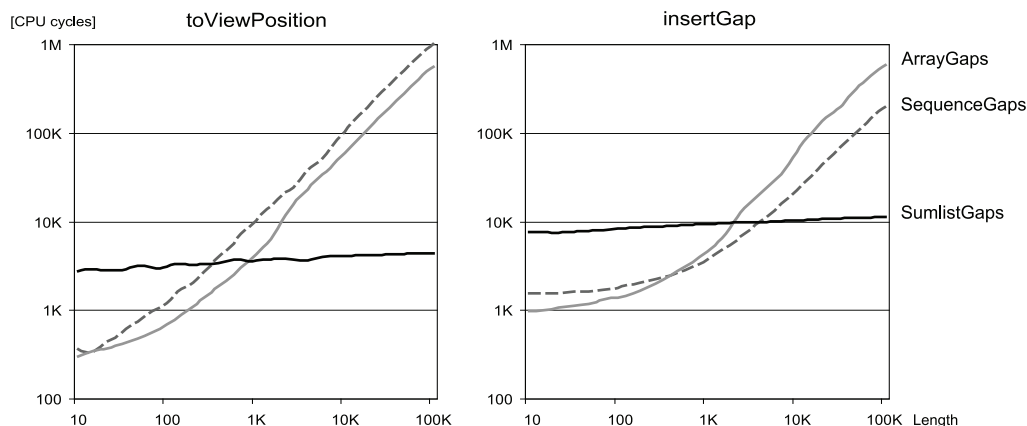


Figure 20: **Gaps Data Structure Run Times.** Run times for (1) converting source to view position of the *last* value and (2) inserting a gap at the *front* of a gapped sequence, depending on the total number of gaps in the gapped sequence. We used a gapped sequence of minimal length, which is *best-case* for **SequenceGaps**.

## 9.2 Alignment Data Structures

Let us write a set of gap patterns  $\{p_1, p_2, \dots, p_k\}$  for  $k \geq 2$  sequences  $a^1, a^2, \dots, a^k$  in a matrix, i.e. the rows are the gap patterns and the columns the view positions, see Figure 21.

$a_1$    - AC - - AAG - CGTAGCA -   gap column  
 $a_2$    - AC - TACGA - G - AGCA -  
 $a_3$    CACTTATG - CC - AG - -

Figure 21: **Example of an Alignment.** An alignment of three sequences  $a^1$ ,  $a^2$ , and  $a^3$ . (Note that the gap column has to be removed in order to get a proper alignment.)

A position  $j$  is called a *gap column*, if it is a blank in all gap patterns. Moreover, if  $j$  is a part of the leading gaps of all  $p_i$ , then  $j$  is a *leading gap column*, and if  $j$  is a part of the trailing gaps of all  $p_i$ , then  $j$  is a *trailing gap column*. The set  $\{p_1, p_2, \dots, p_k\}$  is called an *alignment*  $\mathcal{A}$  of the sequences  $a^1, a^2, \dots, a^k$ , if it contains no gaps columns but, potentially, leading and trailing gap columns. We say that values are ‘aligned’, if they belong to the same column. We can transform an arbitrary set of gap patterns into an alignment by removing all gap columns, e.g. using the function `removeGapCols`. For each proper subset  $M \subset \{1, 2, \dots, k\}$ , a ‘*projection*’  $\mathcal{A}_M$  of the alignment  $\mathcal{A}$  is defined as the set  $\{p_i \mid i \in M\}$  after removing all gap columns. There are two ways for storing alignments in SeqAn: (1) the `Align` data structures, and (2) alignment graphs (see Section 13.2).

The data structure `Align` is implemented as a sequence of `Gaps` objects that store the rows of the alignment and are accessible via the function `rows`. An

alignment can also be considered as sequence of columns, which can be retrieved using the function `cols`. The smallest position that is not a gap column is called the ‘*begin position*’, and the position of the first trailing gap column is called the ‘*end position*’ of the columns sequence. The iterator of the column sequence is implemented as a set of  $k$  iterators, one iterator for each row. This means that iterating the column sequence of an alignment could be costly for alignments that contain many sequences.

Note that `Align` objects support gap columns, so it is the user’s responsibility to remove them if necessary.

## 9.3 Alignment Scoring

### 9.3.1 Scoring Schemes

A *scoring scheme* for alignments is a function that maps alignments to numerical scores like `int` or `double` values. SeqAn supports alignment scoring schemes that are defined (1) by a function  $\alpha$  that scores pairs of aligned values and (2) a function  $\gamma$  for scoring gaps. A gap of size  $l$  scores:

$$\gamma = g_{\text{open}} + g_{\text{extend}} * (l - 1), \quad (9.1)$$

i.e. the first blank in the gap scores  $g_{\text{open}}$ , and  $g_{\text{extend}}$  is added to  $\gamma$  for each further blank in the gap. Usually, we demand  $g_{\text{open}} \leq g_{\text{extend}} \leq 0$ , so  $\gamma$  is a convex function and larger gaps get a discount. If  $g_{\text{open}} = g_{\text{extend}}$ , then we call  $\gamma$  ‘*linear*’, otherwise  $\gamma$  is ‘*affine*’.

<b>Simple</b>	Defines the function $\alpha$ by two values $a_{\text{match}}$ and $a_{\text{mismatch}}$ as follows: $\alpha(x, y) = a_{\text{match}}$ , if $x = y$ , otherwise $\alpha(x, y) = a_{\text{mismatch}}$ . If no other values are specified, this scoring scheme implements the <i>edit distance</i> (Equation 9.4).
<b>ScoreMatrix</b>	Stores the function $\alpha$ in a matrix, which can be loaded from a file. Some common scoring matrices for <code>AminoAcid</code> values by Henikoff and Henikoff (1992) are predefined: <code>Blosum30</code> , <code>Blosum62</code> , and <code>Blosum80</code> . The matrix can be loaded by the function <code>read</code> from a file and stored by the function <code>write</code> to a file, see Section 7.7.
<b>Pam</b>	This is series of common scoring schemes for <code>AminoAcid</code> values by Dayhoff, Schwartz, and Orcutt (1978). A variant by Jones, Taylor, and Thornton (1992) is also available.

Table 10: **Alignment Scoring Schemes.** Specializations for `Score`.

The class `Score` implements some scoring schemes, see Table 10.

If  $\mathcal{A}$  is an alignment of two sequences  $a^1$  and  $a^2$ , then we define the score of  $\mathcal{A}$  by:

$$\text{score}(\mathcal{A}) := \sum_{\text{aligned values } (x, y) \text{ in } \mathcal{A}} \alpha(x, y) + \sum_{\text{gaps } g \text{ in } \mathcal{A}} \gamma(g) \quad (9.2)$$

For alignments  $\mathcal{A}$  of more than two sequences, we define the ‘*sum of pairs*’ *score* to be the sum of the scores of all pairwise sub alignments:

$$\text{score}(\mathcal{A}) := \sum_{i \neq j} \text{score}(\mathcal{A}_{\{i, j\}}) \quad (9.3)$$

### 9.3.2 Sequence Similarity and Sequence Distance

Based on alignment scoring, we define a similarity measure for sequences as follows: The *sequence similarity*  $\text{sim}(a^1, a^2)$  between two sequences  $a^1$  and  $a^2$  with respect to a given scoring scheme *score* is the maximum score alignments between  $a^1$  and  $a^2$  can get, i.e.:

$$\text{sim}(a^1, a^2) := \max(\text{score}(\mathcal{A}) \text{ where } \mathcal{A} \text{ aligns } a^1 \text{ and } a^2)$$

An alignment of  $a^1$  and  $a^2$  with score  $\text{sim}(a^1, a^2)$  is called an *optimal alignment* of  $a^1$  and  $a^2$ . We will describe some algorithms for computing optimal alignments in the next sections.

Note that all scoring schemes in SeqAn are meant to be ‘the higher, the better’, that is alignment algorithms always try to *maximize* scores. However, we can also apply these algorithms for *minimizing* scores simply by maximizing their *negative values*: Let  $\mathcal{A}^*$  be an alignment that scores *minimal* with respect to a scoring scheme *score*, then it is easy to prove that  $\mathcal{A}^*$  also scores *maximal* with respect to the scoring scheme *score'* that is defined by:  $\text{score}'(\mathcal{A}) := -\text{score}(\mathcal{A})$  for each alignment  $\mathcal{A}$ .

A *minimal* alignment score  $\text{score}'(\mathcal{A}^*)$  can be seen as a *distance* between two sequences, so we define the *sequence distance*  $\text{dist}(a^1, a^2)$  of two sequences  $a^1$  and  $a^2$  to be the negative value of their similarity:

$$\text{dist}(a^1, a^2) := -\text{sim}(a^1, a^2)$$

If *score* is defined according to Equation 9.1 and 9.2, and if  $\alpha(x, x) = 0$ ,  $\alpha(x, y) < 0$  for  $x \neq y$ , and  $g_{\text{open}}, g_{\text{extend}} < 0$ , then *dist* is a *metric*, that means it is *positive definite*, *symmetric*, and it holds the *triangle inequality*, i.e. for all sequences  $a^1$ ,  $a^2$ , and  $a^3$ :

$$\begin{aligned} \text{dist}(a^1, a^2) &\geq 0 \text{ and } \text{dist}(a^1, a^2) = 0 \text{ if and only if } a^1 = a^2 \\ \text{dist}(a^1, a^2) &= \text{dist}(a^2, a^1) \\ \text{dist}(a^1, a^3) &\geq \text{dist}(a^1, a^2) + \text{dist}(a^2, a^3) \end{aligned}$$

A well known example of a sequence distance metric called ‘*edit distance*’ or ‘*Levenshtein distance*’ (Levenshtein 1965) is defined by the following scoring scheme:

$$\alpha(x, y) = \begin{cases} 0, & \text{if } x = y \\ -1, & \text{otherwise} \end{cases} \quad g_{\text{open}} = g_{\text{extend}} = -1. \quad (9.4)$$

## 9.4 Alignment Problems Overview

The *alignment problem* means to find an alignment with optimal score in the space of all possible alignments between two or more sequences. There are some variants of alignment problem:

- (1) **Global Alignment Problem.** Alignments between complete sequences are *global alignments*. One way of solving the global alignment problem is *dynamic programming*, which is discussed in the next Section 9.5.
- (2) **Maximum Weight Trace Problem.** Finding an optimal subgraph of a given alignment graph that is compatible with some optimal alignment (i.e. a ‘*trace*’) is called the *maximum weight trace problem*. We will discuss this in Section 13.2.2.
- (3) **Local Alignment Problem.** A *local alignment* between two sequences  $a$  and  $b$  is a global alignment between a substring of  $a$  and substring of  $b$ , and the *local alignment problem* is to find an optimal local alignment. Local aligning is therefore a kind of *motif finding*; we will discuss it in Section 11.1.
- (4) **Semi Global Alignment.** A mix between global and local aligning is the so called ‘*semi global alignment problem*’ that means globally aligning two sequences where some start or end gaps are free. One example for semi global alignment are *overlap alignments*, that is finding the best possible alignment between a suffix of one sequence and a prefix of the other sequence. We will show in Section 9.5.4 how the user can decide in SeqAn, what start gap or end gap should be free when aligning two sequences.

## 9.5 Global Alignments

The *global alignment problem* is defined as follows: For a given set of sequences  $a^1, a^2, \dots, a^k$ , find an alignment  $\mathcal{A}^*$  of these sequences that scores optimal with respect to a given scoring scheme. Finding an optimal alignment of multiple sequences using sum-of-pair scoring (Section 9.3.1) is known to be NP-hard (Wang and Jiang 1994), but for a fixed number  $k$  of sequences, the

alignment problem can be solved in time  $O(n^k)$ , where  $n$  is the length of the sequences.

<b>NeedlemanWunsch</b>	A dynamic programming algorithm by Needleman and Wunsch (1970) for linear gap costs. It aligns two sequences in quadratic time and space.
<b>Gotoh</b>	An extension of the Needleman-Wunsch algorithm that can deal with affine gap costs. (Gotoh 1982)
<b>Hirschberg</b>	An linear space dynamic programming algorithm. (Hirschberg 1975)
<b>MyersHirschberg</b>	A combination of the bit parallel algorithm by Myers (1999) and Hirschberg's algorithm. It aligns two sequences in linear space using edit distance.

Table 11: **Global Alignment Algorithms.** These algorithms base on *dynamic programming*.

Algorithms for finding good global alignments in SeqAn can be accessed by calling the `globalAlignment` function. This function has as arguments (1) an `Align` object, alignment graph object or stream that will be used to store or display the found alignment, (2) a string set that contains the strings to align, if the strings are not already defined by the first argument, (3) a scoring scheme, and (4) a tag that specifies the algorithms that will be used for aligning, see Table 11. The function returns the score of the computed alignment.

### 9.5.1 Needleman-Wunsch Algorithm

In 1970, Needleman and Wunsch introduced an algorithm based on *dynamic programming* (Bellman 1957) to solve the global alignment problem with linear gap costs for two sequences  $a = a_1 \dots a_n$  and  $b = b_1 \dots b_m$ . This algorithm is based on the following observation: Let  $\mathcal{A}_{i,j}$  be an optimal alignment of the prefixes  $a_1 \dots a_i$  and  $b_1 \dots b_j$  for  $i \in \{1, \dots, n\}$  and  $j \in \{1, \dots, m\}$ , and  $M_{i,j} = \text{score}(\mathcal{A}_{i,j})$ . Then the alignment  $\mathcal{A}'_{i,j}$  that we get after deleting the last column  $\mathcal{C}$  from  $\mathcal{A}_{i,j}$  is an optimal alignment, and

$$\text{score}(\mathcal{A}_{i,j}) = \text{score}(\mathcal{A}'_{i,j}) + \text{score}(\mathcal{C}). \quad (9.5)$$

There are three cases: (1)  $\mathcal{C}$  aligns  $a_i$  and  $b_j$ , then  $\text{score}(\mathcal{A}'_{i,j}) = M_{i-1,j-1}$ , or (2)  $\mathcal{C}$  aligns  $a_i$  to a blank, then  $\text{score}(\mathcal{A}'_{i,j}) = M_{i-1,j}$ , or (3)  $\mathcal{C}$  aligns  $b_j$  to a blank, then  $\text{score}(\mathcal{A}'_{i,j}) = M_{i,j-1}$ . Therefore we can compute  $M_{i,j}$  according to the recursion:

$$M_{i,j} \leftarrow \max \begin{cases} M_{i-1,j-1} + \alpha(a_i, b_j) \\ M_{i-1,j} + g \\ M_{i,j-1} + g \end{cases} \quad (9.6)$$

where  $\alpha(a_i, b_j)$  is the score for aligning  $a_i$  and  $b_j$ ,  $g$  is the score for a blank, and  $M_{i,0} = i * p$ , and  $M_{0,j} = j * p$ . Algorithm 2 enumerates all pairs  $(i, j)$  for  $1 \leq i \leq n$  and  $1 \leq j \leq m$  in increasing order for  $i$  and  $j$ , so  $M_{i-1,j-1}$ ,  $M_{i-1,j}$ , and  $M_{i,j-1}$  are already known before  $M_{i,j}$  is computed. FILLMATRIX also protocols in  $T_{i,j}$  which of the three case was applied to compute  $M_{i,j}$ . This information is used in TRACEBACK to construct an optimal alignment. The overall time consumption is  $O(n \times m)$ , and since TRACEBACK requires a complete score matrix  $T$ , the space requirements are also  $O(n \times m)$ .

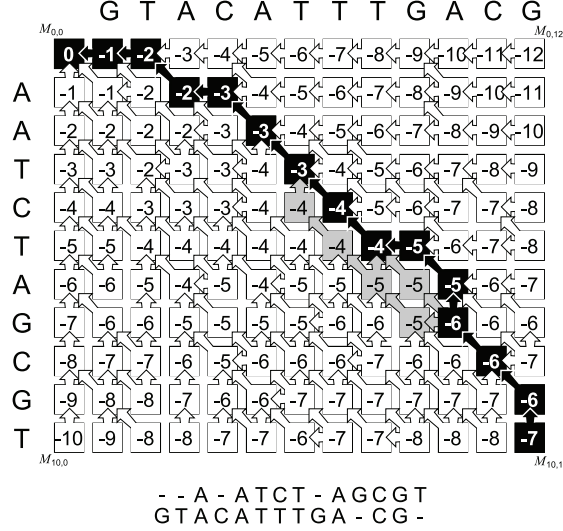


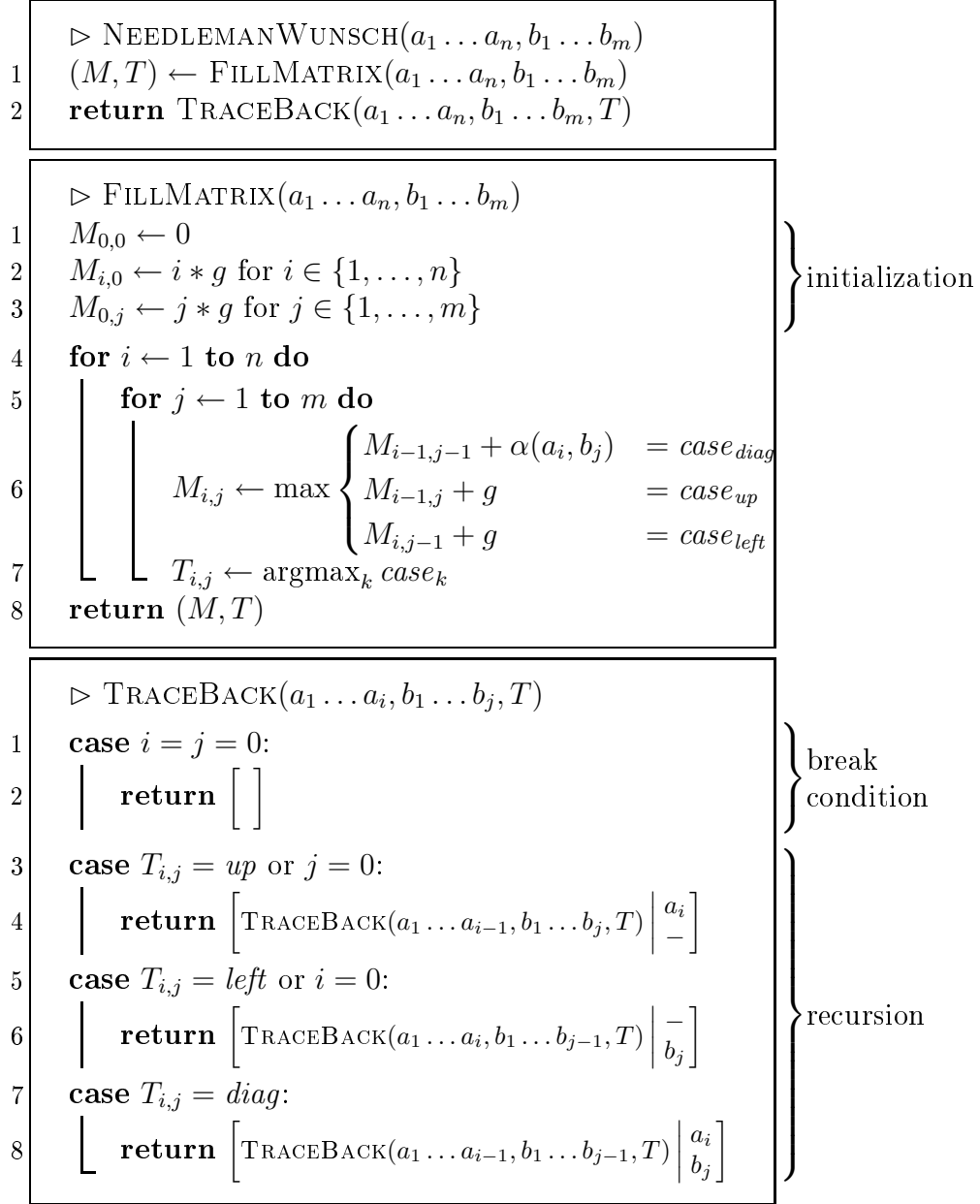
Figure 22: **Needleman-Wunsch Algorithm.** The dynamic programming matrices for aligning  $a = \text{"AATCTAGCGT"}$  and  $b = \text{"GTACATTTGACG"}$ . The values of  $M$  for edit distance scoring, and  $T$  is visualized by pointers to the best predecessors. The optimal alignment on the bottom corresponds to the black printed path, its score  $-7$  is the value in the lower right cell. The gray fields are part of back traces for alternative optimal alignments.

There is no need to store the complete matrix  $M$  during the execution of FILLMATRIX, because at any time at most  $m+1$  cells of  $M$  are needed for proceeding: After computing  $M_{i,j}$ , only the values in  $M_{i,j-1}, \dots, M_{n,j-1}, M_{1,j}, \dots, M_{i,j}$  play a role for the rest of the computation. We can therefore adapt FILLMATRIX to compute the optimal score  $M_{n,m}$  in linear space.

Note that it is possible to generalize the Needleman-Wunsch algorithm in a way that it can compute optimal alignments for arbitrary gap costs in time  $O(n^3)$ , but this algorithm is quite slow and hence rarely applied in practice, so it is not provided by SeqAn.

### 9.5.2 Gotoh's Algorithm

The algorithm by Needleman and Wunsch (Section 9.5.1) does not work for *affine gap costs*, i.e. if gaps of length  $l$  score  $g_{\text{open}} + g_{\text{extend}} * (l - 1)$  with  $g_{\text{open}} \neq g_{\text{extend}}$ . Let  $\mathcal{C}$  be the last column of an alignment  $\mathcal{A}_{i,j} = \mathcal{A}'_{i,j}\mathcal{C}$ . If  $\mathcal{C}$  extends a



Algorithm 2: **Needleman Wunsch Algorithm.**  $\alpha(a_i, b_j)$  is the score for aligning  $a_i$  and  $b_j$ ,  $g$  is the score for a blank.

gap of  $\mathcal{A}'_{i,j}$ , then  $score(\mathcal{A}_{i,j}) = score(\mathcal{A}'_{i,j}) + g_{\text{extend}} \neq score(\mathcal{A}'_{i,j}) + g_{\text{open}}$ , hence equation 9.5 does not hold anymore.

To deal with affine gap costs, Gotoh adapted the algorithm in 1982 such that it computes for each  $i \in \{1, \dots, n\}$  and  $j \in \{1, \dots, m\}$  the following three scores of alignments between  $a_1 \dots a_i$  and  $b_1 \dots b_m$ : (1) the optimal alignment score  $M_{i,j}$ , (2) the best score  $I_{i,j}^a$  of alignments that align  $a_i$  to a blank, and (3) the best score  $I_{i,j}^b$  of alignments that align  $b_j$  to a blank. This can be done by modifying FILLMATRIX as it is shown in Algorithm 3. The asymptotic time and space requirements are the same as for the algorithm by Needleman and Wunsch ( $O(n \times m)$ ) but with larger constant factors, since the algorithm by Gotoh must store and fill three matrices instead of one.

If  $\alpha(a_i, b_j) \geq g_{\text{open}} + g_{\text{extend}}$  for any  $a_i$  and  $b_j$ , then the algorithm can also be implemented by only two matrices  $M$  and  $I$ , where  $I_{i,j}$  stores the best score of alignments  $a_1 \dots a_i$  and  $b_1 \dots b_m$  that either align  $a_i$  or  $b_j$  to a blank. This optimization is currently not provided by SeqAn.

<div style="border: 1px solid black; padding: 10px; min-height: 300px;"> <pre> 1  ▷ FILLMATRIXGOTOH(<math>a_1 \dots a_n, b_1 \dots b_m</math>) 2  <math>M_{0,0} \leftarrow 0</math> 3  <math>M_{i,0} \leftarrow i * g_{\text{extend}}</math> for <math>i \in \{1, \dots, n\}</math> 4  <math>M_{0,j} \leftarrow j * g_{\text{extend}}</math> for <math>j \in \{1, \dots, m\}</math> 5  <math>I_{0,j}^a \leftarrow -\infty</math> for <math>j \in \{1, \dots, m\}</math> 6  <math>I_{i,0}^b \leftarrow -\infty</math> for <math>i \in \{1, \dots, n\}</math> 7  for <math>i \leftarrow 1</math> to <math>n</math> do 8    for <math>j \leftarrow 1</math> to <math>m</math> do 9      <math>I_{i,j}^a \leftarrow \max \begin{cases} M_{i-1,j} + g_{\text{open}} \\ I_{i-1,j}^a + g_{\text{extend}} \end{cases}</math> 10     <math>I_{i,j}^b \leftarrow \max \begin{cases} M_{i,j-1} + g_{\text{open}} \\ I_{i,j-1}^b + g_{\text{extend}} \end{cases}</math> 11     <math>M_{i,j} \leftarrow \max \begin{cases} M_{i-1,j-1} + \alpha(a_i, b_j) &amp; = \text{case}_{diag} \\ I_{i,j}^a &amp; = \text{case}_{up} \\ I_{i,j}^b &amp; = \text{case}_{left} \end{cases}</math> 12   <math>T_{i,j} \leftarrow \text{argmax}_k \text{case}_k</math> 13  return (<math>M, T</math>) </pre> </div>	<div style="display: inline-block; vertical-align: middle;"> <div style="font-size: 2em;">}</div> <div style="display: inline-block; vertical-align: middle; text-align: left;"> initiali- zation </div> </div>
---	---

Algorithm 3: The Recursion of Gotoh's Algorithm.

### 9.5.3 Hirschberg's Algorithm

Unlike the Needleman-Wunsch algorithm (Section 9.5.1) or Gotoh's algorithm (Section 9.5.2), which both take space  $O(n \times m)$  to compute optimal



alignments of two sequences  $a = a_1 \dots a_n$  and  $b = b_1 \dots b_m$ , the algorithm by Hirschberg (1975) only needs linear space. Hirschberg's algorithm (Algorithm 4) applies a divide-and-conquer strategy: It splits both  $a$  and  $b$  into two parts and aligns them separately.

```

1  ▷ HIRSCHBERG( $a_1 \dots a_n, b_1 \dots b_m$ )
2  if  $n < 2$  then
3    |  $\mathcal{A} \leftarrow \text{NEEDLEMANWUNSCH}(a_1 \dots a_n, b_1 \dots b_m)$ 
4  else
5    |  $i \leftarrow \lfloor \frac{n}{2} \rfloor$ 
6    |  $M^{\mathcal{L}} \leftarrow \text{FILLMATRIX}(a_1 \dots a_i, b_1 \dots b_m)$ 
7    |  $M^{\mathcal{R}} \leftarrow \text{FILLMATRIX}(a_n \dots a_{i+1}, b_m \dots b_1)$ 
8    |  $t \leftarrow \text{argmax}_j (M_{i,j}^{\mathcal{L}} + M_{n-i,m-j}^{\mathcal{R}})$ 
9    |  $\mathcal{L} \leftarrow \text{HIRSCHBERG}(a_1 \dots a_i, b_1 \dots b_t)$ 
10   |  $\mathcal{R} \leftarrow \text{HIRSCHBERG}(a_{i+1} \dots a_n, b_{t+1} \dots b_m)$ 
11   |  $\mathcal{A} \leftarrow \mathcal{L}\mathcal{R}$ 
12 return  $\mathcal{A}$ 

```

} find  $t$   
} recursion

Algorithm 4: **Hirschberg's Algorithm.**

The sequence  $a$  is cut at position  $i = \lfloor \frac{n}{2} \rfloor$  (for  $n > 1$ ) into two halves  $a_1 \dots a_i$  and  $a_{i+1} \dots a_n$ . The main problem is to find an appropriate cutting position  $j$  in  $b$ , such that an optimal alignment between  $a$  and  $b$  exists that aligns  $a_1 \dots a_i$  to  $b_1 \dots b_j$  and  $a_{i+1} \dots a_n$  to  $b_{j+1} \dots b_m$ . For any  $j \in \{0, \dots, m\}$ , let  $\mathcal{L}^j$  be an optimal alignment of the prefixes  $a_1 \dots a_i$  and  $b_1 \dots b_j$ , and  $\mathcal{R}^j$  an optimal alignment of the suffixes  $a_{i+1} \dots a_n$  and  $b_{j+1} \dots b_m$ . There is a  $t \in \{0, \dots, m\}$  for which the combination  $\mathcal{A}^t := \mathcal{L}^t \mathcal{R}^t$  is an optimal alignment of  $a$  and  $b$ . For finding a  $t$  that maximizes the total score  $\text{score}(\mathcal{L}^t) + \text{score}(\mathcal{R}^t)$ , we have to compute the scores of all  $\mathcal{L}^j$  and  $\mathcal{R}^j$ . A single call of  $\text{FILLMATRIX}(a_1 \dots a_i, b)$  in line 5 of HIRSCHBERG computes  $M_{i,j}^{\mathcal{L}} = \text{score}(\mathcal{L}^j)$  for all  $j$ . The scores of the  $\mathcal{R}^j$  are computed similarly in line 6 by passing the *reverses* of  $a_{i+1} \dots a_n$  and  $b$  to  $\text{FILLMATRIX}$ : The entry  $M_{n-i,m-j}^{\mathcal{R}}$  of the computed matrix  $M^{\mathcal{R}}$  is the optimal score for aligning  $a_n \dots a_{i+1}$  and  $b_m \dots b_{j+1}$ , which is the same as the best score for aligning  $a_{i+1} \dots a_n$  and  $b_{j+1} \dots b_m$ .  $\text{FILLMATRIX}$  only takes linear space for computing the scores needed. Hence, the total space requirement of HIRSCHBERG is  $O(n + m)$ . It is easy to prove by induction that HIRSCHBERG takes time  $O(n \times m)$ .

The implementation of Hirschberg's algorithm in SeqAn combines it with Gotoh's algorithm (Section 9.5.2) for sequence alignments using affine gap cost schemes.

The alignment algorithm **MyersHirschberg**, which is the fastest algorithm in SeqAn for global sequence alignment (see Table 30), can be used when aligning

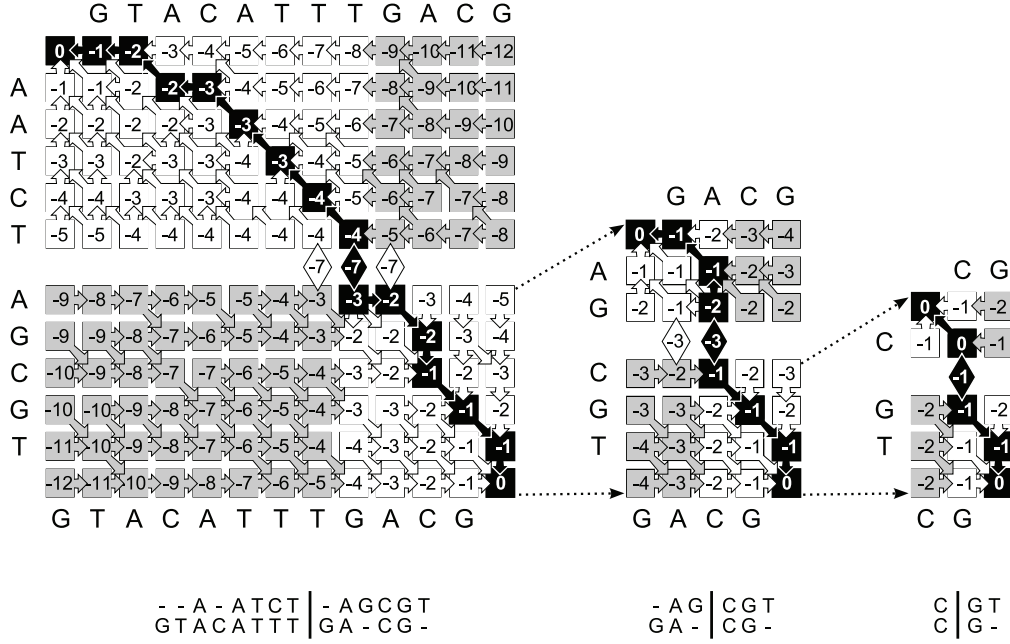


Figure 23: **Hirschberg's Algorithm.** This figure visualizes three recursion steps when aligning  $a = \text{"AATCTAGCGT"}$  and  $b = \text{"GTACATTGACG"}$ . The black printed path corresponds to the alignment that is about to be computed. The gray cells need not to be recomputed during the next recursion step.

two sequences using *edit distance* scoring. This variant of Hirschberg's algorithm uses Myers' bitvector algorithm (see Section 10.3.2) instead of FILLMATRIX for computing the scores for  $\mathcal{L}^j$  and  $\mathcal{R}^j$ .

### 9.5.4 Aligning with Free Start or End Gaps

After some simple modifications, both the Needleman-Wunsch algorithm and Gotoh's algorithm can also be used to compute alignments with free *start gaps* or *end gaps*. A *start gap* contains a blank that is aligned to  $a_1$  or  $b_1$ , and an *end gap* contains a blank that is aligned to  $a_n$  or  $b_m$ . Gap scores are usually non-positive values, and we call a gap *free*, if it scores 0.

Start gaps in  $a$  become free when  $M_{i,0}$  are set to 0 for all  $i \in \{1, \dots, n\}$  (FILLMATRIX, line 2). For free start gaps in  $b$ , we set  $M_{0,j} = 0$  for  $j \in \{1, \dots, m\}$  (FILLMATRIX, line 3).

Let  $i_{\max} = \operatorname{argmax}_{i \in \{1, \dots, n\}} M_{i,m}$  and  $j_{\max} = \operatorname{argmax}_{j \in \{1, \dots, m\}} M_{n,j}$ . The algorithm  $\text{TRACEBACK}(a_1 \dots a_{i_{\max}}, b)$  computes an optimal alignment  $\mathcal{A}_{i_{\max}}$  of  $a_1 \dots a_{i_{\max}}$  and  $b$ . If end gaps in  $a$  – but not in  $b$  – are free, then this is also the best alignment of  $a$  and  $b$ . For free end gaps in  $b$  – but not in  $a$  –, the function call  $\text{TRACEBACK}(a, b_1 \dots b_{j_{\max}})$  returns the optimal alignment  $\mathcal{A}_{j_{\max}}$  of  $a$  and  $b$ . If end gaps are free both in  $a$  and  $b$ , then either  $\mathcal{A}_{i_{\max}}$  or  $\mathcal{A}_{j_{\max}}$  is optimal,

whichever is better.

The class `AlignConfig` can be used to specify, which start gap or end gap are free when calling `globalAlignment`. `AlignConfig` has four `bool` template arguments; `true` means ‘gap is free’. Listing 15 shows an example for using `AlignConfig`.

```
StringSet<CharString> string_set;
appendValue(string_set, a);
appendValue(string_set, b);
Align<CharString> alignment(string_set);
globalAlignment(alignment,
                Score<int>(),
                AlignConfig<false, false, true, false>(),
                NeedlemanWunsch());
```

Listing 15: **Example for Using `AlignConfig`**. The two sequences `a` and `b` are aligned, end gaps for `b` are free.

### 9.5.5 Progressive Alignment

SeqAn also offers a progressive heuristic for finding good alignments between more than two sequences (Rausch, Emde, Weese, Döring, Notredame, and Reinert 2008). We already described the idea of this algorithm in Section 1.2.3 when we discussed the software tool CLUSTAL W (Thompson et al. 1994): The sequences  $a^1, \dots, a^d$  are aligned step by step following a binary *guide tree*  $\mathcal{T}$ , which is constructed by a *hierarchical clustering* algorithm (line 2 of PROGRESSIVEALIGN), on the basis of the pairwise distances between the sequences. SeqAn supports *agglomerative clustering* (complete linkage, single linkage and UPGMA; see e.g. Sneath and Sokal 1973) and *neighbor-joining* (Saitou and Nei 1987). FOLLOWGUIDETREE aligns the sequences following the guide tree from the leaves to the root. At each vertex  $v$ , the alignments  $\mathcal{A}^l$  and  $\mathcal{A}^r$  from both children of  $v$  are aligned (line 8), where we conceive  $\mathcal{A}^l$  and  $\mathcal{A}^r$  as *sequences of columns*, so they can be aligned by any pairwise global sequence alignment algorithm like Needleman-Wunsch (see Section 9.5.1). The score  $\alpha(c^l, c^r)$  for aligning two alignment columns  $c^l$  and  $c^r$  is defined as the (weighted) sum of scores  $\alpha(a^l, a^r)$ , where  $a^l \in c^l$  and  $a^r \in c^r$  (*‘sum of pairs score’*). Inserting a gap into  $\mathcal{A}^l$  or  $\mathcal{A}^r$  means to insert a gap column, i.e. inserting a gap into all sequences of  $\mathcal{A}^l$  or  $\mathcal{A}^r$ , respectively.

The progressive alignment idea was also used in the software tool ‘T-Coffee’ by Notredame et al. (2000) that uses a much more elaborated scoring function  $\alpha$ , which is computed from a given set  $\mathcal{C}$  of pairwise local or global alignments between the sequences  $a^1, \dots, a^d$ . T-Coffee first defines for any pair of characters  $b$  and  $b'$  that stem from different sequences  $a^i$  and  $a^j$  an *individual* score

```

    ▷ PROGRESSIVEALIGN( $a^1, \dots, a^d$ )
1   $D[i, j] \leftarrow \text{dist}(a^i, a^j)$  for all  $i, j \in \{1, \dots, d\}$ 
2   $\mathcal{T} \leftarrow \text{CLUSTERING}(D)$ 
3   $\mathcal{A} \leftarrow \text{FOLLOWGUIDETREE}(a^1, \dots, a^d, \mathcal{T})$ 
4  return  $\mathcal{A}$ 

    ▷ FOLLOWGUIDETREE( $a^1, \dots, a^d, \mathcal{T}$ )
1   $v \leftarrow \text{root of } \mathcal{T}$ 
2  if  $v$  is leaf then
3  |    $\mathcal{A} \leftarrow \text{the sequence } a^i \text{ on } v$ 
4  else
5  |    $\mathcal{T}^l, \mathcal{T}^r \leftarrow \text{left and right subtrees below } v$ 
6  |    $\mathcal{A}^l \leftarrow \text{FOLLOWGUIDETREE}(a^1, \dots, a^d, \mathcal{T}^l)$ 
7  |    $\mathcal{A}^r \leftarrow \text{FOLLOWGUIDETREE}(a^1, \dots, a^d, \mathcal{T}^r)$ 
8  |    $\mathcal{A} \leftarrow \text{align } \mathcal{A}^l \text{ and } \mathcal{A}^r$ 
9  return  $\mathcal{A}$ 

```

Algorithm 5: **Progressive Alignment.** We omit the function CLUSTERING that applies a clustering algorithm to compute the guide tree  $\mathcal{T}$  for a given distance matrix  $D$ .

$\alpha(b, b')$  that depends on the total score of alignments  $\mathcal{A} \in \mathcal{C}$  in which  $b$  and  $b'$  are aligned. In a second step, T-Coffee uses a method called ‘*triplet extension*’, that applies the following rule: If two values  $b$  and  $b'$  are aligned in  $\mathcal{A} \in \mathcal{C}$ , and  $b'$  and  $b''$  are aligned in another alignment  $\mathcal{A}' \in \mathcal{C}$ , then we reinforce the score  $\alpha(b, b'')$  that we get for aligning  $b$  and  $b''$ . The triplet extension helps to find an agreement between the pairwise alignments  $\in \mathcal{C}$ , and this results in much better multiple alignments.

The implementation of T-Coffee in SeqAn stores these values  $\alpha$  as weights on the edges of an alignment graph (Section 13.2) between  $a^1, \dots, a^d$ , and the alignment problem can then be defined as a *maximum weight trace problem* (Section 13.2.2).

## 9.6 Chaining

We saw in Section 9.5 that computing the best alignment between two sequences using dynamic programming takes quadratic time, and the alignment of  $d > 2$  sequences even takes exponential time in  $d$ . Fortunately there are faster heuristics for finding good – but not necessarily optimal – alignments. One way is to search for highly similar substrings, so called *seeds*, and to combine them in a process called *chaining*. A chain of seeds then can be used as a backbone for a *banded alignment* of the two sequences, see Section 9.6.4. We

demonstrate this principle in all details for the algorithm LAGAN in Section 15. This section concerns about chaining seeds to *global* alignments. Similar techniques could also be used to get good *local* alignments, as we will see in Section 11.2.2. How to find seeds will be discussed in Chapter 11.

### 9.6.1 Seeds

Basically, a seed  $\mathcal{S}$  is a set of non-empty segments  $s^1, \dots, s^d$  of sequences  $a^1, \dots, a^d$ , where  $d \geq 2$  is called the *dimension* of  $\mathcal{S}$ . We call  $left_i(\mathcal{S})$  the begin position of  $s^i$  and  $right_i(\mathcal{S})$  the end position of the segment  $s^i$  for  $i \in \{1, \dots, d\}$ . According to the conventions stated in Section 8.1, the position of the value  $a_i$  in a sequence  $a_1 \dots a_m$  is  $i - 1$ , and the begin position of a segment  $a_{left} \dots a_{right}$  is  $left - 1$  and the end position  $right$ .

SeqAn offers a class `Seed` for storing seeds; the specializations of this class are listed in Table 12. All seed types implement the functions `leftPosition` and `rightPosition` to access the begin and end positions of their segments, and the functions `setLeftPosition` and `setRightPosition` to set them. Moreover, each seed  $\mathcal{S}$  stores the score  $weight(\mathcal{S})$  of an optimal alignment between its segments, which can be retrieved by the function `weight` and set by the function `setWeight`. Chaining only requires information about the dimension, borders, and scores of the seeds, i.e. we need not to know the complete alignments.

<b>SimpleSeed</b>	A seed of dimension $d = 2$ . This is the preferred seed type for seed merging, extending, and local chaining algorithms, see Section 11.2. The default specialization of <code>Seed</code> .
<b>MultiSeed</b>	A seed type of arbitrary dimension $d \geq 2$ that was designed for global chaining.

Table 12: **Specializations of class `Seed`.**

### 9.6.2 Generic Chaining

Let in the following the seed dimension  $d$  be fixed. We say that  $\mathcal{S}_j$  can be *appended* to another seed  $\mathcal{S}_k$ , if  $\mathcal{S}_j$  is ‘right of’  $\mathcal{S}_k$ , that is if  $right_i(\mathcal{S}_k) \leq left_i(\mathcal{S}_j)$  for all  $i \in \{1, \dots, d\}$ . Given a set of seeds  $\{\mathcal{S}_1, \dots, \mathcal{S}_n\}$ , we define the *top seed*  $\mathcal{S}_0$  to be the seed with  $left_i(\mathcal{S}_0) = right_i(\mathcal{S}_0) = 0$  for all  $i \in \{1, \dots, d\}$  and  $weight(\mathcal{S}_0) = 0$ . The *bottom seed*  $\mathcal{S}_{n+1}$  is defined by  $left_i(\mathcal{S}_{n+1}) = right_i(\mathcal{S}_{n+1}) = \max_j \{right_i(\mathcal{S}_j)\}$  and  $weight(\mathcal{S}_{n+1}) = 0$ . Note that all seeds can be appended to  $\mathcal{S}_0$  and that  $\mathcal{S}_{n+1}$  can be appended to all seeds. An ordered set  $\mathcal{C} = \mathcal{S}_{j_1}, \mathcal{S}_{j_2}, \dots, \mathcal{S}_{j_k}$  of seeds is called a *chain*, if  $\mathcal{S}_{j_{i+1}}$

can be appended to  $\mathcal{S}_{j_i}$  for each  $i \in \{1, \dots, k-1\}$ . The *score* of a chain is defined by:

$$\text{score}(\mathcal{C}) = \sum_{i=1}^k \text{weight}(\mathcal{S}_{j_i}) + \sum_{i=1}^{k-1} \text{gap\_score}(\mathcal{S}_{j_i}, \mathcal{S}_{j_{i+1}}),$$

where  $\text{gap\_score}(\mathcal{S}_{j_i}, \mathcal{S}_{j_{i+1}})$  is the (usually non-positive) score for appending  $\mathcal{S}_{j_{i+1}}$  to  $\mathcal{S}_{j_i}$ .

<pre> ▷ GENERICCHAINING(<math>\mathcal{S}_1, \dots, \mathcal{S}_n</math>) 1  sort <math>\mathcal{S}_1, \dots, \mathcal{S}_n</math> in increasing order of <math>\text{right}_1(\mathcal{S}_i)</math> 2  compute top seed <math>\mathcal{S}_0</math> and bottom seed <math>\mathcal{S}_{n+1}</math> 3  for <math>j \leftarrow 1</math> to <math>n+1</math> do 4      <math>M_j \leftarrow \text{gap\_score}(\mathcal{S}_0, \mathcal{S}_j) + \text{weight}(\mathcal{S}_j)</math> 5      <math>T_j \leftarrow 0</math> 6      for <math>k \leftarrow 1</math> to <math>j-1</math> do 7          if <math>\text{right}_i(\mathcal{S}_k) \leq \text{left}_i(\mathcal{S}_j)</math> for all <math>i \in \{2, \dots, d\}</math> then 8              <math>\text{score} \leftarrow M_k + \text{gap\_score}(\mathcal{S}_k, \mathcal{S}_j) + \text{weight}(\mathcal{S}_j)</math> 9              if <math>\text{score} &gt; M_j</math> then 10                 <math>M_j \leftarrow \text{score}</math> 11                 <math>T_j \leftarrow k</math> 12  return CHAINTRACEBACK(<math>\mathcal{S}_0, \dots, \mathcal{S}_{n+1}, n+1, T</math>) </pre>	}	compute best prede- cessor for $\mathcal{S}_j$
---	---	--

<pre> ▷ CHAINTRACEBACK(<math>\mathcal{S}_0, \dots, \mathcal{S}_{n+1}, j, T</math>) 1  if <math>j = 0</math> then 2      return <math>\mathcal{S}_0</math> 3  else 4      return CHAINTRACEBACK(<math>\mathcal{S}_0, \dots, \mathcal{S}_{n+1}, T_j, T</math>), <math>\mathcal{S}_j</math> </pre>
---

Algorithm 6: **Generic Chaining Algorithm.**  $\mathcal{S}_1, \dots, \mathcal{S}_n$  is a set of  $d$ -dimensional seeds,  $d \geq 2$ . The algorithm computes a maximal global chain; its score is stored in  $M_{n+1}$ .

The *global chaining problem* is to find a maximal scoring chain  $\mathcal{C}$  that starts with  $\mathcal{S}_0$  and ends with  $\mathcal{S}_{n+1}$ . GENERICCHAINING (Algorithm 6) solves this problem in time  $O(dn^2)$  by *dynamic programming*. The algorithm computes for each seed  $\mathcal{S}_j$  the ‘predecessor’  $\mathcal{S}_k$  for which the chain  $\mathcal{S}_0, \dots, \mathcal{S}_k, \mathcal{S}_j$  gets the optimal score. The score of this chain is stored in  $M_j$  and the index  $k$  of the predecessor is stored in  $T_j$ . The best *global* chain is reconstructed in CHAINTRACEBACK by following  $T$  starting from  $\mathcal{S}_{n+1}$ . The algorithm applies a *sweep line* technique (Shamos and Hoey 1976) by sorting the seeds in line 1 of Algorithm 6. This guarantees that the seed  $\mathcal{S}_j$  can only be appended to

seeds  $\mathcal{S}_k$  with  $k < j$ , hence  $M_k$  was already computed before it is used in line 8 to compute  $M_j$ .

The function `globalChaining` implements chaining algorithms in SeqAn. The actual algorithm is specified by a tag; see Listing 16 for an example.

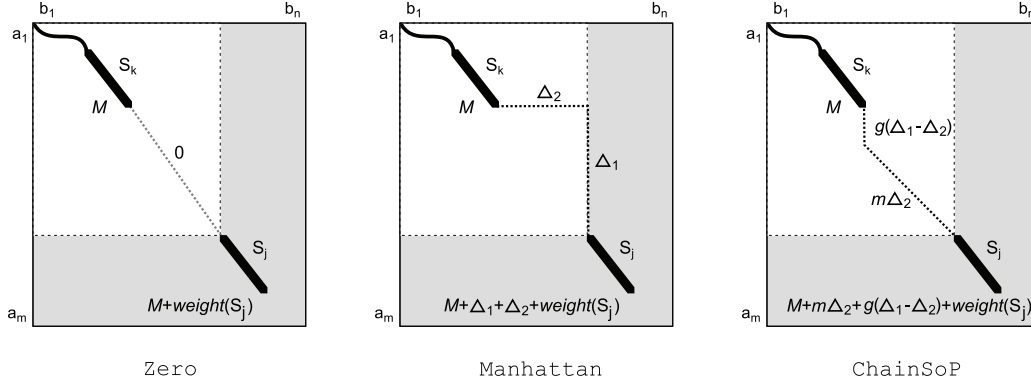
```
String< Seed<int, MultiSeed> > seeds;
...
String< Seed<int, MultiSeed> > chain;
Score<int, Manhattan> scoring;
int score = globalChaining(seeds, chain, scoring);
```

Listing 16: **Global Chaining Example.** We omit the process of filling the container `seeds` with seeds.

### 9.6.3 Chaining Using Sparse Dynamic Programming

The algorithm `GENERICCHAINING` takes quadratic time, because it has to examine a linear number of predecessor candidates  $\mathcal{S}_k$  for each seed  $\mathcal{S}_j$ . We can improve this for some *gapscore* functions by using efficient data structures that allow to determine an optimal predecessor seed in sublinear time. This technique called *sparse dynamic programming* (Eppstein et al. 1992) may speed up chaining as long as the dimension  $d$  of the seeds is small compared to the number  $n$  of seeds. Table 24 shows the gap scoring functions for which SeqAn implements optimized chaining algorithms. For example, Algorithm 7 (described in Gusfield 1997, pages 325–329) solves the chaining problem for  $d = 2$  and *gapscore*  $\equiv 0$  (i.e. the scoring scheme ‘Zero’) in time  $O(n \log n)$ .

`SPARSECHAINING` enumerates all positions  $left_1(\mathcal{S}_j)$  and  $right_1(\mathcal{S}_j)$  in increasing order. If the begin position of a seed  $\mathcal{S}_j$  is processed (lines 9 and 10), then the optimal score  $M_j$  of chains ending in  $\mathcal{S}_j$  is computed, and the algorithm appends  $\mathcal{S}_j$  to a seed  $\mathcal{S}_k \in D$ , where  $D$  is a set of potentially optimal predecessors for subsequent seeds.  $D$  is updated whenever the end position of a seed is processed (lines 12 to 15). Let  $\mathcal{S}_j$  and  $\mathcal{S}_k$  be two different seeds with  $right_2(\mathcal{S}_j) \leq right_2(\mathcal{S}_k)$  and  $M_j \geq M_k$ . Then all subsequent seeds  $\mathcal{S}_i$  that can be appended to  $\mathcal{S}_k$  can also be appended to  $\mathcal{S}_j$  without losing score. If  $\mathcal{S}_j \in D$ , then there is no need to keep  $\mathcal{S}_k \in D$ . We say that  $\mathcal{S}_j$  ‘dominates’  $\mathcal{S}_k$ . A seed  $\mathcal{S}_j$  is added to  $D$  in line 15, if and only if it is not dominated by any other seed in  $D$ , and in this case, all seeds  $\mathcal{S}_k$  that are dominated by  $\mathcal{S}_j$  are extracted from  $D$  in line 14. It follows that (1)  $D$  contains all seeds that were already processed except some seeds that are dominated by other seeds in  $D$ , and (2) no seed in  $D$  is dominated by another seed in  $D$ . Therefore  $D$  contains an optimal predecessor for any seed  $\mathcal{S}_j$  that is about to be appended, and this is the seed  $\mathcal{S}_{T_j} \in D$  found in line 9 because a ‘better’ seed  $\mathcal{S}_k \in D$



**Zero** All gaps between seeds score 0, that is  $gapscore(\mathcal{S}_k, \mathcal{S}_j) = 0$  for all seeds  $\mathcal{S}_k$  and  $\mathcal{S}_j$ .

**Manhattan** The gap score is proportional to the sum of the distances between the segments in the seed, that is

$$gapscore(\mathcal{S}_k, \mathcal{S}_j) = g \sum_{i=1}^d \Delta_i,$$

where  $\Delta_i = left_i(\mathcal{S}_j) - right_i(\mathcal{S}_k)$  and  $g < 0$  is the score for a single blank.

**ChainSoP** This gap scoring scheme was proposed by Myers and Miller (1995). For  $d = 2$ , the segments between  $\mathcal{S}_k$  and  $\mathcal{S}_j$  are aligned as long as possible with mismatches, and the rest is filled up with blanks, that is:

$$gapscore(\mathcal{S}_k, \mathcal{S}_j) = S_{1,2} = \begin{cases} m\Delta_2 + g(\Delta_1 - \Delta_2), & \text{if } \Delta_1 \geq \Delta_2 \\ m\Delta_1 + g(\Delta_2 - \Delta_1), & \text{if } \Delta_1 \leq \Delta_2 \end{cases}$$

where  $\Delta_i = left_i(\mathcal{S}_j) - right_i(\mathcal{S}_k)$  and  $g, m \leq 0$  are the scores for a single blank and a single mismatch.

For  $d > 2$ ,  $gapscore$  is the sum-of-pairs score:

$$gapscore(\mathcal{S}_k, \mathcal{S}_j) = \sum_{1 \leq i < i' \leq d} S_{i,i'}$$

Figure 24: **Gap Scoring Schemes for Chaining.** The score of a gap between a seed  $\mathcal{S}_j$  and a predecessor  $\mathcal{S}_k$ ; three specialization of class **Score** are listed. Note that  $gapscore(\mathcal{S}_k, \mathcal{S}_j)$  is only defined if  $\mathcal{S}_j$  can be appended to  $\mathcal{S}_k$ , i.e. if  $\Delta_i = left_i(\mathcal{S}_j) - right_i(\mathcal{S}_k) \geq 0$  for all  $i \in \{1, \dots, d\}$ .



with  $right_2(\mathcal{S}_k) \leq left_2(\mathcal{S}_j)$  and  $M_k > M_{T_j}$  would dominate  $\mathcal{S}_{T_j}$ . Note that the results of the ‘argmax’ operation in line 9 is well defined, because there is always a seed  $\mathcal{S}_k \in D$  such that  $right_2(\mathcal{S}_k) = 0$ .

If we apply a suitable *dictionary* data structures for storing  $D$  like a *skip list* that sorts the seeds  $\mathcal{S}$  according to  $right_2(\mathcal{S})$ , then each operation for searching, adding, and extracting seeds in  $D$  takes time  $O(\log n)$ . The complete algorithm runs in  $O(n \log n)$ , since each seed  $\mathcal{S}$  is added to  $D$  and extracted from  $D$  only once.

We can apply SPARSECHAINING for the gap scoring scheme **Manhattan**, if we modify the condition for ‘ $\mathcal{S}_j$  dominates  $\mathcal{S}_k$ ’: Suppose that a seed  $\mathcal{S}_l$  can be appended either to  $\mathcal{S}_j$  and  $\mathcal{S}_k$ . Then  $\mathcal{S}_j$  would be preferred, if  $M_j + gapscore(\mathcal{S}_j, \mathcal{S}_l) > M_k + gapscore(\mathcal{S}_k, \mathcal{S}_l)$ . This is equivalent to  $M'_j > M'_k$  where  $M'_* = M_* + \sum_{i=1}^d right_i(\mathcal{S}_*)$ . Note that this is independent from the appended seed  $\mathcal{S}_l$ , so we can define that  $\mathcal{S}_j$  dominates  $\mathcal{S}_k$ , if  $right_2(\mathcal{S}_j) \leq right_2(\mathcal{S}_k)$  and  $M'_j \geq M'_k$ .

<div style="border: 1px solid black; padding: 10px; min-height: 300px;"> <div style="text-align: right; margin-bottom: 5px;">▷ SPARSECHAINING(<math>\mathcal{S}_1, \dots, \mathcal{S}_n</math>)</div> <div style="margin-bottom: 5px;">1    compute top seed <math>\mathcal{S}_0</math> and bottom seed <math>\mathcal{S}_{n+1}</math></div> <div style="margin-bottom: 5px;">2    <math>M_0 \leftarrow 0</math></div> <div style="margin-bottom: 5px;">3    <math>D \leftarrow \{\mathcal{S}_0\}</math></div> <div style="margin-bottom: 5px;">4    <math>S \leftarrow \emptyset</math></div> <div style="margin-bottom: 5px;">5    <b>for</b> <math>j \leftarrow 1</math> <b>to</b> <math>n + 1</math> <b>do</b></div> <div style="margin-bottom: 5px;">6       <math>S \leftarrow S \cup \{ \langle left_1(\mathcal{S}_j), j \rangle \} \cup \{ \langle right_1(\mathcal{S}_j), j \rangle \}</math></div> <div style="margin-bottom: 5px;">7       <b>for each</b> <math>\langle pos, j \rangle \in S</math> <b>in increasing order of</b> <math>pos</math> <b>do</b></div> <div style="margin-bottom: 5px;">8        <b>if</b> <math>pos = left_1(\mathcal{S}_j)</math> <b>then</b></div> <div style="margin-bottom: 5px;">9             <math>T_j \leftarrow \operatorname{argmax}_k \{ right_2(\mathcal{S}_k) \leq left_2(\mathcal{S}_j) \mid \mathcal{S}_k \in D \}</math></div> <div style="margin-bottom: 5px;">10            <math>M_j \leftarrow M_{T_j} + gapscore(\mathcal{S}_k, \mathcal{S}_j) + weight(\mathcal{S}_j)</math></div> <div style="margin-bottom: 5px;">11        <b>if</b> <math>pos = right_1(\mathcal{S}_j)</math> <b>then</b></div> <div style="margin-bottom: 5px;">12             <b>if</b> no seed <math>\in D</math> dominates <math>\mathcal{S}_k</math> <b>then</b></div> <div style="margin-bottom: 5px;">13                  <b>for each</b> <math>\mathcal{S}_k \in D</math> dominated by <math>\mathcal{S}_j</math> <b>do</b></div> <div style="margin-bottom: 5px;">14                       <math>D \leftarrow D \setminus \{\mathcal{S}_k\}</math></div> <div style="margin-bottom: 5px;">15                       <math>D \leftarrow D \cup \{\mathcal{S}_j\}</math></div> <div style="margin-bottom: 5px;">16       <b>return</b> CHAINTRACEBACK(<math>\mathcal{S}_0, \dots, \mathcal{S}_{n+1}, n + 1, T</math>)</div> </div>	<div style="font-size: 3em; line-height: 1; padding: 0 10px;"> <div style="display: inline-block; vertical-align: middle;">}</div> <div style="display: inline-block; vertical-align: middle;">append <math>\mathcal{S}_j</math></div> </div> <div style="font-size: 3em; line-height: 1; padding: 0 10px;"> <div style="display: inline-block; vertical-align: middle;">}</div> <div style="display: inline-block; vertical-align: middle;">update <math>D</math></div> </div>
---	---

Algorithm 7: **Global Chaining by Sparse Dynamic Programming.**  $\mathcal{S}_1, \dots, \mathcal{S}_n$  is a set of 2-dimensional seeds. The algorithm may be used for gap scoring functions **Zero** and **Manhattan**, where the semantic of ‘dominate’ depends on the scoring function.

SeqAn also implements a sparse dynamic programming algorithm by Myers and Miller (1995) with some modifications by Abouelhoda and Ohlebusch (2003) for **ChainSoP** scoring and arbitrary  $d \geq 2$ , see Wöhrle (2006) for more

details. Note that both runtime and space requirements of this algorithm grow exponentially with respect to the seed dimension  $d$ .

### 9.6.4 Banded Alignment

Given a chain  $\mathcal{C} = \langle \mathcal{S}_1, \dots, \mathcal{S}_n \rangle$  of seeds between two sequences  $a = a_1 \dots a_n$  and  $b = b_1 \dots b_m$ , we can find a good alignment using ‘*banded alignment*’. Like the alignment algorithms we described in Section 9.5, this method bases on *dynamic programming*. Remember that the Needleman-Wunsch algorithm (Section 9.5.1) computes  $n \times m$  score  $M_{i,j}$  of the best alignments between the prefixes  $a_1 \dots a_i$  and  $b_1 \dots b_j$ . Since  $\mathcal{C}$  gives us an estimate of the approximate optimal alignment, we need to compute only those values  $M_{i,j}$  that lay near to  $\mathcal{C}$ . This ‘band’ of width  $B$  contains the following pairs of coordinates, see Figure 25: (1) for any two characters  $a_x$  and  $b_y$  that are aligned by a seed  $\mathcal{S}_k \in \mathcal{C}$  all pairs  $\langle i, j \rangle$  with  $|i - x| + |j - y| \leq B$ , and (2) the square of pairs  $\langle i, j \rangle$  with  $right_1(\mathcal{S}_k) - B \leq i < left_1(\mathcal{S}_{k+1}) + B$  and  $right_2(\mathcal{S}_k) - B \leq j < left_2(\mathcal{S}_{k+1}) + B$  for  $k \in \{1, \dots, n\}$ . Computing only these cells of  $M$  speeds up the alignment process. SeqAn provides the function `bandedChainAlignment` that computes an optimal banded alignment following a chain.

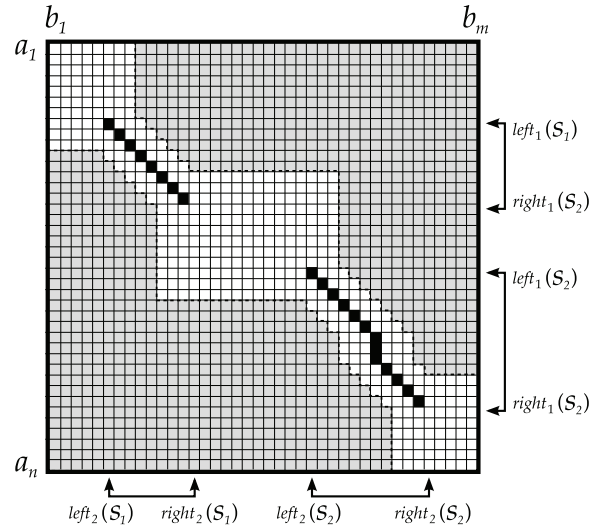


Figure 25: **Banded Alignment.** The white area of matrix  $M$  represents the ‘band’ of width  $B = 3$  around the chain  $\mathcal{C} = \langle \mathcal{S}_1, \mathcal{S}_2 \rangle$ .

# Chapter 10

## Pattern Matching

The *pattern matching* problem is to find a given ‘*needle*’ sequence  $p$  in a ‘*haystack*’ sequence  $t$ , for example to determine where a string  $t$  contains the string  $p$  as a substring. There are several variants of this problem:

- **Exact Matching:** Find substrings  $p$  in  $t$ . See Section 10.1 for single searching needles, and Section 10.2 for searching multiple needles.
- **Approximate Matching:** Find substrings  $s$  in  $t$  with  $\text{score}(s, p) \geq k$  for a given threshold  $k$ . See Section 10.3 for alignment scoring schemes *score*.
- **Complex Pattern Searching:**  $p$  is an expression that encodes a set of strings to be find in  $t$ . See for example in Section 10.4.2 how to search for regular expressions.

In SeqAn, the function `find` finds an occurrence of a needle in a haystack; it can repeatedly called to find all occurrences. `find` needs the following information to work: The haystack, the needle, what kind of algorithm to be used, the current state of the search (e.g. the last found position), and possibly – depending on the algorithm – some preprocessing data. This information is divided onto two objects: (1) the *finder* that holds all information related to the haystack, and (2) the *pattern* that holds information related to the needle, see Figure 26.

Splitting the data this way has several advantages:

- It reduces type dependencies, since finder types are independent from the type of needles, and pattern types are independent from the type of the haystack. This simplifies the code both for using pattern matching algorithms and for implementing new finder or pattern types.
- The preprocessing of the needle is stored in the pattern object and can be re-used when searching this needle in several haystacks.

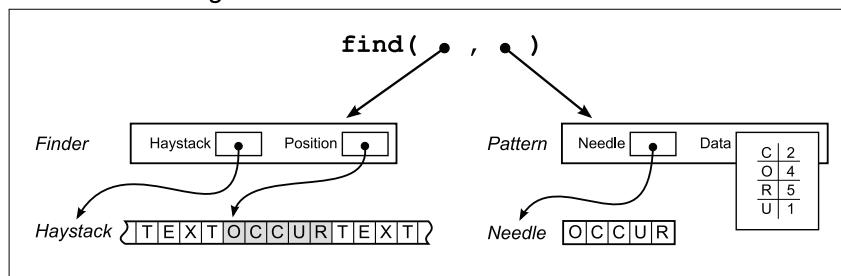
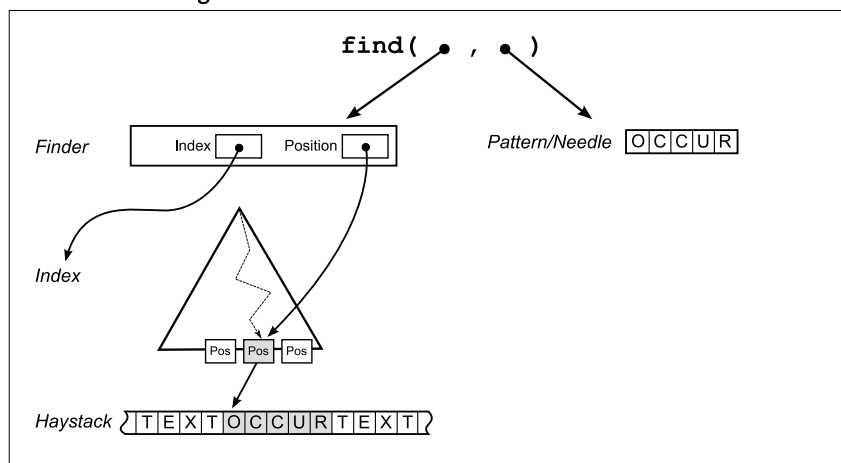
*Online Searching**Index Searching*

Figure 26: **Online Searching and Index Searching.** **Top:** Calling `find` for online searching. The search algorithm is determined by the type of the pattern, which contains all relevant preprocessing data. **Bottom:** Calling `find` for index searching (see Chapter 12). The search algorithm depends on the finder type. The needle sequence acts directly as a pattern.

- The applied algorithm is implicitly selected by the choice of the finder and pattern types. For example, calling `find` with an instance of the `Horspool` specialization of `Pattern` means that Horspool’s algorithm (see Section 10.1.2) is applied.
- If no further information related to the needle has to be stored (e.g. for index searching algorithms, see Chapter 12), then the needle itself acts as pattern, and that again simplifies the handling of SeqAn pattern matching algorithms.

The last found match position is stored in the finder and can be retrieved by the function `position`. This is for most searching algorithms the position of the first value of the match, except for approximate searching algorithms: Since finding the begin position of an approximate match needs some additional overhead, the function `position` returns the position of the *last* character of the match (see Section 10.3). SeqAn also offers the functions `beginPosition` and `endPosition` for determining the begin and end position of matches explicitly, where `endPosition` is immediately available after calling `find`, and `beginPosition` requires in the case of approximate string matching a previous call of the function `findBegin` in order to find the beginning of the match. The position of a finder can also be set by the user via the function `setPosition`. If searching is started or resumed at a specific position *pos*, then only occurrences on positions  $\geq pos$  will be found. Moreover, all algorithms in SeqAn guarantee that the occurrences found by calling `find` are emitted in order of increasing positions.

In this chapter, we will focus on *online searching algorithms*, which solve the pattern matching problems by preprocessing the needle, not the haystack. Most online searching algorithms implemented in SeqAn are also described in (Navarro and Raffinot 2002).

## 10.1 Exact Searching

The *exact searching problem* is to find for a given needle  $p_1 \dots p_m$  and haystack  $t_1 \dots t_n$  all positions  $j$  for which  $p_1 \dots p_m = t_j \dots t_{j+m-1}$ . Table 13 shows some online searching algorithms for exact searching provided by SeqAn. Listing 17 shows how to use these algorithms. The performance of online algorithm depends (beside other things) from the *alphabet size*  $\Sigma$  and the needle length  $m$ , see Figure 28.

### 10.1.1 Brute-Force Exact Searching

The most simple way for searching a needle  $p = p_1 \dots p_m$  in a haystack  $t = t_1 \dots t_n$  is to compare  $p$  with  $t_{pos+1} \dots t_{pos+m}$  for each position  $pos \in$

<b>Simple</b>	A brute-force but generic searching algorithm that can deal with sequences of all value types.
<b>Horspool</b>	Horspool's algorithm (Horspool 1980) is a simple yet fast algorithm with in average sublinear searching time that is suitable for many pattern matching settings.
<b>ShiftOr</b>	An algorithm that uses bit parallelism. Should only be used for patterns that are not longer than a machine word, i.e. 32 or 64 characters. Even for small patterns, it is outperformed by Horspool for alphabets larger than Dna.
<b>BFAM</b>	Backward Factor Automaton Matching is an algorithm that applies an automaton of the reversed needle's substrings. It is a good choice for long patterns.
<b>BndmAlgo</b>	The Backward Nondeterministic DAWG (Directed Acyclic Word Graph) Matching algorithm uses a special automaton to scan through the haystack. It is an alternative to BFAM for medium sized patterns.

Table 13: **Exact Pattern Matching Algorithms.**

```
String<char> t = "he_is_the_hero";
String<char> p = "he";
Finder<String<char> > finder(t);
Pattern<String<char>, Horspool> pattern(p);
while (find(finder, pattern))
{
    std::cout << position(finder) << ", "; //output: 0,7,10
}
```

Listing 17: **Exact Online Searching Example.**

$\{0, \dots, n - m\}$  (Algorithm 8). This method takes time  $O(n \times m)$ , and it is rather slow compared to other algorithms. SIMPLESEARCH has the advantage that it is completely generic and works for arbitrary value types. All other exact searching algorithms in SeqAn need some additional space for storing preprocessed data, and this space could exceed memory when large alphabets are used; SIMPLESEARCH on the other hand is not limited this way, since it needs no preprocessing data at all.

```

1  ▷ SIMPLESEARCH ( $p_1 \dots p_m, t_1 \dots t_n$ )
2  for  $pos \leftarrow 0$  to  $n - m$  do
3  |   if  $p_1 \dots p_m = t_{pos+1} \dots t_{pos+m}$  then
4  |   |   report match at position  $pos + 1$ 

```

Algorithm 8: **Brute-Force Exact Searching.**

### 10.1.2 Horspool's Algorithm

*Horspool's algorithm* (Horspool 1980) (see Algorithm 9) is a simplification of an algorithm by Boyer and Moore (1977). The algorithm compares the needle  $p_1 \dots p_m$  with the substring  $t_{pos+1} \dots t_{pos+m}$  of the haystack, where the search starts at  $pos = 0$ . After each comparison,  $pos$  is increased by a ‘safe shift width’  $k$ , which means that  $k$  is small enough that no possible match in between gets lost. Suppose that  $pos$  is increased by  $i$ , then  $t_{pos+m}$  will be compared to  $p_{m-i}$  during the next comparison step. Hence if  $p_{m-i} \neq t_{pos+m}$  for all  $1 \leq i \leq k$ , then  $k$  is ‘safe’. The maximum safe shift width for each possible value of  $t_{pos+m}$  is stored in a preprocessed table *skip*.

The worst case running time of Horspool's algorithm is  $O(n^2)$ , but in practice it runs in linear or even sublinear time on average. This algorithm is a good choice for most exact pattern matching problems, except (1) if the alphabet (and hence the shift width) is very small compared to the pattern length, since in this case it is outperformed by other algorithms, or (2) for very large alphabets, since storing *skip* gets inefficient then.

Horspool's algorithm is applied when the **Horspool** specialization of the class **Pattern** is used as pattern, see Listing 17 for an example.

### 10.1.3 Shift-Or Algorithm

*Shift-Or* is a simple online algorithm for exact pattern matching that benefits from bit-parallelism. For a given needle  $p = p_1 \dots p_m$  and a haystack  $t_1 \dots t_n$ , we define for each  $j \in \{1, \dots, n\}$  a length- $m$  vector  $b^j$  of booleans

$$b_i^j := “p_1 \dots p_i \text{ does not match to a suffix of } t_1 \dots t_j”,$$

<pre> 1  skip[c] ← m for all c ∈ Σ 2  for i ← 1 to m − 1 do skip[p<sub>i</sub>] ← m − i 3  pos ← 0 4  while pos ≤ n − m do 5      i ← m 6      while p<sub>i</sub> = t<sub>pos+i</sub> do 7          if i = 1 then 8              report match at position pos + 1 9              break 10         i ← i − 1 11     pos ← pos + skip[t<sub>pos+m</sub>] </pre>	<div style="display: inline-block; vertical-align: middle;"> <div style="font-size: 3em; vertical-align: middle;">}</div> <div style="display: inline-block; vertical-align: middle;">preprocessing</div> </div> <div style="display: inline-block; vertical-align: middle; margin-top: 40px;"> <div style="font-size: 3em; vertical-align: middle;">}</div> <div style="display: inline-block; vertical-align: middle;">searching</div> </div>
--	---

Algorithm 9: **Horspool's Algorithm.**

$i \in \{1, \dots, m\}$ . If *not*  $b_m^j$ , then  $p$  matches  $t$  at a position  $j - m + 1$ . At each time  $j$ , SHIFTOR stores  $b^j$  in a bit vector  $b$ . When  $j$  is increased,  $b$  is updated according to the recursion:

$$b_i^j = b_{i-1}^{j-1} \text{ or } (p_i \neq t_j).$$

SHIFTOR applies bit-parallelism, hence this takes only one left-shift operation on  $b$  and one bit-wise *or* operation with a bit vector  $mask[t_j]$  (see Algorithm 10, line 5). The bit vectors  $mask[c]$  are preprocessed for each possible value  $c \in \Sigma$ ;  $mask[c]_i = 0$ , iff  $p_i = c$ .

<pre> 1  mask[c] ← 1<sup>m</sup> for all c ∈ Σ 2  for i ← 1 to m do mask[p<sub>i</sub>]<sub>i</sub> ← 0 3  b ← 1<sup>m</sup> 4  for j ← 1 to n do 5      b ← (b &lt;&lt; 1)   mask[t<sub>j</sub>] 6      if b<sub>m</sub> = 0 then 7          report a match at j − m + 1 </pre>	<div style="display: inline-block; vertical-align: middle;"> <div style="font-size: 3em; vertical-align: middle;">}</div> <div style="display: inline-block; vertical-align: middle;">preprocessing</div> </div> <div style="display: inline-block; vertical-align: middle; margin-top: 40px;"> <div style="font-size: 3em; vertical-align: middle;">}</div> <div style="display: inline-block; vertical-align: middle;">searching</div> </div>
--	---

Algorithm 10: **Shift-Or Algorithm.**

The Shift-Or algorithm is quite fast, as long as  $b$  fits into one machine word, i.e. as long as  $m \leq 32$  or  $64$ . For longer patterns, multiple machine words must be used, but this diminishes the positive effect of the bit-parallelism. Moreover, Shift-Or is outperformed by Horspool's algorithm (Section 10.1.2) for all but



very small alphabets (see Figure 27 on page 110). Shift-Or is therefore best for small patterns and small alphabets.

#### 10.1.4 Backward Factor Automaton Matching

*Backward Factor Automaton Matching* (BFAM) is an exact online algorithm that applies an automaton, e.g. an *oracle* automaton as described by Allauzen, Crochemore, and Raffinot (2001) or a *trie*. The principle of BFAM is *Backward Factor Searching* as presented in Algorithm 11: BF reads a suffix of  $t_1 \dots t_{pos+m}$  from back to front until either a match of  $p$  is found, or  $p$  does not contain any substring (‘factor’) that matches to the read suffix. If  $p$  does not contain  $t_{pos+k} \dots t_{pos+m}$ , then  $k$  is a ‘safe shift’, i.e.  $pos$  can be increased by  $k$  without losing a match, since any substring of  $t$  that starts at a position between  $pos+1$  to  $pos+k$  also contains  $t_{pos+k} \dots t_{pos+m}$ .

```

1  ▷ BF ( $p = p_1 \dots p_m, t_1 \dots t_n$ )
2   $pos \leftarrow 0$ 
3  while  $pos \leq n - m$  do
4       $k \leftarrow m$ 
5      while  $p$  contains  $t_{pos+k} \dots t_{pos+m}$  do
6          if  $k = 1$  then
7              report match at  $pos + 1$ 
8              break
9               $k \leftarrow k - 1$ 
10      $pos \leftarrow pos + k$ 

```

Algorithm 11: **Backward Factor Searching Principle.**

The main question in BF is how to check the condition in line 4 whether  $p$  contains  $t_{pos+k} \dots t_{pos+m}$ . For that purpose, BFAM (Algorithm 12) applies an *factor automaton* on the reverse needle  $p_m \dots p_1$ , i.e. an automaton that accepts all substrings of this sequence. This automaton is processed on  $t_{pos+m} \dots t_{pos+1}$  until either a match is found, or an undefined state is reached because the needle does not contain the string  $t_{pos+k} \dots t_{pos+m}$ .

In SeqAn, the kind of automaton is specified when choosing the specialization of the class **Pattern**: The specialization **BFAM<Oracle>** is used for applying an oracle automaton (see Section 13.1.2) and **BFAM<Trie>** for an suffix trie (see Section 13.1.1).

Oracle automata may also accept strings other than substrings of  $p_m \dots p_1$ , and this may lead to shorter shift widths. Fortunately, the only length- $m$  string accepted by the oracle is  $p_m \dots p_1$  itself, so we need no additional verification in line 6 of Algorithm 11, as it will be necessary for MULTIBFAM in line 15

of Algorithm 15. Oracles are more compact than suffix tries: The oracle of  $p_m \dots p_1$  has only  $m+1$  states and at most  $2 \times m$  transitions, whereas the number of states and transitions of a suffix trie can be quadratic. This parsimony benefits the run time, because a smaller automaton has better chances to stay in cache, and because oracles take less time to be built up. A comparison between the run times of the two variants (Figure 27 on page 110) reveals that  $\text{BFAM} \langle \text{TRIE} \rangle$  is slightly faster than  $\text{BFAM} \langle \text{ORACLE} \rangle$  for small alphabets and needle lengths, whereas for large alphabets or needle lengths the oracle takes advantage of its space efficiency.

<pre> 1  ▷ BFAM (<math>p_1 \dots p_m, t_1 \dots t_n</math>) 2  <math>a \leftarrow \text{BUILDFACTORAUTOMATON}(p_m \dots p_1)</math> 3  <math>pos \leftarrow 0</math> 4  <b>while</b> <math>pos \leq n - m</math> <b>do</b> 5      <math>q \leftarrow \delta_a(\text{initial state of } a, t_{pos+m})</math> 6      <math>k \leftarrow m</math> 7      <b>while</b> <math>q</math> is defined <b>do</b> 8          <b>if</b> <math>k = 1</math> <b>then</b> 9              report match at <math>pos + 1</math> 10             <b>break</b> 11             <math>k \leftarrow k - 1</math> 12             <math>q \leftarrow \delta_a(q, t_{pos+k})</math> 13         <math>pos \leftarrow pos + k</math> </pre>	<div style="display: inline-block; vertical-align: middle;"> <div style="font-size: 3em; vertical-align: middle; margin-right: 5px;">}</div> <div style="display: inline-block; vertical-align: middle;">preprocessing</div> </div> <div style="display: inline-block; vertical-align: middle; margin-top: 10px;"> <div style="font-size: 3em; vertical-align: middle; margin-right: 5px;">}</div> <div style="display: inline-block; vertical-align: middle;">searching</div> </div>
---	---

Algorithm 12: **Backward Factor Automaton Searching.**  $\delta_a$  is the transition function of  $a$ . Note that  $a$  is built on the *reverse* needle  $p_m \dots p_1$ .

### 10.1.5 Backward Nondeterministic DAWG Matching

The *BNDM algorithm* (‘Backward Nondeterministic Directed Acyclic Word Graph Matching’) is a bit-parallel variant of an algorithm by Crochemore et al. (1994). It applies bit-parallelism for tracking the substrings of the needle during backward factor searching (BF, see Algorithm 11, line 4). Let us define for each  $k \in \{1, \dots, m\}$  a length- $m$  vector  $b^k$  of booleans

$$b_i^k := “t_{pos+k} \dots t_{pos+m} \text{ matches a prefix of } p_i \dots p_m”.$$

BNDM (Algorithm 13) stores  $b^k$  in a bit vector  $b$ , which is updated when  $k$  is decreased according to the recursion:

$$b_i^k = b_{i+1}^{k+1} \text{ and } (p_i = t_{pos+k}).$$

This takes two bit-parallel operations: One right shift in line 14 and one bit-wise ‘and’ with a preprocessed bit vector  $mask[t_{pos+i}]$  in line 9 of Algorithm 13. The bit vectors  $mask[c]$  are preprocessed for each possible value  $c \in \Sigma$ ;  $mask[c]_i = 1$ , iff  $p_i = c$ .

BNDM improves the ‘safe’ shift width of BF as follows: If  $b_1^k = 0$ , then  $t_{pos+k} \dots t_{pos+m}$  does not match to a prefix of  $p$ , hence  $p$  does not match  $t$  at position  $pos+k$ . Suppose that  $k$  is a ‘safe’ shift, then  $k+1$  will also be ‘safe’. Hence, we need only to take into account shift widths  $k$  with  $b_1^k = 1$ . The variable *skip* stores the last found  $k$  for which  $b_i = 1$  (line 12). *skip* is then used in line 15 as shift width.

<div style="border: 1px solid black; padding: 10px; position: relative;"> <div style="position: absolute; left: -40px; top: 0; bottom: 0; border-left: 1px solid black; margin-left: -1px;"></div> <div style="position: absolute; right: -40px; top: 0; bottom: 0; border-right: 1px solid black; margin-right: -1px;"></div> <div style="position: absolute; bottom: -40px; left: 0; right: 0; border-top: 1px solid black; margin-top: -1px;"></div> </div>	<pre> ▷ BNDM (<math>p_1 \dots p_m, t_1 \dots t_n</math>) 1  <math>mask[c] \leftarrow 0^m</math> for all <math>c \in \Sigma</math> 2  <b>for</b> <math>i \leftarrow 1</math> <b>to</b> <math>m</math> <b>do</b> <math>mask[p_i]_i \leftarrow 1</math> 3  <math>pos \leftarrow 0</math> 4  <b>while</b> <math>pos \leq n - m</math> <b>do</b> 5      <math>k \leftarrow m</math> 6      <math>skip \leftarrow m</math> 7      <math>b \leftarrow 1^m</math> 8      <b>while</b> <math>b \neq 0^m</math> <b>do</b> 9          <math>b \leftarrow b \wedge mask[t_{pos+k}]</math> 10         <math>k \leftarrow k - 1</math> 11         <b>if</b> <math>b_1 = 1</math> <b>then</b> 12             <b>if</b> <math>k &gt; 0</math> <b>then</b> <math>skip \leftarrow k</math> 13             <b>else</b> report match at <math>pos + 1</math> 14         <math>b \leftarrow b &gt;&gt; 1</math> 15     <math>pos \leftarrow pos + skip</math> </pre>	<div style="display: flex; align-items: center;"> <div style="font-size: 3em; margin-right: 10px;">}</div> <div>preprocessing</div> </div> <div style="display: flex; align-items: center; margin-top: 10px;"> <div style="font-size: 3em; margin-right: 10px;">}</div> <div>searching</div> </div>
--	---	---

Algorithm 13: **Backward Nondeterministic DAWG Matching.**

### 10.1.6 Results

Figure 27 shows the average run times divided by the length of the searched haystack for searching needles of length  $m$ . The machine word size was 32, so the run times of the bit parallel algorithm **ShiftOr** and **BndmAlgo** are discontinuous at multitudes of 32. For small alphabets like DNA ( $|\Sigma| = 4$ ), the fastest algorithms are **ShiftOr** for small  $m$ , **BFAM<Trie>** for middle sized  $m$ , and **BFAM<Oracle>** for large  $m$ . For larger alphabets, e.g. when searching proteins or English texts, either **Horspool** for small  $m$  or **BFAM** for larger  $m$  is the fastest, see Figure 28. Compared with the results of Navarro and Raffinot (2002, Fig. 2.22), our implementation of the BNDM algorithm is outperformed in any parameter setting.

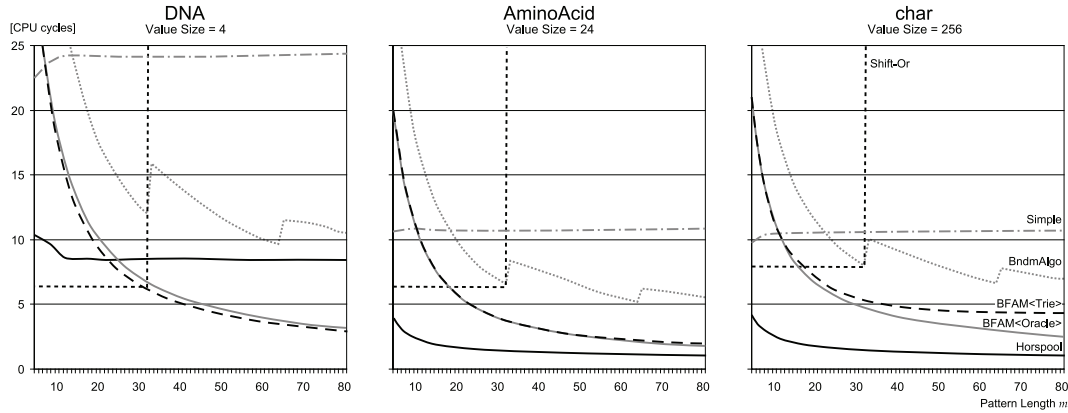


Figure 27: **Run Times of Exact Pattern Matching Algorithms.** The average run times per haystack value for different exact pattern matching algorithms depending on the length  $m$  of the needle. We searched for patterns in (1) the genome of Escherichia Coli, (2) proteins from the Swiss-Prot database, and (3) the English Bible.

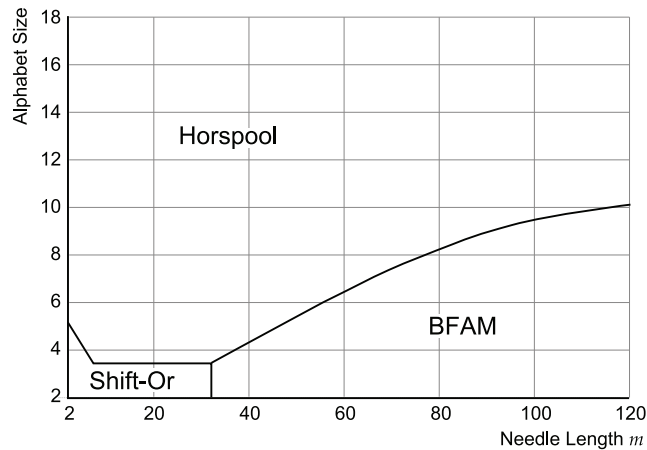


Figure 28: **Fastest Exact Pattern Matching Algorithm.** The best algorithm for searching all exact occurrences of length- $m$  patterns in random haystacks, depending on the size  $|\Sigma|$  of the alphabet. Since the pattern is a random string, Horspool gains ground compared to Figure 27.

## 10.2 Exact Searching of Multiple Needles

In this section, we describe algorithm that search several needles  $p^1, \dots, p^k$  in a haystack  $t_1 \dots t_n$  at once, which is in general faster than searching one needle after the other. We search the pairs  $(i, j)$  for which  $p^i$  matches a prefix of  $t_j \dots t_n$ . Many algorithms proposed in (Navarro and Raffinot 2002) are implemented in SeqAn, see Table 14. Here, we will only describe two of them in more details, since these two outperform all others in almost any case.

<b>WuManber</b>	An extension of Horspool's algorithm (Horspool 1980) for searching multiple needles. The favoured algorithm in many cases.
<b>MultiBFAM</b>	Backward Factor Automaton Matching for Multiple sequences is an extension of BFAM for searching multiple sequences. It applies an automaton that accepts the reversed substrings of the needles' prefixes. This algorithm is a good choice for long patterns, small alphabets, or large needle sets.
<b>AhoCorasick</b>	An algorithm by Aho and Corasick (1975) that uses an extended trie automaton to scan through the haystack sequence. It performs well especially for small alphabets pattern lengths.
<b>MultipleShiftAnd</b>	An extension of the Shift-And algorithm for multiple patterns. This algorithm is competitive only if the sum of the needle lengths is smaller than the size of one machine word.
<b>SetHorspool</b>	Another extension of Horspool's algorithm (Horspool 1980) for multi-pattern searching that applies a trie of the reverse needles. In practice, it is outperformed by WuManber.

Table 14: **Exact Pattern Matching Algorithms for Searching Multiple Needles.**

Listing 18 shows an example how to use the function `find` to search for multiple patterns at once. After each call, `position(pattern)` returns the index number of the needle that was recently found starting at `position(finder)` in the haystack.

Some algorithms ensure that hits emerge in a certain order. For example, the hits found by **WuManber** and **MultiBFAM** are sorted in increasing order by `position(finder)`, and two hits at the same position `position(finder)` are sorted in increasing order by `position(pattern)`. The example program therefore finds first "the" at position 0, then "hero" at position 4, then "theory" at position 11, and finally "the", also at position 11.

### 10.2.1 Wu-Manber Algorithm

The algorithm by Wu and Manber (1994) is an extension of Horspool's algorithm (see Section 10.1.2, page 105) for multiple needles. **WUMANBER**

```

String<char> t = "the_heroes_theory";
String<String<char> > p;
appendValue(p, "theory");
appendValue(p, "hero");
appendValue(p, "the");
Finder<String<char> > finder(t);
Pattern<String<String<char> >, WuManber> pattern(p);
while (find(finder, pattern))
{
    std::cout << "found pattern " << position(pattern)
               << "at position " << position(finder)
               << ", ";
}

```

Listing 18: Multiple Pattern Searching Example.

(Algorithm 14) compares the needles  $p^1, \dots, p^k$  to the haystack  $t$  at a position  $pos$ , which is then increased by a *safe shift width*. For multiple needles, it is not advisable to select the shift width depending on the occurrences of a *single* value  $t_{pos+m}$  within the needles, as it is done by HORSPOOL (Algorithm 9, page 106), since this would lead to rather small shift widths and hence a poor performance, because it is rather probable to find each possible value in the ending region of at least one of the needles. WUMANBER uses therefore  $q \geq 2$  values  $t_{pos+m-q+1} \dots t_{pos+m}$  for determining the shift widths. A preprocessed table *shift* stores for each  $q$ -gram  $w \in \Sigma^q$  a safe shift width. *shift* may use hashing if a table size  $|\Sigma|^q$  would be too large for storing a shift width for each  $q$ -gram in memory. A second table *verify* is used to determine which needles possibly match and are verified.

The expected shift widths are optimal if  $q$  is selected such that the number  $|\Sigma|^q$  of possible  $q$ -grams is about the number of overlapping  $q$ -grams occurring in needles. In practice, the optimal  $q$  may be smaller, because the computation of a hash value needed to access *shift* and *verify* takes time  $O(q)$ , and this dominates the performance of the main loop (lines 11 to 18).

### 10.2.2 Multiple BFAM Algorithm

The Multiple Backward Factor Automaton Matching algorithm extends the BFAM algorithm (see Section 10.1.4, page 107) for searching multiple needles. The factor automaton, e.g. a factor oracle (Section 13.1.2), is built for the reverse needles  $p^1, \dots, p^k$ . Since the maximal safe shift width cannot be larger than the length  $m$  of shortest needle, the automaton considers only the prefixes of the needles that do not exceed that length. During the search, the automaton processes a part  $t_{pos+1} \dots t_{pos+m}$  of the haystack from back to front. If the whole substring can be processed, MULTIBFAM tests all needles in *verify* $[q]$ ,

```

1  ▷ WUMANBER ( $P = \{p^1, \dots, p^k\}, t_1 \dots t_n$ )
2   $m \leftarrow$  minimum length of  $p^j$ 
3   $z \leftarrow m - q + 1$ 
4   $shift[w] \leftarrow z$  for all  $w \in \Sigma^q$ 
5  for  $i \leftarrow 1$  to  $z$  do
6  |   for  $j \leftarrow 1$  to  $k$  do
7  |   |    $shift[p_i^j \dots p_{i+q-1}^j] \leftarrow z - i$ 
8   $verify[w] \leftarrow \{\}$  for all  $w \in \Sigma^q$ 
9  for  $j \leftarrow 1$  to  $k$  do
10 |    $verify[p_z^j \dots p_m^j] \leftarrow verify[p_z^j \dots p_m^j] \cup \{j\}$ 
11  $pos \leftarrow 0$ 
12 while  $pos \leq n - m$  do
13 |    $w \leftarrow t_{pos+z} \dots t_{pos+m}$ 
14 |   if  $shift[w] = 0$  then
15 |   |   for each  $j \in verify[w]$  do
16 |   |   |    $\text{report if } p^j \text{ matches } t \text{ at } pos + 1$ 
17 |   |    $pos \leftarrow pos + 1$ 
18 |   else
19 |   |    $pos \leftarrow pos + shift[w]$ 

```

} build *shift*  
 } build *verify*  
 } searching

Algorithm 14: Wu-Manber Searching of Multiple Needles.

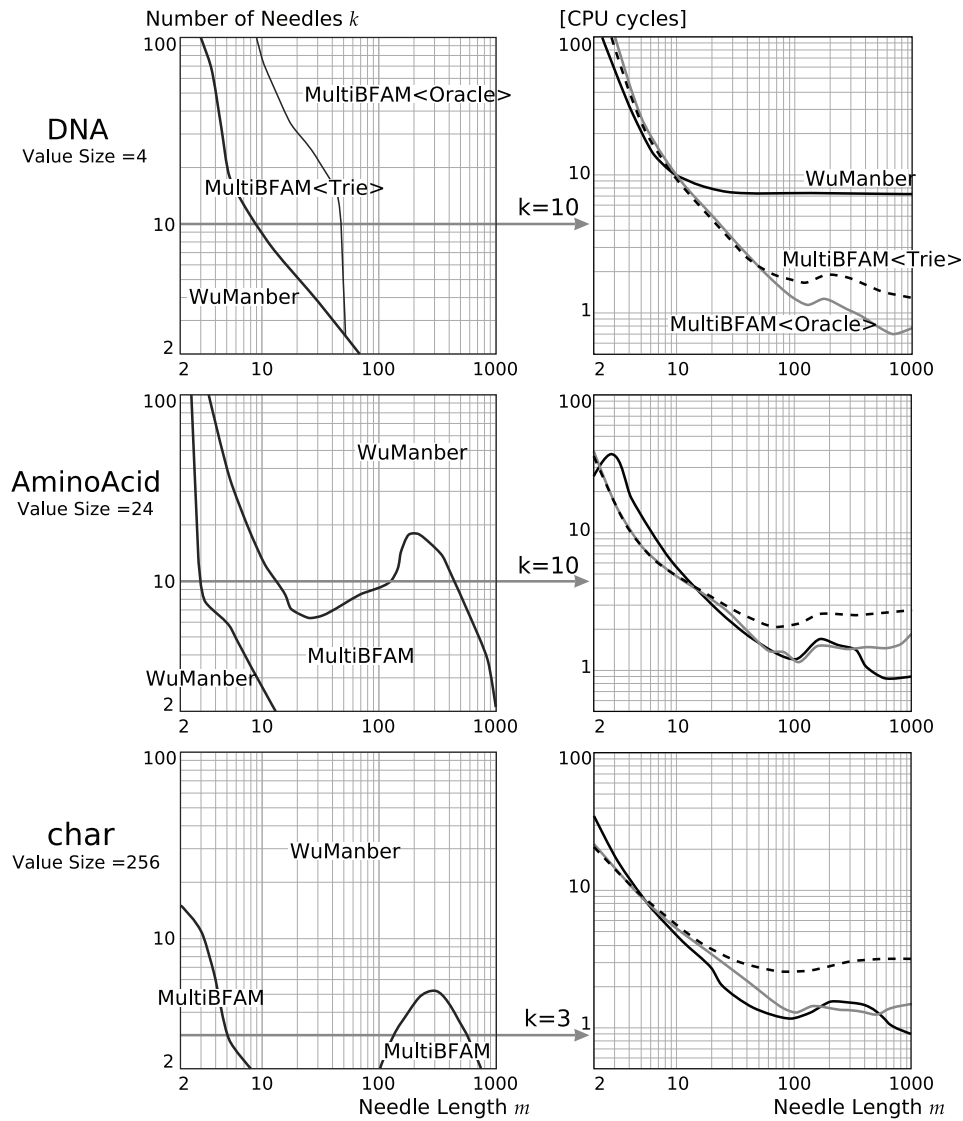


Figure 29: **Fastest Multiple Pattern Matching Algorithm.** **Left:** The optimal algorithm for finding  $k$  patterns of length  $m$  in parts of (1) the Escherichia Coli genome (2) proteins from the Swiss-Prot database, and (3) the English Bible. If  $|\Sigma| = 4$ , then WuManber is optimal for small  $m$  and MultiBFAM for large  $m$ . For larger alphabets, the reverse is true. **Right:** Slices through the left figures show the actual run time per searched haystack value for searching  $k = 10$  (Dna or AminoAcid) or  $k = 3$  (char) patterns. Both algorithms are sublinear, so for large  $m$  the run times may even fall below 1 CPU cycles per haystack value.



which gives for the current automaton state  $q$  the list of needles with prefix  $t_{pos+1} \dots t_{pos+m}$ .

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19	<pre> ▷ MULTIBFAM (<math>P = \{p^1, \dots, p^k\}, t_1 \dots t_n</math>) <math>m \leftarrow</math> minimum length of <math>p^j</math> <math>rev^j \leftarrow p_m^j \dots p_1^j</math> for <math>j \in \{1, \dots, k\}</math> <math>a \leftarrow \text{BUILDFACTORAUTOMATON}(rev^1, \dots, rev^k)</math> <math>verify[q] \leftarrow \{\}</math> for all states <math>q</math> in <math>a</math> <b>for</b> <math>j \leftarrow 1</math> <b>to</b> <math>k</math> <b>do</b>   <math>q \leftarrow</math> the state of <math>a</math> after processing <math>rev^j</math>   <math>verify[q] \leftarrow verify[q] \cup \{j\}</math> <math>pos \leftarrow 0</math> <b>while</b> <math>pos \leq n - m</math> <b>do</b>   <math>q \leftarrow \delta_a(\text{initial state of } a, t_{pos+m})</math>   <math>k \leftarrow m</math>   <b>while</b> <math>q</math> is defined <b>do</b>     <b>if</b> <math>k = 1</math> <b>then</b>       <b>for each</b> <math>i \in verify[q]</math> <b>do</b>         <math>\text{report if } p^i \text{ matches } t \text{ at } pos + 1</math>         <b>break</b>       <math>k \leftarrow k - 1</math>     <math>q \leftarrow \delta_a(q, t_{pos+k})</math>   <math>pos \leftarrow pos + k</math> </pre>	<div style="display: flex; align-items: center;"> <div style="font-size: 3em; margin-right: 5px;">}</div> <div>build <math>a</math></div> </div> <div style="display: flex; align-items: center; margin-top: 10px;"> <div style="font-size: 3em; margin-right: 5px;">}</div> <div>build <math>verify</math></div> </div> <div style="display: flex; align-items: center; margin-top: 10px;"> <div style="font-size: 3em; margin-right: 5px;">}</div> <div>searching</div> </div>
---	---	--

Algorithm 15: **Backward Factor Automaton Matching for Multiple Needles.**

## 10.3 Approximate Searching

So far, we discussed *exact* matching, that means a match of the search is a substring  $s$  of the haystack  $t$  that equals the pattern  $p$ . In this section, we will relax this condition, such that a match  $s$  needs not to be equal but only sufficiently ‘similar’ to  $p$ . More precisely spoken, we want to find all substrings  $s$  of  $t$  for which the distance  $dist(s, p)$  with respect to a certain distance metric  $dist$  does not exceed a certain threshold  $T$ . This is called *approximate string matching*. In its most general form,  $dist$  could be any sequence distance measure based on alignment scores as described in Section 9.3.1, though most approximate search algorithms are specialized for *edit distance*. SeqAn also supports approximate search algorithms that only allow mismatches between  $s$  and  $p$  but no inserts or deletes; we will describe them in Section 10.4.1.

When an approximate searching algorithm searches the haystack  $t$  starting from a position  $pos$ , then we can either search for matching substrings of  $t_{pos} \dots t_n$  (*infix search*), or for matching prefixes of  $t_{pos} \dots t_n$  (*prefix search*). We will focus on infix search in this section. SeqAn offers two algorithms for prefix search, namely the algorithms by Sellers and Myers that can be easily adapted for prefix search.

In SeqAn, finding all infix hits  $s$  in  $t$  is done in two steps:

- (1) First, the function `find` looks for a position in  $t$  at which a match ends. The threshold  $T$  is either set by calling `setScoreLimit` or simply passed to `find` as a third function argument.
- (2) If a match was found, the function `findBegin` can be used to search its begin position. The threshold  $T$  for that approximate search can be passed as a third argument, otherwise the function applies the same threshold as during the last call of `find`. Subsequent calls of `findBegin` may be used to find several begin positions to the same end position.

Technically, `findBegin` is implemented as a prefix search on the *reverse* needle and haystack strings. Listing 19 demonstrates how to use `find` and `findBegin`.

```
String<char> t = "babybanana";
String<char> p = "babana";
Finder<String<char> > finder(t);
Pattern<String<char>, Myers<FindInfix> > pattern(p);
while (find(finder, pattern, -2))
{
    std::cout << "end: " << endPosition(finder) << std::endl;
    while (findBegin(finder, pattern, getScore(pattern)))
    {
        std::cout << "begin: " << beginPosition(finder) << std::endl;
        std::cout << infix(finder) << " matches with score "
                    << getBeginScore(pattern) << std::endl;
    }
}
```

Listing 19: **Approximate String Searching Example.** The program finds six substrings "babyba", "byban", "bybana", "banan", "bybanan", and "banana" of the haystack "babybanana" that match the needle "babana" with at least two errors (edit distance). Note that the two matches "banan" and "bybanan" both end at the same position 9. The third argument of `findBegin` is optional; the default is the score limit  $T$  of the last call of `find`, i.e.  $-2$  in this example. If we use this, six more matches would be found.

The algorithms for approximate string matching supported by SeqAn are listed in Table 15. They are also described in the book of Navarro and Raffinot (2002). Another good survey of approximate string matching algorithms can be found in (Navarro 2001).

<b>DPSearch</b>	An algorithm by Sellers (1980) that is based on the dynamic programming algorithm for sequence alignment by Needleman and Wunsch (1970). It can also be used for prefix search.
<b>Myers</b>	A fast searching algorithm for edit distance using bit parallelism by Myers (1999). It can also be used for prefix search.
<b>Pex</b>	A filtering technique by Navarro and Baeza-Yates (1999) that splits the needle into $k + 1$ pieces and search these pieces exact in the haystack.
<b>AbndmAlgo</b>	Approximate Backward Nondeterministic DAWG Matching, an adaption of the BNDM algorithm for approximate string matching.

Table 15: **Approximate Pattern Matching.** Specializations of **Pattern**.

### 10.3.1 Sellers' Algorithm

The algorithm by Sellers (1980) resembles the dynamic programming alignment algorithm (Needleman and Wunsch 1970) with free start gaps for the haystack  $t$ , as it was described in Section 9.5.4. Remember that  $\text{FILLMATRIX}(p_1 \dots p_m, t_1 \dots t_n)$  (see Algorithm 2 on page 89) uses a matrix  $M$ , where  $M_{i,j}$  is the optimal score for aligning the two prefixes  $p_1 \dots p_i$  and  $t_1 \dots t_j$ . If we initialize  $M_{0,j} \leftarrow 0$  for all  $j \in \{1, \dots, n\}$ , then each  $M_{i,j}$  is filled with the optimal score of alignments between  $p_1 \dots p_i$  and a suffix of  $t_1 \dots t_j$ . The  $j$ -th pass of the outer loop in **SELLERS** (Algorithm 16) computes the  $j$ -th column  $C$  of the matrix  $M$ . The inner loop computes in line 7 the value  $C_i = M_{i,j}$  according to Equation 9.6 (page 87). The variable  $d$  was previously set to  $M_{i-1,j-1}$  (case 1),  $v$  to  $M_{i-1,j}$  (case 2), and  $h$  to  $M_{i,j-1}$  (case 3). At the end of the inner loop (line 10), the value  $v = C_m = M_{m,j}$  is the optimal score of an alignment between  $p = p_1 \dots p_m$  and a suffix  $s$  of  $t_1 \dots t_j$ . If the sequence distance  $-v$  between  $s$  and  $p$  is  $\leq T$ , then  $s$  is an approximate match and its end position  $j$  is reported.

**SELLERS** can easily be extended to support *affine gap costs* following Gotoh's idea (Gotoh 1982), which we described in Section 9.5.2. SeqAn supports both variants for linear and for non-linear gap costs, and selects it according to the applied scoring scheme.

The algorithm can also be adapted for *prefix searching*, we just have to change the initialization to make start gaps in the text non-free. That is, we change the lines 3 and 4 in **SELLERS** to ' $v \leftarrow i \times g$ ' and ' $d \leftarrow (i - 1) \times g$ '.

### Ukkonen's Trick

Sellers' algorithm takes time  $O(nm)$  for finding the occurrences of a pattern  $p_1 \dots p_m$  in a text  $t_1 \dots t_n$ . With a slightly modification (Ukkonen 1985), this can be accelerated for *edit distance* scoring to  $O(km)$  on average, where  $k = -T$

```

1  ▷ SELLERS ( $p_1 \dots p_m, t_1 \dots t_n, T$ )
2   $C_i \leftarrow i \times g$  for each  $i \in \{1, \dots, m\}$ 
3  for  $j \leftarrow 1$  to  $n$  do
4       $v \leftarrow 0$ 
5       $d \leftarrow 0$ 
6      for  $i \leftarrow 1$  to  $m$  do
7           $h \leftarrow C_i$ 
8           $v \leftarrow \max\{d + \alpha(p_i, t_j), \max\{v, h\} + g\}$ 
9           $C_i \leftarrow v$ 
10          $d \leftarrow h$ 
11     if  $-v \leq T$  then report match end position
12      $j$ 

```

} Compute in  $C$   
the  $j$ -th  
column of the  
matrix  $M$

Algorithm 16: **Sellers' Algorithm.**  $\alpha$  returns the score of aligning two values;  $g$  is the (usual negative) gap score.

is the number of allowed errors per match. The ‘trick’ is to compute just the cells  $C_i$  for  $i \leq i_0$ , where  $i_0$  is minimal such that  $C_{i'} < T$  for all  $i' > i_0$ . At the beginning of SELLERS  $C_i$  is initialized to  $C_i = ig = -i$ , so we set  $i_0 \leftarrow -T$ . Suppose that we know an  $i_0$  for a given column  $j$ ; one can easily prove that for the next column  $j + 1$  holds  $C_{i'} < T$  for all  $i' > i_0 + 1$ , i.e. the  $i_0$  must be increased by at most one. After computing the values  $C_0, C_1, \dots, C_{i_0}, C_{i_0+1}$ , we can easily calculate the actual  $i_0$  in (amortized) constant time.

### 10.3.2 Myers' Bitvector Algorithm

Myers (1999) uses bit parallelism to speed up Sellers' algorithm for *edit distance*.<sup>1</sup> Remember that the edit distance between two sequences is the negative score of their optimal alignment where each match scores 0 and each mismatch and gap scores -1 (see Section 9.3.1). The main idea of this algorithm is to encode the  $j$ -th column of the matrix  $M$  of the Needleman-Wunsch algorithm (Algorithm 2) in five bit vectors, each of length  $m$  (the length of the needle):

$$\begin{aligned}
 VP_i &:= (M_{i,j} = M_{i-1,j} - 1) & VN_i &:= (M_{i,j} = M_{i-1,j} + 1) \\
 HP_i &:= (M_{i,j} = M_{i,j-1} - 1) & HN_i &:= (M_{i,j} = M_{i,j-1} + 1) \\
 D0_i &:= (M_{i,j} = M_{i-1,j-1})
 \end{aligned}$$

where  $i \in \{1, \dots, m\}$ . In each pass of the main loop, MYERS (Algorithm 17) computes these five vectors for column  $j$  based on of the vectors for column  $j - 1$ , which takes 15 bit vector operations. The details are explained in Appendix A.1

<sup>1</sup>We present here the variant by Hyyro and Fi (2001).

If the bit vectors does not fit into one machine word, then several machine words per bit vector must be used to store  $VP$  and  $VN$  – all other bit vectors need not to be stored completely. MYERS also keeps track of the current score  $score$  and reports the position  $j$  whenever it climbs above the (negative) score limit  $T$ , i.e. the number of errors falls below  $-T$ .

<div style="border: 1px solid black; padding: 10px; min-height: 300px;"> <div style="text-align: right; margin-bottom: 5px;">▷ MYERS (<math>p_1 \dots p_m, t_1 \dots t_n, T</math>)</div> <div style="margin-bottom: 5px;">1 <math>mask[c] \leftarrow 0^m</math> for all <math>c \in \Sigma</math></div> <div style="margin-bottom: 5px;">2 <b>for</b> <math>i \leftarrow 1</math> <b>to</b> <math>m</math> <b>do</b> <math>mask[p_i]_i \leftarrow 1</math></div> <div style="margin-bottom: 5px;">3 <math>VP \leftarrow 1^m</math></div> <div style="margin-bottom: 5px;">4 <math>VN \leftarrow 0^m</math></div> <div style="margin-bottom: 5px;">5 <math>score \leftarrow -m</math></div> <div style="margin-bottom: 5px;">6 <b>for</b> <math>j \leftarrow 1</math> <b>to</b> <math>n</math> <b>do</b></div> <div style="margin-left: 20px; margin-bottom: 5px;">7 <math>X \leftarrow mask[t_j] \vee VN</math></div> <div style="margin-left: 20px; margin-bottom: 5px;">8 <math>D0 \leftarrow ((VP + (VP \wedge X)) \oplus VP) \vee X</math></div> <div style="margin-left: 20px; margin-bottom: 5px;">9 <math>HN \leftarrow VP \wedge D0</math></div> <div style="margin-left: 20px; margin-bottom: 5px;">10 <math>HP \leftarrow VN \vee \neg(VP \vee D0)</math></div> <div style="margin-left: 20px; margin-bottom: 5px;">11 <math>Y \leftarrow HP \ll 1</math></div> <div style="margin-left: 20px; margin-bottom: 5px;">12 <math>VN \leftarrow Y \wedge D0</math></div> <div style="margin-left: 20px; margin-bottom: 5px;">13 <math>VP \leftarrow (HN \ll 1) \vee \neg(Y \vee D0)</math></div> <div style="margin-left: 20px; margin-bottom: 5px;">14 <b>if</b> <math>HP_m</math> <b>then</b> <math>score \leftarrow score - 1</math></div> <div style="margin-left: 20px; margin-bottom: 5px;">15 <b>else if</b> <math>HN_m</math> <b>then</b> <math>score \leftarrow score + 1</math></div> <div style="margin-left: 20px; margin-bottom: 5px;">16 <b>if</b> <math>score \geq T</math> <b>then</b></div> <div style="margin-left: 40px; margin-bottom: 5px;">17   <math>\text{report match end position } j</math></div> </div>	<div style="font-size: 3em; line-height: 1; padding: 0 10px;">}</div> <div style="text-align: left;">preprocessing</div> <div style="font-size: 3em; line-height: 1; padding: 0 10px;">}</div> <div style="text-align: left;">searching</div>
---	---

Algorithm 17: Myers' Bit-Vector Algorithm.

MYERS would perform a prefix search if we force  $M_{0,j} = M_{0,j-1} - 1$ , i.e.  $HP_0 = 1$ . This can simply be done by changing line 11 to  $Y \leftarrow HP \ll 1|1$ . SeqAn contains two implementations of Myers' algorithm, one for needle length up to one machine word, and a second for longer needles. The algorithm is the fastest approximate searching function in SeqAn for high edit distances (e.g.  $< 60\%$  identity, see Figure 30).

We combined Myers' algorithm with Ukkonen's Trick (Section 10.3.1), so the algorithm will usually compute only the first machine word of the bit vectors at all positions in the haystack but those at which it has regions very similar to the pattern. This makes the average running time roughly independent from the pattern length  $m$ , see Figure 30.

### 10.3.3 Partition Filtering

In this section, we will discuss an algorithm by Navarro and Baeza-Yates (1999) that bases on a simple idea proposed by Wu and Manber (1992): Let  $T$  be

the threshold for the edit distance score when searching the needle  $p$  in the haystack  $t$ , that is we want to find all occurrences of  $p$  in  $t$  with  $\leq k = -T$  errors. If we cut  $p$  into  $k + 1$  pieces, then each approximate match of  $p$  in  $t$  must contain at least one of these pieces unchanged. So we start with an exact multi-pattern search for the pieces and then ‘verify’ each occurrence of one of these pieces, that is we try to find  $p$  in the neighborhood of the found piece, see Algorithm 18. The following lemma guarantees that this approach works:

**Lemma 1 (Pidgeonhole Principle)**

Let  $a^1 \dots a^l = a$  be a partition of the string  $a$  into  $m$  substrings, and let  $r^1, \dots, r^l$  be  $l$  positive integer numbers. If  $a$  matches to a string  $b$  with less than  $r^1 + r^2 + \dots + r^l$  errors, then at least one  $a^i$  matches with less than  $r^i$  errors to a substring of  $b$ .

The lemma follows directly from the fact that, for linear gap costs, the score of an alignment is the sum of the scores of its pieces. Hence, if each  $a^i$  matches to its counterpart in  $b$  with  $\geq r^i$  errors, then  $a$  and  $b$  would match only with  $\geq r^1 + r^2 + \dots + r^m$  errors.

In our case, we use the lemma with  $l = k + 1$  and  $r^1 = r^2 = \dots = r^l = 1$ .

1 2 3 4 5 6	<pre> ▷ PEX (<math>p = p_1 \dots p_m, t = t_1 \dots t_n, -k</math>) divide <math>p</math> into <math>k + 1</math> parts <math>p^1 \dots p^{k+1}</math> <math>pieces \leftarrow \text{FINDEXACT}(\{p^1, \dots, p^{k+1}\}, t)</math> <b>for each</b> <math>(i, pos) \in pieces</math> <b>do</b>     let <math>p^i = p_i \dots p_{i+m}</math>     <math>hits \leftarrow \text{FINDAPPROX}(p, t_{pos-l-k} \dots t_{pos+m-k}, -k)</math>     report all matches in <math>hits</math> </pre>	} verification
----------------------------	--	----------------

Algorithm 18: **Partition Filtering Algorithm.** FINDEXACT could be any exact searching algorithm for multiple needles, e.g. WUMANBER or MULTIBFAM, and FINDAPPROX any other stand-alone approximate string matching algorithm, e.g. SELLERS or MYERS.

We call this technique *filtering*, since all parts of the haystack that do not contain any piece of the pattern are ‘filtered out’ and only the rest is passed to the verification process. This works well for long patterns and small error rates, since in this case the pieces are relatively long, so the expected false positive rate of the filtering is low. A good partitioning strategy is to split the needle  $p_1 \dots p_m$  into  $k + 1$  pieces of approximately equal length  $\geq \lfloor m/(k + 1) \rfloor$ . On the other hand, the costs for each verification move up for large patterns. For that reason, Navarro and Baeza-Yates (1999) applied an optimization called ‘*hierarchical verification*’: Let  $\mathcal{T}$  be a tree with  $k + 1$  leafs that are labeled from  $p^1$  to  $p^{k+1}$ . We use a balanced binary tree, but the idea works for any tree topology. We label each vertex  $v$  of  $\mathcal{T}$  with the concatenated pieces

of  $p$  on the leafs of the subtree rooted in  $v$ , and we call this label  $p(v)$  and the number of leafs in this subtree  $r(v)$ . The root of  $\mathcal{T}$  is therefore labeled with  $p$ . Suppose that we find in the haystack at position  $pos$  an exact match of  $p^i$ , then we follow the path from the  $i$ -th leaf to the root. At each vertex  $v$  of this path, we search  $p(v)$  in the neighborhood of  $pos$  with less than  $r(v)$  errors. If no occurrence of  $p(v)$  is found, the verification stops, otherwise we proceed with parent of  $v$  in  $\mathcal{T}$  until the root has been reached and a match of  $p$  was verified.

It is easy to prove that this kind of verification is correct, i.e. no approximate match gets lost: Let  $v_1, \dots, v_l$  be the children of the root, then  $r(v_1) + \dots + r(v_l) = k + 1$ . Hence, according to Lemma 1, any approximate match of  $p$  with at most  $k$  errors also contains some  $p(v^i)$  with less than  $r(v^i)$  errors. The same argument can recursively applied to  $v^i$ , then to one of its children, and so forth.

## 10.4 Other Pattern Matching Problems

There are many more variants of pattern matching problems, and SeqAn provides algorithms for some of them. In this section, we will describe two of them: (1) the *k-mismatch problem* and (2) searching with *wildcards*.

### 10.4.1 k-Mismatch Searching

A *mismatch* between a sequence  $a = a_1 \dots a_n$  and another sequence  $b = b_1 \dots b_n$  is a position  $i \in \{1, \dots, n\}$  such that  $a_i \neq b_i$ . The number of mismatches between  $a$  and  $b$  is called the ‘*Hamming distance*’ of the two sequences. Given a needle  $p_1 \dots p_m$  and a haystack  $t_1 \dots t_n$ , then *searching with k mismatches* means to find all substrings  $t_{pos+1} \dots t_{pos+m}$  that have Hamming distance to  $p_1 \dots p_m$  of  $\leq k$ . This kind of searching resembles approximate string matching as described in Section 10.3 but without inserts or deletes, gaps are forbidden. Sellers’ algorithm (Section 10.3.1) can be used for searching with mismatches if the costs for gaps are set to  $+\infty$ ; but of course there are also algorithms especially for the *k-mismatch problem*. SeqAn for example offers the specialization `HammingHorspool` of `Pattern`, that implements an adaption of the exact pattern matching algorithm (Section 10.1.2) proposed by Tarhio and Ukkonen (1990).

### 10.4.2 Searching with Wildcards

SeqAn provides the algorithm `WildShifAnd` that is able to search for regular expressions (e.g. Navarro and Raffinot (2002), 4.5.1). Note that this algorithm does not support the complete functionality of usual regular expressions, e.g. it does not support alternatives, and quantifiers like  $*$ ,  $+$ ,  $?$  and  $\{i, j\}$  can

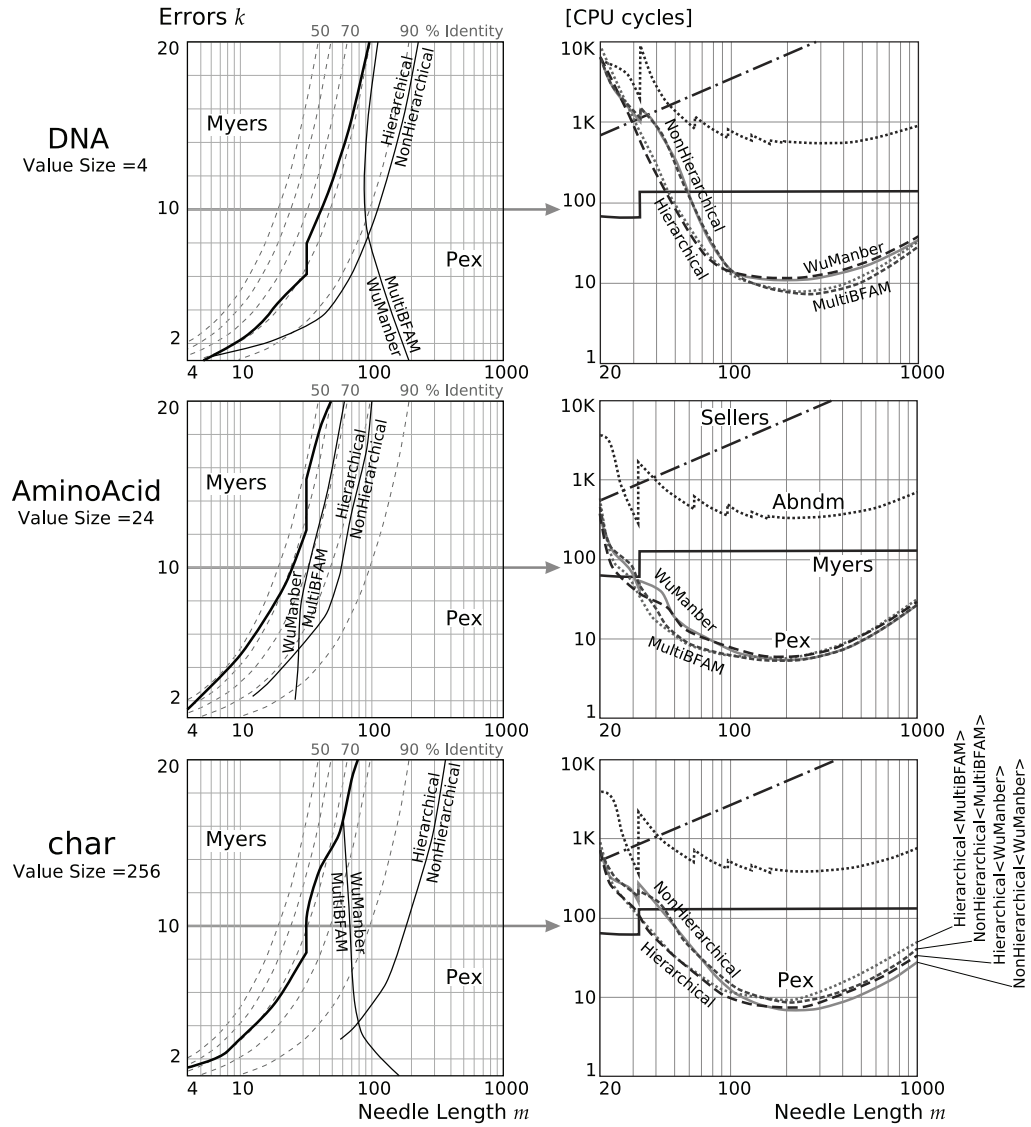


Figure 30: **Fastest Approximate Matching Algorithm.** **Left:** The optimal algorithm for finding a pattern of length  $m$  with at most  $k$  errors (*edit distance*) in parts of (1) the Escherichia Coli genome (2) proteins from the Swiss-Prot database, and (3) the English Bible. **Right:** Slices through the left figures show the actual run time per searched haystack value for  $k = 10$



refer only to single characters or character classes. A pattern for searching in texts  $t_1 \dots t_n$  of the alphabet  $\Sigma$  is a string  $c^1 \dots c^m$  that is the concatenation of *clauses*, where each clause  $c^i$  has one of the forms in Table 16. For example, the pattern

$$[A - Z0 - 9] . *$$

matches to all strings that start with a capital letter or a digit.

Clause	Description
$w$	a string
$.$	any character
$a\{i, j\}$	repeat $a$ at least $i$ and at most $j$ times
$a^*$	repeat $a$ for 0, 1, or more times
$a^+$	repeat $a$ for 1 or more times
$a^?$	optional $a$ , same as $a\{0, 1\}$
$[b^1 \dots b^k]$	a character in a class: Each $b^j$ is either a set of characters from $\Sigma$ , or it has the form ' $a_1 - a_2$ ' which denotes all characters in $\Sigma$ between $a_1$ and $a_2$ .

Table 16: **Regular Expression Syntax.** A regular expression may contain several clauses.  $w$  is a string of values from  $\Sigma$ , and  $a$  is either a single character from  $\Sigma$  or a character class  $[b^1 \dots b^k]$ .



# Chapter 11

## Motif Discovery

*Motif finding* means to find matching substrings of  $d \geq 2$  given sequences  $a^1, \dots, a^d$ , where ‘matching’ can either be meant to be *exact*, i.e. the matching substrings must be exactly the same, or *approximate*, i.e. differences between the substrings are allowed. The matching substrings are called ‘*motifs*’. In the following, we will first concentrate on the *pairwise motif finding* problem, that is finding motifs in  $d = 2$  sequences. In contrast to *pattern matching* (see Chapter 10), where we search for a *complete* needle sequence in a haystack sequence, (pairwise) motif addresses the problem of finding *parts* of the needle within the haystack. In most cases, we are only interested in motifs that fulfill certain criteria of quality, for example a minimal length, a minimum alignment score or – in the case of approximate motif finding – a maximum mismatch count.

SeqAn offers algorithms for solving various kinds of motif finding problems which are spread over several modules of the library:

- **Local Alignments:** Algorithms for solving the global alignment problem (see Section 9.5), that is to find an optimal alignment between two *complete* sequences, can be adapted for motif finding, that is to find optimal alignments between *substrings* of two sequences. This is called ‘*local alignment*’, and we describe it in Section 11.1. The motifs found by this search are local alignments stored in alignment data structures, see Section 9.2.
- **Index Iterators:** Some index data structures, as for example the *suffix trees* or the *enhanced suffix arrays* (ESA), can be used to find exact matches between two or more sequences. SeqAn offers some special iterators (see Section 12.3.2) that can be used to browse through all exact motifs, which are defined by the begin positions and the length of the matching substrings.
- **Seed Based Motif Search:** Algorithms for expanding and combining small motifs (so called ‘seeds’) to larger motifs are introduced in the

Section 11.2. Seeds are represented essentially by the begin and end positions of the matching substrings, which can be stored in a data structure called **Seed**, see Section 9.6.1.

- **Multiple Sequence Motifs:** Algorithms for finding subtle motifs of fixed length in multiple sequences are discussed in Section 11.3. Motifs of this kind are either represented by a consensus sequence or by a position dependent weight matrix.

## 11.1 Local Alignments

### 11.1.1 Smith-Waterman Algorithm

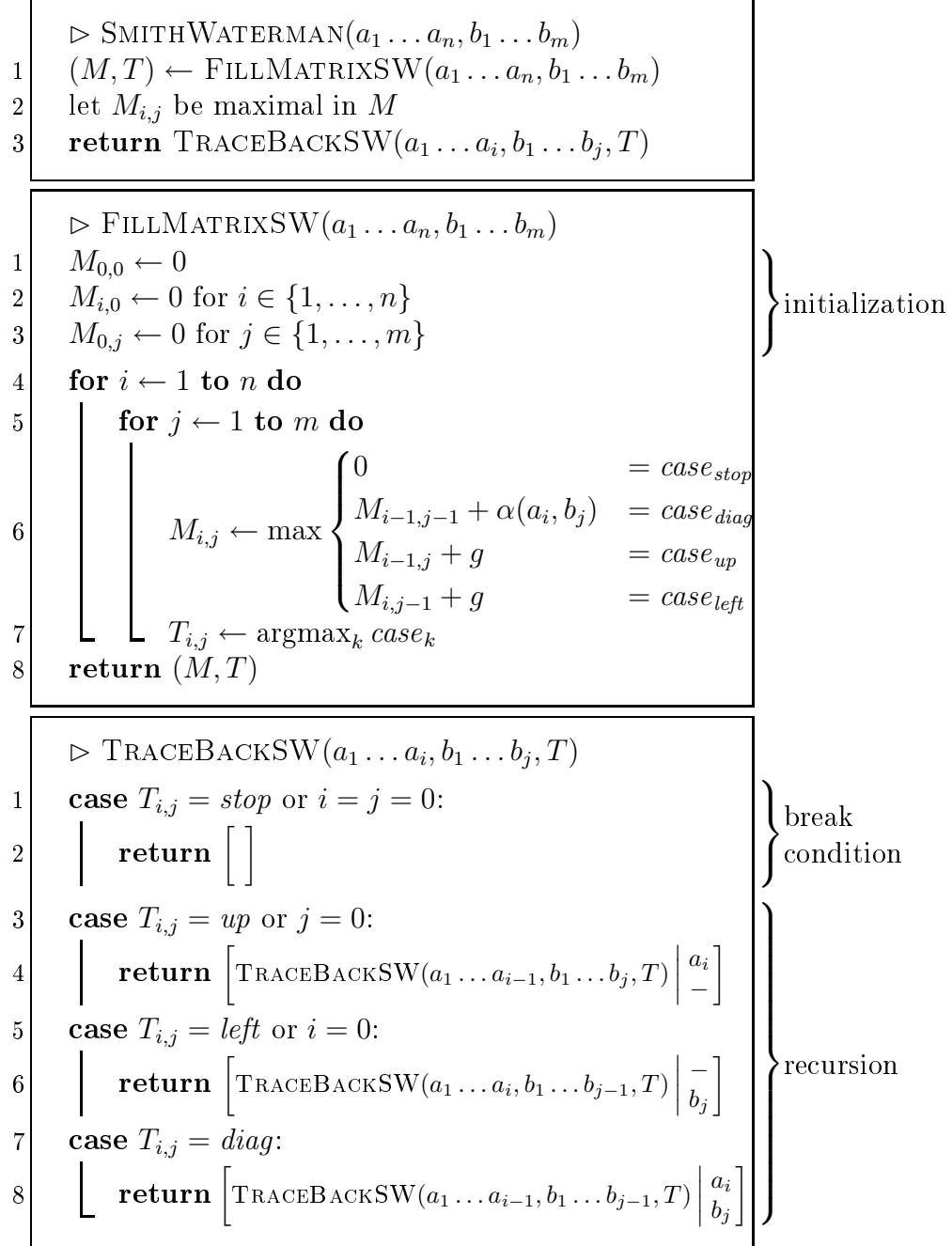
Smith and Waterman 1981 adapted the Needleman-Wunsch algorithm (Needleman and Wunsch (1970), see Section 9.5.1), for finding motifs in two sequences  $a$  and  $b$ : The algorithm finds a substring  $a'$  of  $a$  and a substring  $b'$  of  $b$  and an alignment  $\mathcal{A}$  between  $a'$  and  $b'$ , such that the score of  $\mathcal{A}$  is as least as good as the score of any other alignment between a substring of  $a$  and a substring of  $b$ . In this case, we call  $\mathcal{A}$  an ‘optimal *local alignment*’ between  $a$  and  $b$ .

The Smith-Waterman algorithm works as follows: SMITHWATERMAN (Algorithm 19) computes – just like NEEDLEMANWUNSCH (Algorithm 2) on page 89 – an  $m \times n$  matrix  $M$  of scores, but other than the Needleman-Wunsch algorithm that set  $M_{i,j}$  to the score of the optimal alignment between  $a_1 \dots a_i$  and  $b_1 \dots b_j$ , the Smith-Waterman algorithm computes instead the optimal score of any alignment between a *suffix* of  $a_1 \dots a_i$  and a *suffix* of  $b_1 \dots b_j$ . Let these suffixes be  $a'$  and  $b'$  for a given  $i$  and  $j$ . Note that  $a'$  or  $b'$  or both could be the empty string  $\epsilon$ . If both  $a' = b' = \epsilon$ , then  $M_{i,j} = \text{score}(\epsilon, \epsilon) = 0$ . Otherwise,  $M_{i,j}$  can be computed by recursion (9.6) on page 87. If either  $i = 0$  or  $j = 0$ , then obviously  $M_{i,j} = 0$ . After filling the complete matrix  $M$ , we get an optimal local alignment by starting a trace back at a cell  $M_{i,j}$  with a maximal value, which is the score of the optimal local alignment.

```
Align< String<char> > ali;
appendValue(rows(ali), "aphilologicaltheorem");
appendValue(rows(ali), "bizarreamphibology");
Score<int> scoring(3,-3,-2);
int score = localAlignment(ali, scoring, SmithWaterman());
cout << ali;
```

Listing 20: **Smith-Waterman Algorithm.** Finding an optimal local alignment by using the Smith-Waterman algorithm.

In SeqAn, this algorithm can be used by calling the function `localAlignment`, see Listing 20 for an example.



Algorithm 19: **Smith-Waterman Algorithm.**  $\alpha(a_i, b_j)$  is the score for aligning  $a_i$  and  $b_j$ ,  $g$  is the score for a blank.

### 11.1.2 Waterman-Eggert Algorithm

Sometimes also *suboptimal local alignments* between two sequences  $a_1 \dots a_m$  and  $b_1 \dots b_n$  are of interest. Waterman and Eggert (1987) modified the Smith-Waterman algorithm such that it computes *non-intersecting* local alignments between two sequences. We say that two alignments do not *intersect*, if they have no match or mismatch in common.

```

1  ▷ WATERMANEGGERT( $a_1 \dots a_n, b_1 \dots b_m, limit$ )
2  ( $M, T$ ) ← FILLMATRIXSW( $a_1 \dots a_n, b_1 \dots b_m$ )
3  repeat
4    let  $M_{i,j}$  be the maximal cell in  $M$  not yet used
5    if  $M_{i,j} < limit$  then break
6     $\mathcal{A} \leftarrow$  TRACEBACKSW( $a_1 \dots a_i, b_1 \dots b_j, T$ )
7    Report  $\mathcal{A}$ 
   Recompute  $M$  and  $T$  following  $\mathcal{A}$ 

```

Algorithm 20: **Waterman-Eggert Algorithm.**

WATERMANEGGERT (Algorithm 20) repeatedly calls TRACEBACKSW on different prefixes  $a_1 \dots a_i$  and  $b_1 \dots b_j$  for decreasing  $M_{i,j}$ . Each call computes an alignment of score  $M_{i,j}$ , and the algorithm stops as soon as the score falls below a certain limit. To ensure that the computed local alignments do not intersect, the algorithm modifies  $M$  and  $T$  after each call of TRACEBACKSW: Suppose that the algorithm just computed a local alignment  $\mathcal{A}$  which aligns the two characters  $a_i$  and  $b_j$ . Then subsequent local alignments must not align  $a_i$  and  $b_j$ , so we need to recompute  $M_{i,j}$  and  $T_{i,j}$  such that *case<sub>diag</sub>* (see lines 6 and 7 in FILLMATRIXSW) is forbidden there. If we change  $M_{i,j}$ , then we possibly also need to change  $M_{i+1,j}$ ,  $M_{i,j+1}$ , or  $M_{i+1,j+1}$ . Waterman and Eggert recalculate only the part of  $M$  that need to be updated by enumerating them from top left to bottom right.

Listing 21 shows how to compute non-intersecting suboptimal alignments in SeqAn. Each call of function `localAlignmentNext` performs one step of the Waterman-Eggert algorithm to compute the next best local alignment.

## 11.2 Seed Based Motif Search

Finding exact motifs is relatively easy. For example, we will show in Section 12.1 how to use index data structures to find all common  $q$ -grams between sequences in linear time. Many efficient heuristics to find high scoring but *inexact* local alignments therefore start with such small exact (or at least highly similar) motifs, so called ‘*seeds*’, and extend or combine them to get

```

Align< String<Dna> > ali;
appendValue(rows(ali), "ataagcgtctcg");
appendValue(rows(ali), "tcatagagttgc");

LocalAlignmentFinder<> finder(ali);
Score<int> scoring(2,-1,-2,0);
while (localAlignment(ali, finder, scoring, 2))
{
    cout << "Score=" << getScore(finder) << endl;
    cout << ali;
}

```

Listing 21: **Waterman-Eggert Algorithm.** Applying the Waterman-Eggert algorithm in SeqAn. The algorithm computes non-overlapping local alignments with scores better than 2.

larger motifs. Probably the most prominent tool of this kind is the ‘Basic Local Alignment Search Tool’ (BLAST) (Altschul et al. 1990), which we already discussed in Section 1.2.2, but there are many other examples like FASTA (Pearson 1990) or LAGAN (Brudno et al. 2003) (see Chapter 15).

SeqAn offers the class **Seed** for storing seeds, see Section 9.6.1. In this section, we will primarily use the specialization **SimpleSeed** of this class, which is especially designed for finding good motifs between two sequences ( $d = 2$ ). Suppose that we store a seed that corresponds to an alignment  $\mathcal{A}$  between the two substrings  $a_{left_0} \dots a_{right_0}$  and  $b_{left_1} \dots b_{right_1}$ , then beside the ‘borders’  $left_0$ ,  $right_0$ ,  $left_1$ , and  $right_1$ , **SimpleSeed** also knows two boundaries *lower* and *upper* for the ‘diagonal’  $j - i$  of any two aligned values  $a_i$  and  $b_j$  in  $\mathcal{A}$ , that is  $lower \leq j - i \leq upper$ . The function **bandedAlignment** can be used to retrieve an alignment for given a seed. It applies a variant of the Needleman-Wunsch algorithm (see Section 9.5.1) on  $a_{left_0} \dots a_{right_0}$  and  $b_{left_1} \dots b_{right_1}$  that is ‘banded’ by *lower* and *upper*, i.e. it only computes such values of the matrix  $M_{i,j}$  for which  $j - i$  lays within these boundaries.

There are two main tasks when processing seeds: Extending seeds to make them longer, and chaining several seeds together. In Section 11.2.1, we will describe how to extend seeds in SeqAn. The chaining of seeds to longer motifs will be the topic of Section 11.2.2. More details about seed based motif search in SeqAn can also be found in Kemena (2008).

### 11.2.1 Extending Seeds

Let  $\mathcal{S}$  be a seed. Then we call another seed  $\mathcal{E}$  an *extension* of  $\mathcal{S}$ , if for all  $i$  holds  $left_i(\mathcal{E}) \leq left_i(\mathcal{S})$  and  $right_i(\mathcal{S}) \leq right_i(\mathcal{E})$ . A good method for extending seeds should compute an extension  $\mathcal{E}$  that scores as high as possible for a given seed  $\mathcal{S}$ . SeqAn supports several algorithms for seed extension (see

Table 17). The function `extendSeed` extends a single seed while the function `extendSeeds` extends all seeds that are stored in a container. The user can determine the directions a seed will be extended, i.e. to the ‘left’ or to the ‘right’ or both. We will describe in the following only the extension to the ‘right’; the extension to the ‘left’ works similar.

<b>MatchExtend</b>	A simple extension algorithm that extends seeds until the first mismatch occurs.
<b>UngappedXDrop</b>	An $X$ -drop extension without gaps. The algorithm extends the seed until the score falls more than a given value $X$ .
<b>GappedXDrop</b>	An $X$ -drop extension variant of <code>UngappedXDrop</code> that also allows gaps in the extended seed.

Table 17: **Seed Extension Algorithms**

This simple extension method (see Algorithm 21) extends the seed until the first mismatch occurs. The algorithm does not create gaps. Listing 22 shows an example.

```
SEEDabXcdXefXXX
SEEDabYcdefYYYY
```

```
String<char> a = "SEEDabXcdXefXXX";
String<char> b = "SEEDabYcdefYYYY";
Seed<> seed(0, 0, 4); //left=0; length=4
extendSeed(seed, a, b, 1, MatchExtend());
cout << rightPosition(seed, 0) << endl; //output: 6
cout << rightPosition(seed, 1) << endl; //output: 6
```

Listing 22: **Match Extension Example.** The seed `SEED` is extended to the ‘right’ by `ab`; then the extension stops since `X` and `Y` do not match. The direction of the extension was selected by setting the fourth argument of `extendSeed` to 1.

```

1  ▷ MATCHEXTEND ( $a_1 \dots a_m, b_1 \dots b_n, right_0, right_1$ )
   while ( $a_{right_0+1} = b_{right_1+1}$ ) and
   |   ( $right_0 < m$ ) and ( $right_1 < n$ ) do
2   |    $right_0 \leftarrow right_0 + 1$ 
3   |    $right_1 \leftarrow right_1 + 1$ 
```

Algorithm 21: **Match Extension.**

`MATCHEXTEND` has the disadvantage that a single mismatch stops the extension immediately, so that subsequent matches are lost. Altschul et al. (1990)



therefore preferred an extension algorithm called ‘*X*-drop extension’ that allows some mismatches. An *X-drop* is a part of an alignment that scores  $\leq -X$  for a certain value  $X > 0$ , where  $X$  is called the ‘depth’ of the *X*-drop. The *X-drop extension* stops extending before the alignment ends in an *X*-drop. This guarantees that all drops in the extended part of the alignment have depth  $< X$ , thus the complete seed may contain an *X*-drop (but no  $2X$ -drop), especially if it was extended into both directions.

```
SEEDabXcdXefXXX
SEEDabYcdefYYYY
```

```
String<char> a = "SEEDabXcdXefXXX";
String<char> b = "SEEDabYcdefYYYY";
Seed<> seed(0, 0, 4);           //left=0; length=4
Score<> scoring(1, -1, -1);
extendSeed(seed, 2, scoring, a, b, 1, UngappedXDrop());
cout << rightPosition(seed, 0) << endl; //output: 9
cout << rightPosition(seed, 1) << endl; //output: 9
```

Listing 23: **Ungapped *X*-Drop Extension Example.** In this example, we set  $X = 2$  (this is the second argument of `extendSeed`). The seed **SEED** is extended to the ‘right’ by **abXcd** and **abYcd**.

<pre> 1 2 3 4 5 6 7 8 9 10 11 12 </pre>	<pre> ▷ UNGAPPEDXDROP (<math>a_1 \dots a_m, b_1 \dots b_n, right_0, right_1, X</math>)   score, best <math>\leftarrow 0</math>   <math>i \leftarrow 1</math>   while (<math>right_0 + i &lt; m</math>) and (<math>right_1 + i &lt; n</math>) do     score <math>\leftarrow</math> score <math>+</math> <math>\alpha(a_{right_0+i}, b_{right_1+i})</math>     if score <math>\leq</math> best <math>- X</math> then break     if score <math>&gt;</math> best then       <math>right_0 \leftarrow right_0 + i</math>       <math>right_1 \leftarrow right_1 + i</math>       best <math>\leftarrow</math> score       <math>i \leftarrow 1</math>     else       <math>i \leftarrow i + 1</math> </pre>	<pre>       } extend       } the seed </pre>
---	--	--

Algorithm 22: **Ungapped *X*-Drop Extension.** The function  $\alpha$  returns the score for aligning two values.

SeqAn supports an ungapped (Algorithm 22) and a gapped (Algorithm 23) variant of this algorithm. Listing 23 shows an example for ungapped *X*-drop extension.

For the gapped variant, SeqAn implements an algorithm described by Zhang et al. (2000) that applies dynamic programming similar to the Needleman-Wunsch algorithm (see Section 9.5.1). GAPPEDXDROP (see Algorithm 23) computes values  $M_{i,j}$  for  $i \geq \text{right}_0$  and  $j \geq \text{right}_1$ , where  $M_{i,j}$  is the score of the optimal alignment between  $a_{\text{right}_0+1} \dots a_i$  and  $b_{\text{right}_1+1} \dots b_j$ . The values  $M_{i,j}$  are computed in ascending order of their *antidiagonal*  $k = i + j$ . The highest score found so far is tracked in *best*. The algorithm limits the drop depth by setting all those  $M_{i,j}$  to  $-\infty$  that fall below  $\text{best} - X$ , which means that alignments going through  $M_{i,j}$  will not be continued. Instead, we only need to compute values  $M_{i,j}$  with  $L \leq i \leq U$ , where the bounds  $L$  and  $U$  are computed in lines 20 and 21 in a way that all relevant values are computed. The algorithm stops if either the diagonal  $n + m$  was reached, or all values in the last two antidiagonals  $k - 1$  and  $k$  were assigned to  $-\infty$ , since in this case that all further values would also be  $-\infty$ . The seed is then extended to the maximum  $M_{i,j}$ . Listing 24 shows how to use this algorithm in SeqAn.

SEED**abXcdXef**XXX  
SEED**abYcd-ef**YYYY

```
...
extendSeed(seed, 2, scoring, a, b, 1, GappedXDrop());
cout << rightPosition(seed, 0) << endl; //output: 12
cout << rightPosition(seed, 1) << endl; //output: 11
```

Listing 24: **Gapped X-Drop Extension Example.** The same as in Listing 23 but with GappedXDrop.

## 11.2.2 Combining Seeds

In this section we will show how to combine seeds to larger seeds. We discussed in Section 9.6 how seeds are threaded to get *global seed chains* by appending one seed to another. Remember that we can *append* a seed  $\mathcal{S}_j$  to another  $\mathcal{S}_k$ , if  $\text{right}_i(\mathcal{S}_k) \leq \text{left}_i(\mathcal{S}_j)$  for all  $i \in \{1, 2\}$ . It may also be that  $\text{left}_i(\mathcal{S}_k) \leq \text{left}_i(\mathcal{S}_j) \leq \text{right}_i(\mathcal{S}_k) \leq \text{right}_i(\mathcal{S}_j)$  for all  $i \in \{1, \dots, d\}$ , then we say that  $\mathcal{S}_j$  *overlaps* with  $\mathcal{S}_k$ , and the two seeds can be *merged*. SeqAn supports some methods for both, appending and merging seeds. In both cases, the combination of  $\mathcal{S}_k$  and  $\mathcal{S}_j$  is denoted by  $\mathcal{S}_k \circ \mathcal{S}_j$ .

Combining seeds is certainly useful only if the score of the resulting motif exceeds the scores of both individual seeds, so it is sufficient to consider only neighboring seeds when we are looking for seeds to combine, because seeds that are too widely separated would hardly achieve high scores. This is advantageous compared to global chaining that requires to find a predecessor for each

```

1  ▷ GAPPEDXDROP ( $a_1 \dots a_m, b_1 \dots b_n, right_0, right_1, X$ )
2   $k \leftarrow right_0 + right_1$ 
3   $best, score_k \leftarrow 0$ 
4   $L \leftarrow right_0$ 
5   $U \leftarrow right_0 + 1$ 
6  while  $k < n + m$  do
7       $k \leftarrow k + 1$ 
8       $L \leftarrow \max(L, k - n)$ 
9       $U \leftarrow \min(U, m)$ 
10     for  $i \leftarrow L$  to  $U$  do
11          $j \leftarrow k - i$ 
12          $M_{i,j} \leftarrow \max \begin{cases} M_{i-1,j-1} + \alpha(a_i, b_j) \\ M_{i-1,j} + g \\ M_{i,j-1} + g \end{cases}$ 
13         if  $M_{i,j} \leq best - X$  then
14              $M_{i,j} \leftarrow -\infty$ 
15         if  $M_{i,j} > best$  then
16              $right_0 \leftarrow i$ 
17              $right_1 \leftarrow k - i$ 
18          $score_k \leftarrow \max_i(M_{i,k-i})$ 
19         if  $score_k = score_{k-1} = -\infty$  then break
20          $best \leftarrow \max(best, score_k)$ 
21          $L \leftarrow \min\{i \mid M_{i,k-i} > -\infty \text{ or } M_{i-1,k-i} > -\infty\}$ 
22          $U \leftarrow \max\{i + 1 \mid M_{i,k-i} > -\infty \text{ or } M_{i,k-i-1} > -\infty\}$ 

```

} initialization  
 } compute  $M_{i,j}$  on antidiagonal  $k$   
 } extend the seed

Algorithm 23: **Gapped X-Drop Extension.**  $\alpha(a_i, b_j)$  is the score for aligning  $a_i$  and  $b_j$ ,  $g$  is the score for a blank. We assume  $M_{i,j} = -\infty$  for all  $i$  and  $j$  until  $M_{i,j}$  is set in line 11.

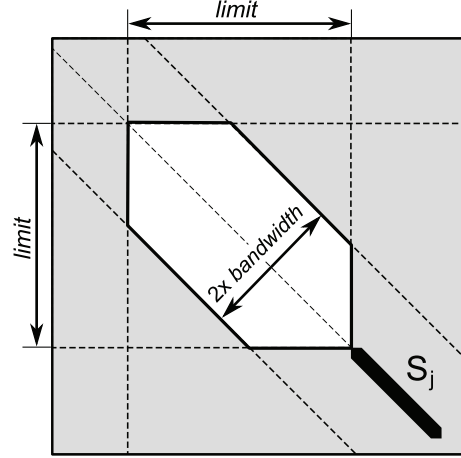


Figure 31: **Range of Possible Predecessors.** The ‘right end’ of a predecessor  $\mathcal{S}_k$  for  $\mathcal{S}_j$  must be within the white area, which is: (1) ‘left’ from  $\mathcal{S}_j$ , (2) between the two diagonals in distance *bandwidth* from the diagonal of the ‘left end’ of  $\mathcal{S}_j$ , and (3) in distance *limit* from  $left_1(\mathcal{S}_j)$ .

```

1  ▷ LOCALCHAINING( $\mathcal{S}_1, \dots, \mathcal{S}_n$ )
2  sort  $\mathcal{S}_1, \dots, \mathcal{S}_n$  in increasing order of  $right_1(\mathcal{S}_i)$ 
3   $D \leftarrow \{\mathcal{S}_1\}$ 
4  for  $j \leftarrow 2$  to  $n$  do
5       $\mathcal{S} \leftarrow \mathcal{S}_j$ 
6      for each  $\mathcal{S}_k \in D$  within the range of  $\mathcal{S}_j$  do
7          if  $right_1(\mathcal{S}_k) < right_1(\mathcal{S}_j) - limit$  then
8              report motif  $\mathcal{S}_k$ 
9               $D \leftarrow D \setminus \{\mathcal{S}_k\}$ 
10             else if  $weight(\mathcal{S}_k \circ \mathcal{S}_j) > weight(\mathcal{S})$  then
11                  $\mathcal{S} \leftarrow \mathcal{S}_k$ 
12             if  $\mathcal{S} = \mathcal{S}_j$  then
13                  $D \leftarrow D \cup \{\mathcal{S}_j\}$ 
14             else
15                  $D \leftarrow D \setminus \{\mathcal{S}\}$ 
16                  $D \leftarrow D \cup \{\mathcal{S} \circ \mathcal{S}_j\}$ 
17     report all motifs  $\in D$ 

```

} find best  
‘partner’  
 $\mathcal{S} \in D$  for  $\mathcal{S}_j$

Algorithm 24: **Greedy Local Chaining Heuristic.** The algorithm combines seeds as long as this benefits the score.  $\mathcal{S}_k \circ \mathcal{S}_j$  is the seed that we get by merging  $\mathcal{S}_j$  and  $\mathcal{S}_k$  or appending  $\mathcal{S}_j$  to  $\mathcal{S}_k$ . The constant *limit* determines the maximal distance between to seeds that may be combined.

seed, no matter of its distance. LOCALCHAINING (Algorithm 24) instead considers only seeds  $\mathcal{S}_k$  that are within a certain ‘range’ relative to a given seed  $\mathcal{S}_j$ . This range is defined by two constants *bandwidth* and *limit* as follows: (1) the diagonal  $rightdiag(\mathcal{S}_k) = right_2(\mathcal{S}_k) - right_1(\mathcal{S}_k)$  of  $\mathcal{S}_k$ ’s right border must be at most *bandwidth* away from the diagonal  $leftdiag(\mathcal{S}_j) = left_2(\mathcal{S}_j) - left_1(\mathcal{S}_j)$  of  $\mathcal{S}_j$ ’s left border, i.e.  $|leftdiag(\mathcal{S}_j) - rightdiag(\mathcal{S}_k)| \leq bandwidth$ , and (2) the distance between the right borders of  $\mathcal{S}_j$  and  $\mathcal{S}_k$  is below *limit*, i.e.  $|left_1(\mathcal{S}_j) - right_1(\mathcal{S}_k)| \leq limit$  and  $|left_2(\mathcal{S}_j) - right_2(\mathcal{S}_k)| \leq limit$ , see Figure 31. The class **SeedSet** in SeqAn implements a suitable data structure  $D$ . It stores all processed seeds in a map that allows fast searching for  $\mathcal{S}_k$  that meet condition (1). Seeds that violate condition (2) are removed from  $D$  in line 6.

The application of **SeedSet** is demonstrated in Section 15.2, see the listings on page 183. One call of function **addSeed** implements the inner loop of Algorithm 24. **addSeed** offers several modes for appending or merging seeds, see Table 18. If the desired mode for adding  $\mathcal{S}_j$  to the **SeedSet** is not possible because it contains no suitable ‘partner’ seed  $\mathcal{S}_k$ , then **addSeed** returns **false**.

<b>Single</b>	The seed is just added to the <b>SeedSet</b> .
<b>Merge</b>	The added seed is merged with a seed in the <b>SeedSet</b> if this benefits the score.
<b>SimpleChain</b>	The added seed is appended to a seed in the <b>SeedSet</b> if this benefits the score.
<b>Chaos</b>	The added seed is appended to a seed in the <b>SeedSet</b> if this benefits the score. Both seeds are expanded in a way that the resulting alignment contains at most one gap. The position of this gap is selected such that the score is maximized.
<b>Blat</b>	The added seed is appended to a seed in the <b>SeedSet</b> if this benefits the score. The gap between the two seeds is tried to be filled up with smaller matches.

Table 18: **Modes for Adding Seeds using addSeed.**

## 11.3 Multiple Sequence Motifs

So far, we discussed methods for pairwise motif finding. SeqAn also implements several algorithms to find motifs in  $d \geq 2$  sequences  $a^1, \dots, a^d$ , see Table 20. Since the complexity for searching approximate motifs grows heavily when we want to find them in more than two sequences, we simplify the problem as

follows:<sup>1</sup> (1) the length  $l$  of the wanted motif is given in advance, and (2) we allow a certain number of *mismatches* between a motif and its occurrences, but no inserts or deletes. An *occurrence* of  $m = m_1 \dots m_l$  in a string  $a^i$  is therefore a substring  $a_{pos+1}^i \dots a_{pos+l}^i$  that differs from  $m$  only in  $\leq k$  values, and the occurrence is *exact*, if they differ in *exactly*  $k$  values.

OOPS	‘one occurrence per sequence’: $m$ is an OOPS motif, if each sequence $a^1, \dots, a^k$ contains exactly one occurrence of $m$ .
OMOPS	‘one ore more occurrence per sequence’: $m$ is an OMOPS motif, if each sequence $a^1, \dots, a^k$ contains at least one occurrence of $m$ .
ZOOPS	‘zero ore one occurrence per sequence’: $m$ is a ZOOPS motif, if at least $\xi k$ out of $k$ sequences $a^1, \dots, a^k$ contain one occurrence of $m$ ( $0 < \xi \leq 1$ ).
TCM	‘two-component mixture’: $m$ is a TCM motif, if at least $\xi k$ out of $k$ sequences $a^1, \dots, a^k$ contain at least one occurrence of $m$ ( $0 < \xi \leq 1$ ).

occurrences	no	one	more
OOPS	0	$k$	0
OMOPS	0	$k$	
ZOOPS	$\leq (1 - \xi)k$	$\geq \xi k$	0
TCM	$\leq (1 - \xi)k$	$\geq \xi k$	

Table 19: **Motif Models.** The table shows the number of sequences  $a^1, \dots, a^k$  with no, one, or more occurrence of motif  $m$ .

SeqAn offers several alternatives to define the concept ‘motif’, which differ in the number of required occurrences in the sequences, see Table 19. We call this the *motif!model*, and together with the decision whether exact or non-exact occurrences are to be counted, the motif model defines the kind of search. The three models OOPS, ZOOPS, and TCM were introduced by Bailey and Elkan (1995), where the latter two depend on a parameter  $0 < \xi \leq 1$  that is the minimum fraction of the  $d$  sequences that must contain occurrences of  $m$ . Moreover, we implemented a new variant OMOPS (Lim 2007) that resembles TCM with  $\xi = 1$ .

Motif searching in SeqAn can be accessed via `findMotif`, which get three arguments: (1) an instance of the class `MotifFinder` that specifies the applied algorithm (Table 20) stores all needed temporary data for processing the search together with all necessary constants, like the number of allowed errors  $k$ , (2) the sequences  $a^1, \dots, a^d$ , and (3) a tag that specifies the motif model (Table 19). See Listing 25 for an example.

<sup>1</sup>Nevertheless, the problem stays NP-hard (Lanctot et al. 1999).

SeqAn supports two kinds of motif finding algorithms: randomized heuristics (**Projection**, **EPatternBranching**) and exhaustive enumeration algorithms (**PMS1**, **PMSP**). As an example for the former, we will sketch in the following **Projection** as an example, and **PMSP** for the latter.

<b>Projection</b>	A heuristic by Buhler and Tompa (2001) that use local sensitive hashing to get promising input estimates for the EM-algorithm, see Section 11.3.1.
<b>EPatternBranching</b>	This heuristic by Davila, Balla, and Rajasekaran (2006) is an extension of the ‘Pattern Branching’ algorithm by Price et al. (2003). It applies local search techniques to optimize motif candidates. The current implementation supports only the motif models <b>OOPS</b> and <b>OMOPS</b> .
<b>PMS1</b>	An enumerating algorithm by Rajasekaran et al. (2005).
<b>PMSP</b>	A space-saving variant of <b>PMS1</b> (Davila et al. 2006), see Section 11.3.2.

Table 20: **Motif Finding Algorithms.** Specializations of **MotifFinder**.

### 11.3.1 The Randomized Heuristic Projection

Suppose that an unknown  $l$ -mer  $m$  was ‘planted’ according to the current motif model with  $k$  or up to  $k$  errors at random positions into  $d$  random sequences  $a^1, \dots, a^d$ , then finding  $m$  can be formulated as *maximum likelihood estimation*: ‘Find an estimate for  $m$  for which the chance of observing  $a^1, \dots, a^d$  is maximal’. Some tools like MEME (Bailey and Elkan 1994) and **Projection** (Buhler and Tompa 2001) apply an ‘expectation-maximization’ algorithm to identify estimates for  $m$ . The *EM-algorithm* (Dempster et al. 1977) has the advantage that it can handle incomplete data, e.g. in our case, that we do not know the actual positions at which the motif was planted into the sequences. Let the ‘model parameters’  $\theta_{h,c}$  be estimates of  $p(m_h = c)$ , and define the ‘unknown variables’  $Z_{i,j}$  as:

$$Z_{i,j} = \begin{cases} 1, & \text{if } m \text{ was planted into } a^i \text{ at position } j \\ 0, & \text{otherwise} \end{cases}$$

The algorithm optimizes  $\theta_{h,c}$  by repeating two steps:

- (1) E-step: We compute the expected values for  $Z_{i,j}$  given the current estimates for  $\theta_{h,c}$ , i.e. we get according to Bayes’ theorem:

$$E(Z_{i,j}|a^i, \theta) \leftarrow \frac{p(a^i|Z_{i,j} = 1, \theta)}{\sum_k p(a^i|Z_{i,k} = 1, \theta)}$$

```

unsigned int const l = 4; //length of motif
unsigned int const k = 1; //number of substitutions

String<DnaString> dataset;
appendValue(dataset,DnaString("ACAGCA"));
appendValue(dataset,DnaString("AGGCAG"));
appendValue(dataset,DnaString("TCAGTC"));

MotifFinder<Dna, PMSP> finder(l, k, true);
findMotif(finder, dataset, OMOPS());
cout << getMotif(finder) << endl;

```

Listing 25: **Motif Finding Example.** The motif model is OMOPS (specified by the third argument of `findMotif`) with exact occurrences (specified by `true` in the third argument of the finder's constructor).

- (2) M-step: We use the values  $Z_{i,j}$  from the last E-step to re-estimate  $p(m_h = c)$  such that the likelihood of getting  $a^1, \dots, a^d$  is maximized.

Dempster, Laird, and Rubin (1977) showed that the model variables  $\theta$  of the EM-algorithm converge to a local maximum for the likelihood of the observed data. Which local maximum is reached depends on the estimate  $\theta$  the algorithm starts.

```

1  ▷ PROJECTION( $a^1, \dots, a^d, l, R, limit$ )
2  repeat for up to  $R$  loop cycles
3       $f \leftarrow$  random locality-sensitive hash function
4       $bucket[x] \leftarrow \emptyset$  for each hash value  $x$ 
5      for each  $l$ -mer  $w$  in  $a^1, \dots, a^d$  do
6           $bucket[f(w)] \leftarrow bucket[f(w)] \cup \{w\}$ 
7      for each  $x$  with  $|bucket[x]| \geq limit$  do
8          generate  $\theta$  from  $bucket[x]$ 
9           $m \leftarrow$  EM-ALGORITHM( $\theta$ )
10         if  $m$  complies motif model then
11             report  $m$ 
12         return

```

Algorithm 25: **Projection Algorithm for Motif Finding.**

PROJECTION (Algorithm 25) applies *locality-sensitive hashing* (Indyk and Motwani 1998) to determine promising inputs for the EM-algorithm. A hashing function  $f$  is called ‘locality-sensitive’, if the probability for collisions (i.e.  $f(a) = f(b)$ ) between two hashed objects  $a \neq b$  is higher for similar objects



than for dissimilar objects. Buhler and Tompa used gapped  $q$ -gram shapes (see Section 12.1.1) as hash functions. The similarity between different occurrences of the motif  $m$  is above the expected similarity between random  $l$ -grams, hence if we apply locality-sensitive hashing to all  $l$ -grams in  $a^1, \dots, a^d$ , then occurrences of  $m$  have an above-average probability to collide with other  $l$ -grams, namely with other occurrences of  $m$ . In reverse, there is a good chance for  $l$ -mers with many collisions to be occurrences of  $m$ , thus those  $l$ -mers are good input candidates  $\theta$  for the EM-Algorithm.

```

▷ PMSP( $a^1, \dots, a^d, l, k$ )
1  for  $j \leftarrow 0$  to  $\text{length}(a^1) - l$  do
2      for  $s \leftarrow 2$  to  $d$  do
3           $N^s \leftarrow \text{GET2KNEIGHBORS}(a^1, a^s, l, k, j)$ 
4          for each  $l$ -gram  $m$  with  $\delta(m, a_{j+1}^1 \dots a_{j+l}^1) = k$ 
5              do
6                  if for each  $s \in \{2, \dots, d\}$  exists  $w^s \in N^s$  with
                       $\delta(m, w^s) = k$  then
                          report motif  $m$ 

▷ GET2KNEIGHBORS( $a, b, l, k, j$ )
1   $N \leftarrow \emptyset$ 
2  for  $i \leftarrow \text{length}(b) - l$  down to 0 do
3       $D_i \leftarrow \begin{cases} \delta(a_1 \dots a_l, b_{i+1} \dots b_{i+l}) & \text{if } j = 0 \\ D_{i-1} - \delta(a_j, b_i) + \delta(a_{j+l}, b_{i+l}) & \text{otherwise} \end{cases}$ 
4      if  $D_i \leq 2k$  then
5           $N \leftarrow N \cup \{b_{i+1} \dots b_{i+l}\}$ 
6  return  $N$ 

```

Algorithm 26: **PMSP Algorithm for Motif Finding.** Motif model is OMOPS, only exact occurrences are counted.  $\delta(x, y)$  is the Hamming distance between two strings  $x$  and  $y$ .

### 11.3.2 The Enumerating Algorithm PMSP

PMSP is an exhaustive motif search algorithm by Davila et al. (2006). SeqAn implements PMSP for all motif models; Algorithm 26 shows it for OMOPS and exact occurrences. OMOPS means, that a motif occurs somewhere in  $a^1$  with  $k$  errors, so PMSP enumerates all  $l$ -grams  $m$  with *Hamming distance*  $k$  to a substring  $a_{j+1}^1 \dots a_{j+l}^1$  of  $a^1$ . If one of these  $m$  also occurs in all other sequences  $a^s$  for  $s \in \{2, \dots, d\}$ , then  $m$  is a motif. The distance between  $m$  and an occurrence  $w^s$  in  $a^s$  is  $k$ , thus the distance between  $w^s$  and  $a_{j+1}^1 \dots a_{j+l}^1$  is  $\leq 2k$ .

To determine all  $l$ -grams in  $a^s$  that fulfills this condition, GET2KNEIGHBORS computes their distances to  $a_{j+1}^1 \dots a_{j+l}^1$  (line 3). For  $j > 0$ , these distances are computed incrementally from the distances for  $j - 1$ .

# Chapter 12

## Indices

*Indices* are data structures that store processed data about a sequence or a set of sequences to facilitate searching in them. For example, indices allow fast exact pattern matching or exact motif finding. Listing 26 shows for example how to search an indexed text in SeqAn. SeqAn implements several index data structures. Table 21 lists the available specializations of the class `Index`.

<code>Index_QGram</code>	A simple hashing table of all (gapped) <i>q</i> -grams of a string or string set, see Section 12.1.
<code>Index_ESA</code>	A <i>suffix array</i> (Manber and Myers 1990), that can be extended by a set of additional tables to an <i>enhanced suffix array</i> (Abouelhoda, Kurtz, and Ohlebusch 2002), see Section 12.2. The index implements iterators that allow using the data structure like a <i>suffix tree</i> (Weiner 1973), see Section 12.3.2.
<code>Index_Wotd</code>	A lazy suffix tree (Giegerich, Kurtz, and Stoye 1999). The index is <i>deferred</i> , which means that it is built up during the use.
<code>PizzaChili</code>	A wrapper for the index structures from the Pizza & Chili Corpus (Ferragina and Navarro ), e.g. for compressed text indexes.

Table 21: **Index Data Structures. Specializations of Index.**

Many indices consist of several parts, we say it is a ‘*bunch*’ of ‘*fibers*’. An enhanced suffix array (`Index_ESA`) for example has at least the fiber `ESA_Text` that is the indexed text and the fiber `ESA_SA` that contains the suffix table. More fibers like the ‘longest common prefix’-table (fiber `ESA_LCP`) can be created when needed. The types of fibers can be determined by the metafunction `Fiber`, and the function `getFibre` is used for accessing the fibers of an index. String indices in SeqAn are in general capable of working on multiple sequences  $a^1, \dots, a^d$  at once. This could be done by building up the index for the concatenated string  $a^1 \dots a^d$ , e.g. by using the *concatenator* of a `StringSet`, see Section 8.8.

```

Index< String<char>, Index_ESA > index_esa("tobeornottobe");
Finder< Index< String<char> > > finder_esa(index_esa);
while (find(finder_esa, "be"))
{
    cout << position(finder_esa) << endl;
}

```

Listing 26: **Exact Searching in an Index.** The program prints out the positions of all occurrences of "be" in "tobeornottobe", i.e. 2 and 11.

## 12.1 $q$ -Gram Indices

### 12.1.1 Shapes

A  $q$ -gram (in the narrow sense) is a string of length  $q$ , and ‘the’  $q$ -grams of a text  $a = a_1 \dots a_n$  are the  $n - q + 1$  length- $q$  substrings of this text. We also call this kind of  $q$ -gram ‘*ungapped*’ since it consists of  $q$  *successive* values  $a_{i+1} \dots a_{i+q}$ . A *gapped*  $q$ -gram on the other hand is a subsequence  $a_{i+s_1} a_{i+s_2} \dots a_{i+s_q}$  of  $a$ , where  $s = \langle s_1, \dots, s_q \rangle$  is an ordered set of positions  $s_1 = 1 < s_2 < \dots < s_q$ . We call  $s$  a *shape*, and we define  $weight(s) = q$  and  $span(s) = s_q$ . Ungapped  $q$ -grams are therefore a special case of gapped  $q$ -grams with the shape  $s = \langle 1, \dots, q \rangle$ .

<b>SimpleShape</b>	An ungapped shape. The length $q$ can be set at run time by calling the function <code>resize</code> .
<b>UngappedShape</b>	An ungapped shape of fixed length, i.e. the length is specified at compile time as a template argument constant.
<b>GappedShape</b>	A generic gapped shape that can be changed at run time. It is defined for example by calling the function <code>stringToShape</code> that translates a string of characters ‘1’ (relevant position) and ‘0’ (irrelevant gap position) into a shape, i.e. the string "11100101" would be translated into the shape $s = \langle 1, 2, 3, 6, 8 \rangle$ .
<b>HardwiredShape</b>	This subspecialization of <code>GappedShape</code> stores a gapped shape that is defined at compile time. The shape $\langle s_1, s_2, \dots, s_q \rangle$ is encoded in a list of the $q - 1$ differences $s_2 - s_1, s_3 - s_2, \dots, s_q - s_{q-1}$ , which are specified as template argument constants of the tag class <code>HardwiredShape</code> . For example, the shape $s = \langle 1, 2, 3, 6, 8 \rangle$ would be encoded as <code>HardwiredShape&lt;1, 1, 3, 2&gt;</code> .

Table 22: **Shape Specializations.**

SeqAn offers several alternative data structures for storing gapped or ungapped shapes, see Table 22 and Figure 8 on page 51. The main purpose of these shape classes is to compute *hash values*: Given a shape  $s = \langle s_1, \dots, s_q \rangle$ , then

we define the hash value of a  $q$ -gram  $a_{s_1}, \dots, a_{s_q}$  to be <sup>1</sup>

$$\text{hash}(a_{s_1} \dots a_{s_q}) = \sum_{i=1}^q \text{ord}(a_{s_i}) |\Sigma|^{q-i}$$

In other words,  $\text{hash}$  regards  $\text{ord}(a_{s_1}) \dots \text{ord}(a_{s_q})$  as a number of base  $|\Sigma|$  with  $q$  digits. Obviously  $\text{hash}(a^1) \neq \text{hash}(a^2)$  for two different  $q$ -grams  $a^1 \neq a^2$ .

The specializations of **Shape** differ in the performance of computing hash values, see Figure 9 on page 51. For example, the ‘ungapped’ specializations are faster than their ‘gapped’ counterparts if we need to compute the hash values of all  $q$ -grams in a text, since the hash value of the  $i$ -th ungapped  $q$ -gram can be incrementally computed in constant time from the hash value of the  $i-1$ -th ungapped  $q$ -gram by:

$$\text{hash}(a_{i+1} \dots a_{i+q}) = \text{hash}(a_i \dots a_{i+q-1})q - a_i |\Sigma|^q + a_{i+q}$$

Moreover, ‘fixed’ variants are faster than their ‘variable’ counterparts, because the the compiler can optimize the code better if the shape is known at compile time.

### 12.1.2 $q$ -Gram Index Construction

Let  $s = \langle s_1, \dots, s_q \rangle$  be a shape. A  $q$ -gram index allows to look up in constant time all occurrences of a given  $q$ -gram  $b$  in a text  $a = a_1 \dots a_n$ . The index consists of two tables (see Figure 32): (1) The *position table*  $P$  that enumerates the starting positions  $j \in \{0, \dots, n - s_q\}$  of all  $q$ -grams in the text ordered by  $\text{hash}_s(a_{j+s_1} \dots a_{j+s_q})$ , and (2) the *directory table*  $D$  that stores for each value  $k \in \{0, \dots, |\Sigma|^q\}$  the number of  $q$ -grams  $b$  in the text with  $\text{hash}_s(b) < k$ . With these two tables, it is easy to look up the occurrences of  $b$  in  $a$  at the positions  $P[D[\text{hash}_s(b)]], \dots, P[D[\text{hash}_s(b) + 1] - 1]$ .

Both tables can be built up together in time  $O(n)$  using *count sort* (Algorithm 27). COUNTSORT sorts the positions  $p_1, \dots, p_m$  by the keys  $k_1, \dots, k_m$  in three steps: (1) For each  $k \in \{0, \dots, Z - 1\}$  it counts in  $D$  the number of  $k_j = k$ , (2) the counts in  $D$  are summed up such that each  $D[k]$  contains the number of  $k_j < k - 1$ , and (3) the  $p_j$  are sorted into  $P$  guided by  $D$ . The sorting is *stable*: If  $k_i = k_j$  for  $i < j$ , then  $p_i$  is sorted before  $p_j$  into  $D$ . BUILDQGRAMINDEX calls COUNTSORT using the hash values of all  $q$ -grams as keys. This algorithm can be implemented ‘lightweight’, i.e. only the space of  $D$  and  $P$  is needed, by computing the hash values ‘on the fly’ when they are needed in the steps (1) and (3).

SeqAn also supports building up  $P$  without  $D$ , which is especially useful if  $|\Sigma|^q$  gets too large. The function `createQGramIndexSAOnly` applies the `sort` algorithm from the standard C++ library.

---

<sup>1</sup>Remember that *ord* returns for each value  $\in \Sigma$  a unique integer  $\in \{0, \dots, |\Sigma| - 1\}$ , see Section 7.4.

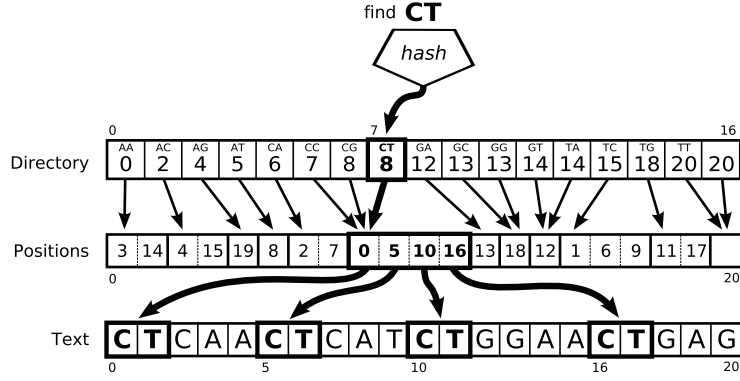
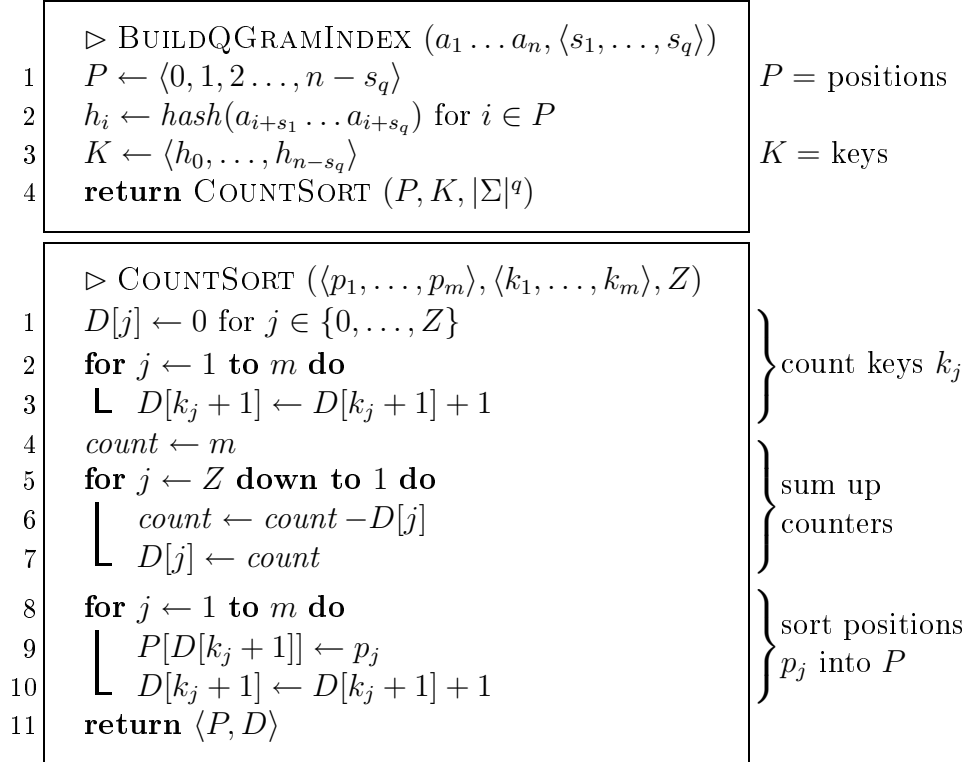


Figure 32:  **$q$ -gram Index.** In this example, the 2-gram index of "CTCAACTCATCTGGAAG" is searched for all occurrences of "CT". We first compute  $hash("CT") = 7$ , and then we find in the 'Positions' table at  $P[D[7]], \dots, P[D[8] - 1]$  the positions 0, 5, 10, and 16. ( $D$  is the 'Directory' table).



Algorithm 27: **Count Sort Algorithm for  $q$ -Gram Index Construction.** The shape  $\{s_1, \dots, s_q\}$  is used for indexing a text  $a_1 \dots a_n$  of alphabet  $\Sigma$ . The algorithm returns both the position table  $P$  and the directory table  $D$ . It is assumed that  $n > s_q$  and  $|\Sigma|^q \geq 2$ .



```
String<char> text = "hello world!";
String<unsigned int> sa;
resize(sa, length(text));
createSuffixArray(sa, text, Skew7());
```

Listing 27: **Using createSuffixArray to Build Up a Suffix Array.** In this example, the applied algorithm is *Skew7*.

SeqAn as default method for suffix array construction, since it is fast, generic, and it is also excellent for building up suffix arrays on external memory (Dementiev et al. 2008). SKEW bases on the idea of *merge sort*: The set of suffixes is divided in two parts  $S^{12}$  and  $S^0$ , each part is sorted separately (by SORTS12 and SORTS0), and then they are merged together by MERGE.

We define for  $z \in \{0, 1, 2\}$  the sets  $S^z = \{j \in S \mid j = z + 3i \text{ for integer } i\}$ , and  $S^{12} = S^1 \cup S^2$ . SORTS12 bases on the following observation: Instead of sorting every third suffix of  $a$ , we can also sort every suffix of a string  $t$ , where each character of  $t$  consists of three successive characters of  $a$ . SORTS12 constructs values  $k^j$  for  $j \in S^{12}$  that conserve the order of first three characters of the suffixes  $b^j$ . Since we want also consider suffixes with length  $< 3$ , we define in line 2 of Algorithm 28 the values  $a_{n+1} = a_{n+2} = a_{n+3} = \$$ , where  $\$$  is a character not used in  $a$  for which holds  $\text{ord}(\$) \leq \text{ord}(c)$  for all  $c \in \Sigma$ .<sup>2</sup> If all these  $k^j$  are different, then sorting the  $k^j$  already sorts the  $b^j$ . Otherwise (lines 6 to 15), SORTS12 applies a recursive call of SKEW to sort the suffixes of a string  $t = t^1 t^2$ , where  $t^1$  and  $t^2$  correspond to the suffixes with positions in  $S^1$  and  $S^2$ , respectively. For minimizing the alphabet size of  $t$ , SORTS12 previously computes values  $T(k^j)$  by enumerating the  $k^j$ , such that the  $T(k^j)$  conserve the order of the  $k^j$  (lines 6 to 9). Hence the alphabet size of  $t$  is bounded by the length of the input sequence  $n$ . Note that appending  $t^2$  to  $t^1$  does not affect the mutual order of the suffixes in  $t^1$ , since the last character in  $t^1$  is unique in  $t$  by construction. After that (in lines 13 to 15), the values in  $S^{12}$  are translated from positions in  $t$  into positions in  $a$ .

SORT0 (Algorithm 29) sort the suffixes of  $a$  at positions in  $S^0$ . Obviously  $b^{j_1} \leq_{\text{lex}} b^{j_2}$  for two positions  $j_1, j_2 \in S^0$ , if either  $a_{j_1+1} < a_{j_2+1}$ , or  $a_{j_1+1} = a_{j_2+1}$  and  $b^{j_1+1} \leq_{\text{lex}} b^{j_2+1}$ . The last condition was already checked in SORTS12, since  $j_1 + 1, j_2 + 1 \in S^1 \subseteq S^{12}$ . Therefore SORT0 first sorts the positions  $j \in S^0$  according to the occurrences of  $j + 1$  in  $S^{12}$  (lines 1 to 5), and then it uses ‘*stable sorting*’ to sort them again by  $a_{j+1}$  (line 7).

MERGE scans both  $S^0$  and  $S^{12}$ , and in each step it appends either  $x = S^0[i]$  or  $y = S^{12}[j]$  to  $S$ , depending on the lexicographical order between  $b^x$  and  $b^y$ . If  $y \in S^1$ , then both  $x + 1 \in S^{12}$  and  $y + 1 \in S^{12}$ . Hence  $b^{x+1} <_{\text{lex}} b^{y+1}$  if  $x + 1$

---

<sup>2</sup>For the implementation of SKEW in SeqAn, we modified the algorithm such that it does not require a special character  $\$$  (Weese 2006).



<b>Skew3</b>	A linear time algorithm by Kärkkäinen and Sanders (2003), which applies a merge sort approach, where two thirds of the suffixes are recursively sorted, see <b>Skew</b> , Algorithm 28, 29.
<b>Skew7</b>	A variant of <b>Skew3</b> that recursively sorts three seventh (instead of two thirds) of the suffixes (Weese 2006). This reduces the number of recursive calls, so <b>Skew7</b> is slightly faster than <b>Skew3</b> .
<b>ManberMyers</b>	The algorithm by Manber and Myers (1990) that bases on ‘prefix doubling’. It is quite slow in practice, though the run time is $O(n \log n)$ .
<b>LarssonSadakane</b>	The algorithm by Larsson and Sadakane (2007).
<b>SAQSort</b>	If this tag is specified, the suffixes are sorted using the function <code>sort</code> of the standard C++ library. This is not recommended when a repetitive text is indexed.

Table 24: **Suffix Array Construction Algorithms.**

comes before  $y + 1$  in  $S^{12}$ , that is if  $I(x + 1) < I(y + 1)$ , where  $I$  is the *inverse suffix array* of  $S^{12}$ . Therefore  $b^x <_{lex} b^y$  if  $a_{x+1} <_{lex} a_{y+1}$  or  $a_{x+1} = a_{y+1}$  and  $I(x + 1) < I(y + 1)$ . If on the other hand  $y \in S^2$ , then  $x + 2, y + 2 \in S^{12}$ , hence  $b^x <_{lex} b^y$  if  $a_{x+1}a_{x+2} <_{lex} a_{y+1}a_{y+2}$  or  $a_{x+1}a_{x+2} = a_{y+1}a_{y+2}$  and  $I(x + 2) < I(y + 2)$ .

### 12.2.2 Searching in Suffix Arrays

Let  $S$  be the suffix array of a text  $t = t_1 \dots t_n$ ,  $p = p_1 \dots p_m$  a pattern, and  $s^j = t_{S[j]+1} \dots t_n$  the suffix in  $S$  at  $j \in \{0, \dots, n - 1\}$ . We define

$$L = \max\{j \mid s^i <_{lex} p \text{ for all } i < j\}$$

$$R = \max\{j \mid s^i \leq_{lex} p \text{ for all } i < j\}$$

Obviously  $L \leq R$ , and if  $L < R$ , then  $p$  occurs in  $t$  at positions  $S[L], S[L + 1], \dots, S[R - 1]$ , and only there. **SEARCHSA** (Algorithm 30) shows how  $L$  can be found using *binary searches*. In each pass of the main loop, the interval  $[left, \dots, right]$  is cut into halves at the position  $mid$ . If  $p \leq_{lex} s^{mid}$ , then  $L \leq mid$  and therefore  $L \in [left, \dots, mid]$ , otherwise  $L \in [mid + 1, \dots, right]$ . The algorithm stops if the interval only contains  $L = left = right$ .

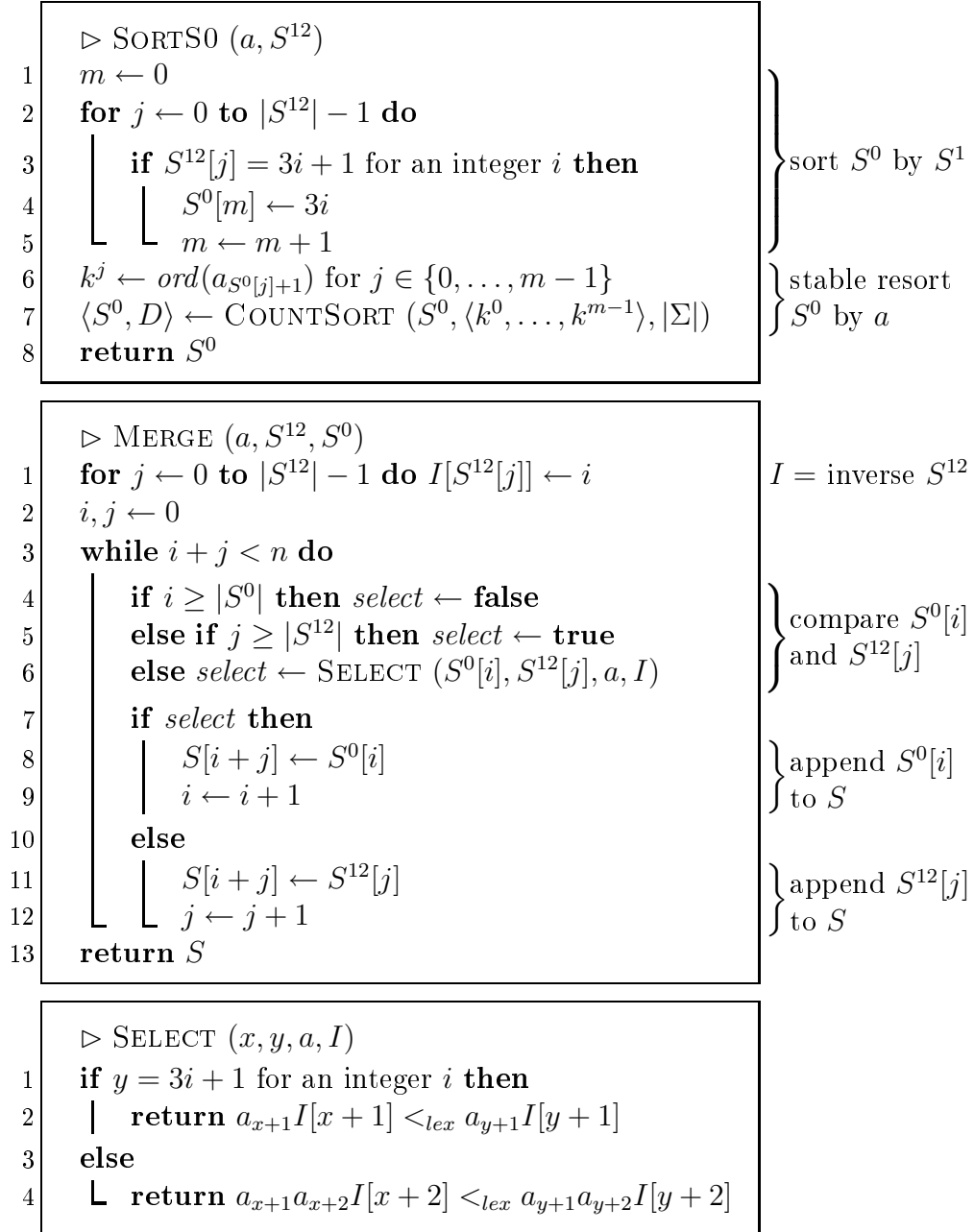
For computing  $R$ , we need to change the condition at line 11 to

$$i \leq n \text{ and } (j > m \text{ or } p_j <_{lex} t_i),$$

which is equivalent to  $p <_{lex} s^{mid}$ . If this condition is fulfilled, then  $R \in [left, \dots, mid]$ , otherwise  $R \in [mid + 1, \dots, right]$ .

<pre> 1  ▷ SKEW (<math>a = a_1 \dots a_n</math>) 2  <b>if</b> <math>n = 1</math> <b>then return</b> <math>\langle 0 \rangle</math> 3  <b>else</b> 4      <math>S^{12} \leftarrow \text{SORTS12}(a)</math> 5      <math>S^0 \leftarrow \text{SORTS0}(a, S^{12})</math> 6      <b>return</b> <math>\text{MERGE}(a, S^{12}, S^0)</math> </pre>	<pre> ▷ SORTS12 (<math>a</math>) 1  <math>S^{12} \leftarrow \langle 1, 2, 4, 5, \dots, 3i+1, 3i+2, \dots \rangle</math> positions <math>\leq n</math> 2  <math>k^j \leftarrow \text{hash}(a_{j+1}a_{j+2}a_{j+3})</math> for <math>j \in S^{12}</math> 3  <math>K \leftarrow \langle k^1, k^2, k^4, k^5, \dots, k^{3i+1}, k^{3i+2}, \dots \rangle</math> 4  <math>\langle S^{12}, D \rangle \leftarrow \text{COUNTSORT}(S^{12}, K,  \Sigma ^3)</math> 5  <b>if</b> <math>k^i = k^j</math> for any two <math>i \neq j</math> <b>then</b> 6      <math>k \leftarrow 0</math> 7      <b>for</b> <math>j \leftarrow 0</math> <b>to</b> <math> \Sigma ^3 - 1</math> <b>do</b> 8          <math>T[j] \leftarrow k</math> 9          <b>if</b> <math>D[j] \neq D[j+1]</math> <b>then</b> <math>k \leftarrow k+1</math> 10     <math>t^1 \leftarrow T[k^1] T[k^4] T[k^7] \dots T[t^{3i+1}] \dots</math> 11     <math>t^2 \leftarrow T[k^2] T[k^5] T[k^8] \dots T[t^{3i+2}] \dots</math> 12     <math>S^{12} \leftarrow \text{SKEW}(t^1 t^2)</math> 13     <b>for</b> <math>j \leftarrow 0</math> <b>to</b> <math> S^{12}  - 1</math> <b>do</b> 14         <b>if</b> <math>S^{12}[j] &lt;  t^1 </math> <b>then</b> <math>S^{12}[j] \leftarrow 3S^{12}[j] + 1</math> 15         <b>else</b> <math>S^{12}[j] \leftarrow 3(S^{12}[j] -  t^1 ) + 2</math> 16     remove from <math>S^{12}</math> all entries <math>\geq n</math> 17 <b>return</b> <math>S^{12}</math> </pre>	<div style="display: flex; align-items: center;"> <span style="font-size: 3em; margin-right: 10px;">}</span> <div> <div style="margin-bottom: 10px;">sort 3-grams</div> <div style="margin-bottom: 10px;">reduce alphabet size</div> <div style="margin-bottom: 10px;">recursion</div> <div>transform positions</div> </div> </div>
---	--	---

Algorithm 28: **Skew Algorithm for Suffix Array Construction (part one)**. In line 2, we define  $a_{n+1} = a_{n+2} = a_{n+3} = \$$ . SORTS0 and MERGE are defined in Algorithm 29, COUNTSORT in Algorithm 27.



Algorithm 29: **Skew Algorithm for Suffix Array Construction (part two).**  
COUNTSORT is defined in Algorithm 27.

SEARCHSA computes two values  $l$  and  $r$  for which the following invariants hold: (1) There is an  $x \leq left$  such that  $p$  and  $s^x$  share the first  $l - 1$  values, and (2)  $p$  and  $s^{right}$  share the first  $r - 1$  values. Therefore each suffix  $s^{mid}$  with  $x \leq mid \leq right$  share at least the first  $\min(l, r) - 1$  with  $p$ , so these values need no further examination when want to compare  $p$  and  $s^{mid}$  (see lines 6 to 10). This speeds up the search, although the worst case runtime is still  $\Omega(m \log n)$ .

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17	<pre> ▷ SEARCHSA (<math>p = p_1 \dots p_m, t = t_1 \dots t_n, S</math>) <math>left \leftarrow 0</math> <math>right \leftarrow n</math> <math>l, r \leftarrow 1</math> <b>while</b> <math>left &lt; right</math> <b>do</b>     <math>mid \leftarrow \lfloor \frac{left + right}{2} \rfloor</math>     <math>j \leftarrow \min(l, r)</math>     <math>i \leftarrow S[mid] + j</math>     <b>while</b> <math>j \leq m</math> and <math>i \leq n</math> and <math>p_j = t_i</math> <b>do</b>         <math>j \leftarrow j + 1</math>         <math>i \leftarrow i + 1</math>     <b>if</b> <math>(j &gt; m)</math> or <math>(i \leq n</math> and <math>p_j &lt;_{lex} t_i)</math> <b>then</b>         <math>right \leftarrow mid</math>         <math>r \leftarrow j</math>     <b>else</b>         <math>left \leftarrow mid + 1</math>         <math>l \leftarrow j</math> <b>return</b> <math>left</math> </pre>	<div style="display: flex; align-items: center;"> <div style="font-size: 3em; margin-right: 5px;">}</div> <div> compare  <math>p</math> and <math>s^{mid}</math> </div> </div> <div style="display: flex; align-items: center; margin-top: 10px;"> <div style="font-size: 3em; margin-right: 5px;">}</div> <div> <math>p \leq_{lex} s^{mid}</math>;  go left </div> </div> <div style="display: flex; align-items: center; margin-top: 10px;"> <div style="font-size: 3em; margin-right: 5px;">}</div> <div> <math>p &gt;_{lex} s^{mid}</math>;  go right </div> </div>
---	---	---

Algorithm 30: **Searching a Suffix Array.**  $S$  is the suffix array of the text  $t$ , and  $p$  the pattern that is searched in  $t$ . The algorithm returns  $L$ ; it would compute  $R$  if the condition in line 11 is changed to ' $i \leq n$  and  $(j > m$  or  $p_j <_{lex} t_i)$ '. The pattern  $p$  occurs in  $t$  at positions  $S[L], S[L + 1], \dots, S[R - 1]$ .

## 12.3 Enhanced Suffix Arrays

The suffix array can be extended to the very powerful data structure '*enhanced suffix array*' by adding some further tables, see Table 25. We will discuss the LCP table (ESA\_LCP) in Section 12.3.1. Together with the suffix table, the LCP table allows a depth-first search in the *suffix tree* of the text, and we will describe applications for it in Section 12.3.2.

ESA_Text	The indexed text.
ESA_SA	The <i>suffix array</i> that contains the positions of the lexicographical ordered suffixes of the indexed text ESA_Text, see Section 12.2.
ESA_LCP	A table that contains the lengths of the <i>longest common substrings</i> between adjacent suffixes in the suffix array ESA_SA, see Section 12.3.1.
ESA_ChildTab	A table that contains all structural information about the <i>suffix tree</i> that is missing in ESA_SA and ESA_LCP (Abouelhoda et al. 2002).
ESA_BWT	The Burrows-Wheeler transformation of the indexed text ESA_Text (Burrows and Wheeler 1994). It contains the preceding character $a_{j-1}$ to each suffix $a_j \dots a_n$ in ESA_SA.

Table 25: Enhanced Suffix Array Fibers.

### 12.3.1 LCP Table

Let  $S$  be the suffix array of the text  $a = a_1 \dots a_n$  and  $b^j = a_{j+1} \dots a_n$  the  $j$ -th suffix of  $a$ . The *LCP table*  $L$  stores in  $L[k]$  the length of the *longest common prefix* between the suffix  $b^{S[k]}$  and its predecessor  $b^{S[k-1]}$  in  $S$ , i.e.  $L[k] = lcp(b^{S[k-1]}, b^{S[k]})$  for  $k \in \{1, \dots, n-1\}$ , where

$$lcp(x_1 \dots x_k, y_1 \dots y_l) = \max\{i \mid x_1 \dots x_i = y_1 \dots y_i\}.$$

The LCP table can be constructed in linear time (Kasai et al. 2001) due to the following observation: If  $b^i <_{lex} b^j$  and  $lcp(b^i, b^j) = h > 0$ , then  $b^{i+1} <_{lex} b^{j+1}$  and  $lcp(b^{i+1}, b^{j+1}) = h - 1$ . Any common prefix between  $b^{j+1}$  and  $b^{i+1}$  is also a prefix of the predecessor of  $b^{j+1}$  in  $S$ , thus the entry in  $L$  for  $b^{j+1}$  is  $\geq h - 1$ . BUILDLCPTAB (Algorithm 31) enumerates the suffixes  $b^0, \dots, b^{n-1}$  and computes for each  $b^j$  its entry in  $L$ . The *inverse suffix array*  $I$  of  $S$  is used to determine the predecessor  $b^i = b^{I[j]-1}$  of  $b^j$  in  $S$ . Suppose that the entry in  $L$  for  $b^i$  is  $h$ , then the entry for  $b^j$  in  $L$  is at least  $h - 1$ , so we can save  $h - 1$  comparisons in line 6. Since  $h \leq n - 1$ , the inner loop is executed at most  $2n$  times, and the runtime of BUILDLCPTAB is therefore  $O(n)$ .

SeqAn implements an ‘in-place’ variant of this algorithm that does not need extra space for storing the inverse suffix array  $I$  (Weese 2006).

### 12.3.2 Suffix Trees Iterators

A *suffix tree* (Weiner 1973)  $\mathcal{T}$  of a text  $a = a_1 \dots a_n$  is the unique rooted tree with the minimum number of vertices that has the following characteristics (see Figure 34): Let  $r = v_1, v_2, \dots, v_k = v$  be the path in  $\mathcal{T}$  from the root vertex  $r$  to a vertex  $v$ , then all edges from  $v_{i-1}$  to  $v_i$  are labeled with non-empty strings  $s^{i-1,i}$ , and the concatenated *path label*  $s^{1,2}s^{2,3} \dots s^{k-1,k}$  is a substring of  $a_1 \dots a_n \$$ , where  $\$$  is a special ‘end-of-string’ character that does not occur

```

    ▷ BUILDLCPTAB ( $a_1 \dots a_n, S$ )
1  for  $j \leftarrow 0$  to  $n - 1$  do  $I[S[j]] \leftarrow j$ 
2   $h \leftarrow 0$ 
3  for  $j \leftarrow 0$  to  $n - 1$  do
4      if  $I[j] \neq 0$  then
5           $i \leftarrow S[I[j] - 1]$ 
6          while  $i + h < n$  and  $j + h < n$  and
               $a_{i+h+1} = a_{j+h+1}$  do
7               $h \leftarrow h + 1$ 
8               $L[I[j]] \leftarrow h$ 
9          if  $h > 0$  then  $h \leftarrow h - 1$ 
10 return  $L$ 

```

$I = \text{inverse } S$

} compare  $s^{I[j]}$  and  $s^{I[j]-1}$

Algorithm 31: **Construction of the LCP Table.**  $S$  is the suffix array of  $a_1 \dots a_n$ .

anywhere in  $a$ . We define  $s(v)$  to be the path label of  $v$  *without* the a trailing \$ character.  $\mathcal{T}$  has exactly  $n$  leaves  $l^i$ , which are labeled with the numbers  $i \in \{0, \dots, n - 1\}$ , such that  $s(l^i) = a_{i+1} \dots a_n$ , i.e. the path labels of the leaves are the suffixes of the text. For any two leaves  $l^i$  and  $l^j$  exists a vertex  $v$  on the paths to  $l^i$  and  $l^j$  such that  $s(v)$  is the longest common prefix of  $s(l^i)$  and  $s(l^j)$ .

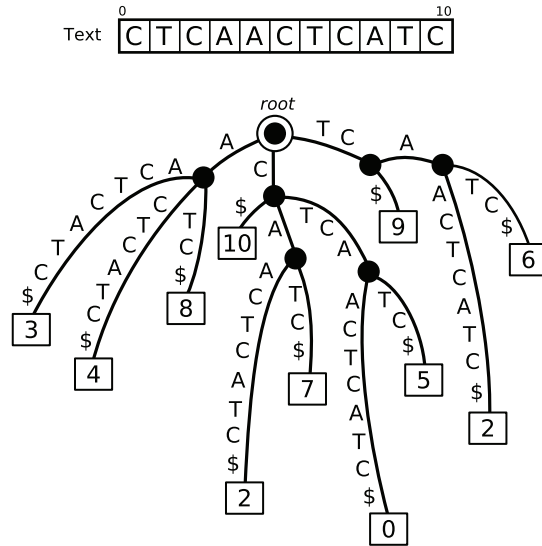


Figure 34: **Suffix Tree.** The suffix tree of "CTCAACTCATC".

The suffix tree is a versatile index data structure with many applications, see Gusfield (1997), chapters 5 to 9. SeqAn ‘emulates’ suffix trees by the more space efficient enhanced suffix array. An enhanced suffix array that

consists of the suffix table **ESA\_SA**, the LCP table **ESA\_LCP**, and the ‘child table’ **ESA\_ChildTab** is capable to replace the suffix tree in all of its applications (Abouelhoda et al. 2004). Algorithm 32 demonstrates that we only need suffix table  $S$  and LCP table  $L$  to emulate a *bottom-up traversal* of a suffix tree  $\mathcal{T}$ , that is to enumerate all vertices in  $\mathcal{T}$  in a way that the children appear earlier than their parents. **BOTTOMUPTRAVERSAL** reports for each vertex  $v$  in  $\mathcal{T}$  the set of its leaves. From the construction of  $\mathcal{T}$  follows, that the path labels of these leaves are the suffixes that share the common prefix  $s(v)$ , hence the labels of the leaves are listed consecutively in the suffix array, e.g. in  $S[l], \dots, S[r-1]$ , and all entries  $L[l+1], \dots, L[r-1] \geq |s(v)|$ , whereas  $L[l], L[r] < |s(v)|$ . **BOTTOMUPTRAVERSAL** therefore scans  $L$  for consecutive runs of values  $\geq m \in L$ , and this way, it finds all inner vertices of  $\mathcal{T}$ .

	$\triangleright$ <b>BOTTOMUPTRAVERSAL</b> ( $S, L$ )
1	<b>for each</b> $\langle l, r, m \rangle$ reported by <b>TRAVERSE</b> ( $S, L, 0, 0$ ) <b>do</b>
2	$\mathbf{L}$ report vertex in $\mathcal{T}$ with leaves $S[l], \dots, S[r-1]$

---

	$\triangleright$ <b>TRAVERSE</b> ( $S, L, l, m$ )
1	$r \leftarrow l + 1$
2	<b>while</b> $r < n$ <b>do</b>
3	<b>if</b> $L[r] < m$ <b>then break</b>
4	<b>else if</b> $L[r] > m$ <b>then</b>
5	$r \leftarrow \text{TRAVERSE}(S, L, r, L[r])$
6	<b>else</b> $r \leftarrow r + 1$
7	report $\langle l, r, m \rangle$
8	return $r$

Algorithm 32: **Emulated Bottom-up Traversal of a Suffix Tree.**  $S$  is the suffix array and  $L$  the LCP table of a length- $n$  text. A vertex of the suffix tree is represented by the set of its leaves. The last reported vertex is the root that covers all leaves of the tree.

SeqAn supports several iterators that emulate a traversal of a suffix tree, see Table 26. We will explain how these iterators work for the example of the **SuperMaxRepeats** iterator specialization, that computes all *supermaximal repeats* of the indexed text  $a$ , see Listing 28 for a code example. A *repeat* is a substring that occurs at least twice in the text, and it is *supermaximal*, if none of its occurrences is a substring of any other repeat. Let  $a$  be a text and  $\mathcal{T}$  the suffix tree of  $a$ , then for each supermaximal repeat  $w$  in  $a$  exists a vertex  $v$  in  $\mathcal{T}$ , such that  $s(v) = w$ . Now let  $v$  be any inner vertex of the suffix tree  $\mathcal{T}$  of  $a$ , and let  $I \subseteq \{0, \dots, n-1\}$  be the set of occurrences of  $s(v)$  in  $a$ , that is  $s(v) = a_{i+1} \dots a_{i+|s(v)|}$  for all  $i \in I$ . Then  $|I| \geq 2$ , and if for all  $i, j \in I$  holds (1)  $a_i \neq a_j$ , and (2)  $a_{i+|s(v)|+1} \neq a_{j+|s(v)|+1}$ , then  $s(v)$  is a supermaximal

repeat. This suggests the simple Algorithm 33 for finding all supermaximal repeats in  $a$  (Abouelhoda et al. 2002). SUPERMAXIMALREPEATS enumerate all inner vertices  $v$  in  $\mathcal{T}$  and reports each  $v$  that fulfills both conditions (1) and (2). Note that we can check these conditions in  $O(|\Sigma|)$ .

```

1  ▷ SUPERMAXIMALREPEATS ( $a_1 \dots a_n, S, L$ )
2  for each  $\langle l, r, m \rangle$  reported by TRAVERSE ( $S, L, 0, 0$ ) do
3      if  $r - l \geq 2$  then
4          if for each  $i, j \in \{l, \dots, r - 1\}$  holds:
              (1)  $a_{S[i]} \neq a_{S[j]}$  and
              (2)  $a_{S[i]+m+1} \neq a_{S[j]+m+1}$  then
                  report supermaximal repeat  $a_{S[l]+1} \dots a_{S[l]+m}$ 

```

Algorithm 33: **Finding All Supermaximal Repeats in a Text.**  $S$  is the suffix array and  $L$  the LCP table of the length- $n$  text  $a$ . In line 3, we define  $a_0 \neq c$  and  $a_{n+1} \neq c$  for each value  $c \in \Sigma$ .

```

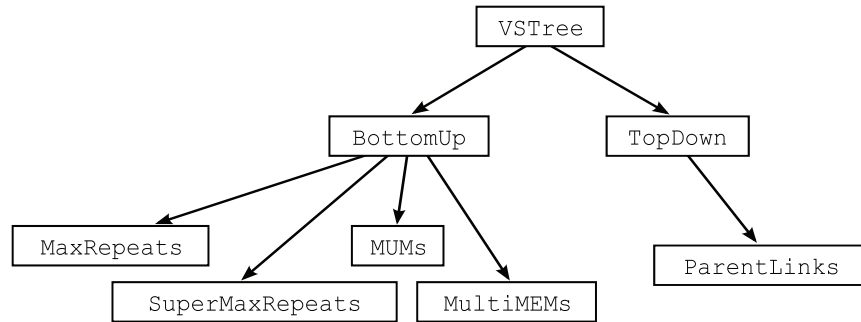
String<char> text = "How many wood would a woodchuck chuck.";
typedef Index< String<char> > TIndex;
TIndex idx(text);

Iterator<TMyIndex, SuperMaxRepeats>::Type it(idx, 3);
while (!atEnd(it))
{
    for (unsigned int i=0; i < countOccurrences(it); ++i)
    {
        cout << getOccurrences(it)[i] << ",";
    }
    cout << repLength(it) << ",";
    cout << representative(it) << endl;
    ++it;
}

```

Listing 28: **Searching Supermaximal Repeats.** This example finds all supermaximal repeats of length  $\geq 3$  in a text. For each supermaximal repeat, the program prints out the positions of its occurrences (`getOccurrences`), the length of the repeat (`repLength`), and the repeat string (`representative`).





<b>BottomUp</b>	Generic bottom-up iterator. It enumerates the vertices of the ‘emulated’ suffix tree during a post-order depth-first-search, see Algorithm 32.
<b>MaxRepeats</b>	A bottom-up iterator that enumerates all pairs of repeat occurrences that cannot be extended to the left or to the right.
<b>SuperMaxRepeats</b>	A bottom-up iterator that enumerates all supermaximal repeats, see Algorithm 33.
<b>MUMs</b>	A bottom-up iterator that enumerates maximal unique matches (MUMs) between two texts, i.e. all substrings that occur exactly once in both texts and that cannot be extended to the left or right.
<b>MultiMEMs</b>	Like MUMs, but for more than two texts.
<b>TopDown</b>	An iterator that allows to go further to any child of the current vertex. For this iterator, the child table <code>ESA_ChildTab</code> is required.
<b>ParentLinks</b>	Like <b>TopDown</b> but with the additional option to move from the current vertex to its parent. The iterator allows therefore any walk up and down through the ‘emulated’ suffix tree. It requires the child table <code>ESA_ChildTab</code> .

Table 26: **Hierarchy of Suffix Tree Iterators.**



# Chapter 13

## Graphs

A *graph*  $\mathcal{G}$  consists of a set  $V$  of *vertices* and a set  $E \subseteq V \times V$  of *edges* between the vertices.  $\mathcal{G}$  is called *undirected*, if for each edge  $e = \langle v, u \rangle \in E$  also the reverse  $\langle u, v \rangle \in E$ , otherwise  $\mathcal{G}$  is *directed*. For an edge  $\langle v, u \rangle$  of a directed graph we say that it ‘goes from  $v$  to  $u$ ’, and that the vertices  $v$  and  $u$  are ‘*adjacent*’.

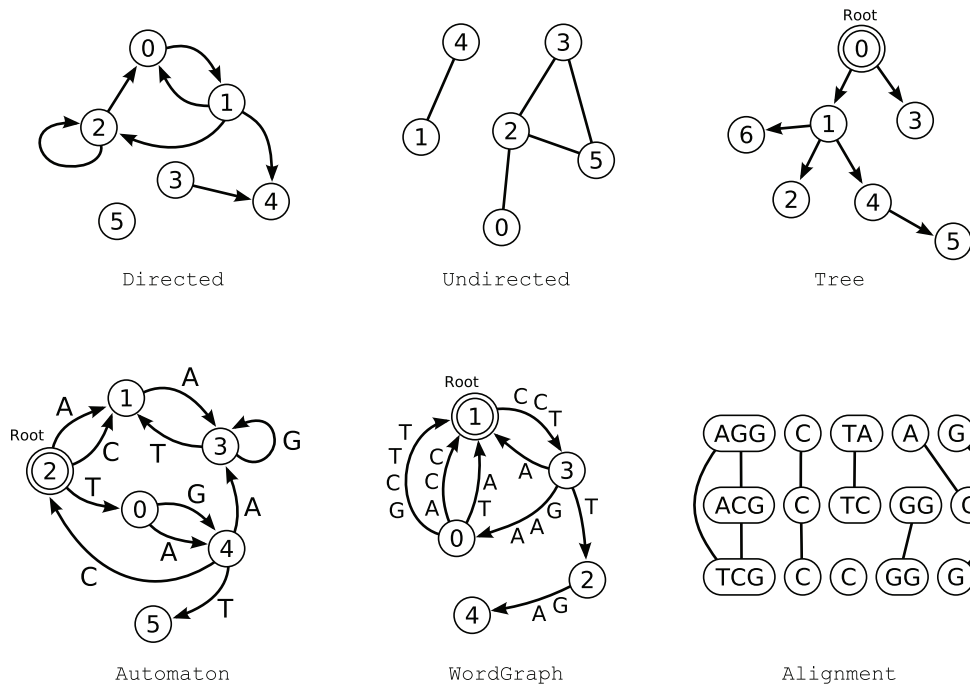
Graphs are very common in computer science, and they have also many applications in sequence analysis, for example automata (Section 13.1) or alignment graphs (Section 13.2). Although SeqAn is no declared graph library like the Boost Graph Library (Siek et al. 2002) or LEDA (Mehlhorn and Näher 1999), it offers a variety of graph types and algorithms. Graph data structures in SeqAn are implemented as specializations of the class `Graph`, see Table 27.

Functions like `addVertex`, `removeVertex`, `addEdge`, or `removeEdge` can be used to add or remove vertices and edges. Each vertex and each edge in a graph is identified by a so called ‘*descriptor*’. The usual descriptor type for vertices is `unsigned int`, it can be determined by calling the metafunction `VertexDescriptor`. The metafunction `EdgeDescriptor` returns the descriptor type for edges, which is usually a pointer to the data structure that holds information about the edge. The following example shows how to build up a simple graph:

```
typedef Graph< Directed<> > TGraph;
typedef VertexDescriptor<TGraph>::Type TVertexDescriptor;
typedef EdgeDescriptor<TGraph>::Type TEdgeDescriptor;

TGraph g;
TVertexDescriptor v = addVertex(g);
TVertexDescriptor u = addVertex(g);
TEdgeDescriptor e = addEdge(g, v, u);
```

Auxiliary information about vertices and edges like ‘vertex colors’ or ‘edge labels’ can be stored in *property maps*. Any container can be used as prop-



<b>Directed</b>	A general purpose directed graph. It stores for each vertex $v \in V$ an ‘adjacency list’ of all vertices $u \in V$ with $\langle v, u \rangle \in E$ .
<b>Undirected</b>	A general purpose undirected graph. As for <b>Directed</b> , the edges are stored in an adjacency list. Functions like <b>addEdge</b> for inserting or <b>removeEdge</b> for removing edges always affect both edges $\langle v, u \rangle$ and its reverse $\langle u, v \rangle$ .
<b>Tree</b>	One vertex of this directed graph is marked as ‘root’. A tree can be built up ‘from the root to the leaves’ by calling the function <b>addChild</b> .
<b>Automaton</b>	A graph with character labeled edges that can be used to scan sequences, see Section 13.1.
<b>WordGraph</b>	A subspecialization of <b>Automaton</b> that labels the edges with sequences instead of single characters.
<b>Alignment</b>	Alignment graphs are a very flexible way for storing alignments between two or more sequences, see Section 13.2.
<b>Hmm</b>	This graph type is used to store hidden Markov models (HMMs).

Table 27: **Graph Data Structures.** Specializations of the class **Graph**.

erty map; for example the following program stores an integer distance value associated to the edge `e` in the graph `g` in a `String<int>` object:

```
String<int> distances;
resizeEdgeMap(g, distances);
assignProperty(distances, e, 100);
cout << getProperty(distances, e);
```

There are several *iterators* in SeqAn for traversing vertices or edges, and to traverse graphs, see Table 28. This is demonstrated by the following example program that enumerates the vertices of the graph `g` and prints out their descriptors:

```
typedef Iterator<TGraph, AdjacencyIterator>::Type TAdjacencyIterator;
for (TAdjacencyIterator it(g); !atEnd(it) ;goNext(it))
{
    std::cout << *it << ",";
}
```

Vertex Iterators	
<code>VertexIterator</code>	Enumerates all vertices of a graph in increasing order of their descriptor.
<code>AdjacencyIterator</code>	Enumerates for a vertex $v$ all vertices $u$ such that $\langle v, u \rangle \in E$ .
<code>DfsPreorder</code>	Starting from a given vertex (e.g. the root in case of a <b>Tree</b> or <b>Automaton</b> ), this iterator enumerates all reachable vertices in depth-first-search ordering.
<code>BfsIterator</code>	Starting from a given vertex (e.g. the root in case of a <b>Tree</b> or <b>Automaton</b> ), this iterator enumerates all reachable vertices in breadth-first-search ordering.
Edge Iterators	
<code>EdgeIterator</code>	Enumerates all edges of a graph.
<code>OutEdgeIterator</code>	Enumerates for a vertex $v$ all edges $\langle v, u \rangle \in E$ .

Table 28: **Graph Iterators.** These tags are used as template arguments for the `Iterator` metafunction to select the iterator type for a **Graph** object.

SeqAn implements a variety of standard algorithms on graphs, see Table 29, most of them are described in Cormen et al. (2001). For lack of space we will focus on algorithms especially for automata (Section 13.1) and alignment graphs (Section 13.2).

Searching	Breadth-first-search: <code>breadth_first_search</code> . Depth-first-search: <code>depth_first_search</code> . (Both can also be done by iterators, see Table 28)
Topological Sort	<code>topological_sort</code> .
Components	<code>strongly_connected_components</code> .
Shortest Path	Single-source shortest path problem: <code>dag_shortest_path</code> , <code>dijkstra</code> , <code>bellman_ford_algorithm</code> . All-pairs shortest path problem: <code>floyd_warshall</code> .
Minimum Spanning Tree	<code>prims_algorithm</code> , <code>kruskals_algorithm</code> .
Maximum Flow	<code>ford_fulkerson</code> .
Transitive Closure	<code>transitive_closure</code> .

Table 29: **Overview of Common Graph Algorithms in SeqAn.** We omit here algorithms especially designed for automata (see Section 13.1) and alignment graphs (see Section 13.2).

## 13.1 Automata

The specialization **Automaton** of **Graph** serves the purpose of storing *deterministic finite automata* (dfa). A dfa  $\mathcal{G}$  is a directed graph that allows multiple edges  $\langle u, v \rangle$  between two vertices  $u$  and  $v$ . The edges are labeled with characters such that two different edges  $\langle u, v \rangle$  and  $\langle u, v' \rangle$  going out from the same vertex  $u$  have different labels. A certain vertex called the ‘root’ can be used as a starting point for a run through the automaton guided by a coincident scan through a sequence: Let  $r = v_1, v_2, \dots, v_k = v$  be a path  $p$  in  $\mathcal{G}$ , and let  $s^{i-1,i}$  be the label of the edge from  $v_{i-1}$  to  $v_i$ , then we call  $s(p) = s^{1,2}s^{2,3} \dots s^{k-1,k}$  the *path label* of  $p$ . By definition, deterministic automata are constructed such that  $s(p^1) \neq s(p^2)$  for two different paths  $p^1$  and  $p^2$  both starting in the root. Let  $a = a_1 \dots a_n$  be a string, then we say that a graph  $\mathcal{G}$  *scans*  $a$ , if there is a path  $p$  starting from the root in  $\mathcal{G}$  with  $s(p) = a$ . If  $\mathcal{G}$  scans  $a$ , then it obviously also scans all prefixes  $a_1 \dots a_i$  of  $a$ . Finding the maximal prefix that is scanned by a given graph is part of pattern matching algorithms like BFAM (Section 10.1.4) or MULTIBFAM (Section 10.2.2), and it can also be done in SeqAn by calling the function `parseString`.

Note that **Graph** objects do not store a set of ‘accept states’ as it is usually supposed in the literature about automata theory (e.g. Hopcroft and Ullman, 1990). If accept states are needed, then we can use a property map of `bool` to store for each vertex whether it is accepting or not.

In the following, we will discuss two special kinds of automata in more detail, namely *tries* (Section 13.1.1) and *factor oracles* (Section 13.1.2).

### 13.1.1 Tries

A dfa is called a ‘*trie*’, if it is also a tree, i.e. if the root has no incoming edges and there is for each vertex  $u$  a unique path from the root to  $u$ , see Figure 35.

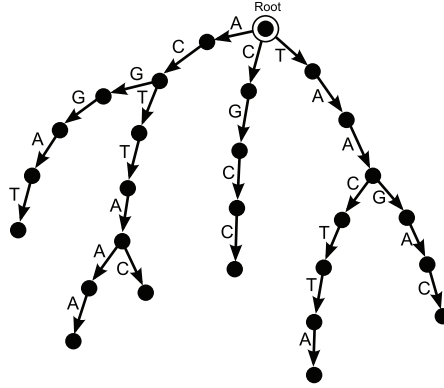


Figure 35: **Trie.** The trie of the sequences "ACGGAT", "ACTTAAA", "ACTTAC", "CGCC", "TAACTTA", and "TAAGAC".

‘The’ trie of a set of sequences  $a^1, \dots, a^d$  is the unique minimal trie that scans all sequences  $a^i$ . In SeqAn, the function `createTrie` implements the simple algorithm BUILDTRIE (Algorithm 34) for building up ‘the’ trie for a given set of sequences. This takes  $O(n)$  time and space, where  $n$  is the sum of the sequence lengths.

<pre> ▷ BUILDTRIE (<math>a^1, a^2, \dots, a^d</math>) 1  <math>\mathcal{G} \leftarrow</math> graph that only contains the root <math>r</math> 2  <b>for</b> <math>i \leftarrow 1</math> <b>to</b> <math>d</math> <b>do</b> 3      <math>n^i \leftarrow</math> length of <math>a^i</math> 4      <math>k \leftarrow \max\{j \leq n^i \mid \mathcal{G} \text{ scans the prefix } a_1^i \dots a_j^i\}</math> 5      <b>if</b> <math>k &lt; n^i</math> <b>then</b> 6          <math>v \leftarrow</math> the vertex in <math>\mathcal{G}</math> with <math>s(v) = a_1^i \dots a_k^i</math> 7          append to <math>v</math> a new branch with labels <math>a_k^i \dots a_{n^i}^i</math> </pre>	}	add $a^i$ to $\mathcal{G}$
--	---	----------------------------

Algorithm 34: **Trie Construction.** The algorithm builds up the trie of the sequences  $a^1, a^2, \dots, a^d$ .

The *suffix trie* of a sequence  $a = a_1 \dots a_n$  is the trie of all suffixes  $a_j \dots a_n$ . A suffix trie of  $a$  scans exactly the substrings of  $a$ . The function `createSuffixTrie` can be used to construct suffix tries.

### 13.1.2 Factor Oracles

Suffix tries for sequences  $a^1, \dots, a^d$  of total length  $n$  have worst case size  $\Omega(n^2)$ . Allauzen, Crochemore, and Raffinot (1999) proposed an alternative dfa called *factor oracle* that also scans all suffixes  $a^1, \dots, a^d$ , but has only  $\leq n+1$  vertices and  $\leq 2n$  edges. The factor oracle may – other than the *factor trie* – also scan sequences that are no substrings of any  $a^i$ . For example, the factor oracle in Figure 36 scans "CAC", which is no substring of "CTCAACTCATC".

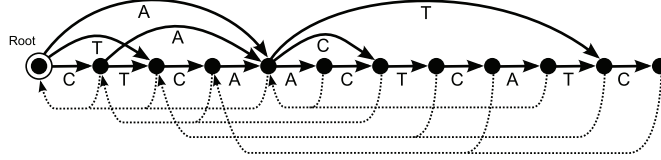


Figure 36: **Factor Oracle.** The oracle of the sequence "CTCAACTCATC". The dotted arrows visualize the supply array  $S$ .

Algorithm 35 shows how to construct a factor oracle in linear time by adding at most  $n$  more edges to the trie of  $a^1, \dots, a^d$ . BUILDORACLE traverses the trie starting from the root  $r$ . Let  $u \in V$  be a vertex and  $s(u) = s_1 \dots s_m$  the label of the path from  $r$  to  $u$  in the trie, let  $v \in V$  be the predecessor to  $u$  on this path, and let  $c$  be the label of its last edge  $\langle v, u \rangle$ . When the main loop reaches  $u$ , then the algorithm extends  $\mathcal{G}$  to ensure that it scans all suffixes of  $s(u)$ . So  $\mathcal{G}$  will scan all suffixes of  $a^1, \dots, a^d$  once all vertices are processed.

Since the vertex  $v$  was processed before  $u$ , we already made  $\mathcal{G}$  to scan all suffixes of  $s(v) = s_1 \dots s_{m-1}$ . Let  $v^i$  be the vertex that is reached if we scan in  $\mathcal{G}$  for the  $i$ -th suffix  $s(v^i) = s_i \dots s_{m-1}$ , where  $i \in \{0, \dots, m-1\}$ . We want  $\mathcal{G}$  to scan also the suffixes  $s(v^i)c$  of  $s(u)$ , so we just need to take care that each  $v^i$  has a  $c$ -edge, which means that  $v^i$  has an outgoing edge labeled with  $c$ . For  $v^0 = v$  such an edge already exists in the trie; all other  $v^i$  can be found by following a linked list stored in the 'supply array'  $S$ . BUILDORACLE constructs  $S$  in a way that  $S[v]$  is the vertex  $v^i \neq v$  with minimal  $i$ , i.e.  $s(S[v])$  is the longest suffix of all suffixes  $s(v^i)$  with  $v^i \neq v$ . All  $s(v^i)$  with  $v^i \neq v$  are suffixes  $s(S[v])$ , so the next largest suffix of  $s(v)$  smaller than  $s(S[v])$  is  $s(S[S[v]])$ . The set  $\{v, S[v], S^2[v], S^3[v], \dots, r\}$  therefore contains all vertices  $v^i$  in descending order their suffix lengths. Enumerating the vertices  $u$  in breadth-first search order (line 4) ensures that the supply values of all vertices  $v^i$  are already computed.

For each vertex  $S^j[v]$  without  $c$ -edge, we add in line 12 an edge  $\langle S^j[v], u \rangle$  labeled with  $c$ . From now on  $\mathcal{G}$  scans the suffix  $s(S^j[v])c$  of  $s(u)$ . Since each scan that reaches  $u$  can go further in the trie,  $\mathcal{G}$  now also scans all suffixes of  $a^1, \dots, a^d$  that start with  $s(S^j[v])c$ . There exists at least one suffix of that kind, and for this suffix we will never need to insert another edge into  $\mathcal{G}$ . The number of edges added in line 12 of BUILDORACLE is therefore bounded by the number of suffixes of  $a^1, \dots, a^d$ , i.e. it is  $\leq n$ .



Note that we can stop the enumeration of the  $S^j[v]$  as soon as we found a vertex  $S^j[v]$  with a  $c$ -edge  $\langle S^j[v], w \rangle$ , since in this case,  $w$  is a vertex that was already processed by  $\mathcal{G}$ , and therefore all further vertices  $S[v^i], S^2[v^i], \dots$  have  $c$ -edges. Therefore the total runtime of BUILDORACLE is  $O(n)$ .

```

1  ▷ BUILDORACLE ( $a^1, a^2, \dots, a^d$ )
2   $\mathcal{G} = \langle V, E \rangle \leftarrow \text{BUILDTRIE}(a^1, a^2, \dots, a^d)$ 
3   $r \leftarrow \text{root of } \mathcal{G}$ 
4   $S[r] \leftarrow \text{nil}$ 
5  for each  $u \in V \setminus \{r\}$  in bfs order do
6     $v \leftarrow \text{the predecessor vertex of } u \text{ in the trie}$ 
7     $c \leftarrow \text{label of } \langle v, u \rangle$ 
8    repeat
9       $v \leftarrow S[v]$ 
10     if exists  $\langle v, w \rangle \in E$  labeled with } c then
11        $S[u] \leftarrow w$ 
12       break
13     insert edge  $\langle v, u \rangle$  into  $E$  with label  $c$ 
14     if  $v = r$  then
15        $S[u] \leftarrow r$ 
16       break
17 return  $\mathcal{G}$ 

```

Algorithm 35: **Factor Oracle Construction.** The algorithm builds up the oracle for the sequences  $a^1, a^2, \dots, a^d$ .

SeqAn implements this algorithm in the function `createOracle`.

## 13.2 Alignment Graphs

### 13.2.1 Alignment Graph Data Structure

Alignment graphs are, beside **Align** data structures (see Section 9.2), the second representation for alignments in SeqAn. They were initially introduced by Kececioğlu (1993) and later extended by Kececioğlu et al. (2000).

An *alignment graph*  $\mathcal{G}$  (see Figure 37) for  $d$  sequences  $a^1, a^2, \dots, a^d$  is an undirected  $d$ -partite graph with a set  $V$  of vertices and a set  $E$  of edges that meet the following criteria:

- (1)  $V = V^1 \cup V^2 \cup \dots \cup V^d$ , where  $V^i$  partitions  $a^i$  into non-overlapping segments (for  $1 \leq i \leq d$ ), i.e. each value in  $a^i$  belongs to exactly one vertex in  $V^i$ .

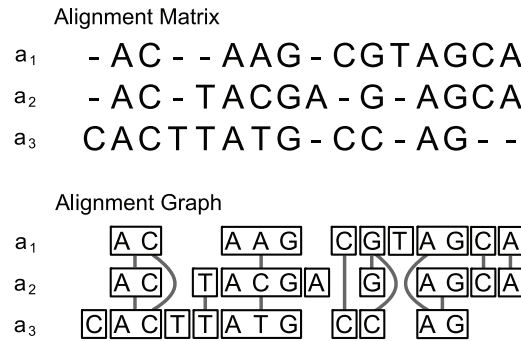


Figure 37: **Alignment Matrix and Alignment Graph.** An alignment of three sequences  $a^1$ ,  $a^2$ , and  $a^3$ , displayed both in matrix style and as alignment graph.

- (2)  $E \subseteq \{\langle v^i, v^j \rangle \mid v^i \in V^i \text{ and } v^j \in V^j \text{ and } 1 \leq i, j \leq d \text{ and } i \neq j \text{ and the segments } v^i \text{ and } v^j \text{ have the same length}\}.$

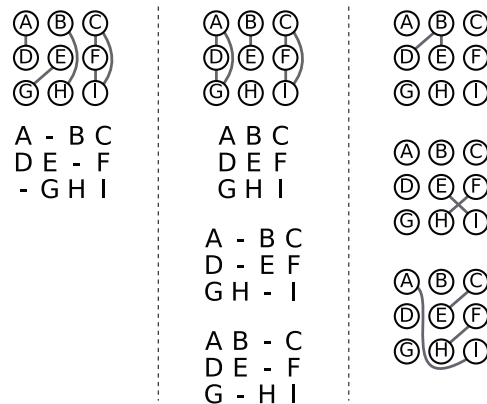


Figure 38: **Alignment Graph Examples.** For the sequences "ABC", "DEF", and "GHI".

**Left:** A unique trace and the compatible alignment. **Middle:** A non-unique trace and compatible alignments. The last two alignments contain changing gaps at columns 2 and 3. **Right:** Three alignment graphs that are no traces.

An alignment  $\mathcal{A}$  and an alignment graph  $\mathcal{G}$  are *compatible*, if for each edge  $\langle v^i, v^j \rangle$  in  $\mathcal{G}$ , the segments  $v^i$  and  $v^j$  are aligned in  $\mathcal{A}$  without gaps. An alignment graph that is compatible to at least one alignment is called a *trace*, and we call a trace '*unique*', if it is compatible to exactly one alignment. Figure 37 and Figure 38 (left) show examples for unique traces.

Some alignments are not compatible to any unique trace, because they *contain changing gaps* (see Figure 38, middle), i.e. in the alignment are two flanking columns  $i$  and  $i + 1$  that together contain at most one value of each sequence. Note that optimal alignments usually do not contain changing gaps, since, for reasonable scoring schemes, the score gets better when the two columns  $i$  and  $i + 1$  are merged (see Section 9.3.1).

For an alignment  $\mathcal{A}$  that does not contain changing gaps, the alignment graph

$\mathcal{G} = \langle V, E \rangle$  with  $V := \{a \mid a \text{ is a value in one of the sequences of } \mathcal{A}\}$  and  $E := \{\langle a, b \rangle \mid a \text{ and } b \text{ are aligned in } \mathcal{A}\}$  is a unique trace compatible to  $\mathcal{A}$ . This alignment graph is constructed for a given `Align` object when calling the function `convertAlignment`. Another function of the same name can be used to convert a unique trace into its compatible alignment.

### 13.2.2 Maximum Weight Trace Problem

Most algorithms in SeqAn for computing optimal global alignments (Section 9.5) or local alignments (Section 11.1) accept alignment graphs for storing the results. Beside of that, we can also use alignment graphs to formulate alignment problems in a new way (Kececioğlu 1993): Given an alignment graph  $\mathcal{G}$  and scores  $weight(e) \geq 0$  for each edge  $e$  in  $\mathcal{G}$ , then the *maximum weight trace problem* is to find a trace  $\mathcal{G}^*$  that is a subgraph of  $\mathcal{G}$  and for which the sum of the edge scores is maximal. This kind of alignment problem is especially interesting for ‘sparse’ alignment graphs  $\mathcal{G} = \langle V, E \rangle$ , because in this case algorithms exists that are more efficient than e.g. the Needleman-Wunsch algorithm (Section 9.5.1), which takes exponential time in the number of sequences.

In the following we concentrate on pairwise alignment graphs  $\mathcal{G} = \langle V^1 \cup V^2, E \rangle$  between two sequences  $a^1$  and  $a^2$ . Since the edges  $\langle v^1, v^2 \rangle \in E$  can be considered as *seeds* between the two segments  $v^1 \in V^1$  and  $v^2 \in V^2$ , and since the segments in  $V^1$  and  $V^2$  do not overlap, the maximum weight trace problem can be formulated as a global chaining problem, see Section 9.6.3. On the other hand, one can also reduces the maximum weight trace problem to the ‘heaviest common subsequence problem’ (Jacobson and Vo 1992), for which SeqAn implements an efficient algorithm in function `heaviestCommonSubsequence`, that allows to compute a maximum weight trace in time  $O(|E| \log |E|)$ . In fact, this algorithm is equivalent to a simplified version of SPARSECHAINING (Algorithm 7, page 99), which uses *sparse dynamic programming* for global chaining, see Section 9.6.3.

$V^1$  is a partition of sequence  $a^1$ , and each seed covers only one segment of  $a^1$ , hence MAXWEIGHTTRACE (Algorithm 36) needs not to handle the ‘left’ and ‘right’ positions of the seeds separately, as it was done in SPARSECHAINING. Let for both  $i \in \{1, 2\}$  the segments in the sets  $V^i = \langle v_1^i, v_2^i, \dots \rangle$  be ordered according to their occurrences in  $a^i$ , and for each edge  $e_j = \langle v_p^1, v_q^2 \rangle \in E$  we define  $pos_1(e_j) = p$  and  $pos_2(e_j) = q$ . A seed  $e_j \in E$  can be *appended* to another seed  $e_k \in E$ , if  $pos_i(e_1) < pos_i(e_2)$  for both  $i \in \{1, 2\}$ , and two edges  $e_j, e_k \in E$  can only be part of the same trace  $\mathcal{G}^*$ , if one of them can be appended to the other.

The algorithm enumerates the seeds  $e_j$  in increasing order of their positions  $pos_1(e_j)$  (line 4), searches in a set  $D$  of active seeds for an optimal predecessor  $e_{T_j}$  (line 5), computes the score of the best chain ending in  $e_j$  (lines 6 to 9), deletes all seeds  $e_k$  in  $D$  that are ‘dominated’ from  $e_j$ , i.e. those  $e_k$  with

```

1  ▷ MAXWEIGHTTRACE ( $\mathcal{G} = \langle V, E = \{e_1, \dots, e_n\} \rangle, w$ )
2  sort  $e_j \in E$  in decreasing order of  $pos_2(e_j)$ 
3  stable sort  $e_j \in E$  in increasing order of  $pos_1(e_j)$ 
4   $D \leftarrow \emptyset$ 
5  for each  $e_j \in E$  in sorted order do
6       $T_j \leftarrow \operatorname{argmax}_{k \in D} \{ pos_2(e_k) \mid pos_2(e_k) < pos_2(e_j) \}$ 
7      if  $T_j$  is defined then
8           $M_j \leftarrow M_{T_j} + weight(e_j)$ 
9      else
10          $M_j \leftarrow weight(e_j)$ 
11     for each  $k \in D$  with  $pos_2(e_k) \geq pos_2(e_j)$  and
12          $M_k \leq M_j$  do
13          $D \leftarrow D \setminus \{k\}$ 
14      $D \leftarrow D \cup \{j\}$ 
15      $j \leftarrow$  last element of  $D$ 
16      $E^* \leftarrow \{e_j\}$ 
17     while  $T_j$  is defined do
18          $j \leftarrow T_j$ 
19          $E^* \leftarrow E^* \cup \{e_j\}$ 
20 return  $\mathcal{G}^* = \langle V, E^* \rangle$ 

```

} find best chain to  $e_j$   
 } update  $D$   
 } build new edge set  $E^*$

Algorithm 36: **Maximum Weight Trace by Sparse Dynamic Programming.** The algorithm is a simplified variation of Algorithm 7 on page 99. It computes a maximum weight trace subgraph  $\mathcal{G}^*$  of  $\mathcal{G}$ , where  $w(e) \geq 0$  are the weights of the edges in  $\mathcal{G}$ . The sorted set  $D$  stores all active seeds,  $M_j$  is the score of the best chain that ends with  $e_j$ , and  $T_j$  the predecessor of  $e_j$  in that chain. Note that  $\operatorname{argmax}$  in line 5 returns ‘undefined’ if it is applied to an empty set. In this case  $e_j$  has no predecessor.

$pos_2(e_k) \geq pos_2(e_j)$  and smaller chain score  $M_k \leq M_j$  (lines 10 to 11), and finally inserts  $e_j$  into  $D$  (line 12). To take care that  $e_j$  is not appended to another seed  $e_k$  with  $pos_1(e_k) = pos_1(e_j)$ , we enumerate seeds with equal position  $pos_1$  in decreasing order of their position  $pos_2$  (line 1). At the end, all edges on the trace back starting from the last item in  $D$  are added to  $\mathcal{G}^* = \langle V, E^* \rangle$ , which is a maximal weight trace subgraph of the input graph  $\mathcal{G}$ .

### 13.2.3 Segment Match Refinement

A good strategy for getting an alignment graph  $\mathcal{G} = \langle V, E \rangle$  which is ‘sparse enough’ to be a viable input for Algorithm 36 of Section 13.2.2 is to add only those edges to  $V$  that have a good chance for becoming part of the optimal alignment, i.e. edges that connect high scoring matches between the sequences  $a^1, \dots, a^d$ . If we want to construct an alignment graph  $\mathcal{G}$  for a given set  $S = \{\mathcal{S}_1, \dots, \mathcal{S}_n\}$  of *seeds*, where each  $\mathcal{S}_j$  aligns segments of two sequences  $\in \{a^1, \dots, a^d\}$ , then this could be problematic because (1) alignment graphs allow only matches between segments of *equal length*, and (2) it is possible that two seeds  $\mathcal{S}_j$  and  $\mathcal{S}_k$  *overlap*, i.e. the aligned segments overlap. Figure 39



Figure 39: **Segment Match Refinement.**

shows the solution for both problems: We need to cut the seeds into a set of parts  $R = \{\mathcal{S}'_1, \dots, \mathcal{S}'_m\}$  such that (1) each  $\mathcal{S}'_j \in R$  aligns two segments of the same length, and (2) a segment that is aligned by a seed  $\mathcal{S}'_j$  is either identical or disjoint to the segments that is aligned by any other seed  $\mathcal{S}'_k \in R$ . Condition (1) is automatically done when the alignment is transformed into its alignment graph representation. Finding a *refinement*  $R$  of minimal size that fulfills condition (2) is called the *segment match refinement problem* (Halpern, Huson, and Reinert 2002). SeqAn implements an algorithm `matchRefinement` that solves this problem for an arbitrary number of sequences (Emde 2007).



# Part IV

## Discussion

*In the final part of this thesis, we will argue that our library is actually capable to fulfill its purpose. In Chapter 14, we will discuss the library's quality, and we explain our measures for quality assurance and propagation of the library. A short program that we propose in Chapter 15 demonstrates how SeqAn can be used to create software tools. Chapter 16 offers a summery and gives an outlook of future plans with SeqAn.*





# Chapter 14

## Library Quality and Applicability

This chapter is about the quality of SeqAn and its core design. First we ask the question: “How good is our *library design*?” There are two possibilities to answer this question: We can either analyze whether the design actually complies with the goals we described in Section 4.2, i.e. we say that our library design is ‘good’, if it excellently meets all design goals. This will be discussed in Section 14.1.

An alternative approach is to derive a statement about the quality of the design from the quality of the library that was implemented using this design, i.e. we say that the design is ‘good’ if it facilitated the implementation of a ‘good’ library. Then the next question is: “What makes a library ‘good’?” The obvious answer is: “A library is ‘good’, if it serves its purpose well, namely the design of algorithms and software tools.” We will discuss this in Section 14.2.

### 14.1 Design Quality

We will now show that our library design (see Chapter 5) complies to all design goals that we demanded in Section 4.2.

#### 14.1.1 Performance

The core design supports the *performance* of the library by enabling *refinements* of data structures and algorithms such that faster solutions for special cases can be implemented. Template subclassing (Section 5.3) allows for faster code than classical *object-oriented programming*, because it avoids the overhead of virtual functions. We argued in Section 5.1 that the choice of using the programming language C++ also results in better performance. During the implementation of SeqAn we put a special emphasis on optimization; for example we abstained in the release version of the library from any time consuming tests about the coherence of function arguments.

We proved the success of our efforts by simply conducting runtime analyses for the various algorithms, for example in Part III of this thesis. Another example can be found in Table 30, which shows that SeqAn provides the fastest edit distance alignment algorithm (Myers-Hirschberg) of all tested libraries.

	linear gap costs		affine gap costs	
	time (s)	space (MB)	time (s)	space (MB)
<b>SeqAn</b>				
Needleman-Wunsch	3.3	236	6.3	236
Hirschberg			14.7	4
Myers-Hirschberg	<b>0.2</b>	3		
<b>NCBI C++ toolkit</b>				
Needleman-Wunsch			<b>4.0</b>	245
Hirschberg			6.6	14
<b>Bio++</b>	<b>13.4</b>	2100	28.0	≈6000
<b>BTL</b>			<b>96162</b>	933
<b>BioJava</b>	<b>76</b>	2000	93	≈6000

Table 30: **Runtimes and Internal Space Requirements for Computing Sequence Alignments.** The table shows average time and space requirements for aligning the genomes of two human influenza viruses, each of length about *15.6kbp*, using alignment functions in SeqAn, the NCBI C++ toolkit, Bio++, the BTL, and BioJava (see Section 2.2.2 on page 12). Runtimes printed in bold face show for each library the time of the fastest algorithm for computing an alignment using *edit distance*.

### 14.1.2 Simplicity

We showed in Chapter 5 that the core design of SeqAn consists only of a handful of techniques, each of limited complexity. Although our approach may be not as common for most programmers than standard object-oriented programming, we think that the code of the library and possible error messages during compiling are still rather understandable. Since *template subclassing* supports both *generality* and *refinements*, the code becomes less redundant. The library design facilitates not only the implementation of the library itself but also its application for programming tools: *Polymorphism* and the application of *tag dispatching* make the interface more comprehensible, and *global interfaces* avoid the confusion between ‘algorithms’ and ‘other functions’ of the STL. The range of functionality in SeqAn comprises many elements that improve coding convenience. For example, SeqAn offers many *shortcuts* for frequently used classes (see Section 5.6.3) like `DnaString` for `String< Dna, Alloc<> >`.

The simplicity of SeqAn is demonstrated in various code examples that can be found throughout this thesis.

### 14.1.3 Generality

*Generic programming* is one of the main techniques for SeqAn (see Section 5.2) and the library uses both generic data structures and generic algorithms that can be applied to all suitable types. The fact that we also pursue the *integration* goal furthermore expands the library's area of application. During the implementation, we tried to find the most general abstract data types that solve the problem without loss of performance. In Chapter 15 of this thesis, we showed that the resulting data structures and algorithms are suitable for solving specific problems. Note that SeqAn was designed for multiple platforms and compilers, see Section 14.2.3.

### 14.1.4 Refineability

*Template subclassing* allows to define unlimited hierarchies of refinements. Within this pedigree of derivation, every single function on every hierarchical level can be overloaded and replaced by a better alternative. We demonstrated this e.g. in Chapter 6. Another example are the numerous `String` data types (Section 8.3) or the rich hierarchy of suffix tree iterators (Section 12.3.2).

### 14.1.5 Extensibility

In contrast to member functions that can only be added by changing the class definition, global functions and metafunctions can be added at any time to the library without changing it. Therefore we can easily extend the library, either by defining completely new functionality, or by overloading already existing functions or metafunctions. Creating new class refinements is also simple, because it only requires a new tag class and possibly the definition of a corresponding template specialization. In order to achieve optimal *extensibility*, every base class template and every tag class in SeqAn has the additional template parameter `TSpec` for defining further specializations. After all, it was only possible to implement SeqAn *because* it was extensible, since nobody can program a software of this size in a single effort.

### 14.1.6 Integration

The idea of *global interfaces* imply the possibility of using *shims*, which make the library adaptable both for additional external data structures and for built-in types. We demonstrated in Section 6.1, that algorithms in SeqAn may be generic to an extend that we called 'library spanning programming', because

they can be used for data structures from arbitrary sources, as soon as the necessary shims are available. SeqAn comes with an adaptor for `basic_string` of the standard library (and its iterators), as well as for C-style strings, i.e. for zero-terminated `char` arrays. However, it is also quite possible to integrate other third party libraries easily into SeqAn.

## 14.2 Stability, Usability, Accessibility

The *quality* of a library has at least the following two aspects:

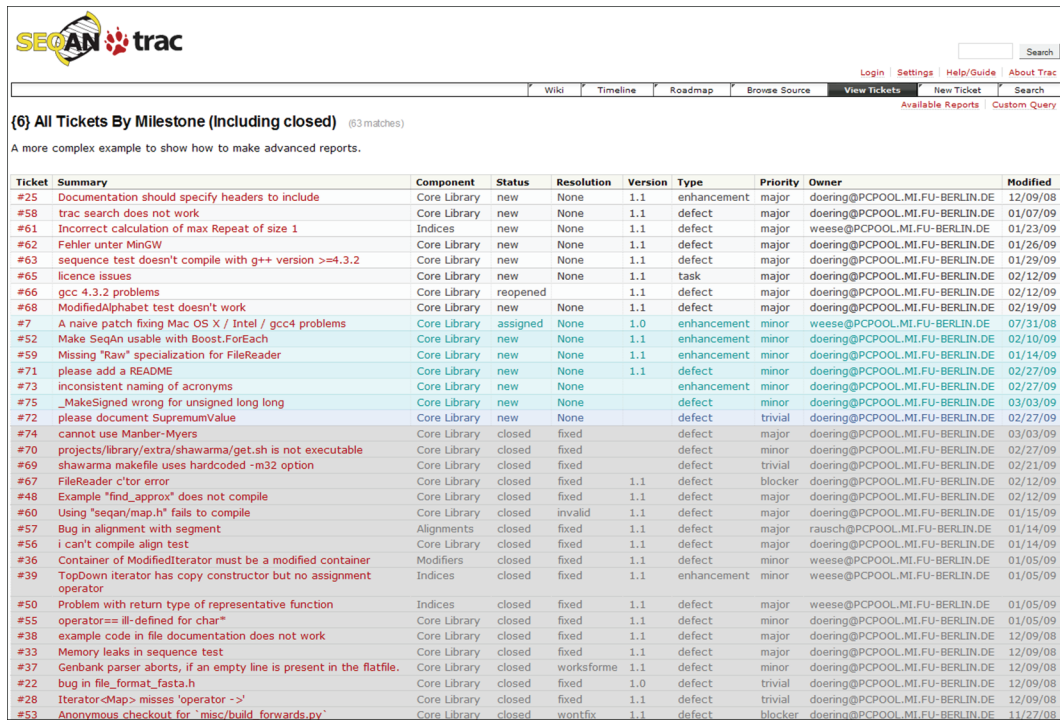
- (1) **Stability:** “*Does the library’s functionality works properly?*” The fact that SeqAn was already applied in some projects shows that its stability was obviously high enough at least in these cases. We discuss in Section 14.2.1 our practice for *testing* all parts of the library.
- (2) **Usability:** “*Does the library meets its intended purpose and can actually be applied by the users to facilitate the software or algorithm development process?*” SeqAn was already applied for the development of several state-of-the-art software, e.g. Schulz et al. (2008), Weese and Schulz (2008) Rausch et al. (2008), Langmead et al. (2009), and Rausch et al. (2009). This demonstrates its usability. We will also substantiate this claim in Chapter 15, where we use SeqAn to re-implement the well known tool LAGAN (Brudno et al. 2003). The library’s compliance to our design goals (see Section 14.1) is also an argument for its usability.

One prerequisite for the usability of a software library is a comprehensive documentation, see Section 14.2.2. Section 14.2.3 describes our efforts to make SeqAn *accessible*, that is how we tried to prepare the way for the user to get and apply our library.

### 14.2.1 Testing

Each published part of SeqAn is exhaustively tested. We used two testing strategies (Myers et al. 2004):

- (1) **Unit Testing:** For each module of SeqAn exists a program which tests all data structures and functions that reside in this module. These are mainly *black-box* tests, i.e. the program does not inspect the actual implementation but only check the correctness of output generated by the tested library part. In many cases the input data is predefined within in the test program; some other tests generate repeatedly random inputs and compare the outputs of alternative implementation.



The screenshot shows the SeqAn Trac interface. At the top, there's a search bar and navigation links like 'Wiki', 'Timeline', 'Roadmap', 'Browse Source', 'View Tickets', 'New Ticket', and 'Search'. Below this, a header indicates '(6) All Tickets By Milestone (Including closed) (63 matches)'. A note says 'A more complex example to show how to make advanced reports.' The main part of the image is a table of tickets with columns: Ticket, Summary, Component, Status, Resolution, Version, Type, Priority, Owner, and Modified. The table lists various issues, such as documentation problems, compilation errors, and feature requests, with their respective statuses and owners.

Ticket	Summary	Component	Status	Resolution	Version	Type	Priority	Owner	Modified
#25	Documentation should specify headers to include	Core Library	new	None	1.1	enhancement	major	doering@PCPOOL.MI.FU-BERLIN.DE	12/09/08
#58	trac search does not work	Core Library	new	None	1.1	defect	major	doering@PCPOOL.MI.FU-BERLIN.DE	01/07/09
#61	Incorrect calculation of max Repeat of size 1	Indices	new	None	1.1	defect	major	weese@PCPOOL.MI.FU-BERLIN.DE	01/23/09
#62	Fehler unter MinGW	Core Library	new	None	1.1	defect	major	doering@PCPOOL.MI.FU-BERLIN.DE	01/26/09
#63	sequence test doesn't compile with g++ version >=4.3.2	Core Library	new	None	1.1	defect	major	doering@PCPOOL.MI.FU-BERLIN.DE	01/29/09
#65	licence issues	Core Library	new	None	1.1	task	major	doering@PCPOOL.MI.FU-BERLIN.DE	02/12/09
#66	gcc 4.3.2 problems	Core Library	reopened	1.1	defect	major	major	doering@PCPOOL.MI.FU-BERLIN.DE	02/12/09
#68	ModifiedAlphabet test doesn't work	Core Library	new	None	1.1	defect	major	doering@PCPOOL.MI.FU-BERLIN.DE	02/19/09
#7	A naive patch fixing Mac OS X / Intel / gcc4 problems	Core Library	assigned	None	1.0	enhancement	minor	weese@PCPOOL.MI.FU-BERLIN.DE	07/31/08
#52	Make SeqAn usable with Boost.ForEach	Core Library	new	None	1.1	enhancement	minor	doering@PCPOOL.MI.FU-BERLIN.DE	02/10/09
#59	Missing "Raw" specialization for FileReader	Core Library	new	None	1.1	enhancement	minor	doering@PCPOOL.MI.FU-BERLIN.DE	01/14/09
#71	please add a README	Core Library	new	None	1.1	defect	minor	doering@PCPOOL.MI.FU-BERLIN.DE	02/27/09
#73	inconsistent naming of acronyms	Core Library	new	None	1.1	enhancement	minor	doering@PCPOOL.MI.FU-BERLIN.DE	02/27/09
#75	_MakeSigned wrong for unsigned long long	Core Library	new	None	1.1	defect	minor	doering@PCPOOL.MI.FU-BERLIN.DE	03/03/09
#72	please document SupremumValue	Core Library	new	None	1.1	defect	trivial	doering@PCPOOL.MI.FU-BERLIN.DE	02/27/09
#74	cannot use Member-Myers	Core Library	closed	fixed	1.1	defect	major	doering@PCPOOL.MI.FU-BERLIN.DE	03/03/09
#70	projects/library/extra/shawarma/get.sh is not executable	Core Library	closed	fixed	1.1	defect	minor	doering@PCPOOL.MI.FU-BERLIN.DE	02/27/09
#69	shawarma makefile uses hardcoded -m32 option	Core Library	closed	fixed	1.1	defect	trivial	doering@PCPOOL.MI.FU-BERLIN.DE	02/21/09
#67	FileReader c'tor error	Core Library	closed	fixed	1.1	defect	blocker	doering@PCPOOL.MI.FU-BERLIN.DE	02/12/09
#48	Example "find_approx" does not compile	Core Library	closed	fixed	1.1	defect	major	doering@PCPOOL.MI.FU-BERLIN.DE	02/12/09
#60	Using "seqan/map.h" fails to compile	Core Library	closed	invalid	1.1	defect	major	doering@PCPOOL.MI.FU-BERLIN.DE	01/15/09
#57	Bug in alignment with segment	Alignments	closed	fixed	1.1	defect	major	rausch@PCPOOL.MI.FU-BERLIN.DE	01/14/09
#56	i can't compile align test	Core Library	closed	fixed	1.1	defect	major	doering@PCPOOL.MI.FU-BERLIN.DE	01/14/09
#36	Container of ModifiedIterator must be a modified container	Modifiers	closed	fixed	1.1	defect	minor	weese@PCPOOL.MI.FU-BERLIN.DE	01/05/09
#39	TopDown iterator has copy constructor but no assignment operator	Indices	closed	fixed	1.1	enhancement	minor	weese@PCPOOL.MI.FU-BERLIN.DE	01/05/09
#50	Problem with return type of representative function	Indices	closed	fixed	1.1	defect	major	weese@PCPOOL.MI.FU-BERLIN.DE	01/05/09
#55	operator== ill-defined for char*	Core Library	closed	fixed	1.1	defect	minor	doering@PCPOOL.MI.FU-BERLIN.DE	01/05/09
#38	example code in file documentation does not work	Core Library	closed	fixed	1.1	defect	major	doering@PCPOOL.MI.FU-BERLIN.DE	12/09/08
#33	Memory leaks in sequence test	Core Library	closed	fixed	1.1	defect	major	doering@PCPOOL.MI.FU-BERLIN.DE	12/09/08
#37	Genbank parser aborts, if an empty line is present in the flatfile.	Core Library	closed	workforme	1.1	defect	minor	doering@PCPOOL.MI.FU-BERLIN.DE	12/09/08
#22	bug in file_format_fasta.h	Core Library	closed	fixed	1.0	defect	trivial	doering@PCPOOL.MI.FU-BERLIN.DE	12/09/08
#28	Iterator<Map> misses 'operator ->'	Core Library	closed	fixed	1.1	defect	trivial	doering@PCPOOL.MI.FU-BERLIN.DE	12/09/08
#53	Anonymous checkout for 'misc/build_forwards.py'	Core Library	closed	won'tfix	1.1	defect	blocker	doering@PCPOOL.MI.FU-BERLIN.DE	11/27/08

Figure 40: **SeqAn Trac.** The screen shot shows a list of messages and issues posted by SeqAn users from around the world.

- (2) **Function Coverage Testing:** The extensively use of C++ templates in SeqAn raises a special testing problem: Usually C++ compilers perform only shallow syntax checks during the parsing of template code, so it is possible that a test program compiles correctly even if some templates contain syntax errors, just because these templates are never instantiated. We therefore apply a *white-box* testing method that ensures each template function to be instantiated at least once in the test. This is done by inserting the preprocessor macro `SEQAN_CHECKPOINT` at the beginning of each template function of the library, and maybe also in some further parts of the program for which we want to check that they are reached by the test. If testing is activated, this macro expands to a short piece of code that protocols the current source file and line of code. At the end of the test, the source files are scanned for `SEQAN_CHECKPOINT` and all occurrences that were never reached are reported. If testing is not activated, than the macro is defined to be empty, so `SEQAN_CHECKPOINT` has in this case no impact to the program's efficiency.

Since even the best testing cannot guarantee the correctness of a program, we used the open source error tracking system *Trac*<sup>1</sup>, so the library's users can report their bugs and give suggestions for improvement, see Figure 40.

<sup>1</sup>See [trac.edgewall.org](http://trac.edgewall.org)

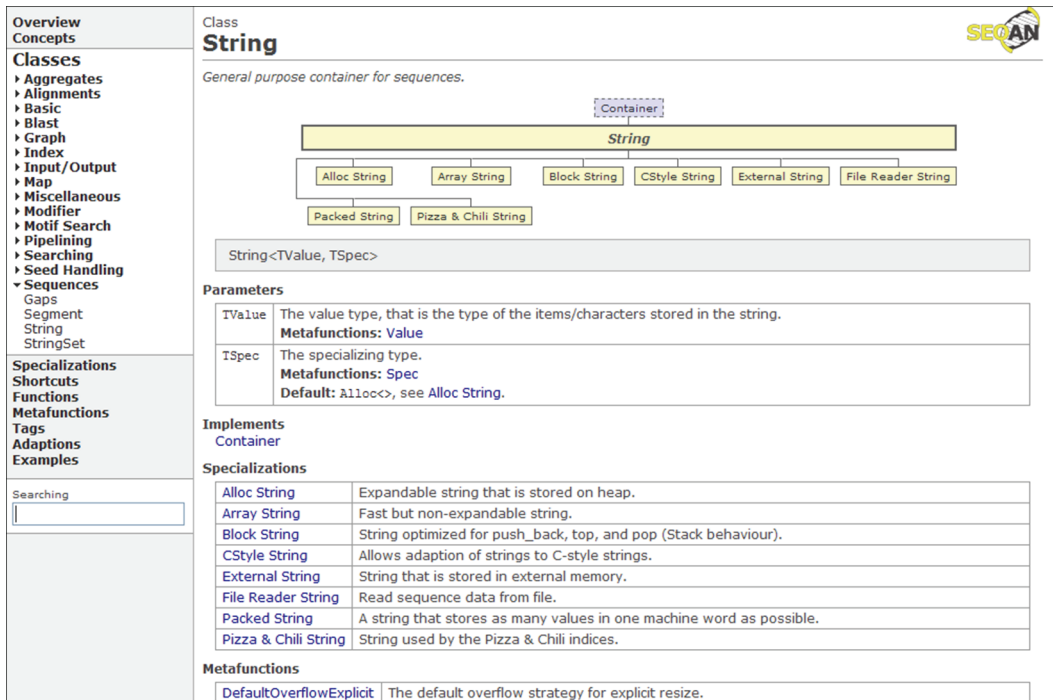


Figure 41: SeqAn Documentation Using DDDOC. The screen shot shows a part of the documentation for the class `String`.

## 14.2.2 Documentation

The common documentation systems for C++ like *Doxygen*<sup>2</sup> are designed with regard to object-oriented programming, so we developed our own documentation system *DotDotDoc* (DDDOC), which is especially suited for documenting generic programming software. The documentation is deposited in C++ comments that are extracted from the library's source files using a Python (Lutz 2006) script. The format orientates on the XML documentation format<sup>3</sup> that is used for Microsoft C#, but it uses a simple human readable notation style instead of XML. DDDOC creates a heavily cross-linked and searchable documentation (See Figure 41), that extensively describes all public classes, specializations, functions and metafunctions available in SeqAn on HTML pages, which can be viewed in common HTML browsers. The SeqAn documentation also contains several tutorials and example programs. It can be downloaded from the SeqAn web site and viewed online on [www.seqan.de/dddoc](http://www.seqan.de/dddoc).

<sup>2</sup>See [www.doxygen.org](http://www.doxygen.org)

<sup>3</sup>see [msdn.microsoft.com](http://msdn.microsoft.com)

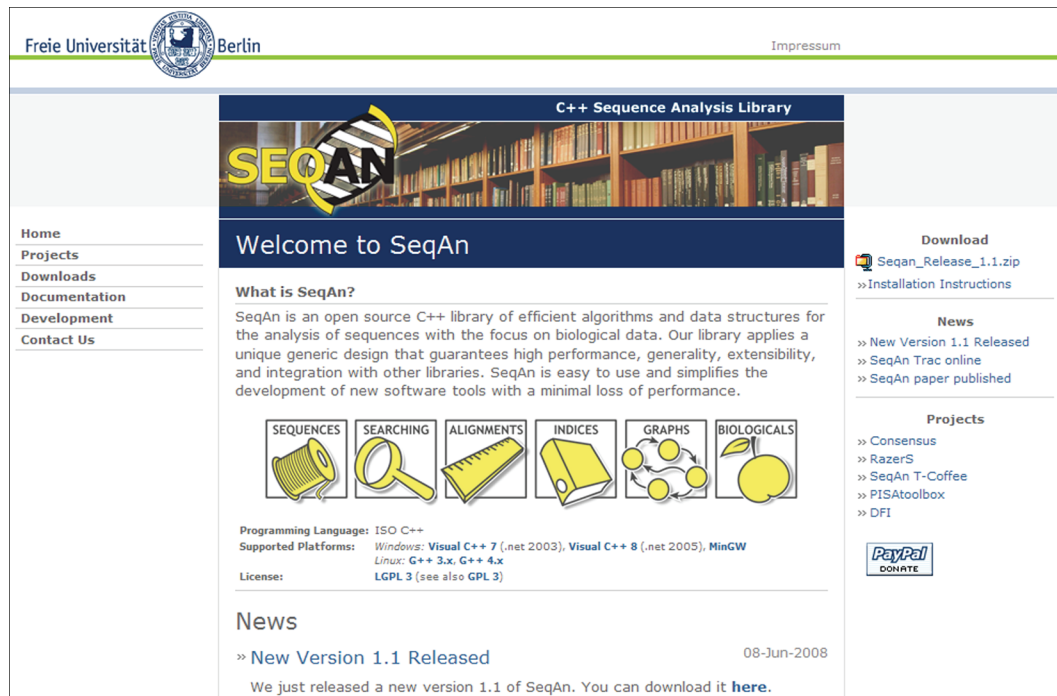


Figure 42: SeqAn Website.

### 14.2.3 Distribution

During the development process we took care to keep SeqAn compatible to multiple platforms. For that reason, we implemented a simple but powerful built-system that allows the compilation of applications and test programs using different compilers and operating systems. Our library now works on all common platforms, namely Microsoft Windows, Mac OS X, Solaris, and several Linux clones, and it was tested for Microsoft Visual C++ compilers (version 7 or above) and GCC compilers (version 3 or above).

SeqAn is open source and free software published under the ‘*Gnu Lesser General Public License*’ (LGPL) version 3.<sup>4</sup> This license allows the free use and distribution of the library also for commercial application. Both the library sources and the documentation can be viewed and downloaded from the SeqAn web site [www.seqan.de](http://www.seqan.de), which was designed to be the central place for all news and information about the project, see Figure 42. Beside detailed descriptions of SeqAn and its associated projects, this web page also contains a bug tracker system that can be used to return feedback to the library’s developers, see also Section 14.2.1.

<sup>4</sup>See [www.gnu.org/licenses/lgpl.txt](http://www.gnu.org/licenses/lgpl.txt)





# Chapter 15

## Example Application LAGAN

In this Chapter, we use SeqAn to re-implement the basic functionality of the common software tool LAGAN by Brudno et al. (2003).

### 15.1 The LAGAN Algorithm

LAGAN is a tool for aligning two long sequences  $a_1 \dots a_n$  and  $b_1 \dots b_m$ , and it uses a seed chaining approach, see Section 9.6. The applied procedure (see Algorithm 37 for line numbers) works in four steps, see Figure 43:

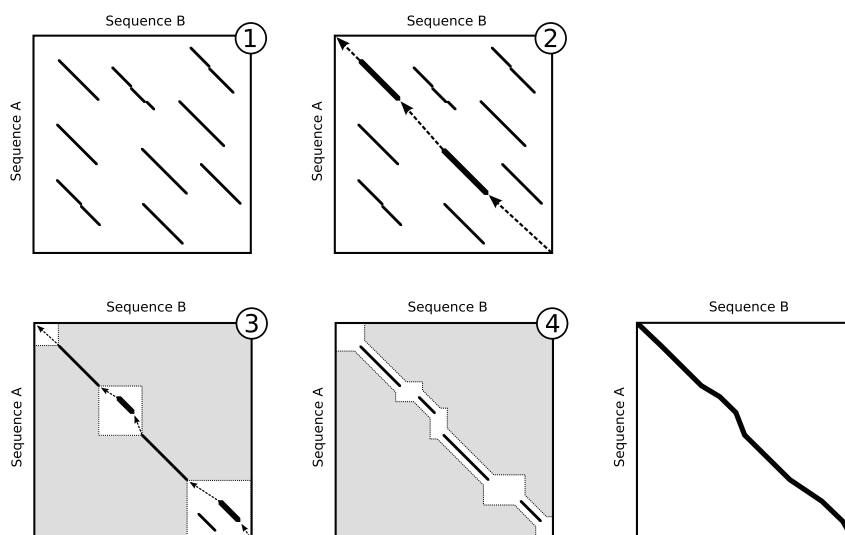


Figure 43: **The Four Steps of LAGAN.** (1) Finding seeds, (2) chaining, (3) recursively filling up gaps, and (4) banded alignment following the best chain. The result is a global alignment between the sequences.

- (1) **Finding Seeds** (lines 2 to 6): For a given length  $q = q_{\max}$ , all common  $q$ -grams of  $a_1 \dots a_n$  and  $b_1 \dots b_m$  are found, e.g. by using a  $q$ -gram index

(Section 12.1), and then combined to a set  $D$  of seeds by *local chaining* (Algorithm 24 on page 134), where the seed extension mode **Chaos** is used (Table 18 on page 135). If no common  $q$ -grams are found, the  $q$  is decreased until a minimal bound  $q_{\min}$  is reached.

1	$\triangleright \text{LAGAN}(a_1 \dots a_n, b_1 \dots b_m)$	steps 1–3
2	$\mathcal{C} \leftarrow \text{LAGANCHAINING}(a, b, q_{\max})$	step 4
3	$\mathcal{A} \leftarrow \text{BANDEDALIGNMENT}(\mathcal{C})$	
3	<b>return</b> $\mathcal{A}$	

1	$\triangleright \text{LAGANCHAINING}(a_1 \dots a_n, b_1 \dots b_m, q)$	
2	<b>if</b> $(n < \text{gapsmax})$ and $(m < \text{gapsmax})$ <b>then return</b> $\langle \rangle$	
2	$Q \leftarrow \emptyset$	
3	<b>while</b> $Q = \emptyset$ and $q < q_{\min}$ <b>do</b>	} step 1
4	$Q \leftarrow$ all common $q$ -grams between $a$ and $b$	
5	$q \leftarrow q - 2$	
6	$D \leftarrow \text{LOCALCHAINING}(Q)$	} step 2
7	$\langle \mathcal{S}_0, \dots, \mathcal{S}_{k-1} \rangle \leftarrow \text{SPARSECHAINING}(D)$	
8	<b>for</b> $i \leftarrow 0$ <b>to</b> $k + 1$ <b>do</b>	} step 3
9	$a' \leftarrow a_{\text{right}_0(\mathcal{S}_i)} \dots a_{\text{left}_0(\mathcal{S}_{i+1})}$	
10	$b' \leftarrow b_{\text{right}_1(\mathcal{S}_i)} \dots b_{\text{left}_1(\mathcal{S}_{i+1})}$	
11	$\mathcal{C}_i \leftarrow \text{LAGANCHAINING}(a', b', q)$	
12	$\mathcal{C} \leftarrow \{\mathcal{S}_1, \dots, \mathcal{S}_k\} \cup \bigcup_{i=0}^{k+1} \mathcal{C}_i$	
13	<b>return</b> sorted $\mathcal{C}$	

Algorithm 37: **The Algorithm of LAGAN**. The size  $q$  starts with  $q_{\max}$  and may go down to  $q_{\min}$ . **LAGANCHAINING** is only used if the lengths of both sequences  $a$  and  $b$  are at least  $\text{gapsmax}$ . For **LOCALCHAINING** see Algorithm 24 on page 134, and for **SPARSECHAINING** see Algorithm 7 on page 99. We omit the details of **BANDEDALIGNMENT**, see Section 9.6.4.

- (2) **Global Chaining** (line 7): A *chaining* algorithm like **SPARSECHAINING** (Algorithm 7 on page 99) computes the optimal *global chain*  $\langle \mathcal{S}_0, \dots, \mathcal{S}_{k-1} \rangle$ , where  $\mathcal{S}_0$  is the *top seed* and  $\mathcal{S}_{k+1}$  the *bottom seed* (see Section 9.6.2), and the rest  $\mathcal{S}_1, \dots, \mathcal{S}_k \in D$ .
- (3) **Filling Up Gaps** (lines 8 to 12): We fill up the gaps between any two successive seeds  $\mathcal{S}_i$  and  $\mathcal{S}_{i+1}$  for  $i \in \{0, \dots, k\}$  by applying step (1) to (3) recursively on the gaps for a smaller  $q$ . This recursion stops if either the length of the gap is in both dimensions smaller than  $\text{gapsmax}$ , or  $q$  falls below  $q_{\min}$  (line 1).

- (4) **Banded Alignment** (line 2): Following the chain  $\mathcal{C}$  that was computed in the steps (1) to (3), a *banded alignment* algorithm (see Section 9.6.4) is used to compute a global alignment  $\mathcal{A}$  between  $a$  and  $b$ .

## 15.2 Implementation of LAGAN

Before we start to implement Algorithm 37 in C++, we have to choose the data structures we want to use. Our objective is to align two DNA sequences  $a$  and  $b$ , so we use `String<Dna>` for storing them. The seeds  $\mathcal{S}_i$  are 2-dimensional, so we apply the specialization `SimpleSeed` of `Seed` (see Section 9.6.1). These seeds are locally aligned, so the most appropriate data structure for  $D$  is `SeedSet`. We apply the ‘scored’ variant, since this supports the functionality that is used in the original tool. For storing the chains  $\langle \mathcal{S}_0, \dots, \mathcal{S}_{k-1} \rangle$  and  $\mathcal{C}_i$ , we need a container class that supports fast insertion operation for merging several chains in line 12 of Algorithm 37, so a list type would be a good choice. We decide to use the class `std::list` from the standard library.

The complete source code of our program is printed in Appendix A.3.<sup>1</sup> It consists of two functions: The `main` function that implements LAGAN of Algorithm 37 and `laganChaining` that implements LAGANCHAINING.

We start the `main` function by loading the two input sequences  $a$  and  $b$  from FASTA files. For that purpose, we use `FileReader` strings (Section 7.7) that are copied to ‘in-memory’ strings of type `String<Dna>` for speeding up the further processing:

```
typedef String<Dna> TString;
TString a = String<Dna, FileReader<Fasta>>(argv[1]);
TString b = String<Dna, FileReader<Fasta>>(argv[2]);
```

Then we call the function `laganChaining`, which is described below, to perform the steps (1) to (3) of Algorithm 37:

```
typedef Seed<int, SimpleSeed> TSeed;
std::list<TSeed> chain;

laganChaining(chain,
              infix(a, 0, length(a)),
              infix(b, 0, length(b)), q_max);
```

The first argument is a list in which `laganChaining` will return a chain of seeds. Since in step (3) of the algorithm, the function will be called repeatedly on varying substrings of  $a$  and  $b$ , it expects the input sequences to be passed

<sup>1</sup>This program bases on Carsten Kemena’s master thesis (Kemena 2008).

as *segment* objects (Section 8.6). The `main` function conveys the complete sequences.

The last argument is the length of the q-grams `laganChaining` will start with. The initial call of `laganChaining` sets the size of the q-grams to 13, and this `q` may fall down to `q_min=7` during the execution.

### Step (1)

In `laganChaining`, we need three data structures to perform step (1) of Algorithm 37: A seed set for storing and merging the seeds, a q-gram index for the input sequence `b`, and a finder for searching the q-gram index:

```
typedef typename Value<TSeed>::Type TPosition;
typedef SeedSet<TPosition, SimpleSeed, DefaultScore> TSeedSet;
TSeedSet seedset(limit, score_min, scoring_scheme);

typedef Index< TSegment, Index_QGram<SimpleShape > > TQGramIndex;
TQGramIndex index_qgram(b);

typedef Finder<TQGramIndex> TFinder;
TFinder finder(index_qgram);
```

The constants `limit` and `score_min` define the area in which local chaining searches for predecessor seeds (Section 11.2.2). The local chaining also needs `scoring_scheme` to compute scores of seeds.

As long as no seeds are found, and `q` is at least `q_min`, we search for common q-grams in `a` and `b` and add them to `seedset`:

```
while (length(seedset) == 0)
{
    if (q < q_min) return;

    resize(indexShape(index_qgram), q);

    for (int i = 0; i < length(a)-q+1; ++i)
    {
        while (find(finder, infix(a, i, i+q)))
        {
            // add q-gram to seedset
            ...
        }
        clear(finder);
    }

    q-=2;
}
```

The variable `i` iterates through all `q`-gram positions in `a`, and the finder then enumerates all occurrences of the `i`-th `q`-gram of `a` in the indexed sequence `b`. In the inner `while` loop, we compute the starting positions of the common `q`-grams relative to the complete sequences, and then add the `q`-gram to `seedset`:

```
// add q-gram to seedset
typedef typename Position<TFinder>::Type TPosition;
TPosition a_pos = beginPosition(a)+i;
TPosition b_pos = beginPosition(b)+position(finder);

if (!addSeed(seedset, a_pos, b_pos, q, 0, Merge()))
if (!addSeed(seedset, a_pos, b_pos, q,
             host(a), host(b), bandwidth, Chaos()))
    addSeed(seedset, a_pos, b_pos, q, Single());
```

So we first try to merge the new `q`-gram  $\mathcal{S}$  with another overlapping `q`-gram on the same diagonal. If no such `q`-gram is available in `seedset`, then we try to find instead a predecessor  $\mathcal{S}'$  within in the area defined by `limit` and `score_min`. If a suitable  $\mathcal{S}'$  is found, then we merge  $\mathcal{S}'$  and  $\mathcal{S}$  to a single seed in the ‘chaos style’, i.e. with a single gap in between (Table 18 on page 135). Otherwise we just add  $\mathcal{S}$  to `seedset`.

### Step (2)

Step (2) of Algorithm 37 just takes a single line of code:

```
globalChaining(seedset, chain);
```

The function `globalChaining` uses sparse dynamic programming to compute the optimal chain of seeds without a penalty for the gaps between the seeds (Section 9.6.3). Note that the resulting chain that is stored in `chain` does *not* contain a *top seed* or a *bottom seed*, but only the ‘inner seeds’ from the chain.

### Step (3)

Note that `q` was decremented at least once during step (1). If `q` is still  $\geq q_{\min}$ , then we enumerate all gaps between two succeeding seeds in `chain` and try to fill them up by recursive calls of `laganChaining`. After each call, all new seeds that are returned in `subchain` are inserted into `chain` between the seeds `*it` and seed `*it2`:

```

list<TSeed> subchain;
typedef typename list<TSeed>::iterator TIterator;

TIterator it = chain.begin();
TIterator it2 = it;
++it2;

while (it2 != chain.end())
{
    laganChaining(subchain,
        infix(host(a), rightDim0(*it), leftDim0(*it2)),
        infix(host(b), rightDim1(*it), leftDim1(*it2)), q);
    chain.splice(it2, subchain);

    it = it2;
    ++it2;
}

```

Note that we have to do the same for the gaps before the first seed and behind the last seed in `chain`.

#### Step (4)

Back in the main function, it remains last step (4) of Algorithm 37: We add  $a$  and  $b$  as rows to an `Align` object (Section 9.2) and call `bandedChainAlignment` (see Section 9.6.4):

```

Align<TString, ArrayGaps> alignment;
resize(rows(alignment), 2);
setSource(row(alignment, 0), a);
setSource(row(alignment, 1), b);
int score = bandedChainAlignment(chain, B, alignment, scoring_scheme);

```

The constant  $B$  is the used band width, and `scoring_scheme` defines the applied scoring scheme.

At the end, we print out the resulting alignment and its score:

```

cout << "Score: " << score << endl;
cout << alignment << endl;

```

## 15.3 Results

The original tool was published in (2003) by Brudno, Do, Cooper, Kim, Davydov, Program, Green, Sidow, and Batzoglou, and it was implemented in a

combination of C programs that were stitched together by several Perl scripts. Its source code is much more extensive than our program, which takes about one hundred lines of code, see Appendix A.3; for example the source code of the tool ‘chaos’ that is responsible for step (1) and (2) of the algorithm is more than twenty fold larger than our program. Although the original tool is certainly more elaborated and therefore more complex than ours, both programs compute alignments of similar quality, and Figure 44 shows that the running times are also comparable.

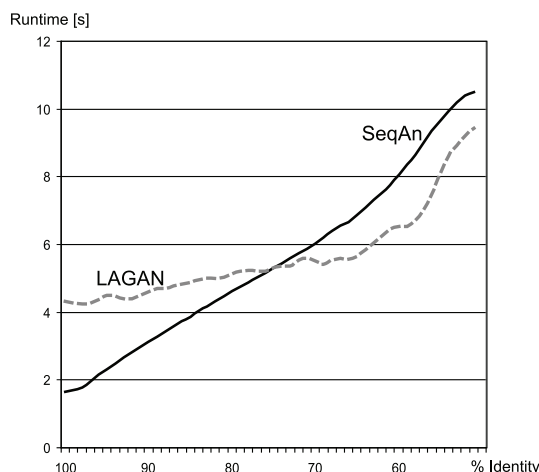


Figure 44: **Runtimes of LAGAN and SeqAn.** We aligned a 100kbp part of the genome of Escherichia Coli with a point mutated counterpart. The figure shows the average runtimes (in seconds) of the original LAGAN tool and the SeqAn program from Appendix A.3 depending on the similarity between the two sequences.

This example shows that programs which were developed with SeqAn can match up with ‘hand written’ tools. Moreover we demonstrated the components provided SeqAn are indeed useful for tool design, and that the application of SeqAn leads to concise and comprehensible solutions.





# Chapter 16

## Conclusion

In this thesis we presented the software library SeqAn that provides various data structures and algorithms for sequence analysis. We described the basic design principles used in SeqAn, and we explained how our library supports the development of new software tools. The SeqAn Project was started in 2003. A first version was published in 2007 (Döring, Weese, Rausch, and Reinert 2008). Presently, four PhD projects are in progress, which concentrate on the development of SeqAn and contribute significantly to its functionality. Some parts of the library were created during the course of master theses by Weese (2006), Wöhrle (2006), Lim (2007), Emde (2007), and Kemena (2008). Many other students contributed smaller parts to the library in their software projects and bachelor theses. While this thesis was in progress, SeqAn has already become an indispensable work bench for ourselves which can be seen by the number of research papers that used SeqAn. The library now covers the basic areas of sequence analysis; this was shown at length in Part III. Nevertheless, considering all the research on bioinformatics done in the last few decades, it is obvious that SeqAn is still far away from being ‘complete’. The library will stay under active development, and its range of functions will be extended, for example by statistical data structures like hidden markov models and statistical context free grammars (see e.g. Durbin et al. 1999). The library will further advanced to support multi threading and massive parallel hardware like graphic cards or multi core processor achitectures. Applying our library to current problems in life science will certainly be increasingly important in our future work. We hope that more and more scientists will use and contribute to the library in the future.



# A

## Appendix

### A.1 Proof to Myers' Bitvector Algorithm

In this section, we will explain Myers' bitvector algorithm for approximate string matching, see Algorithm 17 in Section 10.3.2. This algorithm finds all occurrences of a needle  $p$  in a haystack  $t$  with edit distance below a given threshold. Remember that the edit distance between two sequences is the negative score of their optimal alignment where each match scores 0 and each mismatch and gap scores  $-1$  (see Section 9.3.1).

Let us call in the following  $x := M_{i-1,j-1}$ ,  $v := M_{i-1,j}$ ,  $h := M_{i,j-1}$ , and  $d := M_{i-1,j-1}$ . Using that notation, we can rewrite the main recursion of the Sellers' dynamic programming algorithm (Equation 9.6, page 87) for edit distance as follows:

$$x = \max\{d - (p_i \neq t_j), v - 1, h - 1\} \quad (\text{A.1})$$

A simple induction shows that:

$$\begin{aligned} M_{i,j} - M_{i-1,j} &\in \{-1, 0, 1\} \\ M_{i,j} - M_{i,j-1} &\in \{-1, 0, 1\} \\ M_{i,j} - M_{i-1,j-1} &\in \{-1, 0\} \end{aligned} \quad (\text{A.2})$$

For each  $j \in \{1, \dots, n\}$ , we define five vectors  $VP^j$ ,  $VN^j$ ,  $HP^j$ ,  $HN^j$ , and  $D0^j$  of booleans, each of length  $m$ , as follows:

$$\begin{aligned} VP_i^j &:= (x = v - 1) & VN_i^j &:= (x = v + 1) \\ HP_i^j &:= (x = h - 1) & HN_i^j &:= (x = h + 1) \\ D0_i^j &:= (x = d) \end{aligned}$$

We will now explain how Algorithm 17 computes these five vectors for  $j$  based on of the vectors for  $j - 1$ :

- From (A.2) follows  $x \leq d \leq v + 1$ . If  $VN_i^j$  (that is  $v + 1 = x$ ), then this becomes an equality, so that in this case  $D\theta_i^j$  (since  $x = d$ ) and  $HP_{i-1}^j$  (since  $v = d - 1$ ). The reverse also holds, so that we can state the equivalence:

$$VN_i^j = D\theta_i^j \wedge HP_{i-1}^j$$

The same way we can show:

$$HN_i^j = D\theta_i^j \wedge VP_i^{j-1}$$

- From (A.2) follows  $v - 1 \leq x \leq d$ . If  $HN_{i-1}^j$  (that is  $d = v - 1$ ), then this becomes an equality, so that in this case  $VP_i^j$  (since  $x = v - 1$ ). Suppose now that neither  $D\theta_i^j$  nor  $HP_{i-1}^j$ , hence  $x + 1 = d \leq v$ ; then this becomes an equality due to  $v \leq x + 1$  from (A.2), and it follows again  $VP_i^j$  (since  $v = x + 1$ ). So we can state: if  $HN_{i-1}^j \vee (\neg D\theta_i^j \wedge \neg HP_{i-1}^j)$  then  $VP_i^j$ .

The reverse also holds: Suppose that  $VP_i^j$  and not  $HN_{i-1}^j$ , hence  $x + 1 = v \leq d$ . Then this becomes an equality due to  $d \leq x + 1$  from (A.2), so it follows  $\neg D\theta_i^j$  (since  $x + 1 = d$ ) and  $\neg HP_{i-1}^j$  (since  $v = d$ ). Hence we proved the following equivalence:

$$VP_i^j = HN_{i-1}^j \vee \neg(D\theta_i^j \vee HP_{i-1}^j)$$

The same way we can show:

$$HP_i^j = VN_i^{j-1} \vee \neg(D\theta_i^j \vee VP_i^{j-1})$$

- From (A.2) follows  $v - 1 \leq x \leq d$ . If  $HN_{i-1}^j$  (that is  $d = v - 1$ ), then this becomes an equality, hence  $D\theta_i^j$  (since  $x = d$ ). The same way we can prove that  $VN_i^{j-1}$  implies  $D\theta_i^j$ . A third reason for  $D\theta_i^j$  being true is a match between  $p_i$  and  $t_j$ : From (A.1) follows  $x \geq d - (p_i \neq t_j)$ . If  $p_i = t_j$ , then  $x \geq d$ , and since  $x \leq d$  due to A.2 it follows  $D\theta_i^j$ . So we can state: if  $(p_i = t_j) \vee VN_i^{j-1} \vee HN_{i-1}^j$  then  $D\theta_i^j$ .

The reverse also holds: Suppose that  $D\theta_i^j$  and  $\neg HN_{i-1}^j$  and  $\neg VN_i^{j-1}$ . From  $\neg HN_{i-1}^j$  (that is  $v \leq d$ ) follows  $v - 1 \leq d - 1 \leq d - (p_i \neq t_j)$ , and from  $\neg VN_i^{j-1}$  (that is  $h \leq d$ ) follows  $g - 1 \leq d - 1 \leq d - (p_i \neq t_j)$ . Hence from (A.1) follows:  $x = d - (p_i \neq t_j)$ , and since  $D\theta_i^j$  (that is  $d = x$ ) it follows  $p_i = t_j$ . This proved the equivalence:

$$D\theta_i^j = (p_i = t_j) \vee VN_i^{j-1} \vee HN_{i-1}^j$$

There is a cyclic dependency in the equivalences above:<sup>1</sup>  $D\theta_i$  depends on  $HN_{i-1}$ , which again depends from  $D\theta_{i-1}$ , so we get:

$$D\theta_i = X_i \vee (VP_{i-1} \wedge D\theta_{i-1}) \tag{A.3}$$

---

<sup>1</sup>From now on, we will leave away the  $j$  superscript for a better readability.

where  $X_i := (p_i = t_j) \vee VN_i$ . Even so we can compute  $D0$  using some bit-parallel operations, including the addition

$$S := VP + (VP \wedge X).$$

Let  $C_i$  be 'carry-in' bit during the addition of  $VP_i$  and  $(VP_i \wedge X_i)$ , that is:

$$C_i = \begin{cases} 1, & \text{if } i > 0 \text{ and } (VP_{i-1} + (VP_{i-1} \wedge X_{i-1}) + C_{i-1}) > 1 \\ 0, & \text{otherwise} \end{cases} \quad (\text{A.4})$$

$$S_i = VP_i \oplus (VP_i \wedge X_i) \oplus C_i \quad (\text{A.5})$$

We will show by induction that

$$D0_i = X_i \vee C_i \quad (\text{A.6})$$

For  $i = 0$  this is obviously the case. Now assume that (A.6) holds for  $i-1 \geq 0$ , that is  $D0_{i-1} = X_{i-1} \vee C_{i-1}$ . From (A.4) follows: If  $VP_{i-1}$ , then  $C_i = X_{i-1} \vee C_{i-1} = D0_{i-1}$ . Otherwise, if  $\neg VP_{i-1}$ , then  $C_i = 0$ . So we get:

$$C_i = VP_{i-1} \wedge D0_{i-1}.$$

Together with (A.3) follows (A.6) for  $i$ .

We need only four bit-vector operations to compute  $D0$  given  $X$  and  $VP$ : From (A.5) follows  $C = S \oplus VP \oplus (VP \wedge X)$ , that is  $D0 = X \vee (S \oplus VP \oplus (VP \wedge X))$ , which can be simplified to  $D0 = X \vee (S \oplus VP)$ , hence:

$$D0 = ((VP + (VP \wedge X)) \oplus VP) \vee X$$

The complete recursion is shown in Algorithm 17 on page 119.

## A.2 Sum Lists

In Section 9.1.3, we used a data type called ‘*sum list*’ that was capable to store *gap patterns* efficiently. We will now describe this data type in more details, and we will explain how it was implemented in SeqAn. Let  $\mathcal{L} = \langle \tau^1, \dots, \tau^n \rangle$  be an ordered set of  $d$ -dimensional tuples, where each  $\tau^k$  contains  $d$  numbers  $\tau^k[0], \dots, \tau^k[d-1]$ . The abstract data type that stores  $\mathcal{L}$  is a  $d$ -dimensional *sum list*, if it supports at least the following operations:

- (1) **insert**( $\tau^*, k$ ): Inserts a new tuple  $\tau^*$  at a specific position  $k$ , such that afterwards  $\mathcal{L} = \langle \tau^1, \dots, \tau^k, \tau^*, \tau^{k+1}, \dots, \tau^n \rangle$
- (2) **delete**( $k$ ): Removes a tuple  $\tau^k$  from the list, such that afterwards  $\mathcal{L} = \langle \tau^1, \dots, \tau^{k-1}, \tau^{k+1}, \dots, \tau^n \rangle$
- (3) **change**( $k, i, x$ ): Sets the  $i$ -th value of the  $k$ -th tuple to  $x$ , i.e.  $\tau_i^k \leftarrow x$ .
- (4) **search**( $S, i$ ): Finds the minimal  $k$  such that  $\sigma_i^k \geq S$ , where  $\sigma^k = \sum_{j=1}^k \tau^j$  is the sum of the first  $k$  tuples.

We implemented the sum list in SeqAn such that each operation takes time  $O(d \log n)$ . The data structure that we used for that purpose resembles a *skip list* (Pugh 1990):<sup>2</sup> We store the tuples  $\tau^1, \dots, \tau^n$  in a linked list, where each list element has one pointer to the next list element (we call it the level-0 pointer), and maybe some additional ‘skip pointers’. The number of skip pointers  $h_j$  for the  $j$ -th list element is randomly chosen and geometrical distributed, i.e. the chance for a list element of having  $\geq h$  skip pointers is  $1/2^h$ . The only exception is the first list element which always gets the maximum number  $h_1 = \max_{j=2}^n h_j$  of skip pointers. It is easy to show that the expected value of  $h_1$  is  $O(\log n)$ . For each level  $t \in \{0, \dots, h_1\}$ , all list elements that have  $\geq t$  pointers are chained to a list that is linked by their level- $t$  pointers, see Figure 45. We write at each skip pointer the partial sum of the tuples that it skips, i.e. a skip pointer from  $\tau^k$  to  $\tau^l$  is marked by  $\sum_{j=k}^{l-1} \tau^j$ . If the  $k$ -th list element is the last in the list of a given level  $t$ , then the level- $t$  pointer directs to nil and is marked by  $\sum_{j=k}^n \tau^j$ .

For the  $j$ -th list element, we define  $trace_k$  to be the set of the pointers that *points to* the  $k$ -th list element conjoint with all pointers that *skip* the  $k$ -th list element. Obviously  $|trace_k| = h_1 + 1$ . We can easily compute  $trace_k$  for a given  $k$  by starting a search from the first list element. We begin with level  $t = h_1$ , and follow the level- $t$  pointers as long as possible without skipping the  $k$ -th list element. Then we proceed the search with pointers of level  $t - 1$ , and this

---

<sup>2</sup>Some proofs about the expected height  $h_1$  and the expected runtimes for searching the sum list are identical to corresponding proofs shown by Pugh (1990) for skip lists. Therefore we omit these proofs here.

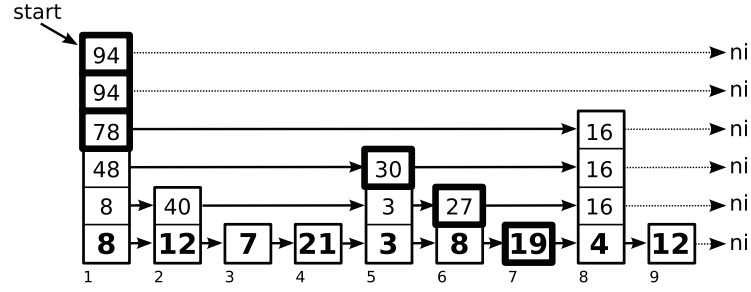


Figure 45: **Sum List**. This example shows a 1-dimensional sum list of  $n = 9$  values. ‘Start’ directs to the pointer at which the search begins. The fat values are the 1-tuples  $\tau_1, \dots, \tau_9$ , and the fat cells correspond to the pointer set *traces*.

is repeated with decreasing  $t$  until we actually find the  $k$ -th list element. One can show that one search takes an average time of  $O(\log n)$ .

The operations **insert**, **delete** or **change** applied to the  $k$ -th list element affect only the pointers in  $trace_k$ , so each operation takes time  $O(d h_1) = O(d \log n)$  for updating these pointers and recalculating their partial sums. **insert** moreover takes  $O(d)$  to create a new list element and update its expected number of 2 pointers and partial sums.

```

1  ▷ SEARCHSUMLIST ( $\mathcal{L} = \langle \tau_1, \dots, \tau_n \rangle, S, i$ )
2   $t \leftarrow h_1$  (the height of  $\mathcal{L}$ )
3   $k \leftarrow 1$ 
4   $\sigma \leftarrow \tau_1$ 
5  while  $t \geq 0$  do
6     $\sigma^{\text{part}} \leftarrow$  partial sum stored at level- $t$  pointer of  $\tau_k$ 
7     $\tau_l \leftarrow$  tuple the level- $t$  pointer of  $\tau_k$  points to
8    if  $(\tau_l = \text{nil})$  or  $(\sigma_i + \sigma_i^{\text{part}} \geq S)$  then
9       $t \leftarrow t - 1$ 
10   else
11      $\sigma \leftarrow \sigma + \sigma^{\text{part}}$ 
12      $k \leftarrow l$ 
13 return  $\langle k, \tau_k, \sigma \rangle$ 

```

Algorithm 38: **Search Sum List**. This algorithm implements the operation **search**( $S, i$ ). It searches in the sum list  $\mathcal{L}$  for the minimal  $k$  with  $\sigma_i^k \geq S$ .

The implementation of **search**( $S, i$ ) requires a slightly different search, see Algorithm 38: When we follow the pointers of level  $t$ , we sum up the partial sums on the pointers to  $\sigma$  as long as  $\sigma_i < S$ . If  $\sigma_i$  would become  $\geq S$ , then we decrease the level by one and proceed with the pointers of level  $t - 1$ . Since this takes on average  $O(\log n)$  steps and each step takes  $O(d)$  time, the total time needed for **search** is  $O(d \log n)$ .

## A.3 LAGAN Sources

This is the complete source code of the LAGAN program that we described in Chapter 15.

```

1  #include <iostream>
2  #include <seqan/seqs.h>
3  #include <seqan/file.h>

4  using namespace seqan;

5  //define some constants
6  int const gaps_max = 1000; //minimal sequence length for chaining
7  int const q_max = 13;      //start value for q
8  int const q_min = 7;      //minimal q
9  int const limit = 20;     //local seed chaining limit
10 int const bandwidth = 5;  //local seed chaining bandwidth
11 int const score_min = 30; //minimal score for local seed chaining
12 SimpleScore const scoring_scheme(3, -2, -1, -3); //scoring scheme
13 int const B = 7;          //width for banded alignment

14 //function laganChaining
15 template <typename TSeed, typename TSegment, typename TSize>
16 void laganChaining(std::list<TSeed> & chain,
17                   TSegment const & a,
18                   TSegment const & b,
19                   TSize q)
20 {
21     if ((length(a) <= gaps_max) && (length(b) <= gaps_max)) return;

22     //Step 1: find seeds
23     typedef typename Value<TSeed>::Type TPosition;
24     typedef SeedSet<TPosition, SimpleSeed, DefaultScore> TSeedSet;
25     TSeedSet seedset(limit, score_min, scoring_scheme);

26     typedef Index< TSegment, Index_QGram<SimpleShape > > TQGramIndex;
27     TQGramIndex index_qgram(b);

28     typedef Finder<TQGramIndex> TFinder;
29     TFinder finder(index_qgram);

```



```

30     while (length(seedset) == 0)
31     {
32         if (q < q_min) return;

33         resize(indexShape(index_qgram), q);

34         for (int i = 0; i < length(a)-q+1; ++i)
35         {
36             while (find(finder, infix(a, i, i+q)))
37             {
38                 typedef typename Position<TFinder>::Type TPosition;
39                 TPosition a_pos = beginPosition(a)+i;
40                 TPosition b_pos = beginPosition(b)+position(finder);

41                 if (!addSeed(seedset, a_pos, b_pos, q, 0, Merge()))
42                     if (!addSeed(seedset, a_pos, b_pos, q, host(a), host(b),
43                                 bandwidth, Chaos()))
44                         addSeed(seedset, a_pos, b_pos, q, Single());
45             }
46             clear(finder);
47         }

48         q-=2;
49     }

50     //Step 2: global chaining
51     globalChaining(seedset, chain);
52     clear(seedset);

53     //Step 3: recursively fill gaps
54     if (q > q_min)
55     {
56         std::list<TSeed> subchain;
57         typedef typename std::list<TSeed>::iterator TIterator;

58         TIterator it = chain.begin();
59         TIterator it2 = it;
60         ++it2;

61         laganChaining(subchain,
62                       infix(host(a), beginPosition(a), leftDim0(*it)),
63                       infix(host(b), beginPosition(b), leftDim1(*it)), q);
64         chain.splice(it, subchain);

65         while(it2 != chain.end())
66         {

```

```

67         laganChaining(subchain,
68             infix(host(a), rightDim0(*it), leftDim0(*it2)),
69             infix(host(b), rightDim1(*it), leftDim1(*it2)), q);
70         chain.splice(it2, subchain);

71         it = it2;
72         ++it2;
73     }
74     laganChaining(subchain,
75         infix(host(a), rightDim0(*it), endPosition(a)),
76         infix(host(b), rightDim1(*it), endPosition(b)), q);
77     chain.splice(it2, subchain);
78 }
79 }

80 int main( int argc, const char* argv[] )
81 {
82     //load sequences
83     typedef String<Dna> TString;
84     TString a = String<Dna, FileReader<Fasta> >(argv[1]);
85     TString b = String<Dna, FileReader<Fasta> >(argv[2]);
86     if ((length(a) == 0) || (length(b) == 0))
87     {
88         std::cout << "Error - file problem" << std::endl;
89         return 1;
90     }

91     //LAGAN
92     typedef Seed<int, SimpleSeed> TSeed;
93     std::list<TSeed> chain;

94     //Step 1 to 3
95     laganChaining(chain,
96         infix(a, 0, length(a)),
97         infix(b, 0, length(b)), q_max);

98     //Step 4: banded alignment
99     Align<TString, ArrayGaps> alignment;
100     resize(rows(alignment), 2);
101     setSource(row(alignment, 0), a);
102     setSource(row(alignment, 1), b);
103     int score = bandedChainAlignment(chain, B, alignment, scoring_scheme);

104     //print results
105     std::cout << "Score: " << score << std::endl;
106     std::cout << alignment << std::endl;

107     return 0;
108 }

```

# Bibliography

- Abouelhoda, M. I., S. Kurtz, and E. Ohlebusch (2002). The enhanced suffix array and its applications to genome analysis. In *WABI '02: Proceedings of the Second International Workshop on Algorithms in Bioinformatics*, pp. 449–463. Springer-Verlag.
- Abouelhoda, M. I., S. Kurtz, and E. Ohlebusch (2004). Replacing suffix trees with enhanced suffix arrays. *J. of Discrete Algorithms* 2(1), 53–86.
- Abouelhoda, M. I. and E. Ohlebusch (2003). Multiple genome alignment: Chaining algorithms revisited. In *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, Volume 2676, pp. 1–16. Springer.
- Aho, A. V. and M. J. Corasick (1975). Efficient string matching: an aid to bibliographic search. *Communications of the ACM* 18(6), 333–340.
- Allauzen, C., M. Crochemore, and M. Raffinot (1999). Factor oracle: A new structure for pattern matching. In *SOFSEM '99: Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics on Theory and Practice of Informatics*, pp. 295–310. Springer-Verlag.
- Allauzen, C., M. Crochemore, and M. Raffinot (2001). Efficient experimental string matching by weak factor recognition. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, Volume 2089 of *Lecture Notes in Computer Science*, pp. 51–72. Springer.
- Altschul, S. F., W. Gish, W. Miller, E. W. Myers, and D. J. Lipman (1990). Basic local alignment search tool. *Journal of Molecular Biology* 215(3), 403–410.
- Arnold, K., J. Gosling, and D. Holmes (2005). *The Java Programming Language (4th Edition)*. Addison-Wesley Professional.
- Austern, M. H. (1998). *Generic Programming and the STL*. Addison Wesley.
- Bailey, T. L. and C. Elkan (1994, August). Fitting a mixture model by expectation maximization to discover motifs in biopolymers. In *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology*, pp. 28–36. AAAI Press.

- Bailey, T. L. and C. Elkan (1995). The value of prior knowledge in discovering motifs with meme. In *Proceedings of the International Conference on Intelligent Systems for Molecular Biology*, Volume 3, pp. 21–29. AAAI Press.
- Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.
- Benson, D. A., I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and D. L. Wheeler (2008). Genbank. *Nucleic Acids Research* 36, D25–30.
- Bentley, D. R. (2006). Whole-genome re-sequencing. *Current Opinion in Genetics and Development* 16(6), 545–552.
- Boyer, R. S. and J. S. Moore (1977, October). A fast string searching algorithm. *Communications of the ACM* 20(10), 762–772.
- Brudno, M., C. Do, G. M. Cooper, M. F. Kim, E. Davydov, N. C. S. Program, E. D. Green, A. Sidow, and S. Batzoglou (2003). Lagan and multigan: Efficient tools for large-scale multiple alignment of genomic DNA. *Genome Research* 13(4), 721–731.
- Buhler, J. and M. Tompa (2001). Finding motifs using random projections. In *RECOMB 2001: Proceedings of the fifth annual international conference on Computational biology*, pp. 69–76. ACM.
- Burrows, M. and D. J. Wheeler (1994). A block-sorting lossless data compression algorithm. Technical Report SRC-RR-124, Digital Systems Research Center.
- Butt, D., A. J. Roger, and C. Blouin (2005). libcov: A C++ bioinformatic library to manipulate protein structures, sequence alignments and phylogeny. *BMC Bioinformatics* 6(138).
- Chapman, B. and J. Chang (2000). Biopython: Python tools for computational biology. *SIGBIO Newsletter* 20(2), 15–19.
- Cormen, T. H., C. E. Leiserson, R. L. Rivest, and C. Stein (2001). *Introduction to Algorithms, second edition*. MIT Press.
- Crochemore, M., A. Czumaj, L. Gasieniec, S. Jarominek, W. P. T. Lecroq, and W. Rytter (1994, November). Speeding up two string-matching algorithms. *Algorithmica* 12(4–5), 247–267.
- Czarnecki, K. and U. W. Eisenecker (2000). *Generative Programming. Methods, Tools, and Applications*. Addison Wesley.
- Darling, A., B. Mau, F. Blattner, and N. Perna (2004). Mauve: Multiple Alignment of Conserved Genomic Sequence with Rearrangements. *Genome Research* 14, 1394–1403.
- Davila, J., S. Balla, and S. Rajasekaran (2006). Space and time efficient algorithms for planted motif search. In *IWBRA 2006: Second Interna-*

- tional Workshop on Bioinformatics Research and Applications*, Volume 3992 of *Lecture Notes in Computer Science*, pp. 822–829.
- Dayhoff, M. O., R. M. Schwartz, and B. C. Orcutt (1978). A model of evolutionary change in proteins. In M. O. Dayhoff (Ed.), *Atlas of Protein Sequence and Structure*, Volume 5(3), pp. 345–352.
- Dementiev, R., J. Kärkkäinen, J. Mehnert, and P. Sanders (2008). Better external memory suffix array construction. *Journal of Experimental Algorithmics* 12, 1–24.
- Dempster, A. P., N. M. Laird, and D. B. Rubin (1977). Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society* 39(1), 1–39.
- Dohm, J. C., C. Lottaz, T. Borodina, and H. Himmelbauer (2007). SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. *Genome Research* 17, 1697–1706.
- Döring, A., D. Weese, T. Rausch, and K. Reinert (2008). SeqAn an efficient, generic c++ library for sequence analysis. *BMC Bioinformatics* 9(11).
- Durbin, R., E. R. Sean, A. Krogh, and G. Mitchison (1999). *Biological Sequence Analysis : Probabilistic Models of Proteins and Nucleic Acids*. Cambridge: Cambridge University Press.
- Dutheil, J., S. Gaillard, E. Bazin, S. Glemin, V. Ranwez, N. Galtier, and K. Belkhir (2006). Bio++: a set of C++ libraries for sequence analysis, phylogenetics, molecular evolution and population genetics. *BMC Bioinformatics* 7(188).
- EMBL User Manual (2008). Release 96, European Bioinformatics Institute, <http://www.ebi.ac.uk/embl>.
- Emde, A.-K. (2007). Progressive alignment of multiple genomic sequences. Master’s thesis, Freie Universität Berlin.
- Eppstein, D., Z. Galil, R. Giancarlo, and G. F. Italiano (1992). Sparse dynamic programming i: linear cost functions; ii: convex and concave cost functions. *Journal of the ACM* 39(3), 519–567.
- Fabri, A., G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr (2000). On the design of CGAL a computational geometry algorithms library. *Software—Practice and Experience* 30(11), 1167–1202.
- Ferragina, P. and G. Navarro. Pizza & chili corpus, compressed indexes and their testbeds. available at <http://pizzachili.dcc.uchile.cl>, <http://pizzachili.di.unipi.it>.
- Gentleman, R. C., V. J. Carey, D. M. Bates, B. Bolstad, M. Dettling, S. Dudoit, B. Ellis, L. Gautier, Y. Ge, J. Gentry, K. Hornik, T. Hothorn, W. Huber, S. Iacus, R. Irizarry, F. Leisch, C. Li, M. Maechler, A. J.

- Rossini, G. Sawitzki, C. Smith, G. Smyth, L. Tierney, J. Y. H. Yang, and J. Zhang (2004). Bioconductor: open software development for computational biology and bioinformatics. *Genome Biology* 5(10).
- Giancarlo, R., A. Siragusa, E. Siragusa, and F. Utro (2007). A basic analysis toolkit for biological sequences. *Algorithms Molecular Biology* 2(10).
- Giegerich, R., S. Kurtz, and J. Stoye (1999). Efficient implementation of lazy suffix trees. In *WAE '99: Proceedings of the 3rd International Workshop on Algorithm Engineering*, pp. 30–42. Springer-Verlag.
- Goto, N., M. C. Nakao, S. Kawashima, T. Katayama, and M. Kanehisa (2003). BioRuby: Open-source bioinformatics library. *Genome Informatics* 14, 629–630.
- Gotoh, O. (1982, Dec). An improved algorithm for matching biological sequences. *J. Mol. Biol.* 162(3), 705–708.
- Griffith, A. (2002). *GCC: The Complete Reference*. McGraw-Hill, Inc.
- Gurtovoy, A. and D. Abrahams (2002). The Boost C++ metaprogramming library.
- Gusfield, D. (1997). *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press.
- Halpern, A. L., D. H. Huson, and K. Reinert (2002). Segment match refinement and applications. In *WABI '02: Proceedings of the Second International Workshop on Algorithms in Bioinformatics*, pp. 126–139. Springer-Verlag.
- Henikoff, S. and J. G. Henikoff (1992, November). Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences USA* 89(22), 10915–10919.
- Hirschberg, D. S. (1975, June). A linear space algorithm for computing maximal common subsequences. *ACM Press* 18(6), 341–343.
- Hohl, M., S. Kurtz, and E. Ohlebusch (2002). Efficient multiple genome alignment. *Bioinformatics* 33(18), 312–320.
- Holland, R., T. Down, M. Pocock, A. Prlic, D. Huen, K. James, S. Foisy, A. Dräger, A. Yates, M. Heuer, and M. Schreiber (2008). BioJava: an open-source framework for bioinformatics.
- Hopcroft, J. E. and J. D. Ullman (1990). *Introduction To Automata Theory, Languages, And Computation*. Addison-Wesley Longman Publishing Co., Inc.
- Horspool, R. N. (1980). Practical fast searching in strings. *Software-Practice and Experience* 10, 501–506.

- Huson, D. H., K. Reinert, S. A. Kravitz, K. A. Remington, A. L. Delcher, I. M. Dew, M. Flanigan, A. L. Halpern, Z. Lai, C. M. Mobarry, G. G. Sutton, and E. W. Myers (2001). Design of a compartmentalized shotgun assembler for the human genome. *Bioinformatics* 17, 132–139. Proceedings of ISMB 2001.
- Hydro, H. and H. H. Fi (2001). Explaining and extending the bit-parallel approximate string matching algorithm of Myers. Technical Report A-2001-10, Department of Computer and Information Sciences, University of Tampere.
- Ihaka, R. and R. Gentleman (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics* 5(3), 299–314.
- Indyk, P. and R. Motwani (1998). Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pp. 604–613. ACM.
- International Human Genome Sequencing Consortium (2001). Initial sequencing and analysis of the human genome. *Nature* 409(6822), 860–921.
- ISO/IEC (1998). *Programming languages – C++, International Standard 14882* (first ed.). American National Standards Institute.
- Istrail, S., G. G. Sutton, L. Florea, A. L. Halpern, C. M. Mobarry, R. Lippert, B. Walenz, H. Shatkay, I. Dew, J. R. Miller, M. J. Flanigan, N. J. Edwards, R. Bolanos, D. Fasulo, B. V. Halldorsson, S. Hannenhalli, R. Turner, S. Yooseph, F. L., D. R. Nusskern, B. C. Shue, X. H. Zheng, F. Zhong, A. L. Delcher, D. H. Huson, S. A. Kravitz, L. Mouchard, K. Reinert, K. A. Remington, A. G. Clark, M. S. Waterman, E. E. Eichler, M. D. Adams, M. W. Hunkapillar, E. W. Myers, and J. C. Venter (2004). Whole-genome shotgun assembly and comparison of human genome assemblies. *Proceedings of the national academy of science (PNAS)* 101(7), 1916–1921.
- Jacobson, G. and K.-P. Vo (1992). Heaviest increasing/common subsequence problems. In *CPM '92: Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching*, pp. 52–66. Springer-Verlag.
- Jones, D. T., W. R. Taylor, and J. M. Thornton (1992). The rapid generation of mutation data matrices from protein sequences. *Bioinformatics, Oxford University Press* 8(3), 275–282.
- Josuttis, N. M. (1999). *The C++ standard library: a tutorial and reference*. Addison-Wesley Longman Publishing Co., Inc.

- Kärkkäinen, J. and P. Sanders (2003). Simple linear work suffix array construction. In *Proceedings of the 30th International Conference on Automata, Languages and Programming*, Volume 2719 of *Lecture Notes in Computer Science*, pp. 943–955. Springer-Verlag.
- Kärkkäinen, J., P. Sanders, and S. Burkhardt (2006). Linear work suffix array construction. *Journal of the ACM* 53(6), 918–936.
- Karlin, S. and S. F. Altschul (1990). Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. In *Proceedings of the National Academy of Sciences USA*, Volume 87, pp. 2264–2268.
- Kasai, T., G. Lee, H. Arimura, S. Arikawa, and K. Park (2001). Linear-time longest-common-prefix computation in suffix arrays and its applications. In *CPM '01: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, pp. 181–192. Springer-Verlag.
- Kececioğlu, J. D. (1993). The maximum weight trace problem in multiple sequence alignment. In *CPM '93: Proceedings of the 4-th Annual Symposium on Combinatorial Pattern Matching*, Number 684 in *Lecture Notes in Computer Science*, pp. 106–119. Springer-Verlag.
- Kececioğlu, J. D., H.-P. Lenhof, K. Mehlhorn, P. Mutzel, K. Reinert, and M. Vingron (2000). A polyhedral approach to sequence alignment problems. *Discrete Applied Mathematics* 104(1-3), 143–186.
- Kemena, C. (2008). Local and global alignment construction using the seed approach. Master's thesis, Freie Universität Berlin, available at <http://www.seqan.de/publications/kemena08.pdf>.
- Kent, W. J. (2002). BLAT – the BLAST-like alignment tool. *Genome Research* 12(4), 656–64.
- Kernighan, B. W. and D. M. Ritchie (1988). *The C programming language, Second Edition*. Prentice-Hall.
- Kurtz, S. (2007). The vmatch large scale sequence analysis software. A Manual. Technical report, Center for Bioinformatics, University of Hamburg, available at <http://www.zbh.uni-hamburg.de/vmatch/virtman.pdf>.
- Kurtz, S., A. Phillippy, A. L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. L. Salzberg (2004). Versatile and open software for comparing large genomes. *Genome Biology* 5(2), R12.
- Lanctot, J. K., M. Li, B. Ma, S. Wang, and L. Zhang (1999). Distinguishing string selection problems. In *SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 633–642. Society for Industrial and Applied Mathematics.



- Langmead, B., C. Trapnell, M. Pop, and S. L. Salzberg (2009). Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology* 10(3).
- Larsson, N. J. and K. Sadakane (2007). Faster suffix sorting. *Theoretical Computer Science* 387(3), 258–272.
- Levenshtein, V. I. (1965). Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission* 1, 8–10.
- Lim, J. H. (2007). Algorithms for motif search. Master’s thesis, Freie Universität Berlin.
- Lutz, M. (2006). *Programming Python*. O’Reilly Media, Inc.
- Manber, U. and G. Myers (1990). Suffix arrays: a new method for on-line string searches. In *SODA ’90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pp. 319–327. Society for Industrial and Applied Mathematics.
- Margulies, M., M. Egholm, et al. (2005). Genome sequencing in microfabricated high-density picolitre reactors. *Nature* 437, 376–380.
- Mehlhorn, K. and S. Näher (1989). LEDA: A library of efficient data types and algorithms. In A. Kreczmar and G. Mirkowska (Eds.), *MFCS*, Volume 379 of *Lecture Notes in Computer Science*, pp. 88–106. Springer.
- Mehlhorn, K. and S. Näher (1999). *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press.
- Myers, E. W. (1999). A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM* 46(3), 395–415.
- Myers, G. and W. Miller (1995). Chaining multiple-alignment fragments in sub-quadratic time. In *SODA ’95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pp. 38–47. Society for Industrial and Applied Mathematics.
- Myers, G. J., T. Badgett, T. M. Thomas, and C. Sandler (2004). *The Art of Software Testing*. John Wiley & Sons.
- Navarro, G. (2001). A guided tour to approximate string matching. *ACM Computing Surveys* 33(1), 31–88.
- Navarro, G. and R. Baeza-Yates (1999). Very fast and simple approximate string matching. *Information Processing Letters* 72(1-2), 65–70.
- Navarro, G. and M. Raffinot (2002). *Flexible Pattern Matching in Strings*. Cambridge University Press.

- Needleman, S. B. and C. D. Wunsch (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J. Molecular Biol.* 48, 443–453.
- Notredame, C., D. G. Higgins, and J. Heringa (2000). T-Coffee: A novel method for fast and accurate multiple sequence alignment. *Journal of Molecular Biology* 302, 205–217.
- Pearson, W. R. (1990). Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods in Enzymology* 183, 63–98.
- Pearson, W. R. and D. J. Lipman (1988). Improved tools for biological sequence comparison. In *Proceedings of the National Academy of Sciences*, Volume 85, pp. 2444–2448.
- Pitt, W. R., M. A. Williams, M. Steven, B. Sweeney, A. J. Bleasby, and D. S. Moss (2001). The Bioinformatics Template Library – generic components for biocomputing. *Bioinformatics* 17(8), 729–737.
- Plauger, P. J., M. Lee, D. Musser, and A. A. Stepanov (2000). *C++ Standard Template Library*. Prentice Hall PTR.
- Price, A., S. Ramabhadran, and P. A. Pevzner (2003). Finding subtle motifs by branching from sample strings. *Bioinformatics* 19, supplement 2, ii149–ii155.
- Pugh, W. (1990). Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM* 33(6), 668–676.
- Rajasekaran, S., S. Balla, and C.-H. Huang (2005, October). Exact algorithms for planted motif problems. *Journal of Computational Biology* 12(8), 1117–1128.
- Rausch, T., A.-K. Emde, D. Weese, A. Döring, C. Notredame, and K. Reinert (2008). Segment-based multiple sequence alignment. In *Proceedings of the European Conference on Computational Biology (ECCB 2008)*.
- Rausch, T., S. Koren, G. Denisov, D. Weese, A.-K. Emde, A. Döring, and K. Reinert (2009). A consistency-based consensus algorithm for de novo and reference-guided sequence assembly of short reads. *Bioinformatics* 25(9).
- Rice, P., I. Longden, and A. Bleasby (2000). EMBOS: The european molecular biology open software suite. *Trends in Genetics* 16(6), 276–277.
- Saitou, N. and M. Nei (1987). The Neighbor-Joining method: a new method, for reconstructing phylogenetic trees. *Molecular Biology and Evolution* 4, 406–425.
- Sanger, F., S. Nicklen, and A. R. Coulson (1977). DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences* 74(12), 5463–5467.

- Schulz, M. H., D. Weese, T. Rausch, A. Döring, K. Reinert, and M. Vingron (2008). Fast and adaptive variable order markov chain construction. In *Proceedings of the 8th International Workshop in Algorithms in Bioinformatics (WABI'08)*, pp. 306–317. LNBI 5251: Springer Verlag.
- Sellers, P. H. (1980). The theory and computations of evolutionary distances: Pattern recognition. *Journal of Algorithms* 1, 359–373.
- Shamos, M. I. and D. J. Hoey (1976, October). Geometric intersection problems. In *Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, pp. 208–215.
- Siek, J., L.-Q. Lee, and A. Lumsdaine (2002). *The Boost graph library: User guide and reference manual*. C++ In Depth Series. Addison-Wesley, <http://www.boost.org>.
- Smith, T. F. and M. S. Waterman (1981). Identification of common molecular subsequences. *Journal of Molecular Biology* 147, 195–197.
- Sneath, P. H. A. and R. R. Sokal (1973). *Numerical taxonomy: The principles and practice of numerical classification*. San Francisco: W. H. Freeman.
- Staden, R. (1997). A strategy of dna sequencing employing computer programs. *Nucleic Acids Research* 6(7), 2601–2610.
- Stajich, J. E., D. Block, K. Boulez, S. E. Brenner, S. A. Chervitz, C. Dagdigan, G. Fuellen, J. G. R. Gilbert, I. Korf, H. Lapp, H. Lehtväslaiho, C. Matsalla, C. J. Mungall, B. I. Osborne, M. R. Pocock, P. Schattner, M. Senger, L. D. Stein, E. Stupka, M. D. Wilkinson, and E. Birney (2002). The Bioperl toolkit: Perl modules for the life sciences. *Genome Research* 12, 1611–1618.
- Stepanov, A. and M. Lee (1995). The Standard Template Library. Technical report, Hewlett-Packard Company.
- Stoeck, P. J. and G. N. Cameron (1991). The EMBL data library. *Nucleic Acids Research* 19, 2227–2230.
- Stroustrup, B. (2000). *The C++ programming language*. Addison-Wesley Longman Publishing Co., Inc.
- Tarhio, J. and E. Ukkonen (1990). Boyer-moore approach to approximate string matching (extended abstract). In *SWAT '90: Proceedings of the second Scandinavian workshop on Algorithm theory*, pp. 348–359. Springer-Verlag New York, Inc.
- Thompson, J. D., D. G. Higgins, and T. J. Gibson (1994). CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position specific gap penalties and weight matrix choice. *Nucleic Acids Research*. 22, 4673–4680.

- Thornton, K. (2003). libsequence: a C++ class library for evolutionary genetic analysis. *Bioinformatics* 19(17), 2325–2327.
- Ukkonen, E. (1985). Finding approximate patterns in strings. *Journal of Algorithms* 6(1), 132–137.
- UniProt Consortium (2008). The universal protein resource (UniProt). *Nucleic Acids Research* 39, D190–195.
- Vahrson, W., K. Hermann, J. Kleffe, and B. Wittig (1996). Object-oriented sequence analysis: SCL – a C++ class library. *Bioinformatics* 12(2), 119–127.
- Vakatov, D., K. Siyan, J. Ostell, and editors (2003). *The NCBI C++ Toolkit [Internet]*. National Library of Medicine, National Center for Biotechnology Information, Bethesda (MD).
- Vandevoorde, D. and N. M. Josuttis (2002). *C++ Templates*. Addison-Wesley.
- Venter, J. C., M. D. Adams, E. W. Myers, P. Li, R. J. Mural, G. G. Sutton, H. O. Smith, M. Yandell, C. A. Evans, R. A. Holt, J. D. Go-cayne, P. Amanatides, R. M. Ballew, D. H. Huson, J. R. Wortman, Q. Zhang, C. Kodira, X. H. Zheng, L. Chen, M. Skupski, G. Subrama-nian, P. D. Thomas, J. Zhang, G. L. G. Miklos, C. Nelson, S. Broder, A. G. Clark, J. Nadeau, V. A. McKusick, N. Zinder, A. J. Levine, R. J. Roberts, M. Simon, C. Slayman, M. Hunkapiller, R. Bolanos, A. Delcher, I. Dew, D. Fasulo, M. Flanigan, L. Florea, A. Halpern, S. Hannenhalli, S. Kravitz, S. Levy, C. Mobarry, K. Reinert, K. Remington, J. Abu-Threideh, E. Beasley, K. Biddick, V. Bonazzi, R. Brandon, M. Cargill, I. Chandramouliswaran, R. Charlab, K. Chaturvedi, Z. Deng, V. D. Francesco, P. Dunn, K. Eilbeck, C. Evangelista, A. E. Gabrielian, W. Gan, W. Ge, F. Gong, Z. Gu, P. Guan, T. A. Heiman, M. E. Higgins, R.-R. Ji, Z. Ke, K. A. Ketchum, Z. Lai, Y. Lei, Z. Li, J. Li, Y. Liang, X. Lin, F. Lu, G. V. Merkulov, N. Milshina, H. M. Moore, A. K. Naik, V. A. Narayan, B. Neelam, D. Nusskern, D. B. Rusch, S. Salzberg, W. Shao, B. Shue, J. Sun, Z. Y. Wang, A. Wang, X. Wang, J. Wang, M.-H. Wei, R. Wides, C. Xiao, C. Yan, A. Yao, J. Ye, M. Zhan, W. Zhang, H. Zhang, Q. Zhao, L. Zheng, F. Zhong, W. Zhong, S. C. Zhu, S. Zhao, D. Gilbert, S. Baumhueter, G. Spier, C. Carter, A. Cravchik, T. Woodage, F. Ali, H. An, A. Awe, D. Baldwin, H. Baden, M. Barnstead, I. Barrow, K. Beeson, D. Busam, A. Carver, A. Center, M. L. Cheng, L. Curry, S. Danaher, L. Davenport, R. Desilets, S. Dietz, K. Dodson, L. Doup, S. Ferriera, N. Garg, A. Gluecksmann, B. Hart, J. Haynes, C. Haynes, C. Heiner, S. Hladun, D. Hostin, J. Houck, T. H. a. Chinyere Ibegwam, J. Johnson, F. Kalush, L. Kline, S. Koduru, A. Love, F. Mann, D. May, S. McCawley, T. McIntosh, I. McMullen, M. Moy, L. Moy, B. Murphy, K. Nelson, C. Pfannkoch, E. Pratts, V. Puri,

- H. Qureshi, M. Reardon, R. Rodriguez, Y.-H. Rogers, D. Romblad, B. Ruhfel, R. Scott, C. Sitter, M. Smallwood, E. Stewart, R. Strong, E. Suh, R. Thomas, N. N. Tint, S. Tse, C. Vech, G. Wang, J. Wetter, S. Williams, M. Williams, S. Windsor, E. Winn-Deen, K. Wolfe, J. Zaveri, K. Zaveri, J. F. Abril, R. Guigo, M. J. Campbell, K. V. Sjolander, B. Karlak, A. Kejariwal, H. Mi, B. Lazareva, T. Hatton, A. Narechania, K. Diemer, A. Muruganujan, N. Guo, S. Sato, V. Bafna, S. Istrail, R. Lippert, R. Schwartz, B. Walenz, S. Yooseph, D. Allen, A. B., J. Baxendale, L. Blick, M. Caminha, J. Carnes-Stine, P. Caulk, Y.-H. Chiang, M. Coyne, C. Dahlke, A. D. Mays, M. Dombroski, M. Donnelly, D. Ely, S. Esparham, C. Fosler, H. Gire, S. Glanowski, K. Glasser, A. Glodek, M. Gorokhov, K. Graham, B. Gropman, M. Harris, J. Heil, S. Henderson, J. Hoover, D. Jennings, C. Jordan, J. Jordan, J. Kasha, L. Kagan, C. Kraft, A. Levitsky, M. Lewis, X. Liu, J. Lopez, D. Ma, W. Majoros, J. McDaniel, S. Murphy, M. Newman, T. Nguyen, N. Nguyen, M. Nodell, S. Pan, J. Peck, W. Rowe, R. Sanders, J. Scott, M. Simpson, T. Smith, A. Sprague, T. Stockwell, R. Turner, E. Venter, M. Wang, M. Wen, D. Wu, M. Wu, A. Xia, A. Zandieh, and X. Zhu (2001). The sequence of the human genome. *Science* 291 (5507), 1145–1434.
- Visual C++ (2002). Microsoft Corporation. *Microsoft Visual C++.Net Language Reference*. Microsoft Press.
- Wall, L. (2000). *Programming Perl*. O'Reilly & Associates, Inc.
- Wang, L. and T. Jiang (1994). On the complexity of multiple sequence alignment. *J. Comput. Biol.* 1, 337–348.
- Waterman, M. S. and M. Eggert (1987). A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons. *Journal of Molecular Biology* 197(4), 723–728.
- Weese, D. (2006). Entwurf und Implementierung eines generischen Substring-Index (german). Master's thesis, Humboldt-Universität zu Berlin, available at <http://www.seqan.de/publications/weese06.pdf>.
- Weese, D. and M. H. Schulz (2008, Jul). Efficient string mining under constraints via the deferred frequency index. In P. Perner (Ed.), *Proceedings of the 8th Industrial Conference on Data Mining (ICDM'08)*, pp. 374–388. LNAI 5077: Springer Verlag.
- Weiner, P. (1973). Linear pattern matching algorithms. *Annual Symposium on Switching and Automata Theory* 0, 1–11.
- Wilson, M. (2004). *Imperfect C++: Practical Solutions for Real-Life Programming*. Addison-Wesley Professional.

- Wöhrle, H. (2006). Multidimensionales Chaining mit Deferred Range Trees (german). Master's thesis, Freie Universität Berlin, available at <http://www.seqan.de/publications/woehrle06.pdf>.
- Wu, S. and U. Manber (1992). Fast text searching: allowing errors. *Communications of the ACM* 35(10), 83–91.
- Wu, S. and U. Manber (1994). A fast algorithm for multi-pattern searching. Report TR-94-17, Department of Computer Science, University of Arizona.
- Zerbino, D. R. and E. Birney (2008). Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research* 18, 821–829.
- Zhang, Z., S. Schwartz, L. Wagner, and W. Miller (2000, Feb). A greedy algorithm for aligning dna sequences. *Journal of Computational Biology* 7(1/2), 203–214.

# Index

- AbndmAlgo** (Spec), 117
- abstraction penalty, 61
- accessibility, *Sec 14.2.1*
  - of Seqan, 174
- addChild** (Func), 158
- addEdge** (Func), 157, 158
- addSeed** (Func), 135, *Tab 18*
- addVertex** (Func), 157
- adjacency list, 158
- AdjacencyIterator** (Tag), 159
- adjacent, 157
- affine gap costs, 84, 88, 117
- agglomerative clustering, 93
- AhoCorasick** (Spec), 111
- algorithm design, 10
- Align** (Class), 83, 84, 87, 163, 165, 184
- AlignConfig** (Class), 93
- Alignment** (Spec), 158, 163
- alignment, 6, 7, *Ch 8*, 83, *Fig 21*
  - compatible, 164
  - data structures, *Sec 9.2*
  - global, *Sec 9.5*
  - graph, 163, *Sec 13.2*, 164, *Fig 38*
  - graph vs. matrix, 164, *Fig 37*
  - intersecting, 128
  - maximum weight trace problem, 166, *Alg 36*
  - score, *Sec 9.3*, 85
  - tools, 9, *Fig 4*
    - Clustal W, *Sec 1.2.3*
- alignment algorithm, 86, *Sec 9.4*
  - Gotoh** (Tag), 88
  - Hirschberg** (Tag), 90
  - MyersHirschberg** (Tag), 91
  - NeedlemanWunsch** (Tag), 87
  - banded, 100, *Fig 25*, *Sec 9.6.4*
  - free end-gaps, *Sec 9.5.4*
  - global, 87, *Tab 11*
  - Gotoh, *Sec 9.5.2*, 90, *Alg 3*
  - Hirschberg, *Sec 9.5.3*, 91, *Alg 4*, 92, *Fig 23*
  - local, *Sec 11.1*
    - Smith-Waterman, 127, *Alg 19*
    - Waterman-Eggert, 128, *Alg 20*
  - maximum weight trace problem, *Sec 13.2.2*
  - Needleman-Wunsch, *Sec 9.5.1*, 88, *Fig 22*, 89, *Alg 2*
  - progressive, 8, *Fig 3*, *Sec 9.5.5*, 94, *Alg 5*
  - runtimes, 172, *Tab 30*
  - seed based, *Sec 11.2*
  - segment match refinement, *Sec 13.2.3*
  - Smith-Waterman, *Sec 11.1.1*
  - Waterman-Eggert, *Sec 11.1.2*
- alignment problem, 86
- Alloc** (Spec), 39, 43, 63, 67–69, *Fig 14*
- allocate** (Func), 56, 58
- Allocator** (Class), 57, *Tab 1*
- allocator, 56
  - runtimes, 58, *Fig 11*
  - usage, 56
- alphabet, 55, 59
  - simple, 59
  - size, 59, 103
  - types, 60, *Tab 1*
- amino acid, 3
- AminoAcid** (Class), 60, 84
- antidiagonal, 132

- append** (Func), 72
- appending seeds, 95, 165
- appendValue** (Func), 66, 69
- approximate
  - motif finding, 125
  - string matching, 115, *Sec 10.3*
    - algorithms, 117, *Tab 15*
    - best algorithm, 122, *Fig 30*
- argument-dependent name lookup, 35
- Array** (Spec), 67–69
- array string, 66, *Sec 8.3.2*
- arrayCopy** (Func), 60
- ArrayGaps** (Spec), 80–82
- assembly, *Sec 1.2.1*
- assign** (Func), 63
- assignable, 28, 29
- assignValue** (Func), 59
- atEnd** (Func), 62
- Automaton** (Spec), 158, 160
- Automaton** (Tag), 159
- automaton, 160, *Sec 13.1*
- Backward Factor Searching, 107
- banded alignment, 94, 100, *Fig 25*, *Sec 9.6.4*, 181
- bandedAlignment** (Func), 129
- bandedChainAlignment** (Func), 100, 184
- basic contents of SeqAn, *Ch 7*
- BATS (software library), 12
- begin** (Func), 62
- begin position, 65, 84
- begin view position, 80
- beginPosition** (Func), 103
- bellman\_ford\_algorithm** (Func), 160
- BFAM (Spec), 104, 109, 111
- BFAM algorithm, *Sec 10.1.4*, 108, *Alg 12*
- BFAM<Oracle> (Spec), 109
- BFAM<Trie> (Spec), 109
- BfsIterator** (Tag), 159
- binary search, 147
- Bio++, 12
- Bioperl, 12
- Biopython, 12
- Bioruby, 12
- Bioconductor, 13
- bioinformatics
  - libraries, *Sec 2.2*
  - sequences, *Sec 1.1*
- BioJava, 13
- biological sequence, 3
- biopolymer, 3
- bits, 69
- BitsPerValue** (Meta), 59, 71
- blank, 79
- BLAST, 6, *Sec 1.2.2*
- Blat** (Tag), 135
- Block** (Spec), 68–70, *Fig 15*
- block string, *Sec 8.3.3*
- Blosum30** (Class), 84
- Blosum62** (Class), 84
- Blosum80** (Class), 84
- BNDM algorithm, 108, *Sec 10.1.5*, 109, *Alg 13*
- BndmAlgo** (Spec), 104, 109
- bottom seed, 95, 180, 183
- bottom-up traversal, 153, *Alg 32*
- BottomUp** (Spec), 155
- breadth\_first\_search** (Func), 160
- brute-force searching, *Sec 10.1.1*
- BTL (software library), 13
- bunch, 141
- byte, 69
- C++, *Sec 5.1*
- C-style string, 33, 72
- capacity, 66
- chain, 95
- chain score, 96
- chaining, 11, 94, *Sec 9.6*, 180
  - gap scoring schemes, 98, *Fig 24*
  - generic, 96, *Alg 6*
  - global, 94



- local, 132, *Sec 11.2.2*, 134, *Alg 24*, *Fig 31*
  - sparse dynamic programming, 99, *Alg 7*
- ChainSoP** (Spec), 98, 99
- changing gaps, 164
- Chaos** (Tag), 135, 180
- ChunkPool** (Spec), 57
- ClassPool** (Spec), 57
- clear** (Func), 56, 57
- Clustal W, *Sec 1.2.3*
- clustering, 7, 93
  - agglomerative, 93
  - neighbor-joining, 93
- codon, 3
- cols** (Func), 84
- comparator, *Sec 8.7*
- compatible alignments, 164
- components of tools, 9, *Fig 4*
- computeGenerousCapacity** (Func), 67
- concat** (Func), 76, 77
- ConcatDirect** (Spec), 76
- concatenator, 77, 141
- concept, 28
- conclusion, *Ch 16*
- container, 55
- container** (Func), 61
- contiguous sequence, 65
- conversion, 55, *Sec 7.6*
- Convert** (Meta), 62
- convert** (Func), 62
- convertAlignment** (Func), 165
- convertInPlace** (Func), 74
- core design, 27
- count sort, 143
- counting of string values, 41, *Alg 1*
- createOracle** (Func), 163
- createQGramIndexSAOnly** (Func), 143
- createSuffixArray** (Func), 145, 146
- createSuffixTrie** (Func), 161
- createTrie** (Func), 161
- dag\_shortest\_path** (Func), 160
- database growth, 4, *Fig 1*
- DDDoc, *see* DotDotDoc
- deallocate** (Func), 56, 58
- DefaultIteratorSpec** (Meta), 61
- DefaultOverflowExplicit** (Meta), 66
- DefaultOverflowImplicit** (Meta), 66
- deferred data structure, 141
- deoxyribonucleic acid, *see* DNA
- Dependent** (Spec), 77
- dependent type, 37
- depth\_first\_search** (Func), 160
- descriptor, 157
- design
  - examples, *Ch 6*
  - goals, *Sec 4.2*
  - of SeqAn, *Ch 4*
  - quality, *Ch 14*
  - techniques, *Ch 5*
- dfa, 160
- DfsPreorder** (Tag), 159
- diagonal, 129
- dictionary, 99
- dijkstra** (Func), 160
- dimension (of a seed), 95
- Directed** (Spec), 158
- directed graph, 157
- directory table, 143
- distance
  - of seeds, 85
  - of sequences, *Sec 9.3.2*
- distribution (of SeqAn), *Sec 14.2.1*
- DNA, 3, 60
- Dna** (Class), 37, 43, 60, 71, 73, 75
- Dna** (Spec), 104
- Dna5** (Class), 60, 73
- documentation, *Sec 14.2.2*
- DotDotDoc, 176, *Fig 41*
- DotDrawing** (Tag), 64
- DPSearch** (Spec), 117
- dynamic function binding, 36

- dynamic programming, 5, 7, 86, 87, 96, 100
  - sparse, 97
  - chaining, 99, *Alg 7*
  - maximum weight trace, 166, *Alg 36*
- edge, 157
- EdgeDescriptor** (Meta), 157
- EdgeIterator** (Tag), 159
- edit distance, 84, 86, 92, 115, 117, 118, 122
- EM-algorithm, 137
- Embl** (Tag), 63, 64
- EMBOSS, 12
- end** (Func), 62
- end gap, 92
- end position, 65, 84
- end view position, 80
- end-gaps free alignment, *Sec 9.5.4*
- endPosition** (Func), 103
- enhanced suffix array, 125, 150, *Sec 12.3*
- EPatternBranching** (Spec), 137
- ESA\_BWT** (Tag), 151
- ESA\_ChildTab** (Tag), 151, 153, 155
- ESA\_LCP** (Tag), 141, 150, 151, 153
- ESA\_SA** (Tag), 141, 151, 153
- ESA\_Text** (Tag), 141, 151
- Exact** (Tag), 66
- exact motif finding, 125
- exact pattern matching, 103, *Sec 10.1*
  - algorithms, 104, *Tab 13*
  - best online algorithm, 110, *Fig 28*
- expandable, 66
- extendSeed** (Func), 130, 131
- extendSeeds** (Func), 130
- extensibility, 24, *Sec 4.2.5*, 34, 173
- extension, *see* seed extension
- External** (Spec), 68, 71
- external string, *Sec 8.3.5*
- factor, *see* segment
  - automaton, 107
  - based searching, 107, *Alg 11*
  - oracle, 160, 162, *Fig 36*
    - construction, 163, *Alg 35*
  - trie, 162
    - construction, 161, *Alg 34*
- False** (Tag), 66
- Fasta** (Tag), 63, 64
- Fiber** (Meta), 141
- fiber, 141
  - of q-gram index, 145, *Tab 23*
  - of suffix array, 151, *Tab 25*
- file formats, 64, *Tab 3*
  - DotDrawing** (Tag), 64
  - Embl** (Tag), 64
  - Fasta** (Tag), 64
  - Genbank** (Tag), 64
  - Raw** (Tag), 64
- file input/output, *Sec 7.7*
- file reader string, 63
- FileReader** (Spec), 63, 181
- fill** (Func), 66
- filtering, 120
- find** (Func), 101, 102, *Fig 26*, 103, 111, 116
- findBegin** (Func), 103, 116
- finder, 101
- finding, *see* motif finding, *see* searching
- findMotif** (Func), 136
- floyd\_warshall** (Func), 160
- ford\_fulkerson** (Func), 160
- free gap, 92
- function overload resolution, 30
- functionals, 11
- gap, 79, 80
  - column, 83
  - costs
    - affine, 84
    - linear, 84
  - leading, 80

- pattern, 79, 192
  - scoring for chaining, 98, *Fig 24*
  - trailing, 80
- gapped q-gram, 45, 142
- gapped sequence, 79, *Sec 9.1*, 80
  - runtimes, 83, *Fig 20*
- GappedShape (Spec), 47, 142
- GappedValueType (Meta), 61
- GappedXDrop (Tag), 130, 132
- Gaps (Class), 79, 80, *Fig 19*, *Tab 8*, 81–83
  - ArrayGaps (Spec), 81
  - SequenceGaps (Spec), 81
  - SumlistGaps (Spec), 82
- GenBank, 4, *Fig 1*
- Genbank (Tag), 64
- gene, 3
- generality, 23, *Sec 4.2.3*, 28, 34, 172
- generic, 21, 33
- generic programming, 11, 12, 16, 24, 27, 28, *Sec 5.2*, 33, 173
- GenericShape (Spec), 50, 51
- Generous (Spec), 76
- Generous (Tag), 66, 67
- genome assembly, *Sec 1.2.1*
- getFibre (Func), 141
- getOccurrences (Func), 154
- GetValue (Meta), 56
- getValue (Func), 56, 65
- global alignment, 86, *Sec 9.5*
- global chaining, 94, 96, 180
- global function, 33
- global interface, 16, 27, *Sec 5.4*, 172, 173
- globalAlignment (Func), 38, 87, 93
- globalChaining (Func), 97, 183
- goNext (Func), 63
- Gotoh (Tag), 87, 88
- Gotoh’s algorithm, *Sec 9.5.2*, 90, *Alg 3*
- Graph (Class), 157, 158, *Tab 27*, 159, 160
  - Alignment (Spec), 163
  - Automaton (Spec), 160
  - Directed (Spec), 158
  - Hmm (Spec), 158
  - Tree (Spec), 158
  - Undirected (Spec), 158
  - WordGraph (Spec), 158
- graph, 157
  - algorithms, 160, *Tab 29*
  - alignment graph, *Sec 13.2*
  - automata, *Sec 13.1*
  - data structures, *Ch 13*
  - directed, 157
  - undirected, 157
- graph iterators, 159, *Tab 28*
  - AdjacencyIterator (Tag), 159
  - BfsIterator (Tag), 159
  - DfsPreorder (Tag), 159
  - EdgeIterator (Tag), 159
  - OutEdgeIterator (Tag), 159
  - VertexIterator (Tag), 159
- greedy, 8
- guide tree, 8, 93
- Hamming distance, 121, 139
- HammingHorspool (Spec), 121
- HardwiredShape (Spec), 50, 51, 142
- hash (Func), 46, 48, 51
- hashing
  - locality-sensitive, 46, *Fig 7*
  - runtimes, 51, *Fig 9*
- hashNext (Func), 49, 51
- haystack, 101
- heaviestCommonSubsequence (Func), 165
- hierarchical
  - clustering, 7, 93
  - verification, 120
- hierarchy of refinements, 24
- Hirschberg (Tag), 39, 87, 90
- Hirschberg’s algorithm, *Sec 9.5.3*, 91, *Alg 4*, 92, *Fig 23*
- Hmm (Spec), 158
- Horspool (Spec), 103–105, 109

- Horspool's algorithm, 105, *Sec 10.1.2*, 106, *Alg 9*
- Human Genome Project, 4
- Index** (Class), 141, *Tab 21*
  - Index\_ESA** (Spec), 150
  - Index\_QGram** (Spec), 142
  - Index\_Wotd** (Spec), 141
  - PizzaChili** (Spec), 141
- index, 141
  - enhanced suffix array, *Sec 12.3*
  - q-gram, *Sec 12.1*
  - suffix array, *Sec 12.2*
- index data structure, *Ch 12*
- Index\_ESA** (Spec), 141, 150
- Index\_QGram** (Spec), 141, 142
- Index\_Wotd** (Spec), 141
- Infix** (Meta), 74
- infix, 74
- infix** (Func), 75
- infix searching, 116
- integration, *Sec 4.2.6*, 34, 173
- interface, 33
- intersecting alignments, 128
- inverse suffix array, 147, 151
- IsContiguous** (Meta), 66
- IsSimple** (Meta), 60
- Iter** (Class), 155, *Tab 26*
  - BottomUp** (Spec), 155
  - MUMs** (Spec), 155
  - MaxRepeats** (Spec), 155
  - MultiMEMs** (Spec), 155
  - ParentLinks** (Spec), 155
  - StdIteratorAdaptor** (Spec), 73
  - SuperMaxRepeats** (Spec), 155
  - TopDown** (Spec), 155
- Iterator** (Meta), 61, 62, 159
- iterator, 11, 55, 61, *Sec 7.5*, 159, *see*
  - Iter** (Class)
  - of suffix trees, 155, *Fig 26*
- iterator types
  - Rooted** (Tag), 61
  - Stable** (Tag), 62
  - Standard** (Tag), 61
- Iupac** (Class), 60
- Java, 13
- k-mismatch problem, 121
- koenig lookup, *see* argument-dependent name lookup
- kruskals\_algorithm** (Func), 160
- LAGAN, *Ch 15*, 180, *Alg 37*
  - algorithm, 179, *Fig 43*
  - run times, 185, *Fig 44*
  - source code, *Sec A.3*
- LarssonSadakane** (Tag), 147
- LCP table, 145, *Fig 33*, 151, *Sec 12.3.1*
  - construction, 152, *Alg 31*
- leading gap, 80
- leftPosition** (Func), 95
- length, 65
- length** (Func), 33, 41, 45, 72
- Levenshtein distance, 86
- lexical, 76
- lexicographic order, 75
- LGPL, 177
- libcov, 13
- library
  - design, 11, 19, 21, *Ch 4*, 28
  - examples, *Ch 6*
  - for bioinformatics, *Sec 2.2*
  - quality, *Ch 14*
  - tool stitching, *Sec 2.2.1*
- library-spanning programming, 45
- libsequence, 13
- linear gap costs, 84
- loadMeta** (Func), 63
- local alignment, 6, 86, 125, 126, *Sec 11.1*
  - Smith-Waterman algorithm, 127, *Alg 19*
  - Waterman-Eggert algorithm, 128, *Alg 20*

- local chaining, 132, *Sec 11.2.2*, 134, *Alg 24, Fig 31*, 180
- localAlignment** (Func), 126
- localAlignmentNext** (Func), 128
- locality-sensitive hashing, 46, *Fig 7*, 138
- Log2** (Meta), 38
- logo of SeqAn, 15, *Fig 5*
- longest common prefix, 75, 151
  - table, *see* LCP table
- longest common substring, 151
- ManberMyers** (Tag), 147
- Manhattan** (Spec), 98, 99
- match refinement, 167, *Fig 39*
- MatchExtend** (Tag), 130
- matchRefinement** (Func), 167
- maximum likelihood estimation, 137
- maximum weight trace problem, 86, 94, 165, *Sec 13.2.2*, 166, *Alg 36*
- MaxRepeats** (Spec), 155
- member function, 33
- memory allocation, *Sec 7.2*
- memory usage
  - TagAllocateStorage** (Tag), 56
  - TagAllocateTemp** (Tag), 56
- Merge** (Tag), 135
- merge sort, 146
- merging seeds, 132
- meta data, 63
- metafunction, 11, 16, 27, 36, *Sec 5.5*
- metaprogramming, *Sec 5.6.1*
- method (in OOP), 33
- metric, 85
- mismatch, 121
  - searching, *Sec 10.4.1*
- ModExpand** (Spec), 60
- modifier, 60, 73
- ModifierString** (Class), 73, *Tab 6*
  - ModReverse** (Spec), 73
  - ModView** (Spec), 73
- ModReverse** (Spec), 73
- ModView** (Spec), 73
- monomer, 3
- most general function, 30
- motif, 125
  - in multiple sequences, *Sec 11.3*
  - model, 136
    - OMOPS** (Tag), 136
    - OOPS** (Tag), 136
    - TCM** (Tag), 136
    - ZOOPS** (Tag), 136
  - occurrence, 136
    - exact, 136
- motif discovery, *see* motif finding
- motif finding, 86, 125, *Ch 11*
  - algorithms, 137, *Tab 20*
  - PMSF, 139, *Alg 26*
  - Projection, 138, *Alg 25*
  - Smith-Waterman, 127, *Alg 19*
  - Waterman-Eggert, 128, *Alg 20*
  - heuristic, *Sec 11.3.1*
- MotifFinder** (Class), 136, 137
- Move** (Tag), 59
- move, 55
  - constructor, 59
  - operation, 58, *Sec 7.2*
- move** (Func), 63
- moveValue** (Func), 59
- mRNA, 3
- MultiBFAM** (Spec), 111
- MultiMEMS** (Spec), 155
- Multiple BFAM algorithm, *Sec 10.2.2*, 115, *Alg 15*
- multiple pattern matching, *Sec 10.2*
  - algorithms, 111, *Tab 14*
  - best online algorithm, 114, *Fig 29*
- multiple sequence motif, *Sec 11.3*
- MultipleShiftAnd** (Spec), 111
- MultiPool** (Spec), 57
- MultiSeed** (Spec), 95
- MUMs** (Spec), 155
- Myers** (Spec), 117

- Myers' algorithm, *Sec 10.3.2*, 119,  
*Alg 17, Sec A.1*
- MyersHirschberg (Tag), 87, 91
- NCBI C++ Toolkit, 14
- needle, 101
- Needleman-Wunsch algorithm, 11,  
*Sec 9.5.1*, 88, *Fig 22*, 89,  
*Alg 2*
- NeedlemanWunsch (Tag), 38, 87
- neighbor-joining, 7, 93
- node, 157
- non-gaps, 80
- nucleobase, 3
- nucleotide, 3
- object-oriented programming, 21, 27,  
32, *Fig 6*, 33, 171
- OMOPS (Tag), 136–139
- online searching, 103
- OOP, *see* object-oriented program-  
ming
- OOPS (Tag), 136, 137
- open-closed principle, 24, 33
- optimal alignment, 85
- oracle, 107, 162, *Fig 36*  
construction, 163, *Alg 35*
- ord, 59, 143
- ordValue (Func), 45, 59
- OutEdgeIterator (Tag), 159
- overflow strategy, 66, *Sec 8.2, Tab 4*
- overlap alignment, 5, 86
- overlapping seeds, 132
- overload resolution, 38
- Owner (Spec), 76, 77
- Owner<ConcatDirect> (Spec), 77
- Packed (Spec), 68, 71, *Fig 17*
- packed string, *Sec 8.3.4*
- pairwise motif finding, 125
- Pam (Spec), 84
- ParentLinks (Spec), 155
- parseString (Func), 160
- partition filtering, *Sec 10.3.3*, 120,  
*Alg 18*
- path label, 160  
in a suffix tree, 151
- Pattern (Class), 103, 105, 107, 117,  
121  
HammingHorspool (Spec), 121  
Horspool (Spec), 105  
WildShifAnd (Spec), 121
- pattern, 101
- pattern matching, 101, *Ch 10*, 125
- performance, 21, 22, *Sec 4.2.1*, 28,  
32, 171
- Pex (Spec), 117
- PEX algorithm, *Sec 10.3.3*
- PizzaChili (Spec), 141
- PMS1 (Spec), 137
- PMSP (Spec), 137, 139, *Alg 25*
- POD (plain old data), 60
- polymorphism, 24, 30, 172
- pool allocator, 57, *Fig 17*, 58
- pop (Func), 69
- position, 65
- position (Func), 103
- position table, 143
- positive definiteness, 85
- Prefix (Meta), 74
- prefix, 74
- prefix (Func), 75
- prefix searching, 116, 117
- prims\_algorithm (Func), 160
- progressive alignment, 7, 8, *Fig 3*,  
*Sec 9.5.5*, 94, *Alg 5*
- Projection (Spec), 137, 138, *Alg 25*
- projection, 83
- property map, 157
- protein, 3
- proxy class, 56
- pseudo container, 55
- push (Func), 69
- q-gram, 7, 142  
gapped, 142

- hashing, 139, 142
  - index, *Sec 12.1*, 144, *Fig 17*
    - construction, 144, *Alg 27*
    - fibers, 145, *Tab 23*
  - ungapped, 142
  - QGram\_Dir** (Tag), 145
  - QGram\_SA** (Tag), 145
  - QGram\_Shape** (Tag), 145
  - QGram\_Text** (Tag), 145
  - quality of SeqAn, *Ch 14*
- R** (programming language), 13
- random access, 65
  - Raw** (Tag), 63, 64
  - read** (Func), 63, 64, 84
  - Reference** (Meta), 56
  - reference, 55
  - refinement, *Sec 4.2.4*, 29, 32, 167, 171, 172
  - regular expression, 123, *Tab 16*
  - removeEdge** (Func), 157, 158
  - removeGapCols** (Func), 83
  - removeVertex** (Func), 157
  - repeat, 153
  - replace** (Func), 66
  - repLength** (Func), 154
  - representative** (Func), 154
  - reserve** (Func), 66
  - residue, 3
  - resize** (Func), 66, 142
  - reusability, 9
  - reverseInPlace** (Func), 74
  - ribonucleic acid, 3
  - rightPosition** (Func), 95
  - Rna** (Class), 60
  - Rna5** (Class), 60
  - Rooted** (Tag), 61
  - rooted iterator, 61
  - rows** (Func), 83
  - runtime
    - for alignment algorithms, 172, *Tab 30*
    - for hashing, 51, *Fig 9*
    - for searching, 110, *Fig 27*
      - approximate, 122, *Fig 30*
      - exact, 110, *Fig 28*
      - multiple, 114, *Fig 29*
    - of allocators, 58, *Fig 11*
    - of gapped sequences, 81, *Tab 9*, 83, *Fig 20*
    - of strings
      - appending values, 70, *Fig 16*
      - for random access, 68, *Fig 13*
  - safe shift width, 105, 112
  - SAQSort** (Tag), 147
  - scan, 160
  - SCL (software library), 14
  - Score** (Class), 64, 84, *Tab 10*, 98
    - ChainSoP** (Spec), 98
    - Manhattan** (Spec), 98
    - Pam** (Spec), 84
    - ScoreMatrix** (Spec), 84
    - Simple** (Spec), 84
    - Zero** (Spec), 98
  - score, 7
    - alignment, 85
    - chain, 96
    - sum of pairs, 85
  - ScoreMatrix** (Spec), 64, 84
  - scoring scheme, 84, *Sec 9.3.1*
  - script languages, *Sec 2.2.1*
  - searching, *Ch 10*, 102, *Fig 26*
    - approximate, *Sec 10.3*, 117, *Tab 15*
      - Myers, *Sec 10.3.2*, 119, *Alg 17*
      - partition filtering, *Sec 10.3.3*, 120, *Alg 18*
    - runtimes, 122, *Fig 30*
    - Sellers, *Sec 10.3.1*, 118, *Alg 16*
  - exact, 104, *Tab 13*
    - BFAM, *Sec 10.1.4*, 108, *Alg 12*
    - BNDM, *Sec 10.1.5*, 109, *Alg 13*
    - brute-force, *Sec 10.1.1*, 105, *Alg 8*

- factor based search, 107, *Alg 11*
- Horspool, *Sec 10.1.2*, 106, *Alg 9*
- runtimes, 110, *Fig 27*, *Fig 28*
- Shift-Or, *Sec 10.1.3*
- k-mismatch problem, *Sec 10.4.1*
- motifs, *see* motif finding
- multiple, 111, *Sec 10.2*, *Tab 14*
  - Multiple BFAM, *Sec 10.2.2*, 115, *Alg 15*
  - runtimes, 114, *Fig 29*
  - Wu-Manber, *Sec 10.2.1*, 113, *Alg 14*
- with mismatches, 121
- with wildcards, *Sec 10.4.2*
- Seed** (Class), 95, *Tab 12*, 126, 129, 181
  - MultiSeed** (Spec), 95
  - SimpleSeed** (Spec), 95
- seed, 7, 94, *Sec 9.6.1*, 128, 165, 167
  - adding modes, 135, *Tab 18*
  - bottom, 95
  - chain, 95, 132
  - chain score, 96
  - combination, *Sec 11.2.2*
  - dimension, 95
  - extension, 129, 130, *Tab 17*
    - gapped X-drop, 133, *Alg 23*
    - match extension, 130, *Alg 21*
    - ungapped X-drop, 131, *Alg 22*
  - top, 95
  - weight, 95
- seed based motif finding, *Sec 11.2*
- seed extension, *Sec 11.2.1*
- SeedSet** (Class), 135, 181
- segment, 74, *Fig 18*, *Sec 8.6*, 182
  - host, 74
  - match refinement, 167, *Sec 13.2.3*
- Sellers' algorithm, *Sec 10.3.1*, 118, *Alg 16*
- semi global alignment, 86
- SeqAn, 15
  - documentation, 176, *Fig 41*
  - library design, *Ch 4*
  - logo, 15, *Fig 5*
  - project, *Ch 3*
- sequence, 65, *Ch 8*
  - adaptor, *Sec 8.4*
  - analysis, 1, *Ch 1*, 11
    - examples, *Sec 1.2*
  - assembly, 5, *Sec 1.2.1*
  - databases, 4, *Fig 1*
  - distance, 85, *Sec 9.3.2*
  - in bioinformatics, *Sec 1.1*
  - modifier, *Sec 8.5*
  - similarity, 6, 85, *Sec 9.3.2*
- sequence analysis, 5
- SequenceGaps** (Spec), 80–82
- sequencing
  - read, 5
  - shotgun, 6, *Fig 2*
- setBegin** (Func), 74
- setEnd** (Func), 74
- SetHorspool** (Spec), 111
- setLeftPosition** (Func), 95
- setPosition** (Func), 103
- setRightPosition** (Func), 95
- setScoreLimit** (Func), 116
- setWeight** (Func), 95
- Shape** (Class), 45, 47, 48, 50, 51, *Fig 8*, 142, *Tab 22*, 143
  - GappedShape** (Spec), 142
  - HardwiredShape** (Spec), 142
  - SimpleShape** (Spec), 142
  - UngappedShape** (Spec), 142
- shape, 45, 142
- Shift-Or algorithm, 105, *Sec 10.1.3*
- ShiftOr** (Spec), 104, 109
- shim, 25, 33, 42, 44, 173
- shortcut, 39, *Sec 5.6.3*, 172
- shotgun sequencing, 5, 6, *Fig 2*
- similarity (of sequences), 85, *Sec 9.3.2*
- Simple** (Spec), 84, 104
- simple type, 59, 65, *see* alphabet



- SimpleAlloc** (Spec), 57
- SimpleChain** (Tag), 135
- SimpleSeed** (Spec), 95, 129, 181
- SimpleShape** (Spec), 47, 48, 50, 51, 142
- simplicity, 23, *Sec 4.2.2*, 30, 32
- Single** (Tag), 135
- single nucleotide polymorphism, 13
- SinglePool** (Spec), 57
- Size** (Meta), 37, 43
- skew algorithm, 145
- Skew3** (Tag), 147
- Skew7** (Tag), 146, 147
- skip list, 99, 192
- Smith-Waterman algorithm, 127, *Alg 19*
- software development, 10
- software library, 1, 9, *Ch 2*
- software tools
  - BLAST, *Sec 1.2.2*
  - Clustal W, *Sec 1.2.3*
  - LAGAN, *Ch 15*
- source, 79
- source** (Func), 81
- source position, 79
- span** (Func), 46–48
- span (of a shape), 142
- sparse dynamic programming, 11, 97, 165
  - chaining, 99, *Alg 7*
  - maximum weight trace, 166, *Alg 36*
- specialization, 24
  - Allocator** (Class), 57, *Tab 1*
  - Gaps** (Class), 80, *Tab 8*
  - Graph** (Class), 158, *Tab 27*
  - Index** (Class), 141, *Tab 21*
  - Iter** (Class), 155, *Tab 26*
  - ModifierString** (Class), 73, *Tab 6*
  - Score** (Class), 84, *Tab 10*
  - Shape** (Class), 142, *Tab 22*
  - StringSet** (Class), 76, *Tab 7*
  - String** (Class), 68, *Tab 5*
  - specializationSeed** (Class), 95, *Tab 12*
- stability, *Sec 14.2*
- Stable** (Tag), 62
- stable iterator, 62
- stable sort, 143
- stable sorting, 146
- Standard** (Tag), 61
- start gap, 92
- static function binding, 36, 51
- StdIteratorAdaptor** (Spec), 73
- streams, 65
- String** (Class), 33, 35, 41, 43, 63, 65, 67, 68, *Tab 5*, 173, 176, 181
  - Alloc** (Spec), 67
  - Array** (Spec), 69
  - Block** (Spec), 69
  - External** (Spec), 71
  - FileReader** (Spec), 63
  - Packed** (Spec), 71
- string, 4, 65, *Sec 8.1*, *see*
  - String** (Class)
  - runtimes for appending, 70, *Fig 16*
  - runtimes for random access, 68, *Fig 13*
  - value counting, 41, *Alg 1*
- string indices, *Ch 12*
- string set, *Sec 8.8*
  - id, 77
- StringSet** (Class), 65, 76, *Tab 7*, 77, 141
  - ConcatDirect** (Spec), 76
  - Dependent** (Spec), 77
  - Generous** (Spec), 76
  - Owner** (Spec), 76, 77
  - Tight** (Spec), 76
- stringToShape** (Func), 142
- suboptimal local alignments, 128
- substring, *see* segment
- Suffix** (Meta), 74
- suffix, 74
- suffix** (Func), 75

- suffix array, 10, 11, 141, 145, *Fig 33*,  
     *Sec 12.2*, 151
  - construction, 145, 147, *Tab 24*,  
 148, *Alg 29*, 149, *Alg 29*
  - enhanced, 141, 150, *Sec 12.3*
  - fibers, 151, *Tab 25*
  - searching, 147, 150, *Alg 30*
- suffix tree, 11, 125, 141, 150–152,  
     *Fig 34*
  - bottom-up traversal, 153, *Alg 32*
  - iterators, 155, *Fig 26*
- suffix trie, 161
- sum list, 82, 192, *Sec A.2*, 193, *Fig 45*
  - searching, 193, *Alg 38*
- sum of pairs score, 85, 93
- SumlistGaps** (Spec), 80–82
- supermaximal repeat, 153, 154,  
     *Alg 33*
- SuperMaxRepeats** (Spec), 153, 155
- supply array, 162
- sweep line, 96
- Swiss-Prot, 4, *Fig 1*
- switch argument, 66
- symmetry (of distance metric), 85
  
- tag class, 31, 38
- tag dispatching, 38, *Sec 5.6.2*, 172
- TagAllocateStorage** (Tag), 56
- TagAllocateTemp** (Tag), 56
- TCM (Tag), 136
- techniques used in SeqAn, *Ch 5*
- template, 28
- template subclassing, 16, 24, 27,  
     *Sec 5.3*, 30, 31, 33, 36, 38,  
     50, 51, 172, 173
- testing, 174, *Sec 14.2.1*
- Tight** (Spec), 76
- tool stitching, *Sec 2.2.1*
- tools, *see* software tools
- top seed, 95, 180, 183
- TopDown** (Spec), 155
- topological\_sort** (Func), 160
- ToStdAllocator** (Class), 58
  
- Trac, 175, *Fig 40*
- trace, 86, 164, *Fig 38*
  - unique, 164
- trailing gap, 80
- transcription, 3
- transitive\_closure** (Func), 160
- translation, 3
- Tree** (Spec), 158
- Tree** (Tag), 159
- triangle inequality, 85
- trie, 107, 160, 161, *Fig 35*
  - construction, 161, *Alg 34*
  - factor trie, 162
  - suffix trie, 161
- triplet extension, 94
- True** (Tag), 66
- type traits, 11
  
- Undirected** (Spec), 158
- undirected graph, 157
- ungapped q-gram, 48, 142
- UngappedShape** (Spec), 48, 50, 51, 142
- UngappedXDrop** (Tag), 130
- unique trace, 164
- unstable, 62
- usability, *Sec 14.2*
  
- Value** (Meta), 36, 41, 42, 45, 55, 72
- value, 55
  - size, 60
  - type, 28, 36, 42, 47, 55
- value** (Func), 41, 44, 45, 65, 77
- valueById** (Func), 77
- ValueSize** (Meta), 37, 41, 43, 45, 59
- vertex, 157
- VertexDescriptor** (Meta), 157
- VertexIterator** (Tag), 159
- view, 80
- view position, 79
- virtual function, 32
  
- Waterman-Eggert algorithm, 128,  
     *Alg 20*
- weight

- of a seed, 95
- of a shape, 142
- weight** (Func), 95
- wildcard searching, 121, *Sec 10.4.2*
  - syntax, 123, *Tab 16*
- WildShifAnd** (Spec), 121
- WordGraph** (Spec), 158
- write** (Func), 63, 64, 84
- Wu-Manber algorithm, *Sec 10.2.1*,  
113, *Alg 14*
- WuManber** (Spec), 111
  
- X-drop, 131
- X-drop extension, 131
  - gapped, 133, *Alg 23*
  - ungapped, 131, *Alg 22*
- X-drop extension, 7
  
- Zero** (Spec), 97–99
- zero terminated string, *see* C-style  
string
- ZOOPS** (Tag), 136