Master Thesis

Genomes per E-Mail

Efficient Compression of Biological Sequences

René Märker

14th June 2011

Supervisor:
Prof. Dr. Knut Reinert

Second advisor:
PD. Dr. Jürgen Kleffe

This work presents a useful data structure to manage multiple biological sequences both efficiently and generically. The objective is to create a data structure to aggregate similarities between biological sequences. The implemented method ensures that the contained sequences can be compressed while simultaneously the efficient analysis of multiple sequences exploiting parallel computing approaches is supported. The development of new techniques in both categories has a strong impact on the state of the art research in the field of genome research, especially since the current development and wide spread use of next generation sequencing is growing exceptionally fast.

In dieser Arbeit wird eine Datenstruktur zur effizienten und generischen Verwaltung von biologischen Sequenzen vorgestellt. Ziel dieser Arbeit ist es eine Datenstruktur zu erschaffen, die Ähnlichkeiten multipler biologischer Sequenzen aggregiert und so zum Einen die Größe der zu verwaltenden Sequenzen komprimiert und zum Anderen die effiziente Analyse multipler Sequenzen unter Ausnutzung von paralleler Berechnungen unterstützt. Die Entwicklung verbesserter Methoden in beiden Kategorien schafft einen bedeutenden Beitrag zur aktuellen Forschung, besonders da die gegenwärtige Entwicklung und ausgedehnte Benutzung von Sequenziertechnologien der zweiten Generation gewaltige Ausmaße annimmt.

I dedicate this thesis to my soul mate Yasmin Rahn, whose patience, understanding and persistent confidence in me, has brought me this far. I owe her for being unselfish and encouraging, especially during the time after our lovely son was born. Her unconditional love is the reason for the successful completion of this work.

I would like to convey my gratitude to all those who have contributed to this work.

Firstly I would like to record my gratitude to Prof. Dr. Knut Reinert and PD. Dr. Jürgen Kleffe for their supervision, advice and guidance.

I gratefully acknowledge David Weese for his advice, supervision, and crucial contributions.

Cordial thanks go in particular to Manuel Holtgrewe, Birte Kehr, Anne-Kathrin Emde, Martin Riese and Konrad Rudolph for their advice and their willingness to share their bright thoughts with me.

It is a pleasure for me to pay tribute to Kathrin Trappe, Lauren Jenkins, Jochen Singer and Felix Kellndorfer. I greatly benefited from their constructive comments on this thesis.

Both my families deserve a special mention. Firstly, I would like to pay sincere gratitude to my parents Jutta and Steffen Märker for their support and encouragement. I also would like to express my whole-hearted appreciation to both of my brothers André and Olaf Märker, who always cared about me and have been paragon siblings to me. Furthermore, I would like to convey my cordial gratefulness to family Rahn for continually being exceptionally supportive during the whole time.

Finally, I would like to thank everybody who has been of importance during the successful realization of this work.

Contents

List of Tables

List of Figures

Introduction

I

> “ ... our work had reached a climax with the DNA sequencing method and ... to continue would be something of an anticlimax.
>
> *(Frederick Sanger, 1988)*

The news of the last few weeks were dominated by a very small but nevertheless dangerous organism, enterohemorrhagic *Escherichia coli* (EHEC). The bacterium caused severe health problems and even killed dozens of people. This might come as a surprise since it is a strain of *Escherichia coli*, an organism commonly found in humans. What is the difference between EHEC and other members of the *Escherichia coli* family.

The answer is hidden in their genetic construction plan, the DNA (Deoxyribonucleic acid) which is the basis of all organisms. Following the instructions of a correct plan creates a functional, healthy organism. The DNA, often referred to as genome, consists of a sequence of bases. An insertion, deletion or the change of a single base into another can cause a totally different outcome of a gene product. If the plan contains errors, diseases, malfunctions or disorders may occur. For this reason analysing DNA is a crucial part of our scientific world.

The interest in DNA sequencing has led to an enormous development in different sequencing techniques which are sensitive enough to decode the genome of an organism. In addition to the increasing number in sequencing projects, newer high-throughput technologies were developed which generate massive amounts of data. Furthermore, the new techniques become more and more affordable leading to personalised genomes which further increases the amount of data to be handled. This development has led to a situation in which data storage has become a problem since buying more and more capacity is expensive and cannot be the solution.

From this point of view, it is obvious that efficient data structures and algorithms need to be developed to handle the already existing data and to fulfil the needs of the future.

This thesis will describe the development, implementation and results of a data structure with algorithms which can store and handle the genome of an organism in a fraction of its original size.

The next sections will describe the observations mentioned above in more detail and give an comprehensive overview.

1.1 A Short History of DNA Sequencing

The first crucial development in DNA sequencing was performed by Frederick Sanger. In 1978, Sanger *et al.* [1978] published the first genome sequence of the bacteriophage $\Phi X174$ with $5,375$ nucleotides by using his “Plus and Minus” sequencing method [Sanger and Coulson, 1975].

In the 1980's he invented the *Didesoxy sequencing* method – more commonly known as *Sanger sequencing* method – [Sanger *et al.*, 1977]. Sanger was able to increase the length and the velocity of sequencing. His pioneer work in decoding genomic sequences laid the foundation for the current state of the art in genomics.

Since then the sequencing technology has improved continually and remarkable projects have been accomplished characterising entire genomes of different organisms (see I.1. The average length of the sequenced genomes increased consistently from $5,368$ base pairs of the Bacteriophage $\Phi X174$ in 1978, to almost $3.3 * 10^9$ base pairs of the Homo Sapiens in 2001 [International Human Genome Consortium, 2001].

Table I.1: A time line of first published genome sequences of different organisms over the past thirty years.[Sanger *et al.*, 1978],[Sanger *et al.*, 1982],[Massung *et al.*, 1993],[Goffeau *et al.*, 1996],[Blattner *et al.*, 1997],[Adams *et al.*, 2000],[International Human Genome Consortium, 2001],[Gregory *et al.*, 2002]

Year (pbl.)	Organism	Genome size
1978	Bacteriophage $\Phi X174$	$5,368$ base pairs
1982	Bacteriophage Λ	$48,502$ base pairs
1993	Smallpox (major strain Bangladesh-1975)	$186,102$ base pairs
1996	Saccharomyces cerevisia	$12,495,682$ base pairs
1997	Escherichia coli $K-12$	$4,639,221$ base pairs
2000	Drosphila Melanogaster	$122,653,977$ base pairs
2001	Homo Sapiens	$3.3 * 10^9$ base pairs
2002	Mouse	$3.4 * 10^9$ base pairs

One of the biggest achievements was the complete characterisation of the first human genome sequence in 2001 [International Human Genome Consortium, 2001]. Initially started in 1990, almost a decade after the Sanger-sequencing approach was invented, the Human Genome Project (HGP) announced to deliver the complete human genome sequence in 2005. As long ago as 2001 the HGP published a first draft covering about 83% of the entire human genome sequence. In 2005 the draft filled in a coverage of almost 92%. They used an advanced automated Sanger sequencing approach and maintained a high accuracy which allows only for one miscall in $1,000$ base pairs (bp). The HGP was a long-term project that was publicly funded with around $\$2.7$ billion[*]. Despite the high accuracy of the automated Sanger-sequencing technology, it was too expensive and time-consuming to become the standard for current biochemical and medical applications.

1.2 The Era of Massive Genome Sequencing

Since 2005, the sequencing technology experienced a fundamental shift away from the automated Sanger approach towards new technologies capable of sequencing entire genomes with dramatically lower costs and unprecedented higher throughput in less time. The new tech-

[*] http://www.genome.gov, received on 2011–03–30

nologies, also referred to as *next-generation sequencing* (NGS), constitute various strategies that rely on a combination of the modern biochemical procedures, sequencing and imaging [Metzker, 2009]. This had led to a new era of massive genome sequencing resulting in an immense drop in sequencing costs in 2008. The market prices declined exponentially to $0.32 for a mega base (Mb) and around $30,000 for an entire human genome in September 2010 [†](see figure I.1).

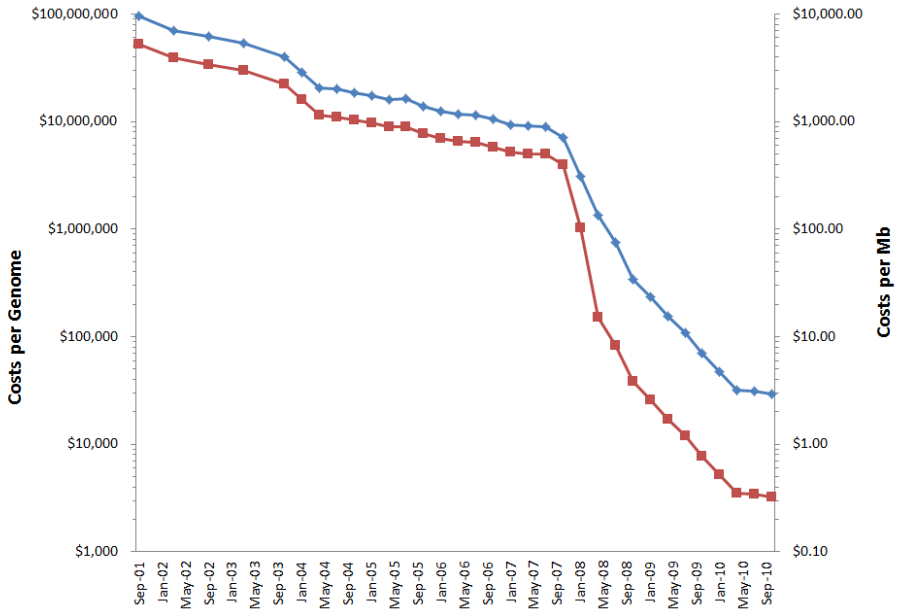


Figure I.1: Costs per genome and per Mb over the last decade. Note that on both y-axis the log-scale is used.

In 2008, the *1000 Genomes Project* was announced [1000 Genomes Project Consortium, 2010]. This venture is one of today's outstanding sequencing projects as it aims to produce the "most detailed map of human genetic variation" by sequencing the genomes of at least one thousand individuals around the world. It has become feasible solely due to major advancements in the technologies of various fields and associated with the immense reduction in sequencing costs.

But still sequencing technology has not reached its climax, yet. Recently, the first prototypes of the so called *nanopore–* or *single molecule third generation sequencing* techniques are emerging [Nan, 2010]. These systems are said to be even faster and more accurate as current NGS technologies [Nan, 2010]. The genetics community expects that the *$1,000-Genome* becomes available in only a few years [‡]. While the tremendous revolution of sequencing technologies is still ongoing, the statement of Frederick Sanger, back in 1988, that the sequencing technology has reached its climax, can be seen as rebutted.

[†]http://www.genome.gov/sequencingcosts/, received on 2011–04–26
[‡]http://www.nature.com/ng/qoty/index.html# stratton, received on 2011–03–31

1.3 Novel Bioinformatical Challenges

As explained above, high-throughput instruments sequence up to one billion bp in a single day at low costs, making large-scale sequencing more available than ever. Thus, bioinformatics has experienced an enormous boom while facing new challenges in analysing and organising the sequencing data. Typical applications for bioinformatics are whole genome mapping assemblies used in genome-wide association studies (GWAS's) trying to detect genomic variations, e.g., single nucleotide polymorphisms (SNP's) [Bentley *et al.*, 2008; Hillier *et al.*, 2008; Ley *et al.*, 2008], or structural variations [Chen *et al.*, 2008]. Further applications are RNA sequencing for expression profiling [Morin *et al.*, 2008] and chromatin-immunoprecipitation sequencing to identify protein binding sites [Barski *et al.*, 2007].

Yet, the above listed applications have focussed on the processing of single sequences only. Certainly, the revolutionary development of NGS has created the need to extend these problems to multiple sequences. Plenty of GWAS's are applied to reveal novel variations affecting the traits of versatile diseases. Such studies include patients with the same pathogenic phenotype. The sequenced genomes of these patients are interrogated to find potential risk factors responsible for the observed pathogenic phenotype. For all included subjects and all sequencing data the same algorithms and pipelines are applied repetitively to each sequence separately. But it is well-observed that individuals of the same population and species reveal only minor differences in their genetic code, because substantial genetic features have been highly conserved during evolution. The efficiency of the listed analysing algorithms could benefit greatly from approaches that exploit the high sequence similarity between individuals of the same species. Therefore, it would be sufficient enough to focus on the differences of each sequence to a common reference.

Another major advancement of such structures would be the reduced memory requirements to store the sequences. Currently, a deluge of sequencing data is produced by NGS applications threatening to swamp the available data archives. Figure I.2 shows the super-exponentially growth rate of such databases. The number of biological sequences is increasing dramatically since 2000 and has already exceeded the 1 million mark in 2007. Thus, novel space-saving sequence formats are dearly required not just to decrease the space consumption but also for economical reasons. Sequencing centres such as the NCBI in the USA or the EBI in the UK, spend millions of dollars to manage the annual data growth. With the emerge of *personalised genomics* this problem might spread out to smaller research institutions working with sequencing data.

Thesis Objectives

In this thesis I will extend the structure described in Pyl [2010]. The introduced approach promises great space saving potential since the amount of differences between two individuals of the same species is usually very low. In Pyl [2010] Pyl encodes similar and different regions between two sequences as a set of positions (see chapter III for details).

[1] http://www.ncbi.nlm.nih.gov, received on 2011–03–31

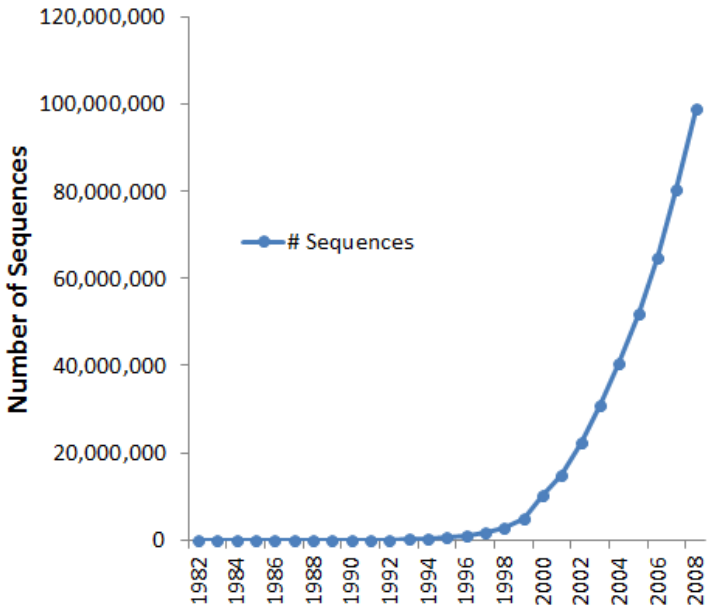


Figure I.2: Growth of GenBank. Shown is the annual growth of the sequence database GenBank since 1982. On the x-axis are remarked the numbers of sequences in 20,000,000 steps.[§]

This concept is realized in SeqAn, providing a data structure referred to as Journal-String that only focused on the efficient management of one sequence to a reference sequence. Journal-Strings have to be generated manually and one can not efficiently manage several sequences with regard to a single reference. I will overcome these shortcomings by:

- designing and implementing a method that automatically generates a Journal-String out of a sequence alignment and
- creating a container structure for the efficient handling of multiple Journal-Strings.

As the focus of this thesis is a space efficient storage of sequences, I will develop a new alignment score function. This score function will be optimized towards a space efficient storage of a Journal-String rather than an optimal similarity between two sequences.

Furthermore, I will develop different formats to store the generated Journal-Strings. These formats consists of a human readable format and a format which uses every single bit in order to obtain a maximal compression.

In addition to the space efficiency, the time efficiency is a crucial aspect of a data structure. I implemented the concept of skip lists which is a probabilistic approach with an optimal asymptotic runtime to increase the speed of handling Journal-Strings.

Besides the space and time efficiency of the data structure, it needs to be flexible to be useful. Reference sequences must be easily and efficiently exchangeable and it should be possible to merge whole sets of Journal-Strings. I will fulfil both requirements within the created data structure.

1.4 Outline

This thesis is divided into six chapters. The "Introduction" is followed by the chapter "Definitions & Fundamentals" in which work related terms and techniques are presented. The third chapter "Background & Related Work" describes the background of journaling edit transcripts exploiting a reference sequence as well as similar approaches that are related to this topic. The main achievements of this thesis are explained in detail in the chapter "Methods and Implementations", where solutions for the above mentioned issues are shown. These are the generation of a journaled sequence out of a pairwise sequence alignment, the detection of a maximal compressed pairwise sequence alignment, a novel approach to transfer multiple journaled sequences to a new reference sequence and the serialisation of entire sequences in a compact form. In the chapter "Results & Discussion" the performance of the implemented algorithms is analysed and the achieved compression factors are examined. Chapter VI, the "Conclusion", outlines the major findings of this work. It discusses some prospective extensions and advancements of the presented approaches and integrates them into the general research goal.

Supplementary materials can be found in the appendix. The corresponding paragraphs in the text are acknowledged accordingly.

II Definitions & Fundamentals

This chapter serves the better understanding of the technologies and methods used during this thesis. In the scope of this work three major issues are applied: the comparison of two strings, the compression of a string and the presentation of data within memory. Before each of the major issues is introduced and the essential items are described, the general term of a string and of the string alphabet is defined.

2.1 Strings & Alphabets

Definition 1 (String). In the following, DNA sequences, or more generally texts, are considered as strings over the finite ordered alphabet Σ. The set of all possible strings of an alphabet is denoted as Σ^* and ϵ is the empty string. A string s is a finite chain of n elements with $s_i \in \Sigma$, $\forall i \in [0, \dots, n)$. Note that the first element of a string is accessed at position 0. A substring of a string is an *infix*, with $s^I = [s_l, \dots, s_m)$, with $0 \leq l < m \leq n \; \forall s_i^I \in \Sigma$. Special derivations of the infix are the *prefix*, denoted as $s^P = [s_0, \dots, s_l)$, with $0 \leq l \leq n \; \forall s_i^P \in \Sigma$, and the *suffix*, denoted as $s^S = [s_m, \dots, s_n)$, with $0 \leq m \leq n \; \forall s_i^S \in \Sigma$. The length of a string s is denoted as $|s|$. In the following the letter r is reserved to denote the "reference string" explicitly.

Two primary alphabets are distinguished within this thesis. Σ denotes the set $\{A, C, G, T\}$ which refers to the four nucleotides of the DNA. $\Sigma^5 = \{A, C, G, T, N\}$ is a five letter alphabet containing the non-biological letter 'N' in addition. The letter 'N' is used whenever the nucleotide at a particular position cannot be determined with a certain accuracy. Strings of 'N's often occur in highly repetitive regions, like centromeres or telomeres, which are difficult to analyse with current technologies.

2.2 String Comparison

A measurement for the similarity, or in contrast, for the diversity of two strings is required by many biological applications. For example, one is interested in the question of how the string a became string b over time. Which operations were applied to convert a to b, and does a certain conserved region indicate an essential genetic feature To solve those questions a global alignment can be conducted on both sequences. The result is an edit transcript which decodes

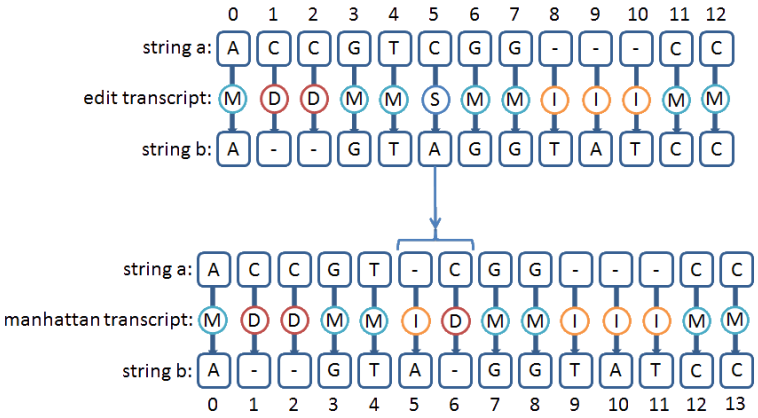


Figure II.1: Edit and manhattan transcript. Shown is the edit transcript between the two sequences $a = ACCGTCGGCC$ and $b = AGTAGGTATCC$. The fifth position of the edit transcript is a substitution. This is replaced by an insertion and a deletion in the manhattan transcript (position 5, 6).

the operations necessary to transform a to b. The terms, edit transcript and global alignment are defined as follows.

Edit Transcript

Definition 2 (Edit transcript). The edit transcript is a string over the alphabet $\Delta = \{M, S, I, D\}$. Usually the edit transcript refers to the Levensthein-Distance [Левенштейн (Levenshtein), 1965] which declares the number of edit operations that are necessary to transform one string into another. Such an operation can be a match (M), a substitution (S), an insertion (I) or a deletion (D) of an element. For two strings $a, b \in \Sigma^*$, a is transformed into b by applying the operations of the edit transcript one-by-one from left-to-right to each element separately. The edit transcript can be specialised by using the manhattan transcript. In that case the alphabet changes to $\Delta^M = \{M, I, D\}$, a subset of the edit transcript alphabet. Therefore the manhattan transcript the special issue of allowing only insertions and deletions are for denoting differences.

Global Alignment

The edit transcript of two strings is strongly related to the global alignment problem. Two strings can be compared on the basis of an alignment which identifies regions that are conserved, or changed. The possibilities to obtain such an alignment between two strings are manifold. One strategy is to compute the best global alignment by minimising or maximising the overall score of the edit transcript. This strategy requires a score function that assigns a score to each applied operation. This raises the question how the best global alignment can be found.

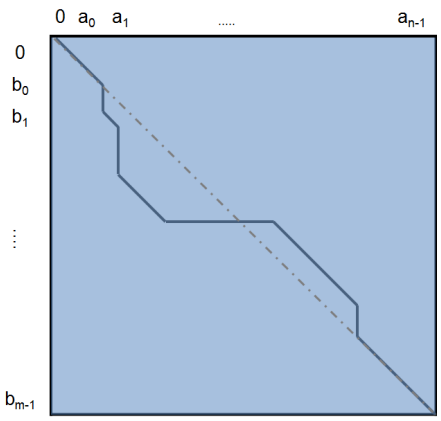


Figure II.2: Illustration of the Neddleman-Wunsch algorithm. The entire matrix is computed in order to determine the best global alignment.

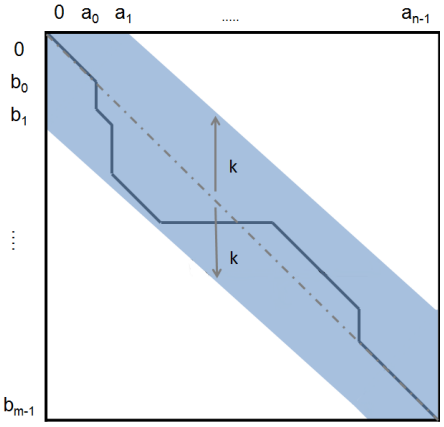


Figure II.3: Illustration of the k-band algorithm. Only the part defined by the k-band is computed.

Definition 3 (Best global alignment). Given two strings $a, b \in \Sigma^*$ with $|a| = n$ and $|b| = m$ and the additional character '$-$', to denote a gap within a string, a global alignment of a and b is a pair of strings (a', b') of length l with $l \geq max(m, n)$ over the alphabet $\Sigma' = \Sigma \cup \{-\}$ for which holds:

(a) $|a'| = |b'| = l$,

(b) there exists no position in the alignment at which a' as well as b' has a gap, such that $a'_i \neq - \vee b'_i \neq - \forall i \in [0, \ldots, l)$.

The pair of strings a' and b' is an equivalent representation of the edit transcript. They denote the changes necessary to transform string a to b and vice versa.

Let $\sigma(x, y)\ \forall x, y \in \Sigma'$ be a score function assigning a score to any possible combination of x and y. The score of an alignment (a', b') is the sum of scores of all columns of the alignment:

$$\sigma(a', b') = \sum_{i=0}^{l-1} \sigma(a'_i, b'_i).$$

The best global alignment $best(a', b')$ can be defined as

$$best(a', b') = min\{\sigma(a', b') | (a', b') \text{ is an alignment of a and b}\},$$

for all possible alignments (a', b').

Linear Gap Costs

Needleman and Wunsch [1970] introduced a dynamic programming approach to compute the best global alignment. This approach is based on organising two strings $a, b \in \Sigma^*$, with $|a| = n$ and $|b| = m$ into a matrix $M^{(n+1)\cdot(m+1)}$ and computing a score for every cell $M(i,j)$. Note that one additional row as well as one additional column is needed for the initialisation. The score of the active cell $M(i,j)$ is computed from the score of one of the preceding cells in the diagonal ($M(i-1,j-1)$), vertical ($M(i,j-1)$) or horizontal ($M(i-1,j)$) direction plus a particular penalty for the respective direction. The recursion is defined as:

$$M(i,j) = max \begin{cases} M(i-1,j-1) & +\sigma(a_i, b_j) \\ M(i,j-1) & +\gamma \\ M(i-1,j) & +\gamma \end{cases} \tag{II.1}$$

The value of the preceding cell is penalised by the score function $\sigma(a_i, b_j)$, with $a_i, b_j \in \Sigma$, $\forall i \in [0,n)$ and $\forall j \in [0,m)$. If two strings $a, b \in \Sigma^*$ are aligned then the edit operations are penalised as:

- $\sigma(a_i, b_i)$, with $a_i = b_i$ is a match from a to b,
- $\sigma(a_i, b_i)$, with $a_i \neq b_i$ is a substitution from a to b,
- $\gamma = \sigma(a_i, -)$ is a deletion from a to b,
- $\gamma = \sigma(-, b_i)$ is an insertion from a to b.

The best alignment can be obtained by tracing back the path that computes the maximal score starting at the last cell $M(n,m)$. In figure II.2 it is shown an abstract scenario of a global alignment. The blue area of the matrix indicates that all entries of the matrix are computed. The dark line through the matrix indicates the path with the maximal score and denotes the best alignment. The corresponding algorithm is represented in pseudo code in the appendix A.1.

If the aligned strings a and b are similar, then the best global alignment will be oriented near the main diagonal of the alignment matrix. Thus, it is not required to compute the entire matrix. In this cases, the use of a *k-banded alignment* is a more efficient solution. The main idea is to compute only a small area around the diagonal of M instead of the entire matrix. Figure II.3 demonstrates this approach with an abstract example. Again the blue marked area highlights the computed cells of the matrix. The best alignment is still located within the band. The corresponding algorithm is described in A.2 and a modification of it to allow for automatically increasing k is explained in A.3 in the appendix A.

Following the trace back of the best global alignment reveals the edit transcript. The relation between alignment and transcript is illustrated in figures II.4 and II.5. In order to obtain a manhattan transcript from the trace back of an alignment the following property must be set to σ:

$$\sigma(a_i, b_j) = -\infty \text{ , iff } a_i \neq b_j \forall i \in [0,n) \text{ and } \forall j \in [0,m),$$

which expresses that no substitutions are allowed.

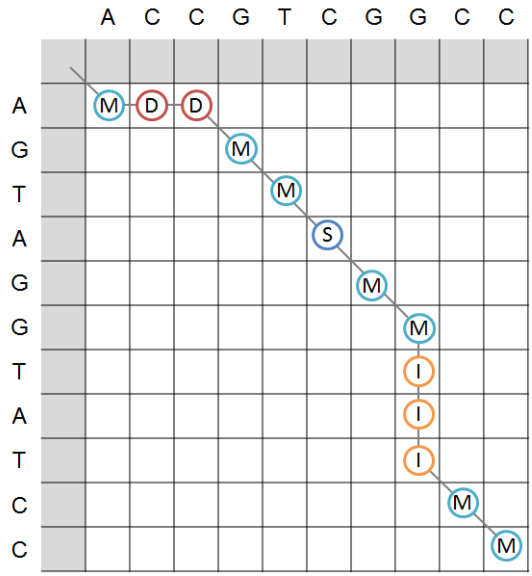


Figure II.4: Relation between best global alignment and edit transcript. The grey marked fields are used for the initialisation.

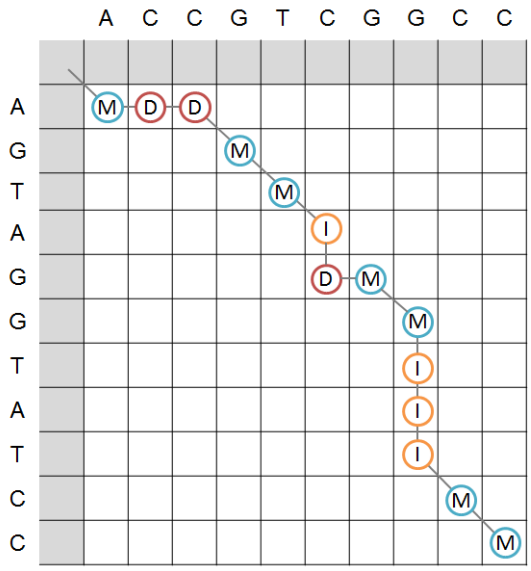


Figure II.5: Relation between best global alignment and manhattan transcript. The grey marked fields are used for the initialisation.

Affine Gap Costs

In addition to the above discussed linear gap function, one can use an arbitrary gap penalty also called affine gap costs. As opposed to linear gap costs, the affine gap function assigns different penalties for opening (o) and extending (e) a gap. Assume that $\gamma(l) = l \cdot \sigma(x_i, -) = l \cdot \sigma(-, y_i)$ is a function that computes the score of l consecutive gaps in either horizontal or vertical direction. Then the affine gap cost function is defined as follows:

$$\gamma(l) = o + (l - 1) \cdot e. \tag{II.2}$$

Figure II.6 illustrates the general issue of affine gap costs. The matrix shows which cells are considered, if the active cell (marked red) with the coordinates (i, j) is computed. For a linear gap function only the direct adjacent cells in the vertical, horizontal and diagonal line (marked dark blue) are considered, while for the affine gap function in addition all cells in the horizontal and the vertical direction, from the beginning of the matrix to the active cell, are considered (marked light blue). The latter refers to the fact that an insertion or deletion could be extended from any position before within the alignment. The best alignment approach for affine gap costs permutes all possible start positions for horizontal and vertical gaps. Otherwise it could miss the best alignment. Gotoh [1982] introduced a dynamic approach exploiting two additional auxiliary tables $D^{(n+1)\cdot(m+1)}, I^{(n+1)\cdot(m+1)}$ to obtain the best global alignment using affine gap costs.

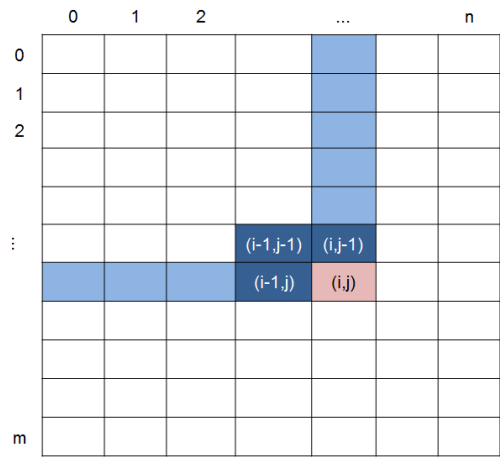


Figure II.6: Comparison of affine and linear gap costs. The active cell is highlighted red. The dark blue cells mark the cells considered for the linear gap costs. The light blue cells mark the cells additionally considered for the affine gap costs.

The initialisation formula expands to:

$$\begin{aligned} M(0,0) &= D(0,0) = I(0,0) = -\infty \\ M(i,0) &= D(i,0) = o + (i-1)\cdot e, I(i,0) = -\infty \text{ for } i = 1,\dots,n \\ M(0,j) &= I(0,j) = o + (j-1)\cdot e, D(0,j) = \infty \text{ for } j = 1,\dots,m, \end{aligned} \tag{II.3}$$

and the recursion formula to:

$$M(i,j) = max\begin{cases} M(i-1,j-1) & +\sigma(a_i,b_j) \\ D(i-1,j-1) & +\sigma(a_i,b_j) \\ I(i-1,j-1) & +\sigma(a_i,b_j) \end{cases} \tag{II.4}$$

$$D(i,j) = max\begin{cases} M(i-1,j) & -o \\ D(i-1,j) & -e \end{cases} \tag{II.5}$$

$$I(i,j) = max\begin{cases} M(i,j-1) & -o \\ I(i,j-1) & -e \end{cases} \tag{II.6}$$

Then

- $M(i,j)$ is the best score up to (i,j), given that a_i is aligned to b_j,
- $D(i,j)$ is the best score up to (i,j), given that a_i is aligned to a gap and
- $I(i,j)$ is the best score up to (i,j), given that b_j is aligned to a gap.

The dynamic programming approach has a worst-case runtime of $O(n \cdot m)$ and in the case of a k-banded alignment of $O(\Delta \cdot n)$ (see proof in appendix A.1). The presented alignment techniques are fundamental approaches applicable for short sequences. They are not suitable for genomic sequences, because of the quadratic runtime and space consumption. There exists more complex approaches for the purpose of aligning two genomic sequences. The Basic Local Alignment Tool (BLAST) developed by Altschul *et al.* [1990] is a well-known member of these alignment approaches. Note that this work designs and develops a novel data structure and algorithmic approaches from scratch. It lays down the foundation for future applications in next-generation sequencing. Thus, it is tested first for the fundamental approaches described above. Afterwards, the developed structure and algorithms can be extended with more complex approaches like BLAST.

> “I have made this letter longer than usual because I lack the time to make it shorter.
>
> *(Blaise Pascal; 1623–1662)*

2.3 Data Compression

Next-generation sequencing results in the massive generation of sequencing data which challenges the available storage capacities. Despite the availability of cheaper and higher capacity hard drives, the management of such huge datasets is an enormous problem. An intuitive way to solve this is to compress the sequence data. In order to do so it is necessary to understand the principles of data compression and the topic related terms.

Notations

Definition 4 (Data Compression). Let a be a non-random string, with $a_i \in \Sigma, \forall i \in [0, m)$. The string a can be compressed by removing redundancy within the string. The inverse operation of the compression is the decompression.

Basically, compression means to remove *redundancy* from the original data in the input. Any non-random collection data has a structure, and this structure can be exploited to achieve a smaller representation of the data.

The redundancy of a string is determined and removed by the *compressor*. A *decompressor* can reconstruct the original string by rewinding the redundancy. A Compression can be grouped into two approaches which are defined as follows:

Definition 5 (Lossless & lossy compression). *Lossless* compression methods only remove redundancy from the original string that the decompressor can reconstruct. Compressors ordinarily achieve better compression factors with *lossy* compression by losing some information which can not be reconstructed by the decompressor.

Lossy compression is often used to compress videos, images or sounds, where information is erased that does not change the data much. For example, in the case of compressing sounds, imperceptible frequencies are removed, although this might lead to some artefacts within the data [Storer, 1988; Salomon, 2000b; Sayood, 2003]. To compress DNA sequences lossy compression is impractical, since the slightest modification that can not be correctly reconstructed by the decompressor could adulterate the sequence information.

The performance of a compression can be measured with the following commonly used quantity:

Definition 6 (Compression Factor). The *compression factor* is defined as:

$$\text{compression factor} = \frac{\text{size of input string}}{\text{size of output string}}. \tag{II.7}$$

A compression factor of 2.0 indicates that the output string occupies 50% of the original string size after compression. Apart from achieving high compression factors, the time that the compressor needs should not overcome the benefit of the compression itself. The time factor for compressing and decompressing depends on the usage of the data. Some data needs to be used more frequently than other, makings long running times for encoding and decoding inappropriate. Faster algorithms might be better suited, even if they achieve only smaller compression factors.

Shannon's Entropy

The expected value of the information contained in a string can be quantified by applying the *Shannon's entropy* [Shannon, 1948]. This is the formal description of *entropy* in *information theory* and represents the limit of the best possible lossless compression [Salomon, 2000a].

Definition 7 (Entropy). Entropy is a measure of the uncertainty associated with a random variable. Given a discrete random variable X that can take a finite number of values $x_1, \dots, x_n$ with probabilities $p_i \geq 0 \; \forall i \in [1, \dots, n]$ and $\Sigma_{i=1}^{n}(p_i) = 1$, then $h(p)$ denotes the uncertainty associated with the random event $X = x_i$ and $i \in [1, \dots, n]$. Let H_n be a function over n variables $p_1, \dots, p_n$, then the average uncertainty can be measured with

$$H_n(p_1, p_2, \dots, p_n) = \sum_{i=1}^{n} p_i \cdot h(p_i). \tag{II.8}$$

Mathai and Rathie [1975] alternatively characterised the function in II.8 as

$$H_n(p_1, p_2, \dots, p_n) = -\sum_{i=1}^{n} p_i \cdot \log_b(p_i). \tag{II.9}$$

which is famously used as *Shannon's Entropy* or the *Measure of Uncertainty*. In the context of information theory b is usually set to 2. Hence the entropy is specified in bits per character (bpc).

For example, the entropy of a series of coin tosses with a fair coin has a maximum entropy. The possible outcome of a coin toss is either head or tail and there is no possibility to predict the next outcome. However, the same series of tosses with a two-headed coin has an entropy of zero, since the outcome of the next toss can only be a head.

Single Genome Compression

A lot of research has gone into developing good lossless compressors for the special purpose of compressing DNA sequences. There are several well-known compression tools, including BioCompress [Grumbach and Tahi], BioCompress-2 [Grumbach and Tahi, 1994], GenCompress [Chen *et al.*, 2001], the CTW+LZ algorithm [Matsumoto *et al.*, 2000], and DNACompress [Chen *et al.*, 2002].

Each of the mentioned compression tools addresses the removing of redundancy due to repetitions. Usually, a DNA sequence consists of several repetitive patterns, i.e. *tandem repeats* [Rivals *et al.*, 1997; Curnow and Kirkwood, 1989]. These are groups of one or more nucleotides that appear repeatedly and are directly adjacent to each other.

It also has been proven that many essential genes occur in several copies, due to duplication events during evolution [Gardner *et al.*, 1991]. This redundancy in genomes can be compressed very efficiently [Grumbach and Tahi, 1994; Chen *et al.*, 2001; Matsumoto *et al.*, 2000; Chen *et al.*, 2002].

In general, compressing DNA sequences is a very difficult task and compressors require a lot of computational effort to determine the redundancy [Grumbach and Tahi, 1994].

Table C.1 in the appendix showed that on average DNACompress revealed the best compression results with almost 1.7254 bpc. Assuming a complete randomly generated DNA sequence, then the absolute limit of lossless compression is 2 bpc. The DNA alphabet consists of 4 letters, such that only 2 bits are necessary to encode each character. Thus, the compression gain of DNACompress is relatively small compared to the maximal entropy of 2 bpc.

Genome Compression Exploiting a Reference Sequence

The modest compression factor gained by single genome compression algorithms makes them ineligible for integrating them in analysis processes of NGS applications. Another approach exploits a reference string as an anchor for the compression. The idea is to store only the differences to a reference string.

Since it is widely-known that the genomes of organisms of the same species are very similar, this compression method is very efficient. The International Human Genome Consortium [2001] revealed an identity grade of 99.9% of two human beings. Even though recent studies[*], it remains a fact that the entropy between two genomes of the same species is very small. Furthermore, this particular compression method is supported by offering 2,800 reference genome assemblies for a wide range of organisms on the NCBI project websites[†]. These

[*] http://www.independent.co.uk revealed human genomes may be more like 99.0% identical, received on 2011–04–07

[†] http://www.ncbi.nlm.nih.gov, received on 2011–04–01

reference assemblies are well-suited to transmit the compressed sequences. The receiver of compressed data sets has easy access to the reference sequence that was used as an anchor for the compression.

Christley *et al.* [2009] compressed the size of the James Watson genome from 3,169,831 Kb to 4,101 Kb exploiting a reference sequence and applying some value transformations on the variation positions. Their compression approach revealed an entropy of 0.0013 bpc which implies a compression factor of almost 773. This is 154 times higher than the compression factor achieved by the single genome compression.

2.4 Memory Representation

In order to allow for transmitting the compressed sequencing data, additional export and import methods are required. Therefore, some potential risk factors that could result in corrupted data must be considered. Such risk factors can occur, if the transmitter uses different hardware, than the receiver of the data. More specifically, the representation of the data in secondary memory depends on the currently used CPU architecture. In the following the major issues of *data structure alignment* and *endianess* are discussed in more detail.

Data Structure Alignment

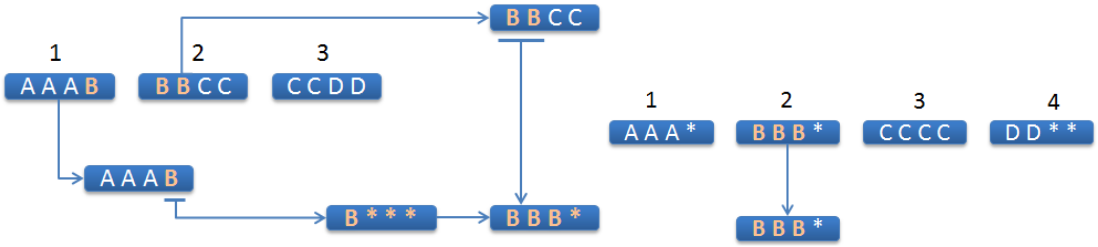
In computing, a *word* refers is to the natural unit of data used by a particular processor family[‡]. The word length indicates the number of bits addressed by a certain processor architecture. The x86-processor family, for instance, uses a chunk of four bytes (32 bits) as the natural unit. Data that is read or written from memory can only be addressed in such word sized chunks. This behaviour increases the system's performance due to the way the CPU handles memory. Hence, the system must ensure that data is put at memory offsets at a multiple of the word size. This is referred to as *data alignment* [Bryant and O'Hallaron, 2010].

Definition 8 (Data alignment). A memory address x, is said to be n-byte aligned when n is a power of two and x is a multiple of n bytes. In this context a byte is the smallest unit of memory access, i.e. each memory address specifies a different byte. A n-byte aligned address would have $log_2(n)$ least-significant zeros when expressed in binary.

Since it is allowed to define data structures of any size it could happen that a data structure has a size which is not a power of two. A second data structure would then not begin at an offset of multiple of four bytes. Hence, additional calculations are required by the CPU in order to access the correct data structure or even worst an alignment fault could occur. To circumvent the fault some meaningless bytes are inserted between both data structures, such that the second structure is correctly aligned [Bryant and O'Hallaron, 2010]. This is referred to as *data padding*.

Assuming there are four different data structures 'A','B','C' and 'D'. Further let 'A' and 'B' have a size of three bytes each, 'C' a size of four bytes and 'D' a size of two bytes. A misaligned

[‡]http://en.wikipedia.org/wiki/Word_(computer_ architecture), received on 2011-06-08



(a) Misaligned data structures (b) Aligned data structures

Figure II.7: Comparison of data alignment. Shown are four data structures A,B,C and D which require different data lengths (A = B = 3 bytes; C = 4 bytes; D = 2 bytes). The left figure shows a misaligned representation of the four data structures packed in 3 word chunks. In the right figure, the four data structures are aligned at a multiple of the word size due to data padding. Here, an additional word is required to store the data structures.

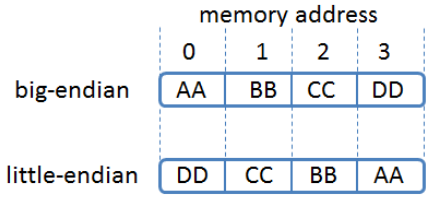


Figure II.8: Endianness. Shown is the byte order of the number AA BB CC DD on a big-endian and on a little-endian machine in memory. AA is the most significant byte and DD is the least significant byte.

representation of the four structures is illustrated in II.7a. The word size in this example is four bytes, symbolised by blue boxes. Since the structures are misaligned, one byte of 'B' is stored in the first word and the remaining two bytes are stored in the second word. In this state the computer cannot process a request to 'B'. First the chunks that hold a part of 'B' are identified. Second, the first byte of 'B' from the first word is extracted and put into a new word. Afterwords the remaining two bytes of 'B' are extracted from the second word and subsequently appended in the new word. In this new state the computer can process requests to 'B'. In contrast figure II.7b illustrates the same scenario using data padding. In that case meaningless bytes, symbolised as $*$ are inserted to align the structures to a multiple of the word size. The CPU can access the structure 'B' directly, and no additional calculations are required.

Endianness

Another problem when exporting the data to secondary memory arises with the different representations of words on different CPU architectures. More specifically, the byte order of a word, also called the *endianness*, can be either big-endian or little-endian. It determines whether the most-significant byte or the least-significant byte is ordered first within a word.

Figure II.8 represents the number $2,864,434,397$ as a hexadecimal number (AA BB CC DD) in a word on a big-endian and a little-endian CPU architecture. There are four columns. Each column represents one byte within the word. Then the hexadecimal number 'AA BB CC DD' is byte wise written to the word, while depending on the CPU architecture the 'AA' is assigned to the first byte of the word, if it is a big-endian machine or assigned to the last byte if it is a little-endian machine.

Most network protocols, for instance, are big-endian in the sense that the most significant byte is sent first. As a result big-endian is often referred to as network-byte order. In contrast, most host systems are little-endian, also called the host-byte order [Bryant and O'Hallaron, 2010].

Background & Related Work

III

In this chapter the fundamental principle of storing only the differences between two strings is introduced. Therefore, the recently developed approach of data journaling for biological sequences by Pyl [2010] was used. He suggested encoding the differences as nodes of a tree. The nodes are classified in three types: insertion-node, deletion-node and the original-node. The node-types encode an infix of the same manhattan operations. This particular infix is specified by three auxiliary values stored within the node. The encoded string can be decoded by applying the operations stored within the tree. In this piece of work, a modified version of this concept was used. The modification is explained in more detail in the subsequent section. In the second part of this chapter, related data structures which are capable of aggregating similar sequence intervals of multiple sequences are introduced.

3.1 Data Journaling

It has already been illustrated in the previous chapter that storing only the differences of a biological sequence to a predefined reference string could enormously reduce the memory requirements. However, the underlying data structure must allow for a swift random access to the elements of the encoded string.

As described above, the concept of data journaling for biological sequences was exploited. In the follow up of this piece of work, this data structure is referred to as *Journal-String*. The Journal-String was modified by means of using only two node-types to specify whether an infix refers to the reference string or to an inserted region. The deletions are encoded by a void within the positions of two adjacent nodes which refer to the reference string. This technique is explained more specifically after the formal definition.

Definition 9 (Journal-String). A Journal-String j is a string encoded by its differences to a reference r. It contains the following information:

- R^j, the reference string of j,
- I^j, the insertion string of j and
- a tree T^j organizing the journaled operations of j.

R^j is the reference string to which the differences are managed. Inserted elements are stored in an additional buffer I^j. Both R^j and I^j are also referred to as the sources of j. A journaled operation is an encoding of a certain infix of j. It is represented as a *Journal-Entry* which stores the following information for an infix:

- the segment source (ss) $\in [0, 1]$, indicating if the underlying infix is from $R^j \Rightarrow 0$ or from $i^j \Rightarrow 1$,
- the physical position (pp) of the begin of the underlying infix,
- the virtual position(vp) of the begin of the underlying infix,
- the length (l) of the underlying infix.

The length of a Journal-String is denoted as $|j|$. ϵ is the empty Journal-String. The k-th position of a Journal-String is denoted as j_k, with $k \in [0, \dots, |j|)$. The number of Journal-Entries is denoted as $|T^j|$. The v-th Journal-Entry of a Journal-String j can be addressed by T^j_v, with $v \in [0, \dots, |T^j|)$. For any two Journal-Entries T^j_x and T^j_y it holds the relation $T^j_x < T^j_y$ iff the virtual position of T^j_x is less than the virtual position of T^j_y $\forall\ x, y \in [0, \dots, |T^j|)$. This also implies that $x < y$.

Furthermore it holds for two adjacent Journal-Entries T^j_{p-1} and T^j_p that the virtual position of T^j_p equals the sum of the virtual position and the length of T^j_{p-1}, $\forall p \in [1, \dots, |T^j|)$.

A Journal-Entry represents a journaled operation in form of four parameters which are shortly described as $ss(pp, vp, l)$ subsequently of this work. The ss value (segment source) before the brackets denotes the underlying source of the current operation. The first value of the triple is the physical position. It refers to the beginning position of the infix within the corresponding source. The virtual position, at the second position of the triple, denotes the absolute position of j. It represents the position of an element after all operations within j left of this position are considered. The physical positions are necessary to map the virtual positions to the correct element within the respective sources. Figure III.2 demonstrates this mapping for three particular cases. Finally, the length of the infix describes the considered range of the encoded operation. How the pairwise sequence alignment showed in figure II.5 can be expressed as a Journal-String and how the random access works for such structures are explained in the following paragraphs.

Encoding of Alignments

Figure III.1 illustrates the encoding of the differences between two strings. In this particular scenario the alignment of both sequences shown in figure II.5 is expressed by the Journal-String j. The horizontal sequence of the alignment ($ACCGTCGGCC$) is taken as the reference r. The vertical sequence ($AGTAGGTATCC$) is then represented as the Journal-Strixng to r.

In the initial state of j, r is assigned to the source R^j and I^j is empty. In this state the tree holding the Journal-Entries consists only of one node T^j_0 which covers the entire source R^j.

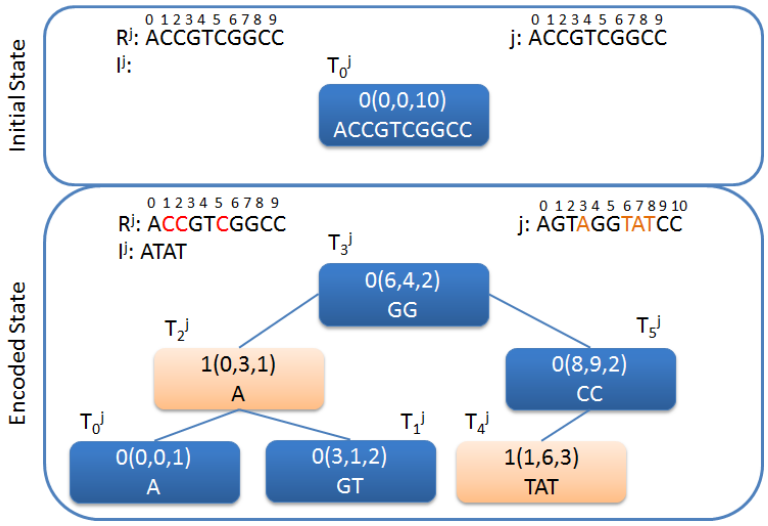


Figure III.1: Encoding within a Journal-String. Shown are the components of a Journal-String j. In the upper left corner of each frame is displayed the underlying reference and insertion source (R^j, I^j). On the upper right corner the decoded string is shown. The Journal-Entries are organised in a tree. Blue nodes indicate the type-0 entries and light orange nodes indicate the type-1 entries. Note that in each node the infix covered by the current node is displayed only for visualisation reasons.

T_0^j meets the following settings: $0(0, 0, 10)$ which means that the physical position as well as the virtual position start at an offset of 0, while the covered infix of the segment source 0 has a length of 10. In this situation decoding the Journal-String j would reveal the sequence of r. The decoded string j is displayed in the upper right border of the frames in figure III.1. A decoded string by means of a Journal-String refers to the string that is obtained if all operations stored in the tree are applied to the source R^j. A decoded Journal-String is also called *flattened* which refers to an tree containing one node covering the entire source R^j.

The entire Journal-String representing the manhattan transcript of the given alignment can be seen in the second frame "Encoded State". It stores the operations necessary to transform the string r into the decoded string j. The virtually deleted positions within r are marked as red letters in R^j. The virtually inserted elements are marked as orange letters in j. Additionally I^j stores all inserted letters consecutively. Assume that the represented tree is a balanced binary-search tree. Note that the underlying data structure to store the journal operations is the essential backbone of the Journal-String. It must provide for fast search, insert and delete operations, unless a fast random access is not required. Chapter IV addresses different implementations for this data structure and explains more specifically how the basic operations are performed. For the understanding of the journaling concept it is sufficient enough to know that the operations are stored in an ascending order by their virtual positions, independent of the underlying data structure.

In the following encoding the alignment, shown in figure II.5, by a Journal-String is demonstrated. The deletion at position 1 of length 2 in r is the first journaled operation of the

alignment. As mentioned before, a deletion is not directly encoded by a Journal-Entry. Instead it is represented as a void between the last physical position of the Journal-Entry covering the part before the deletion and the physical position of the Journal-Entry covering the part after the deletion. In this example, this void is represented between the node T_0^j, with $0(0,0,1)$ and T_1^j, with $0(3,1,2)$. The first node covers an infix of length 1, beginning at position 0 of R^j. It is obvious to see that the first node also starts at the virtual position 0. The virtual position of T_1^j continues at the last virtual position of the predecessor at position 1, according to the definition. The physical position continues after a right shift of two positions at position 3 which corresponds exactly to the deletion length. Hence, the positions 1 and 2 of R^j are no longer covered by the Journal-String. The corresponding values of R^j are marked red.

After a matching region of length 2 ('GT') it follows an insertion of length 1 (T_2^j, with $1(0,3,1)$) marked as an orange node. Again the virtual position of T_2^j continues at the last virtual position of the predecessor (T_1^j). The important difference to the case described before is the physical position of this Journal-Entry. It is set to 0, because it refers to the source I^j now. In this case the corresponding infix starts at position 0. After the insertion, a deletion of length 1 follows. This, again is represented as a void within the physical positions of the current node T_3^j and T_1^j.

After the matching region encoded by T_3^j, an insertion of length 3 appeared. The corresponding infix is represented by the Journal-Entry T_4^j, with $1(1,6,3)$. In the end the last two matching elements of the alignment are again encoded as a Journal-Entry covering the position 8 and 9 of R^j. Now, the Journal-String j encodes the entire alignment.

Random Access

The access to the position specific content of a Journal-String is changed due to the encapsulation of the modified string onto another abstraction layer. While in the usual scenario, as described in section 2.1, the elements could be directly accessed through the string by their position, the elements in a Journal-String may be shifted and thus refer to different positions. For a Journal-String j an element can only be accessed by its virtual position. The virtual position represents the actual string after applying all modifications left of the requested position. In order to obtain this element, the virtual position must be mapped to the physical position of the corresponding source (R^j or I^j).

Figure III.2 illustrates three cases of accessing an element via its virtual position. All three cases show the access of a virtual position within another context. The virtual position in case a) points to an element that actually is deleted in R^j. In case b) an element of an inserted region is accessed and case c) shows the access of an element within an unmodified region. In addition there are coloured arrows showing the trace of accessing an element via T^j, and a corresponding grey dashed arrow indicating the direct way for a non-journaled sequence.

Case a) demonstrates a request of an element at position 15. Since there was a deletion in the range $[10,25)$, all elements right of 24 are shifted to the left by the offset of this particular deletion. The virtual positions of the journal entries right of this deletion are shifted with

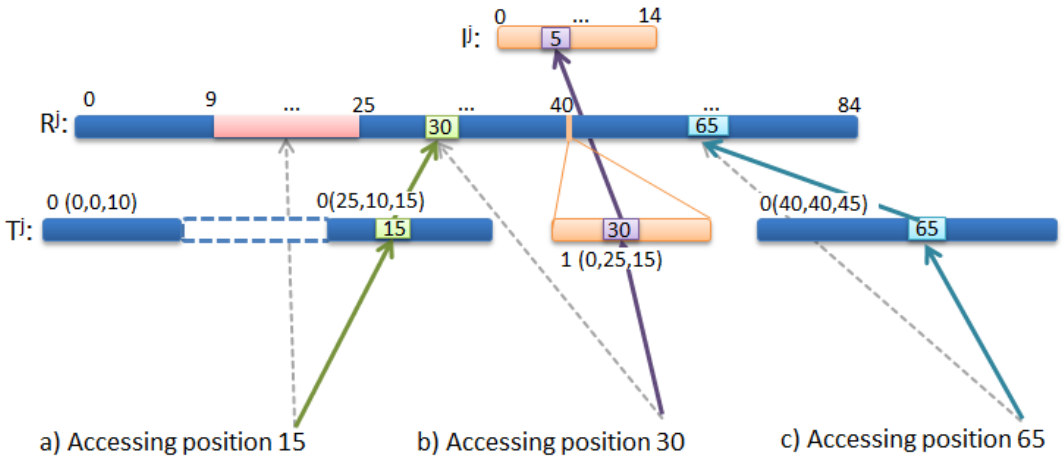


Figure III.2: Accessing elements of a Journaled-String. Shown is an abstract Journal-String j with an insertion string I^j, a reference string R^j and a tree T^j. j encodes a deletion at position 10 of length 15, and an insertion of length 15 at position 25. The deleted area within R^j is highlighted with a red box. The dashed area between the first and the second Journal-Entry indicates the void between their physical positions. The inserted region is marked with an orange fill. The small orange line within R^j, represents the insertion position. The cases a), b) and c) distinguish different results when accessing the corresponding positions of j. The coloured arrows indicate the way the element is accessed, while the grey dashed arrows indicate the direct access if the string would not be journaled.

respect to the offset, too. The region that is covered by the second journal entry (after the deletion), still represents the infix of the range $[25, 40)$. Thus, the accessed element at virtual position 15 maps to the 30-th physical position of R^j, implied by the green arrow tracing the way, accordingly. In contrast, the grey dashed arrow indicates the direct way, when trying to access the actual 15-th position of R^j which lies within the deletion.

In the second scenario, the element at position 30 lies within an inserted region. Instead of accessing the 30-th element of R^j, implied by the grey dashed error in case b), the 5-th physical position of I^j is accessed (purple arrows). Both the deletion and the insertion have a length of 15, such that they neutralize the shifts within the virtual positions. Hence, the last Journal-Entry of T^j has the same virtual positions as physical positions, such that the 65-th element of j is mapped one-on-one to R^j. Both the blue arrow and the grey dashed arrow show the way of how the element is accessed.

3.2 Aggregation of Multiple Sequences

The above discussed subject of journaling the manhattan transcript is focussed on a pair of strings. This thesis addresses the problem of managing multiple sequences by the usage of the Journal-String as a base structure. These multiple sequences share the same reference string, such that similarities between them are aggregated. This aggregation can greatly benefit the

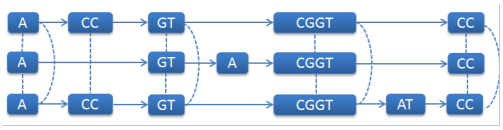


Figure III.3: Alignment graph.

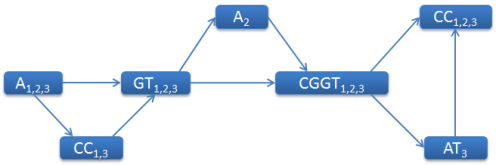


Figure III.4: PO graph.

parallel processing of all sequences and also suggests a good way for a space-saving representation of the strings. There are other approaches to encode similar parts within multiple sequences which are the *generalised suffix tree* [Bieganski *et al.*, 1994] and the *generalised suffix array* [Shei, 1996], the *alignment graph* [Rausch *et al.*, 2008] and the *PO-graph* [Lee *et al.*, 2002].

Generalised Suffix Tree

Known structures for aggregating sequence similarities between multiple sequences are the generalised suffix tree or the generalised suffix array. Since both refer to the same problem only the suffix tree is discussed in the following.

as a path through a acyclic directed graph. The number of leaves is equal to the number of suffixes within the string.

A suffix tree is a structure that represents each suffix of a string a of length n in a tree, such that the paths from the root to the leaves have a one-to-one relationship with the suffixes of a. Each edge is labelled with a non-empty string which is also a common prefix of the suffixes of its subtree. Usually the special character \$ is appended to a to ensure that no suffix is a prefix of another suffix. This also implies that there exists exactly n leaves, one for each suffix of a. The generalised suffix tree represents all suffixes of a set of sequences. Each vertex within the tree is labelled with the sequence numbers that share the particular infix read from the root.

Alignment Graph

Another – also graph based – approach to encode similarities within multiple sequences is the *alignment graph* or the closely related *PO graph*. In the following the three strings with $s_1 = ACCGTCGGTCC$, $s_2 = AGTACGGTCC$ and $s_3 = ACCGTCGGTATCC$ are considered. A possible collinear multiple sequence alignment (MSA) of the three strings is displayed in the table III.1: The alignment graph for k sequences is a k-partite graph. It extracts shared com-

Table III.1: Multiple Sequence Alignment of the three sequences s_1, s_2 and s_3

pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13
s_1:	A	C	C	G	T	-	C	G	G	T	-	-	C	C
s_2:	A	-	-	G	T	A	C	G	G	T	-	-	C	C
s_3:	A	C	C	G	T	-	C	G	G	T	A	T	C	C

ponents of all included sequences and aggregates them in vertices. The vertices are connected

linearly through directed edges from left to right. Additionally, vertices sharing the same infix are connected through undirected edges. They represent the conserved regions of the k strings. The figure III.3 depicts the alignment graph of the above displayed MSA.

PO Graph

In addition the PO graph represents the components of an alignment graph as nodes tagged with the sequences that share this information. It is obtained from an *A-Brujin graph* [Raphael *et al.*, 2004]. The components are connected through arcs indicating the linear trace of the alignment. Figure III.4 illustrates such an PO graph for the above given MSA and the constructed alignment graph.

This thesis proposes an alternative data structure to the above presented structures which can be used to aggregate similar sequence segments of multiple sequences. The main principle of the proposed data structure equals the concept of data journaling. In particular the proposed data structure can be used to store the differences from one string to a reference. Despite the fact, that the known data structures provide the aggregation of similar sequence parts, they have some major disadvantages regarding the dynamic update. Therefore it is difficult to apply modifications to them. For example adding a new sequence to the multiple sequence alignment, would cause the recomputation of it which is computationally expensive. The naive construction of a MSA for k sequences would be the extension of the pairwise alignment problem to a k-dimensional problem. This naive strategy takes $O(n^k)$ time, assuming all sequences have the same length n [Wang and Jiang, 1994]. A more practical approach exists by Lipman *et al.* [1989] which uses pairwise alignments to constrain the k-dimensional search space, but still the MSA remains inflexible to modifications.

Like the alignment graph, the generalized suffix tree is impractical for dynamical updates of the contained sequences. If a new sequence is added, each suffix of the sequence must be added to the tree, while the common paths must be identified. This takes $O(n * |\Sigma|)$ time [Bieganski *et al.*, 1994]. Furthermore inserting or deleting an infix of one of the sequences could cause the replacement of many subtrees, since new suffixes are created which must be considered for the suffix tree. Another disadvantage is the space consumption of suffix trees [Shei, 1996]. The described disadvantages arose due to the fact that the mentioned data structures were not designed to be modified once they are created. Instead, they were optimised towards their construction times. In general the suffix tree can be seen as a static database to which a plenty of queries can be applied. The tree can be used to find all z occurrences of a pattern p in $O(|p| + z)$ time which is asymptotically optimal [Bieganski *et al.*, 1994].

In this thesis the aggregation of similar string infixes of k strings is not managed globally by the data structure, but in each string separately. This encapsulation allows for a high dynamic use of the aggregated information, because a single string can be modified without affecting the remaining strings within the structure.

Methods & Implementation

IV

The abstract behaviour of a Journal-String was described in the previous chapter. The underlying data structure managing the Journal-Entries is the essential element within the Journal-String. It must support fast random access operations in order to allow for a quick navigation through the sequences. The following section discusses the differences between the implementations as a sorted array (SA), an unbalanced tree (UT) and a skip list (SL). Afterwards, the concept of a container for Journal-Strings is presented. This container efficiently manages multiple journaled strings. It collects Journal-Strings in groups which share the same reference. This section also focuses on essential interfaces required to dynamically manage the stored strings. In addition, algorithms were developed which are needed to extract a Journal-String out of an alignment. Furthermore, a score function which computes the smallest possible encoding for two strings in terms of memory was investigated. After that, a new approach to rapidly synchronise k Journal-Strings to a new reference string is explained in more detail. In the last part of this chapter, several ways to export and import data of the container are discussed. Therefore, some further compression techniques are applied in order to enable the transmission of the serialised data using standard communication tools like E-Mail or instant messenger.

All algorithms and data structures were implemented within the sequence analysing library *SeqAn* [Döring *et al.*, 2008]. Some specific program details are presented in the appendix B.

4.1 Data Structures used for Journaled Operations

There are already two existing implementations of a Journal-String. One uses a SA and the other one an UT to manage the Journal-Entries of the strings. Since a fast random access is not only desired but demanded, the structures should provide good runtimes for search, insert and delete operations performed on Journal-Strings. Table IV.1 shows the worst-case runtimes for each operation of a SA and an UT.

The sorted array, if using binary search (see algorithm A.4 in appendix), provides an optimal asymptotic search in $O(\log n)$ time [Knuth, 1998], with n being the number of Journal-Entries. The insert and delete runtimes suffer from the fact that if a certain position of the array is inserted or deleted, then all elements right of this position must be shifted one position to the right, or the left respectively, within memory. Hence, the runtime for the insertion and

deletion scenario is $O(n + \log n)$ which takes $\log n$ operations to find the respective position plus $O(n)$ to shift the elements.

The second implemented data structure, the unbalanced tree, has a worst-case runtime of $O(n)$ for each operation. The worst-case scenario for such a tree would be a linked list, in which each node has only one child. Then the entire list might be scanned before the searched element is found. The insertion or deletion itself can be applied in constant time, since only the relations to the adjacent nodes must be updated which are constant many. The pseudo-code A.5 in the appendix shows the algorithm for searching on a binary tree. There exists several self-balancing tree data structures that keep the depth of the tree at $O(\log n)$ [Knuth, 1998]. But the complex rebalancing operations often lack efficient practical implementations [Pugh, 1990].

A good alternative to the self-balancing trees was presented by Pugh [1990], who described the skip list. This data structure provides asymptotically optimal runtimes for each operation [Pugh, 1990]. Table IV.1 lists the runtimes for each operation. The skip list differs to other data structures because it is a probabilistic data structure. The general idea of this structure is to define h many linked lists for the elements that need to be organised. Each list stores a decreasing number of elements distributed by the probabilistic function $Pr(x)$, where $x \in S$ and S contains all elements plus the empty value $\emptyset$. Let

$$Pr(x) = \frac{1}{2}^{k} \tag{IV.1}$$

be the probabilistic function denoting that k lists contain x and let $S_1 = S$ being the list containing all elements. Further let S_h contains only $\emptyset$. Then the following invariant holds:

$$\emptyset = S_h \subseteq S_{h-1} \subseteq S_{h-2} \subseteq \ldots \subseteq S_2 \subseteq S_1 = S. \tag{IV.2}$$

The probabilistic function can be seen as a series of coin tosses for each element $x \in S$. If the head is consecutively tossed k times, then the lists $S_1, \ldots, S_k$ contain x. Furthermore, let each list S_k be a linked list and each element in S_k has a link to its occurrence in the previous list S_{k-1}. Then Pugh [1990] showed how to search in $O(\log n)$ steps, with $|S| = n$. In addition, an insertion and a deletion can be applied in constant time by performing constant many operations which results in the same runtimes of $O(\log n)$[*]. The search algorithm is explained in pseudo-code in the appendix A.6, accompanied with an illustration.

It has also been proven that each data structure requires theoretically only $O(n)$ space [Knuth, 1998; Pugh, 1990]. In practice this must be differentiated. The SA requires exactly $m \cdot n$ space, where m is the constant size of a Journal-Entry in memory. There are no further information necessary to store the elements. The UT, requires slightly more memory, since each Journal-Entry is linked with its adjacent entries. The links are implemented as pointers. The SL, is a special form of a double-linked list which means that each element requires additional memory to store the connections. This might not be a big difference but can be a crucial

[*]Note that the given running times are probabilistic bounds. Papadakis *et al.* [1990] proved exact bounds for the skip list which are close to the bounds of Pugh [1990]. Further proves the tail estimation with Chernoff bounds [Motwani and Raghavan, 1995] that the sum of t search operation remains optimal with $t * \log n$.

Table IV.1: Worst-case runtimes for the sorted array, the unbalanced tree and the skip list of size n for a search, insert and delete operation.

data structure	search	insert	delete
sorted array	$O(\log n)$	$O(n + \log n)$	$O(n + \log n)$
unbalanced tree	$O(n)$	$O(n)$	$O(n)$
skip list	$O(\log n)$	$O(\log n)$	$O(\log n)$

factor, if the purpose is to obtain the maximal compression. In this case a sorted array would be the better choice.

The search algorithms, explained in the appendix for each data structure, are modified towards the purpose of using Journal-Entries. Journal-Entries cover a range of elements, while usually only one element at a particular virtual position is searched. Hence, the search finds the entry which covers the searched position. This behaviour also implies that different cases for insert and delete operations must be considered, resulting in a shift of the found entry or even an additional split. The different behaviours considered for an insertion are illustrated in figure IV.7 and compared between each data structure.

Figure IV.7 is divided into two columns. The left column illustrates the shift of the Journal-Entries after a new Journal-Entry with a virtual position of 30 and length 10 is inserted. The new element is marked with an orange fill. Note that the Journal-Entries are displayed as abstract boxes which are labelled with fictive virtual positions. The physical positions are omitted as they are not changed by the insertion of an element. Merely, the inserted element is assigned with a new physical position which continues at the last physical position of the insertion buffer. The white filled box with a blue outline marks the found element by the preceding search process. Since the elements of a Journal-String are encoded as intervals of positions, the virtual positions of all Journal-Entries after the insertion must be updated. This process is indicated by the red arrow and applies to all data structures.

The right column demonstrates the special case of splitting a Journal-Entry while inserting a Journal-Entry at position 25. The affected Journal-Entry is split into a Journal-Entry covering the first 5 positions (white box with blue outline) and a second Journal-Entry covering the remaining 5 positions (box labelled with 35). The blue outline of the additionally inserted element indicates that the covered infix continues at the last physical position of the found entry after the split. Again the virtual positions of all entries occurring behind the insertion are shifted by this length.

Figure IV.7 also elucidates different operations necessary to keep the structures in an ordered condition. For the sorted array, this includes the shift of all elements after the insertion in memory. For an unbalanced tree only the links of the entries neighbouring the inserted entry must be updated (red links). It is easy to see that this is only a constant number of links. Basically, the same holds for skip lists. Even though the number of links depends on the tossed height for the inserted entry (compare the height of the inserted entries in the left and the right column), the probabilistic bound for the height of a skip list [Pugh, 1990] ensures that these are constant many.

The process of deleting an infix is omitted due to its great similarity to the insertion process.

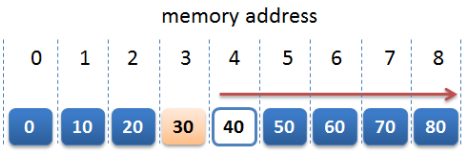


Figure IV.1: Insertion of a Journal-Entry based on a SA

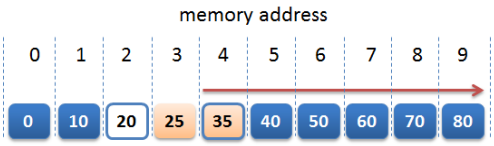


Figure IV.2: Insertion with splitting of a Journal-Entry based on a SA

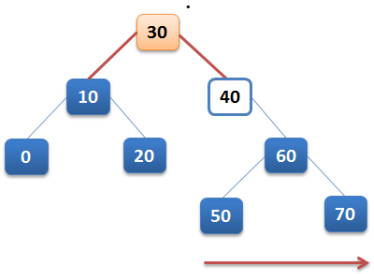


Figure IV.3: Insertion of a Journal-Entry based on a UT

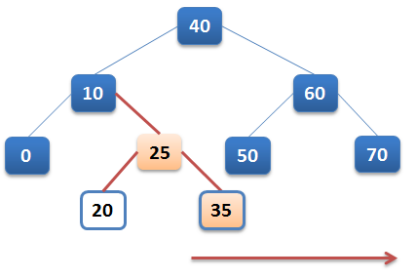


Figure IV.4: Insertion with splitting of a Journal-Entry based on a UT

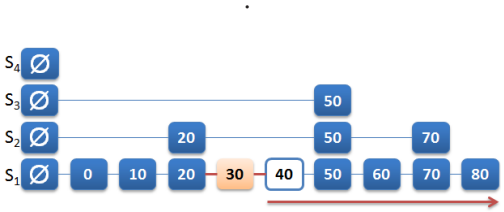


Figure IV.5: Insertion of a Journal-Entry based on a SL

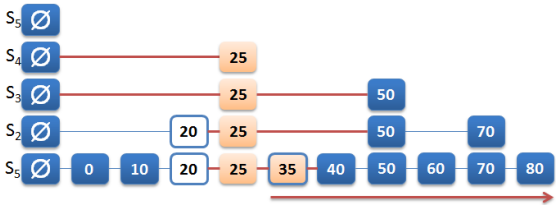


Figure IV.6: Insertion with splitting of a Journal-Entry based on a SL

Figure IV.7: Comparison of insertion between a sorted array, an unbalanced tree and a skip list. Shown are Journal-Entries managed by a SA (top), a UT (middle) and a SL (bottom) data structure. Each entry is labelled with the virtual position. The blue outlined entry represents the found entry by the preceding search. The orange entry marks the insertion. The orange entry with the blue outline marks the second part of the split entry. The red arrow indicates the shift of the virtual positions after the insertion. Red lines in the UT and the SL mark the updated links.

If a prefix or suffix of an Journal-Entry or an entire Journal-Entry is deleted, then only the virtual positions right of this deletion must be updated. If an infix of a Journal-Entry is deleted, then the affected entry must be split. Both cases require the same operations as described above, while the virtual positions are shifted to the left instead.

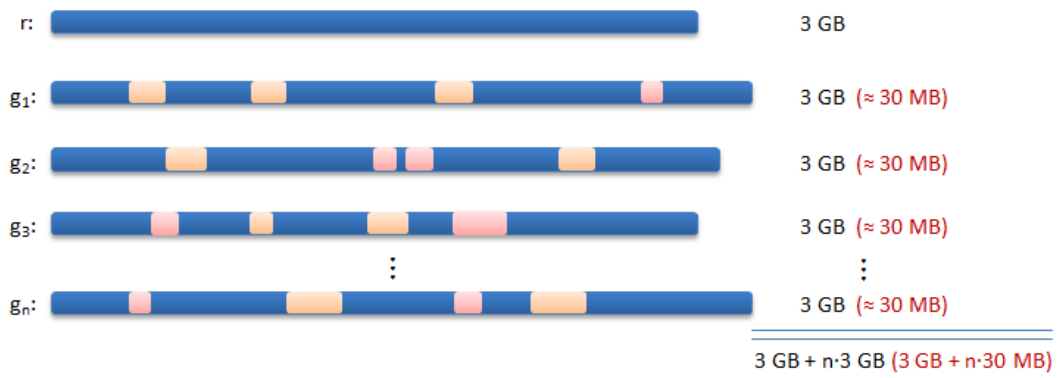


Figure IV.8: General idea of journaling multiple sequences. Shown are n human genomes and their respective sizes in natural and journaled form compared to a common reference r. The manhattan operations are highlighted in blue (matching), orange (inserted) and red (deleted) regions.

4.2 Aggregating Sequence Similarities

Up to now, the concept of the Journal-String and efficient implementations of the underlying data structure were discussed. The following section presents the Journal-Set which is an appropriate container for multiple sequences organised as Journal-Strings. It allows for aggregation of sequence similarities to a common reference sequence. The expected advantage of such an aggregation is displayed in figure IV.8.

Illustrated are n human genomes $(g_1 \dots g_n)$ and their relation to a common reference genome r. Regions that match to the reference are displayed in the same color and differences are highlighted with orange (insertions) or red subregions (deletions) accordingly. If an average size of 3 Gb is assumed for each genome than the expected main memory required for all sequences would be almost $3 + 3 \cdot n$ GB if one character is encoded by one byte. If a journaling approach could be facilitated, the expected main memory required for all sequences would be magnitudes lower leading to a size of around 3 GB + $n \cdot 30$ MB, assuming all genomes differ only in 1% of its sequence. 1% of 3 GB is 30 MB. It is obvious to see that if genomes are 99% identical to each other than the journaling approach could accumulate 100 genomes for the size of one.

This theoretical approach lacks the memory required to store the offsets at which the differences occur. A Journal-Entry requires three position values to encode the corresponding infix and an additional bit for the distinct sources.

As explained in chapter III the known data structures, to aggregate sequence similarities, are relatively static. The purpose of this research is to create a dynamical data structure. For this reason the aggregation of sequence similarities is dislocated from the container itself to each contained sequence. This approach requires some additional substructures increasing the container complexity. The following section describes the implemented design. This design is optimised towards the efficient storage of multiple Journal-Strings.

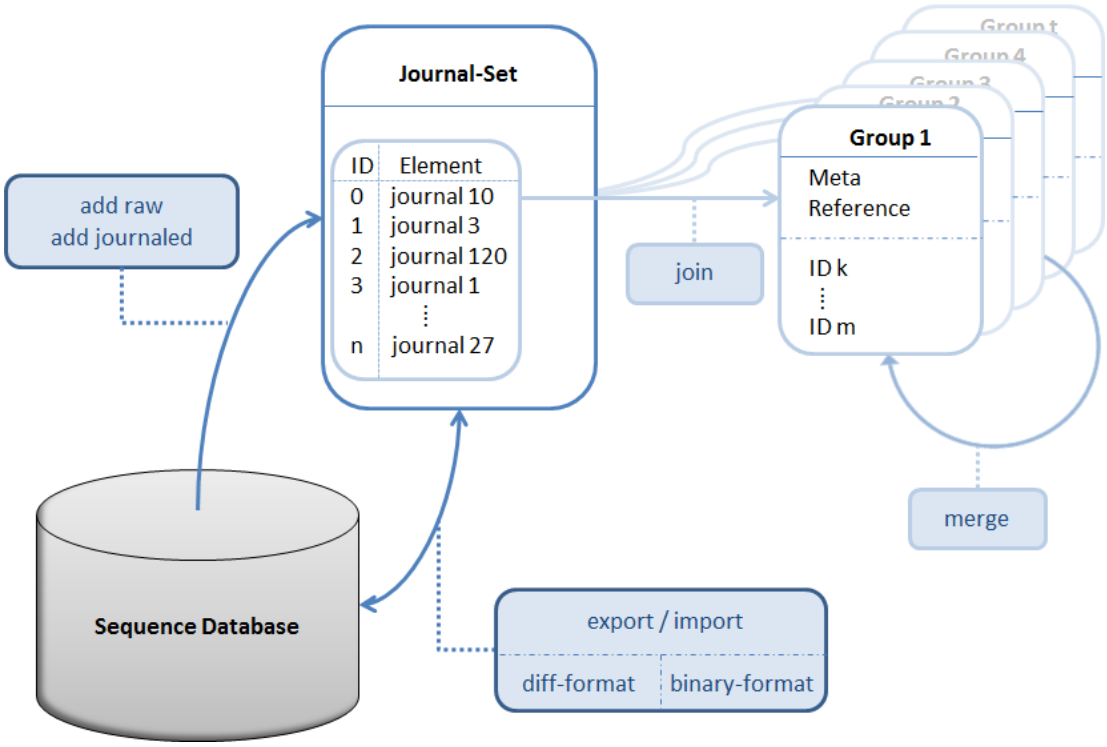


Figure IV.9: Design of the Journal-Set. The central data structure is the Journal-Set which stores the Journal-Strings. A subset of stored strings can be managed by a Journal-Group. In this case the subset refers to the same reference string. The join and the merge methods can be used to add a Journal-String or an entire Journal-Group to another group. Additionally, the container offers solutions to serialise the contained data.

Design

Figure IV.9 shows the general design of the container, also called the *Journal-Set*. The container only stores the assigned Journal-Strings. It does not know the aggregation of its contained strings. The aggregation information is realised by using the *decorator design pattern* described by the "gang of four" Gamma *et al.* [1995]. A decorator adds additional behaviour to an existing object, without changing or adapting its functionality. In this particular case the decorator manages the aggregation of strings within the container. The decorator is referred to as a *Journal-Group*.

Definition 10 (Journal-Set & Journal-Group). The Journal-Set Θ is a container that stores n Journal-Strings. A Journal-Group Γ is a decorator that stores a subset of ids. Ids are used to identify the corresponding Journal-String within Θ. It furthermore, defines a common reference string R^{Γ} for all aggregated Journal-Strings. For each Journal-String in Θ the following relation holds: Each Journal-String is assigned to exactly one Journal-Group. A Journal-Group organises a subset of Journal-Strings. There are maximal n many groups, each representing exactly one Journal-String and at least one group representing n Journal-Strings. The length

of Θ is denoted as $|\Theta| = n$, and the length of Γ is $|\Gamma| = k$ with $1 \leq k \leq n$.

Using this approach it is possible to modify individual sequences without changing or recomputing the entire data structure that stores the strings. This container also allows management of different groups of sequences. Thus it is possible to store entire genomes as groups of chromosomes which is a widely used representation of huge genomic sequences. Since all grouped sequences are journaled with respect to the same reference it is also easy to compare the journaled sequences among each other, by comparing the manhattan operations to the reference at a given virtual position.

One of the major issues that must be addressed is the creation of the manhattan transcript and how a Journal-String can be obtained from such a transcript. The chapter II introduced fundamental algorithmic approaches to compute the best global alignment of two sequences. The following sections discuss efficient methods to generate a Journal-String out of an alignment. Furthermore, it is introduced a scoring function using an adapted affine score function to compute the alignment resulting in a Journal-String which requires minimal memory to store the differences.

4.3 Journal-String Generation

The natural way to construct a Journal-String out of an alignment would be the usage of the insert and delete methods which encode the differences while following the traceback. But this would be very inefficient. For each operation first the corresponding Journal-Entry has to be searched and than inserted or deleted using the above described mechanism. Another problem results from the fact that the traceback usually starts in the last cell (n, m) of the recursion matrix M (explained in section 2.2) and steps back to the origin at the coordinates $(0, 0)$. For example in the case of a Journal-String using the SA this would result in shifting all operations to the right. If the SA contains k values at time t of the traceback, then the next traced operation is inserted at the first position of the SA. It must have a lower virtual position than the k previously applied operations, because the traceback is traversed backwards in the string. Hence, in the next step k entries must be shifted to the right.

The simplest solution of course would be to change the direction of the alignment and of the traceback accordingly. Thus the matrix is computed backwards from cell $M(n, m)$ to $M(0, 0)$ and the traceback is performed in ascending order with respect to the positions of the strings. This approach, however, does not solve the problem for the UT implementation. The operations made during the traceback are in a consecutive order. Hence, the constructed tree would be a linked list, unless balancing operations exist. This particular problem remains independent of the traceback direction.

To solve this problem the **adapter design pattern**, introduced by Gamma *et al.* [1995], was put into practice. The adapter design pattern, often referred to as **wrapper**, translates the interface for an object into a more general interface that is compatible with other objects and methods. For this particular case the interface of the Journal-Descriptor was implemented. Figure IV.10 shows the general design of this adapter. On the left side of the adapter are displayed the three Journal-String specialisations. Each of them can be transformed to a Journal-Descriptor,

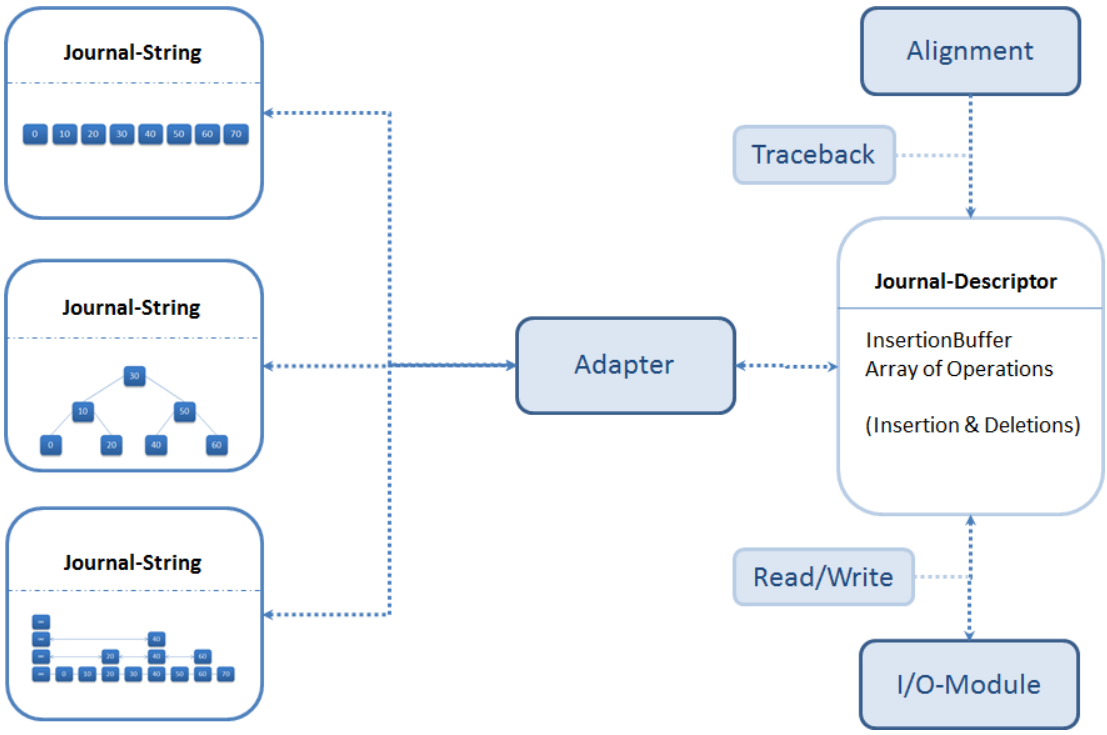


Figure IV.10: The generic interface of the adapter to adapt any type of a Journal-String to a Journal-Descriptor and vice versa.

shown on the right side of the adapter, and vice versa. The Journal-Descriptor is constructed during the traceback of an alignment. It is also the general interface for the I/O-module which is explained in the last part of this chapter.

The Journal-Descriptor only contains the inserted infixes and the Journal-Entries in form of an array. A sorted order of the array is given by sequentially reading the traceback. Based on these raw information it is possible to build each of the Journal-String implementations in linear time. The runtime to adapt a Journal-String requires $O(g + l)$ time, with g being the number of Journal-Entries within the array and l being the length of all inserted infixes.

A major advantage of this approach is the creation of a balanced tree. How a Journal-Des-criptor can be adapted to one of the Journal-String types is described in the following paragraphs.

Adapting a Sorted Array

While performing the traceback over the alignment, Journal-Entries are constructed and appended to the Journal-Descriptor. The entries are stored in a descending order with respect to their virtual positions. The SA can be adapted by reversing the array of the Journal-Descriptor which takes exactly $O(g)$ steps. The algorithm IV.1 shows how the Journal-String j is adapted from the Journal-Descriptor v. In the first step, the algorithm initialises the array of j which is

denoted as T^j in line 3. In the second step, the concatenated, inserted infixes of v (accessed via the parameter `insertionBuffer`) is copied to the insertion source of j (I^j) in line 4. Afterwards, each Journal-Entry of v is copied to the T^j object. Note that v is reversely parsed from the end to the begin (line 7).

```
1  Input: Journal-Descriptor v
2  Output: Journal-String j as sorted array

3  T^j[length(v)]
4  I^j ←v.insertionBuffer
5  for 0 ≤ p < |v| do
6      T^j_p ←v[|v| − 1 − p]
```

Algorithm IV.1: Construction algorithm of the sorted array specialisation from a Journal-Descriptor.

```
1  Input: Journal-Descriptor v
2  Output: Journal-String j as unbalanced tree

3  Initialisation:
4  I^j ←v.insertionBuffer
5  begin ← 0
6  end ← |v|
7  node ← root

8  Recursion: //construct(node, v, begin, end)
9  if begin > end then
10     node ←nil
11 middle ←⌈begin + (end−begin)/2⌉
12 node.cargo ←v[middle]
13 construct(node.leftChild, v, begin, middle − 1)
14 if node.leftChild != nil then
15     node.leftChild.parent ←node
16 construct(node.rightChild, v, middle + 1, end)
17 if node.rightChild != nil then
18     node.rightChild.parent ←node
```

Algorithm IV.2: Construction algorithm of the unbalanced tree specialisation from a Journal-Descriptor.

Adapting an Unbalanced Tree

For the UT specialisation of the Journal-Strings it is possible to use a construction algorithm which guarantees a complete balanced tree without any balancing operations. This algorithm takes only $O(g + l)$ steps. The algorithm IV.2 shows the pseudo code of the algorithm. It exploits the binary search, such that always the middle element contained in the Journal-Descriptor v is picked. This step is shown in line 11. Choosing the middle element partitions the considered interval into two equally sized subintervals. The left subinterval is considered for the left subtree of the current node (line 14) and the right subinterval is considered for the right subtree of the current node (line 17). The method is then recursively called on both subintervals until no elements are left. For each child of the current node, the parent information is set accordingly (line 15 and 18), unless the child is `nil`.

The recursion stops if the $begin$ value of the current interval is greater than the end value (line 9). In this case, there are no further elements left for this particular interval and the corresponding leaf is set to `nil`. Note that the tree consists of nodes which coat the Journal-Entries. The Journal-Entry value is accessed by the `cargo` value of a node (line 12).

Adapting a Skip List

Adapting a skip list can be done in $O(g + l)$ steps, too. The algorithm is explained in IV.3. Again, the concatenated, inserted infixes of v are copied to I^j, the insertion source of the Journal-String j. The skip list is a special form of a double linked list using a second dimension to store the elements on different lists S_r. Each list is also a double linked list and consists of

```
1  Input: Journal-Descriptor v
2  Output: Journal-String j as skip list

3  Initialisation:
4  I^j ←v.insertionBuffer
5  S_1 ←∅

6  for 0 ≤ r < |v| do
7      k ←number of coin tosses until 0 comes up
8      if k ≥ h then
9          for h < p leq k + 1 do
10             create S_p
11             insert ∅ in S_p
12     for k ≥ q geq 1 do
13         insert v[r] in S_q
14         update left and right neighbour of inserted element in S_q
```

Algorithm IV.3: Construction algorithm of the skip list specialisation from a Journal-Descriptor.

a subset of the elements contained in the previous list. The base list S_1 contains all elements. A probability function is used to determine how many lists contain a certain element. In line 5 S_1 is initialised with the empty element $\oslash$. Note that the all lists contain this element and that there exists a list S_h which contains only this element. h defines the maximal height of the current skip list. Subsequent to the initialisation each element of v is inserted into the skip list. The insertion process includes the randomly picked height for the new element in line 7. If the picked height k is greater or equal h, then $k - h + 1$ new lists are created containing the element $\oslash$ (line $8 - 11$). Afterwards the element $v[r]$ is inserted in each list from S_k to S_1. Since v contains the Journal-Entries in descending order and the array is parsed from begin to end, the current element is inserted up front of the skip list behind the $\oslash$ element (line 12, 13). Because of the fixed insert position, the element must not be searched first. After inserting the element, it is linked with its left and right neighbour in the current list line 13.

4.4 Maximal Compression

Since the massive parallel sequencing technologies produce tons of sequencing data that threatens to swamp the available data archives, novel approaches are dearly required to cope with the overwhelming deluge of data. One appropriate solution is the usage of Journal-Strings. A Journal-String represents an alignment of two sequences in a compressed form. Huge blocks of the same information are encoded as a triple of positions as described above. In general, any alignment can be encoded by such a Journal-String. This raises the question, how the alignment that needs fewest memory requirements can be computed. To find such an alignment between two arbitrary sequences a special score function was applied. This score function is a modified version of the affine gap cost function explained in section 2.2.

Recall that a Journal-Entry is a collection of the following four values: segment source (ss), virtual position (vp), physical position (pp) and length (l). They are used to distinguish between the sources an infix refers to and specify the begin and end within the corresponding source and the encoded string. An insertion requires the additional storage of the inserted elements. In contrast, a deletion is represented as a void between the physical positions of two adjacent type-0 entries. The segment source parameter is encoded by a bit. Compared to the `integers` which needs either four or eight bytes, depending on the CPU architecture, this bit does not have a particular influence on the size of a Journal-Entry, such that it can be omitted.

As suggested above the actual size of a Journal-Entry depends on the used CPU architecture. While the `integer` type is a 4 byte sized chunk on 32-bit architectures, it uses 8 bytes on some 64-bit architectures. Based on this conditions, the creation of a Journal-Entry requires either $3 \cdot 4 = 12$ bytes of memory on a 32-bit machine or eventually $3 \cdot 8 = 24$ bytes of memory on a 64-bit machine. Because the number of `integers` of a Journal-Entry is static the actual `integer` size does not matter. For brevity a 32-bit machine is equivalently used for a 64-bit machine in the following. Further let the alphabet values be stored in `char` values which requires one byte. Hence, the inserted infix of length l requires $l \cdot 1$ bytes memory requirements.

A crucial observation of the encoding used by Journal-Strings is that encodings of insertions and deletions require different space consumptions in worst-case. More specifically, an inser-

tion requires the allocation of at most two new Journal-Entries, whereas a deletion requires the additional allocation of at most one Journal-Entry. Furthermore, inserted infixes are stored within the insertion source of the Journal-String, while a deletion does not require additional storage capacities. In contrast to a gap function used to compute sequence similarity, the purposed gap function must distinguish between insertions and deletions. This can be achieved by applying *bi-affine gap costs*. Affine gap costs (section 2.2) share the property of different penalties for the opening and the extension of a gap. The bi-affine gap costs extend this property by distinguishing between a horizontal and a vertical directed gap within the alignment matrix.

Thus, the following bi-affine gap function can be defined for the maximal compression problem as followed:

$$\gamma(l) = \begin{cases} 25 + (l - 1 \cdot 1 & \text{bytes for an insertion of length l} \\ 12 & \text{bytes for a deletion of length l} \end{cases} \tag{IV.3}$$

Opening an insertion requires 25 bytes. 24 bytes for the two new allocated Journal-Entries and 1 byte for the inserted character. If the insertion is extended, then only 1 byte for each appended character must be added. The opening of a deletion requires 12 byte as discussed above. Due to the special encoding of the deletions (explained previously in section 3.1), no additional penalties are considered.

The score function for the maximal compression problem can be defined as:

$$\sigma(a_i, b_j) = \begin{cases} 0 & \text{bytes, if } a_i = b_j \\ \infty & \text{bytes, if } a_i \neq b_j. \end{cases} \tag{IV.4}$$

A match is not penalised by the score function. In the initial state of a Journal-String, there exists exactly one Journal-Entry covering the entire reference string of the Journal-String. This holds for every alignment, such that the initial node can be omitted as a constant factor. Furthermore, each matching character of the alignment is already covered by this node. During the construction of the Journal-String the initial node is modified by the applied operations. These operations do not add new matching characters. On the other side, a mismatch is not allowed which why it is penalised with infinity. As explained in section 2.2 a mismatch can always be substituted with an insertion followed by a deletion or vice versa.

Based on the above given bi-affine gap function and the score function, the formula of section 2.2 defining the initialisation and the recursion of the global alignment algorithm for arbitrary gap costs can be adapted as followed:
Initialisation:

$$\begin{aligned} &M(0,0) = D(0,0) = I(0,0) = \infty \\ &M(i,0) = D(i,0) = 25 + (i-1), I(i,0) = \infty \text{ for } i = 1, \dots, n \\ &M(0,j) = I(0,j) = 12, D(0,j) = \infty \text{ for } j = 1, \dots, m \end{aligned} \tag{IV.5}$$

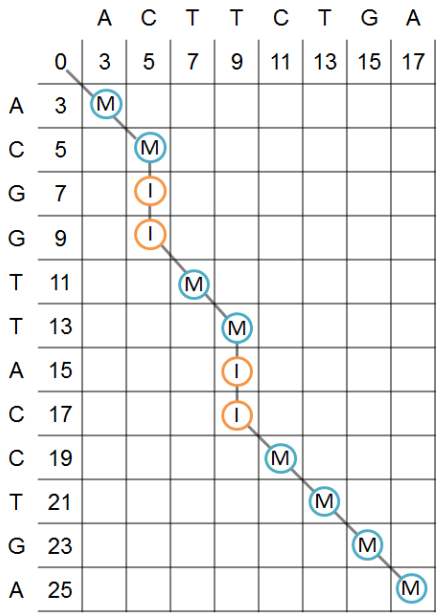


Figure IV.11: Best global alignment problem. The depicted path shows the highest similarity between the two strings.

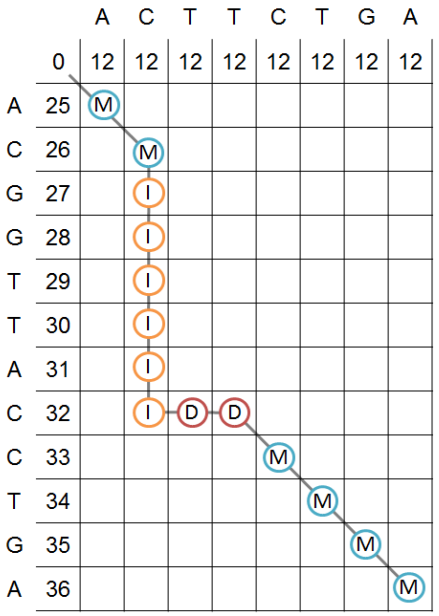


Figure IV.12: Maximal compression problem. The depicted path reveals the highest compression factor between the two strings.

Recursion:

$$M(i,j) = min \begin{cases} M(i-1,j-1) & +\sigma(a_i, b_j) \\ D(i-1,j-1) & +\sigma(a_i, b_j) \\ I(i-1,j-1) & +\sigma(a_i, b_j) \end{cases} \tag{IV.6}$$

$$D(i,j) = min \begin{cases} M(i-1,j) & +12 \\ D(i-1,j) & +0 \end{cases} \tag{IV.7}$$

$$I(i,j) = min \begin{cases} M(i,j-1) & +25 \\ I(i,j-1) & +1 \end{cases} \tag{IV.8}$$

Using the stated initialisation and recursion formula, one can use the Gotoh algorithm 2.2 to compute the maximal compressed alignment with bi-affine gap costs.

The figures IV.11 and IV.12 demonstrate the differences between the best global alignment problem and the the maximal compression problem. Shown are the tracebacks between both strings $a = ACTTCTGA$ and $b = ACGGTTACCTGA$. While the left figure shows the traceback of the best global alignment, the figure on the right side presents the alignment with the maximal compression. In this case both separated insertions of the best global alignment are collected to one big insertion followed by a small deletion to return back to the diagonal. The following computation shows the memory advantage of the right alignment if the operations

of the alignments are journaled.

objective:	similarity	compression
initial size	$12byte$	$12byte$
1. insertion	$26byte$	$30byte$
2. insertion | deletion	$26byte$	$12byte$
memory requirements:	$64byte$	$54byte$

The alignment of figure IV.11 requires 64 bytes to store the encoded transcript. The alignment shown in figure IV.12 requires 54 bytes which are 10 bytes less. Note that the presented figures are only illustrations demonstrating the main difference of both alignment strategies.

4.5 Merging Multiple Journal-Strings

One of the essential features in this thesis is the synchronisation algorithm used to rapidly exchange a reference sequence of a Journal-Group. Such scenarios could happen if, for example, a new update of the underlying reference genome assembly was published. Then it might be of interest to journal the strings contained in the affected Journal-Group according to the new reference. Another possible scenario could be the identification of the string that has the highest similarity to all other strings contained in a group. Such an approach could reduce the memory requirements even more. In either case, the reference string has changed. The naive way to replace the reference string would be to recompute the alignments for each contained Journal-String with respect to the new reference string. This can be a very time-consuming task depending on the number of sequences contained within the Journal-Group.

The developed algorithm is based on a strategy that aligns two sequences based on their encoding as Journal-Strings. This approach assumes a high similarity between the old and the new reference string. Following this approach, it is sufficient to compute an alignment between both reference strings and to construct a Journal-String encoding the alignment between the old reference and the new reference string. This is done in the first part of the algorithm. In the second part, the Journal-String encoding the differences necessary to transform the old reference string into the new reference string, is used to synchronise each Journal-String contained in the affected group with the new reference string.

In the following, the Journal-String of the old reference is referred to as the *inner tree*. Each unsynchronised Journal-String is called *outer tree*. The synchronisation algorithm performs a refinement on the Journal-Entries of the outer tree and the inner tree. The refined set of Journal-Entries is then synchronised linearly following a certain synchronisation rule. Different synchronisation rules exist depending on the current setting of the refined entries of the inner and the outer tree. After running the algorithm, the synchronised Journal-Strings refer to the new reference string.

In the following subsections, the problem is defined more formally and the algorithm is explained using an illustration that matches each synchronisation case.

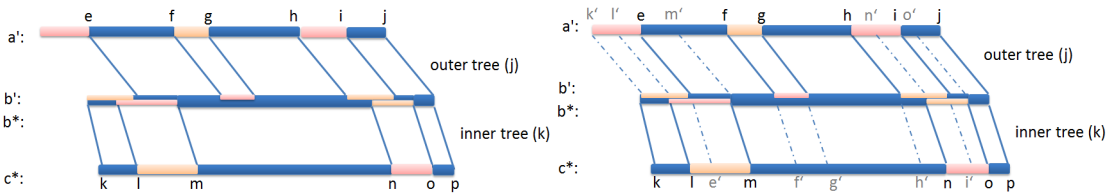


Figure IV.13: Refinement of two Journal-Strings Shown is the outer tree representing the aligned strings a' and b' and the inner tree representing the aligned strings b^* and c^*. The blue lines determine the Journal-Entries of the corresponding Journal-String. The orange and red marked areas indicate insertions and deletions. After the refinement the Journal-Strings are partitioned such that each Journal-Entry of the outer tree has a corresponding Journal-Entry in the inner tree.

Problem Definition

Let $a, b, c \in \Sigma^*$ be three strings. Further let j be the Journal-String (outer tree) encoding an alignment between a and b, such that $R^j = b$ is the underlying reference source of j. Recall that a Journal-String j consists of three data members: the reference source R^j, the insertion source I^j and the set of Journal-Entries T^j. Let k be the Journal-String (inner tree) encoding an alignment between b and c, with $R^k = c$ being the underlying reference source. The outer tree is completely synchronised if $R^j = R^k$ and I^j and T^j encode the differences with respect to the new reference string c.

In order to arrive there, each Journal-Entry of the outer tree must be compared with the corresponding Journal-Entry of the inner tree. Two Journal-Entries are comparable if their virtual positions and lengths are equal. This special entity might not be given for every outer and inner tree. It is more likely that the Journal-Entries of the trees overlap each other. To find a set of comparable Journal-Entries, a refinement, like the segment match refinement discussed by Rausch *et al.* [2008], is computed. Therefore, the set of Journal-Entries of the inner and the outer tree is partitioned in segments.

Let $\omega_{p,q}$ denote a *Journal-Infix*, with p being the begin position and q the end position of the infix, of a Journal-Entry. Let vp_r be the virtual position of the r-th Journal-Entry and l_r its length, then the following must hold: $\omega_{p,q}$ is an infix of T^j_r, if $vp_r \leq p < q \leq vp_r + l_r$. The Journal-Infix $\phi_{u,v}$ is defined analogues for a Journal-Entry contained in the inner tree.

$s = (\omega_{p,q}, \phi_{u,v})$ is called a Journal-Segment between j and k, if $p = u$ and $q = v$. Let S denote the set of all possible segments between j and k. A segment refinement finds a subset S' of S containing a minimal number of Journal-Segments, such that no two Journal-Infixes between the outer and the inner tree overlap.

Figure IV.13 illustrates the segment refinement. It shows a possible set of Journal-Entries for the outer and the inner tree. On the left side the typical scenario of the outer and the inner tree is shown. Both Journal-Strings consists of a set of Journal-Entries. Since the Journal-Entries of both trees overlap each other, a comparison of them is not possible. For example the infix $\omega_{e,f}$ of the outer tree overlaps with the infixes $\phi_{l,m}$ and $\phi_{m,n}$ of the inner tree and both entries in

the inner tree encode different operations.

The right part of figure IV.13 shows the refinement of the outer and the inner tree. Here, the Journal-Entry from e to f of the outer tree is partitioned at position m' into two infixes $\omega_{e,m'}$ and $\omega_{m',f}$. The same holds for the Journal-Entry from l to m at position e' and from m to n at position f' in the inner tree. The refined segment $s' = (\omega_{e,m'}, \phi_{e',m})$ now defines two regions between the outer and the inner tree that can be compared, because they do not overlap any longer and each infix encodes exactly one string of the same operations.

Figure IV.13 already illustrates that different synchronisation rules must be applied. In general, these rules can be categorised in *compatible operations* and *incompatible operations*. In the following the pair (a', b') refers to the strings of the global alignment of a and b (outer tree), and the pair (b^*, c^*) refers to the strings of the global alignment of b and c (inner tree). In the following all possible compatible operations are defined:

$$\text{If } \omega_{p,q} \text{ encodes a matching region and } \phi_{p,q} \text{ encodes a matching region, then for } p \leq t < q : a'_t = b'_t = b^*_t = c^*_t = x, \text{ with } x \in \Sigma. \tag{IV.9}$$

$$\text{If } \omega_{i,p} \text{ encodes a matching region and } \phi_{i,p} \text{ encodes an inserted region, then for } p \leq t < q : a'_t = b'_t = b^*_t = x \text{ is inserted into } c^*_t = -, \text{ with } x \in \Sigma. \tag{IV.10}$$

$$\text{If } \omega_{i,p} \text{ encodes a deleted region and } \phi_{i,p} \text{ encodes a matching region, then for } p \leq t < q : a'_t = - \text{ is deleted from } b'_t = b^*_t = c^*_t = x, \text{ with } x \in \Sigma. \tag{IV.11}$$

$$\text{If } \omega_{i,p} \text{ encodes a deleted region and } \phi_{i,p} \text{ encodes an inserted region, then for } p \leq t < q : a'_t = c^*_t = - \text{ and } b'_t = b^*_t = x, \text{ with } x \in \Sigma, \text{ hence the infix is not stored.} \tag{IV.12}$$

The incompatible operations are defined as follows:

$$\text{If } \omega_{i,p} \text{ encodes an inserted region and } \phi_{i,p} \text{ encodes a matching region, then for } p \leq t < q : a'_t = x \text{ and } b'_t = - \text{ and } b^*_t = c^*_t = y, \text{ with } x, y \in \Sigma. \tag{IV.13}$$

$$\text{If } \omega_{i,p} \text{ encodes an inserted region and } \phi_{i,p} \text{ encodes an inserted region, then for } p \leq t < q : a'_t = x \text{ and } b'_t = - \text{ and } b^*_t = y \text{ and } c^*_t = -, \text{ with } x, y \in \Sigma. \tag{IV.14}$$

$$\text{If } \omega_{i,p} \text{ encodes an inserted region and } \phi_{i,p} \text{ encodes a deleted region, then for } p \leq t < q : a'_t = x \text{ and } b'_t = - \text{ and } b^*_t = - \text{ and } c^*_t = y, \text{ with } x, y \in \Sigma. \tag{IV.15}$$

$$\text{If } \omega_{i,p} \text{ encodes a matching region and } \phi_{i,p} \text{ encodes a deleted region, then for } p \leq t < q : a'_t = b'_t = x \text{ and } b^*_t = - \text{ and } c^*_t = y, \text{ with } x, y \in \Sigma. \tag{IV.16}$$

$$\text{If } \omega_{i,p} \text{ encodes a deleted region and } \phi_{i,p} \text{ encodes a deleted region, then for } p \leq t < q : a'_t = - \text{ and } b'_t = x \text{ and } b^*_t = - \text{ and } c^*_t = y, \text{ with } x, y \in \Sigma. \tag{IV.17}$$

The basic difference between compatible and incompatible operations depends on the values of b'_t and b^*_t, as they are the connection between the values a'_t and c^*_t. In constellations of

encoded operations where $b'_t = b^*_t$, the element a'_t can be directly synchronised with the element c^*_t. Rules IV.9 to IV.12 describe in which way a'_t can be journaled to c^*_t. The easiest case is obviously, when both Journal-Infixes encode a matching region (rule IV.9). In this case, value a'_t has to be equal to c^*_t. Rule IV.10 describes the case in which the value a'_t is inserted into c^*_t. The special case described in rule IV.12 refers to the fact that the value $a'_t = c^*_t = -$ which not allowed per definition of an alignment. Hence, in such cases the corresponding infix need not be considered within the synchronised Journal-String.

Rules IV.13 to IV.17 define the incompatible synchronisation rules. In these cases, the value $b'_t \neq b^*_t$, so there is no connection between the values a'_t and c^*_t. Those cases occur if either the infix of the outer tree encodes an insertion or the infix of the inner tree encodes a deletion. Here, either the values $a'_p, \ldots, a'_{q-1}$ are inserted before c^*_p while the respective values of c^* are shifted to the right, or the values $c^*_p, \ldots, c^*_{q-1}$ are deleted before a'_p while the respective values of a' are shifted to the right. These shifts within the aligned stings can lead to new overlaps of the refined segments. In order to avoid recomputing the refinement in each iteration of the algorithm, the refinement is computed on-the-fly. The actual process is demonstrated in the example below.

Rule IV.15 constitutes an exception, as it is not excluded that some elements of $a'_p, \ldots, a'_{q-1}$ matches some values of $c^*_p, \ldots, c^*_{q-1}$. In order to find such possibly matching elements a partial alignment has to be computed for the considered infixes. Such an approach would be a great extension for the presented algorithm, as it allows the computation of alignments that are closer to the best alignment.

In the following paragraphs, the actual algorithm is explained using an example that covers the distinct synchronisation rules.

Algorithm

The algorithm iterates over the Journal-Entries of the inner tree and applies the refinement on-the-fly. Thus, the main loop of the algorithm distinguishes only four different cases, depending which source is covered by the current Journal-Entry. Let e^k denote the current Journal-Entry of the inner tree (k) and e^j the current Journal-Entry of the outer tree (j). The following cases are distinguished by the algorithm:

- e^j refers to $R^j \wedge$ of e^k refers to R^k
- e^j refers to $R^j \wedge$ of e^k refers to I^k
- e^j refers to $I^j \wedge$ of e^k refers to R^k
- e^j refers to $I^j \wedge$ of e^k refers to I^k

In addition, the algorithm uses three auxiliary values to compute the refinement on-the-fly. These auxiliary values store the shifts in the respective virtual positions of the inner tree or physical positions of the outer tree and are used to determine possible deletions within the respective reference sources of the trees:

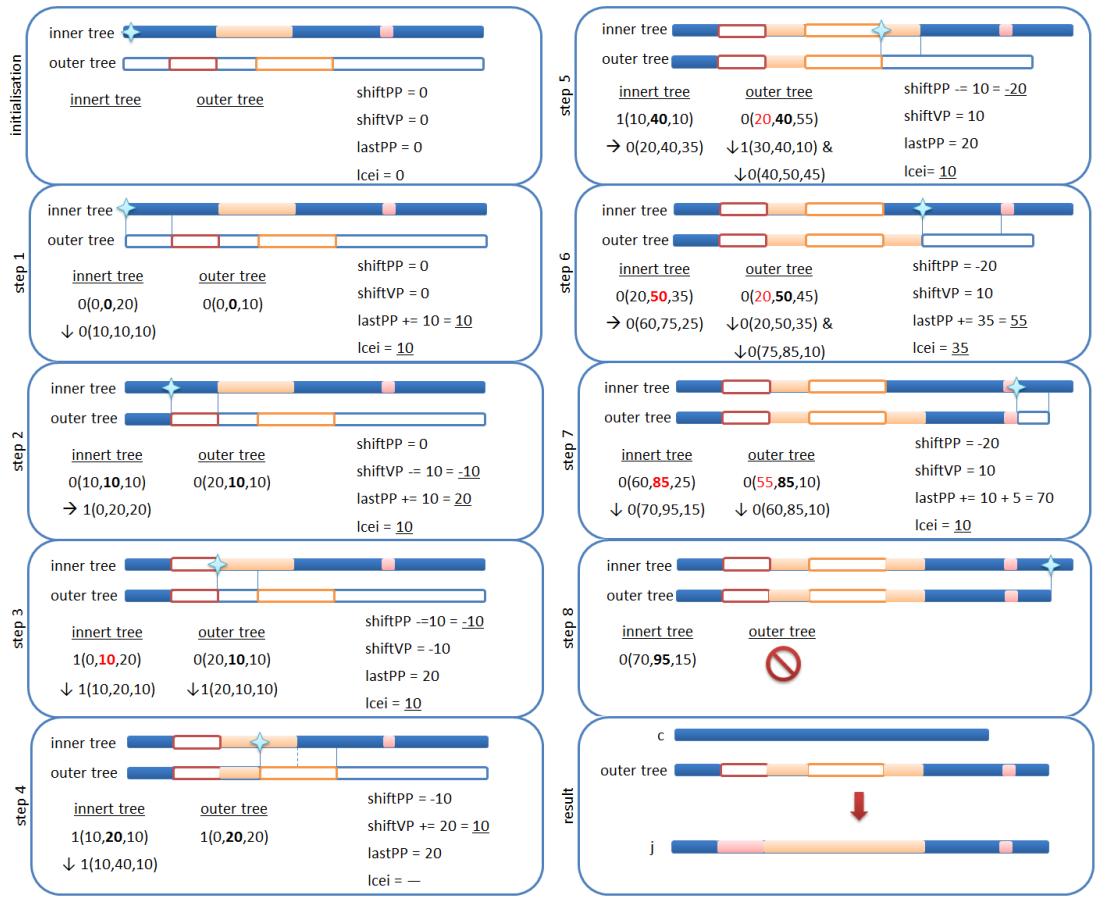


Figure IV.14: Example of the synchronisation algorithm. Shown are the distinct synchronisation cases of the algorithm. Each window contains the current status of the synchronisation and the values of the auxiliary data types. The result is a synchronised Journal-String.

- *shiftPP*: represents the current shift of the physical positions within the outer tree. All up-coming Journal-Entries within the outer tree must be updated accordingly.
- *shiftVP*: represents the current shift of virtual positions within the inner tree. All up-coming Journal-Entries within the inner tree must be updated accordingly.
- *lastPP*: reveals a deletion operation within the outer tree.

The following example describes the iterations of the algorithm step by step and demonstrates the application of the different synchronisation cases.

Figure IV.14 demonstrates the process of the algorithm and table IV.2 lists the Journal-Entries that are presumed for this example. The composition of the outer tree is changed after some synchronisation steps. The tables IV.3 to IV.6 cover the changes of the synchronised outer tree.

Table IV.2: The Journal-Entry of the inner and the outer tree. The Journal-Entries are described in the following way $[ss(pp, vp, l)]$.

No	inner tree	outer tree
1	0(0, 0, 20)	0(0, 0, 10)
2	1(0, 20, 20)	0(20, 10, 10)
3	0(20, 40, 35)	1(0, 20, 20)
4	0(60, 75, 25)	0(30, 40, 55)

Table IV.3: Outer tree after step 3.

No	outer tree
1	0(0, 0, 10)
2	1(20, 10, 10)
3	1(0, 20, 20)
4	0(30, 40, 55)

Table IV.4: Outer tree after step 5.

No	outer tree
1	0(0, 0, 10)
2	1(20, 10, 10)
3	1(0, 20, 20)
4	1(30, 40, 10)
5	0(40, 50, 45)

Table IV.5: Outer tree after step 6.

No	outer tree
1	0(0, 0, 10)
2	1(20, 10, 10)
3	1(0, 20, 20)
4	1(30, 40, 10)
5	0(20, 50, 45)

Table IV.6: Outer tree after step 7.

No	outer tree
1	0(0, 0, 10)
2	1(20, 10, 10)
3	1(0, 20, 20)
4	1(30, 40, 10)
5	0(20, 50, 45)
6	0(60, 85, 10)

Initialisation: The initial situation is shown in the "initialisation" frame in which the outer tree is unsynchronised. Note that the red marked regions within the trees indicate deleted regions, while orange marked regions indicate insertions. Furthermore, the Journal-Entries of the outer tree are represented as unfilled bars, in order to distinguish them from the operations of the inner tree which are filled. The auxiliary parameters are set to 0. The highlighted blue star at the beginning of the inner tree represents the current virtual position. The blue strokes between the trees indicate which refined segment is currently processed by the synchronisation. Note that the refinement is done on-the-fly. The additional parameter *longest common entry infix* (lcei) represents the length of the refined segment.

Note that the sign → before an e^k indicates that the next e^k of the inner tree is selected. The sign ↓ indicates an update of the respective entry. The notations is analogue to an entry of the outer tree. Tables IV.3 to IV.6 cover the modifications within the outer tree after an e^j of the current iteration step was synchronised (indicated by the ↓).

Step 1: The algorithm starts with the selection of the first entry of the inner tree which is $0(0, 0, 20)$. Then the corresponding e^j is selected $(0(0, 0, 10))$. The corresponding e^j is the Journal-Entry that has the same virtual position as the current e^k which is in this case 0. The positions are marked bold within figure IV.14 (frame "Step 1"). The $lcei$ of the selected entries is 10. The e^k as well as the e^j refer to the reference string. Hence, the first compatible rule IV.9 is applied for e^j which remains unchanged. Now, the star is shifted to the right by the size of $lcei$. At the end of the first step the parameter $lastPP$ is updated by the size of $lcei$,

too. Note that the updated parameters are underlined in the respective step.

Step 2: The second step proceeds with the current segment with $0(10, 10, 10)$ for the e^k and $0(20, 10, 10)$ for the e^j. There is a deletion of length 10 within the outer tree that must be considered. The algorithm detects this deletion by comparing the physical position of the current e^j with the value of $lastPP$ from the step before. In this case, the synchronisation rule IV.11 is applied and the corresponding infix within the inner tree of length $lcei$ is deleted. Note that the inner tree is static. The deletion within the inner tree is recognised as a left shift of the virtual positions of the up-coming entries of the inner tree and is recognised in $shiftVP = -10$. Afterwards, the star is shifted to the right by the size of $lcei = 10$, such that the second e^k of the inner tree is selected with $1(0, 20, 20)$. At the end of step two the value of $lastPP$ is updated to 20. The deletion is synchronised now.

Step 3: At the beginning of each step, the virtual position of e^k is updated by the $shiftVP$ parameter, if and only if it has been recently selected (indicated by a $\rightarrow$ in the step before). The same holds for the physical position of e^j which is updated by the $shiftPP$ parameter. In the third step, the virtual position of the selected e^k is updated to 10 (highlighted with red color), because of the $shiftVP$ parameter. This parameter was recently shifted to the left by the size of the synchronised deletion of the outer tree. Then the corresponding e^j is selected which is the same as in step two. This is correct since this node was not directly synchronised due to the deletion. The $lastPP$ value now fits the current physical position of e^j and no deletions must be considered. In the current synchronisation case, e^k covers an insertion and e^j covers a match. In this case, the second formula of IV.10 is applied. The first $lcei$ positions of e^j are converted to an insertion which results in the setting $1(20, 10, 10)$ for the synchronised e^j. Note that the physical position of the updated entry is 20 since there are already 20 values stored in the corresponding insertion source. The physical positions of the up-coming entries within the outer tree must be shifted to the left by the size of the conversion, such that the next e^j continues at the last physical position of the old e^j. The shift is recognised in $shiftPP$ which is now set to -10. Again, the blue star is shifted to the right by the size of $lcei$. $lastPP$ remains unchanged, because the recent synchronisation step did not change the offset of the physical positions of the outer tree.

Step 4: In step four, the algorithm moves into the fourth synchronisation case, in which e^k as well e^j cover an inserted region. The process of this case is equal to the third case of the algorithm, in which e^j covers an insertion and e^k a matching region. In either case e^j is inserted before e^k (see synchronisation rules IV.14 and IV.13). The current settings are $0(10, 20, 10)$ and $1(0, 20, 20)$ for e^k and the corresponding e^j. In this particular case, the entire insertion infix covered by e^j is inserted in the target reference c without partitioning into smaller infixes. The blue strokes cover the entire e^j and $lcei$ is omitted as it is not required for this synchronisation part. e^j remains unchanged and only the blue star is shifted by the length of e^j. The shift within the virtual positions is recognised by the parameter $shiftVP = 10$.

Step 5: The corresponding e^j of the current blue star position is the fourth e^j with the values $0(30, 40, 55)$. Since this entry covers a match, the physical position must first be updated by -10, which is the current value of $shiftPP$. It changes to $0(20, 40, 55)$ which fits the $lastPP$ value. In this particular situation the $lcei$ value is less than the length of e^j. Hence, e^j is split into two entries e_j^1 and e_j^2 at the offset $lcei$. e_j^1 is converted to an insertion $(1(30, 40, 10))$ based on the rule IV.10 and e_j^2 is set to $0(40, 50, 45)$ which continues at the split position. Table IV.4 shows the changed composition of the outer tree. Again, $shiftPP$ is updated to -20, representing the shifts in the physical positions of the outer tree due to the synchronised insertions. The next e^k is selected.

Step 6: First the virtual position of e^k and the physical position of e^j ($0(20, 50, 45)$ and $0(20, 50, 45)$) are synchronised. The resulting $lcei$ value is 35 which is less than the length of e^j. Hence, e^j must be split again into two entries e_j^1 and e_j^2. Because both e^k as well as e_j^1 ($0(20, 50, 35)$) cover a matching region, there is nothing to do (compare synchronisation rule IV.9). The e_j^2 entry continues at the $lcei$ offset ($0(75, 85, 10)$). The physical position within the outer tree has moved such that $lastPP$ is updated. It points to the 55-th position. Table IV.5 covers the modifications made within the outer tree.

Step 7: In this iteration step the last e^k is selected $0(60, 85, 25)$ and the corresponding e^j ($0(75, 85, 10)$) taken. Again the virtual position of e^k and the physical position of e^j must be updated first resulting in $0(60, 85, 25)$ and $0(55, 85, 10)$. The physical position of the current e^j reflects all modifications made before within the outer tree. If this physical position is smaller than the physical position of e^k, then there must be a deletion within the inner tree. In this case, the deletion has a size of 5. Here the synchronisation rule of IV.16 is applied. The $lastPP$ parameter is updated accordingly. Table IV.6 covers the new composition of the outer tree.

Step 8: There is no corresponding e^j for the virtual position ($0(70, 95, 15)$) of the current e^k, such that the algorithm terminates.

Result: The outer tree is now completely synchronised while the algorithm only compared the refined segments. The old Journal-String j refers now to the reference string $c = R^j$. The algorithm did not compare any character and used the refined segments to synchronise long infixes. The runtime has a complexity of $O(|T^j| + |T^k|)$ as each Journal-Entry of the inner and the outer tree is visited only one time.

4.6 Container Serialisation

The last part of this chapter addresses solutions for the serialisation of the Journal-Strings contained in the Journal-Set to secondary memory. The data can be imported and exported in two different formats. Figure IV.15 visualises the IO-module of the Journal-Set.

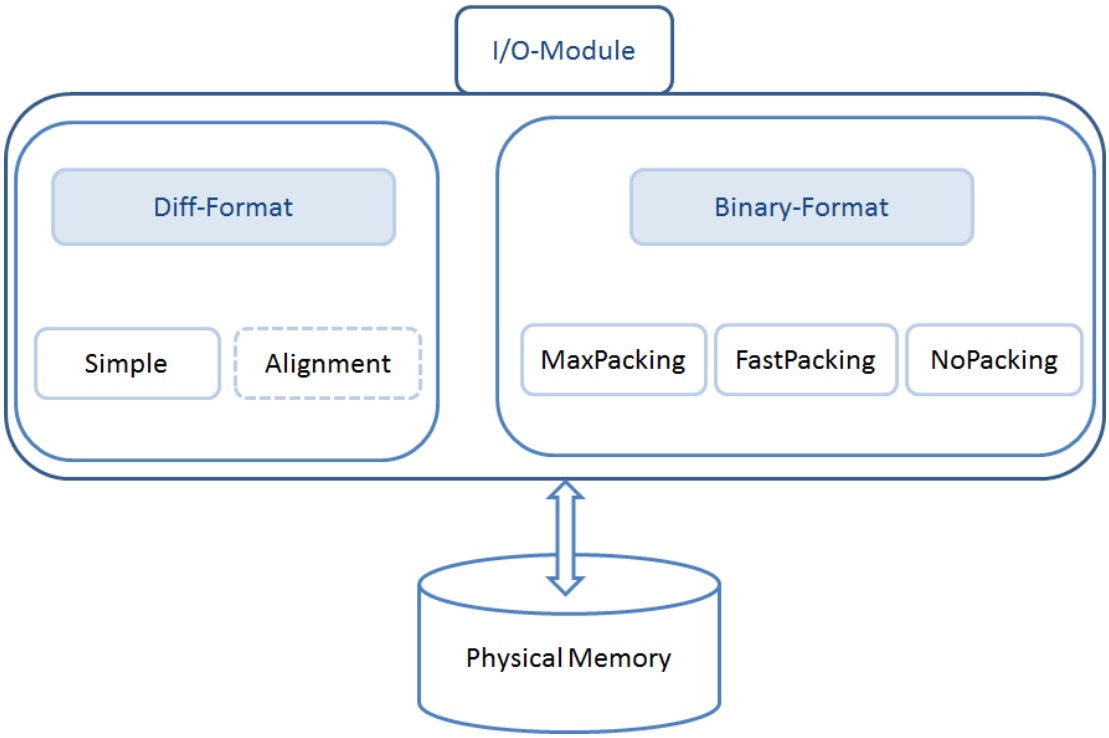


Figure IV.15: I/O-Module interface.

Two distinct formats were designed and implemented. The Diff-Format exports and imports the files in a human-readable format such that the data can be parsed by any other program. This format allows an easy access of the data. The file can directly be modified with any text editor. In contrast, the Binary-Format serialises the files in a space-saving way and can compress the data with additional bit packing strategies.

In either way, the reference strings of the Journal-Groups managed within the set are not serialised to secondary memory. The reference string must be separately specified during the import. For each reference string a checksum can be computed and stored within the exported files. Using this checksum, the reference string can be verified during the import. The import fails if the checksum of the externally loaded reference string is inconsistent with the stored checksum.

In order to achieve higher compression results, for a deletion only the physical position and the length and for an insertion only the physical position and the inserted infix are stored. The virtual position is used to allow fast random access operations to the Journal-String. This is no longer required when the data is exported to secondary memory. Thus, the compressor removes the redundant virtual position. The decompressor can reconstruct the correct virtual positions while reading the stored operations. In addition, only insertions and deletions are stored while the information of matching regions is removed. Since deletions are encoded as voids between the physical positions of two type-0 entries, the compressor replaces such two

Table IV.7: Bits per value for distinct alphabets.

Alphabet	A	C	G	T	N
DNA	00	01	10	11	-
DNA 5	000	001	010	011	100

entries by one entry covering only the start and length of the deletion within the reference. The decompressor reconstructs the two Journal-Entries based on the exported offsets of the deletion.

Diff-Format

The Diff-Format saves the Journal-Set in a human-readable format. Such files can be manipulated easily and can be parsed by other programs if necessary. A demonstration of this file format is shown in the appendix C.1.

The Diff-Format is further categorised by two specialisations: the `Simple` and the `Alignment` method. The first method simply writes the insertion and deletion operations line by line. An insertion is identified by its occurrence within the reference and the corresponding inserted infix exported in "FASTA" format. A deletion is stored as a tupel of begin position and length. The Future versions of this format will also support an `Alignment` method which writes the differences in form of an alignment to the reference string, while the actual operation is extended by some positions to both sides. The `Simple` format is explained in more detail in the appendix C.1.

Binary-Format

In contrast to the Diff-Format, the Binary-Format cannot be manipulated with other programs if they don't know the underlying structure of the binary format. The main difference to the Diff-Format is that no line delimiters are used. Each information is appended consecutively in memory. The structure of the Binary-Format is described in the appendix C.2.

The box for the Binary-Format in figure IV.15 contains three different bit packing methods. Bit packing is a commonly known method to store values of an alphabet as bits. Since a byte is the smallest possible unit that can be addressed, the alphabet values are packed into host values such as the eight bits of a byte. Such approaches can be applied if the number of bits used to encode all values of an alphabet is less than the number of bits of the host value. The Table IV.7 displays the bit patterns used for the DNA and the DNA 5 alphabet.

In the following, a byte is used as the host value for the bit packing method. The DNA alphabet has a maximal entropy of two bpc. Hence, there can be exactly four values packed into the eight bits of one byte. Unfortunately, when dealing with genomic data often the DNA 5 alphabet must be used. The DNA 5 alphabet requires at least three bits per character such that only two complete values can be packed into one byte. The implemented method `FastPacking` fills a byte as long as a complete value can be added to the host value. This implies that using this method on a DNA 5 alphabet and a byte as the host value, each byte would waste at least two bits. In order to exploit the last two bits of a byte even for a DNA 5 alphabet, the

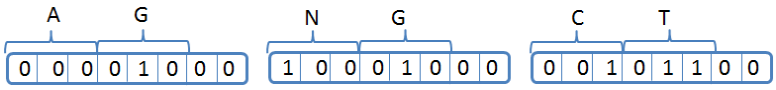


Figure IV.16: The approach for fast bit packing. Used is the DNA 5 alphabet with 3 bits per value.

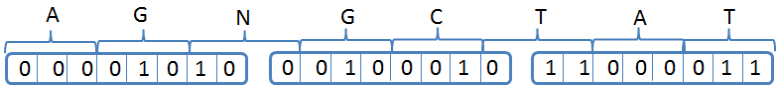


Figure IV.17: The approach for maximal bit packing. Used is the DNA 5 alphabet with 3 bits per value.

method `MaxPacking` was implemented. This strategy computes bit overlaps for a single byte and continues with the remaining bits of the last value in the next byte. Thus maximal seven bits are wasted independent of the length of the insertion string.

Both approaches first compute the size of the host string (p). For the `MaxPacking` strategy this can be computed as:

$$|p| = \left\lceil \frac{|u| \cdot bpv}{8} \right\rceil \tag{IV.18}$$

with u the unpacked string and bpv the bits required for an alphabet value. The division by 8 can be replaced equivalently by shifting the bits three positions to the left which can be performed faster by the CPU. This function is variable for any alphabet size that can be encoded with at most eight bits. For the `FastPacking` approach the length of the packed string can be easily obtained with the following equation:

$$|p| = \left\lceil \frac{|u|}{vph} \right\rceil, \tag{IV.19}$$

with vph being the number of alphabet values that completely fit into one value of the host.

Figure IV.16 and figure IV.17 demonstrate the main differences of both packing strategies. Since the DNA 5 alphabet has five values, there are at least three bits necessary to encode them. Thus maximal two values fit completely into one byte. The `MaxPacking` strategy uses the remaining two bits of the first byte to store the first two bits of the next value instead, and puts the remaining bit of this value into the next byte. The overlaps are determined using the following equation:

$$\text{remaining bits per value} = (8 + \text{remaining bits per value}) \bmod bpv. \tag{IV.20}$$

The result 'remaining bits per value' is the number of remaining bits of the current byte that cannot be occupied by a complete value. In the initialisation 'remaining bits per value' is set to zero. Using a DNA 5 alphabet with three bpv the `MaxPacking` method can pack eight values into three bytes. The following example computation of the 'remaining bits per value' for a DNA 5 alphabet shows that in the third byte all bits are occupied with "real" information:

$$\begin{aligned} 2 &= (8 + 0) \bmod 3 \\ 1 &= (8 + 2) \bmod 3 \\ 0 &= (8 + 1) \bmod 3 \end{aligned}$$

There exists also a `NoPacking` strategy which stores each alphabet value in one byte. This method can be used if the run time to serialise the data is more important than the achieved compression.

Data alignment and Endianness

Section 2.4 discussed the issue of data structure alignment. The Binary-Format is used to obtain a high compression factor for the serialised Journal-Set. Therefore, data padding should be avoided which means that each serialised byte contains "real" information. In order to achieve this, each information is written byte-wise using the *Input/Output Stream Library* of the c++ standard library.

Despite of storing char strings for the meta information and the packed insertion strings, the program needs to serialise `integer` values, too. After transmitting the data to another computer, the byte order could have changed due to a differently used operating system. If the byte-order has changed, the `integer` values eventually are differently interpreted by the system. Hence, the information contained in the String-Set becomes worthless. To avoid the eventually false interpretation, the byte-order of all multi-byte values is set to the host-byte-order.

V Results & Discussion

The following chapter presents various performance and compression results for the designed and implemented data structures and algorithms. Each result section will first describe and then discuss the presented results. In the first section of this chapter the test data and the computing system used for the benchmarks are introduced. The next section analysis the performances of the three underlying data structures of the Journal-String. In the third section the bi-affine gap function (see section 4.4 for more information) will be compared with a standard affine gap function and additionally the runtimes for the synchronisation algorithm described in section 4.5 will be shown. The last section analyses and discusses the achieved compression results for genomic sequence data.

5.1 Benchmark Preparation

Computing System

The benchmarks for this thesis were exclusively performed on a computer cluster consisting of two Intel Xeon CPU X5550, each with a 2.67 GHz Quad Core. Altogether, the cluster has 48 GB RAM and a SAS 15K 600GB RAID 0 system[*].

Test Data

In general, two differently sized sequences are used for the benchmarks. First, the genome of the bacteriophage *EC24b* was used for the performance analysis of the Journal-String structures and the analysis of the join and synchronisation method. The genome of EC24b has a size of almost 57 Kb and was downloaded from the ftp site of the "Wellcome Trust Sanger Institute"[†].

The human genome reference assembly *GRCh37* (hg19) published by the Consortium [2009] was the second sequence used for benchmarks. The original unpacked size of hg19 was 3,069 GB. The sequence was supplied chromosome-wise and compressed with "gzip" [Gailly and Adler]. The overall size still amounted to 822,6 MB in compressed form. The human genome was used to test and analyse the compression methods facilitated in the I/O module of the Journal-Set.

[*] see "Knecht" on https://www.mi.fu-berlin.de/w/IT/ComputeServer, received on 2011–06–09
[†] http://www.sanger.ac.uk/Projects/Phage/, received on 2011–06–09

Random Sequence Generator

In order to analyse the efficiency of the Journal-String the test data had to be modified, such that the original and the modified data showed well defined differences. Those differences were then decoded by the Journal-String and the encoding efficiency could be measured. The implemented sequence generator applies artificial insertions and deletions to a reference sequence. In order to do so two parameters could be specified which determined the difference between the Journal-String and the reference in percentage and the maximal length of insertion and deletion operations. The generator randomly added modifications to the reference until the specified level of difference was reached. The kind of an operation as well as the begin position of an operation was randomly chosen according to an uniform distribution. The modified reference is returned as a Journal-String encoding the artificial modifications to the underlying reference sequence.

5.2 Performance Analysis of Journal-Strings

The underlying data structure of a Journal-String is the essential element regarding efficiency. It must allow fast random access, while the encoded sequence is only organised by its differences to a reference sequence. In this thesis the existing data structures using a sorted array and an unbalanced tree are extended by an implementation of a skip list, because of its optimal runtimes for search, insert and delete operations.

Construction Time Performance

In order to test and analyse the construction time of a Journal-String, the EC24b genome was selected and $100,000$ insertions of length ten were respectively inserted at position 0 of the original genome.

To insert the $100,000$ operations the sorted array needed 43.21 seconds. In contrast, the unbalanced tree needed 850.88 seconds and the skip list needed 840.1 seconds. Although, the sorted array needs to shift all elements right of the inserted element, it outperforms the unbalanced tree and the skip list by orders of magnitude. Even though, the unbalanced tree and the skip list can apply an insertion in constant time.

Random Access Time Performance

While the last section analysed the construction times needed by the sorted array, the unbalanced tree and the skip list, this section examines their performance with regard to their random access times. To do so a random number generator was used to randomly pick $500,000$, $1,000,000$, $2,000,000$, $3,000,000$, $4,000,000$ and $5,000,000$ positions. For each experiment, the time to find the elements of the randomly picked positions was measured. Note, that for the runtime analysis a deletion is equivalent to an insertion, such that only insertions were applied. The probability function of the skip list was set to

$$Pr(x) = \frac{3}{4}^{k} . \tag{V.1}$$

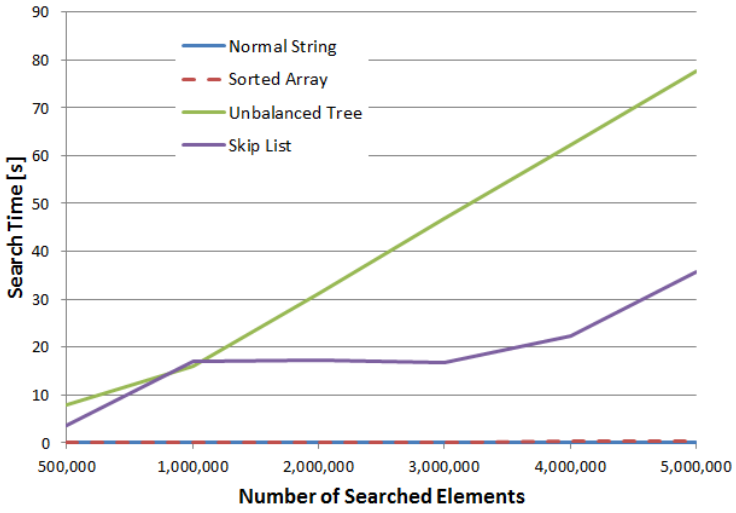


Figure V.1: Comparison of search times of the Journal-String implementations. The x-axis shows the number of searched elements for each experiment. The y-axis shows the search time needed for the different data structures (sorted array: red dashed; unbalanced tree: green; skip list: purple; decoded string: blue).

With this probability function higher values for k are more likely. This step increases the performance of skip lists if many elements are stored because of the higher variation of the different heights.

Figure V.1 shows the search results for the six experiments. The runtimes obtained for the three data structures are compared with the runtimes for searching within a normal string which is not encoded by a Journal-String. The major observation that can be made, is that the sorted array shows the fastest runtimes for searching which are close to the runtimes of searches within the decoded string. The skip list shows better results than the unbalanced tree if more than one million searches are performed. The search time of the unbalanced tree increased linearly with the number of searches. A plateau for one, two and three million search requests within the skip list can be observed, while a linear increase was expected. For higher search requests the search time started to increase again. The skip list, compared to the other data structures, depends on a random variable. Therefore, it could happen that the randomly chosen positions lie on paths through the skip list that can be found faster than others. But this seems unlikely for the huge number of random accesses. The computing cluster on which the experiments were performed was accessible for different research groups and apparently used by other persons during the tests. It is possible that the observed anomaly arose due to different peak times of the computing cluster. Nevertheless, it is necessary to further investigate the cause of the anomaly.

Table V.1: Runtimes for different search experiments of the three data structures sorted array, unbalanced tree and skip list. In addition the runtimes for searching within the decoded string are listed in the row labelled "normal".

# Searched Elements	$0.5 \cdot 10^6$	$1 \cdot 10^6$	$2 \cdot 10^6$	$3 \cdot 10^6$	$4 \cdot 10^6$	$5 \cdot 10^6$
Normal	0.01	0.01	0.03	0.04	0.06	0.06
Sorted Array	0.04	0.06	0.12	0.19	0.25	0.31
Unbalanced Tree	7.97	15.88	31.22	46.81	62.25	77.61
Skip List	3.66	17.13	17.17	16.85	22.43	35.71

Discussion

The basic finding of these experiments is that the simplest data structure, the sorted array, provides the fastest runtimes for insert and random access compared to the unbalanced tree and the sorted array. Moreover, the runtimes of the sorted array are faster by orders of magnitude. The high performance of insert and search operations for the sorted array is strongly related to its simple structure. The unbalanced tree and the skip list are more complex structures that are less efficient in practice than in theory. In general, a skip list is a good alternative to the tree data structure, although the usage of a sorted array seems to be the best choice for fast random access and insert.

5.3 Compression Analysis of the Bi-Affine Gap Function

In order to achieve high compression factors for two sequences a special score function, called bi-affine gap function, was investigated, which extracts an alignment that needs fewest memory requirements if encoded as a Journal-String. The bi-affine gap function was compared with a standard affine gap function. The second score function penalises a gap opening with 2 and a gap extension with 1. A match got the score 0 and a mismatch was not allowed.

Experiments

The EC24b genome and an artificial derivative of it, produced by the sequence generator were used for this experiment. The decoded string of the randomly generated Journal-String was aligned with the EC24b genome using the different score functions. The respective alignment was encoded as a Journal-String and its number of Journal-Entries and the length of the insertion source determined. Table V.2 collects the observed data for each experiment, while figure V.2 displays the corresponding trend of the corresponding compression factor. In each experiment the level of difference between the EC24b genome and the artificial sequence was increased, while the average sequence length of the applied operations persisted 5, so the number of Journal-Entries increased in each experiment. The orange dashed line represents the compression factor received using the bi-affine gap function. The blue line shows the compression factor observed for the standard affine gap function. The y-axis represents the achieved compression factors. The different experiments are shown on the x-axis.

Table V.2: Comparison of score functions for maximal similarity and maximal compression. The first row shows the level of difference in percentage. The second row shows the sizes of the generated sequences. Rows three to five show values of the new Journal-String referring to EC24b based on an alignment that computes the maximal similarity. Rows six to nine show values of the new Journal-String referring to EC24b based on an alignment that computes the maximal compression.

Level of Difference		1%	2%	5%	10%	20%	50%
Old size[B]		57,679	57,714	57,580	57,312	57,646	57,626
Sim	Size [B]	2,084	4,457	11,135	21,306	42,442	102,294
	# Entires	150	322	816	1,565	3,108	7,623
	Insert Length	284	593	1,343	2,526	5,146	10,818
Comp	size [B]	2,060	4,415	10,399	19,190	33,722	53,941
	# Entires	147	315	732	1,268	1,778	779
	Insert Length	296	635	1,615	3,974	12,386	44,593

The data in table V.2 show, that the number of Journal-Entries for the maximal compressed string is always smaller than the number of Journal-Entries for the string with the maximal similarity. On the opposite, the length of the insertion source is always higher. This observation corresponds to the fact that the bi-affine gap function penalised the opening of a new Journal-Entry much harder than the extension of it. Hence, insertions that are closely situated to each other within the alignment were combined to one single insertion. The standard gap function is more sensitive to matching regions, such that the same scenario could result in three Journal-Entries to cover both, the insertions and an eventually the occurred matches between them. This led to smaller insertion infixes but also to an increased number of Journal-Entries which require more memory, resulting in lower compression factors.

Discussion

The experiments clearly show that the compression factor received by the bi-affine gap function always exceeded the compression factor received by the standard affine gap function. Moreover, the higher the difference between the aligned sequences, the higher was the gap between the compression factors of both methods. For a sequence difference of 50% the compression factor received by the bi-affine gap function was still greater than one. In contrast the Journal-String with the maximal similarity needed more memory for the encoding than for the original string, such that a negative compression was observed. Thus, if the maximal compression of the sequences is required, then it is highly recommended to use the bi-affine gap function optimised for the data compression.

5.4 Performance Analysis of the Synchronisation Algorithm

In some cases it might be of interest to exchange the reference string of a group of Journal-Strings. Such scenarios occur, if, for instance, a more recent genome assembly with new sequence information is available, or the reference that has the highest similarity to all other

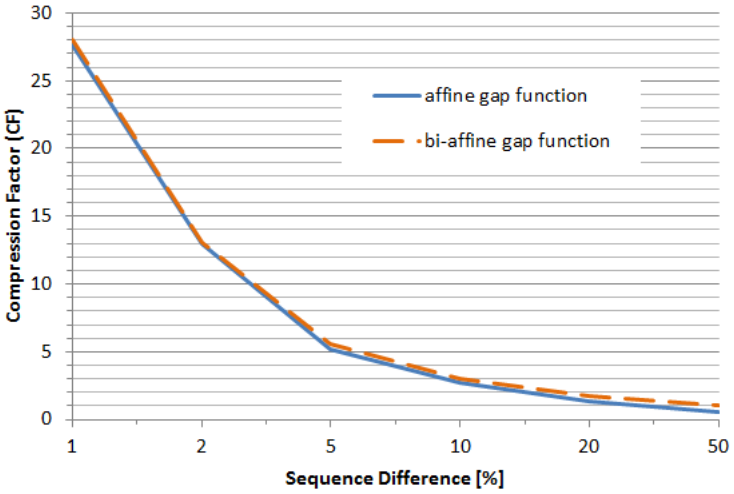


Figure V.2: Comparison of compression factors.

Journal-Strings within the group shall be selected, in order to save additional memory. In such cases, it would be necessary to recompute the alignments for each affected Journal-String with the new reference string. To circumvent this, an algorithm was developed that synchronised each affected Journal-String with the alignment between the old and the new reference. Therefore, only one alignment has to be computed, while the synchronisation only updates the Journal-Entries of the affected Journal-Strings without comparing any character.

Experiments

In this experiment the genome of EC24b was chosen as the basic reference string. In the first step, a second reference string was randomly generated with an average operation length of 5 bases. As in the previously described experiment, six different levels of difference to the EC24b genome were selected. Additionally, 1,000 Journal-Strings, based on the randomly generated reference, were generated with the same settings. These 1,000 Journal-Strings were collected into one group and the randomly generated string was exchanged with the genome of EC24b. Afterwards, an alignment between the EC24b sequence and the randomly generated reference was computed with the bi-affine gap function and the standard gap function presented above. Then, the 1,000 Journal-Strings were synchronised to the new reference string.

The alignment of both reference strings took 6.82 minutes on average. Figure V.3 shows the runtimes for synchronising the 1,000 Journal-Strings for each score function and experiment setting. The levels of difference are displayed on the x-axis. The orange dashed line represents the runtimes obtained from the experiments using the bi-affine gap function. The blue line represents the runtimes for the experiments using the standard affine gap function. The y-axis represents the synchronisation times in seconds.

The data show, that even for very distinct sequences 1,000 Journal-Strings could be synchronised in almost 10 seconds for references aligned with the bi-affine gap function and in

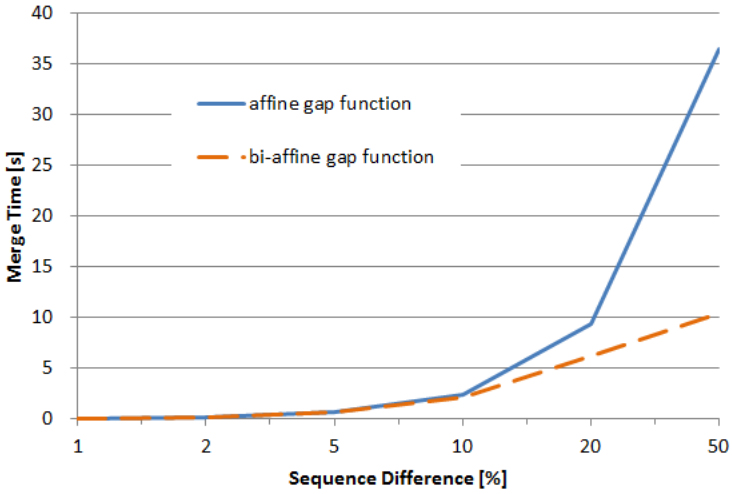


Figure V.3: Comparison of merge times for 1,000 sequences.

almost 36 seconds for references aligned with the standard affine gap function.

Discussion

The synchronisation is a very good alternative to the naive alignment approach. Compared to the time needed to compute the alignment between the reference strings the synchronisation is faster by orders of magnitude, because the algorithm compares entire subregions instead of evaluating each single character. But the synchronisation does not guarantee the best alignment. This might be of interest especially when trying to find a reference that has the highest similarity to all other sequences. To find a better synchronisation by means of the best alignment, the algorithm could be extended by partial alignments within segments that eventually match. Such regions occur for example if an insertion of the outer tree is compared with a deletion of the inner tree. Then, the computed alignment would be more closely oriented to the best alignment. Obviously, this might slow down the algorithm, but assuming similar sequences, this might only be a slight difference compared to the performance obtained without this strategy.

5.5 Performance and Compression Analysis of Data Serialisation

Storing only the differences of sequences to a reference sequence comes along with an enormous advantage in reduced memory requirements. However, the previous results showed that a certain similarity between the sequences and the reference must exist to obtain a compression gain. Fortunately, most biological sequences show low diversities between individuals of the same population as well as the same species. As mentioned in the introduction, human genome sequences seem to differ only in one percent of their genetic code. Hence, the expec-

ted compression factor for a human individual should be very high if only its differences are stored.

Experiments

For this experiments hg19 was selected and modified through artificially applied deletion and insertion operations. These artificial differences covered one percent of the original size of the gnome. For the experiments the DNA 5 alphabet consisting of 'A', 'C', 'G', 'T' and 'N', was used due to unknown regions within the sequence of hg19. Altogether, five experiments with altered operation lengths, capturing 5, 25, 50, 250 and 500 bases (b) on average, were performed. In addition the set containing the alternated genome was written to secondary memory, while the described bit packing strategies ("No Bit Packing", "Simple Bit Packing" and "Maximal Bit Packing") were compared.

Figure V.4 and V.5 shows the sizes of the serialised genome hg19 derivatives. The x-axis in both figures shows the average operation lengths. For each average operation length the three bit packing methods are compared. The left y-axis displays the sizes of the files in bytes (B) and the right y-axis displays the numbers of Journal-Entries. The blue bar (left) represents the standard method using no bit packing. The red bar (middle) represents the simple bit packing strategy and the green bar (right) represents the strategy using maximal bit packing.

The first simple observation that can be made, is that with an increased average operation length the number of Journal-Entries decreased and also the size of the compressed files became smaller. If an average length of 5 bases was entered, the number of Journal-Entries necessary to store the operations was almost $8,5$ million. The size of the unpacked format was round 60 MB, while the maximal bit packing strategy could save 10 MB, which is 2 MB less than the simple bit packing strategy could achieve for the same data set. In addition, the plot shows an exponential decrease over the different average lengths, which drives into a saturation of the sizes of exported Journal-Sets. The size of the unpacked Journal-Set converged to almost 16 MB, the simple bit packing approach converged to around 8 MB and the maximal bit packing strategy converged to around 6 MB. This saturation originates from the size of the insertion string which has to be stored within the files.

As shown in figure V.6 the blue bar converged to a size of $16 \cdot 10^6$ Bytes over the different experiments, while the number of Journal-Entries decreased. Due to the uniform distribution of the operations almost $15 \cdot 10^6$ bases were inserted (0.5% of the original genome size) in each experiment.

Another observation is that the relative difference between the compared packing methods for each experiment was constant which is also attributable to similar sizes of the insertion strings in each experiment.

Figure V.5 shows the same experiments as figure V.4 except for an additional compression of the exported data using gzip. Again, the three packing techniques are compared for each experiment with another average operation length. The first main difference, compared to the results in V.4, is the similar size of the compressed data sets. Furthermore, gzip could achieve a higher compression for the unpacked data set than for the bit packed data sets. While the "gzipped' version of the maximal bit packing approach resulted in a size of $25,636,828$ bytes,

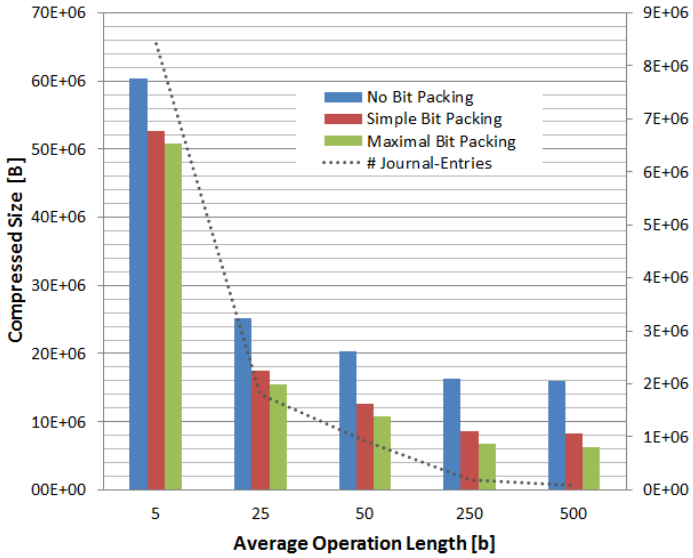


Figure V.4: Compression of hg19 reference genome. The x-axis represents the experiments with different average operation lengths. The left y-axis represents the sizes of the serialised data in bytes and the right y-axis represents the number of Journal-Entries. For each experiment the three bit packing strategies are compared.

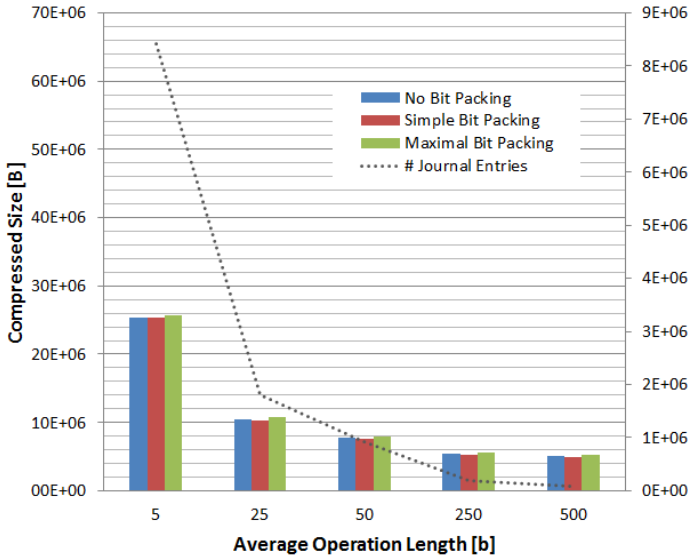


Figure V.5: Compression of hg19 reference genome using gzip in addition. The x-axis represents the experiments with different average operation lengths. The left y-axis represents the sizes of the gzipped, serialised data in bytes and the right y-axis represents the number of Journal-Entries. For each experiment the three bit packing strategies are compared.

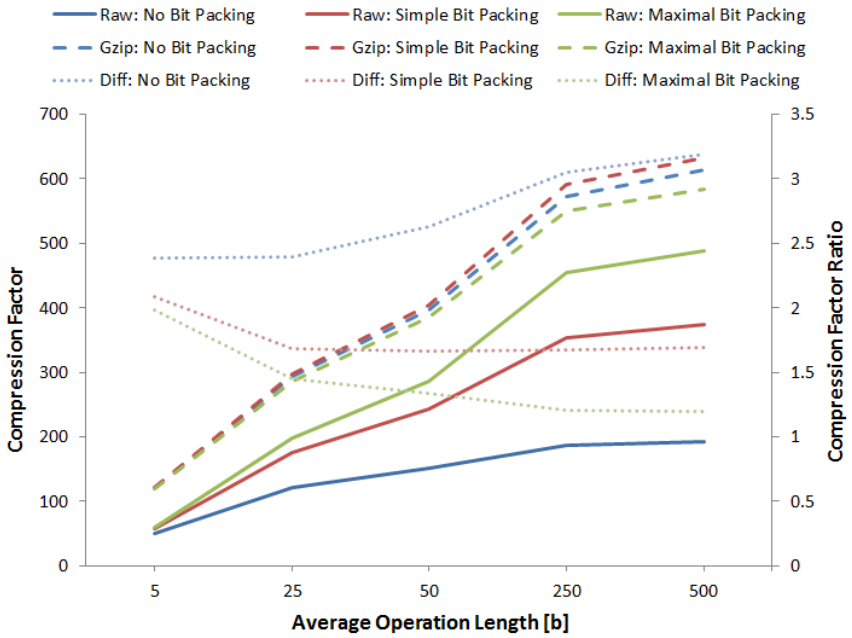


Figure V.6: Comparison of the compression factors. The x-axis represents the experiments with different average operation lengths. The left y-axis represents the compression factor. The unpacked size of the reference genome was used as input string. The solid lines represent the unzipped data and the dashed lines the zipped data. On the right y-axis is shown the ratio of the compression factor between the raw sizes and the "gzipped" sizes.

the "gzipped" version of the unpacked string was slightly lower with a size of 25,363,557 bytes. Even if the difference does not play a significant role in the overall size of the compression, the time needed to compress the files could be relevant. The compression factor and the differences between the raw Journal-Set formats and the "gzipped" versions as well as the time needed to serialise the data are discussed in the following.

Figure V.6 shows the compression factors for each experiment. The blue lines indicate the compression factor of the unpacked serialisation strategy. The red lines represent the simple bit packing approach and the green lines represent the maximal bit packing approach. Again, all data are related to the average operation length displayed on the x-axis. The left y-axis represents the compression factors achieved by the respective method based on the unpacked genome sequence as the input string. The right y-axis represents the compression factor ratios between the raw format and the "gzipped" versions for each experiment. The solid curves represent the compression factors of the raw data sets and the dashed lines the compression factors of the "gzipped" files. The dotted lines represents the ratios of the compression factors.

The primary observation is that for each packing strategy the compression factor increased with the operation length. At the same time the compression factor strongly depended on the number of Journal-Entries. The compression factors diverged more with less Journal-Entries. These observations can be reasoned by the fact that the huge number of the Journal-Entries

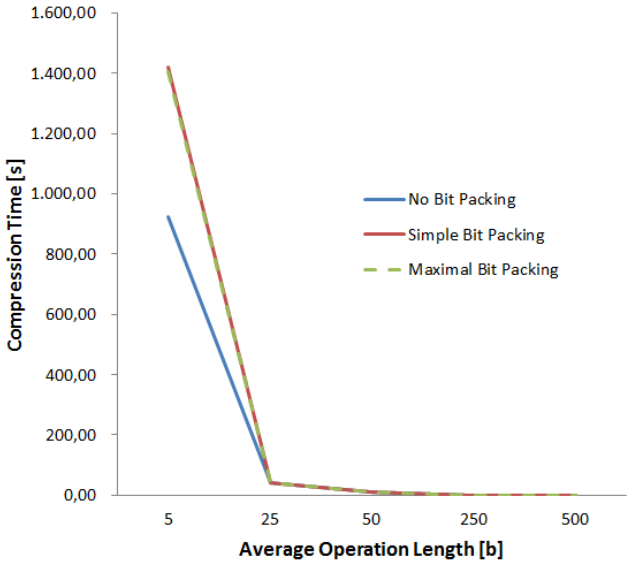


Figure V.7: Serialisation times. Shown are the run-times of the implemented compressor needed to export the data sets for each experiment.

in the first experiment (5 bases average operation length) outweigh the size of the insertion string. The Journal-Entries are stored identically in each packing strategy. With increased average length of the applied operations the number of Journal-Entries was reduced, such that the packing of the insertion string became more important.

These big differences between the unzipped formats were neutralised by the additional gzip compression. Using gzip after the serialisation led to almost the same compression factor for each packing strategy. Figure V.6 shows that gzip gained better compressions for the simple bit packing and the unpacking strategies than for the maximal bit packing method.

These differences between compression factors of the unzipped formats and the "gzipped" formats are elucidated by the ratios between the compression factors for each packing strategy (coloured dotted lines). It shows that the gzip tool used to compress the raw format increased the compression factor for the unpacking strategy up to a multiple of 3.2, while the packing strategies could only be increased by a multiple of almost 2. In addition, the plots reveal that gzip could compress unpacked sequences much better than the packed strings. The more the insertion sequence became the deciding factor for the size of the data set, the less gzip could improve the compression factor for the packed strategies and the better this tool could improve the compression factor for the unpacked string.

Obviously, the compression results promote the usage of the unpacked strategy in combination with gzip, also if there were only minor enhancements. This raises the question which strategy has the better performance. Figure V.7 represents the times needed to store each Journal-Set for each experiment. It clearly shows that only negligible differences existed between both packing strategies. For the average operation length of 5 bases the unpacked

approach performed 7 minutes faster, than the other two packing strategies. It also shows that with decreased number of Journal-Strings the time needed to export the data was reduced rapidly to only a few seconds. The gzip tool needed almost the same time period for each experiment and packing strategy, such that it does not significantly influence the relative difference of the shown runtimes.

Discussion

The results show that the size of a genome (3 GB uncompressed) could be greatly compressed using the journaling approach as discussed in the previous chapters. The compression factor strongly depends on the number of Journal-Entries needed to store the differences. In the experiments the size of the operations was uniformly distributed. In real genomic data the number of single nucleotide polymorphisms is expected to be higher than the number of longer insertions or deletions [1000 Genomes Project Consortium, 2011]. Nevertheless, the insertions and deletions can reach lengths up to several kilo bases [1000 Genomes Project Consortium, 2011]. Hence, the described results are a good measure for the maximal and minimal compression factors expected for real genomic data. The more longer operations are contained within the data, the higher is the expected compression factor and the smaller is the compressed data set. The average size over all experiments for the "gzipped" version is round $10,700,000$ bytes, which results in an average compression factor of around 280. Thus the average entropy of the experimental data accounted to 0.0036 bpc. Since, the experimental data show a similar exponential behaviour as it is expected for real genomic data, the computed average numbers indicate the expected outcome of real data sets.

Furthermore, the experiments show that a usual compression tool like gzip gains some additional compression for the serialised data. Gzip revealed better compression results for the unpacked strategy, than for the packed approaches. This results from the fact that the four DNA values are uniformly distributed within the insertion buffer. In addition, the maximal bit packing method fills all bits of a byte as long as there are values to compress, such that there are barely long stretches of "ones" or "zeros". Obviously, storing one value in a single byte could be better facilitated by gzip when the values are uniformly distributed.

It is also shown that the "gzipped" data sets had almost the same sizes. Therefore, further compression techniques are required in order to outperform standard compressors like gzip. There exists a strategy that can pack three DNA 5 values in a single byte. Note, that a byte can occupy 256 different values. But there exists only 155 different combinations for five values at three positions. Let the values of the DNA 5 alphabet be assigned with the following values: $A = 1$, $C = 2$, $G = 3$, $T = 4$ and $N = 5$. Then, three values can be encoded by the following formula:

$$i = x^1 + y^2 + z^3, \tag{V.2}$$

with $x, y, z \in \Sigma^5$. In order to decode the correct values, one could use a map for each combination, such that the corresponding triplet of values can be accessed in $O(1)$.

Christley *et al.* [2009] discussed additional ways to compress the `integers` of the insertion and deletion offsets. Such methods could be used to gain higher compression results especially if many Journal-Entries are stored. Unfortunately, they provided no time measurements for

their compression approaches. Thus, their compression methods must be examined towards the required compression time. It is desired to have fast compression run-times, especially when dealing with multiple sequences.

In general, the journaling approach is a very efficient strategy to compress the size of a genome and also of multiple genomes. The compressed genome with a size of 10 MB could be transmitted via E-Mail in almost one minute. Despite this huge success, the currently existing compression techniques must be extended in order to increase the compression factor, such that similar transmission times can be expected for multiple genomes.

VI Conclusion

Currently, the colossal advancements within next-generation sequencing technologies demand for novel space-saving sequence formats and algorithmic approaches that can rapidly analyse plenty of sequences.

The presented work describes the initial steps towards handling the enormous growth rate of sequences. In addition, this work suggests sophisticated data structures that serve as foundation for swift sequence analysing methods to meet the requirements of modern research institutions and of booming biochemical and pharmaceutical companies.

This concept is realized in SeqAn, providing a data structure referred to as Journal-String that only focused on the efficient management of one sequence to a reference sequence. Journal-Strings have to be generated manually and one can not efficiently manage several sequences with regard to a single reference. I will overcome these shortcomings by:

In this thesis I successfully accomplished the implementation of tools and algorithms necessary to efficiently work with multiple biological sequences encoded as space efficient Journal-Strings. I designed and developed elaborate methods to obtain a maximal space efficiency for a Journal-String in main memory and in secondary memory. Therefore, I implemented a special score function that computes the maximal compression based on a pairwise sequence alignment and I used different compression techniques to increase the compression when writing the data to secondary memory. Based on the implemented methods I compressed an artificially generated human genome to around 10 MB from originally 3 GB and transmitted it via E-Mail.

As the efficient management of the sequence data is required to be useful for large sequencing ventures, I engineered several algorithms optimised to work on Journal-Strings. One of the essential algorithms was the synchronisation approach, which aligns two Journal-Strings based on their encoded differences to a common reference. This approach was concretely utilised for the exchange of a reference of a group of Journal-Strings. I showed that 1,000 sequences with an average size of around 58 Kb could be aligned in almost 10 seconds. Furthermore, I suggested a way to improve the described algorithm to obtain better alignment results. Next to this major finding, I designed and implemented a method to generate a Journal-String out of a pairwise sequence alignment.

I also extended the Journal-String derivatives by a third structure, a skip list, that provides optimal asymptotic runtimes for search, insert and delete operations. I showed that this struc-

ture is a good alternative to the unbalanced tree version, although the results showed that the version using the sorted array outperforms the other structures.

Given that this piece of work provides an initial glimpse in this field of bioinformatics, only simple alignment algorithms were supported, in order to issue both the general problems and approaches of the designed data structures. In the future, these algorithms must be extended by more complex approaches which are well suited for entire genomes. Such algorithms, for example BLAST [Altschul *et al.*, 1990], LAGAN [Brudno *et al.*, 2003a] or Shuffle-LAGAN [Brudno *et al.*, 2003b] compute local matches and extend these with elaborate methods.

A first prototype of an algorithm generating a Journal-String out of a whole genome sequence alignment was implemented in the last stages of this thesis. Therefore, the recently developed *STELLAR* algorithm[*] within SeqAn was used as the basic genome alignment tool.

In general, this thesis successfully accomplished the first milestone of a project that aims to respond to the described increase of genomic sequence data with algorithmic approaches that benefit from redundancies across multiple datasets. In the following of this work I will engineer enhanced algorithms that are based on this work to process multiple sequences in parallel. Therefore, the aggregation of huge parts of many sequences eminently benefits the parallelisation approach as identical parts need to be processed only once for each aggregated sequence.

Furthermore I am going to extend the existing data structures and methods with algorithm that dynamically build an index over Journal-Strings. Indices are widely used structures in order to enhance the speed of algorithmic approaches such as read mappers [Rumble *et al.*, 2009; Weese *et al.*, 2009; Langmead *et al.*, 2009]. Similar to the applied concept of the Journal-Strings the suggested index method builds only the index over differences to a reference, while it utilises a previously build index of this reference. Furthermore, those indices could be serialised to secondary memory, too. In this case, one could save and load entire sequence databases without rebuilding indices or sequence information, so the purposed analysis methods could be executed instantly.

[*] http://www.seqan.de/projects/stellar.html, received on 2011–06–12

Appendix

Supplementary material, algorithm pseudo-codes and bibliography.

Algorithms

A

In the following, some algorithms used for this thesis are explained and their pseudo codes given.

A.1 Alignment Algorithms

Needleman and Wunsch [1970] constructed a dynamic programming algorithm to compute the best global alignment between two strings. The computation matrix is denoted with M and the tracback matrix with T. Both matrices have a dimension of $(n+1) \cdot (m+1)$, because one row and one column is added for initialisation purposes. A gap is penalised with the function γ and σ returns the score for either a match or substitution. The best score is stored in s. The pseudo code A.1 shows the pairwise alignment algorithm. Since $(n+1) \cdot (m+1)$ cells have to be computed, the running time and space consumption require $O(n \cdot m)$, which is quadratic in the length of the sequences.

An advancement of the Needleman-Wunsch algorithm is the introduction of a k-band (the corresponding pseudo code is listed in A.2). This approach makes sense for similar sequences, because it is expected that the best alignment will be somewhere around the main diagonal of M. It could happen that the initially chosen k is to small and the best alignment leaves the band at some position. The pseudo code A.3 demonstrates a wrapping which finds the optimal alignment by adapting the size of k during the computation. The traceback is omitted since it equals the described traceback in the pseudo code A.1.

In the following it is shown, that the k-band algorithm using the wrapping code is bounded by a factor $\Delta = M \cdot n - s$. Therefore, it is assumed for simplicity that the two aligned sequences a and b both have a length of n. M denotes an uniform match score and d denotes the gap penalty. The optimal score of the best alignment using Needleman-Wunsch algorithm is stored in s. The best alignment obtained by the k-band algorithm is denoted as s_k. Then the runtime of the k-band algorithm is $O(\Delta \cdot n)$. The proof is divided into two parts. The first part shows under which conditions s_k equals s. The second part shows that there exists an upper bound for which the k-band algorithm terminates.

First it is proven, that if $s_k = s$ the following must hold:

$$s_k \geq M \cdot (n - k - 1) - 2 \cdot (k+1) \cdot d. \tag{A.1}$$

Proof. If there exists a best alignment with an optimal score s that does not leave the band for the current k, then obviously $s_k = s$. In all other cases the best alignment leaves the band somewhere. This requires the insertion of at least $k+1$ gaps in one direction to leave the band. To return back to the diagonal there are again $k+1$ gaps necessary. Hence, the best score s_k is at least penalised $2 \cdot (k+1) \cdot d$. This also implies that there can be at most $n-k-1$ matches. Hence, the best score s_k can only be $M \cdot (n-k-1) - 2 \cdot (k+1) \cdot d$. If s_k is greater or equal this score, then the optimal alignment s lies within the band k. □

Proof. As previously shown, the best global alignment for a band k terminates, if

$$\begin{array}{rcl} s_k \geq M(n-k-1) - 2(k+1) \cdot d & \Leftrightarrow \\ s_k - M \cdot n + M + 2 \cdot d \geq -(M + 2 \cdot d) \cdot k & \Leftrightarrow \\ -s_k + M \cdot n - (M + 2 \cdot d) \leq (M + 2 \cdot d) \cdot k & \Leftrightarrow \\ \frac{M \cdot n - s_k}{M + 2 \cdot d} - 1 \leq k \end{array} \tag{A.2}$$

At this point the complexity accounts still to $n + 2n + 4n + 8n + ... + kn \leq 2kn$, which is not better than $O(nm)$. The goal is to find a bound on k, such that the overall complexity is bounded.

It is clear, that if the best global alignment lies within k, the previous score s_k was less than s. Using the equation of A.2, then the following holds for $k/2$.

$$\frac{k}{2} < \frac{M \cdot n - s_{k/2}}{M + 2 \cdot d} - 1. \tag{A.3}$$

Two different cases must be considered.
Case 1: If $s_{k/2} = s_k = s$, then

$$k < 2 \cdot \left(\frac{M \cdot n - s}{M \cdot 2 \cdot d} - 1 \right). \tag{A.4}$$

Case 2: If $s_{k/2} < s_k = s$, then any best global alignment must leave the band somewhere and thus has at least $k/2 + 1$ gaps, such that $s < Match(n - k/2 - 1) - 2(k/2 + 1) \cdot d$. Based on the equation in A.2, this is the same as:

$$k < 2 \cdot \left(\frac{M \cdot n - s}{M + 2 \cdot d} - 1 \right). \tag{A.5}$$

Since $Match + 2 \cdot d$ is a constant and $\Delta = M \cdot n - s$, it follows that k is bounded by $O(\Delta)$ and thus the total bound is $O(\Delta \cdot n)$. □

Input: two strings a and b
Output: optimal alignment plus the corresponding score

Initialisation:
$M(0,0) \leftarrow 0$
$M(i,0) \leftarrow -i * d$ and $T(i,0) \leftarrow (i-1,0)\ \forall i \in [0,\dots,n]$
$M(0,j) \leftarrow -j * d$ and $T(j,0) \leftarrow (0,j-1)\ \forall j \in [0,\dots,m]$

Recurrence:
for i = $1,\dots,n$ **do**
    for j = $1,\dots,n$ **do**

$$M(i,j) \leftarrow max \begin{cases} M(i-1,j-1) & +\sigma(a_i,b_j) \\ M(i,j-1) & -d \\ M(i-1,j) & -d \end{cases}$$

        $T(i,j) \leftarrow$maximizing pair (i',j'); encoded as $(\leftarrow, \nwarrow, \uparrow)$
best score $s \leftarrow M(n,m)$

Traceback:
repeat
    if $T(i,j) = (i-1,j-1)$ **then**
        print $\binom{a_i}{b_j}$
    else if $T(i,j) = (i-1,j)$ **then**
        print $\binom{a_i}{-}$
    else
        print $\binom{-}{b_j}$
    $(i,j) \leftarrow T(i,j)$
until (i,j) = 0

Algorithm A.1: Needleman-Wunsch.

A.2 Search Algorithms

The essence of the different implemented Journal-String specialisations is that each provides a search in $O(\log g)$ time, with g being the number of Journal-Entries in the Journal-String in best-case. In worst-case the unbalanced tree has a runtime of $O(n)$, while the other structures remain optimal. The specialisation based on the sorted array uses the binary search, the unbalanced tree only has optimal search times if it is completely balanced. In this optimal case the binary search tree supports the optimal runtime due to its composition. The skip list provides a search in logarithmic runtime as well. In the following the algorithms are explained in pseudo code as well as some sample illustrations for each search approach are prospected.

```
1  Input: two strings a and b
2  Output: optimal alignment at most k diagonals away from central diagonal plus the corresponding score

3  Initialisation:
4  M(0,0) ←0
5  M(i,0) ←−i ∗ d and T(i,0) ←(i − 1,0) ∀i ∈ [0,...,k]
6  M(0,j) ←−j ∗ d and T(j,0) ←(0,j − 1) ∀j ∈ [0,...,k]

7  Recurrence:
8  for i = 1,...,n do
9      for h = −k,...,k do
10         j ←i + h
11         if 1 ≤ j ≤ n then
12             M(i,j) ←M(i − 1,j − 1) + σ(a_i, b_j)
13             if insideBand(i-1,j,k) then
14                 M(i,j) ←max(M(i,j), M(i − 1,j) − d)
15             else if insideBand(i,j-1,k) then
16                 M(i,j) ←max(M(i,j), M(i,j − 1) − d)
17 return M(n,m), s_k

18 insideBand(i,j,k) ←−k ≤ i − j ≤ k
```

Algorithm A.2: k**-band algorithm.**

```
1  Input: two strings a and b
2  Output: optimal alignment at most k diagonals away from central diagonal plus the corresponding score

3  Initialisation:
4  k ←1
5  repeat
6      compute s_k using k-band algorithm
7      if s_k ≥ Match(n − k − 1) − 2(k + 1) · d then
8          return s_k
9      k ←2 · k
10 until
```

Algorithm A.3: Wrapper of k**-band algorithm.**

Sorted Array: The binary search algorithm, recursively cuts the search space in two smaller intervals in each step, until the searched item is found. The pseudo code A.4 shows the corres-

```
1  Input: sorted array sa, begin, end, vp
2  Output: upper bound to vp

3  Searching:
4  while begin < end − 1 do
5      middle ←[begin + (end − begin)/2]
6      if (vp < sa[middle].vp) then
7          end ← middle
8      else if sa[middle].vp < vp then
9          begin ← middle
10     else
11         return sa[middle]
12 return sa[middle]
```

Algorithm A.4: Binary search on a sorted array.

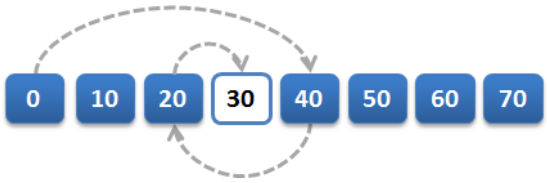


Figure A.1: Binary search within a sorted array.

ponding steps. The input is a sorted array sa as well as the begin and the end position of the respective search space. Additionally, the target element is given, which is the virtual position vp to be found. The algorithm picks the middle element of the initial interval as the pivot element in line 5. If vp is smaller than the virtual position of the element at the middle position of sa, then the right border of the search space is updated to the pivot element, otherwise the left border is updated (in lines 6−9). If the element is not found yet and the termination constraint in line 4 is not fulfilled, then the next pivot element of the cut search space is selected.

Figure A.1 demonstrates the binary search on a sorted array of journal entries. Given is a sorted array containing eight Journal-Entries. Note, that Journal-Entries are displayed as abstract boxes labelled with their virtual position. The length of each element is 10 and the physical positions are omitted as they does not influence the search algorithm. Let $vp = 25$, $begin = 0$ and $end = 8$. The search first selects the fifth element at position 4 as the pivot element. Since 25 is smaller than 40 and end is set to 4. In the next step the third element at position 2 is selected. $20 > 25$, hence $begin = 2$. In the last step the fourth element is picked and the left border updated accordingly. Now the inequality $begin < end - 1$ does not hold any longer, and the element with $vp = 30$ at the third position within sa is returned (white box with blue outline), which is the upper bound of the searched element.

```
1  Input: root, vp
2  Node containing vp

3  Searching:
4  r←root
5  while searching do
6      if vp < r.vp then
7          if hasLeft(r) = 0 then
8              return -1
9          goLeft(r)
10     else if result.vp + r.l ≤ vp then
11         if hasRight(r) = 0 then
12             return −1
13         goRight(r)
14     else
15         return r
```

Algorithm A.5: Binary tree search

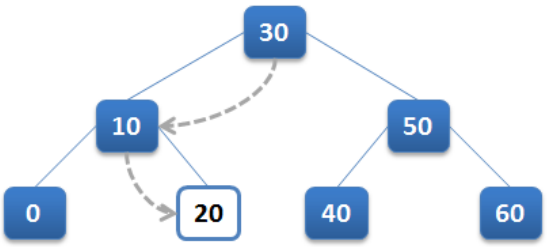


Figure A.2: Search within an unbalanced binary search tree.

Unbalanced Tree: The pseudo code in A.5 shows the search algorithm for a Journal-String implemented as an unbalanced tree. The search begins with the root (line 4). Then for each visited node, the search continues in the left subtree or the right subtree, depending whether $vp < r.\text{vp}$ or $vp > r.\text{vp} + r.\text{l}$, respectively. Otherwise the found node is returned.

Figure A.2 demonstrates the search on the binary search tree. In the case of $root$ being equal to the node with virtual position 30 and the requested virtual position $vp = 25$, the first check reveals that the searched position is somewhere in the left subtree. Hence, the node with virtual position = 10 is selected. Since the searched element is greater than the last position of this Journal-Entry, the searched value must be somewhere in the right subtree. The next selected node contains the searched vp.

Skip List: The skip list is a probabilistic data structure that is bounded by $O(\log n)$ operations necessary to find an element within the structure. The search algorithm is explained in pseudo code in A.6. The algorithm is passed with the maximal height h of the skip list. Recall, that

```
1  Input: S_h, vp
2  Journal-Entry containing vp

3  Searching:
4  i ←h
5  while element containing vp not found do
6      if hasRight(Si) then
7          goRight(S_i)
8          if S_i.cargo.vp + S_i.cargo.l > vp then
9              if S_i.cargo.vp ≤ vp then
10                 return S_1.cargo
11             goLeft(S_i)
12             goDown(S_i)
13     else if hasDown(S_i) then
14         goDown(S_i)
15     else
16         return end of skip list
```

Algorithm A.6: Skip list search

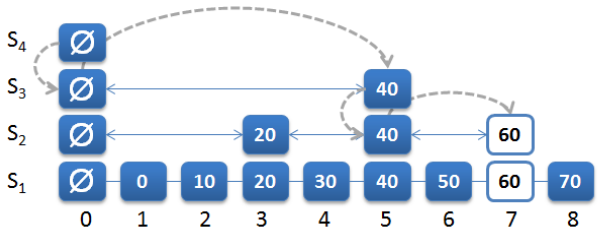


Figure A.3: Search within a skip list.

the list S_h contains only the element ∅.

The search proceeds with the right neighbour that is closest to the current element within S_i (line 7). If there is no right neighbour for the current list, then the search carries on with the same element in the list below S_{i-1} (lines 6, 13, 14). In the other case, if the the searched vp is contained within the currently examined Journal-Entry, then the corresponding entry is found and returned. Note the `cargo` function access the current Journal-Entry of the list S_i. In the other case, if the vp is less then the virtual position of the current Journal-Entry, the search carries on with the last element one list below in S_{i-1} (lines 11, 12). If the element is not contained within the list, then -1 is returned.

Figure A.3 demonstrates the search of the virtual position 62 on the skip list data structure. The search begins in the list S_4 at position 0. Since this list only consists of the value ∅, it has no right adjacent by definition. Hence, in the first step the algorithm proceeds with the list S_3 at the same position. Then the next right element is selected, which is the sixth element of

the S_1 list. Here the inequality $40 + 10 \geq 62$ is false, hence the algorithm tries to go further right. In the example the first four entries are skipped, without missing a possible hit. But there is no element in S_3 at position 5, hence the the algorithm steps to the S_2, the list below. Then the next right neighbour is chosen. Here, the inequality $60 + 10 \geq 62$ is true, such that either a hit or a upper bound is found. In this special case the second inequality $60 \leq 62$ is true, hence the Journal-Entry covering the virtual position 62 is found and can be returned.

C++ & SeqAn

B

The proposed data structure and the accompanied algorithms were implemented in SeqAn [Döring *et al.*, 2008] which is an acronym for **Seq**uence **An**alysis. SeqAn is an efficient C++ [Stroustrup, 1997] based library that provides various bioinformatical sequence analysis algorithms and data structures. In order to accomplish the implementation of the data structures and the algorithms the principles of SeqAn have to be well understood. SeqAn is optimized towards fast analysis of sequence data. It is well-suited for the implementation of the data structures and methods designed in this thesis, since it provides some of the described methods and approaches.

The basic principles of SeqAn are the generic implementation of the algorithms, the concept of meta programming to generally apply the correct methods and values of objects during the compile time and the global call of functions which allows for dynamically usage of the library. Despite the fact that C++ was originally the extension of C considering novel concepts for "object oriented programming", it also came across with the concept of "template programming". Template programming perfectly benefits the generic interface of SeqAn, since abstract objects and methods can be defined that perform well with different data types without implementing a specialisation for each of them.

The listing B.1 demonstrates the usage of the template programming. The function returns the reference to the element with the maximal value. Both elements are of type 'T' which is a abstraction and can be used for any data type that implements the operator ">=". Based on the data types of the call values of `max` the compiler puts the correct data types into 'T'. Listing B.3 shows three different ways of how the function `max` can be called. Line 3 shows the *implicit* way. 'T' is set to `int` since the given elements are of type `int`. In contrast, line 4 shows the *explicit* call of the same function. In this case 'T' is set to `float` and the given elements are treated as such data types. The call of `max` in the 6-th line would cause a compiler error, since the data type of the second value does not match 'T' which is implicitly set to an `int` data type because of the first value. The solution would be the declaration of a `max` function as shown in B.2. Here the function is extended with the second template parameter 'T2'. Note that the return value is of type 'T1', since it cannot be set dynamically during the function call. Hence the returned value is eventually casted to the return value type. Line 7 in the listing B.3 demonstrates the call of the extended max function.

```
1  template <typename T>
2  const T max(const T &valA, const T &valB)
3  {
4          return (valA >= valB) : valA ? valB;
5  }
```

Listing B.1: C++ code for a generic max function.

```
1  template <typename T1, typename T2>
2  const T1 max(const T1 &valA, const T2 &valB)
3  {
4          return (valA >= valB) : valA ? valB;
5  }
```

Listing B.2: C++ code for a generic max function.

```
1  int A = 10;
2  int B = 6;
3  max(A,B);
4  max<float>(0.1, 4);
5  float C = 3.76;
6  max(A,C);       //compiler error

8  max(A,C)        //works fine
```

Listing B.3: C++ code for calling the generic max function.

This examples also demonstrates another feature of C++ which is called template deduction. Template deduction refers to a special behaviour causing a function call of the function which best matches the given input types. For example, the call of `max` in line 3 of the listing B.3 jumps into the `max` function declared in listing B.1, because both values have the same type and thus the first declaration of `max` is more specific than the second declaration.

Another approach used in SeqAn is the principle of meta programming. Using this programming approach values or data types are set at compile time, allowing faster access during the execution, since the program already knows which function, values or objects shall be called or initialised.

This thesis also explains the term of bit packing. Listing B.4 shows how the information of how many bits per value of a certain alphabet in SeqAn is retrieved at compile time. The most unspecific declaration is the first one. If any other value than `Dna` or `Dna5` is used the meta function would return the value 8. If the data type is of type `Dna` it returns 2 and for `Dna5` it returns 3. The following list shows the call and the result:

- `int bpv = BitsPerValue<char>::VALUE` ⇒ 8,

- `int bpv = BitsPerValue<Dna>::VALUE` $\Rightarrow 2$,
- `int bpv = BitsPerValue<Dna5>::VALUE` $\Rightarrow 3$.

```
1   template <typename T>
2   struct BitsPerValue
3   {
4           enum {VALUE = 8};
5   };

7   template <>
8   struct BitsPerValue<Dna>
9   {
10          enum {VALUE = 2};
11  };

13  template <>
14  struct BitsPerValue<Dna5>
15  {
16          enum {VALUE = 3};
17  };
```

Listing B.4: C++ meta programming used to determine the bits per value of a certain data type at compile time.

B.1 SeqAn Data Structures

In the following the basic data structures and the thesis related specialisations are discussed.

B.1.1 StringSet & String

As described in IV the primary data structure is the Journal-Set. It is implemented as a `StringSet` specialisation. Except for the special interfaces optimised managing Journal-Strings, the basic interfaces allow for the handling of any kind of values within the set. A `StringSet` in SeqAn is a generic container for any value but preferably used for `String` data types. The listing B.5 shows the declaration of the Journal-Set within SeqAn. The Journal-Set owns its values which means that it manages a copy of the added value. Note that the structure `Owner<JournalSetBasic>` is referred to as template subclassing. This is an approach used to specify specialisations of specialised methods or objects which again supports the high flexibility of the SeqAn library.

```
1  template <typename TValue>
2  class StringSet<TValue, Owner<JournalSetBasic> >
3  {
4          ...
5  };
```

Listing B.5: Declaration of Journal-Set within SeqAn.

The other primary data structure was the Journal-String which is a specialisation of the SeqAn `String` class. The `String` class is basically nothing but a generic container that stores values as well as the `StringSet` does. However, the `String` is preferably used for more atomic data types, such as an alphabet type like `Dna` or `Dna5`.

```
1  template <typename TValue, typename THostSpec = Alloc<>,
2  typename TJournalSpec = Sorted Array, typename TBuffSpec = Alloc<> >
3  class String<TValue, Journaled<THostSpec, TJournalSpec, TBuffSpec> >
4  {
5          ...
6  };

8  String<Dna5, Journaled<Alloc<void>, SortedArray, Alloc<void> > >();
9  String<Dna5, Journaled<Alloc<void>, UnbalancedTree, Alloc<void> > >();
10 String<Dna5, Journaled<Alloc<void>, SkipList, Alloc<void> > >();
```

Listing B.6: Declaration of Journal-Set within SeqAn.

The listing B.6 demonstrates the declaration of the Journal-String and in lines 6, 7 and 8 the anonymous initialisation of each of the in IV described specialisations of the Journal-String. Note that within the declaration of the different template parameters the last three parameters are assigned with default values. Thus it is not specifically required to initialise each template parameter.

B.1.2 Holder

Another discussed structure is the `Holder` class in SeqAn. This class is used within the Journal-Group or the Journal-String to store the reference string. A `Holder` is a powerful data type that manages a relationship to another object. This relation can be different in the sense of that the `Holder` can be empty, can be the owner of the object, such that it manages a copy of the object or it can be a dependent relationship, such that it only stores a reference to the object (see figure B.1).

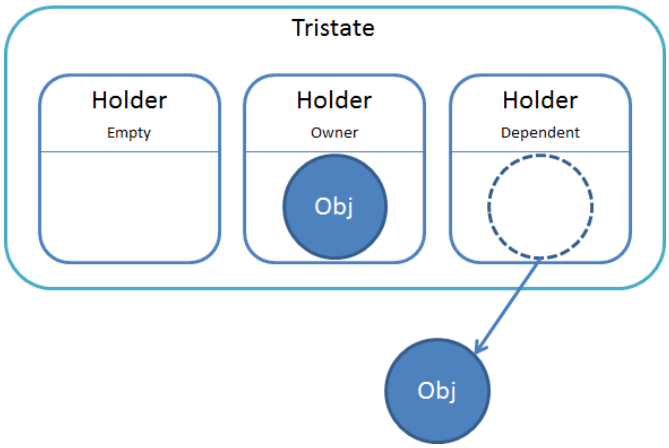


Figure B.1: The Holder interface of SeqAn. Shown are the three different states of the Holder which can be empty, owner or dependent. The default specialisation is tristate which manages all three types of the Holder.

Data

C

Table C.1: Results of different data compression tools for biological sequences obtained from Chen *et al.* [2002]

Sequence	Size	Biocompress-2	GenCompress	CTW+LZ	DNACompress
CHMPXX	121024	1.6848	1.673	1.6690	1.6716
CHNTXX	155844	1.6172	1.6146	1.6129	1.6127
HEHCMVCG	229354	1.848	1.847	1.8414	1.8492
HUMDYSTROP	38770	1.9262	1.9226	1.9175	1.9116
HUMGHCSA	66495	1.307	1.1048	1.0972	1.0272
HUMHBB	73323	1.88	1.8204	1.8082	1.7897
HUMDABCD	58864	1.877	1.8192	1.8218	1.7951
HUMHPRTB	56737	1.9066	1.8466	1.8433	1.8165
MPOMTCG	186608	1.9378	1.9058	1.9000	1.8920
PANMTPACGA	100314	1.8752	1.8624	1.8555	1.8556
VACCG	191737	1.7614	1.7614	1.7616	1.7580
average	⋯	1.7837	1.7434	1.7389	1.7254

C.1 Diff-Format

In the case of the `Simple` specialisation the differences are only displayed through their position in the reference sequence and if it is an insertion the inserted infix is printed in *FastA-Format*. In case of a deletion the position within the reference followed by the length of the operation is displayed. Additionally both operations are marked with either a '+' or a '-' such that the differences are clearly laid out (see figure C.1). The specialisation of `Alignment` is not completely finished yet, and it is marked with a dashed outline. However, it prints the differences as an alignment to the reference string for this particular range.

The begin of a new sequence within a certain group is indicated by the control character '@' followed by the label of the string. If such a label is not given the default name is 'sequence #' and the corresponding encounter of the current sequence. The begin of a new group is

```
> ref=refSeq1
url=file://F:/Development/example_data/testData/host1.txt
format=txt checksum=0
@ sequence 1
+ 5
CCCCCTTTTTTTTCCCC
+ 22
GGGGGGGGG
- 24 10
+ 43
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAACCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCGGGGGGGGGGGGGGGGGGGGGGGGGG
GGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGGTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
- 65 12
> ref=refSeq2
url=file://F:/Development/example_data/testData/host2.txt
format=txt checksum=0
@ sequence 1
@ sequence 2
- 2 12
+ 43
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAACCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

Figure C.1: Diff-Format of the I/O-Module. Stored as the simple specialisation.

marked with the control character '>'. It is followed by the meta informations stored in the Meta-Group object of each group. It consists of

- the label of the reference string prefixed with `ref=`<refernce_name>,
- the url of the reference string prefixed with `url=`<url>,
- the format of the reference string prefixed with `format=`<format> and
- the checksum of the reference string prefixed with `checksum=`<checksum>.

All these informations are optional. They provide information that are useful for the import of a Journal-Set to guarantee correct information. The label supports a better understanding of what this sequence is. The url can be used to specify the location of the reference string. If it is stored on the physical memory, this information can be used to read the reference string during the import if not specified otherwise. The `format` distinguishes which read algorithm is used by SeqAn to read the reference. The following formats are supported:

- FASTA,
- FASTQ,
- GENBANK,
- EMBL,
- RAW.

The big advantage of this data structure is that different sets can be combined by just appending it to one file in the given format. Data structure alignment and Endianness are not concerned by the Diff-Format writer, since the information is serialised as plain text. Thus the representation of numbers and characters depends on the used character encoding of the current system. There might be problems if the information is written using unicode, e.g. unix systems, and read with ANSI encoding, e.g. on windows systems for example ANSI-1252.

C.2 Binary-Format

Figure C.2 shows the particular structure of the binary file. The first column represents the name of the entity followed by a description and the data format. Eventually a value is given for the entity. The first part stores the meta informations of the Journal-Set and the writer options used to serialize the container. It consists of an magic string that identifies it as a file for a Journal-Set. It further encodes a version consisting of a major and minor version number, the number of bits used for a value of the alphabet, for example the both alphabets DNA and DNA 5 in seqan encode the values in two and three bits respectively (see table IV.7), and the number of stored strings and groups.

The next section of the file consists of a list of the groups organized within the set. For each group the meta informations are stored first and then the list of assigned Journal-Strings is

Field	Description	Data Type	Value
magic	BJDIFF magic string	char[4]	BJD\1
version	Major version number	uint8_t	1
version	Minor version number	uint8_t	0
v_bits	Bits used to encode one value	uint8_t	2,3,8
p_format	Packing strategy used	uint8_t	0,1,2
l_set	Number of stored strings	uint32_t	
l_groups	Number of groups within the set	uint32_t	
List of distinct groups			
l_group	Number of strings within the group	uint32_t	
l_label	Length of the label for the reference string	uint32_t	
label	Label of the reference string	char[l_label]	
l_url	Length of the url	uint32_t	
url	The url of the reference string	char[l_url]	
format	Used format to store the reference string	uint8_t	0,1,2,3,4,5
l_ref	Length of the reference string	uint32_t	
checksum	Checksum of the reference string	uint32_t	
List of Journal-Strings within one group			
l_oper	Number of operations need to be stored	uint32_t	
List of operations for each Journal-String			
begin	Begin position of operation within the reference string; For deletions the bitwise-complement of the position is used to distinguish between insertion and deletion	int32_t	
length	Length of the position within the reference string	uint32_t	
l_c_in	Length of the insertion string; all inserted infixes are consecutively appended to one string and compressed using one of the bit packing strategies	uint32_t	
c_insert	The compressed insertion string of all inserted infixes	char[l_c_ins]	

Figure C.2: Binary-Format of the I/O-Module. The modular structure of the binary format.

serialised afterwards. Note that labels, urls and other meta information are stored in plain text. Since there are no delimiters stored the size of the string is stored first, such that the navigation through the file can be ensured even if no delimiters are used. A Journal-String itself is stored by its operations to the reference string. Before the Journal-String is written to a file it is adapted to a Journal-Descriptor as described in the previous section. The Journal-String represents two types of areas. Those who match the reference and those who differ by an insertion to the reference. The information of a deletion is simply encoded as a void between the physical position of two adjacent entries. This is transformed to a representation of deletions before writing the file. The number of Journal-Entries can be reduced a little using this approach. To store an operation only its physical begin position and length is required. In order to distinguish between an insertion and a deletion, the physical position of the deletion is bitwise complemented first. After all operations of one string are stored, the packed insertion buffer is appended.

Bibliography

D

Observatory Nano Briefings, 5. Institute of Nanotechnology, 2010.

The 1000 Genomes Project Consortium. *A map of human genome variation from population-scale sequencing.* Nature, October 2010. **volume 467 (7319)**; pp. 1061–1073.

The 1000 Genomes Project Consortium. *Mapping copy number variation by population-scale genome sequencing.* Nature, Feb 2011. **volume 470**; pp. 59–65.

M. D. Adams, S. E. Celniker, R. A. Holt *et al. The Genome Sequence of Drosophila melanogaster.* Science, 2000. **volume 287 (5461)**; pp. 2185–2195.

S. F. Altschul, W. Gish, W. Miller *et al. Basic local alignment search tool.* Journal of molecular biology, Oct 1990. **volume 215 (3)**; pp. 403–410.

A. Barski, S. Cuddapah, K. Cui *et al. High-resolution profiling of histone methylations in the human genome.* Cell, 2007. **volume 129**; p. 823–837.

D. R. Bentley, S. Balasubramanian, H. P. Swerdlow *et al. Accurate whole human genome sequencing using reversible terminator chemistry.* Nature, 2008. **volume 456**; pp. 53–59.

P. Bieganski, J. Riedl, J. V. Cartis *et al. Generalized suffix trees for biological sequence data: applications and implementation.* In *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences.* 1994, pp. 35–44.

F. R. Blattner, G. Plunkett, C. A. Bloch *et al. The Complete Genome Sequence of Escherichia coli K-12.* Science, 1997. **volume 277 (5331)**; pp. 1453–1462.

M. Brudno, C. Do, G. Cooper *et al. LAGAN and Multi-LAGAN: efficient tools for large-scale multiple alignment of genomic DNA.* Genome Research, Apr 2003a. **volume 13 (4)**; pp. 721–731.

M. Brudno, S. Malde, A. Poliakov *et al. Glocal alignment: finding rearrangements during alignment.* Bioinformatics, 2003b. **volume 19**; pp. 54i–62i. Special Issue on the Proceedings of the ISMB.

R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective.* Addison Wesley Pub Co Inc, 2010, 2nd edition.

W. Chen, V. Kalscheuer, Tzschach A. *et al. Mapping translocation breakpoints by next-generation sequencing.* Genome Research, 2008. **volume 18**; pp. 1143–1149.

X. Chen, S. Kwong and M. Li. *A compression algorithm for DNA sequences.* In *IEEE Engineering in Medicine and Biology Magazine*, chapter 20. 2001, pp. 61–66.

X. Chen, M. Li, B. Ma *et al. DNACompress: fast and effective DNA sequence compression.* Bioinformatics, 2002. **volume 18 (12)**; pp. 1696–1698.

S. Christley, Y. Lu, C. Li *et al. Human genomes as email attachments.* Bioinformatics (Oxford, England), January 2009. **volume 25 (2)**; pp. 274–5.

Human Genome Reference Consortium. *Human Reference Sequence Assembly hg19.* http://genome.ucsc.edu/cgi-bin/hgTracks?hgsid=194675807&chromInfoPage=, Feb 2009.

R. Curnow and T. Kirkwood. *Statistical analysis of deoxyribonucleic acid sequence data – a review.* J. Royal Statistical Society, 1989. **volume 152**; pp. 199–220.

A. Döring, D. Weese, T. Rausch *et al. SeqAn an efficient, generic C++ library for sequence analysis.* BMC bioinformatics, January 2008. **volume 9**; p. 11.

J. Gailly and M. Adler. *gzip.* http://www.gzip.org/

E. Gamma, R. Helm, R. Johnnson *et al. Design Patterns.* Addison-Wesley Professional Computing Series, 1995.

E. J. Gardner, M. J. Sinnoms and D. P. Snustad. *Principles of Genetics.* John Wiley & Sons, 1991, 8th edition.

A. Goffeau, B. G. Barrell, H. Bussey *et al. Life with 6000 Genes.* Science, 1996. **volume 274 (5287)**; pp. 546–567.

O. Gotoh. *An improved algorithm for matching biological sequences.* Journal of Molecular Biology, 1982. **volume 162 (3)**; pp. 705 – 708.

S. G. Gregory, M. Sekhon, J. Schein *et al. A physical map of the mouse genome.* Nature, August 2002. **volume 418**; pp. 743–750.

S. Grumbach and F. Tahi. *Compression of DNA sequences.* In *Data Compression Conference.* IEEE Computer Society Press, pp. 340–350.

S. Grumbach and F. Tahi. *A new challenge for compression algorithms: genetic sequences.* J. Information Processing and Management, 1994. **volume 30 (6)**; pp. 875–866.

L.W. Hillier, G.T. Marth, A.R. Quinlan *et al. Whole-genome sequencing and variant discovery in C. elegans.* Nature Methods, 2008. **volume 5**; pp. 183–188.

The International Human Genome Consortium. *Initial sequencing and analysis of the human genome.* Nature, February 2001. **volume 409 (6822)**; pp. 860–921.

D. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, 1998, 2nd edition.

B. Langmead, C. Trapnell, M. Pop *et al. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome.* Genome Biology, May 2009. **volume 10 (3)**; p. R25.

C. Lee, C. Grasso and M. F. Sharlow. *Multiple sequence alignment using partial order graphs.* Bioinformatics, 2002. **volume 18 (3)**; pp. 452–464.

В. И. Левенштейн (Levenshtein). *Двоичные коды с исправлением выпадений, вставок и замещений символов.* In *Доклады Академий Наук СССР*, volume 163. 1965, pp. 845–848.

Mardis E. Ley, T. and, L. Ding, B. Fulton *et al. DNA sequencing of a cytogenetically normal acute myeloid leukaemia genome.* Nature, 2008. **volume 456**; p. 66–72.

D. Lipman, S. Altschul and J. Kececioglu. *A tool for multiple sequence alignment.* National Academy of Sciences of the USA, 1989. **volume 86 (12)**; p. 4412–4415.

R. F. Massung, J. J. Esposito, L.-I. Liu *et al. Potential virulence determinants in terminal regions of variola smallpox virus genome.* Nature, 1993. **volume 366**; pp. 748 – 751.

A. M. Mathai and P. N. Rathie. *Basic Concepts in Information Theory and Statistics.* Wiley Eastern Ltd., 1975.

T. Matsumoto, K. Sadakane and H. Imai. *Biological sequence compression algorithms.* Universal Academy Press, 2000.

M. L. Metzker. *Sequencing technologies — the next generation.* Nature Reviews Genetics, December 2009. **volume 11 (1)**; pp. 31–46.

R.D. Morin, M.D. O'Connor, M. Griffith *et al. Application of massively parallel sequencing to microrna profiling and discovery in human embryonic stem cells.* Genome Research, 2008. **volume 18**; p. 610–621.

R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

S. B. Needleman and C. D. Wunsch. *A general method applicable to the search for similarities in the amino acid sequence of two proteins.* Journal of Molecular Biology, 1970. **volume 48 (3)**; pp. 443 – 453.

T. Papadakis, I. Munro and P. Poblette. *Exact Analysis of Expected Search Cost in Skip Lists.* Technical report, Dept. of Computer Science, Univ. of Waterloo, 1990.

W. Pugh. *Skip lists: a probabilistic alternative to balanced trees.* Communications of the ACM, June 1990. **volume 33 (6)**; pp. 668–676.

P.-T. Pyl. *Incremental Index Structures.* Master's thesis, Freie Universität Berlin, Algorithmic Bioinformatics, 2010.

B. Raphael, D. Zhi, H. Tang *et al. A novel method for multiple alignment of sequences with repeated and shuffled elements.* Genome Resarch, 2004. **volume 14 (11)**; pp. 2336–2346.

T. Rausch, A.-K. Emde, D. Weese *et al. Segment-based multiple sequence alignment.* Bioinformatics, 2008. **volume 24 (16)**; pp. 187–192.

E. Rivals, O. Delgrange, J.-P. Delahaye *et al. Detection of significant patterns by compression algorithms: the case of approximate tandem repeats in DNA sequences.* CABIOS, 1997. **volume 13 (2)**; p. 131–136.

S. M. Rumble, P. Lacroute, A. V. Dalca *et al. SHRiMP: Accurate Mapping of Short Color-space Reads.* PLoS Comput Biol, 05 2009. **volume 5 (5)**; p. e1000 386.

D. Salomon. *Data Compression.* Springer Verlag, Department of Computer Science, California State University, Northridge, Nortridge, CA 91330–8281, 2000a, 2nd edition.

D. Salomon. *Data Compression The Complete Reference.* Springer Verlag, 2000b, 2nd edition.

F. Sanger and A. R. Coulson. *A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase.* Journal of Molecular Biology, 1975. **volume 94 (3)**; pp. 441 – 446.

F. Sanger, A. R. Coulson, T. Friedmann *et al. The nucleotide sequence of bacteriophage [phi]X174.* Journal of Molecular Biology, 1978. **volume 125 (2)**; pp. 225 – 246. ISSN 0022-2836. doi: DOI:10.1016/0022-2836(78)90346-7.

F. Sanger, A. R. Coulson, G. F. Hong *et al. Nucleotide sequence of bacteriophage [lambda] DNA.* Journal of Molecular Biology, 1982. **volume 162 (4)**; pp. 729 – 773.

F. Sanger, S. Nicklen and A. Coulsen. *DNA sequencing with chain-terminating inhibitors.* Proceedings of the National Academy of Sciences of the United States of America, 1977. **volume 74 (12)**; pp. 5463–5467.

K. Sayood, editor. *Lossless Compression Handbook.* Elsevier Science, 2003.

C. E. Shannon. *A Mathematical Theory of Communication.* Bell System Technical Journal, July 1948. **volume 27**; pp. 379 – 423.

F. Shei. *Lecture Notes in Computer Science*, volume 1179, chapter 2. Springer, 1996, pp. 11–22.

J. A. Storer, editor. *Data Compression Methods and Theory.* Computer Science Press, 1988.

B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1997.

L. Wang and T. Jiang. *On the complexity of multiple sequence alignment.* Computational Biology, 1994. **volume 1 (4)**; p. 337–348.

D. Weese, A.-K. Emde, T. Rausch *et al.* *RazerS - Fast Read Mapping with Sensitivity Control.* Genome Research, Sep 2009. **volume 19**; pp. 1646–1654.

Statement of authorship

E

I declare that this document and the accompanying code has been composed by myself, and describes my own work, unless otherwise acknowledged in the text. It has not been accepted in any previous application for a degree. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Berlin, 14.06.2011

René Märker