Project number: 317871
Project acronym: BIOBANKCLOUD

**Project title:** Scalable, Secure Storage of Biobank Data  
**Project website:** http://www.biobankcloud.eu  
**Project coordinator:** Jim Dowling (KTH)  
**Coordinator e-mail:** jdowling@kth.se

**WORK PACKAGE 2:**  
Storage Architecture

**WP leader:** Jim Dowling  
**WP leader organization:** KTH  
**WP leader e-mail:** jdowling@kth.se

**PROJECT DELIVERABLE**

**D2.2**  
Reference-based Compression Algorithms

**Due date:** 15th June, 2014 (M20)  
**Deliverable version:** 1.0
Editor
Alysson Bessani (FFCUL)

Contributors
Sebastian Wandelt, Marc Bux, Johannes Starlinger and Ulf Leser (UBER)
Fernando Alves, Vinicius Cogo and Alysson Bessani (FFCUL)

Disclaimer
This work was partially supported by the European Commission through the FP7-ICT program under project BiobankCloud, number 317871.

The information in this document is provided as is, and no warranty is given or implied that the information is fit for any particular purpose.

The user thereof uses the information at its sole risk and liability. The opinions expressed in this deliverable are those of the authors. They do not necessarily represent the views of all BiobankCloud partners.
# Contents

1 Introduction .................................................. 1

2 Trends in Genome Compression ................................. 3
   2.1 Introduction .................................................. 3
   2.2 Basic Techniques ............................................. 5
      2.2.1 Naive Bit Manipulation Algorithms ...................... 5
      2.2.2 Dictionary-based Algorithms ............................. 7
      2.2.3 Statistical Algorithms ................................ 10
      2.2.4 Referential Algorithms ................................ 11
   2.3 Whole Genome Compression .................................. 14
      2.3.1 Naive Bit Manipulation Algorithms ...................... 14
      2.3.2 Dictionary-based Algorithms ............................. 14
      2.3.3 Statistical Algorithms ................................ 14
      2.3.4 Referential Algorithms ................................ 15
   2.4 Read compression ............................................. 17
      2.4.1 Dictionary-based Algorithms ............................. 20
      2.4.2 Statistical Algorithms ................................ 20
      2.4.3 Referential Algorithms ................................ 21
   2.5 Discussion and Conclusions ................................ 21

3 FRESCO: Referential Compression of Highly-Similar sequences 23
   3.1 Introduction .................................................. 23
   3.2 Related Work .................................................. 25
   3.3 Referential Compression ...................................... 26
      3.3.1 Representation of Matches into a Reference .............. 26
      3.3.2 Compression Algorithm .................................. 28
   3.4 Improving Compression Ratios ................................ 29
      3.4.1 Selecting a good Reference ............................... 30
      3.4.2 Reference Rewriting ..................................... 31
   3.5 Second-Order Compression .................................... 33
   3.6 Evaluation ...................................................... 35
      3.6.1 Existing Standard Compression Algorithms ............... 36
      3.6.2 Referential Compression Algorithms ...................... 36
      3.6.3 Reference Selection in FRESCO ........................... 38
      3.6.4 Reference Rewriting in FRESCO .......................... 40
      3.6.5 Second-Order Compression in FRESCO ..................... 42
      3.6.6 FRESCO for Compressing Wikipedia Articles ................ 43
   3.7 FRESCO: Open Source Release ................................. 43

BiobankCloud - D2.2
## 4 On-Demand Indexing for Referential Compression

### 4.1 Introduction

### 4.2 Referential Compression

#### 4.2.1 Genome Files

#### 4.2.2 The FRESCO Library

### 4.3 On-Demand Reference Indexing

#### 4.3.1 Compression Algorithm

#### 4.3.2 KmerTable

#### 4.3.3 Decompression Algorithm

#### 4.3.4 File formats

#### 4.3.5 Execution

#### 4.3.6 Java Virtual Machine Configuration

### 4.4 Evaluation

#### 4.4.1 Experimental environment

#### 4.4.2 Compression ratio

#### 4.4.3 Time

#### 4.4.4 Memory footprint

#### 4.4.5 Compressing entire genomes

#### 4.4.6 Discussion

### 4.5 Conclusions and future work

## 5 RCSI: Scalable Similarity Search in Thousand(s) of Genomes

### 5.1 Introduction

### 5.2 Related Work

### 5.3 Foundations

#### 5.3.1 k-approximate Search

#### 5.3.2 Indexing

#### 5.3.3 Referential Compression

#### 5.3.4 Referential Sequence Database Search

### 5.4 Searching Referentially Compressed Sequences

#### 5.4.1 Searching in Referential Match Entries

#### 5.4.2 Searching across Sequence Deviations

#### 5.4.3 Searching Referential Sequence Databases

### 5.5 Best Reference Selection

### 5.6 Evaluation

#### 5.6.1 Data and Queries

#### 5.6.2 Compression

#### 5.6.3 Searching 1,092 Chromosomes

#### 5.6.4 Searching 1,092 Genomes

#### 5.6.5 Competitive Evaluation

#### 5.6.6 Best Reference Selection

### 5.7 Conclusions

## 6 Concluding Remarks
Chapter 1

Introduction

Moving big data files is one of the main problems when working with large volumes of sequenced data [104, 114]. A naive usage of protocols like the FTP would result in several days of transferring time prior to any data analysis since next generation sequencing (NGS) machines easily generates terabytes of data. Thus, using compression is an interesting circumvention to this problem. Although DNA usually does not compress well in itself, the specific setting of NGS projects allows the usage of so-called referential compression. In this approach, first a reference genome is fixed (and uploaded once). All further sequencing data of the same species is mapped to the reference and only the differences (the compressed file) are sent or stored. As differences between individuals of the same species typically are small, the new sequences frequently map very well to the reference, with differences occurring at less than 1% of the bases in general. Thus, the new sequences can be compressed by only submitting location in and differences to the reference.

In this deliverable, we present three contributions from the BiobankCloud consortium regarding the implementation of efficient reference-based compression (Task 2.3) for using such algorithms in the BiobankCloud platform. More specifically, we present a new compression tool – FRESCO – that implements several novel techniques for efficient referential-compression. Furthermore, we designed novel techniques for (1) searching on compressed files, without requiring fully decompression, and (2) removing the reference-indexing step, which accounts for most of the time when compressing single (or few files). These techniques were also made available through tools to be used in the platform.

Deviations from the original plan. The work presented in this deliverable deviates from the original plan (in the DoW) in two aspects. First, we planned to exploit parallel and multi-core machines for making reference-based compression algorithms faster. Second, we wanted to develop efficient algorithms to update compressed files with respect to new versions of the reference genome.

Regarding the first aspect, although we designed some prototypes of parallel algorithms for compression, we found that these algorithms were mostly IO-bound. Consequently, parallelism will only help if we had also several disks (and the aligned files distributed among them). Furthermore, we found there were other areas in which we could innovate, e.g., by designing algorithms that can work directly with the compressed files (removing the need for decompressing them) and removing most of the reference-indexing time (which accounts for more 90% of the latency of the tools when compressing a single file).

Concerning the second aspect, we have found that there is not much room for significant innovation on updating reference files. Furthermore, it will be infeasible to assume all files
managed by the platform would use a single version of the reference file, and thus we will have to keep several references and annotate the compressed files with the unique id of the reference file used in their compression. Notice however that several of the contributions described in this deliverable improved the compression/decompression time, and thus improves the time required for updating a compressed file (by decompressing it using the old version of the reference and compressing it again using the new one).

Organization of the deliverable. Besides this introduction, this deliverable contains four chapters describing different contributions of the project. Chapter 2 reviews related work and surveys the main trends in genome compression. Chapter 3 presents the FRESCO open source tool and evaluates its performance. The on-demand indexing of reference genomes and its implementation in a Java-based compression tool is presented in Chapter 4. Chapter 5 discusses a method and tool for similarity search in compressed genomes and their evaluation. Finally, we conclude the report with some remarks about the use of the contributions here presented in the project (Chapter 6). Since most of these chapters come from independent contributions and publications, we opted to make them as self-contained as possible, without big dependencies one from another.
Chapter 2

Trends in Genome Compression

Chapter Authors:
Sebastian Wandelt, Marc Bux, and Ulf Leser (UBER). ¹

Technological advancements in high-throughput sequencing have lead to a tremendous increase in the amount of genomic data produced. With the cost being down to 2,000 USD for a single human genome, sequencing dozens of individuals is a task that already today is feasible even for smaller project or organizations. However, generating the sequence is only one issue; another one is storing, managing, and analysing it. These tasks become more and more challenging due to the sheer size of the data sets and are increasingly considered as the most severe bottlenecks in larger genome projects. One possible countermeasure is to compress the data; compression saves costs in terms of requiring less hard disk and in terms of requiring less bandwidth if data is shipped to large compute clusters for parallel analysis. Accordingly, sequence compression has recently created much interest in the scientific community. In this chapter, we explain the different basic techniques for sequence compression, point to differences between different compression tasks (e.g. genome versus read compression), and present a comparison of current approaches and tools. To further stimulate progress in genome compression research, we also identify key challenges to future systems.

2.1 Introduction

The introduction of high-throughput sequencing has led to large amounts of biological data. Sequence databases now contain literally more data than any scientist can handle. In the future, the situation will become even worse [55], for several reasons: First, the decreasing cost per genome makes larger and larger projects feasible, taking advantage of the increased statistical power of larger data sets. Second, the falling cost alongside the increasing knowledge on the relationship between genotype and phenotype will also make more and more individuals interested in their genome and its genetic predispositions. Third, sequencing platforms will produce more and longer reads at a growing rate, thus increasing the possible throughput.

The resulting growing amounts of sequenced genomes make it more necessary than ever to carefully think about efficient storage and transmission. Current projects already target sequencing of several thousands of human genomes[51, 118]. A straightforward way to store and manage DNA data in computers is to use ASCII encoded characters, resulting in one byte for each base. Although this representation is wasteful in terms of space because it does not

¹Content of this chapter was previously published in Wandelt et al. 2013 [120]
compress the sequence at all, many scientists are still accustomed to this format. Its main advantages are that it allows applications and programming languages easy access to the data and that it is easy to read by human beings. However, it also has the drawback of being highly space-intensive. Suppose a project aims at sequencing 10,000 humans. Stripped of all quality information, assembled to individual genomes and managed as ASCII files, this would amount to roughly 30TB of data. While storing such a mass of data is no severe problem in today’s hardware, transmitting it to a (possibly remote) compute cluster for analysis would be a highly time-consuming task [114]. The situation becomes much worse when not only finished genomes are stored, but the raw read sets with associated quality scores. Then, the amount of necessary space easily grows by a factor of 20 or more, and managing 600TB already is much more difficult than managing 30TB, especially when fail-safeness, backup, and long-term archival are considered. Finally, while storing 10,000 genomes seems like a lot today, much higher numbers will probably become reality within a couple of years when the current visions for third-generation sequencing become reality [103].

Another aspect of data size is monetary cost. As pointed out by Stein [110], the cost for sequencing is decreasing much faster than the cost for storing sequences. As a consequence, storing sequences (i.e., buying hard disk) will become more costly than producing them in the very near future. One escape from this situation is to simply delete sequences after analysis, thus saving mid- and long-term storage costs, and to re-sequence the original samples if the data is needed again later. However, such a radical approach still is considered inappropriate by most researchers for various reasons, including general rules of good scientific conduct which require storing experimental data for a prolonged period of time to allow re-assessment and reproduction of results.

Another solution, which we discuss in this chapter, is compression. By compression we mean methods to store the information as in the original data but with less space, possibly losing information (see below). There are various methods how this can be achieved. Early approaches focused mainly on bit manipulation techniques, i.e., they packed more bases into one byte of data. These were succeeded by statistical and dictionary-based approaches, which reach considerably higher compression ratios than bit-manipulation techniques. More recently, so-called referential compression algorithms have become popular which allow compression rates that are orders of magnitude higher than that of previous attempts. The field is highly active; only in the last few years dozens of papers appeared that propose new compression schemes or variations of existing methods. At the same time, the field is difficult to understand as schemes may differ substantially, results were published in different communities, and methods often are specialized in only a particular problem (such as compression of bacterial genomes or compressing only coding sequences). Another important difference that sometimes is not obviously reflected in published results is that between compressing genomes and compressing reads.

In this chapter, we survey recent algorithms for all these classes of problems. In Section 2.2 we first explain the four main classes of compression techniques, i.e., bit manipulation, dictionary-based, statistical, and referential compression. We then discuss, class-by-class concrete systems for compressing entire genomes (Section 2.3). In contrast, in Section 2.4, we review recent contributions to the compression of sequence reads, including treatment of quality scores. The chapter is concluded in Section 2.5 with potential directions for future research in the area of genome compression.
2.2 Basic Techniques

The increasing number of (re-)sequenced genomes has lead to many proposals for compression algorithms. In general, compression algorithms can be separated into naive bit encoding, dictionary-based, statistical, or referential approaches.

- **Naive bit encoding** algorithms exploit encodings of two or more symbols into one byte, using fixed-length encodings[17, 44].

- **Dictionary-based** or substitutional algorithms replace repeated substrings by references to a dictionary (a set of previously seen or predefined common strings), which is built at runtime or offline[73, 107].

- **Statistical** or entropy encoding algorithms derive a probabilistic model from the input. Based on partial matches of subsets of the input, this model predicts the next symbols in the sequence. High compression rates are possible if the model always indicates high probabilities for the next symbol, i.e. if the prediction is reliable[16, 21].

- **Referential** or reference-based approaches recently emerge as a fourth type of sequence compression algorithm. Similar to dictionary-based techniques, these algorithms replace long substrings of the to-be-compressed input with references to another string. However, these references point to external sequences, which are not part of the to-be-compressed input data. Furthermore, the reference is usually static, while dictionaries are being extended during compression phase.

In order to compare algorithms in the remaining part of our work, we need to introduce some terminology. Usually, the input of a compression algorithm is a sequence of symbols from a given alphabet. **Lossless compression** allows to reconstruct the complete original input from the compressed output, as opposed to **lossy compression**. A compression scheme allows **random access**, if arbitrary positions of the input stream can be accessed without decompressing the whole stream. Random access can be for instance enabled by splitting the input sequence into blocks.

There exist additional compression techniques, which can be employed in addition to the aforementioned categories. For instance, in **run-length encoding** consecutive occurrences of the same symbol are replaced by a counter. This technique is especially useful to encode a long succeeding occurrences of the same symbol, e.g. \(N\), in sequences.

In the remaining part of the chapter, we only discuss lossless compression algorithms, as in most biomedical applications every single base is important. Lossy compression has more applications like, for instance, image compression (e.g. the JPEG standard, [95]). Further, we do not discuss the compression of protein sequences which usually is considered as more complex due to the larger alphabet size and the fact that repeats are less frequent [90]. For some work on compressing protein sequences, see [47].

In the following subsections, we will discuss all these base techniques in detail. This discussion builds the foundation for evaluating recent systems in Section 2.3 and Section 2.4. Table 2.1 summarizes the compression rates for state of the art compression schemes.

2.2.1 Naive Bit Manipulation Algorithms

Using eight bits (or 256 different states) to encode four different bases obviously constitutes a waste of space. Four bases can easily be encoded with two bits (or four states). Therefore, a
Table 2.1: A comparison of standard compression schemes.

<table>
<thead>
<tr>
<th>Name</th>
<th>Usual compression rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive Bit Manipulation Algorithms</td>
<td>2:1 – 6:1</td>
</tr>
<tr>
<td>Dictionary-based Algorithms</td>
<td>4:1 – 6:1</td>
</tr>
<tr>
<td>Statistical Algorithms</td>
<td>4:1 – 8:1</td>
</tr>
<tr>
<td>Referential Algorithms</td>
<td>1:1 – 400:1</td>
</tr>
</tbody>
</table>

Figure 2.1: Example for Naive Bit Encoding
straight-forward compression technique for DNA sequence data is the encoding of four bases within one byte via bit encoding. One example for naive bit encoding is shown in Figure 2.1. Each symbol in the input is replaced by two bits using the replacement \{A → 00, C → 01, G → 10, T → 11\}.

Current processor architectures provide highly improved bit operations, which basically allow an encoding of DNA sequence data with two bits on the fly. Note that this encoding impacts human readability of data severely, since one needs a lookup table in order to interpret the compressed data. Since the representation of four bases fits exactly into eight bits, byte boundaries or big/little endian issues are circumvented.

If one wishes to encode additional symbols, such as \(N\) for indistinguishable base, the encoding becomes more complex. One approach to encode the five symbols \(A, C, G, T, N\) is to put three consecutive bases into one byte. Seven bits can encode 128 states and since \(5^3 < 128\) one can put three such symbols into one byte. The remaining eighth bit is usually left unused or used as an opcode or flag bit, which could for instance indicate the end of a stream. For reasons of byte boundary limitations mentioned above, it might not be advantageous to use the eighth bit for actual symbol encoding. More than five symbols can be handled in a similar way. However, with increasing alphabet size less symbols fit into one byte.

The compression rate of naive bit manipulation algorithms is 4:1, if the size of the input alphabet is four, or less than 4:1 for more than four symbols. The compression rate can be further improved, if additional compression is applied on top, e.g. run-length encoding.

### 2.2.2 Dictionary-based Algorithms

Dictionary-based encodings are compression schemes which are generally independent of the specific characteristics of the input data. The overall approach is to replace repeated data elements (here: DNA subsequences) of the input with references to a dictionary. Repetitions are usually detected by bookkeeping previously occurring sequences. In many realizations the dictionary is reconstructed at runtime during the decompression process. This means that the dictionary itself does not have to be stored along with the compressed data. One example for a dictionary-based algorithm is shown in Figure 2.2.

Lempel-Ziv-based compression algorithms, such as LZ77 or LZ78, are prominent example of dictionary-based algorithms [134]. In those algorithms, the input sequence is parsed sequentially and reoccurring substrings are detected with some method. Substrings that have not been
encountered before are registered in the dictionary in the form of a reference to a previously encountered substring plus one new character. Algorithms following this scheme mainly differ in the concrete method applied to detect repeated substrings, which is tightly connected to the average length of encoded repeats. Another difference is the concrete method used to encode repeated occurrences of substrings with dictionary indices. This touches the important research question on how to represent integer values in the most space-efficient way. Usually the range of integers is restricted in a way that gives raise to an efficient encoding. Two examples of integer encoding schemes are Golomb codes [42], which encode small numbers more efficiently than large numbers, and Fibonacci codes [62], which are more tolerant to failures. Using such codes, current methods for dictionary-compression (see Table 2.2) reach compression rates between 4:1 and 6:1 depending on the frequency of repeats in the genomes being compressed.
Table 2.2: Alphabetical list of recent non-referential compression schemes for whole genome sequences. Values are taken from the original papers. “–” means that the value is unknown.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Tested with</th>
<th>Data used</th>
<th>Compression ratio</th>
<th>Compression speed (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[9]</td>
<td>–</td>
<td>genes</td>
<td>up to 5.7:1</td>
<td>–</td>
</tr>
<tr>
<td>[81]</td>
<td>–</td>
<td>–</td>
<td>4:1</td>
<td>4.5</td>
</tr>
<tr>
<td>[84] DNASC</td>
<td>–</td>
<td>genes</td>
<td>4.4:1 – 5.3:1</td>
<td>–</td>
</tr>
<tr>
<td>[99]</td>
<td>human genes, genome</td>
<td>5.3:1</td>
<td>0.02</td>
<td></td>
</tr>
<tr>
<td>[100] GBC</td>
<td>–</td>
<td>genes</td>
<td>3.5:1</td>
<td>–</td>
</tr>
<tr>
<td>[117] 2D</td>
<td>bacteria</td>
<td>genome</td>
<td>3.2:1</td>
<td>0.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tool</th>
<th>Tested with</th>
<th>Data used</th>
<th>Compression ratio</th>
<th>Compression speed (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[67] Comrad</td>
<td>bacteria, human genomes</td>
<td>5.5:1 (and better)</td>
<td>0.3</td>
<td></td>
</tr>
<tr>
<td>[56]</td>
<td>–</td>
<td>genes</td>
<td>5.3:1 – 5.7:1</td>
<td>–</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tool</th>
<th>Tested with</th>
<th>Data used</th>
<th>Compression ratio</th>
<th>Compression speed (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[33]</td>
<td>–</td>
<td>genes</td>
<td>4.73:1</td>
<td>–</td>
</tr>
<tr>
<td>[57]</td>
<td>bacteria, human</td>
<td>genes</td>
<td>4:1 – 5:1</td>
<td>–</td>
</tr>
<tr>
<td>[98]</td>
<td>human genome</td>
<td></td>
<td>4.7:1</td>
<td>1.08</td>
</tr>
<tr>
<td>[112]</td>
<td>human genome</td>
<td></td>
<td>5:1 – 5.7:1</td>
<td>–</td>
</tr>
</tbody>
</table>

---

*a* Partial sourcecode is supplementary of original submission  
*b* Partial sourcecode is supplementary of original submission  
*c* Sourcecode is supplementary of original submission  
2.2.3 Statistical Algorithms

Statistical algorithms create a statistical model of the input data, which is in most cases represented in the form of a probabilistic or prefix tree data structure. Subsequences with a higher frequency in the genome are then represented with shorter codes. Accordingly, they can be considered as a variant of dictionary-based compression which solve the problems of repeat detection and reference encoding in one algorithm. Compression rates depend on the quality of the model as well as the existence of detectable patterns in input data.

One of the most commonly used and best understood statistical encodings is Huffman encoding [52]. It uses a variable-length code table derived from estimated probabilities of occurrence for each possible symbol. A binary tree is created in which leaf nodes correspond to symbols and edges are labelled with probabilities and the derived codes. The resulting Huffman code table has to be stored in addition to the compressed stream and thus has to be taken into account when computing the compression ratio. Note that sharing the same code table over many streams can reduce this storage overhead. Huffman encoding generally benefits from large alphabets with an uneven distribution of used characters [11]. It is therefore considered not to be ideal for efficient compression of DNA sequences [80]. One example for a statistical algorithm is shown in Figure 2.3. Often occurring base A is assigned a short code (0), while less likely base T is assigned a longer code (111).

While Huffman encoding is based on finding shortest codes for single individual symbols, arithmetic encoding [85] encodes longer strings – or even whole input streams – as a single number between zero and one. This allows for higher compression rates in applications with small alphabets. Many actual implementations use range encoding, instead of arithmetic encoding, since it is believed that the former one is less encumbered by patents.

[26] proposed a compression technique based on hidden Markov models. This approach can be applied to DNA sequence compression, assuming that DNA sequence data can be approximated by a hidden Markov model. One can distinguish Markov-approaches by the order of the

Figure 2.3: Example for Statistical Algorithms
model. A second order Markov model, for instance, takes the context of the last two symbols into account when predicting the probability of the next symbols.

The compression rate of statistical algorithms is usually between 4:1 and 8:1 (see Table 2.1). The compression rate depends mainly on the distribution of input symbols and the available memory for construction of frequency distributions.

### 2.2.4 Referential Algorithms

While genome research for a long time was focusing on sequencing new genomes, recent advances in sequencing technology [104] and increasing demands from areas such as translational medicine [19] have made resequencing of genomes - which means sequencing different individuals of the same species - more and more popular. Large international projects, such as the ICGC [51] already plan to sequence thousands of human genomes. As in resequencing all genomes are from the same species, the resulting sequences exhibit extremely high levels of similarity. This fact is exploited by so-called referential compression schemes. The key idea is to encode sequences with respect to a an external (set of) reference sequence(s). Given that this reference sequence is available to the decompressor, such techniques allow for very high compression rates. Long matches in the reference are usually found by using index structures, e.g. hash-based or suffix trees. The general algorithm for referential compression in pseudo-code is shown in Algorithm 1. The value of $X$ decides whether short matches are encoded as a reference or as a raw string. In many implementations, any match longer than two characters is encoded as a referential match.

**Algorithm 1** Sketch of a Referential Compression Algorithm

1: **while** Input contains characters **do**
2: \hspace{1em} Find longest matching substring in reference for current input position
3: \hspace{1em} **if** length of match $> X$ **then**
4: \hspace{2em} Encode match as $(\textit{matchposition}, \textit{length})$
5: \hspace{1em} **else**
6: \hspace{2em} Encode match with raw symbols
7: \hspace{1em} **end if**
8: **end while**
One example for a referential compression algorithm is shown in Figure 2.4. There exist two interval matches. For instance, interval match (7, 4) indicates that the current input matches the reference for four symbols starting at position seven. In addition, a short sequence is stored as raw bases, since there exists no good match in the reference sequence for TA.

Rates of compression are the higher, the more similar the to-be-encoded sequence and the reference are. Having highly similar sequences, long stretches of identical DNA must be present in both sequences, interrupted mostly by SNPs and short INDELs. In this case, matched regions can be easily represented by noting the reference sequence identifier together with an interval describing the match. Referentially compressed DNA sequences comprise lists of such interval matches and reach highest compression rates for in-species compression. However, compressing, for instance, a human genome against a mouse genome leads to considerably worse rates, as most most matches of human genomes with respect to a mouse genome are only of length 20-25 (data not shown). The encoding of these short matches (for instance four bytes for position of the match and two bytes for the length) is longer than the naive encoding in two bits (for instance $\frac{20}{4}$ bytes).

In general, finding a proper reference sequence can be non-trivial. The task is simple if the species (and chromosome etc.) of a sequence is known, but much harder in projects from meta genomics, where sequences are sampled at random from a large set of species [105]. Heuristics for finding a good reference sequence can be based on k-mer hashing. High similarity of k-mers indicate high potential for compression with respect to the reference. However, at genome scale, $k$ should be chosen higher than 15, in order to avoid too many random matches.

Besides finding identical matches, there are also more sophisticated matching techniques between an input and a reference sequence, e.g. referential encoding to a complementary sub-sequence of the reference. One may also allow additional kinds of coding blocks in the compressed file, such as short raw snippets of DNA sequences. The inclusion of these short raw sequences makes perfectly sense if no match in the reference can be found such that encoding the reference is shorter than storing the raw sequence itself.

The main challenge for reference-based encoding is to find long matches efficiently. This can be done by indexing sequences either in the form of suffix trees or by using hash-based approaches. As with dictionary-based compression, a second challenge is to find a space efficient encoding of interval matches and other coding blocks.

A wide range of compression rates has been reported for reference-based encoding. Given a good matching reference sequence, compression rates are 400:1 and better (see Table 2.3). However, when comparing different referential compression schemes, it has to be taken into account, that some authors include the reference sequence in the compression ratio, while other authors do not! We think that in the future, compression ratios for referential compression schemes should be always stated in a uniform way, e.g. without including the reference sequence.
Table 2.3: Alphabetical list of recent referential compression schemes for whole genome sequences. Ranges are given as reported in the original papers. Please note that some authors include the reference sequence for the compression ratio, while others do not.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Data source</th>
<th>Compression ratio</th>
<th>Speed of compression (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[12]a</td>
<td>up to 8 GB for full diploid human sequence</td>
<td>24:1 – 400:1</td>
<td>–</td>
</tr>
<tr>
<td>[20]b</td>
<td>Wheeler Genome with regard to HG18</td>
<td>772:1</td>
<td>–</td>
</tr>
<tr>
<td>[43]c</td>
<td>yeast and human genomes</td>
<td>16:1</td>
<td>13.33</td>
</tr>
<tr>
<td>[60]d</td>
<td>few short gene sequences</td>
<td>40:1 – 80:1</td>
<td>0.0005</td>
</tr>
<tr>
<td>[64]e</td>
<td>various text formats</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>[69]f</td>
<td>RLZ</td>
<td>16:1 – 220:1</td>
<td>–</td>
</tr>
<tr>
<td>[70]g</td>
<td>RLZopt</td>
<td>220:1</td>
<td>0.66</td>
</tr>
<tr>
<td>[92]h</td>
<td>HG18 human chromosomes</td>
<td>139:1 – 895:1</td>
<td>–</td>
</tr>
<tr>
<td>[94]i</td>
<td>some genome data and general strings</td>
<td>15:1 – 80:1</td>
<td>–</td>
</tr>
<tr>
<td>[97]j</td>
<td>GReEn</td>
<td>172:1</td>
<td>8.33</td>
</tr>
<tr>
<td>[125]k</td>
<td>GRS</td>
<td>159:1</td>
<td>1.85</td>
</tr>
</tbody>
</table>

ahttp://mammag.web.uci.edu/bin/view/Mitowiki/ProjectDNACompression; not available on 07/11/2012
bhttp://www.ics.uci.edu/ xhx/project/DNAzip; not available on 07/11/2012
chttp://sun.aei.polsl.pl/gdc
ghttp://gmdd.shgmo.org/Computational-Biology/GRS/
2.3 Whole Genome Compression

In the following section we discuss concrete algorithms for compressing whole genomes. Compression of reads will be covered in Section 2.4. Table 2.2 summarizes compression rates and other properties of the non-referential compression schemes we will discuss, while Table 2.3 contains similar information for referential schemes.

2.3.1 Naive Bit Manipulation Algorithms

[117] propose 2D, a compression algorithm that can handle input strings of any format. For the five common DNA symbols (A, C, G, T as well as N), a seven bit encoding for three consecutive symbols is used. Each remaining non-standard symbol is encoded with seven bits per symbol. This way, up to 128 additional symbols can be encoded. The free eighth bit distinguishes whether the byte encodes three of the five basic symbols or a custom character.

GBC, a Java-based GUI for sequence compression, was developed in [100]. Here, run-length encoding is implemented on top of naive 2-bit compression.

[9] also propose to encode three bases with one byte. However, in their compression algorithm they incorporated a sophisticated handling of repetitions of N. The obtained encoding is then compressed using LZ77. The authors claim to factor in concepts of self-chromosomal similarity, cross-chromosomal similarities and identifying longest common subsequences.

Another approach in this class of algorithms build on an Oracle database as back-end [81]. [77] additionally incorporates an algorithm for multi-threaded search in the compressed data. [84] describes a particular run-length encoding scheme. Finally, [99] focuses on the analysis of how to store repeats with variable-size codes.

2.3.2 Dictionary-based Algorithms

Comrad, proposed in [67], performs multiple passes over the input data. Each pass discovers longer sequences and incrementally enriches the dictionary. The algorithm is run until the dictionary does not change any more or a frequency threshold is reached. The created dictionary is used to encode the DNA sequences and partially stored alongside compressed data. Note that this encoding allows for random access to the compressed DNA sequences.

[6] present a splay tree-based compression algorithm. Splay trees are self-adjusting binary search trees in which recently accessed elements are quick to access. This property can improve performance when facing local frequency variations. Splaying refers to the process of rearranging or rotating the tree such that a particular element is placed at the root of the tree. Similar to Huffman trees, symbols close to the root have shorter codes then symbols in leaf nodes.

In [56], the problem of non-uniform distribution of DNA over the genome is addressed by splitting the input into blocks and encode each block separately. A hash-table based approach is used to detect repeats.

2.3.3 Statistical Algorithms

[33] present XM, a compression algorithm using repetition detection and statistics on subsequences. The idea is to have a set of experts predicting the next symbol in a sequence based on different heuristics. Different experts are run in competition and the expert with the shortest possible encoding is used to encode the next symbol. Based on their previous success in encoding, the experts obtain weights. The types of experts are:
• **Markov-expert**: A k-order Markov model, which predicts the probability of a symbol based on the last k symbols. By default, this approach employs a second order model for DNA and a first order model for protein sequences.

• **Context Markov-expert**: A first order Markov model which only uses the last 512 symbols to compute probability distributions. The rationale behind this is that different areas in the sequence might serve different functions and should therefore feature different distributions of symbols.

• **Copy expert**: An expert that considers the next symbol to be part of a repeated region copied from a region with a certain offset.

• **Reverse expert**: Functions similarly to the copy expert yet for complementary symbols.

The statistics underlying these experts do not have to be stored for decompression, since they can be reconstructed at runtime. Predictions are combined via Bayesian hashing. Encoding and decoding times are similar, since both processes apply the same procedure. [98] propose another approach based on combining different Markov models. The authors use six different Markov models the orders 1, 4, 6, 10, 14, and 16.

Gene-Compressor was proposed in [116] as a means to encode non-repetitive parts of DNA sequences. In an initial run, probabilities of symbols are estimated and a corresponding Huffman encoding is chosen. Then, the Huffman-encoded output is split into blocks. Finally, each block is restructured in a way that allows for an efficient run-length encoding, which is employed in a final step.

In [112], the input sequence is fragmented into non-overlapping blocks. For each block, a set of experts competes for encoding: a first order Markov model, the naive two-bit representation of bases, and an approximate repetition finder which identifies repetitions interrupted only by few SNPs. The expert which emits the shortest code length is selected and compressed output undergoes further arithmetic compression. Unfortunately, this approach can only handle four base symbols in input sequences.

[57] propose to encode non-repetitive regions as well as mismatches in repeats (single SNPs) with an arithmetic coder based on a Markov model. The resulting bit stream is split into blocks to allow for random access.

### 2.3.4 Referential Algorithms

[12] proposes to only store the differences between a to-be-compressed input sequence and a reference sequence. They consider three kinds of single-base differences: inserts, deletes, and replacements. The main contribution of their work is an analysis on how to encode integers for absolute and relative reference positions. In particular, they compare fixed (Golomb, Elias) and variable (Huffman) entropy coding formats and report that Huffman encoding achieves slightly better results than Golomb and Elias codes. However, the authors stress that the choice of the reference sequence has more impact on the compression ratio than the actual integer coding scheme.

Similarly, [20] present a referential compression algorithm, which only considers SNPs and multi-base INDELs between input and reference sequences. Each compression entry consists of a positional reference and additional data like match length or raw base sequence. Variable length integers are used to encode positions, where the last bit in a byte is used as a stop bit. If the stop bit is not set, the next seven bits are concatenated to previous bits. An alternative
is discussed by encoding only the deltas between consecutive integers. Huffman encoding is used to compress common k-mers. The authors do not provide a comparative evaluation, but show how each of their own optimizations improve the compression rate. Besides the reference sequence, the algorithm needs a reference SNP map of a size of roughly 1 GB.

GRS [125] is a referential compression tool based on the Unix program *diff*, which attempts to find longest common subsequences in two input strings. In GRS, *diff* is used to compute a similarity measure between an input chromosome and a reference chromosome. If similarity exceeds a given threshold, the difference between input and reference sequences is compressed using Huffman encoding. Otherwise, the input and reference chromosomes are split into smaller blocks and the computation is restarted on each pair of blocks. Note that the user is required to pick an appropriate reference, which can be a difficult task (see above).

In [69], RLZ, an approach based on self-indexing is described. It works as follows: the algorithm compresses input sequences with LZ77 encoding relative to the suffix-array of a reference sequence. Raw sequences are never stored; even very short matches to the reference are encoded. The authors state that careful consideration of the reference sequence is vital, since initial results with cross-species compression are discouraging. In [70], RLZopt is presented as an extension of RLZ. The key aspect is longest increasing subsequence computation that allows to efficiently encode positions. It incorporates several improvements, including local look-ahead optimization. RLZopt supports random access queries.

An LZ77-style compression scheme based on RLZopt was recently proposed in [43]. The main difference is that more than one reference sequence is taken into account and a way for encoding approximate matches in introduced. Also, the Lempel-Ziv parsing scheme originally based on hashing is slightly altered in that the algorithm considers trade-offs between the length of matches and distance between matches. Compression is performed on input blocks with shared Huffman codes, enabling random access. The reference sequence for [43, 70] is taken from the set of input sequences.

GReEn, an expert-based reference compression scheme was recently proposed in [97]. Inspired by the non-referential compression scheme XM, GReEn features a copy expert, which tries to find matching k-mers between input and reference sequences. Raw characters in the form of arbitrary ASCII characters are encoded with arithmetic encoding. The authors distinguish a special case, where input and reference sequences have equal length. In this case, GReEn assumes that sequences are already aligned and merely encodes SNPs.

Further referential compression approaches include: a web-based system[60], another LZ77-style compression scheme with random access[64], and permanent index-structure based [94] and [92]. The important question of choosing a good reference sequence is discussed in [3] and [4]. The authors implemented a sequence alignment tool in MATLAB, which can be utilized to compute a variant of edit distance between pairs of sequences. The sequence exhibiting minimum entropy within the list of edits is chosen as a reference. The time complexity was reported to be quadratic.

[68] investigated the problem of constructing custom reference sequences. The main idea is to identify large repeat regions from different sequences, based on dictionaries for these sequences. Then a reference sequence is constructed from detected repeats. This technique might have the potential to overcome problems with inter-species compression of genome data.
2.4 Read compression

Compression of entire genomes, as discussed in the previous section, is mostly applied in projects where genomes are first assembled and then stored in assembled form. However, in re-sequencing projects the step of assembly is often omitted, also due to the rather short reads in current next generation sequencing devices. Instead, reads are aligned directly to the reference, and this alignment (which usually covers each position of a genome multiple times) is used for further processing such as SNP detection [14]. However, the original reads usually are also kept, for instance to allow re-alignment when new and more accurate references become available. Therefore, compressing read sets is also an important topic.

Besides the fact that reads typically are short, may map anywhere on a genome, and are associated to a ID, the main difference between genome compression and read compression are quality scores. Since DNA sequencing is prone to errors, produced reads have a quality score associated with each sequenced base. This score indicates the probability of the base at this position and is thus an indicator for the probability of an error at this position. Quality scores are important for methods like SNP detection, since they influence an algorithm’s decision whether a given mismatch in a read to the reference is a sequencing error or a true SNP.

Most devices produce Phred-like scores, which means that scores are one from 94 different values. An exemplary set of reads, as a FASTQ file, is given in Figure 2.5. Each entry in the FASTQ file consists of four lines: a sequence identifier, the raw sequence of bases, a possible repetition of the first line, and a quality score for each base of the raw sequence.

Clearly, the entropy of a quality score is much higher than that of the actual base, making the achievement of high compression rate much more challenging than in genome compression. Presumably, this is the reason that we are not aware of any compression algorithms based exclusively on naive bit manipulation. On the other hand, downstream algorithm not necessarily need the full quality information, which makes lossy compression more attractive. Table 2.4 gives an overview of compression rates for recent algorithms in non-referential sequence read compression, while Table 2.5 compares compression rates for referential compression schemes.
<table>
<thead>
<tr>
<th>Tool</th>
<th>Data source</th>
<th>Compression ratio (seq)</th>
<th>Compression ratio (quality)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[18]</td>
<td>233 sequences (not published)</td>
<td>8:1 – 11:1 (sequence and quality scores)</td>
<td></td>
</tr>
<tr>
<td>[30] DSRC&lt;sup&gt;a&lt;/sup&gt;</td>
<td>9 files from 1000 Genomes (human)</td>
<td>3.23:1 – 6.51:1 (sequence and quality scores)</td>
<td></td>
</tr>
<tr>
<td>[49]</td>
<td>88 short yeast sequences</td>
<td>1.658:1</td>
<td>--</td>
</tr>
<tr>
<td>[82] CASToRe</td>
<td>14 genomes of different species</td>
<td>1.909:1 – 2.056:1</td>
<td>--</td>
</tr>
<tr>
<td>[113] G-SQZ&lt;sup&gt;b&lt;/sup&gt;</td>
<td>6 files from 1000 Genomes (human)</td>
<td>2.85:1 – 5.35:1 (sequence and quality scores)</td>
<td></td>
</tr>
<tr>
<td>[119]&lt;sup&gt;c&lt;/sup&gt;</td>
<td>three large datasets from SRA</td>
<td>--</td>
<td>2.5 bits per base (lossless)</td>
</tr>
<tr>
<td>[119]</td>
<td>(human, mouse)</td>
<td>--</td>
<td>1 bit per base (lossy)</td>
</tr>
<tr>
<td>[133] POMA</td>
<td>11 short gene sequences</td>
<td>1.3:1</td>
<td>--</td>
</tr>
</tbody>
</table>

<sup>a</sup>http://sun.aei.polsl.pl/dsrc/<br>
<sup>b</sup>http://public.tgen.org/sqz<br>
<sup>c</sup>http://www.cb.k.u-tokyo.ac.jp/asailab/members/rwan
### Reference-based Compression Algorithms

Table 2.5: Referential compression schemes for sequence reads.

<table>
<thead>
<tr>
<th>Tool</th>
<th>data source</th>
<th>compression ratio</th>
<th>Speed of compression (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[27] GenCompress&lt;sup&gt;a&lt;/sup&gt;</td>
<td>three heterogeneous datasets</td>
<td>9.8:1 – 22.7:1 (seq only)</td>
<td>52 – 80</td>
</tr>
<tr>
<td>[36]&lt;sup&gt;b&lt;/sup&gt;</td>
<td>two samples (human, bacteria)</td>
<td>28:1 – 65:1 (seq),</td>
<td>–</td>
</tr>
<tr>
<td>[36]&lt;sup&gt;c&lt;/sup&gt;</td>
<td>two samples (human, bacteria)</td>
<td>25:1 – 48:1 (seq and lossy qual)</td>
<td></td>
</tr>
<tr>
<td>[63] SLIMGENE</td>
<td>human short reads</td>
<td>39:1 (seq), 2.45:1 (qual)</td>
<td>11</td>
</tr>
</tbody>
</table>

<sup>a</sup>Source code available upon request

<sup>b</sup>Source code is supplementary of original submission

<sup>c</sup>Source code is supplementary of original submission
2.4.1 Dictionary-based Algorithms

A dictionary-based approach based on metasymbols was proposed in [49]. Metasymbols are subsequences consisting of regular alphabet symbols and a gap symbol that matches any alphabet symbol. The authors present an algorithm which identifies a set of metasymbols frequently appearing in multiple sequence reads. This dictionary of metasymbols is called a metadictionary and is iteratively refined using a genetic approach, which results in consecutively higher compression rates.

[82] developed CASToRe, a modification of Lempel-Ziv compression [134]. CASToRe compares sequences against the dictionary, registering new dictionary entries as concatenations of two previously parsed subsequences in the dictionary. This is different to standard Lempel-Ziv compression, in which new dictionary entries are composed of an existing dictionary entry and a single mismatching symbol. One major insight of the paper is that one can categorize genomes by compression statistics.

[133] proposed POMA, a particle swarm optimization-based algorithm for sequence read compression. They differentiate between four distinct kinds of repeat patterns: direct, mirror, pairing, and inverted repeats. Most commonly repeated fragments are identified and added to a dictionary. This procedure is observed and influenced by a learning particle swarm optimizer and an adaptive intelligent single particle optimizer.

2.4.2 Statistical Algorithms

[10] propose a non-referential lossless compression scheme for sequence reads in FASTQ format. The main idea is to split the FASTQ file into four streams and compress each stream separately, using different experts. The four streams correspond to the four components of a sequence read in FASTQ, namely sequence identifier, raw sequence, description, and quality scores. For all four streams, symbol distribution statistics are gathered and an appropriate encoding is chosen for each sequence. Sequence identifiers and descriptions are investigated for redundant information. Raw DNA sequences are compressed using repeat detection and a Markov expert. Assembled dictionaries can be reconstructed at runtime during decompression. Quality scores are compressed with one out of six different delta or run-length encodings.

The major limiting factor when compressing FASTQ files is the quality information. Therefore, [119] focus solely on lossy and lossless compression of quality scores in FASTQ data sets. Phred quality scores are normalized and the frequency distribution is determined. Later, a variety of different encoding techniques for the quality values is compared. The authors show that lossy transformation of quality scores can greatly reduce storage costs, while losing only little information.

The importance of quality information was also addressed in [113] in their compression scheme G-SQZ. The core idea is that bases and quality values are assumed to correlate and can therefore be put together into one byte. An initial scan generates a Huffman code for each (base, quality) pair. In a second scan the Huffman codes are written to a binary file.

[30] describes DSRC, a block-based compression scheme which enables random access for sequence reads. The FASTQ file is split into three streams for separate compression, one each for sequence identifiers, raw bases, and quality scores, respectively. DSRC encodes additional symbols with unassigned quality values in the quality score compression stream. DNA sequence reads are encoded with a LZ77-style compression scheme, in which hashes for reads of length 36 are generated for fast lookup. The authors noticed two patterns commonly appearing in quality score streams, each of which is encoded using a different heuristic: Quasi-random qual-
Reference-based Compression Algorithms

ity score sequences are compressed using different variations of Huffman coding. Repetitive quality streams on the other hand are compressed via run-length encoding.

In [18], Fibonacci codes are used to encode length information (authors claim that Huffman would be too slow for their use case) and 2-bit encoding is used for describing mismatch nucleotides. The main feature of this work is that it presents a complete sequence compression system (and not just an algorithm), including data management and a graphical user interface.

2.4.3 Referential Algorithms

As reads usually are anyway first aligned against a reference genome, usage of referential compression schemes is a straight-forward approach towards their compression. However, mapping a read against a genome for referential compression is also very different from mapping it for further analysis: First, algorithms for the former are free to chose any mapping (and need just one), while methods for the latter are bound to find the best matches (and all of them). Second, the former must take quality scores into account to achieve high compression rates, while the latter often ignore these scores during the alignment.

[27] presented GenCompress. Sequence reads are aligned to a reference sequence with reference entries being composed of a starting position, the match length, and an optional difference list describing mismatches. Since the ends of reads are more prone to sequencing errors, base mismatches are indexed from the end of reads, resulting in smaller integers on average. The focus of the article is on entropy encoding of integers via fixed or variable encoding, such as Golomb, Elias, or Huffman. The authors performed their evaluation on a single chromosome and extrapolated results for the whole genome. GenCompress only supports compression of the four bases and is not able to handle additional symbols, quality scores or unaligned reads.

Following a similar approach, [63] propose SLIMGENE, a lossless or lossy reference-based compression scheme focusing on how to find encodings of integers in order to minimize storage. In their work, they employ Huffman and arithmetic encoding.

[36] presented a compression scheme inspired by image compression techniques based on controlled loss of precision. The positions of matching bases are stored in the form of Huffman-encoded integers. Reads are ordered based on the position in the reference and these positions are delta-encoded using Golomb encoding. The paper also proposes compression of quality scores in the form of quality budgets. A quality budget denotes a trade-off between storage cost and accuracy of quality scores.

2.5 Discussion and Conclusions

In this chapter, we reviewed recent progress concerning DNA compression. We identified four different classes of compression schemes and described a multitude of different algorithms within each class. Furthermore, we highlighted the important differences between genome compression and read compression and separately discussed respective approaches. We found that often novel approaches are only slight variations of each other, which further helps to structure the at first sight highly heterogeneous landscape of different approaches.

The comparisons presented in this chapter are based on the original papers. Clearly, tools ideally should be compared based on their performance as measured on the same hardware using the same data set. Such comparisons are sometimes contained in the original papers, but these often are rather inconclusive, as they only compare to very few other approaches and only using a particular data set with its intrinsic properties like frequency and length of repeats, size,
length of sequences etc. Also, sometimes inappropriate competitors are chosen, like Winzip, or algorithms are compared to others of a different kind, e.g. comparing a statistical with a referential algorithm. We think these comparisons are inappropriate since window sizes of these implementations (chosen many years ago) do not allow to find repeats in (sets of) longer sequences. At the same time, third parties are not able to perform a comprehensive comparison of different tools since most algorithms are not publicly available (see our comparison tables). We therefore see an urgent need for a community effort to define a proper benchmark for DNA compression. Test sequences should come from different species and cover different sizes, from few KB to several hundred GB. The benchmark should also clearly define the metrics to be reported. We think that the following list could be used as a starting point: 1) compression rate, 2) (de)compression time, and 3) maximum main memory usage during (de)compression.

As the flood of sequence data is growing faster than ever, we expect that research into novel compression algorithms will continue to flourish. However, we also believe that not only higher compression rates or faster (de)compression should be addressed, but also other properties of compressed sequence. One particular interesting question is how compressed sequences can be used directly for further analysis. For instance, a referentially compressed read set is very close to an alignment of the read set against the reference; thus, if properly compressed, the actual alignment phase might become superfluous. Also searching in compressed sequence archives is important. Imagine the problem of finding best local alignments of a given sequence in a set of compressed genomes. If these genomes first have to be decompressed for every such search, the space gain of compression is essentially lost. First steps into this direction are reported in [122]. Another interesting research question, to our knowledge yet completely unexplored, is the integration of sequence compression in scientific workflows, which we consider as a prerequisite to fully leverage the advanced computing power of cloud infrastructures.
Chapter 3

FRESCO: Referential Compression of Highly-Similar sequences

Chapter Authors:
Sebastian Wandelt and Ulf Leser (UBER). ¹

In many applications, sets of similar texts or sequences are of high importance. Prominent examples are revision histories of documents or genomic sequences. Modern high-throughput sequencing technologies are able to generate DNA sequences at an ever increasing rate. In parallel to the decreasing experimental time and cost necessary to produce DNA sequences, computational requirements for analysis and storage of the sequences are steeply increasing. Compression is a key technology to deal with this challenge. Recently, referential compression schemes, storing only the differences between a to-be-compressed input and a known reference sequence, gained a lot of interest in this field.

In this chapter, we propose a general open-source framework to compress large amounts of biological sequence data called FRESCO, Framework for REferential Sequence COmpression. Our basic compression algorithm is shown to be 1-2 orders of magnitudes faster than comparable related work, while achieving similar compression ratios. We also propose several techniques to further increase compression ratios, while still retaining the advantage in speed: 1) selecting a good reference sequence and 2) rewriting a reference sequence to allow for better compression. In addition, we propose a new way of further boosting the compression ratios by applying referential compression to already referentially compressed files (second-order compression). This technique allows for compression ratios way beyond state-of-the-art, for instance, 4000:1 and higher for human genomes. We evaluate our algorithms on a large data set from three different species (more than 1,000 genomes, more than 3 TB) and on a collection of versions of Wikipedia pages. Our results show that real-time compression of highly-similar sequences at high compression ratios is possible on modern hardware.

3.1 Introduction

Since the release of the first human genome [24], the cost for sequencing has rapidly decreased. As of now, the price is at approximately 2,000 USD per genome and is expected to fall further once third generation sequencing techniques become available [103]. In contrast to previous years, where typically only one individual of a species was sequenced (like humans, mice,

¹Content of this chapter was previously published in Wandelt and Leser 2013 [123]
E.coli, etc.), the decrease in costs makes it possible to sequence large samples of a given pop-
ulation. Such studies, especially on humans, are interesting from many perspectives, such as
correlation of specific mutations to the risk of developing a disease, to fine-tuned dosages of
therapies, or simply to better understand the relationship between genotype and phenotype.
Examples are the 1000-Genomes project [1]; activities of the international cancer sequencing
consortium [25]; and the UK10K project [13]. These large-scale projects are generating com-
prehensive surveys of the genomic landscape of various diseases by sequencing thousands of
genomes [131]. Managing, storing and analyzing this quickly growing amount of data is chal-
 lenging [55]. It requires large disk arrays for storage, and large compute clusters for analysis.
A recent suggestion is to use cloud infrastructures for this purpose[37, 104, 110]. However, be-
fore being analyzed in a cloud, data first has to be shipped to the cloud, making bandwidth
in file transfer one of the major bottlenecks in cloud-based DNA analysis [114]. Accord-
ingly, sequence compression is a key technology to cope with the increasing flood of DNA
sequences [86, 96].

To store a complete genome of a human being, one needs roughly 3 GB of space, using 1
Byte per nucleotide. Since 8 Bits can encode 256 different symbols in total, this space can be
reduced by encoding each nucleotide with less than 8 Bits. Substitutional or statistic compres-
sion schemes can reduce the space requirements by up to 6:1 (one base is encoded with up to
1.3 Bits) [6, 98]. However, in many projects only genomes from one species are considered.
This means that projects often deal with hundreds of highly similar genomes; for instance, two
randomly selected human genomes are identical to an estimated 99.9%. Similarity between
biological sequences can be exploited using so-called referential compression schemes [20],
which encode the differences of an input sequence with respect to a reference sequence. Using
space-efficient encoding of differences and clever algorithms for finding long stretches of DNA
without differences, the best current referential compression algorithm we are aware of reports
a compression ratios between 500:1 and 1000:1 for human genomes [31].

In this chapter, we propose FRESCO, a Framework for REferential Sequence COmpression.
It builds on a fast referential compression algorithm and its source code is released for free
extension by the community. Our implementation achieves similar compression rates as existing
referential compression implementations, while being at least one order of magnitude faster. In
addition, we discuss three methods on how to increase compression ratios in FRESCO:

- Selection of a reference: We show that the choice of the reference has an impact on
  the compression ratio. Our new approach is to analyze already referentially compressed
  sequences for choosing a good reference. This can decrease the size of compressed se-
  quences by up to 12 percent.
- Rewriting a reference: Our new approach is to analyze already referentially compressed
  sequences and extract frequently occurring mismatches with respect to the reference. In
  a second step, the reference is rewritten based on most often occurring mismatches. This
  can decrease the size of compressed sequences by up to 35 percent.
- Second-order compression: We apply referential compression to referentially compressed
  files. This can further decrease the size of compressed sequences by up to 75 percent,
  achieving compression rates of 4000:1 and more for human genomes, while still being
  more than 5 times faster than existing algorithms.

We evaluate our algorithms on datasets from three species: 1,092 human genomes, 180 genomes
of Arabidopsis thaliana, and 38 yeast genomes.

In addition, we show how our compression algorithm can be used to compress non-biological
datasets. Highly-similar documents are often found in version control systems, which have to
store multiple versions of the same document. For instance, Wikipedia stores the history of each
page with up to several thousand versions per page. The differences between two consecutive
versions are often quite small, e.g. removing typos or adding a new single paragraph. In our
evaluation, we show how FRESCO can be applied directly for compressing different versions
of a Wikipedia-page against the base page.

The remaining part of this chapter is structured as follows. We discuss related work on com-
pression of sequences in Section 3.2. We motivate and formally define our data structures and
algorithm for referential compression in Section 3.3. In Section 3.4, we discuss two heuristics
for increasing compression ratio. First, we propose a method to select a very good reference
sequence from a set of candidate sequences, and second, we discuss how to rewrite a fixed ref-
ERENCE to allow encoding of longer matches into the reference for most of the to-be-compressed
sequences. A third new method for increasing compression ratio is presented in Section 3.5,
called second-order compression. We evaluate all our methods in Section 3.6. Section 3.7
describes the open-source release of FRESCO and the chapter is concluded in Section 3.8.

3.2 Related Work

Naive bit encoding algorithms exploit encodings of two or more symbols into one byte, using
fixed-length encodings[17]. A straight-forward technique is the encoding of one base with two
Bits via bit encoding. In this case, the compression ratio is fixed at 4:1. Dictionary-based
algorithms replace repeated substrings by references to a dictionary (a set of previously seen or
predefined common strings), which is built at runtime or offline[67, 73, 107]. Lempel-Ziv-based
compression algorithms, such as LZ77 or LZ78, are prominent examples of dictionary-based
algorithms [134]. These methods achieve compression ratios between 4:1 and 6:1 depending on
the frequency of repeats in the genomes being compressed. Statistical compression algorithms
derive a probabilistic model from the input. Based on partial matches of subsets of the input,
this model predicts the next symbols in the sequence. High compression ratios are possible
if the model always indicates high probabilities for the next symbol, i.e. if the prediction is
reliable[21, 26, 33]. One of the most commonly used and best understood statistical encodings
is Huffman encoding [52]. The compression ratio of statistical algorithms is usually between
4:1 and 8:1.

Referential compression algorithms recently emerged as a fourth type of sequence compres-
sion algorithm. Similar to dictionary-based techniques, these algorithms replace long substrings
of the to-be-compressed input with references to another string. However, these references point
to external sequences, which are not part of the to-be-compressed input data. Furthermore, the
reference is usually static, while dictionaries are being extended during compression phase.
During the last years several referential compression algorithms emerged[31, 64, 69, 70, 97].

In [69], RLZ, an approach based on self-indexing is described. It works as follows: the
algorithm compresses input sequences with LZ77 encoding relative to the suffix-array of a
reference sequence. Raw sequences are never stored; even very short matches to the reference
are encoded. In [70], RLZopt is presented as an extension of RLZ. The key aspect is longest
increasing subsequence computation that allows to efficiently encode positions. It incorporates
several improvements, including local look-ahead optimization. An LZ77-style compression
scheme, called GDC, based on RLZopt was recently proposed in [31]. The main difference is
that more than one reference sequence is taken into account and a way for encoding approximate
matches is introduced. Also, the Lempel-Ziv parsing scheme originally based on hashing is
slightly altered in that the algorithm considers trade-offs between the length of matches and
distance between matches. Compression is performed on input blocks with shared Huffman
Algorithm 2 Referential Compression Algorithm

**Input:** to-be-compressed string $s$ and reference string $ref$  
**Output:** referential compression $\text{comp}(s, \text{ref})$ of $s$ with respect to $\text{ref}$  

1: Let $\text{comp}(s, \text{ref})$ be an empty list  
2: While $|s| \neq 0$ do  
3: Let $\text{pre}$ be the longest prefix of $s$ occurring in $\text{ref}$, and let $i$ be a position of an occurrence of $\text{pre}$ in $\text{ref}$  
4: If $s \neq \text{pre}$ then  
5: Add $(i, |\text{pre}|, s(|\text{pre}|))$ to the end of $\text{comp}(s, \text{ref})$  
6: Remove the first $|\text{pre}| + 1$ symbols from $s$  
7: Else  
8: Add $(i, |\text{pre}| - 1, s(|\text{pre}| - 1))$ to the end of $\text{comp}(s, \text{ref})$  
9: Remove the prefix $\text{pre}$ from $s$  
10: End if  
11: End while

Another LZ77-style compression scheme with random access is proposed in [64]. [36] presented a compression scheme inspired by image compression techniques based on controlled loss of precision. GRS [125], is another tool for referentially compressing whole genome sequences against a user-selected reference. Depending on a sequence similarity score, the to-be-compressed sequence is optionally cut into blocks first. Then, for each block (or the whole input sequence) the longest shared sequence with the reference is extracted. The remaining differences against the reference are encoded with Huffman coding. The authors report compression times of around half an hour for small human chromosomes. GReEn, an expert-based reference compression scheme, was recently proposed in [97]. Inspired by compression scheme XM [33] and the work on GRS, GReEn features a copy expert, which tries to find matching k-mers between input and reference sequences. Raw characters in the form of arbitrary ASCII characters are encoded with arithmetic encoding. The authors report compression rates for human genomes similar to GRS, while being ten times faster on average.

Compression of entire genomes is mostly applied in projects where genomes are first assembled and then stored in assembled form. However, in re-sequencing projects the step of assembly is often omitted, also due to the rather short reads in current next generation sequencing devices. Therefore, compressing read sets is an important topic as well [10, 27, 63, 82, 119].

### 3.3 Referential Compression

In the following, we describe our referential compression algorithm. First, we discuss three different approaches for storing reference entries and second, we describe the compression algorithm.

#### 3.3.1 Representation of Matches into a Reference

A string $s$ is a finite sequence over an alphabet $\Sigma$. Below we use the terms string and sequence as synonyms. The length of a string $s$ is denoted with $|s|$ and the substring starting at position $i$ with length $n$ is denoted $s(i, n)$. $s(i)$ is an abbreviation for $s(i, 1)$. All positions in a string are zero-based, i.e. the first character is accessed by $s(0)$. The concatenation of two strings $s$ and $t$ is denoted with $s \circ t$. A string $t$ is a prefix of a string $s$, if $s = t \circ u$, for some string $u$. A string $s$ is a substring of string $t$, if there exist two strings $u$ and $v$ (possibly of length 0), such that $t = u \circ s \circ v$.

Referentially compressing a string means to encode the string as a concatenation of substrings from a given reference string. There exist several options for choosing a representation
D2.2 – Reference-based Compression Algorithms

of matches to a reference sequence. One obvious choice is to encode a sequence as a set of pairs, where each pair is composed of the position of a match and a length of a match [31, 69]. Another option is to encode parts of a sequence with original text entries instead of matches into the reference [70, 121]. This approach is advantageous if the referential match entries are often very short and therefore a compact representation of the text uses less space than a referential match entry. A third option is to encode each match into a reference as a triple, composed of the start position of a match, the length of a match, and the first character following the match. This approach shows very good results, if to-be-compressed sequence and reference are highly similar and often only differ by single nucleotide polymorphisms (SNPs).

Example 1 Given a reference \( ref = ATGCGAGCT \), sequence \( s = ATTCGAGACT \) could be represented as

- **Option 1:** \([\langle 0, 2 \rangle, \langle 1, 1 \rangle, \langle 3, 4 \rangle, \langle 0, 1 \rangle, \langle 7, 2 \rangle]\)
- **Option 2:** \([\langle 0, 2 \rangle, T, \langle 3, 4 \rangle, A, \langle 7, 2 \rangle]\)
- **Option 3:** \([\langle 0, 2 \rangle, T, \langle 3, 4 \rangle, A, \langle 7, 1 \rangle, T]\)

The strings \( ref \) and \( s \) have an edit distance of 2 (replacing one \( G \) with a \( T \), inserting an \( A \)). Although both strings are quite similar, we have already 5 entries for Option 1 and Option 2. Entry \( \langle 0, 1 \rangle \) is an example for a spurious reference match in Option 1, which can be avoided. In this work, we pick the third option for representing matches into a reference, since many differences of biological sequences belonging to the same species are often caused by SNPs.

We have chosen the following definition of a match into a reference:

**Definition 1** A referential match entry (RME) is a triple \( \langle \text{start}, \text{length}, \text{mismatch} \rangle \), where \( \text{start} \) is a number indicating the start of a match within the reference, \( \text{length} \) denotes the match length, and \( \text{mismatch} \) denotes a symbol. The length of a referential match entry \( rme \), denoted \( |rme| \), is \( \text{length} + 1 \).

**Definition 2** Given strings \( s \) and \( ref \), a referential compression of \( s \) with respect to \( ref \), is a list of referential match entries,

\[
\text{comp}(s, ref) = [\langle \text{start}_1, \text{length}_1, \text{mismatch}_1 \rangle, ..., \langle \text{start}_n, \text{length}_n, \text{mismatch}_n \rangle],
\]

such that

\[
\text{ref}(\text{start}_1, \text{length}_1) \circ \text{mismatch}_1 \circ \text{ref}(\text{start}_2, \text{length}_2) \circ \text{mismatch}_2 \circ ... \circ \text{ref}(\text{start}_n, \text{length}_n) \circ \text{mismatch}_n = s.
\]

Sometimes we use \( rc \) instead of \( \text{comp}(s, ref) \), if \( s \) and \( ref \) are known from the context. The offset of a referential match entry \( rme \) in a referential compression \( \text{comp}(s, ref) = [rme_1, ..., rme_n] \) corresponds to the position of the entry in the uncompressed string and is denoted with \( \text{offset}(\text{comp}(s, ref), rme_i) \). Given a referential match entry \( \langle \text{start}, \text{length}, \text{mismatch} \rangle \), we write the expression \( \langle \text{start}, \text{length}, \text{mismatch} \rangle \in \text{comp}(s, ref) \), if and only if \( \langle \text{start}, \text{length}, \text{mismatch} \rangle \) is an element in the referential compression \( \text{comp}(s, ref) \).

The inverse of a referential compression is the decompression of a referential compression with respect to the reference, such that we obtain the original input string.
D2.2 – Reference-based Compression Algorithms

Example 2 An example referential compression for the string CGGACAAACTGACGTTCG ACG with respect to the reference GACGATCGACGACGGACAAACA is shown in Figure 3.1. The input is compressed into three referential match entries. The first referential match entry is \((12, 9, T)\), which describes a match for the string CGGACAAACT at position 12 of the reference. The mismatch character is \(T\) (in the reference an \(A\) is found instead of a \(T\)). Although the string of the last RME can be completely found in the reference, we only encode the first five symbols as a link to the reference and add \(G\) as a mismatch symbol. Alternatively, the last RME could also be encoded as \((5, 6, \_\)\), where \(\_\) is a special symbol not occurring in the input alphabet. We think that algorithms working with compressed representations (for instance when searching compressed sequences) are slightly easier to implement without the introduction of such a special mismatch symbol. The offset of referential match entry \((5, 5, G)\) is \(|(12, 9, T)| + |(10, 4, T)| = 15\).

3.3.2 Compression Algorithm

The less referential match entries we require, the longer the matches (i.e. the shared substrings). Therein, for long matches, it does not matter, at which position of the reference these matches lie; because the gain from compressing a long match as a referential match entry easily outweighs the space for representing the position of a match. We exploit this observation in Algorithm 2. To create a referential compression of input string \(s\) with respect to \(ref\), the algorithm matches prefixes of \(s\) with substrings of \(ref\) using a compressed suffix tree on \(ref\). The longest such prefix is removed from \(s\), encoded as a RME and added to \(comp(s, ref)\). The algorithm terminates once \(s\) contains no more symbols. Please note that a referential compression of a string with respect to a reference is not unique. A simple example for a non-unique referential compression with respect to the reference \(ref = A\_A\) is \(\text{comp}(AA, ref) = [(0, 1, A)]\) and \(\text{comp}(AA, ref) = [(0, 1, A)]\).

Algorithm 2 is a greedy algorithm, i.e. it always takes the longest prefix of the to-be-compressed sequence which can be found in the reference. The compression algorithm runs in \(O(n)\), where \(n\) is the maximum length of the strings (reference and to-be-compressed). Any greedy algorithm computes a minimal representation, if the dictionary is fixed and the size of a dictionary entry is constant [22]. Since we apply a kind of delta-encoding for storing positions, the algorithm is not optimal. In delta-encoding the position of a RME is encoded as the difference to the position of the previous RME plus its length plus 1, for instance, \((5, 5, G)\,(12, 5, G)\) is stored as \((5, 5, G)\,(1, 5, G)\), since \(12 - (5 + 5 + 1) = 1\). If to-be-compressed string and reference string are highly similar, this delta-encoding reduces space requirements for compressed representations by up to \(\frac{4}{5}\) in our experiments for human genomes. Experiments for small
Algorithm 3 Reference Selection RSbitX

**Input:** set of to-be-compressed sequences \( s_1, \ldots, s_n \), set of candidate reference sequences \( ref_1, \ldots, ref_m \), a base reference sequence \( ref_{base} \), and a speedup value \( X \)

**Output:** index \( b \) for best reference

1: Compute \( \text{comp}(ref_i, ref_{base}) \) for all \( 1 \leq i \leq m \)
2: for \( 1 \leq j \leq n \) do
3: Split \( s_j \) into 1000 blocks \( b_1, \ldots, b_{1000} \) of equal length
4: Let \( sx_j \) be the concatenation of each \( X \)-th block of \( b_1, \ldots, b_{1000} \)
5: end for
6: Compute \( \text{comp}(sx_j, ref_{base}) \) for all \( 1 \leq j \leq n \)
7: for \( 1 \leq i \leq m \) do
8: Let \( \text{val}_i = 0 \)
9: for \( 1 \leq j \leq n \) do
10: \( \text{val}_i = \text{val}_i + |\text{rsim}(\text{comp}(sx_j, ref_{base}), \text{comp}(ref_i, ref_{base}))| \)
11: end for
12: end for
13: Find the smallest \( \text{val}_{min} \) from \( \text{val}_1, \ldots, \text{val}_m \) and let \( b = \text{min} \)

strings show that the results of greedy compression algorithms are fairly close to non-greedy algorithms [50]. Note that Algorithm 2 is lossless, i.e. we can recreate the original string completely from the compressed representation. The decompression of a single RME is the substring of the reference string with the mismatch character concatenated to the end. For decompression of a referentially compressed string, we traverse the referential compression from left to right and replace each RME with its decompressed string.

### 3.4 Improving Compression Ratios

The reference sequence is the main factor determining compression ratios, given a fixed encoding of referential match entries. For instance, if a human genome is referentially compressed against a mouse genome, the ‘compressed’ output is actually larger than the human input genome. This is caused by many very short referential match entries (around 12 bases long); for each entry we have to encode a position, length, and mismatch character.

Even inside a species, the reference sequence has a significant impact on the compression ratio, for instance, if the reference and to-be-compressed input are closely related by ancestral relationships. With increasing similarity between reference and to-be-compressed sequence, longer referential match entries can be found and the compression ratio is increasing.

**Definition 3** Let \( \text{serSize}(s, ref) \) be the serialized size of \( \text{comp}(s, ref) \). For a collection \( S = \{ s_1, \ldots, s_n \} \), let \( \text{serSize}(S, ref) = \sum_{i \leq n} \text{serSize}(s_i, ref) \). The problem of finding an optimal reference for \( S \) is defined as follows: Find a reference \( ref_1 \), such that there does not exist a reference \( ref_2 \) with \( \text{serSize}(S, ref_2) < \text{serSize}(S, ref_1) \).

Note that we leave the definition of serialized size open: it could be the number of referential match entries or the number of bytes necessary for storage. Finding an optimal reference for a collection of sequences is a hard problem: there are \( 4^n \ (5^n \), including \( N \)) possible references of length \( n \). Since the length of a chromosome is up to several hundred megabases, an exhaustive enumeration of all reference sequences is impossible. In the following, we describe two heuristics for the problem of finding an optimal reference. The first technique, reference selection, restricts the set of reference candidates. The second technique, reference rewriting, improves an existing reference by rewriting it based on the to-be-compressed sequences.
3.4.1 Selecting a good Reference

First, we discuss the selection of a best reference sequence for a single to-be-compressed sequence.

**Definition 4** Given a sequence \( s \) and a set of candidate references \( \{ref_1, ..., ref_m\} \), \( ref_i \) is called a best reference iff there does not exist a \( j \neq i \) with \( |\text{comp}(s, ref_j)| < |\text{comp}(s, ref_i)| \), where \( |X| \) denotes the size of a referentially compressed sequences \( X \).

Note that there can exist more than one best reference, in which case we would randomly choose one. In our experiments this case never occurred.

A naive strategy to find the best reference sequence is to compress all the to-be-compressed sequences against all possible reference sequences and select the reference that yields the least number of referential match entries, named RSbest. If sequences are long, as in our case, this is a highly time consuming undertaking as we need to compute \( n \times m \) referential compressions, where \( m \) is the number of candidate reference sequences and \( n \) is the number of to-be-compressed sequences. If one wants to compress 1,000 sequences, choosing the best reference following this strategy would take several weeks; however, we shall use this strategy on a sample to evaluate the heuristics described next.

Our approach to solving the problem is as follows: Instead of compressing a to-be-compressed sequence against all candidate references, we compare the referential compression of the sequence and the referential compression of the reference candidates with respect to one randomly chosen initial reference. This heuristic only needs to compress each sequence one time with respect to the initial reference, independent of the number of candidate references. The candidate references are chosen randomly. Before introducing our selection heuristics in detail, we first define the similarity of two referential compressions. The idea is that two referential compressions are defined to be more similar if they share more referential match entries.

**Definition 5** The referential similarity of two referential compressions \( rc_1 \) and \( rc_2 \), denoted \( \text{rsim}(rc_1, rc_2) \), is defined as \( \text{rsim}(rc_1, rc_2) = |rc_1 \cup rc_2| - |rc_1 \cap rc_2| \).

Please note that a lower \( \text{rsim} \)-value indicates higher similarity. Two identical referential compressions will have a \( \text{rsim} \)-value of 0. We propose a heuristic for reference selection named RSbitX, which is shown in Algorithm 3. The heuristic follows the same pattern as RSbest, with two differences:

1. We compress to-be-compressed input sequences not against each candidate reference, but only against one chosen base reference sequence \( ref_{base} \). Therefore, the referential compressions used in the inner loop for \( \text{rsim} \)-computation, i.e. \( \text{comp}(s_j, ref_{base}) \) and \( \text{comp}(ref_i, ref_{base}) \) do not have to be recomputed on each iteration.
2. We only partially compress each sequence, hoping that the similarity of partial compressions is representative for the complete sequences. \( X \) determines how much of each sequence is used for partial compression. Each sequence is broken up into 1,000 blocks of equal length and then \( \frac{1}{X} \) of the blocks are used for partial compression (all blocks are taken in case of \( X = 1 \)). We distribute the blocks for partial compression equally over the whole input sequence.

While RSbest needs to compute \( m \times n \) referential compressions, RSbitX only needs to compute \( m + n \) referential compressions, and if \( X > 1 \), then we (roughly) only need to compute \( m + \frac{n}{X} \) referential compressions. The time is reduced by a factor of \( \frac{m \times n}{m + \frac{n}{X}} \), compared to the selection of the best reference. This assumes that the process of compressing a sequence has linear time...
complexity and neglects possible overhead for setting up the data structures for the compression of a sequence.

In our experiments with different numbers of blocks we obtained very similar results. If the block size is small (smaller than 10,000 Bytes), then, for human genomes, the reference selection yields similar results like a random selection strategy. We think that this is caused by larger indels in the datasets (similar regions between two sequences do not end up in the same block). If the number of blocks is smaller than 1,000, then the gain in compression speed is lost. For our datasets 1,000 blocks turned out to be a good compromise.

### 3.4.2 Reference Rewriting

One other approach we investigate is to rewrite a reference sequence in a way that it represents a most likely path through all sequences in the collection of to-be-compressed sequences. In this scenario the number of candidate reference sequences is fixed to one. Rewriting sequences has a biological motivation: different SNPs in a population occur with different frequencies. With reference rewriting we try to identify and apply most-frequent SNPs to the reference. We consider an example first.

**Example 3** Referentially compressing the sequences

\[s_1 = AAAACGGACAATCTGA\]
\[s_2 = AAAACGGACAATCTGT\]
\[s_3 = AAAACGACAATCTGT\]

with respect to the reference \[AAAACGCACAATCTGC\], we obtain the following three referential compressions:

\[rc_1 = \{\langle 0, 6, G\rangle, \langle 7, 8, A\rangle\}\]
\[rc_2 = \{\langle 0, 6, G\rangle, \langle 7, 8, T\rangle\}\]
\[rc_3 = \{\langle 0, 6, A\rangle, \langle 8, 7, T\rangle\}\].

If the seventh position of the reference string contained a \(G\) instead of a \(C\), then it would be possible to compress \(rc_1\) and \(rc_2\) using only one entry each: \(rc_1^{new} = \{\langle 0, 15, A\rangle\}\), \(rc_2^{new} = \{\langle 0, 15, T\rangle\}\).

As can be seen from simple Example 3, it can be beneficial to rewrite the reference sequence in order to reduce the number of referential match entries and thus increase compression ratios. Rewriting steps need to be carefully considered. With a large set of strings, it is highly unlikely that all sequences agree on particular base replacements/inserts/deletions with respect to a reference. However, even if the majority of sequences share the same base deviations from the reference, compression ratios can be improved. Example 3 shows further that we cannot blindly rewrite a reference, since not all sequences agree on the seventh position.

In the following we describe a heuristic for rewriting reference sequences. Our evaluation will show that this rewriting can indeed save up to 20 percent of space on real-life sequences. We identify a set of replacement candidates from a given (set of) compressed sequences. In the remaining part of the work, we will focus on single base rewritings which are either base replacements, base insertions, or base deletions; longer changes are left for future work. Since referential match entries store the mismatches with respect to the reference, replacement candidates are easy to find. The formal criteria for a replacement rewrite candidate is the existence...
Algorithm 4 Reference Rewriting Algorithm

Input: set of referential compressions \( S = \{rc_1, ..., rc_n\} \), a reference string \( ref \), and a threshold \( t \)
Output: rewritten reference \( result \)
1: Let \( result \) be an empty string
2: for \( 1 \leq p \leq |ref| \) do
3: if there exists a most frequent rewrite candidate \( (X, p, c) \) for \( p \) in \( S \), with \( freq((X, p, c), S) \geq t \) then
4: if \( X=\text{REPL} \) then
5: Append \( c \) to \( result \)
6: else if \( X=\text{INS} \) then
7: Append \( c \) to \( result \)
8: Append \( ref(p) \) to \( result \)
9: else if \( X=\text{DEL} \) then
10: do nothing
11: end if
12: else
13: Append \( ref(p) \) to \( result \)
14: end if
15: end for

Example 4 Given Example 3, we have that \( \text{rewr}(rc_1) = \{ (\text{repl}, 6, G) \} \), \( \text{rewr}(rc_2) = \{ (\text{repl}, 6, G) \} \), and \( \text{rewr}(rc_3) = \{ (\text{del}, 6, \_ ) \} \).

The frequency of \( (\text{repl}, 6, G) \) is \( \frac{2}{3} \), i.e. the replacement occurs in two of three compressed strings. The frequency of \( (\text{del}, 6, \_ ) \) is \( \frac{1}{3} \). The most frequent rewrite candidate for position 6 is therefore \( \text{repl}, 6, G \).

The most frequent rewrite candidates for each position in the reference are used to rewrite the reference sequence. Our reference rewriting algorithm is shown in Algorithm 4. The input of the algorithm is a set of referential compressions \( S \), a to-be-rewritten reference sequence \( ref \), and a threshold \( t \). The threshold is used to only select rewrite candidates, which have at least a given relative frequency in \( S \). The algorithm iterates over the reference sequence and checks for each position in the reference, to determine if a most-frequent rewrite candidate

of two consecutive referential match entries, for instance \((0, 6, C)\) and \((7, 8, A)\) in Example 3, such that a replacement with the mismatch character in the reference will yield one combined long interval, instead of two short ones.

Definition 6 A tuple \((\text{repl}, p, c)\) is called a replacement candidate for a referential compression \( rc \), if there exists two consecutive RME \([p_1, l_1, c], [p_2, l_2, c_2] \) \( \in rc \) with \( p_1 + l_1 + 1 = p_2 \land p = p_1 + l_1 \). A tuple \((\text{ins}, p, c)\) is called an insert candidate for a referential compression \( rc \), if there exists two consecutive RME \([p_1, l_1, c], [p_2, l_2, c_2] \) \( \in rc \) with \( p_1 + l_1 + 1 = p_2 \land p = p_1 + l_1 \). A tuple \((\text{del}, p, c)\) is called a deletion candidate for a referential compression \( rc \), if there exists two consecutive RME \([p_1, l_1, c], [p_2, l_2, c_2] \) \( \in rc \) with \( p_1 + l_1 + 2 = p_2 \land p = p_1 + l_1 \). The rewrite candidates of a referential compression \( rc \) with respect to a reference \( ref \), denoted \( \text{rewr}(rc) \), are the union of all replacement candidates, insert candidates, and deletion candidates of \( rc \).

Definition 7 Given a set of referential compressions \( S = \{rc_1, ..., rc_n\} \) with respect to a reference \( ref \), the relative frequency of a rewrite candidate \( (X, p, c) \) is defined as

\[
freq((X, p, c), S) = \frac{|\{rc_i \mid rc_i \in S \land (X, p, c) \in \text{rewr}(rc_i)\}|}{|S|}.
\]

Given a position \( p \), the most frequent rewrite candidate for \( p \) in \( S \) is \((X, p, c)\), if there does not exist a \( X^* \in \{\text{repl, ins, del}\} \) and \( c^* \) with \( freq((X^*, p, c^*), S) > freq((X, p, c), S) \). In case of two equally frequent rewrite candidates, one is chosen randomly.
exists whose relative frequency is higher than the given threshold \( t \). If such a rewrite candidate exists, characters are added to the output of the algorithm \( \text{result} \), depending on the rewriting kind (replacement, insertion, deletion). If no such rewrite candidate exists for position \( p \), the algorithm just appends the original base from position \( p \) of the reference to \( \text{result} \). After the execution of the algorithm, \( \text{result} \) contains the rewritten reference sequence. Note that the choice of the initial reference sequence has only a small impact on the compression ratio in our experiments. Furthermore, in our experiments we recompute each referential compression for the rewritten reference sequence. It is an interesting direction of future work to update referential compressions to reflect the changes in the rewritten reference, without the need of recompression.

**Example 5** If we apply Algorithm 4 to Example 4 with threshold \( t = 0.6 \), we obtain the rewritten reference sequence \( \text{AAAACGCACAATCTGC} \), since there exists only one rewrite candidate with a relative frequency larger than 0.6: rewrite candidate \( (\text{repl}, 6, C) \). If we set \( t = 0.8 \), then the algorithm will not change the reference sequence at all. Please note that the rewrite candidate \( (\text{del}, 6, \_\_\_) \) will never be used during the execution of the algorithm, independent from the threshold, since \( (\text{del}, 6, \_\_\_) \) is dominated by \( (\text{repl}, 6, C) \) for position 6.

It can be seen from Example 5, that the choice of threshold \( t \) has a great impact on the outcome of the rewriting algorithm: too large thresholds will ignore even relatively frequent rewrite candidates, which are shared by many referential compressions. Therefore, we analyse the effectiveness of reference rewriting depending on the threshold \( t \) in Section 3.6.

The complexity for computing rewritings is linear in the number of sequences and length of the sequences. The algorithm has to look at each consecutive pair of RMEs and check, whether it is a rewrite candidate for position \( p \). If yes, then we add an entry annotating position \( p \) in the reference sequence. In the end, we look at each position of the reference, select the most frequent rewriting candidate associated to that position, and rewrite the reference in case the candidates frequency is above threshold \( t \). Thus, the analysis of all sequences takes linear time and the actual rewriting can be done in linear time as well. It is an interesting direction for future work to investigate the rewriting of longer strings, i.e. to identify frequent indels with respect to the reference.

Note that in order to compute the referential compressions against the rewritten reference, we recompress all sequences from the scratch. Given fast compression times of FRESCO, we think that in most cases a recompression is tolerable. However, for frequently changing sequence sets, one should avoid recompression.

### 3.5 Second-Order Compression

An important part of each compression algorithm is the serialization of matches in the reference. Naive approaches can easily deteriorate any benefits of referential compression. One strategy for decreasing the size of serializations is to apply delta-encoding [43]. Our experiments indicate that this modification alone can often increase compression ratios by a factor of 2-4. We also compressed referentially compressed files with gzip, but delta-encoding alone can already outperform gzip significantly. We think that gzip fails to identify the different elements (position, size, mismatch) in referential match entries and therefore the compression ratio is not as high as with delta-encoding.

In the following, we present a new method for increasing the compression ratio of referentially compressed sequences. Our idea is to take referentially compressed sequences as input
for a simplified referential compression algorithm: now the alphabet is not \{A, C, G, T, N\} any more, but each referential match entry is a symbol of the alphabet.

**Example 6** Given the following four referential compressions:

\begin{align*}
rc_1 &= [(0, 4, T), (5, 3, A), (9, 4, T), (15, 3, G)] \\
rc_2 &= [(0, 4, A), (5, 3, A), (9, 4, T), (15, 3, G)] \\
rc_3 &= [(0, 4, T), (5, 3, G), (9, 4, T), (15, 3, G)] \\
rc_4 &= [(0, 4, T), (5, 3, A), (8, 3, T), (15, 3, G)]
\end{align*}

we can view \(rc_1\) as a reference and denote the other three sequences with a mix of standard referential match entries and new entries encoding second-order matches:

\begin{align*}
rc_2 &= [(0, 4, A), (1, 3)^{\scriptscriptstyle rc_1}] \\
rc_3 &= [(0, 4, T), (5, 3, G), (2, 3)^{\scriptscriptstyle rc_1}] \\
rc_4 &= [(0, 2)^{\scriptscriptstyle rc_1}, (8, 3, T), (15, 3, G)],
\end{align*}

where \((p, l)^{\scriptscriptstyle rc_i}\) denotes that the referential match entries \(p\) to \(p + l - 1\) are taken from compressed sequence \(rc_i\).

Our evaluation will show that our method can boost the compression ratio impressively. An informal description of our second-order compression algorithm is shown in Algorithm 5. It is very challenging to find an index structure for sets of compressed sequences for implementation of Algorithm 5. The problem is the sheer size of the alphabet. Most suffix-tree implementations we are aware of can only handle \(2^8\) symbols or \(2^{16}\) symbols at most. The number of unique referential match entries is in worst case quadratic in the length of the sequence and thus, for biological datasets there exists no practical, fixed bound.

Therefore we have implemented our own data structure for looking up prefixes of suffixes in referentially compressed sequences: for each referential match entry we store a hash value. The idea is very similar to a q-gram based index for \(q=1\) (Note that a small \(q\) is sufficient in practice because of the alphabet size). For each compressed sequence we store its RMEs using double-hashing with a fill-degree of roughly 75 percent. Using double-hashing, we can look up a given RME from one compressed sequence in another compressed sequence in constant time. In order to find the matches between two compressed sequences we iterate over all RMEs from one sequence and try to find these seeds in the second sequence. Once such a seed is found we try to extend matches to the right until we find different RMEs in both sequences. The
match between two sequences is then encoded as a second-order entry. Afterwards, the search is continued right of the previously checked RME in the first sequence. In our implementation, the complexity of finding (a subset of) all matching substrings between two sequences is quadratic in the length of the compressed sequences. We think that using a similar idea as in the KMP-string matching algorithm (prefix analysis), the run time complexity can be reduced to linear time.

We evaluate our second-order referential compression algorithms for different dataset and different numbers of reference sequences in Section 3.6. Additional sequences were selected randomly. Our experiments indicate that selecting a particular candidate set will likely not improve compression ratio, compared to random selection (using the same number of additional sequences).

### 3.6 Evaluation

In the following section, we evaluate our proposed compression scheme. All experiments have been run on a Acer Aspire 5950G with 16 GB RAM and Intel Core i7-2670QM, on Fedora 16 (64-Bit, Linux kernel 3.1). All size measures are in byte, e.g. 1 MB means 1,000,000 bytes. Below, the term compression factor is used to denote the inverse of compression ratio, e.g. a compression factor of 100 means a compression ratio of 100:1.

We have evaluated our algorithms for referential compression, reference selection/rewriting, and second-order compression on three biological datasets: a collection of human genomes, a collection of genomes from Arabidopsis thaliana, and a collection of yeast genomes. We have chosen these species since there sequences have different degrees of inner-species similarity caused by levels of repeats and variations. While human genomes are highly-similar to each other, yeast genomes often only have a small degree of similarity. Two Arabidopsis thaliana genomes are considered similar to a degree in between humans and yeast. Therefore, our three datasets cover a whole range of different similarities.

Our first dataset of human genomes was created from 1,092 genomes of the 1000 Genome project[1]. The 1000 Genome project group provides all sequenced genomes in Variant Call Format (VCF)[28] for download. The Variant Call Format describes differences of genomes with respect to a reference sequence, based on SNPs and indels. We have extracted one consensus sequence each for a total of 1,092 genomes. We use H-# to represent the set of all 1,092

### Figure 3.2: Standard compression algorithms for five sequences.
sequences for human Chromosome #, e.g. H-1 for human Chromosome 1. Grouping by chromosome makes sense, since usually sequences from the same chromosome have much higher similarities than sequences from different chromosomes. The union of all 23 human datasets (H-1 to H-22, H-X) is denoted with H-*. The largest human dataset is H-1 at 272.1 GB, the smallest dataset is H-22 at 55.9 GB, and the size of H-* is 3.3 TB.

Our datasets for Arabidopsis thaliana are taken from the 1001 Genomes project[15] from release GMINordborg2010.4. For each strain, a file with SNPs with respect to the reference TAIR9 is provided. We have extracted 180 genomes for each of the 5 chromosomes. The Arabidopsis thaliana datasets are prefixed with AT, e.g. AT-1 stands for 180 Chromosome 1 sequences of Arabidopsis thaliana. The union of all 5 Arabidopsis thaliana datasets is denoted with AT-*. The largest Arabidopsis thaliana dataset is AT-1 at 5.4 GB, the smallest dataset is AT-4 at 3.3 GB, and the size of AT-* is 21.4 GB.

The last dataset is a collection of yeast genomes [83]. In total, we have downloaded 38 yeast strains, each of them was provided in FASTA format. The yeast dataset is denoted with Y-WG. The size of Y-WG is 0.4 GB.

3.6.1 Existing Standard Compression Algorithms

We used three standard compression programs with default parameters to create initial statistics about self-referential compression: gzip, bzip2, and zip. For each species and each chromosome, we randomly selected five sequences and applied each of the compression algorithms. The results are shown in Figure 3.2. bzip2 is the best compression program among the three tested programs. The best average compression ratio is obtained by bzip2 for all three species and bzip2 is the fastest compression program as well, outperforming the other two programs by a factor of two on average. Using bzip2, it should be possible to compress H-* down to 0.7 TB, but the run time is expected to be around 126 hours. AT-* can be compressed down to 5.6 GB in 48 minutes. The compression factor is relatively stable within species for H-*(min: 3.91 for H-3, max: 5.82 for H-22) and AT-*(min: 3.74 for AT-2, max: 3.80 for AT-1). For Y-WG there is only one type of sequence (the whole genome).

3.6.2 Referential Compression Algorithms

We compare existing implementations of referential compression algorithms with FRESCO. The two competitors of FRESCO are GDC [31] and RLZ [70]. RLZ can be seen as one of the pioneers in referential compression, while GDC is the best existing program, when it comes to compression speed and compression ratio.

Our initial comparison is as follows: for each species and each chromosome, we randomly selected ten sequences and applied each of the referential compression algorithms. Please note that GDC applies a kind of reference preselection for a set of input sequences. The time spent on reference selection is not included in our measurements: we have measured solely compression time. RLZ uses suffix arrays for the reference sequence. The time of building the suffix arrays is not included in our measurements (building the suffix array for the reference of HG-1 took around 2 minutes). FRESCO uses a $k$-mer index (with $k = 34$) for the reference sequence and options LO_MD and COMPACT (local matching together with binary encoding of RMEs, see Section 3.7). The choice of $k$ has a big impact on compression speed, but almost no impact on compression ratio. With a value of $k$ smaller than 14, the compression is recognizably

---

4http://1001genomes.org/data/GMI/GMINordborg2010/releases/current/
slower, because FRESCO has to check a lot of spurious matches, which are not relevant for referential compression, because they do not yield long matches. For values of $k$ between 14 and 34 compression speed significantly increased (by a factor of 2-3), while compression ratio did not change recognizably. Increasing the value of $k$ beyond 34 did not change the speed recognizably. The time for constructing the $k$-mer index for each reference sequence is around 1 minute for the largest sequence and not included in the measurements. The results for compressing ten sequences each are shown in Figure 3.3.

GDC achieves the best compression for each dataset in our evaluation (on average 2.0 MB for ten sequences). We guess that this is due to sophisticated encoding techniques for the serialization format and the reference selection mechanism. GDC also tries to find and encode approximate matches into the reference. This idea seems to work well for highly different species. FRESCO achieves the second best compression (on average 2.3 MB for 10 sequences), while RLZ needs most space for each dataset (more than 5 times as much as GDC). The low compression factor of RLZ for Y-WG is likely due to limited optimization techniques in RLZ (especially for short machtes). The average compression factors for H-* are: GDC=635, RLZ=158, and FRESCO=551. The compression factors for AT-* and Y-WG are considerably lower due to decreased similarity among sequences in the collections.

FRESCO has the shortest compression times (on average 8.6 seconds for 10 sequences), while RLZ is around 10 times and GDC around 16 times slower. The compression speeds for H-* are as follows: GDC=11.2 MB/s, RLZ=12.8 MB/s, FRESCO=126.8 MB/s. The average compression speed of GDC for all species is 18.0 MB/s. It seems that GDC is highly optimized for compression of short sequences (or in particular Yeast species): the compression speed of GDC for AT-* and Y-WG is almost 5 times higher than for H-*.

We think that FRESCO is much faster than GDC for three reasons. First, GDC tries to extend the reference sequence with additional small reference parts during compression, while basic FRESCO uses a fixed refer-
D2.2 – Reference-based Compression Algorithms

Figure 3.4: Storage requirements for all against all for H-22.

ence for initial compression. Keeping additional index structures (or update them on the fly) is expensive. Second, GDC encoded approximate matches. While this allows for higher compression rates than basic FRESCO, it seems to be more computationally expensive to identify these matches with small errors. Third, we use a fast k-mer index which uses more memory than GDC, but allows for faster lookups.

For RLZ the average compression speed is at 11.5 MB/s, and for FRESCO the compression speed is roughly constant among all species as well: 128.0 MB/s. Both, RLZ and FRESCO, are slightly slower for Y-WG than for the other species. It can be seen that all three programs have a stable compression speed (with the exception of GDC, which is probably related to the species, and not to the length of the sequence).

We have run experiments with GReEn [97] and a 10-sequence sample of H-1. GReEn needs 183 seconds pure compression time for all 10 sequences (without creating the index structure for the reference). This is almost 10 times slower than FRESCO. The compression ratio is around 250:1. FRESCO-basic (590:1) and GDC (680:1) obtain at least doubled compression ratios. After all, the compression results of GReEn are very similar to those obtained by RLZ.

Note that the maximum read speed of the hard disk in our evaluation was measured at around 145 MB/s. Compression with FRESCO seems to be I/O-bound: we performed additional experiments with sequences in main memory. For H-*, we obtained an average compression speed of 729 MB/s and a maximum compression speed of 1 GB/s with FRESCO. This is up to two orders of magnitudes more than existing compression schemes. Even state-of-the-art SSDs often do not provide such a high throughput. For the other two species, the main memory compression speed is not recognizably higher than from an external hard disk. In our tests (data not shown), referentially compressed files can be decompressed at around 500 MB/s to main memory.

The main memory usage for FRESCO is around 8-10 times the size of the reference sequence, for representing the k-mer index in main memory. In our experiments with compressed suffix trees, the main memory consumption can be reduced down to 2 times the size of the reference plus the size of the to-be-compressed sequence, while compression times are increased slightly (plus 30 percent for H-*).

It is interesting to note that the ranking between the three programs is indeed consistent not only for different chromosomes but even for different species with respect to the two evaluation criteria. In summary, GDC always achieves the best compression, while FRESCO is one order of magnitude faster than RLZ and GDC.

3.6.3 Reference Selection in FRESCO

In order to show the impact of the reference sequence on the compression ratio, we used each chromosome in H-22 as a reference sequence, and referentially compressed all 1,092 Chromo-
In Figure 3.5 and Figure 3.6 we compare the compression factors and compression speed for different reference selection heuristics. RSbit1 and RSbit5 always achieve higher compression factors than RSrand. RSBit1/RSBit5 increases the compression factor by around 10 percent on average for our human genome dataset, compared to a random selection strategy. Base reference and candidate references were selected randomly; all results were averaged. In our experiments we compressed all 1,092 sequences. RSrand is the fastest selection heuristic, followed by RSbit5. For most of the sequences RSbit1 and RSbit5 yields similar compression factors to each other. In the following experiments we have used RSbit1 as a selection heuristic.

In Figure 3.7 we show the results of reference selection with respect to the base reference used before (in Figure 3.3). This time we have used the complete datasets for evaluation, e.g. all 1,092 genomes in H-*. The total run time includes the following steps: initial referential compression against base reference, selecting the best reference with RSbit1 with respect to compressed sequences, and recompression of all sequences against the chosen reference. The effect of reference selection is different for each species. While for H-* the average increase of compression factor is 5.8 percent, it is 2.7 percent for A-*. Selecting a best reference for Y-* did not have any effect on the compression ratio, since these genomes share only few similarities.

Compression is clearly slower when using reference selection (including basic initial referential compression): on average we obtain 74.8 MB/s. However, this is still 4-5 times higher
than for GDC and RLZ, and we obtain almost the same compression ratio as GDC.

### 3.6.4 Reference Rewriting in FRESCO

We analyse the impact of reference rewriting in Figure 3.8 with rewriting threshold as a parameter. The figure shows the impact of the threshold value on the storage requirements for 1,092 Chromosome 19 sequences with two randomly chosen base references. With a threshold value of 47 percent, the necessary storage is reduced to a minimum. In other experiments with human chromosomes (data not shown) values of 47-49 percent always yielded the minimum storage as well. The value of the threshold did not have a measurable effect on compression speed.

In Figure 3.9 we show the results of reference rewriting with respect to the base reference

![Figure 3.7: Selecting references in FRESCO.](image)

![Figure 3.8: Finding a threshold for rewriting sequences.](image)
<table>
<thead>
<tr>
<th>Dataset</th>
<th>C. factor</th>
<th>Total time (s)</th>
<th>C. speed (MB/s)</th>
<th>C. factor increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>H-1</td>
<td>804.3</td>
<td>3,334.8</td>
<td>81.6</td>
<td>+35.4%</td>
</tr>
<tr>
<td>H-2</td>
<td>736.4</td>
<td>3,033.3</td>
<td>87.5</td>
<td>+34.2%</td>
</tr>
<tr>
<td>H-3</td>
<td>697.6</td>
<td>2,520.7</td>
<td>85.6</td>
<td>+33.0%</td>
</tr>
<tr>
<td>H-4</td>
<td>653.0</td>
<td>2,340.8</td>
<td>89.1</td>
<td>+36.3%</td>
</tr>
<tr>
<td>H-5</td>
<td>704.9</td>
<td>2,138.6</td>
<td>92.3</td>
<td>+29.1%</td>
</tr>
<tr>
<td>H-6</td>
<td>643.7</td>
<td>2,311.6</td>
<td>80.8</td>
<td>+29.6%</td>
</tr>
<tr>
<td>H-7</td>
<td>675.1</td>
<td>1,994.4</td>
<td>87.1</td>
<td>+30.7%</td>
</tr>
<tr>
<td>H-8</td>
<td>674.3</td>
<td>1,737.2</td>
<td>92.0</td>
<td>+31.4%</td>
</tr>
<tr>
<td>H-9</td>
<td>834.1</td>
<td>1,612.7</td>
<td>95.5</td>
<td>+32.8%</td>
</tr>
<tr>
<td>H-10</td>
<td>676.1</td>
<td>1,655.1</td>
<td>89.4</td>
<td>+33.5%</td>
</tr>
<tr>
<td>H-11</td>
<td>673.7</td>
<td>1,659.9</td>
<td>88.8</td>
<td>+36.9%</td>
</tr>
<tr>
<td>H-12</td>
<td>698.2</td>
<td>1,586.9</td>
<td>92.1</td>
<td>+34.0%</td>
</tr>
<tr>
<td>H-13</td>
<td>765.9</td>
<td>1,350.8</td>
<td>93.0</td>
<td>+39.5%</td>
</tr>
<tr>
<td>H-14</td>
<td>806.1</td>
<td>1,266.1</td>
<td>92.5</td>
<td>+33.2%</td>
</tr>
<tr>
<td>H-15</td>
<td>864.1</td>
<td>1,190.6</td>
<td>94.0</td>
<td>+33.8%</td>
</tr>
<tr>
<td>H-16</td>
<td>753.6</td>
<td>1,024.3</td>
<td>96.2</td>
<td>+33.1%</td>
</tr>
<tr>
<td>H-17</td>
<td>729.8</td>
<td>1,030.2</td>
<td>86.0</td>
<td>+29.3%</td>
</tr>
<tr>
<td>H-18</td>
<td>671.2</td>
<td>946.6</td>
<td>90.0</td>
<td>+35.9%</td>
</tr>
<tr>
<td>H-19</td>
<td>619.8</td>
<td>846.5</td>
<td>76.2</td>
<td>+25.5%</td>
</tr>
<tr>
<td>H-20</td>
<td>703.1</td>
<td>670.3</td>
<td>102.6</td>
<td>+27.5%</td>
</tr>
<tr>
<td>H-21</td>
<td>769.0</td>
<td>508.2</td>
<td>103.4</td>
<td>+30.8%</td>
</tr>
<tr>
<td>H-22</td>
<td>904.5</td>
<td>548.3</td>
<td>102.0</td>
<td>+26.8%</td>
</tr>
<tr>
<td>H-X</td>
<td>1,018.0</td>
<td>1,993.8</td>
<td>85.0</td>
<td>+28.8%</td>
</tr>
<tr>
<td>AT-1</td>
<td>132.7</td>
<td>104.7</td>
<td>52.3</td>
<td>0.0%</td>
</tr>
<tr>
<td>AT-2</td>
<td>119.9</td>
<td>56.6</td>
<td>62.6</td>
<td>0.0%</td>
</tr>
<tr>
<td>AT-3</td>
<td>120.9</td>
<td>65.8</td>
<td>64.2</td>
<td>+0.1%</td>
</tr>
<tr>
<td>AT-4</td>
<td>119.0</td>
<td>56.1</td>
<td>59.6</td>
<td>0.0%</td>
</tr>
<tr>
<td>AT-5</td>
<td>125.5</td>
<td>75.8</td>
<td>64.1</td>
<td>0.0%</td>
</tr>
<tr>
<td>Y-WG</td>
<td>91.9</td>
<td>22.0</td>
<td>21.5</td>
<td>0.0%</td>
</tr>
<tr>
<td>AVG</td>
<td>613.3</td>
<td>1,299.4</td>
<td>83.0</td>
<td>+25.6%</td>
</tr>
</tbody>
</table>

Figure 3.9: Rewriting references in FRESCO.

Figure 3.10: Compression factors for randomly selected sequences with respect to 5-70 additional references

used before (in Figure 3.3). We have used complete datasets for evaluation, e.g. all 1,092 genomes in H-*. The total run time includes the following steps: initial referential compression against base reference, rewriting the reference with respect to compressed sequences, and re-compression of all sequences against the rewritten reference. The average compression factor is increased by roughly 25 percent for all datasets. Reference rewriting works clearly better for H-* (average increase 33.2 percent) than for AT-* and Y-*. This is again caused by high level of similarities among human genomes, where rewriting even a single SNP, can largely increase the match length. For large parts of the compressed sequences of AT-* and Y-* our algorithm cannot find many reference matches with more than 3-4 symbols. The compression factor after rewriting is clearly better than the compression factor for GDC (613.3 vs. 532.9). During reference rewriting we measured main memory usage of 14-16 times the size of the reference
D2.2 – Reference-based Compression Algorithms

<table>
<thead>
<tr>
<th>Wikipedia article</th>
<th>Versions</th>
<th>Current (KB)</th>
<th>Total (KB)</th>
<th>Time (ms)</th>
<th>Size (KB)</th>
<th>CF</th>
<th>Time (ms)</th>
<th>Size (KB)</th>
<th>CF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helsinki</td>
<td>2,664</td>
<td>406.8</td>
<td>584,178.0</td>
<td>16,419.1</td>
<td>115,970.0</td>
<td>5.0</td>
<td>17,375.1</td>
<td>40,708.8</td>
<td>14.4</td>
</tr>
<tr>
<td>Kardashev scale</td>
<td>1,808</td>
<td>179.7</td>
<td>188,359.0</td>
<td>8,410.9</td>
<td>49,521.2</td>
<td>3.8</td>
<td>8,260.7</td>
<td>23,517.8</td>
<td>8.0</td>
</tr>
<tr>
<td>Fairy_chess_piece</td>
<td>495</td>
<td>217.7</td>
<td>84,015.5</td>
<td>2,171.1</td>
<td>10,954.5</td>
<td>7.7</td>
<td>1,836.9</td>
<td>6,020.3</td>
<td>10.9</td>
</tr>
<tr>
<td>United_States_Numbered_Highways</td>
<td>576</td>
<td>154.5</td>
<td>66,396.7</td>
<td>2,488.4</td>
<td>15,151.4</td>
<td>4.4</td>
<td>2,520.1</td>
<td>6,705.8</td>
<td>9.9</td>
</tr>
<tr>
<td>Vela_Incident</td>
<td>790</td>
<td>164.7</td>
<td>65,542.7</td>
<td>2,696.5</td>
<td>17,629.9</td>
<td>3.7</td>
<td>1,805.3</td>
<td>6,020.3</td>
<td>10.9</td>
</tr>
<tr>
<td>Fan_death</td>
<td>1,099</td>
<td>60.4</td>
<td>60,137.3</td>
<td>2,730.9</td>
<td>18,927.4</td>
<td>3.2</td>
<td>2,165.3</td>
<td>7,593.9</td>
<td>7.9</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>1,238.7</td>
<td>197.3</td>
<td>174,771.5</td>
<td>6,152.8</td>
<td>38,025.7</td>
<td>4.6</td>
<td>5,660.6</td>
<td>14,554.6</td>
<td>13.5</td>
</tr>
</tbody>
</table>

Figure 3.11: Zip and referential compression on all versions of different Wikipedia articles.

sequence. This is caused by management of the rewrite candidates.

In total we need around 10 hours to compress all datasets, starting from raw sequences. This yields an overall compression speed of 88.5 MB/s, which is around 4 times higher than for GDC (18.0 MB/s). If we only look at H-*, than the improvement is the difference in compression speed is even bigger between reference rewriting in FRESCO (88.8 MB/s) and GDC (11.2 MB/s). Please note that GDC performs some kind of hash-based preselection of a reference, whose time was not taken into account. Otherwise the average compression speed of GDC would be reduced to around 5-6 MB/s. We have also run experiments with GDC and our rewritten reference sequence, and the compression ratio did not improve.

3.6.5 Second-Order Compression in FRESCO

In the following, we evaluate second-order compression for H-1, H-22, and AT-1 with a different number of additional compressed references (we obtained similar results for the other datasets). The base reference for each dataset is obtained by rewriting a fixed reference with a threshold of 47 percent. We did not evaluate second-order compression for Y-*, because the number of sequences (38) is too small.

In Figure 3.10, the compression factors for 5-70 additional references are shown. Please note that we have only compressed sequences which are not included in the set of additional references. It can be seen that for all three datasets the compression factor is around four times higher when having 70 additional compressed sequences as reference, leading up to 4000:1. Already with 10 additional references, the compression factor can be almost doubled, leading to an average compression ratio of 1500:1 for H-1 and H-22, and 227:1 for AT-1.

Compared to reference selection/rewriting, second-order compression increases the compression ratio recognizably even for AT-* and Y-*. We think that this is caused by the following: AT-* and Y-* contain many clusters of similar individuals. No matter which sequence we pick as a reference for referential compression, we only can compress sequences from the same cluster more efficiently. On the other hand, second-order compression compresses sequences against multiple references. In this case (even with a random set of references) sequences from different clusters are used as references. In the human dataset we found two major clusters only (see Figure 3.4 for distribution of storage requirements). A reference sequence from one of the two clusters can still be optimized for all the (many) sequences inside the cluster. For 10 additional references, the overhead of second-order compression is small: the compression time is increased by 20-40 percent. For 70 sequences, compression time is already almost doubled for all three datasets. Main memory usage depends on the number of additional references and the (average) number of referential match entries per string. For H-22 we measured around 320 MB plus roughly 2 MB for each additional reference.
### 3.6.6 FRESCO for Compressing Wikipedia Articles

Over time, a WikiPedia article undergoes several modifications by different users. Often, these modifications only address small parts of the documents. We have tested FRESCO on a randomly chosen collection of 100 Wikipedia articles. On average, FRESCO compresses an article with all its versions by a compression factor of 12.3, while zip obtains a compression factor of 3.4 only. FRESCO is usually faster than zip: in average zip needs 0.7 seconds to compress all versions of an article, while FRESCO needs 0.5 seconds. The results for the six most modified articles is shown in Figure 3.11. We have also compared FRESCO with gzip and bzip2. On average (over all our Wikipedia articles), FRESCO is around 15 percent faster compared to gzip. The execution of bzip2 took around 5 times longer compared to FRESCO, while FRESCO still shows a better compression ratio on average. We ran gzip and bzip2 with default options.

### 3.7 FRESCO: Open Source Release

FRESCO, Framework for REferential Sequence COmpression, is the name of our open source release. The software can be found at [https://github.com/hubsw/FRESCO.git](https://github.com/hubsw/FRESCO.git). FRESCO was implemented in C++, using the BOOST library, CST[91], and libz. We have designed FRESCO in a modular way, which makes it easy to replace parts of the compression algorithm, e.g. index structures, with different implementations. The major design choices when implementing a referential compression algorithm are 1) input format, 2) index structure for the reference, 3) compression algorithm, e.g. greedy, and 4) serialization format for compressed files, i.e. the actual encoding of matches. For existing compression algorithms, developers make a choice for either of these criteria at design time. FRESCO contains interfaces for each of these four components and allows to use different implementations interchangeably and to add novel, possible specialized algorithms. In the following, we describe each of these interfaces and their standard implementations in FRESCO in detail.

The sequence interface defines two functions: one for loading a sequence from a file and another one for writing a sequence to a file. FRESCO provides implementations for handling raw-files (one byte per symbol) and FASTA files.

An index is used for looking up matches of the to-be-compressed sequence with respect to the reference. The index is initialized from a given reference sequence, e.g. loaded from a FASTA file. The interface declares a function for looking up the longest prefix match of an input string with respect to the indexed reference. In FRESCO, we provide a standard implementation based on a k-mer hash index, i.e. for each k-mer we store all occurrences in the reference sequence. Once a match for a partial sequence is needed, the k-mer prefix of the partial sequence is used to find the longest match in the reference sequence. Other implementations could use suffix arrays as in [70].

---

<table>
<thead>
<tr>
<th>NCDC</th>
<th>RLZ</th>
<th>FRESCO (reference selection)</th>
<th>FRESCO (reference rewriting)</th>
<th>FRESCO (second-order compression)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF</td>
<td>C.Speed</td>
<td>CF</td>
<td>C.Speed</td>
<td>CF</td>
</tr>
<tr>
<td>H-*</td>
<td>635.0</td>
<td>11.2</td>
<td>158.4</td>
<td>12.8</td>
</tr>
<tr>
<td>AT-*</td>
<td>144.6</td>
<td>43.9</td>
<td>99.3</td>
<td>7.5</td>
</tr>
<tr>
<td>Y-WG</td>
<td>127.3</td>
<td>44.5</td>
<td>1.4</td>
<td>2.6</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>302.3</td>
<td>33.2</td>
<td>86.4</td>
<td>7.6</td>
</tr>
</tbody>
</table>

Figure 3.12: Summary of all techniques (CF=compression factor, C.speed=compression speed in MB/s)
The compression interface defines two functions: one for compressing a sequence into a list of referential match entries and another one for decompressing referential match entries back to a sequence. FRESCO provides three compression algorithms: 1) a greedy (BAS), which always finds the longest possible match, 2) an optimization for finding local matches without expensive index lookups (LO), and 3) an optimization which prefers short, but local matches over longer matches further away from the previous match (LO_MD), a strategy proposed in [31].

A serializer (un)serializes a list of referential matches to/from a file. FRESCO has three standard implementations: 1) plain ASCII format (PLAIN), 2) plain encoding with positions relatively encoded to previous matches (DELTA) [31], and 3) compact binary encoding (COMPACT).

3.8 Conclusions

In Figure 3.12, we show an overview of the main results obtained in our evaluation for biological sequences. It can be seen that all variants of FRESCO outperform existing referential compression algorithms in terms of compression speed. Furthermore, the compression factor for most variants is similar to related work, while the best variant of FRESCO obtains a compression factor 4-5 times higher. Apart from the greedy compression algorithm, the other components of FRESCO are optional. Moreover, it does not always make sense to apply all steps in a row. In particular, the results obtained from greedy referential compression together with second-order compression yields results very similar to those obtained with additional reference rewriting in between.

Our results show that second-order compression on top of greedy compression and reference rewriting, boosts compression ratios far beyond the state-of-the-art. The larger and more similar the set of to-be-compressed sequences is, the more it makes sense to apply second-order compression. In our tests (data not shown), second-order referentially compressed files can be decompressed at around 500 MB/s in main memory, similar to normal referentially compressed files.

We conclude that lossless referential compression of highly-similar sequences referentially can be done in real-time on commodity hardware. Based on our results, it should be investigated whether working on compressed files is feasible, first results are encouraging [122].
Chapter 4

On-Demand Indexing for Referential Compression

Chapter Authors:
Fernando Alves, Vinicius Cogo and Alysson Bessani (FFCUL).

4.1 Introduction

A genome is the complete set of DNA from an organism, and contains all information needed to build it and maintain it alive. In humans, the entire genome is comprised of more than 3 billion DNA base pairs (bp), which are enclosed in all cells containing a nucleus. All these nucleic acids are expressed in more than 3GB of information in a text-based file (one byte per bp).

The cost of genome sequencing has decreased exponentially in the last few years [103, 128], and the number of sequenced and stored genomes is increasing in a similar pace [55, 131]. The impressive mark of $1,000 for the sequencing of a whole human genome was achieved in the beginning of 2014 by the Illumina Inc. with the HiSeq X Ten platform [48, 54]. Sequencing cost is expected to continue falling in the next years, until it reaches the $100 mark, and eventually the $30 mark per human genome. Such trend directly imply in an increase in the amount of obtained data and in the complexity of data management.

In computer science, compression is a process for reducing the size of data sets for efficient data transfer and storage. Compressing genomes is an important step in many bioinformatics workflows since it attenuates the evolutive gap between sequencing and storage technologies. However, traditional compression algorithms are inefficient when working with genomic sequences [44]. Developers of genome handling systems have created compression tools specifically for human genomes, but typically with high latency and resource costs [120]. Compressing genomes is imperative for biobanks to continue increasing their genome collections within their local infrastructures. Lossless compression is the preferable form of compression for biobanks since it reduces the required disk space for genomes, but still stores all information for future usage. Lossy algorithms are more interesting for single purpose researches, where scientists know exactly which portions of data are useless and/or disposable.

As presented in Chapter 2, compression algorithms can be classified according to its approach, and most fit in one of the following groups [40, 120]: naive binary encoding, dictionary based methods, statistical methods and referential compression methods. Referential compression of genomes [12, 120] is an interesting approach for biobanks with human samples, since any two human beings are 99.5% genetically equal from each other. It achieves high compres-
sion ratios, but data has to be previously aligned to the reference genome. It is the starting point of this chapter, since it obtains the best results in compression ratio and execution time. Since this is a recent approach, there is still room for improvement.

This chapter proposes the JDNA tool for the BioBankCloud project. It is a low resource genome referential compression tool for aligned sequences. The JDNA algorithm is based on the algorithm used by FRESCO \[123\], the tool for compressing genomes presented in Chapter 3. We present a new approach to referential compression for very similar files, where the usage of an index structure is reduced to the strict minimum. Our approach shows an improvement in execution time of up to an order of magnitude.

The remainder of the chapter is organized as follows: we recapitulate some important aspects about referential compression in Section 4.2. JDNA is reviewed in detail in Section 4.3. In Section 4.4, we discuss the performance of JDNA and compare it to FRESCO, and conclude the chapter in Section 4.5.

### 4.2 Referential Compression

As described in Chapter 2, the main concept of referential compression is, given a genome to be compressed and a reference genome, generating an output file containing only the differences between the two input files \[12\]. Such approach obtain the best results when organisms are from the same species, which share a high percentage of genetic similarity (e.g., humans are 99.5% genetically similar). The higher the similarity, the higher the compression ratio.

Referential compression is one of the most efficient ways of compressing DNA files, since the current implementing algorithms can obtain ratios of up to 400:1 (four hundred to one), which means the size of the compressed data can be $400 \times$ smaller than the original. Non-referential approaches achieve ratios of up to 8:1 \[120\], which puts referential compression in the state-of-the-art techniques.

Amongst the referential compression tools, the RLZ \[70\], GDC \[31\] and FRESCO \[123\] (described in Chapter 3) are noteworthy. RLZ is one of the pioneers in referential compression and uses suffix arrays for reference indexing. GDC occupied the top for compression ratio and execution time, and uses a reference pre-selection for each genome set. FRESCO is the fastest referential compression tool by surpassing the other solutions by an order of magnitude. However, these FRESCO’s results are only for actual compression times, since they do not include the time for index structure generation. FRESCO uses a k-mer table, which is similar to a hash table with a set of values for each key.

#### 4.2.1 Genome Files

Genomes are large nucleotide sequences composed mainly by four base types: Adenine, Cytosine, Thymine and Guanine. So, the segment “ACC” corresponds to the genomic sequence Adenine-Cytosine-Cytosine. A raw genome file is composed only by a series of symbols correspondent to the nucleotide bases present in actual genomes, which are obtained from sequencing machines. Other base symbols are produced when the sequencing results are inaccurate, for example, the symbol “W” is written when there is uncertainty between the base pairs “A” and “T”. Alternatively, “N” is written when there is not enough information about a base pair to qualify it, thus represents any base symbol.

Libraries work with the base pairs present in genome files in different ways. For example, FRESCO stops its execution if a base pair different from “A”, “C”, “T”, “G” or “N” is found.
4.2.2 The FRESCO Library

FRESCO is a lossless referential compression library for aligned genome files, stored in RAW or FASTA formats. It is written in C++ and is open source [123]. FRESCO covers the following genetic variations: single nucleotide polymorphisms (SNPs), insertions, deletions and substitutions. It executes three internal steps (see Figure 4.1): (1) indexing, (2) compression itself and (3) codification.

Figure 4.1: Conceptual model of FRESCO’s execution. (1) Reference genome indexing. (2) Compression of the input genome, using the indexed reference. (3) Codification of the preliminary results to produce the final file.

(1) Indexing. FRESCO uses a data structure, called K-mer table [66], to index the complete reference genome. It is similar to a hash table, with the difference that it stores multiple values per key, as it is shown in Figure 4.2. The hash of each segment with size $K$ is calculated and the index where the segment was found in the reference is stored in the list indexed by this hash. When all segments of size $K$ from the reference are processed, the structure can be used to find matches between the reference and input files. The complete reference indexing provides a deterministic search property since one can be sure about the presence or absence of any segment of size $K$ in the reference.

(2) Compression. The compression phase uses the referred indexing structure and the genome to be compressed. A segment of size $K$ of the input is hashed, and the hash is looked up in the K-mer table. A successful lookup returns a set of indexes where the segment can be found in the reference. The index that produces the longest match is selected, then the match is extended deterministically. When the match ends, a new entry is created in the intermediate result. This entry is composed by the size of the match, where it begins and the different base pairs until the new match. After that, a new match is looked-up in the table, and this method is repeated until all input file is processed.

(3) Codification. Finally, the intermediate results are compressed through character encoding. FRESCO uses GZIP to compress the intermediate results, which produces the final result as output file.

Figure 4.3 presents an analysis of FRESCO’s execution time for different chromosomes of a genome. Each bar of this graph is composed by the three steps previously detailed. The
Figure 4.2: Indexing model using a K-mer table. The first column stores the hash associated with one or more indexes, and those indexes as stored in the second column.

“Others” element is composed by several small functions, for example, the file reading and writing times, codification, and other operations like variable initialization. This figure shows that most of FRESCO’s execution time is spent in the indexing phase. The other steps represent much smaller portions in comparison to indexing.

Figure 4.3: FRESCO’s execution time separated on its internal steps.

Due to this massive indexing present on FRESCO, we propose an optimization to reduce the total execution time. Note that FRESCO’s speed is much faster than its competitors, but we will later show that we can improve those times.
Finally, in the decompression process, the original genome is recovered from the compressed file and the same genome reference used during its compression. The compressed file is initially composed by pointers to the parts of reference that match the input file, and the base pairs corresponding to the differences between files. The matches and different base pairs are intercalated. For each portion composed by a match and a variation, we write the match and corresponding base pairs that form the differences between files. When writing the match, a sequence from the reference is written in the output, and subsequently, the different base pairs are written directly in the output. We do not focus our attention on the decompression process since it does not benefit from our optimization.

4.3 On-Demand Reference Indexing

As presented in the previous section, the indexing step is responsible for the biggest portion of FRESCO’s execution time. Figure 4.3 indicates that over 95% of the execution time is spent indexing, while only about 2% of the time is spent in the actual compression, and about 3% on the remaining execution steps. Two optimization opportunities arise from this observation: (1) employing parallelism, and (2) replacing the complete reference indexing by an on-demand indexing method. In the first solution, the creation of the index structure would be done in parallel through the usage of synchronization mechanisms while storing the hashes and indexes. However, it would be limited by the machine capacities, and perhaps one does not need the entire index structure. In the second case, instead of indexing fully the reference at the beginning of the program execution, we would employ simple heuristics for sequence comparison to find the matches. We opt for the second opportunity (which we call on-demand indexing), however both solutions are complementary. The detailed description of our method is described in the following sections.

We propose the on-demand indexing method and provide a Java tool, called JDNA, for referential compression using this method. The JDNA is designed to apply referential compression to genome files. There are four types of genomic variations supported by JDNA, namely: SNPs, substitutions, insertions and deletions.

- **SNP**: a single base pair difference in the same position from different genomes.
- **Substitution**: a sequence of base pairs that is different in the same region of two genomes.
- **Insertion**: a sequence of base pairs that is added to a genome’s region. Therefore, in the same region of another DNA that base pair sequence is not present.
- **Deletion**: a base pair sequence that is not present in a genome’s region but is present in the majority of genomes.

Apart from SNPs, which are always a difference of one base pair, the other genomic variations can vary in size. We define a value $\theta$ used to classify a variation as small or big. In this work, the value of $\theta$ is 300, where the variation segment size is small ($SD$) if it is smaller than $\theta$, and big ($BD$) if it is bigger. On-demand indexing stands on the fact that genomes from the same species have a high genetic similarity, thus, most variations are $SD$ and can be easily found by using simple comparison heuristics.
4.3.1 Compression Algorithm

Compression executes in file blocks or with the full file in memory. JDNA is designed to load files of up to 250MB to memory with bigger files loaded in 250MB blocks. Both reference and input files are loaded in this way. Each block is compressed separately, so there is no connection between compressed blocks even if they are from the same file. If block compression is used, only one compression file is generated, which contains the compressed blocks.

In the JDNA, matches are found in three steps:

1. We test deterministically if there is an SNP.
2. If it is not an SNP, then we do a local search of about $\Delta$ base pairs for a match - this covers most SDs; $\Delta$ is 6 times $K$, where $K$ is the K-mer size.
3. If no match is found we index a piece of the reference in the KmerTable and search the current position of the input in the table.

If after these steps still no match is found we write a base pair of the input and repeat the process. We find that it is better to index just a bit of the reference at a time because if there is a large insertion in the input file we could be indexing a lot of the reference for no use. If a match is found (of any type) then we search deterministically how long that match is to increase compression, and then write the match.

The input file is processed searching for matches in the reference until all input has been compared.

The main advantage of this algorithm structure is the usage of on-demand indexing. Since genomes are 99.5% equal, there is no need for an index structure to find the majority of the matches; only when the most direct approaches fail (steps 1 and 2) there is the need to index.

Huffman Encoding. We use Huffman Encoding [129] when writing matches to increase compression by reducing the size of what is written to disk. For example, instead of writing an integer (32 bits) we write only a few bits that describe the integer. After using the said encoding the result is further compressed through GZIP.

4.3.2 KmerTable

We need an index structure for the compression. First JDNA's implementations used classic object implementations (like HashMap) to index. However, we observed that the memory overhead of using a large number of complex objects is huge. Declaring a HashMap with 50000 buckets and populating such structure made our memory usage increase beyond 20GB. We needed to keep memory usage at most to 2GB-2.5GB to make JDNA competitive with FRESCO in all aspects.

We use a hash-like table called KmerTable to store information relevant to the compression algorithm. This table is similar to FRESCO’s table illustrated in Figure 4.2. The KmerTable is similar to a normal MultiValueMap structure, it stores multiple values per key instead of just one value per key. The KmerTable is an integer matrix table and an integer array counter (whose purpose will be explained later). In Java, an integer matrix is a primary array of secondary arrays of integers. Each position in the primary array starts with null. Then, we simulate an ArrayList in each primary array position to insert values in table, by creating a fixed size array and using the counter to control the number of elements in each position of primary. The counter array is used to access the stored values and to know when to expand the arrays.
This effectively creates a *MultiValueMap* with minimal object creation. This implementation brings two benefits. One is the reduced memory usage while using an efficient data structure. The second is re-usability. The KmerTable can be easily re-used by resetting all values in `counter` to zero.

The `put` and `get` operations receive an index $I$ on the reference or input to make a hash $H$. $H$ is obtained by hashing a number of base pairs equal to $K$ ($K$ is the K-mer size) starting on the given index. Then $H$ is used as key to store $I$ or to get all the values present for the key $H$. Each key $H$ stores a set of indexes with hash $H$. All hashes are 32bit.

### 4.3.3 Decompression Algorithm

The decompression mechanism is very similar to FRESCO, but JDNA keeps only a small amount of the reference in memory. As the pointers advance in the reference we *slide* the vector, thus occupying a small and fixed amount of memory.

### 4.3.4 File formats

JDNA accepts two file types as input for compression: RAW or FASTA files. If the file is RAW then a CRAW (for compressed RAW) file is generated; if the file is a FASTA then a CRAW and a CCOM files are generated. The CCOM file has the comments present in the FASTA file and the line numbers of those comments. For decompression, a CRAW file is assigned as input. JDNA will then search for a CCOM file with the same name as the CRAW file. If the CCOM file is found then a FASTA file is generated, otherwise, a RAW file is created. This information is summarized in Figure 4.4.

![File formats for input and output of JDNA.](image-url)
4.3.5 Execution

JDNA is a command line tool, whose arguments are

```
[TASK] [REFERENCE GENOME] [INPUT FILE] [OUTPUT FILE]
```

where task is either COMPRESS or DECOMPRESS. An example for compression is

```
```

4.3.6 Java Virtual Machine Configuration

We tune the Java Virtual Machine (JVM) to increase the performance of JDNA [53]. Our tuning aims to avoid garbage collection during execution, and to reduce the re-execution if it is executed. We have no interest on garbage collection execution because most objects created by JDNA on compression are permanent, where even the parallel garbage collection reduces compression speed drastically. If a garbage collection process occurs, then the objects in use are transferred immediately to the permanent memory area of JVM so that in the next garbage collection these objects are not verified again. We employed the following JVM arguments to run JDNA when compressing chromosomes:

```
-Xms3000m -Xmx3000m -XX:+ScavengeBeforeFullGC -XX:+UseParallelOldGC
-XX:InitiatingHeapOccupancyPercent=80 -XX:InitialTenuringThreshold=20
-XX:NewRatio=1 -XX:SurvivorRatio=1
```

These should be the default options to run JDNA, and the results presented in Section 4.4 use them. No parameters are required for decompression. For entire genome compression, one should change `-Xms3000m` to `-Xms4000m` and `-Xmx3000m` to `-Xms4000m` and maintain the other parameters. For debugging purposes, one may add:

```
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -verbose:gc
```

4.4 Evaluation

In this section, we present results in terms of time, compression ratio and memory footprint, in order to compare the performance of JDNA and FRESCO. In our data set some chromosomes contained base pairs different from “A”, “C”, “T”, “G” and “N”, and for those chromosomes FRESCO provided no data, since it is not prepared to process these files. Each test was executed ten times and the present results are their average values.

4.4.1 Experimental environment

**Hardware.** All tests ran on a machine with the following specifications:

- CPU: 2x Intel Xeon E5520 2.27 GHz (8 Cores, 16 Threads) / 1 MB L2 cache / 8 MB L3 cache
- Memory: 32 GB (8x4GB) / DIMM Synchronous 1066 MHz (0.9 ns)
- Storage: 146 GB / SCSI / 15000 RPM / ST3146356SS; Driver: mptsas
Software. The machine runs an Ubuntu 10.04 64-bit GNU/Linux operating system with the kernel version 2.6.32-21-server. JDNA is developed in Java and tested using Oracle’s Java 7 JVM (build 1.7.0_40-b43). We use FRESCO version 0.2.

We compare JDNA only with FRESCO for two reasons. First, the design of the two tools is very similar. Second, FRESCO was recently shown faster than other similar tools (e.g., GDC [31] and RLZ [70]).

Data sets. We use the reference genome provided by 1000 Genomes Project. For input genomes we have randomly selected seven donors identifiers from 1000 Genomes Project, which compose our test input data. The donors were HG00339, HG01390, NA12546, NA19788, HG00173, HG00619, NA19449. These files were used in raw format, either separated by chromosomes or as a whole genome. The reference is always the same for all executions.

4.4.2 Compression ratio

A compression ratio of \((y:1)\) means that the size of a compressed file is \(y\times\) smaller than the original size. In Table 4.1, we compare the compression ratios between the two tools, and Figure 4.5 presents the achieved compression ratio. The compression ratio is almost identical between JDNA and FRESCO. There are two reasons for the small differences observed for some chromosomes. The first reason is the encoding algorithm employed by each solution. One bit difference in the coding phase causes considerable changes in the compressing ratio, since it is amplified by thousands of matches. JDNA uses a custom version of Huffman encoding and Gzip, while FRESCO only uses GZip to compress each match. The second reason is the complete indexing performed by FRESCO. The deterministic search guaranties that if a match exists it is found, unlike JDNA, where a match is found only by small deterministic searches and small indexing operations.

4.4.3 Time

In this experiment we compare the time efficiency of both tools for compressing and decompressing human genomes. We separate time evaluation in four components:

Full compression: This is the full execution of the library; includes the startup time, file reading, memory allocation, reference indexing, compression and file writing.

Time indexing: Since JDNA introduces on-demand indexing we compare only the indexing times.

Compression time: In this topic we evaluate the performance of the two solutions on compression itself, by measuring the time spent on actual compression.

Decompression time: Here we evaluate the decompression performance of both libraries, measuring the time for complete execution.

We measured both startup times, and JVM with the configurations for JDNA takes in average 0.1 seconds to startup, and FRESCO takes 0.04 seconds. Full compression time was

---

1We use as reference genome the one available from 1000 Genomes Project, on its version 37, which is available at http://ftp-trace.ncbi.nih.gov/1000genomes/ftp/technical/reference/humang1kv37.fasta.gz.
measured using the command line tool `time`, and the results can be seen in Figure 4.7. As previously described, the JDNA avoids indexing, which makes a massive difference in the compression time. These values can be observed in Table 4.1. Even large files (e.g., chromosome 1) take about 5 seconds to compress with this algorithm structure, which is a massive difference for the almost full minute FRESCO takes to compress the same files. JDNA compression time is almost constant (around 3s), dropping to about 2s for smaller chromosomes. These results are around $5 \times$ to $12 \times$ faster than what we observed with FRESCO. This difference is due to the on-demand indexing. Since we do not index the whole reference, JDNA does not have the indexing time component at the beginning of its execution.

The time each library spends indexing the reference genome is measured internally during program execution, and the results can be seen in Table 4.1. JDNA spends almost no time indexing, specially compared to FRESCO, where indexing time is always greater than the time JDNA spends on full compression for any chromosome. A small percentage of base pairs is indexed, as can be seen on Table 4.1 and on Figure 4.8. FRESCO always indexes 100% of the genome reference.

We measure the time spent compressing inside the program. The results are present in Figure 4.7. JDNA spends about as much time as FRESCO on the compression step, which shows that the major difference between the execution time of these tools is the indexing step.

Full decompression time was measured also using the command line tool `time`, and the results are present in Figure 4.9. JDNA decompression time is bigger than FRESCO’s in all chromosomes. This is due to the vector sliding described in Section 4.3.3.
4.4.4 Memory footprint

In this experiment we compare the memory footprint of both tools and analyse the obtained results.

Compression memory. We used an external tool to measure the peak of memory usage [108] of both tools. In Figure 4.10, we can observe the memory usage of JDNA and FRESCO. JDNA has implemented mechanisms that reuse objects and reduce object creation. However, JDNA and FRESCO have similar memory footprints, even after a considerable effort to reduce memory usage of our tool. The bulk of memory usage in JDNA is due to the K-mer Table. Even
though the JDNA indexing is reduced and the K-merTable is just an integer matrix, because each line of the matrix is a new object, the memory overhead is still large comparing to FRESCO, that indexes the complete reference.

**Decompression memory.** As described in Section 4.3.3, decompression uses a fixed amount of memory for reference, resulting in a constant memory usage, as shown in Figure 4.11.
<table>
<thead>
<tr>
<th>Chr</th>
<th>Original Size (MB)</th>
<th>Compression JDNA (KB)</th>
<th>Compression FRESCO (KB)</th>
<th>Index Time JDNA (s)</th>
<th>Index Time FRESCO (s)</th>
<th>Indexes K</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>238</td>
<td>306</td>
<td>306</td>
<td>0.02</td>
<td>62.56</td>
<td>40.48</td>
<td>1.6</td>
</tr>
<tr>
<td>2</td>
<td>232</td>
<td>319</td>
<td>318</td>
<td>0.01</td>
<td>57.6</td>
<td>33.55</td>
<td>1.4</td>
</tr>
<tr>
<td>3</td>
<td>189</td>
<td>269</td>
<td>406</td>
<td>0.01</td>
<td>47.02</td>
<td>34.99</td>
<td>1.8</td>
</tr>
<tr>
<td>4</td>
<td>182</td>
<td>296</td>
<td>299</td>
<td>0.03</td>
<td>45.5</td>
<td>63.79</td>
<td>3.3</td>
</tr>
<tr>
<td>5</td>
<td>173</td>
<td>234</td>
<td>235</td>
<td>0.02</td>
<td>42.58</td>
<td>38</td>
<td>2.1</td>
</tr>
<tr>
<td>6</td>
<td>163</td>
<td>243</td>
<td>249</td>
<td>0.05</td>
<td>40.03</td>
<td>150.23</td>
<td>8.8</td>
</tr>
<tr>
<td>7</td>
<td>152</td>
<td>215</td>
<td>217</td>
<td>0.02</td>
<td>37.2</td>
<td>41.96</td>
<td>2.6</td>
</tr>
<tr>
<td>8</td>
<td>140</td>
<td>202</td>
<td>202</td>
<td>0.01</td>
<td>33.89</td>
<td>26.87</td>
<td>1.8</td>
</tr>
<tr>
<td>9</td>
<td>135</td>
<td>161</td>
<td>162</td>
<td>0.03</td>
<td>28.97</td>
<td>94.12</td>
<td>6.7</td>
</tr>
<tr>
<td>10</td>
<td>129</td>
<td>192</td>
<td>194</td>
<td>0.01</td>
<td>30.94</td>
<td>15.5</td>
<td>1.1</td>
</tr>
<tr>
<td>11</td>
<td>129</td>
<td>204</td>
<td>204</td>
<td>0.02</td>
<td>30.92</td>
<td>30.89</td>
<td>2.3</td>
</tr>
<tr>
<td>12</td>
<td>128</td>
<td>186</td>
<td>186</td>
<td>0.02</td>
<td>30.65</td>
<td>34.81</td>
<td>2.6</td>
</tr>
<tr>
<td>13</td>
<td>110</td>
<td>157</td>
<td>159</td>
<td>0.01</td>
<td>23.3</td>
<td>15.55</td>
<td>1.4</td>
</tr>
<tr>
<td>14</td>
<td>102</td>
<td>126</td>
<td>128</td>
<td>0.01</td>
<td>21.44</td>
<td>19.96</td>
<td>1.9</td>
</tr>
<tr>
<td>15</td>
<td>98</td>
<td>115</td>
<td>115</td>
<td>0.01</td>
<td>19.91</td>
<td>10.13</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>86</td>
<td>113</td>
<td>112</td>
<td>0.01</td>
<td>18.63</td>
<td>15.91</td>
<td>1.8</td>
</tr>
<tr>
<td>17</td>
<td>77</td>
<td>100</td>
<td>100</td>
<td>0.01</td>
<td>18.01</td>
<td>16.13</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>74</td>
<td>114</td>
<td>115</td>
<td>0.01</td>
<td>17.56</td>
<td>7.74</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>56</td>
<td>79</td>
<td>80</td>
<td>0.01</td>
<td>12.81</td>
<td>26.87</td>
<td>4.5</td>
</tr>
<tr>
<td>20</td>
<td>60</td>
<td>79</td>
<td>79</td>
<td>0.01</td>
<td>14.06</td>
<td>22.16</td>
<td>3.5</td>
</tr>
<tr>
<td>21</td>
<td>46</td>
<td>58</td>
<td>61</td>
<td>0.01</td>
<td>8.93</td>
<td>25.11</td>
<td>5.2</td>
</tr>
<tr>
<td>22</td>
<td>49</td>
<td>49</td>
<td>50</td>
<td>0.01</td>
<td>8.89</td>
<td>24.28</td>
<td>4.7</td>
</tr>
<tr>
<td>X</td>
<td>148</td>
<td>109</td>
<td>109</td>
<td>0.01</td>
<td>35.93</td>
<td>18.23</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Table 4.1: JDNA/FRESCO comparison table.

Figure 4.10: Compression memory comparison.
4.4.5 Compressing entire genomes

We obtain a compression ratio of about 750:1 (≈3GB to ≈4MB) when compressing an entire genome separated by its chromosomes. The compression ratio drops to 650:1 (≈3GB to ≈4.5MB) when compressing the entire genome as a single file. Recall that these results vary, depending on the similarity between input and reference files. Additionally, the memory usage increases, needing about 4GB of memory to execute. It also takes more time to compress, from about 80 seconds, to about 4 minutes. The cause for this time increase has been located in the JVM memory management but a solution has not yet been found. Changing the block size or providing more memory to JVM changes the amount of time needed to compress an entire genome. The best result achieved was providing 30GB to JVM and setting the block size to 1.5GB (the maximum block size is the integer maximum value in Java - 2.1GB; since the file is about 3GB even sized blocks provide better memory usage). With these settings, the compression time is reduced to a range between 25s and 120s (depending on the genome). The compression time range is large because of the referred JVM memory management problems.

4.4.6 Discussion

The similarity between the reference and input genomes determines FRESCO and JDNA results, where the greater the similarity, the higher the compression ratio. The values we present as results are averages. The entire compression of a human genome results in a file with size from 4 to 10MB.

The results shown in the evaluation section prove that the on-demand indexing algorithm can be used to build a tool comparable to tools that completely indexing the reference. The results are competitive in all tested characteristics and show a great improvement in total running time. Note that the objective of this chapter is not to present a tool that runs faster than FRESCO (while being competitive on the other properties), the objective is to present a new approach to referential compression.
JDNA has some limitations, where it only compresses genomes and only achieves high compression rates in aligned DNA sequences. Unaligned genomes require a different compression algorithm, not covered by JDNA. This aligned input limitation is shared with FRESCO. JDNA is also limited in performance by the JVM, where the complex memory management of JVM increases the challenge of creating a memory efficient application. The amount of memory provided to JVM seems to influence the performance of the application, even if the application does not use all the memory provided (this has in consideration the different memory areas of JVM). JVM’s memory management requires further studies to improve JDNA’s performance. To compress entire genomes, JDNA’s performance can be increased through parallelism. Recall that JDNA compresses large files in independent blocks that are compressed separately. Parallelism may increase memory footprint, however it would significantly reduce compression time.

The main contribution of JDNA is a new compression algorithm through the introduction of an on-demand reference indexing method. This mechanism combines the best of two worlds: an index structure to cover the main differences between genome files and the simplicity and fast compression of direct matching. This creates a tool with compression ratios roughly equal to a full-indexing tool, but with much lower execution times.

JDNA is solely designed as a compression tool. However, as will be presented in the Chapter 5, there are workloads that may benefit from compressed files and avoid decompressing them for genome analysis. This would create a light tool to analyse genomes since there would be no need to have the whole genomes in memory.

We evaluate JDNA’s results as very positive. It not only we achieved high compression ratios in low time, as we also managed to keep a low memory usage in Java in a high memory usage scenario – referential compression. With compression ratios above 700 times, JDNA can be used to efficiently compress human genomes. However, we believe increasing compression ratio a little further is possible, by reviewing the applied Huffman encoding.

4.5 Conclusions and future work

We developed the JDNA referential compression tool. It is based on the FRESCO algorithm, but we tuned JDNA to be Java-efficient and greatly improved compression times by changing some algorithm steps. We also present on-demand indexing as a mechanism for referential compression of highly similar files. We believe the on-demand indexing mechanism can be applied to other referential tools, and this new approach can bring novelty to this study area.

We must obtain deeper knowledge on JVM memory management in order to improve entire genome compression time. There is also the possibility of developing a tool to analyse differences on genomes using only the compressed version. Another improvement is to add parallelism to block compression since each block is compressed with no connections to other blocks, each can be compressed separately. The last improvement thought so far is to automatically adapt block size and memory usage depending on the available hardware.
Chapter 5

RCSI: Scalable Similarity Search in Thousand(s) of Genomes

Chapter Authors:
Sebastian Wandelt, Johannes Starlinger, Marc Bux, and Ulf Leser (UBER).

Until recently, genomics has concentrated on comparing sequences between species. However, due to the sharply falling cost of sequencing technology, studies of populations of individuals of the same species are now feasible and promise advances in areas such as personalized medicine and treatment of genetic diseases. A core operation in such studies is read mapping, i.e., finding all parts of a set of genomes which are within edit distance $k$ to a given query sequence ($k$-approximate search). To achieve sufficient speed, current algorithms solve this problem only for one to-be-searched genome and compute only approximate solutions, i.e., they miss some $k$-approximate occurrences.

We present RCSI, Referentially Compressed Search Index, which scales to a thousand genomes and computes the exact answer. It exploits the fact that genomes of different individuals of the same species are highly similar by first compressing the to-be-searched genomes with respect to a reference genome. Given a query, RCSI then searches the reference and all genome-specific individual differences. We propose efficient data structures for representing compressed genomes and present algorithms for scalable compression and similarity search. We evaluate our algorithms on a set of 1,092 human genomes, which amount to approx. 3 TB of raw data. RCSI compresses this set by a ratio of 450:1 (26:1 including the search index) and answers similarity queries on a mid-class server in 15 ms on average even for comparably large error thresholds, thereby significantly outperforming other methods. Furthermore, we present a fast and adaptive heuristic for choosing the best reference sequence for referential compression, a problem that was never studied before at this scale.

5.1 Introduction

Since the release of the first human genome [24], the cost for sequencing has rapidly decreased. As of now, the price is at approx. 2,000 USD per genome and is expected to fall further once third generation sequencing techniques become available [103]. In contrast to previous years, where typically only one individual of a species was sequenced (like humans, mice, E.coli etc.), the decrease in cost makes it possible to sequence large samples of a given population. Such

\[^1\]Content of this chapter was previously published in Wandelt et al. 2013 [124]
studies, especially on humans, are interesting from many perspectives, such as correlation of specific mutations to the risk of developing a disease, to fine-tuned dosages of therapies, or simply to better understand the relationship between genotype and phenotype. For instance, the 1000 Genomes Project sequenced 1,092 human genomes to better understand population dynamics [1]; the International Cancer Sequencing Consortium is currently sequencing 50,000 human genomes to study the genetic basis of 25 types of cancer [25]; and the UK-10K project is sequencing 10,000 British individuals to better understand the impact of rare genetic mutations.

Studies at this scale use next generation sequencing (NGS) [103]. A property of NGS is that the sequences (reads) that are directly measured by the device are shorter (length in total of 30-200 base pairs) than with traditional Sanger sequencing, yet there are many more (hundreds of millions). Due to the highly repetitive nature of the human genomes, such short reads cannot be assembled to individual genomes; instead, the position of each read within a genome is typically determined by mapping the read against a single reference genome [46]. This problem is called read mapping or $k$-approximate search: Given a genome and a (short) read, find all substrings in the genome with an edit distance below $k$ to the read, where typical values for $k$ lie in the order of 1-2% of the read length, i.e., in the range of 1-3. $k$-approximate search is different from the classical bioinformatics problems of global alignment (measuring the difference between two entire sequences) or local alignment (finding maximally similar subsequences in the query and the to-be-searched sequence) [45], but of utmost importance when dealing with modern sequencing techniques [74]. A large number of algorithms for solving this problem appeared in recent years [72, 74, 75]. To scale to hundreds of millions of reads, all these algorithms compute an index (typically q-grams or variations of suffix trees) of the reference genome and perform several heuristic pruning tricks during the search, thus trading accuracy for time [8]. An inherent problem of this approach is the dominance of the reference sequence: Variations of individuals are defined by comparison to an arbitrarily chosen other individual. This is a severe constraint without any biological justification [106]; it merely exists for pure technical reasons, as no algorithm yet exists that can efficiently map reads against large sets of genomes.

In this chapter, we present the Referentially Compressed Search Index (RCSI), which follows a radically different approach to the $k$-approximate search problem and scales to a thousand genomes. Given a set of to-be-searched genomes $G$, RCSI first selects a reference $r \in G$. Next, it uses a referential compression algorithm [121] to compress all genomes in $G$ with respect to $r$. The intuition of a referential compression is to encode substrings of a to-be-

---

2See http://www.uk10k.org/
compressed string as positional references into the reference. This compression is lossless, can be computed quickly, and yields very high compression rates if applied to genomes of the same species; for instance, two randomly selected human genomes are approx. 99% identical [101]. The resulting data structure is a space-efficient representation of all common subsequences in $G$ and all differences. RCSI uses this data structure to solve the $k$-approximate search problem for all genomes in $G$ with a single search. Conceptually, RCSI has to search in two data structures per indexed genome: those parts that are identical to $r$ and those parts that are different from $r$; both parts may vary considerably between different genomes. We show how each of these types of information can be represented in a singleton data structure across all genomes by using a compressed suffix tree for the commonalities and another compressed suffix tree for the differences. Both data structures together allow solving the read mapping problem against multiple genomes efficiently and exactly. Our approach has the additional advantage that the index grows only very slowly with more and more genomes, i.e., RCSI scales very well with increasing data sets. The general idea of RCSI is depicted in Figure 5.1.

We evaluate RCSI on a set of 1,092 genomes recently released from the 1000 Genomes Project, a set that is more than 100 times larger than the data used for evaluation of similar methods within the last 13 months [78, 130]. RCSI compresses these 3 TB of raw sequence data down to a 115.7 GB (factor of 26:1) compressed search index. Using this index, RCSI answers $k$-approximate queries over 1,092 chromosomes in 1-20 ms on a laptop, and over 1,092 complete human genomes in 0.02-15.29 ms on a mid-sized server. Already for much smaller datasets, RCSI outperforms its competitors by a factor of at least 7.

RCSI always computes the exact answer to a given $k$-approximate search, regardless of which genome was chosen as reference for the compression. However, the achieved compression rates and the performance of searching do vary with different references. Therefore, we also study the problem of choosing the best reference for compression, i.e., the genome for which the compressed index for all genomes in $G$ is the smallest. We present an efficient heuristic for this problem using partial compressions. This heuristic shows considerable improvement compared to a random selection strategy and is almost as good as the perfect reference selection (computed exhaustively on a sample), while being orders of magnitude faster.

Altogether, our chapter makes the following contributions:

- We present a novel compression algorithm and highly tuned data structure to efficiently create compact, yet lossless representations of genomes with respect to a given sequence.
- We describe efficient algorithms operating on the compressed search index for answering $k$-similarity queries.
- We, for the first time, study the problem of selecting the best reference for referential compression and similarity search among a set of genomes and present an efficient and effective heuristic for solving this problem.
- We evaluate all algorithms on a set of 1,092 genomes, a data set that is highly realistic already today, yet far greater than data sets used in previous studies.

The remainder of this chapter is structured as follows: In the next section we present related work. We introduce the problems of similarity search and of referential compression in Section 3. Section 4 describes our novel algorithm and data structure, RCSI, in detail. Section 5 is devoted to the problem of finding the best reference. Our algorithms are evaluated in Section 6, and Section 7 concludes the chapter.
5.2 Related Work

Approximate search in strings (or sequences; we use both terms synonymously) has a long tradition in computer science [89]. If the to-be-searched string is large or many strings need to be searched, index-based methods provide the best performance. Many index structures have been proposed, such as suffix trees [35], suffix arrays [79], n-gram indexes [111], or prefix trees [102]. Besides the \( k \)-approximate search studied in this chapter, other variations of the problem have been studied intensively, such as approximate dictionary matching [23], searching probabilistic strings [76], searching with generalized distance functions [132] or searching with regular expressions instead of edit-distance constraints [61]. Similarity search over strings is fundamentally important for bioinformatics due to the fact that DNA and proteins can, for many applications, be represented as long strings; in this area, local alignment search is particularly important [5, 58]. The literature on string similarity search in general is vast and cannot be summarized here; we refer the reader to several excellent surveys [34, 87].

The \( k \)-approximate search problem is important in various applications, such as entity extraction [127] or pattern matching in time series [39]. In recent years, its importance in bioinformatics has grown tremendously because novel sequencing machines produce much shorter (yet many more) reads than the previous generation. Those reads cannot be assembled themselves into genomes. Instead, researchers map them to a given reference genome, which suffices to find mutations and variations and thus potential genetic predispositions of certain phenotypes or genetic diseases. This led to the development of a large number of algorithms for this so-called read mapping problem (e.g. [72, 74, 75]), which mostly build on n-gram hash indexes or on some variation of suffix trees or arrays. All these algorithms utilize thresholds for the number of allowed errors, and all perform some form of heuristic pruning to achieve high speed by sacrificing accuracy [8]. Furthermore, all of them map to a single reference, which is highly problematic from a biological point of view [106].

Our work uses a radically different approach. We first compress the set of genomes to be searched with respect to a reference genome. Next, we build an index over these compressed representations which is searched at query time. The idea of using compression as key idea to speed up string matching is not entirely new; recently, a number of works on this topic appeared almost concurrently. In [122], we describe an algorithm for exact search over a single compressed genome; here, we improve on this work by studying the much more important approximate case and by extending the search to multiple genomes. [78] sketches a prototype implementation for executing the popular search algorithm BLAST on compressed genomes; however, this method also gives-up on accuracy for being faster. Its scalability is unknown, as tests were only reported on small sets of small organisms (genomes less than 100 MB in size). The short-read aligner GenomeMapper [106] maps multiple reads simultaneously to multiple genomes using hash-based graph data structures for the reads and the genomes; thus, it actually studies a more general problem than we do (we always map only one read), but it cannot scale beyond a dozen genomes (Korbinian Schneeberger, personal communication). The most similar work to ours is GenomeCompress [130], though the proposed algorithm is considerably different. First, Yang et al. compress genomes using an alignment technique, while we greedily search for longest matches (see Section 3). Accordingly, the information encoded in the compressed genomes is quite different, leading to different search and indexing techniques. We will compare our method to this work in more detail in Section 5.6.5.

To search a large set of genomes as we do, one could also use a conventional method and build a separate index over each genome. Besides being highly space demanding (the size of a suffix tree for a sequence is 3-5 larger than the sequence itself [115], which implies that such an
index structure for 1,000 genomes would require 9-15 TB), searching these indexes in parallel for optimal performance would require significant investments in hardware. In contrast, our algorithm searches the same number of genomes in far less than a second on a modestly strong desktop computer.

There are also other interesting works not explicitly using genome compression. \cite{35, 88} use specialized suffix trees for indexing collections of highly repetitive sequences, thus implicitly performing a sort-of compression by representing common parts only once. Tests were performed on a data set of 500 MB only, and it remains unclear if the space consumption of the suffix tree would be manageable for a data set like ours (6,000 times larger). \cite{65} describes a self-index on LZ77 compressions of highly repetitive collections. The authors evaluate their approach on 37 DNA sequences of S. Cerevisiae, which sum up to an uncompressed size of merely 440 MB. In \cite{109}, multiple alignments of individual genomes are converted into a finite automaton and indexed with an extension of Burrows-Wheeler transform. The method is evaluated on four human chromosomes 18 (each one around 75 MB in size). Finally, \cite{93} also uses reference sequences to speed up global or local alignment of a query, but does not work with compression.

There is also some more theoretical work on searching over compressed string representations. \cite{38} studies the usage of grammars and LZ77 parsing for compression of similar sequence collections and improves complexity bounds with respect to space as well as time. Complexity bounds on searching LZ77 are also studied in \cite{32}. Neither of these papers provide a practical system or experimental evaluation.

5.3 Foundations

In this section, we formally introduce the \(k\)-approximate search problem over collections of large strings, and present the compression algorithm we use for RCSI.

5.3.1 \(k\)-approximate Search

In this work, a sequence \(s\) is a finite string\(^3\) over an alphabet \(\Sigma\). The length of a sequence \(s\) is denoted with \(|s|\) and the subsequence starting at position \(i\) with length \(n\) is denoted with \(s(i, n)\). \(s(i)\) is an abbreviation for \(s(i, 1)\). All positions in a sequence are zero-based, i.e., the first character is accessed by \(s(0)\). The concatenation of two sequences \(s\) and \(t\) is denoted with \(s \circ t\). A sequence \(t\) is a prefix of a sequence \(s\), if we have \(s = t \circ u\), for a sequence \(u\). A sequence \(s\) is a subsequence of sequence \(t\), if there exist two sequences \(u\) and \(v\) (possibly of length 0), such that \(t = u \circ s \circ v\).

**Definition 8** Given two sequences \(s\) and \(t\), \(s\) is \(k\)-approximate similar to \(t\), denoted \(s \sim_k t\), if \(s\) can be transformed into \(t\) by at most \(k\) edit operations. Edit operations are: replacing one symbol in \(s\), deleting one symbol from \(s\), and adding one symbol to \(s\). Given a sequence \(s\) and a sequence \(q\), the set of all \(k\)-approximate matches in \(s\) with respect to \(q\), denoted \(\text{search}(s)_q^k\), is defined as the set \(\text{search}(s)_q^k = \{(i, s(i, j)) \mid s(i, j) \sim_k q\}\).

**Definition 9** A sequence database \(S\) is a set of sequences \(\{s_1, \ldots, s_n\}\). Given a query \(q\) and a parameter \(k\), we define the set of all \(k\)-approximate matches for \(q\) in \(S\) as

\[\text{DBsearch}(S)_q^k = \{(l, \text{search}(s_l)_q^k) \mid s_l \in S\}.

\(^3\)We use the terms strings and sequences interchangeably during the rest of the chapter.
**Example 7** It holds that ATCGG $\sim_1$ ATGG, because the symbol C can be removed from ATCGG with one delete operation to obtain ATGG. If $S = \{s_1, s_2, s_3\}$, with $s_1 = ACACTG$, $s_2 = ACTGA$, $s_3 = GGCTA$, we have

$$DBsearch(S)_{CGA}^1 = \{(1, \{(1, CA)\}), (2, \{(1, CTGA), (2, TGA), (3, GA)\}), (3, \{(2, CTA)\})\}.$$ 

### 5.3.2 Indexing

Since we have several sequences in the sequence database $S = \{s_1, ..., s_n\}$, one can either build one index for each sequence or one combined index. A simple way to compute a combined index is to create an index on $s_1 \circ ... \circ s_n$. However, in many projects only genomes from one species are considered. These projects often deal with hundreds of highly similar sequences. We illustrate this scenario by an example.

**Example 8** Consider $s_1 = AATGAGAGCGTAGTAGAA$, $s_2 = TATGAGAGCGTAGTAGAG$, and $S = \{s_1, s_2\}$. Assume that we search all 1-approximate matches for CGC, i.e., we want to compute $DBsearch(S)_{CGC}$. It is easy to see that $s_1 \sim_2 s_2$, since replacing the first symbol of $s_1$ with T and the last symbol with G does the job. Computing $DBsearch(S)_{CGC}$ sequence by sequence would imply that many substrings are checked twice.

Similarity between sequences can be exploited for sequence compression using so-called referential compression schemes [20], which encode the differences of an input sequence with respect to a pre-selected reference sequence. Using a space-efficient encoding of differences and efficient algorithms for finding long stretches of bases without differences, the best current referential compression algorithm we are aware of reports compression rates of up to 500:1 for human genomes [31], which is much higher than the compression rates of non-referential schemes. Due to the quickly increasing number of sequenced genomes, compression is considered a key technology for genomic labs [96]. In this work, we show that compression can also be used to speed up similarity search.

### 5.3.3 Referential Compression

We define a very general notion for encoding referential matches, similar to [121].

**Definition 10** We define a referential match entry as a triple $rme = (\text{start}, \text{length}, \text{mismatch})$, where start is a number indicating the start of a match within the reference, length denotes the match length, and mismatch denotes a symbol. The length of a referential match entry, denoted $|rme|$, is length + 1.

Given a reference $ref$ and a to-be-compressed sequence $s$, the idea of referential compression is to find a small set of $rme$’s from $s$ with respect to $ref$ that is sufficient to reconstruct $s$.

**Definition 11** Given two sequences $s$ and $ref$, a referential compression of $s$ with respect to $ref$, denoted $\text{comp}(s, ref)$, is a list of referential match entries,

$$\text{comp}(s, ref) = [(\text{start}_1, \text{length}_1, \text{mismatch}_1), ..., (	ext{start}_n, \text{length}_n, \text{mismatch}_n)],$$
such that

$$(\text{ref}(\text{start}_1, \text{length}_1) \circ \text{mismatch}_1) \circ \ldots \circ (\text{ref}(\text{start}_n, \text{length}_n) \circ \text{mismatch}_n) = s.$$ 

Sometimes we also use $rc$ instead of $\text{comp}(s, \text{ref})$, if $s$ and $\text{ref}$ are known from the context or not relevant. The offset of a referential match entry $rme_i$ in a referential compression $rc = [rme_1, \ldots, rme_n]$, denoted $\text{offset}(rc, rme_i)$, is defined as $\sum_{j<i} |rme_j|$. The inverse of a referential compression $\text{comp}(s, \text{ref})$ is denoted $\text{decomp}(\text{comp}(s, \text{ref}), \text{ref})$. Given a referential match entry $(\text{start}_i, \text{length}_i, \text{mismatch}_i)$, we write $(\text{start}_i, \text{length}_i, \text{mismatch}_i) \in \text{comp}(s, \text{ref})$, if and only if $(\text{start}_i, \text{length}_i, \text{mismatch}_i)$ is an element in the referential compression $\text{comp}(s, \text{ref})$.

It is easy to see that $\text{decomp}(\text{comp}(s, \text{ref}), \text{ref}) = s$. The offset of a referential match entry in a referential compression corresponds to the position of the entry in the uncompressed sequence. The inverse of a referential compression is the decompression of a referential compression with respect to the reference, such that we obtain the original input sequence.

Clearly, we require the less $rme$’s, the longer the matches, i.e., the shared subsequences, are. It does not matter at which position of the reference these matches lie; in particular, matches need not be in any particular order. We exploit this observation in Algorithm 2. To create a referential compression of input sequence $s$ with respect to $\text{ref}$, the algorithm matches prefixes of $s$ with substrings of $\text{ref}$ using a compressed suffix tree of $\text{ref}$. The longest such prefix is removed from $s$, encoded as an $rme$ and added to $\text{comp}(s, \text{ref})$. The algorithm terminates once $s$ contains no more symbols. Please note that a referential compression of a sequence with respect to a reference is not unique. A simple example for a non-unique referential compression with respect to the reference $\text{ref} = \text{ATA}$ is $\text{comp}(\text{AA}, \text{ref}) = [(0, 1, A)]$ and $\text{comp}(\text{AA}, \text{ref}) = [(2, 1, A)]$.

The referential compression algorithm is greedy and optimal assuming that the storage necessary for referential match entries is uniform.

### 5.3.4 Referential Sequence Database Search

So far, we have considered only two sequences, an input and a reference. In the following, we study the problem of searching a database of sequences which are first referentially compressed with respect to a reference. We call this the referential sequence database search problem.

**Definition 12** Let $S = \{s_1, \ldots, s_n\}$ be a sequence database and $\text{ref}$ be a reference sequence. A referential sequence database $RS$ for $S$ and $\text{ref}$ is a tuple $\langle \text{ref}, rcs \rangle$, such that $rcs = [rc_1, \ldots, rc_n]$ is a list of referential compressions with $1 \leq i \leq n$, $rc_i = \text{comp}(s_i, \text{ref})$. Given a query $q$ and a parameter $k$, we define $RDB\text{search}(RS)_q^k$ as the set of all $k$-approximate matches for $q$ in the decompressed sequences of $RS$, i.e.,

$$RDB\text{search}(RS)_q^k = \{(l, \text{search}(\text{decomp}(rc_l, \text{ref}))_q^k) | rc_l \in rcs\}.$$ 

By definition, we have $DB\text{search}(S)_q^k = RDB\text{search}(RS)_q^k$ for every sequence database $S$ and each referential sequence database $RS$ for $S$. Solving the referential sequence database search problem will immediately solve the corresponding sequence database search problem.
5.4 Searching Referentially Compressed Sequences

The main contribution of our work is the transformation of the problem of \( k \)-approximate searching in large, highly similar sequences into \( k \)-approximate searching in referentially compressed sequences. We emphasize that it is not necessary to decompress any compressed sequence during the online search phase.

**Example 9** Suppose we want to search the referentially compressed sequence from previous Example 2 for occurrences of a string `TTGA` with \( k = 1 \). The situation is depicted in Figure 5.2. In this example, the query `TTGA` has four 1-approximate matches in `CGGACAAAC`T`GAGCTTCGACG`: substrings `CTGA` and `TGA` (both overlapping referential match entries 1 and 2) and substrings `TTCGA` and `TCGA` (both overlapping referential match entries 2 and 3).

Each match in a referential compression must be either 1) a match inside the reference part of a referential match entry or 2) overlapping at least one mismatch character.

**Proposition 1** Given a sequence \( s \), a referential compression \( \text{comp}(s, \text{ref}) \) for \( s \) with respect to a reference \( \text{ref} \), a query \( q \), and value \( k \), then for each \((p, m) \in \text{search}(s)_k\) there exists at least one \( \text{rme} = (\text{start}, \text{length}, \text{mismatch}) \in \text{comp}(s, \text{ref}) \) such that either

1. \( m \) is a subsequence of \( \text{ref}(\text{start}, \text{length}) \) or
2. \( p \leq \text{offset}(\text{comp}(s, \text{ref}), \text{rme}) + \text{length} \) and \( p + |m| \geq \text{offset}(\text{comp}(s, \text{ref}), \text{rme}) + \text{length} \).

This proposition gives rise to an algorithm for solving \( k \)-approximate string search problems over a single referentially compressed sequence (we extend this algorithm for multiple sequences below in this chapter):

1. Find all \( k \)-approximate matches inside the reference sequence and map these matches to referential match entries in the compressed sequence and
2. find all matches in subsequences overlapping at least one mismatch character of any referential match entry in the referentially compressed sequence.

The first step can be performed by using an index structure on the reference sequence. In our case, we can reuse the index for the reference sequence, which was used to create the referentially compressed sequences. The second step, finding all matches in sequences overlapping mismatch characters, needs more thought. First of all, the number of these overlapping sequences is equal to the number of referential match entries, since we have to create one such sequence for each mismatch character. The maximum length of these overlap sequences depends on the actual query length and error threshold \( k \).
5.4.1 Searching in Referential Match Entries

In order to find all matches inside referential match entries (to be more precise, inside the reference part of the referential match entries), the reference sequence is searched first and then all matches from the reference are post-filtered to identify all matches in referential match entries of the referentially compressed sequence.

The reference sequence is searched with the help of the index structure that was used to referentially compress the sequences. In our case, we have used a compressed suffix tree for the reference sequence. Exact matches can be found with a compressed suffix tree easily. For $k > 0$, we use the “seed-and-extend” paradigm, exploiting the fact that an alignment that allows at most $k$ mismatches must contain at least one exact match (”seed”) of a substring of length $\lceil \frac{|q|}{k+1} \rceil$, where $|q|$ is the length of the query [7]. The query is broken up into $k+1$ parts and each part is searched exactly in the compressed suffix tree. All matches for one of the query parts are extended in order to identify full $k$-approximate matches. The result of the seed-and-extend search, $\text{search}(\text{ref})^k_q$, is a set of $n$ matches of the form $(\text{pos}_i, m_i)$, such that each match is represented with a matching position $\text{pos}_i$ in the reference and the matching sequence $m_i$.

We define a projection operation, which transforms all matches in the reference into matches in referential match entries.

**Definition 13** Given a referential compression $rc$ with respect to sequence $\text{ref}$ and given $\text{search}(\text{ref})^k_q$, the set of projected matches, denoted $\text{project}(\text{search}(\text{ref})^k_q, rc)$, is defined as

$$\text{project}(\text{search}(\text{ref})^k_q, rc) = \{ (p, m) \mid \exists i, \text{pref}.((\text{start}_i, \text{length}_i, \text{mismatch}_i) \in rc \land \exists i, \text{pref}.((\text{start}_i, \text{length}_i, \text{mismatch}_i) \in rc) \land (p_{\text{ref}}, m) \in \text{search}(\text{ref})^k_q \land (p_{\text{ref}} \geq \text{start}_i) \land (p_{\text{ref}} + |m| \leq \text{start}_i + \text{length}_i) \land p = p_{\text{ref}} - \text{start}_i + \text{offset}(rc, (\text{start}_i, \text{length}_i, \text{mismatch}_i))) \}.$$

Depending on the number of referential match entries in the referential compression and the number of results in $\text{search}(\text{ref})^k_q$, different strategies for computing $\text{project}(\text{search}(\text{ref})^k_q, rc)$ show different performance. We have used an index structure for the start positions of all referential match entries (using hash buckets), in order to speed up the lookup of subsuming referential match entries for a set of given matches in $\text{search}(\text{ref})^k_q$. Especially in case of multiple queries, index structures over the referential match entries can improve the time needed for computation of projected matches.

**Example 10** Let sequence $s = \text{GACTATAACAGGATAC}$ and $\text{ref} = \text{AACAGGACTTTTATAC}$. One referential compression of $s$ with respect to $\text{ref}$ is $rc = [(5, 4, A), (10, 2, A), (2, 5, T), (1, 1, C)]$. Now assume a query $\text{AC}$ and $k = 0$. It follows that $\text{search}(\text{ref})^0_{\text{AC}} = \{ (1, \text{AC}), (6, \text{AC}), (13, \text{AC}) \}$. As a result of the projection we obtain $\text{project}(\text{search}(\text{ref})^0_{\text{AC}}, rc) = \{ (1, \text{AC}) \}$, because $(6, \text{AC})$ is contained in the rme $(5, 4, A)$. For the other two matches in $\text{ref}$ we cannot find an rme.

5.4.2 Searching across Sequence Deviations

Finding all matches overlapping at least one mismatch character is described next. In the example from Figure 5.2, the 1-approximate match CTGA overlaps the first and the second referential match entry. The length of these to-be-checked overlapping sequences depends on the
D2.2 – Reference-based Compression Algorithms

Figure 5.3: Extracting one overlap sequence for the referential match entries \((12, 9, T)\) and \((10, 4, T)\) of the referentially compressed sequence of \(CGGACAAACTGACGTTCGACG\) with \(QL_{max} = 4\) and \(k_{max} = 2\).

actual query length and \(k\). In the worst case, for exact string matching, the last (first) character of the query \(q\) matches the mismatch character. At most \(|q| - 1\) characters can be found to the left (right) of the mismatch character. In the case of approximate search, in the worst case, \(k\) symbols might be inserted in the sequence. Therefore, in addition \(k\) characters need to be extracted from the left and the right of the mismatch character. Assuming a maximum query length \(QL_{max}\) and a maximum edit distance \(k_{max}\), the overlap sequence is built by extracting the first \(QL_{max} + k_{max} - 1\) characters from the left of the mismatch character, concatenating the mismatch character, and concatenating the next \(QL_{max} + k_{max} - 1\) characters to the right of the mismatch character.

**Definition 14** Given a referential compression \(rc\) for the sequence \(s\), and a referential match entry \(rme_i \in rc\), with \(rme_i = (\text{start}, \text{length}, \text{mismatch})\), the overlap sequence of \(rme_i\) with respect to \(rc\), denoted \(\text{ovl}_{rc}^{ref}(rme_i)\), is defined as

\[
\text{ovl}_{rc}^{ref}(rme_i) = (\text{decomp}(rc, ref))(\text{offset}(rc, rme_i) + \text{length} - (QL_{max} + k_{max} - 1), 2 \times (QL_{max} + k_{max} - 1) + 1).
\]

The set of overlap sequences for a referential compression \(rc\) with respect to \(ref\), denoted \(\text{overlaps}(ref, rc)\), is defined as

\[
\text{overlaps}(ref, rc) = \{ (\text{offset}(rc, rme) + \text{length} - (QL_{max} + k_{max} - 1), \text{ovl}_{rc}^{ref}(rme)) \mid rme \in rc \}.
\]

In fact, the referentially compressed sequence does not have to be decompressed completely in order to compute \(\text{ovl}_{rc}(rme_i)\). It is sufficient to partially decompress \(QL_{max} + k_{max} - 1\) symbols to the left and to the right of the mismatch character in \(rme_i\).

**Example 11** An example for the extraction of an overlap sequence is shown in Figure 5.3. Given \(QL_{max} = 4\) and \(k_{max} = 2\), the overlap sequence for the referential match entries \((12, 9, T)\) and \((10, 4, T)\) is extracted, yielding \(CAAACTGACGT\). The length of the overlap sequence is \((4 + 2 - 1) + 1 + (4 + 2 - 1) = 11\) symbols. The referential compression for \(CGGACAAACTGACGTTCGACG\) with respect to the reference \(GACGATCGACGACGGACAAACA\) contains three referential match entries. Therefore, two more overlap sequences have to be extracted: \(TGACGTTCGAC\) (for the overlap of entries number two and three) and \(TCGACG\) (for the final referential match entry, which is only extended to the left).
Algorithm 6 Referential Search Algorithm

**Input:** Referential sequence database \( \langle \text{ref}, \text{rcs} \rangle \) with \( \text{rcs} = [\text{rc}_1, \ldots, \text{rc}_n] \), query \( q \), and \( k \).

**Output:** Solution \( \langle \text{ref}, \text{rcs} \rangle^k_q \)

1: \( \langle \text{ref}, \text{rcs} \rangle^k_q = \emptyset \)
// First step – search reference
2: \( \text{refmatches} = \text{search}(\text{ref})^k_q \)
3: for \( 1 \leq i \leq n \) do
4: \( \text{Add} \ (i, \text{project} (\text{refmatches}, \text{rc}_i)) \) to \( \langle \text{ref}, \text{rcs} \rangle^k_q \)
5: end for
// Second step – search overlap sequences
6: for \( 1 \leq i \leq n \) do
7: for \( (\text{pos}, t) \in \text{overlaps}(\text{ref}, \text{rc}_i) \) do
8: for \( (\text{pos}_2, u) \in \text{search}(t)^k_q \) do
9: \( \text{Add entry} \ (i, (\text{pos} + \text{pos}_2, u)) \) to \( \langle \text{ref}, \text{rcs} \rangle^k_q \)
10: end for
11: end for
12: end for

In order to completely search the referentially compressed sequence of \( \text{CGGAC} \text{AAACTGACGTTCGACG} \), the reference sequence needs to be searched (following the seed-and-extend approach), and, in addition, three shorter strings have to be searched, i.e., one for each referential match entry. For the sake of simplicity, the length of referential matches are chosen rather small in this example. In general, the number of overlap sequences is equal to the number of referential match entries in these sequences. However, in case of collections of highly-similar DNA sequences, many extracted overlap sequences turn out to be identical, as most differences between human genomes are rather short and there exist only three possible deviations. This effect is the stronger for shorter maximum query lengths, and it would be weaker if strings over a larger alphabet were searched. We evaluate the number of non-identical overlaps in Section 5.6.

5.4.3 Searching Referential Sequence Databases

The complete referential search algorithm is shown in Algorithm 6. The algorithm solves a referential sequence database search problem \( \langle \text{ref}, \text{rcs} \rangle^k_q \). In Line 2, all \( k \)-approximate matches inside the reference sequence are computed. In our implementation we have used compressed suffix trees (with seed-and-extend for \( k > 0 \)). The for-loop from Line 3 to Line 5 projects these matches from the reference sequence onto the referential match entries in the referential sequence database. The remaining part of the algorithm (Line 6-13) finds all matches in overlapping sequences. The first loop iterates over all the referential compressions in the referential sequence database. The second inner loop (starting Line 7) iterates over all overlapping sequences of the referential compression \( \text{rc}_i \). The innermost loop (starting Line 8) iterates over all \( k \)-approximate matches inside the current chosen overlap sequence \( u \) and adds these matches to the solution set. Note that adding elements to \( \langle \text{ref}, \text{rcs} \rangle^k_q \) might require care, in case a match for a sequence with the same identifier exists already.

**Example 12** We want to search for \( 0 \)-approximate occurrences of \( q = \text{AA} \) (with fixed \( QL_{\max} = 3 \) and \( k_{\max} = 0 \)) in sequences:

- \( s_1 = \text{CGGACA} \text{AACTGACGTTCGACG} \)
We want to compute \( \langle \text{ref}, [\text{rc}_1, \text{rc}_2, \text{rc}_3] \rangle^{0}_{\text{AA}} \). During the first step of the algorithm, we obtain
\[
\text{ref matches} = \text{search} (\text{ref})^{0}_{\text{AA}} = \{(17, \text{AA}), (18, \text{AA})\}.
\]

Projecting these reference matches onto the referential compression \( \text{rc}_1 \), we add \( (1, \{(5, \text{AA})\}) \) and \( (1, \{(6, \text{AA})\}) \) to the solution \( \langle \text{ref}, [\text{rc}_1, \text{rc}_2, \text{rc}_3] \rangle^{0}_{\text{AA}} \).

For \( \text{rc}_2 \) and \( \text{rc}_3 \), we add the results \( \{(2, \{(5, \text{AA})\}), (2, \{(6, \text{AA})\})\} \) and \( \{(3, \{(5, \text{AA})\}), (3, \{(6, \text{AA})\})\} \), respectively. In the second step, all matches in overlapping sequences are added. The overlap sequences are:
\[
\text{overlaps}(\text{ref}, \text{rc}_1) = \{(7, \text{ACTGA}), (12, \text{CGTCC}), (18, \text{ACG})\}
\]
\[
\text{overlaps}(\text{ref}, \text{rc}_2) = \{(7, \text{ACAGA}), (12, \text{CGTCC}), (18, \text{ACC})\}
\]
\[
\text{overlaps}(\text{ref}, \text{rc}_3) = \{(7, \text{ACTGA}), (12, \text{CGTCC}), (17, \text{GAA})\}
\]

The only overlap sequences with 0-approximate match for \( \text{AA} \) is \( \text{GAA} \) for referential compression \( \text{rc}_3 \). Therefore, the algorithm adds the match \( (3, \{(18, \text{AA})\}) \) to \( \langle \text{ref}, [\text{rc}_1, \text{rc}_2, \text{rc}_3] \rangle^{0}_{\text{AA}} \).

The overall result of Algorithm 6 for the example is:
\[
\langle \text{ref}, [\text{rc}_1, \text{rc}_2, \text{rc}_3] \rangle^{0}_{\text{AA}} = \{(1, \{(5, \text{AA})\}), (2, \{(5, \text{AA})\}), (3, \{(5, \text{AA}), (6, \text{AA}), (18, \text{AA})\})\}
\]

One interesting observation from the example is that the number of unique overlap sequences can be smaller than the total number of referential match entries. In total, we have to check nine overlap sequences: one for each referential match entry inside a referentially compressed sequence. However, extracting the actual overlaps yields that we only have six unique overlaps, since \( \text{CGTCC} \) occurs three times and \( \text{ACTGA} \) occurs twice. This observation is important when searching the overlaps for \( k \)-approximate matches: instead of naively searching each overlap sequence, we find (and remove) identical overlap sequences in a preprocessing step. The preprocessing step is implemented using hash tables and can greatly speed up the actual query answering time.

All unique overlap sequences are searched for matches using a very simple approach: all overlaps are concatenated to a large string (separated by \( k_{\text{max}} \) fresh symbols, not contained in the alphabet of sequences). For instance, the overlaps \( \text{AGT} \) and \( \text{AC} \) are stored as \( \text{AGT}^{**} \text{AC} \), if \( k_{\text{max}} = 2 \).

We use a compressed suffix tree to find matches in the concatenated string in the same way as we search the reference sequence. Following the seed-and-extend approach, we can find all \( k \)-approximate matches in all overlaps efficiently. Matches in the large concatenated string are being projected back to the single overlaps. The idea is depicted in Figure 5.4. Although this approach is quite simple, it scales surprisingly well. Standard data structures for \( k \)-approximately searching collections of strings [126] should improve search times. Such an optimization is left for future work. Note that we do not uncompress any overlaps during the search phase, since we build the above compressed suffix tree over all overlaps.
5.5 Best Reference Selection

One open problem when searching compressed sequences is the selection of a best reference sequence with respect to our referential compression algorithm. With increasing similarity between reference and to-be-compressed sequence, longer referential match entries can be found and the compression ratio is increasing. Thus, choosing a proper reference will increase the compression ratio and also reduce search times.

**Definition 15** Given a sequence $s$ and a set of candidate references $\{ref_1, ..., ref_m\}$, $ref_i$ is called an optimal reference iff there does not exists a $j \neq i$ with $|comp(s, ref_j)| < |comp(s, ref_i)|$.

A naive strategy to find an optimal reference sequence is to compress all the to-be-compressed sequences against all possible reference sequences and select the reference that yields the least number of referential match entries, named RSbest. If sequences are long, as in our case, this is a highly time consuming undertaking as we need to compute $m^2$ referential compressions, where $m$ is the number of candidate reference sequences. If one wants to compress 1,000 sequences, choosing the best reference following this strategy does not scale.

In the following, we describe a heuristic which scales well in the number of candidates and, as shown in the evaluation section, in many cases identifies near-optimal references. However, the problem of efficiently finding an optimal reference remains unsolved and is an important topic for future work (see Section 5.7).

Instead of compressing a sequence against all candidate references, we compare the referential compression of the sequence and the referential compression of the reference candidates with respect to one chosen base reference. We then select that reference whose referential compression against this base references has the highest referential similarity to the referential compressions of all sequences to the base reference. The idea is that two referential compressions are more similar if they share more referential match entries.

**Definition 16** The referential similarity of two referential compressions $rc_1$ and $rc_2$, denoted $rsim(rc_1, rc_2)$, is defined as

$$rsim(rc_1, rc_2) = |rc_1 \cup rc_2| - |rc_1 \cap rc_2|.$$  

As a second optimization, we use only parts of the sequences to select the best reference, building on the fact that the degree of similarity between similar long sequences does not change.
Figure 5.5: Compressing human chromosomes against HG19. Values are averaged over 1,092 genomes.

significantly between local stretches. The full algorithm, called RSbitX, is shown in Algorithm 3. RSbit5 stands for compressing only \( \frac{1}{5} \) of each sequence. Note that RSbitX computes \( m + n \) referential compressions and \( n \times m \) referential similarities.

5.6 Evaluation

We evaluated our RCSI algorithm using different data sets, settings, and computers. Most experiments were run without any parallelization on an Acer Aspire 5950G with 16 GB RAM and an Intel Core i7-2670QM processor; the exception is Section 5.6.4, for which we used a server with 1 TB RAM and 4 Intel Xeon E7-4870 (in total, hyperthreading enables the use of 80 virtual cores). Code was implemented in C++, using the BOOST library, CST [91], and libz.

5.6.1 Data and Queries

We tested our method on a set of 1,092 human genomes from the 1000 Genomes Project[1]. The data is originally provided in the Variant Call Format (VCF) [28]4, which we converted into raw consensus sequences for each chromosome of each genome. The total dataset has 3.09 TB uncompressed, or 700 GB when compressed with GZip. Note that in the following experiments we sometimes search on sets of chromosomes and sometimes on the full set of genomes.

We measured search times on different sets of sequences and different edit distance thresholds. For those measurements, we created a set of 50,000 queries of length 120-170 (equally distributed) for each chromosome by (1) randomly extracting substrings and (2) adding modifications: Bases were randomly replaced with a probability of 0.05 and single bases were added/removed with a probability of 0.01 percent, respectively. For scalability studies, we used randomly chosen subsets of the input genomes of sizes 5, 10, 20, 40, 80, 160, 320, 640, and 1,092.

5.6.2 Compression

We first evaluate our compression scheme using as reference the human genome HG19 [59], which is commonly used as a human reference genome (note that the experiments described below used a different – and better from a compression point of view – reference). The size of the

4ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20110521/
uncompressed chromosomes of HG19 varies between 50 MB (Chromosome 22) and 250 MB (Chromosome 1). We created a compressed suffix tree for each chromosome for performing the compression of the 1,092 genomes. These suffix trees are on average 1.72 times larger than the input sequences. The time required for creating them grows linearly with the size of the input and took 17-118 s (Chromosome 22, Chromosome 2), which amounts to a throughput of approx. 2.4 MB/s. Compressing a complete genome took approx. 30 s on average; compressing all 1,092 genomes took roughly eight hours.

Figure 5.5 shows the average sizes of all 1,092*23 compressed chromosomes using different compression techniques. With bit manipulation techniques (BM), i.e., encoding three symbols within one byte, we obtain a ratio of 3:1. GZip achieves a ratio of 4:1 on average. In contrast, RCSI yields an average compression ratio of 436:1.
D2.2 – Reference-based Compression Algorithms

5.6.3 Searching 1,092 Chromosomes

We compressed all 1,092 genomes against the reference chosen by RSbit5 (see Section 5.6.6) and created RCSI for $QL_{\max} = 200$ and $k_{\max} = 5$. In total, this process took 54 hours (40 min for Chr22, 50 MB, and 3.2 hours for Chr2, 250 MB) on our test laptop. The size of the entire search index is 115 GB (1.7 GB for Chr22 and 9.1 GB for Chr2); details are shown in Figure 5.6.

We next ran 50,000 chromosome-specific queries on each of the chromosome-specific indexes (refer to Section 5.6.4 for searching all 1,092 genomes). Figure 5.7 studies the impact of the error threshold and the number of genomes on search performance. For exact string matching, average search time per query is between 0.06 ms (for five sequences) and 0.25 ms (for 1,092 sequences). Runtimes are fairly constant for the different chromosomes, i.e., the size of a chromosome only has a negligible effect on runtime (data not shown). For $k = 1$, the average search time per query is still below 1 ms even for 1,092 chromosomes. Starting from $k = 4$, average search runtimes increase recognizably. For $k = 4$ and 1,092 chromosomes, we need 6 ms to search one query. For $k = 5$, searching 1,092 sequences already takes 23.6 ms on average; however, the median is only 4.6 ms. In total, searching 1,092 sequences takes only 10 times more search time than searching five sequences, showing that the use of compression gives our algorithm very good scalability in the size of the data set.

We further analyze query runtimes in Figure 5.8. While the slowest exact query only takes around 1 ms, the slowest 5-approximate query needs almost 1 s. However, for values of $k < 5$, almost all queries (exactly 98.4%) can be answered in less than 10 ms; even for $k = 5$, 85.5% of queries take less than 10 ms. The reason for this deviation can be seen in Figure 5.9, where we break down search times by result sizes. Due to the high number of repeats in the human genome (up to 60% of the human genome consist of repetitive elements [29]), some queries fall into regions that appear very often throughout the genome (recall that queries were sampled at random). These queries with 10,000 and more results require excessive runtimes, which explains the large deviation between mean and median runtimes. Note however, that in biological applications such regions are typically masked before performing searches as results are uninformative for all but very few questions (see, for instance, RepeatMasker). One way to solve the problem with large result sets might be to introduce some kind of polynomial delay algorithm[41], returning the most similar matches first, or those with the fewest number of occurrences.

5.6.4 Searching 1,092 Genomes

So far, all our measurements were obtained by running chromosome-specific queries on a commodity laptop. Clearly, the obtained speed can be scaled up by using 23 cores (one for each chromosome) and sufficient memory to load the entire index into memory. To show the feasi-
bility of this strategy, we performed an experiment on a mid-size server with 1 TB RAM and 80 virtual cores. We created a workload of 23,000 queries (from each chromosome, 1,000 queries were taken from the set described in the beginning of Section 5.6), and searched each query against all 1,092 genomes in parallel, where each core searched one chromosome-specific index. Average runtimes per query are shown in Figure 5.10. On average, exact matching takes 0.02 ms, 1-approximate matching 0.09 ms, and 5-approximate matching 15.29 ms. The difference between these average runtimes and those reported in previous chapters are due to the more powerful CPU of the server compared to our laptop.

5.6.5 Competitive Evaluation

We are aware of only one other tool that follows a similar approach to RSCI: GenomeCompress. Other algorithms either create indexes that are much larger than the to-be-searched sequences and are thus not applicable for the data sets we target [79, 91], or provide only incomplete solutions and often solve slightly different problems. Still, we find it instructive to compare against such tools as it shows that, for the special setting of searching similar sequences, RSCI scales as well or even better than these heuristics. Therefore, in the following we compare RSCI against GenomeCompress, BLAST, and Bowtie 2.

We compare our runtimes against a range of competitors, namely BLAST, the standard search tool for local alignments, Bowtie 2, a state-of-the-art read mapper, and GenomeCompress, an algorithm following a similar approach as RCSI.

First, we illustrate the speed-up of RCSI compared to the popular sequence search tool BLAST [5]. BLAST is capable of searching high data volumes; for instance, it is used at the GenBank servers where it runs on large clusters to serve thousands of queries per day on the archive currently containing approx. 145 GB (which is 20 times smaller than our data set). Note that RCSI is not directly comparable to BLAST, as RCSI exactly answers $k$-approximate searches, while BLAST is a heuristic to find local alignments; still, we believe that the differences in runtimes are interesting. We indexed only HG19 [59] with BLAST, leading to an index of 4.9 GB. BLAST queries (with default parameters) on this single genome took 12 s on average, with extreme cases taking several minutes. Experiments with one to eight Chromosome 1 showed that search times grow linear with the number of chromosomes (63 ms for one chromosome and 328 ms for eight chromosomes); the same holds for database size (62 MB for one chromosome and 498 MB for eight chromosomes). This is in stark contrast to the runtimes and scalability behaviour of RCSI.

Next, we compared RCSI against Bowtie 2 [71], a state-of-the-art tool for mapping long sequence reads (50-1000 bp) against a reference genome. In contrast to many other read mappers which only cope with base substitutions, Bowtie 2 also allows small gaps in matches and
is thus in principle comparable to RCSI. However, there are also important differences: (1) Bowtie 2 only reports best matches, while RCSI computes all matches within the error threshold; (2) Bowtie uses several heuristics to filter repetitive or generally uninformative matches, while RCSI finds all matches; (3) Bowtie searches a single reference, while RCSI searches many references implicitly in parallel. Indexing HG00096 with Bowtie 2 took around three hours and the index size is 3.9 GB. Query times (with default parameters set) are very fast, with an average time of 0.11 ms, which is about the time RCSI needs to answer 1-approximate queries when working in parallel (see Section 5.6.4). We tried to generate a composite index for several human genomes (by simply concatenating them), but failed to do so as Bowtie 2 can handle only sequences up to around 3.6 GB. For larger sequences the developers of Bowtie 2 propose to split up the input sequence into smaller chunks and create single indexes for each chunk. For 1,092 human genomes, one would need to create more than 1,000 indexes, summing up to more than 3 TB of storage. Unless all these indexes can be queried in parallel (on more than 1,000 cores), the average query time will increase recognizably.

Finally, we compared RCSI to two other methods which solve exactly the same problem. As a baseline, we simply built one compressed suffix tree for each sequence and searched these one-after-the-other (we call this method IndexBased in the following). Further, we installed GenomeCompress [130], a very recent tool that also builds on genome compression (see Section 5.2). GenomeCompress takes so-called delta files as input, i.e., files that describe the difference to a reference. The tool transforms these files into a compressed index over multiple sequences. As delta files represent sets of edit operations, the algorithm to search the compressed index is considerably more complex than ours which essentially searches just strings - either from the reference or from an input sequence or from both. Unfortunately, the code provided with GenomeCompress does not contain an algorithm to generate those delta files (X. Yang, personal communication); however, since delta files are not unique and the concrete representation has an impact on the compressed index, we think that re-implementing this step could add bias to the comparison. Therefore, we could compare GenomeCompress to RCSI only on those sequences used in [130] (for those, delta files are provided). This dataset consists of up to 1,000 sequences taken from the first 10 MB of a Chromosome 1, giving a total size of only approx. 10 GB (uncompressed). Queries were randomly generated as before.

Figure 5.11 shows the size of indexes for a growing numbers of sequences. The size of IndexBased grows linearly since one compressed suffix tree is created for each sequence. For five sequences the index size of GenomeCompress is roughly three times smaller (275 MB) than for RCSI (736 MB), but with increasing number of sequences, index sizes of GenomeCompress and RCSI become very similar. For 1,000 sequences (of length 10 Mbases), GenomeCompress requires 1,550 MB and RCSI needs 1,650 MB. Besides the footprint on disk, we also measured memory consumption of RCSI and GenomeCompress for different number of sequences; see Figure 5.12. Interestingly, the runtime memory footprint of RCSI is larger than that of Genome-
Compress for sets of up to 400 sequences, but grows only very slowly with more sequences. The slope of GenomeCompress is much steeper. For 1,000 sequences, the main memory usage already almost doubled (2,200 MB for GenomeCompress, and 1,300 MB for RCSI). If the main memory usage keeps on growing in this way then GenomeCompress will not be able to manage an index for 1,000 complete genomes in 1 TB of main memory. The indexing times for GenomeCompress are a little bit higher than for RCSI: roughly a factor of five. Note RCSI works directly on raw sequences, while GenomeCompress is run on preprocessed input, produced by a process similar to global sequence alignment. The time for this preprocessing step of all input sequences is not included in the indexing times for GenomeCompress, but will increase indexing times considerably.

Average search time for different numbers of sequences are compared in Figure 5.13. Clearly, search times for IndexBased grow linearly with the number of sequences. Searching 1,000 sequences with IndexBased takes 1 ms (for exact search) and 230 ms (for 3-approximate search; recall that here we only search 10 MB of each Chromosome 1), respectively. GenomeCompress on average needs 8.8 ms for exact search in 1,000 sequences while RCSI needs only 0.07 ms. For 3-approximate search, RCSI is roughly 7 times faster than GenomeCompress (5.3 ms vs. 35 ms) on the full set, and the advantage seems to grow with more compressed and larger sequences. The main memory storage required per sequence is roughly constant for GenomeCompress (1.7 MB/sequence). Therefore, doubling the number of sequences will yield double
D2.2 – Reference-based Compression Algorithms

Figure 5.14: Difference between optimal, average and worst compression rate per chromosome, depending on reference.

Figure 5.15: Space overhead by reference selection method compared to best reference. Except for Chr16, RSbit1 and RSbit5 select references clearly better than RSrand.

main memory usage. For RCSI the necessary storage is decreasing with an increasing size of the database (1.2 MB/sequence at 160 sequences, 0.27 MB/sequence at 1,000 sequences). This shows improved scalability of RCSI over GenomCompress for the very small dataset already. We conjecture that 1,092 complete genomes cannot be kept within even 1 TB of main memory with the current implementation of GenomeCompress.

Please note that in our experiments GenomeCompress did not find all $k$-approximate matches because it cannot find matches shorter than a given threshold when compression is enabled (X. Yang, personal communication).

5.6.6 Best Reference Selection

We tested our methods for finding the best references using a set of nine randomly chosen candidates: HG00236, HG01048, HG01360, NA06994, NA18946, NA19028, NA19445, NA20508, and HG19. Testing against all genomes would require an estimated time of 7,500 hours; however, since the range of obtained compression rates remains fairly robust (exhaustive evaluation of all 1,092 reference candidates for Chromosome 22; data not shown), we believe that using a randomly selected subset is sufficient to show the achievable improvements. Compression sizes for the best, worst, and average reference are shown in Figure 5.14. In total, storing all genomes compressed with respect to the best reference requires 7.1 GB, while storing them with respect to the worst reference needs 8.8 GB (difference approx. 20%). Note that we always chose the optimal reference for each chromosome separately, i.e., the complete reference is composed of chromosomes from different individuals. Compressing all genomes with respect to all nine references took 67 hours.
We compared the space consumption and compression time for RSbit1 and RSbit5 (i.e. RSbitX with X=1 and X=5, see Section 5.5), the exhaustive strategy RSbest, and a random selection strategy (RSrand) on our test set of nine genomes. Figure 5.15 shows the increase in storage depending on the reference selection method with baseline RSbest. RSrand leads to an average increase in storage of 7.6% compared to the optimal reference. RSbit1 performs significantly better and leads to only 1.8% increase. RSbit5 even slightly outperforms RSbit1 (1.7%). Only for Chromosome 16, RSbit1 and RSbit5 choose a reference worse than RSrand. Our experiments indicate that this is caused by the extremely high number of repeats in Chromosome 16 [2].

5.7 Conclusions

We presented RCSI, a novel method for searching thousands of human genomes. RCSI first compresses all genomes with respect to a reference. The resulting data structure, when encoded properly, is much smaller than the raw data and can be searched efficiently, thereby implicitly searching all genomes in parallel. Experiments with 1,092 genomes show that runtimes on a commodity laptop are in the range of 1-20 ms when searching a specific chromosome, or roughly in the same range when searching entire genomes on a mid-class server. We showed that RCSI considerably outperforms close and less close competitors. We also studied the problem of reference selection and presented heuristics that result in an additional 15% space reduction compared to a random selection.

Though this is not a dramatic space reduction, we believe that the topic of reference selection deserves more research in the future. First, the question of efficiently finding an optimal reference remains open. Second, there is no need to choose the reference from the set of sequences to be compressed (as we did); instead, any other or also an artificial sequence could be used. Accordingly, another open problem is that of efficiently creating an optimal reference for a set of to-be-compressed genomes.
Chapter 6
Concluding Remarks

Referential-compression is a fundamental technique for dealing with data avalanche associated with NGS methods. In this deliverable we discussed the state of the art in genome compression and presented three contributions from the BiobankCloud consortium with respect to referential-compression. These contributions comprises (1) an open source tool that matches and outperforms all other genome compression tools in terms of both performance and compression ratio, (2) a novel technique (also made available as a Java programming library) for avoiding most of reference-indexing phase of reference-based compression and (3) a technique and tool for analyzing the similarity between compressed genomes.

These contributions are the main outcome of Task 2.3, which finishes in M20 of the project, and will be fundamental for several other work packages and tasks of BiobankCloud, namely:

- **Compression as a workflow task**: several use cases defined for the BiobankCloud platform consider the use of compression tools (see deliverable 6.2). FRESCO (described in Chapter 3) will be the standard tool available in the BiobankCloud workflow engine to deal with compressed genomes;

- **Genome storage and analysis in the cloud**: the BiobankCloud data sharing infrastructure (Overbank, developed in WP4) considers the use of public clouds as a medium for securely storing (genome) data. Storing this data without compressing it is costly both in terms of performance (these big data files will be sent through the low-bandwidth internet links) and money (clouds charge per stored GB/month). The tools presented in this deliverable will be used for efficient storage of genome data in the cloud. In particular, RCSI can be used to run similarity search tasks directly in the cloud, operating over compressed genomes, without requiring further data transfers.

- **Efficient data transfer between biobanks**: due to legal and regulatory constraints, Biobanks may not be able to store data in public clouds. Therefore, the Overbank must also support the direct transfer of data between cooperating biobanks using the platform (as will be described in the deliverable 4.2). Using the developed compression tools, we expect to make these transfers much more efficient.
Bibliography


