FRESCO: Referential Compression of Highly-Similar sequences

Sebastian Wandelt and Ulf Leser

Abstract—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 paper, 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 1000 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.

Index Terms—Sequences, referential compression, second-order compression, compression heuristics

1 Introduction

Since the release of the first human genome [1], 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 [2]. In contrast to previous years, where typically only one individual of a species was sequenced (like humans, mice, E.coli, etc.), the decrease in costs makes it possible to sequence large samples of a given population. 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 [3]; activities of the international cancer sequencing consortium [4]; and the UK10K project [5]. These large-scale projects are generating comprehensive surveys of the genomic landscape of various diseases by sequencing thousands of genomes [6]. Managing, storing and analyzing this quickly growing amount of data is challenging [7]. It requires large disk arrays for storage, and large compute clusters for analysis. A recent suggestion is to use cloud infrastructures for this purpose [8]–[10]. However, before 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 [11]. Accordingly, sequence compression is a key technology to cope with the increasing flood of DNA sequences [12], [13].

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 compression schemes can reduce the space requirements by up to 6:1 (one base is encoded with up to 1.3 Bits) [14], [15]. 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 [16], 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 [17].

In this paper, 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

• *S. Wandelt and U. Leser are with Knowledge management in bioinformatics, Humboldt-University of Berlin, Berlin, Germany. E-mail: {wandelt,leser}@informatik.hu-berlin.de*

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 sequences 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: 1092 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 paper is structured as follows. We discuss related work on compression of sequences in Section 2. We motivate and formally define our data structures and algorithm for referential compression in Section 3. In Section 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 reference 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 5, called second-order compression. We evaluate all our methods in Section 6. Section 7 describes the open-source release of FRESCO and the paper is concluded in Section 8.

2 Related Work

Naive bit encoding algorithms exploit encodings of two or more symbols into one byte, using fixed-length encodings [18]. 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 [19]–[21]. Lempel-Ziv-based compression algorithms, such as LZ77 or LZ78, are prominent examples of dictionary-based algorithms [22]. 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 [23]–[25]. One of the most commonly used and best understood statistical encodings is Huffman encoding [26]. 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 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. During the last years several referential compression algorithms emerged [17], [27]–[30].

In [27], 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 [28], 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 [17]. 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 codes, enabling random access. Another LZ77-style compression scheme with random access is proposed in [30].

[31] presented a compression scheme inspired by image compression techniques based on controlled loss of precision. GRS [32], 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 compres-

Algorithm 1 Referential Compression Algorithm

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

Let $comp(s, ref)$ be an empty list
while $|s| \neq 0$ **do**
  Let pre be the longest prefix of s occurring in ref, and let i be a position of an occurrence of pre in ref
  if $s \neq pre$ **then**
    Add $\langle i, |pre|, s(|pre|)\rangle$ to the end of $comp(s, ref)$
    Remove the first $|pre| + 1$ symbols from s
  else
    Add $\langle i, |pre| - 1, s(|pre| - 1)\rangle$ to the end of $comp(s, ref)$
    Remove the prefix pre from s
  end if
end while

sion scheme, was recently proposed in [29]. Inspired by compression scheme XM [24] 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 [33]–[37].

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.

3.1 Representation of Matches into a Reference

A *string* s is a finite sequence over an alphabet Σ. 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 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 [17], [27]. Another option is to encode parts of a sequence with original text entries instead of matches into the reference [28], [38]. 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: $[\langle 0, 2, T\rangle, \langle 3, 4, A\rangle, \langle 7, 1, T\rangle]$

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 $(0, 1)$ 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 $\langle start, length, mismatch\rangle$, 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 rme, denoted $|rme|$, is $length + 1$.

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

$$comp(s, ref) = [\langle start_1, length_1, mismatch_1\rangle, ..., \langle start_n, length_n, mismatch_n\rangle],$$

such that

$$(ref(start_1, length_1) \circ mismatch_1) \circ (ref(start_2, length_2) \circ mismatch_2) \circ ... \circ (ref(start_n, length_n) \circ mismatch_n) = s.$$

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

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.

Example 2: An example referential compression for the string $CGGACAAACTGACGTTCGACG$ with respect to the reference

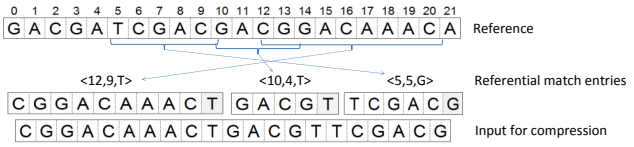

Fig. 1: Example for one referential compression

$GACGATCGACGACGGACAAACA$ is shown in Figure 1. The input is compressed into three referential match entries. The first referential match entry is $\langle 12, 9, T\rangle$, 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 $\langle 5, 6, \_\rangle$, 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 $\langle 5, 5, G\rangle$ is $|\langle 12, 9, T\rangle| + |\langle 10, 4, T\rangle| = 15$.

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 1. 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 $ref = ATA$ is $comp(AA, ref) = [\langle 0, 1, A\rangle]$ and $comp(AA, ref) = [\langle 2, 1, A\rangle]$.

Algorithm 1 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 [39]. 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, $[\langle 5, 5, G\rangle\langle 12, 5, G\rangle]$ is stored as $[\langle 5, 5, G\rangle\langle 1, 5, G\rangle]$, 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 strings show that the results of greedy compression algorithms are fairly close to non-greedy algorithms [40]. Note that Algorithm 1 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.

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 $sersize(s, ref)$ be the *serialized size* of $comp(s, ref)$. For a collection $S = \{s_1, ..., s_n\}$, let $sersize(S, ref) = \sum_{i \leq n} sersize(s_i, ref)$. The problem of *finding an optimal reference* for S is defined as follows: Find a reference $ref1$, such that there does not exist a reference $ref2$ with $sersize(S, ref2) < sersize(S, ref1)$.

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.

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*

Algorithm 2 Reference Selection RSbitX

Input: set of to-be-compressed sequences $s_1, ..., s_n$, set of candidate reference sequences $ref_1, ..., ref_m$, a base reference sequence ref_{base}, and a speedup value X
Output: index b for best reference

Compute $comp(ref_i, ref_{base})$ for all $1 \leq i \leq m$
for $1 \leq j \leq n$ **do**
  Split s_j into 1000 blocks $b_1, ..., b_{1000}$ of equal length
  Let sx_j be the concatenation of each X-th block of $b_1, ..., b_{1000}$[1]
end for
Compute $comp(sx_j, ref_{base})$ for all $1 \leq j \leq n$
for $1 \leq i \leq m$ **do**
  Let $val_i = 0$
  for $1 \leq j \leq n$ **do**
    $val_i = val_i + |rsim(comp(sx_j, ref_{base}), comp(ref_i, ref_{base}))|$
  end for
end for
Find the smallest val_{min} from $val_1, ..., val_m$ and let $b = min$

reference iff there does not exists a $j \neq i$ with $|comp(s, ref_j)| < |comp(s, ref_i)|$, 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 $n * m$ 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 1000 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 $rsim(rc_1, rc_2)$, is defined as $rsim(rc_1, rc_2) = |rc_1 \cup rc_2| - |rc_1 \cap rc_2|$.

Please note that a lower $rsim$-value indicates higher similarity. Two identical referential compressions will have a $rsim$-value of 0. We propose a heuristic for reference selection named RSbitX, which is shown in Algorithm 2. 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 $rsim$-computation, i.e. $comp(s_j, ref_{base})$ and $comp(ref_i, ref_{base})$ 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 1000 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 $m * n$ 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 $\frac{m*n}{m+\frac{n}{X}}$, 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 1000, then the gain in compression speed is lost. For our datasets 1000 blocks turned out to be a good compromise.

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 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 $(repl, p, c)$ is called a *replacement candidate* for a referential compression rc, if there exists two consecutive RME $[\langle p_1, l_1, c\rangle, \langle p_2, l_2, c_2\rangle] \in rc$ with $p_1 + l_1 + 1 = p_2 \wedge p = p_1 + l_1$. A tuple (ins, p, c) is called an *insert candidate* for a referential compression rc, if there exists two consecutive RME $[\langle p_1, l_1, c\rangle, \langle p_2, l_2, c_2\rangle] \in rc$ with $p_1 + l_1 = p_2 \wedge p = p_1 + l_1$. A tuple $(del, p, \_)$ is called a *deletion candidate* for a referential compression rc, if there exists two consecutive RME $[\langle p_1, l_1, c\rangle, \langle p_2, l_2, c_2\rangle] \in rc$ with $p_1 + l_1 + 2 = p_2 \wedge p = p_1 + l_1$. The *rewrite candidates* of a referential compression rc with respect to a reference ref, denoted $rewr(rc)$, 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

$$freq((X, p, c), S) = \frac{|\{rc_i \mid rc_i \in S \wedge (X, p, c) \in rewr(rc_i)\}|}{|S|}.$$

Given a position p, the most frequent rewrite candidate for p in S is (X, p, c), if there does not exist a $X^* \in \{repl, ins, del\}$ 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.

Algorithm 3 Reference Rewriting Algorithm

Input: set of referential compressions $S = \{rc_1, ..., rc_n\}$, a reference string ref, and a threshold t
Output: rewritten reference $result$

```
Let result be an empty string
for 1 ≤ p ≤ |ref| do
    if there exists a most frequent rewrite candidate (X, p, c) for p in S, with freq((X, p, c), S) ≥ t then
        if X=REPL then
            Append c to result
        else if X=INS then
            Append c to result
            Append ref(p) to result
        else if X=DEL then
            do nothing
        end if
    else
        Append ref(p) to result
    end if
end for
```

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

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

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 3. 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 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 $result$, 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 $result$. After the execution of the algorithm, $result$ 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 3 to Example 4 with threshold $t = 0.6$, we obtain the rewritten reference sequence $AAAACGCACAATCTGC$, since there exists only one rewrite candidate with a relative frequency larger than 0.6: rewrite candidate $(repl, 6, C)$. If we set $t = 0.8$, then the algorithm will not change the reference sequence at all. Please note that the rewrite candidate $(del, 6, \_)$ will never be used during

Algorithm 4 Second-order Compression Algorithm

Input: referentially compressed sequence rc and a set of referentially compressed sequences $REF = \{rcref_1, ..., rcref_n\}$
Output: second-order referential compression $rcso$

Let $rcso = \emptyset$
while $|rc| \neq 0$ **do**
  Let pre be the longest prefix of rc occurring in a referentially compressed sequence in REF, and let $rcref_m$ be the matching compressed reference and p be a position of an occurrence of pre in ref
  if $|pre| \geq 2$ **then**
    Add $\langle p, |pre| \rangle^{rcref_m}$ to the end of $rcso$
    Remove the first $|pre|$ referential match entries from rc
  else
    Add $rc(0)$ to the end of $rcso$
    Remove $rc(0)$ from rc
  end if
end while

the execution of the algorithm, independent from the threshold, since $(del, 6, \_)$ is dominated by $(repl, 6, C)$ 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 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.

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 [41]. 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:

$$rc_1 = [\langle 0, 4, T\rangle, \langle 5, 3, A\rangle, \langle 9, 4, T\rangle, \langle 15, 3, G\rangle]$$
$$rc_2 = [\langle 0, 4, A\rangle, \langle 5, 3, A\rangle, \langle 9, 4, T\rangle, \langle 15, 3, G\rangle]$$
$$rc_3 = [\langle 0, 4, T\rangle, \langle 5, 3, G\rangle, \langle 9, 4, T\rangle, \langle 15, 3, G\rangle]$$
$$rc_4 = [\langle 0, 4, T\rangle, \langle 5, 3, A\rangle, \langle 8, 3, T\rangle, \langle 15, 3, G\rangle]$$

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:

$$rc_2 = [\langle 0, 4, A\rangle, \langle 1, 3\rangle^{rc_1}]$$
$$rc_3 = [\langle 0, 4, T\rangle, \langle 5, 3, G\rangle, \langle 2, 3\rangle^{rc_1}]$$
$$rc_4 = [\langle 0, 2\rangle^{rc_1}, \langle 8, 3, T\rangle, \langle 15, 3, G\rangle],$$

where $\langle p, l\rangle^{rc_i}$ 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 4. It is very challenging to find an index structure for sets of compressed sequences for implementation of Algorithm 4. 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,

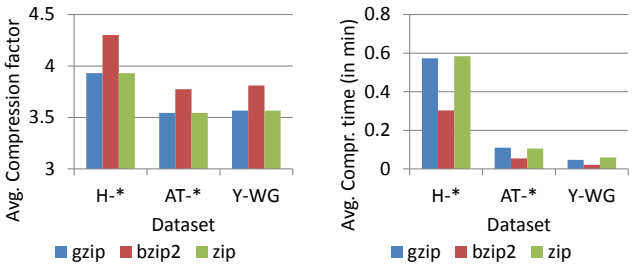

Fig. 2: Standard compression algorithms for five sequences.

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 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).

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 1092 genomes of the 1000 Genome project [3]. The 1000 Genome project group provides all sequenced genomes in Variant Call Format (VCF) [42] for download[2]. 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 1092 genomes. We use H-# to represent the set of all 1092 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 [43] from release GMI-Nordborg2010.[3]. 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 [44]. 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.

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 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).

6.2 Referential compression algorithms

We compare existing implementations of referential compression algorithms with FRESCO. The two competitors of FRESCO are GDC [17] and RLZ [28]. 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

2. ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20110521/

3. http://1001genomes.org/data/GMI/GMINordborg2010/releases/current/

	Compressed size (in MB)			Runtime (in s)			Compression factor			Compression speed (MB/s)		
Dataset	GDC	RLZ	FRESCO	GDC	RLZ	FRESCO	GDC	RLZ	FRESCO	GDC	RLZ	FRESCO
H-1	**3.7**	15.5	4.2	495.2	224.0	**20.0**	**680.0**	160.8	590.6	5.0	11.1	**124.3**
H-2	**3.9**	15.9	4.5	454.9	199.4	**19.4**	**625.5**	152.9	542.8	5.3	12.2	**125.5**
H-3	**3.3**	13.4	3.8	314.6	165.5	**14.9**	**593.6**	147.5	513.9	6.3	11.9	**132.4**
H-4	**3.5**	13.8	4.1	247.0	159.4	**15.0**	**543.8**	138.4	466.1	7.7	12.0	**127.1**
H-5	**3.0**	12.0	3.4	243.4	144.0	**13.9**	**608.2**	150.6	526.3	7.4	12.6	**130.2**
H-6	**3.0**	11.9	3.6	248.0	143.8	**15.3**	**566.1**	143.7	475.1	6.9	11.9	**112.0**
H-7	**2.7**	10.7	3.1	403.1	121.1	**12.8**	**591.2**	148.7	508.8	3.9	13.1	**124.7**
H-8	**2.5**	10.1	2.9	171.8	122.9	**11.6**	**577.5**	144.8	500.5	8.5	11.9	**126.3**
H-9	**2.0**	8.4	2.3	130.0	102.2	**11.0**	**714.3**	168.0	618.2	10.9	13.8	**128.8**
H-10	**2.4**	9.4	2.7	183.6	109.8	**10.9**	**572.2**	144.1	493.4	7.4	12.3	**124.7**
H-11	**2.5**	9.6	2.8	153.6	118.3	**11.0**	**548.3**	140.5	474.3	8.8	11.4	**122.2**
H-12	**2.3**	8.9	2.6	199.2	113.5	**10.0**	**593.0**	150.4	514.1	6.7	11.8	**133.5**
H-13	**1.9**	7.5	2.2	65.5	90.9	**9.2**	**602.5**	153.4	532.2	17.6	12.7	**124.5**
H-14	**1.6**	6.4	1.8	68.5	77.0	**8.6**	**664.7**	167.6	591.1	15.7	13.9	**124.2**
H-15	**1.4**	5.9	1.6	72.2	70.7	**8.1**	**710.1**	173.7	636.9	14.2	14.5	**126.9**
H-16	**1.4**	5.4	1.6	103.1	68.9	**6.9**	**638.5**	167.1	552.5	8.8	13.1	**131.4**
H-17	**1.3**	5.1	1.5	140.3	68.9	**6.5**	**635.3**	159.1	552.8	5.8	11.8	**125.4**
H-18	**1.4**	4.8	1.6	44.6	66.7	**6.6**	**565.2**	162.5	487.0	17.5	11.7	**118.3**
H-19	**1.1**	4.0	1.3	116.8	50.8	**5.3**	**546.7**	147.8	468.0	5.1	11.6	**111.1**
H-20	**1.0**	4.0	1.2	43.8	49.5	**4.5**	**623.7**	157.4	542.5	14.4	12.7	**139.3**
H-21	**0.7**	2.8	0.9	12.3	33.3	**3.5**	**684.3**	171.8	553.0	39.1	14.5	**138.2**
H-22	**0.6**	2.7	0.7	19.3	32.0	**3.7**	**816.9**	189.7	735.9	26.5	16.0	**137.1**
H-X	**1.7**	7.7	2.0	168.2	96.3	**12.1**	**903.6**	201.6	789.0	9.2	16.1	**128.0**
AT-1	**2.0**	6.5	2.3	8.3	41.3	**2.5**	**154.2**	105.3	133.2	36.7	7.4	**123.1**
AT-2	**1.4**	4.5	1.7	4.2	25.4	**1.4**	**145.0**	98.5	119.0	46.9	7.8	**136.8**
AT-3	**1.7**	5.5	2.0	5.5	32.1	**1.6**	**139.8**	96.0	117.2	42.7	7.3	**145.1**
AT-4	**1.3**	4.3	1.6	3.7	24.4	**1.5**	**139.5**	97.2	116.7	50.2	7.6	**126.5**
AT-5	**1.9**	6.1	2.2	6.3	37.5	**1.9**	**144.6**	99.5	121.3	42.8	7.2	**141.2**
Y-WG	**1.0**	86.8	1.4	2.8	47.6	**1.0**	**127.3**	1.4	89.0	44.5	2.6	**124.7**
AVG	**2.0**	10.7	2.3	142.4	90.9	**8.6**	**532.9**	142.8	460.7	18.0	11.5	**128.0**

Fig. 3: Compression statistics for 10 random sequences against a fixed reference (best values bold).

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 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 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.

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 reference 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

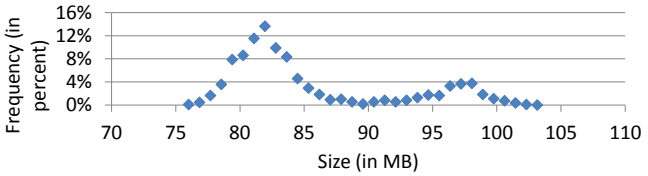

Fig. 4: Storage requirements for all against all for H-22.

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 [29] 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.

6.3 Reference Selection in FRESCO

In order to show the impact of the reference sequence on the compression ratio, we used each chromosome

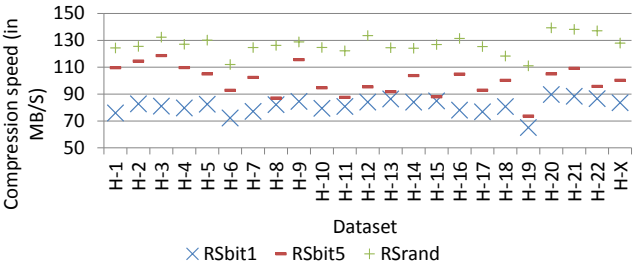

Fig. 5: Compression speed for selection heuristics.

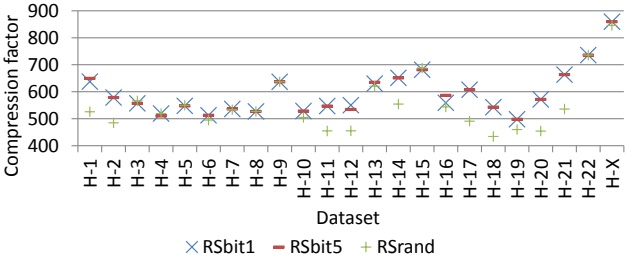

Fig. 6: Compression factors for selection heuristics.

Dataset	C. factor	Total time (s)	C. speed (MB/s)	C. factor increase
H-1	637.5	3,581.1	76.0	+7.3%
H-2	578.6	3,207.5	82.8	+5.5%
H-3	557.0	2,663.8	81.0	+6.2%
H-4	519.2	2,616.5	79.7	+8.7%
H-5	547.5	2,392.4	82.5	+0.3%
H-6	512.9	2,585.0	72.2	+3.3%
H-7	536.1	2,251.9	77.2	+3.8%
H-8	527.0	1,944.2	82.2	+2.7%
H-9	636.7	1,822.0	84.6	+1.4%
H-10	528.6	1,862.3	79.4	+4.4%
H-11	547.0	1,823.0	80.8	+11.2%
H-12	550.4	1,738.6	84.0	+5.7%
H-13	630.1	1,454.6	86.4	+14.8%
H-14	651.3	1,394.4	84.0	+7.6%
H-15	681.4	1,317.4	85.0	+5.5%
H-16	558.9	1,262.7	78.1	-1.3%
H-17	607.4	1,153.5	76.8	+7.6%
H-18	542.9	1,055.1	80.7	+9.9%
H-19	498.1	991.1	65.1	+0.8%
H-20	571.7	766.5	89.7	+3.6%
H-21	663.3	594.4	88.4	+12.8%
H-22	736.0	645.8	86.6	+3.1%
H-X	859.5	2,028.5	83.6	+8.7%
AT-1	138.4	112.2	48.8	+4.3%
AT-2	129.3	61.4	57.8	+7.8%
AT-3	120.8	70.9	59.6	0.0%
AT-4	120.8	60.5	55.3	+1.5%
AT-5	125.1	81.1	59.8	-0.3%
Y-WG	91.9	22.4	21.1	0.0%
AVG	496.7	1,433.1	74.8	+5.1%

Fig. 7: Selecting references in FRESCO.

in H-22 as a reference sequence, and referentially compressed all 1092 Chromosome 22 against the chosen reference. Figure 4 shows the the distribution of storage ranging from 97 MB to 124 MB. Thus, the choice of a reference sequence has a considerable impact on the compression factor. The exhaustive compression against all 1092 reference candidates of small human Chromosome 22, including index generation, took almost six days; this shows that the exhaustive computation is not feasible when dealing with large sets of complete genomes.

In Figure 5 and Figure 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 selec-

tion strategy. Base reference and candidate references were selected randomly; all results were averaged. In our experiments we compressed all 1092 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 7 we show the results of reference selection with respect to the base reference used before (in Figure 3). This time we have used the complete datasets for evaluation, e.g. all 1092 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.

6.4 Reference Rewriting in FRESCO

We analyse the impact of reference rewriting in Figure 8 with rewriting threshold as a parameter. The figure shows the impact of the threshold value on the storage requirements for 1092 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 9 we show the results of reference rewriting with respect to the base reference used before (in Figure 3). We have used complete datasets for evaluation, e.g. all 1092 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 recompression 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

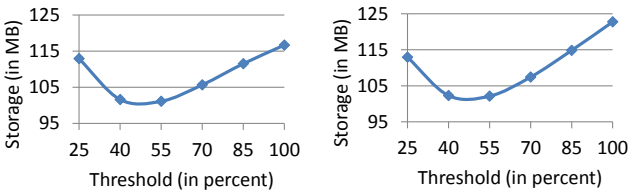

Fig. 8: Finding a threshold for rewriting sequences.

Dataset	C. factor	Total time (s)	C. speed (MB/s)	C. factor increase
H-1	804.3	3,334.8	81.6	+35.4%
H-2	736.4	3,033.3	87.5	+34.2%
H-3	697.6	2,520.7	85.6	+33.0%
H-4	651.0	2,340.8	89.1	+36.3%
H-5	704.9	2,138.6	92.3	+29.1%
H-6	643.7	2,311.6	80.8	+29.6%
H-7	675.1	1,994.4	87.1	+30.7%
H-8	674.3	1,737.2	92.0	+31.4%
H-9	834.1	1,612.7	95.5	+32.8%
H-10	676.1	1,655.1	89.4	+33.5%
H-11	673.7	1,659.9	88.8	+36.9%
H-12	698.2	1,586.9	92.1	+34.0%
H-13	765.9	1,350.8	93.0	+39.5%
H-14	806.1	1,266.1	92.5	+33.2%
H-15	864.1	1,190.6	94.0	+33.8%
H-16	753.6	1,024.3	96.2	+33.1%
H-17	729.8	1,030.2	86.0	+29.3%
H-18	671.2	946.6	90.0	+35.9%
H-19	619.8	846.5	76.2	+25.5%
H-20	703.1	670.3	102.6	+27.5%
H-21	769.0	508.2	103.4	+30.8%
H-22	904.5	548.3	102.0	+26.8%
H-X	1,018.0	1,993.8	85.0	+28.8%
AT-1	132.7	104.7	52.3	0.0%
AT-2	119.9	56.6	62.6	0.0%
AT-3	120.9	65.8	64.2	+0.1%
AT-4	119.0	56.1	59.6	0.0%
AT-5	125.5	75.8	64.1	0.0%
Y-WG	91.9	22.0	21.5	0.0%
AVG	613.3	1,299.4	83.0	+25.6%

Fig. 9: Rewriting references in FRESCO.

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 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.

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 10, the compression factors for 5-70 additional references are shown. Please note that we have only compressed sequences which are not included

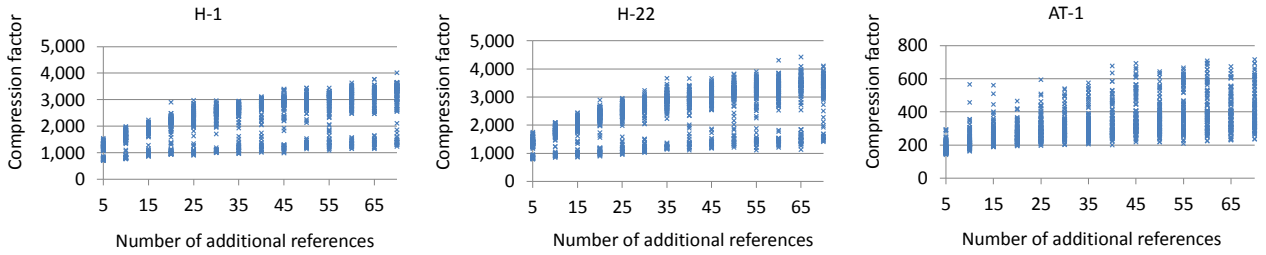

Fig. 10: Compression factors for randomly selected sequences with respect to 5-70 additional references

	Uncompressed			Zip			FRESCO		
Wikipedia article	Versions	Current (KB)	Total (KB)	Time (ms)	Size (KB)	CF	Time (ms)	Size (KB)	CF
Helsinki	2,664	406.8	584,178.0	18,419.1	115,970.0	5.0	**17,375.1**	**40,708.8**	**14.4**
Kardashev_scale	1,808	179.7	188,359.0	8,410.9	49,521.2	3.8	**8,260.7**	**23,517.8**	**8.0**
Fairy_chess_piece	495	217.7	84,015.5	2,171.1	10,954.5	7.7	**1,836.9**	**2,781.2**	**30.2**
United_States_Numbered_Highways	576	154.5	66,396.7	**2,488.4**	15,151.4	4.4	2,520.1	**6,705.8**	**9.9**
Vela_Incident	790	164.7	65,542.7	2,696.5	17,629.9	3.7	**1,805.3**	**6,020.3**	**10.9**
Fan_death	1,099	60.4	60,137.3	2,730.9	18,927.4	3.2	**2,165.3**	**7,593.9**	**7.9**
AVERAGE	1,238.7	197.3	174,771.5	6,152.8	38,025.7	4.6	5,660.6	14,554.6	13.5

Fig. 11: Zip and referential compression on all versions of different Wikipedia articles.

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 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.

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 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.

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. FRESCO was implemented in C++, using the BOOST library, CST [45], 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.

	GDC		RLZ		FRESCO		FRESCO (reference selection)		FRESCO (reference rewriting)		FRESCO (second-order compression)	
	CF	C.Speed	CF	C.Speed	CF	C.Speed	CF	C.Speed	CF	C.Speed	CF	C.Speed
H-*	635.0	11.2	158.4	12.8	550.7	126.8	594.7	81.2	742.4	90.6	3,057.2	58.4
AT-*	144.6	43.9	99.3	7.5	121.5	134.5	126.9	56.3	123.6	60.6	407.7	53.7
Y-WG	127.3	44.5	1.4	2.6	89.0	124.7	89.0	21.1	91.9	21.5	712.8	41.4
AVERAGE	302.3	33.2	86.4	7.6	253.7	128.7	270.2	52.8	319.3	57.5	1,392.6	51.1

Fig. 12: Summary of all techniques (CF=compression factor, C.speed=compression speed in MB/s)

The sequence interface defines two functions: one for loading a sequence from a file and 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 [28].

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 [17].

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) [17], and 3) compact binary encoding (COMPACT).

8 Conclusions

In Figure 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 [46].

References

[1] I. H. G. S. Consortium, "Initial sequencing and analysis of the human genome," *Nature*, vol. 409, no. 6822, pp. 860–921, February 2001.

[2] E. E. Schadt, S. Turner, and A. Kasarskis, "A window into third-generation sequencing." *Human molecular genetics*, vol. 19, no. R2, pp. R227–R240, Oct. 2010. [Online]. Available: http://dx.doi.org/10.1093/hmg/ddq416

[3] 1000 Genomes Project Consortium, "A map of human genome variation from population-scale sequencing," *Nature*, vol. 467, no. 7319, pp. 1061–1073, Oct. 2010. [Online]. Available: http://dx.doi.org/10.1038/nature09534

[4] Consortium ICG, "International network of cancer genome projects," *Nature*, vol. 464, no. 7291, pp. 993–998, Apr. 2010. [Online]. Available: http://dx.doi.org/10.1038/nature08987

[5] C. Brierley. (2010, Jun.) Press release for UK10K. [Online]. Available: http://www.wellcome.ac.uk/News/Media-office/Press-releases/2010/WTX060061.htm

[6] "International Cancer Genome Consortium Data Portal–a one-stop shop for cancer genomics data." *Database : the journal of biological databases and curation*, vol. 2011, no. 0, p. bar026, 2011. [Online]. Available: http://dx.doi.org/10.1093/database/bar026

[7] S. D. Kahn, "On the future of genomic data," *Science*, vol. 331, no. 6018, pp. 728–729, 2011. [Online]. Available: http://www.sciencemag.org/content/331/6018/728.abstract

[8] V. A. Fusaro, P. Patil, E. Gafni, D. P. Wall, and P. J. Tonellato, "Biomedical Cloud Computing With Amazon Web Services." *PLoS Computational Biology*, vol. 7, no. 8, pp. 1–6, 2011.

[9] "Cloud computing and the DNA data race." *Nature biotechnology*, vol. 28, no. 7, pp. 691–693, Jul. 2010. [Online]. Available: http://dx.doi.org/10.1038/nbt0710-691

[10] "The case for cloud computing in genome informatics." *Genome biology*, vol. 11, no. 5, pp. 207+, May 2010. [Online]. Available: http://dx.doi.org/10.1186/gb-2010-11-5-207

[11] O. Trelles, P. Prins, M. Snir, and R. C. Jansen, "Big data, but are we ready?" *Nature Reviews Genetics*, vol. 12, no. 3, p. 224, Feb. 2011. [Online]. Available: http://dx.doi.org/10.1038/nrg2857-c1

[12] E. Pennisi, "Will Computers Crash Genomics?" *Science*, vol. 331, no. 6018, pp. 666–668, Feb. 2011. [Online]. Available: http://dx.doi.org/10.1126/science.331.6018.666

[13] U. Nalbantoglu, D. J. Russell, and K. Sayood, "Data compression concepts and algorithms and their applications to bioinformatics," *Entropy*, vol. 12, no. 1, pp. 34–52, 2010. [Online]. Available: http://www.mdpi.com/1099-4300/12/1/34/

[14] D. Antoniou, E. Theodoridis, and A. Tsakalidis, "Compressing biological sequences using self adjusting data structures," in *Information Technology and Applications in Biomedicine*, 2010.

[15] D. Pratas and A. J. Pinho, "Compressing the human genome using exclusively Markov models." in *PACBB*, ser. Advances in Intelligent and Soft Computing, M. P. Rocha, J. M. C. Rodrguez, F. Fdez-Riverola, and A. Valencia, Eds., vol. 93. Springer, 2011, pp. 213–220. [Online]. Available: http://dblp.uni-trier.de/db/conf/pacbb/pacbb2011.html#PratasP11

[16] S. Christley, Y. Lu, C. Li, and X. Xie, "Human genomes as email attachments." *Bioinformatics (Oxford, England)*, vol. 25, no. 2, pp. 274–275, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btn582

[17] S. Deorowicz and S. Grabowski, "Robust Relative Compression of Genomes with Random Access," *Bioinformatics (Oxford, England)*, Sep. 2011. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btr505

[18] L. Chen, S. Lu, and J. Ram, "Compressed pattern matching in DNA sequences," in *Proceedings of the 2004 IEEE Computational Systems Bioinformatics Conference*, ser. CSB '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 62–68. [Online]. Available: http://dx.doi.org/10.1109/CSB.2004.59

[19] J. Larsson and A. Moffat, "Offline dictionary-based compression," in *Proceedings of the IEEE Data Compression Conference*, Mar. 1999, pp. 296–305.

[20] Y. Shibata, T. Matsumoto, M. Takeda *et al.*, "A Boyer-Moore type algorithm for compressed pattern matching," in *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, ser. COM '00. London, UK, UK: Springer-Verlag, 2000, pp. 181–194. [Online]. Available: http://dl.acm.org/citation.cfm?id=647819.736215

[21] S. Kuruppu, B. Beresford-Smith, T. Conway, and J. Zobel, "Iterative dictionary construction for compression of large dna data sets," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 9, no. 1, pp. 137–149, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1109/TCBB.2011.82

[22] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *Information Theory, IEEE Transactions on*, vol. 23, no. 3, pp. 337–343, 1977.

[23] J. G. Cleary, Ian, and I. H. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Transactions on Communications*, vol. 32, pp. 396–402, 1984.

[24] M. Duc Cao, T. I. Dix, L. Allison, and C. Mears, "A simple statistical algorithm for biological sequence compression," in *Proceedings of the 2007 Data Compression Conference*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 43–52. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251981.1252877

[25] G. V. Cormack and R. N. S. Horspool, "Data compression using dynamic Markov modelling," *Comput. J.*, vol. 30, pp. 541–550, December 1987. [Online]. Available: http://dl.acm.org/citation.cfm?id=44665.44675

[26] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, pp. 1098–1101, September 1952.

[27] S. Kuruppu, S. J. Puglisi, and J. Zobel, "Relative lempel-ziv compression of genomes for large-scale storage and retrieval," in *Proceedings of the 17th international conference on String processing and information retrieval*, ser. SPIRE'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 201–206. [Online]. Available: http://dl.acm.org/citation.cfm?id=1928328.1928353

[28] S. Kuruppu, S. Puglisi, and J. Zobel, "Optimized relative lempel-ziv compression of genomes," in *Australasian Computer Science Conference*, 2011.

[29] A. J. Pinho, D. Pratas, and S. P. Garcia, "Green: a tool for efficient compression of genome resequencing data," *Nucleic Acids Research*, Dec. 2011. [Online]. Available: http://dx.doi.org/10.1093/nar/gkr1124

[30] S. Kreft and G. Navarro, "Lz77-like compression with fast random access," in *Proceedings of the 2010 Data Compression Conference*, ser. DCC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 239–248. [Online]. Available: http://dx.doi.org/10.1109/DCC.2010.29

[31] M. H. Fritz, R. Leinonen, G. Cochrane, and E. Birney, "Efficient storage of high throughput DNA sequencing data using reference-based compression," *Genome Research*, vol. 21, no. 5, pp. 734–740, May 2011. [Online]. Available: http://dx.doi.org/10.1101/gr.114819.110

[32] C. Wang and D. Zhang, "A novel compression tool for efficient storage of genome resequencing data," *Nucleic Acids Research*, vol. 39, no. 7, p. e45, Apr. 2011. [Online]. Available: http://dx.doi.org/10.1093/nar/gkr009

[33] V. Bhola, A. S. Bopardikar, R. Narayanan, K. Lee, and T. Ahn, "No-reference compression of genomic data stored in fastq format," in *BIBM*, 2011, pp. 147–150.

[34] R. Wan, V. N. Anh, and K. Asai, "Transformations for the compression of FASTQ quality scores of next generation sequencing data," *Bioinformatics*, Dec. 2011. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btr689

[35] G. Menconi, V. Benci, and M. Buiatti, "Data compression and genomes: a two-dimensional life domain map." *Journal of Theoretical Biology*, vol. 253, no. 2, pp. 281–288, 2008. [Online]. Available: http://arxiv.org/abs/0803.0465

[36] K. Daily, P. Rigor, S. Christley, X. Xie, and P. Baldi, "Data structures and compression algorithms for high-throughput sequencing technologies." *BMC bioinformatics*, vol. 11, no. 1, pp. 514+, 2010. [Online]. Available: http://dx.doi.org/10.1186/1471-2105-11-514

[37] C. Kozanitis, C. Saunders, S. Kruglyak, V. Bafna, and G. Varghese, "Compressing genomic sequence fragments using slimgene." in *RECOMB'10*, 2010, pp. 310–324.

[38] S. Wandelt and U. Leser, "Adaptive efficient compression of genomes," *Algorithms for Molecular Biology*, vol. 7, p. 30, 2012.

[39] M. Cohn and R. Khazan, "Parsing with prefix and suffix dictionaries," in *Data Compression Conference*, 1996, pp. 180–189.

[40] R. N. Horspool, "The effect of non-greedy parsing in ziv-lempel compression methods," in *Proceedings of the Conference on Data Compression*, ser. DCC '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 302–. [Online]. Available: http://dl.acm.org/citation.cfm?id=874051.874758

[41] S. Grabowski and S. Deorowicz, "Engineering relative compression of genomes," *CoRR*, vol. abs/1103.2351, 2011.

[42] P. Danecek, A. Auton, G. Abecasis, and 1000 Genomes Project Analysis Group, "The variant call format and VCFtools." *Bioinformatics (Oxford, England)*, vol. 27, no. 15, pp. 2156–2158, Aug. 2011. [Online]. Available: http://dx.doi.org/10.1093/bioinformatics/btr330

[43] "Whole-genome sequencing of multiple Arabidopsis thaliana populations," *Nature Genetics*, vol. 43, no. 10, pp. 956–963, Aug. 2011. [Online]. Available: http://dx.doi.org/10.1038/ng.911

[44] "Overview of the yeast genome," *Nature*, vol. 387, no. 6632 Suppl, pp. 7–65, May 1997. [Online]. Available: http://www.nature.com/doifinder/10.1038/42755

[45] E. Ohlebusch, J. Fischer, and S. Gog, "Cst++," in *SPIRE'10*, 2010, pp. 322–333.

[46] S. Wandelt and U. Leser, "String searching in referentially compressed genomes," in *IC3K: International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management 2012*. SciTePress, 2012.

Sebastian Wandelt works as a postdoc in the group Knowledge Management in Bioinformatics at Humboldt-University of Berlin. He received a PhD degree in computer science from Hamburg University of Technology. His research interests are compression of genome data and searching large collections of sequences.

Ulf Leser is a professor at Humboldt-University of Berlin and heads the group Knowledge Management in Bioinformatics . He has a Ph.D. from the Technical University Berlin and the Graduate School for "Distributed Information Systems".