Trends in Genome Compression

Sebastian Wandelt, Marc Bux, and Ulf Leser*

Knowledge Management in Bioinformatics,
Institute for Computer Science,
Humboldt-Universität zu Berlin, Germany

February 24, 2014

Abstract

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 is feasible even for smaller project or organizations already today. However, generating the sequence is only one issue; another one is storing, managing, and analyzing it. These tasks become more and more challenging due to the sheer size of the data sets and are increasingly considered to be the most severe bottlenecks in larger genome projects. One possible countermeasure is to compress the data; compression reduces costs in terms of requiring less hard disk storage and in terms of requiring less bandwidth if data is shipped to large compute clusters for parallel analysis. Accordingly, sequence compression has recently attracted much interest in the scientific community. In this paper, we explain the different basic techniques for sequence compression, point to distinctions 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 for future systems.

Keywords: genome compression, read compression, survey

Key messages of the article:

- Overview of trends in genome compression: bit manipulation, dictionary-based, statistical, and referential compression schemes
- Wide ranges of compression performance (compression ratio, compression time, memory requirements)
- Comparing compression schemes for evaluation purposes is difficult
- Many minor improvements in recent contributions
- Lack of widely accepted and utilized benchmark data

*E-Mail addresses: {wandelt,bux,leser}@informatik.hu-berlin.de

1 Introduction

The introduction of high-throughput sequencing has led to the generation of 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 [1], for several reasons: First, the decreasing sequencing 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 further 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 [2, 3]. 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 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 30 TB 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 [4]. 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 600 TB already is much more difficult than managing 30 TB, 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 [5].

Another aspect of data size is monetary cost. As pointed out by Stein [6], the cost for sequencing is decreasing much faster than the cost for storing sequences. As a consequence, storing sequences (i.e., buying hard disks) 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 paper, is compression. Compression is considered one key technology for data management of population-scale genome analysis[7]. We use the term compression to denote methods for storing the information encoded in the original data in less space, possibly losing information (see below). There are various methods for achieving this goal. Early approaches focused mainly on bit manipulation techniques, i.e., they packed

more bases into one byte of data. These methods 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 by orders of magnitude higher than that of previous attempts. The field is highly active; 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 for particular problems (such as compression of bacterial genomes or compressing only coding sequences). Another important difference that sometimes is not obviously reflected in published results is the distinction between compressing genomes and compressing reads.

In this paper, we survey recent algorithms for all these classes of problems. In Section 2, we first explain the four main classes of compression techniques, i.e., bit manipulation, dictionary-based, statistical, and referential compression. We then discuss concrete systems for compressing entire genomes class by class in Section 3. In contrast, in Section 4, we review recent contributions to the compression of sequence reads, which involves the treatment of quality scores. The paper is concluded in Section 5 with potential directions for future research in the area of genome compression.

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, and referential approaches.

- **Naive bit encoding** algorithms exploit fixed-length encodings of two or more symbols in a single byte [8, 9].
- **Dictionary-based** or substitutional compression algorithms replace repeated substrings by references to a dictionary (i.e., a set of previously seen or predefined common strings), which is built at runtime or offline [10, 11].
- **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 [12, 13].
- **Referential** or reference-based approaches 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 the 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

Table 1: A comparison of standard compression schemes.

Name	Usual compression rate
Naive bit manipulation algorithms	2:1 – 6:1
Dictionary-based algorithms	4:1 – 6:1
Statistical algorithms	4:1 – 8:1
Referential algorithms	1:1 – 400:1

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 for instance be 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 long subsequent occurrences of the same symbol, e.g., N, in sequences.

In the remaining part of the paper, we only discuss lossless compression algorithms, as in most biomedical applications every single base is important. Lossy compression is more suitable for other applications, such as image compression (e.g., the JPEG standard [14]). Further, we do not discuss the compression of protein sequences which is usually considered to be more complex due to the larger alphabet size and the fact that repeats are less frequent [15]. For some work on compressing protein sequences, see [16].

In the following subsections, we will discuss all these base techniques in detail. This discussion builds the foundation for evaluating recent systems in Section 3 and Section 4. Table 1 summarizes the compression rates for state of the art compression schemes.

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 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 1. Each symbol in the input is replaced by two bits using the replacement $\{A \rightarrow 00, C \rightarrow 01, G \rightarrow 10, T \rightarrow 11\}$.

Current processor architectures provide highly improved bit operations, basically allowing 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

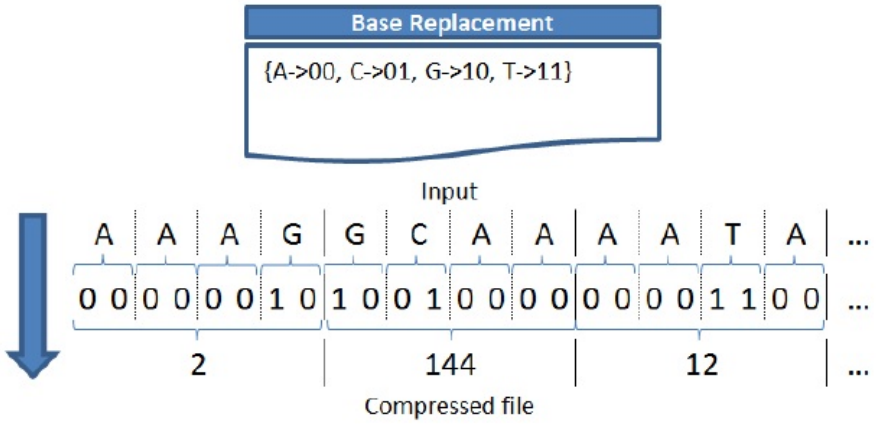

Figure 1: Example for naive bit encoding

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 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.

Lempel-Ziv-based compression algorithms, such as LZ77 or LZ78, are prominent example of dictionary-based algorithms [17]. In those algorithms, the input sequence is parsed sequentially and examined for reoccurring substrings. 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 differences is the concrete method used to encode repeated occurrences of substrings with dictionary indices. This touches

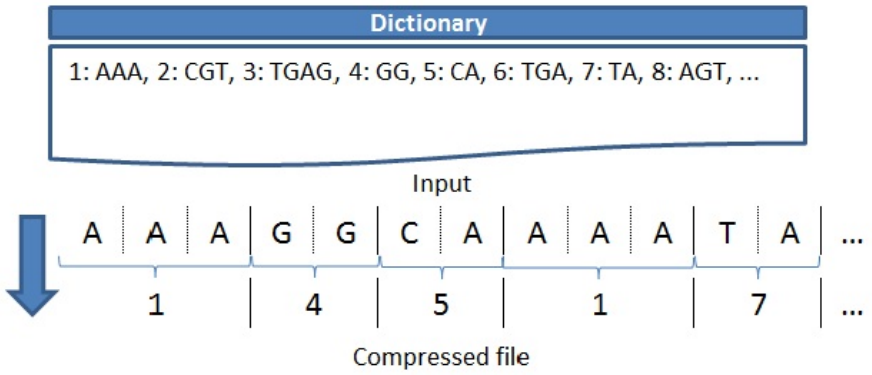

Figure 2: Example for dictionary-based algorithms

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 [18], which encode small numbers more efficiently than large numbers, and Fibonacci codes [19], which are more tolerant to failures. Using such codes, current methods for dictionary-compression (see Table 2) reach compression rates between 4:1 and 6:1 depending on the frequency of repeats in the genomes being compressed.

2.3 Statistical Algorithms

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

One of the most commonly used and best understood statistical encodings is Huffman encoding [20]. It uses a variable-length code table derived from estimated probabilities for the occurrence of each possible symbol. A binary tree is created in which leaf nodes correspond to symbols and edges are labeled 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. 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 [21]. It is therefore considered not to be ideal for efficient compression of DNA sequences [22]. One example for a statistical algorithm is shown in Figure 3. In this example, the frequently 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 [23] encodes longer strings – or even whole

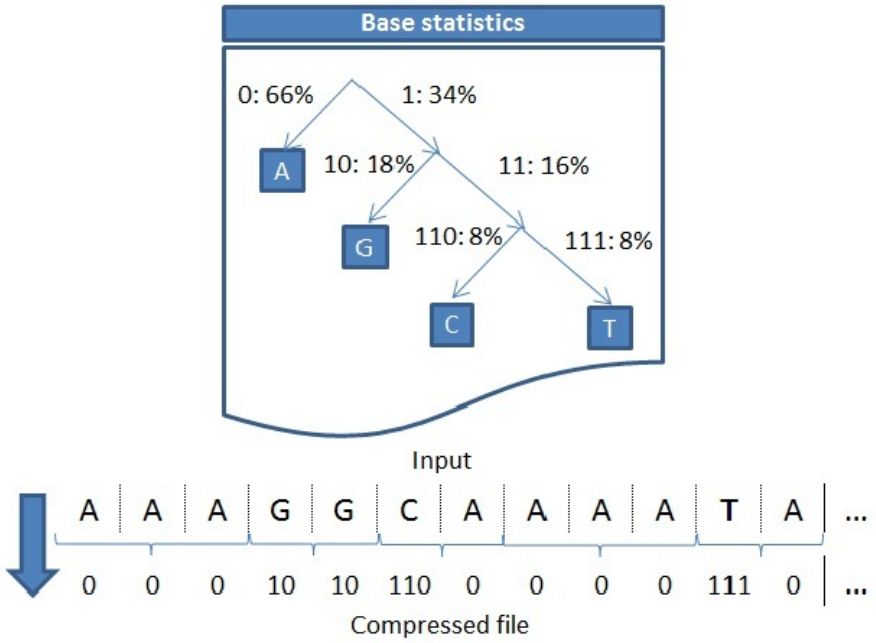

Figure 3: Example for statistical algorithms

input streams – as a single number between zero and one. This allows for higher compression rates in applications with small alphabets. Many concrete implementations make use of range encoding instead of arithmetic encoding, since it is believed that the former is less encumbered by patents.

[24] 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-based approaches by the order of the 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 1). The compression rate depends mainly on the distribution of input symbols and the available memory for construction of frequency distributions.

2.4 Referential Algorithms

While genome research was focusing on sequencing new genomes for a long time, recent advances in sequencing technology [25] and increasing demands from areas such as translational medicine [26] have made the resequencing of genomes – which means sequencing different individuals of the same species – more and more popular. Large international projects, such as the ICGC [3] already plan to sequence thousands of human genomes. Since all resequenced 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

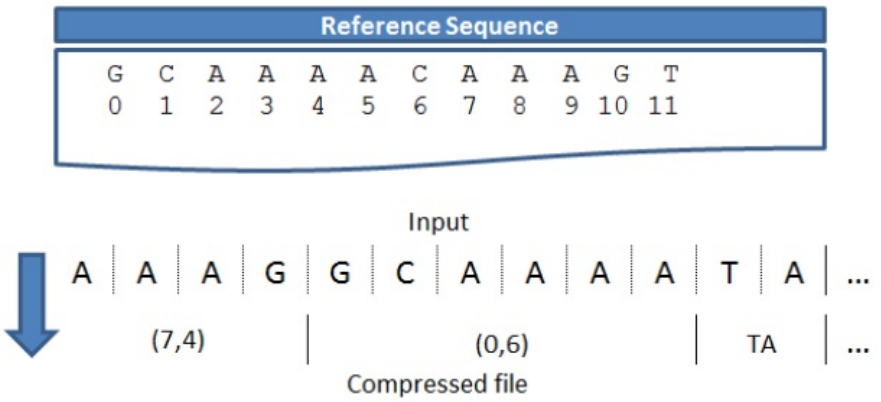

Figure 4: Example for referential algorithms

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[27, 28]. Long matches in the reference are usually found by using index structures, e.g., hash-based structures or suffix trees. The general algorithm for referential compression in pseudocode is shown in Algorithm 1. The value of X determines 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:    find longest matching substring in reference for current input position
3:    if length of match > X then
4:       encode match as (matchposition, length)
5:    else
6:       encode match with raw symbols
7:    end if
8: end while
```

One example for a referential compression algorithm is shown in Figure 4. In this example, there are 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. In highly similar sequences, long stretches of DNA are identical, interrupted mostly by SNPs and short INDELs. If this is the 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 are able to reach the 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 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 the 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 metagenomics, where sequences are sampled at random from a large set of species [29]. 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 subsequence 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 perfect 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 of 400:1 and better are possible (see Table 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 stated in a uniform way, e.g., without including the reference sequence.

3 Whole Genome Compression

In this section we discuss concrete algorithms for compressing whole genomes. Compression of reads will be covered in Section 4. Table 2 summarizes compression rates and other properties of the non-referential compression schemes we will discuss, while Table 3 contains similar information for referential schemes.

3.1 Naive Bit Manipulation Algorithms

[30] 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 presented in [31]. Here, run-length encoding is implemented on top of naive 2-bit compression.

[32] 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 builds on an Oracle database [33]. [34] additionally incorporates an algorithm for multi-threaded search in the compressed data. [35] describes a particular run-length encoding scheme. Finally, [36] focuses on the analysis of how to store repeats with variable-size codes.

3.2 Dictionary-based Algorithms

Comrad, proposed in [37], 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.

[38] 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 than symbols in leaf nodes.

In [39], the problem of a non-uniform distribution of DNA over the genome is addressed by splitting the input into blocks and encoding each block separately. A hash table-based approach is used to detect repeats.

3.3 Statistical Algorithms

[40] 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. [41] propose another approach based on combining different Markov models. The authors use six different Markov models of orders 1, 4, 6, 10, 14, and 16.

Gene-Compressor was proposed in [42] 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 [43], 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.

[44] 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.

3.4 Referential Algorithms

[45] propose 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, [46] 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 [47] 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 the 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 [48], 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 [49], RLZopt is presented as an extension of RLZ. The key aspect is longest increasing subsequence computation, which 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 [50]. 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 [49, 50] is taken from the set of input sequences.

GReEn, an expert-based reference compression scheme was recently presented in [51]. 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 [52], another LZ77-style compression scheme with random access [53], and approaches based on permanent index structures [54, 55]. The important question of choosing a good reference sequence is discussed in [56] and [57]. 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.

[58] 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.

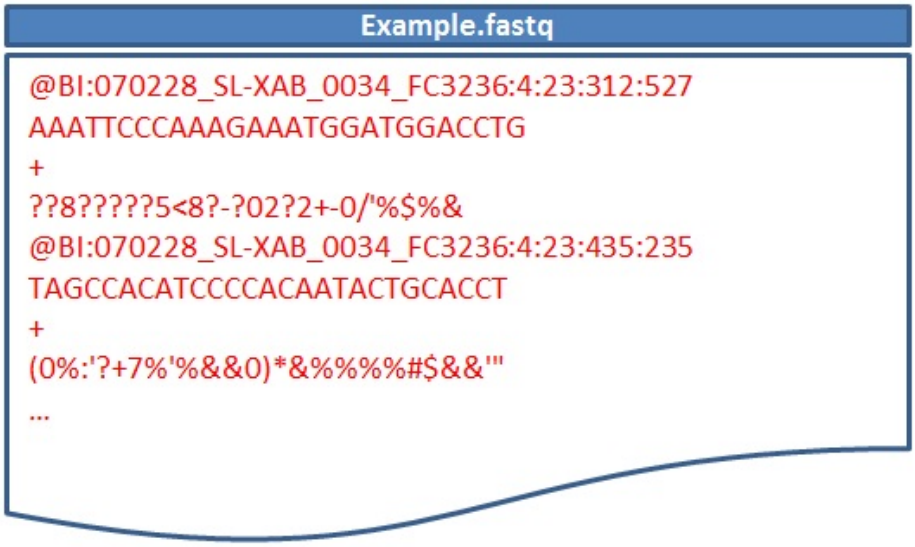

Figure 5: Example for two reads in FASTQ

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 [59]. However, the original reads are usually kept, for instance to allow re-alignment when new and more accurate references become available. For this reason, compressing read sets is an equally important topic.

Besides the fact that reads are typically short, may map anywhere on a genome, and are associated to an 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 denotes the probability of the base actually being at this position and is therefore an indicator for the likelihood 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, in which scores can have 94 different values. An exemplary set of reads in the form of a FASTQ file is given in Figure 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 rates 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 algorithms do not necessarily need the full quality information, which makes lossy compression more attractive. Table 4 gives an overview of compression rates for recent algorithms in non-referential sequence read compression, while Table 5 compares compression rates for referential compression schemes.

4.1 Dictionary-based Algorithms

A dictionary-based approach based on metasymbols was proposed in [60]. 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.

[61] developed CASToRe, a modification of Lempel-Ziv compression [17]. 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.

[62] 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 as well as an adaptive and intelligent single particle optimizer.

4.2 Statistical Algorithms

[63] developed 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, [64] 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 cost, while losing only little information.

The importance of quality information was also addressed in [65] in the 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.

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

In [66], 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 a data management component and a graphical user interface.

4.3 Referential Algorithms

Since aligning reads to a reference genome constitutes the first step in most analysis pipelines, usage of referential compression schemes is a straight-forward approach towards read compression. However, mapping a read against a genome for referential compression is very different from mapping it for further analysis: First, algorithms for the former are free to choose any mapping (and require only 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 may ignore these scores during the alignment.

[67] 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 schemes, 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, [68] 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.

[69] 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.

5 Discussion and Conclusions

In this paper, 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 paper are based on the original papers. Clearly, tools should ideally 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 are often rather inconclusive, as they only compare against very few other approaches and only use 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 that 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 sequences. 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 [70]. Another interesting – yet to our knowledge completely unexplored – research question is the integration of sequence compression in scientific workflows, which we consider as a pre-requisite to fully leverage the advanced computing power of cloud infrastructures.

References

[1] Scott D. Kahn. On the future of genomic data. *Science*, 331(6018):728–729, 2011.

[2] Marc Via, Christopher Gignoux, and Esteban González G. Burchard. The 1000 Genomes Project: new opportunities for research and social challenges. *Genome Medicine*, 2(1):3+, 2010.

[3] Thomas J. Hudson, Warwick Anderson, Axel Artez, et al. International network of cancer genome projects. *Nature*, 464(7291):993–998, 2010.

[4] Oswaldo Trelles, Pjotr Prins, Marc Snir, et al. Big data, but are we ready? *Nature Reviews Genetics*, 12(3):224, 2011.

[5] Eric E. Schadt, Steve Turner, and Andrew Kasarskis. A window into third-generation sequencing. *Human Molecular Genetics*, 19(R2):R227–R240, 2010.

[6] Lincoln Stein. The case for cloud computing in genome informatics. *Genome Biology*, 11(5):207, 2010.

[7] Sebastian Wandelt, Astrid Rheinländer, Marc Bux, Lisa Thalheim, Berit Haldemann, and Ulf Leser. Data management challenges in next generation sequencing. *Datenbank-Spektrum*, 12(3):161–171, 2012.

[8] Stéphane Grumbach and Fariza Tahi. A new challenge for compression algorithms: genetic sequences. *Information Processing & Management*, 30(6):875–886, 1994.

[9] Lei Chen, Shiyong Lu, and Jeffrey Ram. Compressed pattern matching in dna sequences. In *Proceedings of the 2004 IEEE Computational Systems Bioinformatics Conference*, CSB'04, pages 62–68, 2004.

[10] Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Proceedings of the 1999 Conference on Data Compression*, DCC'99, pages 296–305, 1999.

[11] Yusuke Shibata, Tetsuya Matsumoto, Masayuki Takeda, et al. A boyer-moore type algorithm for compressed pattern matching. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, COM'00, pages 181–194, 2000.

[12] John G. Cleary, Ian, and Ian H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32:396–402, 1984.

[13] Minh Duc Cao, T.I. Dix, L. Allison, and C. Mears. A simple statistical algorithm for biological sequence compression. In *Proceedings of the 2007 Conference on Data Compression*, DCC'07, pages 43–52, 2007.

[14] William B. Pennebaker and Joan L. Mitchell. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, New York, USA, 1992.

[15] Craig G. Nevill-Manning and Ian H. Witten. Protein is incompressible. In *Proceedings of the 1999 Conference on Data Compression*, DCC'99, page 257, 1999.

[16] Andrea Hategan and Ioan Tabus. Protein is compressible. In *Proceedings of the 6th Nordic Signal Processing Symposium*, NORSIG 2004, pages 192–195, 2004.

[17] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[18] Solomon W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 12:399–401, 1966.

[19] Shmuel Tomi Klein and Miri Kopel Ben-Nissan. On the usefulness of fibonacci compression codes. *The Computer Journal*, 53(6):701–716, 2010.

[20] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, 1952.

[21] A. Bookstein and S.T. Klein. Is huffman coding dead? *Computing*, 50:279–296, 1993.

[22] Toshiko Matsumoto, Kunihiko Sadakane, Imai Hiroshi, et al. *Currents in Computational Molecular Biology*, pages 76–77. Universal Academy Press Inc., 2000.

[23] Alistair Moffat, Radford Neal, and Ian Witten. Arithmetic coding revisited. In *Proceedings of the 1995 Conference on Data Compression*, DCC'95, pages 202–294, 1995.

[24] Gordon Cormack and Nigel Horspool. Data compression using dynamic markov modelling. *Comput. J.*, 30:541–550, 1987.

[25] Michael Schatz, Ben Langmead, and Steven Salzberg. Cloud computing and the dna data race. *Nature Biotechnology*, 28(7):691, 2010.

[26] Lynda Chin, Jannik N. Andersen, and P. Andrew Futreal. Cancer genomics: from discovery science to personalized medicine. *Nature Medicine*, 17(3):297–303, 2011.

[27] Sebastian Wandelt and Ulf Leser. Adaptive efficient compression of genomes. *Algorithms for Molecular Biology*, 7:30, 2012.

[28] Sebastian Deorowicz and Szymon Grabowski. Compression of dna sequence reads in fastq format. *Bioinformatics*, 27(6):860–862, 2011.

[29] Patrick Schloss and Jo Handelsman. Biotechnological prospects from metagenomics. *Current Opinion in Biotechnology*, 14(3):303–310, 2003.

[30] Gregory Vey. Differential direct coding: a compression algorithm for nucleotide sequence data. *The Journal of Biological Databases and Curation*, 2009.

[31] P. Raja Rajeswari, Allam Apparo, and V. K. Kumar. Genbit compress tool(gbc): A java-based tool to compress dna sequences and compute compression ratio(bits/base) of genomes. *CoRR*, abs/1006.1193, 2010.

[32] Rajendra Kumar Bharti, Archana Verma, and R.K. Singh. A biological sequence compression based on cross chromosomal similarities using variable length lut. *International Journal of Biometrics and Bioinformatics*, 4:217–223, 2011.

[33] Ateet Mehta and Bankim Patel. Dna compression using hash based data structure. *International Journal of Information Technology & Knowledge Management*, 3:383–386, 2010.

[34] Piyuan Lin, Shaopeng Liu, Lixia Zhang, et al. Compressed pattern matching in dna sequences using multithreaded technology. In *3rd International Conference on Bioinformatics and Biomedical Engineering*, ICBBE'09, 2009.

[35] Kamta Nath Mishra, Anupam Aaggarwal, Edries Abdelhadi, et al. An efficient horizontal and vertical method for online dna sequence compression. *International Journal of Computer Applications*, 3(1):39–46, 2010.

[36] Pothuraju Rajeswari and Allam Apparao. Dnabit compress – genome compression algorithm. *Bioinformation*, 5(8):350–60, 2011.

[37] Shanika Kuruppu, Bryan Beresford-Smith, Thomas Conway, et al. Iterative dictionary construction for compression of large dna data sets. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 9(1):137–149, 2012.

[38] Dimitris Antoniou, Evangelos Theodoridis, and Athanasios Tsakalidis. Compressing biological sequences using self adjusting data structures. In *Information Technology and Applications in Biomedicine*, 2010.

[39] Kalyan Kumar Kaipa, Ajit S Bopardikar, Srikantha Abhilash, et al. Algorithm for dna sequence compression based on prediction of mismatch bases and repeat location. In *Bioinformatics and Biomedicine Workshops*, BIBMW, 2010.

[40] Minh Duc Cao, Trevor I. Dix, Lloyd Allison, et al. A simple statistical algorithm for biological sequence compression. In *Proceedings of the 2007 Conference on Data Compression*, DCC'07, pages 43–52, 2007.

[41] Diogo Pratas and Armando J. Pinho. Compressing the human genome using exclusively markov models. In Miguel P. Rocha, Juan M. Corchado Rodrguez, Florentino Fdez-Riverola, and Alfonso Valencia, editors, *PACBB*, volume 93 of *Advances in Intelligent and Soft Computing*, pages 213–220. Springer, 2011.

[42] K. R. Venugopal, K. G. Srinivasa, and Lalit Patnaik. *Probabilistic Approach for DNA Compression*, chapter 14, pages 279–289. Springer, 2009.

[43] I. Tabus and G. Korodi. Genome compression using normalized maximum likelihood models for constrained markov sources. In *Information Theory Workshop*, 2008.

[44] Kalyan Kumar Kaipa, Kyusang Lee, Taejin Ahn, et al. System for random access dna sequence compression. In *International Conference on Bioinformatics and Biomedicine Workshops*, 2010.

[45] Marty C. Brandon, Douglas C. Wallace, and Pierre Baldi. Data structures and compression algorithms for genomic sequence data. *Bioinformatics*, 25(14):1731–1738, 2009.

[46] Scott Christley, Yiming Lu, Chen Li, et al. Human genomes as email attachments. *Bioinformatics*, 25(2):274–275, 2009.

[47] Congmao Wang and Dabing Zhang. A novel compression tool for efficient storage of genome resequencing data. *Nucleic Acids Research*, 39(7):e45, 2011.

[48] Shanika Kuruppu, Simon J. Puglisi, and Justin 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*, SPIRE'10, pages 201–206, 2010.

[49] Shanika Kuruppu, Simon Puglisi, and Justin Zobel. Optimized relative lempel-ziv compression of genomes. In *Australasian Computer Science Conference*, 2011.

[50] Szymon Grabowski and Sebastian Deorowicz. Engineering relative compression of genomes. *CoRR*, abs/1103.2351, 2011.

[51] Armando J. Pinho, Diogo Pratas, and Sara P. Garcia. Green: a tool for efficient compression of genome resequencing data. *Nucleic Acids Research*, 2011.

[52] Hyoung Do Kim and Ju-Han Kim. Dna data compression based on the whole genome sequence. *Journal of Convergence Information Technology*, 4(3):82–85, 2009.

[53] Sebastian Kreft and Gonzalo Navarro. Lz77-like compression with fast random access. In *Proceedings of the 2010 Conference on Data Compression*, DCC'10, pages 239–248, 2010.

[54] Andrew Peel, Anthony Wirth, and Justin Zobel. Collection-based compression using discovered long matching strings. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM'11, pages 2361–2364, 2011.

[55] Pragya Pande and Dhruv Matani. Compressing the human genome against a reference. Technical report, Stony Brook University, 2011.

[56] Heba Afify, Muhammad Islam, and Manal Abdel Wahed. Dna lossless differential compression algorithm based on similarity of genomic sequence database. *CoRR*, abs/1109.0094, 2011.

[57] Heba Afify, Muhammad Islam, and Manal Abdel Wahed. Genomic sequences differential compression model. *International Journal of Computer Science and Information Technology*, 3:145–154, 2011.

[58] Shanika Kuruppu, Simon Puglisi, and Justin Zobel. Reference sequence construction for relative compression of genomes. In *Proceedings of the 18th International Conference on String Processing and Information Retrieval*, SPIRE'11, pages 420–425, 2011.

[59] William Brockman, Pablo Alvarez, Sarah Young, et al. Quality scores and SNP detection in sequencing-by-synthesis systems. *Genome Research*, 18(5):763–770, 2008.

[60] Oscar Herrera and Angel Kuri-Morales. Lossless compression of biological sequences with evolutionary metadictionaries. In *Workshop on Machine Learning and Data Mining*, 2009.

[61] Giulia Menconi, Vieri Benci, and Marcello Buiatti. Data compression and genomes: a two-dimensional life domain map. *Journal of Theoretical Biology*, 253(2):281–288, 2008.

[62] Zexuan Zhu, Jiarui Zhou, Zhen Ji, et al. Dna sequence compression using adaptive particle swarm optimization-based memetic algorithm. *IEEE Transactions on Evolutionary Computation*, 15(5):643–658, 2011.

[63] Vishal Bhola, Ajit Bopardikar, Rangavittal Narayanan, et al. No-reference compression of genomic data stored in fastq format. In *Proceedings of the 2011 IEEE International Conference on Bioinformatics and Biomedicine*, BIBM'11, pages 147–150, 2011.

[64] Raymond Wan, Vo N. Anh, and Kiyoshi Asai. Transformations for the compression of fastq quality scores of next generation sequencing data. *Bioinformatics*, 2011.

[65] Waibhav Tembe, James Lowey, and Edward Suh. G-sqz: compact encoding of genomic sequence and quality data. *Bioinformatics*, 26(17):2192–2194, 2010.

[66] Wei-Hsin Chen, Yu-Wen Lu, Feipei Lai, et al. Integrating human genome database into electronic health record with sequence alignment and compression mechanism. *Journal of Medical Systems*, 36(3):2587–2597, 2011.

[67] Kenny Daily, Paul Rigor, Scott Christley, et al. Data structures and compression algorithms for high-throughput sequencing technologies. *BMC Bioinformatics*, 11(1):514+, 2010.

[68] Christos Kozanitis, Chris Saunders, Semyon Kruglyak, et al. Compressing genomic sequence fragments using slimgene. In *Proceedings of the 14th Annual International Conference on Research in Computational Molecular Biology*, RECOMB'10, pages 310–324, 2010.

[69] Markus H. Fritz, Rasko Leinonen, Guy Cochrane, et al. Efficient storage of high throughput dna sequencing data using reference-based compression. *Genome Research*, 21(5):734–740, 2011.

[70] Sebastian Wandelt and Ulf Leser. String searching in referentially compressed genomes. In *IC3K: International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*, 2012.

6 Large Tables

Table 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.

– Naive Bit Manipulation Algorithms –

Tool	Tested with	Data used	Compression ratio	Compression speed (MB/s)
[32]	–	genes	up to 5.7:1	–
[33]	–	–	4:1	4.5
[35] DNASC	–	genes	4.4:1 – 5.3:1	–
[36][a]	human	genes, genome	5.3:1	0.02
[31] GBC[b]	–	genes	3.5:1	–
[30] 2D[c]	bacteria	genome	3.2:1	0.8

– Dictionary-based Algorithms –

Tool	Tested with	Data used	Compression ratio	Compression speed (MB/s)
[38]	–	genes	4:1	–
[37] Comrad[d]	bacteria, human	genomes	5.5:1 (and better)	0.3
[39]	–	genes	5.3:1 – 5.7:1	–

– Statistical Algorithms –

Tool	Tested with	Data used	Compression ratio	Compression speed (MB/s)
[40][e]	–	genes	4.73:1	–
[44]	bacteria, human	genes	4:1 – 5:1	–
[41]	human	genome	4.7:1	1.08
[43]	human	genome	5:1 – 5.7:1	–
[42] Gene-Compressor	bacteria, human	genes	4.7:1 – 8.8:1	–

[a] Partial sourcecode is available as supplement to original submission
[b] Partial sourcecode is available as supplement to original submission
[c] Sourcecode is available as supplement to original submission
[d] http://www.csse.unimelb.edu.au/~kuruppu/comrad
[e] ftp://ftp.infotech.monash.edu.au/software/DNAcompress-XM/

Table 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.

Tool	Data source	Compression ratio	Speed of compression (MB/s)
[45][a]	up to 8 GB for full diploid human sequence	24:1 – 400:1	–
[46][b]	Wheeler Genome with regard to HG18	772:1	–
[50][c]	yeast and human genomes	16:1	13.33
[52]	few short gene sequences	40:1 – 80:1	0.0005
[53]	various text formats	–	–
[48] RLZ[d]	yeast and human genomes	16:1 – 220:1	–
[49] RLZopt[e]	yeast and human genomes	220:1	0.66
[55]	HG18 human chromosomes	139:1 – 895:1	–
[54]	some genome data and general strings	15:1 – 80:1	–
[51] GReEn[f]	Korean human genome (and rice genome)	172:1	8.33
[47] GRS[g]	Korean human genome (and rice genome)	159:1	1.85

[a] http://mammag.web.uci.edu/bin/view/Mitowiki/ProjectDNACompression; not available on 07/11/2012
[b] http://www.ics.uci.edu/ xhx/project/DNAzip; not available on 07/11/2012
[c] http://sun.aei.polsl.pl/gdc
[d] http://www.genomics.csse.unimelb.edu.au/product-rlz.php
[e] http://www.genomics.csse.unimelb.edu.au/product-rlz.php
[f] ftp://ftp.ieeta.pt/~ap/codecs/GReEn1.tar.gz
[g] http://gmdd.shgmo.org/Computational-Biology/GRS/

Table 4: Alphabetical list of non-referential compression schemes for sequence reads.

Tool	Data source	Compression ratio (seq)	Compression ratio (quality)
[63]	eleven files from 1000 Genomes (human)	4.2:1 – 5.23:1	2.95:1 – 22.63:1
[66]	233 sequences (not published)	8:1 – 11:1 (sequence and quality scores)	
[28] DSRC[a]	nine files from 1000 Genomes (human)	3.23:1 – 6.51:1 (sequence and quality scores)	
[60]	88 short yeast sequences	1.658:1	–
[61] CASToRe	14 genomes of different species	1.909:1 – 2.056:1	–
[65] G-SQZ[b]	six files from 1000 Genomes (human)	2.85:1 – 5.35:1 (sequence and quality scores)	
[64][c]	three large data sets from SRA (human, mouse)	–	2.5 bits per base (lossless) 1 bit per base (lossy)
[62] POMA	eleven short gene sequences	1.3:1	–

[a] http://sun.aei.polsl.pl/dsrc/
[b] http://public.tgen.org/sqz
[c] http://www.cb.k.u-tokyo.ac.jp/asailab/members/rwan

Table 5: Referential compression schemes for sequence reads.

Tool	data source	compression ratio	Speed of compression (MB/s)
[67] GenCompress[a]	three heterogeneous data sets	9.8:1 – 22.7:1 (seq only)	52 – 80
[69][b]	two samples (human, bacteria)	28:1 – 65:1 (seq)	–
		25:1 – 48:1 (seq and lossy qual)	
[68] SLIMGENE	human short reads	39:1 (seq), 2.45:1 (qual)	11

[a]Sourcecode available upon request

[b]Sourcecode is supplementary of original submission