De Novo NGS Data Compression

Gaëtan Benoit, Claire Lemaitre, Guillaume Rizk, Erwan Drezen, Dominique Lavenier

▶ **To cite this version:**

Gaëtan Benoit, Claire Lemaitre, Guillaume Rizk, Erwan Drezen, Dominique Lavenier. De Novo NGS Data Compression. Mourad Elloumi. Algorithms for Next-Generation Sequencing Data, Springer, pp.91-115, 2017, 978-3-319-59826-0. 10.1007/978-3-319-59826-0_4 . hal-01633718

HAL Id: hal-01633718

https://hal.archives-ouvertes.fr/hal-01633718

Submitted on 8 Dec 2017

De Novo NGS Data Compression

Gaetan Benoit, Claire Lemaitre, Guillaume Rizk, Erwan Drezen and Dominique Lavenier

Abstract High throughput sequencing machines decipher billions of nucleotides from DNA molecules at unprecedented speed. This mass of data is stored into large text files structured as a list of small DNA fragments. They represent random overlap regions over one or several genomes. The overlap fragment generate a lot of redundancy that can be advantageously exploited to compress next generation sequencing (NGS) data. This is the main motivation for developing dedicated compressing techniques for this type data over generic text compressors that are not able to capture this kind of redundancy. This chapter focuses on *de novo* NGS data compression, which remains a very challenging issue. Here, no reference genome is considered. Compression and decompression is performed as a standalone process independently of external knowledge. The chapter explains the main NGS compression techniques, including lossless and lossy compression. Additionally, the chapter presents an evaluation of the main state-of-the-art compressors on several real NGS datasets.

Gaetan Benoit
INRIA/IRISA, Rennes e-mail: gaetan.benoit@inria.fr

Claire Lemaitre
INRIA/IRISA, Rennes e-mail: claire.lemaitre@inria.fr

Guillaume Rizk
INRIA/IRISA, Rennes e-mail: guillaume.rizk@inria.fr

Erwan Drezen
INRIA/IRISA, Rennes e-mail: erwan.drezen@inria.fr

Dominique Lavenier
INRIA/IRISA, Rennes e-mail: dominique.lavenier@irisa.fr

1 Introduction

During the last decade, the fast evolution of the sequencing technologies has led to an explosion of DNA data. Every field of life science is now concerned. All of them offer really new insights to approach many biological questions. The low sequencing cost is also an important factor that contributes to their successes. Hence, today, getting molecular information from living organisms is no longer a bottleneck. Sequencing machines can generate billions of nucleotides. On the other hand, managing this mass of data to extract relevant knowledge is becoming a real challenge.

The first step is simply to deal with the information files output by the sequencers. The size of these files can be huge and often ranges from ten to hundreds of Giga bytes. Files have to be stored on the disk storage computing environment (1) to perform the genomic analysis and (2) to be archived. In both cases, the required space storage is high and asks for important hardware resources. Compressing NGS data is thus a natural way to significantly reduce the cost of the storage/archive infrastructure.

Another important point is the exchange of data. The first level is the Internet communication. Practically, transferring files of Tera bytes through Internet takes time and is fastidious (larger the files, longer the transfer time, and higher the probability to be interrupted). A 100 Mbit/s connection will require several hours to transfer a file of 100 Giga bytes. A second level is the internal computer network. Data are generally moved on specific storage bays. When required for processing, they are transferred to the computing nodes. If many jobs involving many different genomic datasets are run in parallel, the I/O computer network can be saturated leading to a drastic degradation of the overall computer performance. One way to limit the potential I/O traffic jam is to directly work with compressed data. It can be faster to spend time in decompressing data than waiting for uncompressed ones.

Compressing NGS data can also be an indirect way for speeding up data processing. Actually, the sequencing coverage provides a significant redundancy that is exploited by DNA data compressors. In other words, similar sequences are more or less grouped together to optimize their description. In many treatments such as sequence comparison, genome assembly or SNP calling (through mapping operation), the aim is to find similarity between sequences. Thus, even if the final objective is different, the way data are manipulated is conceptually very similar. In that sense, a NGS data compressor can be viewed as a pre-processing step before further analysis. But to be efficient, the compressor should be able to directly transmit this pre-processing information to other NGS tools.

Hence, based on the above argumentation, the challenge of the NGS data compression is manifold: (1) provide a compact format to limit computer storage infrastructure; (2) provide a fast decoding scheme to lighten I/O computer network traffic; (3) provide a format to directly populate a data structure suitable for further NGS data processing. Of course, these different criteria are more or less antagonist and compression methods have to make compromises or deliberately privilege one aspect.

Generic compression methods, such as GZIP, do not fit the above requirements. Compression is performed locally (i.e without vision of the complete dataset) and, thus, cannot exploit the coverage redundancy brought by NGS data. However, due to the text format (ASCII) of the NGS files, and the reduced alphabet of the DNA sequences, a compression rate ranging from 3 to 4 is generally obtained. This is far from negligible when such volume of data is involved. Furthermore, the great advantage of GZIP is that it is well recognized, stable, and many NGS tools already take it as input.

Also, independently of not exploiting the global redundancy, gzip-like methods cannot perform lossy compression. Indeed, NGS data include different types of information that can be more or less useful according to downstream processing. Each DNA fragment comes with additional information related to the technological process and the quality control. In some cases, it can be acceptable to suppress or degrade this information to improve the compression ratio. Hence, some compressors propose lossy options that essentially operate on these metadata.

Another way to increase the compression ratio is to rely on external knowledge. If a reference genome is known, NGS data from the same or from a close organism can advantageously benefit from that information. In that case, the compression consists in finding the best mapping for all DNA fragments. Fragments that map are replaced by the coordinates where the best match has been found on the reference together with the differences. This chapter will not discuss anymore of this type of compression as it is dedicated to de-novo compression, i.e. compression without reference. But this approach must not be forgotten. Actually, it could be of great interest in the future as the number of sequenced genomes is growing every day.

The rest of the chapter is structured as follows: section 2 gives an overview of standard string compression methods. Actually, as it will be seen in section 3, NGS files include metadata that can be processed with generic compression methods. It is thus interesting to have a brief overview of these techniques. Section 3 is devoted to methods that have been specifically developed to compressed NGS data. Lossless and Lossy compression are presented. Section 4 provides an evaluation of the different NGS compressors and proposes a few benchmarks. Section 5 concludes this chapter.

2 Generic text Compression

All the methods compressing NGS data files use somehow techniques from generic text compression. These techniques can be adapted or tuned for compressing each component of the NGS files. Therefor, it is essential to know the bases of generic text compression techniques and how the popular methods work.

Text compression consists in transforming a source text in a compressed text whose bit sequence is smaller. This smaller bit sequences is obtained by providing fewer bits to the most frequent data and vice versa. The compression can be lossless or lossy. Lossless techniques rebuild the original text from the compressed text,

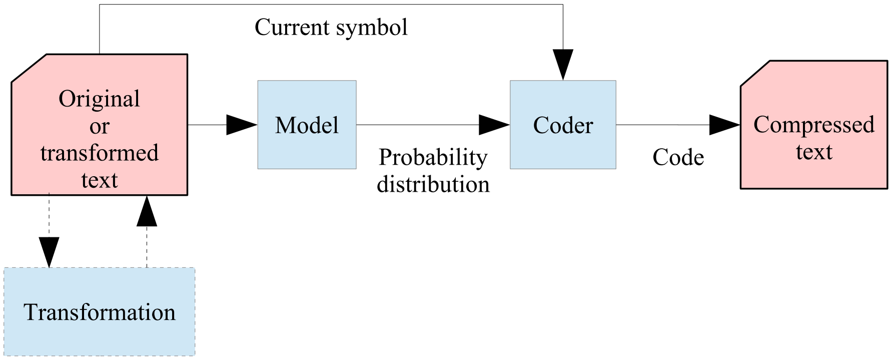

Fig. 1: Standard workflow of a text compressor. Firstly, the original text can be transformed to help further compression. The model then makes a probability distribution of the text and the coder encodes the symbols based on their probability of occurrences.

whereas lossy techniques only retrieve an approximation of the source. In some cases, the loss of information can be desirable, but this section focuses only on lossless methods.

As shown in Figure 1, a text compressor is composed of an optional transformation algorithm, a model and a coder. Transformation algorithms usually produces a shorter representation of the source or a reordering of the source. They can help the compressor in different ways: increasing the compression rate, improving its speed or reducing its memory usage. The model chooses which data is more frequent in the source and estimates a probability distribution of the symbols of this source. The coder outputs a code for each symbol based on the probabilities of the model: the highest probable symbols is encoded by the fewest bit codes.

There are three main kinds of lossless text compression techniques: statistical-based, dictionary-based and those based on the Burrows-Wheeler transform (BWT). The first uses a complex modeling phase to accurately predict the symbols that often appear in the source. The more accurate are the predictions, the better will be the compression rate. The second and the third are transformation algorithms. Dictionary-based methods builds a dictionary containing sequences of the source, each occurrence of these sequences are then replaced by their addresses in the dictionary. Finally, BWT-based methods rearranges the symbols of the source to improve the compression.

2.1 Coding

Coders outputs a code for each symbol depending on the probability distribution computed by the model. The codes must be chosen to provide an unambiguous decompression. Shannon information theory [24] tells us that the optimal code of a symbol s of probability p has a bit length of $-log_2(p)$.

The two most used coders are Huffman coding [11] and arithmetic coding [29]. Huffman coding starts by sorting the symbols based on their probabilities. A binary tree is then constructed, assigning a symbol to each of its leaves. The code of the symbols is obtained by traversing the tree from the root to the leaves. The tree is unbalanced because the codes produced by the encoder must be unambiguously decoded by the decompressor. Frequent symbols will be stored closed to the root, ensuring that frequent data will have the shortest codes, and vice versa. In the rare case where the probability of a given symbol is a power of 1/2, Huffman coding produces optimal code length. In the other cases, this method do not strictly respect the Shannon entropy because the code length are integers.

Arithmetic coding does not bear this limitation and produces codes near of the optimal, even when the entropy is a fractional number. Instead of splitting the source in symbols and assign them a code, this method can assign a code to a sequence of symbols. The idea is to assign to each symbol an interval. The algorithm starts with a current interval [0, 1). The current interval is split in N subintervals where N is the size of the alphabet of the source. The size of the subintervals are proportional to the probability of each symbol. The subinterval of the current symbol to encode is then selected. The boundary of the current interval are narrowed to the range of the selected subinterval. Splitting and selecting intervals continue until the last symbol of the source is processed. The final subinterval specifies the source sequence. The larger is this final interval, the less bits will be required to specify it and vice versa.

Even if Huffman coding can be faster than arithmetic coding, arithmetic coders are most often preferred. The gain in compression rate is really worth compared to the speed penalty.

2.2 Modeling

The model gathers statistics on the source and estimates a probability distribution of the symbols. A model can be static or dynamic. In the static case, a first pass over the source determines the probability distribution, which is used to compress the text in a second pass. The distribution must be stored in the compressed file to allow the decompressor to retrieve the original text. In the dynamic case, the distribution is updated each time a symbol is encoded. The compression is then executed in a single pass over the data. The distribution does not need to be transmitted to the decompressor because the information can be retrieved each time a symbol is decoded. This is a great advantage compared to static models because the distribution can sometimes be heavy. Arithmetic coders perform well on dynamic models whereas Huffman coders are better suited to static models.

2.2.1 Basic modeling

The most used model is a variation of Markov chains : the finite context models. In an order-N context model, the probability of the current symbol is given using the preceding N symbols. This kind of model rely on the fact that the same symbol will often appear after a specific context. A simple example can demonstrate the effectiveness of this model. In an English text, an $order-0$ model (no context) may assign a probability of 2.7% to the letter ”u”. Whereas an $order-1$ model (context of size 1) can assign a probability of 97% to ”u” within the context ”q”, meaning that ”q” is almost always followed by ”u”. A symbol with such a high probability of occurrence can be encoded on one bit or less by the coders which are presented in the following section.

2.2.2 Statistical-based approach

These methods perform a complex modeling phase in order to maximize the efficiency of the coder, usually an arithmetic coder. We have seen in the previous section that order-N context models predict better the probability of the symbols. Increasing N increases the specificity of a context and thus the probabilities of the symbols following it. However, up to a point, the compression rate will decrease because each context has been seen only once in the source and no prediction can be made.

The prediction by partial matching (PPM) [18] aims at solving this problem by using a variable order model. The probability of the current symbol is determined by finding the longest context such that it has already been seen. If the context of size N does not exist in the distribution, the model is updated and the N-1 context is tried out. The compressor reports the change of context by creating an escape symbol. These operations are repeated until the context of size 0 is reached. In this case the same probability is assigned to the 256 possible symbols. The context mixing approach [17] goes further by combining the predictions of several statistical models. A lot of mixing strategies exist, the simplest one is to average the probabilities of each model. The resulting prediction is often better than the prediction of a single model.

The PAQ series [16] is a set of compressor which have implemented a lot of complex models. These tools achieve the best compression rates but they are not so popular, mainly because of their lower speed.

2.3 Transforming

Transformation algorithms produce a shorter representation of the source or a reordering of the symbols to help further compression. After the transformation, a modeling and a coding phase are always required, but simpler models can be used.

2.3.1 Dictionary-based approaches

Dictionary-based methods choose sequences of symbols of the source and replace them by short codewords. The idea is to store the sequences once in a data structure, called the dictionary, their code being simply their index in the dictionary.

Lempel and Ziv implemented LZ77 [32] the first algorithm based on dictionary coding. There are numerous variations of this algorithm, the key point being how they choose the sequences and how they store them in the dictionary. LZW [28] uses a dictionary of 4096 entries, each sequence is thus encoded on 12 bits. During compression, the longest sequence matching with the dictionary is searched. When such a match is found, the algorithm outputs the index of the sequence and a new entry is inserted in the dictionary, which is the concatenation of the sequence and the character following this sequence. The dictionary is initially filled with the 256 possible values of a byte in the 256 first entries. It allows the decompressor to rebuilt the dictionary from the compressed data without having to store anything about it in the compressed file.

LZ variants are implemented in popular compression tools, such as gzip [8] which uses a combination of LZ77 and Huffman coding. Their compression speed tends to be slow because of the construction of the dictionary, but their decompression speed, on the contrary, is very fast. This makes them very useful algorithms for text archiving.

2.3.2 BWT-based approaches

The Burrows-Wheeler transform (BWT) [3] is not directly a compression technique but a reordering of the symbols of the source. The transformation tends to group the identical symbols together when the source contains redundant sequences. A simple compression technique, such as run-length encoding (RLE), can then be used to reduce the sequences of repeated symbols. RLE replaces these sequences by a couple representing the number of occurrence and the repeated symbol, for example the string BBBBBBBCDDD, will become 7B1C3D.

The BWT is obtained by sorting all the rotations of a sequence by alphabetical order. The last symbol of each rotation are concatenated to form the BWT. One of the main advantage of this technique is the reversibility property of the BWT. A compressor can then forget the original text and work only on the BWT.

The popular compressor bzip2 [23] is based on the BWT. Its compression rate is better than the dictionary-based methods but the execution time is slower for both compression and decompression.

3 NGS data Compression

3.1 Introduction

The sequencing machines are highly parallel devices that process billions of small DNA biomolecules. A sequencing run is a complex biochemical protocol that simultaneously reads all these biomolecules. This reading is performed by very sensitive captors that amplify and transform the biochemical reactions into electronic signals. These signals are analyzed to finally generate text files that contain string sequences over the A, C, G, T alphabet, each of them representing one nucleotide. In addition, depending of the quality of the signal, a quality score is associated to each nucleotides.

The standard file format output by the sequencing machines is the fastq format. This is a list of short DNA strings enhanced with quality information. Each sequence has also a specific identifier where different information related to the sequencing process is given. An example of fastq file with only 2 DNA sequences is:

```
@FCC39DWACXX:4:1101:12666:1999#CTAAGTCG/1
NGCCGAAACTTAGCGACCCCGGAAGCTATTCAATTACATGTGCTATCGGTAACATAGTTA
+
BPacceeeggggfhifhiiihihfhiifhhigiiiiiiiffgZegh`gh`geggbeeeee
@FCC39DWACXX:4:1101:16915:1996#CTAAGTCG/1
NTGACTTCGGTTAAAATGTTAAGTTATGGACGAGTTTGAGTTTGTGATTTTAATCTTTCA
+
BPacceeegggggiiiiihhhhihhiiihiiihicggihichhihiiihihiiiiiiiih
```

The line starting with the character @ is the sequence header. It gives the metadata associated to the DNA sequence and to the sequencing process. The following line is the DNA sequence itself and is represented as a list of nucleotides. The N character can also appear for nucleotides that have not been successfully sequenced. The line + acts as a separator. The last line represents the quality. The number of symbols is equal to the length of the DNA sequence. A quality value ranging from 0 to 40, is associated to each symbol, each value being encoded by a single ascii symbol.

Hence, we may consider that NGS files contain three different types of information. Each of them has its own properties that can be exploited to globally optimize the compression. Headers are very similar between each others, with a common structure and fixed fields. They can therefore be efficiently compressed with generic text compression methods, described in the previous section. The DNA sequence stream contains redundant information provided by the sequencing coverage, but this information is spread over the whole file. Therefore, sequences must first be grouped together to benefit from this redundancy. Finally, quality compression is probably the most challenging part since there are no immediate features to exploit.

Actually, the quality information may highly limit the compression rate. So it is legitimate to wonder whether this information is essential to preserve or, at least, if some loss in quality would be acceptable. A first argument is that quality information is often skipped by a majority of NGS tools. A second one is that the computation of the quality scores with such precision (from 0 to 40) does not reflect the real precision of sequencers. The range could be reduced to lower the number of possible values with very minor impact to the incoming processing stages, but with a great added value on compression. Recent studies have even shown that reducing the quality scoring can provide better results on SNP calling [31]. Removing the quality, re-encoding the quality score on a smaller scale, or modifying locally the score to smooth the quality information are also interesting ways to increase the compression rate.

The two next subsections describe in detail quality and NGS sequence compression methods. The header part compression is no longer considered since generic text compression techniques, as described section 2, perform well.

3.2 Quality compression

For each nucleotide, the fastq format includes a quality score. It gives the ”base-calling” confidence of the sequencer, i.e. the probability that a given nucleotide has been correctly sequenced. In its current format, quality scores Q are expressed in an integer scale ranging from 0 to 40, derived from the probability p of error with $Q = -10\log_{10} p$. These scores are usually named *Phred scores*, after the name of the first software that employed it. In the uncompressed Fastq file, these scores are encoded with ascii characters. Each DNA sequence is accompanied by a quality sequence of same length, therefore quality strings takes the same space as DNA in the original file. Its compression is more challenging than DNA compression in several aspects. First, its alphabet is much larger ($\approx$ 40 symbols instead of 4), secondly it does not feature significant redundancy across reads that could be exploited, and lastly, it often looks like a very noisy signal.

We distinguish between two different compression schemes: *lossless* and *lossy*. *lossless* mode ensures the decompressed data exactly match the original data. On the contrary, *lossy* compressors store only an approximate representation of the input data but achieves much higher compression rates. *lossy* compression is traditionally used when the input data is a continuous signal already incorporating some kind of approximation (such as sound data). At first NGS compression software were mostly using *lossless* compression, then several authors explored various *lossy* compression schemes. *lossy* compression schemes can themselves roughly be divided in two categories: schemes that work only from the information contained in the quality string, and schemes that also exploit the information contained in the DNA sequence to make assumptions about which quality score can safely be modified.

3.2.1 Lossless compression

Context model and arithmetic coding

It has been observed that in most sequencing technologies there is a correlation between quality values and their position, and also a strong correlation with the nearby preceding qualities. The use of an appropriate context-model can therefore help predict a given quality value. QUIP uses a third-order context model followed by arithmetic coding to exploit the correlation of neighboring quality scores, DSRC2 uses an arithmetic coder with context length up to 6, and FQZCOMP also uses a mix of different context-models to capture the different kind of correlations [13, 20, 2].

Some contexts used by FASTQZ to predict a quality Q_i are for example Q_{i-1} (immediate preceding quality score), $max(Q_{i-2}, Q_{i-3})$, whether Q_{i-2} equals Q_{i-3} or not, and $min(7, i/8)$, i.e. reflecting score is dependent on position. These contexts are heuristics and may perform differently on datasets generated by varying sequencing technologies. The context-model is then generally followed by an arithmetic coder, coding highly probable qualities with fewer bits.

Transformations

Wan *et al.* explored different lossless transformations that can help further compression: *min shifting* and *gap translating* [27]. With *min shifting*, quality values are converted to $q - q_{min}$, with q_{min} the minimum quality score within a block. With *gap translating* qualities are converted to $q - q_{prec}$. The objective of such transformations is to convert quality to lower values, possibly coded with fewer bits. This can act as a pre-processing step for other methods.

3.2.2 Lossy compression

One major difficulty of quality compression comes from the wide range of different qualities. The first simple idea to achieve a better compression ratio is to reduce this range to fewer different values.

Wan *et al.* explored the effects of three lossy transformations: *unibinning*, *logbinning*, and *truncating* [27]. In the first transformation, the distribution of error probabilities are equally split into N bins. That is, with $N = 5$, the first bin spans quality scores with probability of error from 0% to 20%. All quality values in that bin will be represented by the same score. In the second transformation, the logarithm of error probabilities are equally divided into N bins. This respects the spirit of the original Phred score, if original Phred scores are represented with values from 0 to 63, a logbinning with $N = 32$ amounts to convert each uneven score to its lower nearest even value. Several software use similar discretization scheme, such as FASTQZ and FQZCOMP in their lossy mode) [2].

Other approaches exploit information in the read sequences to modify quality values. Janin *et al.* assume that if a given nucleotide can be completely predicted by the context of the read, then its corresponding quality value can be aggressively compressed, or even completely discarded. They first transform the set of reads to their Burrows-Wheeler transform to make such predictions from the longest common prefix array (LCP) [12].

Yu *et al.* also exploits the same idea, but without computing the time-consuming BWT of reads [31]. Instead, they compute a dictionary of commonly occurring k-mers throughout a read set. Then, they identify in each read kmers within small Hamming distance of commonly occurring k-mers. Position corresponding to differences from common k-mers are assumed to be SNP or sequencing errors, and their quality values are preserved. Other quality scores are discarded, i.e. replaced by a high quality score. However, their method scale to very large dataset only if the dictionary of common k-mers is computed on a sample of the whole dataset. LEON also exploits a similar idea. It counts the frequency of all kmers in the read set, and automatically computes the probable solidity threshold above which k-mers are considered to be error-free. Then, if any given position in a read is covered by a sufficient number of solid kmer, its quality value is discarded. If the initial quality score is low, then a higher number of solid k-mers is required in order to replace the quality score by a high value. The underlying idea is that replacing a low quality score by a high score may be dangerous for downstream NGS analysis, and therefore should be conducted carefully. These approaches are obviously *lossy* since they change the original qualities. However, modifying the quality values based on the information extracted from the reads means that some quality scores are actually *corrected*. This can be viewed as an amelioration instead of a loss, and explains the improvements of downstream NGS analysis discussed in [31].

3.3 DNA sequence compression

The most simple compression method for DNA sequences consists in packing multiple nucleotides in a single byte. The DNA alphabet is a four letter alphabet A, C, G and T, and each base can thus be 2 bit encoded. The N symbol (that actually rarely appears) does not necessarily need to be encoded because its quality score is always 0. During decompression, the symbol is simply inserted when a null quality score is seen. Packing four nucleotides per byte represents however a good compression rate, decreasing the size of the DNA sequences by a factor 4.

To get better compression ratio, two main families of compression techniques have been developed: reference-based methods and reference-free methods, also called *de novo* methods. The idea of the reference-based methods is to store only the differences between already known sequences. On the contrary, The *de novo* methods do not use external knowledge. They extract compression rate by exploiting the redundancy provided by the sequencing coverage. The compression rate of the reference-based methods can be really better, but they can be only used if similar sequences are already stored in databases. This is generally not the case when sequencing new species.

This chapter only focuses on *de novo* compression methods that, from an algorithmic point of view, are more challenging. They have first to fastly extract redundancy form the dataset, and have also to deal with sequencing errors that actually disrupt this redundancy. *De novo* DNA sequence compression methods fall in three categories : (i) approaches using a context-model to predict bases according to their context, (ii) methods that re-order the reads to boost generic compression approaches and (iii) approaches inspired from the reference genome methods.

Historically, the first DNA compressors were only improvements of generic text compression algorithms and belonged to the context-model and re-ordering categories. It is only recently that new methods appeared that fully exploit algorithms and data structures tailored for the analysis of NGS data, such as the *de Bruijn Graph*, mapping schemes or the Burrows Wheeler Transform.

3.3.1 Statistical-based methods

These methods start with a modeling phase in order to learn the full genome structure, i.e. its word composition. With high order context models and a sufficient sequencing coverage, the models are able to accurately predict the next base to encode.

G-SQZ Tembe et al. [26] is the first tool to focus on the *de novo* compression of NGS data. It achieves a low compression rate by simply using an order 0 model followed by an Huffman coder. QUIP [13] and FQZCOMP [2] obtained, by far, a better compression rate by using a higher order context model of length up to 16 and an arithmetic coder. FASTQZ [2] grants more time to the modeling phase by using a mix of multiple context models. DSRC2 [7] can be seen as an improvement of gzip adapted for NGS data. Its best compression rate is obtained by using a context model of order 9 followed by an Huffman coder. But the proposed fully multi-threaded implementation makes it the fastest of all NGS compression tools.

The efficiency of these methods is strongly correlated with the complexity and the size of the target genome. On large genomes, such as the Human one, an order 16 context model will not be sparsely populated and will see more than one of the four possible nucleotides after a lot of context of length 16, resulting in a poor compression rate. A solution will be to use higher order models but a context model is limited by its memory usage of 4^N for the simplest models. These methods also cannot deal with sequencing errors. These errors provoke a lot of unique contexts which will just degrade the compression rate.

3.3.2 Read reordering methods

The purpose of reordering reads is to group together similar reads in order to boost the compression rate obtained by generic compression tool, such as GZIP. In fact, dictionary and BWT based compressors only work on small blocks of data before clearing their indexing structure. This is a solution to prevent a too high memory usage. In the case of DNA sequence compression, such tools miss the whole redundancy of the source because similar reads can be anywhere in the dataset. If the reordering is accurate, some dedicated transformation algorithms can also be used to encode a read using one of its previous reads as reference.

RECOIL [30] constructs a similarity graph where the nodes store the sequences. A weighted edge between two nodes represents their number of shared kmers. Subsets of similar reads are retrieved by finding maximal paths in the graph. Finally, the largest matching region between similar reads are determined by extending their shared kmers. FQC [21] also identifies similar reads by their shared kmers. But this time, the reads are clustered together. And for each cluster, a consensus sequence is built by aligning all of its reads. The compression algorithm]uses the consensus as reference to encode the reads. Both methods are time or memory consuming. They have been applied only on small datasets and are hardly scalable to today's volumes of data.

BEETL [6] proposes an optimization of the BWT which is able to scale on billions of sequences. Classical BWT algorithms cannot handle as many sequences because of their memory limitations. Here, much of the memory is saved during the process by making use of sequential reading and writing files on disk. Cox et al. [6] also showed that the compression ratio is influenced by the sequence order. In fact, each sequence in the collection is terminated by a distinct end-marker. Depending on the assignment of these end-markers to each sequence, hence depending on the order of the sequences in the collection, the BWT can result in longer runs of the same character. The most compressible BWT is obtained by reversing the sequences and sorting them by lexicographic order. To avoid a time and memory consuming pre-processing step of sorting all sequences, the algorithm makes use of a bit-array (called SAP) indicating whether a given suffix is the same as the previous one (except from the end-marker). This enables to sort only small subsets of sequences after the construction of an initial BWT, in order to locally update the BWT.

SCALCE [10] and ORCOM [9] first partition all reads in several bins stored on disk. Each bin is identified by a core string with the idea that reads sharing a large overlap must have the same core string. In SCALCE the core substrings are derived from a combinatorial pattern matching technique that aims to identify ”building blocks” of a string, namely the Locally Consistent Parsing (LCP) method [22]. In ORCOM the chosen core string is called a minimizer, which is the lexicographically smallest substring of size *m* of a read. When the disk bins are built, they are sorted by lexicographically order, with respect to the core string position, to move the overlapping reads close to each other. SCALCE uses generic compression tools, such as gzip, to compress each bin independently whereas ORCOM uses a dedicated compression technique: each read is encoded using one of its previous reads as reference, the best candidate being the one which has the less differences with the current read and is determined by alignment anchored on the minimizer. The differences are encoded using an order-4 PPM model followed by an arithmetic coder.

Reordering methods are well suited to exploit the redundancy of high throughput sequencing data. Some of them can avoid memory problems on large datasets by reordering the reads on disk. For a long time, the main drawback of these methods has been their compression and decompression speed but this problem has been solved recently by ORCOM which is faster and obtains a better compression rate than the other methods. However, it is important to notice that reordering methods is a form of lossy compression. In fact, most of these tools do not restore the original read order, and this can be an issue for datasets containing paired reads (which are the most common datasets). All NGS tools relying on paired reads information (read mapping, *de novo* assembly, structural variation analysis...) can therefore not work on data compressed/decompressed with these tools.

3.3.3 Assembly-based methods

Contrary to the two above categories, methods falling in this third category do not try to find similarities between the reads themselves but between each read and a go-between, called a reference. These approaches are largely inspired from the ones relying on a reference genome.

Reference-based methods require external data to compress and decompress the read file, that is a known reference sequence assumed to be very similar from the one from which reads have been produced. The idea is to map each read to the reference genome and the compressed file only stores for each read its mapping position and the potential differences in the alignment instead of the whole read sequence. However, as already mentioned, relying on external knowledge is extremely constraining and this knowledge is not available for numerous datasets (e.g. *de novo* sequencing or metagenomics). To circumvent these limitations, the scheme of some *de novo* methods is to infer the reference from the read data by itself and then apply the reference-based approaches.

Conceptually, there are no differences between assembly-based methods and reference-based methods. In the first case, however, an assembly step is needed to build a reference sequence. Once this reference is obtained, identical mapping strategies can then be used. An important thing to note is that the reference sequence may not necessarily reflects a biological reality. It is hidden from the user point of view and is only exploited for its compression features.

Two methods follow this strategy : QUIP [13] and LEON [1]. The main difference lies in the data structure holding the reference: in QUIP the reference is a set of sequences, whereas in LEON it is a graph of sequences, namely a *de Bruijn Graph*.

In QUIP, the reference set of sequences is obtained by *de novo* assembly of the reads. Since *de novo* assembly is time and memory consuming, only a subset of the reads (the first 2.5 million by default) are used for this first step. The reads are then mapped on the assembled sequences, called contigs, using a seed-and-extend approach. A perfect match of size 12 between the contigs and the current read is first searched. The remaining of the alignment is then performed, the chosen contig is the one which minimizes the hamming distance. The read is then represented as a position in the chosen contig and the eventual differences.

In LEON, a *de Bruijn Graph* is built from the whole set of reads. The *de Bruijn Graph* is a common data structure used for de novo assembly. It stores all words of size k (kmers) contained in a given set of sequences as nodes and put a directed edge between two nodes when there is a $k-1$ overlap between the kmers. When kmers generated by sequencing errors are filtered out and the value of k is well chosen (usually around 30), the *de Bruijn Graph* is a representation of the reference genome, where each chromosome sequence can be obtained by following a path in the graph. In LEON each read is encoded by a path in such a graph, that is an anchoring kmer and a list of bifurcations to follow when the path encounters nodes with out-degree larger than 1.

The main difference between these two methods is that QUIP executes a complete assembly of the reads whereas LEON stops at an intermediate representation of the assembly which is the *de Bruijn Graph*. In order to decompress the data, the reference must be stored in the compressed file. *A priori*, storing contigs may seem lighter than storing the whole graph. In fact, a naive representation storing each base on 2 bits cost 60 bits per 30-mer which is not acceptable in a compression purpose. The LEON's strategy is possible thanks to a light representation of the *de Bruijn Graph* [19, 5] which stores each kmer on only a ten of bits. The mapping results should be better for LEON than QUIP, since a part of the reads cannot be aligned on contigs: those being at contigs extremities and those which are not assembled.

3.4 Summary

The table 1 shows a summary of the compressors presented in this section. Tools are classified in three main categories : statistical-based methods, reordering methods and assembly-based methods. A brief overview of the compression techniques used is also given. In the Random access column, one can identify the tools that propose the additional feature of quickly retrieving a given read without decompressing the whole file, such as toolG-SQZ and DSRC. BEETL is more than a compressor because its BWT representation can be used for popular indexation structures such as the FM-index. Consequently, BEETL can quickly deliver a list of sequences containing a given kmer.

Some compressors such as G-SQZ and RECOIL are already out-dated. They are not multi-threaded and do not scale on the current datasets. FQZCOMP and FASTQZ can use a maximum of three cores to compress the three streams in parallel. This solution was not qualified as multi-threaded in Table 1, since it is limited and it provides unbalance core activity, the header stream being actually much faster to compress than the two others streams. To take full advantage of multi-core architectures, a solution is to process the data by blocks and compress them independently, as this is the case in DSRC or LEON. Depending on the method, implementing such

Software	Year	Methods	Quality	Random access	Multi-threaded	Remarks
Statistical-based methods						
G-SQZ [25]	2010	Huf		yes	no	
DSRC [7] [20]	2014	Markov, Huf	lossless / lossy	yes	yes	
FASTQZ [2]	2013	CM, AC	lossless / lossy	no	no	
FQZCOMP [2]	2013	Markov, CM, AC	lossless / lossy	no	no	
Reordering methods						
RECOIL [30]	2011	Generic		no	no	Fasta only
FQC [21]	2014	PPM, AC	lossless	-	-	No impl
BEETL [6]	2012	BWT, PPM		yes	no	Fasta only
SCALCE [10]	2012	Generic	lossless / lossy	no	yes	
ORCOM [9]	2014	Markov, PPM, AC		no	yes	Seq only
Assembly-based methods						
QUIP [13]	2012	Markov, AC	lossless	no	no	
LEON [1]	2014	Markov, AC	lossless / lossy	no	yes	
Quality only methods						
RQS [31]	2014	k-mer dictionary	lossy	no	no	
BEETL [12]	2013	BWT, LCP	lossy	no	no	
LIBCSAM [4]	2014	block smoothing	lossy	no	no	

Table 1: Summary of compression methods and software, classified according to their method for the DNA stream. Random access is the ability for the compressor to retrieve a sequence without the need to decompress the whole file. Abbreviations: Huf - Huffman coding, Markov - Context model, CM - Context mixing, AC - Arithmetic coding, Generic - Generic compression tools such as GZIP and BZIP2, PPM - Prediction by partial matching, Seq - Sequence, LCP - Longest common prefix array, Fasta - Fastq format without the quality stream, No impl - No implementation available.

a solution may not be straightforward if one wants to preserve good compression factors.

Compression tools are not generally focusing on one specific king of NGS data. One exception is PATHENC which seems to be specialized for RNA-seq data , although it can still be used on any data [13]. A comparative study of compression performance on different kind of NGS data (whole genome, exome, RNA-seq, metagenomic, ChIP-Seq) is conducted in the LEON and QUIP papers [13, 1]. It shows without surprise that best compression is obtained on data with highest redundancy.

Some tools cannot compress the fastq format. RECOIL and BEETL only accept the fasta format which is a fastq format without quality information. ORCOM only processes DNA sequences and discards header and quality streams. FQC cannot be tested because there is no implementation of the method available.

4 Evaluation of NGS Compressors

This section aims to illustrate the compression methods presented in the previous section by evaluating current NGS compressors. The capacity to compress or decompress files is only tested, even if some tools have advanced functionalities such as the possibility to randomly access sequences from the compress data structure. For each tool, the evaluation process followed this protocol:

1. clear the I/O cache system
2. compress the original file A
3. clear the I/O cache system
4. decompress the compressed file(s) into file B
5. generate metrics

NGS compressors deal with large files, making them I/O data intensive. The way the operating systems handle the read and write file operations may have a significant impact on the compression and decompression time measurements. More precisely, if many compressors are sequentially tested on the same NGS file, the first one will have to read data from the disk, whereas the next ones will benefit of the I/O cache system. Thus, to fairly compare each software, and to have similar evaluation environment, we systematically clear the I/O cache before running compression and decompression. All tools were run on a machine equipped with a 2.50 GHz Intel E5-2640 CPU with 12 cores, 192 GB of memory. All tools were set to use up to 8 threads.

4.1 Metrics

Remember that the fastq format is made of:

- (d) a genomic data stream;
- (q) a quality data stream as phred scores associated to the genomic data;
- (h) an header data stream as a textual string associated to each read.

We now define several metrics allowing compressor software to be fairly evaluated.

Compression factor (CF) This is the principal metric. We define a compression factor (CF) as:

$$CF(s) := \frac{size\ of\ stream\ s\ in\ the\ original\ file}{size\ of\ stream\ s\ in\ the\ compressed\ file}$$

More generally, we define a global compression factor for the whole file:

$$CF := \frac{size\ of\ the\ original\ file}{size\ of\ all\ streams\ in\ the\ compressed\ file(s)}$$

Stream correctness (SC) Depending of the nature of the stream, the correctness is computed differently. For the DNA data stream, we checks that the DNA sequences after the compression/decompression steps are identical. The idea is to perform a global checksum on the DNA sequences only. The DNA data stream correctness is calculated as follows:

$$SC(d) := \begin{cases} ok\ if\ \ Checksum(A,d) == Checksum(B,d) \\ ko\ if\ \ Checksum(A,d)\ != \ Checksum(B,d) \end{cases}$$

Note that A is the original file and B is the decompressed file.

For the header and the quality streams, the correctness depends on the lossy or lossless options. It also depends on the way software handle header compression. We therefore define the stream correctness metric as the following percentage (0 means correct stream):

$$SC(s) := \frac{\sum_i nb\ of\ mismatches\ of\ (ra_i, rb_i)\ in\ stream\ s}{size\ of\ stream\ s} * 100$$

where

$$\begin{cases} s \quad := one\ of\ the\ streams\ (q),\ or\ (h) \\ ra_i := i^{th}\ read\ of\ A \\ rb_i := i^{th}\ read\ of\ B \end{cases}$$

Execution time This is the execution time for compression or decompression expressed in second.

Memory peak This is the maximum memory used (in MBytes) to compress or decompress a file.

4.2 Benchmarks

High coverage benchmarks

NGS compressors are tested with the three following high coverage datasets extracted from the SRA database:

- whole genome sequencing of the bacteria *E. Coli* (genome size ∼ 5 Mbp): 1.3 GB, 116X coverage (SRR959239)
- whole genome sequencing of the worm *C. Elegans* (genome size ∼ 100 Mbp): 17.4 GB, 70X coverage (SRR065390)
- whole genome sequencing of a human individual (genome size ∼ 3 Gbp): 732 GB, 102X coverage (SRR345593/4)

They are real datasets from high throughput sequencing machines (here Illumina, with around 100 bp reads) and are representative of NGS files that are daily gener-

Prog	**Factor**				**Correctness**				**Compress.**		**Decompress.**	
	main	hdr	seq	qlt	sum	hdr	seq	qlt	time (s)	mem. (MB)	time (s)	mem. (MB)
WGS *E. Coli* - 1392 MB - 116x												
gzip	3.9	—	—	—	ok	0	0	0	188	1	25	1
bzip2	4.9	—	—	—	ok	0	0	0	164	9	83	6
dsrc	6.0	—	—	—	ok	0	0	0	21	1995	35	2046
dsrc*	7.6	—	—	—	ok	0	0	81	12	1942	30	1996
fqzcomp	9.9	35.2	12.0	5.7	ok	0	0	0	64	4424	66	4414
fqzcomp*	17.9	35.2	12.0	19.6	ok	0	0	95	64	4165	69	4159
fastqz	10.3	39.9	13.8	5.6	ko	0	75	0	290	1347	320	1347
fastqz*	13.4	39.9	13.8	8.6	ko	0	75	40	248	1347	286	1347
leon	8.4	45.1	17.5	3.9	ok	0	0	0	106	404	45	277
leon*	30.9	45.1	17.5	59.3	ok	0	0	96	49	390	44	239
quip	8.4	29.8	8.5	5.3	ok	0	0	0	206	990	205	807
scalce	8.9	21.0	12.9	5.0	ok	35	75	57	85	1993	47	1110
orcom	—	—	33.51	—	ok	100	75	100	10	2212	15	181
fastqz REF	10.9	40.8	23.5	5.7	ko	0	75	0	306	1316	301	1318
WGS *C. Elegans* - 67 GB - 70x												
gzip	3.8	—	—	—	ok	0	0	0	2218	1	301	1
bzip2	4.6	—	—	—	ok	0	0	0	1808	9	1216	6
dsrc	5.8	—	—	—	ok	0	0	0	200	5355	342	5626
dsrc*	7.9	—	—	—	ok	0	0	86	109	5156	271	4993
fqzcomp	8.1	54.2	7.6	5.2	ok	0	0	0	931	4424	927	4414
fqzcomp*	12.8	54.2	7.6	15.0	ok	0	0	86	921	4169	996	4155
fastqz	7.9	61.9	7.3	5.1	ok	0	0	0	4044	1533	3934	1533
fastqz*	10.3	61.9	7.3	8.7	ok	0	0	76	3703	1533	3312	1533
leon	7.3	48.6	12.0	3.7	ok	0	0	0	1168	1885	446	434
leon*	21.3	48.6	12.0	32.9	ok	0	0	86	704	1886	442	417
quip	6.5	54.3	4.8	5.2	ok	0	0	0	823	782	764	773
scalce	7.7	16.5	10.0	4.7	ok	38	73	58	1316	5285	526	1112
orcom	—	—	24.2	—	ok	94	73	100	283	9488	324	1826
fastqz REF	10.4	63.4	19.2	5.2	ok	0	0	0	3831	1500	3441	1500
WGS Human - 732 GB - 102x												
gzip	3.26	—	—	—	ok				104457	1	9124	1
bzip2	4.01	—	—	—	ok				69757	7	34712	4
dsrc	4.64	—	—	—	ok				1952	408	2406	522
dsrc*	6.83	—	—	—	ok				1787	415	2109	506
fqzcomp	5.35	23.2	4.53	4.2	ok				18695	79	29532	67
fqzcomp*	8.34	23.2	4.53	14.9	ok				20348	80	24867	67
leon	5.65	27.54	9.17	3.03	ok				61563	9607	23814	
leon*	15.63	27.54	9.17	27.1	ok				38860	10598	21687	5870
quip	5.25	16.95	4.47	4.2	ok				52855	798	46595	791
orcom	—	—	19.2	—	ok				29364	27505	10889	62009

Table 2: High coverage benchmarks for *E. Coli*, durability and Human fastq files. Programs with a * are run in lossy compression mode. The overall compression factor is given (main) followed by the compression factor of each stream. The correctness (expressed as a % of differences) is also given for each stream.

ated. Table 2 summarizes the evaluation. Columns have the following meaning:

- **Prog** : name of the NGS compressor. Tools allowing lossy compression on the quality stream are labeled with a * suffix.
- **Factor** : total compression factor, followed by the compression factor for each stream (header, DNA sequence and quality). Note that some tools like GZIP do not have specific stream factors; in such a case, we display only the total factor. The tool ORCOM is specific because it compresses only the DNA sequence stream.
- **Correctness** : results on header, genomic data and quality streams are reported (0 means correct stream)
- **Compression** : system metrics for the compression : execution time (in seconds) and the memory peak (in MBytes)
- **Decompression** : same metrics as above.

The first comment on these results is that, in general, the specific NGS compressors perform better than the generic ones (GZIP and BZIP2), in terms of both compression rate and execution time. The main reason is that these tools exploit the features of NGS data, and especially the redundancy provided by the sequencing coverage. This redundancy is spread over all the file and cannot be locally captured by generic text compressors.

We can also observe that the header streams are always well compressed. As already mentioned, no specific methods have been developed for this data stream, which is mainly composed of repetitive motifs. The generic methods provide near optimal compression and this stream does not constitute the critical part of the NGS data compression challenge.

Concerning the DNA stream, the compression factor depends on the read dataset. It is expected to depend on the read coverage, but the table highlights that it depends also on the size and complexity of the sequenced genome from which the reads are generated: compression factors are worst for the human dataset, even when compared to the *C. elegans* dataset which has a lower coverage (thus less redundancy). This can be explained by the higher complexity of this genome, ie. mainly its amount of repeated sequences. Most compressors try to learn a model representative of the genome, that is its composition in words of size k, that would enable to predict a given word knowing its preceding one: the more the genome contains large repeated sequences (larger than k), the more difficult it is to predict its sequence.

Interestingly, FASTQZ used with a reference genome does not always have the best compression ratio. On the *C. Elegans* dataset, ORCOM outperforms FASTQZ on the DNA sequence stream. However, strictly speaking, the comparison is not perfectly fair since ORCOM reorders the reads. The decompressed set of reads remains the same, but are stored in a different order. For technologies that provide pair-end or mate-pair reads in two different files, this strategy cannot be used.

When the lossy option is activated, the compression ratio can be significantly improved. Techniques that do not consider independently compression for the DNA sequence and the quality streams, but try to exhibit correlations between the two streams, are particularly powerful as demonstrated by the LEON compressor.

Metagenomic benchmark

This test aims to demonstrate that NGS compressors are not well suited for any kind of NGS datasets. In particular, they can perform poorly on NGS metagenomic data. The main reason is that these datasets generally contain low redundancy. Table 3 confirms that the compression rate of DNA sequences is low and is similar to generic compressors. The lossy mode, however, allows fastq files to be significantly compressed.

Prog	**Factor**				**Check**				**Compress.**		**Decompress.**	
	main	hdr	seq	qlt	sum	hdr	seq	qlt	time (s)	mem. (MB)	time (s)	mem. (MB)
gzip	3.4	—	—	—	ok	0	0	0	2206	1	153	1
bzip2	4.4	—	—	—	ok	0	0	0	1578	8	701	4
dsrc	5.0	—	—	—	ok	0	0	0	33	345	25	307
dsrc*	6.8	—	—	—	ok	0	0	78	29	330	39	285
fqzcomp	5.8	32.6	4.5	4.2	ok	0	0	0	360	78	582	67
fqzcomp*	8.9	32.6	4.5	14.8	ok	0	0	90	343	80	456	67
fastqz	6.1	30.0	4.9	4.3	ko				3318	1527		
fastqz*	7.3	30.0	4.9	6.6	ko				2776	1527		
leon	4.8	36.8	4.3	3.1	ok	0	0	0	955	1870	438	1544
leon*	9.0	36.8	4.3	19.8	ok	0	0	91	606	1876	460	1511
quip	5.7	25.6	4.7	4.1	ok	0	0	0	966	774	908	773
orcom	—	—	6.9	—	ok	94	74	89	230	9442	178	1698

Table 3: Metagenomic benchmark. 15 GB (SRR1519083)

SNP calling evaluation

This last part evaluates the loss in quality – or the gain in quality – of downstream processing after a lossy compression. The chosen bioinformatics task is the SNP calling. The evaluation protocol is the following:

- compress file *A* in lossy mode
- decompress file *A* into file *B*
- process file *B* for SNP calling
- compute Recall/Precision metrics

In this experiment, SNPs are called with BWA aligner followed by samtools mpileup [14, 15]. Precision/Recall are computed from a validated set of SNPs from Human chromosome 20, coming from the ”1000 genomes project”, on individual HG00096, read set SRR062634.

Five lossy compression tools, representative of all categories of lossy compression methods are compared. A lossless method was also included in the test, as well as the results obtained when discarding all quality scores, in this case all quality

HG00096 chrom 20			
Prog	**Precision**	**Recall**	**CF**
no quality	57.73	68.66	—
lossless	85.02	67.02	*2.95
FASTQZ	85.46	66.63	5.4
LIBCSAM	84.85	67.09	8.4
FQZCOMP	85.09	66.61	8.9
LEON	85.63	67.17	11.4
RQS	85.59	67.15	12.4

Table 4: Recall and precision of SNP calling after quality values were modified by several lossy compressors. CF is the compression factor. No quality means that qualities were discarded by compression, all replaced by 'H'. For the *lossless* line, the best compression factor obtained by lossless compression tools is given (obtained here with FQZCOMP).

values were replaced by a high value, 'H', for the SNP calling procedure. Results are shown in Table 4.

First, it can be seen that qualities are indeed useful for SNP calling, the *no quality* test has a significant drop in precision, from 85.02 in lossless to 57.7 %. Then, the tools FASTQZ, LIBCSAM and FQZCOMP achieves better compression factor than lossless, with only a slight degradation in precision/recall. Lastly, tools using information coming from the reads, such as LEON and RQS, achieve both a high compression factor and an improvement in precision and recall compared to the original dataset. This can be explained by the fact that, during compression, some qualities are in fact *corrected* using read information.

5 Conclusion

Due to the recent emergence of NGS data, *de novo* NGS compression is still an open and active research field. A large number of methods have already been proposed and have shown that NGS data redundancy can bring a high compression rate. In addition, if lossy compression of the quality information is permitted, much better compression can be reached with, in some cases, an improvement in the the post-processing quality.

Generally, performances of the NGS compressors are first evaluated by their compression rate. In lossless mode, compression rate ranges from 5 to 10. The differences come both from the compression methods and the data themselves. For NGS files with a high sequencing coverage the maximum compression rate on DNA fragments is achieved, but the global compression rate is limited by the quality information that doesn't present the same redundancy property. On the other hand, if lossy mode is allowed, the compression rate can be drastically improved as demonstrated, for example, by the LEON compressor.

Compression and decompression time are another parameter to take into consideration. NGS compressors behave very differently on that point. DSRC, for example, is very fast to compress NGS files, but has a moderate compression rate compared to other concurrent software. Its great advantage is that it is 10 times faster than generic compressors with a better compression rate. As usual, there is a trade-off between speed and quality: longer the compression time, better the compression rate.

The nature of the NGS data may also dictate the choice of the compressors. Metagenomic data, for instance, are not very redundant. As shown in the previous section, compressors perform poorly on these data. In that case, a fast compressor is probably a much better choice since methods optimized to extract redundancy will systematically fail.

However, even if specific compression methods devoted to NGS data have demonstrated their superiority over generic tools (such as GZIP for example), no tool has still emerged as a recognized and routinely used software. The main reasons are:

- **Durability**: tools are often research lab prototypes with a first objective of demonstrating a new compression strategy. The maintainability of these tools over the time is not guaranteed, nor their ascendant compatibility. In that case, decompression of a NGS file can be impossible with higher versions of the compression/decompression software.
- **Robustness**: our tests have shown that, in some cases, NGS compression tools were not able to process large files, or that files, after decompression, were differing from original ones. This is probably due to the youth of these tools that are not yet fully debbuged.
- **Flexibility**: the literature shows there exist no universal NGS compression tool. Each tool has its own special features that do not cover all needs. For example, some compressors only compress DNA sequences, others takes only as input reads of fixed length, and other ones do not propose the lossy/lossless quality option, etc.

Finally, the huge size of the NGS files leads to important compression/decompression time. The compression of a 700 GB file (see previous section) takes several hours. Decompression is often a little bit faster but remains a time-consuming task. A way to avoid these compression/decompression steps is to have bioinformatics tools that directly exploit the compressed format. This is the case for the GZIP format: many tools are able to internally perform this decompression step thanks to the availability of a gzip library that can be easily included in the source codes. However, even if this task becomes transparent for the users, it is still performed. The next step would be to directly exploit the data structure of the compressed files, i.e. without explicitly reconstructing the original list of DNA fragments. As a mater of fact, the majority of the treatments performs on NGS files are detection of small variants (such as SNP calling), mapping on a reference genome, assembly, etc. All these treatments would greatly benefit from the data processing done during the compression step, especially the management of the redundancy. The next generation of compressors should act on this direction and propose formats to favor such usage.

References

[1] G. Benoit, C. Lemaitre, D. Lavenier, and G. Rizk. Compression of high throughput sequencing data with probabilistic de bruijn graph. *arXiv.org*, Dec. 2014. URL http://arxiv.org/abs/1412.5932.

[2] J. K. Bonfield and M. V. Mahoney. Compression of fastq and sam format sequencing data. *PLoS One*, 8(3):e59190, 2013.

[3] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. *Technical Report 124, Digital Equipment Corporation*, 1994.

[4] R. Cánovas, A. Moffat, and A. Turpin. Lossy compression of quality scores in genomic data. *Bioinformatics*, 30(15):2130–2136, 2014.

[5] R. Chikhi and G. Rizk. Space-efficient and exact de bruijn graph representation based on a bloom filter. *Algorithms Mol Biol*, 8(1):22, 2013.

[6] A. J. Cox, M. J. Bauer, T. Jakobi, and G. Rosone. Large-scale compression of genomic sequence databases with the burrows-wheeler transform. *Bioinformatics*, 28(11):1415–1419, Jun 2012.

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

[8] P. Deutsch and J. Gailly. Zlib compressed data format specification version 3.3. *RFC 1950*, 1996.

[9] S. Grabowski, S. Deorowicz, and . Roguski. Disk-based compression of data from genome sequencing. *Bioinformatics*, Dec 2014.

[10] F. Hach, I. Numanagic, C. Alkan, and S. C. Sahinalp. Scalce: boosting sequence compression algorithms using locally consistent encoding. *Bioinformatics*, 28(23):3051–3057, Dec 2012.

[11] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 1952.

[12] L. Janin, G. Rosone, and A. J. Cox. Adaptive reference-free compression of sequence quality scores. *Bioinformatics*, page btt257, 2013.

[13] D. C. Jones, W. L. Ruzzo, X. Peng, and M. G. Katze. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic Acids Res*, 40(22):e171, Dec 2012.

[14] H. Li and R. Durbin. Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[15] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and . G. P. D. P. Subgroup. The sequence alignment/map format and samtools. *Bioinformatics*, 25(16):2078–2079, 2009. doi: 10.1093/bioinformatics/btp352.

[16] M. Mahoney. http://mattmahoney.net/dc/.

[17] M. Mahoney. Adaptive weighing of context models for lossless data compression. *Florida Tech. Technical Report*, 2005.

[18] A. Moffat. Implementing the ppm data compression scheme. *IEEE Transactions on communications*, 1990.

[19] G. Rizk, D. Lavenier, and R. Chikhi. Dsk: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653, Feb 2013.

[20] L. Roguski and S. Deorowicz. Dsrc 2–industry-oriented compression of fastq files. *Bioinformatics*, 30(15):2213–2215, Aug 2014.

[21] S. Saha and S. Rajasekaran. Efficient algorithms for the compression of fastq files. In *2014 IEEE International Conference on Bioinformatics and Biomedicine*, 2014.

[22] S. C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, FOCS '96, pages 320–, Washington, DC, USA, 1996. IEEE Computer Society. URL http://dl.acm.org/citation.cfm?id=874062.875524.

[23] J. Seward. bzip2 : http://www.bzip.org/1.0.3/html/reading.html. 1996.

[24] C. Shannon and W. Weaver. The mathematical theory of communication. *Urbana: University of Illinois Press*, 1949.

[25] W. Tembe, J. Lowey, and E. Suh. G-sqz: compact encoding of genomic sequence and quality data. *Bioinformatics*, 26(17):2192–2194, Sep 2010.

[26] W. Tembe, J. Lowey, and E. Suh. G-sqz: compact encoding of genomic sequence and quality data. *Bioinformatics*, 2010.

[27] R. Wan, V. N. Anh, and K. Asai. Transformations for the compression of fastq quality scores of next-generation sequencing data. *Bioinformatics*, 28(5):628–635, 2012.

[28] T. Welch. A technique for high-performance data compression. *Computer*, 1984.

[29] I. Witten, R. Neal, and J. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 1987.

[30] V. Yanovsky. Recoil - an algorithm for compression of extremely large datasets of dna data. *Algorithms Mol Biol*, 6:23, 2011.

[31] Y. W. Yu, D. Yorukoglu, and B. Berger. Traversing the k-mer landscape of ngs read datasets for quality score sparsification. In *Research in Computational Molecular Biology*, pages 385–399. Springer, 2014.

[32] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans Inf Theory*, 1977.