UNIVERSIDAD
DE LA REPUBLICA
URUGUAY

Compression algorithms for biomedical signals and nanopore sequencing data

Guillermo Dufort y Álvarez Zorrilla de San Martín

Director:
Dr. Ing. Álvaro Martín Menoni

Programa de Doctorado en Informática
PEDECIBA
Universidad de la República

Montevideo – Uruguay
Octubre de 2021

A mi madre Carmen, mi padre
Gustavo y a mi abuela Iaia.

Agradecimientos

Muchas personas me han acompañado durante mis estudios de doctorado. Quiero agradecer a mi madre Carmen y a mi padre Gustavo por su incesante apoyo y amor incondicional. Les debo la pasión por aprender, y la creatividad y tenacidad que me fueron sumamente necesarias para recorrer este camino. A mis hermanas Maru y Valen, a mis sobrinos Sara y Alejo, y a Juani, les agradezco por alivianarme el camino llenando de alegrías todos los fines de semana.

Quiero agradecer muy especialmente a mi abuela Iaia, a quien tantas veces dediqué tanto menos tiempo del que hubiese querido. Ella es una referente y una inspiración de vida para mi. A mis dos familias, a los Dufort y a los Zorrilla, por estar siempre presentes y unidos. En particular, quiero agradecer a Conce por haberme compartido su hogar cuando más lo necesitaba.

Agradezco muy especialmente a Idoia, por haber confiado en mí y recibirme tan amablemente en UIUC. Fue una experiencia que me dio la oportunidad de conocer un mundo diferente lleno de personas fascinantes. Agradezco también los intercambios y los consejos, que fueron claves para mi formación como investigador. También quiero agradecer a Gadiel por sus excelentes clases y por ser una fuente inagotable de consejos y conocimiento.

A mis amigas y amigos que me han acompañado en todo momento, les agradezco de corazón. No hubiese podido transitar este camino de no ser por el apoyo que me brindaron en los momentos de mayor incertidumbre. Un especial agradecimiento a Julián, Ger, Renzo, Santi, Pique, Seba, Vila, Pepa, Pili, Pita, José, Bruno, Cami, Sofi, Agas, Andy, Mau, Mai y Fava. Agradezco también a la Comisión Académica de Posgrado de la Universidad de la República por la financiación provista.

Por último, y no menos importante, a mi tutor Álvaro. Su profundo amor e incansable dedicación a la enseñanza me han inspirado a recorrer y elegir este camino todos los días. Durante todos estos años hemos compartido innumerables charlas y me ha brindado incontables consejos, muchos de ellos más allá de lo estrictamente académico. Siempre me sentí valorado, respetado y capaz de lograr cosas que jamás hubiese imaginado que podría lograr. El valor de sus enseñanzas es algo que voy a atesorar por el resto de mi vida. Muchas gracias.

RESUMEN

La generación masiva de información digital biológica da lugar a múltiples desafíos informáticos, como su almacenamiento y transmisión. Por ejemplo, las señales biomédicas, como los *electroencefalogramas* (EEG), son generadas por múltiples sensores registrando medidas en simultáneo durante largos períodos de tiempo, generando grandes volúmenes de datos. Otro ejemplo son los datos de secuenciación de ADN, en donde la cantidad de datos a nivel mundial está creciendo de forma explosiva, lo que da lugar a una gran necesidad de recursos de procesamiento, almacenamiento y transmisión. En esta tesis investigamos cómo aplicar técnicas de compresión de datos para atacar este problema, en dos escenarios diferentes donde la eficiencia computacional juega un rol importante.

Primero estudiamos la compresión de *señales biomédicas multicanal*. Comenzamos presentando un nuevo compresor de datos sin pérdida para señales multicanal, GSC, que logra obtener niveles de compressón en el estado del arte y que al mismo tiempo es más eficiente computacionalmente que otras alternativas disponibles. El compresor utiliza dos nuevas implementaciones de los esquemas de codificación predictiva y de asesoramiento de expertos para señales multicanal, basadas en aritmética de enteros. También presentamos una versión de GSC optimizada para datos de EEG. Esta versión logra reducir significativamente los tiempos de compresión, sin deteriorar significativamente los niveles de compresión para datos de EEG.

En un segundo escenario estudiamos la compresión de datos de secuenciación de ADN generados por *tecnologías de secuenciación por nanoporos*. En este sentido, presentamos dos nuevos algoritmos de compresión sin pérdida, específicamente diseñados para archivos FASTQ generados por tecnología de nanoporos. ENANO es un compresor *libre de referencia*, enfocado principalmente en la compresión de los valores de calidad de las bases. ENANO alcanza niveles de compresión en el estado del arte, siendo a la vez más eficiente computacionalmente que otras herramientas populares de compresión de archivos FASTQ. Por otro lado, RENANO es un compresor *basado en la utilización de una referencia*, que mejora el rendimiento de ENANO, a partir de un nuevo esquema de compresión de las secuencias de bases. Presentamos dos variantes de RENANO, correspondientes a los siguientes escenarios: (i) se tiene a disposición un genoma de referencia, tanto del lado del compresor como del descompresor, y (ii) se tiene un genoma de referencia disponible solo del lado del compresor, y se incluye una versión compacta de la referencia en el archivo comprimido. Ambas variantes de RENANO mejoran significativamente los niveles compresión de ENANO, alcanzando tiempos de compresión similares y un mayor consumo de memoria.

Palabras claves:
Compresión de datos sin pérdida, Señales multi-canal, Electroencefalogramas, Secuenciación de ADN, Secuenciación por nanoporos, Compresión eficiente.

ABSTRACT

The massive generation of biological digital information creates various computing challenges such as its storage and transmission. For example, biomedical signals, such as *electroencephalograms* (EEG), are recorded by multiple sensors over long periods of time, resulting in large volumes of data. Another example is genome DNA sequencing data, where the amount of data generated globally is seeing explosive growth, leading to increasing needs for processing, storage, and transmission resources. In this thesis we investigate the use of data compression techniques for this problem, in two different scenarios where computational efficiency is crucial.

First we study the compression of *multi-channel biomedical signals*. We present a new lossless data compressor for multi-channel signals, GSC, which achieves compression performance similar to the state of the art, while being more computationally efficient than other available alternatives. The compressor uses two novel integer-based implementations of the predictive coding and expert advice schemes for multi-channel signals. We also develop a version of GSC optimized for EEG data. This version manages to significantly lower compression times while attaining similar compression performance for that specific type of signal.

In a second scenario we study the compression of DNA sequencing data produced by *nanopore sequencing technologies*. We present two novel lossless compression algorithms specifically tailored to nanopore FASTQ files. ENANO is a *reference-free* compressor, which mainly focuses on the compression of quality scores. It achieves state of the art compression performance, while being fast and with low memory consumption when compared to other popular FASTQ compression tools. On the other hand, RENANO is a *reference-based* compressor, which improves on ENANO, by providing a more efficient base call sequence compression component. For RENANO two algorithms are introduced, corresponding to the following scenarios: a reference genome is available without cost to both the compressor and the decompressor; and the reference genome is available only on the compressor side, and a compacted version of the reference is included in the compressed file. Both algorithms of RENANO significantly improve the compression performance of ENANO, with similar compression times, and higher memory requirements.

Keywords:
Lossless data compression, Multi-channel signals, Electroencephalograms, DNA sequencing, Nanopore sequencing, Efficient compression.

Contents

Chapter 1

Introduction

The massive generation of biological digital information creates various computing challenges such as its storage and transmission. For example, biomedical signals, such as *electroencephalograms* (EEG) or *electrocardiograms* (ECG), are recorded by multiple sensors over long periods of time, resulting in large volumes of data. Another example is genome DNA sequencing, in which the chemical composition of DNA fragments is determined using *High-Throughput Sequencing (HTS)* technologies. For HTS technologies, the result of this process is a file with a representation of many thousands of DNA fragments. The costs of sequencing have fallen so rapidly that there is a broad consensus that the amount of genomic information that will be generated in the world will see explosive growth, and, very soon, the costs of computing infrastructure for storing and transmitting the information will exceed those of the sequencing itself [100].

In this thesis we investigate the use of data compression techniques for this problem. In particular, we focus on studying two different scenarios of biological data compression, where computational efficiency is crucial.

First we study the compression of *multi-channel biomedical signals*, in an scenario where processing resources are scarce due to severe restrictions on energy consumption. With the advancement of technology and medicine, real-time clinical studies recorded using wireless acquisition devices are becoming more common. In many cases, the recording systems need to be wearable, and in turn, lightweight, which limits the hardware and battery that can be embedded in the recording system. In this context, efficient compression techniques that are able to run on very limited hardware can help save energy by reducing the amount of data that needs to be wirelessly transmitted.

In a second scenario we focus on studying the compression of DNA sequencing data produced by HTS technologies, in particular, we focus on the compression of data produced by *nanopore sequencing technologies* [93]. In this scenario, the

amount of data that needs to be compressed is massive. For example, the nanopore sequencing device *PromethION 48* can generate up to 14 TB of DNA data in less than 72 hours[1]. In contrast to the previous scenario, the compression of DNA sequencing data is usually performed in computing systems with no specific hardware restrictions, but efficient compression methods are still thoroughly needed, due to the large volume of data that needs to be processed. Also, the massive nature of the data makes efficient memory management a key feature of the algorithms.

The specific types of biological data that we study are important in practical applications, and they represent two different compression scenarios, with different characteristics and requirements, where efficient data compression is needed. For clarity, we address each scenario in a separate part of the thesis.

1.1 Data compression generalities

We start by giving a brief introduction to the basic concepts of data compression, which we explore in more depth in Chapter 2. In practical terms, a *data compressor* is a device that translates a certain type of data into binary strings, i.e, a stream of 0's and 1's, by using an *encoding algorithm*. The general scheme of data compression is depicted in Figure 1.1.

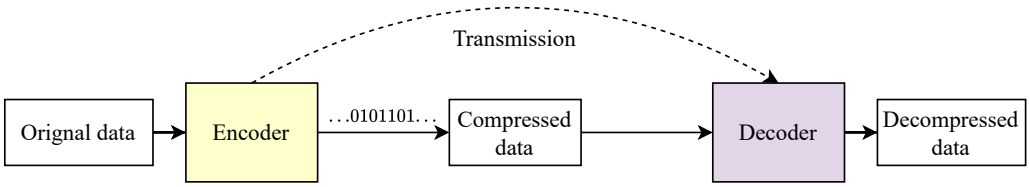


Figure 1.1: General scheme of data compression. The *encoder* receives certain type of data and applies an encoding algorithm that generates a bit stream, which we call the compressed data. The *decoder* receives the compressed data and applies a decoding algorithm to restore the original data, exactly in the case of *lossless* compression, or approximately in the case of *lossy* compression. We say that the encoder *transmits* data to the decoder.

If the compression is *lossless*, the compressor is capable of restoring the exact original data by using a *decoding algorithm* on the compressed data. On the other hand, if the compression is *lossy* the decoding algorithm is usually able to restore a good approximation of the original data. In this thesis we concentrate in lossless compression. We refer to the modules that perform encoding and decoding as the *encoder* and *decoder*, respectively.

We say that the encoder *transmits* data to the decoder. This transmission can be either through time, if the compressed data is stored to be decompressed later,

[1] https://nanoporetech.com/products/promethion

or through space, if the compressed data is transmitted to be decompressed at some other place.

In a nutshell, the main objective of a lossless data compressor is to generate a compressed representation of the data with the smallest possible number of bits, in particular, fewer than the number of bits used for the the original representation of the data. Another objective is to perform the encoding and decoding of the data as efficient as possible, both in terms of speed and memory consumption.

1.2 Multi-channel biomedical signal compression

In Part I of the thesis we study the efficient lossless compression of multi-channel biomedical signals. Many biomedical signals are multi-channel in nature, that is, they are recorded from multiple sensors in parallel (*channels*), such as the electrodes of an electroencephalograph or an electrocardiograph. This induces, in general, correlation between signal samples, either in the same channel or across channels due to, for example, the natural continuity of the signal in time. Signal compression methods seek to exploit these temporal and spatial correlations.

A common strategy, called *predictive coding*, takes advantage of these correlations by predicting each signal sample based on other previously encoded samples. The sample itself is described to the decoder by encoding the *prediction error*, which is the difference between the true sample value and the prediction that both the encoder and the decoder calculate from previously encoded samples [60, 4, 5, 98, 15]. Other methods apply *transformations* to bring the signal into a form in which each transformed channel can be efficiently compressed independently [104, 99, 25]. In the context of low-power embedded systems, predictive coding techniques, and the use of Golomb codes [39], are usually combined due to their algorithmic simplicity, which results in low energy costs.

In predictive coding, the prediction method plays a fundamental role, not only because the compression performance depends directly on how accurate the predictions are, but also because the calculation of the predictions is usually one of the most computationally expensive tasks of the compressor. Among prediction methods, common choices include linear predictors [60, 5, 70, 15], and neural networks [4].

For example, in the compressor *RLS* presented in [15], at each sampling time i a prediction of the i-th sample is calculated for each channel, which is obtained as a weighted average of a set of linear predictions following an *expert advice* [95] scheme. Each prediction is calculated as a linear combination of a specific number of previous samples from the same channel that is being predicted, and from a physically close channel (to take advantage of the spatial correlation). These linear predictors are adaptive; its coefficients are updated sequentially using a *lattice algorithm* [37] as

the signal is compressed. This results in an algorithm that achieves excellent compression results with low latency and low complexity; the time complexity grows linearly with the number of channels and the length of the signal, and the amount of memory required grows linearly with the number of channels.

Another algorithm that has also been used successfully for compression of biomedical signals [55] is the MPEG-4 lossless audio encoding standard (ALS) [49]. This algorithm calculates a linear prediction error for each channel, and then the correlation between channels is exploited by subtracting, from each prediction error of a *target* channel, a linear combination of the prediction errors of a *reference* channel. The signal is divided into blocks, and for each block two passes are made through the data. In the first pass, the set of target-reference channel pairs, and the coefficients for making the linear predictions, are obtained and described to the decoder; the data is encoded in the second pass. The prediction scheme used by this algorithm is significantly more complex than the one proposed in [15], yet it achieves worse compression levels for biomedical signals (EEG and ECG).

On the other hand, *Free lossless audio compression*[1] (Flac) is a popularly used lossless audio compression algorithm, with very low complexity. The compressor uses linear predictive coding, with an efficient recursive method to determine a set of prediction coefficients that fits the signal at hand. However, unlike RLS and ALS, Flac is not specifically designed for signals with more than two channels, resulting in significantly worse compression levels.

In this work we propose a new data compressor for multi-channel biomedical signals, *General Speck Compressor* (GSC), whose general architecture is based on the proposal in [15], which achieves compression ratios similar to the best in the state of art, and, at the same time, is highly computationally efficient.

To build GSC, we use the encoding architecture presented in [15], proposing a new prediction module that is significantly less complex. Specifically, we develop a low complexity multi-channel signal linear prediction algorithm, using only integer arithmetic, which is an extension of a single channel prediction algorithm proposed by Speck in [97]. Furthermore, we propose an efficient implementation, using only integer arithmetic, of the prediction scheme based on expert advice [95], which further reduces the computational cost of calculating a prediction. The construction of GSC is detailed in Chapter 3 of this document. We also evaluate the performance of GSC by comparing it against RLS [15], and the audio compressors Flac and ALS, in a series of publicly available datasets of different multi-channel biomedical signals. We conclude that the proposed new compression algorithm achieves compression ratios very similar to the state of the art compression ratios achieved by RLS, and at the same time reduces execution times by practically half.

[1] https://xiph.org/flac/index.html

In order to achieve an even more efficient compression algorithm, in Chapter 4 we define a series of criteria that we follow to create an optimized version of GSC for a specific type of signal, by strategically choosing a reduced subset of adaptive linear predictors. We also propose other improvements that seek to further reduce the computational cost of the compressor. In Chapter 5, we present a version of GSC optimized for EEG data, called *Optimized Speck Compressor* (OSC), which lowers the computational cost of the compressor by following the criteria and techniques presented in Chapter 4. We evaluate the performance of OSC, and we show that it manages to be almost 6x times faster than GSC, and almost 12x times faster than the RLS compressor, in exchange for a slight decrease in compression performance. Furthermore, despite being adjusted to characteristics of EEG data, OSC achieves competitive compression ratios over other types of signals.

Lastly, we evaluate the trade-off between compression performance and computational efficiency of using different integer coding methods, such as adaptive Golomb coding [39], Simple9 [3], Simple16 [108], and PForDelta [112], for the encoding of the prediction errors generated by OSC. The results show that using faster integer coding methods, such as Simple16 or PForDelta, which are simpler than adaptive Golomb coding, can significantly improve the computational efficiency of the compression algorithm, in exchange of substantially deteriorating the compression performance. Specifically, we estimate that using Simple16 encoding, instead of adaptive Golomb coding, can make the encoder of OSC 14.8% faster, in exchange of deteriorating the compression performance between 13.5% and 17.9%, depending on the compressed dataset.

1.3 DNA sequencing data compression

In Part II of the thesis we turn our focus to the efficient compression of DNA sequencing data. In particular, we investigate compression of DNA data generated by nanopore sequencing technologies stored in the *FASTQ format* [22], which is the standard format used for DNA sequencing data storage.

A FASTQ file stores the result of a sequencing process, which consists of a set of readings of genome fragments, called *reads*. In Figure 1.2 we present an example of a read.

```
@ERR1111170.1
TGACTGTTTGGTGCTGGGTTGCTGCGGATTTAATGGTGGAGTTCTCCATCTTAGGGCGG
+
#''''+<)"!*#$%%%#3!"$*$*#().=)'&($#$%''&-;%&$%)%#,#,#$%&"""""
```

Figure 1.2: Example of a read. The read identifier is in the fist line, followed by the base call sequence in the next line. The 3rd line has a '+' symbol, which can be optionally followed by a copy of the read identifier. The last line has the quality score sequence.

Each read contains an *identifier string*, a *base call sequence* (also referred to as base call string, or, simply read), and a *quality score sequence*. The identifier string is generally a short free text segment, which identifies the read. The base call sequence is a string of letters from the set $\{A, C, G, T, N\}$, where letters A, C, G, and T, represent the nitrogenous bases (base-pairs) of a DNA sequence, and letter N is a special character that represents a nitrogenous base that could not be determined. Finally, the quality score sequence is a string of symbols, of the same length as the base call sequence, where the i-th symbol encodes an estimated probability of the i-th base call being correct.

The characteristics of the base call sequence and the quality score sequence that compose each read vary depending on the technology used to perform the DNA sequencing. For example, the HTS technologies in most common use today are *second-generation sequencing (SGS)* technologies, which produce short reads (a few hundreds base-pair long) generally of fixed length. For these technologies, the quality of the readings is generally high, and quality scores have little correlation to the base call sequence if any. The alphabet of the quality scores in this case ranges from 4 values to about 40, depending on the specific technology. On the other hand, the *Single Molecule Real-Time (SMRT)* sequencing technology, developed by Pacific Biosciences (PacBio), is different in the following sense: it produces long reads with comparatively high error rates. Similarly, the recently developed nanopore sequencing technology, mainly driven by *Oxford Nanopore Technologies* (ONT), also generates very long variable length reads (up to hundreds of thousands base-pair long [84]) of relatively low quality. In contrast to other technologies, dependencies between the quality score sequence and the base call sequence have been observed for nanopore sequencing [48]. In addition, the alphabet size of the quality scores is 94 (Sanger format [22] using ASCII codes 33 to 126). Quality score sequences usually dominate the size of a compressed file generated by compressing SGS technologies data [80]. This is also the case for nanopore sequencing data. For example, in our experimental results (see Section 6.3), we show that the quality score sequences amount to 69% of the compressed size, while base call sequences and read identifiers amount to 29% and 2%, respectively, when running SPRING [17], a state of the art FASTQ compressor. Consequently, most of the research in DNA sequencing data compression is primarily focused on the compression of base call and quality score sequences.

Base call sequences are generally compressed losslessly, as the information of the composition of the DNA sequences is of utmost importance for subsequent analysis tasks. On the other hand, although quality scores play an important role in downstream analyses, it is not clear which level of resolution is needed. In the case of SGS data, many studies [107, 82, 2] have shown that having high resolution for the

quality scores is not always beneficial for downstream analyses tasks, such as variant calling. For this reason, lossy compression of quality scores is common in SGS data. In the case of nanopore sequencing data, this phenomenon has not been thoroughly studied yet, although recent results [59] show that high resolution is not necessarily needed. In the case of SGS sequencing data, most of the available FASTQ compression algorithms offer a fully lossless compression mode for base call and quality score sequences, and numerous tools offer an optional lossy mode for quality scores.

Many algorithms have been proposed in the literature and implemented as specific tools for compression of DNA sequencing data in the FASTQ format. Recent surveys are available in [79, 80, 44]. These compression algorithms can be roughly divided into two categories: *reference-based* methods, in which an external *reference genome* is used to aid compression, and *reference-free* methods, in which no external information is used. A reference genome is usually given as a file in *FASTA* format (a variation of the FASTQ format that does not store quality scores [69]), which stores the base call sequences that compose the chromosomes of a genome.

To improve the compression of the base call sequences of the FASTQ file, reference-based methods exploit the information provided by an external reference genome by *aligning* the base call sequences of the FASTQ file against the base call sequences in the reference genome file, producing a series of *alignments*. Loosely speaking, an alignment is a description of a string q in terms of a reference string r, which describes the editing operations needed on the reference string r to produce the string q. If the aligned string q is similar to the reference string r, then the alignment serves as a compact representation of q. Consequently, reference-based compression algorithms efficiently compress a base call sequence q by encoding the alignment against the reference instead. The decoder can later reconstruct the base call sequence q by decoding the alignment and applying the editing operations described in it to the reference base call sequence r, which is assumed available on the decoder side. Reference-based compression methods require access to a suitable reference genome file, that is, a genome of the same or similar species as the one sequenced and stored in the FASTQ file to be compressed. When such reference is available, these methods usually achieve better compression performance (see, e.g., [61, 45, 9, 54, 57, 6, 42, 46, 62]) than reference-free compressors, by exploiting the similarities between the sequenced and the reference genomes, which, for example, in the case of human genomes exceeds 99% of the base-pairs [63].

The availability of a proper reference genome is not an uncommon scenario in Bioinformatics. In fact, many bioinformatic analysis tasks performed on FASTQ data, such as sequence analysis or gene expression analysis, already require a step in their pipeline where the reads of the FASTQ file are aligned against a reference genome [66], using specialized alignment tools. Even for metagenomic or con-

taminated samples, where several organisms that may not be known in advance are sequenced, an appropriate reference can be readily obtained by concatenating the genomes of the most prevalent species identified by a taxonomic classification tool [85, 105, 56].

On the other hand, reference-free compression methods have the advantage of being self-contained and not needing any external resource to work properly. Some reference-free compressors [54, 7, 58, 109, 43, 71, 36, 90, 18] still obtain an artificial reference genome by constructing and encoding a draft assembly from the reads in the FASTQ file. Other technique that is used by some tools [40, 83, 43, 53, 19, 106, 27] consists on reordering the reads in a FASTQ file by base call sequence similarity. This reordering can improve the performance of the compression itself, for example, when running widespread practical compression schemes that exploit local redundancy in the data, for example, schemes based on *LZ77* [110], *LZ78* [111], *Prediction by Partial Matching* [21], or *Burrows–Wheeler transform* [14]. Some tools, such as SPRING [17], offer the option of storing the original ordering of the reads in the compressed file, so that the original file can be exactly recovered after decoding. However, the ordering of the reads in the FASTQ file has no relevant biological information, and therefore it is irrelevant to most of the subsequent analyses tasks.

Finally, there are compressors that do not apply any pre-processing to the data prior to compression. These compressors usually rely on capturing statistical characteristics of the data through *context models*. In a context model, a probability distribution for a data symbol x is estimated, conditioned on the values of other previously encoded symbols, which are referred to as the *context* in which x occurs. On the decoder side, context symbols have been decoded and are available when decoding x, so the same probability distribution for x can be determined in lockstep with the encoder. From this estimated probability distribution, a code for the encoding of x is determined such that symbols with larger estimated probabilities are encoded more compactly than those with smaller estimated probabilities. For example, in DSRC2 [89], the probability distribution estimated for a base call symbol x depends on the nine bases immediately preceding x. Other compressors that make use of context models for base call sequences compression are Fqzcomp and Fastqz [9], and Slimfastq[1].

Regarding the compression of quality score sequences, context models are usually used [61, 9, 89, 73, 35, 78] for both lossless and lossy compression. Fqzcomp [9], in particular, determines a context for each quality score q as a function of the three quality scores immediately preceding q. According to [80], Fqzcomp achieves the best quality score compression performance among an extensive collection of lossless compressors. In terms of lossy compression of quality scores, many algorithms have

[1] https://sourceforge.net/projects/slimfastq/

been proposed. These methods can be roughly divided into two categories: *horizontal* and *vertical* methods. Horizontal methods [81, 61, 24, 73, 17] sequentially compress each sequence of quality scores individually, independently of the rest of the sequences. These methods usually perform some form of quantization to each individual quality score, and have the objective of minimizing a distortion measure, such as the mean squared error. Horizontal methods have the advantage of being reference-free. On the other hand, vertical methods [101, 7, 41] jointly quantize the quality scores of bases that are mapped to the same position in an alignment. Therefore, these methods are generally reference-based, and require a previous data aligning step. The main objective of vertical methods is to minimize the effects of the loss of information on downstream applications, such as variant calling. This is usually done by heavily quantizing regions of high consensus among bases that are mapped to the same position of the reference, and using a higher precision for quality scores of bases with less consensus. In the case of [10], it mixes both strategies.

Most of the compression algorithms introduced above are optimized for SGS data. As such, they obtain their best performance when applied to genomic files containing short reads of fixed length, and many fail to work on data containing reads of variable length, or on data produced by other sequencing technologies [32]. However, data produced by nanopore technologies is becoming increasingly popular, as the long reads have the potential to decrease the ambiguity associated to short reads, and help in the detection of large structural variants, including copy number variants (CNVs), medium- and large-sized insertions and deletions (INDELs), duplications, inversions, and translocations, among others [64, 96, 51].

FASTQ compressors like LFastqC [1], SPRING [17], and Genozip [62], offer support for compression of long variable length reads generated by nanopore sequencing. In the case of LFastqC, the authors report the compression performance of LFastqC for two nanopore FASTQ files. However, this compressor failed in most of the datasets that we tested. In the case of SPRING, the long read compression support is included for completeness, and it is not a specific target of the tool, which uses mainly the general lossless compressor BSC[1] in this case. The more recent work Genozip presents a compression tool capable of compressing nanopore FASTQ files, which offers both a reference-free mode and a reference-based mode. However, our experiments have shown that, when running both methods on nanopore FASTQ data, the reference-free mode consistently outperforms the reference-based mode, and the reference-based mode fails to compress some of the tested datasets (results of these experiments are presented in Chapter 7).

In this thesis we concentrate in lossless compression for nanopore sequencing data. We leave lossy compression of nanopore quality scores as an interesting chal-

[1] http://libbsc.com/

lenge for future work. In this sense, we present two novel lossless compressors for FASTQ files generated by nanopore sequencing technologies. First, in Chapter 6, we introduce ENANO, which is a lossless reference-free compressor especially designed for nanopore sequencing FASTQ files, which mainly focuses on the compression of the quality score sequences. As we previously mentioned, the quality scores dominate the size of compressed FASTQ files. Specifically, to compress the quality score sequences, ENANO uses a context model algorithm combined with arithmetic coding [86]. The context model used for encoding each quality score is determined as a function of the two previously compressed quality scores and a set of surrounding base-pairs (6 by default), which exploits the statistical dependencies between the base call and quality score sequences. ENANO offers two modes, *Maximum Compression* and *Fast*, which trade-off compression efficiency and speed.

We test ENANO, SPRING, and the general compressor pigz (used as a baseline reference), on several publicly available nanopore datasets. The results show that the proposed algorithm consistently achieves the best compression performance (in both modes) on every considered nanopore dataset, with an average improvement over pigz and SPRING of 24.7% and 6.3%, respectively. In addition, in terms of encoding and decoding speeds, ENANO is 2.9x and 1.7x times faster than SPRING, respectively, with memory consumption up to 0.2 GB.

Next, in Chapter 7, we introduce RENANO, a reference-based lossless FASTQ data compressor, which is specifically tailored to compress FASTQ files generated with nanopore sequencing technologies. RENANO builds on the compressor ENANO by improving the compression of the base call sequence portion of the FASTQ file, and leaving the other parts of ENANO intact. Two novel reference-based compression algorithms are introduced, contemplating different scenarios: in the first scenario, a reference genome is available without cost to both the compressor and the decompressor; in the second, the reference genome is available only on the compressor side, and a compacted version of the reference is transmitted to the decompressor as part of the compressed file.

To evaluate the proposed algorithms, we compare RENANO against ENANO, and to the compressor Genozip, on several publicly available nanopore datasets. In the first scenario considered, RENANO improves the base call sequences compression of ENANO by 39.8%, on average, over all the datasets. As for total compression (including the other parts of the FASTQ file), the average improvement is 12.7%. In the second scenario considered, the base call compression improvements of RENANO over ENANO range from 15.2% to 49.0%, depending on the coverage of the compressed dataset, while in terms of total size, the improvements range from 5.1% to 16.5%. We also show that RENANO consistently outperforms the compressor Genozip in both scenarios.

At the time of writing this thesis document, two new pre-prints with compression methods for nanopore sequencing data were published, NanoSpring [74] and CoLoRd [59]. In the case of NanoSpring, the authors propose a reference-free compression method only for base call sequences, and compare it against both ENANO and RENANO. The results show that NanoSpring achieves better compression of base call sequences when compared against ENANO, but it is significantly outperformed by RENANO. Also, in terms of computational efficiency, both ENANO and RENANO are faster than NanoSpring and consume less memory. CoLoRd is a state-of-the-art reference-free full FASTQ compression algorithm with a quality score compression module inspired by ENANO, and an optional reference-based mode.

1.4 Contributions

Most of the work detailed in this document was developed within the framework of a series of research projects funded by *Comisión Sectorial de Investigación Científica* of the Universidad de la República.

Regarding the compression of multi-channel biomedical signals, the main results of this work were published in [31] (specifically in sections III and V) with a significantly lower level of detail than in this document. In [31] there is also a description of hardware implementations of the algorithms, which were developed by other team members. The idea of using Speck's prediction algorithm as a starting point for the construction of the General Speck Compressor was promoted by Dr. Gadiel Seroussi, at the beginning of the project. The use of fixed predictors described in Section 5.3 of this thesis was also developed from preliminary ideas within the project team. The author of this document had a leading role in the design, implementation and evaluation of the other algorithms, which are presented in Part I of this thesis.

Regarding the compression of nanopore sequencing data, preliminary results of this work were published in [32], and the main results in [33] and [34]. In these works, Drs. Idoia Ochoa and Gadiel Seroussi participated in technical discussions related to the design and implementation of the compression algorithms. Drs. Pablo Smircich and José Sotelo-Silveira contributed with the validation of biological aspects, and the applicability of the algorithms. The author of this thesis had a leading role in the design, implementation and evaluation of all the algorithms presented in Part II.

1.5 Summary of document structure

The rest of this document is organized as follows. In Chapter 2 we present fundamental concepts behind data compression, giving a brief overview of code building

and information theory, and we present some data compression techniques that are used throughout the document. Next, we begin Part I by presenting the construction of GSC in Chapter 3. In Chapter 4, in search of the development of an even more efficient compressor, we study the performance of the various predictors used by GSC on different types of signals. We conclude a series of criteria to select a reduced set of predictors that allow the compressor to perform well on a specific type of signal. Part I ends with Chapter 5, where we introduce OSC, a version of GSC optimized for EEG, using the criteria developed in Chapter 4.

Part II of the thesis begins with Chapter 6, where we present ENANO, a lossless reference-free nanopore sequencing FASTQ data compressor, that achieves state of the art compression performance by exploiting the correlation between base call and quality score sequences. Then, in Chapter 7, we present RENANO, which expands ENANO with two novel reference-based compression algorithms for compressing base call sequences.

Finally, in Chapter 8, we present some conclusions obtained from this work, and we propose some directions for future lines of work.

Chapter 2

An introduction to data compression

In this chapter we review some fundamental concepts on which data compression relies. The objective is to give a brief overview about the construction of codes and Information Theory, which allows us to understand the data compression techniques that we will use throughout the document.

Informally, *lossless data compression* consists of converting some input data (a source file, a sequence of bits) into a smaller representation, in such a way that we can later reverse the process and recover the original data. There are multiple methods for compressing data, which are suitable for different scenarios. However, they are all based on the same principle, that is, they all compress data by removing *redundancy*. This redundancy is manifested through some type of underlying statistical structure, which can be exploited to obtain a more compact representation of the data.

For example, in natural language we can easily see that some letters appear more frequently than others; in English for instance, the most common letter is the letter E. It is no coincidence that the *Morse code*, used to encode the letters of the alphabet using the symbols *dot* and *dash*, assigns the shortest possible code (a single dot) to the letter E. This is because Morse code exploits the redundancy that exists in the language by assigning shorter codes to more frequent letters (E, T, I), and longer codes to less frequent letters (Q, X, Z), in order to transmit as few symbols as possible for a typical message.

The intuitive idea of encoding most probable events with less symbols and rare events with more symbols has been used throughout history to compress data. However, it was not until 1948, when Claude Shannon presented his work *A Mathematical Theory of Communication* [94], that a mathematical theory was developed, called *Information Theory*, which establishes the theoretical bases for data compression. Next we present some of the most important concepts and conclusions reached by

Shannon in his work. Specifically, in Section 2.1 we formalize the problem and give a brief introduction to coding and compression. Then, in Section 2.2 we present the *adaptive coding* technique in the context of statistical modeling, and discuss the implications of model complexity. Finally, in Section 2.3 we give a summary of the concepts presented throughout the chapter.

2.1 Information sources and codes

Let us consider an *information source* that produces messages to be transmitted through a communication system. In the context of this thesis, we think of such information source as a system that produces messages, symbol by symbol, where each symbol is drawn randomly from a discrete alphabet $\mathcal{X}$, according to a certain probability distribution that may depend, in general, on previously generated symbols. In other words, we model an information source as a discrete-time discrete-state *stochastic process*. Depending on the characteristics of the system, we can choose among different classes of stochastic processes. For example, in a *memoryless* source, we model the symbols as generated independently, randomly, and identically distributed (i.i.d). If successive symbols are presumably related to each other, the sequence can be modeled as a Markov process.

Now, suppose we want to encode the messages generated by an information source as a sequence of binary symbols to be transmitted through a communication channel. A *code*, $\mathcal{C} : \mathcal{X} \to \{0,1\}^*$, assigns a *code word*, which is a finite binary string (a sequence of *bits*) to each symbol of the *source alphabet*, $\mathcal{X}$. Naturally, we want the code to be *uniquely decodable*, that is, that any binary sequence that arises from a concatenation of code words, $\mathcal{C}(x_1)\mathcal{C}(x_2)...\mathcal{C}(x_n)$, uniquely determines the original sequence of source symbols, $x_1, x_2, ..., x_n$. In this sense, a simple way of building a uniquely decodable code is by building a *prefix-free code*, where no code word is a prefix of another code word. It is easy to see that a prefix-free code is uniquely decodable, as decoding a bit stream amounts to reading the stream progressively, from left to right, replacing each code word found with its corresponding symbol. As an example, suppose we have a source alphabet $\mathcal{X} = \{x_1, x_2, x_3\}$. The code defined as $\mathcal{C}(x_1) = 0$, $\mathcal{C}(x_2) = 10$, $\mathcal{C}(x_3) = 11$, is prefix-free and, therefore, it is uniquely decodable. Although there are uniquely decodable codes that are not prefix-free, for any of them there is always another code with the same code lengths that is prefix-free. Therefore, in practice, the codes that are generally used are prefix-free (see, e.g., [23]).

For data compression we are interested in codes with an average code word length as small as possible. In other words, given an information source, we want to build

a uniquely decodable code $\mathcal{C}$ that minimizes the expected length of $\mathcal{C}(X)$,

$$\mathrm{E}_p\big[l(X)\big] = \sum_{x\in\mathcal{X}} p(x)l(x), \tag{2.1}$$

where $\mathrm{E}_p\big[\,\cdot\,\big]$ denotes expectation with respect to the probability distribution p, and $l(x)$ denotes the length of $\mathcal{C}(x)$.

In this context, there are some questions that naturally arise. Given the probabilistic law that governs an information source, what is the minimum of (2.1)? Is it possible to build a code that achieves this minimum?

Shannon showed that the *entropy* of a source is the fundamental limit for data compression. For a random variable X, which takes values on the discrete alphabet $\mathcal{X}$, with a probability distribution p, the *entropy* of X is defined as

$$H(X) = \mathrm{E}_p\big[-\log_2 p(X)\big] = -\sum_{x\in\mathcal{X}} p(x)\log_2 p(x), \tag{2.2}$$

where we let $p(x)\log_2 p(x) = 0$ for $p(x) = 0$. The entropy is a lower bound for the expected number of bits required to describe an occurrence of X. In other words, every uniquely decodable code, $\mathcal{C}(X)$, satisfies $H(X) \le \mathrm{E}_p\big[l(X)\big]$.

Notice that $H(X)$ only depends on the probability distribution of X but not on the specific values of the elements of the alphabet. For example, Figure 2.1 shows the entropy of a random variable X, which takes values on a binary alphabet $\mathcal{X} = \{x, x'\}$ with probabilities $p(x) = \theta$, and $p(x') = 1-\theta$, as a function of θ.

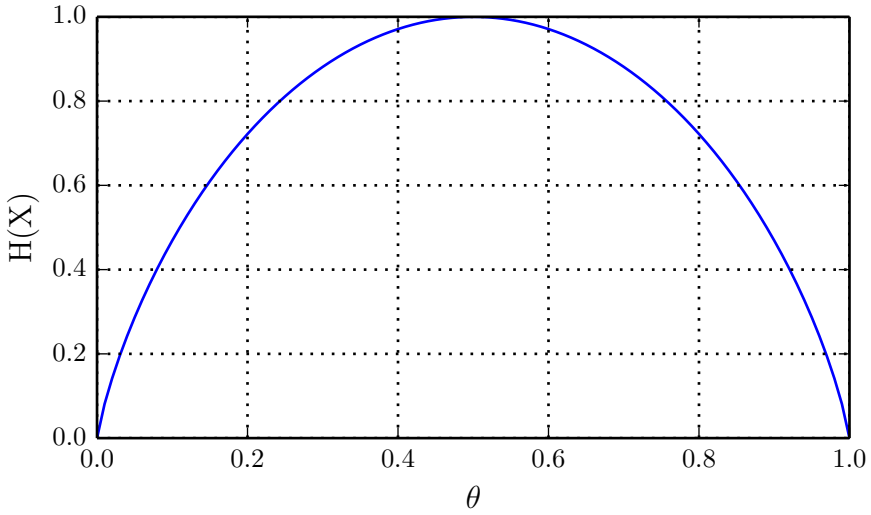


Figure 2.1: Entropy of random variable X that takes two possible values with probabilities θ and $1-\theta$, respectively.

The maximum value of the entropy occurs when the events are equally likely and

it descends to zero as the probability mass concentrates in one of the two events. In fact, this phenomenon generalizes to larger finite alphabets. For a random variable X that takes values on a finite alphabet $\mathcal{X}$, of size $|\mathcal{X}|$, with a probability distribution p, the maximum value of $H(X)$ is $log_2(|\mathcal{X}|)$, which is attained when p is uniform, and we get to $H(X) = 0$ as the probability mass concentrates in a single symbol x with $p(x) = 1$. Intuitively this makes sense, as when most of the probability is concentrated in a small number of symbols, we can assign short code words to those specific symbols, thus decreasing the overall expected code length.

However, even if we know in advance the probability distribution that governs a source of information, it is not immediately apparent how to design a code that achieves an expected length close to the entropy of the source. Consider the following example. We toss a coin in the air as many times as necessary until we get the first tails and we record how many heads in a row came out. We set ourselves the objective to design a uniquely decodable code to encode the number of consecutive heads obtained, using the least number of bits on average. Intuitively, we propose the following code: write a 0 each time the coin flip is heads, and write a 1 when we get tails. Then, decoding the number of heads amounts to counting the number of *zeros* until we observe the first *one*. For example, if 3 heads came out in a row and then tails, the corresponding code word is 0001. It is easy to see that this code is prefix-free, and therefore uniquely decodable, since all the code words have a single one at the end and they all have a different length. In fact, the proposed code is known as *unary encoding*, an encoding that assigns to each natural n a code word that is comprised of n consecutive zeros and a trailing one, so that the length of the code word is $l(n) = n + 1$. Now we ask, is this unary code a good code to use in this example? To find an answer using the concepts introduced by Shannon, we have to define a probabilistic model for the information source. In this case, we model the experiment as a random variable X that takes values over the set of natural numbers, $\mathbb{N}$, with a probability distribution p, where $p(n)$ is the probability of obtaining n heads in a row and then tails. If the coin is balanced, that is, the probability that the coin flip is heads is equal to the probability that it is tails, then $p(n) = (1/2)^{n+1}$. Hence, we have $-\log_2 p(n) = n + 1 = l(n)$, so from equations (2.2) and (2.1) we obtain $H(X) = \mathrm{E}_p\big[l(X)\big]$, which implies that the unary code is optimal in this case.

Notice that in this example we are able to verify the optimality of the proposed code based on the assumed perfect knowledge of the probabilistic law that governs the information source. In fact, given a probabilistic model, there are coding schemes such as *Shannon coding* [94], *Huffman coding* [47], or *arithmetic coding* [86], which yield close to optimal codes. However, in most real data compression scenarios, including the ones we address in this thesis, we do not have access to an exact

probability distribution for the data in advance. Consequently, designing a good data compressor amounts, in essence, to estimating a good statistical model for the information source, and combining it with a proper coding scheme.

2.2 Statistical modeling and adaptive coding

Statistically modeling an information source may be a complex task. A common approximation to this problem is to make a general assumption of the statistical model structure, based on prior knowledge or experimental observations letting some model parameters undefined, which are estimated from the data at compression time. For example, let us return to the *coin toss* example, and suppose that the coin may be *unbalanced*, that is, that one of the outcomes, heads or tails, may be more likely than the other. Although we do not know exactly how unbalanced the coin is, we can generally model an unbalanced coin toss by considering the probability of getting heads as a parameter, θ, $0 < \theta < 1$. Even if the value of θ is a priori unknown, the model seems to correctly capture the statistical structure of the unbalanced coin toss. With this model, the experiment of tossing the coin in the air as many times as necessary until we get the first tails, and then recording how many heads in a row came out, can be modeled as a random variable X that takes values over the naturals, with a *geometric probability distribution*, p, parameterized by θ, where $p(n) = \theta^n(1-\theta)$, $n \in \mathbb{N}$. Note that the example of the balanced coin is a specific case where the parameter θ is $1/2$. Figure 2.2 graphically shows the geometric distribution over the natural numbers for $\theta = 3/4$. The geometric distribution is characterized

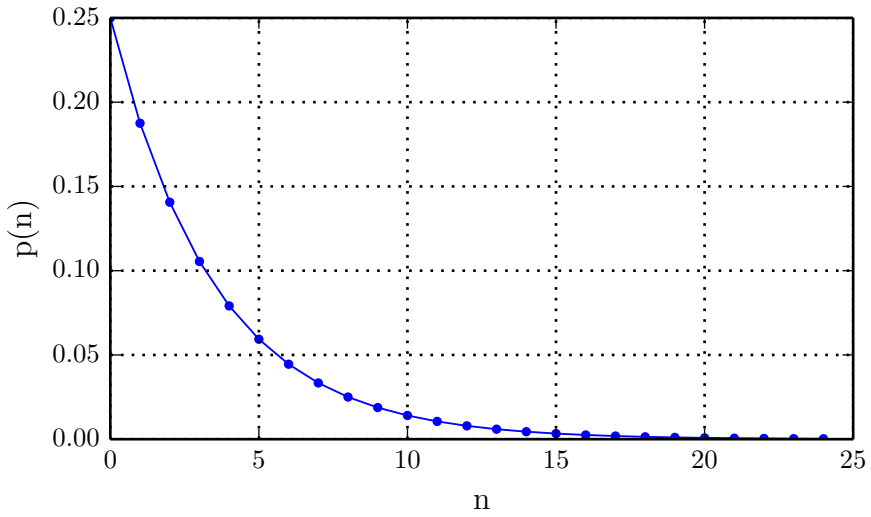


Figure 2.2: Geometric probability distribution over natural numbers for $\theta = 3/4$.

by being strongly concentrated in a few values (the naturals closest to 0), which,

a priori, tells us that the information source is highly compressible. For arbitrary values of θ the construction of an optimal code is obtained using *Golomb codes* [39].

In general, once a statistical model is established for the data, we need to adjust the parameters of the model to the specific data we want to compress. In this sense, there are different strategies we can follow. In a *two-pass scheme*, the data is first scanned to estimate the values of the parameters that best adjust to the data. This parameter values are encoded and then, in a second pass, the data itself is encoded using a coding scheme adjusted to those parameters. For example, for the encoding of consecutive heads count in successive repetitions of the unbalanced coin toss experiment, in a first pass through the data we can empirically determine the total number of heads, n_h, and of tails, n_t, and calculate an estimation of θ as $\hat{\theta} = \frac{n_h}{n_h+n_t}$, which is encoded in first place. Then, in a second pass, we encode each heads count using a Golomb code that is optimal for the value $\hat{\theta}$. This strategy can yield good results, but has the disadvantage of requiring access to the data in advance.

On the other hand, in *adaptive coding schemes* the parameters of the model are sequentially adjusted as the data is being encoded/decoded. For example, in the unbalanced coin toss example, during encoding/decoding we can adaptively maintain an estimator, $\hat{\theta}(i)$, which is calculated from the outcomes up to the i-th run of heads. Then, when encoding/decoding the $(i+1)$-th run, we use an optimal Golomb code for the value $\hat{\theta}(i)$, which is estimated by both the encoder and the decoder. This scheme has the advantage of being capable of adapting to statistical changes in the data, as well as being compatible with online compression applications.

The complexity of the chosen statistical model plays a crucial role, as it determines the number of parameters that need to be adjusted. Larger models can potentially capture more complex statistical dependencies than simpler models, which may result in better compression performance. However, since the model parameters are adjusted from the same data that is compressed (simultaneously), large models may suffer from a large *model cost* [88], which may render a poor compression performance in small data sets. Additionally, implementations of large complex models can have demanding memory requirements. Consequently, designing an efficient compression algorithm requires finding a balance in the trade-off between under and over-fitting of the underlying probabilistic model.

2.3 Summary

In this chapter we briefly explored the basic concepts behind data compression, including the concept of an information source characterized by a probability distribution over an alphabet, and the concept of code. We explained that there is a limit

for data compression called *entropy*, which depends on the probability distribution that governs the information source, and we saw that there are codes with expected code lengths very close to entropy. We also observed that, intuitively, when the probability mass concentrates in few elements of the alphabet, the entropy is low and thus the data can be highly compressed. Finally, we discussed the importance of designing a good statistical model for the data, and how model complexity can have implications on the efficiency of adaptive coding algorithms.

In the rest of the thesis we use the principles described in this chapter as a basis for modeling and compressing data in the different compression scenarios that we investigate.

Part I

Multi-channel biomedical signals data compression

Chapter 3

General Speck Compressor

In the first part of the thesis we investigate the efficient compression of multi-channel biomedical signals in a scenario where processing resources are scarce, and there are severe restrictions on energy consumption. In this sense, in this chapter we present the construction of an efficient multi-channel signal compressor, the General Speck Compressor (GSC), which is based on integer arithmetic, and which combines various strategies to achieve compression performance comparable with the state of the art, while being computational efficient.

The rest of the chapter is organized as follows: in sections 3.1 and 3.2 we describe the general compression techniques in which our compressor is based on. Specifically, in Section 3.1 we describe the *sequential predictive coding scheme* in the context of multi-channel signals, and in Section 3.2 we explain the *expert advice* technique for combining the predictions from a set of predictors into a single prediction. Next, in Section 3.3 we describe the RLS algorithm presented in [15], which uses predictive coding and expert advice to perform multi-channel signal compression, to achieve state of the art compression performance. Taking this algorithm as a starting point, in Section 3.4 we present GSC and describe an efficient implementation of it. Finally, in Section 3.5, we evaluate the performance of GSC in terms of compression and speed, by comparing it to other compressors for multi-channel signals, on a series of publicly available multi-channel signal datasets.

3.1 Sequential predictive coding for multi-channel signals

Multi-channel signals arise in various scenarios where samples are obtained simultaneously from multiple sensors with certain frequency along time. Some examples are the signal recorded by a set of seismographs, the signal recorded by a set of microphones in a recording room, and the signal recorded by multiple electrodes

on an EEG system. Let us picture the following example: suppose a patient in a hospital has 3 sensors, s_1, s_2, s_3, attached to their body, which monitor their body temperature, blood oxygen level, and blood pressure, respectively. If the sensors perform synchronized measurements, in regular time intervals, we can model the sequence of values obtained by the sensors as a 3-channel signal, where the measurements obtained by each sensor correspond to a channel, whose signal consists of the sequence of measurements made.

Generalizing, when we have a signal of C discrete time channels, $C > 1$, we denote the (scalar) sample obtained from the i-th channel, at time n, as $x_i(n)$, $n \geq 1$, and we refer to the vector $(x_1(n), \ldots, x_C(n))$ as the *sample vector* at instant n. Also, we define $\mathbf{x}_i(n) = x_i(1), \ldots, x_i(n)$ as the sequence comprised of the first n samples from channel i. We assume that all scalar samples are quantized to an integer value in a finite interval $\mathcal{X}$.

Returning to the example, we have that the signal obtained by the sensor s_1 from time 1 to time 10, that is, the sequence of recorded temperatures, is $\mathbf{x}_1(10) = x_1(1), \ldots, x_1(10)$, while the vector sample, i.e., the temperature, blood oxygen level, and blood pressure, at time 10, is $(x_1(10), x_2(10), x_3(10))$.

In this context, *sequential predictive coding* is a scheme that has been proven effective for efficiently encoding multi-channel signals. We describe how it works next.

In the sequential predictive coding scheme, as its name indicates, each time a vector sample is obtained, each of its scalar samples $x_i(n)$ is encoded from a prediction $\hat{x}_i(n)$. The prediction $\hat{x}_i(n)$ is calculated sequentially for each sample $x_i(n)$, and this sample is described to the decoder through the encoding of the *prediction error*, $\epsilon_i(n) \triangleq x_i(n) - \hat{x}_i(n)$. The sequence of sample descriptions is *causal*, that is, the order in which the samples are described and the definition of the prediction $\hat{x}_i(n)$ are such that $\hat{x}_i(n)$ only depends on samples that are encoded before $x_i(n)$. Then, a decoder can sequentially calculate the prediction $\hat{x}_i(n)$ from the previous samples, decode $\epsilon_i(n)$, and add both values to reconstruct the original sample $x_i(n)$.

Encoding the prediction error $\epsilon_i(n)$, instead of directly encoding the sample $x_i(n)$, has the benefit that if the predictions are accurate, the absolute values of the prediction errors tend to be close to zero, with histograms that resemble two-sided geometric distributions. Therefore, the sequence of prediction errors can be highly compressed using *adaptive Golomb codes* [39], which have been proven to be optimal for two-sided geometric distributions [75].

The prediction $\hat{x}_i(n)$ is calculated as a function of previously encoded samples, that is

$$\hat{x}_i(n) = f_i(\mathbf{x}_1(n), ..., \mathbf{x}_{i-1}(n), \mathbf{x}_i(n-1), ..., \mathbf{x}_C(n-1)), \tag{3.1}$$

where, assuming that the channels are numbered in the order in which they are encoded, the prediction is calculated from the sequences of samples $\mathbf{x}_j(n)$ of all channels j with $j < i$, and the sequences of samples $\mathbf{x}_j(n-1)$ of the channels j such that $i \leq j \leq C$. Thus, for calculating $\hat{x}_i(n)$ we make use of all the information that is available to both the encoder, and the decoder, at the time of encoding/decoding $x_i(n)$. Algorithm 1 describes the general sequential predictive coding scheme.

Algorithm 1: General multi-channel sequential predictive coding scheme for sequences of N samples from C channels.

Input : Sequences of N samples from C channels, $\mathbf{x}_i(N)$, where $1 \leq i \leq C$
Output: Encoding of the prediction errors $\epsilon_i(n)$, where $1 \leq i \leq C$, and $1 \leq n \leq N$

```
for n = 1, ..., N do
    for i = 1, ..., C do
        x̂_i(n) = f_i(x_1(n), ..., x_{i-1}(n), x_i(n-1), ..., x_C(n-1))
        ε_i(n) = x_i(n) − x̂_i(n)
        Encode ε_i(n)
    end
end
```

Returning to the patient example, in time n a vector sample is obtained $(x_1(n), x_2(n), x_3(n))$. If we encode the samples in the order that the channels are numbered, to encode $x_2(n)$ the prediction error $\epsilon_2(n)$ is calculated from the prediction $\hat{x}_2(n) = f_2(\mathbf{x}_1(n), \mathbf{x}_2(n-1), \mathbf{x}_3(n-1))$. Note that, since channel 1 is encoded before channel 2, the decoder has access to the sample $x_1(n)$ at the time of decoding $x_2(n)$, even though both scalar samples belong to the same vector sample. Later, in Section 3.3, we show that using samples that belong to the same vector sample, that is, samples from the present time, can significantly improve the predictions and, thus, achieve better compression results.

3.2 Prediction with expert advice

The concept behind the expert advice prediction method is based on a process that we naturally go through when having to make a decision. Occasionally, when we are in doubt about choosing among several options, we resort to consulting different experts, which give us advice on what to choose. Then we evaluate the various responses and make a decision. Usually, the opinions of the experts carry different

weights for us, as we tend to give more importance to those opinions that come from sources that we believe are more qualified to weigh on the matter. Expert advice poses an analogous situation, where the decision we need to make is what to predict for the next sample $\hat{x}_i(n)$. To do this, we consult a set of predictors (the experts), obtaining the prediction that each one would make. Once the predictions are in place, they are weighted averaged to obtain the final prediction, where the weight of each prediction depends on the previous success rate of the corresponding predictor (how qualified the expert is).

Formalizing this concept, if we have a finite set of predictors $\mathcal{P}$, the expert advice method proposes to calculate the prediction $\hat{x}_i(n)$ as a weighted average of the predictions $\dot{x}_i^p(n)$ of each predictor p of the set. Specifically, we define the prediction of sample $x_i(n)$ of channel i at time n as,

$$\hat{x}_i(n) = \left\lfloor \frac{1}{W_i(n)} \sum_{p \in \mathcal{P}} \mu_i^p(n) \dot{x}_i^p(n) \right\rceil, \tag{3.2}$$

where $\lfloor \cdot \rceil$ denotes rounding to the nearest integer within the quantization interval $\mathcal{X}$, $\mu_i^p(n)$ is a decreasing exponential function of the sequential quadratic prediction errors accumulated up to time instant $n-1$ for the predictor p of channel i, and $W_i(n) = \sum_{p \in \mathcal{P}} \mu_i^p(n)$ is a normalization factor. If the weights $\mu_i^p(n)$ are exponential functions of the sequential quadratic prediction errors, the authors in [95] demonstrated that the quadratic prediction error per-sample of this predictor is asymptotically as small as the minimum quadratic prediction error among all predictors belonging to $\mathcal{P}$.

In other words, expert advice allows us to asymptotically achieve a performance as good as the performance we would have obtained if we had known in advance which of the predictors of the set is the one that obtains the best average prediction accuracy. It is important to note that the greater the number of predictors, the slower the rate of convergence of the performance of the general predictor by expert advice to that of the best individual predictor. In addition, the greater the number of predictions, the greater the complexity of the algorithm, which results in worse computational efficiency.

3.3 An initial approach to compression of multi-channel biomedical signals

In this section we describe the multi-channel biomedical signals compression algorithm RLS [15], which we take as a starting point for the construction of GSC. This algorithm achieves excellent compression results on EEG and ECG data, while being

computationally efficient. Next we explain how it works.

The RLS compression algorithm combines sequential predictive coding with experts advice. In Section 3.1, we generally defined the prediction, $\hat{x}_i(n)$, of the sequential predictive coding scheme, as a function of all previously encoded samples. It usually happens that the further two samples are separated in time, or the more separated in space are the sensors that recorded them, the lower the correlation between them, which makes each of little help to predict the other. For this reason, in order to achieve simple models, it is beneficial to use predictors that predict from a certain limited number of nearby samples in time and space. When a predictor uses samples from up to d previous sampling times it is said to be a predictor of *order* d.

In [15] the authors propose to use for each channel i, $1 \le i \le C$, a set of predictors $\mathcal{P}$, which are combined with the expert advice method. For $p \in \mathcal{P}$, we let $\hat{x}_i^p(n)$ be a linear prediction of some *finite order*, denoted d_p, which linearly combines: the most recent d_p samples of channel i; the most recent d_p samples of a channel ℓ recorded by a sensor physically close to the sensor of channel i, which is called the *parent* or *helper* channel of i; and the *current* sample of channel ℓ, which is encoded before $x_i(n)$. Hence, the prediction $\hat{x}_i^p(n)$ is defined as,

$$\hat{x}_i^p(n) = \sum_{k=1}^{d_p} a_{i,k} x_i(n-k) + \sum_{k=0}^{d_p} b_{i,k} x_\ell(n-k)\,, \tag{3.3}$$

where $1 \le i, \ell \le C$, $i \ne \ell$, and $a_{i,k}$ and $b_{i,k}$ are real coefficients.

From experimental tests carried out by adjusting the coefficients using least squares, the authors of [15] observe that a predictor of order d_p that uses an appropriate helper channel, together with the present time sample, yields similar or even better results than a predictor of order $2d_p$ that makes its prediction using the past samples from *all* the channels. This suggests that by taking advantage of the samples from a physically close channel, and from its present sample, a large part of the spatial and temporal correlation between channels can be captured.

As the sequence of encodings must be causal with respect to the predictor, not all predictions $\hat{x}_i(n)$ can depend on a sample from time n, so an encoding order that meets the causality constraint must be defined for the samples. The authors propose a way of ordering the channels that, in addition to satisfying the causality restriction, minimizes the sum of the physical distances between the electrodes of the channels i and ℓ, where ℓ is the helper of i, in all channels i, except for the channel whose sample is the first to be described. The rationale behind this is that the correlation between the signals from two electrodes tends to increase as the physical distance between them decreases.

To define an order in which to describe the channels, the authors propose to use a

tree, T, whose set of vertices is the set of channels, $\{1, \ldots, C\}$. An arbitrary channel r is distinguished as *root*, and an orientation is assigned to the edges of T such that there is a single directed path from r to each vertex of T. Since a tree has no cycles, the edges of T induce a causal sequence of sample descriptions, for example, by ordering the edges of T, $e_1, \ldots, e_{C-1}$, in order of depth. T is called the *encoding tree*. Specifically, in the context where the electrode positions are known, T is the *minimum spanning tree* of the complete graph whose set of vertices is $\{1, \ldots, C\}$, and each edge (i, j) is weighted with the physical distance between the electrodes of the channels i, j. In other words, the sum of the distances between the electrodes of the channels i, j, on all the edges (i, j) of T, is minimum among all possible coding trees. Note that, since T depends on the acquisition system but not on the signal samples, this order of description can be determined beforehand. The sample $x_r(n)$ is predicted from the samples up to time $n-1$ of the channels r, i, where (r, i) is the edge e_1; all other predictions, $\hat{x}_i(n)$, $i \neq r$, depend on the sample at time n of the channel ℓ and the past samples of the channels ℓ, i, where (ℓ, i) is an edge of T. Algorithm 2 summarizes the encoding process proposed in [15].

Algorithm 2: Encoding algorithm proposed in [15] with an encoding tree for sequences of N samples from C channels.

Input : Sequences of N samples from C channels, $\mathbf{x}_i(N)$, where $1 \leq i \leq C$, and the encoding tree T
Output: Encodings of the prediction errors $\epsilon_i(n)$, where $1 \leq i \leq C$, and $1 \leq n \leq N$

1 **for** $n = 1, \ldots, N$ **do**
2 Let (r, i) be the edge e_1 of T
3 $\hat{x}_r(n) = f_r(\mathbf{x}_r(n-1), \mathbf{x}_i(n-1))$
4 $\epsilon_r(n) = x_r(n) - \hat{x}_r(n)$
5 Encode $\epsilon_r(n)$
6 **for** $k = 1, \ldots, C-1$ **do**
7 Let (ℓ, i) be the edge e_k of T
8 $\hat{x}_i(n) = f_i(\mathbf{x}_i(n-1), \mathbf{x}_\ell(n))$
9 $\epsilon_i(n) = x_i(n) - \hat{x}_i(n)$
10 Encode $\epsilon_i(n)$
11 **end**
12 **end**

To complete the description of the encoder proposed in [15], we must define the prediction functions, f_i, $1 \leq i \leq C$, which are used in the steps 3 and 8 of Algorithm 2.

For a predictor p, of order d_p, the authors define $\mathbf{a}_i^p(n) = \{a_{i,k}(n), b_{i,k}(n)\}$ as the set of coefficients, $a_{i,k}$, $b_{i,k}$, which, when substituted in (3.3), minimize the total

quadratic prediction error up to time n,

$$E_i^p(n) = \sum_{j=1}^{n} \lambda^{n-j} \Big(x_i(j) - \hat{x}_i^p(j) \Big)^2 , \tag{3.4}$$

where λ, $0 < \lambda < 1$, is an *exponential decay factor*. This parameter has the effect of preventing $E_i^p(n)$ from growing without limit with n, and of assigning a greater weight to more recent samples, which makes the prediction algorithm adapt more quickly to statistical changes of the signal.

A *sequential linear predictor* p, of order d_p, uses the coefficients $\mathbf{a}_i^p(n-1)$ to predict the value of the sample at time n as

$$\dot{x}_i^p(n) = \sum_{k=1}^{d_p} a_{i,k}(n-1) x_i(n-k) + \sum_{k=0}^{d_p} b_{i,k}(n-1) x_\ell(n-k) , \tag{3.5}$$

and, after observing $x_i(n)$, updates the set of coefficients from $\mathbf{a}_i^p(n-1)$ to $\mathbf{a}_i^p(n)$, and proceeds to perform the next sequential prediction.

This determines a total weighted *sequential absolute prediction error* defined as

$$\mathcal{E}_i^p(n) = \sum_{j=1}^{n} \lambda^{n-j} \Big| x_i(j) - \dot{x}_i^p(j) \Big| . \tag{3.6}$$

Note that each prediction $\dot{x}_i^p(j)$ in (3.6) is calculated with a set of model parameters, $\mathbf{a}_i^p(j-1)$, which only depends on the samples that are described before $x_i(j)$ in Algorithm 2. These model parameters vary, in general, with j (note the difference with (3.4) where the coefficients involved in the calculation of $\hat{x}_i^p(j)$ are fixed).

The authors propose to use a *lattice algorithm* [37] to efficiently calculate $\mathbf{a}_i^p(n)$ from $\mathbf{a}_i^p(n-1)$ simultaneously *for all* possible model orders d, up to a predefined maximum order P. Thus, in the proposed encoding algorithm, the set of experts $\mathcal{P}$ is defined as the set formed by all the adaptive sequential linear predictors p, whose orders d_p satisfy that, $0 \le d_p \le P$. The final prediction, using expert advice, is then defined for $i \neq r$ using the prediction function

$$f_i(\mathbf{x}_i(n-1), \mathbf{x}_\ell(n)) = \left\lfloor \frac{1}{W_i(n)} \sum_{p \in \mathcal{P}} \mu_i^p(n) \dot{x}_i^p(n) \right\rfloor , \tag{3.7}$$

with

$$\mu_i^p(n) = \exp\Big\{ -\frac{1}{c} \mathcal{E}_i^p(n-1) \Big\} , \tag{3.8}$$

where $\mathcal{E}_i^p(n-1)$ is defined in (3.6), and c is a constant that depends on the quantization interval $\mathcal{X}$ [95]. Note that in (3.8) the weight is exponential in the accumulated absolute error instead of the accumulated quadratic error as in [95]; this responds

to a slight improvement in the prediction performance observed empirically in [15].

In summary, the work in [15], presents the design of an encoder that combines sequential predictive coding, and expert advice, to achieve an efficient compression algorithm. The experts used are adaptive linear predictors, which use a fixed number of past samples obtained from channels physically close to each other, and which take special advantage of using samples from the present time. To define the encoding order of the samples, a coding tree is built (offline) that allows to satisfy the causal encoding restrictions of the channels, and also minimizes the physical distance between them when taken in pairs. In order to sequentially adapt the coefficients of the predictors, a lattice algorithm is used that allows to calculate efficiently, and simultaneously, for all the orders d, $0 \leq d \leq P$, the coefficients at time n from the coefficients at time $n-1$. Finally, the final prediction function of each sample is defined as the weighted mixture of all predictions by expert advice, where the weight assigned to each predictor is an exponentially decreasing function of its absolute prediction errors.

3.4 General Speck Compressor

The encoder presented in Section 3.3 strategically combines a set of techniques that allow to achieve excellent compression results while being of low algorithmic complexity. In this section we present GSC, a low complexity compressor whose general architecture is based on the algorithm RLS proposed in [15]. GSC considerably improves the computational efficiency of RLS without significantly hindering the compression performance.

To achieve this, we tackle a key point of the proposed coding architecture, which is the calculation of predictions. To start, in the Section 3.4.1, we present a low complexity adaptive linear prediction algorithm for multi-channel signals, which is an extension of a single-channel prediction algorithm proposed in [97]. This linear predictor is extremely simple and is implemented using exclusively integer arithmetic. Then, in Section 3.4.2, we propose an efficient implementation of the expert advice algorithm, which also operates entirely with integer arithmetic.

3.4.1 A multi-channel extension of the Speck algorithm

In [97], Speck introduces a digital image compression algorithm that uses adaptive linear predictors. The algorithm can easily be adapted to encode single-channel signals; however, its extension to multi-channel signals is not trivial. In this section we explain the algorithm proposed in [97], and then propose an extension for multi-channel signals.

A single-channel linear predictor p, of order d_p, for a sample $x_i(n)$ as a function of past samples of the same channel i, $x_i(1) \dots x_i(n-1)$, has the form

$$\hat{s}_i^p(n) = \sum_{k=1}^{d_p} a_{i,k} x_i(n-k)\,, \tag{3.9}$$

where $a_{i,k}$ are real coefficients, $1 \le k \le d_p$. The Speck algorithm defines $a_{i,k} = \hat{a}_{i,k}/K$ as a rational number, where $\hat{a}_{i,k}$ are integer coefficients, $1 \le k \le d_p$, and K is an integer normalization constant (usually a power of two, so that division by K can be carried out by a bit-shift operation). This definition is very convenient from a practical implementation perspective, since the coefficients admit a simple fixed point representation, where the precision is given by the value of K.

The coefficients $\hat{a}_{i,k}$ are sequentially adapted by comparing the prediction $\hat{s}_i^p(n)$ with the actual value of the sample, $x_i(n)$. The initialization and adaptation methods of the coefficients for single-channels are the following:

- *Initialization:* Coefficients $\hat{a}_{i,k}$ are initialized as

$$\hat{a}_{i,k} = K \,/\, d_p + \begin{cases} 1, & k \le K \mod d_p\,, \\ 0, & \text{otherwise}\,, \end{cases} \tag{3.10}$$

  where K/d_p and $K \mod d_p$ denote the quotient and the remainder of the integer division, respectively.

- *Adaptation:* Let $\epsilon_i^p(n) = x_i(n) - \hat{s}_i^p(n)$ be the *prediction error* at time n, and $\operatorname{sgn}(\epsilon_i^p(n))$ its sign, where $\operatorname{sgn}(\epsilon_i^p(n))$ is equal to 1 when $\epsilon_i^p(n) > 0$, 0 when $\epsilon_i^p(n) = 0$, and -1 otherwise. If $\epsilon_i^p(n) = 0$, no adaptation is performed; otherwise, the coefficients $\hat{a}_{i,k}$ associated with the largest and smallest of the last d_p (signed) samples, $x_i(n-k), 1 \le k \le d_p$, are decremented and incremented respectively by $\operatorname{sgn}(\epsilon_i^p(n))$; ties are decided by some fixed policy, for example, choosing the coefficient with smallest index. Note that when adding or subtracting the sign of the error to a coefficient $\hat{a}_{i,k}$, we are adjusting the real coefficient $a_{i,k}$ implicitly by a value of $\delta = 1/K$.

The initialization and update procedures explained above ensure that the coefficients $a_{i,k}$ add up to one for every channel i; however, it is worth noting that some of them can take negative values and therefore the prediction $\hat{s}_i^p(n)$ is not necessarily an average of the past samples.

Recall from Section 3.3 that, in the scheme proposed in [15], the prediction $\dot{x}_i^p(n)$ for channel i depends on the d_p most recent samples of channel i, the most recent d_p samples of the helper channel ℓ, and the current sample of channel ℓ. Therefore,

we have that

$$\dot{x}_i^p(n) = \sum_{k=1}^{d_p} a_{i,k} x_i(n-k) + \sum_{k=0}^{d_p} b_{i,k} x_\ell(n-k), \tag{3.11}$$

where $a_{i,k}$ y $b_{i,k}$ are real (adaptive) coefficients.

To adapt Speck's algorithm to multi-channel signals, the initialization and adaptation procedures could be directly extended to the concatenation of the samples from the two channels, $(x_i(n-k) : k = 1, \ldots, d_p)$ and $(x_\ell(n-k) : k = 0, \ldots, d_p)$. However, we observe that this direct adaptation results in poor performance when the mean values of channels i and ℓ differ significantly. On the other hand, when applying the procedures on zero-centered versions of the channels, the results obtained are remarkably good.

To obtain zero-centered versions of the channels, we subtract from each channel an estimation of its moving average, $\bar{x}_i(n)$, given by

$$\bar{x}_i(n) = (1-\beta)\bar{x}_i(n-1) + \beta x_i(n), \tag{3.12}$$

where β is a parameter in range $0 < \beta < 1$. The calculation in (3.12) corresponds to an *exponentially weighted moving average*, where the influence of each sample on the mean decreases exponentially with time. The value of β determines the magnitude of the exponential decay, so that the greater the value of β, the faster the decay over time of the influence that each sample has on the moving average.

The computation of $\bar{x}_i(n)$ can be efficiently implemented using only integer arithmetic. For this, we define an auxiliary variable $\alpha_i(n) \triangleq \beta^{-1}\bar{x}_i(n)$ and rewrite (3.12) as

$$\begin{aligned} \alpha_i(n) &= \beta^{-1}\bar{x}_i(n-1) - \bar{x}_i(n-1) + x_i(n) \\ &= \alpha_i(n-1) - \bar{x}_i(n-1) + x_i(n). \end{aligned} \tag{3.13}$$

The recursion 3.13 allows to sequentially calculate $\alpha_i(n)$ by simply performing an addition and a subtraction operation from $\alpha_i(n-1)$. Also, by choosing β as a negative power of two, $\beta = 2^{-b}$, we get $\bar{x}_i(n) = \alpha_i(n) \gg b$, where $\gg$ denotes the bit-shift operation to the right.

Defining $z_i(n) = x_i(n) - \bar{x}_i(n-1)$, we rewrite (3.11) as

$$\dot{x}_i^p(n) = \bar{x}_i(n-1) + \frac{\sum_{k=1}^{d_p} \hat{a}_{i,k} z_i(n-k) + \sum_{k=0}^{d_p} \hat{b}_{i,k} z_\ell(n-k)}{K},$$

where $\hat{a}_{i,k}$ and $\hat{b}_{i,k}$ are coefficients that are initialized and adapted following the Speck procedure applied to the concatenation of the zero-centered samples of the channels i, ℓ. This completes our definition of the prediction computation for the multi-channel extension of the Speck algorithm. In the sequel, we refer to a predictor

that uses the Speck efficient adaptation algorithm, single-channel or multi-channel, as *single-channel Speck predictor* or *multi-channel Speck predictor*, respectively.

3.4.2 An efficient implementation of the expert advice scheme

Expert advice is a well studied method with a strong theoretical justification [95]. Recalling equation (3.7), the final prediction of a sample is calculated as a weighted average of the outputs of a set of predictors $\mathcal{P}$ working in parallel,

$$\hat{x}_i(n) = \left\lfloor \frac{1}{W_i(n)} \sum_{p \in \mathcal{P}} \mu_i^p(n) \dot{x}_i^p(n) \right\rfloor ,$$

where $\mu_i^p(n)$ is a positive weight that decays exponentially with the average of the absolute values of the prediction errors of a predictor p at time n, and $W_i(n) = \sum_{p \in \mathcal{P}} \mu_i^p(n)$ is a normalization factor. Next we propose a variant inspired by these same operating principles, focused on computational efficiency.

We define $\bar{e}_i^p(n)$ as the average of the absolute values of the prediction errors of predictor p of channel i at time n, estimated using the same method and parameters as in (3.12), on the sequence of the absolute values of the past errors of predictor p, i.e, the sequence of $|\epsilon_i^p(j)|$ with $j < n$. For the sake of computational efficiency, we calculate the weight corresponding to a predictor p using a base-2 exponential function with a positive exponent, which can be implemented with a low computational cost operation, such as the left bit-shift. Specifically, we define

$$\mu_i^p(n) = \mathbb{1}_{\{s_{\max} \geq c_i(n) \bar{e}_i^p(n)\}} \times 2^{s_{\max} - c_i(n) \bar{e}_i^p(n)} , \tag{3.14}$$

where $\mathbb{1}_{\{s_{\max} \geq c_i(n) \bar{e}_i^p(n)\}}$ is the indicator function that is 1 when $s_{\max} \geq c_i(n) \bar{e}_i^p(n)$ and 0 otherwise, $s_{\max}$ is a positive integer constant, and $c_i(n)$ a scaling factor that is adjusted in each step according to the performance of the predictors, as explained later. The factor $c_i(n)$ is defined as a negative power of 2, $c_i(n) = 2^{-b_i(n)}$, $b_i(n) \in \mathbb{N}$, such that we are able to perform the operation through a right bit-shift.

Equation (3.14) uses the average of the absolute errors, $\bar{e}_i^p(n)$, as a way of assessing the performance of each predictor. As a predictor p makes less accurate predictions, $\bar{e}_i^p(n)$ grows, the value $s_{\max} - c_i(n) \bar{e}_i^p(n)$ decreases, and the same happens with the weight attributed to predictor p. In particular, when $c_i(n) \bar{e}_i^p(n) > s_{\max}$ the weight assigned to the predictor is 0. We use the variable $c_i(n)$ to adjust the total sum of the weights, $W_i(n)$, within a preestablished range $[W_{\min}, W_{\max}]$. Keeping $W_i(n)$ in a narrow range prevents performance differences between predictors from fading out due to lack of numerical precision. In practice, we observe that this forces the algorithm to weight predictors whose average errors are similar, but not the same, with different weights, which results in better compression levels.

If $W_i(n)$ falls below the lower limit, W_{min}, we increment the value of $b_i(n)$, which translates into a decrease of $c_i(n)$ and thus an increment of the weights (see (3.14)). Similarly, if $W_i(n)$ is larger than W_{max}, the value of $b_i(n)$ is decreased. In summary, the update procedure of $b_i(n)$ with respect to $W_i(n)$ is defined as

$$b_i(n) = \begin{cases} b_i(n) + 1, & \text{if } W_i(n) < W_{min}, \\ b_i(n) - 1, & \text{if } W_i(n) > W_{max} \text{ and } b_i(n) > 0. \end{cases}$$

This procedure is repeated until $W_i(n)$ is within the desired range or $b_i(n) = 0$. This definition ensures that there is always at least one predictor with a positive weight, and that the total sum of the weights is in the desired range, except when the value $b_i(n) = 0$ is reached . This condition only occurs when the average of absolute error values for all predictors is so small that $W_i(n)$ exceeds W_{max}. Notice that this implies that even for $b_i(n) = 0$, i.e, $c_i(n) = 1$, the performance of all the predictors is exceptionally good and, thus, the weighting of the predictors is of little importance for the final prediction.

Notice that the proposed version of the expert advice presented only uses integer-based arithmetic and bit-shift operations, which results in a lower computational complexity compared to the algorithm used in [15].

After defining the Speck predictors, and the new integer-based implementation of the expert advice algorithm, we conclude the definition of the prediction module of GSC, by defining what predictors make up the set $\mathcal{P}$. We start by defining a parameter P, which establishes the maximum order of all the predictors used in $\mathcal{P}$. For the root channel r, $\mathcal{P}$ is composed of P single-channel Speck predictors of order d, $1 \leq d \leq P$. For each channel i other than r, whose helper channel is ℓ, $\mathcal{P}$ is composed of P single-channel Speck predictors, one for each order d, $1 \leq d \leq P$, and P multi-channel Speck predictors, one for each order d, $1 \leq d \leq P$, which predict samples of channel i using channel ℓ as a helper. We use the notation GSC-x to refer to a GSC with maximum order $P = x$.

3.5 Experimental evaluation of GSC

To evaluate the performance of GSC, described in Section 3.4, we run the compressor on multiple publicly available datasets of different types of multi-channel biomedical signals, and compare it to other compressors that achieve state of the art results. In addition, we evaluate the compressors on a seismic dataset, which, although not biomedical, is usually modeled with similar statistical tools. In Section 3.5.1, we describe the specific metrics used to evaluate compressor performance. In Section 3.5.2 we describe the datasets used and the experimentation environment. In Section 3.5.3 we specify the compressors used and their settings. Finally, in

Section 3.5.4, we present the obtained results.

3.5.1 Compression performance metrics used for evaluation

For each dataset, we encode and decode each file individually, and calculate the *compression ratio*, measured in *bits per sample* (*bps*), defined as $CR = L/N$, where N is the total sum of the number of scalar samples present in all the files in the dataset, and L is the sum of the sizes in bits of all compressed files in the dataset. Note that lower values of CR indicate better compression performance. To facilitate the comparison between compression ratios, we define the *percentage relative difference* of CR_2 with respect to CR_1, as $\frac{CR_2 - CR_1}{CR_1} \times 100$. Negative values of this measure indicate that CR_2 is better than CR_1, while positive values indicate that CR_2 is worse.

As a measure of encoding and decoding speed we take the average time, in microseconds (μs), that the compressor takes to process a scalar sample, using the same computer system for all experiments. We call this measure *encoding time per scalar sample* (ETPS) and we calculate it as ETPS = T_c/N, where T_c is the sum of the compression times of all the files of the dataset. Similarly, we calculate the *decoding time per scalar sample* (DTPS) by dividing the sum of the decoding times for all files, T_d, by the number of scalar samples decoded from the dataset, N. All time measurements include those for reading and writing files.

3.5.2 Datasets and experimentation environment

Here we present the datasets used in our experiments, specifying characteristics of the equipment, and procedure, used for acquisition.

- Phys-a and Phys-b [38, 92] (BCI2000 instrumentation system): 64-channel EEG, at 160Hz, and 12 bps, of 109 subjects using the BCI2000 system. The dataset consists of 1308 2-minute recordings of subjects imagining the performance of motor tasks (Phys-a), and 218 1-minute calibration recordings (Phys-b).

- BCI-a and BCI-b [30] (BCI Competition III, [1] dataset IV): 118-channel EEG, at 1000Hz, and 16 bps, of 5 subjects imagining the performance of motor tasks (BCI-a). The average duration of the 8 recordings in the dataset is 39 minutes, with a minimum of at least 13 minutes and a maximum of at least 50 minutes. BCI-b is a 100Hz *subsampled* version of BCI-a.

[1] http://bbci.de/competition/iii/

- Comp [8] (BCI Competition IV [1]): 59-channel EEG, at 1000Hz, and 16 bps, of 7 subjects imagining the performance of motor tasks. The dataset consists of 14 recordings of different duration, ranging from 29 to 41 minutes, with an average duration of 35 minutes.

- Neur [26]: 31-channel EEG, at 1000Hz, and 16 bps, of 15 subjects performing classification and image recognition tasks. The dataset consists of 373 recordings with an average duration of 3.5 minutes, a minimum of 3.3 minutes, and a maximum of 5.5 minutes.

- ECG [38, 11] (Physikalisch-Technische Bundesanstalt (PTB) Diagnostic ECG dataset): Standard 12-lead ECG[2], at 1000Hz, and 16 bps. This dataset consists of 549 recordings taken from 290 patients, with an average duration of 1.8 minutes, a minimum of 0.5 minutes, and a maximum of 2 minutes. For each of the files in the dataset, the derivations i, ii, $v_1 ... v_6$ were extracted to form 8-channel signals, since the remaining 4 are linear combinations of the other ones.

- MGH [38, 103] (The Massachusetts General Hospital / Marquette Foundation (MGH / MF) Waveform dataset): 8-channel recordings, at 360 Hz, and 16 bps, recording different biomedical critical care signals. Specifically, 3 channels are used for ECG, one for blood pressure, one for pulmonary arterial pressure, one for central venous pressure, one for respiratory impedance, and one for CO_2 airway waveforms. Recordings vary between 12 and 86 minutes and are on average one hour long.

- Sism [16] (Southern California Earthquake Center, Caltech.dataset): Data from the Arizona sensor network (181 channels, 1Hz sample rate, at 32 bps) and the Baja California sensor network (64 channels, sampling rate 100Hz). The test dataset consists of 12 files from each network, each consisting of an hour long logs taken at time 0:00 each month between September 2013 and August 2014.

In summary, we have seven EEG signal datasets, one ECG signal dataset, one dataset that is a combination of different biomedical critical care signals, and one dataset of seismological data. To ease the testing compression and execution times, all files were converted to flat binary format, which can be directly input to all the evaluated compressors.

[1] http://bbci.de/competition/iv/

[2] Cardiac leads are the recording of the difference in electrical potentials between two points, either between two electrodes (bipolar lead) or between a virtual point and an electrode (monopolar leads).

All tests were performed on the same desktop computer (Intel i7, single thread, 3.4GHz), with operating system Ubuntu version 14.04.

3.5.3 Evaluated compressors and their configurations

To evaluate the performance of the proposed compressor we compare it with publicly available implementations of other compression algorithms in the state of the art, including RLS [15]. We use the same configuration parameters as in [15]; in particular we take $P = 7$, which is equivalent to 8 multi-channel predictors of order d, $0 \leq d \leq 7$. In general we refer to the RLS compressor configured with the parameter $P = x$ as RLS-x; in this case we use the RLS-7 compressor.

Both RLS and GSC require a previously determined encoding tree to establish the order in which the channels are encoded for each dataset. For the EEG datasets we use the minimal spanning trees described in the Section 3.3. In the ECG dataset, each of the leads (channels) measures the voltage between two electrodes along a certain direction vector. To build the coding tree, following the same criteria as in [15], we calculate the minimum spanning tree by taking the angles between the direction vectors as distances. For the MGH dataset, we manually built a coding tree by observing the shapes of the graphs of the signals produced by the different biomedical sensors, joining channels in the tree if their signals subjectively seemed to be correlated to the naked eye. Lastly, for the seismic dataset we used a minimum spanning tree, taking the physical distance between the sensors as measure. For all trees, the root channel is arbitrarily determined, as experimental tests performed in [15] indicate that the choice of the root channel does not have a significant impact on the performance of the compressor.

Another compressor we use for evaluation is the reference implementation[1] of the multi-channel audio lossless compression algorithm *MPEG-4 audio lossless coding standard [49]* (MP4-ALS), which supports 2^{16} channels, and resolutions up to 32 bps. The algorithm has been applied before on biomedical signals, achieving good results [55]; for EEG MP4-ALS obtains better compression ratios than specialized algorithms [99, 25] evaluated on the same datasets. For our tests, MP4-ALS was configured with the command line parameter -z3, which is the one that experimentally achieved the best results, maintaining a balance between compression ratio and encoding speed.

Lastly, we use the *Free Lossless Audio Codec*[2] (Flac), a compressor which is also built for the lossless compression of audio signals. Flac supports multi-channel encoding of up to 8 channels, and resolutions of up to 24 bps. However, most of the

[1] http://www.nue.tu-berlin.de/menue/forschung/projekte/beendete_projekte/mpeg-4_audio_lossless_coding_als

[2] https://xiph.org/flac/index.html

datasets used in our tests have files with more than 8 channels. In order to run Flac on this datasets, each file is divided into multiple smaller files with 8 channels each, and one with 8 or fewer channels. Regarding the seismic dataset Sism, as it has a 32-bit resolution, it is not possible to get results for Flac. For all tests, we run Flac with the command line parameter *--best*, which optimizes the compression ratio.

For GSC, we set the maximum order $P = 8$ (GSC-8). For the Speck predictors we set the parameter $K = 2^7$, and for the exponential decay of the channel averages we use $\beta = 2^{-7}$. As configuration parameters of the expert advice algorithm we use $s_{max} = 12$, $W_{min} = 1$, $W_{max} = 2^{s_{max}-1}$, and for the calculation of the average absolute errors, $\bar{e}_i^p(n)$, we set $\beta = 2^{-7}$.

3.5.4 Results and analysis of the performance of GSC

In Table 3.1 we present the results of executing the compressors GSC-8, RLS-7, MP4-ALS and Flac, on the datasets presented in Section 3.5.2. For each dataset, and each compressor, the table shows the compression ratio obtained, measured in bits per sample (bps), and the encoding and decoding times per scalar sample, measured in μs. The table also shows, in parenthesis, the percentage relative difference between the *CR* of each compressor and the *CR* of RLS-7.

The first thing we observe is that the RLS-7 compressor consistently achieves the best compression results over all datasets. The next best compressor is GSC-8, which in the BCI-b dataset equals the compression ratio obtained by RLS-7. Specifically, the maximum percentage relative difference in compression ratio of GSC-8 with respect to RLS-7 is 5.0% (for the Neur dataset), while the average percentage relative difference over all datasets is 1.6%.

On the other hand, if we compare the compression ratios of GSC-8 against MP4-ALS, and Flac, we see that GSC-8 consistently achieves better results. In the case of MP4-ALS, the average percentage relative difference with respect to RLS is 11.1%, while for Flac it is 23.5%.

Regarding encoding speed, Flac is considerably faster than the rest of the compressors, with an ETPS of $0.09 \pm 0.03\mu$s. The next fastest compressor is GSC-8, which is around 5x times slower, with an ETPS of $0.45 \pm 0.03\mu$s. However, compared to RLS-7 and MP4-ALS, GSC-8 is considerably faster. Specifically, it is almost twice as fast as RLS-7, which has an average ETPS of $0.87 \pm 0.06\mu$s, and more than twice as fast as MP4-ALS which has an average ETPS of $1.05 \pm 0.05\mu$s.

For the RLS-7 and GSC-8, the encoding and decoding speeds are practically the same; this is because their compression and decompression architectures are symmetric (the encoding and decoding processes carry a similar computational load). However, both MP4-ALS and Flac have an asymmetric architecture, resulting in the decoding speed faster than the encoding speed (especially in the case of Flac).

Table 3.1: Compression ratios and encoding and decoding time per scalar sample for GSC-8, RLS-7, MP4-ALS, and Flac compressors, on the datasets presented in Section 3.5.2. The percentage relative difference of the *CR*s of GSC-8, MP4-ALS and Flac with respect to that of RLS-7 is shown in parentheses.

Dataset	Compressor	*CR* (bps)	ETPS (μs)	DTPS (μs)
Phys-a	GSC-8	4.79 (1.9)	0.47	0.47
	RLS-7	**4.70**	0.93	0.92
	MP4-ALS	5.57 (18.5)	1.06	1.06
	Flac	6.31 (34.3)	**0.11**	**0.02**
Phys-b	GSC-8	4.93 (2.9)	0.47	0.48
	RLS-7	**4.79**	0.93	0.93
	MP4-ALS	5.72 (19.4)	1.07	1.09
	Flac	6.85 (43.0)	**0.12**	**0.02**
Comp	GSC-8	5.45 (0.6)	0.46	0.47
	RLS-7	**5.42**	0.92	0.91
	MP4-ALS	5.90 (8.9)	1.06	1.00
	Flac	6.40 (18.1)	**0.08**	**0.01**
Neur	GSC-8	3.76 (5.0)	0.45	0.45
	RLS-7	**3.58**	0.89	0.89
	MP4-ALS	3.86 (7.8)	1.07	1.07
	Flac	4.45 (24.3)	**0.07**	**0.01**
BCI-a	GSC-8	5.29 (1.5)	0.47	0.47
	RLS-7	**5.21**	0.92	0.92
	MP4-ALS	5.82 (11.7)	1.06	1.07
	Flac	6.37 (22.3)	**0.08**	**0.01**
BCI-b	GSC-8	**6.93** (0.0)	0.47	0.48
	RLS-7	**6.93**	0.93	0.92
	MP4-ALS	7.99 (15.3)	1.07	0.88
	Flac	8.71 (25.7)	**0.08**	**0.02**
ECG	GSC-8	4.80 (0.4)	0.43	0.44
	RLS-7	**4.78**	0.81	0.81
	MP4-ALS	5.07 (6.1)	1.09	1.09
	Flac	5.45 (14.0)	**0.09**	**0.02**
MGH	GSC-8	2.65 (1.5)	0.43	0.43
	RLS-7	**2.61**	0.96	0.97
	MP4-ALS	2.82 (8.0)	1.00	0.98
	Flac	2.77 (6.1)	**0.07**	**0.01**
Sism	GSC-8	7.75 (0.8)	**0.48**	**0.48**
	RLS-7	**7.69**	0.95	0.94
	MP4-ALS	8.03 (4.4)	1.04	1.05
	Flac	-	-	-

Specifically, for these datasets Flac the average DTPS of Flac is $0.01 \pm 0.01 \mu$s, which means its decompression process is more than 4x times faster than its compression process.

From these results, we conclude that the proposed compressor, GSC-8, manages to significantly improve the encoding and decoding speeds compared to RLS-7. Specifically, the ETPS and DTPS values show that GSC-8 is almost twice as fast as RLS-7. Furthermore, as expected, the decrease in computational cost achieved by simplifying GSC-8 prediction module results in a degradation in compression ratios

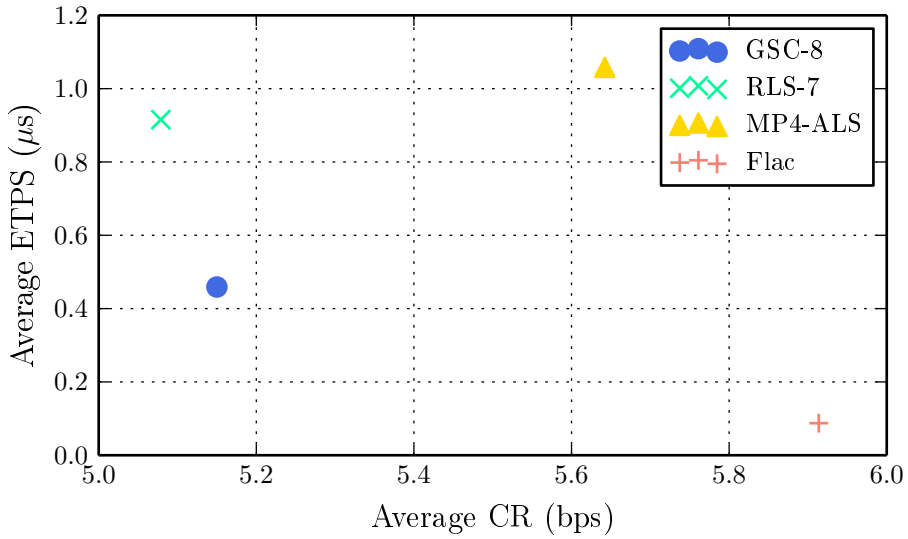


Figure 3.1: Average ETPS vs. average compression ratio, for the different compression algorithms.

compared to RLS. However, the results show that the compression ratios obtained by GSC-8 are still highly competitive.

Figure 3.1 illustrates the advantages and disadvantages of each of the evaluated compression algorithms. The figure shows for each compressor a point in the plane that represents the average compression ratio on the X axis, and the ETPS on the Y axis. Both measurements are averaged across all datasets. In this graph, the optimal value is at the point (0,0), and the values of both metrics deteriorate as we move away from the axes. We can easily see that the MP4-ALS compressor is significantly outperformed in terms of compression ratio and speed by GSC-8. Flac is the compressor that achieves the best encoding speed by a wide margin, but its compression ratio is considerably worse than all other compressors. In terms of the balance between compression speed and compression ratio, we can say that GSC-8 offers an excellent trade-off.

Chapter 4

Analysis of the performance of the predictors of GSC

As we explained in Section 3.3, in general, we do not know, a priori, what predictors from $\mathcal{P}$ will perform better than others. Expert advice is responsible for differentiating which predictors are performing well from those that are not, and weights the prediction of each predictor based on its previous performance. Therefore, the set $\mathcal{P}$ should include various predictors of different orders, so that, ideally, there is at all times a predictor that accurately predicts the samples of the signal.

However, increasing the number of predictors of different orders can lead to high execution times, and does not always improve compression levels. For example, for $d_1 < d_2$, it is easy to see that any linear predictor of order d_1 is a special case of one of order d_2 (simply set to zero the coefficients of order larger than d_1). Therefore, a predictor of order d_1 would be, in principle, redundant with respect to one of order d_2, and not having it would make the compression process more efficient.

On the other hand, taking into account that the predictors are adaptive, and that the statistical characteristics of a signal can vary over time, not always the same predictors have the best performance. For example, low-order predictors adapt faster to the characteristics of the signal than high-order ones, since less coefficients need to be adjusted. Therefore, low-order predictors usually perform better in moments where the statistical characteristics of the signal are changing. At the same time, higher-order predictors are able to capture statistical regularities of more complex signals compared to low-order predictors, and therefore achieve better results when they have the time to adapt to the signal.

In order to optimize the execution time of GSC, and, at the same time, achieve good compression results, it is key to determine how many predictors to use in the expert advice algorithm, and the orders of these predictors. A large number of predictors covers many possible different types of signals. Fewer predictors, on the

other hand, require less execution time in predicting and adjusting coefficients.

For the construction of a more efficient compressor, we intend to find a balance between the two conditions, strategically choosing a reduced subset of predictors to improve execution times, keeping the predictors that show the best performance for the type of signal we want to compress. In addition, as mentioned, the larger the number of predictors the slower the convergence rate of the performance of the weighted predictor to that of the best predictor in $\mathcal{P}$, which implies that an excessively large number of predictors could lead to poor compression performance.

Next, in Section 4.1, we present a brief analysis of the impact of the number of predictors on the execution time and compression performance of GSC. Then, in Section 4.2, we define a set of metrics for the purpose of evaluating the performance of each predictor within the expert advice algorithm. Finally, in Section 4.3, we present the results of evaluating these metrics on each type of signal from each dataset, and determine a way forward for the construction of a more efficient compressor.

4.1 Analysis of the impact of the number of predictors on execution time and compression performance

As a first step, we analyze the impact of reducing the number of predictors on execution time and compression ratios. Specifically, we compare the performance between the compressor GSC-8 (16 predictors) and GSC-2 (4 predictors).

Table 4.1: Compression ratios, and encoding and decoding time per scalar sample, for GSC-8 and GSC-2, on the datasets presented in Section 3.5.2. The percentage relative difference of the compression ratio of GSC-2 with respect to GSC-8 is shown in parentheses.

Dataset	Compressor	CR (bps)	ETPS (μs)	DTPS (μs)
Phys-a	GSC-8	**4.79**	0.47	0.47
	GSC-2	4.87 (1.7)	**0.12**	**0.12**
Phys-b	GSC-8	**4.93**	0.47	0.48
	GSC-2	4.99 (1.2)	**0.13**	**0.13**
Comp	GSC-8	**5.45**	0.46	0.47
	GSC-2	5.70 (4.6)	**0.12**	**0.12**
Neur	GSC-8	**3.76**	0.45	0.45
	GSC-2	4.33 (15.2)	**0.12**	**0.12**
BCI-a	GSC-8	**5.29**	0.47	0.47
	GSC-2	5.70 (7.8)	**0.12**	**0.12**
BCI-b	GSC-8	**6.93**	0.47	0.48
	GSC-2	7.06 (1.9)	**0.12**	**0.13**
ECG	GSC-8	**4.80**	0.43	0.44
	GSC-2	4.92 (2.5)	**0.12**	**0.13**
MGH	GSC-8	**2.65**	0.43	0.43
	GSC-2	2.91 (9.8)	**0.11**	**0.12**
Sism	GSC-8	**7.75**	0.48	0.48
	GSC-2	8.43 (8.8)	**0.13**	**0.13**

Table 4.1 shows the compression ratios, and the average encoding and decoding times, for GSC-8 and GSC-2. We observe that the compression ratios of GSC-2 are worse than those of GSC-8 in each of the datasets, with the largest relative difference being 15.2%, in the Neur dataset, while the smallest difference is of 1.2% on the Phys-b dataset. These results show that compression ratios can vary considerably depending on the set of predictors available to GSC. On the other hand, with only 4 predictors, similar results are achieved on the Phys-b dataset, which indicates that we can achieve good compression performance without the 12 additional predictors of GSC-8. If we look at the encoding and decoding times, GSC-2 is between 3x and 4x times faster than GSC-8 on each of the datasets, which represents a very significant improvement.

Indeed, using the profiling tool *gprof*[1] we determined that GSC-8 spends approximately 91% of the execution time making predictions and updating the predictors, while for GSC-2, the prediction and update functions spend approximately 73% of the execution time.

The comparison of execution times on the datasets, and the different profiling results between compressors GSC-8 and GSC-2, indicate that reducing the number of predictors leads to a significant improvement in execution time. On the other hand, the compression results suggest that if the predictors are chosen strategically, the compression results may not deteriorate considerably, as in the case of dataset *Phys-b*.

4.2 Definition of metrics for evaluating the performance of the predictors in expert advice

To evaluate the performances of the predictors in the expert advice algorithm we define a series of metrics that allow to measure the performance of a specific predictor in the algorithm. Expert advice assigns to each predictor p, of each channel i, in time n, a weight $\mu_i^p(n)$ that is proportional to the average of absolute prediction errors of the predictor obtained in the recent past samples, as explained in Section 3.4.2.

If the predictor p of the channel i has weight $\mu_i^p(n)$, in time n, then we say that its *percentage influence* on the prediction $\dot{x}_i^p(n)$ is

$$u_i^p(n) = \frac{\mu_i^p(n) * 100}{\sum_{p \in \mathcal{P}} \mu_i^p(n)}. \tag{4.1}$$

Using the percentage influence $u_i^p(n)$ as an indicator of the performance of the predictor, instead of directly using the absolute prediction error average, has the

[1] https://sourceware.org/binutils/docs/gprof/

advantage of being independent of the scale of error values, which may be different for each test file.

To evaluate the performance of a predictor p over time on a channel, we use the *average percentage influence* defined as

$$\bar{u}_i^p(n) = \frac{1}{n}\sum_{j=1}^{n} u_i^p(j).$$

For a file a with n_a vector samples taken from a set $C(a)$ of channels we define the *average percentage influence* of a predictor p on file a as

$$\bar{u}^p(a) = \frac{1}{|C(a)|}\sum_{i\in C(a)} \bar{u}_i^p(n_a).$$

Similarly, for a dataset b that has a set of files $A(b)$, we define the *average percentage influence* of a predictor p on dataset b as

$$\bar{u}^p(b) = \frac{1}{|A(b)|}\sum_{a\in A(b)} \bar{u}^p(a).$$

We can use the average percentage influence on a dataset as a guide to decide which predictors perform better than others for a certain type of signal. In this sense, to define a reduced set of predictors we want to choose the predictors with the highest average percentage influence. Moreover, we could speculate that if two predictors have similar average percentage influences, they could be performing similar predictions, and, therefore, we could discard one of them. However, the direct comparison between average percentage influences can be misleading, since two predictors can have the same average percentage influence by achieving good performances at different times. In that case, it would be convenient to use the two predictors, since they complement each other.

To have a better understanding of the relation between the average percentage influences of the predictors, we define the *influence distance* between predictors, which is calculated from the absolute difference between the percentage influences of two predictors. Specifically, given a pair of predictors (p, q), we define the distance between them for a channel i in time n as

$$d_i^{p,q}(n) = |u_i^p(n) - u_i^q(n)|,$$

where $u_i^p(n)$ and $u_i^q(n)$ are the average percentage influences of the predictors in time n, as defined in (4.1). The *average influence distance* over time between predictors

p, q on channel i, is defined as

$$\bar{d}_i^{p,q}(n) = \frac{1}{n} \sum_{j=1}^{n} d_i^{p,q}(j).$$

The average influence distance between predictors p, q on a file a, is defined as

$$\bar{d}^{p,q}(a) = \frac{1}{|C(a)|} \sum_{i \epsilon C(a)} \bar{d}_i^{p,q}(n_a),$$

and finally, the average influence distance between predictors p, q on a dataset b is defined as

$$\bar{d}^{p,q}(b) = \frac{1}{|A(b)|} \sum_{a \epsilon A(b)} \bar{d}^{p,q}(a).$$

Combining the information provided by the average percentage influence of each predictor with the average influence distance between the predictors allows us to build a better picture of how each predictor performs on each dataset.

4.3 Analysis of the percentage influences and distances

For each dataset detailed in Section 3.5.2, we carry out a series of measurements to evaluate the performance of each predictor of GSC-8 compressor on each dataset. Specifically, for each of the 8 single-channel predictors $s(i)$, $1 \leq i \leq 8$, and for each of the 8 multi-channel predictors $m(j)$, $1 \leq j \leq 8$, we calculate the average percentage influence on each dataset, and the average influence distance between each predictor and the rest. To make the results easier to read, in each graph we assign a color to each predictor of the compressor. To the single-channel predictors we assign a color gradient that goes from yellow to orange, with yellow being the color of s(1) and orange being the color of s(8). To the multi-channel predictors we assign a color gradient that goes from fuchsia to light blue, with fuchsia being the color of m(1) and light blue being the color of m(8). In addition, for each predictor, the graphs represent the standard deviation between files of a dataset as the distance that exists between the average value (the top of the bar) and any of the two horizontal lines that are above or below it.

4.3.1 Results for EEG signals

In Figure 4.1 we present a bar graph that shows, in order from highest to lowest, the average percentage influence of each predictor of the compressor GSC-8 on the Comp EEG signal dataset. To start, notice that the predictors that have a higher average percentage influence are the multi-channel predictors, to the extent that the worst multi-channel predictor, m(1), has an average percentage influence approximately 4

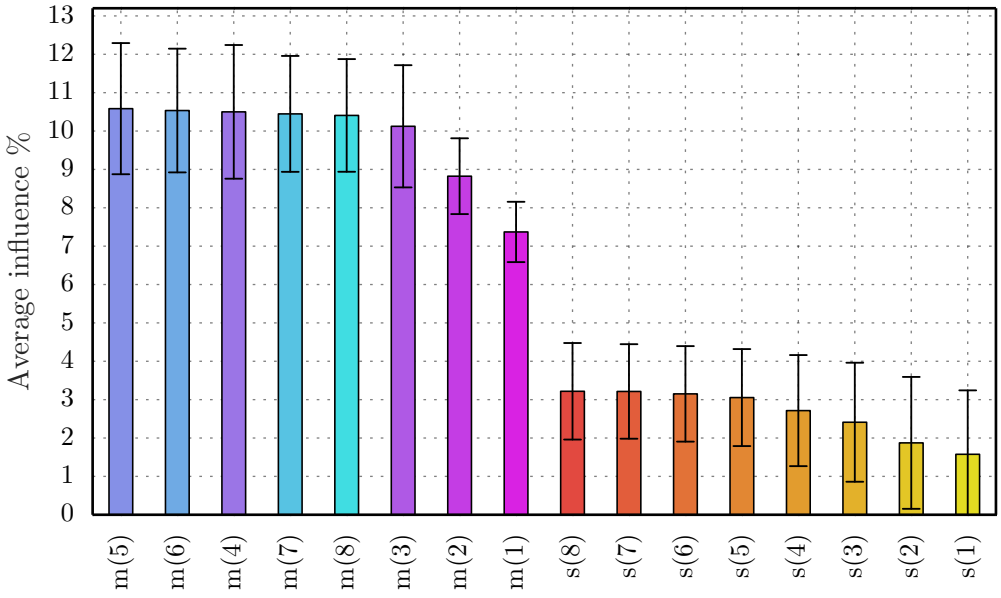


Figure 4.1: Average percentage influence of each predictor of compressor GSC-8 on the Comp EEG dataset, ordered from highest to lowest.

units above the best single-channel predictor, s(8). We could anticipate this, since the good compression results achieved by GSC-8 are partly due to the fact that multi-channel predictors exploit the high correlation between channels in order to make better predictions.

However, this does not imply that single-channel predictors are not important. For example, it could happen that in some pairs of channels the correlation is very high, while in others it is very low, and, in the latter, the single-channel predictors are indispensable.

Continuing with the analysis of Figure 4.1, both in single-channel and multi-channel predictors, the average percentage influence increases with prediction order until a certain order in which the growth slows down. In the case of single-channel predictors, the performance does not seem to improve significantly after order 5, while for the multi-channel predictors, there does not seem to be significant improvements beyond order 4 (in both cases the improvements are considerably less than 1 percentage point, and, in the multi-channel case, the performance deteriorates as the order grows). As we see later, this behavior also occurs on the other datasets. When the average percentage influence of the predictors of a specific type (single or multi-channel), stops growing considerably from certain order d on, we say that d is a *threshold order*.

On the other hand, we observe that the standard deviations of the average percentage influence follow a relatively regular pattern for all the predictors, from which

we conclude that, although we are observing an average of the results on different files, in general the performance of the predictors is similar on files of the same dataset.

The threshold order predictor appears to be a predictor that adequately captures the statistical characteristics of the signal, and such that all the higher order predictors show a similar performance. To reinforce this hypothesis, we analyze the average influence distance between the threshold order predictor, both for single and multi-channel, and the others predictors. In the case of the Comp dataset, we select the threshold predictors m(4) for multi-channel, and s(5) for single-channel.

Figure 4.2 shows the average influence distances of the threshold order multi-channel predictor m(4) with respect to the other predictors, where, by definition, the average influence distance with itself is 0. The average influence distance of the multi-channel predictors ranging from order 3 to 8, differ from the multi-channel predictor of order 4 by less than 1 in all cases.

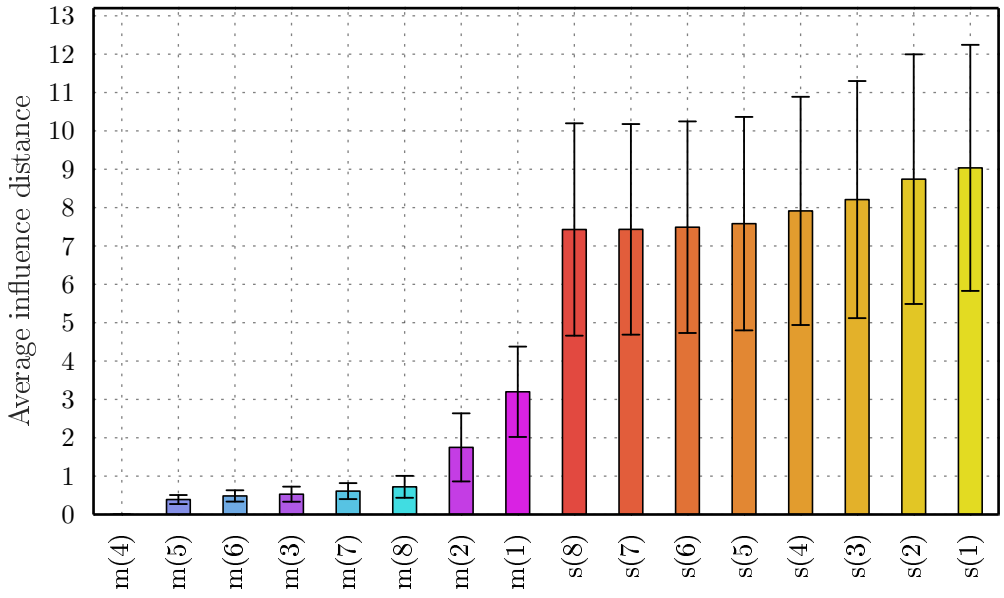


Figure 4.2: Average influence distance between predictor m(4) and the rest of the predictors of compressor GSC-8 on the Comp EEG dataset, ordered from lowest to highest.

In Figure 4.3 we show the average influence distances of the single-channel predictor of threshold order, s(5), with respect to the other predictors, and we observe a very similar behavior to that of the multi-channel case. The average influence distances of single-channel predictors of order greater than 5 are always less than 1 with respect to s(5).

We also observe that, for both single and multi-channel, the standard deviations of the influence distances for the predictors of order greater than the threshold

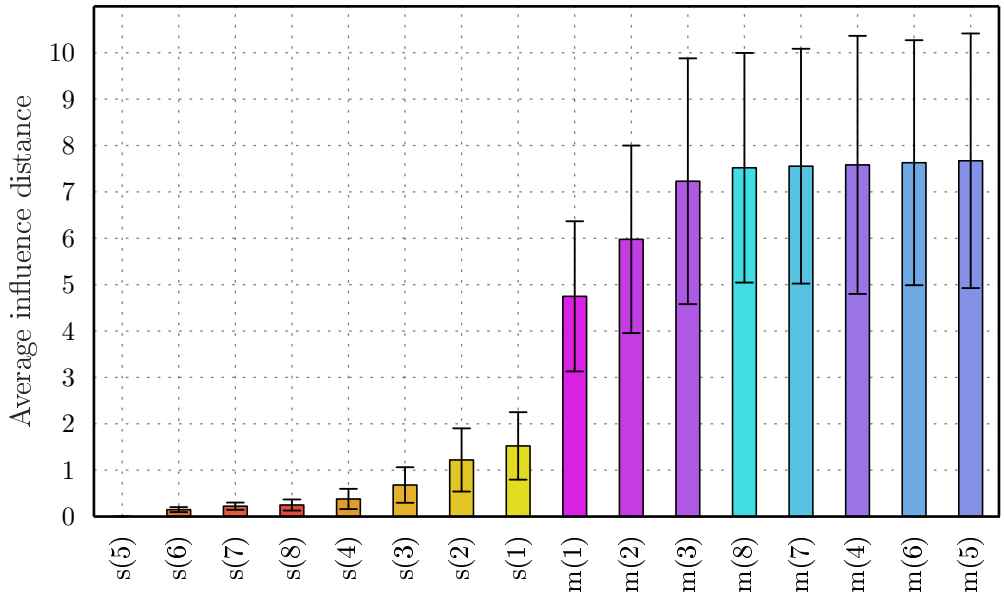


Figure 4.3: Average influence distance between predictor s(5) and the rest of the predictors of compressor GSC-8 on the Comp EEG dataset, ordered from lowest to highest.

are relatively small. This seems to indicate that predictors with similar average percentage influence have similar performances, which means that with fewer predictors (discarding predictors of order higher than the threshold) we could achieve similar results in terms of compression.

The results obtained for the rest of the EEG signal datasets are similar to those obtained for Comp, in the sense that multi-channel predictors show better performance than single-channel predictors and, furthermore, in all the datasets we observe that from a certain threshold order onwards, both in single-channel and multi-channel, the average percentage influence of the predictors does not vary considerably. However, in the Neur dataset the results show some different behaviors.

If we look at Figure 4.4, we observe that not all multi-channel predictors surpass all single-channel predictors in average percentage influence. Specifically, all single-channel predictors of order 3 onwards outperform multi-channel predictors of order 1 and 2.

This observation becomes more interesting in light of the results presented in the Table 4.1 for the compressors GSC-8 and GSC-2. In the Neur dataset, the compression ratio obtained by GSC-2 (of predictors s(1), s(2), m(1), and m(2)) is 0.57 bps above that of GSC-8, which represents a percentage relative difference of 15.2% in compression ratio. This significant deterioration may be due to the fact that GSC-2 does not have a predictor with a performance close to the threshold order predictor, which in this case is m(3) for multi-channel and s(4) for single-channel.

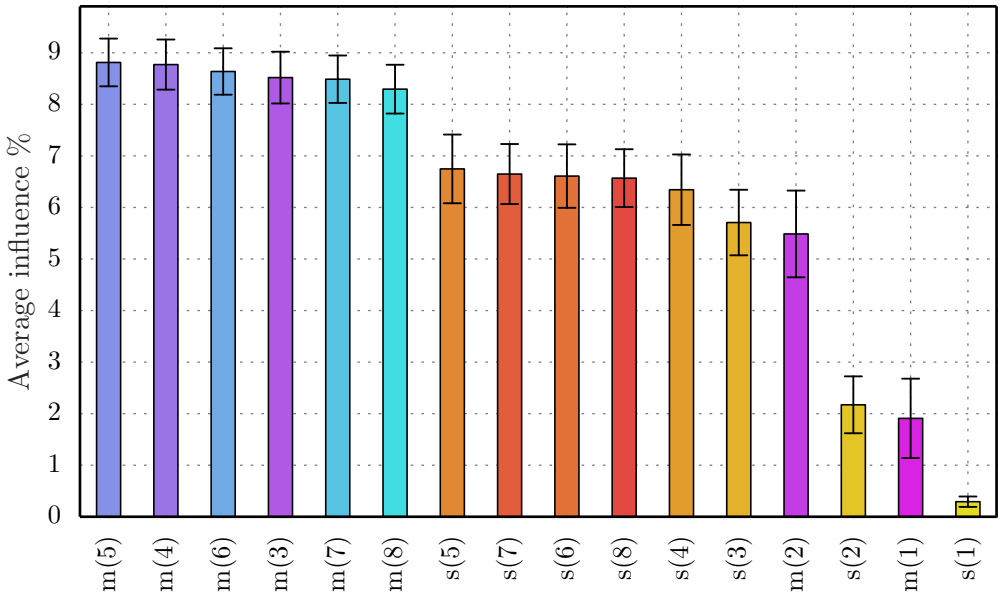


Figure 4.4: Average percentage influence of each predictor of GSC-8 on the Neur EEG dataset, ordered from highest to lowest.

However, if we look at the results reported in Table 4.1 for the Comp dataset, the compression ratio of GSC-2 differs from that of GSC-8 by 0.25 bps, which represents a percentage relative difference of 4.6%, which is significantly less in relative terms than the decline seen on Neur. This makes sense, since predictors m(2) and s(2), which are present in GSC-2, have average percentage influences closer to those of the threshold order predictors (see Figure 4.1).

The rest of the EEG signal datasets present similar results to those of the Comp dataset. In the case of Phys-a, Phys-b, and BCI-a, the multi-channel predictors outperform the single-channel predictors widely, and the average percentage influence stabilizes at threshold order 4 for multi-channel, and threshold order 4 for single-channel.

The only EEG dataset where the average percentage influence does not stabilize from a certain order onwards is the BCI-b dataset. What we observe is that the average percentage influence, both for single-channels and multi-channels, increases considerably until order 3, and then decreases. From a brief analysis of the data, we can speculate that this phenomenon may be due to the fact that there are many channels that frequently present measurement saturation (sudden jumps in the value of the samples to the top of the measurement scale), which may significantly affect predictors of order greater than 3. To better understand the reason behind this phenomenon, a more in-depth study should be performed in the future.

4.3.2 Results for ECG, biomedical critical care, and seismic signals

In the case of ECG, the results are similar to those of EEG. In Figure 4.5 we observe that all the multi-channel predictors present better results than the single-channel predictors, and that the average percentage influence increases with the order until it stabilizes. In the case of single-channel the average percentage influence

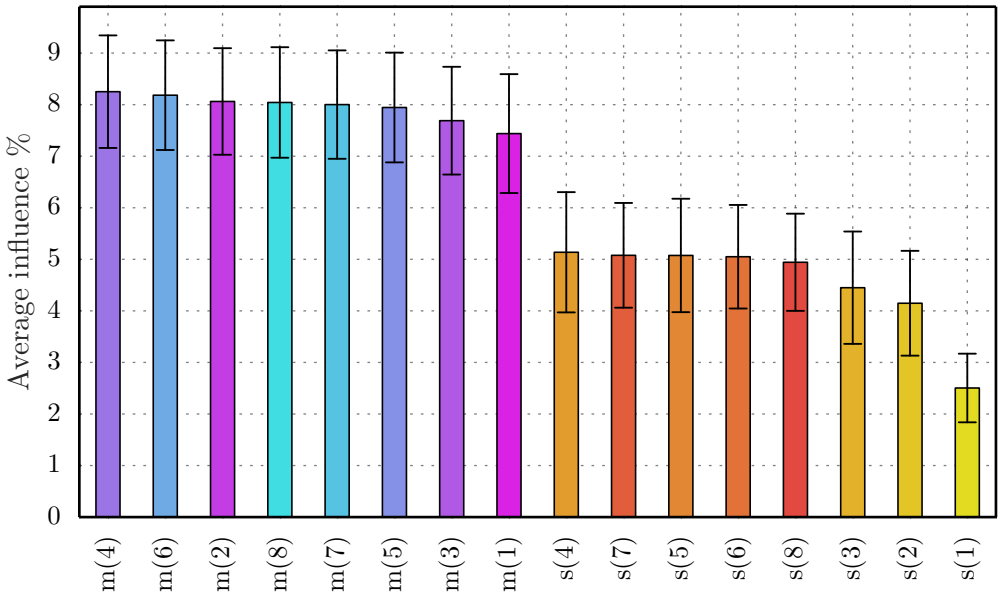


Figure 4.5: Average percentage influence of each predictor of compressor GSC-8 on the *ECG* dataset, ordered from highest to lowest.

stabilizes from threshold order 4 onwards, and in multi-channel we could practically say that it stabilizes from threshold order 2 onwards. Looking at the differences in compression ratio between GSC-2 and GSC-8 in Table 4.1, we notice that the compression ratio of GSC-2 is greater than that of GSC-8 by a difference of 0.12 bps, which represents a relative difference of 2.5%. This slight deterioration may be due to the fact that predictors m(1) and m(2) also belong to GSC-2, and yield good results in terms of average percentage influence, with m(2) being the threshold order predictor. On the side of the single-channel predictors, the predictor s(2) has an average percentage influence relatively lower than that of threshold order 4, which would mean that having the predictor s(4) instead of s(2) could lead to compression ratio improvements.

In Figure 4.6 we present the results of the average percentage influence of each predictor of GSC-8 on the seismology dataset. For the seismic signal, the results are completely different from those obtained for EEG and ECG. The predictors that show the best performance are single-channel, although some multi-channel

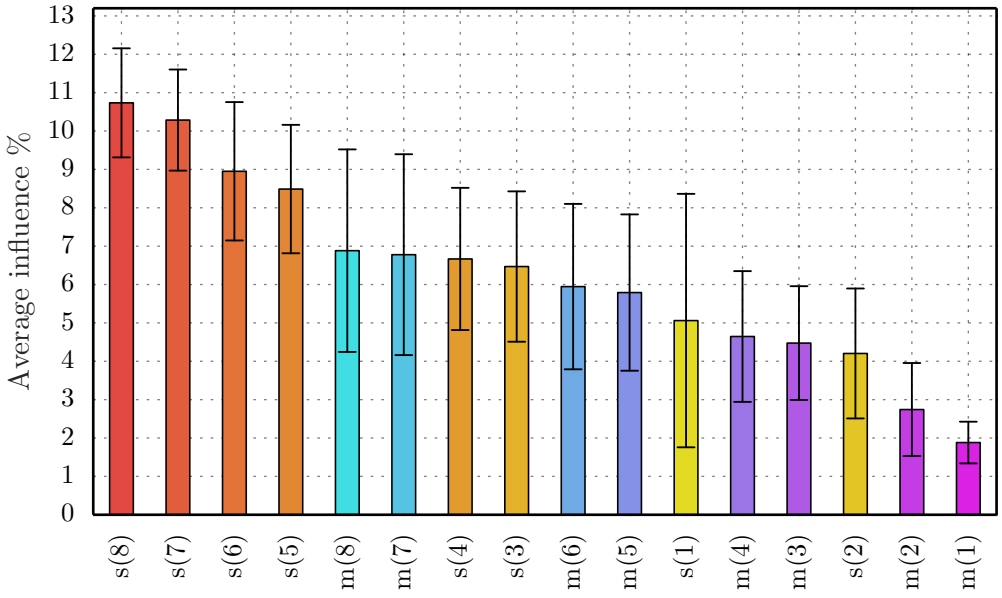


Figure 4.6: Average percentage influence of each predictor of compressor GSC-8 on the *Sism* seismology dataset, ordered from highest to lowest.

predictors, especially the higher order ones, have relatively large average percentage influences. This indicates that the correlation between paired channels seems to be less significant for this type of signals. However, this phenomenon may not be homogeneous across all channel pairs in the coding tree, as there could be pairs of channels with high correlation, leading to an increase in the average percentage influence of the multi-channel predictors. This could be explained by the particular characteristics of this dataset, in which there are very close sensors (in the same geographical location), which record different types of measurements, and others very far from each other. Again, this could be the subject of further study in the future.

In contrast to the results obtained for EEG and ECG datasets, we do not observe that the average percentage influence stabilizes at any order, but it grows steadily with the order. In future work, tests with more predictors, and of a higher order, could be carried out. In addition, it would be interesting to try matching the channels of the files with a criteria other than sensor closeness. There are other characteristics such as the angles at which the seismographs are positioned, that could allow channels to be associated in pairs with greater correlation between them.

Finally, in Figure 4.7 we present the results on the MGH dataset, which consists of files of critical care biomedical signals.

In this case, the results show that the single-channel predictors perform better, in general, than the multi-channel predictors, although the average percentage influ-

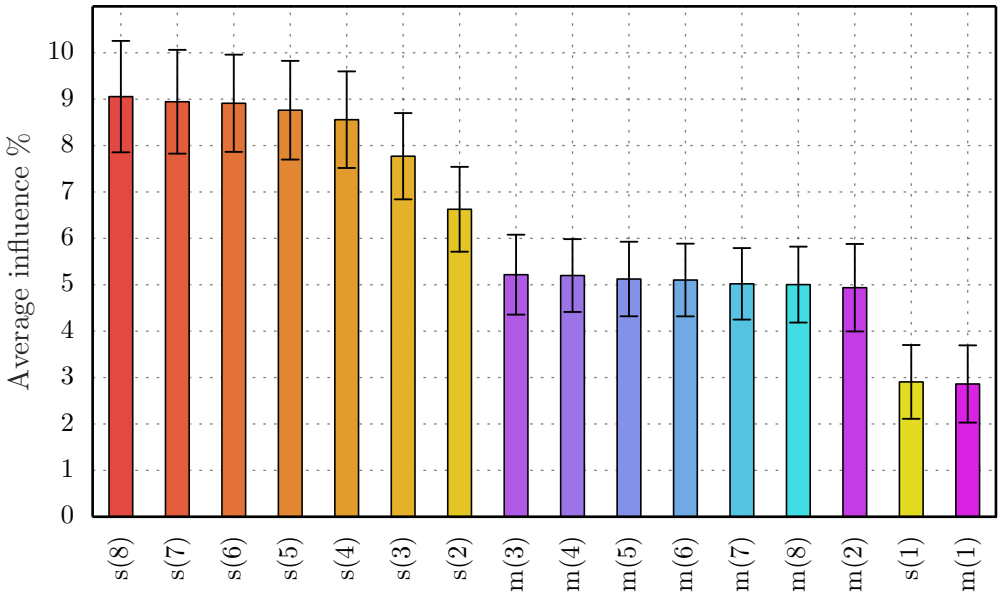


Figure 4.7: Average percentage influence of each predictor of compressor GSC-8 on the *MGH* dataset, ordered from highest to lowest.

ences of the latter are significant. As in the seismology dataset, it could happen that there is large correlation between certain channels, and not in others. Unlike the results in seismology, both in single-channel and multi-channel predictors, the average percentage influences stabilize from a threshold order onwards. In the case of multi-channel, it could be said that the threshold order is 2, while in single-channel, there is a steady improvement with the order, but that it is no longer significant from the threshold order 6 onwards.

4.3.3 Results on short blocks

In order to understand which are the best performing predictors when facing changes in the statistical behavior of a signal, we calculate the average percentage influences on short fragments of sample files. This allows us to visualize what predictors achieve the best results with less adaptation time, since in the analysis of average percentage influence on an entire file, the adaptation moments may be few, and the information on which predictors have the best performance during those moments is lost in the average.

Although the adaptation speed in sporadic situations may not be of utmost relevance in offline file compression systems, it may be important in real time transmission systems.

To perform this analysis, we divide each file from each dataset into multiple files

of 1000 vector samples each, and then we calculate the average percentage influence for each predictor on each dataset. We refer to the results obtained for each dataset as *by-block results*.

Figure 4.8 shows the results of the average percentage influence by blocks obtained for the Comp dataset. As expected, the lowest order predictors, in this case both for single-channels and multi-channels, are the ones with the best performance. It is interesting to note that the average percentage influence decreases rapidly as the orders of the predictors grow. This is because the higher order predictors require a greater number of updates to adapt to the signal. For this reason, it is essential to have low-order predictors in the expert advice algorithm if we want to have predictors with relatively good performance at all times.

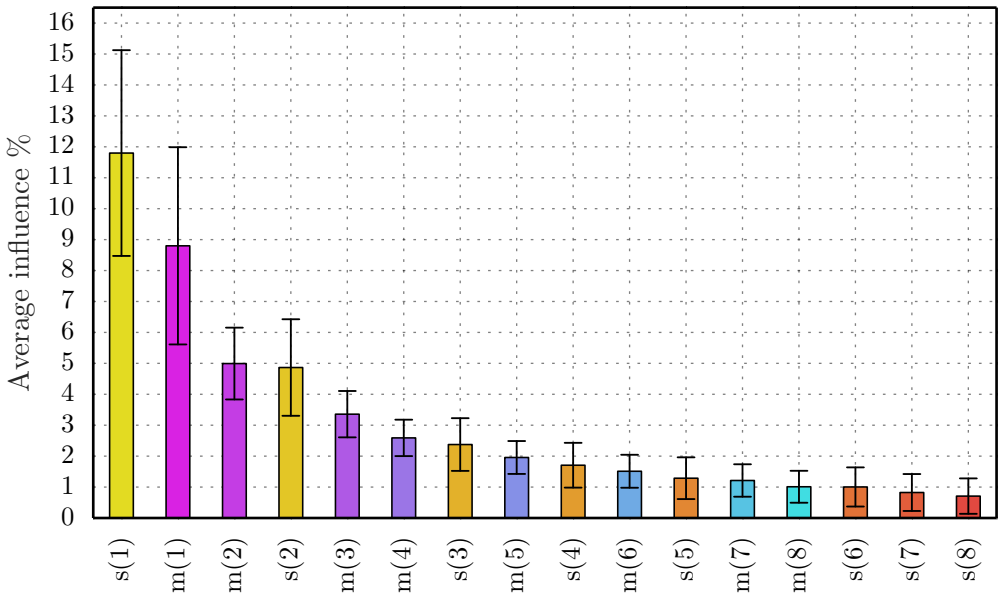


Figure 4.8: Average percentage influence of each predictor of GSC-8 on the *Comp* dataset, divided into files of 1000 vector samples, ordered from highest to lowest.

Another interesting fact is that for the Comp dataset, the low order multi-channel predictors also show good performance in the by-block results. Given the way multi-channel predictors are constructed, a single-channel predictor of the same order has fewer coefficients to be updated, and yet, due to the high correlation between channels, low-order multi-channel predictors rapidly achieve superior results.

The by-block results are different for the dataset Neur. Figure 4.9 shows the by-block results for the Neur dataset and, unlike the results obtained for Comp, all the single-channel predictors show better results than the multi-channel predictors. This may be due to the fact that in the Neur dataset the multi-channel predictors require many more samples than in Comp to adjust their coefficients to the statistical

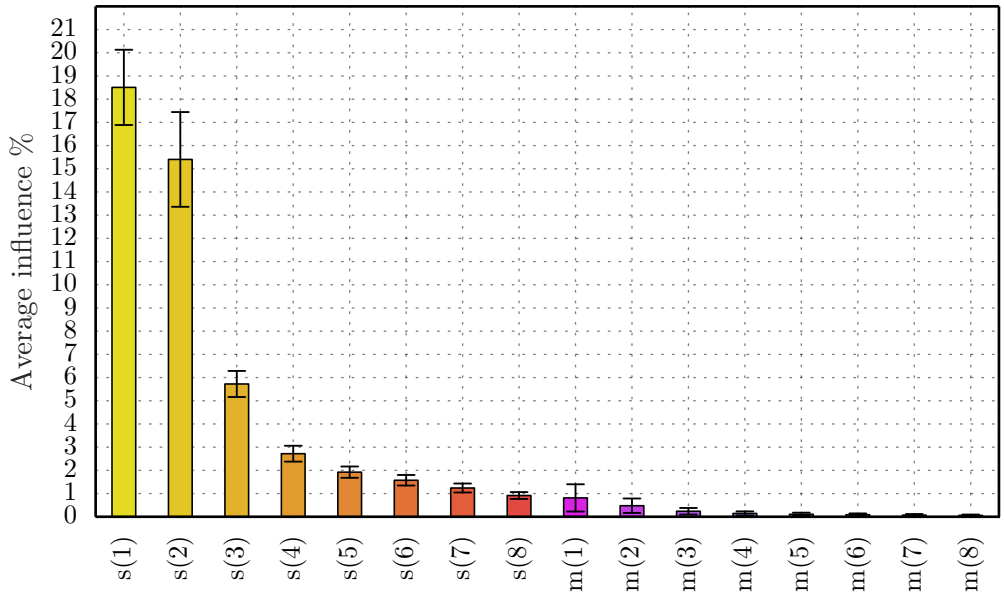


Figure 4.9: Average percentage influence of each predictor of GSC-8 on the *Neur* dataset, divided into files of 1000 vector samples, ordered from highest to lowest.

behaviour of the signal. However, once the coefficients are adjusted the multi-channel predictors outperform the single-channel predictors (see Figure 4.4).

4.4 Criteria for selecting predictors for the expert advice algorithm

The analysis presented throughout this chapter contributes elements that help to understand how to improve the execution times of GSC by selecting a subset of predictors.

In the experiments we observe that, for all signal types, decreasing the number of predictors in the compressor deteriorates compression ratios, but considerably improves execution times. Specifically, lowering the number of predictors from 16 to 4 reduces execution times between 3 and 4 times, while compression ratios deteriorate by a magnitude that depends on the chosen predictors.

Our analysis of the average percentage influence and the average influence distance indicates that for the different types of signals there are certain predictors that perform better than others. Specifically, the performance of the predictors, both for single-channel and multi-channel, is similar from a certain threshold order onwards. Consequently, choosing threshold order predictors, both for single-channel and multi-channel, seems like a good option for minimizing the deterioration in compression performance.

On the other hand, for the set of predictors to perform well at all times, the results on files of 1000 samples per channel show that the low-order predictors, both single-channel and multi-channel, perform considerably better than the high-order predictors when only 1000 adaptation samples are available.

In light of this analysis, we determine a series of general criteria to follow to select a subset of predictors that manages to improve execution times, and, at the same time, minimize the deterioration in compression levels.

- **Select both single-channel and multi-channel predictors** - Although multi-channel predictors show better overall performance, single-channel predictors achieve better performance when the channels of the multi-channel predictor have low correlation between them.
- **Select threshold order predictors** - Threshold order predictors attain a performance similar to that of the predictors of higher orders, and, therefore, the latter are not necessary.
- **Select low-order predictors** - When facing changes in the statistical behavior of the signal, the predictors that show the best performance are the low-order predictors.

Chapter 5

Speck compressor optimized for EEG signals

In this chapter we describe the construction of a compressor optimized for EEG signals, based on GSC, which we call *Optimized Speck Compressor* (OSC). In Section 5.1 we apply the criteria established in Section 4.4 to determine the set of predictors used by OSC in the expert advice algorithm. Then, in Section 5.2, we further optimize the compressor by proposing a selective parameter update process. In Section 5.3 we define a new compressor, called FC, which uses a set of *fixed predictors* (non-adaptive), which are significantly less complex than the Speck predictors. We use FC as a basis for comparison with OSC. In Section 5.4, we present the results of running the new compressors on the datasets presented in Section 3.5.2, and we compare these results to those reported in Section 3.5. Finally, in Section 5.5 we evaluate the trade-off between compression performance and computational efficiency of different integer coding methods for the prediction errors generated by OSC on the datasets presented in Section 3.5.2.

5.1 Selection of predictors

To choose what predictors to use in the expert advice algorithm we follow the criteria proposed in 4.4. The criteria suggests including single-channel and multi-channel predictors, both of high order (threshold) and low order. We decide to choose a set of 4 predictors: a multi-channel low-order predictor, a multi-channel threshold order predictor, a single-channel low-order predictor, and a single-channel threshold order predictor.

The threshold order of the multi-channel predictors for an EEG dataset is either 3 or 4 depending on the dataset. Since, in the long run, threshold predictors perform similarly to higher-order predictors, we decide to use the predictor m(4). For the

single-channel, threshold order is either 4 or 5. For the sake of computational performance, we choose the predictor s(4), as having a single-channel and a multi-channel predictor of the same order, which is a power of two, simplifies various calculations within the Speck update algorithm.

We choose s(1) as the low-order single-channel predictor, since it is the single-channel predictor that achieves the best results on short blocks of 1000 samples, in all EEG datasets. Furthermore, predictor s(1) is very robust as it is not adaptive (it always predicts that the next sample will coincide with the last observed sample of the channel).

Finally, we choose m(2) as a multi-channel low-order predictor, since it is an adaptive predictor that contributes something substantially different from s(1) while being able to adapt its coefficients considerably faster than m(4).

In summary, for OSC we use the following 4 predictors:

- Fixed predictor of order one, where $\hat{x}_i(n) = x_i(n-1)$.
- Single-channel Speck predictor of order 4.
- Multi-channel Speck predictor of order 2.
- Multi-channel Speck predictor of order 4.

5.2 Selective parameters update

As we recall from Section 4.1, a significant portion of the computation time of GSC is spent updating the adaptive parameters, such as the coefficients of the Speck predictors, and the weights in the expert advice algorithm. At the same time, we observe that the adaptive parameters tend to stabilize after a while, changing only when the statistical behavior of the signal is significantly altered. To avoid unnecessary updates, we monitor the overall scheme performance, and update the coefficients of Speck predictors and/or weights of the expert advice algorithm when we observe significant performance deterioration.

For the Speck predictors, we update the coefficients of each predictor of a channel i only when $e_i(n) > \bar{e}_i(n)$, where $e_i(n) \triangleq |\epsilon_i(n)|$ is the absolute value of the prediction error of channel i at time n, and $\bar{e}_i(n)$ is an average of the absolute values of the prediction errors of channel i at time n, estimated using the same method and parameters as in Section 3.12, over the sequence $e_i(j)$, $j < n$.

On the other hand, we observe that once the predictors stabilize, the weights attributed to the predictions by the expert advice algorithm usually do not change, even for long periods of time. Based on this observation, if an update of the weights takes place at time n, the next update is evaluated $T(n)$ samples afterwards, where

$T(n)$ is defined as follows. For $n = 1$ the weights are updated, and we define $T(1) = 1$. Then, if the update of weights is evaluated at time n, $n > 1$, and none of the weights of the predictors are modified, then $T(n) = \min\{2T(n-1), T_{\max}\}$, where $T_{\max}$ is a constant. Otherwise, if some of the weights are modified, then $T(n) = \max\{\frac{T(n-1)}{4}, 1\}$. At time $n + T(n)$ we evaluate if a new update is performed. Essentially, the waiting time between updates, $T(n)$, grows exponentially up to a pre-established maximum, $T_{\max}$, as long as no weight changes. In the event that any change in the weights is detected, the waiting time decreases even faster.

5.3 Fixed predictors

As an even simpler alternative to Speck predictors, we implement a prediction module that uses only fixed predictors. Fixed predictors have the advantage that predictions are calculated with static coefficients (not with an adaptive weight system). Therefore, there is no updating process, offering a simple and fast prediction calculation in exchange of less accurate predictions, and, consequently, lower compression levels.

Each fixed predictor p presented below is conceptually based on a simple geometric extrapolation.

- Fixed predictor of order one, defined as $\hat{x}_i(n) = x_i(n-1)$, i.e., predict that $\hat{x}_i(n)$ is equal to the last observed sample from channel i.

- Fixed predictor of order two, defined as $\hat{x}_i(n) = 2x_i(n-1) - x_i(n-2)$, i.e., predict that $(n, \hat{x}_i(n))$ lies on the line passing through $(n-2, \hat{x}_i(n-2))$ and $(n-1, \hat{x}_i(n-1))$.

- Fixed predictor of order three, defined as $\hat{x}_i(n) = 3x_i(n-1) - 3x_i(n-2) + x_i(n-3)$, i.e., predict that $(n, \hat{x}_i(n))$ lies on the parabola passing through $(n-3, \hat{x}_i(n-3))$, $(n-2, \hat{x}_i(n-2))$, and $(n-1, \hat{x}_i(n-1))$.

- Multi-channel fixed predictor, defined as $\hat{x}_i(n) = x_i(n-1) + x_\ell(n) - x_\ell(n-1)$, where $x_i(n-1)$ is the last sample of channel i, and $x_\ell(n)$ and $x_\ell(n-1)$ are the last two samples of the helper channel ℓ, determined using the coding tree described in Section 3.3, i.e, predict that $(0, n, \hat{x}_i(n))$ lies on the plane passing through $(0, n-1, \hat{x}_i(n-1))$, $(1, n-1, \hat{x}_\ell(n-1))$, and $(1, n, \hat{x}_\ell(n))$. This predictor is inspired by the JPEG-LS image compression standard [102].

We build FC by substituting the fixed predictors for the Speck predictors in OSC.

5.4 Evaluation and analysis of OSC on EEG signals

Table 5.1 presents the results of running the FC and OSC on the EEG datasets described in Section 3.5.2, together with the results obtained for the compressors GSC-8, RLS-7, MP4-ALS, and Flac, on the same datasets. For OSC we use the same parameter settings as for GSC-8, and in addition we define T_{max} = 128. The table also shows, in parenthesis, the percentage relative difference of FC and OSC with respect to GSC-8.

Table 5.1: Compression ratios, and encoding and decoding time per scalar sample, for OSC, FC, GSC-8 RLS-7, MP4-ALS and Flac compressors on the EEG signal datasets. In parenthesis, the percentage relative difference of OSC and FC with respect to GSC-8 is shown.

Datasets	Compressor	CR (bps)	ETPS (μs)	DTPS (μs)
Phys-a	OSC	4.81 (0.4)	0.08	0.09
	FC	5.08 (6.1)	**0.06**	0.06
	GSC-8	4.79	0.47	0.47
	RLS-7	**4.70**	0.93	0.92
	MP4-ALS	5.57	1.06	1.06
	Flac	6.31	0.11	**0.02**
Phys-b	OSC	4.94 (0.2)	0.09	0.09
	FC	5.18 (5.1)	**0.06**	0.07
	GSC-8	4.93	0.47	0.48
	RLS-7	**4.79**	0.93	0.93
	MP4-ALS	5.72	1.07	1.09
	Flac	6.85	0.12	**0.02**
Comp	OSC	5.47 (0.4)	0.08	0.09
	FC	5.90 (8.3)	**0.05**	0.06
	GSC-8	5.45	0.46	0.47
	RLS-7	**5.42**	0.92	0.91
	MP4-ALS	5.90	1.06	1.0
	Flac	6.40	0.08	**0.01**
Neur	OSC	3.81 (1.3)	0.07	0.08
	FC	4.35 (15.7)	**0.05**	0.06
	GSC-8	3.76	0.45	0.45
	RLS-7	**3.58**	0.89	0.89
	MP4-ALS	3.86	1.07	1.07
	Flac	4.45	0.07	**0.01**
BCI-a	OSC	5.34 (0.9)	0.08	0.08
	FC	5.96 (12.7)	**0.05**	0.06
	GSC-8	5.29	0.47	0.47
	RLS-7	**5.21**	0.92	0.92
	MP4-ALS	5.82	1.06	1.07
	Flac	6.37	0.08	**0.01**
BCI-b	OSC	6.97 (0.6)	0.08	0.08
	FC	7.41 (6.9)	**0.05**	0.06
	GSC-8	**6.93**	0.47	0.48
	RLS-7	**6.93**	0.93	0.92
	MP4-ALS	7.99	1.07	0.88
	Flac	8.71	0.08	**0.02**

As expected, the more complex GSC-8 outperforms both FC and OSC in terms

of compression ratio, on all datasets, which means that RLS-7 continues to achieve the best compression ratio results on all the datasets. Specifically, the maximum percentage relative difference of the compression ratio of OSC with respect to GSC-8 is 1.3%, for the Neur dataset, while in the rest of the datasets the percentage relative difference is less than 1%, being 0.6% the average among all the datasets.

On the other hand, the results obtained by FC show that the average percentage relative difference of FC with respect to GSC-8 is 9.13%, significantly higher than that obtained for OSC. This shows how the use of predictors that are too simple has a significant impact on compression performance.

In terms of ETPS and DTPS, OSC is much faster than GSC-8 and RLS-7, and even faster for encoding than Flac (ETPS). OSC reaches an encoding time per sample of 0.08 ± 0.01 μs, a time almost 6x times faster than that of GSC-8, 0.46 ± 0.01 μs, and almost 12x times faster than RLS-7, 0.91 ± 0.02 μs.

To graphically compare OSC against other compressors, in Figure 5.1 we show, for each compressor, a point on the plane that represents the compression ratio on the X axis, and the ETPS on the Y axis. Both measurements are averaged over

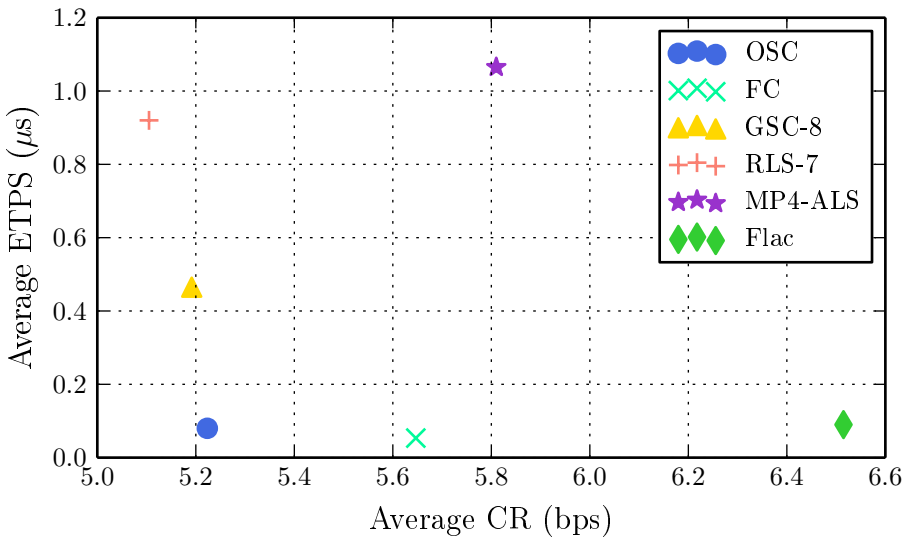


Figure 5.1: Average ETPS vs. average compression ratio of the different compression algorithms on the EEG datasets.

all EEG datasets. The figure clearly reflects the advantage in terms of ETPS that OSC, FC, and Flac compressors have over the rest. On the other hand, in terms of compression ratio, the compressors that achieve the best results are RLS-7, GSC-8 and OSC. We notice that OSC achieves a good trade-off between compression ratio and computational efficiency, significantly improving encoding time compared to GSC-8 and RLS-7, and, at the same time, achieving very similar compression ratios.

Table 5.2 shows the results of running OSC on non EEG signals, namely the ECG, MGH, and Sism datasets. For the ECG dataset, the results obtained by OSC are similar to those obtained in EEG datasets. This makes sense, since we recall from Figure 4.5 that the predictors s(4) and m(4), which are used by OSC, are threshold predictors for the ECG dataset. On the other hand, the results show that on the MGH and Sism datasets OSC has a percentage relative difference with respect to GSC-8 of 3.4% and 3.6% respectively, which are significantly larger than the average 0.6% observed on EEG datasets. This fact is consistent with the average percentage influences on the Sism and MGH datasets shown in figures 4.6 and 4.7, respectively. In both Sism and MGH the single-channel predictors are the ones with the best performance. Specifically, in MGH the single-channel threshold predictor is s(6), which is not used by OSC, while in the Sism dataset the average percentage influence of the predictors steadily increases with the order, being s(8) the best performing predictor, which is not used by OSC either. However, although OSC does not use the best performance predictors for MGH and Sism, the compressor still achieves similar or better results than MP4-ALS and Flac.

Table 5.2: Compression ratios, and encoding and decoding time per scalar sample for OSC GSC-8 RLS-7, MP4-ALS and Flac compressors on the ECG, critical care, and seismic signal datasets. In parenthesis, the percentage relative difference between OSC and GSC-8 is shown.

Datasets	Compressor	CR (bps)	ETPS (μs)	DTPS (μs)
ECG	OSC	4.85 (1.0)	**0.08**	0.09
	GSC-8	4.80	0.43	0.44
	RLS-7	**4.78**	0.81	0.81
	MP4-ALS	5.07	1.09	1.09
	Flac	5.45	0.09	**0.02**
MGH	OSC	2.74 (3.4)	**0.07**	0.08
	GSC-8	2.65	0.43	0.43
	RLS-7	**2.61**	0.96	0.97
	MP4-ALS	2.82	1.00	0.98
	Flac	2.77	**0.07**	**0.01**
Sism	OSC	8.03 (3.6)	**0.10**	**0.10**
	GSC-8	7.75	0.48	0.48
	RLS-7	**7.69**	0.95	0.94
	MP4-ALS	8.03	1.04	1.05
	Flac	-	-	-

As a last remark, we point out that the tested EEG datasets cover a wide range of different scenarios, with sampling frequencies between 160 Hz and 1000 Hz, and number of channels between 31 and 118. Note also that even for signals for which OSC was not designed, like ECG, Sism, and MGH, the compressor achieves relatively good results.

5.5 Integer coding method for prediction errors

Another part of the compression algorithm that can be addressed to improve the computational efficiency of OSC (and other compression algorithms) is the integer coding method used to encode the prediction errors.

As we explain in Section 3.1, in predictive coding, if the predictions are accurate, the absolute values of the prediction errors tend to be close to zero, with histograms that resemble two-sided geometric distributions. This makes Golomb codes an attractive choice both for its theoretical compression performance guarantee [75] and its computational efficiency. Adaptive Golomb coding is the integer coding method used for the prediction errors by Flac, MP4-ALS, RLS and GSC. However, there are integer coding methods simpler than adaptive Golomb coding, such as Simple9 [3], Simple16 [108], and PForDelta [112], which have been shown to achieve better results than Golomb coding on some scenarios, such as compression of *longest common prefix* arrays [13], both in terms of compression ratio and computational performance. In this section we evaluate the integer coding methods Simple9, Simple16, and PForDelta, on the sequence of prediction errors generated by OSC for the datasets presented in Section 3.5.2. Specifically, we compare adaptive Golomb coding, as implemented in [15], against the integer coding algorithms Simple9, Simple16, and PForDelta, as implemented in the high quality C++ library FastPFor [67][1].

Table 5.3 shows the compression ratio in bits per sample and the average encoding time per sample obtained by each tested integer coding method, on each of the tested datasets. The table also shows the percentage relative difference between each of the tested methods and adaptive Golomb coding.

The results show that, as expected, adaptive Golomb coding achieves the best compression ratios on each of the tested datasets. Among Simple9, Simple16, and PForDelta, the method that achieves the best compression ratio on average, is Simple16, with a relative difference with respect to Golomb between 13.5% and 17.9%. In terms of encoding speed, Simple16 is on average 71.0% faster than Golomb, while the fastest method is PForDelta, with an average ETPS of 2.92 nanoseconds, which is on average 77.8% faster than adaptive Golomb coding. In terms of compression ratio PForDelta has a relative difference with respect to Golomb between 13.6% and 21.3%. Among the tested methods Simple9 achieves the worse compression ratio, with a relative difference with respect to Golomb between 22.8% and 29.3%, and it is slower than PForDelta, with an average ETPS of 3.02 nanoseconds, which is on average 77,0% faster than adaptive Golomb coding.

We observe that replacing adaptive Golomb coding by methods like Simple16,

[1] We select method *fastpfor128* of FastPFor as the implementation for PForDelta.

Table 5.3: Compression ratios and encoding time per sample of the sequence of prediction errors generated by OSC, for the integer coding methods adaptive Golomb, Simple9, Simple16, and PForDelta, on the datasets presented in Section 3.5.2. The percentage relative difference of the compression ratio of each method with respect to adaptive Golomb coding is shown in parentheses.

Dataset	Compressor	CR (bps)	ETPS (ns)
Phys-a	Golomb	**4.81**	12.94
	Simple9	6.22 (29.3)	**3.14**
	Simple16	5.63 (17.0)	3.91
	PForDelta	5.83 (21.2)	3.15
Phys-b	Golomb	**4.94**	12.84
	Simple9	6.36 (28.7)	3.12
	Simple16	5.70 (15.4)	3.92
	PForDelta	5.83 (18.0)	**3.00**
Comp	Golomb	**5.47**	13.52
	Simple9	6.95 (27.1)	3.03
	Simple16	6.29 (15.0)	3.82
	PForDelta	6.22 (13.7)	**2.72**
Neur	Golomb	**3.81**	11.69
	Simple9	4.86 (27.6)	**2.91**
	Simple16	4.49 (17.8)	3.21
	PForDelta	4.62 (21.3)	3.28
BCI-a	Golomb	**5.34**	13.17
	Simple9	6.84 (28.1)	3.07
	Simple16	6.06 (13.5)	3.82
	PForDelta	6.16 (15.4)	**2.80**
BCI-b	Golomb	**6.97**	14.90
	Simple9	8.56 (22.8)	2.85
	Simple16	8.22 (17.9)	4.16
	PForDelta	7.92 (13.6)	**2.59**

and PForDelta, can improve the execution time of a compression algorithm, in exchange for significantly deteriorating the compression performance of the algorithm when compared to adaptive Golomb coding.

The improvement in the overall computational efficiency of an algorithm depends on the relative weight of the integer coding method with respect to the total compression time. In our implementation of OSC, specifically, the time for encoding the prediction errors, without considering input/output times, determined with the profiling tool *gprof*[1], is approximately 19.0% of the execution time of the compression algorithm. Therefore, an improvement of 77.8% in the encoding time per prediction error of the integer coding method represents an overall improvement of 14.8% in total compression time.

[1] https://sourceware.org/binutils/docs/gprof/

Part II

Nanopore DNA sequencing data compression

Chapter 6

ENANO: Encoder for NANOpore FASTQ files

In Part II of the thesis we turn our focus to the compression of DNA sequencing data. Specifically, we study the efficient compression of data generated by nanopore sequencing technologies [93] and stored in the FASTQ format [22].

In this chapter we introduce ENANO, a novel lossless compression algorithm especially designed for nanopore sequencing FASTQ files, which mainly focuses on the compression of the quality scores. Although ENANO can also compress FASTQ files generated with SGS sequencing technologies[1], it is not its intended use.

The rest of the chapter is organized as follows: in Section 6.1 we introduce the general compression techniques used by ENANO. Specifically, we give a brief description of *arithmetic coding [86]*, and how it can be combined with *context modeling [87]*. Next, in Section 6.2 we describe the compression scheme of ENANO, where we present a novel quality score compression algorithm. Finally, in Section 6.3 we evaluate the performance of ENANO in terms of compression, speed, and memory requirements, by comparing it against other FASTQ compressors that achieve state of the art compression performance, on a series of publicly available datasets.

6.1 Arithmetic coding and context modeling

Arithmetic coding is a widely used general compression technique. In particular, it has been successfully used for compression of DNA sequencing data in the FASTQ format in, e.g,[9, 89, 17]. We describe how it works next.

Let $x_1^n = x_1, x_2, \ldots, x_n$ be a sequence of symbols to be encoded, where each symbol x_i belongs to an alphabet $\mathcal{X}$. For example, the alphabet of symbols for the base call sequences of the reads of a FASTQ file is $\{A, C, G, T, N\}$. To encode x_1^n,

[1] Paired-end reads from SGS data are treated as single-reads by ENANO.

an arithmetic encoder receives as input a sequence of n probability distributions over $\mathcal{X}$, one for each symbol x_i in position i, $1 \leq i \leq n$, of the sequence, where each distribution is estimated as a function of other symbols encoded before x_i. In turn, it sequentially produces a binary string, which can be losslessly decoded by an arithmetic decoder into the original sequence x_1^n, as long as it has access to the same sequence of probability distributions. Assuming equal initial conditions for the encoder and the decoder, this is indeed the case, as the probability distribution for x_i depends only on symbols that have been decoded before x_i. If $P(x_1^n)$ is the total probability assigned, sequentially, to the sequence x_1^n, then the binary sequence produced by the encoder has length $-\log_2 P(x_1^n) + O(1)$, which is the shortest possible, in a well defined sense, up to an additive constant [86].

To estimate a probability distribution over $\mathcal{X}$ for each symbol of the sequence, a commonly used technique is *context modeling* [87]. A context model assigns a *context*, selected from a pre-defined finite set of contexts C, to each symbol position i, $1 \leq i \leq n$, as a function of symbols that are encoded before x_i. Each context from C is an abstract class grouping sequence symbols that, according to certain probabilistic model, we expect to follow the same probability distribution. In other words, denoting by $\mathcal{C}_i$ the context assigned to position i, we model symbols x_i and x_j as samples drawn from the same probability distributions whenever $\mathcal{C}_i = \mathcal{C}_j$. In this case, we say that the symbols x_i and x_j *occur* in the same context. The probability distribution over $\mathcal{X}$ associated to each context of C is adaptively determined by statistics calculated from the symbols that occur in that context, sequentially, as the input sequence is compressed. The compressor updates the statistics associated to the context $\mathcal{C}_i$ *after* encoding the symbol x_i; the decompressor works in lockstep, symmetrically, updating the same statistics upon decoding x_i. The definition of a context model poses the same kind of trade-off as the choice of predictors discussed in Chapter 4. The choice of number of contexts in C is crucial for the performance of the compression system. A number of contexts that is too small may be insufficient to capture complex statistical dependencies in the data, leading to a poor compression performance. An excessively large context set, on the other hand, may lead to a large model cost, caused by context statistics dilution, which may also result in poor compression performance. In addition, from a computational perspective, a large context set translates, in general, into high memory consumption.

The context models that we present in the sequel are designed with this model size optimization in mind.

6.2 Compression scheme of ENANO

We recall from Section 1.3 that the result of a sequencing process is a set of reads, where each read is comprised of: an identifier string, which is a short free text segment; a base call string (also referred to as base call sequence[1]); and a quality score sequence, of the same length as the base call string. For nanopore files, the alphabet size of the quality scores is 94, which are represented with ASCII codes 33 to 126.

Due to the different nature of read identifiers, base call strings, and quality score sequences ENANO applies a different compression method to each part. Hereafter we refer to these parts as *streams*. To compress each stream, ENANO uses context modeling combined with arithmetic coding, as explained in Section 6.1.

For each stream of the FASTQ file, we use an especially designed context model, which we briefly describe next.

- **Read identifiers:** The read identifier (ID) compression in ENANO is based on Fqzcomp [9]. As the IDs of consecutive reads tend to share the same structure and be similar to each other, the algorithm exploits this when encoding an ID by first describing the parts in which the current ID coincides with the previous ID, and then describing the remaining differences. Specifically, the algorithm first encodes the lengths of the longest prefix and suffix common to both IDs. Each length is encoded using as context the corresponding length in the encoding of the previous ID. The total length of the ID is also encoded using as context the previous ID length. The part of the ID that is not covered by the longest prefix or suffix shared with the previous ID is split into words, delimited by a white space or a colon character. This word partition is used, in turn, to determine a context, derived from the previous ID, for each symbol remaining to encode in the current ID. Full details can be found in [9].

- **Base call string:** The sequence is modeled as a kth order Markov process, using the previous k base calls as the context. The value of k is determined at run-time as an input parameter of the encoder.

- **Quality score sequence:** We propose a novel context model that exploits the statistical dependence among neighbor quality scores and also between quality scores and the corresponding base call string. The proposed compression method is explained in full detail in Section 6.2.1.

To speed up the algorithm, ENANO parses and compresses the FASTQ file in blocks. As nanopore sequencing generates reads of variable length, the sizes of

[1] we use the term *string*, which is customary for pattern matching algorithms that we apply extensively in Chapter 7

the blocks of the file are sequentially and dynamically determined as the reads are parsed, so that block boundaries always fall on read boundaries, and, therefore, blocks always contain a whole number of reads. We set the maximum block size to 10 MB by default. Each block contains the three streams discussed above: read identifiers, base call strings, and quality score sequences. For reasons of causality, which will be discussed in the sequel, we assume that the base call strings are described before the corresponding quality score sequences. For each block of the file, the streams are encoded independently, each one with an arithmetic encoder. Once the encoding process of each stream is finished, the output for each stream is sequentially written into the output file, by first writing the size (in bytes) of the encoded stream, followed by the encoded stream itself. This ensures the decoder can successfully retrieve each encoded stream of the block, even though the sizes of the encoded streams vary between blocks.

Once enough samples have been encoded for a given context, the small improvement in compression performance obtained by updating the model does not pay off for the computational cost involved in this update. This is due to the fact that, as observed, the estimated probability distributions conditioned on each context tend to stabilize. Thus, it may be advantageous to stop adaptation and freeze the distributions after processing a certain number of blocks. Aside from the complexity advantage, fixing the probability distributions of the context models allows for easy parallelization of the proposed encoding and decoding algorithms, as we explain in Section 6.2.2. Therefore, ENANO operates in two modes:

- **Fast mode (Fast)** (default mode): The file is encoded while the statistics are adaptively updated for each context model, for a certain number of blocks of the file. Afterwards, the statistics are fixed, allowing for fast parallel compression and decompression. The number of blocks processed before freezing is a configurable parameter, with a default value determined experimentally, as will be discussed in Section 6.3.2, and is signaled to the decoder.

- **Max Compression mode (MC)**: The statistics are adaptively updated throughout the full encoding (and decoding) of the file.

6.2.1 Quality score sequence compression

Denote by q_i the quality score at position i within a read that we wish to encode, and by b_i the base call symbol at position i. We define the *neighborhood* of q_i as consisting of two parts: the pair $\mathcal{Q}_i = (q_{i-1}, q_{i-2})$ of quality scores immediately preceding q_i, and the ℓ-tuple, $\ell \geq 0$, of base call symbols with indices closest to i, namely, $\mathcal{B}_i^{\ell} = (b_{i-\lfloor(\ell-1)/2\rfloor}, \ldots, b_i, \ldots, b_{i+\lceil(\ell-1)/2\rceil})$, with prescribed conventions for

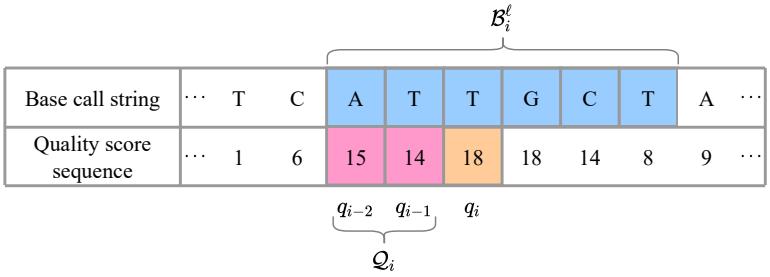


Figure 6.1: Example of neighbor values for $q_i = 18$ and $\ell = 6$.

border cases[1]. We refer to $\mathcal{Q}_i$ and $\mathcal{B}_i^\ell$ as the *quality score neighborhood* and the *base call neighborhood* of q_i, respectively. An example for $\ell = 6$ is shown in Figure 6.1. To ensure successful decoding, we describe the base call string *before* the quality score sequence. Thus, base call symbols at, previous to, and following position i can be used for the base call neighborhood, and are available to the decompressor when decoding q_i.

Using the neighborhood of q_i directly as a context for the encoding of q_i can produce state of the art compression ratios if the compressed files are large enough [32]. However, the number of possible neighborhoods grows exponentially with ℓ, which can eventually lead to poor compression performance if the number of symbols encoded in each context is not statistically significant to estimate an accurate probability distribution of the quality scores given the context. Moreover, having a large number of contexts directly correlates with the compressor having high memory consumption as it has to store a probability distribution for each context. For example, if the alphabet sizes are 5 and 94 for the base calls and the quality scores, respectively (which is the case for nanopore FASTQ files), for $\ell = 6$ the number of possible neighborhoods is larger than 138 million.

Here we propose a novel *neighborhood grouping* algorithm that maps the neighborhood of q_i to a reduced set of contexts, which is able to quickly capture the relevant information from the neighboring values to achieve state of the art compression performance, while keeping memory consumption low.

6.2.1.1 Neighborhood grouping algorithm

In this section we define a neighborhood grouping algorithm that receives as input the neighborhood values $\mathcal{Q}_i = (q_{i-1}, q_{i-2})$, and $\mathcal{B}_i^\ell$, and outputs a context, $\mathcal{C}_i$. We present the steps involved in the calculation of $\mathcal{C}_i$ in Algorithm 3, accompanied with a graphical representation in Figure 6.2; the arrow labels in the figure refer to step

[1]For $i \leq 1$, we arbitrarily let $q_{i-1} = 0$ and $q_{i-2} = 0$, and $b_j = A$ for all negative values of j. Similarly, we let $b_j = A$ for all values of j that are beyond the end of a read.

numbers in the algorithm. We denote by $\lceil x \rfloor$ the rounding of x to the nearest integer (rounding up in case of ties). Next, we present a brief overview of the algorithm, followed by a detailed explanation afterwards.

Algorithm 3: Algorithm for determining the context $\mathcal{C}_i$ for a quality score q_i.

Input : Neighborhood values $\mathcal{Q}_i = (q_{i-1}, q_{i-2})$, and $\mathcal{B}_i^\ell$
Output: The context associated with q_i, $\mathcal{C}_i$

Calculate $\hat{q}_{i-1} = Q_{\text{bin}}(q_{i-1})$
Calculate $\hat{d}_i = Q_{\text{dif}}(q_{i-2} - q_{i-1})$
Let $\hat{\mathcal{Q}}_i = (\hat{q}_{i-1}, \hat{d}_i)$
Get $\hat{\mathcal{B}}_i^\ell$ from $\mathcal{B}_i^\ell$ by substituting A for N
Let $\mathcal{N}_i^\ell = (\hat{\mathcal{B}}_i^\ell, \hat{\mathcal{Q}}_i)$
Retrieve the $\bar{q}_{\mathcal{N}}$ associated to $\mathcal{N}_i^\ell$
Retrieve the $\bar{e}_{\mathcal{N}}$ associated to $\mathcal{N}_i^\ell$
Retrieve the $\bar{e}_{\hat{\mathcal{Q}}}$ associated to $\hat{\mathcal{Q}}_i$
Calculate $\hat{a} = Q_{\text{bin}}(\lceil \bar{q}_{\mathcal{N}} \rfloor)$
Calculate $\hat{r} = Q_R(\bar{e}_{\mathcal{N}}, \bar{e}_{\hat{\mathcal{Q}}})$
Let $\mathcal{C}_i = (\hat{\mathcal{Q}}_i, \hat{a}, \hat{r})$

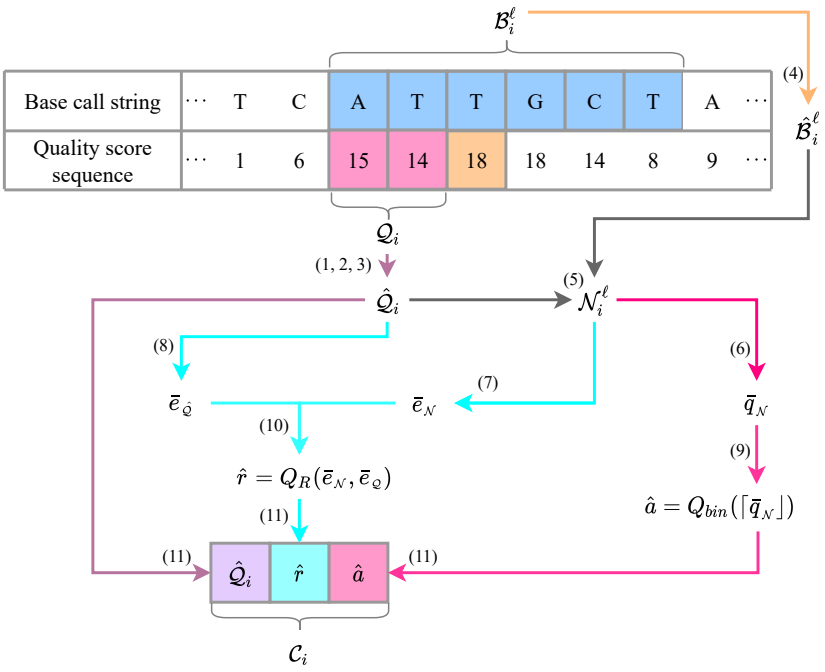


Figure 6.2: Diagram of the calculation of a context, $\mathcal{C}_i$, to encode q_i. Each arrow is numbered as the step it represents in Algorithm 3.

The algorithm starts by performing a quantization of the quality score neighborhood $\mathcal{Q}_i$. This is done through steps 1 to 3, by defining two quantization functions, Q_{bin} and Q_{dif}, that map q_{i-1} and the difference $(q_{i-2} - q_{i-1})$ to a reduced set of predetermined *bins*, respectively. The quantized version of $\mathcal{Q}_i$, $\hat{\mathcal{Q}}_i$, is defined as a vector comprised of the results of both scalar quantization functions. In step 4 we build a *restricted base call neighborhood*, denoted by $\hat{\mathcal{B}}_i^\ell$, by replacing every occurrence of N in $\mathcal{B}_i^\ell$ with the regular base call symbol A. Let $\mathcal{N}_i^\ell = (\hat{\mathcal{B}}_i^\ell, \hat{\mathcal{Q}}_i)$. Steps 6 through 10 perform a joint grouping of base call and quality score neighborhoods, by mapping the set of possible values for $\mathcal{N}_i^\ell$ into a reduced set, which we define as the set of contexts for the encoding of quality scores. To this end, we maintain three statistics $\bar{q}_{\mathcal{N}}$, $\bar{e}_{\mathcal{N}}$, and $\bar{e}_{\hat{\mathcal{Q}}}$, which are used to derive the values $\hat{a}$ and $\hat{r}$ in steps 9 and 10, respectively. We define the context $\mathcal{C}_i$ for a quality score q_i as the triplet $(\hat{\mathcal{Q}}_i, \hat{a}, \hat{r})$.

With the proposed grouping algorithm for the neighborhood values of q_i the number of contexts is constant, as it does not depend on the value of ℓ, and the total number of contexts in ENANO's implementation is 7,168.

In the rest of the section, we explain each step of the algorithm in more detail. For conciseness, we sometimes omit the sub-index i in some intermediate computations leading to $\mathcal{C}_i$, when there is no risk of ambiguity.

Quantization of quality score neighborhoods (steps 1 through 3 of Algorithm 3): The algorithm starts by quantizing $\mathcal{Q}_i = (q_{i-1}, q_{i-2})$. First, we define a partition of the space of possible values of q_{i-1}, namely, $\{0, 1, \ldots, 93\}$, into a sequence of Q *bins*, with indices $0, 1, \ldots, Q-1$, respectively. The value of q_{i-1} is quantized in Step 1 by a function Q_{bin} mapping q_{i-1} to the index $\hat{q}_{i-1}$ of the bin it belongs to. For ENANO we choose $Q = 16$, with the specific bins specified in Table 6.1. Notice that each quality score in the range $0 \ldots 11$ defines its own individual bin. With these definitions we have, for example, $Q_{\text{bin}}(25) = 14$.

Bins	Quality Scores
$0 \ldots 11$	$0 \ldots 11$
12	$12, 13$
13	$14, 15, 16, 17$
14	$18, 19, \ldots, 30$
15	$31, 32, \ldots, 93$

Table 6.1: Quality score bins used for the function Q_{bin}.

We observe, experimentally, that values of q_{i-1} and q_{i-2} that are close to each other are much more frequent than pairs where the values are significantly different. Therefore, rather than quantizing q_{i-2} independently, Step 2 quantizes the difference

$q_{i-2} - q_{i-1}$. We define $\hat{d}_i = Q_{\text{dif}}(q_{i-2} - q_{i-1})$, where $Q_{\text{dif}} : [-93..93] \to [0..D-1]$ is a function that maps each possible difference between quality scores to one of D bins. For ENANO, we choose $D = 7$ and we define Q_{dif} so that larger differences are quantized more heavily. Specifically, the bins we use for Q_{dif} in ENANO are detailed in Table 6.2.

Bins	Difference value
0	0
1	-1
2	1
3	$-4, -3, -2$
4	$2, 3, 4$
5	$-93, \cdots, -5$
6	$5, \cdots, 93$

Table 6.2: Bins used for the function Q_{dif}.

We define the quantized version of $\mathcal{Q}_i$, $\hat{\mathcal{Q}}_i$, as the pair $(\hat{q}_{i-1}, \hat{d}_i)$; this is calculated in Step 3. In Figure 6.3 we illustrate the steps taken in Algorithm 3 to perform the quality score neighborhood quantization.

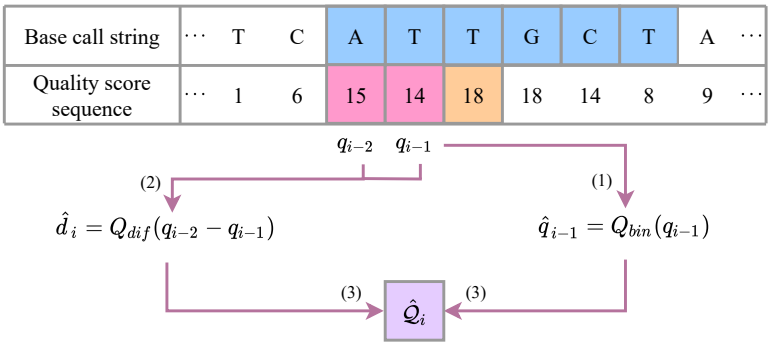


Figure 6.3: Quality score neighborhood quantization. Each arrow is numbered as the step it represents in Algorithm 3.

Grouping of base call neighborhoods (step 4 of Algorithm 3): We restrict the alphabet of the base call string neighborhoods to a set of size 4, by disregarding letters N. Specifically, in ENANO we build a *restricted base call neighborhood, denoted as* $\hat{\mathcal{B}}_i^\ell$, by substituting A for N in $\mathcal{B}_i^\ell$. Since the frequency of the letter N is usually insignificant compared to the frequencies of the other letters, this allows for significantly reducing the number of neighborhoods with no significant effect on the compression ratio. Notice that this basecall replacement is only performed on the (previously encoded/decoded) base call string with the only purpose of neighborhood grouping for quality score context determination. Thus, the original basecall

stream is decoded without any loss of information.

Joint grouping of base call and quality score neighborhoods (steps 5 through 10 of Algorithm 3): Let $\mathcal{N}_i^\ell = (\hat{\mathcal{B}}_i^\ell, \hat{\mathcal{Q}}_i)$. We maintain two statistics, $\bar{q}_\mathcal{N}$ and $\bar{e}_\mathcal{N}$, for each value of $\mathcal{N}_i^\ell$, and one statistic, $\bar{e}_{\hat{\mathcal{Q}}}$, for each value of $\hat{\mathcal{Q}}_i$. These statistics are calculated using integer-only arithmetic as exponentially weighted moving averages as in (3.12), which we recall that, for a sequence of values $z_1^n = z_1, z_2, \ldots, z_n$, is defined recursively as

$$\bar{z}_j = (1-\beta)\bar{z}_{j-1} + \beta z_j\,, \quad 1 \leq j \leq n\,, \tag{6.1}$$

where β is a parameter, $0 < \beta < 1$, and we let $\bar{z}_0 = 0$. In ENANO we use this procedure, with $\beta = 2^{-4}$, for the calculation of $\bar{q}_\mathcal{N}$, $\bar{e}_\mathcal{N}$, and $\bar{e}_{\hat{\mathcal{Q}}}$, each corresponding to a different choice of underlying sequence z_1^n.

For $\bar{q}_\mathcal{N}$, the sequence z_1^n is comprised of quality scores that occur under the same quantized quality score neighborhood, $\hat{\mathcal{Q}}_i$, and the same restricted base call string neighborhood, $\hat{\mathcal{B}}_i^\ell$, i.e., quality scores q_i that occur under the same value of $\mathcal{N}_i^\ell$. The encoding (resp. decoding) algorithm maintains a table with the current value of $\bar{q}_\mathcal{N}$ for each possible value of $\mathcal{N}_i^\ell$, and exactly one entry of this table is updated using (6.1) each time a quality score is encoded (resp. decoded). Therefore, the implementation of Step 6 in Algorithm 3 amounts to retrieving the entry of this table that is associated to the value of $\mathcal{N}_i^\ell$ defined in Step 5.

For the calculation of $\bar{e}_\mathcal{N}$, each value of the sequence z_1^n is the absolute difference between such a quality score, q_i, and the value of $\bar{q}_\mathcal{N}$ associated to $\mathcal{N}_i^\ell$ at the time of the encoding (resp. decoding) of that particular quality score. Similarly to $\bar{q}_\mathcal{N}$, the algorithm maintains a table with the current value of $\bar{e}_\mathcal{N}$ for each possible value of $\mathcal{N}_i^\ell$, and one of these entries is updated using (6.1) with each encoding (resp. decoding) of a quality score. Analogously to Step 6, Step 7 consists of retrieving the value of $\bar{e}_\mathcal{N}$ from this table.

For the calculation of $\bar{e}_{\hat{\mathcal{Q}}}$, a table is maintained with the current value of $\bar{e}_{\hat{\mathcal{Q}}}$ for each possible quantized quality score neighborhood. Each time a quality score q_i is encoded (decoded) and, subsequently, the value of $\bar{e}_\mathcal{N}$ is calculated for $\mathcal{N}_i^\ell$, this value of $\bar{e}_\mathcal{N}$ is in turn averaged, taking the role of an element of the sequence z_1^n in (6.1), to yield the current value of $\bar{e}_{\hat{\mathcal{Q}}}$ for quantized quality score neighborhood $\hat{\mathcal{Q}}_i$. Step 8 of Algorithm 3 obtains $\bar{e}_{\hat{\mathcal{Q}}}$ by looking up in this table, using the value of $\hat{\mathcal{Q}}_i$ calculated in Step 3 as a key.

Our neighborhood grouping algorithm relies on quantizations of $\bar{q}_\mathcal{N}$ and the quotient $\frac{\bar{e}_\mathcal{N}}{\bar{e}_{\hat{\mathcal{Q}}}}$, which we denote by $\hat{a}$ and $\hat{r}$, respectively. In Step 9 of Algorithm 3, $\hat{a}$ is obtained by first rounding $\bar{q}_\mathcal{N}$ to the nearest integer and then applying the quantization function Q_{bin}, defined previously in this section. In Step 10, $\hat{r}$ is calculated

by a function $Q_R(\bar{e}_{\mathcal{N}}, \bar{e}_{\hat{\mathcal{Q}}}) : \mathbb{R} \times \mathbb{R} \to [0 : R-1]$, defined as follows (for ENANO we choose $R = 4$):

- If $\bar{e}_{\hat{\mathcal{Q}}} = 0$, set $\hat{r} = 0$. Notice that $\bar{e}_{\mathcal{N}}$ must be zero in this case.
- Else:
  - If $\frac{\bar{e}_{\mathcal{N}}}{\bar{e}_{\hat{\mathcal{Q}}}} < \frac{1}{2}$, set $\hat{r} = 0$.
  - If $\frac{1}{2} \leq \frac{\bar{e}_{\mathcal{N}}}{\bar{e}_{\hat{\mathcal{Q}}}} < 1$, set $\hat{r} = 1$.
  - If $1 \leq \frac{\bar{e}_{\mathcal{N}}}{\bar{e}_{\hat{\mathcal{Q}}}} < 2$, set $\hat{r} = 2$.
  - If $2 \leq \frac{\bar{e}_{\mathcal{N}}}{\bar{e}_{\hat{\mathcal{Q}}}}$, set $\hat{r} = 3$.

6.2.1.2 Memory optimization

As mentioned, nanopore FASTQ files use a much larger alphabet for quality scores, in comparison to traditional sequencing technologies. However, our experiments show that quality scores tend to concentrate at the lower end of the alphabet. For example, we observe that in two of our experimental datasets, *npd* and *hs1* (detailed in Section 6.3.1), 98% and 63% of the quality scores concentrate in the lowest 32 quality score values, of the possible 94, respectively. We exploit this observation by defining a fixed threshold, T_q, and encoding quality score values that are greater than or equal to T_q in a two-step fashion. Specifically, we encode a quality score q_i by encoding, in a first step, the value $\min\{q_i, T_q\}$ with the adaptive probability distribution estimated for context $\mathcal{C}_i$. The value T_q serves as an escape code, which indicates that $q_i \geq T_q$. In this case, we complete a full description of q_i by encoding the difference $q_i - T_q$ with a global adaptive probability distribution, independent of $\mathcal{C}_i$. In Figure 6.4 we show a diagram of the proposed method.

The global model estimates the probability distribution of high quality scores regardless of their context. This allows for a significant reduction in memory usage as we store, for each context, a probability distribution for values in the range $[0, T_q]$, and we use only one global probability distribution to model all the high quality score values. In ENANO, we set $T_q = 31$. This optimization did not have a negative effect on compression performance.

6.2.2 Parallelization of the compression algorithm

We recall from Section 6.2 that the compressor, when run in Fast (default) mode, operates in two phases: the *update phase*, where the statistics are adaptively updated for each context model, and the *fast coding phase*, where the statistics are fixed. For this purpose, in Fast mode, the encoding algorithm receives a parameter b specifying the number of blocks used to update the statistical models during the

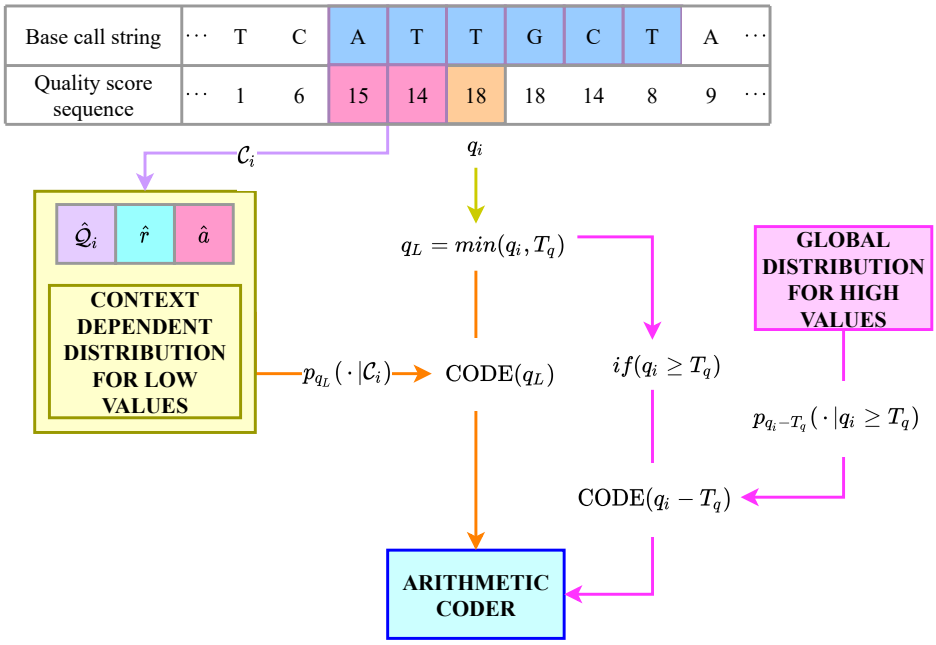


Figure 6.4: Compression scheme for low and high quality score values.

update phase (32 by default), before going into the fast coding phase. To accelerate the algorithm, we use the OpenMP[1] library to run several parallel *threads*, each processing blocks independently. The maximum number of threads to use is determined by the parameter t (8 by default). We proceed to describe the two coding phases in detail (we only describe the encoding algorithm as the decoding algorithm works symmetrically).

6.2.2.1 Update phase

As a first step, a single block of the file is encoded. The statistical model that results of compressing the first block is then replicated into B independent compressors, where B is a positive integer. The B compressors are then run in parallel with t threads, each one with a different block. Once every compressor is done, the encoded blocks are sequentially written to the compressed file in their original order, and the statistics of the compressors are merged into one individual *general model*, by averaging their probability distributions. The general model is then replicated to each compressor, and B new blocks are encoded. The process is repeated until b blocks are encoded.

A lower value of B allows for better compression ratio as the b blocks are encoded with more frequently updated models. However, the value of B determines the

[1] https://www.openmp.org/

number of blocks that can be coded in parallel during the update phase, as well as how many times the B statistical models are merged into a general model. In ENANO we use $B = 4$, as we found, based on experimental results, that it offers a good trade-off between compression performance and speed.

6.2.2.2 Fast coding phase

In the fast coding phase, the last general model calculated in the update phase is replicated into t independent compressors. In order to speed up the compression algorithm, the remaining part of the file is then compressed in batches of t blocks, without updating the probability distributions of the context models. The statistical values $\bar{q}_{\mathcal{N}}$, $\bar{e}_{\mathcal{N}}$, and $\bar{e}_{\hat{\mathcal{Q}}}$, used to determine the contexts for the quality score sequence, continue to be updated during the compression of each block. However, to avoid having dependency between the values of the statistics of different blocks, $\bar{q}_{\mathcal{N}}$, $\bar{e}_{\mathcal{N}}$, and $\bar{e}_{\hat{\mathcal{Q}}}$, are set to the last value they held in the last general model calculated, at the beginning of the encoding of each block.

Although ENANO in its current implementation does not support random access to specific reads, the fact that blocks are independently encoded during ENANO's fast coding phase sets the stage for including this feature in future versions of ENANO. For example, ENANO could store an index table that maps each block with the number of reads it contains and its compressed size. Then, to decode a specific read, it would suffice to decode the index table, the blocks encoded during the update phase, and the block that contains the read in question (if not contained in the blocks used in the update phase).

6.3 Experimental results of ENANO

In this section we report on a set of experiments performed on a collection of datasets of nanopore FASTQ files. The datasets are described in Section 6.3.1. In Section 6.3.2 we analyze the impact of the parameters ℓ, k, b, and t on the compression performance of ENANO. In Section 6.3.3 we evaluate the performance of ENANO by comparing it against the performance of various compression tools.

We measure the performance of a compressor on a dataset by its compression ratio, in bits per symbol, defined analogously to Section 3.5.1. Specifically, we compress each file of the dataset separately and calculate $CR = L/N$, where N is the total size in bytes of all the files in the dataset, and L is the total size in bytes of the compressed files. Recall that smaller values of CR correspond to better compression performance. We also expand the definition of CR to the different streams of the FASTQ file (*ids*, *base call strings*, and *quality scores*) by taking N as the total size in bytes of the specific stream over all files in the dataset, and L as

the total size in bytes of the compressed specific streams. To compare compression ratios, we use the percentage relative difference, as defined in Section 3.5.1. Finally, to compare overall performances between compressors we report *simple* and *weighted* averages of the results, the latter computed by weighting each result by the size of its corresponding dataset. The weighted average is highly influenced by larger datasets and, in particular, by the specific choice of datasets available for evaluation. This can be misleading when assessing the capability of the tools to process different types of datasets. Hence, in the rest of this section we report both averages, but we generally refer to the simple average when discussing and comparing the evaluated tools, unless otherwise specified.

All experiments were conducted in a server with 80 64 bit x86 Intel Xeon CPUs, 503.5GB of RAM memory, and CentOS Linux release 7.7.1908.

6.3.1 Datasets

We evaluate our algorithm on several publicly available datasets, described in Table 6.3.

The selected datasets cover a wide variety of organisms including human, animal, virus, plant, fungi and bacteria. The datasets also cover different possible compression scenarios, such as having 268 GB of data in a single file (*hss*), or having 113 GB of data divided in 336 different files (*npd*).

Name	Num. of files	Size (GB)	Tot. cov.	Description
sor	4	134	94x	Sorghum bicolor Tx430 [29]
bra	18	46	24x	Doubled haploid canola (Brassica napus L.) [72]
lun	13	15	*	Human lung bacterial metagenomic [20]
joi	9	5	*	Infected orthopaedic devices metagenomic [91]
vir	10	4	*	Direct RNA sequencing (HSV-1) [28]
hss	1	268	41x	Human GM12878 Utah/Ceph cell line [52]
hs2	50	194	30x	Human GM12878 Utah/Ceph cell line [12][1]
npd	336	113	*	Multiple organisms[2] [32]

Table 6.3: Nanopore sequencing datasets used for evaluation. The total coverage was estimated by dividing the size of the reads (only the nucleotide sequences) by the size of the genome of the corresponding organism. * indicates that the coverage could not be determined due to the metagenomic, or viral, nature of the sequenced organisms.

6.3.2 Impact of the configurable parameters on the performance of ENANO

There are four configurable parameters that may affect the performance of ENANO:

[1] We selected the first 50 files from the dataset.

[2] The sequenced samples correspond to viruses, bacteria, fungi, humans, animals, and metagenomic material.

- The order k of the Markov model used to compress the base call string (see page 6.2). In the software implementation of ENANO, this parameter can be set in run-time with the flag **-k**.
- The length of the neighbor base call string ℓ used to determine the quality score context (see Section 6.2.1). In the software implementation of ENANO, this parameter can be set in run-time with the flag **-l**.
- The number of blocks b used to update the statistics of the context models, in Fast mode (see Section 6.2.2). In the software implementation of ENANO, this parameter can be set in run-time with the flag **-b**.
- The maximum number of threads t used for compression and decompression. In the software implementation of ENANO, this parameter can be set in run-time with the flag **-t**.

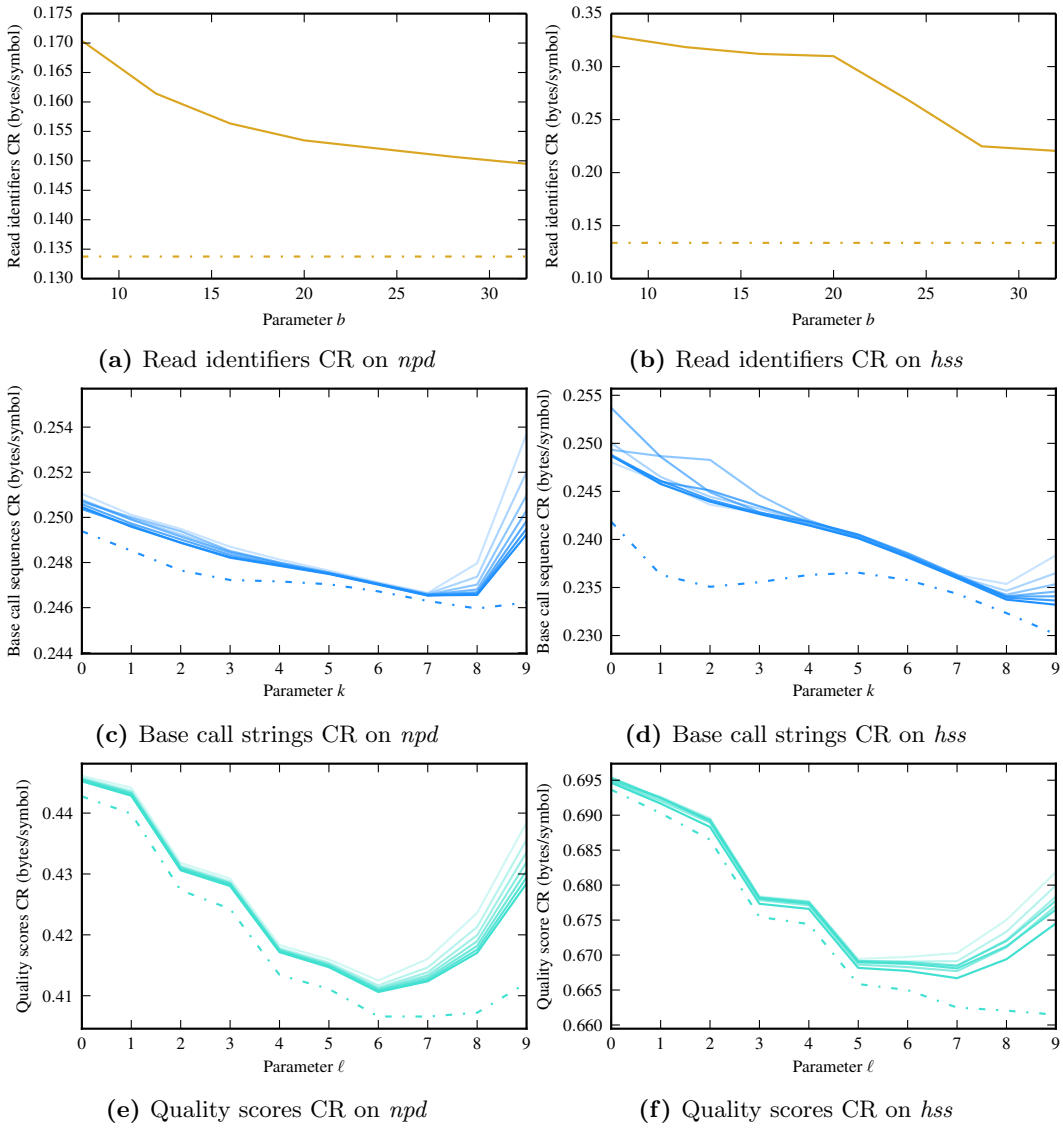
The choice of values for the parameters k, ℓ, and b, plays a key role in the compression performance of the algorithm. Parameter k determines the number of contexts in the base call string context model, while ℓ determines the number of neighborhoods in the quality score sequence context model. On the other hand, the value of b determines the number of blocks used to train the model in Fast mode. Hence, a higher value of b generates a more extensively trained model, which, in general, results in a better compression ratio, in exchange for higher encoding and decoding times.

To evaluate the trade-offs between different parameter values, we run ENANO with various parameter configurations on the *npd* and the *hss*[1] datasets, as they present two significantly different compression scenarios: *npd* has multiple files of different sizes, while *hss* is a single large file.

In Figure 6.5 we show how the compression ratios for different components of the FASTQ files vary with the values of the configurable parameters, on the *npd* dataset on the left column, and on the *hss* dataset on the right column. The dashed line shows the result for running ENANO in MC mode. For graphs 6.5c, 6.5d, 6.5e, and 6.5f, the variation of the parameter b is represented by lines with different color intensities. We report the results for $b \in \{8, 12, 16, 20, 24, 28, 32\}$, the lightest curve corresponding to $b = 8$ and the darkest to $b = 32$.

The first thing we observe from Figure 6.5 is that, as expected, the MC mode outperforms the Fast mode in every parameter configuration, as better trained models directly translate to better compression ratios. We also observe that in Fast mode the results tend to be better as the value of b increases. We choose 32 as the default value for b in ENANO.

[1] We use the first 4 million reads of the file (around 50GB of data) for testing time convenience.



(a) Read identifiers CR on *npd*

(b) Read identifiers CR on *hss*

(c) Base call strings CR on *npd*

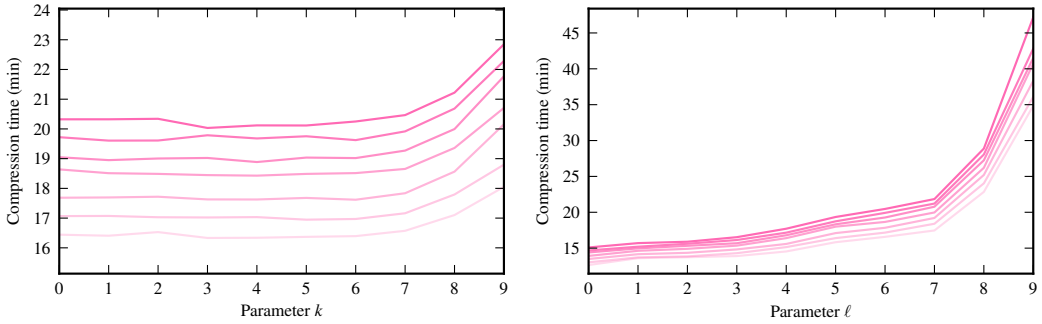
(d) Base call strings CR on *hss*

(e) Quality scores CR on *npd*

(f) Quality scores CR on *hss*

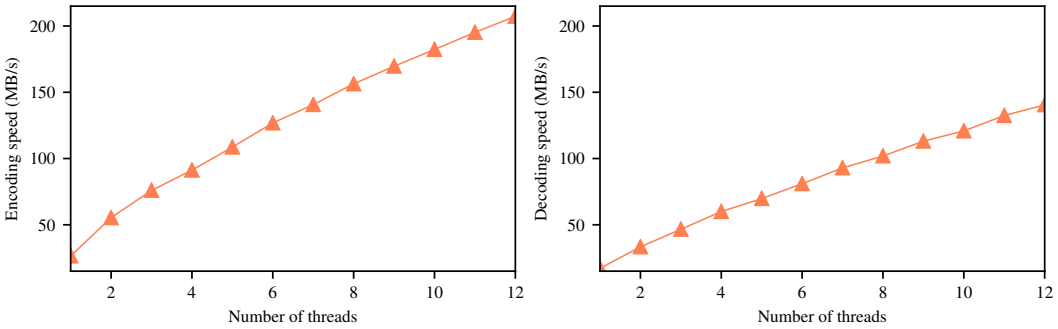
Figure 6.5: Compression ratio for read identifiers, base call strings, and quality scores, for various parameter configurations on the *npd* (left) and *hss* (right) datasets. The dashed lines represent the results obtained by running ENANO in MC mode. For graphs 6.5c – 6.5f, the color intensity of a continuous line represents the value of parameter b, ranging from $b = 8$ (lightest) to $b = 32$ (darkest).

In the plots 6.5c and 6.5d we notice that for $k = 7$ we obtain a compression ratio that is close to the minimum in all cases. Since, as we discuss soon, larger values of k require larger computational resources, we choose 7 as the default value for k in ENANO. The plots 6.5e and 6.5f show that for both *npd* and *hss* the value of parameter ℓ has a significant impact on the compression ratio of quality scores. For *npd*, we observe that the best compression ratio is achieved for $\ell = 6$ in both modes of ENANO. For *hss*, there is a slight compression ratio improvement for values of ℓ larger than 6 in MC, and almost no improvement in Fast. Again, accounting for



(a) Compression time for $\ell = 6$. **(b)** Compression time for $k = 7$.

Figure 6.6: Compression time of ENANO in Fast mode for the dataset *npd* with various parameter configurations. The color intensity of a line represents the value of parameter b, ranging from $b = 8$ (lightest) to $b = 32$ (darkest).



(a) Encoding speed on *hss* **(b)** Decoding speed on *hss*

Figure 6.7: Encoding and decoding speeds (MB/s) for ENANO's Fast mode algorithm with different number of threads on dataset *hss*.

computational resources considerations, we choose 6 as the default value for ℓ.

In Figure 6.6 we show the compression time obtained with ENANO for the dataset *npd* by fixing one of the parameters, ℓ or k, and letting the other vary. Specifically, in Figure 6.6a we fix $\ell = 6$, and in 6.6b we fix $k = 7$. In Figure 6.6a we observe that for values of k larger than 6 the compression time increases significantly. We notice a similar behaviour for values of ℓ larger than 5 in Figure 6.6b.

In Figure 6.7 we show the encoding and decoding speeds, in Megabytes per second, obtained with ENANO in Fast mode (with default parameters $\ell = 6$, $k = 7$ and $b = 32$), with different number of threads, for the dataset *hss*. Figures 6.7a and 6.7b show that, for a moderate number of threads, both encoding and decoding speeds consistently increase with the number of threads used. Clearly, the choice of number of threads depends on the available hardware. However, there will be a saturation of the speed as the non parallelized parts of the algorithm overtake the parallelized ones.

6.3.3 Comparative experimental results

We evaluate the two proposed modes of ENANO, by comparing their performance against state of the art FASTQ compression tools, on the nanopore datasets specified in Table 6.3. Most of the FASTQ compression tools such as DSRC2 [89], Fqzcomp [9], Fastqz [9], Slimfastq, FQC [35], LFQC [78], SCALCE [43], Quip [54], Leon [7], and KIC [109], are not specifically designed to compress long variable length reads and, as a consequence, they fail to execute successfully on any of the datasets we considered for evaluation. Similarly, although [1] reports on the compression performance of LFastqC for two nanopore FASTQ files, this compressor failed in most of the datasets that we tested. Therefore, we do not report results for these tools.

We compare our compressor to the state of the art FASTQ compression tool, SPRING[17], which has a built-in mode for compressing long reads. In the long read mode, SPRING separates and compresses the base call strings, quality score sequences, and read identifiers, in blocks of a fixed number of reads. The base call and quality score sequences are compressed using the BSC compressor[1], while the read identifiers are compressed using a specialized identifier compressor similar to the one proposed in [9]. For our experiments, we run SPRING in the long read mode, using the $-l$ flag, and we set the input buffer size to 1000 reads, as recommended by the authors of SPRING. We also compare ENANO with the general purpose compressor pigz,[2] a paralellized version of Gzip, for being widely used to compress FASTQ files. This general compressor separates the input file into blocks of 128 Kilobytes, and compresses each block in parallel, using a combination of LZ77 [110] and Huffman coding [47]. We run pigz in best compression mode (with flag *--best*). Each compressor is configured to run with 8 threads, except for ENANO in MC mode that runs on a single thread.

The compression ratios obtained by the evaluated tools on each dataset are shown in Table 6.4, as well as the quality scores compression ratio for SPRING and ENANO. The table also shows the percentage relative difference between the compression ratios obtained by SPRING and ENANO, using SPRING as the reference (thus, negative values indicate an advantage for ENANO). Finally, the last two rows of the table show the simple and weighted averages of the results.

The results show that both modes of ENANO outperform pigz and SPRING in all the datasets, both for total size and for quality scores. In particular, ENANO total compression ratio is on average 6.3% better than SPRING, for Fast mode, and 6.6% for MC mode. With respect to pigz, we observe that the gain of ENANO is on average 24.6% better in Fast mode and 24.9% better in MC mode. In terms of quality

[1] http://libbsc.com/
[2] https://zlib.net/pigz/

Dataset	Total				Quality scores		
	pigz	SPRING	ENANO Fast	ENANO MC	SPRING	ENANO Fast	ENANO MC
npd	0.442	0.349	0.327 (-6.1)	**0.325 (-6.7)**	0.452	0.410 (-9.2)	**0.406 (-10.1)**
sor	0.491	0.393	0.374 (-4.8)	**0.373 (-5.2)**	0.544	0.505 (-7.2)	**0.503 (-7.6)**
bra	0.480	0.378	0.356 (-5.8)	**0.356 (-6.0)**	0.521	**0.474 (-9.0)**	**0.474 (-9.0)**
lun	0.508	0.404	0.382 (-5.5)	**0.381 (-5.6)**	0.563	0.524 (-7.0)	**0.523 (-7.2)**
joi	0.469	0.368	0.329 (-10.7)	**0.327 (-11.1)**	0.532	0.485 (-8.8)	**0.485 (-8.8)**
vir	0.496	0.385	0.355 (-8.0)	**0.354 (-8.1)**	0.531	0.478 (-10.0)	**0.477 (-10.1)**
hss	0.538	0.472	0.452 (-4.4)	**0.449 (-5.0)**	0.709	0.670 (-5.5)	**0.669 (-5.8)**
hs2	0.513	0.416	0.397 (-4.6)	**0.397 (-4.7)**	0.596	0.560 (-6.2)	**0.559 (-6.3)**
S. average	0.492	0.396	0.371 (-6.3)	**0.370 (-6.6)**	0.556	0.513 (-7.9)	**0.512 (-8.1)**
W. average	0.504	0.418	0.398 (-4.9)	**0.396 (-5.3)**	0.598	0.558 (-6.8)	**0.556 (-7.1)**

Table 6.4: Compression ratio for all the compressors on all the dataset, and percentage relative difference with respect to SPRING (in parenthesis). The table also shows compression ratios of quality scores for ENANO's modes and SPRING, and the simple (S.) and weighted (W.) averages of the results. Best results for each dataset are bold-faced.

score compression, ENANO improves SPRING compression ratios by an average of 7.9% in Fast, and 8.1% in MC. When comparing the two modes of ENANO, we observe, as expected, that MC mode achieves the best compression ratio in all datasets, although the difference with the Fast mode is relatively small. Given the gain in running time obtained with the Fast mode (see Section 6.3.3.1), ENANO is run in Fast mode by default.

In Figure 6.8 we show a graphical representation of the percentage relative difference improvement of ENANO in Fast mode over SPRING, both for total and quality score compression ratios, over the various datasets.

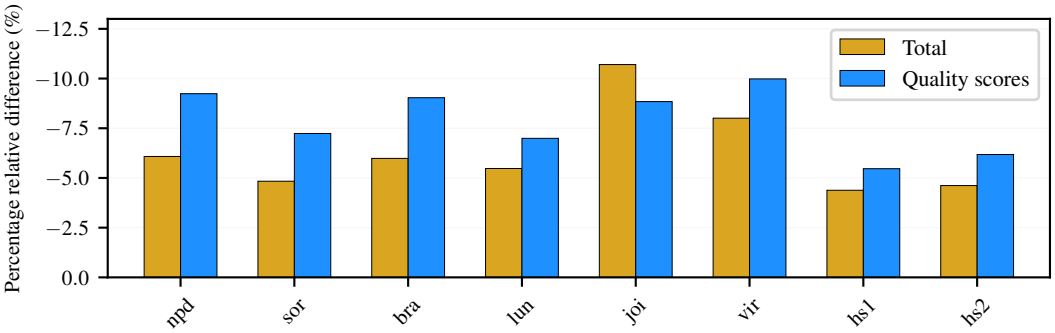


Figure 6.8: Compression ratio improvement of ENANO in Fast mode over SPRING in each dataset, expressed as a percentage relative difference.

For completeness of the results, in Table 6.5 we show the read identifiers and base call strings compression results for SPRING and for the two modes of ENANO. The table also shows the percentage relative difference between SPRING and ENANO modes.

With respect to read identifiers compression, the results in Table 6.5 show that both modes of ENANO outperform SPRING in most datasets, except for two, *sor* and *bra*. In these datasets ENANO in MC mode, and SPRING, achieve the same

Dataset	Read identifiers			Base call strings		
	SPRING	ENANO Fast	ENANO MC	SPRING	ENANO Fast	ENANO MC
npd	0.167	0.149 (-11.0)	**0.118 (-29.7)**	0.247	0.247 (-0.3)	**0.246 (-0.4)**
sor	**0.053**	0.198 (275.0)	**0.053 (0.0)**	**0.244**	0.245 (0.3)	0.245 (0.3)
bra	**0.059**	0.139 (133.3)	**0.059 (0.0)**	**0.237**	0.238 (0.6)	0.238 (0.6)
lun	0.484	0.301 (-37.8)	**0.294 (-39.2)**	0.243	**0.241 (-0.6)**	**0.241 (-0.6)**
joi	0.463	0.287 (-38.0)	**0.267 (-42.3)**	0.191	**0.177 (-7.0)**	**0.177 (-7.0)**
vir	0.468	0.270 (-42.3)	**0.252 (-46.2)**	**0.235**	0.236 (0.2)	0.236 (0.2)
hss	0.525	0.344 (-34.5)	**0.133 (-74.6)**	**0.234**	0.236 (0.8)	0.235 (0.1)
hs2	0.472	0.309 (-34.7)	**0.276 (-41.6)**	**0.236**	**0.236 (-0.1)**	**0.236 (-0.1)**
S. average	0.337	0.250 (26.3)	**0.182 (-34.2)**	0.233	**0.232 (-0.8)**	**0.232 (-0.8)**
W. average	0.351	0.268 (29.8)	**0.155 (-41.5)**	**0.238**	0.239 (0.2)	**0.238 (-0.0)**

Table 6.5: Comparison of compression ratios obtained for read identifiers and base call strings with SPRING and ENANO, on all the datasets, and percentage relative difference (in parenthesis). The table also shows the simple (S.) and weighted (W.) averages of the results. Best results for each part of the FASTQ file and each dataset are bold-faced.

compression ratio, while ENANO in Fast mode is worse. For base call strings, the compression performance of ENANO and SPRING is similar; the absolute values of the percentage relative differences are lower than 1% for most datasets, except for *joi*, where both ENANO modes are 7% better than SPRING.

For visualization of the results, in Figures 6.9 and 6.10, we show a graphical comparison of the compression ratios obtained with the evaluated compressors. Specifically, in Figure 6.9 we show, for each compressor, the average of the compression ratios obtained by that compressor on each dataset. In addition, for SPRING and ENANO these results are presented for each FASTQ file component separately.

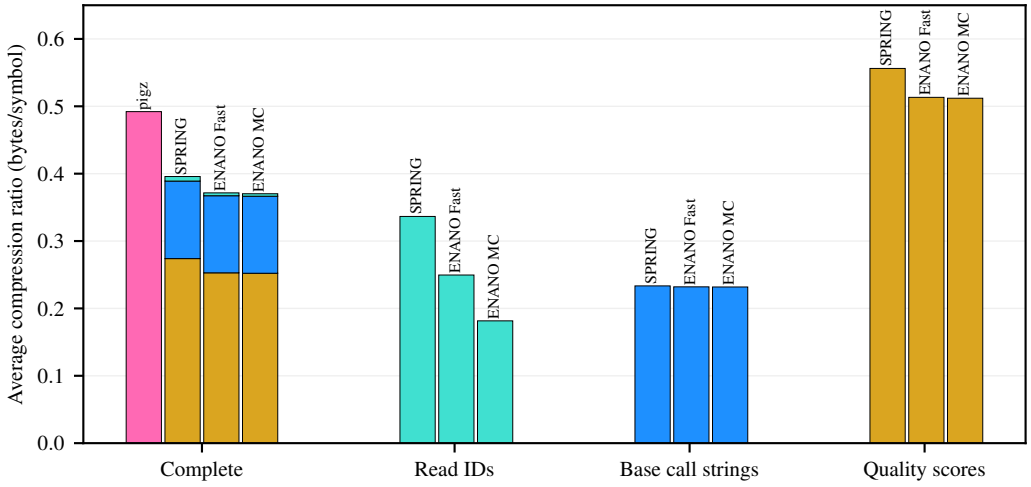


Figure 6.9: Average compression ratio obtained with the evaluated compressors. For SPRING and ENANO modes we also show the average compression ratios for each FASTQ file component separately. The *Complete* bars of SPRING and ENANO modes are subdivided in three parts, bottom up in following order: quality scores, base call strings, and Read IDs. The size of each subdivision is relative to the size the part occupies, on average, in the compressed datasets.

In Figure 6.10 we show a comparison of the compression ratios of the different components of the FASTQ file, between ENANO in Fast mode and SPRING, over each individual dataset.

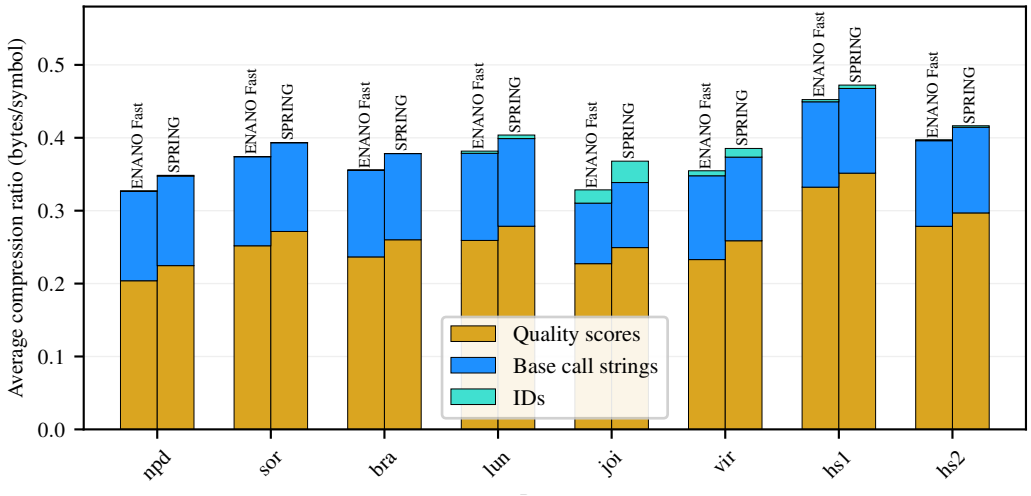


Figure 6.10: Comparison of the compression ratios of ENANO in Fast mode vs SPRING in each of the datasets, for each of the components of the FASTQ file.

6.3.3.1 Running time

In Table 6.6 we show the total encoding and decoding times for each compressor on each dataset. The results show that ENANO's Fast mode is the fastest encoder in all the datasets, except for *lun* where SPRING is faster. In terms of decoding, pigz is the fastest, followed by ENANO in Fast mode, except for *lun* where SPRING is faster than ENANO.

Dataset	pigz		SPRING		ENANO Fast		ENANO MC	
	enc	dec	enc	dec	enc	dec	enc	dec
npd	1:14:57	**0:09:49**	0:43:14	0:30:46	**0:20:52**	0:25:30	1:13:15	1:51:16
sor	1:40:51	**0:13:54**	1:08:25	0:36:59	**0:16:07**	0:19:21	1:40:53	2:03:16
bra	0:34:04	**0:04:19**	0:20:04	0:12:20	**0:06:58**	0:07:33	0:34:27	0:42:02
lun	0:09:31	**0:01:28**	**0:05:35**	0:03:39	0:05:59	0:05:43	0:12:14	0:15:27
joi	0:02:17	**0:00:25**	0:02:01	0:01:10	**0:00:43**	0:00:57	0:03:37	0:04:20
vir	0:02:27	**0:00:24**	0:01:50	0:01:08	**0:01:05**	0:01:07	0:03:35	0:04:24
hss	4:03:30	**0:23:57**	2:04:18	1:33:52	**0:36:49**	0:51:49	3:24:17	4:22:58
hs2	2:17:53	**0:18:41**	1:21:24	1:01:49	**0:33:02**	0:33:00	2:36:46	3:12:09

Table 6.6: Encoding and decoding times (in *h:mm:ss* format) for all the compressors on all the datasets. Best results, for each dataset, both for encoding and decoding, are bold-faced.

In Figure 6.11 we show, for each compressor, the total compression and decompression speeds, measured in MB per second, calculated as the accumulated size of all the datasets divided by the total time required to compress and decompress all files in all datasets, respectively.

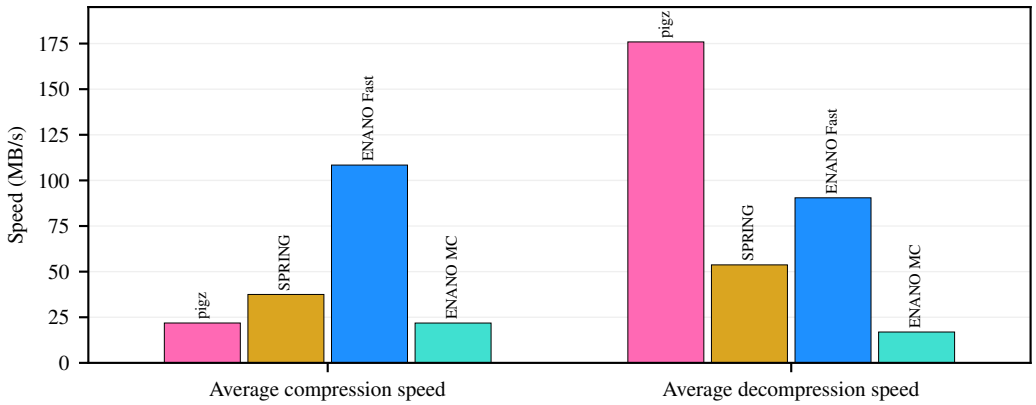


Figure 6.11: Total compression and decompression speeds over all the nanopore datasets for the different compressors.

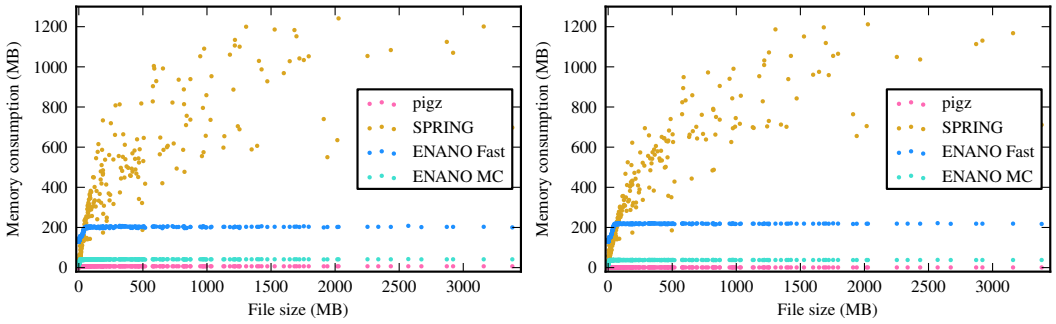
The compression speed of ENANO in Fast mode is 108 MB/s, which is 2.9x times faster than SPRING's 37 MB/s, and 5.0x times faster than pigz's 22MB/s. In terms of decoding speed, pigz is the fastest with 176 MB/s, while ENANO's 90 MB/s are 1.7x times faster than SPRING's 55 MB/s. In Fast mode, ENANO is 5.0x and 5.3x times faster than in MC mode in encoding and decoding, respectively.

6.3.3.2 Memory requirements

In Table 6.7 we show the maximum memory required by each compressor during the encoding and decoding processes on all the files of each dataset. The results show that ENANO, in any mode, generally requires less memory than SPRING, while pigz is the most efficient in terms of memory for each dataset. We also observe that ENANO's memory usage is much lower in MC mode than in Fast. This happens because ENANO Fast mode runs multiple compressors in parallel, each one with their own buffers and statistical models.

Dataset	pigz		SPRING		ENANO Fast		ENANO MC	
	enc	dec	enc	dec	enc	dec	enc	dec
npd	**7**	**1**	1222	1214	208	221	43	39
sor	**7**	**1**	3252	3215	208	222	43	39
bra	**7**	**1**	1166	1203	211	226	42	38
lun	**7**	**1**	363	411	213	224	41	38
joi	**7**	**1**	200	191	204	219	41	38
vir	**9**	**1**	254	259	205	219	42	38
hss	**7**	**1**	8720	8743	214	226	45	40
hs2	**9**	**1**	848	869	221	232	61	52
Maximum	**9**	**1**	8720	8743	221	232	61	52

Table 6.7: Maximum memory usage in MB registered during the encoding and decoding processes, for all the compressors, on all the files of each dataset. Lowest memory usage, for each dataset, both for encoding and decoding, are bold-faced.



(a) Encoder memory vs. file size.

(b) Decoder memory vs. file size.

Figure 6.12: Encoding and decoding maximum memory consumption versus file size, for pigz, SPRING, and ENANO on the *npd* dataset.

Notice that ENANO's maximum memory consumption, for both encoding and decoding, varies minimally between datasets. This occurs because ENANO's memory consumption does not depend on the size of the input file. In Figure 6.12 we show the encoding and decoding maximum memory consumption for pigz, SPRING, and ENANO versus the size of the processed file, for all the files in the *npd* dataset. The results show that, while pigz's and ENANO's memory consumption are almost constant, SPRING's memory consumption varies considerably with the sizes of the files, both in compression and decompression. As SPRING uses a fixed number of reads per block, and nanopore reads have long variable sizes, the final sizes of the blocks can vary widely between blocks, which can ultimately have great impact on the memory consumption. This is not the case for pigz and ENANO, as they use blocks of fixed size.

Chapter 7

RENANO: a REference-based compressor for NANOpore files

In Chapter 6 we presented ENANO, a lossless compression algorithm especially designed for nanopore FASTQ files, introducing a novel quality score sequence compression algorithm that exploits the specific characteristics of nanopore data. In this chapter we focus on further improving the compression of nanopore FASTQ files by addressing, specifically, the compression of the base call strings. In this note we introduce RENANO, a lossless nanopore FASTQ data compressor that builds on ENANO, introducing two novel reference-based compression algorithms for base call strings.

We build RENANO by replacing the base call strings compression scheme of ENANO with one of two novel reference-based compression algorithms introduced in Section 7.1, denoted by RENANO$_1$ and RENANO$_2$, while keeping the processing of the other streams intact. The two variants of RENANO thus constructed are complete FASTQ file compression schemes. However, since the rest of the chapter deals only with the compression of base call strings, we will slightly abuse terminology and use RENANO$_1$ and RENANO$_2$ also to refer specifically to the compression algorithms for base call strings. For RENANO$_1$, which we present in Section 7.1.2, we assume that a reference genome is available without cost to both the compressor and the decompressor. On the other hand, RENANO$_2$, presented in Section 7.1.3, does not require a reference genome on the decompression side.

The rest of the chapter is organized as follows. Section 7.1 introduces basic notation, terminology, and tools and, as mentioned, describes the two proposed reference-based base call string compression algorithms. Section 7.2 presents experimental results, and comparisons of RENANO to ENANO and to the scheme Genozip of [62].

7.1 Compression scheme of RENANO

In this section we present RENANO$_1$ and RENANO$_2$, two novel long-read base call string compression algorithms that use the alignment information (against a reference sequence) to improve compression. The general idea behind the proposed compression schemes is to encode a large portion of each base call string as a series of alignments to a reference genome, such that the alignments can be described more compactly than the raw base call string, thus yielding better compression. For both schemes, we assume that a reference sequence (e.g., a genome) in FASTA format is available on the compressor side. For RENANO$_1$, we consider the scenario in which the reference sequence is also available without cost to the decompressor, while for RENANO$_2$ we consider the scenario in which the reference sequence is *not* available to the decompressor, but a compacted version of that sequence is stored as part of the compressed output, thus incurring a code length cost. In both cases, we assume that the alignment information is obtained from an available alignment tool, such as Minimap2 [68], which generates a file in PAF format. Note that even though the current implementation of our compressor expects a PAF file as input, the algorithm could readily be adapted to other formats if they become popular in the future. Finally, both algorithms assume the base call strings are stored in the widely used FASTQ format [22].

Both of the presented compression algorithms take the alignment information generated by the alignment tool and transform it to an internal representation, which, together with the reference sequence, allows for perfect reconstruction of the original base call strings. The ability to encode the internal representation of alignments compactly is at the core of the schemes. In Section 7.1.1 we present the general notations and definitions needed to formalize the proposed internal alignment representation. Sections 7.1.2 and 7.1.3 describe RENANO$_1$ and RENANO$_2$, respectively, in detail. Finally, Section 7.1.4 provides specific details on how the PAF formatted output of Minimap2 [68] is transformed into our internal representation.

7.1.1 Notations and definitions

Let $\Sigma = \{A, C, G, T, N\}$ be the alphabet of base calls, where $\{A, C, G, T\}$ represent the DNA nucleotides, and N represents an undetermined base call. For a symbol $b \in \Sigma$, $\bar{b}$ is its complementary nucleotide, that is, $\bar{A} = T, \bar{C} = G, \bar{G} = C$, and $\bar{T} = A$, and for symbol N we define $\bar{N} = N$. We say that a string $s = s_1 s_2 ... s_n$, $s_i \in \Sigma$, is a *base call string* of length $|s| = n$, and define its *reverse complement* as $\bar{s} = \bar{s}_n \bar{s}_{n-1} ... \bar{s}_1$. We also use the notation $s[i : j] = s_i ... s_j$, to denote a substring of length $j - i + 1$, starting at position i and ending at position j; if $j < i$, we let $s[i : j]$ be the empty string, denoted by λ. For a base call string s and $d \in \{0, 1\}$, we define the *strand*

function $\pi(s, d)$ as $\pi(s, d) = s$ if $d = 0$, and $\pi(s, d) = \bar{s}$ if $d = 1$. We refer to d as the *strand direction indicator* of $\pi(s, d)$.

We define an *encoding transformation*, ϕ, as a sequence of $K + 1$ *transformation steps* that converts a base call string, s, into another base call string, $\phi(s)$, where K is a positive integer. The first K transformation steps construct a string s', starting from $s' = \lambda$, while scanning two input strings: s, and an additional (given) string $\mathcal{I}$. Each such transformation step is represented by a triplet of non-negative integers, (I, S, M), that can be interpreted as driving three elementary string operations on s', in order:

1. append a copy of the next I symbols from string $\mathcal{I}$ to s';
2. skip the following S symbols of s;
3. append a copy of the next M symbols from s to s'.

The $(K + 1)$-th, and final, step of the encoding transformation consists of applying the strand function to the constructed string s', obtaining the result $\phi(s) = \pi(s', d)$, given a strand direction indicator value d. More specifically, we let $\phi = \Big(\{(I_k, S_k, M_k)\}_{1 \le k \le K}, \mathcal{I}, d\Big)$, where each triplet (I_k, S_k, M_k) is comprised of an *insertion* length I_k, a *skip* length S_k, and a *match* length M_k; $\mathcal{I}$ is the *insertions base call string* of ϕ, with length $|\mathcal{I}| = \sum_{k=1}^{K} I_k$; and d is the strand direction indicator. We always apply $\phi(s)$ to strings s of length equal to $\sum_{k=1}^{K} \Big(S_k + M_k\Big)$. Algorithm 4 summarizes the encoding transformation process. An example of the application of Algorithm 4 is shown in Figure 7.1.

Algorithm 4: Convert string s into string $\phi(s)$.

Input : String s, and $\phi = \Big(\{(I_k, S_k, M_k)\}_{1 \le k \le K}, \mathcal{I}, d\Big)$
Output: String $\phi(s)$

```
Let s' = λ
Let i = j = 1
for k = 1 to k = K do
    Get (I_k, S_k, M_k) from φ
    Append I[i : i + I_k − 1] to s'
    i = i + I_k
    j = j + S_k
    Append s[j : j + M_k − 1] to s'
    j = j + M_k
end
return π(s', d)
```

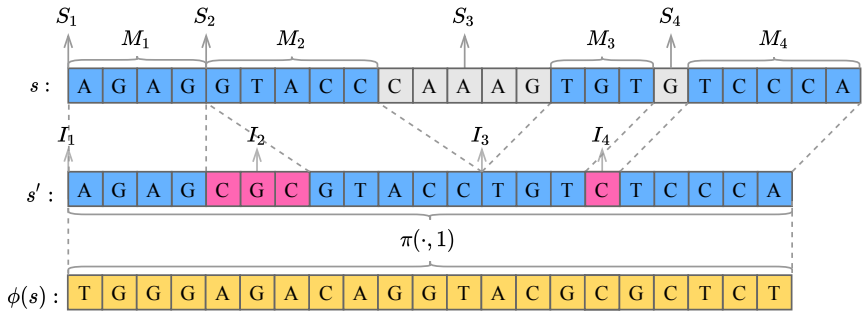


Figure 7.1: Constructing $\phi(s)$ by executing Algorithm 4 with inputs: s, and encoding transformation $\phi = \Big(((0,0,4),(3,0,5),(0,5,3),(1,1,5)), \mathcal{I} = \text{CGCC}, d = 1\Big)$. Notice that when an insertion length or a match length is 0, no symbols are appended to s' from $\mathcal{I}$ or s, respectively. Also, if the strand direction indicator is $d = 0$, then $\phi(s) = s'$.

A reference genome is usually composed of multiple base call strings; for example, in the case of the human genome there is at least one base call string for each chromosome. Therefore, we represent a reference genome as an ordered set, $\mathcal{R} = \{r_k\}_{1 \le k \le |\mathcal{R}|}$, of base call strings. Now, let q be a read base call string, and let $r \in \mathcal{R}$ be a reference base call string. We define an *atomic alignment* between q and r, denoted $\alpha(q, r)$, as an encoding transformation from a substring of r to a substring of q. More specifically, we let $\alpha(q, r) = (i, j, i', j', \phi)$, where ϕ is an encoding transformation such that $\phi(r[i' : j']) = q[i : j]$. Finally, we define a *full alignment* between q and $\mathcal{R}$, as a sequence of atomic alignments $A(q, \mathcal{R}) = \{\alpha_t(q, r_{k_t})\}_{1 \le t \le L_A}$, where k_t is the index of a reference string in $\mathcal{R}$, and L_A is the length of (i.e., number of atomic alignments in) $A(q, \mathcal{R})$. We consistently use the same subscript t of an atomic alignment $\alpha_t(q, r_{k_t})$ to denote all of its components, i.e., $\alpha_t(q, r_{k_t}) = (i_t, j_t, i'_t, j'_t, \phi_t)$. In Figure 7.2 we show an example of a full alignment.

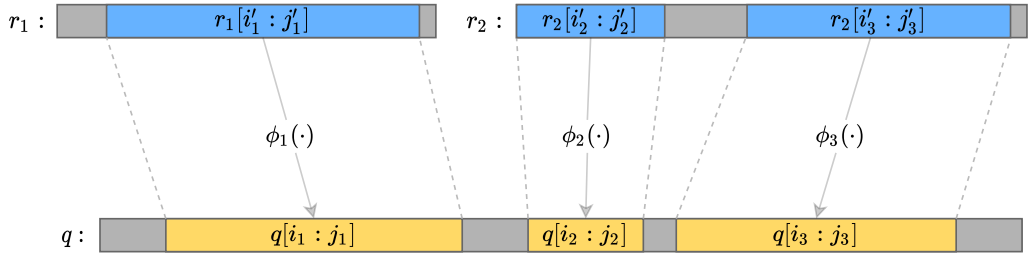


Figure 7.2: Example of a full alignment $A(q, \mathcal{R}) = \{\alpha_1(q, r_1), \alpha_2(q, r_2), \alpha_3(q, r_2)\}$, between a base call string q and a set of reference base call strings $\mathcal{R} = \{r_1, r_2\}$.

We refer to a full alignment $A(q, \mathcal{R})$ as *non-overlapping*, if none of the aligned substrings of q overlap (such as the one shown in Figure 7.2). Notice that the atomic alignments in the example do not cover the whole read q, a situation that

may also occur in the general case. We refer to segments of q not covered by $A(q, \mathcal{R})$ as *unaligned*. Later on, we will discuss how these segments are dealt with in our algorithms.

7.1.2 RENANO$_1$: A reference-dependent compression and decompression scheme

In this section we present RENANO$_1$, an algorithm that compresses the base call strings of a FASTQ file assuming that the set of reference strings $\mathcal{R}$ is available to both the compressor and the decompressor. We further assume that the full alignment $A(q, \mathcal{R})$ for each base call string q against the set of reference strings $\mathcal{R}$ is available during the compression stage. In practice, this means that we assume an alignment tool has been run on the base call strings in the file against the set of reference strings $\mathcal{R}$, and that $A(q, \mathcal{R})$ has been derived from the alignment information obtained (details on how this is done are provided in Section 7.1.4). For convenience, we also assume the atomic alignments in $A(q, \mathcal{R}) = \{\alpha_t(q, r_{k_t})\}_{1 \leq t \leq L_A}$ to be non-overlapping (as exemplified in Figure 7.2). Again in Section 7.1.4, we show that any full alignment can be transformed to satisfy this assumption.

As discussed in Section 7, RENANO builds over its predecessor ENANO [33], which parses and compresses the FASTQ file in blocks that always contain a whole number of reads. For consistency, our algorithm compresses the sequence of read base call strings contained in a block of the FASTQ file, denoted by $Q = q_1, \cdots, q_{|Q|}$, independently of the base call strings contained in other blocks. This makes the compression scheme compatible with the parallelization strategy implemented in ENANO.

Recall from Section 7.1.1 that, given an atomic alignment $\alpha_t(q, r) = (i, j, i', j', \phi)$, we can reconstruct the substring $q[i : j] = \phi(r[i', j'])$ by executing Algorithm 4 with inputs ϕ and $r[i', j']$. As we have access to the reference string r during decompression, encoding the atomic alignment $\alpha_t(q, r)$ suffices to describe $q[i : j]$. Extending this idea, RENANO$_1$ compresses the aligned substrings of a read base call string q by encoding the full alignment $A(q, \mathcal{R})$, while the unaligned parts of q are encoded as raw base call strings, as in ENANO. We now proceed to describe the compression scheme in detail.

7.1.2.1 The encoding algorithm

To encode Q, we split the various elements of the representations of its read base call strings q_i, i.e, the aligned substrings represented in $A(q_i, \mathcal{R})$ and the unaligned substrings, into separate sequences, which we call *streams*. We refer to the stream values that comprise a representation of any of these elements as its *stream rep-*

resentation. To help improve compression performance, the streams are designed so that each stream gathers parts of the representations that we expect to be correlated. The streams, in turn, are encoded independently. Different streams may contain different types of data. In particular, as described in more detail below, we will have streams comprised of raw base call symbols, streams comprised of binary symbols, and streams comprised of non-negative integers. For each stream S of the latter, we define a parameter η_S that determines the bit size representation of the integers, where η_S is, for convenience, assumed to be a positive multiple of 8. The values of these parameters used in our experiments are discussed later in Section 7.2.3. Specifically, we define the following streams:

- $\mathcal{B}$: base call strings stream used to encode individual raw base call symbols (A, C, G, T, N), which include: the unaligned base call strings, and the insertion base call strings $\mathcal{I}$ that are part of the encoding transformations.
- $\mathcal{L}$: base call string lengths stream used to store the non-negative integer $|q|$, i.e., the length of each read basecall string q, using an $\eta_{\mathcal{L}}$-bit representation.
- $\mathcal{Q}$: full alignment sizes stream used to store the non-negative integers L_A, using an $\eta_{\mathcal{Q}}$-bit representation.
- $\mathcal{S}$: starting position increments stream for aligned substrings of q, i.e., the non-negative integers $i_t - i_{t-1}$, with the convention $i_0 = 0$, using an $\eta_{\mathcal{S}}$-bit representation.
- $\mathcal{E}$: lengths stream for aligned substrings of q, i.e., the non-negative integers $z_t = j_t - i_t + 1$, using an $\eta_{\mathcal{E}}$-bit representation.
- $\mathcal{U}$: aligned reference string identity indexes stream used to store the non-negative integers r_{k_t}, using an $\eta_{\mathcal{U}}$ -bit representation.
- $\mathcal{S}'$: aligned reference substring starting positions stream, which stores the non-negative integers i'_t, using an $\eta_{\mathcal{S}'}$-bit representation.
- $\mathcal{D}$: strand direction indicator stream, which stores the binary values of $d_t \in \{0, 1\}$.
- $\mathcal{N}$: insertion lengths stream that stores the non-negative integers I_k, using an $\eta_{\mathcal{N}}$-bit representation.
- $\mathcal{K}$: skip lengths stream, which stores the non-negative integers S_k, using an $\eta_{\mathcal{K}}$-bit representation.
- $\mathcal{M}$: match lengths stream used to store the non-negative integers M_k, using an $\eta_{\mathcal{M}}$-bit representation.

For convenience, we define $\mathbb{S} = \{\mathcal{B}, \mathcal{L}, \mathcal{Q}, \mathcal{S}, \mathcal{E}, \mathcal{U}, \mathcal{S}', \mathcal{D}, \mathcal{N}, \mathcal{K}, \mathcal{M}\}$ as the set of all the proposed streams.

Algorithm 5: Encode a sequence of read base call strings.

Input : Sequence of read base call strings $Q = q_1, \cdots, q_{|Q|}$, full alignment $A(q_i, \mathcal{R})$ of each q_i in Q

Output: An encoding of Q is output to a bit stream F

1 Let $\mathbb{S} = \{\mathcal{B}, \mathcal{L}, \mathcal{Q}, \mathcal{S}, \mathcal{E}, \mathcal{U}, \mathcal{S}', \mathcal{D}, \mathcal{N}, \mathcal{K}, \mathcal{M}\}$ be a set of empty streams
2 Output a η_F-bit representation of the non-negative integer $|Q|$ to F
3 **for** $i = 1$ *to* $i = |Q|$ **do**
4 Incorporate a stream representation of q_i into $\mathbb{S}$ using Algorithm 6 (see Section 1.2.1.1) with q_i and $A(q_i, \mathcal{R})$ as inputs
5 **end**
6 **foreach** *stream* $S \in \mathbb{S}$ **do**
7 Generate an encoding X of S as explained in Section 1.2.1.2
8 Let n_X be the length of X, in bytes. Output a η_F-bit representation of n_X to F
9 Output X to F
10 **end**

In Algorithm 5 we present the general compression scheme for a sequence of base call strings $Q = q_1, \cdots, q_{|Q|}$. The algorithm receives Q as input together with a full alignment $A(q_i, \mathcal{R})$ for each q_i in Q, and generates an encoding of Q that is output to a bit stream F. All the streams in $\mathbb{S}$ are initialized as empty sequences in Step 1. In Step 2, the algorithm outputs the number of read base call strings in Q, so that the decoder can retrieve how many read base call strings are encoded into the streams. In Step 3, the algorithm loops over each base call string q_i, splitting a representation of q_i into the various streams in $\mathbb{S}$ by running Algorithm 6, which we discuss in Subsection 1.2.1.1. Then, the loop in Step 6 encodes the content of each stream S in $\mathbb{S}$. To this end, in Step 7, S is encoded using compression techniques that we present later in Subsection 1.2.1.2. Next, the encoding of the stream is output to F preceded by its length (in bytes). This ensures that the decoder can access each stream separately. Both numbers output in steps 2 and 8 are encoded as η_F-bit integers, where η_F is an implementation parameter discussed in Section 7.2.3. We now proceed to describe steps 4 and 7 of the algorithm in detail.

1.2.1.1 Splitting the representation of each base call string into separate streams (Step 4 of Algorithm 5).

To generate a stream representation of an individual read base call string q, we

use Algorithm 6.

Algorithm 6: Describe a full alignment $A(q, \mathcal{R})$ and the needed parts of base call string q using the streams in $\mathbb{S}$.

Input : Read base call string q, full alignment $A(q, \mathcal{R})$
Output: Updated streams in $\mathbb{S}$

1 Append the string length $|q|$ to stream $\mathcal{L}$
2 Append the number of atomic alignments L_A to stream $\mathcal{Q}$
3 Sort the atomic alignments in $A(q, \mathcal{R})$ in increasing order of i_t
4 Let $i = 1$
5 **for** $t = 1$ *to* $t = L_A$ **do**
6 **if** $i < i_t$ **then**
7 Append raw base call string $q[i : i_t - 1]$ to stream $\mathcal{B}$
8 **end**
9 $i = i_t$
10 Append reference string index k_t to stream $\mathcal{U}$
11 Append a stream representation of $\alpha_t(q, r_{k_t})$ to the streams in $\mathbb{S}$ using Algorithm 7 (see below)
12 $i = j_t + 1$
13 **end**
14 **if** $i \leq |q|$ **then**
15 Append base call string $q[i : |q|]$ to stream $\mathcal{B}$
16 **end**

The algorithm starts by appending the length of q to stream $\mathcal{L}$ (Step 1), and the number of atomic alignments in $A(q, \mathcal{R})$ to stream $\mathcal{Q}$ (Step 2). A representation of q is incorporated into the streams progressively, scanning q from start to end as the algorithm iterates over the atomic alignments in $A(q, \mathcal{R})$, which are previously sorted in increasing order of i_t (Step 3). The variable i, initialized to $i = 1$ in Step 4, maintains the starting position of the portion of q that remains to be incorporated into the stream representation. In Step 5, the algorithm loops over each atomic alignment in $A(q, \mathcal{R})$. Each iteration starts by checking, in Step 6, if the current value of i is smaller than the starting position of the current atomic alignment, i_t. If so, then $q[i : i_t - 1]$ is an unaligned substring of q, which is directly appended to stream $\mathcal{B}$ in Step 7. The index k_t of the reference string of the current atomic alignment, $\alpha_t(q, r_{k_t})$, is appended to the stream $\mathcal{U}$ in Step 10, followed by an execution of Algorithm 7, which appends a representation of $\alpha_t(q, r_{k_t})$ itself to the streams in $\mathbb{S}$. The iteration ends by updating the value of i to the next position yet to be represented, $i = j_t + 1$. By the end of the loop in Step 5, every symbol in q has been represented as part of either a raw base call string in Step 7 or an atomic alignment in Step 11, with the possible exception of the trailing symbols in an unaligned suffix

of q, which are incorporated in Step 15.

Notice that the starting and ending positions of the unaligned substrings are not explicitly described, as they are fully determined by the starting and ending positions of the atomic alignments, together with the total length of q.

To generate a stream representation of each atomic alignment $\alpha_t(q, r_{k_t})$, we use Algorithms 7 and 8.

Algorithm 7: Generate a stream representation of an atomic alignment.

Input : Full alignment $A(q, \mathcal{R})$ and index $t \geq 1$ of an atomic alignment $\alpha_t(q, r_{k_t})$ in $A(q, \mathcal{R})$

Output: A representation of $\alpha_t(q, r_{k_t})$ is appended to the streams in $\mathbb{S}$

1 Let $\delta = i_t - i_{t-1}$ `/* recall that` $i_0 = 0$ `*/`
2 Append δ to stream $\mathcal{S}$
3 Append $z = j_t - i_t + 1$ to stream $\mathcal{E}$
4 Append the reference starting index i'_t to stream $\mathcal{S}'$
5 Append the insertions base call string $\mathcal{I}$ of ϕ_t to stream $\mathcal{B}$
6 Append the strand direction indicator d of ϕ_t to stream $\mathcal{D}$
7 **foreach** $(I_k, S_k, M_k) \in \phi_t$ **do**
8 | Run Algorithm 8 (see below) with triplet (I_k, S_k, M_k) as input
9 **end**

Algorithm 7 starts by appending a differential representation of the starting position of the aligned substring, i_t, to stream $\mathcal{S}$ (Step 2). With this representation, the positions i_t are implicitly determined by the successive differences, $i_t - i_{t-1}$, which are stored in $\mathcal{S}$ (recall that by convention, $i_0 = 0$, and hence the difference is well defined for all $t \geq 1$). Since $A(q, R)$ is sorted in increasing order of i_t, these differences are non-negative and, in general, with a high frequency of relatively low values, which we exploit to achieve efficient compression. In Step 3, the length of the aligned substring, $z = j_t - i_t + 1$, is appended to stream $\mathcal{E}$. We choose to store the length of the substring instead of its ending position, as this also generally results in a high frequency of relatively low values. Step 4 appends the starting position of the reference substring, i'_t, to stream $\mathcal{S}'$, and the following steps generate a stream representation of the encoding transformation ϕ_t. In Steps 5 and 6, the insertions base call string $\mathcal{I}$ and the strand direction indicator d are appended to streams $\mathcal{B}$ and $\mathcal{D}$, respectively. We do not store the length of the insertions base call string $\mathcal{I}$, as it can be calculated as the sum of the insertion lengths, $|\mathcal{I}| = \sum_{k=1}^{K} I_k$. Finally, in Step 7, the algorithm loops through every triplet of the encoding transformation, and incorporates its stream representation into $\mathbb{S}$ using Algorithm 8.

We observed, empirically, that most of the values that typically compose triplets (I, S, M) lie within a narrow range of small non-negative integers, with, possibly, a

Algorithm 8: Recursively generate a stream representation of a triplet (I, S, M), with constrained-length operations.

Input : Triplet (I, S, M)
Output: A representation of (I, S, M) is appended to the streams in $\mathbb{S}$

```
1  if I > T_N then
2  |  Run Algorithm 8 with triplet (T_N, 0, 0) as input
3  |  Run Algorithm 8 with triplet (I − T_N, S, M) as input
4  else if S > T_K then
5  |  Run Algorithm 8 with triplet (0, T_K, 0) as input
6  |  Run Algorithm 8 with triplet (I, S − T_K, M) as input
7  else if M > T_M then
8  |  Run Algorithm 8 with triplet (I, S, T_M) as input
9  |  Run Algorithm 8 with triplet (0, 0, M − T_M) as input
10 else
11 |  Append I, S, and M to streams N, K, and M, respectively
12 end
```

few outliers. For this reason, it is convenient to choose small values for the bit size parameters $\eta_{\mathcal{N}}$, $\eta_{\mathcal{K}}$, and $\eta_{\mathcal{M}}$, used to represent I, S, and M in the streams $\mathcal{N}$, $\mathcal{K}$, and $\mathcal{M}$, respectively. These parameters, in turn, determine thresholds, $T_{\mathcal{N}} = 2^{\eta_{\mathcal{N}}} - 1$ for insertion lengths, $T_{\mathcal{K}} = 2^{\eta_{\mathcal{K}}} - 1$ for skip lengths, and $T_{\mathcal{M}} = 2^{\eta_{\mathcal{M}}} - 1$ for match lengths. Since a stream S, for $S \in \{\mathcal{N}, \mathcal{K}, \mathcal{M}\}$, does not admit a single-value representation of an operation length larger than the threshold T_S, every triplet (I, S, M) with such an operation length is divided by Algorithm 8 into a sequence of triplets, with trimmed operation lengths that do satisfy these constraints. This substitute sequence represents a combination of string operations that produce the same result as the original operations represented by (I, S, M).

Algorithm 8 proceeds recursively. If an operation length, say I, is larger than the corresponding threshold, $T_{\mathcal{N}}$, then the algorithm is recursively invoked for two triplets. One of them represents a single insertion of maximal length, $T_{\mathcal{N}}$. The other represents the remaining operations, in this case an insertion of length $I - T_{\mathcal{N}}$ together with the original skip and match operations, of lengths S and M, respectively. Skip and match lengths exceeding the thresholds $T_{\mathcal{K}}$ and $T_{\mathcal{M}}$, respectively, are handled analogously. Since an invocation of the algorithm for triplets (I, S, M) having at least one value exceeding the corresponding threshold produces recursive calls for triplets (I', S', M'), where at least one of I', S', M' is strictly smaller than I, S, M, respectively (and none of them is larger), then eventually no operation length exceeds its threshold, and the algorithm generates a stream representation of (I, S, M) directly in Step 11, with no further recursive calls.

As a last comment regarding Algorithm 7, notice that j'_t is not explicitly represented in any stream. Nevertheless, the length, z', of the reference substring is implicitly determined by the skip and match lengths of the encoding transformation,

$$z' = \sum_{k=1}^{K} \left(S_k + M_k \right),$$

where K is the number of transformation steps. The value of j'_t, in turn, can be calculated as $j'_t = i'_t + z' - 1$. We also notice that the number of transformation steps, K, is not explicitly represented either; we show later in Section 1.2.2.1 that it can be reconstructed by the decoder from the streams in $\mathbb{S}$.

1.2.1.2 Encoding the streams (Step 7 of Algorithm 5).

The algorithms described so far construct a set $\mathbb{S}$ of streams of various data types, from which, given the reference strings $\mathcal{R}$, the base call strings in the original FASTQ file can be fully reconstructed. To complete a compression algorithm, we must encode these streams, as efficiently as possible, into bitstreams.

To encode each stream we combine context modeling with arithmetic encoding, as explained in Section 6.1. Specifically, to encode a base call symbol from stream $\mathcal{B}$, we use the previous k base call symbols as the context. For the strand direction indicator stream, $\mathcal{D}$, we directly encode each binary value $d_t \in \{0, 1\}$ as a single bit, as experiments show that the strand direction indicators are close to uniformly distributed. The remaining streams are comprised of sequences of non-negative integers of various representation sizes. We use the same method to encode each such stream. To encode an η-bit integer stream, where η is a multiple of 8, we split the bit representation of each integer of the stream into $\frac{\eta}{8}$ bytes. We encode each byte separately, starting from the least significant to the most significant byte, using a specific context for each of the $\frac{\eta}{8}$ byte positions. In other words, all the least significant bytes of the integers in the stream are encoded in the same context, all the second least significant bytes are encoded in the same separate context, and so forth. Since, as mentioned, these non-negative integer sequences tend to concentrate towards the lower end of their range, the higher order bytes end up being highly compressible, and there is little penalty in over-estimating the selected integer size η for each stream.

7.1.2.2 The decoding algorithm

In Algorithm 9 we present the general decompression scheme that decodes the sequence of base call strings $Q = q_1, \cdots, q_{|Q|}$ given the set of reference base call strings $\mathcal{R}$ and the compressed file, accessed through an input bitstream, F. Notice that,

as the encoding and decoding algorithms work in lockstep, at any stage of the decoding process, the decoder knows exactly what stream to read from and how much information it needs from it.

Algorithm 9: Decode the sequence of read base call strings in a block of the FASTQ file.

Input : Set of reference base call strings $\mathcal{R}$, input stream F
Output: Sequence of read base call strings $Q = q_1, \cdots, q_{|Q|}$

1 Let Q be an empty sequence
2 Let $\mathbb{S} = \{\mathcal{B}, \mathcal{L}, \mathcal{Q}, \mathcal{S}, \mathcal{E}, \mathcal{U}, \mathcal{S}', \mathcal{D}, \mathcal{N}, \mathcal{K}, \mathcal{M}\}$ be a set of empty streams.
3 Retrieve the number of encoded read base call strings from F into n_Q
4 **foreach** *stream* $S \in \mathbb{S}$ **do**
5 Retrieve the length in bytes, n_X, of the encoding of S from F
6 Retrieve n_X bytes from F into X
7 Decode S from X
8 **end**
9 **for** $i = 1$ *to* $i = n_Q$ **do**
10 Reconstruct q_i using Algorithm 10 (see Section 1.2.2.1) with $\mathbb{S}$ and $\mathcal{R}$ as inputs
11 Append q_i to Q
12 **end**
13 **return** Q

Algorithm 9 reverses the steps of Algorithm 5. It starts by initializing Q as an empty sequence in Step 1 and all the streams in $\mathbb{S}$ as empty streams in Step 2. The contents of both Q and $\mathbb{S}$ are constructed in subsequent steps. First, the number of base call strings to be decoded and added to Q, n_Q, is obtained from F in Step 3. Next, each stream in $\mathbb{S}$ is decoded individually. This is done by retrieving the length, in bytes, of the encoding of the stream in Step 5, followed by the encoding itself in Step 6, which is decoded in Step 7. Once the streams in $\mathbb{S}$ are decoded in Step 10, the algorithm reconstructs each of the read base call strings of Q from its stream representation by running Algorithm 10 with $\mathbb{S}$ and $\mathcal{R}$ as inputs. Next, we describe Step 10 in more detail.

1.2.2.1 Reconstructing read base call strings (Step 10 of Algorithm 9).

Algorithm 10 reconstructs a base call string q from its stream representation in $\mathbb{S}$ and the set of reference strings $\mathcal{R}$.

Algorithm 10 reverses the steps of Algorithm 6. In the algorithm, q is initialized to an empty string in Step 1, and it is reconstructed progressively, appending symbols

Algorithm 10: Reconstruct base call string q.

Input : Reference strings $\mathcal{R}$, set of streams $\mathbb{S}$
Output: Read base call string q

1 Let $q = \lambda$ and $i = 1$
2 Retrieve the number of symbols to be added to q from $\mathcal{L}$ into n_q
3 Retrieve the number of atomic alignments L_A from $\mathcal{Q}$
4 **for** $t = 1$ *to* $t = L_A$ **do**
5 Retrieve reference string index k_t from $\mathcal{U}$
6 Retrieve i_t from $\mathcal{S}$
7 **if** $i < i_t$ **then**
8 Extend q by retrieving $q[i : i_t - 1]$ from $\mathcal{B}$
9 **end**
10 Reconstruct $\alpha_t(q, r_{k_t}) = (i_t, j_t, i'_t, j'_t, \phi_t)$ by retrieving j_t, i'_t, j'_t, and ϕ_t using Algorithm 11 (see below) with i_t and $\mathbb{S}$ as inputs
11 Extend q by executing Algorithm 4 to obtain $q[i_t : j_t]$ from ϕ_t, and $r_{k_t}[i'_t : j'_t]$
12 Let $i = j_t + 1$
13 **end**
14 **if** $i \leq n_q$ **then**
15 Extend q by retrieving $q[i : n_q]$ from $\mathcal{B}$
16 **end**
17 **return** q

up to length n_q (obtained in Step 2). The variable i, initialized in Step 1, maintains the position of the next symbol to be added to q. In Step 3, the algorithm retrieves the number of atomic alignments L_A that encode substrings of q, and loops over each atomic alignment in Step 4. Each atomic alignment $\alpha_t(q, r_{k_t}) = (i_t, j_t, i'_t, j'_t, \phi_t)$ is retrieved in a two stage fashion. Firstly, k_t and i_t are obtained in Steps 5 and 6, respectively, of Algorithm 7. The index i_t is used to determine, in Steps 7–9, the length of an eventual unaligned string that may precede the atomic alignment. The remaining components of the atomic alignment, i.e., j_t, i'_t, j'_t, and ϕ_t, are retrieved in a second stage by calling Algorithm 11 in step 10 with i_t and $\mathbb{S}$ as inputs.

Algorithm 11 reverses the steps of Algorithm 7. Notice that, during the reconstruction process, we do not have direct access to the number of transformation steps, K, of the encoding transformation ϕ. However, the length of a string resulting from applying ϕ to a reference string equals the sum of the lengths of the insertion and match operations in ϕ. Therefore, for the length of the aligned substring, z, retrieved in Step 1, we have

$$z = \sum_{k=1}^{k=K} \Big(I_k + M_k \Big).$$

Algorithm 11: Retrieve an atomic alignment from its stream representation in $\mathbb{S}$.

Input : Starting position i_t, and set of streams $\mathbb{S}$
Output: Atomic alignment $(i_t, j_t, i'_t, j'_t, \phi_t)$

1 Retrieve z from $\mathcal{E}$
2 Let $j_t = i_t + z - 1$
3 Retrieve the reference starting index i'_t from $\mathcal{S}'$
4 Let $z_1 = 0, j'_t = i'_t - 1, k = 1, \mathcal{I} = \lambda$
5 **repeat**
6 Retrieve I_k from stream $\mathcal{N}$
7 Retrieve S_k from stream $\mathcal{K}$
8 Retrieve M_k from stream $\mathcal{M}$
9 Add (I_k, S_k, M_k) to ϕ
10 Retrieve I_k base calls from $\mathcal{B}$ and append them to $\mathcal{I}$
11 Let $j'_t = j'_t + S_k + M_k$
12 Let $z_1 = z_1 + I_k + M_k$
13 Let $k = k + 1$
14 **until** $z_1 = z$
15 Retrieve strand direction indicator d from $\mathcal{D}$
16 **return** $(i_t, j_t, i'_t, j'_t, \phi_t)$

Consequently, the loop in Step 5 retrieves triplet operations, accumulating the lengths of the insertion and match operations in z_1, until z_1 reaches z.

7.1.3 RENANO$_2$: a reference-dependent compression scheme, with a reference-independent decompression scheme

In this section we present RENANO$_2$, a variation of RENANO$_1$ that makes the decompression process independent of the set of reference base call strings $\mathcal{R}$. The main idea is to create a new artificial reference string, r', composed of carefully selected parts of the set of reference base call strings $\mathcal{R}$ and encode it in the compressed file. The atomic alignments associated to the read base call strings are then modified to align against r' instead of the original strings in $\mathcal{R}$. At that point, we can compress the read base call strings of the FASTQ file by applying the same encoding scheme presented in Section 7.1.2.1, with $\mathcal{R} = \{r'\}$. Subsequently, we can decode the read base call strings by first decoding r', followed by applying the decoding scheme of Section 7.1.2.2, again with $\mathcal{R} = \{r'\}$. Notice that, since $\mathcal{R}$ consists of a single reference string, the index r_{k_t} of each alignment is unnecessary and, thus, RENANO$_2$ omits the encoding of the aligned reference string identity indexes stream $\mathcal{U}$.

To create the new reference string r', we start by analyzing the full alignments $A(q, \mathcal{R})$ obtained against the full reference set $\mathcal{R}$. We are interested in keeping

the parts of the reference base call strings in $\mathcal{R}$ that are deemed useful for the compression of the read base call strings q. More concretely, we first say that an atomic alignment, $\alpha_t(q, r_{k_t}) = (i_t, j_t, i'_t, j'_t, \phi)$, *uses* a position h of r_{k_t} if $i'_t \leq h \leq j'_t$. Naturally, the positions of a reference string r_k that are useful for compression are those that are actually used by atomic alignments, and we would like to discard positions that are not used by any atomic alignment. Moreover, taking into account that describing r' incurs a coding cost, positions used by only one atomic alignment are not beneficial for compression either, as both the raw base call symbol for that position in the reference r' and the alignment information need to be encoded. We empirically observe that good compression results are obtained by keeping the positions of the reference base call strings that are used by at least two atomic alignments, with no significant improvement for larger thresholds. Consequently, we say that h is a *surviving position* of r_k if it is used by at least two atomic alignments. We define the new reference r' as the concatenation of the bases in surviving positions of the reference strings. Figure 7.3 shows an example of the proposed construction of r'.

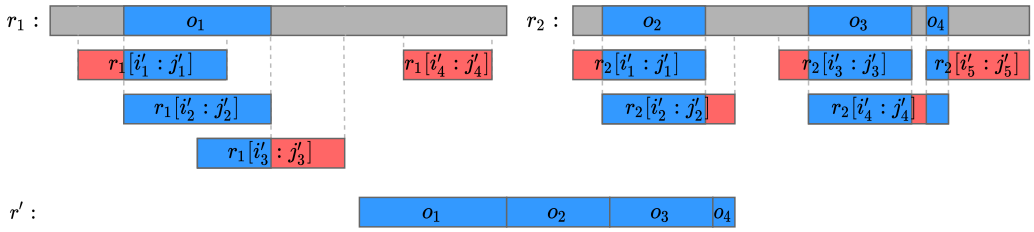


Figure 7.3: Example of the construction of an artificial reference string r'. The top row represents the original reference strings r_1 and r_2. The bottom row represents the constructed new reference r'. The three middle rows represent reference substrings, $r_{k_t}[i'_t : j'_t]$, used by the atomic alignments. The string r' is constructed by concatenating the bases in surviving positions (condensed in four segments, o_1, o_2, o_3, and o_4, marked in blue) of r_1 and r_2. Not-surviving positions (marked in gray) are discarded. In the reference substrings of the atomic alignments, the substrings that correspond to surviving positions are marked in blue, while substrings that correspond to not-surviving positions are marked in red.

Clearly, r' can be constructed in a single pass through all atomic alignments, by keeping count of the number of uses of each position of r_j, $1 \leq j \leq |\mathcal{R}|$. Moreover, as a byproduct of this construction, we can obtain a mapping ψ that maps each surviving position h of r_j to its corresponding position, $\psi(j, h)$, in r'.

Once the new reference string r' is built, RENANO$_2$ generates a binary encoding, denoted by $\text{Enc}(r')$, using the same technique proposed for encoding the stream of base call strings $\mathcal{B}$ in Section 1.2.1.2. The algorithm then stores $\text{Enc}(r')$ into the compressed file by outputting a fixed-size binary representation of its length $|\text{Enc}(r')|$, followed by $\text{Enc}(r')$ itself. Clearly, this process can be reversed on the

decoder side, giving the decoder access to r'.

To generate the artificial reference r', we made use of the information in the atomic alignments, which were aligned against the original reference strings in $\mathcal{R}$. However, the decoder has access to r', but not to $\mathcal{R}$. Therefore, before being described to the decoder, each atomic alignment, $\alpha_t(q, r_{k_t}) = (i_t, j_t, i'_t, j'_t, \phi_t)$, must be modified to align against r'. If the atomic alignment uses no surviving positions (see $r_1[i'_4 : j'_4]$ in Figure 7.3), it is fully discarded and removed from its respective full alignment. Otherwise, the atomic alignment is assigned a new reference substring of r', $r'[i''_t : j''_t]$, with $i''_t = \psi(k_t, h_i)$ and $j''_t = \psi(k_t, h_j)$, where h_i and h_j are the smallest and largest surviving positions used by $\alpha_t(q, r_{k_t})$, respectively. In addition, ϕ_t is modified so that $q[i_t : j_t] = \phi_t(r'[i''_t : j''_t])$, taking into account that some positions of the original reference substrings are no longer present in r'. Specifically, all triplet operations (I_k, S_k, M_k) where a portion of the match or skip operations refers to not-surviving positions, are transformed: match operations are transformed into insertion operations, and the portions of skip operations that lie on not-surviving positions are removed. If during this process a triplet operation (I_k, S_k, M_k) ends up with a value of $M_k = 0$, and it is not the last triplet of the encoding transformation, then the triplet is merged with the next triplet operation, $(I_{k+1}, S_{k+1}, M_{k+1})$, resulting in a single triplet operation $(I_k + I_{k+1}, S_k + S_{k+1}, M_{k+1})$.

At first sight, it may seem that the construction of r' requires loading all the atomic alignments into memory at once. In practice, however, we can build r' by loading, initially, only the information of the atomic alignments strictly required to determine the surviving positions, that is, the indexes of the aligned substrings k_t, i'_t and j'_t. The remaining information of each atomic alignment is loaded only at the time of its encoding, where also the transformation of ϕ_t is performed.

7.1.4 Alignment Information

In this section, we describe the format in which the alignment information is provided to RENANO$_1$ and RENANO$_2$, and how we construct, for each base call string q in the FASTQ file, a full alignment $A(q, \mathcal{R})$ (as defined in Section 7.1.1, in our internal representation) against the set of reference sequences $\mathcal{R}$.

To obtain the alignment information of a specific FASTQ file against a reference genome, our software implementation of RENANO receives an input text file in PAF format, typically used for nanopore sequence alignment. Each line of a PAF file, which represents an alignment between a *query* base call string and a *target* base call string, consists of a list of TAB-delimited fields. For our application, each query base call string is a portion of a read base call string of a FASTQ file to be compressed, and each target base call string is a portion of a base call string of a reference genome, stored in a FASTA file. In Table 7.1 we describe the fields that

are relevant to RENANO$_1$ and RENANO$_2$. Most fields are identified by position, with the only exception of the field labeled *cs*, which is identified through a tag at the beginning of the field.

Field	Type	Description
1	string	Query string name
3	int	Query start coordinate
4	int	Query end coordinate
5	char	'+' if query/target on the same strand; '-' if opposite
6	string	Target string name
8	int	Target start coordinate on the original strand
9	int	Target end coordinate on the original strand
cs	string	Base-alignment string in *cs* format

Table 7.1: Fields of the PAF alignment format relevant to RENANO$_1$ and RENANO$_2$.

Our implementation reads and parses the input PAF file, line by line, to generate full alignments for each base call string q of the FASTQ file. The first field in each line of a PAF file is the sequence identifier of the read base call string in the FASTQ file. This read base call string, together with the starting and ending positions in fields 3 and 4 of the PAF file line, respectively, determine the query base call string of the alignment represented by this line. Each read base call string of a FASTQ file, q, may participate in zero or more alignments; each one is represented by a separate line in a PAF file, all of which bear the same sequence identifier of q in the first field. If a read base call string q has no associated alignments, then $A(q, \mathcal{R})$ is set to an empty list, and $L_A = 0$. Otherwise, for each line in the PAF file where q is the query string, we build an atomic alignment $\alpha_t(q, r_{k_t}) = (i_t, j_t, i'_t, j'_t, \phi_t)$. Recall from Section 7.1 that RENANO$_1$ assumes that the atomic alignments of a full alignment $A(q, \mathcal{R})$ are non-overlapping, that is, that the aligned substrings, $q[i_t : j_t]$, of the atomic alignments in $A(q, \mathcal{R})$ do not overlap. However, this condition is not necessarily satisfied by the alignments extracted from the PAF file. Consequently, we first create an auxiliary full alignment $\bar{A}(q, \mathcal{R})$ by parsing every atomic alignment associated to q in the PAF file, and we later use $\bar{A}(q, \mathcal{R})$ to generate a non-overlapping full alignment $A(q, \mathcal{R})$.

To generate an atomic alignment $\alpha_t(q, r_{k_t}) = (i_t, j_t, i'_t, j'_t, \phi_t)$ from a line of the PAF file, we use the values in fields 3 and 4 to determine the aligned q substring, $q[i_t : j_t]$, where i_t is the value in field 3 and j_t is the value in field 4. We also use the values in fields 6, 8, and 9, to determine the reference substring of the atomic alignment, $r_{k_t}[i'_t : j'_t]$, where the index of the reference substring k_t is determined by the name of the target string in field 6, and fields 8 and 9 determine i'_t and j'_t, respectively. Finally, to determine the encoding transformation ϕ_t, we use field 5 and the *cs* tag. Field 5 indicates if the alignment is between the original query string and the target string, in which case the value is '+', or if it is between the

reverse complement of the query string and the target string, in which case the value is '-'. We use this field to determine the strand direction indicator d of ϕ_t, such that $d = 0$ if field 5 is '+', and $d = 1$ if field 5 is '-'. Finally, the *cs* field has a base-alignment string, called *cs*-string, which encodes the difference between the query string and the target string, as a sequence of string operations, which include: *matches*, *insertions*, *substitutions* (single nucleotide polymorphisms), and *deletions*. For more information on *cs*-strings we refer the reader to the Supplementary Data of [68]. We use Algorithm 12 to process the *cs*-string, and obtain the triplet sequence $\{(I_k, S_k, M_k)\}_{1 \leq k \leq K}$ and the insertions base call string, $\mathcal{I}$, of an encoding transformation ϕ_t, such that $q[i_t : j_t] = \phi_t(r_{k_t}[i'_t : j'_t])$.

The value of input d determines if the *cs*-string alignment utilizes the original query string, or its reverse complement. Thus, in Step 1, the algorithm starts by defining the auxiliary string $q' = \pi(q[i_t : j_t], d)$, which is used to build the insertions base call string. In Step 2, the algorithm initializes the insertions base call string $\mathcal{I}$ as the empty string, and the triplet sequence T as an empty sequence. The triplet operations and the insertions base call string are progressively constructed by scanning the *cs*-string and q' from start to end as the algorithm iterates over the *cs*-string operations, appending triplets to T and base call strings to $\mathcal{I}$ as necessary. The variable i, initialized to $i = 1$ in Step 3, maintains the starting position of the portion of q' that remains to be processed. The auxiliary variable I represents the insertion length value of the next triplet to be appended, while S represents the skip length value of the next triplet to be appended. Both variables are initialized to 0 in Step 3.

In Step 4, the algorithm loops over each operation, X, in the *cs*-string, and takes a different action depending on the type of operation being processed. If the operation is an insertion, the algorithm adds the length of the operation, L, to the current insertion length, I (Step 7), and appends the corresponding inserted base calls, $q'[i : i + L - 1]$, to string $\mathcal{I}$ (Step 8). The algorithm proceeds to adjust the value of index i accordingly (Step 9). If the operation is a deletion, the algorithm adds the deletion length L to the current skip length S, in Step 11. We interpret a substitution operation as a combination of an insertion and a deletion both of length 1. Therefore, for this kind of operations, the algorithm adds 1 to the value of I (Step 13), appends the base call in position i of q, $q'[i : i]$, to $\mathcal{I}$ (Step 14), and adds 1 to the current skip value S (Step 15). The index i is adjusted accordingly in Step 15. Finally, if the operation is a match, the algorithm checks, in Step 17, if the match length L is greater than a *minimum match length* threshold parameter, m_L. If it is, the triplet (I, S, L) is appended to T, and the variables I and S are reset to 0, in steps 18 and 19, respectively. Otherwise, the match operation is interpreted as an insertion and a deletion, both of length L. Hence, the algorithm adds L to

Algorithm 12: Obtain the sequence of triplet operations and the insertions base call string, from a *cs*-string, the aligned substring of q, and the strand direction indicator d.

Input : *cs*-string, base call substring $q[i_t : j_t]$, strand direction indicator d
Output: Sequence of triplets $\{(I_k, S_k, M_k)\}_{1 \leq k \leq K}$, and insertions base call string $\mathcal{I}$

Let $q' = \pi(q[i_t : j_t], d)$
Let $\mathcal{I} = \lambda$ be an empty string, and $T = \{\}$ an empty sequence
Let $i = 1$, $I = 0$, and $S = 0$
foreach *operation* X *in the cs-string* **do**
    Let L be the length of operation X
    if X *is an insertion* **then**
        $I = I + L$
        Append $q'[i : i + L - 1]$ to $\mathcal{I}$
        $i = i + L$
    else if X *is a deletion* **then**
        $S = S + L$
    else if X *is a substitution* **then**
        $I = I + 1$
        Append $q'[i : i]$ to $\mathcal{I}$
        $i = i + 1$, $S = S + 1$
    else if X *is a match* **then**
        if $L > m_L$ **then**
            Append (I, S, L) to T
            $I = 0$, $S = 0$
        else
            $I = I + L$, $S = S + L$
            Append $q'[i : i + L - 1]$ to $\mathcal{I}$
        end
        $i = i + L$
    end
end
if $I > 0$ *or* $S > 0$ **then**
    Append $(I, S, 0)$ to T
end
return T *and* $\mathcal{I}$

both I and S (Step 21), and appends string $q'[i : i + L - 1]$ to $\mathcal{I}$ (Step 22). Index i is adjusted accordingly in Step 24. As a last step, if there is a remaining triplet operation that has yet to be appended to T, that is, if I or S are greater than zero, the corresponding triplet is appended to T (Step 28).

The parameter m_L regulates the trade-off between the cost of encoding a potentially long sequence of atomized small matches, against the cost of encoding fewer insertion operations, which however incur an additional cost of encoding extra symbols in the insertions base call string. The effect of this parameter on compression performance is studied in Section 7.2.3, where we choose a default value for it.

Once the alignments in the PAF file associated to a base call string q are parsed and transformed into an auxiliary full alignment sequence $\bar{A}(q, \mathcal{R}) = \{\alpha_t(q, r_{k_t})\}_{1 \le t \le L_{\bar{A}}}$, we generate a non-overlapping full alignment, $A(q, \mathcal{R})$, by executing Algorithm 13 with $\bar{A}(q, \mathcal{R})$ as input.

Algorithm 13: Generate a non-overlapping full alignment $A(q, \mathcal{R})$.

Input : Auxiliary full alignment $\bar{A}(q, \mathcal{R}) = \{\alpha_t(q, r_{k_t})\}_{1 \le t \le L_{\bar{A}}}$ obtained from the PAF file.
Output: Non-overlapping full alignment $A(q, \mathcal{R})$

1 Sort sequence $\bar{A}(q, \mathcal{R}) = \{\alpha_t(q, r_{k_t})\}_{1 \le t \le L_{\bar{A}}}$ in decreasing order of j_t
2 Let $A(q, \mathcal{R}) = \{\alpha_1(q, r_{k_1})\}$
3 Let $\hat{\imath} = i_1$
4 **for** $t = 2$ *to* $t = L_{\bar{A}}$ **do**
5 **if** $j_t < \hat{\imath}$ **then**
6 Append $\alpha_t(q, r_{k_t})$ to $A(q, \mathcal{R})$
7 $\hat{\imath} = i_t$
8 **else if** $i_t < \hat{\imath}$ **then**
9 $\hat{\jmath}_t = \hat{\imath} - 1$
10 Obtain ϕ'_t, such that $q[i_t : \hat{\jmath}_t] = \phi'_t(r_{k_t}[i'_t : j'_t])$, by modifying ϕ_t
11 Append atomic alignment $\alpha_t(q, r_{k_t}) = (i_t, \hat{\jmath}_t, i'_t, j'_t, \phi'_t)$ to $A(q, \mathcal{R})$
12 $\hat{\imath} = i_t$
13 **end**
14 **end**
15 **return** $A(q, \mathcal{R})$

The algorithm starts by sorting the sequence of atomic alignments, $\bar{A}(q, \mathcal{R}) = \{\alpha_t(q, r_{k_t})\}_{1 \le t \le L_{\bar{A}}}$, in decreasing order of j_t (Step 1). In Step 2, the sequence $A(q, \mathcal{R})$ is initialized with the first atomic alignment of the sorted sequence, $A(q, \mathcal{R}) = \{\alpha_1(q, r_{k_1})\}$. The algorithm progressively appends atomic alignments to $A(q, \mathcal{R})$, such that $A(q, \mathcal{R})$ always holds a non-overlapping sequence of atomic alignments. Notice that $A(q, \mathcal{R})$ is initialized as a non-overlapping sequence, as it

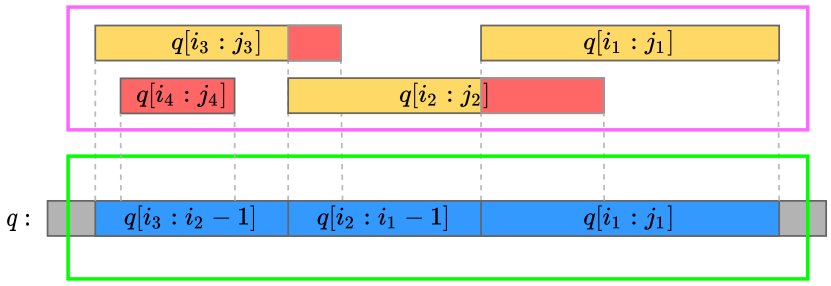


Figure 7.4: Example of a full alignment, $\bar{A}(q, \mathcal{R}) = \{\alpha_t(q, r_{k_t})\}_{1 \le t \le L_{\bar{A}}}$, obtained from a PAF file with overlapping between aligned q substrings, $q[i_t : j_t]$. The segments in the top box (pink) represent the original overlapping q substrings, while the segments in the bottom box (green) represent the result of running Algorithm 13 with $\bar{A}(q, \mathcal{R})$ as input.

contains a single atomic alignment. An example of a sequence of overlapping atomic alignments, sorted in decreasing order of j_t, is presented in the top (pink) box in Figure 7.4.

The variable $\hat{\imath}$, initialized to $\hat{\imath} = i_1$ in Step 3 of the algorithm, maintains the starting position of the last atomic alignment appended to $A(q, \mathcal{R})$. In Step 4, the algorithm loops over the remaining atomic alignments in $\bar{A}(q, \mathcal{R})$, and checks, in Step 5, if the ending position of the current atomic alignment, j_t, is smaller than $\hat{\imath}$. If so, the current alignment does not overlap with the alignments in $A(q, \mathcal{R})$ (as $i_t < j_t$), and it is directly appended to $A(q, \mathcal{R})$ in Step 6, while $\hat{\imath}$ is updated to $\hat{\imath} = i_t$, in Step 7. Otherwise, if j_t is greater or equal than $\hat{\imath}$, there is overlapping between $q[i_t : j_t]$ and $A(q, \mathcal{R})$. In this case, the algorithm checks, in Step 8, if i_t is smaller than $\hat{\imath}$. If so, the prefix $q[i_t : \hat{\imath} - 1]$ of $q[i_t : j_t]$ does not overlap with $A(q, \mathcal{R})$ and, thus, an atomic alignment for this portion of q is added to $A(q, \mathcal{R})$ (see the yellow portions of the examples $q[i_2 : j_2]$ and $q[i_3 : j_3]$ in Figure 7.4). First, the variable $\hat{\jmath}_t$ is initialized to $\hat{\jmath}_t = \hat{\imath} - 1$ in Step 9, which is the ending of the non-overlapping portion of $q[i_t : j_t]$. The algorithm then proceeds to construct ϕ'_t, such that $q[i_t : \hat{\jmath}_t] = \phi'_t(r_{k_t}[i'_t : j'_t])$, in Step 10, by modifying ϕ_t. Specifically, the parts of the triplet sequence and the insertions base call string of the encoding transformation ϕ_t that operate from position $\hat{\imath}$ to j_t are discarded. In Step 11, the modified version of the atomic alignment, $\alpha_t(q, r_{k_t}) = (i_t, \hat{\jmath}_t, i'_t, j'_t, \phi'_t)$, is appended to $A(q, \mathcal{R})$, and $\hat{\imath}$ is updated to $\hat{\imath} = i_t$ in Step 12. Notice that, if i_t is greater or equal than $\hat{\imath}$, i.e., if the condition in Step 8 is not satisfied, then $q[i_t : j_t]$ fully overlaps with $A(q, \mathcal{R})$ in which case it is discarded (see example $q[i_4 : j_4]$ in Figure 7.4). Finally, $A(q, \mathcal{R})$ is returned in Step 15. In Figure 7.4, the bottom (green) box shows the result of executing Algorithm 13 with the auxiliary full alignment in the top (pink) box as input.

7.2 Experimental results of RENANO

In this section we report on experiments performed on a collection of datasets of nanopore FASTQ files. We describe the datasets in Section 7.2.1. In Section 7.2.2 we explain how we generate the input PAF files that are necessary to run the proposed compression algorithms. In Section 7.2.3 we discuss the default values used for various algorithm parameters of RENANO$_1$ and RENANO$_2$. Finally, in Section 7.2.4 we evaluate the performance of RENANO$_1$ and RENANO$_2$ by comparing them against other compression tools.

To measure the performance of a compressor on a dataset, we compress each file of the dataset separately and calculate the compression ratio, *CR*, as defined in Section 6.3. To compare compression ratios, we use the percentage relative difference, as defined in Section 3.5. Finally, to compare overall performance between compressors we report *simple* and *weighted* averages of the results over the test datasets, the latter computed by weighting each result by the size of its corresponding dataset.

All experiments were conducted on a server with 80 64 bit x86 Intel Xeon CPUs, 503.5GB of RAM memory, and CentOS Linux release 7.7.1908.

7.2.1 Datasets

We evaluate the proposed algorithms on a collection of publicly available datasets, described in Table 7.2. The collection includes a metagenomic dataset (last entry in the table), which stores the result of sequencing a collection of genetic material from a mixed community of microbes.

Name	Num. Files	Total size (GB)	Description
hss	1	268	Human GM12878 Utah/Ceph cell line [52]
bra	18	46	Brassica napus L. [72]
sor	4	134	Sorghum bicolor Tx430 [29]
fly	1	17	Drosophila ananassae [76]
yst	5	6	Saccharomyces cerevisiae S288C [50]
mic	1	12	Microbial community (metagenomic) [65]

Table 7.2: Nanopore sequencing datasets used for evaluation.

The selected datasets cover a variety of dissimilar organisms including human, plant, animal, fungi, and bacteria. In the case of non-metagenomic datasets, for each dataset we obtained a reference genome file from the NCBI database [77], from which we extract the reference base call strings used in our algorithms. The metagenomic bacterial dataset, *mic*, demonstrates a scenario in which we do not know in advance the species that are present in the sequenced samples. For this case, we propose a pipeline of operations for constructing a reference sequence. It consists of performing a taxonomic classification of the dataset reads, and then concatenating the

reference genomes of the most prevalent organisms in the dataset. Several tools are available for the taxonomic classification step, such as FALCON [85], Kraken2 [105], and Centrifuge [56]. Specifically, the proposed pipeline identifies the most prevalent species in a FASTQ file by running and analyzing the output of Kraken2, downloads the corresponding reference genomes directly from the NCBI database, and concatenates these genomes into a single FASTA file, which serves as the reference sequence for compression.

In Table 7.3 we present the total size of the read base call strings in each dataset, the identification string of the reference genome file associated to the dataset (except for the metagenomic dataset), the total size of the reference base call strings of the genome, and the coverage of the dataset (total and average per file).

Name	Total reads sz. (Mbp)	Reference ID	Ref. sz. (Mbp)	Tot. cov.	Avg. cov. per file
hss	132931	GCF_000001405.39	3272	41x	41x
bra	23038	GCF_000686985.2	976	24x	1x
sor	66488	GCF_000003195.3	709	94x	23x
fly	8363	GCF_003285975.2	217	39x	39x
yst	3177	GCF_000146045.2	12	265x	53x
mic	6169	metagenomic*	61	101x	101x

Table 7.3: Read base call string information of the different datasets and their associated reference genomes. The total sizes of the read base call strings and the reference strings are presented in mega base-pairs (Mbp). The total coverage is calculated by dividing the total number of base call symbols in the dataset reads by the number of base call symbols in the strings of the reference genome. The average coverage per file was calculated by dividing the total coverage of the dataset by the corresponding number of files. *The reference sequence used for the microbial metagenomic dataset was constructed by first performing a taxonomic classification of the dataset reads, followed by concatenating the references of the most prevalent organisms.

The datasets cover different possible compression scenarios, such as having 268 GB of human data in a single file (hss), with 41x coverage, or having 46 GB of plant data distributed in 18 files (bra), with 1x average coverage per file, or having 12 GB of microbial metagenomic data in a single file, with 101x coverage.

7.2.2 PAF files generation

For our experiments, we generate PAF files with Minimap2 [68], a state of the art sequence alignment tool for long reads. We run Minimap2 with the FASTQ file we want to compress and the proper reference genome file as inputs. In turn, the tool outputs the result of aligning each base call string of the FASTQ file against the reference genome, in PAF format. Specifically, we execute Minimap2 with the following configuration options:

- *-x map-ont*: an option that sets the configuration of the tool to be optimized for reads generated with nanopore technologies. This option is recommended.

- *--cs*: an option that makes the tool perform base-alignment between the aligned sections of the read base call strings and the reference strings. The base-alignment is expressed as a *cs-string*. This option is necessary, as the base-alignment information is needed for our algorithms.

- *--secondary=no*: an option that makes the tool output only primary alignments. This prevents the tool from outputting multiple alignments for the same sections of the read base call strings. This option is recommended.

7.2.3 Algorithm parameters

In this section we specify the values of the parameters used in our implementations. First, we address the sizes of integers in the streams defined in Section 7.1.2.1. The following values were used in our experiments, but can eventually be modified as they are easily re-configurable: $\eta_{\mathcal{L}} = 24, \eta_{\mathcal{Q}} = 8, \eta_{\mathcal{S}} = 24, \eta_{\mathcal{E}} = 24, \eta_{\mathcal{U}} = 16, \eta_{\mathcal{S}'} = 32, \eta_{\mathcal{N}} = 16, \eta_{\mathcal{K}} = 16$, and $\eta_{\mathcal{M}} = 16$.

Furthermore, in Algorithm 12, we introduce the minimum match length threshold, m_L. This parameter determines which match operations in the base-alignment *cs*-strings are interpreted as match operations when transformed into the triplets of encoding transformations described in Section 7.1.1, and which ones are disregarded and interpreted as an insertion. To determine the default value of m_L, first we evaluate its impact on the compression performance of RENANO$_1$, by running the algorithm with different values of m_L, ranging from $m_L = 1$ to $m_L = 6$, on the datasets specified in Table 7.2. The compression results for the base call strings of each dataset are shown in Table 7.4.

Dataset	$m_z = 1$	$m_z = 2$	$m_z = 3$	$m_z = 4$	$m_z = 5$	$m_z = 6$
hss	0.118	**0.117**	0.118	0.119	0.120	0.122
bra	**0.141**	**0.141**	0.142	0.143	0.145	0.147
sor	0.161	**0.160**	0.161	0.162	0.164	0.167
fly	0.118	**0.117**	0.118	0.119	0.121	0.123
yst	0.174	**0.173**	**0.173**	0.174	0.175	0.177
mic	**0.158**	**0.159**	0.160	0.161	0.162	0.164

Table 7.4: Reads base call strings compression ratio of RENANO$_1$, with different values of minimum match length m_L. Best results for each dataset are bold-faced.

The results show that the best compression performance for most of the tested datasets is obtained at $m_L = 2$, which we set as the default value in RENANO$_1$ and RENANO$_2$ (the minimum is rather shallow, though, so the precise choice of value is not critical).

7.2.4 Comparative experimental results

In this section we evaluate RENANO$_1$ and RENANO$_2$ by comparing their performance against ENANO, and Genozip, a recently developed compression tool that can be used for nanopore data and offers both reference-based and reference-free compression modes.

To evaluate the performance of RENANO$_1$ and RENANO$_2$ against ENANO, we run each compressor on the datasets specified in Table 7.2. Each compressor is configured to run in its default configuration (see Section 7.2.3 for RENANO and [33] for ENANO). The read base call strings and total compression ratios obtained on each dataset are shown in Table 7.5.

Dataset	Read base call strings			Total		
	ENANO	RENANO$_1$	RENANO$_2$	ENANO	RENANO$_1$	RENANO$_2$
hss	0.236	**0.117 (-50.3)**	0.123 (-47.8)	0.452	**0.393 (-13.0)**	0.396 (-12.4)
bra	0.238	**0.141 (-40.7)**	0.202 (-15.2)	0.355	**0.307 (-13.6)**	0.337 (-5.1)
sor	0.245	**0.160 (-34.6)**	0.171 (-30.3)	0.374	**0.332 (-11.3)**	0.337 (-9.9)
fly	0.242	**0.117 (-51.5)**	0.123 (-49.0)	0.354	**0.292 (-17.3)**	0.295 (-16.5)
yst	0.236	**0.173 (-26.9)**	0.178 (-24.7)	0.297	**0.266 (-10.4)**	0.269 (-9.6)
mic	0.244	**0.159 (-35.0)**	0.162 (-33.8)	0.407	**0.364 (-10.5)**	0.365 (-10.2)
S. average	0.240	**0.145 (-39.8)**	0.160 (-33.5)	0.373	**0.326 (-12.7)**	0.333 (-10.6)
W. average	0.239	**0.133 (-44.4)**	0.146 (-39.2)	0.415	**0.362 (-12.6)**	0.368 (-11.1)

Table 7.5: Read base call strings and total compression ratios (*CR*) for ENANO, RENANO$_1$, and RENANO$_2$, on all the datasets. The percentage relative difference of RENANO$_1$, and RENANO$_2$with respect to ENANO are shown in parenthesis. The table also shows the simple (S.) and weighted (W.) *CR* averages. Best results for each dataset are bold-faced.

The results show that both RENANO$_1$ and RENANO$_2$ outperform ENANO for all the datasets. In particular, the best result for each dataset is achieved by RENANO$_1$. For read base call strings compression, RENANO$_1$ shows improvements relative to ENANO ranging from 26.9% (in *yst*) to 51.5% (in *fly*), with an average improvement of 39.8% over all the datasets. As for total compression, the improvements range from 10.4% (in *yst*) to 17.3% (in *fly*), with an average improvement of 12.7%. RENANO$_2$ also consistently improves over ENANO. For read base call strings, the improvements range from 15.2% (in *bra*) to 49.0% (in *fly*), while in terms of total size, the improvements range from 5.1% (in *bra*) to 16.5% (in *fly*).

Compared to RENANO$_1$, RENANO$_2$ achieves similar compression results in the datasets with high coverage per file (*hss* 41x, *sor* 23x, *fly* 39x, and *yst* 53x), with a relative percentage deterioration with respect to RENANO$_1$ ranging from 2.9% (in *yst*) to 6.9% (in *sor*). In the case of dataset *bra*, where the average coverage per file is 1x, the relative percentage deterioration between RENANO$_2$ and RENANO$_1$ reaches 43.3%. This is due to RENANO$_2$ directly benefiting from having multiple atomic alignments that use the same sections of the reference strings, which is less

likely to happen in files with low coverage. However, even for the dataset *bra*, which has an average coverage per file as low as 1x, RENANO$_2$ improves the compression of the read base call strings by 15.2% relative to ENANO, which leads to a total compression improvement of 5.1%. We also notice that for the metagenomic dataset, *mic*, for which we constructed a reference sequence following the pipeline described in Section 7.2.1, both RENANO$_1$ and RENANO$_2$ improve the base call strings compression performance of ENANO by 35.0% and 33.8%, respectively.

In Tables 7.6 and 7.7 we show the total compression and decompression times (in *h:mm:ss* format) and speeds (in MB/s), respectively, for each algorithm on each dataset. All the algorithms were run in multi-threading environments, using eight threads in each run.

Dataset	Total compression time (h:mm:ss)			Total decompression time (h:mm:ss)		
	ENANO	RENANO$_1$	RENANO$_2$	ENANO	RENANO$_1$	RENANO$_2$
hss	**0:37:09**	0:49:56	0:51:24	0:52:55	**0:49:40**	0:51:19
bra	**0:05:26**	0:11:32	0:08:30	**0:07:40**	0:07:52	0:09:34
sor	**0:13:09**	0:14:51	0:17:34	0:19:45	**0:19:15**	0:20:33
fly	**0:01:41**	0:03:23	0:02:08	0:02:30	**0:02:19**	0:02:26
yst	**0:00:49**	0:01:33	0:01:00	**0:01:05**	**0:01:05**	0:01:06
mic	**0:01:16**	0:01:20	0:01:33	0:01:53	**0:01:48**	0:01:51

Table 7.6: Encoding and decoding times (in *h:mm:ss* format) for ENANO, RENANO$_1$, and RENANO$_2$, on all the datasets. Best results, for each dataset, both for encoding and decoding, are bold-faced.

Dataset	Compression speed (MB/s)			Decompression speed (MB/s)		
	ENANO	RENANO$_1$	RENANO$_2$	ENANO	RENANO$_1$	RENANO$_2$
hss	**120**	90	87	84	**90**	87
bra	**142**	67	91	**101**	98	81
sor	**169**	150	127	113	**116**	108
fly	**168**	83	132	113	**122**	116
yst	**131**	70	107	99	**100**	97
mic	**162**	153	133	109	**114**	111
S. average	**149**	102	113	103	**107**	100
W. average	**139**	105	101	95	**100**	94

Table 7.7: Encoding and decoding speeds in MB/s for ENANO, RENANO$_1$, and RENANO$_2$, on all the datasets. The table also shows the simple (S.) and weighted (W.) averages of the results. Best results, for each dataset, both for encoding and decoding, are bold-faced.

The results show that ENANO is the fastest compressor for all the datasets, while RENANO$_1$ is on average the fastest decompressor. Specifically, during compression ENANO was 1.3x and 1.4x times faster on average than RENANO$_1$ and RENANO$_2$, respectively. During decompression, RENANO$_1$ was 1.05x and 1.1x times faster on average than ENANO and RENANO$_2$, respectively.

In table 7.8 we show the maximum memory required by each compressor during the encoding and decoding processes on all the files of each dataset.

Dataset	Max. compression memory use (GB)			Max. decompression memory use (GB)		
	ENANO	RENANO$_1$	RENANO$_2$	ENANO	RENANO$_1$	RENANO$_2$
hss	**0.220**	3.723	10.399	**0.231**	3.441	3.790
bra	**0.210**	1.187	1.539	**0.226**	1.182	0.868
sor	**0.214**	0.943	2.998	**0.227**	0.923	0.933
fly	**0.210**	0.441	1.134	**0.225**	0.439	0.410
yst	**0.205**	0.240	0.345	**0.223**	0.238	0.238
mic	**0.342**	0.432	0.701	**0.304**	0.370	0.345
Maximum	**0.342**	3.723	10.399	**0.304**	3.441	3.790

Table 7.8: Maximum memory usage (in GB) registered during the encoding and decoding processes, for all the compressors, on all the files of each dataset. Lowest memory usage, for each dataset, both for encoding and decoding, are bold-faced.

The results show that ENANO is the most efficient in terms of memory for all considered datasets, both for compression and decompression. This is in part due to RENANO$_1$ and RENANO$_2$ having to load a reference genome file into memory during the encoding and decoding processes (in the case of RENANO$_2$ the artificial reference is loaded during decoding). Also, in the case of RENANO$_2$, to generate the artificial reference during encoding, the algorithm needs to load information of each atomic alignment obtained from the PAF file, which makes the memory usage grow with the number of alignments. However, even for the alignment of the *hss* file of 286 GB, the memory usage is of 10.4 GB, which is manageable by a personal computer with 16 GB of RAM.

We also compare RENANO$_1$ and RENANO$_2$ against the tool Genozip [62], which has two modes: a reference-free compression mode (which we refer a to as *Gen*), and a reference-based compression mode (which we refer to as *Gen-ref*). For this comparison we run the selected tools on the nanopore datasets specified in Table 7.2. We execute both modes of Genozip configured to maximize compression (i.e., in their default configurations). In the case of the reference-based compression mode, we use the reference genome files presented in Table 7.3. In Table 7.9 we present the read base call strings, and total, compression ratios obtained by running both modes of Genozip, RENANO$_1$, and RENANO$_2$, on the selected datasets.

With respect to the results obtained by the two modes of Genozip, the reference-free mode achieved better results than the reference-based mode in all the datasets. Note also that the reference-based mode failed to compress the *hss* and the *sor* datasets. Therefore, to evaluate the performance of RENANO$_1$ and RENANO$_2$ we compare them against the reference-free mode of Genozip (Gen).

The results show that both RENANO$_1$ and RENANO$_2$ significantly outperform Genozip in each of the datasets. Specifically, regarding read base call strings com-

File from	Read base call strings				Total			
	Gen	Gen-ref.	RENANO$_1$	RENANO$_2$	Gen	Gen-ref.	RENANO$_1$	RENANO$_2$
hss	0.233	-	**0.117 (-49.5)**	0.123 (-46.9)	0.476	-	**0.393 (-17.5)**	0.396 (-16.9)
bra	0.236	0.282	**0.141 (-40.2)**	0.202 (-14.5)	0.370	0.400	**0.307 (-17.2)**	0.337 (-9.0)
sor	0.244	-	**0.161 (-34.0)**	0.171 (-30.1)	0.390	-	**0.332 (-14.8)**	0.337 (-13.6)
fly	0.244	0.282	**0.117 (-51.9)**	0.123 (-49.4)	0.385	0.400	**0.292 (-24.0)**	0.295 (-23.2)
yst	0.236	0.286	**0.173 (-27.0)**	0.178 (-24.8)	0.313	0.339	**0.266 (-14.8)**	0.269 (-14.0)
mic	0.244	0.281	**0.159 (-34.9)**	0.162 (-33.7)	0.417	0.435	**0.364 (-12.6)**	0.365 (-12.3)
S. average	0.240	0.283	**0.145 (-39.6)**	0.160 (-33.2)	0.392	0.394	**0.326 (-16.8)**	0.333 (-14.8)
W. average	0.237	0.282	**0.133 (-43.7)**	0.146 (-38.6)	0.435	0.400	**0.362 (-16.8)**	0.368 (-15.3)

Table 7.9: Read base call strings, and total, compression ratios for both modes of Genozip, RENANO$_1$, and RENANO$_2$, on all the datasets. The percentage relative difference of RENANO$_1$ and RENANO$_2$ with respect to *Gen* is presented in parenthesis. Best results for each dataset are bold-faced. The table also shows the simple (S.) and weighted (W.) averages of the results. The symbol '-' indicates the tool failed to compress the dataset.

pression, RENANO$_1$ and RENANO$_2$ improve on Genozip by 40.5% and 33.1% on average, respectively, and in total compression by 17.7% and 15.3% on average, respectively.

To summarize, in Figure 7.5 we show a comparison of the base call sequences *CR* obtained by RENANO$_1$, RENANO$_2$, ENANO, and Genozip, on the datasets specified in Table 7.2. For the results of RENANO$_2$ we also represent the space used for the compressed reference as the dashed area of the bars.

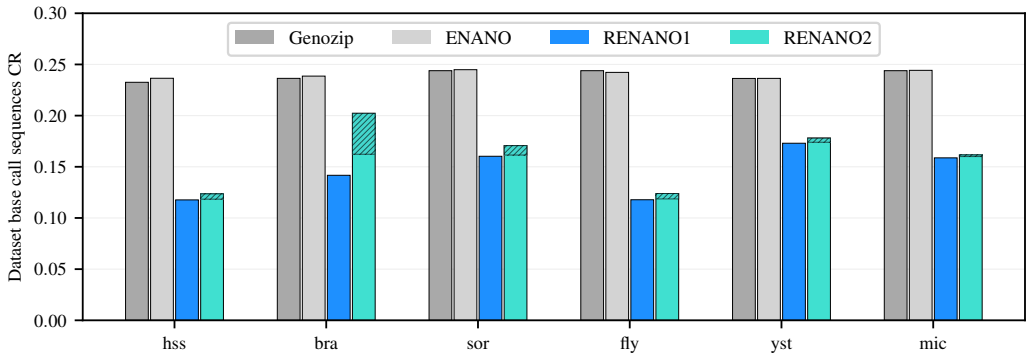


Figure 7.5: *CR* of each compressor on each tested dataset. The dashed area of the bars for RENANO$_2$ represents the space used for the compressed reference.

The results show that both RENANO$_1$ and RENANO$_2$ significantly outperform the previous compressors.

Chapter 8

Conclusions and Future work

We investigated the use of efficient compression techniques for two different scenarios of biological data compression, where computational efficiency is crucial.

In Part I, we studied the efficient compression of multi-channel biomedical signals, in an scenario where processing resources are scarce due to severe restrictions on energy consumption. In this sense, in Chapter 3 we developed GSC, a lossless multi-channel signal compressor based on the architecture proposed in [15], which reduces computational costs by using a simplified integer-arithmetic based prediction module. The obtained results show that the proposed module effectively reduces the complexity of the prediction algorithm, which results in better encoding and decoding times, without considerably hindering compression performance. The simplification of the prediction module was performed through the use of an algorithm proposed by Speck in [97]. Although the algorithm showed excellent results in practice for biomedical multi-channel signals, there does not exist any analysis of its performance. In this sense, a theoretical analysis of the performance of a Speck predictor remains an open problem, and a promising direction for further research.

From our experiments we observed that both the quantity and the order of the predictors used in the expert advice algorithm played a fundamental role in the computational cost of the algorithm. In this regard, in Chapter 4, we analyzed the performance of the predictors in the expert advice algorithm, and defined a series of criteria to be followed to create an optimized version of GSC for a specific type of signal.

Following the proposed criteria, in Chapter 5 we built OSC, a lossless multi-channel compressor optimized for EEG signals, by selecting a reduced set of predictors for the expert advice algorithm, and applying a series of selective updating techniques for the adaptive coefficients. We showed that OSC achieves a good trade-off between compression ratio and computational efficiency, by testing it on a series of EEG datasets, which cover a considerable number of different scenarios, contem-

plating sampling frequencies that vary between 160 and 1000 Hz, and number of channels that vary between 31 and 118. Specifically, we showed that OSC is significantly faster than GSC while achieving similar compression performance. For signals for which OSC was not designed, like seismographic or ECG signals, the compressor still performs well, even better than other popular multi-channel compression algorithms like MP4-ALS and Flac, although the gap with GSC is more notorious. These results reflect that, in general, a priori knowledge of a specific type of signal can improve both the compression and the computational performance of a compressor specifically designed for this kind of data.

Regarding the practical applicability of the proposed algorithms, in [31] a software implementation of OSC was embedded in a wireless electroencephalograph prototype, leading to significant improvement in the overall energy consumption of the instrument.

As a line of future research, it would be interesting to further investigate the applicability of GSC on other types of multi-channel signals, by constructing optimized compressors following the criteria presented in Chapter 4.

In Part II of the thesis we turned our focus to the compression of DNA sequencing data produced by HTS technologies, in particular, to the compression of data produced by nanopore sequencing technologies. In this sense, in Chapter 6, we presented ENANO, a lossless compression algorithm especially designed for nanopore sequencing FASTQ files, with its main focus being on the compression of the quality scores, which dominate the size of the compressed files. The results of testing ENANO showed that the proposed algorithm consistently achieves the best compression performance on every considered nanopore dataset against the compressors pigz and SPRING. In addition, in terms of encoding and decoding speeds, ENANO was significantly faster than SPRING, respectively, with a low memory consumption.

In line with the results obtained for biomedical signals, we managed to achieve state of the art performance, both in terms of compression and computational efficiency, by exploiting prior knowledge of a specific type of data, in this case of data generated by nanopore sequencing.

Recent work [59], has shown that the lossy compression of nanopore sequencing quality scores does not significantly affect (and sometimes even improves) the results obtained from some downstream bioinformatic tasks, such as variant calling. It would be interesting to further develop this research, by investigating how the lossy compression of quality scores can affect other bioinformatic tasks such as de novo genome assembly. In light of this research, it would also be interesting to add a module to ENANO for the lossy compression of quality scores sequences, and design the module to minimize the impairment of downstream tasks. Recently, there have been promising advancements in this line of work, which are reported in [59].

Lastly, in Chapter 7 we introduced RENANO, a reference-based lossless data compressor that improves on ENANO, by providing a more efficient base call sequence compression component. In this regard, two compression algorithms were introduced, corresponding to the following scenarios: a reference genome is available without cost to both the compressor and the decompressor; and the reference genome is available only on the compressor side, and a compacted version of the reference is included in the compressed file. Our experimental results showed that both modes of RENANO significantly improve the base call sequences compression of ENANO, and consequently, the total file compression performance. We also showed that both modes consistently outperform the recent general-purpose genomic compressor Genozip.

An apparent drawback of the current implementation of RENANO is that it receives the alignment information of the FASTQ file against a reference genome as an input produced by an external tool. Recent works [74, 59] have proposed compressors that perform their own alignment against a reference, or between the reads of the FASTQ file, achieving good compression results, but with demanding computational requirements. In this sense, developing an efficient internal module of alignment for RENANO, optimized for compression, is a natural direction for further research.

Bibliography

[1] Al Yami, S. and Huang, C.-H. (2019). LFastqC: A lossless non-reference-based FASTQ compressor. *PLOS ONE*, 14(11):1–10.

[2] Alberti, C., Daniels, N., Hernaez, M., Voges, J., Goldfeder, R. L., Hernandez-Lopez, A. A., Mattavelli, M., and Berger, B. (2016). An evaluation framework for lossy compression of genome sequencing quality values. In *2016 Data Compression Conference (DCC)*, pages 221–230.

[3] Anh, V. N. and Moffat, A. (2005). Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1):151–166.

[4] Antoniol, G. and Tonella, P. (1997). EEG data compression techniques. 44(2):105–114.

[5] Arnavut, Z. and Koak, H. (2009). Lossless EEG signal compression. In *Proc. 5th Int. Conf. Soft Computing, Computing with Words and Perceptions in System Analysis, Decision and Control*.

[6] Arram, J., Pflanzer, M., Kaplan, T., and Luk, W. (2015). Fpga acceleration of reference-based compression for genomic data. In *2015 International Conference on Field Programmable Technology (FPT)*, pages 9–16.

[7] Benoit, G., Lemaitre, C., Lavenier, D., Drezen, E., Dayris, T., Uricaru, R., and Rizk, G. (2015). Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph. *BMC Bioinformatics*, 16:288:1–288:14.

[8] Blankertz, B., Dornhege, G., Krauledat, M., Müller, K.-R., and Curio, G. (2007). The non-invasive Berlin brain–computer interface: Fast acquisition of effective performance in untrained subjects. *NeuroImage*, 37(2):539 – 550.

[9] Bonfield, J. K. and Mahoney, M. V. (2013). Compression of FASTQ and SAM format sequencing data. *PLOS ONE*, 8(3):1–10.

[10] Bonfield, J. K., McCarthy, S. A., and Durbin, R. (2018). Crumble: reference free lossy compression of sequence quality values. *Bioinformatics*, 35(2):337–339.

[11] Bousseljot, R., Kreiseler, D., and Schnabel, A. (1995). Nutzung der EKG-Signaldatenbank CARDIODAT der PTB über das Internet. *Biomedizinische Technik*, 40(1):S317–S318.

[12] Bowden, R., Davies, R. W., Heger, A., Pagnamenta, A. T., de Cesare, M., Oikkonen, L. E., Parkes, D., Freeman, C., Dhalla, F., Patel, S. Y., Popitsch, N., Ip, C. L. C., Roberts, H. E., Salatino, S., Lockstone, H., Lunter, G., Taylor, J. C., Buck, D., Simpson, M. A., and Donnelly, P. (2019). Sequencing of human genomes with nanopore technology. *Nature Communications*, 10(1):1869.

[13] Brisaboa, N. R., Ladra, S., and Navarro, G. (2013). Dacs: Bringing direct access to variable-length codes. *Information Processing Management*, 49(1):392–404.

[14] Burrows, M. and Wheeler, D. (1994). A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*. Citeseer.

[15] Capurro, I., Lecumberry, F., Martiń, , Ramírez, I., Rovira, E., and Seroussi, G. (2017). Efficient sequential compression of multichannel biomedical signals. *IEEE Journal of Biomedical and Health Informatics*, 21(4):904–916.

[16] Center, S. C. E. (2013). Caltech.dataset.

[17] Chandak, S., Tatwawadi, K., Ochoa, I., Hernaez, M., and Weissman, T. (2018a). SPRING: a next-generation compressor for FASTQ data. *Bioinformatics*, 35(15):2674–2676.

[18] Chandak, S., Tatwawadi, K., Ochoa, I., Hernaez, M., and Weissman, T. (2018b). Spring: A next-generation compressor for fastq data. *Bioinformatics*.

[19] Chandak, S., Tatwawadi, K., and Weissman, T. (2017). Compression of genomic sequencing reads via hash-based reordering: algorithm and analysis. *Bioinformatics*, 34(4):558–567.

[20] Charalampous, T., Kay, G. L., Richardson, H., Aydin, A., Baldan, R., Jeanes, C., Rae, D., Grundy, S., Turner, D. J., Wain, J., Leggett, R. M., Livermore, D. M., and O'Grady, J. (2019). Nanopore metagenomics enables rapid clinical diagnosis of bacterial lower respiratory infection. *Nature Biotechnology*, 37(7):783–792.

[21] Cleary, J. and Witten, I. (1984). Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402.

[22] Cock, P. J. A., Fields, C. J., Goto, N., Heuer, M. L., and Rice, P. M. (2009). The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research*, 38(6):1767–1771.

[23] Cover, T. M. and Thomas, J. A. (2006). *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. Wiley-Interscience.

[24] Cánovas, R., Moffat, A., and Turpin, A. (2016). CSAM: Compressed SAM format. *Bioinformatics*, 32(24):3709–3716.

[25] Dauwels, J., Srinivasan, K., Reddy, M. R., and Cichocki, A. (2013). Near-lossless multichannel eeg compression based on matrix and tensor decompositions. *IEEE Journal of Biomedical and Health Informatics*, 17(3):708–714.

[26] Delorme, A., Rousselet, G. A., Macé, M. J.-M., and Fabre-Thorpe, M. (2004). Interaction of top-down and bottom-up processing in the fast visual analysis of natural scenes. *Cognitive Brain Research*, 19(2):103 – 113.

[27] Deorowicz, S. (2020). FQSqueezer: k-mer-based compression of sequencing data. *Scientific Reports*, 10(1):578.

[28] Depledge, D. P., Srinivas, K. P., Sadaoka, T., Bready, D., Mori, Y., Placantonakis, D. G., Mohr, I., and Wilson, A. C. (2019). Direct rna sequencing on nanopore arrays redefines the transcriptional complexity of a viral pathogen. *Nature Communications*, 10(1):754.

[29] Deschamps, S., Zhang, Y., Llaca, V., Ye, L., Sanyal, A., King, M., May, G., and Lin, H. (2018). A chromosome-scale assembly of the sorghum genome using nanopore sequencing and optical mapping. *Nature Communications*, 9(1):4844.

[30] Dornhege, G., Blankertz, B., Curio, G., and Muller, K. R. (2004). Boosting bit rates in noninvasive EEG single-trial classifications by feature combination and multiclass paradigms. *IEEE Transactions on Biomedical Engineering*, 51(6):993–1002.

[31] Dufort y Álvarez, G., Favaro, F., Lecumberry, F., Martiń, , Oliver, J. P., Oreggioni, J., Ramírez, I., Seroussi, G., and Steinfeld, L. (2018). Wireless EEG system achieving high throughput and reduced energy consumption through lossless and near-lossless compression. *IEEE Transactions on Biomedical Circuits and Systems*, 12(1):231–241.

[32] Dufort y Álvarez, G., Seroussi, G., Smircich, P., Sotelo, J., Ochoa, I., and Martín, Á. (2019). Compression of nanopore FASTQ files. In *Bioinformatics and Biomedical Engineering*, pages 36–47, Cham. Springer International Publishing.

[33] Dufort y Álvarez, G., Seroussi, G., Smircich, P., Sotelo, J., Ochoa, I., and Martín, (2020). ENANO: Encoder for NANOpore FASTQ files. *Bioinformatics*, 36(16):4506–4507.

[34] Dufort y Álvarez, G., Seroussi, G., Smircich, P., Sotelo-Silveira, J., Ochoa, I., and Martín, (2021). RENANO: a REference-based compressor for NANOpore FASTQ files. *Bioinformatics*. btab437.

[35] Dutta, A., Haque, M. M., Bose, T., Reddy, C. V. S. K., and Mande, S. S. (2015). FQC: a novel approach for efficient compression, archival, and dissemination of FASTQ datasets. *Journal of bioinformatics and computational biology*, 13(03):1541003.

[36] Ginart, A. A., Hui, J., Zhu, K., Numanagić, I., Courtade, T. A., Sahinalp, S. C., and Tse, D. N. (2018). Optimal compressed representation of high throughput sequence data via light assembly. *Nature communications*, 9(1):1–9.

[37] Glentis, G.-O. and Kalouptsidis, N. (1995). A highly modular adaptive lattice algorithm for multichannel least squares filtering. *Signal Processing*, 46(1):47–55.

[38] Goldberger, A. L., Amaral, L. A. N., Glass, L., Hausdorff, J. M., Ivanov, P. C., Mark, R. G., Mietus, J. E., Moody, G. B., Peng, C.-K., and Stanley, H. E. (2000). PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals. *Circulation*, 101(23).

[39] Golomb, S. (1966). Run-length encodings (corresp.). *IEEE Transactions on Information Theory*, 12(3):399–401.

[40] Grabowski, S., Deorowicz, S., and Roguski, (2014). Disk-based compression of data from genome sequencing. *Bioinformatics*, 31(9):1389–1395.

[41] Greenfield, D. L., Stegle, O., and Rrustemi, A. (2016). GeneCodeq: quality score compression and improved genotyping using a Bayesian framework. *Bioinformatics*, 32(20):3124–3132.

[42] Guerra, A., Lotero, J., Édinson Aedo, J., and Isaza, S. (2019). Tackling the challenges of fastq referential compression. *Bioinformatics and Biology Insights*, 13:1177932218821373.

[43] Hach, F., Numanagić, I., Alkan, C., and Sahinalp, S. C. (2012). SCALCE: boosting sequence compression algorithms using locally consistent encoding. *Bioinformatics (Oxford, England)*, 28(23):3051—3057.

[44] Hernaez, M., Pavlichin, D., Weissman, T., and Ochoa, I. (2019). Genomic data compression. *Annual Review of Biomedical Data Science*, 2(1):19–37.

[45] Hsi-Yang Fritz, M., Leinonen, R., Cochrane, G., and Birney, E. (2011). Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Res*, 21(5):734–740.

[46] Huang, Z.-A., Wen, Z., Deng, Q., Chu, Y., Sun, Y., and Zhu, Z. (2017). Lw-fqzip 2: a parallelized reference-based compression of fastq files. *BMC Bioinformatics*, 18(1):179.

[47] Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101.

[48] Ip, C., Loose, M., Tyson, J., de Cesare, M., Brown, B., Jain, M., Leggett, R., Eccles, D., Zalunin, V., Urban, J., Piazza, P., Bowden, R., Paten, B., Mwaigwisya, S., Batty, E., Simpson, J., Snutch, T., Birney, E., Buck, D., Goodwin, S., Jansen, H., O'Grady, J., and Olsen, H. (2015). MinION analysis and reference consortium: Phase 1 data release and analysis [version 1; referees: 2 approved]. *F1000Research*, 4(1075).

[49] ISO/IEC MP4-ALS (2006). 14496-3:2005/Amd.2:2006, Information technology—Coding of audio-visual objects—Part 3: Audio, 3rd Ed. Amendment 2: Audio Lossless Coding (ALS), new audio profiles and BSAC extensions.

[50] Istace, B., Friedrich, A., d'Agata, L., Faye, S., Payen, E., Beluche, O., Caradec, C., Davidas, S., Cruaud, C., Liti, G., Lemainque, A., Engelen, S., Wincker, P., Schacherer, J., and Aury, J.-M. (2017). De novo assembly and population genomic survey of natural yeast isolates with the Oxford Nanopore MinION sequencer. *GigaScience*, 6(2).

[51] Jain, M., Koren, S., Miga, K. H., Quick, J., Rand, A. C., Sasani, T. A., Tyson, J. R., Beggs, A. D., Dilthey, A. T., Fiddes, I. T., et al. (2018a). Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature biotechnology*, 36(4):338.

[52] Jain, M., Koren, S., Miga, K. H., Quick, J., Rand, A. C., Sasani, T. A., Tyson, J. R., Beggs, A. D., Dilthey, A. T., Fiddes, I. T., Malla, S., Marriott, H., Nieto, T., O'Grady, J., Olsen, H. E., Pedersen, B. S., Rhie, A., Richardson, H., Quinlan, A. R., Snutch, T. P., Tee, L., Paten, B., Phillippy, A. M., Simpson, J. T., Loman, N. J., and Loose, M. (2018b). Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nature Biotechnology*, 36:338 EP –.

[53] Janin, L., Schulz-Trieglaff, O., and Cox, A. J. (2014). BEETL-fastq: a searchable compressed archive for DNA reads. *Bioinformatics*, 30(19):2796–2801.

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

[55] Kamamoto, Y., Harada, N., and Moriya, T. (2008). Interchannel dependency analysis of biomedical signals for efficient lossless compression by MPEG-4 ALS. In *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, pages 569–572.

[56] Kim, D., Song, L., Breitwieser, F., and Salzberg, S. (2016). Centrifuge: Rapid and sensitive classification of metagenomic sequences. *Genome Research*, 26(12):1721–1729. Publisher Copyright: © 2016 Szalaj et al.

[57] Kingsford, C. and Patro, R. (2015a). Reference-based compression of short-read sequences using path encoding. *Bioinformatics*, 31(12):1920–1928.

[58] Kingsford, C. and Patro, R. (2015b). Reference-based compression of short-read sequences using path encoding. *Bioinformatics*, 31(12):1920–1928.

[59] Kokot, M., Gudyś, A., Li, H., and Deorowicz, S. (2021). Colord: Compressing long reads. *bioRxiv*.

[60] Koski, A. (1997). Lossless ECG encoding. *Computer Methods and Programs in Biomedicine*, 52(1):23 – 33.

[61] Kozanitis, C., Saunders, C., Kruglyak, S., Bafna, V., and Varghese, G. (2011). Compressing genomic sequence fragments using SlimGene. *J Comput Biol*, 18(3):401–413.

[62] Lan, D., Tobler, R., Souilmi, Y., and Llamas, B. (2021). Genozip - A Universal Extensible Genomic Data Compressor. *Bioinformatics*.

[63] Lander, E. S. et al. (2001). Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921.

[64] Laver, T., Harrison, J., O'neill, P., Moore, K., Farbos, A., Paszkiewicz, K., and Studholme, D. J. (2015). Assessing the performance of the oxford nanopore technologies minion. *Biomolecular detection and quantification*, 3:1–8.

[65] Leidenfrost, R. M., Pöther, D.-C., Jäckel, U., and Wünschiers, R. (2020). Benchmarking the minion: Evaluating long reads for microbial profiling. *Scientific Reports*, 10(1):5125.

[66] Leipzig, J. (2016). A review of bioinformatic pipeline frameworks. *Briefings in Bioinformatics*, 18(3):530–536.

[67] Lemire, D., Boytsov, L., Kaser, O., Caron, M., Dionne, L., Lemay, M., Kruus, E., Bedini, A., Petri, M., Araujo, R. B., Damme, P., Dai, X., and Pavlov, P. (2021). The FastPFOR C++ library: Fast integer compression. https://github.com/lemire/FastPFOR. Accessed: 2021-12.

[68] Li, H. (2018). Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100.

[69] Lipman, D. and Pearson, W. (1985). Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441.

[70] Liu, Q., Sun, M., and Sclabassi, R. (2002). Decorrelation of multichannel EEG based on Hjorth filter and graph theory. In *Proc. 6th Int. Conf. Signal Proc.*, volume 2, pages 1516–1519 vol.2.

[71] Liu, Y., Yu, Z., Dinger, M. E., and Li, J. (2019). Index suffix–prefix overlaps by (w, k)-minimizer to generate long contigs for reads compression. *Bioinformatics*, 35(12):2066–2074.

[72] Malmberg, M. M., Spangenberg, G. C., Daetwyler, H. D., and Cogan, N. O. I. (2019). Assessment of low-coverage nanopore long read sequencing for snp genotyping in doubled haploid canola (brassica napus l.). *Scientific Reports*, 9(1):8688.

[73] Malysa, G., Hernaez, M., Ochoa, I., Rao, M., Ganesan, K., and Weissman, T. (2015). QVZ: lossy compression of quality values. *Bioinformatics*, 31(19):3122–3129.

[74] Meng, Q., Chandak, S., Zhu, Y., and Weissman, T. (2021). Nanospring: reference-free lossless compression of nanopore sequencing reads using an approximate assembly approach. *bioRxiv*.

[75] Merhav, N., Seroussi, G., and Weinberger, M. (2000). Coding of sources with two-sided geometric distributions and unknown parameters. 46(1):229–236.

[76] Miller, D. E., Staber, C., Zeitlinger, J., and Hawley, R. S. (2018). Highly contiguous genome assemblies of 15 drosophila species generated using nanopore sequencing. *G3: Genes, Genomes, Genetics*, 8(10):3131–3141.

[77] NCBI (2020). National Center for Biotechnology Information (NCBI). https://www.ncbi.nlm.nih.gov/. Accessed: 2020-08.

[78] Nicolae, M., Pathak, S., and Rajasekaran, S. (2015). LFQC: a lossless compression algorithm for FASTQ files. *Bioinformatics*, 31(20):3276–3281.

[79] Numanagić, I. (2016). *Efficient high throughput sequencing data compression and genotyping methods for clinical environments*. PhD thesis, Simon Fraser University.

[80] Numanagić, I., Bonfield, J. K., Hach, F., Voges, J., Ostermann, J., Alberti, C., Mattavelli, M., and Sahinalp, S. C. (2016). Comparison of high-throughput sequencing data compression tools. *Nature Methods*, 13(12):1005–1008.

[81] Ochoa, I., Asnani, H., Bharadia, D., Chowdhury, M., Weissman, T., and Yona, G. (2013). Qualcomp: a new lossy compressor for quality scores based on rate distortion theory. *BMC Bioinformatics*, 14(1):187.

[82] Ochoa, I., Hernaez, M., Goldfeder, R., Weissman, T., and Ashley, E. (2017). Effect of lossy compression of quality scores on variant calling. *Briefings in bioinformatics*, 18(2):183–194.

[83] Patro, R. and Kingsford, C. (2015). Data-dependent bucketing improves reference-free compression of sequencing reads. *Bioinformatics*, 31(17):2770–2777.

[84] Payne, A., Holmes, N., Rakyan, V., and Loose, M. (2018). BulkVis: a graphical viewer for Oxford nanopore bulk FAST5 files. *Bioinformatics*, 35(13):2193–2198.

[85] Pratas, D., Pinho, A. J., Silva, R. M., Rodrigues, J. M. O. S., Hosseini, M., Caetano, T., and Ferreira, P. J. S. G. (2018). FALCON-meta: a method to infer metagenomic composition of ancient DNA. *bioRxiv*.

[86] Rissanen, J. (1976). Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3):198–203.

[87] Rissanen, J. (1983). A universal data compression system. *IEEE Transactions on Information Theory*, 29(5):656–664.

[88] Rissanen, J. (1984). Universal coding, information, prediction, and estimation. *IEEE Transactions on Information Theory*, 30(4):629–636.

[89] Roguski, L. and Deorowicz, S. (2014). DSRC 2–Industry-oriented compression of FASTQ files. *Bioinformatics*, 30(15):2213–2215.

[90] Roguski, Ł., Ochoa, I., Hernaez, M., and Deorowicz, S. (2018). Fastore–a space-saving solution for raw sequencing data. *Bioinformatics*, 1:9.

[91] Sanderson, N. D., Street, T. L., Foster, D., Swann, J., Atkins, B. L., Brent, A. J., McNally, M. A., Oakley, S., Taylor, A., Peto, T. E. A., Crook, D. W., and Eyre, D. W. (2018). Real-time analysis of nanopore-based metagenomic sequencing from infected orthopaedic devices. *BMC Genomics*, 19(1):714.

[92] Schalk, G., McFarland, D., Hinterberger, T., Birbaumer, N., and Wolpaw, J. (2004). BCI2000: a general-purpose brain-computer interface (BCI) system. 51(6):1034–1043.

[93] Schneider, G. F. and Dekker, C. (2012). DNA sequencing with nanopores. *Nature Biotechnology*, 30(4):326–328.

[94] Shannon, C. E. (1948). A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423.

[95] Singer, A. C. and Feder, M. (1999). Universal linear prediction by model order weighting. *IEEE Transactions on Signal Processing*, 47(10):2685–2699.

[96] Sović, I., Šikić, M., Wilm, A., Fenlon, S. N., Chen, S., and Nagarajan, N. (2016). Fast and sensitive mapping of nanopore sequencing reads with graphmap. *Nature communications*, 7:11307.

[97] Speck, D. (1995). Fast robust adaptation of predictor weights from min/max neighboring pixels for minimum conditional entropy. In *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 234–238 vol.1.

[98] Srinivasan, K., Dauwels, J., and Reddy, M. R. (2011). A two-dimensional approach for lossless EEG compression. *Biomedical Signal Processing and Control*, 6(4):387 – 394.

[99] Srinivasan, K., Dauwels, J., and Reddy, M. R. (2013). Multichannel EEG compression: Wavelet-based image and volumetric coding approach. *IEEE Journal of Biomedical and Health Informatics*, 17(1):113–120.

[100] Stephens, Z. D., Lee, S. Y., Faghri, F., Campbell, R. H., Zhai, C., Efron, M. J., Iyer, R., Schatz, M. C., Sinha, S., and Robinson, G. E. (2015). Big data: astronomical or genomical? *PLoS biology*, 13(7):e1002195.

[101] Voges, J., Ostermann, J., and Hernaez, M. (2017). CALQ: compression of quality values of aligned sequencing data. *Bioinformatics*, 34(10):1650–1658.

[102] Weinberger, M. J., Seroussi, G., and Sapiro, G. (2000). The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS. *IEEE Transactions on Image Processing*, 9(8):1309–1324.

[103] Welch, J. P., Ford, P. J., Teplick, R. S., and Rubsamen, R. M. (1991). The Massachusetts General Hospital-Marquette Foundation Hemodynamic and Electrocardiographic Database – Comprehensive collection of critical care waveforms. *Journal of Clinical Monitoring*, 7(1):96–97.

[104] Wongsawat, Y., Oraintara, S., Tanaka, T., and Rao, K. (2006). Lossless multichannel EEG compression. In *Proc. 2006 IEEE Int. Symp. Circuits and Systems*.

[105] Wood, D. E., Lu, J., and Langmead, B. (2019). Improved metagenomic analysis with kraken 2. *Genome Biology*, 20(1):257.

[106] Yanovsky, V. (2011). ReCoil-an algorithm for compression of extremely large datasets of DNA data. *Algorithms for Molecular Biology*, 6(1):1–9.

[107] Yu, Y. W., Yorukoglu, D., Peng, J., and Berger, B. (2015). Quality score compression improves genotyping accuracy. *Nature biotechnology*, 33(3):240–243.

[108] Zhang, J., Long, X., and Suel, T. (2008). Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08, page 387–396, New York, NY, USA. Association for Computing Machinery.

[109] Zhang, Y., Patel, K., Endrawis, T., Bowers, A., and Sun, Y. (2016). A FASTQ compressor based on integer-mapped k-mer indexing for biologist. *Gene*, 579(1):75 – 81.

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

[111] Ziv, J. and Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536.

[112] Zukowski, M., Heman, S., Nes, N., and Boncz, P. (2006). Super-scalar ram-cpu cache compression. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, page 59, USA. IEEE Computer Society.