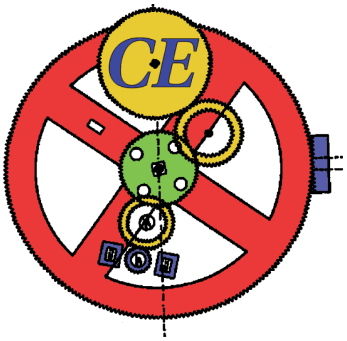

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2014

MSc THESIS

Compression of Next-Generation DNA Sequencing Data

Georgios Kathareios

CE-MS-2014-08

Abstract

DNA sequencing is the process of determining the ordered sequence of the four nucleotide bases in a strand of DNA, for storage in an electronic medium.
Since the mid-2000s, with the advent of Next-Generation sequencing technologies, the production rate of sequencing data has surpassed the rate with which hard disc prices are decreasing, meaning that storage hardware is becoming increasingly expensive. Thus, efficient methods to compress this kind of data are crucial for the future of genetic research.
Traditionally, this kind of data is compressed using generic compression techniques like gzip and bzip2. These techniques however do not distinguish between the three kinds of data present in the input file - identifier strings, base sequences and quality score sequences - and therefore cannot fully exploit their respective statistical dependencies, resulting in poor compression ratios.
In this master thesis, a specialized algorithm for the lossless compression of sequencing data is presented, aiming at high compression. A different compression technique is used for each part of a read: delta encoding for the id strings, linear predictive coding for the quality scores and high order Markov chain modelling for the nucleotide bases. Arithmetic coding is implemented and used as an entropy encoder, based on a different Markov chain modelling scheme for each part. Prior to the selection of these techniques, methods such as hidden Markov modelling and the Burrows-Wheeler transform were investigated, aspiring to improve the compression, but were eventually proven impractical. The resulting algorithm achieves compression rates as low as 19% of the initial size, comparable to state of the art algorithms, compressing the ids, bases and quality scores to 5.5%, 19.7% and 31.3% of their original size, respectively. Finally, we investigate the viability of achieving very high compression speeds for DNA sequencing data, by using a hardware implementation of the DEFLATE standard, which promises a 2GB/s compression speed. We show through simulations that ultra fast compression is possible, with a small degradation of approximately 1.38% in the compression ratio compared to the also DEFLATE-complying gzip.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Compression of Next-Generation DNA Sequencing Data

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Georgios Kathareios
born in Patra, Greece

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Compression of Next-Generation DNA Sequencing Data

by Georgios Kathareios

Abstract

DNA sequencing is the process of determining the ordered sequence of the four nucleotide bases in a strand of DNA, for storage in an electronic medium.

Since the mid-2000s, with the advent of Next-Generation sequencing technologies, the production rate of sequencing data has surpassed the rate with which hard disc prices are decreasing, meaning that storage hardware is becoming increasingly expensive. Thus, efficient methods to compress this kind of data are crucial for the future of genetic research.

Traditionally, this kind of data is compressed using generic compression techniques like gzip and bzip2. These techniques however do not distinguish between the three kinds of data present in the input file - identifier strings, base sequences and quality score sequences - and therefore cannot fully exploit their respective statistical dependencies, resulting in poor compression ratios.

In this master thesis, a specialized algorithm for the lossless compression of sequencing data is presented, aiming at high compression. A different compression technique is used for each part of a read: delta encoding for the id strings, linear predictive coding for the quality scores and high order Markov chain modelling for the nucleotide bases. Arithmetic coding is implemented and used as an entropy encoder, based on a different Markov chain modelling scheme for each part. Prior to the selection of these techniques, methods such as hidden Markov modelling and the Burrows-Wheeler transform were investigated, aspiring to improve the compression, but were eventually proven impractical. The resulting algorithm achieves compression rates as low as 19% of the initial size, comparable to state of the art algorithms, compressing the ids, bases and quality scores to 5.5%, 19.7% and 31.3% of their original size, respectively. Finally, we investigate the viability of achieving very high compression speeds for DNA sequencing data, by using a hardware implementation of the DEFLATE standard, which promises a 2GB/s compression speed. We show through simulations that ultra fast compression is possible, with a small degradation of approximately 1.38% in the compression ratio compared to the also DEFLATE-complying gzip.

Laboratory : Computer Engineering
Codenumber : CE-MS-2014-08

Committee Members :

Advisor:	Dr. Ir. Zaid Al-Ars, CE, TU Delft
Chairperson:	Prof. Dr. Ir. Koen Bertels, CE, TU Delft
Member:	Dr. Ir. Stephan Wong, CE, TU Delft
Member:	Dr. Ir. Fernando Kuipers, NAS, TU Delft

Dedicated to my family and friends

Contents

List of Figures

List of Tables

List of Acronyms

BWT Burrows-Wheeler Transform

DNA Deoxyribonucleic acid

DTMC Discrete Time Markov Chain

GenWQE Generic WorkQueue Engine

HMM Hidden Markov Model

IUPAC International Union of Pure and Applied Chemistry

LPC Linear Predictive Coding

LZ Lempel-Ziv

MTF Move-To-Front

NGS Next-Generation Sequencing

RNA Ribonucleic acid

SAM Sequence Alignment/Map

UMC Utrecht Medical Center

Acknowledgements

First of all, I would like to express my sincere thanks to Dr. Zaid Al-Ars, who believed in me, supervised, and trusted me to work individually with this project. Without his guidance, support and advice, none of this would have been possible. I would also like to thank Dr. Fernando Kuipers and Marcus Märtens for their precious suggestions, and Dr. RangaRao Venkatesha Prasad for his advice and interest in this work. Last but not least, I would like to thank Dr. Stephan Wong and Dr. Koen Bertels for agreeing to be in my examination committee.

Georgios Kathareios
Delft, The Netherlands
August 19, 2014

1 Introduction

1.1 Context

The field of genetics studies the genes − molecular units of heredity of living organisms − by gathering, processing and analysing the encoded information in DNA molecules that is passed from one generation to the next. Since the study of genetics and molecular biology is a key component in identifying and battling genetic diseases, such as cancer and leukemia, interest in the field has sky-rocketed in the past half century and continues to increase to this day. This increase in interest has been facilitated by technological advances of computer science and has brought forth technological breakthroughs such as the DNA sequencing process. Nowadays, modern genetics have been strongly linked with and are heavily dependent on the study of information management systems.

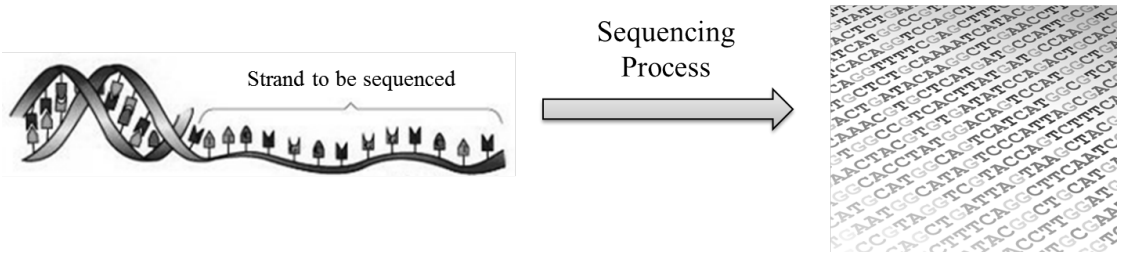

Figure 1.1: The DNA sequencing process.

The backbone of genetics is the *DNA sequencing* process (Fig. 1.1), the process of determining the precise order of nitrogenous bases within a strand of DNA [6]. The first methods for determining DNA sequences emerged in the 1970s, with Sanger's Nobel-winning chain-termination sequencing [38] becoming the method of choice. Early methodologies suffered from the limitation that only a few hundred base pairs could be sequenced in each experiment, meaning that approximately five million experiments would have to take place in order to sequence the whole human genome [6]. Therefore, in the following years researchers focused on optimizing these methods to be faster and able to sequence multiple samples in parallel. More recently, in the 1990s, the pyrosequencing method [36] revolutionized the process, as it is substantially faster and can be straightforwardly automated. There has been a bloom in the advent of new techniques since. Array-based pyrosequencing, sequencing-by-synthesis and sequencing-by-ligation techniques, developed after 2005, have exponentially increased the speed of the sequencing process and brought forth the Next-Generation sequencing (NGS) data [44, 29].

Due to these advances, a massive amount of NGS data is constantly being produced, processed and stored, and hence DNA sequencing is often categorized as a big data application [9, 44]. However, this rapid expansion of data generation creates a major

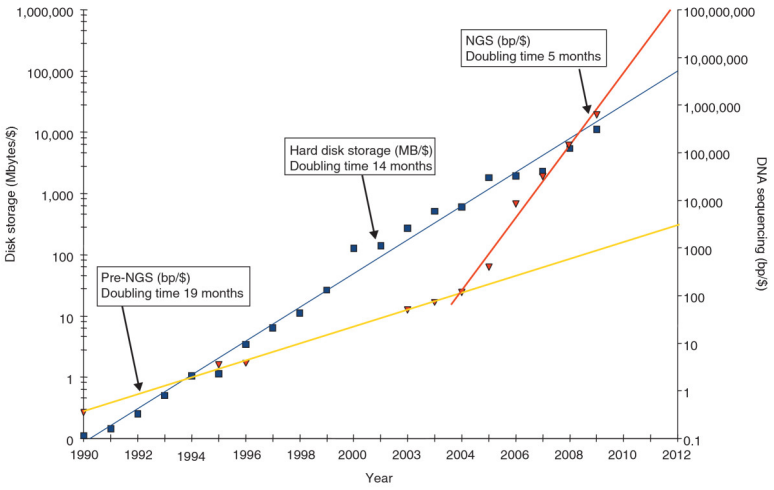

Figure 1.2: Historical trends in storage prices versus DNA sequencing costs. [44]

problem, as the need in storage space increases by the same rate. The problem is illustrated in Fig. 1.2, which shows the exponential decrease in the cost of sequencing in recent years. This trend is contrasted with the decrease in the cost of hard drive storage over the same time period, as predicted by Kryder's Law [47]. In the coming years, if the same trend persists, the cost of a DNA sequencing pipeline will be dominated by the storage cost, rather than the sequencing itself.

Therefore, efficient data compression methods are crucial in keeping the cost of such a system in manageable levels. Nowadays, generic data compressors such as gzip and bzip2 are widely used by biologists. However, these compressors are specifically designed to produce adequate results for a wide variety of input data types. By doing so, they overlook important statistical properties of NGS data that should be used towards more efficient compression results. A custom-made compressor that exploits these properties would be able to outperform the currently used techniques, and would greatly help in reducing the storage medium needs, effectively reducing the cost of the DNA sequencing process.

1.2 Background

This section aims to briefly present basic notions and background information that are required to follow the rest of the work. For a more detailed description on each topic, the reader should refer to the respective cited material.

1.2.1 Shotgun Sequencing

As already mentioned, the sequencing process can only identify a few hundred base pairs per experiment. A human DNA molecule contains approximately 3 billion base pairs. Hence, the practice of shotgun sequencing [6] is used to enable the identification of a whole genome. It involves the segmentation of the DNA strand in numerous fragments

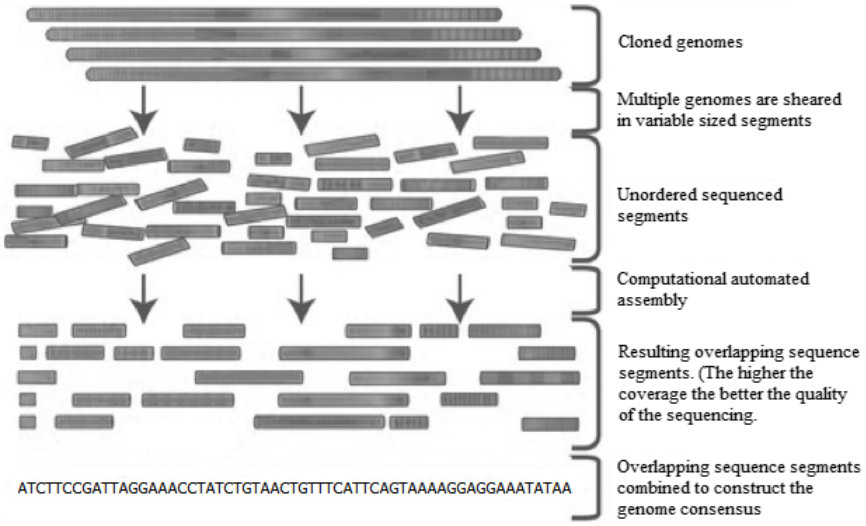

Figure 1.3: Illustration of the shotgun sequencing method.

that are individually sequenced. The sequenced fragments vary in length, typically in the range of 25 to 750 base pairs according to the sequencing technology, and are called short reads or simply *reads*. By performing the fragmentation and sequencing step multiple times over clones of the same DNA molecule, the resulting reads overlap.

Given a set of reads created by shotgun sequencing a genetic sequence, *coverage* is defined as the average number of reads representing a given nucleotide in the reconstructed sequence. Given the length G of the original genome, the number N of reads in the set, and the average length L of each read, the coverage of the set is calculated as follows:

$$C = \frac{N * L}{G} \tag{1.1}$$

Thanks to this induced read overlap, it is possible to recreate the initial genome almost entirely, by aligning and merging reads in a process called *sequence assembly* (Fig. 1.3). Two main variants of the assembly process exist:

- *De-Novo assembly* where the genome is recreated using only the available reads.
- *Mapping assembly* where the genome is recreated by utilizing a similar genome as reference.

Mapping assembly is fundamentally faster than de-novo assembly, but is not always applicable, as a reference genome may not always be available. The accuracy that modern assembly algorithms achieve in reconstructing the original sequence depends on the coverage of the provided reads. A coverage of 4 is generally required for an acceptable level of accuracy, while a coverage between 8 and 10 is required before the reconstructed sequence can be declared complete [6].

As a result, a high degree of data redundancy is introduced in high-coverage read sets, which increases the space required for storage in a hard disc drive. In the context of this work, the focus is given in the genetic data that are created directly after the

sequencing process, in the form of unaligned reads before the assembly, as they are the most demanding in terms of storage space requirements.

1.2.2 The FASTQ File Format

The FASTQ file format has emerged as a de facto format for storing NGS data in the form of reads [8]. It arose ad-hoc from several sources, primarily Sanger and Illumina. Its popularity is mainly attributed to its simplicity and the popularity of its predecessor, the FASTA format. However, it lacks standardization and several incompatible variants exist.

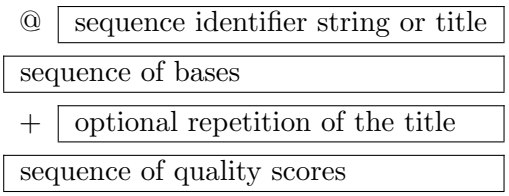

Figure 1.4: Read representation format in FASTQ.

FASTQ is a text-based, human-readable, ASCII encoded file format. The file is separated in 4-line blocks, each representing a different read. The 4 lines are formatted as shown in Fig. 1.4. An example of the representation of a simple read in the FASTQ format is shown in Fig. 1.5.

```
@HWI-ST867:117:CORCJACXX:2:1101:1439:1932 1:Y:0:CGATGT
NTTTCTATGTGTCTCACTTTNNCTGTACATATCANTGCAGNAATAAGACTA
+
#0;==?>?=>?@@:@@<<??##335==:?9);8>#119=?#07=?<??><?
```

Figure 1.5: Example of a read in FASTQ format.

The first line starts with the character '@' and contains an identifier string for the read. It is a free format field with no size limit, where arbitrary information may be added. According to the sequencing instrumentation used, the field may contain encoded information on the sequencing method, the instrument's name, various ids that correspond to the experiment, or simply an index and the read's length.

The second line contains the main information of the read, which is the sequence of bases that comprise it. Each base is represented with a single upper case character, taken from the IUPAC single letter code for DNA or RNA nucleotide bases (Table 1.1). In practice only A, C, T, G, and N are used, and white spaces or any other characters are not permitted in this field.

The third line begins with the character '+' and originally contained the same identifier string as the first line. However, that practice has been dropped in favour of smaller files, and thus the rest of the line is left empty.

IUPAC nucleotide code	Base
A	Adenine
C	Cytosine
G	Guanine
T (or U)	Thymine (or Uracil)
R	A or G
Y	C or T
S	G or C
W	A or T
K	G or T
M	A or C
B	C or G or T
D	A or G or T
H	A or C or T
V	A or C or G
N	any base
. or -	gap

Table 1.1: IUPAC representation codes for DNA or RNA nucleotide bases.

The fourth line contains the quality scores sequence, a sequence of characters equal in length with the nucleotide base sequence. Each character in this line is the encoded representation of the quality score of the corresponding nucleotide base of the second line. The quality score (a.k.a. PHRED score) of each base measures how probable it is for this particular base to have been incorrectly identified. These scores are being assigned to the bases during the sequencing process and are used by downstream applications to determine whether the identification of the particular base is trustworthy. There are two main variants on how the quality scores are computed, shown below (p_e is the probability of erroneous identification of a base and the $round()$ function rounds to the closest integer):

$$Q_{Sanger} = round\big(-10\ log_{10}p_e\big) \tag{1.2}$$

$$Q_{Solexa} = round\big(-10\ log_{10}(\frac{p_e}{1-p_e})\big) \tag{1.3}$$

The calculated values of the quality scores are then encoded using a single ASCII character. The printable ASCII characters 33 - 126 are reserved, with character 33 representing the quality value 0. However, Illumina pipelines after version 1.8 utilize only the quality scores in the range 0-41, corresponding to the characters in the range '!'-'J'. This range is also assumed in this work, without the loss of generality.

The FASTQ files that are used as benchmarks in this thesis were provided by the Utrecht Medical Center (UMC) and originate from human DNA sequencing. They are the ones shown in Tables 1.2 and 1.3.

Filename	Abbreviation
Non-injected_1_CGATGT_L002_R1_001	N21
ControleMan1_H879KADXX_CCGTCC_L001_R1_001	C11
ControleMan1_H879KADXX_CCGTCC_L001_R2_001	C12
ControleMan1_H879KADXX_CCGTCC_L002_R1_001	C21
ControleMan1_H879KADXX_CCGTCC_L002_R2_001	C22

Table 1.2: FASTQ benchmarks, courtesy of UMC

Benchmark	Size (GBs)	Number of reads	Read size
N21	2.33	15400080	51
C11	9.88	40424844	101
C12	9.88	40424844	101
C21	9.82	40206923	101
C22	9.82	40206923	101

Table 1.3: File properties of the FASTQ benchmarks

1.2.3 Data Compression

1.2.3.1 General Information

Compression is the process of transforming the representation of some input data X to a new representation X_C that requires fewer bits and is achieved through the elimination of redundant information. The inverse process involves the conversion of the compressed data X_C to the reconstructed data Y, and is called *decompression*. In this thesis, the term compression will refer to the pair of compression and decompression processes.

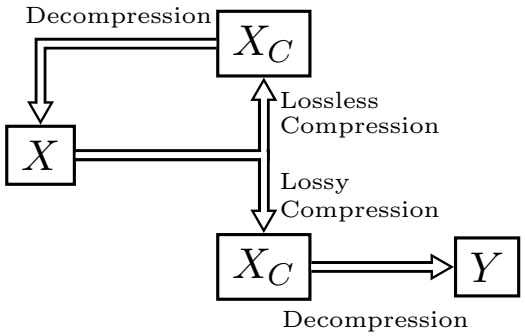

Figure 1.6: Lossless and Lossy compression.

There are two broad classes of compression (Fig. 1.6) :

- **Lossless compression:** When Y is identical to X.
- **Lossy compression:** When the reconstructed Y cannot be identical to X given only the information contained in X_C.

The usage of each of these types of compression depends on the type of input data and the reconstruction requirements [40]. Lossy compression achieves higher compression by removing information that is deemed useless from the input data. For example, in the compression of audio data, frequencies outside the acoustic range of the human auditory system are filtered and thus are not reconstructed. Therefore, this type of compression is ideal for media applications, and is indeed widely used in audio, image and video compression standards such as: MP3, JPEG, MPEG and H.264.

On the other hand, in lossless compression, no drop of information is acceptable [39]. Text compression is an important example of the utilization of lossless techniques, since the loss of information there could lead to the destruction of the text's coherence or distortion of its meaning.

In the context of genetic data compression, the loss of information is generally undesirable, since a wrong reconstruction of a set of nucleotide bases could cause a big portion of the data to be rendered useless. It would certainly be possible to drop information from the reads' identifier strings, or exclude nucleotide bases below a certain threshold of quality score to achieve higher compression. However, it is impossible to make such decisions, without knowledge of the information needed by downstream applications. As there are numerous applications that may need every piece of data, focus is given only on lossless compression techniques in this work.

1.2.3.2 Input Data

The input data of a compressor is a sequence of symbols. A *sequence* $S = s_1 s_2..s_N$ is defined as the concatenation of the symbols $s_i,\ i \in [1, N]$. The discrete *r-ary alphabet* $\mathcal{U} = \{u_1, u_2, .., u_r\},\ r \geq 2$ is the finite set containing all possible values for each symbol s_i. For example, the sequence 010110 is a sequence over the binary alphabet $\{0, 1\}$ and the word "compression" is a sequence over the alphabet of the English language.

A sequence is created by an *information source* $X(t)$, a discrete random variable which assumes the value $x_t \in \mathcal{U}$ at time point t. We denote as $X_i^j = x_i x_{i+1}..x_j$ the concatenation of the values of $X(t)$ in the time interval $[i, j]$. Any sequence S over $\mathcal{U}$ of length N can be generated by the source $X(t)$, starting at time point $t = i$, with probability $Pr\{X_i^{i+N} = S\}$.

The simplest information source is a *memoryless* or *independent and identically distributed* (iid) source. An information source is independent if and only if:

$$Pr\{X_i^{i+N} = S\} = \prod_{j=i}^{i+N} Pr\{X(j) = s_j\} \tag{1.4}$$

The independence property of the source indicates that the probability of encountering a specific value on a symbol does not depend on the value of any other symbol in the sequence. In addition, the symbols in the source sequence are identically distributed based on the probability mass function (pmf) $p()$[1], if Equation 1.6 holds.

$$Pr\{X(t) = u\} = p(u),\ \forall u \in \mathcal{U}, \forall t \in [1, N] \tag{1.6}$$

[1]For a sequence created by an iid source, the pmf can be calculated by measuring the frequency of

In simpler terms, the identical distribution property guarantees that the probability of encountering a specific symbol is the same at any position of the sequence. However, most information sources that are encountered in nature are not memoryless, and genetic data is not an exception to that rule.

1.2.3.3 Steps of the Compression

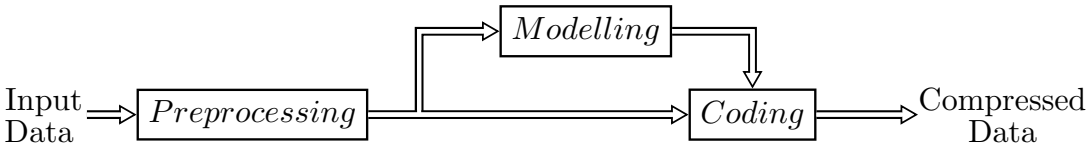

Figure 1.7: Lossless compression components.

There are three main steps in the compression process as shown in Fig. 1.7. The first step of *preprocessing* the data involves an initial transformation of the input sequence S_1 over $\mathcal{U}_1$ to a different sequence S_2 over $\mathcal{U}_2$ that is more highly compressible. It is of the utmost importance that the transformation that is used can be inverted on the decompressor, in order to reconstruct the data. In lossy compression, this is the step where the information drop occurs. An example of preprocessing in compression algorithms is the Lempel-Ziv algorithm in gzip and the motion prediction and motion compensation steps in the H.264 codec. Depending on the input data type, this step may be omitted, as the data may already be in an highly compressible form.

The next step is *modelling*. As the name suggests, the aim of this step is to create a model of the information source of the input data, which will in turn be used in the coding step. In order for the decompression to produce the desired result, the same model that was used to compress the data has to be utilized. This can be achieved in three ways:

- Both the compressor and decompressor can use a previously known model. For example, statistical models for the English language exist and can be used when compressing text of the English literature.
- The compressor creates a model based on the input data, and includes it in the compressed file. This technique is used in gzip compression, where the model is embedded in the compressed bitstream in the form of Huffman trees.
- The compressor can create the model gradually while encoding the input data. In this technique, the model does not need to be included in the compressed data, as long as the method of its creation is known to the decompressor. If this holds, the decompressor can construct the exact same model by performing the same model-construction method gradually as it decompresses the data.

each symbol:

$$p(u) = \frac{1}{N}\sum_{i=1}^{N} f_u(s_i), \text{ where } f_u(s_i) = \begin{cases} 1, & s_i = u \\ 0, & else \end{cases} \tag{1.5}$$

The final step of the compression is the *coding* step. In this step, the $S_2 = s_{21}s_{22}..s_{2N}$ sequence over $\mathcal{U}_2$ that has been created after preprocessing is transformed to a sequence S_b, usually over the binary alphabet, which is the final product of the compression. The model that was created in the previous step is utilized in order to achieve a reduction in the size needed to store S_b compared to the size of S_1.

A special case of coding is *entropy encoding*. Entropy encoding achieves size reduction by assigning a unique code to each symbol $u_i \in \mathcal{U}$, with the property that the size of the code representation of u_i is inversely proportional to the probability $Pr\{s_{2k} = u_i\}, \forall k \in [1, N]$. Two widely used forms of entropy encoding are Huffman encoding and arithmetic coding, which will play an important role in this thesis. Entropy encoders usually outperform other coding schemes like universal codes, but heavily depend on correct statistical modelling of the input information source.

1.2.3.4 Performance Measures

The basic measure of effectiveness for a compression technique is the *compression ratio* (CR). It is simply defined as the percentage of the input data size needed to store the compressed data:

$$Compression\ Ratio = \frac{Compressed\ Data\ Size}{Input\ Data\ Size} \tag{1.7}$$

Thus, a compression method that reduces a 10GB file to a 2.5GB file achieves 0.25 compression rate for that file[2].

Another useful metric for compression methods is the *speed of compression*. This is defined as the rate of converting an input file to a compressed one. The *compression speed* (CS) is measured in bytes per second, and can be calculated as follows:

$$Compression\ Speed = \frac{Size\ of\ uncompressed\ file}{Time\ needed\ to\ compress} \tag{1.8}$$

The *decompression speed* (DS) is defined similarly. It is the rate of recovering data from a compressed file, and can be calculated as follows:

$$Decompression\ Speed = \frac{Size\ of\ uncompressed\ file}{Time\ needed\ to\ decompress} \tag{1.9}$$

In addition, Shannon entropy [43] can be used as a measure of the amount of information present in a sequence. Assuming that a sequence S over the alphabet $\mathcal{U} = \{u_1, u_2, .., u_M\}$ is created by an iid source, its entropy can be calculated as follows:

$$H(S) = -\sum_{i=1}^{M} p(u) log(p(u)) \tag{1.10}$$

where $p()$ is the pmf used in Eq. 1.6. Depending on the base of the logarithm that is used, the result is measured in *bits* for a base of 2, or *nats* for a base of e.

[2]Sometimes also written as 4:1.

The entropy of a sequence quantifies the amount of its information. Therefore, it can be interpreted as the theoretical lower limit on the average number of bits needed to encode the sequence without the loss of information.

It should be noted that when the sequence is created by a non-iid source, Equation 1.10 can only approximate the sequence's entropy. Nevertheless, it is still useful for measuring the information content of a sequence.

1.2.4 Discrete Time Markov Chain

Discrete Time Markov Chains (DTMCs), or simply Markov Chains (MCs) [15, 30] are statistical processes used to model information sources. The modelled system is assumed to always be in some known state, with the possibility to undergo a transition to another state randomly in the discrete time domain.

A first-order DTMC λ is defined as the tuple $(N, \mathcal{S}, A, q_{init})$. $\mathcal{S}$ is a set of the N possible states of the system, comprising the *state-space* of the process. The state at time-point t is $s(t)$. The transition matrix $A : \mathcal{S} \times \mathcal{S} \rightarrow [0, 1]$ is a stochastic matrix with $A(q, q')$ being the probability for the system to undergo a transition from state q to state q'. Therefore, $A(q, q') = Pr\{s(t+1) = q' | s(t) = q\}$ and the following equation must hold:

$$\sum_{q' \in \mathcal{S}} A(q, q') = 1, \ \forall q \in \mathcal{S} \tag{1.11}$$

Finally, $q_{init} \in \mathcal{S}$ is the initial state of the model. An example of a simple DTMC with three states is shown in Fig. 1.8.

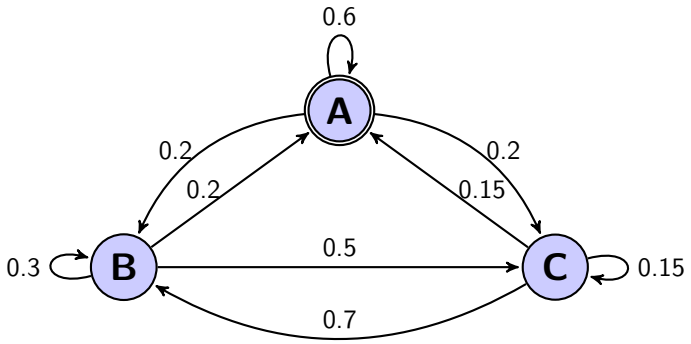

Figure 1.8: Example of a simple first-order Markov Chain.

The first-order DTMC holds the Markov property, that is, the probability of a transition only depends on the current state and not the states that came before it. In other words:

$$Pr\{s(t+1) | s(1), s(2), ..., s(t-1)\} = Pr\{s(t+1) | s(t)\} \tag{1.12}$$

Besides the first-order DTMC, it is possible to define a n-order DTMC, in which the probability of a transition only depends on the previous n states. That means that:

$$Pr\{s(t+1) | s(t), s(t-1), ..., s(1)\} = Pr\{s(t+1) | s(t), s(t-1), .., s(t-n)\} \tag{1.13}$$

An n-order DTMC is again defined as $\lambda_n = (N, \mathcal{S}, A, q_{init})$, with the only difference being the fact that the transition matrix is now a $\overbrace{\mathcal{S} \times .. \times \mathcal{S}}^{n \text{ times}} \rightarrow [0, 1]$ stochastic matrix.

In order to model an information source over the alphabet $\mathcal{U}$ with an n-order DTMC, we assume that transitions take place every time a symbol is created. The states correspond to the last n symbols emitted from the source, thus, there are $N = |\mathcal{U}|^n$ states in the model. Naturally, that would mean that the transition matrix contains $N \times N = |\mathcal{U}|^{2n}$ probabilities. However, a closer look to the modelled system would reveal that from the state $s_1 s_2 .. s_n$ the model can go to the states $s_2 s_3 .. s_n x,\ x \in \mathcal{U}$ only, since a single symbol is transmitted every time. Therefore the probability to undergo a transition to another state is equal to 0. Given that information, the transition matrix only needs to contain $N \times |\mathcal{U}| = |\mathcal{U}|^{n+1}$ probabilities.

The probabilities of the transition matrix can be calculated by measuring the frequency of each symbol and taking under consideration its previous n symbols. However, in the case of very long sequences, creating the model before the start of the compression can be a time-consuming task. This problem can be solved by *adaptive modelling*: parameters are trained and updated as data are compressed. The model is initialized with some arbitrary frequencies for each state. As more and more symbols are encountered, their respective frequencies are recorded, and the probabilities are updated. In addition, by downscaling the measured frequencies at given intervals, weight is given to more recently observed data, exposing local statistical properties of the data.

This technique gives some very important advantages to the modelling step. First, once the model is trained properly, it achieves a tight fit with the actual information source, as it takes advantage of the spatial locality of data. Second, the modelling is performed at the same time as the actual compression of the data, eliminating the need of two passes over the data, resulting in faster compression. Lastly, there is no overhead for storing the model along with the compressed data. As long as the initial parameters and the training method is known to the de-compressor, the same technique can be followed, and the model can be recreated on-the-go. The only drawback of adaptive modelling is the fact that until the model is properly trained, it lacks in accuracy. However, considering the huge amount of available data that is considered for the application of this thesis, proper training can be achieved with a small percentage of it, and as such this drawback can be overlooked.

Using a n-order DTMC as a model for an information source is rather straightforward. The n last symbols $s_{t-1}, s_{t-2}, .., s_{t-n}$ that were generated by the source point to the state $q_{t-1} = s_{t-1} s_{t-2} .. s - t - n$ that the DTMC currently has. The probabilities of a transition from this state to the next ones (which can only be the states $s_{t-2} s_{t-3} .. s_{t-n} x$, where $x \in \mathcal{U}$) correspond to the probabilities of encountering the symbol x next. Thus, the coding of symbol x can be performed using these probabilities, and subsequently the model parameters -probabilities and state- are updated. It is important that the update step comes after the compression step for the symbol x, so that the de-compressor is able to imitate this behaviour and produce the desired result. If for example the model was updated with x's value before the actual compression of the symbol, the de-compressor would not be able to make the same update, simply because it does know the exact value of x yet.

1.2.5 Arithmetic Coding

Arithmetic coding [37, 39, 40] is a technique for entropy coding, which, compared to other methods, stands out for its effectiveness and versatility. The following are some of its strong points:

- Provably optimal compression for iid sources.
- Near-optimal compression for non-iid sources.
- Clear distinction between modelling and coding - changing the modelling parameters (for example in adaptive modelling) does not interfere with the coding process.
- Effective in a wide range of applications.
- Its main process consists of arithmetic operations, which can be efficiently implemented in high-performance hardware.

Arithmetic coding is different from other entropy coding methods which represent each symbol with a unique binary value[3]. It codes one symbol at a time, and assigns to it a real-valued number of bits, or in other words, it assigns a unique binary value to the whole sequence.

More specifically, a sequence $S = s_1 s_2..s_N$ over the alphabet $\mathcal{U} = \{0, 1, ..., M-1\}$[4] is converted to a *code value* v, a real number in the interval $[0, 1)$. The code value of every possible sequence is unique and is represented in the output file by its *codeword* d, which is the fractional part of v in binary format. For example, if a sequence is assigned with the code value 0.19287109375, we have:

$$\begin{array}{lll} \text{code value} & v = & 0.192871093754_{10} = \quad 0.\underbrace{0011000101100_2}_{0011000101100} \\ \text{codeword} & d = & \end{array}$$

Therefore, a mapping is defined between the infinite number of sequences of symbols from the alphabet $\mathcal{U}$ to the infinite real values in the interval $[0, 1)$.

In order to acquire the mapping, the probability of each symbol at some point of the sequence, i.e

$$p_k(u) = Pr\{s_k = u\}, \forall k \in [1, N], \forall u \in \mathcal{U}, \tag{1.14}$$

is assumed known from the modelling step of the compressor. Also, we assume

$$p_k(u) \neq 0, \ \forall k, u \tag{1.15}$$

and define the cumulative distributions $c_k(u)$ and $d_k(u)$:

$$c_k(0) = 0 \tag{1.16}$$

$$c_k(u) = \sum_{i=0}^{u-1} p_k(i), \ u = 1, ..., M-1 \tag{1.17}$$

$$d_k(u) = \sum_{i=0}^{u} p_k(i), \ u = 0, 1, ..., M-1 \tag{1.18}$$

[3] Like Huffman [22] or Golomb [19] encoding.

[4] Every alphabet can be mapped to a $\{0, 1, ..., M-1\}$ alphabet, so this convention is assumed for simplicity without the loss of generality.

1.2.5.1 Encoding process

The arithmetic encoding process consists of creating a sequence of nested intervals in the form $\Phi_k(S) = [\alpha_k, \beta_k),\ k = 0, 1, .., N$, with $0 \leq \alpha_k \leq \alpha_{k+1}$ and $\beta_{k+1} \leq \beta_k \leq 1$. Starting from $\Phi_0(S) = [0, 1)$, the interval $\Phi_k(S)$ is created by dividing $\Phi_{k-1}(S)$ in M portions, each proportional to the probability $p_k(u),\ u = 0, 1, .., M-1$. Then, the interval corresponding to the currently encoded symbol s_k is selected as the interval $\Phi_k(S)$. Thus, the interval creation can be summarized with a set of recursive equations:

$$\Phi_0(S) = [0, 1) \tag{1.19}$$

$$\Phi_k(S) = [\alpha_k, \beta_k) = [\alpha_{k-1} + l_{k-1} \cdot c_k(s_k),\ \ \alpha_{k-1} + l_{k-1} \cdot d_k(s_k)),\ k = 1, .., N \tag{1.20}$$

where

$$l_k = \beta_k - \alpha_k$$

Once $\Phi_N(S)$ is defined, any real number in that interval can be selected as the uniquely decodable code value v of that sequence. Hence, the value with the shortest codeword is selected. One way to achieve this is by using the following recursive equations to find an interval $\Psi_K = [\gamma_K, \delta_K) \subset \Phi_N(S)$:

$$\Psi_0 = [\gamma_0, \delta_0) = [0, 1) \tag{1.21}$$

$$\Psi_k = [\gamma_k, \delta_k) = \begin{cases} [\gamma_{k-1},\ \gamma_{k-1} + \frac{\delta_{k-1} - \gamma_{k-1}}{2}),\ \text{if } \beta_N < \delta_k \\ [\gamma_{k-1} + \frac{\delta_{k-1} - \gamma_{k-1}}{2},\ \delta_{k-1}),\ \text{else} \end{cases} \tag{1.22}$$

In simpler words: Starting from $[0, 1)$ again, the Ψ_k interval is halved recursively and one of the halves is selected. The procedure continues until an interval $\Psi_K = [\gamma_K, \delta_K)$ that is entirely contained in $\Phi_N(S)$ is reached. At this point, γ_K is selected as the code value of S and the codeword will have K bits.

k	**symbol**	$\mathbf{a_k}$	$\mathbf{b_k}$
0	-	0	1
1	2	0.6	1
2	1	0.68	0.84
3	0	0.68	0.712

Table 1.4: Calculation of the interval $\Phi_N(3)$

As an example, consider the encoding of the sequence $S = 210$ over the alphabet $\mathcal{U} = \{0, 1, 2\}$, with $p_k(0) = 0.2$, $p_k(1) = 0.4$ and $p_k(2) = 0.4$ for all k. The encoding process is shown in Fig. 1.9. First the interval $\Phi_N(S)$ is determined using the recursive equations 1.19-1.20, as shown on Table 1.4. Afterwards, the codeword is determined by finding the interval Ψ_k that entirely includes $[0.68, 0.712)$, using the equations 1.21-1.22. This interval is found to be $[0.703, 0.6875)$ and thus, the code value selected for S is $0.6875 = 0.101100_2$, which gives the codeword $d = 101100$.

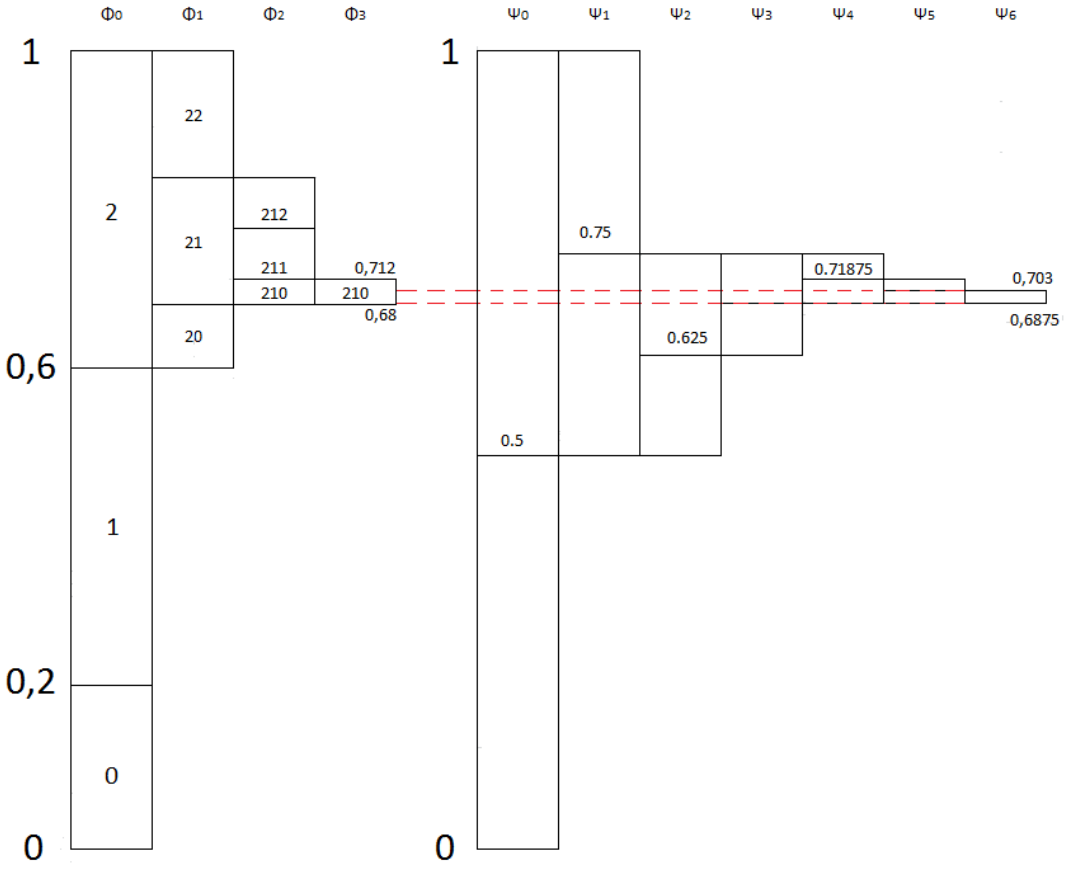

Figure 1.9: Example of how the intervals change in arithmetic encoding. On the left: The encoding process to produce the interval $\Phi_N(S)$, on the right: Determining the codeword for that interval.

1.2.5.2 Decoding process

The decoding process consists of the recreation of the nested intervals $\Phi_k(S)$, given the codeword b (which is converted to the code value v) and the same model as the encoder. In the same manner, starting from $[0, 1)$, the interval is partitioned according to the probability of each symbol. Then, the interval that contains the real number associated with the codeword is selected and the process is repeated. The equations used for decoding are:

$$\Phi'_0(S) = 0 \tag{1.23}$$

$$s_k = \{s : low_k(s) \leq v < high_k(s)\},\ k = 1, ..., N \tag{1.24}$$

where:

$$low_k(s) = \alpha_{k-1} + l_{k-1} \cdot c_k(s)$$
$$high_k(s) = \alpha_{k-1} + l_{k-1} \cdot d_k(s)$$

$$l_k = \beta_k - \alpha_k$$

$$\Phi'_k(S) = [low_k(s_k),\ high_k(s_k)),\ k = 1, ..., N \tag{1.25}$$

There are two ways to determine when the decoding iterations should end. The first is to specify the end of the sequence with a special character and the second is to know the length N before-hand. In this thesis the latter is used, as the size of all sequences is known.

1.2.5.3 Implementation issues

In the Equations 1.19-1.20 and the example in Fig. 1.9, it is obvious that the length of the interval $\Phi_k(S)$ shrinks with each iteration. In hypothetical infinite-precision systems this would not pose any threats, however, there is a possibility of underflow in the finite precision representation of real-life systems.

This problem can be solved by rescaling any interval with length that drops below 0.5. However, the rescaling has to happen in a manner that preserves the information gathered up to that point and at the same time, it can be mimicked exactly by the decoding process. Thus, the encoding is carried out incrementally, meaning that any gathered information is stored before the rescaling step as the sequence is being compressed, rather than wait for end of the whole process. When the interval to be rescaled is contained entirely in the upper of the lower half of $[0, 1)$, the first bit of the codeword is decided[5]. Thus, this bit can be stored and the interval rescaled without the loss of information. In this case, the rescaling is performed by mapping $[0.5, 1)$ or $[0, 0.5)$ to $[0, 1)$ (mapping E_1 and E_2 respectively). Fig. 1.10 shows an example.

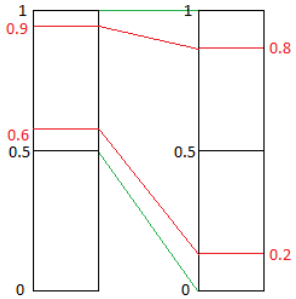

Figure 1.10: Interval rescaling in arithmetic coding.

In the case where the interval straddles the midpoint of the unit interval, the rescaling maps the interval $[0.25, 0.75)$ to $[0, 1)$ instead (mapping E_3). However, the first bit is not decided yet, so there is nothing to store at this point. Instead, the encoder counts the number of successive E_3 mappings it performs and on the next E_1 or E_2 mapping, it stores more than 1 bits to show that E_3 mappings have been performed. This process is then imitated by the decoder, by performing the exact same rescaling on the interval, but on the code value v as well.

[5] 1 for the upper half, since $v > 0.5 = 0.1_2$ and 0 for the lower half.

The second issue that needs to be addressed for an implementation of an arithmetic coder is the fact that all computations require real arithmetic. In the case where a floating point representation is used, the speed of an arithmetic coder would be hindered in most platforms. Therefore, there is a need to use fixed-point representation and integer arithmetic. This can indeed be performed by selecting the precision P to use for the implementation, and mapping the interval limits from the real range $[0,\ 1)$ to the integer range $[0,\ 2^P-1)$. For example, with 32 bit precision, the interval $[0.23,\ 0.5)$ is represented as $[987842477, 2147483648)$. This means that the minimum interval length that can be represented is $1/2^P$, but interval rescaling already takes care of this problem. Also, some precision is inevitably lost in the calculations, although the impact in compression is minimal with a high enough P, while the benefit in speed is considerable. In addition, all probabilities used in the computations can be represented as frequencies of occurrence of the respective symbol, thus eliminating the need for any real arithmetic.

The complete algorithms for arithmetic coding and decoding, including interval rescaling and integer arithmetic, that are implemented in this thesis are shown in Appendix A.

1.3 Problem Statement

The aim of this thesis is to create a custom compression algorithm for Next Generation DNA Sequencing data, with the following characteristics:

- Lossless, non-reference based compression
- High compression performance
- Manageable execution times

To work towards this goal, different techniques have to be investigated for each of the three types of information that are present in this kind of files, hoping to uncover the statistical properties of each one of them, so that they can be used for decreasing the size of the compressed files.

1.4 Thesis Outline

This thesis report is organized as follows:

Chapter 2 presents the related work which includes the most popular general compression algorithms, along with prominent custom compression techniques.

Chapter 3 describes our efforts for compressing the sequences of quality scores, using two different techniques: the first performs preprocessing with linear prediction, while the second attempts more accurate modelling with hidden Markov models.

Chapter 4 is dedicated in the compression of the identifier strings and base sequences. For the first ones, only one technique is considered, since it proves to be extremely effective. For the base sequences on the other hand, preprocessing with the Burrows-Wheeler transform is attempted initially, but is eventually skipped, in favour of a simple compressor with Markov chain modelling and arithmetic coding.

Chapter 5 discusses the resulting compression performance that is achieved by the techniques that were deemed best in the previous two chapters, by presenting our final implementation.

Chapter 6 investigates the viability of real-time compression with a hardware implemented, DEFLATE-compatible compressor, and finally, Chapter 7 concludes the thesis and proposes some recommendations for future work.

Related Work 2

In this chapter, the main compression techniques that are used for the compression of FASTQ files are presented. These are divided in two main categories: General compression techniques and domain-specific compression techniques.

Section 2.1 presents the fist category, general compression techniques. These techniques were designed to suit a wide variety of applications, and have existed since before the need of compression on NGS data. However, thanks to their widespread use in various contexts, they were adopted as easily available solutions in the compression of NGS data, as well.

On the other hand, domain-specific compression techniques (Section 2.2) are the ones that were designed specifically to compress NGS data. They generally achieve better compression rates, since they take advantage of the statistical properties of this kind of data. Despite their increased performance, they are not as widely used by geneticists as the legacy applications of general compression techniques. This trend is likely to change in the near future, as the need for better compression increases.

2.1 General compression techniques

2.1.1 Gzip

Gzip is a file format and software application used for file compression and decompression, designed for text, but applicable to any file type. It was developped by Jean-loup Gailly for the GNU project [17] in 1993, and it has been extremely popular for applications where a high compression rate is not crucial.

The compression method of Gzip is based on the DEFLATE format specification [13, 12], the same specification that is used by the PNG format for images [23] and the popular compression library zlib [14].

Gzip is a block-based compression method. As the name suggests, the input file is divided in blocks, each of which is individually compressed. Block-based compression has the advantage that in the case of data corruption, only the data confined within a block is lost. Nonetheless, this technique hinders the compression performance, since redundancies between blocks cannot be identified and removed. The block sizes in gzip are arbitrary, with an upper limit of 65535 bytes of the input file. The compressor is free to choose the size of a block as it is compressed, or start a new block, if it "believes" that it will improve the compression.

The heart of Gzip is the preprocessing step which uses the LZ77 algorithm [49], an adaptive-dictionary-based technique [40], which is used as a pre-processing step. The purpose of this step is to identify recurring patterns in the data, and eliminate redundant repetitions. This is achieved by finding matches of identical strings of symbols within a

given block. Afterwards, these strings, apart from the first occurrence, are converted to pointers in the form of a (*length*, *distance*) pair, containing the length of the matching string and the distance from the previous occurrence. Especially in DEFLATE, the length can be from 3 bytes up to 258, and the distance from 1 byte up to 32768 (meaning that a repetition may overlap with itself). A very simplistic example of this conversion is the following:

	Previously compressed	To be compressed
Input:	$daadbbc\overline{abraca}$	$d\underline{abr\bar{a}r\bar{r}\bar{a}rra}d$
Output:		$d(7,4)r(5,3)d$

Following the conversion, the alphabet of the 255 ASCII character symbols and the alphabet of the length symbols are combined to form an alphabet of size 286. Values 0 – 255 are are the characters, 256 is the end-of-block symbol and the rest 29 indices represent the lengths, by utilizing extra bits. Another alphabet is created for the distance symbols. The modelling step is a simple count of the frequency of occurrence of each symbol in the two alphabets, and based on that, a Huffman code [22] is created for each, to constitute the final entropy coding step.

Gzip gives the liberty to the user to select one of 9 compression levels, to balance the speed/compression ratio trade-off. Level 1 provides the higher speed, but poor compression performance, while level 9 gives the highest ratio at the expense of speed.

2.1.2 bzip2

The bzip2 file compressor [41] is another popular general-compression technique. It is an open-source program, developed and distributed by Julian Seward [42].

Same as gzip, bzip2 uses block-based compression, with block sizes in the range of 100 to 900 kB. However, while gzip attempts to discover repeated strings of characters, bzip2 takes advantage of repeats of single characters, as will be shown in this section.

It's strong point is its preprocessing step, which performs a number of transformations on the data of each input block.

The first transformation is *run-length encoding*. It is a very simple, but highly effective process, which transforms runs of identical characters to just the character and the length of the run, thus representing a number of bytes with only 2 values. In the case of bzip2, sequences of 4 to 255 duplicate symbols are transformed to a sequence of 4 symbols and the accompanying run length minus 4. For example, the sequence `AAAAAAABBBBCCCD` is transformed to `AAAA\3BBBB\0CCCD`.

Afterwards, the *Burrows-Wheeler transformation* (BWT) [7] is performed. This transformation consists the core of bzip2, since it is a very effective lossless compression method [1]. Interestingly, it does not have any effect on the size of the data whatsoever. Instead, it rearranges the symbols in a way that identical symbols are grouped together, forming runs of duplicate characters and thus strengthening their statistical properties. More importantly, it does so without the need to store additional data.

The transform is performed by sorting all rotations of a string in lexicographical order. The output of the process consists of the last character of each rotation in the

sorted order. For example, the string `aardvark`, terminated by the special character $, is permuted in the string `k$avrraad`, as shown in Table 2.1. As long as the terminating character is included in the string, the transformation is reversible [7].

The benefits of the transformation are clear in this example. The initial string had only one run larger than 1 character, while the transformed one has 2. This may not seem as a big improvement, since the input string a short one, but the average run lengths increase as the size of the input string increases.

Rotations	Sorted Rotations
aardvark$	$aardvark
ardvark$a	aardvark$
rdvark$aa	ardvark$a
dvark$aar	ark$aardv
vark$aard	dvark$aar
ark$aardv	k$aardvar
rk$aardva	rdvark$aa
k$aardvar	rk$aardva
$aardvark	vark$aard

Table 2.1: The string aardvark$ is permuted to k$avrraad by performing the BWT .

In order to reap the benefits of the BWT, the next step of preprocessing is the *Move-To-Front* (MTF) [4, 39] transformation. This is another simple transformation, which replaces every symbol by the number of different symbols which have appeared since its last occurrence in the data stream. For example, the string `aaaabbcbcabbb` is transformed to `0000102112200`, assuming that `cba` preceded it. When the MTF transformation is performed after the BWT, it manages to skew the frequencies of occurrence of the symbols, thus enabling better compression by entropy encoders. Even in the previous example, the highest frequency of occurrence in the initial string is 5 for the symbol `a`, while the frequency of `0` in the result is 8.

After the MTF, another round of run-length encoding takes place, to conclude the preprocessing step. Finally, modelling similar to gzip's is performed, where the frequency of occurrence of each symbol is measured and Huffman codes for each block are created for the entropy coding step.

2.2 Domain-specific compression techniques

Domain-specific compression techniques for FASTQ files can be categorized in two main groups: *Non-reference based* and *reference-based techniques*. The difference lies in the fact that reference-based compressors are preceded by mapping assembly with some reference genome. Therefore, they only need to compress the position of a sequence in the reference and any difference between them, instead of the whole sequence. This way they manage to achieve superior compression to non-reference based techniques.

However, this category has a number of disadvantages. Firstly, the data cannot be

compressed until the mapping is performed. Therefore, data from experiments cannot be directly compressed after sequencing, and have to stored uncompressed for a period of time. Moreover, an appropriate reference sequence database may not always be available, as in the case of metagenomic sequencing − sequencing of genetic material recovered directly from environmental samples, as opposed to cultivated clonal samples. Last but not least, the compressed files of reference-based approaches are not self-contained. Their decompression requires precisely the same reference database used for compression, and in the case that this reference is not available, the initial data cannot be recovered.

Given all the above, we opt to focus on non-reference based compression in this thesis. In this section, we present some prevalent examples of such compression-techniques, and we briefly mention some reference-based ones for completeness.

2.2.1 SAMtools

SAMtools [27] is a software package for processing NGS data, initially developped by Heng Li. It contains software and APIs for sorting and aligning reads in order to perform mapping assembly, which use their own compressed data format.

Instead of FASTQ, SAMtools use their own *Sequence Alignment/Map* (SAM) file format for NGS data. This format stores the same information as FASTQ for each read and additional information on the alignment of each read to a reference [46].

A SAM file contains *header lines* and *read lines*. Header lines start with the character @, and contain information on the file in the form TAG:VALUE. This information can either concern the whole file, e.g., the format version used, or a group of reads, e.g., the reference genome used for their alignment. As the name suggests, read lines contain the actual reads, with 11 tab-delimited fields of information. These are in order of appearance:

- **QNAME:** Query template name of the read: Essentially the name of a group of reads that originate from the same sequence.
- **FLAG:** Bitwise flag that denotes different types of reads.
- **RNAME:** Name of the reference sequence of the alignment.
- **POS:** Position of the first matching base for the alignment.
- **MAPQ:** Mapping quality.
- **CIGAR:** String that encodes the alignment of the read sequence with the reference.
- **RNEXT:** Reference sequence name of the next read in the DNA sequence.
- **PNEXT:** Position of the alignment of the next read in the DNA sequence.
- **TLEN:** Observed length of the assembled sequence.
- **SEQ:** Sequence of nucleotide bases of the read, same as the sequence string in the FASTQ format.

- **QUAL:** Quality string, same as the quality string in the FASTQ format.

In the end of each real line, optional information in the form TAG:VALUE can be added. All of the above fields have reserved characters or values for the absence of the respective piece of data. Therefore, a FASTQ file can be converted to a SAM file, by An example of 2 reads in a SAM file is shown in Fig. 2.1.

```
@HD VN:1.5 SO:coordinate
@SQ SN:ref LN:45
r001  163  ref  7  30  8M2I4M1D3M  =  37  39  TTAGATAAAGGATACTG  *
r002  0    ref  9  30  3S6M1P1I4M  *  0   0   AAAAGATAAGGATA     *
```

Figure 2.1: The SAM file format [46].

SAM files are compressed by being converted to their binary representation, called the *BAM file fomat*. This format utilizes the BGZF compression format, which is a block compression technique implemented on top of the standard gzip file format [46], which also uses DEFLATE-compliant compression. As such, a BAM file can be decompressed by any 'gunzip compatible' program, making it a popular choice among researchers.

The innovation of BGZF over regular gzip compression, is the fact that it can be indexed, allowing efficient random access to the reads within the file. In order to achieve this, a BGZF file contains a series of concatenated BGZF blocks, each being a gzip-compliant file by itself. The gzip file format, allows for the inclusion of extra application-specific fields on the file header, which are used by the BGZF format to hold the file offset of each block.

The SAM/BAM format is probably the most popular compression scheme among geneticists at the moment. The unique ability of BAM files to access specific reads, without the need for the decompression of the whole file, allows for the creation of downstream applications that can work entirely on this format. However, the compression ratio of this approach cannot exceed the ratio of gzip. Despite being a domain-specific compression technique, BGZF does not take advantage of the unique statistical properties of the data included in the file, which would lead to better compression performance. Moreover, extra effort is needed to create the index, which in turn has the prerequisite that the reads are sorted in the order of their alignment to the reference. On top of that FASTQ files cannot be directly compressed, but have to be converted to SAM before being compressed to BAM.

2.2.2 Quip

Unlike the SAM/BAM compression method, *Quip* [24] is a domain-specific compression algorithm that uses the statistical properties of NGS data to achieve high compression rates. It incorporates three basic modes of operation:

- Regular non-reference based compression
- Reference-based compression

- Non-reference based compression aided by de-novo assembly

In all three modes, different compression techniques are used for the read's identifier string, sequence and quality scores, each being suitable for the type of data in the field.

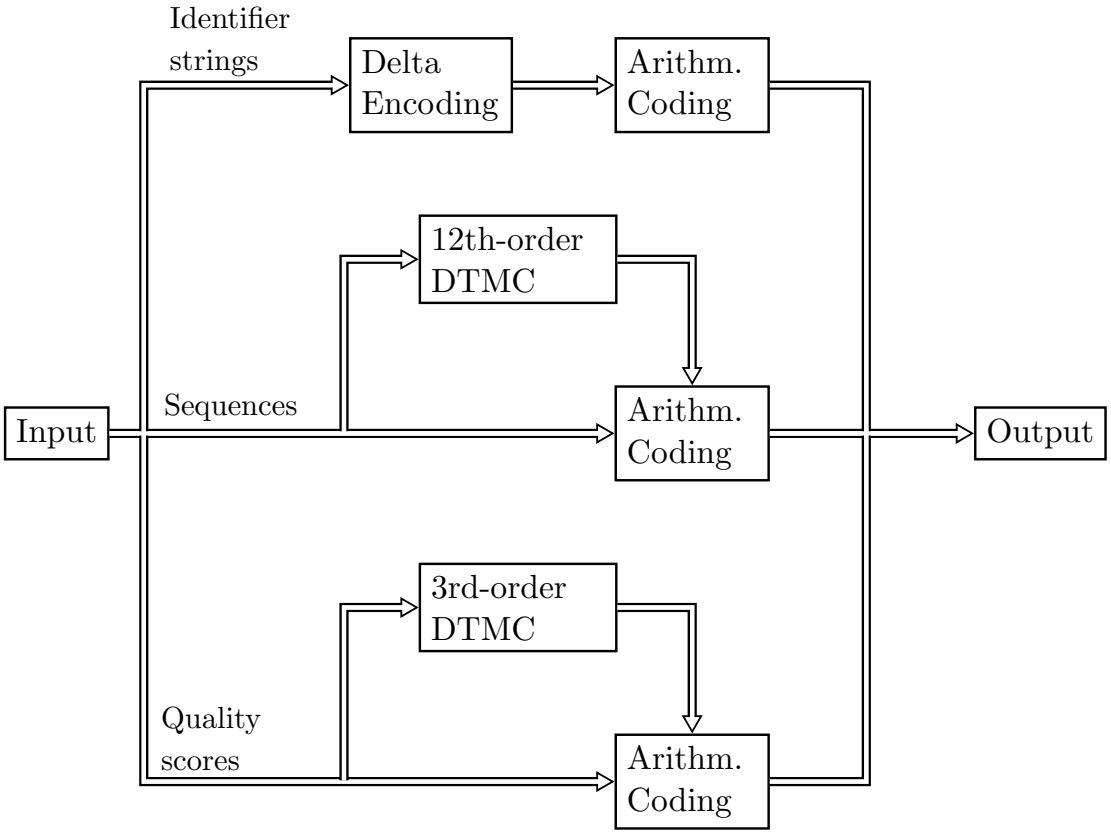

Figure 2.2: Structural blocks of Quip's non-reference based compression.

When regular non-reference based compression mode, Quip performs minimal preprocessing, exclusively on the identifier strings. Noting that these strings have large portions of characters identical from read to read, the developers chose to perform a form of *delta encoding* as the preprocessing step on them.

In delta encoding, instead of compressing a complete sequence, only the differences from a previous sequence are encoded. If these two sequences are similar, then the difference data are only a portion of the original data, therefore some reduction has already taken place. In the same manner, Quip parses and tokenizes every identifier string in separate fields, and compares them to the identifier of the previous read. Tokens that happen to be identical with the previous read can be compressed to a negligible size. Non-identical tokens on the other hand, are divided in a prefix identical to the previous read, which is compressed as previously, and the remaining suffix which is encoded as is. The developers do not provide any information on what kind of modelling is used for the identifiers, but arithmetic coding is used as the final entropy coder.

The sequence of bases on the other hand does not get preprocessed. Instead, Quip uses a 12th-order adaptive DTMC to model the sequences, which means that a nucleotide is predicted based on the 12 previous bases that have been compressed. The prediction, in the form of probabilities for each symbol, is then used by an arithmetic coding step for the final compression.

The same compression technique is used for quality scores as well: no preprocessing, adaptive DTMC modelling and arithmetic coding. However, since the size of the alphabet for quality scores is much larger than the one of bases (41 instead of 4), a 3rd-order DTMC is used instead.

The other two modes of operation enhance the compression of the base sequences. In the second one, that of reference-based compression, Quip accepts SAM or BAM files with pre-aligned reads, and the reference to which they are aligned. Using the alignment information, only the position of the read's sequence on the reference, and any deviation from it needs to be compressed, instead of the whole sequence. However, being reference-based compression, this mode of operation has all the advantages and disadvantages as presented in the beginning of the section.

In the final mode of operation, Quip attempts to eliminate the disadvantages of reference-based compression. Thus, instead of using an external reference to compress the data, it creates a reference for the input file, by performing de-novo assembly on the first 2.5 million reads of the file. This reference is then used for reference-based compression. The assembly process can be mimicked during decompression, and as such, it does not have to be contained in the compressed file, rendering it entirely self-contained and eliminating the need for external references.

Combining both fast compression and high ratios, Quip outperforms other compression techniques and is considered state-of-the-art at the moment.

2.3 Comparison of FASTQ compression techniques

In order to evaluate the compression techniques presented in this chapter, the benchmark C11 (Table 1.3) is used[1]. The testing platform was a server with a 4-core Intel(R) Xeon(R) E5420 processor and 32GB of RAM. Figure 2.3 shows the compression ratios achieved, and Fig. 2.4 shows the respective speed in compression and decompression, calculated by measuring the execution time of each program.

From these figures, it is evident that gzip's performance, both in means of compression and execution time, can vary greatly according to the level used. Yet, even on level 9 (gzip -9), which gives the best compression, it lacks in comparison with other techniques. Gzip's decompression speed is noteworthy. It is far higher than the other techniques, thanks to the very simple procedure it follows. BAM, being build on top of gzip, has a similar compression performance, but far lower speeds, due to fact that the conversion from FASTQ to SAM has to precede the actual compression. bzip2 on the other hand, performs impressively for a general-purpose compressor, but it lacks in speed, due to the high number of transformations that take place.

[1]For FASTQ to BAM compression, Picard (http://picard.sourceforge.net/) was used, since SAMtools does not support conversion of unaligned reads.

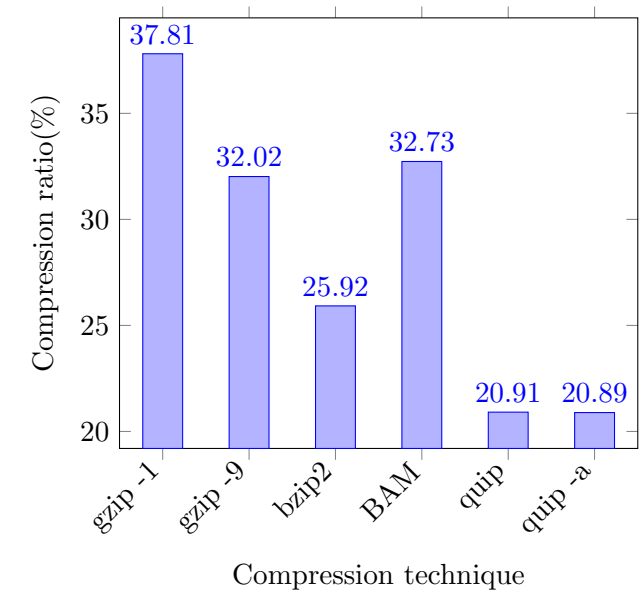

Figure 2.3: Compression ratios of popular FASTQ compression techniques.

Quip is superior to the other alternatives in terms of compression, and at the same time maintains a respectable throughput. However, using the de-novo assembly-aided mode (quip -a) severely hinders the compression speed, with a negligible gain in compression.

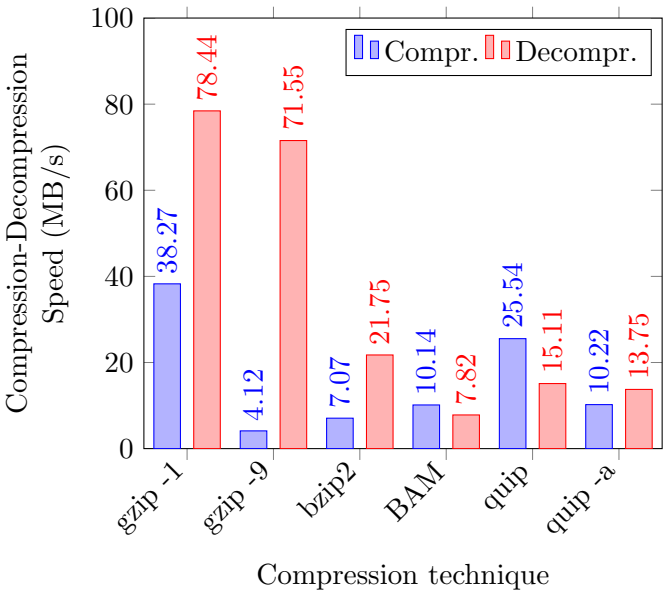

Figure 2.4: Compression and decompression speed of popular FASTQ compression techniques.

2.4 Additional domain-specific techniques

Bonfield et al. [5] present fastqz and fqzcomp, two lossless domain-specific compression programs that also use adaptive modelling and arithmetic coding. Kozanitis et al. [26] proposed reference-based compression with SlimGene, and analysed the effects of lossy compression of the quality scores in downstream applications, concluding that "There are dozens of downstream applications and much work needs to be done to ensure that coarsely quantized quality values will be acceptable for users". Nevertheless, lossy compression techniques have emerged. CRAMtools [16], performs reference-based compression and discards the quality scores of reads that match a reference sequence. SCALCE [20], Qscores-Archiver [48] and more recently HUGO [28] cluster the quality scores, coarsening the alphabet, but losing all other values. Moreover, a noteworthy technique for lossless, non-reference-based compression of only the base sequences is presented in [10]. Finally, [11] presents an overview of a number of domain-specific techniques.

3 Compression of Quality Scores

3.1 Introduction

As mentioned in previous chapters, a FASTQ file contains three types of data: the identifier strings, the base sequences and the sequences of quality scores. General compression methods that do not distinguish among the three of them lack in compression performance. Therefore, in order to achieve high compression rates, different techniques have to be used for each type of data. These techniques have to be designed to suit their respective data type, so that their statistical correlations are uncovered, and used to an advantage.

Out of the three types, the quality scores have the highest information content. This statement has already been mentioned in related bibliography [24, 26, 16], but it is relatively straightforward for the following reasons.

For identifiers, in the vast majority of reads, only one or two characters change compared to the previous read. This means that the rest of the information of the identifier is redundant, enabling very high compression ratios.

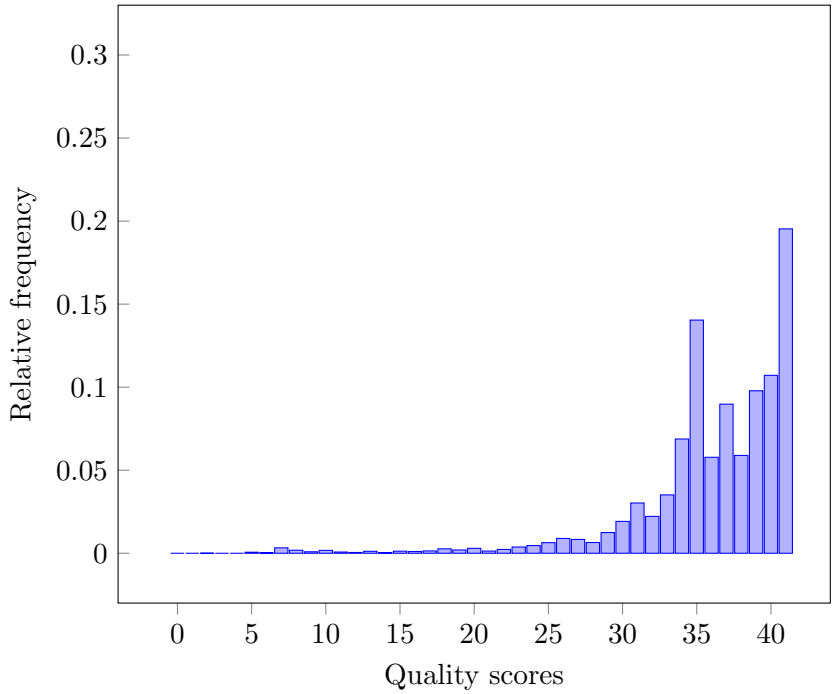

Figure 3.1: Relative frequency distribution of the quality scores in C11.

Nucleotide base sequences contain less redundant information than identifiers. Most

of the redundancy in this type of data comes from the fact that the reads overlap (Fig. 1.3), although it is more difficult to uncover this redundancy without performing mapping assembly first. In FASTQ files specifically, the fact that ASCII characters are used to represent the 5 symbols of these sequences (A,C,T,G,N) is itself a waste of storage space. ASCII characters use 8 bits per symbol, whereas $\lceil log_2(5) \rceil = 3$ bits per symbol is enough to encode the whole alphabet. Taking into consideration the fact that N is highly improbable, an average of close to 2 bits per symbol can be very easily achieved for nucleotides, which by itself presents a compression ratio of 25%. More on this will be discussed in Chapter 4.

Quality scores on the other hand, have a much larger alphabet, with a size of 42 symbols. Inherently, that means that the simplest binary encoding needs a maximum of $\lceil log_2(42) \rceil = 6$ bits per symbol. Measuring the frequency of occurrence of each quality score value in benchmark C11 (Table 1.2) reveals a skewed distribution (shown in Fig. 3.1), with over 90% of the scores being above 30. The calculation of the entropy on this type of data shows that at least 3.74 bits per symbol are needed, if their source is considered an iid one.

In this chapter, we present our attempts to compress the sequences of quality scores in FASTQ files. In Section 3.2, linear predictive coding is used as a preprocessing step, along with adaptive DTMC modelling and arithmetic coding. In Section 3.3 an attempt in more complex modelling is performed, by using Hidden Markov Models to model the source of the quality scores, aspiring to enhance the compression achieved by arithmetic coding.

3.2 Linear Predictive Coding

Kozanitis et al. [26] and Jones et al. [24] note that a quality score is highly correlated with the quality scores in preceding positions. This means that the value of a quality score at position n is most probably close to the values in positions $n-1, n-2, ...$ This property shows a data redundancy that can possibly be eliminated for the sake of compression. In order to do so, a preprocessing step of *Linear Predictive Coding* (LPC) is used.

Predictive coding is a mathematical process, in which the value of an entity is predicted based on previously observed values. As a notion, it has been used extensively in video compression [35] and audio signal modelling [33].

Since the compression process is performed on a sequence $S = s_1 s_2 ... s_N$ of input symbols, the compressor can create a prediction $\hat{s}_i$ for the symbol s_i based on the symbols $\{s_j : j < i\}$. Moreover, the decompression of data symbol s_i is performed after the decompression of all data symbols $\{s_j : j < i\}$. Therefore, the decompressor is able to create the same prediction for the data symbol s_i that the compressor did, if provided with the details of the prediction technique.

Obviously, the prediction will not always be accurate. Therefore, an error e_i between the actual value of d_i and the predicted value will exist. In the case of lossless compression, only this error need be stored, in order to be able to reconstruct the initial data sequence, without the loss of information.

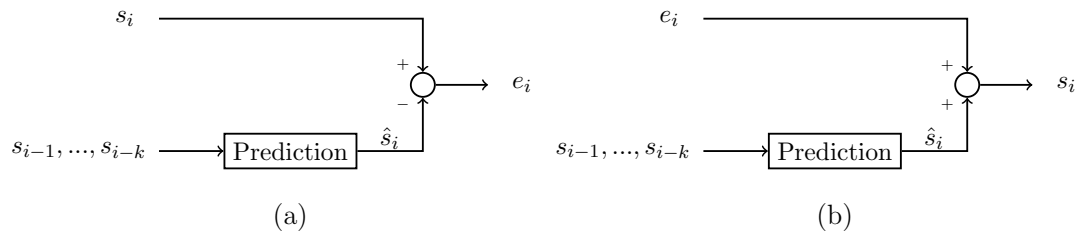

Figure 3.2: The predictive coding transformation: (a) During compression, (b) During decompression.

The described process defines a transformation from the input sequence S to the sequence of error values $S_e = e_1 e_2 ... e_N$. This transformation can be used for compression by observing that if the prediction is sufficiently accurate, the error values will most frequently be small. This in turn means that the error sequence takes most of its values from a small subset of the whole alphabet, which is the driving force behind entropy coding. Hence, a good prediction scheme can skew the probabilities of occurrence of the symbols to create a sequence more suitable for compression.

3.2.1 Linear Prediction

In the case of quality values, *linear prediction* [45] can be used for the predictive coding step. As the name suggests, in this kind of prediction, the value of a quality value is predicted as a linear combination of its predecessor values.

A sequence of quality scores, containing the N values corresponding to the nucleotide bases of a read can be seen as a discrete function :

$$q : \{1, ..., N\} \rightarrow \{0, ..., 41\} \tag{3.1}$$

A prediction of $q(n)$ can be created by the previous p quality values, using *p-order forward linear prediction* [1]. The prediction of $q(n)$ is denoted as $\hat{q}(n)$ and is calculated as follows:

$$\hat{q}(n) = \sum_{k=1}^{p} a_k \cdot q(n-k) \tag{3.2}$$

where $\mathbf{a} = [a_1, a_2, ..., a_p]$ is the vector of *weighting coefficients* of the linear predictor, and p is the order of prediction, or in other words, how many previous values are used for the prediction. For completeness we consider $q(n) = q(1),\ \forall n < 1$.

The *error of the prediction* is defined as the difference between the predicted value and the actual value of the quality scores:

$$e(n) = q(n) - \hat{q}(n) = q(n) - \sum_{k=1}^{p} a_k \cdot q(n-k) \tag{3.3}$$

[1] A forward-backwards linear predictor can also be defined, when the prediction is created from both previous and later values. However, in our case such a predictor is of no practical value, since the decompressor would not be able to know the following values and create the prediction.

Now lies the problem on how to create an optimal predictor, which minimizes the values of the error. It is obvious that the accuracy of the prediction depends on the weighted coefficients **a** of the predictor.

We define the total squared prediction error of the prediction:

$$E = \sum_{n=1}^{N} [e(n)]^2 = \sum_{n=1}^{N} [q(n) - \hat{q}(n)]^2 = \sum_{n=1}^{N} \left([q(n)]^2 - 2 \cdot q(n) \cdot \hat{q}(n) + [q(n)]^2 \right) \tag{3.4}$$

In signal processing terms, Eq. 3.4 gives a value indicative of the energy of the error signal. Obviously, it is desirable to choose the predictor coefficients so that the value of E is minimized. The optimal minimizing values can be determined through differential calculus, i.e. by calculating the partial derivative of E with respect to each coefficient and setting that value equal to zero:

$$\frac{\partial E}{\partial a_k} = 0, \ \forall 1 \leq k \leq p$$

$$\Rightarrow \frac{\partial}{\partial a_k} \left(\sum_{n=1}^{N} \left([q(n)]^2 - 2 \cdot q(n) \cdot \hat{q}(n) + [q(n)]^2 \right) \right) = 0$$

$$-2 \sum_{n=1}^{N} \left(q(n) \cdot \frac{\partial}{\partial a_k} \hat{q}(n) \right) + 2 \sum_{n=1}^{N} \left(\hat{q}(n) \cdot \frac{\partial}{\partial a_k} \hat{q}(n) \right) = 0$$

$$\sum_{n=1}^{N} \left(q(n) \cdot \frac{\partial}{\partial a_k} \hat{q}(n) \right) = \sum_{n=1}^{N} \left(\hat{q}(n) \cdot \frac{\partial}{\partial a_k} \hat{q}(n) \right) \tag{3.5}$$

From Eq. 3.2 follows that:

$$\frac{\partial}{\partial a_k} \hat{q}(n) = q(n-k) \tag{3.6}$$

Thus, Eq. 3.5 becomes:

$$\sum_{n=1}^{N} (q(n) \cdot q(n-k)) = \sum_{n=1}^{N} (\hat{q}(n) \cdot q(n-k)) \tag{3.7}$$

Using Eq. 3.2 again, we get:

$$\sum_{n=1}^{N} (q(n) \cdot q(n-k)) = \sum_{n=1}^{N} \left(\sum_{i=1}^{p} (a_i \cdot q(n-i)) \cdot q(n-k) \right)$$

$$\sum_{n=1}^{N} (q(n) \cdot q(n-k)) = \sum_{i=1}^{p} \left(a_i \cdot \sum_{n=1}^{N} q(n-i) \cdot q(n-k) \right) \tag{3.8}$$

For the sake of brevity, the correlation function ϕ is defined:

$$\phi(i,k) = \sum_{n=1}^{N} q(n-i) \cdot q(n-k) \tag{3.9}$$

Thus, Eq. 3.8 becomes:

$$\phi(0,k) = \sum_{i=1}^{p} a_i \cdot \phi(i,k),\ k \in [1,p] \tag{3.10}$$

Equation 3.16 defines a system of p linear equations, with p unknowns: a_1 through a_p. These are called the *normal equations* [45] of the predictor, and their solution is the vector of coefficients that minimize the prediction error.

Given all the above, the linear predictive coding step of the quality scores compressor performs the following steps:

1. Solves the system of normal equations.
2. Stores the coefficients in the compressed file for the decompressor to use.
3. Performs the prediction using the calculated coefficients.
4. Transforms the sequence of quality scores to the sequence of prediction errors and compresses them.

Accordingly, the decompressor performs the following steps:

1. Reads the prediction coefficients from the compressed file.
2. Decompresses the sequence of prediction errors.
3. Calculates the prediction for each quality value and adds the error to recreate the actual value.

The benefits of LPC can be better understood with an example. Using benchmark N21 (Table 1.2), 1st-order LPC was performed on each read's quality score sequence separately. The relative frequency of each quality score before the LPC is shown in Fig. 3.3. The highest symbol frequency is 26.8% for the quality value 41, and the entropy is 3.44 bits per symbol.

After 1st-order LPC, the relative frequencies of the error values are the ones shown in Fig. 3.4. It is obvious that the distribution is much more skewed now, with the highest relative frequency being 52.6% and the entropy being 2.94 bits per symbol: a reduction[2] ratio of 1.17 in entropy. This reduction is expected, as information is 'removed' from the sequence during the transformation and moved to the coefficients.

Increasing the order of prediction, increases this reduction in entropy, as shown in Fig. 3.5. This happens because the prediction becomes more accurate as the order increases, and as consequence, over 90% of the error values are 0 for 50th-order LPC, which has an entropy of 0.61 bits per symbol.

Nevertheless, two problems arise from LPC. The first is the fact that the alphabet of all possible error values is fundamentally twice in size the alphabet of all possible quality scores. When the scores vary in the range $[0, 41]$, the error assumes values in the range $[-41, 41]$. Yet, this is not a concern when the prediction is accurate, as the frequency

[2]Entropy reduction is defined as the quotient of the initial entropy over the reduced entropy.

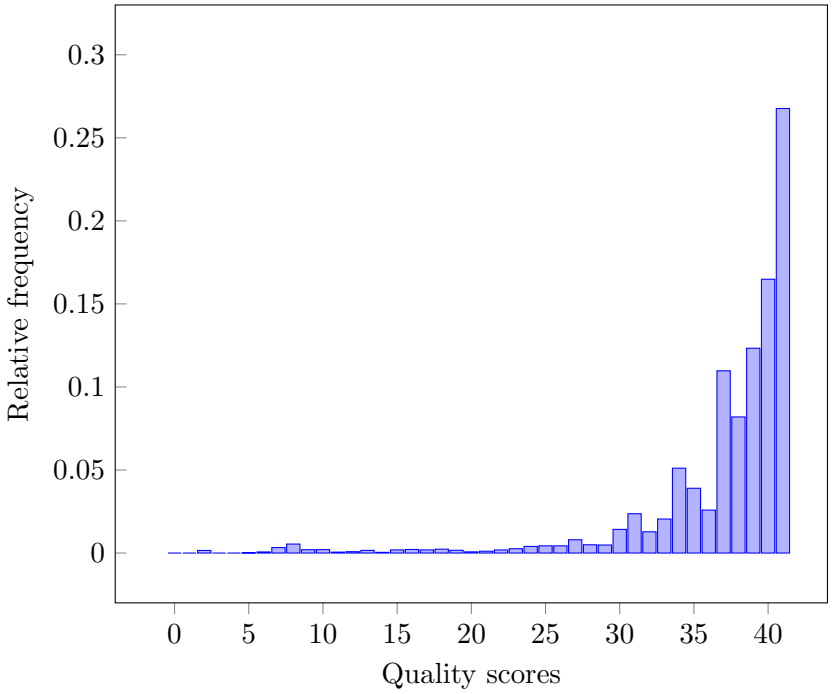

Figure 3.3: Relative frequency distribution of the quality scores in N21.

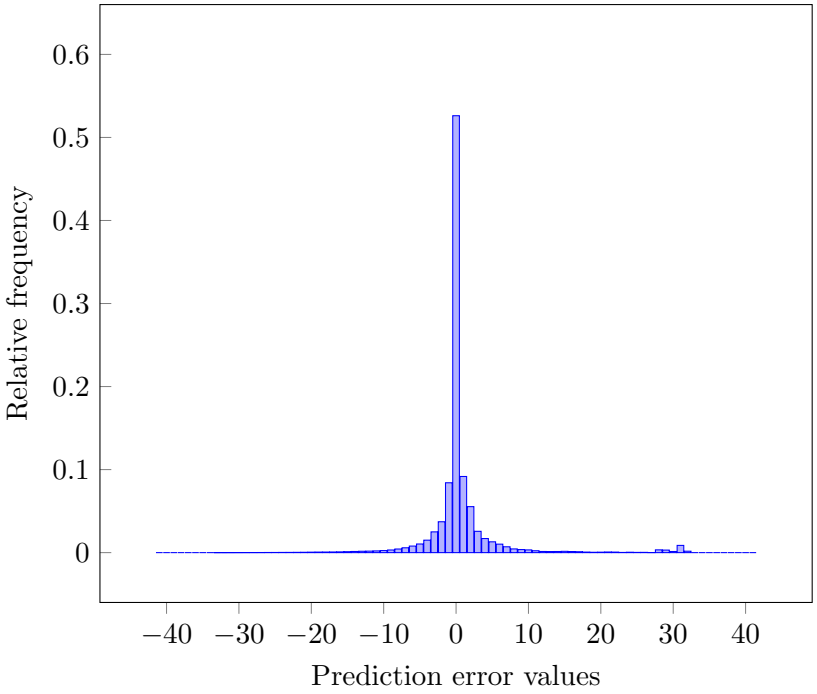

Figure 3.4: Relative frequency distribution of the 1st-order LPC error values in N21.

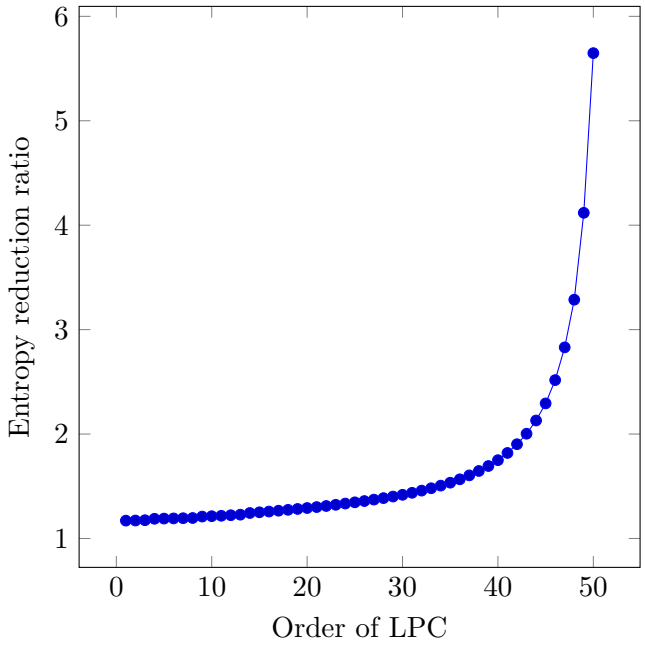

Figure 3.5: Entropy reduction between the quality score sequences and LPC error values in N21, as a function of the order of prediction.

distribution of the symbols will be skewed resulting in lower information content, as shown in the above example. The main consequence is the fact that a larger alphabet requires a larger model during the modelling step.

The second problem is the fact that the coefficients are real numbers, and as such originate from a huge alphabet. This means that they are not easily compressable, and in fact they would be harder to compress than the original quality scores. Therefore, the practise of creating separate optimal coefficients for each read that was described above in not usable in practise. A solution for that problem is described in the next section.

3.2.2 Blocked LPC

Unfortunately, LPC on each read's quality score sequence separately is not an option, as there would be significant overhead to store the vector of coefficients for each of them. Therefore, the linear prediction scheme must be altered, so that a single vector of coefficients is used to predict the values of more than one read. Of course, with this alteration to the prediction scheme, the normal equations for the calculation of the coefficients will also be altered.

The set of reads in the input file is divided in blocks, with each one containing M reads, each with a quality sequence of size N. For each block, its own vector of coefficients $\mathbf{a} = [a_1, a_2, ..., a_p]$ needs be calculated.

The sequence of quality values for each read in a block is denoted as $q_m,\ m \in [1, M]$. In contrast to regular LPC, $q_m(n) = c$ for zero or negative values of n, where c is the arithmetic mean of all $q_m(1)$ quality values. From all the above, it follows that the

p-order prediction will be calculated as:

$$\hat{q}_m(n) = \sum_{k=1}^{p} a_k \cdot q_m(n-k),\ \forall m \in [1, M],\ \forall n \in [1, N] \tag{3.11}$$

and the corresponding prediction error:

$$e_m(n) = q_m(n) - \hat{q}_m(n),\ \forall m \in [1, M],\ \forall n \in [1, N] \tag{3.12}$$

The total error that needs to be minimized now is the following:

$$E_{bl} = \sum_{m=1}^{M} \left[\sum_{n=1}^{N} [e_m(n)]^2 \right] \tag{3.13}$$

which is the sum of the errors of prediction of all the reads in the block.
Minimizing the error E_{bl} is achieved as before:

$$\frac{\partial E_{bl}}{\partial a_k} = 0,\ \forall 1 \leq k \leq p \tag{3.14}$$

Following the same train of thought as before, we define the correlation function for each read:

$$\phi_m(i, k) = \sum_{n=1}^{N} [q_m(n-i) \cdot q_m(n-k)] \tag{3.15}$$

The normal equations of the predictor now become:

$$\sum_{m=1}^{M} \phi_m(0, k) = \sum_{i=1}^{p} \left[a_i \cdot \sum_{m=1}^{M} [\phi_m(i, k)] \right],\ k \in [1, p] \tag{3.16}$$

We define:

$$\phi_{bl}(i, k) = \sum_{m=1}^{M} [\phi_m(i, k)] \tag{3.17}$$

Therefore, the normal equations have a similar form as previously:

$$\phi_{bl}(0, k) = \sum_{i=1}^{p} a_i \cdot \phi_{bl}(i, k),\ k \in [1, p] \tag{3.18}$$

The steps that the compressor and decompressor have to take using the blocked LPC are the same as for regular LPC, with the only difference that the normal equations (Eq. 3.18) are solved once for each block.

Naturally, the reduction in the entropy of the quality scores will be less in blocked LPC, compared to calculating a different coefficients' vector per read in regular LPC. When the coefficients are fitted to one quality sequence only, they optimally reflect the sequence's properties. On the opposite case, the coefficients are created to reflect the properties of a whole block of reads, and as a result, they do not optimally specify the properties of each individual read in the block. The resulting entropy reduction when

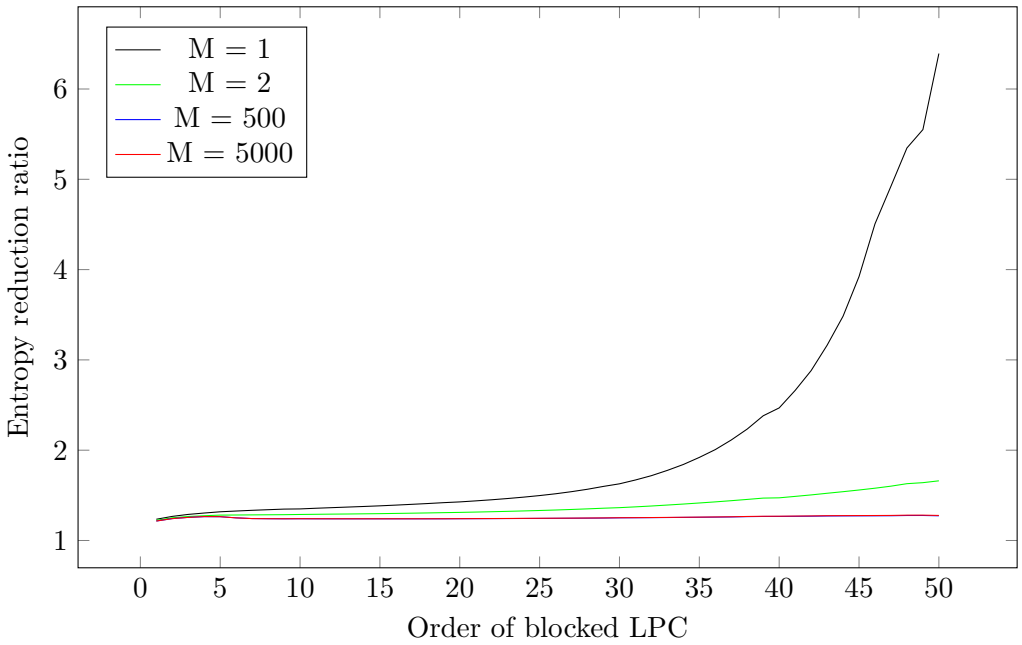

Figure 3.6: Entropy reduction between the quality score sequences and blocked LPC error values in N21, as a function of the order of prediction, for different block sizes.

using different block sizes in blocked LPC of the quality scores of benchmark N21 (Table 1.2) is shown in Fig. 3.6.

A block with only a single read (M=1) behaves exactly as regular LPC. Yet, as the block size increases, the effect of blocked LPC is apparent: With only 2 reads per block, the entropy reduction's scales much slower with the entropy reduction. However, in order to eliminate the overhead of storing the coefficients on the compressed file, the block needs to be as large as possible. Further increasing the block size, reveals that with this prediction method, the entropy reduction does not scale with the increasing order of the prediction. On the contrary, the entropy reduction stays close to 1.3 for higher block sizes.

Further increasing the block size is desirable, as long as it does not affect the entropy reduction, since the overhead of storing the coefficients becomes negligible. Figure 3.7 shows the entropy reduction in the conversion from quality scores to error values, achieved with larger blocks, for 4th order LPC on the same benchmark as before. Apparently, the difference between them in terms of entropy reduction is very small, although higher sizes perform better. This is due to the fact that the system of normal equations of a larger block tends to be more robust towards a minority of reads that are potentially divergent in terms of statistical properties than the rest. Therefore, we have the freedom to choose the block size among the higher values. However, care should be taken with very large blocks, as there is the potential hazard of overflow in solving the normal equations for files with large reads sizes.

An entropy reduction of 1.26 times may seem trivial in comparison to the respective numbers that are seen in regular LPC. Nevertheless, blocked LPC transforms the qual-

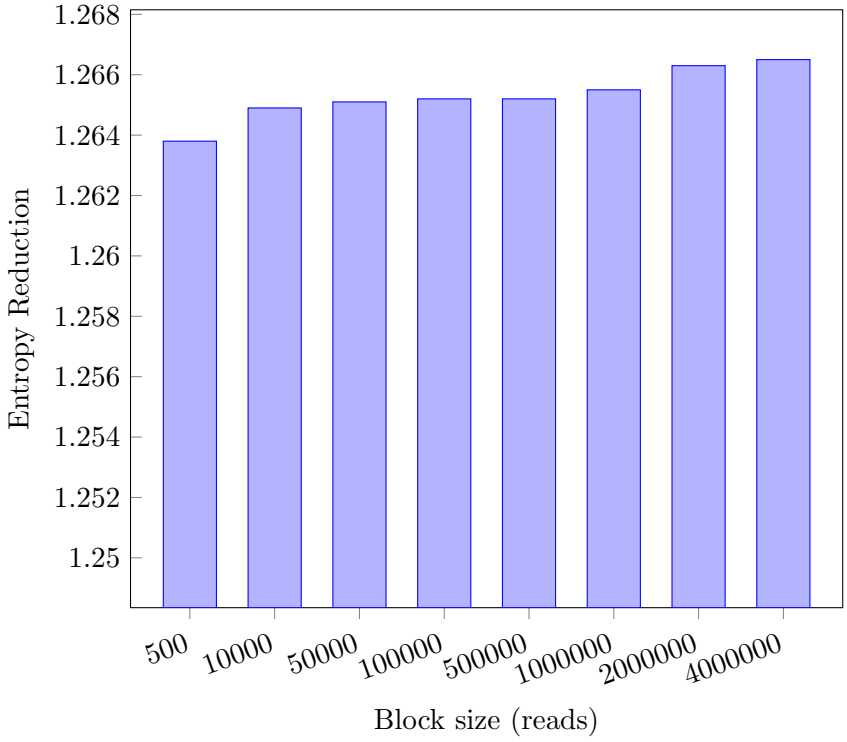

Figure 3.7: Entropy reduction between quality scores and prediction error values with 4th-order blocked LPC for large sized blocks in N21.

ity scores' sequences in the more highly compressible error sequences, with negligible overhead and therefore it can be effectively used as a preprocessing step.

3.2.3 Modelling and Coding

In order to complete the compression process, the blocked LPC preprocessing step is followed by a modelling step using adaptive DTMCs, and arithmetic coding as the final entropy coder.

The modelling step is carried out by a set of adaptive DTMCs. Each one of them corresponds to the context of modelled quality scores. The different contexts depend on the position of an error symbol in its sequence, and by the number of high jumps in error values[3] that have been observed at a particular read. The former enables the compressor to have a different model for each part of a sequence. This is needed because the quality scores depend on their position in the read, and this property is passed to the prediction error during LPC. The later enables the compressor to keep different models for sequences with highly variable error values.

Each individual DTMC has an order of l, meaning that it measures the frequency of occurrence of any symbol of the error value alphabet $\mathcal{U}_{er} = \{-41, ..., 41\}$, given the

[3]A high jump is defined as a difference larger than 1 in two consecutive error values.

previous l values. More precisely, they are represented by the transition matrix $\mathbf{A}$[4] containing the cumulative frequencies d_k of the algorithms in Appendix A. The adaptive nature of the model is implemented by scaling down by a factor of 2 the measured frequencies of a state, every time one of the surpasses a threshold value (N_{thres}).

If we define N_{pos} as the number of different context models depending on the position of a symbol and N_{jump} as the number of different context models depending on the number of jumps, the compressor uses $N_{pos} \cdot N_{jump}$ models in total, each containing $|\mathcal{U}|^{l+1} = 83^{l+1}$ parameters. Using 2 bytes to store every parameter, the models need $2 \cdot N_{pos} \cdot N_{jump} \cdot 83^{l+1}$ bytes in memory.

Finally, the arithmetic coder has 32 bits precision for the fixed arithmetic and encodes all sequences of a block in a single codeword.

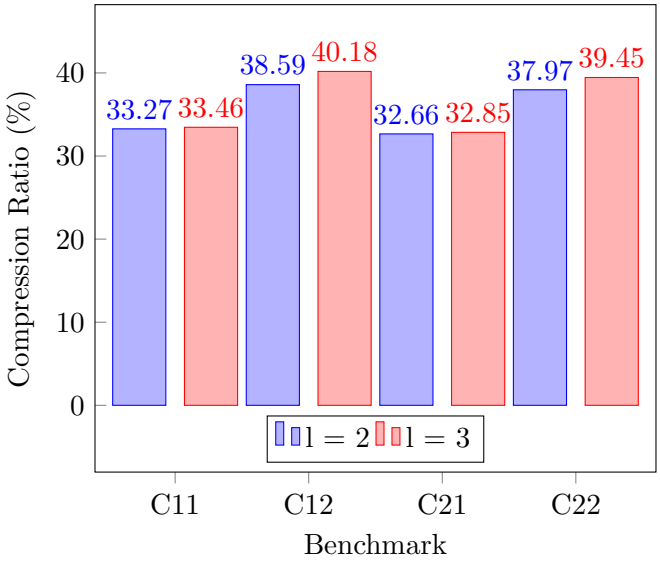

Figure 3.8: Compression ratio on the quality scores with different values on the order of the DTMCs ($\mathbf{N_{pos}} = 5$, $\mathbf{N_{jump}} = 2$, $\mathbf{N_{thres}} = 8192$, $\mathbf{p} = 4$).

3.2.4 Evaluation of Blocked LPC

In this section, the effect of the parameters l, N_{pos}, N_{jump}, N_{thres} and p, as described in the previous section, is shown through experiments. The evaluation criterion will be the compression ratio that is achieved on the quality scores of our benchmarks (Table 1.3). The uncompressed size of this type of data in the FASTQ format is:

$$\text{Number of reads} * (\text{sequence size} * 1 \text{ byte per symbol} + 1)$$

The plus one is added for the newline character in the end of each quality score sequence. The block size M will be 1000000 reads, as it large enough for adequate blocked LPC.

First, the value for the order of the DTMCs l is chosen. The memory needed for the models increases exponentially with the value of l, and as a consequence, only the

[4]See Section 1.2.4.

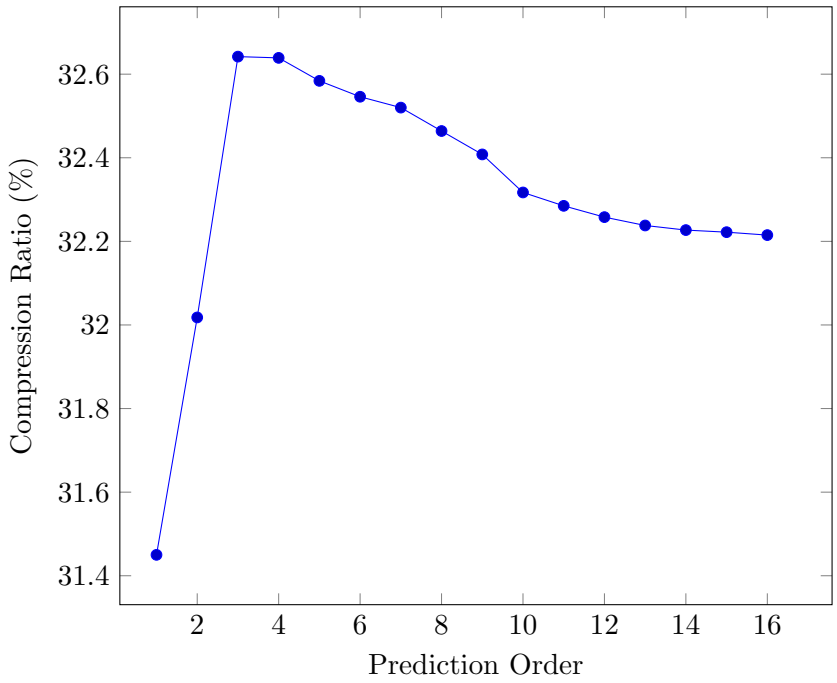

Figure 3.9: Compression ratio on the quality scores of C11 with different values of prediction order ($\mathbf{l} = 2$, $\mathbf{N_{pos}} = 15$, $\mathbf{N_{jump}} = 15$, $\mathbf{N_{thres}} = 8192$).

values 2 and 3 can be considered. Larger models have the advantage of most accurate modelling, and the disadvantage that they need a large amount of input data to be trained. Figure 3.8 shows the compression ratio achieved using both values for l on our benchmarks. The results verify the fact that larger models need more data to be trained. As a consequence, there is a lack in compression performance, up to the point that the models are adequately trained. On larger input files, this problem could be alleviated, however, using 2nd-order DTMCs has the advantage of smaller memory consumption, which allows for higher N_{pos} and N_{jump} values. As a result, the value for l will be set to 2 for the rest of the experiments.

Figure 3.9 shows the compression ratio achieved with a variable order of blocked LPC. As it turns out, in blocked LPC, 1st-order prediction creates the smallest prediction error, which is translated to a better compression ratio.

Figure 3.10 shows the effect of N_{pos} on the compression ratio achieved on C11 (Table 1.2), and Figure 3.11 shows the effect of N_{jump} on the compression ratio achieved on the same benchmark. Same as l, but to a smaller extend, these parameters effect the compression ratio by modifying the number of models that are used. Therefore, appropriate values must be selected to balance the trade-off between the amount of data needed to train the models and their accuracy.

Finally, Fig. 3.12 shows how N_{thres} effects the compression performance. High values of the threshold would make the models unable to react fast enough to changes in the statistical properties of the modelled information. On the other hand, low values of the threshold would make the models disregard information about the past and only depend

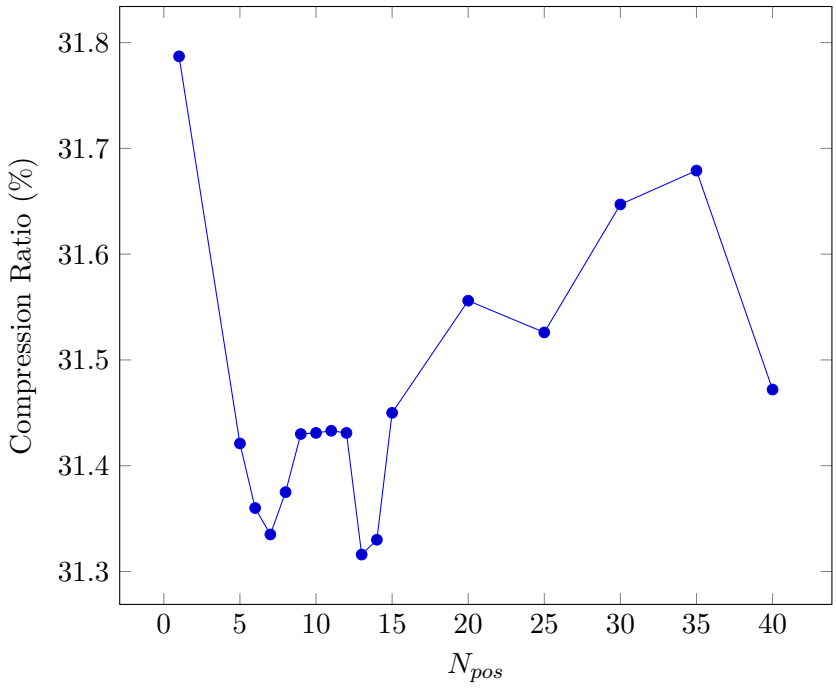

Figure 3.10: Compression ratio on the quality scores of C11 with different values of N_{pos} ($\mathbf{l} = 2$, $\mathbf{N_{jump}} = 15$, $\mathbf{N_{thres}} = 8192$, $\mathbf{p} = 1$).

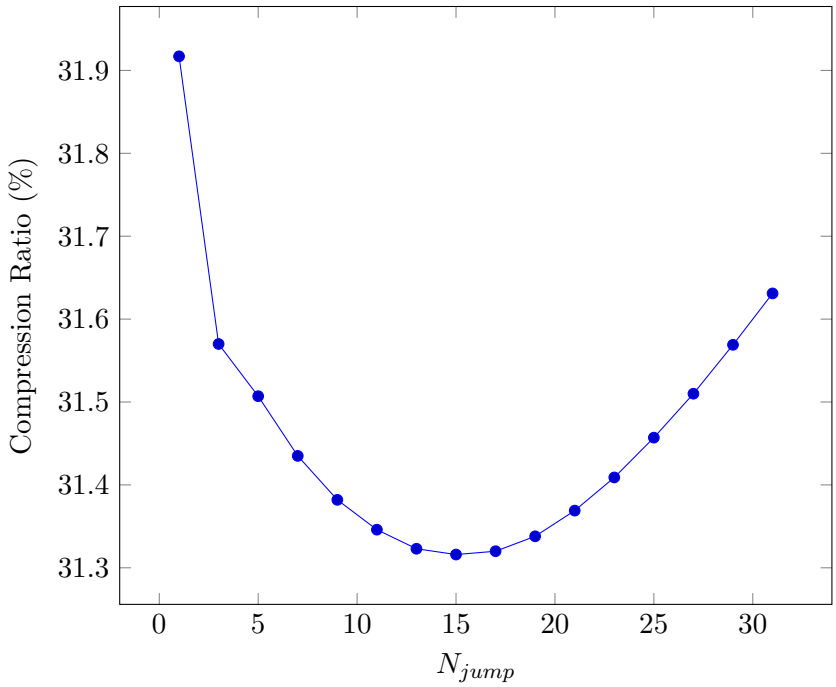

Figure 3.11: Compression ratio on the quality scores of C11 with different values of N_{jump} ($\mathbf{l} = 2$, $\mathbf{N_{pos}} = 13$, $\mathbf{N_{thres}} = 8192$, $\mathbf{p} = 1$).

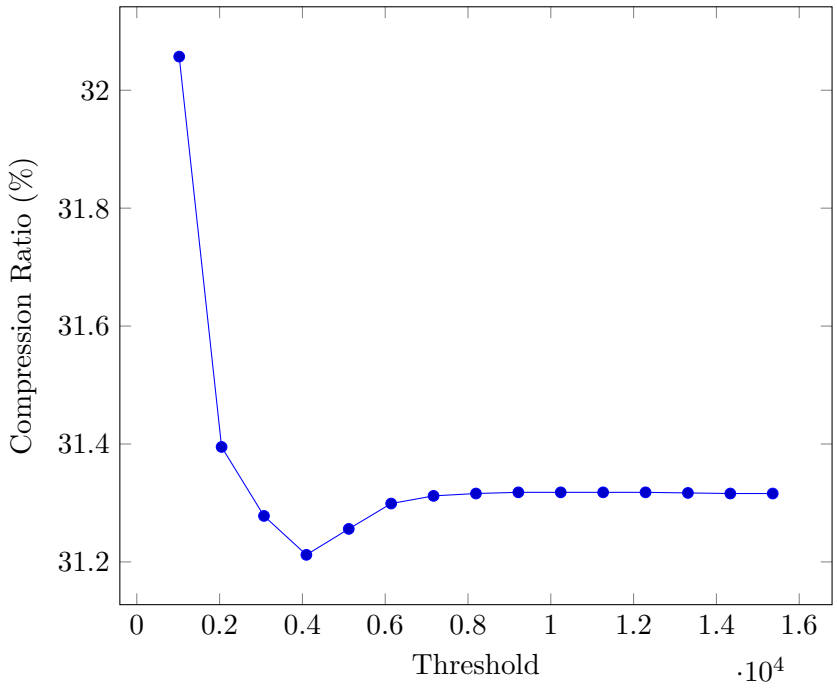

Figure 3.12: Compression ratio on the quality scores of C11 with different values of N_{thres} ($\mathbf{l} = 2$, $\mathbf{N_{pos}} = 13$, $\mathbf{N_{jump}} = 15$, $\mathbf{p} = 1$).

on recent observations, meaning that they cannot predict the future after encountering a divergence in the statistical properties of the input.

3.2.5 Adaptive LPC

The results on the compression with blocked LPC show that the optimal prediction coefficients are not necessary to achieve a reduction on the entropy of the sequence that is being encoded. Nevertheless, a disadvantage of blocked LPC as a preprocessing step lies in the fact that it performs two passes over the input data; one for calculating the coefficients, and one for calculating the prediction error and performing the actual compression. However, it is shown that the prediction is best when its calculation method is kept simple, that is, when 1st-order prediction is used, so that it can capture the 'trend' of the quality scores in the block as generally as possible. This fact can be used for converting the two-pass preprocessing step to a one-pass one, by sacrificing some compression ratio to achieve better compression speed.

In blocked LPC, in the case of 1st-order prediction, the set of normal equations is reduced to just one equation:

$$a_1 = \frac{\sum_{m=1}^{M}\sum_{n=1}^{N}[q_m(n)q_m(n-1)]}{\sum_{m=1}^{M}\sum_{n=1}^{N}[q_m^2(n)]} \tag{3.19}$$

Therefore, the value of the coefficient depends on the products $q(n)q(n-1)$ and $q(n)q(n)$ of all quality scores in a block, and needs a first pass over the data to calculate them, prior to encoding any value in the block.

In order to use LPC with only one pass, the notion of blocks is dropped, and these products are calculated for all quality scores prior to the one that is being encoded. In other words, when the k-th quality score of the whole input file is encoded, its a_1 coefficient is calculated based on the products $q(n)q(n-1)$ and $q(n)q(n)$ for all $n \in [1, k-1]$. The products are also weighted with a weight $w(n)$ to favour the most recent scores, in order to implement the adaptive nature of the LPC and to avoid overflow problems. Therefore, the coefficient calculation becomes:

$$a_1(k) = \frac{\sum_{n=1}^{k-1} [w(n) \cdot q(n)q(n-1)]}{\sum_{n=1}^{k-1} [w(n) \cdot q^2(n)]}, \ \forall k : q(k) \text{ is the k-th quality score of the input file} \tag{3.20}$$

By defining the nominator and denominator of Equation 3.20 as $A(k)$ and $B(k)$, respectively, the weighting is implemented with the following recurrence relations:

$$A(1) = 1$$

$$A(k) = \begin{cases} A(k-1) + q(k)q(k-1), & A(k-1) < N_{Thres} \wedge B(k-1) < N_{Thres} \\ \frac{A(k-1)}{2} + q(k)q(k-1), & A(k-1) > N_{Thres} \vee B(k-1) > N_{Thres} \end{cases}, \ k > 1 \tag{3.21}$$

$$B(1) = 1$$

$$B(k) = \begin{cases} B(k-1) + q^2(k), & A(k-1) < T \wedge B(k-1) < T \\ \frac{B(k-1)}{2} + q^2(k), & A(k-1) > T \vee B(k-1) > T \end{cases}, \ k > 1 \tag{3.22}$$

In these equations, T is a threshold for the $A(k)$ and $B(k)$ values that signifies how fast the nominator and denominator adapt to the input data.

The adaptive LPC is followed by the same modelling and arithmetic coding step as blocked LPC for the compression of the sequence of prediction errors. Figure 3.13 shows the compression ratio achieved for various values of T. As the parameter increases, the a_1 coefficient adapts more slowly to the recent data, since more quality scores are highly weighted. The compression ratio decreases for higher T values, which means that the coefficient benefits from including information about earlier data, showing that information is repeated, and a general 'trend' of the quality scores can be assumed.

Moreover, the results show that the compression is almost 1% worse that blocked LPC. This is attributed to the fact that blocked LPC takes into consideration 'future' quality values when encoding the k-th score, since the prediction coefficient has been calculated with all values of a block. Nontheless, adaptive LPC achieved a speedup of 1.245 compared to blocked LPC (13,8 MB/s for adaptive LPC compared to 11.08 MB/s for blocked LPC)

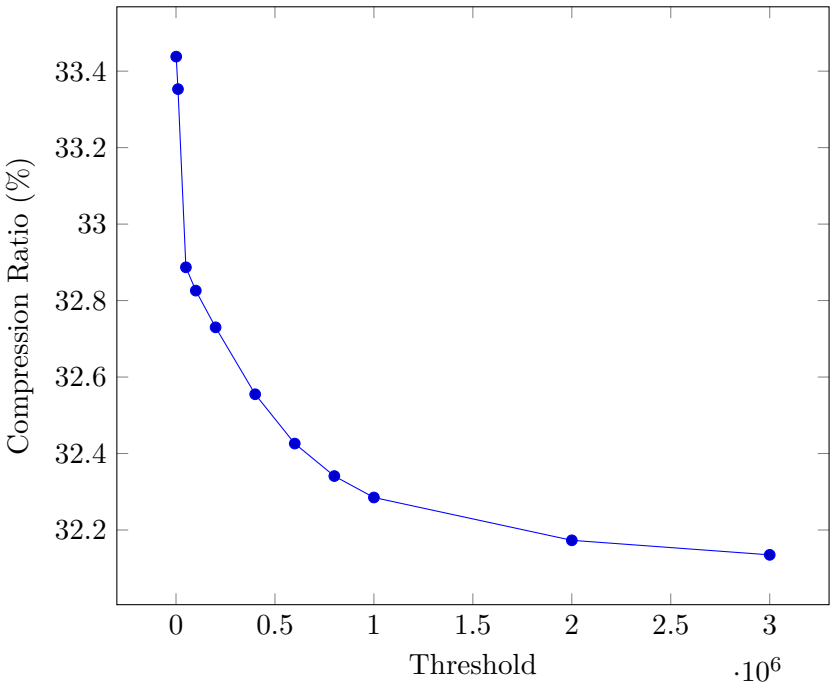

Figure 3.13: Compression ratio on the quality scores of C11 with adaptive LPC and different values of T ($\mathbf{l} = 2$, $\mathbf{N_{pos}} = 13$, $\mathbf{N_{jump}} = 15$, $\mathbf{p} = 1$, $\mathbf{N_{Thres}} = 4096$).

3.3 Modelling with Hidden Markov Models

DTMCs are very often used as models for compression purposes [31, 24, 5]. Their main advantages include being very simple in implementation, and being very effective models for input sequences that own the Markov property.

Nonetheless, it would be interesting to investigate the effectiveness of more complex modelling for the compression of quality scores. As such, an attempt is made in using *Hidden Markov Models* (HMM) [32] for modelling the sequences. No preprocessing is performed so as not to alter the statistical properties of the data, and the final entropy coder is once again arithmetic coding.

DTMC modelling assumes that the internal state of the information source of a sequence uniquely corresponds to the last symbols it has produced. However, if this is not the case for the source, and the information source can create the same sequence of symbols from two different internal states, the DTMC will miss this distinction.

HMMs, on the other hand, assume that the internal state-space of the information source is not directly observable, but it can be predicted based on the sequence of symbols it has produced. Therefore, an HMM models the information source as transitioning between a set of states that do not specifically correspond to observable physical events (as the name suggests, the states are hidden).

A Hidden Markov Model $\lambda = (A, B, \pi)$ is characterized of the following:

- N, the number of hidden states in the model. The set of states is denoted as $S = \{S_1, S_2, ..., S_N\}$. Every time that a state transition takes place, an observable

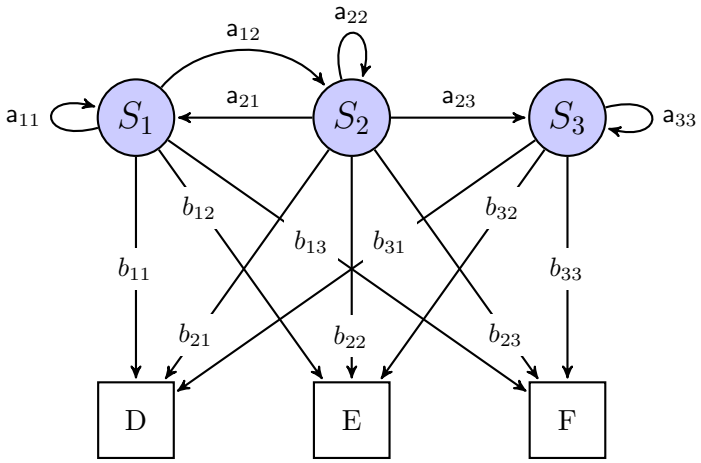

Figure 3.14: Representation of a simple Hidden Markov Model.

symbol is produced. The state at time point t is denoted as q_t.

- M, the number of possible values for the produced symbols, or in other words, the size of the alphabet of the produced symbols. The observation symbols correspond to the physical output of the system being modelled, in our case the sequence of quality scores. The individual symbols are denoted as $V = \{v_1, v_2, ..., v_M\}$
- The transition probability matrix $A = \{a_{ij}\}$, where:

$$a_{ij} = Pr\{q_{t+1} = S_j | q_t = S_i\},\ \forall i, j \in [1, N]$$

- The probability distribution for the symbol production, for state S_j, $B = \{b_j(k)\}$, where:

$$b_j(k) = Pr\{v_k \text{ produced at time } t | q_t = S_j\},\ \forall i \in [1, N], \forall k \in [1, M]$$

- The initial state probability distribution $\pi = \{\pi_i\}$ where:

$$\pi_i = Pr\{q_1 = S_i\},\ \forall i \in [1, N]$$

An example of a simple HMM with $N = 3,\ M = 3$ is shown in Fig. 3.14.

There are two main problems to be encountered when using HMMs. The first involves the training of the model with a subset of the input data in order to reflect the information source, and the second involves extracting information from the model.

3.3.1 Training an HMM

Determining the parameters of an HMM to model the source of a training sequence is a very difficult problem. In fact, there is no known way to analytically solve for the model

which maximizes the probability of the observation sequence [32]. However, an HMM $\lambda = (A, B, \pi)$ can be chosen such that it locally maximizes the probability of creating a training sequence $O = O_1O_2...O_T$, thus modelling its information source. This can be achieved with the Baum-Welsh method. Prior to presenting this method, the *forward variable* α_t and the *backward variable* β_t have to be calculated.

The forward variable $\alpha_t(i)$ is the probability of the partial observation sequence $O_1O_2...O_t$ while the state of λ is S_i at time t:

$$\alpha_t(i) = Pr\{O_1O_2...O_t,\ q_t = S_i|\lambda\} \tag{3.23}$$

The values of the forward variable can be calculated inductively:

$$\alpha_1(i) = \pi_i b_i(O_1),\ 1 \leq i \leq N \tag{3.24}$$

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i)\mathrm{a}_{ij}\right] b_j(O_{t+1}),\ 1 \leq t \leq T-1,\ 1 \leq j \leq N \tag{3.25}$$

Accordingly, the backward variable $\beta_t(i)$ is the probability of the partial observation sequence $O_{t+1}O_{t+2}...O_T$, given that the state of λ is S_i at time t:

$$\beta_t(i) = Pr\{O_{t+1}O_{t+2}...O_T|q_t = S_i, \lambda\} \tag{3.26}$$

Again, the values of the backward variable can be calculated inductively:

$$\beta_T(i) = 1,\ 1 \leq i \leq N \tag{3.27}$$

$$\beta_t(i) = \sum_{j=1}^{N} \mathrm{a}_{ij} b_j(O_{t+1})\beta_{t+1}(j),\ T-1 \geq t \geq 1,\ 1 \leq i \leq N \tag{3.28}$$

Using the forward and backward variables, we can calculate $\xi_t(i, j)$, which is the probability of the HMM being in state S_i at time t and state S_j at time $t + 1$:

$$\xi_t(i, j) = Pr\{q_t = S_i, q_{t+1} = S_j|O, \lambda\} \tag{3.29}$$

This is calculated as follows:

$$\xi_t(i, j) = \frac{\alpha_t(i)\mathrm{a}_{ij} b_j(O_{t+1})\beta_{t+1}(j)}{\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_t(i)\mathrm{a}_{ij} b_j(O_{t+1})\beta_{t+1}(j)} \tag{3.30}$$

Finally, the probability of the HMM being in state S_i at time t is $\gamma_t(i)$:

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{\sum_{i=1}^{N} \alpha_t(i)\beta_t(i)} = \sum_{j=1}^{N} \xi_t(i, j) \tag{3.31}$$

The Baum-Welsh method defines reestimation formulas for adjusting the parameters of an HMM based on a training sequence of data. More specifically, given the HMM $\lambda = (A, B, \pi)$ and the training sequence $O = O_1 O_2 ... O_T$, the identically sized HMM $\bar{\lambda} = (\bar{A}, \bar{B}, \bar{\pi})$ can be calculated using the following equations:

$$\bar{\pi}_i = \gamma_1(i) \tag{3.32}$$

$$\bar{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} \tag{3.33}$$

$$\bar{b}_j(k) = \frac{\sum_{t=1 \ s.t. \ O_t=v_k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(i)} \tag{3.34}$$

The HMM $\bar{\lambda}$, calculated with Equations 3.32 - 3.34, is provably a better model than λ in the sense that $Pr\{O|\bar{\lambda}\} > Pr\{O|\lambda\}$ [2], or the model that locally maximizes this probability, in which case $\bar{\lambda} = \lambda$. Therefore, by iteratively substituting λ with $\bar{\lambda}$ and calculating the re-estimation equations, λ will converge to an HMM trained to produce the training sequence O, thus modelling its information source.

There are two implementation issues for the Baum-Welsh method. The first involves around the fact that the values of $\alpha_t(i)$ tend exponentially towards zero as i increases (Eq. 3.25) and the same happens to $\beta_t(i)$ values, as i decreases (Eq. 3.28). Therefore, when the size of a training sequence increases, the precision needed to store each of these values exceeds the precision range of any machine. In order to avert this problem, the values of these variables are scaled at each time point by the coefficient c_t:

$$c_t = \frac{1}{\sum_{j=1}^{N} \alpha_t(j)} \tag{3.35}$$

Therefore, the scaled values are:

$$\hat{\alpha}_t(i) = c_t \cdot \alpha_t(i), \ \hat{\beta}_t(i) = c_t \cdot \beta_t(i) \tag{3.36}$$

Thus, the values of the two variables are substituted by the scaled ones in the re-estimation equations and the effect of the coefficients is cancelled as they appear both on the nominators and the denominators[5].

The second implementation issue arises from the fact that the quality scores are divided in reads. This means that there is not one single training sequence, but a set of K sequences:

$$\mathbf{O} = [O^{(1)}, O^{(2)}, ..., O^{(K)}] \tag{3.37}$$

[5]The full proof can be found in [32]

Therefore, the goal of the re-estimation equations is now to maximize:

$$Pr\{\mathbf{O}|\lambda\} = \prod_{k=1}^{K} Pr\{O^{(k)}|\lambda\} = \prod_{k=1}^{K} P_k \tag{3.38}$$

The initial re-estimation equations depend on summing frequencies of occurrence of symbols in the observation. So, they are modified for multiple observation sequences by summing over all of them, and by weighting the frequencies accordingly to the probability of their sequence.

By using scaling and multiple observation sequences, and in addition by selecting $\pi_1 = 1,\ \pi_i = 0,\ i \neq 1$, the re-estimation equations become:

$$\bar{\mathrm{a}}_{ij} = \frac{\sum_{k=1}^{K} \frac{1}{P_k} \sum_{t=1}^{T_k-1} \hat{\alpha}_t^{(k)}(i)\mathrm{a}_{ij}b_j(O_{t+1}^{(k)})\hat{\beta}_{t+1}^{(k)}(j)}{\sum_{k=1}^{K} \frac{1}{P_k} \sum_{t=1}^{T_k-1} \sum_{j=1}^{N} \hat{\alpha}_t^{(k)}(i)\mathrm{a}_{ij}b_j(O_{t+1}^{(k)})\hat{\beta}_{t+1}^{(k)}(j)},\ 1 \leq i,j \leq N \tag{3.39}$$

$$\bar{\beta}_j(l) = \frac{\sum_{k=1}^{K} \frac{1}{P_k} \sum_{\substack{t=1 \\ s.t.\ O_t=v_l}}^{T_k-1} \hat{\alpha}_t^{(k)}(i)\hat{\beta}_t^{(k)}(i)}{\sum_{k=1}^{K} \frac{1}{P_k} \sum_{t=1}^{T_k-1} \hat{\alpha}_t^{(k)}(i)\hat{\beta}_t^{(k)}(i)},\ 1 \leq j \leq N,\ 1 \leq l \leq M \tag{3.40}$$

where

$$P_k = \sum_{i=1}^{N} \hat{\alpha}_T^{(k)}(i) \tag{3.41}$$

3.3.2 Extracting information from the HMM

So far, we have described how an HMM can be trained to model the information source of the quality scores of a FASTQ file. However, it is not straightforward how to use the information contained in the model at the arithmetic coding step.

At the process of encoding the i-th symbol of the sequence of quality scores $O = O_1O_2...O_K$, the following procedure is followed:

1. Find the probability to be in each state of the HMM based on the sequence $O_1O_2...O_{i-1}$. In effect, this is the calculation of the forward variable $\hat{\alpha}_i(j),\ \forall j \in [1, N]$.

2. Calculate the probability of the state S_j being the next state q_i, without taking under consideration the knowledge of the symbol O_i, so that the decompressor can imitate this calculation. The probability is calculated as follows:

$$Pr\{q_1 = S_j\} = \begin{cases} 1,\ j = 1 \\ 0,\ else \end{cases} \tag{3.42}$$

$$Pr\{q_i = S_j\} = \sum_{l=1}^{N} \hat{\alpha}_{i-1}(l)\mathrm{a}_{ij},\ i > 1,\ 1 \leq j \leq N \tag{3.43}$$

3. Find the most probable next state S_p, $p = arg\max_j\{Pr\{q_i = S_j\}\}$

4. The symbol distribution for the most probable state is $b_p()$, which is used as the probability of Equation 1.14. Thus, we can easily calculate the cumulative distributions of Equations 1.17 and 1.18 for the arithmetic coding step.

The parameters of the trained HMM are embedded in the compressed file and are therefore known to the decompressor. Thus, the decompression process follows exactly the same steps to produce the cumulative distributions needed by the decoding algorithm of the arithmetic coder.

3.3.3 Implementation

The implementation of the described compressor is divided in two phases: The training phase and the compression phase.

During the training phase, the parameters of the HMM are estimated using sample reads from the input file as training sequences. If the set of training reads was comprised from consecutive reads in the file, there would be the danger of training the HMM to reflect some local statistical property that would be present in that particular subset, but not in the entire file. Thus, we opt to form the set of training reads by sampling the reads of the whole file, with a sampling period of T_s reads (that is, every T_s-th read is added to the training set).

The model has N hidden states and each state produces one of the $M = 41$ observable symbols in the alphabet of quality values. As stated in [32], the probability matrix A can be initialized with random values (that do not violate the stochastic property of the matrix), but the probability distributions in B have to be carefully selected for the selected application. Thus, the vectors b_j are initialized with a skewed probability distribution, much like the one shown in Fig. 3.3.

The training process is carried out in double precision floating point arithmetic, due to the need for high precision in the re-estimation equations 3.39 - 3.40. Ideally, the training step would iterate over these equations, up to the point where the model converges to the local maximum (i.e. $\bar{\lambda} = \lambda$). This practice however, could potentially lead to the need of a very high number of iterations if the initial model estimation is too far from the point of convergence, meaning that there exists a high scale of unpredictability on the timing performance of the training step. This fact, combined with the high computational load of double precision arithmetic, would render the training step too slow, to the point that it would be unusable. Therefore, it is imperative to enforce a maximum allowed number of iterations L, and the model that occurs after that many iterations is used for modelling, regardless of whether it has converged or not.

From the above, it is evident that there are three parameters that affect the compression performance of the compressor:

- N, the number of hidden states
- L, the maximum number of iterations allowed in the training step
- T_s, the sampling period for the creation of the training data

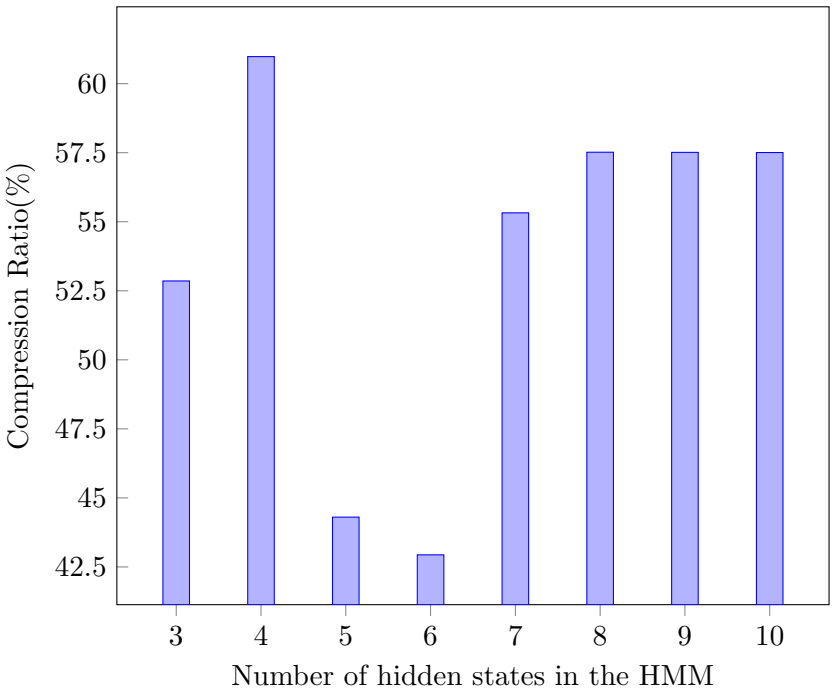

Figure 3.15: Compression ratio on the quality scores with HMM modelling using different numbers of hidden states in C11 ($\mathbf{L} = 100,\ \mathbf{T_s} = 10000$)

The actual effect of the parameters is investigated in the case of the C11 benchmark (Table 1.2).

The first one, N, is arguably the most important. It determines the size of the model, which in turn determines the ability of the model to reflect the statistical properties of the quality scores' information source. The effect of this parameter on the compression ratio achieved on the quality scores of is shown in Fig. 3.15. As shown, an incorrectly sized model has a huge impact on the compression ratio. Too few states fail to distinguish between the actual states of the information source and more than one can be mapped to the states of the HMM. On the other hand, when assuming too many states for the model, the excess states do not have any physical correspondence. Yet, some observations are inadvertently attributed to them, reducing the probability of transitioning to the actual state that created the observation.

Given the size of the model, the details of the training process have to be specified. First we investigate the number of iterations L needed for an adequate training of the model; Figure 3.16 illustrates the effect of this parameter. It must be noted that for these values of L the training did not converge to the local extremum[6]. By using the same initial values for the stochastic matrix A, this figure can also be interpreted as the learning curve of the model. For a small amount of iterations, the model is not adequately trained to model the specific properties of the input file. However, at around

[6]In fact, convergence for this particular setup and input file was not reached even after 650 iterations (for a precision of 0.01 for each parameter). The actual number of iterations was not determined and the attempt was aborted after the first 24 hours of execution.

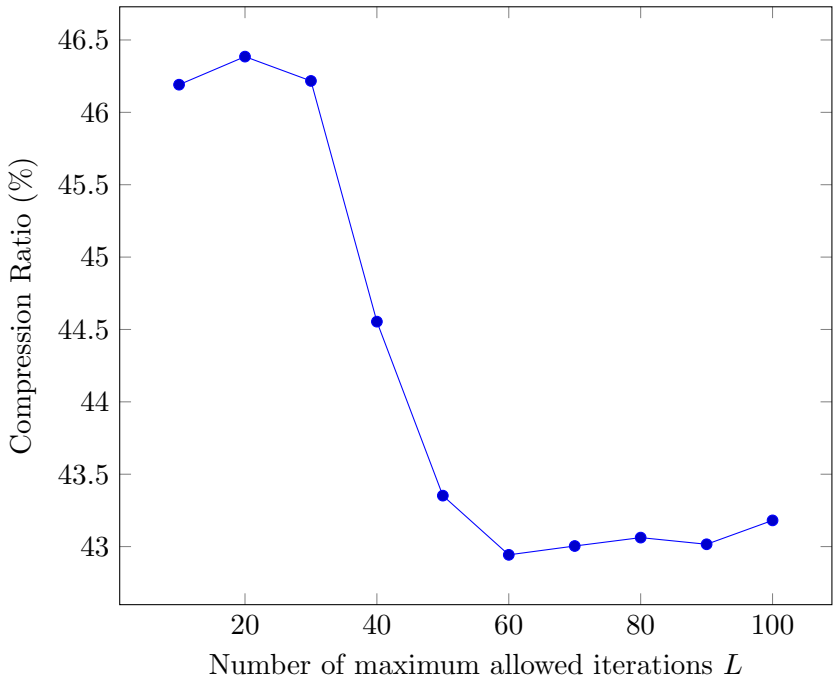

Figure 3.16: Compression ratio on the quality scores of C11 with different values of L for the HMM training ($\mathbf{N} = 6$, $\mathbf{T_s} = 10000$).

60 iterations, the training reaches close to the point of convergence, and remains close in the following iterations. A slight increase in the compression ratio takes place at this point, which is explained by the fact that the training procedure increases the possibility that the model can recreate the training sequences, but does not guarantee that it will reflect the whole file.

Finally, the size of the subset of the input file's reads that comprise the training set has to be determined. The size of the training set is the $1/T_s$-th of the set of all quality score sequences in the input file, and thus, it can be selected by modifying the sampling period T_s. Figure 3.17 shows the effect of different training sizes on the compression ratio. Generally, small sampling periods achieve better compression performance, since they correspond to larger sizes of the training set. Small training sets (larger T_s values) fundamentally contain less state transitions and thus result in an underfitted model, which does not adequately represent the information source. On the contrary, larger training sets result in better trained models, but they render the training process more computationally intensive; the number of computations in Equations 3.39-3.40 increases linearly with K.

3.4 Evaluation

Comparing the two compression methods that were presented in this chapter shows that blocked LPC is far superior to HMM modelling. As shown in Figures 3.12 and 3.17, the first method achieved approximately 9% lower compression rate when comparing

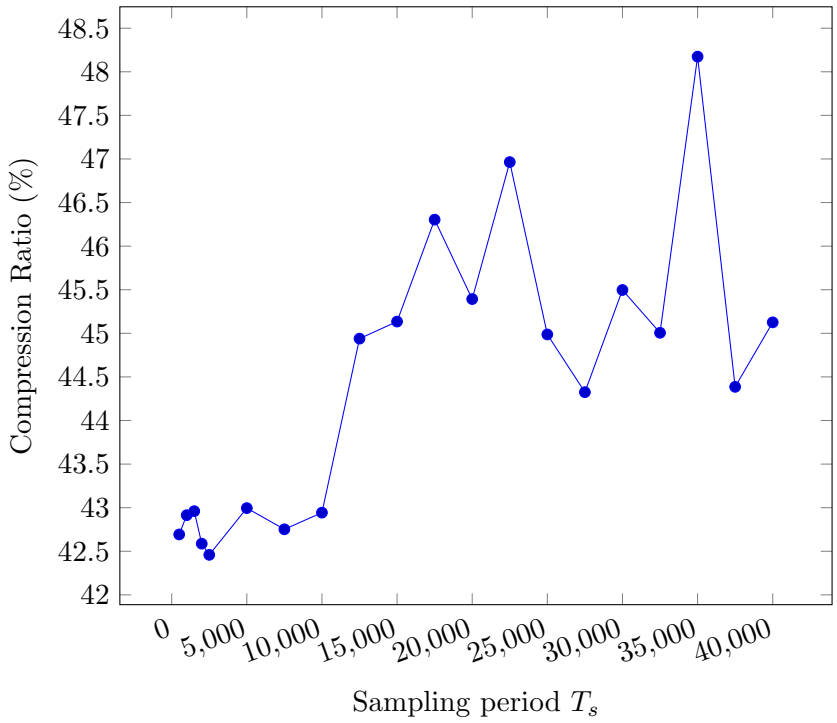

Figure 3.17: Compression ratio on the quality scores of C11 with different values of T_s for the HMM training ($\mathbf{N} = 6$, $\mathbf{L} = 60$).

the best observed results (31.2% and 42.4%, respectively). This is mainly a result of HMM modelling lacking the adaptive nature of blocked LPC. The same model is used for all quality score sequences throughout the file, and as a result, local properties are not taken under consideration. Generally, HMM modelling that adapts to the local input would be possible by training a new model for blocks of input data. However, training of an HMM is a very demanding task in terms of computational power, as will be shown below. Thus, performing it multiple times over the course of the compression of a single file would result in prohibitive low compression speeds.

In addition to blocked LPC consistently achieving better compression, all experiments showed that its performance is more robust with respect to the design parameters, as only small changes in the compression result were observed. On the contrary, HMM modelling requires perfectly calibrated parameters, and changes in the quality scores' information source would require the definition of the parameters anew.

What is more, the process of HMM training is too computationally demanding to be used in a big data compressor. For example, in the experiment with $N = 6$, $L = 60$, $T_s = 2500$ which proved to be the best in compression ratio terms, the training phase needed 6193 seconds to complete, while the actual compression needed only 1638 seconds in our testing system. This amounts to a compression speed of approximately 0.5 MB/s, which is extremely low, compared to the 15.08 MB/s measured for the blocked LPC. The reason behind the low speed is that fact that training a model is an iterative process, and as such, the program iterates over the training data multiple times. In addition, the

reestimation equations 3.39-3.39 contain a fair amount of calculations, which has to be performed in double precision arithmetic, thus contributing to the heavy computational requirements. On the other hand, blocked LPC performs only two passes over the data, one for calculating the prediction coefficients, and one for performing the actual compression. In addition, the only demanding computations are performed in solving the set of normal equations (Eq. 3.18), but for the small values of the order of the prediction that proved to be best, very few calculations are needed.

The only advantage of HMM modelling lies in the fact that it requires less memory at run-time, compared to the DTMC used to model the prediction error in the other method. However, this is not enough of an advantage to counter its low compression and speed performance.

Compression of Identifiers and Bases

4

As mentioned in the previous chapter, the identifier strings and the sequences of bases in a read carry less information content than the sequences of quality scores. Nonetheless, they comprise a big portion of a FASTQ file; the sequences of bases take exactly as much storage space as the quality scores, and the id strings are usually 50-60 characters long per read. Therefore, the efficient compression of this type of data plays a vital role in the compression ratio achieved for the entire file.

Fortunately, it is relatively easy to eliminate redundancies in id strings, using a simple form of delta encoding. Section 4.1 shows how delta encoding is used as a preprocessing step, along with DTMC modelling and arithmetic coding.

Redundancy in base sequences is more difficult to uncover. In Section 4.2, we present two different methods that were tested, each taking advantage of a different statistical property of the sequences. The first uses the fact that the alphabet of this type of data is limited to only 5 characters, while the second uses the fact that the reads are overlapping as a result of the sequencing procedure, which means that some information is repeated.

4.1 Identifier Strings

Generally, the entirety of the reads contained in a single FASTQ file comes from the same experiment. This means that all identifier strings contain the same information on the sequencing technology, the origin of the sequenced genome and any more relevant information. Usually, only one or two index numbers change between consecutive reads, meaning that the rest of the information of the identifier is redundant, and can be eliminated during the compression process, achieving very high compression easily.

HWI-D00267:38:H879KADXX:1:1101:1146:2170 1:N:0:CCGTCC
HWI-D00267:38:H879KADXX:1:1101:1238:2215 1:N:0:CCGTCC
HWI-D00267:38:H879KADXX:1:1101:1172:2215 1:N:0:CCGTCC
HWI-D00267:38:H879KADXX:1:1101:1123:2221 1:N:0:CCGTCC
HWI-D00267:38:H879KADXX:1:1101:1349:2099 1:N:0:CCGTCC
HWI-D00267:38:H879KADXX:1:1101:1452:2103 1:N:0:CCGTCC
HWI-D00267:38:H879KADXX:1:1101:1490:2166 1:N:0:CCGTCC
HWI-D00267:38:H879KADXX:1:1101:1463:2183 1:N:0:CCGTCC
HWI-D00267:38:H879KADXX:1:1101:1446:2209 1:N:0:CCGTCC
HWI-D00267:38:H879KADXX:1:1101:1492:2230 1:N:0:CCGTCC

Table 4.1: The first ten identifier strings of C11.

As an example, Table 4.1 shows the first ten identifier strings of benchmark C11 (Table 1.2). In this example, only 6 out of the total 53 characters change between

consecutive reads, with the rest being redundant. In the following, we show how this fact can be used to achieve a good compression ratio.

4.1.1 Preprocessing

A straightforward way of eliminating the redundant information in id strings is delta encoding. In this form of encoding, only the differences of a string to a previous one are encoded, instead of the entirety of characters. Thus, in the above example, it is possible to store only the 6 characters that change, with some overhead to indicate the positions of the changes.

In the context of id strings, we use a preprocessing step to transform the identifier string that is currently compressed, to its difference from the previous read. The difference between two strings is defined as the sequence of the differences in the ASCII values of characters in respective positions. In the case that the two stings are not equally sized and the string being compressed is larger than the previous, the shorter one is concatenated with zeros until it reaches the size of the larger one.

Therefore, for the id sequences $S_i = s_{i1}s_{i2}...s_{iN_i}$ and $S_j = s_{j1}s_{j2}...s_{jN_j}$, both over the alphabet $\mathcal{U}$ = ASCII codes = $\{0, ..., 127\}^1$, the difference D_{ij} over the alphabet $\mathcal{U}_D = \{-127, ..., 127\}$ is defined as:

$$D_{i,j}(n) = \begin{cases} s_{in} - s_{jn}, & 1 \leq n \leq N_j \\ s_{in}, & n > N_j \end{cases} \quad , \forall n \in [1, N_i] \tag{4.1}$$

Id 1	H	W	I	-	D	0	0	2	6	7	:	3	8	:	H	8	7	9	K	A	D	X	X	:	1	:	1	1	0	1	:	1	2	3	8	:	2	2	1	5	1	:	N	:	0	:	C	C	G	T	C	C
Id 2	H	W	I	-	D	0	0	2	6	7	:	3	8	:	H	8	7	9	K	A	D	X	X	:	1	:	1	1	0	1	:	1	1	4	6	:	2	1	7	0	1	:	N	:	0	:	C	C	G	T	C	C
$D_{2,1}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	-1	2	0	0	1	-6	5	0	0	0	0	0	0	0	0	0	0	0	0

Table 4.2: Delta encoding on the identifier strings (without misalignments).

For example, the difference of the second read from Table 4.1 from the first read is shown in Table 4.2. It is obvious from this example that the vast majority of the values in the difference strings will be zero, which means that the sequence has a very low information content.

Nevertheless, it is possible for some number to be in a different order of magnitude than it was in the previous read. When this happens, as a result of the ASCII representation, the rest of the string will be misaligned, resulting in non-zero difference values as shown in Table 4.3.

In order to alleviate this problem, a quick parsing step takes place before the calculation of the difference, converting all numbers in the string from their ASCII representation to their binary equivalent. In more detail, every number in the string with an ASCII representation of 1 to 9 bytes is converted to the tuple (*header byte*, *binary representation*). In the case that the ASCII representation of a number is larger than 9 bytes, it is split to smaller strings of that range.

[1]Extended ASCII is not used in the id string, thus all byte values are in the range $[0, 127]$.

Id i	<prefix>	:	1	:	1	1	0	1	:	1	1	3	4	:	1	0	0	2	3		1	:	N	:	0	:	C	C	G	T	C	C							
Id i+1	<prefix>	:	1	:	1	1	0	1	:	1	4	7	6	:	2	1	3	5		1	:	N	:	0	:	C	C	G	T	C	C							
$D_{i+1,i}$	0...0	0	0	0	0	0	0	0	0	-3	-4	-2	0	-1	-1	-3	-3	19	-17	-9	-20	20	10	-10	-9	0	-4	-13	17	0									

Table 4.3: Misaligned delta encoding on the identifier strings. The prefix is `HWI-D00267:38:H879KADXX`.

The header byte has a value equal to the number of bytes in the original ASCII representation. Therefore, the values 1 through 9 are reserved for this use, since they represent special characters and are never encountered in an id string. This way, the header byte specifies that the following bytes are not actual characters of the string and should not be interpreted as such. It is also needed in order to be able to distinguish and recreate numbers in the string that start with one or more 0s, for example 0123 instead of 123.

Value of the header byte	**Size of bin. representation in bytes**
1,2	1
3	2
4,5	3
6,7,8	4
9	5

Table 4.4: Size of the binary representation according to the value of the header byte.

Moreover, the header byte determines the size of the binary representation of a number. The number of bytes used for each header byte is shown in Table 4.4. Using these sizes, it is guaranteed that misalignments occur much less frequently, since numbers in different orders of magnitude have representations of the same size.

In the bytes of the binary representation, only the 7 less significant bits are used, and the most significant one stays at 0. As an example, the number 10023 will be transformed to the tuple $(\texttt{0x05}, \{\texttt{0x00}, \texttt{0x}4E, \texttt{0x27}\})$. With this convention, the values of these bytes are always in the range $[0, 127]$ and therefore, the difference between them and any other byte in the string is always in the range $[-127, 127]$ which can be stored in a single byte. This way, the difference string remains equal in size to the largest of the two id strings that are compared.

In overall, the transformation of numbers to their binary representation only increases the size of the id string for single digit numbers, as 1 header byte and 1 byte of bin. representation are needed. Yet, that increase has a trivial impact to the compression ratio, if the difference values are 0, since the relative frequency of this particular symbol is significantly higher than any other symbol, which in turn means that arithmetic coding would need close to 0 bits to encode it. For numbers of other sizes, the transformed number is actually smaller than or equal to the initial size.

As an example of the effect of this transformation, we will see its effect on the two previous examples in Tables 4.5 and 4.6. These show the delta encoding procedure with

Read 1	H	W	I	-	D	0x05	0x00	0x02	0x0B	:	0x02	0x26	:	H	0x03	0x06	0x6F	K	A	D	X	X	:	0x01	0x01	:	0x04	0x00	0x08
Read 2	H	W	I	-	D	0x05	0x00	0x02	0x0B	:	0x02	0x26	:	H	0x03	0x06	0x6F	K	A	D	X	X	:	0x01	0x01	:	0x04	0x00	0x08
Diff	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Read 1	0x4D	:	0x04	0x00	0x09	0x56	:	0x04	0x00	0x11	0x27		0x01	0x01	:	N	:	0x01	0x00	:	C	C	G	T	C	C
Read 2	0x4D	:	0x04	0x00	0x08	0x7A	:	0x04	0x00	0x16	0x7A		0x01	0x01	:	N	:	0x01	0x00	:	C	C	G	T	C	C
Diff	0	0	0	1	-1	36	0	0	1	5	83	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4.5: Delta encoding on the identifier strings of Table 4.2 using the binary representation transformation. The red colour signifies the values of the bytes, while black colour signifies ASCII characters.

i	<prefix>	0x04	0x00	0x08	0x6E	:	0x05	0x00	0x4E	0x27		0x01	0x01	:	N	:	0x01	0x00	:	C	C	G	T	C	C
$i+1$	<prefix>	0x04	0x00	0x0B	0x44	:	0x04	0x00	0x10	0x57		0x01	0x01	:	N	:	0x01	0x00	:	C	C	G	T	C	C
Diff	0...0	0	0	3	-42	0	-1	0	-62	48	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 4.6: Delta encoding on the identifier strings of Table 4.3 after the binary representation transformation. The red colour signifies the values of the bytes, while black colour signifies ASCII characters. The prefix is HWI-D0x05 0x00 0x02 0x0B:0x02 0x26:H0x03 0x06 0x6FKADXX:0x01 0x01:0x04 0x00 0x08 0x4D:.

the binary transformation for the read identifiers used in Tables 4.2 and 4.3, respectively. In the first case, the transformation does not interfere with the result of the delta encoding, as the same number of symbols are non-zero. In the second case, the transformation eliminates the misalignment problem, and increases the relative frequency of symbols with a value of zero in the difference string, resulting in a sequence that is more highly compressible.

4.1.2 Modelling and Coding

Following the transformation described above, and delta encoding, the distributions of the symbols in the $D_{i,i-1}$ sequences, are significantly skewed, with 0 being the value of the vast majority of them. In addition, the alphabet is quite large with 255 distinct values, and thus, a high order DTMC would be extremely memory consuming and difficult to train.

Given the above, the model size can be kept small, while maintaining its accuracy, by reasoning on whether past values are 0 or not, rather than using their exact value. Hence, a pth-order adaptive DTMC model is used for modelling, but with the following modification: it has 2^p states, each corresponding to whether the previous p values of a difference sequence are 0 or not. Each of the states has 255 probabilities, each probability p_v corresponding to the probability of encountering the symbol v. Figure 4.1 shows part of a such a 3rd-order DTMC. Once again, the model holds cumulative frequencies in each state instead of the actual probabilities, with the frequencies of each state updated each time that particular state is reached. This way, the values of the cumulative frequencies can be used directly from the arithmetic coder, without the need for summation. In addition, the model is adaptive, meaning that every time a certain threshold (N_{thres}) on any measured frequency is reached, the frequencies of that specific state are scaled to

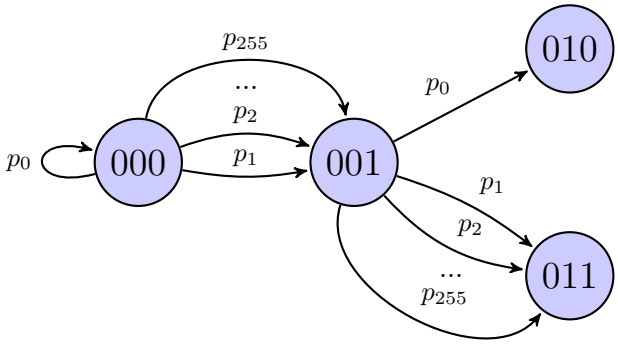

Figure 4.1: Part of a 3rd-order modified DTMC. 0s and 1s in the states correspond to whether the corresponding previous symbol had the value 0 or not, respectively.

half, adapting the model to more recent observations. The final step of compression is the standard arithmetic coder that was implemented, using 32 bits of precision.

4.1.3 Evaluation

As described above, the compression performance on the id strings depends on two parameters:

- **p**: the order of the adaptive DTMC.
- $\mathbf{N_{thres}}$: the highest value that can be observed on some measured frequency of any state in the DTMC. When the value is surpassed at some state, all measured frequencies for that state are scaled down by a factor of 2.

The performance metric is the compression ratio for the N21 benchmark (Table 1.2), the id strings of which amount to 814.97 MBs to storage space, with an average id string size of 55.5 characters.

Figure 4.2 shows the effect of different values of p. This parameter directly affects the size of the model. Therefore, as the value increases, the model becomes more accurate in predicting the symbols, as more and more previous values can be taken into account. The drawback on larger values however is the fact that the size increases exponentially with p, therefore increasing the memory needed by the compressor.

Figure 4.3 shows the effect of the value of N_{thres} on the compression ratio. Small values of this parameter cause the modelling step to depend more heavily on recently encoded symbols, and as the value increases, the weight on older increases as well. In the context of id strings, the higher the value, the better the compression, which means that frequent rescaling does not benefit the compression. This in turn means that the statistical properties of the difference strings do not change in the course of the file, and therefore the scaling procedure harms the compression by keeping the measured frequencies in small values, losing accuracy when used for probability calculations. For example, let's assume that a symbol has been observed 25 times over the last 1000 observations, meaning that it has a 2.5% probability of occurrence. If however the

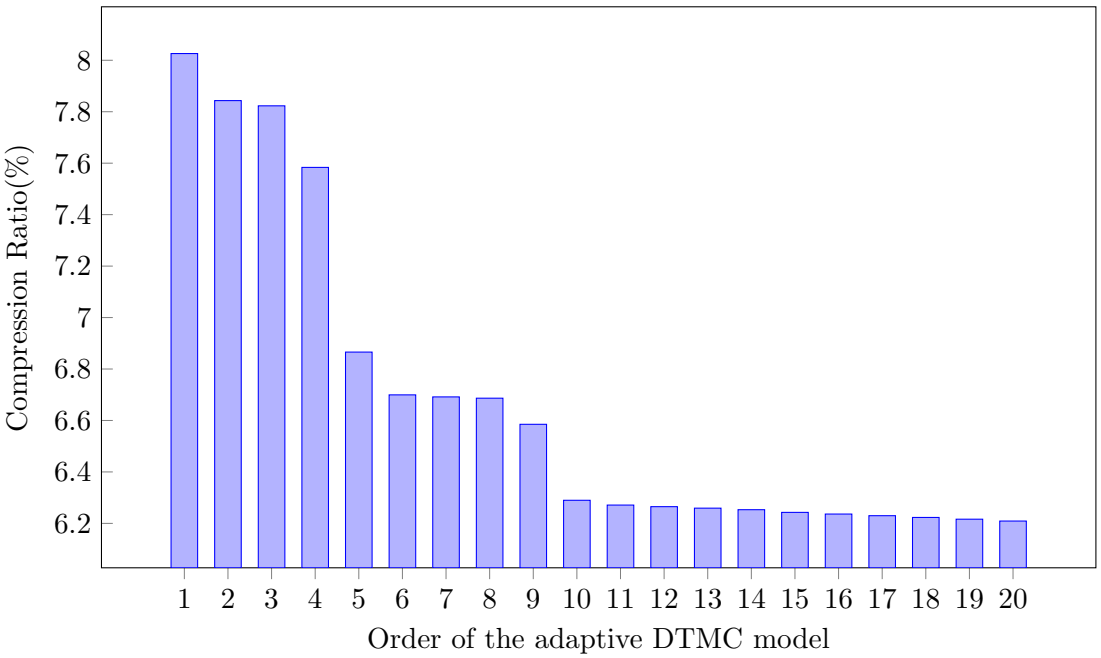

Figure 4.2: Compression ratio on the identifier strings using different values for the order of the DTMC in N21 ($N_{thres} = 16384$).

measured frequencies are scaled to half, the probability is perceived as 12/500=2.4%, producing an error of 0.1%. Given the fact that most symbols in the difference strings are very rare, even such small errors lead to incorrect modelling, thus harming the compression performance.

Nevertheless, this observation does not render the adaptive modelling obsolete. The opposite case of using a static model would result in modelling that is not flexible to a variety of inputs, and simply abolishing scaling would lead to overflow problems.

Finally, it is obvious that the compression ratio that is achieved for the id strings is far superior to the one achieved for the quality scores. This fact confirms the statement in the beginning of the chapter that there is less information content in this type of data, and as such there is no need to explore other methods of preprocessing or modelling.

4.2 Sequences of Nucleotide Bases

From the symbols that comprise the sequences of nucleotide bases, A, C, T, and G are almost equally frequent, with N being much rarer, as shown in Table 4.7 for the N21 benchmark (Table 1.2).

There are two main properties of the base sequences that can be taken advantage of in order to eliminate redundant information. In this section two methods for compressing the sequences of bases are presented and compared, each taking advantage of a different property.

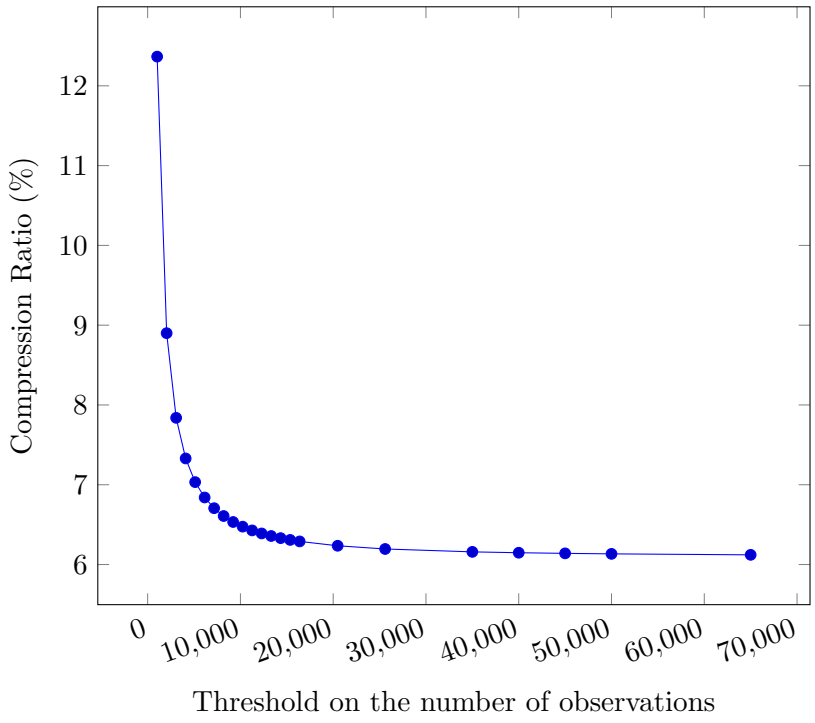

Figure 4.3: Compression ratio on the identifier strings of N21 with different values of N_{thres} ($p = 10$).

Symbol	Frequency	Relative Frequency
A	213705700	0.27210
C	180910184	0.23034
T	212836224	0.27099
G	176721194	0.22501
N	1230778	0.00157

Table 4.7: Frequencies of occurrence of each symbol in the base sequences of N21, with a total number of 785404080 bases in the file.

4.2.1 Utilizing BWT and MTF preprocessing

The first property is the fact that these sequences have a very small alphabet, consisting of only 5 characters: A, C, T, G, and N. As a first method, BWT and MTF transformations (see Section 2.1.2) are performed in the preprocessing step, while adaptive DTMC and arithmetic coding are used for modelling and coding, respectively.

The BWT transformation groups identical characters of a sequence, creating runs of characters that can be coded more efficiently than single characters. As the alphabet is small, the runs generated by the transformation tend to be large, and therefore, after the MTF transformation, the resulting sequence is more highly compressible than the initial base sequence.

The BWT transformation becomes increasingly more effective as the the size of the

input increases [1]. Thus, a number of M base sequences $S_1, S_2, ..., S_M$, each with a size of N characters, are concatenated to form the string S. Considering that each base sequence is comprised of N characters, i.e. $S_i = s_{i,1}s_{i,2}...s_{i,N}$, the concatenation is the following:

$$S = s_{1,1}s_{1,2}...s_{1,N}s_{2,1}s_{2,2}...s_{M,N}Z = s_1s_2...s_{N'}Z \tag{4.2}$$

S contains the $N' = N * M$ characters of the sequences plus the character Z, which is appended as a terminating character. This specific character is selected, since it is never encountered in the base sequences, and is lexicographically larger than any other character[2].

The result of the transformation, $BWT(S)$, is the sorted sequence of the characters s_i, $\forall i \in [1, N' + 1]$, where $s_i > s_j$, when the string $s_{i+1}s_{i+2}...s_{N'}Zs_1...s_i$ is lexicographically greater than the string $s_{j+1}s_{j+2}...s_{N'}Zs_1...s_j$. A simple implementation of the transform can be implemented with any sorting algorithm, such as quicksort [21].

The MTF transformation converts $BWT(S)$ to S', transforming from the alphabet $\mathcal{U} = \{A, C, T, G, N, Z\}$ to the alphabet $\mathcal{U}' = \{0, 1, 2, 3, 4, 5\}$. Same as in bzip2, the purpose of the MTF is to uncover the beneficial effects of the BWT, by skewing the symbols' probability distribution, so that lower values have higher probability of occurrence.

The transformation is performed by creating a list of the 6 symbols of the input alphabet. Each of the 6 positions in the list is associated with a corresponding number; the symbol at the top of the list is assigned the number 0, the next one is assigned the number 1, and so on. Every time a symbol is observed in the input string (thus, for every symbol in $BWT(S)$ in this case), the number assigned to it is concatenated to S', and the symbol is moved to a higher position in the list - hence the name Move-To-Front for the transform. Therefore, when runs of the same character are encountered in the input, the character moves to the top of the list and runs of 0s are concatenated in the output.

Input		-	A	A	A	G	G	G	C	C	Z	N
Output		-	0	0	0	3	0	0	2	0	5	4
List	**0**	A	A	A	A	G	G	G	C	C	C	C
	1	C	C	C	C	A	A	A	G	G	G	G
	2	T	T	T	T	C	C	C	A	A	A	A
	3	G	G	G	G	T	T	T	T	T	T	N
	4	N	N	N	N	N	N	N	N	N	N	T
	5	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z

Table 4.8: MTF transform on the sequence AAAGGGCCZN.

Since some statistics for the input sequence are known, the way each symbols moves u in the list can be controlled for better results. As such, when one of the characters A,C,T or G are observed in the input, this character moves to the top of the list, thus

[2]The transformation requires a terminating character that is either lexicographically larger or smaller than all other characters.

transforming the rest of the run to 0s. N on the other hand is much rarer, and therefore its runs in the $BWT(S)$ string will be much shorter. Therefore, each time this character is observed, it only moves one position towards the top of the list. Finally, the character Z is unique in S. So, when it is observed, it is certain that it will not occur a second time, and therefore, it stays always in the bottom of the list and is exclusively assigned the number 5. An example of how the list changes during the transformation is shown in Table 4.8.

Usually, the MTF transform is followed by a run-length encoding step, at least for the highly frequent 0 value. However, in this case, this is not needed, as arithmetic coding inherently performs run-length encoding. A closer look to the encoding algorithm (see Appendix A and Section 1.2.5) shows that output bits are only emitted at the rescaling step. However, when the frequency of a symbol is high, the newly calculated interval Φ_k is close to Φ_{k-1}, and thus rescaling is scarce. On the other hand, a symbol with low frequency of occurrence, and therefore low probability, will result in a small interval, and possibly more than one rescalings. So, a whole run of zeros will be coded in a small amount of bits, while the rare 5 will require more bits, exactly as in run-length encoding.

Therefore, the result of the MTF, S', is modelled by a pth-order adaptive DTMC, with scaling by a factor of 2 on the measured frequencies of a state every time one of them reaches the threshold N_{thres}, same as previous usages of this kind of modelling. Finally, the result of the modelling is used by a 32-bit precision arithmetic coder, that compresses the entirety of the M sequences of bases in one codeword.

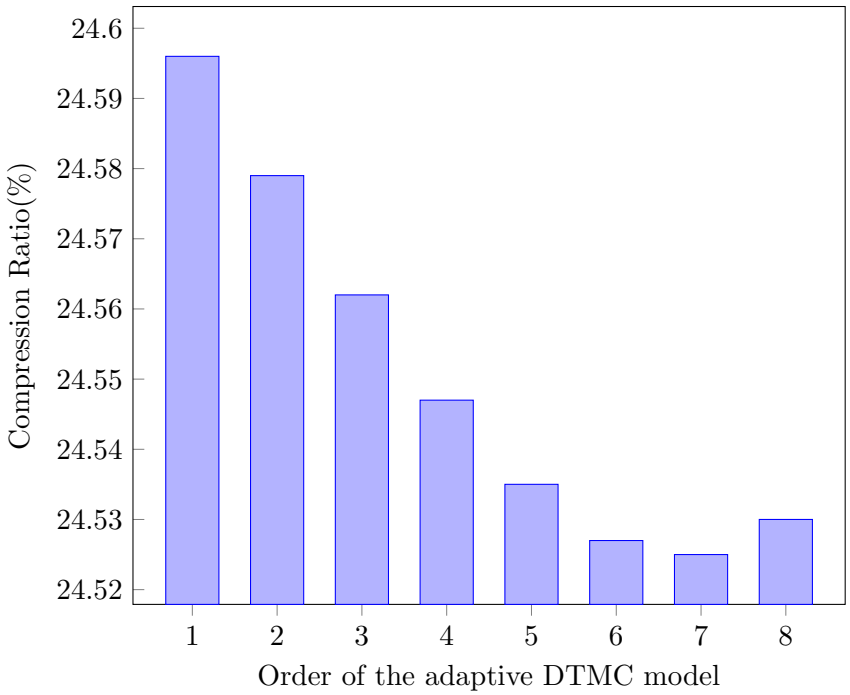

Figure 4.4: Compression ratio on the base sequences using different values for the order of the DTMC in N21, with BWT and MTF preprocessing ($N_{thres} = 4096,\ M = 50$).

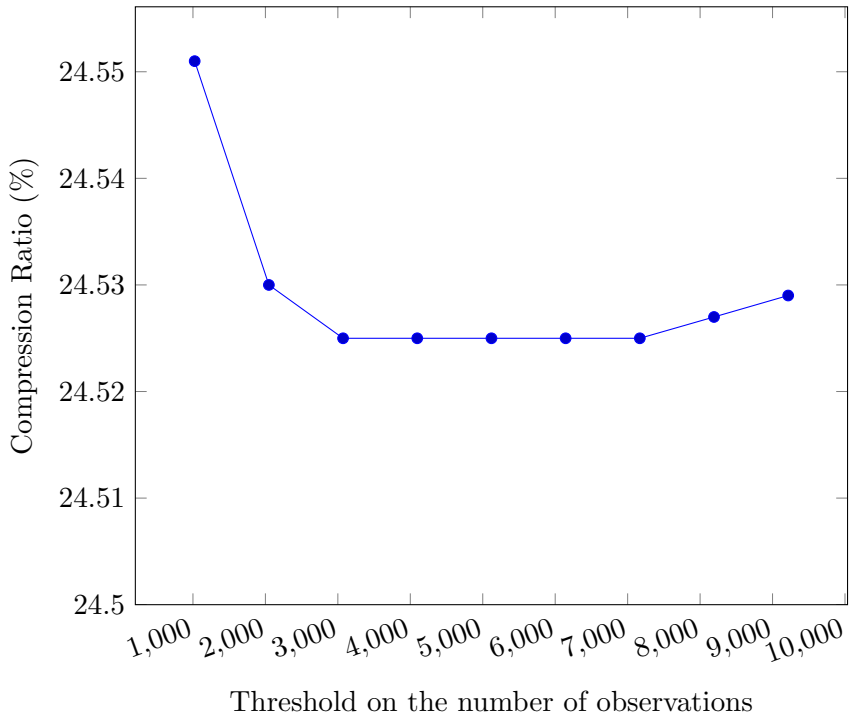

Figure 4.5: Compression ratio on the bases of N21 with different values of N_{thres}, using BWT and MTF preprocessing ($p = 7,\ M = 50$).

The parameters that influence the compression ratio in this method are p, M and N_{thres}.

Figure 4.4 shows the effect of the order of the modelling DTMC. It is evident that this parameter does not have a large influence on the compression, since the difference is in the order of 0.01%. Since a 1st-order DTMC is almost as effective as a higher order one, we can deduce that the values of the result of the MTF transform depend mostly on the value of the previous symbol and not on the values of any symbols before that. Similarly, the value of N_{thres} has a minimal effect in the achieved compression ratio, as shown in Figure 4.5.

The parameter M on the other hand, that is, the number of base sequences that are concatenated to create S, plays a more vital role. Figure 4.6 shows how the compression ratio is affected by this parameter. The ratio improves as the number of reads in S increases, since the BWT creates larger runs of characters in larger input sequences. Also shown in the same figure is the compression speed achieved with a simple implementation that uses the quicksort [21] sorting algorithm (calculated for compressing the bases only, but reading from the whole FASTQ file). As this algorithm has an $O(nlogn)$ average performance, the compression speed is $O(\frac{1}{nlogn})$, as shown in the figure. It is evident that the value of M balances the trade-off between the compression ratio and compression speed. However, despite the fact that the measured implementation is rather simplistic and could be greatly improved, the preprocessing step takes a heavy toll on the compression speed, which is the main drawback of bzip2 compressor as well. This is the

reason that a faster method for compressing the nucleotide bases is investigated in the following.

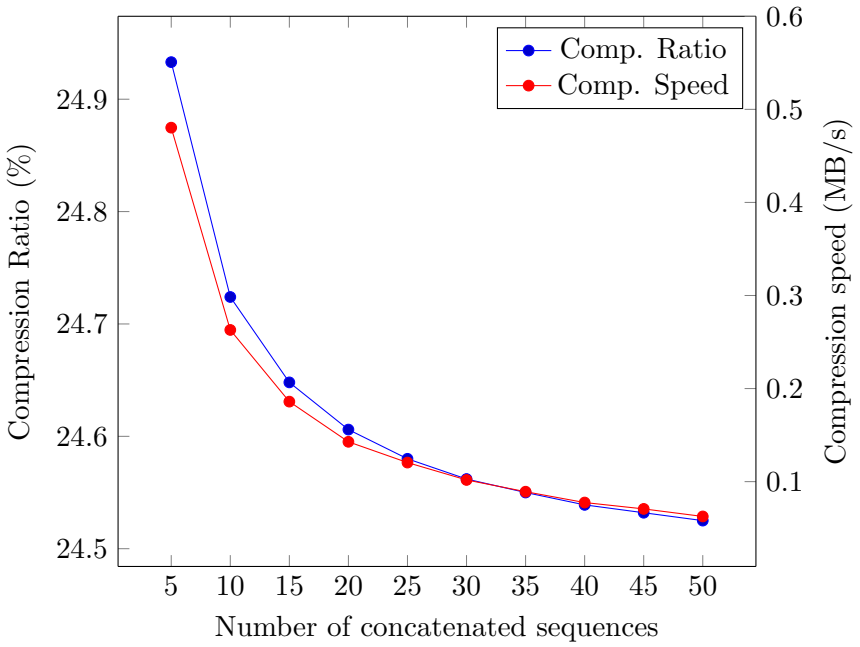

Figure 4.6: Compression ratio and speed on the bases of N21 with different values of M, using BWT and MTF preprocessing ($p = 7,\ N_{thres} = 4096$).

4.2.2 Skipping the preprocessing

The second property of the id strings that can be used for compression is derived from the content and construction method of the data.

The sequence of bases in a DNA molecule consists of sub-sequences that belong in two categories, namely *coding* and *non-coding DNA* [34]. Coding DNA contains information for the creation of amino acids, and since there is a finite amount of amino acids in nature, certain sub-sequences are repeated to create this part. Non-coding DNA is the part of the base sequence that is not involved in the creation of amino acids. Instead, non-coding DNA is used for a number of other functions, not all of which are known yet, and can possibly be non-functional or evolutionary residues. Nevertheless, even this category of DNA consists of sequences that are present in multiple copies in the genome, aptly named *repetitive DNA*. In overall, DNA consists of structured information with a large number of repetitive sub-sequences.

Moreover, during the sequencing procedure, a number of identical clones of the same DNA molecule are fragmented to create the reads. Therefore, the reads overlap and they can recreate the initial sequence of bases in the molecule, via the assembly procedure. However, this overlap also leads in duplicate identical sub-sequences spread over the reads (see Fig. 1.3), which could be eliminated.

Thus, the base sequences of the reads contain a large number of repetitive sub-sequences of varying sizes. Identifying these repeated sub-sequences directly is possible (with the LZ algorithm [49] for example), however it would be an extremely arduous task to perform on a large file size.

Instead, the knowledge that the input data contains a large number of repetitions can be used to predict the next character given its predecessors. Given that the p last characters that have been encoded are probably part of a repeated sub-sequence, previous observations of that particular p -sized sequence can give us the next character.

The described procedure can be implemented with a pth-order adaptive DTMC model, measuring the number of times each of the 5 bases is encountered, given the previous p bases.

Therefore, in this method for the compression of bases, no preprocessing is performed, so that the sub-sequence repetitions remain unchanged. Instead, a pth-order DTMC is used to model the sequences, and a 32bit-precision arithmetic coder is used for the creation of the output bitstream. The described model would contain $5^p \cdot 5$ parameters. Intuitively, as the value of p increases, larger repetitions can be taken into account, resulting in more accurate modelling. Nevertheless, the value of p has to have an upper limit, in order to keep the model size in manageable levels. In order to alleviate this restriction, we take under consideration the fact that the character N is much rarer than the rest. So, when determining the state of the model, N is regarded as the character A, reducing the number of states to 4^p and the number of parameters in the model to $4^p \cdot 5$, with minimal effect on the accuracy of the model. For example, in a 5th-order model, if the last encoded characters are `ACTGN`, the model will be in the state associated with the sequence `ACTGA`. Same as in previous usages of an adaptive DTMC, the adaptivity parameter N_{thres} is the value that causes the measured frequencies of a state to be scaled down by a factor of two when being surpassed.

Figure 4.7 shows the effect of p on the compression ratio achieved with this method on the N21 benchmark (Table 1.2), verifying that higher model order values result to more accurate modelling and therefore better compression ratios. Figure 4.8 shows that the value of N_{thres} has a minimal effect on the compression ratio, after a certain value is reached. It should be noted that this method achieves a compression speed of approximately 11 MB/sec for $p = 12$, in the same setup as the measurements for the previous method (calculated for compressing the bases only, but reading from the whole FASTQ file).

4.2.3 Evaluation

Comparing the two methods that were described earlier, it is evident that omitting the preprocessing step is superior both in terms of compression ratio and compression speed.

The superior ratio reveals that the repetition of sub-sequences in the DNA molecule is a property that can be very advantageous when used for compression. Preprocessing with the Burrows-Wheeler transform fails in comparison, because it 'breaks' this property by permuting the characters. Moreover, it works in a local context only; it does not take under consideration any information outside the block of M reads.

On the other hand, the method that skips the preprocessing step adapts the DTMC

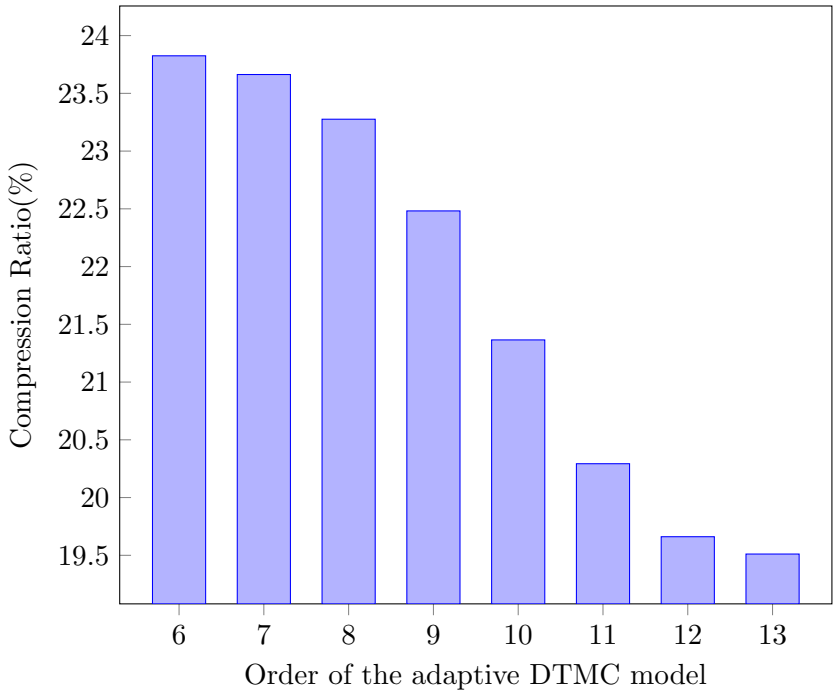

Figure 4.7: Compression ratio on the base sequences using different values for the order of the DTMC in N21, without preprocessing ($N_{thres} = 4096$).

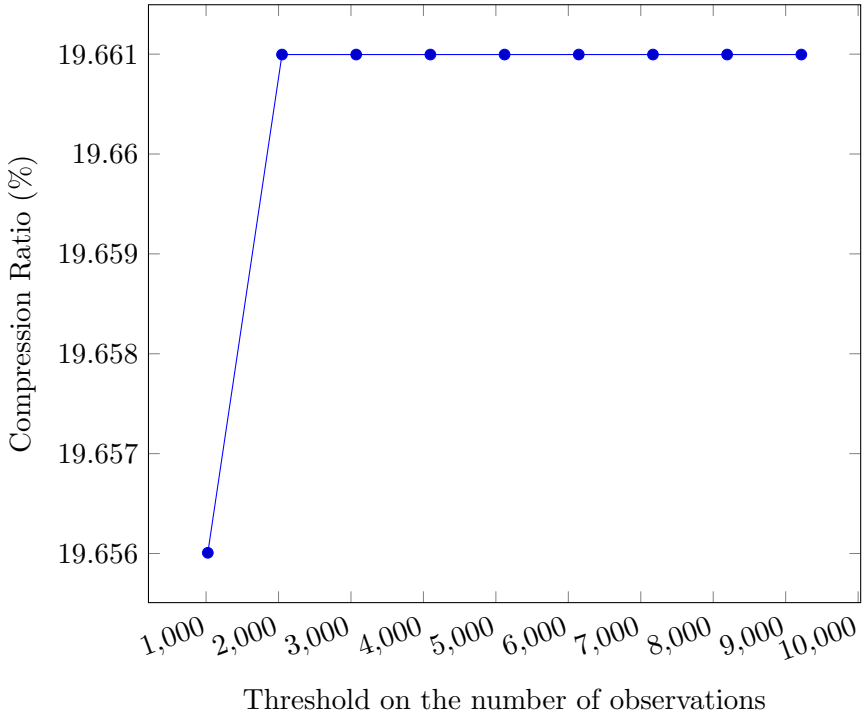

Figure 4.8: Compression ratio on the bases of N21 with different values of N_{thres}, without preprocessing ($p = 12$).

model as the compression progresses. Thus, the model contains information from the beginning of the file, up to the last encoded character, and the larger the input data, the closest the model is fitted to the information it contains. Since, the entirety of the compressed data is used for training the model, there is not danger of overfitting, as was the case on HMMs.

In addition, the second method's superiority in compression speed is a result of its simplicity. It needs very little computational power, with the only major computations being the ones performed by the arithmetic coder. Rather, its main bottleneck derives from the large amount of memory needed by the DTMC, which leads to cache misses and high memory latency.

5 Complete Compressor

Based on the techniques described on the previous chapters, a simple implementation of a compressor/decompressor pair was developed, incorporating the methods that were deemed best through experimentation. The implemented compressor includes the following three parts:

- The id strings compressor, which uses the delta encoding preprocessing step, along with DTMC modelling and arithmetic coding. The DTMC has an order of 10, so as to achieve adequate precision in modelling (see Fig. 4.2) while at the same time keeping the memory consumption low. The N_{Thresh} value was kept at 65000.
- The bases compressor, which performs no preprocessing on the data, but encodes with arithmetic coding, based on DTMC modelling. In this case, the order of the model is kept at the value of 12, even though a higher order shows better compression. This is chosen as such, because the memory needed for storing the model increases exponentially with the order, and a 12-th order model already requires 160 MB. The N_{Thresh} value was kept at 2000.
- The quality scores compressor, which performs adaptive LPC preprocessing, and models the prediction errors with DTMCs and codes once again with arithmetic coding. The parameter values that are used are: $l = 2$, $N_{pos} = 13$, $N_{jump} = 15$, $p = 1$, $N_{Thres} = 4096$, $T = 3000000$, which proved to be best during the experiments.

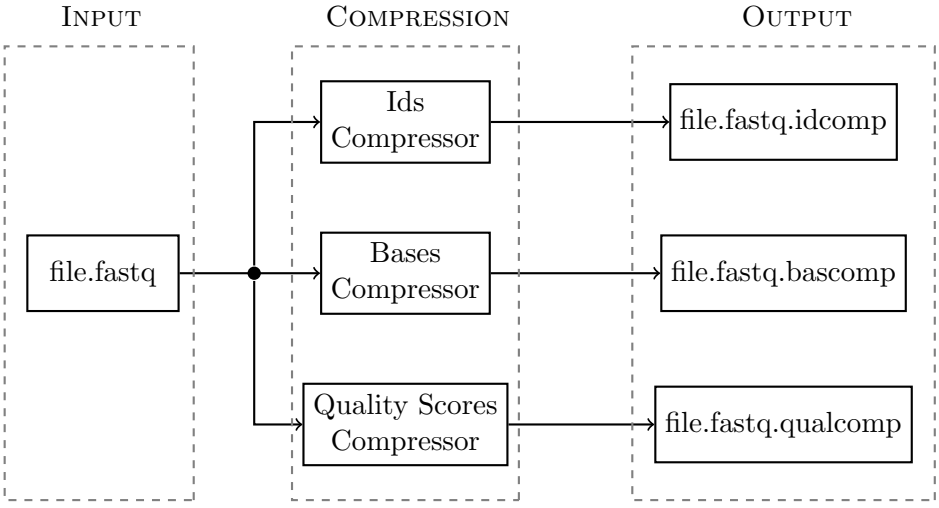

Figure 5.1: The compressor.

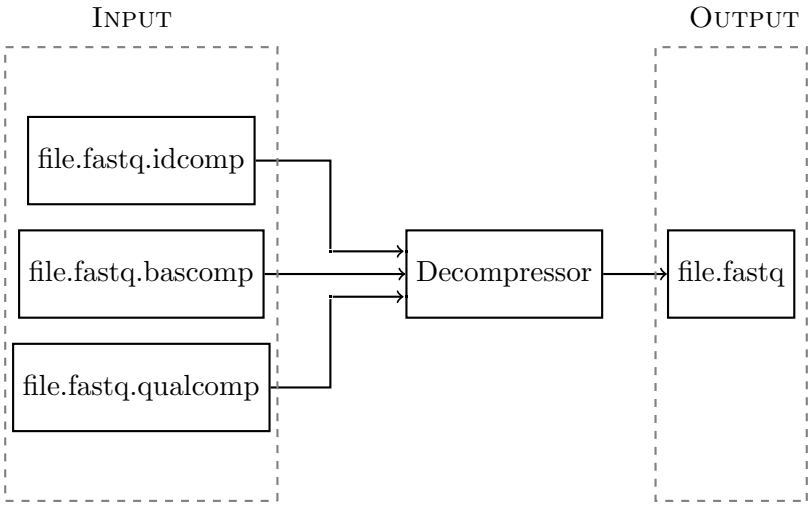

Figure 5.2: The decompressor.

The compression for each of the three types of data is performed independently from the others, with three distinct programs (Fig. 5.1). Thus, the result of the compression is given in three distinct files, that contain all the information needed to losslessly reconstruct the original file.

The decompression process on the other hand does not follow the same technique (Fig. 5.2), since the decompressed results have to be multiplexed to recreate the FASTQ format, serializing the accesses on the three files.

5.1 Compression performance

The compression results that are achieved for the five benchmarks of Table 1.3 are shown in Table 5.1. The compressor is able to compress a FASTQ file in almost a fifth of its original size, without the loss of any kind of information. The results once again prove the claim that id strings contain very little information, while the quality scores have the highest information content and are the most difficult to compress.

Moreover, we can see that the level of compression that can be achieved on sequences quality scores and bases can vary according to the input file. Especially for the quality scores, the compression ratio depends on the variety of scores that are more frequently encountered in the file. A highly skewed distribution of the quality scores contains less information and results in more efficient compression, as evidenced between the benchmarks C11 and C12 (Fig. 5.3) which have approximately 6% difference in the compression ratios of quality scores. This difference is even more apparent when comparing to the distribution of N21 (Fig. 3.3), which contains high quality reads, resulting from higher quality samples for the sequencing process.

A similar effect can be observed for the compression the base sequences. However, this is not attributed to the distribution of individual base characters, since these are usually almost uniform. Rather, the difference lies in the fact that different files have

Uncompressed File (measurements in bytes)

Benchmark	Ids	Bases	Quality scores	Total Size
N21	854555441	800804160	800804160	2502364001
C11	2243457790	4123334088	4123334088	10611400498
C12	2243457790	4123334088	4123334088	10611400498
C21	2231388138	4101106146	4101106146	10554221199
C22	2231388138	4101106146	4101106146	10554221199

Compressed File (measurements in bytes)

Benchmark	Ids	Bases	Quality scores	Total Size
N21	53750474 (6,3%)	157436440 (19,7%)	252127786 (31,5%)	463314700 (18,5%)
C11	124423670 (5,5%)	938021598 (22,7%)	1308203087 (31,7%)	2370648355 (22,3%)
C12	124424019 (5,5%)	951909077 (23,1%)	1551848623 (37,6%)	2628181719 (24,8%)
C21	123801431 (5,5%)	933508524 (22,8%)	1282775960 (31,3%)	2340085915 (22,2%)
C22	123801813 (5,5%)	942813817 (23,0%)	1525230664 (37,2%)	2591846294 (24,6%)

Table 5.1: Compression results of the implemented compression technique (the compression ratios are shown in the parentheses).

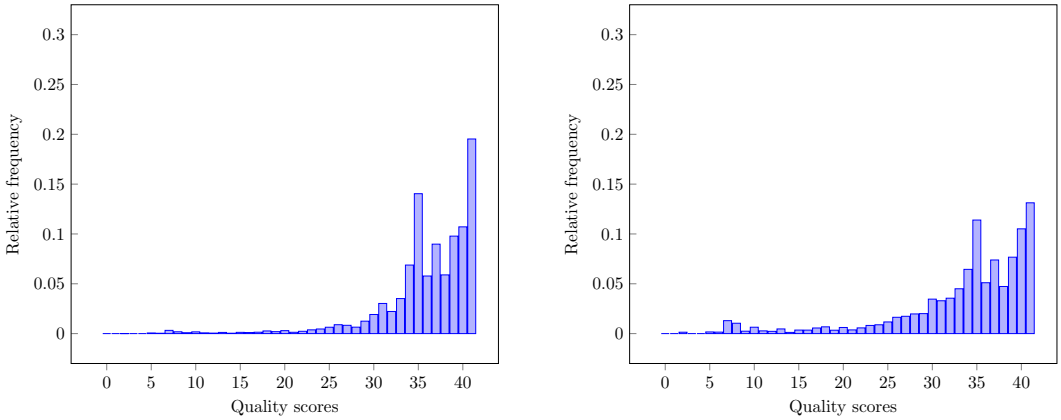

Figure 5.3: Relative frequency distribution of the quality scores in C11 (left) and C12 (right).

different values for the coverage value of the set of reads that they contain. Thus, files with a higher coverage are bound to contain more repetitions of the same subsequences of bases, which the DTMC eventually learns and can predict with a high degree of accuracy. This effect is shown in the compression of the base sequences of N21, which contains a high coverage of a smaller sample than the one sequenced for the C11-C22 benchmarks.

As a final remark, the results show that files with small read sizes are bound to be compressed more than files with larger reads. This is evidenced by the compression ratio achieved on N21 which has a read size of 51 as opposed to the other benchmarks that have a read size of 101. As the read size of a set of reads decreases, the portion of the file that contains id strings increases. For example, in the N21 benchmark, the id strings

take up almost 34% of the file, while in C11, the portion is only 21.1%.

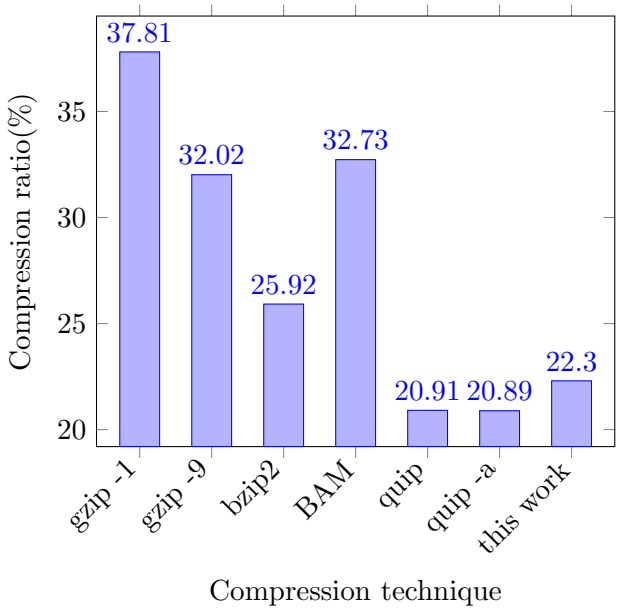

Figure 5.4: Comparison of the compression ratios of this work to popular compressors for the C11 benchmark.

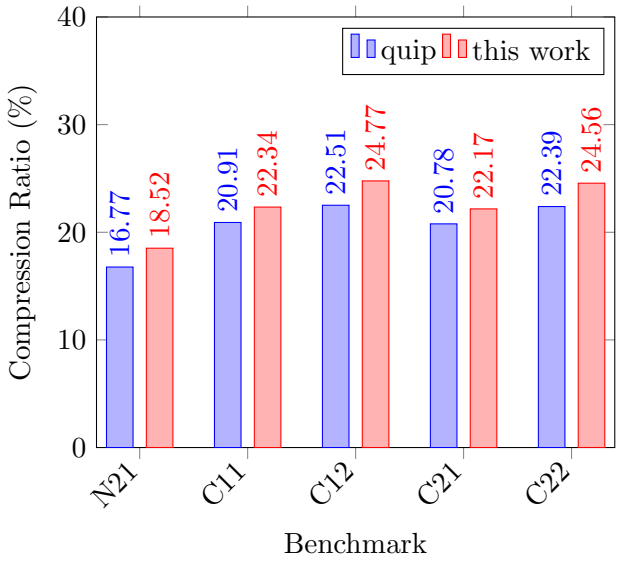

Figure 5.5: Comparison of the compression ratios achieved by quip compared with this work.

Compared to other available techniques, the compressor achieves compression ratios close to the state-of-the-art, as shown in Fig. 5.4. This result is consistent with all benchmarks, where this work achieves a compression ratio within 1.8% on average of

the ratio that is achieved by quip [24], the state-of-the-art compressor (Fig. 5.5). The slightly improved compression performance of quip suggests that the preprocessing of the data may not always beneficial. Preprocessing alters the statistical properties of the data, meaning that even though LPC decreases the 1st order entropy of the sequence of quality scores, it may destroy dependencies between consecutive values, that would be otherwise uncovered with DTMC modelling.

5.2 Compression speed

The compressor and decompressor pair that were implemented aimed at evaluating the compression performance of the selected techniques, and not at creating a fast compressor. Therefore, there is enough room for improvement on this matter. Apart from some decisions in the selection of the techniques for reasonable execution times (like the choice of adaptive LPC instead of blocked LPC, and the creation of 3 programs that are executed in parallel during compressing), no optimizations are performed for improving the caching behaviour or increasing the degree of parallelism. Still, this simple implementation achieves 15.08 MB/s compression speed on average, and 2.1 MB/s decompression speed.

The reason behind the extremely low decompression speed is uncovered by profiling the application, which shows that 75% of the time is spent on the decoding algorithm of the arithmetic coding process (Appendix A.2). In addition, decoding the bases or the quality scores alone can be performed at a speed of almost 7.5 MB/s, which shows that the decoding of the id strings is the bottleneck of the whole program.

The difference that id strings have over the bases and quality scores is the fact that they consist of a large alphabet. This in turn means that the lines 6-13 of the algorithm are repeated many times before the encoded symbol is found, as opposed to the compression process which only executes them once per symbol. These lines consist the most computational intensive lines of code in the whole process, as they contain both multiplications and divisions for 64-bit numbers[1], and therefore repeating over them is bound to decrease the compression speed.

[1]The products $w \cdot d_k(s_j)$ and $w \cdot d_k(s_j)$ must both be 64-bit numbers to avoid overflow

6 Towards real-time compression

Most compression techniques that are used on sequencing data achieve high compression ratios, but the speed with which they compress and decompress are far from achieving transparent compression. Modern solid-state drives achieve reading and writing in speeds of hundreds of MB per second [25], and real-time compression should be able to reach these speeds.

The Computer Engineering laboratory of TUDelft has obtained the Generic WorkQueue Engine (GenWQE), a PCIe Accelerator card from IBM, which can perform DEFLATE[12, 14, 13] compatible compression at reported speeds of 2GB/s. Unfortunately, the drivers for the card are not yet disclosed from the manufacturer, as the card is still in experimental stages of production, but in this chapter the work focuses on investigating whether the card can produce adequate compression ratios based on simulations.

In section 6.1 we present how the card works, and based on that, in section 6.2 we show how to determine the configuration of the card for compressing FASTQ files, and simulate the results that the card would achieve with this configuration.

6.1 The GenWQE card

The GenWQE card performs DEFLATE-compatible compression, which means that the compressed files can be decompressed with any other DEFLATE compatible decompressor. The compression process is similar to regular the regular gzip process (see Section 2.1.1).

First, the LZ77 [49] algorithm is performed on the input data to identify string repetitions which are replaced by a (length, distance) tuple. The data to be compressed consists of length, distance and literal symbols [1], which are represented using 2 distinct alphabets. The first contains the values 0-285, with the first 256 ones corresponding to literals with the same ASCII representation, value number 256 corresponding to the end of block character and values 257-285 corresponding to the lengths in tuples (the lengths take values in the range 3-258, and thus their representation may require some extra bits, as defined in the standard [12]). The second alphabet is used to represent the 32768 different distance values in 30 symbols: 0 through 29. In order to do so, extra bits are utilized after a symbol's code, as defined in the standard.

In general, the DEFLATE standard allows the compressor to encode the literals, lengths and distances of a block of the input data in 3 different ways:

- No compression

[1] Literals are the characters that are encoded as is, since they do not belong to a repeated string.

- Compression with fixed Huffman Trees (defined in the standard)
- Compression with dynamic Huffman Trees

Gzip in general chooses the third method and creates two sets for Huffman codes [22] (one for each alphabet) for each input block, in order to be able to detect local statistical properties of the input file and adapt to them. The GenWQE card, on the other hand, drops this adaptability for the sake of fast compression, and compresses all literals, lengths and distances with Huffman codes that are provided beforehand by the user. This way, no computation of Huffman trees need to be performed, and only one pass over the data is needed for their compression.

Therefore, the card can be configured with a different couple of trees, according to the type of input data it is meant to compress. In the next section we present how these trees can be created for FASTQ files, in a process that can be mimicked for any other type of input file.

6.2 Configuring for FASTQ files

The compression performance of the GenWQE card greatly depends on the selection of the Huffman trees for the literals-lengths and distances alphabets. In order to achieve as good a compression ratio as possible, the card must be provided trees specially designed for the type of input file that it compresses.

The FASTQ file format gives a lot of opportunities for repeated strings of characters; id strings are almost identical and base sequences have already been shown to have a lot of repetitions. Moreover, the set size of reads favours certain distance values, making their probability distribution skewed.

Therefore, in order to create a couple of trees for the GenWQE card that aim at FASTQ compression, we measure the frequency of occurrence of every symbol in the result of the LZ77 algorithm for a set of files of this type. For performing that, the popular open source library *zlib* [18] was modified to report every symbol that is output from the LZ algorithm, and it was subsequently used in a simple implementation of gzip. The set of input files consisted of the five benchmarks of Table 1.2, plus the following six open source files from the GenBank [3] database of the National Center for Biotechnology Information (NCBI): SRR400039.fastq, SRR125858.fastq, SRR359032.fastq, SRR372816.fastq, ERR030867.fastq, and ERR030894.fastq.

Performing the compression with the modified zlib library allows us to measure the frequency of occurrence of every symbol and to calculate Huffman trees for them with the method described in [22] and [12], with the DEFLATE requirement of having a maximum code length of 15 and the card-only requirement of having a code for every symbol, whether it is encountered or not. The calculated trees can be found in Appendix B.

The measured frequencies (and therefore the Huffman trees as well) show that the most frequent literals are in the range 40-74, which correspond to quality scores, meaning that it is harder to find repetitive subsequences in them. As for the length of repetitive subsequences, values between 4 and 58 are most commonly encountered (values 257-275

in the literal-length tree), with the repetition most probably being more than 8193 bytes earlier in the file (since values above 26 are most common in the distance trees).

Even though it is not possible to configure the lab's GenWQE card yet, it is possible to calculate its compression performance, since we know the frequency of occurrence of every symbol ($s_i,\ i \in [0, 285]$ for the literal-length alphabet and $v_i,\ i \in [0, 29]$ for the distance alphabet) and the number of bits that are needed to compress it (which is equal to the number of bits needed for its Huffman codeword (h_j) plus any extra bits defined by the standard (k_j)). Thus, the total number of bits L of the compressed file would be:

$$L = \sum_{i=0}^{285} [f(s_i) * (h_{s_i} + k_{s_i})] + \sum_{i=0}^{29} [f(v_i) * (h_{v_i} + k_{v_i})] \tag{6.1}$$

where $f()$ is the measured frequency of a symbol.

Benchmark	Calculated Comp. ratio (%)	Gzip -9 comp. ratio (%)
N21	29,01	28,67
C11	32,23	32,02
C12	35,84	34,45
C21	32,02	31,80
C22	34,5	34,27

Table 6.1: Simulated compression ratio achieved with the GenWQE card and comparison with gzip -9.

Table 6.1 shows a comparison between the compression ratio achieved with regular gzip (compression level 9) and the calculated compression ratio that the GenWQE card would achieve with the Huffman trees calculated before, on the benchmarks of Table 1.3. The results show that the card would achieve slightly worse compression than gzip, which was expected, since the dynamism of different Huffman codes throughout the file was dropped. Nevertheless, the difference is not high, with the highest observed difference at 1.39%.

Therefore, the GenWQE card with this configuration can definitely be used for high-throughput compression for FASTQ files, as it achieves compression very close to regular gzip. It should be noted however that the Huffman codes that were created aim only at FASTQ compression, but the same methodology that was utilized to create them can be used for other file formats as well, like SAM or FASTA.

Conclusions and recommendations

7

7.1 Conclusions

The DNA sequencing process is the backbone of modern genetics, the process of determining the precise order of nitrogenous bases within a strand of the DNA molecule. Recent advances in sequencing technology have exponentially increased the speed of the process and brought forth the Next-Generation sequencing. This rapid expansion of data generation, causes the need in storage space to increase by the same rate. This means that if the same trend persists, the cost of a DNA sequencing pipeline will be dominated by the storage cost, rather than the sequencing itself. In order to alleviate this problem, efficient compression is imperative. However, the generic compression techniques, that are used nowadays, such as gzip and bzip2, cannot provide high compression ratios. Instead, specialized solutions must be investigated, which take under consideration unique statistical properties of this kind of data to achieve superior compression.

In this thesis, we investigate techniques for achieving high compression ratios on FASTQ files, while at the same time keeping the compression speed in manageable levels. In order to achieve better compression performance than generic techniques, the input files have to be disintegrated to expose the three different kind of information that they hold: the identifier strings, the sequences of bases and the bases of quality scores. The final goal is achieved by identifying the statistical properties of every specific kind of information so as to use a suitable compression method.

The sequences of quality scores are most difficultly compressed, since they have the highest information content. Two compression methods are investigated. The first uses linear prediction coding as a preprocessing step to transform the sequence of quality scores to a sequence of prediction errors, with reduced entropy. Subsequently, the information source of the prediction errors is modelled with a discrete time Markov chain, and compressed with an arithmetic coder. The second method skips the preprocessing step and attempts to model the information source of the quality scores with hidden Markov models, and in turn compress using again an arithmetic coder for entropy coding. The comparison of the two techniques showed that the LPC method was superior to both compression performance (31.2% compared to 42.4% for the second method) and speed (15.08 MB/s compared to 0.5 MB/s), and in addition was more robust in respect with design parameters. The hidden Markov models on the other hand required too much computation resources, while at the same time failed to model the source of the quality scores adequately.

The identifier strings of the reads proved much more highly compressible, since there exists a huge amount of redundant information between the ids of consecutive reads. Therefore, using a simple parsing and delta encoding preprocessing step, complemented by Markov chain modelling and arithmetic coding managed to compress to approximately

5.5% of its initial size, needing in average 0.4 bits per symbol.

The sequences of bases are in between ids and quality scores in terms of information content. As the alphabet size of these sequences is small, the Burrows-Wheeler transform was investigated for transforming the data in a more highly compressible form. However, the results show that the transform has the opposite result; the sequences of bases are highly structured by nature and therefore predictable, while the result of the transform is more unpredictable. As a result, the use of a high order Markov chain for modelling along with arithmetic coding, led to better compression with a 19.7% ratio, and at the same time, better timing performance.

Furthermore, a compression/decompression pair was implemented using the techniques that proved best for each of the three kinds of information. The implementation achieved compression performance very close to state-of-the-art compressors, compressing files close to 19% of their original size.

Finally, we investigated how a GenWQE compression card can be used to achieve moderate compression at very high speeds (reportedly 2 GB/s), conforming to the DEFLATE standard. We created specialized Huffman codes for FASTQ files, that can be provided to the card, resulting in estimated compression ratios at maximum 1.4% worse than the highest compression level of gzip.

7.2 Recommendations

As this thesis focused on achieving high compression for FASTQ files, further research can move towards two main directions: improving the compression performance further, or improving the compression speed.

For improving the compression, research should focus in increasing the compression on quality scores. A very promising idea lies in the fact that each quality score corresponds to a nucleotide base, and therefore there may exist some correlation between specific sequences of bases and sequences of quality scores. This may seem counter-intuitive, in the sense that quality scores depend on the quality of the sample and not on the sequence of the bases, yet, it would not be unimaginable if the sequencing process 'favours' some base sequences in terms of quality, or 'struggles' with others.

Moreover, the compression could be greatly improved if the possibility for lossy compression was present. Yet, this requires the knowledge of the whole sequencing pipeline, and the certainty that losing information does not influence downstream applications.

In terms of improving the compression speed, there is a lot of room for improvement. Achieving transparent processing through hardware implementations of specialized compression methods, would surpass the obvious disadvantage of the GenWQE card (or similar general compression hardware implementations), which is the reduced compression performance. Indeed, the specialized compressor that was presented in this work is an example of a streaming application, that is a good candidate for FPGA implementation.

Finally, the specialized compression techniques could be extended to the increasingly popular SAM file format, which stores a superset of the data of FASTQ files.

Bibliography

[1] D. Adjeroh, T. Bell, and A. Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[2] L. Baum and G. Sell. Growth transformations for functions on manifolds. *Pacific Journal of Mathematics*, 27(2):211–227, 1968.

[3] D. Benson, I. Karsch-Mizrachi, D. Lipman, J. Ostell, and E. Sayers. GenBank. *Nucleic acids research*, 37(Database issue):D26–31, January 2009.

[4] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Commun. ACM*, 29(4):320–330, April 1986.

[5] J. Bonfield and M. Mahoney. Compression of FASTQ and SAM Format Sequencing Data. *PLoS ONE*, 8(3), 2013.

[6] T. Brown. *Genomes 3*. Garland Science, 2006.

[7] M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital SRC Research Report, 1994.

[8] P. Cock, C. Fields, N. Goto, M. Heuer, and P. Rice. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Res*, 38(6):1767–1771, April 2010.

[9] F. Costa. Big data in biomedicine. *Drug Discovery Today*, October 2013.

[10] A. Cox, M. Bauer, T. Jakobi, and G. Rosone. Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform. *CoRR*, abs/1205.0192, 2012.

[11] S. Deorowicz and S. Grabowski. Data compression for sequencing data. *Algorithms for Molecular Biology*, 8:25, 2013.

[12] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, May 1996. http://www.ietf.org/rfc/rfc1951.txt.

[13] P. Deutsch. GZIP file format specification version 4.3. RFC 1952, May 1996. http://www.ietf.org/rfc/rfc1952.txt.

[14] P Deutsch and J. Gailly. ZLIB Compressed Data Format Specification version 3.3. Internet RFC 1950, May 1996. http://www.ietf.org/rfc/rfc1950.txt.

[15] A. Drake. Discrete-state markov processes. In *Fundamentals of applied probability theory*, chapter 5. McGraw-Hill, 1988.

[16] M. Fritz, R Leinonen, G. Cochrane, and E. Birney. Efficient storage of high throughput sequencing data using reference-based compression. *Genome Research*, 21:734–740, 2013.

[17] J. Gailly. GNU Gzip documentation. `http://www.gnu.org/software/gzip/manual/gzip.html`, 2013. [Online; accessed 1-July-2014].

[18] J. Gailly and M. Adler. zlib: A Massively Spiffy Yet Delicately Unobtrusive Compression Library. `http://www.zlib.net/`, 2014. [Online; accessed 15-August-2014].

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

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

[21] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321–, July 1961.

[22] D. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952.

[23] International Organization for Standardization. Information technology — computer graphics and image processing — portable network graphics (png): Functional specification. ISO/IEC 15948:2004, 2004.

[24] D. Jones, W. Ruzzo, X Peng, and M. Katze. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *CoRR*, abs/1207.2424, 2012.

[25] M. Jung and M. Kandemir. Revisiting Widely Held SSD Expectations and Rethinking System-level Implications. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, pages 203–216, New York, NY, USA, 2013. ACM.

[26] C. Kozanitis, C Saunders, S. Kruglyak, V. Bafna, and G. Varghese. Compressing Genomic Sequence Fragments Using SlimGene. *Journal of Computational Biology*, 18(3):401–413, 2011.

[27] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and 1000 Genome Project Data Processing Subgroup. The Sequence Alignment/Map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, August 2009.

[28] P. Li, X. Jiang, S. Wang, J. Kim, H. Xiong, and L. Ohno-Machado. HUGO: Hierarchical mUlti-reference Genome cOmpression for aligned reads. *JAMIA*, 21(2):363–373, 2014.

[29] M. L. Metzker. Sequencing technologies - the next generation. *Nat Rev Genet*, 11(1):31–46, January 2010.

[30] J. Norris. *Markov chains*. Cambridge series in statistical and probabilistic mathematics. Cambridge University Press, 1998.

[31] I. Pavlov. LZMA SDK (Software Development Kit). `http://7-zip.org/sdk.html/`. [Online; accessed 14-July-2014].

[32] L. Rabiner. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[33] L. Rabiner and R. Schafer. *Digital Processing of Speech Signals*. Englewood Cliffs: Prentice Hall, 1978.

[34] J. Reece, L. Urry, M. Cain, S. Wasserman, P. Minorsky, and R. Jackson. *Campbell Biology (9th Edition)*. Benjamin Cummings, 9 edition, October 2010.

[35] I. Richardson. *The H.264 Advanced Video Compression Standard*. Wiley, 2011.

[36] M. Ronaghi, M. Uhlén, and P. Nyrén. A Sequencing Method Based on Real-Time Pyrophosphate. *Science*, 281(5375):363–365, July 1998.

[37] A. Said. *Introducing to Arithmetic Coding - Theory and Practice*. HPL-2004-76. Imaging Systems Laboratory, HP Laboratories Palo Alto, April 2004.

[38] F Sanger, S Nicklen, and AR Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of The National Academy of Sciences of The United States Of America*, 74:5463–5467, 1977.

[39] K. Sayood. *Lossless Compression Handbook*. Communications, Networking and Multimedia. Elsevier Science, 2002.

[40] K. Sayood. *Introduction to Data Compression, Third Edition (Morgan Kaufmann Series in Multimedia Information and Systems)*. Morgan Kaufmann, Third edition, December 2005.

[41] J. Seward. bzip2 official website. `http://www.bzip.org/`. [Online; accessed 1-July-2014].

[42] J. Seward. bzip2 and libbzip2, version 1.0.5: A program and library for data compression, documentation. `http://www.bzip.org/1.0.5/bzip2-manual-1.0.5.html`, 2007. [Online; accessed 1-July-2014].

[43] C. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, July, October 1948.

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

[45] P. Strobach. *Linear Prediction Theory: A Mathematical Basis for Adaptive Systems*. Springer series in information sciences. Springer-Verlag, 1990.

[46] The SAM/BAM Format Speicification Working Group. Sequence Alignment/Map Format Specification. `http://samtools.github.io/hts-specs/SAMv1.pdf`, 2014. [Online; accessed 1-July-2014].

[47] C. Walter. Kryder's Law. *Scientific American*, August 2005.

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

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

Algorithms for Arithmetic Coding

A

Definitions:

P = Integer representation precision
WHOLE = $2^P - 1$
HALF = $WHOLE/2$
QUARTER = $WHOLE/4$
N = the size of the sequence to be encoded
$S = s_1 s_2 ... s_N$: the sequence to be encoded over the alphabet $\mathcal{U},\ |\mathcal{U}| = M$
$V = v_1 v_2 ... v_L$: the sequence of bits that consist the codeword to be decoded
$p_k(u)$ is represented as frequencies of occurrence:

$$p_k(u) = \frac{r_k(u)}{R_k},\ R_k = \sum_{i=0}^{M} r_k(i),\ r_k(i) \in \mathbb{Z}$$

$c_k(0) = 0$
$c_k(j) = \sum_{i=0}^{j-1} r_k(i),\ j = 1, ..., M,\ k = 1, ..., N$
$d_k(j) = c_k(j) + r_k(j),\ j = 1, ..., M,\ k = 1, ..., N$

A.1 Encoding Algorithm

Algorithm: Arithmetic Encoding

Input: N, S, $c_k()$, $d_k()$ $\forall k \in [1, N]$
Output: emitted bits

```
// Initialize:
a ←0, b ←WHOLE, s ←0;
// Iterate over the sequence:
for k ← 1 to N do
    // Calculate the new interval:
    w ←b-a;
    b ←a +round(w ·d_k(s_k)/R_k);
    a ←a +round(w ·c_k(s_k)/R_k);
    // Rescaling: Mappings E_1 and E_2
    while b <HALF or a >HALF do
        if b <HALF then
            emit 011...1 (s ones), s ← 0;
            a ← 2·a, b ← 2·b;
        else if a >HALF then
            emit 100...0 (s zeros), s ←0;
            a ← 2 · (a −HALF), b ← 2 · (b −HALF);
    // Rescaling: Mapping E_3
    while a >QUARTER and b <3·QUARTER do
        s ←s +1;
        a ← 2 · (a −QUARTER), b ← 2 · (b −QUARTER);
// Finalize:
s ←s +1;
if a ≤QUARTER then
    emit 011...1 (s ones)
else
    emit 100...0 (s zeros)
```

A.2 Decoding Algorithm

Algorithm: Arithmetic Decoding

Input: N, V, $c_k()$, $d_k()$ $\forall k \in [1, N]$
Output: S

```
// Initialize:
a ←0, b ←WHOLE, z ←0 i ←P+1;
z ← (v1v2...vP)2; // Read the first P bits
// Iterate over the sequence:
for k ← 1 to N do
    // Calculate the new interval for all symbols:
    for j ← 1 to M do
        w ←b −a;
        b 0 ←a +round(w ·dk(sj)/Rk);
        a 0 ←a +round(w ·ck(sj)/Rk);
        if a 0 ≤ z <b 0 then
            sk ← sj;
            a ←a 0, b ←b 0;
            break;
    // Rescaling: Mappings E1 and E2
    while b <HALF or a >HALF do
        if b <HALF then
            a ← 2·a, b ← 2·b  z ← 2·z;
        else if a >HALF then
            a ← 2 · (a −HALF), b ← 2 · (b −HALF), z ← 2 · (z −HALF);
        // Read the next codeword bit:
        if vi=1 then z ←z +1;
        i ←i +1;
    // Rescaling: Mapping E3
    while a >QUARTER and b <3·QUARTER do
        a ← 2 · (a −QUARTER), b ← 2 · (b −QUARTER);
        z ← 2 · (z −QUARTER);
        // Read the next codeword bit:
        if vi=1 then z ←z +1;
        i ←i +1;
```

Huffman trees for GenWQE

B

Value	Code	Code size (bits)
0	07FE	11
1	00F6	8
2	07FF	11
3	03FE	10
4	00F7	8
5	01FE	9
6	00F8	8
7	00F9	8
8	00FA	8
9	00FB	8
10	00FC	8
11	00FD	8
12	00FE	8
13	007A	7
14	003A	6
15	003B	6
16	18	5
17	003C	6
18	19	5
19	001A	5
20	001B	5
21	001C	5
22	8	4
23	9	4
24	000A	4
25	000B	4
26	0	3
27	1	3
28	2	3
29	3	3

Table B.1: Distance Huffman code

Value	Code	Code size (bits)	Value	Code	Code size (bits)	Value	Code	Code size (bits)
0	7F24	15	101	7F5F	15	202	7FC4	15
1	7F25	15	102	7F60	15	203	7FC5	15
2	7F26	15	103	7F61	15	204	7FC6	15
3	7F27	15	104	7F62	15	205	7FC7	15
4	7F28	15	105	7F63	15	206	7FC8	15
5	7F29	15	106	7F64	15	207	7FC9	15
6	7F2A	15	107	7F65	15	208	7FCA	15
7	7F2B	15	108	7F66	15	209	7FCB	15
8	7F2C	15	109	7F67	15	210	7FCC	15
9	7F2D	15	110	7F68	15	211	7FCD	15
10	7F2E	15	111	7F69	15	212	7FCE	15
11	7F2F	15	112	7F6A	15	213	7FCF	15
12	7F30	15	113	7F6B	15	214	7FD0	15
13	7F31	15	114	7F6C	15	215	7FD1	15
14	7F32	15	115	7F6D	15	216	7FD2	15
15	7F33	15	116	7F6E	15	217	7FD3	15
16	7F34	15	117	7F6F	15	218	7FD4	15
17	7F35	15	118	7F70	15	219	7FD5	15
18	7F36	15	119	7F71	15	220	7FD6	15
19	7F37	15	120	7F72	15	221	7FD7	15
20	7F38	15	121	7F73	15	222	7FD8	15
21	7F39	15	122	7F74	15	223	7FD9	15
22	7F3A	15	123	7F75	15	224	7FDA	15
23	7F3B	15	124	7F76	15	225	7FDB	15
24	7F3C	15	125	7F77	15	226	7FDC	15
25	7F3D	15	126	7F78	15	227	7FDD	15
26	7F3E	15	127	7F79	15	228	7FDE	15
27	7F3F	15	128	7F7A	15	229	7FDF	15
28	7F40	15	129	7F7B	15	230	7FE0	15
29	7F41	15	130	7F7C	15	231	7FE1	15
30	7F42	15	131	7F7D	15	232	7FE2	15
31	7F43	15	132	7F7E	15	233	7FE3	15
32	7F44	15	133	7F7F	15	234	7FE4	15
33	1FC6	13	134	7F80	15	235	7FE5	15
34	7F45	15	135	7F81	15	236	7FE6	15
35	03F4	10	136	7F82	15	237	7FE7	15
36	7F46	15	137	7F83	15	238	7FE8	15
37	1FC7	13	138	7F84	15	239	7FE9	15
38	03F5	10	139	7F85	15	240	7FEA	15
39	03F6	10	140	7F86	15	241	7FEB	15
40	00F4	8	141	7F87	15	242	7FEC	15
41	00F5	8	142	7F88	15	243	7FED	15
42	01F4	9	143	7F89	15	244	7FEE	15
43	00F6	8	144	7F8A	15	245	7FEF	15
44	01F5	9	145	7F8B	15	246	7FF0	15
45	01F6	9	146	7F8C	15	247	7FF1	15
46	00F7	8	147	7F8D	15	248	7FF2	15
47	01F7	9	148	7F8E	15	249	7FF3	15
48	68	7	149	7F8F	15	250	7FF4	15
49	69	7	150	7F90	15	251	7FF5	15
50	006A	7	151	7F91	15	252	7FF6	15
51	006B	7	152	7F92	15	253	7FF7	15
52	006C	7	153	7F93	15	254	7FF8	15
53	006D	7	154	7F94	15	255	7FF9	15
54	006E	7	155	7F95	15	256	3F90	14
55	006F	7	156	7F96	15	257	11	5
56	28	6	157	7F97	15	258	2	4
57	29	6	158	7F98	15	259	3	4
58	002A	6	159	7F99	15	260	4	4
59	002B	6	160	7F9A	15	261	0	3
60	002C	6	161	7F9B	15	262	5	4
61	002D	6	162	7F9C	15	263	12	5
62	002E	6	163	7F9D	15	264	72	7
63	002F	6	164	7F9E	15	265	73	7
64	000C	5	165	7F9F	15	266	74	7
65	000D	5	166	7FA0	15	267	75	7
66	000E	5	167	7FA1	15	268	76	7
67	000F	5	168	7FA2	15	269	13	5
68	30	6	169	7FA3	15	270	77	7
69	31	6	170	7FA4	15	271	00F8	8
70	32	6	171	7FA5	15	272	01F9	9
71	10	5	172	7FA6	15	273	78	7
72	70	7	173	7FA7	15	274	79	7
73	71	7	174	7FA8	15	275	00F9	8
74	01F8	9	175	7FA9	15	276	07EF	11
75	7F47	15	176	7FAA	15	277	07F0	11
76	7F48	15	177	7FAB	15	278	0FE2	12
77	7F49	15	178	7FAC	15	279	7FFA	15
78	07EE	11	179	7FAD	15	280	7FFB	15
79	7F4A	15	180	7FAE	15	281	7FFC	15
80	7F4B	15	181	7FAF	15	282	3F91	14
81	7F4C	15	182	7FB0	15	283	7FFD	15
82	7F4D	15	183	7FB1	15	284	7FFE	15
83	7F4E	15	184	7FB2	15	285	7FFF	15
84	33	6	185	7FB3	15			
85	7F4F	15	186	7FB4	15			
86	7F50	15	187	7FB5	15			
87	7F51	15	188	7FB6	15			
88	7F52	15	189	7FB7	15			
89	7F53	15	190	7FB8	15			
90	7F54	15	191	7FB9	15			
91	7F55	15	192	7FBA	15			
92	7F56	15	193	7FBB	15			
93	7F57	15	194	7FBC	15			
94	7F58	15	195	7FBD	15			
95	7F59	15	196	7FBE	15			
96	7F5A	15	197	7FBF	15			
97	7F5B	15	198	7FC0	15			
98	7F5C	15	199	7FC1	15			
99	7F5D	15	200	7FC2	15			
100	7F5E	15	201	7FC3	15			

Table B.2: Literal-length Huffman code