Lossless Compression Methods for Archiving Nanopore DNA Signal Data

SASHA JENNER

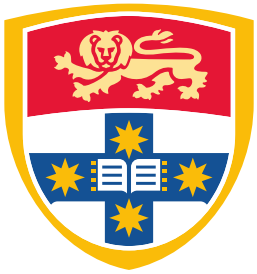
SID: 490390494

Supervisor: Dr. John Stavrakakis
Associate Supervisor: Dr. Hasindu Gamaarachchi
Associate Supervisor: Dr. Ira Deveson

This thesis is submitted in partial fulfillment of
the requirements for the degree of
Bachelor of Science and Bachelor of Advanced Studies (Honours)

School of Computer Science
The University of Sydney
Australia

6 November 2022



THE UNIVERSITY OF
SYDNEY

Student Plagiarism: Compliance Statement

I certify that:

I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure;

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to the University commencing proceedings against me for potential student misconduct under Chapter 8 of the University of Sydney By-Law 1999 (as amended);

This Work is substantially my own, and to the extent that any part of this Work is not my own I have indicated that it is not my own by Acknowledging the Source of that part or those parts of the Work.

Name: Sasha Jenner

Signature: 

Date: November 6, 2022

Abstract

Lossless compression of nanopore DNA signal data with more space saving than the state-of-the-art is highly desirable due to the magnitude and rate of research and clinical nanopore data being produced. Nanopore DNA sequencing produces a characteristic signal which represents the ionic current over time as a molecule's DNA strand passes through a nanoscale protein pore. This signal is used to determine the order of DNA nucleobases (A, C, T & G) in the molecule; useful for medical diagnosis and forestic biology amongst other applications. Data compression is necessary for reducing the signal's long-term storage costs which are estimated to be in the order of an extra $6000 each year for a medium clinical research facility.

In this thesis, we provide a systematic analysis of nanopore DNA signal data including the articulation of its characteristic features and transformations. The problem space of suitable compression methods is theorised and several novel approaches to compressing nanopore signal data are detailed including a novel variable byte encoding vbbe21, a static Huffman encoding, optimal subsequence searching, stall encoding and separation into the jumps, falls and flats. The first comprehensive benchmark of existing and novel compression methods for nanopore signal data is conducted on the NA12878 data set. A new state-of-the-art space saving of 0.666 is achieved using dstall-fz compared to the previous best of 0.659. This combines vbbe21 with dynamic stall encoding and range coding modelled by order-0 order-1 secondary symbol estimation (SSE).

For archiving purposes, it is advised to use the dstall-fz-1500 method which closely approximates the decision boundary of dstall-fz in half the compression time – achieving a space saving of 0.666. For faster compression and decompression speed than dstall-fz-1500 but better compression than the previous state-of-the-art, rc1-vbbe21-zd is recommended which achieves a space saving of 0.661. If analysis speed is the utmost priority, it is advised to continue using the previous state-of-the-art zstd-svb-zd until faster implementations of the articulated methods are available.

Acknowledgements

First and foremost, I would like to thank my supervisors for guiding my research and providing useful feedback on my writing. Dr John Stavrakakis, thank you for always motivating me to do better and inspiring me with new ideas. Dr Ira Deveson, I am very grateful that you got me back into the office and always value your feedback. Dr Hasindu Gamaarachchi, thank you for helping me with any technical problems I faced and for your quality humour. Also, thank you Daniel Lemire for generously providing the LaTeXsource code for your Stream VByte paper (Lemire et al., 2018) so that I could modify and include your illustrative figures. Finally, a big thanks to the computer science faculty for the "CS Honours Room" which was always silent and had a nice view.

CONTENTS

Student Plagiarism: Compliance Statement ii

Abstract iii

Acknowledgements iv

List of Figures vii

List of Tables x

Chapter 1 Introduction 1

1.1 Contributions 3

1.2 Thesis Outline 3

Chapter 2 Literature Review 4

2.1 Nanopore Sequencing 4

2.2 Data Compression 5

2.2.1 Introduction 5

2.2.2 Entropy Coding 8

2.2.3 Dictionary Coding 9

2.2.4 Other Encodings 10

2.2.5 Nanopore Compression 14

2.3 The State-of-the-Art 15

Chapter 3 The Data 18

3.1 Introduction 18

3.2 Characteristics 24

3.3 Transformations 27

Chapter 4 Problem Space 33

Chapter 5 Methodology 37

5.1 One Byte, Two Byte Exceptions 37
5.1.1 Exceptions Encoding 40
5.1.2 Huffman 45
5.1.3 Range Coding 47
5.2 Exploiting the Signal 49
5.2.1 Subsequence Searching 49
5.2.2 The Stall 55
5.2.3 The DNA Section 58

Chapter 6 Results **65**

Chapter 7 Discussion **75**
7.1 Experiments 75
7.2 Future Work 77

Chapter 8 Conclusion **79**

Appendix **80**
8.1 Subsequence Searching 80

Bibliography **81**

List of Figures

1.1 The MinION is a highly portable nanopore sequencing device. 1

1.2 A DNA molecule passing through a nanopore in an electrolytic solution. 2

2.1 The MinION recording nanopore signal data as a DNA molecule is unwound and ratcheted through the nanopore. 5

2.2 Shannon's general communication system. 6

2.3 The sequence of numbers $1024, 12, 10, 524\,288$ in the bit packed format. 11

2.4 An example of $1024, 12, 10, 1\,048\,576, 0, 1, 2, 1024$ compressed with classical and 0-based Stream VByte. 13

2.5 Compressed 16-bit Stream VByte bytes for $1024, 12, 10, 4096, 0, 1, 2, 1024$. 13

2.6 The zig-zag encoding applied to integers -15 to 15. 17

3.1 The read with ID e9f08690-171f-476f-9119-5330d0290126 from the NA12878 data set. 18

3.2 The frequency of raw signal values. 22

3.3 Histogram of the number of data points in (length of) each read for the data set. 23

3.4 The first 1500 data points of the read with ID e9f08690-171f-476f-9119-5330d0290126 split into four sections: surge, stall, pre-adapter surge and adapter sequence. 25

3.5 An example of 200 data points from a DNA section in the read with ID e9f08690-171f-476f-9119-5330d0290126. 26

3.6 An example of a homopolymer section (repetition of 'T') from the read with ID e9f08690-171f-476f-9119-5330d0290126. 27

3.7 An example of 3800 data points from a slow section in the read with ID 99671b17-feb4-492b-b119-77daf8e5794e. 28

3.8 An example of an incongruent section. 29

3.9 The read with ID 515e4fd0-8ab1-4845-8866-6772e779712b from the data set. 30

3.10 The deltas of the read with ID e9f08690-171f-476f-9119-5330d0290126. 30

3.11 The frequency of raw signal deltas. 31

3.12 The zig-zag deltas of the read with ID e9f08690-171f-476f-9119-5330d0290126. 31

3.13 The frequency of raw signal zig-zag deltas. 32

5.1 The vbe21 encoding. 37

5.2 An example of $1024, 12, 10, 4096, 0, 1, 2, 1024$ encoded with vbe21. 38

5.3 Histogram of the number of exceptions per read. 41

5.4 Histogram of the zig-zag delta exceptions. 42

5.5 The compact vbbe21 encoding. 43

5.6 The vbbe21 encoding. 43

5.7 The naive encoding of the Huffman table. 45

5.8 The Huffman code length of each one-byte zig-zag delta from the shared Huffman table. 47

5.9 The stall from the read with ID e9f08690-171f-476f-9119-5330d0290126. 55

5.10 The stall encoding. 57

5.11 The dstall encoding. 57

5.12 A scatter plot of the compression methods with the highest compression ratio on each read out of rc01s-vbbe21-zd and stall-fz. 58

5.13 An example of 300 data points from a DNA section in the read with ID e9f08690-171f-476f-9119-5330d0290126. 59

5.14 300 data points from a DNA section in the read with ID e9f08690-171f-476f-9119-5330d0290126. 61

5.15 300 data points from a DNA section in the read with ID e9f08690-171f-476f-9119-5330d0290126. 62

5.16 The jumps encoding. 63

5.17 The alternative jumps encoding. 64

6.1 The compression ratio of each method on the data. 69

6.2 The space saving of each method on the data. 69

6.3 The average number of bits used per symbol and total compressed size of each method on the data. 72

6.4 The (de)compression time (in hours per TiB) versus space saving of various methods. 73

6.5 A scatter plot of the methods which have a space saving greater than or equal to the state-of-the-art. 74

List of Tables

2.1 The number of bytes used and their control code for different supported integer ranges in classical 32-bit, 0-based and 16-bit Stream VByte. 12

2.2 A timeline of the major events in the literature. 15

3.1 Some constant metadata of the NA12878 data set. 19

3.2 The variable metadata of the read with ID e9f08690-171f-476f-9119-5330d0290126 from the NA12878 data set. 19

3.3 The original NA12878 data set. 21

3.4 The downsampled NA12878 data set. 21

3.5 Summary statistics of the data's raw signal values. 22

3.6 Summary statistics of the data's read lengths. 23

3.7 Summary statistics of the data's raw signal values (None) and its various transformations. 26

3.8 The entropy of the data and its transformations. 32

5.1 Summary statistics of the number of exceptions per read and the zig-zag delta exceptions themselves in the data. 40

5.2 The estimated expected size of the exceptions section for encodings vbe21-zd, vbbe21-zd and compact vbbe21-zd on the data. 44

5.3 Summary statistics of the data's stall lengths. 56

6.1 Specifications of the server used for the experiments. 65

6.2 The compression ratio, bits used per symbol and compressed size (in GiB) of the data set after compressing each read sequentially using each of the given methods. 68

6.3 The total compression and decompression time in hours per TiB and the compression ratio after compressing each read from the data sequentially using each of the given methods. It is sorted by the compression time (fastest to slowest). 70

6.4 The total compression and decompression time in hours per TiB and the compression ratio after compressing each read from the data sequentially using each of the given methods. It is sorted by the decompression time (fastest to slowest). 71

6.5 A two-way table of the space saving after applying a compression method in the first layer followed by one in the second layer. 72

CHAPTER 1

Introduction

The DNA sequence of an organism encodes the information necessary for its survival and reproduction. Because of its key role in living organisms, DNA sequencing has become a burgeoning field in both academic research and clinical medicine. Nanopore sequencing is one such approach used to determine the order of DNA nucleobases (A, C, T and G) in a molecule.

Nanopore sequencing records the ionic current as a DNA molecule passes through a nanoscale protein pore (or *nanopore*). This process is depicted in Figure 1.2. Disturbances in the ionic current signal are used by computational algorithms to determine the DNA sequence. Oxford Nanopore Technologies (ONT) is the leading producer of nanopore sequencing machines – such as the portable MinION shown in Figure 1.1.

The problem is that nanopore sequencing produces a huge volume of data which is being recorded at an increasing rate and often needs to be archived for several years due to certain data regulations. For instance, the Genomic Technologies Group at the Garvan Institute of Medical Research regularly perform between 5 and 10 clinical human DNA sequencing runs a week which need to be archived for up to 10 years. This amounts to between 260 and 520 terabytes of new data each year, which costs up to USD $6000 a year to archive using Google Cloud Storage. Over 10 years, this amount of data would accumulate to almost 28 petabytes or roughly USD $330 000 in storage costs.

FIGURE 1.1. The MinION is a highly portable nanopore sequencing device manufactured by ONT which weighs only 450g.

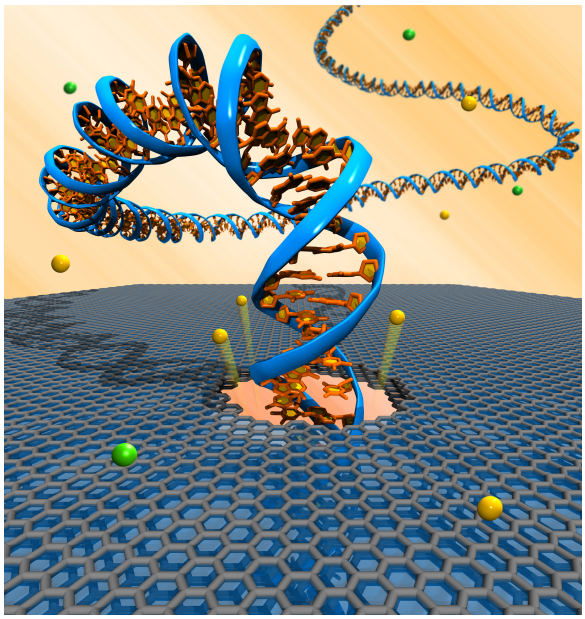


FIGURE 1.2. A DNA molecule passing through a nanopore in an electrolytic solution.

Data compression solves this problem by identifying and removing statistical redundancies in the data and hence decreasing its overall size. Compression methods which achieves this without losing any information are known as *lossless*. These methods can then be reversed to decompress the compressed data and reobtain the original information. One such lossless compression method, known as entropy encoding, attempts to approach the entropy of the data – the level of uncertainty inherent in the data's distribution – by replacing each symbol with a binary code. Huffman coding is a famous example of entropy encoding which uses the probability distribution of the symbols to construct optimal-length binary codes.

The state-of-the-art method used to compress nanopore signal data begins by taking the differences between each successive point. These differences, or *deltas*, are then transformed to the positive integers by the zig-zag encoding. Each zig-zag delta is recorded using the minimum number of bytes possible in an encoding known as Stream VByte. Then, this is compressed using zstd – a very fast compression algorithm which combines entropy encoding with dictionary-matching. This method reduces the data to 34.1% of its original size. However, it remains to discover whether more space can be saved through a better understanding of nanopore signal data.

1.1 Contributions

The first systematic analysis of nanopore DNA signal data architecture was conducted including the articulation of characteristic features as relevant to data compression. A novel measure for comparing the practical suitability of two compression methods is proposed.

A new state-of-the-art compression method was designed which reduces the data to 33.4% of its original size (0.7% better than the previous state-of-the-art). It combines a novel encoding of the signal data's zig-zag deltas known as vbbe21 with dynamic domain-specific stall encoding and predictive range coding. Three range coding model predictors – order-0, order-1 and secondary symbol estimation (SSE) – were evaluated and found to reduce the data to 34.1%, 33.9% and 33.4% of its original size respectively. An alternative static Huffman compression method is proposed which also outperforms the state-of-the-art in terms of space reduction, reducing the data to 33.9% of its original size.

The first detailed and comprehensive benchmark of nanopore signal data compression, including both existing and novel methods, was conducted and released for public access.

1.2 Thesis Outline

To begin with, we provide a detailed introduction to nanopore sequencing, information theory and data compression (Chapter 2). Then, we explore the nature of the data through a systematic analysis which involves identifying its characteristics and understanding its relevant transformations (Chapter 3).

Next, the problem space is defined with respect to the requirements of practicing bioinformaticians and a mathematical metric for comparing the suitability of two compression methods is presented (Chapter 4). Afterwards, several novel encodings and compression methods for nanopore signal data are proposed and detailed, including the evaluation of their running times and space savings (Chapter 5).

The results of a comprehensive benchmark of the existing and novel methods are then contextualised, presented and visualised (Chapter 6). Following this is a detailed discussion of the results including their impact and limitations, as well as an overview of future work in the landscape of nanopore signal compression (Chapter 7). Finally, we conclude by summarising the crucial findings of this thesis (Chapter 8).

CHAPTER 2

Literature Review

Nanopore sequencing is a sub-topic of sequencing and gene informatics, all of which belong to the broader field of bioinformatics. On the other hand, data compression – also known as source coding – is founded on the principles of information theory which lies at the intersection of statistics, computer science and information engineering. In this chapter, we begin with a review of nanopore sequencing and ONT sequencing devices. The data compression fundamentals follow this, starting from the principles of information theory and concluding with the state-of-the-art compression method used on nanopore signal data.

2.1 Nanopore Sequencing

The primary interest of bioinformatics is to understand biological data. This is typically achieved through the development of biology-aware software tools. Nanopore sequencing, which emerged in the 1980s, is one approach employed in bioinformatics to determining the primary structure of biopolymers such as DNA and RNA (Deamer et al., 2016). It involves recording the ionic current as a biopolymer passes through a nanoscale protein pore (or *nanopore*) with a voltage applied to its surrounding membrane (Wang et al., 2021). See Figure 2.1 for a pictorial overview of nanopore sequencing. The resulting ionic current signal is then used to determine the nucleotide sequence of the biopolymer.

Oxford Nanopore Technologies (ONT) is the leading producer of nanopore sequencing machines and is used in most peer-reviewed studies involving nanopore sequencing. Several ONT sequencing machines are commercially available including (in order of throughput) the MinION, GridION and PromethION. Each machine uses one or many flow cells to sequence DNA or RNA molecules. Each *flow cell* contains thousands of nanopores which are grouped into fours with each group corresponding to its own channel. Each channel is assigned an electrode in the flow cell's sensor chip which is controlled and measured by an application-specific integrated circuit (ASIC) chip (Wang et al., 2021). The nanopores in a flow cell

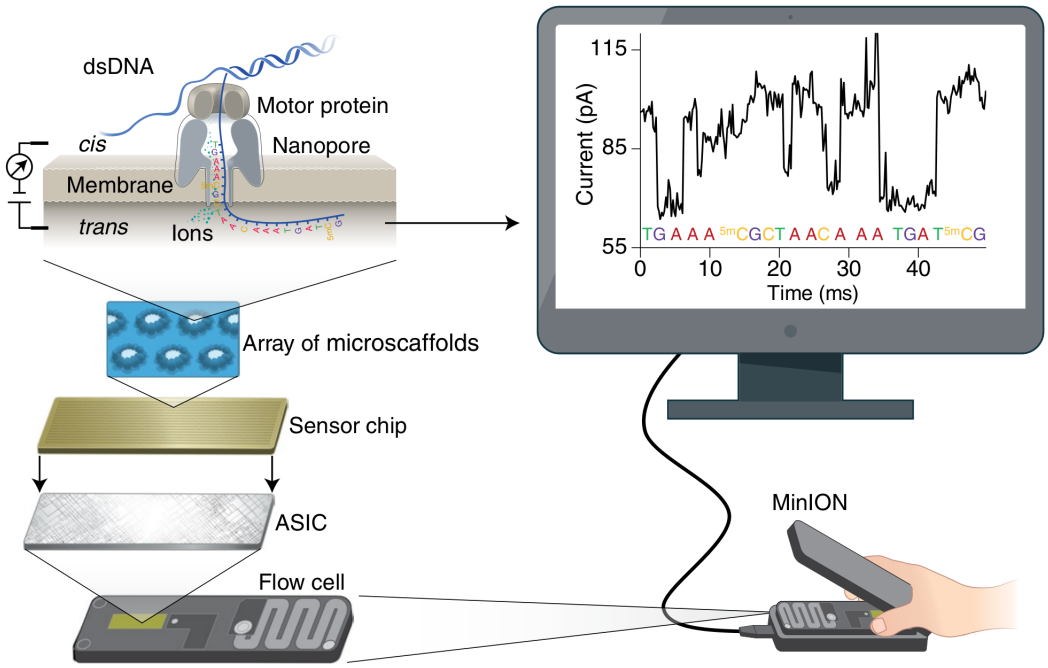


FIGURE 2.1. The MinION, the first commercially-available nanopore sequencing device, recording nanopore signal data as a double-stranded DNA (dsDNA) molecule is unwound and ratcheted through the nanopore[1].

deteriorate during their sequencing lifetime since they are protein-based. For example, a PromethION flow cell can produce a yield of up to 290 GiB and can be reused up to five times using an ONT Wash Kit (ONT, 2022). The signal recorded by such machines is digitised and recorded as a sequence of 16-bit signed integers. This sequence, hereafter referred to as *nanopore signal data*, is the focus of this thesis.

2.2 Data Compression

2.2.1 Introduction

Data compression or source coding is the process of encoding information using fewer bits. There are two main classes of data compression; lossless and lossy. Lossless data compression does not lose information in the process of encoding. Rather, it eliminates statistical redundancy or 'wasted space' in the information source. On the other hand, lossy data compression partially discards information such that the original data cannot be fully reconstructed. For this reason, lossy compression is also known as irreversible compression.

[1]Figure 2.1 was taken from Fig. 1 in (Wang et al., 2021).

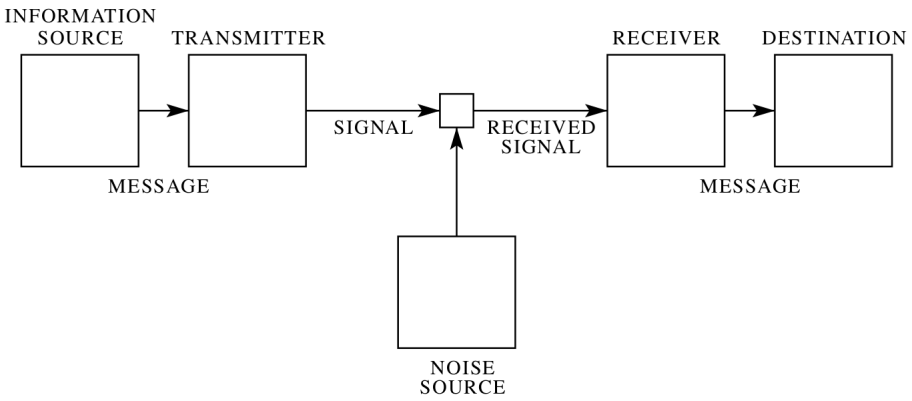


FIGURE 2.2. Shannon's general communication system consists of an *information source* which produces a message, a *transmitter* which manipulates the message to produce a signal suitable for transmission over the channel, a *channel* which serves as the medium for signal transmission from the transmitter to the receiver, a *receiver* which reconstructs the message from the signal and a *destination* which is the message's intended point of delivery[2].

Data compression relies on the framework of information theory established by Claude E. Shannon in "A Mathematical Theory of Communication" (Shannon, 1948). Information theory studies the transmission of digital information through communication systems. The communication system Shannon theorises is composed of five parts: the information source, transmitter, channel, receiver and destination. See Figure 2.2. Depending on the information transmitted through the system, it is classified into three main categories; discrete, continuous and mixed.

In the discrete case, the information source can be represented by a stochastic process

$$\{X_t : t \in \mathbb{N}^+\}$$

with a discrete state space S. Governed by this mathematical model, the information source produces a sequence of symbols from its alphabet which is received by the transmitter. If the probability of the transmitter observing the next symbol depends only on the current symbol and is conditionally independent of previous symbols, the process satisfies the Markov property. In this case, the information source can be represented by a Markov chain with a finite state space $S_1, S_2, \ldots, S_n$ and a set of transition probabilities $p_i(j)$ defined as the probability that the system will transition from state S_i to S_j.

The entropy rate, H, of an information source is a measure of how much information it 'produces' per symbol on average. Or rather, how uncertain the transmitter is of the next symbol on average. It is

[2]Figure 2.2 was taken from Fig. 1 in (Shannon, 1948).

strictly positive unless we are certain of the outcome in which case it is 0. For a discrete random variable X where each symbol $s_1, s_2, \ldots, s_n$ is independent of the next; the entropy is given by

$$H(X) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

where p_i is the probability of observing the i-th symbol (Shannon, 1948). The units here are in bits. For an information source represented as a stochastic process, the entropy rate is given by

$$H(X) = \lim_{n\to\infty} \frac{1}{n} H(X_1, X_2, \ldots, X_n)$$

when the limit exists, where $H(X_1, X_2, \ldots, X_n)$ is the joint entropy of the first n members of the process (Cover and Thomas, 1991). The limit is known to exist for a stationary process where the joint probability distribution is invariant to shifts in time. For example, for a stationary Markov chain the entropy rate simplifies to

$$H(X) = -\sum_{i,j} \mu_i p_i(j) \log_2 p_i(j)$$

where μ is the stationary distribution of the chain.

The Hartley function H_0 or absolute rate R of an alphabet S is the maximum possible entropy rate that can be achieved using it. It is defined as the logarithm of the alphabet's cardinality

$$H_0(S) = R = log_2|S|.$$

A binary source code, C, of a random variable X is a mapping from its possible outcomes Ω to a set of finite length strings of symbols from $\{0, 1\}$ known as codewords. A code is described as a prefix code or instantaneous code when no codeword is a prefix of another. In such a case, any encoded string can be conveniently decoded without reference to future codewords. For uniquely decodable codes, the expected length of a codeword cannot be smaller than the entropy of X. That is,

$$\sum_i p_i l_i \geq H(X)$$

where $l_i = |C(s_i)|$ is the length of the i-th symbol's codeword. With equality possible if and only if the probability distribution is dyadic. That is, when $\forall i \exists n \in \mathbb{N}_0$ such that $p_i = 2^{-n}$. Similarly, the entropy rate also defines a lower limit on the expected number of bits per symbol for a source encoding of an information source represented as a stochastic process.

2.2.2 Entropy Coding

Many lossless data compression methods have been proposed since Shannon's formative paper in 1948. Entropy coding is one such scheme that does not depend on the data's characteristics. It typically encodes each fixed-width symbol with a variable-length binary prefix code as described above. Fundamentally, it attempts to achieve a bit rate as close as possible to the entropy rate of the information source.

2.2.2.1 Huffman coding

Huffman coding is a well-known entropy coding technique which minimises the expected length of prefix codes (Huffman, 1952). It solves the following optimisation problem:

$$\begin{aligned} &\underset{l_i}{\text{minimise}} \sum_i p_i l_i \\ &\text{subject to} \sum_i 2^{-l_i} \leq 1 \end{aligned}$$

where $l_i \in \mathbb{N}^+$. The inequality constraint is known as the Kraft-McMillan inequality and defines a necessary condition for the existence of a uniquely decodable binary code (McMillan, 1956). The Huffman coding algorithm is a greedy algorithm which constructs a binary tree from the bottom-up. Recursively, it takes the two symbols with the lowest probabilities and constructs a new node with these two as its children. This new node is considered to be a new symbol with the combined probability of its children. Once the tree is constructed, the edges to the left are labelled as 0 and those to the right as 1. A hash table mapping symbols to binary codewords is then constructed by traversing the tree from the root to each leaf node and concatenating the edge labels along the path taken. The time complexity for this algorithm is $O(n \log n)$ or $O(n)$ if the probabilities are already sorted, where n is the number of symbols (Leeuwen, 1976).

However, perhaps a more appropriate measure of time complexity is the number of passes over an input string of length m. In this case, Huffman coding takes $O(m)$ time as it requires one pass if the probability distribution is already known. Otherwise, the probability distribution can be determined by a preliminary pass over the input. Or instead, an adaptive Huffman coding algorithm such as the Faller-Gallager-Knuth (FGK) or Vitter algorithm can be used (Knuth, 1985; Vitter, 1987). In the latter cases, the hash table or Huffman tree must also be encoded in the output.

Among all source codes when the probability distribution of the alphabet is known beforehand, Huffman coding is indeed optimal. However, once these restrictions are relaxed, other techniques prove to be more compressible. In particular, Huffman coding tends to be inefficient on small alphabets such as the binary numbers $\{0, 1\}$ wherein no compression is possible without an alphabet extension (Rissanen, 1976). Similarly, if the probability distribution is not dyadic, other non–source-coding techniques are more optimal.

2.2.2.2 Arithmetic and range coding

Arithmetic coding is one such entropy coding technique which encodes the input string as a fraction in the range $[0, 1)$ (Rissanen, 1976). The starting interval $[0, 1)$ is divided into sub-intervals corresponding to each symbol in the alphabet with the length of each sub-interval equal to its symbol's expected probability of occurrence. Each time a symbol is read from the input string its sub-interval is chosen and divided as before. Recursively, this process is repeated until the final symbol is read, at which point any fraction in the final interval can be returned. Arithmetic coding overcomes the weaknesses of Huffman coding described above and requires the same number of passes over the input. However, in practice it is more difficult to implement and face issues of computational accuracy.

Range coding is an equivalent technique with a slightly different implementation (Martin, 1979). Instead of encoding the input string as a fraction within $[0, 1)$, it encodes the input string as a number within a range. It can be implemented twice as fast as arithmetic coding but results in a minimally worse compression ratio.

2.2.3 Dictionary Coding

Dictionary coding is another lossless compression scheme which searches in the input data for strings stored in a dictionary and substitutes matches with a reference to their location in the dictionary. There are two main classes of dictionary coders; ones that use a static predetermined dictionary and others which dynamically update their dictionary.

Byte-pair encoding falls under the second class and recursively replaces the most frequently occurring pair of adjacent bytes with a new byte not found in the input data (Gage, 1994). Despite being so simple, Gage claims it is on par with the Lempel, Ziv and Welch (LZW) method (Welch, 1984). It is however quite slow at compression, requiring many passes of the data, but takes only one pass for decompression.

Lempel-Ziv coding is another dictionary coding technique which is widely used in practice and has many variants. LZ77 is the first such algorithm to be introduced in the literature and replaces repeated occurrences of substrings with references to their original location in the input (Ziv and Lempel, 1977). It maintains a fixed-width sliding window represented as a circular buffer in order to keep track of the most recent data whilst keeping the algorithm's complexity under control. Each match is encoded by the substring's length and the distance to its original occurrence. LZ78 is very similar but instead constructs an explicit dictionary of substrings and replaces repeated occurrences with references to their location in the dictionary (Ziv and Lempel, 1978). Both variants require one pass over the input string and have been proven to be asymptotically optimal as the length of the string grows to infinity. LZW is an improved implementation of LZ78 published in 1984 which is used in the Unix utility `compress` and GIF image format (Welch, 1984; Group, 2004; W3C, 1990).

2.2.4 Other Encodings

The DEFLATE format is a widely popular encoding which combines LZ77 and Huffman coding. It is used in ZIP, gzip and the PNG file format, and is supported by the zlib library (IANA, 1993; IETF, 2012; for Standardization, 2004). Zstandard or *zstd* (pronounced 'zee standard') is a lossless data compression algorithm which combines LZ77 and a fast entropy-coding stage (Collet and Kucherawy, 2018). It has a similar compression ratio to DEFLATE but performs much faster, especially during decompression.

There are many other lossless data compression schemes described in the literature. Run-length encoding is one such method which encodes substrings consisting of repeated consecutive symbols, known as *runs*, with their repeated symbol and length. First employed in 1967 for transmission of analogue television signals, run-length encoding proves beneficial when there are many runs, especially of long length (Robinson and Cherry, 1967). Unfortunately, nanopore signal data does not contain many such runs and would likely encode poorly under this method without first applying some transformation.

Burrows-Wheeler transform is a data compression preprocessing step used to rearrange data in order to increase its number of runs (Burrows and Wheeler, 1994). It is easily reversible, such that the original untransformed data can be obtained from the Burrows-Wheeler transformed data. It has been used to great effect in bioinformatics to compress to basecalled genomic data in the form of FASTQ files (Cox et al., 2012). bzip2 is a popular implementation of the Burrows-Wheeler transform which is more effective than DEFLATE and LZW but performs slower (Seward, 2019).

4×20 bits or 10 bytes

0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0
1	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	1	0	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
1	0	0	0	0	0	0	0

FIGURE 2.3. The sequence of numbers $1024, 12, 10, 524\,288$ in the bit packed format. The data is read from left to right, top to bottom with the most significant bits first[3].

2.2.4.1 Integer compression

Bit packing is the process encoding an array of unsigned integers $x_1, x_2, \ldots, x_n$ using the smallest number of bits $\hat{b}$ per integer such that no information is lost. Consider the integers $1024, 12, 10, 524\,288$. The largest $524\,288$ can be represented using 20 bits, hence bit packing represents each integer using $\hat{b} = 20$ bits. Refer to Figure 2.3 for a visual depiction. Assume that each integer is typically represented using b bits. Then, this encoding saves

$$\lfloor \frac{n(b - \hat{b})}{8} \rfloor$$

bytes. This equates to a compression ratio of

$$\begin{aligned} \text{Compression Ratio} &= \frac{\text{Uncompressed Bytes}}{\text{Compressed Bytes}} \\ &= \frac{\lceil \frac{nb}{8} \rceil}{\lceil \frac{n\hat{b}}{8} \rceil} \\ &\overset{n \to \infty}{\longrightarrow} b/\hat{b}. \end{aligned}$$

In most programming languages, the integers $1024, 12, 10, 524\,288$ would each be represented using 4 bytes or $b = 32$ bits. Hence, on this example the compression ratio would be

$$\frac{\lceil \frac{4 \times 32}{8} \rceil}{\lceil \frac{4 \times 20}{8} \rceil} = 1.6.$$

Compression and decompression using this method can be efficiently computed using scalar processing. It can be computed even faster by using an alternative bit packing layout and exploiting single-instruction-multiple-data (SIMD) processing (Lemire et al., 2014).

TABLE 2.1. The number of bytes used and their control code for different supported integer ranges in classical 32-bit, 0-based and 16-bit Stream VByte.

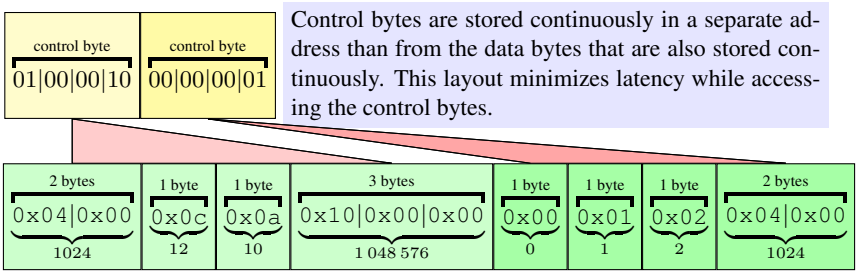
Stream VByte Method	Integer Range	Number of Bytes Used	Control Code
Classical 32-bit	$[0, 255]$	1	00
	$[256, 65535]$	2	01
	$[65536, 16777215]$	3	10
	$[16777216, 4294967295]$	4	11
0-based	$\{0\}$	0	00
	$[1, 255]$	1	01
	$[256, 65535]$	2	10
	$[65536, 4294967295]$	4	11
16-bit	$[0, 255]$	1	0
	$[256, 65535]$	2	1

The Frame-Of-Reference (FOR) encoding subtracts the minimum value from all integers before proceeding with bit packing (Goldstein et al., 1998). This potentially decreases the number of bits $\hat{b}$ used per integer and is especially successful if the integers have a small range. The minimum value is recorded before the bit packed data.

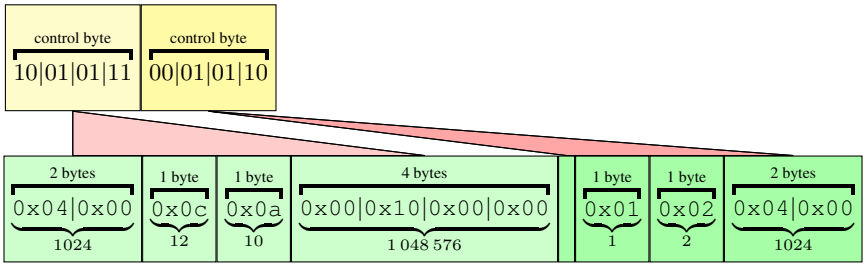
Stream VByte is a specialised codec for compressing 32-bit unsigned integers (Lemire et al., 2018). It stores each integer using a variable number of bytes (1 to 4) depending on its size. Integers in the range $[2^{8(n-1)}, 2^{8n} - 1]$ are represented using $n > 0$ bytes. For example, integers in the range $[1, 255]$ are losslessly represented using 1 byte. The integer 0 is a special case missing from the above formula which is classically represented using 1 byte. There is however a variation which instead encodes 0 using 0 bytes and integers in the 3 byte range $[2^{16}, 2^{24} - 1]$ with 4 bytes rather than 3. This is advantageous if 0 occurs more often than integers in the 3 byte range. See Table 2.1 for a comparison of the classical and 0-based encoding. The number of bytes used for each integer is stored in an array of control bytes which prefaces the actual data. 2-bit words are used to store the number of bytes used for each integer with 00, 01, 10 and 11 corresponding to 1, 2, 3 and 4 bytes respectively (or 0, 1, 2 and 4 bytes in the 0-based variation). See Figures 2.4 for an example of how the encodings are structured.

Another variation to this encoding for 16-bit unsigned integers, known as Stream VByte 16, was developed by ONT in 2022 for compressing nanopore signal data in the POD5 file format (nanoporetech, 2022a). It is the same as the classical Stream VByte encoding described above except that each integer is stored using 1 or 2 bytes rather than 1 to 4 bytes. Since there are two different byte lengths, each of the byte lengths is stored using 1 bit; byte lengths 1 and 2 correspond to bit values 0 and 1 respectively.

[3]Figure 2.3 is heavily inspired by Figure 1 from (Lemire et al., 2014).



(A) Compressed classical Stream VByte bytes



(B) Compressed 0-based Stream VByte bytes

FIGURE 2.4. An example of $1024, 12, 10, 1\,048\,576, 0, 1, 2, 1024$ compressed with classical and 0-based Stream VByte[4].

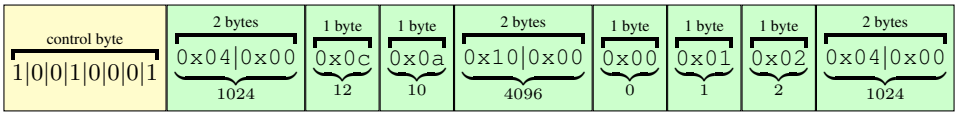


FIGURE 2.5. Compressed 16-bit Stream VByte bytes for $1024, 12, 10, 4096, 0, 1, 2, 1024$[4].

See Table 2.1 and Figure 2.5. If each integer lies in the range $[0, 2^{16})$ this method saves one bit per integer on average versus classical 32-bit Stream VByte. Due to their similarities, I hypothesise that the compression and decompression speed of Stream VByte 16 is similar if not better than classical Stream VByte. But there is no existing benchmark in the literature which compares both algorithms.

For the remainder of the thesis, let *svb*, *svb0* and *svb16* refer to classical 32-bit, 0-based and 16-bit Stream VByte respectively.

2.2.4.2 Signal compression

A *wavelet* is a brief oscillation which starts and ends with an amplitude of zero. Sets of complementary wavelets are desired for wavelet-based compression since they can reversibly decompose a signal. Wavelet transforms are more useful than Fourier transforms for approximating signal data with sharp

[4]Figures 2.4 and 2.5 are modified versions of Figures 1–3 from (Lemire et al., 2018).

peaks and non-periodic behaviour. Wavelet compression is most applicable to image data but has been successfully applied to audio and video data as well as electrocardiograph (ECG) signals (Ramakrishnan and Saha, 1997). It is used in the JPEG 2000 standard.

Wavelet compression decomposes a signal into the coefficients of mathematical functions. Typically, the wavelet transform approximates the signal meaning that in order to achieve lossless compression the residuals between the approximation and the actual signal must be encoded. This encoding is more compressible if the approximation is accurate. Furthermore, the coefficients must also be recorded in order to reconstruct the wavelet transform during decompression.

Discrete wavelet transforms (DWT) is the most commonly used wavelet transform in signal compression because it can be implemented naturally using a computer since it discretely samples each wavelet. There are several DWT forms namely Haar, Daubechies and undecimated.

2.2.5 Nanopore Compression

There have been many studies investigating genomic data compression (Hernaez et al., 2019). Few, however, have successfully focussed on losslessly compressing nanopore signal data. One tool called Picopore does not produce any novel insights, but rather, increases the level of gzip compression to the highest possible (Gigante, 2017).

Rather interestingly, there has been some recent research effects to investigate lossy compression of nanopore signal data. Chandak et. al. found that lossy time-series compressors (LFZip and SZ) tend to have a very small impact on the downstream analysis of nanopore data (Chandak et al., 2020a,b). However, they seem to overlook the prospect of a more efficient lossless compression technique for nanopore signal data:

> "obtaining further improvements in lossless compression is challenging due to the inherently noisy nature of the current measurements" (Chandak et al., 2020a).

For this reason and perhaps due to the novelty of nanopore sequencing, little has been further attempted in the literature to losslessly compressing nanopore signal data.

A timeline of the major events in the data compression and nanopore sequencing literature is presented in Table 2.2.

TABLE 2.2. A timeline of the major events in the literature.

1948	Shannon publishes "A Mathematical Theory of Communication" (Shannon, 1948)
1952	Huffman publishes a method for finding optimal prefix codes (Huffman, 1952)
1967	Run-length encoding first described (Robinson and Cherry, 1967)
1976	Pasco and Rissanen develop arithmetic coding independently (Rissanen, 1976)
1977	Lempel and Ziv publish LZ77 (Ziv and Lempel, 1977)
1978	Lempel and Ziv improve upon LZ77 with LZ78 (Ziv and Lempel, 1978)
1979	Martin proposes range coding (Martin, 1979)
1984	Welch publishes LZW based on LZ78 (Welch, 1984)
1987	Vitter improves upon the adaptive Huffman coding algorithm FGK (Vitter, 1987)
1994	Burrows and Wheeler publish the Burrows-Wheeler transform (Burrows and Wheeler, 1994); and Gage submits byte-pair encoding as a C article (Gage, 1994)
2014	ONT release the first portable nanopore sequencing device
2017	A tool for reducing nanopore data, Picopore, is released (Gigante, 2017)
2018	Lemire publishes Stream VByte (Lemire et al., 2018)
2019	VBZ is first released (nanoporetech, 2022b)
2020	Lossy compression of nanopore signal data is investigated (Chandak et al., 2020a)

2.3 The State-of-the-Art

This leads us to the current state-of-the-art approach for compressing nanopore signal data which we will appropriately name zstd-svb-zd, although VBZ is the name ONT uses to describe it (nanoporetech, 2022b).

zstd-svb-zd consists of the following encodings applied successively:

(1) differential coding,

(2) zig-zag encoding,

(3) svb and

(4) zstd.

Differential coding is a bijective function from $\mathbb{R}^n \to \mathbb{R}^n$ classically defined as

$$(x_1, x_2, \ldots, x_n) \mapsto (\delta_1, \delta_2, \ldots, \delta_n) = (x_1, x_2 - x_1, x_3 - x_2, \ldots, x_n - x_{n-1}).$$

That is, the original data is replaced by the differences (or *deltas*) between successive data points. This transformation is beneficial for compression if the deltas are smaller than the data itself or more repetitive. Both the encoding and decoding algorithms for differential coding only require one pass of the data. However, computation of the *prefix sum* during decompression, defined as

$$x_1, x_1 + \delta_2, x_1 + \delta_2 + \delta_3, \ldots$$

is typically a bottleneck in SIMD optimised applications. For this reason, there are SIMD-based variations which compute larger deltas, such as $\delta_i = x_i - x_{i-4}$, in exchange for faster decompression times (Lemire et al., 2014). For nanopore signal data, larger deltas may not actually result in poor compression ratio performance and may be desirable if fast query time is of higher interest.

The zig-zag encoding is another bijection defined as

$$z : \mathbb{Z} \to \mathbb{N}_0,$$
$$z(x) = 2|x| - \mathbb{1}_{x<0}(x).$$

That is, the negative integers are interleaved with the positive integers such that 0 maps to 0, -1 to 1, 1 to 2, -2 to 3 and so forth. See Figure 2.6 for a mathematical depiction of the zig-zag encoding. This encoding allows downstream compression methods to assume all integers are positive and hence ignore the sign-bit from the two's-complement representation. Whilst also keeping the relative magnitude of numbers the same – numbers close to zero remain close to zero and vice-versa for numbers far from zero. The zig-zag encoding z of a b-bit integer x can be efficiently computed and decoded using standard bitwise operations:

```
z = (x << 1) ^ (x >> (b - 1))
x = (z >>> 1) ^ -(z & 1)
```

This technique is used for signed integers in Google Protocol Buffers and is briefly described in their Developer Guide (Google, 2022).

Altogether, the state-of-the-art compression method first applies differential coding followed by zig-zag encoding to obtain the 'zig-zag deltas' of the data. Then, svb is applied to the zig-zag deltas followed by the zstd algorithm. ONT believe that using svb16 instead of svb here achieves better compression, but the benchmark results presented in Chapter 6 reveal otherwise. Hence, zstd-svb-zd is the current state-of-the-art approach for losslessly compressing nanopore signal data.

zstd-svb-zd is fast in practice for both compression and decompression. This is mostly because its sub-methods (zstd and svb) have been highly optimised. Furthermore, it only requires at least four passes of the data during compression; finding the zig-zag deltas takes two, svb takes one and zstd requires at least one. The same is true for decompression. However, the method is very generic, only using the

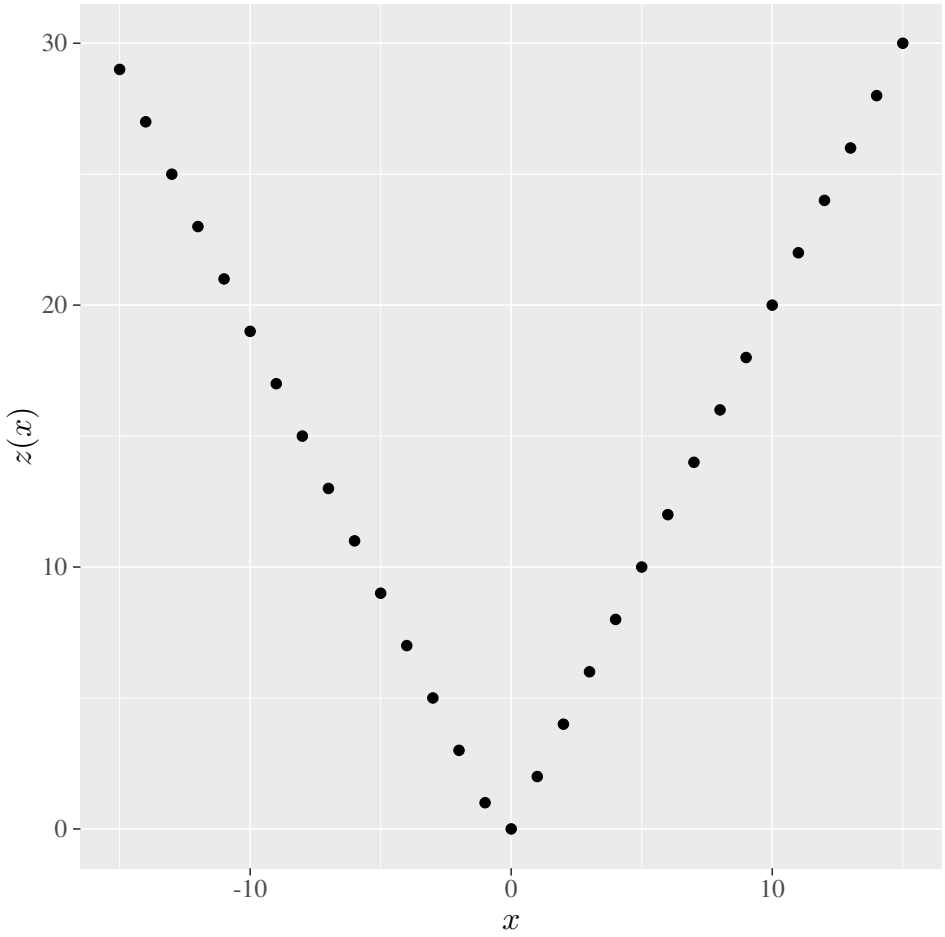


FIGURE 2.6. The zig-zag encoding applied to integers -15 to 15. As we can see, it is similar to the absolute value function but is doubled and shifted on the negative x-axis to ensure injectivity.

fact that the signal's deltas are small to achieve better compression. Hence, by analysing the data and understanding its characteristics, we should be able to save more space.

This leads us to the following chapter, where we will conduct a systematic analysis of the data, its characteristics and transformations.

CHAPTER 3

The Data

3.1 Introduction

The data consists of many sequences of unsigned integers known as *reads* and their associated metadata. Each read represents the quantised ionic current measured across a nanopore as a single-stranded DNA or RNA molecule is driven through it. The current is typically recorded or 'sampled' 4000 times a second by an electrode in the sensor chip and digitised using an analogue-to-digital converter (ADC). This recording frequency is known as the *sampling rate* and is stored as metadata for each read. Disturbances in the ionic current caused by the molecule's biological structure can be used to determine its nucleic acid sequence.

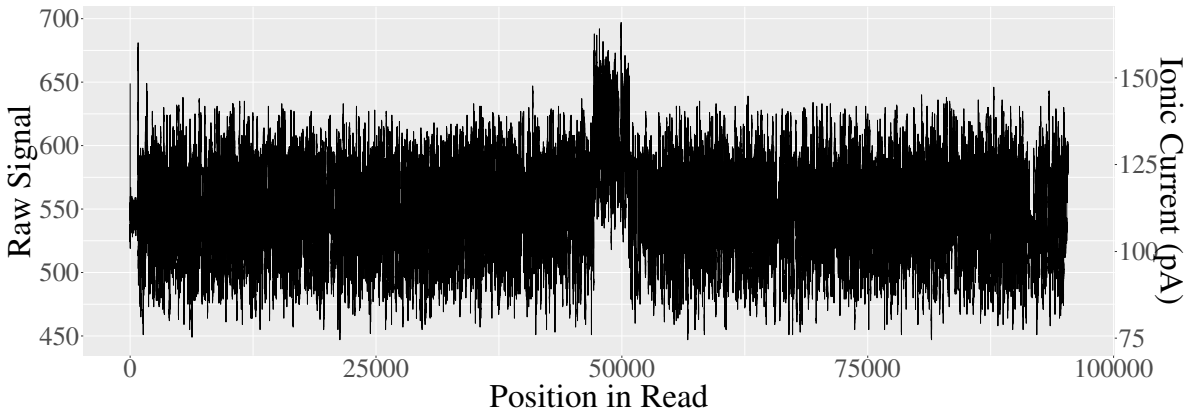


FIGURE 3.1. The read with ID e9f08690-171f-476f-9119-5330d0290126 from the NA12878 data set plotted against two axes. The primary y-axis (left) is the raw signal values and the secondary y-axis (right) is the ionic current found using equation 3.1. $535, 531, 529, 519, 535, \ldots$ is the sequence starting from position 25000 and $106.73, 105.27, 104.54, 100.88, 106.73, \ldots$ is the same sequence after converting to picoamperes (rounded to 2 decimal places).

TABLE 3.1. Some constant metadata of the NA12878 data set.

Sampling Rate (Hz)	4000
ASIC Start Temperature (°C)	31.996552
Device Type	PromethION 48
Start Time (ISO 8601)	2020-10-27T05:41:50Z
Duration (mins)	4320
Experiment Type	DNA
Digitisation	2048
Range (pA)	748.580139

TABLE 3.2. The variable metadata of the read with ID e9f08690-171f-476f-9119-5330d0290126 from the NA12878 data set.

Length	95350
Offset	-243
Channel No.	2642
Pore No.	4
Channel Read Count	437
Channel Data Point Count	5599213
Median Before (pA)	238.772018
End Reason	Signal Positive

Let each read (or 'raw signal') be represented by

$$x := (x_i \mid x_i \in \mathbb{Z} \cap [0, 2^{16}))$$

where $i \in \mathbb{Z} \cap [0, n)$. Computationally, x is an unsigned 16-bit integer array with n elements. However, in practice the full range of 2^{16} integers is never met; a property which will be further explored and heavily exploited. See Figure 3.1 for an example of a read.

There are many metadata fields which are constant for all reads of the same sequencing run. This includes the sampling rate and every header attribute. Common header attributes include metadata relating to the ASIC chip such as its temperature at the start of the run, the ONT device being used, the start time of the run, its duration in minutes and whether DNA or RNA is being used (Gamaarachchi et al., 2022). See Table 3.1.

The main read-variable metadata is the length n and the information necessary for converting the read from quantised raw signal values into picoamperes (pA). This information is comprised of three fields: the digitisation, offset and range. The digitisation is the number of quantisation levels in the ADC which is typically 2048 (2^{11}) or 4096 (2^{12}). Each data point in a read is recorded using 16 bits; it is the closest multiple of a byte after 11 and 12 bits. The next field is the offset which is the ADC's offset error whilst

the range is the sensor chip's full measurement range in picoamperes. To convert a raw signal value into picoamperes one does the following computation

$$(raw + offset) \times \frac{range}{digitisation}. \tag{3.1}$$

The digitisation and range fields are actually constant between reads since they are hardware-defined properties; it is the offset which varies among reads. See Table 3.2.

There are other read-variable metadata fields which are optionally included in the data but nonetheless may prove useful for a compression algorithm. These include the reason the read ended, flow cell location data comprised of the channel and pore number, the estimated median current level before the start of the read, and a count of the number of reads and data points previously recorded by the channel.

For the remainder of the thesis, the data which we will use for analysing the characteristics of nanopore signal data and for comparing compression methods is a downsampling of a typical human genome sequencing data set, generated from a reference sample known as NA12878. The NA12878 data set has been well studied and is used prolifically for benchmarking genomics tools. It originates from a Utah woman whose DNA reference was artificially recreated, treated using a short read eliminator kit and sequenced on an ONT PromethION 48 using two version R9.4.1 flow cells to obtain $\sim$30$\times$ coverage of her genome. The data set was then downsampled a factor of $\sim$18 by obtaining the first $500\,000$ reads. The first reads usually contain more data than later reads as each nanopore deteriorates over its sequencing lifetime. Downsampling produces a smaller data set which speeds up testing, and the compression results found will scale up to the whole data set.

It is highly likely that human DNA is the most commonly sequenced molecule of all sequencing technologies, as well as specifically of ONT devices. Naturally then, most data sets that exist are the result of sequencing human DNA and hence share patterns inherit in human biology and the DNA sequencing process. Hence, it makes sense to consider a human DNA data set in a study of data compression. Furthermore, the PromethION flow cell produces the most yield out of all ONT flow cells so PromethION data is also highly suitable. Thus, the NA12878 data set is a good choice for the comparison of nanopore signal compression methods.

The original data set contains $9\,083\,052$ reads and consumes $\sim$2 TiB in raw binary or $\sim$707 GiB using VBZ compression. After downsampling, the data set contains $500\,000$ reads and $\sim$57 billion data points, consuming $\sim$106 GiB in raw binary or $\sim$37 GiB when stored using VBZ. See Tables 3.3 and 3.4 for an

TABLE 3.3. The original NA12878 data set.

Description	Adult Utah Female DNA
No. of reads	9 083 052
Size (BLOW5 none)	$\sim$ 2 TiB
Size (BLOW5 VBZ)	$\sim$ 707 GiB

TABLE 3.4. The downsampled NA12878 data set.

Description	Adult Utah Female DNA Downsampled
No. of reads	500 000
No. of data points ($\times 10^9$)	$\sim$ 57
Avg. read length	$\sim$ 113 471
Size (BLOW5 none)	$\sim$ 106 GiB
Size (BLOW5 VBZ)	$\sim$ 37 GiB

overview of the data sets. From now onwards, *the data set* refers to the downsampling of the original NA12878 data set and can be downloaded from `https://slow5.page.link/na12878_prom_sub_slow5`.

The metadata accounts for lesser than 0.05% of the raw binary data set's size, meaning that without storing any metadata the data set consumes roughly the same size. From a compression point of view there is much more to gain from focussing on compressing the read data. For this reason, future comparisons of file size including compression ratios will not represent the cost of storing metadata, only read data.

The raw signal values in the data set range from 158 to 1748 with a median of 474 and standard deviation of $\sim$ 35. See Table 3.5. The range is 1590 but the middle 99% of the data ranges from 350 to 622. Figure 3.2 shows the singular-width bin histogram of the raw signal values between 295 and 687. As one can observe, the distribution is fairly symmetric around the centre with the median and mean very close to one another. However, it is slightly right-skewed with its right tail extending much further than its left. This is most evident by observing that the maximum is more than four times further from the median than the minimum is. Most interestingly, the histogram is not smooth with the frequency of adjacent signal points varying drastically. For example, the value 455 occurs $\sim$6.3 times more often than 456. In fact, on close inspection there is a cyclical pattern in the relative frequency of adjacent signal points in Figure 3.2 which appears to repeats every 16 values. For example, consider the frequencies of values 456–471 and values 472–487. This is most likely a result of the ADC unevenly digitising the analogue ionic current signal in some deterministic way.

TABLE 3.5. Summary statistics of the data's raw signal values.

Min	158
Q1	439
Q2	474
Q3	511
Max	1748
Mean	475.2245
Mode	487
SD	35.0675

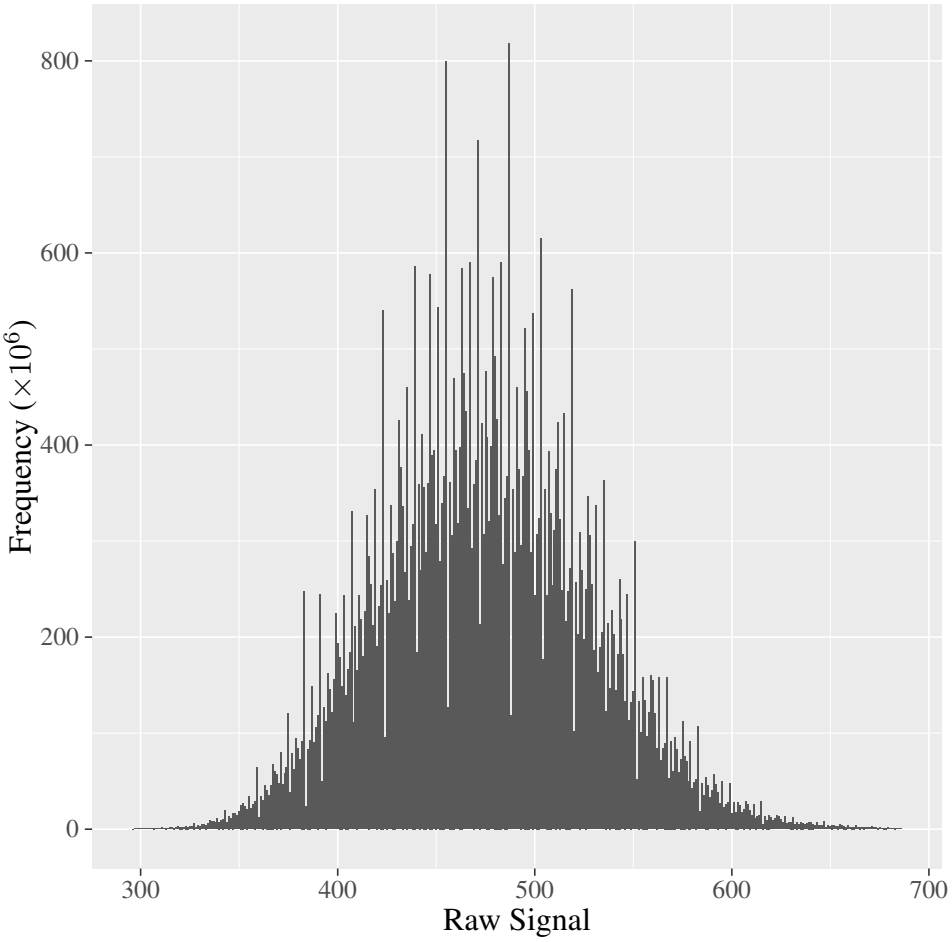


FIGURE 3.2. The frequency of raw signal values 295 to 687 in millions for the data set. The raw signal values outside this range occurred less than 1 million times (or with a probability lesser than 0.002%).

The read lengths range from 2024 to 5 724 000 with a median of 80304.5. See Table 3.6 for an overview and Figure 3.3 for the histogram with bin width one. As one can observe, there are a lot of large read lengths which skew the distribution to the right.

TABLE 3.6. Summary statistics of the data's read lengths.

Min	2024
Q1	29389.75
Q2	80304.5
Q3	163 825
Max	5 724 000
Mean	113471.4
Mode	11 923
Sample SD	110535.7

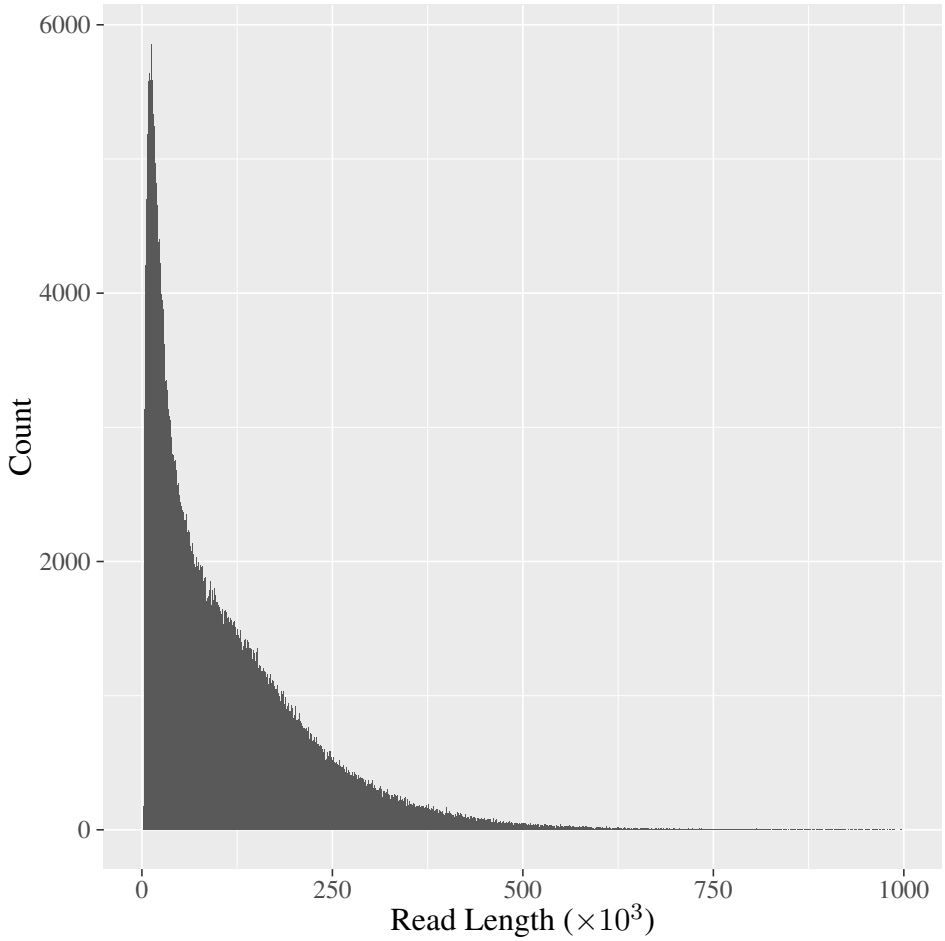


FIGURE 3.3. Histogram of the number of data points in (length of) each read for the data set. The distribution appears to modelled by an exponential or Gamma distribution.

3.2 Characteristics

We will now discuss the characteristics of the data – human DNA reads. General DNA/RNA reads should exhibit similar properties but there are several differences and the primary focus of this thesis is human DNA data.

Each read consists of several sections with recognisable characteristics; the surge, stall, pre-adapter surge, adapter, DNA, homopolymer and slow sections. Figure 3.4 shows the beginning of a read with the surge, stall, pre-adapter surge and adapter sections delimited.

The surge typically consists of one signal at the beginning of the read which is significally above the median. During the surge, the pore is in its open state (no molecules within it) at the time of recording and MinKNOW, the recording software, has failed to completely trim the pre-data section. This results in a large current surge since there are no DNA molecules being kinetically propelled by the electric field.

The stall also occurs at the beginning of the read and after the surge if it exists. It consists of hundreds to thousands of data points which oscillate with small variation around the median of the read. This section is rarely missing from the read and possibly occurs due to the motor protein stalling before beginning to unwind the molecule.

The adapter sequence is the first DNA sequence recorded in the signal data. The adapter connects the motor protein to the DNA strand and has a predictable molecular sequence which differs from sequencing kit to kit. There is typically a surge between the stall and the adapter section consisting of several signal values which we will name the *pre-adapter surge*.

The DNA sections are the typical data sections of a read and store information necessary for determining the original DNA sequence. This type of section accounts for the large majority of data points in a typical read. Characteristically, DNA sections oscillate in smaller low-variance sections of 10–100 data points with steep transitions between them. The adapter sequence is quite similar in behaviour since it also represents molecular information. See Figure 3.5 for an example.

The homopolymer and slow sections typically occur infrequently at random positions in the read and consist of hundreds to thousands of data points which oscillate with small variation around some value. See Figures 3.6 and 3.7 respectively. This is quite similar to the stall section except that the stall oscillates near the median value of the read, whilst the homopolyer and slow sections can occur at any feasible

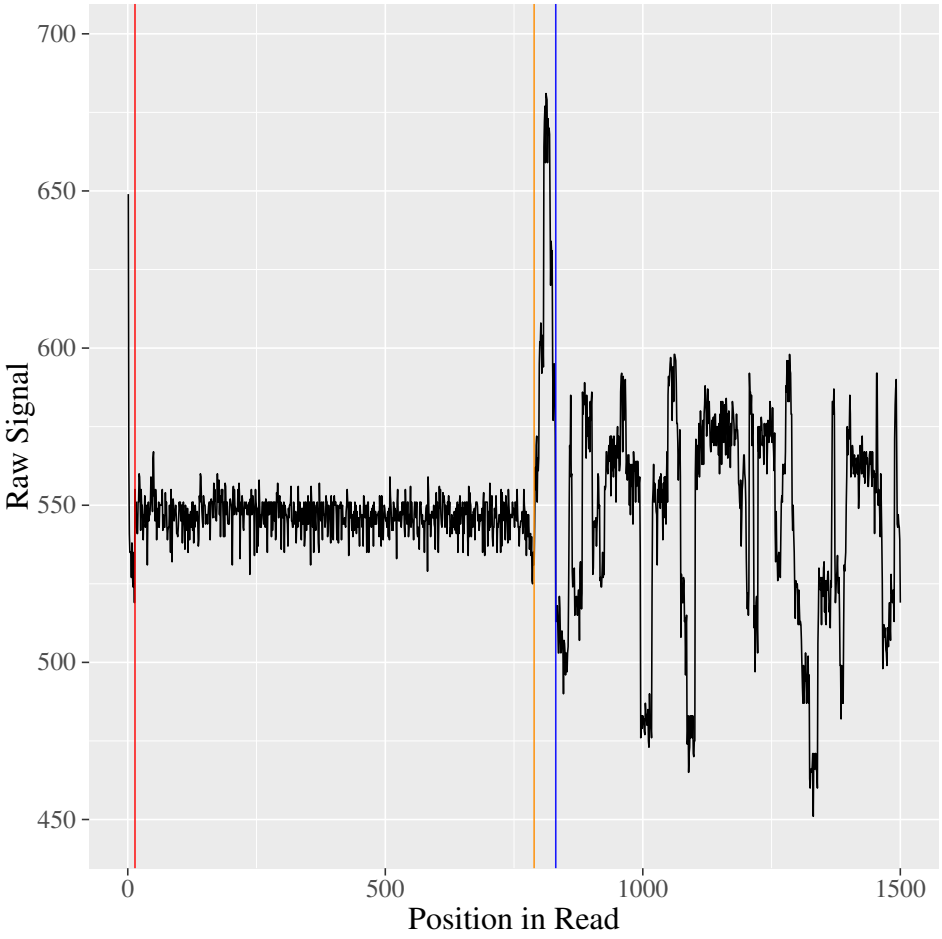


FIGURE 3.4. The first 1500 data points of the read with ID e9f08690-171f-476f-9119-5330d0290126 split into four sections: surge (before the red line), stall (between red and orange), pre-adapter surge (between orange and blue) and adapter sequence (after blue). In order from left to right the vertical lines are coloured red, orange and blue.

value. The difference between the homopolymer and slow sections is that the homopolymer section represents the repetition of one or more DNA bases, whilst the slow section represents several DNA bases which are moving through the nanopore relatively slowly, or have gotten temporarily stuck, and so are recorded over many more data points than usual. There is sometimes a slow-like section which terminates the read as well.

However predictable some of these sections are, each read can have strange unpredictable behaviour. For example, a sudden significant shift in values can be seen in Figure 3.8. Whilst, an unexpected drop terminates the read in Figure 3.9.

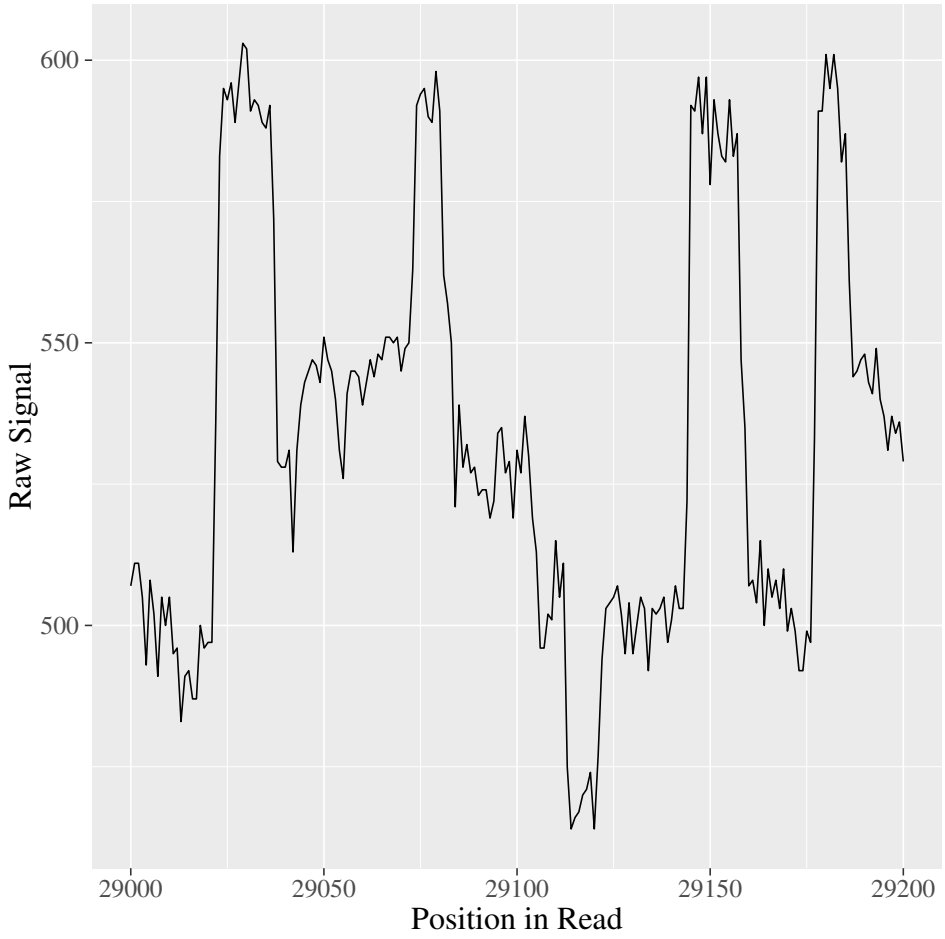


FIGURE 3.5. An example of 200 data points from a DNA section in the read with ID e9f08690-171f-476f-9119-5330d0290126. It is characterised by smaller low-variance sections with steep transitions between them.

TABLE 3.7. Summary statistics of the data's raw signal values (None) and its various transformations.

Transformation	None	Delta	Zig-Zag Delta
Min	158	-1159	0
Q1	439	-5	4
Q2	474	0	10
Q3	511	5	18
Max	1748	913	2317
Mean	475.2245	~ 0	15.5679
Mode	487	0	0
SD	35.0675	13.0625	20.6060

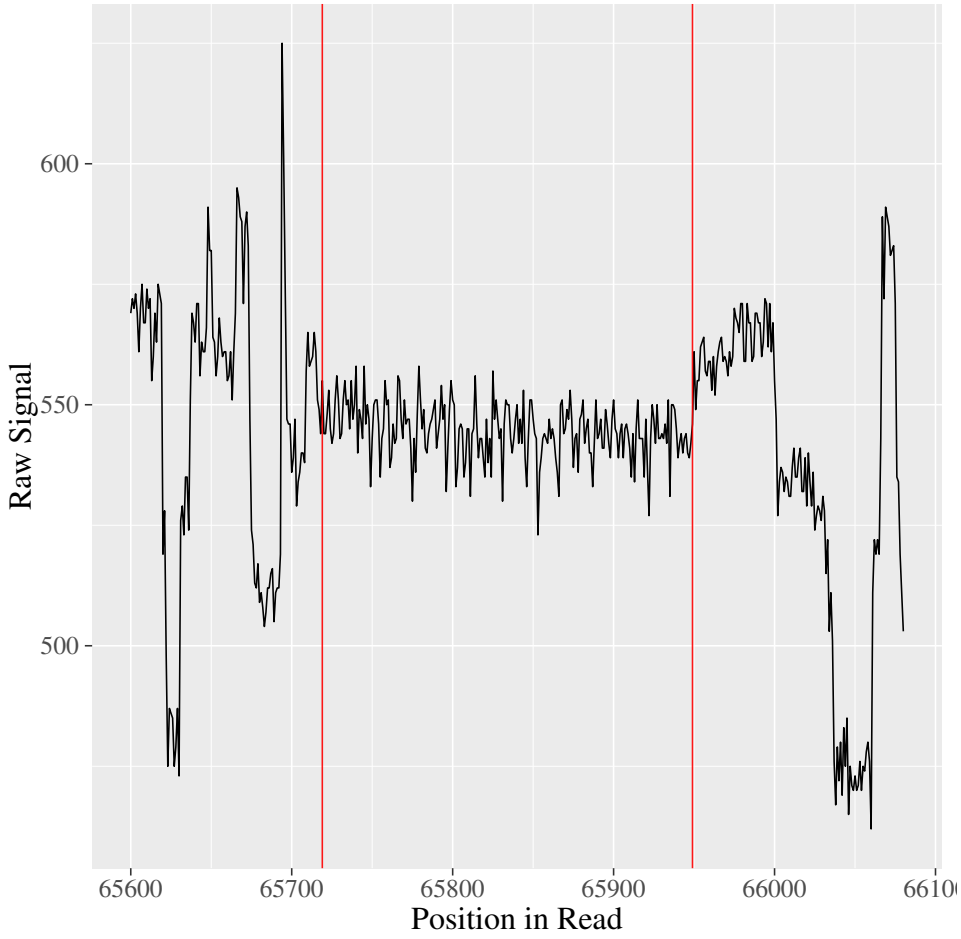


FIGURE 3.6. An example of a homopolymer section (repetition of 'T') from the read with ID e9f08690-171f-476f-9119-5330d0290126. The thymine DNA base is repeated 33 times in the homopolymer.

3.3 Transformations

As previously discussed, differential coding has been applied to the compression of nanopore signal data. Let's now consider the deltas between each read data point. Figure 3.10 plots the differential encoding of the read with ID e9f08690-171f-476f-9119-5330d0290126. In comparison to the standard representation, as shown in Figure 3.1, differential coding removes inconsistent shifts in the data such that it has fewer repeated outliers. For example, the upward shift around position 50000 in Figure 3.1 is no longer clearly evident in the differential encoding. The deltas range from -1159 to 913 with a median, mean and mode of 0, and a standard deviation of $\sim$13. See Table 3.7. The middle 99% lies tightly in the range from -46 to 63 in a normal-like distribution. See Figure 3.11 for a histogram of the deltas. Interestingly, -2 occurs $\sim 23 \times 10^6$ more times than -1 which is likely a result of the ADC.

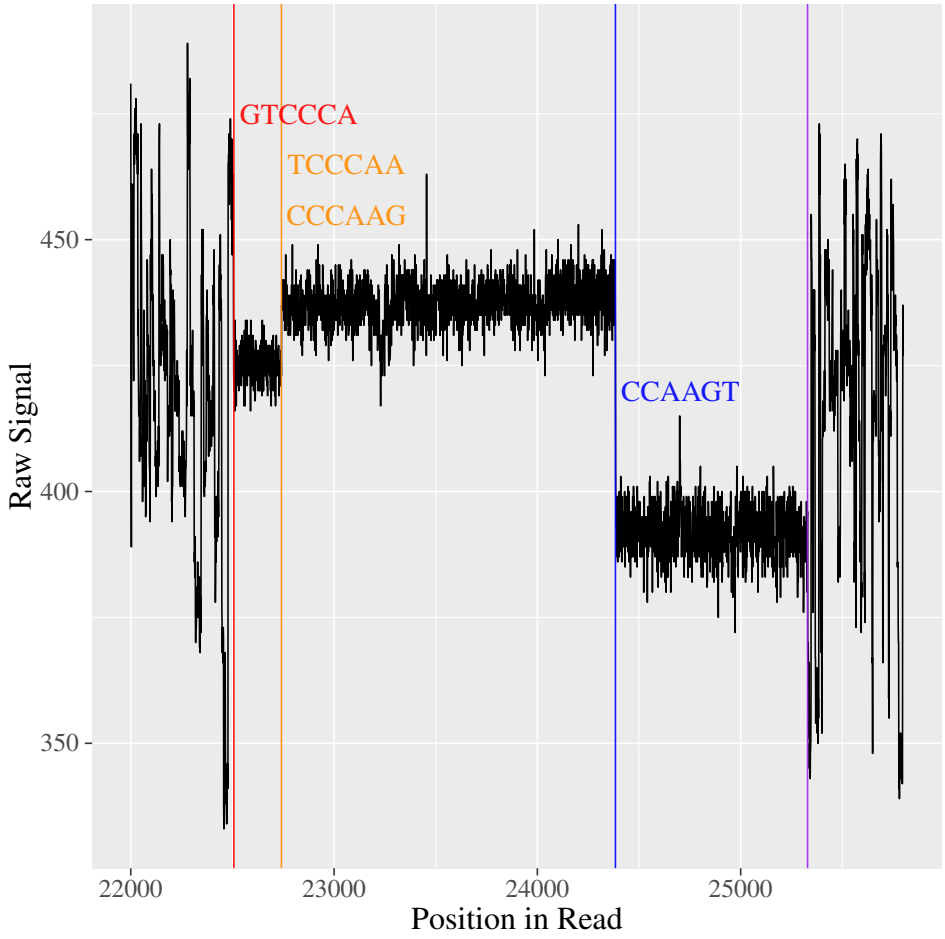


FIGURE 3.7. An example of 3800 data points from a slow section in the read with ID 99671b17-feb4-492b-b119-77daf8e5794e. It is characterised by extended low-variance sections between regular DNA sections. The 6-mers GTCCCA, TCCCAA and CCAAGT are mapped to the regions between the red and orange, orange and blue, and blue and purple vertical lines respectively. CCCAAG is also mapped to the orange-blue region but not as cleanly. In order from left to right the vertical lines are coloured red, orange, blue and purple.

Also, notice that there are more small negative than positive deltas in the histogram. This implies that there are more small decreases in the signal than increases. Next, notice that the right tail extends much further than the left tail in the histogram. This suggests that there are significantly more large positive than negative deltas. In other words, there are more large jumps in the signal than falls. In fact, the probability that a delta is lesser than -128 and greater than 127 is $\sim 6.0 \times 10^{-6}$ and $\sim 3.7 \times 10^{-5}$ respectively. Combining both probabilities we find that the probability that a delta is outside of the signed 1 byte range is $\sim$0.004%. This is very small and will be exploited in section 5.1.

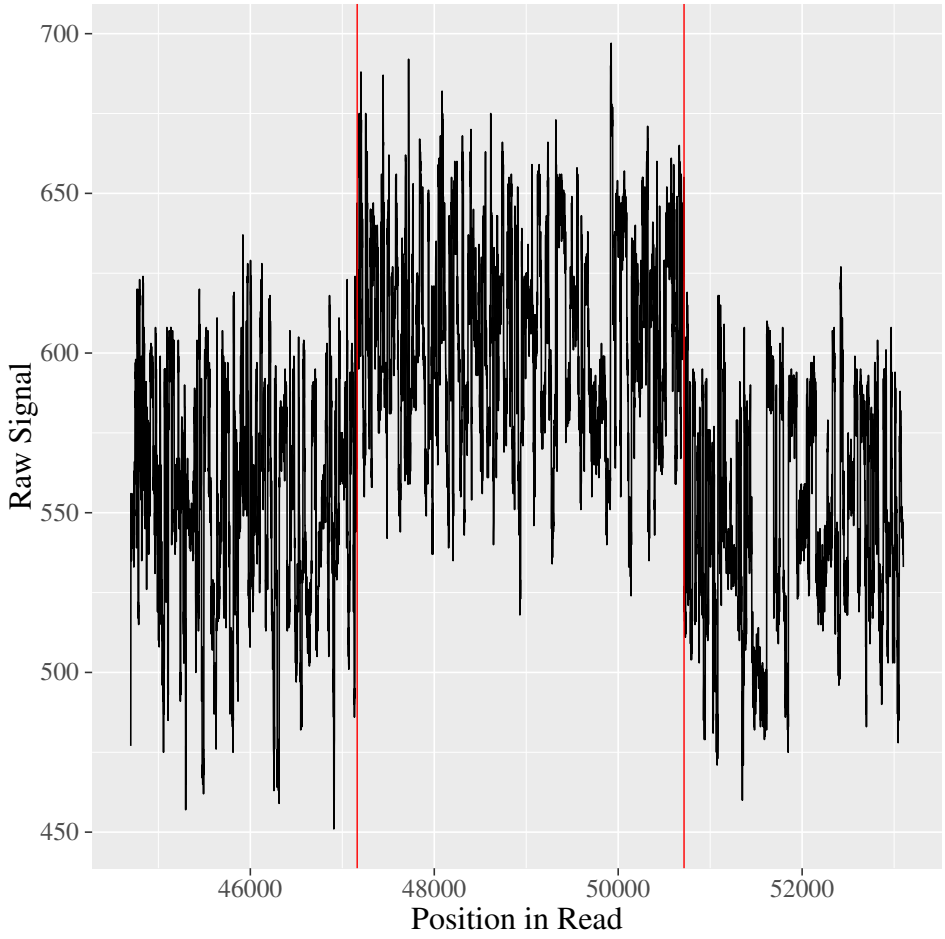


FIGURE 3.8. An example of an incongruent section from the same read used in Figures 3.4, 3.5 and 3.6. The incongruent section's median signal level is ~ 620 compared to the whole's read's median which is ~ 550.

The zig-zag deltas (deltas then zig-zag encoding) range from 0 to 2317 with a median of 4, mean of ~ 15.5679 and standard deviation of ~ 20.6060. See Table 3.7. The zig-zag encoding interleaves the positive and negative deltas resulting in a geometric-like distribution which is not completely decreasing. See Figure 3.13. Notice the 'dotted' right tail; every box with an even offset from value 150 onwards is much larger than those with an odd offset. This is due to the higher frequency of large positive deltas in comparison to large negative deltas, as discussed above.

To observe the positive effect that differential coding has on the compression of nanopore signal data, let's calculate the entropy of the data before and after the delta transformations. This assumes the data is generated by an independent and identically distributed random variable which follows the data's distribution. Table 3.8 shows the resulting calculations. On average for the whole data set, the entropy is 7.70

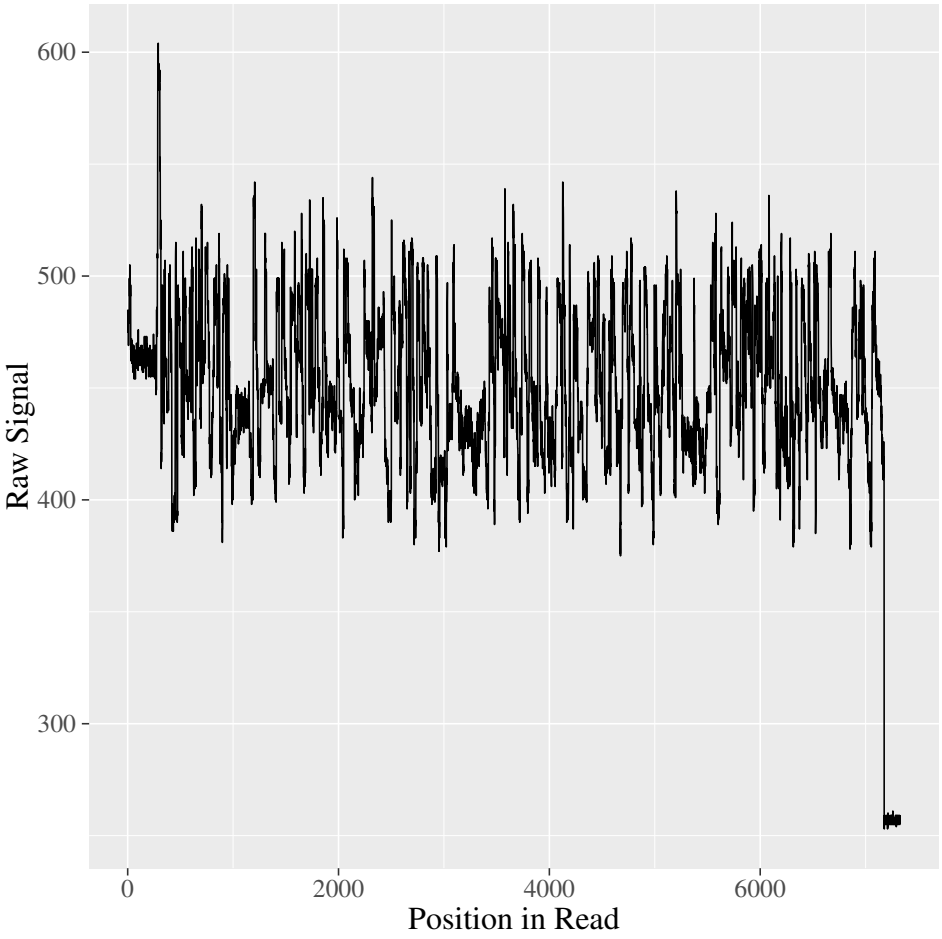


FIGURE 3.9. The read with ID 515e4fd0-8ab1-4845-8866-6772e779712b from the data set. There is an unpredictable fall at the end of the read possibly related to its shorter length of 7329.

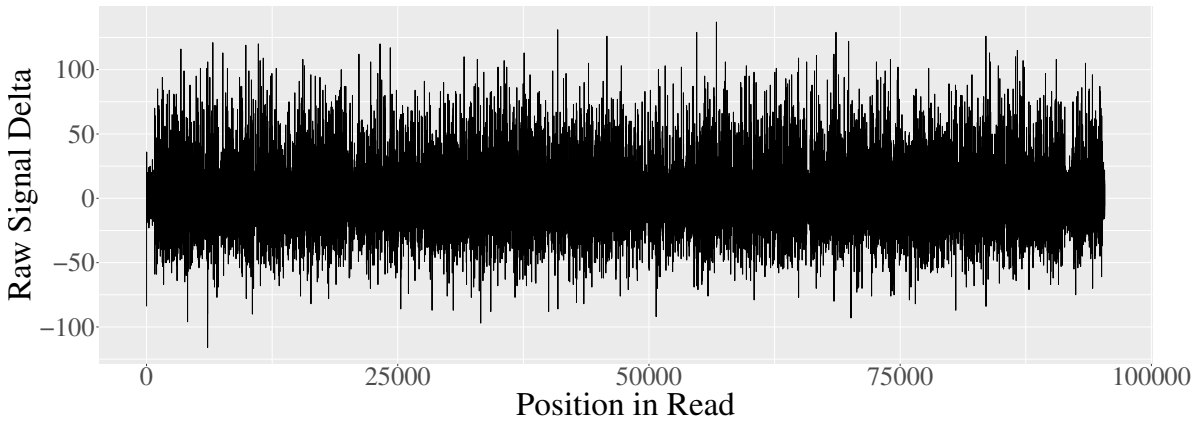


FIGURE 3.10. The deltas of the read with ID e9f08690-171f-476f-9119-5330d0290126. $-4, -2, -10, 16, \ldots$ is the sequence starting from position 25000.

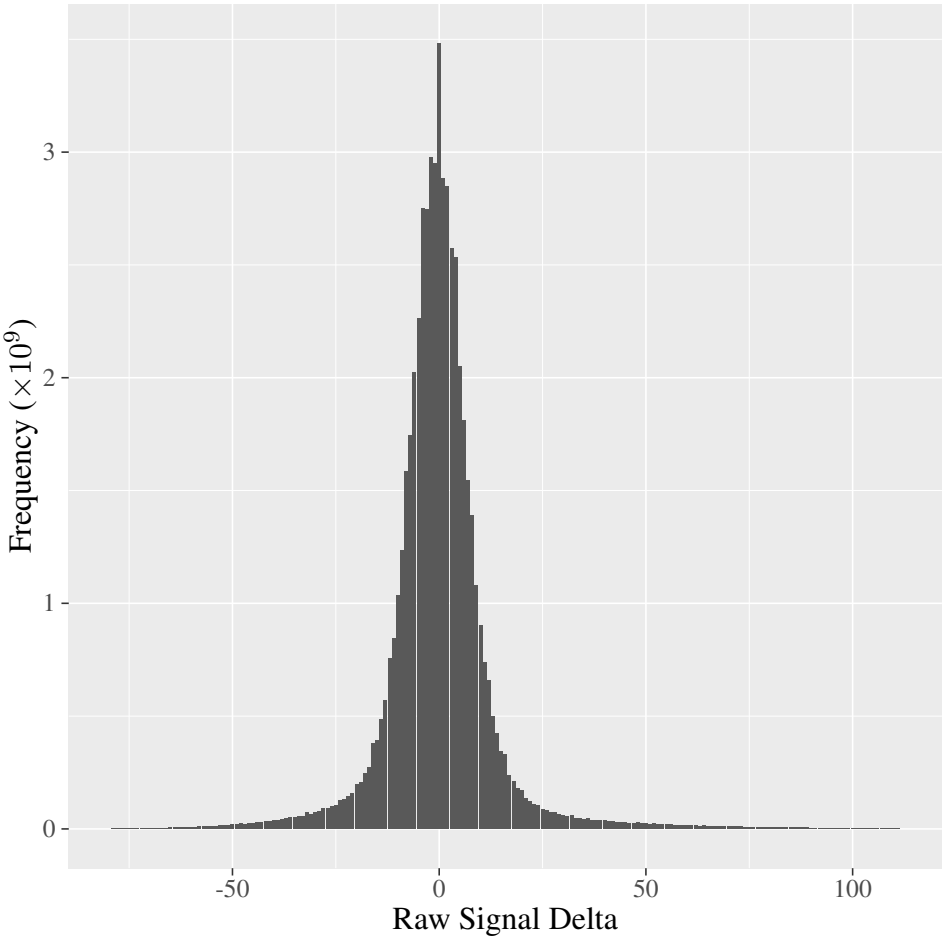


FIGURE 3.11. The frequency of raw signal deltas -80 to 112 in millions for the data set. The deltas outside this range occurred less than 1 million times.

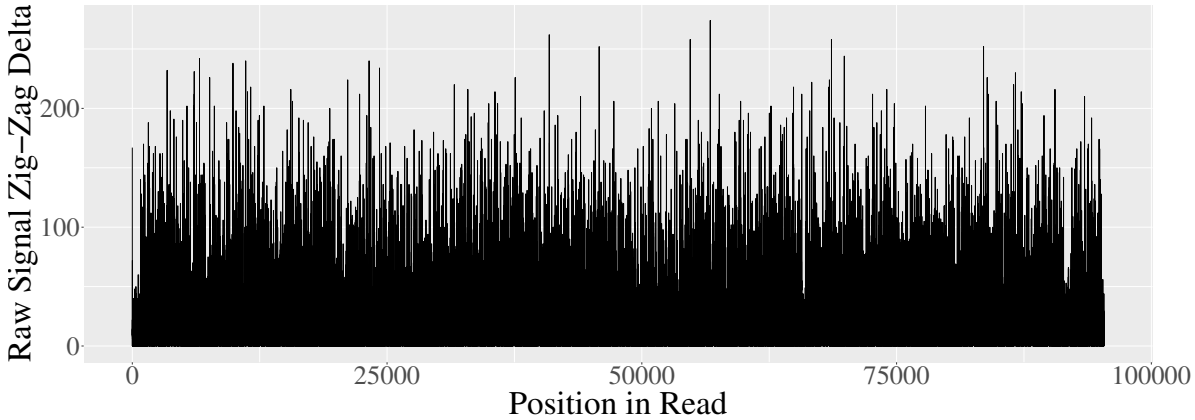


FIGURE 3.12. The zig-zag deltas of the read with ID e9f08690-171f-476f-9119-5330d0290126. $7, 3, 19, 32, \ldots$ is the sequence starting from position 25000.

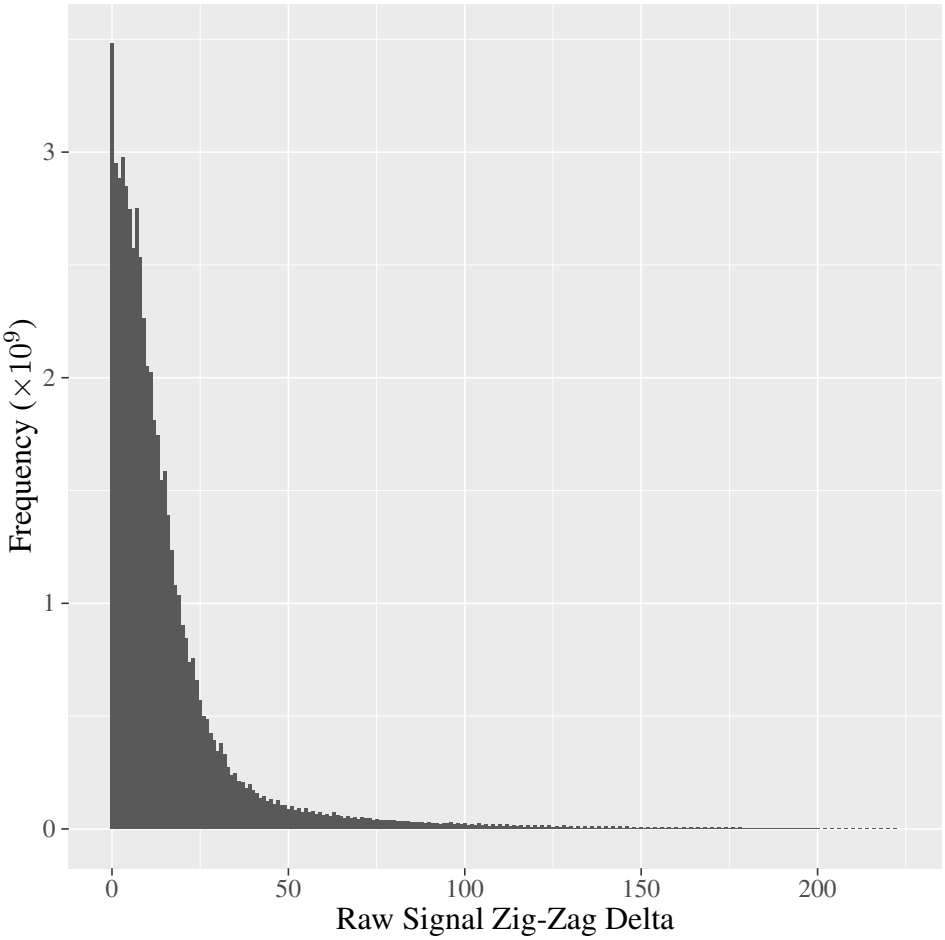


FIGURE 3.13. The frequency of raw signal zig-zag deltas 0 to 224 in millions for the data set. The zig-zag deltas outside this range occurred less than 1 million times.

TABLE 3.8. The entropy of the data and its transformations.

Transformation	Entropy (bits per point)
None	7.699965
Delta	5.391453
Zig-Zag Delta	5.391453

bits per data point compared to 5.39 for the delta and zig-zag delta transformations. This is a difference of 2.31 bits per point which is quite large and clearly demonstrates how differential coding helps take the distribution closer to its true entropy which is unknown. From this it is clear that downstream entropy coding methods would benefit from operating on the deltas rather than the original raw signal data.

CHAPTER 4

Problem Space

What makes a compression method suitable for nanopore signal data?

This is a question we must first answer before proceeding with our search for novel methods. The region of suitable compression methods is known as the *problem space* and is the focus of this chapter. But in order to explore the problem space we must first define it; a task which depends entirely on how one intends to use the data.

The intention behind most holders of nanopore data is either research or clinical diagnosis. In both events, the data is usually useless without first basecalling it, after which it can be analysed using an assortment of downstream analysis software. If it is compressed it will need to be decompressed in order to perform these steps. After analysis the data will typically be put aside for many years with infrequent or no use due to many legal regulations requiring clinical and published research data to be archived for several years.

Hence, there are two main use-cases for nanopore data: analysis and archival which are short- and long-term tasks respectively. In the short-term, a bioinformatician performing analysis would desire a balance between fast analysis speed and a high compression ratio. However, in the long-term, an archivist would desire the highest compression ratio achievable as long as there is still a practical decompression speed. This provides us with two problem subspaces to further define.

Let's define a suitable compression method for nanopore signal data as one which

(1) is lossless and
(2) uses less bits per point than the entropy of the data (7.70 bits per point).

This means that a suitable method has a compression ratio of at least $16/7.70 \approx 2.08$. Moreover, it must be lossless because for most research and all clinical applications accuracy is essential. For the analysis subspace let's further define a suitable compression method as one

(3) which can independently decompress each read.

This is necessary because many reads are typically requested by an analysis program from random locations in the data. Rather than decompressing the whole data set, it is faster to jump to the beginning of a requested read and decompress it independently. Necessarily this means that each read must be compressed separately. Furthermore, this allows multiple read requests to be handled in parallel by a multi-core processor which significantly speeds up analysis.

However, beyond the above definitions, the suitability of a compression method for each subspace is highly subjective, depending on the type of analysis being performed, the amount of data being archived, its frequency of retrieval, the level of access to computational resources, etc. Each specific application demands different requirements from the problem space which are hard to predict. The best we can do is to generally define an order of suitability for each subspace by which one compression method can be definitively compared to another. Given two suitable compression methods M_1 and M_2, we say that M_2 is more suitable than M_1 (i.e. $M_2 >_s M_1$) if

$$\Delta(M_1, M_2) := \delta_2 - \delta_1 - \alpha(t_2^c - t_1^c) - \beta(t_2^d - t_1^d) > 0$$

and M_2 is equally as suitable as M_1 (i.e. $M_2 \equiv_s M_1$) if

$$\Delta(M_1, M_2) = 0$$

where α and β are user-defined constants, δ_i is the space saving and t_i^c, t_i^d are the average times to compress and decompress 1 TiB (in hours) respectively using one thread with method M_i. The space saving which ranges from 0 to 1 is the proportion of the uncompressed size that was reduced via compression. This model is making a lot of assumptions about the linearity of the problem space but let's consider it for simplicity.

Let's define (α, β) for each subspace based on the desires of bioinformaticians working with nanopore signal data. Consider yourself a bioinformatician intending to analyse a nanopore signal data set which you have compressed with method M_1. You envisage a new method M_2 with the same space saving as M_1 ($\delta_2 = \delta_1$) but decompression is an hour quicker per TiB ($t_2^d = t_1^d - 1$). What is the maximum

extra number of hours per TiB that you would be willing to spend during compression to use M_2 over M_1? The answer depends on the bioinformatician and whether decompression is a bottleneck during analysis which is often dominated by basecalling. But let's say you would wait an extra 3 hours per TiB ($t_2^c = t_1^c + 3$). Then, $M_1 \equiv_s M_2$ since any more than 3 hours per TiB and you would instead choose M_1. Hence,

$$\begin{aligned} \Delta(M_1, M_2) &= 0 \\ \delta_2 - \delta_1 - \alpha(t_2^c - t_1^c) - \beta(t_2^d - t_1^d) &= 0 \\ 0 - 3\alpha + \beta &= 0 \\ \beta &= 3\alpha. \end{aligned}$$

Now, imagine a different method M_3 which takes the same time during decompression as M_1 ($t_3^d = t_1^d$) but you save 10% more of the original space (i.e. 102.4 GiB for 1 TiB) ($\delta_3 = \delta_1 + 0.1$). Again, I ask: What is the maximum extra number of hours per TiB that you would be willing to spend during compression to use M_3 over M_1? This is a significant space saving and depending on the amount of time and storage available the answer could range from 2 to 20 extra hours per TiB. However, for the average storage-conscious bioinformatician let's say you would wait up to 10 extra hours per TiB and no longer ($t_3^c = t_1^c + 10$). Then,

$$\begin{aligned} \Delta(M_1, M_3) &= 0 \\ \delta_3 - \delta_1 - \alpha(t_3^c - t_1^c) - \beta(t_3^d - t_1^d) &= 0 \\ 0.1 - 10\alpha - 0 &= 0 \\ \alpha &= 0.01 \\ \implies \beta &= 0.03. \end{aligned}$$

Hence, for the analysis subspace $(\alpha, \beta) = (0.01, 0.03)$. For example, suppose there exists another compression method M_4 which takes the same time during compression as M_1 ($t_4^c = t_1^c$) but takes an

extra hour per TiB during decompression ($t_4^d = t_1^d + 1$). Then, according to our calculations

$$\begin{aligned} \Delta(M_1, M_4) &= 0 \\ \delta_4 - \delta_1 - 0.01(t_4^c - t_1^c) - 0.03(t_4^d - t_1^d) &= 0 \\ \delta_4 - \delta_1 - 0.03 &= 0 \\ \delta_4 &= \delta_1 + 0.03. \end{aligned}$$

That is, you would allow up to an extra hour per TiB during decompression for at least 3% more of the original space saved (i.e. 30.72 GiB for 1 TiB) if you were using the data for analysis.

Now, let's apply the same thought experiments to the archival subspace. Recall that M_2 has the same space saving as M_1 but decompression is an hour quicker per TiB. What is the maximum extra number of hours per TiB that you would be willing to spend during compression to use M_2 over M_1? Since compression is only performed once during archiving and decompression may be performed a handful of times or more, the answer is likely somewhere between 5 and 10 extra hours per TiB depending on the expected frequency of decompression. But let's say you would wait at most 5 extra hours per TiB. Furthermore, recall that M_3 takes the same time during decompression as M_1 but you save 10% more of the original space. Again: What is the maximum extra number of hours per TiB that you would be willing to spend during compression to use M_3 over M_1? Since storage is the primary concern during archiving, the time spent is likely up to 96 extra hours per TiB. This may seem like a long time but remember that this is using only one thread and archives will often need to be stored untouched for several years. Using these answers we find that $(\alpha, \beta) = (\frac{1}{960}, \frac{1}{192})$ for the archival subspace.

These parameters will be used to justify comparisons made in Chapter 6. However, in the meantime let's simply explore the analysis subspace which is a subset of the archival subspace. That is, methods which are (1) lossless, (2) use less bits per point that the entropy of the data and (3) can independently decompress each read. Thus, for the remainder of this thesis we will be investigating lossless per-read compression methods. The bits used per point restriction is not so difficult to achieve as we'll discover once the delta transformation is applied.

CHAPTER 5

Methodology

In this chapter we will design several novel lossless compression methods for nanopore signal data. Firstly, we begin by presenting a more appropriate container for downstream compression known as vbbe21 – with the intention to replace the role of svb. Then, we estimate the effect of applying Huffman coding and range coding to the one byte data of vbbe21.

Attempts to exploit the characteristics of the signal data follow. Optimal and approximating subsequence searching algorithms are presented and analysed. Encoding the stall separately is then explored, followed by a partitioning of the data into three types of sequences: jumps, falls and flats.

All the methods presented are implemented in the GitHub repository available here: `https://github.com/sashajenner/honours`. The benchmarking program is also provided which can be used on any nanopore signal data set as long as it's in the SLOW5 file format.

5.1 One Byte, Two Byte Exceptions

The one byte, two byte exceptions encoding, abbreviated to *vbe21*, encodes a list of integers using one byte for each integer except for integers which cannot fit into one byte, known as *exceptions*, which are encoded using two bytes. See Figure 5.1. Rather than using a control code to mark where normal data

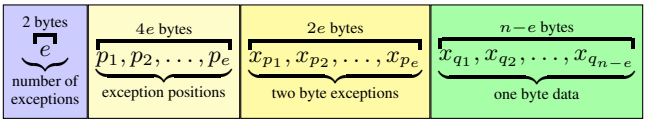


FIGURE 5.1. The vbe21 encoding takes two byte integers $x_1, x_2, \ldots, x_n$ and encodes those which cannot fit into one byte as *exceptions* at the beginning of the stream. There are e exceptions which are recorded by their original positions $p_1, p_2, \ldots, p_e$ and values $x_{p_1}, x_{p_2}, \ldots, x_{p_e}$. Following this is the regular one byte data where q_i is the original position of the i-th one byte data point. This is beneficial when there are few exceptions in the data.

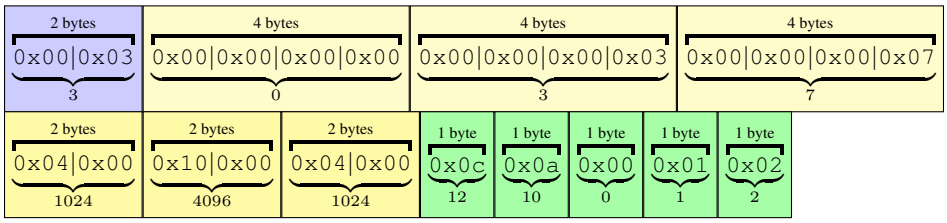


FIGURE 5.2. An example of $1024, 12, 10, 4096, 0, 1, 2, 1024$ encoded with vbe21. The encoding order is read from left to right and top to bottom. There are 3 exceptions located at positions 0, 3 and 7 with values 1024, 4096 and 1024. It uses 25 bytes in total.

and exceptions occur as in the Stream VByte codec (Section 2.2.4.1), the exceptions are encoded at the beginning since it is expected that exceptions will occur with small probability.

The number of exceptions is written using 2 bytes, followed by the exceptions' positions in the list using 4 bytes each, the exceptions themselves using 2 bytes each and finally the regular one byte data. For example, consider the sequence of integers $1024, 12, 10, 4096, 0, 1, 2, 1024$. There are three exceptions 1024, 4096 and 1024. Hence, vbe21 would use

$$2 + 3(4 + 2) + 8 - 3 = 25$$

bytes for this sequence. See Figure 5.2.

Now, consider applying encoding A to a read. Let $C_A : \Omega \to \mathbb{N}_0$ be a random variable measuring the resulting compressed size in bytes where Ω is the space of possible nanopore reads. Then

$$\begin{aligned} C_{vbe21} &= \text{exceptions} + \text{one byte data} \\ &= (2 + 6X) + (N - X) \\ &= 2 + 5X + N \end{aligned} \tag{$*$}$$

where N and X are random variables measuring the read length and number of exceptions respectively. Recall from Section 3.3 that the probability that a zig-zag delta is greater than 255 and therefore outside the one byte range is $\sim 5 \times 10^{-5}$ for the data. Thus, the expected number of zig-zag delta exceptions is

$$E[X] = (5 \times 10^{-5})E[N]$$

where the expected read length is

$$E[N] = 113471.4$$

from Table 3.6. So $E[X] \approx 5.67$, that is we expect roughly 5 to 6 exceptions per read when using the zig-zag delta transformation. Then, the expected compressed size is given by

$$\begin{aligned} E[C_{vbe21-zd}] &= 2 + (2.5 \times 10^{-4})E[N] + E[N] \\ &= 2 + 1.00025E[N] \\ &\approx 113502 \text{ bytes} \end{aligned}$$

where vbe21-zd means first apply the zig-zag delta encoding then vbe21. In comparison, using the Stream VByte 16 (or *svb16*) encoding, the compressed size is given by

$$\begin{aligned} C_{svb16} &= \lceil N/8 \rceil + 2X + (N - X) \\ &= \lceil N/8 \rceil + X + N. \end{aligned}$$

Then, the expected compressed size is

$$\begin{aligned} E[C_{svb16-zd}] &= \lceil E[N]/8 \rceil + (5 \times 10^{-5})E[N] + E[N] \\ &= \lceil E[N]/8 \rceil + 1.00005E[N] \\ &\approx 127661 \text{ bytes} \\ &> E[C_{vbe21-zd}]. \end{aligned}$$

This translates into roughly 8 bits per data point for vbe21-zd versus 9 for svb16-zd. Intuitively these numbers make sense considering vbe21-zd essentially represents most points using one byte each and svb16-zd stores one extra bit per point in the control byte section. Compare this to the entropy of the data which is 7.70 bits per point or 5.39 for the zig-zag deltas. So it is clear that vbe21-zd saves more space than svb16-zd: roughly 14 KiB on average per read or an estimated 6.6 GiB for the whole data set. However, this is merely a container for the data which is useful for downstream compression; alone it doesn't provide great compression results as is obvious when comparing it to the entropy of the data.

The encoding time complexity of vbe21-zd is linear since it requires applying the zig-zag delta transformation and checking which values are exceptions. The same time complexity is true for decoding since the inverse process is conducted. Recall from the literature review that the time complexity of the Stream VByte codec family is also linear. Hence, both strategies have the same asymptotic time complexity for encoding and decoding.

TABLE 5.1. Summary statistics of the number of exceptions per read and the zig-zag delta exceptions themselves in the data.

Statistic	Number of Exceptions Per Read	Exceptions
Min	0	256
Q1	1	260
Q2	3	266
Q3	6	278
Max	3616	2317
Mean	4.8546	277.0465
Mode	0	256
SD	18.0595	36.8207

5.1.1 Exceptions Encoding

There are better ways of storing the exceptions than in the raw sequential format of vbe21. As we have seen there are usually 5 to 6 exceptions per read meaning that storing the number of exceptions using two bytes, which has a range of 0 to 65535, for all reads is wasteful. In particular, the number of exceptions per read ranges from 0 to 3616 and 99% of the reads have between 0 and 42 exceptions. This is much smaller than 65535. See Table 5.1 for an statistical overview of the zig-zag delta exceptions. Its distribution, shown in Figure 5.3, could be nicely fitted by an exponential distribution. In fact, around 20% of the reads have no exceptions at all. There is clearly wasted space being used.

Instead of storing two bytes for the number of exceptions we can bit pack the number of exceptions using four bits to represent the number of bits used. Refer back to Section 2.2.4.1 for a review of bit packing. The maximum number of exceptions, 3616, requires at least 12 bits to represent it. Using the bit packing approach, $4 + 12 = 16$ bits or two bytes would be used to represent it – which is equivalent to the naive approach. That is, in the worst case bit packing uses the same space as the naive approach. In the best case when there are zero exceptions only four bits are required, and as long as there are less than 16 exceptions up to one byte is used. Although this approach clearly saves space, we are talking about saving at most 12 bits which is miniscule compared to the scale of the data.

Similarly, we can store the exceptions' data in a more compact form. The exceptions themselves range from 256 to 2317 but 99% of the exceptions do not go higher than 481. The mean and mode are $\sim$277 and 256 respectively. Refer back to Table 5.1 for a summary and Figure 5.4 for the histogram. Notably, the frequency of even-numbered exceptions is much greater than adjacent odd-numbered exceptions from 256 up to $\sim$330. This is due to large positive deltas occuring more frequently than large negative

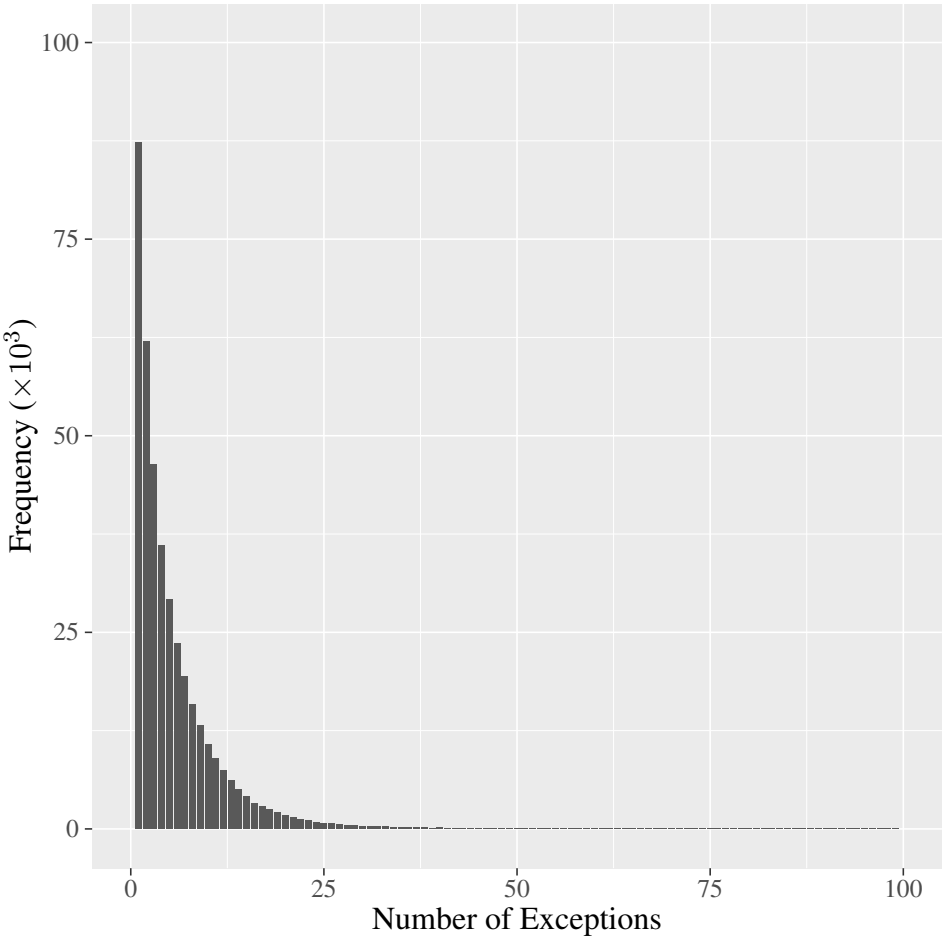


FIGURE 5.3. Histogram of the number of exceptions per read up to 100 which appears to nicely follow an exponential distribution. Values after 100 occur highly infrequently – there are only 546 out of $500\,000$ reads with more than 100 exceptions.

deltas as previously discussed in Section 3.3. One simple approach to encoding the exceptions is to subtract 256 since this is the minimum exception value and then apply bit packing using 4 bits as before. The average exception can then be represented using 0 to 5 bits which is better than the naive 16-bit per exception encoding. This results in an expected decrease of $(16 - 5) \times 5 - 4 = 51$ bits per read. In the worst case, the number of bits used per exception for this data will be 12 or $4 + 12e$ in total since 2317 is the maximum exception. This is better than the naive approach for $e > 1$ and equivalent for $e = 1$. The implications of this is that bit packing the exceptions' data never consumes more space than the naive strategy.

Another improvement we can make is on the storage of the exceptions' positions. Since the positions are strictly increasing we can take the deltas between each position and subtract one. This will result in

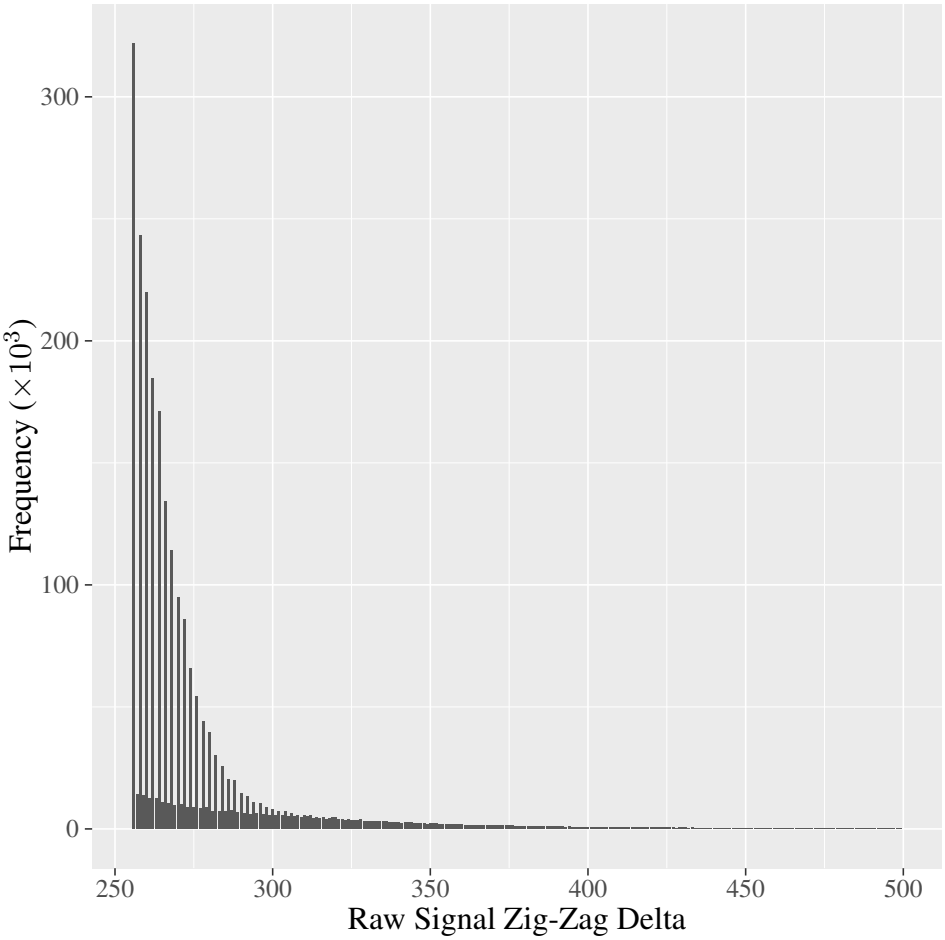


FIGURE 5.4. Histogram of the zig-zag delta exceptions up to 500. This figure is the tail of Figure 3.13. The right and left tail of Figure 3.11 are 'meshed' together during the zig-zag transformation causing the stripped pattern.

smaller values which we can then bit pack using 5 bits instead of 4 for the number of bits used. 5 bits are required instead of 4 to account for any outliers given the maximum read length is roughly 6 million. If 4 bits were used only position deltas up to $2^{2^4-1} - 1 = 32767$ could be represented which is clearly short of 6 million.

The strategy which combines all of the above exception bit packing strategies and still stores the regular one byte data we will name *compact vbbe21*. Notice the extra 'b' in *vbbe21*. This refers to the bit packing strategy in use. See Figure 5.5 for a visual representation of this encoding. Notice the byte boundary padding which aligns the one byte data to byte boundaries for downstream compression methods. For implementation simplicity consider a similar strategy which does not bit pack the number of exceptions, bit packs the exception' positions and data using one byte for the number of bits used rather than 5 and 4

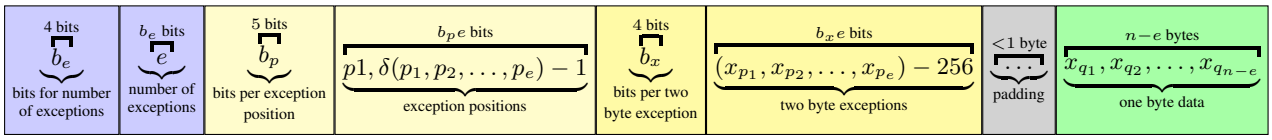


FIGURE 5.5. The compact vbbe21 encoding takes n unsigned 16-bit integers and finds those which cannot fit into one byte. These are encoded by bit packing the number of exceptions, the deltas of the exceptions' positions and the two byte exceptions subtracted by 256. Less than one byte is used for padding to align the bit packed data to the next byte boundary. Then, the one byte data is recorded as in vbe21.

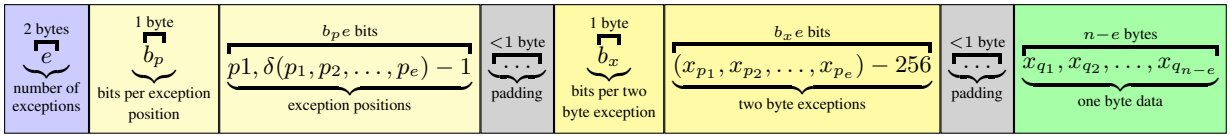


FIGURE 5.6. The vbbe21 encoding takes n unsigned 16-bit integers and finds those which cannot fit into one byte. These are encoded by storing the number of exceptions using two bytes, bit packing the deltas of the exceptions' positions and bit packing the exceptions themselves. Bit packing is performed using one byte for the number of bits and padding is used between the exception's positions, data and the one byte data. This is easier to implement than compact vbbe21.

respectively, and uses padding between the exceptions' positions and data. We shall name this strategy the *regular vbbe21* encoding or *vbbe21* for short. See Figure 5.6 for a pictorial representation.

The storage gap between vbbe21 and compact vbbe21 has an upper bound of

$$\begin{aligned}\text{vbbe21} - \text{compact vbbe21} &= \text{number of exceptions gap} + \text{positions gap} \\ &\quad + \text{two byte exceptions gap} + \text{padding gap} \\ &< (16-4) + (8-5) + (8-4) + 16 \\ &= 35\end{aligned}$$

bits and a lower bound of

$$\text{vbbe21} - \text{compact vbbe21} > (16-12) + (8-5) + (8-4) - 8 = 3$$

bits. That is, compact vbbe21 saves between 3 and 35 bits compared to regular vbbe21. However, an expected measure of the storage gap is a more useful metric for comparison. Let's name the non–one-byte-data where the exceptions are handled the *exceptions section*. Using Equation $*$, the expected size of the vbe21-zd exceptions section in bytes is given by

$$2 + 6E[X] = 36.02.$$

TABLE 5.2. The estimated expected size of the exceptions section for encodings vbe21-zd, vbbe21-zd and compact vbbe21-zd on the data.

Method	Expected Size of Exceptions Section (Bytes)
vbe21-zd	36.02
vbbe21-zd	18
compact vbbe21-zd	15.56

Now, round up the expected number of exceptions from Table 5.1 so it is 5. Then, the expected number of bits used for the number of exceptions is roughly 3. Let the expected number of bits used on the positions be 16 since the expected read length is $113\,471$ and this accounts for position deltas up to $65\,535$. Furthermore, let the expected number of bits used on the exceptions be 5 since the average exception is 277 and this accounts for exceptions up to 287. Also, let the expected padding size be 3.5 bits as this is halfway on the padding's range of 0 to 7 bits. Then, an estimate of the expected size of the vbbe21-zd exceptions section is given by

$$\begin{aligned}&\text{number of exceptions} + \text{positions} + \text{two byte exceptions} + \text{padding}\\ &= \frac{16 + (8 + 5 \times 16) + (8 + 5 \times 5) + 2 \times 3.5}{8}\\ &= 18\end{aligned}$$

bytes, versus

$$\frac{(4 + 3) + (5 + 5 \times 16) + (4 + 5 \times 5) + 3.5}{8} = 15.56$$

bytes for the compact vbbe21-zd exceptions section. These sizes are displayed in Table 5.2 for ease of comparison. Now, we can see that the estimated storage gap between vbbe21-zd and compact vbbe21-zd is 2.44 bytes or 19.5 bits. Furthermore, vbbe21-zd and compact vbbe21-zd are expected to save 17 and 19.44 bytes per read respectively when compared to vbe21-zd. So, there is a storage benefit to bit packing the exceptions over storing them in a raw sequential format.

In comparison with vbe21, the compact and regular vbbe21 algorithms still take linear time during encoding and decoding although they require more computational operations to encode and evaluate the bit packed data. However, the exceptions section usually accounts for less than 0.05% of the vbe21-zd encoding compared to the one byte data. Hence, focussing on compressing the one byte data section has much more potential benefit. This will be heavily explored in the remaining sections.

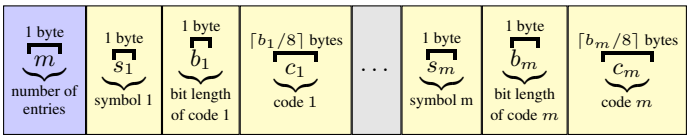


FIGURE 5.7. The naive encoding of the Huffman table where c_i is the Huffman code of symbol s_i with bit length b_i; i ranges from 1 to m (the number of table entries).

5.1.2 Huffman

Consider compressing the one byte data using the Huffman (shortened to *huff*) coding algorithm. This requires determining the data's distribution on an initial pass. Then, the Huffman table is recorded and each byte is encoded with its Huffman code. The problem is that there is an overhead with storing the table. Naively, one can store the table by writing the number of entries in the table (1 byte) then each entry's symbol (1 byte), code length (1 byte) and code (code length in bytes). See Figure 5.7 for a picture. The resulting table consumes

$$1 + 2m + \sum_{i=1}^{m} \lceil b_i/8 \rceil$$

bytes where $m \in \mathbb{Z} \cap [0, 255]$ is the number of entries in the table and b_i is the length of the i-th entry's code in bits. The number of entries is the number of unique one byte values in the data. If we use huff on the one byte zig-zag deltas of the vbbe21-zd encoding (let's name this strategy *huff-vbbe21-zd*), the number of entries is dependent on the read's length. Short reads may only contain 100 unique one byte deltas whilst long reads may easily contain more than 250 entries. If we estimate that this is roughly 200 and the average code length is 2 bytes then the table consumes

$$1 + 2 \times 200 + 200 \times 2 = 801$$

bytes. The maximum number of entries is 256 so the maximum table size is roughly 1025 bytes.

However, rather than encoding the exact Huffman table for each read and consuming roughly 800 bytes in storage, it makes more sense to use a shared Huffman table which approximates the zig-zag delta distribution of most reads well. We are at an advantage here in that the zig-zag deltas follow a similar distribution between reads. For example, 0 is the most common zig-zag delta among reads so naturally its code will have the fewest number of bits and this will be efficient for the majority of reads. One simple approach is to construct the optimal Huffman table for the whole data set of zig-zag deltas. Then, we can store the code table once in the source code and encode each read using the same table. Let's call

this method the static Huffman algorithm which is being applied to the one byte zig-zag deltas, so *shuff-vbbe21-zd* for short. This table consumes 1042 bytes but we can store it statically alongside the source code, so unless we are considering the Kolmogorov complexity of the data its size is not calculated in the compression size. The distribution of code lengths from this table is shown in Figure 5.8. The figure shows the distribution splits into two alternating distributions from about 116 onwards which represents the higher frequency of large positive compared to large negative deltas, as previously discussed.

huff-vbbe21-zd consumes more space than the shuff-vbbe21-zd algorithm since it records a different Huffman table for each read in the compressed data. The space consumed by the Huffman table turns out to be more than the size difference between the globally approximated and read-optimal Huffman encoding of the zig-zag deltas. Furthermore, huff-vbbe21-zd takes longer than shuff-vbbe21-zd during compression and decompression. During compression, huff-vbbe21-zd must do an initial pass of the read to count the data's frequencies and then construct the Huffman tree and table. During decompression, huff-vbbe21-zd must read the table and construct the tree before decoding the data. shuff-vbbe21-zd doesn't need to perform any of these operations since it pre-calculates the shared table and tree and stores them in static memory.

During compression, calculating the frequencies takes $O(n)$ time. Then, constructing the tree takes $O(x^2 \log x)$ time using the bottom-up construction algorithm where $0 \leq x \leq 256$ is the number of unique bytes or exceptions when being applied to vbe21. Constructing the table requires traversing each branch on the tree which takes $O(x)$ time since there are $2x - 1$ nodes in a Huffman tree. Then, recording the table takes $O(x)$ time and encoding the bytes takes $O(n)$ time using the table. In total, the huff-vbbe21-zd algorithm takes $O(n + x^2 \log x)$ time for compression while shuff-vbbe21-zd takes $O(n)$ time. Compared to n, the size of the input, x is usually a lot smaller so it has a small but true effect on the compression time.

During decompression, huff-vbbe21-zd take $O(n + x)$ time since it must construct the tree whilst shuff-vbbe21-zd takes $O(n)$ time. But in practicality they take the same time. All in all, as long as there is a space benefit to using shuff-vbbe21-zd, which there usually is if the approximation is good, there is no reason to using huff-vbbe21-zd since shuff-vbbe21-zd is faster during compression and decompression.

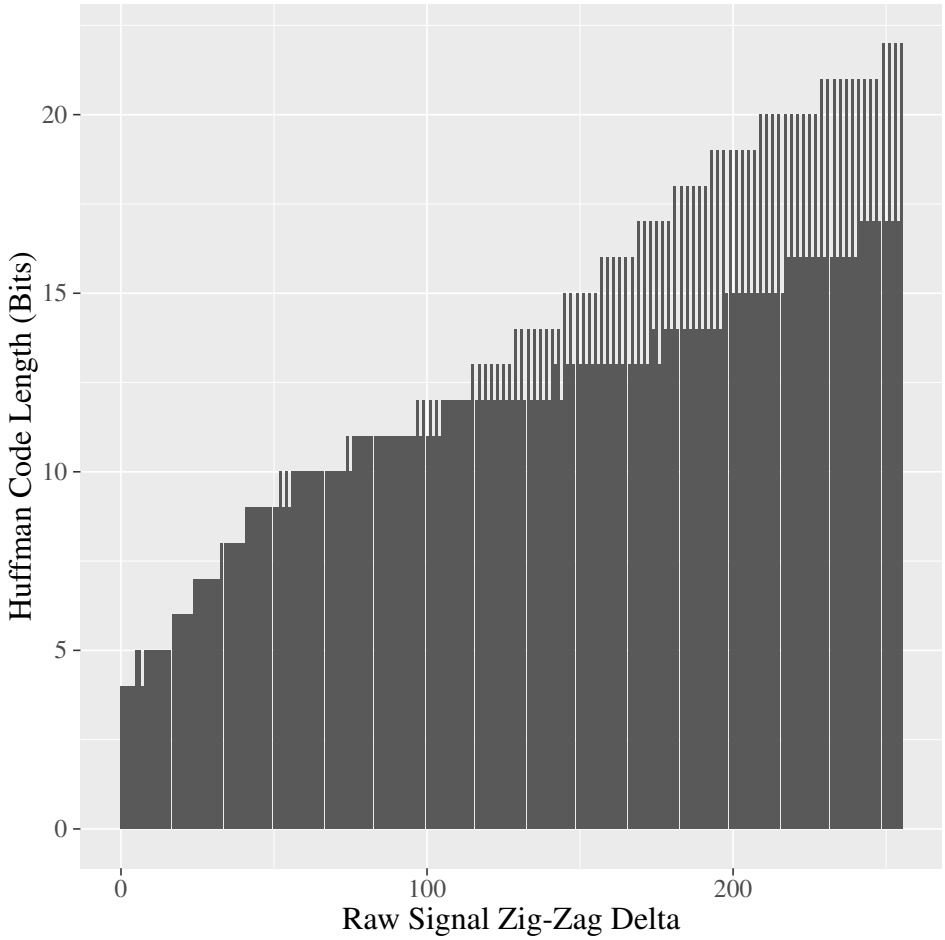


FIGURE 5.8. The Huffman code length of each one-byte zig-zag delta from the shared Huffman table, generated using the frequency distribution of the whole data set.

5.1.3 Range Coding

Now consider encoding the one byte data using range coding rather than Huffman coding. Recall from Section 2.2.2.2 that range coding encodes all of the data into one number. To begin with, an initial range of numbers is chosen. Then, for each symbol in the data stream the current range is narrowed down based on the current symbol's probability of occurrence. A representative number from the final range is then chosen as the output.

The probability distribution of the data can be predetermined, calculated by an initial pass or predicted adaptively. Most available implementations use models for predicting the probability distribution of the next symbol. Some common models include order-0, order-1 and order-2 models, context mixing and

secondary symbol estimation (SSE). These models are used during encoding and decoding meaning that no distribution needs to be encoded with the compressed data.

The order-n model gives the probability distribution of the next symbol given the previous n symbols (known as the *context*). It is updated at each step to increase the revealed symbol's probability when the same context appears again. Basically the order-n model queries and updates a frequency table of size S^{n+1} where S is the size of the symbol's alphabet. It finds the probability of the next symbol by querying the table and dividing this count by the total count of the symbols read thus far. To avoid a probability of 0 occuring, every value in the table is initialised to the same count – typically a small value between 2 and 10 or a much larger value such as 2^{14} as in TurboRC. There are bytewise and bitwise models which operate at the byte and bit level respectively. Bytewise models operate over symbols from the alphabet $\{0, 1, \ldots, 255\}$ whilst bitwise models keep track of the frequency of the binary digits 0 and 1. Let the range coding algorithms which use the order-0 and order-1 models respectively be named *rc0* and *rc1*.

Context mixing algorithms combine two models to yield higher accuracy predictions. They predict one bit at a time and are usually based on one of two mixing algorithms: linear or logistic mixing. Linear mixing performs a weighted average of the models whilst logistic mixing transforms the probabilities using logistic modelling which gives more weight to probabilites closer to 0 and 1.

Secondary symbol estimation (SSE) is a prediction postprocessing approach which uses the combined prediction of context mixing and a small context to generate a refined prediction.

Consider the range coding algorithm which applies order-0 and order-1 context mixing with SSE. We'll abbreviate this strategy to *rc01s*. The TurboRC implementation initialises the order-0, order-1 and SSE models in $O(1)$ time. Then, for each byte it iterates over its bits and updates the range followed by the models. A normalisation procedure built into the encoding algorithm is required to ensure that there is always a large enough range for further sub-ranges. Overall, the algorithm is $O(n)$ but since it iterates over bits rather than bytes and there are many update procedures – it is slower than huff in practice. The same is true for decoding. However, the compression ratio of rc01s is estimated to be better than huff and shuff. This is because range coding is as or more optimal as Huffman coding but without the drawback that each code length must be a whole number of bits. Furthermore, combined with accurate modelling range coding performs just as well without requiring the storage of a probability distribution as in huff.

5.2 Exploiting the Signal

As discussed in Section 3.2, there are several predictable characteristics among reads which can be exploited to gain more compression. We have already used statistical information about the zig-zag deltas to our advantage in the vbbe21 encoding (Section 5.1) but we can even further exploit the signal's characteristics by intimately tying this knowledge to a compression algorithm. This understanding should improve the compression ratio since compression is fundamentally an artificial intelligence problem of understanding and prediction.

5.2.1 Subsequence Searching

Recall that there are many sections of a read with very different behaviour. See Section 3.2 for a recap of the different types of sections. The position and presence of these sections is often predictable. For example, the stall usually begins within the first 50 data points and is a highly likely occurrence. However, many sections such as the homopolymer and slow sections occur unpredictably both in terms of position and frequency. Furthermore, incongruent sections with uncommon behaviour are difficult to handle.

Hence, one idea is to divide the signal into adjacent blocks which capture these sections, compress each block separately and then combine the resulting data into a final compressed stream. The problem then becomes how to divide the blocks such that the compression ratio is minimised. This problem is formally defined below.

Consider partitioning the nanopore signal into adjacent variable length blocks. Recall that the signal can be represented by

$$x = (x_0, x_1, \ldots, x_{n-1}).$$

Let $s = (s_0, s_1, \ldots, s_{m-1} \mid s_i < s_{i+1})$ be the partitioner of x such that the signal is divided into m blocks with the j-th block starting at index s_j and ending at index $s_{j+1} - 1$ except for final block which starts at index s_{m-1} and ends at index $n - 1$. Let $P(x, s)$ be the resulting partitioning of x according to partioner s defined by

$$P(x, s) = (p_j) := ((x_{s_j}, x_{s_j+1}, \ldots, x_{s_{j+1}-1}))_{j \in \mathbb{Z} \cap [0,m)}.$$

For example, if $x = (656, 527, 515, 527, 526)$ and the partitioner $s = (0, 3, 4)$ then there are $m = 3$ blocks beginning at indices 0, 3 and 4 such that

$$P(x, s) = ((656, 527, 515), (527), (526)).$$

Given the partitioning P we would like to compress each block p_j separately and concatenate the results. Let $C_M(p)$ be the compressed bytes of block p after applying compression method M. Then, the compressed bytes of signal x under partitioner s is the concatenation of the compressed bytes of each block $C_M(p_j)$ given by

$$C_M(x, s) := (C_M(p_j) \mid p_j = P(x, s)_j).$$

The compressed size of the partitioning is equal to the sum of each block's compressed size:

$$|C_M(x, s)| = \sum_j |C_M(p_j)|.$$

Now, our desire is to minimise the compressed size $|C_M(x, s)|$ by choosing the optimal partitioner s. Let $\hat{s}$ be the partitioner of x which minimises the number of compressed bytes. That is,

$$|C_M(x, \hat{s})| = \min_s |C_M(x, s)| = \min_s \sum_j |C_M(p_j)|.$$

We can solve this optimisation problem by using the following recursive relationship

$$|C_M(x, \hat{s})| = |C_M(x)| \qquad n = 1$$

$$|C_M(x, \hat{s})| = \min\{|C_M(x)|, \min_{0 \le k \le n-2} \{|C_M(x_L, \hat{s}_{x_L})| + |C_M(x_R, \hat{s}_{x_R})|\}\} \qquad n \ge 2$$

where $x_L = (x_0, x_1, \ldots, x_k)$ and $x_R = (x_{k+1}, \ldots, x_{n-1})$. That is, the compressed bytes with minimum length are found by either compressing the whole signal as usual or by dividing the signal into two blocks and concatenating each block's optimal compressed bytes. Algorithm 1 uses the above relationship to calculate the minimum compressed size.

Algorithm 1 Find the minimum compressed size $|C_M(x, \hat{s})|$ recursively.

```
n ← |x|
l ← |C_M(x)|
if n ≥ 2 then
  for k = 0 to n − 2 do
    x_L ← (x_0, x_1, ..., x_k)
    x_R ← (x_{k+1}, ..., x_{n−1})
    l_split ← |C_M(x_L, ŝ)| + |C_M(x_R, ŝ)|
    if l_split < l then
      l ← l_split
    end if
  end for
end if
return l
```

Let's analyse the time complexity of this algorithm. For a signal of size n, the number of comparisons is given by the recurrence relation

$$\begin{aligned} c_1 &= c_{M,1} \\ c_n &= c_{M,n} + \sum_{k=0}^{n-2}(c_{k+1} + c_{n-k-1} + 1) && n \geq 2 \end{aligned}$$

where $c_{M,n}$ is the number of comparisons for the compression method M on input size n. This can be simplified as follows

$$\begin{aligned} c_n &= c_{M,n} + \sum_{k=1}^{n-1}(2c_k + 1) \\ &= c_{M,n} + n - 1 + 2\sum_{k=1}^{n-1} c_k && n \geq 2 \end{aligned}$$

Let's write c_n in terms of c_{n-1} in order to solve the recurrence more easily.

$$\begin{aligned} c_{n-1} &= c_{M,n-1} + n - 2 + 2\sum_{k=1}^{n-2} c_k && n \geq 3 \\ c_n - c_{n-1} &= c_{M,n} - c_{M,n-1} + 1 + 2c_{n-1} \\ c_n &= c_{M,n} - c_{M,n-1} + 1 + 3c_{n-1} && n \geq 2 \end{aligned}$$

Unrolling this we find that

$$c_n = c_{M,n} + \sum_{k=1}^{n-1} 3^{k-1}(2c_{M,n-k} + 1) \qquad n \geq 2.$$

If the compression method has linear time complexity (i.e. $c_{M,n} = O(n)$) then

$$\begin{aligned} c_n &= O(n) + \sum_{k=1}^{n-1} 3^{k-1}(2O(n-k)+1) \\ &= O(n) + \sum_{k=1}^{n-1} 3^{k-1}O(n-k) \\ &= O(3^n) \qquad \text{See Appendix 8.1} \end{aligned}$$

That is, it takes exponential time to find the minimum compressed size using this recursive algorithm given M takes linear time during compression.

However, this algorithm naively re-computes the compressed bytes of each block many times rather than calculating it once and caching the data. A better approach is to use bottom-up dynamic programming which avoids recursion and hence stack overflows in implementation. The approach would compute the minimum number of compressed bytes for all blocks of length 1 then 2, 3 and so forth up to n. The results are stored in a triangular matrix $OPT \in \mathbb{N} \times \mathbb{N}$ where $OPT_{i,j}$ is the minimum number of compressed bytes of the block starting at index i and ending at index j. It is triangular since we are not interested in blocks which end before they begin. That is, we require that $i \leq j$. Algorithm 2 shows the pseudocode for this algorithm.

Algorithm 2 Find the minimum compressed size $|C_M(x, \hat{s})|$ using dynamic programming.

```
n ← |x|
for k = 1 to n do
  for i = 0 to n − k do
    y ← (x_i, ..., x_{i+k−1})
    l ← |C_M(y)|
    for j = i to i + k − 2 do
      l_split ← OPT_{i,j} + OPT_{j+1,i+k−1}
      if l_split < l then
        l ← l_split
      end if
    end for
    OPT_{i,i+k−1} ← l
  end for
end for
return OPT_{0,n−1}
```

For a signal of size n, the algorithm has the following number of comparisons

$$\tilde{c}_n = \sum_{k=1}^{n}(n-k+1)(c_{M,k}+k-1)$$

since there are $n-k+1$ blocks of size k and each must compare compressing itself to concatenating $k-1$ pairs of compressed subdivisions. Suppose as before that the compression method takes linear time. Then $\tilde{c}_n$ simplifies to

$$\begin{aligned}\tilde{c}_n &= \sum_{k=1}^{n}(n-k+1)(O(k)+k-1) \\ &= \sum_{k=1}^{n}(n-k+1)O(k) \\ &= O(n^3) \qquad \text{See Appendix 8.1}\end{aligned}$$

This is cubic in time complexity which is an improvement from exponential time but at the cost of storing OPT which uses $O(n^2)$ space.

We can speed up the algorithm without a significant loss in compression by ignoring blocks smaller than k' and moving between block lengths and blocks of the same length at a step of δ where k' is divisible by δ and $k' > \delta$. The idea is that for compression method M, there should exist some block size k' for which all blocks smaller than k' are minimally compressed without partitioning with high probability. Whilst all blocks of size greater than or equal to k' have a higher probability of being compressed smaller by partitioning. That is, $\exists k' \in \mathbb{N}^+$ such that for block y

$$P(C_M(y, \hat{s}) = C_M(y) \mid |y| < k') > 1 - \epsilon$$

and

$$P(C_M(y, \hat{s}) = C_M(y) \mid |y| \geq k') \leq 1 - \epsilon$$

with small $\epsilon \leq 0.05$. Furthermore, it is not necessary to calculate blocks of all lengths or all blocks of the same length since blocks which differ by a couple signals should not provide a significantly better partition. Rather, we want to calculate the minimum number of compressed bytes for blocks of length $k', k'+\delta, k'+2\delta, \ldots, k'+\lfloor\frac{n-k'}{\delta}\rfloor\delta$. Then for blocks of a particular length k we calculate the minimum number of compressed bytes for blocks δ apart, of the form $(x_{b\delta}, x_{b\delta+1}, \ldots, x_{b\delta+k-1})$ where $0 \leq b \leq \lfloor\frac{n-k}{\delta}\rfloor$. Furthermore, for the final block of length n we simply take the partitioning of the largest block of length $\lfloor\frac{n-k'}{\delta}\rfloor\delta$ which may or may not equal n. See Algorithm 3 for the pseudocode of this approximation algorithm.

Algorithm 3 Approximate the minimum compressed size $|C_M(x, \hat{s})|$ using dynamic programming.

$n \leftarrow |x|$
for $a = 0$ to $\lfloor \frac{n-k'}{\delta} \rfloor$ **do**
  $k \leftarrow k' + a\delta$
  for $b = 0$ to $\lfloor \frac{n-k}{\delta} \rfloor$ **do**
    $i \leftarrow b\delta$
    $y \leftarrow (x_i, \ldots, x_{i+k-1})$
    $l \leftarrow |C_M(y)|$
    for $c = 0$ to $\lfloor \frac{k-k'}{\delta} \rfloor$ **do**
      $j \leftarrow i + k' + c\delta$
      $l_{split} \leftarrow OPT_{i,j-1} + OPT_{j,i+k-1}$
      if $l_{split} < l$ **then**
        $l \leftarrow l_{split}$
      end if
    end for
    $OPT_{i,i+k-1} \leftarrow l$
  end for
end for
return $OPT_{0,\lfloor \frac{n-k'}{\delta} \rfloor \delta - 1}$

This should not improve the asymptotic time complexity of the algorithm but significantly decrease its running time in practice by a factor of δ or even greater. This is because by stepping to the next block or block size by δ, the algorithm is effectively addressing only $1/\delta$ of the problem.

These optimal-seeking algorithms have a compression time complexity which is unfavourable in practice, especially considering the size of each read which can range up to roughly 6 million points. Furthermore, the cubic running time of the dynamic programming algorithm makes the poor assumption that the time complexity of the base compression algorithm is linear. This is not always true or desirable. It also requires quadratic space to maintain the minimum compressed size calculations. The space required is further increased when modifying the algorithm to actually compress the input since it must maintain the compressed data of each block or instead the optimal partitioning of each subsequence. Decompression of the blocks should not be time consuming and can be done in $O(mT)$ time where m is the number of blocks and T is the time complexity of the base compression algorithm during decompression. We can likely treat m to be some function of n so this is polynomial is the best of cases but a lot faster than compression.

For all the above reasons, a faster approximation of the optimal blocks is desired to improve the compression ratio without significantly impacting the speed of compression.

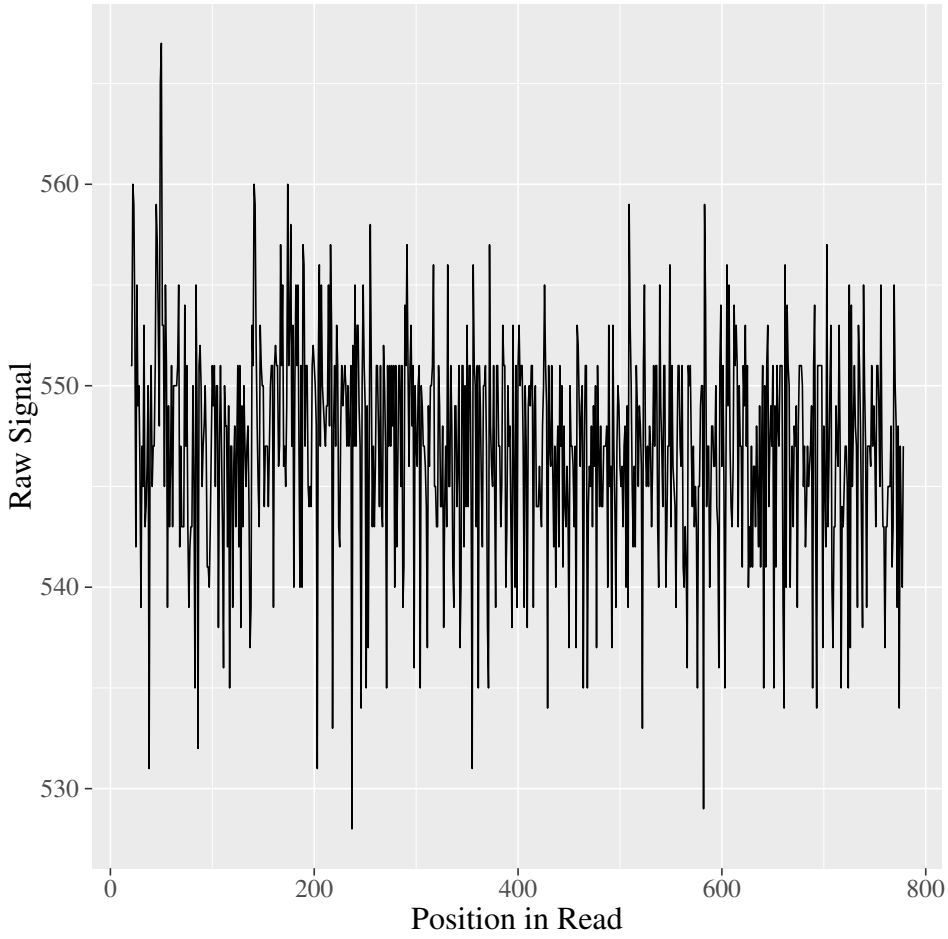


FIGURE 5.9. The stall from the read with ID e9f08690-171f-476f-9119-5330d0290126 spanning data points 20 to 778.

5.2.2 The Stall

We will now explore stall-specific compression strategies. Recall that the stall is the section of a read which occurs at the beginning between the surge and pre-adapter surge. See Figure 3.4 for some context and Figure 5.9 for a close-up of the section. It is thought to occur due to the motor protein 'stalling' before it begins to unwind the molecule through the nanopore. It consists of hundreds to thousands of data points which oscillate with little variation around the read's median. The stall is a highly likely occurrence in any read, for instance, only 6 reads in the data set do not have a stall. However, its length varies from read to read, ranging from 34 to 37 128 with a mean of 1140.33. See Table 5.3 for more information.

TABLE 5.3. Summary statistics of the data's stall lengths.

Min	34
Q1	314
Q2	771
Q3	1156
Max	37128
Mean	1140.33
Mode	39
Sample SD	1214.781

Since this section oscillates with little variance around the read's median, the maximum and minimum are much closer than in the DNA section. Notably, the standard deviation of raw signal values in a stall is 5.72 compared to 35.07 for the whole data set. Instead of computing the zig-zag deltas we could transform the stall by subtract its minimum point from all other stall points (FOR encoding) then apply a suitable compression algorithm such as range coding. The idea is that this could create a distribution with a lower entropy than the stall's zig-zag deltas and hence more potential for compression.

Now we will define this strategy more formally. The *stall* encoding stores the stall's starting index (2 bytes), length (2 bytes), compressed size (2 bytes) and compressed data (variable bytes) followed by the non-stall's compressed size (4 bytes) and compressed data (variable bytes). The stall is compressed using a *stall-specific* encoding, whilst the non-stall is compressed using a *generic* algorithm. See Figure 5.10. This method is more space-advantageous if

$$C_{stall}(r) < C_{generic}(r)$$

where

$$C_{stall}(r) = C_{specific}(r_s) + C_{generic}(r \setminus r_s) + 10.$$

In the above equations, $r \in \Omega$ is a read from the space of possible reads and r_s is the read's stall section. That is, this method is advantageous if its total compressed size, consisting of the compressed stall, the compressed non-stall and 10 bytes of metadata, is less than the usual encoding.

The encoding could dynamically make the above comparison to decide whether the stall is worth encoding or not. Then it could store an extra bit (or byte for convenience) to flag whether the stall or generic algorithm is being used. Let's name this algorithm the *dynamic stall* encoding or *dstall* for short. See Figure 5.11. However, this extra byte is a waste of space if the large majority of reads benefit from separate stall encoding.

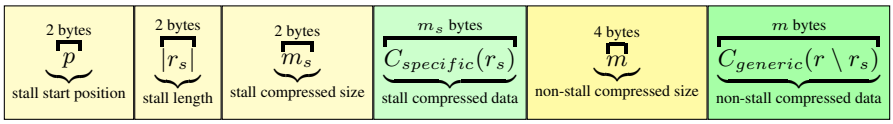


FIGURE 5.10. The stall encoding records the stall's starting position in the read, length, compressed size and compressed data, followed by the non-stall's compressed size and compressed data. The specific and generic compression algorithms used are known beforehand and hence are not stored. stall-fz uses rc01s-vbbe21-for and rc01s-vbbe21-zd as the specific and generic algorithm respectively.

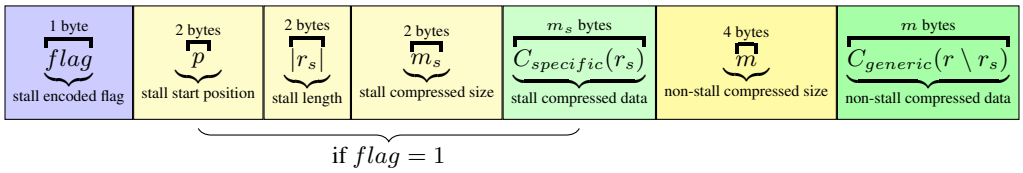


FIGURE 5.11. The dstall encoding stores an extra byte at the beginning to mark whether the stall is being encoded or not. If it is being encoded the remaining data matches the stall encoding. Otherwise, r_s is empty and the whole read is compressed using the generic algorithm as usual. That is, the read's compressed size (4 bytes) and data follows the flag.

Consider our stall-specific encoding to be the FOR encoding followed by the regular vbbe21 encoding (vbbe21) and range coding (altogether *rc01s-vbbe21-for*). Furthermore, let the generic algorithm be the vbbe21-zd encoding followed by range coding (altogether *rc01s-vbbe21-zd*). We shall name the stall encoding which uses the stall-specific encoding rc01s-vbbe21-for and the generic encoding rc01s-vbbe21-zd: *stall-fz*. Similarly, we shall name *dstall-fz* the dstall encoding which uses the same stall-specific and generic encodings as stall-fz.

The compression ratio of stall-fz outperforms the generic method rc01s-vbbe21-zd more often than not on reads with stalls of length greater than or equal to 1500. See Figure 5.12. This means we could simply choose to encode the stall separately when its length is at least 1500. Let's name this strategy *dstall-fz-1500*. Since most stalls are smaller than 1500 in size let's store an extra byte at the beginning of the encoding to flag whether the stall is being encoded or not. If not, the stall specific section is not recorded at all. This is equivalent to the dstall encoding in Figure 5.11 with the exception of how the stall encoding decision is made. The advantage is that the read only needs to be compressed once rather than twice as in the dstall algorithm. This is because dstall must compare compressing the stall separately to not and so must check both ways before proceeding. On the other hand, the dstall-fz-1500 strategy approximates the dstall-fz decision boundary and hence should be faster during compression with no

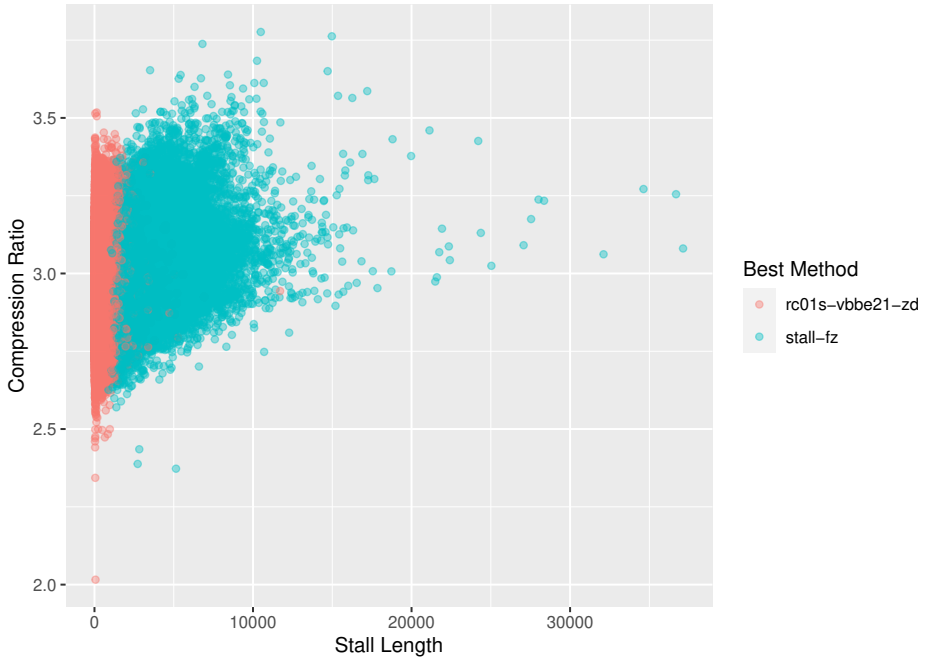


FIGURE 5.12. A scatter plot of the compression methods with the highest compression ratio on each read out of rc01s-vbbe21-zd and stall-fz. The best compression ratio is plotted against the length of the read's stall. Reads with a stall length $\sim$1500 and greater are more likely to be compressed smaller with stall-fz rather than rc01s-vbbe21-zd.

speed difference during decompression. The compression ratio of dstall-fz-1500 will not be better than dstall-fz but there should be little difference if the decision boundary approximation is good.

5.2.3 The DNA Section

The large majority of the data consists of DNA sections so any gains in compression discovered by understanding this type of section should result in the largest compression gains overall. Recall that the DNA sections of nanopore signal data are characterised by many low-variance subsequences of 10–100 data points with steep transitions between them. See Figure 5.13 for an example.

The problem with compressing the DNA sections is that there are a lot of apparent irregularities as is expected with information saturated sensor data. Furthermore, the zig-zag delta transformation is quite fast and produces a lower entropy distribution which is readily compressible making it difficult to outperform.

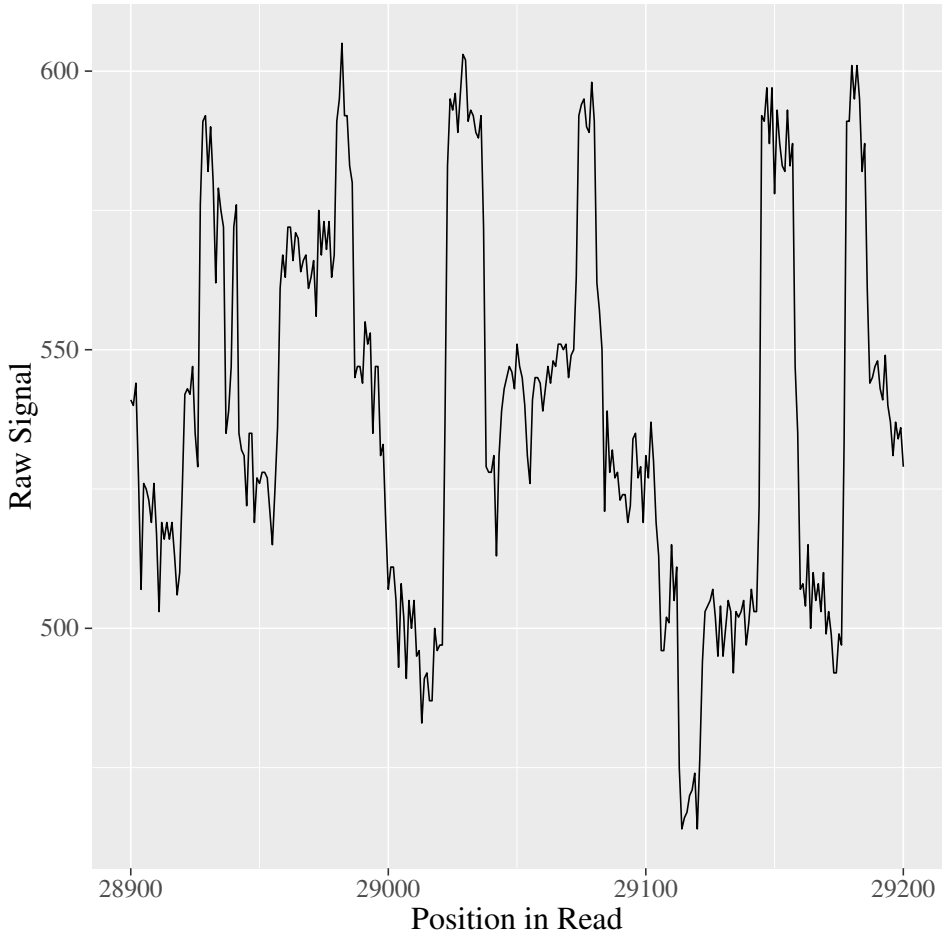


FIGURE 5.13. An example of 300 data points from a DNA section in the read with ID e9f08690-171f-476f-9119-5330d0290126. Notice the sudden transitions up and down (jumps and falls) between low-variance sections (flats).

5.2.3.1 Separating the Jumps, Falls and Flats

However, since we expect many low-variance sections with steep transitions between them we can separate these two types of sections and compress each separately. We will name the low-variance sections *flats* and the steep transitions up and down; *jumps* and *falls* respectively. Since flats oscillate around some level, they should have small zig-zag deltas. On the other hand, jumps and flats should have larger absolute deltas.

For this reason, consider separating the read into two distributions: the zig-zag deltas of flats, and the absolute deltas of jumps and falls. This should benefit compression since each distribution will have its own unique properties which benefit from different compression strategies. However, storing the

metadata necessary for the reconstruction of the read from each distribution may be too high of a cost to incur any space saving.

Let's define a jump or fall as

- a sequence of length $m \geq 2$ which is
- strictly increasing or decreasing respectively ($\forall i\ \delta_i > 0$ or $\forall i\ \delta_i < 0$) with
- at least one absolute delta greater than some ϵ ($\exists i\ s.t.\ |\delta_i| > \epsilon$).

The idea with the third restriction is to actually capture sudden movements rather than slowly increasing or decreasing sections, without which all non-zero deltas would be labelled part of a jump or fall. The choice of which ϵ depends on the data and how much separation between flats and jumps/falls one desires. For example, consider Figure 5.14. Here, each jump and fall is coloured by the maximum epsilon for which it is still considered a jump or fall. In other words, it is coloured by its largest absolute delta ($\max_i\{|\delta_i|\}$). Furthermore, it is labelled by this maximum if it is greater than 20. From visual observation it appears that setting $\epsilon = 24$ may define the jumps and falls as is expected. Figure 5.15 highlights the jumps and falls for $\epsilon = 24$.

The *jumps* encoding separates the two distributions by storing the number of jumps and falls, the starting indices and lengths of the jumps and falls, the absolute deltas of the jumps and falls, and finally the zig-zag deltas of the flats. See Figure 5.16 for more details. Since the deltas are being encoded, the first point in the read must be recorded for invertibility. Furthermore, because the falls have negative deltas, the absolute value of their deltas is recorded. This means the jumps and falls deltas are all positive whilst their distance from zero remains the same – unlike what happens during the zig-zag transformation where the data's distance from zero is roughly doubled.

A caveat with the jumps encoding is that the starting positions of the jumps and flats must be recorded since the interleaving order of the jumps and flats is not maintained. An alternative approach would be to take the zig-zag deltas of the jumps and flats and maintain their order. In which case, recording the starting positions would not be necessary since the order has been maintained in the zig-zag deltas data stream. The length of jumps which are followed by falls and vice-versa could then be represented by a place-holder such as zero since the end of the jump would be marked by a negative delta. This works because a jump/fall or flat will never have a length of zero. However, a new data stream would need to be used to mark when a jump/fall and a flat occurs. Consider a bit stream where 1 is a jump/fall and 0 is a flat. The interesting observation is that since flats will always be preceded and followed by a

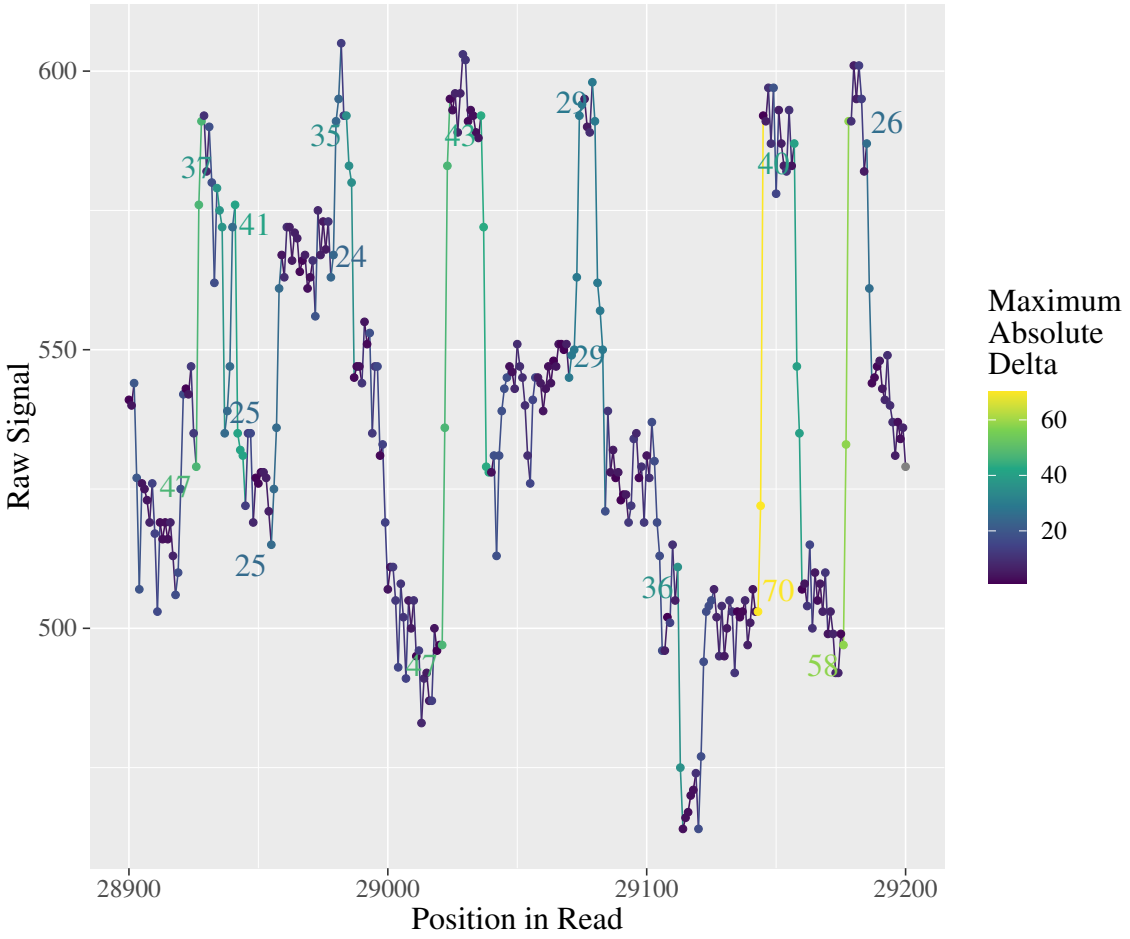


FIGURE 5.14. 300 data points from a DNA section in the read with ID e9f08690-171f-476f-9119-5330d0290126. The strictly increasing and decreasing sequences are coloured by their maximum absolute delta. Those with a maximum absolute delta greater than 20 are labelled by this value.

jump/fall, 0 will always be surrounded by 1s in the bit stream. This gives the potential for a reasonable compression strategy on the bit stream. In addition, the length of each flat would need to be recorded, such as by combining them with the lengths of the jumps and falls. See Figure 5.17 for a representation of the *alternative jumps* encoding. Let's treat this encoding as a theoretical proposal and return back to compressing the regular jumps encoding.

We now consider how to compress each data stream in the regular jumps encoding. The starting indices of the jumps is a strictly increasing array given by

$$p_{j_1} < p_{j_2} < \cdots < p_{j_{n_j}}$$

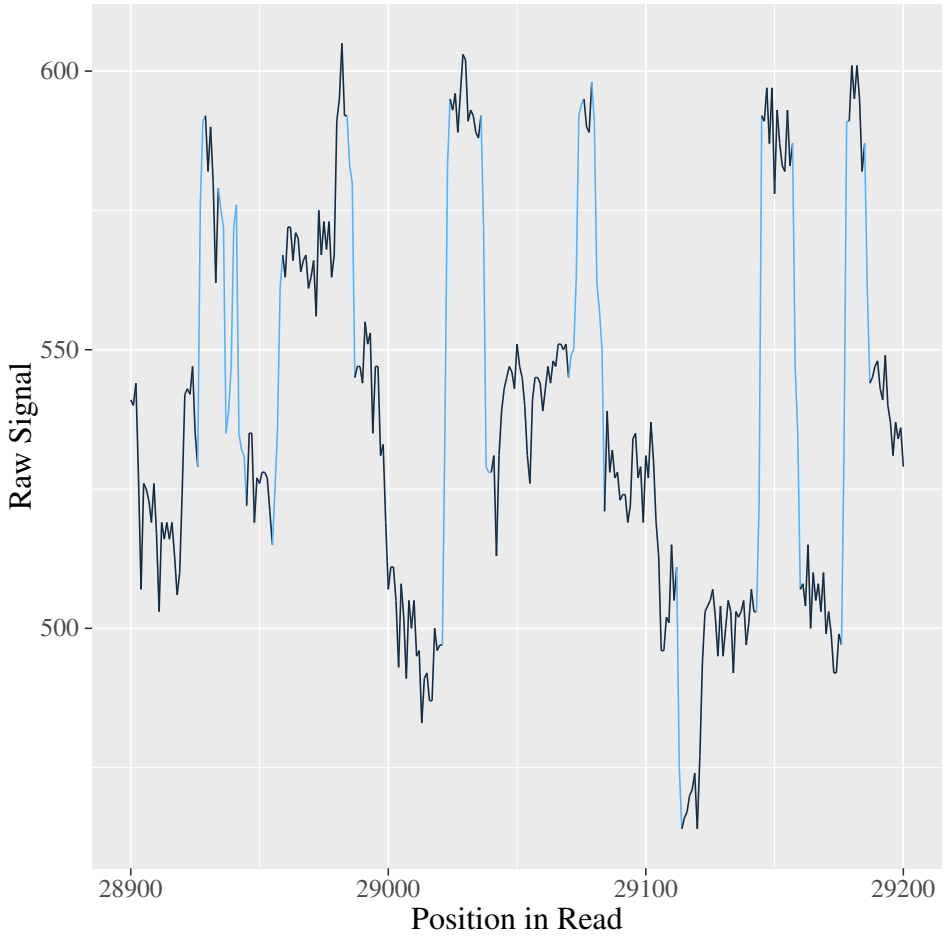


FIGURE 5.15. 300 data points from a DNA section in the read with ID e9f08690-171f-476f-9119-5330d0290126. The jumps and falls are highlighted for $\epsilon = 24$.

For this reason, one idea is to take the deltas between each adjacent starting index and minus one since no two jumps start at the same position. Using this transformation we obtain the following

$$p_{j_1}, p_{j_2} - p_{j_1} - 1, \ldots, p_{j_{n_j}} - p_{j_{n_j-1}} - 1.$$

This represents the distance between the start of adjacent jumps minus one. We could actually decrease this delta further by instead finding the distance between the end of one jump and the start of the next. This would require the use of the length data and would look like

$$p_{j_1}, p_{j_2} - p_{j_1} - |j_1|, \ldots, p_{j_{n_j}} - p_{j_{n_j-1}} - |j_{n_j-1}|.$$

We could then apply rc01s-vbbe21. This strategy works similarly for the starting indices of the falls. For the lengths of the jumps and falls we can simply minus one from each length and then apply range

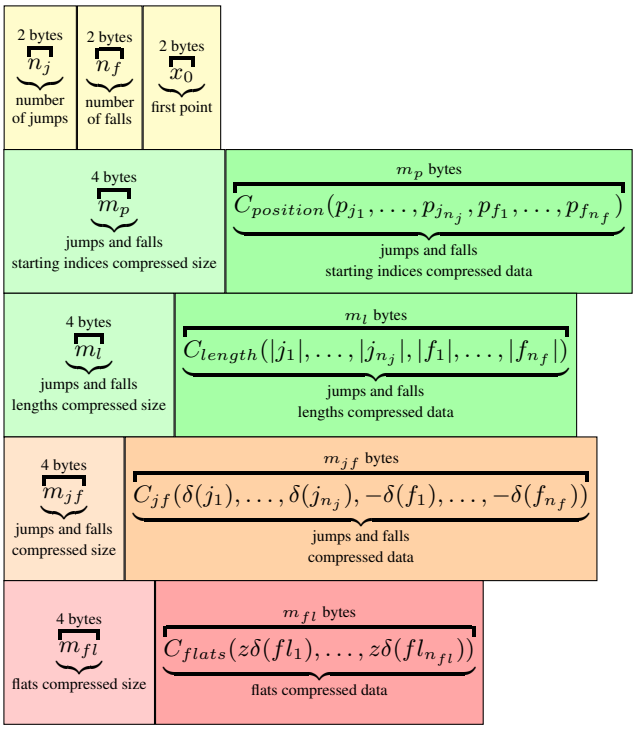


FIGURE 5.16. The jumps encoding stores the number of jumps and falls, and the first data point in the read since the deltas are being encoded. The starting indices of the jumps and falls are compressed then their lengths are compressed. Next, the deltas of the jumps and the negative deltas of the falls are compressed together, followed by the zig-zag deltas of the flats. There are opportunities for four different compression techniques given the different data streams.

coding without vbbe21 since the lengths of the jumps and falls are not expected to exceed 255. Similarly, we can subtract one from the deltas of the jumps and falls since no jump/fall can have a delta of zero and then apply rc01s-vbbe21. Finally, the flats' absolute deltas are less than or equal to ϵ so their zig-zag deltas range from 0 to 2ϵ meaning we can simply apply range coding without two byte exception handling if $\epsilon \leq 127$.

Overall, this jumps compression strategy has $O(n)$ time complexity for compression and decompression but in practice it is time consuming since the positions and length streams add an additional $2(n_j+n_f) = O(n)$ data points. Also an extra pass is required to determine which sequences are jumps/falls and during decompression there are many transformations required to recombine the jumps/falls with the flats.

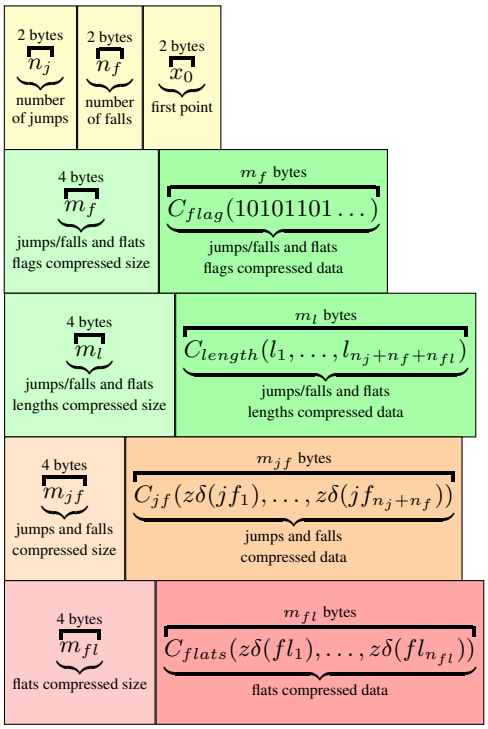


FIGURE 5.17. The alternative jumps encoding which differentiates jumps from falls by recording their zig-zag deltas rather than absolute deltas. The jumps/falls and flats are differentiated from each other by a flag bit stream where 1 represents a jump/fall and 0 a flat. The lengths of the jumps/falls and flats are interleaved.

CHAPTER 6

Results

To evaluate the performance of the existing and novel compression strategies, each read from the data was compressed and decompressed sequentially using each method (within reason[1]). To ensure every method fit the suitability requirement of lossless compression, the decompressed data was tested for equality against the original uncompressed data. For each read the following metrics were calculated:

- the read's uncompressed and compressed size (in bytes),
- the time taken to compress the read (in seconds) and
- the time taken to decompress it (in seconds)[2].

Then, for each method the total sum of the per-read metrics over the whole data set was calculated and recorded.

The optimal subsequence searching algorithms were found to be too time consuming in practice and the approximating subsequence algorithm did not achieve a competitive compression ratio. Furthermore, the jumps encoding did not perform well either. It was found that setting ϵ to larger values, such that fewer jumps/falls were defined, caused the compression ratio to increase and approach that the flats

[1] Some novel methods proved too time consuming or didn't achieve a high enough compression ratio to warrant implementing decompression.

[2] The time was calculated using the `clock()` (Kerrisk, 2021) C system call.

TABLE 6.1. Specifications of the server used for the experiments.

Description	Dell PowerEdge C4140 Server Rack
CPUs	2× Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz
CPU cores	2×10
CPU threads	2×20
RAM (GiB)	376
Disk System	6.4 TiB NVMe drive
File System	ext4
OS	Ubuntu 18.04.5 LTS

encoding. That is, separating the signal as defined in Section 5.2.3.1 only decreased the compression ratio. Wavelet-based methods were also briefly explored, but they were found to be too space consuming to maintain the requirement of lossless compression.

The experiments were executed on a rack-mounted server with the typical specifications of a small high performance computing server. The server's specifications are displayed in Table 6.1. Despite the potential to compress and decompress the reads in parallel using multiple threads, each read was compressed and decompressed sequentially. For this reason, consider the time results as highly scalable.

The space results from these experiments are shown in Table 6.2. For each method, the compression ratio, space saving, number of bits used per symbol on average and total compressed size (in GiB) are displayed. Recall that the compression ratio and space saving are defined as

$$\text{Compression Ratio} = \frac{\text{Uncompressed Size}}{\text{Compressed Size}}$$

and

$$\text{Space Saving} = 1 - \frac{\text{Compressed Size}}{\text{Uncompressed Size}}.$$

The compression ratio can be interpreted as the factor by which the uncompressed size is decreased as a result of compression. Whilst the space saving is interpreted as the proportion of the uncompressed data which is now available (or saved) as a result of compression.

The time results are then shown in Tables 6.3 and 6.4 which are sorted (fastest to slowest) by the compression and decompression time respectively. The time is displayed in the units hours per TiB such that the results are independent of the data's size. The compression ratio is also shown as a means of comparison with the space results.

For all the Tables 6.2, 6.3 and 6.4, the state-of-the-art method zstd-svb-zd is highlighted in light grey and the methods which achieved more space saving than zstd-svb-zd are highlighted in a darker grey.

For ease of comparison, the space results are plotted using bar plots in Figures 6.1, 6.2 and 6.3. In all these Figures, the dark gray bar represents the state-of-the-art method and the solid and dotted vertical lines represents the entropy of the data and its deltas respectively. Figures 6.1 and 6.2 show the compression ratio and space saving respectively for all methods. Whilst Figure 6.3 shows the bits used per symbol on average and the total compressed size (in Gib) for each method using a dual x-axis.

In order to compare the effect of the vbe21 and vbbe21 encodings, a two-way table of the space savings of the methods constructed by applying one encoding from the first layer (svb-zd, svb16-zd, vbe21-zd and vbbe21-zd) followed by another from the second layer (none, zlib, zstd and rc01s) is displayed in Table 6.5. The constructed method with the highest space saving for each method in the second layer is highlighted in grey and the method with the highest space saving overall is highlighted in a darker grey. The 'none' compression method in the second layer means that no compression was applied in that step.

The compression and decompression time (in hours per TiB) is plotted against the space saving for all methods in Figures 6.4a and 6.4c respectively. Then, for better clarity, enlarged and simplified plots of the same data are shown in Figures 6.4b and 6.4d respectively. These enlarged Figures depict the methods which save as or more space than the state-of-the-art and also exist on the space–(de)compression-time frontier. For each method on the space–(de)compression-time frontier there is no other method which has a greater space saving and (de)compresses in less time. These methods are labelled in all sub-plots of Figure 6.4 and coloured in red is the state-of-the-art method.

Finally, in Figure 6.5 the compression time is plotted against the decompression time (in hours per TiB) for all methods which save as or more space than the state-of-the-art method. The scatter plot is coloured by this space saving value and all the methods are labelled with there respective point.

All the above results will now be discussed in detail in the following chapter.

TABLE 6.2. The compression ratio, bits used per symbol and compressed size (in GiB) of the data set after compressing each read sequentially using each of the given methods. It is sorted by compression ratio (lowest to highest) and the state-of-the-art method is highlighted in grey. The methods which have a greater compression ratio than the state-of-the-art are highlighted in a lighter grey. The two horizontal lines represent the entropy of the data (7.70 bits per symbol) and the entropy of the deltas (5.39 bits per symbol). Hence, methods below the first horizontal line have fewer bits per symbol than the entropy of the data and so are suitable. A hyphen ('-') between methods means that they are applied to the original data in layers from right to left.

Method	Compression Ratio	Space Saving	Bits Per Symbol	Compressed Size (GiB)
none	1.000000	0.00000000	16.000000	105.67848
svb-zd	1.599930	0.37497255	10.000527	66.05195
svb0-zd	1.682548	0.40566348	9.509468	62.80858
svb16-zd	1.777690	0.43747228	9.000523	59.44707
zstd	1.790916	0.44162666	8.934052	59.00804
vbe21-zd	1.999519	0.49987982	8.001993	52.85194
vbbe21-zd	1.999714	0.49992849	8.001215	52.84680
zlib	2.001465	0.50036604	7.994214	52.80056
zlib-svb0-zd	2.697205	0.62924589	5.932118	39.18073
bzip2-svb16-zd	2.742621	0.63538529	5.833887	38.53193
bzip2	2.750089	0.63637539	5.818045	38.42729
zlib-svb-zd	2.783474	0.64073678	5.748262	37.96639
zlib-svb16-zd	2.786146	0.64108121	5.742751	37.92999
zstd-svb0-zd	2.789808	0.64155240	5.735212	37.88020
zlib-vbe21-zd	2.790276	0.64161254	5.734250	37.87384
zlib-vbbe21-zd	2.790488	0.64163978	5.733814	37.87096
flac	2.893409	0.65438689	5.529859	36.52387
huff-vbe21-zd	2.927298	0.65838802	5.465840	36.10103
huff-vbbe21-zd	2.927709	0.65843599	5.465072	36.09596
zstd-vbe21-zd	2.928103	0.65848199	5.464336	36.09110
zstd-svb16-zd	2.928344	0.65851007	5.463887	36.08814
zstd-vbbe21-zd	2.928413	0.65851816	5.463758	36.08728
zstd-svb-zd	2.928430	0.65852009	5.463727	36.08708
rc0-vbe21-zd	2.930661	0.65878001	5.459568	36.05961
rc0-vbbe21-zd	2.931079	0.65882867	5.458789	36.05447
rc1-vbe21-zd	2.947403	0.66071828	5.428555	35.85477
shuff-vbe21-zd	2.947726	0.66075550	5.427960	35.85084
rc1-vbbe21-zd	2.947826	0.66076694	5.427777	35.84963
shuff-vbbe21-zd	2.948147	0.66080385	5.427186	35.84573
rc01s-svb-zd	2.990472	0.66560461	5.350373	35.33840
rc01s-svb16-zd	2.990579	0.66561660	5.350182	35.33713
rc01s-vbe21-zd	2.990877	0.66564996	5.349648	35.33360
stall-fz	2.991124	0.66567752	5.349207	35.33069
rc01s-vbbe21-zd	2.991313	0.66569862	5.348869	35.32846
dstall-fz-1500	2.991704	0.66574236	5.348169	35.32384
dstall-fz	2.991729	0.66574516	5.348124	35.32354

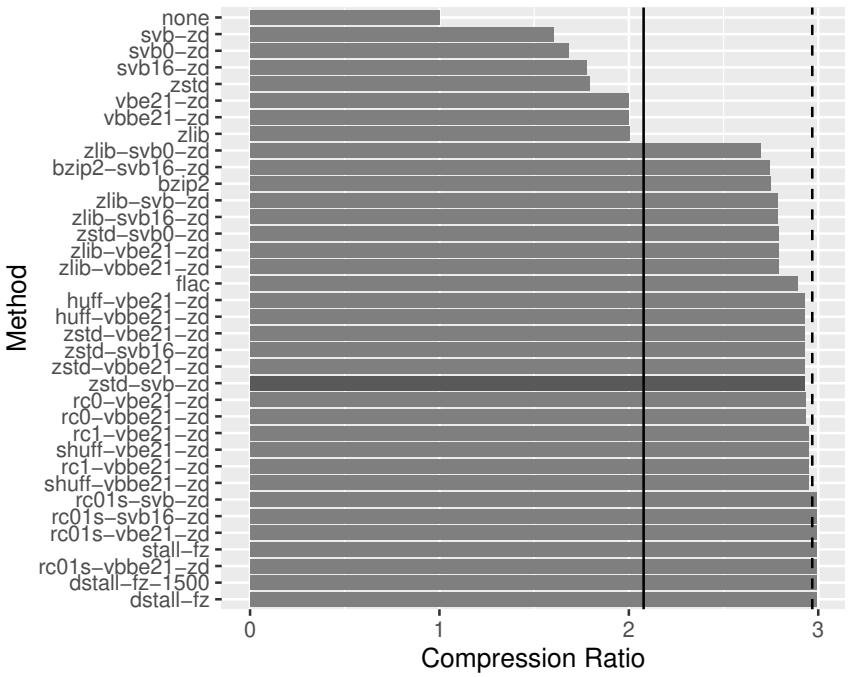


FIGURE 6.1. The compression ratio of each method on the data. The state-of-the-art method is highlighted in a darker grey. The solid and dotted vertical lines represents the compression ratio equivalent of the entropy of the data and its deltas respectively.

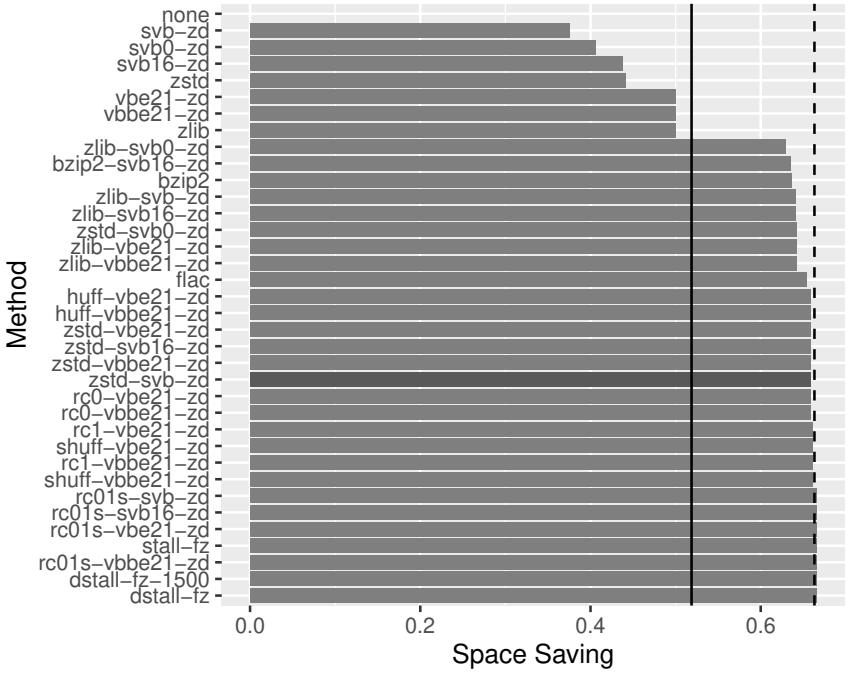


FIGURE 6.2. The space saving of each method on the data. The state-of-the-art method is highlighted in a darker grey. The solid and dotted vertical lines represents the space saving equivalent of the entropy of the data and its deltas respectively.

TABLE 6.3. The total compression and decompression time in hours per TiB and the compression ratio after compressing each read from the data sequentially using each of the given methods. It is sorted by the compression time (fastest to slowest) and the state-of-the-art method is highlighted in grey. The methods which have a greater compression ratio than the state-of-the-art are highlighted in a lighter grey.

Method	Compression Time (hr / TiB)	Decompression Time (hr / TiB)	Compression Ratio
none	0.00000000	0.00000000	1.000000
svb-zd	0.73883975	0.73005926	1.599930
svb16-zd	0.78877395	0.95850039	1.777690
svb0-zd	0.83445915	0.75598838	1.682548
zstd-svb-zd	1.04458386	1.02763119	2.928430
zstd-svb16-zd	1.08681230	1.24932700	2.928344
zstd-svb0-zd	1.34129640	1.12928189	2.789808
vbe21-zd	1.36213730	1.11785785	1.999519
zstd-vbe21-zd	1.65240171	1.40322205	2.928103
vbbe21-zd	1.84884704	1.53871491	1.999714
zstd	2.05226509	0.70892108	1.790916
zstd-vbbe21-zd	2.24252232	1.94868791	2.928413
flac	4.15288902	1.21167452	2.893409
rc0-vbbe21-zd	5.76588458	7.16783267	2.931079
rc1-vbe21-zd	5.81153924	6.64682951	2.947403
rc0-vbe21-zd	5.82912774	7.51768231	2.930661
rc1-vbbe21-zd	5.95591447	6.85270715	2.947826
zlib-svb16-zd	8.12190118	2.15773132	2.786146
zlib-svb-zd	8.22530063	1.96858373	2.783474
zlib-vbe21-zd	8.48832267	2.28195836	2.790276
shuff-vbe21-zd	8.98396994	10.10209679	2.947726
shuff-vbbe21-zd	9.07391018	10.03256857	2.948147
zlib-vbbe21-zd	10.85605856	2.98523591	2.790488
zlib-svb0-zd	11.23940746	2.00888074	2.697205
huff-vbe21-zd	11.33375058	10.01709248	2.927298
huff-vbbe21-zd	11.99385775	10.55591243	2.927709
rc01s-vbbe21-zd	15.68806122	17.87252018	2.991313
rc01s-svb16-zd	15.76907766	15.99817668	2.990579
rc01s-vbe21-zd	15.82961402	18.28172897	2.990877
bzip2-svb16-zd	16.00204323	10.66535592	2.742621
dstall-fz-1500	16.19271379	16.83622670	2.991704
stall-fz	16.27122516	16.89594957	2.991124
rc01s-svb-zd	17.96375449	15.56391103	2.990472
bzip2	25.73918708	12.66451322	2.750089
zlib	28.39558473	1.54142078	2.001465
dstall-fz	30.99850195	16.98327384	2.991729

TABLE 6.4. The total compression and decompression time in hours per TiB and the compression ratio after compressing each read from the data sequentially using each of the given methods. The data is equivalent to Table 6.3 but it is sorted by the decompression time (fastest to slowest). The state-of-the-art method is highlighted in grey and the methods which have a greater compression ratio than the state-of-the-art are highlighted in a lighter grey.

Method	Compression Time (hr / TiB)	Decompression Time (hr / TiB)	Compression Ratio
none	0.00000000	0.00000000	1.000000
zstd	2.05226509	0.70892108	1.790916
svb-zd	0.73883975	0.73005926	1.599930
svb0-zd	0.83445915	0.75598838	1.682548
svb16-zd	0.78877395	0.95850039	1.777690
zstd-svb-zd	1.04458386	1.02763119	2.928430
vbe21-zd	1.36213730	1.11785785	1.999519
zstd-svb0-zd	1.34129640	1.12928189	2.789808
flac	4.15288902	1.21167452	2.893409
zstd-svb16-zd	1.08681230	1.24932700	2.928344
zstd-vbe21-zd	1.65240171	1.40322205	2.928103
vbbe21-zd	1.84884704	1.53871491	1.999714
zlib	28.39558473	1.54142078	2.001465
zstd-vbbe21-zd	2.24252232	1.94868791	2.928413
zlib-svb-zd	8.22530063	1.96858373	2.783474
zlib-svb0-zd	11.23940746	2.00888074	2.697205
zlib-svb16-zd	8.12190118	2.15773132	2.786146
zlib-vbe21-zd	8.48832267	2.28195836	2.790276
zlib-vbbe21-zd	10.85605856	2.98523591	2.790488
rc1-vbe21-zd	5.81153924	6.64682951	2.947403
rc1-vbbe21-zd	5.95591447	6.85270715	2.947826
rc0-vbbe21-zd	5.76588458	7.16783267	2.931079
rc0-vbe21-zd	5.82912774	7.51768231	2.930661
huff-vbe21-zd	11.33375058	10.01709248	2.927298
shuff-vbbe21-zd	9.07391018	10.03256857	2.948147
shuff-vbe21-zd	8.98396994	10.10209679	2.947726
huff-vbbe21-zd	11.99385775	10.55591243	2.927709
bzip2-svb16-zd	16.00204323	10.66535592	2.742621
bzip2	25.73918708	12.66451322	2.750089
rc01s-svb-zd	17.96375449	15.56391103	2.990472
rc01s-svb16-zd	15.76907766	15.99817668	2.990579
dstall-fz-1500	16.19271379	16.83622670	2.991704
stall-fz	16.27122516	16.89594957	2.991124
dstall-fz	30.99850195	16.98327384	2.991729
rc01s-vbbe21-zd	15.68806122	17.87252018	2.991313
rc01s-vbe21-zd	15.82961402	18.28172897	2.990877

TABLE 6.5. A two-way table of the space saving after applying a compression method in the first layer followed by one in the second layer. The method with the highest space saving for each second layer is highlighted in grey, and the method with the highest space saving overall is highlighted in a darker grey.

	First Layer			
Second Layer	svb-zd	svb16-zd	vbe21-zd	vbbe21-zd
none	0.37497255	0.43747228	0.49987982	0.49992849
zlib	0.64073678	0.64108121	0.64161254	0.64163978
zstd	0.65852009	0.65851007	0.65848199	0.65851816
rc01s	0.66560461	0.66561660	0.66564996	0.66569862

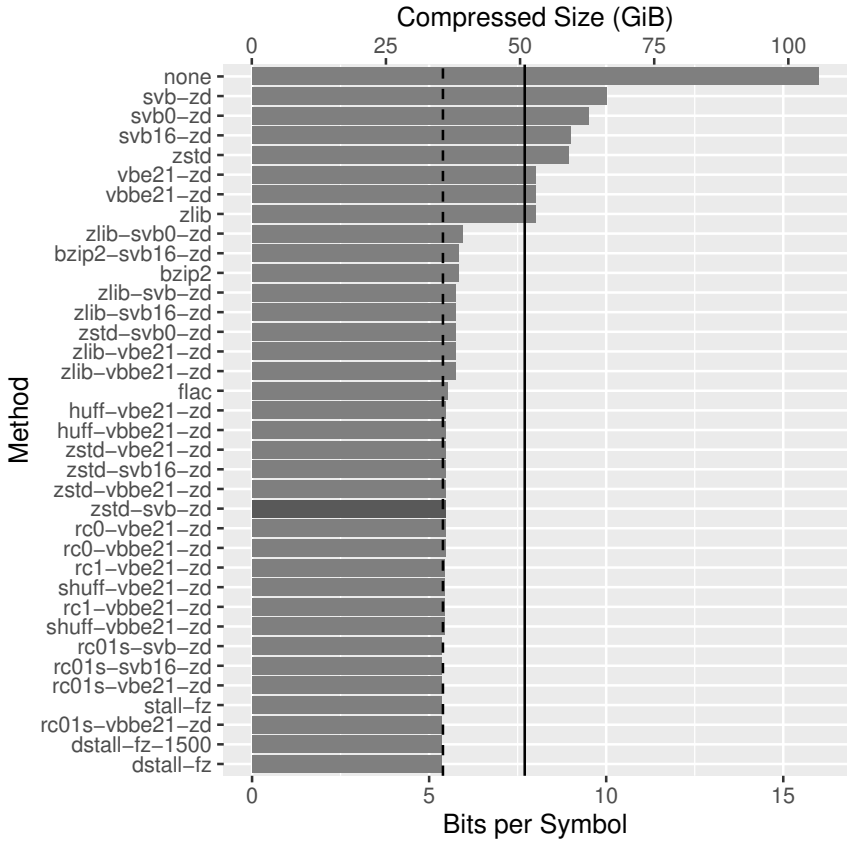


FIGURE 6.3. The average number of bits used per symbol and total compressed size of each method on the data. The state-of-the-art method is highlighted in a darker grey. The solid and dotted vertical lines represents the bits per symbol equivalent of the entropy of the data and its deltas respectively.

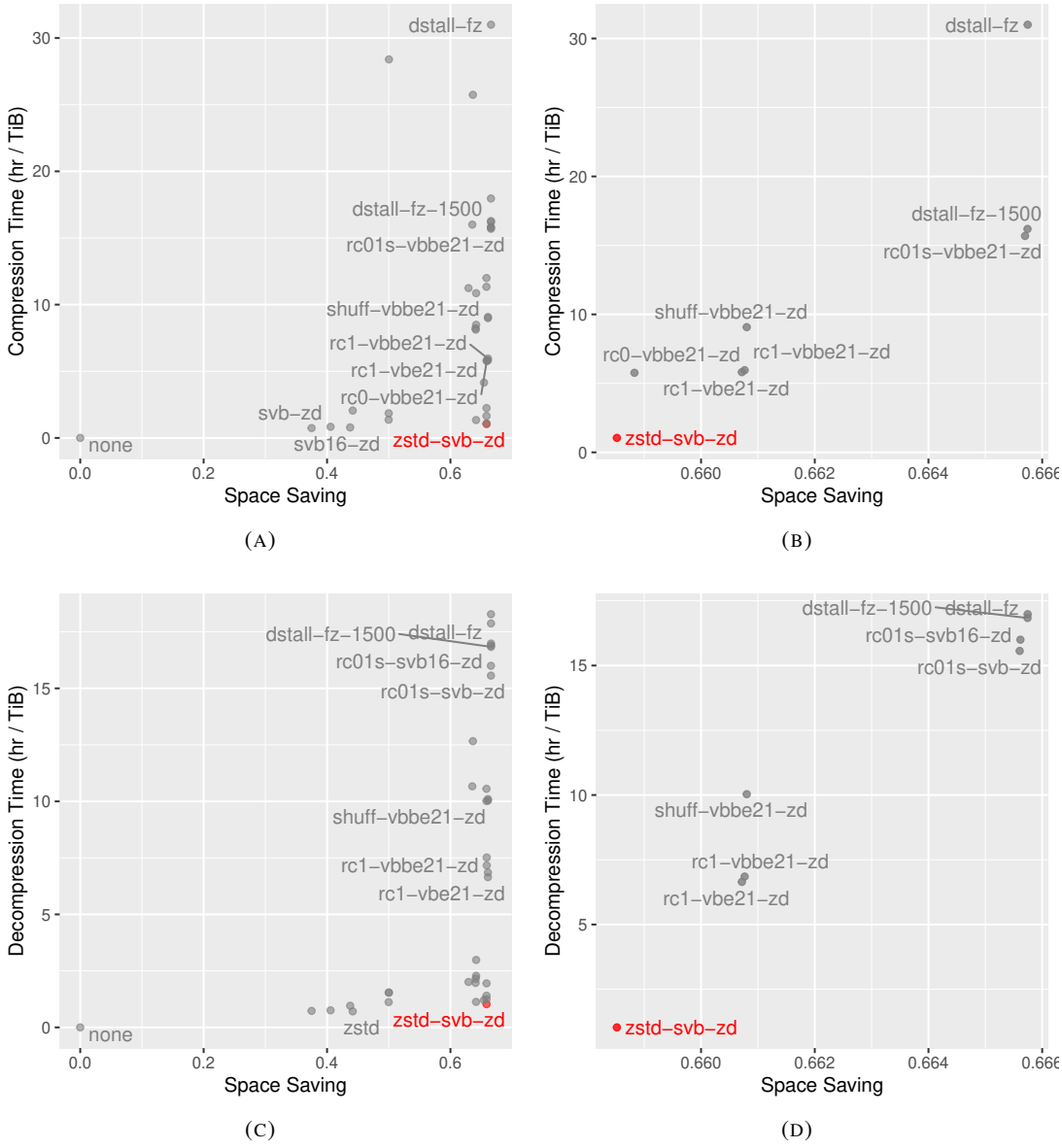


FIGURE 6.4. The (de)compression time (in hours per TiB) versus space saving of various methods. The state-of-the-art method is coloured in red and the labelled methods are on the space–(de)compression-time frontier. That is, for each labelled method in Figures 6.4a and 6.4b there is no other compression method which produces a greater space saving in less time. Whilst for each labelled method in Figures 6.4c and 6.4d there is no other compression method which has a greater space saving and decompresses in less time. Figures 6.4a and 6.4c show all the methods. Whilst Figures 6.4b and 6.4d show the methods which are on their respective froniter and have a space saving greater than or equal to the state-of-the-art.

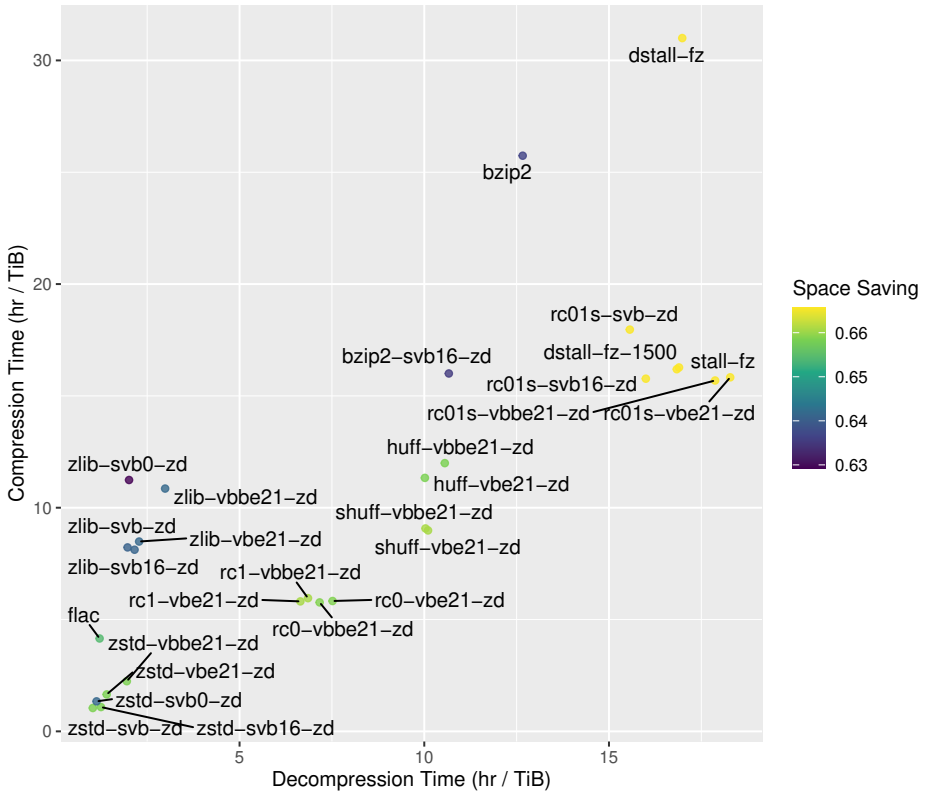


FIGURE 6.5. A scatter plot of the methods which have a space saving greater than or equal to the state-of-the-art. Compression time is plotted against decompression time (in hours per TiB) and each point is coloured by its space saving. The state-of-the-art method zstd-svb-zd is in the bottom-left corner.

CHAPTER 7

Discussion

In this chapter, we provide a final discussion of the results obtained thus far. Firstly, we evaluate the results presented in Chapter 6 as well as those found through side-experimentation (Section 7.1). We conclude with a discussion of open problems in the domain and future work to be explored (Section 7.2).

7.1 Experiments

Overall, we found that range coding (rc0, rc1 and rc01s), static huffman (shuff) and stall encoding outperformed the state-of-the-art method (zstd-svb-zd) in terms of overall space saving. The order-0 order-1 SSE range coder (rc01s) exceeded the entropy of the zig-zag deltas (5.39 bits per symbol) with the best method rc01s-vbbe21-zd achieving 5.35 bits per symbol. The dynamic stall encodings (dstall-fz-1500 and dstall-fz) narrowly performed better than this by 0.0007 bits and 0.00075 bits per symbol respectively. rc01s and stall-based methods saved between 0.708% and 0.722% more space than the state-of-the-art. This is equates to roughly 7.3 GiB per TiB which isn't much except when you consider the amount of data being archived in practice. For every 140 TiB an extra TiB is saved compared to the state-of-the-art. This is an estimated saving of $283 a year per 140 TiB when using Standard Google Cloud data storage in Sydney. For data which is archived for up to 10 years, this results in a significant saving of almost $3000 per 140 TiB.

Let's now explore the effect of the regular and compact one byte, two byte exceptions encodings (vbe21 and vbbe21) versus cannonical and 16-bit Stream VByte on compression. Table 6.5 reveals that for range coding (rc01s) the exceptions encodings produce a greater space saving than Stream VByte and, in particular, vbbe21 results in the greatest space saving overall. This is most likely because range coding is not only dependent on the data's distribution but also on the input size. To support this claim notice that the first and fourth rows in Table 6.5 share the same ordering of space saving. That is, the order of space saving before compression (worst to best: svb-zd, svb16-zd, vbe21-zd and vbbe21-zd)

remains unchanged after compressing with range coding. zlib also observes the same pattern, most likely because it primarily uses Huffman coding. However, zstd favours svb-zd followed by vbbe21-zd, svb16-zd and vbe21-zd. This likely related to the composition of svb-zd which will consist of many `00` words followed by (mostly) one byte data. The repetition of `0` in the control byte header will be more compressible for zstd since it heavily relies on the dictionary-based encoding LZ4. It is clear that vbbe21 has a positive impact on space saving, not only alone but also for downstream compression methods.

The fastest method to compress the data beyond its entropy was the state-of-the-art method zstd-svb-zd at roughly 1 hour per TiB. The same is true for decompression with zstd-svb-zd taking roughly the same amount time. The next fastest method was flac at roughly 4 hours per TiB during compression and 1.2 hours per TiB during decompression. However, its compression ratio is worse than zstd-svb-zd meaning that it does not lie on the space-time frontiers depicted in Figure 6.4 and so is less successful according to all metrics. The next fastest family of methods were the range coding order-0 and order-1 (rc0 and rc1) based methods; visible as a cluster of four in Figure 6.5. These methods took around 6 times as long as the state-of-the-art during compression and decompression but incured an added space saving of roughly 0.2% or 2 GiB per TiB. rc1-vbbe21-zd produced the most space saving for this family at around 0.661. This implies that the order-1 model is more successful than the order-0 model at predicting the zigzag-deltas. The order-1 models also took on average half an hour quicker per TiB than the order-0 models during decompression and roughly the same time during compression.

The range coding family modelled by order-0 order-1 SSE (rc01s) including the stall encodings all gave similar space savings and time costs – with the exception of dstall-fz which took twice as long during compression. The space saving of this family is roughly 0.666 at a cost of 16–17 hours per TiB during compression and decompression. This equates to 0.7% more space saved than the state-of-the-art but at a factor of 16–17 times slower. The SSE range coding family is much slower than the simple order-n models since it must maintain an order-0 and order-1 model as well as process their output whilst operating at the bit rather than byte level. However, by combining both models it is the only family to overcome the entropy of the zig-zag deltas meaning that it understands the data more intimately than a simple distribution-based model.

Huffman coding performed somewhere between that of the two range coding families. Notably, the static Huffman (shuff) method outperformed the regular Huffman (huff) method for both space and time metrics. As expected, this means that the Huffman table consumed more space than the difference between the globally approximated and read-optimal Huffman encodings. shuff-vbbe21-zd was the best

performing method in this family, with a space saving of 0.661, compression time of 9 hours per TiB and decompression time of 10 hours per TiB. However, this space saving is only marginally better than rc1-vbbe21-zd whilst it took roughly 1.5 times longer during both compression and decompression. In theory, both methods should have the same asymptotic running time, but in practice, the efficiency of different implementations becomes a driving factor which influences the final time results.

Recall from Chapter 4 that we can compare two suitable compression methods by calculating

$$\Delta(M_1, M_2) := \delta_2 - \delta_1 - \alpha(t_2^c - t_1^c) - \beta(t_2^d - t_1^d).$$

If $\Delta(M_1, M_2) > 0$ then M_2 is more suitable than M_1. Moreover, recall that we derived that $(\alpha, \beta) = (0.01, 0.03)$ for the analysis subspace and $(\alpha, \beta) = (\frac{1}{960}, \frac{1}{192})$ for the archival subspace. Using this equation we can compare the suitability of the novel methods to that of the state-of-the-art for the analysis and archival subspace. In particular, for an increase in space saving by 0.007 – which is what the best method achieves – the compression and decompression time must satisfy

$$t^c + 3t^d - 4.7 < 0$$

to be more suitable for the analysis subspace and satisfy

$$t^c + 5t^d - 12.72 < 0$$

to be more suitable for the archival subspace. None of the novel methods with a space saving around 0.007 satisfy these equations. The problem is that the state-of-the-art method is highly optimised whilst the novel methods have been simply implemented to work. As a result, for now these novel methods are perhaps unsuitable for analysis and archival unless time is not an issue. However, future work dedicated to optimisation and multithreading could easily increase the suitability of these methods.

7.2 Future Work

There is a lot of room for future work now that increases in space saving have been discovered. In particular, integrating the novel methods into slow5lib which has a pre-existing multithreading framework would easily increase the suitability of these methods. Also, better implementations of each method could help decrease their time costs.

Beyond implementation specifics, there are many ideas yet to explore. For instance, the metadata of each read such as its length, pore and channel could be exploited during compression. This hints at the idea of a multi-read compression strategy which may be great for archival purposes where decompression is infrequent. One multi-read strategy could exploit the fact that reads generated by the same channel and pore will have been recorded by the same electrode and so will likely have similar properties. This is especially true for reads which were generated at similar timestamps as each nanopore deteriorates over the course of its sequencing lifetime.

We have exploited the stall in this thesis but homopolymer and slow sections exhibit similar characteristics to the stall and hence could also be separately compressed. Furthermore, there may be more successful transformations of the stall which increase space saving than the current strategy which combines FOR and range coding.

There are also other nanopore data sets to analyse and understand. Non-human data sets may present other challenges or advantages. The same is true for RNA and non-PromethION data.

CHAPTER 8

Conclusion

Nanopore sequencing is a next-generation approach to DNA sequencing known for its speed, portability, long reads and real-time analysis. Its increasing popularity has been further influenced by the advent of the COVID-19 pandemic which resulted in approximately one quarter of all SARS-CoV-2 virus genomes being sequenced by nanopore devices (Marks, 2021). This popularity has caused a dramatic increase in data storage costs pertaining to the magnitude of the data which is around 1 TB per human DNA sequencing run. The current approach to compressing this data does not consider its unique characteristics and thus misses crucial space saving opportunities.

As a result, we set out to discover lossless compression methods for archiving nanopore signal data which achieve more space saving that the current state-of-the-art. Data compression is fundamentally an artificial intelligence problem of understanding the data and making the correct predictions (Mahoney, 2013). Hence, after providing the background material for the thesis, we conducted a systematic analysis of the signal data including an examination of its characteristics and suitable transformations. Then, we explored several strategies which exploit the signal data's characteristics including the vbbe21-zd encoding, static Huffman, optimal subsequence searching, stall-specific encoding and separating the jumps and falls from the flats. After setting up the first comprehensive benchmark of existing and novel compression methods for nanopore signal data, a large experiment was conducted wherein each read from the downsampled NA12878 data set was compressed and decompressed sequentially using each method. Space and time metrics were collected and a superior space saving of 0.666 versus 0.659 – which outperforms the entropy of the deltas – was achieved using the dstall-fz method. This strategy combines vbbe21 with dynamic stall encoding and range coding modelled by order-0 order-1 SSE. Furthermore, an alternative dynamic stall encoder, dstall-fz-1500, was found to approximate the decision boundary of dstall-fz whilst achieving the same space saving of 0.666 in half the compression time.

Appendix

8.1 Subsequence Searching

Using the arithmetico-geometric series formula

$$S_n = \frac{A_1G_1 - A_{n+1}G_{n+1}}{1-r} + \frac{dr}{(1-r)^2}(G_1 - G_{n+1})$$

$$\begin{aligned}
c_n &= O(n) + \sum_{k=1}^{n-1} 3^{k-1}O(n-k) \\
&= O(n) + \frac{O(n-1)-0}{1-3} - \frac{3}{(1-3)^2}(1-3^{n-1}) \\
&= O(n) - \frac{O(n-1)}{2} + \frac{3}{4}(3^{n-1}-1) \\
&= O(n) - \frac{O(n-1)}{2} + \frac{3^n}{4} - \frac{3}{4} \\
&= O(3^n)
\end{aligned}$$

$$\begin{aligned}
\tilde{c}_n &= \sum_{k=1}^{n}(n-k+1)O(k) \\
&= nO(1) + (n-1)O(2) + \cdots + 2O(n-1) + O(n) \\
&= \frac{1}{6}n(n+1)(n+2) \\
&= O(n^3)
\end{aligned}$$

Bibliography

Michael Burrows and David J. Wheeler. 1994. A block-sorting lossless data compression algorithm. Digital Equipment Corporation.

Shubham Chandak, Kedar Tatwawadi, Srivatsan Sridhar, and Tsachy Weissman. 2020a. Impact of lossy compression of nanopore raw signal data on basecalling and consensus accuracy. *Bioinformatics*, 36(22-23):5313–5321.

Shubham Chandak, Kedar Tatwawadi, Chengtao Wen, Lingyun Wang, Juan Aparicio Ojea, and Tsachy Weissman. 2020b. Lfzip: Lossy compression of multivariate floating-point time series data via improved prediction. In *2020 Data Compression Conference Proceedings (DCC)*, volume 2020-March, pages 342–351.

Yann Collet and Murray Kucherawy. 2018. Zstandard Compression and the application/zstd Media Type. RFC 8478.

Thomas M. Cover and Joy A. Thomas. 1991. *Elements of Information Theory*. Wiley series in telecommunications. John Wiley & Sons.

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

David Deamer, Mark Akeson, and Daniel Branton. 2016. Three decades of nanopore sequencing. *Nature Biotechnology*.

International Organization for Standardization. 2004. Iso/iec 15948:2004 information technology – computer graphics and image processing – portable network graphics (png): Functional specification. `https://www.iso.org/standard/29581.html`.

Phillip Gage. 1994. A new algorithm for data compression. *The C User Journal*.

Hasindu Gamaarachchi, Hiruna Samarakoon, Sasha P. Jenner, James M. Ferguson, Timothy G. Amos, Jillian M. Hammond, Hassaan Saadat, Martin A. Smith, Sri Parameswaran, and Ira W. Deveson. 2022. *SLOW5 Specification (version 0.2.0)*. `https://hasindu2008.github.io/slow5specs/slow5-v0.2.0.pdf`.

Scott Gigante. 2017. Picopore: A tool for reducing the storage size of oxford nanopore technologies datasets without loss of functionality. *F1000Research*, 6.

J Goldstein, R Ramakrishnan, and U Shaft. 1998. Compressing relations and indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering*, ICDE '98, pages 370–379. IEEE Computer Society, Washington, DC, USA.

Google. 2022. Protocol buffers: Encoding: Signed integers. `https://developers.google.com/protocol-buffers/docs/encoding#types`.

The Open Group. 2004. *compress - Shell and Utilities Reference*, third edition. https://pubs.opengroup.org/onlinepubs/009695399/utilities/compress.html.

Mikel Hernaez, Dmitri Pavlichin, Tsachy Weissman, and Idoia Ochoa. 2019. Genomic data compression. *Annual Review of Biomedical Data Science*, 2(1):19–37.

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

IANA. 1993. *Registration of a new MIME Content-Type/Subtype - application/zip*. https://www.iana.org/assignments/media-types/application/zip.

IETF. 2012. *The 'application/zlib' and 'application/gzip' Media Types*. https://datatracker.ietf.org/doc/html/rfc6713.

Michael Kerrisk. 2021. *clock(3) - Linux manual page*. https://man7.org/linux/man-pages/man3/clock.3.html.

Donald E Knuth. 1985. Dynamic huffman coding. *Journal of Algorithms*, 6(2):163–180.

Jan Van Leeuwen. 1976. On the construction of huffman trees.

Daniel Lemire, Leonid Boytsov, and Nathan Kurz. 2014. SIMD compression and the intersection of sorted integers. *CoRR*, abs/1401.6399.

Daniel Lemire, Nathan Kurz, and Christoph Rupp. 2018. Stream VByte : Faster byte-oriented integer compression. *Information Processing Letters*, 130:1–6.

Matt Mahoney. 2013. Data compression explained. http://mattmahoney.net/dc/dce.html#Section_522.

Lara Marks. 2021. Nanopore sequencing. https://www.whatisbiotechnology.org/index.php/science/summary/nanopore/nanopore-sequencing-makes-it-possible-to-decode-the.

G. Nigel N. Martin. 1979. Range encoding: An algorithm for removing redundancy from a digitized message. In *Video & Data Recording Conference*. Southampton, UK.

B. McMillan. 1956. Two inequalities implied by unique decipherability. *IRE Transactions on Information Theory*, 2(4):115–116.

nanoporetech. 2022a. pod5-file-format. https://github.com/nanoporetech/pod5-file-format.

nanoporetech. 2022b. vbz_compression. https://github.com/nanoporetech/vbz_compression.

ONT. 2022. Flow cells and nanopores. https://nanoporetech.com/how-it-works/flow-cells-and-nanopores.

A.G. Ramakrishnan and S. Saha. 1997. Ecg coding by wavelet-based linear prediction. *IEEE Transactions on Biomedical Engineering*, 44(12):1253–1261.

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

A.H. Robinson and C. Cherry. 1967. Results of a prototype television bandwidth compression scheme. *Proceedings of the IEEE*, 55(3):356–364.

Julian Seward. 2019. *bzip2 and libbzip2, version 1.0.8*. `https://sourceware.org/bzip2/manual/manual.html`.

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

Jeffrey Scott Vitter. 1987. Design and analysis of dynamic huffman codes. *Journal of the ACM*.

W3C. 1990. *Graphics Interchange Format, Version 89a*. `https://www.w3.org/Graphics/GIF/spec-gif89a.txt`.

Yunhao Wang, Yue Zhao, Audrey Bollas, Yuru Wang, and Kin Fai Au. 2021. Nanopore sequencing technology, bioinformatics and applications. *Nature Biotechnology*, 39(11):1348–1365.

Terry Welch. 1984. A technique for high-performance data compression. *Computer*, 17(6):8–19.

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

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