The University of Melbourne
Department of Computing and Information Systems

Practical Compression for Multi-Alignment Genomic Files

by

Rodrigo Cánovas

Submitted to The University of Melbourne in total fulfillment of the requirements of the degree of Doctor of Philosophy

June 2015
Abstract

Next generation sequencing machines produce vast amounts of genomic data, generating files in the gigabyte range within an hour or less of technician time, and at a cost of just a few hundred dollars. This data is valuable for the insights it allows into the health of individuals and whole populations, and will continue to be of benefit into the future as medical knowledge grows. Therefore is essential to provide long-term storage for this information. Furthermore, in Bioinformatics it is starting to be fundamentally important to find optimal ways to be able to store and manipulate genomic data. This work examines and introduces a method in response to the combined challenge of compressing genomic data, while providing fast access to particular regions of interest without full decompression of whole files.

More specifically, this thesis considers genomic data stored in SAM (Sequence Alignment Map) format files. These files can contain millions of reads, each produced as a continuous fragment of data extracted from the processing of a single genome. The reads are stored as a string of bases, letters that indicate the fundamental molecules of DNA. In SAM files, each read is associated with extra meta-data information containing details of how the read was obtained, forming what is defined as alignment read data. The more alignment reads extracted from each genome, the greater the probability that similar reads representing the same genome area are found. This is known as the coverage of the sequence, which indicates the average number of reads aligned over the same nucleotide in the reference sequence. SAM files generally exhibit high coverage (meaning that many reads overlap and their associated meta-data elements contain many repetitions), thus contain considerable redundancy. This property of SAM files leads us to study how to exploit this redundancy in order to compress the information.

In this work we introduce CSAM (Compressed SAM format), a new compression approach offering lossless and lossy compression for SAM files. The structures and techniques proposed are suitable for representing SAM files, as well as supporting a mechanism for fast access to the compressed information. The methods exhibited in this work permit a more compressed lossless approach than BAM, which is currently the preferred lossless compressed SAM-equivalent format. Moreover, the approaches described in this thesis are the first existing lossy SAM compressor that provides random access to the data. Finally, all the compression method proposed are self-contained, that is, they do not depend of any additional input to compress or decompress SAM files; this is not so for the majority of the existing SAM compressors.
Declaration

This is to certify that:

(i) the thesis comprises only my original work towards the PhD except where indicated in the Preface,

(ii) due acknowledgement has been made in the text to all other material used,

(iii) the thesis is fewer than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Rodrigo Antonio Cánovas Barroso
Department of Computing and Information Systems
The University of Melbourne
June, 2015
Preface

The contributions made in this thesis are listed in the following publications of which I was the primary author in collaboration with my supervisors. Each publication is associated with a chapter.


Acknowledgments

There are many people I should thank for their support and help during the process of finishing this work. To start, I would like to thank my supervisors Alistair Moffat and Andrew Turpin for all the time and effort that they gave in order for me to complete this thesis. Also I would like to thank NICTA and The University of Melbourne for funding and supporting my research. Thanks to my advisory committee consisting of Wei Shi and Anthony Wirth for their valuable advice at progress meetings.

There are also many friends that supported me, listened to me, and helped me during the process. Their presence was indispensable for me and my mental health. Without any special order, thanks to Lizzie, Daniel, Lorena, Peter, Jorge, Moshen, Jessica R., Jessica M., Yi, Claude, Nicolas, Bernardita, Angeles, Chris, Whitney, Daniela, Adina, and so on. The list of people that in one way or another have helped is too big, but thanks to all of them.

Special thanks to my mother (Silviana), father (Rodrigo), sister (Camila), pseudo-brother (Pablo), grandparents (Silvia, Maruja and Rene) who always believed in me (but they still do not have any idea of what I am doing) and have been with me all the time. Without their existence I would never have come so far.

Finally I would like to thank Terry Pratchett (Good Omens, Going Postal, Mort, The Thief of Times, Men at Arms, etc...), Douglas Adams (The Ultimate Hitchhiker’s Guide to the Galaxy), Patrick Rothfuss (The Name of the Wind and The Wise Man’s Fear), Eliezer Yudkowsky (Harry Potter and the Methods of Rationality) and many other authors, who provided me with distraction and relaxation whenever I needed to disconnect from this world and find peace in a book that was not related with my thesis.


Contents

1 Introduction .............................................. 3
   1.1 Thesis Structure and Contributions .................. 6

2 Background ................................................. 9
   2.1 Compression Modalities ............................... 10
   2.2 Space Measure: Empirical Entropy .................... 12
   2.3 Integer Bit Encoding ................................ 13
      2.3.1 Unary Codes ...................................... 15
      2.3.2 Elias Gamma Codes .................................. 15
      2.3.3 Minimal-Length Binary Code .................... 16
      2.3.4 Golomb Codes ...................................... 17
      2.3.5 Parameterized VByte Coding .................... 18
   2.4 Bit Encoding of Non-Integers ......................... 20
      2.4.1 Huffman Code ..................................... 20
      2.4.2 Canonical Huffman Code .......................... 21
      2.4.3 Arithmetic Coding ................................. 23
   2.5 Succinct Data Structures .............................. 26
      2.5.1 Rank and Select .................................. 26
      2.5.2 The Burrows-Wheeler Transform (BWT) ............. 30
   2.6 General Purpose Compression Tools ................... 32
      2.6.1 Gzip .............................................. 33
      2.6.2 Bzip2 .............................................. 34
   2.7 Multi-Alignment Genomic Data ......................... 35
      2.7.1 DNA Sequence ...................................... 35
## CONTENTS

2.7.2 Genome Sequencing ................................................. 36
2.7.3 Sequence Alignment .................................................. 37
2.8 Genomic Data Formats .................................................. 39
  2.8.1 FASTA format ...................................................... 39
  2.8.2 FASTQ format ...................................................... 40
  2.8.3 CIGAR .............................................................. 41
  2.8.4 SAM format ......................................................... 43
2.9 Summary ............................................................... 47

3 Storing Multi-Alignment Files ............................................ 49
  3.1 DNA Sequence Compression ............................................. 50
    3.1.1 Human Genome Compression ...................................... 50
    3.1.2 SLIMGENE Tool Sequences ....................................... 52
    3.1.3 Block and SuperBlock DNA Compression .......................... 54
    3.1.4 ASCII DNA Compression .......................................... 55
    3.1.5 ORCOM .......................................................... 57
    3.1.6 Genomic SQueeZ (G-SQZ) ......................................... 58
  3.2 Quality Score Compression ............................................. 60
    3.2.1 SLIMGENE Tool Qualities ........................................ 61
    3.2.2 Block and SuperBlock Quality Score ............................... 62
    3.2.3 Binning Quality .................................................. 63
    3.2.4 QUALCOMP Tool .................................................. 65
  3.3 SAM File Compression ................................................ 66
    3.3.1 BAM Format ...................................................... 67
    3.3.2 CRAM Format ...................................................... 68
    3.3.3 NGC Compressor .................................................. 70
  3.4 Downstream Applications .............................................. 74
  3.5 Random Access Operations ............................................ 76
  3.6 Summary ............................................................. 77

4 Read Sequence Compression .............................................. 81
  4.1 Overlap Presumed Sequence .......................................... 82
    4.1.1 First-Base-Encountered ......................................... 82

x
## CONTENTS

4.1.2 Most-Frequent-Base ........................................... 85
4.2 Analysis and Tuning ............................................ 85
  4.2.1 Component Compression ................................. 93
4.3 Supporting Random Access ................................. 101
4.4 Experimental Results ........................................ 105
4.5 Summary ...................................................... 112

5 Quality Field Compression ............................. 115
  5.1 Fidelity Measures ........................................... 115
  5.2 Block Compression .......................................... 116
    5.2.1 P-Block .............................................. 118
    5.2.2 R-Block .............................................. 120
  5.3 Experimental Evaluation .................................. 122
    5.3.1 Coding Methods .................................... 126
    5.3.2 Distance Measure Experiments .................. 129
    5.3.3 Downstream Application: Variation Calling .... 134
  5.4 Summary ...................................................... 144

6 CSAM: Compressed SAM ................................. 147
  6.1 Compression Scheme ....................................... 147
  6.2 SAM Lossless and Lossy Compression .................. 153
    6.2.1 Lossless Compression ................................ 155
    6.2.2 Lossy Compression .................................. 162
  6.3 Supporting Random Access .............................. 167
    6.3.1 Buffer and Sample Sizes .......................... 167
    6.3.2 Comparative Experiments ......................... 174
  6.4 Downstream Applications: Feature Count ............ 180
  6.5 Summary ...................................................... 186

7 Conclusions .................................................. 189

Bibliography .................................................. 195
List of Figures

2.1 Compression modalities scheme .......................... 11
2.2 Example Huffman code ...................................... 22
2.3 Example Arithmetic code ................................. 24
2.4 Example of rank, select, and access ...................... 27
2.5 Example Wavelet Tree ..................................... 29
2.6 Example of the BWT array values ......................... 31
2.7 Example of Sequence Alignment .......................... 38
2.8 Example of a FASTA format file ......................... 40
2.9 Example of a FASTQ format file ......................... 41
2.10 Example of a CIGAR string .............................. 42
2.11 Example of a SAM file ................................. 45
2.12 Space percentages used for each component of the SAM file 47

3.1 Example of Christley et al.’s variations .................. 51
3.2 Example of SLIMGENE Tool over sequences ............. 53
3.3 Example ASCII DNA Compression ........................ 56
3.4 Example of the G-SQZ Tool’s methodology .............. 59
3.5 The quality compression schemes followed by Wan et al. 64
3.6 Theoretical transition scheme CRAMTOOLS ............. 68
3.7 Actual transition scheme CRAMTOOLS .................. 69
3.8 Example of the vertical run approach .................... 72
3.9 The NGC tool’s compression scheme ..................... 74
3.10 Example of a VCF file format ........................... 75
3.11 FeatureCounts program scheme .......................... 76
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.12</td>
<td>Example of a SAF file format.</td>
<td>76</td>
</tr>
<tr>
<td>4.1</td>
<td>Example first-base-encountered presumed sequence</td>
<td>83</td>
</tr>
<tr>
<td>4.2</td>
<td>Example last-read-rule presumed sequence</td>
<td>84</td>
</tr>
<tr>
<td>4.3</td>
<td>Example most-frequent-base presumed sequence</td>
<td>86</td>
</tr>
<tr>
<td>4.4</td>
<td>Read sequences field components</td>
<td>91</td>
</tr>
<tr>
<td>4.5</td>
<td>Offset position histogram</td>
<td>94</td>
</tr>
<tr>
<td>4.6</td>
<td>Copy array values histogram</td>
<td>98</td>
</tr>
<tr>
<td>4.7</td>
<td>Replace array values histogram</td>
<td>99</td>
</tr>
<tr>
<td>4.8</td>
<td>Example cluster division of reads data</td>
<td>102</td>
</tr>
<tr>
<td>4.9</td>
<td>Cluster and reads data structure example</td>
<td>103</td>
</tr>
<tr>
<td>4.10</td>
<td>GZIP vs BZIP2</td>
<td>109</td>
</tr>
<tr>
<td>5.1</td>
<td>P-BLOCK example</td>
<td>119</td>
</tr>
<tr>
<td>5.2</td>
<td>R-BLOCK example</td>
<td>123</td>
</tr>
<tr>
<td>5.3</td>
<td>Quality training files histogram of the quality scores</td>
<td>125</td>
</tr>
<tr>
<td>5.4</td>
<td>Quality training files average value per position</td>
<td>125</td>
</tr>
<tr>
<td>5.5</td>
<td>BINARY Global vs Local</td>
<td>126</td>
</tr>
<tr>
<td>5.6</td>
<td>Manhattan and Chebyshev Distance trade-offs</td>
<td>130</td>
</tr>
<tr>
<td>5.7</td>
<td>Mean Square Error and Max:Min Distance trade-offs</td>
<td>131</td>
</tr>
<tr>
<td>5.8</td>
<td>Soergel and Lorentzian Distance trade-offs</td>
<td>132</td>
</tr>
<tr>
<td>5.9</td>
<td>Definition of $TN$, $TP$, $FN$, and $FP$</td>
<td>135</td>
</tr>
<tr>
<td>5.10</td>
<td>Precision/recall for single letter quality</td>
<td>136</td>
</tr>
<tr>
<td>5.11</td>
<td>F-Score vs size for NA12878_low.quality</td>
<td>137</td>
</tr>
<tr>
<td>5.12</td>
<td>F-Score vs size for HG01477_quality</td>
<td>138</td>
</tr>
<tr>
<td>5.13</td>
<td>F-Score vs size for NA12878_high.quality</td>
<td>138</td>
</tr>
<tr>
<td>5.14</td>
<td>F-Score vs size for file data set</td>
<td>139</td>
</tr>
<tr>
<td>6.1</td>
<td>CSAM compression scheme</td>
<td>148</td>
</tr>
<tr>
<td>6.2</td>
<td>Example CSAM interval extraction</td>
<td>153</td>
</tr>
<tr>
<td>6.3</td>
<td>Lossless decompression time versus compression ratio</td>
<td>161</td>
</tr>
<tr>
<td>6.4</td>
<td>Lossy decompression time versus compression ratio</td>
<td>166</td>
</tr>
<tr>
<td>6.5</td>
<td>Random access time for test files</td>
<td>179</td>
</tr>
</tbody>
</table>
6.6 FeatureCounts measuring time scheme .......................... 182
6.7 FeatureCounts time performance ................................. 185
List of Tables

2.1 Examples of integer bit encoding ........................................ 14
2.2 Examples of CANONICAL HUFFMAN codeword assignment ............ 23
2.3 SAM format fields .......................................................... 44
3.1 Example quality scores ...................................................... 60
3.2 Related works summary ....................................................... 79
4.1 Training files used for read sequence compression ...................... 88
4.2 Overlap information for the training files ................................ 88
4.3 Accuracy percentage of overlapping bases ................................ 89
4.4 Theoretical space for read sequence and position fields ............... 92
4.5 Reads special symbols rates ................................................. 96
4.6 Column-wise versus Row-wise runlength encoding ...................... 97
4.7 Space used to store the RNAME, POS, and Read Sequence fields .... 107
4.8 Compression and decompression time for RPS files .................... 108
4.9 Random access time for SEQ, POS, and RNAME fields ............... 111
5.1 Examples of fidelity measures .............................................. 116
5.2 Training files used for quality scores compression ..................... 124
5.3 Example storage space required by P-BLOCK ............................ 127
5.4 Example storage space required by R-BLOCK ............................ 128
5.5 Measured recall and precision for NA12878\_low\_quality ............... 140
5.6 Measured recall and precision for HG01477\_quality .................... 141
5.7 Measured recall and precision for NA12878\_high\_quality ............... 142
5.8 F-Score file data set ....................................................... 143
6.1 Tested SAM Files ........................................ 154
6.2 Lossless compression command lines ...................... 156
6.3 SAM lossless compression sizes .......................... 158
6.4 Lossless compression speed ................................ 159
6.5 Lossless decompression speed ............................... 160
6.6 SAM lossy compression sizes ............................... 163
6.7 Lossy compression speed ................................... 164
6.8 Lossy decompression speed .................................. 165
6.9 Buffer size versus average time per line extracted ...... 169
6.10 Block sample index versus average time per line extracted .... 171
6.11 Position sample index versus average time per line extracted .... 173
6.12 Random access time for a low coverage SAM file ........ 176
6.13 Random access time for a high coverage SAM file ........ 177
6.14 FeatureCounts time performance for lossless files ......... 183
6.15 FeatureCounts time performance for lossy files ........... 184
List of Algorithms

1. **Unary**\( (x) \): returns the UNARY code representation of \( x \)  
2. **EliasGamma**\( (x) \): returns the GAMMA code representation of \( x \)  
3. **MinimalLengthBinary**\( (x, b) \): returns the MINIMAL-LENGTH BINARY code representation of \( x \) depending on the value of \( b \)  
4. **Golomb**\( (x, b) \): returns the GOLOMB code representation of \( x \) depending on the value of \( b \)  
5. **Huffman**\( (S) \): returns the HUFFMAN tree for the set \( S \)  
6. **Rank**\( _s (B, A, i) \): returns the number of occurrences of the symbol \( s \) in the sequence \( S \) until the position \( i \)  
7. **CountOcc**\( (P, m) \): returns the number of occurrences of \( P \) in a text \( T \)  
8. **GetInterval**\( (rname) \): returns the reads associated with the reference \( rname \)  
9. **ComputingBlock**\( (Q, \ell, \theta) \): returns the chosen representative values and their respective runlengths for the quality values \( Q \)  
10. **GetInterval**\( (rname, x, y) \): outputs the reads and their associated fields that are within given parameters
Chapter 1

Introduction

In the study of life forms, DNA is a major source of knowledge, carrying the genetic information and instructions used in the development and functioning of all cellular organism. Enormous effort has been made to fully understand DNA composition and functionality, discovering how the structure of DNA defines different aspects of each life’s existence. In the case of human beings, DNA composition (or the human genome) has been almost completely determined at a syntactic level, but its semantics and interpretation are still not fully understood. The differences between two individuals is on the order of 0.1 per cent of the total human genome, and understanding this small difference could help to bring more knowledge about how our body works and how, for example, different illnesses affect different individuals.

In the search for unraveling the functionalities and meaning of genetic information, Bioinformaticians are increasingly turning to Computer Science experts, whose role is to develop computational tools and formats to manipulate and analyze genomic data, allowing better management of information which in turn can lead to new biological discoveries.

The process of determining DNA composition is called sequencing. During the last fifty years, DNA sequencing methods have evolved at the same astonishing rate as computing technology. Currently, next-generation sequencing technologies,
which parallelize the sequencing process, produce millions of small DNA fragments (reads) at once [Chu06, Mar08, Ans09, MBJ12], generating file sizes in the gigabyte range at a cost of just a few hundred dollars. Each generated read is a continuous fragment of data extracted from the processing of a single genome, stored as a string of bases. In this thesis we considered reads composed of four fundamental bases Adenine (A), Cytosine (C), Guanine (G), and Thymine (T), with the inclusion of the letter N, which is used to symbolize bases that could take any value. For example, “GAACNGTA” is a read of length eight.

A number of meta-data fields are associated with each read to form alignment read information. Some of these fields use more space to be stored than the sequence of bases; one of currently most used fields information, beside the reads, is the Quality field, which gives a measurement of how accurate the bases of each read are with respect to a reference genome used to identify regions of similarity [EG98, EHWG98, Ric98]. In this thesis, one of the focuses is a study of how to store the Quality field, exploring different lossy compression approaches which minimize the possible effects on the data stored and results obtained when this data is used after being lossily compressed and then decompressed.

A later process, called sequence alignment, assembles the reads together, using their meta-data fields, to create a contiguous sequence that represents the DNA genome. In Bioinformatics, the mechanics of effective storing, extracting, and analyzing the information of reads that have being aligned has become fundamentally necessary for the future of the field. As a result, several standard formats to store alignment reads have been adopted, each aiming to make it easy to manipulate and parse the information using text-processing tools. The most common are the FASTA, FASTQ [CFG+10], and SAM (Sequence Alignment Map) [LHW+09] formats. Of these, SAM is the dominant format including more information about each alignment than the other formats.

All these formats keep the information in raw form, as plain ASCII text, which is useful for text-processing tools like Perl\(^1\) and Python\(^2\), but their main drawback

---

\(^1\)http://www.bioperl.org/wiki/Main_Page, December 2014
\(^2\)http://biopython.org/wiki/Main_Page, December 2014
is the space required to store the information. For example, for storing a DNA sequence, consisting of four bases (A, C, G, and T), only two bits are required to represent each base, whereas in FASTA, FASTQ, and SAM files use one byte per letter.

We consider the problem of effective storage of genome sequence data when it is denoted as alignment reads relative to a set of reference sequences. More specifically we consider data stored in SAM-style formats, which has become a standard format for storing alignment data, and is generated by many aligners\(^3\). For example, the compressed version of the SAM format, BAM, is the preferred format used by the 1000 Genome Project\(^4\), which is the first worldwide project to sequence the genomes of a large number of people, and will provide a comprehensive free public resource on human genetic variation.

In this work, different approaches that compress some or all of the fields associated with SAM files are explored. Most of the techniques described focus on methods that compress reads and/or their associated Quality fields, which, as we will describe, are the fields that dominate the space requirement of compressed SAM files. Another reason for focusing on these two fields is that most of the remaining fields can be derived from these two. We also consider the problem of random access into the stored data, providing data structures that allow the extraction of segments of the information stored without the need to decompress the whole compressed file.

This thesis introduces a new compressed SAM format, CSAM, which uses less space for storing the data than the BAM format, and also supports queries over multiple alignments without requiring whole files to be decompressed. The CSAM format is currently the only lossless SAM compressor, beside BAM itself, that offers a full compression technique that supports random access to the data. CSAM uses less space than BAM and takes similar or lower times to compress, decompress and access the data. Furthermore, CSAM is the first SAM lossy compression approach allowing random access to the stored data. We also explore how the proposed compression techniques affect the performance of possible future uses of the

\(^3\)http://samtools.sourceforge.net/swlist.shtml, December 2014
\(^4\)http://www.1000genomes.org/, December 2014
compressed data, studying the trade-offs between performance time and accuracy of the result obtained.

In general, the existing SAM compression techniques use an external reference sequence as input for compressing and decompressing data, with the availability of the reference essential to both processes. The CSAM format compresses the data without using any external extra information, at the same time offering better compression ratios than BAM, and similar functionality.

1.1 Thesis Structure and Contributions

The following are the research questions and the respective chapters that cover the content of this thesis.

How are genomic data files currently stored?

To answer this question we begin in Chapter 2, describing an overview of the basic knowledge of the algorithms, compression techniques, and genomic file formats necessary for understanding existing genomic data compression approaches. Also in this chapter we explain the concept of compression modality, genomic data formats and multi-alignment genomic data, reviewing how these files are generated, and describing their components and importance.

Chapter 3 then explores the existing genomic file compression techniques. More precisely, Chapter 3 discusses how FASTQ and SAM files, or some of their fields are currently being compressed, separating the approaches that compress DNA reads from those that focus on compressing the Quality field, and those that offer complete SAM file compression. This chapter also includes information about the two downstream applications that are used later (in Chapter 5) to measure the impact of lossy compressed SAM files.

How can the Read Sequence field be efficiently and effectively compressed?

Having studied different methods to store reads, Chapter 4 introduces a new
technique to compress the Read Sequences field, including an analysis of the practical implementation of the ideas proposed. The approach described in this chapter compresses the Read Sequences field without using an external reference as input (which most other read compressors do), instead creating and storing a presumed sequence which acts as an artificial reference sequence. Chapter 4 describes how the reads, and their respective position and reference name values (compulsory fields for SAM files), are compressed using the presumed sequence approach. The initial scheme of the ideas discussed in this chapter was published at the Thirty-Sixth Australasian Computer Science Conference, ACSC 2013 [CM13].

*How can the Quality field be efficiently and effectively compressed?*

Chapter 5 describes our proposal of how to compress the Quality field, and compares this technique against existing methods. Two main approaches are discussed, P-BLOCK and R-BLOCK, where both algorithms group quality scores using a given predefined criteria, offering trade-offs of space versus fidelity of the data stored. The evaluation includes a range of fidelity measures, and also evaluates the impact on the output of the downstream application of Variant Calling. This work was published in *Bioinformatics* [CMT14].

*How can CSAM be structured to allow fast interval queries?*

With the Read Sequences and Quality fields compressed, the next step is to combine the approaches showed in order to compress complete SAM files. Chapter 6 introduces CSAM, a new lossless and lossy compressed representation for SAM files, which supports random access to the stored data. In this chapter we compare CSAM with the other compress SAM methodologies, analyzing their differences and trade-offs. Also we compare how CSAM and BAM perform when random access queries are computed, and show how CSAM affects performance of the downstream application Feature Counting.

Finally Chapter 7 summarizes the conclusions obtained from this thesis and discusses possible further lines of research.
Chapter 2

Background

Before presenting our study about compression of multi-alignment genomic files, we need an overview of some basic knowledge of compressed data structures, genomic data, and their relationships. The following sections introduce the algorithms, compression techniques, and genomic file formats necessary for understanding the chapters ahead.

The amount of digital information available is growing at an astonishing rate, and storage is of vital importance. For example, in the area of Bioinformatics, during the last fifty years, DNA data extraction methods have evolved, producing millions of data sequences at once [Chu06, Mar08, SJ08, Ans09, MBJ12] and generating files in the gigabyte range at a cost of just a few hundred dollars. Given the increase of data, it has became necessary to find storage techniques to maintain the data in a manageable amount of space.

Compressed data structures aim to store information using less space than the raw data. This concept raises two questions, how can we measure compression levels; and how can we compress the information. To address these question we need to define the notation used in this thesis. A string is a sequence of characters, its alphabet is denoted by $\Sigma$ and the alphabet size is $|\Sigma| = \sigma$, and $\Sigma$ assumed to be totally ordered. The length (number of characters) of a string $S$ is denoted $|S|$, and
a substring between positions $x$ and $y$ (both inclusive) of $S$ is denoted as $S[x..y]$. Additionally, all logarithms are in base 2 unless stated otherwise, and whenever we express a size using megabytes (MB), we assume that 1 MB = 1,048,576 bytes (we recognize that the correct notation should be mebibyte\textsuperscript{1}, but the unit has not yet been widely adopted\textsuperscript{2-3}).

The following sections explain conceptually which compression modalities can be found, and present the techniques used in this work to measure and compress information. Furthermore, in this chapter we also go through some basic concepts of genomic data formats and multi-alignment genomic data, reviewing how these files are generated, and explaining their components and importance.

### 2.1 Compression Modalities

We categorize the compression modalities into three distinct classes: *lossless*, *information-preserving*, and *lossy*. Each class has advantages and disadvantages, depending on the importance of the information that is stored, and on the eventual use of the data.

*Lossless*, or exact compression, ensures that the decompressed data is exactly the same as the original, and means that the compressed version must contain sufficient information for the decompressor to generate a bit stream identical to the one that was input to the compressor.

In some cases the information that a file contains is more important than the exact order in which it is stored; for example, if the data in the file is a set of independent facts. An *information-preserving* compression modality guarantees that all the information is stored, but the order in which it is stored and regenerated, might be different than in the original file. That is, if we regard the order in which the data is presented as being of no importance, then shifting the input

\begin{itemize}
  \item \footnotesize{1\textsuperscript{http://physics.nist.gov/cuu/Units/binary.html}, January 2015}
  \item \footnotesize{2\textsuperscript{http://www.iec.ch/si/binary.htm}, January 2015}
  \item \footnotesize{3\textsuperscript{http://www-cs-faculty.stanford.edu/~uno/fasc1.ps.gz}, page 94, January 2015}
\end{itemize}
into a particular arrangement might allow better compression. In such cases, the decompressed output will also be in that permuted order, and so the original input file cannot be exactly regenerated. Nevertheless, there might also be a sense in which all of the actual information embedded in the source data has been preserved, even if the physical representation has not. For example, a dictionary might be ordered and then coded [WMB99].

*Lossy* compression modalities store an approximate representation of the input data, trading loss in fidelity of reproduction for enhanced compression effectiveness. Lossy compression methods are typically applied to data originally sampled from continuous domains, and are based on the recognition that the process of turning that data into digital form can, within limits, be further approximated to save space. This technique has proven to be highly useful for image, video, and sound compression, including the widely-used *JPEG* and *MPEG3* standards [SP11,BW12]. For example, digital cameras take images that can be stored in either .jpg form (lossy compression) or as .raw files (larger uncompressed files).
Figure 2.1 illustrates these notions. A lossless compression mechanism must exactly recreate the input file, in all syntactic detail, as indicated by the bidirectional arrows. An information preserving regime will not be able to reproduce the original file (denoted by a single arrow), but once it has been decompressed a first time into its new form, it can be compressed and decompressed a second time without further change taking place (the bidirectional arrows at the top of the diagram). A lossy scheme cannot reproduce the input file; nor is there even any guarantee that a second iteration of compression and decompression will achieve the same file.

2.2 Space Measure: Empirical Entropy

A common measure of the compressibility of a sequence is its empirical entropy [SW49, CT06]. For a text $T$ of length $n$ drawn from an alphabet $\Sigma$ of size $\sigma$, the zero-order empirical entropy, $H_0$, is defined as

$$H_0(T) = \sum_{c \in \Sigma, n_c > 0} \frac{n_c}{n} \cdot \log \frac{n}{n_c},$$

where $n_c$ is the number of occurrences of the symbol $c$ in $T$. The formula $H_0(T)$ represents the average number of bits needed to represent a symbol of $T$ using a zero-order model, and is a lower bound on the compression that can be achieved when each symbol is encoded independently of the surrounding symbols.

In this work we use the zero-order empirical entropy as an estimate of the minimum number of bits needed to identify each symbol of a given text or set of symbols. For example if we have a text, $T$, of length $n$ and $\Sigma = \{A,C,G,T\}$, where the probabilities of occurrence of each letter is $\frac{2}{5}$, $\frac{3}{10}$, $\frac{1}{5}$, and $\frac{1}{10}$ respectively, then $H_0(T) \approx 1.85$, meaning that, if each symbol in $T$ is independent, the lower bound of bits needed to represent the text would be $1.85 \cdot n$. Note that, if all the symbols have the same probability, computing the zero-order entropy is the same
as calculating the number of bits needed to represent \( \sigma \) symbols. For the same alphabet,

\[
H_0(T) = \sum_{c \in \{A,C,G,T\}} \frac{n_c}{n} \cdot \log \frac{n}{n_c} = \sum_{c \in \{A,C,G,T\}} \frac{1}{4} \cdot \log 4 = 2,
\]

which is the same as computing \( \log \sigma \).

The zero-order entropy considers each symbol separately, but in some cases the symbol order of appearance creates a predictive context. For example, if we consider the context \( th \) in English, it is likely to find an \( a, e, i, o \) or \( u \) following \( th \), but it is very unlikely to find a \( k \). The definition of empirical entropy can be extended to consider contexts [Man01] as

\[
H_k(T) = \sum_{s \in \Sigma^k} \frac{|T^s|}{n} \cdot H_0(T^s),
\]

where \( T^s \) is the sequence of symbols in \( T \), preceded by the context \( s \) and \( \Sigma^k \) is the set of all strings of length \( k \) that occur in \( T \). This is called the "k th order Empirical Entropy".

### 2.3 Integer Bit Encoding

In the process of compressing a sequence of symbols, it is necessary to represent them as bits. That is, once the set of symbols to be coded has been determined, the compressor encodes each symbol into a bit sequence. For example, the binary code of a non-negative number (integers greater or equal to zero) is represented as a sequence of zeros and ones, where the \( i \) th position of the sequence represents the value \( 2^{i-1} \). Hence the number 13 is represented by the sequence \( 1101 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \).
### 2.3 Integer Bit Encoding

<table>
<thead>
<tr>
<th>Integer</th>
<th>BINARY ((b = 3))</th>
<th>UNARY</th>
<th>GAMMA</th>
<th>GOLOMB ((b = 3))</th>
<th>P-VBYTE ((b = 2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>-</td>
<td>-</td>
<td>0 0</td>
<td>000</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>0</td>
<td>0</td>
<td>0 10</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>010</td>
<td>10</td>
<td>10 0</td>
<td>0 11</td>
<td>010</td>
</tr>
<tr>
<td>3</td>
<td>011</td>
<td>110</td>
<td>10 1</td>
<td>10 0</td>
<td>011</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>1110</td>
<td>110 00</td>
<td>10 10</td>
<td>100 000</td>
</tr>
<tr>
<td>5</td>
<td>101</td>
<td>11110</td>
<td>110 01</td>
<td>10 11</td>
<td>100 001</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>111110</td>
<td>110 10</td>
<td>110 0</td>
<td>100 010</td>
</tr>
<tr>
<td>7</td>
<td>111</td>
<td>1111110</td>
<td>110 11</td>
<td>110 10</td>
<td>100 011</td>
</tr>
<tr>
<td>8</td>
<td>-</td>
<td>11111110</td>
<td>111 000</td>
<td>110 11</td>
<td>101 000</td>
</tr>
</tbody>
</table>

Table 2.1: Examples of BINARY, UNARY, GAMMA, GOLOMB, and P-VBYTE codes for the integers 0 to 8, assuming that values in positive ranges are to be coded. The BINARY code shown here uses 3 bits per integer (and hence has an upper limit of 7 and while BINARY could return the 3 less significant bits of 8, we decided to not display this result in the table to avoid confusion); the GOLOMB code is constructed using parameter \(b = 3\); and P-VBYTE use size of block \(b = 2\). In this work we do not encode the zero integer value using UNARY or GAMMA codes but most applications of the other three codes need to be able to represent the value zero. If required, ranges can be shifted by one, without loss of generality.

A fixed-length binary code or variable-length prefix-free code is often used to represent symbols contained in data, with the choice dependent on the probability distribution that arises over the set of symbols. In this work we make use of several different such codes for non-negative integers. As a further point of distinction, the BINARY code assumes that all of the symbols have equal probability and that the range of input symbols is finite; whereas variable-length prefix-free codes, in general, assume that the symbol probabilities are non-increasing, and place no upper limit on the values that can be represented. Moffat and Turpin [MT02] present all of these mechanisms in detail.

Table 2.1 shows examples of some of the integer bit encoding used in this thesis, covering the first 8 positive integers plus (in some cases) the zero value. In the following subsections we explain the details of each of the variable-length prefix-free codes presented in Table 2.1.
Algorithm 1  *Unary*(*x*) takes an integer *x* > 0 as input, and returns the *Unary* code representation of *x*.

1: \(rep \leftarrow \{}\)
2: for \(i \leftarrow 1\) to \(x - 1\) do
3: \(rep \leftarrow rep + "1"\)
4: end for
5: \(rep \leftarrow rep + "0"\)
6: return \(rep\)

### 2.3.1 Unary Codes

The simplest variable-length prefix-free code is *Unary*, which represents a positive integer number *x* as a sequence of \(x - 1\) one bits, followed by a zero bit. For example, the number 13 is encoded into the sequence 111111111110. Algorithm 1 gives pseudocode that returns a string containing the *Unary* code of an input positive integer, where \(\{}\) represents an empty string and the symbol \("+"\) between strings means concatenation.

*Unary* has the advantage of being able to represent arbitrarily large values, but requires as many bits as the value being coded. Small numbers have short codes, for instance, in a machine where each integer number uses 32 bits in *Binary*, the numbers 1 to 31 can be represented using *Unary* code saving space, but for greater numbers the codeword length grows quickly, using more space than simply using a 32-bit *Binary* encoding.

### 2.3.2 Elias Gamma Codes

The Elias Gamma code [Eli75] (or just Gamma code) is more complex than *Unary*, but generally more effective too, encoding a positive integer *x* as the concatenation of the length of its *Binary* representation in *Unary*, and its *Binary* representation omitting the most significant bit. For example, the Gamma encoding of 13, knowing that its *Binary* code uses 4 bits, is the sequence 1110 101. Algorithm 2 presents the pseudocode, which returns a string containing the Gamma code of an input positive
Algorithm 2 $\text{EliasGamma}(x)$ takes an integer $x > 0$ as input, and returns the Gamma code representation of $x$.

1: $\text{rep} \leftarrow \{\}$
2: $\ell \leftarrow 1 + \lceil \log(x) \rceil$
3: $\text{rep} \leftarrow \text{Unary}(\ell)$
4: $\text{rep} \leftarrow \text{rep} + \text{Binary}(x, \ell - 1)$
5: \textbf{return} $\text{rep}$

integer, where, as in Algorithm 1, “{}” represents an empty string and the symbol “+” between strings means concatenation. Also, $\text{Unary}(x)$ returns the output of Algorithm 1 using $x$ as input, and $\text{Binary}(x, b)$ returns a string representation of the $b$ least significant bits of the non-negative integer $x$ (if $b$ is zero, it returns an empty string).

Gamma uses $1 + 2 \cdot \lceil \log x \rceil$ bits to encode a positive integer $x$, and, while the number of bits might appear to be double the space needed by the Binary representation of the number, it allows encoding of a set of positive integers using variable-length codes for each value, which is preferable when the values to code follow a probability distribution where small values are more frequent than large values, or it is not possible to know in advance how large the values can be.

### 2.3.3 Minimal-Length Binary Code

The Minimal-length Binary codes [WMB99, page 119] offer a small variation over Binary codes, depending on the number of different possible values to be encoded. If the number of possible values is $b$, it is easy to Binary encode each of them using $\lceil \log b \rceil$ bits per value and a mapping between the values and the range 0 to $b - 1$. Minimal-length Binary codes represent a value $x$ depending on $b$: if $b$ is a power of 2, then $x$ is encoded using $\lceil \log b \rceil$ bits; else, if $b$ is not a power of 2, the encoding of $x$ uses $\lceil \log b \rceil - 1$ bits if $x$ is smaller than $2^{\lceil \log b \rceil} - b$, otherwise the minimal-length Binary encodes $x$ as $x + 2^{\lceil \log b \rceil} - b$ using $\lceil \log b \rceil$ bits. For example, if $b = 13$, then minimal-length Binary encodes the values from 0 to 2 using 3 bits per value, and the values 3 to 12 are incremented by 3.
Algorithm 3 \textit{MinimalLengthBinary}(x,b) takes a non-negative integer $x < b$ and a parameter $b > 0$ as input, and returns the Minimal-length Binary code representation of $x$ depending on the value of $b$.

1: $rep \leftarrow \{\}$
2: $n \leftarrow \lceil\log(b)\rceil$
3: \textbf{if} $x < 2^n - b$ \textbf{then}
4: \hspace{1em} $rep \leftarrow \text{Binary}(x, n - 1)$
5: \textbf{else}
6: \hspace{1em} $rep \leftarrow \text{Binary}(x + 2^n - b, n)$
7: \textbf{end if}
8: \textbf{return} $rep$

and Binary encoded using 4 bits per value. Algorithm 3 presents the pseudocode that computes the Minimal-length Binary encoding of an integer $x < b$, where $b$ is a user-defined parameter. As before, the function \textit{Binary}(x,n) returns a string representation of the $n$ least significant bits of the binary integer $x$.

While Minimal-length Binary codes is not usually used as an independent variable-length code, it forms part of more complex bit encodings such as Golomb codes.

### 2.3.4 Golomb Codes

The Golomb code \cite{Golomb66}, as for Gamma, consists of a Unary/Binary combination. Golomb encodes a positive integer $x$ making use of an user-defined parameter $b$, storing the quotient $q = \lfloor x/b \rfloor$ with Unary codes (note that we need to add one to $q$ so it can be represented using this encoding), and a Minimal-length Binary component containing the remainder $r = x - b \cdot q$. For example, if $b = 3$ and the value to encode is $x = 13$, the Golomb representation of $x$ is the Unary encoding of 4+1, followed by the Minimal-length Binary encoding (using $b$ as parameter) of 1, giving the final encoded sequence equal to 11110 01. Algorithm 4 shows the pseudocode of this process.
Algorithm 4 \texttt{Golomb}(x,b) takes an integer \( x > 0 \) and a parameter \( b > 0 \) as input, and returns the \texttt{GOLOMB} code representation of \( x \) depending on the value of \( b \).

1: \( \texttt{rep} \leftarrow \{ \} \)
2: \( q \leftarrow \lfloor x/b \rfloor \)
3: \( r \leftarrow x - b \cdot q \)
4: \( \texttt{rep} \leftarrow \texttt{Unary}(q + 1) \)
5: \( \texttt{rep} \leftarrow \texttt{rep} + \texttt{MinimalLengthBinary}(r, b) \)
6: \( \text{return} \ \texttt{rep} \)

It is clear that the number of bits used to store an element depends on the value of \( b \), which should be small if the elements to be encoded are in general small, and larger if most of the elements have big values. In other words, it is desirable to choose the parameter \( b \) according to the characteristics of the set of values to be stored. Gallager and Van Voorhis [GV75] demonstrate that the \texttt{GOLOMB} code is optimal for encoding geometric distributions if the value of \( b \) is calculated as

\[
b \approx 0.69 \cdot \frac{N}{f},
\]

assuming that \( f/N \ll 1 \), where \( f \) is number of values to be encoded, and \( N \) is the number of possible values to be found in the set. In this work whenever we mention a geometric distribution, we will refer to set of values that follow the equation

\[
g(x) = t \cdot (1 - t)^x,
\]

where \( t \in (0, 1] \) is the parameter of the distribution. Note that if we are using \texttt{GOLOMB} codes, with \( b \approx 0.69 \cdot \frac{N}{f} \), then this encoding technique should be optimal for a set of values that approximate a geometric distribution with \( t = f/N \).

### 2.3.5 Parameterized VByte Coding

The \texttt{PARAMETERIZED VBYTE} coding [WZ99,BLN09] (or \texttt{P-VBYTE}) encodes a non-negative integer \( x \) by splitting the number of bits needed to represent the value (in \texttt{BINARY} using \( \lceil \log x \rceil \)), into blocks of \( b \) bits and storing each block in a chunk of
b + 1 bits. The highest bit of each chunk indicates if another chunk is necessary to represent the number, that is, the first chunk contain the b most significant bits of x, preceded by a one if another chunk is needed, otherwise preceded by a zero. If another block is added, its indicator bit shows if another is required, and so on until the number is fully represented. For example, if \( x = 13 \) and \( b = 3 \), the binary representation of \( x \) is 1101, which means that two blocks are needed. Then P-VBYTE encodes \( x \) as 1001 0101. In practice the P-VBYTE encoding technique is a little more complicated. When more than one chunk is used, the first bit of each chunk is considered part of the block information of the next block, forcing modification of the information stored, and involving a small amount of additional computation in the encoding and decoding steps. For example if \( b = 3 \), then the number 13 needs two chunks, where the first bit of the first chunk indicates that the number to be coded is bigger than 8 (because 3 bits were not enough), and the bits of the blocks represents the binary value of the difference between 13 and 8, finally 13 being coded as 1000 0101. This process allows us to win one extra bit to represent more numbers using the same amount of chunks.

P-VBYTE is normally used to encode arrays of non negative numbers, where if \( b \) is small, few bits will be use to encode smaller values but more chunks are needed for larger values, expending an extra bit per each extra chunk used. Otherwise, if \( b \) is large, larger numbers need less chunks, but smaller values are represented using more bits. The selection of \( b \) depends on the set of values to represent, with the optimal choice [BLN13] computed as

\[
b = \sqrt{\frac{N_0}{n}},
\]

where \( n \) is the number of values to encode and \( N_0 \) is the sum of the bits needed to encode each value using Binary codes (\( \lceil \log x \rceil \), where \( x \) is the value to be coded). In general, to maximize encoding and decoding speed, the size of \( b \) is of length 7 (meaning that each chunk use 1 byte), which is better known as VBYTE coding.
2.4 Bit Encoding of Non-Integers

In cases where the values to be encoded are not integers, or are not in the desirable range of valid elements to be coded, it is possible to create a mapping between the elements and numbers that can be coded. For example, when the input contains negative values, a simple mapping can be used to fold the whole integer number line on to the non-negative half, taking $x' = 2x - 1$ when $x > 0$, and $x' = -2x$ when $x < 0$. In the following subsections we describe bit encoding methodologies that offer more elaborate solutions to the problem of encoding series of values, when the values to be encoded could be any possible symbol; that is, not just numbers.

2.4.1 Huffman Code

If a set of non integer symbols is to be encoded, it would be advantageous to represent the symbols with higher occurrence probability using small bit sequences than the less probable symbols in the set. A simple solution is to store a table mapping the symbols to integers, arranged by their probabilities, so that codes such as the ones in Table 2.1 can be used. As we described, each of these codes offer different tradeoffs depending the frequency distribution of the values to be encoded.

In cases when the probability distribution is irregular and does not fit one of those standard distributions, Huffman [Huf52] presented a methodology, known as a Huffman code, which creates a binary tree with the symbols at the leaves that gives a minimum-redundancy code for the symbol probabilities. At the beginning of the algorithm each symbol is considered as a node with a weight equal to the probability of the symbol. Then at each step, the two nodes with lowest probabilities are linked to form a parent node that has the sum of the probabilities as its weight. At the end a “Huffman” tree, is formed, which defines the length of the codeword used to encode the symbols. Algorithm 5 shows the pseudocode of how to construct the Huffman tree of a set $S$, where each $s \in S$ is a binary node with an attribute $s.freq$ gives the frequency of each of the symbols in the alphabet to be coded. The method $ExtractMinimum(Q)$ extracts the element in $Q$ with lowest $freq$ value, deleting it
Algorithm 5  \textit{Huffman}(S) \ takes a set \( S \) as input, where the elements of \( S \) are binary nodes containing the frequency, \( freq \), of each symbol of the alphabet to be coded, returning the \textsc{Huffman} tree for the set \( S \).

\begin{algorithm}
1: \( n \leftarrow |S| \)
2: \( Q \leftarrow S \)
3: \textbf{for} \( i \leftarrow 1 \) to \( n - 1 \) \textbf{do}
4: \hspace{1em} allocate a new node \( z \)
5: \hspace{1em} \( z.\text{left} \leftarrow \text{ExtractMinimum}(Q) \)
6: \hspace{1em} \( z.\text{right} \leftarrow \text{ExtractMinimum}(Q) \)
7: \hspace{1em} \( z.\text{freq} \leftarrow z.\text{left}.\text{freq} + z.\text{right}.\text{freq} \)
8: \hspace{1em} \text{Insert}(Q, z) \)
9: \textbf{end for}
10: \textbf{return} \( \text{ExtractMinimum}(Q) \)
\end{algorithm}

from the set; and \text{Insert}(Q, z) \ add the element \( z \) to the set \( Q \).

Given a symbol, a simple way to specify a bit sequence to be encoded, is by traversing the \textsc{Huffman} tree from the root to the leaf that contains the symbol, recording the direction in which we move from one node to another using a zero or one bit. Figure 2.2 shows an example of a the construction of the \textsc{Huffman} tree for a sequence of letters, and a corresponding encoded bit sequence.

The representation of \textsc{Huffman} code presents two problems: the \textsc{Huffman} tree, which could need a large amount of space; and each time a symbol is decoded, it is necessary to traverse the tree.

### 2.4.2 Canonical Huffman Code

The \textsc{Canonical Huffman} code [SK64, HL90] is a variation of the \textsc{Huffman} code, which does not store the \textsc{Huffman} tree. Instead, it stores the symbol’s codeword length, ordering the symbols sequentially by this length. By creating the \textsc{Huffman} tree it is possible to obtain the length of each codeword, and the number of symbols which are encoded using the same number of bits. A more efficient method [MK95, MT97, WMB99] to determine the codeword lengths is using a heap-based algorithm that computes the codeword length of each symbol given its
2.4 Bit Encoding of Non-Integers

Sequence: A G A G C A A T N A A C

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Count</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>6</td>
<td>50.0</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>16.7</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
<td>16.7</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td>8.3</td>
</tr>
<tr>
<td>N</td>
<td>1</td>
<td>8.3</td>
</tr>
</tbody>
</table>

encoded sequence: 0 101 0 101 100 0 0 110 111 0 0 100

Figure 2.2: Example of the Huffman tree and the resulting output encoded sequence of “AGAGCAATNAAC”. To decode the encoded sequence the stored bits are visited sequentially while traversing the tree, obtaining the symbols each time a leaf is reached.

Another possible option is to use the in-place method of Moffat and Katajainen [MK95].

In the Canonical Huffman code method, the codeword length of each symbol is the same as in Huffman code, but the codewords are carefully reassigned. Symbols with the same codeword length are grouped together, and within the same length, increasing numerical values are assigned to symbols. To achieve this, codewords are assigned in incremental order, starting from zero and each time that a new codeword length is reached the value is increased by one and multiplied by two (same as shifting the bit representation by one bit), ensuring that the codes generated are not prefixes of longer codewords. Table 2.2 shows the codewords assigned by Canonical Huffman, when the example from Figure 2.2 is used.

The encoding process only needs the list of symbols ordered by codeword length, and a sorted array with the lengths. While, to decode a symbol, the decoder uses the sorted list of symbols plus an array containing the codeword of the first
Symbol | **HUFFMAN** | **Codeword length** | **CANONICAL HUFFMAN**
--- | --- | --- | ---
A   | 0   | 1   | 0
C   | 100 | 3   | 010
G   | 101 | 3   | 011
T   | 110 | 3   | 100
N   | 111 | 3   | 101

Table 2.2: Examples of **Canonical Huffman** codeword assignation, when the encoded sequence is “AGAGCAATNAAC”. We also included the HUFFMAN codeword originally assigned using the HUFFMAN tree from Figure 2.2.

symbol of each codeword length. The decoder first finds the last codeword in the stored codeword array whose value is lower than the codeword to be decoded, then computes the offset between these values. Finally, using the calculated offset, the decoder obtains the value of the symbol within the sorted symbol list.

In this work, whenever we mention that a HUFFMAN code is being used, we are referring to its **Canonical** implementation.

### 2.4.3 Arithmetic Coding

**Arithmetic** coding [Ris76,WNC87,MNW98] presents a methodology that, instead of assigning a codeword to each symbol, encodes the stream of symbols as a floating value in the interval \([0, 1)\), which is later represented as a stream of bits. The algorithm divides the interval \([0, 1)\) into subintervals, where each represents the probability of a symbol. For example, using the probabilities in Figure 2.2, if \(\Sigma = \{A, C, G, T, N\}\) the algorithm would assign the intervals \([0, 0.50)\), \([0.50, 0.67)\), \([0.67, 0.83)\), \([0.83, 0.92)\), and \([0.92, 1)\), to the symbols \(A, C, G, T\) and \(N\) respectively. When a new symbol is to be coded, say the symbol presented in the interval \([x, y)\), the interval \([x, y)\) becomes the main interval, and is divided into subintervals based on the symbol probabilities used. This process continues until the last symbol to be encoded is read, and a floating number within the last interval is chosen as the representation of the sequence of symbols visited. Figure 2.3 shows this process for the sequence “AGAGC”, using the distributions just presented.
The encoder only needs to store the final representative number, the probability distribution of the symbols used, and the length of the sequence encoded, in order for the decoder to reproduce the sequence of symbols. The decoding of the encoded sequence is performed by moving from the initial distribution intervals in the interval [0, 1) through the intervals where the representative floating number selected belongs, updating the new intervals after each symbol is decoded until the length of the sequence is reached.

The only complication left is how to chose the representative number and how to encode this number. ARITHMETIC code encodes the number using binary fraction coding, which works similarly to BINARY codes but the bit positions represent $2^{-1}$, $2^{-2}$, $2^{-3}$, $2^{-4}$, and so on. This coding presents the problem that some fractional
numbers are not possible to be precisely represented (for example, they may need an infinite number of bits). Given this problem, the number used to represent the final interval is chosen so it is the first precise finite binary fraction encoded number found within the range, minimizing the number of bits used to identify the floating number.

The algorithms presented for HUFFMAN and ARITHMETIC use static symbol probabilities, which do not change during the coding and decoding process. It is also possible to use these codes in an adaptive mode, where the probability of each symbol may change after each codeword is produced [MT02]. As long as the encoding and decoding process use the same update algorithm for the probabilities, and the encoder only uses the information that would be available to the decoder at time of decoding, then the correct data can be recovered. It is also possible to use the codes in a semi-static mode where the probability per symbol is computed using the whole data before the encoding process starts. Adaptive codes are good for adjusting codewords to local changes in data, and may required more than one context (see page 13), which is not the case in the compression applications in this thesis. For now on, we only consider codes in static mode and probabilities derived in either a static or semi-static fashion.

ARITHMETIC codes, in general, offer better compression than using HUFFMAN codes, but its static mode is slower, and does not support random access to the symbols encoded, given that all of the sequence is coded as a single binary fraction floating number. Note that using ARITHMETIC coding one can modify the algorithm so that the representative number is encoded at the same time that symbols are processed, by checking in each step the higher and lower limits of the new interval, outputting any known bits, and updating the binary fraction code range using this information [WMB99].
2.5 Succinct Data Structures

When the amount of information to be stored is large, compression techniques can be applied to reduce the storage space, but this leaves the problem of how to manage and access the information after it has been compressed. A succinct data structure [Jac88] is a compressed representation of a structure that uses an amount of space close to the information theoretic lower bound, together with efficient algorithms for carrying out any required access operations. In other words, succinct data structures offer compression techniques that store the data in a compressed way so it can still be queried or extracted depending on the purpose and use of the stored data.

Compressed indexes are an important example of succinct data structures. Compressed indexes take advantage of the regularities of the text to operate in space proportional to the size that would be occupied by a compressed (not indexed) version of the same text. An even more powerful concept is that of a self-index, which is a compressed index that, in addition to providing search functionality, contains enough information to efficiently reproduce any text substring. A self-index can regenerate the entire original text and hence can be used to replace the text.

2.5.1 Rank and Select

Two of the most basic operations used in compressed data structures are known as rank and select. Given a sequence $S$ of symbols, $\text{Rank}_a(S, i)$ counts the number of occurrences of the symbol $a$ in $S$ until position $i$, and $\text{Select}_a(S, i)$ finds the position in $S$ of the $i$th occurrence of $a$. The operation $\text{Access}(S, i)$, that returns the symbol at position $i$ in the sequence, is normally also required, and has often been supported using rank and select.

Binary Sequences

The most basic case is when the sequence has $\sigma = 2$ and $\Sigma = \{0, 1\}$, which is called a binary sequence [GGMN05, CN08]. Figure 2.4 shows an example of rank, select,
and access queries over a binary sequence.

Whenever a sequence has an alphabet of size two, it can be easily be transformed into a sequence with $\Sigma = \{0, 1\}$. The advantage of doing this transformation is that only one bit is used to represent each symbol and that for these bit sequences there exist many data structures that answer rank, select, and access. A simple data structure that solves these operations, storing the binary sequence plus some extra information is introduced by Munro et al. [Mun96]. Given a binary sequence $B$ of length $n$, their solution used $n$ bits to store $B$ itself and $o(n)$ additional bits for answering rank and select queries. This solution was implemented in practice by González et al. [GGMN05], and it requires 5% extra space over the binary sequence, supporting rank in $O(1)$ time and select in $O(\log n)$ time. The extra space is used to store rank values every $s = 20 \cdot 32 = 640$ bits, plus precomputed tables containing solutions for smallest cases. Using this information, rank is computed by searching the closest sample position and scanning byte-wise the last 640 bits using precomputed tables, while for select, a binary search over the sampled values is performed, finishing with a sequential scan as well. Given that the binary sequence is also stored, the operation access is direct.

While several succinct and compressed data structures solutions had been presented [RRR02, OS07, CN08], for the purpose of this thesis, we only used the implementation explained above given its simplicity and low query time for the rank operation.
Arbitrary Sequences

When the alphabet size, $\sigma$, is greater than two symbols, the operations \textit{rank}, \textit{select} and \textit{access} can not be solved in the same way as before. To solve these operations another data structure is used.

A wavelet tree [GGV03,NM07] is a tree representation of a sequence $S$, that uses a binary split at each node to solve the operations \textit{rank}, \textit{select}, and \textit{access}. To do that we need to be able to transform the input sequence into a collection of binary sequences. The root of the tree is represented as a binary sequence of the same length as $S$, where each symbol belonging to the first half of the alphabet of $S$ is represented by a zero bit, and each symbol in the second half of the alphabet by a one bit. If a symbol is marked with a zero bit, then it is allocated to the left node of the root, otherwise it is represented in the right node. This process is repeated recursively for each node until no further division is possible, resulting in a tree with $\sigma$ leaves (note that the leaves do not need to be stored) that requires $n \cdot \lceil \log \sigma \rceil$ bits (a total of $n$ bits per level), where $n$ is the length of the original sequence. Figure 2.5 shows an example of a wavelet tree.

In order to support the three basic operations, every binary sequence in the tree must be capable of answering binary \textit{access}, \textit{rank} and \textit{select} queries. In the overall sequence, to access the value of a given position $i$, at the root we check the bit value at that position, also computing the rank of that bit. Then we descend to the left or right child depending on the bit value at the position (in our example, zero to the left child and one to the right child), repeating the same process in the next node but using as position the value obtained from the first \textit{rank} operation. This continues recursively until reaching a leaf and reporting the corresponding symbol. The \textit{rank} operation for a symbol $s$ up to a position $i$ is computed a similar way as \textit{access}, where instead of considering the bit at position $i$ we consider the bit value of the symbol $s$ at the given tree level. That is, for the example in Figure 2.5 when $\text{Rank}_A(S,6)$ is computed, the first step is to get the bit value at position 6, then, observe that it is a zero, and compute the \textit{rank} of zero until position 6 in the root binary sequence. The go into the left child and repeat the same process over that
Figure 2.5: The wavelet tree of the sequence “CABRA_ABRACADABRA”. The white space is written as an underscore for clarity, and it is lexicographically smaller than the characters “A” to “Z”.

Level but using as parameter position the result of the \( \text{rank} \) of zero operation. This continues recursively until reaching the leaves where the binary sequence position will correspond to the desired answer. Algorithm 6 shows the pseudocode of the \( \text{rank} \) operation, where \( \text{Rank}_x(B, i) \), with \( x \in \{0, 1\} \), returns the number of \( x \) in \( B \) until position \( i \).

The \( \text{select} \) operation does the reverse, starting at the bottom of the tree. To select the \( i \)th occurrence of symbol \( s \), we start at the leaf where \( s \) is represented going to its parent node and computing the selecting the \( i \)th \( b \), where \( b \) is the bit of \( s \) corresponding to that level. We move to the parent and run select using the position obtained and the bit value corresponding \( s \) in this level, and so on. At the root, the value of the position found is returned.
Algorithm 6 \( \text{Rank}_s(B,A,i) \) takes a binary sequence \( B \), an alphabet \( A \), a symbol \( s \), and a position \( i < |B| \) as input, where \( B \) is the binary sequence representing the current node visited. This function returns the number of occurrences of the symbol \( s \) in the sequence \( S \), used to create the wavelet tree, until the position \( i \). We use \( \{B_l,A_l\} \) and \( \{B_r,A_r\} \) to refer to the binary sequences and alphabet of the left and right child of the current node, and \( \text{Rank}_x(B,i) \), with \( x \in \{0,1\} \), returns the number of \( x \) in \( B \) until position \( i \).

1: if \(|A| = 1\) then
2: \quad return \( i \)
3: else
4: \quad if \( s \in A_l \) then
5: \quad \quad return \( \text{Rank}_s(B_l,A_l,\text{Rank}_0(B,i)) \)
6: \quad else
7: \quad \quad return \( \text{Rank}_s(B_r,A_r,\text{Rank}_1(B,i)) \)
8: \quad end if
9: end if

2.5.2 The Burrows-Wheeler Transform (BWT)

As was already noted, an important concept is that of a self-index. In general a self-index exploits repeated patterns in the stored data, creating easy access points to the information, and compressing data allowing the entire original file to be regenerated if required. In this work, given its common use in Bioinformatics and being the base data structure of the traditional compression tools including \textsc{bzip2}, we explain the Burrows-Wheeler Transform (BWT) [BW94].

To compute the BWT of a given a text \( T \) of length \( n \), initially we need to compute an imaginary matrix \( M_T \) of dimensions \( n \cdot n \), whose rows are the rotations (cyclic shifts) of \( T \) in incremental order. Then we sort the rows of \( M_T \) in lexicographic order. The last column (\( L \) array) of the resulting matrix is the BWT of \( T \). Figure 2.6 shows an example of how the values of the BWT are obtained.

Assuming that the final symbol of \( T \) is unique and lexicographically smaller than all the others, it is possible to reverse the transformation for all the positions of the text. The basic case is \( T[n - 1] \) which is located at \( L[1] \), since the first element in the sorted matrix starts with \( T[n] \), which is smaller than every other.
symbol. To reverse the other positions of $T$ from $L$, we first need a way to move from the last column ($L$) to the value in the front column ($F$) of $M_T$. The BWT defines the function $LF(i) = C[L[i]] + \text{Rank}_{L,\Sigma}(L, \Sigma, i)$, where $C[s]$ is the number of symbols lexicographically smaller than $s$ in $T$ and $\Sigma$ is the alphabet of $T$. The value of $C[L[i]]$ gives the position in $F$ where the occurrences of the symbol $L[i]$ begins, and $\text{Rank}_{L,\Sigma}(L, \Sigma, i)$ gives the offset for the particular occurrence of the symbol $L[i]$ in the $L$ array. The $LF$ mapping allows us to navigate $T$ backwards, $T[n - 2] = L[LF(1)]$, and for any position $k$, $T[n - k] = L[LF^{k-1}(1)]$.

The self-index consists of the array $C$ plus a representation of $L$ which supports rank over the symbols. In general, $C$ is stored as an array containing the alphabet in lexicographic order plus a binary sequence supporting rank operations over $F$, where 1 indicates that a new symbol is found, otherwise the value is 0. The array $L$ is represented using a wavelet tree.
Algorithm 7 \textit{CountOcc}(P, m) takes a pattern $P$ and its length $m$ as input, and returns the number of occurrences of $P$ in a text $T$ assuming that the BWT of $T$ was previously created, and the alphabet $\Sigma$ of $T$ is known.

\begin{algorithm}
\begin{algorithmic}
\STATE 1: $i \leftarrow m - 1$
\STATE 2: $sp \leftarrow 1$
\STATE 3: $ep \leftarrow n - 1$
\WHILE{$sp \leq ep$ and $i \geq 0$}
\STATE 4: $c \leftarrow P[i]$
\STATE 5: $sp \leftarrow C[c] + \text{Rank}_c(L, \Sigma, sp - 1) + 1$
\STATE 6: $ep \leftarrow C[c] + \text{Rank}_c(L, \Sigma, ep)$
\STATE 7: $i \leftarrow i - 1$
\ENDWHILE
\IF{$ep < sp$}
\STATE 8: \textbf{return} 0
\ELSE
\STATE 9: \textbf{return} $ep - sp + 1$
\ENDIF
\end{algorithmic}
\end{algorithm}

One of the advantages of the BWT is that it can be used for searching patterns, using a method called \textit{backwards search}, which is implemented by the family of indexes known as FM-indexes [NM07]. Algorithm 7 shows the pseudocode for counting the occurrences of a pattern $P$ in a text $T$ using the BWT. There are several variants of the FM-Index [FM00,FMMN04,MN05,FMMN07,NM07,FGNV09] which offer different trade-offs between space used to store the structure and query time. Despite their great elegance and considerable usefulness, we refrain from further discussion of FM-Indexes, as we do not discuss pattern search in genomic data in this thesis.

## 2.6 General Purpose Compression Tools

We make use of several well known file compression tools, which are used as part of the benchmark of compression on the experiments of this thesis. More specifically,
we use two off such compressors, BZIP2\textsuperscript{4} and GZIP\textsuperscript{5}, given that their compression techniques are most frequently used as base or part of Bioinformatics compression tools [LD09, LHW\textsuperscript{+}09, TLS10, CBJR12, PvH13].

In this thesis we made use of GZIP and BZIP2 to compare different compression approaches against general purpose compressors. The GZIP compression technique, as we will present in this thesis, has been used as part of other compression approaches given that it offers fast compression and decompression of large amounts of data, and its memory usage is low. Meanwhile BZIP2, in general, provides better compression ratios than GZIP, but takes longer to compress and decompress the data.

There are many other general purpose compression tools (for example xz, LZMA, and LZ77), all offering full compression and decompression of files and different trade-offs between processing time and compression levels. GZIP and BZIP2 methodologies have been used successfully in many Bioinformatics studies [BS02, LHW\textsuperscript{+}09, LD09, PvH13] and so we use these two in our work.

### 2.6.1 Gzip

GZIP is one of the most commonly known compressor tools, and is based on a combination of two techniques, Huffman codes (Section 2.4.1) and LZ77.

The LZ77 [LZ77] algorithm is based in the following idea: if some part of the text to be compressed is repeated, it is only encoded the first time that it is found, as any later appearances can be referenced back to that earlier point. A way to do this, given a text $T$, is moving sequentially over the text, and whenever we arrive to a position $i$, search for the longest prefix in $T[i \ldots n - 1]$ commencing in $T[0 \ldots i - 1]$, returning the length of the match and a reference to where the match was found, encoding the information depending of their values. Some complications arise with the algorithm: if the match is too small or not found, it is possible that storing the

\textsuperscript{4}http://www.bzip.org/, September 2014
\textsuperscript{5}http://www.gzip.org/, September 2014
reference and length uses more space than storing the sequence; and, if the text is too big, it is too expensive to search over every time a new position is visited.

LZ77 uses a fixed-size window approach over the text. Given a window size $w$, it searches for the longest match within $T[i-w \ldots i-1]$, instead of using the complete text, meaning that the maximum length of a match is $w$. In LZ77, if there is no match, it outputs a null-pointer and the character at the coding position, else, if a match is found, it outputs a pointer to the match stored and the first symbol that didn’t match. The focus of activity then moves $l + 1$ positions in the text, where $l$ is the length of the last match found, and the algorithm repeats from $i + l + 1$, continuing until the end of the text is reached. In GZIP the window size is limited to 32 kilobytes, and the lengths of the match founded are limited to 258 bytes.

GZIP offers a time and space trade-off depending on its level parameter (1 to 9), which indicates the runtime limit used to find a match to a given string. So, depending on this parameter, the compress process does not always find the longest possible match but generally finds a match which is long enough.

### 2.6.2 Bzip2

BZIP2 is a generic purpose compressor, which divides the file to be compressed into blocks of size between 100 and 900 kilobytes, offering a time and space trade-off depending on the block size chosen. The bigger the block the smaller the size of the compressed file (generally), but the compression takes slightly more time. Once the block size is chosen, a BWT (Section 2.5.2) is used to reorganize the data in each block.

The BWT representation of each block is then compressed using several layers of compression techniques, such as those presented in the integer and symbols bit encoding sections (Section 2.3 and 2.4), applied to different parts one after another. More specifically, for each block, BZIP2 computes its BWT, without altering the size of the processed block, storing the $L$ column (Section 2.5.2), and then applying move to front (MTF) [Rya80,BSTW86] to $L$. The MTF algorithm maintains a list
of the symbols in the alphabet (if the alphabet is not known, the alphabet contains all possible symbols). For each symbol in the sequence, it encodes the position of this symbol in the list, and then modifies the list by moving the symbol to the front of the list. Then immediately recurring symbols are replaced by the index zero, while other symbols are remapped according to their local frequency. After transforming $L$ using MTF, bzip2 uses run-length encoding over the generated sequence, where strings of repeated symbols in the sequence are replaced by the symbol followed by the length of the repetitions. Finally, the sequence is Huffman coded.

To decompress, bzip2 follows the reverse order of the algorithm just presented. In general, while bzip2 offers a good compression of files, it is a slow compression scheme, given that it needs to suffix-sort each block while computing the BWT.

2.7 Multi-Alignment Genomic Data

As the title of this thesis suggests, our work is focused on genomic files. More specifically, genomic data containing DNA (deoxyribonucleic acid) information. In this section we introduce the basic concepts of Bioinformatics necessary to the understanding of the work presented in this thesis.

2.7.1 DNA Sequence

The DNA of a living organisms carries the genetic information and instructions describing that form of life. It is a long chain of nucleotides (large biological molecules), composed of four fundamental bases Adenine (A), Cytosine (C), Guanine (G), and Thymine (T).

In general, a DNA molecule is formed by the union of two DNA sequences giving a double helix structure [WC53], resembling a spiral staircase, which gives to the DNA molecules, the property of being double stranded. The union at the level of bases, is not random, allowing only Adenine and Thymine to join to each other and the same applying between Cytosine and Guanine. Normally for studying a DNA
molecule, it is sufficient to have just one of the strands, given that the information contained by the other strand is the same but reversed (because, each strand is read in different direction) and complemented (because the bases contained in one strand are the complement of the other strand). For example, if one strand is “GATCGTA” the corresponding reverse complement is “TACGATC”.

The DNA sequence of a living organism is packed into structures called chromosomes, which form the genome of the organism. Each chromosome identifies a section of the genome of an organism and the recognition of this region helps studies of the DNA’s properties and functionalities. Some organisms have copies of each chromosome that, in general, are inherited. For example in the human genome, each chromosome is repeated twice, called a pair chromosome, where each part is from each of the parents. More precisely the human DNA has 46 chromosomes: 1 pair of sex chromosomes (containing some genetic traits); and 22 pairs of autosomes chromosomes (containing the rest of the genetic hereditary information).

The set \( \{A,C,G,T\} \) is the standard nucleotide base alphabet for DNA sequences, but, given that, in general, the bases are determined by laboratory experiments which can contain errors, this alphabet is extended. In cases where the value of a base is ambiguous, the alphabet includes symbols that represent all possible combinations of the four nucleotide bases. That is, symbols to represent \( \{A \text{ or } C, \ C \text{ or } T, \ A \text{ or } G \text{ or } T, \ldots\} \) can also be used. The rules of how these ambiguities are recorded, were presented by the International Union of Pure and Applied Chemistry (IUPAC) [CB85]. For the purpose of this thesis we assumed that the DNA sequences are only formed by standard nucleotide bases, with the inclusion of the letter \( N \), which is used to represent a nontemplated nucleotide (that is, a base that could take any value). Finally a template is a DNA sequence which is used to construct and obtain other sequences, normally of smaller length.

### 2.7.2 Genome Sequencing

Sequencing is the process of determining the order of the nucleotide bases in a molecule of DNA, allowing us to discover and record the substance holding the
Chapter 2 Background 2.7 Multi-Alignment Genomic Data

genetic makeup information of living organisms. Over the last few decades the mechanisms for DNA sequencing have evolved at an astonishing rate. These methods are generically referred to as being next-generation sequencing technologies [Chu06, Mar08, Ans09, MBJ12], parallelizing the sequencing process, and producing millions of sequences at once.

Sanger et al. [SSNC77] presented the first method for DNA sequencing. The main steps consist of cutting the sequence into small fragments, cloning each fragment multiple times, then labeling each fragment in some way (normally with chemical fluorescence), and pairing them with (thousands to millions) of short known DNA sequences to detect and record the nucleotides in the fragments. The result of this process is thousands to millions of short fragments, called reads, which are reassembled to form a contiguous sequence that represents the DNA genome.

Detecting the position of a read is not an easy task. For example with complex genomes, where the abundance of repetitive sequences is high, similar reads might have come from completely different parts of the genome, meaning that the sequencing process sometimes assigns incorrect positions to reads. Zhang et al. [ZCY+11] summarize existing genome assembly software tools.

Next-generation sequencing follows, in essence, the same broad steps, the main difference being that the techniques are parallelized. This considerably speeds up the process, but at the price that the number of errors in short reads may increase. Due to the rapid technological advancements, the platforms that uses next-generation sequencing (for example, Illumina\(^6\)) output large quantities of sequence data compared to the Sanger sequencing methodology, at a much lower cost.

2.7.3 Sequence Alignment

A concept that is highly related to the sequencing technologies is sequence alignment, which is defined as “arranging the sequences of DNA, to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships

\(^6\)http://www.illumina.com/, September 2014
between the underlying organisms the sequence were obtained from” [Mou04]. In this thesis we will refer to an aligned read as the DNA sequence that has been used as input during the sequence alignment process. The result of aligning two sequences is a list of operations over one of the sequence, which include match, substitute (change the base), and indels (insert a set of bases or delete a number of bases). Figure 2.7 shows an example of the alignment that has been identified between two reads.

The process of aligning two sequences offers multiple possible solutions, but in practice each operation is associated with a cost. The sequence alignment process finds a combination of operations that minimizes the cost. This process can easily been done by hand when we are presented with small problems such as the example in Figure 2.7, but in general, sequence alignment involves many cases where the sequences are of different length, highly variable (many substitutions and/or indels), and/or several sequences are aligned at the same time. Normally the sequence alignment tools use a global, local, or mixed alignment approach.

Global alignment [NW70] aligns the full length of each sequence, while the local alignment [SW81] detects and aligns only the regions where the sequences are related. The global approach is preferable when whole sequences are expected to be related and of similar size. Local alignment is often used to find similarity between short reads and a long reference sequence.

In the case of multi-alignment sequences, where many short reads are to be aligned against a long reference, semi-global alignment is preferred. A semi-global
alignment is a mix of the global and local alignment approaches, and aligns the short sequences to all possible places using, depending on the tool used, different algorithms. For example, the BWA (Burrows-Wheeler Aligner) tool [LD09] is a software package based on the BWT (Section 2.5.2), which uses a semi-global approach to align short variable sequences against a large reference genome.

An important concept for this thesis, related to multi-alignment sequences, is the coverage of a sequence. Coverage is defined as the average number of reads aligned over the same nucleotide in the reference sequence. In multi-alignment sequences, a high coverage is desirable, so errors can be overcome or be detected more easily given the high number of reads including the representation of each base. The coverage is computed as

\[ \text{coverage} = \frac{N \cdot L}{G}, \]

where \( N \) is the total number of reads, \( L \) the average read length, and \( G \) is the length of the reference sequence used in the alignment process.

2.8 Genomic Data Formats

Several standard formats for storing alignment reads have been adopted in Bioinformatics during the last decade. These formats aim to facilitate the manipulation and parsing of data using text processing tools such as Perl and Python. Most of them add metadata, which is used for more complex analysis. For the purpose of this work, three formats are presented: FASTA, FASTQ, and SAM.

2.8.1 FASTA format

The FASTA format is one of the most basic formats for storing alignment reads. It was originally invented as input to a software package [LP85, LP88] which was designed for genomic sequence similarity search. The name of the software package was also FASTA meaning “FAST-All”, because it works with protein and nucleotide
information. Nowadays the FASTA format is considered a standard format in the field of Bioinformatics.

Files in FASTA format are a plain text file used to represent either nucleotide sequences (over the extended alphabet for DNA) or protein sequences (over an extended alphabet of around 20 to 25 different bases) as a set of lines each no longer than 120 characters.

A FASTA file includes optional comments about the sequences stored. These comment lines appear at the beginning of the file and start with the symbol ‘;’. The rest of the file contains the alignment reads, where each alignment read is represented by a single-line description, followed by lines of sequence data. Description lines start with the symbol ‘>’ followed by the identifier of the sequence and optional extra description of the sequence. Figure 2.8 shows an example of a FASTA file, containing two description lines.

### 2.8.2 FASTQ format

Originally the FASTQ format [CFG+10] was developed at the Wellcome Trust Sanger Institute to add a Quality field to the FASTA format, and was formally presented in 2009 (but had been in use before that date). The quality score component is a string of a length equal to the length of the read component.

```
;This is a comment line
>Sequence|read 1|2012-10-01
AACCCCTAAACCTAATCCCTGAACCCCTGAAACCCTAAACCCTAAACCCTGAACCCCC
>Sequence|read 2|2012-10-01
TTTACGGGTTCAAGGTTCAAGGTTCAAGGGTATTAGGGGTCAGGTTTAGGTTC
...
```

Figure 2.8: Example of a FASTA format file.
Each position of the string stores the quality of the respective base in the sequence component. Quality is defined in detail in Section 3.2.

A FastQ file contains four lines per alignment read. The first line begins with the symbol ‘@’ followed by a sequence identifier and an optional description. The second line contains the raw sequence. The third line begins with the symbol ‘+’ optionally followed by an identifier and more description, and the fourth line stores the quality scores of the sequence. Figure 2.9 shows an example of a FastQ file containing two alignment reads.

FastQ was rapidly adopted as a standard format for short reads given that it included the quality score data, making it the preferable storage format of high throughput sequencing instruments. In the last four years FastQ files are starting to be overtaken by a new format which we will present shortly.

2.8.3 CIGAR

In Section 2.7.3 we briefly presented the output operations obtained from the sequence alignment process, without formally defining how it is recorded. These output operations between a reference sequence and a read are called CIGAR (Compact Idiosyncratic Gapped Alignment Report) strings, and describe an edit string that transforms one sequence into another. The standard CIGAR description
Figure 2.10: Example of a CIGAR string between two sequences. The symbol ‘*’ indicates that the position should not be counted as part of the sequence. The CIGAR string in this example indicates that the first 8 nucleotides of Sequence 1 match with the corresponding items in Sequence 2; that it is necessary to insert ‘AG’ to Sequence 1; that the next 4 nucleotides match; that the nucleotide ‘G’ should be deleted from Sequence 1; and finally that the last 3 nucleotides match.

of pairwise alignment defines three operations: ‘M’ for match/mismatch, ‘I’ for insertion compared to the reference and ‘D’ for deletion. Figure 2.10 shows an example of a CIGAR string.

Some genome data formats include the CIGAR string; for example as part of the optional description in FASTQ files. The CIGAR string represents a sequence relative to a reference, but the information stored is not enough to replace the sequence. For example, the operation ‘M’ is ambiguous, being impossible to tell if a base is equal or different to the referenced base.

In the last few years a new genome data format has arisen as the standard output of next-generation technologies, the SAM format. This format includes, as part of its main information fields, an enhanced CIGAR string, adding four more operations: ‘N’ for skipped symbols on the reference, ‘S’ for soft clipping (unaligned symbols), ‘H’ for hard clipping (part of the aligned sequence that is not present in the sequence field), and ‘P’ for padding (aligns with the inserted sequences, but is not compulsory). While this information is still not enough to allow complete reproduction of the aligned sequence, it adds information that can be used in downstream applications. An example is the operation ‘S’, which appears in reads in which certain parts may be unaligned to the reference. The identification of these reads helps, in addition to other things, the study of diseases [SYS+11, SHB+14].
2.8.4 SAM format

The *Sequence Alignment/Map* (SAM) format [LHW+09] is a more comprehensive format than FASTQ, normally generated as output from software alignment tools, which read FASTQ files as input, and compute their association with a known reference genome. This format has become one of the standard output formats of all sequence alignment technologies.

SAM format data is stored as a multi-line, TAB-delimited, plain text file that contain two kinds of lines: header lines; and alignment lines. The header lines are optional, and contain commentary, information about the reference sequence used for the alignment, the program used to generate the alignments, and the read groups (an identifier for reads that are together for some purpose). Header lines, if they are present, are found at the beginning of the SAM file, and start with '@'. That flag is then followed by two letters that define the information that is contained in that header line.

Each alignment line contains 11 or more fields, where the first 11 must be present, but might be ‘*’ or ‘0’, meaning that the information is not supplied. The order in which these fields appear in each line is always the same. Table 2.3 gives an overview of the fields.

The QUAL field stores quality scores, as in the FASTQ format, of the sequence stored in the SEQ field. The probability that the base in a given position is wrong can be computed in different ways and depends on which program was used to generate the SAM file. Section 3.2 provides more details about quality scores.

Each read stored in the SEQ field may have been aligned against an external reference sequence, and, in the case that this sequence is known, its reference name is stored into the correspondingly RNAME field. If RNAME is ‘*’ the fields POS and CIGAR cannot be considered to be valid information. It is important to note that the reference sequence is not copied into the SAM file; instead, the description of reference sequences might be contained in the HEADER lines (lines preceded by the flag @SQ). This means that before any further analysis over the sequence...
<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QNAME</td>
<td>String</td>
<td>Query template name.</td>
</tr>
<tr>
<td>FLAG</td>
<td>Int</td>
<td>Bitwise flag (values between 1 and 1160), that give some properties of the alignment (for example, if the sequence is reverse complemented).</td>
</tr>
<tr>
<td>RNAME</td>
<td>String</td>
<td>Reference sequence name.</td>
</tr>
<tr>
<td>POS</td>
<td>Int</td>
<td>Leftmost mapping position of the first matching base.</td>
</tr>
<tr>
<td>MAPQ</td>
<td>Int</td>
<td>Mapping quality ($-10 \log_{10} Pr{\text{mapping position is wrong}}$).</td>
</tr>
<tr>
<td>CIGAR</td>
<td>String</td>
<td>The CIGAR string.</td>
</tr>
<tr>
<td>RNEXT</td>
<td>String</td>
<td>Reference sequence name of the next segment in the template.</td>
</tr>
<tr>
<td>PNEXT</td>
<td>Int</td>
<td>Position of the next segment in the template.</td>
</tr>
<tr>
<td>TLEN</td>
<td>Int</td>
<td>Signed observed template length.</td>
</tr>
<tr>
<td>SEQ</td>
<td>String</td>
<td>Sequence of nucleotides bases of the read used in the alignment.</td>
</tr>
<tr>
<td>QUAL</td>
<td>String</td>
<td>Quality string containing the error probability of each base ($-10 \log_{10} Pr{\text{base is wrong}} + 33$).</td>
</tr>
<tr>
<td>OTHER</td>
<td>String</td>
<td>Optional fields of the form TAG:TYPE:VALUE.</td>
</tr>
</tbody>
</table>

Table 2.3: Information fields of each read alignment in the SAM format. All values are stored using the printable ASCII character set.

alignment in a SAM file can be carried out, it may be necessary to obtain the reference sequences related to the SAM file.

Figure 2.11 shows an example of the SAM format. Reads r001/1 and r001/2 constitute a read pair (meaning that this read refers to the two ends of the same DNA molecule), r003 is a chimeric read (an experimental error wherein fragments from two different parts of the genome are combined together into a single read), and r004 represent a split alignment (an alignment where only some part of the read is aligned and the inter positions are not consider as part of the alignment). The ‘*’ means that the corresponding position should not be counted as part of the sequence, and a ‘.’ means that the position can have any value. Note that in this example the component QUAL is not provided, and is represented with a ‘*’ value. Note also that the presence of lower-case bases indicates uncertainty in the base stored, which are register as hard clipping or soft clipping in the CIGAR string (letters ‘H’ and ‘S’ respectively) and are replaced by uppercase bases if they are
Chapter 2 Background

2.8 Genomic Data Formats

Figure 2.11: (a) Example of a reference sequence and different reads. (b) The corresponding SAM format.\(^8\)

\(^8\)Derived from the image at \url{http://samtools.sourceforge.net/SAM1.pdf}
considered in the SAM file. The SAM project web page\footnote{http://samtools.sourceforge.net/, October 2014}, provides more details of SAM format and its fields.

To give an idea of the proportion of the space used by the various components in a SAM file, we extracted each of them into separate files, to measure the space used per component. We also compressed each component file using GZIP, which provides a sense of how repetitive and compressible they are. Figure 2.12 presents an example of the percentage of space occupied by each component before and after compressing the file using GZIP -9 (maximum compression possible using GZIP, see Section 2.6.1). All the components that use less than 10% of the total space, are grouped under the label of “Small Components”. The graphs were generated over one of the sample files that will be used in the experiments described later in this thesis.

From Figure 2.12 we can see that the bulk of the space consumption of the original SAM files generally arises in the fields SEQ, QUALS, and OTHER. While it is correct to assume that in the original file, SEQ and QUAL are the larger components in the files, we can not say the same about OTHER because, even though it seems to use most of the space in this SAM file, it is an optional field which is not always present in SAM files.

In this work we further study the SEQ and QUAL fields, as those two still occupy significant amount of space after compressing the SAM files (the OTHER field is highly repetitive and compresses well with GZIP). Furthermore, from Figure 2.12, we can assume that the space used by the remaining components, after compressing with GZIP, is small compared with the space used by SEQ and QUAL.

As we will show in the following chapters, the information stored in these two components is related. These relationships have been used to enhance the compression proving that in some cases it is better to treat components as a combination. Moreover, some researchers have showed that many components do not need to be stored, given that they can be partially or completely deduced from the rest of the file information. An example of this is the CIGAR string, which, if
Chapter 2 Background

2.9 Summary

This chapter introduced the basic compression modalities, lossless, information preserving, and lossy, that are used in this thesis. We also described and discussed different variable length encoding techniques used to represent integer and symbols, and we briefly presented the concept of succinct data structure and self-indexes.
that compress the input, and enable queries such as random access to be performed on the compressed input.

Additionally, in this chapter we introduced the basic terminology of DNA, furthermore explaining what information is stored into multi-alignment genomic data files, and how this information is generated and stored. Finally we presented details of the SAM format, which stores aligned reads including extra macro data, or information fields, for each aligned read.
Chapter 3

Storing Multi-Alignment Files

Several researches have developed techniques to compress multi-alignment files, however few of them include in their compression approaches all the fields of the SAM format, currently containing the most information about the aligned reads (see Section 2.8.4). Many focus exclusively on how to compress DNA sequences, while others focus on how to compress FASTQ files and their components. In this chapter we review the related work, separating the presentation between those approaches that compress DNA reads and those that focus on the compression of quality scores. As was shown at the end of Section 2.8.4, these two fields generally occupy more space after compressing the SAM file, also being influenced by a wide range of factors and, because some SAM fields can be derived from them, are essential components.

Section 3.1 presents the related work for DNA sequence compression, while Section 3.2 describes the techniques that have been developed for the Quality field. Section 3.3 includes information about existing attempts to fully compress SAM files, and Sections 3.4 and 3.5 briefly introduce downstream applications and operations that are of interest for SAM files and their compressed formats.
3.1 DNA Sequence Compression

Assuming that DNA sequences are formed only by the standard nucleotide bases (A, C, G, and T), it is easy to encode each base using a 2 bit encoding (for example: A → 00, C → 01, G → 10, and T → 11). However, DNA sequences are sometimes highly repetitive and, depending on how they are stored and generated, may present some attributes that make further compression possible. In this section we describe some of the existing compression approaches concerning DNA sequences when they are stored as short reads obtained from the multi-alignment sequence process (Section 2.7.3).

As we will demonstrate, most of the approaches assume the existence of a known reference DNA sequence, available as an external component, which is not considered part of the information to be compressed. Note that DNA compression methodologies followed by approaches that fully compress SAM format files, are discussed separately on Section 3.3.

3.1.1 Human Genome Compression

In 2009, Christley et al. [CLLX09] considered a series of techniques to reduce the size of storing human genome sequences, given a reference human genome. Their techniques, based on the assumption that two human genomes differ by at most one per cent, proposed to store only the variations between the sequences and a given human genome reference sequence. These variations store the position where the event occurred plus an operation to be performed over the reference sequence. Christley et al. presented three possible operations: substitution (S), which store the base that replaces the one at the given position in the reference sequence; insert (I), which store the sequence of bases to be added at the event position; and delete (D), which store the number of bases that are to be erased from the reference sequence. Figure 3.1 illustrates an example of these variations, where Sequence 1 is stored using the following instructions: substituting the base G to C at position 9; insert “AG” after position 13; and, delete 2 bases after the position 17.
Chapter 3 Storing Multi-Alignment Files

3.1 DNA Sequence Compression

Figure 3.1: Example of Christley et al. [CLLX09] variations.

Different approaches of how to store these variations were explored [CLLX09], where all of them consider the use of VBYTE code (see Section 2.3.5) to minimize the number of bytes used to represent a number. VBYTE is a variation of the P-VBYTE encoding discussed in Section 2.3, in which the block size used is \( b = 7 \), having chunks of one byte size, speeding the decoding of the encoded values. The drawback with Christley et al.’s approach, as described so far, is that the position value where variation might be found, it can could take any value between zero to the length of the human genome used for the reference (for example, if the variations are found at the end of the reference). Given that the position variation distribution is not know in advance, it is desirable to represent the small positions using lower space than that needed for the largest ones, hence the VBYTE encoding methodology chosen.

Three methodologies were proposed by Christley et al. to store the positions. The first approach uses VBYTE to encode the positions for all operations, plus, in the case of an insert or substitution, two bits for each base stored, and, in case of deletion, recording the length to be deleted also using VBYTE. The second approach proposed storing each position as a relative position (\( \Delta \)) from the position of the previous variation, instead of storing the absolute positions. Finally, a third approach separated the sequence of bases to be inserted into \( k \)-mers (sequence of length \( k \)), and encode them using a Huffman code.

While Christley et al.’s work shows that it is possible compress the variations, it is still necessary to refer to a reference sequence at the time of compression and decompression. Also, Christley et al.’s approach focuses only on storing
3.1 DNA Sequence Compression

Chapter 3 Storing Multi-Alignment Files

the variations between an input human genome and a known reference human genome; and its methodologies cannot be readily applied to multi-alignment data files. Variations on the ideas, however, have been used in other research on multi-alignment data [KSK+10].

3.1.2 SLIMGENE Tool Sequences

In 2010, Kozanitis et al. [KSK+10] presented the SLIMGENE tool, which compresses genomic sequence reads using, as part of the input information, a reference sequence. Given that the compression is done over a reference sequence, as in Christley et al., only the difference between the reads and the reference sequence need be stored.

The difference between SLIMGENE and Christley et al.’s approach is that the data to be stored consists of a set of short reads, instead of a complete human genome, thus it is important to know where each read begins with respect to the reference used. In this case a position bit sequence of the same size as the reference sequence is stored, where position \( i \) is one, if at least one read started at that position, and zero otherwise. If all the reads match with the reference without error, then the position bit vector would be enough to store all the reads (assuming that all the reads are of the same size), in addition to a count of the number of reads that start at the same position for each marked position. Nevertheless, in general, many of the reads differ from the reference at some position.

Kozanitis et al. proposed solution to store the read information, consist in a refined vector, representing each read individually in position order. A read is stored as a prelude of three bits followed by a variable-length bit sequence storing the error information. Figure 3.2 shows an example of the components stored using the SLIMGENE tool.

The first bit of the prelude has the same bit value as in the previous read if both of the reads start at the same position, otherwise the first bit changes value, in other words, the equivalent of a unary coded counter. The second bit is one if the
Figure 3.2: Example compressed sequence components over three fragments and a given reference sequence using the SLIMGENE tool.

read aligns with the forward strand, while the last bit is one if the alignment does not contain errors against the reference. In the case that the last bit of the prelude is one, the error information is empty. For example, in Figure 3.2, Read 3 changes value of its first bit in the prelude, given that its start position differs from Read 2, and we also can notice that Read 2 does not contain errors, indicated by the third bit of its prelude bit sequence.

The errors are reported after the prelude, where, for each error, an extra bit is included at the beginning of the information indicating if it is the last error in the read (equivalent to coding the number of errors in unary). The error information consists of a local offset, the number of bases since the last local error, and a description of the error. The description of the error outlines the base substitution using two bits that specify the distance between the reference and the substitute base in a circular chain ($A \rightarrow C \rightarrow G \rightarrow T \rightarrow A$), reserving the 00 code to indicate other kind of errors such as insertions, deletions, or the occurrence of symbols different to the standard bases. Special errors are coded using Huffman codes.

While Kozanitis et al.’s approach does not support random access to the stored data, it would be possible to support this operation creating an index containing
pointers to the information stored, given that their techniques independently encode each read.

### 3.1.3 Block and SuperBlock DNA Compression

One of the first approaches that considered random access to the compressed read sequence data was the work of Deorowicz and Grabowski [DG11]. The algorithm compresses FASTQ files by dividing the file into blocks of read alignments, where each block represents $b$ reads and their respective quality scores. Then the blocks are grouped into sets, or superblocks, containing $s$ blocks each set. The parameters $b$ and $s$ are predefined by the user, and the blocks and superblock are compressed.

Each superblock contains a header which precedes the block data in the superblock. The header contains statistical information about the blocks within each superblock. These include, the number of block inside the superblock (which would be always $s$ with the possible exception of the last superblock in the file), a flag indicating if the read lengths are all the same or not, the number of DNA symbols present in the reads, and so on.

Finally, each superblock is compressed independently, and the blocks inside a superblock are also independently compressed, but using and sharing the statistical information stored in the header of the superblocks. A block contains $b$ reads plus its respective quality scores, but the reads and quality scores are compressed separately. Their quality score compression approach is explained in Section 3.2.2.

To compress the DNA sequences Deorowicz and Grabowski pack the bases in bytes, and, depending on the information stored in the blocks, apply different compression schemes. In the rare case that the sequences covered by a superblock contains at least one symbol that is not part of the allowed DNA alphabet, they completely Huffman encode (see Section 2.4.1) the superblock.

Deorowicz and Grabowski took advantage of the observation that stored reads generally overlap, which suggests encoding the superblocks using a $LZ77$-style [KPZ10,LZ77] (see Section 2.6.1) approach. This structure keeps a record of text
already read, and an index to it. When a new fragment of the text is read, fast
match search can be accomplished using the index to the records. In their case, the
index encodes the matches using a hash over read subsequences of length 36, arguing
that for shorter matches it is cheaper to encode the literals one by one, rather than
encode a match.

In the Deorowicz and Grabowski approach it would be possible to random access
the information of each superblock. Their work suggested the creation of an index
to the information stored in each superblock, with the purpose of allowing random
access to the data, but no further studies were presented.

### 3.1.4 ASCII DNA Compression

Priyanka and Goel [PG14] propose a simple algorithm to compress a DNA sequence
assuming that it only contains the standard four bases (A, C, G, and T). Their
approach consisted of a lossless two phase compression over the DNA bases, reducing
the storage space used in comparison when the DNA sequence is compressed using
two bits per base. In contrast to many of the DNA compression techniques already
discussed, Priyanka and Goel compressed the sequences without considering any
external reference or biological knowledge over the input file, hence it is possible to
apply their algorithm to any data text with an alphabet of size four.

The first compression phase applies runlength encoding; that is, represent the
values of the sequence as a string of values and a count, where subsequences
of identical values are encoded as a single value followed by the length of
the subsequence. For example, the sequence “AAAAAGGTCC” is encoded
as “4A2G1T2C”. Using this approach Priyanka and Goel store the runlength
information in three different files: a file containing the length of the runs (count
file); a file storing only the symbols of each run (data file); and a file containing the
position where each run starts in the file that store only the symbols (index file).
Runs of length lower than three were not encoded as runlength, instead storing
the values in the data file without recording their count nor index information.
Figure 3.3 (a) shows an example of this first phase and the output files obtained.
3.1 DNA Sequence Compression

Chapter 3 Storing Multi-Alignment Files

a)

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Data File</th>
<th>Count File</th>
<th>Index File</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTTACATAAGGAAAATG</td>
<td>0 1 2 3 4 5 6 7 8 9 0</td>
<td>3 3 4</td>
<td>0 5 8</td>
</tr>
</tbody>
</table>

Figure 3.3: (a) Example of the three output files (data, count, and index) generated by the Priyanka and Goel approach after applying their first phase methodology to the sequence “TTTACATAAGGAAAATG”. (b) Example of the second phase of the data file, where the bases are separate in groups of 4, representing each base with 2 bits and a group with the corresponding ASCII value.

The second phase of the Priyanka and Goel compression algorithm takes the three files generated in the previous phase and encodes the data stored, by transforming the data into ASCII values\(^1\). There are 256 ASCII values, and each of them is represented using 8 bits. In order to transform the information stored in the files, they followed different approaches for each case. The data file encodes each base using 2 bits, and every 8 bits (4 bases) they store the corresponding ASCII value. The count file stores the corresponding ASCII value of the counts stored, assuming that the counts are always lower than 256, otherwise the count is divided in two (supposing that it is not possible to find runlength longer than 512), adding a base into the data file and another position into the index file. While Priyanka and Goel propose to store the index file by transforming the positions into its binary representation and then store the corresponding ASCII values, they do not describe how the original positions are decoded. Figure 3.3 (b) shows an example of the

\(^1\)http://www.asciitable.com/
second phase applied over the example data file.

While Priyanka and Goel approach offer a valid compression for DNA sequences containing 4 bases, normally DNA sequences include special characters making this approach impractical. Also, this algorithm completely depends on the size and number of runlengths within the input DNA sequence, which in general are not long runs. Moreover, in the case of multi-alignment genomic data files, which contain many short reads with many repetitions, this method fails to exploit between-read similarities which can improve compression. Finally, this methodology does not support random access to the compressed data, forcing the user to decompress the whole file before any further analysis.

### 3.1.5 ORCOM

Recently, Grabowski et al. [SGR15] described ORCOM (Overlapping Read COmpression with Minimizer), a new compression method for the Read Sequences field of FastQ files. The methodology proposed by Grabowski et al. separates the reads into different “bins”, given a defined classification, and then processed each bin independently. The classification applied by ORCOM detects similar (overlapping) reads using the idea that “two reads which largely overlap, are likely to share the same minimizer”. The minimizer [RHH⁺04] of a read of length \( l \), given a user-defined parameter \( k \), is a lexicographical smaller \( k \)-mer in the read, where \( k \ll l \). The chosen minimizer of a read is called its signature.

Once the signatures of all reads have been computed, ORCOM sorts the reads in lexicographical order considering the reads as circular strings (the end and start of the read are connected) and the start comparison position for each read is where its signature is found. Grabowski et al. argue that by reordering the reads in this way, highly overlapping reads are close to each other in the file.

The last phase consist of compressing each bin. In order to do this, ORCOM uses a sliding window technique where, given a parameter \( m \), it keeps the information of the last \( m \) reads and the start positions of their respective signatures. For each
read ORCOM finds the read from the current window that offers the lowest number of differences with respect the analyzed read. Then the reads are stored using the following stream of data information.

- **Flags**: Indicates if the read overlap with any read from the current window information.
- **Lengths**: Contains the length of each read.
- **LetterX**: Stores the mismatches symbols between the read and the reference chosen from the window information, when the base in the reference is $X (\in \{A,C,G,T,N\})$.
- **Prev**: Indicates the location of the read used as reference.
- **Shift**: Stores the offset between the read and the reference used.
- **Matches**: Contains the positions of the mismatches in the current read.
- **HReads**: Stores the reads that are not similar to any of the reads in the current window information.
- **Rev**: Indicates if the read was processed directly or as its reverse-complement.

Finally each of the streams of data stored is compressed using general purpose compressors.

Grabowski et al.’s methodology reorders the reads, demonstrating in their work that this reordering permits better compression of the reads, but without discussing how this could affect the possibility of offering random access to the data if it is desired.

### 3.1.6 Genomic SQueeZ (G-SQZ)

So far all the compression approaches reviewed in this section compressed the read sequences independently from their quality scores, making it impossible to exploit
any relationships between them. In 2010 Tembe et al. [TLS10] presented the G-SQZ tool, which losslessly compresses DNA reads and their associated quality scores using a Huffman coding-based approach. Their G-SQZ tool receives as input a FASTQ file (see Section 2.8.2), and compresses the data by treating each base in each read and its respective quality score as a single combined symbol \((\text{base, quality})\).

The methodology proposed by Tembe et al. computes the frequency of each \((\text{base, quality})\) symbol within the input file, then calculates a Huffman code (see Section 2.4.1) of these symbols using the computed frequencies, and finally encodes the symbols. Figure 3.4 illustrates an example of how G-SQZ tool works over a small artificial FASTQ file.

Given the simplicity of their approach and that the only information used is the frequency of \((\text{base, quality})\) symbols, random access to the information stored would
3.2 Quality Score Compression

Table 3.1: Example quality scores using Phred + 33. For example, if the quality score is the letter ‘F’, it is equivalent to a quality score of 70, meaning that the base at the same position in the read has an estimated 0.02% probability of being wrong.

<table>
<thead>
<tr>
<th>Letter</th>
<th>Quality value</th>
<th>Estimated error probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>40</td>
<td>20%</td>
</tr>
<tr>
<td>7</td>
<td>55</td>
<td>0.6%</td>
</tr>
<tr>
<td>F</td>
<td>70</td>
<td>0.02%</td>
</tr>
<tr>
<td>U</td>
<td>85</td>
<td>0.0006%</td>
</tr>
<tr>
<td>d</td>
<td>100</td>
<td>0.00002%</td>
</tr>
</tbody>
</table>

be possible without the need to decode the encoded data from the start to the end of the compressed file, if an index format to the encoded data was provided as extra information.

Tembe et al. described the first approach that combines the read and quality score information, showing that it is possible to obtain competitive compression ratios compared to compressing these fields using general purpose compressors (see Section 2.6); but their results are sub-optimal compared with approaches that work with each field separately [KSK+10,DG11,PvH13].

3.2 Quality Score Compression

Quality scores show how accurate the represented base values are with respect to the reference sequence used in the alignment process [EG98,EHWG98,Ric98]. The values symbolize quantized probability estimates, mapped to ASCII letters. More specifically, the quality scores are stored as a Phred-scaled base error probabilities [EG98], calculated as \([-10 \cdot \log_{10} Pr\{\text{base is wrong}\}]\). These are then scaled (Phred + 33 or Phred + 64) so that they can be displayed as printable ASCII characters. Table 3.1 shows examples for Phred + 33, the scheme assumed in the examples throughout this thesis.

The probability that a base in a given position is wrong can be computed in different ways, and depends on the hardware and software that are used in the
sequencer. While the exact details of how the quality scores are computed is not
the same in all the sequencers, it is accepted that is possible to introduce flexibility
in these values, allowing lossy compression and providing a trade-off between space
and information loss. Until recently it was not clear how well the quality scores
component of a genomic file could be compressed, given that typically they are
considered together with all other components of the file, avoiding an independent
analysis. Several recent approaches have shown that the quality scores can be highly
compressed using lossy compression.

In this section we explore the previous approaches used to compress quality
scores using lossless and lossy compression.

3.2.1 SLIMGENE Tool Qualities

One of the first approaches that considered separate compression of quality scores
was described in the SLIMGENE tool by Kozanitis et al. [KSK⁺10], previously
discussed in Section 3.1.2. Their approach is based on the observation that
neighboring quality scores, in general, have values close to each other. They also
work under the assumption that higher quality scores are most often found at the
beginning of the sequences, and lower values closer to the end. Using this premise,
they describe two lossless methodologies and a lossy approach for compressing
quality score sequences. Section 5.3 (page 125) presents an analysis over the SAM
files used in this thesis, showing that the premise previously mentioned, can not be
assumed for all quality scores files.

The first lossless approach encodes $\Delta$-values; that is, each quality score sequence
is transformed to a set of difference values between adjacent quality scores. This
set is then encoded using a Huffman code. In the second approach Kozanitis et
al. note that given a value $q$, the number of different quality scores that follow $q$ is
normally restricted to a small group of possible values. A Markov-encoding [CH87]
method was proposed, in which a first-order automaton $M$ containing one node
for each possible quality score, and transitions between all the nodes, is computed.
Each transition $q \rightarrow q'$ has a probability equal to the number of times that $q$ is
followed by $q'$ in the Quality field, divided by the number of times that $q$ is present in the field. The last step, given the probabilities of the transitions, is to construct a Huffman code for each node, and encode each field using the corresponding Huffman encoding scheme. In their work, to store this model, they store the probabilities of each transition $q \rightarrow q'$, and encode each transition using a Huffman code of approximately $-\log(\text{probability of } q \rightarrow q')$ bits.

Kozanitis et al. also explore a lossy compression of quality scores, arguing that a lossy encoding of these values could further enhance compression and that the information loss would not necessarily affect downstream applications. They define a lossy quality score, given a parameter $b$, as:

$$q_{\text{lossy}} = q_{\min} + (q_{\max} - q_{\min}) \cdot LQ\text{-score}_b(q),$$

where

$$LQ\text{-score}_b(q) = rround\left(\frac{q \cdot 2^b}{q_{\max} - q_{\min}}\right),$$

and

$$rround(x) = \begin{cases} 
[x] & \text{with probability } x - \lfloor x \rfloor \\
\lfloor x \rfloor & \text{with probability } \lceil x \rceil - x 
\end{cases}$$

Kozanitis et al. argue that the use of the function $rround$ (or randomized rounding), given that quality scores computation already involves a loss of precision, help to reduce the error of precision of the transformed quality scores. After all quality scores are transformed to $q_{\text{lossy}}$, they are encoded with the new lossy quality information with the Markov-encoding scheme previously discussed.

As mentioned in Section 3.1.2, it should be possible to support random access to the stored data, but it was not discussed in Kozanitis et al.’s work.

### 3.2.2 Block and SuperBlock Quality Score

Deorowicz and Grabowski [DG11], following their block and superblock scheme (Section 3.1.3), also describe a compression method for the quality scores within each block. They categorize the quality score sequences into three kinds: sequences...
where the quality scores are quasi-random distributed and some small positional
dependences can be found; sequences with a quasi-random distribution and mild
positional dependency, but finishing with many low quality scores together (in
general a sequence of ‘#’ symbols); and sequences with strong local correlations
between neighboring quality scores.

To compress each quality score sequence, Deorowicz and Grabowski first identify
the category to which the sequence belongs, storing this information at the beginning
of each encoded quality score sequence, and then handling each category in a
different way. If the quality score sequences within a block are from the first two
kinds, they are Huffman encoded (see Section 2.4.1), using the frequencies of the
quality scores inside the superblock containing the block. When the sequence is
suffixed by many equal low quality scores, these are removed, and the length stored.

When the quality score sequence has strong local correlations, often indicated by
consecutive quality scores having equal values, Deorowicz and Grabowski propose
representing these cases by storing equal consecutive values as one, and adding a
runlength coding to the final stored sequence. For example, if the quality sequence
is “AFAAASFSAFFS”, the sequence stored is “AFAFSAFS”, plus a the runlength
code 0, 0, 2, 0, 0, 0, 1, 0. The new sequence and the runlength code are stored using
Huffman codes, which are also computed using the frequency of the values within
the superblock containing the block.

As stated in Section 3.1.3, random access to the information is discussed, but no
implementation studies of performance were carried out.

3.2.3 Binning Quality

Wan et al. [WAA12] discuss several compression methods for quality scores using
schemes that are lossless, lossy, or lossy followed by a lossless transformation before
the encoding process. Figure 3.5 illustrates the possible paths followed by the quality
scores in their compression process.

In order to control memory usage during construction of the data structure,
the input is processed in independent chunks, where a chunk of size \( k \) represents \( k \) quality score lines. Each chunk incurs a small extra storage cost, as the header information about the transformations performed, and the encoding used for that chunk, must be included in the compressed file.

Wan et al.’s lossy approaches transform quality scores into a bin number, where each bin symbolizes an interval of values. Three lossy transformations are defined: UniBinning, Truncating, and LogBinning. The UniBinning transformation uniformly divides the values given their probability. For example, if we divide the values in four (and hence use two-bit Binary codes), the first bin contains all the quality scores corresponding to an error probability between 0 percent and 25 percent, the second between 25 percent and 50 percent, and so on. The Truncating transformation separates all the quality scores independently, creating a special bin for the \( l \) largest values, with \( l \) a user-defined parameter. For example, if \( l \) is equal to 3 and the set of possible quality scores is \{68, 69, 70, 72, 75, 78, 79\}, then the values 75, 78, and 79 are represent by the same bin value, while the rest of the quality scores get assigned an independent bin for each. Finally, the LogBinning transformation is similar to UniBinning, but instead of separating the values by the error probability, the bins are separated according to their Phred-scaled value, by constructing equal-sized segments according to quality scores. Note that in all the previous transformations each quality score is mapped by the decoder to the smallest quality score, found in the input quality, within the interval of each bin.

Coming back to the Truncating transformation example, \{68, 69, 70, 72\} does not
change, but \{75, 78, 79\} are all changed to 75. This reduces the number of distinct quality scores in the transformed input, increasing compressibility.

Three lossless transformations are explored, each it can be applied as complementary to the lossy transform: \textsc{MinShifting}, \textsc{FreqOrdering}, and \textsc{GapTranslating}. The \textsc{MinShifting} transform stores the minimum quality score of each chunk, and all the values $q$ of the chunk are transformed to $q - q_{\text{min}}$ before being coded. The \textsc{FreqOrdering} transformation maps all the quality scores of a given chunk by their frequency, giving the shortest codes to the more frequent values, when \textsc{Gamma} or \textsc{Golomb} are employed. The \textsc{GapTranslating} transformation changes all values of a chunk to the \(\Delta\)-values proposed by Kozanitis et al.

Finally, after a lossless, lossy, or lossy followed by a lossless transformation, Wan et al. encode the transformed quality scores via one of the codes described in Section 2.3.

### 3.2.4 QUALCOMP Tool

Ochoa et al. [OAB\textsuperscript{+}13] present the \textsc{QualComp} tool, which compresses quality scores using a lossy transformation. It is based on the assumption that quality scores are drawn from a Gaussian distribution [Lap97] parameterized by the position within the quality score line; that is, there is a mean and standard deviation of quality scores in each position $i \in [1, \ell]$ in the read. Their work compresses the quality scores using only a number of bits as specified by the user, minimizing the distortion between the original quality scores and the transformed values. Ochoa et al. approach consider the \textit{mean square error} (or MSE, see page 116), as the measure of distortion between two quality score streams, calculated as the average of the MSE across all of the scores in all of the quality score sequences.

Assuming that reads are of the same length, $\ell$, and that quality scores across different reads of the same regions are not related, Ochoa et al. model the quality scores as an approximated Gaussian distribution $\mathcal{N}(\mu, \Sigma)$, where the mean and covariance matrix of quality scores are empirically computed from the entire original
input set. They perform singular value decomposition (SVD) over Σ to recover the standard deviations to be used as parameters of the Gaussian. This allows the use of optimization techniques from rate distortion theory to optimally allocate bits to minimize the MSE.

The optimization problem is parametrized by a user input which is the number of output bits the quality scores can consume, expressed as a rate r. The solution to the problem specifies, for position i in a read, the number of bits, β, that should be used to encode the quality score in position i. For each quality score, if it is to be coded in β_i bits, the Gaussian for that position is split into β_i regions and a bit pattern representing the region in which the quality score falls is emitted. The decoder reads in the Gaussian parameters and r, and performs the optimization to recover the β_i values, and then emits a representative corresponding to the region specified by each codeword.

Note that if r = 0, meaning lossless compression, not all the reconstructed quality scores receive the same value as in the original quality file. Instead, the reconstructed quality lines will be equal to the empirical mean of the original quality score sequences, but each quality score within the sequence will be different than its original value (except if its value was equal to the empirical mean). Also, given that Ochoa et al. use an approximate Gaussian model, even when the number of bits assigned are enough to achieve a lossless compression, the reconstructed quality scores after decompression may not be the same as the original input.

### 3.3 SAM File Compression

The SAM format (Section 2.8.4) has become one of the most used formats for storing alignment data, in no small part because it is the output format generated by many aligners\(^2\). For example, the compressed version of the SAM format, BAM, is currently the preferred storage file format of the 1000 Genome Project\(^3\). In this


\(^3\)\url{http://www.1000genomes.org/}, July 2014

66
section we discuss the existing previous works that fully compresses SAM files.

### 3.3.1 BAM Format

The BAM format [LHW+09] is a binary representation of the SAM file that uses about 30 per cent of the original SAM file size by employing lossless compression techniques. This format uses the *BGZF* (Blocked GNU Zip Format) compression format, which is a block compression technique on top of the standard *gzip* file format (see Section 2.6.1). Each block contains the *gzip* compression of the data within the block, preceded by some extra fields that give specific information about the file that has been compressed.

The BAM format also supports queries over the compressed data. For example, *SAMTools*, allows the query \( \text{getInterval}(\text{rname}, x, y) \) to be answered without decompressing the whole compressed file. This query returns the set of aligned reads whose RNAME field (see Section 2.8.4) is equal to \( \text{rname} \), and which have alignment starting positions (POS field) contained in the interval \([x, y]\).

In order to support this query, the BAM file must be ordered by reference name and alignment start positions, which can be done using *SAMTools*. For more information about how a BAM file is constructed, refer to the SAM format specification manual, which can be found in the project web page\(^4\). In addition, an index over the BAM file must be included to support the query. This index contains information about the block data stored in *BGZF* format, permitting fast retrieval of alignments given a specified region, only decompressing and getting the data from the block of interest.

\(^4\)http://samtools.github.io/hts-specs/SAMv1.pdf, July 2104
3.3 SAM File Compression

3.3.2 CRAM Format

The CRAM format [FLCB11] is a compressed representation of SAM/BAM files. These files are generated using CRAMTOOLS\(^5\), which is a set of Java tools and APIs that compress the DNA sequence and quality data. In the current version of the tool it is possible to compress SAM and BAM to the CRAM format, but CRAMTOOLS only supports decompression from CRAM to BAM. Figure 3.6 illustrates the compression and decompression paths available using SAMTOOLS and CRAMTOOLS.

The aim of CRAM is to offer better information-preserving compression than BAM, supporting fast transition between these two formats [FLCB11]. A lossy compression mode is also supported, enabling users to choose which data should be preserved. The only dependency of the CRAM format is to an external reference genome. Sequences are stored as the difference between itself and the external reference. Therefore, the external reference genome must be provided each time compression or decompression is done. One strong requirement over the input file (SAM or BAM) is that it must be ordered by sequence reference name and alignment start position. In this work, and in general, it is assumed that the order in which the reads are presented into the input files does not have any meaning, and thus that it is possible to reorder them without losing information.

Unlike what is showed in Figure 3.6, in practice the information flow between SAM, BAM and CRAM files is not completely preserved. The main reason is that SAMTOOLS and CRAMTOOLS encode and decode the information in

---

\(^5\)http://www.ebi.ac.uk/ena/about/cram_toolkit, July 2014

68
different ways. Hence any BAM file generated with SAMTOOLS can be losslessly decompressed to SAM but, if the BAM file was generated with CRAMTOOLS, it cannot be losslessly decompressed to SAM. CRAMTOOLS does not preserve the SAM’s TLEN field in CRAM files, instead it gets calculated when converting from CRAM to BAM, which might generate a slightly different value than the original. Similar problems can be found with the OTHERS field, where CRAMTOOLS does not preserve all possible tags, therefore it calculates them when these are not in the original data. Figure 3.7 shows the actual transitions between these formats. The file sizes vary, and normally File2.SAM uses more storage space than File.SAM, and also the BAM files generated with SAMTOOLS always use less space than the BAM files generated using CRAMTOOLS.

To effectively compress a read sequence, CRAMTOOLS stores only the information of the parts of the read that are identical or near identical to the input reference sequences, otherwise the information is not stored. When the read is near identical, locations of the bases that differ from the reference sequence are stored using GOLOMB codes. Also using GOLOMB codes, the start positions where each read

---

**Figure 3.7:** Actual compression and decompression flow using SAMTOOLS and CRAMTOOLS.
was aligned with the referenced sequence is stored as differences between successive values. Any variation from the reference is stored as an offset relative to the start position of the read, along with the base identities.

In addition, CRAMTOOLS allows several user-defined options that control the lossy compression of the quality score information. Three combinations are explored in the experiments reported in Section 5.3: `preserve`, `bin-preserve`, and `match-bin-preserve`. The `preserve` mode only stores quality scores for reads whose mapping quality scores are higher than a user-defined parameter. The other two approaches are variations of the `preserve` mode, where prefix `bin` means that the quality scores are stored using eight bins like Wan et al.'s LogBinning (Section 3.2.3); and the prefix `match` indicates that only quality scores of bases that match with the reference are preserved. All quality scores under the lossy parameter are encoded and stored using a Huffman-based code.

CRAMTOOLS offers fast transition between CRAM and BAM files, but does not supply random access to the information in CRAM format. Random access queries are supported using SAMTOOLS over BAM files, requiring CRAM files to be decompressed first.

### 3.3.3 NGC Compressor

In 2013 Popitsch and von Haeseler developed their NGC tool [PvH13], offering a mechanism that compresses data stored in SAM/BAM format, covering only reads for which mapped information was available (that is, the RNAME value is known). In their work they assume that input files are ordered by RNAME and POS fields, and that unmapped reads (in which RNAME is unknown) are pruned or not considered part of the data to be stored.

The proposed approach compresses each field of SAM format files separately, applying different techniques depending on the respective stored values and their distributions. Popitsch and von Haeseler present a pseudo information-preserving solution, and a lossy solution. The lossy solution controls how the quality score
sequences are stored, also using user-defined parameters indicating how other fields might be changed. For example, NGC enables the user to modify the field QNAME, pruning the assigned names or re-assigning new names (this is normally done to save space).

In the pseudo information-preserving solution explored by Popitsch and von Haesele, while the information obtained from the process of compressing and decompressing a SAM/BAM file differs from the original input file, the general information contained by the data stored is preserved. For example, NGC does not preserve the order of the OTHER field and some parts of this field are not stored when the file is compressed, but instead they are recomputed when the compressed file is decompressed, not necessarily generating the same data stream as in the original file. Also, NGC does not store CIGAR fields, which are reconstructed using the information stored in the remaining fields, and hence they might also differ from the original CIGAR field values.

NGC focuses on compressing the read and quality score sequences, without paying much attention to the remaining fields of the SAM file. Most of these fields, excluding the Read Sequences and the Quality field, are stored using runlength encoding over their values, or the sequences are generated by computing the difference between consecutive values. The only two fields that follow a different path are POS and QNAME, where Popitsch and von Haeseler encode the positions using a Golomb code of the difference between consecutive positions, utilizing as Golomb parameter the mean difference between neighboring position values. The QNAME fields, for reads coming from the same alignment process, usually prefixed by the same description, followed by an identification number, for example “SRR032209.5132052.1”. Popitsch and von Haeseler suggest to prefix-encode this field, where for every $n$ (a predefined parameter) read names, the longest common prefix between the reads is computed and recorded once, while the remaining part of each read name is stored separately.

To compress the Read Sequences field, NGC assumes that the reference sequence is an external known component provided by the user at compression and decompression time. As for many of the methods presented in Section 3.1,
they propose to store the differences between the reads and the reference sequence, instead of compressing the reads independently. With this purpose, the NGC tool first transforms all the read information, replacing the bases that are equal to the respective reference sequence by a new symbol ‘E’. Figure 3.8 shows an example of how four reads are transformed using a given reference sequence. Once the reads are transformed, the algorithm computes the runlength over the transformed reads in column-wise order, ensuring the reads are ordered by their aligned position with the reference sequence. Each column is considered a
continuation of the previous column, computing the runlength over the complete set of reads and also storing the length of each read (if all the reads were of the same length, NGC only store the length once). In the example in Figure 3.8, the runlengths for the first six columns are \((9, E), (2, A), (1, E),\) and \((2, A)\). Finally each vertical runlength is represented using a fixed sized 8-bit code word (a byte), splitting the run if the runlength value is greater than 256, and also using an eight bit codeword per base stored. The final generated data stream is then compressed using a general purpose compressor such as gzip or bzip2 (see Section 2.6).

The NGC tool offers a simple lossless compression of the quality scores, where all the quality scores are compressed as a whole text file using bzip2. If lossy compression is desired, NGC provides a methodology that using a binning strategy as in Wan et al.’s LOGBINNING (Section 3.2.3), which transforms quality scores whose values are within a predefined interval and assigns the same representative value. The difference with Wan et al.’s approach is that not all the quality scores are treated equally, distinguishing two categories: quality scores of bases that match with the reference sequence; and quality scores of bases that mismatch with the reference sequence. Furthermore, both categories are divided into values that occur in columns where all the bases are the same and those that occur in columns where different bases are found.

Popitsch and von Haeseler propose storing each quality score category using a different size of the interval for the binning transformation, arguing that changes in the categories have different impacts on downstream applications. For example, they argue that qualities whose respective bases match with the reference sequence have a weak impact in the future use of the quality score, so recommend wider bin intervals to represent these quality scores, and “fine-grained” intervals otherwise. Also, their tool offers the option to “lock” some column’s quality scores, in cases where the user knows in advance that the values are important for downstream applications. After the quality scores are transformed, the NGC tool, depending on a user-defined parameter, stores all the values as a byte stream or traverses the quality scores vertically, computing runlength codes as presented for the reads. Finally the generated data stream is compressed using a general purpose compressor.
The final NGC scheme works over each field individually, transforming and replacing their sequences and values, applying the mechanisms just discussed; and then each component is compressed using a general purpose compressor. Figure 3.9 shows this arrangement. Note that the approach shown by Popitsch and von Haeseler does not present any study or support for random access to the compressed data. It is a storage scheme only.

### 3.4 Downstream Applications

In this thesis we make use of two downstream applications, with the purpose of testing how SAM compressed approaches affect the biological results obtained. A common downstream use of next-generation sequencing data is the discovery of variations from paired sequences. These variations often determine the genotype for each individual at each site, allowing, for example, the study of diseases [NPAS11] or common characteristics between different sequences. The process of finding variations is know as *Single Nucleotide Polymorphism (SNP)* and *insert/deletion*
(indel) detection. In this context, the first downstream application that we consider uses the SAMTools software to compute variations. The normal output of this process is a set of SNPs and indels, which can be stored in a standard format such as the Variant Call Format (VCF) [DAA+11]. Each line in a VCF file represents a different variation, storing its position and allele in the reference genome, plus the bases related with the variation, a quality score, and extra information about the variation. Figure 3.10 shows an example of a VCF file.

The process of finding variations is commonplace when next-generation sequencing data is analyzed. In Chapter 5 we use VCF files to measure the effect of lossy transformations over quality scores on SNP and indel detection.

Another important process in the analysis of genomic data information, is computing the coverage, that is, the number of reads stored that represent a desirable zone; for example exon locations (parts of DNA that are converted into mature messenger RNA) or a specific gene [LHP+13, LSS14, APH15]. We refer to these zones as genomic features. In this work we considered the use of featureCounts program [LSS14], which “is a highly efficient general-purpose read summarization program that counts mapped reads for genomic features”\(^6\). The program input consists of one or more SAM/BAM files and a list of genomic features, which can be in general feature format (GFF)\(^7\) or simplified annotation format (SAF)\(^8\), and the output is the number of reads assigned

---


\(^7\) [http://www.sanger.ac.uk/resources/software/gff/spec.html](http://www.sanger.ac.uk/resources/software/gff/spec.html), October 2014

3.5 Random Access Operations

In general, when large amounts of data are compressed and stored, it is desirable to be able to extract information from the compressed data without the need to decompress the entire data set. In other words, it is beneficial to support random access operations that extract only the part of the data that is of interest.

Figure 3.11: FEATURECOUNTS program scheme.

<table>
<thead>
<tr>
<th>GeneID</th>
<th>Chr</th>
<th>Start</th>
<th>End</th>
<th>Strand</th>
</tr>
</thead>
<tbody>
<tr>
<td>245938</td>
<td>chr20</td>
<td>68351</td>
<td>68408</td>
<td>+</td>
</tr>
<tr>
<td>81623</td>
<td>chr20</td>
<td>126056</td>
<td>126392</td>
<td>+</td>
</tr>
<tr>
<td>140850</td>
<td>chr20</td>
<td>138186</td>
<td>138234</td>
<td>+</td>
</tr>
<tr>
<td>140850</td>
<td>chr20</td>
<td>139415</td>
<td>139804</td>
<td>+</td>
</tr>
<tr>
<td>245939</td>
<td>chr20</td>
<td>170216</td>
<td>170264</td>
<td>-</td>
</tr>
<tr>
<td>1457</td>
<td>chr20</td>
<td>476363</td>
<td>476446</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 3.12: Example of a SAF file format.

to each feature plus statistical information for the overall summarization results. Figure 3.11 shows a scheme of the input and output of the FEATURECOUNTS program. We make use of SAF format files consisting of five columns: feature identifier, chromosome name, start position, end position, and strand; where each row is a feature. Figure 3.12 presents an example of a SAF file.

In Chapter 6 we use the FEATURECOUNTS downstream application showing how our compression approach could be included as input to the program, obtaining the same output, and measuring the time required, comparing against the times obtained using SAM or BAM input files.

3.5 Random Access Operations

In general, when large amounts of data are compressed and stored, it is desirable to be able to extract information from the compressed data without the need to decompress the entire data set. In other words, it is beneficial to support random access operations that extract only the part of the data that is of interest.
To achieve this goal, some extra auxiliary information about how the data is stored may be needed. For example, the BAM format generates an extra index (Section 3.3.1) to allow queries over the compressed data, supporting two basic queries:

- \textit{getInterval}(rname): Return the set of data lines in which their reads were aligned against the reference \textit{rname}.

- \textit{getInterval}(rname, x, y): Return all the data lines in which their reads were aligned against the reference \textit{rname}, and their start aligned positions are contained in the interval \([x, y]\).

In the following chapters we will discuss how we support these two queries in the compressed format proposed, and also how it is possible to allow some additional operations.

## 3.6 Summary

This chapter introduced existing research approaches that compress \textsc{FastQ}, \textsc{SAM} files, or some of their fields. A distinction was made between the proposals that compress DNA reads, those that focus on compressing quality score sequences, and those that offer complete \textsc{SAM} file compression.

Section 3.1 showed that most of the solutions assumed the existence of a known reference sequence, which was used as an external input for the compression and decompression process. These methodologies store the positions and bases that differ from the reference sequence.

Section 3.2 introduced the concept of quality scores for DNA sequences and explored different compression techniques, focusing on the ones that offered lossy compression modality. In general all the techniques presented transform the quality scores, by mapping the original quality scores to a smaller set of values, with the purpose of reducing the alphabet size and improving compression ratios. From these
procedures, the most interesting ones considered grouped the quality scores in bins (for example, all values between 0 and 20 form a bin) and all the values contained within a bin were represented with a unique value. Another interesting approach proposed to transform the values so the mean square error within a quality score line is minimized. In Section 5.3 we examine how these compression methodologies perform in more details.

Section 3.3 discussed three SAM compression techniques, BAM, CRAM, and NGC. For our purposes, BAM is the most important, given that beside compressing SAM files, it also offers the option of creating an index to allow random access to the data stored. On the other hand, CRAM and NGC only offer full (non-lossless) compression and decompression of the data. In Chapter 6 we study and compare these approaches in more detail.

This chapter also described the two downstream applications that are used in later chapters to measure the impact of lossy compressed SAM files. The first downstream application was variation detection, where using an external reference sequence, differences against the data stored were located, depending also on the quality scores of the reads. The second downstream application considered, counts the number of reads occurring (the coverage) in a given interval of positions and/or conditions over the SAM fields (features).

In the final section of this chapter, we listed the main operation supported by BAM; getInterval, which returns all the aligned reads within a user-defined interval and reference (that is, given a reference name and a range of positions).

Table 3.2 summarizes the compression techniques presented in this chapter, indicating which compression modality each of these methodologies offer for each of the SAM fields. Also in Table 3.2, we indicate if the solution proposed needed a external reference and if random access to the compressed data is supported. Note that the approaches that do not offer random access but could be indexed, are labeled as “Potentially” indexed.
<table>
<thead>
<tr>
<th>Section</th>
<th>Extern Reference</th>
<th>Indexed</th>
<th>Read Sequences Field</th>
<th>Quality Field</th>
<th>Remaining SAM Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Christley et al. [CLLX09]</td>
<td>3.1.1</td>
<td>Yes</td>
<td>No</td>
<td>Lossless</td>
<td>None</td>
</tr>
<tr>
<td>SLIMGENE Tool [KSK+10]</td>
<td>3.1.2, 3.2.1</td>
<td>Yes</td>
<td>Potentially</td>
<td>Lossless</td>
<td>Lossless and Lossy</td>
</tr>
<tr>
<td>Block and SuperBlock [DG11]</td>
<td>3.1.3, 3.2.2</td>
<td>No</td>
<td>Potentially</td>
<td>Lossless</td>
<td>Lossless</td>
</tr>
<tr>
<td>ASCII DNA [PG14]</td>
<td>3.1.4</td>
<td>No</td>
<td>No</td>
<td>Lossless</td>
<td>None</td>
</tr>
<tr>
<td>ORCOM [SGR15]</td>
<td>3.1.5</td>
<td>No</td>
<td>No</td>
<td>Lossless</td>
<td>None</td>
</tr>
<tr>
<td>G-SQZ [TLS10]</td>
<td>3.1.6</td>
<td>No</td>
<td>Potentially</td>
<td>Lossless</td>
<td>Lossless</td>
</tr>
<tr>
<td>Binning Quality [WAA12]</td>
<td>3.2.3</td>
<td>No</td>
<td>Potentially</td>
<td>None</td>
<td>Lossless and Lossy</td>
</tr>
<tr>
<td>QUALCOMP Tool [OAB+13]</td>
<td>3.2.4</td>
<td>No</td>
<td>No</td>
<td>None</td>
<td>Lossy</td>
</tr>
<tr>
<td>BAM Format [LHW+09]</td>
<td>3.3.1</td>
<td>No</td>
<td>Yes</td>
<td>Lossless</td>
<td>Lossless</td>
</tr>
<tr>
<td>CRAM format [FLCB11]</td>
<td>3.3.2</td>
<td>Yes</td>
<td>No</td>
<td>Lossless</td>
<td>Lossless and Lossy</td>
</tr>
<tr>
<td>NGC [PvH13]</td>
<td>3.3.3</td>
<td>Yes</td>
<td>No</td>
<td>Lossless</td>
<td>Lossless and Lossy</td>
</tr>
</tbody>
</table>

Table 3.2: Summary of the approaches discussed in Chapter 3. For each approach we indicate if an extern reference is needed, if the compressed data is indexed and support random access, and which compress modality offer for each of the SAM fields. Note that the approaches that do not offer random access but could be indexed, are label as “Potentially” indexed.
Chapter 4

Read Sequence Compression

In order to compress SAM files we will deal with each of its fields (see Section 2.8.4) individually. As was demonstrated in Figure 2.12, in general the bulk of the space used is contained in the reads and their respective quality scores. This chapter will describe our approaches to compressing the Read Sequences field of SAM files.

In the SAM format, multiple reads might be described as being relative to the same external reference sequence. Those alignments might also relate to each other, sharing similar information when the reads overlap. In the following approaches it will be assumed that the original reference sequences are not available as part of the information used during the compression process. The information of the reference name (RNAME) and relative position (POS) fields are included as part of the read information, as they are important components of the proposed compression approach.

The solution presented in this thesis assumes that the input SAM file is ordered by reference name and relative position, otherwise, if not, this can be achieved using the command SortSAM from Picard tool\(^1\) over the original SAM file, or using SAMTools with the command sort over the compressed BAM version of the file.

\(^1\)http://broadinstitute.github.io/picard/, January 2015
This chapter introduces a detailed description of our approach, followed by a study and analysis of the proposed practical implementation of the ideas discussed. The analysis will be conducted using a set of training files, including tuning the parameters of the implementation. Finally, we show results over a new set of test files to corroborate the solution obtained from the training set.

4.1 Overlap Presumed Sequence

An overlap is defined as two or more reads that are aligned against the same reference sequence, where the difference between the starting positions of the reads is within the length of the first-occurring read. In general, as we will demonstrate in the experimental evaluation, SAM files have a high presence of overlap reads.

Assuming a high overlap presence suggests that it may be possible to store the reads effectively by exploiting the similarity of the read lines when they do overlap. Given that the reads are ordered by reference name and relative position, it is likely to be able to identify a presumed sequence that represents the overlap bases, and then being sufficient to store this sequence, plus the position and bases where each read component differs relative to the presumed sequence. That is, to reconstruct any read, assuming that its reference and positions fields are known, it is sufficient to extract the representative segment of the presumed sequence and replace the bases at the position where differences were stored.

This section introduces two approaches for computing the presumed sequence over a set of overlapping reads, briefly discussing the advantages and disadvantages that we might encounter at the time of implementation.

4.1.1 First-Base-Encountered

The first methodology used for forming a presumed sequence consist on choosing the first base encountered in each position as the representative base in the presumed sequence. Once that is done, subsequent reads that refer to the same position
need to indicate whether they are the same as the presumed base or not; and, if different, a record of the base and the position at which the respective read is different to the presumed sequence is kept. Figure 4.1 gives an example of a first-base-encountered presumed sequence for a set of overlapping reads. As the figure shows, the reads are processed in reference (assuming only one reference for the figure) and position order, and each time a read is encountered, we sequentially check its bases storing the bases that do not match with the generated reference. When the presumed sequence does not extend to the position being checked, the corresponding base is concatenated to the existing presumed sequence.

While this approach only need visit each read once to create the presumed sequence and compute the differences, it may not lead to space savings. Depending on how well the first read’s bases represent the bases of the following reads in the same position, the number of bases that differ with the presumed sequence can vary a lot. For example, if the first read encountered perfectly stand for all the following bases that occurred at the same positions, the number of differences would be zero. At the other extreme, if the first read visited does not mirror any of the following overlap sequences, then all the subsequent overlap reads will be completely different.
4.1 Overlap Presumed Sequence

Chapter 4 Read Sequence Compression

Figure 4.2: Example the last-read-rule approach, where the presumed sequence is generated using the first-base-encountered methodology, but the overlapped base differences are computed against the last overlap read visited. Arrows indicate the segment that is used as reference sequence, and the circles enclose the bases that differ with respect to the corresponding reference.

and all their bases would be stored as exceptions.

Whereas SAM files can be ordered by the position field, there is not a set rule over which read must be stored first if more than one read start at the same position, meaning that using first-base-encountered to generate the presumed sequence it does not ensures that the first-stored read is a good representation of the ones that follow.

Another way to store the reads is based on the assumption that consecutive reads should be similar. In this case, instead of using a presumed sequence to compute the exceptions, for each read, the last overlap visited read is used as reference. We will refer to this technique as last-read-rule. Note that a presumed sequence, generated using the first-base-encountered approach, is still needed, as it is used to obtain the bases of the reads that have not previously overlapped any other read. Figure 4.2 shows an example of how this methodology works. Assuming that similar reads are consecutively stored, the number of differences between reads should be small. For example, in the figure the number of exceptions found in the Read 3 and 4 are reduced if the last-read-rule technique is used to compute the exceptions instead of just using the first-base-encountered presumed sequence as reference for all the reads.
The methodologies discussed present two possible complications: for the last-read-rule to extract a read, it is necessary first to extract all previous overlapped reads; and, as for the first-base-encountered, there is not a set rule over the similarity of consecutive stored reads, being impossible to guarantee the last overlap read would be a good representation of the one that follows it.

4.1.2 Most-Frequent-Base

An alternative to minimize the number of differences between the presumed sequence and the overlapping reads, is to generate presumed sequence using the bases that are most frequent in each overlapping position, when all the overlaps are aggregated together. This approach requires first the creation of the presumed sequence, and then, once this artificial reference has been computed, revisiting all of the overlapping sequences, keeping a record of the positions at which the respective read lines differ with the presumed sequence. In this case we do not allow the symbol N to be part of the presumed sequence. Assuming low frequency of finding a symbol N in the read sequences, and being preferable to have a presumed sequence with an alphabet of size 4, we propose to encode the positions of the N symbols separately. Figure 4.3 shows an example of how this most-frequent-base methodology works.

Whereas the process to build the presumed sequence is more complex than in the previous approaches, extracting a read is done as for the first-base-encountered approach, i.e., in general terms, extract the representative segment of the presumed sequence and replace the bases at the position that exceptions were stored.

4.2 Analysis and Tuning

In this section we explore and study the performance of the methodologies to generate presumed sequences, by computing the distributions and properties of each component of the solutions. Also, this section will introduce compression methodologies to store the components of the presumed sequence solutions and it
4.2 Analysis and Tuning  Chapter 4 Read Sequence Compression

Figure 4.3: (a) Example of construction of the presumed sequence using the most frequent bases in each position between all the overlapping sequences, with $N$ symbols considered to be non-voting. (b) Example of second step where each sequence is revisited, keeping a record of the position at which the respective read line differs with respect to the presumed sequence, again ignoring any $N$s.

86
will discuss how the proposed implementation supports random access to the data.

As a training files, for the purpose of exploring attributes of the input data, we used the *NA12878* genome, which is the most sequenced human individual from the *1000 Genomes* project\(^2\). We work with the NA12878 read alignments that were aligned with chromosome 20 (*chr20*)\(^3\). Two versions of this file were explored; a low coverage version obtained from the *1000 Genomes* project web page; and a high coverage sequenced version that can be found in the Galaxy Data Library\(^4\). This second version was also aligned with chromosome 20, but using the complete *Homo sapiens*\(^5\) genome as reference instead of the isolated chromosome information. We also take genome *HG01477*, with read alignments that were aligned with chromosome 11 (*chr11*)\(^3\), to experiment with a training file which is aligned over a different chromosome. Table 4.1 presents details of these input files, also including information of the reference sequences used in the alignment process.

As previously mentioned, files in SAM format usually contain multiple overlapping reads. To quantify this claim, we measured the percentage of reads that overlap at least in one base, and the number of bases that overlap over each of the training files. Table 4.2 shows the results obtained including the median, maximum, and average, taken over the set of all bases present in the file, of the number of bases that share the same position in regard to the same reference sequence. Given that, in the training files, all the reads are of the same length, the average reported in Table 4.2 is an upper limit of the coverage value of the files.

Two approaches to generate a presumed sequence were proposed, first-base-encountered and most-frequent-base, plus the last-read-rule which is a variation of how to compute the exceptions within the reads. To compare them we compute,

\(^2\)http://www.1000genomes.org/
\(^3\) Chromosome 11 and 20 were extracted from the human genome file at ftp://ftp.1000genomes.ebi.ac.uk/vo11/ftp/technical/reference/phase2_reference_assembly_sequence/hs37d5.fa.gz
\(^4\)https://usegalaxy.org/library_common/browse_library?show_deleted=False&controller=library&use_panels=False&id=f9ba60b9a2e6ba6d
\(^5\)http://www.broadinstitute.org/ftp/pub/seq/references/Homo_sapiens_assembly19.fasta
4.2 Analysis and Tuning

Chapter 4 Read Sequence Compression

<table>
<thead>
<tr>
<th>Files</th>
<th>Size (MB)</th>
<th>Reads</th>
<th>Read length</th>
<th>H0</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA12878.low.reads</td>
<td>318.94</td>
<td>3,278,702</td>
<td>101</td>
<td>2.026</td>
</tr>
<tr>
<td>NA12878_low.pos</td>
<td>27.56</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>NA12878_low.ref</td>
<td>9.38</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>NA12878_high.reads</td>
<td>5,017.98</td>
<td>51,585,658</td>
<td>101</td>
<td>2.032</td>
</tr>
<tr>
<td>NA12878_high.pos</td>
<td>433.66</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>NA12878_high.ref</td>
<td>147.59</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>HG01477.reads</td>
<td>617.22</td>
<td>6,407,925</td>
<td>100</td>
<td>1.978</td>
</tr>
<tr>
<td>HG01477.pos</td>
<td>56.15</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>HG01477.ref</td>
<td>18.33</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 4.1: Training read sequence files. The reads and their respective reference and position in the reference were extracted from three SAM files, NA12878.chrom20.ILLUMINA.bwa.ceu_low_coverage.20120522.sam, NA12878.HiSeq.WGS.bwa.cleaned.recal.hg19.20.sam, and HG01477.chrom11.ILLUMINA.bwa.CLM.low_coverage.20130415.sam. The columns show the size of the files when stored as ASCII bytes, the number of reads, the length of each read, and the entropy ($H_0$, in bits per base) of the read sequences.

<table>
<thead>
<tr>
<th>References</th>
<th>Size (MB)</th>
<th>–</th>
<th>–</th>
<th>–</th>
</tr>
</thead>
<tbody>
<tr>
<td>chr11.fasta (from hs37d5.fa)</td>
<td>130.90</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>chr20.fasta (from hs37d5.fa)</td>
<td>61.11</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Homo_sapiens_assembly19.fasta</td>
<td>2,995.26</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 4.2: Statistics for the three training files in terms of overlaps between the stored reads. The final three rows show the median, maximum, and average, taken over the set of overlap present in the file, of the number of bases that share the same position in regard to the same reference sequence.

<table>
<thead>
<tr>
<th></th>
<th>NA12878_low</th>
<th>NA12878_high</th>
<th>HG01477</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overlapping reads (%)</td>
<td>99.3</td>
<td>100.0</td>
<td>98.7</td>
</tr>
<tr>
<td>Overlapping bases (%)</td>
<td>82.3</td>
<td>98.9</td>
<td>80.0</td>
</tr>
<tr>
<td>Median overlap</td>
<td>5</td>
<td>87</td>
<td>5</td>
</tr>
<tr>
<td>Maximum overlap</td>
<td>314</td>
<td>4899</td>
<td>171</td>
</tr>
<tr>
<td>Average overlap</td>
<td>5.6</td>
<td>87.7</td>
<td>5.0</td>
</tr>
</tbody>
</table>

Table 4.3: Statistics for the three training files in terms of overlaps between the stored reads. The final three rows show the median, maximum, and average, taken over the set of overlap present in the file, of the number of bases that share the same position in regard to the same reference sequence.

using the training files, the percentage of the bases that overlap and do not have to be stored as exceptions, depending which method was used to build the presumed sequence and computed these differences. Table 4.3 shows the results obtained. As expected, the most-frequent-base approach better represents the overlapping bases, covering a higher percentage of the bases using only the presumed sequence. Is
possible to infer from the results that, while the first-base-encountered methodology might allow faster construction of the presumed sequence, it does not lead to the largest number of overlapped reads. Also, while the last-read-rule technique offers a middle point between the approaches, it has the drawback that for decompressing a read it is necessary to extract all the preceding overlapping reads. Finally, the most-frequent-base, as mentioned, better represents the overlap base information, but the construction of the presumed sequence may take longer than for the other approaches.

Given that the aim of this research is to obtain a compressed version of the data, and support fast random access, we choose as our preferred method the most-frequent-base approach. For the rest of this thesis whenever the presumed sequence is used, it will be built using the most-frequent-base methodology. We decide to not explore methodologies that mix the approaches previously presented, assuming that the number of exceptions to be stored would not be significantly reduced.

Note that storing the reads using a presumed sequence requires recording, for each read, its start location in the presumed sequence, and the positions and bases that differ relative to the presumed sequence.

<table>
<thead>
<tr>
<th></th>
<th>NA12878_low</th>
<th>NA12878_high</th>
<th>HG01477</th>
</tr>
</thead>
<tbody>
<tr>
<td>Percentage of bases that overlap and are the same as the first-base-encountered presumed sequence</td>
<td>80.3</td>
<td>79.3</td>
<td>93.0</td>
</tr>
<tr>
<td>Percentage of bases that overlap and are not variant, if the last-read-rule approach is used.</td>
<td>81.6</td>
<td>80.4</td>
<td>93.5</td>
</tr>
<tr>
<td>Percentage of bases that overlap and are the same as a presumed sequence based on the most frequent bases</td>
<td>87.7</td>
<td>88.7</td>
<td>95.5</td>
</tr>
</tbody>
</table>

Table 4.3: Percentage of overlapping bases, in the training files, that are correctly represented depending on the approach chosen, first-base-encountered, last-read-rule, or most-frequent-base.
Before explaining how to implement our proposed approach, we study the estimated space needed for storing the components used to represent the read sequences. Several factors were into consideration. First we compute the length of the presumed sequence and its entropy. Note that at this point of the analysis the reads that have an unknown reference or are non-reference were separately considered and are not part of the presumed sequence. The number of bases that differ between the reads and the generated presumed sequence are computed; storing these bases into the four separate subsequences NotA, NotC, NotG, and NotT, where each of them stores only the bases that differ between the reads and the presumed sequence when the base stored in the presumed sequence is A, C, G, and T, respectively. The letter N is omitted and replaced temporarily by the base displayed at the respective same position in the presumed sequence, given that we assume that it is an uncommon symbol.

We also make use of four further arrays: a length array, which contains the length of each read stored; an offset array, that records the distance between the read components; a copy array, which contains the length of the chunks of the read that are equal to the presumed sequence; and a replace array, which stores the length of the chunks that need to be replaced with the bases of the Not sets. Finally we keep an overall list of locations in the reads that are N, called IsN.

The combination of these components is enough, as it will be shown, to compress the information of the Read Sequences and position (POS) fields of SAM files. Figure 4.4 illustrates an example of these various components, assuming random access is not required. Note that not all of these components are necessary in every case. For example, the length of all the reads is often the same within a SAM file, thus the length array can be omitted and the length stored only once. Table 4.4 shows a lower bound on the size needed to store each component of our approach, computed by multiplying the number of elements of the component by the zero order entropy of the elements (assuming that same number of bits are used to code each symbol, allowing random access to their values). The only exclusion is IsN, which in Table 4.4 illustrates the space used if each N position is encoded using
Figure 4.4: Example of components needed to represent the read sequences field.

\[
\begin{align*}
\text{Presumed Sequence} &: \text{A A T G A A G C A T G G T A A C T C C T T A G A T A G T T T A A C A G G T} \\
\text{Read 1} &: \text{A A T G N C A T G G T A A T C N T T A G A} \\
\text{Read 2} &: \text{A A T G C A A G T A A C T C C T T A G A T A G T T T A A C A G G T} \\
\text{Read 3} &: \text{A A T G C A A G T A A C T C C T T A A C A G G T} \\
\text{Read 4} &: \text{A A T G C A A G T A A C T C C T T A A C A G G T}
\end{align*}
\]

\[
\begin{align*}
\text{Length Array} &: 24, 24, 24, 24 \\
\text{Offset Array} &: 0, 2, 0, 11 \\
\text{Copy Array} &: 5, 9, 8, 23, 7, 11, 3, 3, 1, 14 \\
\text{Replace Array} &: 1, 1, 1, 2, 1, 1, 1
\end{align*}
\]

\[
\begin{align*}
\text{NotA} &: T T \\
\text{NotC} &: A \\
\text{NotG} &: A C C \\
\text{NotT} &: A G C \\
\text{IsN} &: \text{(Read 1: 5, 19), (Read 4: 21)}
\end{align*}
\]

log(read length) bits, showing a higher bound of the compression expected for this component.

From Table 4.4, as a general overview of the values showed, is possible to notice that the entropies shown for the Not sets and the presumed sequence are close to the value of the logarithm base two of the number of different symbols for each component (four for the presumed sequence and three for each of the Not sets), implying a similar percentage of occurrences per symbol within the component. Also, in our training files the length array is not necessary, given that all the reads are of the same size (entropy equal zero). We infer that the copy and replace array have similar distributions of their values, and are some of the biggest components to be store. Finally, the position offset has the biggest entropy, but, given that only
<table>
<thead>
<tr>
<th>Position Offset</th>
<th>Length Size</th>
<th>Length Size</th>
<th>Length Size</th>
<th>Length Size</th>
<th>Length Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1,58</td>
<td>5,634,119</td>
<td>302</td>
<td>2,612</td>
<td>1,58</td>
<td>3,634,119</td>
</tr>
<tr>
<td>2,57</td>
<td>2,168,591</td>
<td>322</td>
<td>1,975</td>
<td>2,57</td>
<td>2,168,591</td>
</tr>
<tr>
<td>3,56</td>
<td>1,282,008</td>
<td>345</td>
<td>1,886</td>
<td>3,56</td>
<td>1,282,008</td>
</tr>
<tr>
<td>4,55</td>
<td>0,000</td>
<td>357</td>
<td>1,988</td>
<td>4,55</td>
<td>0,000</td>
</tr>
<tr>
<td>5</td>
<td>0,000</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0,000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.4: Size, in MB, required to store the read sequences information of the three training files of Table 4.1 in a compressed form using a zero order model and a perfect entropy coder when stored using a presumed sequence based on the most frequent bases. The objective is to closely approximate these sizes with practical codes.
one offset value is stored per read, the number of elements to be stored is small compared with the rest of the components.

Based on the results of Table 4.4, we now seek a suitable compressed representation of each component, which is (ideally) a close approximation to the space limit showed. With this purpose we analyze how the values of these components are distributed, and identify an appropriate compression method, without forgetting that our aims includes storage of the components in a way that the random access operations discussed in Section 3.5 are supported.

### 4.2.1 Component Compression

For a better understanding of how the components should be stored first it will be necessary to describe, in general terms, how the data of the input file are manage in order to both compress and support random access. We propose to divide the data into clusters, where a cluster contain all the reads whose start position are between a fixed sample length $\rho$. For example, if take $\rho = 1000$ and given a reference $r$, the first cluster of the reference $r$ would contain all the reads which have $r$ as its reference sequence and an aligned starting position between 0 and 999. Later, in Section 4.3, it will be discuss in more detail how the clusters are stored, indexed, and used for supporting random access to compressed data.

The first component examined is the position at which each read is aligned against the given reference. To compress the positions information, given the cluster division, we store, for each cluster, the offset between the start of the cluster and the first member read, while the rest of the positions of the reads contained inside a cluster are stored as offsets between consecutive reads. Figure 4.5 shows the histogram of the offset values over the training files of Table 4.1 obtained using this approach taking $\rho = 1000$ as the sample length for the cluster division.

Figure 4.5 shows that the offset values approximate a geometric distribution, that is, smaller values are more frequent and the occurrences for bigger values rapidly decrease. To demonstrate the previous statement, Figure 4.5 include a line showing
4.2 Analysis and Tuning

Chapter 4 Read Sequence Compression

Figure 4.5: Histogram over all offsets values for each of the training files of Table 4.1. Note that vertical axis is the frequency of each offset value displayed as a percentage over the total number of offsets values in the file. Also, a fitted geometric distribution is plot for $x \geq 1$. The values within the horizontal axis are increased by one given the logarithmic scale used in this axis.

A geometric distribution, $g(x) = t \cdot (1 - t)^x$, for a given value of $t$, which we will soon explain how to calculate. Assuming a geometric distribution, GOLOMB codes present a good approach to encode the offset values, as described in Golomb’s work [Gol66]. We propose to encode the values using this encoding technique, adjusting the selected parameter for the code within each cluster. In other words, the offset values inside of a given cluster, are encoded using GOLOMB codes with parameter $b$, which is computed using the equation

$$b \approx 0.69 \cdot \frac{N}{f},$$
(presented in Section 2.3.4). In this case, $f$ is the number of offset values stored inside the cluster, and $N$ is the sum of the number of possible values (which is the sample length $\rho$ used for the clusters) and the maximum number of possible times that a value can be found repeated ($f$ times).

In Section 2.3.4 it was argued that, if we are using Golomb codes, with $b \approx 0.69 \cdot \frac{N}{f}$, then this encoding technique should be optimal for a set of values that approximate a geometric distribution with $t = \frac{f}{N}$. Using this statement, Figure 4.5 display a geometric distribution where the value of $t$ was calculated as 0.69 divided by the average of the Golomb code parameter $b$ used to code the offsets (this equation can be inferred using the two previews equations). After encoding the offset values of the training files, it is clear from Figure 4.5, that these values follow a geometric distribution, with the exception of the two smallest offset values in each case. Nonetheless, Golomb assigns the smallest codewords length to these values, which is desired since these values are those that are more frequently found in the training files.

A simple component to be stored is the length array. As mentioned, reads within the same SAM file often are of the same length, making the length array useless, but in case the reads length vary, we stored the different lengths using $\lceil \log(\text{number of possible lengths}) \rceil$ bits to store the length of each read.

In our approach we compute a presumed sequence over all reads, and then store the difference between the presumed sequence and the reads. These exceptions are stored in the “Not” components, where Not$X$ stores the bases that differ when the corresponding base in the presume sequence is $X \in \{A,C,G,T\}$. In this work we assume that the reads are formed only by the bases $\{A,C,G,T\}$, treating other letters as special cases, which are stored separately (all the special cases are assume to be the symbol $N$). It is easy to see that each set can only contain three different bases, which can be stored using a three value code where, for example, patterns 0, 10, and 11 are used to represent the bases. As in Huffman codes (see Section 2.4.1), we simply need to assign these codes to the symbols depending on their frequency, using the one bit codes to symbolize the most frequent symbol.
As show in Table 4.4, the volume of special cases (bases equal to $N$) is small, and it is preferable to store them separately from the “Not” components, reducing the code space for these items, given that they would not be considered as exceptions. Table 4.5 displays the percentage of reads in the training files, that contain at least one special symbol, and the average number of special symbols within a read if present. From the table it is possible to conclude that special symbols concentrate in a negligible number of the reads, and, in general, most of these reads contained several special symbols. To minimize the space used to store these symbols, a bit sequence of length equal to the number of reads in the SAM file is used, with each bit specifying if the corresponding read contains any special symbol or not. Then, if the read contains special symbols, these are stored as the local total number of special symbols, followed by their positions encoded as gaps between consecutive occurrences. Given the insignificant number of special symbols, we decided not to deeply explore how to obtain the best compression possible, and encode the count and positions of special symbols using Gamma codes, which offer good compression.

Following the proposed components to encode a read it is necessary also to store a sequence of counts of bases that are “copy” and “replace” relative to the presumed sequence. Section 3.3.3 described the Popitsch and von Haeseler [PvH13] approach, where they computed the copy and replace array in column-wise order. Popitsch and von Haeseler also propose that the runs have to be computed in a continuous way, where the end of one column is attached to the beginning of the next column. Table 4.6 shows the empirical zero order entropy and number of runs found in each of the training files, when a column-wise procedure is applied to compute the runs. We include the results obtained in the case that each column is consider.
Table 4.6: Column-wise versus Row-wise runlength encoding for the copy and replace arrays. The table includes the empirical entropy and the number of elements in each case. We consider treating each column/row independently or as a continuation of the neighboring column/row. The Relative costs refer to the total space used to store the runlengths when they are compressed using a zero order model and a perfect entropy coder. This cost is expressed as a percentage over the space used by the row-wise approach as a reference (Table 4.4 shows the sizes in MB used by the copy and replace array for this methodology).

From Table 4.6 is clear that for the training files, if the total number of runs and the space used wanted to be reduce, then a row-wise approach is preferable. In our implementation we represent each row independently, assuming that while treating the rows in a continuous way reduces the space needed to store the copy and replace arrays, it would slow the random access to the rows. We assume that this slow down

<table>
<thead>
<tr>
<th></th>
<th>NA12878_low</th>
<th></th>
<th>NA12878_high</th>
<th></th>
<th>HG01477</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$H_0$ Length</td>
<td>$H_0$ Length</td>
<td>$H_0$ Length</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(bps) (symbols)</td>
<td>(bps) (symbols)</td>
<td>(bps) (symbols)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ind. Row-wise</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copy array</td>
<td>3.00 14,025,639</td>
<td>2.74 215,603,901</td>
<td>2.87 13,534,419</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replace array</td>
<td>2.67 11,460,818</td>
<td>2.90 176,222,306</td>
<td>2.72 7,679,196</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relative cost</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ind. Column-wise</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copy array</td>
<td>3.10 78,254,404</td>
<td>4.54 494,493,198</td>
<td>3.21 146,316,627</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replace array</td>
<td>0.72 28,445,796</td>
<td>0.98 444,785,879</td>
<td>0.56 20,485,090</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relative cost</td>
<td>361.97%</td>
<td>243.32%</td>
<td>822.29%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cont. Row-wise</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copy array</td>
<td>3.07 11,117,130</td>
<td>2.79 170,722,371</td>
<td>3.51 7,406,393</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replace array</td>
<td>2.71 11,090,626</td>
<td>2.93 169,518,179</td>
<td>2.75 7,399,298</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relative cost</td>
<td>88.32%</td>
<td>88.31%</td>
<td>79.20%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cont. Column-wise</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copy array</td>
<td>4.03 22,587,491</td>
<td>4.70 437,312,089</td>
<td>4.40 16,144,813</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replace array</td>
<td>1.11 21,689,977</td>
<td>1.03 436,708,407</td>
<td>0.82 15,171,310</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relative cost</td>
<td>158.38%</td>
<td>227.37%</td>
<td>142.66%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6: Column-wise versus Row-wise runlength encoding for the copy and replace arrays. The table includes the empirical entropy and the number of elements in each case. We consider treating each column/row independently or as a continuation of the neighboring column/row. The Relative costs refer to the total space used to store the runlengths when they are compressed using a zero order model and a perfect entropy coder. This cost is expressed as a percentage over the space used by the row-wise approach as a reference (Table 4.4 shows the sizes in MB used by the copy and replace array for this methodology).

continuous and independently. Table 4.6 also displays the same statistic for the row-wise methodology.

From Table 4.6 is clear that for the training files, if the total number of runs and the space used wanted to be reduce, then a row-wise approach is preferable. In our implementation we represent each row independently, assuming that while treating the rows in a continuous way reduces the space needed to store the copy and replace arrays, it would slow the random access to the rows. We assume that this slow down
would be caused because runs encoded continuously over the rows would incur in extra algorithms step to be able to extract independent reads. Also, if the rows are considered separately, it is possible use less space by isolate the special case when a read is a complete copy from the presumed sequence, which it would not be possible if the runs were computed considering the row continuously.

In Table 4.6 we already computed the entropy of the copy and replace runlength arrays. In order to achieve this entropy, it will be needed to choose a code based on the frequency distribution of the runlengths. With this in mind we analyzed the copy and replace runlength arrays computing the histogram over possible values for each of them in the training files. Figure 4.6 and 4.7 shows the occurrence percentages relative to the total length of each array for each of the training files. Considering these results, it can be deduced that, using the presumed sequence approach, two common outcomes arise when a read is copy/replace runlength coded: most or the whole read is a copy of the presumed sequence; or the read is formed by an alternation of short copy and replace operations.
From Figure 4.6 we observe that the copy array has higher number of occurrences for a single base copy, rapidly decreasing the frequency for higher copy values, reaching an approximately steady zone with a frequency percentage close to 0.1 percent per value when copies runlength are higher than 20 bases. The only exception is the other high frequency copy value found in the training files, where the complete read is identical to the presumed sequence stored. Making a similar analysis over Figure 4.7, as expected, most of the replace runlength values were smaller values, being the most common case replacing only one base. Also we notice that the occurrences decrease drastically for higher values getting occurrence frequency percentage lower than 0.001 percent per value, meaning that replace runlength values higher than 20 can be considered extremely unusual cases.

One way to code the copy/replace runlength is to use a binary sequence over each read, where each bit indicates if the respective base is identical to the base in the presumed sequence, or if the base differs. While this representation would be useful in the cases when the read is formed by an alternation of small copies and
replace values, it uses a lot of space in the cases where the copy or replace values are not small. For example if all the bases of a read of length 100 are identical to the selected bases for the presumed sequence, then it would be better to store the number 100 than have a binary sequence of length 100.

A better way to store the copy/replace array is to code these arrays so that variable length codes are used, assigning smaller codes to the most frequent values. In this case, given the results obtained from the training files and assuming that the value distribution found holds for any file, we need a variable length code technique that assigns small codewords to smallest values. In Section 2.3.2 we described an encoding technique that help with this problem, the Gamma codes.

We propose to store the copy and replace runlength values using Gamma codes, but this encoding technique still has the problem that for large values it can required many bits and, in our case, many reads are identical to their presumed sequence representation, translating in long copy runlength values. Another problem that arises when a read is coded using the copy/replace arrays, is that the decoder will need to know if a read start copying or replacing values. To solve these problems we used a bit sequence having one bit per read in the SAM file, where each bit indicates if the first operation is copy or replace. Also, using this bit vector, if the first operation is copy, then an extra bit is included to specified the special case when the complete read is a single copy, expending one extra bit for reads that start with a copy operation, but avoiding Gamma encoding the copy value when it spans the whole read.

The last component to be encoded is the presumed sequence. This component only admits four symbols (the four bases A, C, G, and T) and, as shown in Table 4.4, the entropy is close to 2.0 meaning that all the symbols are approximately equally distributed in the sequence, and suggests that using a simple 2 bit coding per symbol would be enough. Whereas it may be possible to improve the compression by looking for similarities and repetitions within the sequence (a higher order model), we avoid further compression over this component. As we will describe in the next section this component is used to support random access, and a more complex compression of this component would risk decreasing the speed of accessing the data.
4.3 Supporting Random Access

In the previous section we studied each of the components of our proposed approach, describing suitable encoding techniques for each component, but avoiding further discussion of how they are put together so that the getInterval queries, presented in Section 3.5, could be supported. Section 4.2.1 briefly introduced how the input reads are separated into clusters, where a cluster contains all the reads whose start positions are between a fixed sample parameter $\rho$. In order to use this cluster separation, we encode each read individually, reordering and mixing the data of the components associated with each read within each cluster.

Before discussing how the whole data structure for storing reads works, we need to describe in more detail the cluster model that is employed. Figure 4.8 illustrates an example of how the reads are separated into clusters, showing how this model stores the information associated with seven reads, with same RNAME (see Section 2.8.4) field value “Chr 11”, and POS field with values \{45, 420, 916, 1110, 1366, 3040, 3800\} respectively. The cluster sample used was $\rho = 1000$, meaning that every $\rho$ positions in the reference “Chr 11” a bit is added to a bit sequence (see Section 2.5.1), to indicate if at least one read starts within the position range. In case the interval is not empty, we stored a synchronization point to the compressed representation of the first read in the range. There are another two components from Figure 4.8, that are important to the cluster approach: an array containing the different RNAME values existing in the respective SAM file, where each RNAME is associated with the start position in the bit sequence where the first cluster of each RNAME is stored; and another bit sequence over the cluster ranges indicating if any of the reads of the previous cluster range overlapped with the current range, storing also the maximum number of bases that overlapped. We label all these components the “cluster index”. Note that, if all the reads are of the same size, the cluster overlap information is not necessary, as we will explain soon.

Given this model, each cluster is stored independently, including a header containing the number of reads encoded within the cluster and a synchronization point to the first position of the presumed sequence associated with the first read.
in the cluster. After the header, the reads belonging to the cluster are sequentially stored. For each read we store its components, adding extra information, in the following order: the position offset value; the length of the read (unless all the reads are of the same length in the whole file); a bit that indicates if the first operation is replace or copy: If copy is the first operation, a second bit is added to indicate if the read is a full copy of the presumed sequence; if not, it is followed by a sequence of replace and copy runlength values that cover the length of the read, where after each replace runlength with value $r$, the respective $r$ values from the “Not” sets are included sequentially; finally a bit indicates if special symbols were found within the read; and the special symbol positions are encoded at the end if there are any.
Figure 4.9 illustrates an example of this data structure for the first cluster and its related reads from Figure 4.8.

Having this data structure plus the cluster index, it is now possible to solve the operations \text{getInterval}(rname) and \text{getInterval}(rname, x, y) (see Section 3.5) over the Read Sequences, POS, and RNAME fields, without the need to fully decompress the whole file. Algorithm 8 gives the proposed pseudocode to answer \text{getInterval}(rname).

More specifically, each time a non-empty cluster is visited, its data is accessed to determine the number of reads contained, and the start position in the presumed sequence associated with that cluster. Another information needed is if any read of the previous cluster overlap with the current cluster. This is important given that the presumed sequence contains the reads information without storing information about the empty positions gaps, which could generate errors when the reads are regenerated. For example, in Figure 4.8, if the overlap information from r3 is unknown, after extracting r4, our methodology would start extracting the r5 information using the presumed sequence from the point where r4 finished. This case miss aligns the r5 with the presumed sequence, which could be avoided if we knew that part of the presumed sequence in that sector is cover by a read from...
Algorithm 8 `getInterval(rname)` takes a reference name `rname` as input, and returns the reads associated with the given reference. Assume that the data structures from Figure 4.8 exist.

1: `data ← {}`
2: Using the Cluster Index, obtain first and last positions associated with the reference `rname` in the bit sequence “non-empty”
3: `currentCluster ← first`
4: `while currentCluster ≤ last do`
5:  `if currentCluster is non-empty then`
6:   `for all reads within the current cluster do`
7:    `Obtain offset (generating the position) and length of the read`
8:    `if First operation is a copy then`
9:     `if Is a full copy then`
10:        `Generate read using only the presumed sequence information`
11:     `else`
12:        `Generate read using the presumed sequence alternating copy and replace operations`
13:    `end if`
14:  `else`
15:    `Generate read using the presumed sequence and alternating replace and copy operations`
16:  `end if`
17:  `if Special symbols are present then`
18:    `Modify read adding the special symbol information`
19:  `end if`
20: `Append read and position information to data`
21: `end for`
22: `end if`
23: `currentCluster ← currentCluster + 1`
24: `end while`
25: `return data`
Our implementation resolved $getInterval(rname, x, y)$ operations using a similar process to the one just described, with the distinction that after locating $rname$ it move into the bit sequence until the cluster that defines the range that contain $x$. From that point we start extracting the reads information reporting only the reads where their POS field value is between $x$ and $y$. This process continues until reach a position greater than $y$, or until the last cluster related to $rname$ is reached.

4.4 Experimental Results

After analysing all the components of the cluster model and defined compression methodologies for each of them, we compute the space used by the components of Table 4.4, but now using the selected approaches to store the components elements and adding the extra data information needed. All the experimental results described in this section, and in the rest of this thesis, were performed on a computer with Intel(R) Core(TM) i7-2600 processor up to 3.40 GHz, 4 GB of main memory, and 8 MB of cache. The operating system installed was Ubuntu 12.04, version 3.2.0-70-generic Linux kernel. All of our algorithms were implemented using C++, using g++ 4.6.3 compiler version.

To assess the space used by individual components, every time that a read is encoded, the number of bits used by each of its elements are counted. Finally the total sum of bits used by the members of each component is calculate. Table 4.7 shows the sizes, in MB, occupied by each component when the training files are used as input. The bit vector CoRe indicates if a read starts copying or replacing values from the presumed sequence, and the bit vector HasN says if a read contains special symbols or not. Table 4.7 also includes the space used to store the cluster index information (with $\rho = 1000$), which allows us to support random access to the reads. Finally, as a point of comparison, we added to the table the space used when each of the training files are compressed using gzip and bzip2 in the following cases: when RNAME, POS, and Read Sequences fields are treated as separate files; and when all these fields are contained together in one single file. In this work we
utilized GZIP version 1.4 and BZIP2 1.0.6, with arguments set to obtain maximum compression (see Section 2.6).

The results displayed in Table 4.7 show that, for the training files, the proposed approach offers better compression than the generic compression techniques GZIP and BZIP2. Other approaches for compressing the RNAME, POS, and Read Sequence fields (see Section 3.1 and 3.3) are not included, because those algorithms compress whole genomic files without allowing us to isolate these field’s information; or because they compress pure DNA sequences (in general, assuming an alphabet of only four bases), making it impossible to apply their methodologies without modifying the information contained in the input files. Another common theme in the approaches discussed in Section 3.1 is that most of them consider an external reference sequence, which is used to compress and decompress the input files, but it is not accounted for as part of the final storage space. In the compression technique discussed in this section, the “presumed sequence” works as reference sequence, making the suggested solution self-contained and independent of any external reference file.

Compression and decompression times for GZIP, BZIP2, and our solution, when the input data file includes for each line the RNAME, POS, and Read Sequences fields (files symbolized with the extension “.rps”), were also measured. The results are listed in Table 4.8, also showing the average number of bits used to represent each line of the input file, and the average time used to extract each line in the decompression process. The compression and decompression times were obtained by computing the mean and standard deviation of running each process 10 times, after an initial run to prime the cache memory. From Table 4.8 we can notice that while the proposed approach allows a better compression and faster compression time than the general purpose compressors used (GZIP, BZIP2), it does not always offer better decompression time. The cause of why our solution offers faster compression times is because the methodology described does two sequential reads over the input file, where the first pass computes the presumed sequence and the necessary compression parameters, and the second pass directly compresses each line of the file using the information computed. On the other hand the compression methodologies applied
### 4.4 Experimental Results

<table>
<thead>
<tr>
<th></th>
<th>NA12878_low</th>
<th>NA12878_high</th>
<th>HG01477</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Position Offset</td>
<td>2.20</td>
<td>14.51</td>
<td>4.43</td>
</tr>
<tr>
<td>Length Array</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>Presumed Sequence</td>
<td>14.01</td>
<td>14.17</td>
<td>30.54</td>
</tr>
<tr>
<td>Copy array</td>
<td>4.56</td>
<td>63.68</td>
<td>4.11</td>
</tr>
<tr>
<td>Replace array</td>
<td>3.89</td>
<td>65.36</td>
<td>2.66</td>
</tr>
<tr>
<td>Bit vector CoRe</td>
<td>0.39</td>
<td>6.15</td>
<td>0.76</td>
</tr>
<tr>
<td>NotA</td>
<td>1.75</td>
<td>30.39</td>
<td>1.24</td>
</tr>
<tr>
<td>NotC</td>
<td>1.56</td>
<td>26.63</td>
<td>1.00</td>
</tr>
<tr>
<td>NotG</td>
<td>1.50</td>
<td>26.09</td>
<td>0.99</td>
</tr>
<tr>
<td>NotT</td>
<td>1.76</td>
<td>30.63</td>
<td>1.23</td>
</tr>
<tr>
<td>N-array (Positions)</td>
<td>0.30</td>
<td>4.27</td>
<td>0.01</td>
</tr>
<tr>
<td>Bit vector HasN</td>
<td>0.39</td>
<td>6.15</td>
<td>0.76</td>
</tr>
<tr>
<td>Cluster Data</td>
<td>0.48</td>
<td>0.48</td>
<td>1.03</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>32.79</td>
<td>288.51</td>
<td>48.76</td>
</tr>
</tbody>
</table>

|                  |             |              |         |
| Read Sequence Field (GZIP -9) | 35.93    | 270.29       | 64.43   |
| Position Field (GZIP -9)       | 7.64      | 67.73        | 15.42   |
| RNAME field (GZIP -9)          | 0.01      | 0.14         | 0.02    |
| **Total**                     | 43.58     | 338.16       | 79.87   |

|                  |             |              |         |
| Read Sequence Field (BZIP2 -9) | 51.89    | 358.48       | 96.42   |
| Position Field (BZIP2 -9)      | 9.96      | 103.03       | 19.72   |
| RNAME field (BZIP2 -9)         | 0.00      | 0.01         | 0.00    |
| **Total**                     | 61.85     | 461.51       | 116.14  |

|                  |             |              |         |
| SEQ + POS + RNAME (GZIP -9)   | 46.70      | 354.98       | 86.06   |
| SEQ + POS + RNAME (BZIP2 -9)  | 60.39      | 438.86       | 113.32  |

Table 4.7: Size, in MB, used to store the reads, their respective reference name information, and aligned positions. The components in the proposed approach are the same as in Table 4.4, but are compressed using the methodologies chosen after analyzing the input data. The bit vector CoRe indicates if a read starts by copying or replacing values from the presumed sequence, and bit vector HasN shows if a read contains special symbols or not. The Cluster Data row represents the space used by the cluster index information (including the compressed information of the RNAME field). The table also includes the space obtained when the input data is compressed using GZIP and BZIP2 parameterized to obtain their maximum compression.
### 4.4 Experimental Results

#### Chapter 4 Read Sequence Compression

<table>
<thead>
<tr>
<th>File</th>
<th>Method</th>
<th>Bits per line</th>
<th>Compression time (sec)</th>
<th>Decompression time (sec)</th>
<th>Time per line (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA12878_low.rps</td>
<td>Bzip2 -9</td>
<td>154.52</td>
<td>31.39 ± 0.73</td>
<td>9.37 ± 0.46</td>
<td>∼ 2.86</td>
</tr>
<tr>
<td></td>
<td>Gzip -9</td>
<td>119.49</td>
<td>72.58 ± 0.26</td>
<td>5.02 ± 0.63</td>
<td>∼ 1.53</td>
</tr>
<tr>
<td></td>
<td>Our</td>
<td>86.23</td>
<td>15.44 ± 0.07</td>
<td>8.08 ± 0.35</td>
<td>∼ 2.46</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File</th>
<th>Method</th>
<th>Bits per line</th>
<th>Compression time (sec)</th>
<th>Decompression time (sec)</th>
<th>Time per line (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA12878_high.rps</td>
<td>Bzip2 -9</td>
<td>71.37</td>
<td>623.41 ± 7.69</td>
<td>148.06 ± 7.18</td>
<td>∼ 2.87</td>
</tr>
<tr>
<td></td>
<td>Gzip -9</td>
<td>57.73</td>
<td>523.73 ± 8.28</td>
<td>111.46 ± 7.11</td>
<td>∼ 2.16</td>
</tr>
<tr>
<td></td>
<td>Our</td>
<td>47.07</td>
<td>209.27 ± 2.19</td>
<td>101.61 ± 1.86</td>
<td>∼ 1.97</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File</th>
<th>Method</th>
<th>Bits per line</th>
<th>Compression time (sec)</th>
<th>Decompression time (sec)</th>
<th>Time per line (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HG01477.rps</td>
<td>Bzip2 -9</td>
<td>148.35</td>
<td>61.13 ± 0.77</td>
<td>18.65 ± 0.74</td>
<td>∼ 2.91</td>
</tr>
<tr>
<td></td>
<td>Gzip -9</td>
<td>112.66</td>
<td>131.98 ± 0.57</td>
<td>6.62 ± 1.43</td>
<td>∼ 1.03</td>
</tr>
<tr>
<td></td>
<td>Our</td>
<td>66.46</td>
<td>29.02 ± 0.77</td>
<td>14.53 ± 1.47</td>
<td>∼ 2.27</td>
</tr>
</tbody>
</table>

Table 4.8: Compression and decompression times for the training files listed in Table 4.1. The table also includes an estimate of the average time taken to extract each line, and the number of encoded bits used per line. Times are averages plus-minus standard deviations over 10 runs.

by GZIP and BZIP2 (see Section 2.6) involves a continuous search for similarities within the data already compressed (for GZIP) or requires more complex algorithms over the input data (for BZIP2). Meanwhile, a reason why the solution implemented is sometimes slower than GZIP for decompression, is that at decompression time each line is treated independently slowing the process when full decompression is required, but, as it will be discuss, this way of working with the reads permit faster random access to the stored data.

An unusual result in Table 4.7 and 4.8, is that the compression offered by GZIP is lower than BZIP2. We explored the causes of this unexpected behavior by studying how the data was stored in our input data files. A detail about the experimental input data files used (rps files) is that, in general, they consist of several lines, where each line contains the same reference name, a position, and a read which normally is similar to the reads of neighboring lines (typically they overlap). The technique followed by GZIP (Section 2.6.1) is better equipped to take advantage of the overlap between nearby lines, saving space in the compression process, given the smaller distance between string repetitions in the input data, while BZIP2 (Section 2.6.2)
Chapter 4 Read Sequence Compression 4.4 Experimental Results

Figure 4.10: Average bits per line used by GZIP and BZIP2 depending on the offset between consecutive lines, and the coverage using 50 as the offset.

gzip does not make effective use of this information, being unable to exploit the fact that the repetitions are highly localized. To corroborate this assumption we generated different files of one million lines, where each line contained a string of 100 characters (using the standard 128 ASCII symbols). A fixed offset parameter is used to indicate from which point a line is different from its previous line. For example, if the offset is 25, the first 75 characters of each line are equal to the last 75 characters of the previous line, and the next 25 characters are randomly generated. As a second part of this experiment the offset is set to 50 and study what happened if the coverage was increased. That is, if the coverage is 10, then after a line is generated, the line is repeated 10 times before a line is generated with new data. Figure 4.10 shows the average number of bits used to store a line in the generated files, depending on the offset and coverage between consecutive lines.

From the graphs on Figure 4.10 is possible to see that GZIP has a linear behavior between the offset and the number of bits used (note that offset 100 means that all the lines are randomly generated over a 7-bit alphabet, hence the 700 bit per line limiting value). On the other hand, BZIP2 does not offer a good option when the repetitive lines are near each other within the input file. Figure 4.10 also shows that when the coverage of consecutive lines is higher than 200, BZIP2 start to offer a better compression than GZIP. In the input training files used in this chapter, and as Table 4.2 presented, the average coverage are lower than 200 and the offsets, in
general, as Figure 4.5 display, are smaller than 10. That combination leads to gzip offering better compression than bzip2.

Our approach not only offers whole compression and decompression of SAM files, it also supports random access to the information. To test how the solution described (for compressing the RNAME, POS, and Read Sequence fields) randomly accessed the data, we use a modified version of the operation $getInterval($rname, $x, $y)$ introduced in Section 3.5, which, given the parameters $rname$, $x$, and $y$, extracts only the values of the RNAME, POS, and Read Sequences fields from the compressed file, where their values are within the interval indicated by the user-defined parameters. The cluster index of the proposed solution was generated using $\rho = 1000$ as parameter, and the experiments considered two parameters for generating the queries and two ways of how the order in which queries are executed. The parameters are the number of occurrences extracted given an interval ($nc$), and the number of queries to be executed ($nq$). For example, if $nc = 100$ and $nq = 1000$, then the experiment would consist on 1000 randomly select positions intervals, with the corresponding reference name, from the original SAM file, where the number of lines contained in the interval is exactly a 100. The intervals were queried in two ways, in random order, and ordered by their reference name and position. As before, we measure the mean time (including the standard deviation) to extract a line, by running sets of different sample intervals, given a number of occurrences and queries. Each experiment was run 10 times, after an initial run to prime the cache memory. Table 4.9 shows the results obtained.

From the results shown, it is clear that the time used to extract a line decreases rapidly as the number of lines extracted per query increased. The same conclusion can be made about the time taken, when the number of queries processed is increased. In both cases, when the number of lines extracted per query and/or the number of queries performed is increased, the cost of accessing the compressed file is amortized, which is reflected in the results from Table 4.9. We also studied what happens when the query intervals are ordered by reference name and position. Fewer internal jumps in the stored files are needed, and the time decreases to almost half of the time compared to when the queries are performed in random order.
<table>
<thead>
<tr>
<th># queries (nq)</th>
<th>Occurrence per query (nc)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td>NA12878_low</td>
<td></td>
</tr>
<tr>
<td>random sample</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
</tr>
<tr>
<td>ordered sample</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
</tr>
<tr>
<td>NA12878_high</td>
<td></td>
</tr>
<tr>
<td>random sample</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
</tr>
<tr>
<td>ordered sample</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
</tr>
<tr>
<td>HG01477</td>
<td></td>
</tr>
<tr>
<td>random sample</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
</tr>
<tr>
<td>ordered sample</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
</tr>
<tr>
<td></td>
<td>10,000</td>
</tr>
</tbody>
</table>

Table 4.9: Average access time, in microseconds, per line extracted from the compressed representation the training files listed in Table 4.1, given the number of occurrences to be extracted (nc) and the number of intervals consecutively queried (nq). The cluster index parameter used was $\rho = 1000$.  

111
4.5 Summary

This chapter analyzed new methods for storing the RNAME, POS, and Read Sequences fields from SAM files. The described approach compress these fields without using an external reference sequence as input for compressing or decompressing the input data. Also, this implementation creates and stores a presumed sequence, which is used as an artificial reference sequence for the reads to be stored. The presumed sequence is best generated using the most-frequent-base technique, where each base stored symbolizes the more frequent bases found in each of the positions covered by the reads. We described how the reads data, and their respective position and reference name values, are compressed using the presumed sequence approach, showing that the results obtained for the training files, outperformed the compression times and space and decompression times obtained using GZIP and BZIP2.

This chapter also introduced the used of a index over the compressed data,
allowing extraction of reads, positions, and reference names, without decompressing
the whole stored data, given a desirable reference name and position interval. In
Chapter 6 we will discuss in more details how this index is used when entire SAM
files are used as input.
Chapter 5

Quality Field Compression

Section 3.2 described the existing lossless and lossy approaches used to compress the Quality field from FASTQ and SAM files. In this section we introduce new techniques for storing the Quality field, and compare them against the existing methodologies.

For lossy compression mechanisms, it is necessary to define a measure of how the lossy transformation affects the fidelity of the output with respect to the original file. This concept is not new, for example, Ochoa et al. [OAB+13] use mean squared error as a fidelity measure. In the next section we list a range of fidelity criteria that can be applied. A further alternative is to measure the impact of the lossy transformations on the output of downstream applications. This approach is used in the experiment evaluation discussed in Section 5.3.

5.1 Fidelity Measures

When using lossy compression, the maximum degree of information loss that can be tolerated may be specified as an input to the compressor. That requirement presupposes that a measure of loss has been defined, taking as its inputs two data sequences of the type being represented, and providing (using some meaningful
5.2 Block Compression

Table 5.1: Examples of fidelity measures between strictly positive numeric vectors $X$ and $Y$ of length $\ell$ (adapted from [Cha07]). Scores of zero indicate that $X$ and $Y$ are identical except for the Max:Min measure, where one is the minimum value.

<table>
<thead>
<tr>
<th>Measure</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Manhattan Distance</td>
<td>$\frac{1}{\ell} \sum_{i=1}^{\ell}</td>
</tr>
<tr>
<td>Max:Min Distance</td>
<td>$\max_{1 \leq i \leq \ell} \frac{\max(X_i, Y_i)}{\min(X_i, Y_i)}$</td>
</tr>
<tr>
<td>Mean Squared Error</td>
<td>$\frac{1}{\ell} \sum_{i=1}^{\ell} (X_i - Y_i)^2$</td>
</tr>
<tr>
<td>Chebyshev Distance</td>
<td>$\max_{1 \leq i \leq \ell}</td>
</tr>
<tr>
<td>Soergel Distance</td>
<td>$\frac{\sum_{i=1}^{\ell}</td>
</tr>
<tr>
<td>Lorentzian Distance</td>
<td>$\sum_{i=1}^{\ell} \log_2(1 +</td>
</tr>
</tbody>
</table>

5.2 Block Compression

We introduce a new compression approach that, as in Kozanitis et al. (Section 3.2.1) and Ochoa et al. (Section 3.2.4), is based on local properties of the quality scores.
The methodology proposed exploits the observation that neighboring quality scores are generally numerically close. The algorithm separates the quality scores into blocks of variable size, where all the values contained within each block comply with a chosen parameter $\theta$ according to some measure criterion (see Table 5.1). For each block the length and a representative value are stored, where the representative value depends on which criterion is used. This approach ensures that all the values stored in a block are limited by a user-defined parameter and a given per-value distance criterion.

Algorithm 9 gives pseudocode for computing the block sizes and representative values. The notation “[ ]” symbolize the empty list, and the “++” operator is used to append a single value to a list of the same type of object. The algorithm...
5.2 Block Compression

Chapter 5 Quality Field Compression

sequentially visits each quality score of the input quality line, and as it does, the method *FulfillCriteria* internally checks if it is possible to include that quality score into the current block, or whether it is necessary to start a new block. That is, given a block of values, which already fulfills a user-defined criteria (for example, a fidelity measure between the modified values and the original values), and a new value, the function *FulfillCriteria* determines if the criteria is still valid if the new value is added to block. Each block is represented with a single value, so if we calculate the fidelity measure related to the criteria chosen, between the original values of the block and a vector of the same size where all the values are the representative value of the block, then this value must satisfy the user criteria. Method *CalculateRepresentative* computes the best representative value for a given block depending on the criteria, supposing that at least one valid suitable value exists. In the following sections the *FulfillCriteria* and *CalculateRepresentative* process used in this thesis are described in more detail.

After the algorithm has been applied to a quality line, the elements in the *Representatives* array need to be stored. For example, if the array size is $k$, then the elements can be stored as minimum and maximum, followed by the $k$ suitable values chosen in the defined range using a *Binary* code of $\lceil \log_2(\max_k *Representatives - \min_k *Representatives + 1) \rceil$ bits per value, and storing the *RunLengths* array using a *Gamma* code. In Section 5.3 we explore these and other methodologies for storing these two arrays.

Note that the criterion or formula used in the method *CalculateRepresentative* does not need to be same as the criterion used in the method *FulfillCriteria*. This allows different criteria to be chosen depending on the use of quality scores after being compressed. In this thesis we introduce two modalities of this algorithm, which we refer as P-BLOCK and R-BLOCK.

5.2.1 P-Block

The first modality studied, the P-BLOCK mechanism, uses *Mean Manhattan Distance* as criterion for *FulfillCriteria* and *CalculateRepresentative*, and controls it
Chapter 5 Quality Field Compression

5.2 Block Compression

Figure 5.1: Example of the quality scores block division, using P-Block when \( p = 1 \), and it’s respective representative values and run-lengths arrays. In this example the first quality score with value 72 must be placed into a new block because 69 to 71 are already needing to be covered, and this value causes the end of the first block.

with a user-define parameter \( \theta = p \). That is, each block must satisfy that when the Mean Manhattan Distance is computed between the original values of the block and the transformed values (in this approach all the transformed values within a block are the same), the distance obtained must be lower or equal to \( p \). A sufficient, but not necessary, way to achieve this condition is that none of the values in each block are more than \( p \) from the corresponding representative value. Thus, the difference between the maximum and minimum value of each block is lower than or equal to \( 2 \cdot p \), and the representative value is computed as the midpoint between the minimum and maximum values of the block. Figure 5.1 illustrates the application of the P-BLOCK to an example sequence, using \( p = 1 \).

In this approach \( \text{FulfillCriteria}(qvals, p) \) is implemented by computing

\[
\max_i qvals[i] - \min_i qvals[i] \leq 2 \cdot p,
\]

where the range of \( i \) is \( 0 \leq i < |qvals| \). Within a set that meets this criterion, the value

\[
rep = \left\lfloor \frac{\max_i qvals[i] + \min_i qvals[i]}{2} \right\rfloor
\]
guarantees that the \textit{Mean Manhattan Distance} for the whole sequence will be not greater than $p$. To demonstrate that $rep$ satisfies that the distance with respect all the quality scores of the block to be encoded is no greater than $p$, we need to consider the cases when the distance between the maximum and minimum quality score within a block is an even or odd number. If the distance is an even number, then the two values are $2 \cdot x$ from each other, and by definition $rep$ (which is the middle point) is at $x$ from the minimum and maximum quality scores. Given how \textit{FulfillCriteria} is implemented obviously $x \leq p$. Otherwise, if the distance between the minimum and maximum quality scores within a block is an odd number, it is possible to demonstrate using a similar analysis that the middle point chosen as the representative is $x - 1$ from the minimum value and $x$ from the maximum value, with $x \leq p$. This shows that $rep$ fulfills the condition that

$$\forall i \in [0, \ell) \ |qvals[i] - rep| \leq p.$$ 

Note that for any given block there may also be other integer values $rep'$ for which

$$\forall j \in [0, \ell) \ |qvals[j] - rep'| \leq p,$$

and in that case, method \textit{CalculateRepresentative} is free to use a secondary criterion, such as minimizing the average distance between the selected representative and the block’s values. When $p$ is larger than or equal to the difference between the maximum and minimum quality score of a quality line divided by two, only one representative value is stored for that line. For example, in Figure 5.1, if $p \geq 4$ all the quality scores are blocked together, and can be represented using the value 70 with runlength of 16.

\section{5.2.2 R-Block}

The second approach proposed recognizes that in Bioinformatics it may be preferable to be more precise in representing low quality scores (that is, when the corresponding base is assessed as being more likely to be in error) than high ones.
Following this idea, the R-BLOCK mechanism uses Max:Min Distance metric as criterion for the method FulfillCriteria and controls it with a user-defined parameter $\theta = r$. That is, for each block, R-BLOCK checks if a representative value exists such that the Max:Min Distance is not higher than $r$ for all the values within the block when they are compared with respect to the representative value. Given that the quality scores are a Phred scaled base error, we subtract the scale factor (33) from the quality scores, so the values start from 0 (without loss of generality in this thesis we assumed that in the worst case the normalized quality score is 1). This normalization of the values needs that, depending on the parameter $r$, the transformation of lower quality scores happen rarely (unless $r$ is assigned a high value).

In this case $FulfillCriteria(qvals, r)$, if we are using Phred+33 quality scores, needs to check if an integral $rep$ exists such that:

$$(\max_i qvals[i] - 33)/(rep - 33) \leq r \text{ and } (rep - 33)/(\min_i qvals[i] - 33) \leq r$$

If the value exists, it can be calculated as:

$$rep = \left\lfloor \sqrt{(\max_i qvals[i] - 33) \cdot (\min_i qvals[i] - 33)} + 33 \right\rfloor$$

In this case the values of $rep$ is computed using the floor of the previous equation, as in general is better to avoid to transform quality scores to higher values than the original. Assuming that all the values in a block are normalized, it is possible to demonstrate that the representative value chosen, satisfies the Max:Min Distance for that block. That is,

$$\forall i \in [0, \ell) \frac{\max(qvals[i], rep)}{\min(qvals[i], rep)} \leq r .$$

In order to prove this, we have to analyze the two extreme cases, when $qvals[i]$ takes values $\max_j qvals[j]$ and $\min_j qvals[j]$. For the first case, replacing $rep$ by its value, we obtain the following result,
5.3 Experimental Evaluation

Chapter 5 Quality Field Compression

\[ \max_{1 \leq i \leq t} \frac{\max(X_i, rep)}{\min(X_i, rep)} = \frac{\max_j qvals[j]}{\text{rep}} = \sqrt{\frac{\max_j qvals[j]}{\min_j qvals[j]}}, \]

which, given the conditions that \( \max_i qvals[i]/\text{rep} \leq r \) and \( \text{rep}/\min_i qvals[i] \leq r \), we obtain that,

\[ \sqrt{\frac{\max_j qvals[j]}{\min_j qvals[j]}} \leq \sqrt{\frac{r \cdot \text{rep}}{\min_j qvals[j]}} \leq \sqrt{\frac{r \cdot \text{rep}}{\text{rep}/r}} = r. \]

Using the same analysis it is possible to prove this inequality when \( X_i = \min_j qvals[j] \).

As for P-BLOCK, \textit{CalculateRepresentative} is free to use a secondary criterion to decide which value use as representative. In our case we do not apply another criteria, other than always assign the lower valid representative given a block, in order to try to avoid too many quality scores being transformed to higher values. Figure 5.2 illustrates the application of the R-BLOCK to an example sequence, using \( r = 1.055 \).

When \( r \) is sufficiently large, only one value is stored for each read, symbolizing every quality score in that alignment. For example, in Figure 5.2, if \( r \geq 1.06 \) all the quality scores are blocked together, and can be represented using the value 69 with a runlength of 16.

5.3 Experimental Evaluation

This section focuses on analyzing the performance of lossy compression methodologies for quality scores, and their impact on the process of variation calling (see Section 3.4). To compare the effect of lossy compression on the Quality field we compute the difference between the quality scores of the original file and the quality scores generated after compressing and decompressing the file using different approaches. The criteria measures listed in Table 5.1 were used, showing the trade-
Chapter 5 Quality Field Compression  5.3 Experimental Evaluation

Figure 5.2: Quality scores being formed into blocks using the R-Block transformation when \( r = 1.055 \). The first step consists in normalizing the quality scores, and then calculate the blocks over this new sequence. In this example the first normalized quality score with value 35 must be placed into a new block because, while it exists a valid representative value for the interval 37 to 39 when \( r = 1.055 \), there is not one if the interval start from 35.

off between space and fidelity measure offered by each of the methodologies. All the results obtained are computed by measuring each criteria of each quality line in the file and then calculating the average across these values.

The implementations of the proposed solutions (P-Block and R-Block) also incorporates an index that allows random access to the compressed Quality field. The generated index, given a user-defined parameter \( \mu \), contains synchronization points to the compressed data every \( \mu \) compressed lines, providing fast random access to blocks. That is, for every \( \mu \cdot x \) line, where \( x \) is an integer bigger than zero, the index stores the start encoding positions of these lines, allowing the decoder to start extracting data from these points instead of requiring whole file decompression. The space used to stored the index is directly proportional to the parameter \( \mu \) and the number of lines in the file. In this thesis the index information used an insignificant fraction of the total space required by the final compressed file. In this chapter, given that random access to only quality scores is not required, we set \( \mu = 1000 \).
5.3 Experimental Evaluation

<table>
<thead>
<tr>
<th>File</th>
<th>Size (MB)</th>
<th>Quality lines</th>
<th>Line length</th>
<th>$H_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>NA12878.low.quality</td>
<td>318.94</td>
<td>3,278,702</td>
<td>101</td>
<td>4.029</td>
</tr>
<tr>
<td>NA12878.high.quality</td>
<td>5,017.98</td>
<td>51,585,658</td>
<td>101</td>
<td>3.384</td>
</tr>
<tr>
<td>HG01477.quality</td>
<td>617.22</td>
<td>6,407,925</td>
<td>100</td>
<td>3.937</td>
</tr>
<tr>
<td>chr20.fa</td>
<td>61.31</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>chr11.fa</td>
<td>131.33</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Homo_sapiens_assembly19.fa</td>
<td>2,995.26</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.2: Training quality files. The quality scores were extracted from three SAM files, NA12878.chrom20.ILLUMINA.bwa.CEU.low_coverage.20120522.sam, NA12878.HiSeq.WGS.bwa.cleaned.recal.hg19.20.sam, and HG01477.chrom11.ILLUMINA.bwa.CLM.low_coverage.20130415.sam; the reference chromosomes (chr11 and chr20) were obtained from http://hgdownload.cse.ucsc.edu/goldenPath/hg19/chromosomes/. Also as reference we use the Homo_sapiens_assembly19.fa which can be found at http://www.broadinstitute.org/ftp/pub/seq/references/. The columns show the size of the quality scores as ASCII bytes, the number of reads, the length of each read, and the zero-order entropy ($H_0$, in bits per quality score) of the quality score sequence.

Later, in Section 6.3 it will be explained how this index is used and why $\mu = 1000$ is a suitable parameter for the solution presented.

As training files we use the quality scores of the files present in Section 4.2. Table 5.2 shows more details of the quality scores components of these sources.

The first step for storing the quality files is to study the data contained on the files and their properties. As shown in Table 5.2 the zero-order entropy of each file is around 4 bits per quality score, meaning that it should be possible to losslessly compress the quality scores using codewords of length close to that number of bits per value. A second analysis step explores how the values are distributed in the quality lines and, assuming that all the lines in a file have the same length, also calculates the average quality score per position in the line. Figure 5.3 shows a histogram of the quality scores in the data files, and Figure 5.4 displays the average value per position over all the quality lines in each training file. As shown in the figures, the quality scores of the files are mainly between 60 and 75, with an isolated peak at 33. Also from Figure 5.4 it can be observed that most of the higher values are in general in the middle of the quality lines, and that the start and end of each line tend to contain lower quality scores.
Chapter 5 Quality Field Compression

5.3 Experimental Evaluation

Figure 5.3: Histogram of the quality scores for each of the training files listed in Table 5.2. The vertical axis shows the frequency of each quality score represented as a percentage over the total number of quality scores.

Figure 5.4: Average quality score per position for NA12878_low.quality, NA12878_high.quality, and HG01477.quality files. The mean standard deviation of the values at each position is 12.6 for NA12878_low.quality, 10.6 for NA12878_high.quality, and 6.3 for HG01477.quality.
5.3 Experimental Evaluation  

5.3.1 Coding Methods

Before comparing the approaches described in the previous section, we study different ways to code the Block Compression components. While the RunLengths array values are stored using Elias Gamma code (see Section 2.3), three different techniques are proposed for representing the Representatives array values: as plain ASCII bytes; as BINARY values using the number of bits indicated by the range of quality scores stored in the whole file; and as BINARY values using per line parameters, with two additional Gamma coded values per line to indicate the lower bounds of the local BINARY code and the number of bits needed to store each of the local values. Table 5.3 and 5.4 shows the space in MB used for each of these methodologies over the sample files using P-Block and R-Block respectively.

Figure 5.5 illustrates the difference in size (measure in MB) of using BINARY Global and BINARY Local techniques to store the Representatives. As can be seen, when the parameter used is smaller than 4 for P-Block, and smaller than 1.2 for R-Block, the BINARY Local approach is preferable. In the sample files from Table 5.2 is possible to notice that overall BINARY Global presents a better alternative to store the Representatives array when the parameter for the Block Compression methodology used is high.
### Table 5.3: Storage space required by lossy P-Block using Gamma code to store RunLengths and Representatives using three different methodologies. As a reference point, we also include the space required if the quality scores are losslessly compressed using a zero order model combined with a perfect entropy coder, which is an estimate of the losslessly compressed size of the data. While this methodology is not meant to be used in a lossless way, this table includes the case when \( p = 0 \).

<table>
<thead>
<tr>
<th>Component</th>
<th>Code</th>
<th>NA12878_low Size (MB)</th>
<th>NA12878_high Size (MB)</th>
<th>HG01477 Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fidelity parameter ( p = 0 )</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RunLengths array</td>
<td>Gamma</td>
<td>39.8</td>
<td>626.9</td>
<td>83.2</td>
</tr>
<tr>
<td>Representatives array</td>
<td>ASCII</td>
<td>224.9</td>
<td>3,270.7</td>
<td>491.0</td>
</tr>
<tr>
<td>Representatives array</td>
<td>Binary (global)</td>
<td>168.7</td>
<td>2,453.0</td>
<td>368.3</td>
</tr>
<tr>
<td>Representatives array</td>
<td>Binary (local)</td>
<td>143.0</td>
<td>1,977.7</td>
<td>295.3</td>
</tr>
<tr>
<td><strong>Fidelity parameter ( p = 1 )</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RunLengths array</td>
<td>Gamma</td>
<td>36.8</td>
<td>497.5</td>
<td>81.5</td>
</tr>
<tr>
<td>Representatives array</td>
<td>ASCII</td>
<td>119.0</td>
<td>1,465.0</td>
<td>271.4</td>
</tr>
<tr>
<td>Representatives array</td>
<td>Binary (global)</td>
<td>89.2</td>
<td>1,098.8</td>
<td>203.6</td>
</tr>
<tr>
<td>Representatives array</td>
<td>Binary (local)</td>
<td>79.2</td>
<td>960.3</td>
<td>166.7</td>
</tr>
<tr>
<td><strong>Fidelity parameter ( p = 2 )</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RunLengths array</td>
<td>Gamma</td>
<td>27.3</td>
<td>352.8</td>
<td>66.6</td>
</tr>
<tr>
<td>Representatives array</td>
<td>ASCII</td>
<td>69.1</td>
<td>828.6</td>
<td>160.8</td>
</tr>
<tr>
<td>Representatives array</td>
<td>Binary (global)</td>
<td>51.8</td>
<td>621.4</td>
<td>120.6</td>
</tr>
<tr>
<td>Representatives array</td>
<td>Binary (local)</td>
<td>48.7</td>
<td>585.3</td>
<td>102.3</td>
</tr>
<tr>
<td><strong>Fidelity parameter ( p = 4 )</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RunLengths array</td>
<td>Gamma</td>
<td>21.0</td>
<td>247.8</td>
<td>38.0</td>
</tr>
<tr>
<td>Representatives array</td>
<td>ASCII</td>
<td>38.8</td>
<td>418.0</td>
<td>60.5</td>
</tr>
<tr>
<td>Representatives array</td>
<td>Binary (global)</td>
<td>29.1</td>
<td>313.5</td>
<td>45.4</td>
</tr>
<tr>
<td>Representatives array</td>
<td>Binary (local)</td>
<td>28.9</td>
<td>321.9</td>
<td>43.6</td>
</tr>
<tr>
<td>( H_0 \cdot Total\ number\ of\ quality\ scores )</td>
<td>159.1</td>
<td>2,101.8</td>
<td>300.7</td>
<td></td>
</tr>
</tbody>
</table>
### Table 5.4: Storage space required by lossy R-Block using Gamma code to store RunLengths and Representatives using three different methodologies. As a reference point, we also include the space required if the quality scores are losslessly compressed using a zero order model combined with a perfect entropy coder, which is an estimate of the losslessly compressed size of the data. While this methodology is not meant to be used in a lossless way, this table includes the case when $r = 1.0$.

<table>
<thead>
<tr>
<th>Component</th>
<th>Code</th>
<th>NA12878\textsubscript{low} Size (MB)</th>
<th>NA12878\textsubscript{high} Size (MB)</th>
<th>HG01477 Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fidelity parameter $r = 1 + 0.00$</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RunLengths array</td>
<td>GAMMA</td>
<td>39.8</td>
<td>626.9</td>
<td>83.2</td>
</tr>
<tr>
<td>Representatives array</td>
<td>ASCII</td>
<td>224.9</td>
<td>3,270.7</td>
<td>491.0</td>
</tr>
<tr>
<td>Representatives array</td>
<td>BINARY (global)</td>
<td>168.7</td>
<td>2,453.0</td>
<td>368.3</td>
</tr>
<tr>
<td>Representatives array</td>
<td>BINARY (local)</td>
<td><strong>143.0</strong></td>
<td>1,977.7</td>
<td><strong>295.3</strong></td>
</tr>
<tr>
<td><strong>Fidelity parameter $r = 1 + 0.05$</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RunLengths array</td>
<td>GAMMA</td>
<td>29.0</td>
<td>496.9</td>
<td>79.6</td>
</tr>
<tr>
<td>Representatives array</td>
<td>ASCII</td>
<td>90.9</td>
<td>1,472.4</td>
<td>261.9</td>
</tr>
<tr>
<td>Representatives array</td>
<td>BINARY (global)</td>
<td>68.2</td>
<td>1,104.3</td>
<td>196.4</td>
</tr>
<tr>
<td>Representatives array</td>
<td>BINARY (local)</td>
<td><strong>62.6</strong></td>
<td><strong>965.6</strong></td>
<td><strong>160.5</strong></td>
</tr>
<tr>
<td><strong>Fidelity parameter $r = 1 + 0.1$</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RunLengths array</td>
<td>GAMMA</td>
<td>23.3</td>
<td>291.3</td>
<td>50.4</td>
</tr>
<tr>
<td>Representatives array</td>
<td>ASCII</td>
<td>54.8</td>
<td>655.7</td>
<td>102.3</td>
</tr>
<tr>
<td>Representatives array</td>
<td>BINARY (global)</td>
<td>41.1</td>
<td>491.8</td>
<td>76.7</td>
</tr>
<tr>
<td>Representatives array</td>
<td>BINARY (local)</td>
<td><strong>39.7</strong></td>
<td><strong>479.7</strong></td>
<td><strong>68.8</strong></td>
</tr>
<tr>
<td><strong>Fidelity parameter $r = 1 + 0.2$</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RunLengths array</td>
<td>GAMMA</td>
<td>13.3</td>
<td>182.2</td>
<td>25.2</td>
</tr>
<tr>
<td>Representatives array</td>
<td>ASCII</td>
<td>21.5</td>
<td>296.9</td>
<td>35.2</td>
</tr>
<tr>
<td>Representatives array</td>
<td>BINARY (global)</td>
<td><strong>16.1</strong></td>
<td><strong>222.7</strong></td>
<td><strong>26.4</strong></td>
</tr>
<tr>
<td>Representatives array</td>
<td>BINARY (local)</td>
<td>17.7</td>
<td>243.5</td>
<td>28.1</td>
</tr>
</tbody>
</table>

$H_0 \cdot \text{Total number of quality scores}$  

159.1, 2,101.8, 300.7
For the rest of this thesis P-BLOCK and R-BLOCK will code the RunLengths using Gamma code and Representatives using the best approach possible depending of the parameter chosen. In other words, if $p < 4$ or $r < 1.2$ we will use Binary Local, otherwise the Representatives values are coded using Binary Global.

The implementations explored in the experiments described next, include Wan et al.’s LogBinning and UniBinning [WAA12]; the QualComp mechanisms introduced by Ochoa et al [OAB+13]; and our P-BLOCK and R-BLOCK solutions. Results using the CRAMTools lossy models [FLCB11] are also included, in recognition of their extensive use in data repositories such as the 1000 Genomes Project. Finally, to establish a benchmark for lossless compression, compression using gzip is also added to the experimental results.

Each of the implementations provides different trade-offs that adjust the balance between fidelity and compression rate. The graphs are plotted using QualComp rates $r = \{0, 0.5, 1, 2\}$; P-BLOCK parameters $p = \{1, 2, 4, 8, 16, 32\}$; R-BLOCK ratios $r = 1 + \{0.05, 0.1, 0.2, 0.4, 0.8, 1.6, 3.2, 6.4\}$; UniBinning thresholds $b = \{80, 100\}$; and LogBinning thresholds $b = \{5, 10, 20, 30, 40, 60\}$. For CRAM, lower bound mapping qualities of 40, 50, and 60 were used with the Preserve, Bin-Preserve and Match-Bin-Preserve modes.

5.3.2 Distance Measure Experiments

No single one of the metrics listed in Table 5.1 can be argued as being the “best” to compare lossy transformation techniques, therefore we explored results using a spectrum of criteria and analyzed the results obtained.

Figure 5.6 shows the various trade-offs measured when fidelity is assessed using Mean Manhattan Distance (Figure 5.6a) and Chebyshev Distance (Figure 5.6b). For these two criteria P-BLOCK, R-BLOCK, and QualComp offer better trade-offs than the rest of the approaches. Figure 5.6b also illustrate that QualComp, given that it assumes a Gaussian distribution over the quality scores, normally had a bigger difference between the maximum and minimum value stored per alignments.
Figure 5.6: Average Manhattan Distance and Chebyshev Distance versus space trade-offs for the quality score sequence NA12878_low.quality, NA12878_high.quality, and HG01477.quality. In the case of P-Block and R-Block, low values of the parameter rise to points in the top left; as the parameter is increased, the distance measure decreases, and the space required to store the quality scores increases. The lower left quadrant represent the desirable zone.
Figure 5.7: Average Max:Min Distance and Mean Squared Error versus space trade-offs for the quality score sequence NA12878_low.quality, NA12878_high.quality, and HG01477.quality. In the case of P-Block and R-Block, low values of the parameter rise to points in the top left; as the parameter is increased, the distance measure decreases, and the space required to store the quality scores increases. The lower left quadrant represent the desirable zone.
Figure 5.8: Average Soergel Distance and Lorentzian Distance versus space trade-offs for the quality score sequence NA12878_low_quality, NA12878_high_quality, and HG01477_quality. In the case of P-BLOCK and R-BLOCK, low values of the parameter rise to points in the top left; as the parameter is increased, the distance measure decreases, and the space required to store the quality scores increases. The lower left quadrant represent the desirable zone.
Figure 5.7 displays the same trade-off curves for Max:Min Distance (Figure 5.7a) and the Mean Squared Error (Figure 5.7b). Both measure criteria favor a particular method, with QualComp minimizing the Mean Square Error and R-Block controlling the maximum Max:Min Distance allowed. As expected, QualComp offers a competitive trade-off when the measure criterion used is the Mean Squared Error but does not perform well with the Max:Min Distance. Also, as expected, R-Block outperforms all the other methods when the criteria used is the Max:Min Distance, and both of our approaches displayed trade-offs competitive with QualComp for the Mean Squared Error criterion.

Figure 5.8 shows the trade-offs when the fidelity measures are the Soergel Distance (Figure 5.8a) and the Lorentzian Distance (Figure 5.8b). The graphs and trade-offs obtained using Soergel Distance are similar to the ones of Manhattan Distance (Figure 5.6a), which was expected given the similarity of how this two criteria are computed. In the other hand, note that calculate the Lorentzian Distance between two vectors is equivalent to compute the number of bits needed to code the differences over the values being compared. Figure 5.8b shows that for the Block Compression approaches, the larger the parameter $\theta$, the bigger the Lorentzian Distance is between the original quality score sequences and sequence on the lossy transforms quality files. In the case of QualComp the lossy quality scores are restricted to minimize the Mean Squared Error and the number of bits chosen to code the quality scores given an arbitrary parameter.

From these experiments we conclude that the QualComp, P-Block, and R-Block techniques outperform the other methodologies studied. The two new approaches, P-Block and R-Block, are particularly-well suited to the Max:Min Distance, as was the intention; while Ochoa et al.’s QualComp offers good performance when fidelity is quantified using Mean Squared Error. It is worth noting that QualComp allows a further modality, allowing the quality file to be separated into $c$ sub-files applying the same algorithm to each file separately. The results obtained via this approach (separating the file into 2 or 3, as in Ochoa et al. [OAB+13]) did not result in any significant difference compared with the other methodologies explored.
5.3 Experimental Evaluation

5.3.3 Downstream Application: Variation Calling

The previous section showed that P-BLOCK, R-BLOCK, and QUALCOMP always offer better trade-offs between space and fidelity between all the methodologies studied in this thesis. This section study the impact of lossy transforms on the quality scores in terms of their subsequent use. As discussed in Section 3.4, an important application of alignment sequence data is to find variations between the alignment sequence and a reference sequence. This variation calling process makes use of the aligned sequences and their associated quality score sequences, to find differences against a reference, which may contain valuable information about known and unknown variations that, for example, help in the study of diseases [NPAS11]. The quality scores indicate how accurately the related bases were identified and aligned, helping to differentiate between real variants and possible computing errors at the time of calculate the quality scores.

To analyses the effect of lossy transform the files we compute the VCF files (see Section 3.4) from each lossy file generated, and compare it with the VCF generated using the original file. To compare two VCF files we adapt the methodology used by Ochoa et al. [OAB+13], and define a true positive (TP) as a variation that is found in both VCF files; a true negative as a location at which neither file records a variation; a false negative (FN) as a variation that is only found in the VCF of the lossless file; and a false positive (FP) as a variation that is found only in the VCF of the lossy transformed file. Figure 5.9 illustrates how TP, TN, FN, and FP variants are related. Ochoa et al. further separate the TP values into half and equal TP, where equality is registered if the same variation is found in both VCF files, half is registered if a variation is found in the same position in both files, but the type of variation is not exactly the same. In all the experiments in this sections the amount of “half TP” was no higher than 0.004 per cent of the total TP value, and they were included as part of the FP count without affecting the results.

Using TP, FP, TN, and FN we can compute four useful quantities: precision (fraction of variations in the lossy file that match the original); recall (fraction of the variations in the original file that are found in the lossy VCF file); true negative rate
Figure 5.9: Distribution of $TN$, $TP$, $FN$, and $FP$ of variants reported between an original file and its lossy representation.

(proportion of negatives variations which are correctly identified as such); and $F$-Score (the harmonic mean of precision and recall). These four scores can then be used as a comparison against the results generated by the original sequence. Mathematically these values are expressed as:

$$\text{precision} = \frac{TP}{TP + FP} \quad \text{recall} = \frac{TP}{TP + FN} \quad \text{true negative rate} = \frac{TN}{TN + FP}$$

$$F$-Score = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Many of the lossy methodologies discussed in Section 3.2 offer a “fully lossy” compression where all the quality scores are ignored and replaced by a predefined value. Normally all the quality scores are replaced by just one possible value, for example CRAMTOOLS replaces all quality scores with “?”, assigning low quality score to all the reads, but not too low to be considered wrongly aligned. We first study how the variant calling process is affected when all the quality scores of each file are changed into a single value. Figure 5.10 presents the precision/recall trade-off curves obtained when all the quality scores are replaced by a single value. When the quality score is small the variation calling process found fewer variations, mainly calling the variation found without using the quality information. These cases have higher precision but lower recall. However, when the quality score is higher, the
number of variations found grows, offering high recall but low precision.

To test the performance of each of the lossy compression approaches, we selected a range of the methodologies studied in the previous section and applied their techniques to the sample files. Table 5.5 lists the precision and recall scores for the selected range of methods applied to the NA12878.low.quality file; and Figure 5.11 plots the relationship between the corresponding F-Score and the size (in megabytes) of the stored data for each method. Similar results were obtained using the HG01477.quality file, included in Table 5.6 and Figure 5.12. While the results showed for NA12878.high.quality are similar to the previous presented, we selected some extra trade-off alternative points to Table 5.7 and Figure 5.13 in order to show that the Block Compression approach proposed attains recall and precision over 99 per cent with a lower memory space usage than the other solutions.

Note that Figures 5.11 and 5.12 include an additional point, labeled ONEQUAL, showing what would happen if no quality scores at all were stored, and every quality
score was assumed to be the same, which was calculated as the average between all the quality scores into the file. To understand the effect of **OneQual**, refer back to Figure 5.10.

As Figures 5.11 and 5.12 show, **QualComp**, **P-Block**, and **R-Block** outperform other techniques for a given level of precision or recall, and allow more compact storage of quality scores than do other mechanisms. This experiment also demonstrate that **P-Block** and **R-Block** allow a wide range of alternatives which permit low storage space and over 99.0 per cent recall and precision in variation detection compared with the lossless alternative. Other approaches can secure low storage space, over 99.0 per cent of precision or/and recall, but not all three simultaneously. Tables 5.5, 5.6, and 5.7 permit this observation to be assessed in terms of precision and recall: if 99.0 per cent is regarded as being a minimum threshold for having high-confidence in the outcomes, then of the methods compared, **R-Block** with $r = 1.4$ generates the most compact representation for **NA12878_low_quality**, and **P-Block** with $p = 4$ and $p = 6$ in the case of
Figure 5.12: F-Score and effect of lossy compression techniques on VCF computation for the sample file HG01477.quality. Each line represents a different lossy compression method.

Figure 5.13: Effect of lossy compression techniques on VCF computation for the sample file NA12878.high.quality. Each line represents a different lossy compression method.
Having selected parameters on the three training files, we now compute the trade-off between size and F-Score for 10 new files, choosing a representative of each approach based on the experiments over the sample files. Given the results attained in the training files, it is possible to conclude that the first method that achieve precision and recall over 99.0 per cent for all the sample files, is \( r = 1.4 \) for R-Block and \( p = 6 \) for P-Block. To compare these approaches we computed the F-Score and sizes used by QUALCOMP when \( r \) is equal to 0.5 and 1. Note that \( r = 0.5 \) is a point close to the lower space achieved with R-Block and \( r = 1 \) is a point that uses more space than the obtained using \( p = 6 \) in P-Block. Table 5.8 lists the 10 files used, including the sources and accessed date.

Figure 5.14 shows how the approaches chosen perform over the files listed in Table 5.8. As expected, while QUALCOMP ensures the space consumption given the parameter chosen, it does not obtain an F-Score over 99 per cent for all the files.
<table>
<thead>
<tr>
<th>Method</th>
<th>Size (MB)</th>
<th>Variations</th>
<th>TP</th>
<th>FN</th>
<th>FP</th>
<th>Recall</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONEQual, ( q = \text{&quot;C&quot;} = 67 )</td>
<td>0.0</td>
<td>69,762</td>
<td>59,615</td>
<td>331</td>
<td>10,147</td>
<td>99.4</td>
<td>85.5</td>
</tr>
<tr>
<td>R-Block, ( \theta = 4.2 )</td>
<td>8.3</td>
<td>58,681</td>
<td>57,969</td>
<td>1,977</td>
<td>712</td>
<td>96.7</td>
<td>98.8</td>
</tr>
<tr>
<td>P-Block, ( p = 16 )</td>
<td>9.3</td>
<td>61,006</td>
<td>59,383</td>
<td>563</td>
<td>1,623</td>
<td>99.1</td>
<td>97.3</td>
</tr>
<tr>
<td>R-Block, ( \theta = 1.4 )</td>
<td>15.9</td>
<td>60,108</td>
<td>59,504</td>
<td>442</td>
<td>604</td>
<td>99.3</td>
<td>99.0</td>
</tr>
<tr>
<td>QualComp, ( r = 0.5 )</td>
<td>19.7</td>
<td>61,836</td>
<td>59,590</td>
<td>356</td>
<td>2246</td>
<td>99.4</td>
<td>96.4</td>
</tr>
<tr>
<td>P-Block, ( p = 6 )</td>
<td>28.1</td>
<td>60,082</td>
<td>59,537</td>
<td>409</td>
<td>545</td>
<td>99.3</td>
<td>99.1</td>
</tr>
<tr>
<td>QualComp, ( r = 1 )</td>
<td>39.5</td>
<td>60,907</td>
<td>59,636</td>
<td>310</td>
<td>1271</td>
<td>99.5</td>
<td>97.9</td>
</tr>
<tr>
<td>UniBinning, ( b = 100 )</td>
<td>46.5</td>
<td>51,885</td>
<td>51,664</td>
<td>8,282</td>
<td>221</td>
<td>86.2</td>
<td>99.6</td>
</tr>
<tr>
<td>P-Block, ( p = 4 )</td>
<td>50.2</td>
<td>60,044</td>
<td>59,703</td>
<td>243</td>
<td>341</td>
<td>99.6</td>
<td>99.4</td>
</tr>
<tr>
<td>LogBinning, ( b = 5 )</td>
<td>54.6</td>
<td>57,074</td>
<td>56,873</td>
<td>3,073</td>
<td>201</td>
<td>94.9</td>
<td>99.6</td>
</tr>
<tr>
<td>CRAM Bin-Preserve, 50</td>
<td>56.2</td>
<td>52,518</td>
<td>51,417</td>
<td>8,529</td>
<td>1,101</td>
<td>85.8</td>
<td>97.9</td>
</tr>
<tr>
<td>CRAM Match-Bin-Preserve, 50</td>
<td>57.4</td>
<td>52,952</td>
<td>49,111</td>
<td>10,835</td>
<td>3,841</td>
<td>81.9</td>
<td>92.7</td>
</tr>
<tr>
<td>R-Block, ( \theta = 1.10 )</td>
<td>63.0</td>
<td>59,892</td>
<td>59,708</td>
<td>238</td>
<td>184</td>
<td>99.6</td>
<td>99.7</td>
</tr>
<tr>
<td>P-Block, ( p = 2 )</td>
<td>76.0</td>
<td>59,957</td>
<td>59,769</td>
<td>177</td>
<td>188</td>
<td>99.7</td>
<td>99.7</td>
</tr>
<tr>
<td>QualComp, ( r = 2 )</td>
<td>79.0</td>
<td>60,247</td>
<td>59,693</td>
<td>253</td>
<td>554</td>
<td>99.6</td>
<td>99.0</td>
</tr>
<tr>
<td>LogBinning, ( b = 20 )</td>
<td>91.4</td>
<td>59,340</td>
<td>59,247</td>
<td>699</td>
<td>93</td>
<td>98.8</td>
<td>99.8</td>
</tr>
<tr>
<td>CRAM, Preserve 50</td>
<td>132.6</td>
<td>52,334</td>
<td>51,387</td>
<td>8,559</td>
<td>947</td>
<td>85.7</td>
<td>98.2</td>
</tr>
<tr>
<td>LogBinning, ( b = 60 )</td>
<td>137.5</td>
<td>59,832</td>
<td>59,797</td>
<td>149</td>
<td>35</td>
<td>99.8</td>
<td>99.9</td>
</tr>
<tr>
<td>gzip (lossless)</td>
<td>148.1</td>
<td>59,946</td>
<td>59,946</td>
<td>0</td>
<td>0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
</tbody>
</table>

Table 5.5: Measured recall and precision for VCF outputs generated after different lossy methods applied to the quality scores in the file \texttt{NA12878.quality}. The rows are ordered by increasing compressed size.
<table>
<thead>
<tr>
<th>Method</th>
<th>Size (MB)</th>
<th>Variations</th>
<th>TP</th>
<th>FN</th>
<th>FP</th>
<th>Recall</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OneQual, ( q = &quot;E&quot; ) = 69</strong></td>
<td>0.0</td>
<td>162,929</td>
<td>150,544</td>
<td>307</td>
<td>12,385</td>
<td>99.8</td>
<td>92.4</td>
</tr>
<tr>
<td><strong>R-Block, ( \theta = 4.2 )</strong></td>
<td>15.4</td>
<td>149,565</td>
<td>146,815</td>
<td>4,036</td>
<td>2,750</td>
<td>97.3</td>
<td>98.2</td>
</tr>
<tr>
<td><strong>P-Block, ( p = 16 )</strong></td>
<td>16.0</td>
<td>153,739</td>
<td>149,383</td>
<td>1,468</td>
<td>4,356</td>
<td>99.0</td>
<td>97.2</td>
</tr>
<tr>
<td><strong>R-Block, ( \theta = 1.4 )</strong></td>
<td>24.0</td>
<td>151,263</td>
<td>149,665</td>
<td>1,186</td>
<td>1,598</td>
<td>99.2</td>
<td>99.0</td>
</tr>
<tr>
<td><strong>QualComp, ( r = 0.5 )</strong></td>
<td>38.6</td>
<td>156,696</td>
<td>150,072</td>
<td>779</td>
<td>6,624</td>
<td>99.5</td>
<td>95.8</td>
</tr>
<tr>
<td><strong>P-Block, ( p = 6 )</strong></td>
<td>46.8</td>
<td>151,497</td>
<td>150,086</td>
<td>765</td>
<td>1,411</td>
<td>99.5</td>
<td>99.1</td>
</tr>
<tr>
<td><strong>QualComp, ( r = 1 )</strong></td>
<td>77.2</td>
<td>154,112</td>
<td>150,199</td>
<td>652</td>
<td>3,913</td>
<td>99.6</td>
<td>97.5</td>
</tr>
<tr>
<td><strong>UniBinning, ( b = 100 )</strong></td>
<td>80.5</td>
<td>128,298</td>
<td>127,888</td>
<td>22,963</td>
<td>410</td>
<td>84.8</td>
<td>99.7</td>
</tr>
<tr>
<td><strong>P-Block, ( p = 4 )</strong></td>
<td>83.4</td>
<td>151,030</td>
<td>150,183</td>
<td>668</td>
<td>847</td>
<td>99.6</td>
<td>99.4</td>
</tr>
<tr>
<td><strong>LogBinning, ( b = 5 )</strong></td>
<td>113.5</td>
<td>136,052</td>
<td>135,772</td>
<td>15,079</td>
<td>280</td>
<td>90.0</td>
<td>99.8</td>
</tr>
<tr>
<td><strong>R-Block, ( \theta = 1.10 )</strong></td>
<td>119.3</td>
<td>150,806</td>
<td>150,308</td>
<td>543</td>
<td>498</td>
<td>99.6</td>
<td>99.7</td>
</tr>
<tr>
<td><strong>CRAM Bin-Preserve, 50</strong></td>
<td>136.0</td>
<td>150,966</td>
<td>148,953</td>
<td>1,898</td>
<td>2,013</td>
<td>98.7</td>
<td>98.7</td>
</tr>
<tr>
<td><strong>CRAM Match-Bin-Preserve, 50</strong></td>
<td>137.5</td>
<td>138,738</td>
<td>130,168</td>
<td>20,683</td>
<td>8,570</td>
<td>86.3</td>
<td>93.8</td>
</tr>
<tr>
<td><strong>LogBinning, ( b = 20 )</strong></td>
<td>140.1</td>
<td>148,322</td>
<td>148,151</td>
<td>2,700</td>
<td>171</td>
<td>98.2</td>
<td>99.9</td>
</tr>
<tr>
<td><strong>QualComp, ( r = 2 )</strong></td>
<td>154.3</td>
<td>151,670</td>
<td>150,247</td>
<td>604</td>
<td>1,423</td>
<td>99.6</td>
<td>99.1</td>
</tr>
<tr>
<td><strong>P-Block, ( p = 2 )</strong></td>
<td>169.0</td>
<td>150,831</td>
<td>150,408</td>
<td>443</td>
<td>423</td>
<td>99.7</td>
<td>99.7</td>
</tr>
<tr>
<td><strong>LogBinning, ( b = 60 )</strong></td>
<td>252.4</td>
<td>150,409</td>
<td>150,369</td>
<td>482</td>
<td>40</td>
<td>99.7</td>
<td>100.0</td>
</tr>
<tr>
<td><strong>CRAM, Preserve 50</strong></td>
<td>283.9</td>
<td>151,775</td>
<td>150,379</td>
<td>472</td>
<td>1,396</td>
<td>99.7</td>
<td>99.1</td>
</tr>
<tr>
<td><strong>gzip (lossless)</strong></td>
<td>313.6</td>
<td>150,851</td>
<td>150,851</td>
<td>0</td>
<td>0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
</tbody>
</table>

Table 5.6: Measured recall and precision for VCF outputs generated after different lossy methods applied to the quality scores in the file `HG01477.quality`. The rows are ordered by increasing compressed size.
<table>
<thead>
<tr>
<th>Method</th>
<th>Size (MB)</th>
<th>Variations</th>
<th>TP</th>
<th>FN</th>
<th>FP</th>
<th>Recall</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>OneQual, $q = &quot;?&quot; = 63</td>
<td>0.0</td>
<td>83,859</td>
<td>81,893</td>
<td>4,169</td>
<td>1,966</td>
<td>95.2</td>
<td>97.7</td>
</tr>
<tr>
<td>R-Block, $\theta = 4.2$</td>
<td>137.5</td>
<td>86,971</td>
<td>85,497</td>
<td>565</td>
<td>1,474</td>
<td>99.3</td>
<td>98.3</td>
</tr>
<tr>
<td>P-Block, $p = 16$</td>
<td>140.3</td>
<td>87,172</td>
<td>85,716</td>
<td>346</td>
<td>1,456</td>
<td>99.6</td>
<td>98.3</td>
</tr>
<tr>
<td>R-Block, $\theta = 1.4$</td>
<td>255.4</td>
<td>87,038</td>
<td>85,935</td>
<td>127</td>
<td>1,103</td>
<td>99.9</td>
<td>98.7</td>
</tr>
<tr>
<td>QualComp, $r = 0.5$</td>
<td>310.5</td>
<td>86,757</td>
<td>85,705</td>
<td>357</td>
<td>1,052</td>
<td>99.6</td>
<td>98.8</td>
</tr>
<tr>
<td>P-Block, $p = 6$</td>
<td>337.0</td>
<td>86,764</td>
<td>85,930</td>
<td>132</td>
<td>834</td>
<td>99.8</td>
<td>99.0</td>
</tr>
<tr>
<td>QualComp, $r = 0.6$</td>
<td>372.6</td>
<td>86,707</td>
<td>85,709</td>
<td>353</td>
<td>998</td>
<td>99.6</td>
<td>98.8</td>
</tr>
<tr>
<td>P-Block, $p = 4$</td>
<td>561.7</td>
<td>86,495</td>
<td>85,965</td>
<td>97</td>
<td>530</td>
<td>99.9</td>
<td>99.4</td>
</tr>
<tr>
<td>QualComp, $r = 1.0$</td>
<td>621.1</td>
<td>86,543</td>
<td>85,802</td>
<td>260</td>
<td>741</td>
<td>99.7</td>
<td>99.1</td>
</tr>
<tr>
<td>R-Block, $\theta = 1.1$</td>
<td>771.4</td>
<td>86,353</td>
<td>85,997</td>
<td>65</td>
<td>356</td>
<td>99.9</td>
<td>99.6</td>
</tr>
<tr>
<td>CRAM Bin-Preserve, 50</td>
<td>821.4</td>
<td>86,593</td>
<td>85,672</td>
<td>390</td>
<td>921</td>
<td>99.5</td>
<td>98.9</td>
</tr>
<tr>
<td>LogBinning, $b = 10$</td>
<td>829.5</td>
<td>86,952</td>
<td>85,685</td>
<td>377</td>
<td>1,267</td>
<td>99.6</td>
<td>98.5</td>
</tr>
<tr>
<td>CRAM Match-Bin-Preserve, 50</td>
<td>867.6</td>
<td>88,008</td>
<td>85,511</td>
<td>551</td>
<td>2,497</td>
<td>99.4</td>
<td>97.2</td>
</tr>
<tr>
<td>P-Block, $p = 2$</td>
<td>938.5</td>
<td>86,195</td>
<td>85,985</td>
<td>77</td>
<td>210</td>
<td>99.9</td>
<td>99.8</td>
</tr>
<tr>
<td>LogBinning, $b = 20$</td>
<td>1,055.3</td>
<td>86,506</td>
<td>85,836</td>
<td>226</td>
<td>670</td>
<td>99.7</td>
<td>99.2</td>
</tr>
<tr>
<td>LogBinning, $b = 5$</td>
<td>1,167.7</td>
<td>86,551</td>
<td>84,827</td>
<td>1,235</td>
<td>1,724</td>
<td>98.6</td>
<td>98.0</td>
</tr>
<tr>
<td>UniBinning, $b = 100$</td>
<td>1,213.3</td>
<td>86,830</td>
<td>85,237</td>
<td>825</td>
<td>1,593</td>
<td>99.0</td>
<td>98.2</td>
</tr>
<tr>
<td>QualComp, $r = 2.0$</td>
<td>1,242.2</td>
<td>86,355</td>
<td>85,913</td>
<td>149</td>
<td>442</td>
<td>99.8</td>
<td>99.5</td>
</tr>
<tr>
<td>CRAM, Preserve 50</td>
<td>1,704.8</td>
<td>86,636</td>
<td>85,768</td>
<td>294</td>
<td>86</td>
<td>99.7</td>
<td>99.0</td>
</tr>
<tr>
<td>LogBinning, $b = 60$</td>
<td>1,774.4</td>
<td>86,211</td>
<td>86,017</td>
<td>45</td>
<td>194</td>
<td>99.9</td>
<td>99.8</td>
</tr>
<tr>
<td>GZIP (lossless)</td>
<td>1,901.0</td>
<td>86,062</td>
<td>86,062</td>
<td>0</td>
<td>0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
</tbody>
</table>

Table 5.7: Measured recall and precision for VCF outputs generated after different lossy methods applied to the quality scores in the file NA12878.high.quality. The rows are ordered by increasing compressed size.
### Table 5.8: File data set

Note that sizes are in MB. The quality scores were extracted from each file after being transform to SAM format. The reference used for all the files was the hs37d5.fa obtained from ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/technical/reference/phase2_reference_assembly_sequence/.

<table>
<thead>
<tr>
<th>Index</th>
<th>File Name</th>
<th>Size BAM</th>
<th>Size Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NA12891.chrom20.ILLUMINA.bwa.CEU.high_coverage.20120522.bam</td>
<td>4,383.81</td>
<td>4,378.17</td>
</tr>
<tr>
<td>2</td>
<td>NA12892.chrom20.ILLUMINA.bwa.CEU.high_coverage.20120522.bam</td>
<td>4,001.33</td>
<td>3,972.15</td>
</tr>
<tr>
<td>3</td>
<td>NA21126.chrom11.ILLUMINA.bwa.GIH.low_coverage.20121211.bam</td>
<td>1,387.86</td>
<td>1,395.74</td>
</tr>
<tr>
<td>4</td>
<td>HG04238.chrom11.ILLUMINA.bwa.ITU.low_coverage.20130415.bam</td>
<td>1,025.50</td>
<td>1,010.84</td>
</tr>
<tr>
<td>5</td>
<td>HG03875.chrom11.ILLUMINA.bwa.ITU.low_coverage.20130415.bam</td>
<td>908.92</td>
<td>901.52</td>
</tr>
<tr>
<td>6</td>
<td>HG02142.chrom11.ILLUMINA.bwa.KHV.low_coverage.20130415.bam</td>
<td>912.83</td>
<td>895.44</td>
</tr>
<tr>
<td>7</td>
<td>NA18865.chrom11.ILLUMINA.bwa.YRI.low_coverage.20130415.bam</td>
<td>897.07</td>
<td>890.14</td>
</tr>
<tr>
<td>8</td>
<td>HG00109.chrom11.ILLUMINA.bwa.GBR.low_coverage.20130415.bam</td>
<td>905.30</td>
<td>882.07</td>
</tr>
<tr>
<td>9</td>
<td>HG03781.chrom20.ILLUMINA.bwa.ITU.low_coverage.20130415.bam</td>
<td>642.76</td>
<td>636.38</td>
</tr>
<tr>
<td>10</td>
<td>NA20894.chrom20.ILLUMINA.bwa.GIH.low_coverage.20120522.bam</td>
<td>533.19</td>
<td>523.31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>URL</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/technical/working/20130103_high_cov_trio_bams/NA12891/alignment/</td>
<td>03/2014</td>
</tr>
<tr>
<td>2</td>
<td>ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/technical/working/20130103_high_cov_trio_bams/NA12892/alignment/</td>
<td>03/2014</td>
</tr>
<tr>
<td>3</td>
<td>ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data/NA21126/alignment/</td>
<td>03/2014</td>
</tr>
<tr>
<td>4</td>
<td>ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data/HG04238/alignment/</td>
<td>03/2014</td>
</tr>
<tr>
<td>5</td>
<td>ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data/HG03875/alignment/</td>
<td>03/2014</td>
</tr>
<tr>
<td>6</td>
<td>ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data/HG02142/alignment/</td>
<td>03/2014</td>
</tr>
<tr>
<td>8</td>
<td>ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/data/HG00109/alignment/</td>
<td>03/2014</td>
</tr>
</tbody>
</table>
even when $r = 1$. Meanwhile R-BLOCK with $r = 1.4$ offers F-Scores over 99 per cent for all the files, using similar or less space than QualComp. Also it is possible to notice that, although P-BLOCK with $p = 6$ generates a larger compressed quality score sequence than R-BLOCK with $r = 1.4$, it still offers better F-Score value requiring less storage space than QualComp.

5.4 Summary

This chapter described our lossy Quality field compression approaches, P-BLOCK and R-BLOCK. Both methodologies group neighboring quality scores into blocks in which a specified criteria, Mean Manhattan Distance and Max:Min Distance respectively, is satisfied.

We compared the space and fidelity trade-offs offered by the proposed solutions and previous methods, concluding that the P-BLOCK, R-BLOCK, and QualComp (see Section 3.2.4) techniques outperform the other compression procedures, when the fidelity criteria presented in Table 5.1 are used to compare the effect of lossily transforming the quality scores.

Finally we showed how these lossy Quality field compression methodologies affect the variation calling downstream application. With this purpose, VCF files (see Section 3.4) were computed for each lossy file generated, using the lossy transformed quality scores, and compared it with the VCF generated from the original file. Precision, recall, and F-Score values were calculated using these VCF files and different lossy compression methodologies. From the results obtained we concluded that P-BLOCK, R-BLOCK, and QualComp outperform other techniques for a given level of precision or recall, allowing more compact storage of quality scores than do other mechanisms. Furthermore, P-BLOCK and R-BLOCK offer the best lossy compression when precision and recall over 99 per cent are required.

This chapter does not measure encoding and decoding time speeds for the Quality field, given that most of the methodologies explored process this field at the same time than the Read Sequences and/or other fields, being unfair to
compare them against the proposed techniques which can work with the Quality field independently. Moreover, the input for the approaches used was not the same for all the methodologies, while some work with while SAM files, other received as input FASTQ files. In Chapter 6 we will compare and discuss the compression and decompression times offer for some of these methodologies, centering our investigation when the inputs are SAM files.
Chapter 6

CSAM: Compressed SAM

In Chapters 4 and 5 we studied and analyzed approaches for compressing the Read Sequences and Quality fields of SAM files. This chapter explains how to combine these methodologies to get a new compressed representation for SAM files, CSAM (Compressed SAM), that supports random access to the stored data. Section 6.2 compares the compression attained using CSAM against the alternative techniques discussed in Section 3.3, exploring their differences, advantages and disadvantages. Section 6.4 describes how CSAM can be used as part of the feature count (Section 3.4) downstream application, and discusses other operations that could also be supported.

6.1 Compression Scheme

As presented in Section 2.8.4, the SAM format consists of 11 mandatory fields together with some optional fields. Our first aim is to attain better compression than the BAM format by compressing each of the SAM fields separately, exploiting their relationships where convenient. Already we have shown how to compress the Read Sequences (Chapter 4) and Quality (Chapter 5) fields, which, in general, contain the bulk of space required by the compulsory fields of SAM files. The remaining
6.1 Compression Scheme

Chapter 6 CSAM: Compressed SAM

Figure 6.1: CSAM compression scheme. The fields RNAME, POS, and Read Sequences are compressed using the presumed sequence approach, the Quality field is lossless represented using GZIP or lossy compressed using the block quality approach presented in Section 5.2. The remaining fields are separately compressed with GZIP.

fields, including the optional fields (but excluding the RNAME and POS fields which are considered as part the Read Sequence field information), are compressed independently using the general purpose compressor GZIP, assuming that the gain in space using other lossless techniques would not offer a significant improvement. This is supported by the results obtained in Chapter 5 and previous compression approaches over the Quality field, where lossless compression of the Quality field offers little advantage over a general purpose compressor as GZIP. Figure 6.1 illustrates the compression scheme followed by CSAM, where the fields RNAME, POS, and the Read Sequences are compressed using the presumed sequence approach shown in Chapter 4, the Quality field is losslessly compressed using GZIP, or, if lossy modality is desired, using one of the block quality compression approaches. Each of the remaining fields are separately compressed using GZIP.
This thesis also considers the problem of randomly accessing the compressed data, specifically we want CSAM to support the operation \textit{getInterval} described in Section 3.5. Previous chapters briefly discussed how the proposed compression approaches for the Read Sequences and Quality fields could create an index containing synchronization points within the compressed data to allow extraction of data starting from these points, instead of fully decompressing the file. Two different criteria for choosing these synchronization points were explored: after a fixed number of data lines, or after a fixed number of base positions. The first of these criteria requires storing an index into the compressed data indicating the start decoding position for every \( \mu \) encoded data lines (see Section 5.3). That is, the information contained between two synchronization points represents \( \mu \) data lines. The second criteria requires storing a pointer to the first data line encoded after every \( \rho \)’th base position (see Section 4.3). We will refer to these criteria as \textit{block sample} and \textit{position sample} respectively.

The BAM format provides another possible criteria which stores synchronization points every time a certain number of bytes \( \lambda \) has been compressed. In the BAM index, pointers to blocks of \( \lambda = 64 \text{ KB} \) of compressed information are stored, in addition to data about the reads (for example the references and alignment start position) contained within that 64 KB.

For any given file, the space used for the index is controlled by the parameter selected (\( \mu, \rho, \) or \( \lambda \)) depending on which approach is chosen to generate the index.

In order to support the \textit{getInterval\text((rname, x, y)\text)} operation (where \textit{rname} is a reference name, and \([x, y]\) defines an interval to be search), CSAM needs to store extra information about the data contained between synchronization points, depending on the method used to generate the index. For example, if a block sample or the BAM approach is used to generate the index, then is necessary to know the positions (POS field value) and reference name (RNAME field value) relative to the first read aligned at each synchronization point. This information is needed when \textit{getInterval\text((rname, x, y)\text)} is queried, in which a search over the stored positions and reference names has to be performed to find which synchronization points need to be accessed. After finding this point, a sequential search for the required data is done.
starting from the selected pointer. Note that in both cases, block sample and the 
BAM indexes, the search for the first read aligned within the desired range since the 
synchronization point, is limited by the parameter chosen to create the index. On 
the other hand, if position samples are used to generate the index, then selecting the 
synchronization point to access given a query is direct (the \(\lfloor x/\rho \rfloor \) th pointer assigned 
to \textit{rname}), without the need to search for this pointer, but the sequential search 
to the first read aligned within the desirable range is not limited by the parameter 
chosen, instead depending on the coverage of the genomic file stored.

In the case of CSAM the aim is to create an index that permits faster extraction 
of reads and their related SAM fields when \textit{getInterval} operations are to be 
supported. With this in mind, CSAM generates an index using a mixture of the 
methodologies presented. Section 4.3 described how a position sample index (see 
cluster index description in Section 4.3) can be stored and used for querying the 
fields RNAME, POS and Read Sequences offering fast access to the data, but the 
performance obtained was computed without considering the remaining fields in the 
SAM files. We implemented this index for RNAME, POS and Read Sequences 
fields, assuming that the time used to search for a position interval using a position 
sample scheme would be less than the time used to find the desired synchronization 
points and then search for the required data, if block sample or the BAM approach 
to create the index were used. To minimize unnecessary data extraction during 
sequential search, CSAM stores a separate index for the Quality field and the 
remaining SAM fields (excluding the RNAME, POS, and Read Sequences fields). 
This is done given that the information contained in these fields is generally used 
as extra conditions in the \textit{getInterval} query (for example, once the desired data 
range is found, filter the reads within the range that have a mapping quality higher 
than a value specified by the user). That is, the information stored in these fields 
will not be need to be accessed until the Read Sequence data is found. Finally, to 
connect the position sample index of the RNAME, POS, and Read Sequences with 
the block sample index of the remaining fields, CSAM stores an extra value per 
synchronization point in the position sample indicating the number of data lines 
compressed until that point. That is, each position sample contain a tuple \((n,d)\)
Algorithm 10 \textit{getInterval}(rname, x, y) takes a reference name \textit{rname} and an interval \([x, y]\) as input, and outputs the reads and their associated fields that are within given parameters. The index is constructed with parameter \(\rho\) and the index information is assumed to exist.

1: Using the cluster index, set \textit{posSample} to be the beginning of the position samples for \textit{rname}
2: Let \((n, d)\) be the position sample information for \textit{posSample} + \(\lfloor x/\rho \rfloor\)
3: \textit{skipLines} \leftarrow \textit{getLines} \leftarrow 0
4: \textit{SeqData} \leftarrow \textit{RemData} \leftarrow \{ \}
5: Commence reading read information at address \textit{d}, storing (Read Sequence, POS, and RNAME) into variable \textit{lineSeq}
6: \textbf{while} \textit{lineSeq.RNAME} = \textit{rname} \textbf{and} \textit{lineSeq.POS} \leq \textit{y} \textbf{do}
7: \hspace{1em} \textbf{if} \textit{lineSeq.POS} < \textit{x} \textbf{then}
8: \hspace{2em} \textit{skipLines} \leftarrow \textit{skipLines} + 1
9: \hspace{1em} \textbf{else}
10: \hspace{2em} \textit{getLines} \leftarrow \textit{getLines} + 1
11: \hspace{2em} \textbf{Add} \textit{lineSeq} to \textit{SeqData}
12: \hspace{1em} \textbf{end if}
13: Get the next read line and store it in \textit{lineSeq}
14: \textbf{end while}
15: \textit{blockSample} \leftarrow \lfloor (n + \textit{skipLines})/\mu \rfloor
16: \textit{skipLines} \leftarrow n + \textit{skipLines} - \mu \cdot \textit{blockSample}
17: Position file pointer ready to read the remaining field information to \textit{lineRem} at the address \textit{blockSample} in the block sample index
18: Read \textit{skipLines} lines from the remaining fields, discarding the data
19: Read the next \textit{getLines} lines from the remaining fields, storing them in \textit{RemData}
20: \textbf{return} result of joining \textit{SeqData} and \textit{RemData}

where \(n\) is the number of lines up to the position sample, and \(d\) is a pointer to the location of the beginning of the read information for that position.

CSAM uses the two indexes just discussed to support \textit{getInterval} operations. Algorithm 10 gives pseudocode for this process, when the inputs are a reference \((\textit{rname})\) and a position interval \([\lfloor x, y \rfloor]\). Note that in the return statement, which reorders the extracted information to form a set of valid data lines, it would be possible to check for extra restrictions over the remaining fields, if required.

If the number of lines to be extracted in an execution of Algorithm 10 is too large for them to be stored in RAM, then it would not be possible to generate the
data sets \textit{SeqData} and \textit{RemData}. That is, steps 6 to 14 might fill the RAM, with no room for the data of step 19. In the actual CSAM software we interlace the process of extracting the Read Sequence information and the remaining fields extracting the information of all the fields controlling the amount of RAM used.

For example if $\rho = 1000$ for the position sample index, $\mu = 500$ for the block sample index, and the query to be performed is: “obtain all the data lines where RNAME is \textit{chr}20 (chromosome 20), the reads start between position (POS field) 1500 and 2500, and its CIGAR string contains at least 10 soft clipping”, then the implementation of Algorithm 10 would follow these steps:

- Assuming that the reference and interval are valid, get entry $\lfloor 1500/\rho \rfloor$ from the position sample index for RNAME “\textit{chr}20” which gives: $d$, the byte address where reads start for the block containing POS 1500; and $n$, the number of lines that occur before $d$ (as in step 2).

- Sequentially process lines from the compressed reads beginning at $d$ until its POS value is higher or equal to 1500, keeping a count of lines read, $c$ (Equivalent to computing \textit{skipLines} in steps 6 to 8).

- Now perform steps 15 to 18, looking up the block sample index at position $\lfloor \frac{n+c}{\mu} \rfloor$, which gives the disk location from where the data associated with the remaining fields can be extracted.

- Finally interleave steps 6 to 14 and step 19, extracting data from all the fields until the value of POS in an extracted line is higher than 2500, only keeping lines in which the CIGAR string contains at least 10 soft clipping. The interleaving is performed so that \textit{SeqData} and \textit{RemData} both fit in RAM, and if they get too large the information is outputted (as in step 20), and the looping continue.

Figure 6.2 illustrates the example just described.
6.2 SAM Lossless and Lossy Compression

In the previous chapters we had already analyzed and selected parameters for compressing the Read Sequences and Quality field components using our proposed approaches. In this section, using these parameters, we measure the performance of CSAM. In order to conduct this study, we make use of all the training and testing files used in the previous chapters, exploring the results obtained when the files are compressed, decompressed, and randomly accessed using different approaches. Table 6.1 summarizes the files used in this section, indicating their space in MB, in SAM format and including information about the reference files needed by the
<table>
<thead>
<tr>
<th>Index</th>
<th>File Name</th>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NA12878.HiSeq.WGS.bwa.cleaned.recal.hg19.20.sam</td>
<td>22,216.87</td>
</tr>
<tr>
<td>2</td>
<td>NA12891.chrom20.ILLUMINA.bwa.CEU.high_coverage.20120522.sam</td>
<td>19,786.82</td>
</tr>
<tr>
<td>3</td>
<td>NA12892.chrom20.ILLUMINA.bwa.CEU.high_coverage.20120522.sam</td>
<td>17,951.36</td>
</tr>
<tr>
<td>4</td>
<td>NA21126.chrom11.ILLUMINA.bwa.GIH.low_coverage.20121211.sam</td>
<td>6,097.97</td>
</tr>
<tr>
<td>5</td>
<td>HG04238.chrom11.ILLUMINA.bwa.ITU.low_coverage.20130415.sam</td>
<td>4,441.58</td>
</tr>
<tr>
<td>6</td>
<td>HG03875.chrom11.ILLUMINA.bwa.ITU.low_coverage.20130415.sam</td>
<td>3,960.71</td>
</tr>
<tr>
<td>7</td>
<td>HG02142.chrom11.ILLUMINA.bwa.KHV.low_coverage.20130415.sam</td>
<td>3,923.07</td>
</tr>
<tr>
<td>8</td>
<td>NA18865.chrom11.ILLUMINA.bwa.YRI.low_coverage.20130415.sam</td>
<td>3,910.51</td>
</tr>
<tr>
<td>9</td>
<td>HG00109.chrom11.ILLUMINA.bwa.GBR.low_coverage.20130415.sam</td>
<td>3,867.63</td>
</tr>
<tr>
<td>10</td>
<td>HG03781.chrom20.ILLUMINA.bwa.ITU.low_coverage.20130415.sam</td>
<td>2,780.69</td>
</tr>
<tr>
<td>11</td>
<td>HG01477.chrom11.ILLUMINA.bwa.CLM.low_coverage.20130415.sam</td>
<td>2,708.67</td>
</tr>
<tr>
<td>12</td>
<td>NA20894.chrom20.ILLUMINA.bwa.GIH.low_coverage.20120522.sam</td>
<td>2,277.28</td>
</tr>
<tr>
<td>13</td>
<td>NA12878.chrom20.ILLUMINA.bwa.CEU.low_coverage.20120522.sam</td>
<td>1,437.41</td>
</tr>
</tbody>
</table>

Table 6.1: SAM files used to test different compression approaches, their respective sizes in MB, and the references used to generate them. The reference used for all the files was the `hs37d5.fa` with the exception of `NA12878.HiSeq.WGS.bwa.cleaned.recal.hg19.20.sam` which used `Homo_sapiens_assembly19.fasta`. 
compressor/decompressor (depending of the method used). The origins of the files can be found in Section 5.3. For brevity, from this point we will refer to each file by its index assigned in Table 6.1.

### 6.2.1 Lossless Compression

As mentioned, the first aim of CSAM is to attain better compression at least as good as the BAM format, which led us to explore how the CSAM compression approach behaves when using lossless compression. In order to compare CSAM against BAM we computed the space (in MB) used by these compressed formats, also measuring the compression and decompression times obtained when the test files from Table 6.1 are used. The compression and decompression times were obtained by computing the mean and standard deviation of running each process 10 times, after an initial run to prime the cache memory. The same methodology was used for the two general purpose compressors (gzip and bzip2). We also explored the lossless compression modalities of two other approaches, CRAM (Section 3.3.2) and NGC (Section 3.3.3), which require the reference sequences used to create the files as extra input data.

Table 6.2 lists the command lines and parameters used for the compression methodologies used in the experimental process.

Several of these approaches required parameter choices or extra input data to compress the files. For example, we used 1000 as parameter for the two indexes containing the synchronization points within the CSAM format, which uses less than 4 MB for the biggest files, and a few KB for smaller files to store this information. We explore this choice in Section 6.3.

Of the existing approaches to compressing SAM files, is it important to note that CRAM and NGC need the reference sequence at compression and decompression time, and the reference is not stored as part of the final compressed SAM file. Furthermore, these two implementations do not offer true lossless compression, as both do not completely store the OTHER field and some of the remaining fields,
6.2 SAM Lossless and Lossy Compression  

<table>
<thead>
<tr>
<th>Methodology</th>
<th>Command Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAM</td>
<td>samtools view -bS file.sam &gt; file.bam</td>
<td>samtools version 0.1.18</td>
</tr>
<tr>
<td>GZIP</td>
<td>gzip -9 file.sam</td>
<td>GZIP version 1.4</td>
</tr>
<tr>
<td>BZIP2</td>
<td>bzip2 -9 file.sam</td>
<td>bzip2 version 1.0.6</td>
</tr>
<tr>
<td>CRAM</td>
<td>java -jar cramtools-1.0.jar cram -I file.sam -R reference.fasta -input-is-sam -capture-all-tags -Q -n -O file.cram</td>
<td>cramtools version 1.0</td>
</tr>
<tr>
<td>NGC</td>
<td>java -jar -Xmx3G ngc-core-0.0.1-standalone.jar compress -i file.sam -r reference.fasta -best -truncateNames -o file.ngc -validationStringency SILENT</td>
<td>ngc-core version 0.0.1</td>
</tr>
<tr>
<td>CSAM</td>
<td>CompressSAM file.sam -q 0 -s 1000 -p 1000</td>
<td>parameters s and p indicate the values of ρ and µ for the position and block sample indexes</td>
</tr>
</tbody>
</table>

Table 6.2: Command Lines used to losslessly compress SAM files using each of the methodologies studied.

Instead recomputing them at decompression time using the stored information and the external reference. The process of recalculating some of the fields causes byte-wise differences from the original files, but generally these approaches (CRAM and NGC) retain all of the same information.

Table 6.3 shows the space obtained (in MB) after compressing the test files using lossless approaches. In addition, Table 6.3 indicates the attributes of each of the compressors used, including if the compression format is indexed (can be random accessed), and if an external input reference is needed in the process of compression and decompression. If fully lossless or information preserving modality were desired, then CRAM and NGC would need to include all the information of the unstored fields, instead of recomputing them. To give an idea of the extra space needed, the extra columns at the end of Table 6.3 show the space used when the field OTHER (in general, the largest field among the ones recomputed by CRAM and NGC),
compressed using gzip -9, is added to on. Furthermore, Table 6.3 includes the average compression ratio (that is, compressed size divided by original size of the file) offered by each methodology.

Table 6.4 displays encoding rates, expressed as MB compressed per second. These numbers are the result of dividing the original size of the SAM file by the time taken to compress the file. The same information was computed for the decompression process, shown in Table 6.5, where the numerator of the division is the same, but the denominator is the decompression time, giving the number of MB extracted per second. In both cases, the bigger the number, the lower the time taken to perform the operation. A special case is the CRAM approach, which does not allow decompression of CRAM to SAM, only offering a transition to the BAM format. We assumed that the time obtained should be similar to extract CRAM to SAM, given that we presumed that CRAMTools internally extracts the SAM information sequentially and transforms it, within the same process, to BAM format. Both tables include an overall average of MB compressed/decompressed.

Table 6.3 shows that NGC gives the best size compression, outperforming all the other approaches using around 60 percent of the space used by BAM. But the big drawback of NGC is its compression and decompression speed, being 6 to 8 times slower than all the other compressors tested, and also the only functionality offered is full compression and decompression of the files, that is, no random access is supported. In these terms gzip and bzip2 provide a better option. While using more storage space than NGC, both offer faster compression and decompression times. Amongst the lossless methods that do not permit random access, CRAM is the best option, providing similar compression to NGC, and processing times closer to those obtained using bzip2. The downside of CRAM is that it cannot be decompressed to SAM files, and, as for NGC, it needs the reference sequence as extra input to the decompression and compression process. A further drawback of CRAM is that it does not really lossless compress all the SAM fields, instead avoiding storing some fields, and recomputing them at decompression time using the information from the fields that are stored, leading to data that could differ from the original SAM file.
### Table 6.3: SAM lossless compression sizes results in MB. Information about the original SAM file sizes can be found on Table 6.1. The approaches are separated by: plain lossless compression (gzip and bzip2); lossless compression methodologies that include an index to query the compressed data (BAM and CSAM); and compression techniques that compress the data but some fields of the SAM file are not stored, but recomputed at decompression time obtaining similar values to the original SAM fields (CRAM and NGC, which are labeled as Lossy* modality given that the information stored is not exactly the same as the original, but it is argued that the theoretical meaning of the data stored is not affected). The last two columns show the space used by CRAM and NGC if the OTHER field, after being gzip compressed, is added to the total required space, which could allow an information preserving compression of the original file.

<table>
<thead>
<tr>
<th>File Index</th>
<th>GZIP</th>
<th>BZIP</th>
<th>BAM</th>
<th>CSAM</th>
<th>CRAM</th>
<th>NGC</th>
<th>CRAM+OTHER.gz</th>
<th>NGC+OTHER.gz</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5,058.58</td>
<td>4,269.57</td>
<td>5,760.24</td>
<td>4,494.87</td>
<td>3,956.83</td>
<td>3,612.30</td>
<td>5,620.65</td>
<td>5,276.12</td>
</tr>
<tr>
<td>2</td>
<td>3,968.21</td>
<td>3,160.43</td>
<td>4,383.81</td>
<td>3,383.04</td>
<td>2,857.88</td>
<td>2,648.11</td>
<td>3,212.47</td>
<td>3,002.70</td>
</tr>
<tr>
<td>3</td>
<td>3,618.59</td>
<td>2,894.73</td>
<td>4,001.34</td>
<td>3,090.83</td>
<td>2,612.25</td>
<td>2,424.63</td>
<td>2,935.93</td>
<td>2,748.31</td>
</tr>
<tr>
<td>4</td>
<td>1,253.99</td>
<td>1,056.39</td>
<td>1,387.86</td>
<td>1,050.87</td>
<td>866.66</td>
<td>817.82</td>
<td>988.64</td>
<td>939.80</td>
</tr>
<tr>
<td>5</td>
<td>926.22</td>
<td>804.59</td>
<td>1,025.50</td>
<td>774.57</td>
<td>649.20</td>
<td>613.00</td>
<td>721.55</td>
<td>685.35</td>
</tr>
<tr>
<td>6</td>
<td>819.18</td>
<td>711.41</td>
<td>908.92</td>
<td>681.15</td>
<td>568.01</td>
<td>537.01</td>
<td>631.19</td>
<td>600.19</td>
</tr>
<tr>
<td>7</td>
<td>816.64</td>
<td>707.57</td>
<td>912.83</td>
<td>680.63</td>
<td>563.51</td>
<td>532.62</td>
<td>620.03</td>
<td>589.14</td>
</tr>
<tr>
<td>8</td>
<td>808.59</td>
<td>702.12</td>
<td>897.07</td>
<td>671.58</td>
<td>558.84</td>
<td>528.40</td>
<td>620.15</td>
<td>589.71</td>
</tr>
<tr>
<td>9</td>
<td>817.20</td>
<td>714.35</td>
<td>905.30</td>
<td>680.88</td>
<td>570.35</td>
<td>541.08</td>
<td>631.95</td>
<td>602.68</td>
</tr>
<tr>
<td>10</td>
<td>581.00</td>
<td>487.07</td>
<td>642.76</td>
<td>494.09</td>
<td>403.33</td>
<td>383.63</td>
<td>462.30</td>
<td>442.60</td>
</tr>
<tr>
<td>11</td>
<td>566.45</td>
<td>492.31</td>
<td>627.65</td>
<td>466.03</td>
<td>379.63</td>
<td>359.57</td>
<td>417.23</td>
<td>397.17</td>
</tr>
<tr>
<td>12</td>
<td>478.46</td>
<td>408.20</td>
<td>533.19</td>
<td>400.82</td>
<td>331.40</td>
<td>313.42</td>
<td>371.51</td>
<td>357.53</td>
</tr>
<tr>
<td>13</td>
<td>282.99</td>
<td>247.79</td>
<td>322.21</td>
<td>239.63</td>
<td>194.68</td>
<td>183.14</td>
<td>217.28</td>
<td>205.74</td>
</tr>
</tbody>
</table>

<p>| Avg. Ratio | 0.208 | 0.177 | 0.231 | 0.175 | 0.146 | 0.137 | 0.168 | 0.159 |</p>
<table>
<thead>
<tr>
<th>File Index</th>
<th>GZIP</th>
<th>BZIP2</th>
<th>BAM</th>
<th>CSAM</th>
<th>CRAM</th>
<th>NGC</th>
<th>CRAM+OTHER.gz</th>
<th>NGC+OTHER.gz</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11.19 ± 0.05</td>
<td>10.75 ± 0.02</td>
<td>27.16 ± 0.07</td>
<td>15.44 ± 0.09</td>
<td>10.88 ± 0.03</td>
<td>2.69 ± 0.02</td>
<td>6.43 ± 0.03</td>
<td>2.30 ± 0.02</td>
</tr>
<tr>
<td>2</td>
<td>26.82 ± 0.13</td>
<td>12.03 ± 0.06</td>
<td>36.95 ± 0.16</td>
<td>16.30 ± 0.08</td>
<td>13.24 ± 0.06</td>
<td>3.09 ± 0.02</td>
<td>11.52 ± 0.05</td>
<td>2.99 ± 0.02</td>
</tr>
<tr>
<td>3</td>
<td>27.67 ± 0.11</td>
<td>12.16 ± 0.04</td>
<td>37.62 ± 0.13</td>
<td>17.53 ± 0.05</td>
<td>13.27 ± 0.02</td>
<td>3.10 ± 0.02</td>
<td>11.53 ± 0.03</td>
<td>3.00 ± 0.02</td>
</tr>
<tr>
<td>4</td>
<td>25.98 ± 0.17</td>
<td>12.40 ± 0.04</td>
<td>36.30 ± 0.14</td>
<td>16.19 ± 0.06</td>
<td>10.37 ± 0.03</td>
<td>3.32 ± 0.03</td>
<td>9.24 ± 0.04</td>
<td>3.20 ± 0.03</td>
</tr>
<tr>
<td>5</td>
<td>26.05 ± 0.31</td>
<td>12.14 ± 0.05</td>
<td>37.89 ± 0.25</td>
<td>16.19 ± 0.06</td>
<td>10.77 ± 0.03</td>
<td>3.33 ± 0.02</td>
<td>9.45 ± 0.06</td>
<td>3.19 ± 0.02</td>
</tr>
<tr>
<td>6</td>
<td>24.62 ± 0.30</td>
<td>12.22 ± 0.03</td>
<td>37.52 ± 0.64</td>
<td>16.68 ± 0.08</td>
<td>10.58 ± 0.02</td>
<td>3.33 ± 0.02</td>
<td>9.32 ± 0.07</td>
<td>3.19 ± 0.02</td>
</tr>
<tr>
<td>7</td>
<td>27.07 ± 0.22</td>
<td>11.86 ± 0.08</td>
<td>32.67 ± 0.54</td>
<td>16.78 ± 0.14</td>
<td>10.53 ± 0.02</td>
<td>3.39 ± 0.03</td>
<td>9.43 ± 0.06</td>
<td>3.27 ± 0.03</td>
</tr>
<tr>
<td>8</td>
<td>27.01 ± 0.32</td>
<td>12.08 ± 0.03</td>
<td>36.14 ± 0.51</td>
<td>15.82 ± 0.13</td>
<td>10.21 ± 0.05</td>
<td>3.34 ± 0.01</td>
<td>9.07 ± 0.07</td>
<td>3.21 ± 0.01</td>
</tr>
<tr>
<td>9</td>
<td>26.06 ± 0.11</td>
<td>11.70 ± 0.08</td>
<td>33.91 ± 0.62</td>
<td>16.20 ± 0.16</td>
<td>10.83 ± 0.08</td>
<td>3.27 ± 0.01</td>
<td>9.54 ± 0.13</td>
<td>3.14 ± 0.02</td>
</tr>
<tr>
<td>10</td>
<td>24.84 ± 0.47</td>
<td>12.50 ± 0.11</td>
<td>33.83 ± 0.86</td>
<td>17.22 ± 0.22</td>
<td>12.36 ± 0.04</td>
<td>3.16 ± 0.01</td>
<td>10.81 ± 0.10</td>
<td>3.05 ± 0.02</td>
</tr>
<tr>
<td>11</td>
<td>25.95 ± 0.41</td>
<td>11.41 ± 0.03</td>
<td>34.89 ± 0.51</td>
<td>16.57 ± 0.12</td>
<td>10.36 ± 0.06</td>
<td>3.36 ± 0.03</td>
<td>9.30 ± 0.11</td>
<td>3.24 ± 0.03</td>
</tr>
<tr>
<td>12</td>
<td>28.41 ± 0.38</td>
<td>12.92 ± 0.09</td>
<td>41.82 ± 0.38</td>
<td>16.91 ± 0.24</td>
<td>12.34 ± 0.08</td>
<td>3.20 ± 0.01</td>
<td>10.79 ± 0.13</td>
<td>3.09 ± 0.01</td>
</tr>
<tr>
<td>13</td>
<td>25.39 ± 0.69</td>
<td>12.36 ± 0.04</td>
<td>15.14 ± 0.04</td>
<td>18.72 ± 0.08</td>
<td>13.20 ± 0.08</td>
<td>3.39 ± 0.03</td>
<td>11.52 ± 0.09</td>
<td>3.26 ± 0.03</td>
</tr>
<tr>
<td>Overall</td>
<td>25.11 ± 0.28</td>
<td>12.04 ± 0.05</td>
<td>33.99 ± 0.37</td>
<td>16.66 ± 0.12</td>
<td>11.46 ± 0.05</td>
<td>3.23 ± 0.02</td>
<td>9.84 ± 0.07</td>
<td>3.09 ± 0.02</td>
</tr>
</tbody>
</table>

Table 6.4: Compression speed in MB compressed per second using different lossless compression approaches over the test files from Table 6.1. The methodologies are grouped following the description given in Table 6.3. A description of the test process is provided in page 155. The last row lists the overall averages of the speed and their standard deviation over all the sample files for each method.
Table 6.5: Decompression speed in MB extracted per second using different lossless approaches. CRAMTools do not offer a CRAM to SAM transition, so the stats presented for CRAM are the MB extracted per second when the final output is a BAM file. The methodologies are grouped following the description given in Table 6.3. A description of the test process is provided in page 155. The last row lists the overall averages of the speed and their standard deviation over all the sample files for each method.
CSAM and BAM are the only lossless compressors studied that provide a full lossless compression technique containing an index that allows random access to the data. While CSAM clearly use less space than BAM, BAM offers faster compression, and both methods offer similar full decompression times. The advantage of CSAM over BAM is that each field is treated separately, enabling the user to decompress and/or random access each field individually without the need to access the whole blocks of data as in BAM. Also, as will be discussed in the following section, CSAM includes a lossy compression modality for the Quality field which greatly improves the compression at the cost of reduced precision in the quality scores stored.

Figure 6.3 graphically illustrates the average decompression time versus the average compression ratio for all methods. In this figure, the “desirable” quadrant is the upper left region.


6.2 Lossless and Lossy Compression

6.2.2 Lossy Compression

In Chapter 5 we discussed and studied lossy compression approaches for the Quality field. This work allows us to offer two lossy SAM compression techniques, which we refer as CSAM-P and CSAM-R, using the P-Block (Section 5.2.1) and R-Block (Section 5.2.2) quality compression techniques respectively. Furthermore Section 5.3 demonstrated that the best compression allowed by each of the proposed methodologies, maintaining over 99.0 per cent of recall and precision in the variation call downstream application, were given when the lossy parameters chosen for P-Block and R-Block were \( p = 6 \) and \( r = 1.4 \), respectively.

In addition to the CSAM lossy approaches, CRAM and NGC also provide lossy compression over the Quality field, and the option to not store some fields. We resolved to not explore compression variations that completely discarded fields. That is, in this thesis we only use CRAM and NGC compression parameters that allow lossy compress the Quality field, while all the remaining fields are losslessly compressed. Section 5.3 explored different lossy parameters used by CRAM to compress the Quality field, concluding that CSAM allows higher compression than CRAM whenever the lossy compressed data must fulfill that recall and precision is over 99.0 percent in the variation call downstream application when using the original SAM files as ground truth. In this section every time that we refer to the CRAM results, we will be using CRAM with parameter of lower bound mapping qualities of 50 and the Bin-Preserve mode (see Section 3.3.2), which were the setting that offered the best trade-off in the experiments presented in Section 5.3.

Studies of the NGC Quality field lossy compression technique were not included in Section 5.3, given its high dependency on the values of the remaining fields and the use of external reference sequences. Popitsch and von Haeseler \([PvH13]\) showed an exhaustive experimental study (in the supplementary material\(^1\)) demonstrating that, between all the optional parameters offered, the modality \( m1 \) was the only one that assured over 99.0 per cent of the original variants were recovered after

\(^1\)http://nar.oxfordjournals.org/content/suppl/2012/10/11/gks939.DC1/nar-01707-met-n-2012-File002.pdf
Table 6.6: SAM lossy compression sizes results in MB. Information about the original SAM file size can be found on Table 6.1. The approaches are separated by: lossy compression methodologies that include an index to query the compressed data (CSAM-P, with \( p = 4 \), and CSAM-R, with \( r = 1.4 \)); and compression techniques that compress the data but some fields of the SAM file were not stored, instead being recomputed at decompression time obtaining similar values than the original SAM fields (CRAM, using as parameter of lower bound mapping qualities of 50 and the Bin-Preserve mode (see Section 5.3), and NGC, using its \( m1 \) modality). The last two columns lists the space used by CRAM and NGC if the OTHER field, after being \texttt{gzip} -9 compressed, is included.
Table 6.7: Compression speed in MB compressed per second using different lossy approaches. The methodologies are grouped following the description given on Table 6.6, using the same lossy parameters indicated there. A description of the test process is provided in page 155. The last row lists the overall averages of the speed and their standard deviation over all the sample files for each method.

<table>
<thead>
<tr>
<th>File Index</th>
<th>CSAM-P</th>
<th>CSAM-R</th>
<th>CRAM</th>
<th>NGC</th>
<th>CRAM+OTHER.gz</th>
<th>NGC+OTHER.gz</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20.60 ± 0.14</td>
<td>20.21 ± 0.08</td>
<td>12.48 ± 0.05</td>
<td>4.26 ± 0.05</td>
<td>6.95 ± 0.04</td>
<td>3.35 ± 0.04</td>
</tr>
<tr>
<td>2</td>
<td>23.84 ± 0.06</td>
<td>23.98 ± 0.07</td>
<td>12.30 ± 0.05</td>
<td>5.50 ± 0.06</td>
<td>10.82 ± 0.04</td>
<td>5.18 ± 0.06</td>
</tr>
<tr>
<td>3</td>
<td>23.44 ± 0.07</td>
<td>24.31 ± 0.06</td>
<td>11.53 ± 0.03</td>
<td>5.64 ± 0.07</td>
<td>10.20 ± 0.03</td>
<td>5.30 ± 0.07</td>
</tr>
<tr>
<td>4</td>
<td>23.50 ± 0.13</td>
<td>24.11 ± 0.14</td>
<td>10.16 ± 0.02</td>
<td>6.46 ± 0.03</td>
<td>9.07 ± 0.03</td>
<td>6.00 ± 0.04</td>
</tr>
<tr>
<td>5</td>
<td>24.20 ± 0.11</td>
<td>23.91 ± 0.17</td>
<td>9.81 ± 0.02</td>
<td>7.07 ± 0.03</td>
<td>8.71 ± 0.04</td>
<td>6.48 ± 0.04</td>
</tr>
<tr>
<td>6</td>
<td>23.10 ± 0.22</td>
<td>24.61 ± 0.15</td>
<td>9.97 ± 0.02</td>
<td>7.25 ± 0.04</td>
<td>8.84 ± 0.07</td>
<td>6.64 ± 0.06</td>
</tr>
<tr>
<td>7</td>
<td>23.38 ± 0.23</td>
<td>23.52 ± 0.13</td>
<td>10.38 ± 0.01</td>
<td>7.58 ± 0.06</td>
<td>9.31 ± 0.06</td>
<td>7.00 ± 0.07</td>
</tr>
<tr>
<td>8</td>
<td>22.81 ± 0.15</td>
<td>24.40 ± 0.10</td>
<td>10.09 ± 0.02</td>
<td>7.13 ± 0.03</td>
<td>8.98 ± 0.04</td>
<td>6.56 ± 0.04</td>
</tr>
<tr>
<td>9</td>
<td>23.75 ± 0.24</td>
<td>23.41 ± 0.16</td>
<td>10.22 ± 0.02</td>
<td>4.13 ± 0.01</td>
<td>9.06 ± 0.08</td>
<td>3.93 ± 0.02</td>
</tr>
<tr>
<td>10</td>
<td>23.96 ± 0.09</td>
<td>24.47 ± 0.20</td>
<td>11.57 ± 0.05</td>
<td>5.77 ± 0.04</td>
<td>10.19 ± 0.10</td>
<td>5.40 ± 0.05</td>
</tr>
<tr>
<td>11</td>
<td>24.53 ± 0.41</td>
<td>25.13 ± 0.30</td>
<td>10.05 ± 0.02</td>
<td>7.57 ± 0.04</td>
<td>9.05 ± 0.07</td>
<td>6.98 ± 0.06</td>
</tr>
<tr>
<td>12</td>
<td>25.10 ± 0.16</td>
<td>23.74 ± 0.24</td>
<td>12.27 ± 0.05</td>
<td>6.38 ± 0.02</td>
<td>10.74 ± 0.11</td>
<td>5.94 ± 0.04</td>
</tr>
<tr>
<td>13</td>
<td>25.55 ± 0.21</td>
<td>25.94 ± 0.09</td>
<td>14.79 ± 0.07</td>
<td>6.44 ± 0.04</td>
<td>12.71 ± 0.08</td>
<td>6.0 ± 0.04</td>
</tr>
</tbody>
</table>

Overall 23.71 ± 0.17 23.98 ± 0.15 11.20 ± 0.03 6.24 ± 0.04 9.59 ± 0.06 5.75 ± 0.05
CRAMTools do not offer a CRAM to SAM transition, so the stats presented for CRAM are the MB extracted per second when the final output is a BAM file. The methodologies are grouped following the description given on Table 6.6, using the same lossy parameters indicated there. A description of the test process is provided in page 155. The last row lists the overall averages of the speed and their standard deviation over all the sample files for each method.

<table>
<thead>
<tr>
<th>File Index</th>
<th>CSAM-P</th>
<th>CSAM-R</th>
<th>CRAM</th>
<th>NGC</th>
<th>CRAM+OTHER.gz</th>
<th>NGC+OTHER.gz</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>51.32 ± 0.68</td>
<td>50.16 ± 0.41</td>
<td>32.70 ± 0.14</td>
<td>6.01 ± 0.02</td>
<td>24.93 ± 0.41</td>
<td>5.69 ± 0.04</td>
</tr>
<tr>
<td>2</td>
<td>53.28 ± 0.42</td>
<td>55.17 ± 0.31</td>
<td>32.19 ± 0.18</td>
<td>6.44 ± 0.02</td>
<td>26.34 ± 0.52</td>
<td>6.16 ± 0.04</td>
</tr>
<tr>
<td>3</td>
<td>55.67 ± 0.28</td>
<td>55.66 ± 0.19</td>
<td>33.46 ± 0.14</td>
<td>6.35 ± 0.02</td>
<td>27.68 ± 0.37</td>
<td>6.11 ± 0.03</td>
</tr>
<tr>
<td>4</td>
<td>54.16 ± 0.87</td>
<td>54.30 ± 0.53</td>
<td>31.79 ± 0.08</td>
<td>7.09 ± 0.04</td>
<td>26.22 ± 0.19</td>
<td>6.77 ± 0.05</td>
</tr>
<tr>
<td>5</td>
<td>56.37 ± 0.78</td>
<td>57.02 ± 0.65</td>
<td>31.31 ± 0.20</td>
<td>8.25 ± 0.04</td>
<td>27.56 ± 0.24</td>
<td>7.97 ± 0.05</td>
</tr>
<tr>
<td>6</td>
<td>55.00 ± 1.97</td>
<td>59.34 ± 0.36</td>
<td>30.97 ± 0.16</td>
<td>7.84 ± 0.04</td>
<td>26.08 ± 0.32</td>
<td>7.49 ± 0.06</td>
</tr>
<tr>
<td>7</td>
<td>55.17 ± 1.53</td>
<td>58.96 ± 0.49</td>
<td>32.01 ± 0.19</td>
<td>8.25 ± 0.07</td>
<td>27.36 ± 0.23</td>
<td>7.91 ± 0.07</td>
</tr>
<tr>
<td>8</td>
<td>56.52 ± 0.51</td>
<td>57.12 ± 0.62</td>
<td>30.96 ± 0.23</td>
<td>8.22 ± 0.07</td>
<td>26.65 ± 0.27</td>
<td>7.88 ± 0.07</td>
</tr>
<tr>
<td>9</td>
<td>54.06 ± 1.76</td>
<td>57.12 ± 0.47</td>
<td>31.69 ± 0.31</td>
<td>7.67 ± 0.02</td>
<td>27.62 ± 0.31</td>
<td>7.40 ± 0.02</td>
</tr>
<tr>
<td>10</td>
<td>54.03 ± 0.83</td>
<td>53.84 ± 0.85</td>
<td>30.84 ± 0.19</td>
<td>6.62 ± 0.03</td>
<td>26.39 ± 0.29</td>
<td>6.39 ± 0.04</td>
</tr>
<tr>
<td>11</td>
<td>56.44 ± 0.39</td>
<td>56.92 ± 0.20</td>
<td>29.99 ± 0.17</td>
<td>7.74 ± 0.01</td>
<td>26.01 ± 0.35</td>
<td>7.45 ± 0.03</td>
</tr>
<tr>
<td>12</td>
<td>51.48 ± 0.80</td>
<td>50.66 ± 0.76</td>
<td>31.10 ± 0.20</td>
<td>7.09 ± 0.02</td>
<td>26.62 ± 0.26</td>
<td>6.83 ± 0.03</td>
</tr>
<tr>
<td>13</td>
<td>52.58 ± 2.02</td>
<td>54.41 ± 0.58</td>
<td>36.36 ± 0.40</td>
<td>6.90 ± 0.02</td>
<td>31.57 ± 0.55</td>
<td>6.71 ± 0.03</td>
</tr>
</tbody>
</table>

| Overall   | 54.31 ± 0.99   | 55.44 ± 0.49   | 31.95 ± 0.20 | 7.27 ± 0.03 | 27.00 ± 0.33 | 6.98 ± 0.04 |

Table 6.8: Decompression speed in MB extracted per second using different lossless decompression approaches. CRAMTools do not offer a CRAM to SAM transition, so the stats presented for CRAM are the MB extracted per second when the final output is a BAM file. The methodologies are grouped following the description given on Table 6.6, using the same lossy parameters indicated there. A description of the test process is provided in page 155. The last row lists the overall averages of the speed and their standard deviation over all the sample files for each method.
lossy compression of the data. This m1 modality, being the one used in this thesis to illustrate the NGC lossy compression, stores quality scores of bases that match the external reference sequence and occur in columns where all the bases of the overlapping reads in that position match the reference (see Section 3.3.3).

Table 6.6 shows the space, in MB, obtained after compressing all the test SAM files using the different lossy approaches discussed. As in Table 6.3, we indicate if the compression format is indexed and if an external input reference is needed. Table 6.6 also includes the space used by the lossy versions of CRAM and NGC when the space used by the field OTHER, after being compressed using gzip -9, is added to the total storage space, with the purpose of giving an idea of how much space it is used by the fields that are not stored by these approaches. Finally we include the average compression ratio offered by each of these methodologies.

Tables 6.7 and 6.8 show, respectively, the compression and decompression rates (± standard deviation). These numbers are calculated as discussed in the previous section. Also, both tables include an overall average throughput, averaged across the sample files.
Chapter 6 CSAM: Compressed SAM 6.3 Supporting Random Access

From the results obtained, as in Section 6.2.1 NGC offers higher compression outperforming all the other compressors described, having the same drawback on the compression and decompression performance time, being 4 to 5 times slower than all the other approaches. Also NGC’s only functionality is full compression and decompression of the files. Given these reasons, CRAM remains the best option if the only functionality needed is full compression and decompression of SAM files and the space used to store the reference sequence is not taken in account. On the other hand, the method proposed in this thesis, CSAM-P and CSAM-R, while using slightly more space, are the only lossy compression approach that support random access to the data, are independent of any external reference sequence, and allow faster compression and decompression compared with the other lossy compression techniques shown. Figure 6.4 graphically illustrates the analysis for the average decompression time versus the average compression ratio for these lossy methods.

6.3 Supporting Random Access

The previous section discussed and showed how CSAM compressed SAM files, and compared the compression and decompression performance against other approaches. However, our CSAM compressor also aims to allow random access to the stored data, specifically, supporting the queries getInterval presented in Section 3.5.

6.3.1 Buffer and Sample Sizes

At the beginning of this chapter we explained how to solve the getInterval query using two indexes (block sample and position sample index, see page 149) over the compressed data. The proposed compression approach used by CSAM orders the fields of SAM files in three components (see Figure 6.1), storing them separately within the same final compressed file. Given this division of the data and the corresponding indexes, each time that CSAM needs to access the complete
information of a read alignment, it needs to read data from three different parts of the file. If this process is repeated each time a line is extracted, the program will need to repeatedly move the disk pointer, slowing down the extraction of information. This may be significant on machines where the cost of moving the disk pointer is high (eg. traditional hard disk).

An alternative is to interlace the information of the three CSAM components, so only one disk operation is needed to access a complete read aligned. But the problem with this alternative is that when independent SAM fields are to be extracted or analyzed, it forces the program to extract all the remaining fields at the same time. That is, the cost of extracting one part goes up, in order to reduce the cost of extracting all of the parts.

Given that we chose to prioritize to be able to independently treat each SAM field, a better option is to keep the information in RAM. But this approach is limited by the machine used and could involve reading more data from disk than is needed. Reading from disk is slow, so in our case when random access queries are performed, it is desirable to minimize the number of times that the disk is accessed and, at the same time, avoid reading data that is not required to answer queries. In order to control the amount of data stored in RAM, CSAM uses three buffers of the same size (one per component). The maximum amount of data that can be in RAM is then limited by the size assigned to the buffers. Each time that CSAM accesses its compressed data, it checks if the query data is within the buffers, and if is not, any buffers that require replenishment are refilled. Depending on the number of read alignments that must be accessed and returned as a result of a query, different amounts of disk may be accessed. Given this behavior, using a buffer that is too big could mean that most of time the buffers are used to read unnecessary data, and using a buffer that is too small could involve too frequent accesses to disk, slowing down the process.

In order to chose a suitable buffer size, a suite of interval queries were executed, and for each experiment the average and standard deviation of the time used to extract each aligned read was computed. For the purpose of testing the extreme cases from the test files listed in Table 6.1, we used one big file with high coverage,
### Table 6.9: Average time in microseconds per line extracted from the CSAM compressed representation of the files 1 and 13 from Table 6.1, given the number of occurrences to be extracted (nc) and the number of interval queries (nq), versus the maximum size used per buffer to store data read from disk memory. CSAM used its lossless version with $\rho = 1000$ and $\mu = 1000$ for the position sample and block sample index respectively. The sizes tested are 0.06 (64 KB), 0.5 (512 KB), 1, 10, 100, and 200 MB, indicating the amount of information read from disk each time a buffer is fill. The lowest time obtained in each test is highlighted.

<table>
<thead>
<tr>
<th>Buffer Size (MB)</th>
<th>$nq = 10$</th>
<th>$nq = 100$</th>
<th>$nq = 1,000$</th>
<th>$nq = 10,000$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$n_c = 10$</td>
<td>$n_c = 100$</td>
<td>$n_c = 1,000$</td>
<td>$n_c = 1,000$</td>
</tr>
<tr>
<td>NA12878_low</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.06</td>
<td>631.92 ± 21.75</td>
<td>54.60 ± 3.01</td>
<td>14.16 ± 0.19</td>
<td>9.67 ± 0.29</td>
</tr>
<tr>
<td>0.50</td>
<td>555.73 ± 26.27</td>
<td>52.74 ± 2.71</td>
<td>13.26 ± 0.12</td>
<td>9.65 ± 0.30</td>
</tr>
<tr>
<td>1.00</td>
<td>542.45 ± 20.03</td>
<td>52.12 ± 2.22</td>
<td>12.54 ± 0.69</td>
<td>9.31 ± 0.28</td>
</tr>
<tr>
<td>10.00</td>
<td>894.78 ± 49.89</td>
<td>56.51 ± 2.97</td>
<td>12.70 ± 0.33</td>
<td>8.77 ± 0.19</td>
</tr>
<tr>
<td>100.00</td>
<td>1,265.42 ± 98.16</td>
<td>58.67 ± 3.66</td>
<td>13.54 ± 0.53</td>
<td>8.72 ± 0.44</td>
</tr>
<tr>
<td>200.00</td>
<td>1,346.27 ± 56.20</td>
<td>59.89 ± 3.45</td>
<td>14.66 ± 1.01</td>
<td>8.91 ± 0.26</td>
</tr>
<tr>
<td>NA12878_high</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.06</td>
<td>6,001.11 ± 174.96</td>
<td>497.83 ± 20.06</td>
<td>47.78 ± 1.75</td>
<td>14.20 ± 0.63</td>
</tr>
<tr>
<td>0.50</td>
<td>5,937.49 ± 166.31</td>
<td>459.27 ± 24.42</td>
<td>46.22 ± 1.99</td>
<td>14.21 ± 0.42</td>
</tr>
<tr>
<td>1.00</td>
<td>5,641.90 ± 184.27</td>
<td>454.68 ± 21.87</td>
<td>39.68 ± 1.76</td>
<td>12.18 ± 0.30</td>
</tr>
<tr>
<td>10.00</td>
<td>23,730.80 ± 784.06</td>
<td>1,357.66 ± 88.56</td>
<td>52.36 ± 2.02</td>
<td>9.77 ± 0.24</td>
</tr>
<tr>
<td>100.00</td>
<td>139,882.67 ± 2,563.43</td>
<td>3,290.27 ± 104.44</td>
<td>54.05 ± 2.53</td>
<td>9.52 ± 0.07</td>
</tr>
<tr>
<td>200.00</td>
<td>223,277.50 ± 3,160.92</td>
<td>3,799.74 ± 118.78</td>
<td>65.35 ± 2.60</td>
<td>9.57 ± 0.12</td>
</tr>
</tbody>
</table>
and a small file with low coverage, and taking $\rho = 1000$ and $\mu = 1000$ for the position sample and block sample index respectively. As in Section 4.4, two parameters for generating the query to test were considered: the number of occurrences extracted given an interval ($nc$); and the number of queries ($nq$). In the experiments of this section, we only display the results obtained when the number of occurrences and queries are the same (diagonal of the tested combinations), but in the following sections we will show the rest of the outcomes for the parameter chosen. Queries were ordered by start positions, with the purpose of effective use the information stored into the buffers (see page 110). Each experiment was run 10 times, after an initial run to prime the cache memory. Table 6.9 shows the results obtained. In these experiments we modify the maximum size available per buffer, starting with small buffers of 64 KB rapidly increasing their size to 200 MB.

From Table 6.9 we can see that the bigger the buffer size selected, the longer the time taken to extract information lines except if the number of occurrences is high ($\geq 1000$). It is possible to conclude that, if the size of the queries and the number of queries to be executed are unknown, then a good buffer size parameter is 1 MB. As shown in Table 6.9, using buffer of size 1 MB for CSAM allows faster times for small results size, while offering slightly slower times for larger result sets. When the queries cover most of the whole information contained into the file (in our experiments when $nq = oq = 10,000$), it would be better to use a bigger buffer size, as it is demonstrate in the results obtained. While it would be desirable to dynamically assign the buffer size depending on the query to be performed, we decide to use 1 MB as the buffer size for all the following experiments.

After setting the buffer size to 1 MB, we studied how the values of $\rho$ and $\mu$ (position sample and block sample index parameters) affect the times obtained for the operation $getInterval$, if the same experiments are executed. Table 6.10 shows the average time taken per line extracted, when $\mu$ takes values 500, 1000 and 2000, maintaining the parameter $\rho = 1000$ as constant.

The results in Table 6.10 allow us to deduce that the bigger the parameter $\mu$, the longer the query times are. This is because the bigger the block of data stored between synchronization points, the more unnecessary data may be read
Table 6.10: Average time in microseconds per line extracted from the CSAM representation of files 1 and 13 from Table 6.1, as a function of the number of occurrences to be extracted (\( nc \)) and the number of interval queries (\( nq \)), and the value of \( \mu \) used for the block sample index of CSAM. The position sample index parameter \( \rho = 1000 \) is maintained constant. The lowest time obtained in each experiment is highlighted.
while searching for a particular interval, depending on the distance between the synchronisation point accessed and the start of the interval queried. High $\mu$ values worked better when the number of queries and occurrence per query increased. Additionally it can be inferred that for low coverage files is better to use a small $\mu$ parameter to avoid accessing blocks of data that contain mainly non-query information, but when the number of queries and occurrences increase, it is better to use a higher parameter for the block sample index. In the case of high coverage files, in general, it is expected that a small $\mu$ forces too many accesses to the data (high number covering the same area of the reference) which may slow down the process. Assuming that the number and sizes of the queries can not be predicted, we selected $\mu = 1000$ as the parameter for the block sample index used for CSAM in all the experiments in this chapter, because it offers the best option for high coverage files, while for low coverage files allows a good trade-off between performance time and number and size of the queries performed.

After setting a size for the buffers, and fixing $\mu = 1000$ as the parameter for the block sample index, we tested how the value of the parameter $\rho$ for the position sample index of CSAM files affected query time performance. It is important to mention that, given how CSAM is implemented, a condition that $\rho$ must fulfill is that it has to be larger than the maximum read length within the input SAM file. This is done to avoid reads that could overlap with more than one of the stored synchronisation points, which could causes ambiguities in the decoding process. Also, as discussed when we tested the parameter $\mu$, it is desirable to avoid having large amounts of data between synchronisation points. If we assume that for a high coverage file at least one read starts at each position and a low coverage file covers less than half of the positions, then based on the results obtained for $\mu$, we measured $\rho$ values of 500, 1000 and 2000. Table 6.11 shows the average time taken per line extracted, when the parameter $\rho$ is altered.

As was the case with the block sample index, the bigger the value of $\rho$, the longer the time taken to execute each query. Given that one of the main aims of the CSAM methodology is to answer the query *getInterval* (page 152), it is important to find the closest synchronisation point of the position sample index to
<table>
<thead>
<tr>
<th>Index position sample</th>
<th>$nq = 10$</th>
<th>$nq = 100$</th>
<th>$nq = 1,000$</th>
<th>$nq = 10,000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rho$</td>
<td>$nc = 10$</td>
<td>$nc = 100$</td>
<td>$nc = 1,000$</td>
<td>$nc = 10,000$</td>
</tr>
<tr>
<td>NA12878.low</td>
<td>500</td>
<td>600.77 ± 22.67</td>
<td>53.51 ± 3.00</td>
<td>12.49 ± 0.51</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>542.45 ± 20.03</td>
<td>52.12 ± 2.22</td>
<td>12.54 ± 0.69</td>
</tr>
<tr>
<td></td>
<td>2,000</td>
<td>845.53 ± 35.43</td>
<td>55.60 ± 2.04</td>
<td>12.78 ± 0.88</td>
</tr>
<tr>
<td>NA12878.high</td>
<td>500</td>
<td>5,689.12 ± 176.99</td>
<td>477.87 ± 18.60</td>
<td>42.76 ± 2.09</td>
</tr>
<tr>
<td></td>
<td>1,000</td>
<td>5,641.90 ± 184.27</td>
<td>454.68 ± 21.87</td>
<td>39.68 ± 1.76</td>
</tr>
<tr>
<td></td>
<td>2,000</td>
<td>5,767.14 ± 200.16</td>
<td>569.16 ± 23.00</td>
<td>44.43 ± 1.91</td>
</tr>
</tbody>
</table>

Table 6.11: Average time in microseconds per line extracted from the CSAM representation of files 1 and 13 from Table 6.1, as a function of the number of occurrences to be extracted ($nc$) and the number of interval queries ($nq$), and the value of $\rho$ used for the position sample index of CSAM. The block sample index parameter $\mu = 1000$ is maintained constant. The lowest time obtained in each experiment is highlighted.
the interval desired. The bigger the parameter chosen for this index, the greater the chance that the closest synchronization point may start from a position far from the first position desired. Depending on the coverage of the file and the parameter $\rho$, the sequential search from the synchronization point selected might involve reading many unnecessary lines before arriving at the target interval. With a high number of queries and occurrence per queries, the time taken in the sequential search from the synchronization point to the first relevant data line becomes negligible compared with the time taken to extract the required data. Following the same argument to select the parameter value of $\mu$, we selected $\rho = 1000$ as parameter for the position sample index used for CSAM in all the experiments on this chapter.

Summarizing, in the remaining experiments of this chapter, every time we use the CSAM approach to compute the operation \textit{getInterval}, the buffer size used is 1 MB, and the indexes of CSAM are generated using parameters $\mu = 1000$ and $\rho = 1000$.

### 6.3.2 Comparative Experiments

With the index parameters defined for CSAM, it is possible to compare the performance of executing \textit{getInterval} using CSAM against the results obtained using BAM. In order to test this operation we randomly generated sets of queries where the number of occurrences extracted given an interval, and the number of queries performed, were as previously defined in Section 4.4 (see page 110). The interval queries used are in increasing order, that is, ordered by reference and start position. For each of the methodologies tested, the mean time (including the standard deviation) to extract a line was measured by running sets of different sample intervals. Each experiment was run 10 times, after an initial run to prime the cache memory. The implementations used were: the lossless CSAM (Section 6.2.1); the two lossy CSAM variants used in Section 6.2.2 (CSAM-P with $p = 6$ and CSAM-R with $r = 1.4$); and the SAMTools software which is used to compute \textit{getInterval} over pre-indexed BAM format files. All these methods receive an input file consisting of 3 columns (reference name, start position, end position) that are
the queries to be executed sequentially. Note that this is a simple version of the query; it is possible to add more specifications such as minimum mapping quality desired, number of soft-clipping bases, and so on. Tables 6.12 and 6.13 show the results obtained when \textit{getInterval} operations are executed using two extreme SAM files cases: a large file with high coverage and a small file with low coverage (files 1 and 13 from Table 6.1). While SAMTOOLS allows input from a file containing all queries to be processed, it is also possible to run the queries in a sequential, brute force form, where each query is given as a command line parameter to the tool. The drawback of running the queries sequentially is that SAMTOOLS is not able to detect overlapping intervals, returning a SAM file which could contain duplicate information. Tables 6.12 and 6.13 include the results obtained using both variants, where the command lines used in each case were:

- **BAM**: samtools view -L queries-file file.bam > output.sam
- **BAM Brute Force**: samtools view file.bam query-1 query-2 query-3 ... > output.sam

From the results in Tables 6.12 and 6.13 it is possible to deduce that the average time taken to extract a line, independently of the approach used to compress the SAM file, rapidly decreases as soon as the number of queries to be executed is increased and/or the number of occurrences extracted per query is large. The reason behind this behavior it is that the time used to find the first occurrence within the interval searched dominates the total time consumed when small queries are executed. This behavior can be noticed in Table 6.12 and 6.13, if the results obtained for the low and high coverage example test files are compared, when the number of occurrences per query is equal to 10. Clearly the results displayed for the high coverage file take longer time given that finding the data within a user-defined interval involves a longer search. We also infer that when the number of queries to be executed is increased, most of the approaches presented store information in RAM memory, making use of this information to speed the process when queries are run. If the number of occurrences extracted per query grows, the time taken to find the first occurrence within an interval, becomes insignificant compared to the time taken to process the complete query.
Table 6.12: Average (± standard deviation) time in microseconds per line extracted from BAM and our lossless and lossy compressed representation of a low coverage SAM file (file 13 from Table 6.1), given the number of occurrences to be extracted (nc) and the number of interval queries (nq). The BAM Brute Force methodology for processing the queries is also included.
# queries (\(nq\)) & Occurrence per query (\(nc\)) & \(10\) & \(100\) & \(1,000\) & \(10,000\) \\
BAM & & & & & & \\
\(10\) & 725.748.33 ± 5.160.59 & 73.913.17 ± 3.489.90 & 6.796.75 ± 61.75 & 683.03 ± 3.51 \\
\(100\) & 74.794.17 ± 2.677.56 & 7.123.17 ± 101.22 & 721.53 ± 11.81 & 81.99 ± 0.90 \\
\(1,000\) & 8.404.50 ± 64.11 & 870.39 ± 6.85 & 100.72 ± 4.34 & 20.55 ± 1.45 \\
\(10,000\) & 831.04 ± 37.77 & 88.93 ± 2.01 & 17.54 ± 0.35 & 11.40 ± 1.24 \\
CSAM Lossless & & & & & & \\
\(10\) & 5.641.90 ± 184.27 & 533.42 ± 41.11 & 60.96 ± 2.59 & 17.37 ± 1.31 \\
\(100\) & 3.917.93 ± 85.45 & 454.68 ± 21.87 & 44.92 ± 2.52 & 14.87 ± 0.48 \\
\(1,000\) & 2.525.23 ± 78.86 & 267.52 ± 6.27 & 39.68 ± 1.76 & 14.50 ± 0.11 \\
\(10,000\) & 846.32 ± 25.08 & 99.20 ± 0.78 & 21.32 ± 0.36 & 12.18 ± 0.30 \\
CSAM-P & & & & & & \\
\(10\) & 4.984.29 ± 394.02 & 532.00 ± 37.01 & 56.85 ± 3.30 & 16.49 ± 0.61 \\
\(100\) & 2.681.92 ± 214.45 & 192.77 ± 20.87 & 30.51 ± 1.56 & 12.45 ± 0.31 \\
\(1,000\) & 1.225.27 ± 29.38 & 135.47 ± 5.50 & 26.80 ± 1.04 & 12.23 ± 0.20 \\
\(10,000\) & 531.62 ± 29.99 & 66.39 ± 0.63 & 16.65 ± 0.58 & 8.78 ± 0.07 \\
CSAM-R & & & & & & \\
\(10\) & 4.895.60 ± 320.60 & 524.29 ± 28.72 & 53.72 ± 3.02 & 16.18 ± 0.46 \\
\(100\) & 2.401.02 ± 199.25 & 185.19 ± 22.36 & 28.17 ± 1.51 & 12.36 ± 0.19 \\
\(1,000\) & 1.121.09 ± 20.52 & 119.10 ± 2.45 & 25.00 ± 0.75 & 11.21 ± 0.11 \\
\(10,000\) & 491.57 ± 16.45 & 64.24 ± 0.30 & 14.65 ± 0.31 & 8.26 ± 0.03 \\
BAM Brute Force & & & & & & \\
\(10\) & 1.963.33 ± 139.95 & 194.17 ± 14.78 & 23.98 ± 2.31 & 8.95 ± 2.11 \\
\(100\) & 1.680.83 ± 92.42 & 159.65 ± 8.53 & 22.07 ± 1.62 & 9.03 ± 0.27 \\
\(1,000\) & 1.425.08 ± 13.99 & 145.37 ± 7.09 & 22.01 ± 0.52 & 11.57 ± 1.02 \\
\(10,000\) & 1.160.27 ± 12.77 & 122.71 ± 1.24 & 19.39 ± 1.68 & 9.11 ± 0.53 \\

Table 6.13: Average (± standard deviation) time, in microseconds per line extracted from BAM and our lossless and lossy compressed representation of a high coverage SAM file (file 1 from Table 6.1), given the number of occurrences to be extracted (\(nc\)) and the number of interval queries (\(nq\)). The BAM Brute Force methodology for processing the queries is also included.
Also from Tables 6.12 and 6.13, and looking at the results listed in Tables 6.3 and 6.6, it can be concluded that, in general all the CSAM approaches offer competitive or better performance times for the getInterval operation compared with using BAM to solve the same query. CSAM shows an overall better performance when the number of queries and occurrences per query are below 1000, and otherwise has similar times to the BAM variations. An important detail to notice is that BAM Brute Force unexpectedly showed better performance than the SAMTools built-in operation. The problem with the SAMTools built-in operation is that it does not use the BAM index to find the intervals, making the process extremely slow for queries with a small number of answers. Finally, one of the reasons why the BAM approach obtains better results for higher number of queries and higher number of occurrences per query is because in the experiments explored in general all the information from the compressed files used was extracted. Given that BAM used a BGZF approach (see Section 3.3.1) to store the data, the decompression of data without the need to navigate the extracted information is extremely fast. Meanwhile, CSAM approaches still follow the scheme of decompressing line by line independently, instead of decompressing a complete block of data as BAM does. Note that, in case that the intervals queried cover the whole input file, BAM and CSAM takes similar time to extract the information as when these methodologies were used to decompress the complete file.

In order to validate the previous analysis, we computed the average times used per extracted line and the average compression ratio for each of the files in Table 6.1, using a selection of query number-occurrences tuples. In this case we use the sample when the number of queries to be executed sequentially ($qn$) and the number of occurrences extracted per query ($nc$) are the same, taking the values 10, 100, 1000, and 10000. Figure 6.5 shows the results obtained.

Figure 6.5 confirms that the CSAM approaches offer an overall better performance when the number of queries and occurrences per query are below 1000, giving competitive results for the other cases. From the figure it can be seen that the lossless version of CSAM uses slightly less space than BAM offering similar times for the getInterval operation. Furthermore, CSAM allows lossy compression
Figure 6.5: Average time in microseconds per line extracted from BAM and the CSAM lossless and lossy compressed representation of all the test files listed in Table 6.1, given the number of occurrences to be extracted (nc) and the number of interval queries to be executed (nq) set to be the same values in this view of the data. The horizontal axis indicates the compression ratio offered by each methodology (see Tables 6.3 and 6.6) and the vertical axis shows the average time taken to extract a line. Additionally, the figure include the BAM Brute Force methodology used for processing the queries. Each cluster represents a different compression approach.

of the Quality field, permitting a reduction in the compressed file size and, in the variations presented, ensuring minimal loss of information. In the cases presented in Figure 6.5, these lossy approaches (CSAM-P with \( p = 6 \) and CSAM-R with \( r = 1.4 \)) use less than 45 percent of the space used by the BAM files.

Finally from Figure 6.5 we also notice that the bigger the number of queries and number of occurrences extracted per query, the larger times obtained start to
convert to a single value, independently of the size and coverage of the input file used. As mentioned before, this is when time taken to extract the intervals start to dominate the time to find the start of an interval within the compressed file.

### 6.4 Downstream Applications: Feature Count

As discussed in Section 3.4, an important process in the analysis of genomic data is computing the coverage (number of reads stored) within user-defined desirable chromosome intervals. From the tools that support this application [LHP+13, LSS14, APH15], we choose to use Liao et al.’s featureCounts [LSS14] downstream application, which currently is the most used. This program receives as input one or more SAM/BAM files and a list of genomic features (chromosome position intervals in addition to optimal extra parameters), and outputs a count of the number of reads occurring in each interval plus statistical information for the overall summarization results. The featureCounts program internally prearranges the query intervals and conditions received as input (that is, the genomic features in GFF or SAF format), and then traverses the SAM/BAM files searching for aligned reads that fall into the queried features.

When featureCounts runs a query, it searches over the aligned read information where the condition indicated by the genomic features are fulfilled. For example, searching if the alignment start position is within a given interval (obtained using the POS and CIGAR fields), or if the strand direction is positive or negative (obtained using the FLAG field), or if the overall mapping quality of the read is over a certain value (obtained using the MAPQ field), and so on. All the possible queries start by indicating an interval and optionally adding more conditions to the desired properties of the align reads to be counted. From this process it is clear that, depending on the genomic feature queries, not all the SAM fields need to be examined. We defined the Simple SAM Notation format (SSN) as a SAM file where all the fields that are not involved in the computation, are replaced by an empty value, instead of being extracted when the subset SAM file is computed.
In this section we study how to combine CSAM with **featureCounts**. While we were not in a position to internally include our format in the **featureCounts** program, it was possible to emulate the process that the program should be internally doing when the input are BAM files. That is, given a list of genomic features, extract only the areas of the files that may contain relevant information and use this information to compute the coverage of the indicated features. In order to reproduce this process using CSAM, we introduce the `getIntervalSSN` operation, which is similar to `getInterval` but receives an extra parameter indicating which SAM fields need to be extracted. Using this function and the list of genomic features, it is possible to generate a subset SSN file containing only the aligned reads whose start positions are likely relevant to the desired query intervals. Finally this file, in addition to the list of genomic features, is given as input to **featureCounts**, generating the same output as when original complete SAM files are used.

In order to assess the performance of CSAM when used as input, we measured the time since the generation of the SSN file (using CSAM’s `getIntervalSSN` operation and the given genomic feature list information), and then run **featureCounts** using this file as input. For SAM and BAM the start point to measure the time would be the moment when **featureCounts** receive any of these files as input. Figure 6.6 illustrates the timing process.

The genomic feature file used in this thesis, as in Liao et al.’s work [LSS14], are the exon locations for the human reference hg19, defined as in the NCBI human RefSeq annotation (build 37.2)\(^2\). This file can be obtained via the `getInBuiltAnnotation` method of the `Rsubread` tool [LSS13], which “is an **R** package that provides powerful and easy-to-use tools for analyzing next-generation sequencing read data”\(^4\).

The files from Table 6.1 were taken as test files, using their SAM, BAM, and CSAM (the lossless version and two representative lossy versions, CSAM-P with

\(^3\)http://www.r-project.org/, January 2015  
6.4 Downstream Applications: Feature Count Chapter 6 CSAM: Compressed SAM

Figure 6.6: Scheme used for measuring the time taken to generate a count report using the featureCounts application, when the input files used are a Feature File, and a SAM or BAM or CSAM file. The arrows indicate the starting point from where the execution time were measured for each case.

\[ p = 6 \text{ and } \text{CSAM-R with } r = 1.4 \] storage format. The performance times of featureCounts using SAM or BAM files were obtained by running the following command:

- featureCounts -t exon -g gene_id -a hg19_RefSeq_exon.saf -F SAF -o CountReport.txt file.(sam/bam)

For the CSAM approaches, we considered two variations for generating the SSN file: a complete SSN file where all the SAM fields are extracted, which is denoted in the experiments as CSAM-Lossless, CSAM-P and CSAM-R; and a minimal SSN file where a minimal set of the fields are extracted (the fields extracted are enough to support almost all the queries supported by featureCounts, being the FLAG, RNAME, POS, MAPQ, CIGAR and Read Sequence fields), which is denoted in the experiments as CSAM-Lossless-Min, CSAM-P-Min and CSAM-R-Min. In these cases the the performance times were obtained by running the following two commands:

- GetIntervalSSN file.csam -a hg19_RefSeq_exon.saf -f field_selection
Table 6.14: Time (in seconds) to compute the output from featureCounts when the input files used are the hg19_RefSeq_exon.saf and a SAM or BAM representation of the files in Table 6.1. The results obtained using the CSAM lossless compression methodology, where an SSN file is computed and then passed as input to featureCounts. The CSAM-Lossless-Min represent the process where the SSN file obtained contain just the main SAM fields (FLAG, RNAME, POS, MAPQ, CIGAR and Read Sequence fields) needed to calculate coverage, given any valid parameters for featureCounts. The lowest time obtained in each test is highlighted.

<table>
<thead>
<tr>
<th>File Index</th>
<th>SAM</th>
<th>BAM</th>
<th>CSAM-Lossless</th>
<th>CSAM-Lossless-Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>272.26 ± 2.22</td>
<td>68.25 ± 0.81</td>
<td>40.09 ± 1.05</td>
<td>17.71 ± 0.23</td>
</tr>
<tr>
<td>2</td>
<td>228.46 ± 2.44</td>
<td>61.24 ± 0.58</td>
<td>39.52 ± 1.20</td>
<td>17.72 ± 0.14</td>
</tr>
<tr>
<td>3</td>
<td>227.31 ± 2.04</td>
<td>55.76 ± 0.52</td>
<td>35.71 ± 0.67</td>
<td>15.12 ± 0.19</td>
</tr>
<tr>
<td>4</td>
<td>91.71 ± 2.82</td>
<td>18.63 ± 0.25</td>
<td>22.23 ± 0.64</td>
<td>8.22 ± 0.09</td>
</tr>
<tr>
<td>5</td>
<td>50.87 ± 2.15</td>
<td>13.66 ± 0.27</td>
<td>19.44 ± 0.47</td>
<td>7.19 ± 0.07</td>
</tr>
<tr>
<td>6</td>
<td>35.63 ± 0.33</td>
<td>12.22 ± 0.22</td>
<td>18.47 ± 0.64</td>
<td>6.60 ± 0.06</td>
</tr>
<tr>
<td>7</td>
<td>37.17 ± 0.53</td>
<td>12.00 ± 0.14</td>
<td>17.28 ± 0.28</td>
<td>6.23 ± 0.03</td>
</tr>
<tr>
<td>8</td>
<td>37.13 ± 0.44</td>
<td>12.03 ± 0.16</td>
<td>18.35 ± 0.51</td>
<td>6.78 ± 0.05</td>
</tr>
<tr>
<td>9</td>
<td>37.42 ± 0.53</td>
<td>11.94 ± 0.30</td>
<td>18.88 ± 0.45</td>
<td>6.76 ± 0.09</td>
</tr>
<tr>
<td>10</td>
<td>3.30 ± 0.04</td>
<td>8.69 ± 0.09</td>
<td>13.00 ± 0.56</td>
<td>5.42 ± 0.09</td>
</tr>
<tr>
<td>11</td>
<td>3.27 ± 0.07</td>
<td>8.36 ± 0.08</td>
<td>14.69 ± 0.39</td>
<td>5.36 ± 0.04</td>
</tr>
<tr>
<td>12</td>
<td>2.78 ± 0.22</td>
<td>7.08 ± 0.09</td>
<td>11.81 ± 0.05</td>
<td>4.70 ± 0.03</td>
</tr>
<tr>
<td>13</td>
<td>1.88 ± 0.11</td>
<td>4.57 ± 0.04</td>
<td>8.30 ± 0.14</td>
<td>3.83 ± 0.02</td>
</tr>
</tbody>
</table>

- featureCounts -t exon -g gene_id -a hg19_RefSeq_exon.saf -F SAF -o CountReport.txt SSNfile.sam

In the previous commands the function GetIntervalSSN returns the desired SSN file where the fields to be extracted are defined by the parameter field_selection. We computed the mean and standard deviation of running each process 10 times, after an initial run to prime the cache memory. Table 6.14 lists the results obtained when lossless approaches are used, and Table 6.15 shows the performances obtained with the lossy method. Additionally, Figure 6.7 graphically compares times when the input file format used to run featureCounts changes. Note that in Figure 6.7 the vertical axis is logarithmic and the files index in the horizontal axis are in decreasing SAM file size order, which leads to have big time differences between files 3 and 4 (~11 GB), 4 and 5 (~2 GB), 9 and 10 (~1 GB), and 12 and 13 (~1 GB).
Table 6.15: featureCounts time performance (in seconds) when the input files used are the hg19_RefSeq.exon.saf and a lossy CSAM (CSAM-P with $p = 6$ and CSAM-R with $r = 1.4$) representation of the files in Table 6.1, where a SSN file is computed and then passed as input to featureCounts. The Min versions of the CSAM approaches represent the process where the SSN file obtained contain just the main SAM fields (FLAG, RNAME, POS, MAPQ, CIGAR and Read Sequences fields) needed to calculate coverage, given any valid parameters for featureCounts. The lowest time obtained in each test is highlighted.

<table>
<thead>
<tr>
<th>File Index</th>
<th>CSAM-P</th>
<th>CSAM-P-Min</th>
<th>CSAM-R</th>
<th>CSAM-R-Min</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>35.82 ± 1.01</td>
<td>17.54 ± 0.18</td>
<td>35.44 ± 0.96</td>
<td>17.00 ± 0.10</td>
</tr>
<tr>
<td>2</td>
<td>31.32 ± 1.00</td>
<td>16.55 ± 0.13</td>
<td>31.27 ± 0.91</td>
<td>16.17 ± 0.14</td>
</tr>
<tr>
<td>3</td>
<td>29.95 ± 0.69</td>
<td>15.05 ± 0.13</td>
<td>29.68 ± 0.37</td>
<td>15.42 ± 0.16</td>
</tr>
<tr>
<td>4</td>
<td>18.57 ± 0.24</td>
<td>8.09 ± 0.08</td>
<td>17.27 ± 0.26</td>
<td>8.23 ± 0.07</td>
</tr>
<tr>
<td>5</td>
<td>15.44 ± 0.24</td>
<td>7.13 ± 0.04</td>
<td>14.83 ± 0.41</td>
<td>7.19 ± 0.06</td>
</tr>
<tr>
<td>6</td>
<td>14.96 ± 0.37</td>
<td>6.61 ± 0.08</td>
<td>14.35 ± 0.30</td>
<td>6.61 ± 0.05</td>
</tr>
<tr>
<td>7</td>
<td>14.03 ± 0.14</td>
<td>6.19 ± 0.05</td>
<td>13.36 ± 0.14</td>
<td>6.23 ± 0.06</td>
</tr>
<tr>
<td>8</td>
<td>15.13 ± 0.45</td>
<td>6.76 ± 0.06</td>
<td>14.95 ± 0.49</td>
<td>6.95 ± 0.07</td>
</tr>
<tr>
<td>9</td>
<td>14.85 ± 0.55</td>
<td>6.69 ± 0.05</td>
<td>14.96 ± 0.51</td>
<td>6.60 ± 0.06</td>
</tr>
<tr>
<td>10</td>
<td>10.54 ± 0.07</td>
<td>4.99 ± 0.04</td>
<td>10.13 ± 0.05</td>
<td>4.97 ± 0.05</td>
</tr>
<tr>
<td>11</td>
<td>11.56 ± 0.18</td>
<td>5.31 ± 0.05</td>
<td>11.02 ± 0.13</td>
<td>5.31 ± 0.03</td>
</tr>
<tr>
<td>12</td>
<td>10.00 ± 0.17</td>
<td>4.67 ± 0.04</td>
<td>9.82 ± 0.11</td>
<td>4.70 ± 0.04</td>
</tr>
<tr>
<td>13</td>
<td>7.10 ± 0.12</td>
<td>3.77 ± 0.05</td>
<td>6.73 ± 0.18</td>
<td>3.83 ± 0.04</td>
</tr>
</tbody>
</table>

From the description given by Liao et al. [LSS14] of how featureCounts operates with SAM files and Figure 6.7, it is possible to confirm that in these cases featureCounts sequentially search over all the aligned reads which ones fall into the specified genomic features. Following this method, the larger the input SAM file is, the longer the time taken to finish the process of counting the coverage for each genomic feature. For small size SAM files (files 10 to 13 in Table 6.1), featureCounts offered a faster alternative, giving that the time used decompressing other format files information is longer than completely reading the SAM file. Meanwhile, when the BAM or the CSAM approaches are used as input, the time required by featureCounts oscillates, with BAM generally giving slightly faster times for low coverage and CSAM slightly faster for high coverage files.

Finally, we tested the CSAM-Min variations, which offer the faster approach
Figure 6.7: featureCounts time performance using as input hg19_RefSeq_exon.saf and the files in Table 6.1 in their SAM, BAM, or our CSAM approaches formats. The Min versions of the CSAM approaches represent the process where the SSN file obtained contain just the main SAM fields (FLAG, RNAME, POS, MAPQ, CIGAR and Read Sequences fields) needed to calculate coverage, given any valid parameters for featureCounts. For most of the files tested, except when the SAM files are small, in which case using the original SAM file as input for featureCounts is faster. While results when the SSN file extracted all the fields and a minimum set of fields were showed, we did not present what happens when a different number of fields are extracted to generate the SSN files. Experimentally we try adding or resting fields from the minimal set, but the only point where the times obtained were more than slightly altered was when the Quality field or the OTHER fields were also extracted, and none of these fields are used by any of the operations allowed by featureCounts.

Beside comparing the time obtained using featureCounts with SAM, BAM, and CSAM, it is also important to consider the storage space used by each of the input files. From Tables 6.3 and 6.6 can be observed that BAM uses on average around 23.1%, CSAM-Lossless 17.5%, CSAM-P (with \( p = 6 \)) 8.3%, and CSAM-R (with \( r = 1.4 \)) 7.5% of the space used by the original SAM file. Also it has to be taken into consideration that, compared with SAM and BAM, the CSAM
approaches were used as pre-processor for the featureCounts application. That is, the generated SSN file had to be written to disk and then passed as input to featureCounts where it was opened and analyzed. All of this slows down the time used in the process, which could be avoided if featureCounts was modified so that the CSAM format was handled directly.

6.5 Summary

This chapter described and studied our lossless and lossy CSAM compression approaches. We showed how our methods to compress the Read Sequences and Quality fields (Chapters 4 and 5 respectively) are merged, offering a new compressed representation for SAM files, which also supports random access to the stored data.

Different lossless SAM compressors were compared, arguing that, while NGC and CRAM offer better compression space, both methodologies need a reference sequence as extra input for the decompression and compression process. Also these methods only allow full compression and decompression of the files, without supporting random access to the data. Furthermore NGC and CRAM are not strictly lossless, as they do not store some fields of the SAM file, instead recomputing them at decompression time. Our CSAM methodology and BAM are the only losslessly approaches that lossless compress SAM files, and also support random access to the data. Between these two lossless compressors, CSAM offered better compression rates, and with similar or better compression and decompression times.

Similar results were obtained when lossy SAM compression approaches were compared. NGC and CRAM remain the best options if the only functionality needed is full compression and decompression of SAM files (not taking in account the performance time and the space used to store the reference sequence). The lossy approaches proposed, CSAM-P and CSAM-R, use slightly more space than NGC and CRAM, but both fully provide random access to the data and are independent of any external reference sequence. Furthermore the CSAM lossy variations allow faster compression and decompression than all of the other techniques explored in
this thesis.

We also compared the performance of CSAM and BAM when random access queries were computed (operation \textit{getInterval} from Section 3.5). The results showed that CSAM, using less storage space, offered similar time performance to BAM when interval queries were executed. It was also demonstrated that CSAM generally gives faster results when the number of occurrences per query and number of queries to be sequentially executed are medium size (in our experimental setup, below 1000), while for other cases the time are similar with the ones obtained using BAM.

Finally we discussed how CSAM might affect the performance of the \texttt{featureCounts} downstream application, demonstrating that it is possible to speed the process of computing coverage given a list of genomic features (chromosome intervals and aligned read attributes), if CSAM is used as a pre-processing phase for \texttt{featureCounts}.
Chapter 7

Conclusions

This thesis introduced and studied CSAM, a new compressed representation for genomic data files originally produced in SAM format, which supports queries over the stored information without requiring whole files to be decompressed. Each SAM field is treated individually, with a focus on compressing the Read Sequences and Quality fields, as those two fields use the largest amount of space amongst the compulsory SAM fields. Moreover, CSAM does not make use of any external reference file, making it self-contained.

Chapter 2 summarized the required concepts to understand the studies contained in this thesis, and Chapter 3 surveyed the existing approaches to compress and work with SAM files, or some of their fields.

Chapter 4 described the proposed technique for compressing the Read Sequences field including the information of the RNAME and POS fields. The proposed method creates a presumed sequence that symbolizes the reads within the SAM file, and then stores the differences between the reads and the presumed sequence, using it as an artificial reference. The presumed sequence is generated using the most-frequent-base technique, where each base stored identifies the most frequent base found in each of the positions covered by the reads. We demonstrated that this technique correctly represented over 85 per cent of the total number of bases in the Read
Sequences field.

In Chapter 4, we compared our Read Sequence field compression technique against gzip and bzip2, showing that it was faster for both compression and decompression times, and generated more compact output than these general purposes compressors. Furthermore, Chapter 4 also showed how it was possible to compute and use an index over the proposed compressed structure, allowing extraction of reads, positions, and reference names, without fully decompressing the whole data.

The second SAM field on which this thesis focused is the Quality field. Chapter 5 studied this field, centering the investigation on lossy compression schemes, seeking to minimize the possible effects on the results obtained using downstream applications. Two approaches were discussed, P-BLOCK and R-BLOCK, where both of them group neighboring quality values into blocks, following a specified criteria (*Mean Manhattan Distance* and *Max:Min Distance* respectively) controlled by a user-defined parameter. From the experiments performed, it is possible to concluded that P-BLOCK, R-BLOCK, and the QUALCOMP method of Ochoa et al. [OAB+13] (Section 3.2.4) offer better space versus fidelity trade-offs than all the other methods studied in this thesis, when different fidelity criteria (see Table 5.1) were used to compare the effect of lossy transform on the quality scores. Additionally, Chapter 5 includes a study of the effect of using lossy Quality field compression methodologies in the variation calling process. From the experiments it was evident that P-BLOCK, R-BLOCK, and QUALCOMP give higher accuracy than other techniques for a given level of precision or recall (with respect to the variants found using the original files), allowing more compact storage of quality scores than other mechanisms. Furthermore, the results showed that our approaches offered better compression than QUALCOMP, when precision and recall over 99.0 per cent are required.

Finally, in Chapter 6 we described CSAM, a new SAM file compression format that offers lossless or lossy compression. The performance of CSAM was compared with other specialized SAM compressors (BAM, NGC and CRAM). These experiments concluded that, while it was possible to obtain higher compression
Chapter 7 Conclusions

with other methods, these approaches need a reference sequence as input (which was not considered as part of the space occupied by the final compressed file) to the decompression and compression processes. Moreover, with the exception of BAM, all the existing methods only offer full compression and decompression of files, without supporting random access to the stored data.

BAM and CSAM are currently the only lossless formats in which all of the original values of the all fields are unchanged for SAM files, and that also support random access to the data. CSAM uses less space than BAM and takes similar, or lower, times to compress and decompress the data. In the lossy case, CSAM-P and CSAM-R (the suffix P and R referring to the lossy Quality field method used) require slightly more space than the lossy versions of NGC and CRAM, but the CSAM approaches are self-contained and independent of external reference. Another reason why these techniques are preferable is that they provide faster compression and decompression times than the other SAM compressors that were measured, particularly so when random access is required. Also in Chapter 6 we demonstrate that CSAM offers competitive time performance to BAM for the operation getInterval which, given a reference name and a position interval, extracts the reads and their respective associated fields that are within the interval.

Additionally, with the intention of exploring how CSAM could affect possible downstream applications of SAM and/or BAM files, we showed that it is possible to speed the process of computing coverage in a given list of genomic features (chromosome intervals and aligned read attributes desirable). In order to demonstrate this result we used the downstream application featureCounts [LSS14] which receive as input a SAM or BAM file and the list of genomic features.

During this investigation relevant information from CSAM files was extracted and then given to featureCounts as input. From the experiments it was observed that, depending on the genomic feature queries, not all the SAM fields needed to be examined, which led us to introduce the Simple SAM Notation intermediate format. This format is a SAM file where all the fields that are not involved in the coverage computation process, are replaced by an empty value. Using this format, and modifying the getInterval operation of CSAM, to only extract the fields needed,
Chapter 7 Conclusions

it was experimentally determined that it is possible to decrease the time taken by featureCounts.

The results discussed in this thesis showed that there is still room for improvement in the compression of SAM files. This conclusion is inferred by the evidence that it was possible to reduce the space used to store the files, by treating SAM fields independently. Moreover, given that Bioinformatics is a new, fast-growing research area, the format of genomic files will likely continue to change and evolve. This fact means that new data fields may appear, some of the existing fields may change or disappear, or that even that the SAM format will stop being considered standard format, giving way to a complete new format. A method that compresses each field separately is better equipped for these possibilities, offering compression techniques that would be useful not only for SAM files, but for potential new genomic data storage formats.

Future Work

This work mainly focuses on compressing the Read Sequences and Quality fields of SAM files. An obvious future step of this work is to explore better ways to represent each of the remaining fields, instead of just compressing them using general purpose compressors. Also it would be desirable to include a study of the possible associations between fields, and their importance in the downstream applications studied, or to new prospective applications.

In this thesis we described two different downstream applications, variation calling and feature counting, both of which use SAM/BAM files as input. While the impact of applying our proposed compression approaches to the data used by these two applications has been demonstrated, we did not include a study of how these tools work (that is, how the input data is processed) so that the proposed methods could be added internally to valid input of these tools. An interesting future research topic would be to explore other uses of stored genomic data, and support these functionalities within the compressed data format. For example, it
would be desirable to be able to find variations and output the respective VCF files without the need for any external tool.

Most of the compression methods chosen within CSAM were selected by studying and analyzing the components involved in the compressed representation of SAM files. These studies are based on statistical analysis of the values in each of the components, without considering the biological meaning of the data. For example, we did not study how the compression of the Read Sequences field could be affected if we considered indels, soft clipping, paired ends reads, or any other information available from the SAM fields. Including these factors could lead to higher compression at the cost of requiring more compression-time analysis.

Bioinformatics is a new, rapidly evolving field. Since the core work of this thesis was completed, new ideas have continued to emerge. As future work it would be desirable to include these new works and compare their performance against CSAM. An example is the ORCOM method proposed by Grabowski et al. [SGR15] (described in Section 3.1.5) which was officially published this year. A future obvious step is to compare the results obtained with ORCOM to the ones displayed in this work, also studying how the methodologies proposed by Grabowski et al. could be changed, with the purpose of making random access to the stored data possible.
Bibliography


200


Minerva Access is the Institutional Repository of The University of Melbourne

Author/s:
CANOVAS, RODRIGO

Title:
Practical compression for multi-alignment genomic files

Date:
2015

Persistent Link:
http://hdl.handle.net/11343/55622