Compression of Short Read data sets

Master’s thesis

Eirik Mistereggen
Institute of Informatics
University of Oslo

August 4, 2011
Abstract

With the rapid progression in the field of DNA sequencing, an appreciable part of total costs has moved from the wet lab to the server room. New platforms are generating data that increases in size faster than the storage capacity. More effective compression than generic techniques is possible due to the traits inhabited by the sequencing output. In this thesis we look at using differential compression both for raw unmapped reads and reads that been mapped to a reference genome. For the aligned reads this means using a reference-based solution where the information in the reads is stored as a differencing against a reference library. For the unaligned reads we use an intra-frame differencing similar to frame-to-frame approach in video compression. With this approach we are able to show a proof-of-concept where significant compression rates are attainable in both cases.
# Contents

1 Introduction 8  
1.1 Problem and contributions .......................... 8  
1.2 Research questions ................................. 9  

2 Background 10  
2.1 DNA Sequencing ...................................... 10  
2.1.1 DNA ............................................ 11  
2.1.2 Conventional Sequencing .......................... 11  
2.1.3 Next-generation Sequencing technologies ........ 15  
2.1.4 Sequence assembly ................................ 16  
2.1.5 Sequence Read Archive ............................ 18  
2.2 Compression .......................................... 18  
2.2.1 GZIP .......................................... 20  
2.2.2 Edit distance ...................................... 22  
2.2.3 Delta encoding .................................... 22  
2.2.4 Minimal string difference ........................ 23  
2.2.5 Block encoding ................................... 24  
2.2.6 Range encoding ................................... 24  

3 Design 26  
3.1 Research method ...................................... 27  
3.2 Encoding ............................................. 27  
3.2.1 Differential compression .......................... 27  
3.2.2 An edit distance metric tailored for short reads . 29  
3.3 Processing an unaligned independent short read set .... 30  
3.3.1 Processing of the data .............................. 30  
3.3.2 Complementing traditional GZIP ................. 31  
3.3.3 Accessibility ..................................... 31  
3.4 Processing aligned reads with reference genome .... 32
## Implementation

4.1 A prototype script ........................................ 33
  4.1.1 Close match generation ......................... 33
  4.1.2 Block Encoding .............................. 35
  4.1.3 Data structure ................................ 37
  4.1.4 Shortcomings ............................. 37

4.2 Object-Oriented rewrite ............................. 37
  4.2.1 Class overview ................................ 37
  4.2.2 Range Encoding .............................. 42

4.3 Modifications for processing of aligned reads with reference genome ................. 45

4.4 Unit Testing ..................................... 46

## Results

5.1 Compression of independent short read data set ................. 48
  5.1.1 Illumina run with a read length of 36 base pairs ........ 49
  5.1.2 Illumina run with read length of 50 base pairs ........ 51
  5.1.3 Illumina run with read length of 75 base pairs ........ 54

5.2 Compression of aligned reads with reference genome ............ 58
  5.2.1 Illumina with 36 base pairs .................... 58
  5.2.2 Illumina with 76 and 67 base pairs ............ 60

## Discussion

6.1 Analysis of results .................................. 63
  6.1.1 Compression of unaligned vs aligned reads .... 65

6.2 Design choices ................................... 65

6.3 Implementation issues ................................ 65

6.4 Related work ..................................... 66
  6.4.1 General DNA compression ........................ 66
  6.4.2 Read processing ................................ 66
  6.4.3 Recent advancements ............................. 67

6.5 Usefulness ....................................... 67
  6.5.1 Time versus space ................................ 68
  6.5.2 Future in Sequencing .............................. 68
  6.5.3 Workstation compression .......................... 68

## Future work

7.1 Encoding improvements ................................ 70

7.2 Machine profiling ................................... 71

7.3 Parallelization .................................... 71

7.4 Possible combination with Fritz et.al .......................... 72
List of Tables

3.1 Our edit operations ................................................. 30
5.1 36 bp Independent results ....................................... 49
5.2 50 bp Independent results ........................................ 51
5.3 75 bp Independent results ......................................... 54
5.4 36 bp Aligned results ............................................. 58
5.5 76/67 bp Aligned results ......................................... 60
# List of Figures

2.1 Chemical structure of a Nucleotide. Source: Science In School. 2011 - Issue 18 - Uracil. ............................. 12
2.2 Two DNA chains as a double helix. Source: Science In School. 13
2.3 An incremental reading of all the terminated fragments sorted by length. .............................................. 14
2.4 The relation of throughput vs read lengths for different sequencing platforms ........................................ 17
2.5 Overlapping reads assembled into a contig. ................. 17
2.6 An overview of the data structure for an example run in SRA. 19
2.7 Figure showing LZ dictionary coding technique, referencing increasingly longer strings. .......................... 21
2.8 A figure showing the relationship between frame types in video compression. Source: Wikipedia. ...................... 23
2.9 Range encoding example for alphabet; A, B and EOM. Message is AABA<EOM>. Source: Wikipedia. ...................... 25

3.1 An example of read coverage and overlap for a segment of a genome. Source: [5] ........................................... 28
3.2 Shows the layout of one line of processed output. ............. 31
3.3 Shows the layout of one line of processed output for aligned reads. ....................................................... 32

4.1 First encoding design .............................................. 35
4.2 Class overview from Reader to OperationSets. ................. 38
4.3 Further view from OperationSets to Operations. ............... 39
4.4 Range encoding implementation ................................. 42
4.5 UTF-8 Multi-byte design. Source: UTF Reference manual. 44

5.1 The spread of operations on compression of a single Illumina read set with 36 base pairs ........................... 50
5.2 The number of hits each key frame read found in its operations on a read set with 36 base pairs .................... 50
5.3 Operations used in compressing a Illumina set with longer 50 bp reads ........................................... 52
5.4 Distribution of uncompressed key frames and frames represented as deltas with 50 bp reads ............... 53
5.5 Operations used in compressing a Illumina set with 75 bp reads ..................................................... 55
5.6 Distribution of uncompressed key frames and frames represented as deltas with 75 bp reads ............... 56
5.7 Shows the amount of compressed close matching frames found for the 150k first key frames .................. 57
5.8 Operations used in compressing a Illumina 36bp set with reference ..................................................... 59
5.9 Distribution of compressed matches and uncompressed reads on 36bp Illumina set with reference ........ 59
5.10 The spread of operations on compression of a set with HG19 as a reference ..................................... 61
5.11 Damerau-Levenshtein distance from the dataset to the reference genome ........................................... 62
Listings

4.1 Enumerating possible corrections for a word with Damerau-Levenshtein distance of 1 ........................................ 33
4.2 First implementation of a close match generation for reads ... 34
4.3 The first range is calculated. ........................................ 43
4.4 Example output lines from unaligned read processing. .......... 44
4.5 Example input lines from aligned reads ............................ 45
4.6 Example output lines from aligned reads ............................ 45
4.7 Example of a test for a close match generation class. .......... 46
Acknowledgements

Thanks go out to supervisors Geir Kjetil Sandve and Anja Kristoffersen for their help and guidance. I also want to thank the other lecturers in Introduction to Bioinformatics for welcoming us into this field, and the rest of the students on the tenth floor for the comradery. Additionally apologies goes out to friends and family, who had to endure the entire summer of 2011 without my awesome presence.

Eirik Mistereggen
University of Oslo
August 2011
Chapter 1

Introduction

1.1 Problem and contributions

In today’s DNA research environment there is a luxury problem; we are generating enormous amounts of biological data with interesting research applications, but it’s almost too much too fast. To fully exploit these results they need to be stored and processed, and we need to do this in ways that will scale as the data continues to flow. One obvious solution is to compress the data and process it for convenient storage, while keeping it indexed for easy access for later reprocessing. The most common tools today are generic compression tools not created for DNA data specifically. They achieve compression with varying unsatisfactory results, while leaving the data in a binary data format.

In this thesis we explore the possibility of applying techniques that takes information about DNA data into account:

- We suggest two different approaches for unaligned and aligned reads respectively in chapter 3. For aligned reads we suggest a differential encoding against a reference library. For unaligned data we introduce a intra-frame differencing, utilizing genome coverage and resulting redundancy to reduce size.

- We go over the actual implementation details in chapter 4, where we follow the evolution from a simple script to a flexible program.

- We present the results for unaligned reads in chapter 5, where we show that in the best case we are able to achieve a size of about 5% of the
original size when used together with GZIP. Here we also show how the performance of our approach degrades strongly with increasing read lengths.

- We present how the inverse is true for aligned reads in section 5.2, and that this leads to us being able to achieve a size of 4% of the original size together with GZIP.

1.2 Research questions

The problem consists of these research questions:

1. Is a differential compression approach feasible for unaligned read data?
2. Can we represent the information in aligned reads through positions and differences to a reference genome?
Chapter 2

Background

This chapter is written in a way that it may serve as a reference when we later are explaining some design choice, research or implementation, and it would be too much of a digression to explain the concept in depth at that point in the thesis. Therefore you may be best served just skimming the outline of some of the more deeply covered subjects, such as compression. The sections concerned with the biology background of the thesis is heavily biased as to provide a superficial understanding from a programmers point of view, just giving a sufficient overlook of the sources of the data we will be handling.

First we will briefly introduce the information storage that is DNA molecules. After that we look at conventional sequencing techniques from the late 70’s, before we delve into modern sequencing, which is where the data used in this thesis comes from. Secondly we look at a popular general compression algorithm and its components.

2.1 DNA Sequencing

Sequencing of DNA has a variety of usages.

Comparative Genomics is the study of how the genome of different species compare to each other, to see the similarities in structure and function and how these have evolved. Comparative approaches can also be used to find out which parts of a gene is non-coding and if not what function the functional parts have [6].
We can sequence DNA from mutated cells, e.g. a cancer cell, and compare it to a healthy gene. This way it is possible to better understand the cancer genome. There is hope that with the advent of modern sequencing we will see a growth in personal genomics, where individuals easily can get their genome sequenced. This information can then be used in fields like predictive medicine, possibly preventing future diseases for individuals.

2.1.1 DNA

DNA is a chain-like molecule, where every link is called a nucleotide. The 4 famous nucleotides in DNA are

- **A** Adenine
- **C** Cytosine
- **G** Guanine
- **T** Thymine

The nucleotides are structurally similar. In common they have a sugar ring, a phosphate part and an organic base ring [2]. What differs them is the organic bases, composed of carbon, nitrogen and oxygen. The nucleotides forms a chain by connecting together via chemical bonds. DNA chains mostly exists as the well known double helices, where two single DNA-chains is positioned so their nucleobases can interact with each other and form a base pair [25]. The A nucleotides always pairs with the T, and the G with the C. This means that we only need one of the chains to duplicate the entire structure. This is central to the concept that leads to the sequencing of the bases.

2.1.2 Conventional Sequencing

The original method of sequencing was to separate the double helix form, then generate a set of fragments that are terminated to differ in length from each other by a single base [20]. The terminators are fluorescently labelled so they can be detected in automated machines. This way the bases at the end of the fragments, sorted by length, are identified. The infrastructure surrounding this type of sequencing is very laborious and costly. The preparing of the samples to be sequenced is inefficient and the sequencing produces small amounts of data by today’s standards.
Figure 2.1: Chemical structure of a Nucleotide. Source: Science In School. 2011 - Issue 18 - Uracil.
Figure 2.2: Two DNA chains as a double helix. Source: Science In School.
Figure 2.3: An incremental reading of all the terminated fragments sorted by length.
2.1.3 Next-generation Sequencing technologies

Up until the 2000’s the field mostly revolved around incremental improvements to the conventional method, by improved enzymes to sequence with and better ways to detect the fluorescent data. The demand to drive costs down fueled research into newer and faster methods. Here we will look at only a few of them. Also note that finding up to date and reliable data about the different technologies is a challenge. Many promise more than they can deliver due to the intense competition in the field. Information about the technologies ages fast due to the frequent advances made by the different companies. The most valuable information for us however is the individual novelties for the different systems, which tend to be bound to the underlying concept of their sequencing approach, and thus doesn’t change in the small incremental advancements in the equipment. [21]

454 Life Sciences

This platform was the first available of the next-generation machines. It uses what is called pyrosequencing with emulision pcr [18]. Compared to the other platforms it provides quite costly sequencing at $60 per megabase (as of 2008), but it produces the by far longest reads of all the technologies, with 250 base pairs being a standard length [21], making it ideal for de novo sequencing. Because of a curiosity in the sequencing of homopolymers (consecutive sequences of the same base) they determine the length of these by a fickle signal intensity, leading to a high error rate for this specific occurrence. It is important that we note that this implicates a high number of insert-deletion errors in the read data.

Illumina Genome Analyzer

This is another machine that reads the bases from sequential images. It uses a different technology with dye-terminators, but the concept of capturing images of fluorescently labeled nucleotides remains similar. The machine can sequence up to 240 million reads in one run. Read length starts at 36 base pairs, which we will later see is suitable for our methods in this thesis. When using paired-end reads it is capable of longer reads at 100 base pairs and increasing with time. Longer reads is also possible, but generally inferring a penalty to the error rate. We make note of the fact that the most prominent type of error in the reads from this machine is substitutions [21]. The Illumina
machine is widely used and is the machine that the data used in this thesis is from. The machine has a successor called HiSeq2000, which basically uses the same method, but has improved pipeline size (one more flowcell) and better hardware (camera, laser). This makes the machine capable of sequencing 2 complete genomes in 8 days.

**Life Technologies SOLiD**

This technology also uses emulsion pcr, but sequences with a synthesis called ligase. It provides sequencing at a cost similar to Illumina, $1 per megabase. It also shares the fact that the most prone errors are substitution. Even though it has these substitution errors, it is listed as having 99.94% accuracy.

**Summary of the platforms**

As you can see in figure 2.4, the different platforms represents a trade-off between throughput and read length. Throughput here is the data amount per cost and time. In addition comes the question of the accuracy of the reads. One advantage of the new technologies is with resequencing experiments, variation detection in germs and mutations. The cost of sequencing has been shifted from the biological infrastructure to the data postprocessing work.

**2.1.4 Sequence assembly**

Since it is unfeasable to read whole genomes in one go, they are read in reads ranging from 20 to 1000 base pairs in length depending on the technology used. This makes the resulting data consist of millions of these small pieces, with high physical redundant coverage of the genome. The overlap is used to assemble the reads back together in contiguous sequenced regions, or contigs, as seen in figure 2.5.

**De novo assembly**

If the reads are a result of a de-novo sequencing run, there exist no previously known assembled sequence of this DNA. This is the most demanding type of assembly since every read has to be compared to each other.
Figure 2.4: The relation of throughput vs read lengths for different sequencing platforms

Figure 2.5: Overlapping reads assembled into a contig.
Mapping assembly

In this case you have a reference genome to map every read onto, and is therefore less computationally intensive. Because many model organisms, like Homo Sapiens, have already been assembled as a whole genome short reads become more convenient to use. This is the usual case in variation detection sequencing.

2.1.5 Sequence Read Archive

Since data is trimmed and information lost when doing alignments on sequencing output, there is a need to somehow store the unaltered output. When research is published, the raw sequencing data can be uploaded to SRA and others can later go back and do reanalysis of the data sets [12]. This is beneficial when there is interest in confirming the results of a study, or reusing the raw data in a way not envisaged by the original study authors. Perhaps the most common issue is if a study used an old genome assembly and someone wants to redo the assembling to a newer reference genome.

An easily understandable analogy would be of photographers shooting and storing pictures in the Raw image format. In the same way this data is minimally processed and contains the maximal amount of data from the camera image sensor, the raw read data contains direct output of the sequencing platforms. This gives the photographer flexibility in altering the image later, processing it and change the output entirely. Raw read data gives researches the same options.

Format

We see an overview of the data format used for SRA in figure 2.6. What is novel about the SRA format is the mix of a file-based management with a relational database. In this paper we deal only with the “Read” column of the data format, leaving the other columns in the data storage untouched.

2.2 Compression

Information intended to be used by humans tends to be highly redundant. The prime example in texts explaining this tends to be natural language,
Figure 2.6: An overview of the data structure for an example run in SRA.
but ironically the information humans contains, DNA, is also redundant in nature. Combined with the modern high-throughput sequencing mentioned previously we have data that seems prime for compression. We first take a look at the most standard compression choice.

2.2.1 GZIP

GZIP will be used as a benchmark, de facto compression standard and final compression step in this thesis, so it serves us well to take a deeper look at what it does. It is used here to quickly zip the data and make it smaller for transfer and or storage. As prof. Skiena suggests, what Lempel-Ziv does, it does very good. It would be a perilous journey to try to outdo it on its home field of low level compression, but with some insight into its ways perhaps we will see some ways we can preprocess our data to make it better suited for GZIP.

GZIP depends on an algorithm called DEFLATE. Mostly popular for its independency from patents, Deflate combines several techniques to compress the data. Here is an overview of the steps in the algorithm.

Lempel-Ziv

Lempel-Ziv explained two lossless data compression algorithms in 1977 and 78 and thus called LZ77 and LZ78 respectively. LZ78 was a small modification, but for various patent issues this has been less popular, mostly used in the implementation known as LZW. More implementations also exist, LZSS and LZMA, but we chose to only look at GZIP and its LZ77 process here.

LZ77 is a dictionary coder, which reads the stream and keeps track of the most recent data in a sliding window. If it encounters a sequence it has in its dictionary it saves a backpointer to this data. LZ78 also check the future of the stream against the dictionary.

We see that in the beginning of the sequence it gets stored uncompressed, but quickly backpointers make up most of the data.
Huffman coding

All these backpointers may end up pointing to rather short sequences of bytes, and end up needing to be compressed themselves. To do this GZIP uses Huffman codes, described by David A. Huffman in 1952 [11].

The method takes a set of symbols and their probability mass function, and finds a prefix-free binary code for every symbol that yields a minimal codeword length with regards to the probability of every symbol.

GZIP uses this internally in a number of ways, but the important part is that it is able to dynamically update this table of binary codes while encountering the sequence. This means that there is little we can do to gain on the algorithm with regards to the DNA alphabet, GZIP already has it covered.

Run-length encoding

The simplest form of compression, which changes runs of the same data value into the data value and the length of the run. GZIP doesn’t actually do run-length encoding in a strict sense, but uses a generalization that can encode runs of a string of characters. It does this by RLE encoding the backpointers. GZIP also uses RLE internally on its own headers and Huffman tables.

All in all we see that we can trust GZIP to do the basic compression tasks for the short reads. Conventional Huffman coding theory would say that we need one specially tailored for the probability mass function that the set of short reads yields, but GZIP solves this by adapting to the input statistics. The sliding window size is a cause for concern. Since we know that our data has long common substrings, spread across a gigantic data set, it would seem that GZIP would be stuck in the small window and pointing back to small
sequences of DNA, when in reality it is possible to point to almost entire reads.

2.2.2 Edit distance

Levenshtein distance, often referred to as edit distance, is a difference metric between two sequences. The metric is defined as the number of operations needed to transform one string into the other, where the operations are insertion, deletion and substitution. There are several metrics for measuring edit distance. Another metric that is relevant to us is DamerauLevenshtein, designed for human misspellings. It adds the transposition of two adjacent characters to the set of operations possible in the distance function. This is helpful since transpositions might be the variation between two strands of DNA.

Basically we will get the need to do our own edit distance metric, and choose operations that seems to serve us best.

2.2.3 Delta encoding

Delta encoding is a data transformation technique that saves two sets of data as the first set and the difference from that set to the other. Delta encoding is a key approach to reduce data redundancy, and is therefore of interest to us because of the nature of short read datasets covered earlier.

One of the most used implementations is the Unix utility diff, which is used to show deltas (or diffs) over entire files. Many standardized formats for delta encoding exists, but none of them produce minimal diffs, and those who do are for highly specialized applications. One notable case of specializing a diff algorithm for increased effectiveness comes in the form of software updates in the Google browser project Chromium [7]. They explain how they use knowledge about the data they are compressing to maximize common substrings and make differential encoding more effective for them than generic binary diff.

Video compression

Another example of differential compression is found in video compression. Every relevant video compression codec uses some form of interframe com-
Interframe compression utilizes the fact that most of the frames in the sequence is not an entire set of new information, but rather a product of one or more earlier frames and a set of changes. For the sake of the transformation of this idea to the area of short reads we will explain some of the different frame types used, as can be seen in figure 2.8.

- An I-frame, or key frame, is a fully specified frame with all the information required to reconstruct that frame by itself. This is the frame type that other frames will be represented by.
- A P-frame, or delta frame, is a frame that holds only the difference in the picture from the previous frame, the key frame.
- A B-frame, is specified by the difference from both the preceding and the following frame.

In practice video encoders split up the images into blocks. Key frames consists only of full information blocks and take up the most space. Delta frames can consist of predicted blocks that only consist of delta information from the preceding key frame, and full information blocks if no efficient delta was found. We will end up using some of the terminology from video compression, as the concept used is somewhat similar.

### 2.2.4 Minimal string difference

The problem of representing the difference between two strings in the minimal amount of space is well known. Goldberg deals with the problem of representing string differences for a general corpus and strings of any length [8]. It also has no limitation as to the count of different operations on the origin string to the destination string, leading to arbitrary sized differences. This combined leads to an algorithm that is quadratic in time and space, making it somewhat unsuitable for large amount of strings of some length.
2.2.5 Block encoding

Block encoding is a concept used in coding theory for reliable transmission of digital data, covered in [14]. The basic idea is transforming a sequence of symbols into a fixed-length code word. Transforming the symbols into the code word adds redundancy to the information sent. In coding theory this is used to create error codes to be able to recover from missing data, but in our case we will try to transform our edit operations into code words as effectively as possible, described in 4.1.2.

2.2.6 Range encoding

Similar to block encoding, range encoding encodes all symbols of a message into one code word, representing a number [15]. In the basic case initial ranges are set up with size proportional to the probability of the symbols in the alphabet. For each symbol added to the message, you continue down into the probability assigned range, and divide up into subranges. We see an example of this in figure 2.9. The start range is [0, 100000), and the probability of each symbol is represented in area size; 60%, 20% and 20%. We simply follow the range given by the symbol down to the last <EOM>symbol, showing that any number between 25056 and 25920 would represent the sequence given.

Problems with this method in general compression is selecting a range that is so large that you don’t run out of space as you add symbols to the message. You also need to make sure that the ranges either are static and that both the encoder and decoder has the same, or you can have a dynamic range and add that as a header to the message. As we will see in 4.2.2 we decide to go with a static range.
Figure 2.9: Range encoding example for alphabet; A, B and EOM. Message is AABA<EOM>. Source: Wikipedia.
Chapter 3

Design

[...] it is very difficult to beat Lempel-Ziv by using an application-specific algorithm. My recommendation is not to try. If there are obvious application-specific redundancies that can safely be eliminated with a simple preprocessing step, go ahead and do it. But don’t waste much time fooling around.

S. Skiena - Algorithm design manual

From the beginning of the solution design process, the case of processing the raw sequence output from a sequencing platform has been central. Even though the necessity of archiving read data and its various data types is a debated topic, there seems to be a widely accepted opinion that read data should be archived for a slew of different reasons.

From the SRA Handbook [12];

- For re-alignments when improved alignment algorithms become available.
- To regenerate an alignment when a reference assembly is updated
- To generate an alignment to another reference assembly
• To pool data across different experiments or create experimental subsets from within an experiment

In the background we have covered the nature of short reads and their traits. In this chapter we introduce how we attend to apply some of the different encoding techniques mentioned in 2.2 to exploit redundancies in the data. We then describe a design for two different use cases, processing with and without a reference genome, aligned and unaligned.

3.1 Research method

Our thesis resembles in part an empirical cycle of research. From observing short read data, a hypothesis emerged; that short reads can efficiently be compressed with differential compression, between the reads themselves for unaligned reads, and to a reference genome for aligned reads.

We design and implement a prototype with the processing of short read data, and deduce how this prototype should perform on different sets of data. We then run the prototype on different forms of short reads, with different lengths were we suspect our results will worsen.

3.2 Encoding

3.2.1 Differential compression

Our general plan of attack comes from the realm of video compression, and how it uses differential compression.

The idea is that we can use the same approach to process short reads with some alterations. We will view every short read as a frame of information from the complete genome. From this we will aim to represent a few reads as key frames, with the full set of information, but represent a much larger set of reads as delta frames. We believe that this will be able to give a good compression because of the nature of short reads. The fact that they are all snapshots of a single genome with overlap and repeated data makes them similar to a set of images with small and no differences, as seen in figure 3.1.
Figure 3.1: An example of read coverage and overlap for a segment of a genome. Source: [5]
In 2.2.4 we mentioned the problem of representing string difference in minimal space. In our case we have strings of a concrete length n, all read data used here has either one or two constant lengths, thus we don’t need to consider the operations that alter the length of the string. Since the algorithm is quadratic and we need to do this step for millions of reads we need another approach. Instead we just generate all the frames that are 1 edit distance away from the key frame. This means that we will not be able to match reads that are 2 edits away from each other, but in return we will be able to represent the difference in little space and generate fast.

Reads differs from video in the way that the reads that are close to each other in difference does not have to be close in the sequenced data. In video it will always be the subsequent frame that we try to represent as a difference, but in our case there is no limit as to how many or where the read is. This means we will have to search the read data for these generated matches.

3.2.2 An edit distance metric tailored for short reads

We mentioned Levenshtein distance in 2.2.2 and its human error related Damerau-Levenshtein, two ways for measuring difference between two sequences with a set of allowed operations. Now we could just use the operations in these metrics, but examining our data we see it doesn’t quite fit the description of our solution.

Our reads are of constant length. Meaning from one key frame to a close match we will never want to insert or delete a base pair. Stripping these two operations from the set we now have what is called the Hamming distance metric [17].

From the Damerau-Levenshtein distance we add the transposition operation, since this is common in genomic variation and sequencing errors.

We would probably cover a lot of reads with these common operations, but looking at figure 3.1, another trivially encoded set of operations strikes us as promising: shifts. We see that a lot of reads overlap while being shifted slightly. With partially overlapping reads like this we should be able to represent more similar reads by shifting the data left and right.

We therefore end up with a distance metric looking like this;
Table 3.1: Our edit operations

<table>
<thead>
<tr>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replace</td>
</tr>
<tr>
<td>Transposition</td>
</tr>
<tr>
<td>Single shifts</td>
</tr>
<tr>
<td>Double shifts</td>
</tr>
</tbody>
</table>

3.3 Processing an unaligned independent short read set

The main focus in this thesis is on the processing of raw short read data. This is the direct output from the platforms mentioned in 2.1.3. As mentioned earlier this is the format that is best suited for long term storage if there is reason to believe that anyone would want to revisit the experiment later [12]. Throughout the design we will keep SRA suitability in the back of our heads, but the focus is more on a general proof of concept than an out of the box compatibility with the SRA format.

3.3.1 Processing of the data

In its raw form the data from the sequencing platform often contains several meta data fields that accompanies the actual nucleotide reads from the sequence. They can be seen in the SRA format overview in figure 2.6. Most important is perhaps the Quality score, which is the basis for tools with applications like error correction and reliability measures. The other column that is pictured, intensity data, is less essential to store for most projects, but SRA gives the option to store it if needed.

For the sake of convenience we extract the read data from the raw data, but it is an important note that in theory we are able to attach the other SRA columns and be a lossless process that can recreate the original data in its entirety. Other tools like FASTX-Toolkit provides an easy way to do processing like collapsing all equal reads into one, thus losing the connection to meta-data. We avoid this by still having the reads as an entity that can have a reference to associated data. This associated data can be stored in separate files and be conventionally compressed on their own.
3.3.2 Complementing traditional GZIP

As noted in section 6.4.2, GZIP gets severely limited when applied to short read data sets, by the fact that it has a small sliding window of the data it has read, which it can reference to when encountering new data [24]. Even though you can spend memory and computation to increase the size of this window, the sheer size of the data sets makes this futile. Consider the case that we have a dictionary coder that references the entire space of the data set. The references would become too big themselves to be of much use.

What also is hampering compression with GZIP is in its nature as a dictionary coder; its sequentiality. Meaning each read is an independent set of data, not depending on where it is in the set. We can completely disregard this, and skip past a huge hurdle that GZIP has as a lossless encoder. There simply is some entropy in the set of reads that we don’t need, and can get rid of. We do this by mimicking video compression and move all the similar frames directly behind the key frame, negating any space taking sequence position reference, as seen in figure 3.2.

By moving the close matching reads to a key frame like this, we not only save space directly in referencing it, but we also get maximum efficiency out of the basic compression methods in GZIP, like advanced run length encoding of the repeating byte-encoded differences.

3.3.3 Accessibility

The design in figure 3.2 also offers benefits in accessibility directly, with no uncompression, compared to standard GZIP. When searching for a read in the data set, one can now do the same process reversed. Compute the deltas of the string you are searching for, and see if any of them are in the set as a key frame. When you find the key frame you can easily iterate through the encoded frames to find your hit, and the associated data.
3.4 Processing aligned reads with reference genome

Previously we have dealt with the case of a set of raw reads, but we also have a second possible use case. After the raw read data set has been run through an alignment process mentioned in 2.1.4, we get a file consisting of the reads accompanied by the position they have been aligned to in a reference genome.

When the reads have been aligned there is no longer a need for searching the complete set to find data to difference against. It is now more resembling of the video case with having a close matching frame available directly in the reference genome.

The reads either come in chronological sequence, or we can process them to be chronological. This enables us to encode the position in the reference genome relative to the position of the previous read, which makes it easier to compress.

Figure 3.3: Shows the layout of one line of processed output for aligned reads.
The design ideas covered in the previous chapter has been implemented in an iterative fashion. In this chapter we will describe both an early prototype script and the beginnings of a more comprehensive flexible program. We first had a raw immutable script, with hardcoded alphabets, lengths and operations. We then transformed this into an Object-Oriented program, which is suitable to be extended and modified.

4.1 A prototype script

4.1.1 Close match generation

In the design we covered how we want to find all the reads that are candidates for being represented as a delta of the key frame. Given that reads are not ordered, we need to search the set for these matches, and generate these candidates that we are searching for.

This problem is very similar to how spell checkers are implemented. In Peter Norvigs small article 'How to Write a Spelling Corrector' [16], he provides a basic python function to generate the set of all possible spelling corrections c for the word w. We see the code he suggests in listing 4.1.

Listing 4.1: Enumerating possible corrections for a word with Damerau-Levenshtein distance of 1

```python
def edits1(word):
```
These all have an Damerau-Levenshtein distance of 1 from the corrected word. As Norvig notes, “This can be a big set. For a word of length n, there will be n deletions, n-1 transpositions, 26n alterations, and 26(n+1) insertions, for a total of 54n+25” corrections.

We need to implement a similar generation for our own edit distance metric that we defined previously. Since our reads have constant length, we remove the deletion and insertion sets. For the reasons mentioned in 3.2.2, we also want to add generators for shifted candidates of the read.

In listing 4.2 we see our first implementation of an adopted version of the close match generator.

Listing 4.2: First implementation of a close match generation for reads

```python
def close_matches(read):
    splits = [(read[:i], read[i:]) for i in range(len(read) + 1)]
    deletes = [a + b[1:] for a, b in splits if b]
    transposes = [a + b[1] + b[0] + b[2:] for a, b in splits if len(b) > 1]
    replaces = [a + c + b[1:] for a, b in splits for c in alphabet if b]
    inserts = [a + c + b for a, b in splits for c in alphabet]
    return set(deletes + transposes + replaces + inserts)
```
Figure 4.1: First encoding design

```
right_shift + double_left_shift +
double + right_shift)
```

Since we will have a smaller alphabet than the natural language implementation, the size of our set looks a bit different. The length n will generally be longer in our case. We have the same n-1 transpositions, and let’s say the alphabet size is the standard 4 nucleotides, so we have 4n replaces. In addition we have added one and two position shifts to our set, which gives us 2*4 and 2*16. Our new set has a total of 5n+39 close match candidates for every read. Our first prototype had read lengths of 30bp, which gave us a set size of 189 matches to encode. Even these reads were enough to show that encoding differences with an edit distance of 2 would be impractical, as that would give us a set of 35721 matches, with this distance metric.

### 4.1.2 Block Encoding

So we have the same set of operations for all the key frames we select. The problem is now to represent all these operations using the least amount of space.

The most trivial approach is block encoding, as introduced in 2.2.5. For every single variable in a operation, enumerate it and encode it in binary, and encode all the different types of operations in binary and add that to
the individual encodings. We can see this approach in figure 4.1. We encode the operation in the first two bits as seen. E.g. swap becomes 01, replace 10, shift operations 11 and so on. The replace operation would be represented by first encoding the position of the replace, and then an enumeration of the character in the alphabet, 00 for A, 01 for C, 11 for G and 10 for T. Shifts would be encoded by one field for the direction, one for the length, and then the characters.

This approach was coded in the prototype, and it works as expected. However both the overall idea and resulting code was bad in many ways. First and foremost it wastes space in the encoding by giving every variable in the operations its own amount of binary fields. We will seldom (or never) have the perfect combination of string length (the size of the position fields) or alphabet size (size of the character fields) to effectively use all the fields. It also created some problems in the implementation. We want to be able to help compress several kinds of short reads. With different lengths of the reads and different alphabet. This of course influences the size of the encoded operations in any solution, but it is hard to code an implementation that is both flexible in this regard, as well as having a consistent encoded representation of the different variables.

One apparent problem is that each operation has a different information structure, in that they need different fields and because of this take up a different amount of space. When assigning the first two bits to an operation type we are basically locking out large amount of information space that will remain unused for the low information operations. For the operations with higher information requirement there isn’t enough space to fit in one byte, even though the amount of operations clearly are able to fit in that space.

A quick hack to solve this for 36 base pair reads was to let operations that needed more space, like replace operations, borrow space from the equal operation. We split the position information in half, and let one half be represented by switching a third bit in the equal code word. This enabled us to practically use 5 operations and more efficiently use up the code space.

For the quick development of a prototype the read length and alphabet was kept static (hard coded). There was no easy way to make this encoding dynamic and handle longer reads and different nucleotide symbols. However this enabled us to get a rough estimate of the compression potential for an overall approach like this.
4.1.3 Data structure

To get full compression we need to find as many close matches as possible for every key frame. This means we need to search the whole set when looking for matches, and we need to have it in memory. In the first prototype we had a binary sorted list with the python bisect module [23]. To rapidly develop a prototype, we needed a simple datastructure. The easiest thing to do was just to process all the reads, sort them, and do binary searches to find our frames in satisfactory $O(\log n)$ time.

4.1.4 Shortcomings

This early prototype had several issues that we wanted to address in an eventual refactoring.

- Static close match generation left us with little flexibility in the way of choosing which operations to use in finding delta frames.
- Static block encoding was ineffective in its use of space and inelegant in the way it is impossible to extend the space to encode longer reads and bigger operation sets.
- The data structure had slow lookup and required an unnecessary sorting step of the data, which is very expensive considering the size of our data.

Despite all this it still revealed some promise in the design idea and served as a springboard for a re-design.

4.2 Object-Oriented rewrite

Because of the static nature of the script mentioned earlier, it was hard to improve a section of it without requiring changes overall. It became obvious that a decoupling of the different sections was necessary to make it easier to extend the program later [1].

4.2.1 Class overview

We can see a quick overview of the design in figure 4.2 and figure 4.3
Figure 4.2: Class overview from Reader to OperationSets.
Figure 4.3: Further view from OperationSets to Operations.

The overview is split in two for easy reading. It is split on the class OperationSet, with the only classes used in both being Alphabet and Storage interface.

Reader

Reader is thought of as an interface that has methods that actual implementing classes will provide to process data into the internal structure. The sentiment is that we in this way decouple the code that actually reads one or more input files, and easily plug in different readers for different tasks and platforms. In our case we have only used two different readers.

IndependentReadReader

This is the reader that was hard coded in our script. It is a simple class that just starts the process of making all reads available in the Storage class. It then transforms the read data into the internal structure of the Frame class and starts the encoding routine.

AlignedReadReader

This is a reader for the second use case mentioned in 3.4. Here we process the data that has the format of a file with reads and their positions aligned to a genome that is in a reference file. Here we don’t use the Storage construction,
but create a NumpyReferenceMap for lookup of referenced positions instead. It transforms the reads into a different frame class PositionFrame.

Frame

This is the main information holding class of the implementation. It represents what can be thought of as the Key Frames in the case of a single independent set of reads, i.e. the reads that will be represented uncompressed in the end result. It holds a reference to a Data class, that can be used to keep track of meta-data for the read through the compression, such as quality scores, intensity etc. Each frame has an OperationSetCollection that is instantiated at creation with each of the existing OperationSet classes.

PositionFrame

This was a class added to handle the frames when used on a aligned read set with a reference genome. It encodes a relative position from the last read, and tries to match an operation to the read in the reference genome. If not it encodes as the relative position and the full read.

NumpyReferenceMap

A thin wrapper around NumPy that handles reading the reference genome from a memmap by the position listed in the aligned file.

OperationSet

A super class that holds common methods for every type of OperationSet. The only method required to implement is a routine to generate all possible operations of that type for the input read. Methods common to all sets are methods to search the storage for the reads generated by all the operations in the set, methods to encode these hits with the correct range passed down from the OperationSetCollection as described in 4.2.2. Every new operation added to the distance metric requires a new OperationSet inheritor as well as another Operation.
**OperationSetCollection**

The collection class that the frame types can use to manipulate their Operations. It holds references to all the different OperationSet classes and the top range table for the range encoding to be passed down to the Operations. It holds convenience functions for the frame to initiate generation and compression of the sets, and searching for reads in the sets.

**Operation**

A super class for the different types of Operations we allow on a read with our edit distance metric. It holds the single read that has been transformed from the key frame through the 1 operation. It holds a common method for the operation to search for itself in the Storage. This class together with OperationSet is what needs to be extended when changing the edit distance metric, and every new operation needs to implement the encode method. The encode function takes every parameter of the Operation, which is different for every operation, and creates sub-ranges that is used in the range encoding.

**Alphabet**

Since we want to be able to compress reads from a variety of platforms and different experiments it is insufficient to hard code an alphabet with A, C, G and T. This means we should be able to take everything in the IUPAC notation [13]. This class wraps a sorted list that holds the characters. It can be instantiated with a complete sequence of the alphabet or added dynamically. It provides methods for the operations to get the encoded representation of a character, as well as getting the size of the alphabet.

This encoded representation of course needs to be consistent in that it always returns the same encoding for a given alphabet. To be able to stay compatible with eventual future versions we would theoretically encode headers for the sets to represent the alphabet and operations used.

**Storage**

The Storage interface is a minimal layer that enables us to decouple the chosen data structure, so it can easily be changed. It provides functions to
insert a read and a reference to its meta-data, as well as a function to search for a read. It also provides function to tag a read as visited, to avoid that a read is encoded more than once.

**HashMap**

This class is a simple wrapper around a python dictionary that implements our storage interface. Every hash key has a list value that contains the meta-data references for every identical read that has that hash key. This way the meta-data can stay with the reads getting compressed as well, and handled in a separate file. We mark the key as visited by changing the key to point to a global value VISITED.

### 4.2.2 Range Encoding

We have differences represented as a set of operations. We want to express these operations as dynamically as possible with as little space as possible. Range encoding usually take a set of symbols and encodes them all into one
number, as mentioned in 2.2.6. Similarly we want to take set of operation parameters, and encode them all into one number.

Since the sets of the different operations are of different size, they get a different part of the range of the number, but this size is static, so there is no need to guess how long the overall range should be. We can precalculate it and have no redundancy in space. We do this simply by running our close match generation and using the size of the different operation sets as ranges for the encoder. We see in listing 4.3 how the outermost range is calculated after the operation set sizes are known. This information can then be written to a header.

Listing 4.3: The first range is calculated.

```python
self.operation_sets = operation_sets
self.offset = []

def populate_offset_table(self):
    total_size = 0
    prev = 0
    for s in self.operation_sets:
        self.offset.append((total_size + prev, s))
        total_size += prev
        prev = s.__len__()
```

For simplicity we give each operations the same probability. In this way we can ensure compatibility with a decoder by just using an extendable header that contains the operation ranges and setup. If we should choose to add completely new operations we can mark that in the header.

We see our new range encoding table in figure 4.4. When decoding the integer, we can immediately see which type of operation is encoded. We then peel another layer to get one parameter of the operation, then we can continue down coding ranges for as many parameters as the operation has. Replace has position and character, Swap has only position, one shift has direction and character, two shifts has direction and two characters, and so on.

Unicode

Now the encoding of operations was made dynamic and able to extend encoding to an arbitrary length of reads and different operations. This means
we would use up the coding space provided in ASCII characters when reads got longer than 43 base pairs \(5 \times 43 + 39 = 254\) as shown in 4.1.1). Instead of switching to use 2 bytes for every compressed read, we wanted to have a Huffman-like [11] way to have variable-length codes. This comes inherent in the UTF-8 design as shown in figure 4.5.

Most importantly it is backward-compatible with ASCII and would represent all operations encoded in the range up to 7F as plain ASCII characters. Then when the encoding table goes outside that range we will get two-byte encoding of those operations without changing our internal structure.

Unicode also has the benefit of being a known format, and enables us to avoid writing out binary data. This is one of the problems that sequence storage systems has with formats like zip. Using unicode our output can easily be read (even though some of the output will be whitespace and control characters). In listing 4.4 we see example output lines from a run on unaligned reads.

Listing 4.4: Example output lines from unaligned read processing.

```plaintext
TATTACTAGAAGAGGAAAACCTTACGTCGTATN <U+008F><times 38>
AGTNTATTAAGGAGAAAGCATACTCGGGGTCGTATN . <U+008F>
TCATATCTTCACATTTCGAAGTACTGTGGGCTTCGN x <times 23>
```
4.3 Modifications for processing of aligned reads with reference genome

As mentioned in 3.4, we thought of a way we could extend an approach to aligned read data, given that the decoder has the reference genome used to align the reads. In the first implementation this would require a complete rewrite into a new program, but with the more separated implementation we could attach the design with a few new classes.

We add a new reader class to read in the data in the form of a read and a position as mentioned in 4.2.1. It parses data in the form shown in listing 4.5. This is just an example as different assemblers can add various fields to the output, such as edit distance. Our reader just strips the lines down to the read and the aligned position.

Listing 4.5: Example input lines from aligned reads

```
14424 36M GATTGGTGTTGGCGTTTTCTCTGAAGCGCTCTTAAAGA
   NM:i:0
14433 36M GCCGTTTTTCTCTGAAGCGCTCTTAAAGAACACAGTG
   NM:i:0
14457 36M GGAGCACAGTGCGGAGGTGGTGAGCCGTCCCC
   NM:i:2
14463 36M CAGTGGCGCGGCTGGGGAAGCGGCTCCCCCATGG
   NM:i:0
```

For ease of coding, we generate the operation set in the same way we do for the other reads, and match these operation reads against the read that is at the position given in the reference genome. In case of a hit of one of the operations we output the encoded operation and the position.

We add these reads as key frames, i.e. every frame is a key frame in our internal structure, but with the added information of a position. Since the positions take a lot of space we encode the positions relative to the last aligned read position. We store the previous position when encoding the current read. We see an example of output lines from our program in listing 4.6.

Listing 4.6: Example output lines from aligned reads

```
1888 ^@
64 ^@
71 K
```
Here we see relative positions and deltas encoded in a Unicode character. The “^@” character refers to an equal read, meaning a perfect match with the reference genome.

### 4.4 Unit Testing

Before we started work on the rewrite, a unit testing framework was set up. Given the sizes of data involved it takes a very long time to check new code against a full set of reads, and the nature of implementation for unaligned reads means that we need the entire set to see the results that we want. This means that to be able to regularly check the effects of new code we needed unit tests for every component.

The decoupled design makes it easy to test every class without being too dependent on other parts of the program. We use mockito [22] to mock responses from the dependencies that we don’t want to fully instantiate. We see an example test in listing 4.7 for a class that generates shift operations. Here we have mocked a dependency and manually computed the set of shifts that should be generated.

Listing 4.7: Example of a test for a close match generation class.

```python
class TestDoubleShiftSet(unittest.TestCase):
    def setUp(self):
        self.read = "ACTG"
        stor_mock = mock(Storage)
        self.shift_set = DoubleShiftSet(self.read,
                                         Alphabet("ACTG"), stor_mock)

    def test_generate_operations(self):
        shifts = ("TGAA", "TGAC", "TGAG", "TGAT",
                  "TGCA", "TGCC", "TGCG", "TGCT", "TGGA",
                  "TGGC", "TGGT", "TGGG", "TGTA", "TGTC",
                  "TGTT", "TGTG", "AAAC", "ACAC", "ATAC",
                  "AGAC", "CAAC", "CCAC", "CTAC", "CGAC",
```
"TAAC", "TCAC", "TTAC", "TGAC", "GAAC",
"GCAC", "GTAC", "GGAC")
self.shift_set.generate_operations(self.read)
self.assertEqual(len(shifts),
    len(self.shift_set))
self.shift_set.debug_print()
for s in shifts:
    self.assertTrue(self.shift_set.contains_read(s))
Chapter 5

Results

In this chapter we present the results of several runs of processing on short read data sets with different parameters. The results are presented in two parts; runs that are pure compression on read sets without a reference genome in 5.1 and runs with a reference genome in 5.2.

For each run we present an overview table of the size impact of the processing. First we list the size of the original read data and the size after our processing alone. Then we give the sizes of a standard run with gzip on the original data and the data from our script. For easy comprehension the compression ratios are listed at the bottom.

We also present some charts to help us understand the result and motivate reasoning of the performance in the discussion chapter 6.

5.1 Compression of independent short read data set

Since read length is expected to be a deciding factor for the performance of our program, we present here runs on 3 different read length data sets. Each run is from the Illumina platform. For each data set the reads are extracted from the machine output and run through our program. No training or preconceived knowledge of the specific platform is used in the form of tweaking the edit operations used. The standard operations Replace, Transpose (Swap), SingleShift and Doubleshift are used for all the reads.
5.1.1 Illumina run with a read length of 36 base pairs

<table>
<thead>
<tr>
<th>Compression results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Read length</td>
</tr>
<tr>
<td>Machine</td>
</tr>
<tr>
<td>Size</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Time</td>
</tr>
<tr>
<td>Compression rate</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Table 5.1: 36 bp Independent results

To begin with, we present our base case, an Illumina run with 36 base pairs per read with high accuracy.

To get an overlook on the general outcome we look at the table with raw size results. After we have stripped away meta-data the read set is about 500 Mb.

After running our program it comes out in 81 Mb, about 17% of the original size, compared to a plain gzip of the original which comes down to 126 Mb. When we finally gzip the output from the script, the final size is 22 Mb, 4.5% of the original size and 17% of the size with gzip alone. For the prototype Python speed of 40 minutes, this compression looks promising.

Even with this relatively small data set we are able to store only 13% of the reads as complete key frames, with the rest of the set, 87%, being stored as difference encoded operations in one Unicode character.

We inspect the distribution of operations that the reads are getting encoded as in figure 5.1. We immediately notice the fact that 60% of the reads are perfect repeats of their key frame.

After that comes reads that either has small errors or are repeats with small alterations. Shifts also accounts for a large amount of reads at 14%, but surprisingly transpositions are almost non-existant. This is probably an indicator of the quality of the reads. One would guess that transpositions would be most apparent with variation experiments or comparative genomics.
Figure 5.1: The spread of operations on compression of a single Illumina read set with 36 base pairs

Figure 5.2: The number of hits each key frame read found in its operations on a read set with 36 base pairs
We see in the distribution of hits in figure 5.2 that most key frames get 0-5 hits, but what really ends up giving us a substantial compression is that some key frames find an enormous amount of candidates to compress into a character. In this set the key frame with the most hits got 168289 close matching reads. We look more closely at this effect in 5.1.3.

It is evident that this first run is what we had in mind when first designing the processing approach. We see a nice 6:1 size reduction by the script itself, leaving the read in a usable state, but with a substantially smaller size.

We see that over half of the processing work is simple cataloguing of repeat reads. Doing this is still relevant because of the meta-data referencing involved. Because of this high number of equal reads, it would benefit us to do run length encoding as mentioned in 2.2.1 ourselves, keeping the accessibility of our format and decreasing the size more effectively before zip. The amount of equal reads will probably fluctuate between experiments, depending on the uniqueness factor of the genome being sequenced. TODORef: Uniqueness.

A quarter of the reads are 1 replace operation away from the key frame.

At this length our approach works very well because the closeness of the reads in our edit distance metric. At this length shifting plays a minor role, as most reads can just be matched by replace and being pure equals.

### 5.1.2 Illumina run with read length of 50 base pairs

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read length</td>
<td>bp</td>
<td>50</td>
</tr>
<tr>
<td>Machine Platform</td>
<td>Illumina</td>
<td></td>
</tr>
<tr>
<td>Size</td>
<td>Original (Mb)</td>
<td>810</td>
</tr>
<tr>
<td></td>
<td>after script</td>
<td>377</td>
</tr>
<tr>
<td></td>
<td>gzip original</td>
<td>252</td>
</tr>
<tr>
<td></td>
<td>script &amp; gzip</td>
<td>119</td>
</tr>
<tr>
<td>Time</td>
<td>minutes</td>
<td>62</td>
</tr>
<tr>
<td>Compression rate</td>
<td>script to original</td>
<td>2.1</td>
</tr>
<tr>
<td></td>
<td>gzip script to gzip</td>
<td>2.1</td>
</tr>
</tbody>
</table>

Table 5.2: 50bp Independent results
Figure 5.3: Operations used in compressing a Illumina set with longer 50 bp reads

With the read length upped to 50bp we see a substantial hit to our compression rate. In figure 5.4 we see that we are not able to get close to that same low amount of key frames as with the shorter reads. Figure 5.3 shows a decline of repeats and reads within 1 replace distance of each other. The amount of shift operations are mostly unchanged from the shorter reads, and get a bigger slice given the decline of the other operations.

With just 14 base pairs longer reads we see a giant increase in the need of key frames to match all the reads. This increase is enough to make pure equals much more unlikely, and the same thing occurs with being able to replace just one nucleotide across so large reads.

We also have to consider that with 50 base pairs the program is computing 289 candidates for each read, and thus have an encoding table that is sized accordingly. This means we have 161 operations that now require two bytes in the encoding, since UTF-8 encodes 128 characters in one byte, substantially increasing the size of some of the compressed reads.

Shifting is now takes over to represent over 40% of the compressed reads. This has more to do with the rate of shifts being constant across all the read lengths, than shifting becoming more effective. As shifting is able to find
Figure 5.4: Distribution of uncompressed key frames and frames represented as deltas with 50 bp reads
partial overlaps it should be independent of the length of the reads.

5.1.3 Illumina run with read length of 75 base pairs

<table>
<thead>
<tr>
<th>Compression results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Read length</td>
</tr>
<tr>
<td>Machine</td>
</tr>
<tr>
<td>Size</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Time</td>
</tr>
<tr>
<td>Compression rate</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Table 5.3: 75bp Independent results

We see here a continuing decline of effectiveness from the shorter read sets. In figure 5.6 we see that the uncompressed key frames now outnumber the compressed frames, making 2:1 compression impossible. In figure 5.5 we see the distribution of operations used.

Also note the huge increase in running time for this larger read set. This comes mostly from the fact that we are now generating 414 close matches for each key frame, from the computation in 4.1.1 We are also doing this computation for a much larger set of key frames, since less matches are found overall.

In figure 5.7 we see how sequentially the earlier reads processed into key frames vacuums for close matches across the genome, finding large amounts of close matches. This effect mostly dies off after the 100k first reads. For the read sets with shorter reads the effect is even stronger and lasts for more key frames.

We see that the time used to compute the encode has increased over as much as 7 times over the 50 base pair read set. As noted with 75 bp we are computing over 400 candidates and doing it for a large amount of frames.

Encoding continues to decline, both because we are matching less candidates and need more key frames, and because the compressed frames are
Figure 5.5: Operations used in compressing a Illumina set with 75 bp reads
Figure 5.6: Distribution of uncompressed key frames and frames represented as deltas with 75 bp reads
Figure 5.7: Shows the amount of compressed close matching frames found for the 150k first key frames
becoming bigger, with 286 operations now using two unicode bytes in the encoding.

5.2 Compression of aligned reads with reference genome

For the runs in this section all data sets used the HG19 genome as a reference lookup. All runs are of chromosome 1 from the experiment only.

5.2.1 Illumina with 36 base pairs

<table>
<thead>
<tr>
<th>Compression results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Read length</td>
</tr>
<tr>
<td>Machine</td>
</tr>
<tr>
<td>Size</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Time</td>
</tr>
<tr>
<td>Compression rate</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: 36 bp Aligned results

With this small data set we see that we are able to match the aligned reads to the reference genome pretty well with our 1 step distance metric. In figure 5.8 we see that 3/4 of the alignments are complete matches, meaning that this sample was very close to the reference with little variation. Even though this may be a best case scenario it shows the promise that lies in this technique of using a reference library.

Figure 5.9 shows that 12% was uncompressed from its initial state, leaving the 88% that did compress to take up 7% of the size. This shows that a general differencing of all the reads, without limiting edit distance, would be able to compress down to under 10% in some cases.
Figure 5.8: Operations used in compressing a Illumina 36bp set with reference

Figure 5.9: Distribution of compressed matches and uncompressed reads on 36bp Illumina set with reference
5.2.2 Illumina with 76 and 67 base pairs

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read length</td>
<td>bp</td>
<td>76 &amp; 67</td>
</tr>
<tr>
<td>Machine</td>
<td>Platform</td>
<td>Illumina</td>
</tr>
<tr>
<td>Size</td>
<td>Original (Mb)</td>
<td>172</td>
</tr>
<tr>
<td></td>
<td>after script</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>gzip original</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>script &amp; gzip</td>
<td>6.5</td>
</tr>
<tr>
<td>Time</td>
<td>minutes</td>
<td>71</td>
</tr>
<tr>
<td>Compression rate</td>
<td>script to original</td>
<td>7.1</td>
</tr>
<tr>
<td></td>
<td>gzip script to gzip</td>
<td>4.1</td>
</tr>
</tbody>
</table>

Table 5.5: 76/67 bp Aligned results

We see that the compression rate actually increases with longer reads. This is simply because the longer read length still is compressed down into our 1 or 2 byte difference encoding, leaving the ratio between these two sizes to be increased when the uncompressed read is longer. Still, in this case as well, a general string-to-string difference algorithm would better suited. As we see in figure 5.11 we are largely unaffected by the 1 edit distance limitation because the number of reads that is further away is limited.
Figure 5.10: The spread of operations on compression of a set with HG19 as a reference
Figure 5.11: Damerau-Levenshtein distance from the dataset to the reference genome
Chapter 6

Discussion

6.1 Analysis of results

To investigate how and why we get the results as covered in 5 we will look at how the program would perform under some hypothetical constraints. These constraints will explain the performance on the real data by gradually adding parameters to the read sets. We will look at how repeating elements, variation and sequencing errors affects our program.

Perfect genome, perfect reads

First we consider the case of a perfect genome. A genome with no long repeated sequences and no variation in the sample being sequenced. Depending on coverage the reads from this genome will have a high degree of uniqueness and there will be no matching across the genome for unaligned read processing, due to there being no repeated sections. This means there will be no replace or transposition operations. The only reads matched will be perfect equals when the same section has been sequenced with perfect overlap, and shifts for when the overlap is shorter.

For aligned reads the entire process will return only compressed equal operations, since every read will align without error.
Repeating elements

We add repeating elements to our perfect genomes. Now we see the introduction of the effect in figure 5.7. The first reads selected as key frames will match repeats across the genome and combine large sections of reads as close matches.

Repeating elements changes little for the aligned read process, as the reads are still perfectly sequenced and matched they are still equal.

Variation

Many experiments deals with individual variations over the sequenced genome. For unaligned reads this means little, it will match in the same way as the previous section.

For aligned reads it is clear from the implementation that variation has a big impact on the encoding. Since we have decided to use the design of operations from the other use case, extending it beyond 1 edit distance is impractical. Considering this we will miss every read that has been aligned with a longer edit distance than 1.

Quality

We decrease the quality of the sequencing platform and introduce sequencing errors. Even though the technology is moving fast, there is no such thing as a perfect genome sequence. We set the error rate to a lower bound 1 in 100,000 base pairs, which means that even for smaller experiments we will have hundreds of errors.

Introducing errors hurts the compression of unaligned reads. Repeating sections that were able to be encoded as shifts or equals are now thrown off due to extra edit distance due to error. Replace are able to encode the reads that are equals if there are only one error, but not the shifts.

For aligned reads educed quality hurts aligned reads in the same way variation does, it becomes unable to match wrongly sequenced sections to the reference genome.
6.1.1 Compression of unaligned vs aligned reads

Obviously the case of aligned reads has the potential for greater compression simply because of how much of the information is able to be stored in the reference library. Still we see that unmapped reads are able to be compressed fairly well when their length is limited, with the length having the inverse effect on the compression rate from what it has on aligned reads. This is important since unmapped read often make up between 10% to 40% of the data in experiments [10].

6.2 Design choices

When starting the design process there was only one case that was looked at; unprocessed unaligned reads from the Illumina platform with 36bp length. This is the most prevalent experiment output in our research environment and was therefore a natural starting point.

In this data uniqueness is lower simply because of the shorter read length and the reads have longer common substrings. This data intrinsically has compression steps that can be applied and achieve good compression. It was not immediately apparent to which degree this would degrade with increasing read length.

Because of this single edit distance exhaustion is not ideal for creating difference coding for aligned reads. As the read lengths increase the edit distance of the differencing needs to scale as well.

6.3 Implementation issues

The case of aligned reads not only needs a separate design, but within our design that we used, it could use a whole different implementation. For speed of development it was added as a different read container class and associated methods for processing. Since the unaligned reads already had a tried and tested way of generating close matches to search for, the positional aligned reads used the same methods. This is awkward since when we do it for unaligned reads we are usually getting hits on a lot on those generated reads, but when using on aligned reads there is one, and only one, possible match that we are looking for. Also when generating a large set of reads
for unaligned sets, we are eliminating the need to run the same process for those reads, so for every hit we are removing work to be done. For aligned reads the generation has to run for every read regardless, which makes its effectiveness more important.

It seems clear that a more general minimal string difference algorithm would be more beneficial to implement in this case. This could get a difference encoding no matter the quality of the reads or the variation, even though that would effect the compression rate also in this case.

6.4 Related work

6.4.1 General DNA compression

Adjustment of general Lempel-Ziv to compression of biological data has been of interest since the early 90’s [3] [4] [9]. However in these papers the focus is on a complete genome sequence and compression of it, which is natural since at the time slow sanger sequencing with low throughput was the norm. The need for read compression wasn’t there since the amount of data was low.

The motivation for research has varied, as in this paper where statistical analysis of the genome were in focus:

The primary purpose of the work is not to compress strings, and in particular DNA strings, so as to save computer storage space or to reduce data transmission costs. Rather, the purpose is to model the statistical properties of the data as accurately as possible and to find patterns and structure within them.

(Excerpt from Compression of Strings with Approximate Repeats [3])

In this way, our approach can perhaps more be viewed as processing of reads rather than a Lempel-Ziv adaption.

6.4.2 Read processing

There exists some work on preprocessing reads for GZIP compression [24]. This consists of only doing a sorting run on the reads, thus making traditional GZIP much more effective on the data. This is due to the sliding window size as mentioned in 2.2.1. Sorting the reads enables LZ to always have the right
data in its compression window, negating the effect of having overlapping reads spread across the data and escaping compression. Important to note is their end result of achieving up to 12% reduction in size.

### 6.4.3 Recent advancements

In May 2011 an interesting article by Fritz et.al. appeared in Genome research [10], called Efficient storage of high throughput DNA sequencing data using reference-based compression. This paper applies some of the same approaches as us, but in a more refined way. For aligned reads the idea is basically the same. Compare the read to a reference genome and save the difference, but they do the differencing and compression with minimal size result independently of any external processing like GZIP. They do this using Golomb codes for the positions and operations, which enables them to have no cap on the edit distance for the encoding. This means that they efficiently encodes every mapped read in the set. In addition to this they also provide the ability to store the quality scores in an adjustable degree.

When it comes to unmapped reads they suggests methods to compress about 20% of those reads. For the rest of the data they write

> [..] this left a considerable number of reads (83%) that cannot be easily compressed. [..] Furthermore, the reads do not show evidence, repeated kmer frequencies; 57% of 21 mers were unique in the data set and <1% showed >10 frequencies. This strongly suggests that there is not a systematic source of sequence at high coverage in the remaining unaligned sequences.  

(Excerpt from [10])

It is possible that the reads in their example has a high uniqueness and is simply not able to be compressed by any substantial rate with differencing between the reads, but it would be interesting to try. Even with low coverage close matching reads should be able to compress some if the ceiling for edit distance was increased. We hypothesize a way we could combine the techniques in 7.4.

### 6.5 Usefulness

As said in the quote in 3 by Skiena, compression tailored to specific data sets is often wasted effort. He says do simple preprocessing and otherwise stay
out of it. The question of if we crossed the border of “simple” in this thesis is hard to answer. We certainly spent probably more time both in work and CPU processing hours than what “simple” would imply.

6.5.1 Time versus space

There are many problems and algorithms in computer science where we are given the option to trade time for space and vice versa. With the new sequencing technologies we are almost bound to try, just from the immensity of the information that is interesting to store. We have deliberately tiptoed around the question of exactly how much time we are trading in for the space in this thesis. We feel that the issue of time spent is only interesting if the algorithm is sufficiently optimized, and optimizing a python prototype went out of the scope of this thesis.

6.5.2 Future in Sequencing

As mentioned in the background about future/state of the art sequencing platforms, they are getting better and better at sequencing longer reads [19]. Even though the specifics in this thesis works optimally on shorter reads, the general approach of difference encoding reads still holds some promise for even longer reads. The reads will still have very long common substrings, but the margin of error needs to be far longer than the 1 edit distance used for simplicity in this assignment. This also means that the rather simple range encoding of operations into one Unicode character will not suffice, and a more general diff algorithm will be needed. We feel that the simple difference method shows that difference encoding of reads is an idea that shows some merit.

6.5.3 Workstation compression

The compression times reported in the result section in 5 are times that are achieved running on a super computer (Invitro) at UiO. With storage generally getting cheaper and cheaper, the general benefit of compressing the raw reads from the sequencing platforms would seem to be just as much the transferring of data before any alignment has been done. Thus it would be most beneficial to be done on a local workstation instead of transferring the complete set up to some powerful server for compression.
The fact is that the highly un-optimized program as it runs in the results is the slowest possible way for it to run, even if it were switched to a regular workstation. It runs unparallel, only utilizing a single core on the cluster, and using less memory than is usually available on a modern workstation. If memory becomes a problem the dataset can be split into smaller pieces. The size of the sets in the results section is fairly small, and shows that it compresses fairly well even with those small number of reads. With more memory and more reads it compresses even better.
Chapter 7

Future work

7.1 Encoding improvements

As it is now we leave long sequences of equal operations in their entirety after the key frame, and leave GZIP to optimize this redundancy away. We could easily use run length encoding on this ourselves.

Right now all the different operations get their own byte without consideration to the probability of that operation. Some operations are vastly more likely than others. We could Huffman encode our own internal structure, making sure that at least the most used operations were packed into the end of the table that was 1-byte encoded.

As mentioned in section 6.2, the rather trivial method of range encoding used in this thesis has limited effect on data sets with increasing read length. The next step would be to generalize the difference encoding scheme, so as to remove the cap of 1 edit for close matching reads.

As for now, without removing the cap, for compression with reference genome, when not finding a match because the edit distance is greater than 1, we could run a all those candidates in a new generation. This would require a lot of time since this is a massive search space, but would guarantee almost full coverage.
7.2 Machine profiling

Since the machines have different error footprints, it would be better to have the platform as an input to the program, rather than using the same algorithm for all the platforms. By algorithm here we mean just the edit distance measuring metric. It would seem advantageous if we could do a large amount of runs on the data from different platforms. We could then profile this data, investigating the probability distribution of the different operations, and then skew that into new operations and profile again. E.g. one machine seems to have a large amount of shifted data, we could then increase the number of shift operations possible, and maybe get even more hits in the next run. So in reality in a finished program an input switch would tell it which edit distance metric to load, and populate the encoding table accordingly. Ideally this feature would be made completely dynamic, so that you can start profiling from scratch, without the program being aware of the platform type at all.

This can be relatively trivial, with profiling and edit distance metric data stored in configuration files. Since the encoding table would depend on these parameters the encoded file would need to reference the metric used as well.

A future thesis assignment would perhaps be to do this training on machines to have a static input statistics, thus beating the LZW methods that are dynamically guessing.

7.3 Parallelization

In the case of compressing a set of reads with a known reference genome available, it is easy to see that this is parallelizable. Here the compression of a read is completely separated from the computing of the encoding for the rest of the reads. It is simply a case of finding a minimal string difference for a large set of reads. We could use a normal parallel library to spawn a queue of these jobs and spread the load over the CPU’s when compressing on a large cluster.
7.4 Possible combination with Fritz et.al

As mentioned in section 6.4.3 the method used in [10] compress the aligned reads very well, however unaligned reads go highly uncompressed. It should be explored if some combination of an uncapped intra-frame compression could be used together with their all-round referenced based compression. If the cap of 1 edit is removed, it is that not some sort of compression ratio should be attainable for the large set they have of unaligned reads, even though the uniqueness of the reads could be high.
Bibliography


