Optimizing Lempel Ziv Welch for DNA Compression

A Thesis

Presented to

The Division of Mathematics and Natural Sciences

Reed College

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Arts

Caden Corontzos

May 2023

Approved for the Division
(Computer Science)

Eitan Frachtenberg

Acknowledgements

I want to thank a few people.

Preface

This is an example of a thesis setup to use the reed thesis document class (for LaTeX) and the R bookdown package, in general.

List of Abbreviations

EOF	End of file
LZW	Lempel Ziv Welch
RLE	Run Length Encoding

Table of Contents

List of Tables

List of Figures

Abstract

The Lempel Ziv Welch compression algorithm is a lossless data compression algorithm used for numerous applications, including the Unix file compression utility `compress` and the GIF image format. Storing, reading, and transferring enormous amounts of data is often an issue in the biological field, especially when concerning DNA. This thesis explores the application of Lempel Ziv Welch to the compression of DNA. A variety of different optimization of the original LZW algorithm are explore included palatalizing, multiple dictionaries, and some other cool thing here broh.

Dedication

You can have a dedication here if you wish.

Introduction

When dealing with DNA, it

Chapter 1

Backround and Motivations

This thesis deals with some high-level topics and uses language specific to compression research. This chapter tries to give brief summaries and examples of the relevant topics to be discussed so readers of all experience levels can put our results into context.

1.1 What is information?

Suppose you had an idea that you wanted to share with another person. Humans have many ways to communicate information; you could send a text message, you could tell them with words, you could tell them with sign language. But regardless of the medium, you have some idea that you want to get across. Does it matter if the other person gets your message exactly? If someone asks you "Where library", despite the lack of prepositions, you still understand what they mean. So did that person convey any less information than a person who asks "Where is the library"? Clearly, information is fundamental to how humans interact and how they understand the world, but defining it proves difficult. For our purposes, lets assumed that infomation is data with some sort of significance that makes it worth perserving and conveying.

Information on computers can take many forms, such as text, audio, and video. This information can travel through many channels including the internet, wires, screens, etc. To maximize the amount of information that can be transmitted through a channel with a limited capacity, we need to encode the information in a way which minimizes its size, while also preserving its essential features. This process is called compression.

1.2 Compression Metrics

1.2.1 Compression Ratio

Compression Ratio is the measure of size reduction achieved by a compression algorithm. It is typically expressed as a ratio of the size of the uncompressed data's original size (OS) to the size of the compressed size (CS).

$$CR = \frac{OS}{CS}$$

So a higher compression ratio means a more effective compression algorithm, and means that we were able to store more information in less space, allowing for easier storage and transfer.

1.2.2 Run Time

The run time is also an important part of evaluating the effectiveness of a compression algorithm. Run time is typically defined as the length of time a program takes to complete a task. Sometimes, if time is constrained, you may care less about saving space. For example, if you have the option of two compression algorithms, one with a compression ratio of 2.0, and another with a compression ratio of 2.15 but takes twice as long as the other, you may opt for a lower compression ratio to save time.

1.2.3 Memory Usage

Memory usage is closely tied with runtime when it comes to compression algorithms. Memory generally refers to storage where information that programs track as they are running is stored. So to reduce our run time and make a more effective compression algorithm, we want to be saving only the most important data that our algorithm needs in order to reduce our memory usage. Throughout this thesis, we will assume that most of memory usage is encapsulated in our measurement of run time.

1.3 Lossless vs. Lossy Compression

1.3.1 Lossy

Lossy compression is based on the idea that not all information is vital. For instance, when saving a picture on your computer, your computer may save it in the .jpeg

format to save space. Jpegs lose some of the information in the original picture and produce an overall lower quality picture, but the general information in the picture is preserved. Another example is MP3 audio files. MP3 compression discards some of the information and sound quality in exchange for a file that takes up less space, which is often favorable for small devices like MP3 players and cellphones.

1.3.2 Lossless

Lossless compression is the compression of data with the goal of preserving all the information in the data so that it can be reproduced perfectly on decompression. As a result, lossless compression algorithms usually don't compress as well as their lossy counterparts. Lossless algorithms are important for use cases in which data needs to be wholly recovered, like scientific data, archiving (e.g a .zip folder), and high end audio recording. Examples of lossless compression algorithms are Huffman Encoding and Lempel-Ziv-Welch, which is the focus of this thesis.

1.4 Classic Compression Algorithms

1.4.1 Run Length Encoding

Run Length Encoding (RLE) is one of the simplest and most intuitive forms of compression. We can take advantage of redundant runs of characters in a sequence by just giving the number of times each character appears. Suppose you want to send the following message

$$AAGCTTTTTTTTGGGGGCCCT$$

Even if this message did mean something, we can get the information across without repeating ourselves. When writing a grocery list, you don't write "egg egg egg egg", you say "4 eggs". RLE uses this same strategy.

$$2A1G1C8T5G3C1T$$

We could compress this even further if we omit the 1 on characters that only appear once. Although not as sophisticated as other methods, RLE is effective when used on texts that have a lot of repeating characters.

1.4.2 Huffman

Huffman Encoding is a strategy that assigns variable length code to certain symbols in the data. The goal is to assign short codes to frequently appearing symbols and longer codes to less frequent symbols.

Suppose we have a message "ACAGGATGGC". We can calculate the frequency of each letter by counting the number of times each letter shows up and dividing by the total number of letters

Then, we can use the frequencies to build a tree, which will assign short codes for frequent letters and longer code for less frequent letters. The more frequent character occur higher up in the tree, giving them a shorter length. The less frequent characters occur farther down on the tree. If you follow the branches in Figure **??** down to a letter, it will tell you the code associated with that letter. So $G = 0$, $A = 10$, $T = 111$

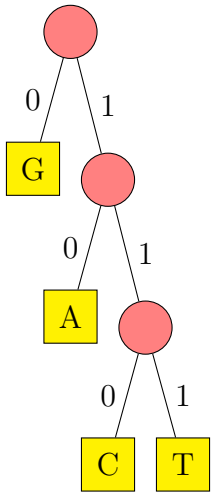

Figure 1.1: Example Huffman tree

and $C = 110$. Notice that none of the encodings are prefixes of one another, which makes it unambiguous in decoding.

So our message would be encoded to 10110100010111100110.

1.4.3 Arithmetic

Arithmetic encoding is another lossless compression algorithm that uses probability to assign codes to symbols in the message. Unlike Huffman, arithmetic encoding assigns a single code to the whole message, rather than separate codes for each symbol.

Here is a simple example. Say we want to encode a string of characters "ACGT". Arithmetic Encoding also requires the encoder and decoder know the probabilities of

each of the characters that could possibly be in the message. Let's say the probability of each symbol in the message are

- P(A) = 1/10
- P(C) = 2/10
- P(G) = 4/10
- P(T) = 3/10

We want to represent the message as a fractional number between 0 and 1. We will divide the interval [0,1] into sub intervals using the probabilities of each character in the message. That way, each symbol is represented by the sub-interval that corresponds to its probability.

Since 'A' comes first, we divide [0,1] into [0.0,0.1). Since 'C' is next, we go from [0.0,0.1] to [0.01, 0.03). Then since 'G is next, we go from [0.01, 0.03) to [0.016, 0.02). Finally, since 'T' is last, we go from [0.016, 0.02) to [0.0188, 0.02).

So any number in the interval can be used to represent our message.

Arithmetic encoding can have a better compression ratio that Huffman in some cases, but the computation time is often not worth the payoff.

TODO: Add source

1.4.4 Lempel-Ziv-Welch

Lempel-Ziv-Welch is another lossless compression algorithm. When compressing, LZW builds a dictionary of codewords, where codewords represent strings previously seen in the message. As it compresses the message, the dictionary grows. The compression algorithm leaves behind the codewords and some of the original characters, allowing the decompression algorithm to build up the same dictionary as it decompresses the message.

Here is a simple example. We may be sending messages with the characters 'A', 'C', 'T', 'G', so we can start by assigning those strings codewords. So our dictionary will start as {'A' = 0, 'C' = 1, 'T' = 2, 'G' = 3}. Say we want to send the message

AAGGAATCC

When we compress, we start at the beginning of the message and scan through. We ask ourselves, "Is"A" in our dictionary?"

AAGGAATCC

Step	Input String	Dictionary State	Encoded String
1	**A**AGGAATCC	A: 0, G: 1, T: 2, C: 3	-
2	**AA**GGAATCC	A: 0, G: 1, T: 2, C: 3	0A
3	AA**G**GAATCC	A: 0, G: 1, T: 2, C: 3, AA: 4	0A
4	AA**GG**AATCC	A: 0, G: 1, T: 2, C: 3, AA: 4	0A1G
5	AAGG**A**ATCC	A: 0, G: 1, T: 2, C: 3, AA: 4, GG: 5	0A1G
6	AAGG**AA**TCC	A: 0, G: 1, T: 2, C: 3, AA: 4, GG: 5, GA: 6	0A1G
7	AAGG**AAT**CC	A: 0, G: 1, T: 2, C: 3, AA: 4, GG: 5, GA: 6	0A1G4T
8	AAGGAAT**C**C	A: 0, G: 1, T: 2, C: 3, AA: 4, GG: 5, GA: 6, AAT: 7	0A1G4T
9	AAGGAAT**CC**	A: 0, G: 1, T: 2, C: 3, AA: 4, GG: 5, GA: 6, AAT: 7	0A1G4T3C
10	AAGGAATCC	A: 0, G: 1, T: 2, C: 3, AA: 4, GG: 5, GA: 6, AAT: 7, CC: 8	0A1G4T3C

Table 1.1: An example of LZW ran on the input "AAGGAATCC"

Since we started with “A” in our dictionary, we can move on. We then add on the next character in the sequence and ask “Is”AA” in our dictionary?”

AAGGAATCC

We have not seen “AA” before, so we should add it to our dictionary. So now our dictionary looks like this {‘A’ = 0, ‘C’ = 1, ‘T’ = 2, ‘G’ = 3, ‘AA’ = 4}. Next time we see “AA”, we know it is associated with the codeword 4. To indicate this, in our resulting string we will output the code for “A”, the part of the string we’ve seen before, and the character “A”.

So the encoded string will look something like

0A....

When we are decoding, we start with the same dictionary. We see “0A” and know that that means “Take the string that has codeword 0, add on the character ‘A’ and add that new string to the dictionary. We would add that to the dictionary and assign it to our next available codeword, 4. As you can see, while decoding, we are able to build up the same dictionary as was used for encoding, as long as we use the same starting dictionary.

This method has several convenient properties:

- When we send this encoding to someone else, we don't need to send a "codebook" (our dictionary). They are able to build it up themselves as they decode.
- you only need to go over the data once to encode and decode. This means that the run time of the algorithm should roughly increase linearly with the length of the input.
- As runs get longer, we will start to see more and more repeating patterns, and replacing them with codewords will become more and more effective.

To decode, we can simply start with the same dictionary. We see codewords followed by one character, so we decode that codeword and add the character to the end. Since the decoder continues to look for codewords, we need some special character at the end of our encoding to let the decoder know when the message is done.

Now that we have laid out how the algorithm works, we can get more specific for our use case. For us, the input is a file on the computer, and the output is also a file (hopefully a smaller one). We read all the characters in the file, encode them, and put them into a new file. Then, when we want to decode, we read the encoded file and output the decoded characters. Again, LZW is lossless, so the original file and the decoded file should be identical.

Here is some example pseudo code on what this algorithm would look like.

```
LZWEncode(input):

    Dictionary dictionary; // where we store our string => codeword mappings
    dictionary.inititalize; // initialzie the dictionary with single characters

    codeword; // the unique numbers we assign strings
    output; // where we output the encoded characters

    currentBlock = first character of input;
    for every nextCharacter in the input:

        // this function returns the longest string we've already seen
        // starting at our current place in the input
        nextLongestRun = findLongestInDict();

        if currentCharacter + nextLongestRun.length > input.length:
            break;
```

```
        // output the code of the next longest run and next character
        code = dictionary.lookup(nextLongestRun);
        nextCharacter = input[nextLongestRun + 1]
        output(code);
        output(nextCharacter);


        dictionary.add(currentBlock + nextCharacter, map it to codeword);

        codeword = codeword + 1;
        input = input + nextLongestRun + 1;


    output(special end of file character);
```

The decoding is much simpler. The only real difference is that we are now mapping codewords to strings, since the encoded string contains codewords.

```
LZWDecode(input):

    Dictionary dictionary; // where we store our codeword => string mappings
    dictionary.inititalize; // initialzie the dictionary with single characters

    codeword; // the unique numbers we assign strings
    result; // where we output the encoded characters

    while we don't see the end of file character:

        codewordFound = input.readCodeword()
        nextCharacter = input.readCharacter()

        sequence = dictionary.lookup(codewordFound) + nextCharacter
        result.output(sequence)

        dictionary.add(sequence = codeword)
        codeword = codeword + 1;
```

This is the basic strategy we will start with for our LZW algorithm. In the next chapter, we will go over parts of the algorithm in depth in C++. Here is a quick summary of terms repeated throughout the next two chapters.

- `dictionary`: a key-value system. Like a real dictionary holds words and their corresponding definitions, our dictionary holds codewords and their corresponding strings of characters. In C++, this is called a `map` and uses a hash table, but the concept is the same.
- `codeword`: a number used to take the place of a string in our encoding.
- `run`: the next run of characters in our input that are already in our dictionary. So if we are encoding "ACTG", and "A", "AC", and "ACT" are in the dictionary but "ACTG" is not, we have a run of 3.
- `EOF`: end of file, the special character that we need to output at the end of the encoding.

1.5 Related work

The idea of compressing DNA is not novel, nor is the idea of using LZW for this purpose. DNA compression is a significant research area, in the intersection of bioinformatics, computer science, and mathematics.

There has been several attempts to optimize LZW by computer science researchers. One paper made use of multiple indexed dictionaries in order to speed up the compression process (Keerthy, 2019). The concept is simple: rather than a single large dictionary, have multiple dictionaries, one for each possible string length That way, the dictionaries grow more slowly and accesses are faster. This paper also used Genomic data to gather their metrics and compared their algorithm to other popular DNA compression techniques, which makes it particularly relevant for this thesis.

Another paper used simple parallelization techniques to improve compression speed (Pani, Mishra, & Mishra, 2012). Rather than compressing the whole file linearly, the researches broke the file into portions and compressed them with LZW in parallel, which greatly increased the compression speed at the cost of a reduced compression ratio.

Yet another paper made use of Chinese Remainder Theorem to augment Lempel-Ziv-Welch (Ibrahim & Gbolagade, 2020). They saw great reduction in compression time without compromising compression ratio, although these results could not be verified. The details of their implementation were not clear from the paper. We tried multiple different methods of utilizing CRT given the pseudocode in their paper, but

we could not get anything that looked like it may improve compression time. We reached out to the authors of the paper, but we were not able to further our progress on this method and thus the it is not used in this thesis.

DNA-specific compression algorithms have also been around for a while. These papers do not focus on LZW, but they do consider some similar methods.

One of the first papers exploring this was published in 1994 (Grumbach & Tahi, 1994). It proposes an algorithm called `biocompress2`, expanding on a previous paper by the same author. They focus on encoding palindromes in DNA sequences, which doesn't do much to help the compression ratio. However, this paper has been cited by many following papers sparking interest in DNA compression, and the collection of sequences that it uses for algorithm comparison is used in this thesis.

Chen et al. proposed an algorithm called GenCompress, which uses approximate matching (Chen, Kwong, & Li, 2001). It matches sequnces to sequences already seen, and maps those sequences using various edits to turn one sequence into another. They are able to achieve a great compression ratio with this method, although their technique is computationally expensive.

In 2007, Minh Cao et al. published a paper detailing another algorithm, XM, which uses statistical methods to try and predict the next character while encoding and decoding (Cao, Dix, Allison, & Mears, 2007). This method was found to outperform both Biocompress2 and GenCompress in terms of compression ratio.

As a whole, these papers give us some guidance in terms of where to aim our research. Most of them boast great compression ratios, but their methods can be very computationally intensive in some cases, and thus, slow. We will aim to use previous research on LZW to make a very fast implementation for DNA sequences, then try and use characteristics of the sequences to improve compression ratio. Our hope isn't necessarily to create the best compression ratio out of all these methods, but to make a fast LZW implmentation with a respectable compression ratio. If we are able to make the algorithm very fast, it may be preferable to these other algorithms in some cases if the file is very large.

Chapter 2

Optimizing LZW: Approach

To restate the goal of this thesis, we seek to optimize LZW for use in compression of DNA. I chose to write in C++. A majority of the work in this thesis involved rewriting, refactoring, and reconfiguring code to improve performance. The various methods we used for this process are discussed throughout the chapter.

While we may not end up creating the best DNA compressor available, the objective is to explore the boundaries of LZW and to tailor it as best we can for the task of DNA compression. As you will see, the algorithm has limitations.

Our Strategy was as follows:

1. Implement a basic version of LZW in C++.
2. Optimize the algorithm. Make it as fast as possible, and specifically focus on compressing DNA
3. Once the algorithm is fast, try to entropy encode (Huffman, arithmetic encoding, etc) the compressed files to improve compression ratio

2.1 Corpora

Most compression papers make use of a corpus, which is a collection of files to run a compression algorithm on in order to evaluate performance and to compare the performance of different algorithms to one another.

In the world of DNA compression, there are several academic papers on the subject. One of the first and most popular of the papers was published in 1994, and the selection of DNA sequences used in the paper have become an informal corpus for the subject of DNA compression, cited by more than thirty publications (Grumbach & Tahi, 1994).

Table 2.1: Corpus 1

Name	bytes
chmpxx	121024
chntxx	155844
hehcmv	229354
humdyst	38770
humghcs	66495
humhbb	73308
humhdab	58864
humprtb	56737
mpomtcg	186609
mtpacga	100314
vaccg	191737

Another, newer paper aimed to create a corpus specifically for compressing DNA (Pratas & Pinho, 2018). They put together a corpus of DNA sequences for this purpose, as summarized below. Since the papers publishing, it has been cited by several DNA compression papers.

Table 2.2: Corpus 2

Name	bytes
AeCa	1591049
AgPh	43970
BuEb	18940
DaRe	62565020
DrMe	32181429
EnIn	26403087
EsCo	4641652
GaGa	148532294
HaHi	3890005
HePy	1667825
HoSa	189752667
OrSa	43262523
PlFa	8986712

Name	bytes
ScPo	10652155
YeMi	73689

The dataset in Table 2.2 is publicly available at this `https://tinyurl.com/DNAcorpus`.

2.2 Evaluating Performance

Evaluating performance of a program is difficult. There is a notion of theoretical run time, but on an actual computer there are many processes running in the background, so it can be hard to get a consistent reading on performance.

To attempt to counteract this, we ran the function on the same file multiple times, and took the median of the compression and decompression times for all the runs. Also, for any graphs or tables in this thesis, all versions of the algorithm for a particular table were run on the same computer. We also did our best to mitigate any other programs running on the computer at the time of data collection to prevent interference.

Most graphs in this section will refer to throughput and average compression time. Throughput is defined as the number of bytes that the program processes per second. The higher the throughput, the more efficient the algorithm. The average compression time is taken as a harmonic average, where the times are weighted by the size of the file.

2.3 A Starting Point

As stated previously, we thought it was best to get a working implementation of LZW in C++ on regular text files, , and then optimize it for DNA. We want to try various techniques tried by researches in the field, but it is important to have a fast baseline from which we can compare and improve upon. If the initial implementation is inefficient, it makes us harder to tell if the different techniques we have are affecting performance.

2.3.1 Growing Codewords and Bit Output

When reading files on the computer, most characters are stored as bytes, which are made up of 8 bits. For instance `01000001` stands for the letter 'A' in ASCII encoding. Numbers in binary are simpler to display, so `00000001` is 1, `00000010` is 2, and so on.

But if we are translating numbers to binary, we don't need all of the bits in a byte. In binary, `1` is the same as `01` is the same as `00000000000001`. So when we are outputting codewords for LZW, we don't necessarily need to output a whole byte. We can have growing codewords.

As the number of codewords grows, the number of bits needed to represent it also grows. So if we are on codeword 8, we need 4 bits since 8 is `1000`. As our dictionary grows, we can grow the number of bits needed to display a codeword and save a lot of space in our compressed document.

So we needed a method of outputting bits one by one, and reading in bits one by one. This is not something that is supported in C++ on its own. We were able to create this functionality by defining a class.

```
// BitInput: Read a single bit at a time from an input stream.
// Before reading any bits, ensure input stream still has valid input
class BitInput {
 public:
  // Construct with an input stream
  BitInput(const char* input);

  BitInput(const BitInput&) = default;
  BitInput(BitInput&&) = default;

  // Read a single bit (or trailing zero)
  // Allowed to crash or throw an exception if past end-of-file.
  bool input_bit();

  int read_n_bits(int n);
}

// BitOutput: Write a single bit at a time to an output stream
// Make sure all bits are written out when exiting scope
class BitOutput {
```

```
public:
  // Construct with an input stream
  BitOutput(std::ostream& os);

  // Flushes out any remaining bits and trailing zeros, if any:
  ~BitOutput();

  BitOutput(const BitOutput&) = default;
  BitOutput(BitOutput&&) = default;

  // Output a single bit (buffered)
  void output_bit(bool bit);

  void output_n_bits(int bits, int n);
}
```

So when we are encoding and need to output a codeword, we can `output_n_bits`, where `n` is the number of bits needed to display our greatest codeword. When decoding, we can just `read_n_bits`.

2.3.2 Getting EOF to work

One of the very early issues with the implementation was how to denote the end of a file. The early implementation would work for some files, but for others the very last part of the file would be lost after encoding and then decoding.

In theoretical implementations of LZW, computer scientists tend to denote the end of a message with a special character, one that isn't seen anywhere else in the file. In this initial implementation, that wasn't possible because we wanted to be able to compress any file with any characters.

The solution was to reserve a codeword to mark the end of the file. So we start with a starting dictionary containing all ASCII characters.

```
    std::unordered_map<std::string, int> dictionary;
    for (int i = 0; i < 256; ++i){
        std::string str1(1, char(i));
        dictionary[str1] = i;
    }
```

As stated in Section 1.4.4, we will need to reserve a codeword to output when we are done encoding the file so that the decoder knows where the end is. So the algorithm goes along reading a file. It builds up a current string character by character, adding the character to the string and checking if it has seen that sequence before. Once it find the end of file, we stop and output the EOF codeword.

The problem was, what about what is left over? Suppose we are reading a file, and the file ends with "ACCT". If "A" is in the dictionary, we see if "AC" is in the dictionary, and so on. This leaves us with three possible cases when we reached the end of the file

1. "ACC" was in the dictionary but "ACCT" was not. This means we can output the codeword for "ACC", follow it by the character "T", and we are done. This is the ideal scenario, because nothing is left over when we output the EOF codeword
2. "ACCT" was in the dictionary: This means we have one more codeword to output, but since we reached the end of the file, we never got to output it.
3. "AC" was in the dictionary, but "ACC" was not: in this case, we would output the codeword for "AC" output the character "C", and then start looping again starting at "T". But we reach the end of the file, so we output EOF before outputting T.

We solved this issue by adding 2 extra bits after the EOF codeword. These bits denote the case that occurred

```
// after we've encoded, we either have
// no current block (case 0)
// we have a current block that is a single character (case 1)
// otherwise we have a current block > 1 byte (default)
switch (currentBlock.length()){
case 0:
    bit_output.output_bit(false);
    bit_output.output_bit(false);
    break;
case 1:
    bit_output.output_bit(false);
    bit_output.output_bit(true);
    bit_output.output_n_bits((int) currentBlock[0], CHAR_BIT);
```

```
    break;
default:
    bit_output.output_bit(true);
    bit_output.output_bit(true);

    int code = dictionary[currentBlock];
    bit_output.output_n_bits(code, codeword_size);
    break;
}
```

So when the decoder is reading and encounters the EOF codeword, it can look at the next two bits to see if anything is left over.

At this point, there was a working implementation that was able to compress and decompress files. Here is the performance of this version on the two corpora.

2.3.3 Using Constants

The early version of the code was not clean. There were hard coded variables, unspecified integer types, and generally messy naming conventions that made the code difficult to read and debug.

The next major step in the code was to start using constants for everything, including

- `STARTING_CODEWORD`: What codeword we should start at
- `EOF_CODEWORD`: What we should output when we reach end of file
- `STARTING_DICT_SIZE`: At this stage, we had a starting dict size of 256 to hold all possible bytes, but later we will specialize for DNA

It also made sense to start using a specific type for codewords. At this stage, we opted for a 32 bit unsigned integer.

There are several tools at a developers disposal when looking to debug and optimize code. One tool used for this thesis was `callgrind` which is a tool of `valgrind` a profiling tool. Profiling tools are used to look at how your code works, where the bottlenecks are, and what can be changed/improved for the performance of your code.

Callgrind in particular is a profiling tool which associates assembly instructions to lines of code, indicating to the programmer which lines take a lot of instructions and which take less. For those unfamiliar, assembly instructions are what code is turned

into so that it can be ran on your computer's processor. In general, more instructions means that code takes longer to run.

The callgrind output drew attention to one particular part of the code. A C++ `unordered_map` uses iterators, basically pointers into the dictionary. If an entry is not present in the dictionary, the `find()` function will return a iterator to the end of the dictionary.

The check for this in our algorithm looked like this.

```
// if we've already seen the sequence, keep going
if (dictionary.find(currentBlock + next_character) != dictionary.end()){
    currentBlock = currentBlock + next_character;
}
```

Here is the callgrind output for that line.

```
105,030,135 ( 0.18%)          if (dictionary.find(currentBlock + next_character) != dictio
13,537,450,317 (22.83%)  => /usr/include/c++/9/bits/basic_string.h:std::__cxx11::basic_str
11,653,779,430 (19.65%)  => /usr/include/c++/9/bits/unordered_map.h:std::unordered_map<std
2,108,383,120 ( 3.56%)  => /usr/include/c++/9/bits/basic_string.h:std::__cxx11::basic_stri
956,941,242 ( 1.61%)  => /usr/include/c++/9/bits/unordered_map.h:std::unordered_map<std::_
241,180,314 ( 0.41%)  => /usr/include/c++/9/bits/hashtable_policy.h:bool std::__detail::op
```

As shown, this line is taking a significant amount of instructions, and it needs to pull the end() of the dictionary each time it is ran. If we use cend() instead and save that iterator in a variable called end, we can save a significant amount of instructions.

```
89,470,115 ( 0.61%)          if (dictionary.find(currentBlock + next_character) != end ){
3,353,009,053 (22.78%)  => /usr/include/c++/9/bits/basic_string.h:std::__cxx11::basic_stri
2,833,786,025 (19.26%)  => /usr/include/c++/9/bits/unordered_map.h:std::unordered_map<std:
420,120,704 ( 2.85%)  => /usr/include/c++/9/bits/basic_string.h:std::__cxx11::basic_string
50,570,065 ( 0.34%)  => /usr/include/c++/9/bits/hashtable_policy.h:bool std::__detail::ope
```

Of course, there are several issues with this method. It is difficult to associate instructions with a single line of code. Some lines are interdependent, and assembly often behaves differently than the code that produces it. Another thing is that compliers are very advanced, and sometimes small optimizations like this are done by the compiler automatically.

Despite these issues, this change was still worth making, if not to save time then for sake of clarity and readability of the code. Also, despite the inaccuracy of callgrind,

like many profiling tools, its job is not necessarily to provide exact measurements of code performance, but to give indications to trouble spots which can be improved.

Here are the runs after this optimization.

2.3.4 Extraneous String Concatenations

The LZW algorithm is built on iteration: we go through each character, adding it to our current block. If we've seen that current block before, we keep going. If not, we add that block to the dictionary and start over.

Another thing that I noticed from the callgrind output was that a lot of time/instructions are being spent on string concatenation. In general, string concatenation in most language, including C++, have a lot of overhead. A lot of implementations involve creating a new string every time you concatenate two existing string, which can have a significant performance penalty.

In the version of the algorithm at the time, every time we have already seen a sequence, we have to concatenate a character. I noticed that I was doing this concatenation multiple times without needing to.

```
// we concatenate the strings here
if (dictionary.find(currentBlock + next_character) != end ){
    // and here
    currentBlock = currentBlock + next_character;
}
else{

    // other code here ommitted



    // and here!
    dictionary[currentBlock + next_character] = codeword;
}
```

If I just save `currentBlock + next_character` into a new variable, that will save me from doing the concatenation 2 more times.

```
// save concatenation here
std::string string_seen_plus_new_char = current_string_seen + next_character;
```

```
if (dictionary.find(string_seen_plus_new_char) != end ){
    current_string_seen = string_seen_plus_new_char;
}
else{

// other code omitted here

    dictionary[string_seen_plus_new_char] = codeword;
}
```

Here are the statistics on the version of the algorithm after this change.

2.3.5 Dictionary Lookups

Dictionary lookups can be expensive, especially with the standard library. We learned from `callgrind` that along with string operations, our program spent a lot of time doing these lookups.

We looked for ways to reduce the volume of lookups. At the time, the way the algorithm worked was that it looked up the current string and the next character in the dictionary. If that string is in the dictionary, we keep going. If not, we output the codeword for the current string.

But, the current string on this iteration is the current string from the last iteration, plus one character. So when we were on the previous iteration of the loop, we could save that lookup and prevent a second lookup.

```
while(next_character != EOF){

    // code omitted


    // if we've already seen the sequence, keep going
    std::string string_seen_plus_new_char = current_string_seen + next_character;
    // save this iterator`
    if (dictionary.find(string_seen_plus_new_char) != not_in_dictionary ){
        current_string_seen = string_seen_plus_new_char;
    }
    else{
```

```
        // shouldn't look up again
        int code = dictionary[current_string_seen];


        // code omitted
    }
    next_character = input.get();
}
```

We can save that lookup, like so.

```
while(next_character != EOF){

    // code ommitted

    // if we've already seen the sequence, keep going
    std::string string_seen_plus_new_char = current_string_seen + next_character;
    codeword_seen_now = dictionary.find(string_seen_plus_new_char);
    if (codeword_seen_now != not_in_dictionary ){
        current_string_seen = string_seen_plus_new_char;
        codeword_seen_previously = codeword_seen_now; // save codeword here
    }
    else{

        // on the next iteration, we use it here
        int code = codeword_seen_previously->second;

        // code omitted

    }
    next_character = input.get();
}
```

2.3.6 Using Const Char *

The algorithm works by reading through the entire file, so we know that at some point, we will need to see every byte of the entire file.

When reading a byte stream of the file, the file may not be in memory the way we want it. The `ifstream` class in C++ also has many extraneous feature that we don't need. If we map the file directly into memory using `mmap` and pass around a pointer to that data, it will simplify and speed up the scanning process. Also, using a `char*` opens the possibility to getting rid of `std::string` entirely, which means way less overhead and decreased compression time.

2.3.7 Comparison

Taking metrics of the algorithm at each of the stages listed in this chapter, we can make a graph showing the improvements in performance.

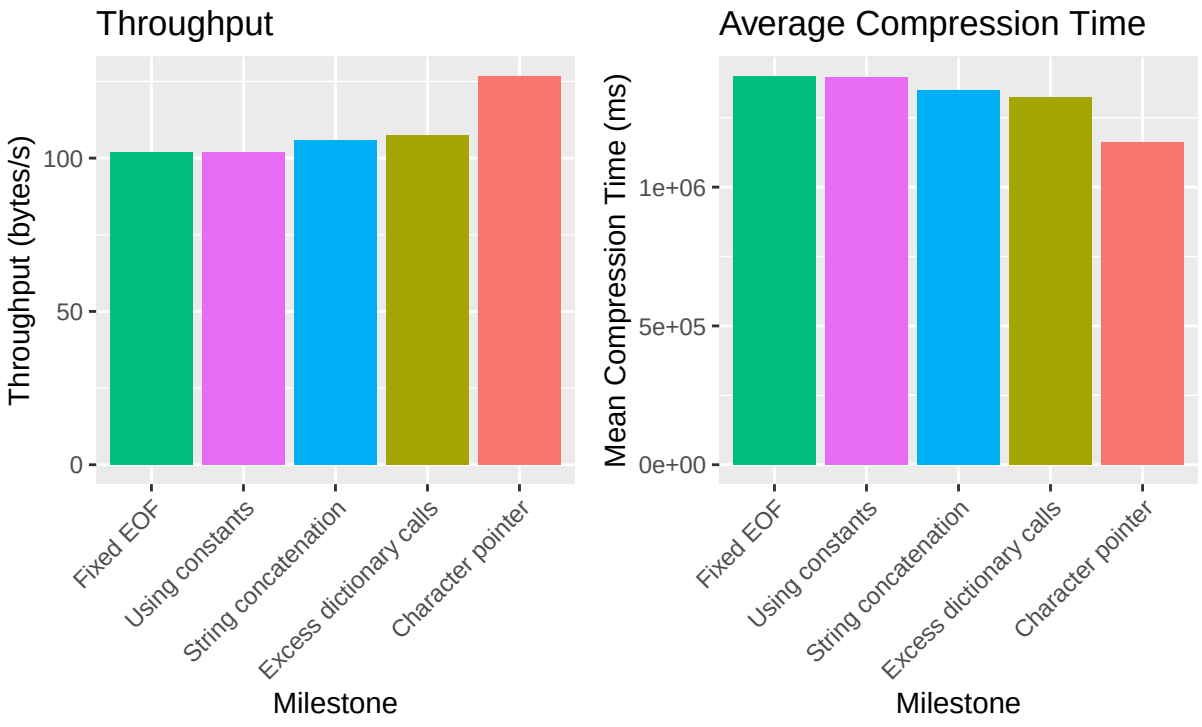

Figure 2.1: Comparison of the performance of the different milestones

Figure 2.1 shows the performance of the algorithm at the different milestones mentioned in this section. All graphs use the files from both corpora unless stated otherwise. On the left side of the figure you see that throughput increases for the

program with each optimization. On the right, observe that mean compression time decreases with each optimization.

2.4 Trying Different Dictionaries

A lot of the stress of the LZW algorithm is on the dictionary. We are constantly looking strings up and placing others. Because of the reliance on this data structure, we know that the dictionary accesses and lookups are a bottleneck, so improvements in those areas could greatly increase the efficiency of our program.

So another step towards an efficient LZW seemed to be to abstract out the C++ `std::unordered_map` and have multiple different dictionary implementations to try and experiment with in our attempt to optimize LZW for DNA compression.

2.4.1 Direct Map

In our analysis of the two corpora, we found some interesting statistics in the redundancy of the data. Tables **??** and **??** show stats on the run lengths of the "runs"

AVERAGE RUN LENGTH	MAXIMUM RUN LENGTH	MEDIAN RUN LENGTH	SD RUN LENGTH
6.135398	17	6	1.237217

AVERAGE RUN LENGTH	MAXIMUM RUN LENGTH	MEDIAN RUN LENGTH	SD RUN LENGTH
10.96825	190	11	2.494305

of data, where a run is a string added to the dictionary during a run of the LZW algorithm, and a histogram of runs from both corpora can be seen in Figure 2.2.

X	AVERAGE RUN LENGTH	MAXIMUM RUN LENGTH	MEDIAN RUN LENGTH	SD RUN LENGTH
1	10.95318	190	11	2.505904

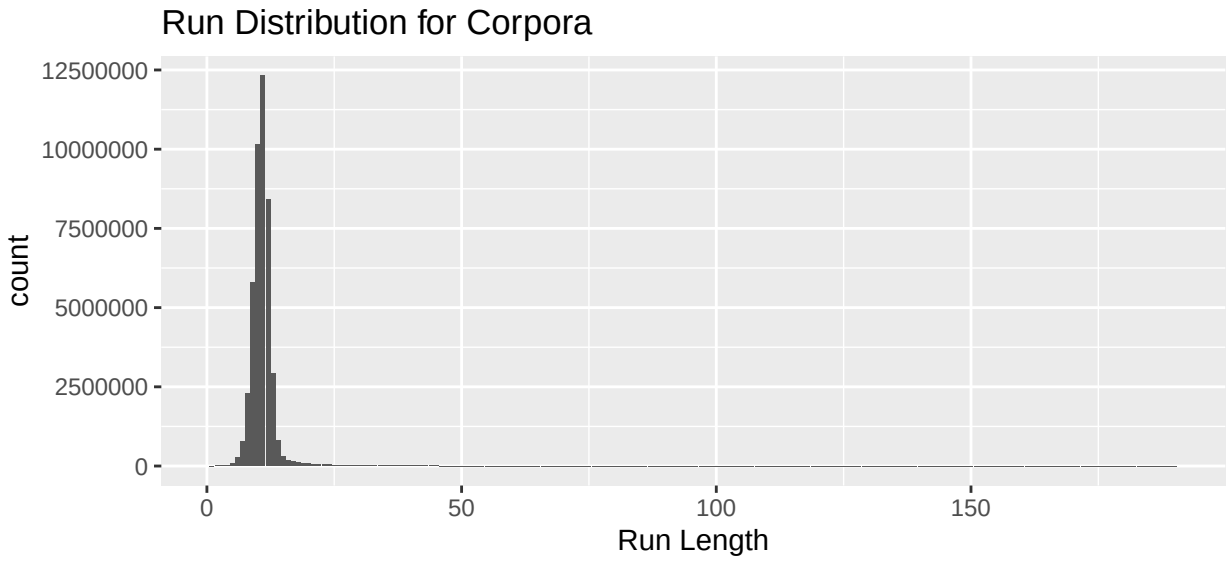

Figure 2.2: A histogram showing the lengths of runs for both copora.

Given this data, it is clear that it would be advantageous to speed up the dictionary for smaller run sizes, since most of the runs are below size 15. As stated before, there have been research papers about the possibility of using multiple indexed dictionaries for LZW, including Keerthy (Keerthy, 2019). To achieve a similar effect to multiple indexed dictionaries, we opted for a unique approach. Rather than use a hashmap, we can map the strings directly into memory. Since all of the strings only contain four characters ('A', 'C', 'T', and 'G'), we can represent the characters with two bits. So for a length `n` string, we can represent it with `2n` bits.

So we can create an indexed dictionary directly in memory for all strings below a certain length. We can use the `2n` bit representation of the string to index into an array of codewords.

For each string size 1 to n, we have an array with enough slots for every possible string. For example, for strings of length 3, we have an array of size 4^3, since there are 4^3 possible strings. In each of those 4^3 slots, we have space for a codeword. All strings of length 3 can be represented by 6 bits, and since 6 bits can represent $2^6 = 4^3$ values, we can use the bit representation to index into the dictionary. If the codeword at that place in the dictionary is 0, we have never seen it before. If it is non-zero, we have found the codeword for that string. For all strings greater than `n`, we can just use a hashmap on top to handle those.

As seen in Figure **??**, the both the average compression time and throughput are greater for a direct map dictionary with a max string length of 15 as opposed to a max string length of 10. This makes sense, because any string over the max requires

an entry in a hashmap, which takes much more time than a dictionary access. Using a string length of over 15 is difficult because the amount of memory required increases exponentially.

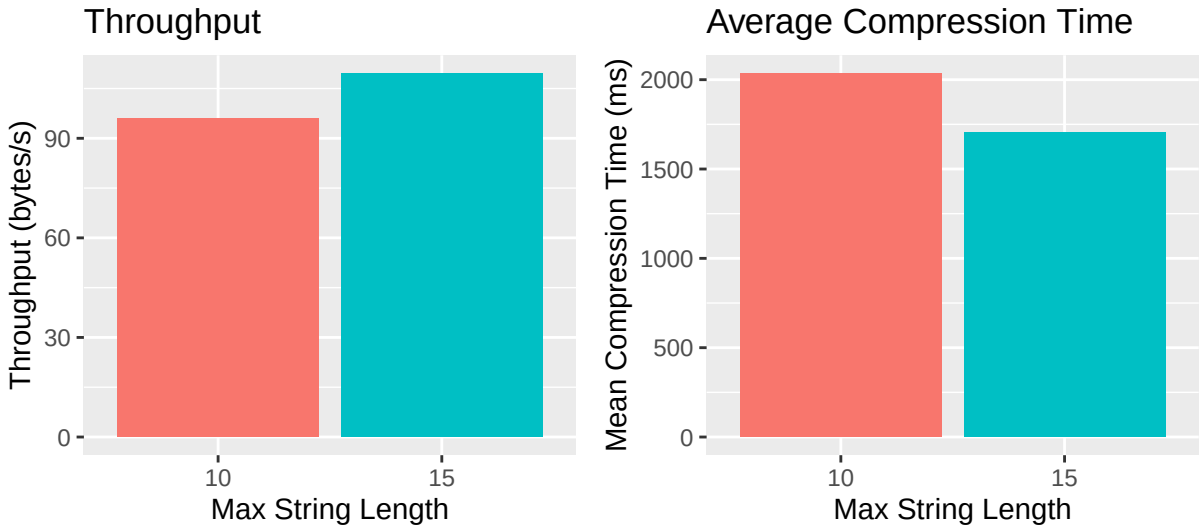

Figure 2.3: Comparing different max lengths for Direct Map

It's also worth noting that since we are allowing strings of all lengths, the compression ratio has not changed.

2.4.2 Multiple Indexed Dictionaries

Similar to the Direct Mapped approach, we use dictionaries for each string size up to a certain size `n` , and for all strings of length greater than `n`, we use a regular dictionary. As with the direct map dictionary, we need to specify a max string length. We collected metrics for different choices of max string length. As seen in Figure 2.4, the throughput tends to increase and the average compression time tends to increase as we increase the number of indexed dictionaries. This result was not necessarily one we expected, but it does make sense that there is a certain amount of overhead that is required for a hashmap. The hash function takes about the same amount of time no matter the number of elements in the map, and resizing is rare. So adding more dictionaries only adds more overhead, which tends to slightly decrease efficiency.

Compression ratio still remains the same as strings of all lengths are accommodated. This logic is supported by Figure 2.5, which shoes one `std::unordered_map` compared the multiple indexed model.

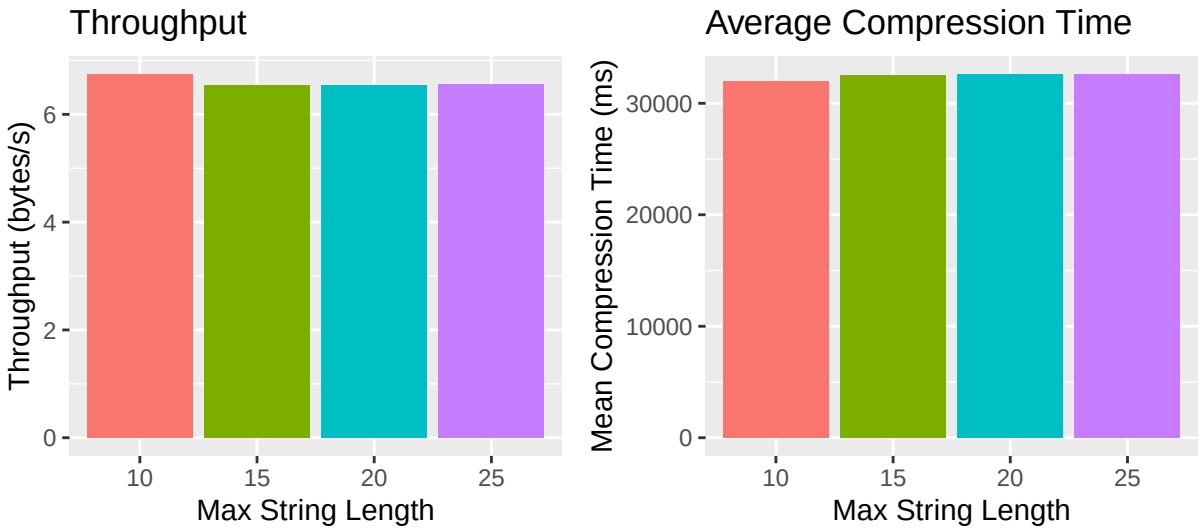

Figure 2.4: Comparing different max lengths for Direct Map

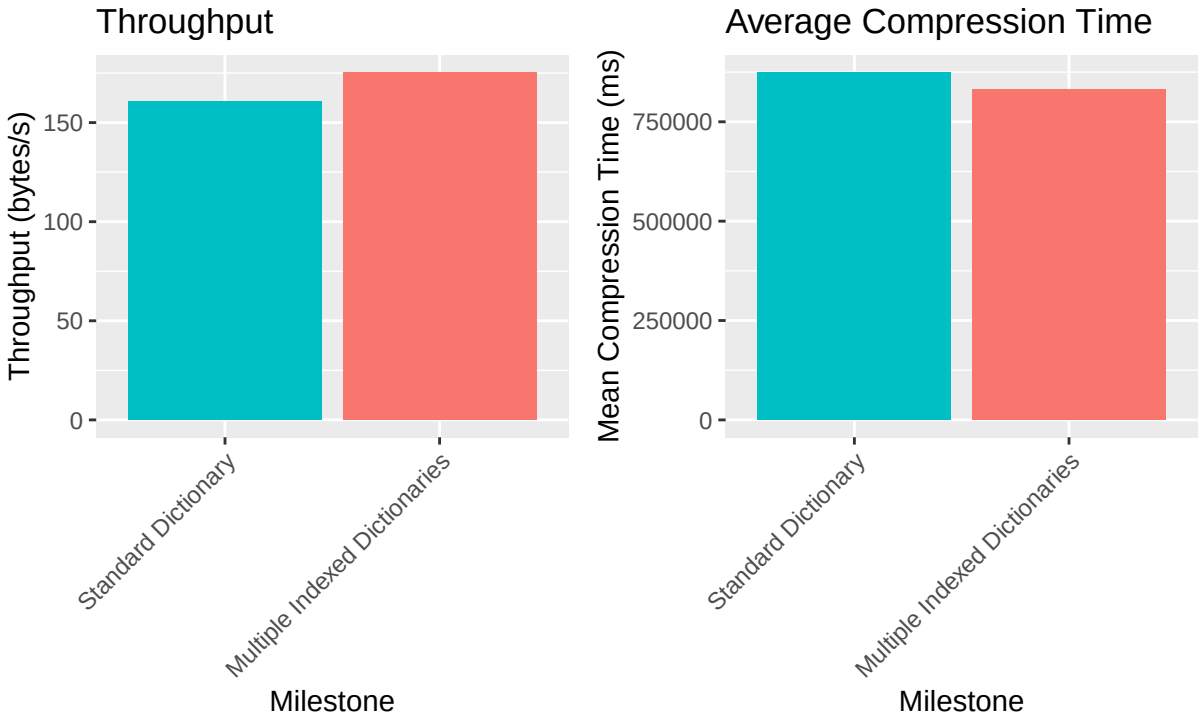

Figure 2.5: Comparing one Dict to Mult Dict

2.4.3 Comparison

Figure 2.6 shows a comparison of all three techniques: Standard Dictionary, Multiple Standard Dictionaries, and Direct Mapped Dictionary. As shown, the Direct Map technique greatly increases throughput and thus decreases average compression time. Given these results, we decided to shift our focus onto the Direct Map and try to optimize this scheme as much as possible.

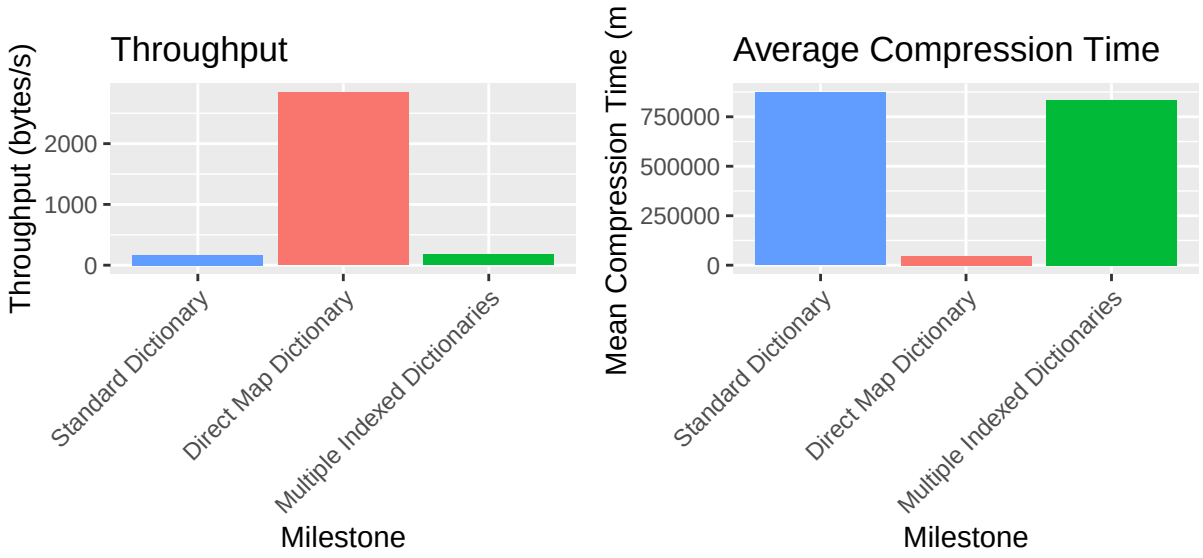

Figure 2.6: Comparing the three types of dictionaries. The Direct Map has a max length of 15 and the Mult Dict has a max length of 10.

2.5 Optimizing Direct Map Even more

Given that the Direct Map dictionary showed great performance improvements, we decided to narrow in on the scheme to see if there were ways to improve it even further.

2.5.1 Finding the Longest Runs

Our dictionary data structures all have a `get_longest_in_dict` function. This function does the boring work of iterating through the input from the start, checking if each substring is in the dictionary.

Given the statistics of our corpora, we know that this process can be faster. Since most runs are above 6-7, we waste a lot of time by starting from the bottom.

Another strategy would be to start from the maximum string length of the dictionary, so 15. We can check if the next string of the max length is in the dictionary. If it is, we need to check strings longer than the max, so we can iterate up. If it isn't, we need to check strings shorter, so we can either iterate down . Here is some pseudocode that mimics this proposed algorithm.

```
find_longest(input_string){

    // calculate the index of the next 15 chars
    // where index = converting each char to two bits
    index_of_next_15_chars = calculate_index(input_string[0:15]);

    // look up our string
    lookup = dictionary[index_of_next_15_chars];

    if(lookup is in dictionary){
        loop_up();
    }
    else {
        loop_down();
    }

}
```

Calculating the index, however, takes time. While we are looping up or down, we could just use the index of the next 15 characters as a starting point. If we are looping up, we can add on the next character's two bit representation as we loop. For instance, suppose our string has an index of `00101100`. If the next character is 'A', we can simply tack the two bit representation for 'A' to the end of our index, yielding `00101100|00`.

Similarly, we can chop off two bits at a time while looping down. We can name this process looping "on the fly", since we are constructing our index on the fly rather than recalculating it every time. So our modified pseudocode would look like the code below:

```
find_longest(input_string){

    // calculate the index of the next 15 chars
```

```
    // where index = converting each char to two bits
    index_of_next_15_chars = calculate_index(input_string[0:15]);

    // look up our string
    lookup = dictionary[index_of_next_15_chars];

    if(lookup is in dictionary){
        loop_up_on_fly(index);
    }
    else {
        loop_down_on_fly(index);
    }

}
```

Of course, there are theoretically quicker ways of iterating than looping up or down. We could use binary search.

Binary search is a searching technique for sorted lists, but we can use it in this scenario as well. Suppose we have a string of characters, and we are searching for the longest string already in the dictionary. The algorithm works by repeatedly dividing the search interval in half, comparing the middle element of the subarray to the target value, and then deciding whether to continue the search on the lower half or upper half of the subarray.

If the middle element is equal to the target value, the search ends and the position of the element is returned. If the middle element is greater than the target value, the search continues on the lower half of the subarray. Conversely, if the middle element is less than the target value, the search continues on the upper half of the subarray.

Binary search has a time complexity of O(log n), which makes it a very efficient algorithm for searching large sorted arrays. In our case, we search for where the string of length n is in the dictionary, but the string of length n+1 is not.

```
find_longest(input_string){

    // code omitted...

    if(lookup is in dictionary){
        loop_up_on_fly(index);
```

```
    }
    else {
        loop_down_binary_search(index);
    }

}
```

Of course, we could also calculate the indexes for binary search on the fly. So we now have 5 different schemes of finding the longest run: looping up or down, looping up or down on the fly, binary search, binary search on the fly, and looping up from 0 like we were doing before. Figure 2.7 shows a comparison of these methods.

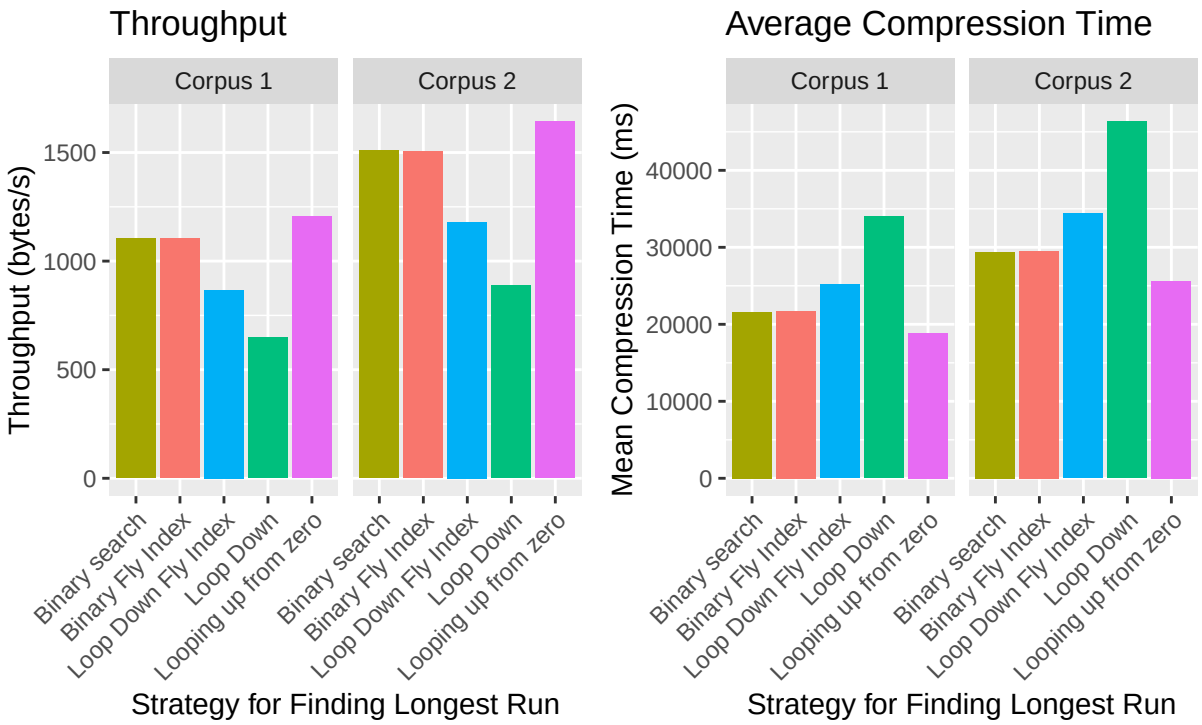

Figure 2.7: Comparing the different ways of finding the longest run

We see that calculating the index on the fly does improve performance. However, we can also see that none of the other strategies are better than just looping up from zero. This could be because most of the runs are very short, which we can see in 2.2. This means that if we start looking from runs around length 10, we should see a performance improvement.

2.5.2 Finding the Longest From The Average

As we can see in Figure 2.8, looping or doing binary search from the average run length, which we approximate at 7, increases throughput and decreases mean compression time relative to looping up from zero.

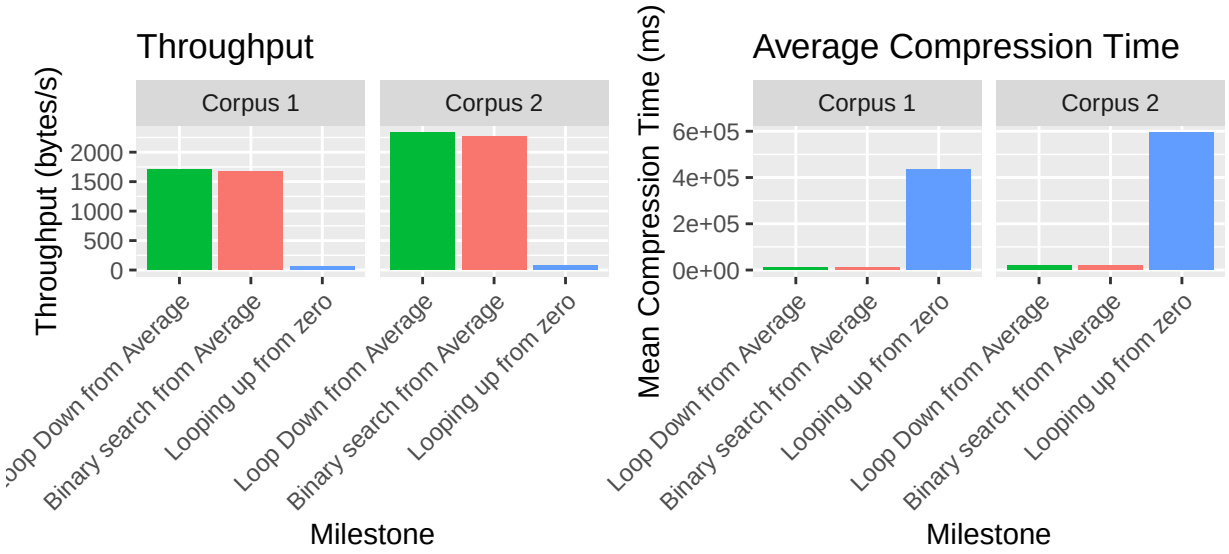

Figure 2.8: Comparing the different ways of finding the longest run

As we predicted, this greatly improves performance. We are capitalizing on the fact that most runs are short. There is the occasional run that is very long, but from the run statistics we saw that the standard deviation for both corpora was pretty small. So on the edge cases in which there are very long runs, we could be wasting a lot of time.

2.5.3 Not Allowing strings over max

One way to avoid the edge case where we encounter a very long run is to just not allow strings in the dictionary longer than the max string length, in this case, 15. This means we won't have to deal with the overhead of the hashmap on top of our dictionary, but we will take a hit in compression ratio. Figure 2.9 summarizes the results of the different methods with the max length rule enforced. There isn't too much of a performance change, which makes sense because long runs are very rare.

Table 2.3: Compression Ratio change from disallowing long strings

cr_before	cr_after
2.909175	2.907686

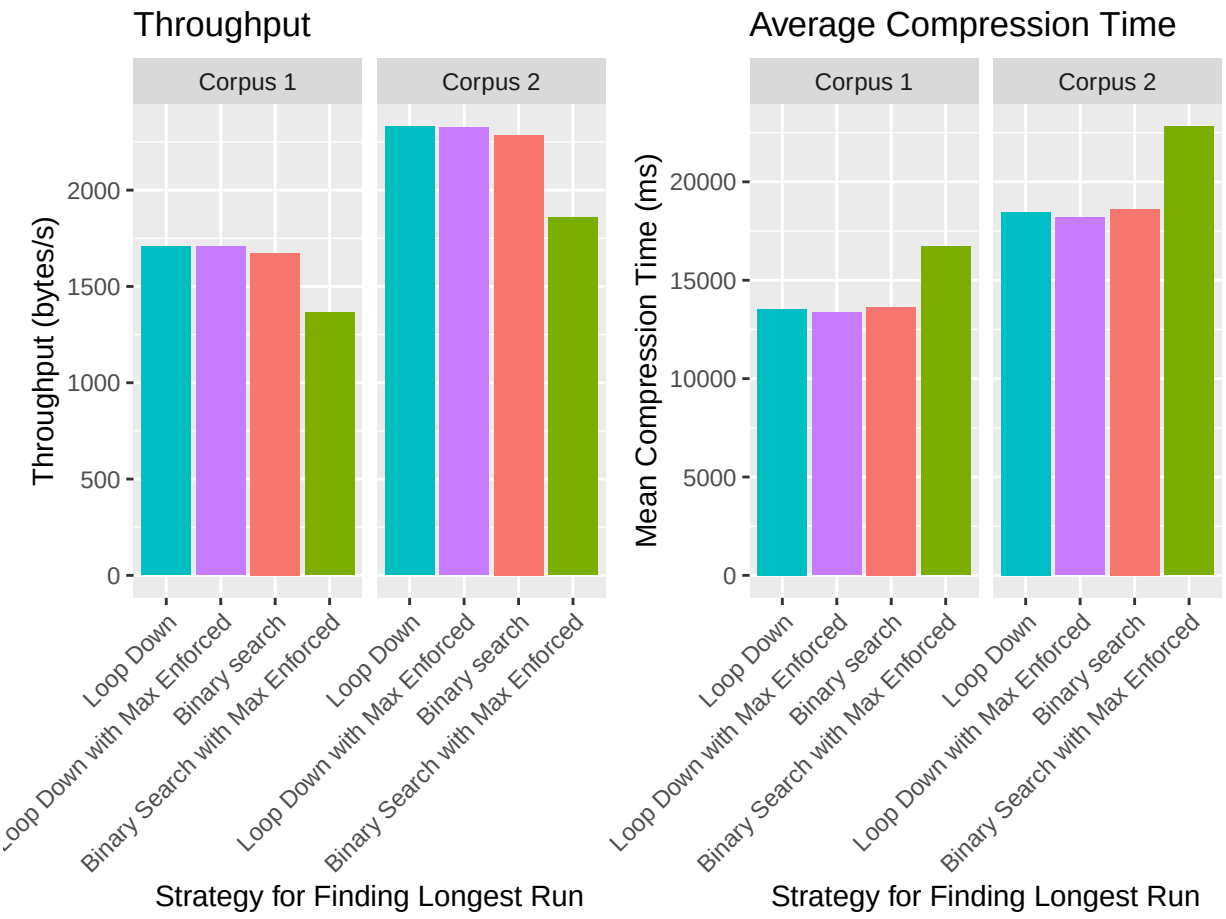

Figure 2.9: Comparing the different ways of finding the longest run starting at average with strings over max not accepted

Of course, not allowing string over max length does mean that our compression ratio will change. Up until this point, all the different versions of the Direct Mapped Dictionary had the same compression ratio. Table 2.3 shows the affect of enforcing the max string length on the total compression ratio over both corpora. As we can see, it is minimal, which makes sense because long runs are very rare.

2.5.4 Using `pext`

One potential bottleneck of finding the longest run is converting a run of characters into an index. We can try to do it on the fly as we loop up or down, but we could also use machine instructions.

Our strategy is to use `pext`, which extracts bits in parallel, meaning at the same time. The `pext` instruction is a recent addition to the x86 istruction set, so it has not been widely used yet as optimization technique.

We give `pext` a string of characters, say 'ACTG', and a bit mask, and it will extract those bits from our string. Figure 2.10 details this process for a string of length 4.

It theoretically does this in one machine instruction, which could be much more efficient than looping over all the characters. We can apply this technique to our

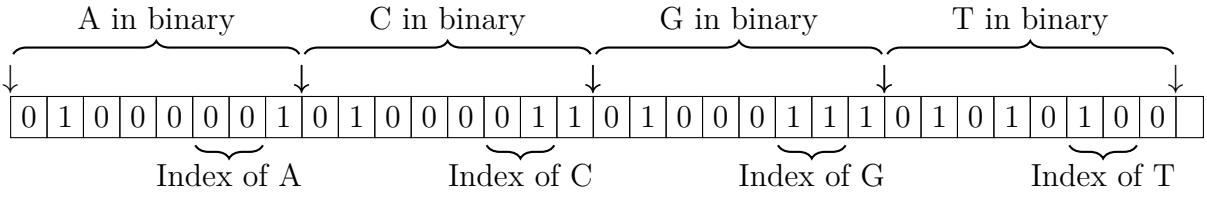

Figure 2.10: How 'pext' extracts bits

`find_longest` function: we can extract the index of a string using pext very quickly, then use that index rather than recalculating it every time. Figure 2.11 hows the results of this application for both looping and binary search. In both cases, using pext to extract the index of the string increases throughput and decreases average compression time.

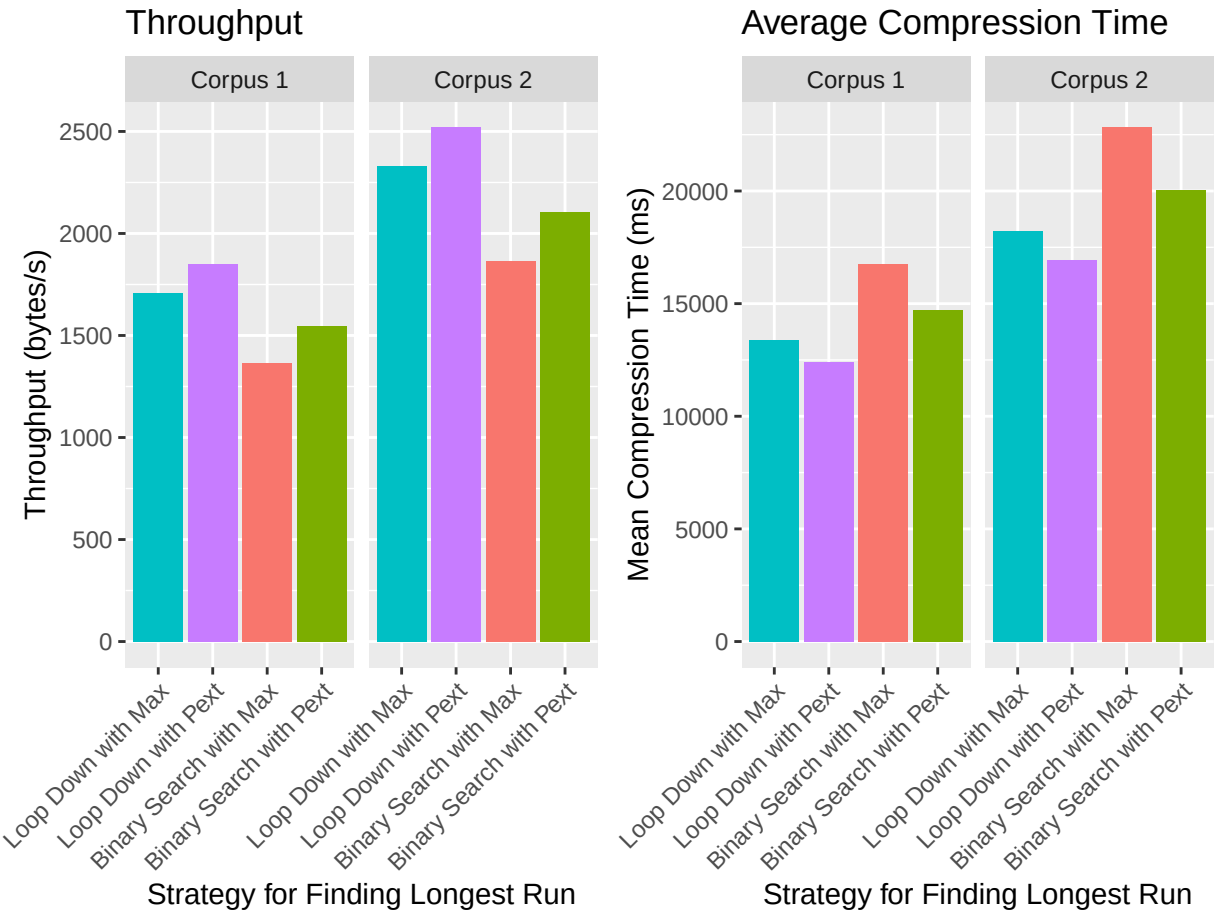

Figure 2.11: Comparing the different ways of finding the longest run with pext

2.6 Returning to Compression Ratio

After spending a lot of time debugging, we returned our attenting to the compression ratio. Most of our optimizations didn't have much of an effect on compression ratio, with the exception of disallowing strings over a certain length. It is also worth noting that the Direct Map dictionary and the Std dictionary implementations will have different compression ratios, as the Direct map has a max codeword size of 16 bits while the Std dict has a max codeword size of 32 bits. Std dict also allows longer strings.

2.6.1 A point of Comparison

It's worth noting that for sequences of DNA, if the bases are encoded in ASCII, there is a simple and effecient algorithm to compress the file with a 4.0 compression ratio every time. Since there are only 4 bases, we can represent each base with 2 bits. This is a simple conversion, and it requires no dictionaries or codewords.

So the takeaway is, if a DNA compression algorithm can't compress with a compression ratio of higher than 4.0 (2 bits per base), than this simple algorithm would be perferable every time.

We implemented this method using pext, and the results are detailed in the next chapter.

2.6.2 Entropy Encoding

Our initial idea was that we would compress the DNA with LZW, then use a entropy encoder like arithmetic encoding or Huffman to further compress the data. This was an oversight, as the algorithm as we have described it thus far does not lend itself well to entropy encoding.

Our algorithm outputs codewords and two bits representing the next character. For a codeword size of 16 bits, this means each loop of our algorithm outputs 18 bits. So any repretitiveness or reuse will not be detectable by an entropy encoder which works on data of 8, 16, or 32 bit chunks.

So maybe we break the compressed file into two seperate files; a file of codewords and a file of the two bit representations of the following characters. Now we can compress these two files, right?

The issue is that entropy encoders rely on repeating numbers of a high frequency. We can cut down the number of bits a certain entry takes to make it more compressible, and make less common entries take more bits (see Figure 1.1). DNA neucleotides show up with roughly the same frequency, so having 2 bits per character is hard to beat. So the file with all the charaters in it can't be compressed further.

What about the codewords? Well, a key realization is that any codeword will only show up 4 times in a single run of the program. Say we add the codeword `1234="ACT"` to our dictionary. When will we output this codeword ? Well, we will output it when we see "ACT" followed by another character. Since there are only 4 characters, once you see those 4 other strings ("ACTA", "ACTG", "ACTC", and "ACTT"), we will never output it again because we are looking for the longest run possible.

There are two cases in which we will output a codeword more than 4 times. If

we run out of codewords and we have never encoded "ACTG", any time "ACTG" comes up, we will output `1234G`. This case is hard to predict as we have no way of controlling what strings are left when we run out of codewords. The other case is for strings that are the max length. These long runs may show up multiple times, and since we won't add any strings longer than the max to our dictionary, it will never be overwritten. However, these runs are very rare, or else the compression ratio wouldn't be an issue.

So on average, every codeword shows up a maximum of 4 times. This means that for long strands of DNA, we have output nearly all of the codewords with relatively even frequency. In other words, entropy encoding will not shorten the length of the codewords.

2.6.3 A New Approach

Given that we are not able to achieve a compression ratio over 4.0 with the current LZW, we need to alter our approach. The issue with our dataset is that long runs are rare. Every once in a while, you may replace a run of 15 with a codeword, but most of the time you are replacing a run less than 8. Eight characters can be represented by 16 bits, so any run under 8 that is replaced loses bits

The fact that we are not getting a compression ratio over 4.0 led us to think of a new twist on the algorithm specifically tailored for this situation. The scheme uses three streams of data: characters, codewords, and indicator bits. The algorithm works as follows.

Compression:

- if the next longest run is less than 8 characters, we add it to the dictionary, but rather than output the codeword, we just output the 2 bit representation of the characters. We output a 0 to the indicator stream to indicate this choice.
- if the next longest run is equal to or greater than 8, we output a codeword to the codeword stream and the next character to the character stream. We also output a 1 to the indicator stream to indicate a codeword was output.

Decompression: We start by reading an indicator bit

- if the bit is 1, we read a codeword from the codeword stream and the next character from the character stream. We add this to the dictionary like the old algorithm.

- if the bit is 0, we read the next 8 characters. We then use `find_longest` to find the longest run and add it to the dictionary. We can then put the 8 characters into the output stream and move on

This is, in essence, a greedy algorithm. We have the choice of outputting a codeword or 8 characters every time, and we choose the choice which results in the least number of bits output.

The other advantage to this algorithm is that it works well for entropy encoding. The indicator bits are mostly 0, since most runs are less than 8. The codeword stream is also compressible, since none of the codewords for strings less than length 8 are output. The performance of this new scheme is assessed in the next chapter.

Chapter 3

Comparison to other tools

Now that we have implemented several versions of LZW, we want to compare them to each other to see which would be preferable in different situations. We also want to compare the performance of these different implementations to the 4 to 1 direct translation and other professional compression tools.

3.1 Comparison of our Implementations

3.2 Compression Algorithms in Literature

As discussed in the related work section of chapter 1, there have been several other compression algorithms proposed and tested in the field tailored for DNA. Not all of these are publicly available, thus we can only compare to the numbers that they reported.

Note that I wasn't able to get compression times for these algorithms, only the compression ratios.

3.3 Comparison to Other Professional Tools

Sequence	BioCompress2	GenCompress	XM
CHMPXX	1.6848	1.6730	1.6577
CHNTXX	1.6172	1.6146	1.6068
HEHCMVCG	1.8480	1.8470	1.8426
HUMDYSTROP	1.9262	1.9231	1.9031
HUMGHCSA	1.3074	1.0969	0.9828
HUMHBB	1.8800	1.8204	1.7513
HUMHDAB	1.8770	1.8192	1.6671
HUMHPRTB	1.9066	1.8466	1.7361
MPOMTCG	1.9378	1.9058	1.8768
MTPACG	1.8752	1.8624	1.8447
VACCG	1.7614	1.7614	1.7649

Conclusion

If we don't want Conclusion to have a chapter number next to it, we can add the `{-}` attribute.

More info

And here's some other random info: the first paragraph after a chapter title or section head *shouldn't be* indented, because indents are to tell the reader that you're starting a new paragraph. Since that's obvious after a chapter or section title, proper typesetting doesn't add an indent there.

Appendix A

The First Appendix

This first appendix includes all of the R chunks of code that were hidden throughout the document (using the `include = FALSE` chunk tag) to help with readibility and/or setup.

In the main Rmd file

```
# This chunk ensures that the thesisdown package is
# installed and loaded. This thesisdown package includes
# the template files for the thesis.
if (!require(remotes)) {
  if (params$`Install needed packages for {thesisdown}`) {
    install.packages("remotes", repos = "https://cran.rstudio.com")
  } else {
    stop(
      paste('You need to run install.packages("remotes")",
            "first in the Console.')
    )
  }
}
if (!require(thesisdown)) {
  if (params$`Install needed packages for {thesisdown}`) {
    remotes::install_github("ismayc/thesisdown")
  } else {
    stop(
      paste(
        "You need to run",
```

```
        'remotes::install_github("ismayc/thesisdown")',
        "first in the Console."
      )
    )
  }
}
library(thesisdown)
if (!require(DiagrammeR)) {
  install.packages("DiagrammeR")
}
library(DiagrammeR)
# Set how wide the R output will go
options(width = 70)
```

In Chapter ??:

Appendix B

The Second Appendix, for Fun

References

Cao, D., Dix, T., Allison, L., & Mears, C. (2007). A simple statistical algorithm for biological sequence compression. In *In Proceedings of the Conference on Data Compression* (pp. 43–52). http://doi.org/10.1109/DCC.2007.7

Chen, X., Kwong, S. T. W., & Li, M. (2001). A compression algorithm for DNA sequences. *IEEE Engineering in Medicine and Biology Magazine*, *20*, 61–66.

Grumbach, S., & Tahi, F. (1994). A New Challenge for Compression Algorithms: Genetic Sequences. *Information Processing and Management*, *30*. Retrieved from `https://hal.inria.fr/inria-00180949`

Ibrahim, M., & Gbolagade, K. (2020). Enhancing computational time of lempel-ziv-welch-based text compression with chinese remainder theorem. *Journal of Computer Science and Its Application*, *27*. http://doi.org/10.4314/jcsia.v27i1.9

Keerthy, P. (2019). Genomic sequence data compression using lempel-ziv-welch algorithm with indexed multiple dictionary. *International Journal of Engineering and Advanced Technology*.

Pani, A., Mishra, M., & Mishra, T. (2012). Parallel lempel-ziv-welch (PLZW) technique for data compression. *International Journal of Computer Science and Information Technology*, *3*, 4038–4040.

Pratas, D., & Pinho, A. (2018). A DNA sequence corpus for compression benchmark. In (pp. 208–215). http://doi.org/10.1007/978-3-319-98702-6_25