# 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

Thank you to Eitan Frachtenberg, who helped me constantly throughout the whole year writing this. I never thought thesising would be so much fun, and I hope you enjoyed it as much as I did. Thanks also to all the other staff and faculty who helped me through my four years at Reed. Thank you to B Hunter, who was enormous help to me in getting jobs and opportunites while at Reed. Thank you also to David Ramirez, whose advice and mentorship I am always grateful for.

# List of Abbreviations

| | |
|---|---|
| **DNA** | Deoxyribonucleic acid |
| **EOF** | End of file |
| **LZW** | Lempel Ziv Welch |
| **RLE** | Run Length Encoding |

# Table of Contents

**Chapter 1: Background and Motivations** . . . . . **1**
1.1 What is information? . . . . . 1
1.2 Compression Metrics . . . . . 2
1.2.1 Compression Ratio . . . . . 2
1.2.2 Compression Time . . . . . 2
1.2.3 Memory Usage . . . . . 2
1.3 Lossy vs. Lossless Compression . . . . . 3
1.3.1 Lossy . . . . . 3
1.3.2 Lossless . . . . . 3
1.4 Classic Compression Algorithms . . . . . 3
1.4.1 Run Length Encoding . . . . . 3
1.4.2 Huffman . . . . . 4
1.4.3 Arithmetic . . . . . 5
1.4.4 Lempel-Ziv-Welch . . . . . 6
1.5 Related work . . . . . 11

**Chapter 2: Optimizing LZW: Approach** . . . . . **13**
2.1 Corpora . . . . . 13
2.2 Evaluating Performance . . . . . 15
2.3 A Starting Point . . . . . 15
2.3.1 Growing Codewords and Bit Output . . . . . 16
2.3.2 Getting EOF to work . . . . . 17
2.3.3 Using Constants . . . . . 19
2.3.4 Extraneous String Concatenations . . . . . 21
2.3.5 Dictionary Lookups . . . . . 22
2.3.6 Using Const Char * . . . . . 24
2.3.7 Comparison . . . . . 24
2.4 Trying Different Dictionaries . . . . . 25

2.4.1 Direct Map . . . . . 26
2.4.2 Multiple Standard Dictionaries . . . . . 29
2.4.3 Comparison . . . . . 31
2.5 Optimizing Direct Map Even more . . . . . 32
2.5.1 Finding the Longest Runs . . . . . 32
2.5.2 Finding the Longest From The Average . . . . . 36
2.5.3 Not Allowing strings over max . . . . . 36
2.5.4 Using `pext` . . . . . 38
2.6 Returning to Compression Ratio . . . . . 39
2.6.1 A point of Comparison . . . . . 39
2.6.2 Entropy Encoding . . . . . 40
2.6.3 A New Approach . . . . . 42

**Chapter 3: Comparison to other tools . . . . . 45**
3.1 Comparison of our Implementations . . . . . 45
3.2 Compression Algorithms in Literature . . . . . 48
3.3 Comparison to Other Professional Tools . . . . . 49

**Conclusion . . . . . 51**

**Appendix A: Appendix: The Code . . . . . 53**

**References . . . . . 55**

# List of Tables


1.1 An example of LZW ran on the input "AAGGAATCC" . . . . . 8

2.1 Corpus 1 . . . . . 13
2.2 Corpus 2 . . . . . 14
2.3 Run statistics in Corpus 1 . . . . . 26
2.4 Run statistics in Corpus 2 . . . . . 26
2.5 Comparison of Compression Ratios between Direct Map and Standard Dictionaries . . . . . 29
2.6 Compression Ratio change from disallowing long strings . . . . . 38
2.7 Comparing compression ratios of the two Dictionary versions. . . . . 40

3.1 Performance metrics for our final implementations. . . . . 46
3.2 Compression ratios of related works on DNACorpus 1, reported as bits per base . . . . . 49
3.3 Performance metrics for other professional tools. Xz, bzip, and gzip were ran with option -9, and genozip was ran with –input=generic. . 50

# List of Figures


1.1 Example Huffman tree . . . . . 4
1.2 Example of arithmetic encoding. . . . . 5

2.1 Comparison of the performance of the different milestones. . . . . 25
2.2 A histogram showing the lengths of runs for both copora. . . . . 26
2.3 Comparing different max string lengths for the Direct Map Dictionary. 28
2.4 Comparing different max string lengths for Multiple Standard Dictionaries. . . . . 30
2.5 Comparing one Standard Dictionary to Multiple Standard Dictionaries (with a max string length of 10). . . . . 31
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. . . . . 32
2.7 Comparing the different ways of finding the longest run. . . . . 35
2.8 Comparing the different ways of finding the longest run starting from the average. . . . . 36
2.9 Comparing the different ways of finding the longest run starting at average with strings over max (15) not accepted. . . . . 37
2.10 How ‘pext‘ extracts bits . . . . . 38
2.11 Comparing the different ways of finding the longest run with pext. . . 39

3.1 Performance comparison of our final implementations. . . . . 46
3.2 Average Decompression time of our final implementations. . . . . 48
3.3 Average compression time of other methods and our final implementations. . . . . 50

# Abstract

Large files containing DNA sequences can be bottlenecks for genetic research, as it can be very difficult to store and transfer files that are gigabytes or even terabytes in size. In this thesis, we focus on compressing DNA sequences using the Lempel-Ziv-Welch (LZW) compression algorithm. LZW relies heavily on a dictionary type data structure, hindering its speed in compression. We propose a data structure we call the Direct Map Dictionary. Using the machine instruction `pext`, we were able to make the Direct Map Dictionary very fast and specifically tailored for DNA. Using this new data structure, we implement a greedy version of LZW we call Three Stream LZW which is able to compress large DNA files with a good compression ratio and notable speed, even when compared to professional tools.

# Chapter 1

# Background and Motivations

This thesis deals with some high-level topics and uses language specific to compression research. This chapter gives 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 use words, you could use sign language. Regardless of the medium, there is some important idea 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, let's assume that information is data with significance that makes it worth preserving 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, and screens. 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 original, uncompressed size ($OS$) to 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 Compression 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. In this case, we are interested in compression time. Sometimes, if time is constrained, you may care less about saving space. For example, suppose 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. If the one with the higher compression ratio 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 is generally the storage a program uses while it is running. 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 compression time.

## 1.3 Lossy vs. Lossless 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 photo, 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 devices with limited storage 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.

## 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 encoding the number of times each character appears. Suppose you want to send the following message

$$AAGCTTTTTTTTGGGGGCCCT$$

We can still get this message across without repeating ourself quite as much. When writing a grocery list, you don't write "egg egg egg egg", you say "4 eggs". RLE uses this same strategy.

$$2A1G1C8T5G3C1T$$

Although not as sophisticated as other methods, RLE is effective when used on applications with large runs of repetitive data. For instance, photos that have solid backgrounds, like a logo, can be effectively compressed by RLE.

### 1.4.2 Huffman

Huffman Encoding is a lossless compression algorithm 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 characters 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 1.1 down to a letter, it will tell you the code associated with that letter. So $G = 0$, $A = 10$, $T = 111$

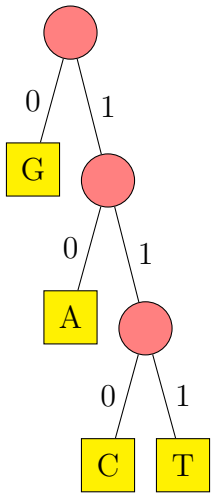


Figure 1.1: Example Huffman tree

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 "ACGGT". 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 probabilities of each symbol in the message are

- $P(A) = a_1 = 2/10$
- $P(C) = a_2 = 2/10$
- $P(G) = a_3 = 4/10$
- $P(T) = a_4 = 2/10$

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.

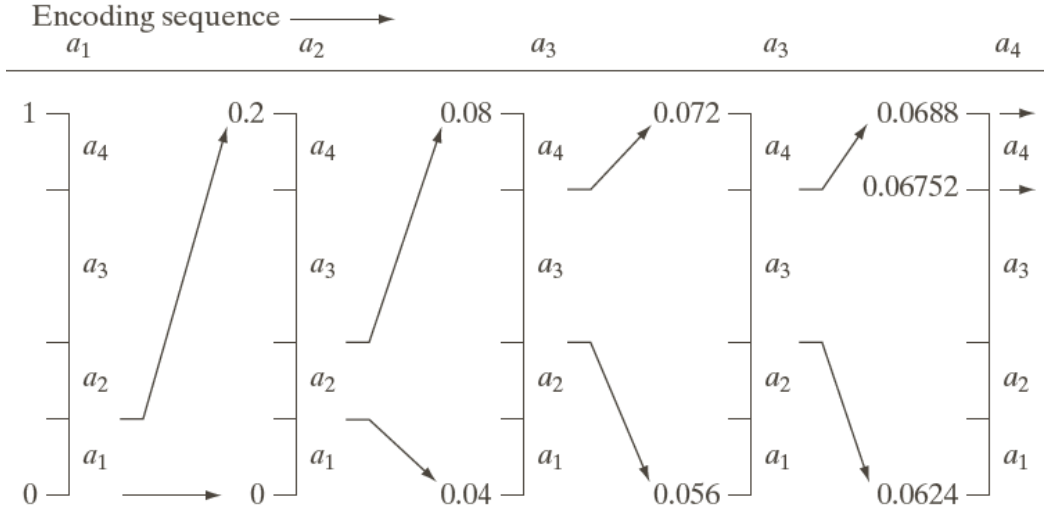


Figure 1.2: Example of arithmetic encoding.

Since 'A' comes first, we divide [0,1] into [0.0,0.2) since $P(A) = 2/10$. Since 'C' is next, we divide the new fraction of the space [0.0, 0.2) into chunks based on the probabilities of the characters and choose the chunk corresponding to 'C'. Figure 1.2 shows this process through the whole string "ACGGT". We end up with an interval from 0.0688 to 0.06752.

So any number in the interval can be used to represent our message as long as the decoder knows the probabilities that we used to encode it.

### 1.4.4 Lempel-Ziv-Welch

Lempel-Ziv-Welch is another lossless compression algorithm. When compressing, LZW builds a dictionary of codewords. A dictionary is a key value system: when you give it a key, it either gives you a value or tells you that the key does not exist. In this case, the keys are strings of characters and the values are codewords, which are numbers meant to represent those strings.

Here is a simple example. Suppose we decide that 'AA' = 2. In other words, any time we see the number 2, we know that it actually means 'AA'. So if we wanted to send a message

AACAAC

We could just say

2C2C

As long as the person recieving this message understood that 'AA' = 2, they could easily determine what I meant. LZW works in the same way. This rule that we started with, that 'AA' = 2, is essentially a dictionary. It holds keys and values. To make the example more complex, suppose we decide that {'AA' = 1, 'C' = 2}. We could encoding the message "AACAAC" as

1212

As you can see, our message is getting shorter. The problem is, how do we decide what elements start in our dictionary? Well, we can start with a small dictionary, and build up a larger dictionary as we go along. Let me explain. Suppose we are still trying to encode "AACAAC". Assume we start with the dictionary {'AA' = 1}. We can scan through the message, looking for instances of 'AA' and replace them. We see "AAC" at the beginning of the string. We can encode this as "1C". Here is our message so far

1CAAC

But what if we see the string "AAC" again? Well, when we output "1C", we can

add this new string, "AAC", to our dictionary. So our dictionary now looks like {'AA' = 1, 'AAC' = 2}. Notice we just assign the new string whatever number we left off on while creating the dictionary.

Now, as we get to the end of the message, we see "AAC" again. Now that we have a codeword for "AAC", we can just output 2. So our final message looks like

1C2

Now the person decoding, when they see "1C", they know that that means "take the string that has codeword 1, add on the character 'C', and add that new string to your dictionary". So they will add 'AAC' = 2 to their dictionary. When they see the following 2, they know that 2 = 'AAC', so they are able to decode the message. In this way, we can build up the dicionary as we encode and decode, and as long as we start with the same starting dictionary, the message will always be preserved.

Here is a more complex 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}. So in other words, 'A' is synonymous with 0, 'C' is synonymous with 1, and so on. Say we compress 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?"

**A**AGGAATCC

Well, we started with 'A' in our dictionary, so we can check in the dictionary and confirm it is there. We then add on the next character in the sequence and ask "Is"AA" in our dictionary?"

**AA**GGAATCC

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".

| 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"

So the encoded string will look something like

0A....

Table 1.1 illustrates this example fully.

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 “AA” to the dictionary and assign it to our next available codeword, 4. Again, 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. LZW 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.
- There only needs to be one run over the data to encode and decode. This means that the run time of the algorithm should 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 pseudocode

```
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 `std::unordered_map` and uses a hash table, but the concept is the same. During this thesis, we will use `std::unordered_map` and Standard Dictionary interchangeably.
- `codeword`: a number that is assigned to a string in our encoding and decoding. The it is called a `codeword` because it is a code for the string it represents.

- `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

Deoxyribonucleic acid (DNA) is the building block of life. In four letters (A, C, G, and T) called nucleotides, DNA holds the genetic information required for reproduction in all living organisms. DNA can be billion of characters long, which can mean gigabytes or terabytes of storage on a computer. 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 have 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. This allows each of the dictionaries to grow more slowly, allowing accesses to be 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.

Another paper made use of Chinese Remainder Theorem (CRT) 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 were not able to achieve similar results. We reached out to the authors, but we were not able to further our progress on this method and thus it is not used in this thesis.

DNA-specific compression algorithms have also been a growing subsection of com-

puter science for decades. 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 allows them to achieve an above average compression ratio, though performance is not evaluated. 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 sequences of nucleotides to sequences already seen in the file, and maps those sequences using various edits to turn one sequence into another. They are able to achieve a great compression ratio with this relative matching method, although their technique is computationally expensive.

In 2007, 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.

A paper in 2021 uses segmenting and partitioning to create a parallel compression algorithm called `genozip` (Lan, Tobler, Souilmi, & Llamas, 2021). This paper also published their code online in a convenient format which allows others to compare and test their implementation.

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 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 use C++. A majority of the work in this thesis involved writing, 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 compression ratio 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 algorithm to others available.

Table 2.1: Corpus 1

| Name | bytes |
|---|---|
| chmpxx | 121024 |

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

In the world of DNA compression, there are several academic papers on the subject. As mentioned in Chapter 1, one of the first and most popular of the papers was published in 1994. 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). We call this Corpus 1, summarized in Table 2.1.

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. Since the papers publishing, it has been cited by several DNA compression studies.

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 |

| Name | bytes |
|---|---|
| HoSa | 189752667 |
| OrSa | 43262523 |
| PlFa | 8986712 |
| ScPo | 10652155 |
| YeMi | 73689 |

The dataset in Table 2.2, which we call Corpus 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, the stats were taken on the same computer. Our test machine was a PC with an AMD Ryzen 5 5500 processor 32 gigabytes of RAM running Ubuntu version 20.04. 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 the time of compression divided by the total number of files. All graphs use all the files from both corpora unless stated otherwise.

## 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 it harder to tell if the different techniques we have are affecting

performance.

This section chronologically covers some milestones reached when implementing an efficient initial version of LZW. These milestones are compared to one another at the end of the section.

### 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, the typical encoding scheme for plain English text. 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 them also grows. We need to always have enough bits to represent our current largest codeword, otherwise the decoder won't know how many bits each codeword is. So if we are on codeword 8, we need 4 bits since 8 is `1000`. We will need 4 bits until our codeword reaches 16, which is `10000` in binary. From then on, we will need to use 5 bits for every codeword. 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.

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 discussed in Section 1.4.4, if we don't mark the end of the file, the decoder won't be able to know when to stop decoding. So the decoding 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 finds the end of file, we stop and output the EOF codeword.

The problem was, what about what is left over? When compressing, the algorithm could be part of the way through a search when it reaches the end of the file. 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.

### 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 developer's disposal when looking to debug and optimize code. One tool used for this thesis was `callgrind` which is 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 tool which associates assembly instructions to lines of code, indicating which lines take a lot of instructions and which take less. For those unfamiliar, assembly instructions are what code is converted 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
11,653,779,430 (19.65%)  => /usr/include/c++/9/bits/unordered_map.h
2,108,383,120 ( 3.56%)  => /usr/include/c++/9/bits/basic_string.h
956,941,242 ( 1.61%)  => /usr/include/c++/9/bits/unordered_map.h
241,180,314 ( 0.41%)  => /usr/include/c++/9/bits/hashtable_policy.h
```

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 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
2,833,786,025 (19.26%)  => /usr/include/c++/9/bits/unordered_map.h
420,120,704 ( 2.85%)  => /usr/include/c++/9/bits/basic_string.h
50,570,065 ( 0.34%)  => /usr/include/c++/9/bits/hashtable_policy.h
```

There are several potential issues with what `callgrind` attempts to do by associating machine instructions with individual lines. 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 compilers 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 indicate trouble spots in the code which can be improved.

### 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 concatenations in most languages, including C++, have a lot of overhead. A lot of implementations of string concatenation involve creating a new string every time you concatenate two existing strings, which can have a significant performance penalty.

In this version of the algorithm, 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. You may have noticed this extraneous concatenation in the if statement in Section 2.3.3.

```
// 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 we just save `currentBlock + next_character` into a new variable, that will prevent doing the concatenation 2 extra 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;
}
```

### 2.3.5 Dictionary Lookups

Dictionary lookups can be expensive, especially with the Standard Dictioary. We learned from `callgrind` that along with string operations, our program spent a large fraction of the total runtime doing these lookups.

We looked for ways to reduce the volume of lookups. At the time, the algorithm looked up the current string and the next character in the dictionary. If that string was in the dictionary, it keeps adding characters. If the string was not in the dictionary, then the algorithm outputs the codeword for the current string.

But, the current string on this iteration is just 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.

Here is some of the code to further explain this point.

```
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
    }
    else{

        // on the next iteration, we use it here
```

```
        int code = codeword_seen_previously->second;

        // code omitted

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

This should reduce the amount of time the program spends on dictioary lookups.

### 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 always be in memory, which is the fastest part of our storage. The `ifstream` class in C++, the typical method of reading a file, has many extraneous features 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. Rather than concatenate strings at all, we can pass around a pointer to the beginning of the data and a number which is an offset into the data.

### 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.

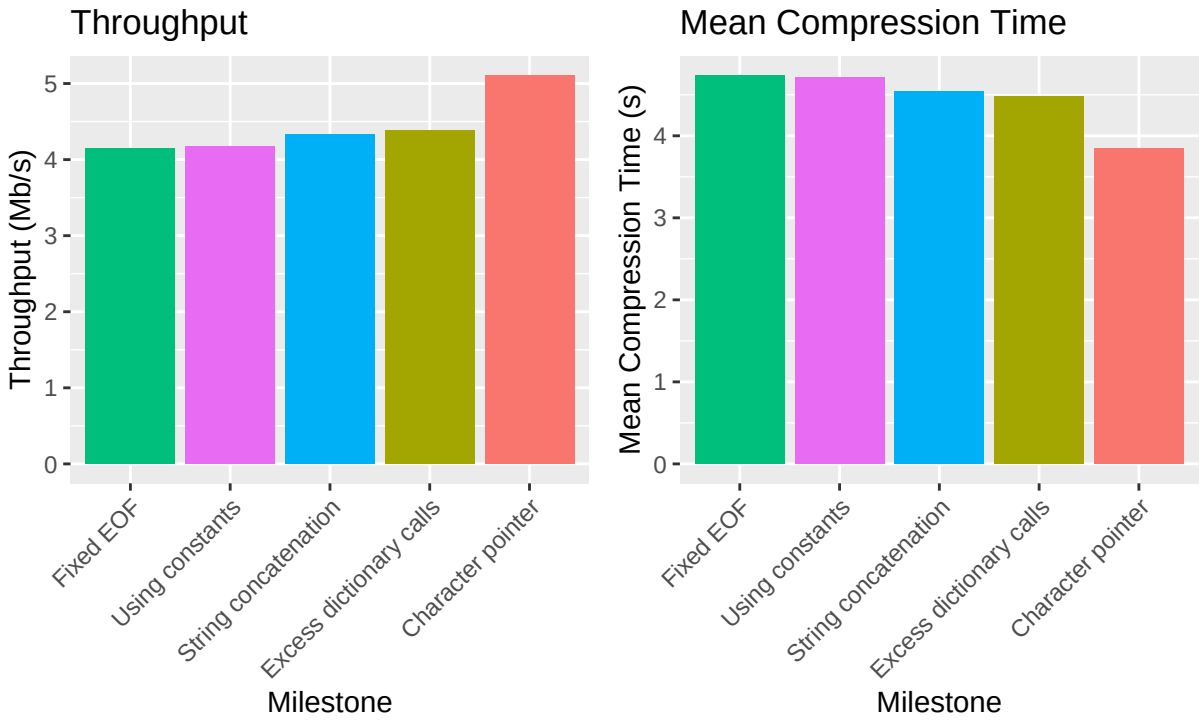


Figure 2.1: Comparison of the performance of the different milestones.

Figure 2.1 shows the performance of the algorithm at the different milestones mentioned in this section. The bars represent distinct versions of the algorithm, and they are listed chronologically from left to right. 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 inserting 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 was 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.

Table 2.3: Run statistics in Corpus 1

| Average Run Length | Maximum Run Length | Median Run Length | Sd Run Length |
|---|---|---|---|
| 6.135398 | 17 | 6 | 1.237217 |

Table 2.4: Run statistics in Corpus 2

| Average Run Length | Maximum Run Length | Median Run Length | Sd Run Length |
|---|---|---|---|
| 10.96825 | 190 | 11 | 2.494305 |

Tables 2.3 and 2.4 show stats on the run lengths of the "runs" of data, where a run is a string added to the dictionary during the execution of LZW. So if there is a run of length 8, that means we are replacing 8 characters in the original text with a codeword. A histogram of runs from both corpora can be seen in Figure 2.2.

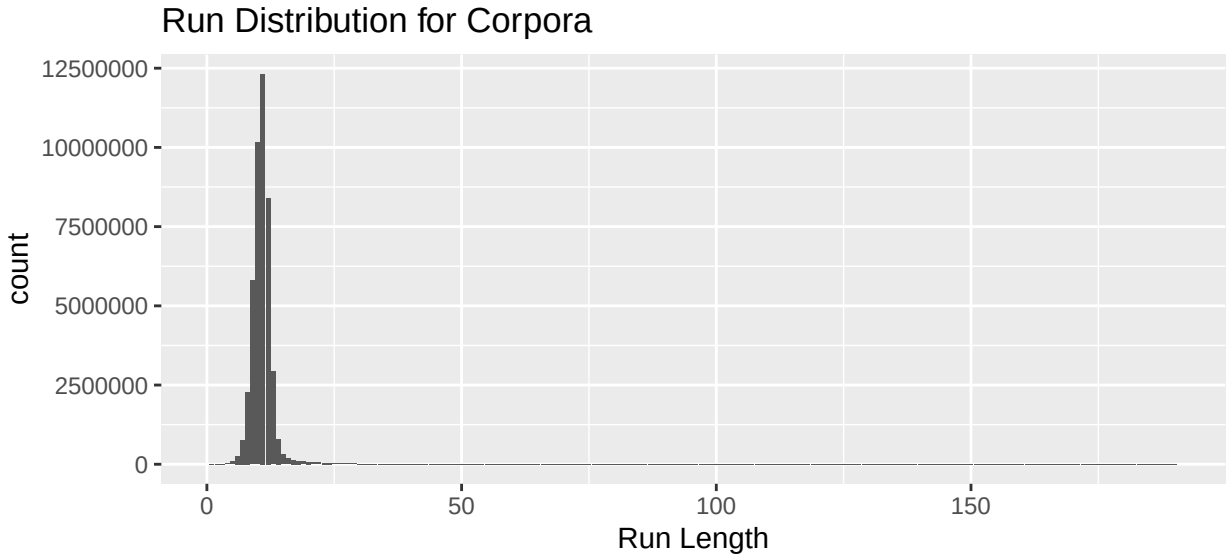


Figure 2.2: A histogram showing the lengths of runs for both copora.

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 slow dictionary implementation like the `std::unordered_map` in C++, we can try to map the strings directly into memory.

Our first realization was that since all of the strings only contain four characters ('A', 'C', 'T', and 'G'), we can represent the characters with two bits. We can assign 'A' to be `00`, 'C' to be `01`, and so on. So for any string of length `n`, we can uniquely represent that string with `2n` bits.

Assume for each string size 1 to n, we have an array with enough slots for every possible string of that length. 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 single `std::unordered_map` on top to store those longer strings. Since we are storing all these codewords in memory, it also makes sense to use shorter codewords, such as 16 bit unsigned integers rather than the 32 bit codewords we were using in previous implementations.

We call this data structure the Direct Map Dictionary. Different numbers `n` may provide different performance results.

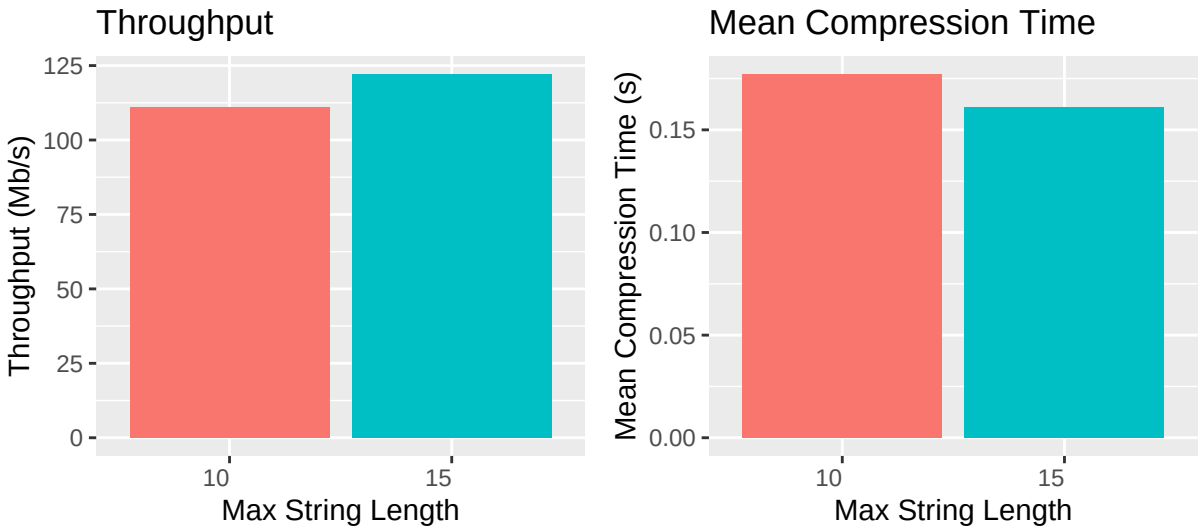


Figure 2.3: Comparing different max string lengths for the Direct Map Dictionary.

As seen in Figure 2.3, having a max string length of 15 rather than 10 leads to an increase in throughput and a decrease in mean compression time. This makes sense, because any string over the max requires an entry into the `std::unordered_map`, which takes much more time than a Direct Map Dictionary access. Ideally, we would have this Direct Map data structure support strings of longer lengths than 15. However, supporting string lengths of over 15 is difficult because the amount of memory required increases exponentially. The number of possible strings is quadrupled as the length of the string increases by 1, and our computers are limited in the amount of data that can be stored in memory.

Since we are limiting the size of codewords, we will run out of codewords faster than before. Thus, the compression ratio may be different. As seen in Table 2.5, using the Direct Map Dictionary decreases the compression ratio on every input file relative to the Standard Dictionary Implementation. This is a trade off between compression time and compression ratio. Our hope is that the Direct Map version will be much faster, and we will be able to make up some of the compression ratio lost by using entropy encoding.

It's also worth noting that at this point, we are outputting codewords followed by characters. If we output codewords followed by two bits representing the next character, we will save 6 bits per iteration. However, the relative differences in compression ratio between the Direct Map and Std Dictionaries would not change.

Table 2.5: Comparison of Compression Ratios between Direct Map and Standard Dictionaries

| | | Compression Ratio | |
|---|---|---|---|
| File Name | Original File Size | Direct Map | Standard Dictionary |
| DNACorpus1/chmpxx | 121024 | 2.449 | 2.797 |
| DNACorpus1/chntxx | 155844 | 2.402 | 2.683 |
| DNACorpus1/hehcmv | 229354 | 2.460 | 2.689 |
| DNACorpus1/humdyst | 38770 | 2.103 | 2.565 |
| DNACorpus1/humghcs | 66495 | 2.227 | 2.625 |
| DNACorpus1/humhbb | 73308 | 2.243 | 2.626 |
| DNACorpus1/humhdab | 58864 | 2.200 | 2.618 |
| DNACorpus1/humprtb | 56737 | 2.192 | 2.616 |
| DNACorpus1/mpomtcg | 186609 | 2.414 | 2.666 |
| DNACorpus1/mtpacga | 100314 | 2.377 | 2.737 |
| DNACorpus1/vaccg | 191737 | 2.486 | 2.746 |
| DNACorpus2/AeCa | 1591049 | 2.833 | 2.861 |
| DNACorpus2/AgPh | 43970 | 2.105 | 2.548 |
| DNACorpus2/BuEb | 18940 | 1.914 | 2.447 |
| DNACorpus2/DaRe | 62565020 | 2.954 | 3.194 |
| DNACorpus2/DrMe | 32181429 | 2.886 | 3.031 |
| DNACorpus2/EnIn | 26403087 | 2.938 | 3.067 |
| DNACorpus2/EsCo | 4641652 | 2.863 | 2.914 |
| DNACorpus2/GaGa | 148532294 | 2.832 | 3.170 |
| DNACorpus2/HaHi | 3890005 | 2.937 | 2.978 |
| DNACorpus2/HePy | 1667825 | 2.913 | 2.943 |
| DNACorpus2/HoSa | 189752667 | 2.966 | 3.317 |
| DNACorpus2/OrSa | 43262523 | 2.882 | 3.058 |
| DNACorpus2/PlFa | 8986712 | 3.024 | 3.104 |
| DNACorpus2/ScPo | 10652155 | 2.885 | 2.967 |
| DNACorpus2/YeMi | 73689 | 2.325 | 2.727 |

### 2.4.2 Multiple Standard Dictionaries

Similar to the Direct Mapped approach, we can use an individual `std::unordered_map` for each string size up to a certain size `n` , and for all strings of length greater than `n`, we use a single `std::unordered_map`. 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 decrease 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 `std::unordered_map`. The hash

function, which is what maps keys to their values (i.e. strings to their codewords), 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 remains the same as a single Standard Dictionary, since strings of all lengths are accommodated and we are using 32 bit codewords. This logic is supported by Figure 2.5, which shows one `std::unordered_map` compared to Multiple Standard Dictionaries.

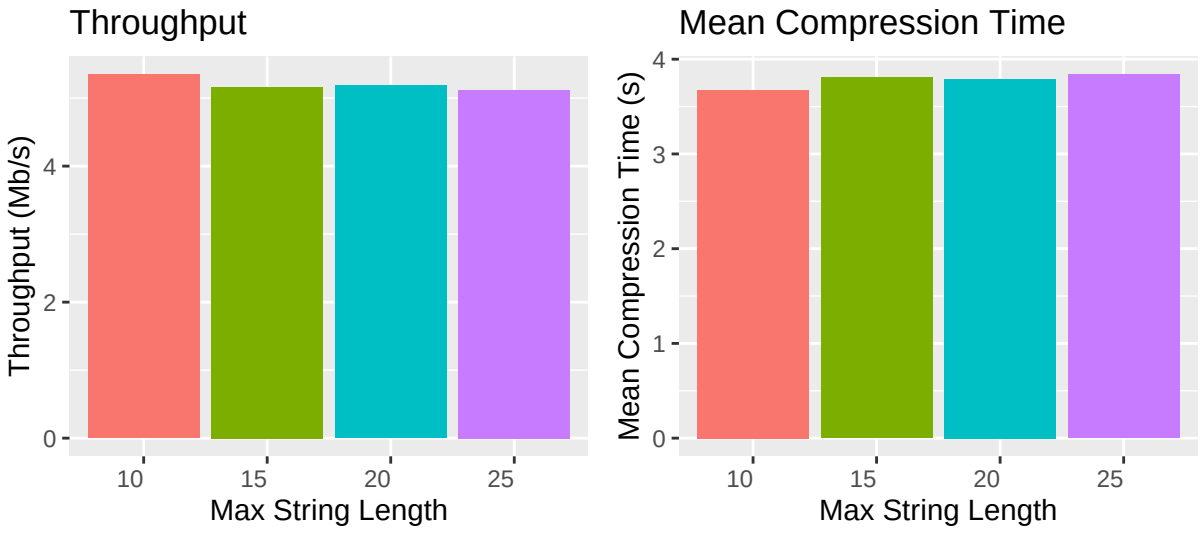


Figure 2.4: Comparing different max string lengths for Multiple Standard Dictionaries.

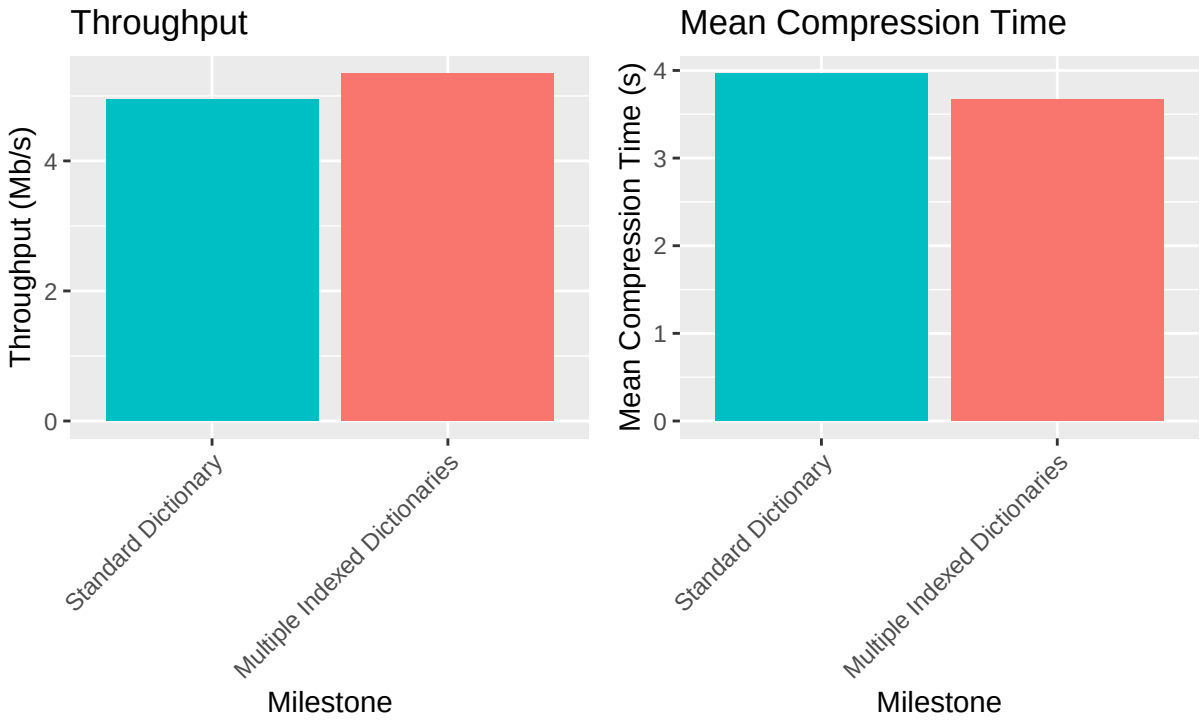


Figure 2.5: Comparing one Standard Dictionary to Multiple Standard Dictionaries (with a max string length of 10).

### 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.

Again, taking the Direct Map approach means decreasing compression ratio overall. The hope is that we will be able to use entropy encoding on the output of the Direct Map version of LZW to make up for that loss in compression ratio.

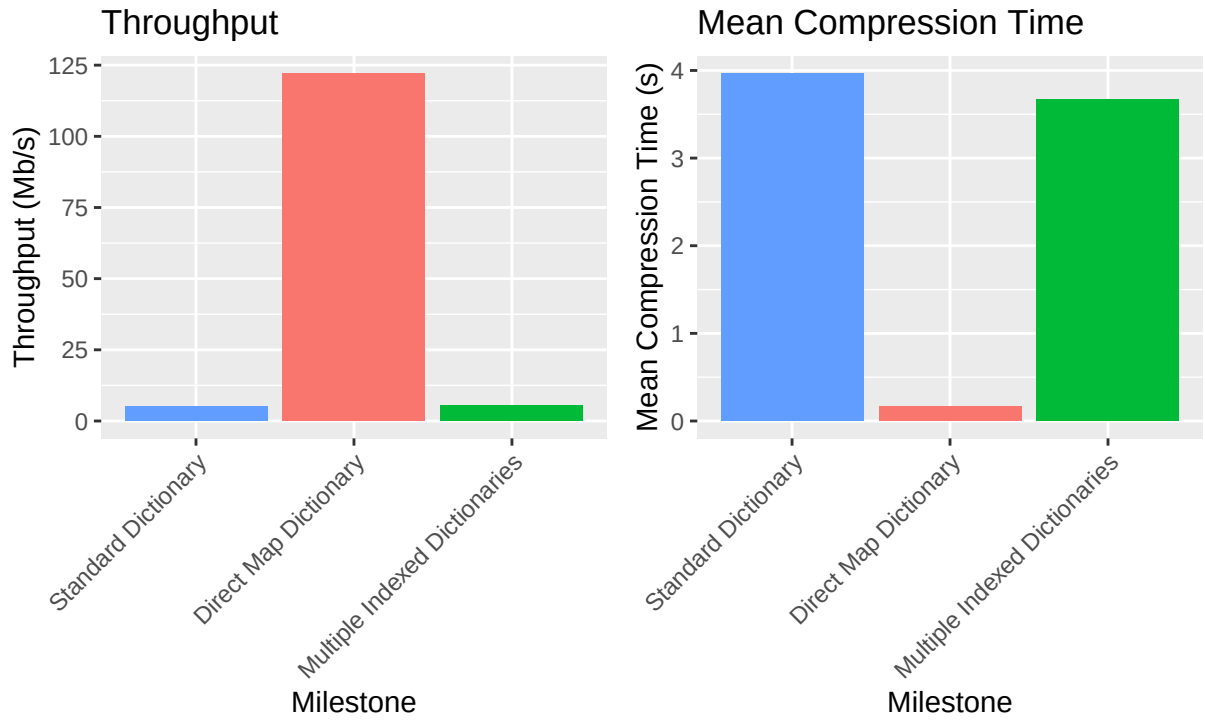


Figure 2.6: Comparing the three types of dictionaries. The Direct Map has a max length of 15 and the Mult Dict has a max length of 10.

## 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 looping up from zero.

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. Keep in mind that in

order to look up a string in a Direct Map Dictionary, you need to convert the string to it's two bit representation, which we call an index. 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), meaning most times, we don't have to search through all the values to find the one we are looking for. This 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.

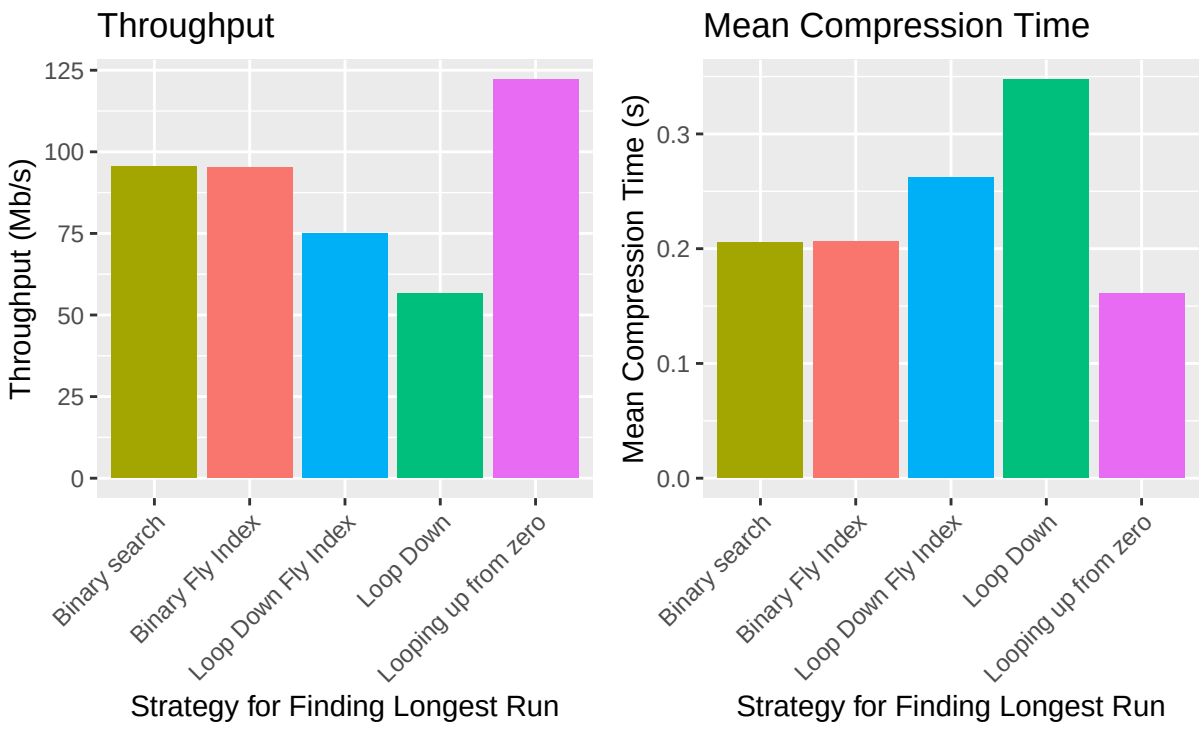


Figure 2.7: Comparing the different ways of finding the longest run.

We see that calculating the index on the fly does tend to 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.

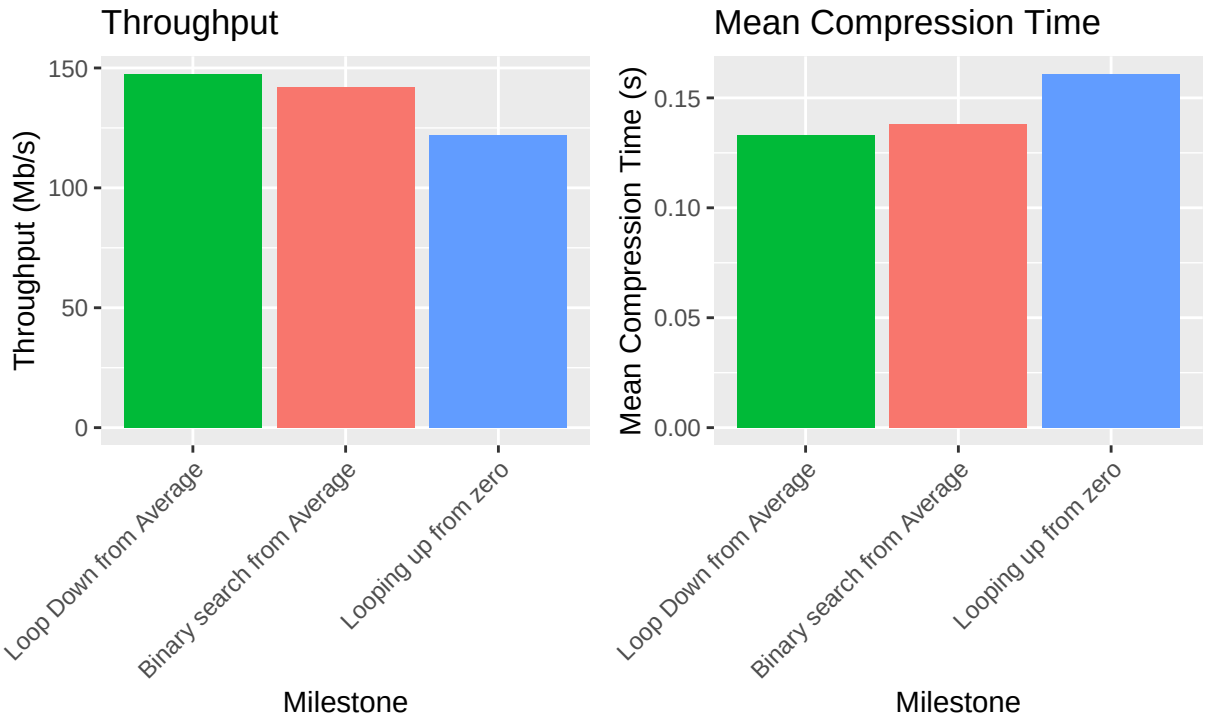


Figure 2.8: Comparing the different ways of finding the longest run starting from the average.

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 `std::unordered_map` on top of our Direct Map 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 is a slight performance improvement, which makes sense because long runs are very rare.

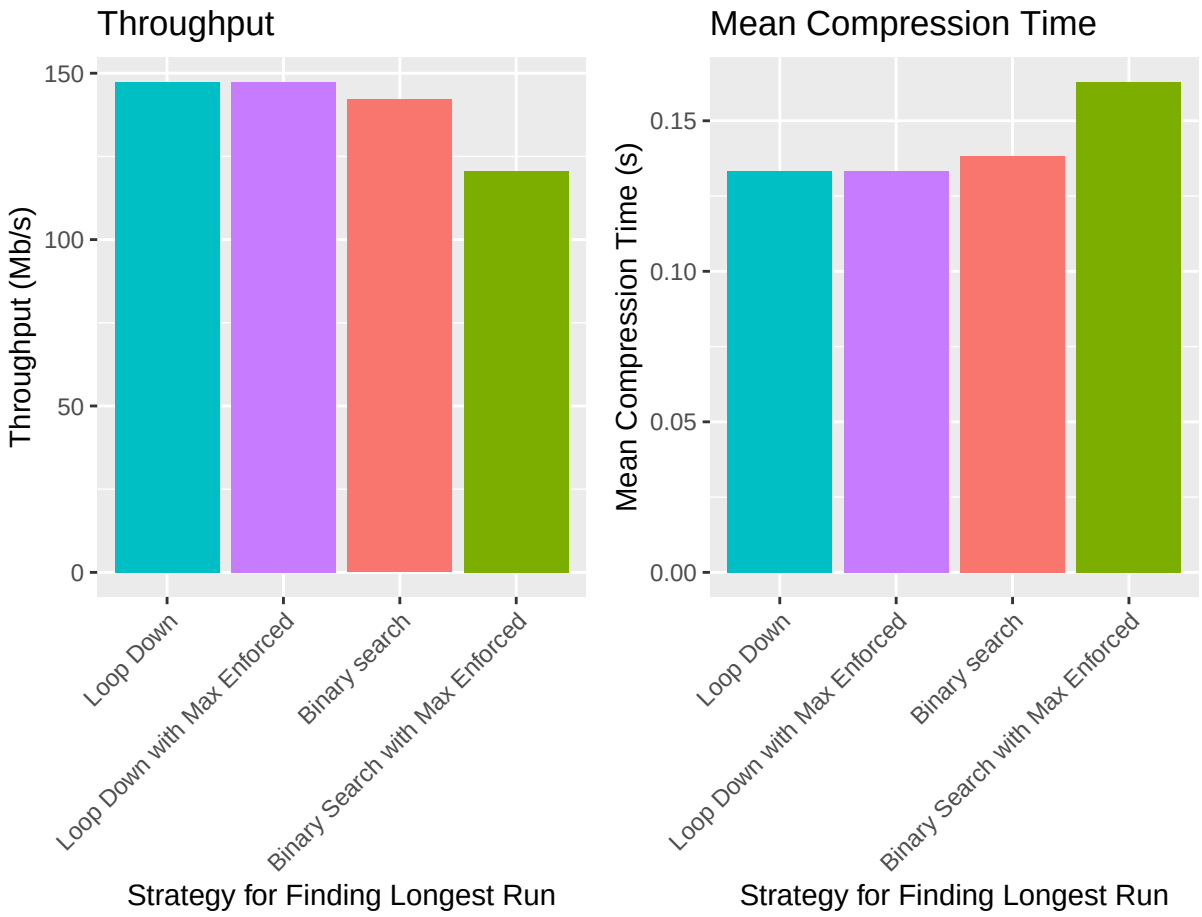


Figure 2.9: Comparing the different ways of finding the longest run starting at average with strings over max (15) not accepted.

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 represented in Table 2.5. Table 2.6 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. This difference in compression ratio is very slight, but remember that we already took a decrease in compression ratio by selecting the Direct Map Dictionary over the Standard Dictionary. Given the improvement in performance, we will now be not allowing strings over the max length in the Direct Map Dictionary unless stated

Table 2.6: Compression Ratio change from disallowing long strings

| CR Before | CR After |
|---|---|
| 2.909175 | 2.907686 |

otherwise.

### 2.5.4 Using `pext`

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

We could use `pext`, which is a command that extracts bits in parallel, meaning at the same time. The `pext` instruction is a recent addition to the x86 instruction set, so it has not yet 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 did get lucky in that the 6th and

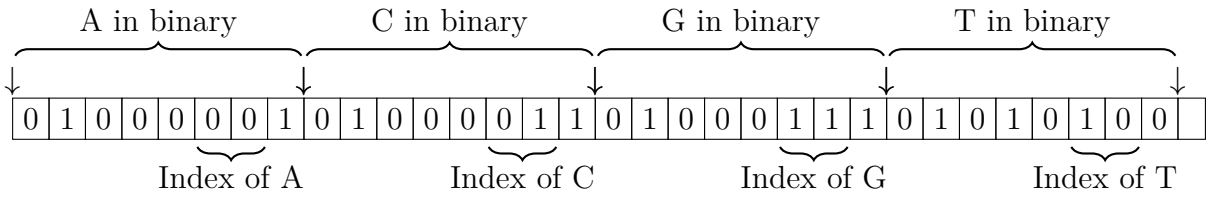


Figure 2.10: How 'pext' extracts bits

7th bit (from left to right) on A, C, T, and G give a unique 2 bit values. This means we can just extract the 6th and 7th bit from every character without knowing what it is.

We can apply this technique to our `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.This means that if we precompute the index with `pext`, a lookup in the Direct Map Dictionary can theoretically happen in constant time. In other words, the actual dictionary queries are almost instantaneous. 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.

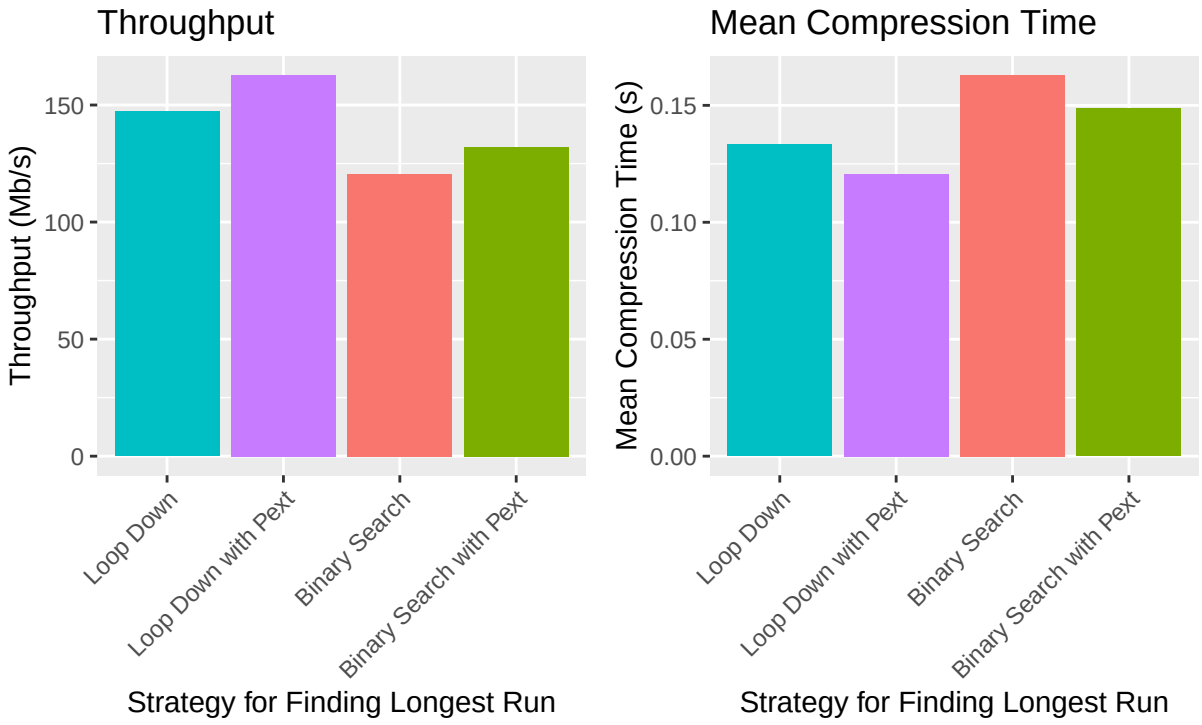


Figure 2.11: Comparing the different ways of finding the longest run with pext.

## 2.6 Returning to Compression Ratio

After spending a majority of the time optimizing, we returned our attention to the compression ratio. Most of our optimizations for the Direct Map Dictionary didn't have much of an effect on compression ratio, with the exception of disallowing strings over a certain length. We did note that the Direct Map already has a lower compression ratio than the Standard Dictionary implementation due to a smaller codeword size. In Table 2.7, we can see a comparison in compression ratios of the final implementations of the Direct Map and Standard Dictionaries. Note that the Multiple Standard Dictionaries version has the same compression ratios as a single Standard Dictionary.

### 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. Since

Table 2.7: Comparing compression ratios of the two Dictionary versions.

| | | Compression Ratio | |
|---|---|---|---|
| File Name | Original File Size | Direct Map Dictionary | Std Dictionary |
| DNACorpus1/chmpxx | 121024 | 3.266 | 3.915 |
| DNACorpus1/chntxx | 155844 | 3.203 | 3.722 |
| DNACorpus1/hehcmv | 229354 | 3.279 | 3.701 |
| DNACorpus1/humdyst | 38770 | 2.804 | 3.690 |
| DNACorpus1/humghcs | 66495 | 2.969 | 3.721 |
| DNACorpus1/humhbb | 73308 | 2.991 | 3.712 |
| DNACorpus1/humhdab | 58864 | 2.934 | 3.727 |
| DNACorpus1/humprtb | 56737 | 2.923 | 3.729 |
| DNACorpus1/mpomtcg | 186609 | 3.218 | 3.682 |
| DNACorpus1/mtpacga | 100314 | 3.169 | 3.844 |
| DNACorpus1/vaccg | 191737 | 3.315 | 3.794 |
| DNACorpus2/AeCa | 1591049 | 3.778 | 3.786 |
| DNACorpus2/AgPh | 43970 | 2.807 | 3.653 |
| DNACorpus2/BuEb | 18940 | 2.552 | 3.595 |
| DNACorpus2/DaRe | 62565020 | 3.928 | 4.013 |
| DNACorpus2/DrMe | 32181429 | 3.847 | 3.836 |
| DNACorpus2/EnIn | 26403087 | 3.918 | 3.893 |
| DNACorpus2/EsCo | 4641652 | 3.817 | 3.791 |
| DNACorpus2/GaGa | 148532294 | 3.776 | 3.943 |
| DNACorpus2/HaHi | 3890005 | 3.916 | 3.884 |
| DNACorpus2/HePy | 1667825 | 3.884 | 3.894 |
| DNACorpus2/HoSa | 189752667 | 3.953 | 4.118 |
| DNACorpus2/OrSa | 43262523 | 3.843 | 3.858 |
| DNACorpus2/PlFa | 8986712 | 4.031 | 4.000 |
| DNACorpus2/ScPo | 10652155 | 3.847 | 3.813 |
| DNACorpus2/YeMi | 73689 | 3.099 | 3.859 |

each ASCII character takes up 1 bytes, or 8 bits, the compression ratio is always 8/2 = 4.0. This is a simple conversion, and it requires no dictionaries or codewords.

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 preferable every time.

We call this method Four to One encoding. We implemented this in C++ 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 encoding method 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 repetitiveness 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 separate 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 nucleotides show up with roughly the same frequency, so having 2 bits per character is hard to beat. So the file with all the characters in it can't be compressed further.

What about the codewords? Well, a key realization is that any codeword will only show up an average of 4 times in a single run of the program. Say we add the codeword `120="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 `120G`. 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 Direct Map LZW, we need to alter our approach. The issue with our dataset is that long runs are rare. Every once in a while, there may be an opportunity to replace a run of 15 with a codeword, but most of the time the runs being replaced are of length 8 or less. Eight characters can be represented by 16 bits via the 2 bit encodings, so any run under 8 that is replaced actually increases the total number of bits in the output file.

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 next 8 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. A greedy algorithm is one that, when given a choice, it chooses the path that is most advantageous to the overall goal. This new algorithm 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 in the output file.

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. We call this new scheme Three Stream LZW, and its performance is evaluated in the next chapter.

# Chapter 3

# Comparison to other tools

Now that we have implemented several versions of LZW, we will 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 Four to One implementation and other professional compression tools.

## 3.1 Comparison of our Implementations

We have five different implementations at this point: LZW with Direct Map Dictionary, LZW with the Standard Dictionary, LZW with Multiple Standard Dictionaries, Three Stream LZW, and the Four To One translation. Figure 3.1 shows the performance of these methods on the corpora.

We can also see the raw values in Table 3.1. While the Four to One implementation completely dominates in terms of throughput, the Direct Map version is substantially faster than the Standard and Multiple Dictionary versions. The Three Stream LZW comes in between the Direct Map and Four to One implementations in terms of compression time, but it actually has the best compression ratio out of all LZW versions on most of the input files. This means that by switching from the Standard Dictionary to the Direct Map and adding on entropy encoding, we were able to regain the lost ground in terms of compression ratio while not totally compromising our performance improvements.

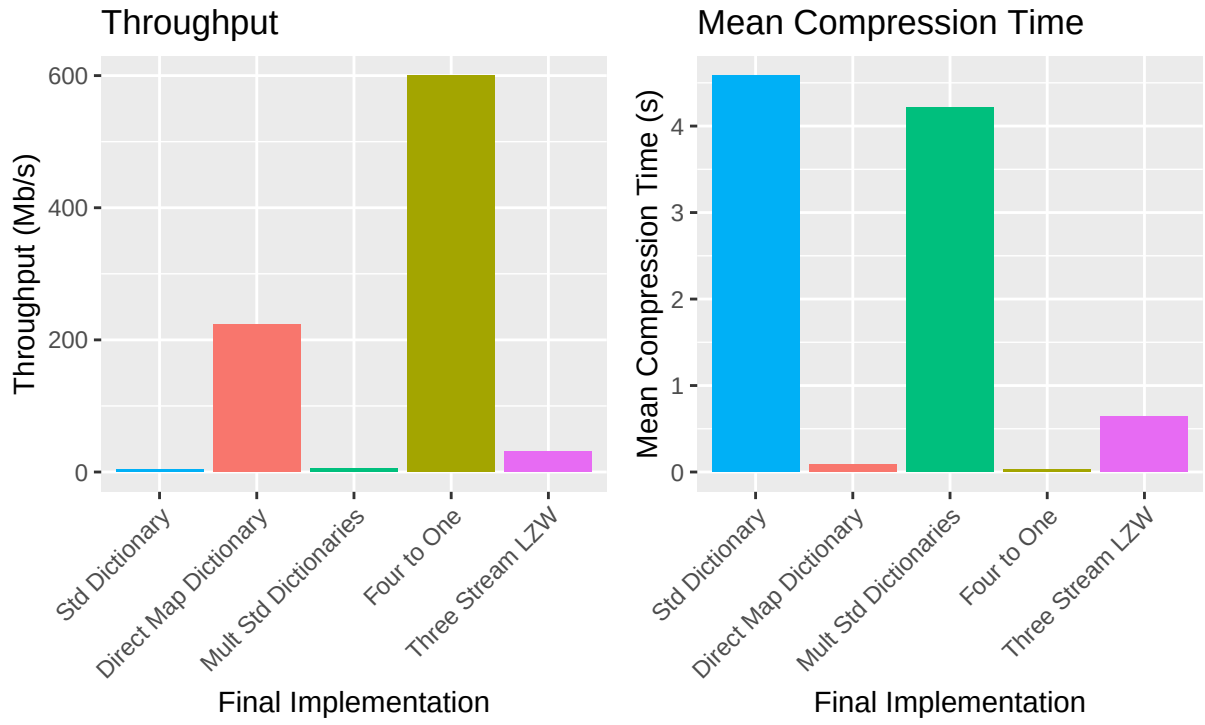


Figure 3.1: Performance comparison of our final implementations.

Table 3.1: Performance metrics for our final implementations.

| | | Compression Ratio | | | | |
|---|---|---|---|---|---|---|
| File Name | Original File Size | Direct Map Dictionary | Four to One | Mult Std Dictionaries | Std Dictionary | Three Stream LZW |
| DNACorpus1/chmpxx | 121024 | 3.266 | 3.999 | 3.915 | 3.915 | 3.885 |
| DNACorpus1/chntxx | 155844 | 3.203 | 3.999 | 3.722 | 3.722 | 3.888 |
| DNACorpus1/hehcmv | 229354 | 3.279 | 3.999 | 3.701 | 3.701 | 3.885 |
| DNACorpus1/humdyst | 38770 | 2.804 | 3.997 | 3.690 | 3.690 | 3.954 |
| DNACorpus1/humghcs | 66495 | 2.969 | 3.998 | 3.721 | 3.721 | 3.933 |
| DNACorpus1/humhbb | 73308 | 2.991 | 3.998 | 3.712 | 3.712 | 3.935 |
| DNACorpus1/humhdab | 58864 | 2.934 | 3.998 | 3.727 | 3.727 | 3.938 |
| DNACorpus1/humprtb | 56737 | 2.923 | 3.998 | 3.729 | 3.729 | 3.927 |
| DNACorpus1/mpomtcg | 186609 | 3.218 | 3.999 | 3.682 | 3.682 | 3.896 |
| DNACorpus1/mtpacga | 100314 | 3.169 | 3.999 | 3.844 | 3.844 | 3.875 |
| DNACorpus1/vaccg | 191737 | 3.315 | 3.999 | 3.794 | 3.794 | 3.874 |
| DNACorpus2/AeCa | 1591049 | 3.778 | 4.000 | 3.786 | 3.786 | 3.874 |
| DNACorpus2/AgPh | 43970 | 2.807 | 3.997 | 3.653 | 3.653 | 3.957 |
| DNACorpus2/BuEb | 18940 | 2.552 | 3.993 | 3.595 | 3.595 | 3.964 |
| DNACorpus2/DaRe | 62565020 | 3.928 | 4.000 | 4.013 | 4.013 | 4.035 |
| DNACorpus2/DrMe | 32181429 | 3.847 | 4.000 | 3.836 | 3.836 | 3.865 |
| DNACorpus2/EnIn | 26403087 | 3.918 | 4.000 | 3.893 | 3.893 | 3.928 |
| DNACorpus2/EsCo | 4641652 | 3.817 | 4.000 | 3.791 | 3.791 | 3.847 |
| DNACorpus2/GaGa | 148532294 | 3.776 | 4.000 | 3.943 | 3.943 | 3.897 |
| DNACorpus2/HaHi | 3890005 | 3.916 | 4.000 | 3.884 | 3.884 | 3.944 |
| DNACorpus2/HePy | 1667825 | 3.884 | 4.000 | 3.894 | 3.894 | 3.942 |
| DNACorpus2/HoSa | 189752667 | 3.953 | 4.000 | 4.118 | 4.118 | 4.075 |
| DNACorpus2/OrSa | 43262523 | 3.843 | 4.000 | 3.858 | 3.858 | 3.908 |
| DNACorpus2/PlFa | 8986712 | 4.031 | 4.000 | 4.000 | 4.000 | 4.010 |
| DNACorpus2/ScPo | 10652155 | 3.847 | 4.000 | 3.813 | 3.813 | 3.851 |
| DNACorpus2/YeMi | 73689 | 3.099 | 3.998 | 3.859 | 3.859 | 3.881 |

It is also worth noting that until now, we have not had much discussion about decompression time. That is because it is usually not as interesting as compression

time; for instance, with LZW implementations, we are just looking up codewords in the dictionary to their corresponding string, tacking another character onto that string, and outputting the result. This tends to be much faster than having to look for the longest run, thus decompression is almost always faster than compression when it comes to LZW. Nevertheless, we can still look at the decompression times for our implementation. As seen in Figure 3.2, the ordering of average decompression time is mostly unsurprising. The Multiple Standard Dictionaries and the single Standard Dictionaries take the most time, with Three Stream LZW coming in second. The surprising part is that Direct Map LZW is slightly faster than Four to One in decompression time. Four to One is also the only implementation which had a larger average decompression time than average compression time. The reason for this is that the decompression implementation for Four to One is not the fastest possible: right now it just reads the bytes one by one and translates the 2 bit encodings back into full characters. A faster implementation would start with a pre-loaded dictionary of all possible byte values mapped to their corresponding 4 character string. This wasn't implemented due to time constraints, which is why Figure 3.2 doesn't look quite like you might expect.

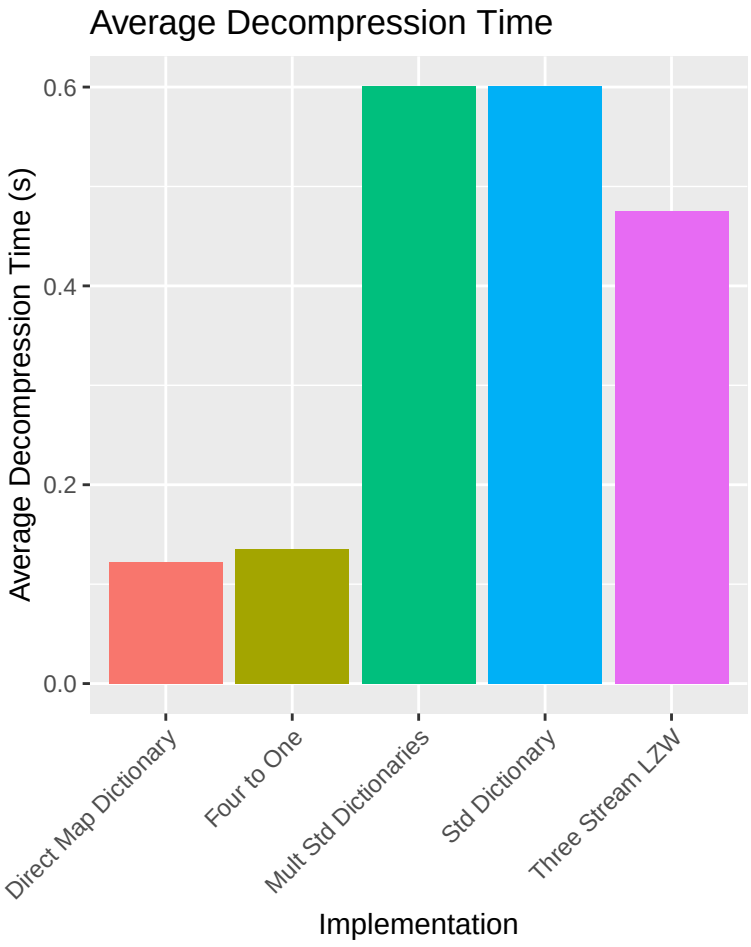


Figure 3.2: Average Decompression time of our final implementations.

## 3.2 Compression Algorithms in Literature

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

Note that I wasn't able to get compression times for these algorithms, only the compression ratios (Cao et al., 2007). As you can see, the compression ratio for these other algorithms are much better. This makes sense, because these algorithms were specifically created with DNA in mind, while we started from an algorithm, LZW, which was created for the redundancies present in human text. Our algorithms also tend to do better on longer texts, which puts them at a disadvantage because Corpus 1 has mostly smaller files. In fact, the average size of the files in Corpus 1 is surprisingly

Table 3.2: Compression ratios of related works on DNACorpus 1, reported as bits per base

| | Bits per Base | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Name | BioCompress2 | GenCompress | XM | Standard_Dictionary | Direct_Map_Dictionary | Mult_Std_Dictionaries | Four_to_One | Three_Stream_LZW |
| DNACorpus1/chmpxx | 1.6848 | 1.6730 | 1.6577 | 2.043628 | 2.449762 | 2.043628 | 2.000529 | 2.059162 |
| DNACorpus1/chntxx | 1.6172 | 1.6146 | 1.6068 | 2.149329 | 2.497831 | 2.149329 | 2.000411 | 2.057801 |
| DNACorpus1/hehcmv | 1.8480 | 1.8470 | 1.8426 | 2.161480 | 2.439513 | 2.161480 | 2.000297 | 2.059280 |
| DNACorpus1/humdyst | 1.9262 | 1.9231 | 1.9031 | 2.168274 | 2.853547 | 2.168274 | 2.001754 | 2.023420 |
| DNACorpus1/humghcs | 1.3074 | 1.0969 | 0.9828 | 2.150056 | 2.694819 | 2.150056 | 2.000993 | 2.034318 |
| DNACorpus1/humhbb | 1.8800 | 1.8204 | 1.7513 | 2.155181 | 2.674960 | 2.155181 | 2.000873 | 2.032848 |
| DNACorpus1/humhdab | 1.8770 | 1.8192 | 1.6671 | 2.146779 | 2.726828 | 2.146779 | 2.001087 | 2.031394 |
| DNACorpus1/humprtb | 1.9066 | 1.8466 | 1.7361 | 2.145619 | 2.737261 | 2.145619 | 2.001234 | 2.037048 |
| DNACorpus1/mpomtcg | 1.9378 | 1.9058 | 1.8768 | 2.172671 | 2.485925 | 2.172671 | 2.000375 | 2.053449 |
| DNACorpus1/mtpacga | 1.8752 | 1.8624 | 1.8447 | 2.081225 | 2.524314 | 2.081225 | 2.000678 | 2.064557 |
| DNACorpus1/vaccg | 1.7614 | 1.7614 | 1.7649 | 2.108805 | 2.413347 | 2.108805 | 2.000365 | 2.065162 |

small for the number of papers that have used them. If you had files that are only about 100 thousand bytes, you don't really need to compress them at all. For smaller files, the Four to One version or even Direct Map LZW version may be preferable because of how fast they are. These other papers do not provide performance metrics like compression time, but we believe that our implementations would be much faster on longer files.

## 3.3 Comparison to Other Professional Tools

The other tools that we choose to use are `gzip`, `bzip`, `xz`, and `genozip`. All three of `xz`, `bzip`, and `gzip` are open source, general compression tools. As mentioned earlier `genozip` was created specifically for DNA compression and the compression of other common filetypes in biological research (Lan et al., 2021). Table 3.3 summarizes the compression ratios of both the other tools and our 5 implementations. Not surprisingly, all of our implementations do better than general compressors `bzip` and `gzip` in terms of compression ratio on the larger files. Both of these tools were created to work on a broad array of files, and thus they do not take into account the specific characteristics of genetic sequences that we saw such as there only being four characters. On the other hand, `genozip` does outperform our implementation in terms of compression ratio on most files, which also makes sense as it was also created with DNA in mind. It is also worth noting that `xz` does surprisingly well, beating even `genozip` on some of the larger files. However, if you look at the performance data in Figure 3.3, you can see that both our Three Stream LZW and Direct Map LZW are significantly faster than all these other tools, including `genozip` and `xz`. In fact, `xz` is over 10 times slower than Three Stream LZW in terms of average compression time. So for large files, you if you want a balance between compression ratio and compression time, Three Streams may be a viable option. Of course, if you are looking for

Table 3.3: Performance metrics for other professional tools. Xz, bzip, and gzip were ran with option -9, and genozip was ran with –input=generic.

| | | Compression Ratio | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| File Name | Original File Size | bzip | Direct Map Dictionary | Four to One | genozip | gzip | Mult Std Dictionaries | Std Dictionary | Three Stream LZW | xz |
| DNACorpus1/chmpxx | 121024 | 3.77 | 3.266 | 3.999 | 3.95 | 3.60 | 3.915 | 3.915 | 3.885 | 3.84 |
| DNACorpus1/chntxx | 155844 | 3.66 | 3.203 | 3.999 | 3.85 | 3.49 | 3.722 | 3.722 | 3.888 | 3.70 |
| DNACorpus1/hehcmv | 229354 | 3.68 | 3.279 | 3.999 | 3.86 | 3.50 | 3.701 | 3.701 | 3.885 | 3.70 |
| DNACorpus1/humdyst | 38770 | 3.66 | 2.804 | 3.997 | 3.24 | 3.36 | 3.690 | 3.690 | 3.954 | 3.60 |
| DNACorpus1/humghcs | 66495 | 4.62 | 2.969 | 3.998 | 5.56 | 5.15 | 3.721 | 3.721 | 3.933 | 7.04 |
| DNACorpus1/humhbb | 73308 | 3.72 | 2.991 | 3.998 | 3.61 | 3.58 | 3.712 | 3.712 | 3.935 | 3.90 |
| DNACorpus1/humhdab | 58864 | 3.86 | 2.934 | 3.998 | 3.56 | 3.61 | 3.727 | 3.727 | 3.938 | 3.97 |
| DNACorpus1/humprtb | 56737 | 3.81 | 2.923 | 3.998 | 3.52 | 3.58 | 3.729 | 3.729 | 3.927 | 3.87 |
| DNACorpus1/mpomtcg | 186609 | 3.68 | 3.218 | 3.999 | 3.84 | 3.50 | 3.682 | 3.682 | 3.896 | 3.76 |
| DNACorpus1/mtpacga | 100314 | 3.76 | 3.169 | 3.999 | 3.84 | 3.58 | 3.844 | 3.844 | 3.875 | 3.82 |
| DNACorpus1/vaccg | 191737 | 3.81 | 3.315 | 3.999 | 3.94 | 3.65 | 3.794 | 3.794 | 3.874 | 3.88 |
| DNACorpus2/AeCa | 1591049 | 3.71 | 3.778 | 4.000 | 4.00 | 3.57 | 3.786 | 3.786 | 3.874 | 3.84 |
| DNACorpus2/AgPh | 43970 | 3.64 | 2.807 | 3.997 | 3.28 | 3.36 | 3.653 | 3.653 | 3.957 | 3.59 |
| DNACorpus2/BuEb | 18940 | 3.62 | 2.552 | 3.993 | 2.59 | 3.25 | 3.595 | 3.595 | 3.964 | 3.45 |
| DNACorpus2/DaRe | 62565020 | 3.88 | 3.928 | 4.000 | 4.39 | 3.76 | 4.013 | 4.013 | 4.035 | 4.99 |
| DNACorpus2/DrMe | 32181429 | 3.70 | 3.847 | 4.000 | 4.03 | 3.60 | 3.836 | 3.836 | 3.865 | 4.01 |
| DNACorpus2/EnIn | 26403087 | 3.72 | 3.918 | 4.000 | 4.14 | 3.61 | 3.893 | 3.893 | 3.928 | 4.56 |
| DNACorpus2/EsCo | 4641652 | 3.70 | 3.817 | 4.000 | 4.02 | 3.57 | 3.791 | 3.791 | 3.847 | 3.91 |
| DNACorpus2/GaGa | 148532294 | 3.74 | 3.776 | 4.000 | 4.15 | 3.65 | 3.943 | 3.943 | 3.897 | 4.10 |
| DNACorpus2/HaHi | 3890005 | 3.77 | 3.916 | 4.000 | 4.14 | 3.63 | 3.884 | 3.884 | 3.944 | 3.94 |
| DNACorpus2/HePy | 1667825 | 3.77 | 3.884 | 4.000 | 4.14 | 3.67 | 3.894 | 3.894 | 3.942 | 4.01 |
| DNACorpus2/HoSa | 189752667 | 3.89 | 3.953 | 4.000 | 4.22 | 3.73 | 4.118 | 4.118 | 4.075 | 4.48 |
| DNACorpus2/OrSa | 43262523 | 3.73 | 3.843 | 4.000 | 4.21 | 3.65 | 3.858 | 3.858 | 3.908 | 4.62 |
| DNACorpus2/PlFa | 8986712 | 3.82 | 4.031 | 4.000 | 4.28 | 3.77 | 4.000 | 4.000 | 4.010 | 4.28 |
| DNACorpus2/ScPo | 10652155 | 3.68 | 3.847 | 4.000 | 4.07 | 3.57 | 3.813 | 3.813 | 3.851 | 3.91 |
| DNACorpus2/YeMi | 73689 | 3.78 | 3.099 | 3.998 | 3.72 | 3.58 | 3.859 | 3.859 | 3.881 | 3.81 |

speed, you should use the Four to One implementation, which still blows all others out of the water.

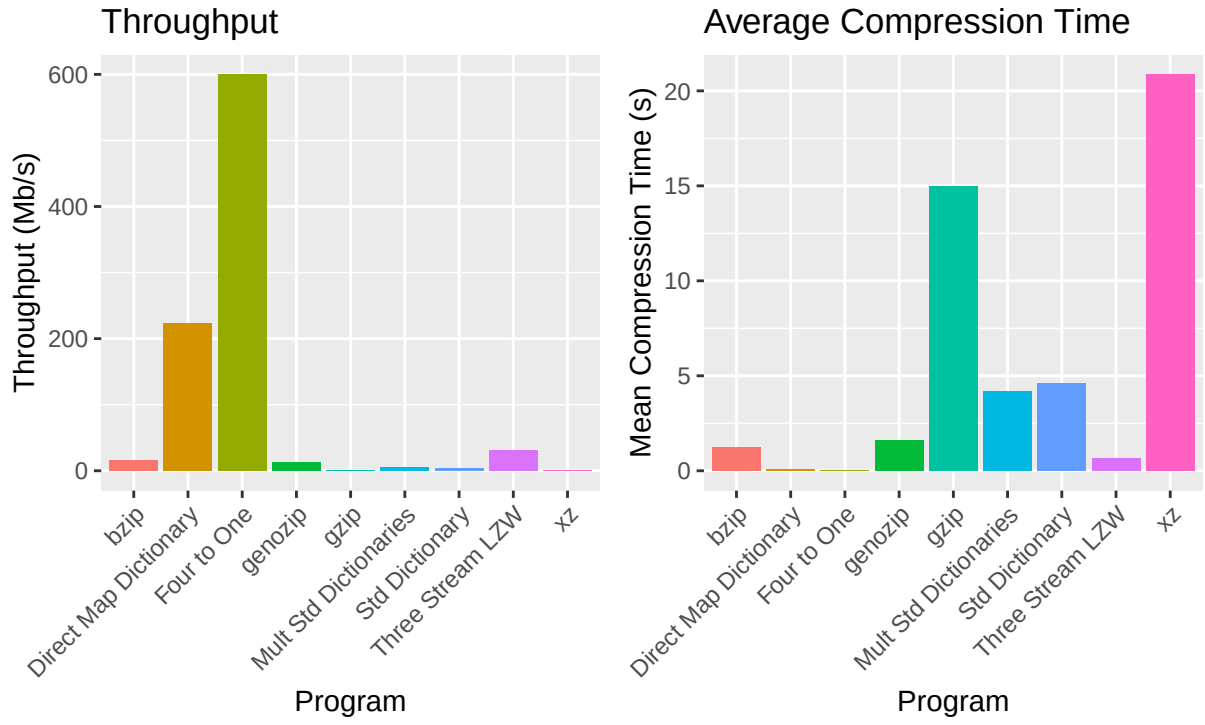


Figure 3.3: Average compression time of other methods and our final implementations.

# Conclusion

We started with a naive implementation of Lempel Ziv Welch, and our goal was to optimize it for DNA. Although we were not able to achieve as high of a compression ratio as other algorithms in literature and professional DNA compresion tools, we were able to make a very fast algorithm that can do better than a 4.0 compression ratio on large files. There is a trade off when it comes to choosing a compression algorithm, with speed and space. If you needed to compress DNA files as small as possible, you should use the professional tools or one of the algorithms proposed in other research papers. If you want to compress DNA very quickly and you are okay with a moderate compression ratio, than you should use the Four to One implementation which we discussed previously. If you want something that is in between, with a moderate compression ratio and fast compression time, the results in Chapter 3 tell us that the Three Stream implementation of LZW would work very well for large files.

The concept of an indexed, Direct Map dictionary is one that could prove useful for other applications. Further research could be used to apply a similar idea to a different algorithm, one more suited to the redundancies present in DNA. In other words, could we take one of the research papers discussed in the Related Works section and apply the idea of the Direct Map to the algorithm presented in it? This could also be useful in other areas of compression in which the workload has a limited number of characters, because the size of the dictionary grows exponentially with as the number of characters needing to be accounted for increases. The usage of `pext` is also worth noting. Because of the 2 bit encoding possible for DNA sequences, we can create an index for a string of DNA very quickly. This, combined with the Direct Map Dictionary, allows for the Direct Map Dictionary to have dictionary accesses that are almost as fast as array accesses.

The Four to One implementation, another use of `pext`, is also notable. The Four to One implementation is extremely fast, and could also be useful for other compression applications where there are fewer characters than the encoding scheme accounts for (bytes can be used to store 256 different values, but only 4 are used for DNA).

Overall, our thesis has produced a few notable implementations of LZW, created a Direct Map Dictionary data structure which has high potential for other compression applications, and shown the relevance of `pext` in optimizing compression algorithms.

# Appendix A

# Appendix: The Code

All code for this thesis, including the Rmarkdown used to generate this document, can be found at `https://github.com/cadencorontzos/lzwfordna`.

# 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`

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

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*.

Lan, D., Tobler, R., Souilmi, Y., & Llamas, B. (2021). Genozip: a universal extensible genomic data compressor. *Bioinformatics*, *37*(16), 2225–2230. http://doi.org/10.1093/bioinformatics/btab102

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

Welch, T. A. (1984). A technique for high-performance data compression. *Computer*, *17*(6), 8–19. http://doi.org/10.1109/MC.1984.1659158

Ziv, J., & Lempel, A. (1977). A universal algorithm for sequential data compression. In *IEEE transactions on information theory* (Vol. 23, pp. 337–343).

http://doi.org/10.1109/TIT.1977.1055714