Optimizing Lempel Ziv Welch for DNA Compression

A Thesis

Presented to

The Division of Mathematics and Natural Sciences

Reed College

In Partial Fulfillment

of the Requirements for the Degree

Bachelor of Arts

Caden Corontzos

May 2023

Approved for the Division
(Computer Science)

Eitan Frachtenberg

Acknowledgements

I want to thank a few people.

Preface

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

List of Abbreviations

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

Table of Contents

List of Tables

List of Figures

Abstract

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

Dedication

You can have a dedication here if you wish.

Introduction

When dealing with DNA, it

Chapter 1

Backround and Motivations

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

1.1 What is information?

Suppose you had an idea that you wanted to share with another person. Humans have many ways to communicate information; you could send a text message, you could tell them with words, you could tell them with sign language. But regardless of the medium, you have some idea that you want to get across. Does it matter if the other person gets your message exactly? Or can it be part of the message? 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 us assume that information is something that can be interpreted to glean data that you didn't know before.

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

1.2 Compression Metrics

1.2.1 Compression Ratio

Compression Ratio is the measure of size reduction achieved by a compression algorithm. It is typically expressed as a ratio of the size of the uncompressed data (OS) to the size of the compressed data (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 data in less space, allowing for easier storage and transfer.

1.2.2 Runtime

The runtime is also an important part of evaluating the effectiveness of a compression algorithm. Runtime is typically defined as the length of time a program takes to complete a task. If you have the option of two compression algorithms, one with a compression ratio of 2.0, and another with a compression ratio of 2.15 but takes twice as long as the other, you may opt for a lower compression ratio to save time.

1.2.3 Memory Usage

Memory usage is closely tied with runtime when it comes to compression algorithms. Memory generally refers to storage where information that programs track as they are running is stored. So do reduce our runtime 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.

1.3 Lossless vs. Lossy Compression

1.3.1 Lossy

Lossy compression is based on the idea that not all information is vital. For instance, when saving a picture on your computer, your computer may save it in the .jpeg format to save space. Jpegs lose some of the information in the original picture and produce an overall lower quality picture, but the general information in the picture

is preserved. Another example is MP3 audio files. MP3 compression discards some of the information and sound quality in exchange for a file that takes up less space, which is often favorable for small devices like MP3 players and cellphones.

1.3.2 Lossless

Lossless compression is the compression of data with the goal of preserving all the information in the data. 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 preserved, like scientific data, archiving (think a .zip folder), and high end audio recording.Examples of lossless compression algorithms are Huffman Encoding and Lempel Ziv Welch, which is the focus of this thesis.

1.4 Examples of Compression Using Classic Compression Algorithms

1.4.1 Run Length Encoding

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

$$AAGCTTTTTTTTGGGGGCCCT$$

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

$$2A1G1C8T5G3C1T$$

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

1.4.2 Huffman

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

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

Then, we can use the frequencies to build a tree, which will assign short codes for frequent letters and longer code for less frequent letters. So $G = 0$, $A = 10$, $T = 111$

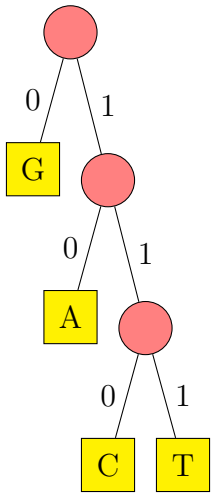


Figure 1.1: Example Huffman tree

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

So our message would be encoded to 10110100010111100110.

1.4.3 Arithmetic

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

Here is a simple example. Say we want to encode a string of characters "ACGT". Arithmetic Encoding also requires the encoder and decoder know the probabilities of each of the characters that could possibly be in the message. Let's say the probability of each symbol in the message are

- P(A) = 1/10

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

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

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

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

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

TODO: Add graphic

1.4.4 Lempel Ziv Welch

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

Here is a simple example. We may be sending messages with the characters {'A' = 0, 'C' = 1, 'T' = 2, 'G' = 3}, so I will start with those in my dictionary. Say we want to send the message

AAGGAATCC

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

AAGGAATCC

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

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"

AAGGAATCC

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

So the encoded string will look something like

0A....

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

This is nice for several reasons

- When we send this encoding to someone else, we don’t need to send a “code-

book" (our dictionary). They are able to build it up themselves as they decode.
- you only need to go over the data once to encode and decode. This means that the run time of the algorithm should roughly increase linearly with the length of the input.
- As runs get longer, we will start to see more and more repeating patterns, and replacing them with codewords will become more and more effective.

Since the algorithm loops, we need some special character at the end of our encoding to let the decoder know when the message is done.

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

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

```
LZWEncode(input):

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

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

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

        currentBlock.add(nextCharacter);

        if currentBlock and the nextCharacter is in dictionary:
            currentBlock.add(nextCharacter);

        else:
            code = dictionary.lookup(currentBlock);
            output(code);
            output(nextCharacter);
```

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

        codeword = codeword + 1;


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

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

```
LZWDecode(input):

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

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

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

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

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

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

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

- `dictionary`: a key value system. Like a real dictionary holds words and their corresponding definitions, our dictionary holds codewords and their corresponding strings of characters. In C++, this is called a `map`, but the concept is the same.
- `codeword`: a number used to take the place of a string in our encoding.

- `run`: the next run of characters in our input that are already in our dictionary. So if we are encoding "ACTG", and "A", "AC", and "ACT" are in the dictionary but "ACTG" is not, we have a run of 3.
- `EOF`: end of file, the special character that we need to output at the end of the encoding.

1.5 Related work

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

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

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

Yet another paper made use of Chinese Remainder Theorem to augment Lempel Ziv Welch (Ibrahim & Gbolagade, 2020). They saw great reduction in compression time without compromising compression ratio, although these results could not be verified. The details of their implementation were not clear from the paper. We tried multiple different methods of utilizing CRT given the pseudocode in their paper, but we could not get anything that looked like it may improve compression time. We reached out to the authors of the paper, but we were not able to further our progress on this method and thus the it is not used in this thesis.

Chapter 2

Optimizing LZW: Approach

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

2.1 Corpora

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

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

Table 2.1: Corpus 1

Name	Size.bytes.
chmpxx	121024
chntxx	155844
hehcmv	229354
humdyst	38770
humghcs	66495
humhbb	73308

Name	Size.bytes.
humhdab	58864
humprtb	56737
mpomtcg	186609
mtpacga	100314
vaccg	191737

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

Table 2.2: Corpus 2

Name	Size.bytes.
AeCa	1591049
AgPh	43970
BuEb	18940
DaRe	62565020
DrMe	32181429
EnIn	26403087
EsCo	4641652
GaGa	148532294
HaHi	3890005
HePy	1667825
HoSa	189752667
OrSa	43262523
PlFa	8986712
ScPo	10652155
YeMi	73689

This particular dataset is publicly available at this link.

2.2 Evaluating Performance

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

To attempt to counteract this, we ran the function on the same file multiple times, and took the median of the compression and decompression times for all the runs. Also, for any graphs or tables in this thesis, all versions of the algorithm for a particular table were ran on the same computer.

2.3 A Starting Point

As a starting point, we thought it was best to get a working implementation of LZW in C++ on regular text files, optimize it as much as we could, and then try variations from there, optimizing 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.

2.3.1 Growing Codewords and Bit Output

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

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

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

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

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

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

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

  int read_n_bits(int n);
}

// BitOutput: Write a single bit at a time to an output stream
// Make sure all bits are written out when exiting scope
class BitOutput {
 public:
  // Construct with an input stream
  BitOutput(std::ostream& os);

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

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

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

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

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

2.3.2 Getting EOF to work

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

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

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

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

Then use the code 256 to denote the end of file. So the algorithm goes along reading a file. It builds up a current string character by character, adding the character to the string and checking if it has seen that sequence before. Once it find the end of file, we stop and output the EOF codeword.

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

1. "ACC" was in the dictionary but "ACCT" was not. This means we can output the codeword for "ACC", follow it by the character "T", and we are done. This is the ideal scenario, because nothing is left over when we output the EOF codeword
2. "ACCT" was in the dictionary: This means we have one more codeword to output, but since we reached the end of the file, we never got to output it.

3. "AC" was in the dictionary, but "ACC" was not: in this case, we would output the codeword for "AC" output the character "C", and then start looping again starting at "T". But we reach the end of the file, so we output EOF before outputting T.

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

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

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

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

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

Table 2.3: Corpus 1 Stats after fixing EOF

File.Name	Original.File.Size	Compressed.Size	Compression.Ratio	Compression.Time	Decompression.Time
DNACorpus1/humprtb	56737	21902	2.590	7	2
DNACorpus1/humdyst	38770	15300	2.534	5	1
DNACorpus1/vaccg	191737	70067	2.736	18	7
DNACorpus1/hehcmv	229354	85526	2.682	23	9
DNACorpus1/mpomtcg	186609	70254	2.656	18	7
DNACorpus1/humhdab	58864	22699	2.593	8	2
DNACorpus1/chmpxx	121024	43516	2.781	13	4
DNACorpus1/mtpacga	100314	36862	2.721	11	4
DNACorpus1/chntxx	155844	58336	2.671	16	6
DNACorpus1/humghcs	66495	25552	2.602	7	2
DNACorpus1/humhbb	73308	28134	2.606	8	3

Table 2.4: Corpus 2 stats after fixing EOF

File.Name	Original.File.Size	Compressed.Size	Compression.Ratio	Compression.Time	Decompression.Time
DNACorpus2/YeMi	73689	27235	2.706	9	3
DNACorpus2/DaRe	62565020	19586457	3.194	15034	1898
DNACorpus2/EnIn	26403087	8609993	3.067	5427	781
DNACorpus2/HePy	1667825	566972	2.942	145	40
DNACorpus2/OrSa	43262523	14148071	3.058	9790	1350
DNACorpus2/EsCo	4641652	1593404	2.913	569	121
DNACorpus2/GaGa	148532294	46851765	3.170	40810	4773
DNACorpus2/ScPo	10652155	3590856	2.966	1761	295
DNACorpus2/HaHi	3890005	1306708	2.977	445	97
DNACorpus2/HoSa	189752667	57200209	3.317	53818	5913
DNACorpus2/AeCa	1591049	556535	2.859	141	40
DNACorpus2/DrMe	32181429	10619042	3.031	7190	1015
DNACorpus2/BuEb	18940	7893	2.400	2	0
DNACorpus2/PlFa	8986712	2895744	3.103	1367	238
DNACorpus2/AgPh	43970	17442	2.521	6	2

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 integer.

There are several tools at a developers disposal when looking to debug and optimize code. One tool used for this thesis was `callgrind` which is a tool of `valgrind`

a profiling tool. Profiling tools are used to look at how your code works, where the bottlenecks are, and what can be changed/improved for the performance of your code.

Callgrind in particular is a profiling tool which associates assembly instructions to lines of code, indicating to the programmer which lines take a lot of instructions and which take less. For those unfamiliar, assembly instructions are what code is turned into so that it can be ran on your computer's processor. In general, more instructions means that code takes longer to run.

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

The check for this in our algorithm looked like this.

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

Here is the callgrind output for that line.

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

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

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

Of course, there are several issues with this method. It is difficult to associate instructions with a single line of code. Some lines are interdependent, and assembly

often behaves differently than the code that produces it. Another thing is that compliers are very advanced, and sometimes small optimizations like this are done by the compiler automatically.

Despite these issues, this change was still worth making, if not to save time then for sake of clarity and readability of the code. Also, despite the inaccuracy of callgrind, like many profiling tools, its job is not necessarily to provide exact measurements of code performance, but to give indications to trouble spots which can be improved.

Here are the runs after this optimization.

Table 2.5: Corpus 1 Stats after changing constants

File.Name	Original.File.Size	Compressed.Size	Compression.Ratio	Compression.Time	Decompression.Time
DNACorpus1/humprtb	56737	21902	2.590	7	2
DNACorpus1/humdyst	38770	15300	2.534	5	1
DNACorpus1/vaccg	191737	70067	2.736	19	7
DNACorpus1/hehcmv	229354	85526	2.682	23	9
DNACorpus1/mpomtcg	186609	70254	2.656	19	7
DNACorpus1/humhdab	58864	22699	2.593	7	2
DNACorpus1/chmpxx	121024	43516	2.781	12	4
DNACorpus1/mtpacga	100314	36862	2.721	11	4
DNACorpus1/chntxx	155844	58336	2.671	16	6
DNACorpus1/humghcs	66495	25552	2.602	9	2
DNACorpus1/humhbb	73308	28134	2.606	10	3

Table 2.6: Corpus 2 Stats after changing constants

File.Name	Original.File.Size	Compressed.Size	Compression.Ratio	Compression.Time	Decompression.Time
DNACorpus2/YeMi	73689	27235	2.706	9	3
DNACorpus2/DaRe	62565020	19586457	3.194	14940	1880
DNACorpus2/EnIn	26403087	8609993	3.067	5401	769
DNACorpus2/HePy	1667825	566972	2.942	144	40
DNACorpus2/OrSa	43262523	14148071	3.058	9782	1344
DNACorpus2/EsCo	4641652	1593404	2.913	569	118
DNACorpus2/GaGa	148532294	46851765	3.170	41060	4787
DNACorpus2/ScPo	10652155	3590856	2.966	1738	292
DNACorpus2/HaHi	3890005	1306708	2.977	444	97
DNACorpus2/HoSa	189752667	57200209	3.317	53774	5854
DNACorpus2/AeCa	1591049	556535	2.859	139	40
DNACorpus2/DrMe	32181429	10619042	3.031	7188	1009
DNACorpus2/BuEb	18940	7893	2.400	2	1
DNACorpus2/PlFa	8986712	2895744	3.103	1352	237
DNACorpus2/AgPh	43970	17442	2.521	5	2

2.3.4 Extraneous String Concatenations

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

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

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

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

    // other code here ommitted


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

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

```
// save concatenation here
std::string string_seen_plus_new_char = current_string_seen + next_character;
if (dictionary.find(string_seen_plus_new_char) != end ){
    current_string_seen = string_seen_plus_new_char;
}
else{

// other code omitted here

    dictionary[string_seen_plus_new_char] = codeword;
}
```

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

Table 2.7: Corpus 1 Stats after reducing string concatenations

File.Name	Original.File.Size	Compressed.Size	Compression.Ratio	Compression.Time	Decompression.Time
DNACorpus1/humprtb	56737	21902	2.590	6	2
DNACorpus1/humdyst	38770	15300	2.534	4	1
DNACorpus1/vaccg	191737	70067	2.736	16	7
DNACorpus1/hehcmv	229354	85526	2.682	20	9
DNACorpus1/mpomtcg	186609	70254	2.656	17	7
DNACorpus1/humhdab	58864	22699	2.593	6	2
DNACorpus1/chmpxx	121024	43516	2.781	12	4
DNACorpus1/mtpacga	100314	36862	2.721	10	4
DNACorpus1/chntxx	155844	58336	2.671	15	6
DNACorpus1/humghcs	66495	25552	2.602	8	2
DNACorpus1/humhbb	73308	28134	2.606	8	3

Table 2.8: Corpus 2 Stats after reducing string concatenations

File.Name	Original.File.Size	Compressed.Size	Compression.Ratio	Compression.Time	Decompression.Time
DNACorpus2/YeMi	73689	27235	2.706	8	3
DNACorpus2/DaRe	62565020	19586457	3.194	14450	1876
DNACorpus2/EnIn	26403087	8609993	3.067	5186	762
DNACorpus2/HePy	1667825	566972	2.942	129	38
DNACorpus2/OrSa	43262523	14148071	3.058	9392	1329
DNACorpus2/EsCo	4641652	1593404	2.913	525	119
DNACorpus2/GaGa	148532294	46851765	3.170	39700	4741
DNACorpus2/ScPo	10652155	3590856	2.966	1640	289
DNACorpus2/HaHi	3890005	1306708	2.977	410	96
DNACorpus2/HoSa	189752667	57200209	3.317	51901	5848
DNACorpus2/AeCa	1591049	556535	2.859	126	40
DNACorpus2/DrMe	32181429	10619042	3.031	6861	1001
DNACorpus2/BuEb	18940	7893	2.400	2	0
DNACorpus2/PlFa	8986712	2895744	3.103	1258	234
DNACorpus2/AgPh	43970	17442	2.521	5	2

2.3.5 Dictionary Accesses

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

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

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

See the code below

```
while(next_character != EOF){

    // code omitted


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

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


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

We can save that lookup, like so.

```
while(next_character != EOF){

    // code ommitted

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

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

        // code omitted

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

Here are the stats of the version of the algorithm after this change.

Table 2.9: Corpus 1 Stats after reducing dictionary accesses

File.Name	Original.File.Size	Compressed.Size	Compression.Ratio	Compression.Time	Decompression.Time
DNACorpus1/humprtb	56737	21902	2.590	5	2
DNACorpus1/humdyst	38770	15300	2.534	4	1
DNACorpus1/vaccg	191737	70067	2.736	17	7
DNACorpus1/hehcmv	229354	85526	2.682	20	9
DNACorpus1/mpomtcg	186609	70254	2.656	17	7
DNACorpus1/humhdab	58864	22699	2.593	6	2
DNACorpus1/chmpxx	121024	43516	2.781	11	4
DNACorpus1/mtpacga	100314	36862	2.721	11	3
DNACorpus1/chntxx	155844	58336	2.671	16	6
DNACorpus1/humghcs	66495	25552	2.602	7	2
DNACorpus1/humhbb	73308	28134	2.606	8	3

Table 2.10: Corpus 2 Stats after reducing dictionary accesses

File.Name	Original.File.Size	Compressed.Size	Compression.Ratio	Compression.Time	Decompression.Time
DNACorpus2/YeMi	73689	27235	2.706	7	3
DNACorpus2/DaRe	62565020	19586457	3.194	14146	1869
DNACorpus2/EnIn	26403087	8609993	3.067	5087	758
DNACorpus2/HePy	1667825	566972	2.942	127	39
DNACorpus2/OrSa	43262523	14148071	3.058	9278	1329
DNACorpus2/EsCo	4641652	1593404	2.913	519	115
DNACorpus2/GaGa	148532294	46851765	3.170	39265	4702
DNACorpus2/ScPo	10652155	3590856	2.966	1610	290
DNACorpus2/HaHi	3890005	1306708	2.977	405	97
DNACorpus2/HoSa	189752667	57200209	3.317	50936	5833
DNACorpus2/AeCa	1591049	556535	2.859	125	39
DNACorpus2/DrMe	32181429	10619042	3.031	6773	999
DNACorpus2/BuEb	18940	7893	2.400	2	0
DNACorpus2/PlFa	8986712	2895744	3.103	1254	231
DNACorpus2/AgPh	43970	17442	2.521	5	2

2.3.6 Using Const Char *

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

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

Table 2.11: Corpus 1 Stats after using character pointers

File.Name	Original.File.Size	Compressed.Size	Compression.Ratio	Compression.Time	Decompression.Time
DNACorpus1/humprtb	56737	21902	2.590	7	2
DNACorpus1/humdyst	38770	15300	2.534	7	1
DNACorpus1/vaccg	191737	70067	2.736	15	6
DNACorpus1/hehcmv	229354	85526	2.682	15	8
DNACorpus1/mpomtcg	186609	70254	2.656	16	6
DNACorpus1/humhdab	58864	22699	2.593	9	2
DNACorpus1/chmpxx	121024	43516	2.781	12	4
DNACorpus1/mtpacga	100314	36862	2.721	10	3
DNACorpus1/chntxx	155844	58336	2.671	14	5
DNACorpus1/humghcs	66495	25552	2.602	9	2
DNACorpus1/humhbb	73308	28134	2.606	9	2

Table 2.12: Corpus 2 Stats after using character pointers

File.Name	Original.File.Size	Compressed.Size	Compression.Ratio	Compression.Time	Decompression.Time
DNACorpus2/YeMi	73689	27235	2.706	9	2
DNACorpus2/DaRe	62565020	19586457	3.194	12264	1764
DNACorpus2/EnIn	26403087	8609993	3.067	4070	708
DNACorpus2/HePy	1667825	566972	2.942	96	35
DNACorpus2/OrSa	43262523	14148071	3.058	7627	1252
DNACorpus2/EsCo	4641652	1593404	2.913	348	109
DNACorpus2/GaGa	148532294	46851765	3.170	32906	4456
DNACorpus2/ScPo	10652155	3590856	2.966	1205	265
DNACorpus2/HaHi	3890005	1306708	2.977	271	88
DNACorpus2/HoSa	189752667	57200209	3.317	44733	5517
DNACorpus2/AeCa	1591049	556535	2.859	92	35
DNACorpus2/DrMe	32181429	10619042	3.031	5266	940
DNACorpus2/BuEb	18940	7893	2.400	5	0
DNACorpus2/PlFa	8986712	2895744	3.103	879	220
DNACorpus2/AgPh	43970	17442	2.521	7	1

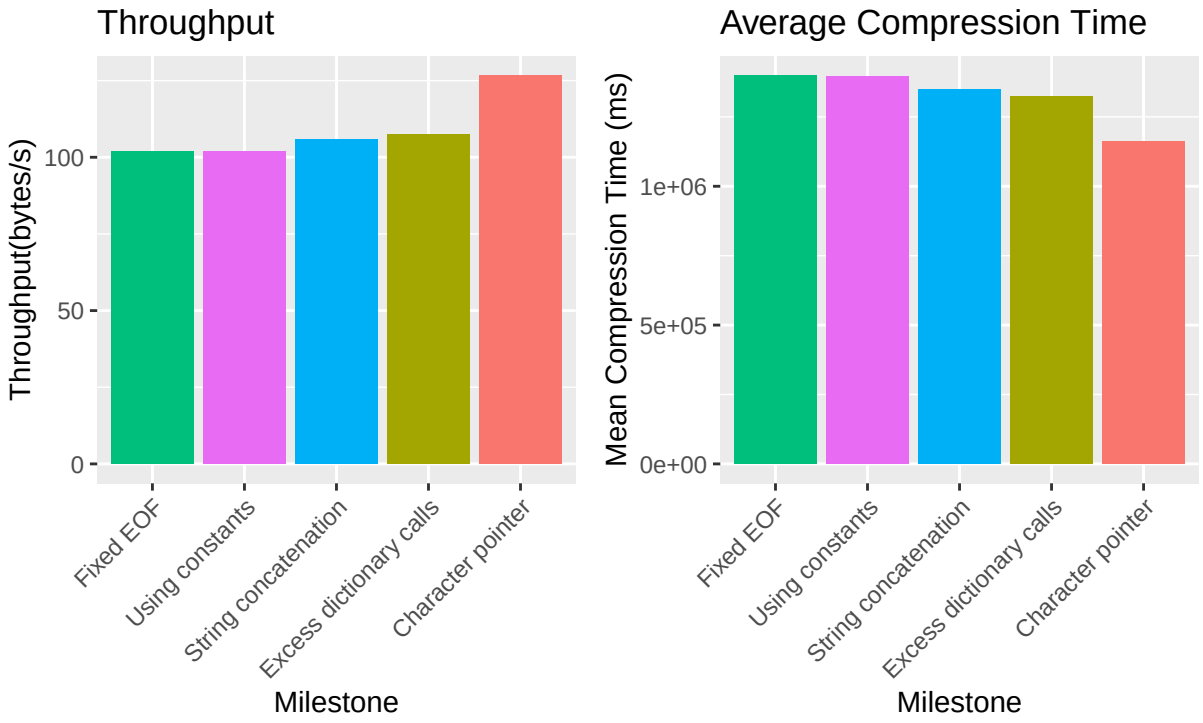


Figure 2.1: Comparison of the performance of the different milestones

2.4 Trying Different Dictionaries

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

So the another step towards an efficient LZW seemed to be to abstract out the `std_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.

Below is a table which shows stats on the runs lengths of the "runs" of data, where a run is a string added to the dictionary during a run of the LZW algorithm. And here is a histogram of all the runs2.2.

Table 2.13: Run Lengths for Corpus 1

average_run_length	maximum_run_length	median_run_length	sd_run_length
6.135398	17	6	1.237217

Table 2.14: Run Lengths for Corpus 2

average_run_length	maximum_run_length	median_run_length	sd_run_length
10.96825	190	11	2.494305

Table 2.15: Run lengths for both corpora

X	average_run_length	maximum_run_length	median_run_length	sd_run_length
1	10.95318	190	11	2.505904

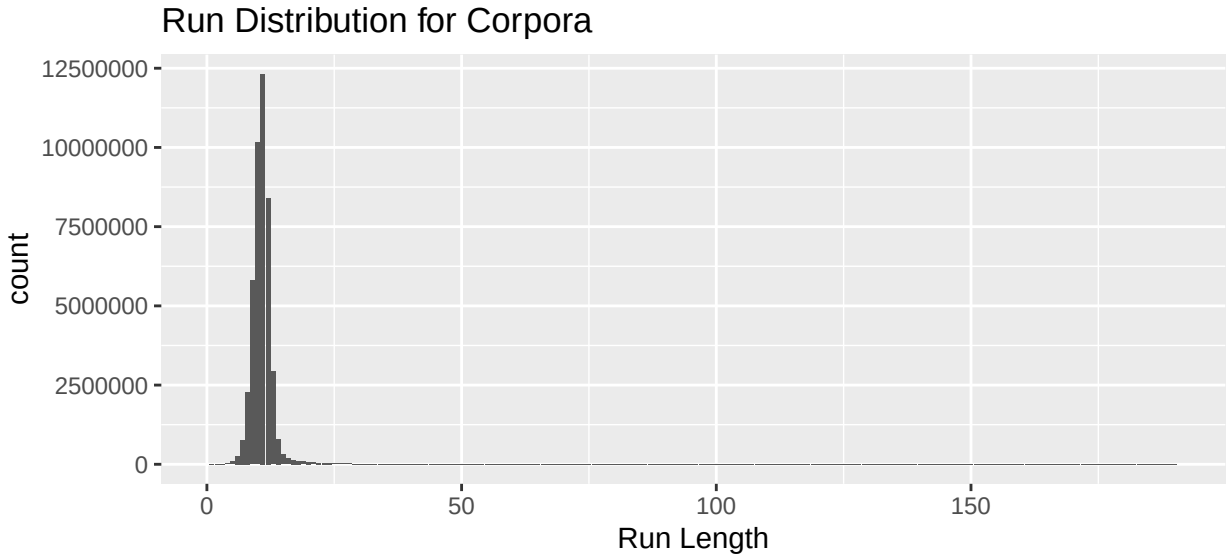


Figure 2.2: A histogram showing the lengths of runs for both copora.

Given this data, it is clear that it would be advantageous to speed up the dictionary for smaller run sizes, since most of the runs are below size 15.

To achieve this, we opted for a unique approach. Rather than use a hashmap, we can map the strings directly into memory. Since all of the strings only contain four characters (‘A’, ‘C’, ‘T’, and ‘G’), we can represent the characters with two bits. So for a length `n` string, we can represent it with `2n` bits.

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

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

Here is the data for this version of the dictionary of length 10. Here is a

Table 2.16: Corpus 1 Stats for Direct Map of max length 10

File.Name	Original.File.Size	Compressed.Size	Compression.Ratio	Compression.Time	Decompression.Time
DNACorpus1/humprtb	56737	25883	2.192	1	2
DNACorpus1/humdyst	38770	18437	2.103	1	1
DNACorpus1/vaccg	191737	77120	2.486	4	4
DNACorpus1/hehcmv	229354	93251	2.460	2	5
DNACorpus1/mpomtcg	186609	77315	2.414	4	4
DNACorpus1/humhdab	58864	26752	2.200	1	2
DNACorpus1/chmpxx	121024	49414	2.449	3	3
DNACorpus1/mtpacga	100314	42203	2.377	2	2
DNACorpus1/chntxx	155844	64879	2.402	3	3
DNACorpus1/humghcs	66495	29864	2.227	1	2
DNACorpus1/humhbb	73308	32681	2.243	1	2

Table 2.17: Corpus 2 Stats for Direct Map of max length 10

File.Name	Original.File.Size	Compressed.Size	Compression.Ratio	Compression.Time	Decompression.Time
DNACorpus2/YeMi	73689	31700	2.325	1	2
DNACorpus2/DaRe	62565020	21182510	2.954	656	442
DNACorpus2/EnIn	26403087	8986109	2.938	271	184
DNACorpus2/HePy	1667825	572507	2.913	26	17
DNACorpus2/OrSa	43262523	15008736	2.882	431	319
DNACorpus2/EsCo	4641652	1621508	2.863	51	42
DNACorpus2/GaGa	148532294	52441876	2.832	1475	1080
DNACorpus2/ScPo	10652155	3691943	2.885	108	83
DNACorpus2/HaHi	3890005	1324571	2.937	51	32
DNACorpus2/HoSa	189752667	63973137	2.966	2035	1319
DNACorpus2/AeCa	1591049	561527	2.833	22	18
DNACorpus2/DrMe	32181429	11150498	2.886	332	237
DNACorpus2/BuEb	18940	9894	1.914	0	1
DNACorpus2/PlFa	8986712	2972072	3.024	106	67
DNACorpus2/AgPh	43970	20885	2.105	1	1

comparison for different max string lengths.

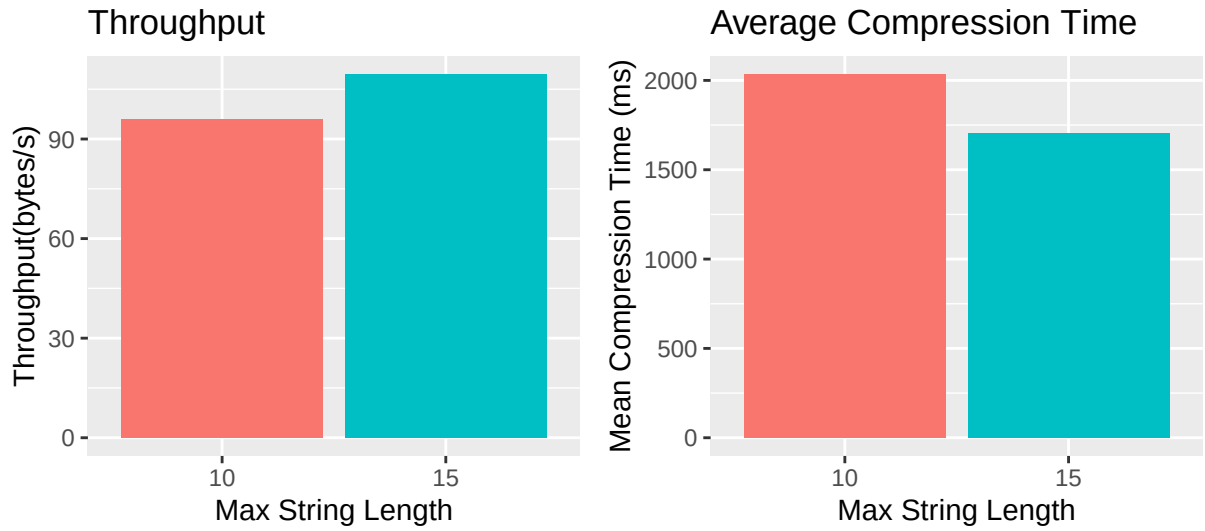


Figure 2.3: Comparing different max lengths for Direct Map

The direct map method is very memory intensive, so having a max length over 15 is not feasible. At this point, if any runs are longer than the max string length, we just use a regular dictionary to accommodate them.

2.4.2 Multiple Indexed Dictionaries

As stated before, there have been research papers about the possibility of using multiple indexed dictionaries for LZW, including Keerthy (Keerthy, 2019).

Similar to the Direct Mapped approach, we use dictionaries for each string size up to a certain size `n` , and for all strings of length greater than `n`, we use a regular dictionary. As with the direct map dictionary, we need to specify a max string length. We collected metrics for different choices of max string length. Here are the results for this dictionary.

Table 2.18: Corpus 1 Stats for Multiple Dictionaries of max length 10

File.Name	Original.File.Size	Compressed.Size	Compression.Ratio	Compression.Time	Decompression.Time
DNACorpus1/humprtb	56737	21687	2.616	5	2
DNACorpus1/humdyst	38770	15116	2.565	3	1
DNACorpus1/vaccg	191737	69820	2.746	14	6
DNACorpus1/hehcmv	229354	85279	2.689	16	7
DNACorpus1/mpomtcg	186609	70007	2.666	15	6
DNACorpus1/humhdab	58864	22484	2.618	5	2
DNACorpus1/chmpxx	121024	43269	2.797	10	4
DNACorpus1/mtpacga	100314	36647	2.737	8	3
DNACorpus1/chntxx	155844	58089	2.683	12	5
DNACorpus1/humghcs	66495	25336	2.625	6	2
DNACorpus1/humhbb	73308	27918	2.626	7	2

Table 2.19: Corpus 1 Stats for Multiple Dictionaries of max length 10

File.Name	Original.File.Size	Compressed.Size	Compression.Ratio	Compression.Time	Decompression.Time
DNACorpus2/YeMi	73689	27019	2.727	7	2
DNACorpus2/DaRe	62565020	19585959	3.194	8878	1721
DNACorpus2/EnIn	26403087	8609527	3.067	3347	696
DNACorpus2/HePy	1667825	566631	2.943	94	34
DNACorpus2/OrSa	43262523	14147605	3.058	5551	1223
DNACorpus2/EsCo	4641652	1593032	2.914	364	105
DNACorpus2/GaGa	148532294	46851235	3.170	22817	4323
DNACorpus2/ScPo	10652155	3590421	2.967	1099	258
DNACorpus2/HaHi	3890005	1306336	2.978	302	86
DNACorpus2/HoSa	189752667	57199679	3.317	31978	5449
DNACorpus2/AeCa	1591049	556194	2.861	91	34
DNACorpus2/DrMe	32181429	10618575	3.031	3865	905
DNACorpus2/BuEb	18940	7740	2.447	1	0
DNACorpus2/PlFa	8986712	2895340	3.104	892	211
DNACorpus2/AgPh	43970	17258	2.548	4	1

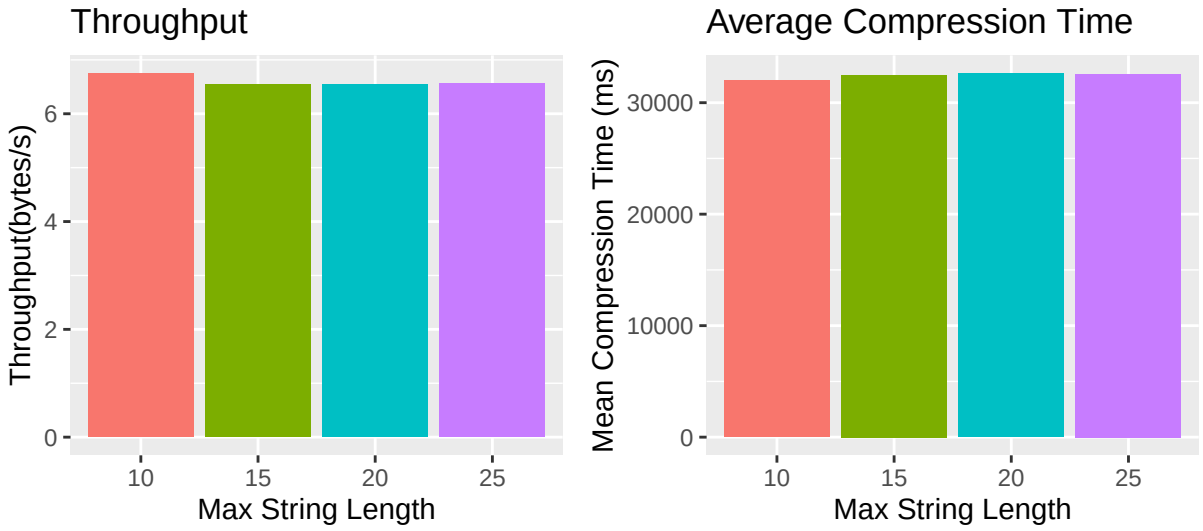


Figure 2.4: Comparing different max lengths for Direct Map

Here is a comparison of the Multiple Dictionaries versus one standard dictionary.

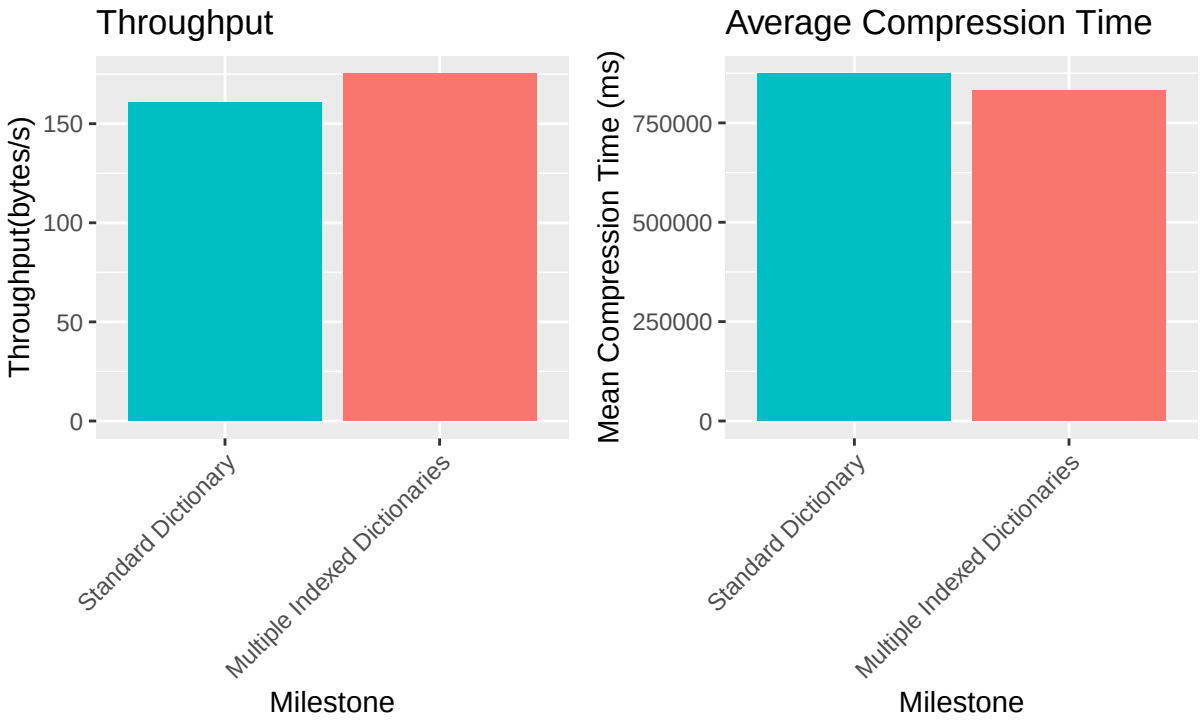


Figure 2.5: Comparing one Dict to Mult Dict

2.4.3 Comparison

Here is a comparison of all three techniques: Standard Dictionary, Multiple Standard Dictionaries, and Direct Mapped Dictionary.

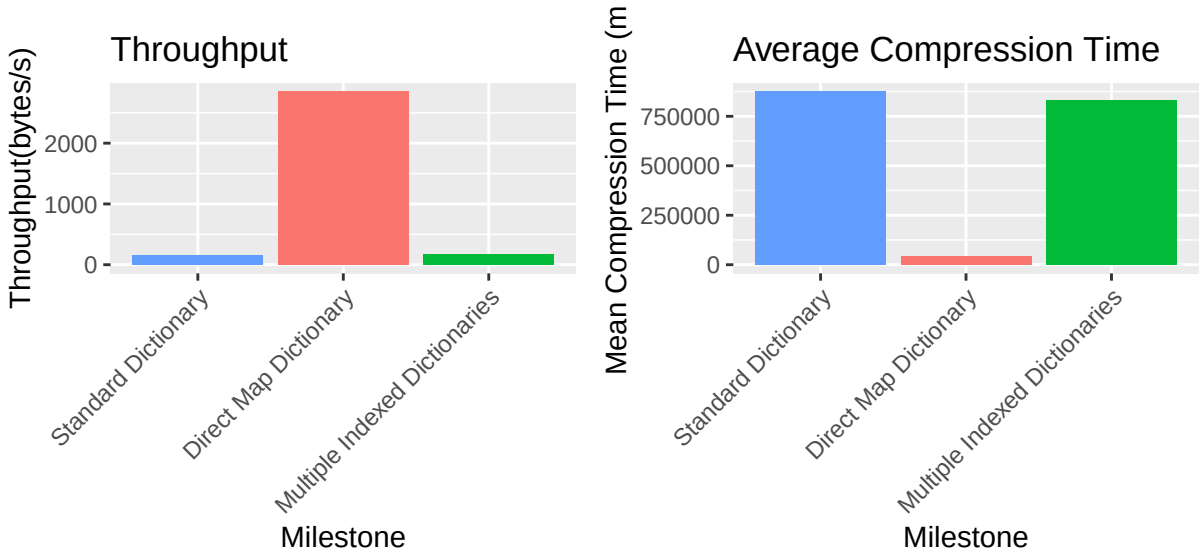


Figure 2.6: Comparing the three types of dictionaries. The Direct Map has a max length of 15 and the Mult Dict has a max length of 10.

2.5 Optimizing Direct Map Even more

Since the Direct map dictionary was a standout in performance, we decided to focus on optimizing it even further.

2.5.1 Finding the Longest Runs

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

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

Another strategy would be to start from the maximum string length of the dictionary. We can check if the next string of the max length 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 or binary search down from the max. Here is a table showing the performance of these different strategies.

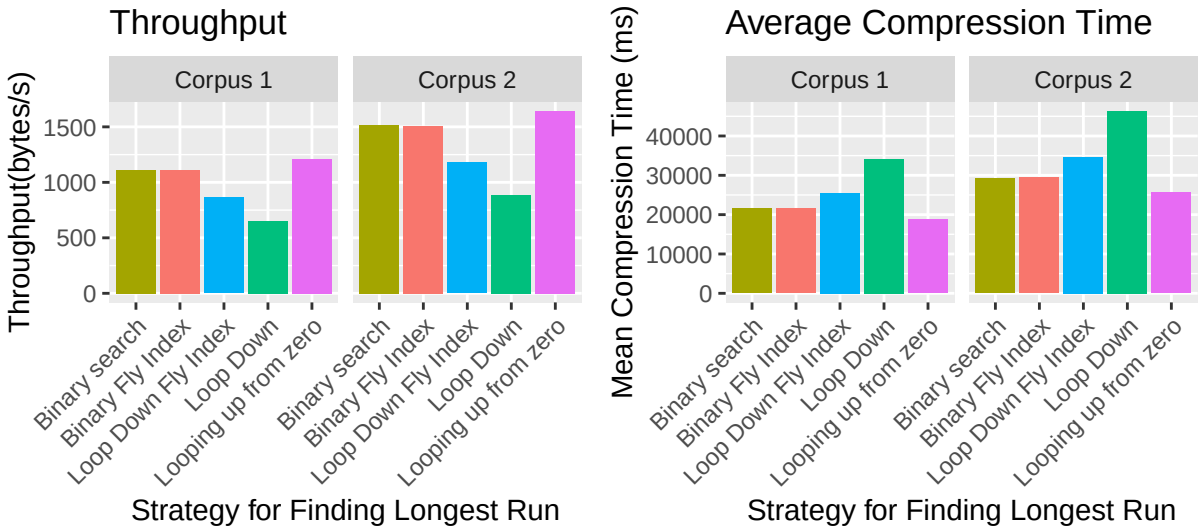


Figure 2.7: Comparing the different ways of finding the longest run

We 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

Here are the results when we try looking for runs starting at the average.

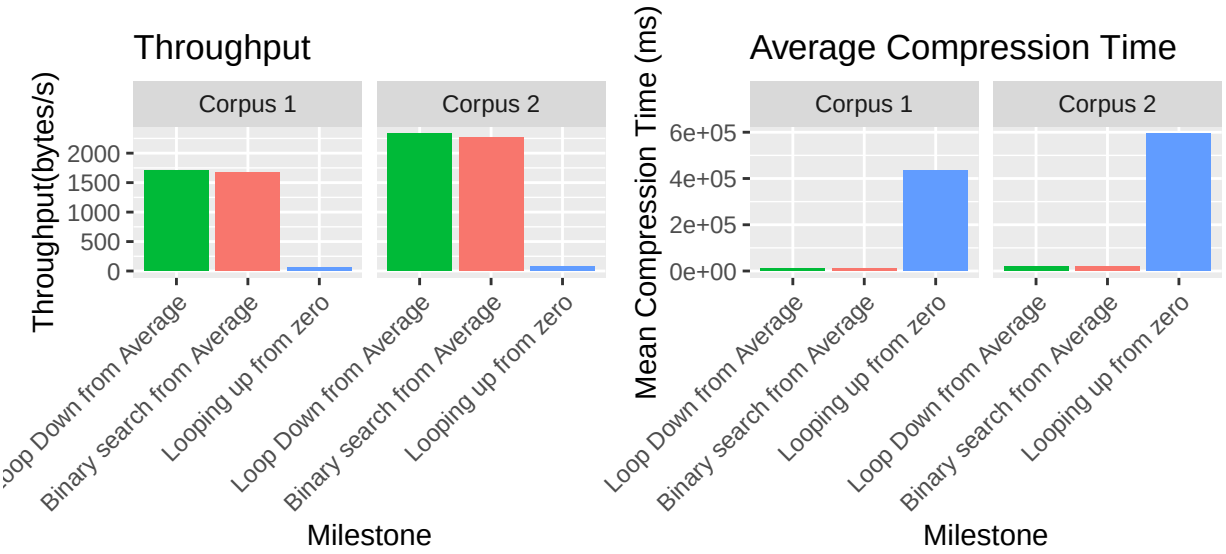


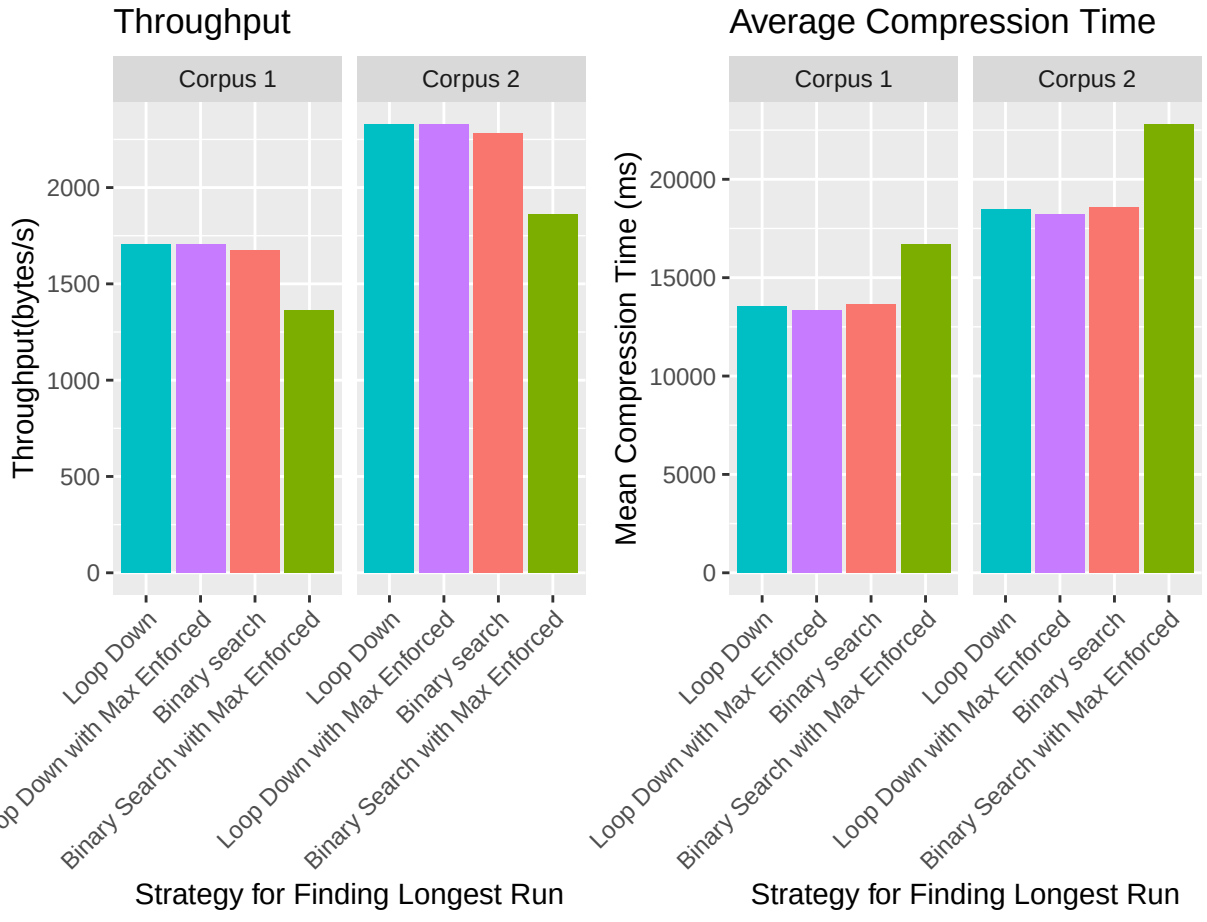
Figure 2.8: Comparing the different ways of finding the longest run

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

2.5.3 Not Allowing strings over max

One way to avoid the edge case where we encounter a very long run is to just not allow strings in the dictionary longer than the max string length, in this case, 15. This means we won't have to deal with the overhead of the hashmap on top of our dictionary, but we will take a hit in compression ratio.

Here is a graph comparing the different strategies.



There isn't too much of a performance improvement, which makes sense because long runs are very rare. Let's see how it affects our compression ratio. The effect on the total compression ratio is minimal

Table 2.20: Compression Ratio change from disallowing long strings

cr_before	cr_after
2.909175	2.907686

2.5.4 Using `pext`

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

Our strategy is to use `pext`, which extracts parallel bits. We give `pext` a string of characters, say 'ACTG', and a bit mask, and it will extract those bits from our string. It theoretically does this in one machine instruction, which could be much more efficient than looping over all the characters.

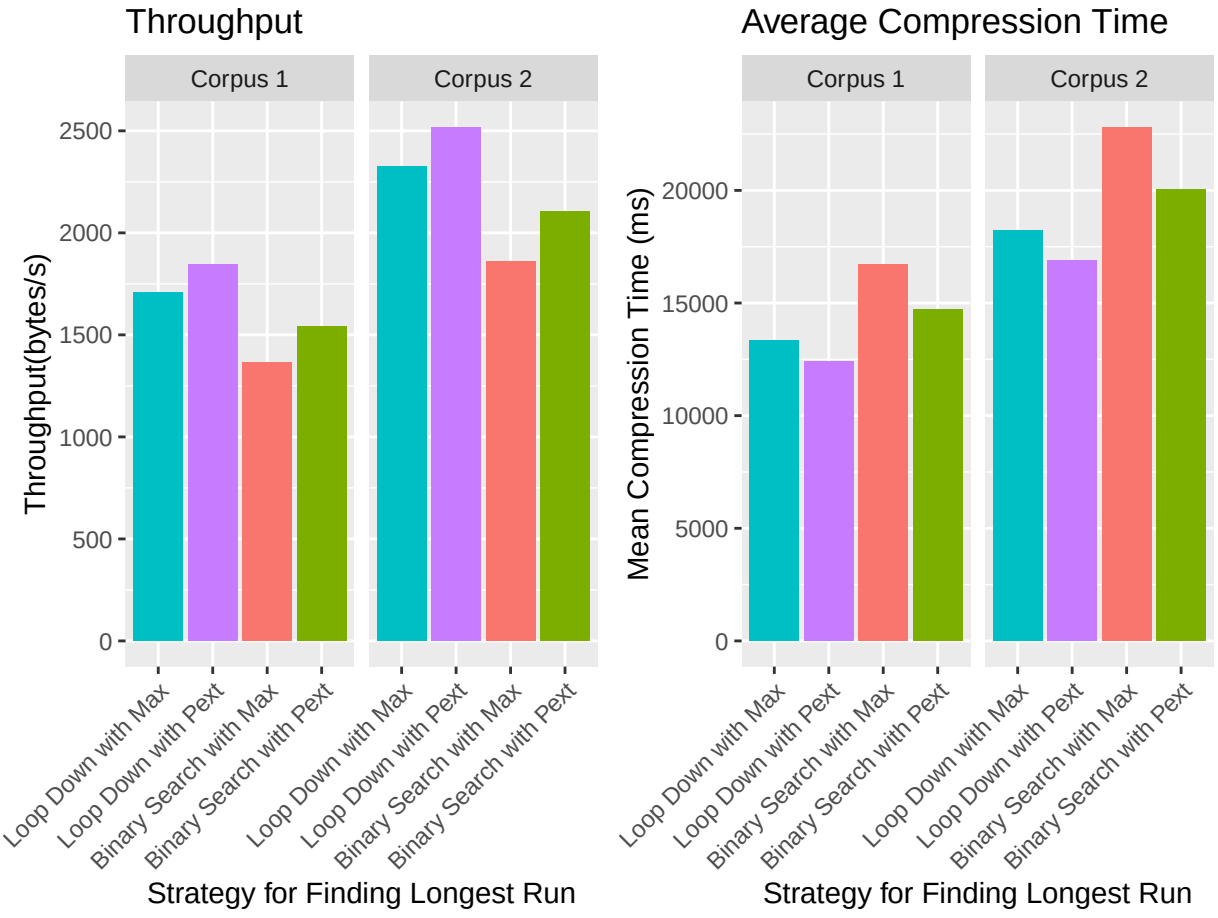


Figure 2.9: Comparing the different ways of finding the longest run with pext

Why doesn't it work?

Chapter 3

Comparison to other tools

Here, in this chapter, we will investigate how our implementation holds up against other compression methods.

Conclusion

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

More info

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

Appendix A

The First Appendix

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

In the main Rmd file

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

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

In Chapter ??:

Appendix B

The Second Appendix, for Fun

References

Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. Pr

Sayood, K. (2017). Introduction to data compression. Academic Press.

Witten, I. H., Neal, R. M., & Cleary, J. G. (1987). Arithmetic coding for data compr

Sayood, K. (2017). Introduction to data compression. Academic Press.

Pratas, Diogo & Pinho, Armando. (2018). A DNA Sequence Corpus for Compression Benchm

Angel, E. (2000). *Interactive computer graphics : A top-down approach with OpenGL*. Boston, MA: Addison Wesley Longman.

Angel, E. (2001a). *Batch-file computer graphics : A bottom-up approach with QuickTime*. Boston, MA: Wesley Addison Longman.

Angel, E. (2001b). *Test second book by angel*. Boston, MA: Wesley Addison Longman.

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

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

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

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

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