Codes Summative 2021 Assignment

Ali'sa-Falaq Hussain

Thursday 21st January, 2021

1 Introduction

The objective of this assignment was to develop an encoder and decoder for lossless compression of LaTeXfiles, receiving a *.tex* document as input and outputting a compressed *.lz* file. The following report outlines the key ideas applied in our algorithm, justifying and evaluating the choices. Note the assignment objective is compression size oriented, as opposed to compression speed. As a result. the ideas discussed have often required a speed trade-off.

2 Approach Overview

Our approach recognises the need to exploit statistical and structural redundancy, exploring how to achieve this most effectively. The foundational methods build upon in the algorithm are Lempel-Ziv-77 and Arithmetic Coding. Lempel-Ziv (LZ) algorithms are renowned for reducing structural redundancy, and conversely, arithmetic coding (AC) works to eliminate statistical redundancy. Our report will assume basic understanding of these compression strategies and build off this groundwork knowledge.

3 Ideas

3.1 Lempel-Ziv-77

In a similar manner to the *DEFLATE* algorithm which is widely used from `png` compression to `gzip`, an LZ approach is used to eliminate structural redundancy before further processing. For our implementation, both LZW and LZ-77 were explored. The key difference between the two is that LZW creates one-field tokens for the of building dictionaries on both the compression and decompression sides, whereas the compressed result of LZ-77 behaves as a dictionary in itself, with a list tuples of the structure (*offset, length, next_character*). Both algorithms were implemented and tested on LaTeXfiles of different sizes (see Table 1.) As expected, LZW proved more effective, producing smaller compressed files. These findings are further supported by literature [1].

Table 1: A comparison of LZW and LZ-77

Compression Type		
Original Size	LZW (size, *ratio*)	LZ-77 *(size, ratio)*
12,054B	7,728B *1.56*	7,930B *1.52*
155,421B	71,150B *2.18*	76,811B *2.02*
393,476B	171,532B *2.29*	182,805B *2.15*

Although appearing counter-intuitive, we decided to overlook the immediate benefits of LZW over LZ-77 and evaluate what would be most effective in conjunction with an Arithmetic Encoder (AE). We note that AC works by assigning probabilities of enountering a symbol, a_1, and this is extended to the probability of a symbol a_1 following another, a_2, i.e. p($a_1 a_2$). Therefore, in order to optimise the performance of an AE, our aim was to use LZ in a manner to produce clear repetitions of symbols and sequence of symols.

We saw that LZ-77 could be adapted so the compressed text still consisted of tuples (*offset, length, next_character*), but each offset would likely be followed by a particular length. For example, any offset of 25 would normally be followed by a length of 3, meaning p(25,3) in would be relatively high when analysed by the AE. (*See **3.2** for further details*). Consistent patterns play to an AE's strengths. Therefore the decision to incorporate and adapt an LZ-77 algorithm was made.

3.2 Reverse Offset Approach

In continuation from *3.1*, we aim to convert the original text into tuples of the format *(offset, length, next_character)* but in a way such that the same sequences are prevelant across the document, particularly *offset* and *length* pairs.

The standard way to encode with LZ-77 is by detecting the longest match to subsequent character(s) from characters in a window `original_text[current_index - maximum_offset:current_index]`, where maximum_offset is the largest offset value that can be represented and current_index is the current pointer position. Note that since they are relative to the current pointer position, any two offsets with the same value represent different positions, and there will be no pattern between *offset* and *length*. Our solution to craft a pattern within the LZ encoded data is to redefine *offset* as an index from the beginning of the file. This means that any two "offsets" of the same value will point to the same position, hence two identical tuples will represent the same set of text. To represent a new character, rather than the standard offset of 0, which now points to the first character in the file, can be replaced with an unused ASCII character, or the largest value that can be represented by the offset bits.

To demonstrate this concept, take a set of text that begins with *"Caecilius ..."*, all further instances of "Caecilius" will be represented by (0, 8, next character).

Tests indicated this approach was effective in producing repeated patterns of offset and length. For instance, in a *.tex* file of size 400KB, tokens with the (offset, length) pair (3688, 7) appeared 20 times, whereas, in the standard approach, sequence repetition to this extend was not prevelant

A disadvantage to this approach is that the text able to be referenced by the LZ encoder is limited to the maximum index that can be represented by the given bits, and common words that appear exclusively in a further section of the text may not be able to be referenced. To overcome this, the encoding could be completed in sections, with the index counter restarted for each section. However, since the given file is said to be 1MB, tests show it is unlikely for this to be necessary, unless content varies significantly, e.g. there is a change in language .

3.3 Next character removal

We saw an opportunity to eliminate the need to encode some of the "next_character" fields. This is a significant discovery since it allows a decrease in the encoded LZ of up to $\sim\frac{1}{3}$. The only time the next_character field is essential is when a new character is encountered. Assuming the only characters in the file are printable ASCII ones, this means the field only needs to be used a maximum of $127 - 32 = 95$ times.

This in turn proves more advantageous for the AE for two reasons. Firstly, less needs to be encoded, making the final compressed file smaller. Secondly, since the patterns within the LZ-encoded data stem from offset and length pairs, and the third field tends to be less consistent, repeated sequences become more prevelant. Therefore, theoretically, an AE would be more effective with this technique in place.

In practice, this technique increased the compression by an average of 15.1 % when tested on files of various sizes between 12KB and 393KB. A few examples are shown in Table 2.

Table 2: Impact of removing the next character

Original size	Next character preserved	Next character removed
100KB	48KB	32KB
50KB	20KB	15B

3.4 Adaptive Arithmetic Encoding

There are several approaches discussed in lectures and literature targeting the elimination of statistical redundancy. We investigate to determine the most effective for this context, out of Shannon Fano [1], Huffman [2], arithmetic [3] and adaptive arithmetic encoding.

In Huffman encoding, codewords are defined on a symbol level, rather than a sequence level. Even if a file consisted of solely 100 '*a*'s, the minimum size of the encoding would be 100 bits, when 1 bit would be sufficient. Evidently, this technique would not be suited for coding the patterns we have produced in the LZ-encoding as effectively as a technique that looks on a sequence level. Shannon Fano similarly encodes on a symbol level, so we eliminate these both from consideration.

Alternatively, arithmetic coding surveys on a sequence level, assigning tags and probabilities to a particular series of characters. Since we are finding that each offset is often followed by a certain length, arithmetic coding should be able to recognise that, and assign a large probability interval to this sequence.

Extending this, adaptive arithmetic encoding changes frequency tables as the data is being processed. This results in frequency tables that are more tailored to the given input. In practise, this proved to be more effiective than just Arithmetic Encoding.

3.5 Dictionary Pre-Processing

The incorporation of a dictionary is standard in many encoding approaches. The aim is to encode a sequence of characters - usually those particularly long or frequent, as a shorter token to be encoded with fewer bytes. We explore how the concept of a dictionary can be applied to this context.

Given that the input file is LaTeX, we are aware of many common commands that have a high probability of appearing in a given *.tex* file. Examples include \documentclass, \begin{centre} and \section. To obtain common words and commands used in LaTeXfiles, 40 lecture *.tex* files from the "L3 Codes and Cryptography" module were analysed. Word frequencies were calculated, and candidates for the dictionary were manually selected. This was to ensure words such as "a" and "&", two of the most common, were ignored. Words of length 1 would not benefit from being replaced with a token of the same length. Similarly, short words such as "of" would be normally be encoded into a token after at most its second encounter, hence including it in the dictionary would have insignificant impact on file size. Rather, prevelant commands such as \textbf and \begin{align*} were added. This is advantageous since these commands no longer need to be built up over several tokens, letter-by-letter, at each encounter. A further benefit is that more symbols and sequences can now be referenced with the limited number of bits representing the index \"offset".

Substring occurences of dictionary terms are also replaced. For example, the code can immediately reference \textbf{ 's replacement from the dictionary, and treat it as any other symbol, encoding it with the rest of its word.

It is important to ensure dictionary replacements will not already exist in the text, to avoid interference. An easy solution is to use unprintable characters - in ASCII these are values 0-31. However, using these values appeared to interfere with the encoding. As an alternate approach, dictionary tokens were be extended to two bytes, one being an unprintable character, 07 [BELL], flagging the use of a dictionary term, and the second any other printable ASCII character. Larger dictionary tokens are used for more infrequent, longer sequences and commands. As expected, tailoring the dictionary to common LaTeXsequences and commands proved more effective than when using a regular English text corpus.

3.6 Ensemble of methods

Our adaptations were very effective at producing high compression ratios, superior to LZ-77 and arithmetic coding alone. However, after implementation, we experimented further, taking new routes and implementing other techniques, comparing them with our novel approach. We found that a PPM approach build on the arithmetic encoder proved more effective in some circumstances, but not others. Further reading birthed the idea of executing both strategies, and keeping the smaller final file, indicating the used method by prepending the compressed file with a byte-length token. We decided to incorporate this approach to generate as optimal an encoding as possible. Although this comes at the expense of an extra byte and encoding time, often the final compressed size justifies its use. The decoding is significantly faster, since only one method needs to be applied, as indicated by the token.

3.7 Conclusion

To summarise, we explored the use of different mainstream compression algorithms, but from a novel perspective, considering pre-processing methods and transforms to optimise the reduction of structural and statistical redundancy. Our changes have proved successful, producing performance superior to both LZ-77 and adaptive arithmetic coding. Each idea discussed positively impacted compression performance, some more than others. For instance, using an adaptive approach for AE, idea *3.4* was beneficial, but not nearly as impactful as idea *3.3*, next character removal, which increased compression by an average of 15.1%. Given more time, future improvements would include a more extensive dictionary after analysis of general LaTeXfiles, rather than restricting to files from one module on my course. It may appear that word frequencies from a more general file pool, especially including different document types e.g. slides, have a significantly varied distribution of words. Furthermore, we would restart LZ encoding and indexing later into the document for larger files, since earlier words may not be representative of later content. For example, the end of a paper may have more math symbols, which may not have been encoded in the given index bits. Finally, ideas presented have certainly come at the expense of speed. This is not practical, so further advancements would look to address this. Overall, this project has successfully explored and developed an informed approach for effective lossless compression.

4 Implementation notes

In order to view the progress bars when running code, ensure to `pip install tqdm`

References

[1] S. Shanmugasundaram and R. Lourdusamy, "A comparative study of text compression algorithms," *International Journal of Wisdom Based Computing*, vol. 1, no. 3, pp. 68–76, 2011.

[2] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[3] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, 1987.