Stack-based LaTeX File Compression

wcrr51

January 2021

1 Introduction

This report will describe six ideas used in the implementation of a LaTeX file compression scheme. This scheme works under the assumption that only one LaTeX file is being encoded and decoded at a time, however, there is nothing to stop multiple files being independently encoded and decoded.

Additionally, it is assumed that input LaTeX files are encoded using ASCII (1 bit of padding plus 7 bits per character) and may be mal-formed (in terms of syntax and semantics), no assumptions are made regarding the line ending format.

2 Compression Stack

The first idea is that of a compression scheme using a *stack* model. In the same way the OSI model is used in networking to wrap and encode data the more low-level it becomes, this scheme may combine independent techniques to compress data at each level of abstraction.

Broadly speaking, when encoding, this goes from the application-level to the text-level through encoding of the input plaintext LaTeX file, followed by the text-level to the binary level and finally any binary level compression. Conversely, decoding simply does these operations in reverse, by sending the input code back up the stack. Each layer may wrap the level above it (with code-specific metadata), or store auxiliary information such as pre-determined dictionaries or frequency tables. The advantage of this separation-of-concerns approach is that each layer can be tested independently and layers can be enabled, disabled and interchanged as required.

Because of this, standard context-agnostic compression algorithms can be inserted at any point as long as the output type from the previous layer matches the input type to the next (and vice versa). In addition to these standardised approaches, this scheme benefits the most from the ability to use context-optimised approaches and/or novel context-specific approaches. As an aside, it is worth noting that, if desired, this scheme would also enable encryption and error-correction layers to be added at any point.

It is important that compression methods are stacked carefully. Different ordering is usually not commutative, as such it is probable that ordering affects compression ratio, and, beyond a certain point, additional layers of compression may end up increasing the file size.

3 English and LaTeX Dictionary Compression

This idea aims to reduce the size of the file using prior knowledge of LaTeX commands and symbols. LaTeX is, for the most part, very well-structured. Ideally, the entire document would be parsed and stored efficiently as its parse-tree, leaving only parameters and blocks of text to be (potentially independently) encoded. However, since this approach would be very complex and would require the testing of many edge cases, it is not used in the implementation.

Despite this, encoding and decoding at the text level still presents a good opportunity to improve the compression ratio beyond that of algorithms which do not take context into account. Hence, the trivial idea with a custom implementation is proposed for a static dictionary.

In the same way that common LaTeX commands are placed in the dictionary, common English words can also be encoded. However, there are diminishing and even negative returns when increasing the dictionary size. For given matches, this method marginally out-performs most LZ-family algorithms (when using a 2-byte pointer) as there is no need for the original copy of a dictionary item to be stored. Training and testing the dictionary compression with Huffman encoding on the dataset described in Parameter Optimisation and Dictionary Selection revealed that better compression was obtained by using a 2-byte encoding with a 30,000 word dictionary (about a 1.6 compression ratio) vs 3-byte encoding using a 500,000 word dictionary (about a 1.5 compression ratio).

4 Huffman Coding

Huffman is a form of entropy encoding that creates a varying-length binary codeword for each symbol, the bit length of the code is dependent on the probability of its symbol occurring. Its elegance and relative simplicity has seen its continued use in zip files since 1989 as one of the two algorithms used in deflate. When encoding, each input symbol is simply replaced by its codeword (which is generated from a Huffman tree corresponding to the symbol frequencies). When decoding, the opposite is done by traversing the same Huffman tree.

For an accurate probability-mass-function, Huffman encoding produces the optimal binary coding for single symbols. For a constant frequency table the tree and codes can be pre-calculated, they could be statically stored without the need to encode them alongside the data. This can however be improved by using a dynamic frequency table, this is discussed in Dynamic Context-based Coding. A full custom implementation of Huffman coding is included. In the compression stack, this is run after the dictionary encoding.

5 Dynamic Context-based Coding

A naive approach to account for this would be that of simple adaptive coding. This is where the frequency table is updated at each "read" and "write" to reflect the frequencies of each symbol in the read / written code so far. For this application however, a static frequency table trained on LaTeX files would yield similar compression performance without the overhead caused by an initial flat-frequency table. A small improvement could be to initialise the adaptive frequency table to such a trained adaptive frequency table.

The better approach would be to recognise that English text has a varying probability of a character occurring based on the previous character. To address this, Prediction by Partial Matching (PPM) looks to dynamically predict symbol probabilities based on previously seen symbols using a Markov model of a specified order. It can be applied during Huffman coding as a replacement of a static (or encoded) frequency table where the Huffman tree is updated to reflect the new contextual probabilities at the encoding of each symbol.

The PPM implementation used here is based on that used in [1] (except adapted for use with the custom Huffman implementation). This works by forming a wrapper around the Huffman coder. It keeps track of the Markov model and frequency table history in its own state and updates the Huffman tree with the current frequencies upon the reading or writing of each symbol.

6 Parameter Optimisation and Dictionary Selection

The next idea is that of obtaining optimal (or near-optimal) parameters for the various compression algorithms. For this, a machine learning approach is taken. [2] demonstrates that the frequencies of commands used varies significantly based on the top they are being used for. Because of this, a test suite of 25 LaTeX files of varying sizes and disciplinary backgrounds are used. While all of these files are used for unit-testing of individual coding algorithms, only 20 are used for training and parameter tuning. The remaining 5 are used to evaluate the changes made.

For the dictionary, the English part consists of the 30,000 most common English words (from [3]) with words of length less than 3 not included to prevent unnecessary bloating. The LaTeX part of the dictionary is appended to the English part and is trained using the 20 training files. This is achieved by extracting all LaTeX backslash commands and storing them in a set. Doing so produces a dictionary containing a map from 30,449 words and commands. This is ideal because they can all fit nicely into 2-byte ((1 + 15)-bit) codewords (where the first bit is used as a flag to signal a codeword), matching that of many LZ-family implementations.

For PPM, the primary parameter is the order for which the Markov model is trained to. The value for this is primarily a trade-off between time / space complexity and the compression performance. The time and space required is exponential as a tree of frequency tables needs to be generated. Another consideration is that for smaller files, a large order will not have much effect as it would still effectively be training itself by the end of the file. To get around this, like with adaptive coding, an improvement (especially for smaller files) would be to pre-train and save a static Markov model for the LaTeX files. That said, based on testing of orders between one and five (on the 25-file test suite), an order of three seems to produce a good compromise and when combined with the dictionary coding can produce compression ratios of around three on large files while not taking too long (relatively speaking).

7 LZSS Compression

LZSS is the other of the two compression algorithms used in Deflate. It works by storing dictionary references to symbol sequences within a fixed-length sliding window before the sequence being encoded. These references are stored in-line and are differentiated from literals by either prepending a flag bit to each byte or storing a byte consisting of the flags for the next 8 elements. If used at the top-level of compression, it could exploit the fact that the characters are all encoded using ASCII. This frees up the most significant bit of each byte to be used as the reference flag, withdrawing the need for the flag bit to take up valuable extra bytes.

If the top-level dictionary layer was not being used, a pre-defined dictionary could be implemented into LZSS by prepending it to the data being encoded. This is done for both encoding and decoding by initialising the search buffer to the dictionary at each chunk before running LZSS. This means that theoretically, given a large-enough sliding-window size, the entire LaTeX specification (or the most commonly used subset of it) could be used as the dictionary to reduce all LaTeX to the pointer size. The same can be achieved with a dictionary consisting of the most commonly used words in the English language.

Since a form of dictionary encoding is already used with a large dictionary trained on LaTeX files, LZSS is not included implementation.

References

[1] "Reference Arithmetic Coding." `https://github.com/nayuki/Reference-arithmetic-coding`.

[2] "Latex Command Frequencies." `https://www.johndcook.com/blog/2020/04/19/latex-command-frequencies/`.

[3] "High-frequency Vocabulary." `https://github.com/derekchuank/high-frequency-vocabulary`.