Compression Report

jcmk46 (jcmk46@durham.ac.uk)

January 6, 2021

The purpose of the exercise is to produce a program to losslessly encode a tex file. The only way to have better performance than pre-existing techniques is to leverage the specificity of the task to tex files.

1. Idea 1

Fundamentally, if I cannot beat the performance of a library that can implement a basic zip compression function, like that of the Zipfile library, then that will be the default implementation. The library can utilise deflate, bzip2 and LZMA compression methods, and given there is no time aspect to the task, the one with typically the best compression ratios would be used.

2. Idea 2

That said, this is not a very interesting approach to the problem. Given the nature of a tex file, there are some assumptions that can be made. The majority of such a file should be in sensible english, and not a stream of random letters. Such a format lends itself well to contextual compression, and PPM is a suitable candidate to effectively compress text data. At present I have found one such library capable of ppm but this relies on a C backend and a C compiler to be installed on the machine so this goes a bit beyond the requirements of the task. This leaves the only way to utilise PPM as a personal implementation.

3. Idea 3

Beyond the basic connection of sensible english to a contextual compression algorithm, other details of a tex file include the nature of the layout, with regularly occurring commands like `\section` and `\item`. Moreover, we can conclude things like every `\begin` command will have a corresponding `\end` command. We would also expect for every open bracket, parenthesis or brace to have a corresponding closing bracket, parenthesis or brace, but that is not always the case, as shown here). Although such a circumstance is probably very rare, and checking for such an event is very easy, I feel it would be easy to get carried away, and be misguided from more useful features. A simple solution for the first point, however, regarding `\begin` and `\end` statements is having a heap that stores the associated parameter like `enumerate` or `document` and then popping that parameter again whenever the `\end` codeword is read.

4. Idea 4

Perhaps in the wrong order, but the next point again refers to the prior discussed commands in latex, and given the more regular use of some commands over others, and given the limited number of commands that would exist in a document. An entirely uneducated guess on my behalf would be that a typical document uses less than 30 unique commands, at present it is 14 in this document. Converting these commands to a huffman tree would mean in this case that those 30 commands are expressed in at most 6 bits, converting possibly 5 or 6 bytes into 0.75 bytes, loosely a 6x compression ratio. That said, the tree itself would need to be stored, unless a pre defined dictionary is used based on the typical use case, and this is kept with the encoder and decoder instead. Searching online for the most commonly used 256 commands would produce a dictionary with codes at most around 8 bits long, but given how unlikely a tex file spanning more than 256 commands is, it may be more apt to have a set dictionary for the most prevalent 64 commands, and then append a custom dictionary as required for repeated code in a tex file that is not included in the pre set

dictionary, as I expect if someone was to use a rarer command, they may use it repeatedly, particularly if the user is proficient with latex and implements custom commands. Such a dynamic dictionary would have to be appended to the compressed file, costing storage. Then the rest of the encoding that does not appear in the dictionary is encoded according to some other method.

5. Idea 5

Having loosely fiddled with the pre-existing compression programs that exist to determine a sort of baseline for performance, the LZMA solution used by zipfile is the best compressor I have so far tested, compared to the python library solutions for deflate and bzip2, as well as the 7zip implementation of PPMd. For a 200KB dummy tex file, they all completed essentially instantly, which is nice even if not relevant. LZMA produced an incredibly compressed file, on an order of 60x compression ratio, though the dummy file was an approx. 30KB file with repeated contents. Regardless, with plenty of repetition the LZMA algorithm performs excellently. A second test with a more realistic dummy file led to a different conclusion, with bzip2 taking the crown, although the dummy passage was generated with zero repetition, which LZMA may be able to capitalise better on. Either way, LZMA performed strongly, and outperformed the contextual PPMd decisively. Moreover, given the speed of the implementations taking only a second, performing all compressions and returning the smallest compressed file is straightforward.

6. Idea 6

The final step is to develop a means of reconciling the potential insights about the tex format with the raw performance of the general compression techniques in practical use today. A primary step of reading in the file to produce a partial compression of predominantly the commands based on an existing dictionary, before a secondary action of passing this intermediary step to each existing compression algorithm may produce better results, though this is not guaranteed. Other than this, and the similar `\begin` and `\end` clauses, I do not believe that other amendments will produce result better than what the existing algorithms are capable of. However, I am tempted to try approaches like changing all characters to lower case, and inserting a unique symbol before capital letters, but this may be redundant for some compression techniques, as well as the fact this introduces a whole byte when the letter itself is just a byte anyway. Moreover, a caveat to an intermediate stage would mean that any replacements made would not be in binary and would have a minimum size of a byte, meaning that on one hand the dictionary could be quite large at 256 entries, but the cost is that every entry uses those 8 bits instead of potentially just a few bits for the more common entries. The alternative would be to interpret the file straight as binary in the intermediate step, but then this becomes more challenging. The only remaining step is to decide how to flag when the next byte requires a dictionary look up. Fortunately, there are many ascii characters that are unused normally, so a rudimentary check to be sure a character is unused should be sufficient. Then the following byte is used as a key in a predefined dictionary for the most common commands, and potentially the most common words as well. Having implemented now the encoder and decoder, the improvements over the existing methods are guaranteed, though minor. looking back, it may have been a better idea to appropriate 2 bytes to the top 65,000 words or so, and in the case of a word not being in the dictionary then having a pair of markers for the very rare word that is not in the top 65,000. This would perhaps produce a script of length 2n, where n is the number of words, or on average, a 4.7/2 times improvement, since the average word is 4.7 characters long. With a typical compressing algorithm on top of this, perhaps could perform drastically better than the rudimentary solution I have concocted, which also uses 2 bytes per shortened phrase. However somehow aggregating 65,000 words is an impressive feat in itself.