Codes and Cryptography - Data Compression

1 Introduction

Having implemented and experimented with multiple algorithms including: LZW followed by huffman encoding (specifying the exact bits stored and with efficient tree storage), BWT followed by move to front transform and huffman encoding, PPM and PAQ(via a python package). It became clear that PAQ was by far the best and likely unbeatable by any other implementation. Therefore, I decided to use PAQ as my fundamental algorithm, focussing my ideas around suitable preprocessing. Please also note that my code can take quite a long time to run, likely over 10 minutes and potentially over 30.

2 Idea 1 - StarNT

StarNT, first proposed in [1] by Sun et al, works on the principle that representing words in the text with smaller combinations of symbols will lead to an increase in compression by PAQ. This would ideally involve mapping the most frequent words found in the text to the smallest representations used. Sun et al try to approximate this by using a predefined dictionary of the 143,000 most frequent words in the english language, the first 312 stored in order of their frequency, the rest in decreasing order of length. They then map each word to a specfic symbol:

- The first 52 are mapped to the letters a, ..., z, A, ..., Z
- The next 52*52 are mapped to the letters aa, ..., az, aA, ..., aZ, ba, ..., ZZ
- The final 52*52*52 are mapped to the letters aaa, ..., aZa, ..., aZZ, baa, ..., ZZZ

The text, split into words according to spaces, is then iterated over and every word encountered that exists in the dictionary of frequent words is substituted with its corresponding symbol. Due to the fact that not all words will exist in the dictionary, an additional symbol must be added in such cases. The original paper does this by prepending any word not in the dictionary with a flag, the '*' symbol. This then also leads to the need for an escape flag where a flag is used in the original text, for this I use '/'. My implementation follows this idea, however, after experimentation it became clear that sorting all but the first 312 words by length instead of frequency was actually detrimental. Therefore, I ordered words only by frequency in my implementation.

3 Idea 2 - Improved flags

Given that words starting with a captial letter or made only of capital letters convey almost the same information as their lower case equivalent it does not make sense to store non lower case words in the dictionary previously described. Instead flags can be used to indicate what transform must be applied to a lower case word to get that found in

the text. Sun et al. [1] do use this idea. If a transformed version of a word present in the dictionary is encountered then the corresponding symbol for that word is appended to the output stream. Then a flag identifying the required transform is appended after this symbol. If the transform is making the first letter a capital a '~' is appended. If the transform is making the entire word capitalised then a ' ' (backwards apostrophe) is appended. However, this can be greatly improved with a very simple modification. [2] use this same process but instead of appending flags to the respective symbols of lower case words, they prepend these symbols, also placing a space between the flag and the symbol itself. This makes these flags easier to predict since they usually follow some punctuation while the added space improves the likelihood of longer contexts being found, making it more similar to that of the lower case equivalent in the middle of a sentence.

4 Idea 3 - Alternative symbols

Sun et al. state that compression is most effective when only alphabetical symbols are used, hence the need for the '*' flag, distinguishing the symbols of words in the dictionary from untransformed words. Skibinski et al. show that this is false. Instead they recommend only using symbols not found in the original text, given all symbols are ascii this is anything greater that 128. This removes the need for the '*' flag and ultimately leads to large reductions. Specifically, Skibinski et al. suggest using the ordering 128-228, 229-237, 238-246, 247-255 i.e in python 0 would be chr(128) while 101 would be chr(128)chr(229). Through experimentation I have found that this ordering is very dependent on the size of the input file. However, for large files I have found that this ordering is generally best and, therefore, use it in my implementation.

5 Idea 4 - Improved word seperation

After seeing some of the words not present in the dictionary printed to the terminal it became clear that the main reason for this was because of punctuation being attached to the word i.e 'self;' while the word 'self' was in fact in the dictionary. This was due to words being defined as blocks of characters seperated by spaces. To solve this, for every non alphabetical symbol I inserted two spaces, one either side of the symbol. This now meant that when the split operation was applied all words were either only alphabetical characters or all non alphabetical characters. This led to a large increase in compression. Annoyingly, after implementing this I then later realised that [2] had also suggested this exact same idea further on than I had read in their paper.

6 Idea 5 - File specific dictionary adaptations

An ideal dictionary would be ordered according to the word frequencies of the specific input file it was being used for. However, this would then require storing this dictionary in the compressed file every time, making it counterproductive. An alternative was to calculate the ideal dictionary ordering and then, while beneficial, apply and store transforms on the original dictionary (described in section 1) that would move it closer to this ideal dictionary. For each word in the input text I calculated Ideal_Sizes[word] = (Length of symbol for word according to ideal dictionary) * (frequency of word in input text) and Current_Sizes[word] = (Length of symbol for word according to original dictionary) * (frequency of word in input text). With these values I was then able to calculate

Under = max(Current_Sizes[word] - Ideal_Sizes[word]), the word which if encoded with its ideal size would lead to the greatest reduction in output stream size and Over = max(Ideal_Sizes[word] - Current_Sizes[word]) the word that due to being encoded according to the default dictionary, wastes the most space. Swapping Under and Over moves both Over and Under closer to their positionings in the ideal dictionary, without displacing other words. The locations of the words swapped are then appended to the end of the token stream for decompression. This is repeated until length(token stream) + length(swapped) stops reducing after a swap. This led to a significant reduction in the file sizes, especially for files that made use of lots of usually uncommon words.

One alternative transform that I also tried was instead of performing a swap between Under and Over, simply moving Under to the front of the dictionary similar to a move to front transform. This required only storing one additional value per transform instead of two but also led to the displacement of following words. In some cases this was more effective but on the whole the swap transform would appear to be better.

7 Idea 6 - LZW for file specific dictionary

Continuing with this motivation of moving towards an ideal dictionary I instead tried to use lzw for its generation in each file. The reasoning behind this was that the most frequent blocks of text would all use the same dictionary entry, therefore, making their encoding the most frequent in the overall encoding. This preservation of frequency would enable PAQ to work just as effectively but the encoded symbols used would be smaller and there would be no need to store a dictionary. However, this was obviously wrong as LZW leads to lots of specific sequences being used meaning that both the size of tokens increases rapidly and the frequency of words is not represented accurately. To try to fix this I limited what could be encoded in the LZW dictionary, specifically only alphabetical characters, leaving non alphabetical characters in the stream. This would limit the words present in the dictionary to only those used in the text and their prefixes. Unfortunately this also was not effective likely still because of the smallest encodings being wasted in representing prefixes and the fact that the most frequent words are not necesssarily encountered first. Due to the reduced dictionary size I thought that a large number of iterations of idea 5 could also be performed, however, this had little effect. Therefore, I did not use this idea in my final implementation. With more time I would also explore starting the LZW with a larger dictionary and with the move to front transform described in idea 5.

8 Packages used

Requires:

- pip install paq, see https://pypi.org/project/paq/ for more
- pip install wordfreq, see https://pypi.org/project/wordfreq/ for more

References

[1] Welfeng Sun, Nan Zhang and A. Mukherjee, "A dictionary-based multi-corpora text compression system," Data Compression Conference, 2003. Proceedings. DCC 2003, Snowbird, UT, USA, 2003, pp. 448-, doi: 10.1109/DCC.2003.1194067.

[2] Skibiński, P., Grabowski, S. and Deorowicz, S. (2005), Revisiting dictionary-based compression. Softw: Pract. Exper., 35: 1455-1476. https://doi.org/10.1002/spe.678