COMP3491
Data Compression Summative Report

Jack Byrne
znfw19@durham.ac.uk

2021-01-09

1 Abstract

For this summative project I implemented the Prediction by Partial Matching ("PPM") algorithm, using Adaptive Arithmetic Coding ("AAC") as the core encoding scheme. My implementation consistently achieves a better compression ratio than {zip, gzip, xz} on English language TEX documents.

2 Implementation

2.1 Symbol Alphabet

All compression algorithms using PPM/AAC operate on a symbol alphabet. An intuitive choice is to use 256 symbols, one for each possible value of a given input byte. In any case, the encoder creates two additional symbols, one to indicate the end of the message (`<EOF>`) and an escape symbol (`<ESC>`), the role of which is discussed further later in this report. Thus, in this schema, the alphabet ultimately has 257 symbols.

This approach is reasonable effective and has the benefit of being completely ambivalent towards the structure of the input. However, since my encoder was designed specifically to compress English language TEX files, it makes sense to capitalise on this prior knowledge about the likely nature of the input data. It is a sensible inference that most of the bytes in the input data are likely to be letters of the Latin alphabet, in which case they will fall in the second quarter of byte values (put another way, their byte values will begin with `01`). By a happy coincidence, the most common TEX syntax characters "{", "}", and "\" also fall within this range.

I therefore designed a more sophisticated scheme for mapping bytes into my symbol alphabet. Rather than assigning a symbol to each byte value, I use just 64 basic symbols, which by default correspond to the byte values 64 to 127. I then employ two shift symbols, `<S1>` and `<S2>`, to map the typically rarer bytes from the first, third, and fourth quarters of byte value space into the smaller symbol space. Byte values from the first quarter are encoded with a preceding `<S1>`, those from the fourth quarter are preceded by `<S2>`, and those from the third quarter are preceded by `<S1><S2>`. This reduction in the size of the symbol alphabet significantly improves coding efficiency, improving compression ratio by as much as a quarter on plain English files.

In principle, one could carefully engineer an optimized mapping of byte values to symbols, to further improve compression. For example, an obvious flaw in my scheme is that the common space character does require a shift character, while the DEL character, which does not usually exist in text files, does not. However, I opted not to do this, as I felt that on balance it would have only modest benefit, be inelegant, and lack generality.

2.2 Adaptive Arithmetic Coding

At the core of my compression algorithm is an implementation of Adaptive Arithmetic Coding. Implementing AAC robustly and efficiently is not easy. The core of the algorithm, conceptually speaking, involves performing operations on arbitrary rational numbers in the range $0 \leq n < 1$. Of course floating-point numbers are not precise enough to handle encoding more than a few bytes, so practical implementations of AAC must use integers as "sliding windows" into the arbitrarily long fixed-point numbers upon which conceptually they operate. A particular difficulty comes in ensuring that the algorithm does not underflow the precision of these "windows" as it successively selects smaller and smaller sub-intervals of the real line at each encoding step. The solution to this difficulty is interval-doubling; a systematic method for ensuring the current interval always spans at least half the space from 0 to 1 - crucially, without requiring modification of more than a few bits at the "working end" of the encoded bitstring.

In writing my implementation of this I was greatly aided by the superb paper "Practical Implementations of Arithmetic Coding" [**howard1992practical**], which describes an effective interval-doubling scheme. This scheme prescribes three different methods of doubling the interval. All involve doubling the distance of both the left and right ends of the current interval from some point on the real line. The first, for when the current interval is wholly below ½, entails doubling the distance from 0; such that [0.2, 0.3] becomes [0.4, 0.6] and so on. This can be implemented simply by left-shifting off the top 0 of both interval ends. The second, for when the current interval is wholly above ½, involves doubling the distance from 1; and can similarly be implemented by left-shifting the top 1 off both ends. The third, for when the current interval is wholly contained within the interval $\frac{1}{4} \leq n < \frac{3}{4}$, involves doubling the distance from ½, and can be implemented by left-shifting off the second highest bit while retaining the highest bit, again for each end of the interval. In all cases it is most efficient to shift 0 onto the low end and 1 onto the high end to keep the interval as wide as possible. It is necessary to keep track of the bits that are shifted off as they are used to build the encoded bitstring.

I configured my AAC implementation with a "window magnitude" of 32; that is, the working range of integer values for my interval ends was $0 \leq n < 2^{32}$. I found that this was enough to provide the necessary resolution given the size of my symbol alphabet, and that higher values of this parameter made the algorithm slower without yielding improvements to compression ratio.

2.3 Prediction by Partial Matching

Having implemented AAC, I proceeded to enshroud it with an implementation of Prediction by Partial Matching, an algorithm for assigning likelihoods (and thus, AAC intervals) to symbols in varying contexts. The core data-structure for PPM is the trie, which I wrote a serviceable implementation of using Python dictionaries as the backing structure. A nicety of my trie implementation is that it allows for contexts of arbitrary length, and indeed, I made good use of this when experimenting with variable-length context orders, although without any great success!

The most important parameter for any conventional implementation of PPM is the initial order - that is, the longest context with which the algorithm attempts to encode a given symbol. The literature tends to suggest that for a bounded-order algorithm, an initial order of 5 is optimal [**cleary1997unbounded**], and this was borne out by my own experimentation on English language TEX files. As such, it is the chosen parameterisation in my implementation.

2.4 Escape Encoding

A classic question when it comes to implementing PPM is how best to assign a frequency to the escape symbol `<ESC>`. A popular approach, referred to in the literature as "PPMD" [**howard1992practical**], prescribes the following method: when a new symbol is encountered in a given context, assign it a frequency of ½, and increment the frequency of `<ESC>` by ½, such that the sum weight added to the frequency table is 1, just as it is when any other symbol is decoded.

I found PPMD to be relatively effective, but empirically I achieved higher compression ratios with an alternative approach that I believe to be novel. In my method, when a new symbol is encountered, I set its frequency to 1, but I increment the frequency of `<ESC>` by $(6 - order) \times \frac{1}{4}$, where order is the length of the current context. This means that when a new symbol is encountered, the frequency of the escape symbol is increased more substantially for shorter contexts than for longer ones. The intuition here is that future escapes are more probable in smaller, more generic contexts than in more specific ones - `ug` could plausibly be followed by a great deal of un-encountered symbols, but `thoug` is unlikely to ever be followed by anything other than `h`, so a large escape frequency is inappropriate.

2.5 Symbol Exclusion

One of the most effective improvements one can make to the standard PPM algorithm is the inclusion of exclusion sets [**steinruecken2015lossless**]. The idea is quite a simple one; if the algorithm escapes from a given context, then the current symbol cannot be any of those which are present in the frequency table for that context, so they should be excluded from consideration in the contexts below. For this reason, as the algorithm "falls through" from each context to the next, it should maintain a set of excluded symbols, so that it can divide the interval exclusively between those symbols which can still possibly be encoded.

This simple and elegant improvement to PPM was easily implemented and, empirically, improved compression ratios on my test files by around 20%, at the cost of a modest though not insignificant speed decrease.

2.6 Ineffectual Ideas

Finally, I discuss some interesting ideas that did not yield a compression improvement.

I implemented a Burrows-Wheeler Transform in the hope that this would enable PPM/AC to achieve a greater compression ratio. However, in actuality, not only was the transform slow on large files, but it decreased the performance of the encoder. I found that this was true even when I tried tweaking the parameterisation of the encoder to be more sympathetic to the transformed input. I think the reason this approach failed is because the transform actually strips more structure from my test files than it adds, but I would not rule out the possibility that I simply erred somewhere in my implementation.

I also tried implementing a frequency decay routine that periodically walked the context trie and decremented the frequencies recorded there, going so far as to delete nodes with very low frequency counts. Not only was this phenomenally slow, but it dramatically worsened the performance of the encoder. I think this is because the files I tested the encoder on were relatively homogenous in structure; that is to say, information learned early in the encoding process remained applicable right through to the end of the file. This approach may have been more appropriate on more heterogenous input data, for example email files with encoded image data appended to the end of them.