Compression algorithms R&D

By Yonatan Ehrenreich and Ilan Prais

September 24, 2022

1 Introduction

Today, there are many different types of compression algorithms. Every different type is best suitable for a specific type of data. In this project, we have researched and implemented some various compression algorithms on different types (such as entropy based, dictionary based, etc.) in order to compare their performance on a given text file, and find the algorithm with the optimal compression ratio. Additionaly, we added our own improvements, to make the algorithm more efficient.

2 Description of the algorithms

2.1 Run Length Encoding with Dynamic Optimal Repetition Storage size and Burrows-Wheeler Transform

On paper, Run-Length encoding is a fairly simple compression algorithm. Take a bunch of repeating characters, and convert them to 1 character and 1 number which represents the number of occurrences. This seems nice and simple, however on its own is not very efficient - it will only save space if the repeated occurrences are very common. Otherwise, the compressed file will be actually larger because for most characters we save more information than before.

The first improvement we did, is make use of the *Burrows Wheeler Transform.* This transformation will take the text and return a new text and a short number (index), such that the new text contains many runs of similar characters. And the magic is, that the transformation is 1-1 and we can restore the original text only with the index (and the transformed text). So, If we apply *Burrows Wheeler Transform* on the text, then encode the index and the transformed text with Run-Length encoding, we can reach much better results.

After doing this, we noticed a slight problem - The number character repetitions vary a lot. For example, let's have a look at the string *aaaabcdeedxxxxxxx*. How do we choose in how many bits to save the repetition value? (the data which indicates how many times the characte has repeated). We could just take the longest repetition - which in this case is 7, hence 3 bits, and decide on 3 bits for every character. This would work, but is very wasteful. Many characters with short or no repetition will consume empty space. Instead, out solution is to calculate the best optimal size for the whole data. We calculate the distribution of the amount of repetitions, and then go over all the repetition count storage size options (2 bits, 3 bits, ...), and for each option make an estimation of the size of the encoding (This can be done by summing the sizes for each [letter, num of repetition] pair given the repetition storage size). Then, we can choose the repetition storage size which gives the smallest encoding size estimation.

For example, for 100k characters of *dickens.txt*, the optimal count number size is 3 bits. So, for every [letter, num of repetition] we will save in total 11 bits (8 for letter, 3 for repetitions). The general encoding format will be

$$[rep-storage-size][letter][reps][letter][reps][letter][reps]... \qquad (1)$$

Notice that it is important to save in the encoding the repetition storage size we decided on, in order to be able to decode it later.

This method should be good, however we can notice that still some space is wasted - because we are using a constant reps size for all letters. i.e even for letter with only 1 repetition, we will save a high number of bits to represent it. (In out example, we will save 001 for those). In order to improve this, we can save for each letter the exact amount of bits needed to represent the rep size, and save the number of bits we saved before this, in a constant size. So now we will save always $log(reps)$, plus the exact amount of bits for the reps. The general encoding format will be

$$[letter][reps - size][reps][letter][reps - size][reps][letter][reps - size][reps]... \quad (2)$$

How can we decide on the constant size of reps-size? We can do something similar to before. Estimate the encoding size for each number of bits (1, 2, 3, ...), and the option with the smallest encoding size will be chosen.

2.2 LZ77 with Burrows Wheeler Transform

LZ77 is another lossless compression algorithm, published Abraham Lempel and Jacob Ziv (Together with LZ78). The general idead behind LZ77 is to find occurences of the same data, and saving that data just once, while the rest "point" to the original data. We decided to implement this alogrithm in a fairly simplistic way. There is a "window size" and a "forward size". The meaning is, while iterating the data, we will look for occurences up to *window size* letters back, and limit the match to be *forward size* long. This can let us set a limit on these values, in order to set a size limit to these "pointers". Each pointer is (number of steps back, length of the occurrence). If we can limit the size of this tuple, we can make the encoding size smaller. Finally, we transformed the data with *Burrows Wheeler Transform* for more efficient encoding

2.3 Arithmetic Coding With an efficient serialization method

Arithmetic coding is a pretty used compression algorithm, which is belong to the "entropy based" algorithms. The whole point of the entropy based algorithms is to attempt achieving the lower bound according to the **shannon source coding theorem**, which says that the encoding length of a lossless data compression algorithm must be greater or equal to the entropy of the source. So entropy based algorithms are trying to achieve that by basically looking at the distribution of the symbol occurrences, and encoding each symbol according to its probability.

We will not get in to the exact details of the algorithm because it is not the point of that paper, but will explain some optimizations that we did there.
For a short introduction - arithmetic coding basically works by calculating the symbol occurrences distribution, and by that, split the range [0, 1] to separate parts, one for each symbol, according to its probability, and by iterate on the input stream, iteratively read one symbol at a time, find its part on the original range [0, 1], squeeze its part to the size of the current range (which is initialized to be [0, 1] at the beginning, and iteratively updated according to the next symbol), and continue with the squeezed range as the current range. So, the result of the algorithm is the lower bound of the final range (or average between the two bounds), which is a huge floating point, as the fraction contains one digit for each encoded symbol.
So there is a problem - How can we save such a floating point with huge fraction, with a exact accuracy?

We implemented there an elegant solution - first, we treated that fraction as a ratio between two big integers, which gave us a maximal accuracy. So the real problem here is to serialize that number into a file, and deserialize it. Saving the numerator and the denominator into a file will be a bad idea - every one of them might be in half of the size of the data, so both together can easily reach to the size of the whole data, and even bigger.
So what we did is to save only the numerator. How can it be done?
For each symbol, we adjusted its frequency to be the (negative) exponent of 2 which is the closest to the

number. Then, as long as the sum of the frequencies was smaller than 1, we increased the frequencies of the frequent symbols by multiplying them by 2. So now all of the frequencies are exponent of 2.
It gives us an interesting thing - as I explained above, the algorithm works by taking iteratively the part of [0, 1] which is belong to the next symbol in the input stream (and its size equals to the frequency of that symbol), and squeezing that part into the current range, and continue iteratively.
So in each iteration, we take the next frequency (which its denominator is a power of 2), multiplying it with the current range size (which its denominator is also a power of 2, because it is actually the frequency of the previous symbol in the input stream), and summing it with the lower bound of the range (which its denominator is a number of 2, as this number equals to a sum of the range sizes of some other symbols, and as explained, each range size is a frequency, which its denominator is a power of 2).

So the final encoded result will be a fraction which its denominator is a power of 2, so we can actually save only the numerator, and instead of the denominator save its base-2 logarithm, so the total size that we have to save is a little bigger than the size of the numerator itself!

2.4 Adaptive Arithmetic Coding

One disadvantage of the arithmetic coding algorithm is that the frequency table (a table with the frequency of each symbol in the input stream) should also be saved along with the encoded result. The "adaptive" version of the algorithm is trying to overcome this - the algorithm is same as the original arithmetic coding, instead of that the frequency table is "learnt" during the encoding and decoding process, so it should not be saved. It is done by starting with equal frequencies for all of the symbols, and once seeing a symbol, increase its frequency by one.

So in the implementation of the algorithm, we used the same "trick" that we used in the arithmetic encoding, so the denominator of each frequency was a power of 2, so the encoded result could be saved efficiently, same as before.

3 Results

We used the compression algorithms above, to compress a subset of *dickens.txt*. The resuls are as following:

Algorithm	Run-Length	LZ77	Arithmetic	Adaptive
Compression Ratio	1.33	1.8	1.92	1.85

References

[Gad22] Ahmed Gad. Lossless data compression using arithmetic encoding in pytho and its applications in deep learning. 2022.

[Gui20] Tim Guite. Compression with lz77. 2020.

[rle22] Run length encoding and decoding. 2022.