File Compression Report

rxgd33

January 2021

1 LZ77

In order to remove word repetition redundancy, we use an LZ77 Encoder, whereby repeated occurrences of data are referenced to an already existing copy of that data earlier in the data-stream. They are referenced as tuples, $(offset, length, char)$ where $offset$ denotes how many characters ago the last occurrence was found, $length$ denotes the length of the occurrence, and $char$ denotes the next immediate character.

Using the algorithm as such allows us to take advantage of commonly repeated words in text, and therefore allows for high compression rates in files containing multiple occurrences of the same data. In the context of latex files, often we find certain expressions commonly repeated, for example, in this latex file, we use `\section` multiple times. LZ77 can exploit this redundancy and consequently achieve higher compression rates.

My implementation of this algorithm allows for an adjustable sliding window W and look-ahead buffer L, which we set the values to $2^{16} - 1$ and $2^8 - 1$ respectively. Using these values allows us to minimise the number of bytes we store our offset (2 bytes) and length (1 byte), whilst maximising the compression efficiency of LZ77.

2 Huffman Coding

Whilst LZ77 compression removes one type of redundancy, statistical compression methods such as Huffman Coding can reduce another. Huffman coding allows us to store more frequent characters with less storage, by constructing a set of prefix codes, whereby the length of the prefix code is determined by the frequency of the letter. In turn, we produce a set of

uniquely decodable codes. This set of uniquely decodable codes are generated from a Huffman tree, that we calculate by using letter frequencies.

Consider the ASCII value of 'e', '1100101'. As this is the most common letter in the English language, it is likely that rather than storing 'e' as '1100101', we can map it to a codeword, i.e. '110' and hence reduce the amount of storage we provide for this character. NOTE: in my implementation of Huffman coding, codewords are created based on the frequency of character occurrences in the input-file rather than more general assumptions on the frequency of letters in the English language.

3 Canonical Huffman Coding

Extending on our idea of Huffman Coding, I have chosen to implement Canonical Huffman coding rather than just basic Huffman coding. In order to reconstruct characters from their Huffman codeword, we must reconstruct the Huffman tree. Reconstructing this Huffman tree requires the encoder to transmit a sequence of bits, representing the arbitrary tree.

In Canonical Huffman coding we go with the knowledge that characters are mapped in order of their codeword bit lengths, where if two codewords have the same bit length we sort them according to their lexicographical order. Using Canonical Huffman coding, we can encode this tree in much fewer bits (more precisely we simply encode the set of unique characters along with their codeword lengths) and therefore reach higher compression rates.

4 Deflate

Similar to the Deflate compression algorithm, my file compressor works by combining both LZ77 and Canonical Huffman Encoding. We first run the file contents through my LZ77 algorithm, which creates tuples $(offset, length, char)$. With the knowledge that the maximum value of the offset and length is $2^{16} - 1$ and $2^8 - 1$ respectively, we can encode all offsets as $2n$ bytes and all lengths as n bytes, where n is the total number of tuples.

My Canonical Huffman encoder is applied to the concatenation of all n $chars$, significantly reducing the size, and allowing further compression. The encoder applies a header to each of the offsets and length bit-streams,

indicating their length to allow my decoder to break the bit-stream down into it's constituent parts.

5 Priority Selection

Whilst LZ77 is extremely efficient on large files (as repetition often increases the bigger the file), on smaller files LZ77 can possibly 'compress' a file into larger version of it's original form. In order to resolve this, I compare the results of both my deflate-based algorithm and a standalone Canonical Huffman encoding. If the standalone Canonical Huffman encoding is smaller, we use this, rather than the deflate-based one.

To signal to my decoder which encoding scheme is used, a single byte is attached to the front of the compression, allowing my decoder to identify which encoding scheme has been used.