Codes & Cryptography Coursework - Compression

gwwb64

January 2021

1 Introduction

This compression algorithm is designed for LaTeXfiles. A top level overview of the structure of the algorithm and the lz files it outputs are provided below. The algorithm is reversed for decompression.

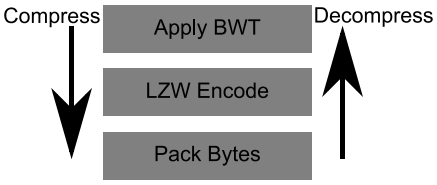

Figure 1: Algorithm Structure

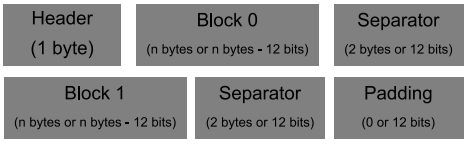

Figure 2: Output (lz) File Structure

2 Point 1 - Burrows-Wheeler Transform

The first phase of the algorithm applies the Burrows-Wheeler Transform (BWT) (Burrows and Wheeler 1994) to a block of binary data. This transform takes a block and lexicographically sorts circular right shifts of the data within the block. The implementation used here uses a single character to denote the end point of the string. This is required to denote where the rotations start for the inverse process in decompression. An alternative would be to pack the index of the start point within the block at the end of the output block. This would make the implementation a lot more complex and would require a additional bytes at the end of the block (depending on the max block size) but may be advantageous. The end marker we have chosen is the NULL byte (0x00) which does mean the NULL byte cannot be in the original LaTeXfile but it seems unlikely to be a common character anyway since it is unprintable. This algorithm can be quite slow so a small efficiency improvement has been added via the use of suffix arrays of rotations for encoding. The BWT moves identical characters (or in our implementation, bytes) next to each other which can improve compression ratios.

3 Point 2 - Lempel–Ziv–Welch

The second phase applies the Lempel–Ziv–Welch (LZW) (Welch 1984) dictionary algorithm. This starts with an initial dictionary size of 257 representing the 256 ASCII characters + a reserved 0th dictionary index which will be used to print NULL bytes between blocks. This has been specifically chosen so that the NULL bytes generated by the BWT will be encoded as 0x01 and will not conflict with our required block delimiter. This does mean that the entirety of ASCII cannot fit in 1 byte but this is not a problem since our dictionary maximum size will be greater than a single byte. LZW reads a character and if the previously stored string with the new character appended is found in the dictionary, the new character is appended and the process continues. If it is not found, the previously stored string is output and as long as the dictionary is not full, the old string with the new character appended is added to the dictionary. The new character is then stored in the string. Our dictionary is optimised in the compression direction via the use of a dict structure which provides $O(1)$ lookups of strings. A regular list is used in the decompression direction which provides $O(1)$ lookups of indexes. While LZW does not require lookaheads so can operate on individual characters at a time and hence process a whole file, we are limited by the BWT block size so must run LZW on the block level.

From tests, this works well but under some circumstances may penalise compression. Ideally, the LZW algorithm could be improved in future by making it context aware (like PPM is) with different dictionaries for different contexts which can increase number of symbols encoded in a dictionary entry and also decrease the number of bits required to store an entry if the max. dictionary size can be lowered. The compression ratio could also be monitored and the dictionary reset if it falls too low like LZC, however from testing, the compression ratio never fell too much in any .tex file.

4 Point 3 - Byte Packing

During the LZW phase, output is sent to a custom buffered output system. Depending on the parameters the algorithm decides on, the indexes generated by LZW will either be packed into 2 bytes and written directly to the output file (without being buffered) or packed into 2 bytes and held in the buffer for another 2 bytes to arrive. Big-endian is used for packing. For small files, using a dictionary of size 65536 and packing that into 2 bytes when only a few thousand entries are used is wasteful. For this reason, we can use a smaller dictionary of size 4096. However, to be worthwhile, this requires packing the indexes into bit sequences of length 12 (a 25% reduction). Since 12 is not a multiple of 8, we must wait for enough data to create 2x12 bits (3 bytes) before writing to file. When there are at least 4 bytes in the buffer and a write is attempted, each stored block of 4 bytes will be converted into 3 bytes by removing the 4 most significant bits of each byte pair. For example, the byte sequence 00001011 10010011 00000110 11111111 will be manipulated using bit shifts, bit masks, bitwise and/or to become 10111001 00110110 11111111. When reading the compressed file, this packing process must be reversed. In this scenario, since the buffer can remain unfilled if there are an odd number of indexes to output, it must be flushed at the end of the compressor. The flush just adds a zero to the end which pads out the remaining byte and a half.

5 Point 4 - Parameter Selection

There are 3 important parameters that the compressor sets to predefined values depending on the file size. BWT_CHUNK_SIZE represents the size of a block that will be read from the original file at once and passed to the BWT algorithm. In the current version of the compressor, this is a fixed value of 64KB. This was chosen experimentally as a trade-off between time to compress a file and the output compression ratio. Ideally we would like to run the BWT over the entire file but due to the circular shifting and sorting steps, the time taken is impractical. It is also impractical to store an entire file in memory at once for large files especially considering that a naive BWT implementation would require $O(n^2)$ space. This particular value ensures that files in the dozens of KB range can be processed in one block while also approximately maintaining the same compression ratio for similar larger files which require multiple blocks. MAX_DIC_LENGTH represents the maximum length of the dictionary. For files smaller than or equal to the BWT_CHUNK_SIZE, MAX_DIC_LENGTH is set to 4096. For larger files it is set to 65536. This value will also change how the bytes are packed (see Section 4). The settings for this were determined experimentally. It may seem like a good idea to just set a hard limit at 4096 and pack into 12 bits each time. However, tests showed that for larger files, having a larger dictionary actually outweighed the benefits of a 25% reduction in index length. The final parameter is the READ_WIDTH which is 2 or 3. This is used by the decoder to determine how many bytes must be read at once to get a complete representation of values. If the width is 2, 2 bytes are read and these are unpacked as a single index value $0 <= x < 65536$. If the width is 3, 3 bytes are read (for the 12bit system) and these are unpacked as 2 index values $0 <= x < 4096$. These 3 settings are combined into a single parameter FILE_TYPE which is stored as a single magic header byte and is the first byte in the compressed file. This magic byte must be read by the decoder to determine settings but must not be passed to any decompression algorithm.

6 Point 5 - Special Behaviour

Given the construction of this algorithm, there are a couple of notable quirks. Firstly, when decompression is being performed, LZW can sometimes end up requiring the index that is currently being processed. To overcome this, if the index matches the length of the current dictionary, the existing string will have the first character appended to it and this will be added as an entry to the dictionary too. The current string is stored into the previous variable as normal if this happens. Since this system is limited by BWT_CHUNK_SIZE as to how much data can be processed at once, there needs to be a way to determine where one block starts and the next one ends. Naively, it is possible to just read a BWT_CHUNK_SIZE block in the decompression algorithm. However, due to the construction of our algorithm, after the LZW step, the block size will be smaller (this is the point). Therefore, where a block ends can be determined by outputting a NULL byte as a separator after each block. However, before this can be done, it is essential that any remaining string generated by LZW is output. The string used internally by LZW must be reset after each block. While this may seem like a natural place to do so, since this is not an LZC implementation, the dictionary is not reset. Resetting the dictionary after each 64KB block was attempted, however experiments showed a very poor compression ratio. This may be beneficial if each block had very different data but English LaTeXclearly does not vary enough.

7 Point 6 - Removed Parts

Run-Length Encoding (RLE) may seem like a natural choice when using BWT. The BWT generates long sequences of repeated values which is perfect to encode by reducing each value to a $value + numberofrepeats$ pair. This was implemented in a version of the algorithm between the BWT and the LZW step. Unfortunately, testing showed that this performed incredibly poorly. This is probably because while there were many runs which could be reduced in length, there was limited correlation between neighbouring runs. This meant that LZW ended up just outputting indexes which referenced one value followed by indexes representing a number of repeats. This actually increased the "compressed" file size beyond the original input by an order of magnitude at least. A BWT + RLE encoding pair alone did reduce the file size but not as much as the BWT followed by the LZW. RLE is not a suitable part of this setup. Huffman Coding was also attempted after the LZW step but this also proved worthless. For small files, it often increased the compressed file size above what was achieved by BWT + LZW + Intelligent Packing. For large files ($\sim 400KB$), an improvement of only 3KB was recorded so the additional processing was wasteful. It would be better to add the aforementioned context awareness to LZW instead.

8 Conclusion

Some experiments show that this compression algorithm performs decently well though is worse than the optimised version of DEFLATE commonly used. Additionally, even with the BWT block size limits, the BWT process is slow so running a test on large3.tex took 3-4 minutes on a 6 core 3.6GHz processor. Ultimately, this compression algorithm will not beat reasonable compression settings for the common algorithms such as DEFLATE and definitely won't win against PPMd. However, it does sometimes perform better than a simple DEFLATE and is an interesting construction that could be adapted to become faster (with BWT changes) and improve the compression ratio (with context aware LZW).

Name	Original Size	Compressed Size	Ratio
lecture20.tex	18464	6991	2.64
test.tex	1497	1186	1.26
large.tex	415194	164693	2.52
large3.tex	1245582	456461	2.73

References

Burrows, M. and D. Wheeler (1994). “A Block-sorting Lossless Data Compression Algorithm”. In: vol. Technical Report 124. Digital Equipment Corporation.

Welch (1984). “A Technique for High-Performance Data Compression”. In: *Computer* 17.6, pp. 8–19. DOI: 10.1109/MC.1984.1659158.