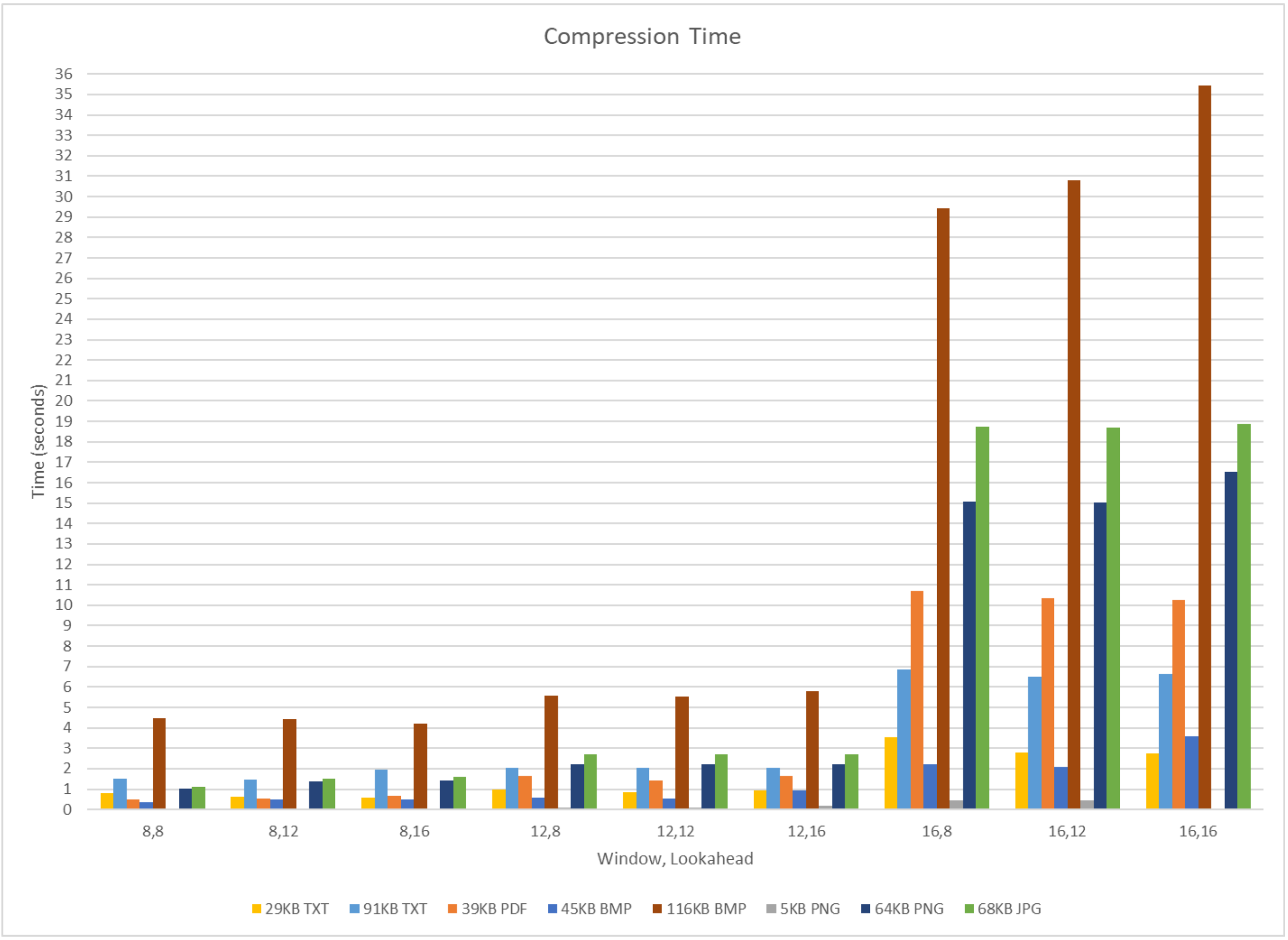
Digital Communication 2018-19
Lempel-Ziv 77 Compression

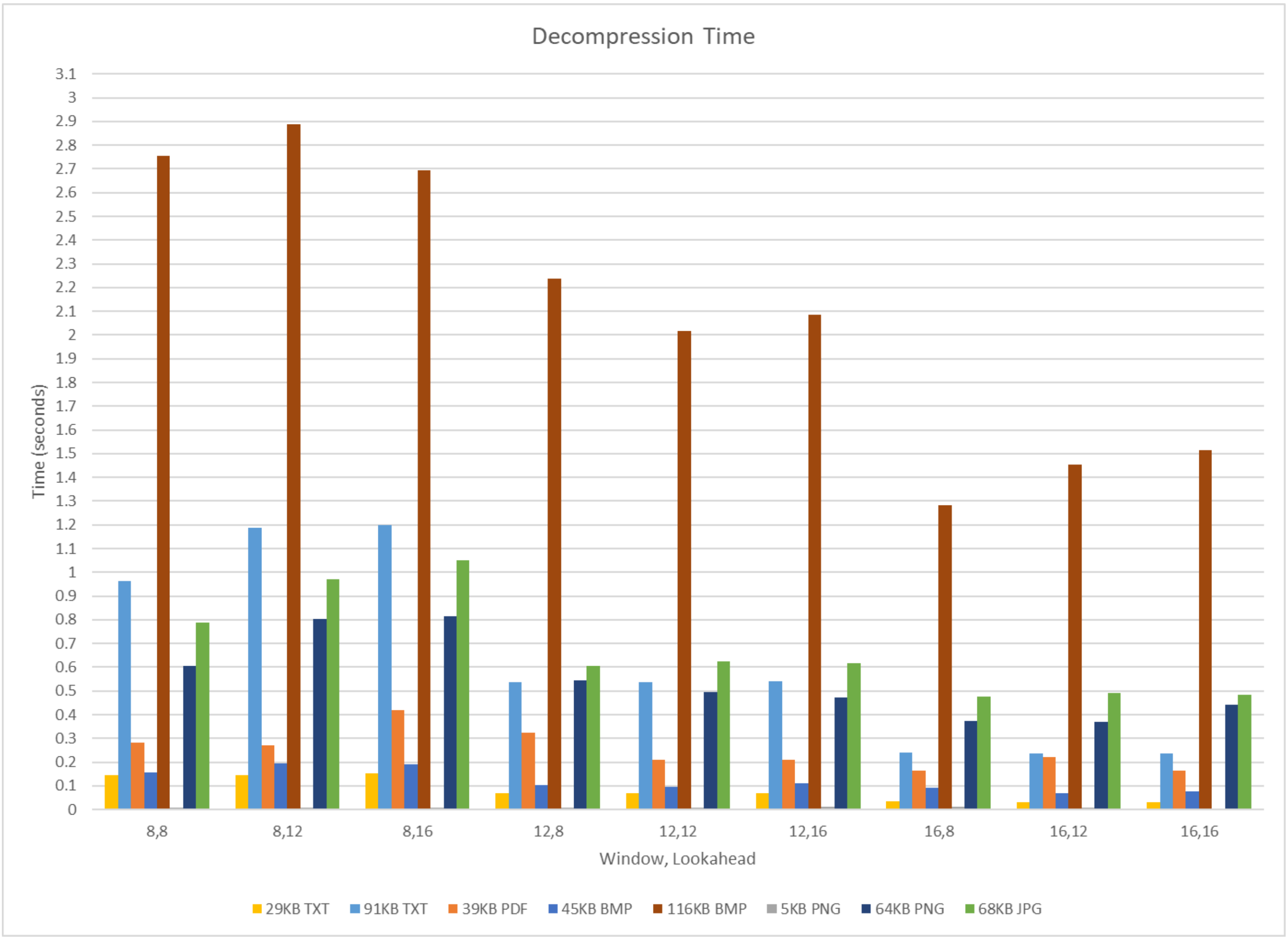
Running Time of Encoder:

The encoder was run on various file types and sizes with a Window and Lookahead of 8 bits, meaning they could contain up to 255 characters each. Generally, files required the same amount of time to compress, around 0.1-1 second, except for a 116KB BMP file. This was likely since this image file did not compress well, and as such would have taken longer to process due to having to constantly search through the Window more often as there were less frequent matches. The encoder was then run using various Window and Lookahead pairs of 8, 12, and 16 bits. Differing the Lookahead size did not have much effect on the run time, with the only large and noticeable difference being when the Window was 16 bits and the encoder was run on the BMP file mentioned previously. On the other hand, the Window size had a significant impact on run time when it was changed, regardless of the file. The run time increased exponentially as the Window size increased, with an extremely large increase for all files when the Window was put to 16 bits. There was also a growth in run time with an increase in file size, although some larger files, such as a 512KB BMP took much shorter times than others, though this was dependent on how well the file compressed. The following file comparisons are with a Window of 16 bits and a Lookahead of 8 bits. TXT files ran relatively efficiently at smaller file sizes, with files of around 90KB taking an average of 7 seconds to compress. Larger files took significantly longer however, with a 500KB file taking an average of 140 seconds to compress. PDF files took significantly longer than TXT files, with a 39KB PDF taking an average of 4 seconds longer than a 91KB TXT file. BMP files had rather extreme results. A 45KB BMP took an average of 3.5 seconds to compress, which was fast compared to other files, but this was likely due to the file compressing well. On the other hand, the 116KB BMP file was the slowest of all the files, averaging at 29.5 seconds to compress, a full 10 seconds slower than the next slowest file type. The 512KB BMP, however, took an average of 51 seconds to compress, which was a full 89 seconds faster than a similarly sized TXT file. JPG and PNG files also appeared to be relatively slow in compressing, excluding the 5KB PNG, but this was likely due to it being such a small file size. Running the encoder on more/less powerful machines also gave faster/slower compression times, respectively, though this was to be expected.



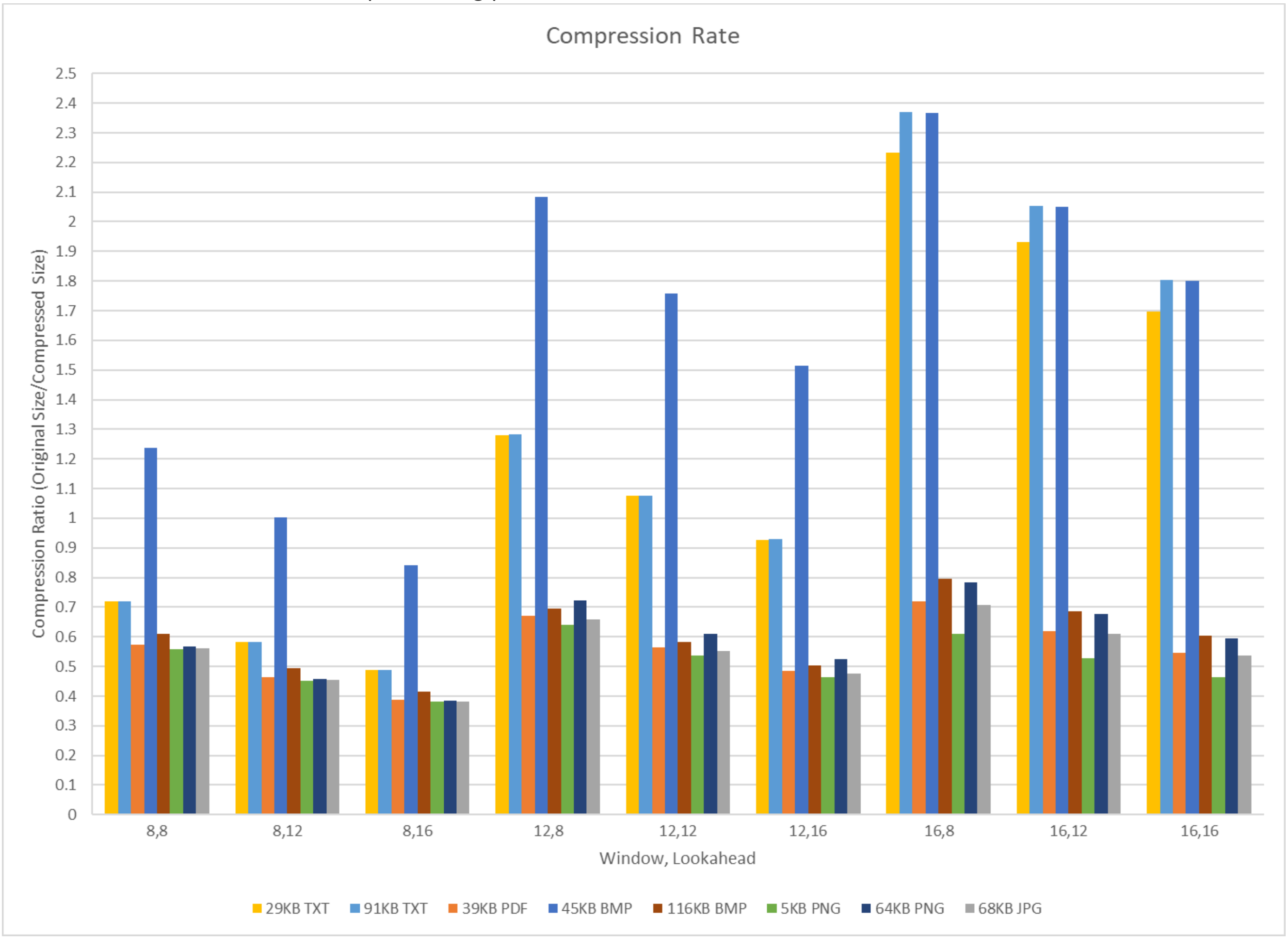
Running time of Decoder:

On average, the decoder was significantly faster than the encoder, with the run time of the files shown on the graph having a maximum of 3 seconds, whilst the maximum run time of the encoder had been 36 seconds. In all cases the decoder ran faster than the encoder, however there was a more distinct relationship between the run time of the decoder and the actual compression rate. This would have been due to the number of tuples that would have to be read by the decoder in order to reach the original file, as more compression would mean less tuples for the same file than with less compression. It was seen that the average speed of decompression changed proportionally with the actual compression rate of the file, regardless of file type or file size. This meant that files which compressed more were able to be decompressed much faster than the same files which had been compressed less. As file size increased however, the run time of the decoder also increased. For example, the 500KB TXT took an average of 93 seconds to decompress, and the 512KB BMP took an average of 43 seconds to decompress. This difference was, again, most likely due to the compression rate of the files, as the TXT had a compression rate of 2.45 while the BMP had a compression rate of 3.94. Much like the encoder speed, this was also faster/slower on stronger/weaker machines respectively.



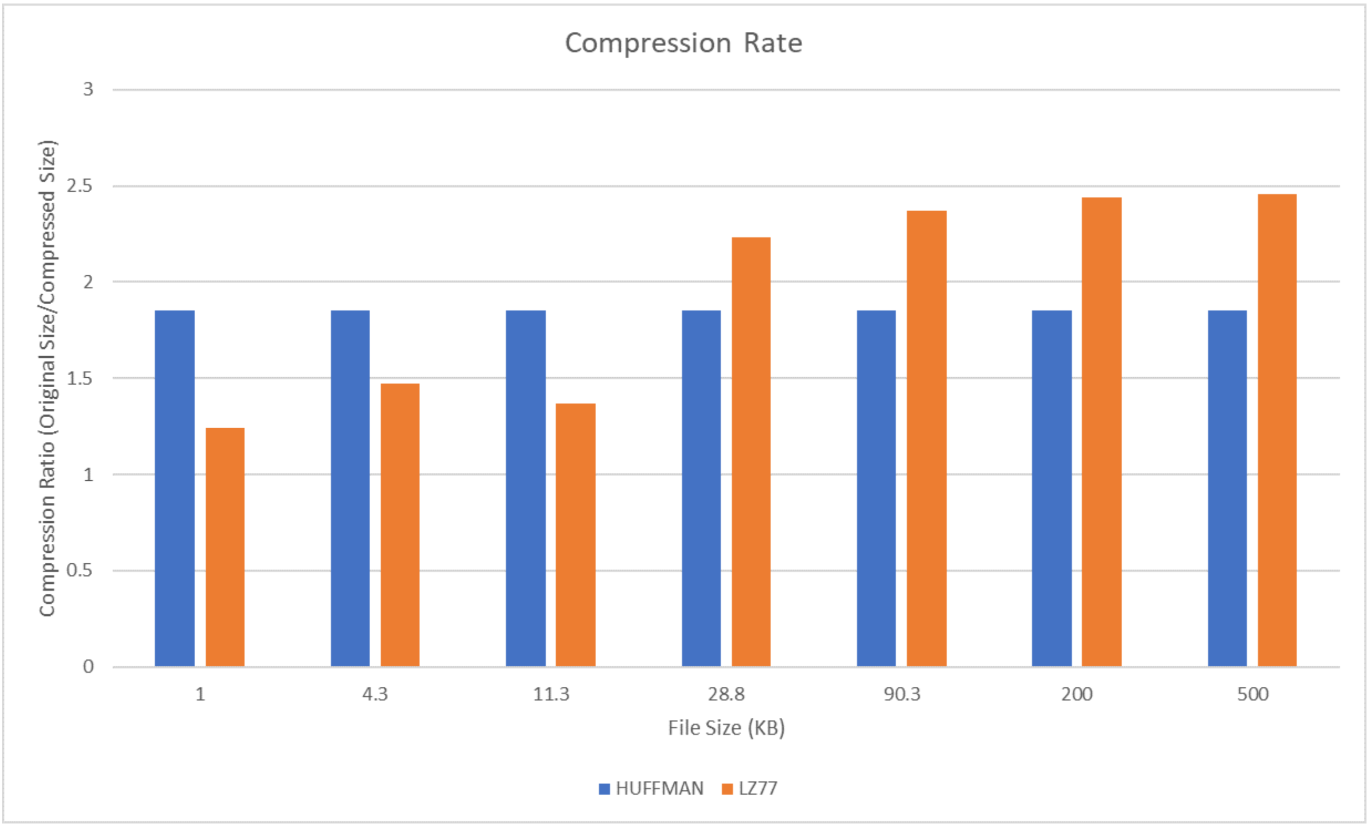
Compression Ratio:

The best average compression rate was achieved when using a 16-bit Window, and 8-bit Lookahead. Varying these values gave differing results for different files however, with some smaller files achieving better compression rates when run using smaller Windows, though on average 16 and 8 gave the most consistently good results than the other batch tested values. Varying the Lookahead gave different, but usually worse, compression on all files, with 8 being found as being the most efficient. When tested, the TXT files had the most significant increase in compression rate, especially when moving from 12-bits to 16-bits. One point of note was that for 8- and 12-bit Windows, both TXT files had roughly the same compression rate, but at 16-bits the compression rate of the smaller 29KB TXT started to fall behind the larger 91KB TXT, likely due to the Window being slightly too large for the smaller TXT for it to efficiently compress. At a 16-bit Window the 91KB TXT also achieved the same compression rate as the 45KB BMP, which had previously had a significantly larger compression rate at smaller Window sizes. The greatest compression rate achieved was on the 512KB BMP, which achieved a compression rate of 3.94, compressing to 130KB. However, this was a graphical image of a rainbow, and as such would have had many similar bytes and so would have compressed extremely well. Other files types did not achieve very good compression rates, regardless of size, and the encoded files were frequently larger than their original counterparts. This was also tested on various other PDFs as well as DOCX files, but with similar results. One reason for this would have been due to certain file types, such as JPGs for example, already being compressed, and so being difficult to compress further. Running on DOC files yielded differing results, again, dependent on the actual content of the file. As an example, one 245KB DOC Lorem Ipsum sample file had a compression rate of 2.4, while a 622KB DOC of a book had a compression rate of 0.8. Running the encoder on different machines had no impact on the actual compression rate, as it was reliant only on the data in the files themselves, rather than processing power.

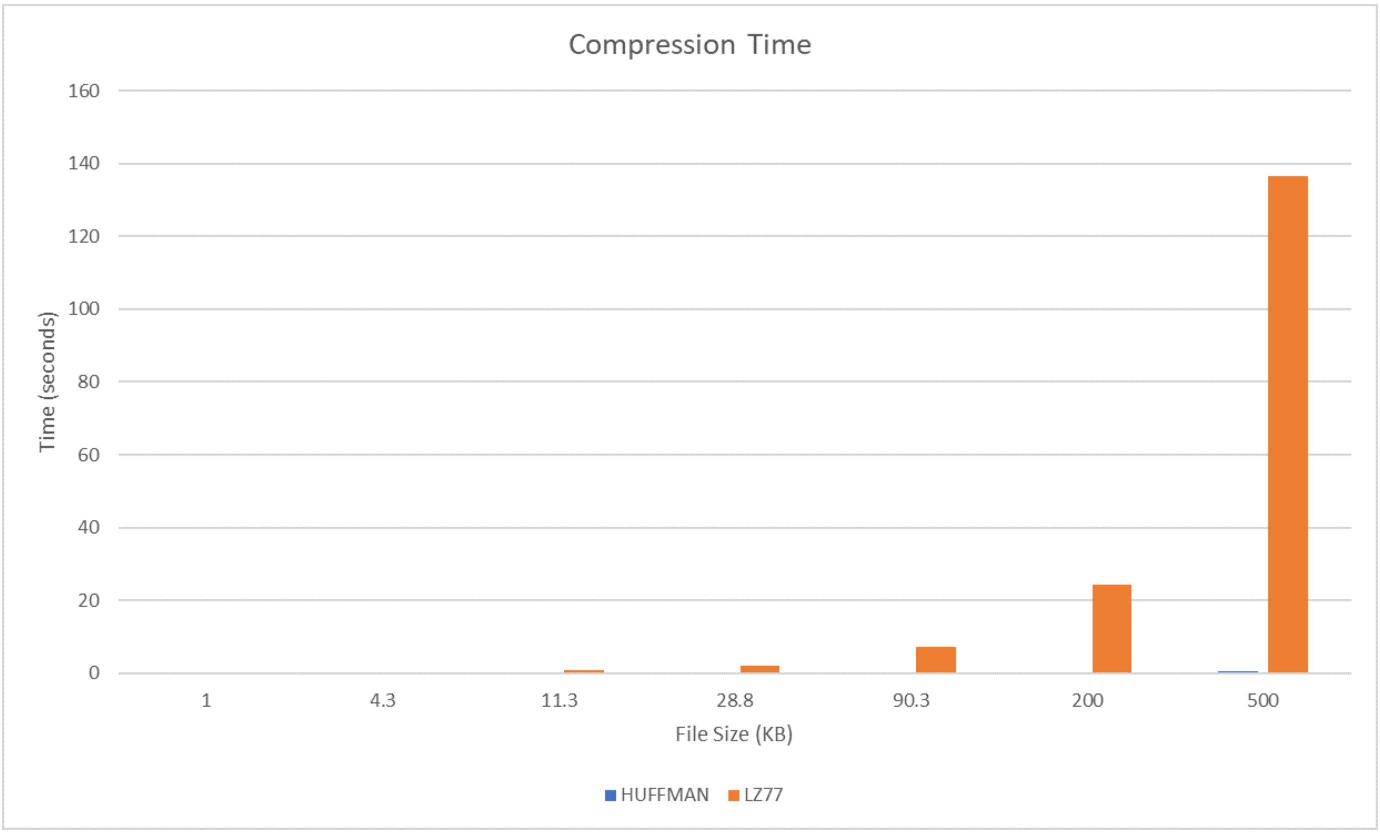


Comparison:

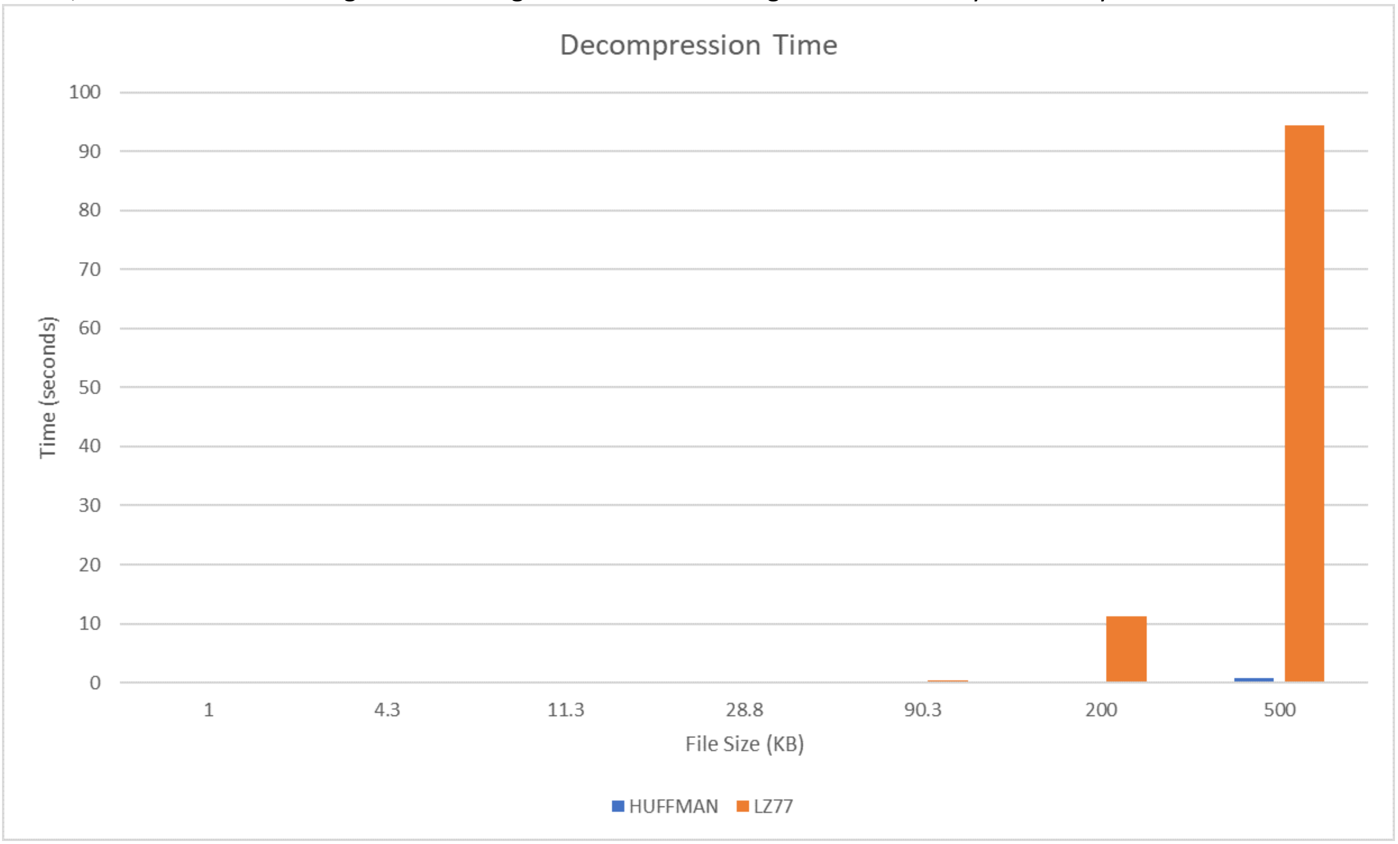
In order to better understand the limitations and strengths of the code, it was compared to a Python implementation of Huffman Coding, which can be viewed here:
https://github.com/bhrigu123/huffman-coding
This implementation only worked on TXT files, as it worked on characters rather than bits/bytes as the LZ77 implementation did, so this was one aspect where LZ77 was superior. Regardless of file size, the Huffman Coding always achieved a compression rate of 1.85, and so was much more efficient at compressing smaller file sizes, where the LZ77 was unable to compress them as well. As the file size increased, however, LZ77 quickly overtook the Huffman Coding, and was able to achieve compression rates between 2 and 2.5, for files between 28.8KB and 500KB.



The downside of this was that LZ77 began to slow down exponentially as file size increased, so while it did achieve much better compression rates, it took much longer than the Huffman Coding did, which maintained the same level of speed regardless of file size with only slight, linear, increases. Due to the extremely high speed of the Huffman Coding, more powerful machines had little impact on the run time, although LZ77 did have a slightly reduced run time, making it slightly more efficient than it was on weaker machines.



Similarly, decompression took exponentially longer on larger files for LZ77, while still being extremely fast for Huffman Coding. For larger files the decompression took longer for Huffman Coding than it did to compress the files, while for LZ77 the opposite was true. Once again, more processing power meant that the decoding was able to run faster, but there was not a significant change for Huffman Coding as it was already extremely fast.



From what has been seen it can reasonably be said that this implementation of Huffman Coding is far better to use on TXT files than the implementation of LZ77. One way to improve on LZ77 would be to run in on Bytes rather than Bits, which is what this implementation does, and this would likely improve the run time significantly, as there would be much less data to process.
In order to compare other file types, GZip within Python was used on the same files that LZ77 was tested on. GZip had a similar speed to Huffman Coding when it was run using the default settings, encoding and decoding almost instantly. It also achieved extremely good compression rates, such as 10.4 on the 512KB BMP file where LZ77 achieved only 3.94. It was also able to compress all the other file types, such as DOC, DOCX, JPG, and PDF, which LZ77 had failed to compress and had instead made larger, showing that there is still much room for improvement.
One final test of GZip showed that it was able to compress a 5.4MB TXT down to 1.9MB, in an average of 2 seconds, which had taken LZ77 roughly 4 hours to run on, and even then, it achieved no significant compression.