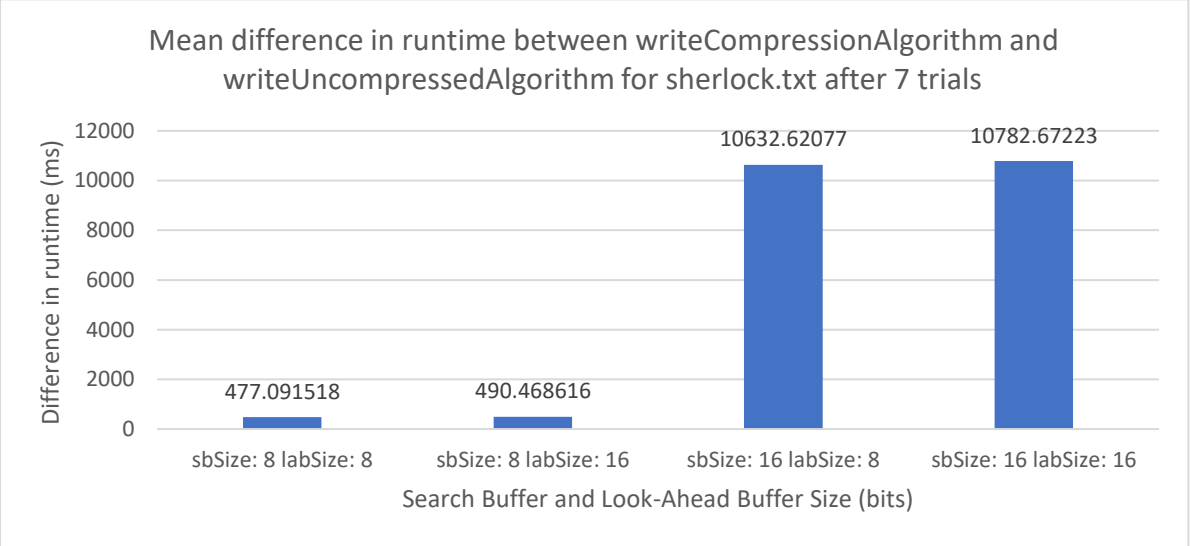
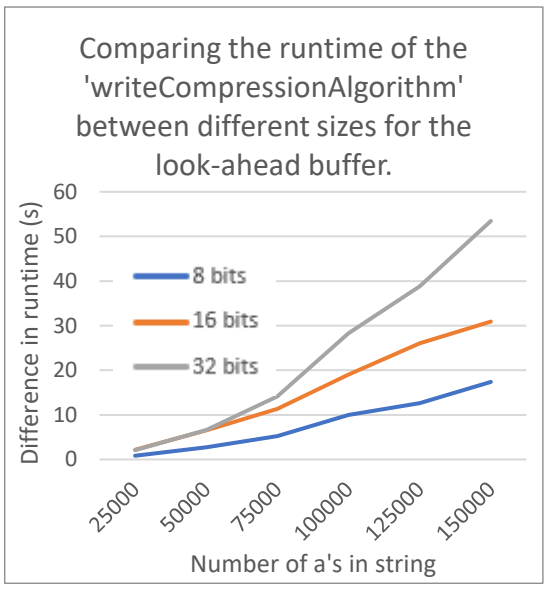
1. Running time of the encoder (compression)

Lempel-Ziv uses a sliding window (known as the search buffer) as a dictionary for the characters found ahead of the coding position in the look-ahead buffer. An encoder is used to convert the substrings created in the look-ahead buffer into triplets, storing the distance between the coding position and repeated string found in the search buffer, the length of the substring found in the look-ahead buffer and the next character in the sequence.

The Java program I have implemented can be used to encode strings of data via the LZ77 algorithm and writes the results to binary files. Variables such as the search and look-ahead buffer sizes can be edited in the main method which impacts the running time of the encoder as demonstrated:



Here, the 'sherlock.txt' example makes it evidently clear that changes in the search buffer size makes a big impact in runtime performance. This is because every character added to the look-ahead buffer triggers a search for a new substring in the search buffer. The larger the buffer, the longer it takes for each search to iterate through the sequence of characters. Comparatively, changes in the look-ahead buffer size has little impact on the runtime of the encoder for LZ77. Dependant on the file, the likelihood of a string exceeding the look-ahead buffer size varies. For a story such as the 'sherlock.txt', it is rare for a repeated substring to exceed 256 characters (2^8) and highly unlikely for a repeated substring to be of a length greater than 65536 characters (2^16). Therefore, the difference in runtime between different look-ahead buffer sizes is minimal.



On the other hand, the size of the look-ahead buffer can have a huge influence on runtime performances for repetitive strings. For instance, changing the buffer size for a string full of repeated of a's significantly varies the performance of the algorithm as shown in the graph. For an 8-bit look-ahead buffer, each 'a' added to the sequence of characters approximately increases the encoder's runtime by 0.13ms (for strings of length greater than 25000). Whenever the look-ahead buffer reaches full

capacity, a new triplet is stored, and the look-ahead buffer is cleared. Temporarily, this causes an increase in encoding speed before the buffer begins to fill up once more. In this example, the effects the number of characters found in the buffer has on runtime is most evident for 16-bits. Between a string of 75000 a's and 125000 a's, the number of seconds it takes to run the 'writeCompressionAlgorithm' significantly increases. This is because the 16-bit look-ahead buffer flushes every 65536 characters (2^16), meaning the number of characters in the buffer builds up between 65536 and 131072 characters. Likewise, for 150000 characters, the buffer would have been cleared at 131072 characters, diminishing the rate of increase in runtime duration for the remaining characters. Hence, a string of a's length 150000 has a faster encoding speed per character than strings of length 125000. Lastly for 32 bits, as the look-ahead buffer clears every 2^32 characters, the runtime of encoding strings of a's appears to increase arithmetically by 0.05ms such that $a_{n+1} = a_n + 0.05$ since the buffer never gets cleared between 0 and 150000.

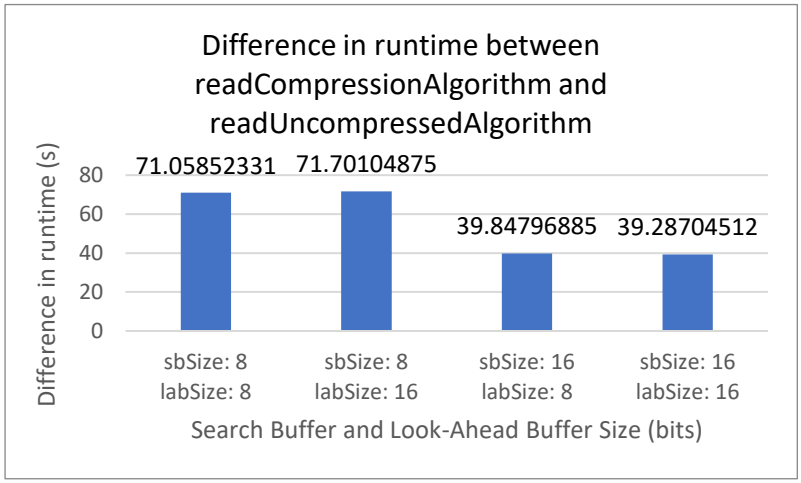
It is near impossible to calculate an average runtime of the encoder as the size of each repeated string found in the search buffer is wholly determined by the data input and hence varies drastically.

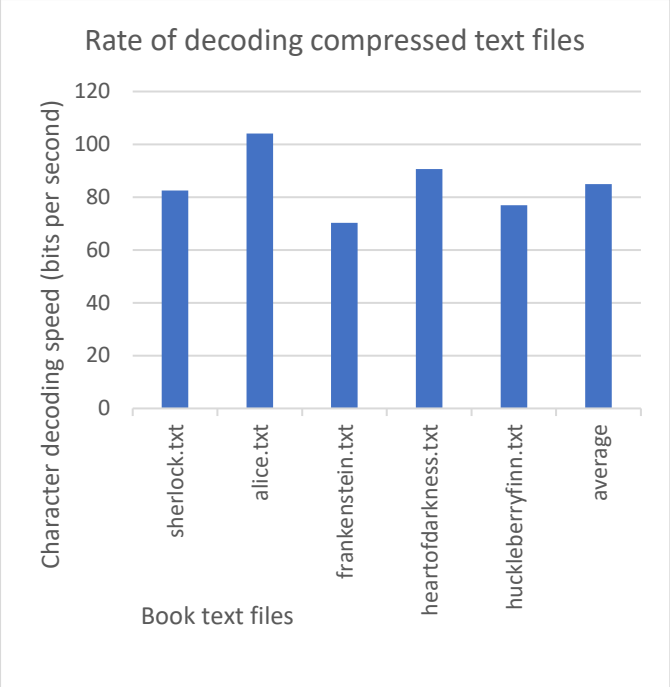
Running the algorithms on other computing devices made massive changes in the encoder's runtime, but similar patterns for the two examples above could still be derived.

2. Running time of the decoder (decompression)

When decoding the triplets of data, the implementation reads each triplet from a binary file and checks whether the distance between the coding position and repeated string found in the search buffer is greater than the length of the substring found in the look-ahead buffer. If the distance is greater than the length, the algorithm purely calculates start and end pointers for the substring to copy from the decoded string. If the length is greater than the distance, the algorithm initially finds the string to copy from the decoded string and concatenates it. Next, the same substring is added '|length/distance|' times followed by potentially part of the substring. Lastly, the next character stored in the triplet is added to string before moving on to the following triplet.

Contrary to the 'writeCompressionAlgorithm', a search buffer of 16-bits decodes and prints out the contents of the 'sherlock.txt' LZ77 encoded file in nearly half the duration of 8-bits. This is because the decoder doesn't need to iterate through the search buffer but instead uses start and end pointers derived from the distances and lengths stored in the triplets. Consequently, the primary factor that effects runtime is the merely the number of triplets that need to be decoded. As a 16-bit search buffer stores far more characters than an 8-bit search buffer (2^16 - 2^8) = 65280 to be precise), the likelihood of finding repeated strings greatly increases, and hence fewer triplets need to be decoded for the 16-bit buffer. For reasons identical to the encoder, the size of the look-ahead buffer has little impact on the decoder's runtime. Since repeated strings in stories such as 'sherlock.txt' are unlikely to surpass 256 characters, the only difference a change in the look-ahead buffer size makes to the runtime is reading twice as many bits for each length value in the triplets.



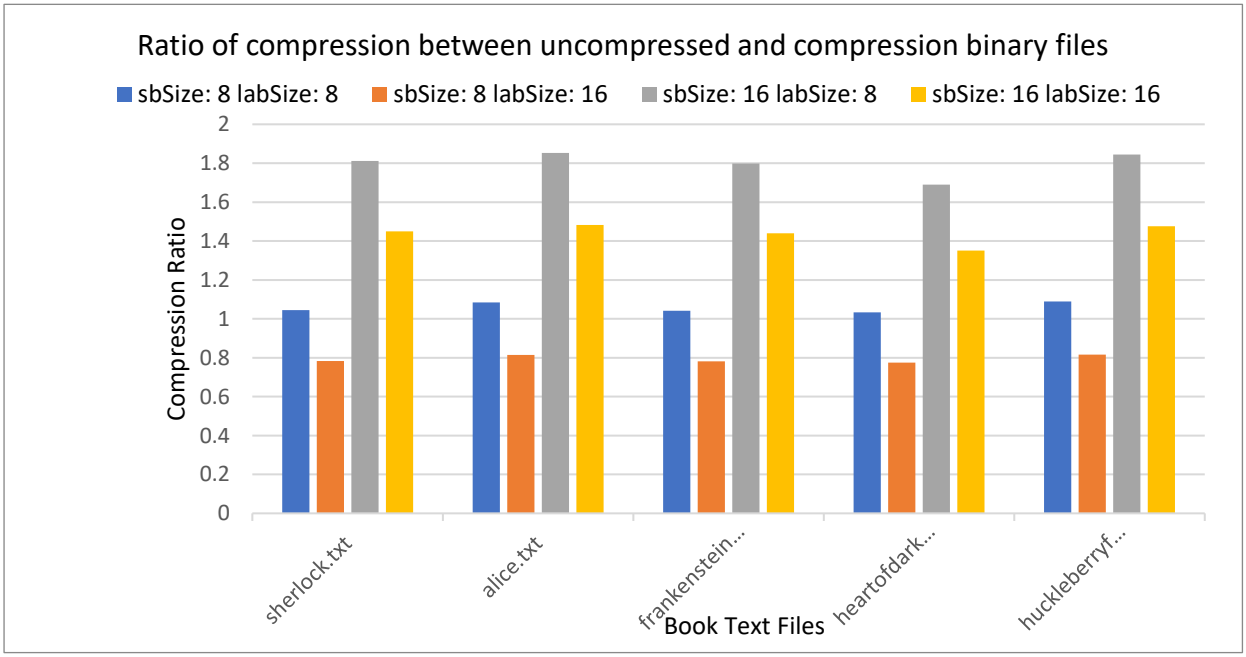


As the decoder's primary influence on runtime is the number of triplets that need to be decoded, an approximation for the average running time for each triplet to be decoded can be made. Across five different compressed text files on books (where the search buffer is 16 bits and the look-ahead buffer is 8 bits), the variance of the runtime is 136 bits per seconds. Moreover, with a coefficient of variation (standard deviation/mean) of 0.138, an approximation of a text file's runtime being 85 bits per seconds is relatively accurate.

However, as with the encoder, running the decoder algorithm on other computing devices made evident changes in runtime.

3. Compression ratio

As the size of each repeated string found in the search buffer is wholly determined by the data input, the compression ratio varies from file to file. For each of the book text files, where the



uncompressed binary files average at 3281326.4 bits, a search buffer of size 16 and look-ahead buffer of size 8 has the biggest impact on reducing the bit length the most. As expected, look-ahead buffers of size 8 are far more effective at compressing the file when compared with the 16-bit buffers, since repeated strings in the look-ahead buffer are unlikely to reach 256 characters long. Similarly, as previously mentioned, the likelihood of finding repeated strings greatly increases for 16-bit (in comparison to 8-bit) search buffers. Accordingly, fewer triplets are required for 16-bit search buffers.

Running the LZ77 encoder straight on images such as .png and .jpeg file extensions greatly increases file sizes and hence decompresses the images. This is because these image file extensions have already undergone compression processes. Using the 'eiffel.jpg' example, converting the image to a base64 string enlarges the file to 3669248 characters. Running the LZ77 encoder further enlarges the file for all combinations of 8 and 16-bit search and look-ahead buffers.

For a maximum compression ratio scenario to occur, the string stored in the file must consist of only a singular character. Using the string of a's example where the search buffer is set to 32 bits, the following number of bits are stored in the compressed file for a string of length 'x':

Number of characters	**8-bit look-ahead buffer**	**16-bit look-ahead buffer**	**32-bit look-ahead buffer**
1 (8 bits)	48	56	72
2 (16 bits)	96	112	144
12 (96 bits)	96	112	144
256 (2048 bits)	96	112	144
257 (2056 bits)	96	112	144
258 (2064 bits)	144	112	144
25000	4752	112	144
50000	9408	112	144
75000	14064	168	144
100000	18768	168	144
125000	23424	168	144
150000	28080	224	144

For each respective buffer size, the maximum compression ratio has been achieved. Trivially, for strings 'a' and 'aa', a triplet is used to store each character, no matter the size of the window. For any string of length smaller than 12, the file cannot be compressed with LZ77 using a combined window size of (32+8 =) 40 bits, no matter what the contents are. As an 8-bit look-ahead buffer cannot store strings greater than 256 characters, a new triplet needs to be created every 256 characters for the string of a's example. Since the second triplet starts on the second char, a third triplet is created on the 258th character at which point a 16-bit look-ahead buffer is more effective at compressing the string. More generally, a new triplet is created at $2^x + 2$ characters (where 'x' is the look-ahead buffer size). Using 16 bits as an example, a new triplet is required at 2, 65538 and 131074 characters.

4. Comparison with other compression techniques

In 1977, the revolutionary LZ77 algorithm was published by Abraham Lempel and Jacob Ziv. In the year following, the pair released a new compression algorithm known as LZ78 that generates and uses a static dictionary. Instead of using lengths and distances found in LZ77 triplets, the LZ78 algorithm only requires a dictionary and the character that next follows. Unlike LZ77, the LZ78 algorithm works on future data, forward scanning the input buffer until the substring no longer matches a string in the dictionary. Then the substring is added to the dictionary along with the next, following character. As new patterns are held in the dictionary indefinitely, there is no bound to the maximum size of the dictionary. This lengthens the duration of compression.

In 1982, a modification of LZ77 known as LZSS was made by James Storer and Thomas Szymanski. LZSS improves on LZ77 as it can detect whether adding a distance/length pair over a specific character will result in a lower bit count. Ahead of every encoding point, a single bit flag is added to the encoded string, indicating whether the following string of bits represents a character or

distance/length pair. Stripping the ‘next character’ property from the LZ77 triplets further helps compress text files.

Another variation of the LZ77 algorithm, DEFLATE, was invented in 1993 by Phil Katz. Combing the properties of LZ77 and Huffman coding, moderately compressed files can be produced quickly. Huffman coding works by assigning a code to input characters, where the more frequent are given small codes of short bit lengths. The DEFLATE algorithm initially compresses the data using LZSS with a sliding window of 32kB and length matches between 3 and 258. Afterwards, the data is further compressed using Huffman-encoding (known as LZH) following two additional rules: elements that have shorter codes are placed left of those with longer codes and that for all codes of a given bit length, those that come first are placed to the left. The trees produced are encoded by their respective code lengths. Once the sequence of code lengths is fully assembled, it is further compressed using run-length compression.

Using java.util.zip.Deflater, the algorithm reduces the ‘sherlock.txt’ file from 4759296 bits to 1815072 bits. Comparatively, the most effective parameters for the LZ77 algorithm resulted in compressed file of 2626880 bits, which shows the DEFLATE algorithm is clearly an upgrade to the LZ77 algorithm.

In 1998, the Lempel-Ziv Markov Chain Algorithm was published. Using a .7z file format, it achieves better compression than the DEFLATE algorithm in most cases. LZMA begins compression using a modified LZ77 algorithm, before undergoing arithmetic coding. Depending on the LZMA algorithm, multiple techniques can be applied to further compression. As the LZ77 algorithm operates at a bitwise level (not bytewise), the final bit count is usually considerably lower than other compression algorithms.

References:

LZ77: https://en.wikipedia.org/wiki/LZ77_and_LZ78

LZ78: http://www.ijesit.com/Volume%204/Issue%203/IJESIT201503_06.pdf

LZSS: https://ethw.org/History_of_Lossless_Data_Compression_Algorithms

DEFLATE algorithm: https://www.tutorialspoint.com/javazip/javazip_deflater_deflate.htm/