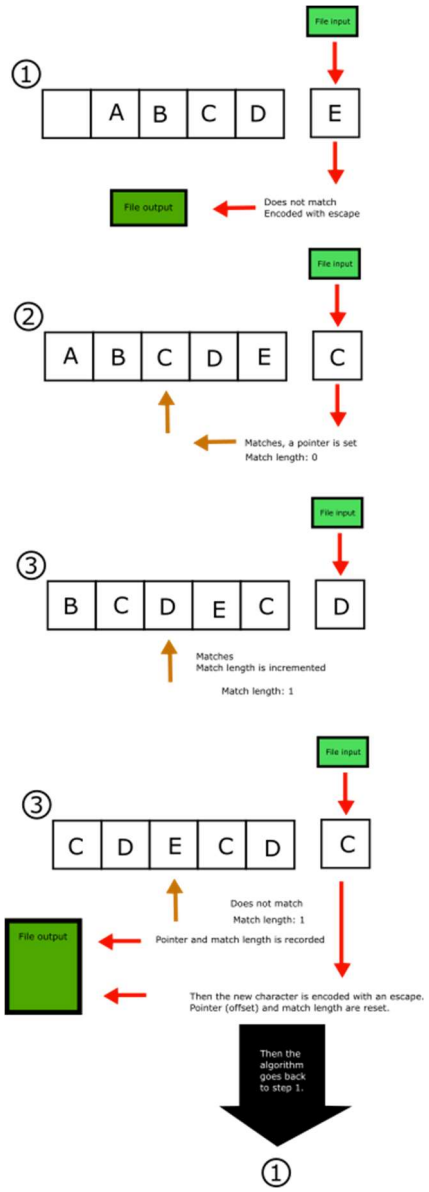
Alternative compression algorithms

LZ77 [1] [2]

A compression algorithm proposed by Abraham Lempel and Jacob Ziv in 1977. This algorithms is used in a number of dominant compression algorithms still in use today, such as png, gif or zip.



The encoder holds a certain length of the encoded file in memory in a so called ‘sliding window’ to which newly read expressions are periodically appended, while the earliest read expressions are removed from the window once it reaches a certain length. (So that only the ‘last’ X expressions are held in memory.)

During encoding the encoder first initialises match length counter, a simple integer, and an offset, sometime called distance. When a new expression is supplied to the encoder, it searches back through the window until it can find a matching one. If that is not possible, the expression is encoded with an escape character; else, encoder finds a match and it sets the pointer to the matching expression as offset. Either way the new expression is appended to the window. If the window has reached its terminal length, the earliest read expression is discarded.

If the next passed expression matches the expression the offset is now pointing to, (it will be pointing to an expression following the expression that originally matched because the sliding window has moved,) then the match length counter is incremented by one; else, the pointer is written into the file along with the current match length counter. Then the new expression is recorded onto the sliding window. If the expression originally matched, the this paragraph is repeated. If it has not, the process is repeated from the start of the previous paragraph.

During decoding the sliding window is once again initialised. If the decoder reads an escape character, it simply copies the following character into the sliding window and into the output file. If it however comes across offset and match length pair, it will read the expression at the offset in the sliding window, append it to the window and to the output file and decrease the match length by one. If the match length remains non-negative, it will repeat the process. Since the process is performed expression by expression, the match length can be greater than the offset, it can even be larger than is the size of the sliding window.
The encoder will keep repeating this until either the match length variable becomes negative, at which point it will read the next expression from the encoded file. If it reaches the end of the file, it closes the output and exits. The file was fully encoded.

Performance of this algorithm varies depending on the window size. If the window size is larger the compression will take longer and require more memory, but the resulting file may be smaller.

Another way to enhance this compression algorithm is to have it attempt to match the whole expression to an earlier occurrence in the sliding window in case the currently read expression does not match the expression following in the sequence.
Assume the last six expressions of a sliding window are ABCABD.
If A is read, it will be matched to -3. If A is followed by B, the sequence continues. If the B is followed by C, the originally described algorithm would terminate the sequence and attempt to match C as a start of a new sequence.
However, with increased code complexity and additional computing resources, it might be possible to match the ABC sequence to a sequence starting at -6.

Run-length encoding [3]

Run-length encoding is one of the simplest encoding algorithms. It is effective on data of low variability between expressions, which does not alternate often. A classical example is a serialised black and white picture.

Run-length encoding reads and expression and creates a counter of times it has encountered the expression. If the following expression is identical, it increments the counter. If the following expression is different, it records the original expression along with the number of repetitions before resetting the counter and repeating the whole process. The encoder simply terminates once it reaches the end of the file.

Decoder reads an expression and then reads the number of repetitions. Then it writes the expression into the output file as many times as the repetition suggested. Once it reaches end of the file it terminates.

However, this algorithm has found a lot of use with lossy compression of pictures, since it is possible to reduce a gradient of multiple pixels into a single value without human eye recognising the difference.

Data structures and algorithms used

String

“The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.” [4]

Strings are used throughout the program to represent sequences of bits with 1s and 0s.

TreeNode

The TreeNode class represents specific nodes in a binary tree used for implementing Huffman compression algorithm. Each, except the root node, node is contained within a different node. The root node is stored throughout the program.

Each node has its address which is stored as a String of 1s and 0s and is assigned shortly after the tree is created. The root node’s address is an empty string. Each subsequent node takes address of its parent node and appends either 0 or 1 to its end depending whether it on its relative position to the parent’s node.

The tree’s leaves also contain an Integer which’s last 8 bits represent the byte the leaf represents.

Decoder

The Decoder class represents an object which decodes a specific input stream according to a binary tree consisting of TreeNodes. Each instance has a String buffer which contains a sequence of 1s and

0s representing read bits. The bits are processed and the binary Huffman tree is traversed according to them.

Encoder

The Encoder class represents an object which encodes a specific input stream according to a binary tree consisting of TreeNodes. Each instance has a String buffer which contains a sequence of 1s and 0s representing bits to be written. The bits are returned eight at a time. Each instance also contains a HashMap which maps specific bytes onto a sequence of bits.

TreeBuilder

The TreeBuilder class represents and object which constructs a Huffman binary tree by reading and processing a specific file.

TreeBuilder goes through the whole file and for each unique byte in the file it creates a TreeNode leaf stored in an HashMap under its byte. Each leaf also keeps track of how many times its byte has occurred in the file.

Once the whole file has been traversed, all leaves are stored in an array which is sorted with Java's internal Merge Sort. Then the last two nodes are selected, added as child nodes to a new node which replaces one of them in the array while the other leaf is replaced with an empty node. An external variable keeping track of non-empty nodes in the array is decremented and the whole process keeps repeating as long as there are more than one non-empty node in the array.

When the root node is found it is returned.

LinkedList

LinkedList class consists of series of linked objects, pages, where each contains a pointer to following and superseding page and an actual content.

LinkedList can either be traversed through by an iterator or the first page may be retrieved or another added to the end.

LinkedLists are used where stored objects are not accessed out of order in which they were inserted, such as queues or orders.

HashMap

"Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time." [5]

HashMaps provide constant access performance to all entries as it maps key objects onto values. In the program HashMap is most notably used to read Bytes onto a sequence of 1s and 0s representing bits inside the Encoder class.

Weekly log

1st week (11th – 16th January)

The specification was read, and it was decided that ideally a whole week be required to produce this report and around a week will be required to produce the actual code.

2nd week (1st – 7th March)

On Wednesday, the implementation was designed.

On Thursday and Friday, the Encode, Encoder, Decode and Decoder classes were implemented.

On Saturday and Sunday, the TreeNode and TreeBuilder classes were implemented.

3rd week (8th – 12th March)

On Monday and Tuesday, the generateContent method of the TreeNode class as well as TreeRebuilder class were implemented.

On Wednesday, the documentation for the code (JavaDocs) was created. Weekly log and Potential Areas for improvement subsection of the Performance Analysis section of this report were written.

On Thursday, the rest of the analysis was performed and the section of this report was written. Moreover, Data structures and algorithms used section was written.

On Friday, the Alternative Compression Algorithms section was written.

Performance analysis

Analysis

First performance analysis was performed on Grimms' Fairy Tales. In English, Portuguese and French. First all three books .txt format were compressed using Huffman compression and a tree generated from them. In the second stage the three books were compressed using tree generated for English translation generated in the first stage. For sizes and compression times please see figure 1.

Language	**Original size**	**Compressed size**		**Compression ratio**	**Compression time**
Using native .tree file					
		Content	*.tree*		
En	548 KB	314 KB	372 bytes	57.3%	137 ms
Fr	80 KB	48 KB	416 bytes	60%	56 ms
Pt	98 KB	58 KB	424 bytes	59.18%	59 ms
Using alien .tree file					
En	548 KB	314 KB		57.3%	127 ms
Fr	80 KB	47 KB		58.75%	48 ms
Pt	98 KB	59 KB		60.2%	48 ms

Fig. 1

In the second stage the same was performed with Othello.

Language	**Original size**	**Compressed size**		**Compression ratio**	**Compression time**
Using native .tree file					
		Content	*.tree*		
En	148 KB	107 KB	368 bytes	58.15%	137 ms
De	109 KB	65 KB	404 bytes	59.63%	56 ms
Pt	151 KB	87 KB	424 bytes	57.62%	59 ms
Using alien .tree file					
En	184 KB	107 KB		58.15%	127 ms
De	109 KB	68.7 KB		63.03%	48 ms
Pt	151 KB	93.2 KB		61.72%	48 ms

Fig. 2

As can be seen in both figures, using alien .tree file, a Huffman tree generated for a different file, has led to a steady decrease of compression time as reconstruction of the tree from a parsed file is faster than reading the file and generating its tree. This difference would be even more significant on larger trees. Moreover, it can be seen the algorithm operates at around 60% compression ratio which is consistent with later findings on real bulk data.

An interesting find is that the compression rate has improved for French translation of the book when an alien tree was used. This can be explained away in multiple ways, any byte undefined in the tree is replaced by a series of zeroes, since the more times occurring node is assigned to zero, a string of zeroes will lead to the shortest series of bits that represent a byte. Moreover, since there is a lesser number of bytes defined in the English tree, each of them can be represented with shorter series of bits. However, this behaviour is still undesired since the decompressed file will be corrupted.

Third performance analysis was performed on bulk files of three types, real, pseudo-real and artificial. Kernel [6] was representing real data, DNA [7] representing pseudo-real and fib41 [8] representing artificial data. All of the data sets were compressed using their native tree.

File	**Original size**	**Compressed size**		**Compression ratio**	**Compression time**		**Decompression time**	
		File	.tree		ms	converted	ms	Converted
artificial/fib41	261,636 KB	45197 KB	12 bytes	17.27%	16452	16 s	28677	28 s
pseudo/dna	102400 KB	28593 KB	24 bytes	27.92%	7873	8 s	20950	20 s
real/kernel	251916 KB	170227 KB	644 bytes	67.57%	36120	36 s	111220	1m 50 s

Fig. 2

As can be seen, decompressing the file takes much longer, possibly due to many method calls and context shifts or many String operations being performed with each processed byte.

Files which contain more repetitive data, as evidenced by smaller .tree files, can be compressed much more efficiently. It can also be seen that despite growing size of the original file, the .tree file grows negligibly. And there is theoretical maximum size since there will only be 256 terminal nodes and each node will only ever take exactly two bytes to record.

Potential areas for improvement

Encoder class

Populating the HashMap with a <byte,TreeNode> pair is superfluous. Populating it with <byte,address> would have been more efficient as it would save at least one call.

Saving generated Huffman tree

Two distinct changes could be made to the way the Huffman tree is saved after being generated.

It could be further reduced in size if the controlling characters of the tree (‘{‘, ‘}’ and ‘-‘ encoded according to ASCII_US) were instead two bits long. This change would further reduce the size of the tree file and it would prevent the tree from being human readable.

Second possible change is to inline the tree into the encoded file. This could be easily achieved with a simple change of the coding and defining a new controlling character representing end of the tree. It was decided against doing this since the assignment called for reusing generated trees for other files.

Strings of 1s and 0s

In a number of sections bits are represented as Strings of 1s and 0s. This choice was made because it was simple and intuitive to implement. However, it would be more efficient to instead operate with Linked Lists of Booleans instead. As removing the first item and appending adding item to the end is less computationally expensive than editing strings.

Undefined bytes

Using a tree generated for another file opens the program to the possibility of encountering an undefined byte. A without defined sequence of bits to represent it.

The current solution to the problem is to encode the undefined byte as a byte defined by sequence of zeroes. This risks corrupting the final file. The approach was chosen since it still produces an encoded file which, when decoded, has a correct byte count.

There are alternative approaches. It would be possible to just ignore a byte that is undefined in the tree, however that resulting file would have lost bytes which breaks the paradigm of loss-less compression. It would also be possible to change the original tree so that this new character is amended to it, but that would corrupt any files that were encoded previously using the same tree.

References

[1] J. Z. a. A. Lempel, "A universal algorithm for sequential data compression," *IEEE TRANSACTIONS ON INFORMATION THEORY,* vol. 23, no. 3, pp. 337--343, 1977.

[2] "DataCompression," [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/data-compression/lossless/lz77/index.htm. [Accessed 11 03 2021].

[3] A. H. R. a. C. Cherry, "Results of a prototype television bandwidth compression scheme," *Proceedings of the IEEE,* vol. 55, no. 3, pp. 356-364, 1967.

[4] Oracle, "Java® Platform, Standard Edition & Java Development Kit," 2. [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html. [Accessed 11 3 2021].

[5] Oracle, "Java® Platform, Standard Edition & Java Development Kit," [Online]. Available: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/HashMap.html. [Accessed 11 3 2021].

[6] "Pizza&Chili Corpus," 11 3 2021. [Online]. Available: http://pizzachili.dcc.uchile.cl/repcorpus/real/kernel.7z.

[7] "Pizza&Chili Corpus," 11 3 2021. [Online]. Available: http://pizzachili.dcc.uchile.cl/repcorpus/pseudo-real/dna.001.1.7z.

[8] "Pizza&Chili Corpus," 11 3 2021. [Online]. Available: http://pizzachili.dcc.uchile.cl/repcorpus/artificial/fib41.7z.