Lossless Compression Project Report

Compression algorithm is commonly used in today's digital age, anything involves handling a piece of data on a device would be processed by some algorithms behind the scenes. From texting people on mobile phone to watching a movie on a media streaming platform, data needs to be transmitted with a reason able size in order to maintain the balance between processing speed and its quality, so the user experience can be optimised. I.e., receive and load text messages quickly, and reduce buffering interruption of an online video; From saving an image on a smart phone to managing information's in a website's database, data is preferred to be compressed in order to free up extra storage space and reduce loading to time, which then further lower the maintaining cost. E.g., buying a new SD card for the smart phone, and adding new storage facility in a data centre. [1]

There are two categories for data compression algorithm, either lossy or lossless, which are used in many different types of files, for storing files like image, audio, and video, with varied space-time performance in compression and decompression.

Lossy compression reduces the size of a file by removing parts that are less import to the result. For example, there is JPEG for image lossy compression, MPEG for video lossy compression and MP3 for audio lossy compression. These format all can lower the quality of the file, but it saves storage space and required less time on transmitting. Hence, these methods of lossy compression are commonly used in many media platform because it is good enough for the user experience. However, removing data means the damage is irreversible so there is no way to convert it back to its origin size. [2]

JPEG (Joint Photographic Experts Group) is a common format to compress an image (offer both lossy and lossless). JPEG uses lossy compression (DCT) [3] to reduce the size of a large image file into 1/10 compares to its origin size, meanwhile the result is still within acceptable level of quality reduction. [2] This format is widely adopted by photo capturing device such as smart phone and digital camera, and by the web to store and send images across the internet.

MPEG (Moving Photographic Experts Group) is a format to compress video (also for audio) that uses the same lossy compression basis (DCT) to reduce down to 1/26 (for VHS-quality video) and 1/6 (for CD audio) [4] compare to their original size, without significant loss in quality. This file format (MPEG-1) was used to encode both video and audio in a portable storage device such as CD and DVD, and to broadcast contents in television and satellite. Now, (MPEG-DASH) it is also a standard used in streaming high-quality media content off the internet.

MP3 (MPEG-1 Audio Layer III) is a common lossy compression coding format largely used for storing audio file, which uses MDCT [5] to achieve data size reduction without any noticeable quality damage to the user. This format was retained from a subsequent part of the MPEG audio compression encoding. It is now a standard for audio file with multiple-audio channel and extra bitrates. The mainstream media platform uses this file format to widely distribute musical contents to countless users via portable devices that accepts this format, including MP3 player, smartphones, and laptop.

Lossless compression is another type of data compression that allows compressed data to be converted back to the original data without any data loss. In contrast to lossy compression, although the operation does not sacrifice any data, there is no lossless compression that can compress all data efficiently according to the pigeonhole principle. Unless there are some particulars input or assumptions made to certain types of uncompressed data before compression, or the compressed data does not get smaller. The reason to use lossless compression is when the quality of the data is more important than the size reduction of the data, so the decompressed data would be identical to the original data. For example, image, text-based document, source code of a program and executable.

JP2 (JPEG 2000) is a lossless data compression format (also offer lossy) on storing images. It does not use the same compression approach (DCT) as JPEG but with a wavelet-based approach (DWT) instead to preserve data quality, rather than dramatic reduction in size on a usual occasion. This

compression standard has application in transmitting medical imagery, storing meteorological data, and broadcasting high-definition television feed.

MJ2 (Motion JPEG 2000) is a lossless data compression format on storing video and audio. This format shares the exact compression technology used in JPEG 2000, which its image compression standard can be adopted to compression video with DWT. Again, the MJ2 does not offer larger size reduction so it would be costly to store and require more bandwidth in transmission. It was selected to be the standard for video coding in the film industry in 2004 [6].

Huffman encoding is the lossless data compression algorithm I used to implement my project. It is a compression technique developed by David Huffman in 1952 [7] to reduce the size of a given file by using a binary tree that is frequency sorted. This is still amongst the most efficient methods hence it is widely used for generating prefix-free code [8] and works well in uniformly distributed input.

In practice, the Huffman coding takes the frequency of every characters in a text file. Initially, each character is treated as a node with its frequency as the weight. To constructure a Huffman binary tree, two nodes with the smallest weight are connected to a parent node with the sum of their weight, which forms a binary. The similar nodes combining process is repeated until all nodes (and the ones with daughters) are combined into a single binary tree.

With the frequency ordered binary tree, it is then used to encode a text document. The encoding follows a simple principle: if a traverse is made to the left of the tree, the encoding 0 is added; otherwise, the encoding 1 is added. The tree traversal for each character ends when the leaf, which is a node that contains the character, is reached. And then, the traverse process begins from the root again until all characters are encoded. Afterwards, a unique Huffman tree and a Huffman encoded data is produced.

To decompress the encoded data, the Huffman binary code is used to traverse the Huffman tree it produced. Similar to the compression principle, traverse to the left if it is a 0, vice versa. When a character is found in a leaf, the traverse process begins from the root again until the Huffman binary code exhausted.

The second available lossless compression approach is run-length encoding (RLE). The idea of this algorithm is to reduce a series of consecutive data that repeats itself by representing it into the count of its repetition and the value of that single data. For example, a text document may have paragraphs with lines of sentences. Each sentence that contains consecutive repeated letters are replaced with a number of repetition and the character itself, otherwise, the section remains the same.

This method was largely used in image compression, especially efficient for image stored in CLUT-based bitmaps such as icons, until its popularity was replaced by a more complex format, GIF, later. Digital data format that uses this technique includes ILBM, PCX, TARGA and PackBits.

Another lossless compression approach is LZ77 (a Lempel-Ziv compression), developed by Abraham Lempel and Jacob Ziv in 1977 [9] to achieve data reduction by substituting data that has repetitive appearance with references to an identical copy of the exact data earlier in the original data stream.

For instance, in a text document, a search buffer is used during the compression process to hold processed character to hopefully match the next character(s) in a sentence. The process separates the character stream with two sections: a search buffer, which consist of symbols seem and processed earlier; and a look ahead buffer, which consist of symbols that have not been seem and processed yet by the process. The size of the search buffer is usually larger than the look ahead buffer.

The encoder looks at the first character in the look ahead buffer and tries to find a match in the search buffer, if a match is found then the encoder will look at the next character and combine it with the previous character to find the longest match in the search buffer. Otherwise, with a settled match, the

program returns a token which consist of references including the distance from this character(s) to the one in the search buffer; the length of matched character(s); and the next character in the sentence (the look ahead buffer). Afterwards, the separation window between two buffers shift to the next character. If the character/ symbol is found for the first time, a null token that contains two elements of zeros and the symbol itself is returned. The encoding process is achieved by creating these tokens for the entire data stream, then combing each of the channels in their associative lanes.

The LZ77 compression algorithm is then modified with a variated compression approach in the search buffer and look ahead buffer such as LZMA (used in 7 ZIP) [10], LZSS [11] (used in WIN RAR), DEFLATE (used in WIN ZIP) [12] are some recognisable variants that use similar principle in LZ77.

List of data structures and algorithm used in my lossless compression implementation:

Tuple is an immutable fix-sized data structure. It is used to form a node that stores the character or a node object, and the weight/ frequency of the that node in the node constructor. It is also used to store the daughter item(s) of a node which are returned by the getter method in the node class.

List is a mutable resizable data structure. It is used to store all the available nodes for the Huffman tree before and after the tree construction.

Stack is an abstract data type where the operation follows the first in last out order. It is used in the depth first search algorithm for traversing the binary tree by the recursive function.

Linked list is a collection of data where the next element is pointed by the pointer. It is used to implement the Huffman tree, where the list contains a tuple that holds an object points to its daughter.

Binary tree is a type of tree which each node can have up to two children at most. The tree represents the Huffman binary search tree when performing both compression and decompression in my program.

Hash table associate keys with their own specific values. It is used by the dictionary when counting the frequency of each characters, which stored in the compressed file. It is then converted into a list of tuples to be used in constructing a tree in both compression and decompression process. It is also used to store the huffle encoding for each character in the compression process.

Depth-first search is the algorithm used in search a tree. It is used in traversing the deepest nodes recursively (the leaf) in the Huffman tree to produce the encoding for the characters and decoding for the binary code.

The following is the weekly log of my progress.

Week 1: I watch explain video on the coursework specification. Understanding the abstract process for the implementation through the project decipher document. I read the Huffman code section in the Cormen's book [13] for some idea on the topic.

Week 2: I found an e-book in text file with uft-8 encoding [14]. I used python to test the reading file function I coded. It collected word count frequency, which then was stored in a dictionary. The dictionary pick up all the characters in the text file by using the ".update()" method to add new character to the dictionary. I noticed there is an encoding phenomenon where the character "\ufeff" is collected. I used the new encoding "utf-8-sig" to get rid of it after looking up on the python documentation [15].

Lossless Compression Project Report

After reading the project document, the first data structure I can think of is using dictionary to implement tree in python. But I then realized dictionary does not allow duplicated keys, I abandon this approach. Instead, I can use the dictionary ADT as a hash table for the Huffman encoding when the tree is generated and store the characters' corresponding Huffman encoding in it.

Week 6: Through the failure of my first attempt implementing the tree, I noticed every node in the tree act as a blueprint with different content inside. That implies the tree can be implemented with an OOP approach. I used the official python document to aid my understanding on OOP [16]. Inside a node object class, I coded a node constructor, which combine 2 sub-items (left or right) together into an object and a getter method function that returns a node's daughters.

To constructure a tree, I combine the node or character with the lowest frequency or weight, with the constructer function. I used a loop to repeat this process until there is only one node left after all the nodes are combined. Obtaining the encoding for each character from the tree requires a traversal algorithm, I tried using depth-first search algorithm in a recursive manner to traverse all the leaf in the tree. Each recursion has access to the getter method function to read a node's object's daughters. If a node object has a data type of a string, then it must be a leaf. All traversal results were stored in a dictionary with each characters' corresponding Huffman binary encoding, which would be used by the compression process. I used the official python document [17] to help me understand how to measure processing time in my code.

Week 9: To write the file in binary, I used the method mentioned in Chico's email by using a bit writing library [18] since it is easier to implement, with the aid of the websites he included. Also, he mentioned to convert the Huffman tree in ascii binary code along with the compressed file, so I tried to convert the dictionary into a long string with 'str()' and convert the string back to dictionary with 'dict()', but a 'ValueError' occurred as a result. I researched the issue online and found a solution [19] to the conversion issue by using the json module, which allows me to convert a dictionary into a string, and then back to a dictionary object when needed. After converting the dictionary into string, I looked up a python tutorial website [20] that shows how to convert a string into binary.

I noticed I need to find a way to let the program to recognize the binary section for both the decoding dictionary and the Huffman code, so I added a separation of an ascii binary code between the sections as a constant 'catchphrase' that can be identified every time for any compressed file.

After I managed to compress in a binary file, I started on the decompression code. I wrote a for loop to identify the catchphrase and extracted the dictionary binary string from the compressed file. To convert the dictionary from binary to string. I experienced some errors when trying to convert the dictionary string back to a dictionary. After some investigation, the length of the dictionary string is the same as the length of the dictionary binary code, which empty spaces are added before the actual dictionary string during the binary string conversion. I used a while loop to remove all the unwanted empty space.

With the recovered dictionary and a Huffman encoded content, I re-constructed a Huffman tree with the method I used in the compression section. Then, I wrote a loop to go through the entire binary string to traverse the tree with every 1s and 0s. Once a traverse reached a leaf, the character in the leaf was saved to a new decompressed string, and another traversal begin from the tree's root again until the end of the binary string.

To test the program, I used a sample utf-8 encoding book I found [14]and the program return the identical decompressed file compared to the original file. Afterwards, I created a simple menu that let user to either compress a text file or decompress a compressed file depending on the file name entered on the interface.

Lossless Compression Project Report

Performance analysis part A: (Compressed file contains both encoding and the Huffman tree)

Two books' English encoding's effectiveness against Italian and Finnish version.

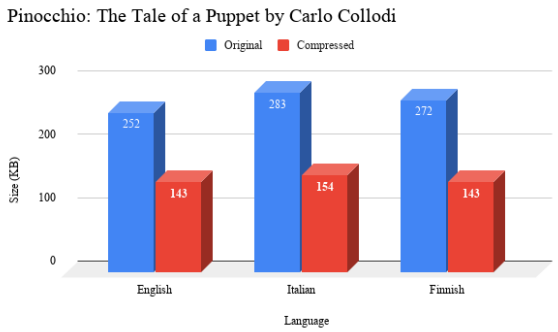

Figure 1. Book [21] [22] [23].

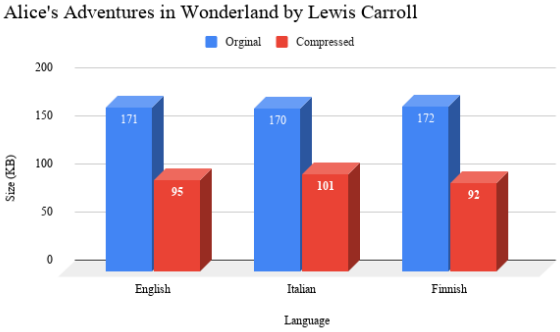

Figure 2. Book [24] [25] [26].

Conclusion: The results in figure 1 shows the English textbook has achieved compression reduction down to around 57%; Italian, around 54%; Finnish, around 53%. The results in figure 2 shows the English textbook has achieved compression reduction down to around 56%; Italian, around 59%; Finnish, around 53%.

The compression effectiveness of the English Huffman encoding on the Italian and Finnish languages textbook produces a similar reduction results, range between 53-59% to the original language text. This could be the fact that the English Huffman encoding tree does not account some of the special characters in Italian and Finnish, which caused, especially in figure 1, to produce a relative lower size reduction compares to English, similarly for results in figure 2. Also, the reduced size for Italian is higher than English and Finnish, which could suggest English encoding does not work as effective compares to Finnish in this scenario.

Performance analysis part B: (Compressed file contains both encoding and the Huffman tree)

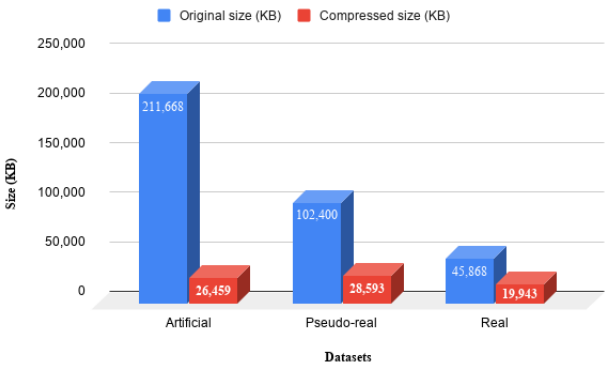

Figure 3. Compression efficiency across different datasets of repetitive text in varied patterns [27].

Conclusion: The compression result for "artificial" (Run-Rich String Sequence) demonstrated a size reduction down to 12.5% from its original data. This suggests my Huffman encoding performs efficiently on man-made repetitive strings. This could be the fact that the file contains repeated content that are uniform probability distribution, which provides an optimal environment for Huffman encoding.

The compression result for "pseudo-real" (DNA) demonstrated a size reduction down to 28.0% from its original data. This suggests my Huffman encoding performs relatively slightly less efficient then "artificial" on partially men made repetitive strings. This could be the fact that the file contains

content that are not as uniform probability distribution, which provides a environment just below optimal for Huffman encoding.

The compression result for “real” (world leaders) demonstrated a size reduction down to 43.5% from its original data. This suggests my Huffman encoding performs relatively less efficient then “pseudo-real” on real repetitive strings that naturally occurs in text. This could be the fact that the file contains content that are not as identically distributed, which provides a pessimal environment for Huffman encoding.

References

[1] futurelearn, "Everyday compression," futurelearn, 19 4 2020. [Online]. Available: https://www.futurelearn.com/info/courses/representing-data-with-images-and-sound/0/steps/53155. [Accessed 16 03 2021].

[2] A. Mazen and A.-A. Jamil, "JPEG Based Compression Algorithm," *International Journal of Engineering and Applied Sciences (IJEAS),* vol. 4, no. 4, pp. 95-96, 2017.

[3] I. T. UNION, "INFORMATION TECHNOLOGY DIGITAL COMPRESSION AND CODING OF CONTINUOUS-TONE STILL IMAGES REQUIREMENTS AND GUIDELINES," *TERMINAL EQUIPMENT AND PROTOCOLS,* pp. a-m, 1992.

[4] M. Adler, H. Popp and M. Hjerde, "MPEG-FAQ: multimedia compression," faqs.org, 2 June 1996. [Online]. Available: http://www.faqs.org/faqs/mpeg-faq/part1/. [Accessed 18 March 2021].

[5] J. Guckert and U. o. Utah, "The Use of FFT and MDCT in MP3," March 2012. [Online]. Available: http://www.math.utah.edu/~gustafso/s2012/2270/web-projects/Guckert-audio-compression-svd-mdct-MP3.pdf. [Accessed 18 March 2021].

[6] C. S. Swartz, "Understanding Digital Cinema:," in *A Professional Handbook*, Taylor & Francis, 2005, p. 147.

[7] D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," September 1952. [Online]. Available: http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf. [Accessed 18 March 2021].

[8] R. Arshad, A. Saleem and D. Khan, "Performance Comparison of Huffman Coding and," *The Sixth International Conference on Innovative Computing Technology,* p. 361, 2016.

[9] J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory,* vol. 23, no. 3, pp. 337-343, 1977.

[10] I. Pavlov, "LZMA SDK (Software Development Kit)," 7-zip, 13 June 2013. [Online]. Available: https://www.7-zip.org/sdk.html. [Accessed 19 March 2021].

[11] J. A. Storer and T. G. Szymanski, "Data Compression via Textual Substitution," *Journal of the ACM,* pp. 928-951, 1982.

[12] WinZip, "Additional Compression Methods Specification," WinZip, 19 May 2009. [Online]. Available: https://www.winzip.com/win/en/comp_info.html. [Accessed 19 March 2021].

[13] T. H. Cormen, in *Introduction To Algorithms (third edition)*, MIT Press, 2009, pp. 428-433.

[14] J. Austen, "Pride and Prejudice by Jane Austen," 1 June 1998. [Online]. Available: https://www.gutenberg.org/files/1342/1342-0.txt. [Accessed 18 January 2021].

[15] python.org, "codecs — Codec registry and base classes," python.org, 5 October 2020. [Online]. Available: https://docs.python.org/3/library/codecs.html. [Accessed 18 January 2021].

[16] python.org, "9. Classes," python.org, 5 October 2020. [Online]. Available: https://docs.python.org/3/tutorial/classes.html. [Accessed 15 February 2021].

[17] python.org, "timeit — Measure execution time of small code snippets," python.org, 5 October 2020. [Online]. Available: https://docs.python.org/3/library/timeit.html. [Accessed 15 February 2021].

[18] S. Griffiths, "bitstring 3.1.7," Griffiths, Scott (shared on Github), 5 May 2020. [Online]. Available: https://pypi.org/project/bitstring/. [Accessed 3 March 2021].

[19] T. Eaves, "Convert a python dict to a string and back," Stackoverflow, 28 December 2010. [Online]. Available: https://stackoverflow.com/questions/4547274/convert-a-python-dict-to-a-string-and-back. [Accessed 12 March 2021].

[20] kite.com, "How to convert binary string to and from ASCII text in Python," kite.com, 16 December 2019. [Online]. Available: https://www.kite.com/python/answers/how-to-convert-binary-string-to-and-from-ascii-text-in-python. [Accessed 12 March 2021].

[21] C. Collodi, "Pinocchio: The Tale of a Puppet by Carlo Collodi," 13 October 2005. [Online]. Available: http://www.gutenberg.org/ebooks/16865. [Accessed 18 March 2021].

[22] C. Collodi, "Le avventure di Pinocchio: Storia di un burattino by Carlo Collod," 3 July 2016. [Online]. Available: http://www.gutenberg.org/ebooks/52484. [Accessed 18 March 2021].

[23] C. Collodi, "Pinocchion seikkailut: Kertomus marioneteista by Carlo Collodi," 18 September 2016. [Online]. Available: http://www.gutenberg.org/ebooks/53077. [Accessed 18 March 2021].

[24] L. Carroll, "Alice's Adventures in Wonderland by Lewis Carroll," 27 June 2008. [Online]. Available: http://www.gutenberg.org/ebooks/11. [Accessed 18 March 2021].

[25] L. Carroll, "Le avventure d'Alice nel paese delle meraviglie by Lewis Carroll," 20 March 2009. [Online]. Available: http://www.gutenberg.org/ebooks/28371. [Accessed 18 March 2021].

[26] L. Carroll, "Liisan seikkailut ihmemaassa by Lewis Carroll," 12 August 2014. [Online]. Available: http://www.gutenberg.org/ebooks/46569. [Accessed 18 March 2021].

[27] P. Ferragina and G. Navarro, "Pizza&Chili Corpus Compressed Indexes and their Testbeds Repetitive Corpus," October 2010. [Online]. Available: http://pizzachili.dcc.uchile.cl/repcorpus.html. [Accessed 18 March 2021].

[28] B. Stoker, "Dracula by Bram Stoker," 1 October 1995. [Online]. Available: https://www.gutenberg.org/ebooks/345. [Accessed 18 March 2021].