The Average Efficiency of Data Compression Algorithms

Is it possible for a compression algorithm to successfully compress a file, on average?

Math Extended Essay

Word Count: 3705

Table of Contents

Introduction

The purpose of this essay is to answer the question: "Is it possible for a compression algorithm to successfully compress a file, on average?". This essay aims to do this by analyzing the PNG image format and its compression mechanisms, namely the DEFLATE algorithm which utilizes LZ77 and Huffman coding, through mathematical modelling, general-purpose formulae, and smaller subset analysis. This essay uses previous academic outlines of compression algorithms, such as the papers outlining the LZ77 and Huffman coding computer science principles, as well as practical documentation from the creators of PNG and DEFLATE on its inner mechanisms.

Brief Overview of Data Compression

At its core, data compression is the concept of encoding a set of data in a specific way to reduce its size or length. Data compression plays a major role in information technology, specifically when dealing with the internet. Every website, image, video, song, and program is compressed before it's sent over the internet and decompressed before it is read. After data is compressed, it needs to be decompressed to be processed properly: a common example is decompressing a ZIP file. Compression is deemed successful if the compressed data is smaller than the decompressed data (which is not always the case, counter-intuitively).

The main focus of data compression in this essay focuses on file compression, a subset of data compression. Namely, file compression operates on computer files using compression algorithms: similar mathematically to a function that takes in a variable *x*, the file, performs operations on it, and outputs variable *y*, the newly compressed file. Famous examples of file

compression include the ZIP archive file, image compression formats such as JPG and PNG, and audio compression formats such as MP3 and OGG.

Inherently, all data compression is a space-time complexity trade-off. To perform compression (and subsequent decompression), non-zero time must be invested into the process to save space, such as spending time to zip a file. For the purpose of this essay, we will completely favour space savings: this allows us to analyze data compression in a perfect scenario.

Lossless versus Lossy Compression

There are two main subsets of file compression: lossless and lossy compression. Lossy compression is predicated on throwing away redundant or not useful data to reduce file size. The problem with lossy compression is that it is irreversible. Once this data is discarded, it is not possible to regain that data on decompression. In theory, this also means that every lossy compression algorithm can reduce file size, as it can always throw away information. Therefore, while it is an interesting field, it is not ideal to gauge the overall efficiency of data compression as a concept.

The second subset of compression is lossless compression. As the name implies, lossless compression does not discard any data, and instead uses statistical redundancy and probabilistic modelling to reduce file sizes. A simple form of lossless compression exists in arithmetic: instead of writing out 5+5+5+5+5, one could write out 5*5, writing out less characters while still keeping the original message intact (and reversible). On decompression, the algorithm then replaces the smaller reference marker with repeating artifact. Since lossless compression fully

retains the data, it is easier for one to analyze and quantify, and also is consistent with the research question in mind.

A Note On True Perfect Compression: Dirichlet's Box Principle

The concept of true perfect compression, or an algorithm that can *always* reduce the size of a file while being reversible, is mathematically impossible. A simple explanation comes from the Dirichlet Box Principle[1]: as defined by the Encyclopedia of Mathematics, Dirichlet's Box Principle states that "any sample of n sets containing in total more than n elements comprises at least one set with at least two elements."[2] While this seems rather intuitive, it has broad implications on data compression, and this essay.

Assume an algorithm which can always reduce a file by one bit, algorithm *X*. If algorithm *X* is applied to all files that are 8 bits long. Since a bit can be either 0 or 1, there are 2^8= 256 possible files. If every single 8-bit file is reduced to 7 bits, there are only 2^7 = 128 possible files that can exist, yet we still have 256 possible files. Using Dirichlet's Box Principle, there are 256 elements but only 128 sets: therefore, at least one set, or 7-bit file, is compressed to from two different elements, or 8-bit files. When decompressing, the algorithm cannot possibly tell which 8-bit file this 7-bit file decompresses to, and therefore the compression is irreversible (and is not true data compression). This means that no algorithm *X* can exist: perfect compression is impossible.

[1] "Dirichlet's Box Principle", Encyclopedia of Math, accessed April 7th 2017, https://www.encyclopediaofmath.org/index.php/Dirichlet_box_principle
[2] "Dirichlet's Box Principle"

A Note On Kolmogorov Complexity

Kolmogorov complexity, simply put, is the smallest computer program required to create a certain object. The Kolmogorov complexity of an object is therefore represented in the shortest amounts of bits possible, and is fully compressed: it cannot be reduced in size further. Knowing this, calculating the Kolmogorov complexity of any object seems like the surefire answer to any data compression related exploration. Unfortunately, Kolmogorov complexity is not computable[3]. That being said, we can always approximate Kolmogorov complexity. Certainly, LZ77 (the compression method explored later in this paper) will not always be the best approximation of Kolmogorov complexity, but that's not necessarily the focus of this essay. Compression algorithms are still functional even when they don't reach Kolmogorov complexity, and this analysis can still function without reaching Kolmogorov complexity.

Hypothesis

Since perfect compression isn't possible, we can shift the focus of our question: is file compression efficient on average? Say, if one were to apply algorithm *x* on every file of size *y*, would the mean file size increase or decrease?

The hypothesis for this essay is that the file size would increase, on average. This is based on two ideas: firstly, that statistical redundancy is not particularly common, and secondly, an extension of Dirichlet's Box Principle. As previously established, the majority of lossless data compression relies on statistical redundancy. Yet, large-scale redundancy is generally not common. If one

[3] "COS597D: Information Theory in Computer Science Lecture 10", Princeton Computer Science Courses, last accessed April 18th 2017, https://www.cs.princeton.edu/courses/archive/fall11/cos597D/L10.pdf

chooses three random letters from the alphabet, it's much more likely that they aren't the same than the possibility that two are, let alone three. This kind of repetition is by nature a statistical outlier, and shouldn't be common enough to create savings on average. In addition, lossless data compression still expresses the same data in a different format. According to Dirichlet's Box Principle, it shouldn't be possible to express the same amount of data in a smaller amount of data, and a natural extension of this is applying it to an average file as well. That being said, this is most certainly not clearly proven: there are mechanisms, such as run-length encoding which replaces large amounts of repetitive data with a multiplicative statement, that might increase most files by a nominal amount, but greatly decrease the size of specific files (and therefore may or may not lower the mean of a compressed file). It is certainly an interesting problem to analyze, and this essay aims to find more concrete proof.

Case Study: Portable Network Graphics (PNG) and DEFLATE

Unfortunately, it's virtually impossible to analyze every possible form of data compression. Narrowing in on specific formats is more feasible, and subsequently can result in a higher level of analysis. One of the best candidates for a narrower scope is the Portable Network Graphics format (PNG), which is an image compression and storage format. PNG one of the most popular image formats in the world, largely due to its ease of use and relative efficiency.

PNG files go through four states as compression is applied: input file, pre-compression filtering, compression (DEFLATE algorithm), and output file[4]. Compression is the most important part of the analysis, as it is the titular focus of this essay. PNG uses *zlib*, an implementation of the

[4] The PNG Group, *PNG: The Definitive Guide*, (Independent Publisher, 1999), Chapter 9

DEFLATE algorithm with a few useful toolsets[5]. The next few sections aim to deconstruct the compression steps that PNG uses to compress files, and assess their efficiency.

As a side note, we'll assume that we are using an 8-bit grayscale colour scheme, which represents how much gray is in a certain pixel from 0 to 255 (the amount of distinct values possible in 8 bits/one byte). Colour formats quantify colour, and do not impact the outcome of mathematical calculations as long as they are consistent before and after compression. PNG also has a vast set of other content features such as interlacing and image tracing that don't impact visibility of the image. As such, we'll strip the file of all of these artifacts.

DEFLATE

In the PNG format, the DEFLATE algorithm conducts the main principle of data compression, eliminating statistical redundancy[6]. DEFLATE itself is based upon Huffman coding and the LZ77 algorithms[7], which are both mechanisms to reduce or remove redundant data. PNG uses *zlib*'s implementation of the DEFLATE algorithm, which is the flavour that will be analyzed in this essay. While *zlib* contains a few quirks, we can "configure" our theoretical version of *zlib* to act in the way we want: specifically, to perform maximum compression when possible.

[5] *PNG: The Definitive Guide,* Chapter 9
[6] *PNG: The Definitive Guide,* Chapter 9
[7] "DEFLATE Compressed Data Format Specification version 1.3", Internet Engineering Task Force, accessed April 7th 2017, https://tools.ietf.org/html/rfc1951

LZ77

LZ77 is a lossless compression algorithm developed by IEEE members Jacob Ziv and Abraham Lempel in their 1977 paper titled *A Universal Algorithm for Sequential Data Compression*[8]. In their paper, Ziv and Lempel outline an algorithm that replaces sets of repeating data with references to previous occurrences of the data, to reduce redundancy. With long sets of repeating data, as often is the case in images, LZ77 can be very effective.

Figure 1: Sample Byte Sequence 1[9]

Position	1	2	3	4	5	6	7	8	9
Byte	A	A	B	C	B	B	A	B	C

[8] Jacob Ziv and Abraham Lempel, "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory* IT-23 No. 3 (1977): 337

[9] "LZ77 Compression Algorithm", Microsoft Developers Network, last accessed April 7th 2017

Figure 2: Step-by-step Analysis of LZ77 on Figure 1[10]

Step	Pointer Position	Previous Byte(s) Match	New Byte	Output
1.	1	--	A	(0,A)
2.	2	A	--	(1,1)
3.	3	--	B	(0,B)
4.	4	--	C	(0,C)
5.	5	B	--	(2,1)
6.	6	B	--	(1,1)
7.	7	A B C	--	(5,3)

In every step, the "pointer", or the current byte being processed, moves one byte to the right. It then iterates through the entire sequence that it has already parsed (as there is no sliding window), to look for any matching bytes or byte sequences. In Step 1, 3, and 4, it finds no matching sequences: as such, it outputs the new byte. In all the other steps, it has found a previous byte that matches it: instead of re-outputting this redundant byte, it stores how many bytes back the pointer moves to reach the closest redundant byte, as well as the length of the

[10] "LZ77 Compression Algorithm", Microsoft Developers Network, last accessed April 7th 2017

redundant bytes. This is especially useful in step 7, where the algorithm realises that "ABC" is repeated already in positons 2-4, and therefore can encode positions 7-9 in two bytes.

From our previous analysis, it is clear that LZ77 has specific uses in compressing data with many redundancies. Only 1/3rd of the entire sequence is unique, and by using the LZ77 algorithm one is able to exploit this redundancy and reduce the size of our sequence. The ability for LZ77 to encode large repeating sets of data in smaller segments, such as encoding positions 7-9 (which is 1/3rd of our entire sequence) in one byte, makes it very versatile for certain types of files.

Using this analysis of the LZ77 algorithm, one can develop a formula to represent the efficiency of its compression:

$$pre\ LZ77 = encoding\ header + (N\ units)(unit\ length)$$

$$post\ LZ77 = encoding\ header + (P\ unique\ units)(unique\ identifier\ length) + (O\ duplicate\ units)(marker\ length)$$

Where *encoding header* is an identifying set of units at the beginning of a file the state of the file, *N units* is how many of the smallest division of data exist in a certain file, *unit length* is how long each *unit* is, *P unique units* is how many units are unique, *O duplicate units* is how many units are not unique, *unique identifier length* is how long the identifier for a unique unit is, and *marker length* is how long the back-referencing marker is in units. In addition, in the majority of cases, the *encoding headers* for both files will be the same length (as they are often reserved by filesystems), so they can be ignored in calculations.

This formula can be used on the previous example. Assuming ASCII encoding (which uses 8 bits, or one byte, per character[11]), Figure 1 *pre LZ77* is $(9\ units)(1\ byte) = 9\ bytes$ long. If LZ77 is then applied to Figure 2, it is then $(3\ unique\ units)(2\ bytes) + (4\ duplicate\ units)(2\ bytes) = 14\ bytes$ long (note that there are 4 *duplicate units*, and not 6, as "ABC" counts as one *duplicate unit*). This is rather surprising: for a *compression* method, LZ77 has somehow made our file drastically larger!

Figure 3: Sample Byte Sequence 2

Position	1	2	3	4	5	6	…	22	23	24
Byte	A	B	C	A	B	C	…	A	B	C

In this scenario, the string "ABC" simply repeats itself eight times. The output set from LZ77 compression would be (0,A)(0,B)(0,C)(3,3)(6,6)(12,12). We can once again apply our formula to see if this compression is efficient.

Figure 3 *pre LZ77* is $(24\ units)(1\ byte) = 24\ bytes$ long. If LZ77 is then applied to Figure 3, it is then 12 bytes long. This value checks up with our output set from LZ77 compression above (6 pairs of 2 bytes is 12 bytes). In this scenario, there is successful compression: largely due to the fact that this sequence is very, very repetitive.

[11] "ASCII Table", ASCII Table Website, accessed April 5th 2017, http://www.asciitable.com/

Mathematical Modelling

Both of these are only specific examples, and don't necessarily prove or disprove anything. There exists thousands of examples that can be compressible, but that doesn't necessarily make LZ77 efficient on average. Yet, it is possible to calculate pre-compression and post-compression values for every single possible file of a sequence length, and every amount of possible values in a single unit.

The pre-compression formula is quite simple. It uses the *k*-permutations of *n* formula,

$$P(n,k) = \frac{n!}{(n-k)!}$$

This formula tells us how many permutations exist in set *n* with *k* possible values. Substituting possible values (*pv*) for *n* and *sequence length* for *k*, and multiplying everything by sequence length (*sl*) and unit length (*ul*), which is the size of an individual file, we get:

$$pre = \frac{(pv)!}{(pv-sl)!}(sl)(ul)$$

Creating a formula for post-LZ77 compression is a bit harder. Unlike the pre-compressed file, we cannot multiply the file size by the number of files, as the file size is variable. Instead, we need to do a summation of all possible different file lengths (with *n* as a counter based on how many duplicate characters there are), divided by the different amount of summations that we did, or the sequence length. This is similar to adding up all values in a set, and taking an average. The first step inside the summation is to pretend that all back references, or duplicate values, didn't exist.

$$\frac{\sum_{n=0}^{sl} \frac{(pv)!}{\big(pv-(sl-n)\big)!}}{sl}$$

The next step is to multiply our numerator (inside the summation) by the number of possible permutations that exist for our duplicate files. Unlike the previous permutation calculation, we are okay with repetition, as repeated duplicates are allowed.

$$\frac{\sum_{n=0}^{sl} (sl-n)^n \frac{(pv)!}{\big(pv-(sl-n)\big)!}}{sl}$$

Finally, we multiply the summation by the size of each file with *n* duplicates, using the post LZ77 formula we derived earlier.

$$post = \frac{\sum_{n=0}^{sl} \big((sl-n)(ul)+(n)(ml)\big)(sl-n)^n \frac{(pv)!}{(pv-(sl-n)\,)!}}{sl}$$

If, on average, LZ77 is able to compress files, then *pre* > *post*. If not, *post* > *pre* (we'll discount ties, which is highly unlikely). Seeing that our hypothesis believes that *post* > *pre*, let's test that.

Theorem: *pre* < *post*

$$\frac{(pv)!}{(pv-sl)!}(sl)(ul) < \frac{\sum_{n=0}^{sl} \big((sl-n)(ul)+(n)(ml)\big)(sl-n)^n \frac{(pv)!}{(pv-(sl-n)\,)!}}{sl}$$

Assuming *ul* = *ml* , which is a very generous assumption, and multiplying both sides by *sl*:

$$\frac{(pv)!}{(pv-sl)!}(sl^2)(ul) < \sum_{n=0}^{sl} (ul)(sl)(sl-n)^n \frac{(pv)!}{(pv-(sl-n)\,)!}$$

$$\frac{(pv)!}{(pv-sl)!}(sl) < \sum_{n=0}^{sl} (sl-n)^n \frac{(pv)!}{\big(pv-(sl-n)\big)!}$$

$$\frac{(pv)!}{(pv-sl)!}(sl) < \frac{(pv)!}{(pv-sl)!}(sl) + \sum_{n=1}^{sl} (sl-n)^n \frac{(pv)!}{(pv-(sl-n)\,)!}$$

This last statement is impossible! The first term of the summation $\sum_{n=0}^{sl}(sl-n)^n \frac{(pv)!}{(pv-(sl-n))!}$ is going to be $\frac{(pv)!}{(pv-sl)!}(sl)$, or the right side of the inequality. Every additional summation will be another positive integer added to the right side of the inequality, which will always guarantee that left side < right side. Therefore, by proof of impossibility, *post* > *pre*, and therefore compression is inefficient on average.

Testing

I was able to test this theorem using a computer program that I wrote in Python (attached in Appendix 1.1), by converting our theorem into a function. Applying our formula for a sequence length of 8 and a possible value of 256 yielded a data compression rate of 0.06, or that the pre-compressed data was 0.06 times as large as the post-compressed data. A sequence length of 256 yielded an abysmal data compression ratio of 0.0007. From our computer algorithm, we can confirm our mathematical proof: compression is inefficient on average.

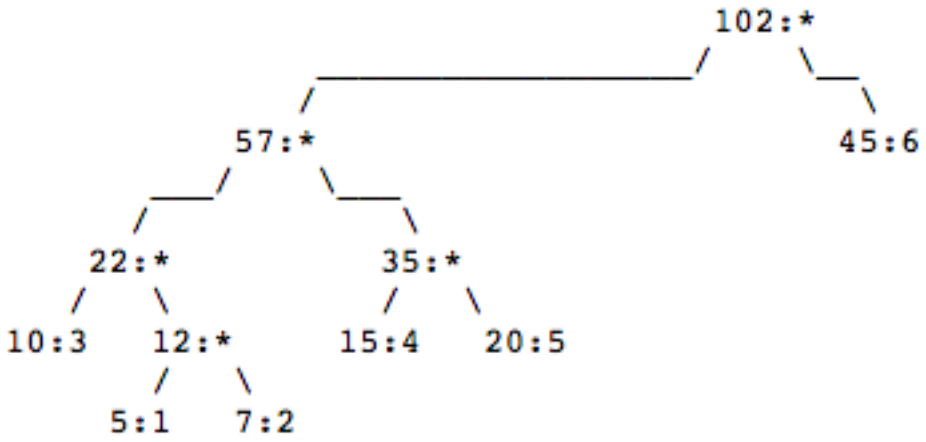
Huffman Coding

Huffman coding is another mechanism that DEFLATE uses to compress data, using the work that David A. Huffman outlined in his 1952 paper "*A Method for the Construction of Minimum-Redundancy Codes*"[12]. Essentially, Huffman coding uses a binary tree to assign higher frequency characters lower bits, and lower frequency characters higher bits. Giving high frequency characters lower bit assignments optimizes compressed file size, especially for files that have

[12] David A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", *Proceedings of the I.R.E.* (1952): 1098-1102

chunks that are large in size and often repeated. In DEFLATE, Huffman coding is applied to the post-LZ77 compressed file[13], and is usable on both character data and reference markers.

Figure 4: Sample Huffman Coding Table and Tree[14]



Character	Frequency	Code
1	5	0010
2	7	0011
3	10	000
4	15	010
5	20	011
6	45	1

[13] "DEFLATE Specification", zlib Website, last accessed April 9th 2017
[14] "A Quick Tutorial on Generating a Huffman Tree", Siggraph, last accessed April 8th 2017, https://www.siggraph.org/education/materials/HyperGraph/video/mpeg/mpegfaq/huffman_tutorial.html

In the above figure, a sample set of data and Huffman coding is present. Huffman algorithms don't require the positions of the characters, but rather only the frequency. Then, a tree is generated using the frequency of each character. Starting from the two lowest frequency characters ("1" and "2" in this case), a branch of a tree is built, with the branch converging on the additive frequency of both characters. This process is then repeated until one whole tree is built, with the top-most number being the additive frequency of all characters. Then, the code is built: when traversing the tree, a "0" signifies going down the left branch, and a "1" signifying the right. This final code table (just columns 1 and 3 in the above figure) is then stored along with the compressed file, which is simply the file size of all characters + the codes. In some cases, Huffman tables aren't stored as part of the data file, but rather are intrinsic to the compiler; for the sake of this essay, we'll assume that they're stored with the data. Something important to note is that the code only uses 0's and 1's: each character in the code is a single bit.

In Figure 4, the pre-Huffman file is 102 bytes long. After applying the Huffman algorithm, the Huffman code is $228\ bits$ long, or 28.5 bytes long. Adding the Huffman table to this set, which is 11.625 bytes long (adding up all characters and codes), the final post-Huffman file is 40.125 bytes long. In situations with large amounts of repetition, such as Figure 4, Huffman coding is very effective.

Mathematical Modelling

Mathematically modelling the Huffman tree is particularly hard, and even harder when not generating every individual Huffman tree. But, we can look at the effectiveness Huffman coding in the most probable scenarios. In the previous mathematical model, we established that we can

calculate the number of possible decompressed files with $pre = \frac{(pv)!}{(pv-sl)!}(sl)(ul)$, and the number of decompressed files with duplicates with $(sl-n)^n \frac{(pv)!}{(pv-(sl-n))!}$.

We can create a rudimentary (but inaccurate) estimate for the length of a Huffman tree + table, where *m* is the number of unique elements, *ul* is the length of a unique element, *acl* is the average code length, and *sl* is the sequence length.

$$length = (m)(ul + acl) + (sl)(acl)$$

Let's say that we have a file that is 256 bytes long, with 256 possible values. We can have a generous 50% duplicate rate, with half of the files being duplicates of the other half. We also assume that *acl* is 0.5, which is also quite generous for a tree with so many nodes. Applying that to our rudimentary formula,

$$length = (128)(1.5) + 128(0.5) = 256\ bytes$$

Zero compression. This is one of the better case scenarios with a lot of duplicates.

The number of 256-byte files that exist $(256)!/(256-256)!)(256) = (256!)(256)$. It is much more likely that a file has less duplicates than more (this is intuitive, but can also be explained with probabilities using the *k*-permutations of *n* theorem previously discussed), and therefore the average Huffman coded file in this example will not successfully compress. While this is a case-by-case example, this same kind of logic extends to all sets of Huffman coded data.

Conclusion

Ultimately, our hypothesis was confirmed. Using PNG's compression mechanisms, average effective data compression is not possible. While in theory this doesn't extend to every type of file compression, DEFLATE is an industry standard that falls very, very short in achieving average compression: it seems reasonable to assume that other mechanisms also have similar results. In addition, the proof of the rather extreme rarity of large statistical redundancy, demonstrated in the LZ77 and Huffman coding analysis, hint that other methods that take advantage of statistical redundancy would also be inefficient.

One thing to note is that all of the posed calculations have a margin of error: they don't necessarily reach Kolmogorov complexity, and the models for LZ77 and Huffman coding are both approximate. Yet, the difference in size of files is so large (factors of 10^3 in our LZ77 example) that even with a generous margin of error average data compression is still not possible. Of course, more refined modelling, and possibly actually performing LZ77/Huffman coding would confirm this (unfortunately, hard to do due to lack of computational resources).

Then, why is data compression used at all? Well, the majority of files that undergo compression aren't randomly generated, "average" files: most images have groups of pixels where colours are the same or very similar, and most text follows patterns (this essay, for example, has many repeated occurrences of the word "compression"). In addition, concepts such as conditional algorithms help bolster the overall efficacy of compression, without the huge negative outcomes that exist when the file just isn't right.

References and Bibliography

The PNG Group, *PNG: The Definitive Guide*, (Independent Publisher, 1999), accessed from http://www.libpng.org/pub/png/book/

PNG: The Definitive Guide is an online e-book that documents the inner-workings of the Portable Graphics Network image storage format, written by the creators of PNG (the PNG Group). This book was a valuable resource as an initial point of research: it covered a broad set of topics about PNG, including compression, and gave other sources to further research specific libraries and methodologies.

Jacob Ziv and Abraham Lempel, "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory* IT-23 No. 3 (1977): 337, accessed from https://www.cs.duke.edu/courses/spring03/cps296.5/papers/ziv_lempel_1977_universal_algorithm.pdf

IEEE members Jacob Ziv and Abraham Lempel published *A Universal Algorithm for Sequential Data Compression* in *IEEE Transactions on Information Theory* in 1977, outlining a ground-breaking sliding window compression algorithm, later used in the LZ77 algorithm. This paper was an invaluable resource for me, as it delved into the finder technical details of the algorithm, and explained the reasons why LZ77 was so important in the history of computer science.

David A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", *Proceedings of the I.R.E.* (1952): 1098-1102, accessed from http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf

Then MIT student David Huffman published *A Method for the Construction of Minimum-Redundancy Codes* in 1952, and gave initial insight into what would be known as Huffman coding. His work was used in the DEFLATE algorithm, and his paper was great insight into the revolutionary ideas of Huffman coding and how it functioned.

"zlib Website", zlib Group, last modified April 3rd 2017, http://www.zlib.net/, http://www.zlib.org/rfc-deflate.html, http://www.zlib.net/feldspar.html

The *zlib* website is a practical documentation set that explains the mechanisms of the *zlib* library, written by the zlib Group. It helped fill in gaps in documentation from official academic papers and the PNG Group's work on the topic, especially expanding on Huffman Coding and the entire DEFLATE process.

"LZ77 Compression Algorithm", Microsoft Developers Network, last accessed April 7th 2017, https://msdn.microsoft.com/en-us/library/ee916854.aspx

This webpage, part of the Microsoft Developers Network documentation set, gave specific examples on the mechanisms of LZ77. It was helpful in understanding the bit-to-bit process of LZ77, something which abstract academic papers often lack.

Appendix 1.1: Sample Code for Estimation of LZ77 Compression Length

```
import math # Required for math.factorial

sequence_length = 256 # Length of Sequence
possible_values = 256 # Number of possible data that can go into one unit
unit_size = 1 # How large one unit is.
unique_length = 2 # How large a unique reference is.
marker_length = 2 # How large a backreference marker is.
total_precompressed_size = (math.factorial(possible_values)/math.factorial(possible_values-sequence_length)) * sequence_length
* unit_size # This is the permutation formula applied to the pre-compressed files.
total_postcompressed_size = 0 # This starts off as zero, as the for loop below increments it.

print "Sequence Length: " + str(sequence_length)
print "Possible Values: " + str(possible_values)
print "Size of all pre-compressed files: " + str(total_precompressed_size)

for i in range(sequence_length):
    # In this for loop, i is a counter for how many duplicate units there are.
    # This function is a bit complicated. It loops through how many possible files exist at i number of duplicate units, and
    multiplies that quantity by the post-LZ77 filesize of that file.
    # In order to calculate how many files exist at i number of duplicates, we pretend that the duplicates don't exist; find
    the number of files that fulfill only the uniques, and then use repetitive permutation to find what sets of characters
    could fill the unique sets.
    total_possible = math.factorial(possible_values)/(math.factorial(possible_values - (sequence_length-i))) *
    (sequence_length-i)^i
    total_postcompressed_size += (total_possible) * (sequence_length-i) * ((sequence_length-i) * unique_length + (i) *
    marker_length)
    # Note: the rightmost block starting with "((sequence_length-i" is just our LZ77 post-compression formula turned into
    python code.

total_postcompressed_size = (total_postcompressed_size * unit_size)/sequence_length

print "Size of all post-compressed files: " + str(total_postcompressed_size)

# Data compression ratio is just pre-compressed/post-compressed
print "The data compression ratio is " + str(float(total_precompressed_size)/float(total_postcompressed_size))
```

Appendix 1.2: Sample Code Output

```
10688:EE matthew.wang$ python possible-set-duplicates.py
Sequence Length: 8
Possible Values: 256
Size of all pre-compressed files: 132141121548226560000
Size of all post-compressed files: 2120778021134308470440
Data Compression Ratio: 0.0623078512
10688:EE matthew.wang$ python possible-set-duplicates.py
Sequence Length: 64
Possible Values: 256
Size of all pre-compressed files: 154650235387378427392367245031499710911076048239435
179215831292966090978516359515654705751351677501920294332286819453564212449303225958400000000000000000
Size of all post-compressed files: 198951145819809185568051540990343144083637176673807485985245126820532904651187305146372541423481352515850277450813922274814591990449663936792243101831176320
Data Compression Ratio: 0.0077732769
10688:EE matthew.wang$ python possible-set-duplicates.py
Sequence Length: 256
Possible Values: 256
Size of all pre-compressed files: 219601350487767719454485061550395552040392069191548318684071846496580979853605782880624714588275560563296226647638165582376242389681586297682506057855341062573987157136222409329324853152459407408120030200709412686771731846821491574965551955481813216339379921659078059976052267584133892763774801681443393915153984721295221626108680874422460963657268540097742904028339508124437286935936498543899930042103875292571203497583365847281713000556701679616000000000000000000000000000000000000000000000000000000000000000000000000
Size of all post-compressed files: 303254014314153534353973884057649238860023226757496643166190675920472695869091673823389087330764196891973813849947264826511784527390533368737714221026584203916531925822038878570384587353753602908248796315730128122150377412831764213115119855299924792362735815610317738440436307686444842483193177962877219806380390922475705855905919969271420280122493522097574040244102875187027484214160371136707895727543994638192899984491260504937382682804318537237240703539739190903426991193768691983908045962848079224310188622592
Data Compression Ratio: 0.0007241498
```