

Domain Based Lossless Text Compression — Report

Prateek Rungta
Supervised by Arun Mani

FIT 2044 Second Year Advanced Project
Semester 2, 2007

Table of Contents

Introduction	1
Understanding GNU zip	2
Huffman Coding	
Lempel Ziv (LZ77)	
DEFLATE	
Domain Based Compression	3
Domain Knowledge	
Attempt 1	3
Attempt 2	4
Attempt 3	5
Conclusion	6
References	7

Introduction

Text Compression

There are a lot of different file types in today's computer systems, but we shall limit ourselves to plain text files that are made up of only ASCII characters. Text files are, in general, relatively smaller to binary and multimedia files.¹ Nevertheless, a significant amount of resources can be saved by reducing the sizes of these files too. The two most obvious area where one might expect to see gains are in disk space and bandwidth usage. This is what text compression aims to achieve.

Domain Based and Lossless Compression

Lossless compression means preserving all the data in the original file. The uncompressed file should be exactly the same as the original file, down to the last bit. This is important because, as we will see later, keeping the compression lossless adds certain constraints.

Domain based compression methods are those which make use of domain knowledge of the file for its compression. The project was to explore domain based compression techniques.

¹ Of course, there are times when certain text files may be larger than most binary files. A good example would be log files.

Understanding GNU zip

Before delving into ways in which a file's domain knowledge could be used for better text compression, we wanted to understand how text compression works. GNU zip, more commonly known as gzip was the tool we chose to analyze. We chose gzip for two reasons. It is arguably the most popular and widely used text compression tool in the industry and it is an open-source project.

Huffman Coding

I started by looking at the Huffman Coding algorithm. From Wikipedia, "Huffman coding is an entropy encoding algorithm used for lossless data compression." ("Huffman coding", Wikipedia). The algorithm works by substituting input symbols (characters for example) with codes. First, a probability distribution tree is built by traversing the entire file and recording the number of occurrences of each symbol in the file. A variable length, prefix-free code is then assigned to each symbol in the tree. The code is assigned such that more frequently occurring symbols are represented by shorter codes in order to increase the compression ratio². All symbols in the file are now replaced with the code assigned to them. Finally, the probability distribution tree of the symbols is appended to the file. This is done so that the decoder can re-substitute the codes with the original symbols. Huffman coding was published in 1952 by David A. Huffman in his paper "A Method for the Construction of Minimum-Redundancy Codes".

Lempel Ziv (LZ77)

Lempel Ziv is another text compression algorithm named after its creators – Abraham Lempel and Jacob Ziv. The LZ77 algorithm works by using a length-distance pair of numbers to encode data. The encoder starts reading the data stream to be compressed. It keeps track of what it has read in a fixed and finite size buffer. Whenever it encounters data that it has already read, that portion is replaced with a pointer back to the previous occurrence of the data. The pointer is a length-distance pair, indicating that this string of length 'l' characters is the same as the string of 'l' characters distance 'd' places behind. Noted that it can use relative distances. Since a fixed size buffer is used, it cannot keep track of all the data that has been passed through. The most recent data is kept and the old one is discarded as the encoder moves through, giving it the name "sliding window compression". ("LZ77 and LZ78", Wikipedia)

DEFLATE

DEFLATE is the text compression algorithm used in gzip and was developed by Phil Katz ("DEFLATE", Wikipedia). It is a combination of both Huffman Coding and LZ77. The file to be compressed is split into blocks. It first runs the LZ77 algorithm replacing duplicate patterns with pointers. It then applies dynamic-Huffman encoding on each block to encode the symbols with shorter codes.

Source Code

The GNU zip source code is freely available from the GNU foundation. The software is written in C by Jean-loup Gailly and Mark Adler ("The gzip home page", GNU). The first version (0.1) was released in 1992.

This was my first experience with analyzing a software with a code base as large as gzip's and one that has been around for about 15 years. It took little time getting comfortable with C after having used both C++ and Java. After two code-walkthroughs with my supervisor, I was able to grasp some of the inner workings of the software.

² Compression ratio = (original size - compressed size) ÷ original size

Domain Based Compression

After having looked at gzip, we decided to implement our own compression techniques which would make use of the file's domain knowledge to achieve a better compression result.

Domain Knowledge

There are two vital pieces of information that are available by knowing a file's domain. They are structure and content information. These are the two characteristics that enable us to split files into different domains in the first place, ensuring similar structure and content for all files belonging to a single domain.

Let us consider an XHTML file as an example to aid in the understanding of these two features. Being an XML file, we can expect it to have a strict Document Object Model (DOM) structure. Similarly, we can expect the tag names to be those defined in the Document Type Definition (DTD).

However, we note that there is no guarantee that files are going to be syntactically and semantically valid. A file may not follow the grammar defined for its domain. Since we are looking at Lossless compression, it is necessary to retain each file's individual semantics, thus limiting us from making use of the domain's grammar rules.

Attempt 1

The first attempt was a proof-of-concept. We wanted to see if replacing commonly occurring content with shorter codes would provide any gains. Content here was defined as individual words, henceforth referred to as keywords. Since this was a proof-of-concept, we chose to operate on a domain with a very small keyword-set – Cascading Style Sheets (CSS).

Concept

The approach was to hard-code a set of predetermined keywords into the program and assign a code to each of the keywords. The program would go through the file to encode and replace any matching keyword with its assigned code. The decoder would do the same, only this time it would replace the codes with the keyword it represented.

Results

Original	GNU zip	Attempt 1 + GNU zip	Difference
7578 bytes	2221 bytes	1987 bytes	10.53%
3329 bytes	1262 bytes	1113 bytes	11.80%

Reflection

From this exercise we were able to draw a conclusion that replacing keywords with shorter codes can provide some gains in text compression. However, we do not expect to see similar gains as CSS because, as mentioned before, CSS has a very limited set of keywords that form the bulk of the file's contents.

Another reason why this approach could not be used is that picking out keywords for most domains is neither straightforward nor a scalable task. We should also keep in mind that the codes assigned to the keywords have no relation to the actual frequency of their occurrence in files other than the programmer's intuition.

Attempt 2

From the first attempt we realized that while substituting words with shorter codes works, we needed a better method to both pick out the keywords to encode and to assign codes to those keywords. The domains chosen for this attempt were C source code and an English novel in plain text format.

Concept

The program analyses the input file in its first pass and builds a keyword–frequency table. The table is then sorted in descending order according to the keyword’s frequency. Codes are assigned to the keywords from the top of table, resulting in shorter codes being assigned to more frequently occurring keywords. The program replaces the keywords with their codes in the second pass. Finally, it appends the list of keywords to the end of the file, maintaining the descending order. The decoder extracts this table and uses it to re-build the keyword–code table. It then goes over the entire file, replacing any codes with the corresponding keywords.

Keywords have a minimum length of 4. This is to prevent cases when the code for a keyword will be as long as or longer than the keyword it is meant to replace. The total number of keywords is also limited to a certain number to keep the length of the generated codes small. Codes are generated using a set of ASCII characters (33 to 254 with a few exceptions) and are prefixed with the ASCII character 255. Also, only keywords with a frequency of 10 or more are replaced.

Results

Original	GNU zip	Attempt 2 + GNU zip	Difference
53244 bytes	16915 bytes	16676 bytes	1.41%
1527 Kilobytes	562 Kilobytes	568 Kilobytes	-1.05%

Reflection

This attempt gave interesting results along with a time overhead. The delay was caused since we needed significant amount of pre-processing on every file to be compressed. The results were interesting because, as the second result shows, gzip was able to perform a better compression without the keywords being replaced by shorter codes.

We also noted that this program performed exactly the same operation irrespective of the domain of the file. In other words, there was no *real usage* of the file’s domain knowledge. While that might not necessarily be a bad thing, the aim of our project was to make use of the domain knowledge.

Attempt 3

From the first two attempts, we realized that keyword substitution can result in a better compression ratio. However, we wanted to make better use of the file's domain information. Our third and final attempt thus was a combination of the previous two methods. This being our final attempt, we tested it on a lot of domains—C source, HTML, XML, plain English.

Concept

The program, like the previous two, replaces keywords with codes, but unlike the other two, does not have the keywords hardcoded or generated by analyzing each file individually. Instead, domain-specific keywords are generated by a one-time analysis of a group of sample files belonging to that domain. The resultant keyword table (sorted in descending order) is then stored in a 'keys' file. Thus, each domain has its own 'keys' file. In order to encode a file, the program uses that domain's 'keys' file to populate the codes and keywords table, and then substitutes them in one pass. Because the keywords are saved in an external file, there is no need to append the keyword table at the end of the encoded file. The decoder also populates its table by making use of that domain's 'keys' file.

The keyword specifications etc. remain the same as in Attempt 2.

HTML Results

Original	GNU zip	Attempt 3 + GNU zip	Difference
13954 bytes	4189 bytes	4230 bytes	-0.97%
33351 bytes	10504 bytes	10601 bytes	-0.92%

Monash HTML Results

Original	GNU zip	Attempt 3 + GNU zip	Difference
7192 Kilobytes	1224 Kilobytes	1203 Kilobytes	1.73%
767876 bytes	116006 bytes	114030 bytes	1.70%
149307 bytes	14885 bytes	14722 bytes	1.09%

(Wikipedia) XML Results

Original	GNU zip	Attempt 3 + GNU zip	Difference
65548 bytes	24817 bytes	24364 bytes	1.82%
17.5 MB	6.5 MB	6.4 MB	1.57%

Reflection

Let us discuss the variation in results first. For the first set (HTML Results), the keys file was created by analyzing the homepages of 25 different websites. As we can see, our program hindered gzip from doing a better compression job. However, for the second set of results we restricted the domain a bit further. The keys file was generated by analyzing about 600 different pages from the Monash University website (www.monash.edu). That keys file was then used to encode other pages from the Monash University website and, as we can see, we were able to achieve better compression ratios. Similar improvements in the compression ratio can be seen in the Wikipedia (en.wikipedia.org) XML files. This proves that restricting the domains can result in better compression by our program.

Since the pre-processing in this case is done beforehand and only once, it reduces the time overhead that the second attempt introduced.

Conclusion

The Project

From the 10 weeks of work that we did on this project, we saw that while the naïve, or rather unsophisticated approach of ‘find and replace’ is a good first step, it may not always result in compression benefits when post-compressing with gzip. In order to achieve a much higher percentage gain, one will probably need to take up a radically different approach.

Making use of the domain knowledge for compression is limited greatly by the need to ensure lossless compression.

The Process

This project was a ‘first-of-its-kind’ experience for me. It gave me experience in dealing with a real-world problem/solution. I gained knowledge about how text compression works and the problems that need to be tackled by a lossless text compression program. I learnt how to approach a problem, about the importance of domain knowledge and the properties that can be determined from a file’s domain.

I also learnt a lot about reading and understanding large and old code bases that have been optimized and scrutinized by a lot of people over the years.

The project was a platform for me to apply the things I had learnt in my course so far, as well as to explore and learn new techniques used in the computer science field.

It was also my first experience of working individually under a supervisor and it turned out to be a very pleasant experience.

References

1. Lempel, Abraham and Ziv, Jacob. "A Universal Algorithm for Sequential Data Compression." *IEEE Transactions on Information Theory*. Vol. IT-23, No. 3, (1977): 337-43.
2. Gailly, Jean-loup. "ALGORITHM.DOC" documentation file. *GZIP 1.3.12 Source*. <<http://cudlug.cudenver.edu/GNU/gnu/gzip/gzip-1.3.12.tar.gz>>
3. "Huffman coding." *Wikipedia, The Free Encyclopedia*. 29 Oct 2007, 00:13 UTC. Wikimedia Foundation, Inc. 4 Nov 2007 <http://en.wikipedia.org/w/index.php?title=Huffman_coding&oldid=167738557>.
4. "LZ77 and LZ78." *Wikipedia, The Free Encyclopedia*. 4 Nov 2007, 07:48 UTC. Wikimedia Foundation, Inc. 4 Nov 2007 <http://en.wikipedia.org/w/index.php?title=LZ77_and_LZ78&oldid=169115229>.
5. "DEFLATE." *Wikipedia, The Free Encyclopedia*. 28 Oct 2007, 02:12 UTC. Wikimedia Foundation, Inc. 4 Nov 2007 <<http://en.wikipedia.org/w/index.php?title=DEFLATE&oldid=167547597>>.
6. "Gzip." *Wikipedia, The Free Encyclopedia*. 1 Nov 2007, 10:00 UTC. Wikimedia Foundation, Inc. 4 Nov 2007 <<http://en.wikipedia.org/w/index.php?title=Gzip&oldid=168479458>>.
7. "The gzip home page." *GNU Foundation*. Last modified 27 July 2003 <<http://www.gzip.org>>