Data Compression

Lucas Garron: August 4, 2005

I. Introduction

In the modern era known as the "Information Age," forms of electronic information are steadily becoming more important. Unfortunately, maintenance of data requires valuable resources in storage and transmission, as even the presence of information in storage requires some power. However, some of the largest files are those that are in formats replete with repetition, and thus are larger than they need to be. The study of data compression is the science which attempts to advance toward methods that can be applied to data in order to make it take up less space. The uses for this are vast, and algorithms will need to be improved in order to sustain the inevitably larger files of the future. Thus, I decided to focus my research on the techniques that successful methods use in order to save space. My research question:

What common features do good methods of data compression share?

II. Mathematical Context

The history of data compression is not so much a continent of improvement as it is an archipelago of dispersed -but related- innovations in the subject of information theory. The reason for this is mostly its relatively new development. Many topics in mathematics are now mostly researched in terms of computing. However, most of these subjects were already fairly developed before the arrival of computers. Cryptography, for example, was used since ancient times to keep information secret, and has only now developed into methods that assume the use of a computer. In contrast, computers are almost a requirement for data compression as a theory to be of practical utilization: Analog information can easily be compressed by recording it in less space. Anyhow, there is rarely a need to store information into smaller space; it would make little sense to spend time turning text into a jumble to save some size if simply writing it smaller would suffice (and would also be much easier to read).

The chronicle of modern data compression begins in the late 1940's with a method designed by Claude Shannon and Robert Fano, logically named Shannon-Fano coding. It takes the frequencies of the symbols in a file to represent the more recurrent ones using shorter coding. This was improved upon by one of Fano's students, David Huffman, into an optimal version, and has obsoleted Shannon-Fano coding (except for historical purposes).

The next step up, arithmetic coding, is similar to, and even better than Huffman coding, but it requires more resources to implement. Also, it is burdened with patentship, one of the most adverse (in terms of adoption) properties of data compression algorithms. The same also applies to others such as LZW, MP3 and more obscure ones. LZW, standing for Lempel-Ziv-Welch, was developed by Terry Welch, and is still patented, and used in .gif and .pdf files.

Mp3, developed by the German 'Fraunhofer Society' in the 1990's, is used for compressing sounds by removing inaudible redundancy and frequencies. It is, however, a *lossy* compression algorithm, called so because it loses some information that is not essential in order to save a lot of space. I decided to gear research mostly to lossless compression algorithms, and will mention little more about these.

The ZIP file format is also worthy of mention in any account of data compression. It was developed by Phil Katz in the late 1980's. Its usefulness for archiving by putting multiple files into one archive helped it spread, and now it is almost the default algorithm used for common compression.

Methods used in data compression:

During the course of my research, I decided to focus not so much on algorithms themselves as the methods that they employ, some of which are occasionally referred to as algorithms themselves.

Most compression algorithms are designed to run on the format of data used by virtually all computers nowadays:

The basic unit is a bit, either a one or a zero.

A byte is a collection of 8 bits (in order). Since $2^8=256$, there are 256 possible bytes, from 0 to 255. These can represent either numerical information or certain characters. In the latter case, the character is most often encoded by the ASCII (American Standard Code for Information Interchange), where every number stands for a letter or symbol ("A"=65, "7"=55, "$"=36). Successive characters together are commonly referred to as a "string," denoted by putting it in quotes. Note, though, that I will often use the word "character" loosely to refer to any byte (in the sense of a unit of information), including numbers, though it may not be coded by eight bits.

A. Useless Coding

It is very easy to create a code that makes any given message longer: Prepend some bits (that is, add a few extra ones to the front), add a useless bit into each byte (giving a byte length of nine, that is), or do something else just as ineffective. However, it is impossible for an algorithm to *always* shorten a message, since all data would eventually end up at one bit. But if an algorithm makes *some* data shorter, it must also make some data *longer*, since it impossible to "map" a larger set of potential data for a given length into a smaller set. There is also the possibility of creating a method that doesn't change the length, such as reversing, or switching two successive digits.

As an interesting note, it is possible to express any file in unary, that is using only a long (*really, really, really, really, really, really, really, really, really* long) string of ones. At first, this may seem obvious: Since it's binary, it's just a number, and can be written in unary by using that many 1's. However, files can start with zeros, and zeros prepended to a number don't change it. This can be solved by simply adding an extra "1" before any file initially, or by specifying the number of initial zeros using a self-delimiting character (see later)

B. Run Length Encoding (RLE)

Run length encoding is usually given as a simple example of compression, and though it is not used much on its own, many algorithms use it to save extra space. The main concept behind it is to shorten repetitions in data by replacing them with one copy of the repetitive chunk and one byte denoting how often it is to be repeated. A little-used character can be used to flag that the next two characters indicate repetition. If for some reason, the character itself is in the data to be compressed, we also need a way to use the next letter to show this, maybe by repeating the symbol. For example, if we choose the exclamation mark "!" to be the character, we could compress text as follows (spaces are for emphasis):

Compresssssing is funnnn! → Compre!6sing is fu!4n!!

Though it can be useful for repetitions of more than three characters, RLE is useless for runs of three and worse for 2 (since you use three bits to represent what could be encoded with two). There is some optimizing and improving that could still be added, but the concept remains the same.

C. Delta Encoding (Delta Modulation)

(Named after the Greek letter delta (Δ), often used to denote variance)

Sometimes, there may appear patterns of numbers that are practically unpredictable, but with adjacent terms close to each other, such as readings of temperature. What one could do is to record the first value, and from then on record the difference to the next. For example:

{23,27,25,24,21,19,22,22,24,27,26}→{23,+2,-2,-1,-3,-2,+3,+0,+2,+3,-1}

Since the increments are from zero to three, we only need two bits to store them. Note, however, that the increments can be either positive or negative. This can be handled by prepending a first bit in the representative byte for the sign of the number (+ or -) or using the concept of negabinary numbers, where all integers –positive, zero, and negative– can be represented as whole numbers.

This technique can be used in several innovative ways, such as in pictures by grouping adjacent bits, recording their average, and then their respective deviations from that number.

D. Incremental encoding

An analogue of delta encoding for ordered text, incremental coding shortens lists by specifying lengths to be repeated from the beginning of the last string. This is often useful in indexes that are ordered in a fairly alphabetical fashion, such as dictionaries.

E. Self-Delimiting Codes

Self-delimiting codes are methods of lossless data compression that represent frequent characters using shorter codes at the expense of lengthening less frequent ones. They must be designed so that it is possible to clearly distinguish adjacent characters. Taking off the zeros at the beginning of a character, for example, doesn't work, since there's no way to tell how many zeros were taken. This dilemma is similar to the one mentioned earlier. However, there are several safe methods, two of which are described in E2 and E3.

E1. Arbitrarily Sized Characters

It is worthy of mention that there are ways to ensure that a character can represent any given number, no matter how large it may be, (useful if the character would else have to be fit into too short a space). For example, an algorithm could be designed to ensure that there are never any two successive 1's in a character except at the end (see below). Then, any amount of zeros can be inserted in a predefined manner to represent a number. However, in most encoding schemes, this would be too tedious. In those cases, the location of the arbitrary number can be delimited, and the number then encoded in this manner. Another way, one that I have not yet come across in my research (though I have used it with much success), goes as follows: From 0 to 254, the numbers are simply represented using eight bits. For 255-510, 16 bits are used; 8 1's followed by the code for the difference between the number and 255. In essence, eight ones are added for each multiple of 255 in the character, along with eight bits for the remainder. This stores any character as several whole bytes, and is easily adapted to bytes of any length (See BLM, below).

E2. Fibonacci Coding

This method is a system based upon Fibonacci numbers. It can be defined by declaring that two 1's successively appear only at the end of a character, and then just constructing successive numbers as in binary (reversed), skipping those with double 1's:

1	11	8	000011
2	011	9	100011
3	0011	10	010011
4	1011	11	101011
5	00011	12	0000011
6	10011	13	0100011
7	01011	14	0010011

These can also be described more accurately using Fibonacci numbers; thus the name. This is also equivalent to writing a number in base Φ (The golden ratio), taking the integer part, reversing it, and appending a one.

Other than being capable of representing arbitrarily large numbers, Fibonacci coding has the unusual property of being capable of correcting itself if a bit is missing. If this is the case, the worst that could happen is that one of two consecutive ones is removed, or that two separate ones are put together, subtracting or adding a bit. A normal 8-bit code would be ruined by an accidental shift, since it throws off all the rest. This is very relevant if a single bit may be lost in unreliable transmission. However, while in normal coding a change in one bit will change only one character, in Fibonacci coding it may change a character, split a character into two, or merge two. If a certain number of characters are expected, this could be fatal.

E3. Huffman Coding

Fibonacci coding, unfortunately, is not quite "perfect" since it doesn't (for example) allow three consecutive ones in a single character. This means that it does not take advantage of all possible orders of digits. For this, we need something more optimal. This is where Huffman coding comes in. What it does is to assign each character a code that delimits itself, but in such a way that the resulting file is compressed maximally. It is often described in a probability tree, where the two least nodes are connected, and then assigned zero and one. This is continued, giving more frequent characters shorter branches, and thus shorter codes.

F. LZW

LZW is an algorithm that is the last in a succession of improvement, developed into its final state by Terry Welch in the 1980's. Its exact nature is a bit too complicated to describe (I don't quite completely comprehend it myself), but it fundamentally relies on the concept of taking advantage of repetitions of strings in source data. If a string is found that occurred earlier, it simply refers to the previous instance. In order to keep track of these, it uses a *dictionary* with 256 entries that changes depending on the text in the file to be compressed.

III. BLM: Experimentation

In doing this research, I completed an algorithm I had worked on as early as December of last year (2004). I refer to it as "Byte Length Minimizing," "BLM" for short. Though it is crude and inefficient, I am somewhat proud of it, and will describe its basic nature here for the sake of completeness.

Characters generally are bytes of length 8 (bits). What could be done is to simply represent the bytes using fewer bits for the more frequent ones, in a self-delimiting manner. Unfortunately, this requires the longer characters to have at least 8 more bits than the shorter ones, in a manner quite like the second method I described for recording arbitrarily sized characters. In order to try to compress the file, the algorithm simulates the results of all the possible compressions of the file after gathering frequencies, and applies the set that produced the shortest file in order to get the optimal output. If the file contains too many of the less frequent characters with respect to the more frequent ones, it results in output with repetitions that can be run length encoded using shorter bytes. The code for its *Mathematica* implementation can be found in Appendix B.

Comparing this algorithm to others, such as Huffman coding, no new features arise, although it uses a method capable of being applied to characters of any size, as in Fibonacci and Huffman Coding (in fact, most self-delimiting codes are of this manner). It does depend upon a prepended key that, if hidden and known to both the sender and the receiver, can be used for elementary encryption, though perhaps at a loss of some compression.

IV. Discussion

During this project, I discovered many interesting courses of potential research that each would have been just as interesting to study on their own. For example, complexity theory, the study of quantifying the "complicatedness" of things, is very applicable in data compression. Also, the ways in which an algorithm exploit the patterns in data brings up interesting questions: If an algorithm does so-and-so well on this data, how well does it perform on random data? How would you generate the random data? Is an algorithm better if it exploits more possibilities of digits; that is, is an algorithm more efficient if its output looks more random? Could that be used as a random-number generator? Would it be possible to encrypt the data while compressing? How can one take best advantage of asymmetric algorithms? How is it possible to get the best trade-off between compression and security (in redundancy and checks)? If it is impossible to create an algorithm that always compresses data, why is it possible to ensure that it will be at most only one bit longer? How do some algorithms do so well in practicality if in theory they are useless on arbitrary (random) input?

V. Conclusion

Having studied a broad, but somewhat superficial, range of concepts in the study of data compression for over two weeks, I now understand much of the trouble that people went through to save a tremendous amount of resources with zips and rars. Many files stored on computers use repetitive characters, and simple archiving saves a lot of space, as well as resources needed to store data in that space. There is so much more that I would like to mention, but for now, I must abide by the motto of moderation, and bid you farewell.

Appendix A: References

"Data Compression." *Wikipedia: The Free Encyclopedia*. 10 August 2005, 15:29. 2 Aug 2005 <http://en.wikipedia.org/wiki/Data_compression>.

The following articles on Wikipedia were also accessed on August 2, 2005, along with minimal reference from several other articles:

"Algorithmic information theory." <http://en.wikipedia.org/wiki/Algorithmic_information_theory>

"Arithmetic Coding." <http://en.wikipedia.org/wiki/Arithmetic_coding>

"Delta Encoding." <http://en.wikipedia.org/wiki/Delta_encoding>

"Fibonacci coding." <http://en.wikipedia.org/wiki/Fibonacci_coding>

"Fibonacci Number." <http://en.wikipedia.org/wiki/Fibonacci_number>

"Huffman coding." <http://en.wikipedia.org/wiki/Huffman_coding>

"Incremental encoding." <http://en.wikipedia.org/wiki/Incremental_encoding>

"Information theory." <http://en.wikipedia.org/wiki/Information_theory>

"Lossless data compression." <http://en.wikipedia.org/wiki/Lossless_data_compression>

"LZ77 (algorithm)." <http://en.wikipedia.org/wiki/LZ77>

"LZW." <http://en.wikipedia.org/wiki/LZW>

„Minimum description length." <http://en.wikipedia.org/wiki/Minimum_description_length>

"Run-length Encoding." <http://en.wikipedia.org/wiki/Run-length_encoding>

"Shannon-Fano coding." <http://en.wikipedia.org/wiki/Shannon-Fano_coding>

"Wikipedia:Citing Wikipedia" <http://en.wikipedia.org/wiki/Citing_Wikipedia>

"Historical Notes: History [of data compression]." [Excerpt from book; A New Kind of Science, by Stephen Wolfram] ©2002 Stephen Wolfram. Wolfram Science. 31 July 2005 <http://www.wolframscience.com/reference/notes/1069b>.

Lynch, Thomas D. Data Compression: Techniques and Applications. Belmont, California, Lifetime Learning Productions. ©1985

Solomon, David. Data Compression: The Complete Reference. New York, Springer Verlag. ©1998

Appendix B: *Mathematica* Code for BLM

```
BLMCompress[file_List]:=
Module[{file1,ab,CompressByte,n=Infinity,d},
file1=file;
ab=Sort[{Length[#]-1,First[#]}&/@Split[Sort[Join[file1,Range[0,255]]]]];
CompressByte[e_,m_,t_]:=Join[
    Table[1,{Min[t,Quotient[e,2^m-1]]*m}],
    If[e<(2^m-1)*t,
     IntegerDigits[Mod[e,2^m-1],2,m],
     IntegerDigits[e-(2^m-1)*t,2,8]]];
Do[(If[#<n,n=#;a={i, j},]&)[
    Total[Table[(If[#<j,i*(Floor[#]+1),i*j+8]&)[n/(2^i-1)]*ab[[256-
n,1]],{n,0,255}]]],
   {i,1,8},{j,Ceiling[256/(2^i-1)]}];
ab=Join[Reverse[Take[ab,-(2^a[[1]]-1)*a[[2]]]],Sort[Drop[ab,-(2^a[[1]]-
1)*a[[2]]]]];
d=Table[ab[[t,2]]->CompressByte[t-1,a[[1]],a[[2]]],{t,256}];
file1=Flatten[file1/.d];
Return[(FromDigits[#,2]&)/@Partition[Join[
      IntegerDigits[a[[1]]-1,2,3],
      IntegerDigits[a[[2]]-1,2,Floor[Log[2,Ceiling[256/(2^a[[1]]-1)]]]+1],
      IntegerDigits[Mod[1-(Length[file1])-Floor[Log[2,Ceiling[256/(2^a[[1]]-
1)]]],8],2,3],
      Join@@((IntegerDigits[#,2,8]&)/@Table[ab[[t, 2]],{t,1,(2^a[[1]]-
1)*a[[2]]}]),
      file1,
      Table[0,{Mod[1-(Length[file1])-Floor[Log[2,Ceiling[256/(2^a[[1]]-
1)]]],8]}]]
      ,8]]
]
```

```
BLMDeCompress[file_List]:=
 Module[{file1,f,g,decode,runs=0},
  file1=file;
  file1=Flatten[(IntegerDigits[#,2,8]&)/@file1];
  f=FromDigits[Take[file1,3],2]+1;
file1=Drop[file1,3];
g=FromDigits[Take[file1,Floor[Log[2,Ceiling[256/(2^f-1)]]]+1],2]+1;
file1=Drop[file1,Floor[Log[2,Ceiling[256/(2^f-1)]]]+1];
file1=Take[file1,{4,-1-FromDigits[Take[file1,3],2]}];
decode=(FromDigits[#,2]&)/@Partition[Take[file1,(2^f-1)*g*8],8];
decode=MapIndexed[(#2[[1]]-1->#&),Join[decode,Complement[Range[0,255],decode]]];
file1=Drop[file1,(2^f-1)*g*8];
file1=Reap[While[0<Length[file1],If[runs==g,
      Sow[FromDigits[Take[file1,8],2]+(2^f-1)*g];file1=Drop[file1,8];runs=0,
      If[
       Take[file1,f]==Table[1,{f}],
       runs++;file1=Drop[file1,f],
       Sow[(2^f-1)*runs+FromDigits[Take[file1,f],2]];file1=Drop[file1,f];runs=0]
      ]]][[2,1]];
Return[file1/.decode];
]
```