Loss-Less Compression:
It depends upon what you know

J. Christian Russ, ADE Optical Systems, Inc.

Abstract:

Loss-less compression is exactly what it sounds like. Data is compressed and later restored to its original form – modem transmission, disk and tape storage, and RAM compression all depend upon data being restored EXACTLY as it was before. But some forms of compression are better than others. In the last twenty years the field of data compression has changed significantly both because of better algorithms and faster computers. This paper surveys the current state of compression and examines some of the pitfalls.

RLE – Run Length Encoding

The first kind of compression that we all learn about is RLE or Run Length Encoding. It is performed by replacing strings of values that are identical (like a sequence of 0's) with two numbers – the value and the number of times the value appears.

For binary images, this is simplified — now there is no need to indicate whether the value is a 1 or a 0, rather the number of 1's is followed by the number of 0's, etc, until the end of the line (typically a 0, 0 combination). For some kinds of images (FAX) this works very well.

Frequency of English letters

In many cases, RLE is not good enough, especially when looking at the English language. There just aren't that many words that have sequences of the same letter repeating enough times to be worthwhile. Instead, we know some things about the frequency of the letters. (For an example of this, watch Wheel of Fortune – all of the players know the relative frequency of the letters and guess the most frequent letters first.)

As you would guess, the letter E is the most frequent in the alphabet, and the letters Q and Z are the least frequent. Because of this, if there were a way to use fewer bits to represent the E and more bits to represent the Q and Z, it would be worthwhile. More on this later with Huffman coding.

Infinite number of monkeys

It has been proposed that with an infinite number of monkeys, typewriters, and bananas you could eventually reproduce all of the great works of man. Part of the reason this takes so long is that a monkey doesn't know to hit E's more often than Z's. The monkey that has an equal probability for all of the letters is an Order Zero Monkey. He's not very bright.

If we had a slightly smarter breed of monkey that typed E's more frequently than the Z's it would take less time to reproduce Hamlet and the rest. This is an Order One Monkey.

Let us have a still smarter monkey – one that knew after a Q, most of the time the right letter to type is a U. This monkey knows the probability of a second letter after a first and is an Order Two Monkey.

As the monkeys come from higher orders (Three, Four, etc.), the time it takes for the great plays to be reproduced gets smaller, but the storage requirements for the tables gets bigger (26^2, 26^3, …, 26^n).

This process can be simulated by building probability tables of the letters of the alphabet from some source material and then using a random number generator to make letters (a Random Monkey). With an order one probability table the text is pretty poor. It is rare to get whole words that are meaningful and nearly impossible to get a whole sentence. (Our monkey needs 26 letters, a space, and some punctuation.)

As the order of the probability table increases the quality of the text that is produced by our Random Monkey improves markedly. By the time the order gets above four, the Random Monkey is reproducing entire strings of the original text, very much in the original style. If the original source material was Shakespeare then the text would look like very familiar indeed.

Thus, we have found a way to quantify knowledge of the English language in the form of a probability table. This knowledge can be used for compression.

Some Random Monkey

Order 0: Equal Probability for all of the letters

Order 1: Knows the probability of the letter 'E'

Order 2: Knows the probability of the letter 'U' after the letter 'Q'

Order 3: Knows the probability of the letter 'N' after the letters 'I' and 'O'.

Huffman coding - 1st order probabilities

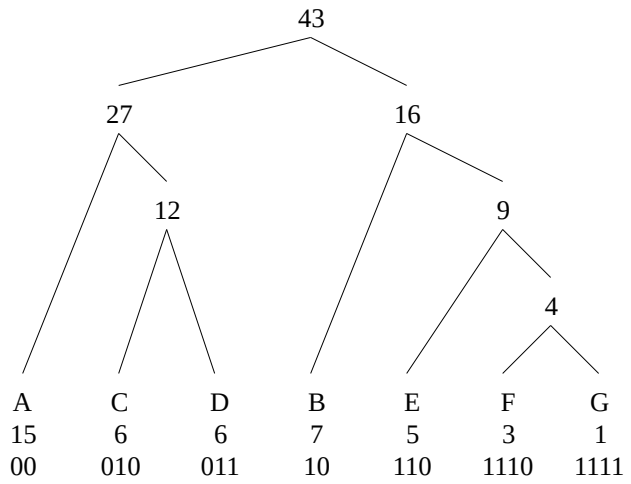
Huffman coding (1952 paper "A Method for the Construction of Minimum Redundancy Codes") creates variable-length codes based upon probabilities. These codes have an integral number of bits. For instance, if the letter 'E' occurred 95% of the time, it could be represented with just one bit '1'. Less frequent letters would take more bits, and of course the least frequent letters take the most number of bits – potentially many more than eight bits. The bit stream is decoded with a binary tree, and the algorithm is as follows:

1. Each of the letters is laid out as a leaf of a binary tree (and is added to a list of free nodes).
2. The two free nodes (letters or parent) with the lowest probabilities are located.
3. A parent node for these free nodes is created with a weight that is the sum of the two nodes.
4. The parent is then added to the list of free nodes, and the two children are removed from the list.
5. One of the children is assigned a 1. The other is assigned a 0.
6. Repeat this sequence from #2 until there is only one free node left (the root of the tree).

Therefore, the letters A thru G with the frequencies:

A	B	C	D	E	F	G
15	7	6	6	5	3	1

Would yield a tree that looks like this:



In this example, the 'A' is fairly frequent and only takes two bits to represent, but the 'G' is less frequent and takes four bits to represent. Since there are only seven letters, you could use three bits to represent everything.

There are two ways to build up statistics on a selection of text or other data. The first is to have a common probability table built up from every bit of data around that could be accessable and the second is to only look at the selection of data that is to be sent.

In the former case (static table) it is possible for the probability table to be located with both the compressor and the decompressor and therefore it need not be sent with each data set.

In the second case (dynamic table) the probability table needs to be sent with the data set and as the order of this table increases, the size increases exponentially.

Huffman coding is essentially a variable-length cypher, and is the best of such methods that produce integral-length results. The disadvantage, is that in this case, the best compression is 3:2 even if there are long repeating strings.

In various spy stories there is a reference to cyphers and One Use Pads. A one use pad is a Huffman tree where the leaves of the tree are long messages. The goal is to only use the tree once so that nobody can figure out what you said.

Arithmetic Coding, fractional bits

One of the problems with Huffman coding is that the smallest compression possible is to reduce a character to one bit. Thus, the greatest possible compression ratio (in the case of text with 6 bits) is 6:1. Or in the case of ASCII it is 8:1, if there is a really frequent letter (say an E).

What if it were possible to use fractional bits to compress? Well, it is possible. The method is called arithmetic coding.

If there was a letter that had a probability of 90%, the optimal size would be 0.15 bits – much smaller than the one bit that Huffman would assign to it.

The method is to assign a position between 0 and 1 along a probabilility line for each letter. The order is unimportant. The same sequence from before would yield:

A	B	C	D	E	F	G
.349	.512	.651	.791	.907	.977	1.0

So, in this case, the letter A is represented by the range 0 to 0.349, the letter B is represented by the range 0.350 to 0.512, and the letter G is represented by the range 0.978 to 1.0.

The general idea is that each long word in the compressed sequence is used as a shift register where the number corresponding to the letter is added and then the entire number is multiplied by the range that letter occupied. Decompression is similar to the Huffman method.

Adaptive Compression – Huffman

But everything doesn't stop here. It is usually impossible to get statistics on the entire stream before the data is sent, or the statistics may change in mid-stream, so the statistics for one part of the data is different than for the next part.

In this case, an adaptive method would be helpful – how do we change the probability tree as we go?

It would be possible to reserve one of the infrequent codes to correspond to a new tree being transmitted so that future data uses the new tree. But in most cases, only a little of the tree would change. Therefore you would just send the probabilities of the letters that changed position in the binary tree.

This has the advantage of being better than Huffman since the probabilities can be changed. The disadvantage is that some opcodes must be retained to send the tree, and the tree must be re-sent when the statistics change.

Dictionary Based Compression

There is another way. Longer sequences than just one letter can be turned into entries in the Huffman tree. Then the bit patterns of the entries can be sent, or more simply, the address of the entries can be sent. These are tokens within a dictionary. Words that are not known are composed of letters using the old Huffman methods, but the words that are known are easily compressed and sent.

This method is strongly related to a type of encryption that was used in World War I. Cypher books with five-letter combinations were produced and sent out one to a ship. The combinations translated to various words so that messages could be sent. If some kind of frequency ordering could be done with these combinations (or tokens) compression could be achieved.

(Unfortunately, if you use the cypher too many times, and the enemy knows what the message was about, the cypher can be broken.)

> The relationship between compression and encryption – "If they don't know how the data is compressed it is effectly encrypted."

Building a dictionary can be a bit of a problem, but computers are fast so there must be some way of doing it.

Static Dictionary

If the dictionary is static – computed way in advance – there is no need to transmit it along with the message. This can markedly improve compression. However, if the word you're trying to send is not in the dictionary, or the probabilities of the words in the table do not match those in the dataset being sent, then it is very far from optimal.

In general, a static dictionary is good for small datasets, where there is not enough data for good

statistics, but some assumptions can be made – whether the data is Text, Code, Pictures, etc.

Adaptive Dictionary

Adaptive dictionaries are constantly changing as the stream comes through. They must be computed on the fly (implying a moderate amount of searching) and must be sent along with the message stream.

But, if the statistics of the dataset are not known, it is vastly more efficient than a static dictionary. The other problem is how to make new entries in the dictionary – some entries may have to get disposed of in order to make space, so the additions to the dictionary get sent and it is maintained by both the compressor and decompressor.

Enter Ziv, Lempil, and Welch

In 1977 Jacob Ziv and Abraham Lempel published an article in IEEE Transactions on Information Theory. Its successor was published in 1978. These two papers "A Universal Algorithm for Sequential Data Compression" and "Compression of Individual Sequences via Variable-Rate Coding" are the core of the LZ77 and LZ78 algorithms. They are two very different techniques, but have dominated the industry since their introduction.

LZ77 uses a "sliding window." The dictionary is a set of fixed length phrases within the window. The window size is typically between 2K and 16K with phrase lengths typically up to 16 or 64 bytes.

LZ78 is completely different in how the dictionary is built up – it builds up phrases one symbol at a time when matches occur.

In both cases, as new elements get added to the dictionary, they are added to the data stream, so on the decoding side, the dictionary is also rebuilt.

In 1984, Terry Welch of Sperry Research Center (now owned by Unisys) published an implementation of the LZ78 algorithm. This algorithm is generally referred to LZW.

One of the big problems with compressed data is that a flipped bit can have highly disasterous effects. If the flipped bit occurs in a Huffman-encoded file, then the rest of the file can be damaged. In the dictionary methods, a flipped bit in the dictionary can cause very unpredictable results, including crashing the host computer if the code is not protected.

Because of this, there is a need to add a CRC (Cyclical Redundancy Coding) method to catch errors within the data. Repair is usually impossible unless a lot of CRC data is used, often undoing any advantage gained by compression.

Common implementations:

ARC (LZ78)

PKZIP (LZ77)

MNP-5 (Microcom – dynamic Huffman)

V.42*bis* (Dictionary scheme related to LZW)

QIC-122 (Stac's LZ77-based method)

GIF (LZW variant)

TimesTwo Driver Level Compression

TimesTwo was a SCSI driver that took every block of data that was written by the operating system, compressed it, and wrote it in whatever available empty spaces there were on the disk. (It used the Stac QIC-122 engine, although it could use any engine.) Underneath was a hash table that kept track of what portion of the disk was actually used, including starting addresses and lengths of the blocks that were there. In addition, especially with very small files, TimesTwo was very good at allocating just enough space for the data being written, rather than put a small file into a large allocation block, it could be as small as a 512-byte physical block.

Unfortunately, on the driver level it is very difficult to tell what portion of the disk is in use and what is not – merely that certain physical blocks are written together (and usually read together). Random access structures behaved very badly because compression works well on medium sized and large sized pieces of

data, not small things like allocation blocks. In addition, if a record within a file was changed, such as a resource file or a database, performance was considerably degraded. Fragmentation was another pitfall. It was also impossible to access the disk without the driver, since no other drivers knew about the compression on the disk., including disk recovery packages. A big problem, though, was that if you filled up the compressed volume, it was impossible to decompress it since the data was bigger than your disk – disk compression was a one way street.

Another sticky problem, that was nearly as big, was how to tell how much space is left on the disk? Let's say that there is 1MB of available physical blocks on the volume. Do we tell the user that there is 2MB free? What if he writes a file that is 1.9MB and it doesn't fit? We finally settled on a compromise – if there was 1MB left, we'd report 1:1 how much space was left, but if there was more than 1MB we'd report the doubled amount (with a fudge factor for a smooth transition). Also, there was a wild scavenger VBL task that searched through the VIB looking for allocation blocks that had just been deleted and could be released from the underlying hash table.

But, when writing something that was only going to be read and not modified, it behaved marvelously. The ideal use for something like TimesTwo would be in the mastering of CD's.

In the end, it was a race of two competing problems: 1) a bug was causing the hash table to keep getting corrupted, so files would just disappear (a **major** customer service nightmare), and 2) ultimately hard drive prices fell through the floor – why use compression and take the risk of trashing your disk performance (and in some cases your data) when a 1GB drive was under $1000 (and now under $200).

Other ideas included compressing floppies, but when a floppy costs $0.40 why try to compress the data on it?

Patents

Unisys (Sperry) has a patent on LZW. V.42*bis* is based upon LZW. Modem manufacturers may license it for a one time $25,000 fee. Having an industry standard based upon a patented algorithm is generally a bad idea, though.

Microsoft and Stac fought a major court battle over the compression in MSDOS 6.0. Stac is one of the few companies that has achieved a victory in dealing with Microsoft. Pyrrhic though it may be.

RAM Doubler

RAMDoubler gives your computer another method of having virtual memory. Essentially it provides virtual memory to compressed memory, rather than going to disk as would be the classic method. The compression in RAMDoubler depends upon three principles:

- A lot of memory isn't in use all at once – even within an application. Whenever a program does a _DisposeHandle or _DisposePtr that memory can be filled with 0's and compressed extremely well. (Unfortunately, Photoshop is a really bad counter example – it allocates all of the memory from its partition, fills it with stuff, and keeps it that way.)
- As long as the swapping that goes on (locate a block to swap out, compress it, swap in the target block and decompress) is faster than hard disk access it is better than classic virtual memory.
- If there is no space to swap out to, the hard disk is a good last resort.

As we saw above with TimesTwo, it is easier to compress moderate-sized pieces of data than small pieces. The block-size can be chosen carefully so that good compression statistics can be maintained.

Also, the swapping algorithm is not LRU (Least Recently Used), rather it should be a combination of LRU and MC (Most Compressible). If it doesn't compress well, it shouldn't be swapped out. There are some obvious problems with variable sized storage of the compressed pieces.

The future for compression

The next steps for compression go beyond adaptive dictionaries:

1) Combination Dictionaries: LZ77 & LZ78 methods build up a dictionary as they go along and transmit it, but there were advantages with the static dictionaries, too. Why not combine the two methods? If TimesTwo had a static dictionary as a starting point – based upon the four most common data types on the Mac (Code, Text, Resources, and Picts) – and then only transmitted the additions to the dictionary it would have operated much better on small blocks of data, which they ALL were. Also compression factors better than the typical 1.7:1 would have been achieved. (2.0:1 was claimed

and achieved by having a fragment manager under the driver, the extra

2) We've also ignored a different type of pattern in a dictionary where

Conclusion

Compression and encryption seem to be related in many ways. Encryption attempts to equalize and distribute the entropy in a message

You can encrypt data that has been compressed quite successfully.

You cannot compress data that has been encrypted. (Unless it has been

One additional trick with encryption is to add superfluous random data that will confuse decryption. This, of course, defeats the purpose of compression.

The goal of compression is to find as much redundant knowledge in a message and remove it in such a way as it can be put back later, either for storage or transmission. New methods tend to be discovered when the economics of compression are great – storage costs or transmission costs are high.

Loss-less compression knows fewer things about the content of the messages (than lossy) and assumes that there is some self-similar property that it can exploit. As increasing amounts of data are sent over low-speed telephone lines (28.8k Baud) with the World-Wide-Web, the demand for better compression will increase.