Data Compression

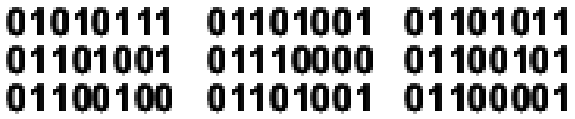
Contents

Chapter 1

Coding

1.1 Coding

Most people think that compression is mostly about coding. Well at one time it was. Before information theory, people spent years developing the perfect code to store data efficiently. To understand the limits of coding as a compression mechanism, we have to understand what coding is.



The ASCII binary translation of the word Wikipedia morphs into the English translation.

The computer really understands only one type of data, strings of 0s and 1s. You know 0 is off, 1 is on, and all that rot. These actually have to do with the state of a memory cell, where 0 is a low voltage output of the cell, and 1 is a slightly higher voltage. The actual voltage spread involved is getting smaller as the size of the circuits gets smaller and the voltage needed to jump the gap between parallel circuits gets smaller.

Hardware configuration used to define the size of a code, by limiting the number of bits as the two state memory element is called, that could be moved at a single pass. An example is the 4bit 8bit, 16bit 32bit, 64bit, and now 128bit architectures found during the development of microcomputers.

Now essentially everything in a computer is stored as sequences of bits. The storage capacity of the computer is limited by how many of those bits, are wasted. Ideally we don't want to waste bits, so we tend to use codes that minimize the number of bits that are wasted for any particular architecture of computer they will run on. The ultimate code is the code that covers the exact amount of variations that are possible in a data type, with the least amount of wasted bits.

It is interesting to find that this is not a consideration in the human brain, in fact we have had to define a term to explain the redundant information carried in the codes for the brain, and that term is degenerate coding, a pejorative term that really just means that there is more than one code for the same information.

Consider a string of numbers, that mean something. Are they a string of digits, in which case we want to code them to minimize the storage for each digit, An integer value, in which case we want to code them to minimize the storage for an arbitrarily sized integer, A floating point value, in which case we want to code them to minimize the storage for an arbitrarily sized mantissa, and include information on what power the number is raised to, and how many bits past the point we are going to store. Or an address string in which case we want to code it to minimize the number of bits it takes for a character in a specific language.

To a human all of these different types of storage look the same on the printed page. But it takes time to convert between the different types of storage, so you might not want to translate them into the tightest coding possible while you are using them. However later when you want to store them away for posterity, it looks like a good idea to compress them down to the smallest size possible. That is the essence of compression.

1.2 limits to coding density

So the limit to coding density, is determined by the type of data, you are trying to code and also by the amount of information that is available from a separate location.

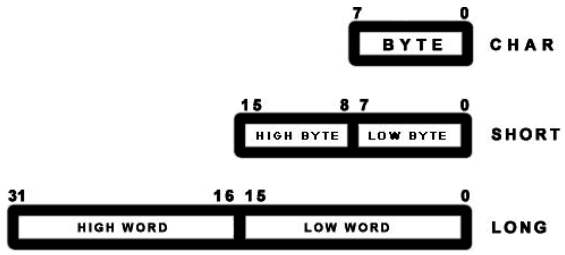
As an example, the military have the practice to limit radio chatter during a campaign, this is often considered good thing. Even taking it to the point of limiting themselves to a click on their mike, to trigger the next phase of the attack. To a person who is fully briefed on it, a click on a mike will be all that is needed to send a wealth of information about the state of the operation. The event will have no meaning to the enemy waiting in the trenches, so that the first evidence they have of attack is when guns start going off.

Ideally all storage could be achieved in one bit files that contained whole encyclopedias. In actuality that is not likely to happen. Coding is still important, but it is very limited in what it can now do for compression. New algorithms for compression are increasingly hard to come

by, hence the path of specialization on the source data new approaches take, true advances have recently been reduced to the hardware where the code will run.

1.3 bits, bytes, symbols, pixels, streams, files

In the field of data compression, it is convenient to measure the "size" of data in terms of the fundamental unit of data: bits.



3 basic common primitive types char,short int,long int.

Many data compression algorithms produce a compressed data stream that is a stream of bits -- with no particular alignment to any other size.

It is sometimes convenient to consider the input data in terms of "symbols". Some algorithms compress English text in terms of the symbols from an input and process them converting them in a reversible representation on the compressed output. Symbols are usually sets of bits but they in turn can represent any type of character representation/encoding, even pixels or waveforms.

When compressing CD-quality audio, the symbols are the 2^16 possible amplitude levels -- the 2^16 possible physical positions of the speaker cone.

Some file compression utilities consider the uncompressed file in terms of 257 different symbols. At any point in the file, the "next symbol" could be any one of 257 symbols: one of the 256 possible bytes -- or we could have already reached the end of the file, the 257th symbol indicating "end-of-file".

When comparing image compression routines, sometimes the term "bpp" (bits per pixel) is used. An uncompressed full-color bitmap image is 24 bpp. An uncompressed 2 color bitmap image (containing only black pixels and white pixels) is 1 bpp. Some image compression algorithms can compress some images to much less than 0.1 bpp. The same image compression algorithm may be doing pretty good to compress some other image to 7.5 bpp.

1.4 fixed-length coding

1.5 variable-length coding

With variable-length coding, we can make some symbols very short -- shorter than any fixed-length encoding of those symbols. This is great for compressing data.

However, variable-length coding almost always end up make other symbols slightly longer than best fixed-length encoding of those symbols.

Using more bits store a symbol sounds ridiculous at first -- Why don't we simply store a shorter fixed-length version of such symbols?

OK, yeah, *other* symbols could be shorter. So ... why can't we use the variable-length version when that would be shorter, and use the fixed-length version when that would be shorter? The best of both worlds? Perhaps use a "1" prefix to indicate "the fixed-length representation follows" ?

We discuss variable-length coding in more detail in the entropy coding section of this book.

1.6 Representing integers

Digital information can be encoded as a series of integers. With modern digital electronics, we typically first (1) encode the data (etc.) in some relatively simple (but verbose) encoding as a series of integers, then (2 & 3) data compression algorithms take that series of integers, and find some other way to represent the entire series in fewer bits.

For example, when we store music in a MP3 file, the machine often first (1) digitizes the analog sound with a microphone and an ADC as a (relatively simple, but verbose) 16-bit PCM encoding at 44.1 kHz sampling rate per channel (Compact Disc digital audio, uncompressed WAV audio, etc.). Then the software processes that raw data first to (2) transform it into a representation that requires fewer integers, then (3) represents each of those integers as bits in a way that the entire song can be stored in relatively few bits.

(In the early days of computers, people spent a lot of time figuring out how to encode music (etc.) into a reasonable number of keystrokes, both compressing and encoding the data "manually" with human brainpower before entering it into the computer).

People have invented a surprisingly large number of ways of representing integers as bits.

... (say a few words here about 32-bit unsigned integers) ...

... (say a few words here about 2's complement signed integers) ...

Floating-point numbers are often stored as pairs of signed integers -- the significand (the significant digits) and the

exponent.

1.6.1 ZigZag encoding

ZigZag encoding is one way to represent signed integers. It maps "small" signed integers (close to zero) to "small" unsigned integers (the most-significant bits are all zero).[1][2] It's designed to have a one-to-one mapping between 32-bit ZigZag-encoded signed integers and 32-bit 2's complement signed integers. ZigZag encoding (like delta encoding) doesn't save any space by itself, but it often transforms data in such a way that other downstream compression algorithms work better. (Delta encoding followed by ZigZag encoding often produces better results -- with the same downstream compression algorithm -- than either one alone).[3]

encoded integer 0000 0005 : −3 0000 0003 : −2 0000 0001 : −1 0000 0000 : 0 0000 0002 : +1 0000 0004 : +2 0000 0006 : +3

[1] Google Developers. "Protocol Buffers :Encoding".

[2] "protobuf-c.c"

[3] "Delta zigzag encoding binary packing"

Chapter 2

Evaluating Compression Effectiveness

When an application programmer is deciding which compression algorithm to use, there are a number of factors that must be balanced: [1]

- Speed: how fast is it?
- compression ratio: There's really no point if it doesn't make our data smaller than the uncompressed size.
- complexity: How big is the executable? (This has both hard costs -- how much ROM or disk space does the software use? -- as well as soft costs -- if I need to change it, how long does it take to go through all the source code to find the right thing to change, and once changed, how long does it take to test every part of the code to make sure the change doesn't break something else?)
- space: how much RAM does it need to run?
- latency: how long do I have to wait before I hear the first note of the song?
- interoperability: Will the files I generate be readable by any standard archive utility?

When a compression library programmer has tweaked a compression algorithm and trying to decide if it is really better or if he should revert to the previous version, he uses the same criteria.

2.1 Speed

When evaluating data compression algorithms, speed is always in terms of uncompressed data handled per second.

$$\text{speed} = \frac{\text{Uncompressed bits}}{\text{seconds to process}}$$

For streaming audio and video,

> speed = number of uncompressed bits that can be handled in one second.

Some applications use data compression techniques even when they have so much RAM and disk space that there's no real need to make files smaller. File compression and delta compression are often used to speed up copying files from one end of a slow connection to another. Even on a single computer, some kinds of operations are significantly faster when performed on compressed versions of data rather than directly on the uncompressed data.[2] In particular, some compressed file formats are designed so that compressed pattern matching -- searching for a phrase in a compressed version of a text file -- is significantly faster than searching for that same phrase in the original uncompressed text file.[3][4][5][6][7][8][9][10][11]

(Is "zgrep" or "zcat" relevant here?)

Some compression algorithms specifically designed to compress bitmaps -- the byte-aligned bitmap code (BBC), the word-aligned hybrid code (WAH), the position list word aligned hybrid (PLWAH), the compressed adaptive index (COMPAX), and the compressed 'n' composable integer set (CONCISE) -- allow bitwise operations to be directly applied to compressed data.[12] This can be much faster than decompressing the data, applying the bitwise operation, and then re-compressing the result.

(FIXME: move some of the above to compressed domain processing.)

2.1.1 decompression speed

$$\text{speed} = \frac{\text{Uncompressed bits}}{\text{seconds to decompress}}$$

In many applications, the decompression speed is critical. If a particular implementation of an audio decompressor running on a prototype portable music player hardware cannot sustain 1.4 Mbit/s to the headphones, then it is unusable. No one will buy it unless you switch to a different implementation or faster hardware (or both) that can keep up with standard stereo audio speeds.

2.1.2 compression speed

$$\text{speed} = \frac{\text{Uncompressed bits}}{\text{seconds to compress}}$$

In a few applications, the compression speed is critical. If a particular implementation of an audio compressor running on a prototype voice recorder cannot sustain 7 bits/sample/channel x 1 channel x 8 kSamples/s = 56 kbit/s from the microphones to storage, then it is unusable. No one wants their recorded voice to have silent gaps where the hardware could not keep up. No one will buy it unless you switch to a different implementation or faster hardware (or both) that can keep up with standard telephone-quality voice speeds.

2.1.3 decompression speed and compression speed

The speed varies widely from one machine to another, from one implementation to another. Even on the same machine and same benchmark file and same implementation source code, using a different compiler may make a decompressor run faster.

The speed of a compressor is almost always slower than the speed of its corresponding decompressor.

Even with a fast modern CPU, compressed filesystem performance is often limited by the speed of the compression algorithm. Many modern embedded systems -- as well as many of the early computers that data compression algorithms were first developed on -- are heavily constrained by speed. There are a limited number of compression algorithms that are fast enough to be usable on extremely speed-constrained systems: [1]

- RLE;
- algorithms in the LZSS family;
- algorithms in the LZW family;
- fixed byte pair encoding such as PalmDoc;
- delta encoding + adaptive Huffman.

... *any others?* ...

2.2 space

Many modern embedded systems -- as well as many of the early computers that data compression algorithms were first developed on -- are RAM-limited. When available RAM is so small that there is not enough room for both decompressed text and also a separate dictionary -- such as the 12 KByte dictionary needed by a GIF decoder for the LZW dictionary -- very few compression algorithms work under those constraints. [13] With so little RAM,

- If we need the entire decompressed text in RAM -- such a self-extracting executable -- then we are forced to rule out LZW and use the decompressed text as the dictionary, like LZ77 and Pucrunch
- If we don't need the entire decompressed text in RAM -- such as external modems decompressing data sent over "slow" telephone links before forwarding the plaintext to a PC -- then for any given amount of RAM, LZW-like compressed formats usually give better compression than LZ77-like compressed formats. For typical English text, the more words and phrases you store in the decompressor's dictionary, the better the compression. LZ77 uses a lot of RAM to hold common words that repeat frequently. LZW uses the available RAM to store each unique word exactly once, and so gives better compression than LZ77 when RAM is limited.

When designing the compressed file format, there is typically a speed/space tradeoff between variable-length formats and byte-aligned formats. Most systems can handle byte-aligned formats much faster. The variable-length formats typically give better compression. The byte-aligned formats can and often do use data sizes other than 8 bit data. [13] For example, many LZ77-like decompressors use byte-aligned formats with 16-bit "items" that the decompressor breaks into a 3 bit length and a 13 bit offset. Some decompressors use a mix of 1-bit, 8-bit, and 16-bit items, where (for speed) the 1 bit items are carefully packaged into 8-bit "control bytes" so everything else can stay byte-aligned. (Later in the book we discuss byte-aligned formats in more detail: Data Compression/Dictionary compression#Implementation tips and tricks).

2.3 Evaluating Compression Ratio

In this book, we define the compression ratio as

$$\text{Compression Ratio} = \frac{\text{Uncompressed Size}}{\text{Compressed Size}}$$

A algorithm that can take a 2 MB compressed file and decompress it to a 10 MB file has a compression ratio of 10/2 = 5, sometimes written 5:1 (pronounced "five to one").

For streaming audio and video, the compression ratio is defined in terms of uncompressed and compressed bit rates instead of data sizes:

$$\text{Compression Ratio} = \frac{\text{Uncompressed Data Rate}}{\text{Compressed Data Rate}}$$

For example, songs on a CD are uncompressed with a data rate of 16 bits/sample/channel x 2 channels x 44.1 kSamples/s = 1.4 Mbit/s. That same song encoded at (lossy "high quality") 128 kbit/s Vorbis stream (or 128 kbit/s MP3 stream or a 128 kbit/s AAC file) yields a compression ratio of about 11:1 ("eleven to one").

That same song encoded with a typical lossless audio compressor such as FLAC or WavPack typically gives a

compression radio of about 2:1 to 3:1 ("three to one"), although a few songs give no compression (1:1) and some kinds of classical music give better than 3:1 compression with such a lossless audio compressor.

Using this definition of "compression ratio", for a given uncompressed file, a higher compression ratio results in a smaller compressed file.

(Unfortunately, a few other texts define "compression ratio" as the inverse, or with arbitrary other factors of 8 or 100 inserted, or some other formula entirely).

Some typical compression ratios for lossless compression of text files:

- 2:1 or worse: Extremely simple and fast algorithms such as LZRW1-a that can saturate a 100 MB/s hard drive with compressed data on a 2GHz CPU.
- 2.5:1 to 3.5:1 compression on text files: Slightly more sophisticated (and therefore slightly slower) algorithm such as DEFLATE[14]
- 5:1 Unfortunately, all known algorithms for getting a compression factor better than 5:1 on English language text all run extremely slowly, 5 or more hours to decompress 100 MB of uncompressed data on a 2GHz P4.
- 6.27:1 The current (2009) Hutter prize world record compression

All lossless data compression algorithms give different data compression ratios for different files. For almost any data compression algorithm, it is easy to artificially construct a "benchmarketing" file that can be compressed at amazingly high compression ratio and decompressed losslessly. Unfortunately, such artificially high compression ratios tell us nothing about how well those algorithms work on real data. A variety of standard benchmark files are available. Using a large collection of such benchmark files helps a programmer avoid accidentally over-tuning an algorithm so much that, while it works great on one particular file, it is horrible for other files.

Some standard benchmark files are listed later in this book -- Data Compression/References#Benchmark files.

2.3.1 generic compression

Some programmers focus on "generic compression" algorithms that are not tied to any particular format, such as text or images.[15][16] These programmers tune their algorithms on a collection of benchmark files that include a variety of formats.

2.3.2 format-specific compression

Other programmers focus on one specific kind of file -- video compression, still image compression, single-human speech compression, high-fidelity music compression, English text compression, or some other specific kind of file. Often these format-specific programmers try to find some kind of redundancies in the raw data that can be exploited for lossless compression -- for example, music often has one dominant tone that repeats over and over at one specific frequency for a few tenths of a second -- but each repeat is never quite exactly the same as any of the others -- then a similar dominant tone that repeats over and over at some other frequency for a few more tenths of a second. Often these format-specific programmers try to find limits to human perception that can be exploited for lossy compression. Often these programmers come up with ways to "transform" or "preprocess" or "de-correlate" the data, and then hand the intermediate results off to some "generic compression" algorithm. We discuss this more at Data Compression/Multiple transformations.

For the particular format it was tuned for, such format-specific compression algorithms generally give much better results than a generic compression algorithm alone. Alas, such algorithms generally give worse results than a generic compression algorithm for other kinds of files.

2.4 Latency

Latency refers to a short period of delay (usually measured in milliseconds) between when an audio signal enters and when it emerges from a system.

Compression adds 2 kinds of latency: compression latency and decompression latency, both of which add to end-to-end latency.

In some audio applications, in particular 2-way telephone-like communication, end-to-end latency is critical. The recommended maximum time delay for telephone service is 150 milliseconds[*citation needed*](Wikipedia:1 E-1 s). This rules out many popular compression algorithms. For example, a standard Huffman compressed block size of 150 milliseconds or longer won't work. Standard Huffman compression requires analyzing an entire block before sending out the compressed prefix code for the first symbol in the block. In this case the Huffman compressor uses up all the time allowed waiting for the block to fill up, leaving no time for time-of-flight transmission or decoding. (A block size of 150 milliseconds or longer may work fine with an adaptive decoder).

In other audio applications, end-to-end latency is irrelevant. When compressing a song for later distribution, or the soundtrack of a movie, the compressor typically has the whole thing available to it before it begins compressing. Such applications may use low-latency algorithms when they are adequate; but they also allow other algorithms to be used that may give better net compression or a lower peak bit rate.

In a few applications, only decompression latency is important. For example, if a particular implementation of an audio decompressor running on a prototype portable music player hardware has a latency of 10 minutes, then it is almost unusable. No one wants to wait 10 minutes after selecting a song before starting to hear it. No one will buy it unless you switch to a different implementation or faster hardware (or both).

Many compression algorithms have a minimum information-theoretic latency, measured in bits. *(Is there a better name for what this paragraph and the next discusses than "information-theoretic latency"?)* Given a constant uncompressed bitrate, this corresponds to the worst-case delay between when a (uncompressed) bit goes into the compressor, to the time the corresponding (uncompressed) bit comes out the decompressor, in situations where the bitrate is so slow that we can neglect the time required for the computations done in the compressor and the decompressor and the time-of-flight.

A fixed prefix coder or a adaptive Huffman coder typically has an extremely short information-theoretic latency. Many of them have latencies of less than 16 bits.

MP3 compressors sacrifice latency and quality to gain much higher compression ratios. They have a latency of at least 576 time-domain 16-bit samples at 44.1 kHz, giving a latency of at least 9,216 bits or 13 milliseconds, often longer to take advantage of the "byte reservoir".

2.5 energy

There has been little research done on the amount of energy used by compression algorithms.

In some sensor networks, the purpose of compression is to save energy. By spending a little energy in the CPU compressing the data, so we have fewer bytes to transmit, we save energy in the radio -- the radio can be turned on less often, or for shorter periods of time, or both.[17]

The best compression algorithms for such sensor networks are the ones that minimize the total energy, and so maximize the runtime, the length of time between battery replacements. Such algorithms are in a niche between, on one side, algorithms that produce smaller compressed files, but use up too much CPU energy to produce them; and on the other side, algorithms that use less CPU energy, but produce (relatively) longer files that take far more energy to transmit.

2.6 Comparisons

We use the game-theory term "dominates" to indicate that one algorithm is faster, gives smaller compressed files, and has lower latency than some other algorithm. Of course, some particular implementations are not very well optimized, and so they could be tweaked to run *much* faster and still implement the same algorithm with the same compressed file format. But any abstract algorithm necessarily requires some minimum number of operations, and so it is unlikely that an algorithm that requires a couple orders of magnitude more operations than some currently-faster algorithm can ever be optimized enough to dominate that faster algorithm.

Many historically important compression algorithms are now obsolete, having been dominated by some other, more useful algorithm. But currently (and for the foreseeable future) there is no one "best" compression algorithm even for a fixed set of benchmark files -- there is a spectrum of many "best" algorithms along the Pareto frontier; that spectrum of algorithms together dominates and makes obsolete all other known algorithms. Blazingly fast algorithms that give some but not a lot of compression sit at one end of the Pareto frontier. At the far end of the Pareto frontier sit the most compact known ways to compress benchmark files -- but, alas, running so slowly that they are not very useful.

2.7 Further reading

[1] "Hacking Data Compression" by Andy McFadden 1993

[2] "A Survey of Compressed Domain Processing Techniques" by Brian C. Smith

[3] "SASE: Implementation of a Compressed Text Search Engine" (1997) by Srinidhi Varadarajan And , Srinidhi Varadarajan , Srinidhi Varadarajan , Tzi-cker Chiueh , Tzi-cker Chiueh

[4] Shmuel T. Klein and Miri Kopel Ben-Nissan. "On the Usefulness of Fibonacci Compression Codes". 2004. 2009.

[5] "A Text Compression Scheme That Allows Fast Searching Directly In The Compressed File" 1993 by Udi Manber

[6] "Phrase-based pattern matching in compressed text" by J. Shane Culpepper and Alistair Moffat

[7] Wikipedia: Compressed suffix array

[8] Shmuel T. Klein and Dana Shapira. "Searching in Compressed Dictionaries".

[9] Carlos Avendaño Pérez, and Claudia Feregrino Uribe. "Approximate Searching on Compressed Text".

[10] Udi Manber. "A text compression scheme that allows fast searching directly in the compressed file".

[11] Yusuke Shibata , Takuya Kida , Shuichi Fukamachi , Masayuki Takeda , Ayumi Shinohara , Takeshi Shinohara , Setsuo Arikawa. "Byte Pair Encoding: A Text Compression Scheme That Accelerates Pattern Matching" 1999. "pattern matching in BPE compressed text is even faster than matching in the original text."

[12] Wikipedia: bitmap index: compression. *FIXME: replace this reference with better reference(s).*

[13] "Pucrunch: An Optimizing Hybrid LZ77 RLE Data Compression Program, aka Improving Compression Ratio for Low-Resource Decompression" by Pasi Ojala

[14] "Unzipping the GZIP compression protocol" by Joe Rash

[15] Matt Mahoney. "Generic Compression Benchmark". Further discussion: "benchmark for generic compression".

[16] IETF. "6LoWPAN Generic Compression of Headers and Header-like Payloads"

[17] Christopher M. Sadler and Margaret Martonosi "Data Compression Algorithms for Energy-Constrained Devices in Delay Tolerant Networks"

Chapter 3

Models

3.1 Models in Compression

After Information Theory showed how to build a perfect code for any specific symbolic system, people found that it was relatively easy to come up with new coding conventions for new types of data, and the digital revolution was on. At one point it was wondered just what couldn't be stored in a computer.

It was about this time, that scientists started looking for more extraordinary ways to reduce storage costs. One scientist hit on a solution. He said, if we model our data, we will find that certain patterns develop in the data. If we can code the longest and most prevalent patterns with the shortest codes and the shorter patterns with less prevalence with longer codes, we can reduce the size of a file without losing any data.

Unfortunately what he found was that the model needed to optimize the length of a file in one type of data was not the same as the model needed to optimize the length of a file in another type.

Another scientist noticed that when he looked at the binary codes of his data, there were strings of repetitive patterns. Why not, he said, abstract out the patterns, and simply count how many of them ran in a row and code the file so that the code efficiently handled statistically significant numbers of repetitions. So for instance consider a letter on a paper. Using this technique all white space that consisted of blanks would be reduced to a single space, and a number to indicate how many were needed to fill out the line.

In each of these cases, the scientist modelled their data, and used something they recognized in their model to significantly reduce storage. The model was what gave them the leverage to reduce the storage size without losing information.

Now a cautionary note, I once worked with a man that had great ideas about how to encode data. He wanted to develop a compression scheme that could be re-entered and compress even further. But his chosen technique didn't include a model of how encoding the data changed the data. He is still looking for a model today.

Chapter 4

Markov models

4.1 Markov algorithms

Markov algorithms are data compression algorithms that have the Markov assumption: the plaintext was produced by something that can be approximated by a Markov model -- i.e., a Markov chain or hidden Markov model or variable-order Markov model or something similar.

Dictionary algorithms can always be emulated by Markov algorithms, but not the converse. [1]

Fully general Markov models are used in some of the most complicated algorithms with the best compression ratio known -- alas, they are also some of the slowest algorithms in use.

4.1.1 PPP Predictor Compression Protocol

Perhaps the simplest possible Markov algorithm is the PPP Predictor Compression Protocol, as implemented by Timo Raita and standardized by RFC 1978.[2]

It looks at (approximately) the 3 or 4 most-recently-decoded bytes, finds the last time that sequence occurred, and guesses that the following byte will be the same following byte as last time.

The inner loop of the Predictor decompressor is something like this:

- The Predictor decoder reads a flag byte, containing 8 flags. Each flag bit corresponds to one byte of reconstructed decompressed data.
- For each of the 8 flag bits:
    - If the flag bit is 0, then the guess was wrong. The Predictor decoder reads a literal byte from the compressed data, and pass it straight through to the decompressed text. Also store that literal byte in the GuessTable[context], so perhaps the next time this happens we will guess right.
    - If the flag bit is 1, then the guess was right. Read the guess from GuessTable[context] and write it to the decompressed text.
    - Update the context from the byte we just wrote out to the decompressed text.
- Repeat until there are no more compressed bytes to read.

In any given context, one byte value is the most-likely next-byte. The Predictor compresses that most-likely next-byte to 1 bit. When any other byte occurs, the Predictor expands it to 9 bits. The Predictor can get pretty good compression even when it guesses wrong half the time.

4.1.2 1st order Markov

The 1st order Markov decompressor algorithm is a little more sophisticated than the Predictor algorithm.

After decompressing some byte, a 1st order Markov decompressor looks at every time that byte has ever occurred before.[3] In any given 1-byte context, one byte value is the most-likely next-byte; some other byte value is the next-most-likely next-byte ... and so on; typically there are a few bytes that have only occurred once each in this context -- those are the next-least-likely next-byte values; and typically there are a few bytes that have never occurred in this context -- those are the least-likely next-byte values.

The 1st order Markov decompressor uses some entropy coder to assign each possible next-byte value a unique prefix codeword, such that the more-likely bytes are given a short codeword, the less-likely bytes are given a longer codeword (perhaps 8 bits), and the least-likely bytes are given a long codeword (longer than 8 bits).

In some cases, where the next-byte is overwhelmingly likely to be one particular byte (for example, the byte following 'q' is almost always 'u' in English text), 1st order Markov compresses those cases just as well as the Predictor algorithm. In other cases, Markov compresses significantly better than the Predictor algorithm, because it can compress 'likely' bytes even when they are not the one most-likely byte.

Higher-order letter-level and word-level Markov chains are better at predicting “likely” English phrases. Some

people create "parody generators" out of Markov decoders by allowing them to build a Markov chain, but instead of feeding the decoder the particular compressed codewords that give the Markov decoder enough clues to reconstruct some particular plaintext, they feed the Markov decoder randomly-generated bits.[4]

4.2 model mixing

The decompression algorithms with the best compression typically use several different models, each one giving predictions of the next bit or byte, and somehow mix those predictions together to get a final prediction to feed into the entropy coder. As long as the final prediction is not zero or one (never-happens or always-happens), the decompressor can decode all files without error (see The zero-frequency problem for details).

Alas, so far we do not yet know the "best" way to combine such predictions. There are several ad-hoc ways that people have developed to mix predictions together. Some mixing techniques appear to be "better", in that they produce smaller compressed files than other mixing techniques (and better than using any one of the underlying predictions alone).

Before we get to the actual "context mixing" algorithms, let's look at PPM and DMC, which directly use the predictions from one single context or another (rather than mixing predictions):

4.2.1 PPM

Prediction by partial matching[5] is perhaps the simplest to understand mixing technique. PPM maintains Markov models of several different orders. A PPM(5) decoder uses a 5th order Markov model whenever possible. If no prediction can be made based on all 5 context symbols -- i.e., the 5 most-recently decoded bytes have never occurred in that order before -- then it falls back on a 4th order Markov model. If the 4th order model cannot make a prediction -- i.e., the 4 most-recently-decoded bytes have never occurred in that order before -- then it falls back on a 3rd order Markov model. The PPM decoder will eventually find some Markov model it can use to compress the text, even if it has to fall all the way back to the default zero order Markov model.

4.2.2 Dynamic Markov compression

Dynamic Markov compression was developed by Gordon Cormack and Nigel Horspool.[6] Gordon Cormack published his DMC implementation in C in 1993, under a non-commercial license.[7] Dynamic Markov compression is a lossless data compression algorithm very similar to PPM, except it predicts one bit at a time, rather than predicting a byte at a time. This makes it slower but gives slightly better compression. It is used as a model or submodel in several highly experimental implementations.

The predictions are fed into an arithmetic coder. (Because the predictions are made one bit at a time, simple Huffman coding rather than arithmetic coding would give no compression).

4.3 model mixing

Some model mixing / context mixing approaches include:

Context tree weighting[8] is another mixing technique.

4.4 Further reading

[1] Ross Williams. "LZRW4: ZIV AND LEMPEL MEET MARKOV". 1991.

[2] RFC 1978

[3] Rather than literally scanning the entire plaintext-so-far for each byte it decompresses, most programmers use a much faster approach that gives the same results: the program updates an internal hash table or 2D array for each byte it decompresses, then directly fetches the data it needs for the next byte from that data structure.

[4] Jon Bentley. "Generating Text" (Section 15.3 of Programming Pearls) 2000.

[5] Wikipedia: Prediction by partial matching

[6] Wikipedia: Dynamic Markov compression

[7] Gordon V. Cormack. "Dynamic Markov Compression (DMC) Version 0.0.0". 1993. dmc.c

[8] Wikipedia: Context tree weighting

Chapter 5

The zero-frequency problem

One of the biggest problems of inferring a model is the zero frequency problem.[1]

While decompressing a file, an **adaptive Huffman decoder** keeps changing its estimate of which letters are likely and which are unlikely according to the letters seen so far in the plaintext. It predicts future letter probabilities based on counting how often each letter has already occurred. The decoder will then decode the next letter using its current table of codewords, which assigns a short codeword to each letter that it predicts is highly likely, and a long codeword to each letter that it predicts is possible but unlikely.

For example, say a text file begins with an extended quotation from “Gadsby” (1939). The simplest approach -- the frequency so far of 'e' is zero, so I'm going to assume that it will continue to be zero -- will not work.

If a Huffman decoder ever gets it in its head that some byte has zero frequency, it is not possible to convince that decoder to emit that byte -- not now, not ever. That byte is no longer part of its dictionary. And so it will not be possible for the decoder to correctly emit plaintext that eventually does use the letter 'e'.

So we must give the Huffman decoder some non-zero estimate of the future frequency of 'e', if we are to have any hope of correctly decoding plaintext that begins with a quotation from “Gadsby”, then later includes the letter 'e'. Even an inaccurate estimate -- as long as it’s not exactly zero -- allows the decoder to correctly emit plaintext without error. An accurate estimate will produce a shorter compressed file than an inaccurate estimate. Alas, it is difficult to accurately estimate the frequency of something that has never occurred yet.[2]

Context-mixing adaptive decompressors have a similar problem. To get good compression, the decompressor needs to accurately predict the probability that the next bit is a 1, P(x). After decompressing enough text, the decompressor can see that event A has occurred Ax times, and the next bit was a one in A1 of those times. So in the absence of other information, when A occurs again, it estimates that the next bit is a one with probability P(x|A) = A1/Ax. If both A and B have occurred simultaneously many times ABx before (and the next bit was a 1 in AB1 of those cases), then when A and B occur simultaneously again, our best estimate of P(x|A,B) -- our estimate that the next bit is a one -- is AB1/ABx, completely ignoring A1 and Ax and B1 and Bx.

However, the first time A and B occur simultaneously, we can't do that -- we can't calculate 0/0. All we can do is guess that P(x|A,B) is either the same as P(x|A), or the same as P(x|B), or somehow combine the predictions, possibly using a heuristic such as the Lincoln index that takes into account how many times each event occurred. Given the event counts A1 and Ax and B1 and Bx, what is a good estimate of P(x|A,B)? Alas,

“Probability theory doesn't help us, even if we assume A and B are independent.”

5.1 References

[1] Zero frequency problem and escape codes by Arturo San Emeterio Campos. 1999.

[2] Robert W. Lucky. “Black Swans: What if Gaussian engineering is clear, simple, and wrong? 2010.

Chapter 6

data differencing

6.1 data differencing

Data compression can be viewed as a special case of data differencing.[1] Just as data compression involves 2 processes that often run on 2 different machines -- the “data compressor” and the “data expander” -- data differencing involves 2 processes that often run on 2 different machines -- the “differ” and the “patcher”, respectively.

Some algorithms and programs used in data differencing are very similar to algorithms used in “small-window compression” (aka traditional data compression) -- such as “large-window compression” (aka “data folding”[2]) -- while others are very different -- such as “data deduplication”[3][4] and the rsync algorithm.

A few algorithms used in traditional data compression of large files are similar to data differencing algorithms -- such as some techniques for storing many Huffman frequency tables described in Huffman: storing the frequency table.

Some algorithms and programs developed for data differencing include:

- diff ("w: delta compression")
- rsync
- rdiff
- lzip
- rzip
- IZO (Information Zipping Optimizer)[2]
- xdelta[5]
- bsdiff 4 and bspatch[6]
- bsdiff 6 and Percival’s universal delta compressor[6][7]

With other forms of compression, more information generally gives better compression. However, experiments by Factor, Sheinwald, and Yassour 2001 seem to indicate that, when using data differencing with LZ77-like compression, trying to use a large collection of shared files that resemble the target file often gives worse compression than using a single reference file from that collection.[8]

6.1.1 pre-loaded dictionary

When compressing very large files, it usually doesn't make much difference exactly what the dictionary is preloaded to, so many systems start with an empty dictionary.

However, pre-loading the dictionary can produce significantly smaller files in two cases:

- when compressing lots of short strings, or
- when data differencing -- compressing some big file that is almost the same as some file the receiver already has.

Some implementations of data compression algorithms already have support for pre-loading the dictionary.

- The popular zlib compression library supports deflateSetDictionary() and inflateSetDictionary() functions for pre-loading the dictionary.[9]

One possible approach to data differencing involves tweaking some compression algorithm to “pre-load” its internal data structures (it’s “window” or “dictionary”) with data from the files we already have, to allow the compression algorithm to use that data to (hopefully) give better compression on the next file.[10][11]

For example, the IPSW algorithm (the in-place sliding window of Shapira and Storer) initializes the source sliding window of a LZ77-like decompressor to some shared file S,[8] allowing common substrings of the target file T to be copied out of S, as well as the (like other LZ77-like compressors) allowing repeated substrings to be copied from previously-decompressed parts of the target file T, or if all else fails, from literal values in the compressed file.

Sometimes such “pre-loading” implementations use unmodified “small-window compression” algorithm, as in

US Patent RE41152 2010. Many early compression algorithms pre-loaded the dictionary with a "static dictionary" (Huffword, PalmDoc, etc.). The LZW algorithm gives better compression than the very similar LZ78 algorithm. One way of thinking about LZW is to imagine that the 256 literal byte values are not a separate "special case", but are, in effect, pre-loaded into the dictionary; while the LZ78 algorithm, in effect, starts with an empty dictionary, and so gives worse compression.

With many compression algorithms, such pre-loading is limited by a "window" or "history limit" -- when the window or limit is, say, 32 kByte, it doesn't matter what or how much data you try to pre-load, only the last 32 kByte is going to make any difference.

This leads some people working on data differencing to use a window or history limit much larger than other compression researchers.

6.1.2 window size

The LZ77-style decompressors have a fixed-size "window" of recently-decompressed text, and the "copy references" can only refer to text inside that window.

Many compression algorithms, in theory, have no inherent "history limit" -- such as LZ78-style algorithms and adaptive Huffman algorithms. However, in practice, most implementations of compression algorithms periodically reset their dictionary and start fresh. So they have a block size the length of the maximum text between dictionary resets.

With many early compression algorithms, an easy way to improve performance is to increase the window size. (i.e., either literally increase the "window" buffer for LZ77 algorithms, or simply reset the dictionary less often for LZ78-style algorithms). These early compression algorithms were typically developed on machines that had far less RAM than modern machines, and so the "window size", the "reset frequency", and "internal dictionary size" were constrained by that limit. (Those early machines also ran much slower than modern machines, and many algorithms -- such as the LZRW series of algorithms -- were heavily constrained by these speed limits; it is unclear what effect this had on dictionary size).

For simplicity, we will consider the effect of doubling the window size on a variety of algorithms. The original "small window" algorithm has some fixed window size W, and the widened "larger window" algorithm has some fixed window size 2*W, which can be thought of as the "near window" with all the same stuff in it as used by the small-window algorithm, and the "far window" that has stuff that is inaccessible to the small-window algorithm, but perhaps the larger-window algorithm can exploit that stuff to get better compression.

Alas, there are diminishing returns to increasing the size of the window. In fact, almost always there is some data-dependent "optimum" window size. Increasing the size of the window beyond the optimum leads to worse compression (larger compressed files).

There are 4 reasons that windows larger than that optimum size are counter-productive:

1. Some LZ77-style algorithms use a fixed-length linear offset. Doubling the size of the window necessarily increases the length of each and every copy item by 1 bit. With typical 16-bit copy items, that makes the compressed file longer (worse compression) by a factor of 17/16, unless the compressor can find strings in the far window with matches that are longer than any strings in the near window.

2. Other LZ77-style algorithms use variable-length offsets. Typically more distant offsets are longer. Doubling the size of the window generally leaves the length of "extremely close" copy items exactly the same, increases the size of a few copy items near the outer limit of the near window by 1 bit, and requires even larger copy items to refer to stuff in the far window. So there's hardly any penalty for increasing the window size for this (2) category compared to the (1) category. However, the wins are not quite as good. In many cases, even when there is an excellent long match in the far window, longer than any match in the near window, it doesn't make the compressed file any smaller -- the long copy item needed to refer to that distant matching string may be the same length or even longer than two short copy items that can re-assemble that same string from two places in the near window.

3. As Jeff J. Kato et al point out,[12] It can even be counter-productive to have data from one kind of file in the window/dictionary when trying to compress data with somewhat different characteristics, or when trying to compress a file whose characteristics change ("evolve") from one part to the next. Often it's better to reset the dictionary and start from scratch.

However, sometimes these "large window" compressors have great gains -- in particular, when we're currently trying to compress a file that is not merely the same kind of file, but is bit-for-bit identical to some early file. Some large-window techniques also work well when a file is *mostly* identical with relatively few edits here and there.

The default window size or block size for some compression software is:

- gzip: 32 kByte sliding window
- bzip2: blocks of 900 kB
- DEFLATE (as used in ".zip", ".jar", ".png", etc.): 32 kByte sliding window
- The GIF standard mentions two kinds of LZW encoders:

    - some GIF encoder implementations reset the dictionary every time it gets full --

i.e., the compressor sends the “clear code” to reset the dictionary after every (max dictionary size - number of hard-wired entries) = 2048-(256+2) compressed symbols.

- - All GIF decoders are required to support “deferred clear”. Many images can be stored in a smaller .gif compressed file if the compressor uses such a “deferred clear”.[13][14][15]

). The GIF LZW encoder becomes non-adaptive after the dictionary is full, while it is “waiting” for the next clear code.

- lzip
- lrzip and SuperREP has “infinite” window (not limited by available RAM)
- rzip has a 900 MB window (1000 times larger than bzip2) http://en.wikipedia.org/wiki/Rzip
- Information Zipping Optimizer (IZO) also has a window not limited by available RAM https://www.usenix.org/conference/lisa-08/izo-applications-large-window-compression-virtual-machine-management
- xdelta

http://en.wikipedia.org/wiki/xdelta

[1] w:data differencing

[2] http://static.usenix.org/event/lisa08/tech/full_papers/smith/smith_html/

[3] Dutch T. Meyer and William J. Bolosky. “A Study of Practical Deduplication”.

[4] Wikipedia: data deduplication

[5] http://code.google.com/p/xdelta/

[6] “By using suffix sorting (specifically, Larsson and Sadakane’s qsufsort) and taking advantage of how executable files change, bsdiff routinely produces binary patches 50-80% smaller than those produced by Xdelta, and 15% smaller than those produced by .RTPatch ... A far more sophisticated algorithm, which typically provides roughly 20% smaller patches, is described in my doctoral thesis.” -- Colin Percival. “Binary diff/patch utility”

[7] Colin Percival. “Matching with Mismatches and Assorted Applications”. thesis. 2006.

[8] Dana Shapira and James A. Storer. “In Place Differential File Compression”. 2005.

[9] http://www.gzip.org/zlib/manual.html#deflateSetDictionary http://www.gzip.org/zlib/manual.html#inflateSetDictionary

[10] “Can compression algorithm “learn” on set of files and compress them better?"

[11] comp.compression: “Employing a set of data files guaranteed to be available at the time of decompression”

[12] Jeff J. Kato, David W. Ruska, David J. Van Maren. US Patent 4847619 “Performance-based reset of data compression dictionary”. 1987.

[13] Attila Tarpai. “Deferred clear code in GIF”. 2010.

[14] “Cover Sheet for the GIF89a Specification: Deferred clear code in LZW compression”

[15] GIF file format: Portability Warning 2: The Deferred Clear Code Problem

Chapter 7

Dictionary compression

7.1 Dictionary Compression

Remember the wealth of information a soldier gets from a mike click? With certain types of data, it makes sense that if you could encode the data using information that is not stored in the actual datafile you are compressing, you could significantly reduce known forms of storage and the requirements for communication. Some algorithms replicate process by including an internal data table, often referred to as dictionary, hence the term *dictionary compression*.

Due to its intrinsic characteristic of permitting gains when used in communications, as the example of the mike click demonstrates, there is a pre-understood code system agreed and shared between the emitter and the receptor. It is not without a cause that most communication network oriented compression algorithms rely in some form of dictionary compression.

An example is a simple dictionary built into the compression algorithm scheme. If you already know the most compact storage scheme for a specific type of data, then you can simply substitute the more compact state for the actual data. Consider the word "supercalifragiliciousexpialadocious" Now suppose that in your dictionary there are only 11 words starting with super, and this is the eleventh. Well it's obvious that Super11 would be unique enough that given the same dictionary at the time of decompression, you could recover the full word with no data loss. With A and An, however it is not quite so obvious that you could store information about the word in less space than everything but the one letter word.

Now consider that you have the Internet, and you want to store an article by LaVogue. You could store the file on your own computer, or, alternately you could store a link to the file similar to the hypertext links you have on your homepage. Is it compression? Maybe not, but you can recover the original text with a slight delay while it downloads.

The mind boggles at the efficiency of a compression system that contains all knowledge in the world on a single server-farm, and uses it to reduce the storage of information on the client computer. Even more fantastic would be the legal snarl such a system would create and the privacy concerns and financial costs associated with recovery of compressed information that was proprietary in the first place. But a net-aware compression server, might be able to achieve truly radical compression rates. Essentially all that the home file would have to contain was the key sequence to trigger the decompression of a specific data cloud. The home user could even compress his compression files because multiple files each containing a link to a specific compression file structure, could be themselves turned into a single file, that contained the compression file structure of all the composite files. Infinite compression at the user level, but truly massive storage at the server level.

7.2 non-adaptive dictionary compression

7.3 Text compression using 4 bit coding

Pike's 1981 compression algorithm, like several early text compression algorithms, used a more or less literal dictionary of English words.

Pike's algorithm used a fixed dictionary composed of 205 of the most popular English words, and the ASCII characters. By using a variable-length coding of 4, 8, or 12 bits to represent these words and the remaining ASCII characters, and some clever handling of uppercase letters and the space between words, Pike's algorithm achieved a compression ratio of 3.87 bits per character for typical English text. [1]

Adding more English words to the dictionary helps compression, but with diminishing returns. Adding certain non-words to the dictionary -- common letter pairs and triplets, common prefixes, common suffixes, etc. -- helps compression, since they can be used to at least partially compress words that aren't in the dictionary, but adding such non-words also gives diminishing returns.

Later compression algorithms kept the idea of a short codeword representing something longer, but generalized

it: a dictionary coder expands codewords to a sequence of bytes, a sequence that may or may not line up on English word boundaries.

7.4 Smaz

Smaz is a simple compression library suitable for compressing very short strings, such as URLs and single English sentences and short HTML files.[2]

Much like Pike's algorithm, Smaz has a hard-wired constant built-in codebook of 254 common English words, word fragments, bigrams, and the lowercase letters (except j, k, q). The inner loop of the Smaz decoder is very simple:

- Fetch the next byte X from the compressed file.
  - Is X == 254? Single byte literal: fetch the next byte L, and pass it straight through to the decoded text.
  - Is X == 255? Literal string: fetch the next byte L, then pass the following L+1 bytes straight through to the decoded text.
  - Any other value of X: lookup the X'th "word" in the codebook (that "word" can be from 1 to 5 letters), and copy that word to the decoded text.
- Repeat until there are no more compressed bytes left in the compressed file.

Because the codebook is constant, the Smaz decoder is unable to "learn" new words and compress them, no matter how often they appear in the original text.

7.5 Adaptive dictionary algorithms

Most dictionary algorithms are adaptive: rather than a fixed, hard-wired dictionary like Pike's algorithm, as the decoder compresses the text, when it encounters letter triplets or entire words that are not yet in the dictionary, the decoder adds them to the dictionary.

If there are too many rare triplets, syllables, or words in the dictionary, it increases the codeword size of the more common ones, hurting compression ratio. If some triplet, syllable, or word is left out of the dictionary, then when it does occur, it cannot be represented by a simple dictionary reference and must be spelled out using more bits, also hurting compression ratio. This balancing act makes it difficult to build an optimal dictionary.[3]

The hard-wired dictionary used to initialize the data compression algorithm has a big effect on the compression ratio in the early phase of the compression. An adaptive dictionary algorithm eventually adds to the dictionary all the syllables and common words used in that specific text, and so the initial hard-wired dictionary has little effect on the compression ratio in later phases of the compression of large files.[3]

7.6 LZ77 algorithms

LZ77 type decoders expand the compressed file by expanding "copy items" in that compressed file into larger strings. This enables "duplicate string elimination": only the first copy of a string is stored in the compressed file. All other copies refer to a previous copy.

These decoders have the general algorithm:

- Is the next item a literal? Then copy it directly from the compressed file to the end of the decoded text.
- Is the next item a copy item? Then break it into a "length" and a "distance". Find the text that starts that "distance" back from the current end of decoded text, and copy "length" characters from that previously-decoded text to end of the decoded text.
- Repeat from the beginning until there is no more items in the compressed file.

While all LZ77 style algorithms have this same decoder structure, there is a huge variety of ways that decoders decide whether an item is a literal or a copy items, and also a variety of ways that they extract the "length" and the "distance" from the copy item.

Many decoder algorithms combine LZ77 with other compression ideas. As of 2008, the most popular LZ77 based compression method is called DEFLATE; it combines LZ77 with Huffman. We discuss DEFLATE in a later chapter of this book, Data Compression/Multiple transformations#DEFLATE.

7.6.1 LZSS and LZRW1-A

Lempel-Ziv-Storer-Szymanski (LZSS) was created in 1982 by James Storer and Thomas Szymanski. LZRW1-A was published in 1991 by Ross Williams.

The LZSS and LZRW1-A decompressors have a very similar form:

For each copy item, fetch a "literal/copy" bit from the compressed file.

- 0: literal: the decoder grabs the next byte from the compressed file and passes it straight through to the decompressed text.
- 1: copy item: the decoder grabs the next 2 bytes from the compressed file, breaks it into a 4 bit "length" and a 12 bit "distance". The 4 "length"

bits are decoded into a length from 3 to 18 characters. Then find the text that starts that "distance" back from the current end of decoded text, and copy "length" characters from that previously-decoded text to end of the decoded text.

- Repeat from the beginning until there is no more items in the compressed file.

7.6.2 LZJB

The LZJB[4] decoder fetches a "literal/copy" bit from the compressed file.

- 0: literal: the decoder grabs the next byte from the compressed file and passes it straight through to the decompressed text.
- 1: copy item: the decoder grabs the next 2 bytes from the compressed file, breaks it into a 6 bit "length" and a 10 bit "distance". The 6 "length" bits are decoded into a length from 3 to 66 characters. Then find the text that starts that "distance" back from the current end of decoded text, and copy "length" characters from that previously-decoded text to end of the decoded text.
- Repeat from the beginning until there is no more items in the compressed file.

After pulling a 1 bit from the control word, a copy item is in only one format:

- LLLLLLdd dddddddd : copy L+3 bytes from d bytes before the most-recently-decoded byte.

7.6.3 LZ77

LZ77[5] The original LZ77 decoder fetches a "distance/length/literal" block from the compressed file.

The LZ77 decoder algorithm is:

- Find the text that starts that "distance" back from the current end of decoded text, and copy "length" characters from that previously-decoded text to end of the decoded text. (Sometimes the "length" is 0, so this "copy" does nothing).
- Then copy the literal character from the compressed file to the end of the decoded text.
- Repeat from the beginning until there is no more items in the compressed file.

7.6.4 BARF

The "BARF" archiver[6] (when it's not cheating) uses a simple byte-oriented LZ77-like code. The inner loop is:

- Fetch a byte X.
  - X in the range 0...31? Literal: copy the next X literal bytes from the compressed stream to the end of the decoded text.
  - X in the range 32...255? Copy item: copy the two bytes from X-32 places back in the decoded text.
- Repeat from the beginning until there is no more items in the compressed file.

7.6.5 LZF

LZF is a fast compression algorithm that takes very little code space and working memory (assuming we have access to the most recent 8 kByte of decoded text).

LibLZF by Marc Lehmann is designed to be a very small, very fast, very portable data compression library for the LZF compression algorithm.

LibLZF source code

LZF is used in TuxOnIce and many other applications.

The LZF decoder inner loop is: For each copy item, the LZF decoder first fetches a byte X from the compressed file, and breaks the byte X into a 3 bit length and a 5 bit high_distance. The decoder has 3 cases: length==0, length==7, and any other length.

-
  - length == 0? (i.e., X in the range 0...31?) Literal: copy the next X+1 literal bytes from the compressed stream and pass them straight through to the current location in the decompressed stream.
  - length in 1...6 ? short copy: Use that value as length (later interpreted as a length of 3 to 8 bytes).
  - length == 7? long copy: fetch a new length byte and add 7. The new byte could have any value 0 to 255, so the sum is 7 to 262 (which is later interpreted as 9 to 264 bytes).
  - Either kind of copy: Fetch a low_distance byte, and combine it with the 5 high_distance bits to get a 13-bit distance. (This implies a 2^13 = 8KByte window).
  - Either kind of copy: Find the text that starts that "distance" back from the current end of decoded text, and copy "length+2" characters from that previously-decoded text to end of the decoded text.

- Repeat from the beginning until there is no more items in the compressed file.

Each LZF "item" is one of these 3 formats:

- 000LLLLL <L+1> ; literal reference
- LLLddddd dddddddd ; copy L+2 from d bytes before most recently decoded byte
- 111ddddd LLLLLLLL dddddddd ; copy L+2+7 from d bytes before most recently decoded byte

7.6.6 FastLZ

FastLZ by Ariya Hidayat is a free, open-source, portable real-time compression library for the FastLZ algorithm. *(Ariya Hidayat apparently developed the FastLZ algorithm based on the LZF algorithm?)* FastLZ is a drop-in replacement for Marc Lehmann's LibLZF. It has pretty much the same compressed size and compression time as LibLZF, and significantly faster decompression -- about 2/3 the time of LibLZF. [7][8]

The FastLZ decoder inner loop is very similar to the LZF inner loop:

- Fetch a byte X. Divide the byte X into the 3 bit length and a 5 bit high_distance.
  - length == 0? (i.e., X in the range 0...31?) Literal: copy the next X+1 literal bytes from the compressed stream to the end of the decoded text.
  - length in 1...6 ? Copy: Use that value as length (later interpreted as a length of 3 to 8 bytes).
  - length == 7? Copy: fetch a new length byte and add 7. The new byte could have any value 0 to 254, so the sum is 7 to 261 (which is later interpreted as 9 to 263 bytes).
  - (When the new byte is all-ones (255), it serves as an escape to even longer lengths ...)
  - Fetch a low_distance byte, and combine it with the 5 high_distance bits to get a 13-bit distance. (This implies a 2^13 = 8KByte window).
  - If the distance is all-ones (0x1FFF): fetch 2 new distance bytes to give a full 16 bit distance, and add it to the 0x1FFF all-ones distance. (This implies a 2^16 + 2^13 = 64 KByte + 8 KByte window)
  - find the text that starts that "distance" back from the current end of decoded text, and copy "length+2" characters from that previously-decoded text to end of the decoded text.
- Repeat from the beginning until there is no more items in the compressed file.

If a compressed file only uses distances within the 2^13-1 window, and lengths less than 2^8+7, both the FastLZ and the LZF decoders will produce the same decompressed plaintext.

Each FastLZ "item" is one of these 5 formats (the LZF formats, plus a few extensions):

- 000LLLLL <L+1> ; literal reference
- LLLddddd dddddddd ; copy L+2 bytes from d bytes before most recently decoded byte
- 111ddddd LLLLLLLL dddddddd ; copy L+2+7 bytes from d bytes before most recently decoded byte
- LLL11111 11111111 dddddddd dddddddd ; copy L+2 bytes from d+0x1FFF bytes before most recently decoded byte
- 11111111 LLLLLLLL 11111111 dddddddd dddddddd ; copy L+2+7 bytes from d+0x1FFF bytes before most recently decoded byte

It's not clear why FastLZ is so much faster than LZF. Its "extensions" are apparently not being used, since the compressed file sizes on the benchmark files are not significantly different. If the same implementation techniques -- 16-bit copies whenever possible, special-case for "run" (pseudo-RLE), etc. -- were applied to LZF, would it be just as fast? Or is it the case that these "extensions" actually are being used a lot, and somehow they are faster (but roughly the same size) as the LZF approach?

7.6.7 miniLZO

LZO is a compressed data format and portable lossless data compression library written in ANSI C developed and maintained by Markus F.X.J. Oberhumer.[9] LZO is slightly slower and produces somewhat better compressed file sizes than LZF. LZO is much faster than DEFLATE/gzip -- it runs in about half the time -- and produces somewhat worst file sizes than DEFLATE/gzip.

miniLZO is a lightweight subset of the LZO library.

7.6.8 LZ4

LZ4 is a compressed data format and portable lossless data compression library developed and maintained by Yann Collet.

LZ4 is used in many compressed file systems including OpenZFS, GNU Grub, HAMMER, and Squashfs.[10] Open source implementations are available for a variety of languages and (cross-platform) operating systems, and there is an active discussion group.[11]

The LZ4 decoder inner loop is:[12]

For each copy item, the LZF decoder first fetches a byte X from the compressed file, and breaks the byte X into a 4 bit literal_length and a 4 bit copy_length.

The literal decoder has 2 cases: literal_length==7, and any other length.

- literal_length == 0...6 ? copy the next literal_length literal bytes from the compressed stream and pass them straight through to the current location in the decompressed stream. (If the literal_length field is 0, then there is no literal).
- literal_length == 7? long copy: fetch a new length byte and add 7. The new byte could have any value 0 to 254, so the sum is 7 to 261 bytes of literal.

(... say something about the special case of literal_length == 7 followed by any number of additional 255 bytes ...) Then copy the sum literal bytes from the compressed stream and pass them straight through to the decompressed stream.

After the literal bytes (if any) is the low byte of the distance, followed by the high byte of the distance. (This implies a 2^16 = 64KByte window). A distance of 1 indicates "current position - 1 byte"; a distance of 0xffff indicates "current position - 65535 bytes"; a distance of 0 is invalid and should never happen. The actual number of bytes to copy is decoded similar to the actual number of literals to copy, except the minimum length of a match is 4 bytes.

- copy_length == 0...6 ? length = copy_length + 4 bytes.
- copy_length == 7? long copy: fetch a new length byte. length = that new length byte + copy_length + 4 bytes. The new byte could have any value 0 to 254, so the sum is 11 to 265 bytes of literal.

(... say something about the special case of copy_length == 7 followed by any number of additional 255 bytes ...)

- Either kind of copy: Find the text that starts that "distance" back from the current end of decoded text, and copy "length" characters from that previously-decoded text to end of the decoded text.
- Repeat from the beginning until there is no more items in the compressed file.

Like most LZ77-style algorithms, LZ4 supports pseudo-RLE -- the copy_length can be longer than the offset distance; matches can overlap forward.[13]

In order to ensure that the decoder will never read beyond the input buffer, nor write beyond the output buffer, each block starts with a "block size" field that indicates exactly how many bytes of *compressed* data are in that block; and the last 5 bytes of the block are always literals. The last copy item is incomplete and stops right after its literals.

The "LZ4-HC" ("high compression") and "LZ4" use exactly the same format and the same decoding function; the difference is in the encoder.[14][15]

7.6.9 QuickLZ

QuickLZ [16]

7.6.10 LZS

LZS: Lempel–Ziv–Stac compression. LZS compression is specified for various Internet protocols, including RFC 1967, RFC 1974, RFC 2395, and RFC 3943.[17] Stac Electronics sued Microsoft for infringement of LZS patents. Since then, several of those patents have expired.

The LZS decoder fetches a "literal/copy" bit from the compressed file.

- 0: literal: the decoder grabs the next byte from the compressed file and passes it straight through to the decompressed text.
- 1: copy item: the decoder grabs the next bit from the compressed file.
  - That bit selects how many bits of "distance" to pull from the file:
    - 1: the decoder grabs a 7 bit "distance".
      - If this distance is the nonsensical value "0", it really represents the end-of-message. Exit the loop.
    - 0: the decoder grabs a 11 bit "distance" (this implies a 2 kibiByte sliding window).
  - Then the decoder grabs a "length" value from the compressed file.
    - However, the length is not encoded in a fixed-length field, but in a variable-length pattern:
    - 00 represents length 2
    - 01 represents length 3
    - 10 represents length 4
    - 1100 represents length 5
    - 1101 represents length 6
    - 1110 represents length 7
    - 1111 0000 represents length 8
    - 1111 0001 represents length 9
    - 1111 1111 0010 represents length 25
    - In general, except for the first 6 special cases listed above, "1111" repeated N

times followed by 4 more bits that represent a value Y in the range 0...14, are decoded into a length value (N*15 + Y - 7).

- Then find the text that starts that "distance" back from the current end of decoded text, and copy "length" characters from that previously-decoded text to end of the decoded text.

- Repeat from the beginning, until the special "zero distance" value was decoded.

The complex variable-length encoding technique used in LZS allows "recent" byte pairs to be compressed into 11 bits, and even "less recent" byte pairs, as long as they are in the 2 kibiByte sliding window, to be compressed in 15 bits. This variable-length encoding technique is one of several kinds of "universal code", one of many kinds of "fixed Huffman encoding" tables, which we discuss in the Entropy chapter of this Wikibook.

In text files, repeated pairs are far more common than longer repeated strings. LZS gives at least 1 bit of compression on this common form of redundancy. This gives LZS an advantage over compression algorithms which are unable to compress repeated strings shorter than 3 characters, and so must store such short repeats as-is, or even expand them by a bit.

7.6.11 Snappy

The "Snappy" algorithm is tuned for very high speeds (even at the cost of mediocre compression). On benchmark files, Snappy is an order of magnitude faster than zlib but compressed files 20% to 100% bigger than zlib.[18] Snappy is faster than LZF.[19]

The inner loop of the "Snappy" decompressor decodes "elements" that always begin with a tag byte in one of these formats:

- xxxxxx00 ; literal follows
- xxxxxx01 ; copy with 1-byte offset
- xxxxxx10 ; copy with 2-byte offset
- xxxxxx11 ; copy with 4-byte offset

Details:

- LLLLLL00 <L+1> ; where L in 0..59: literal reference for 1...60 (inclusive) literal bytes.
- LLLLLL00 <L-59> <length> ; where L in 60..63: these special "lengths" indicate 1, 2, 3, or 4 bytes, respectively, of actual length bytes follow the tag byte. Those length bytes are followed by the literal itself.
- dddLLL01 dddddddd ; copy with 1-byte offset: copy L+4 bytes from d bytes before the most recently decoded byte. *(Why L+4, giving lengths 4..11 ? Would L+3, giving lengths 3..10, give better compression? Perhaps even L+2 allowing us to copy the occasional byte pair?)*
- LLLLLL10 dddddddd dddddddd ; copy with 2-byte offset: copy L+1 bytes from d bytes before the most recently decoded byte.
- LLLLLL11 dddddddd dddddddd dddddddd dddddddd ; copy with 4-byte offset: copy L+1 bytes from d bytes before the most recently decoded byte.

7.6.12 PalmDoc

PalmDoc combines LZ77 with a simple kind of byte pair compression. PalmDoc Algorithm: [20]

PalmDoc files are decoded as follows:

- Read a byte from the compressed stream. If the byte is
  - 0x00: "1 literal" copy that byte unmodified to the decompressed stream.
  - 0x09 to 0x7f: "1 literal" copy that byte unmodified to the decompressed stream.
  - 0x01 to 0x08: "literals": the byte is interpreted as a count from 1 to 8, and that many literals are copied unmodified from the compressed stream to the decompressed stream.
  - 0x80 to 0xbf: "length, distance" pair: the 2 leftmost bits of this byte ('10') are discarded, and the following 6 bits are combined with the 8 bits of the next byte to make a 14 bit "distance, length" item. Those 14 bits are broken into 11 bits of distance backwards from the current location in the uncompressed text, and 3 bits of length to copy from that point (copying n+3 bytes, 3 to 10 bytes).
  - 0xc0 to 0xff: "byte pair": this byte is decoded into 2 characters: a space character, and a letter formed from this byte XORed with 0x80.
- Repeat from the beginning until there is no more bytes in the compressed file.

7.7 dictionary algorithms

7.7.1 fixed byte pair encoding

The simplest dictionary algorithm is byte pair encoding with a fixed dictionary of byte pairs (a bigram dictionary).

Simple byte-pair encoding, also called dual-tile encoding or DTE, is often used to compress text in video game ROMs.[21]

A byte pair decoder assumes that English text uses only a relatively small "alphabet" of letters and symbols. A byte pair decoder assumes that certain bytes never occur in English language text (the "high" bytes 0x80 to 0xFF and a few other bytes). When the "impossible" bytes occur in the compressed text, the decoder looks that byte up in a fixed dictionary, and emits the corresponding pair of bytes. Otherwise, the decoder copies the compressed byte to the plaintext more or less as-is. Often the 0x00 byte marks the end of the compressed string, so standard C string-handling routines can be re-used to handle compressed text. The dictionary should include all the most-frequent letter pairs used in the plaintext ("th", "he", " t", etc.). There are several different implementations of this algorithm, each one using a different (incompatible) dictionary.[22][23]

7.7.2 byte pair encoding

DTE can be decoded in a single pass -- each byte of compressed text is either a "normal" printable character that stands for itself, or else it is a "unprintable" byte that stands for exactly two normal printable characters. Recursive byte pair encoding loosens that restriction: With recursive byte pair encoding, a "unprintable" byte stands for two bytes. Sometimes both of those two bytes are normal printable characters, just like DTE. Sometimes one or both bytes may be (some other) "unprintable" byte, requiring repeated decoding passes until only printable characters remain.[24]

[25] [26]

7.7.3 SCZ byte pair encoding

SCZ is a dynamic byte pair algorithm and compressed format developed by Carl Kindman, who released one implementation and its test suite under a LGPL license.[27]

The SCZ compressed format is a series of segments. Each segment has a few bytes of header and footer, including a iteration count and a checksum, surrounding a repeatedly-compressed block.

The inner loop of the SCZ decoder is given a dictionary of 256 entries, and a special escape symbol. The inner loop scans through a block of "compressed segment data" and generates a new uncompressed block something like this:

- Read the next byte from the compressed block.
- If the byte read is the special escape byte, discard the escape byte and read the following compressed byte as a literal byte, and write that literal byte to the decompressed block.
- Otherwise look up that byte in the dictionary.
  - If the byte is a "marker character", the dictionary has a byte pair to replace it with -- write those 2 bytes to the decompressed block.
  - Otherwise the dictionary indicates the byte is a literal byte -- write the byte read to the decompressed block.
- Repeat until no more data in the compressed block.

The "escape byte" makes it possible to compress a file that has a byte pair repeated many times, even when the file already uses all 256 possible byte values -- the compressor deliberately expands some rare byte into two bytes (the escape byte and the rare byte), in order to re-use that rare byte as a "marker byte" to represent that byte pair (compressing each instance of those two bytes into one byte).

The middle loop of the SCZ decoder is given an iteration count and a block of data. The middle loop and generates the corresponding plaintext something like this:

- If the iteration count is zero, the rest of the block is fully decompressed data -- done!
- Read the header from the compressed block
  - read a byte from the compressed block; this byte becomes the new escape symbol.
  - Read the symbol count N, and read N dictionary entries into a dictionary. (Each entry is 3 bytes: the "marker" byte as stored in the compressed text, and the pair of bytes it represents in the uncompressed text.
- Pass the escape symbol, the dictionary, and the rest of the compressed block (the compressed segment data) to the inner loop, decompressing it to an uncompressed block.
- swap the (now-empty) compressed block with the uncompressed block.
- Decrement the iteration count and repeat from the beginning.

The outer loop of the SCZ decoder breaks the file up into a series of segments, breaks each segment up into a header (which contains the iteration count, among other things), a block of compressed data (which the outer loop passes to the inner loop for decoding), and a footer (which contains a checksum used to detect problems).

7.7.4 ISSDC: digram coding

7.7.5 LZ78 algorithms

LZ78

7.7.6 LZW

A well-known example of the dictionary based technique is the LZW data compression, since it operates by replacing strings of essentially unlimited length with codes that usually range in size from 9 to 16 bits.

The LZW algorithm, as used in the GIF file format, is perhaps the most famous and controversial compression algorithm.

While decoding a codeword to a “word” of 1 or more letters and emitting those letters to the plaintext stream, a LZW decompressor adds a new “word” to the dictionary. The new word is the concatenation of the previously-decoded word and the first letter of the word represented by the current codeword.

Larger codewords allow larger dictionaries, which typically improves compression.[28]

[29]

GIF

The GIF file format typically starts with 9 bit codewords and gradually increases the width of the codeword, as necessary, up to a maximum codeword width of 12 bits. Many GIF implementations statically allocate a dictionary with 2^12 = 4096 dictionary entries, one for each possible codeword.

LZMW

LZMW (1985) by V. Miller, M. Wegman is a modification of LZW.

While decoding a codeword to a “word” of 1 or more letters and emitting those letters to the plaintext stream, a LZMW decompressor adds a new “word” to the dictionary. The new word is the concatenation of the previously-decoded word and the entire word (possibly the same word) represented by the current codeword. This allows dictionary entries to grow in length much more rapidly than LZW.

LZAP

LZAP (1988) by James Storer) is a modification of LZMW.

While decoding a codeword to a “word” of 1 or more letters and emitting those letters to the plaintext stream, a LZAP decompressor adds at least 1 and often more new “words” to the dictionary. The new words are the concatenation of the previous match with every initial substring of the current match. (“AP” refers to “all prefixes”). For example, if the previous match is “com” and the current match is “press”, then the LZAP decompressor adds 5 new “words” to the dictionary: “comp”, “compr”, “compre”, “compres”, and “compress”, where LZW would only add “comp”, and LZMW would only add “compress”.

This allows the number of words in the dictionary to grow much more rapidly than LZW or LZMW.

LZWL and word-based LZW

In 2005, Jan Lánský introduced syllable-based compression. .[3]

The LZWL compressor uses a syllable detector to divide the plain text into syllables. [30][31] When the LZWL decompressor pulls a “phrase index” from the text, it adds a new phrase to the dictionary -- formed from the entire previous phrase concatenated with the first syllable of the current phrase.

A similar “word-based LZW” divides the plain text into English words (in general, alternate maximal strings of alphanumeric characters and non-alphanumeric characters).[32]

Character-based compression seems to give better compression with smaller files; word-based compression seems to give better compression with larger files. Some researchers suggest that syllable-based compression methods might give better compression with intermediate-sized files -- such as HTML pages.[3]

LZW implementation tips

The dictionary used by the decompressor doesn't really need to store a literal string for each dictionary entry. Since each dictionary entry is composed of some previous dictionary entry concatenated with a single character, all the decompressor needs to store for each entry is an integer and a character: the index of some previous dictionary entry, and a literal suffix character. [28]

This same compact dictionary also works with LZAP. With LZMW, rather than each entry storing a single suffix character, it will instead store a second integer: the index to some, possibly the same, previous dictionary entry.

It’s usually simpler and faster for the decoder to special-case decode any codeword with a value less than 256, and immediately output the literal byte it represents, rather than go through the motions of looking up such a codeword in the dictionary. [28]

Some LZW compressors use a hash table to rapidly map the next few plain text characters to a dictionary entry. This works about the same as the hash table used to speed up LZ77-style compressors.

Some implementations of LZW and other LZ78 variants use a special search tree that takes advantage of the dictionary structure.[33]

7.7.7 LZX

(We discuss LZX in a later chapter, Data Compression/Order/Entropy#LZX)

7.7.8 Reduced offset Limpel Ziv (ROLZ)

The above LZ77 variations of LZ77 encode a linear, uncoded, fixed-length offset. The possible values of that offset at any particular instant of decoding often include many values that will *never* be used. For example, once the decompressor pulls a copy length of 4, it may see dozens of copies of "the " and "and " in the text, but it knows that the compressor will always pick the most recent one -- all the values of the offset that select some other copy of "the " or "and " will never occur.

If we could somehow "skip over" those impossible offsets, we could encode the offset in fewer bits, and still be able to use the same (offset,length) phrase that the above LZ77 encoders would use.

Also, many possible values of the offset are highly unlikely to occur. For example, once the decoder finishes emitting a phrase that ends in "ch", in normal English text the next character is almost never another consonant. So all the offsets that point to a phrase starting with some consonant will almost never be used. By skipping over those highly unlikely offsets, we could encode the offset of the next phrase in fewer bits, and almost always be able to copy the same (offset, length) phrase that the above LZ77 encoders would use. (In some rare cases, the above LZ77 encoder picks some phrase that a ROLZ algorithm deems "unlikely" and skips over, forcing us to copy some other, shorter phrase; but hopefully sacrificing a few bits in those situations is paid back in saving a few bits in almost every other situation).

By "reduced offset", we mean that we have reduced the number of bits in the compressed file used to select a particular offset, when compared to using a linear, uncoded, fixed-length offset. We do *not* mean that the maximum possible offset is reduced -- in fact, most ROLZ compressors can copy a phrase from a much more distant offset than the typical 2^10 or 2^12 byte limit of linear, uncoded offsets.

Using something other than a linear, uncoded, fixed-length offset necessarily requires more computation at decode-time to translate that compressed selection value to the actual position in the decompressed-so-far text. This is yet another tradeoff between speed and compression ratio.

Ross Williams points out that all LZ77 and LZ78 algorithms can be cast into the form of Markov algorithms.[34]

Malcolm Taylor wrote one (the first?) implementation of ROLZ as part of the WinRK compression suite. Ilia Muraviev wrote the first open-source implementation of ROLZ compression.[35]

LZRW3-a Of all the compressors in the LZRW series of "fast" data compressors, the LZRW3-a(8) gives the best compression on the large text benchmark. Unlike most other ROLZ compressors, it completely ignores the most-recently-decoded byte of plaintext context. The inner loop of the LZRW3-a decoder first fetches a "literal/copy" bit from the compressed file.

- 0: literal: the decoder grabs the next byte from the compressed file, which represents itself. Pass it straight through to the decompressed text. Every literal byte is "interesting", so a pointer to this byte is (eventually) added to the "table of 2^12 interesting locations in the previously-decoded text". Hopefully next time this byte can be copied as part of a long phrase via that pointer, rather than laboriously spelled out with literals again.
- 1: copy item: the decoder grabs the next 2 bytes from the compressed file, which represents a phrase. The decoder breaks it into a 4 bit "length" and a 12 bit selector.
- The 4 "length" bits are decoded into a length from 3 to 18 characters.
    - It uses that 12 bit selector to pick a location from the "table of 2^12 interesting locations in the previously-decoded text", and copies length characters from that location to the end of the decoded text. Also: add to the table a pointer to the beginning of this phrase.
- Repeat from the beginning until there is no more items in the compressed file.

LZRW3-a(8) tries to (approximately) preserve the 8 most-recent instances of every 3-byte phrase that occurs in the plaintext. (The specific details of which partition the decompressor adds the pointer to, and the pseudo-random replacement used to discard old pointers when a partition is full, are beyond the scope of this document). (In effect, the decompressor expands a 12 bit code to a 20 bit uncoded offset into the 1 MB plaintext buffer). *(too much detail)*

LZRW4 (Ross Williams, 1991) can be seen as a kind of ROLZ algorithm. Rather than a linear, uncoded offset for each phrase, the offset is represented by a 5 bit value. At the start of each phrase, the LZRW4 decoder hashes the two previous characters to obtain a partition number. The 5 bit value selects one of the 32 pointers in the context-selected partition. That points to some previously-decoded location in the 1 MB plaintext buffer. (In effect, the decompressor has expanded the 5 coded bits to a 20 bit uncoded offset). Then the decompressor pulls 3 bits from the compressed text and decodes that into 8 possible length values (2,3,4,5,6,7,8,16), then copies the selected (offset,length) phrase to the plaintext. Then the context partitions are updated. Because a "copy

item" is only 9 bits (flag bit + offset selector + length), it does a good job compressing common byte pairs. Literal items are also 9 bits (flag bit + literal octet). (To increase speed, Williams doesn't store a pointer to *every* time a 2-byte context occurs in the partition table, but only the 2 byte contexts that occur at the *end* of some phrase). (To reduce memory use, Williams doesn't keep track of the *exact* 2-byte context using 2^16 partitions, but instead hashes the 2 previous bytes of context into a 12 bite context with 2^12 partitions). *(too much detail ...)*

Perhaps the most extreme form of "reduced offset" is to reduce the offset to zero bits. This special case of ROLZ is called LZP.

7.7.9 LZP

LZP compression was invented by Charles Bloom.[36] A LZP decompressor examines the current context (perhaps previous 3 bytes), finds the most recent occurrence of that context in the plain text (or some approximation), and uses only that one fixed offset. That gives us an extremely short compressed phrase (a flag bit + a match length). In some cases, that one fixed offset is the same one that LZ77 would have used -- in these cases, we've saved a lot of bits because we didn't store the offset. In other cases, that one fixed offset is not exactly the same one that LZ77 would have used, but a larger offset that gives us exactly the same copied phrase -- likewise, we've saved lots of bits. Inevitably, sometimes that one fixed offset gives us fewer useful bytes than the offset LZ77 would have chosen -- in the worst case, zero useful bytes. LZP requires more bits to encode those bytes than LZ77 would have done in that situation. But overall, LZP typically compresses better than LZ77.

A few implementations make sure the context table always points exactly to the most recent occurrence of every 3 byte context -- this requires updating the context table on every byte the decompressor emits. Many LZP implementations approximate this by only updating the table at the start of a phrase or literal, skipping the updates for all the bytes copied during that phrase. That speeds up decompression, and Arturo San Emeterio Campos[36] claims it actually improves the compression ratio on longer files, although it hurts the compression ratio on shorter files.

A LZP decoder works something like this:

- Look up the current context in the context table to get the Position. If this context has *never* occurred before, then we have a literal. Otherwise, fetch a "literal/copy" bit from the compressed file.
- (In either case, remember that Position, but update the context table to point to the current location in the decompressed text. The next time we have the same context, we get this location, rather than that less recent Position)
- 0: literal: the decoder grabs the next byte from the compressed file and passes it straight through to the decompressed text.
- 1: copy item: the decoder grabs a few bits from the compressed file and decodes them into a length. Starting at the Position where this context occurred before, copy "length" characters from that previously-decoded text to end of the decoded text. Then grabs the next literal byte from the compressed file and passes it straight through to the decompressed text.
- Repeat from the beginning until there is no more items in the compressed file.

Some documents[37] describe a byte-oriented version of LZP:

- Look up the current context in the context table to get the Position. (Remember that Position, but update the context table to point to the current location in the decompressed text. The next time we have the same context, we get this location, rather than that less recent Position).
- Fetch the next byte from the compressed text.
- If this context has *never* occurred before, *or* the fetched byte is not the escape byte, then we have a literal; otherwise, we have a copy item. (The compressor should have picked a value for the escape byte that occurs rarely in the plain text).
- literal: pass it straight through to the decompressed text.
- copy item: Discard the escape byte, and grab the length byte. Starting at the Position where this context occurred before, copy "length" characters from that previously-decoded text to end of the decoded text. Then grab the next literal byte from the compressed file and pass it straight through to the decompressed text, even if it happens to be the escape byte. (Occasionally the length will be 0, such as when the decompressed text *needs* to contain the special byte we use as the escape byte).
- Repeat from the beginning until there are no more items in the compressed file.

7.7.10 Statistical Lempel Ziv

Statistical Lempel-Ziv was first proposed by Yu Fan Ho as the research topic of his Master degree in Computer Science, supervised by Dr. Sam Kwong. They first published it as an application to personal digital assistants.[38] Later Yu Fan Ho applied to it ring tones for mobile phones, where it outperformed other available lossless compressors on compression ratio and decompression speed. The

application of statistical Lempel Ziv to melody data is patented.[39]

7.8 ACB

Associative Coding by Buyanovsky (ACB) starts with an LZ77-like compressor, and does many things to squeeze out a lot of the redundancy in the (offset, length) items. [40]

7.9 Tunstall code

The Huffman algorithm gives a fixed-to-variable-length code that (under certain constraints) is optimal.

The Tunstall algorithm gives a variable-to-fixed-length code that (under certain constraints) is optimal.[41][42][43]

7.10 Implementation tips and tricks

7.10.1 compressed format tips and tricks

Most data compression decompression algorithms are most naturally described in terms of reading the compressed file as an undifferentiated stream of bits (a bitstream). Most programming languages do file I/O a byte at a time.

So decompression implementations typically keep a separate “bit buffer” (initially empty). When you want to read a bit, if the bit-buffer is empty, read a (byte-aligned) byte from the compressed bytestream, store the last not-yet-needed 7 bits in the bit-buffer, and use the first bit. Otherwise read the next bit from the bit-buffer.

Many file formats are designed to maintain “8 bit alignment” (some even try to maintain “16 bit alignment”) wherever possible, in order to speed things up by reducing the amount of bit-shifting.

Often when decompressing, you've read some odd number of bits, and the abstract decompression algorithm mentions something like “read the next 8 bits and pass that literal byte directly to the output”. There are three possible ways to design a file format to handle “read the next 8 bits":

- “undifferentiated bitstream": get the remaining bits in the bit buffer, reload the bit buffer, and get a few more bits, then shift-and-align to re-assemble the byte.
- Throw away the remaining “padding bits” in the bit buffer, and simply read the next (byte-aligned) byte from the compressed bytestream. Faster than the bitstream approach, but the unused bits hurt the compression ratio, and so we will speak no more of this approach.
- “byte aligned": Temporarily ignore the bit-buffer; read the (byte-aligned) byte directly from the compressed bytestream. (Only use the bit-buffer when the decompression algorithm reads a bit or two or seven).

Compared to an undifferentiated bitstream format, a byte-aligned format for compressed text is precisely the same number of bytes long and has exactly the same bits; the bits are merely arranged in a slightly different order. So picking a byte-aligned format gives precisely the same compression ratio, and it often makes things faster.

7.10.2 compressor implementation tips and tricks

hash tables

The obvious way to implement a LZ77-style (offset, length) compressor involves scanning the entire window for a match to the following plaintext, for each and every phrase we eventually write into the compressed file. Then picking the longest (and nearest) match.

The obvious way to implement a LZ78-style (dictionary entry) compressor involves scanning the entire dictionary for a match to the following plaintext, for each and every dictionary index we eventually write into the compressed file.

For a file of length N this is O(N) for each compressed phrase. This requires O(N^2) to compress a file of length N. (Schlemiel the Painter’s algorithm). (I'm assuming the LZ77 window covers at least half of the file, and assuming a roughly constant compression ratio). This is excruciatingly slow.

Most LZ77-style and LZ78-style compressor implementations use a speedup trick: Whenever they output a compressed phrase, they do a little O(1) “extra” work updating a hash table. Then, instead of sequentially scanning the entire window for a match, they look up the following plaintext (typically the next 3 bytes) in the hash table. In a hash table, searches immediately succeed or fail in O(1), and therefore it only takes O(N) to compress the entire file. This is far faster.

Many LZ77-style compressors deliberately sacrifice the maximum possible compression in two ways: (a) Some compressors only update the hash table once per phrase, rather than once per every byte. (b) Some compressors use a direct-indexed cache -- a very simple hash table without collision detection -- the table remembers only the single most-recent item with that hash index -- rather than a standard hash table that remembers *every* item (so far) with that hash index. Typically each of (a) or (b)

hurts compression by a few percent, but can roughly double or triple the speed of the compressor. Many LZ77 style compressors do both (a) and (b), hurting compression by a few percent, but getting roughly four or ten times the speed.

pseudo-RLE

RLE can be emulated with overlapping copies in most LZ77-like decompressors ("pseudo-RLE mode"). The decompressor must use a method that handles such overlapping copies properly -- in particular, the decompressor must not use memcpy() or memmove() on such overlapping copies. On some machines, it's faster to simply use the same "copy one byte at a time" copy_bytes() routine for all copies. On other machines, programmers find that memcpy() is so much faster than that it reduces the total decompression time if the decompressor includes extra code that, for each copy item, decides whether it overlaps or not, and uses the slower copy_bytes() only when memcpy() won't work.

7.11 References

[1] J. Pike. "Text compression using a 4 bit coding scheme". 1981.

[2] https://github.com/antirez/smaz#readme

[3] Tomáš Kuthan and Jan Lánský "Genetic algorithms in syllable based text compression". 2007.

[4] LZJB source code

[5] Jacob Ziv and Abraham Lempel; *A Universal Algorithm for Sequential Data Compression*, IEEE Transactions on Information Theory, 23(3), pp.337-343, May 1977.

[6] "Better Archiver with Recursive Functionality (BARF)" by Matt Mahoney

[7] "replacing liblzf with FastLZ"

[8] Hadoop: "Implement FastLZCodec for fastlz/lzo algorithm"

[9] Wikipedia: Lempel–Ziv–Oberhumer

[10] "LZ4 - Extremely fast compression" versions of lz4 in many programming languages hosted at: https://github.com/Cyan4973/lz4 (was once hosted at https://code.google.com/p/lz4/).

[11] "LZ4c Discussions on LZ4 compression algorithm & open source implementations".

[12] 0 LZ4 Compressed block format.

[13] Yann Collet. "LZ4 explained".

[14] "LZ4-HC : High Compression LZ4 version is now Open Sourced".

[15] jpountz. "LZ4 compression for Java". quote: "two compression methods that both generate a valid LZ4 stream: ... fast scan (LZ4) ... high compression (LZ4 HC)"

[16] http://www.quicklz.com/

[17] http://www.ietf.org/ietf-ftp/IPR/hifn-ipr-draft-friend-tls-lzs-compression.txt

[18] Snappy in C++

[19] Snappy in pure Java

[20] PalmDoc Algorithm

[21] RedComet. "Dual-Tile Encoding: NES/Famicom Implementation". "Dual-Tile Encoding (commonly referred to as DTE in w:ROM hacking#Hex editing romhacking circles)"

[22] "short data compression"

[23] "Pattern-Based String Compression Function" by Frank Cox 2010

[24] "Alice's bubbles (Part 2)"

[25] stackoverflow: optimizing byte-pair encoding

[26] Wikipedia: byte pair encoding

[27] Carl Kindman. "SCZ - Simple Compression Utilities and Library".

[28] "Lempel-Ziv-Welch (LZW) Encoding: Discussion and Implementation" by Michael Dipperstein

[29] Wikipedia: Lempel–Ziv–Welch

[30] Wikipedia: LZWL

[31] Katsiaryna Chernik, Jan Lánský, and Leo Galamboš. "Syllable-based Compression for XML Documents".

[32] "Constructing Word-Based Text Compression Algorithms (1992)" by Nigel Horspool and Gordon Cormack.

[33] Juha Nieminen. "An efficient LZW implementation: Fast LZW dictionary search". 2007.

[34] "LZRW4: ZIV AND LEMPEL MEET MARKOV" by Ross Williams 1991

[35] "Anatomy of ROLZ data archiver"

[36] "Lzp" by Arturo San Emeterio Campos 1999

[37] http://mattmahoney.net/dc/text.html

[38] "A Statistical Lempel-Ziv compression algorithm for personal digital assistant (PDA)". IEEE Transactions on Consumer Electronics. 2001-Feb.

[39] Ho; Yu Fan. United States Patent 7,507,897. "Dictionary-based compression of melody data and compressor/decompressor for the same".

[40] Encode: "ACB" discussion

[41] Michael Drmota, Yuriy A. Reznik, and Wojciech Szpankowski. “Tunstall Code, Khodak Variations, and Random Walks”. 2010. “Tunstall Code, Khodak Variations, and Random Walks” 2008.

[42] David Salomon. [http://books.google.com/books?id=ujnQogzx_2EC&pg=PA61&lpg=PA61&dq=tunstall+code&source=bl&ots=FolApG2roS&sig=J0d3UB0Y5xbikfzE9fYtLjBB3-E&hl=en#v=onepage&q=tunstall%20code&f=false “Data Compression. 2.4 Tunstall Code"] p. 61. 2007.

[43] “Signal Compression: Variable to Fix Encoding”

Chapter 8

grammar-based compression

8.1 grammar-based compression

Grammar-based compression was proposed in 2000.[1]

Several data compression methods can be viewed as grammar-based compression:

- The Sequitur data compression algorithm uses a context-free grammar to represent a string such as a data file. It is relatively fast (linear-time encoding and linear-time decoding).[2]
- Longest Match
- Most frequent digram
- The Sequential data compression algorithm is an improvement on Sequitur.[3]
- Although it pre-dates grammar-based compression, LZ78 and LZW (but not LZ77) can be interpreted as a kind of grammar.[3]
- It has been shown that "a structured grammar" gives better compression than "an irreducible grammar".[4]

When the output of the grammar transform is compressed with a zero-order entropy coder (such as a zero-order arithmetic coder), grammar compression outperforms the Unix Compress and Gzip algorithms.[1]

The Re-Pair algorithm is a grammar based compression algorithm. Its decoder operates as follows:[5]

FIXME: fill in details.

8.2 Further reading

[1] En-hui Yang and John C. Kieffer. "Efficient Universal Lossless Data Compression Algorithms Based on a Greedy Sequential Grammar Transform—Part One: Without Context Models". 2000.

[2] Richard Ladner. "Sequitur" p. 56. 2007.

[3] Eric Lehman and Abhi Shelat. "Approximation Algorithms for Grammar-Based Compression". "Approximation Algorithms for Grammar-Based Compression". 2002.

[4] John Kieffer and En-hui Yang. "Structured grammar-based codes for universal lossless data compression". 2002.

[5] Matej Mežik. "Compression Using Context Free Grammers". 2011.

- Re-Store is a phrase browsing system based on the Re-Pair compression algorithm.

Chapter 9

Entropy

9.1 Entropy

Entropy is a measure of unpredictability. Understanding entropy not only helps you understand data compression, but can also help you choose good passwords and avoid easily-guessed passwords.

For example, 1000 bits of data representing 1000 consecutive tosses of a fair coin has an entropy of 1000 bits, since there is no way to predict whether “heads” or “tails” will come up next. 1000 bits of data representing 1000 consecutive tosses of an unfair two-headed coin has an entropy of zero (0) bits, since the coin will always come up heads.

The entropy of a message is in a certain sense a measure of how much information it really contains.[1]

By taking the consistencies out of a storage element, such as the redundancies, similar components, longest most used words, etc., Compression increases the entropy of the file, until there is so much entropy in the file that it can't be further compressed. Compression when it shrinks a file, can lead to a higher entropy density per length of file. Which makes a properly compressed file when followed by an encryption pass harder to break in ciphertext only attacks than the file encrypted without the compression done first. See:

http://en.wikipedia.org/wiki/Unicity_distance

Of course it does not help with most plain text attack modes if attacker gets to pick the plain text you encrypt.

At the same time, the decompression algorythm adds more and more consistency and order to the entropic file, until it means something. One of the reasons that compression does not make a good form of encryption, is that there is information hidden in the entropy of the file, that indicates to the decompression program how to recover it. In fact in extreme compression the amount of data needed to recover the compressed text, is often a greater percentage of the file, than the actual remaining data that has yet to be compressed.

Entropy is always relative to some model. It is not a property of a particular string of bits that you are given, but of the set of bitstrings that you could plausibly have been given. For a given set of bits, the model that gives the lowest entropy is the model that best predicts those bits and therefore compresses the data the smallest. Most compressed file formats, store (a) what model we used, (b) the particular parameters of that model (if any), and (c) the data compressed according to that model. Some kinds of models have a lot of “parameters” and require a lot of space to store, so often we use completely unrealistic models -- but ones that require very little space to describe -- in order to reduce the net compressed file size (model description + compressed data).

One of the the simplest models is “Every byte of the file was independently and randomly generated by throwing darts at a wall while blindfolded. Some letters occur more frequently because they have a larger area on the wall.” This is the order-0 Markov model, also known as the Shannon entropy when each byte is considered an independent message. Using that model, the entropy $H(F_i)$ of any particular letter F_i in a file F is

$$H(F_i) = -log_2 p(F_i),$$

(This is the number of bits required to represent that letter using an entropy coder)

And the entropy $H(F)$ of the entire file is the sum of the entropy of each letter in the file, (the number of bits required to represent the entire file is the sum of the number of bits required to represent each letter in that file)

$$H(F) = \sum_{i=1}^{N} -\log_2 p(F_i),$$

In terms of the number of unique possible messages n (any particular letter in the file is one of a list n possible letters, from $x_1..x_n$, any of which may occur 0, 1, or possibly N times)

$$H(F) = \sum_{i=1}^{n} -p(x_i) \log_2 p(x_i),$$

Where

- F_i is some unique letter, for example 'e'

* $f(x_i)$ is the number of times that unique letter x_i occurs in the file

- $p(x_i)$ is the probability of the symbol x_i (e.g. the character 'e') in the message
- $f(F_i)$ is the number of times the letter at position i in the file occurs in the file (obviously this is at least 1)
- N is the length of the file (the total number of messages in the file)
- n is the number of possible messages, often n=257
- $p(k) = \frac{f(k)}{N}$ is the probability that some letter k occurs.

Often some particular letter x_i in the character set never occurs anywhere in some particular file -- the way the letter 'e' never occurs in *Gadsby* (1939). There can be any number of such letters that never actually occur. Letters that never occur make no difference to the entropy of any particular letter that does occur, or to the total of them, which is the entropy of the entire file under the order-0 model. When some letter x_i exists in the character set, but it is never actually used in some particular file, that letter's frequency is zero, its probability is zero, and the value $0log_2(0)$ is effectively zero.

Any arbitrary file, when analyzed using a order-1 Markov model ("first order model"), will give a number for the entropy that is smaller or the same as the order-0 entropy.

We discuss order-1 Markov and higher-order Markov models in more detail later in this book (Markov models).

9.1.1 How do you further compress data

Compression at some point becomes how do you further compress already entropic data, that consists mostly of instructions to decompress a minimal amount of data?

There are two approaches that might be tried, the first approach is to inject order, without increasing the length of the file, so that you can recover more compression, and the other approach is much more controversial because it suggests the injection of entropy, without changing the length of the file, so that you can recover more compression.

The latter approach is based on a little known scientific principle first explained in "A New Kind of Science", which suggested that entropy and order are cyclical at least in incremental or decremental type IV cellular automata. This approach which might explain self-organization despite the Second Law of Thermodynamics would mean that sometimes when enough entropy is injected into a data field, the data self-organizes into an orderly state again. If we can keep track of the entropic states that it has gone through to get to the self-organization level, we can compress the new order, with some hope of recovering the original state.

9.2 entropy coding

statistical symbol compression

prefix coding

9.2.1 Huffman encoding

For each prefix code in the compressed data, a Huffman decompressor looks up the compressed symbol in a table and emits the corresponding plaintext symbol.

In the following examples, we assume the plaintext symbol is some 8 bit byte. (Most Huffman decompressors have 257 symbols in their alphabet: the 256 possible bytes, and a special 257th code that indicates "end-of-block". Some Huffman decompressors have additional compressed symbols that decompress to common byte pairs or entire words, which can lead to smaller compressed file sizes). [2][3]

Where does the translation table of code-to-letters come from? The particular source chosen is perhaps the biggest difference between one Huffman compressed file format and another.

- fixed Huffman codes
- static Huffman codes (sometimes called "semi-adaptive Huffman")[4] (The RFC 1951 definition of DEFLATE calls this "dynamic Huffman codes")
- adaptive Huffman codes

(Yes, it's confusing to use "static" as a synonym for "dynamic". Is it too late to change this now?)

When a decompressor uses a fixed Huffman code, it uses a translation table hard-wired into the decompressor that never changes.

When a decompressor uses a static Huffman code, it reads the translation table at the beginning of the compressed block. Then it uses that data to decode the rest of the block. Most of the "optimality proofs" of Huffman assume this sort of static code.

Often one block of text will have slightly different letter frequencies from another, even inside the same file. Most static Huffman decompressors watch for a special 257th "end-of-block" symbol, and then discard the old Huffman table and read in a new static compressed table. The decompressor uses that table to decode the new block.

DEFLATE and some other decompressors put a few bits of "model selector" metadata at the beginning of every block. That commands the decompressor to either

- (a) fetch the static Huffman table metadata immediately after the model selector;
- (b) copy the default fixed Huffman table stored inside the decompressor;
- (c) copy the raw data without trying to decompress it -- useful for "uncompressible" data; or
- (d) use some other compression algorithm entirely.

When a decompressor uses an adaptive Huffman code, it begins with some default translation table. However, it updates that table with every symbol that goes through.

9.2.2 getting the compressor and the decompressor to use the same codewords

To get lossless compression, when the compressor represents the byte "e" with the bitstring "010", the decompressor has to somehow know that the bitstring "010" must be decoded into the byte "e".

There are several ways for a Huffman decompressor to "know" the symbol-to-bitstring map (sometimes called a "frequency table") used by the corresponding Huffman compressor:

- fixed Huffman codes, hard-wired into the decompressor. Avoids sending a frequency table, but can give poor compression -- perhaps even expanding the file -- if the actual probability distribution is significantly different than the assumed probability distribution. The estimated probability distribution changes only when the compressor software changes. This has the lowest latency -- the first few symbols can be immediately encoded and transmitted.
- one dynamic Huffman code, with a single frequency table sent before the file. This requires the compressor to store the entire file, which may be arbitrarily long, while counting symbol frequencies, before generating and sending the first compressed codeword -- this has the worst latency. The estimated probability distribution changes once per file.
- several dynamic Huffman codes, with the file broken up into several blocks and a frequency table sent before each block. This has less latency than one-table-per-file. In cases where the probability distribution changes significantly from one block to the next, the total number of bits needed to store all the codewords can be much less -- but the overhead of the number of bits to send the tables can be much more. Also, the compressor can spend a lot of time trying to squeeze out a few more bits by "optimizing" where exactly to split the file. The estimated probability distribution changes once per block.
    - A few file formats (such as bzip2), rather than always sending a complete frequency table that indicates the exact frequency of the next block, sometimes send a short reference to one of the previously-sent frequency tables that (hopefully) is close enough to the frequency of the next block that the expansion (if any) of the codewords in that block is more than paid back by not having to send a complete frequency table.
- adaptive Huffman compression: The estimated probability distribution changes (slightly) once per symbol. The probability distribution of the letters that have streamed past is used to estimate the future probability distribution. This has the second-lowest latency -- the first few symbols can be more-or-less immediately transmitted -- but the decompressor does more work to update the tables on each symbol.

Per-file and per-block dynamic Huffman codes can guarantee that certain symbols never occur in a block, and so some people write code that entirely eliminates those never-occurs symbols from the tree, reducing the number of bits in at least one codeword that does occur in the compressed text.

Many compression algorithms (including fixed Huffman codes and adaptive Huffman compression) estimate probabilities in a way that cannot guarantee that some symbol will never occur. In particular, if some symbol is so rare that it has not yet occurred in the training text -- it has zero frequency so far -- it is tempting to assign an estimated zero probability to that symbol. That leads to The zero-frequency problem.

Canonical Huffman code

A canonical Huffman code is a kind of Huffman code that can be very compactly described. That makes it super-wonderful for static Huffman compression. A large text file may have thousands of sections, each with a slightly different letter frequency. Every unnecessary byte in the format we choose for storing the description of the Huffman code table bloats that file by thousands of unnecessary bytes.

For example, the article on x-rays has a much higher frequency of the letter x than the previous article on yellow-bellied sapsuckers. Giving each section its own optimized Huffman table minimizes the number of bits required to store that section, if we pretend that the Huffman ta-

ble overhead is not significant. Alas, that Huffman table overhead is often significant. Many compressors will spend a lot of time thinking about whether the total file will be shorter if we (a) use a separate Huffman table for each of several consecutive sections or (b) merge consecutive sections together into a single block and use a single compromise Huffman table for the whole block.

"drop some trailing ones, and then increment"

The decompressor needs two things to regenerate a block of plaintext: the exact Huffman code used by the compressor, and the series of compressed codewords.

Given any one Huffman code, one can construct many other equivalent codes, all of which have exactly the same codeword length for any particular plaintext symbol, but vary in the exact bit pattern assigned to each plaintext symbol. Since any particular plaintext symbol is assigned a codeword of a certain length, a length that is the same for every equivalent code, no matter which particular code you choose, any series of plaintext symbols will be compressed into a series of codewords of exactly the same total length. If the overhead of the prefix code description were insignificant, then it would not matter which of these many equivalent codes you would use -- just pick one at random. All equivalent codes compress that data to the same number of bits.

However, rather than picking one of them randomly and sending that to the decompressor, we can save bits in the prefix code description by picking one particular canonical Huffman code.

Given only the length, in bits, of the codeword for each symbol of any prefix code, we can construct the canonical Huffman code that gives every symbol the same length as that original prefix code.[5][6][7][8][9] What makes a code a canonical Huffman (CH) code is that code words are lexicographically ordered by their code lengths;[10] A given set of probabilities does not result in a unique CH code; one is free to reorder symbols that have the same code length.[11]

The steps to create a canonical Huffman (CH) code are as follows:

1. Construct a Huffman tree and throw out all information other than the symbols and the lengths of the codewords.
2. Sort the symbols by increasing code lengths.
3. Store the number of symbols with each code length along with the sorted symbols themselves.

9.3 Something about trees

Every prefix code can be thought about as analogous to a tree.

Every leaf on the tree is tagged with a plaintext letter or other symbol.

The length, in bits, of each codeword is sometimes called the path length of the symbol.

9.4 Dynamic Huffman: storing the frequency table

(This is a continuation of "getting the compressor and the decompressor to use the same codewords", for the common case of dynamic Huffman).

Inside a typical compressed file, there is a header before each block of compressed data. For blocks compressed with dynamic Huffman, that header for that block tells the decompressor the set of codewords used in that block, and the mapping from codewords to plaintext symbols.

Typically the compressor and the decompressor have some hard-coded alphabet of S plaintext symbols they need to handle. Often S=258 symbols, the 256 possible bytes and a few other special symbols such as the end-of-block symbol. Each symbol in the alphabet is represented, in the compressed data, by some codeword with a length in bits from 1 bit to some MAX_CODEWORD_LENGTH bits.

Many simple Huffman file formats simply store that list of 258 bitlengths as a block of 258 bytes. Each position represents some plaintext symbol -- even ones that are never used in the plaintext -- typically position 0 represents the 0 byte, position 32 (0x20) represents the space character, position __ (0x65) represents letter 'e', position 255 represents the 255 byte, position 256 represents some special command, and position 257 represents the end-of-block indicator.

The number at each position represents the length, in bits, of the codeword corresponding to that plaintext symbol. (Since we use canonical Huffman codes, the length of the codeword corresponding to each plaintext symbol is all the decompressor need to regenerate the actual set of codewords used in this block). Usually the special "length" of "0 bits" is reserved to indicate letters that never occur in this block, and so should not be included in the Huffman tree.

That list of S numbers is often called the "frequency table". Sometimes that list of S numbers is called the "Kraft vector", because it must satisfy w: Kraft's inequality.

That block of 258 bytes in the header represents a lot of overhead, especially if the compressed file includes many separate blocks, each with its own header. Some Huffman compressors, such as English-word-oriented Huffman compressors, use vastly more symbols (one for each word in the English dictionary it uses), requiring a much longer Kraft vector. There are a variety of ways to get smaller compressed file sizes by designing the file format

in a way that reduces the size of this block, including:

- Use exactly the same codeword-to-plaintext-symbol dynamic Huffman mapping, but store that mapping in a more compact form:
  - use some kind of length-limited Huffman compression. If there are no codewords longer than 15 bits, then we can store each length in 4 bits, rather than a full 8 bit byte.
  - compress the frequency table in isolation, perhaps using RLE or some variable-length code, such as fixed Huffman compression or even another dynamic Huffman "pre-table".
  - When a datafile alternates between two kinds of data -- for example, English text and long tables of numbers -- use simple differential compression to compactly represent later tables as a reference to some earlier table, as in bzip2.
  - When a one block of data has slightly different frequencies than the previous block -- for example, an article on X-rays has about the same letter frequencies, except much more 'X' and somewhat fewer 'y', than a previous article on yellow-bellied sap-suckers -- use differential compression to compactly represent the new table by giving the (hopefully relatively small) changes from the previous table, as in LZX. We discuss other kinds of differential compression in a later chapter, data differencing.
  - or some combination of the above.
- Use some other codeword-to-plaintext-symbol mapping other than dynamic Huffman mapping
  - adaptive Huffman: estimate the future letter frequency on-the-fly from the letters already decoded in the plaintext
  - universal codes: ...

Implementation tips and tricks

People have spent a long time figuring out ways to make Huffman compression and Huffman decompression really, really fast. These are just a few of the techniques used to make decompressors (or compressors, or both) that give bit-identical the same output that a non-optimized program gives, while running faster.

Software to implement Huffman decompressors typically has 2 parts:

- the setup pulls a compact "frequency table" header, and expands it into some sort of lookup table arranged to make the next part really fast
- the inner loop that converts codewords to plaintext symbols.

The inner loop of a Huffman decompressor finds the codeword that matches the next part of the compressed text, and emits the corresponding plaintext symbol. It's possible for a codeword to be more than 32 bits long, even when compressing 7-bit ASCII text, and bit-aligning and comparing such long words can be a bit of a hassle. There are three approaches to avoiding such long comparisons: (a) length-limited Huffman code. Often "natural" files never use such long codewords, only specially-constructed specifically designed to be worst-case files. Some compressors choose to deliberately limit the length of the Huffman code words to some maximum length. This makes such worst-case files, after compression, slightly longer than they would have been without the limit; it has no effect on the length of compressed "natural" files. Most file formats intended to be decompressed in hardware (video decompressed with a FPGA, etc.) limit the length in order to reduce the cost of that hardware. (b) A short, fast table that handles short codewords and has a "long codeword" flag, and other table(s) that handle longer codewords. Since the most frequent Huffman codes are short, the decoder can usually decode a codeword quickly with one hit to the short, fast table that fits entirely in cache; occasionally the decoder would hit a flagged entry, and take a little longer to look up the long code in the other table(s). There's a time/space tradeoff on exactly how short to make the first-level table: a decoder program with a smaller first-level table, compared to a decoder program that uses a larger first-level table, can still correctly decode the same file. The smaller-table decoder uses less RAM, the larger-table decoder decodes more words in the first hit and therefore runs faster (unless the table gets too big to fit in the cache). (c) rather than comparing directly to codeword bitstrings, first do a zero-count (to the next 1 bit, or to the end of file, whichever comes first) finding Z zeros, discard the 1 bit, and then the next n bits.

Prefix code decompressor implementations can be distinguished by how many bits they read from the compressed stream at one time, and how they use those bits to decide what to do next. People have implemented "fast" decompressors a variety of ways, including:

- One bit at a time: A "simple" prefix decoder typically reads bit-by-bit, one bit at a time, from the compressed stream. This is probably the slowest way.[12]
- always max_codelength bits at a time: one big lookup table indexed directly by multiple bits read from the compressed stream
- one byte (8 bits) at a time: one big lookup table indexed directly by a "tree position" state and a byte read from the compressed stream

- X initial bits, followed by Y following bits, per symbol (where X is some constant, often chosen to be half the number of bits as max_codelength[12], and Y typically varies depending on the particular value read in the first X bits; X+Y is at most max_codelength):
    - a first lookup table indexed by some bits read from the compressed stream, which chooses one of the secondary index tables, indexed by more bits read from the compressed stream
- any number of zero bits, followed by F trailing bits: (This works for canonical Huffman codes, but may not be useful for other prefix codes).
    - one big lookup table indexed by a zero count and bits read from the compressed stream
    - a first lookup table indexed by a zero count of bits from the compressed stream, which chooses one of the secondary index tables, indexed by more bits read from the compressed stream.

Conceptually the simplest one is the "one big lookup table" approach:

- one big lookup table

This is probably the fastest technique if your maximum codelength is relatively short.[12]

One fast inner loop for a fast prefix code decoder reads the compressed data a byte at a time (rather than the bit-by-bit for a "simple" prefix decoder). The "current tree position" and the compressed byte are used as an index into a large table that holds the resulting "new tree position" and up to 8 decompressed plaintext symbols.[13]

- (new_tree_position, [list of decompressed symbols]) := lookup_table(current_tree_position, compressed_byte)

Any codeword can be represented by two numbers, (Z count of initial zeros, F value of trailing bits). For example, the codeword "000101" can be represented by the two values (3,5). The codeword "11" can be represented by the two values (0,3).

One quirk of the canonical Huffman code is that, if all our plaintext symbols fit in a n bit fixed-length code, then Z can be represented in n bits, and F can also be stored in n bits. (This is *not* true for all prefix codes). (When the decompressor does the actual zero-count in the compressed text, consecutive all-zero code words can lead to an indefinite number of consecutive zeros -- what I'm trying to point out here is that the number of initial zeros in any one particular codeword can be represented in n bits). This allows several different time/speed tradeoffs that all give identically the same output file. Perhaps the fastest inner loop is something like For each symbol:

- bit-count the zeros into Z
- discard the following 1 bit, keep the "next bit to read" pointer pointing to the *following* bit, but peek ahead at the following n-1 bits and put them into F.
- Pack the least-significant n bits of Z and the n-1 bits of F into a 2n-1 bit index.
- (plaintext_symbol, useful_bits) := lookup_table[index]
- Bump the "next bit to read" pointer by useful_bits bits -- which can be 0 bits for codewords with only one bit after the zeros (1, 01, 001, 0001, etc.) up to n-1 bits for codewords with the full n-1 bits after the first 1 bit in the codeword.
- emit the plaintext symbol.

Repeat until we hit the end of file.

This ignores the special-case handling of all-zeros codeword. Many Huffman coders, such as DEFLATE, design a Huffman table with a "fake" entry such that the compressed text never uses the all-zeros codeword; so the decompressor never has to deal with that complication. This also ignores special handling of codewords that map to some special command, rather than mapping to a literal plaintext symbol. This requires a lookup table of size 2^(2n-1). For a typical Huffman code with n=9 bits, able to handle all 2^8 bytes a few other special symbols, using a canonical Huffman code with this kind of implementation requires a lookup table of size 2^(17) = 131 072 entries. If a plaintext symbol is mapped to a codelength that is less than the maximum codelength, then multiple entries in that table map to the same symbol. There are other implementations that use far less RAM. Other implementations may even be faster, since they allow more of the working memory to fit in cache.

In general, for *alphabet_size* plaintext symbols, the worst-case maximum "compressed" Huffman code length is *alphabet_size - 1* bits long.[12]

For 257 plaintext symbols, the worst (unlikely) case is a maximum bitlength of 2 characters that are "compressed" to 256 bits, so the maximum zero count (except when we have 1 or more consecutive all-zero codes) is 255.

Handling the end-of-text is often awkward, since the compressed string of bits is usually stored into some integer number of bytes, even though the compressed string of bits is usually *not* a multiple of 8 bits long. Most Huffman decompressors use one or the other of the following techniques for handling end-of-text:

- Both the compressor and the decompressor include a special end-of-text symbol in the tree (a 257th symbol when doing binary byte-oriented compression, or the null byte when compressing a null-terminated C string). Since it only occurs once,

its frequency is 1 and the code that represents it is therefore is tied in length with one or more longest-possible-length symbols.

  - The decompressor immediately returns when it sees the end-of-text symbol, and ignores any pad bits after that symbol that fill up the last byte of storage. Or,
  - The compressor tells the decompressor exactly how many valid input bytes are used to hold the compressed bitstring. The decompressor forces the code representing the end-of-text symbol to be the all-zeros symbol. The decompressor fakes reading in zero bits past the last valid input byte -- possibly reading a few zero bits at the end of some other valid symbol -- eventually reading the all-zero end-of-text symbol.

- Some decompressors have no special end-of-text symbol.
  - The compressor tells the decompressor exactly how many output symbols are represented by the input compressed bitstring, and the decompressor immediately returns after emitting exactly that many symbols. Or,
  - The compressor tells the decompressor exactly how many valid input bytes are used to hold the input compressed bitstring. Occasionally the last valid symbol ends cleanly at the end of the last byte, and the decompressor returns. Other times there are a few leftover bits not long enough to make a complete valid codeword -- all codewords with that prefix are longer. When the leftover bits are all zero, throw away those leftover zero bits and return. (This technique *assumes* that the all-zeros code is at least 8 bits long -- which is always true when there are at least 2^7+1 = 129 symbols in the tree and you use a canonical Huffman code. If you have fewer than 129 symbols, you must use some other technique to handle end-of-text).

9.4.1 improving on Huffman

The original Huffman algorithm is optimal for symbol-by-symbol coding of a stream of unrelated symbols with a known probability distribution.

However, several algorithms can be used to compress some kinds of files smaller than the Huffman algorithm. These algorithms exploit one or another of the caveats in the Huffman optimality proof:

- unrelated symbols: most real data files have some mutual information between symbols. One can do better than plain Huffman by "decorrelating" the symbols, and then using the Huffman algorithm on these decorrelated symbols.
- symbol-by-symbol: Several compression algorithms, such as range coding, give better compression than binary Huffman by relaxing the binary Huffman restriction that each input symbol must be encoded as an integer number of bits.
- known probability distribution: Usually the receiver does not know the exact probability distribution. So typical Huffman compression algorithms send a frequency table first, then send the compressed data. Several "adaptive" compression algorithms, such as Polar tree coding, can get better compression than Huffman because they converge on the probability distribution, or adapt to a changing probability distribution, without ever explicitly sending a frequency table.

This is reminiscent of the the way many algorithms compress files smaller than the "optimum" Fixed Bit Length Encoding codeword length[14] by exploiting the caveat in the Fixed Bit Length Encoding optimality proof:

Fixed Bit Length Encoding is optimal for symbol-by-symbol coding into fixed-length codewords of a stream of symbols with a known finite alphabet size A. The optimum codeword length for Fixed Bit Length Encoding is $L = \lceil log_2 A \rceil$.

- finite alphabet: Fixed Bit Length Encoding is useless for infinite alphabets; "universal codes" (described below) handle such alphabets easily.
- symbol-by-symbol: Several compression algorithms, such as LZW, have fixed-length codewords with better compression than Fixed Bit Length Encoding, by using some codewords to represent symbol pairs or longer strings of symbols.
- fixed-length codewords: variable-length codes often give better compression.

9.4.2 arithmetic coding and range coding

9.5 prefix codes

Huffman coding, Shannon-Fano coding, and Polar tree coding all map some relatively small known alphabet of symbols to prefix codes. [15] [16]

The "universal codes" map an infinitely large known alphabet -- often the set of all positive integers, or the set of all integers -- to prefix codes.

9.5.1 universal codes

There are many "universal codes", a kind of prefix code.

Any particular prefix code (including any universal code) can be seen as a kind of Huffman code for a particular corresponding frequency distribution.

Typical audio data, and typical audio data prediction residues, have approximately Laplacian distribution. Prediction error coding for image compression and geometry compression typically produces an approximately Laplacian distribution.[13] Encoding a Laplacian distribution with a general Huffman code gives a prefix code equivalent to using a Golomb code. The Golomb code is much simpler and faster than a general Huffman code to compress and decompress. If the audio data is "close enough" to a Laplacian distribution, a Golomb code may give even better net compression than a general Huffman code, because a Golomb decoder requires a smaller overhead of 1 parameter, while general Huffman decoder requires the overhead of a Huffman table of probabilities.

Rice coding is a subset of Golomb coding that is much faster to compress and decompress. Because Rice codes closely approximate Huffman codes for a Laplacian distribution, they compress almost as well as general Huffman codes.

Todo: add a few words about Fibonacci codes, a kind of "universal code".[17]

9.6 audio compression

fourier transform

perceptual coding

"The Gerzon/Craven theorems show that the level of the optimally de-correlated signal is given by the average of the original signal spectrum when plotted as dB versus linear frequency. ... this dB average can have significantly less power than the original signal, hence the reduction in data rate. ... In practice, the degree to which any section of music data can be 'whitened' depends on the content and the complexity allowed in the prediction filter." [18]

Typical audio data, and typical audio data prediction residues (after decorrelating), have approximately Laplacian distribution, and so they are often decoded with a Golomb decoder or Rice decoder rather than a general Huffman decoder.

Some audio decoders can be told to use Rice decoders during passages of music that compress well with them, and later told to switch to some other decoder during passages of music (such as pure sine wave tones, or all values equally probable white noise) that don't compress well with Rice decoders.

In some audio applications, in particular 2-way telephone-like communication, end-to-end latency is critical. The recommended maximum time delay for telephone service is 150 milliseconds[*citation needed*](Wikipedia:1 E-1 s). This rules out many popular compression algorithms. We discuss latency in more detail in the Data_Compression/Evaluating_Compression_Effectiveness#latency section of this book.

9.7 arithmetic coding

arithmetic coding

Context-adaptive binary arithmetic coding

...

9.8 combination

see Data Compression/Multiple transformations

9.9 References

[1] Wikipedia: entropy (information theory). Retrieved 2011-07-29.

[2] "Constructing Word-Based Text Compression Algorithms (1992)" by Nigel Horspool and Gordon Cormack.

[3] Tomáš Kuthan and Jan Lánský "Genetic algorithms in syllable based text compression". 2007. "The number of unique syllables in one language is measured in tens of thousands ... HuffSyllable is a statistical syllable-based text compression method. This technique was inspired by the principles of HuffWord algorithm."

[4] "Hacking Data Compression" by Andy McFadden

[5] "Canonical Huffman" by Arturo San Emeterio Campos. 1999.

[6] Canonical Huffman Codes by Michael Schindler.

[7] "Huffman Code Discussion and Implementation", including Canonical Huffman Codes. by Michael Dipperstein.

[8] "Decoding of Canonical Huffman Codes with Look-Up Tables" by Yakov Nekritch. 2000.

[9] "Space- and time-efficient decoding with canonical Huffman trees" by Shmuel T. Klein 1997.

[10] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes*. New York: Van Nostrand Reinhold, 1994. ISBN 9780442018634. pp. 33-35.

[11] "Huffman encoding [DRAFT"] apparently by Lara Hopley and Jo van Schalkwyk (?). mentions canonical Huffman: "We will allocate '1's to shorter codes." mentions "even a canonical Huffman tree needn't be unique. We still have some scope for playing!" (describing situations

where, even though we assign *different* lengths to letters and therefore different canonical Huffman trees, we end up with the same number of compressed bits)

[12] Michael Schindler. "Practical Huffman coding"

[13] Renato Pajarola. "Fast Prefix Code Processing". "Proceedings IEEE ITCC Conference". 2003.

[14] NES development: Fixed Bit Length Encoding

[15] http://en.wikipedia.org/wiki/User:C-processor/Polar_Tree

[16] Andrew Polar. "Non-Huffman binary tree". July, 2010.

[17] Shmuel T. Klein, Miri Kopel Ben-Nissan. "On the Usefulness of Fibonacci Compression Codes". 2004. via

[18] "MLP Lossless Compression" by JR Stuart, PG Craven, MA Gerzon, MJ Law, RJ Wilson 1999. section 4.2 "Prediction".

Chapter 10

Lossy vs. Non-Lossy Compression

10.1 Lossy Compression vs Non-Lossy Compression

Lossy image compression and lossy video compression (such as JPEG compression, MPEG compression, and fractal image compression) give much better reduction in size (much higher compression ratio) than we find in almost any other area of data compression.

Lossy compression is best used to reduce the size of video data, where defects in the picture can be hidden as long as the general structure of the picture remains intact. This type of compression is called lossy compression because part of the reason that it compresses so well is that actual data from the video image, is lost and then replaced with some approximation.

Fractal compression is based on the concept of self-similarity in fractals. It uses fractal components that have self-similarity to the rest of the surrounding area in the picture.

The storage is so inaccurate that by replacing some of the fractal data with encrypted text, it is very difficult to see a difference in the picture, which means that important messages can be embedded in video images with little fear of detection. [*citation needed*]

If we could achieve the same level of compression without loss of data, we wouldn't use Lossy Compression because there is an imperceptible loss of quality in the image.

One of the reasons that there is still some interest in new loss-less compression techniques, is that only very inexact data structures can survive Lossy Compression. It is often the case that loss of a single bit, renders a whole phrase or line of data inaccurate. This is why we attempt to build more and more stable memory systems. The recent shift from RD to SD ram for instance was partially because RD ram needed more interactive maintenance of its data.

If we are so protective of the memory of data, then it makes sense that we must also be protective of the compression scheme we use to store and retrieve data. So the only places Lossy Compression can be used, are places where the accuracy at the bit level, does not materially affect the quality of the data.

10.2 image compression

Most image compression and video compression algorithms have 4 layers: *Is there a standard terminology for these things?*

- A "modeling" or "pre-conditioning"[1] or "filtering"[1] or "de-correlation" layer converts the raw pixels into more abstract information about the pictures -- and, at the decoder, a "rendering" part converts that abstract information back into the raw pixels.
- A "quantizer" layer that throws away some of the details in that abstract information that humans are unlikely to notice -- and, at the decoder, a "dequantizer" that restores an approximation of the information.
- A "entropy coder" layer that compresses and packs each piece of information into the bitstream to be transmitted -- and, at the decoder, the "entropy decoder" that accepts the bitstream and unpacks each piece of information.
- A layer that adds synchronization, interleaving, and error detection to the raw bitstream just before transmitting it.

Since we talk about entropy coding elsewhere in this book, and the Data Coding Theory book discusses synchronization and error detection, this section will focus on the other layers.

Some kinds of "modeling" process gives us information that is useful for things other than data compression, such as "de-noising" and "resolution enhancement".

Some popular "modeling" or "de-correlation" algorithms include:

- delta encoding
- Fourier transform -- calculated using the fast Fourier transform (FFT) -- in particular, the discrete cosine transform (DCT)

- wavelet transform -- calculated using a fast wavelet transform (FWT) -- in particular, some discrete wavelet transform (DWT)
- motion compensation
- matching pursuit
- fractal transform[2][3][4][5]

There is a huge amount of information in a raw, uncompressed movie. All movie and image compression algorithms can be categorized as either:

- Lossless methods: methods that make no changes whatsoever to the image; the uncompressed image is bit-for-bit identical to the original.
- "Nondegrading methods" or "transparent methods": methods that make some minor changes to the image; the uncompressed image is not exactly bit-for-bit identical, but the changes are (hopefully) invisible to the human eye.
- "Degrading methods" or "low-quality methods": methods that introduce visible changes to the image.

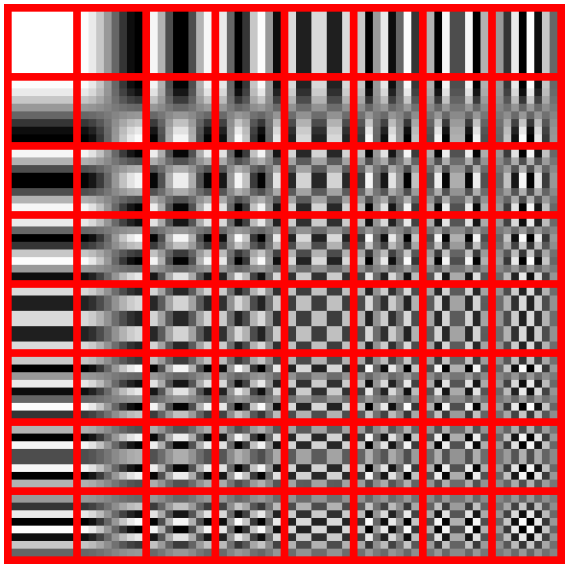
... "idempotent" one-time loss vs. "generational" loss at every iteration ...

In theory, a "perfectly" compressed file (of any kind) should have no remaining patterns. Many people working compression algorithms make pictures of the compressed data as if it were itself an image. [6] The human visual system is very sensitive to (some kinds of) patterns in data. If a human can describe a repeating pattern in the compressed data precisely enough, you can use that description to help model the original image more accurately, leading to a smaller compressed image file.

Alas, in situations where there is no pattern at all (true random data), the human eye often sees "clumps" and other "patterns" that are not useful for data compression. (Some artists take randomly-positioned data and carefully nudge the points to make them more evenly spaced to make them *look* random, but the result is actually less random and can often be compressed at least a little, unlike true random data).

10.2.1 DCT

Standard JPEG uses a (lossy) discrete cosine transform (DCT), and standard MP3 uses a (lossy) modified discrete cosine transform (MDCT). The difference between the original input and the decoded output caused by these standard transforms are smaller than the errors caused by quantization, which in turn are generally so small as to be imperceptible.



The two-dimensional "basis functions" of the JPEG DCT.

10.2.2 integer transforms

Several wavelet transforms use only integer coefficients, such as the ones used in Progressive Graphics File (PGF), lossless JPEG 2000, CineForm, ICER, etc.

A few DCT-like transforms have only integer coefficients, such as the MPEG integer IDCT standard,[7] the 4×4 and 8×8 integer transforms used in H.264, and the 4x4 transform used in JPEG XR.

Most of these transforms were originally developed so they would run quickly on embedded systems -- systems where the floating-point transforms used by standard JPEG/MPEG run too slowly.

In 2012, compression researcher Rock Brentwood posted several lossless approximations to the 1D DCT and IDCT, including the following:[8]

LDCT forward transform transformation matrix:

17 17 17 17 17 17 17 17 24 20 12 6 −6 −12 −20 −24
23 7 −7 −23 −23 −7 7 23 20 −6 −24 −12 12 24 6 −20
17 −17 −17 17 17 −17 −17 17 12 −24 6 20 −20 −6
24 −12 7 −23 23 −7 −7 23 −23 7 6 −12 20 −24 24
−20 12 −6

ILDCT reverse transform inverse transformation matrix is the transpose of the LDCT matrix:

17 24 23 20 17 12 7 6 17 20 7 −6 −17 −24 −23 −12
17 12 −7 −24 −17 6 23 20 17 6 −23 −12 17 20 −7
−24 17 −6 −23 12 17 −20 −7 24 17 −12 −7 24 −17
−6 23 −20 17 −20 7 6 −17 24 −23 12 17 −24 23 −20
17 −12 7 −6

10.2.3 JPEG

The JPEG data compression algorithm and file format are extensively covered in the wikibook JPEG - Idea and Practice.

10.2.4 JFIF: JPEG File Interchange Format

10.2.5 JPEG 2000

JPEG 2000 is useful in applications that require very high quality -- such as the DICOM medical image format -- because the compressor can choose higher quality settings -- including a completely lossless mode -- than are available in earlier JPEG standards.

JPEG 2000 is a image coding system based on wavelet technology. [9] The JPEG 2000 compression standard uses the biorthogonal CDF 5/3 wavelet (also called the LeGall 5/3 wavelet) for lossless compression and a CDF 9/7 wavelet for lossy compression. However, a variety of other kinds of wavelets have been proposed and used in experimental data compression algorithms. [10]

Open-source implementations of JPEG 2000 are available. [11]

10.3 color transformation

Most image compression algorithms use some kind of color space transformation to de-correlate the R, G, B values of each pixel. We discuss color transforms in Data_Compression/Multiple_transformations#color_transforms.

10.4 audio compression

Compression as we have seen does not depend only on the algorithm that is applied, but to the type of data that is intended compressed and to the quality of the copy generated at decompression time. This as covered in the previous section is extremely important to images but is also for sound.

One important fact that distinguish sound from image is that the general public does not have in general the capacity to make a qualified distinction regarding audio, by the general subjectiveness of the sense and for lack of education in regards to the intrinsics of an complex audio production.

Today due to the increasing importance of digital transferences of audio files. That seems finally to have gotten the support of the music media conglomerates, the importance of quality in regards to the consumer is of extreme importance.

10.4.1 mp3

Mp3 classifies not only a type of compression but a data format. It is used for lossy compression of audio and as a facilitator for digital audio distribution and categorization.

Most people will not be unable differentiate between an original (natural audio) and a 256kbps mp3 lossy reproduction.

An audio file that undergoes heavily sound compression when applied to a mp3 compressions, will generally sound worse to the human ear that the original, especially if encoded at 128 kbps (or less).

10.4.2 AAC

AAC is used mainly by Apple, sometimes encoded with the apple lossless audio codec (ALAC), however AAC is compatible with a huge range of non-Apple devices; see Wikipedia's page on AAC

10.4.3 FLAC

FLAC uses lossless compression.

10.5 lossy and residual give lossless

Sometimes people download a (highly compressed) image or video or music file, using it to decide if they really want to spend a (much longer) time downloading the high-resolution lossless version of the file.

Rather than delete the original (lossy) version of the file and start downloading the lossless version from scratch, several people are experimenting with methods of somehow using the partial information in the lossy version of the file in order to reduce the time required to download "residue", also called the "residual" -- the "rest" of the file (i.e., fill in the details that the lossy compressor discarded).

[12] [13] [14] [15] [16] [17] [18] [19] [20]

As long as the size of the (compressed) residue is significantly less than the size of the file stored in a stand-alone lossless format, the user has saved time -- even though the total lossy + residue size is usually larger than a stand-alone lossless format.

GIF and PNG, image file formats designed to be downloaded over relatively slow modems, have some of this characteristic -- they are designed to support "partial downloads".

Such lossless formats include JPEG XR, DTS-HD Master Audio, MPEG-4 SLS (lossless audio compression), Wavpack Hybrid, OptimFROG DualStream, ...

10.6 References

[1] Greg Roelofs. “PNG: The Definitive Guide: Chapter 9. Compression and Filtering”.

[2] Wikipedia: fractal transform

[3] “Fractal based Image compression algorithm (and source code)"

[4] Fractals/Iterated function systems

[5] "FAQ:What is the state of fractal compression?"

[6] “Technical Overview of Cartesian Perceptual Compression” (c) 1998-1999 Cartesian Products, Inc.

[7] http://www.reznik.org/software.html#IDCT

[8] Rock Brentwood. “Lossless & Non-Degrading Lossing DCT-Based Coding”

[9] Joint Photographic Experts Group : JPEG 2000 standard

[10] The Wavelet Discussion Forum

[11] The JasPer Project by Michael D. Adams, an implementation of JPEG-2000 Part-1.

[12] “Flexible 'Scalable to Lossless’?"

[13] Detlev Marpe, Gabi Blättermann, Jens Ricke, and Peter Maaß “A two-layered wavelet-based algorithm for efficient lossless and lossy image compression” 2000.

[14] GCK Abhayaratne and DM Monro. “Embedded to lossless coding of motion compensated prediction residuals in lossless video coding”

[15] G.C.K. Abhayaratne and D.M. Monro. “Embedded to Lossless Image Coding (ELIC)". 2000.

[16] CH Ritz and J Parsons. “Lossless Wideband Speech Coding”. 2004. “lossless coding of wideband speech by adding a lossless enhancement layer to the lossly baselayer produced by a standardised wideband speech coder”

[17] Nasir D. Memon, Khalid Sayood, Spyros S. Magliveras. “Simple method for enhancing the performance of lossy plus lossless image compression schemes” 1993.

[18] Qi Zhang Yunyang Dai Kuo, C.-C.J. Ming Hsieh “Lossless video compression with residual image prediction and coding (RIPC)". 2009.

[19] Gianfranco Basti and M. Riccardi and Antonio L. Perrone. “Lossy plus lossless residual encoding with dynamic preprocessing for Hubble Space Telescope fits images” 1999.

[20] Majid Rabbani and Paul W. Jones. “Digital image compression techniques”. 1991. Chapter 8: Lossy plus lossless residual encoding.

Chapter 11

Inference Compression

11.1 Inference Compression

The idea behind *inference compression* is that since the algorithm and its creators do not previous knowledge about the source data but compression can be optimized to different types of data, they will base the compression on a process of type inference. This process is mostly dynamic and non-deterministic consisting in reading part of the source data into a buffer (sniff) and execute some tests to infer its type/characteristics as to decide what compression algorithm to apply. This selection is often rule based (pre-defined on the algorithm).

This of course permits some latitude on how to perform the inference, some algorithms can use for instance the file extension (if present) or even by the file header alone (using file type identifiers like magic numbers or file designator strings), to recognize some well known file formats. They will then use that information to optimize the guessing process to determine the best compression scheme. An example might be the LHA and LHZ compression algorithms that contain a number of compression rules, permitting dynamic optimization in response to rule-base differentiation of the different file types.

Depending on some limitations and requirements (speed and memory utilization for instance), the rule base selection can be dynamic, for instance based in an *asymmetric compression* scheme, that will allow for multiple inferences, and let them compete to see which compression scheme will do the best job of compressing the data.

The idea is that the more closely you can infer the type of file, the more tightly you can compress it with a compression rule based particularly on that type of data. By tuning to the exact nature of the data, compression can often generate files that are significantly smaller than techniques which only approximate the type of data. For instance, a compression scheme that assumes Shakespearean text -- lots of common English words -- will probably not deal well with stock market tables -- rows of data showing the ticker symbol of a company, its closing price, etc.

11.2 Markov models

We discuss Markov models in more detail in a later chapter, Markov models.

11.3 data differencing

We discuss data differencing in more detail in a later chapter, data differencing.

Chapter 12

Streaming Compression

12.1 file vs streaming compression

Many data compression algorithms are file oriented. They assume that underlying transmission protocols have transmitted every bit of the compressed data file without error.

A few data compression algorithms are designed to be used with streaming one-way broadcast. For example, the MPEG-2 compression algorithm used by almost all over-the-air digital TV. At any time, a receiver may be turned on and start listening to the broadcast. In order for a receiver to start up in mid-stream -- or recover after a few seconds of severe transmission errors -- such systems typically break up the stream into "blocks". The decoder uses synchronization marks or checksums to detect the beginning of the next block, and starts decoding from there. At each new block, it re-initializes and restarts the decoding process. Block decoders that use a static per-block dictionary or static Huffman table forget the old one and read a new dictionary or table at the beginning of each block. Block decoders that use an adaptive dictionary forget the old dictionary, reinitialize the default dictionary, and begin building a new dictionary.

The block size is a compromise between compression (longer blocks usually give better compression) and quick restarts (shorter blocks allow the decompressor to restart more quickly after an error or after turning on). Because practically all errors can be detected with CRC codes, a streaming decoder knows which blocks are in error, and typically rejects the entire block. Streaming audio decoders use a variety of techniques to mask such errors -- they may attempt to "fill in" with a prediction that previous tones will continue. To avoid "clicks", they may choose to ramp down the volume to silence at the end of the last good block, play silence during the bad block, and then ramp back up to normal volume at the beginning of the next good block.

There has been little research on alternatives to "blocks". Some people speculate that, compared to using a block size of B bytes, it may be possible to achieve better compression and just as quick restarts with a (non-blocking) converging decoder designed so that all decoders, no matter when they are turned on, eventually converge on the same adaptive dictionary after at most B bytes. An adaptive block decoder has very little context (and poor compression) at the beginning of each block, gradually rising to lots of context (and much better compression) at the end of the block. A non-adaptive block decoder has the overhead of a dictionary or Huffman table at the beginning of each block (in effect, zero compression during the dictionary or table, and high compression during the data in the remainder of the block). The converging decoder, in theory, has some intermediate context (and some intermediate compression) at all times, without dictionary or table overhead or times of poor compression.

12.1.1 packet header compression

12.1.2 tape storage compression

One of the earliest applications for data compression was to compress data for backup tapes. Many tape drives have an implementation of a data compression algorithm embedded in the firmware of the tape drive itself. (With modern computers, one can get better compression by turning off this "hardware compression" and using a modern "software compression" algorithm. Software compression algorithms takes advantage of the much faster CPU and much larger RAM available to the main processor than to the processor inside the tape drive. Early computers could not use software compression because they were not fast enough to keep up with the tape).

Many tape drive manufacturers use a LZ algorithm. IBM and QIC use the ALDC algorithm.[1] Exabyte uses the IDRC algorithm. DLT uses the DLZ1 algorithm.

The VXA tape backup format (developed by Ecrix and produced by Exabyte) uses ALDC data compression.

Adaptive Lossless Data Compression algorithm (ALDC) is standardized by ECMA-222.[2]

The "Linear Tape-Open" (LTO) tape backup format (produced by several manufacturers) uses LTO-DC data compression, also called Streaming Lossless Data Compression (SLDC).

For legal reasons, many data compression algorithm patents are described as being part of such tape backup

storage systems.

12.1.3 implementation details

end handling

Most data compression algorithms are described in terms of abstract "streams" of data of indefinite length. In practice, data almost always comes in finite-length blocks with a distinct beginning and end. Systems that compress and decompress such data eventually come to the end of the block of data.

How does the decompressor know when to stop?

Some people argue that end-handling is "outside of the scope of data compression". And so many programmers use the "single responsibility principle" try separate the "end handling" code from the "decompression" code -- "end handling" is considered the responsibility of some other layer in the protocol stack. (Those other layers are described in more detail in other books, such as Data Coding Theory, Communication Networks, Serial Programming, etc.).

Archive file formats generally store compressed data in a "container format" that stores a block of compressed data with some metadata. There are 5 popular approaches to implementing such archive formats and streaming protocols:

1. The metadata includes both

   - the compressed length (this makes it much easier to skip over compressed files in an archive to decompress just one desired file), and
   - the decompressed length (this makes implementing the decompressor simpler)

2. The metadata describes exactly how many bits/bytes/symbols/pixels are in the *decompressed* data. The software that decodes the container format passes this "uncompressed length" and the compressed data to the decompressor, and the decompressor keeps a count of exactly how many bits/bytes/symbols/pixels it has produced so far, and stops when it produces enough. (Sometimes it is simpler to allow the decompressor to decode a few extra bits/bytes/symbols/pixels into a temporary buffer, and then copy the correct number into the final output). In some systems, the decompressor is required to return a count of how many *compressed* bytes it consumed, so the container handler knows where to continue decoding the next part of the file.

3. The compressed data is stored in some kind of "container format" that includes metadata describing exactly how many significant bits are in the *compressed* data. The software that decodes the container format passes this "compressed length" and the compressed data to the decompressor, and the decompressor keeps a count of exactly how many bits it eats up so far. In some systems, the decompressor is required to return a count of how many *decompressed* bytes it produces, so the container handler knows how many decompressed bytes in the buffer are "real" data.

4. The compressed data is stored in some kind of "container format" that includes metadata describing exactly how many bytes are in the *compressed* data, but the number of significant bits in the last byte is unknown. (Often the compressor pads out the compressed data with 0 bits until it fills out a whole byte. Somehow the decompressor needs to figure out the difference between "this is a codeword that happens to end with some (significant) zeros" -- in particular, the all-zeros codeword -- vs "this is not a real codeword but only zero padding". Software to handle this distinction is notoriously difficult to get exactly right and bug-free in all cases). In some systems, the decompressor is required to return a count of how many *decompressed* bytes it produces, so the container handler knows how many decompressed bytes in the buffer are "real" data.

5. The compressed data is stored or transmitted in a way so that neither the "compressed length" nor the "uncompressed length" are known, and somehow the decompressor is responsible for detecting the end and returning both the decompressed data and that "end point" metadata to the rest of the software.

Container formats are sometimes described as having separate "metadata sections" and "compressed data sections". But when Evaluating Compression Effectiveness, we must consider *all* the data given to the decompressor as the "compressed size" or "compressed rate" -- both the "compressed data sections" and data such as "uncompressed data size", "compressed size in bytes", and "compressed size in bits" that may normally be considered part of the "metadata sections".

A few people prefer compressed data formats designed such that concatenating a bunch of compressed files into one big file, and then later decompressing that big file, "works correctly" -- i.e., it produces one big decompressed file that is identical to concatenating all the original decompressed files into one big output file.[3][4][5][6]

12.2 Further reading

- Digital Television

[1] Craft, D. J. "A fast hardware data compression algorithm and some algorithmic extensions". IBM Journal of Research and Development. 1998.

[2] ECMA. "Standard ECMA-222: Adaptive Lossless Data Compression Algorithm"

[3] "Fast Concatenation of Multiple GZip Files".

[4] “Uncompress, edit, compress and concatenate files”.

[5] “Combine two or more compressed files”.

[6] “Append to a compressed stream”

Chapter 13

compressed file systems

13.1 file compression vs compressed file systems

FIXME: Is there a better place in this book to discuss compressed file systems and flash memory?

Many data compression algorithms are whole-file oriented. They assume that the entire original file is available up-front, and people will want to decompress the entire thing from beginning to end.

Compressed file systems (and especially flash-memory file systems) break that assumption. The ZFS file system uses the LZJB compression algorithm. Many internet routers and internet firewalls use a compressed flash-memory file system, often the cramfs file system. (See Reverse Engineering/File Formats#Compression, Encryption & Scrambling for an example). Some "solid-state hard drives" (SSDs) internally use compression algorithms to reduce the amount of flash "used up" by files -- this doesn't actually give users any more storage space, but the larger amount of empty storage space can be used internally by the SSD in ways that extend the lifetime of the drive.

These systems require relatively rapid random-access to data stored in the file system -- decompressing everything from the beginning would be too slow.

One approach is to use an index that, given some file name we want to look up (or a "logical block number"), it tells where exactly on the disk (or in the bulk flash) to start decompressing, combined with a streaming compression algorithm -- so the decompressor can jump into the middle of the compressed data and start decompressing from there.

A much more difficult requirement of these systems is to allow files to be edited. (This is so difficult that some compressed file systems, such as cramfs, such as python executable zipfiles[1], don't allow it -- they must be created all-at-once from a directory of uncompressed files. After a cramfs system is created, it can only be mounted as read-only).

Several people have built what is effectively a virtual file system that can read and write into a standard ".tgz" or ".zip" file: TrueZIP,[2] Java Zip File System Provider,[3] etc.

FIXME: go into a little more detail on why unmodified file-oriented compression algorithms won't work for compressed file systems, and what techniques are used to either (a) modify those algorithms so they are suitable, or (b) other algorithms that are useful for compressed flash file systems, or (c) combinations of both.

FIXME: say a few words about w: SquashFS.

FIXME: say a few words about w: initramfs/w: initrd.

13.2 virtual memory compression

FIXME: say a few words about zswap and zRAM and similar ideas related to virtual memory compression.

... using lz4 (a variant of lzo) ...

[1] Radomir Dopieralski. "Your Python Application as a Single File"

[2] TrueZIP https://truezip.java.net/

[3] Java Zip File System Provider http://docs.oracle.com/javase/7/docs/technotes/guides/io/fsp/zipfilesystemprovider.html

Chapter 14

Asymmetric Compression

14.1 Asymmetric Compression

The idea of Asymmetric Compression is quite simple, given a method of processing compression in such a way as to continually improve the compression ratio[*citation needed*], A server would be able to pack the data into a tighter and tighter file structure[*citation needed*], while replacing the data with a series of instructions that would then recover the data more quickly and in fewer steps. An example of this might be a program that tries different combinations of order injection or entropy injection until it finds the one that allows the greatest compression. Then incorporates a signifier that allows the decompression system to find the exact order/entropy injection used so that it can reverse the effects once the new compression step is completed, as part of the decompression algorythm. [1]

This concept is attractive to the server industry, however so far no such algorythm has become practical.

When a data compression algorithm takes about the same time to generate a compressed file as it does to uncompress that file, we call that algorithm "symmetric".

When a data compression algorithm takes much longer to generate a compressed file than it does to uncompress a file, we call that algorithm "asymmetric".

*(Are there any cases where a data compression algorithm takes much *less* time to generate a compressed file than it does to uncompress the file?)*

All video compression algorithms use asymmetric compression.

In particular, motion compensation is notorious for requiring a lot of time on the compression side. To get the best compression, the motion compensation subroutine during compression must compare every 16x16 block of pixels in the current frame with every 16x16 block of pixels in the previous frame, which takes many, many comparisons, and figure out which block of the previous frame gives the best match. During decompression, the motion compensation routine takes the coordinates of the "best match" from the compressed data stream, and copies the data from that location in the previous frame to the current 16x16 block of pixels in the current frame.

There are many computers that are plenty fast enough to play back motion-compensated compressed video in real time, but not quite fast enough to motion-compensate and compress video in real time. If you have such a computer, and you want to compress video, you can either:

- upgrade to a faster computer, one fast enough to do compression with full motion compensation in real time at high compression.
- use your slow computer to compress it in "batch mode" much slower than real time (perhaps overnight). Do all the same operations as that fast computer, resulting in a bit-for-bit identical compressed video file.
- use your slow computer, but do less work -- perhaps compare each 16x16 block of pixels with only a small neighborhood around the corresponding block in the previous frame. Since this slower computer does less work per frame, perhaps it will be fast enough to keep up with the video in real time. But because it doesn't do an exhaustive search for the "best" match, it doesn't compress as well, resulting in a larger compressed video file.

14.2 Further reading

[1] "Data Compression Explained" by Matt Mahoney: "In particular, it is not possible to compress random data or compress recursively."

- Wikipedia: motion compensation
- Wikipedia: data compression symmetry

Chapter 15

Multiple transformations

15.1 combination

Early text data compression techniques were divided into two very different-seeming approaches:

- fixed-to-variable decompressors that decompress fixed-size "references" into plaintext strings of varying length (Ziv and Lempel), and
- variable-to-fixed decompressors that decompress variable-length "prefix codes" or "arithmetic codes" into plaintext strings of some fixed length (Markov).

Researchers are searching for some way to combine them, to get the benefits of both. [1]

A typical paragraph of text has several different kinds of patterns. Some archive software applies a series of different transformations, each transformation targeting a different kind of pattern in the text.

Such a transformation is often based on some kind of model of how the file was produced.

Compression software (especially image compression software) often sends the raw data through one or more stages of decorrelation -- such as Karhunen–Loève transform[2], color space transformation, linear filtering, DFT, transform coding, etc. -- the "precompressor" stages -- before feeding the result into an entropy compressor.

Many transforms used in compression, counter-intuitively, produce an intermediate result that requires more bits of dynamic range to store. Transforming raw 24-bit RGB pixels into 26-bit YCoCg-R pixels[2] or transforming raw 16-bit audio samples into 17-bit delta coded differences may seem to "waste" bits, but it improves net compression performance: In many cases, the entropy compressor gives smaller compressed file sizes when given such "de-correlated" intermediate data than when given the original raw data. (The improvement in results resulting from such transformations is often called "coding gain").

15.1.1 color transforms

The JPEG 2000 Reversible Color Transform (RCT) is a YUV-like color space that does not introduce quantization errors, so it is fully reversible.

The compressor converts RGB to YCbCr with:

$$Y = \left\lfloor \frac{R + 2G + B}{4} \right\rfloor ; C_B = B - G; C_R = R - G;$$

and then compresses the image.

The decompressor decompresses the image, then takes YCbCr pixels and restores the original RGB image with:

$$G = Y - \left\lfloor \frac{C_b + C_r}{4} \right\rfloor ; R = C_R + G; B = C_B + G.$$

A pixel in the JPEG 2000 YCbCr format requires 26 bits to represent losslessly.

The Red, Green, and Blue layers of an image are typically highly correlated. So compressing the R, G, and B layers independently ends up storing much of the same data 3 times.

Even though "decorrelating" a 24-bit-per-pixel RGB image into a YCbCr requires *more* bits (26 bits) to hold each pixel, decorrelating helps later stages in the compression pipeline do a better job. So independently compressing the Y, Cb, and Cr planes usually results in a smaller compressed file than independently compressing the R, G, and B layers of the same file.

Most (but not all!) lossless color space transforms, including the JPEG 2000 RCT, are "expanding". "Expanding" color space transforms convert 24 bit RGB pixels to intermediate YCbCr pixels that require *more* than 24 bits to represent losslessly.

A few lossless color space transforms are non-expanding. "Non-expanding" color space transforms convert 24 bit RGB pixels to intermediate YCoCg24 pixels that require the same 24 bits to represent losslessly.

All color space transforms that convert 24 bit RGB pixels to something less than 24 bits per pixel are lossy. (Do I need to spell out why?).

15.1.2 audio transforms

music

... (fill in more details) ...

Most modern lossy audio codecs use modified discrete cosine transform (MDCT) as the decorrelation stage (as in Vorbis, AAC and AC-3) or as one of the decorrelation stages (as in MP3 and ATRAC).

speech

Speech compression often uses a relatively simple model of the human voice. The glottis produces a buzz at some frequency. The vocal tract resonates or dampens that frequency and its harmonics. The tongue and lips occasional producing hissing sibilants and popping plosive sounds.

A fairly intelligible speech sound can be reconstructed at low bit rate, by updating a linear predictive coding model at 30 to 50 frames per second.

There are several ways to represent the linear predictive filter coefficients.

While in principle the linear predictive filter coefficients could be transmitted directly, typically the system is set up so (after entropy decoding) the received bits represent reflection coefficients,[3] log area ratios, line spectral pairs, or some other representation of the filter coefficients.

15.1.3 Codec2

... (fill in details) ...

15.1.4 DEFLATE

DEFLATE combines LZ77 with Huffman encoding. It is specified in RFC 1951.

It is extremely widely used to decode ZIP files and gzip files and PNG image files.

zlib

The "zlib compressed data format specification" is a standardized way of packaging compressed data in a file, with a simple header that indicates the type of compression and helps detect the most common kinds of data corruption. It is often used to store data compressed with DEFLATE with a window size up to 32K.

The "zlib compressed data format specification" is specified in RFC 1950.

15.1.5 LZX

The LZX file format and compression algorithm was invented by Jonathan Forbes and Tomi Poutanen. LZX is used in a popular stand-alone AMIGA file archiver. Microsoft Compressed HTML Help (CHM) files use LZX compression. Windows Imaging Format (".WIM") files support several data compression methods, including LZX. Xbox Live Avatars use LZX compression.

LZX uses both LZ77-style sliding window compression and dynamic Huffman codes, somewhat like DEFLATE uses both. The LZ77-style window size (WINDOW_SIZE) is some power-of-two, at least 2^15 and at most 2^21. LZX uses several Huffman tree structures, each of them canonical.

The most important Huffman tree in LZX is the "main tree", composed of 256 elements corresponding to all possible ASCII characters, plus 8*NUM_POSITION_SLOTS elements corresponding to matches. (The NUM_POSITION_SLOTS is in the range of 30 slots to 50 slots). The second most important Huffman tree in LZX is the "length tree", composed of 249 elements. Other trees have smaller roles.

The LZX compressor arbitrarily divides up a file into one or more blocks. Each block is one of 3 types: "verbatim block", "aligned offset block", "uncompressed block". The "uncompressed block" has a few more things in the header, and then a literal copy of the uncompressed data.

In the data portion of the "verbatim block" and "aligned offset block" blocks, LZX uses length-limited Huffman codewords; LZX limits each codeword to at most 16 bits wide. Each "verbatim block" and "aligned offset block" has a larger header that gives the information (in a very compact differentially-compressed form) required for the decoder to regenerate the various Huffman trees used in that block.

The header is decoded to get the codeword length for each symbol, has a length of 0 to 16 bits (inclusive), where "0 bits" indicates the corresponding symbol never occurs in this block, and "16 bits" are used for symbols that occur extremely rarely. A variety of techniques are used to compress the header into relatively few bits.

When the "distance" value is decoded, there are several special values. It happens quite often that two long phrases of English text are *almost* identical, but there is a small typo or substitution in the middle of the otherwise-identical long phrase. With the original LZ77, the result is 3 copy items: [long distance X to first part, length of first part], [distance Y to the insertion, length of insertion], [exactly the same long distance X to start of second part, length of second part]. LZX, rather than repeat exactly the same long distance X twice, uses a special "distance" value to indicate that we are "continuing" to copy a previous block after making a substitution in the middle. (The special distance value ends up occupy-

ing fewer bits than the full long distance).

- "distance" == 0: "repeat the most recent match distance, which was not itself a repeated distance"
- "distance" == 1: "repeat the second-most-recent match distance, which was not itself a repeated distance"
- "distance" == 2: "repeat the third-most-recent match distance, which was not itself a repeated distance"
- distance == 3: copy starting from 1 byte back, the most-recently-decoded byte (pseudo-RLE on bytes), the closest allowable.
- distance == 4: copy starting from 2 bytes back, the next-most-recently-decoded byte (pseudo-RLE on byte pairs)
- distance == 5: copy starting from 3 bytes back
- distance == 6: copy starting from 4 bytes back
- ...
- distance == 1000: copy starting from 998 bytes back
- ...
- distance = x+2: copy starting from x bytes back
- ...
- distance == WINDOW_SIZE-1 (all bits high): copy starting from WINDOW_SIZE-3 bytes back, the most distant copy possible.

After the decompressor uses the block header to set up all the trees, the decompressor's main loop in a verbatim block is:

- Read a codeword from the compressed text using the main tree and decode it into a "main element".
- If the "main element" is less than 256, emit the "main element" as a literal plaintext byte.
- If the "main element" is 256 or more, we're going to have a copy item.
    - Break the "main element - 256" into two parts: a "slot" and a 3 bit "length".
    - If the "length" is the special all-ones value "7", read another codeword from the compressed text using the "length tree" and decode it into a "long length"; add that to the 7 already in the length.
    - If the "slot" is in the range 0..3, then set the distance the same as the slot.
    - If the "slot" is in the range 4..37, then extra = ((slot >> 1) - 1) more bits are needed to get the exact distance.
        - read those extra raw "footer" bits from the compressed text, (not using any Huffman tree, just the raw bits). The distance is (in binary) an implied 1 bit, followed by the (slot & 1) bit, followed by the extra raw footer bits.
    - If the distance is 0..2, it indicates a "repeated distance" (as described above) -- replace it with the actual distance we used before.
    - Do a normal LZ77-style copy from the previously decoded plaintext: copy from a distance (as described above) a total of "length + 2" bytes into the decoded plaintext.
- repeat until we've decoded the total number of bytes that the header says the uncompressed plaintext should contain.

After the decompressor uses the block header to set up all the trees, the decompressor's main loop in an aligned block is:

- Read a codeword from the compressed text using the main tree and decode it into a "main element".
- If the "main element" is less than 256, emit the "main element" as a literal plaintext byte.
- If the "main element" is 256 or more, we're going to have a copy item.
    - Break the "main element - 256" into two parts: a "slot" and a 3 bit "length".
    - If the "length" is the special all-ones value "7", read another codeword from the compressed text using the "length tree" and decode it into a "long length"; add that to the 7 already in the length.
    - If the "slot" is in the range 0..3, then set the distance the same as the slot.
    - If the "slot" is in the range 4..37, then extra = ((slot >> 1) - 1) more bits are needed to get the exact distance.
        - if extra > 3, read (extra - 3) raw "footer" bits from the compressed text, (not using any Huffman tree, just the raw bits). Then read another codeword from the compressed text using the "aligned tree" and decode it to 3 bits (0 to 7). The distance is (in binary) an implied 1 bit, followed by the (slot & 1) bit, followed by the (extra - 3) raw footer bits, followed by the 3 aligned bits.

    - if extra = 3, read another codeword from the compressed text using the "aligned tree" and decode it to 3 aligned bits (0 to 7). The distance is (in binary) an implied 1 bit, followed by the (slot & 1) bit, followed by the 3 aligned bits.
    - if extra is 1 or 2 bits, read those extra raw "footer" bits from the compressed text, (not using any Huffman tree, just the raw bits). The distance is (in binary) an implied 1 bit, followed by the (slot & 1) bit, followed by the extra raw footer bits.
  - If the distance is 0..2, it indicates a "repeated distance" (as described above) -- replace it with the actual distance we used before.
  - Do a normal LZ77-style copy from the previously decoded plaintext: copy from a distance (as described above) a total of "length + 2" bytes into the decoded plaintext. (The minimum copy length is 2 bytes).
- repeat until we've decoded the total number of bytes that the header says the uncompressed plaintext should contain.

The Microsoft "cabinet" (".CAB") compressed archive file format supports 3 data compression methods: DEFLATE, LZX, and Quantum compression. The compressor that puts executable files into a ".CAB" archive may optionally "detect "CALL" instructions, converting their operands from relative addressing to absolute addressing, thus calls to the same location resulted in repeated strings that the compressor could match, improving compression of 80x86 binary code." The ".CAB" decompressor, after doing LZX decompression described above, and if the "preprocessing" bit in the header was set to one, must convert those operands back to relative addressing. We will discuss such preprocessor/decorrelators in general, and Executable Compression specifically, in more detail in a later chapter.

[4][5][6][7][8]

15.1.6 LZX DELTA

LZX DELTA (LZXD) is a variation of LZX modified to support efficient delta compression (patching).

A LZX DELTA decompressor takes the old version of some plaintext file (this must be exactly the same as the old version used by the compressor) and the LZXD-compressed patch file, and uses them to generate the new version of that plaintext file.

The main difference between LZXD and LZX is that the window is pre-loaded with the reference file, with the decompressor decoding the first byte of the output file into the window at the location immediately after the last byte of the reference file. Normal LZX copy items have a distance that is at most the full distance back to the first byte of the output file; LZXD allows a distance that reaches much further back to allow copies from the reference file.

LZXD supports many more "slots", up to 290 slots, which are encoded into the main tree. This allows LZXD copy items to grab things much further into the past, allowing it to take advantage of very long reference files.

LZXD also supports much larger "length" values. Much as LZX has a "short length" of 0 to 7, where "7" is used as an escape for to get a "long length" from 7 to 255, LZXD uses the same "short length" and "long length" and goes one step further, using a long length of "257" as an escape for an "extra-long length" that can copy 32 KByte in a single copy.

[9]

15.1.7 LZARI

LZARI uses LZSS pre-processor followed by arithmetic encoding. LZARI was developed by Haruhiko Okumura.

15.1.8 LZH

LZH uses LZSS pre-processor followed by Huffman coding. LZH was developed by Haruyasu Yoshizaki for the LHA archiver, based on LZARI.[10]

15.1.9 LZB

A LZB compressor uses a LZSS pre-processor followed by Elias gamma coding of the "match length" and of the "offset". (That extra step makes LZB give a better compression ratio, but run slower, than LZSS alone).

LZB gets compression roughly equal to LZH and LZARI, but is simpler and faster,[10] because a universal code such as Elias gamma coding is simpler and faster to decode than Huffman or arithmetic encoding (respectively).

15.1.10 LZMA

The Lempel–Ziv–Markov chain algorithm (w:LZMA) uses a LZ77-like compression algorithm, whose output is encoded with range coding.

LZMA is one of the compression methods used in the open-source 7-Zip file archiver.

The LZ77-like compression part of LZMA has many similarities with the corresponding LZ77-like compression part of LZX. LZMA does not use Huffman codes -- instead, it uses context-coded binary arithmetic codes.

15.1.11 LZHAM

LZHAM (LZ, Huffman, Arithmetic, Markov) is a general purpose lossless data compression library by Richard Geldreich, Jr. Both the unstable/experimental version and the "bitstream compatibility" stable version of LZHAM are released under the MIT license.[11][12]

15.2 LZW and Huffman Encoding

Some people combine LZW and Huffman encoding.

The decoder for such a system reads a series of Huffman codes from the compressed file. The decoder decodes each variable-length Huffman code into an intermediate fixed-length 12-bit index. With a typical plaintext file, some of those 12-bit numbers are used far more frequently than others, and so are assigned Huffman codes of less than 12 bits. The decoder then uses a LZW decoder to recover the original variable-length plaintext of 1 or more characters associated with that intermediate index.

[13]

15.3 implementation tips

Many compressed file formats must be decompressed using an entropy decoder, followed by a whole series of transformations -- a kind of Wikipedia: Pipeline (software).

There are 4 ways to implement such decompressors (and the corresponding compressors):

- Multi-pass compression: each inner transformation reads all the input from some temporary file, and writes all its output to some temporary file, before the next stage starts.
- Run every stage more-or-less simultaneously
  - Use co-routines or threads or processes or Unix pipes. Alas, these techniques are not supported in many embedded system environments.
  - Use decompression functions that call "down" to fetch more data. The outer application directly calls only the final transformation stage, which calls some intermediate transformation stage, which calls some other intermediate transformation stage, which calls the entropy decoder at the beginning of the pipeline, which calls the routine that fetches bytes from the file. The outer application repeatedly calls that final transformation stage until all the data is processed.
  - Use decompression functions that return "up" to fetch more data. The outer application directly calls each stage in turn. Each stage reads data from its input buffer, updates its internal state, and writes data to its output buffer. That output buffer is the input buffer of the next stage. The outer application repeatedly cycles through all the stages until all the data is processed.

Converting software from a "call down to fetch more data" API to a "return up to fetch more data" API is difficult.[14]

15.4 encryption

Because encrypt-then-compress doesn't make the file any smaller, many people do compress-then-encrypt.[15][16][17][18][19][20][21][22][23]

Because modern CPUs process data faster than data can be read off a spinning disk or most external flash drives, it is possible to read and write a compressed, encrypted file in less time than it takes to read and write a normal plain text file containing the same data. [24]

A few people are doing research on algorithms that compress and encrypt simultaneously.[25][26][27][28][29][30]

Some researchers test encryption algorithms by trying to compress the encrypted data. Many encryption algorithms are specifically designed to produce ciphertext that is "indistinguishable from random noise".[31] When trying to implement such an algorithm, If compression produces a compressed output file that is smaller than the encrypted input file, that tells the researcher that there is a flaw in the encryption software -- either a bug in the implementation, or it's a correct implementation of an insecure algorithm.

15.5 implementation tips

Some people using lossless compression prefer to use "idempotent" *(Is there a better term for this?)* compression software, even if it takes a little longer to compress.[32] In other words, they expect that when you compress a file with "maximum compression" (often "−9n") to create one compressed file, then uncompress it, then re-compress it, they expect the resulting second compressed file is identical to the first compressed file.

This generally requires that each and every transformation implementation (at least when passed the "−9n" option) is idempotent. Some things that are not idempotent (and so should be avoided, at least when passed the "−9n" option) are:

- Some compressors store the original filename and

timestamp of the uncompressed file in the compressed file. To be idempotent, either the compressor must leave out the name and timestamp (that's what the "-n" typically does) or else the decompressor must restore the original name and timestamp.

- color transforms (if any) should be reversible (lossless)[2]
- For any given implementation of a decompression algorithm, there are almost always many different ways for a compressor to represent the original file -- there are many compressed files that will produce identically the same output file. (The exception is "bijective compression"). When attempting to gain "maximum compression", we select the way(s) that are the shortest -- but even then there are usually several ways that will give the same compressed filesize. To be idempotent, the implementation of the compressor must always pick the same way.

15.6 some information theory terminology

(FIXME: replace "idempotent" above with "non-drifting lossy", as defined below? Or is there an even better term?)

Some information theory people[33] classify transformations into 4 categories of loss:

- bijective (ED = 1 and DE = 1)
- idempotent (lossless but not bijective: ED = 1 but DE !=1). Nearly all compression algorithms designed for text or executable code are in this category.
- non-drifting lossy (EDE = E). Most lossy compression algorithms designed for audio and video and still images are in this category. Quantization is a non-drifting lossy transform. Starting from an initial image I0, the first encode/decode cycle gives you a slightly different image I1 (hopefully different only in ways normal humans are unlikely to notice). Starting from I1 and encode/decode to produce I2, then encode/decode I2 to produce I3 doesn't give any more loss: I1 == I2 == I3 == I4 ...
- The general case: degrading. Every encode/decode cycle generally produces a slightly different image (such as a series of FAX transmissions or a multi-generational copy using physical paper copy machines).

Some transforms are known not to completely remove all redundancy. In particular, researchers have designed special LZ77 encoders and LZW encoders that encode extra "redundant bits" in the choice of which copy was selected, in a way that is backwards-compatible: Standard LZ77 or LZW decoders can extract the original file. Special LZ77 or LZW decoders can extract the original file plus these extra "redundant bits". These extra bits can be used for Reed-Solomon or other forms of error correction, or to store metadata about the original file such as a public signature or message authentication code.[34][35]

15.7 References

[1] "LZRW4: Ziv and Lempel meet Markov" by Ross Williams. 23-Jul-1991.

[2] Henrique S. Malvar, Gary J. Sullivan, and Sridhar Srinivasan. "Lifting-based reversible color transformations for image compression". "the Karhunen-Loève transform... provides maximum decorrelation."

[3] Nimrod Peleg. "Linear Prediction Coding". p. 55

[4] Wikipedia: LZX (algorithm)

[5] "Microsoft LZX Data Compression Format"

[6] "lzx.c - LZX decompression source code"

[7] "lzx.c - LZX decompression source code" (has many comments on the difference between the documentation and the implementation, clarifying several details the "official" documentation neglects to specify)

[8] "Solve the File Format Problem: LZX".

[9] "LZX DELTA Compression and Decompression"

[10] Andy McFadden. "Hacking Data Compression Lesson 11 LZH, LZARI, and LZB". 1993.

[11] "LZHAM codec - unstable/experimental repo."

[12] github: LZHAM - Lossless Data Compression Codec (was Google Code: lzham) by Rich Geldreich.

[13] Steve Wolfman. "Compression with LZW and Huffman Encoding".

[14] "Hacking Data Compression" by Andy McFadden

[15] "Can I compress an encrypted file?"

[16] "Compress and then encrypt, or vice-versa?"

[17] "Composing Compression and Encryption"

[18] "Compress, then encrypt tapes"

[19] "Is it better to encrypt a message and then compress it or the other way around? Which provides more security?"

[20] "Compressing and Encrypting files on Windows"

[21] "Encryption and Compression"

[22] "Do encrypted compression containers like zip and 7z compress or encrypt first?"

[23] "When compressing and encrypting, should I compress first, or encrypt first?"

[24] Bill Cox. "TinyCrypt".

[25] Dahua Xie and C.-C. Jay Kuo ENHANCED MULTIPLE HUFFMAN TABLE (MHT) ENCRYPTION SCHEME USING KEY HOPPING http://viola.usc.edu/Publication/PDF/ISCAS/2004_ISCAS_Xie.pdf 2004

[26] "Encryption Using Huffman Coding". quote: "If you take a data file and compress it using Huffman coding techniques, you get something that looks random. Make the coding table the key... You could make it more difficult by arbitrarily permuting the ones and zeros and choosing random digraphs to encode... several dozen common digraphs as single symbols".

[27] mok-kong shen. "PREFIXCODING, an encryption scheme with pseudo-random prefix codes substitution and transposition".

[28] Douglas W. Jones. "Splay Tree Based Codes". quote: "Splay compression performs unexpectedly well on images... It is only an accident that splay trees are useful for encryption, and the security of variants on the algorithm is still largely an open question."

[29] Qing Zhou, Kwok-wo Wong, Xiaofeng Liao, Yue Hu. "On the security of multiple Huffman table based encryption" 2011.

[30] Gillman, Mohtashemi, and Rivest. "On Breaking a Huffman Code". 1996. doi:10.1.1.309.9256 quote: "The idea of using data compression schemes for encryption is very old, dating back at least to Roger Bacon in the 13th century".

[31] Wikipedia: ciphertext indistinguishability#Indistinguishable from random noise

[32] "gzip −9n sometimes generates a different output file on different architectures"

[33] https://groups.google.com/forum/?fromgroups=#!topic/comp.compression/r96nB43uuLs

[34] Stefano Lonardi and Wojciech Szpankowski. "Joint Source-Channel LZ'77 coding". Data Compression Conference, 273-282, Snowbird, 2003.

[35] Yonghui Wu, Stefano Lonardi, Wojciech Szpankowski. "Error-Resilient LZW Data Compression". Data Compression Conference, 193-202, Snowbird, 2006.

Chapter 16

Executable Compression

16.1 executable software compression

In this chapter, we are careful to distinguish between 3 kinds of software:

- "data compression software", the implementation of algorithms designed to compress and decompress various kinds of files -- this is what we referred to as simply "the software" in previous chapters.
- "executable files" -- files on the disk that are not merely some kind of data to be viewed or edited by some other program, but are the programs themselves -- text editors, word processors, image viewers, music players, web browsers, compilers, interpreters, etc.
- "source code" -- human-readable files on the disk that are intended to be fed into a compiler (to produce executable files) or fed into an interpreter.

Executable files are, in some ways, similar to English text -- and source code is even more similar -- and so data compression software that was designed and intended to be used on English text files often works fairly well with executable files and source code.

However, some techniques and concepts that apply to executable software compression that don't really make sense with any other kind of file, such as:

- self-decompressing executables, including boot-time kernel decompression[1][2][3][4]
- dead-code elimination and unreachable code elimination (has some analogy to lossy compression)
- refactoring redundant code, and the opposite process: inlining (has some analogy to dictionary compression)
- some kinds of code size reduction may locally make a subroutine, when measured in isolation, appear run slower, but improve the net performance of a system. In particular, loop unrolling, inlining, "complete jump tables" vs "sparse tests", and static linking, all may make a subroutine appear -- when measured in isolation -- to run faster, but may increase instruction cache misses, TLB cache misses, and virtual memory paging activity enough to reduce the net performance of the whole system.[5]
    - procedural abstraction[6]
    - cross-jumping, also called tail merging[6]
- pcode and threaded code
- compressed abstract syntax trees are, as of 1998, the most dense known executable format, and yet execute faster than bytecode.[7]
- JavaScript minifiers (sometimes called "JavaScript compression tools")[8] convert JavaScript source into "minified" JavaScript source that runs the same, but is smaller. CSS minifiers and HTML minifiers work similarly.[9]
    - Simple minifiers strip out comments and unnecessary whitespace.
    - Closure Compiler does more aggressive dead-code elimination and single-use-function inlining.
- code compression[10]
- run-time decompression[11]
- demand paging, also called lazy loading, a way to reduce RAM usage
- shared library
- PIC shared library and other techniques for squeezing a Linux distribution onto a single floppy[12][13] or a single floppy X Window System thin client.[14]
- copy-on-write and other technologies for reducing RAM usage[15]

- various technologies for reducing disk storage, such as storing only the source code (possibly compressed) and a just-in-time in-memory compiler like Wikipedia: Tiny C Compiler (or an interpreter), rather than storing only the native executable or both the source code and the executable.
- Selecting a machine language or higher-level language with high code density[16]
- various ways to "optimize for space" (including the "-Os" compiler option)[17]
- using newlib, uClibc, or sglibc instead of glibc[17][18][19]
- code compression for reducing power[20]
- Multiple "levels" or "waves" of unpacking: a file starts with a very small (machine-language) decompressor -- but instead of decompressing the rest of the file directly into machine language, the decompressor decompresses the rest of the file into a large, sophisticated decompressor (or interpreter or JIT compiler) (in machine language) and further data; then the large decompressor (or interpreter or JIT compiler) converts the rest of the file into machine language.[21]

16.2 compress then compile vs compile then compress

Which stage of compilation gives the best compression:

-
  - compress the final machine-specific binary executable code?
  - compress the original machine-independent text source code? For example, JavaScript minifiers.
  - compress some partially compiled machine-independent intermediate code? For example, "Slim Binaries" or the "JAR format"[22]

Some very preliminary early experiments[23] give the surprising result that compressed high-level source code is about the same size as compressed executable machine code, but compressing a partially-compiled intermediate representation gives a larger file than either one.

16.3 filtering

Many data compression algorithms use "filter" or "preprocess" or "decorrelate" raw data before feeding it into an entropy coder. Filters for images and video typically have a geometric interpretation. Filters specialized for executable software include:

- "detect "CALL" instructions, converting their operands from relative addressing to absolute addressing, thus calls to the same location resulted in repeated strings that the compressor could match, improving compression of 80x86 binary code." (Wikipedia: LZX (algorithm)#Microsoft Cabinet files)
- recoding branches into a PC-relative form[6]
- Instead of decompressing the latest version of an application in isolation, starting from nothing, start from the old version of an application, and patch it up until it is identical to the new latest version. That enables much smaller update files that contain only the patches -- the differences between the old version of an application and the latest version. This can be seen as a very specific kind of data differencing.
  - The algorithm used by BSDiff 4 uses using suffix sorting to build relatively short patch files[24]
  - Colin Percival, for his doctoral thesis, has developed an even more sophisticated algorithm for building short patch files for executable files.[24]
- "disassemble" the code, converting all absolute addresses and offsets into symbols; then patch the disassembled code; then "reassemble" the patched code. This makes the compressed update files for converting the old version of an application to the new version of an application much smaller.[25]

Several programmers believe that a hastily written program will be at least 10 times as large as it "needs" to be. [26] [27]

A few programmers believe that 64 lines of source code is more than adequate for many useful tools.[28]

The aim of the STEPS project is "to reduce the amount of code needed to make systems by a factor of 100, 1000, 10,000, or more." [29]

Most other applications of compression -- and even most of these executable compression techniques -- are intended to give results that appear the same to human users, while improving things in the "back end" that most users don't notice. However, some of these program compression ideas (refactoring, shared libraries, using higher-level languages, using domain-specific languages, etc.) reduce the amount of source that a human must read to understand a program, resulting in a significantly different experience for some people (programmers). That time savings can lead to significant cost reduction.[30] Such "compressed" source code is arguably better than the original; in contrast to image compression and other fields where compression gives, at best, something identical to the original, and often much worse.

16.4 References

[1] Embedded Linux wiki: "Fast Kernel Decompression"

[2] Smallcode: "Self-extracting executables " by Peter Kankowski, with a wiki discussion at strchr: "creating_self-extracting_executables"

[3] UPX is a portable executable packer for several different executable formats including linux/elf386, vmlinuz/386 and win32/pe.

[4] ficl "Ficl ... LZ77 compressed ... with the runtime decompressor, the resulting Ficl executable is over 13k smaller"

[5] "Managing code size"

[6] "Enhanced Code Compression for Embedded RISC Processors" by Keith D. Cooper and Nathaniel McIntosh

[7] "Java: Trees Versus Bytes" thesis by Kade Hansson. "A Compact Representation For Trees ... is the most dense executable form that he knows of. ... The creation of compact tree representations ... achieving better compression and developing faster codecs will no doubt be a growth area of research in coming years. One such project in this area may lead to an eventual replacement for the Java class file format."

[8] Javascript Compression tools

[9] w: minification (programming)

[10] "Code compression under the microscope" by Jim Turley 2004

[11] Charles Lefurgy, Eva Piccininni, and Trevor Mudge. "Reducing Code Size with Run-time Decompression".

[12] "Brian Writes about His BOEL (Part 1)" (a Linux distribution that fits on a single floppy)

[13] "BOEL, Part 2: Kernel Configuration and Booting"

[14] "2-Disk Xwindow embedded Linux": "a single floppy X Window System thin client."

[15] embedded Linux wiki: "Technologies for decreasing system size"

[16] Microprocessor Design/Code Density

[17] "Optimizing for Space : Measurements and Possibilities for Improvement" by Árpád Beszédes, Tamás Gergely, Tibor Gyimóthy, Gábor Lóki, and László Vidács 2003 from the GCC ARM Improvement Project at the University of Szeged

[18] uClibc FAQ

[19] "Embedding with GNU: Newlib" by Bill Gatliff 2001; "Embedding GNU: Newlib, Part 2" by Bill Gatliff 2002

[20] "COMPASS - A tool for evaluation of compression strategies for embedded processors" by Menon and Shankar, which cites "A Class of Code Compression Schemes for Reducing Power Consumption in Embedded Microprocessor Systems" by Benini, Menichelli, and Olivieri

[21] "A new visualization for packed and self-modifying programs".

[22] "Slim Binaries"

[23] "Code Size When Compiling to JavaScript" http://mozakai.blogspot.com/2011/11/code-size-when-compiling-to-javascript.html

[24] Colin Percival, Naive differences of executable code, http://www.daemonology.net/bsdiff/, 2003. [http://www.daemonology.net/bsdiff/

[25] "How Courgette works" by Stephen Adams 2009 at The Chromium Projects.

[26] "Thoughtful Programming and Forth" by Jeff Fox

[27] "1x Forth" by Charles Moore 1999

[28] "useful source code in 64 lines or less"

[29] "STEPS Toward The Reinvention of Programming" 2007 apparently (?) written by Alan Kay, Ted Kaehler, Ian Piumarta, Yoshiki Ohshima, Alex Warth, Takashi Yamamiya, Dan Amelang, Scott Wallace, Dan Ingalls, Andreas Raab, Jim Clune, John Maloney, Dave Smith, Kim Rose, Chuck Thacker, Vishal Sikka, and David Reed.

[30] "Code Reduction is Job #1"; "Should Code Reduction be Job #1?"

Chapter 17

References

17.1 Benchmark files

- The Canterbury Corpus (1997) is the main benchmark for comparing compression methods. Of these 11 files, the largest is roughly 1 MByte. That web page also links to a few other test files which are useful for debugging common errors in compression algorithms.
- The Silesia Corpus (2003) contains files between 6 MB and 51 MB. The 12 files includes two medical images, the SAO star catalog, a few executable files, etc.
- Matt Mahoney has published a large benchmark text file used in the "Large Text Compression Benchmark"
  - Large Text Compression Benchmark is a file "enwik9"(1,000,000,000 bytes), the first 10^9 bytes from the English Wikipedia dump on Mar. 3, 2006.
  - The the Hutter Prize involves compressing a file "enwik8"(100,000,000), the first 10^8 bytes of enwik9, ultimately from Wikipedia.
- A large-file text compression corpus, maintained by Andrew Tridgell, is oriented towards relatively large, highly redundant files. It contains 5 files between 27 MB and 85 MB (uncompressed), mostly English text and Lisp, assembly, and C source code. It helps test (implementations of) compression algorithms designed to detect and compress very long-range redundancies, such as lzip and rzip.
- "The Calgary corpus" is a series of 14 files, most of them ASCII text, and was the de facto standard for comparing lossless compressors before the Canterbury Corpus.
  - "The Calgary corpus Compression & SHA-1 crack Challenge" (formerly known as the "The Calgary Corpus Compression challenge") by Leonid A. Broukhis, has paid out several prizes around $100 each for "significantly better" compression of all 14 files in the Canterbury Corpus.
- "The Data Compression News Blog" edited by Sachin Garg. Sachin Garg has also published benchmark images and image compression benchmark results.
- Lasse Collin uses open-source software in his executable compression benchmark.
- Elephants Dream: Original lossless video and audio available: Matt suggests "It would be great to see Elephants Dream become the new standard source footage for video and audio compression testing!"
- Alex Ratushnyak maintains the Lossless Photo Compression Benchmark.
- "Xiph.org Video Test Media (derf's collection)" -- it includes the "SVT High Definition Multi Format Test Set".

17.2 open-source example code

Most data compression creators release open-source implementations of their algorithms. This makes it much easier to evolve the algorithms by combining combines clever ideas from many different sources.

- Compression Interface Standard by Ross Williams. Is there a better interface standard for compression algorithms?
- jvm-compressor-benchmark is a benchmark suite for comparing the time and space performance of open-source compression codecs on the JVM platform. It currently includes the Canterbury corpus and a few other benchmark file sets, and compares LZF, Snappy, LZO-java, gzip, bzip2, and a few other codecs. (Is the API used by the jvm-compressor-benchmark to talk to these codecs a good interface standard for compression algorithms?)
- inikep has put together a benchmark for comparing the time and space performance of open-source compression codecs that can be compiled with C++.

It currently includes 100 MB of benchmark files (bmp, dct_coeffs, english_dic, ENWIK, exe, etc.), and compares snappy, lzrw1-a, fastlz, tornado, lzo, and a few other codecs.

- "Compression the easy way" simple C/C++ implementation of LZW (variable bit length LZW implementation) in one .h file and one .c file, no dependencies.
- BALZ by Ilia Muraviev - the first open-source implementation of ROLZ compression[1]
- QUAD - an open-source ROLZ-based compressor from Ilia Muraviev
- LZ4 "the world's fastest compression library" (BSD license)
- QuickLZ "the world's fastest compression library" (GPL and commercial licenses)
- FastLZ "free, open-source, portable real-time compression library" (MIT license)
- The .xz file format (one of the compressed file formats supported by 7-Zip and LZMA SDK) supports "Multiple filters (algorithms): ... Developers can use a developer-specific filter ID space for experimental filters." and "Filter chaining: Up to four filters can be chained, which is very similar to piping on the UN*X command line."
- libarchive (win32 LibArchive): library for reading and writing streaming archives. The bsdtar archiving program is based on LibArchive. Libarchive is highly modular. "designed ... to make it relatively easy to add new archive formats and compression algorithms". LibArchive can read and write (including compression and decompression) archive files in a variety of archive formats including ".tgz" and ".zip" formats. BSD license. libarchive WishList.
- WebP is a new image format that provides lossless and lossy compression for images on the web. "WebP lossless images are 26% smaller in size compared to PNGs. WebP lossy images are 25-34% smaller in size compared to JPEG images at equivalent SSIM index." WebP is apparently the *only* format supported by web browsers that supports both lossy compression and an alpha channel in the same image. When the experimental "data compression proxy" is enabled in Chrome for Android, all images are transcoded to WebP format.[2] BSD license.
- VP8 and WebM video compression ...
- The Ogg container format, often containing compressed audio in Vorbis, Speex, or FLAC format, and sometimes containing compressed video in Theora or Dirac format, etc.
- libPFG, library for reading and writing files in Progressive Graphics File "PGF" format. Uses fast wavelet transform; lossless and lossy compression. Supports alpha transparency. LGPL.

17.3 Further reading

- Fedora And Red Hat System Administration/Archives And Compression has some practical information on how to use compression
- JPEG - Idea and Practice has more detailed information on the specific details of how compression techniques are applied to JPEG image compression.
- Data Coding Theory/Data Compression
- Kdenlive/Video codecs briefly mentions the most popular video codecs
- Movie Making Manual/Post-production/Video codecs discusses the most popular video codecs used in making movies and video, in a little more detail.
- Movie Making Manual/Cinematography/Cameras and Formats/Table of Formats lists the most popular compressed and uncompressed video formats
- Probability
- The hydrogenaudio wiki has a comparison of popular lossless audio compression codecs.
- a data compression wiki
- a data compression wiki

non-wiki resources

[1] "Anatomy of ROLZ data archiver"

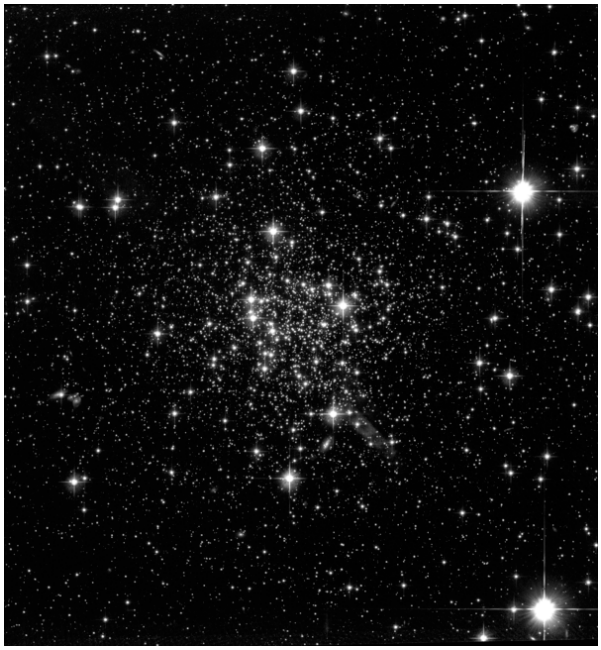
[2]

- "comp.compression" newsgroup
  - "Comp.compression FAQ"
  - comp.compression Frequently Asked Questions by Jean-loup Gailly 1999. *(is there a more recent FAQ???)*
- http://data-compression.info/ has some information on several compression algorithms, several "data compression corpora" (benchmark files for data compression), and the results from running a variety of data compression programs on those benchmarks (measuring compressed size, compression time, and decompression time).

- "Data Compression Explained" by Matt Mahoney. It discusses many things neglected in most other discussions of data compression. Such as the practical features of a typical archive format (the stuff in the thin wrapper around your precious compressed data), the close relation between data compression and artificial intelligence, etc.
- Mark Nelson writes about data compression
    - Mark Nelson and Jean-loup Gailly. "The Data Compression Book". 1995. ISBN 1-55851-434-1.
- the Encode's Forum claims to be "probably the biggest forum about the data compression software and algorithms on the web".
- "The LZW controversy" by Stuart Caie. (LZ78, LZW, GIF, PNG, Unisys, patents, etc.)

Chapter 18

Data Compression



There are many algorithms -- and even more implementations of those algorithms -- that accept unencoded (plain) information and encode it to use fewer bits. Perhaps even more important is the paired algorithm that accepts the encoded bits and extracts the information.

Each pair of algorithms -- one that creates the encoded form, and the other that accepts the encoded form and extracts the information -- is called a **data compression algorithm**. These type of algorithms are increasing abundant, as are their variations, most utilize dictionary based schemes and statistical methods. They are also becoming increasingly specialized in compressing specific kinds of data -- text, speech, music, photos, or video...

Data compression is useful in some situations because "compressed data" will save time (in reading and on transmission) and space if compared to the unencoded information it represent. Many programmers attempt to develop new algorithms to tightly compress the data into as few bits as possible (while still being able to recover the relevant information). However, other programmers -- after testing several algorithms -- deliberately pick some algorithm *other* than the one that would give them the fewest compressed bits. They deliberately sacrifice a few bits in order to improve latency, reduce compression time, or reduce decompression time.

Some communication systems carefully and deliberately add small amounts of "redundancy" -- the Data Coding Theory Wikibook has more details.

There are also several data compression benchmarks available for comparing data compression algorithms -- even one 50,000 euro cash prize for compressing one particular benchmark file as small as possible (and, of course, uncompressing it afterwards).

In this book, we describe the decompressor first, for several reasons: [1]

- Decompression routines are usually much easier to understand than the paired compression routine.
- Too many people write the compressor first, then later discover that it doesn't save enough information to reconstruct the original file.
- Once we have a decompressor, we can often improve the compressor alone without changing the file format or the decompressor.
- Many compression systems only document and standardize the decompressor and the file format. (Probably because of the above reasons).

18.1 Contents

- Order/Entropy
- Lossy vs. Non-Lossy Compression
- Inference Compression
- Streaming Compression
- compressed file systems
- Asymmetric Compression
- Multiple transformations
- Executable Compression
- References

18.2 References

[1] “Compression Basics” by Pasi 'Albert' Ojala 1997

18.3 Text and image sources, contributors, and licenses

18.3.1 Text

- **Data Compression/Coding** *Source:* https://en.wikibooks.org/wiki/Data_Compression/Coding?oldid=3034097 *Contributors:* DavidCary, Panic2k4, Graeme E. Smith and Anonymous: 2
- **Data Compression/Evaluating Compression Effectiveness** *Source:* https://en.wikibooks.org/wiki/Data_Compression/Evaluating_Compression_Effectiveness?oldid=3053313 *Contributors:* DavidCary, Panic2k4, Adrignola and Graeme E. Smith
- **Data Compression/Models** *Source:* https://en.wikibooks.org/wiki/Data_Compression/Models?oldid=2446931 *Contributors:* DavidCary and Graeme E. Smith
- **Data Compression/Markov models** *Source:* https://en.wikibooks.org/wiki/Data_Compression/Markov_models?oldid=2736472 *Contributors:* DavidCary and Anonymous: 1
- **Data Compression/The zero-frequency problem** *Source:* https://en.wikibooks.org/wiki/Data_Compression/The_zero-frequency_problem?oldid=3049401 *Contributors:* DavidCary, Panic2k4, QuiteUnusual and QUBot
- **Data Compression/data differencing** *Source:* https://en.wikibooks.org/wiki/Data_Compression/data_differencing?oldid=2947293 *Contributors:* DavidCary
- **Data Compression/Dictionary compression** *Source:* https://en.wikibooks.org/wiki/Data_Compression/Dictionary_compression?oldid=3066289 *Contributors:* DavidCary, Panic2k4, Graeme E. Smith, ToddWC, Avicennasis and Anonymous: 5
- **Data Compression/grammar-based compression** *Source:* https://en.wikibooks.org/wiki/Data_Compression/grammar-based_compression?oldid=2756138 *Contributors:* DavidCary and Cic
- **Data Compression/Order/Entropy** *Source:* https://en.wikibooks.org/wiki/Data_Compression/Order/Entropy?oldid=2586438 *Contributors:* DavidCary, Damian Yerrick, Graeme E. Smith, Avicennasis and Anonymous: 3
- **Data Compression/Lossy vs. Non-Lossy Compression** *Source:* https://en.wikibooks.org/wiki/Data_Compression/Lossy_vs._Non-Lossy_Compression?oldid=2466315 *Contributors:* DavidCary, Panic2k4, Graeme E. Smith, Avicennasis and Anonymous: 1
- **Data Compression/Inference Compression** *Source:* https://en.wikibooks.org/wiki/Data_Compression/Inference_Compression?oldid=2947294 *Contributors:* DavidCary, Panic2k4, Graeme E. Smith and Avicennasis
- **Data Compression/Streaming Compression** *Source:* https://en.wikibooks.org/wiki/Data_Compression/Streaming_Compression?oldid=3047043 *Contributors:* DavidCary, Adrignola, Graeme E. Smith and Anonymous: 3
- **Data Compression/compressed file systems** *Source:* https://en.wikibooks.org/wiki/Data_Compression/compressed_file_systems?oldid=2961000 *Contributors:* DavidCary and Cic
- **Data Compression/Asymmetric Compression** *Source:* https://en.wikibooks.org/wiki/Data_Compression/Asymmetric_Compression?oldid=2150734 *Contributors:* DavidCary and Graeme E. Smith
- **Data Compression/Multiple transformations** *Source:* https://en.wikibooks.org/wiki/Data_Compression/Multiple_transformations?oldid=3066315 *Contributors:* DavidCary, QUBot and Mamun070
- **Data Compression/Executable Compression** *Source:* https://en.wikibooks.org/wiki/Data_Compression/Executable_Compression?oldid=3051677 *Contributors:* DavidCary, Graeme E. Smith and Anonymous: 2
- **Data Compression/References** *Source:* https://en.wikibooks.org/wiki/Data_Compression/References?oldid=3031705 *Contributors:* DavidCary, Panic2k4, Graeme E. Smith, Avicennasis and Anonymous: 1
- **Data Compression** *Source:* https://en.wikibooks.org/wiki/Data_Compression?oldid=2947289 *Contributors:* DavidCary, Panic2k4, Whiteknight, Adrignola, Graeme E. Smith and Anonymous: 1

18.3.2 Images

- **File:50_percents.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/6/62/50_percents.svg *License:* CC0 *Contributors:* File:50%.svg *Original artist:* Ftiercel
- **File:Clipboard.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/1/1f/Clipboard.svg *License:* GPL *Contributors:* Self-made. Based on Image:Evolution-tasks-old.png, which was released into the public domain by its creator AzaToth. *Original artist:* Tkgd2007
- **File:Dctjpeg.png** *Source:* https://upload.wikimedia.org/wikipedia/commons/2/23/Dctjpeg.png *License:* Public domain *Contributors:* No machine-readable source provided. Own work assumed (based on copyright claims). *Original artist:* No machine-readable author provided. FelixH~commonswiki assumed (based on copyright claims).
- **File:PrimitiveTypes.png** *Source:* https://upload.wikimedia.org/wikipedia/commons/b/b9/PrimitiveTypes.png *License:* Public domain *Contributors:* original, derived work ([1]) *Original artist:* Panic2k7
- **File:Terzan_7.png** *Source:* https://upload.wikimedia.org/wikipedia/commons/b/bc/Terzan_7.png *License:* Public domain *Contributors:* http://archive.stsci.edu/cgi-bin/mastpreview?mission=hst&dataid=U37G0201R *Original artist:* Rodrigo Ibata
- **File:Warning_icon_WikiBooks.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/1/1f/Warning_icon_WikiBooks.svg *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* Dtom
- **File:Wikipedia-logo.png** *Source:* https://upload.wikimedia.org/wikipedia/commons/6/63/Wikipedia-logo.png *License:* GFDL *Contributors:* based on the first version of the Wikipedia logo, by Nohat. *Original artist:* version 1 by Nohat (concept by Paullusmagnus);
- **File:Wikipedia_in_binary.gif** *Source:* https://upload.wikimedia.org/wikipedia/commons/7/77/Wikipedia_in_binary.gif *License:* CC BY-SA 3.0 *Contributors:* Own work *Original artist:* User:Atyndall (See account on English Wikipedia)
- **File:Wikiversity-logo.svg** *Source:* https://upload.wikimedia.org/wikipedia/commons/9/91/Wikiversity-logo.svg *License:* CC BY-SA 3.0 *Contributors:* Snorky (optimized and cleaned up by verdy_p) *Original artist:* Snorky (optimized and cleaned up by verdy_p)

18.3.3 Content license

- Creative Commons Attribution-Share Alike 3.0