Data Compression Explained

Matt Mahoney

Last update: Mar. 10, 2010.

About this Book

This book is for the reader who wants to understand how data compression works, or who wants to write data compression software. Prior programming ability and some math skills will be needed. Specific topics include:

This book is intended to be self contained. Sources are linked when appropriate, but you don't need to click on them to understand the material.

1. Information Theory

Data compression is the art of reducing the number of bits needed to store or transmit data. Compression can be either lossless or lossy. Losslessly compressed data can be decompressed to exactly its original value. An example is 1848 Morse Code. Each letter of the alphabet is coded as a sequence of dots and dashes. The most common letters in English like E and T receive the shortest codes. The least common like J, Q, X, and Z are assigned the longest codes.

All data compression algorithms consist of at least a model and a coder (with optional preprocesing transforms). A model estimates the probability distribution (E is more common than Z). The coder assigns shorter codes to the more likely symbols. There are efficient and optimal solutions to the coding problem. However, optimal modeling has been proven not computable. Modeling (or equivalently, prediction) is both an artificial intelligence (AI) problem and an art.

Lossy compression discards "unimportant" data, for example, details of an image or audio clip that are not perceptible to the eye or ear. An example is the 1953 NTSC standard for broadcast color TV, used until 2009. The human eye is less sensitive to fine detail between colors of equal brightness (like red and green) than it is to brightness (black and white). Thus, the color signal is transmitted with less resolution over a narrower frequency band than the monochrome signal.

Lossy compression consists of a transform to separate important from unimportant data, followed by lossless compression of the important part and discarding the rest. The transform is an AI problem because it requires understanding what the human brain can and cannot perceive.

Information theory places hard limits on what can and cannot be compressed losslessly, and by how much:

1. There is no such thing as a "universal" compression algorithm that is guaranteed to compress any input, or even any input above a certain size. In particular, it is not possible to compress random data or compress recursively.
2. Given a model (probability distribution) of your input data, the best you can do is code symbols with probability p using $\log_2 1/p$ bits. Efficient and optimal codes are known.
3. Data has a universal but uncomputable probability distribution. Specifically, any string x has probability (about) $2^{-|M|}$ where M is the shortest possible description of x, and |M| is the length of M in bits, almost independent of the language in which M is written. However there is no general procedure for finding M or even estimating |M| in any language. There is no algorithm that tests for randomness or tells you whether a string can be compressed any further.

1.1. No Universal Compression

This is proved by the counting argument. Suppose there were a compression algorithm that could compress all strings of at least a certain size, say, n bits. There are exactly 2^n different binary strings of length n. A universal compressor would have to encode each input differently. Otherwise, if two inputs compressed to the same output, then the decompresser would not be able to decompress that output correctly. However there are only 2^n - 1 binary strings shorter than n bits.

In fact, the vast majority of strings cannot be compressed by very much. The fraction of strings that can be compressed from n bits to m bits is at most 2^{m-n}. For example, less than 0.4% of strings can be compressed by one byte.

Every compressor that can compress any input must also expand some of its input. However, the expansion never needs to be more than one symbol. Any compression algorithm can be modified by adding one bit to indicate that the rest of the data is stored uncompressed.

The counting argument applies to systems that would recursively compress their own output. In general, compressed data appears random to the algorithm that compressed it so that it cannot be compressed again.

1.2. Coding is Bounded

Suppose we wish to compress the digits of π, e.g. "3141592653589793238462643...". Assume our model is that each digit occurs with probability 0.1, independent of any other digits. Consider 3 possible binary codes:

Digit	BCD	Huffman	Binary
0	0000	000	0
1	0001	001	1
2	0010	010	10
3	0011	011	11
4	0100	100	100
5	0101	101	101
6	0110	1100	110
7	0111	1101	111
8	1000	1110	1000
9	1001	1111	1001
bpc	4.0	3.4	not valid

Using a BCD (binary coded decimal) code, π would be encoded as 0011 0001 0100 0001 0101... (Spaces are shown for readability only). The compression ratio is 4 bits per character (4 bpc). If the input was ASCII text, the output would be compressed 50%. The decompresser would decode the data by dividing it into 4 bit strings.

The Huffman code would code π as 011 001 100 001 101 1111... The decoder would read bits one at a time and decode a digit as soon as it found a match in the table (after either 3 or 4 bits). The code is uniquely decodable because no code is a prefix of any other code. The compression ratio is 3.4 bpc.

The binary code is not uniquely decodable. For example, 111 could be decoded as 7 or 31 or 13 or 111.

There are better codes than the Huffman code given above. For example, we could assign Huffman codes to pairs of digits. There are 100 pairs each with probability 0.01. We could assign 6 bit codes (000000 through 011011) to 00 through 27, and 7 bits (0111000 through 1111111) to 28 through 99. The average code length is 6.72 bits per pair of digits, or 3.36 bpc. Similarly, coding groups of 3 digits using 9 or 10 bits would yield 3.3253 bpc.

Shannon and Weaver (1949) proved that the best you can do for a symbol with probability p is assign a code of length $\log_2 1/p$. In this example, $\log_2 1/0.1 = 3.3219$ bpc.

Shannon defined the information content or equivocation (now called entropy) of a random variable X as its expected code length. Suppose X may have values X_1, X_2,... and that each X_i has probability p(i). Then the entropy of X is $H(X) = E[\log_2 1/p(X)] = \Sigma_i\ p(i) \log_2 1/p(i)$. For example, the entropy of the digits of π, according to our model, is $10\ (0.1 \log_2 1/0.1) = 3.3219$ bpc. There is no smaller code for this model that could be decoded unambiguously.

The information content of a set of strings is at most the sum of the information content of the individual strings. If X and Y are strings, then $H(X,Y) \le H(X) + H(Y)$. If they are equal, then X and Y are *independent*. Knowing one string would tell you nothing about the other.

The *conditional entropy* $H(X|Y) = H(X,Y) - H(Y)$ is the information content of X given Y. If X and Y are independent, then $H(X|Y) = H(X)$.

If X is a string of symbols $x_1x_2...x_n$, then by the chain rule, p(X) may be expressed as a product of the sequence of symbol predictions conditioned on previous symbols: $p(X) = \Pi_i\ p(x_i|x_{1..i-1})$. Likewise, the information content H(X) of random string X is the sum of the conditional entropies of each symbol given the previous symbols: $H(X) = \Sigma_i\ H(x_i|x_{1..i-1})$.

Entropy is both a measure of uncertainty and a lower bound on compression. The entropy of a string is the limit to which you can compress it. There are efficient coding methods, such as arithmetic codes, which are for all practical purposes optimal in this sense. It should be emphasized, however, that entropy can only be calculated for a known probability distribution. But in general, the model is not known.

1.3. Modeling is Not Computable

We modeled the digits of π as uniformly distributed and independent. Given that model, Shannon's coding theorem places a hard limit on the best compression that could be achieved. However, it is possible to use a better model. The digits of π are not really random. The digits are only unknown until you compute them. An intelligent compressor might recognize the digits of π and encode it as a description meaning "the first million digits of pi", or as a program that reconstructs the data when run. With our previous model, the best we could do is $(10^6 \log_2 10)/8 \approx 415{,}241$ bytes. Yet, there are very small programs that can output π, some as small as 52 bytes.

The counting argument says that most strings are not compressible. So it is a rather remarkable fact that most strings that we care about, for example English text, images, software, sensor readings, and DNA, are in fact compressible. These strings generally have short descriptions, whether they are described in English or as a program in C or x86 machine code.

Solomonoff (1960, 1964), Kolmogorov (1965), and Chaitin (1966) independently proposed a universal a-priori probability distribution over strings based on their minimum description length. The algorithmic probability $K_L(x)$ of a string x is defined as the fraction of random programs in some language L that output x, where each program M is weighted by $2^{-|M|}$ and |M| is the length of M in bits. This probability is dominated by the shortest such program.

Algorithmic probability and complexity of a string x depend on the choice of language L, but only by a constant that is independent of x. Suppose that M1 and M2 are encodings of x in languages L1 and L2 respectively. For example, if L1 is C++, then M1 would be a program in C++ that outputs x. If L2 is English, the M2 would be a description of x with just enough detail that it would allow you to write x exactly. Now it is possible for any pair of languages to write in one language a compiler or interpreter or rules for understanding the other language. For example, you could write a description of the C++ language in English so that you could (in theory) read any C++ program and predict its output by "running" it in your head. Conversely, you could (in theory) write a program in C++ that input an English language description and translated it into C++. The size of the language description or compiler does not depend on x in any way. Then for any description M1 in any language L1 of any x, it is possible to find a descrption M2 in any other language L2 of x by appending to M1 a fixed length description of L1 written in L2.

It is not proven that algorithmic probability is a true universal prior probability. Nevertheless it is widely accepted on empirical grounds because of its success in sequence prediction and machine learning over a wide range of data types. In machine learning, the minimum description length principle of choosing the simplest hypothesis consistent with the training data applies to a wide range of algorithms. It formalizes Occam's Razor. Occam noted in the 14'th century, that (paraphrasing) "the simplest answer is usually the correct answer". Occam's Razor is universally applied in all of the sciences because we know from experience that the simplest or most elegant (i.e. shortest) theory that explains the data tends to be the best predictor of future experiments.

To summarize, the best compression we can achieve for any string x is to encode it as the shortest program M in some language L that outputs x. Furthermore, the choice of L becomes less important as the strings get longer. All that remains is to find a procedure that finds M for any x in some language L. However, Kolmogorov proved that there is no such procedure in any language. Suppose there were. Then it would be possible to describe "the first string that cannot be described in less than a million bits" leading to the paradox that we had just done so. (By "first", assume an ordering over strings from shortest to longest, breaking ties lexicographically).

Because determining the length of the shortest descriptions of strings is not computable, neither is optimal compression. It is not hard to find difficult cases. For example, consider the short description "a string of a million zero bytes compressed with AES in CBC mode with key 'foo'". To any program that does not know the key, the data looks completely random and incompressible.

1.4. Compression is an Artificial Intelligence Problem

Optimal compression, if it were computable, would optimally solve the artificial intelligence (AI) problem under two vastly different definitions of "intelligence": the Turing test (Turing, 1950), and universal intelligence (Legg and Hutter, 2006).

Turing first proposed a test for AI to sidestep the philosophically difficult question (which he considered irrelevant) of whether machines could think. This test, now known as the Turing test, is now widely accepted. The test is a game played by two humans who have not previously met and the machine under test. One human (the judge) communicates with the other human (the confederate) and the machine through a terminal. Both the confederate and the machine try to convince the judge that each is human. If the judge cannot guess correctly which is the machine 70% of the time after 10 minutes of interaction, then the machine is said to have AI. Turing gave the following example of a possible dialogue:

> Q: Please write me a sonnet on the subject of the Forth Bridge.
> A: Count me out on this one. I never could write poetry.
> Q: Add 34957 to 70764.
> A: (Pause about 30 seconds and then give as answer) 105621.
> Q: Do you play chess?
> A: Yes.
> Q: I have K at my K1, and no other pieces. You have only K at K6 and R at R1. It is your move. What do you play?
> A: (After a pause of 15 seconds) R-R8 mate.

It should be evident that compressing transcripts like this requires the ability to compute a model of the form p(A|Q) = p(QA)/P(Q) where Q is the context up to the current question, and A is the response. But if a model could make such predictions accurately, then it could also generate responses indistinguishable from that of a human.

Predicting transcripts is a similar problem to predicting ordinary written language. It requires in either case vast, real-world knowledge. Shannon (1950) estimated that the information content of written case-insensitive English without punctuation is 0.6 to 1.3 bits per character, based on experiments in which human subjects guessed successive characters in text with the help of letter n-gram frequency tables and dictionaries. The uncertainty is due not so much to variation in subject matter and human skill as it is due to the fact that different probability assignments lead to the same observed guessing sequences. Nevertheless, the best text compressors are only now compressing near the upper end of this range.

Legg and Hutter proposed the second definition, universal intelligence, to be far more general than Turing's human intelligence. They consider the problem of reward-seeking agents in completely arbitrary environments described by random programs. In this model, an agent communicates with an environment by sending and receiving symbols. The environment also sends a reinforcement or reward signal to the agent. The goal of the agent is to maximize accumulated reward. Universal intelligence is defined as the expected reward over all possible environments, where the probability of each environment described by a program M is algorithmic, proportional to $2^{-|M|}$. Hutter (2004, 2007) proved that the optimal (but not computable) strategy for the agent is to guess after each input that M is the shortest program consistent with past observation.

Hutter calls this strategy AIXI. It is, of course, is just our uncomputable compression problem applied to a transcript of past interaction. AIXI may also be considered a formal statement and proof of Occam's Razor. The best predictor of the future is the simplest or shortest theory that explains the past.

1.5. Summary

There is no such thing as universal compression, recursive compression, or compression of random data.

Most strings are random. Most meaningful strings are not.

Compression = modeling + coding. Coding is a solved problem. Modeling is provably not solvable.

Compression is both an art and an artificial intelligence problem. The key to compression is to understand the data you want to compress.

2. Benchmarks

A data compression benchmark measures compression ratio over a data set, and sometimes memory usage and speed on a particular computer. Some benchmarks evaluate size only, in order to avoid hardware dependencies. Compression ratio is often measured by the size of the compressed output file, or in bits per character (bpc) meaning compressed bits per uncompressed byte. In either case, smaller numbers are better. 8 bpc means no compression.

Generally there is a 3 way trade off between size, speed, and memory usage. The top ranked compressors by size require a lot of computing resources.

2.1. Calgary Corpus

The Calgary corpus is the oldest compression benchmark still in use. It was created in 1987 and described in a survey of text compression models in 1989 (Bell, Witten and Cleary, 1989). It consists of 14 files with a total size of 3,141,622 bytes as follows:

```
  111,261 BIB    - ASCII text in UNIX "refer" format - 725 bibliographic references.
  768,771 BOOK1  - unformatted ASCII text - Thomas Hardy: Far from the Madding Crowd.
  610,856 BOOK2  - ASCII text in UNIX "troff" format - Witten: Principles of Computer Speech.
  102,400 GEO    - 32 bit numbers in IBM floating point format - seismic data.
  377,109 NEWS   - ASCII text - USENET batch file on a variety of topics.
   21,504 OBJ1   - VAX executable program - compilation of PROGP.
  246,814 OBJ2   - Macintosh executable program - "Knowledge Support System".
   53,161 PAPER1 - UNIX "troff" format - Witten, Neal, Cleary: Arithmetic Coding for Data Compression.
   82,199 PAPER2 - UNIX "troff" format - Witten: Computer (in)security.
  513,216 PIC    - 1728 x 2376 bitmap image (MSB first): text in French and a line graph.
   39,611 PROGC  - Source code in C - UNIX compress v4.0.
   71,646 PROGL  - Source code in Lisp - system software.
   49,379 PROGP  - Source code in Pascal - program to evaluate PPM compression.
   93,695 TRANS  - ASCII and control characters - transcript of a terminal session.
```

Early tests sometimes used an 18 file version of the corpus that included 4 addtional papers (PAPER3 through PAPER6). Programs were often ranked by measuring bits per character (bpc) on each file separately and reporting them individually or taking the average. Simply adding the compressed sizes is called a "weighted average" since it is weighted toward the larger files.

The Calgary corpus is no longer widely used due to its small size. However, it has been used since 1996 in an ongoing compression challenge run by Leonid A. Broukhis with small cash prizes. The best compression ratios established as of Feb. 2010 are as follows.

Table. Calgary Compression Challenge History

Size	Date	Name
759,881	Sep 1997	Malcolm Taylor
692,154	Aug 2001	Maxim Smirnov
680,558	Sep 2001	Maxim Smirnov
653,720	Nov 2002	Serge Voskoboynikov
645,667	Jan 2004	Matt Mahoney
637,116	Apr 2004	Alexander Ratushnyak
608,980	Dec 2004	Alexander Ratushnyak
603,416	Apr 2005	Przemyslaw Skibinski
596,314	Oct 2005	Alexander Ratushnyak
593,620	Dec 2005	Alexander Ratushnyak
589,863	May 2006	Alexander Ratushnyak

The rules of the Calgary challenge specify that the compressed size include the size of the decompression program, either as a Windows or Linux executable file or as source code. This is to avoid programs that cheat by hiding information from the corpus in the decompression program. Furthermore, the program and compressed files must either be packed in an archive (in one of several specified formats), or else 4 bytes plus the length of each file name is added. This is to prevent cheating by hiding information in the file names and sizes. Without such precautions, programs like barf could claim to compress to zero bytes.

Submissions prior to 2004 are custom variants of compressors by the authors based on PPM algorithms (rk for Taylor, slim for Voskoboynikov, ppmn for Smirnov). Subsequent submissions are variants of the open source paq6, a context mixing algorithm. For comparison, zip -9 (InfoZIP 2.32., option -9 for best compression) compresses the Calgary corpus to 1,020,495 bytes, not including the size of the unzip program.

2.2. Large Text Compression Benchmark

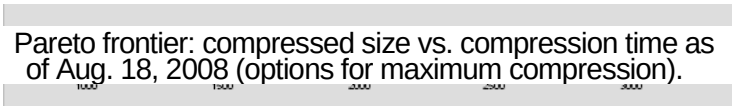
The Large Text Compression Benchmark consists of a single Unicode encoded XML file containing a dump of Wikipedia text from Mar. 3, 2006, truncated to 10^9 bytes. Its stated goal is to encourage research into artificial intelligence, specifically, natural language processing. As of Feb. 2010, 128 different programs (889 including different versions and options) were evaluated for compressed size (including the decompression program source or executable and any other needed files as a zip archive), speed, and memory usage. The benchmark is open, meaning that anyone can submit results.

Programs are ranked by compressed size with options selecting maximum compression where applicable. The best result obtained is 127,784,888 bytes by D. Shkarin for a customized version of durilca using 13 GB memory. It took

1398 seconds to compress and 1797 seconds to decompress using a size-optimized decompression program on a 3.8 GHz quad core Q9650 with 16 GB memory under 64 bit Windows XP Pro on July 21, 2009. The data was preprocessed with a custom dictionary built from the benchmark and encoded with order 40 PPM. durilca is a modified version of ppmonstr by the same author. ppmonstr is in turn a slower but better compressing ppmd program which is used for maximum compression in several archivers such as rar, WinZip, 7zip, and freearc.

By comparison, zip -9 compresses to 322,649,703 bytes in 104 seconds and decompresses in 35 seconds using 0.1 MB memory. It is ranked 92'nd.

The benchmark shows a 3 way trade off between compressed size, speed, and memory usage. The two graphs below show the Pareto frontier, those compressors for which no other compressors both compress smaller and faster (or smaller and use less memory). The graphs are from Aug. 2008, but the current data shows a similar trend. In particular, no single algorithm (shown in parenthesis) is the "best".



Pareto frontier: compressed size vs. compression time as of Aug. 18, 2008 (options for maximum compression).

Pareto frontier: compressed size vs. memory as of Aug. 18, 2008 (options for maximum compression).

Note that speed tests may be run on different machines, and that only the options for maximum compression for each program are used. Nevertheless, the general trend remains valid. Individual compressors often have options that allow the user to make the same 3 way trade off.

2.3. Hutter Prize

The Hutter prize is based on the first 10^8 bytes (the file enwik8) of the Large Text Compression benchmark with similar rules and goals. It is a contest in which prize money (500 euros per 1% gain) is awarded for improvements of 3% or more over the previous submission, subject to time and memory limits on the test computer. The best result is 15,949,688 bytes for an archive and a decompresser submitted by A. Ratushnyak on May 23, 2009. It requires 7608 seconds and 936 MB memory to decompress on a 2 GHz dual core T3200 under 32 bit Windows Vista. The submission is based on two open source, context mixing programs paq8hp12 and lpaq9m with a custom dictionary for preprocessing.

By comparison, zip -9 compresses the same data to 36,445,373 bytes and uncompresses in 3.5 seconds using 0.1 MB memory.

2.4. Maximum Compression

The maximum compression benchmark has two parts: a set of 10 public files totaling 53 MB, and a private collection of 510 files totaling 301 MB. In the public data set (SFC or single file compression), each file is compressed separately and the sizes added. Programs are ranked by size only, with options set for best compression individually for each file. The set consists of the following 10 files:

```
  842,468 a10.jpg       - a high quality 1152 x 864 baseline JPEG image
of a fighter jet.
 3,870,784 acrord32.exe - x86 executable code - Acrobat Reader 5.0.
 4,067,439 english.dic  - an alphabetically sorted list of 354,941
English words.
 4,526,946 FlashMX.pdf  - PDF file with embedded JPEG and zipped BMP
images.
20,617,071 fp.log       - web server log, ASCII text.
 3,782,416 mso97.dll    - x86 executable code from Microsoft Office.
 4,168,192 ohs.doc      - Word document with embedded JPEG images.
 4,149,414 rafale.bmp   - 1356 x 1020 16 bit color image in 24 bit RGB
format.
 4,121,418 vcfiu.hlp    - OCX help file - binary data with embedded
text.
 2,988,578 world95.txt  - ASCII text - 1995 CIA World Factbook.
```

The top ranked program as of Dec. 31, 2009 with a total size of 8,813,124 bytes is paq8px, a context mixing algorithm with specialized models for JPEG images, BMP images, x86 code, text, and structured binary data. WinRK 3.1.2, another context mixing algorithm, is top ranked on 4 of the files (txt, exe, dll, pdf). WinRK uses a dictionary which is not included in the total size. 208 programs are ranked. zip 2.2 is ranked 163 with a size of 14,948,761.

In the second benchmark or MFC (multiple file compression), programs are ranked by size, compression speed, decompression speed, and by a formula that combines size and speed with time scaled logarithmically. The data is not available for download. Files are compressed together to a single archive. If a compressor cannot create archives, then the files are collected into an uncompressed archive (TAR or QFC), which is compressed.

In the MFC test, paq8px is top ranked by size. freearc is top ranked by combined score, followed by nanozip, winrar, and 7zip. All are archivers that detect file type and apply different algorithms depending on type.

2.5. Generic Compression Benchmark

The Generic Compression Benchmark has the goal of evaluating compression algorithms in the context of universal prediction or intelligence, as defined by Legg and Hutter (2006). By this definition, data sources are assumed to have a universal Solomonoff distribution, i.e. generated by random programs with a preference for smaller or simpler programs. The evidence for such a distribution is the success of applying Occam's Razor to machine learning and to science in general: the simplest theories that fit the observed data tend to be the best predictors. The purpose of the test is to find good compression algorithms that are not tuned to specific file types.

The benchmark does not publish any test data. Rather, it publishes a program to generate the data from a secret seed or an internally hardware generated random number. The data consists of the bit string outputs of one million random Turing machines, truncated to 256 bits and packed into null terminated byte strings. The average output size is about 6.5 MB. The test allows public verification while eliminating the need to measure the decompresser size because it is not possible to hide the test data in the decompresser without knowing the cryptographic random number seed. The test produces repeatable results with about 0.05% accuracy. Programs are ranked by the ratio of compressed output to the compressed output of a reference compressor (ppmonstr) to improve repeatability.

Unfortunately the benchmark fails to completely eliminate the problem of tuning compressors to public benchmarks. The top ranked program is a stationary context mixing model configuration implemented in zpaq using a preprocessor by J. Ondrus that splits each string into an incompressible prefix and a bit repeat instruction. Its score is 0.8750, compared to 1.3124 for zip -9. Generally, however, the rank order of compressors is similar to that of other benchmarks.

2.6. Compression Ratings

Compression Ratings by Sami Runsas ranks programs on 5.4 GB of various data types from public sources using a score that combines size and speed. The data includes English text, Windows executable code, RGB and grayscale images, CD quality audio, and a mix of data types from two video games. None of the test data is compressed (JPEG, PDF, ZIP, etc). Each of the 10 data sets contains multiple files. Single file compressors are tested on an equivalent tar file. Programs must pass a qualifying round with minimum compression ratio and time requirements on a small data set. The benchmark includes a calculator that allows the user to rank compressors using different weightings for the importance of size, compression speed, and decompression speed. The default scale is "1/10 lower ratio or twice the total time is worth half the rating". This is effectively the same formula used by maximumcompression.com. Compressed sizes include the size of the (not compressed) Windows executable decompression program. As of Feb. 2010, 427 qualifying combinations of program versions and options were tested. The top ranked programs for the default settings were nanozip followed by freearc, CCM, flashzip, and 7-zip. Runsas is also the author of nanozip.

2.7. Other Benchmarks

Some other benchmarks are mentioned briefly.

Squeeze Chart by Stephen Busch, ranks programs on 6.4 GB of mostly private data of various types by size only. The top ranked is paq8px_v67 as of Dec. 28, 2009.

Monster of Compression by Nania Francesco Antonio, ranks programs by size on 1,061,420,156 bytes of mostly public data of various types with a 40 minute time limit. There are separate tests for single file compressors and archivers. As of Dec. 20, 2009 the top ranked archiver is nanozip 0.7a and the top ranked file compressor is ccmx 1.30c. Both use context mixing.

UCLC by Johan de Bock contains several benchmarks of public data for compressors with a command line interface (which is most of them). As of Feb. 2009, paq8i or paq8p was top ranked by size on most of them.

Metacompressor is an automated benchmark that allows developers to submit programs and test files and compare size (excluding the decompresser), compression time, and decompression time.

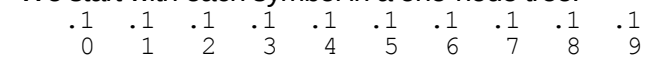
3. Coding

A code is an assignment of bit strings to symbols such that the strings can be decoded unambiguously to recover the original data. The optimal code for a symbol with probability p will have a length of $\log_2 1/p$ bits. Several efficient coding algorithms are known.

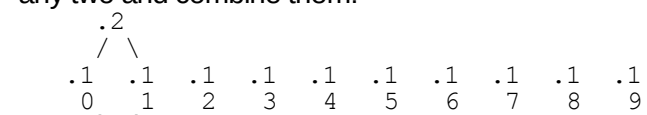
3.1. Huffman Coding

Huffman (1952) developed an algorithm that calculates an optimal assignment over an alphabet of n symbols in $O(n \log n)$ time. deflate (zip) and bzip2 use Huffman codes. However, Huffman codes are inefficient in practice because code lengths must be rounded to a whole number of bits. If a symbol probability is not a power of 1/2, then the code assignment is less than optimal. This coding inefficiency can be reduced by assigning probabilities to longer groups of symbols but only at the cost of an exponential increase in alphabet size, and thus in run time.

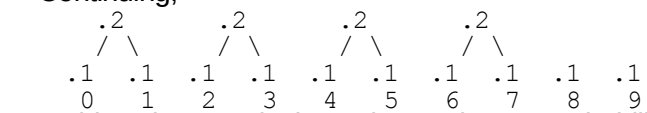
The algorithm is as follows. We are given an alphabet and a probability for each symbol. We construct a binary tree by starting with each symbol in its own tree and joining the two trees that have the two smallest probabilities until we have one tree. Then the number of bits in each Huffman code is the depth of that symbol in the tree, and its code is a description of its path from the root (0 = left, 1 = right). For example, suppose that we are given the alphabet {0,1,2,3,4,5,6,7,8,9} with each symbol having probability 0.1. We start with each symbol in a one-node tree:



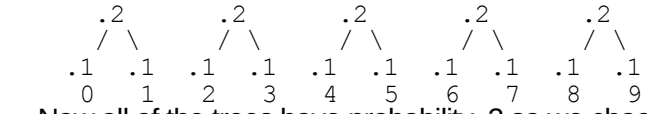
Because each small tree has the same probability, we pick any two and combine them:



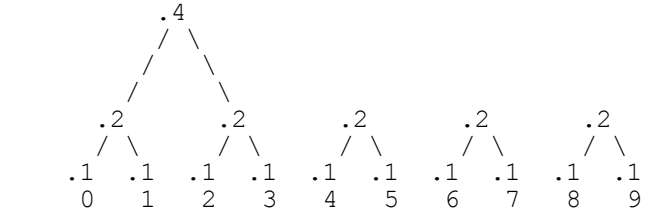
Continuing,



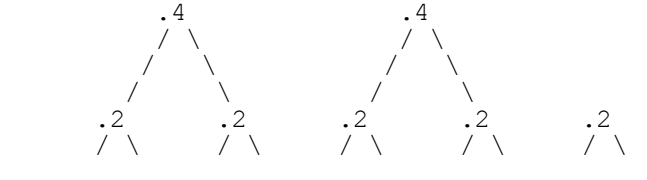
At this point, 8 and 9 have the two lowest probabilities so we have to choose those:

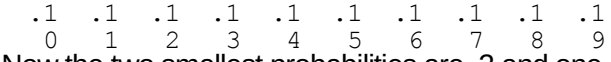


Now all of the trees have probability .2 so we choose any pair of them:

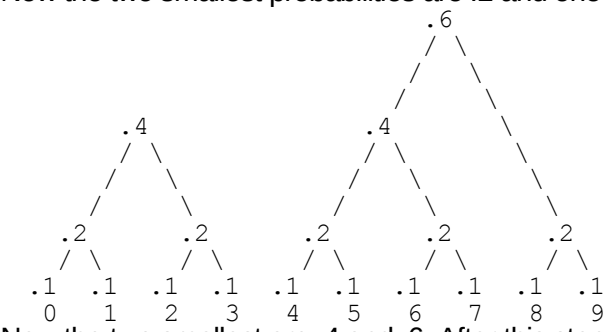


We choose any two of the three remaining trees with probability .2:

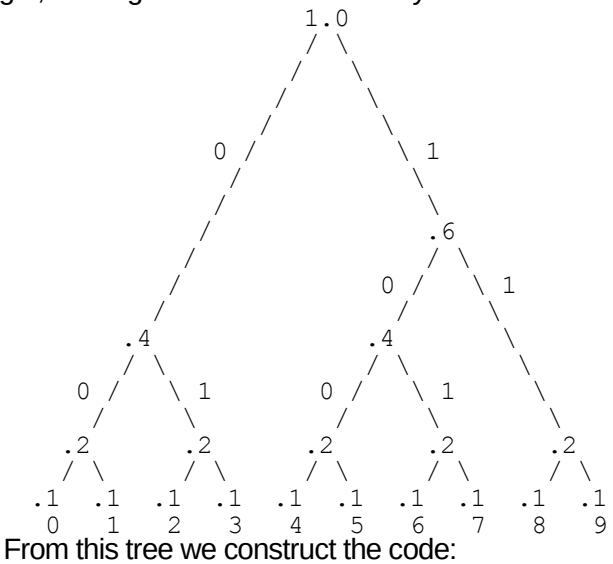




Now the two smallest probabilities are .2 and one of the .4:



Now the two smallest are .4 and .6. After this step, the tree is finished. We can label the branches 0 for left and 1 for right, although the choice is arbitrary.



From this tree we construct the code:

Symbol	Code
0	000
1	001
2	010
3	011
4	1000
5	1001
6	1010
7	1011
8	110
9	111

A code may be static or dynamic. A static code is computed by the compressor and transmitted to the decompresser as part of the compressed data. A dynamic code is computed by the compressor and periodically updated, but not transmitted. Instead, the decompresser reconstructs the code using exactly the same algorithm using the previously decoded data to estimate the probabilities. Neither method compresses better because any space saved by not transmitting the model is paid back by having less data with which to estimate probabilities.

Huffman codes are typically static, mainly for speed. The compressor only needs to compute the code once, using the entire input to compute probabilities. To transmit a Huffman table, it is only necessary to send the size of each symbol, for example: (3,3,3,3,4,4,4,4,3,3). Both the compressor and decompresser would then assign codes by starting with the shortest symbols, counting up from 0, and appending a 0 bit whenever the code gets longer. This would result in the following different but equally effective code:

Symbol	Size	Code
0	3	000
1	3	001
2	3	010
3	3	011
8	3	100
9	3	101
4	4	1100
5	4	1101
6	4	1110
7	4	1111

For file compression, Huffman coded data still needs to be packed into bytes. JPEG packs bits in MSB (most significant bit) to LSB (least significant bit) order. For example, the codes `00001 00111` would be packed as `00001001 11......`. The deflate format used in zip, gzip, and png files packs bits in LSB to MSB order, as if each byte is written backward, i.e. `10010000 ......11`.

One other complication is that the last byte has to be padded in such a way that it is not interpreted as a Huffman code. JPEG does this by not assigning any symbol to a code of all 1 bits, and then padding the last byte with 1 bits. Deflate handles this by reserving a code to indicate the end of the data. This tells the decoder not to decode the remaining bits of the last byte.

3.2. Arithmetic Coding

Huffman coding has the drawback that code lengths must be a whole number of bits. This effectively constrains the model to probabilities that are powers of 1/2. The size penalty for modeling errors is roughly proportional to the square of the error. For example, a 10% error results in a 1% size penalty. The penalty can be large for small codes. For example, the only possible ways to Huffman code a binary alphabet is to code each bit as itself (or its opposite), resulting in no compression.

Arithmetic coding (Rissanen, 1976), also called range coding, does not suffer from this difficulty. Let P be a model, meaning that for any string x, P(x) is the probability of that string. Let P(< x) be the sum of the probabilities of all strings lexicographically less than x. Let P(≤ x) = P(< x) + P(x). Then the arithmetic code for a string x is the shortest binary number y such that P(< x) ≤ y < P(≤ x). Such a number can always be found that is no more than 1 bit longer than the Shannon limit $\log_2 1/P(x)$.

An arithmetic code can be computed efficiently by expressing P(x) as a product of successive symbol predictions by the chain rule, $P(x) = \Pi_i P(x_i \mid x_1x_2...x_{i-1})$ where x_i means the i'th symbol (bit or character) in x. Then the arithmetic code can be computed by updating a range [low, high) (initially [0, 1)) for each symbol by dividing the range in proportion to the probability distribution for that symbol. Then the portion of the range corresponding to the symbol to be coded is used to update the range. As the range shrinks, the leading bits of low and high match and can be output immediately because the code y must be between them. The decompresser is able to decode y by making an identical sequence of predictions and range reductions.

Most modern data compressors use arithmetic coding. Early compressors used Huffman coding because arithmetic coding was patented and because its implementation required multiplication operations, which was slow on older processors. Neither of these issues are relevant today because the important patents have expired and newer processors have fast multiply instructions (faster than memory access).

The most common arithmetic coders code one byte at a time (PPM) or one bit at a time (CM, DMC). Free source code with no licensing restrictions for a bytewise encoder can be found in the source code for ppmd (D. Shkarin). Bitwise coders licensed under GPL can be found in the source code for PAQ based compressors including FPAQ, LPAQ, and ZPAQ, the BWT compressor BBB, and the symbol ranking compressor SR2. The simplest of these is the order 0 coder fpaq0.

The following is the arithmetic coder from zpaq 1.10 It encodes bit y (0 or 1) with probability p = P(1) * 65536 (a scaled 16 bit number) and codes to FILE* out. The encoder range is represented by two 32 bit integers (unsigned int) low and high, which are initially 1 and 0xffffffff respectively. After the range is split, a 1 is coded in the lower part of the range and a 0 in the upper part. After the range is split and reduced, it is normalized by shifting out any leading bytes that are identical between low and high. The low bits shifted in are all 0 bits for low and all 1 bits for high.

```
  // Initial state
unsigned int low = 1, high = 0xffffffff;

// Encode bit y with probability p/65536
inline void Encoder::encode(int y, int p) {
  assert(out);  // output file
  assert(p>=0 && p<65536);
  assert(y==0 || y==1);
  assert(high>low && low>0);
  unsigned int mid=low+((high-low)>>16)*p+((((high-
low)&0xffff)*p)>>16); // split range
  assert(high>mid && mid>=low);
  if (y) high=mid; else low=mid+1; // pick half
  while ((high^low)<0x1000000) { // write identical leading bytes
    putc(high>>24, out);  // same as low>>24
    high=high<<8|255;
    low=low<<8;
    low+=(low==0); // so we don't code 4 0 bytes in a row
  }
}
```

The range split is written to avoid 32 bit arithmetic overflow. It is equivalent to:

```
  unsigned int mid=low+((unsigned long
long)(high-low)*p>>16);
```

where (unsigned long long) is a 64 bit unsigned type, which not all compilers support. The initialization of low to 1 instead of 0 and the statement

```
  low+=(low==0);
```

discard a tiny bit of the range to avoid writing 4 consecutive 0 bytes, which the ZPAQ compressor uses to mark the end of the encoded data so it can be found quickly without decoding. This is not a requirement in general. It is not used in the rest of the PAQ series. The decoder looks like this:

```
  // Initial state
unsigned int low=1, high=0xffffffff, curr;
for (int i=0; i<4; ++i)
  curr=curr<<8|getc(in);  // first 4 bytes of input


// Return decoded bit from file 'in' with probability p/65536
inline int Decoder::decode(int p) {
  assert(p>=0 && p<65536);
  assert(high>low && low>0);
  if (curr<low || curr>high) error("archive corrupted");
  assert(curr>=low && curr<=high);
  unsigned int mid=low+((high-low)>>16)*p+((((high-
low)&0xffff)*p)>>16); // split range
  assert(high>mid && mid>=low);
  int y=curr<=mid;
  if (y) high=mid; else low=mid+1; // pick half
  while ((high^low)<0x1000000) { // shift out identical leading bytes
    high=high<<8|255;
    low=low<<8;
    low+=(low==0);
    int c=getc(in);
    if (c==EOF) error("unexpected end of file");
    curr=curr<<8|c;
  }
  return y;
}
```

The decoder receives as input p, the 16 bit probability that the next bit is a 1, and returns the decoded bit. The decoder has one additional variable in its state, the 32 bit integer curr, which is initialized to the first 4 bytes of compressed data. Each time the range is rescaled, another byte is read from FILE* in. high and low are initialized as in the encoder.

One additional detail is how to handle the end of file. Most of the PAQ series compressors encode the file size separately and perform 8 encoding operations per byte. After the last encoding operation, the 4 bytes of either high or low or some value in between must be flushed to the archive because the decoder will read these 4 bytes in.

ZPAQ encodes 9 bits per byte, using a leading 1 bit modeled with probability p = 0 to mark the end of file. The effect on the encoder is to set mid = low and cause 4 bytes to be flushed. After that, 4 zero bytes are written to mark the end of the compressed data. When the end of file bit is decoded, the decoder reads these 4 zero bytes into curr, resulting in low = 1, curr = 0, high = 0xffffffff. Any further decoding would result in an error because the condition low ≤ curr ≤ high fails.

The assertions are not necessary to make the code work. Rather, they are useful because they make the programmer's assumptions explicit in the form of self documenting run time tests. The code is compiled with -DNDEBUG to remove them after debugging.

3.3. Asymmetric Binary Coding

Most high end compressors use arithmetic coding. However, another possibility with the same theoretical coding

and time efficiency for bit strings is asymmetric binary coding or ABC (Duda, 2007). An asymmetric coder has a single n-bit integer state variable y, as opposed to two variables (low and high) in an arithmetic coder. This allows a lookup table implementation. A bit y (0 or 1) with probability p = P(y = 1) (0 < p < 1, a multiple of 2^{-n}) is coded in state x, initially 2^n:

```
if y = 0 then x := ceil((x+1)/(1-p)) - 1
if y = 1 then x := floor(x/p)
```

To decode, given x and p

```
y = ceil((x+1)*p) - ceil(x*p)  (0 if fract(x*p) < 1-p, else 1)
if y = 0 then x := x - ceil(x*p)
if y = 1 then x := ceil(x*p)
```

x is maintained in the range 2^n to 2^{n+1} - 1 by writing the low bits of x prior to encoding y and reading into the low bits of x after decoding. Because compression and decompression are reverse operations of each other, they must be performed in reverse order by storing the predictions and coded bits in a stack in either the compressor or the decompresser.

The coder is implemented in the order-0 compressors fpaqa, fpaqb, and fpaqc and the context mixing compressor lpaq1a from the PAQ series. fpaqa uses lookup tables for both compression and decompression. It uses a 10 bit state and the probability is quantized to 7 bits on a nonlinear scale (finer near 0 and 1 and coarse near 1/2). The stack size is 500,000. Increasing these numbers would result in better compression at the expense of a larger table. fpaqb uses direct calculations except for division, which uses a table of inverses because division is slow. fpaqc is fpaqb with some speed optimizations. The coder for fpaqc is used in lpaq1a (a context mixing program), replacing the arithmetic coder in lpaq1.

Although a lookup table implementation might be faster on some small processors, it is slower on modern computers because multiplication is faster than random memory access. Some benchmarks are shown for enwik8 (100 MB text) on a 2.0 GHz T3200 processor running on one of two cores. Ratio is fraction of original size. Compression and decompression times are nanoseconds per byte.

	Ratio	Comp	Decomp	Coder
fpaqa	.61310	247	238	ABC lookup table
fpaqb	.61270	244	197	ABC direct calculation
fpaqc	.61270	246	173	ABC direct calculation
fpaqa -DARITH	.61280	130	112	arithmetic (fpaqa compiled with -DARITH)

For high end compressors, CPU time and memory are dominated by the model, so the choice of coder makes little difference. lpaq1 is a context mixing compressor, a predecessor of lpaq9m, ranked third of 127 on the large text benchmark as of Feb. 2010. lpaq1a is the same except that the arithmetic coder was replaced by the asymmetric binary coder from fpaqb. (Timed on a 2.188 GHz Athlon-64 3500+).

	Ratio	Comp	Decomp	Coder
lpaq1a	.19755	3462	3423	ABC direct calculation (fpaqb)
lpaq1	.19759	3646	3594	arithmetic

The arithmetic coder in lpaq1 and fpaqa -DARITH compresses slightly worse than the ABC coder because it uses 12 bits of precision for the probability, rather than 16 bits as in ZPAQ.

3.4. Numeric Codes

Numeric codes are Huffman codes for numbers with some common probability distributions, such as might appear in coding prediction errors in images or audio, or run lengths in highly redundant data. Generally such codes have simpler descriptions to transmit to the decoder than a full Huffman code length table.

3.4.1. Unary Codes

A unary code for a non negative integer n is n ones followed by a zero (or n zeros followed by a one). For example, 5 would be coded as 111110. A unary code implies a probability $P(n) = 1/2^{n+1} = 1/2, 1/4, 1/8, 1/16$, etc.

Signed integers can be encoded by mapping positive n to 2n and negative n to -2n-1 to form the sequence 0, -1, 1, -2, 2, -3, 3, etc.

3.4.2. Rice and Golomb Codes

A Golomb code for non negative integer n with parameter M divides n into a quotient q = floor(n/M) and and a remainder r = n - Mq. Then q is encoded in unary and r in binary. If M is a power of 2, then the code is called a "Rice code", and all of the remainers are coded in $\log_2$ M bits. Otherwise, the smaller remainders are coded using floor($\log_2$ M) bits and the larger ones with one more bit. A Golomb code with M = 1 is just a unary code. For example:

n	M=1 (unary) q r code	M=2 (Rice) q r code	M=3 (Golomb) q r code	M=4 (Rice) q r code
0	0 0 0	0 0 00	0 0 00	0 0 000
1	1 0 10	0 1 01	0 1 010	0 1 001
2	2 0 110	1 0 100	0 2 011	0 2 010
3	3 0 1110	1 1 101	1 0 100	0 3 011
4	4 0 11110	2 0 1100	1 1 1010	1 0 1000
5	5 0 111110	2 1 1101	1 2 1011	1 1 1001
6	6 0 1111110	3 0 11100	2 0 1100	1 2 1010

A Golomb code with parameter M increases in length once every M values. Thus, it implies an approximate geometric distribution where n has probability proportional to $2^{-n/M}$.

3.4.3. Extra Bit Codes

An "extra bit" code uses a Huffman code to encode a range of numbers, then encodes a value within that range (the extra bits) in binary. This method can be used to approximate any distribution that is approximately smoothly varying.

JPEG uses extra bit codes to encode discrete cosine transform coefficients. The ranges are 0, 1, 2..3, 4..7, 8..15, increasing by powers of 2. It also uses a sign bit for all of the ranges except 0. The deflate format used in zip and gzip uses extra bit codes to encode match lengths and offsets using a more complex set of ranges.

A floating point number is a special case of an extra-bit code where the reals are approximated by mapping to integers on a logarithmic scale, and where each range is the same size and equally probable. A IEEE 64 bit floating point number (type double in C/C++) has one sign bit s representing -1 or 1, an 11 bit exponent e representing a number in the range -1022 to +1023, and a 52 bit mantissa m representing a number in the range 1/2 to 1 (or 0 to 1 for the smallest exponent). It encodes the real number $sm2^e$. A μ-law or mu-law code, used to encode sampled telephone speech in North America and Japan, is an 8 bit floating point number with one sign bit, 3 exponent bits and 4 mantissa bits.

3.5. Archive Formats

Compressed files are stored in archives. An archive may contain a single file or multiple files. For example, gzip compresses a file, adds a **.gz** extention to the name and deletes the original. Decompression restores the original and deletes the compressed file. zip stores multiple files in an archive with a **.zip** file name extension. It has commands to list, add, update, delete, and extract files from the archive (without deleting the input files). However, both programs use the same compression algorithm.

Compression can sometimes be improved by compressing similar files together to take advantage of mutual information. Some archivers support this using a "solid" format, which requires that all of the files be compressed or extracted at the same time. zip is optimized for speed, so its archives are not solid. PAQ is optimized for maximum compression, so archives are always solid. It is optional for some archivers such as WinRAR.

A file compressor such as gzip can be used to create solid archives by first collecting the input files together using a non-compressing archiver such as tar and then compressing. The combination of tar + gzip is commonly given a file name extension of **.tar.gz** or **.tgz**.

The tar format was originally designed for UNIX or Linux backups on tape. Thus, it is possible to make a tar file of an entire file system and restore its structure unchanged. The tar program appends a 512 byte header to each file containing the file name, path, size, timestamp, owner, group, permissions, and padding with zero bytes. Then the files are all appended together. The files are also padded

with zero bytes to a multiple of 512. These zero bytes are easily compressed.

Archivers zip, WinRAR, and 7-zip have features similar to tar in that they preserve timestamps and can compress and extract directory trees. However they do not store owner, group, and permission information in order to be compatible with Windows, which has a different security model.

3.5.1. Error Detection

Archivers sometimes add a checksum to detect transmission errors. The checksum is computed and stored before compression and verified after decompression. The need for a checksum is questionable because networks and storage media typically use error correction so that errors are very unlikely. Decompression errors are more likely to result from software bugs or deliberate manipulation. A single bit change in an archive can produce a completely different file. A checksum increases the archive size slightly (usually by 4 bytes) and also increases compression and decompression time by the time it takes to compute it. Some common checksum functions:

3.5.1.1. Parity

The checksum is computed by counting the number of 1 bits in the input stream. The checksum is 1 if the total is odd or 0 if even. All single bit errors are detected, but other errors are detected only with probability 1/2.

3.5.1.2. CRC-32

CRC-32 is a commonly used cyclic redundancy check. It detects all burst errors up to 31 bits and all other errors with probability $1 - 2^{-32}$.

The input is a stream of bits which are shifted through a 32 bit window. Each time a 1 bit is shifted out, the new window contents is XORed with 0x04C11DB7, otherwise it is unchanged. The final contents of the window is the 32 bit checksum. A fast implementation will shift and XOR one byte at a time using a 256 by 32 bit lookup table.

3.5.1.3. Adler-32

The Adler-32 checksum is used in the zlib implementation of deflate, which is used in zip and gzip. It is a 32 bit checksum which detects errors with probability of $1 - 65521^{-2}$, which is almost $1 - 2^{-32}$.

The checksum has two 16-bit parts. The first part is 1 plus the sum of all input bytes, modulo 65521. The second part is the sum of all the intermediate sums of the first part, modulo 65521. The computation is very fast because the (slow) mod operation only needs to be calculated infrequently using 32 bit arithmetic. 65521 is the largest prime number less than 2^{16}.

3.5.1.4. Cryptographic Hash Functions

CRC-32 and Adler-32 detect most accidental errors but it is easy to deliberately alter the input without changing the checksum. Cryptographic hash functions are designed to resist even deliberate attempts to find collisions (two different inputs that produce the same checksum). They are designed so that the best possible algorithms are no better than randomly guessing the inputs and testing the outputs. The disadvantage is that they are larger (usually 20 to 32 bytes) and take longer to compute (although still reasonably fast).

ZPAQ optionally adds a 160 bit (20 byte) SHA-1 checksum. It detects errors with probability $1 - 2^{-160}$, even if deliberately introduced. The checksum calculation adds 5.6 ns per byte on a 2 GHz T3200, compared to 5.0 ns for CRC-32 and 5.3 ns for Adler-32 using the SlavaSoft fsum v2.51 implementation and testing on a 1 GB text file.

3.5.2. Encryption

Some archives will encrypt data as well as compress it. I do not recommend adding this feature unless you know what you are doing. It is tempting to design your own algorithm, perhaps combining it with compression by altering the initial state of the model in a way that depends on a password. Such an attempt will almost certainly fail. Just because the output looks random doesn't mean it is secure. There is no test for randomness.

If you add encryption, use a well tested algorithm such as AES. Compress your data before encryption, because encrypted data cannot be compressed. Use published encryption code. Don't write it yourself. Then publish *all* your source code and have security experts look at it. Others will try to break your code, and you should make their job as easy as possible by clearly documenting what your code does and how it works. Just because you use a secure, well tested algorithm doesn't mean the rest of your program is secure. For example, if your archiver deletes the plaintext, it might still be recovered from unallocated disk sectors or the swap file. Likewise for the password or password derived data in memory that gets swapped to disk. There are many subtle things that can go wrong.

Do not believe that a secret algorithm or an executable only release is more secure. An encryption algorithm is not considered secure unless it can withstand chosen plaintext attacks by adversaries with full knowledge of the algorithm. Most security experts will not even waste their time trying to attack a closed system. They will just use other, well tested solutions. If your system ever does become popular, then others *will* reverse engineer it and they *will* break it.

4. Modeling

A model is an estimate of the probability distribution of inputs to a compressor. Usually this is expressed as a sequence of predictions of successive symbols (bits, bytes, or words) in the input sequence given the previous input as context. Once we have a model, coding is a solved problem. But (as proved by Kolmogorov) there is no algorithm for determining the best model. This is the hard problem in data compression.

A model can be static or adaptive. In the static case, the compressor analyzes the input, computes the probability distribution for its symbols, and transmits this data to the decompresser followed by the coded symbols. Both the compressor and decompresser select codes of the appropriate lengths using identical algorithms. This method is often used with Huffman coding.

Typically the best compressors use dynamic models and arithmetic coding. The compressor uses past input to estimate a probability distribution (prediction) for the next symbol without looking at it. Then it passes the prediction and symbol to the arithmetic coder, and finally updates the model with the symbol it just coded. The decompresser makes an identical prediction using the data it has already decoded, decodes the symbol, then updates its model with the decoded output symbol. The model is unaware of whether it is compressing or decompressing. This is the technique we will use in the rest of this chapter.

4.1. Fixed Order Models

The simplest model is a fixed order model. An *order n* model inputs the last n bytes or symbols (the context) into a table and outputs a probability distribution for the next symbol. In the update phase, the predicted symbol is revealed and the table is updated to increase its probability when the same context next appears. An order 0 model uses no context.

4.1.1. Bytewise Encoding

A probability distribution is typically computed by using a counter for each symbol in the alphabet. If the symbols are bytes, then the size of the alphabet is 256. The prediction for each symbol is the count for that symbol divided by the total count. The update procedure is to increment the count for that symbol. If the arithmetic coder codes one byte at a time, then you pass the array of counts and the total to the arithmetic coder. For compression, you also pass the byte to be coded. For decompression, it returns the decoded byte. The procedure looks like this:

```
 const int CONTEXT_SIZE = 1 << (n*8);   // for order n
int count[CONTEXT_SIZE][256] = {{0}};  // symbol counts
int context = 0;                       // last n bytes packed together

// Update the model with byte c
void update(int c) {
  ++count[context][c];
  context = (context << 8 | c) % CONTEXT_SIZE;
}

// compress byte c
void compress(int c) {
  encode(count[context], c);  // predict and encode
```

```
  update(c);
}

// decompress one byte and return it
int decompress() {
  int c = decode(count[context]);
  update(c);
  return c;
}
```

The functions encode() and decode() are assumed to be encoding and decoding procedures for a bytewise arithmetic coder. They each take an array of 256 counts and divide the current range in proportion to those counts. encode() then updates the range to the c'th subrange. decode() reads the compressed data and determines that it is bounded by the c'th range and returns c. The update() procedure stores the last n bytes in the low bits of the context.

There are a few problems with this method. First, what happens if a byte has a probability of zero? An ideal encoder would give it a code of infinite size. In practice, the encoder would fail. One fix is to initialize all elements of count to 1. Sometimes it improves compression if the initial count were smaller, say 0.5 or 0.1. This could be done effectively by increasing the increment to, say, 2 or 10.

A second problem is that a count can eventually overflow. One solution is that when a count becomes too large, to rescale all of the counts by dividing by 2. Setting a small upper limit (typically 30 to several hundred) can improve compression of data with blocks of mixed types (like text with embedded images) because the statistics reflect recent input. This is an *adaptive* or *nonstationary* model. Data with uniform statistics such as pure text are compressed better with *stationary* models, where the counts are allowed to grow large. In this case, probabilities depend equally on all of the past input.

A third problem is that the table of counts grows exponentially with the context order. Some memory can be saved by changing count[] to unsigned char and limiting counts to 255. Another is to replace the context with a hash, for example:

```
const int k = 5;  // bits of hash per byte
const int CONTEXT_SIZE = 1 << (n*k);  // order n
...
context = (context * (3 << k) + c) % CONTEXT_SIZE; // update context hash
```

The multiplier can be any odd number left shifted by another number k in the range 1 through 8. Then the last nk bits of context depend on the last n bytes of input. A larger k will result in better compression at the expense of more memory.

A fourth problem is that straightforward bytewise arithmetic coding is inefficient. The decoder must compute 256 range intervals to find the one containing the compressed data. This could be solved by using cumulative counts, i.e. count[context][c] is the sum of counts for all byte values ≤ c, but that only moves the inefficiency to the update() function, which must increment up to 256 values. One solution is to encode one bit at a time using the bitwise encoder like the one described in section 3.2.

4.1.2. Bitwise encoding

The idea is to encode one bit at a time by using the previous bits of the current byte as additional context. Only two values are stored: a count of ones, $count_1$, and a total count. The prediction is $count_1$/count. The update procedure is to increment count and to increment $count_1$ if the bit is 1. We handle zero probabilities, overflow, and large contexts as before.

Alternatively, we can avoid a (slow) division operation by storing the prediction directly. Each bitwise context is associated with a prediction that the next bit will be a 1 (initially 1/2) and an update count (initially 0). The update rate is initially fast and decreases as the count is incremented, resulting in a stationary model. Alternatively, the count can be bounded, resulting in an adaptive model.

```
// Prediction and count for one bitwise context
struct Model {
  double prediction;  // between 0 and 1 that next bit will be a 1
  int count;          // number of updates
  Model(): prediction(0.5), count(0) {}
};

Model model[CONTEXT_SIZE][256]; // context, bit_context -> prediction and count
int context = 0;                // bytewise order n context

// Compress byte c in MSB to LSB order
void compress(int c) {
  for (int i=7; i>=0 --i) {
    int bit_context = c+256 >> i+1;
    int bit = (c >> i) % 2;
    encode(bit, model[context][bit_context].prediction);
    update(bit, model[context][bit_context]);
  }
  context = (context << 8 | c) % CONTEXT_SIZE;
}

// Decompress and return a byte
int decompress() {
  int c;  // bit_context
  int bit;
  for (c = 1; c < 256; c = c * 2 + bit) {
    bit = decode(model[context][c].prediction);
    update(bit, model[context][c]);
  }
  c -= 256;  // decoded byte
  context = (context << 8 | c) % CONTEXT_SIZE;
  return c;
}

// Update the model
void update(int bit, Model& m) {
  const double DELTA = 0.5;
  const int LIMIT = 255;
  if (m.count < LIMIT) ++m.count;
  m.prediction += (bit - m.prediction) / (m.count + DELTA);
}
```

The compress() function takes a byte c and compresses it one bit at a time starting with the most significant bit. At each of the 8 steps, the previously coded bits are packed into a number in the range (1..255) as a binary number 1 followed by up to 7 earlier bits. For example, if c = 00011100, then bit_context takes the 8 successive values 1, 10, 100, 1000, 10001, 100011, 1000111, 10001110. In decompress(), c plays the same role. After 8 decoding operations it has the value 100011100 and the leading 1 is stripped off before being returned.

As before, the context may also be a hash.

The update function computes the prediction error (bit - m.prediction) and adjusts the prediction in inverse proportion to the count. The count is incremented up to a maximum value. At this point, the model switches from stationary to adaptive.

DELTA and LIMIT are tunable parameters. The best values depend on the data. A large LIMIT works best for stationary data. A smaller LIMIT works better for mixed data types. On stationary sources, the compressed size is typically larger by 1/LIMIT. The choice of DELTA is less critical because it only has a large effect when the data size is small (relative to the model size). With DELTA = 1, a series of zero bits would result in the prediction sequence 1/2, 1/4, 1/6, 1/8, 1/10. With DELTA = 0.5, the sequence would be 1/2, 1/6, 1/10, 1/14, 1/18. Cleary and Teahan (1995) measured the actual probabilities in English text and found a sequence near 1/2, 1/30, 1/60, 1/90... for zeros and 1/2, 19/20, 39/40, 59/60... for consecutive ones. This would fit DELTA around 0.07 to 0.1.

A real implementation would use integer arithmetic to represent fixed point numbers, and use a lookup table to compute 1/(m.count + DELTA) in update() to avoid a slow division operation. ZPAQ packs a 22 bit prediction and 10 bit count into a 32 bit model element. As a further optimization, the model is stored as a one dimensional array aligned on a 64 byte cache line boundary. The bytewise context is updated once per byte as usual, but the extra bits are expanded in groups of 4 in a way that causes only two cache misses per byte. The leading bits are expanded to 9 bits as shown below, then exclusive-ORed with the bytewise context address.

```
0 0000 0001
0 0000 001x
0 0000 01xx
0 0000 1xxx
1 xxxx 0001
```

```
1 xxxx 001x
1 xxxx 01xx
1 xxxx 1xxx
```

ZPAQ fixes DELTA at 1/2 but LIMIT is configurable to 4, 8, 12,..., 1020. The following table shows the effect of varying LIMIT for an order 0 model on 10^6 digits of π (stationary) and orders 0 through 2 on the 14 file Calgary corpus concatenated into a single data stream (nonstationary). Using a higher order model can improve compression at the cost of memory. However, direct lookup tables are not practical for orders higher than about 2. The order 2 model in ZPAQ uses 134 MB memory. The higher orders have no effect on π because the digits are independent (short of actually computing π).

	pi	Calgary corpus		
LIMIT	order-0	order-0	order-1	order-2
4	455,976	1,859,853	1,408,402	1,153,855
8	435,664	1,756,081	1,334,979	1,105,621
16	425,490	1,704,809	1,306,838	1,089,660
32	420,425	1,683,890	1,304,204	1,091,029
64	417,882	1,680,784	1,315,988	1,101,612
128	416,619	1,686,478	1,335,080	1,115,717
256	415,990	1,696,658	1,357,396	1,129,790
512	415,693	1,710,035	1,379,823	1,141,800
1020	415,566	1,726,280	1,399,988	1,150,737

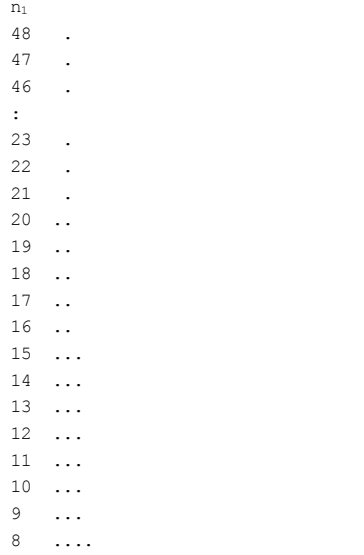
4.1.3. Indirect Models

An indirect context model answers the question of how to map a sequence of bits to a prediction for the next bit. Suppose you are given a sequence like 0000000001 and asked to predict what bit is next. If we assume that the source is stationary, then the answer is 0.1 because 1 out of 10 bits is a 1. If we assume a nonstationary source then the answer is higher because we give preference to newer history. How do we decide?

An indirect model learns the answer by observing what happened after similar sequences appeared. The model uses two tables. The first table maps a context to a bit history, a state representing a past sequence of bits. The second table maps the history to a prediction, just like a direct context model.

Indirect models were introduced in paq6 in 2004 A bit history may be written in the form (n_0, n_1, LB) which means that there have been n_0 zeros, n_1 ones, and that the last bit was LB (0 or 1). For example, the sequence 00101 would result in the state (3, 2, 1). The initial state is (0, 0, -) meaning there is no last bit.

In paq6 and its derivatives (including ZPAQ), a bit history is stored as 1 byte, which limits the number of states to 256. The state diagram below shows the allowed states in ZPAQ with n_0 on the horizontal axis and n_1 on the vertical axis. Two dots (:) represents two states for LB=0 and LB=1. A single dot represents a single state where LB can take only one value because the state is reachable with either a 0 or 1 but not both. (LB is the larger of the two counts). In general, an update with a 0 moves to the right and an update with a 1 moves up. The initial state is marked with a 0 in the lower left corner. The diagram is symmetric about the diagonal. There are a total of 219 states.



There are some exceptions to the update rule. Since it is not possible to go off the end of the diagram, the general rule is to move back to the nearest allowed state in the direction of the lower left corner (preserving the ratio of n_0 to n_1). There is another rule intended to make the model somewhat nonstationary, and that is when one of the counts is large and the other is incremented, then the larger count is reduced. The specific rule from the ZPAQ standard is that if the larger count is 6 or 7 it is decremented, and if it is larger than 7 then it is reduced to 7. This rule is applied first, prior to moving backward from an invalid state. For example, a sequence of 10 zeros, 43 ones and a zero results in:

Input	State	Rule
0000000000	(10,0,0)	Normal case, move right
1	(7,1,1)	Discount larger count
1	(6,2,1)	Discount
1	(5,3,1)	Discount
1	(5,4,1)	Normal case, move up
1	(5,5,1)	Normal case, move up
1	(4,5,1)	Move up off diagram, then back
1	(4,6,1)	Normal case, move up
1	(3,5,1)	Move up off diagram, then back
111	(3,8,1)	Normal, move up
1	(2,6,1)	Move off diagram and back
111111111	(2,15,1)	Normal
1	(1,8,1)	Move off diagram and back
111..(20x)	(1,48,1)	Normal, move up
1	(1,48,1)	Can't go any further
0	(2,7,0)	Discount

A bit history is mapped to a prediction like a direct context model, except that there is no count and the learning rate is fixed at 1/4096 of the error. The initial prediction for each bit history is $(n_1 + 0.5)/(n_0 + n_1 + 1)$.

The details of the design were determined experimentally to improve compression slightly over the PAQ8 series, which uses a similar design.

An indirect model is more memory efficient because it uses only one byte per context instead of four. In all of the PAQ series, it is implemented as a hash table. In the PAQ8 series, the hash table is designed to allow lookups with at most 3 cache misses per byte. In ZPAQ, there are 2 cache misses per byte, similar to the direct model. The ZPAQ hash table maps a context on a 4 bit boundary to an array of 15 bit histories and an 8-bit checksum. The histories represent all 15 possible contexts that occur after up to 3 more bits. The steps are as follows:

- Once per byte, a user specified context hash is computed.
- Once every 4 bits, the context hash is combined with the first 4 bits (if any) and a hash table lookup is done.
- A hash index h and an 8 bit checksum are extracted from the 32 bit hash.
- We look for a matching checksum at addresses h, h XOR 1 and h XOR 2 (to stay within the cache line) and return the first match found.
- If no match is found, then the array with the smallest $n_0 + n_1$ in the current context is replaced.

In a direct context model, we don't check for hash collisions because they have only a very small effect on compression. The effect is larger for indirect models so we use an 8 bit confirmation. There is still about a 1.2% chance that a collision won't be detected but the effect on compression is very small.

The following sizes were obtained for π and the Calgary corpus with order 0 through 5 models and a hash table size of 2^{28} (268 MB). For comparison, the best results for direct context models are shown. Direct models 3, 4, and 5 use context hashes with LIMIT set to 32 and the same memory usage.

Model	Direct	Indirect
pi order 0	415,566	426,343
Calgary order 0	1,680,784	1,716,974
Calgary order 1	1,304,204	1,289,769
Calgary order 2	1,089,660	1,048,050
Calgary order 3	1,017,354	964,942
Calgary order 4	1,069,981	1,010,329
Calgary order 5	1,232,997	1,148,677

There are many other possibilities. For example, M1, a context mixing compressor by Christopher Mattern, uses 2 dimensional context models taking 2 quantized bit histories as input.

4.2. Variable Order Models (DMC, PPM, CTW)

Fixed order models compress better using longer contexts up to a point (order 3 for the Calgary corpus). Beyond that, compression gets worse because many higher order contexts are being seen for the first time and no prediction can be made. One solution is to collect statistics for different orders at the same time and then use the longest matching context for which we know something. DMC does this for bit level predictions, and PPM for byte level predictions.

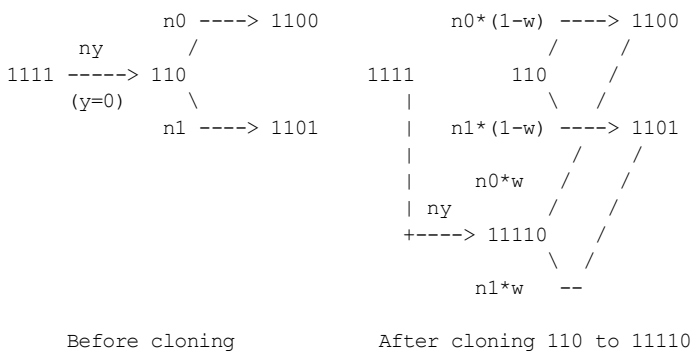
4.2.1. DMC

DMC (dynamic Markov coding) was described in a paper by Gordon Cormack and Nigel Horspool in 1987 and implemented in C The compressor hook by Nania Francesco Antonio was written in 2007 and is based on this implementation. DMC was implementd separately in ocamyd by Frank Schwellinger in 2006.

DMC uses a table of variable length bit level contexts that map to a pair of counts n_0 and n_1. It predicts the next bit with probability $n_1/(n_0+n_1)$ and updates by incrementing the corresponding count. This implements a stationary, direct context model. There are other possibilities, of course.

Contexts are not stored or looked up in the table. Rather, each table entry has a pair of pointers to the next entries corresponding to appending a 0 or 1 bit to the current context. The next context might also drop bits off the back. Normally it is arranged so that each context starts on a byte boundary. In DMC and HOOK, the initial table contains 64K entries consisting of all contexts of length 8 to 15 bits that start on a byte boundary, i.e. bytewise order 1.

In addition to updating counts, we may also add new table entries corresponding to the input just observed. We do this by "cloning" the next state with one that does not drop any bits off the back. Consider the case below where the context is 1111 and we update the model with a 0 bit. Without cloning, we would increment ny, transition to state 110, and the next prediction would be n1/(n0+n1).



The cloning procedure is to allocate a new state (labeled 11100) and copy its two output pointers from 110. The new state will also "inherit" the same prediction by copying the two counts. However we proportionally reduce the two original and two new counts in proportion to the contribution from the original input (1111) and from other states. The weight w = ny/(n0+n1) is the fraction of counts in state 110 that came from 1111 after a 0 bit. The newly cloned state gets that fraction of the counts and the original gets the rest. This scaling maintains the condition that the input and output counts are equal. The counts are implemened as fixed or floating point numbers to allow fractional values. The transition from 1111 to 110 is changed to point to the new state 11110.

Before cloning (which uses memory), there should be sufficient justification to do so. The two conditions are that the cloned state appears often enough (ny exceeds a threshold) and that there are sufficient other transitions to the original state that would remain after cloning (n0+n1-ny exceeds a threshold). In the original DMC, both thresholds are 2. Normally when the state table is full, the model is discarded and re-initialized. Setting higher thresholds can delay this from happening.

hook v1.4 also has an LZP preprocessor to encode long, repeated strings with the match length from the last matching context prior to DMC encoding. It compresses calgary.tar to 851,043 bytes in 1.9 seconds with 70 MB memory on a 2.0 GHz T3200. It compresses the files individually to a total of 840,970 bytes.

4.2.2 PPM

PPM (prediction by partial match) uses a byte-wise context model. Each context up to some maximum order is mapped to an array of counts for the next byte that occurred in this context. To predict the next byte, we find the longest context seen at least once before and allocate probabilities in proportion to those counts. The main complication is that some of the counts might be zero but we must never assign a zero probability to any byte value because if it did occur, it would have an infinite code length. This is handled by assigning an "escape" probability that the next byte will not be any of the ones that have nonzero counts, and divide those according to the frequency distribution of the next lower context. This is repeated all the way down to order 0. If any of the 256 byte values have still not been assigned a probability, then the remaining space is divided equally (an order -1 model).

Example: Suppose the input is BANANABOAT and we are at the point of coding the second B.

- The order 6 context BANANA has not been seen previously.
- The order 5 context ANANA has not been seen previously.
- The order 4 context NANA has not been seen previously.
- The order 3 context ANA has been seen once before, followed by N. The symbol we want to code is different so we code an "escape" symbol.
- The order 2 context NA was seen once, followed by N. Because N was already considered, we exclude it. Since there are no other predictions, there is nothing to code.
- The order 1 context is A. This was seen twice, in both cases followed by N which was excluded, so again there is nothing to code.
- The order 0 context was seen 6 times with values B (once), A (3 times) and N (twice). After excluding N we have 4 symbols with B having 1/4 of the probability and A having 3/4. We still need an escape probability P_{esc} that the next symbol will be other than B, A, or N. We then code B with probability $(1 - P_{esc})/4$. We would have coded A with probability $3(1 - P_{esc})/4$ and any other symbol with probability $p_{esc}/253$.

Estimating the escape probability can be complex. Suppose you draw 10 marbles from an urn. There are 8 blue, 1 green, and 1 white. What is the probability that the next marble will be a different color not seen before?

- By method "C" (Bell, Witten, and Cleary, 1989), 3 of the 10 marbles you drew had a novel color, so the probability would be 0.3.
- But novel colors would be expected to show up early. By method "X" (Witten and Bell, 1991), 2 of the colors appeared exactly once, so the probability would be 2/10 = 0.2.
- Method "X" can fail if no colors appeared exactly once (because the escape probability would be 0). Method "XC" is to use "X" when possible, falling back to "C" when needed. Method XC was shown experimentally to compress better than C.
- A secondary escape estimation (SEE) model would look at other cases with the same or similar distribution as {8, 1, 1}, and take the fraction of those where a novel color appeared next. This improves on XC.

Method X was shown to be optimal under certain assumptions, including that the source is stationary. Of course, that is not always the case. Suppose you receive the

sequence "BBBBBBBBWG" and are asked to predict whether the next character will be novel. The answer might be different for the sequence "WGBBBBBBBB".

Method "C" was implemented in ha, an order 5 PPMC archiver by Harry Hirvola in 1993. Later, Charles Bloom (1998) used SEE in PPMZ. Dmitry Shkarin (2002) refined this method in ppmd. ppmd variant I, released as source code in 2002, is used in several archivers such as WinZIP, freearc, 7zip, and WinRAR. (WinRAR uses an older variant H). A newer variant J in 2006 gets slightly better compression than I. The code is the basis of slower but better compressing programs such as ppmonstr and durilca. A variant of durilca using 13 GB memory is top ranked on the large text benchmark.

ppmd uses a complex SEE model. It considers 3 cases:

1. binary context - in the highest order context only one byte value has appeared (one or more times).
2. nm-context - two or more values have appeared in the highest order context.
3. m-context - two or more values have appeared, and one or more have appeared (are masked) in a higher order context.

In the binary context, a 13 bit context to a direct context model is constructed:

- 7 bits for the quantized count of the one symbol that appeared.
- 2 bits for the quantized alphabet size of the next lower order context.
- 1 bit for the quantized probability of the previously coded byte.
- 1 bit to indicate whether the two high order bits of the previous byte are 00 (to distinguish letters from other characters in text).
- 1 bit to indicate whether the two high order bits of the predicted byte are 00.
- 1 bit for the quantized number of successive bytes that were not escaped.

In the nm-context, the program fits the frequency distribution to a geometric approximation such that the n'th most frequent value is proportional to r^n. Then r is the context.

In the m-context, the SEE context is constructed from:

- The number of unmasked values with counts > 0, quantized to 25 levels.
- 2 bits based on comparison of the alphabet sizes of the current and next higher and lower order contexts.
- 1 bit to indicate the two high bits of the previous byte are 00.
- The average frequency per value, quantized to 4 levels.

ppmonstr uses an even more complex SEE context, and additionally uses interpolation to smooth some of the quantized contexts. It also adjusts the prediction for the most probable byte using secondary symbol estimation (SSE). This is a direct context model taking as input the quantized prediction and a (very complex) context and outputting a new prediction.

Both programs use other techniques to improve compression. They use partial update exclusion. When a character is counted in some context, it is counted with a weight of 1/2 in the next lower order context. Also, when computing symbol probabilities, it performs a weighted averaging with the predictions of the lower order context, with the weight of the lower order context inversely proportional to the number of different higher order contexts of which it is a suffix.

Statistics are stored in a tree which grows during modeling. When the memory limit is reached, the tree is discarded and rebuilt from scratch. Optionally, the the statistics associated with each context are scaled down and those with zero counts are pruned until the tree size becomes smaller than some threshold (25%). This improves compression but takes longer.

Although bytewise arithmetic encoding can be inefficient, ppmd is in practice faster than many equivalent bitwise context mixing models. First, a byte is encoded as a sequence of escape symbols followed by a non-escape. Each of these encodings is from a smaller alphabet. Second, the alphabet within each context can be ordered so that the most likely symbols are first. This reduces the number of operations in the majority of cases.

Shown below are compressed sizes of the Calgary corpus as a tar file and separate files. Compression and decompression times are the same. Option -o16 means use maximum order 16. -m256 says use 256 MB memory. -r1 says to prune the context tree rather than discard it.

Compressor	Options	calgary.tar	14 files	Time
ppmd J	-o16 -m256 -r1	754,243	740,737	2 sec
ppmonstr J	-o16 -m256 -r1	674,704	668,459	8 sec
durilca 0.5	-o128 -m256	672,752	666,216	10 sec

4.2.3. CTW

CTW (Context Tree Weighting) is an algorithm developed by Frans Willems, Yuri Shtarkov and Tjalling Tjalkens in 1995 and implemented by Erik Franken and Marcel Peeters in 2002. In the U.S. and Europe it is protected by patents EP0913035B1, EP0855803, US6150966, US5986591, US5864308 owned by Koninklijke KPN NV, Netherlands.

CTW, like DMC, predicts one bit at a time using bit count statistics for contexts that begin on byte boundaries. Like PPM and unlike DMC, it combines the predictions of different order contexts from order 0 up to some maximum order D. Each context is associated with two bit counters, n_0 and n_1, and a mixing weight, w. Each context order i from 0 through D computes an estimator e_i that the next bit will be a 1 depending on the two counts. The highest order context predicts $p_D = e_D$. All of the lower order contexts predict p_i which is a weighted average of e_i and the next higher order p_{i+1}. The final prediction is either p_0 or p_1.

On update with bit y (0 or 1), the count n_y is incremented and w is updated to favor the more accurate prediction in each context, either toward e_i or p_{i+1}.

Specifically, CTW uses the Krichevski-Trofimov (KT) estimator:

$$e_i(0) = (n_0 + 1/2)/(n_0 + n_1 + 1)$$
$$e_i(1) = (n_1 + 1/2)/(n_0 + n_1 + 1) = 1 - e_i(0).$$

except for the special case where one of the counts is 0, which we consider later. Then the prediction, expressed as a ratio, is:

$$p_D(0) = e_D(0)$$
$$p_D(1) = e_D(1)$$
$$p_i(1)/p_i(0) = (w_i e_i(1) + p_{i+1}(1))/(w_i e_i(0) + p_{i+1}(0)),\ i < D.$$

Given bit y, the weight and one count for each context is updated:

$$w_i := w_i e_i(y)/p_{i+1}(y),\ i < D$$
$$n_y := n_y + 1.$$

CTW uses a zero redundancy estimator for the special case where one of the two counts is zero. The idea is to predict more strongly the other bit. Consider the case of $n_0 = 0$, i.e. only ones have been observed. Then define k(n) as the cumulative probability of n consecutive bits by the KT estimator, and define the ZR estimator e() as:

$$k(0) = 1/2$$
$$k(i) = k(i-1)(i+1/2)/(i+1),\ i > 0$$
$$e(1) = (n_1 + 1/2)/(n_0 + n_1 + 1),\ n_1 > 0,\ n_0 > 0 \text{ (KT estimator)}$$
$$e(1) = k(n_1)/(2(n_1 + 1)k(n_1) + n_1 + 1),\ n_1 > 0,\ n_0 = 0 \text{ (ZR estimator)}$$

All estimators are symmetric with respect to 0 and 1. The probability of a 1 given the sequence 00000 is the same as the probability of 0 given 11111. The ZR estimator gives a smaller probability for these cases than the KT estimator. The following table gives the probability that a 1 will follow n_0 zeros and no ones with each estimator.

n_0	KT	ZR
0	0.500000	0.500000
1	0.250000	0.107143
2	0.166667	0.064103
3	0.125000	0.044192
4	0.100000	0.032984
5	0.083333	0.025908
6	0.071429	0.021089
7	0.062500	0.017625
8	0.055556	0.015032
9	0.050000	0.013029
10	0.045455	0.011441
100	0.004950	0.000499
1000	0.000500	0.000017

For large runs of identical bits, the total code length for KT grows logarithmically but the ZR estimator is bounded to 2 bits. However, ZR adds 1 bit of redundancy when the other bit value eventually occurs. For large n_0, the ZR estimated probability of a 1 approaches 0 at the rate $n_0^{-3/2}$. This is faster than n_0^{-1} for the KT estimator.

The CTW implementation uses a context tree to store the two counts and w. Each count is 8 bits. If one count exceeds 255, then both counts are divided by 2. It stores log(w) rather than w and does all arithmetic in the log domain so that multiplication and division are implemented using addition and subtraction. The KT and ZR estimators are looked up in a table indexed by the two counts.

A context tree is a 256-ary tree where each node at depth N+1 from the root represents an order-N context. The parent of each node is the suffix obtained by removing one byte. The children of the root node represent the 255 possible partial byte order 0 contexts of length 0 to 7 bits. To save space, children of contexts that have only occurred once are not stored. Each tree node uses 8 bytes, including a 24 bit pointer to the context in a separate file buffer.

CTW is best suited for stationary sources. The published CTW implementation compresses the Calgary corpus to 767,594 bytes as 14 separate files in 62 seconds with options -n16M -b16M -d12 set for maximum compression. With the same options, it compresses calgary.tar to 965,855 bytes. -d12 sets the maximum context order to 12. -b16M sets the file buffer to 16 MB. -n16M limits the tree to 16 million nodes. When the tree is full, no new contexts are added but the counts and weights continue to be updated. These settings require 144 MB memory, the maximum that the published implemention can use.

4.3. Context Mixing

Context mixing algorithms based on the PAQ series are top ranked on many benchmarks by size, but are very slow. These algorithms predict one bit at a time (like CTW) except that weights are associated with models rather than contexts, and the contexts need not be mixed from longest to shortest context order. Contexts can be arbitrary functions of the history, not just suffixes of different lengths. Often the result is that the combined prediction of independent models compresses better than any of the individuals that contributed to it.

DMC, PPM, and CTW are based on the premise that the longest context for which statistics is available is the best predictor. This is usually true for text but not always the case. For example, in an audio file, a predictor would be better off ignoring the low order bits of the the samples in its context because they are mostly noise. For image compression, the best predictors are the neighboring pixels in two dimensions, which do not form a contiguous context. For text, we can improve compression using some contexts that begin on word boundaries and merge upper and lower case letters. In data with fixed length records such as spreadsheets, databases or tables, the column number is a useful context, sometimes in combination with adjacent data in two dimensions. PAQ based compressors may have tens or hundreds of these different models to predict the next input bit.

A fundamental question is how do we combine predictions? Suppose you are given two predictions pa = P(y=1|A) and pb = P(y=1|B), probabilities that the next bit y will be a 1 given contexts A and B. Assume that A and B have occurred often enough for the two models to make reliable guesses, but that both contexts have never occurred together before. What is p = P(Y=1|A,B)?

Probability theory does not answer the question. It is possible to create sequences where p can be anything at all for any pa and pb. For example, we could have pa=1, pb=1, p=0. But intuitively, we should do some kind of averaging or weighted averaging. For example, if we wish to estimate P(car accident | dark and raining) given P(car accident | dark) and P(car accident | raining), we would expect the effects to be additive.

In most PAQ based algorithms, there is a procedure for evaluating the accuracy of models and adjusting the weights to favor the best ones. Early versions used fixed weights.

4.3.1. Linear Mixing

In PAQ6 (Mahoney, 2005a), a probability is expressed as a count of zeros and ones. Probabilities are combined by weighted addition of the counts. Weights are adjusted in the direction that minimizes coding cost in weight space. Let n_{0i} and n_{1i} be the counts of 0 and 1 bits for the i'th model. The combined probabilities p_0 and p_1 that the next bit will be a 0 or 1 respectively, are computed as follows:

$$S0 = \varepsilon + \sum_i w_i n_{0i} = \text{evidence for 0}$$
$$S1 = \varepsilon + \sum_i w_i n_{1i} = \text{evidence for 1}$$
$$S = S0 + S1 = \text{total evidence}$$
$$p_0 = S0/S = \text{probability that next bit is 0}$$
$$p_1 = S1/S = \text{probability that next bit is 1}$$

where w_i is the non-negative weight of the i'th model and ε is a small positive constant needed to prevent degenerate behavior when S is near 0.

The optimal weight update can be found by taking the partial derivative of the coding cost with respect to w_i. The coding cost of a 0 is -log p_1. The coding cost of a 1 is -log p_0. The result is that after coding bit y (0 or 1), the weights are updated by moving along the cost gradient in weight space:

$$w_i := \max[0, w_i + (y - p_i)(S\, n_{1i} - S1\, n_i) / S0\, S1]$$

Counts are discounted to favor newer data over older. A pair of counts is represented as a bit history similar to the one described in section 4.1.3, but with more aggressive discounting. When a bit is observed and the count for the opposite bit is more than 2, the excess is halved. For example if the state is $(n_0, n_1) = (0, 10)$, then successive zero bits will result in the states (1, 6), (2, 4), (3, 3), (4, 2), (5, 2), (6, 2).

4.3.2. Logistic Mixing

PAQ7 introduced logistic mixing, which is now favored because it gives better compression. It is more general, since only a probability is needed as input. This allows the use of direct context models and a more flexible arrangement of different model types. It is used in the PAQ8, LPAQ, PAQ8HP series and in ZPAQ.

Given a set of predictions p_i that the next bit will be a 1, and a set of weights w_i, the combined prediction is:

$$p = \text{squash}(\sum_i w_i\, \text{stretch}(p_i))$$

where

$$\text{stretch}(p) = \ln(p) / \ln(1-p);\ \text{squash}(x) = \text{stretch}^{-1}(x) = 1/(1 + e^{-x})$$

The probability computation is essentially a neural network evaluation taking stretched probabilities as input. Again we find the optimal weight update by taking the partial derivative of the coding cost with respect to the weights. The result is that the update for bit y (0 or 1) is simpler than back propagation (which would minimizes RMS error instead).

$$w_i := w_i + \lambda\, (y - p)\, \text{stretch}(p_i)$$

where λ is the learning rate, typically around 0.01, and (y - p) is the prediction error. Unlike linear mixing, weights can be negative.

Compression can often be improved by using a set of weights selected by a small context, such as a bytewise order 0 context.

In PAQ and ZPAQ, squash() and stretch() are implemented using lookup tables. In PAQ, both output 12 bit fixed point numbers. A stretched probability has a resolution of 2^{-8} and range of -8 to 8. Squashed probabilities are multiples of 2^{-12}. ZPAQ represents stretched probabilities as 12 bits with a resolution of 2^{-6} and range -32 to 32. Squashed probabilities are 15 bits as an odd multiple of 2^{-16}. This representation was found to give slightly better compression than PAQ.

ZPAQ allows different components (models and mixers) to be connected in arbitrary ways. All components output a stretched probability, which simplifies the mixer implementation. ZPAQ has 3 types of mixers:

- AVG performs weighted averaging of two (stretched) predictions with fixed, user specified weights that add to 1.
- MIX2 is like AVG except that weights are updated with the constraint that they add to 1. The user specifies λ and a context to select a pair of weights.
- MIX is like MIX2 except that it takes any number of inputs and does not constrain the weights to add to 1. A 2 input MIX often gives better compression than a MIX2.

Mixer weights in PAQ are 16 bit signed values to facilitate vectorized implementation using MMX/SSE2 parallel instructions. In ZPAQ, 16 bits was found to be inadequate for best compression. Weights were expanded to 20 bit signed values with range -8 to 8 and precision 2^{-16}.

4.3.3. Secondary Symbol Estimation (SSE)

SSE (secondary symbol estimation) is implemented in all PAQ versions beginning with PAQ2. Like in ppmonstr, it inputs a prediction and a context and outputs a refined prediction. The prediction is quantized typically to 32 or 64 values on a nonlinear scale with finer resolution near 0 and 1 and sometimes interpolated between the two closest values. On update, one or both values are adjusted to reduce the prediction error, typically by about 1%. A typical place for SSE is to adjust the output of a mixer using a low order (0 or 1) context. SSE components may be chained in series with contexts typically in increasing order. Or they may be in parallel with independent contexts, and the results mixed or averaged together. The table is initialized so that the output prediction is equal to the input prediction for all contexts.

SSE was introduced to PAQ in PAQ2 in 2003 with 64 quantization levels and no interpolation. Later versions used 32 levels and interpolation with updates to the two nearest values above and below. In some versions of PAQ, SSE is also known as an APM (adaptive probability map).

ZPAQ allows a SSE to be placed anywhere in the prediction sequence with any context. Recall that ZPAQ probabilities are stretched by mapping to ln(p/(1-p)) as a 12 bit fixed point number in the range -32 to +32 with resolution 1/64. The SSE input prediction is clamped and quantized to an odd multiple of 1/2 between -15.5 and 15.5. The low 6 bits serve as an interpolation weight. For example, if stretch(p) = 2.7, then the two table entries are selected by below=2.5 and above=3.5, and the interpolation weight is 0.2. Then the output prediction is SSE[context][below]*(1-w) + SSE[context][above]*w. Upon update with bit y, the table entry nearest the input prediction (SSE[context][below] in this example) is updated by reducing the prediction error (y - output) by a user specified fraction.

There are other possibilities. CCM, a context mixing compressor by Christian Martelock, uses a 2 dimensional SSE taking 2 quantized predictions as input.

4.3.4 Indirect SSE (ISSE)

ISSE (indirect secondary symbol estimation) is a technique introduced in paq9a in Dec. 2007 and is a component in ZPAQ. The idea is to use SSE as a direct prediction method rather than to refine an existing prediction. However, SSE does not work well with high order contexts because the large table size uses too much memory. More generally, a large model with lots of free parameters (each table entry is a free parameter) will overfit the training data and have no predictive power for future input. As a general rule, a model should not be larger than the input it is trained on.

ISSE does not use a 2-D table. Instead it first maps a context to a bit history as with an indirect context map. Then the 8-bit bit history is used as a context to select the pair of weights for a 2 input mixer taking the input prediction and a fixed constant as its two inputs. The weights are initialized to (1.0, 0.0) meaning that the initial output prediction is equal to the input.

PAQ9A and the default compression mode of ZPAQ both start with an order 0 model prediction and refine it using a chain of ISSE components in increasing order.

In ZPAQ, the weights are 20 bit signed, fixed point numbers with range -8 to 8 and precision 2^{-16} like in a MIX. The fixed input is 4.0 and the learning rate is fixed at $\lambda = 2^{-8}$.

4.3.5. Match Model

A match model finds the last occurrence of a high order context and predicts whatever symbol came next. The accuracy of the prediction depends on the length of the context match. Longer matches generally give more confidence to the prediction. Typically a match model of order 6-8 is mixed with lower order context models. A match model is faster and uses less memory than a corresponding context model but does not model well for low orders.

Match models are used in PAQ and ZPAQ. They consist of a rotating history buffer and a hash table mapping contexts to pointers into the buffer. In ZPAQ, a pointer to the match is maintained until a mismatching bit is found. The model will then look for a new match at the start of the next byte. On each byte boundary, the buffer is updated with the modeled byte and the hash table is updated so that the current context hash points to the end of the buffer. ZPAQ allows both the hash table size and buffer size to be user specified (as powers of 2). For best compression, the history buffer should be as large as the input (if this is practical) and the hash table size is typically 1/4 of this. Because each pointer is 4 bytes, both data structures use the same amount of memory.

Match models in PAQ maintain multiple context hashes of different orders and multiple pointers into the buffer. The prediction is indirect by mapping the match length to a prediction through a direct context model. ZPAQ uses a simpler match model with just one pointer and one hash, although it is possible to have multiple, independent match models. The prediction for a match of L bytes (or 8L bits) is that the next bit will be the same with probability 1 - 1/8L.

The user may specify the context length by using a rolling hash that depends on the desired number of characters. If h is the context hash, c is the input byte, then the update:

```
h = h*((2*k+1) << m) + c;
```

specifies an order ceil(n/m) context hash for a hash table size of 2^n and any k. For example, the hash update "h=h*40+c;" (m = 3) is an order 6 context hash for a table size of 2^{18}. Only the low 18 bits of h would be used to index the hash table of this size, and these bits depend only on the last 6 values of c.

4.3.6 PAQ Models

The high compression ratio (and slow speed) in PAQ comes from using many different context models for different types of data. These are described in historical order.

Schmidhuber and Heil (1996) developed an experimental neural network data compressor. It used a 3 layer network trained by back propagation to predict characters from an 80 character alphabet in text. It used separate training and prediction phases. Compressing 10 KB of text required several days of computation on an HP 700 workstation.

Mahoney (2000) made several improvements that made neural network compression practical.

- The neural network predictes one bit at a time instead of one character.
- The training is online, eliminating multiple passes.
- The first layer of the neural network is replaced by a hash function that selects one neuron per context for each of the orders 1 through 5.

These changes make the algorithm about 10^5 times faster, mainly because only a few neurons (out of millions) are active at any one time. To make the training online, it is necessary to add a pair of counters to each weight (to count 0 and 1 bits) so that the learning rate is initially large. The rate decreases in inverse proportion to the smaller of the two counts.

There are 3 versions: P5, P6, and P12. P5 uses 256 KB memory to represent 5 orders using 2^{16} input neurons (each representing a context hash) and one output (the bit prediction). P6 uses 2^{20} input neurons. P12 is the same size as P6 but adds a whole word model. This context hash depends only on alphabetic characters mapped to lower case, and is reset after a nonalphabetic character. It improves compression of text files.

In PAQ1 (Mahoney, 2002), it was realized that the counts alone could be used to make predictions, so the weights were eliminated. Predictions are combined by adding the 0 and 1 counts associated with each context. Each counter is 1 byte.

PAQ2 added SSE by Serge Osnach in May 2003. It uses 64 quantization levels without interpolation.

PAQ3 modified SSE to use 32 levels with interpolation in Sept. 2003.

PAQ3N by Serge Osnach in Oct. 2003 added sparse models: three order-2 models that skipped 1, 2, or 3 bytes of context between the two context bytes. This improves compression of some binary files.

PAQ4 (Nov. 2003) uses adaptive linear weighting of models as described in section 4.3.1. It also introduced a record model. It identifies structures that repeat at regular intervals, as found in spreadsheets, tables, databases, and

images, and adds contexts of adjacent bytes in two dimensions.

PAQ5 (Dec. 2003) has some minor improvements over PAQ4, including word models for text, models for audio and images, an improved hash table, and modeling of run lengths within contexts. It uses two mixers with different contexts to select their weights. The two mixer outputs are averaged together. It uses about 186 MB of memory.

PAQ6 (Jan. 2004) adds models for x86 code (modeling jump/call addresses) and CCITT images and more aggressive discounting of opposing bit counts. It takes options allowing up to 1616 MB memory. It is the basis of a number of forks and dozens of versions. An early version won the Calgary challenge. Many other models and optimizations were added by Berto Destasio, Johan de Bock, David A. Scott, Fabio Buffoni, Jason Schmidt, Alexandar Ratushnyak (PAQAR), Przemyslaw Skibinski (PASQDA, text preprocessing), Rudi Cilibrasi, Pavel Holoborodko, Bill Pettis, Serge Osnach, and Jan Ondrus.

PAQAR (v1.0 to 4.0, May-July 2004) by Alexander Ratushnyak is a PAQ6 fork which is the basis of several winning submissions to the Calgary Challenge. The primary difference is a greatly increased number of mixers and SSE chains.

PAQ7 (Dec. 2005) was a complete rewrite. It uses logistic mixing rather than linear mixing, as described in section 4.3.2. It has models for color BMP, TIFF, and JPEG images. The BMP and TIFF models use adjacent pixels as context. JPEG is already compressed. The model partially undoes the compression back to the DCT (discrete cosine transform) coefficients and uses these as context to predict the Huffman codes.

PAQ8A (Jan. 2006) adds a E8E9 preprocessor to improve compression of x86 (EXE and DLL) files. The preprocessor replaces relative CALL and JMP addresses with absolute addresses, which improves compression because an address may appear multiple times. Many other compressors use this technique.

PASQDA (v1.0-v4.4, Jan. 2005 to Jan. 2006) is a fork by Przemyslaw Skibinski. It adds an external dictionary to replace words in text files with 1 to 3 byte symbols. This technique was used successfully in the Hutter Prize and in the top ranked programs in the large text benchmark. PAQ8A2, PAQ8B, PAQ8C, PAQ8D, PAQ8E, and PAQ8G (Feb. to Mar. 2006) also use this technique, as does PAQAR 4.5 by Alexander Ratushnyak.

PAQ8F (Feb. 2006) adds a more memory efficient context model and a new indirect model: The byte history within a low order context is modeled by another low order context.

PAQ8L (Mar. 2006) adds a DMC model. Its predictions are mixed with those of other models.

As of Feb. 2010, development remains active on the PAQ8 series. There have been hundreds of versions with improvements and additional models. The latest is PAQ8PX_V67. Most of the improvements have been for file types not included in the Calgary corpus such as x86, JPEG, BMP, TIFF, and WAV.

A benchmark for the Calgary corpus is given below for versions of PAQ from 2000 to Jan. 2010 showing major code changes. About 150 intermediate versions with minor improvements have been omitted. Older programs marked with * were benchmarked on slower machines such as a 750 MHz Duron and have been adjusted to show projected times on a 2.0 GHz T3200, assumed to be 5.21 times faster. Sizes marked with a D use an external English dictionary that must be present during decompression. The size shown does not include the dictionary, so it is artificially low. However, including it would give a size artificially high because the dictionary is not extracted from the test data. All versions of PAQ are archivers that compress in solid mode, exploiting similarities between files. Decompression time is about the same as compression time.

Compressor	Calgary	Seconds	Memory	Date	Author	Major changes
P5	992,902	6.1*	256 KB	2000	Mahoney	64K x 1 neural network
P6	841,717	7.4*	16 MB	2000	Mahoney	1M neurons
P12	831,341	7.5*	16 MB	2000	Mahoney	Word context model
PAQ1	716,704	13*	48 MB	2002	Mahoney	Linear mixing with fixed weights
PAQ2	702,382	18*	48 MB	May 2003	Osnach	SSE
PAQ3	696,616	15*	48 MB	Sep 2003	Mahoney	Interpolated SSE
PAQ3N	684,580	30*	80 MB	Oct 2003	Osnach	Sparse models
PAQ4	672,134	43*	84 MB	Nov 2003	Mahoney	Adaptive mixer weights, record models
PAQ5	661,811	70*	186 MB	Dec 2003	Mahoney	Models for text, audio, images, runs, 2 mixers
PAQ6 -6	648,892	99*	202 MB	Jan 2004	Mahoney	Models for PIC, x86
PAQAR 4.0 -6	604,254	408*	230 MB	Jul 2004	Ratushnyak	Many mixers and SSE chains
PAQ7 -5	611,684	142*	525 MB	Dec 2005	Mahoney	Logistic mixing, image models
PAQ8A -4	610,624	152*	115 MB	Jan 2006	Mahoney	E8E9 preprocessor
PASQDA 4.4 -7	D 571,011	283*	470 MB	Jan 2006	Skibinski	PAQ7 + external dictionary
PAQAR 4.5 -5	D 570,374	299*	191 MB	Feb 2006	Ratushnyak	PAQAR + external dictionary
PAQ8F -6	605,650	161*	435 MB	Feb 2006	Mahoney	Bytewise indirect model, memory tuning
PAQ8L -6	595,586	368	435 MB	Mar 2007	Mahoney	DMC model
PAQ8PX_V67 -6	598,969	469	421 MB	Jan 2010	Ondrus	Improved JPEG, TIFF, BMP, WAV models

Since 2007, development has contined on PAQ. In addition to PAQ8PX, there are 3 additional forks, no longer under active development.

- PAQ8HP by Alexander Rathusnyak, a basis for the Hutter Prize. The series was optimized for this data and the large text benchmark. It uses a dictionary transform based on XWRT which replaces words from a dictionary with 1 to 3 byte codes. The dictionary for the paq8hp series is optimized for these benchmarks, which is allowed under the rules. There were 12 versions from Aug. 2006 through May 2007.
- LPAQ by Matt Mahoney with later versions by Alexander Ratushnyak. This was a "lite" PAQ, faster but with less compression. Later versions were tuned for text. It includes a mix of contexts of different orders and a match model. There were 24 version from July 2007 through Feb. 2009.
- PAQ9A (Dec. 2007) was an experiment in LZP preprocessing followed by chained ISSE modeling. Compression is similar to early versions of LPAQ. The LZP preprocessor removes long redundant strings. It includes sparse and text models but no match model because high order contexts were removed.

4.3.7. ZPAQ

Of the hundreds of PAQ versions, no program can decompress files compressed by any other version. The goal of the proposed ZPAQ standard is to solve this problem. It specifies a format in which a description of the compression algorithm is stored in the compressed archive. The specification includes a reference decoder.

The specification does not describe the compression algorithm. However, several compression programs and models are available on the ZPAQ page. There is a ZPAQ program that takes a configuration file to describe the compression algorithm, as well as other programs like ZPIPE that use a fixed compression algorithm. All of these produce files that can be decompressed with the reference decoder or by any of these programs. The standard was published in Mar. 2009 by Matt Mahoney.

ZPAQ describes an archive format, although it may be used for single file compression or memory to memory compression. A compressed stream consists of a sequence of blocks that are independent and can be reordered. Each block starts with a header that describes the decompression algorithm. A block consists of a sequence of compressed segments that must be decompressed in sequence from the start of the block. A segment may be a file or a part of a file. Each segment has an optional file name, an optional comment (file size, timestamp, etc.), and ends with an optional 20 byte SHA-1 checksum. If the file name is omitted, then the decompresser must supply it.

An algorithm description consists of a network of components (each making a prediction), a program that computes the bytewise contexts for each component, and an optional program that post-processes the output. Both programs are written in a language called ZPAQL which is compiled or interpreted by the decompresser. If the post-processing program is present, then it is appended to the front of the first uncompressed segment and compressed along with its input data. If not, then the data is compressed directly with a one byte header to indicate its absence.

Up to 255 components are placed in an array. Each component in the model inputs a context hash and possibly the predictions of earlier components, and outputs a stretched prediction. The output of the last component is squashed and used to arithmetic code the data one bit at a time. The components are as follows:

- CONST - outputs a fixed prediction in the stretched range -16 to 16 in increments of 1/16.
- CM - a direct context model (section 4.1.2). Inputs a context and outputs a prediction, which is then updated to reduce the prediction error. The table size (and thus the context size) may be any power of 2. The count limit, which controls the learning rate, ranges from 4 to 1020 in increments of 4.
- ICM - an indirect context model (section 4.1.3). Maps a context to a bit history, which is mapped to a prediction. The size may be any power of 2 at least 64 bytes.
- MATCH - (section 4.3.5). Predicts the next bit from the last matching context in the history buffer. The hash table size and buffer size may be any power of 2.
- AVG - weighted average of any 2 predictions. The weight ranges from 0 to 1 in increments of 1/256.
- MIX - (section 4.3.2). Adaptively averages predictions using weights selected by a context. Specifies a context size, a range of inputs, a learning rate in increments of 1/4096 up to 1/16, and an 8 bit mask to turn on or off bits of the current byte in the order 0 context.
- MIX2 - (section 4.3.2). Like a MIX but with any 2 inputs, and the weights constrained to add to 1.
- SSE - (section 4.3.3). Maps a context and a prediction to a new prediction using a 2-D table. Specifies a context size, input prediction, and initial and maximum counts for the context model (0..255 and 4..1020 by 4 respectively) and a bit mask for the order 0 context like in a MIX or MIX2.
- ISSE - (section 4.3.4). Maps a context and a prediction to a new prediction using a bit history to select the weights for a 2 input mixer with the other input constant. Specifies a context size and input prediction.

Contexts are computed by a ZPAQL program that is called once per modeled byte with that byte as input. The program places the context hashes in an array H of 32 bit unsigned values. Each element of H is the input for one component, beginning with H[0] for the first one. Only the low bits of the hash are used, depending on the table size of each component. Because each bit must be modeled, the context hashes are combined with the previous bits of the current byte. This is done by expanding the previous bits to a 9 bit value (ICM or ISSE) or 8 bits (all others) and adding it to the bytewise context.

ZPAQL is designed for fast execution rather than ease of programming. It resembles an assembly language instruction set. A ZPAQL program runs on a virtual machine with the following state, all initialized to 0 at the start of a block:

- A 16 bit program counter.
- 32 bit registers A, B, C, D, R0 through R255. The accumulator A is used for input and the results of most computations.
- A 1 bit flag F to hold the result of comparisons.
- An array H of 32 bit values, indexed by D. The first 256 elements of H hold computed contexts.
- An array M of 8 bit values, indexed by B or C.

The sizes of H and M are specified as powers of 2 in the block header. Most instructions are either 1 byte, or 2 bytes including a numeric operand. The instruction set is as follows:

- Assignment, for example A=B meaning copy B into A. The left and right operands may be A, B, C, D, *B, *C, or *D. *B and *C mean the element of M addressed by the low bits of B or C (depending on the size of M). *D means an element of H. The right operand may also be a constant from 0 through 255.
- Arithmetic, for example, A+=B meaning add B to A. Operands are as above except that the left operand is always A. Operations are += -= *= /= %= &= |= ^= <<= >>= &~ with their usual meanings in C/C++. Division or mod by 0 is 0. &~ means &=~
- Comparison < == > only. The left operand is A. The result goes in F.
- Unary operations ++ (increment) -- (decrement) ! (bit compliment) <>A (swap with A), =0 (clear). The operand may be A, B, C, D, *B, *C, or *D, always written first as in B! to compliment the bits of B (B = ~B or B= -B-1;). A<>A is not valid.
- HASH meaning A = (A + *B + 512) * 773, a convenient hashing function.
- HASHD meaning *D = (*D + A + 512) * 773.
- OUT meaning output A (used only in postprocessing).
- HALT meaning end execution.
- Conditional jumps such as JT -4 meaning jump back 4 bytes (from the next instruction) if F is true. JF means jump if false. JMP is unconditional. The operand is -128 to 127.
- Long jump LJ with a 2 byte operand 0 through 65535 from the start of the program.
- Access to R0 through R255 for example R=A 5 meaning R5=A. Operations are A=R, B=R, C=R, D=R, R=A.

The post-processor (called PCOMP), if it is present, is called once per decoded byte with that byte in the A register. At the end of each segment, it is called once more with -1 in A. The decompresser output is whatever is output by the OUT instruction.

The context model (called HCOMP) is always present. It is called once per decoded byte. It puts its result in H. OUT has no effect. HCOMP sees as input the PCOMP code (if present) followed by a contiguous stream of segments with no separator symbols.

The ZPAQ program is a development environment for writing and debugging models. It allows programs to be single stepped or run separate from compression. It accepts control statements IF/IFNOT-ELSE-ENDIF and DO-WHILE/UNTIL/FOREVER and converts them to conditional jumps. It allows passing of numeric arguments and comments in parenthesis. If a C++ compiler is present, then ZPAQL code is compiled by converting it to C++ and then running it. Otherwise the code is interpreted. Compiling makes compression and decompression 2 to 4 times faster.

The default configuration for both ZPAQ and ZPIPE is described by the file mid.cfg below.

```
(zpaq 1.07+ config file tuned for average compression.
Uses 111 * 2^$1 MB memory, where $1 is the first argument.)

comp 3 3 0 0 8 (hh hm ph pm n)
  0 icm 5        (order 0...5 chain)
  1 isse 13 0
  2 isse $1+17 1
  3 isse $1+18 2
  4 isse $1+18 3
  5 isse $1+19 4
  6 match $1+22 $1+24  (order 7)
  7 mix 16 0 7 24 255  (order 1)
hcomp
  c++ *c=a b=c a=0 (save in rotating buffer M)
  d= 1 hash *d=a   (orders 1...5 for isse)
  b-- d++ hash *d=a
  b-- d++ hash *d=a
  b-- d++ hash *d=a
  b-- d++ hash *d=a
  b-- d++ hash b-- hash *d=a (order 7 for match)
  d++ a=*c a<<= 8 *d=a       (order 1 for mix)
  halt
post
  0
end
```

The comment about $1 means that the model can be run with additional memory to improve compression. For example:

```
zpaq ocmid.cfg archive files...
```

will compress with 111 MB memory, and

```
zpaq ocmid.cfg,3 archive files...
```

will compress with 111 * 2^3 = 888 MB memory. Decompression requires the same amount. The effect of ",3" is to make substitutions like "$1+17" with 20 throughout the configuration file. Up to 9 parameters (to $9) are allowed.

The command "oc" means optimize (compile the ZPAQL into C++) and compress. If the "o" is dropped then no external C++ compiler is required, but compression and decompression takes twice as long.

The configuration file is divided into 3 sections. COMP describes the arrangement of components. HCOMP contains ZPAQL code that computes the contexts and puts them in H. POST 0 indicates that there is no post-processing.

COMP is followed by 5 arguments: hh, hm, ph, pm, n. hh and hm specify the sizes of H and M in HCOMP as powers of 2 (2^3 = 8 each). ph and pm are 0 because these arrays are not used. (Their size is actually 1). n = 8 is the number of components. They must be numbered 0 through n-1 in the COMP section. Except for the line numbers, each token compiles to one byte of ZPAQL. (Thus, ZPAQ requires "A= 10" be written exactly like this and not "A=10" or "A = 10" to indicate it is a 2 byte instruction).

The line

```
0 icm 5
```

describes an indirect context model with a table size of 64 * 2^5 bytes. It takes its context from the low 15 bits of H[0]. The low 7 bits index a table of 16 byte arrays, and the next 8 bits are the checksum to detect collisions. Since this is an order 0 context, H[0] is left at 0 and only the bitwise context (a 9 bit value) is used. The line

```
1 isse 13 0
```

describes an indirect SSE using 64 * 2^{13} bytes taking its input from component number 0 and context from H[1]. The line

```
2 isse $1+17 1
```

describes an indirect SSE using 64 * $2^{\$1+17}$ bytes, where $1 is the argument passed to mid.cfg. For example, if the argument is 3, it uses 64 * 2^{20} bytes. $1 defaults to 0. It gets its input prediction from component 1 and its context from H[2]. The line

```
6 match $1+22 $1+24
```

specifies a match model with a hash table size of 2^{22} and history buffer of size 2^{24} (taking 16 MB each if $1 is 0). Its context is H[6]. The line

```
7 mix 16 0 7 24 255
```

specifies a mixer with 2^{16} sets of weights selected by the low 16 bits of the context, taking as input predictions from the range of components 0 through 7-1, with a learning rate of 24/4096, and no masking (AND the bitwise context with 255). The context is H[7].

The HCOMP section computes the contexts and puts them in H. It puts order 0 through 5 context hashes in H[0] through H[5], an order 7 context in H[6] for the match model, and an unhashed order 1 context in bits 8..15 of H[7] for the mixer. It leaves bits 0..7 clear because the bitwise context will be added to this. This is not a concern for the other contexts because they are hashed.

HCOMP is called once after modeling each byte with the byte in the A register. All state information except A and PC (which is reset to the first instruction) is preserved between calls.

This program uses M as a rotating history buffer of 8 bytes (hm = 3) with the low 3 bits of C pointing to the last byte stored. It uses B as a working pointer to compute hashes and D as a pointer into H to store the result. The instructions

```
c++ *c=a b=c a=0
```

increment C, store the input byte in M[C], copy C to B, and clear A.

```
d= 1 hash *d=a
```

assigns 1 to D so that it points to H[1]. The hash instruction takes input from M[B] and combines it with A (0), so A now contains a hash of the last input byte. It is stored in H[D] = H[1] as an order 1 context for component 1. Subsequent instructions store order 2, 3, 4, 5, and 7 hashes in H[2] through H[6]. Note that the space in "d= 1" is required because it is a 2 byte instruction. "a=0" doesn't require this because there is a special 1 byte instruction for clearing a register.

```
d++ a=*c a<<= 8 *d=a
```

computes the mixer context by putting the input byte (saved in *C) into bits 8..15 of D[7] and leaving the other bits at 0. Execution ends at the halt instruction.

The following is the configuration max.cfg, which gets better compression but is slower.

```
(zpaq 1.07+ config file tuned for high compression (slow)
Uses 245 x 2^$1 MB memory, where $1 is the first argument.

comp 5 9 0 0 22 (hh hm ph pm n)
  0 const 160
  1 icm 5  (orders 0-6)
  2 isse 13 1 (sizebits j)
  3 isse $1+16 2
  4 isse $1+18 3
  5 isse $1+19 4
  6 isse $1+19 5
  7 isse $1+20 6
  8 match $1+22 $1+24
  9 icm $1+17 (order 0 word)
  10 isse $1+19 9 (order 1 word)
  11 icm 13 (sparse with gaps 1-3)
  12 icm 13
  13 icm 13
  14 icm 14 (pic)
  15 mix 16 0 15 24 255  (mix orders 1 and 0)
  16 mix 8 0 16 10 255   (including last mixer)
  17 mix2 0 15 16 24 0   (order -1)
  18 sse 8 17 32 255     (order 0)
  19 mix2 8 17 18 16 255 (order 0)
  20 sse 16 19 32 255    (order 1)
  21 mix2 0 19 20 16 0   (order -1)
hcomp
  c++ *c=a b=c a=0 (save in rotating buffer)
  d= 2 hash *d=a b-- (orders 1,2,3,4,5,7)
  d++ hash *d=a b--
  d++ hash *d=a b--
  d++ hash *d=a b--
  d++ hash *d=a b--
  d++ hash b-- hash *d=a b--
  d++ hash *d=a b-- (match, order 8)
  d++ a=*c a&~ 32 (case insensitive words)
  a> 64 if
    a< 91 if (if a-z)
      d++ hashd d-- (update order 1 word hash)
      *d<>a a+=*d a*= 20 *d=a (order 0 word hash)
      jmp 9
    endif
  endif
  (else not a letter)
    a=*d a== 0 ifnot (move word order 0 to 1)
      d++ *d=a d--
    endif
    *d=0  (clear order 0 word hash)
  (end else)
  d++
  d++ b=c b-- a=0 hash *d=a (sparse 2)
  d++ b-- a=0 hash *d=a (sparse 3)
  d++ b-- a=0 hash *d=a (sparse 4)
  d++ a=b a-= 212 b=a a=0 hash
    *d=a b<>a a-= 216 b<>a a=*b a&= 60 hashd (pic)
  d++ a=*c a<<= 9 *d=a (mix)
  d++
  d++
  d++ d++
  d++ *d=a (sse)
  halt
post
  0
end
```

The COMP section begins with an ISSE chain of orders 0 through 6 like mid.cfg (with one extra ISSE). "const 160" provides a bias for the mixers that follow later. It outputs a fixed prediction of (160-128)/16 = 2 (stretched). The match model is order 8. As with mid.cfg, M is used as a rotating history buffer, but with a size of $2^{hm} = 2^9 = 512$. H is $2^{hh} = 32$ elements. There are n = 22 components.

Components 9 and 10 are an ISSE chain of word-oriented order 0 and 1 contexts for modeling text. These form a separate chain. Generally, the best compression is obtained when each ISSE context contains the lower order context of its input. Otherwise the models should be independent and mixed later. The context is formed by mapping upper and lower case letters together and discarding all other input. The order 0 context is a hash of all of the letters since the beginning of the word or 0 if the last byte was not a letter. The order 1 hash combines this with the previous word.

Components 11 through 13 are sparse order 1 contexts with a gap of 1 to 3 bytes between the context and the current byte. These are useful for modeling binary files.

Component 14 is a model for CCITT binary fax images (PIC in the Calgary corpus). The image width is 1728 pixels or 216 bytes, mapped one bit per pixel in MSB to LSB order (0=white, 1=black). The context is the 8 bits from the previous scan line and 2 additional bits from the second to last scan line.

Components 15 and 16 are order 1 and 0 mixers taking all earlier components as inputs. The second (order 0) mixer has a slower learning rate because it has fewer free parameters. Those two mixer outputs are mixed by the context free (size 0) MIX2 at 17. Its output is refined by the order 0 SSE at 18. The input and output of the SSE are mixed at 19. That prediction is refined by the order 1 SSE at 20. Finally the input and output of that SSE are mixed by the context free MIX2 at 21.

The code for computing the order 0 and 1 word contexts in H[9..10] starts at

```
d++ a=*c a&~ 32 (lowercase words)
```

This increments D to point to H[9] (the order 0 word model), retrieves the input byte saved in M[C], and clears bit 5 (meaning a &= ~32) which converts lower case to upper case. Then

```
a> 64 if
a< 91 if (if a-z)
```

tests if A is in the range A..Z (ASCII 65..90). The "if" is converted to a conditional jump to the matching "else" or "endif". If the test passes then the two word hashes are updated. The instruction *d<>a means swap H[D] with A. The hash in D[9] is a rolling hash but in D[10] is cumulative. JMP 9 skips 9 bytes past the commented "else" clause. If the input is not a letter then H[9] is moved to H[10] and H[9] is cleared.

The following results are for the Calgary corpus as a solid archive when supported. Compression is timed in seconds on a 2 GHz T3200.

Program	Size	Time	Memory (MB)	
zip -9	1,020,831	0.6	0.5	
ppmd	804,992	0.6	7.5	(calgary.tar)
ppmd -m256 -o16	754,243	1.3	62	(calgary.tar)
ppmonstr	669,497	8	51	(calgary.tar)
lpaq1 6	682,512	8	195	(calgary.tar)
lpaq9m 6	686,161	8	198	(calgary.tar)
paq9a -6	676,914	12	209	
zpaq ocmid.cfg	699,474	8	111	
zpaq ocmax.cfg	644,433	20	246	
zpaq ocmax.cfg,1	644,320	20	477	
paq8l -6	595,474	368	435	
paq8px_v67 -6	598,969	469	421	

5. Transforms

A transform converts data into a sequence of symbols which can be compressed with a simpler or faster model, or one using less memory, such as an order 0 context model. Those symbols still need to be modeled and coded as described in sections 4 and 3 respectively.

A transform should ideally be a bijection. For each input, there should be exactly one possible representation. More often, the transform is an injection, and its inverse a surjection. An input may have more than one valid representation, either of which is transformed back to the original data during decompression. This increases the information content of the transformed data because the arbitrary choice of representation has to be modeled and coded, taking up space.

5.1. Run Length Encoding

A run length code replaces a long repeating sequence of identical symbols with two symbols representing a count and the value to be repeated. For example, the string "AAAAA" would be coded as (5,"A").

5.2. LZ77 and Deduplication

In LZ77 compression, duplicate strings in the input are replaced by pointers to previous occurrences. LZ77 is not a bijection. For example, given the string:

```
AB..BC..ABC
```

"ABC" could be coded as:

- a pointer to AB, and literal C,
- a literal A and pointer to BC,
- or 3 literals, A, B, C.

A pointer consists of an offset and a length. It is allowed for the copied string to overlap the output. For example "AAAAA" could be coded as a A,(-1,4) meaning write a literal "A" and then go back 1 and copy the next 4 bytes. Thus, LZ77 may also encode run lengths.

LZ77 decompression is extremely fast, faster than compression. The compressor must search for matching strings, typically using a hash table or tree. The decompresser only needs to maintain an output buffer from which to copy repeated strings, and then write a copy of its output to the buffer.

The name "LZ77" comes from Lempel and Ziv, who described it in a 1977 paper (Ziv and Lempel, 1977).

5.2.1 LZSS

LZSS (Lempel-Ziv-Storer-Szymanski, 1982) uses 1 bit flags to mark whether the next symbol is a literal or a pointer. LZSS is used in NTFS file compression in Windows when the folder property is set to store files compressed. Its primary goal is to be extremely fast (faster than disk access) rather than provide good compression. The exact format was not published. Rather, it was reverse engineered (in Russian) in 1998. 16 literal/pointer flags are packed into 2 bytes. This is followed by 16 symbols which are either 1 byte literals or 2 byte pointers. The offset is variable length with a maximum of 12 bits. Any remaining bits are allocated to the length, which has a minimum value of 3. Thus, after 2K of input, each pointer is a 12 bit offset and a 4 bit length ranging from 3 to 18.

Windows indicates that a compressed folder containing the Calgary corpus occupies 1,916,928 bytes. On the large text benchmark, the 1 GB text file enwik9 compresses to 636 MB, slightly larger than an order 0 coder and about twice the size of zip. Copying enwik9 between 2 uncompressed folders takes 41 seconds on the test machine (a laptop with a 2.0 GHz T3200). Copying from a compressed folder to an uncompressed folder takes 35 seconds, i.e. decompression is faster than copying. Copying from an uncompressed folder to a compressed folder takes 51 seconds. This is equivalent to compressing the Calgary corpus in 0.03 seconds over the time to copy it.

The NTFS implementation of LZSS is very similar to lzrw1-a implemented by Ross Williams in 1991. lzrw1-a uses a fixed 12 bit offset and 4 bit length.

5.2.2. Deflate

The widely popular deflate format is used in zip and gzip and is supported by many other archivers. It is used internally in PNG images, PDF documents, and Java JAR archives. The format is documented in RFC 1951 (1996) and supported by the open source zlib library.

In the deflate format, pointer offsets range from 1 to 32768 and length from 3 to 258. Literals and lengths are coded in a 286 symbol alphabet which is Huffman coded followed by up to 5 extra uncompressed bits of the length. A length code is followed by an offset from a 30 symbol Huffman coded alphabet followed by up to 13 extra uncompressed bits. Specifically the alphabets are as follows:

```
0..255 = literal byte
256 = end of data
257..264 = lengths 3..10
265..268 = lengths 11..18 followed by 1 extra bit
269..272 = lengths 19..34, 2 extra bits
273..276 = lengths 35..66, 3 extra bits
277..280 = lengths 67..130, 4 extra bits
281..284 = lengths 131..257, 5 extra bits
285 = length 258
```

Lengths are followed by an offset coded from a 30 symbol alphabet:

```
0..3 = offset 1..4
4..5 = offset 5..8 followed by 1 extra bit
6..7 = offset 9..16, 2 extra bits
8..9 = offset 17..32, 3 extra bits
...
28..29 = offset 16385..32768, 13 extra bits
```

The format allows either a default or a custom Huffman code. The default code lengths are as follows:

```
Literal/length
0..143 = 8 bits
144..255 = 9 bits
256..279 = 7 bits
280..287 = 8 bits

Offset
0..29 = 5 bits
```

If a custom Huffman table is used, then the table is transmitted as a sequence of code lengths. That sequence is itself compressed by run length encoding using another Huffman code to encode the literals and run lengths. It uses a 19 symbol alphabet:

```
0..15 = code lengths of 0..15
16 = copy the previous code 3..6 times, followed by 2 extra bits
17 = copy 3..10 times, 3 extra bits
18 = copy 11..138 times, 7 extra bits
```

The Huffman table for these codes are sent as a sequence of up to 19 3-bit numbers. This sequence is further compressed by reordering the sequence so that the values

most likely to be 0 (not used) are at the end, and sending the sequence only up to the last nonzero value. A 4 bit number indicates the sequence length. The order is: 16, 17, 18, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15. All Huffman codes are packed in LSB to MSB order.

zip and gzip take an option -1 through -9 to select compression level at the expense of speed. All options produce compressed data in deflate format which decompresses at the same speed (much faster than compression) with the same algorithm. The difference is that with the higher options, the compressor spends more time looking for encodings that compress better. A typical implementation will keep a list of 3 byte matches (the shortest allowed) in a hash table and test the following data to find the longest match. With a higher option, the compressor will spend more time searching. It is also sometimes possible to improve compression by encoding a literal even if a match is found, if it results in a longer match starting at the next byte. Such testing also increases compression time. kzip performs an extreme level of optimizations like this. Compressed sizes and compression times on a 2.0 GHz T3200 are shown below for the 14 file Calgary corpus.

Program	Size	Time	
zip -1	1,194,720	.17 sec.	
zip -2	1,151,711	.23	
zip -3	1,115,078	.25	
zip -4	1,072,909	.25	
zip -5	1,041,083	.33	
zip -6	1,028,171	.40	(default)
zip -7	1,025,244	.42	
zip -8	1,021,607	.50	
zip -9	1,020,831	.67	
kzip	978,707	24.21	
unzip		.10	

5.2.3. LZMA

LZMA (Lempel Ziv Markov Algorithm) is the native compression mode of 7-zip. Compression is improved by using a longer history buffer (selectable up to 4 GB) which allows more matches to be found. Symbols are arithmetic coded using a context model.

5.2.4. LZX

LZX is an LZ77 variant used in Microsoft CAB files and compressed help (CHM) files. It uses a history buffer of up to 2 MB and Huffman coding. To improve compression, it uses shorter codes to code the 3 most recent matches.

5.2.5, ROLZ and LZP

The idea of using shorter codes for recent matches can be extended. The compressor for lzrw3 builds a dictionary (a hash table) of pointers into the history buffer as usual to find matching strings, but instead of coding the offset, it codes the index into the table. The decompresser builds an identical hash table from the data it has already decompressed, then uses the index to find the match. The length is coded as usual.

ROLZ (reduced offset LZ) extends this idea further by replacing the large hash table with many smaller hash tables selected by a low order context. This reduces the size of the offset, although it can sometimes cause the best match to be missed. ROLZ was implemented in WinRK.

The extreme case of ROLZ is to use one element per hash table. In this case, only a literal or length must be coded. This algorithm is called LZP. It was first described by Charles Bloom in 1995. LZP works best with a high order context. Thus, it is often used as a preprocessor to a low or moderate order context model, rather than a fast order 0 model like LZ77.

5.2.6. Deduplication

Deduplication is the application of LZ77 to a file system rather than a data stream. The idea is to find duplicate files or files containing large blocks of data duplicated elsewhere, and replace them with pointers.

5.3. LZW and Dictionary Encoding

Dictionary methods substitute codes for common strings from a table or dictionary. A dictionary code may be, fixed, static or dynamic. In the fixed case, the dictionary is specified as part of the algorithm. In the static case, the compressor analyzes the input, constructs a dictionary, and transmits it to the decompresser. In the dynamic case, both the compressor and decompresser construct identical dictionaries from past data using identical algorithms.

5.3.1. LZW

LZW (Lempel-Ziv-Welch) is a dynamic dictionary method. It is used by the UNIX compress program, GIF images, and is one of the compressed modes in TIFF images. The algorithm was patented by both Sperry (later Unisys) in 1981 and by IBM and 1983 when the USPTO did not realize that they were the same algorithm. Unisys was criticized for waiting until GIF became an established standard before demanding royalties from makers of software that could read or write GIF images. Both patents are now expired.

LZW starts with a dictionary of 256 1-byte symbols. It parses the input into the longest possible strings that match a dictionary entry, then replaces the string with its index. After each encoding, that string plus the byte that follows it is added to the dictionary. For example, if the input is ABCABCABCABC then the encoding is as follows:

```
 65 = A   (add AB to dictionary as code 256)
 66 = B   (add BC as 257)
 67 = C   (add CA as 258)
256 = AB  (add ABC as 259)
258 = CA  (add CAB as 260)
257 = BC  (add BCA 261)
259 = ABC (end of input)
```

Dictionary codes grow in length as it becomes larger. When the size is 257 to 512, each code has 9 bits. When it is 513 to 1024, each code is 10 bits, and so on. When the dictionary is full (64K = 16 bits), it is discarded and re-initialized.

A Windows version of the UNIX compress program compresses the Calgary corpus to 14 files totaling 1,272,722 bytes in 0.34 seconds and decompresses in 0.23 seconds.

Other variants of LZW may use larger dictionaries, or may use other replacement strategies like LRU (least recently used), or other strategies for adding new symbols such as concatenating the last two coded symbols instead of a symbol plus the next byte.

5.3.2. Dictionary Encoding

Dictionary encoding improves the compression of text files by replacing whole words with symbols ranging from 1 to 3 bytes. Fixed English dictionaries are used in WinRK, durilca, and in some versions of PAQ such as PAsQDacc 4.3c -7, which compresses the Calgary corpus to 567,668 using a dictionary extracted from the corpus itself, but not included in the compressed size. It is, of course, possible to compress to arbitrarily small sizes using this technique. The extreme case is barf. It has a built in 14 word dictionary, one for each file of the Calgary corpus. When the compressor detects a match, it "compresses" the file to 1 byte, which the decompresser correctly expands.

For this reason, the large text benchmark and contests like the Calgary challenge and Hutter prize include the size of the decompression program and all other files needed to decompress. Still, it may be useful to use a dictionary for one or more languages if the input is expected to contain text in those languages.

Of more interest are static dictionaries. These are used by the top 3 programs on the large text benchmark (as of Feb. 2010), and in all of the Hutter prize winners. Some of the later Calgary challenge winners also use small dictionaries.

5.3.2.1. Modeling Text

Recall from section 1.4 that text compression is an AI problem. This can be seen by playing Shannon's character guessing game which he used to estimate that the entropy of written English is about 1 bpc (Shannon, 1950). Try partially covering some text with your hand and guessing what letters come next from the earlier context, for example: "the cat caught a mo___". Humans can beat computers at the game because the prediction problem requires vast understanding of English and of the world. Nevertheless, some of the constraints of natural language can be modeled. These rules are categorized as follows:

- lexical: "moqse" is wrong because it is not a word.

- semantic: "moose" is wrong because it is not associated with things that a cat would chase.
- syntactic or grammatical: "moves" is wrong because we expect "a" to be followed by a singular noun phrase.

While playing the game, you will notice that useful contexts start on word boundaries. Thus, "a mo_" and "caught a mo_" are useful contexts, but "ght a mo_" is no more useful than the lower order "a mo_". Thus, text models in PAQ construct contexts that start on word boundaries.

It should be irrelevant if a context spans a line break. Thus, word contexts in PAQ discard the characters between words. Furthermore, it should be irrelevant if the context is upper or lower case, because it does not change the meaning. Thus, there are word models that ignore case.

Semantics can be modeled by associating each pair of words like (cat, mouse) with a co-occurrence frequency over a small window. Words that frequenly occur near each other tend to have related meanings. This can be modeled with a sparse order-1 word model, skipping one or more words in between the context and the predicted word. Many PAQ versions have sparse word models with small gaps of 1 to 3 words.

For modeling semantics, it is useful to split text into "meaningful" units or morphemes if possible. For example, "moves" really has two independent parts, the stem "move" and suffix "s". Ideally these should be modeled as separate words.

Grammar constrains text to make certain sequences more likely, such as (the NOUN) or (a ADJ NOUN). It is possible to learn the parts of speech by observing when words occur in similar contexts and grouping them. For example "the dog", "the cat", and "a dog" could be used to predict the unseen sequence "a cat". This works by grouping "the" and "a" into one semantic category and "dog" and "cat" into another category.

A dictionary transform works by replacing the input text with a sequence of highly predictive symbols corresponding to morphemes, independent of capitalization and punctuation. This improves compression both by allowing simpler models to be used, and by reducing the size of the input, which improves speed and reduces pressure on memory. Compression can be improved further by grouping semantically or grammatically related words so that the compressor can merge them into single contexts by ignoring the low bits of dictionary codes. Care should be taken not to remove useful context, which can happen if a dictionary is too large or divides words in the wrong places. For example, coding "move" and "remove" as unrelated symbols would not help compression.

5.3.2.2. Capitalization Modeling

A capitalization model replaces upper case letters with a special symbol followed by a lower case letter. For example, "The" could be coded as "Athe" where "A" directs the decompresser to capitalize the next letter. Alternatively, a more sophisticated model might automatically capitalize the first letter after a period, and insert a special symbol to encode exceptions.

5.3.2.3. Newline Modeling

Because newlines are semantically equivalent to spaces, it is sometimes useful to replace then with spaces, and encode in a separate stream the information to put them back. A simple transform is space stuffing, where a space is inserted after every newline. For example, this has the effect of merging the order 4 contexts " the" and "\nthe" (where \n is a linefeed) by replacing the latter with "\n the". Space stuffing does not help with multi-word contexts that span lines. However the alternative is to remove context that could predict newlines, such as periods at the end of a paragraph.

5.3.2.4. Word Encoding

Word encoding is done in two passes. In the first pass, the text is parsed into words (sequences of A-Z, a-z, and possibly UTF-8 characters in non-English alphabets) and counted. Words with counts below a threshold are discarded. In the second pass, words found in the dictionary are replaced with 1 or 2 byte codes (or 3 bytes for large dictionaries). The dictionary is listed at the beginning of the output, followed by the encoded data. Words not found in the dictionary and non-letters are passed unchanged.

Words are typically encoded with bytes from the part of the ASCII set that does not normally appear in text, namely 0..8, 11..12, 14..31, and 127..255. If capitalization modeling was done, then 65..90 (A-Z) may also be used. If such bytes do appear, they must be preceded by an escape byte, designated as one of the above. The remainder of the alphabet may be used to encode words.

XWRT (XML Word Reducing Transform) by Przemyslaw Skibinski in Oct. 2007 performs dictionary encoding. The dictionary is appended as a header to the output with one word per line in decreasing order of frequency. The most frequent words are encoded with one byte.

5.3.2.5. Results

The following table shows the effect of simple capitalization modeling (using "A" followed by lowercase), space stuffing, and word encoding using XWRT on book1 from the Calgary corpus on 4 compressors. The -f option selects the minimum word frequency for inclusion in the dictionary. The number in parenthesis shows the dictionary size that results. (The exact options are -o -l0 -c -s -n -w -m256 for xwrt 3.2 to turn off other transform options. There is no space or capitalization modeling).

	book1	zip -9	sr2	bzip2 -9	ppmonstr
No transform	768,771	312,502	276,364	232,598	203,247
Capitalization	785,101	311,696	275,124	231,594	202,650
Space stuffing	785,393	313,640	275,161	229,988	202,274
Both	801,723	312,856	273,864	229,861	201,706
xwrt -f3 (4307)	366,323	265,721	246,897	233,928	211,382
xwrt -f6 (2806)	378,289	267,522	246,760	231,675	208,801
xwrt -f20 (789)	449,233	278,639	254,227	230,243	204,897
xwrt -f100 (174)	542,670	290,268	262,575	228,904	202,832

The table shows that space stuffing and capitalization usually help, but that word encoding becomes less effective as the compression improves. It is nevertheless useful for compressing the large text benchmark where memory constraints are severe because it reduces the size of the input. The top 3 programs use it. Capitalization modeling is also useful, but space stuffing is not because line breaks are only used to separate paragraphs.

XWRT is ranked sixth (as of Feb. 2010) on the large text benchmark when used with its built in LPAQ6 compressor. The optimal setting in this case is -f200 to select a dictionary size of about 40,000 words. It does slightly better (ranking fifth) as a preprocessor to ppmonstr with option -f64.

paq8hp12 and drt|lpaq9m, both by Alexander Ratushnyak, are ranked second and third on the large text benchmark and are the basis of winning entries for the Hutter prize. These both use a custom dictionary of about 44,000 words. The higher frequency words are grouped semantically, such as "son" with "daughter" and "monday" with "tuesday". Among the lower frequency words, they are grouped by common suffix (alphabetical order when reversed) to make the dictionary compress smaller.

durilca_kingsize is the top ranked program on the large text benchmark, but only because it uses 13 GB of memory, vs. 2 GB. It uses a dictionary of about 124,000 words. These are also grouped semantically, but by an automated process that clustered words in context space. The algorithm was not documented, but the idea is roughly to group words together if they are likely to appear in the same context.

5.4. Symbol Ranking

Symbol ranking, or move-to-front (MTF), is a transform in which the alphabet is maintained in a queue from newest to oldest and coded by its position. The idea is that the most recently seen symbol is the most likely to occur in the future.

srank is a symbol ranking compressor by Peter Fenwick in 1996. An order 3 context hash without collision detection is mapped to a queue of length 3 representing the 3 most frequently seen bytes in that context. These are Huffman coded with 1, 3, or 4 bits respectively. Long runs of first place bytes are run length encoded with 12 bits to encode the run length. Bytes not seen in the queue are modeled in an order 0 pseudo-MTF queue using 7 bits for the first 32 positions and 12 bits for the other 224. It is called "pseudo-MTF" because when a byte is observed, it is moved only about half way to the front with some random dithering. This is an

optimization for speed. It allows a fast update of an index into the queue. The order 3 hash table maximum size is 2^{18} queues (1 MB memory).

sr2 is an improved (but slower) symbol ranking compressor by Matt Mahoney in Aug. 2007. An order 4 context hash is mapped to a table of 2^{20} 3 byte MTF queues and a counter for consecutive first place hits ranging from 0 to 63. When the first place byte is observed, the counter is incremented. For all other values, the counter is reset to 1 if in the queue or 0 if not. The new value is pushed to the front of the queue and the others pushed back. For example, the sequence ABCCC would result in the queue (C,B,A,3) with C at the front. A subsequent B would result in (B,C,A,1). A subsequent D would result in (D,B,C,0).

A byte is first Huffman coded and then arithmetic coded. The point of the Huffman code is to reduce the number of arithmetic coding operations for better speed. Suppose the queue contains (c1,c2,c3,n). The coding and next state is as follows:

```
  Input     Code         (c1  c2  c3  n) next
state
  -----     ----         --------------------------
  Initial                (0,  0,  0,  0)
    c1      0            (c1, c2, c3, min(n+1, 63))
    c2      110          (c2, c1, c3, 1)
    c3      111          (c3, c1, c2, 1)
 other c    10cccccccc   (c,  c1, c2, 0)
```

The bits are coded using a direct context model with a count ranging from 2 to 128 (section 4.1.2). For n ≥ 4, the context is order 1 plus n plus the previous bits of the current symbol. For n < 4, the model is the same except order 2.

The following comparison is for the Calgary corpus as 14 files compressed separately.

```
Program        Size      Compr  Decompression
---------    ---------   ----   --------
srank -C8    1,281,984   0.20   0.20 sec.
sr2            975,208   0.48   0.50 sec.
```

During development, it was observed that an order 3 context sometimes compressed better on smaller files, but order 4 works better on larger files. Increasing the hash table beyond 2^{20} did not help much, in spite of the fact that more memory almost always helps any algorithm.

Arithmetic decoding is slightly slower than encoding. Recall that the steps to compress are:

1. predict next bit
2. code the bit
3. update the model

For decompression:

1. predict next bit
2. decode the bit
3. update the model

Modern processes can reorder instructions and execute them in parallel. During compression, steps 2 and 3 are independent, so they can overlap. During decompression, the model cannot be updated until the bit has finished being decoded.

5.5. Context Sorting (BWT)

A Burrows-Wheeler Transform sorts the input by its right context. By bringing together characters with similar contexts, the transformed data can be more easily compressed with a fast adapting order 0 model. Shown below is a portion of the Burrows Wheeler transform of book1 from the Calgary corpus (with newlines converted to spaces for clarity). The column in bold is the BWT.

```
BWT block --+    +--- Sorted on this column
            \  /
             vv
  ing, Her culpability lay in her m
  e of the instability of a woman ?
  hat the desirability of her exist
  tion, from inability to guide inc
   husband's inability to meet the
  nervous excitability, he returned
  stimony, probability, induction -
  le of respectability, were now si
  of a ship's cabin, with wood sild
   new riding-habit -- the most ele
   mostly her habit hen excited, he
  's virtuous habit of entering the
   new riding-habit of myrtle-green
  aracter and habit, and seemed so
  e no riding-habit, looked around
  besides the habitable inn itself,
  ceived no inhabitant for the spac
    by the inhabitants of Caster+
  ttle <P 61> habitation, and here
  every human habitation, and the h
  those old-inhabited walls. It was
  ntly to old habits and usages, si
  o imply his habitual reception of
  rgrass, who habitually spoke on a
  at everybody abjured her -- for w
  osed all the able-bodied men upon
  en the favourable-con+ junction s
  ame to the stable-door and looked
  d been answerable .! ' " We must
```

The BWT is the column in **bold**. It is "...ptrnntbtchhhhhhhhhhhhhhh rtr...".

BWT is best suited for stationary sources. For example, a sorted list of words would be compressed poorly because local rules (newline is followed by "A", later changing to "B") become spread throughout the transform. For these cases, separating different data types into independently compressed blocks can improve compression. Otherwise, the largest possible block size should be used.

BWT compression depends on the alphabet order. Best compression is obtained when related symbols such as letters or digits that make similar predictions are grouped together. The ASCII character set already has this property, but is not optimal.

5.5.1. Forward transform

Compressors that use BWT are called context sorting or block sorting algorithms. A typical implementation is to divide the input into fixed sized blocks (as large as memory allows) and sort an array (of the same size) of pointers into the block. The sort order is the lexicographical order of the string to which it points, wrapping around to the beginning of the block if necessary. Wrapping around can be avoided by appending a virtual "end of block" symbol which is lexicographically less than any of the symbols in the alphabet (e.g. -1).

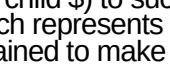
A straightforward sort using a fast sorting algorithm like a radix sort or quicksort can be very slow on highly redundant or compressible data. Quicksort requires on average $O(n \log n)$ string comparisons, but worst case string comparison is $O(n)$, as in the case of a block of identical bytes. This would result in $O(n^2 \log n)$ run time. There are several ways to avoid the problem of long common prefixes.

- Flip a small fraction of the bits using a pseudo-random algorithm to break up long strings. This hurts compression slightly.
- Preprocess the data using a high order LZP or LZ77 preprocessor.
- Use a Schindler transform, a variation in which the sort order is based on a truncated string comparison. However, this method is protected by patent 6,199,064 in the U.S. The patent expires Nov. 14, 2017.
- Use a suffix array sort, which is $O(n)$ worst case.

There are fast suffix sorting algorithms that use 5 or 6 times the block size in memory. The algorithms are variations of a suffix tree sorting algorithm developed by Ukkonen (1995), which is also $O(n)$ but requires more memory. The idea is to build a tree in which each path from root node to a leaf node represents one of the n suffixes of the block. Each edge is labeled with a string of one or more bytes (represented as a pair of pointers into the block) Each of the n leaf nodes is a pointer to the start of the suffix. Each internal node except possibly the root node is required to have two or more children (in lexicographical order from left to right). This limits the number of nodes to at most 2n. The algorithm is to build the tree and then output the sorted suffix pointers by a recursive inorder traversal from the root node:

```
traverse(node):
 if the node is a leaf then
   output block[leaf-1]
 else
   for each child from left to right
     traverse(child).
```

A suffix tree for the string "BANANA" with end of block symbol "$" is shown below. The downward arrows are called transitions. Each transition is labeled with a string of one or more bytes. The dotted lines are called suffix links. They form a path from the longest suffix represented by a nonterminal node (having a child $) to successively shorter suffixes back to the root (which represents the empty suffix). The suffix links are maintained to make the tree building algorithm efficient.



A suffix tree of "BANANA$" (from Wikipedia).

Ukkonen's algorithm is to start with just a root node and scan the block from left to right in a single pass. For each byte, the suffix pointers form a linked list of nodes that need to be updated. An update consists of appending a child node, splitting edges as needed, and updating the suffix pointers. The suffix pointer traverse stops when there is already a child node for the character being added.

A suffix array represents a suffix tree using just the block, the sorted list of pointers, and the longest common prefix (LCP), which is the number of

initial bytes shared with the previous suffix after sorting. The suffix array of "BANANA$" is shown:

```
  Index  Suffix  LCP
  -----  ------  ---
    6    $        0
    5    A$       0
    3    ANA$     1
    1    ANANA$   3
    0    BANANA$  0
    4    NA$      0
    2    NANA$    2
```

Yuta Mori has benchmarked several suffix array sorting algorithms. The fastest are his own libdivsufsort, MSufSort, used in M99 and M03 by Michael Maniscalco, and archon4r0 used in the dark archiver by Dima Malyshev. All of the sorting algorithms are open source.

5.5.2. Inverse transform

It is rather surprising that a BWT block can be inverted to recover the original data. The only other information needed is the new position of the original first byte. We are given a BWT string BWT[0..n-1] of length n, and the location p (0 ≤ p < n) of the position of the first byte BWT[p]. The algorithm to output the original string is:

```
  T = sort(BWT)
  Repeat n times:
    output BWT[p]
    move p from the i'th location of c in T to the i'th
location of c in BWT
```

For example, suppose that BWT[0..5] = "NNBAAA" is the BWT of "BANANA". as shown. (To simplify the explanation, we omit the $, but it works the same either way).

```
  BWT  Sorted context
   \/
   NABANA
   NANABA
   BANANA p=2
   ABANAN
   ANABAN
   ANANAB
```

Create T[0..5] = "AAABNN". We now have:

```
              p
        0 1 2 3 4 5
  BWT = N N B A A A
  T   = A A A B N N
```

The steps are:

```
 output BWT[2] = B
p is the third A in T. Move to p=5, the third A in BWT.
output BWT[5] = A
p is the second N in T. Move to p=1, the second N in BWT.
output BWT[1] = N
p is the second A in T. Move to p=4, the second A in BWT.
output BWT[4] = A
p is the first N in T. Move to p=0, the first N in BWT.
output BWT[0] = N
p is the first A in T. Move to p=3, the first A in BWT.
output BWT[3] = A.
```

As an optimization, we may represented the sorted array T solely by the starting position of each of the 256 sequences of byte values, for example A=0, B=3, C=3,..., N=4, O=6. Furthermore, we can construct in advance a list NEXT[0..n-1] such that NEXT[p] is the next move for p. For example, NEXT[2] = 5 would be the first move. To build this list we scan BWT and use T to count the occurrence of each byte value.

```
    for i in 0..n-1 do
    NEXT[T[BWT[i]]++] = i
```

In C++, the inverse BWT looks like this:

```
 // Invert and output the BWT in bwt[0...n-1] starting at p
 void invert_BWT(unsigned char *bwt, int n, int p) {

    // Collect cumulative counts of bwt:
    // t[i] = number of bytes < i
    int t[257]={0}; // cumulative counts
    for (int i=0; i<n; ++i)
      ++t[bwt[i]+1];
    for (int i=1; i<257; ++i)
      t[i]+=t[i-1];
    assert(t[256]==n);

    // Build linked list
    int *next=calloc(n, sizeof(int));  // linked list
    assert(next);  // out of memory?
    for (int i=0; i<n; ++i)
      next[t[bwt[i]]++]=i;
    assert(t[255]==n);

    // Traverse and output list
    for (int i=0; i<n; ++i) {
      assert(p>=0 && p<n);
      putc(bwt[p], out);
      p=next[p];
    }
    free(next);
  }
```

5.5.3. bzip2

bzip2 is a popular open source BWT based file compressor developed in 1996 by Julian Seward. It takes an option -1 through -9 to select a block size of 100 KB to 900 KB. -9 generally gives the best compression. The compression algorithm is as follows:

1. The input is run length encoded to remove some (not all) high order redundancy. Sequences of 4 to 255 repeated bytes are coded as the first 4 bytes followed by one byte (0..251) representing the remaining count. For example, "AAAAA" is coded as "AAAA",1.
2. BWT.
3. Move to front (section 5.4). Each byte is coded as its position in a queue, then moved to the front of the queue. Runs of identical characters thus become runs of zeros.
4. Run length encoding of of zeros. The run length is coded in binary in LSB to MSB order by two symbols (RUNA, RUNB) that have values 1 and 2 (instead of 0 and 1). Runs of length 1 through 10 would be coded as 1, 2, 11, 21, 12, 22, 111, 211, 121, 221.
5. The symbols RUNA, RUNB, queue positions 1..255 (0 is always run length encoded) and end of data symbol are Huffman coded.

bzip2 uses 2 to 6 Huffman tables, which are selected every 50 symbols to make the code adaptive. The tables are kept in a MTF queue. The selection code is unary coded. A unary code for a number n is n 1 bits and a 0. For example, 4 = 11110.

The Huffman tables are coded as a sequence of lengths. The lengths are delta coded, i.e. as the difference from the previous length. A difference is coded as 0 = 0, 10 = -1, 11 = +1, repeating as needed. A bitmap is used to mark unused queue selection codes, which are omitted from the sequence. The bitmap is divided into 16 16-bit words, where a 0 bit means the code is not used. If all 16 bits are 0, then the word is omitted. One additional 16 bit word is used to mark which words are omitted.

The original bzip was arithmetic coded, which is better suited for a fast adapting model. It was replaced with a Huffman code due to patents (now expired) on arithmetic coding.

5.5.4. BBB

BBB (big block BWT) is an open source file compressor written by Matt Mahoney in Aug. 2006. It has two innovations: a "slow" mode that allows blocks up to 80% of available memory, and a context mixing model of the BWT sequence. When released, it was top ranked on the large text benchmark among BWT compressors because it was the only program that could fit the 1 GB test file into a single block on a 2 GB machine.

5.5.4.1. Slow Mode BWT

To context sort a large block, it is first divided into 16 smaller blocks of 4-byte pointers which are sorted with wraparound using std::stable_sort() (normally a generic comparison sort such as quicksort). The pointers are then written to 16 temporary files, which are merged to produce the final result.

The inverse transform does not build a linked list, because this takes 4 times as much memory as the block size. Recall that the inverse transform first sorts the bytes in the block into an array T (represented by 256 cumulative counts), and that p points to the next output byte in the block. The steps to be iterated are:

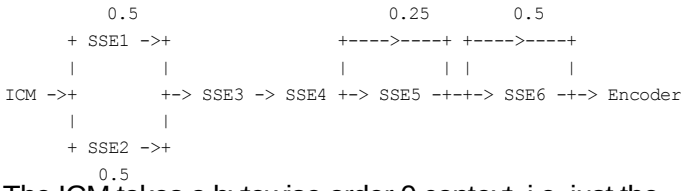
1. output BWT[p]
2. if T[p] is the i'th occurrence of c in T, then set p to the i'th occurrence of c in BWT.

Normally, step 2 is done by traversing a link in the list NEXT. Instead, BBB searches the block for the i'th

occurrence of c. To do this efficiently, it first consults an index that locates every 16'th occurrence of c in BWT to get the approximate location, and searches linearly from there. This index takes 1/4 as much memory as the BWT block.

5.5.4.2. Modeling

BBB uses an order 0 indirect context model (section 4.1.3) followed by 6 SSE stages and bitwise arithmetic coding. The model uses 5 MB of memory. It looks like this:



The ICM takes a bytewise order 0 context, i.e. just the previous bits of the current byte. Recall that an ICM maps a context to a bit history (an 8 bit state), which is mapped to a slow adapting probability table.

Each of the SSE (section 4.3.3) maps a context and a probability (stretched and quantized to 32 levels) to a new probability interpolated between the two nearest quantized outputs. SSE1 and SSE2 are both order 0, but SSE1 is fast adapting (learning rate 1/32) and SSE2 is slow adapting (learning rate 1/512). The two predictions are averaged in the linear domain. The SSE in BBB update both quantized table entries above and below the input probability, unlike ZPAQ which updates only the nearest.

SSE3 takes a bytewise order 1 context. SSE4 takes the previous but not the current byte as context, plus the run length quantized to 4 levels (0, 1, 2..3, 4+).

SSE5 takes a sparse order 1 context of just the low 5 bits and a gap of 1 byte, i.e. 5 of the last 16 bits, plus the current byte: ...xxxxx ..... The output is averaged linearly with the input with weight 3/4 to the output.

SSE6 takes a 14 bit hash of the order 3 context. It is averaged linearly with its input with weight 1/2 each.

The table below compares the models for bzip2 and BBB on the Calgary corpus with each file compressed separately. In both cases the block size is 900 KB, which is large enough to hold each file in a single block. BBB is run in both fast and slow modes. About half of the compression time in both cases is due to PIC, which has long runs of 0 bytes. Unlike bzip2, BBB has no protection against long string comparisons while sorting.

Note also that compressing all of the data together as a tar file makes compression worse. As mentioned, BWT is poorly suited for mixed data types.

Program	Calgary	Compr	Decomp (seconds)	calgary.tar
bzip2 -9	828,347	0.68	0.42	860,097
bbb cfk900	785,672	10.33	1.46 (fast mode)	800,762
bbb ck900	785,672	13.74	5.54 (slow mode)	

5.6. Predictive Filtering

A predictive filter is a transform which can be used to compress numeric data such as audio, images, or video. The idea is to predict the next sample, and then encode the difference (the error) with an order 0 model. The decompresser makes the same sequence of predictions and adds them to the decoded prediction errors. Better predictions lead to smaller errors, which generally compress better.

5.6.1. Delta Coding

The simplest predictive filter is a delta code. The predicted value is just the previous value. For example, the sequence (5,6,7,9,8) would be delta coded as (5,1,1,2,-1). A second pass would result in (5,-4,0,1,-3).

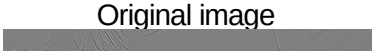
Delta coding works well on sampled waveforms containing only low frequencies (relative to the sampling rate), such as blurry images or low sounds. Delta coding computes a discrete derivative. Consider what happens in the frequency domain. A discrete Fourier transform represents the data as a sum of sine waves of different frequencies and phases. In the case of a sine wave with frequency ω radians per sample and amplitude A, the derivative is another sine wave with the same frequency and amplitude ωA. From the Nyquist theorem, the highest frequency that can be represented by a sampled waveform is π or half the sampling rate.

Frequencies above 1 radian per sample will increase in amplitude after delta coding, and lower frequencies will decrease. Thus, if high frequencies are absent, it should be possible in theory to reduce the amplitude to arbitrarily small values by repeated delta coding.

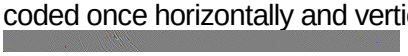
Eventually this fails because any noise (which is not compressible) in the prediction is added to noise in the sample with each pass. (Noise has a uniformly distributed spectrum, so its high frequency components are amplified by delta coding). Noise can come either from the original data or from quantization (rounding) errors during sampling. These are opposing sources. Decreasing the number of quantization levels removes noise from the original data but adds quantization noise.

The images below show the effects of 3 passes of delta coding horizontally and vertically of the image lena.bmp (a widely used benchmark image). The original image is in BMP format, which consists of a 54 byte header and a 512 by 512 array of pixels, scanned in rows starting at the bottom left. Each pixel is 3 bytes with the numbers 0..255 representing the brightness of the blue, green, and red components. The image is delta coded by subtracting the pixel value to the left of the same color, and again on the result by subtracting the pixel value below. (The order of the two encodings does not matter). To show the effects better, 128 is added to all pixel values (which does not affect compression). Thus, a pixel equal to its neighbors appears medium gray.

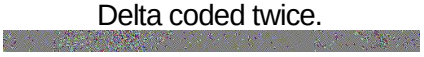
Original image



Delta coded once horizontally and vertically.



Delta coded twice.



Delta coded 3 times.

The original image is 786,486 bytes (with or without delta coding). The following table shows the compressed sizes when compressed with an order 0 indirect context model (ICM-0), with each of the 3 colors compressed in a separate stream.

	ICM-0
lena.bmp	569,299
delta 1	511,316
delta 2	645,634
delta 3	768,154

For comparison, Image Magick compresses to PNG format with size 474,573, and the top ranked paq8px_v67 -6 to 412,641 bytes.

Details: The ICM-0 model was implemented in ZPAQ 1.10 using the following configuration:

```
comp 0 0 0 0 1
  0 icm 7  (indirect context model using 2^(7+6) bytes)
hcomp
  b++ a=b a== 3 if  (b is 0,1,2 depending on color)
    a=0 b=0
  endif
  a<<= 9 *d=a halt  (context is color in bits 10..9 + order 0)
post
  0
end
```

5.6.2. Color Transform

lena.bmp can be compressed to 499,139 bytes by performing the color transform (red, green, blue) to (red-green/4, green, blue-green*3/4), then delta coding and modeling with ICM-0. The transform was tuned to this image, but is not optimal for all images. For many others, the transform (red-green, green, blue-green) works well. The transform works because when one color is brighter, the others tend to be too. Thus, one color can predict the others. The ideal transform depend on the average color of the image.

5.6.3. Linear Filtering

A linear filter is a finite impulse response filter with n taps that predicts a sample x_i in the sequence $x_1 x_2 ...$ as follows:

$$\text{prediction} = \Sigma_{j=1..n} w_j x_{i-j}$$

where w_j is called the j'th coefficient. A delta filter is the special case of the coefficient array n = 1, **w** = (1). Two

passes of a delta filter is equivalent to n = 2, **w** = (2, -1), and 3 passes is equivalent to n = 3, **w** = (3, -3, 1).

An adaptive filter is a linear filter whose coefficients are adjusted to reduce prediction errors. A simple update rule is:

$$w_j := w_j + \lambda x_{j-i}(x_i - \text{prediction}), j = 1..n$$

where λ is the learning rate. The update rule is unstable because of a positive feedback loop: when the error is large, it can lead to large updates which could increase the error even more. An adaptive filter must compensate by limiting the magnitudes of the weights and updates.

5.7. Specialized Transforms

It is often possible to find transforms that improve compression for specialized data types. We mention two.

5.7.1. E8E9

The E8E9 transform is used to compress x86 executable code (EXE or DLL files). In x86, a CALL or JMP instruction (E8 or E9 hex) is followed by a 4 byte address (LSB first) relative to the program counter. Compression can be improved by converting to an absolute address, because code often contains many calls or jumps to the same address. The transform consists of searching for a byte with the value E8 or E9 hex (232 or 233), interpreting the next 4 bytes as a 32 bit number, and adding the offset from the beginning of the input file. The decompresser does the same, except that it subtracts the offset. E8E9 is used in CAB format (for CALL instructions only) and in many top end compressors. Recent versions of PAQ8PX by Jan Ondrus also transform conditional branch addresses.

In x86-64, all references to static memory (not just JMP and CALL) are relative addresses. Currently transforms for x86-64 are not yet widely used.

5.7.2. Precomp

Precomp is a program by Christian Schnaader that searches its input for segments of data compressed in deflate (zip) format and uncompresses them. This can improve compression if the (now larger) data is compressed with a better algorithm. Many applications and formats use deflate compression internally, including PDF, PNG, JAR (Java archive), ODT (OpenOffice) and SWF (Shockwave Flash).

To make the inverse transform bitwise identical, precomp tests by recompressing the data with zlib and comparing it. Recall that LZ77 is not a bijection. There are many different ways to compress a string that will decompress the same way. Precomp relies on the fact that most applications use zlib rather than write their own implementation. Still, precomp must test 81 combinations of options to find one that compresses to exactly the original data, and then stores those options in the output. If it fails to find a match (even in valid deflate data), then it must insert additional data.

Precomp can be used by itself. It is also built into two compressors, lprepaq (precomp+lpaq6) and paq8o8pre (precomp+paq8o8). paq8o8pre -7 compresses flashmx.pdf from the Maximum Compression corpus to 1,821,939 bytes in 692 seconds. As of Feb. 2010 the program has not yet been benchmarked and the best result is 3,549,197 bytes by WinRK 3.1.2. The improvement is obtained partly by unzipping many embedded BMP images and compressing them with paq8o8's specialized BMP model (which is also top ranked on rafale.bmp).

5.8. Huffman coding

The open source file compressor M1x2 v0.6 by Christopher Mattern in Feb. 2010 uses order 1 Huffman coding as a preprocessor to a context mixing model. The idea is to reduce the size of the input to make compression faster. The model contexts are aligned on Huffman code boundaries instead of byte boundaries. The order 1 coder is actually 256 tables selected by the previous byte. Huffman codes are limited to 12 bits in length to simplify the implementation.

6. Lossy Compression

Lossy compression refers to discarding unimportant information. Generally this means compressing images, video, or audio by discarding data that the human perceptual system cannot see or hear.

Lossy compression is a hard AI problem. To illustrate, speech could theoretically be compressed by transcribing it into text and compressing it with standard techniques to about 10 bits per second. We are nowhere near that.

Even worse, we could imagine a lossy video compressor translating a movie into a script, and the decompresser reading the script and creating a new movie with different details but close enough so that the average person watching both movies one after the other would not notice any differences. We may use a result by Landauer (1986) to estimate just how tiny this script could be. He tested people's memory (over a period of days) over a wide range of formats such as words, numbers, pictures and music, and concluded that the human brain writes to long term memory at a fairly constant rate of about 2 bits per second. Currently we need 10^7 bits per second to store DVD quality MPEG-2 video.

The state of the art is to apply lossy compression only at a very low level of human sensory modeling, where the model is well understood.

6.1. Image Compression

All image formats, even BMP, may be regarded as a form of lossy image compression. An uncompressed image is normally a 2 dimensional array of pixels, where each pixel has 3 color components (red, green, blue) represented as an integer with a fixed range and resolution. A pixel array is an approximation of a 2 dimensional continuous field where the light intensity at any point would be properly described as a continuous spectrum. Note how lossy compression is applied:

- The eye can't see detail much smaller than 0.1 mm, so there is no need for an image to have more than a few thousand pixels in each dimension.
- The eye can't detect differences in brightness of less than about 1%, so there is no need to quantize brightness to more than a few hundred levels.
- The eye has 3 types of cones sensitive to red, green, and blue. Combinations of these colors can reproduce every color that the eye can see. There is no need to distinguish pure spectral yellow emitted by a rainbow from the apparent yellow from a monitor produced from a mixture of red and green light, even though there are instruments such as a spectrograph that can make such distinctions.
- The eye detects brightness on a logarithmic scale, so there is no need to use more bits to represent brighter lights. Sunlight is 1000 times brighter than room light, but doesn't look like it.

6.1.1. BMP

A BMP image uses 8 bits per pixel per color, which matches the resolution of most monitors. Each value is an 8 bit number ranging from 0 (darkest) to 255 (brightest). The values are proportional to apparent light intensity, not actual intensity. The actual intensity is gamma corrected by the monitor by raising it to the power of γ = 2.2. Thus, a pixel value of 200 is a little over 4 times as bright as a pixel value of 100, although it only looks twice as bright.

6.1.2. GIF

The GIF image format is lossless except that it uses a color palette of up to 256 colors. The format is an array of 8 bit indexes into the palette. The limited number of colors noticeably reduces the quality of color photographs, although it is sufficient for grayscale or diagrams. A GIF file may contain multiple images for animations.

GIF uses LZW compression (section 5.3.1) with a maximum dictionary size of 4K. When the table is full, it is discarded and re-initialized. It reserves two codes to initialize the table and to mark the end of data.

Use of GIF was discouraged due to a patent on LZW, which is now expired.

6.1.3. PNG

PNG is a lossless image format. Images are normally 8 bits per pixel but can be more. A pixel has 3 color components and an optional fourth component for an alpha channel to indicate transparency.

PNG is compressed by predictive filtering (section 5.6) followed by deflate (section 5.2.2). There are 5 filters which can be selected for each scan line. The image is scanned left

to right starting at the top. Let A, B, and C be the previously coded neighboring pixels of the predicted pixel x:

```
C B
A x
```

The 5 possible predicted values are 0, A, B, (A+B)/2, or Paeth. The Paeth filter is to predict A, B, or C, whichever is closest to A + B - C. The Paeth filter usually gives the best compression.

6.1.4. TIFF

TIFF is an image container format. Most commonly it is used for uncompressed images when BMP cannot be used because more than 8 bits per pixel or more than 3 color components are needed. TIFF supports several lossless compression modes. The most common is delta coding (subtracting the pixel to the left) followed by LZW.

6.1.5. JPEG

JPEG is the most widely used representation for photographic images. It uses lossy compression. It exploits two limitation of human visual perception. First, the eye has different degrees of sensitivity to brightness variation depending on spatial frequency, peaking at 0.1 to 0.2 degrees (a few pixels). Second, the eye is much less sensitive to color variation at high spatial frequencies. The compression steps for baseline JPEG are as follows:

- Color transform from RGB to YCbCr.
- Downsampling the two chroma components Cb and Cr.
- 8 by 8 discrete cosine transform (DCT).
- Variable quantization depending on color and spatial frequency.
- Delta coding the DC coefficient.
- Reordering the coefficients in zigzag order from low to high frequency.
- Huffman coding with run length encoding of zeros.

The color transform from RGB (red, green, blue) to YCbCr is:

Y = 0.299 R + 0.587 G + 0.114 B (black-white)
Cb = 128 - 0.168736 R - 0.331264 G + 0.5 B (yellow-blue)
Cr = 128 + 0.5 R - 0.418688 G - 0.081312 B (green-red)

The eye is less sensitive to fine detail in Cb and Cr than in Y, so these two are (optionally) downsampled 2 to 1 by averaging 2 by 2 blocks of pixels into 1 pixel. The inverse transform is:

R = Y + 1.402(Cr - 128)
G = Y - 0.34414(Cb - 128) - 0.71414(Cr - 128)
B = Y + 1.772(Cb - 128)

The DCT represents 8 by 8 blocks of pixels in the spatial frequency domain. The 64 DCT coefficients S_{uv}, u, v in 0..7, of the 8 by 8 pixel block S_{xy}, x, y in 0..7, are computed as follows:

$$S_{uv} = \alpha(u)\alpha(v) \Sigma_{x=0..7}\Sigma_{y=0..7} S_{xy} \cos[\pi/8 (x+1/2) u] \cos[\pi/8 (y+1/2) v]$$

where $\alpha(0) = 1/8^{1/2}$ and $\alpha(1..7) = 1/4$ are normalizing factors. u is the horizontal spatial frequency and v is the vertical spatial frequency. The image below shows the contribution of each of the 64 S_{uv} DCT coefficients to an 8 by 8 pixel block with u reading across and v reading down. The S_{00} coefficient is at the top left.

Left: 8 by 8 DCT (from Wikipedia). Right: zig-zag encoding order of the S_{uv} coefficients (from Wikipedia).

The inverse DCT has the same form:

$$S_{xy} = \alpha(x)\alpha(y) \Sigma_{u=0..7}\Sigma_{v=0..7} S_{uv} \cos[\pi/8 (u+1/2) x] \cos[\pi/8 (v+1/2) y]$$

Each of the 64 coefficients in Y and the 64 in Cb and Cr are quantized by dividing by one of 128 values from two quantization tables and rounding. Because the eye is less sensitive to high spatial frequencies (u and v large), especially in the two chroma components, these divisors can be larger.

The coefficient S_{00} is the average brightness of the 8 by 8 block. It is called the "DC" coefficient. It is the only one that depends significantly on neighboring blocks, so it is delta coded by subtracting the DC coefficient of the last coded block of the same color. The other 63 coefficients are called "AC".

For most images, the high spatial frequencies will be small except in regions with fine detail. Therefore the coefficients are reordered in zigzag order by increasing u+v, resulting in the largest coefficients appearing first.

The coefficients are grouped into runs of R zeros followed by one nonzero value, where R ranges from 0 to 15. The nonzero coefficient is a 12 bit signed number, but is usually near 0. It is coded as an S bit signed number, where S ranges from 1 to 12, followed by S extra bits. For example, the sequence 0,0,0,0,0,3 would be coded as R=5,S=2,11. The RS code (52) would be Huffman coded, and the two bits "11" (binary 3) would follow uncompressed. Negative numbers are coded by subtracting 1 and sending the same number of bits. For example, -3 would be coded as "00", which are the last 2 bits of -4. After the last nonzero coefficient, a RS code of 00 marks the end of block.

There are 4 Huffman tables for the RS codes, one each for DC-Y, DC-color, AC-Y, and AC-color. The tables are transmitted by sending 16 lists of RS codes (1 byte each) having code lengths of 1 through 16. Each list is preceded by one byte to indicate the length of the list. Other data such as the quantization tables are sent uncompressed. Huffman codes are packed in MSB to LSB order.

JPEG supports inserting restart codes into the Huffman coded data to mark the start of independently compressed image slices for error recovery. There is also an end of image symbol. The Huffman code is designed so that symbols can be found without decoding. Symbols are marked with a starting FF byte (11111111 binary). No symbol is assigned a Huffman code of all 1 bits. Also, if a byte of all 1 bits is coded, then it is followed by a 0 byte which the decoder skips.

The JPEG specification describes several modes in addition to baseline, described above. About 95% of images are baseline JPEG. The rest are mostly progressive mode. In this mode, a coarse approximation of the image is sent first so that the receiver can start displaying it before the rest of it is received. Progressive mode uses two techniques to do this. One is spectral selection, in which the low frequency DCT coefficients are sent first. The other is progressive approximation, in which the high order bits of the coefficients are sent first. Usually both techniques are combined.

JPEG allows up to 4 colors (for an alpha channel) and 12 bits per pixel. These modes are rare, but are supported by the IJG reference implementation. Images may also be grayscale by dropping the Cb and Cr components.

The JPEG standard also describes arithmetic coding as an alternative to Huffman coding, and a lossless hierarchical mode in which successively higher resolution images are sent as differences from the previous one. Neither of these two were implemented by IJG or any subsequent software because the methods were patented at the time.

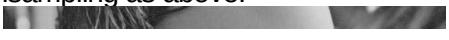
In 2002, Forgent claimed U.S. patent 4,698,672 on JPEG, specifically the invention of using a single code to represent a run length followed by a second value, which is used in the RS codes. By April 2004, Forgent announced that it had collected US$90 million from 30 companies and filed patent infringement suits against 31 others. In May 2006 the USPTO ruled that the claims of the patent related to JPEG were invalid due to prior art. The patent expired 5 months later.

The images below show the effects of the color transform and DCT. The first image was created using IJG's public domain software `cjpeg` to convert lena.bmp (786,486 bytes) to JPEG (23,465 bytes). This is 17.5 times smaller than the best result obtained with lossless compression.

Left: lena.bmp, 786,486 bytes. Right: JPEG created with `cjpeg -quality 50 -optimize`, 23,465 bytes.

The -quality setting sets the quantization tables. These range from 16 for the Y-DC coefficient to 99 for high frequency coefficients. The -optimize parameter creates the best possible Huffman tables.

The images below show a JPEG image separated into its three color components by setting all of the other coefficients to 0. All images below are high quality (-quality 100) without chroma downsampling as above.



Y coefficients only.

Left: Cb only. Right: Cr only.

The images below separate some of the different frequency coefficients.

S_{00} (DC)

First two AC coefficients. Left: S_{01}. Right: S_{10}. Note that these are *color* images.

Left: S_{20}. Right: S_{11}.

The following image illustrates the eye's insensitivity to fine detail in Cb and Cr. All of the 63 AC coefficients in Cb and Cr are set to 0, and yet the effect is barely noticeable. Compare with S_{00} above when the Y AC coefficients are also removed.

All 63 AC chroma coefficients set to 0.

6.1.6. JPEG Recompression

Even though JPEG is already compressed, there are lossless algorithms that can improve the compression by 20% to 30% and decompress to the original JPEG image.

6.1.6.1. Stuffit

On Jan. 7, 2005, Darryl Lovato of Allume (now Smith Micro) announced that their Stuffit archiver would compress JPEG files by 30%. Until then, it was widely believed that JPEG files could not be compressed further because they were already compressed. Indeed, most compression algorithms will not compress JPEG files more than about 1 to 2%. On the ACT benchmark by Jeff Gilchrist, an early version compressed 3 test images by 25% to 30%. The Maximum Compression JPEG benchmark gives similar results for newer versions.

According to U.S. Patent 7,502,514, filed Jan. 4, 2005, the compressor uses a transform that partially decompresses the JPEG image back to the quantized DCT coefficients, undoing all of the lossless steps. These values are then compressed with a proprietary model. Some details are described in the patent. There are separate models for S_{00} (the DC coefficient), S_{0v}, S_{u0} (DC in one dimension), and S_{uv} for u, v > 0. The DC coefficient is modeled with a predictive filter. The decompresser decompresses the coefficients and inverts the transform by repeating the normal lossless JPEG compression steps to restore the original image. JPEG encoding from the DCT coefficients onward is deterministic, so the result is bitwise identical. The patent was issued to Lovato and Yaakov Gringeler. Gringeler was the deleloper of the (now unsupported) Compressia archiver.

6.1.6.2. PAQ

PAQ versions beginning with PAQ7 in Dec. 2005 also include a JPEG model for baseline images (but not the less common progressive mode). Both PAQ and Stuffit compress to about the same size, but the PAQ compressor is much slower. PAQ is open source, so its algorithm can be described in better detail. It uses a context model instead of a transform. The context model decodes the image back to the DCT coefficients like the Stuffit algorithm, but instead this information is used as context to predict the Huffman coded data for both compression and decompression without a transform. The PAQ algorithm is not patented.

The PAQ model has evolved over time but all versions work basically the same way. In paq8px_v67 released in Nov. 2009, the Huffman codes are predicted using 28 indirect context models. These are mixed with 3 mixers. The output of those 3 are adaptively mixed and passed through 2 SSE stages and then arithmetic encoded. All contexts are computed on Huffman code boundaries and combined with earlier bits in the same Huffman code.

The most important context is the combination of the component (Y, Cb, Cr) and the u,v coordinates (horizontal and vertical spatial frequency) to one of 192 values. JPEG uses effectively only 4 values because it uses the same Huffman tables for Cb and Cr, and the same tables for all 63 AC components. The PAQ model distinguishes them. Coefficients with large u and v tend to be smaller. Note that what is being modeled is a Huffman code representing a run of zeros along a zig-zag path followed by a nonzero coefficient with extra uncompressed bits appended.

The second most important context is a prediction based on the partial DCT in one dimension of the 8 pixels along each of the two adjacent edges in the neighboring 8 by 8 blocks to the left and above. This model was added by Jan Ondrus beginning in July 2007 (starting with paq8fthis). The model reflects the constraint that neighboring pixels in adjacent blocks should have similar values. More precisely, the 1 dimensional vertical DCT of the two adjacent columns of 8 pixels in two horizontally adjacent blocks should have similar values. Likewise, the horizontal DCT of adjacent rows of pixels in vertically adjacent blocks should be similar. A partial DCT is computed using one of the two summations in the DCT equation above, for example in the horizontal direction:

$$S_u = \alpha(u) \sum_{x=0..7} S_x \cos[\pi/8\ (x+1/2)\ u],$$
where $\alpha(0) = 1/8^{1/2}$, $\alpha(u) = 1/4$, $u > 0$.

At the beginning of a block, the context is computed for the two neighboring blocks. For the block to the left, the DCT is inverted in the horizontal direction producing 8 columns each containing a DCT of the 8 pixels in each column. The rightmost column coefficients $C_{70}..C_{77}$ provide the context. For each coefficient S_{uv} being predicted, only the coefficient C_{7v} of the neighboring block is used as context. Furthermore, the context C_{7v} is updated by computing the partial inverse DCT of the current block in the horizontal direction and incrementally subtracting from C_{7v} the coefficient of the leftmost column, S_{0v}. Thus, $C_{70}..C_{77}$ represents at all times the difference between the vertical DCT of the two neighboring columns of pixels. For a smooth image, these coefficients will all tend toward small values as the higher frequency coefficients of the current block are coded. A similar calculation is done on the top row of the current block and the bottom row of the block above.

There are many other contexts which have a smaller effect on compression. These include preceding coefficients or their RS or Huffman codes within the same block, or corresponding coefficients in neighboring blocks or in earlier components (colors) in the same block. All of these contexts may be rounded or combined to provide additional contexts. The two most important contexts are also used as contexts for the mixers and SSE stages.

6.1.6.3. WinZIP

WinZIP added a JPEG compression algorithm to version 12.0 in Sept. 2008. The format is specified precisely to allow third party developers to decompress WinZIP compressed JPEG files. The algorithm is very fast, but does not compress as well as PAQ or Stuffit. On the Maximum Compression JPEG test, it is ranked third at about 18% compression, vs. 24% for PAQ and Stuffit. The algorithm handles the most common formats: baseline, progressive, and extended sequential with 8 or 12 bits per pixel and 1 to 4 component colors.

WinZIP uses a transform to partially decode a JPEG image back to the DCT coefficients. The Y, Cb, and Cr components are compressed in separate streams. Within each 8 by 8 DCT block, the coefficients are coded in reverse zigzag order from highest spatial frequency (S_{77}) to DC (S_{00}) starting with the first nonzero coefficient. The DC value is predicted using a more sophisticated model and replaced with a prediction error. The coefficients are then encoded using a variable length code similar to an exponential Golomb code. The numbers are coded in the order: range, magnitude, and sign, each modeled with different contexts, with special cases for magnitudes of 0 and 1. Contexts depend on u, v, and other coefficients in the same block and the blocks to the left and above the current block. Only one context is computed for each bit. Then the context is passed to a binary arithmetic coder described in U. S. patent 4,791,403 by IBM (now expired). The context model is tightly integrated into the coder. It is essentially a direct context model with a fixed learning rate. The encoder is modified to support encoding extra bits without compression with a fixed probability of 1/2.

The WinZIP algorithm describes 3 functions used to compute contexts:

- **sum(S_{uv})** or sum(u,v) predicts the magnitude of S_{uv} based on the magnitude of higher frequency coefficients in the same block. sum(u,v) is the sum of the quantized absolute values of all previously coded coefficients $S_{u..7,v..7}$

in the lower right corner of the coefficient array. For example sum(5,6) = $|S_{57}/Q_{57}| = |S_{66}/Q_{66}| = |S_{67}/Q_{67}| = |S_{76}/Q_{75}| = |S_{77}/Q_{77}|$ where Q_{uv} is the quantization scale factor of S_{uv}. Coefficients are represented in the JPEG as integers S_{uv}/Q_{uv}.

- **avg(u,v)** predicts the magnitude of S_{uv} based on the neighboring blocks L to the left of S and U up. avg(u,v) is the average of $|U_{u,v}|$, $|U_{u,v-1}|$, $|U_{u-1,v}|$, $|U_{u-1,v-1}|$, and $4|L_{uv}|$. The context only applies where u > 0 and v > 0. On the top and left edges of the image, nonexistent U_{uv} or L_{uv} are omitted from the average.
- **bdr(u,v)** is a predictor for coefficients S_{u0} and S_{0v} for AC coefficients where avg(u,v) does not apply. It uses the general principle that L_{uv} predicts S_{uv} when u is even and $-S_{uv}$ when u is odd. Likewise, U_{uv} predicts S_{uv} when v is even and $-S_{uv}$ when v is odd. The prediction is weaker at higher frequencies so it is used only for u = 0 or v = 0. The predictor is:

$bdr(0,v) = L_{0v} - L_{1v} - S_{1v}$.
$bdr(u,0) = U_{u0} - U_{u1} - S_{u1}$.

Magnitudes uses as contexts are typically scaled logarithmically and quantized to n discrete levels using the function **cat(x, n)** = min(n-1, ceil($\log_2$(x+1))).

Coefficients are represented using a variable length code of the form (zero, pivot, range, magnitude, sign) with each part coded using different contexts. The parts have the following meaning:

- zero = 0 to encode 0, in which case the other parts are omitted.
- pivot = 0 to encode -1 or 1, in which case only a sign bit follows.
- range = 0, 10, 110, 1110,... = a unary encoding of a range of size 2, 4, 8, 16...
- magnitude = n+1 bits, where n is the unary number encoded by the range.
- sign = 0 if positive, 1 if negative.

For example:

Number	zero	pivot	range	magnitude	sign
-2	1	1	0	1	1
-1	1	0			1
0	0				
1	1	0			0
2	1	1	0	0	0
3	1	1	0	1	0
4	1	1	10	00	0
5	1	1	10	01	0
6	1	1	10	10	0
7	1	1	10	11	0
8	1	1	110	000	0
9	1	1	110	001	0

For each block, the position of the first nonzero coefficient is coded as an ordinary 6 bit number. The context is the previous 0 to 5 bits and cat(sum(U_{00}) + sum(L_{00}), 13), i. e. the sum of the absolute values of the 126 adjacent AC coefficients of the two neighboring blocks. For the AC coefficients S_{uv}, the contexts are as follows:

- For the zero bit: u, v, cat(sum(u,v), 6) and either cat(bdr(u,v), 3) if u = 0 or v = 0, or cat(avg(u,v), 3) otherwise.
- For the pivot bit: as above with 7 and 5 levels instead of 6 and 3 respectively.
- For the range: the bit position, whether u = 0, whether v = 0, cat(sum(u,v), 9) and either cat(bdr(u,v), 9) or (cat(avg(u,v), 9).
- For the magnitude: whether u = 0, whether v = 0, and cat(zigzag(u,v)-4, 6), where zigzag(u,v) means the position of S_{uv} in zigzag order from 0 to 63.
- For the sign bit: a prediction is computed as +, -, or 0. If 0, then the sign bit is coded with fixed probability 1/2. Otherwise the prediction, u and v are the context. A prediction is made only for u, v < 2. For u = 0 or v = 0, the prediction is the sign of bdr(u,v). For u = 1, the prediction is the sign of U_{1v}. For v = 1, the prediction is the sign of L_{u1}. For S_{11} the prediction is the sign of both U_{11} and L_{11} if they are the same, or 0 if they differ.

The DC coefficient is predicted by the weighted average of predictions p_U from the block above and p_L from the block to the left.

$$\text{prediction} = (w_U p_U + w_L p_L) / (w_U + w_L).$$

The prediction is estimated from the neighboring block and the gradient given by the first AC coefficient of both blocks.

$$p_U = U_{00} - 2.2076(U_{01} - S_{01})$$
$$p_L = L_{00} - 2.2076(L_{10} - S_{10})$$

The prediction weights account for the smoothness of the image in the horizontal and vertical directions. This is indicated by the absence of high frequency coefficients in the predicted direction.

$$w_U = 2^{-|\Sigma_{i=1..7}(|U_{i0}| - |S_{i0}|)|}$$
$$w_L = 2^{-|\Sigma_{i=1..7}(|L_{0i}| - |S_{0i}|)|}$$

The DC coefficient prediction error is coded as above. The context for the zero, pivot, range, and magnitude is cat(sum(0,0), 13). The context for the sign is the 3 bits from the signs of the two neighboring DC signs and the sign of the prediction.

Note that contexts do not include the previously coded bits of the current number. This context is implicit in the bit position for the zero, pivot, range, and sign, but not in the magnitude. This does not hurt compression too much because the magnitude bits tend to be random.

The context and bits are fed to the IBM binary arithmetic coder. The coder has a tightly integrated context model which is essentially a direct context model with a fixed learning rate. The coder is designed to avoid multiplication and division operations by representing the range in the logarithmic domain and using addition and subtraction instead. It uses lookup tables to convert to and from the linear domain when addition or subtraction is required. In 1987 when the coder was designed, table lookup was faster than multiplication, although this is no longer true on modern processors.

The coder maintains a range (low, low+R). It receives as input a bit to be coded, y, a prediction for the most probable symbol, MPS=0 or MPS=1, and a prediction, p, for the MPS between 1/2 and 1. It always codes the MPS in the lower part of the range. Thus, most of the updates have the form (low, R) := (low, pR), which is a simple addition in the log domain. In the less likely case that y ≠ MPS, then the update is (low, R) := (low+(1-p)R, (1-p)R), which requires a table lookup to convert low to the linear domain and back. Some care must be taken with the tables so that the two symbol probabilities don't add up to more than 1 due to rounding.

The coder represents the range and probabilities as 16 bit fixed point integers with 10 bit fractional parts in the linear domain and 12 bit fractional parts in the log domain. It outputs and normalizes one byte at a time. This also requires a table lookup to convert to the linear domain.

The context model quantizes probabilities into 48 discrete values between 1/2 and 1. This results in a coding inefficiency of less than 0.2% for most of the probability range.

Each entry in the context table contains a prediction, p, for the MPS, the choice of MPS, a counter k, and a threshold, LRM. The counter and threshold are used to decide when it is time to update the prediction. When the MPS or LPS (least probable symbol) is coded, the prediction should be increased or decreased respectively, but increasing or decreasing p by one step every time would be too fast.

Initially k is 0 and LRM is set to a fraction of the current range, R, depending on a table lookup of the current value of p. Each time an MPS is coded, the range gets smaller. When the range reaches LRM, p is increased and LRM is reset again as a fraction of R depending on p. The amount of the increase depends on k, the LPS count. If k is 5 or more, then p is unchanged. If k is in the range 2 to 4, then p is increased by 1 in the log domain, which has the effect of reducing the LPS probability (1-p) by about 10% to 15%. If k is 1 then (1-p) is halved. If k is 0 then (1-p) is halved again.

When an LPS is coded, p should be decreased, but again not by a whole step. If k is less than 5 then it is incremented and p is unchanged. Otherwise the fraction of the distance moved by R toward LRM is used as an implicit count of MPS symbols to determine the update. If R has moved less than 1/4 of the way to LRM then p is decremented once in the log domain, or about 10% to 15% in the linear domain. If R has moved between 1/4 and 1/2 of the way toward LRM, then p is decremented twice. If R has moved between 1/2 and 3/4 then p is halved in the linear domain (decreased by about 6 to 10). If R has moved more than 3/4, then p is divided by 4. Then k is reset to 0 and a new LRM is chosen based on a fraction of the range depending on p. The table is designed so that for any p, the ratio of increments to decrements is 1-p to p, so that p reflects the true probability.

It should be noted that the IBM model and coder was developed for encoding binary and grayscale images. Because the context model is tightly integrated with the coder, it lacks the flexibility required to implement context mixing algorithms or to fine tune the learning rate for stationary or adaptive models. Its design goals were for

speed, not maximum compression, at a time when computation was expensive.

6.2. Video Compression

Video approximates continuously moving images by using a sequence of still images, called frames. The neural circuitry of the human visual system has a delayed response to light on the order of tens of milliseconds. Thus, a frame rate of at least 24 to 30 per second produces a sensation that is nearly indistinguishable from continuous motion. However, simply flashing images at this rate would produce a noticeable flicker. The eye can detect flicker at rates of up to about 75 flashes per second. Sensitivity to flicker increases in bright light, toward the blue end of the spectrum, and in peripheral vision away from the fovea where visual acuity is sharpest. Thus, a computer monitor viewed up close requires a higher refresh rate than a television viewed from a distance. Movie theatres display 24 frames per second and remove flicker by flashing each frame on the screen two to four times.

Video frames do not have to have as much resolution as still images. The eye moves in saccades, jumping from one part of the image to another at a rate of 30 to 70 times per minute. In still images, the eye is attracted to regions of high contrast such as edges or corners, and to areas of interest such as faces. In text, the eye jumps from word to word. This requires all of the image to be displayed in fine detail. In video, there is not enough time to look at more than one part of a frame before the next frame is displayed. Thus, the rest of the frame can be displayed at a low resolution.

Between saccades, the eye tracks moving objects smoothly. If a frame is displayed more than once or for more than a small fraction of the frame interval, then the effect is to blur the object as the eye moves across each frame.

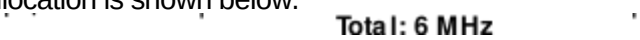
Time sampling of images can produce artifacts such as the wagon wheel effect, where a spoked wheel appears to spin slowly backward. This artifact is analogous to the Moire effect caused by spatial sampling of a repeating pattern in still images.

6.2.1. NTSC

NTSC is one of three standards for analog television, the others being PAL and SECAM, used in different parts of the world. NTSC standardized black and white television in 1941 and color TV in 1953 in North America. It was used until 2009 for over the air broadcasts in the U.S., when it was replaced by HDTV.

NTSC is displayed at 29.97 frames per second. Each frame consists of 525 horizontal scan lines starting at the top left corner of the screen. To reduce flicker, the display is interlaced: each frame is divided into two fields which alternately display the even and odd numbered lines. (PAL and SECAM use 625 scan lines at a rate of 50 fields or 25 frames per second). NTSC is an analog format, so there is no concept of a "pixel". However, the luma (brightness) signal is transmitted over a band that extends about 4.5 MHz above the carrier. This corresponds to a Nyquist sampling rate of 9 million pixels per second, equivalent to about 571 pixels per scan line.

When color TV was introduced in 1953, a chroma signal was added without increasing the bandwidth or breaking compatibility with black and white TV sets. The spectrum allocation is shown below.



NTSC frequency allocation (from Wikipedia).

The video signal is split into three color components similar to YCbCr as in JPEG. The black-white (luma) signal is unchanged. It is amplitude modulated in the same band of 4.95 MHz. The blue-yellow and red-green signals are transmitted in a smaller band with a width of 2 MHz that overlaps the luma signal. A narrower band is possible because the eye is less sensitive to high spatial frequencies in chroma (especially blue-yellow) than luma. The two signals are amplitude modulated 90 degrees out of phase, allowing them to be separated by the receiver.

Because the luma and chroma overlap, the color signal can produce black and white artifacts and vice versa. The carrier frequencies are carefully chosen so that the artifacts of successive frames cancel out, reducing their visibility.

6.2.2. MPEG

MPEG is the most widely used format for video compression. The most commonly used versions are as follows.

- MPEG-1 is the original version of the standard, published in 1993. It specifies non-interlaced video at bit rates up to 1.5 Mbits/second. All patents on the video portion of the specification have expired. MPEG-1 layer 3 audio (MP3) is still patent protected.
- MPEG-2 extends MPEG-1 to interlaced video and higher bit rates to support digital television. It is the format used for most DVD video and for HDTV. In spite of minor differences between MPEG-1 and MPEG-2, it is protected by about 600 patents by dozens of companies. Licenses are managed by the MPEG Licensing Authority (MPEG-LA).
- MPEG-4 part 10, also known as H.264 or AVC (Advanced Video Codec) compresses video to about half the size of MPEG-1 or 2 at a similar quality level. It is widely used in Youtube and Google Video. It is also patented and licensed by MPEG-LA.

Although video files can stand alone, they are more often embedded in a container format such as AVI or streamed through a Flash player. MPEG-2 defines a transport stream for over the air transmission of HDTV by encapsulating the data in 188 byte packets with error correction.

MPEG-1 and MPEG-2 use a compression algorithm similar to JPEG, but obtain additional compression by delta coding between frames with motion compensation. There are 3 types of frames, designated I (inter-frame), P (predictive), and B (bidirectional). An I-frame can be decoded by itself. A P-frame is described in terms of differences from the previous frame. A B-frame is described in terms of difference from both the previous and next frame. A typical pattern is one I-frame every 0.5 seconds, repeating the sequence IBBPBBPBBPBBPBB. To facilitate decoding, the frames are sent out of order with the B frames sent after any future frame it depends on. The decoder then reorders the frames before displaying them. The reason for having I frames every 0.5 seconds is to allow decoding to start from the middle of a video after rewinding or fast forwarding.

Motion compensation is implemented by dividing a frame into 16 by 16 macroblocks. A macroblock in a P or B frame is decoded using a motion vector which points to a same sized region of the previous (or next) decoded image with a specified offset horizontally and vertically. After motion compensation, P and B frames are encoded using a JPEG-like algorithm as with I frames.

It is up to the encoder to find good matches in adjacent frames for encoding macroblocks. The encoder calculates the differences from the decoded image, not the original, by encoding and then decoding the adjacent frame.

Frame compression is like JPEG except that it uses fixed Huffman tables (called variable length codes) and fixed quantization tables with only a scale factor transmitted. MPEG is often transmitted at a constant bit rate, which is achieved by adjusting the quantization scale factor as needed. For DVDs, the maximum bit rate is about 10 Mbits/second. The result is that scenes with lots of motion are transmitted at a lower resolution.

MPEG-4/AVC (H.264) differs from MPEG-1/2 in that it supports arithmetic coding in addition to Huffman coding. It also supports variable sized macroblocks, from 4 by 4 to 16 by 16, with motion vectors pointing to any of 16 adjacent frames (or 32 fields) in 1/4 pixel resolution. Fractional motion vectors are obtained by using a 6 tap filter to infer half pixel intensities, followed by simpler interpolation. It also uses a deblocking filter.

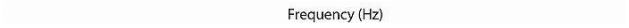
6.3. Audio Compression

Lossy audio compression uses a psychoacoustic model to determine which part of the signal can be discarded without changing the original sound. The human ear can only perceive sounds in the range 20 Hz to 20 KHz. Sensitivity peaks around 1 KHz to 5 KHz, the middle of the range of human speech. Frequency resolution is 3.6 Hz in the range 1 KHz to 2 KHz.

Like the eye, the ear perceives sound on a logarithmic scale. The range of hearing is from 0 decibels, the threshold of hearing, to 120 decibels, which is loud enough to be painful and cause hearing damage. An increment of 10

decibels (dB) represents an increase in power by a factor of 10, although we perceive it as closer to twice as loud. An increment of 20 dB represents an increase in amplitude by a factor of 10 (because power is proportional to amplitude squared).

The logarithmic scaling is partially due to the masking effect, in which a sound decreases sensitivity to other sounds at different frequencies that occur at the same time (frequency masking) or other sounds at the same frequency that occur earlier or later (temporal masking). The graph below illustrates how frequency masking affects the threshold of hearing. Temporal masking has an exponentially decaying effect, starting from about 20 milliseconds before the sound to 100 milliseconds afterward.



The effect of frequency masking on the threshold of human hearing (from Wikipedia).

Humans can perceive the direction of a sound source to an accuracy of about 3 degrees. Stereoscopic sound perception depends on two effects. First, a sound is louder in the ear closer to the source. Second, there is a time delay in reaching the further ear because sound travels at about 300 meters per second through air. Earphones can reproduce both of these effects, but stereo speakers typically do not. The sound seems to come from one speaker or the other or from some point in between. This has led to sound systems with more than 2 channels.

High frequency sounds are harder to locate when the distance between the ears is more than 1/2 wavelength because the phase shift is ambiguous and because neurons can't fire fast enough to transmit phase information. This occurs at around 1.5 KHz. This suggests an approach of transmitting stereo information (left minus right) at a lower bandwidth.

The simplest form of lossy audio compression is to filter out the high frequencies where most of the information is located. AM radio discards frequencies above 10.5 KHz. FM uses a bandwidth of 15 KHz for the mono signal (left plus right) and 13 KHz for the stereo signal.

DS0 digital telephony uses a sampling rate of 8 KHz, which requires filtering out all audio above the Nyquist rate of 4 KHz. In practice, audio above 3.3 to 3.5 KHz is filtered out. Each sample is 8 bits which is companded, or quantized on a logarithmic scale. Essentially, each 8 bit value is a floating point representation of a 14 bit integer (78 dB dynamic range) using a sign bit, 3 exponent bits and 4 mantissa bits. This 64 Kbit/second signal is sufficient to reproduce speech in spite of the fact that some sounds such as /s/ are almost entirely outside the bandwidth (4 to 8 KHz).

CD audio is stored uncompressed. It consists of two channels sampled at 44.1 KHz (22.05 maximum frequency) at 16 bits per sample (90 dB range). The bit rate is 1411.2 Kbit/second.

Lossy audio formats such as MP3, AAC, Dolby, and Ogg Vorbis are based on dividing the audio into blocks of samples, computing the modified (overlapped) discrete cosine transform (MDCT), quantizing, and transmitting the coefficients without further compression. Quantization uses the psychoacousitic model to determine the appropriate precision at each frequency. All formats support a wide range of sampling rates, compressed bit rates, and number of channels. All but Ogg Vorbis are protected by patents.

All of these formats support joint frequency encoding, a technique which compresses the stereo (left minus right) signal. Because the ear can detect intensity differences but not timing differences at high frequencies, this part of the stereo signal can be removed and replaced with information to control the overall intensity for each channel.

MP3 (MPEG-1 layer III) was the first widely used compressed format for encoding music on the Internet. Audio is divided into blocks of either 576 samples, or 192 samples to encode transients (rapid changes in the audio signal). Two channels (left and right) are supported. Good quality is achieved at a bit rate of 128 Kbits/seconds, or 9% of uncompressed CD audio. Bit rates can be variable.

AAC (Advanced Audio Codec, MPEG-2 part 7) is the default audio format used by Apple's iPod and iTunes. It is understood by most music players, phones, and video game consoles. AAC supports a greater range of bit rates and sampling rates than MP3. Block sizes are 1024 and 128 samples (or 960 and 120 depending on sampling rates). Good quality audio is about 96 Kbits/second.

Dolby Digital (AC-3, ATSC A/52) is the audio format used in DVDs and HDTV. It supports 5.1 Surround Sound. The "5.1" refers to 5 channels (left front, right front, left rear, right rear, center) and a low frequency subwoofer channel.

Ogg is a free, open source container for the Vorbis audio format. At a bit rate of 96 Kbits/second it has an audio quality slightly better than AAC and better than MP3.

The MDCT computes N frequency coefficients from 2N samples $x_0..x_{2N-1}$ in 2 adjacent blocks. The overlap is necessary to prevent artifacts where the blocks join together. The coefficients are computed:

$X_k = \Sigma_{i=0..2N-1}\ w_i\ x_i \cos[(\pi/N)(i + (N+1)/2)(k + 1/2)],\ k = 0..N-1$

The coefficients X_k represent frequencies ranging from $1/2\pi N$ up to the Nyquist rate 1/2. The inverse transform (IMDCT) has the same form:

$y_i = (w_i/N)\ \Sigma_{k=0..N-1}\ X_k \cos[(\pi/N)(i + (N+1)/2)(k + 1/2)],\ i = 0..2N-1$

The inverse transform has N inputs and 2N outputs. To complete the transform, the two overlapping sets of samples, y_i and y_{i+N}, from adjacent blocks are added together. To minimize boundary artifacts, the window function weights $w_{0..2N-1}$ are selected such that the weights at the ends (near 0 and 2N-1) go to 0 and are 1 in the middle, usually with a rounded shape. Because each weight is used twice in each block, they must satisfy $w_i^2 + w_{i+N}^2 = 1$. MP3 and AAC use the window:

$w_i = \sin[(\pi/2N)(i + 1/2)]$.

Vorbis uses:

$w_i = \sin\{(\pi/2)\ \sin^2[(\pi/2N)(i + 1/2)]\}$

Dolby AC-3 uses a Kaiser-Bessel derived window, which has a similar shape.

Conclusion

Data compression is the art of finding short descriptions for long strings. Every compression algorithm can be decomposed into zero or more transforms, a model, and a coder. Coding is a solved problem. Given a symbol with probability p, Shannon proved that the best you can do is code it using $\log_2 1/p$ bits. An arithmetic coder does this efficiently.

There is no general procedure for finding good models or prediction algorithms. It is both an art and a hard problem in artificial intelligence. There is (provably) no test to tell you if a string can be compressed or if a better model exists.

Prediction is closely related to understanding. If you understand Chinese, then you can predict a sequence of Chinese symbols. This principle can be applied to context modeling. Useful contexts are semantically independent units, for example, words in text, instructions in executable code, fields in a database, or recognizable features in an image. Different symbols that have similar meanings should be treated as if they were the same context. For example, in text, it is useful to merge upper and lower case, spaces and newlines, or related words like "someone" and "somebody" or "cold" and "wet". A context model for images would distinguish blue from green pixels but ignore fine differences. The best compressors combine the predictions of many independent models.

Preprocessing transforms are optimizations that sacrifice compression for speed and memory. Often, the output can be compressed with a simple order 0 or low order model. A transform by itself does not compress. It may hide useful contexts and add arbitrary information that makes the output ultimately larger. A good transform should minimize these effects. Thus, a BWT compressor that works on words as units would compress text better than the usual case of sorting bytes. Likewise, an LZ77 compressor that replaced duplicate strings on whole word boundaries would be preferred. Dictionary preprocessors improve both BWT and LZ77 compression by forcing those transforms to split the input on word boundaries. The best compressors on the large text benchmark use dictionary preprocessing, but I believe that is because the benchmark is tightly constrained by memory. When computers with hundreds of gigabytes become available, I believe that the top ranked programs will no longer use large dictionaries.

Lossless compressors ignore meaningless data in selecting contexts. Meaningless or random data has no predictive value and is itself not compressible. A lossy compressor not only ignores the meaningless data, but also discards it completely. Deciding which data is meaningful is a hard AI

problem that applies to both lossless and lossy compression. Both require a deep understanding of human cognitive psychology.

Coding theory says that the vast majority of strings do not have simple descriptions, so it is rather remarkable that compressible strings are so common in practice. Solomonoff, Kolmogorov, and Chaitin independently proposed that strings have a universal probability proportional to $2^{-|M|}$, where M is its shortest description, independent of the language in which M is written. Kolmogorov proved that there is no algorithm for finding such descriptions in any language. Hutter showed that the compression problem, if it were computable, would solve the general AI problem of optimizing arbitrary utility functions. In effect, he proved Occam's Razor, which is the foundation of all science: the simplest theory that explains the past is the best predictor of future events.

The prevalence of compressible strings, and thus science, depends on two facts. First, that all strings are the result of computable (or finitely describable) processes, and second, that shorter programs or descriptions are more likely than longer ones. To show the second, consider a probability distribution over the infinite set of all finite length descriptions. *Any* such distribution must favor shorter descriptions. Consider any description M having probability $p > 0$. There can be at most a finite number $(1/p)$ of more likely descriptions, and therefore an infinite number of less likely descriptions. Of the latter, there can be at most a finite number $(2^{|M|} - 1)$ that are shorter than M. Therefore there must be an infinite number of less likely descriptions that are longer than M, for all M.

The question remains whether all strings found in the real world are created by computable or finitely describable processes. This must be true for finite strings, but there are known to exist, at least in mathematics, infinite length strings such as Chaitin's constant Ω (the probability that a random program will halt) that are not computable. In fact, the vast majority of infinite length strings do not have finite length descriptions. Could there exist phenomena in the real world that have infinite length descriptions that are not compressible? For example, would it be possible to take an infinite number of measurements or observations, or to measure something with infinite precision? Do there exist infinite sources of random data?

The laws of physics say no. At one time it was believed that the universe could be infinitely large and made up of matter that was infinitely divisible. The discoveries of the expanding universe and of atoms showed otherwise. The universe has a finite age, T, about 13.7 billion years. Because information cannot travel faster than the speed of light, c, our observable universe is limited to an apparent 13.7 billion light years, although the furthest objects we can see have since moved further away. Its mass is limited by the gravitational constant, G, to a value that prevents the universe from collapsing on itself.

A complete description of the universe could therefore consist of a description of the exact positions and velocities of a finite number (about 10^{80}) of particles. But quantum mechanics limits any combination of these two quantities to discrete multiples of Planck's constant, h. Therefore the universe, and everything in it, must have a finite description length. The entropy in nats (1 nat = 1/ln(2) bits = 1.4427 bits) is given by the Bekenstein bound as 1/4 of the area of the event horizon in Planck units of area $hG/2\pi c^3$, a square of 1.616×10^{-35} meters on a side. For a sphere of radius Tc = 13.7 billion light years, the bound is 2.91×10^{122} bits.

We now make two observations. First, if the universe were divided into regions the size of bits, then each volume would be about the size of a proton or neutron. This is rather remarkable because the number is derived only from the physical constants T, c, h, and G, which are unrelated to the properties of any particles. Second, if the universe were squashed flat, it would form a sheet about one neutron thick. Occam's Razor, which the computability of physics makes true, suggests that these two observations are not coincidences.

Acknowledgements

I thank Szymon Grabowski, Aki Jäntti, Christopher Mattern, Jan Ondrus, Friedrich Regen, Steve Richfield, Sami Runsas, David A. Scott, Eugene Shelwien, Ali Imran Khan Shirani, Yan King Yin, and Bulat Ziganshin. for helpful comments on this book. Please send corrections or comments to matmahoney at yahoo dot com.

References

T. Bell, I. H. Witten. J. G. Cleary (1989), Modeling for Text Compression, ACM Computing Surveys (21)4, pp. 557-591.

C. Bloom (1998), Solving the Problems of Context Modeling.

M. Burrows, D. J. Wheeler (1994), A Block-sorting Lossless Data Compression Algorithm, Digital Systems Research Center.

G. Chaitin (1966), On the length of programs for computing finite binary sequences. Journal of the ACM, 13:547-569.

G. Cleary, W. J. Teahan (1995), Experiments on the zero frequency problem, Proc. Data Compression Conference, 480.

G. Cormack, N. Horspool (1987), Data compression using dynamic Markov modeling, Computer Journal 30:6 (December).

P. Deustch (1996), RFC 1951 - DEFLATE Compressed Data Format Specification version 1.3.

J. Duda (2007), Optimal encoding on discrete lattice with translational invariant constrains using statistical algorithms (section 3).

M. Gagliolo (2007), Universal search. Scholarpedia, 2(11):2575.

D. Huffman (1952), A Method for the Construction of Minimum-Redundancy Codes. Proc. I.R.E.: 1098-1101.

M. Hutter (2004), Universal artificial intelligence: Sequential Decisions based on algorithmic probability. Springer, Berlin.

M. Hutter et al. (2007), Algorithmic probability. Scholarpedia, 2(8):2572.

A. Kolmogorov (1965), Three approaches to the quantitative definition of information. Problems Inform. Transmission, 1, 1-7.

T. Landauer (1986), How much do people remember? Some estimates of the quantity of learned information in long term memory, Cognitive Science (10) 477-493.

S. Legg, M. Hutter (2006), A Formal Measure of Machine Intelligence, Proc. Annual machine learning conference of Belgium and The Netherlands (Benelearn-2006). Ghent.

L. A. Levin (1973), Universal sequential search problems. Problems of Information Transmission, 9(3):265--266.

L. A. Levin (1984), Randomness Conservation Inequalities: Information and Independence in Mathematical Theories. Information and Control, 61:15-37.

M. Mahoney (2000), Fast Text Compression with Neural Networks, Proc. AAAI FLAIRS, Orlando.

M. Mahoney (2002), The PAQ1 Data Compression Program

M. Mahoney (2005a), Adaptive Weighing of Context Models for Lossless Data Compression, Florida Tech. Technical Report CS-2005-16.

G. Manzini and P. Ferragina (2002) Engineering a lightweight suffix array construction algorithm (extended abstract).

J. Rissanen (1976), Generalized Kraft Inequality and Arithmetic Coding, IBM Journal of Research and Development 20(3): 198–203.

J. Schmidhuber, S. Heil (1996), Sequential Neural Text Compression, IEEE Trans. on Neural Networks 7(1): 142-146.

C. Shannon, W. Weaver (1949), The Mathematical Theory of Communication, Urbana: University of Illinois Press.

C. E. Shannon (1950), Prediction and Entropy of Printed English, Bell Sys. Tech. J. 3:50-64.

D. Shkarin (2002), PPM: one step to practicality, proc. DCC.

R. Solomonoff (1960), A Preliminary Report on a General Theory of Inductive Inference, Report V-131, Zator Co., Cambridge, Ma.

R. Solomonoff (1964), A Formal Theory of Inductive Inference, Information and Control, 7(1) 1-22, 7(2) 224-254.

A. M. Turing (1950), Computing Machinery and Intelligence, Mind, 59:433-460.

E. Ukkonen (1995), On-line construction of suffix trees, Algorithmica 14(3): 249-260.

I. H. Witten, T. C. Bell (1991), The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression, IEEE Trans. on Information Theory, 37(4): 1085-1094.

J. Ziv, A. Lempel (1977), A universal algorithm for sequential data compression, IEEE Trans. Information Theory 23(3) 337-343.

J. Ziv, A. Lempel (1978), Compression of Individual Sequences via Variable-Rate Coding, IEEE Transactions on Information Theory, 24 (5), 530-536.

Note: Website links are current as of Feb. 2010.