EXPERIMENTAL STUDY OF SHANNON-FANO, HUFFMAN, LEMPEL-ZIV-WELCH AND OTHER LOSSLESS ALGORITHMS

D. Dueck and W. Kinsner, VE4WK

Department of Electrical and Computer Engineering
university of Manitoba
Winnipeg, Manitoba, Canada R3T-2N2
Fax: (204) 275-0261
e-Mail: Kinsner@ccm.UManitoba.CA
E-mail: VE4WK@VE4KV.MB.CAN.NA

Abstract

Knowledge of the statistics of a source bit stream is required when selecting the most efficient statistical data compression techniques and designing the best codes. This paper presents a program called Statistical Analysis of Files (STAF) that analyzes and supplies the statistics on such bit streams. The two key statistics are the ***entropy*** (a measure of information content) and the ***frequency*** of ***occurrence*** table. The entropy measure is used is establishing the compression limit for statistical techniques, while the frequency of occurrence is vital in the designing of optimal variable-length codes such as Huffman and Shannon-Fano. Other statistics analyzed for optimal code compression techniques are run-length encoding, half-byte packing, and diatomic character encoding. In the entropy report, the statistical techniques are compared with a popular adaptive nonstatistical dictionary encoding algorithm, the LZW technique, to give a comparison with other methods of lossless compression on a set of benchmark files.

1. INTRODUCTION

Transmission of information requires symbols (data). If the data representation of information is compact (i.e., no redundancy is present) the information can be transferred faster than with redundant data, given the same data transfer rate, expressed in bits per second (bps). Thus, every bit of information is transferrable to others, at the right price. A Stocks and Bonds newspaper can be very expensive to send, but the price is worth it -to a stockbroker. Similarly, last night's sports scores are very important to the sports fan, but for someone who dislikes sports, the information is useless.

Information can be sent by a variety of methods, and each method has its own benefits and pitfalls. For a given probability of error, a shorter transmission would be less susceptible to error than its longer corresponding uncrunched source. Nevertheless, the shorter bit stream is more sensitive to errors, such that an error of one bit could destroy the whole information, whereas an error in the uncompressed bit stream could likely be corrected. Which then, is the best method? Clearly, the best method is the fastest, most perfect method, though which one is best for a particular type of information is the $64M question. Data compression can be employed to reduce the *Length* of the data bit stream, without losing any of the information itself. Most data is actually very repetitive (redundant), and can be "crunched" so as to remove the repetitiveness. A good elementary example of data compression is secretarial short-hand. People who know the code can transfer information quickly from a spoken format into a written format. Then at a more convenient time, the code can be expanded into a normal textual format. Elimination of repetitiveness, or redundancy, then is an improvement to information transmission.

Information exchange fills each of our days. If it can be improved in a manner that does not hinder our understanding, it should be done. Information can be represented in a variety of ways, such as pictures, speech, books, television, and computer files. All of these formats, (and many others) require a transfer from a source, to a destination. Thus, the transfer is what can be improved, and just about everyone can appropriate it.

In selecting or designing proper statistical data compression methods and techniques, the knowledge of the statistical properties of the data is essential A thorough review of exact and lossy data compression methods and techniques was recently presented by Kinsner [Kins91a], [Kins91b], [Kins91c]. The exact techniques compress and reconstruct source bit stream without any losses, and are required for transfer of critical data such as programs and financial files. The best techniques in this category require knowledge about the statistical properties of the bit stream to be compressed. The key statistical property is the frequency of occurrence of each pattern in the stream.

A program designed to analyze any bit stream and provide the necessary statistical information for code design has recently been described [DuKi91]. The Statistical Analysis of Files (STAF) program provides the user with a portrait of a file's statistics. Then, on the basis of the report, the user can then decide which type of data compression would be the most suitable for their particular piece of data. The STAF program generates two types of reports: a short and a full-length report. The short report is a one or two page report containing the following two segments: (i) the entropy analysis, and (ii) the sorted frequency of occurrence. The full report gives the following results: (i) entropy analysis, (ii) a full character frequency report, (iii) the half-byte, (iv) run-length encoding, and (v) diatomic character analyses.

3. DESCRIPTION OF PROGRAM MODULES

The STAF computer program has five modules. The two types of reports (short and full) are similar in their scope: The short report gives the entropy analysis and the sorted character frequency chart, while the long report gives additional half-byte analysis, repeated character string analysis, diatomic analysis, and the critical ASCII frequency of occurrence; chart.

3.1 Entropy Report Module

The entropy analysis module is divided into an uncompressed and compressed section. The ***uncompressed analysis*** is the most basic part of the program. Firstly,, it gives the count of the total number of characters in the file, which is also converted into bits. Next the number of distinct characters in the file is printed. Finally, the source, entropy, H_s, is calculated [Kins91c]. The ***compressed*** analysis gives six important statistics useful for determining whether a statistical compression method is feasible. The statistics are described next.

Theoretical Statistical Compression

Let us consider a source containing 200 characters: 100 *Es,* 50 each of T and A. The entropy is calculated as

$$H_s = -\sum_{i=1}^{m} p_i \log_2 p_i \qquad (3.1)$$

$$= -(0.50\log_2 0.50 + 2 \times 0.25\log_2 0.25)$$

$$= 1.5 \text{ bits/character}$$

where p_i is the probability of occurrence of each symbol in the source, and m is the number of symbols in the source text or alphabet (hereafter referred to as source).

The ultimate theoretical statistical compression (TSC) is given as the barrier which every statistical code seeks to equal, but is rarely, if ever achieved. The ultimate TSC percentage, U_p, is calculated from

$$U_p = \frac{S_a - H_a}{S_a} \times 100 \qquad (3.2)$$
$$= \frac{8 - 1.5}{8}$$
$$= 81.25\ \%$$

where S_a is the number of bits used to represent a character (normally eight, according to ASCII convention), and Ha is the source code entropy. Notice for this example we have a perfect code, one that is statistically as concise as possible. But in normal practice, a three symbol code is next to useless, and a larger and undoubtedly less perfect code would be created with a larger symbol set.

Theoretical Variable Length Codewords

While the theoretical statistical compression, U, is the standard to measure up to, the theoretical variable-length codes entropy (V') is the estimate of the entropy when variable length coding is used. While it would be necessary to actually construct the Shannon-Fano or Huffman codes to actually know the number of bits needed to encode each symbol, the *estimated* number of bits, λ_i, required to encode a symbol with probability Of p_i is

$$\lambda_i = \lceil \log_2 p_i \rceil \qquad (3.3)$$

The theoretical variable length *code* entropy, VH, *can be* calculated from

$$V_H = -\sum_{i=1}^{m} p_i \lambda_i \qquad (3.4)$$

In practice, the actual S-F or Huffman code entropy would approach, but never be smaller than U_H, such that

$$U_H \leq \text{S-F}_{Hc} \text{ or } \text{Huff}_{Hc} \leq V_H \qquad (2.8)$$

is always true.

Tk Shannon-Fano Compression Technique

The Shannon-Fano (S-F) coding module calculates a possible S-F code and the code entropy. A separate program was developed to calculates a number of S-F codes using a number of different heuristics, but one heuristic consistently created the best code every time, so the STAF program uses only this heuristic. Once the entropy is calculated, it is a simple matter to calculate the length of a file coded in this manner, and the percentage compression that may be gained with this method

The heuristic used to calculate the S-F tree is this: Starting from the top down, the whole array containing the sorted character frequency from 0 to the number of distinct characters (symbols) is scanned,, and split in half as close as possible to the middle of the frequency. The program splits the top half, (with fewer characters, but appearing more often) and gives each character in this section a prefix of "zero", and the bottom half a "one." Now the array is split into two approximately equivalent parts (according to frequency), and the same heuristic for splitting the initial array in half is called again, recursively, to split the top, and bottom halves of the array, and add the next character in the S-F code, until all the codes have been generated.

In the routine that actually finds the middle of the array, a check is added that has been coined "Heat-Seeking; Capability". Just as a heat-seeking missile or torpedo searches for a source of heat as its target, so this "Find the middle of the array" routine seeks the: closest point to the middle. The less advanced "Find the Middle" heuristics checks the array, returning the point exactly in the middle or earlier, while the heat seeking capability checks on both sides of the exact middle point of the array, and returns the closest point, whether it be on the top, or bottom side of the array. The old method can be compared to the way contestants had to guess the prices of items on the popular T.V. game show "The Price Is Right", where the winning answer was "The contestant closest to the actual retail price, but not over is...". This method seems a little unfair because the winner was not necessarily the one who was the closest, but the closest one who guessed underneath the actual price. The "Heat-Seeking" capability eliminates the disparity, and returns the split **closest to the middle**.

The Shannon-Fano codes are pleasing to display and analyze, since it is very apparent that the code is self-separating. Huffman codes (explained next) are more complicated in creation, and this results in a code that is not as obviously self separating.

The Huffman Compression Technique

The Huffman codes are created using a single heuristic. Just as in the S-F module, a separate program was developed which coded a number of Huffman codes and compared them. For each file tested, the same heuristic gave the best result. Each code had the same entropy, but one heuristic produced codes where the maximum code length was less than others. This is the heuristic that has been implemented in the program. The Huffman code entropy is calculated, and this appears in the entropy report module.

Briefly, Huffman codes are created by fusion [Huff52], [Kins91a]. Shannon-Fano codes are created by splitting an array into successive halves, quarters, (etc... (called "top-down splitting"), while Huffman codes are created by repeatedly (recursively) merging the two symbols with the smallest frequency, creating a new entry with the combined frequency (called "bottom-up binary fusing") and adding this new node to the list/array from which the two symbols were located. The two individual symbols are also removed from the list, since they are now represented by a single combined (and thus higher) frequency.

This process continues, always subtracting two nodes from the list, and merging their composite probability back to the list, until one large binary tree is formed. The Huffman code can then be read from the: tree, starting from its corresponding leaf going up through the branches to the root, where each step up "left" is assigned a "one", and each step up "right" is assigned a "zero". A more detailed description, with examples of both Huffman and S-F codes is given in [Kins91a].

The LZW Compression Technique

This algorithm is a non-statistical dictionary algorithm, and thus it is possible (and **likely)**

that coding in this case could exceed the symbol based entropy, H_a. Currently the only available statistics are those compiled by actually running the source bit stream through this algorithm. The technique creates a dictionary, and this heuristic clears the dictionary when it fills up. Since the figures included in the entropy module are for comparison against the statistical techniques, a detailed description of the algorithm can be found in [KiGr91], [Kins91b], [Welc84], [Stor88].

3.2 Frequency Report Module

This second module in the STAF program provides the user with two different types of charts: (i) standard, and (ii) sorted charts. The full length report calls for both charts, while the short report calls far only the sorted chart.

Sorted Frequency Chart

The sorted frequency chart looks at the whole file to be analyzed, and sorts the symbols appearing in the source file in a descending order. Characters in the normal ASCII character set which do not appear in the file are not shown, for clarity. The chart shows each character, the integer count of the number of times it appears in the source, and the frequency or percentage, p_i, of the file that is that specific character. A typical text file's sorted chart would probably begin with <Space>, followed probably by <CR> (carriage return) and then perhaps the frequent letters E,T,A, and so on. Recall that the Shannon-Fano and Huffman codes are constructed on the basis of this chart.

Standard Frequency Chart

The Standard Frequency Chart prints the entire 256-character ASCII character set each with its corresponding count and frequency. This chart would be useful to show how the different characters are used, and which, if any blocks of characters are either used extensively or not at all. A typical Analog-to-Digital converter supplying 8-bit data could create data where this chart or a graphical representation of it could be useful in conjunction with another type of data compression using patterns derived from inspection of the chart. Coding could take advantage of "clusters" of certain symbols being used more often, and codes could be calculated accordingly.

3.3 Half-Byte Encoding Module

Certain types of data contain extensive numerical figures. Business charts, spreadsheets, scientific measurements (or even the sorted frequency charts the STAF program generates) contain many numerical figures which could be compressed. Normally, a character byte takes eight-bits (S_a), but if the compression algorithm anticipates a numerical string, a four-bit code could be utilized to encode the digits zero to nine, creating a compression ratio of nearly 50%.

This four-bit code means 16 characters can be encoded this way, so that after the ten digits are assigned, there are six extra unused codes, which could be classified as "numerical" and encoded as such. For example, a phone number (1-800-555-1212) contains 14 characters, including the three hyphens. This could be encoded into nine characters (A starting control byte, length of string, and the 14 nibbles) for a 64% saving. For longer numerical strings, a savings approaching 50% could be achieved. Typically a compression program would assign the six extra characters to be ones that normally show up in association with numbers. ("$*/-,."). Thus, the following types of strings would be encodable: l-800-668-1234, 1984,1988,1990-91, 08/07/91-5/01/68, $*** 1,234.56, and 3-2212-01176-3284. For highly statistical and numerical files, a compression of up to 50% could be attained with half-byte encoding.

3.4 Run-Length Encoding Module

Run-length encoding (RLE) is a very simple technique, useful for highly repetitive characters strings. The first type of RLE is when the encoder anticipates a number of blanks. These can be replaced with two characters: a special character, and the count, n, of the number of blanks. Thus, in a chart or graph that has 10 blanks in a row, they could be encrypted as a special control signal character, and then the number ten, to signify "10 blanks in a row." An extrapolation of this technique is a three character code used to represent repeated characters. For example, if the encoder encounters six Rs, they could be encrypted as a [special character| the original character| repeated 6 times].

Normally it is quite rare to have more than two or three characters repeated in a row, so this technique would not be a very fruitful one. Multiple spaces however, are a little more common, and are prime candidates for compression. This module only counts repeated strings of three or more characters, because compression savings only start with repetitions of more than three characters.

3.5 Diatomic Analysis Module

The diatomic compression technique is another very specific technique, which takes certain strings of characters and replaces them with a shorter code. In this case, the program looks for all *pairs* of characters, and replaces the most common pairs with a single character code. This results in a compression percentage of 50%, though it probably would not be possible because of the limited number of available characters to signify pairs of characters. Thus only the most common pairs would be encrypted. The most common pairs in the english language are E_, _T, TH, _A, and S_ [Held87].

4. EXPERIMENTAL RESULTS

Experiments were performed on a variety of files, most for the IBM PC. The files were chosen for benchmarking the results. The STAF program was run on a 33-MHz IBM Compatible computer, and output sent to an ASCII file on disk. Table 4.1 summarizes the output the STAF program created on each of the files.

4.1 Benchmark Files Description

README.DOC - This text file contains mostly upper and lowercase letters, with a sprinkling of a few numbers. It is Borland's Turbo C++ help file containing the last minute changes to their product. The file contains 82 distinct characters, and would represent a typical word processor letter or document.

AMIHELP2.HLP - The Windows 3.0 HELP reference file for the Desktop Publishing program *AMI*, available to the PC user when <F1> is pressed. The file consists of mostly ASCII text, but contains a generous sprinkling of control and non-printable characters.necessary for Windows to be able to interpret it,

WORKS.EXE - One of two executable files tested. This is the .EXE code from the popular Word-Processing/DataBase/Spreadsheet/Communications software package from Microsoft. This fits in the category of non-windows applications.

MILLEBV4.EXE - Another executable file, this one is an implementation of the popular card game Mille Bornes. A good mouse-based, windowed (Not MS 'Windows) program with excellent graphics.

Table 4.1 STAF analysis of Benchmark Files

FILE	Entropy / Compr Len / Compr Pct	Theoretical Var. Length Codes	Shannon-Fano Coding	Huffman Coding	LZW Coding
README.DOC (16203 bytes) (82 symbols)	4.825 bits/char 9773 bytes 39.7 %	5.362 10861 32.97 9b	4.92168 9968 38.479 %	4.8565 1 9836 39.294%	3.904 7908 51.194 %
AMIHELP2.HLP (279655 bytes) (251 symbols)	4.898 bits/char 171232 bytes 38.77%	5.337 186561 33.289 %	4.94101 172722 38.237%	4.93135 172384 38.358%	2.870 100329 64.124 %
WORKS.EXE (381776 bytes) (256 symbols)	7.409 bits/char 353556 bytes 7.392%	7.956 379676 0.55%	7.45306 355674 6.837%	7.43783 354947 7.027 %	7.822 373275 2.227%
MILLEBV4.EXE (216455 bytes) (256 symbols)	7.142 bits/char 193254 bytes 10.719%	7.697 208250 3.791%	7.23544 195768 9.557 %	7.16612 193892 10.424 %	5.934 160549 25.828%
TIGER.BMP (212086 bytes) (55 symbols)	1.065 bits/char 28246 bytes 86.682%	1.292 34246 bytes 83.853 %	1.23237 32671 bytes 84.595 %	1.23224 32667 bytes 84.597 %	0.153 4069 bytes 98.081%
FIT.PCM (20556 bytes) (202 symbols)	6.469 bits/char 16622 bytes 19.14%	17997 7.004 bytes 12.45 %	6.587 16924 bytes 17.67 %	6.4998 16701 bytes 18.75 %	6.279 16134 bytes 21.51 %
NUMBERS.TXT (16738 bytes) (226 chars)	3.733 bits/char 7810 bytes 53.342 %	4.375 9155 bytes 45.307 %	3.78104 7910 bytes 52.737 %	3.75881 7864 byes 53.015%	2.534 5302 bytes 68.324 %
SHORT.TXT (894 bytes) (34 symbols)	4.137 bits/char 463 by&es 48.294%	4.582 512 bytes 42.729 %	4.1868 467 bytes 47.679 %	4.1656 465 bytes 47.945 %	5.038 563 bytes 37.025%

v5

TIGER.BMP - A very simple Windows 3.0 Paintbrush drawing of a cat's face. Although **this may** not be as detailed **as** a typical drawing, it illustrates the waste that can be eliminated **from** every drawing.

FIT.PCM - A digital voice sample of a person speaking the word "FIT." The sample is **stored** in pulse code modulation (PCM) form. This sample bit stream would appear random, unless displayed in **graphical PCM format.**

NUMBERS.TXT - The STAF program creates reports which contain many numbers. The **file NUMBERS.TXT** is actually the STAF **report** for the README.DOC **file.** The sample of this

file can be created by running STAF on README.DOC, and then running STAF on the output file README.ST2, after first renaming it to a different name.

SHORT.TXT - A shorter memo, created to illustrate Huffman and S-F codes. It contains roughly 30 distinct symbols, and is used to illustrate these two types of coding simply.

4.2 Observations

The specific implementation of the Huffman code is better than the: S-F code in every instance of the benchmarks that were tested. From the selected files, it appears that the more symbols in the source bit stream, the smaller the difference becomes. The two EXE file (WORKS, MILLEBV4), each using all 256 8-bit codes exhibit differences in code entropy of 0.015 and 0.07 bits/character. The two text files on the other hand (README.DOC (82 symbols) and NUMBERS.TXT (226 symbols) exhibit differences in code entropy of 0.65 and 0.02. The difference decreases in proportion to the number of symbols in the bit stream. The AMIHELP2.HLP file also supports this observation: It too has more than 250 symbols in the file, and the Huffman and S-F difference in entropy is approximately 0.01.

The TIGER.BMP picture is quite simple, and comes in with an entropy of a resounding 1.065 bits/character. The Huffman and S-F code entropy is virtually identical, though the S-F entropy is still larger.

The file SHORT.TXT is a small paragraph typed in from a book, using lowercase letters only. Containing 30 characters, it is a short memo, illustrating clearly Huffman and S-F codes that can easily be reconstructed. The LZW (37.%) algorithm does not appear to perform as efficiently as the variable length coding routines (48%). This is due to the brevity of the file. If the file becomes any larger, the LZW will perform better, since it will be able to find more redundancy and patterns in the text.

The LZW compression algorithm is non-statistical, and thus its statistics can really only be used for comparison. The LZW algorithm is better for each file by approximately 15-20%, except for the WORKS.EXE program, where the LZW routine makes a remarkably poor showing. Typical LZW gives compressions of 50-60% for normal text files, and only 20-30% for more evenly (or random) distributed probabilities like the .EXE files and digitized speech. The LZW routine compresses and amazing 98% on the TIGER.BMP picture, though the picture is a relatively simple, 8-color diagram. Nevertheless, all BMP stored format pictures can be compressed - some more than others.

5. CONCLUSIONS

This paper presents a statistical analysis of files computer program developed to design optimal statistical codes for data compression. A set of benchmark files has been selected to test relative merits of the Shannon-Fano (S-F), Huffman, and Lempel-Ziv-Welch (LZW) optimal codes. Entropy calculations and frequency of occurrence help in assessing the best statistical techniques that can be applied to the given data stream. The frequency of occurrence is used to design optimal Huffman and S-F codes for that stream. The optimal codes are designed on a set of heuristics also evaluated in the study. The specific implementation of the Huffman code appears to better than the optimal S-F code for all the files tested. Our implementation of the LZW algorithm is also better than Huffman by 15-20%. The LZW gives compressions of 50-60% for normal text files, and 20-30% for executable files. This statistical analysis of files program could be used as a tool in designing and implementing compression algorithms in future packet radio BBSs and other data transfer systems.

ACKNOWLEDGEMENTS

This work was supported in part by the University of Manitoba and the Natural Sciences and Engineering Research Council (NSERC) of Canada.

REFERENCES

[DuKi91] D. Dueck and W. Kinsner, "A program for *statistical* analysis of files," *Technical Report,* DEL91-8, Aug. 1991, 50 pp.

[Held87] G. Held, *Data Compression: Techniques and Applications, Hardware and Software Considerations. New* York (NY): Wiley, 1987 (2nd ed.), 206 pp. (QA76.9.D33H44 1987)

[Huff52] D.A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRE,* vol. 40, pp. 1098-1101, Sept. 1952.

[KiGr91] W. Kinsner and R.H. Greenfield, "The Lempel-Ziv-Welch (LZW) data compression algorithm for packet radio," *Proc. IEEE Conf. Computer, Power, and Communications Systems,* (Regina, SK; May. 29-30, 1991), 225-229 pp., 1991.

[Kins91a] W. Kinsner, "Review of data compression methods, including Shannon-Fano, Huffman, arithmetic, Storer, Lempel-Ziv-Welch, fractal, neural network, and wavelet *algorithms," Technical Report,* DEL91-1, Jan. 1991,157 pp.

[Kins91b] W. Kinsner, "Lossless and lossy data compression including fractals and neural networks," *Proc. Int. Conf. Computer, Electronics, Communication, Control, (Calgary,* AB; Apr. 8-10, 1991), 130-137 pp., 1991.

[Kins91c] W. Kinsner, "Lossless data compression for packet *radio," Proc. 10th Computer Networking Conf., (San* Jose, CA; Sept. 29-30, 1991), this Proceedings, 1991.

[Stor88] J.A. Storer, *Data Compression: Method and Theory. New* York (NY): Computer Science Press/W.H. Freeman, 1988,413 pp. (QA76.9.D33S76 1988)

[Welc84] T.A. Welch, "A technique for high-performance data compression," *IEEE Computer,* vol. 17, pp. 8-19, June 1984.