Proc. ARRL 10th Computer Netwotking Conference, 1991

LOSSLESS DATA COMPRESSION ALGORITHMS FOR PACKET RADIO

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

This paper is a review of important data compression methods and techniques, showing their evolution from the simplest data suppression to the modem adaptive (universal) methods. Selected lossless techniques for critically important data, requiring perfect compression and reconstruction are presented. Lossy techniques for imperfect data such as speech, images, biological signals, and casual text are mentioned.

1. INTRODUCTION

1.1 Motivation

Transmission of data over telephone lines and packet radio, using standard data and file transfer protocols, as well as storage of data on magnetic media and in other storage devices can both be improved by data compression techniques. Such techniques reduce space, bandwidth, and input/output load requirements in digital system. For example, the statistical variable-length Huffman technique [Huff52] compresses text by 20%. This technique requires prior knowledge about the statistics of the file to be compressed Even better results may be obtained with the arithmetic coding technique, as implemented by Witten, Neal and Cleary [WiNC87]. The run-length encoding used in facsimile can compress a single 81/2x11 inch sheet so that its transmission can be done in 30 seconds on a voice-grade line at 9600 bps. The popular adaptive Lempel-Ziv-Welch (LZW) general purpose technique [Welc84] can compress data (text., numeric, mixed, and bitmapped images) by 40 to 60%. This technique does not require *a priori* knowledge of the file structure, data types, or usage statistics, and can operate on files of any length. The compression is noiseless and reversible in that a decompressed file is the exact image of the source file. This contrasts with data reduction and abstraction techniques in which data are deleted from the source. Nevertheless, such lossy data compression with fractals, neural networks, and wavelets are important techniques for research and practical applications, as they can achieve impressive compression ratios as high as 10,000:1 [Kins91a].

There are many other techniques capable of compressing and decompressing data efficiently [Stor88], [Held87], [Lync85], [Regh81], [Kins91a]. Which one is suitable for a given data or file structure? How should we measure the efficiency of the techniques? How easy is it to implement them? These and other questions have to be answered before using the best methods and techniques. Therefore, the purpose of this paper is to provide a review of the basic data compression methods and techniques, and to show their evolution from the simplest data suppression to the modem adaptive (universal) and lossy methods.

1.2 Models of Data Compression

Data compression refers to the removal of redundancy from a source by a proper mapping into codewords, carrying all the necessary information about the source so that &compression could be possible without loss of information. The compression and decompression processes are illustrated in Fig. 1.

A stream of p input message symbols (source characters) M is compressed into a smaller string of q codewords $\Delta(M)$ according to a particular algorithm, and passed through a medium (data storage or communication links). At the receiver, the compressed data A(M) are mapped back into the original source M, without any losses. The compression can be done in either hardware, software, firmware, or any combination of them. Software solutions may be applicable for slow data streams (Mbit/s), while modern parallel pipelined hardware solutions may provide speeds of hundreds of Mbit/s.

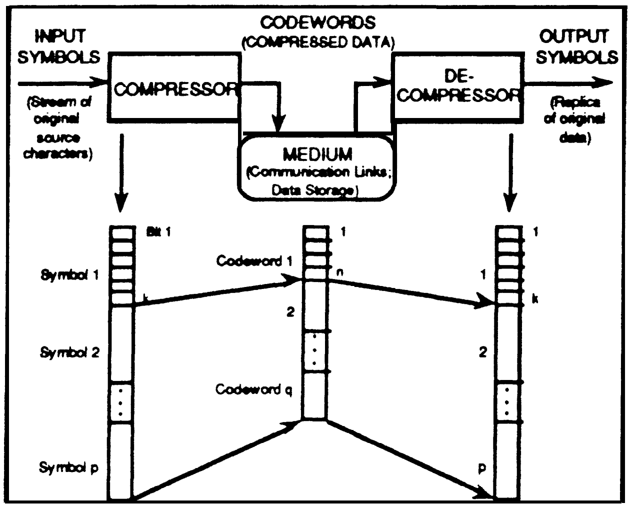

Fig. 1. A model of transparent data compression.

The compression and decompression processes may have a number of attributes, as shown Fig. 2. A compression is reversible if the source data can be reconstructed from the compressed codewords. The compression is noiseless when no information is added into the decompressed data, thus making it the exact replica of the source. It is also *lossless* if no information is removed from the recovered data. For example, the Huffman, LZW and WNC techniques are noiseless and lossless. In contrast, nonreversible (or *lossy*) mapping (data compaction or abstraction) removes redundancy using approximate methods, and the exact reconstruction of the source is not possible. For example, speech compressed using the linear predictive coding (LPC) or adaptive differential pulse code modulation (ADPCM) algorithms cannot be reconstructed exactly.

Compression is called ***transparent*** when it is done outside any interaction with a computer programmer. Compression that is not transparent is also called ***inferactive.*** Compression may be either statistical (e.g., Huffman) or nonstatistical *(e.g., LZW)*. ***In the statistical*** techniques, symbol usage statistics or data types must be **provided in advance, based on** either an average or local analysis of the actual data. Since the statistics gathering process requires a single pass and the compression another, these techniques are also called ***two-pass*** **techniques. In contrast,** ***nonstatistical*** techniques employ ***ad-hoc*** rules designed to compress data with some success. The statistical techniques may produce codewords that are either of ***fixed length*** (LZW) of ***variable length*** (Huffman), with the former giving higher compression ratio.

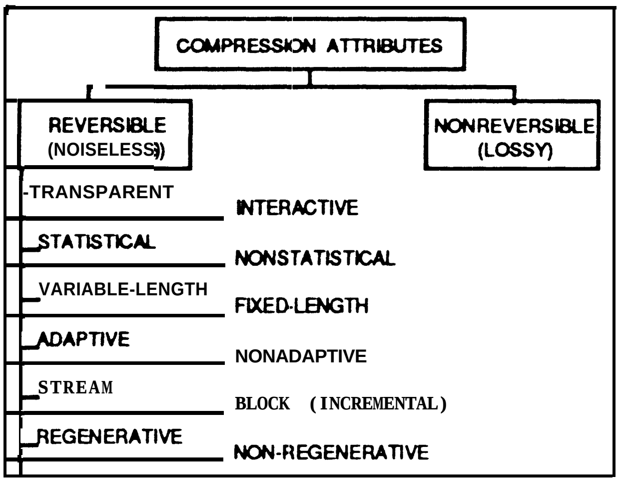

Fig. 2. Attributes of data compression methods.

The statistical and nonstatistical techniques may also be classified as either adaptive or nonadaptive. The ***adaptive*** (or ***dynamic***) compression does not require advanced knowledge, and develops the necessary translation tables (e.g., LZW) or the data statistics (e.g., dynamic Huffman and WNC) based exclusively on the incoming source stream. They are also called one-puss techniques. The ***nonadaptive*** (or ***static)*** techniques generate codewords, without affecting the original translation rules or data statistics. Depending on the method of outputting the codewords, a compression technique may be classified as stream or block. A ***stream*** technique outputs a codeword as soon as it is available, while the ***block*** technique must wait until the compression of a block is completed. For example, the arithmetic coding is a block technique, with recent improvements that include ***incremental*** operation whereby the entire block is broken into smaller ones that are output more often (e.g., WNC). Compression may also be regenerative or nonregenerative. In the ***non-regenerative*** techniques, the translation table must be transmitted to the receiver, or else decompression is not possible. ***The regenerative*** methods do not require the transmission of the translation tables, because they are capable of reconstructing the table from the codewords. If the compression and decompression phases take the same effort **(time and real *estate*), then it is called *symmetric.*** Clearly, the methods that are reversible, noiseless, lossless, transparent, adaptive, regenerative, and symmetric seem to be the most desirable.

1.3 Redundancy and Entropy

The amount of data compression can be measured by the compression ratio defined as

$$R_c = \frac{pk}{qn} \tag{1}$$

where *k* is the number of bits per symbol in the original message, *p* is the number of source characters in a **message, *n* is the number of bits in a codeword, and *q* is** the number of codewords. Thus, *pk is the* number of bits in the original data string, and *qn* is the number of bits in the compressed string. For example, $R_c = 2:1$ **signifies compression** by **50%.**

Redundancy can also be expressed in terms of entropy of a *code alphabet, Γ, corresponding* to a *string (source) alphabet, S*, which in turn is taken from a *symbol alphabet Σ. The* code alphabet is also called *dictionary, D,* which contains a set of strings, but may also include *Σ or* even *S. The* string alphabet *is first-order if the* **probability** of taking the next symbol does not depend on the previous symbol in the generation process of the string.

Entropy of a first-order source alphabet, S, taken **from** a binary symbol alphabet $\Sigma\{0,1\}$ (also representing the number of levels in the signal used to transmit the message) can be derived in the following form

$$H_a = -\sum_{i=1}^{m} p_i \log_2 p_i \tag{2}$$

where p_i is the probability of **occurrence** of **each** symbol in S, $m=|S|$ is the number of symbols in the source **alphabet, and the base *b* of the logarithm is equal to the** length of the symbol alphabet $b=|\Sigma|$ (**2** in this example).

Redundancy R_a in the source alphabet, *S*, is measured as the **difference** between a unit of information H_1 **for** the **alphabet,** S, **and** entropy H_a **as** given by **Eq.** 2. **Thus,** if H_1 *is*

$$H_1 = \log_2 m \tag{3}$$

then

$$R_a = \log_2 m - H_a \tag{4}$$

which indicates that if the character probabilities p_i are **equal,** the entropy must be $H_a = \log_2 m$, and there is no **redundancy in the source alphabet, $R_a = 0$, implying that a random source cannot be compressed.**

The number of bits, λ_i, required to encode a symbol whose probability is p_i can be estimated from

$$\lambda_i = \lceil -\log_2 p_i \rceil \tag{5}$$

where $\lceil x \rceil$ is the ceiling function producing the closest integer greater or **equal to x.** In **practice,** the codeword **length may not be exactly equal to this estimate. For the actual code, we can calculate the entropy of the code alphabet, Γ, corresponding to** the **source alphabet, *S*, by** taking the sum of products of the individual codeword **probabilities p_i** and the **actual corresponding codeword lengths, λ_{ci},**

$$H_c = \sum_{i=1}^{m} p_i \lambda_{ci} \tag{6}$$

This **difference** between the **code** entropy and the source entropy shows the quality of the **actual** code; if both are **equal,** *then the code is* **called** *perfect in tk information-theoretic sense.* For example, **Huffman** and **Shannon-Fano** codes **are** close to perfect in that sense. Clearly, no statistical **code** will be able to have entropy smaller than the **source** entropy.

1.4 Classification

Figure 3 shows the major **lossless** data compression methods, such as the run-length, statistical and **adaptive,** as well as a class of lossy compression **methods used to** reduce the bit rates of speech signals, images and biological signals. The **programmed** and hybrid **methods** may be either bssless **or** lossy or combinations of both, and will **not** be discussed **here** in detail. This section **presents an overview of the major methods, and discusses** some techniques belonging to the simple run-length, **statistical and adaptive methods.**

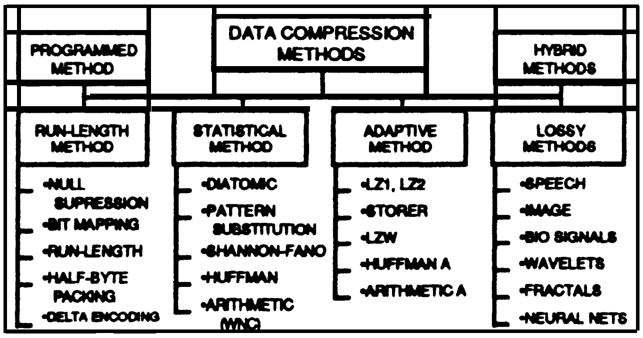

Fig. 3. Compression methods.

2. COMPRESSION METHODS

2.1 The Programmed Method

The programmed method requires interaction with the application programmer, who determines an efficient way of removing application-specific redundancy from the source, and programs it into the application. Such software is not very portable, and the development cost may be prohibitive. So, this nontransparent approach will not be discussed here. Instead, we shall concentrate on methods that have universal techniques and corresponding algorithms that can be made transparent

Another form of a programmed data compression is an object-oriented description of an image, and a programmed description of attributes. For example, Adobe's PostScript may reduce the facsimile image by 10: 1 and considerably reduce the description of fonts transmitted for printing.

2.2 The Run-Length Method

As shown in Fig. 3, this method includes the following techniques: null suppression with a pair, null suppression with a bit map, arbitrary character suppression with a triple, nibble (half-byte) packing, and delta encoding. These techniques will be described briefly next.

An early compression technique, the ***null suppression with a pair,*** was designed to suppress blanks (nulls) in the IBM 3780 bisync protocol, with 30-50% throughput gain. A sequence of blanks is substituted by an ordered pair of characters (S_C, Count) where S_C is a special suppression indicator, and is followed by the number of blanks to be suppressed. For example, a string ABCbbbbbbbXYZ is reduced to ABCS_C7XYZ.

Another technique is ***the null suppression with a bit nap.*** If the specific characters such as blanks are distributed randomly within a string, the blank suppression technique cannot be used. Instead, we can subdivide the string into substrings whose length matches the number of bits in each character (word), and the position of the blanks in a substring can be marked with individual bits in the word, thus creating a bit map whose compact representation can be placed within each substring.

This general run-length technique may be modified for facsimile which has many white (0) and black (1) pixels. The pixels may be represented by individual bits in a word in a bit-mapped manner. Thus, four bytes can carry 32 pixels. If we encode a group of white pixels by a single byte and black pixels by another byte, then the run-length compression can be improved. For example, a four-byte sequence containing the numbers 150/10/220/5 represents 150 white pixels, followed by 10 black pixels, which are followed by 220 white and 5 black pixels. This has the potential of 32:1 compression ratio (total 1024 pixels against 32 when each bit represents a single pixel only).

Another technique is the ***arbitrary character suppression with a triple*** which is a direct extension of the null suppression to an arbitrary character. Instead of the ordered pair characters, we must now use three characters: the special suppression indicator, followed by the repeated character and the character count.

The ***nibble (half-byte) packing*** is a modification of the run-length and bit-mapping procedures. Rather than suppressing the repetitive characters, different characters may have identical upper nibbles which could be suppressed. For example, the upper nibble in ASCII financial characters could be suppressed, and the lower nibbles could be packed. The same would apply to the EBCDIC representation.

Still another technique, the ***delta encoding,*** is an important extension of nibble packing. This relative encoding scheme applies to substrings that are similar, thus the difference (delta) between the substrings would contain only few non-zero bits or characters, and such delta substrings could be compressed using the above suppression techniques. Examples of such strings could be found in telemetry and facsimile.

In telemetry data, if measurement data are slowly varying within a period of time, their values do not change significantly and only differences could be transmitted, leading to smaller numbers which could have more compact representation. The delta encoding could run until either the range of the small numbers is exceeded or data integrity would require a periodical reset of the absolute value.

In facsimile data, the straightforward run-length encoding can be used to suppress the white or black pixels. The delta technique further reduces the scan line data. If a reference line has been transmitted without compression, only the difference of the subsequent lines may be required to reconstruct the source data.

2.3 The Statistical Method

This method includes the following techniques:

diatomic encoding, pattern substitution, and variable-length encoding.

The *diatomic encoding* technique takes two characters and replaces them with a single character, thus having fixed length and a possible 2:1 compression ratio. Since the number of special characters is limited, not all the pairs can be compressed. Instead, the most frequent pairs are selected for compression. The pairs must first be identified by a file analysis program, and the most frequent pairs ar selected as candidates for compression. Extensive studies have been done on different data types to identify such pairs for English, text, FORTRAN, COBOL and BASIC [e.g., Held87].

The pattern substitution technique is an extension of the diatomic encoding in that it also assigns special short codes to common multicharacter groups. This scheme may benefit files containing computer languages (e.g., WRITE, READ, etc.) or natural languages ("and", "the", etc.). The substitution can be done by single codes or special pairs such as $n, otherwise never appearing in the string. Compressions of 20% were reported.

Another technique is *the variable-length encoding.* All the previously discussed techniques employ fixed-size character codes and produce fixed-size codewords. A more compact code can be achieved by assigning shorter codewords to frequent characters and longer codewords to less frequent characters or their groups. This approach was used by Samuel Morse when he selected short sounds for the frequent characters (E,T,A,N) and longer sounds for the less frequent ones. There are three effective variable-length techniques such as the ordinary and generalized step codes, the Shannon-Fano, Huffman, and WNC arithmetic coding, as described in Section 3.

2.4 The Adaptive (Universal) Method

Adaptive (or universal) techniques can be seen as generalizations of many of the previous techniques. The key idea here is that *no statistical knowledge* is necessary to convert the available input stream into codewords and vice versa. Instead, an optimal conversion is possible, *using* a form of *learning* about *the structure* of either the source stream or the codeword stream. Thus, the techniques are capable of producing good codewords, either without prior knowledge about the data statistics, or even better codewords with data statistics acquired by observation during the compression phase. Examples of such adaptive statistical nonregenerative techniques include the Huffman and arithmetic code WNC. Examples of adaptive nonstatistical regenerative techniques are due to Lempel and Ziv (LZ1 [LeZi76], [ZiLe77] and LZ2 [ZiLe78]), Storer (heuristic dictionary algorithms) [Stor88], and Lempel, Ziv and Welch (LZW) [Welc84], [KiGr91]. Other adaptive methods are being developed, including an algorithm for binary sources and binary code alphabet, a memory efficient technique based on fused trees, techniques requiring a small number of operations per encoded symbol. Some of these techniques are described by Kinsner [Kins91a], and the basic ideas behind the LZW technique are presented in Section 3.

2.5 Lossy (Approximate) Methods

The previous methods and techniques are lossless and noiseless in that nothing is lost during the data compression and nothing is added during the reconstruction processes, respectively. These properties are essential in perfect data in which a loss of a single bit may be catastrophic [Kins90]. On the other hand, imperfect data such as speech, images, biological signals, and casual electronic mail text may tolerate minor changes in the source and still be sufficient after their reconstruction, according to a distortion measure. This class of lossy techniques may produce extremely large compression ratios (10,000: 1). There are three emerging classes of compression techniques with enormous potential based on fractals, neural networks, and wavelets. We are working on all three techniques related to speech and pictures [Kins91a], [Kins91b].

3. EXAMPLES OF ALGORITHMS

3.1 The Shannon-Fano Coding

The Shannon-Fano (S-F) code has efficiency approaching the theoretical limit and is the shortest average code of all statistical techniques. The code is also good because it has the self-separating property (or the prefix property) because no codeword already defined can become a prefix in any new codeword. This property in the S-F code leads to the ability to decode it "on the fly", without waiting for a block of the code to be read first. The main question is how to generate such an optimal, information efficient and self-separating code? The answer is in the application of an old principle of binary *search tree*, BST (or halving, or binary chopping). How should we halve the entire set of symbols to achieve the optimum variable-length code? *A* viable approach is to use the overall (joint) probability of symbols and assign the first 0 (or 1 –the choice is arbitrary at this first step) to the symbols above 0.5 and the first 1 (or 0) to those with joint probability not greater than 0.5. This subdivision continues on the halves, quarters and so on, until all the symbols are reached, as shown in Fig. 4.

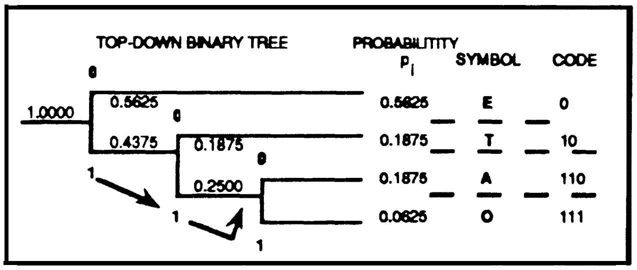

Fig. 4. Shannon-Fano top-down binary tree.

3.2 The Huffman Coding

The Huffman coding scheme [Huff52] belongs to the same class as the Shannon-Fano technique in that both are statistical, variable-length, optimal-length, self-separating, and use the binary decision principle to create the code. The difference lies in the application of the BST; i.e., the S-F code is created by top-down binary chopping, while *the* Huffman code is formed by bottom-up ***binary fusing***. As already described, the S-F code is created by subdividing the symbol table set into two groups whose joint probability is as close to 0.5 as possible. Then, the halves are divided into quarters, and so on, until all the individual symbols are covered. The method is termed top-down because the process starts from the root of the tree where the global probability is 1.0 and progresses to the leaves.

In contrast, the Huffman code formation starts from the leaf level and progresses to the root of the tree by fusing the probabilities of the individual leaves and branches, as shown in Fig. 5. Thus, this method emphasizes the small differences between the leaves, while the S-F technique operates on averages. Although the two techniques appear to be the same on short codes, the differences will be discussed later in the section.

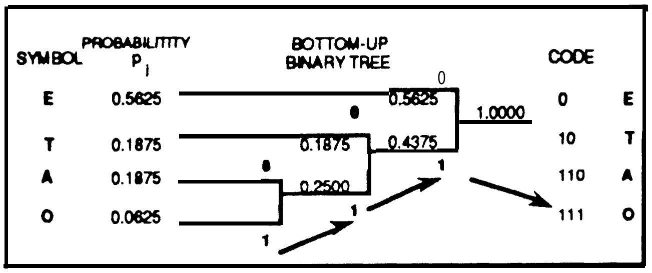

Fig. 5. Huffman bottom-up binary tree.

We have implemented both S-F and Huffman compression techniques, using the C language on the IBM PC. The code is described in [DuKi91a], and experimental results are presented in [DuKi91b, this conference].

3.3 Dynamic and Higher-Order Huffman

The S-F is better than the Huffman code on alphabets with spread probabilities (large variance), while the Huffman is better on alphabets with an even distribution of the probabilities (small variance). Although very attractive from the efficiency point of view, both methods have problems. The problems include the limited size of the translation table (a 256-entry table for 8-bit symbols is sufficient for single characters only), difficult decoding process on binary trees, the ***a priori*** knowledge of the alphabet statistics, leading to two passes over the data to be compressed, and variability in the statistics on large files requiring an adaptive analysis of the data and transmission of more than one translation table.

There are many modifications of the basic static Huffman and S-F algorithms to cope with those and other problems [Welc84, Regh81, Held87, Stor88]. In the original paper [Huff52], Huffman considered non-binary symbol alphabets, $|\Sigma|>2$ in addition to the binary case. Others considered code alphabet Γ with unequal cost, as well as source alphabet with inaccurate probabilities and trie (to distinguish it from a tree, [Stor88]) construction. Implementation issues are discussed in [Pech82] and [McPe85]. Many other references are provided by Storer [Stor88, Chapter 2] and [Kins91a].

The ***dynamic*** Huffman code was introduced by Faller [Fall73] and Gallager [Gall78]. A naive dynamic Huffman encoding and decoding would start with a source alphabet, S, whose symbols have equal probability. The probability distribution is then updated after a character has been encoded or decoded. The trie (not tree) construction algorithm must be deterministic for both the encoder and decoder so that the key procedures such as halving decisions and tie breaking are done identically. Although optimal, this method is inefficient because the trie must be reconstructed after each character in the message stream. Improvements and generalization of the dynamic algorithm, as well as an extension of the balanced trie concept is the splay tree and an alternative to this approach the transposition heuristic, are discussed in [Kins91a].

3.4 Arithmetic Coding

Arithmetic coding belongs to the statistical method, but is different from the Huffman class of coding techniques in that it develops a single codeword from the input symbol stream by interval scaling, using the fractal similarity principle. (Notice that this fractal similarity principle has never been pointed out in the arithmetic coding literature before). In contrast, the Huffman technique develops codewords by a table lookup procedure, with the table obtained from a bottom-up BST. Both techniques use statistical models of data that can be either obtained *a priori* (fued model) or adaptively (dynamic model). While Huffman coding generates codewords in response to individual symbols (stream technique), arithmetic coding either waits for all the symbols to arrive prior to sending the codeword (block technique) or outputs partial codewords when the interval has been located up to a predefined threshold (incremental technique). Arithmetic coding, and particularly the implementation by Witten, Neal *and* Cleary [WiNC87] is shown to be superior to Huffman coding. A tutorial on arithmetic coding is provided by Langdon [Lang84], [Kins91a], and a class of such techniques is presented by Rissanen and Langdon [RiLa79]. We are working on an implementation of the arithmetic coding.

3.5 Lempel-Ziv-Welch (LZW) Coding

The well known static LZ1 [LeZi76], [ZiLe77] and dynamic LZ2 ([ZiLe78]) nonstatistical algorithms due to Lempel and Ziv can be classified as adaptive (universal) techniques. The key i&a here is that ***no statistical knowledge*** is necessary to convert the input stream into codewords and vice versa. Instead, an optimal conversion is possible using a form of ***learning*** about the structure of either the source stream or the codeword stream, thus justifying the name “adaptive”. This class of techniques converts variable-length input strings to constant-length output codewords, using a dictionary (also called a conversion table). In the case of Huffman-type coding, the input strings are of constant length, while the output codewords are of variable length, and a ***fixed-length*** dictionary is provided to both the encoder and decoder. In the case of Lempel-Ziv-type coding, the dictionary is of ***variable length,*** and the encoder creates its local dictionary, D, in step with the incoming strings. Similarly, the decoder reconstructs ***D in*** step with the incoming codewords, thus making the method regenerative.

The learning in the algorithm depends on the following four heuristics: (i) initialization heuristic, ***IH*****,** **(ii) matching heuristic, *MH*,** (iii) updating heuristic, ***UH*****, and** (iv) deletion heuristic, ***DH*** **[Stor88], [Kins91a].** The dictionary ***D is*** first ***initialized to an*** initial string set D_0 which includes at least the string alphabet ***S*** in Storer’s dictionary approach, or is empty in the Lempel-Ziv approach. The encoder **dictionary *D* is constructed** by **repeatedly *matching*** the incoming character stream to the **entries in its *D*, until a new string s_m is found (or until any** other significant event occurs). The compression results from a substitution of s_m with an index **representing the string (provided the length of the index is** smaller than $|s_m|$ where $|\cdot|$ signifies the length of an object $(\cdot)$). ***Now, D*** is ***updated*** according to the matching s_m and the current contents of ***D. If the new*** **update exceeds the size of *D*, then some entries must be *deleted*, *using*** a deletion strategy. Since none of the four major activities has a single form, they are called heuristics. It is seen that a large number of adaptive **techniques can be derived, depending on the choice of the heuristics.**

The LZ1 adaptive (universal) technique has received extensive attention in literature because it can be the perfect algorithm in the information-theoretic sense and may be the best algorithm for many applications. Serial and parallel implementations of the scheme are discussed in [Stor88]. The LZW algorithm is another implementation of the LZ1 algorithm [Welc84]. Our LZW code is based on Nelson’s and Regan’s implementations [Nels89], [Rega90], and is included in [DuKi91a]. This C language implementation runs on an IBM PC, and is portable to machines supporting 16-bit integers and 32-bit longs. It is limited to arrays smaller than 64 Kbytes on the MS-DOS machines (12 to 14 bits used). Experimental results obtained with our benchmarks are presented in [DuKi91b, this conference].

4. STATISTICAL ANALYSIS OF FILES PROGRAM

In addition to implementing the Shannon-Fano, Huffman and LZW algorithms, we have also developed a statistical analysis of files (STAF) program for data compression [DuKi91a]. The program is required by all the statistical techniques to design optimal codes for the **data streams to be compressed. It first gathers all the vital statistics** about the specific file and then estimates the best run-length or statistical technique for the file. Finally, it reports on the findings, and develops an optimal Shannon-Fano and Huffman codes automatically. Notice that the program is not required by nonstatistical techniques such as the LZW.

5. OTHER IMPLEMENTATIONS

The popular adaptive LZW algorithm has very simple logic, leading to inexpensive and fast implementations. Good LZW implementations use 9- to 16-bit codes, handling most applications. A 12-bit code is suitable for medium-size files. Efficiency improves with larger codes. A tight coding of the algorithm can compress *75* Kbytes in a second on a 1 MIPS machine. The LZW technique can be found in several file compression utilities for archival storage.

For example, the MicroSoft MS DOS environment has enjoyed archival programs such as ARC by System Enhancement Associates of Wayne, NJ, as well as PKZIP by PKWARE of Glendale, WI. The ARC program has been ported to Unix, CP/M and other operating system environments. Machines with Unix can use the COMPRESS and COMPACT utilities. The Berkeley Unix has a COMPRESS command which is an implementation of the LZ algorithm, with a table of up to 64 K entries of at least 24 bits each (total of over 1,572 kbits or over 1% kbytes on most machines). The Apple Macintosh environment has several good programs, including PackIt IX/III and UnpIt by Harry Chelsey [Ches84], as well as StuffIt and UnStuffIt by Ray Lau [Lau87]. The StuffIt shareware is written in Lightspeed C. An improved version of StuffIt is now distributed as a commercial package [Stuf90]. The ARC 5.12 program uses a simple RLE and LZW algorithms for compression. The PackIt program uses Huffman encoding only. In the StuffIt, compression is done by the LZW and/or Huffman algorithms, and when they fail, by the less efficient RLE algorithm. Its LZW implementation is similar to the ARC 5.12 LZW, but uses 14 bits with a hashing table of size 18,013, rather than the 12/13 bits used in ARC. The ARC LZW implementation, in turn, is similar to that of the public domain COMPRESS utility in Unix. A recent LZW implementation for packet radio by Anders Klemets [Klem90] is designed to work in conjunction with the IBM PC implementation of the TCP/IP protocols by Phil Kam of Bellcore.

In addition, the LZW algorithm can be employed not only on files requiring perfect transmission (e.g., financial data), but also on imperfect data such as e-mail text of non-critical nature, weather data, and digitized speech transmitted using the store-and-forward mode. Files with poor data structures (sparse encoding of data and empty spaces) can also benefit from LZW compression.

6. CONCLUSIONS

This paper presents a classification of the major data compression methods and a number of useful compression techniques that could be suitable for packet radio. Although the top-down Shannon-Fano technique is better than Huffman on alphabets with large variance, while the bottom-up Huffman technique is better on uniform alphabets, the latter may employ heuristics to make it better on all alphabets. The arithmetic coding technique is better than Huffman. The Storer static and dynamic sliding dictionary techniques are implementations of the LZ1 and LZ2 algorithms, with essential generalizations to heuristic algorithms. The popular Lzw technique is also an implementation of the LZ algorithm.

In addition to the lossless techniques, lossy algorithms for compression of imperfect data such as noncritical electronic mail text, images, speech and other biological data, should also be considered in packet radio.

ACKNOWLEDGEMENTS

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

REFERENCES

[Ches84] H. Chesley, "PackIt." (Address: 1850 Union St. #360; San Francisco, CA 94123.)

[DuKi91a] D. Dueck and W. Kinsner, "A program for statistical analysis of files," *Technical Report,* DEL91-8, Aug. 1991, 50 pp.

[DuKi91b] D. Dueck and W. Kinsner, "Experimental study of Shannon-Fano, Huffman, Lempel-Ziv-Welch and other lossless algorithms," *Proc. 10th* Computer Networking *Conf.*, (San Jose, CA; Sept.- 29-30, 1991), this Proceedings, 1991.

[Fall73] N. Faller, "An adaptive system for data *compression," Proc. Seventh IEEE Asilomar Conf. Circuits & Systems,* pp. 593-597, Nov. 1973.

[Gall78] R.G. Gallager, "Variations on a theme by Huffman," *IEEE Trans. Information Theory,* vol. *IT-24,* pp. 668-674, June 1978.

[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. 10984101, 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 System,* (Regina, SK; May. 29-30, 1991), 225-229 pp., 1991.

[Kins90] W. Kinsner, "Forward error correction for imperfect data in packet radio," *Proc. 9th Computer Networking Conf.*, (London, *ON;* Sept. 22, 1990), pp. 141-149, 1990.

[Kins91a] W. **Kinsner,** "Review of **data compression** methods, including **Shannon-Fano,** Huffinan, 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.

[Klem90] A. **Klemets, "LZW** implementation for **KA9Q Internet** *Package* **NOS,"** *Packet Radio Bulletin, 5* **Oct.** 1990. **(The source of the code is available from sics.se under filename archive/packet/ka9q/nos/lzw.arc, and the** NOS is available **from thumper.bellcore.com. Address: Anders Klemets,** SMORGV, Sikv 51, **S-**13541, **Tyreso,** Sweden.)

[Lang84] G.G. **Langdon,** "An introduction to arithmetic coding," *IBM J. Res. Dev.,* vol. 28, pp. 135-149, March 1984.

[Lau87] R. **Lau, "StuffIt."** (Address: 100-04 70 Ave.; Forest Hills, NY 1137505133.)

[LeZi76] A. Lempel and J. Ziv, "On the complexity of *finite sequences," IEEE Trans. Inform. Theory, vol. IT-22,* pp. 75-8 1, Jan. 1976.

[Lync85] TJ. Lynch, *Data Compression: Techniques and Applications.* Belmont, CA: Lifetime Learning, 1985,345 pp. **(ISBN** o-534-03418-7)

[McPe85] D.R. McIntyre and M.A. Pechura, "Data compression using static **Huffman** code-decode **tables," J.** *ACM,* **vol. 28,** pp. **612416, June 1985.**

[Nels89] M.R. Nelson, "LZW data compression," *Dr. Dobb's J., vol.* 17, pp. 29-36, Oct. 1989.

[Pech82] M. Pechura, "File archival techniques using *data compression," Comm. ACM,* vol. 25, pp. **605-609, Sept.** *1982.*

[Rega90] S.M. Regan, "LZW revisited," *Dr. Dobb's J.,* **vol.** 18, pp. **126-127 and** 167, Jun. 1990.

[Regh81] ELK. **Reghbati, "An overview** of data *compression* **techniques,"** *IEEE Computer,* vol.14, pp. **71-75,** Apr. 1981.

[RiLa79] J. **Rissanen** and G.G. **Langdon,** Jr., *"Arithmetic coding," IBM J. Res. Dev., vol. 23,* pp. **149-162,** March 1979.

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

[Stuf90] "StuffIt Deluxe," *MacUser Magazine,* **vol.** 28, *pp.* **68-70,** Dec. 1990. (Address: **AladDin** systems, Deer Park Center, Suite **23A-171, Aptos,** CA 95003; Tel. **408-685-9** 175)

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

[WiNC87] I.H. **Witten,** R.M. Neal, and J.G. Cleary, "Arithmetic coding for data compression," *Comm. ACM,* vol. 30, pp. **520-540,** June 1987.

[ZiLe77] J. Ziv and A. Lempel, "A universal algorithm for **sequential data** compression," *IEEE Trans. Information Theory,* vol. *IT-23, pp. 337443, May* 1977.

[ZiLe78] J. Ziv and A. Lempel, "Compression of **individual sequences** *via variable-rate coding," IEEE Trans. Information Theory,* **vol. IT-24, pp. 530-536,** Sept. 1978.