1. Introduction

This short paper will present a number of compression options and alternatives. It is hoped that it will provide ample information to choose an appropriate compression algorithm and method for implementation in the Novell Wide Area Networking product.

It is desired to perform some type of compression on the data to be transmitted to a remote site to increase the apparent bandwidth of the (probably) low-speed link. This compression can take many forms, and can be applied at one or more layers of a networking protocol. In this paper, implementation at the data link layer will be explored.

In selecting a compression algorithm, it is necessary to consider the following items:

- Compression ratio
  This is the ratio of the data emitted from the compression routine versus the data received by the routine. For example, 300,000 bytes received and 100,000 bytes transmitted would be a 3:1 compression ratio.

- Speed
  The algorithm's speed must be considered, especially when Novell protocols are involved.

- Size of memory required to implement the protocol
  The amount of dictionary space required in RAM for each protocol.

1.1 Description of Method

To test the various compression methods, some representative data was required. This data was obtained from a public domain collection of binary, text and image files, collected by T.C. Bell and J.G. Cleary. Collectively, these files are referred to as the Calgary/Canterbury text compression corpus. This corpus is used in the book:

> Bell, T.C., Cleary, J.G. and Witten, I.H. Text compression. Prentice Hall, Englewood Cliffs, NJ, 1990

and in the survey paper

> Bell, T.C., Witten, I.H. and Cleary, J.G. "Modeling for text compression," Computing Surveys 21(4): 557-591; December 1989

The corpus is used to evaluate the practical performance of various text compression schemes. Several other researchers are now using the corpus to evaluate text compression schemes.

Nine different types of text are represented, and to confirm that the performance of schemes is consistent for any given type, many of the types have more than one representative. Normal English, both fiction and non-fiction, is represented by two books and papers (labeled book1, book2, paper1, paper2, paper3, paper4, paper5, paper6).

More unusual styles of English writing are found in a bibliography (bib) and a batch of unedited news articles (news). Three computer programs represent artificial languages (progc, progl, progp). A transcript of a terminal session (trans) is included to indicate the increase in speed that could be achieved by applying compression to a slow line to a terminal. All of the files mentioned so far use ASCII encoding. Some non-ASCII files are also included: two files of executable code (obj1, obj2), some geophysical data (geo), and a bit-map black and white picture (pic). The file geo is particularly difficult to compress because it contains a wide range of data values, while the file pic is highly compressible because of large amounts of white space in the picture, represented by long runs of zeros.

More details of the individual texts are given in the book mentioned above. Both book and paper give the results of compression experiments on these texts.

The corpus itself constitutes files bib, book1, book2, geo, news, obj1, obj2, paper1, paper2, paper3, paper4, paper5, paper6, pic, progc, progl, progp and trans. (The book and paper above do not give results for files paper3, paper4, paper5 or paper6.)

TABLE 1. Input File descriptions

Filename	Description	Lines	Words	Characters
bib	Bibliographic files (refer format)	6280	19274	111261
book1	Hardy: Far from the madding crowd	16622	141274	768771
book2	Witten: Principles of computer speech	15634	101221	610856
geo	Geophysical data	18	617	102400
news	News batch file	10059	53939	377109
obj1	Compiled code for Vax: compilation of progp	87	495	21504
obj2	Compiled code for Apple Macintosh: Knowledge support system	1213	4600	246814
paper1	Witten, Neal and Cleary: Arithmetic coding for data compression	1250	8512	53161
paper2	Witten: Computer (in)security	1731	13829	82199
paper3	Witten: In search of "autonomy"	1100	7219	46526
paper4	Cleary: Programming by example revisited	294	2166	13286
paper5	Cleary: A logical implementation of arithmetic	320	2099	11954
paper6	Cleary: Compact hash tables using bidirectional linear probing	1019	6753	38105
pic	Picture number 5 from the CCITT Facsimile test files (text + drawings)	0	49	513216
progc	C source code: compress version 4.0	1487	6313	39611
progl	Lisp source code: system software	2244	9235	71646
progp	Pascal source code: prediction by partial matching evaluation program	1966	4847	49379
trans	Transcript of a session on a terminal	2737	9255	93695
Total		64061	391697	3251493

These files, on their own, do not correctly represent traffic between routers. They do, however, give a good base indication of common types of files that are likely to occur. To more closely represent real traffic, a separate program was written to combine all 20 of these files into a single data stream, with PPP (Point-to-Point Protocol); IPX (Internetwork Packet eXchange); and NCP (Netware Core Protocol) headers prepended to the data. Data from each of the 20 files was read, 512 bytes at a time, until the data was exhausted. The 20 separate data streams were linear multiplexed into a single data stream, of 3,531,333 bytes (including headers). For this (contrived) example, the IPX and NCP overhead was about 8%. The program in the following figure produced this data stream.

Figure 1. Makepack.c

```
#include <stdio.h>
#include "ipx.h"
#define NUMHOSTS 18

struct NCPPACK Hosts[NUMHOSTS];

char *Files[NUMHOSTS] = {
"../files/bib","../files/book1","../files/book2","../files/geo",
"../files/news","../files/obj1","../files/obj2","../files/paper1",
"../files/paper2","../files/paper3","../files/paper4","../files/paper5",
"../files/paper6","../files/pic","../files/progc","../files/progl",
"../files/progp","../files/trans"};

unsigned long Filepos[NUMHOSTS];


main(argc,argv)
int argc;
char **argv;
{
    int x,y,done;
    FILE *fo;
    struct ipx *ipx;
    struct ncp *ncp;
    if (argc < 2) {
       fprintf(stderr,"Usage: %s outfile0,argv[0]);
       fprintf(stderr,"Builds an output file with PPP/IPX/NCP headers from the standard0);
       fprintf(stderr,"compression benchmark test0);
       fprintf(stderr,"Version 1.00  Tue Nov 10 13:34:02 1992      Dave Rand0);
       exit(1);
    }
    for (x=0; x< NUMHOSTS; x++) {
       ipx = &Hosts[x].ipx;
       ipx->ipx_sum = 0xffff;
       ipx->ipx_len = 512 + sizeof(struct ncp) + sizeof(struct ipx);
       ipx->ipx_tc = 0;
       ipx->ipx_pt = NW_PROTO_NCP;
       ipx->ipx_dna.x_net = rand() + (rand() << 16);
       for (y=0; y<6; y++)
           ipx->ipx_dna.x_node[y] = rand();
       ipx->ipx_dna.x_sock = NW_SOCK_FS;
       ipx->ipx_sna.x_net = rand() + (rand() << 16);
       for (y=0; y<6; y++)
           ipx->ipx_sna.x_node[y] = rand();
       ipx->ipx_sna.x_sock = NW_SOCK_FS;
       ncp = &Hosts[x].ncp;
       ncp->ncp_op = NCP_REPLY;
       ncp->ncp_seq = 0;
       ncp->ncp_conn = rand();
       ncp->ncp_task = rand();
       Filepos[x] = 0L;
    }
```

```
    fo = fopen(argv[1],"wb");
    if (fo == NULL) {
       fprintf(stderr,"Can’t open file %s for output!0,argv[1]);
       perror(argv[0]);
       exit(2);
    }
    done = 0;
    while (done != NUMHOSTS) {
       done = 0;
       for (x=0; x < NUMHOSTS; x++) {
           if (Filepos[x] == -1L) {
                                 done++;
           } else {
                                 packit(Files[x],&Filepos[x],&Hosts[x],fo);
           }
       }
    }
    fclose(fo);
}

unsigned char buf[512];
packit(file,pos,host,fo)
char *file;
unsigned long *pos;
struct NCPPACK *host;
FILE *fo;
{
    FILE *fi;
    int size;
    fi = fopen(file,"rb");
    if (fi == NULL) {
       fprintf(stderr,"Can’t open %s - continuing0,file);
       *pos = -1L;
       return(1);
    }
    fseek(fi,*pos,0);
    size = fread(buf,1,512,fi);
    if (size != 512)
       *pos = -1L;
    else
       *pos = ftell(fi);
    fprintf(fo,"377 03");
    fwrite(host,sizeof(struct NCPPACK),1,fo);
    fwrite(buf,size,1,fo);
    host->ncp.ncp_seq++;
    fclose(fi);
}
```

2. Test Results

The results of running the various compress programs are presented in the following table.

TABLE 2. Test results

Compression Program	Method	Max ratio	Output size	Ratio	Time	K/sec	RGF
HPACKA 0.75c0	LZA		1272999	2.774	307	11.50	4146
Info-ZIP 1.9	LZSSA	905	1297993	2.721	90	39.24	14422
Gandalf FZA	FZA		1310367	2.695			
ARJ 2.30NG	LZSSA	1658	1313249	2.689	58	60.89	22642
LHA 2.13	LZAH	1883	1470858	2.401	66	53.51	22286
AR002	LZAH	1859	1470865	2.401	176	20.06	8357
HYPACK 2.5	LZAH	23810	1528534	2.310	47	75.13	32522
PKZIP 1.10	LZSF	519	1549210	2.279	53	66.63	29231
Unix COMPRESS	LZW	550	1580342	2.235			
LZ	LZH	48	1641731	2.151	274	12.89	5992
DWC A5.01	LZW	532	1747822	2.020	11	321.03	158892
UNIX ARC	LZW	541	1872769	1.886			
PKPAK 3.61	LZW	541	1912705	1.846	19	185.86	100668
LARC 3.33	LZSS	8	1914044	1.845	84	42.04	22786
MDCD 1.00	LZW	515	1917596	1.842	35	100.90	54788
ZOO 2.10	LZW	503	1917643	1.841	80	44.14	23971
SQZ 1.08.2	LZW		1937345	1.823	270	13.08	7175
STAC 3.1 /p255	LZS	30	1949600	1.811	22	160.52	88618
Sea ARC	LZW	3003	2024142	1.745			
PRED 1.02	Prediction	8	2117640	1.668	5	706.27	423728
V.42bis (12 bit)	LZW	471	2274812	1.552	65	54.39	34997
STAC 3.1 /p0	LZS	30	2314985	1.525	13	271.64	178076
NSWP3 1.020	RLE-Huffman	336	2339012	1.510	46	76.77	50848
SPLAY 1.00	Splay-tree	8	2426472	1.455	71	49.74	34176
RLE 1.00	RLE-only	86	3157471	1.118	5	706.27	631494
Original input file			3531333	1.000			

Compression program is the program used to evaluate the particular compression method. Notice that the same algorithm implemented by different programs can produce different results. This usually reflects the programmer's choice of string search algorithm.

Method is the generic compression method applied to the input file.

Max ratio is the best-case compression ratio of the algorithm expressed. It was determined by compressing a 1,000,000 byte file filled with hex 00 characters.

Output size is the total output file size, including any header information required to correctly decompress the file.

Time is the time required to compress the given file. In all cases, decompression time was significantly better than compression time, and is therefore not given in this table.

K/sec is the Kilobytes per second rate of the algorithm, as seen from the input side (the ratio of the input file size and the execution time).

RGF is the Rand Goodness Factor for the particular algorithm. Higher numbers are better, and (coincidentally) illustrate the expected bytes-per-second of output bandwidth achievable with each implementation. For example, a RGF of 4,146 would be an excellent choice for a 19,200 bps async modem channel, but not very effective on a T1 (throughput would be lower than without compression, on this test data). The RGF is not a cure-all number; the compression ratio, the destination link bandwidth, and the input data set must be considered.

All of these algorithms, in their current form (most of them are implemented in C or Pascal) are capable of performing well on a <64Kbps link. The slower algorithms (LHA) will show an adequate-to-very good compression ratio. Faster links are a problem, however. Few of the algorithms in their current form are adequate for medium to high speed links (>1Mbps, <10Mbps).

3. Link issues

For any of these compression algorithms to be implemented, a reliable link between the source and destination must be implemented. This is required due to the fact that even a one bit change in the compressed data stream may affect hundreds or thousands of bytes in the decompressed data stream.

There are many forms of reliable data delivery systems available. I suggest that the best one for the task at hand is the LAPB (Link Access Protocol B) standard. This protocol has low overhead, known robustness, and is quite easy to implement. It is also the protocol under review by the PPP compression group - though not accepted as a standard at this time.

In order to maximize the bandwidth of the link, and to allow for dynamic and transparent on-demand allocation of additional dial-up links (including load sharing), the upper-level packet structure will NOT be maintained through to the data link level. Since a packet may (potentially) compress to just a few bits, it is worthwhile to re-packetize the data. This offers better line utilization (due to larger 2K packets), lower per-packet overhead, with no increase in latency.

By observing the ‘backlog’ of packets outstanding to send on a link, line utilization figures can be obtained. When the line utilization is at (for example) 150%, a new link can be established to the target system. As more and more links are established to the target system, the bandwidth increases linearly. By watching the highest-numbered line (most recently established) traffic, that link can be closed when the traffic through it falls below some threshold level. Notice as well that latency is reduced somewhat as additional links are added. Links that are dropped due to line failure need not be re-established, unless they are the only link to the target system. Outbound traffic is always sent starting at the lowest numbered line (least recently established), and traffic is sent through additional lines only if warranted.

The following figure shows a single link in action

Figure 2. Data Compressor Diagram

(not included)

As can be seen, the data compressor resides in the TSM. This allows the TSM to open and establish additional links to the target system by connecting to additional WHSM’s.

Data will be sent to the compressor in the form of a two-byte (network order) length (of data in packet), the contents of the packet, and a CRC-16 as found in RFC-1331. The cost of computing this CRC is less than 0.5 milliseconds for a 1.5K input packet. The addition of the CRC-16 permits the compress/decompress pair to know when they are out of synchronization, and force a reset operation. It also allows for errors that may occur between the WHSM and the TSM.

LAPB will be performed with CRC-32, as found in the LAPB specification. This ensures the maximum reliability in data transmission between source and destination, with a very slight (<0.1 %) increase in data size. In the case of synchronous data transmission, it will also offer faster throughput due to software calculation of the CRC. In the current hardware, it will provide almost a 2% increase in speed.

3.1 CRC 16/CRC 32 Computation Overhead

CRC 16 can also be referred to as the HDLC FCS, or more properly, CRC-CCITT. As described in RFC-

1331, this expresses the polynomial (omitted). CRC 16 is useful for catching all single bit, all two bit, most burst errors of less than 16 bits, and some larger errors in data streams of less than 32,767 bits in length (4K byte frames), with a failure rate of 0.0015%. Multi-bit errors drastically reduce its effectiveness.

CRC 32 catches all single bit, two bit, and burst errors less than 31 bits. In packets of less than 2,147,483,648 bits (256 Mbytes), it catches larger errors with a failure rate of 0.0000000023%. It expresses the polynomial (omitted). CRC 32 is used commonly in Ethernet applications, and (while it looks more complex) is just as easy to compute as CRC 16.

To compute CRC 16, one initializes a table of possible results of each of the 256 possible input states, initializes a variable to 0xFFFF, and iterates over the data with the following code:

```
fcs = (fcs >> 8) ^ fcstab[(fcs ^ DATA_BYTE) & 0xff];
```

When all of the data has been fed through the CRC generator, the variable fcs will contain the CRC 16 value for the code. It is normal practice to complement this, and append it to the data. In the case of CRC 16, when this is done the appended (complemented) CRC can also be fed through the CRC generator. If the result of this is 0xF0B8, the frame is good. The CRC can also be compared to the CRC in the stream, and not run through the CRC generator (if desired).

An implementation of this in 8086 assembler would be as follows:

```
; AL has new byte to be computed
;
                mov         cx,fcs            ; get current FCS value
                mov         bl,cl             ; get lsb of fcs
                xor         bl,al             ; xor in current data byte
                xor         bh,bh             ; value is in the range 00-FF
                shl         bx,1              ; multiply bx by 2 to get word index
                mov         ax,fcstab[bx]     ; get new value from table
                xor         al,ch             ; xor old fcs value
                mov         fcs,ax            ; save current fcs
```

In theory, this code should run in (1+1+1+1+1+1+1 = 7) 7 clocks on a 486, for a throughput of 4.7 megabytes per second (or about 325 microseconds for a 1.5Kbyte packet) on a 33 MHz 486 system. On an 8088, this routine takes (19 +2+2+2+2+22+19 = 68) 68 clocks, for a throughput of 70 Kbytes/sec (or 22 milliseconds for a 1.5Kbytes packet). Note that in the case of the 486, the throughput exceeds that of the ISA bus. Of course, this can be optimized - placing the FCS in a register will raise the throughput to 6.6 Meg/sec; using 32 bit instructions will raise throughput to 8.25 Meg/sec.

To compute CRC 32, one initializes a table of possible results of each of the 256 possible input states, initializes a variable to 0xFFFFFFFF, and iterates over the data with the following code:

```
fcs = (fcs >> 8) ^ fcstab32[(fcs ^ DATA_BYTE) & 0xff];
```

When all the data has been fed through the CRC generator, the variable fcs will contain the CRC 32 value for the code. It is normal practice to complement this, and append it to the data. In the case of CRC 32, when this is done the appended (complemented) CRC can also be fed through the CRC generator. If the result of this is 0xDEBB20E3, the frame is good. The CRC can also be compared to the CRC in the stream, and not run through the CRC generator (if desired).

An implementation of this in 8086 assembler would be as follows:

```
; AL has new byte to be computed
;
                mov         cx,fcs              ; get lsw of current FCS value
                mov         dx,fcs+2            ; get msw of FCS
                mov         bl,cl               ; get lsb of fcs
                mov         cl,ch               ; shift FCS right 8 bits
                mov         ch,dl               ; dh has msb
                xor         bl,al               ; xor in current data byte
                xor         bh,bh               ; value is in the range 00-FF
                shl         bx,1                ; multiply by 4 as
                shl         bx,1                ; each entry is 1 doubleword
                xor         cx,fcstab[bx]       ; xor table with old fcs value
                mov         fcs,cx              ; save current fcs
                mov         ax,fcstab+2[bx] ; get msw of fcstable
                xor         al,dh               ; xor with old fcs value
                mov         fcs+2,ax            ; save result
```

In theory, this code should run in (1+1+1+1+1+1+1+1+1+1+1+1+1+1 = 14) 14 clocks on a 486, for a throughput of 2.4 megabytes per second (or about 651 microseconds for a 1.5Kbyte packet) on a 33 MHz 486 system. On an 8088, this routine takes (19+19+2+2+2+2+2+2+2+22+19+22+2+19 = 136) 136 clocks, for a throughput of 35 Kbytes/sec (or 44 milliseconds for a 1.5Kbytes packet). Of course, this can be optimized - placing the FCS in a register will raise the throughput to 3.3 Meg/sec; using 32 bit instructions will raise throughput to 8.25 Meg/sec.

4. Recommendations

I suggest that at least three algorithms be used to perform the compression. The choice of the algorithm should be made based on the aggregate speed of the data link to the target system, with override capability by a configuration option. The following table illustrates the recommendation.

TABLE 3. Compression recommentations

Line speed	Compression algorithm	Compression ratio	Effective speed / Max speed
<= 64Kbps	LZSSA	2.721	174 Kbps / 58 Mbps
<= 2 Mbps	LZW (max cw 16 bits)	2.235	4.5 Mbps / 1100 Mbps
<=10 Mbps	RLE	1.668	16.7 Mbps / 80 Mbps

The maximum speed is the theoretical maximum compression ratio for the algorithms specified. It is extremely unlikely that these speeds would ever be seen in practice. The effective speed shows the speed that could be obtained running at the maximum specified line speed, with the sample data stream detailed in the method section of this document. Real, effective speeds will (of course) depend on the data set. Fully random data will achieve effective line speeds of approximately 90% of available bandwidth (ignoring hardware limitations).

5. Background Material

The following material was written by Haruhiko Okumura, and gives some background on each of the various compression algorithms.

Data Compression Algorithms of LARC and LHarc

5.1 Introduction

In the spring of 1988, I wrote a very simple data compression program named LZSS in C language, and uploaded it to the Science SIG (forum) of PC-VAN, Japan’s biggest personal computer network.

That program was based on Storer and Szymanski’s slightly modified version of one of Lempel and Ziv’s algorithms. Despite its simplicity, for most files its compression outperformed the archivers then widely used. Kazuhiko Miki rewrote my LZSS in Turbo Pascal and assembly language, and soon made it evolve into a complete archiver, which he named LARC.

The first versions of LZSS and LARC were rather slow. So I rewrote my LZSS using a binary tree, and so did Miki. Although LARC’s encoding was slower than the fastest archiver available, its decoding was quite fast, and its algorithm was so simple that even self-extracting files (compressed files plus decoder) it created were usually smaller than non-self-extracting files from other archivers.

Soon many hobby programmers joined the archiver project at the forum. Very many suggestions were made, and LARC was revised again and again. By the summer of 1988, LARC’s speed and compression have improved so much that LARC-compressed programs were beginning to be uploaded in many forums of PC-VAN and other networks.

In that summer I wrote another program, LZARI, which combined the LZSS algorithm with adaptive arithmetic compression. Although it was slower than LZSS, its compression performance was amazing. Miki, the author of LARC, uploaded LZARI to NIFTY-Serve, another big information network in Japan. In NIFTY-Serve, Haruyasu Yoshizaki replaced LZARI’s adaptive arithmetic coding with a version of adaptive Huffman coding to increase speed. Based on this algorithm, which he called LZHUF, he developed yet another archiver, LHarc.

In what follows, I will review several of these algorithms and supply simplified codes in C language.

5.2 Simple coding methods

Replacing several (usually 8 or 4) "space" characters by one "tab" character is a very primitive method for data compression. Another simple method is run-length coding, which encodes the message "AAABBBBAACCCC" into "3A4B2A4C", for example.

5.2.1 LZSS coding

This scheme is initiated by Ziv and Lempel [1]. A slightly modified version is described by Storer and Szymanski [2]. An implementation using a binary tree is proposed by Bell [3]. The algorithm is quite simple: Keep a ring buffer, which initially contains "space" characters only. Read several letters from the file to the buffer. Then search the buffer for the longest string that matches the letters just read, and send its length and position in the buffer. If the buffer size is 4096 bytes, the position can be encoded in 12 bits. If we represent the match length in four bits, the <position, length> pair is two bytes long. If the longest match is no more than two characters, then we send just one character without encoding, and restart the process with the next letter. We must send one extra bit each time to tell the decoder whether we are sending a <position, length> pair or an unencoded character.

The accompanying file LZSS.C is a version of this algorithm. This implementation uses multiple binary trees to speed up the search for the longest match. All the programs in this article are written in draft-proposed ANSI C. I tested them with Turbo C 2.0.

5.2.2 LZW coding

This scheme was devised by Ziv and Lempel [4], and modified by Welch [5]. The LZW coding has been adopted by most of the existing archivers, such as ARC and PKZIP. The algorithm can be made relatively fast, and is suitable for hardware implementation as well.

The algorithm can be outlined as follows: Prepare a table that can contain several thousand items. Initially register in its 0th through 255th positions the usual 256 characters. Read several letters from the file to be encoded, and search the table for the longest match. Suppose the longest match is given by the string "ABC". Send the position of "ABC" in the table. Read the next character from the file. If it is "D", then register a new string "ABCD" in the table, and restart the process with the letter "D". If the table becomes full, discard the oldest item or, preferably, the least used. A Pascal program for this algorithm is given in Storer’s book [6].

5.2.3 Huffman coding

Classical Huffman coding was invented by Huffman [7]. A fairly readable account is given in Sedgewick [8].

Suppose the text to be encoded is "ABABACA", with four A’s, two B’s, and a C. We represent this situation as follows:

```
4                 2    1
|                 |    |
A                 B    C
```

Combine the least frequent two characters into one, resulting in the new frequency 2 + 1 = 3:

```
4                   3
|                  / \
A                 B   C
```

Repeat the above step until the whole characters combine into a tree:

```
     7
    / \
   /   3
  /   / \
 A   B   C
```

Start at the top ("root") of this encoding tree, and travel to the character you want to encode. If you go left, send a "0"; otherwise send a "1". Thus, "A" is encoded by "0", "B" by "10", "C" by "11". Altogether, "ABABACA" will be encoded into ten bits, "0100100110".

To decode this code, the decoder must know the encoding tree, which must be sent separately.

A modification to this classical Huffman coding is the adaptive, or dynamic, Huffman coding. See, e.g., Gallager [9]. In this method, the encoder and the decoder processes the first letter of the text as if the frequency of each character in the file were one, say. After the first letter has been processed, both parties increment the frequency of that character by one. For example, if the first letter is ’C’, then freq[’C’] becomes two, whereas every other frequencies are still one. Then the both parties modify the encoding tree accordingly. Then the second letter will be encoded and decoded, and so on.

5.2.4 Arithmetic coding

The original concept of arithmetic coding is proposed by P. Elias. An implementation in C language is described by Witten and others [10].

Although the Huffman coding is optimal if each character must be encoded into a fixed (integer) number of bits, arithmetic coding wins if no such restriction is made. As an example we shall encode "AABA" using arithmetic coding. For simplicity suppose we know beforehand that the probabilities for "A" and "B" to appear in the text are 3/4 and 1/4, respectively.

Initially, consider an interval:

```
0 <=  x  < 1.
```

Since the first character is "A" whose probability is 3/4, we shrink the interval to the lower 3/4:

```
0 <=  x  < 3/4.
```

The next character is "A" again, so we take the lower 3/4:

```
0 <=  x  < 9/16.
```

Next comes "B" whose probability is 1/4, so we take the upper 1/4:

```
27/64 <=  x  < 9/16,
```

because "B" is the second element in our alphabet, {A, B}. The last character is "A" and the interval is

```
27/64 <=  x  < 135/256,
```

which can be written in binary notation

```
0.011011 <= x  < 0.10000111.
```

Choose from this interval any number that can be represented in fewest bits, say 0.1, and send the bits to the right of "0."; in this case we send only one bit, "1". Thus we have encoded four letters into one bit! With the Huffman coding, four letters could not be encoded into less than four bits.

To decode the code "1", we just reverse the process: First, we supply the "0." to the right of the received code "1", resulting in "0.1" in binary notation, or 1/2. Since this number is in the first 3/4 of the initial interval 0 <= x < 1, the first character must be "A". Shrink the interval into the lower 3/4. In this new interval, the number 1/2 lies in the lower 3/4 part, so the second character is again "A", and so on. The number of letters in the original file must be sent separately (or a special ’EOF’ character must be appended at the end of the file).

The algorithm described above requires that both the sender and receiver know the probability distribution for the characters. The adaptive version of the algorithm removes this restriction by first supposing uniform or any agreed-upon distribution of characters that approximates the true distribution, and then updating the distribution after each character is sent and received.

5.2.5 LZARI

In each step the LZSS algorithm sends either a character or a <position, length> pair. Among these, perhaps character "e" appears more frequently than "x", and a <position, length> pair of length 3 might be commoner than one of length 18, say. Thus, if we encode the more frequent in fewer bits and the less frequent in more bits, the total length of the encoded text will be diminished. This consideration suggests that we use Huffman or arithmetic coding, preferably of adaptive kind, along with LZSS.

This is easier said than done, because there are many possible <position, length> combinations. Adaptive compression must keep running statistics of frequency distribution. Too many items make statistics unreliable. What follows is not even an approximate solution to the problem posed above, but anyway this was what I did in the summer of 1988.

I extended the character set from 256 to three-hundred or so in size, and let characters 0 through 255 be the usual 8-bit characters, whereas characters 253 + n represent that what follows is a position of string of length n, where n = 3, 4 , ... These extended set of characters will be encoded with adaptive arithmetic compression. I also observed that longest-match strings tend to be the ones that were read relatively recently. Therefore, recent positions should be encoded into fewer bits. Since 4096 positions are too many to encode adaptively, I fixed the probability distribution of the positions "by hand." The distribution function given in the accompanying LZARI.C is rather tentative; it is not based on thorough experimentation. In retrospect, I could encode adaptively the most significant 6 bits, say, or perhaps by some more ingenious method adapt the parameters of the distribution function to the running statistics.

At any rate, the present version of LZARI treats the positions rather separately, so that the overall compression is by no means optimal. Furthermore, the string length threshold above which strings are coded into <position, length> pairs is fixed, but logically its value must change according to the length of the <position, length> pair we would get.

5.2.6 LZHUF

LZHUF, the algorithm of Haruyasu Yoshizaki’s archiver LHarc, replaces LZARI’s adaptive arithmetic coding with adaptive Huffman. LZHUF encodes the most significant 6 bits of the position in its 4096-byte buffer by table lookup. More recent, and hence more probable, positions are coded in less bits. On the other hand, the remaining 6 bits are sent verbatim. Because Huffman coding encodes each letter into a fixed number of bits, table lookup can be easily implemented. Though theoretically Huffman cannot exceed arithmetic compression, the difference is very slight, and LZHUF is fairly fast.

The accompanying file LZHUF.C was written by Yoshizaki. I translated the comments into English and made a few trivial changes to make it conform to the ANSI C standard.

5.3 References

[1] J. Ziv and A. Lempel, IEEE Trans. IT-23, 337-343 (1977).
[2] J. A. Storer and T. G. Szymanski, J. ACM, 29, 928-951 (1982).

[3] T. C. Bell, IEEE Trans. COM-34, 1176-1182 (1986).
[4] J. Ziv and A. Lempel, IEEE Trans. IT-24, 530-536 (1978).
[5] T. A. Welch, Computer, 17, No.6, 8-19 (1984).
[6] J. A. Storer, Data Compression: Methods and Theory (Computer Science Press, 1988).
[7] D. A. Huffman, Porch IRE 40, 1098-1101 (1952).
[8] R. Sedgewick, Algorithms, 2nd ed. (Addison-Wesley, 1988).
[9] R. G. Gallager, IEEE Trans. IT-24, 668-674 (1978).
[10] I. E. Witten, R. M. Neal, and J. G. Cleary, Commun. ACM 30, 520-540 (1987).

6. Legal Issues

Regretfully, no compression algorithms are free of patent contention. The following items come from the comp.compression frequently asked questions list. I make no claims as to the validity of any of the patent issues, nor of any of the implementations under investigation.

- Run length encoding
  Tsukiyama has two patents on run length encoding: 4,586,027 and 4,872,009 granted in 1986 and 1989 respectively. The first one covers run length encoding in its most primitive form: a length byte followed by the repeated byte. The second patent covers the ’invention’ of limiting the run length to 16 bytes and thus the encoding of the length on 4 bits. Here is the start of claim 1 of patent 4,872,009, just for pleasure:

  1. A method of transforming an input data string comprising a plurality of data bytes, said plurality including portions of a plurality of consecutive data bytes identical to one another, wherein said data bytes may be of a plurality of types, each type representing different information, said method comprising the steps of: [...]

- LZ77
  The Gibson & Graybill patent 5,049,881 covers the LZRW1 algorithm previously discovered by Ross Williams. Claims 4 and 12 are very general and could be interpreted as applying to any LZ algorithm using hashing (including all variants of LZ78):

  4. A compression method for compressing a stream of input data into a compressed stream of output data based on a minimum number of characters in each input data string to be compressed, said compression method comprising the creation of a hash table, hashing each occurrence of a string of input data and subsequently searching for identical strings of input data and if such an identical string of input data is located whose string size is at least equal to the minimum compression size selected, compressing the second and all subsequent occurrences of such identical string of data, if a string of data is located which does not match to a previously compressed string of data, storing such data as uncompressed data, and for each input strings after each hash is used to find a possible previous match location of the string, the location of the string is stored in the hash table, thereby using the previously processed data to act as a compression dictionary.

  Claim 12 is identical, with ’method’ replaced with ’apparatus’. Since the ’minimal compression size’ can be as small as 2, the claim could cover any dictionary technique of the LZ family. However the text of the patent and the other claims make clear that the patent should cover the LZRW1 algorithm only.

  The following papers, published before the patent was filed, describe applications of hashing to LZ77 compression:

  Brent, R.P. "A Linear Algorithm for Data Compression", Australian Computer Journal, Vol.19, No.2 (May 1987), p.64.

  Bell, T. "Longest match string searching for Ziv-Lempel compression" Res. Rept. 6/89, Dept. of Computer Science, Univ. of Canterbury, New Zealand (Feb 89).

  Phil Katz, author of pkzip, also has a patent on LZ77 (5,051,745) but the claims only apply to sorted hash tables, and when the hash table is substantially smaller than the window size.

Robert Jung, author of 'arj', has recently been granted patent 5,140,321 for one variation of LZ77 with hashing. This patent covers the LZRW3-A algorithm, also previously discovered by Ross Williams. LZRW3-A was posted on comp.compression on July 15, 1991. The patent was filed two months later on Sept 4, 1991. (The US patent system allows this because of the 'invention date' rule.)

Fiala and Greene obtained in 1990 a patent (4,906,991) on all implementations of LZ77 using a tree data structure. Claim 1 of the patent is much broader than the algorithms published by Fiala and Greene in Comm.ACM, April 89. The patent covers the algorithm published by Rodeh and Pratt in 1981 (J. of the ACM, vol 28, no 1, pp 16-24). It also covers the algorithm previously patented by Eastman-Lempel-Ziv (4,464,650), and the algorithms used in lharc, lha and zoo.

IBM patented (5,001,478) the idea of combining a history buffer (the LZ77 technique) and a lexicon (as in LZ78).

- LZ78
  The LZW algorithm used in 'compress' is patented by IBM (4,814,746) and Unisys (4,558,302). It is also used in the V.42bis compression standard and in Postscript Level 2. (Unisys sells the license to modem manufacturers for a onetime $25,000 fee.) The IBM patent application was filed three weeks before that of Unisys, but the US patent office failed to recognize that they covered the same algorithm. (The IBM patent is more general, but its claim 7 is exactly LZW.)

  AP coding is patented by Storer (4,876,541).

- Other data compression algorithms
  IBM holds a patent on the Q-coder implementation of arithmetic coding. The arithmetic coding option of the JPEG standard requires use of the patented algorithm. No JPEG-compatible method is possible without infringing the patent, because what IBM actually claims rights to is the underlying probability model (the heart of an arithmetic coder).

  Bacon has patented (4,612,532) some form of Markov modeling.

As can be seen from the above list, all the most popular compression programs (compress, pkzip, zoo, lha, arj) are now covered by patents. (This says nothing about the validity of these patents.)

Here are some references on data compression patents. A number of them are taken from the list maintained by Michael Ernst <mernst@theory.lcs.mit.edu> in mintaka.lcs.mit.edu:/mitlpf/ai/patent-list (or patent-list.Z).

4.464.650
Apparatus and method for compressing data signals and restoring the compressed data signals. Inventors Lempel, Ziv, Cohn, Eastman. Assignees Sperry Corporation and AT&T Bell Laboratories. Filed 8/10/81, granted 8/7/84

4,558,302
High speed data compression and decompression apparatus and method. Inventor Welch. Assignee Sperry Corporation (now Unisys). Filed 6/20/83, granted 12/10/85. The text for this patent can be ftped from rusmv1.rus.uni-stuttgart.de (129.69.1.12) in /info/comp.patents/US4558302.Z.

4,586,027
Method and system for data compression and restoration. Assignee Hitachi, inventor Tsukimaya et al. Filed

08/07/84, granted 04/29/86

4,612,532
Inventor Bacon. Granted 9/1986

4,814,746
Data compression method. Inventors Victor S. Miller, Mark N. Wegman. Assignee IBM. Filed 8/11/86, granted 3/21/89. A previous application was filed on 6/1/83, three weeks before the application by Welch (4,558,302)

4,872,009
Method and apparatus for data compression and restoration. Assignee Hitachi, inventor Tsukimaya et al. Filed 12/07/87, granted 10/03/89.

4,876,541
Stem [sic] for dynamically compressing and decompressing electronic data. Inventor James A. Storer. Assignee Data Compression Corporation. Filed 10/15/87, granted 10/24/89

4,955,066
Compressing and Decompressing Text Files. Inventor Notenboom, L.A. Assignee Microsoft. Filed 10/13/89, granted 09/04/90.

5,001,478
Method of Encoding Compressed Data. Filed 12/28/89, granted 03/19/91. Inventor Michael E. Nagy. Assignee IBM.

5,049,881
Apparatus and method for very high data rate-compression incorporating lossless data compression and expansion utilizing a hashing technique. Inventors Dean K. Gibson, Mark D. Graybill. Assignee Intersecting Concepts, Inc. Filed 6/18/90, granted 9/17/91.

5,051,745
String searcher, and compressor using same. Inventor Phillip W. Katz (author of pkzip). Filed 8/21/90, granted 9/24/91.

4,906,991
Textual substitution data compression with finite length search window. Inventors Fiala,E.R., and Greene,D.H. Filed 4/29/1988, granted 3/6/1990. Assignee Xerox Corporation.

5,109,433
Compressing and decompressing text files. Assignee Microsoft.

5,140,321
Data compression/decompression method and apparatus. Filed 9/4/91, granted 8/18/92. Inventor Robert Jung. Assignee Prime Computer

Abbreviations for the compression method are: Huffman - Dynamic Huffman coding; RLE - Run Length Encoding; LZW - Lempel-Ziv-Welch dictionary coding; LZH - Lempel-Ziv static Huffman coding; LZSF - Lempel-Ziv with Shannon-Fano tree coding; LZAH - Lempel-Ziv with Adaptive Huffman coding; LZA - Lempel- Ziv with Arithmetic coding; LZSS - Lempel-Ziv, as modified by Storer and Szymanski; LZSSA - LZSS with adaptive coding.

Time to run algorithm on an AST 486/33, excluding file I/O time.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Data Compression for Wide Area Networks

Revision 1.01   5/18/93

Dave Rand
Novell, Inc.
San Jose, California