A Fast Data Compression Method

by

Samuel T. Chanson & Jee Fung Pang

Dept. of Computer Science,
University of British Columbia,
Vancouver, B.C., Canada V6T 1W5.

TR 83-10 (August 1983)

ABSTRACT

This paper presents a new data compression scheme. The scheme uses both fixed and variable length codes and gives a compression ratio of about one-third for English text and program source files (without leading and trailing blank suppression). This is very respectable compared to existing schemes. The compression ratio for numbers ranges from 52% for numbers in scientific notation to 65 % for integers. The major advantage of the scheme is its simplicity. The scheme is at least six times faster than Huffman's code and takes about half the main memory space to execute.

Acknowledgement

This work was supported in part by the Canadian Natural Sciences and Engineering Research Council under grant No. 3554.

1. Introduction and Motivation

Data compression techniques have often been used to counter the ever growing demand for mass storage by effectively increasing the storage capacity of computer systems. This is especially important for small systems where storage capacity is more restrictive. More recently, the need to move large amount of data over computer communication networks give added significance to the usefulness of compressing data. The cost of sending huge volumes of information via satellites, long distance telephone lines or cables can be reduced by first compressing the data to be sent and then expanding the compressed data to their original form at the receiving end. As compression requires encoding the data, it also provides a measure of data security.

The Department of Medical Genetics at the University of British Columbia operates a VAX 11/750 running 4.1BSD UNIX. Large volumes of data, program source as well as electronic mail messages have to be regularly exchanged with other research centres. As well, to cope with the small disk capacity available, many files have to be frequently compressed and expanded. An efficient data compression algorithm which requires minimal CPU time and main memory space and which gives reasonable data reduction ratio is therefore very desirable.

The files requiring compression can be classified into three categories:

1. English text files, including electronic mail messages.
2. Files containing mostly numbers (both integers and real numbers).
3. Source codes (Fortran, Cobol, C, Pascal etc.).

Except for source codes, most files do not contain many contiguous blanks. As well, trailing blanks on each line are easily and almost always removed after the files are created so that leading and trailing blank suppressions by themselves do not save much space for our applications.

Many existing algorithms have been studied [1-8]. As well, 4.1 BSD UNIX provides a Huffman-like [3] compression routine. However, most methods require larger overhead than we would like. Our algorithm employs both the variable and fixed length codes. The most important advantage of the scheme is its simplicity. Encoding and decoding are straightforward and require very little CPU time. As well, the amount of main memory space required is less than that for most existing schemes.

2. The Compression Process

Like most existing compression algorithms, our scheme requires two passes over the text to be compressed. However, because a large portion of the characters are assigned fixed length codes, the algorithm is less sensitive to code assignment than most existing algorithms and can be reduced to a single pass with only slight performance degradation (see below). The first pass counts the frequencies of all characters appearing in the text and sorts them in decreasing order. The v most frequently used characters (Group A) are assigned variable length codes. The next f characters (Group B) are assigned fixed length codes. The remaining characters (Group C) are again given variable length codes. The format of the variable length codes (Groups A and C) is n 1's followed by a 0, where n=1,2,3... The fixed length codes (Group B) are (v+ 1) bits long and start with a 0. Since the codes in this group have the same length, code assignment within the group has no effect on the compression ratio. For English text files (and also for program source files) v is chosen to be 5. The fixed length codes are therefore 6 bits long and there can be up to 32 characters in Group B.

The number of characters selected for each group is not arbitrary. From a collection of mostly electronic mail messages of English text consisting of over 140,000 characters (including punctuation marks and spaces but with no distinction between upper and lower cases (more on this latter)), the cumulative occurrences of the top j most frequently used characters, expressed as a percentage of the entire text is given in Table 2.1 below.

Top 5 characters	36.7%	Group A
Next 32 characters	60.0%	Group B
Next 2 characters	0.8%	Group C
the rest	2.5%	

Table 2.1 Percentage of occurrence of characters in file.

As can be seen, for v=5, about 96.7% of the characters can be expressed in 6 bits or less. Characters in Group B represent a saving of 25% over the standard 8-bit ASCII or EBCDIC codes. Those in Group A will give even more savings whereas with the exception of the two most frequently used characters, more than 8 bits will be required to encode the characters in Group C (about 2.5% of the text).

Reducing the length of the fixed length codes will push more characters into Group C whereas increasing it beyond 6 bits do not result in much space saving (7-bit code gives only a 12.5% reduction and an 8-bit code offers on reduction at all).

For files containing only numbers, the total number of distinct characters is small, usually between 11 and 15. In this case, performance will improve if the number of characters in Group A (i.e., v) is set to 3, so that the fixed length codes for Group B characters will consist of only 4 bits. Groups A and B cover 11 characters (sufficient for all of the decimal digits) and no code will be longer than 8 bits. It is obvious (see section 5) that the compression ratio will always be better than the 50% provided by BCD codes.

If the total number of distinct characters exceeds 15 but is less than 25 or so (which is uncommon for most files) then v=4 is best.

This scheme will work well even if the total number of different characters is small and when the frequencies of character occurrence is fairly even. It will not work well if the total number of different characters is very large which is not the case for the types of files under consideration.

The encoding scheme is very simple and may follow the method described below.

After sorting the frequencies of occurrence of the characters in decreasing order, an array consisting of 256 elements (for the 8-bit ASCII or EBCDIC codes) is constructed. The original 8-bit code of the character is used as an index into the array which contains the position of the character in the frequency list (i.e., 1 for the most frequently used character, 2 for the next and so on). If this value is P for a particular character CH, then

if $P <= v$ then code(CH) = P 1's followed by a 0 (A)
if $v < P <= v + 2^v$ then code(CH) = lower order $v+1$ bits
of the binary number $P-(v+1)$ (B)
if $P > v + 2^v$ then code(CH) = $(P - 2^v)$ 1's followed by 0 (C)

Notice that unlike Huffman's scheme [3], no tree construction and traversal are needed. Upper and lower case letters can be taken into account by converting all letters into either upper or lower cases and using a case shift code (e.g., the first code in Group B, i.e., (v+1) 0's) to indicate a case shift for the following letters up to the next occurrence of the case shift code. (Upper case is assumed initially). This way, the total number of characters that needs to be distinguished is kept low and performance is only marginally worse than the case where letters are all of one case (see section 5).

Notice that if the case shift character is assigned the code (v+1) 0's, then (B) should be

if $v<P<=v+2^v$ then code(CH) = lower order $v+1$ bits
of the binary number $P-v$ (B1)

A simple example of compressing the English line 'Tom goes to school.' will now be given to illustrate the algorithm. The frequency counts are given in Table 3.1 and the code assignments are shown in Table 3.2. The value of v is set to 3.

Character	o	blank	t	s	m	g	e	c	h	l	.
Frequency Count	5	3	2	2	1	1	1	1	1	1	1

Table 3.1 Frequency Counts

Group	Character	Code Assignment
A	o	10
	blank	110
	t	1110
B	case shift	0000
	s	0001
	m	0010
	g	0011
	e	0100
	c	0101
	h	0110
	l	0111
C	.	11110

Table 3.2 Code Assignment

The compressed data is **1110 0000 10 0010 110 0011 10 0100 0001 110 1110 10 110 0001 0101 0110 10 10 0111 11110.**

The compression ratio is 55%. If the letter cases are not distinguished, the compression ratio would have been 58%.

3. Decoding Process

Decoding is equally simple. The most important thing in a variable length scheme is to be able to detect the end of a character code. Two tables are passed to the decoding program together with the encoded text. Table A contains the original characters in Groups A and C. Table B contains the Group B characters. If the first bit of the code to be expanded is a 1, then the character code is terminated by 0. The number of 1's in the code is counted and is used to index into Table A to obtain the original character. If the first bit is 0, then the code is 6 bits long. The 6-bit code is then interpreted as an integer and used to index into

Table B to retrieve the original character.

When the case shift code is encountered, it may be necessary to add (or subtract) a constant to the original character code from the table to get the upper (or lower) case equivalent for letters.

4. Extensions

a) Suppressing leading and trailing blanks.

As mentioned before, our files typically do not contain many leading or trailing blanks. However, this feature can be added easily and with little additional overhead (about 7%). The performance of this addition is given in the next section.

b) Case distinctions.

For many files, no case distinction is necessary. This is for example the case with Fortran, PL/1, Cobol, Basic and some other source codes. It is obviously also unnecessary for files containing only numbers. In the case of English text, case distinction may be desirable. The use of a case shift code can handle this adequately as described in the last section. In this case, one character will be pushed from Group B into Group C. The degradation in both the overhead and compression ratio are not significant (see section 5).

c) Standard tables.

By creating standard tables for specific classes of files (e.g., 1 for English text, 1 for Fortran source etc.), it is possible to eliminate pass 1 in the compression process. In addition to reducing the CPU overhead, it also reduces i/o activities and the amount of main memory required.

In our scheme, more than half of the characters to be compressed belongs to Group B (see Table 2.1) where fixed length code assignment produces no effect on the compression ratio. It follows that this scheme will suffer less in the use of standard tables than those employing only variable length encoding (such as Hahn's and Huffman's schemes). The decrease in compression ratio was found to be only about 5% for Fortran codes (see section 5).

5. Performance

Huffman's code [3] is perhaps the most well-known data compression scheme. It is also optimal for encoding individual characters (rather than groups of characters). However, to obtain the minimal redundancy codes, much work is needed to build and to traverse trees. One of the fastest data compression schemes which gives reasonable compression ratio is the one by Hahn [2]. Both take two passes over the original data to be compressed. We shall compare our method to both these algorithms. Hahn's algorithm was implemented in C on the Vax 11/750. A one-pass Huffman compression program (using standard tables) already exists on the system.

In the case of English text compression, a large file of system manuals consisting of over 140,000 characters was used as the input data. The results are shown in Table 5.1 below.

	CPU Time (sec)	Memory Requirement (bytes)	# Disk i/o's	Reduction Ratio
Huffman's Code (1 pass)	141.1	22K	248	44.54%
Hahn's Scheme (2 passes)	45.5	12K	397	33.52%
Our Scheme (2 passes)	31.1	11K	390	31.14%
Our Scheme (2 passes, leading blank suppression)	33.4	11K	410	33.80%

Table 5.1 Performance Comparisons for English text files

Thus the new algorithm is more than four times faster than Huffman's code and uses half the main memory space in the compression of English text. It is, however, also about a third worse in compression ratio. Its performance is about the same as that of Hahn's scheme but is approximately 25% faster. The overhead for case distinction in our scheme is negligible and suppressing leading blanks adds about 7% to the CPU time.

Standard tables have been prepared for Fortran source files to eliminate the first pass in the compression process. Over 60,000 characters of Fortran codes were compressed. The results are shown in Table 5.2.

	CPU Time (sec)	Memory Requirement (bytes)	# Disk i/o's	Compression Ratio
Huffman's Code (1 pass)	76.4	19K	100	39.2%
Our Scheme (1 pass)	11.7	11K	94	32.1%
Our Scheme (2 passes)	14.2	12K	150	33.9%

Table 5.2 Performance Comparison using standard tables for Fortran Source

It is observed that using standard tables to eliminate the first pass decreases the compression ratio by only 5%. However, both the memory requirement and disk activities are reduced. The CPU time is decreased by about 17% and it is now more than six times faster than the Huffman code.

Finally, we tested Pascal source as well as number files on our compression scheme (2 passes, no leading and trailing blanks suppression). The compression ratios for various file types are listed in Table 5.3.

File type	Compression Ratio
Integers	64.5%
Signed real numbers in scientific notation	52.3%
Pascal Source	31.3%
English text	31.1%
Fortran Source	33.9%

Table 5.3 Compression ratio of our scheme for different file types

6. Conclusions

We have presented a new data compression scheme. The scheme gives a compression ratio of about one-third for English text and program source files (without leading and trailing blank suppression). This is very respectable compared to existing schemes. The compression ratio for numbers ranges from 52% for numbers in scientific notation to 65% for integers. The major advantage of the scheme is its simplicity. The scheme is at least six times faster than Huffman's code and takes about half the main memory space to execute. We believe it is the fastest scheme for the range of compression ratio it provides. The scheme can easily be implemented in hardware.

References

[1] Cortesi, D., "An effective text-compression algorithm", Byte, vol.7, no.1, Jan. 1982, pp.397-403.

[2] Hahn, B.,"A new technique for compression and storage of data", Comm.ACM, vol.17, no.8, Aug. 1974, pp.434-436.

[3] Huffman, D.A., "A method for the construction of minimum redundancy codes", Proc. IRE, vol.40, no.9, Sept. 1952, pp.1098-1101.

[4] Pechura, M., "File archival techniques using data compression", Comm.ACM, vol.25, no.9, Sept. 1982, pp.605-609.

[5] Rubin, F., "Experiments in test file compression", Comm.ACM, vol.19, no.11, Nov. 1976, pp.617-623.

[6] Ruth, S.S. and Kreutzer, P.J., "Data compression for large business files", Datamation, vol.18, no.9, Sept. 1972, pp.62-66.

[7] Tropper, R., "Binary-coded text, a text-compression method", Byte, vol.7, no.4, April 1982, pp.398-413.

[8] Wells, M., "File compression using variable length encodings", The Computer Journal, vol.15, no.4, 1972, pp.308-313.