A New Technique for Compression and Storage of Data

Bruce Hahn
University of Waterloo

The widespread tendency toward storage of large programs and blocks of text has produced a need for efficient methods of compressing and storing data. This paper describes techniques that can, in most cases, decrease storage size by a factor of from two to four. The techniques involve special handling of leading and trailing blanks, and the encoding of other symbols in groups of fixed size as unique fixed point numbers. The efficiency of the system is considered and pertinent statistics are given and compared with statistics for other information coding techniques.

Key Words and Phrases: file maintenance, information retrieval, utility programs, text compression, coding techniques, data storage, data management

CR Categories: 3.70, 3.73, 4.49

Author's address: Department of Applied Analysis and Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

Introduction

In the past several years, the growing storage requirements for large programs and large blocks of English text have created a very real need for text compression techniques which can offer significant savings of storage space. Owing to the advent of interactive systems and the increased tendency to store data on online direct-access devices, efficient data storage techniques are of interest.

There are several reasons why more efficient compression of text is possible. Firstly, text often contains records with many leading and trailing blanks which take up a great deal of space. Secondly, most computer systems use a coding scheme which allows for either 128 different possible symbols (e.g. 7-bit ASCII) or 256 possible symbols (e.g. EBCDIC). However, most text uses a relatively small percentage of all the possible symbols. According to statistics given by Reza [1], the seven most common symbols in English text (one case, not counting punctuation) account for over 60 percent of most text.

In addition, much text consists of only uppercase letters rather than both uppercase and lowercase (programming languages for instance). The net result is that much space is wasted by the usual methods of text storage.

One of the more common methods of alleviating this problem is to store only nonblank symbols. WYLBUR [4], a text-editing system, does this by storing symbols in SEGMENTS, each of which can describe up to 15 blanks followed by up to 15 nonblank symbols. Each record of text is then stored as two principal fields: the first gives the number of segments necessary to describe the record; the second is the segments themselves.

HASP [5], part of the IBM/360 operating system, uses a similar algorithm, but recognizes strings of duplicate symbols and long strings of blanks, and treats them specially.

Where blanks are important only as delimiters, other methods have been employed. The storage algorithm used by the PL/C compiler to store diagnostic messages [6] is phrase-oriented and heavily dependent upon context.

However, to date, most text compression systems of the same type as used by WYLBUR and HASP fail to store the nonblank symbols any more efficiently than the machine's internal representation. Furthermore, unlike the system described in this paper, most text compression systems are neither easy to understand and implement nor portable.

Compression

Basically, compression may be achieved in two ways: (1) trailing and leading blanks are "squeezed off" of each input record; (2) the remaining nonblank symbols are encoded in groups of a fixed size as unique fixed point numbers [2].

The numerical encoding works as follows. The nontrivial symbols are considered in groups of length N. Each of the symbols of a group is looked up in a dictionary, and the position of each symbol is noted. The symbols are encoded by their positions in the dictionary. Later when decoding takes place, the numerical positions are used in conjunction with the dictionary to produce the original text.

Let B be the number of symbols in the dictionary, and P_i be the position of the ith symbol of a group ($1 \leq i \leq N$) in the dictionary ($1 \leq P_i \leq (B - 1)$). The B'th position is reserved for an escape character to permit an extension of the size of the dictionary.

A group of symbols with positions $P_1, P_2, \ldots, P_N$ is encoded as the unique fixed point number: $P_1*B^{N-1} + P_2*B^{N-2} + \cdots + P_{N-1}*B + P_N$. In the actual implementation, the values $B^{N-1}, \ldots, B^2$ may be calculated once and stored, in order to eliminate redundant arithmetic operations. More than B different symbols may be allowed by use of the Bth position in the dictionary as an escape character. If the B'th position was not reserved, the value $B*B^i$ could appear, which could cause an ambiguity. The escape character signifies that the following symbol has a position in the range $B + 1$ to $2B - 1$. More than one escape character may be used to extend the length of the dictionary even further. The first B symbols are said to comprise the PRIMARY DICTIONARY.

Thus, in general, if p is the position of a given symbol in the dictionary, the symbol is encoded as $\lfloor p/B \rfloor$ escape characters, followed by MOD(p,B), where $\lfloor x \rfloor$ denotes the integer part of x.

The escape character is coded as a zero. Every time a zero is encountered in the decoding process, the value B is added to the next nonzero position number that is decoded, to find the true position of the coded symbol in the dictionary.

For example, say the length of the primary dictionary is 21 (i.e. $B = 21$) and the symbols are to be encoded in groups of 7 (i.e. $N = 7$). Let the symbols to be encoded have the positions 5, 7, 20, 25, 45, 6, 9, 22, 10, and 15 in the dictionary. These symbols would be encoded as two fixed point numbers: the first having the value $5*21^6 + 7*21^5 + 20*21^4 + 0*21^3 + 4*21^2 + 0*21 + 0 = 461{,}310{,}696$; and the second having the value $3*21^6 + 6*21^5 + 9*21^4 + 0*21^3 + 1*21^2 + 10*21 + 15 = 283{,}553{,}949$. Note that the symbol with position 25 is encoded as 0,4 and the symbol with position 45 is encoded as 0,0,3.

To minimize the number of escape characters that will be needed, a portion of the text is processed during a PRELIMINARY SCAN, which estimates the probable occurrence of each symbol.The dictionary is then constructed with the symbols in descending order of predicted occurrence. Both the number of characters scanned and the starting contents of the dictionary can be modified by the user.

If at any time an unknown symbol is encountered, it is dynamically added to the dictionary. For this reason the user need not concern himself with the presence of any unprintable characters in the text. However, this causes a problem because the final form of the dictionary (which is unique and is needed by the decoding process) is not known until after the processing of all the input text is complete.

The current implementation handles this problem by writing the encoded text onto a temporary file. When the encoding is complete, the dictionary is written as the first record of the output file for use by the decoding routine. The compressed text is then copied from the temporary file and concatenated onto the permanent output file.

The compressed record has the following format consisting of three fields. The first two, which are stored in the left and right halves of one computer word, contain the number of leading blanks and the number of nontrivial symbols following the last leading blank. The third field is of variable length and consists of enough words to store all the nonblank symbols in the record.

Example. (1) Using the encoding example given above, if an 80 character input record consisted of 10 blanks—the 10 symbols whose positions in the dictionary are as given above—followed by 60 blanks, the compressed record would be stored in the following manner (note that each long box represents one computer-word):

| 10 | 10 | | 461310696 | | 283553949 |

(2) A record consisting of 80 blanks would be compressed-as follows:

| 80 | 0 |

The compressing routines process an input file one record at a time. Because of this, compressed records, which are of variable length, are written onto an output buffer in a packed format, the buffer being written onto the output file when it becomes full.

Optimization of Compression

The user of this system has control over the values of the variables B (the length of the primary dictionary) and N (the size of the encoding groups). Ideally, the values of B and N are picked so as to try to minimize the average number of bits/character needed to encode the input text. For each value of N, there exists an optimum value of B. This optimum value of B is the

largest integer value of B such that $B^N - 1 \leq L$, where L is the largest fixed point number that can be stored in one word. This is because the largest possible number that can be produced is clearly $B*B^{N-1} + B*B^{N-2} + \cdots + B = B^N - 1$.

On the IBM 360, $L = 2^{31} - 1$. This gives rise to values of B corresponding to those values of N which are reasonable; see Table I. The value of B is obtained from the equation: $B = \lfloor (2^{31} - 1)^{1/N} \rfloor$. That is, B is picked to be as large as it can be, without any possible overflow, for a given value of N.

Table I.

Value of N	*Optimum value of B*
5	73
6	35
7	21
8	14

Note: any smaller values of N would pack symbols no better than the 360 already does, while any larger value of N would necessitate a value of B that would be too small to be workable.

Statistical Analysis

The statistics given in Table II were obtained for different values of B and N. Note that the Fortran text consisted of over 31,000 symbols; the English text, over 27,000 symbols; and the Assembler text, over 84,000 symbols.

It is interesting to compare these statistics with those for other techniques using the same word length. Table III is taken from Lesk [3].

Table II.

Kind of text	*Value of B*	*Value of N*	*Approx. bits/char*
Fortran	14	8	2.09
	21	7	2.01
	35	6	2.01
	73	5	2.27
Assembler	14	8	2.90
	21	7	2.83
	35	6	2.93
	73	5	3.30
English uppercase only	14	8	3.17
	21	7	3.17
	35	6	3.40
	73	5	3.90
English uppercase and lowercase	14	8	3.45
	21	7	3.38
	35	6	3.48
	73	5	3.92

Table III.

Code	*Entropy-English*	*Entropy-Fortran*
Huffman Code	4.2	5.1
Neighbour Code	4.0	4.6
Digram Code	4.2	4.7
Word Codes	4.5	5.0

A Huffman Code is a minimum redundancy code for single-symbol encoding, and can be shown to have the lowest entropy possible for straight single-symbol encoding. A Neighbour Code represents each symbol by a code which depends on the previous symbol. A Digram Code encodes symbols in digrams (two-letter groups). Word Codes use words as encoding units rather than single characters.

The low values for the entropy of the different examples given in Table II, depend in part on the special handling of leading and trailing blanks, which is not done for the codes given in Table III. If these blanks are not handled specially, the entropy for English text is about 4.7 bits/character for the method described in this paper.

Conclusions

Because of the simplicity and the machine independence of this coding scheme, a viable alternative exists for the compression of stored text. This technique is particularly suited to situations in which files are not often changed but are read frequently. Specific applications lie in the areas of data management of online files, and the archiving onto tape or cards of seldom used files.

Appendix. Sample Run-Time Statistics

The text compression system was used to compress the source file of the compressing routine. The file consisted of 417 card images, including the necessary job statements, the Fortran source code, and comments. The file was compressed to 106 card images with an approximate entropy of 2.03 bits/character. The run-time compression took approximately 4.0 sec of cpu time on an IBM 360/75.

The expansion of the compressed records took approximately 2.67 sec of cpu time.

References

1. Reza, F.M. *An Introduction to Information Theory* (Chap. 4.). McGraw-Hill, New York, 1961.
2. Hagamen, W.D., et al. Encoding verbal information as unique numbers. *IBM Systems J. 11*, 4 (1974), 273.
3. Lesk, M., Compressed text storage. Computer Science Technical Rept. #3 (Nov. 1970), Bell Telephone Laboratories, Murray Hill, N.J.
4. Fajman, R., and Borgelt, J. WYLBUR: An interactive text editing and remote job entry system. *Comm. ACM 16*, 5 (May 1973), 314–322.
5. Multileaving. The Hasp System. IBM Pub., Feb. 26, 1971, pp. 1139–1153.
6. Wagner, R.A. Common phrases and minimum-space text storage. *Comm. ACM 16*, 3 (Mar. 1973), 148.