AD-A035 786 PRC DATA SERVICES CO MCLEAN VA F/G 9/2
A REVIEW OF DATA COMPRESSION ALGORITHMS.(U)
MAY 76 C HOLBOROW, J MCNEMAR, P STONEBURNER DCA100-73-C-0015
UNCLASSIFIED CCTC-TM-122-76 NL

1 OF 2
ADA035786

C C T C

DEFENSE COMMUNICATIONS AGENCY

TECHNICAL MEMORANDUM
TM 122-76
1 MAY 1976

(2)
B.S.

COMMAND & CONTROL TECHNICAL CENTER

A REVIEW OF DATA COMPRESSION ALGORITHMS

COMMAND AND CONTROL TECHNICAL CENTER

(9) TECHNICAL MEMORANDUM 122-76

(11) 1 May 1976 (12) 152p.

(6) A REVIEW OF DATA COMPRESSION ALGORITHMS

(10) C./Holborow, J./McNemar
P./Stoneburner

(18) CCTC (19) TM-122-76

SUBMITTED BY:

R. Marion

R. Marion
Project Officer

APPROVED BY:

James A Painter

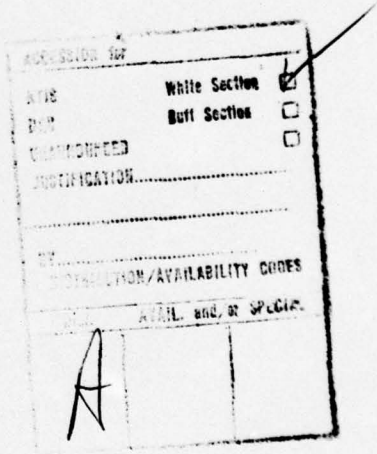
Dr. J.A. Painter
Technical Director
CCTC WAD

(15) DCA100-73-C-0015

[redacted]

ACKNOWLEDGMENT

This document was prepared by C. Holborow, J. McNemar and P. Stoneburner of Planning Research Corporation, Data Services Company, under Contract Number DCA100-73-C-0015, Task 613. Technical assistance was provided by the CCTC Project Officer, Robert A. Marion.



ABSTRACT

This memorandum describes various methods for compressing digital computer data files. The objective of the methods described is to reduce the physical space required to store data while maintaining a complete representation of the information. There are several potential benefits associated with compression. It provides more efficient use of storage devices, it improves data transfer rates (through shorter message packets) and it permits faster data base access (through greater data density per I/O storage block).

The document first discusses logical compression techniques and identifies some data base methods which minimize storage. Next, the document describes methods which achieve compression through various encoding schemes. The concepts for the development and operation of these methods are discussed, and guidance is provided for their appropriate application. Performance characteristics are delineated when operational statistics are known.

CONTENTS

CONTENTS

(Continued)

ILLUSTRATIONS

TABLES

INTRODUCTION

The Problem

Computer files tend to grow. In fact, a modified form of Parkinson's law seems appropriate: "Computer files expand to fill the storage available." Some bounds can be placed on user files by strictly limiting the space allocated to each user. However, this is not possible with a data base to which new data is constantly being added faster than old data is being deleted. The only way to contain such files within a fixed physical space is to find some way of packing the data into the available physical space more efficiently. The high cost of redesigning a data base and rewriting the programs to accommodate the new design usually makes this an impractical solution.

Recently, there has been some interest in the less drastic alternative of compressing the data. The compressed data occupies less space but is still a complete representation of the original data. The original file or parts of it can be completely reconstructed when needed. The need for "information retaining" compression clearly separates the problem from telemetry data compression, where the reconstructed data is only an approximate representation of the original data.

The interest in computer data compression has been stimulated by several factors, including:

- The increasing installation of large on-line data bases which has involved more people in the problem.
- The realization that in such systems the processor is often only lightly loaded and a major factor in determining performance is I/O time.
- The publication of several descriptions of successful compression schemes which both saved on equipment costs and yielded improved

performance due to the reduced I/O time required to transfer the compressed data.

Within the WWMCCS environment there appears to be considerable application for data compression. As well as compressing data files there is the possibility of reducing time spent in transmitting information across a network by sending a compressed version of the information.

This report is the result of an extensive literature search and contains detailed descriptions of all the compression techniques found. Only the algorithms are discussed here. Other reports describe the compression software and test results.

Organization of this Document

The compression techniques are described in the section titled Compression Techniques. The same format is used for each discussion to facilitate comparisons between the methods. For each algorithm, there are two sections: General and Detailed Description.

The General section is subdivided into:

1. Technique: The technique is classified and the method is briefly described.

2. Data Types: The type of data that the routine is designed to compress (alphanumeric, binary, text, etc.).

3. File Types: The kinds of files for which the technique appears suitable (active, index, back-up, etc.)

4. Relative Effectiveness: This section summarizes available performance data and gives rough estimates of the resources used. It compares each technique with other competing techniques and gives recommended applications.

The Detailed Description contains:

1. Algorithm: A detailed description of the algorithm.

2. Tuning: A description of how the algorithm can be tuned to optimize its performance on a specific file, where this is possible.

3. Performance Details: For a few routines, a detailed description of their performance is given. This section is used to give details of the performance figures summarized in the Relative Effectiveness section for those routines where these details would unnecessarily clutter up the Relative Effectiveness discussion.

The references for each routine are listed with the description of the routine. All these references (plus some others) are accumulated in the Bibliography.

The routines described under Compression Techniques have been grouped according to type, as is apparent from the table of contents.

The section titled Variable Length Codes describes in detail several variable length binary codes. The best known of these are Huffman codes, and the most efficient ways to generate and use these codes are described. Possible modifications to Huffman codes are described. These trade a slight loss in compression for some reduction in overhead. Gilbert-Moore alphabetic codes are also described.

COMPRESSION TECHNIQUES

Logical Compression

General

Most data, if put in a data base, in original form, tends to make very inefficient use of storage space. By reducing the physical size of the data, substantial savings in storage cost can be obtained. Also, reduced size reduces the amount of I/O time required to physically transfer data between secondary and primary memory. Since I/O time tends to be the pacing factor when processing large data bases, the lapse time for programs using the data base can often be reduced by compressing the data.

One of the first steps in designing a data base should be to provide for as much data reduction as possible in the basic design of the data. The various methods for achieving this "precompression" are called logical compression. Logical compression is composed of the myriad methods available for data reduction in the design phase. There are too many specific methods to describe in a paper of this scope, and the methods are very data dependent. Therefore, several representative techniques will be described here to identify the main concepts and thus provide the basis for implementing logical compression in a particular application.

Logical Compression Techniques

A simple example of logical compression is the use of the single character "M" or "F" in a field to indicate sex. This technique both reduces the size of the field and makes the field a fixed length. However, since the field can only be one of two choices, the size can be reduced further by allocating only a single bit to indicate sex. Thus, an on bit can indicate male and an off bit female. In order to encode and decode the sex field, a table must be created which describes the coding scheme. The table contains such information

as field name, beginning position in record, length of field, and the code (F=0, M=1).

Another field which occurs frequently in data bases is a date field. Many data bases contain more than one date per record. It is usually not practicable to insert a date into a data base in its longhand form (e.g., March 7, 1976) so usually provision is made to insert the numeric equivalents of each of the three subfields, month, day, and year (030776). This data field can be further compressed into 14 bits using a binary numbering scheme.

The minimum size of each of the subfields is $[\log_2 N]$ (where [x] is the least integer greater than or equal to x) where N is the number of values permitted in the field. For example, four bits are needed for the month and five bits each for the day and year (using 20-year span) subfields. The bit codes used could be as follows:

Month	Code	Day	Code	Year	Code
01	0000	01	00000	70	00000
02	0001	02	00001	71	00001
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
12	1011	31	11110	89	10100

The codes are concatenated in left to right order giving the appropriate date. Total field length is 14 bits. In order to extract the year, month, or day value in the date field, the appropriate subfield is isolated using AND operators and a mask for the subfield.

A faster access coding scheme can be generated for a 13-bit date field. In this case, a large compression coding table is generated. It specifically enumerates each one of the 7,305 dates that actually occur in the 20-year period. Compression and decompression are very rapid. However, this new

scheme requires a large compression table and is not amenable to readily extracting a specific month or day subfield. Thus, it would be more difficult to extract, for instance, all relative dates occurring in December of the last 5 years. With appropriate modifications, the techniques above can be used for fields other than date fields.

In cases where very large data bases are used there may be a significant amount of redundant information. For example, a file used by the Navy may have multiple fields within each record containing the name of a ship or port. The most feasible method of removing this redundancy is the use of a code for each ship or port. The code rather than the entire name of the ship is placed in the records. A table in core contains each code and the ship or port which the code represents. This technique eliminates using redundant data values in the records and the wasted space which occurs when a short data value must have spaces added to it to fill out the fixed field size.

When some data values occur much more frequently than other data values, it may be feasible to use a variable length compression code for that field. For example, consider an inventory file with a field for manufacturer. Four thousand manufacturers are specified in the inventory. If a fixed length coding is chosen, 12 bits are required to specify this field. If, however, 48 manufacturers are responsible for 80% of the items in the inventory, then 2 different field sizes are appropriate. A short 6-bit field is used to represent the 48 frequent manufacturers and a long 12-bit field is used to represent the 3,952 remaining manufacturers. To these fields must be added a single bit to indicate whether the field is short or long. Thus, the final field sizes are 7 and 13 bits respectively. However, the average field size is $7 \times .8 + 13 \times .2 = 8.2$. This is significantly less than the 12-bit fixed length field.

Variable length fields can be used in many other applications as well, yielding further reductions in space used. Name, address, and comments fields are all amenable to variable length field type compression. Field

extraction algorithms are more complicated when variable length fields are used. However, in some applications, the size reductions permitted by variable length codings may offset the field extraction cost.

Difference encoding is a method to record sequences of related numbers or dates. In this method an initial value for the date or number is set up and the data fields reflect only the difference in values. For example, if there is a field containing a date or a transaction and the earliest transaction recorded is 030755, then this date is the initial value. The encoding for the date 030855 (the day after the initial value) would simply be a 1 and 030756 would be encoded as 366 (1956 was a leap year). Recording sequential numbers can be accomplished by this technique or by averaging. In averaging, the possible numbers are averaged and this average value is the initial number for differencing.

In conclusion, there are many techniques available for logical compression which should be considered during data base design. By applying these methods a significant reduction in space, I/O time, and retrieval time can be realized, resulting in greater overall data base efficiency.

References

Alsberg, P. A.

SPACE AND TIME SAVINGS THROUGH LARGE DATA BASE COMPRESSION AND DYNAMIC RESTRUCTURING; Proc. of the IEEE, August 1975

Martin, J.

DATABASE ORGANIZATION; Prentice-Hall, 1975, Ch. 31

Mulford, J.E. and Ridall, R.K.

DATA COMPRESSION TECHNIQUES FOR ECONOMIC PROCESSING OF LARGE COMMERCIAL FILES; Proc. of the Symposium on Information Storage and Retrieval, ACM, pp. 207-215, 1971.

Fixed Length Coding For Character Strings

Character Repeat Suppression

General:

1. Technique: Character repeat suppression is a simple method of compaction which yields appreciable savings in certain cases. This technique consists of replacing a string of repeating characters with a code which describes the character string composition. The code usually consists of three characters. The first is a special character which is unused in most data samples (such as an underline or backward slash). This character indicates that this is the beginning of a character suppression code. The next character of the code is a copy of the repeating character in the data which is being suppressed. The last character of the code is the number of times the character is repeated. A binary count occupying one character position is used.

2. Data Types: Mostly used with character encoded data, but can be used with Huffman codes (see section titled Variable Length Codes) and elsewhere where advantageous.

3. File Types: Any file may use repeat suppression. It is especially useful in formatted files such as report files and program source files which are known to have many long strings of blanks.

4. Relative Effectiveness: The effectiveness of the technique is highly dependent on the type of file being compressed. In general, files which compress well with this technique also compress well with interrecord comparison techniques (see Interrecord Word Comparison (Bit Mapping)). These latter methods give slightly more compression and require less CPU time than character repeat suppression. However,

character repeat suppression is such a simple and basic technique that it should be considered for any file which is being compressed. One possibility is to suppress repeated characters and then apply other compression techniques to the resulting partially compressed output file. Core requirements are small (about 100 words) and execution is very fast, since there is little processing involved.

Detailed Description:

1. Algorithm: The input record is scanned character by character. Each character is compared with the previous character and, if the same, the repeat count is incremented. If they are different and the previous character is not part of a repeat string, the previous character is written out. If the previous character is the last character of a repeat string, the repeat string is encoded and written out. If there are two or three characters in the repeat string, the two or three repeated characters are written out. If the repeat string consists of four or more characters, a special character (rarely or never appearing in the input data) is written, followed by the repeated character and a binary count of the length of the repeated string. This count occupies one character position. If the repeated string is longer than the number of repeats that can be defined in one count, further repeat suppression strings must be written out. One detail remains and that is what to do when the repeat special character (e.g. backward slash) is encountered in the data. This problem is overcome by encoding it as a repeat of one, e.g., a backward slash is encoded as \\1. This uses three characters to encode one, but the problem should occur so rarely that the loss in compression is negligible. An alternative is to double each occurrence of the special character, e.g., a\b is encoded as a\\b.

   A variation of the algorithm may be useful on certain files where most repeated strings consist of only a few characters such as blanks and zeros. A different repeat suppression indicator is used for each of the repeated characters, and the repeat suppression string con-

sists of just two characters - the suppression indicator and a count. For example, if / indicates blank suppression, \ indicates zero suppression and @ indicates suppression of anything else, the string A␣␣␣␣␣␣␣XYYYYYYMN/P00000000 (27 characters) can be written as A/7X@Y6MN@/1P\8 (15 characters). Note that the character / in the original string is represented by @/1 in the compressed string. It could not have been replaced by //, since the second character would be interpreted as a count. Occurrences of @ in the original data could be replaced by either @@ or by @@1. The choice can be based on which substitution gives the simplest program.

2. Tuning: Tuning is not possible, apart from selecting appropriate special characters to indicate repeat suppression.

Substitution For Character Pairs

General:

1. Technique: This method (Snyderman and Hunt, 1970) makes use of the fact that, for some code sets, the number of bit codes available is a great deal larger than the number of characters in the standard character set. These unused codes are substituted for the more frequently occurring pairs of characters in a string of data. The set of actual characters used is defined to have three subsets. Master characters (MC) are used as the first character of a combined character pair, while combining characters (CC) make up the second character of the pair. The noncombining characters (NC) are always stored in their original form. Whenever a valid MC-CC character pair appears in the data string it is replaced by the unused character which is assigned to that MC-CC character pair. The pair substitution algorithm (see Detailed Description below) can easily be combined with substitution for frequently occurring 3-character and 4-character strings.

2. Data Types: This method is designed for text compression. It is based on the fact that text uses standard composition rules and spelling. Therefore, the choice of character pairs which occur frequently can be applied to almost any text file to give substantial compression, regardless of the actual subject matter of the text. This method can, however, be tuned to any type of data by changing the character pairs that are replaced.

3. File Types: This method appears to be well suited for use on active files. The compression and decompression routines are relatively fast and require little core (a few hundred words for the routines and tables) which makes it applicable to active files where overhead must be kept to a minimum. Backup text files are also suitable for this type of compression.

4. Relative Effectiveness: This method does not give quite as much compression as Huffman coding, but it executes faster. In particular, decompression is much faster than with Huffman coding. This makes it more suitable than Huffman coding for files which are read much more often than they are written.

A problem involved in this method is the generation of effective codes. Unlike Huffman coding, where the code generation procedure is well defined, the code generation process for this algorithm is mostly a process of educated guesswork, based on whatever statistics the user chooses to collect. In spite of this drawback, good codes are not difficult to find provided the user spends a little time experimenting. The compression factor achieved is normally in the range 1.5 to 1.8 for text data. It can never be better than 2 since only pairs of characters are being substituted for. It is easy to combine this algorithm with a fixed substitution for a small number of common character strings longer than two characters.

Detailed Description:

1. Algorithm: Many machines use an 8-bit code to represent data. Others (particularly machines with a 36-bit word) use a 9-bit code for ASCII data. Since there are only 128 ASCII characters, it is obvious that in such machines many codewords are unused. For files which used a subset of the ASCII character set (nearly 1/4 of the ASCII characters are used almost exclusively for communications purposes), the number of unused codewords is even greater. In this compression technique, these unused codewords are used to represent character pairs. The calculation of a substitution code for a character pair is very fast and does not involve searching a table of all pairs for which substitutions are possible.

   We define several character sets:

   L = set of characters occurring in the file
   MC = set of "master characters"
   CC = set of "combining characters"
   CP = set of all ordered pairs (MC, CC)

   MC and CC are subsets of L. They can have common members. The logic of the compression routine is slightly simpler if MC is a subset of CC, but this is not necessary. Assume there are M characters in the MC set and N characters in the CC set. We will denote the members of the MC set by MC_1, MC_2,..., MC_M. Assume there are C characters in set L.

   The algorithm assigns codes thus:

   0000 } C codewords: one per character
   C-1 }

C
C+N-1 } N codewords for the character pairs (MC_1, CC)

C+N
C+2N-1 } N codewords for the character pairs (MC_2, CC)

C+2N
.
.
C+(M-1)N

C+(M-1)N
C+MN-1 } N codewords for the character pairs (MC_M, CC)

Obviously, MN+C must not exceed the total number of codes available.

The algorithm is very simple. the input record is examined character by character. If a character is not a master character, it is translated into a single character code. (If L is the whole source alphabet, then no transliteration is necessary.) If it is a master character, the next character is examined to see if it is a combining character. If not, the first character is written out in its single character code and the second character is checked against the set of master characters. If the second character is a combining character, the MC-CC pair is encoded into a single character. The substitution code for the pair (MC_I, CC_J) is:

$$C + (I-1)\,N + J-1 \qquad 1 \le I \le M, \quad 1 \le J \le N$$

The only code tables necessary are lists of the master characters and combining characters. If transliteration of the single characters is performed, it is usually a fixed subtraction from their original code, so a code table is not required. The implementation of Snyderman and Hunt is for an IBM 360, and their set L contains 88 characters: 52 upper and lower case alphabetics, 10 numerics and 26 special and punctuation symbols. The remaining 168 codes are used to code 8 MC x 21 CC combinations. The character sets are:

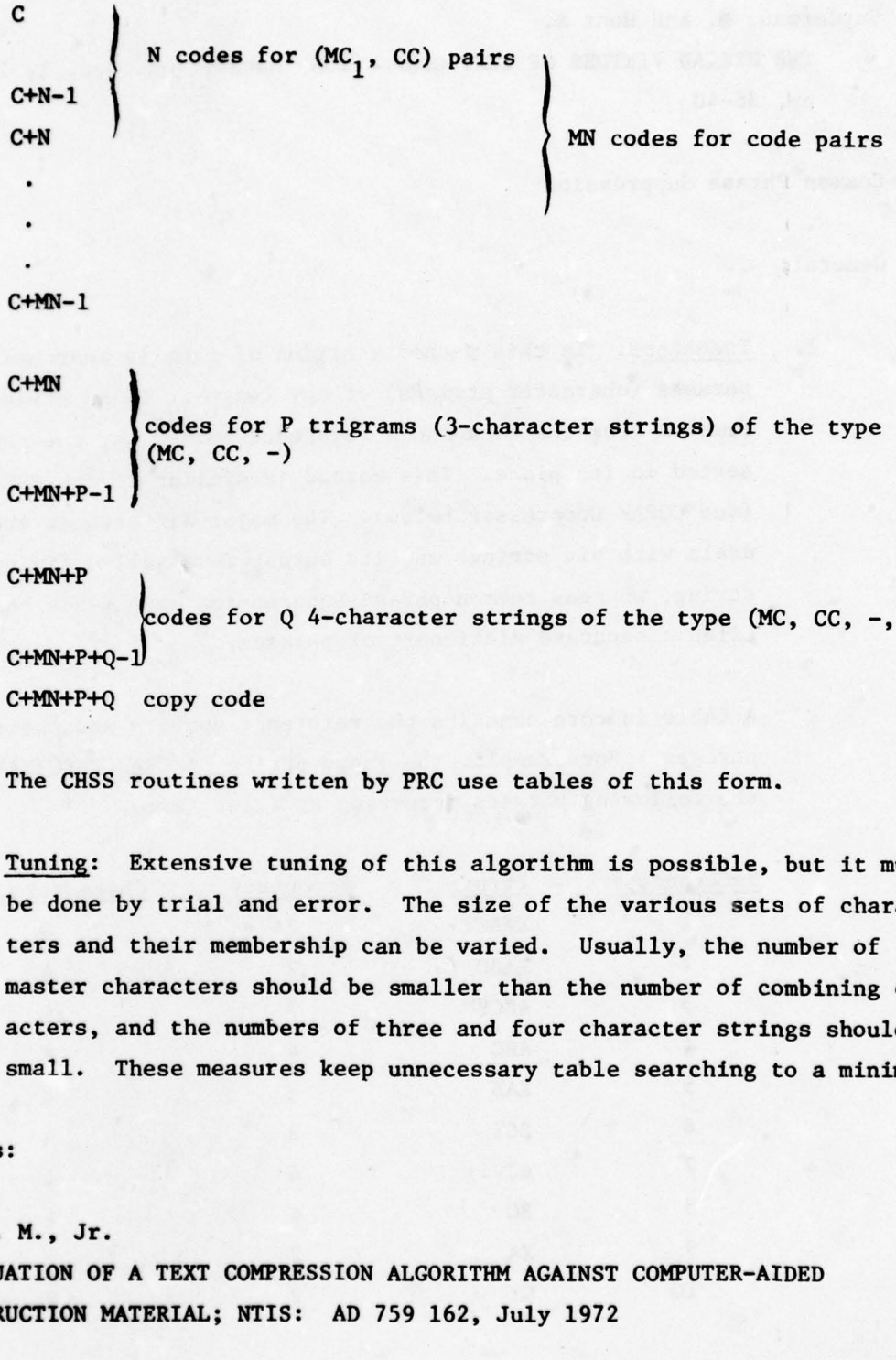
MC = space, A,E,I,O,N,T,U
CC = space, A thru I, L thru P, R thru W

It is usually desirable to have a "copy code." This is a special codeword (often the largest possible codeword) which indicates that the character following was copied as is from the source file. This preserves rare characters in the source file which are not in the set L being used.

It is simple to combine this algorithm with a table search for common 3- and 4-character strings. The substitution codes for these strings should be at the top end of the code set, and to avoid unnecessary table searching they should all begin with a (MC, CC) character pair.

The code assignment for this more complex version are:

0 ⎫
⎪ Single character codes
C-1 ⎭



The CHSS routines written by PRC use tables of this form.

2. Tuning: Extensive tuning of this algorithm is possible, but it must be done by trial and error. The size of the various sets of characters and their membership can be varied. Usually, the number of master characters should be smaller than the number of combining characters, and the numbers of three and four character strings should be small. These measures keep unnecessary table searching to a minimum.

References:

Knight, J. M., Jr.
EVALUATION OF A TEXT COMPRESSION ALGORITHM AGAINST COMPUTER-AIDED INSTRUCTION MATERIAL; NTIS: AD 759 162, July 1972

Snyderman, M. and Hunt B.

THE MYRIAD VIRTUES OF TEXT COMPACTION; Datamation, Dec. 1, 1970, pp. 36-40

Common Phrase Suppression

General:

1. Technique: In this method a string of data is searched for repeating phrases (character strings) of any length. These phrases are then removed from the data and a reference number for the phrase is inserted in its place. This method is similar to the COPAK compressor (see COPAK Compressor below). The major differences are that COPAK deals with bit strings and its output is a self-defining binary string, whereas common phrase suppression compresses character strings using a separate dictionary of phrases.

   A table in core contains the reference numbers and their associated phrases. For example, the input string 'ABCXABCYABCZXABCY' contains the following phrases occurring at least twice.

Reference #	Phrase	Frequency	Characters Saved
1	XABCY	2	8
2	XABC	2	6
3	ABCY	2	6
4	ABC	4	8
5	XAB	2	4
6	BCY	2	4
7	AB	4	4
8	BC	4	4
9	XA	2	2
10	CY	2	2

There are two separate problems in using this method. The first is to find a good set of phrases to use in the substitution process, and the second is to use the phrases in a way that will give maximum compression. Using the above set of phrases, replacing 'XABCY' first and 'ABC' second yields:

ABCXABCYABCZXABCY	length = 17
ABC (1) ABCZ (1)	length = 9
(4) (1) (4)Z(1)	length = 5

while replacing 'ABC' first yields:

ABCXABCYABCZXABCY	length = 17
(4)X (4)Y (4)ZX (4)Y	length = 9

It is assumed that the substitution algorithm is not iterative., i.e., that it does not recognize that X(4)Y is the same as XABCY and can be replaced by (1). While the algorithm could be made iterative, the processing overhead would increase drastically since each record must be scanned until a complete scan occurs with no substitutions. An algorithm to determine how each data string should use the available phrases to minimize its storage requirement will be given in the detailed description. (See Detailed Description below). Note that substituting for the longest phrase first does not necessarily give the most compression. An algorithm to choose the phrases is also given.

2. Data Types: This method can be used on all types of data.

3. File Types: Active files could be compressed with this technique as well as backup or stable files.

4. Relative Effectiveness: The following published figures demonstrate that his technique is very effective. However, the overhead is very high. The greatest amount of overhead is in the compression time. This is a result of the algorithm which is needed for finding the optimal phrases to be suppressed. Once the set of phrases has been obtained, the analysis routine need not be run again unless the file is extensively modified.

An 11,221 byte file consisting of PL/C compiler diagnostic messages was compressed to 8,194 bytes (a compression factor of 1.37). This included the space required for the common phrase table. This experiment used a fixed length 8-bit code for the phrase references. (Wagner, 1973).

It is possible to use this method with variable length codes. McCarthy (1974) compressed material from 8-bit bytes using Huffman codes for his phrase references and characters. His compression factors (original size divided by compressed size) were:

English test	2.38 (3.36 bits/character)
Name and address list	3.25
COBOL Source	5.91
360 Object Module	1.68

Detailed Description:

Algorithm: Two algorithms will be described in this section. First, McCarthy's method of selecting the set of phrases to be used in the encoding process will be described. This will be followed by Wagner's algorithm to maximize the compression by making the correct series of substitutions.

Phrase Selection Algorithm: McCarthy used the following algorithm to select his set of phrases.

1. Set m to be an upper limit on the length of the phrase to be considered. Set n to the number of characters to be used in the sample which will be analyzed. The sample is denoted by $c_1c_2....c_n$.

2. Scan the sample setting up a file, F1, of n-m+1 phrases, each of length m characters where $phrase_i$ is the substring $c_ic_{i+1}....c_{i+m-1}$. Discard overlapping duplicates in this file, i.e., if $phrase_i$ = $phrase_j$ and $|i-j|<m$, discard one of them. e.g., if m=6 then in the string ABCDABCDABGHIJ.... $phrase_1$ and $phrase_5$ (both ABCDAB) are overlapping duplicates.

3. Sort the phrases in F1 into alphabetical order. This simplifies subsequent scanning of the file.

4. Scan the sorted file of phrases, and, for each phrase of length at least 2 and its subphrases which start at its left, count how many times the phrase or subphrase occurs. If the frequency is sufficiently large (see below), enter the appropriate phrase, together with its length and frequency, as a record in a new file, F2. The subphrases consist of the first 2,3,...,m-1 characters of the phrase. There are up to (n-m+1) x (m-1) phrases to be considered. In deciding whether or not to enter a phrase into F2, McCarthy chose to do so if use of the phrase gave a compression of 0.2% or more. The saving in space by using a phrase is approximated by:

$$F(L_S-1) - N_S/1500 \text{ bytes}$$

where: F is the number of times the phrase occurs
L_S is the length of the phrase
N_S is the length of the string to be compressed
$N_S/1500$ allows for the increase in the lengths of the codewords due to the necessity of encoding another phrase.

For 0.2% savings or more:

$$\frac{N_s}{500} < F\,(L_s-1) - \frac{N_s}{1500}$$

or approximately, $F(L_s-1) > N_s/400$

This is the selection criterion used.

5. Scan the file F2 to find the phrase which will yield "maximum" compression, i.e. the phrase for which $F(L_s-1)$ is maximum, and place it, again with its length and frequency, in file F3 (the final list of phrases to be used in encoding).

6. Amend the remaining records in F2 as follows:

   a. If any phrase in F2 is contained by the selected phrase as a substring, then that phrase has its frequency reduced by the frequency of the selected phrase.

   b. If any phrase in F2 contains the selected phrase as a substring its frequency n' is replaced by n'(1-L/L') where L and L' are respectively the lengths of the selected phrase and the phrase which contains it.

   c. If there is a partial overlap between the selected phrase and a string in F2, then either rescan the sample to determine the new frequency or subtract the frequency of the selected phrase from the overlapping phrase. The latter alternative can save a lot of computer time (especially for a large sample) with only a small loss in compression.

7. Repeat steps 5 and 6 until no phrases remain which would give enough further compression or until the specified number of phrases has been selected.

Phrase Substitution Algorithm: Wagner assumed that a list of phrases to be replaced has already been compiled. The file is compressed in sections because, as will be apparent from the algorithm, the overhead will increase beyond reason if the strings to be compressed are too long.

The algorithm works by starting at the end of the string to be compressed and working back towards the beginning, finding the best substitutions possible at each intermediate step. The compressed string is a string of phrase references and character strings and is terminated by an end mark. The space taken by these three items is:

- Phrase reference - 2 bytes (a phrase number and the length of the phrase)
- End mark - 1 byte
- Character string - 2 bytes + length of string (the two extra bytes are the character string indicator and the length of the string)

The length of the character string is not necessary, but its use speeds processing because the string does not have to be searched character by character for the next phrase reference or end maker. There does not seem to be any need to store the phrase lengths in the compressed string - this information should be in the phrase dictionary.

Let P denote the set of phrases to be suppressed, and p a phrase in P. /p/ is the length of the phrase p. Let Q(j) be the subset of P for which the phrases match the j, j+1,...., j+/p/-1 characters of the string to be compressed. i.e., $p \in Q(j) \Leftrightarrow$ p is identical to the j, j+1,...., j+/p/-1 characters of the string. Let the string have N characters. Define the functions:

G(j) = The least space needed to store character j, ..., N of the string provided that the final form of the compressed string begins with a character string.

H(j) = The least space needed to store characters j, ..., N of the message regardless of the form of the first component of the compressed string.

The algorithm finds H(1). Provided the steps to finding it are retained, the string can be compressed to this value. The function G(j) is needed to account for the effect that the leading component of the message has when prefixed by another character. If that leading component is a character string, the added cost of absorbing a single character is one byte whereas if it is a phrase reference it costs three bytes to absorb a preceding single character by encoding it as a separate character string.

The algorithm is:

1. Set: $i = N$
   $G(N+1) = 3$ $\quad H(N+1) = 1$

2. Find the set Q(i)

3. Calculate: $G(i) = \min [G(i+1)+1, H(i+1)+3]$
   $H(i) = \min [H(i+/p/)+2, G(i)]$
   where the minimum for H(i) is over all $p \in Q(i)$

4. If i = 1, stop. Else decrement i by one and go to step 2.

Example of Phrase Substitution Algorithm

String to be compressed = PAUL␣RUN

N = 8

P = {PAUL, AUL, AUL␣, L␣, ␣R, ␣RU, RUN, UN}

i = 8 String = N

G(9) = 3

H(9) = 1

Q(8) = null

G(8) = min [4,4] = 4

H(8) = 4

i = 7 String = UN

Q(7) = {UN}

G(7) = min [5,7] = 5

H(7) = min [H(9)+2,5] = 3

i = 6 String = RUN

Q(6) = {RUN}

G(6) = min [6, 3+3] = 6

H(6) = min [H(9)+2, 6] = 3

i = 5 String = ␣RUN

Q(5) = {␣R, ␣RU}

G(5) = min [7, 6] = 6

H(5) = min [H(7)+2, H(8)+2, 6] = 5

i = 4 String = L␣RUN

Q(4) = {L␣}

G(4) = min [7,7] = 7

H(4) = min [H(6)+2, 7] = 5

i = 3 String = UL␣RUN

Q(3) = null

G(3) = min [8, 8] = 8

H(3) = 8

i = 2 String = AUL$RUN

Q(2) = {AUL, AUL$}

G(3) = min [9, 11] = 9

H(3) = min [H(5)+2, H(6)+2, 9] = 5

i = 1 String = PAUL$RUN

Q(1) = {PAUL}

G(1) = min [10, 8] = 8

H(1) = min [H(5)+2, 8] = 7

The compressed string is:

PAUL	+	$R	+	UN	+	end marker
2 bytes		2 bytes		2 bytes		1 byte

References:

Wagner, R. A.

COMMON PHRASES AND MINIMUM-SPACE TEXT STORAGE; Comm. of the ACM, V. 16, pp. 148-152, 1973

Weaver, A. C.

DATA COMPRESSION FOR CHARACTER STRINGS; Univ. of Illinois, July 1974, NTIS: PB 234 775

McCarthy, J. P.

AUTOMATIC FILE COMPRESSION

In: International Computing Symposium, 1973 (eds A. Gunther, B. Levrat, and H. Lipps) American Elsevier N.Y., 1974, pp. 511-516

Adaptive Character String Substitution (Pattern Substitution)

General:

1. Technique: The main feature of this technique is that it adapts itself to the data is compressing. Since this is obviously a much more complex process than a fixed substitution, the overhead in both processing time and memory required is an order of magnitude greater than fixed substitution techniques. The compression achieved is very high and no preliminary activities, such as generation of a code table, are necessary.

   The compressor starts with its code tables empty, except for one entry for each character in the source character set. The compressor scans the input data and keeps count of the occurrence of each character pair. When the count for a character pair reaches a threshold value (which may be settable by the user) the compressor defines a substitution code for that character pair. The definition is passed to the decoder as a special instruction in the compressed data. The process is iterative in the sense that counts are kept for the use of defined substitution codes in combination with other substitution codes or characters. Thus, although each substitution code is defined in terms of two other characters or substitution codes, it may represent a long string of characters in the original data. For example, a long string of X's in the input data will result in the definition of a code for XX (say @), then the definition of a code for @@ (say $) which represents XXXX in the input data, then the of a code for $$, representing XXXXXXXX in the input data, and so on. Obviously, the compressor requires some large tables and spends a considerable amount of time searching and managing them. The decompressor is much simpler. It only has to recognize a new substitution code definition and update its table accordingly. Decompression consists of substituting the correct character string for each code in the compressed data.

From the above description, it is clear that the compression achieved depends on how regular the data is and how much data has been processed. Initially, very little compression is achieved because only a few substitution codes have been defined. After a few thousand words, many substitutions will have been defined (unless the data is random) and compression will approach the maximum possible with this method.

2. Data Types: This method is extremely effective on any data which is not purely random.

3. File Types: Due to the "warm up" required, the method is suitable only for files of several thousand words or longer. The very high overhead of both compression and decompression probably makes it unsuitable for active files unless a high compression factor must be achieved.

4. Relative Effectiveness: This is an extremely effective compression technique. On most large files it gives 1 1/2 to 2 times as much compression as an optimal Huffman code. The price for this performance is that compression and decompression respectively take about 10 and 5 times as much CPU time as Huffman coding, and the routines occupy several thousand words of memory, compared with several hundred words for Huffman coding.

Detailed Description:

The only implementation of this algorithm known to the authors is a package written by the Lambda Corporation for the Government. Only a user's manual was available to PRC, so no detailed description can be given.

Only one detail can be added to the general description given earlier. The package assumes that the input data is BCD, and reads it as

6-bit characters. The output, however, is in 9-bit codes so that 512 substitution codes are available. In spite of the fact that the input is read in 6-bit characters, the package does effectively compress ASCII files. However, unless the file is very long, the advantage over Huffman coding is not as great as with BCD files. The performance with ASCII files demonstrates the power of the algorithm.

References:

Lambda Corporation

DATA COMPRESSION SYSTEM FOR WORLD-WIDE MILITARY COMMAND AND CONTROL SYSTEM, USERS MANUAL (DRAFT)

March 15, 1973, Arlington, Virginia

Variable-Length Coding For Characters And Character Strings

Huffman Codes

General:

1. Technique: Huffman codes are variable length codes which take advantage of the statistical probabilities of occurrence of message units (characters) so that short representations are used for characters which occur frequently, and longer representations for characters which occur infrequently. When variable length codes are used there must be a way to tell where one character ends and the next one begins. This can be done if the code has the prefix property, that is, that no short code group is duplicated as the beginning of a longer group. Huffman codes have this prefix quality and in addition are optimum in the sense that data encoded in these codes could not be expressed in fewer bits by any code based on the same source alphabet.

2. Data Types: Business type data files have been the most frequent type of data compressed with Huffman codes. However, text and almost any other highly redundant data can be compressed effectively as well.

3. File Types: Huffman codes lose effectiveness if the statistical properties of the file change over a period of time. Thus, a new code may be needed for a file if the character frequencies have changed considerably. This is unlikely to be necessary unless the type of data in the file has changed or the file size has more than doubled. Apart from this, there are no limitations on file types.

4. Relative Effectiveness. Huffman coding is very effective, particularly when combined with repeat suppression. It is effective on

unformatted files such as text files. On formatted files, it is only slightly more effective than interrecord comparison techniques and the extra CPU time required by Huffman coding may not be justified by the small amount of extra compression obtained. The only type of data on which it is not effective is data like compressed decks, where the use of the source characters is quite uniform.

Detailed Description:

1. Algorithm: For a detailed description of how Huffman codes are encoded and decoded refer to the section of this document titled Variable Length Codes. The most important factor toward developing suitable Huffman codes will be described here.

   This factor is the careful selection of the base character set used to derive the codes. If a file consisting of only text data is to be compressed, the selection is straightforward. In this case the English alphabetic characters, spaces, and punctuation marks are used as the character set from which the code is derived. If, however, the file is not pure text, but say, an inventory file, a more detailed analysis of actual data is desirable. An inventory file would probably have a greater proportion of numbers, repeated blanks, and proper names that a text file. Thus statistics derived from text would not be accurate and a code derived from text statistics would not be optimal.

   In the Ruth and Kreutzer study, many character sets were tried before an acceptable compression ratio was achieved. Ruth and Kreutzer considered strings of 2, 3, 4 and 5 BCD zeros, binary zeros and blanks to be single characters for the purpose of encoding. The additional patterns of zeros and blanks took advantage of the fact that when default values occurred in the file, they tended to occur in contiguous field sized units. Only by including these patterns in the source character set did Huffman coding provide a 2 to 1 compression ratio.

An alternative to having separate codewords for repeated strings of various lengths is to build repeat suppression into the encoding algorithm. A special codeword indicates a repeat string, and this is followed by the codeword for the repeated character and a fixed field count of the number of repeats. For files where there are a lot of zero runs and blank runs, special codewords can be used indicating these two types of repeats. They need only be followed by a repeat count.

If a Huffman code is based on a subset of the possible character set, a copy code should be provided. This is a special codeword which is used to indicate that the character following it is reproduced exactly as it occurred in the source file. This allows characters which rarely occur in the data to be excluded from consideration when the Huffman code is derived.

2. Tuning: Ideally, Huffman coding uses an optimum code derived for each file to be compressed. In this case, tuning is not really carried out. Once the type of encoding algorithm (with or without repeat suppression and a copy code) and the base character set are chosen, the remaining processes are fixed procedures. Tuning involves only the trial of various encoding algorithms and base character sets to determine which ones are most effective.

   In fact, it is possible to use one code table on similar files with very little loss in compression. For example, card image source language programs can be compressed with one table, irrespective of the language. Tuning in this case involves deriving several similar codes and finding which one gives the best overall performance.

   If repeat codewords are used, the statistics gathering and code generation procedures should reflect this fact. The character counts used in code generation are those that will occur in the compressed file, not those that occur in the original file. Repeat

codewords and the copy codeword, if used, should be Huffman codewords, not special fixed length codewords which are guaranteed to not occur in the Huffman encoded file.

Example: The following Huffman code was derived using the procedures described in Chapter 3. The data file was a small single case ASCII text file. The character counts and the Huffman codeword for each character are shown. All numbers in the table are octal. Characters not found in the source file are assigned the copy code (17777), which is listed as character 201. Character 200 is the repeat codeword (only one is used). Most codewords start with a binary 1, so the length of most codewords is the minimum number of bits needed to express the octal number; e.g., 35 is a 5-bit codeword (11101). Where the codeword starts with a zero, the length is given in parentheses.

Character (Octal)	Count (Octal)	Codeword (Octal)
00	2	3776
01-05	0	17777
06	1	17776
07-37	0	17777
40	1734	0 (3 bits)
41	0	17777
42	14	766
43-45	0	17777
46	2	3775
47	5	1773
50	11	773
51	11	772
52-53	0	17777
54	31	367
55	27	370
56	66	167
57	12	770

Character (Octal)	Count (Octal)	Codeword (Octal)
60	40	364
61	43	171
62	25	371
63	33	366
64	13	767
65	6	1771
66	0	17777
67	20	765
70	4	1774
71	6	1770
72	2	3774
73	3	3772
74-76	0	17777
77	1	7776
100	0	17777
101	504	07 (4 bits)
102	76	166
103	176	64
104	267	25
105	1106	1 (3 bits)
106	132	70
107	104	164
110	263	26
111	521	05 (4 bits)
112	20	764
113	11	771
114	230	31
115	152	66
116	436	24
117	437	11
120	142	67
121	34	365
122	454	10

Character (Octal)	Count (Octal)	Codeword (Octal)
123	511	06 (4 bits)
124	636	04 (4 bits)
125	242	30
126	55	170
127	126	71
130	5	1772
131	76	165
132	2	3773
133-176	0	17777
177	246	27
200 (repeat codeword)	156	65
201 (copy codeword0	0	17777

References:

Kreutzer, P. J.

DATA COMPRESSION IN LARGE BUSINESS ORIENTED FILES;
Navy Fleet Material Support Office, Mechanicsburg, Pa., Oct. 5, 1971.
NTIS: AD 734 394

Maurer, W. D.

FILE COMPRESSION USING HUFFMAN CODING
In: Computing Methods in Optimization Problems v. 2, pp. 247-256.
Proc. of Conf. in San Ramo, Italy in Sept. 1968. Academic Press,
N.Y., 1969.

McCarthy, J. P.

AUTOMATIC FILE COMPRESSION; In: International Computing Symposium,
1973 (eds A. Gunther, B. Levrat, and H. Lipps) American Elsevier N.Y.,
1974. pp. 511-516

Monmens, J. H. and Raviv, J.

CODING FOR DATA COMPACTION; IBM Research Report No. RC5150, Yorktown Hgts, N.Y., Nov 26, 1974.

Ruth, S. S., and Kreutzer, P. J.

DATA COMPRESSION FOR LARGE BUSINESS FILES; Datamation, Sept 1972, pp. 62-66.

Word Dictionary Techniques

Split Dictionary Encoding

General:

1. Technique: This section covers word dictionary encoding methods with the characteristic that the dictionary is divided into several distinct sections. This allows the synthesis of long words instead of having a separate entry for each word to be encoded, as occurs in an integrated dictionary. These techniques may be regarded as a sophisticated extension of the nonadaptive character string substitution methods already discussed.

   In a single dictionary encoding, the list of "words" (character strings) is stored in a table and each has associated with it a unique code. The input string is scanned and, whenever one of the words in the dictionary is found, it is replaced by its associated code. In a split dictionary, there are several dictionary tables. In a stem and suffix system, there are separate dictionaries for word stems and suffixes. The input string is scanned and whenever the stem of a compound word is found in the stem dictionary, the suffix dictionary is searched to see if it contains the suffix of the compound word. It it does, the word is replaced by codes for the stem and suffix. In a simple encoding program, only complete substitutions are made for compound words; i.e., both stem and suffix must be in the dictionaries for any substitution to be made. If only one or the other is in the dictionaries, no substitution is made. The suffix dictionary contains many entries such as -e, -ly, -lly, -able, -ible, ed, -y, -d, -ing, le, so that virtually all compound words with stems in the stem dictionary will have suffixes in the suffix dictionary.

   Schwartz (1963) shows that using a split dictionary with separate sections for stems and suffixes allows more words to be encoded for

a given dictionary size that could be encoded with an integrated dictionary. An integrated dictionary allows more compact and faster encoding than a split dictionary, so one can attempt to have the best of both worlds by including some compound words as separate entries in the dictionary. For example, "under" and "understand" could both be stems and "stand" could be the suffix dictionary. Then "understand" could be encoded either as a single stem or as a stem-plus-suffix. The search routine has to be carefully designed to ensure that "understand" will be encoded as a stem, since this gives more compression. Having a spearate stem entry for "understand" allows such words as "understandable" and "understanding" to be encoded as stem-plus-suffix. This allows the synthesis program to be simplified to create only stem-plus-suffix words without losing the ability to synthesize words with multiple suffixes.

The binary code used to represent the words may be fixed length or variable length. Wh[illegible] (1967) used a fixed length code, but calculated that he could [illegible]e obtained about 4% more compression if he had used a Huffman code.

These techniques do not have a dictionary entry for all possible words, so they include a spelling mode in which words which cannot be synthesized using the dictionary are spelled out character by character. Special codes in the compressed data tell the decoder to switch modes to follow the changes in mode of the compressor. The spelling mode is terminated by a special character which is interpreted by the decompressor as an instruction to switch to the substitution mode. Similarly, the substitution mode is terminated by a special code which acts as an instruction to start the spelling mode. The spelling mode is also used for punctuation and special characters. Frequently used character strings can also be included in the character part of the dictionary.

2. Data Types: These techniques are primarily intended for text compression. They should be effective on program source provided the dictionary is correctly chosen. They may be effective on data files with a large number of frequently used fixed length data items.

   The encoding program can be written to gather word use statistics. These statistics can be used to modify the dictionary to improve compression or to adapt it to a specific file if statistics are generated for a sample of the file. If a fixed length code is being used for dictionary entries, limited dynamic tuning can be accomplished by starting compression with part of the dictionary empty and filling in the empty spaces with words appearing in the early part of the file which are not in the dictionary.

3. File Types: There is a considerable overhead in storing the dictionary, so the techniques are most effective on rather large files. However, if a dictionary with 1,000 entries is used, the dictionary storage overhead is much less than that required for a complete dictionary (as in the section Intermediate Dictionary Compression) so these methods could be used on files too small for Intermediate Dictionary Compression.

   A fixed code is used throughout the file, so the file can be searched while still compressed by compressing the query. If the file is compressed a page at a time, and a page dictionary is stored at the front of the file, updates are also possible without decompressing the entire file. Thus, these methods are suitable for active files. They are not suitable for index files because the mixed word and character encoding makes it impossible to do magnitude comparisions or alphabetic comparisons on an item without decompression. Only a match/no match query can be answered in the compressed state.

4. Relative Effectiveness: White reported a compression factor of approximately 2:1 on a series of news stories taken from the Associated

Press wire service. The input character set consisted of 64 characters, so the compressed text used about 3 bits per character.

Schwartz and Kleiboemer (1967) report on a test of Schwartz's dictionary of over 5000 words, suffixes, symbols, and characters. On 19,170 words of general text taken from magazine articles they achieved a compression to 3.19 bits per character.

Both core requirements and processing overhead are strongly dependent on the total size of the encoding dictionary. Dictionary size can range from 500 entries to 10,000 entries, and some special applications may use dictionaries outside these limits. White used dictionaries of approximately 850 and 1,350 entries and Schwartz used a dictionary with 5,208 entries. Using more than 1,000 words in the dictionary does not appear to improve the compression greatly for most text data.

Using a variable length encoding of the dictionary will increase the overhead and improve compression by about 5%.

It appears that an encoding program using a 1,000 word dictionary could be implemented in 5-10 K words of core. The decoding program would require somewhat less core and would operate much faster than the encoding program.

These techniques give performance slightly better than Huffman coding, but at a considerably higher cost in both memory and CPU resources used. They use fewer resources than adaptive character string substitution, but they don't achieve as much compression since they cannot follow changes in the input data characteristics. These routines provide a compromise in compression and resources used between Huffman coding and adaptive character string substitution. This compromise appears unlikely to suit many users.

Detailed Description:

1. Algorithm: The total size of the dictionary (stems, suffixes, special characters, individual characters and frequently used character strings) is an important parameter in these systems. Schwartz (1963) analyzed several studies and concluded that "a vocabulary of between 500 and 1,000 unique words can constitute the basis of a dictionary which will cover approximately 75% of any word sample." In one study, 590 words made up about 80% of the 4.26 million words of text examined. It appears that fewer than 100 words will match 50% of most word samples (White, 1967, figure 2). Thus, the extra compression attainable by extending the dictionary becomes smaller as the dictionary gets larger. Furthermore, a larger dictionary requires a very efficient search routine to avoid unnecessary processor overhead. Schwartz, with a 5,000 item dictionary, used a table-lookup to find the range of addresses in which to do a binary search. This initial address range was found by matching the first three characters of the word to be encoded. Once a match is found, the resulting binary search of dictionary entries with this trigram is always limited to less than 100 items. This required adding all compound words to the dictionary which have one and two letter stems, since these stems cannot be used in the synthesis procedure. (The alternative is to spell out such words in character mode.) White, with a much smaller dictionary of 1,340 entries, based his search on matching the first letter of the word.

   The dictionary can be built by collecting statistics for the file to be encoded, or by using standard word counts and counts of suffixes and common character strings. (Pratt, 1942; Thorndike and Lorge, 1944).

   In order to keep the programs fairly simple, only one suffix is added to each stem. Multiple suffixes can be accommodated by adding

a compound word to the dictionary as a separate item, as described previously.

Dictionary searching is facilitated by left justifying the words so that each dictionary word starts on a machine word boundary. The words can then be compared in binary, since in both BCD and ASCII codes the letters are numbered consecutively.

Most words of greater than 12 characters are complex and can be synthesized. They are also not very common, so little compression is lost by limiting dictionary entries to 12 character words and spelling out the words that cannot be synthesized.

The exact encoding algorithm depends on: (1) how big the dictionary is to be; (2) how complex the program can be; and (3) special characteristics of the source file.

White's source file was newspaper copy. It contained variable interword spacing, many hyphenated words at the ends of lines, and both upper and lowercase letters. Both his dictionary and his encoding algorithm reflected the nature of his source material. A special dictionary contained the spacing characters, shift symbols and nine special symbols to indicate hyphenation after the first, second, ..., ninth character. Both White and Schwartz suppressed interword spaces, since the decoder always knows when a complete word has been decoded. White used a given space symbol until a different space symbol was detected. This new space symbol was used until yet another space symbol was detected. This mode of operation was suitable because interword spaces within a line were all the same. (Recall that White's data was newspaper copy formatted for newspaper columns.)

White coped with upper and lowercase letters by: (1) having a special section of his dictionary for words which always began with capitals and words which frequently appeared with the first

letter capitalized; (2) having upper and lower case symbols so a capitalized word could be spelled out; and (3) having a symbol in the special dictionary which indicated that the following word should have its first letter capitalized. This meant that any word in the dictionary could be capitalized at a cost of one code-word.

White did not attempt to implement grammatical rules. Capitalized words were first searched for in the capital word dictionary. If not found there, the word dictionary was searched. This yielded at least the first letter (coded as upper case, letter, lower case) and the remainder of the word was searched for in the suffix dictionary. This yielded the codes for character strings or single characters if the word had to be spelled out. Uncapitalized words were encoded similarly, but the search of the capital word dictionary was omitted. White used a fixed length code for all dictionary entries.

Schwartz implemented several rules in his synthesis program. Schwartz Huffman-encoded his words and characters separately, and used a special mode symbol to switch from word encoding to character encoding (for spelling out words not in the dictionary and which could not be synthesized). Using separate codes and a mode symbol gave better compression than using one code for both. Schwartz tagged all the entries in his stem dictionary with one of the following tags:

SYNTAG 0: Word can not appear in complex form; irregular form appears as word type; or word is regular and suffix is added without change in word; e.g., build, field.

SYNTAG 1: Final E of word is deleted upon adding a suffix beginning with a vowel; e.g. file, live.

SYNTAG 2: Final consonant of a word is doubled upon adding a suffix beginning with a vowel; e.g., run, pop.

SYNTAG 3: Final Y of word is changed to I upon adding a suffix beginning with letter other than I; e.g., fly, modify.

By associating a SYNTAG with each dictionary word and by identifying the initial letter of each suffix, simple routines for the synthesis and decomposition of complex words were devised.

The synthesis tags (SYNTAG) were dividied as follows in the dictionary:

SYNTAG 0		3,819
SYNTAG 1	819	
SYNTAG 2	212	
SYNTAG 3	303	1,334
		5,153

The dictionary search routine had to search on both sides of the word to be synthesized, since "love" appeared after "lovable" but before "loving", and "reply" appeared before "replying" but after "replies". Since 74% of the words in the dictionary were SYNTAG 0, it is debatable whether the effort of programming the routines and keeping the SYNTAGs was worthwhile. An alternative would have been to store only the truncated stems of SYNTAGS 1 and 3 and to encode the original words as stem-plus-suffix by including the letters e, i, y in the suffix dictionary. Words with SYNTAG 2 could be handled by storing extra stems with the final consonant doubled. The saving in program size and SYNTAG bits may have offset the extra dictionary entries.

2. Tuning: Extensive tuning of this method to the data to be compressed is apparent in the detailed description of the algorithms above. The numerous special cases and the effects of the dictionary size mean that careful matching of the three components (algorithm, dictionary and data) is necessary for efficient performance.

3. Details of Published Performance: White reports performance figures for two sizes of dictionaries. His source material was a randomly selected set of news stories taken from the Associated Press wire service. The 64 character code used was transliterated into the 64 character computer code set. No attempt was made to remove spacing and other special symbols in the linotype text, but the dictionary and encoding algorithm were adapted to cope with the special characteristics of this text.

White optimized his dictionary for 115,000 characters of text and achieved a compression factor (input/output) of 1.89 for this material when using a 1340-entry dictionary. Using this same dictionary on a further 13,000 characters of text gave a compression factor of 1.82. The dictionary was reduced to 831 entries by eliminating the least used entries and this smaller dictionary gave a compression factor of 1.75 on the 115,000 character text sample. White used a fixed-length code for his dictionary encodings. He estimated that an improvement of 4% could be obtained by using Huffman code. In this case, the compression factor would approach 2.0 or 3 bits/character.

Schwartz constructed a dictionary of the 5,153 most frequently occurring words in approximately 4.5 million words of magazine articles. To these he added numerals, punctuation marks, geographic names and 43 suffixes to increase the dictionary to a total of 5,208 entries. This dictionary was used to encode 7 articles from 4 magazines -- a total of 19,710 words. The final data rate was 3.19 bits/character. The total number of different words in this sample was about 4,200. Of these, almost 2,000 were in the dictionary, approximately 1,250 could be synthesized as stem-plus-suffix using the dictionary, and the remaining 950 had to be spelled. In the encoded text stream, approximately 80% of the words were dictionary entries, 12% were synthesized and 8% were spelled (Schwartz and Kleiboemer, 1967).

References:

Pratt, F.

SECRET AND URGENT; Bobbs-Merrill, New York, 1942

Thorndike, E.L. and Lorge, I.

THE TEACHERS WORD BOOK OF 30,000 WORDS; Columbia University, New York, 1944

Schwartz, E. S.

A DICTIONARY FOR MINIMUM REDUNDANCY ENCODING; Jnl. ACM, V. 10, pp. 413-439, 1963

Schwartz, E. S. and Kleiboemer, A. J.

A LANGUAGE ELEMENT FOR COMPRESSION CODING; Information and Control, V. 10, pp. 315-333, 1967

White, H. E.

PRINTED ENGLISH COMPRESSION BY DICTIONARY ENCODING; Proc. IEEE, V. 55, pp. 390-396, 1967

Intermediate Dictionary Compression

General:

1. Technique: This is basically a word dictionary technique. It uses Huffman coding and run length coding (to compress binary strings with many zeros and a scattering of ones) as integral parts of the method. There is no provision for spelling out frequently used words, i.e., all words used must be in the dictionary. The method could be modified to include a spelling mode. A feature of this method (which could also be applied to other compression techniques) is that Huffman codes are defined algorithmically so that no code table is required. (See section titled Variable Length Codes.)

The entire file is scanned to compile a complete dictionary of words and break characters (asterisk, period, parentheses, coma, etc.). A character count is made for the break characters but not for the words. Based on this count, the break characters are Huffman coded. The words are Huffman coded assuming they are equally probable. This is almost the same as assigning binary numbers to them. At the cost of more CPU overhead, they could be Huffman coded based on the frequency with which they occur.

The file is encoded as follows. A string of 1,500 words plus break characters (total) is taken from the file. A binary "presence vector" is constructed. It has one bit for every word in the dictionary. If the word corresponding to a bit is present in the string to be encoded, then that bit is set to 1. All other bits are zero. This presence vector defines the intermediate dictionary, which consists of all the words corresponding to the bits set to 1. The total number of words in the intermediate dictionary is counted and each word is assigned a Huffman codeword based upon its position in the intermediate dictionary. (As noted above, this is almost the same as assigning a binary number to each word.) The encoding for the string then consists of:

a. The compressed presence vector (it is run-length encoded, since it contains many more zeros than ones.)

b. The encoded words and break characters comprising the string in the order in which they appear. These are encoded by the concatenation of a 0 bit and the Huffman codeword for each break character, and a 1 bit and the Huffman codeword assigned to each word.

The main dictionary is included at the start of the compressed file and is followed by the Huffman code for the break characters and then by the codes for all the strings which are the contents of the uncompressed file.

2. Data Types: This method is intended for text data. With a suitable choice of words, it could probably be adapted for program source and may be adaptable for some data files. It is not suitable for program object files or binary files.

3. File Types: Due to the considerable overhead involved in storing the dictionary, the method is only really effective for large files. The entire file must be re-encoded if the original file is updated, and any searching must be done sequentially and would be easiest to do on the decoded file. All these points lead to the conclusion that the method is most suited to large inactive text files which are not subject to frequent searching.

4. Relative Effectiveness: On a rather unfavorable source file reported by Cullum (short words with frequent misspelling) a compression to 2.82 bits/character was achieved. This is good for text compression.

There is considerable overhead in both encoding and decoding. Encoding involves scanning the entire file twice -- the first scan is needed to build the dictionary and the second scan is needed to do the encoding. The processing time during the scans depends on how elaborate an encoding one wishes to do. In his experiment, Cullum used several simplifications to reduce processing time without appearing to sacrifice much in final compression.

Decoding is much faster than encoding. The limiting factor is the speed at which the decoded data can be written on the peripheral.

A detailed description of Cullum's experiment follows in the section titled Detailed Description.

On an IBM 360/75 Cullum estimates that encoding can be done at "a few thousand characters per second" and decoding can be done about 250,000 characters per second.

The core requirements are high. The method uses a dictionary which must reside in core. The dictionary must contain all words in the file, so it occupied several tens of thousands of words (see Detailed Description below).

Detailed Description:

1. Algorithm: The encoding and decoding algorithms will be described step by step. Alternatives or simplifications for each step will be described along with that step. Most of these simplifications are based on Cullum's description of his programs.

   a. Encoding: Encoding is a two stage process. In the first stage, the file to be compressed is searched and the dictionary is built. Since most of the overhead in this stage is the time spent searching the current dictionary, this should be as efficient as possible. Cullum split his dictionary into 3 segments according to word length. There are very few English words longer than 20 letters, and none (except place names) longer than 30 letters. Splitting the dictionary into more segments speeds the search but requires more core. The tradeoff depends on the available resources. It is reasonable to base the dictionary segments on the number of characters in a machine word. A machine with 6 characters per word could have its dictionary split into 4 segments: words of length 1-6 characters, 7-12 characters, 13-18 characters, and 19 characters or more. A machine with 4 characters per word could use a three-part dictionary (1-8 characters, 9-16 characters, 17 characters and over) or could improve its search time by using five segments (1-4 characters, 5-8 characters, 9-12 characters, 13-16 characters, over 16 characters).

      Cullum combined dictionary building with an intermediate encoding. Since he did not sort his dictionary according to usage, he encoded the source file on a scratch tape such that each word was

identified by the section of the dictionary it was in and the position of the word in that section. The break characters were encoded similarly.

The second stage of the encoding begins by writing the size of each dictionary section on to the output file. This is followed by the complete dictionary (a code more efficient than ASCII should be used) with the words separated by a blank. The number of words in the source file, or, in Cullum's case, the number of words plus break characters (excluding redundant spacing characters) is written to the output file and this is followed by the size of the subtexts into which the file is broken for final encoding. Cullum Huffman-encoded the break characters separately, so he next wrote the list of break characters along with the length of their codewords so that the decoder could reconstruct the code. (Cullum specifies a Huffman code by an algorithm, as described in the section Variable Length Codes, so a dictionary of codewords is not necessary.)

The source file is now encoded. Cullum used intermediate strings of 1,500 words plus break characters, which corresponded to about 1,000 words. His theoretical study had shown that, for his input file, this was close to optimal. The cost curve had a broad null between 500 and 1,500 words, and even using 250 or 4,000 word segments would only have affected the compression by a few percent. This does not appear to be a critical parameter.

The intermediate string is read and the binary presence vector (the intermediate dictionary or ID) is constructed. This is compressed, since it is mostly zeros. Cullum used the following compression method. He recoded the string into a set of 24 codewords, defined thus:

For $0 \leq i \leq 14$, S_i is a string of i zeros followed by a one.
For $15 \leq i \leq 23$, S_i is a string of $15 \times 2^{i-15}$ zeros.

The ID was encoded using as few codewords as possible. A frequency count was done on these 24 codewords and a Huffman code was used to represent them. The lengths of the Huffman codes for each of the S_i was written on the output file, followed by the number of ones in the original ID and then the Huffman encoded ID. The encoded text follows. The Huffman code for the encoded text is derived assuming that all words in the ID are equally likely. Because of this, some words will have k bit codewords and some will have k + 1 bit codewords. If there are n words in the ID then $2^k \leq n \leq 2^{k+1}$. We give the first m words k bit codewords. Then $m=2^{k+1}-n$ and the i-th word in the ID has the codeword of length k representing (in binary) the number i-1 if $i \leq m$ and if $i > m$ it has a k+1 bit codeword which represents the number m+i-1.

The text is actually written to the output file using an extra bit on each codeword to distinquish words from break characters. Break characters are represented by a zero followed by the codeword for the break character and words are represented by a one followed by the appropriate codeword.

b. Decoding: Decoding begins by reading in the dictionary and transliterating it back into the normal machine character representation if it was stored using a different character representation. Cullum constructed his decoding dictionary with the number of letters in each word (in binary) preceding that word. The dictionary was packed and a separate table containing the starting byte address of each word was built.

The first compressed ID and its associated text string is then read in. The ID is decompressed and a table of pointers to the

main dictionary is built which contains the start address of the i-th word in the ID as the i-th entry in the table. The compressed text can then be decoded and written on the output file. The Huffman decoding procedure is explained under Variable Length Codes. A decoding table of all the Huffman codewords is not needed.

2. Tuning: This algorithm is self-tuning in that deriving the Huffman codes is an integral part of it.

3. Published Performance: The routine was programmed to compress a section of the Bible which was 75,970 words long. The file contained many spelling errors as well as numerous special formatting characters. No attempt was made to correct errors or eliminate spurious characters, so the dictionary was larger than it would otherwise have been. The average word length was about 4 characters. Normal English text has as average word length in the range of 4.5 to 5.5 characters. Both of these factors undoubtedly caused a loss in compression compared to what could be achieved with normal text.

The text file contained 404,970 characters. 98,526 of these were "break" characters. The eleven break characters are: asterisk, period, comma, dollar sign, left parenthesis, right parenthesis, equal sign, dash, plus sign, blank and slash. Whereever two words were separated only by an asterisk (which was used instead of a blank in the source file), the asterisk was not encoded. The decoder automatically inserts an asterisk between two consecutive words. This eliminated 50,439 characters (equivalent to logical compression of about 12%). The remaining 354,531 characters were encoded into 1,141,185 bits. Thus the compressed file used 3.22 bits per character for the characters actually encoded, and 2.82 bits per character for the original file. (This corresponds to a compression ratio of 2.13 since the original file was in a 6-bit code.) These figures include the space

required for the dictionary, which was encoded using a 5-bit code for the characters.

In this experiment a fairly simple version of the encoding algorithm was used. Only one ID was used. Theoretical studies indicate that using two levels of ID would produce about 10% improvement in the compression and require roughly double the encoding time. The dictionary was not ordered according to word frequency, although the commonly used words would tend to be at the front of the dictionary. This meant that a true Huffman code was not constructed for the words. They were coded in a way that was almost the same as binary numbering. (A Huffman code was used, but it was assumed that all words had equal probability.)

Another simplification was that break characters and text were encoded separately. Cullum shows that, provided the number of break characters is within 50% of the number of word occurrences (i.e., number of break characters is between 50% and 150% of the number of words), the loss in compression is negligible. If this is not true, then break characters and words have to be coded together for maximum compression.

The presence vector which defines each ID is a sparse binary string (i.e., it is mostly zeros - the ones are sparsely distributed along the string of zeros). This is compressed using run length encoding of some sort. Cullum used an efficient if nonoptimal encoding. Since the compressed presence vector is only a small part of the final compressed file, it is not worth using elaborate compression techniques to compress it as much as possible. Cullum states that the most elaborate ID encoding will always produce less than 10% improvement in the overall compression.

Cullum ran his experiment on an IBM 7094 computer. Total encoding time for his 404,970 character file was almost 400 seconds. The

dictionary search routines were coded in assembly language, and most of the rest of the program was coded in Fortran. Some time savings could be made by programming more of the code in assembly language, but most of the time was spent in dictionary searching. Cullum states that on a machine like an IBM 360/75, encoding could be done at the rate of several thousand characters per second (or several times faster than he achieved on the 7094).

Cullum calculates that decoding is much faster and a rate of about 250,000 characters per second could be achieved on an IBM 360/75. This speed is determined by his assumption that the decompressed output is being written on a tape at the rate of 2^{21} bits per second. Cullum calculates that the CPU can decode at the rate of about 500,000 characters per second.

The core overhead is largely determined by the necessity of having the dictionary in core. For fast decoding, a pointer table to the start of each word in the dictionary is also required. For a text of approximately 2^{20} words, a dictionary containing approximately 2^{15} entries can be expected. The decoding tables will occupy about $3x2^{17}$ bytes of core (Cullum's figures). Thus the core overhead for decompression would be in the range 60-100 K words.

Theoretical studies by Cullum indicate that optimum compression using one ID requires a text of at least half a million words, while optimum compression with two (or more) levels of ID requires a text of two million words or more. This shows that the preceding figures for core overhead (based on 2^{20} words of text) are typical rather than minimal for one level of ID encoding.

References:

Cullum, R. D.

A METHOD FOR THE REMOVAL OF REDUNDANCY IN PRINTED TEXT

NTIS: AD 751 407, September 1962

For detailed description of Cullum's implementation of Huffman coding, see Variable Length Codes.

Binary Data Compression

Interrecord Word Comparison (Bit Mapping)

General:

1. Technique: This is a machine word based bit-mapping method. Repeated machine words in corresponding positions of consecutive records are suppressed. A bit-map for each record keeps track of which words in the record have been suppressed.

2. Data Types: This method is only effective for data with a high amount of redundancy between records. The routine is insensitive to the character code of the file. It works well on most files with formatted records.

3. File Types: The low overhead of this routine makes it suitable for all active and backup files on which it gives good compression. Files must be decompressed for any searches or changes, because a record cannot be decompressed without decompressing all previous records in the file. (For a modified version of the algorithm, only a portion of the preceding file must be decompressed - see Detailed Description below.

   It should be noted that a bit map of at least one word is added to every record, regardless of whether or not compression was achieved in that record. If there is very little record-to-record redundancy, it is possible for the method to expand the file, or to compress it very little but still require the overhead of decompression before the file can be used. This will not cause problems unless the data is not suited to this compression technique.

4. Relative Effectiveness: This method is one of the fastest and most effective available for files with formatted records, such as card

image program source files, printer format files and transactions files. It is usually about as effective as character repeat suppression but, since it handles the data as words or half-words rather than as characters, the processing overhead is less than with character repeat suppression. It is not normally as effective as Huffman coding, but the processing overhead is 1/3 to 1/6 that of Huffman coding. Huffman coding usually gives about a 20% higher compression factor than interrecord word comparison.

Detailed Description:

1. Algorithm: The basic algorithm compares a logical record with the previous record in the data file. If a word in the record is identical to the word which was in the same position in the previous record, that duplicated word is not written to the compressed file. A bit-map consisting of one word is added to the front of each compressed record. Each bit in the bit-map which is turned on specifies the position of a word in the compressed record which is present because it was found to be different from the corresponding word in the previous record. Thus, each compressed record has a one word bit-map followed by the nonredundant words of that record. Decompression is achieved by reading each record and using its bit-map to retrieve from the previous record those duplicated word(s) of data which must be inserted in the record.

   The basic algorithm just described can be enhanced in several obvious ways. Instead of writing one bit-map per record (which limits the length of the records that can be handled) a bit map can be written for every N words in the data record, where N is the number of bits in a machine word. For example, for a machine with 36 bit words, a bit map is written for every 36 words in the data record. This allows the algorithm to handle records longer than N (36 for our example) words. The bit maps can be collected at the start of the record or distributed through it, preceding the data words with which the bit map is associated.

Another useful enhancement is to allow compression against one or more "standard" records (such as a record of all blanks or a record of all zeros) as well as the previous record. The compression program calculates the compression possible against each of the candidate records and chooses the one which gives the most compression. The processing overhead is increased, but an improvement of 10-20% in the compression factor is obtained. The decoder is informed of the use of a certain "previous record" by reserving one or more bits in the bit map for this purpose. The number of input record words per bit map must be reduced accordingly. For records which use more than one bit map, each segment of the record can be compressed independently of other segments in the record, provided all the bit maps contain reserved bits to indicate the choice of "previous record".

The compression is normally improved by using the algorithm on a half-word basis instead of a full word basis. The number of bit maps will not double unless the records are all very long, e.g., on a 36-bit machine with a 6-bit character code, a card image file has 14-word records. Using a half-word based algorithm still only results in one bit map per record, and the compression factor can only improve (and usually does so by 10-20%). The processing time increases by about 40%, but is still low compared to any character based compression routine.

2. Tuning: Tuning this routine consists of selecting the exact algorithm to the used and, if "standard records" other than the previous record are candidates for use by the compressor, selecting these other "standard records." The programs should be written so that a user can change his standard records in the middle of processing a file. He can then use a small number of standard records provided by the program to simulate many standard records. This allows him to use a set of good standard records for each record type that occurs in his file, if he desires to do so. Managing these changes in the standard records must be the user's responsibility.

References:

Pollerman, T. A.

DATA COMPACTION SYSTEM; HSLUA Library No. GES 1075, Sept. 1972

Run Length Encoding

General:

1. Technique: Some data tends to be in the form of "sparse binary strings", or "low-density binary strings", i.e. strings that are mostly zeros with a few one bits scattered along the string, or mostly ones with a few zero bits. Such strings arise in some of the compression methods described in this document. There are many ways to compress such strings, and some of the possible methods will be described here.

2. Data Types: Low-density binary strings.

Detailed Description:

1. Fibonacci Codes: Fibonacci codes are variable length binary codes which represent the positive integers. They have the special property that no codeword has a run of s consecutive ones, where s is an integer dependent on the code (Kautz, 1965). We represent the binary string by a sequence of numbers giving the count of zero bits between successive one bits. These numbers can then be encoded with a Fibonacci code. The codewords can be separated by strings of s ones. Since no codeword contains s consecutive ones, this will allow the decoder to separate the codewords.

   An integer x is represented in a Fibonacci code of order s as $(C_n\ C_{n-1}\ \ldots\ C_1)$ where:

$$x = \sum_{i=1}^{n} C_i W_i$$

and

$$W_j = \begin{cases} 2^{j-1} & 1 \le j \le s \\ W_{j-1} + W_{j-2} + \ldots + W_{j-s} & j > s \end{cases}$$

For s = 2, the sequence of W_j's is:

$W_1=1$ $W_2=2$ $W_3=3$

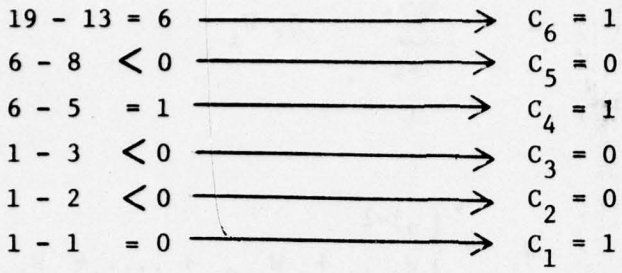
$W_4=5$ $W_5=8$ $W_6=13$ etc.

Using this code, we represent the integers as (including the 11 prefix)

0 ⟶ 110
1 ⟶ 111
2 ⟶ 1110
3 ⟶ 11100
4 ⟶ 11101
5 ⟶ 111000
6 ⟶ 111001
7 ⟶ 111010
8 ⟶ 1110000

A second order code (s=2) is best if the proportion of ones in the binary string is greater than 2%. A third order code is optimum if the proportion of ones is in the range 2% to .001%. Below .001% a code with s = 4 should be used. (Kautz, 1973)

An integer is encoded by diminishing it by whichever weights in the sequence W_n, W_{n-1},..., W_1 will not produce a negative result, where the integer is less than W_{n+1}. For example, for X = 19 and s=2 :



$$19 = 13 + 5 + 1$$
$$= W_6 + W_4 + W_1$$

Including the 11 prefix, 19 is represented by 11101001.

Example: Original String:

0010000000010100000010001101000000001

We represent the string as a series of counts of zeros between the ones. The ones themselves are omitted.

2/8/1/6/3/0/1/7/

The Fibonacci representations for these numbers are found. For an order 2 code, they are (omitting the 11 prefixes):

10/10000/1/1001/100/0/1/1010/

These codewords are now concatenated into one string, separated by 11 (a string of ones of length s, where s = 2).

10111000011111100111100110111111101011

The original string of 36 bits is now represented by a string of 37 bits. This is not surprising, because the original string contained 8 ones (over 20%) and this technique is intended for low

density (<10%) strings. Decoding is exactly the reverse of encoding. We know that (1) no codeword can contain two consecutive ones, (2) all codewords are separated by two consecutive ones, and (3) all codewords except 0 start with a 1 (if the 11 prefix is included, this means all codewords start with 111, and all except codewords for 0 and 1 start with 1110 from item (1)). This knowledge allows us to split the encoded string up into individual codewords, and these codewords are decoded into integers. (The reader should convince himself that this is true by decoding the example above.) The original string is reconstructed by writing strings of zeros of length specified by the integers, and inserting a 1 in between each string of zeros. A zero string of length 0 represents 11 in the original data -- see the example above. Fibonacci codes are one of the most compact ways to encode sparse binary strings.

2. Exponent-Mantissa Encoding: This is a alternative way to encode the counts of zeros between successive ones. We encode each integer as an r-digit exponent (r is fixed for the code) followed by a mantissa having a number of digits equal to the binary value of the exponent. For r = 2, the code is:

Integer	Exponent	Length of Mantissa	Code
0	00	0	00
1	01	1	010
2	01	1	011
3	10	2	1000
4	10	2	1001
5	10	2	1010
6	10	2	1011
7	11	3	11000
8	11	3	11001
.	.	.	.
.	.	.	.
.	.	.	.
14	11	3	11111

The maximum value that can be encoded is $2^{2^r} - 2$. The value of r must be chosen to accommodate the maximum zero run expected or else the largest codeword must be reserved to indicate that the zero string is not ended. For example, 23 could be encoded using the above code as 13 + 10, with 11111 representing 13 and indicating that the value of the following codeword is to be added to 13.

This coding technique is about as effective as Fibonacci encoding.

Example: We shall use the r = 2 code above to encode the string used in the Fibonacci code example. The counts of zeros are:

2/8/1/6/3/0/1/7/

In this method, no codeword separators are necessary because the exponent defines the codeword length. The codewords are just concatenated to give the encoded string. The encoded string is:

01111001010101110000001011000

The encoded string is 29 bits long, so this method has done better than Fibonacci coding on this particular string.

3. Asynchronous Compaction: This method applies the following transform to the original binary string:

$$00 \longrightarrow 0$$
$$01 \longrightarrow 11$$
$$1 \longrightarrow 10$$

The transform reduces the number of zeros in the string and is applied repeatedly until no further compaction results. Since the transform has a unique inverse, the original string can be reconstructed provided the number of times the transform was applied is

known. This could be supplied in a control field, along with the length of the compacted string.

This method works well when the original string is not very sparse. Long strings would have to be compressed in sections.

Example: We will use the same string as in the previous two examples. The slashes show how the string is partioned for encoding. Original string (length 36):

00/1/00/00/00/00/1/01/00/00/00/1/00/01/1/01/00/00/00/01/

Apply transform once (resulting length 28)

01/00/00/01/01/1/00/01/00/1/1/1/01/1/00/01/1/

Apply transform again (resulting length 29)

11001111100110101010111001110

The encoded string used is the one achieved after one application of the transform.

4. Block Encoding: This method encodes the counts of zeros between successive ones into b-bit blocks. Each integer $<2^b-1$ is encoded into its b-bit binary representation. Integers $\geq 2^b-1$ are coded as the b-bit code 11 ... 1 followed by the code for (integer - 2^b + 1). For b = 3, the code is:

0 ⟶ 000	1 ⟶ 001
2 ⟶ 010	3 ⟶ 011
4 ⟶ 100	5 ⟶ 101
6 ⟶ 110	7 ⟶ 111000
8 ⟶ 111001	9 ⟶ 111010
.	.
.	.
.	.
13 ⟶ 111110	14 ⟶ 111111000
15 ⟶ 111111001	

To find what value of b to use, we solve

$$p = 0.7\ b\ 2^{-b} \qquad (2.5.2.1)$$

for b, where p is the fraction of ones in the binary string. The less dense the string, the larger b should be. (Kautz, 1973). This method is very simple and quite effective.

Example: We use the same string as before. The zero runs in this string are:

2/8/1/6/3/0/1/7/

For this string, $p = \frac{8}{36} \doteq .22$

The following table gives the right hand side of equation 2.5.2.1.

b	$0.7\ b2^{-b}$
2	.35
3	.26
4	.18
5	.11
6	.07

We shall use b = 3.

The encoded string is:

0101110010011100110000001111000

The 36-bit string has been compressed to 30 bits. Note the encoding for 8.

5. Huffman Coding of Binary Strings: This method was used by Cullum (1972) who had very long strings to encode. First, he encoded the binary string into 24 codewords. These codewords, designated s_i, were:

   a. For $0 \leq i \leq 14$, s_i is a string of i zeros followed by a 1.

   b. For $i \geq 15$, s_i is a string of $15 \times 2^{i-15}$ zeros.

   This set of codewords allows very long zero runs to be encoded with only a few codewords. Each string was encoded into as few codewords as possible and then a Huffman code was derived for the codewords based on their use. The codewords s_i were replaced by their Huffman codeword and a compact description of the Huffman code (see Variable Length Codes) was added to the string.

   This method allowed efficient encoding of the long, very low density strings Cullum was using.

References:

Cullum, R. D.
A METHOD FOR THE REMOVAL OF REDUNDANCY IN PRINTED TEXT;
NTIS: AD 751 407, September 1972

Kautz, W. H.
FILE COMPRESSION FOR SIMPLE ASSOCIATIVE SEARCH;
November 1973, AD 771 314

Kautz, W. H.
FIBONACCI CODES FOR SYNCHRONIZATION CONTROL; IEEE Trans. on
Information Theory, V. IT-11, pp. 284-292, 1965

Kautz, W. H. and Singleton, R. C.
NON-RANDOM BINARY SUPERIMPOSED CODES; IEEE Trans. on Information
Theory, V. IT-10, pp. 363-377, 1964

Schaltwijk, J. P. M.
AN ALGORITHM FOR SOURCE CODING; IEEE Trans. on Information Theory,
V. IT-18, pp. 395-399 (1972)

COPAK Compressor

General:

1. Techniques: The COPAK (combined compressor) is a multistage compressor originally developed for use in the Self-Organizing Large Information Dissemination System (SOLID System). The alphanumeric compression component of COPAK is discussed here because of its widespread applicability.

   The COPAK alphanumeric compressor is a recursive bit-pattern recognition technique. It is fully automatic and stores all control information necessary for decompression with the compressed data. The input can be any arbitrary string of characters, numbers, codes or bits. Compression is achieved with two basic bit-pattern recognition routines (Type I and Type II) which operate in one of two modes (SLOW-MODE and FAST-MODE).

In Type I compression, a code word is substituted for a recurring bit-pattern in the data string to be compressed. A code word is a 6- or 8-bit (BCD or ASCII) character which does not appear in the input string. Depending on the data, (unused) punctuation and arithmetic characters may be available for use as code words. In Type II compression, code words are removed from the string, and their locations are indicated by bit-maps. A bit-map is a bit string with one bit for each character position in the input string. Each bit that is turned on discloses a position in the data string where the particular code word is to be inserted during decompression. e.g., The string "eat berries evenly" could be represented as "e(10000 1000100101000) at brris vnly". Note that the final three zeros in the bit map can be omitted. If we do this, and also use a bit map for r, the string becomes "e(100001000100101)r(000011) at bis vnly". In decoding, the substitution for r must precede the substitution for e, since the bit map for e has positions for r's in the string.

The control information stored with the compressed data string contains the code words, the bit patterns they replace, and the bit-maps for the code words if used. Thus, decompression is accomplished by stepping backwards through the control information of the string.

In SLOW-MODE compression, the input data is searched to determine the most frequently recurring bit-patterns to be replaced by code words. If the recurring bit-patterns are supplied to the COPAK compressor by the user, this step is eliminated, giving FAST-MODE compression. The differences in processing time of these two modes can be very great. SLOW-MODE can take several hundred times longer than FAST-MODE.

2. Data Types: The COPAK compressor is effective for nearly all types of data since it is based solely upon recognizing bit-patterns. The composition of the data is transparent to the compressor. Some tuning to the data is possible if the user supplies the recurring

bit-patterns to be suppressed (FAST-MODE). However, the SLOW-MODE procedure for compression is essentially a self-tuning mechanism in which the data is searched for recurrent patterns.

3. File Types: Any files which do not require frequent updating or searching are suitable for COPAK compression. Files which are updated or searched frequently would undergo the compression and decompression procedure constantly. The considerable processing overhead entailed is the prime consideration in deciding whether such files should be compressed.

4. Relative Effectiveness: The amount of core required to encode and decode data appears to be relatively small (2-4K). The processing time constitutes the greatest amount of overhead. This can be greatly reduced by operating in the FAST-MODE. Tests indicate that FAST-MODE is between 200 and 300 times faster than SLOW-MODE. With alphanumeric data, decompression is between 1.5 and 5.0 times faster than compression. A typical 2400 foot reel of business data (New York Personnel Records) was compressed at the rate of 9,000 bytes/second to give a compression factor of 3. Decompression occurred at the rate of 14,000 bytes/second. English and German language texts (Calvin's Nobel Prize Address) yielded compression factors between 1.7 and 3 at a throughput rate of 10K bytes/second. Experiments with about 250,000K bytes of information produced the following compression factors:

Business Data	2.2-4
Natural Language Texts	1.7-3.3
Machine Language Programs	1.3-2
Higher Language Programs (viz. COBOL, FORTRAN, etc.)	5-20

It should be noted that these compression factors are based on compressing an 8-bit byte. Therefore, a compression factor of 2 yields a file encoded with an average of 4 bits per character.

This routine is one of the most effective compression routines found in this study. Its main drawback is its high processing time. In general, its performance appears to be comparable with the pattern substitution routine described under Adaptive Character String Substitution.

Detailed Description:

Algorithm: There have been two separate implementations of the COPAK compressor. The first version is the one originally used and second has minor modifications which greatly simplify the processing. The first version is described in detail since it is better suited to a 36-bit word machine and is the more general algorithm. This first version was implemented on the experimental PILOT computer which had a word length of 68 bits. Attempting to efficiently use the long word length made the algorithm quite complex. The newer version, which was implemented on the System 360, is considerably less complicated because the 360 is byte oriented and has a shorter word length. The 360 version is different in the following ways:

- The number of binary units in a CODE WORD is fixed. The 8-bit byte is used as the coding basis. Thus there are 256 different possible code words. Fixing the length of the codeword greatly simplified the algorithm.

- A CORD contains up to 12 consecutive bytes (or code words) in the segment of information that is being compressed. CORDS, which are also called bit-patterns, found in the SLOW-MODE, are stored in the PCORD's table.

- The new version can operate either in the FAST or SLOW MODE. In the SLOW-MODE, the computer finds those cords which will yield savings. Each cord which makes a savings in the SLOW-MODE is stored in the array PCORDS. In the FAST-MODE only cords in the PCORDS table are used to compress the segment of information. There

are provisions in this new version for entering cords into PCORDS from cards, and for automatically going from the SLOW to the FAST-MODE after a specified number of segments of information have been compressed in the SLOW-MODE.

1. Definitions

It is supposed that a string of JI machine words of N1 bits is to be compressed. Here the string will be considered a single word (T) with N2 (= JI . N1) bits. The following definitions are associated with the procedures.

A Code contains 2^{CW} code words, each with CW bits. N1/CW must be a positive integer. Thus T can be regarded as a sequence of code words.

A Lexicon (TL) discloses which of the 2^{CW} code words have been used to achieve compression and in what manner.

A Cord (CD) contains R code words consecutive in the string T. N3 (=R .CW), the number of bits in the cord, cannot exceed N1; R is a positive integer.

A Bit Map (BM) of one of the 2^{CW} code words discloses the positions of that code word in the string T. Terminal zeros in a bit map are omitted, e.g., for T=101/011/010/101/010/100/010/000 the bit map of 101 is 1001, meaning that 101 is the first and fourth (and only these) of the successive code words of length CW in T. (Note: Bit maps are used only in Type II compression).

In Type I Compression, an unused code word is substituted for a cord.

In Type II Compression, code words are removed from the string, and their locations are designated by bit maps.

A string is irreducible if compression cannot be achieved.

2. Compression Procedure

Step I: The smallest value of CW is computed from N1 and the input information. For numeric information, the initial value of CW is the smallest number greater than four which divides N1 exactly. For alphanumeric information, CW is set equal to six, eight or nine. Six is for BCD data, eight is for ASCII data (16 or 32 bit word length) and nine is for ASCII data on 36-bit word machine.

Step II: The lexicon (TL) associated with the CW-bit code is constructed as follows. An array Y is constructed which consists of 2^{CW} consecutive machine words, initially set equal to zero, corresponding in a definite order to the 2^{CW} possible CW-bit binary words. The code words of string T (the input data string) are examined, and the Ith machine word in Y is used as an indicator of the presence of the Ith binary word (in the specified ordering) as a codeword in T. Then the zero words remaining in array Y are tallied in NRL, and the corresponding unused code words are stored in the array TL.

Example: Suppose that T (the input string) is in BCD code. Then CW=6 and $2^{CW} = 64$. The array Y is simply 64 consecutive words corresponding to the BCD characters octal 0 through octal 77. The string T is scanned, and each character found has its corresponding word in Y set to some non-zero value. After the string has been scanned, NRL = # zero words in Y and TL contains the BCD characters not found in T (there are NRL entries in TL).

Step III: The value of R is set to its maximum. The search begins with the longest cord, i.e. maximum R, so that shorter cords which

are contained in the long cord are not replaced by a code word first. If this did occur, the savings achieved would be smaller. However, it is realized that this somewhat arbitrary choice of beginning with maximum R may result in less savings in certain cases. (See section Common Phrase Suppression for an optimum solution to this problem.)

Step IV: NR, a counter, is set equal to zero. (Counts iterations of step V.)

Step Va: If NRL $\neq$ 0, Step Vb is executed. For NRL = 0, both R and NRL are set equal to one, and Step Vb is executed.

Step Vb: The N3-bit cord, CD_{N3} (where N3 = R . CW), is set equal to bits (NR . CW + 1) to (NR . CW + N3) in string T.

Example: Assume R=10, NR=0, CW=6 and the input data is BDC. If the input string is:

THE*BROWN*FOX****J*U*M*P*E*D****OVER*THE*BROWN*LOG

then N3=60 and CD_{60} = THE*BROWN*

Step Vc: A search of string T with CD_{N3} discloses whether or not a compression can be achieved. (The criterion for successful compression is that the number of bits which can be removed from the string must be greater than the number of bits which must be added to the string to permit automatic decompression.) In this searching procedure, if there is a match between CD_{N3} and the N3 bit cord in the string, the next attempted match will be with a cord in the string beginning CW bits (the code word length) further along. If R>1, compression is achieved by substituting the first unused code-word in TL for CD_{N3} wherever it occurs (Type I Compression). For R = 1, a bit map for CD_{N3} (here the code word) is constructed and the string

is compressed by removing the cord wherever it is found and NRL is decreased by one (Type II Compression). If a saving is achieved, a composite code word (CCW_1) in the array TL is constructed in one of the following forms:

Type I (Code word substituted for cord) (R>1)

Code Word (CW bits)	R (four bits)	Cord (N3 bits)

Type II (Bit map of code word) (R = 1)

Code Word (CW bits)	R (four bits)	No. bits in Bit Map (NB) (five bits)	Bit Map (NB bits)

Example: Continuing the previous example, we can use the symbol Q (which doesn't appear in the data) for CD_{60} and do Type 1 compression. The entry in TL is:

Q	10	THE*BROWN*
6 bits	4 bits	60 bits

and the compressed string is

QFOX****J*U*M*P*E*D****OVER*QLOG

The saving is 38 bits (18 characters eliminated minus 70 bits for the lexicon (TL) entry).

Step Vd: If compression was achieved, the above procedure beginning with Step IV is repeated with the compressed string. If no compression was achieved, NR is incremented by one and control goes to Step V. If all N3-bit cords (CD_{N3}) have been examined (i.e., NR . CW + N3 = N2), control goes to Step VI.

Step VI: R is decreased by one. If $R \geq 1$, control goes to Step IV; for R = 0, control goes to Step VII.

Example: Continuing the previous example, control will go to Step IV, Step V and Step VI until R=4. (An eyeball check indicates that no further compression will take place until R=4.) At R=4, NR=4 a substitution will take place for ****. We can assign the symbol A to stand for ****. The entry in TL is:

A	4	****
6 bits	4 bits	24 bits

and the compressed string is

QFOXAJ*U*M*P*E*DAOVER*QLOG

This substitution results in a saving of only 2 bits, since 6 characters (36 bits) are eliminated but 34 bits are used in the TL entry.

At R=1 a bit map for * will save a further 4 bits. The entry in TL is:

*	1	00000010101010100000001
6 bits	4 bits	22 bits

The compressed string is:

QFOXAJUMPEDAOVERQLOG.

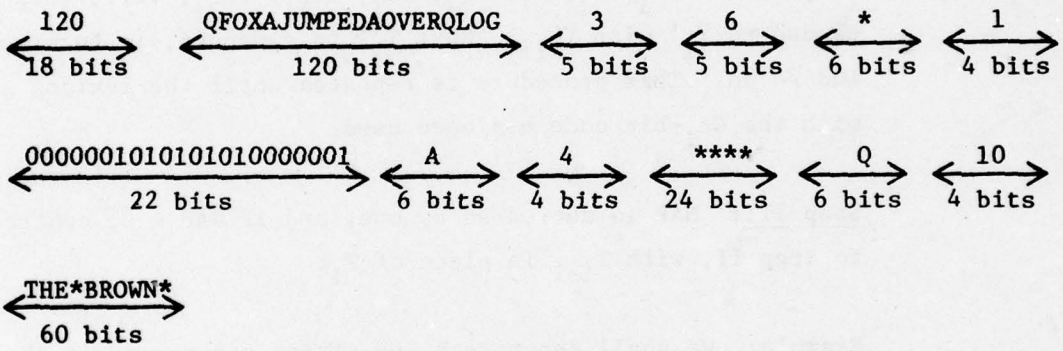
Step VII: If compression was achieved, control goes to Step VIII for the new string assembly. If no compression was achieved, CW is incremented by steps of one until N1/CW is again an integer. If N1=CW, the compression is complete and control goes to the calling system. Otherwise, control goes to Step II, where the lexicon associated with the new code is constructed.

Step VIII: The irreducible string (T_i) and its associated lexicon (TL) are combined in a compact self-defining string (I) thus:

$$BJI_i \quad T_i \quad ND_i \quad CW_i \quad TL_{1i} \quad TL_{2i} \ \ldots\ \ldots \quad TL_{ri} \ \ldots\ \ldots\ \ldots \quad (I)$$

Here, BJI_i, is the number of bits in the irreducible string (T_i). ND_i is the number of composite words in the lexicon for the code with CW_i bits; these composite words (TL_{1i}, TL_{2i}, etc.) are arranged in the reverse order from that in which they were constructed. NAP (the number of successful compressions with different strings link I) equals i. The new string I is processed, beginning with Step II, with the value of CW unaltered.

Example: The string I from out continuing example is:



CW incremented to 9 (if N1=36) and control returns to Step II with this string as the input data string. Notice that in our example the final string is 284 bits long compared with an input string of 300 bits. The small savings during compression (totalling 44 bits) more than offset the control fields BJI_i, ND_i and CW_i (a total of 28 bits).

This procedure (with newly defined strings) is repeated until no further saving can be achieved. (See Step VII). The final form of the compressed information consists of a single string like I

plus one word (NAP) which gives the number of values of CW for which compression was achieved.

3. Decompression Procedure

To regenerate the original string T the following procedure is executed:

Step I: If NAP = 0, no compression was achieved and control returns to the calling system, otherwise it goes to Step II.

Step II: The string T_i, with i = NAP, is expanded by using the ND_i composite words consecutively in the reverse order from that in which they were constructed. (They are arranged in this order in the compressed string.) This means that I is first split into its components BJI_i, T_i, and ND_i, CW_i, TL_{1i}, TL_{2i}, ...; then T_i is expanded to T_i' with TL_{1i}. Next T_i' is expanded, in turn, with TL_{2i} and so on. This procedure is repeated until the lexicon associated with the CE_i-bit code has been used.

Step III: NAP is decreased by one, and if NAP $\neq$ 0, control goes to Step II, with T_{i-1} in place of T_i.

Example: We shall decompress the string compressed in the previous section. We begin with NAP=1. We substitute in turn using the bit map, and then the composite words for A and Q. The resulting strings are:

a. Substitute using bit map.
QFOXAJ*U*M*P*E*DAOVER*QLOG

b. Substitute **** for A.
QFOX****J*U*M*P*E*D****OVER*QLOG

c. Substitute THE*BROWN* for Q.
THE*BROWN*FOX****J*U*M*P*E*D****OVER*THE*BROWN*LOG

In step III NAP = 0 and we terminate with the original string.

Notice how the compressed string is self-defining. BJI_i tells us the length of T_i. It is followed by fixed field entries for the number of substitutions (ND_i) and the length of the code words (CW_i). These define the TL_i terms. Within the TL_i terms, R_{ji} tells us whether the following field is a character string ($R_{ji} > 1$) or a fixed field count followed by a bit map ($R_{ji} = 1$). In the latter case, the fixed field count gives the length of the bit map.

4. Structure of Compressed Information

The compressed information consists of a single compact self-defining string, like I, with a mixture of fixed and variable fields. The lexicon of composite code words (TL_{ji}) associated with the code with CW_i bits and NAP = i, also contains fixed and variable field information thus:

Type I Compression ($R_{ji} = 1$)

ACW_{ji} R_{ji} CD_{ji}

Here ACW_{ji} is the jth code word associated with the CW_i-bit code and NAP=i. CD_{ji} is the cord which was replaced by ACW_{ji}; R_{ji} is the number of code words in cord CD_{ji}.

Type II Compression ($R_{ji} = 1$)

ACW_{ji} $R_{ji} = 1$ NB_{ji} BM_{ji}

Here BM_{ji} is the bit map associated with the code word ACW_{ji}. NB_{ji} indicates the number of bits in the bit map (BM_{ji}), which has no terminal zeros. The bit map actually defines the locations of the code word ACW_{ji} in the string.

The fixed fields in I, (BJI_i, ND_i, CW_i, R_{ji}, and NB_{ji}), are defined thus:

$\underline{BJI}_i$ (18 bits) is the number of bits in the string T_i.

$\underline{ND}_i$ (5 bits) is the number of composite code words in lexicon TL_{ji} associated with the CW_i-bit code.

$\underline{CW}_i$ (5 bits) is the number of bits in the code associated with NAP = i.

$\underline{R}_{ji}$ (4 bits) is the number of code words in the associated cord (CD_{ji}).

$\underline{NB}_{ji}$ (5 bits) indicates the number of bits in the bit map (BM_{ji}) if R_{ji}=1 and type II compression was achieved. Although this figure appears in more than one place in NBS-TN413, the author does not say why a bit map of only 31 bits is sufficient. If the number stood for the number of length CW characters in the bit map, then 5 bits would be sufficient for most files. In the 360 version of the compressor, the bit map length is given in bytes.

The variable fields (T_i, ACW_{ji}, and BM_{ji}) are defined next:

$\underline{T}_i$ Is the irreducible string obtained by compressing the string which precedes I. This may have been the original string (i=1) or may itself have been constructed from an irreducible string and its lexicon (i$>$1).

$\underline{ACW}_{ji}$ Is the jth code word associated with the CW_i bit code.

$\underline{CD}_{ji}$ Is the cord associated with ACW_{ji}.

$\underline{BM}_{ji}$ Is the bit map associated with the code word ACW_{ji}.

References:

DeMaine, P.A.D., Kloss, K. and Marron, B.A.

THE SOLID SYSTEM, II. NUMERIC COMPRESSION. THE SOLID SYSTEM, III. ALPHANUMERIC COMPRESSION, NBS Technical Note 413, August 15, 1967.

DeMaine, P.A.D., and Springer, G.K.

DETAILS OF THE SOLID SYSTEM; July 1968.
Computer Science Dept. University of Pennsylvania, 1968

DeMaine, P.A.D., and Springer, G.K.

THE COPAK COMPRESSOR
In: File Organization. Selected papers from File 68-
An I.A.G. Conference.
Swets and Zetlinger, Amsterdam, N.V. pp. 149-159, 1969

Marron, B.A. and DeMaine, P.A.D.

AUTOMATIC DATA COMPRESSION
Comm. of the ACM, V. 10, pp. 711-715, 1967

Irreversible Compression Codes

Introduction

The codes described in this section are intended for use in information retrieval. They are suitable for creating directories to large data files. The usual problem is to transform sets of variable length words into fixed length codes that will maximally preserve word to word discrimination. The encoding is specified by an algorithm which is applied to the file entry to derive the directory and to the input query. Code tables are not used. The different codes described have specific uses and careful selection is necessary to ensure that the code chosen has the desired attributes. The following four examples are cases in which these codes are useful. The codes mentioned are discussed individually in the following sections.

1. Create a file key for extraction of words in approximate file order. A typical code construction rule is to take the first six letters.

2. Create a file key for extraction of records under conditions of uncertainty of spelling (the so-called airline reservation problem). Typical codes used are Vowel Elimination and Soundex.

3. Create a file key for extraction of records from accurate input, with the objective of maximum discrimination of similar entries (catalog searching problem). Suitable codes are Recursive Decomposition Codes and Transition Distance Codes.

4. Create a file key for human readability and high word-to-word discrimination. Alphacheck Coding or truncation plus a terminal check are suitable codes.

Good discrimination in these codes is achieved by equalizing the use of the letters in the alphabet through the use of some randomizing algorithm to map the source letters into the code letters. Letter selection codes cannot do this well because they cannot increase the usage of the lower frequency characters.

Transition Distance Coding

General:

1. Technique: Transforms a variable length word into a shorter fixed length alphabetic or alphanumeric string such that there is a very low probability that different words will map into the same code-word. The code is formed from the modulo product of primes associated with transition distances of (i.e., distances between) permuted letters. It is an irreversible encoding.

2. Data Types: Alphabetic strings. Algorithm is simple to modify to cope with alphanumeric data.

3. File Types: The code is intended to create a file key with maximum discrimination between similar entries. The key will not be meaningful to a human reader.

4. Relative Effectiveness: Converts variable length input words to fixed length code words with more discrimination than the other methods described in this chapter. A relatively complex algorithm is used, and it is not suitable for manual calculation.

Detailed Description:

Algorithm:

1. Permute the characters of the natural language word. Take the middle letter (or the letter to the right of middle for words with an even number of letters), the first, the last, the second, the next-to-last, etc.

EXAMPLE: JOHNSEN ⟶ NJNOEHS

2. Determine the transition distances of the characters as follows. Assign letters a position value corresponding to their normal alphabetic positions (A=1, B=2, etc.) except assign 0 to Z. Measure distance unidirectionally in alphabetic order and cyclically from Z to A. Thus BX has transition distance 22 and XB a transition distance 4 (note that 22 + 4 = 26).

EXAMPLE: Continuing the processing of JOHNSEN

NJNOEHS ⟶ (14,10,14,15,5,8,19) letter numbers

⟶ (22,4,1,16,3,11) distances

3. Associate with each transition distance a corresponding prime number from table 1. The primes in the table start at 5 so that they are all relatively prime to 26 and 36.

EXAMPLE:

(22,4,1,16,3,11) ⟶ (89,13,5,61,11,41)

distances primes

4. Multiply these primes, modulo the capacity of the computer (i.e., integer multiply ignoring overflow).

EXAMPLE: Assume a 16-bit machine. The maximum integer representation possible is:

$$2^{16} - 1 = 65,535$$

$$89 \times 13 \times 5 \times 61 \bmod (2^{16}-1) = 352,885 \bmod (2^{16}-1)$$

$$= 25,210$$

$$25,210 \times 11 \bmod (2^{16}-1) = 277,310 \bmod (2^{16}-1)$$

$$= 15,170$$

$$15,170 \times 41 \bmod (2^{16}-1) = 621,970 \bmod (2^{16}-1)$$

$$= 32,155$$

5. Express the number derived above as an integer base 26 (alphabetic form) or base 36 (alphanumeric form) using a 4-digit code. In the case of alphabetic representation, use the letters to represent the numbers of their original position (A=1, B=2, etc.), and use Z as zero. In alphanumeric form, use the digits 0 to 9 to represent this range, and use the letters A to Z to represent the range from 10 to 35.

EXAMPLE: We will use the alphanumeric form and use a 3-digit code (i.e., ignore the multiplier of 36^3).

$$32{,}155 = 24 \times 36^2 + 29 \times 36^1 + 7 \times 36^0$$

$$(24,29,7) \longrightarrow (O,T,7)$$

The resulting code is OT7. Ignoring the multiplier of 36^3 results in very little loss in discrimination since it can be only 0 or 1. To obtain a 4-digit alphabetic code, the number at the end of step 4 is expressed as:

$$32{,}155 = 1 \times 26^3 + 21 \times 26^2 + 14 \times 26 + 19$$

The code is AUNS.

The range of 4-digit alphabetic representation extends to $(26^4 - 1) = 456{,}975$; the range of 4-digit alphanumeric representation extends to $(36^4 - 1) = 1{,}679{,}615$. Hence, the 4-bit alphabetic representation is sufficient for up to 18 bit machines (with little loss for 19 bit machines) and the 4-bit alphanumeric representation is sufficient for up to 20-bit machines.

Table 1

Letter Positions and Primes Used In

Transition Distance Coding and Alphacheck Coding

Letter	Letter Position and Distance Value	Prime Number
A	1	5
B	2	7
C	3	11
D	4	13
E	5	17
F	6	19
G	7	23
H	8	29
I	9	31
J	10	37
K	11	41
L	12	43
M	13	47
N	14	53
O	15	59
P	16	61
Q	17	67
R	18	71
S	19	73
T	20	79
U	21	83
V	22	89
W	23	97
X	24	101
Y	25	103
Z	0	107

References:

Nugent, W. R.
COMPRESSION WORD CODING TECHNIQUES FOR INFORMATION RETRIEVAL:
Jnl. Library Automation, V. 1, pp. 250-260, 1968

Alphacheck Coding

General:

1. Technique: This is a compromise coding technique. It attempts to maintain both readability and randomness. The first five characters of the key are retained and a sixth check character is generated using a method very similar to Transition Distance Coding (see Transition Distance Coding section).

2. Data Types: Alphabetic strings or alphanumeric strings.

3. File types: The code is intended to create a file key where both readability and randomness are desired.

4. Relative Effectiveness: The code has a 50% chance of uniquely resolving in the alphacheck symbol seven otherwise identical five-letter truncations of source words.

Detailed Description:

Algorithm: The algorithm to derive the alphacheck symbol is similar to Transition Distance Coding (TDC) which was described in Section 2.6.2. The steps are:

1. If word is six letters or less, take whole word; otherwise, take first five letters and compute an Alphacheck character for the sixth, based on omitted letters.

EXAMPLE: JOHNSTEN

First 5 letters: JOHNS, Remainder: TEN

2. Take transition distances of the omitted letters (as in TDC).

EXAMPLE:

TEN $\longrightarrow$ (20,5,14) $\longrightarrow$ (11,9)

positions distances

3. Associate with each transition distance a corresponding prime number (as in TDC). If only one transition distance exists, additionally associate prime numbers with the remaining letters. If only two transition distances exist, additionally associate a prime number with the last letter.

EXAMPLE: Use Table 2.6.1

11 $\longrightarrow$ 41, 9 $\longrightarrow$ 31, N $\longrightarrow$ 53

The prime for N is used because there are only two transition distances.

4. Multiply these primes, modulo the capacity of the computer (as in TDC).

EXAMPLE: For a 16-bit computer,

$41 \times 31 \times 53 \bmod (2^{16} - 1) = 67{,}363 \bmod (65{,}535)$

$= 1828$

5. Convert to alphanumeric form in 1 symbol, modulo 36, in which

0 $\longrightarrow$ 1, ..., 9 $\longrightarrow$ 9, 10 $\longrightarrow$ A, 11 $\longrightarrow$ B, ..., 35 $\longrightarrow$ Z.

EXAMPLE: $1828 \bmod (36) = 28$

Check symbol is S

JOHNSTEN $\longrightarrow$ JOHNSS

References:

Nugent, W. R.
COMPRESSION WORD CODING TECHNIQUES FOR INFORMATION RETRIEVAL;
Jnl. Library Automation, V. 1, pp. 250-260, 1968.

Recursive Decomposition Coding

General:

1. Technique: This method is an alternative to Transition Distance Coding (see Transition Distance Coding section). The code uses a frequency ordering of the letters, and selection or rejection of a particular letter is based on that letter's relative order in the table with respect to the previous letter.

   The frequency ordering used may be any of the standard ones, such as that contained in Pratt (1939). The resolution of the code is not sensitive to minor variations in the frequency ordering.

2. Data Types: Alphabetic strings. Using an appropriate frequency ordering would allow alphanumeric strings to be encoded.

3. File Types: The code is intended to create a file key with maximum discrimination between similar entries. The key will not be meaningful to human readers.

4. Relative Effectiveness: The prime advantages of the method are its computational simplicity and its resolution. The elimination requires only table lookup and no multiplications, and the compression is readily done manually. The resolution is apparently as good as one can get with a selected letter compression code. If effectively flattens the high portions of the letter frequency curve, though,

unlike a randomizing code such as Transition Distance Coding, it cannot totally equalize the distribution. The resolution, however, is quite good. Specifically, in a test of 4,862 words (chosen from the secretary's handbook "20,000 Words"), only 30 of the 6-letter ciphers (about 0.61%) were nonunique and of nonunique ciphers all were simple pairs except for one instance of three occureences. The method compresses quickly; since all noninitial letters have a .5 probability of being retained, the expected length, L, of an n letter word after r recursions is:

$$L = 1 + \frac{n-1}{2^r}$$

This indicates that a 43-letter word may be expected to compress to six letters in three recursions.

Detailed Description:

Algorithm: - Choose some frequency ordering of letters, such as Pratt's (1939):

ETAONRISHDLFCMUGYPWBVKXJQZ

The algorithm is: If a source word is longer than six letters, select the first letter and subsequent letters of lesser or equal ordering that the prior letter, and continue the process recursively until six letters remain. Words of six letters or less are reproduced in full and filled out with null symbols, where necessary, until a total of six characters is reached. For words of more than six letters, the algorithm may be stated in steps:

1. Select the second/next letter in the word.

2. Compare this letter with the preceding letter, even if the preceding letter is marked for deletion. If the preceding letter is to the right of the selected letter in the frequency ordering, mark the selected letter for deletion. (Note that if the two letters are the

same, the selected letter is not marked for deletion.)

3. If the selected letter is not the last letter in the word, go to step 1.

4. If there are no letters marked for deletion, truncate the string to six letters. (Use of this step will be extremely rare.)

5. Delete marked letters from left to right until only six letters remain. If all marked letters are deleted and more than six letters remain, go to step 1. Otherwise end.

Several examples will illustrate the system. Omitted letters are shown bracketed, and successive cycles are shown by arrows.

1. B[I]B[LIO]G[RA]P[H]ER ⟶ BBGPER
2. I[N]F[O]RM[AT]I[O]N ⟶ IFRMIN
3. SH[A]K[E]SP[E]AR[E] ⟶ SHK[S]PAR ⟶ SHKPAR
4. SMITH ⟶ SMITH ⊬
5. K[IN]G[S]F[O]RD[-S]M[IT]H ⟶ K[G]FRDMH ⟶ KFRDMH
6. K[R]ISH[NA]M[O]OR[T]H[I] ⟶ K[I]SHM[O]RH ⟶ KSHMRH

In some very rare cases, an emerging cipher may have more than six letters in descending sequence, so that it will not decompose further. In such cases the final letters are eliminated until six remain as stated in step 4.

Most words, however, will reduce in one or two cycles. In a test of 55,000 words only one was found requiring four cycles. A few extreme cases do exist, however: the longest ever found required six cycles:

AD-A035 786 PRC DATA SERVICES CO MCLEAN VA F/G 9/2
A REVIEW OF DATA COMPRESSION ALGORITHMS.(U)
MAY 76 C HOLBROW, J MCNEMAR, P STONEBURNER DCA100-73-C-0015
UNCLASSIFIED CCTC-TM-122-76 NL

2 OF 2
ADA035786

END

DATE
FILMED
3-77

7. AN[T]ID[I]S[E]S[T]AB[LI]SHM[E]N[T]ARI[A]NISM ⟶
ANID[S]S[A]B[S]HM[NA]RI[N]ISM ⟶
ANID[S]B[H]M[R]IISM ⟶
ANIDB[MI]ISM ⟶
ANIDB[I]SM ⟶
ANIDB[S]M ⟶
ANIDBM

Even Mary Poppin's sesquipedalian ecphonesis crumbles to six letters in three recursions:

8. SUP[E]RC[A]L[I]F[RA]G[I]L[I]S[T]IC[E]X[PIA]L[I]
D[O]C[O]U[S] ⟶
SUPRC[L]FG[LSI]CX[LD]CU ⟶
SUP[CF]G[C]X[C]U ⟶
SUPGXU

References:

Nugent, W. R.
COMPRESSION WORD CODING TECHNIQUES FOR INFORMATION RETRIEVAL;
Jnl. Library Automation, V. 1, pp. 250-260, 1968.

Pratt, F.
SECRET AND URGENT, THE STORY OF CODES AND CIPHERS;
Blue Ribbon Books, New York, 1939

The Soundex Code

General:

1. Technique: The Soundex Code, attributed to Remington Rand, is a phonetic code that tends to create identical codes for similar sounding names. It is useful for name searching under conditions of uncertain spelling.

2. Data Types: Proper Names.

3. File Types: Files with a key of proper names.

Detailed Description:

Algorithm - The code has five steps:

1. Retain first letter of name as first letter code.

2. Eliminate vowels, plus W, H, and Y.

3. Eliminate the second consonant of a double consonant pair. e.g., JTTK ⟶ JTK

4. Replace the following letters by numbers (except when the letter is the first letter of the name):

B,P,F,V	1
C,G,J,K,Q,S,X,Z,SC,CK	2
D,T	3
L	4
M,N	5
R	6

5. Take the first three or four symbols, and add zeros if insufficient phonetic sounds.

EXAMPLE:

JOHNSEN ⟶ JNSN ⟶ J525 ⟶ J52
JOHNSON ⟶ JNSN ⟶ J525 ⟶ J52
JOHNSTON ⟶ JNSTN ⟶ J5235 ⟶ J52
JOHNSTONE ⟶ JNSTN ⟶ J5235 ⟶ J52

References:

Nugent, W. R.
COMPRESSION WORD CODING TECHNIQUES FOR INFORMATION RETRIEVAL;
Jnl. Library Automation, V. 1, pp. 250-260, 1968

Ruecking's Bibliographic Retrieval Method

General:

1. Technique: This method was developed in an attempt to automate the searching of the card catalogue of a large library. A large library may contain several million volumes. These are shelved according to the number assigned to them. Most large libraries use the Library of Congress numbering system. To find a book or to determine its status if it is not on the shelves, one needs to know the number. This is usually found by searching the card catalogue. The catalog contains several cards for each book and is arranged alphabetically. There is one card for each author and at least one card for the title. There may be several title cards depending on whether the title splits into parts. For example, a title such as "SIGOPS 1969: Progress in Signal Processing" may have title cards under "SIGOPS" and "Progress in Signal Processing."

   The problem in searching such a large catalog is that reference data (author, title, publisher, date of publication, edition number,

or, for periodicals, the journal title and volume) may be inaccurate or incomplete. Many bibliographies cite only an author last name and title. Typographical errors can cause spelling mistakes and volume numbers or dates may be incorrect. The person conducting the search can often compensate for such errors in the reference by checking possible alternatives and determining which card best matches the supplied information. This involves considerable diligence, judgment and experience on the part of the person conducting the search. Ruecking attempted to automate this search process. He states his hypothesis thus:

"It is hypothecated that retrieval of correct bibliographic entries can be obtained from unverified, user-supplied input data through the use of a code derived from the compression of author and title information supplied by the user. It is assumed that a similar code is provided for all entries of the data base using the same compression rules for main and added entry, title and added title information.

It is further hypothecated that use of weighting factors for individual segments of the code will provide accurate retrieval in those cases when exact matching does not occur."

2. Data Types: User supplied bibliographic references. Only author and title were automated in Ruecking's experiment but the inclusion of date, publisher and edition would be simple extensions to implement.

3. File Types: The file to be searched is assumed to be a compressed file of library card catalog information containing up to several million items.

4. Relative Effectiveness: The algorithm appears to have promise for this very specialized application, but it needs considerable refinement before it can be used as a routine tool. Whether it can ever totally replace manual searches is open to serious doubt. See the section

below titled Published Performance for details of the tests conducted.

Detailed Description:

Algorithm: The following words are deleted from the title: a, an, and, by, if, in, of, on, the, to. Each remaining word in the title is compressed to four characters. Four 4-character abbreviations are retained for the compressed title. The rules for compressing the title words are:

1. Delete all suffixes and inflections which terminate a title word. (see Table 2)

2. Delete all vowels from the end of the stem until a consonant is located or the stem is reduced to four characters.

3. If the stem is longer than four characters, take the final consonant string and, if this is less than four characters, fill it out to four characters with letters from the initial character string.

EXAMPLE 1: "BUILDING LIBRARY COLLECTIONS"

Step 1 yields "BUILD LIBR COLLECT"
Step 2 gives no change, since all stems end in consonants.
Step 3 yields "BULD LIBR COCT"
Final result is BULDLIBRCOCTbbbb

EXAMPLE 2: "ANCIENT HUNTERS OF THE FAR WEST"

Step 1 yields "ANCI HUNT FAR WEST"

Note that even though IENT is in table 2.6.2, the i is retained to keep the stem four characters long. ENT is also in the table.
No further compression of this title is needed.
The final result is ANCIHUNTFARbWEST

Table 2 - Deleted Suffixes and Inflections

-ic	-ive	-in	-et
-ed	-ative	-ain	-est
-aged	-ize	-on	-ant
-oid	-ing	-ion	-ent
-ance	-og	-ation	-ient
-ence	-log	-ship	-ment
-ide	-olog	-er	-ist
-age	-ish	-or	-y
-able	-al	-s	-ency
-ible	-ial	-es	-ogy
-ite	-ful	-ies	-ology
-ine	-ism	-ives	-ly
-ure	-um	-ess	-ry
-ise	-ium	-us	-ary
-ose	-an	-ous	-ory
-ate	-ian	-ious	-ity

EXAMPLE 3: "ANALYZING PHILOSOPHICAL ARGUMENTS"

Step 1 yields "ANALYZ PHILOSOPH ARGU"

Step 2 gives no change

Step 3 yields "ANAZ PHPH ARGU"

The final result is ANAZPHPHARGUbbbb

Note that Y is regarded as a vowel in step 3.

Author names (both personal and corporate) are compressed by the algorithm above, with some modifications. Meeting names (symposium, conference, etc.) are considered as a secondary subset of nonsignificant words. Names of organizational divisions (bureau, department, etc.) are treated similarly.

Rules 1 and 2 are applied to corporate names but not personal names, whereas rule 3 is applied to both types of author names. Only the last name of an author is compressed.

EXAMPLE 1: POURADE, RICHARD F.

Only the last name is compressed. Steps 1 and 2 are not applied to a personal name. Step 3 gives POUD.

EXAMPLE 2: HEINRICHS

Step 3 gives HCHS

Searching is accomplished by comparing the compressed bibliographic information supplied by the user to entries in the compressed catalog file. A "retrieval value" is calculated based on how well the two items being compared agree. If the retrieval value is greater than or equal to a threshold, a match is declared and the search terminates.

The rules for calculating the threshold are not described clearly. They appear to be: For a title which compresses to three or four 4-character words use a threshold of 12, for a title which compresses to two 4-character words use a threshold of 10, and for a single 4-character word compressed title use 6.

The retrieval value is calculated by adding 4 to the retrieve total for every 4-character word in which the query and catalog title entries agree, and adding 2 to the total for every agreement in the author field. The search algorithm reorders the title words in an attempt to obtain a match and raises the threshold by an unspecified amount when it does so.

EXAMPLE 1

Catalog entry: ANALYZING PHILOSOPHICAL ARGUMENTS,
MCGREAL

Query entry: ANALYZING PHILOSOPHICAL ARGUMENTS,
MCGREAF

Compressed catalog entry: ANAZ PHPH ARGU MCGL
Compressed query entry: ANAZ PHPH ARGU MCGF

Threshold = 12
Agreement in 3 title fields gives retrieve contribution of 12.
Disagreement in author field gives retrieve contribution of 0.
Total retrieve value = 12
Retrieve is successful (retrieve value $\geq$ threshold)

EXAMPLE 2

Catalog entry: THE AMERICAN THEATER TODAY, DOWNER
Query entry: THE AMERICAN THEATRE TODAY, DOWNER

Compressed catalog entry: AMER THET TODA DOWR
Compressed query entry: AMER THTR TODA DOWR

Threshold = 12
Agreement in 2 title fields gives contribution of 8.
Agreement in author fields gives contribution of 2.
Total retrieve contribution = 10
Retrieve fails.

This example illustrates the problems of the method as described here. Webster lists both spellings of "theater" as correct. The thresholds seem high, and there does not appear to be a good reason to weigh the retrieve contributions from the author and title fields differently. In catalog searching, false hits are much less severe faults than retrieve failures, since in the latter case a full manual search must be undertaken to verify that the reference is not present. If the search procedure lists all matches found, the false hits can readily be eliminated by a short manual inspection of the entries.

Published Performance: Ruecking used a source file containing 4,800 titles. His query file contained 2,874 items. Of these, 1,392 were actually in the data base of 4,800 titles. The search algorithm recorded 1,184 correct hits and 16 false hits. Thus it correctly located 1,184 titles, failed to locate 192 titles and incorrectly located 16 titles. In this test the algorithm was successful about 85% of the time and its accuracy was 98.7%. The accuracy could have been improved to over 99% by rectifying some oversights in the compression routines. Ruecking concluded that the effect of spelling errors had been reduced by 30% and that the use of added author and title entries was essential to good performance of the algorithm.

A severe limitation of Ruecking's experiment was the small size of his source file (less than 5,000 titles). As the size of the source file grows it is inevitable that more false hits will be recorded, reducing the accuracy.

Lipetz et al ran a small scale test of Ruecking's algorithm on a large source file (3.5 million books). For a "rigidly randomized" sample of library users, they recorded the original bibliographic information available to the searcher. They selected the 126 manual catalog searches in the sample which had been successful. The original bibliographic information was hand encoded according to Ruecking's algorithm and compared with the hand-encoded catalog card information. They could then determine whether a machine search would have successfully retrieved the correct catalog entry or not. This was all that could be determined - no attempt was made to see if false hits were likely for those cases where the correct card would not have been retrieved.

Of the 126 searches in Lipetz's sample, Ruecking's algorithm would have been successful in 88 cases. This is a recall rate of 70%. Some of the 126 searches involved foreign language references. However, 106 searches were for English language references and 77 of these were retrieved - a recall rate of 73%. The compression coding had "healed" mismatches of data and allowed retrieval in 11 cases out of the 49 cases where there were data mismatches. The recall rate could have been raised to 76% making some simple modifications to Ruecking's algorithm.

References:

Lipetz, B., Stangl, P. and Taylor, K.F.

PERFORMANCE OF RUECKING'S WORD COMPRESSION METHOD WHEN APPLIED TO MACHINE RETRIEVAL FROM A LIBRARY CATALOG; Jnl. Library Automation, V. 2, pp. 266-271, 1969.

Ruecking, F. H., Jr.

BIBLIOGRAPHIC RETRIEVAL FROM BIBLIOGRAPHIC INPUT: THE HYPOTHESIS AND CONSTRUCTION OF A TEST: Jnl. Library Automation, V. 1, pp. 227-238, 1968.

VARIABLE LENGTH CODES

Introduction

Several of the compression techniques discussed in this document can be implemented with either fixed length codes or variable length codes. If the statistics describing the usage of the source alphabet are known accurately, the use of a correctly chosen variable length code will always produce additional compression over that obtainable with a fixed length code, unless the source letters are all used with equal frequency. The use of a variable length code involves additional processing overhead in the encoding and decoding operations. Whether this extra processing time is worth the compression achieved is a matter for the user to decide. The improvement in compression tends to be greater if the probability distribution of the source alphabet is highly skewed.

If the source letters are used with about the same frequency, little extra compression will be achieved by using a variable length code. From this point of view, character string substitution with a fixed length code may be regarded as a method of transforming the original source into one which has a reasonably uniform probability distribution of its source alphabet. The character strings which are mapped into a single codeword in the fixed length code are chosen so that the probability distribution of the codewords is as uniform as possible.

The choice of a source alphabet depends in part on the number of codewords available. Within limits, a large source alphabet will give more compression than a small one. (Schwartz and Kleiboemer, 1967) The extra source symbols (character strings to be encoded into one codeword) must be chosen to maximize the compression. Choosing an appropriate source alphabet is separate from but related to the problem of choosing a code to use. In this chapter, some of the available variable length codes will be described. It will be assumed that the source alphabet has already been chosen and that a sample of the file has been used to generate a probability distribution for this source alphabet.

The codes considered here will be instantaneously decodable codes, or codes with the prefix property (often just called *prefix* codes). A binary string is a prefix of another binary string if the second string is just the first string with some digits added on the end of it. For example, the prefixes of 1001101 are 1, 10, 100, 1001, 10011, and 100110. If a prefix code contained 1001101 as a codeword, then none of its prefixes would be codewords and 1001101 would not be a prefix of any other codeword. This restriction means that a codeword can be recognized as soon as it is received - there is no decoding delay. Nonprefix codes can be found which will give more compression than prefix codes, but there is no systematic way to construct them. Decoding them is also more complex because a delay is usually involved -- the decoder cannot decode a received codeword until it has checked the following received digits to make sure that the codeword recognized is not in fact the prefix of a longer codeword. For some codes, it is not always possible to decode them because there exist sequences for which the delay is infinite. Because there is no systematic way to construct a very effective nonprefix code with a known maximum delay, only prefix codes will be considered further.

The best known variable length compression codes are Huffman codes. (Huffman, 1952; Abramson, 1963) These codes are optimal in the sense that for a given source alphabet with a given probability distribution, Huffman codes provide the maximum compression achievable by a prefix code. (Note that a different source alphabet for the same source might give better compression. This is why the source alphabet must be chosen carefully.) There are some tricks that can be used to reduce considerably the overhead involved in using Huffman codes. These will be described in the section titled Huffman Codes.

Although Huffman codes are optimum, there are other codes which are only slightly less effective and which present some advantages. Gilbert and Moore described a way to generate a code which is "alphabetical" in the sense that the codeword for source letter j represents a larger binary number than the codeword for source letter i if $j > i$. For example, the codewords for b and c may be 10 and 110 respectively. The Gilbert-Moore codes are "strongly alphabetical" in the sense that sorting the left-justified encoded words into numerically increasing order is equivalent to alphabetically ordering the source words.

See Gilbert-Moore Codes for a discussion of these codes.

It is possibly to modify a Huffman code in ways which make it easier to decode, and only affect the compression slightly. The two principal modifications are:

1. Limit the maximum length of the codewords.

2. Make all codewords of a given length have the same prefix.

For example, the codewords may be limited to be 12 bits or shorter, and all codewords longer than 4 bits may be chosen so that the first 4 bits are characteristic of the length of the codeword. The first action in decoding will be to examine the first 4 bits of the codeword and jump to the appropriate decoding table. Provided these two restrictions are not used too stringently, compression will be nearly optimum. The effectiveness of the second restriction depends on the decoding algorithm used. For the efficient decoding algorithm in the section entitled Huffman Codes, this restriction does not provide a useful gain in decoding speed.

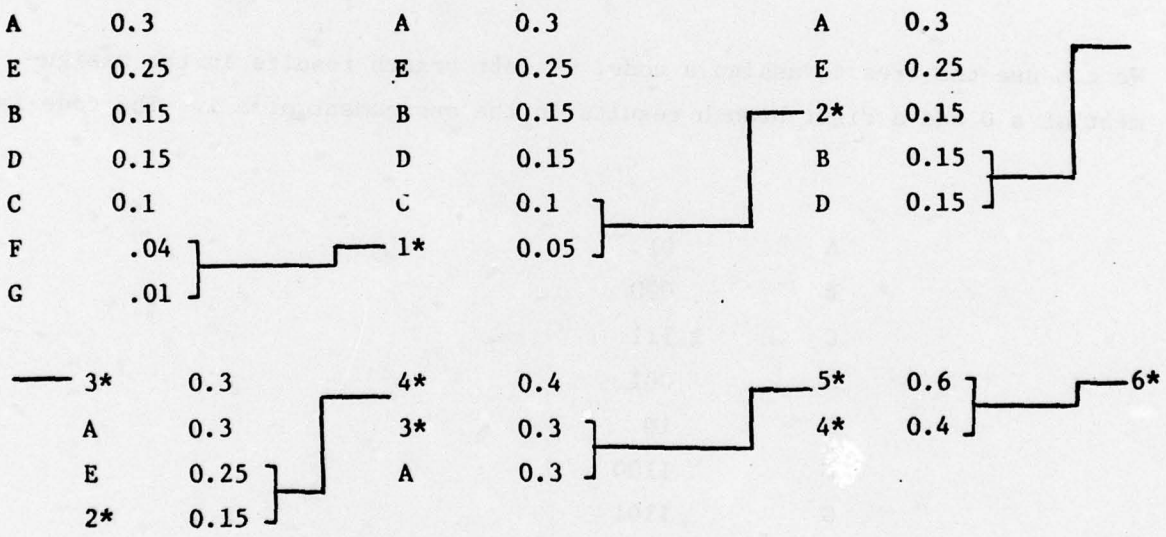
The remainder of this chapter considers Huffman codes and their implementation, modifications to Huffman codes, state dependent coding and, finally, Gilber-Moore alphabetical codes. Some of the techniques discussed here may be protected by patents (see references).

Huffman Codes

The algorithm for generating a Huffman Code is most easily understood by following an example. Assume a 7-letter alphabet, with the following probability distribution:

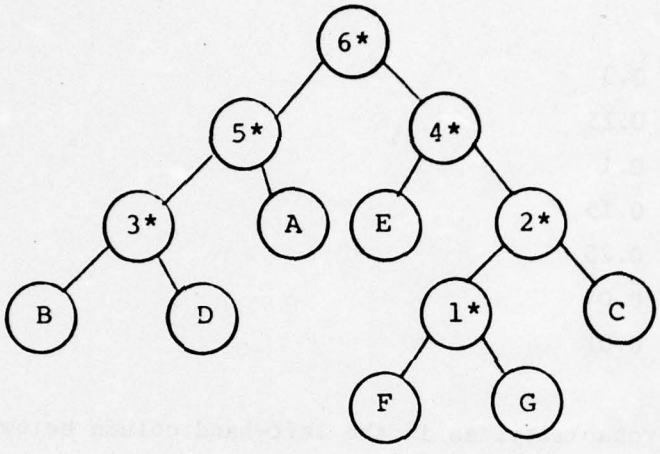
A	0.3
B	0.15
C	0.1
D	0.15
E	0.25
F	0.04
G	0.01

Sort the alphabet by probability, as in the left-hand column below:



Now merge the two states (letters) at the bottom of the list to form a new state with probability equal to the sum of the probabilities of the two merged states. Place this new state in its correct place in the alphabet according to its probability. If it has probability equal to another state in the list, place the new state above the old state(s) which have equal probability. Schwartz (1964) shows that this will minimize the codeword lengths.

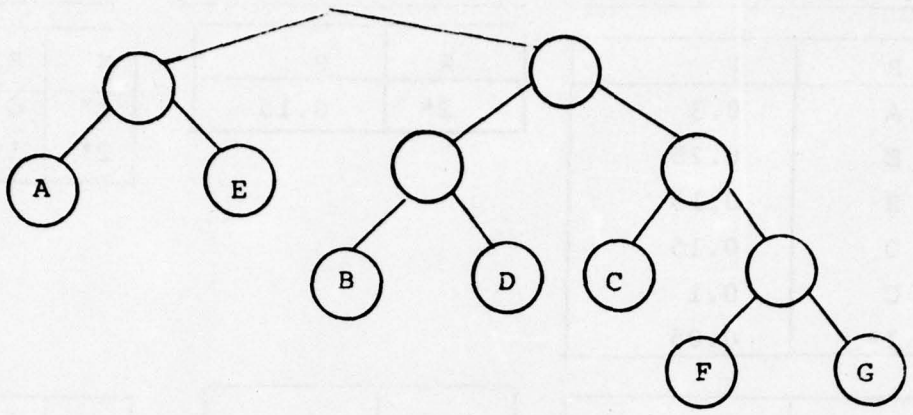
Continue the merging process on the bottom two states in the list until two are left, as shown in the example. This merging process is shown below in terms of a binary tree.



We can use the tree to assign a code. *A left branch results in the assign*ment of a 0 and a right branch results in the assignment of a 1. The code is:

A	01
B	000
C	111
D	001
E	10
F	1100
G	1101

For decoding purposes, it is more convenient to have a different code with all the codewords of the same length adjacent to each other, and with the length of the codewords increasing from left to right across the tree. The rearranged tree and code are:



A	00
B	100
C	110
D	101
E	01
F	1110
G	1111

In going from the original tree to the new tree, the only information that is retained is the length of the codewords. This information can be found from the original merge graph by counting the number of merges each state undergoes. A tabular method for doing the merges and determining the codeword lengths is described by Schwartz and Kallick (1964). For the example just worked, their merge algorithm produces the following tables:

Pass 1

Rank-Probability Table

R	P
A	0.3
E	0.25
B	0.15
D	0.15
C	0.1
F	0.04
G	0.01

Combined Probability

R	P
1*	0.05

Node Table

M	R_1	R_2
1*	G	F

Rank-Probability Table | Combined Probability | Node Table

Pass 2

R	P
A	0.3
E	0.25
B	0.15
D	0.15
C	0.1
1*	0.05

R	P
2*	0.15

M	R_1	R_2
1*	G	F
2*	1*	C

Pass 3

R	P
A	0.3
E	0.25
2*	0.15
B	0.15
D	0.15

R	P
3*	0.3
4*	0.4

M	R_1	R_2
1*	G	F
2*	1*	C
3*	D	B
4*	2*	E

Pass 4

R	P
4*	0.4
3*	0.3
A	0.3

R	P
5*	0.6

M	R_1	R_2
1*	G	F
2*	1*	C
3*	B	D
4*	2*	E
5*	A	3*

Pass 5

R	P
5*	0.6
4*	0.4

R	P
6*	1

M	R_1	R_2
1*	G	F
2*	1*	C
3*	B	D
4*	2*	E
5*	A	3*
6*	4*	5*

The initial rank-probability table is transformed into the node table. The process can be speeded up by merging more than one pair of states in each pass.

Successive pairs in the rank-probability table may be merged provided that the greater probability of a pair is less than the sum of the initial combination in the pass. This occurs in pass 3.

The node table is searched with the routine shown in figure 1. This routine determines the lengths of the codewords for each initial state by finding the states at which there is a change in the length of the codewords. This routine examines R_1 before R_2, so in order for it to work correctly the node table must be filled in by writing the upper state of a combined pair in R_2 and the lower state in R_1. This makes probability $(R_1) \leq$ probability (R_2) at any M.

The output of the routine is a list of codeword lengths and letters. For example worked previously, it is:

1,0; 2,E; 3,C; 4,G

The letters are the last letters (going down the initial ranking of the source alphabet) which have a codeword of the length associated with the letter. Thus the above list means that:

A and E have codewords of length 2,
B, D and C have codewords of length 3, and
F and G have codewords of length 4.

The routine searches the segment of the node table from M_1 to M_2 for the first occurrence of an unstarred state which corresponds to the last codeword of length i. The segment is then searched for the first occurrence of a starred state which is taken as M_1 for the i+1'th step with M_2 equal to the previous M_1.

1. The first code assigned consists of i_1 zeros where i_1 is the shortest codeword length.

2. Subsequent codes (if any) of length i_1 are obtained by binary addition of 1 until all codes of length i_1 have been assigned.

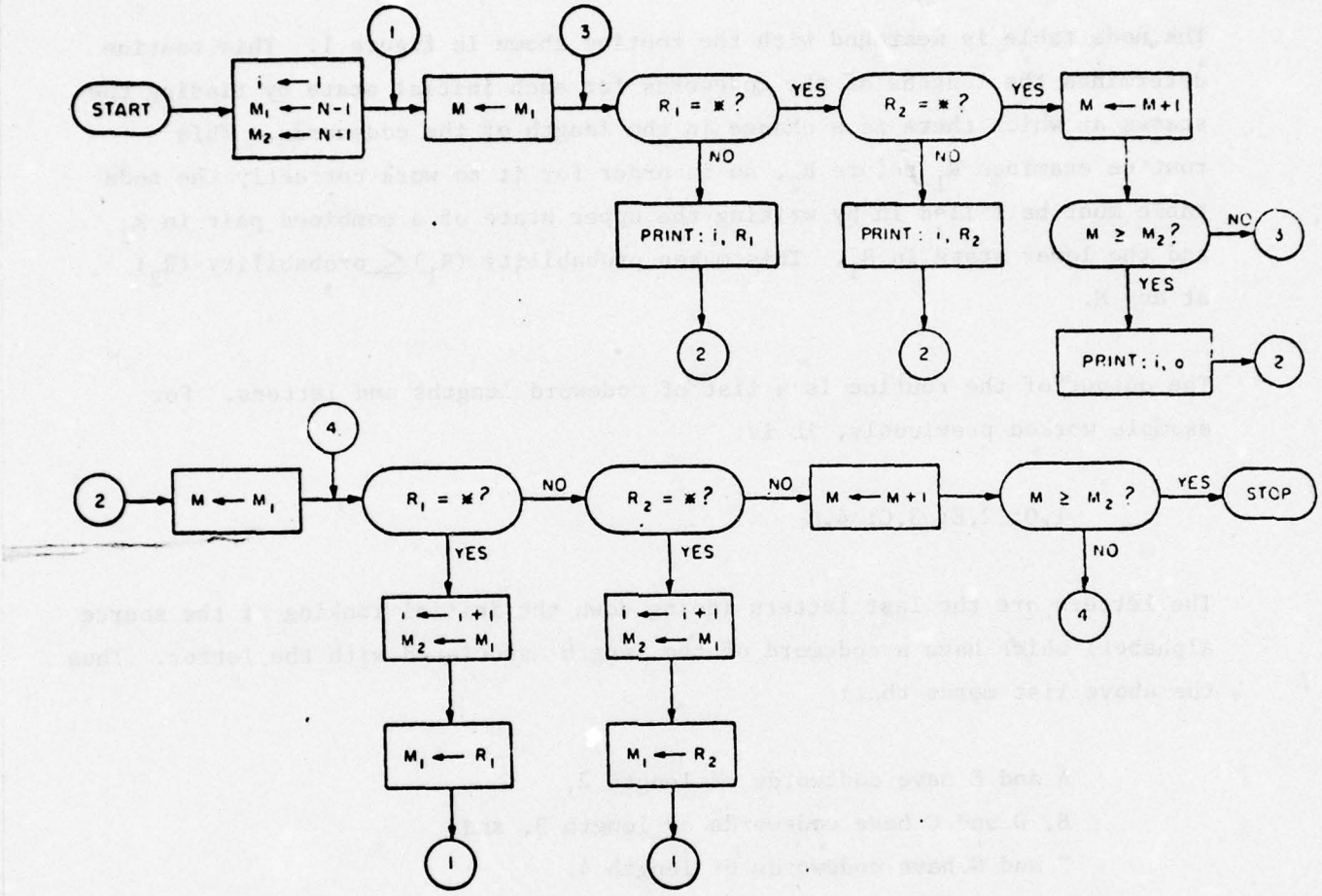


Figure 1. Node Table Search Routine

3. The next code assigned is obtained by binary addition of 1 followed by affixing $i_2 - i_1$ zeros where i_2 is the next codeword length.

4. Step 2 and 3 are repeated for all values of i which are codeword lengths.

These rules generate the code previously obtained using a rearranged binary tree. This algorithmic definition of a Huffman code allows the code to be stored very compactly without using a code table. We need only store the segments of the code alphabet (in the order in which codewords are assigned) together with the length of the codewords to be assigned to each segment.

Decoding is also very simple with this description of the code. The decoding algorithm is described by Cullum (1972):

1. Set i to the shortest codeword length. Set p = -1.

2. Compare the first i digits in the message with the codeword of that length. If the i message digits are smaller than or equal to the codeword, then they represent a codeword of length i and can be decoded by finding the (j-p)th letter in the set of codewords with length i, where j is the binary value of the message digits. Go to step 4.

3. If the message digits are greater than the codeword in step 2, set p = one less than the binary value of the first codeword of length i', where i' is the next codeword length above i. (Usually i' = i + 1). Set i = i' and return to step 2.

4. If the message has not been completely decoded, remove the decoded word from the message and return to step 1.

This algorithm will be illustrated by decoding the message 111000101 using the example code developed earlier.

Step 1: i=2 p = -1 j = 11_2 = 3

Step 2: E has codeword 01_2, so j > E

Step 3: p = 100_2 -1 = 11_2 = 3
i' = 3 $\longrightarrow$ i

Step 2: i = 3 j=111_2 = 7
C has codeword 110, so j > C

Step 3: p = 1110-1 = 1101_2 = 13
i' = 4 $\longrightarrow$ i

Step 2: i = 4 j = 1110 = 14
G = 1111, so codeword is (14-13) or 1st in the group of codewords of length 4. This is F.

Step 4: Discard 1110.

Step 1: i = 2 p = -1 j = 00

Step 2: E has codeword 01, so codeword represents the (0-(-1)) or 1st codeword in the group of codewords of length 2. This is A.

Step 4: Discard 00.

Step 1: i = 2 j = 10_2 = 2

Step 2: E has codeword 01, so j > E

Step 3: p = 11_2 = 3
i' = 3 $\longrightarrow$ i

Step 2: $i = 3 \quad j = 101_2 = 5$

C has codeword 110, so codeword is (5-3) or 2nd codeword in group with codewords of length 3. This is D.

Step 4: The message is decoded as FAD.

Another algorithm for decoding Huffman codes is described in the section Gilbert-Moore Alphabetic Codes. This alternative algorithm has a much more complex set of decoding tables than the method just discussed, but the alternative method does not require that the codewords be assigned in order of increasing length as is necessary for the method just given. The method in the section referred to above is, in fact, applicable to any prefix code.

Modified Huffman Codes

The advantage of modifying Huffman codes so that all codewords of a given length have the same prefix is apparent from the decoding algorithm. Once the length of the codeword is known, decoding is immediate using step 2. Long searches for the correct value of i can be eliminated if we can determine i directly from the first few bits in the message. However, the length indicating prefix cannot be too short unless a significant sacrifice in compression is acceptable. This means that only the longer (less common) codewords will have a length indicating prefix, so the actual savings will be small.

Limiting the maximum length of the codewords is primarily to avoid excessive bit stream manipulation every time a very long codeword is encountered. If we have a source alphabet of N letters, it is theoretically possible to have codeword lengths up to N-1. Since the average length will be more like $\log_2 N$, such long codewords are inconvenient to handle. Limiting the codeword length to 12 bits barely affects compression for a code with a source alphabet of 100 characters or less.

Two other features are often useful in Huffman codes. These are a copy feature and a run length coding feature. The copy capability can be used to reduce the number of symbols to be Huffman encoded. The less frequent symbols are grouped and their probabilities are added so that only one codeword is assigned to the group. Each time a letter in the group has to be encoded, the Huffman codeword for the group is written and it is followed by the character to be encoded. The Huffman codeword for the group is a "copy code" indicating that the character following is not a Huffman codeword.

Run length coding can be achieved in at least two ways. One of the Huffman codewords may be designated as a "repeat code." A string of repeated characters can be encoded as the repeat code followed by the repeated character and a (fixed field) binary count of the number of repeats. Alternatively, each character likely to be repeated can be assigned a repeat code of its own. Thus there can be separate codewords for strings of blanks, zeros, etc. These codewords need only be followed by a (fixed field) binary count of the number of

repeats. The two methods can be combined with, for example, separate codewords for strings of blanks and strings of zeros plus a repeat codeword for strings of any other character.

State Dependent Coding

This method attempts to take advantage of the dependencies between adjacent letters in a nonrandom character string by using several variable length codes to encode the string. The code used to encode a letter depends on the previous letter encoded. In the most elaborate scheme, there is a separate Huffman code associated with each of the N letters in the alphabet, considered as preceding letters. To derive the code for "e," for example, the file is scanned and a count is made of the number of times each character immediately follows an "e." These statistics are then used to derive a Huffman code for the alphabet. Similarly, all other letters in the alphabet have Huffman codes associated with them as preceding characters. Each letter in the N-character alphabet has N different codewords, and the codeword used to encode it is determined by the letter which precedes it. For example, when "t" is encoded, if it occurs as "at" the code associated with "a" is used but if it occurs as "et" the code associated with "e" is used. Similarly, if "t" occurs as "att," the code associated with "a" is used and then the code associated with "t" is used.

The reason for using this technique is that it gives substantial additional compression over simple Huffman coding. The frequency distribution of the character set varies depending on the preceding character. For example, the frequency distribution of characters as first letters of a word (i.e., following a blank) is quite different from the overall frequency distribution. ("e," the most commonly used character in English, is relatively uncommon as the initial letter of a word.) As a second example, consider the letters following a "t." "t" is the second most frequently used letter in English text, but is relatively rare following another "t." "h" and vowels are much more common letters to find following a "t." Using a separate code for each preceding character takes advantage of the dependencies built into the language and improves the compression.

The language of the discussion that follows is simplified by the idea of the state of the encoder. We associate a state with each character and then say that the process of encoding a particular character leaves the encoder in the

state associated with that character. If we assign state 1 to a blank, state 2 to "a," state 3 to "b," state 4 to "c," etc. then encoding "a" leaves the encoder in state 2, encoding "f" leaves the encoder in state 7, etc. To say that the encoder is in state 8 merely tells us that the encoder has just encoded a "g." The usefulness of the concept of the state of the encoder will become apparent shortly.

Since we originally associated a set of Huffman (or other variable length compression) codes with letters occuring as preceding characters, we can instead associate the codes with the states. Then, instead of saying that the code used to encode a character depends on the previous character encoded, we say that the code used depends on the state of the encoder.

The discussion so far has assumed that there is a separate state for each character. This maximizes both the compression and the overhead. It is possible to merge states which have similar codes so that the overhead is reduced without losing very much compression. In general, then, a state may contain one character, several different characters, or even some character strings (we will not investigate this last possibility). The fewer the number of states, the smaller the overhead and the greater the loss in compression. The state merging process is quite complex, as will become apparent.

To decode correctly, the decoder must know the state of the encoder when it did the encoding. This is usually accomplished by using a convention such as starting each record in state 0, a special state to indicate the start of a record.

The power of this method of compression is illustrated by the following test results. Mommens and Raviv (1967) compressed a short section of text which was originally in 8-bit ASCII. Using a single Huffman code, they achieved a compression ratio of 1.88 (4.25 bits/character). Using two codes (i.e. two states), they achieved compression ratios of 2.62 to 2.16 (3.05 to 3.7 bits/character) depending on the size of the decoding tables used.

The disadvantages of the technique are that the size of the decoding tables is increased substantially and generating the codes is a much more complex procedure than generating a single Huffman code. (The example later demonstrates this point.) Encoding and decoding are also considerably slower than in a simple Huffman code. Unfortunately, there is almost no published data by which to evaluate this technique. The test cited above was too small to do more than indicate the desirability of further research. This technique is an alternative to fixed length encoding of character strings and takes the dependencies between adjacent letters into account in a completely different way. The theoretical basis for the method is discussed by Ott (1967) and the implementation is described in Mommens and Raviv (1974). They describe its use with Huffman codes, but any variable length compression code could be used. The description and example which follow are adapted from their report.

We start by assigning a separate state to each letter in the alphabet. Since some states have a low probability of occurrence, we can use a suboptimal code and hardly affect the overall compaction. In addition, two or more states may have very similar conditional probability vectors, i.e., very similar coding tables associated with them. Therefore, the optimal code for one of these states may constitute a good suboptimal code for the other, and using one coding table for these "combined" states would not result in a significant loss of compaction.

In general, we can reduce the original number of states N to a much smaller number N' using a step by step clustering procedure. The following is a clustering procedure which is clearly not optimal but is known to give good results. At each step we combine two states into one in such a way that we keep the loss in compaction to a minimum. The frequency of occurrence of a character in the new combined state is equal to the sum of the frequencies of the character in the two original states. This clustering procedure is illustrated in the example which follows. At the beginning of the clustering procedure, since we combine either very infrequent states or states whose conditional probability vectors are similar, we hardly lose any compaction; but as the number of states diminishes,

the loss in compaction at each step of clustering gets bigger, while the gain in encoding and decoding table sizes stays constant.

It is possible to reduce this effect by ordering the conditional frequency vectors in descending order before adding them. When we combine states whose conditional frequencies are ordered, the order is maintained. The "most frequent character" in state S_1 and state S_2 will be the "most frequent character" in the combined state S_{12}, but their true identity will be lost unless we keep track of the ordering procedure. Therefore, we have to keep an extra mapping table containing a mapping vector for each state that we order. Using this procedure, we reduce both the reduction in the size of the encoding and decoding tables and the loss in compaction, but the net result is very favorable, i.e., the reduced loss in compaction outweighs the increase in coding table storage required to keep the mapping table (see figures 2 and 3). This two-step clustering procedure can be summarized as follows:

1. Combine states step by step up to a certain number M.

2. Order the conditional frequencies for each of these M states and keep track of the sorting, i.e., keep M permutations.

3. Resume the clustering procedure, now on the ordered states, to a final number W of states (we refer to these final combined states as "coding sets." The choice of the numbers M and W depends mainly on the amount of compaction that we are ready to give up for a specific reduction in space requirements for the encoding/decoding tables.

Note that as long as the main upper curve in figure 3 is steep, i.e., the loss in compaction is small and the gain in coding table size is large, it does not pay to reorder the frequency vectors and incur the mapping table overhead. Clearly, the smaller the number of states left after the first clustering stage (at the time of reordering) the smaller the mapping table which must be kept.

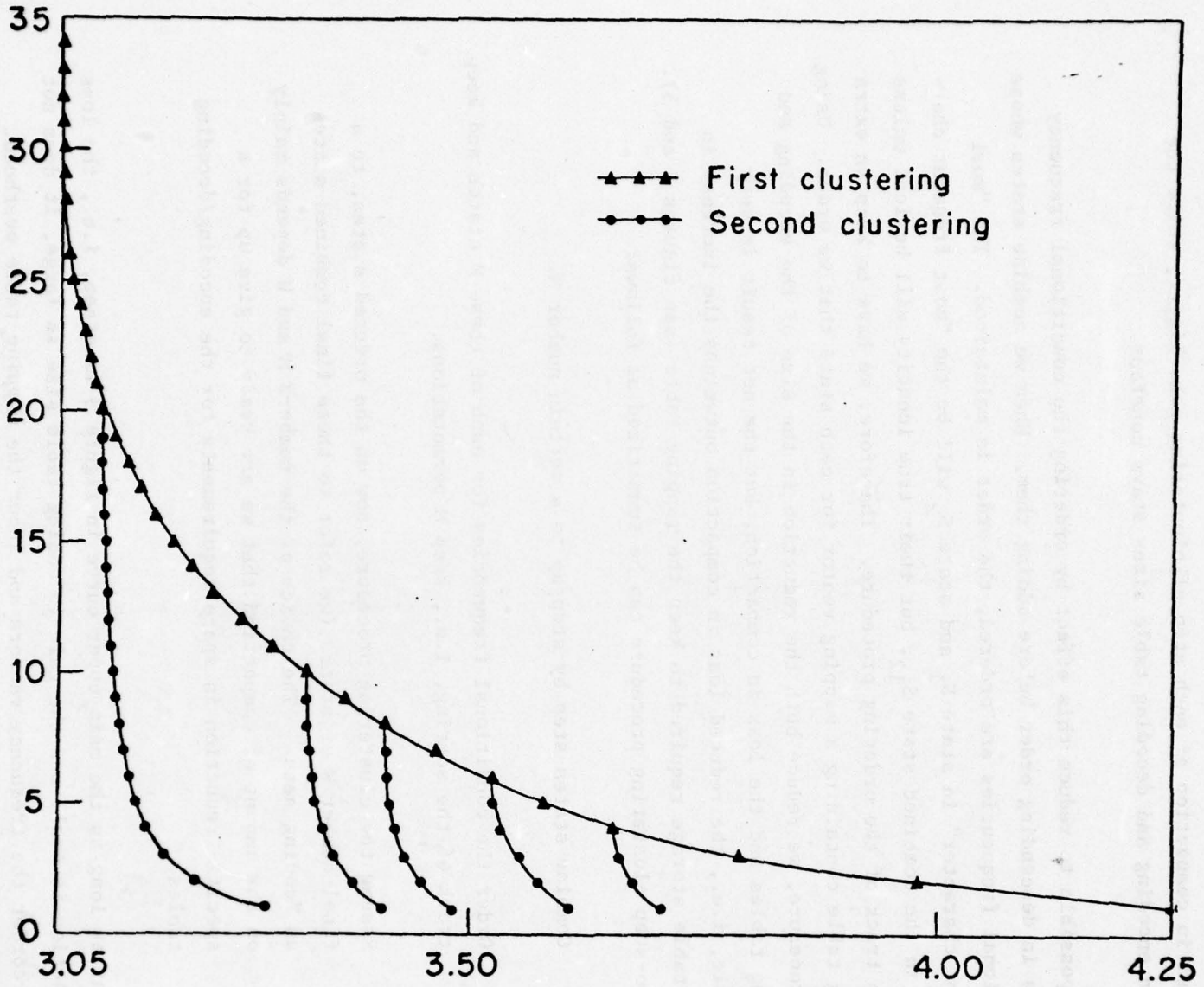


Figure 2. Variation of Compression with Number of States

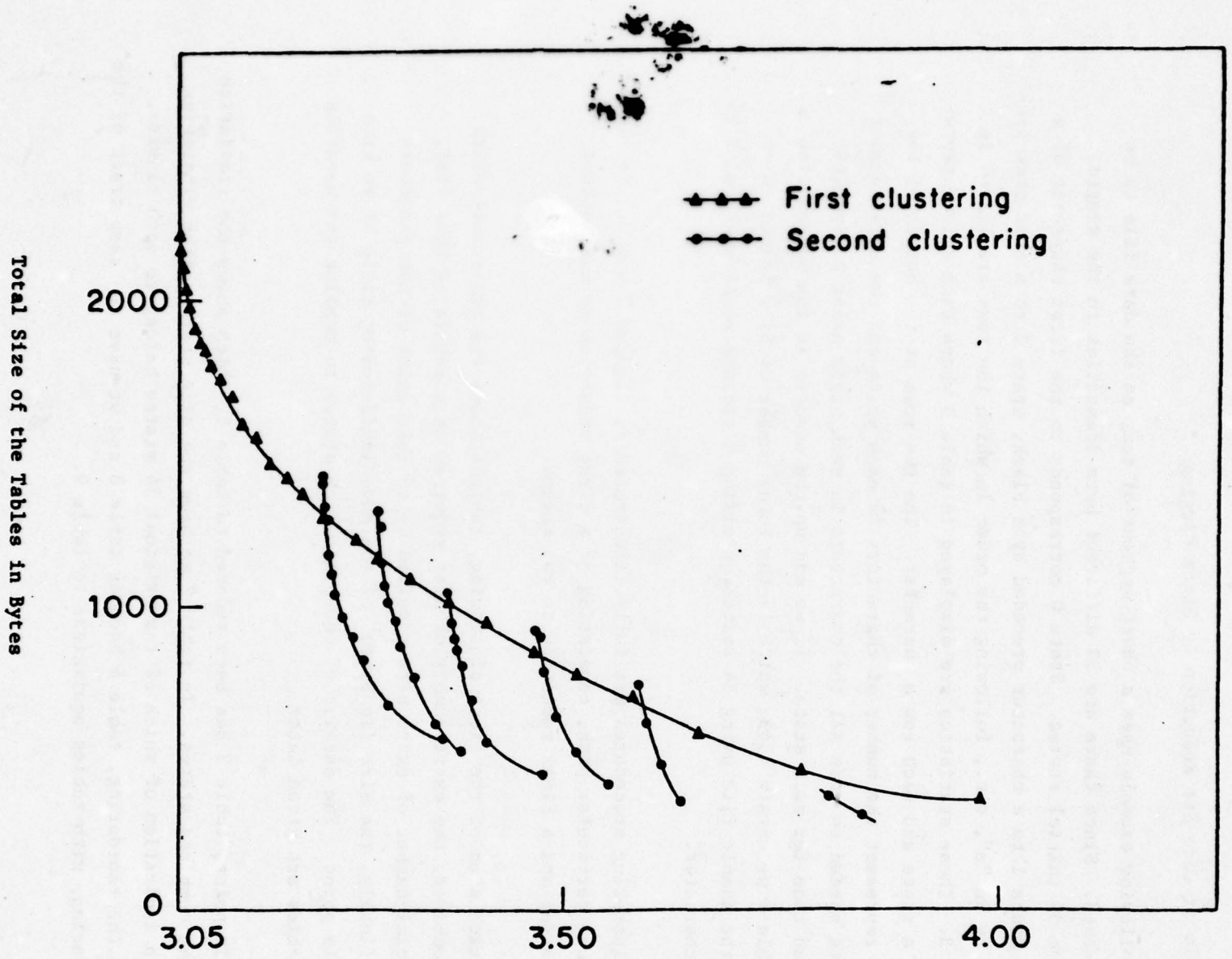


Figure 3. Variation of Code Table Size with Merging

Example of Code Set Reduction By State Merging

The following example uses a short segment of text as the data file to be compressed. Since there are 33 different byte identities in the sample, we have 34 initial states. State 0 corresponds to the first character of a line, state 1 to a character preceded by a blank, state 2 to a character preceded by an "e", etc., following the order in which the rows are listed in table 3. These statistics are displayed in table 3 where each column represents a state and each row a character. The two rows at the bottom of the table represent the number of characters in each state and the total number of bits needed to code all the characters in each state using a separate Huffman code for each state. If we add up the numbers in the bottom line of table 3 we obtain 4343, which is the total number of bits needed to encode the sample file using 34 states, yielding a storage requirement of 3.05 bits/character.

The clustering procedures are fully illustrated in figures 2 and 3. We shall show one particular path, consisting of a first reduction to nine states, reordering and a final reduction to two states.

For each step of the first clustering, table 4 shows the two states which are combined, the extra number of bits required as a result of this step, the total number of extra bits required up to this point of the procedure and, finally, the size (in bytes) of the encoding/decoding table if we stop at this point. The details of choosing which states to combine and updating the tables are given later.

At this point, table 3 has been reduced to table 6, which shows the statistics for the combined states. In table 7 we have the nine states after clustering and an indication of which of the original 34 states belong to each cluster. After the reordering, table 6 becomes table 8 and we have to keep track of the reordering, with tables equivalent to table 9.

Table 3 - Character Counts for Each State

Characters	states 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
blank	3	0	54	17	11	12	4	15	0	23	0	10	12	1	0	11	4	2	1	9	0	12	8	11	0	0	1	3	1	0	0	0	0	1
E	0	8	1	15	0	24	0	9	4	8	5	28	17	6	6	0	5	6	5	5	3	0	1	0	8	0	0	0	0	0	0	0	0	0
T	4	29	1	0	13	5	3	11	16	11	10	0	0	1	0	0	0	0	0	2	0	0	3	1	0	0	0	2	0	0	0	1	0	
A	0	25	2	11	0	14	0	0	0	2	4	13	5	2	4	0	2	5	1	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
R	1	5	27	3	18	2	10	0	0	0	0	0	0	11	0	4	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0
O	1	20	1	3	0	7	1	5	4	0	22	3	0	4	0	2	2	2	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
N	2	7	19	0	16	0	13	1	22	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
I	1	15	0	12	0	4	1	2	0	3	2	3	8	0	9	2	3	1	8	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S	0	12	19	5	3	2	3	5	8	2	1	0	2	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0
C	2	24	4	0	13	0	3	6	5	0	3	0	0	0	1	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
H	1	5	0	41	0	0	0	0	0	2	11	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
D	1	6	6	0	0	5	19	6	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
P	1	9	4	1	1	0	4	0	0	2	0	0	0	1	0	0	0	3	0	0	0	0	1	0	0	1	0	1	0	0	0	0	0	0
B	2	14	1	0	2	0	2	0	1	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
F	1	4	2	0	0	0	10	0	1	0	0	0	0	0	0	3	0	0	0	0	3	0	0	0	0	1	0	0	0	0	0	0	0	0
M	0	5	1	0	6	0	0	0	4	0	0	0	0	3	1	0	3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
W	0	5	1	0	2	0	3	0	4	1	0	0	0	0	0	3	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0
G	1	19	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
U	1	2	1	0	2	0	0	14	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
.	0	2	0	0	0	0	2	3	0	0	3	4	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0
V	0	0	2	0	0	2	0	1	2	4	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	3	0	0	0	0	0
(	0	1	2	4	1	2	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	2	0	0	2	0	1	0	3	0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0
X	0	2	3	0	3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
)	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
v	0	0	3	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
/	0	0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0
;	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
K	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
No. of characters in each state	22	226	161	112	91	81	81	79	76	63	62	62	46	29	26	25	21	23	22	20	20	13	15	12	9	5	3	6	5	4	1	1	1	1
No. of bits needed to code all characters in each state	81	524	530	313	291	250	279	262	241	189	173	139	107	75	60	57	60	64	50	43	60	13	30	12	9	12	5	9	7	4	1	1	1	1

Table 4 - First Clustering Procedure

states combined		extra bits required	total no. of extra bits up to this point	E/D tables if clustering stops
15	33	0	0	2213
23	32	0	0	2203
29	30	0	0	2159
11	24	1	1	2147
0	31	2	3	2111
22	23	2	5	2101
22	27	2	7	2071
21	26	3	10	2033
9	19	6	16	1967
21	22	8	24	1903
0	25	9	33	1857
28	29	9	42	1815
16	18	10	52	1781
11	12	14	66	1715
9	21	17	83	1637
14	16	20	103	1585
13	17	21	124	1541
0	6	24	148	1453
15	20	27	175	1411
7	28	28	203	1353
5	14	33	236	1279
2	15	44	280	1213
5	13	46	326	1145
4	8	53	379	1081
1	10	60	439	1005

Table 5 - Second Clustering Procedure

New states combined		extra bits required	total extra bits to this point	size in bytes of E/D tables if clustering stops
0	4	1	440	1043
0	5	5	445	941
0	6	5	450	895
8	8	6	456	851
3	7	8	464	804
0	2	18	482	687
0	1	26	508	546

Table 6 - First Clustering Results

characters	New States 0	1	2	3	4	5	6	7	8
B	7	0	66	17	11	20	16	67	22
E	0	13	4	15	4	52	9	14	53
T	7	39	1	0	29	6	11	20	0
A	0	29	3	11	0	28	0	2	19
R	11	5	33	3	18	13	0	0	0
O	2	42	3	3	4	20	5	0	3
N	15	7	20	0	38	0	1	0	0
I	2	17	2	12	0	25	2	5	11
S	3	13	23	5	11	2	5	2	2
C	6	27	5	0	18	1	6	0	0
H	1	16	0	41	0	2	0	2	0
D	21	7	6	0	1	6	6	0	0
P	6	9	4	1	1	4	0	4	0
B	5	14	1	0	3	3	0	0	0
F	12	4	8	0	1	0	0	0	0
L	0	5	2	0	10	7	0	0	0
M	3	5	8	0	6	0	0	1	0
W	3	19	0	0	0	0	0	0	0
G	1	2	1	0	2	0	14	1	0
U	2	5	0	0	0	1	7	0	5
.	0	0	2	0	2	2	4	6	0
Y	0	1	2	4	1	7	0	0	0
,	0	0	2	0	0	3	2	4	1
V	0	2	3	0	4	0	0	0	0
(	0	5	0	0	0	0	0	0	0
8	0	2	0	0	0	0	1	0	0
X	0	0	3	0	3	0	0	0	0
)	0	0	1	0	0	0	0	3	1
0	0	0	4	0	0	0	0	0	0
<	1	0	0	0	0	0	0	0	0
/	0	0	0	0	0	0	0	1	0
:	0	0	0	0	0	0	0	1	0
K	1	0	0	0	0	0	0	0	0

Table 7 - Original State to Intermediate State Mapping

original states	New States 0	1	2	3	4	5	6	7	8
	0	1	2	3	4	5	7	9	11
	6	10	15		8	13	28	21	12
	25		20			14	29	22	24
	31		33			17	30	19	
						16		26	
						18		27	
								23	
								32	

Table 8 - Reordering Results

characters	New States 0	1	2	3	4	5	6	7	8
	21	42	66	41	38	52	16	67	53
	15	39	33	17	29	28	14	20	22
	12	29	23	15	18	25	11	14	19
	11	27	20	12	18	20	9	6	11
	7	19	8	11	11	20	7	5	5
	7	17	8	5	11	13	6	4	3
	6	16	6	4	10	7	6	4	2
	6	14	5	3	6	7	5	3	1
	5	13	4	3	4	6	5	2	1
	3	13	4	1	4	6	4	2	0
	3	9	4	0	4	4	2	2	0
	3	7	3	0	3	3	2	1	0
	2	7	3	0	3	3	1	1	0
	2	5	3	0	2	2	1	1	0
	2	5	3	0	2	2	0	1	0
	1	5	2	0	1	2	0	0	0
	1	5	2	0	1	1	0	0	0
	1	5	2	0	1	1	0	0	0
	1	4	2	0	1	0	0	0	0
	0	2	2	0	0	0	0	0	0
	0	2	1	0	0	0	0	0	0
	0	2	1	0	0	0	0	0	0
	0	1	1	0	0	0	0	0	0
	0	0	1	0	0	0	0	0	0
	C	0	.....						0
	0	0	.....						0

We are now ready to start the second clustering. The results of this procedure are illustrated in table 5 (beneath table 4). Notice that the size of the encoding/decoding table at the top of table 5 (after the first step of the second clustering) is slightly higher than the value at the bottom of table 4. This is due to the fact that now we have to add the size of the "unscrambling tables" table 9. The difference is small, however, since we keep this information only for the reordered states which are combined with one another (after the first step, we have to remember what happened to former state 4 only). States are not reordered until they are merged.

We can see in more detail how the clustering procedure works by looking at the second stage (where the matrices are smaller). For each possible pair of states, we compute how many extra bits are needed to code the sample if these two states are combined. (We can use any other equivalent "distance" for efficiency in computation.) This gives us the triangular matrix table 10. The minimum value in this matrix tells us which states we shall actually combine, in this case 0 and 4. Next, we update the matrix. Updating column 0 requires that we derive a new Huffman code for the combined state and recalculate how many extra bits are needed if we combine the new state 0 with one of the other states in the set $\{1,2,3,5,6,7,8,\}$. We update column 0 (corresponding to the new combined state) and ignore line and column 4 (corresponding to the "absorbed" state). Here we have a symbolic -1 to indicate ignore. Thus we obtain table 11. Then we select the minimum of the new matrix, 5, which corresponds to the couple 0-5 etc.

In this example we continue the second clustering procedure until two clusters are left. Now we produce the final codes referred to as coding set I and coding set II and displayed on table 13.

The last thing we shall show is how to encode, and decode. The data string is "␣IN␣GENERAL".

␣ is in the initial state 0 (start of record), state 0 after the first

Table 9 - Mapping Table (Unscrambling Table)

State	1	2	3	. . .	Reordered Character	. . .	32	33
0	12	7	15	5	1	3	10	13	14	9	17	18	6	8	20	11	19	30	33	2	4	16	21	22	23	24	'25	26	27	28	29	31	32
1	6	3	4	10	18	8	11	14	2	9	13	7	12	5	16	17	20	25	15	19	24	26	22	1	21	23	27	28	29	30	31	32	33
2	1	5	9	7	15	17	12	10	2	13	29	4	6	24	27	8	16	21	22	23	3	14	19	28	11	18	20	25	26	30	31	32	33
3	11	1	2	8	4	9	22	5	6	13	3	7	10	12	14	15	16	17	18	19	20	21	23	24	25	26	27	28	29	30	31	32	33
4	7	3	5	10	1	9	16	17	2	6	24	14	27	19	21	12	13	15	22	4	8	11	18	20	23	25	26	28	29	30	31	32	33
5	2	4	8	1	6	5	16	22	3	12	13	14	23	9	11	21	10	20	7	15	17	18	19	24	25	26	27	28	29	30	31	32	33
6	1	19	3	2	20	10	12	6	9	21	8	23	7	26	4	5	11	13	14	15	16	17	18	22	24	25	27	28	29	30	31	32	33
7	1	3	2	21	8	13	23	28	4	9	11	17	19	31	32	5	6	7	10	12	14	15	16	18	20	22	24	25	26	27	29	30	33
8	2	1	4	8	20	6	9	23	28	3	5	7	10	11	12	13	14	15	16	17	18	19	21	22	24	25	26	27	29	30	31	32	33

Entry gives offset of corresponding original character.

Example The fourth character in state 1 is the tenth character in the original list, which is "c".

Table 10 - Distance Between Tables

State								
1	3							
2	6	20						
3	17	43	21					
4	1	17	8	16				
5	5	29	10	9	4			
6	6	12	12	9	4	7		
7	38	56	26	7	37	40	33	
8	43	67	33	6	36	39	34	10
	0	1	2	3	4	5	6	7

State

Table 11 - Updated Distance After 0-4 Merge

1	19							
2	11	20						
3	17	43	21					
4	-1	-1	-1	-1				
5	5	29	10	9	-1			
6	6	12	12	9	-1	7		
7	38	56	26	7	-1	40	33	
8	43	67	33	6	-1	39	34	10
	0	1	2	3	4	5	6	7

State

Table 12 - Updated Distance After 0-5 Merge

State								
1	33							
2	14	20						
3	16	43	21					
4	-1	-1	-1	-1				
5	-1	-1	-1	-1	-1			
6	5	12	12	9	-1	-1		
7	38	56	26	7	-1	-1	33	
8	42	67	33	6	-1	-1	34	10
	0	1	2	3	4	5	6	7

State

Tabel 13 - Final Codes

Coding Set I

(1)	(2)	(3)	(4)
1	235	2	11
2	158	3	101
3	118	3	100
4	105	3	011
5	72	4	0101
6	62	4	0100
7	51	5	00111
8	43	5	00110
9	37	5	00101
10	34	5	00100
11	26	6	000111
12	21	6	000110
13	19	6	000101
14	15	6	000100
15	14	6	000011
16	11	7	0000101
17	10	7	0000100
18	10	7	0000011
19	8	7	0000010
20	4	8	00000011
21	3	8	00000010
22	3	8	00000001
23	2	9	000000001
24	1	9	000000000

Intermediate states 0,4,5,6,2,1 belong to coding set 1.

(1) Character
(2) Relative frequency
(3) Length of code word
(4) Code word.

Coding Set II

(1)	(2)	(3)	(4)
1	161	1	1
2	59	3	011
3	48	3	010
4	29	4	0011
5	21	4	0010
6	12	5	00011
7	10	5	00010
8	7	6	000011
9	6	6	000010
10	3	7	0000011
11	2	7	0000010
12	1	8	00000011
13	1	8	00000010
14	1	8	00000001
15	1	8	00000000

Intermediate states 3,8,7 belong to Coding Set II.

../

clustering (table 7) and after the second clustering in coding set I (table 13). Looking in table 9, we see that a ∅ (character 1 in table 3) occupies the fifth position in state 0. Therefore, we shall use the fifth code word in coding set I, i.e., 0101. The next character, an 'I' being preceded by a blank, is in initial state 1, intermediate state 1, coding set I. After reordering, I (which is the 8th character in table 3) occupies the 6th position in state 1. We code it as the 6th code word in coding set I. (We will say that we encode it as a 6 in coding set 1.) Next:

N initial state 8, intermediate state 4, coding set I is a 1
∅ initial state 7, intermediate state 6, coding set I is a 1
G initial state 1, intermediate state 1, coding set I is a 20
E initial state 19, intermediate state 7, coding set II is a 3
N initial state 2, intermediate state 2, coding set I is a 4
etc. which yields

0101010011110000001101001101110110100111011

To decode, we obviously start with a character at the beginning of a line: we know that the initial state is 0 - intermediate 0 - coding set I. Therefore, we decode using the table corresponding to the first group. There we find that 0101 is the codeword for 5. Referring now to table 9 we find that the fifth number in row 0 is 1. Finally, using table 3, the character is a blank. We know that the second character will be in initial state 1, intermediate state 1, coding set I, so we know which table to use to decode. We find a 6, which, referring to the row 1, of table 9, leads us to decode an I; etc.

GILBERT-MOORE ALPHABETIC CODES

These codes are variable length binary codes which give nearly as much compression as Huffman codes, but which have an alphabetic property. Binary ordering of encoded words (left justified) is equivalent to alphabetic ordering of the original words. This property makes these codes suitable for compressing alphabetic lists which are subject to searching, such as indexes of names. The following description of the general code generating algorithm is taken from the paper by Gilbert and Moore (1959). It is illustrated by an example which follows the description of the algorithm. A "prefix set" is a set of letters which have codewords beginning with the same prefix.

In general, the method builds up the best alphabetical encoding for the entire alphabet by first making best alphabetical encodings for certain subalphabets. In particular, the subalphabets considered are only those which might form a prefix set in some alphabetical binary encoding of the whole alphabet. Since only those sets of letters consisting exactly of all those letters which lie between some pair of letters can serve as a prefix set, we call such a set an "allowable" subalphabet.

We denote the allowable subalphabet consisting of all of those letters which follow L_i in the alphabet (including L_i itself) and which precede L_j (again including L_j itself) by (L_i,L_j). When referring to the ordinary English alphabet, the symbol # is used for the space symbol. Thus, (#, B) is the subalphabet containing the three symbols space, A and B. (A,A) denotes the subalphabet containing only the letter A.

If it were desired to find an optimum encoding satisfying certain kinds of restrictions other than the alphabetical one, different allowable subalphabets could be used, with the rest of the algorithm remaining analogous. This method of building up an encoding by combining encodings for subalphabets is analogous to the method used by Huffman except that he was able to organize his algorithms such that no subalphabets were used

except those which actually occurred as prefix sets in his final encoding. However, we consider all allowable subalphabets, including some which are not actually used as part of the final encoding.

The term "cost of an encoding" is used to refer to the average number of binary digits per letter of a transmitted message, that is, $\sum_i p_i N_i$, where N_i is the length of the codeword for the i-th source symbol. Since we are constructing an encoding for each allowable subalphabet, we also use the corresponding sum for each subalphabet. But, since the probabilities p_i do not add up to 1 for proper subalphabets, the sum $\sum_i p_i N_i$ does not correspond exactly to the cost of transmitting messages, and so the corresponding sum is called a partial cost.

The algorithm takes place in n stages, where n is the number of letters in the alphabet. At the k-th stage, the best alphabetical binary encoding for each k-letter allowable subalphabet is constructed and its partial cost is computed. For k=1, each subalphabet of the form (L_i, L_i) is encoded by the trivial encoding wihich encodes L_i with the null sequence; it has cost 0 since the number of digits in the null sequence is zero. For k=2, each subalphabet of the form (L_i, L_{i+1}) is encoded by letting the code for L_i be 0 and the code for L_{i+1} be 1. The partial cost of this encoding is $P_i + P_{i+1}$. In general, the k-th stage of algorithm, in which it is desired to find the best alphabetical binary encoding for each subalphabet of the form (L_i, L_{i+k-1}) and its partial cost, proceeds by making use of the codes and the partial costs computed in the previous stages.

For each j between i+1 and i+k-1, we define a binary alphabetical encoding as follows: Let $C_i, C_{i+1}, \ldots, C_{j-1}$ be the codes for $L_i, L_{i+1}, \ldots, L_{j-1}$ given by the (previously constructed) best alphabetical encoding for (L_i, L_{j-1}), and let $C'_j, C'_{j+1}, \ldots, C'_{j+k-1}$ be the codes for $L_j, L_{j+1}, \ldots, L_{j+k-1}$ given by the (previously constructed) best alphabetical encoding for (L_j, L_{i+k-1}). Then the new encoding for $L_i, L_{i+1}, \ldots, L_{j-1}, L_j, L_{j+1}, \ldots, L_{i+k-1}$ will be $0C_i, 0C_{i+1}, \ldots,$

$0C_{j-1}$, $1C'_j$, $1C'_{j+1}$, ..., $1C'_{i+k-1}$. Such an encoding is defined for each j. and the encoding is exhaustive. It follows that the best encoding for this subalphabet is given by one of the k-1 such encodings which can be obtained for the k-1 different values of j. The partial cost of such an encoding made up out of two subencodings is the sum of the partial costs of the two subencodings plus $p_i + p_{i+1} + ... + p_{i+k-1}$. To perform the algorithm, it is not necessary to construct all of p_{i+k-1}. To perform the algorithm, it is not necessary to construct all of these encodings, but only to compute enough to decide which one of the k-1 different encodings has the lowest partial cost. This is done by taking the sums of each of the k-1 pairs of partial costs of sub-encodings and constructing the best encoding only.

After the n-th stage of this algorithm has been completed for an n-character alphabet, the final encoding obtained is the best alphabetical encoding for the entire original alphabet, and the final partial cost obtained is the cost of this best alphabetical encoding.

EXAMPLE

We wish to encode the following 5-letter alphabet, with the probability of each letter in parentheses following the letter.

A (0.3), B (0.2), C (0.1), D (0.3), E (0.1)

k = 2 For (L_i, L_{i+1}) the partial cost is $p_i + p_{i+1}$

(A,B) 0.5, (B,C) 0.3, (C,D) 0.4, (D,E) 0.4

k = 3 (A,C) can be encoded (A,A) (B,C) or (A,B) (C,C). From here on we denote these splits by A.BC and AB.C. The sums of the partial costs are:

A.BC	Cost of A subalphabet	= 0
	Cost of BC subalphabet	= 0.3
	Incremental partial cost	= 0.3

AB.C Cost of AB subalphabet = 0.5
Cost of B subalphabet = 0
Incremental partial cost = 0.5

Choose A.BC
Partial cost = Incremental partial cost +
$p_A + p_B + p_C$
$= 0.3 + 0.3 + 0.2 + 0.1$
$= 0.9$

The encoding is: A 0
B 10
C 11

(B,D) can be encoded as:
B.CD (incremental partial cost = 0.4)
BC.D (incremental partial cost = 0.3)
Choose BC.D
Partial cost $= 0.3 + p_B + p_C + p_D$
$= 0.9$

The encoding is: B 00
C 01
D 1

(C,E) can be encoded as:
C.DE (incremental partial cost = 0.4)
CD.E (incremental partial cost = 0.4)

We can use either of these, and choose CD.E
Partial cost = 0.9

AB.C Cost of AB subalphabet = 0.5
Cost of B subalphabet = 0
Incremental partial cost = 0.5

Choose A.BC

Partial cost = Incremental partial cost + $P_A + P_B + P_C$

$= 0.3 + 0.3 + 0.2 + 0.1$

$= 0.9$

The encoding is:

A	0
B	10
C	11

(B,D) can be encoded as:

B.CD (incremental partial cost = 0.4)

BC.D (incremental partial cost = 0.3)

Choose BC.D

Partial cost $= 0.3 + P_B + P_C + P_D$

$= 0.9$

The encoding is:

B	00
C	01
D	1

(C,E) can be encoded as:

C.DE (incremental partial cost = 0.4)

CD.E (incremental partial cost = 0.4)

We can use either of these, and choose CD.E

Partial cost = 0.9

The codes are: C 00
D 01
E 1

K = 4 (A,D) can be encoded as:

A.BCD Cost of A subalphabet = 0
Cost of BCD subalphabet = 0.9
Incremental partial cost = 0.9

AB.CD Cost of AB subalphabet = 0.5
Cost of CD subalphabet = 0.4
Incremental partial cost = 0.9

ABC.D Cost of ABC subalphabet = 0.9
Cost of D subalphabet = 0
Incremental partial cost = 0.9

We can choose any of these, and choose A.BCD

Partial cost = $0.9 + p_A + p_B + p_C + p_D$

= 1.8

The encoding is: A 0
B 100
C 101
D 11

(B,E) can be encoded as:

(incremental partial costs follow the splits)

B.CDE 0.9

BC.DE 0.7

BCD.E 0.9

Choose BC.DE

Partial cost = 1.4

The encoding is:

B	00
C	01
D	10
E	11

k = 5 (A,E) can be encoded as:

(incremental partial costs follow the splits)

A.BCDE	1.4
AB.CDE	1.4
ABC.DE	1.3
ABCD.E	1.8

Choose ABC.DE

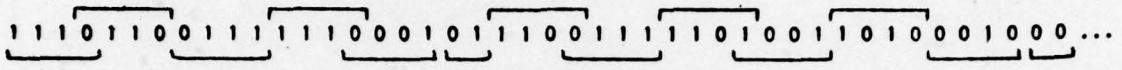
Total cost of the code = 2.3 bits/character

The codes are:

A	00
B	010
C	011
D	10
E	11

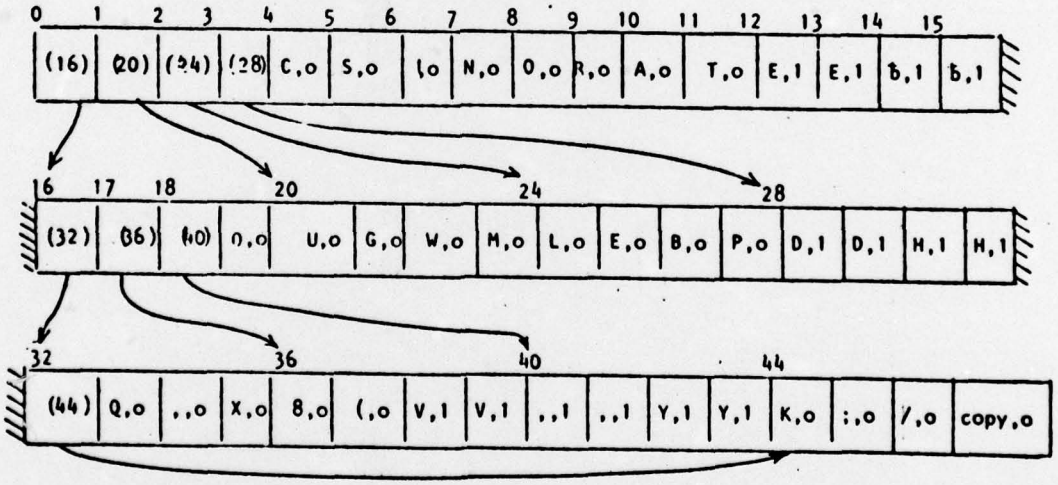
These codes are implemented by a table look-up procedure. For encoding, the ASCII or BCD symbol can be used to generate a table address, since the letters of the alphabet are consecutive numbers in both those codes. The table entry must contain the length of the codeword (fixed field) and the codeword itself. For decoding, several bits (say four bits) of the incoming stream can be used to jump to one of the 16 tables which will allow the codeword to be decoded without a long table search. The table entry must either contain a letter and the number of bits to

retain or a pointer to another table. In the latter case, one examines the next few bits to determine the relative address to consult in the second table. This process continues until a letter is found. This is the "window" decoding procedure (Mommens and Raviv, 1974). The following example from Mommens and Raviv illustrates the method. The initial window length is four bits, and subsequent window lengths are two bits.



first window 1110 = 14: we can decode a blank ␣ and discard 3=(4-1) bits
next " 0110 = 6 " " " an 1 " " 4 bits
" " 0111 = 7 " " " " N " " 4 bits
" " 1110 = 15 " " " a bland ␣ " " 3 bits
" " 0001 = 1 points to a subtable starting at location 20; discard 4 bits.
" " 01 = 1 we can decode a 'G' at 20+1=21 location and discard 2 bits (2-0).

Decoding Table Layout



This scheme for decoding can be used for any variable length codes, including Huffman codes.

Table 14 lists a Gilbert-Moore alphabetical code derived from letter probabilities in English text.

Table 14 - Gilbert-Moore Alphabetical Code Based on Letter Probabilities for English Text

Letter	Probability	Alphabetical Code
Space	.1859	00
A	.0642	0100
B	.0127	010100
C	.0218	010101
D	.0317	01011
E	.1031	0110
F	.0208	011100
G	.0152	011101
H	.0467	01111
I	.0575	1000
J	.0008	1001000
K	.0049	1001001
L	.0321	100101
M	.0198	10011
N	.0574	1010
O	.0632	1011
P	.0152	110000
Q	.0008	110001
R	.0484	11001
S	.0514	1101
T	.0796	1110
U	.0228	111100
V	.0083	111101
W	.0175	111110
X	.0013	1111110
Y	.0164	11111110
Z	.0005	11111111

References

Abramson, N.

INFORMATION THEORY AND CODING; McGraw-Hill, New York, 1963.

Cocke, J., Mommens, J.H., Raviv, J.

METHODS OF AND APPARATUS FOR DECODING VARIABLE-LENGTH CODES HAVING LENGTH-INDICATING PREFIXES; United States Patent, 3,701,111, *Oct.*, 24, 1972.

Cocke, J., Mommens, J.H., Raviv, J.

PROCESSING OF COMPACTED DATA; United States Patent, 3,717,851, Feb. 20, 1973.

Cullum, R.D.

A METHOD FOR THE REMOVAL OF REDUNDANCY IN PRINTED TEXT, NTIS: AD 751 407, Sept, 1972.

Gilbert, E.N. and Moore, E.F.

VARIABLE-LENGTH BINARY ENCODINGS, Bell System Technical Journal, v. 38, pp. 933-967, July 1959.

Huffman, D.A.

A METHOD FOR THE CONSTRUCTION OF MINIMUM REDUNDANCY CODES, Proceedings of the IRE, v. 40, pp. 1098-1101, 1952.

Karp, R.M.

MINIMUM-REDUNDANCY CODING FOR THE DISCRETE NOISELESS CHANNEL, IEEE Trans. on Information Theory, v. IT-7, pp. 27-38, 1961.

Loh, L.S., Mommens, J.H., Raviv, J.

CODE PROCESSOR FOR VARIABLE-LENGTH DEPENDENT CODES; United States Patent, 3,701,108, Oct. 24, 1972.

Loh, L.S., Mommens, J.H., Raviv, J.
METHOD OF ACHIEVING DATA COMPACTION UTILIZING VARIABLE-LENGTH DEPENDENT CODING TECHNIQUES; United States Patent, 3,694,813, Sept. 26, 1972.

Mommens, J.H., Raviv, J.
CODING FOR DATA COMPACTION IBM Research Report No. RC5150, Yorktown Hts., New York, Nov. 26, 1974.

Ott, G.
COMPACT ENCODING OF STATIONARY MARKOV SOURCES; IEEE Trans. on Information Theory, v. IT-13, pp 82-86, 1967.

Raviv, J.
DATA COMPACTION USING MODIFIED VARIABLE-LENGTH CODING; United States Patent, 3,675,211, July 4, 1972.

Raviv, J., Wesley, M.A., Somers
DATA COMPACTION USING VARIABLE-LENGTH CODING; United States Patent, 3,675,212, July 4, 1972

Schwartz, E.S.
AN OPTIMUM ENCODING WITH MINIMUM LONGEST CODE AND TOTAL NUMBER OF DIGITS; Information and Control, v. 7, pp. 37-44, 1964.

Schwartz, E.S. and Kallick, B.
GENERATING A CANONICAL PREFIX ENCODING; Comm. of the ACM, v. 7, pp. 166-169, 1964

Schwartz, E.S. and Kleiboemer, A.J.
A LANGUAGE ELEMENT FOR COMPRESSION CODING; Information and Control, v. 10, pp. 315-333, 1967

Varn, B.
OPTIMUM VARIABLE LENGTH CODES; Information and Control, v. 19, pp. 289-301, 1971

BIBLIOGRAPHY

Abramson, N.

INFORMATION THEORY AND CODING

McGraw-Hill, New York, 1963

Alsberg, P.A.

SPACE AND TIME SAVINGS THROUGH LARGE DATA BASE COMPRESSION AND DYNAMIC RESTRUCTURING

Proc. of the IEEE, August 1975

Belford, G.G., Bunch, S.R., Day, J.D. et al

(Center for Adv. Computation, Univ. of Ill.)

A STATE-OF-THE-ART REPORT ON NETWORK DATA MANAGEMENT AND RELATED TECHNOLOGY

Joint Technical Support Activity Report No. UIUC-CAC-DN-75-150, Contract No. DCA 100-75-C-0021, April 1, 1975, Available from NTIS

Bemer, R.W.

DO IT BY THE NUMBERS - DIGITAL SHORTHAND

Comm. of the ACM, v. 3, pp. 530-536, 1960

Bobrow, D.G.

A NOTE ON HASH LINKING

Comm. of the ACM, v. 18, pp. 413-415, 1975

Cocke, J., Mommens, J.H., Raviv, J.

METHOD OF AND APPARATUS FOR DECODING VARIABLE-LENGTH CODES HAVING LENGTH-INDICATING PREFIXES

United States Patent, 3,701,111, Oct. 24, 1972

Cocke, J., Momm.ens, J.H., Raviv, J.

PROCESSING OF COMPACTED DATA

United States Patent 3, 717,851, Feb. 20, 1973

Cullum, R.D.

A METHOD FOR THE REMOVAL OF REDUNDANCY IN PRINTED TEXT
NTIS: AD 751 407, September 1972

D.D.C.

REPORT ON DATA COMPRESSION
DDC Report No. CW4687, March 31, 1975

D.D.C.

A REPORT BIBLIOGRAPHY
Search Control No. 028472

deMaine, P.A.D., Kloss, K. and Marron, B.A.

THE SOLID SYSTEM, II. NUMERIC COMPRESSION
THE SOLID SYSTEM, III. ALPHANUMERIC COMPRESSION
NBS Technical Note 413, August 15, 1967

deMaine, P.A.D. and Springer, G.K.

DETAILS OF THE SOLID SYSTEM, July 1968.
Computer Science Dept., University of Pennsylvania, 1968

deMaine, P.A.D. and Springer, G.K.

THE COPAK COMPRESSOR
In: File Organization. Selected papers from File 68 -
An I.A.G. Conference
Swets and Zetlinger, Amsterdam, N.V. pp. 149-159, 1969

Gilbert, E.N. and Moore, E.F.

VARIABLE-LENGTH BINARY ENCODINGS
Bell System Technical Journal, v. 38, pp. 933-967, July 1959

Huffman, D.A.

A METHOD FOR THE CONSTRUCTION OF MINIMUM REDUNDANCY CODES
Proceedings of the I.R.E., v. 40, pp. 1098-1101, 1952

Karp, R.M.

MINIMUM-REDUNDANCY CODING FOR THE DISCRETE NOISELESS CHANNEL
IEEE Trans. on Information Theory, v. IT-7, pp. 27-38, 1961

Kautz, W.H.

FIBONACCI CODES FOR SYNCHRONIZATION CONTROL
IEEE Trans. on Information Theory, v. IT-11, pp. 284-292 (1965)

Kautz, W.H.

FILE COMPRESSION FOR SIMPLE ASSOCIATIVE SEARCH
November 1973, NTIS: AD 771 314

Kautz, W.H. and Singleton, R.C.

NON-RANDOM BINARY SUPERIMPOSED CODES
IEEE Trans. on Information Theory, v. IT-10, pp. 363-377 (1964)

Kerpelman, C. (Ed.)

PROPOSED ANS: IDENTIFICATION OF STATES ... FOR INFORMATION INTERCHANGE
Comm. of the ACM, v. 13, pp. 514-515, 1970

Knight, J.M., Jr.

EVALUATION OF A TEXT COMPRESSION ALGORITHM AGAINST COMPUTER-AIDED INSTRUCTION MATERIAL
NTIS: AD 759 162, July 1972

Kreutzer, P.J.

DATA COMPRESSION IN LARGE BUSINESS ORIENTED FILES
Navy Fleet Material Support Office, Mechanicsburg, PA.,
Oct. 5, 1971. NTIS: AD 734 394

Lipetz, B., Stangl, P., and Taylor, K.F.

PERFORMANCE OF RUECKING'S WORD COMPRESSION METHOD WHEN APPLIED TO MACHINE RETRIEVAL FROM A LIBRARY CATALOG
Jnl. Library Automation, v. 2, pp. 266-271, 1969

Loh, L.S., Mommens, J.H., Raviv, J.
METHOD OF ACHIEVING DATA COMPACTION UTILIZING VARIABLE-LENGTH DEPENDENT CODING TECHNIQUES
United States Patent, 3,694,813, Sept. 26, 1972

Loh, L.S., Mommens, J.H., Raviv, J.
CODE PROCESSOR FOR VARIABLE-LENGTH DEPENDENT CODES
United States Patent, 3,701,108, Oct. 24, 1972

Lohse, E. (Ed.)
DATA CODE FOR CALENDAR DATE FOR MACHINE-TO-MACHINE DATA INTERCHANGE
Comm. of the ACM, v. 11, pp. 273-274, 1968

Marron, B.A. and deMaine, P.A.D.
AUTOMATIC DATA COMPRESSION
Comm. of the ACM. v. 10, pp. 711-715, 1967

Martin, J.
DATA BASE ORGANIZATION
Prentice-Hall, 1975, Ch. 31

Maurer, W.D.
FILE COMPRESSION USING HUFFMAN CODING
In: Computing Methods in Optimization Problems
v. 2, pp. 247-256. Proc. of Conf. in San Ramo, Italy in Sept. 1968. Academic Press, N.Y. 1969

McCarthy, J.P.
AUTOMATIC FILE COMPRESSION
In: International Computing Symposium, 1973 (eds A. Gunther, B. Levrat, and H. Lipps) American Elsevier N.Y., 1974, pp. 511-516

Minsky, M. and Papert, S.
PERCEPTIONS: AN INTRODUCTION TO COMPUTATIONAL GEOMETRY
MIT Press, 1969, Section 12.6

Mommens, J.H. and Raviv, J.
CODING FOR DATA COMPACTION
IBM Research Report No. RC5150, Yorktown Hgts, N.Y. Nov. 26, 1974

Mulford, J.E. and Ridall, R.K.
DATA COMPRESSION TECHNIQUES FOR ECONOMIC PROCESSING OF LARGE COMMERCIAL FILES
Proc. of the Symposium on Information Storage and Retrieval, ACM, pp. 207-215, 1971

Nugent, W.R.
COMPRESSION WORD CODING TECHNIQUES FOR INFORMATION RETRIEVAL
Jnl. Library Automation, v. 1, pp. 250-260, 1968

Ott, G.
COMPACT ENCODING OF STATIONARY MARKOV SOURCES
IEEE Trans. on Information Theory, v. IT-13, pp. 82-86, 1967

Pollerman, T.A.
DATA COMPACTION SYSTEM
HLSUA Library No. GES 1075, September 1972

Pratt, F.
SECRET AND URGENT, THE STORY OF CODES AND CIPHERS
Blue Ribbon Books, New York, 1939

Raviv, J.
DATA COMPACTION USING MODIFIED VARIABLE-LENGTH CODING
United States Patent, 3,675,211, July 4, 1972

Raviv, J., Wesley, M.A., Sommers
DATA COMPACTION USING VARIABLE-LENGTH CODING
United States Patent, 3,675,212, July 4, 1972.

Ruecking, F.H., Jr.

BIBLIOGRAPHIC RETRIEVAL FROM BIBLIOGRAPHIC INPUT: THE HYPOTHESIS AND CONSTRUCTION OF A TEST
Jnl. of Library Automation, v. 1, pp. 227-238, 1968

Ruth, S.S., and Kreutzer, P.J.

DATA COMPRESSION FOR LARGE BUSINESS FILES
Datamation, Sept, 1972, pp. 62-66

Schaltwijk, J.P.M.

AN ALGORITHM FOR SOURCE CODING
IEEE Trans. on Information Theory, v. IT-18, pp. 395-399 (1972)

Schwartz, E.S.

A DICTIONARY FOR MINIMUM REDUNDANCY ENCODING
Jnl. of the ACM, v. 10, pp. 413-439, 1963

Schwartz, E.S.

AN OPTIMUM ENCODING WITH MINIMUM LONGEST CODE AND TOTAL NUMBER OF DIGITS
Information and Control, v. 7, pp. 37-44, 1964

Schwartz, E.S. and Kallick, B.

GENERATING A CANONICAL PREFIX ENCODING
Comm. of the ACM, v. 7, pp. 166-169, 1964

Schwartz, E.S. and Kleiboemer, A.J.

A LANGUAGE ELEMENT FOR COMPRESSION CODING
Information and Control, v. 10, pp. 315-333, 1967

Snyderman, M. and Hunt, B.

THE MYRIAD VIRTUES OF TEXT COMPACTION
Datamation, Dec 1, 1970, pp. 36-40

Stoneburner, P.

HALF-WORD INTRARECORD REPEAT SUPPRESSION
Unpublished Memo, PRC, May 1975

Varn, B.

OPTIMUM VARIABLE LENGTH CODES
Information and Control, v. 19, pp. 289-301, 1971

Villers, J.J. et al

BIBLIOGRAPHY OF DATA COMPACTION AND DATA COMPRESSION LITERATURE WITH ABSTRACTS
Navy Fleet Material Support Office, Mechanicsburg, PA., Feb 1, 1971, NTIS, AD 723 525

Wagner, R.A.

COMMON PHASES AND MINIMUM-SPACE TEXT STORAGE
Comm. of the ACM, v. 16, pp. 148-152, 1973

Weaver, A.C.

DATA COMPRESSION FOR CHARACTER STRINGS
Univ. of Illinois, July 1974, NTIS: PB 234 775

Wells, M.

FILE COMPRESSION USING VARIABLE LENGTH ENCODINGS
Computer Journal, Nov 1972, pp. 308-318

White, H. E.

PRINTED ENGLISH COMPRESSION BY DICTIONARY ENCODING
Proc. IEEE, v. 55, pp. 390-396, 1967