# Data Compression Using Hierarchical Dictionaries

Daniel K. Chang

*The Graduate School and University Center, The City University of New York, New York*

This paper describes a new data compression/expansion algorithm which overcomes some of the shortcomings of the best algorithms known. The algorithm uses multiple (two will be used as an example) dictionaries for storing previously encountered strings of the data file to be compressed. The first dictionary is a short one; it has 128 entries and needs seven bits as its address. The second dictionary is a long one; it has 32,768 entries and requires 15 bits as its address. Each of these two dictionaries is initialized with frequently occurring strings. An input data file is compared with the dictionaries so that the longest substring which can be found in a dictionary will give its position in the dictionary as an output code. Compression and expansion use the same procedure to form and update dictionaries. A compressed result consists of a series of pointers to the dictionaries with a byte as a pointer to the short dictionary and two bytes as a pointer to the long dictionary. During compression/expansion, the more frequently occurring strings will be dynamically swapped into the short dictionary. The two dictionaries are used as scratch pads during the time of program execution and need not be stored/transmitted otherwise.

## 1. INTRODUCTION

Data compression/expansion is a very useful tool in modern computer technology to effectively store and transmit an ever-increasing large amount of information.

This paper describes a universal data compression algorithm. It competes favorably with other compression algorithms in balance of performance, speed, and space requirement. For a survey of data compression see Lelewer and Hirschberg [6].

Currently, there are three major compression methods: Huffman, Arithmetic, and Lempel-Ziv (LZ) [7, 11, 12] codings. The LZ coding is more popular because of its speed and space requirement; its performance is also good compared with other codings [9].

The LZ algorithm uses the method of substitution. A variation of LZ algorithm, the Lempel-Ziv-Welch (LZW) algorithm [10], is briefly described here. LZW compressor builds a string table which contains strings which occur in the input data file. Assume all single character strings are also in the string table. Suppose a string $\alpha$ is read from the input data file and is matched with a string in the string table (initially, $\alpha$ is a single character, hence there must be a match). Then an extension character K is read from the data file, LZW searches the string table to see whether there is a string $\alpha$K: if there is none, then the position of $\alpha$ in the string table is outputed as an output code and the string $\alpha$K is put in the string table (this is called the "*greedy method*"), or else make $\alpha$K to be the new $\alpha$ and a new character K is read from the data file. The above process is repeated until the end of the input data file is reached. LZW decompressor mimics the process of the compressor in forming the string table and outputing a character string from each code of the compressed file by consulting the string table.

## 2. DESIGN CONSIDERATION OF THE NEW ALGORITHM

The objective of this paper is to design a universal compression algorithm that can compress a wide variety of data files without prior knowledge of their statistics. An *adaptive* scheme is preferred since a good nonadaptive scheme often requires two passes to process the input data: in the first pass it gathers statistics and in the second pass it performs the compression job. A good adaptive scheme executes at most only marginally worse than a two-pass nonadaptive scheme, most often the former is much better because it does not need to store or transmit the statistics. Of course, the speed of the former is much better than that of the latter. The LZW algorithm can remember long strings in its string table of reasonable size of 4095 entries. The algorithm which is described in this article is a modified version of LZW and corrects LZW's major shortcomings. It will be called **LZWC** algorithm in the following discussion.

*Address correspondence to Daniel K. Chang, P.O. Box 791, Holmdel, NJ 07733.*



0164-1212/91/$3.50

## 3. ADVANTAGES OF THE PROPOSED NEW LZWC ALGORITHM

In LZW, if a string is in the string table, then all its proper prefix substrings are also in the string table. Therefore, a potential problem with LZW is the overflow of its string table. To save storage space, LZWC does not include all proper prefixes of a string which is in a dictionary. LZWC uses multiple dictionaries[1] instead of the one used in LZW. To illustrate the LZWC algorithm, an example of two dictionaries is used: a short dictionary contains frequently used strings and can be referenced by seven bits, and a long dictionary contains less frequently used strings and can be referenced by 15 bits (other lengths, such as 11 and 13 bits, can also be studied). The algorithm appends a bit of value zero to the short pointer of seven bits; and appends a bit of value one to the long pointer of 15 bits. Therefore, an output code is either one byte or two bytes for each input string. The string tables are initialized with all single character strings, some frequently occurring character strings and some strings for images. An image consists of lines of pixels or dots. A black pixel is coded as value one of one bit and a white pixel is coded as value zero of one bit. For example, a black run length of 32 pixels will be represented by hex symbol string **FFFFFFFF** since each **F** represents four black pixels. Data compression at the beginning of the process can be improved even without sufficient statistics of the data file at that time. Each dictionary is organized in the binary tree form of *lexicographic order* [1]. Each node of the tree contains the following fields: length of the string, the first character of the string, a pointer to the remaining characters, a pointer to its left son, a pointer to its right son, a pointer to its father, two pointers to doubly link other nodes in the order of frequency of occurrence, a count that describes the current occurrences of the string in the input data file.

The value of the field "length of the string" is used to determine the length of the string without actually storing the characters in the storage if the remaining characters are the same as the first character. The pointers linking other nodes in the order of frequency count and the count itself are used to select the least recently used entry in a dictionary for replacement.

By using the above data structure, string matching, inserting, replacing, and swapping can be performed efficiently; therefore, compression and expansion can be reasonably fast. Since the LZWC algorithm uses replacement strategy for a new string if no empty slot can be found from the dictionaries, there is no overflow problem with the LZWC algorithm, and it can handle a very large data file without requiring to reset the string tables.

To achieve the best compression ratio, all entries of the dictionaries should be initially filled with frequently occurring strings.

[1] From now on each of these two dictionaries can also be called a tree because it is linked together in a binary tree; they can also be called string tables because they are mainly two tables that store strings.

## 4. LZWC ALGORITHM

The LZWC algorithm is an adaptive data compression/expansion scheme, which uses two dictionaries for storing strings. A short dictionary has 128 entries and requires an address of seven bits to point to any of its entries. A long dictionary has 32,768 entries and needs an address of 15 bits to point to any of its entries. These two dictionaries are initialized with entries of all 256 permutations of eight bits (because of the popularity of 8/16/32 bits computers including PCs) and other frequently occurring strings based on the statistics of the English language and graphical images. The short dictionary, of course, contains still more frequently occurring strings. Each of these dictionaries is organized in a binary tree of lexicographic order. An input data file to be compressed is read character by character such that a longest substring $\alpha$ which matches an entry in one of the two dictionary trees is obtained, and the position of the dictionary is outputed; for short dictionary, one bit of value zero is appended to the front of the other seven bits, for long dictionary one bit of value 1 is appended to the front of the other 15 bits. Therefore, the output codes pointing to the short and long dictionaries are always one byte or two bytes respectively. The frequency count of the string $\alpha$ in the dictionary is incremented every time $\alpha$ is referenced. If the string $\alpha$ is in the long dictionary and its count is greater than any in the short dictionary, then the swapping of these two entries will be performed. A next character K is read from the input file, and the string $\alpha$K will be put into the long dictionary (this is again the "greedy method"). If the long dictionary is already full, the least recently used entry will be replaced by the new entry $\alpha$K. Now, we make K to be the new $\alpha$. The process of compression continues until the end of the input data file is reached. The expander (also known as decompressor) mimics the process of the compressor in forming and updating the dictionaries and outputs a character string from each code read from the compressed data file. The two dictionaries are thus used as local scratch pads during the algorithm execution and need not be stored/transmitted otherwise.

LZWC updates the two dictionaries during the process of compression/expansion. The compressor and

J. SYSTEMS SOFTWARE
1991; 15: 233-238



the expander use the same initial dictionaries and the same algorithm to update dictionaries so that the dictionaries will be synchronized. For readability, the LZWC algorithm of compression/expansion is described below in a pseudo programming language.

There is a special case which we wish to consider. It is described here as an improvement of the LZWC algorithm for a particular case. Assume the input string is $K\mu K\mu K$ where K is a character and $\mu$ is a string. Suppose the string $K\mu$ is already an entry in a dictionary. When we see the second "K," suppose "$K\mu K$" is not in any of the dictionaries, by the "greedy method" the compressor puts "$K\mu K$" in the long dictionary, say entry **y**. Then the substring $K\mu K$ starting from the second "K" would be encoded as **y**. Therefore, the whole string $K\mu K\mu K$ is encoded as **xy**. When the expander receives the code **xy**, from **x**, the expander knows that the string corresponding to **x** is $K\mu$; when the expander sees a **y**, the entry **y** in the dictionary has not been created by the expander yet because the first character "K" of the string corresponding to code **y** has not been decoded yet. But we know in this case that the code **y** would point beyond the current entries if the long dictionary is not full yet, and the string corresponding to **y** must be $K\mu K$ where $K\mu$ is the previous substring which has been decoded. Therefore the expander will put $K\mu K$ into the long dictionary as **y** entry of the dictionary. When the dictionaries are already full and the compressor sees the string "$K\mu K\mu K$" as described above, the compressor will encode the string "$K\mu K\mu K$" as **xx** for the substring "$K\mu K\mu$" and then put "$K\mu K$" in the long dictionary by replacing the least recently used entry. The expander also follows this procedure.

### A. The Pseudo Code of the LZWC Compression.

```
Initialize the two dictionaries with single-character
strings, other more frequently occurring strings for
English text and some strings for representing graphical
images by using the statistics data file. Each dictionary
is in binary tree form.
α < −'\0'.      /* α is the current substring of the
                   input file, it is initialized with an
                   empty string '\0' */
μ.      /* μ is a substring of α and μ can be found in
           either the short or the long dictionaries */
flag < −0.      /* A flag for putting a string in the
                   long dictionary in the next run by the
                   greedy method */
ß.      /* A buffer for storing a string by greedy method
           */
While (the end-of-file of the input file not reached ) do
    {
    /* Loop until the end of input file is reached */
    Read next character K of the input data file.
    While (αK is found to be an entry in a dictio-
           nary) do
           {
           μ < −αK.
           α < −αK.
           If K is empty, then break, i.e., exit the
           innermost while loop.
           Read next character K from the input
           data file.
           If K is empty, then break.
           }
    saved_code < −position of μ in the dictionary.
    If the dictionary is the short one, set the 8th
    significant bit of saved_code to be 0, and output
    its lower byte; else set its 16th significant bit to
    1, and output the two bytes of saved_code.
    If K is empty then compression is finished and
    we are done.
    Increment the count of occurrences of the entry
    μ.
    If (flag is equal to 1)
           {
/* Use the string ß saved by the greedy method in
   last run */
           If ß is not in any dictionary, put the
           string ß in the long dictionary by replac-
           ing the least recently used string in that
           dictionary.
           }
    /* Now, use the greedy method to try to put the
    string μK into the long dictionary. */
    If the long dictionary is not full, then
           {
           Put the string μK into the next available
           slot of the long dictionary.
           }
           else
           {
           /* The long dictionary is full */
           /* Set flag so that we wait until the next
           cycle time to put the greedy string in the
           long dictionary as explained above for
           the case of KμKμK */
           ß < −μK.    /* save the string for the
                          use in the next cycle */
           flag < −1.  /* Set flag, so that we can
                          do next time */
           }
           α < −K.     /* start again with the new
                          α */
           }
```

**B. The Pseudo Code of the LZWC Expansion.**

Initialize the two dictionaries with single-character strings, other more frequently occurring strings and some strings for representing graphical images in the same way as the compressor inserts strings in the dictionaries by using the same statistics data file. Each dictionary is in binary tree form, same as in compression.

```
ß < − '\0'.  /* A buffer for storing a string by
      greedy method */
Read code i of one byte of the input compressed file.
While the code i is not empty do .
   {
   If the most significant bit of the code i is 0
      {
      From the ith entry of the short dictionary
         {
         Find the string μ and output it.
         Increment the count of μ in the
         dictionary.
         }
      }
      else
      {
      /* It is the long dictionary */
      Read next byte j.
      Strip the most significant bit of the code i.
      m < − i*256 + j.
      If the long dictionary is full or m is not out of
      bound of the current entries, then
         {
         From the mth entry of the long dictio-
         nary find the string μ and output it.
         Increment the count of μ in the diction-
         ary.
         If μ's count is greater than that of an
         entry in the short dictionary, then swap
         these two entries.
         }
      else
         {
         /* KμK case */
         /*  ß is the previously saved substring,
            and F denotes the first character of
            ß */
         Output the string ßF.
         μ < − ßF.
         }
      }

/* K denotes the first character of μ */
If ßK is not in any dictionary, put ßK in the long
dictionary either in the empty slot or by replacing the
least recently used entry.
ß < − μ.
Read code i of the input compressed file.
}
```

## 5. A STATISTICS DATA FILE FOR BUILDING INITIAL DICTIONARIES

A statistics data file of strings with high frequency of occurrences is created according to their descending order of the frequency of occurrences. The file will include all eight-bits combinations, most frequently occurring English words [3, 5], and some run lengths of null characters and hex F characters for images. The statistics data file will be used to form the initial dictionaries in lexicographic order. It is obvious that the same statistics data file must be used by both compressor and expander to build the same initial dictionaries.

Assume the part of the statistics data file that builds the long dictionary is shown as follows:

> . . . the be of and in he to have it for they with not then on she has at by this we you from do . . .

The tree structure of a part of the long dictionary is shown in the appendix after the long dictionary is constructed.

Knuth [4] has an algorithm of forming an optimum binary search tree (binary tree with lexicographic order) if the statistics of the entries are known.

## 6. ERROR SUSCEPTIBILITY

Because an output code of the LZWC algorithm is either one byte or two bytes for each input substring, one type of bit error, amplitude error [6], will not propagate easily if the most significant bit of a byte/word pointer is not corrupted.

## 7. EXAMPLE

The two dictionaries can be initialized with the statistics file. To demonstrate the effectiveness of the LZWC algorithm, the following sample English text file in Table 1 is to be compressed:

If we assume a tab before every paragraph, a single space between adjacent words on a same line, and a CR and an LF after each line, the original text occupies 1473 bytes storage. Since the two dictionaries can have 32K words, we can reasonably assume almost all En-

**Table 1. Abraham Lincoln's Address at Gettysburg, Pennsylvania**

Four score and seven years ago, our fathers brought forth on this continent a new nation, conceived in liberty and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war, testing whether that nation or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war. We have come to dedicate a portion of that field, as a final resting place for those who here gave their lives that that nation might live. It is altogether fitting and proper that we should do this.

But, in a larger sense, we can not dedicate—we can not consecrate—we can not hallow this ground. The brave men, living and dead who struggled here, have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember what we say here, but it can never forget what they did here. It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced. It is rather for us to be here dedicated to the great task remaining before us—that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion—that we here highly resolve that these dead shall not have died in vain—that this nation, under God, shall have a new birth of freedom—and that government of the people, by the people, for the people shall not perish from the earth.

glish words in Table 1 can be found from one of the dictionaries. The final result is that the compressed file needs 310 one-byte pointers and 161 two-byte pointers. Also we find that the number of distinct one-byte pointers is less than 128, the size of the short dictionary. Therefore, the total storage required is 310 + 161*2 = 632 bytes. The compression ratio = 1473/632 = 2.33, and the compression percentage = 632/1473 = 42.9%.

A popular shareware compression program PKZIP/PKUNZIP [9] version 0.92, which uses the LZW compression algorithm, was used to experiment the aforementioned example file. The compressed result has 978 bytes. Among these 978 bytes, 104 bytes are used for header information, only the remaining 978 − 104 = 874 bytes are for the file itself. Hence, the compression ratio = 1473/874 = 1.69, and the compression percentage = 874/1473 = 59.3%.

By comparing the result from PKZIP and that from LZWC, we can certainly see that the result of LZWC is substantially better than that of the PKZIP program.

## 8. REDUNDANCY

On one hand, the aim of data compression is to try to reduce as much redundancy of a data file as possible; on the other hand, a damaged file which has been compressed very effectively can hardly be deciphered. Therefore, some controlled redundancy can be purposely put into the compressed file such that we can recover the original file from its damaged compressed counterpart if the damage does not exceed a certain threshold. One such method was suggested by Rabin [8], and was called *Information Dispersal Algorithm (IDA)*. IDA can split a file F of length L into L/m pieces, each of length m. By manipulating these pieces, n (> m) messages can be formed, any m messages among these n can be used to reconstruct the original file. For example, suppose we assume n = 17 and m = 16, a file F of length L = 64 can be splitted and manipulated into n = 17 pieces, each of which is of length L/16 = 64/16 = 4, if any one piece is lost or damaged the other 16 pieces can be used to easily reconstruct the original file. In this case, the amount of redundancy is (17 − 16)/16 = 6.25%. For a simple illustration of IDA, see Ref. [2].

## 9. CONCLUSION

The LZWC algorithm is a universal data compression, i.e., it compresses various kinds of data files effectively without prior knowledge of the statistics of the data files. The LZWC algorithm is adaptive, i.e., it can gather the statistics of the data file during the compressing process. The statistics file is decided beforehand. The dictionaries contain most recently used entries, and they will never overflow. On the other hand, the LZW algorithm can easily flood its string table of size 4095 entries. By organizing the dictionaries in *hierarchical* levels, LZWC uses only one byte as a pointer to a most frequently occurring string, and two bytes as a pointer to a less frequently occurring string. As long as the most significant bit of a pointer is not corrupted, bit corruption will be contained and thus will not easily propagate. As the example of compressing the Gettysburg Address shows, even though the text file is short (1473 bytes), an impressive compression result is obtained. By comparing the popular shareware program PKZIP/PKUNZIP (an LZW implementation) with the LZWC algorithm, we know that the performance of the latter is significantly better than that of the former. Unlike adaptive Huffman and adaptive arithmetic codings, the LZWC algorithm needs to update at most one branch of the tree of each dictionary when the statistics of any entry changes substantially. Since the LZWC algorithm is universal, the precious short dictionary can contain some special symbols for other data files. Other data files, such as C programs, can also be effectively compressed.

This paper describes two-level hierarchical dictionaries of fixed sizes as an example. Other variations of the LZWC algorithm can also be investigated. Three of them are suggested as follows:

A. The first variation of the LZWC algorithm: The short dictionary still having 128 entries and thus need-

ing seven bits as its address, but the long dictionary now has 8192 entries and needs 13 bits as its address. If we try this variation of the LZWC algorithm to compress the Gettysburg Address, and we assume the words are still in the dictionaries (a reasonable assumption), then every pointer to the long dictionary is now 13 instead of 15, then we need 14 bits in the code instead of 16 for the long dictionary. This results 310 + 161*2*(14/16) = 592 bytes of storage. Now the compression ratio is 1473/592 = 2.49, and the compression percentage is 592/1473 = 40.2%.

B. The second variation of the LZWC algorithm: The short dictionary still has 128 entries and needs seven bits as its address, but the long dictionary initially has 8192 entries and needs 13 bits as its address, the compressor (as well as the expander) will decide whether a longer dictionary, say a dictionary of 32768 entries, is required in future compression after the long dictionary is filled.

C. The third variation of the LZWC algorithm: There are three-level hierarchical dictionaries. The first dictionary has eight entries and has three bits as its address. The second dictionary has 64 entries and has six bits as its address. The third dictionary has 16384 entries and has 14 bits as its address. The code generated by the compressor determines which dictionary it points to. If the most significant bit of a code is zero, then the next three bits are used as a pointer to the first dictionary; else if the second most significant bit is zero, then the next six bits are used as a pointer to the second dictionary or else the next 14 bits are used as a pointer to the third dictionary.

ACKNOWLEDGMENT

I would like to thank Professor Jacob Rootenberg for originally arousing my interest in data compression and for his generous advice, encouragement, and comments for preparing this article.

## REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
2. D. K. Chang and J. Rootenberg, *A Simple Description of Efficient Dispersal of Information for Fault Tolerance*, ISMM International Conference on Parallel and Distributed Computing, and System, New York, October, 1990.
3. W. N., Francis, H. Kucera, and A. W. Mackie, *Frequency Analysis of English Usage: Lexicon and Grammar*, Houghton Mifflin Company, Boston, 1982.
4. D. E. Knuth, Optimum Binary Search Trees, *Acta Inf.* 1, 14-25 (1971).
5. H. Kucera and W. N. Francis, *Computational Analysis of Present-Day American English*, Brown University Press, Providence, R.I., 1967.
6. D. A. Lelewer and D. S. Hirschberg, Data Compression, *ACM Computing Surveys*, 19, 261-296 (1987).
7. V. S. Miller and M. N. Wegman, Variations on a theme by Ziv and Lempel, *IBM Res. Rep.* RC 10630 (#47798), 7/31/84.
8. M. O. Rabin, Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance, *Journal of ACM*, 36, 335-348 (1989).
9. S. J. Vaughan-Nichols, Saving Space, *BYTE*, pp. 237-243, March 1990.
10. T. A. Welch, A Technique for High-Performance Data Compression, *Computer*, 17, 8-19 (1984).
11. J. Ziv and A. Lempel, A Universal Algorithm for Sequential Data Compression, *IEEE Trans. Info. Theory* IT-23, 337-343 (1977).
12. J. Ziv and A. Lempel, Compression of Individual Sequences Via Variable-Rate Coding, *IEEE Trans. Info. Theory*, IT-24, 530-536 (1978).

## Appendix. Tree Structure of a Part of the Long Dictionary