Persistent Memory Requirement Reduction, Required in Text Data Storage Using Lookup Method

By

Md. Abul Kalam Azad
Shabbir Ahmad
Francis Palma

Department of Computer Science and Engineering,
International Islamic University Chittagong.
December 2004

Persistent Memory Requirement Reduction, Required in Text Data Storage Using Look-up Method

By

Mohammad Abul Kalam Azad C011054

Shabbir Ahmad C011051
Francis Palma C013038

Submitted to

International Islamic University Chittagong, Bangladesh.
in partial fulfillment of the requirements for
B.Sc. in Computer Science and Engineering

Acknowledgement

The authors would like to express their sincerest gratitude and profound indebtedness to the supervisor S. M Kamruzzaman, Assistant Professor, Department of Computer Science & Engineering (CSE), International Islamic University Chittagong (IIUC), for his patronizing, guidance, encouragement, insightful advice, and endless patience throughout the progress of the work without which the thesis would not have been successful.

The authors would like to give heartiest thanks to their Dean and Head, Dr. Md. Nurul Islam, Professor, Department of CSE and External Dr. M. Shahadat Hossain, Assistant Professor, Department of CSE, University of Chittagong, for their valuable suggestions.

The authors would like to express thanks to their teacher Md. Sazzad Hossain, Assistant Professor, Department of CSE, Md. Mahadi Hasan, Lecturer, Department of CSE, Md. Zia Uddin, Lecturer, Department of CSE and Md. Abdullah Al Mamun, Lecturer, Department of CSE for their wonderful advice and co-operation.

Abstract

For storing a word or the whole text segment, we need a huge storage space. Typically a character required 1 Byte for storing himself on memory. Compression of data is very important for data management. In data compression for text type data lossless data compression is needed. Here we suggest a lossless data compression for text data compression. The proposed compression method will compress the text segment or the text file based on a word lookup table not using traditional indexing system or currently available compression methods. The word lookup table will be the part of the operating system and the compression will be done by the operating system. According to this method each record will be replaced by an address value. This method can reduce the size of any persistent memory required for text data quite effectively.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A text segment is a collection of words and a word consists of characters. All the characters are unique and they are the basic units of word. That's why to store a text segment it is needed to store all the words separately. To store a word, all the characters that the word contains needed to be stored. For this type of storing mechanism a huge amount of disk space is needed and in the current world this technique is used. But this type of system of storing text makes the text segment heavy in size. Let us suppose that a text segment contains some word "n" times, which is 8 characters in lengths (take it as average length) then for repeated presence of the same word of "n" times we need n*8 Bytes. If we do some sort of indexing within the text segment, then the text segment size can be reduced. But this process is still not effective because it needs extra space to make the indexing table and sometimes it may increase the file size rather than decreasing. Now if we can make a word index table which will be used to index the text segment for reducing text segment size and the index table will not be a part of the text segment then it will be an effective way to reduce the text segment portion size. This proposed method decreases the persistent memory size approximately 50%. In this process we will store the text data in persistent memory as a binary file but during the time open and modification it will create a temporary file which will communicate with the reduced and expended file.

1.1 Motivation

All the current operating system is based on the previous UNIX system. Early UNIX system was developed for work stations which have a slow processor and a very small amount of memory space. For that reason, in early UNIX systems the text file storage or transmission was based on ASCII format [13,14]. In the current world we have high powerful processors and high capability storages devices not only in the micro computer but also in PDA's. That's why it is not difficult to store or to manipulate a file. Here we will use the concept that the operating system has a lookup table of words and with a reference address that will be consist of some bits as like as color lookup table. The operating system will replace the words in the text file by the reference according to the lookup table [17]. If the relative indexing address of word need fewer bits the text size can be reduced. In this proposal we used this method to reduce the text segment size. And in file transfer as the operating system will contain the word lookup table the indexed text portion can be easily transferred from one terminal to another terminal.

1.2 Background

These are a total of 16 million colors in real world. The representation of those colors is a very mammoth task and also storing the values of those colors is a quite memory consuming task. This situation is handled in Computer Graphics using color lookup table. That table is able to index between 16 million colors. The amount of color that the table is able to handle is based on the number of bits that the table needs to address the colors. This lookup table helps to represents those color using lesser bits. From this basis we thought that if color lookup table can compress the address needed to allocate for the color so why not the technology can be implemented for the case of word storing and representation. We thought this method will be that much feasible that is for color lookup. Finally we found our anticipation true

1.3 Text Data Storage Using Look up Method

The lookup table is, a special tabular data file containing additional attributes for features stored in an associated feature attribute table. The table can be an external attribute table or an info table that describes coverage features. Also a lookup table is where document property values are defined. In the client interface, a lookup table pops up to display the possible entries for a field. And a word lookup table is a special tabular data file containing the text dimension of a word as an attribute of an address, which is used to pop up text to display the possible text data for a field [3, 6]. For our purpose we decide to use a 19-bit word lookup table.

1.4 Challenges of Text Data Storage Minimization

As any language is not a rigid body that is. Languages are always expanding and day by day is enriching with the invention of new words and also by adoption new words. So we faced some challenges in choosing the boundary of languages. More over we have to consider the following things too.

1.4.1 Spelling Mistake

The base of the our proposed methodology is the word in the English dictionary but due to spelling mistake there may be lots of mistakes which will not be matched with no entries in the dictionary.

1.4.2 Technical and Biological Term

Technical word, Chemical names and names of different species of animals is in amount more than 1 millions but those words are not generally added in the dictionaries. So we needed decide whether we will allow those names in the lookup table.

1.4.3 Other Language words

Sometimes word from other language that have not been yet adopted in the English may be found in the text segment. Though the word may be common in use it will be considered as a special word.

1.5 Goal of Our Research

Nowadays compression is the urgent need for communication. For information technology compression is needed in all area. That’s why we motivated to doing research paper in compression system. When we need to transfer something through communication medium such as text segment, at first we need to compress the whole segment into a compact size for accelerate the transferring and make sure that the transferred data must be noise free. On the other hand Personal Digital Assistant (PDA), Palmtop or other small electronic device that bearing short storage, sometimes required more data to store. But there is storage problem. So, if we need to store more data like text, we need compressed data. That's only required small storage if we use our proposed idea of compression. Compression of text segment has a popular demand and practical applications. The application area will be any text segment. Nowadays, peoples have more than gigabytes of text data. By using a word lookup table, which will use 6.00 MB (approximately) the text data size in the work station memory can be reduced to half. That will save a huge amount of memory space. It is still difficult to transfer file or data through communication medium. The reason is that the signal capacities of the carriers are not sufficient enough. This problem is deeply felt in internet communication. One of the main features of word lookup table will be that it will decrease the text size which will increase the portability of any text segment and make any text segment able to access at faster speed. A mail server or a web server contains a huge amount of text data. Using the lookup table, the text data size can be reduced a lot, which will reduce the memory space occupied in the storage.

Chapter 2

Literature Review

In this chapter we have tried to introduce different terminologies that are required for our thesis. The Proposed Persistent Memory Requirement Reduction, Required in Text Data Storage Using Look-up Method is a mammoth system because here the there are lots of data and those are needed to be stored perfectly.

2.1 Lookup Table

A look up table "LUT" is actually an Sram or a flash memory (or at least the first LUT was), most LUTs nowaday are 4X1 which means that they are capable of implmenting any 4 -input to 1-output function. "4-address RAM with one bit o/p". Virtually any combinational logic can be implemented with cascaded and parallel LUTs. Lookup table is, a special tabular data file containing additional attributes for features stored in an associated feature attribute table. The table can be an external attribute table or an info table that describes coverage features. Also a lookup table is where document property values are defined. In the client interface, a lookup table pops up to display the possible entries for a field. And a word lookup table is a special tabular data file containing the text dimension of a word as an attribute of an address, which is used to pop up text to display the possible text data for a field [3, 6].

2.2 ASCII Character Characterization

Acronym for the American Standard Code for Information Interchange. Pronounced ask-ee, ASCII is a code for representing English characters as numbers, with each letter assigned a number from 0 to 127. For example, the ASCII code for uppercase M is 77. Most computers use ASCII codes to represent text, which makes it possible to transfer data from one computer to another. For a list of commonly used characters and their

ASCII equivalents, refer to the ASCII page in the Quick Reference section. Text files stored in ASCII format are sometimes called ASCII files. Text editors and word processors are usually capable of storing data in ASCII format, although ASCII format is not always the default storage format. Most data files, particularly if they contain numeric data, are not stored in ASCII format. Executable programs are never stored in ASCII format. The standard ASCII character set uses just 7 bits for each character. There are several larger character sets that use 8 bits, which gives them 128 additional characters. The extra characters are used to represent non-English characters, graphics symbols, and mathematical symbols. Several companies and organizations have proposed extensions for these 128 characters. The DOS operating system uses a superset of ASCII called extended ASCII or high ASCII. A more universal standard is the ISO Latin 1 set of characters, which is used by many operating systems, as well as Web browsers.

2.5 Data Compression

Data compression is the methodology of reducing the memory space to represent those data. In the current world data compression is greatly needed. Because daya by day the amount of data is increasing.

2.5.1 Lossy Data Compression

A lossy data compression method is one where compressing a file and then decompressing it retrieves a file that may well be different to the original, but is "close enough" to be useful in some way. This type of compression is used a lot on the Internet and especially in streaming media and telephony applications. These methods are typically referred to as codecs in this context. There are two basic lossy compression schemes:

In lossy transform codecs, samples of picture or sound are taken, chopped into small segments, transformed into a new basis space, and quantized. The resulting quantized values are then entropy coded.

In lossy predictive codecs, previous and/or subsequent decoded data is used to predict the current sound sample or image frame. The error between the predicted data and the real data, together with any extra information needed to reproduce the prediction, is then quantized and coded.

In some systems the two techniques are combined, with transform codecs being used to compress the error signals generated by the predictive stage. The advantage of lossy methods over lossless methods is that in some cases a lossy method can produce a much smaller compressed file than any known lossless method, while still meeting the requirements of the application. Lossy methods are most often used for compressing sound or images. In these cases, the retrieved file can be quite different to the original at the bit level while being indistinguishable to the human ear or eye for most practical purposes. Many methods focus on the idiosyncrasies of the human anatomy, taking into account, for example, that the human eye can see only certain frequencies of light. The psychoacoustic model describes how sound can be highly compressed without degrading the perceived quality of the sound. Flaws caused by lossy compression that are noticeable to the human eye or ear are known as compression artifacts.

2.5.2 Lossless Data Compression

Lossless data compression is a class of data compression algorithms that allow the original data to be reconstructed *exactly* from the compressed data. Contrast with lossy data compression. Lossless data compression is used in software compression tools such as the highly popular ZIP file format, used by PKZIP, WinZip and Mac OS 10.3, and the Unix programs bzip2, gzip and compress. Other popular formats include Stuffit and RAR. Lossless compression is used when it is important that the original and the decompressed data are exactly identical, or when no assumption can be made on whether certain deviation is uncritical. Typical examples are executable programs and source code. Some image file formats, notably PNG, use only lossless compression, while others like TIFF and MNG may use either lossless or lossy methods. GIF uses a technically lossless compression method, but most GIF implementations are incapable of representing full color, so they quantize the image (often with dithering) to 256 or fewer colors before encoding as GIF. Color quantization is a lossy process, but reconstructing the color image and then re-quantizing it produces no additional loss. (Some rare GIF implementations make multiple passes over an image, adding 255 new colors on each pass.)

2.5.2.1 Run Length Encoding

One of the simplest forms of data compression is known as "run length encoding (RLE)", which is sometimes known as "run length limiting (RLL)". Suppose you have a text file in which the same characters are often repeated, one after another. This redundancy provides an opportunity for compressing the file. Compression software can scan through the file, find these redundant strings of characters, and then store them using an escape character (ASCII 27), followed by the character and a binary count of the number of times it is repeated. For example, the 50 character sequence:

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX that's all, folks! --

can be converted to:

<ESC>X<31> that's all, folks!

This eliminates 28 characters, compressing the text by more than a factor of two. Of course, the compression software must be smart enough not to compress strings of two or three repeated characters, since for three characters run length encoding would have no advantage, and for two it would actually increase the size of the output file. As described, this scheme has two potential problems. First, an escape character may actually occur in the file. The answer is to use two escape characters to represent it, which can actually make the output file bigger if the uncompressed input file includes lots of escape characters. The second problem is that a single byte cannot specify run lengths greater than 256. This difficulty can be dealt with by using multiple escape sequences to compress one very long string. Run length encoding is actually not very useful for compressing text files, since a typical text file doesn't have a lot of long, repetitive character strings. It is very useful, however, for compressing bytes of a monochrome image file, which normally consists of solid black picture bits, or "pixels", in a sea of white pixels, or the reverse. Run-length encoding is also often used as a preprocessor for other compression algorithms. As the next chapter explains, for example, it is used as one of the many pieces of the JPEG image compression scheme.

2.5.2.2 HuffmanCoding

A more sophisticated and efficient lossless compression technique is known as "Huffman coding", in which the characters in a data file are converted to a binary code, where the most common characters in the file have the shortest binary codes, and the least common have the longest. To see how Huffman coding works, assume that a text file is to be compressed, and that the characters in the file have the following frequencies:

A:	29
B:	64
C:	32
D:	12
E:	9
F:	66
G:	23

In practice, we need the frequencies for all the characters used in the text, including all letters, digits, and punctuation, but to keep the example simple we'll just stick to the characters from A to G. The first step in building a Huffman code is to order the characters from highest to lowest frequency of occurrence as follows:

66	64	32	29	23	12	9
F	B	C	A	G	D	E

First, the two least-frequent characters are selected, logically grouped together, and their frequencies added. In this example, the D and E characters have a combined frequency of 21:

```
                                        :
                                     ........
                                     :  21  :
                                     :      :
  66     64     32     29     23     12     9
  F      B      C      A      G      D      E
```

This begins the construction of a "binary tree" structure. We now again select the two elements the lowest frequencies, regarding the D-E combination as a single element. In this case, the two elements selected are G and the D-E combination. We group them together and add their frequencies. This new combination has a frequency of 44:

```
                                    :
                              ............
                              :    44    :
                              :          :
                              :      ........
                              :      :  21  :
                              :      :      :
  66     64     32     29     23     12     9
  F      B      C      A      G      D      E
```

We continue in the same way to select the two elements with the lowest frequency, group them together, and add their frequencies, until we run out of elements. In the third iteration, the lowest frequencies are C and A:

```
                                    :
                              ............
                              :    44    :
                   :          :          :
                ........      :      ........
                :  61  :      :      :  21  :
                :      :      :      :      :
  66     64     32     29     23     12     9
  F      B      C      A      G      D      E
```

The next iterations give:

```
                      :
               ..............
               :     105    :
               :            :
               :        ..........
               :        :   44    :
               :        :         :
            .......     :     .......
:     :     :  61 :     :     :  21 :
:     :     :     :     :     :     :
66    64    32    29    23    12    9
F     B     C     A     G     D     E
```

```
                      :
               ..............
               :     105    :
               :            :
               :        ..........
               :        :   44    :
   :           :        :         :
.......     .......     :     .......
: 130 :     :  61 :     :     :  21 :
:     :     :     :     :     :     :
66    64    32    29    23    12    9
F     B     C     A     G     D     E
```

```
             :
   ....................
   :        235       :
   :                  :
   :           ..............
   :           :     105    :
   :           :            :
   :           :        ..........
   :           :        :   44    :
   :           :        :         :
.......     .......     :     .......
: 130 :     :  61 :     :     :  21 :
:     :     :     :     :     :     :
66    64    32    29    23    12    9
F     B     C     A     G     D     E
```

:

The result is known as a "Huffman tree". To obtain the Huffman code itself, each branch of the tree is labeled with a 1 or 0. It doesn't matter how the 1s and 0s are assigned, though a consistent scheme obviously is easier to deal with:

```
             :
   ....................
   :0                 :1
   :                  :
   :           ...............
   :           :0          :1
   :           :             :
   :           :        ...........
   :           :        :0        :1
   :           :        :         :
.......     .......     :      .......
:0    :1    :0    :1    :      :0    :1
:     :     :     :     :      :     :
F     B     C     A     G      D     E
```

Tracing down the tree gives the "Huffman codes", with the shortest codes assigned to the characters with the greatest frequency:

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

A Huffman coder will go through the source text file, convert each character into its appropriate binary Huffman code, and dump the resulting bits to the output file. The Huffman codes won't get mixed up in decoding. The best way to see that this is so is to envision the decoder cycling through the tree structure, guided by the encoded bits it reads, moving from top to bottom and then back to the top. As long as bits constitute legitimate Huffman codes, and a bit doesn't get scrambled or lost, the decoder will never get lost, either.

There is an alternate algorithm for generating these codes, known as "Shannon-Fano coding". In fact, it preceded Huffman coding and one of the first data compression schemes to be devised, back in the 1950s. It was the work of the well-known Claude Shannon, working with R.M. Fano. David Huffman published a paper in 1952 that modified it slightly to create Huffman coding. Shannon-Fano coding achieves the same results as Huffman coding, but works from the top down, instead of the bottom up. Using the same set of example characters as above, we arrange them in order of frequency:

F B C A G D E
66 64 32 29 23 12 9

Now we divide the set into two parts, as close to equal as possible. The sum of the frequencies of F and B is 130, while the sum of the rest of the characters is 105, and we can't break the set more closely than that:

```
       130  :   105
   ...................
   :                 :
   :                 :
 F   B        C   A   G   D   E
 66  64       32  29  23  12  9
```

We continue breaking down each branch until they can't be broken down further. Breaking down the branch with the F and B is easy, since there's only two characters there. For the other branch, the frequencies of C and A add up to 61, while the frequencies of the rest of the characters in that branch add up to 44, which is again as close to equal as we can get:

```
       130  :   105
   ...................
   :                 :
   :             61  :  44
.......         ..............
:     :         :            :
:     :         :            :
F     B      C   A       G   D   E
66    64     32  29      23  12  9
```

We can complete the tree using the same procedure, and then assign 0s and 1s to the branches as before:

```
       130  :    105
   .....................
   :                   :
   :               61  :  44
   :            ...............
   :            :             :
   :            :          23 : 21
.......      .......     ...........
:     :      :     :     :         :
:     :      :     :     :         :
F     B      C     A     G       D   E
66    64     32    29    23      12  9
```

```
       130  :    105
   .....................
   :0                  :1
   :               61  :  44
   :            ...............
   :            :0            :1
   :            :          23 : 21
   :            :        ...........
   :            :        :0        :1
   :            :        :         :
.......      .......     :      .......
:0    :1     :0    :1    :      :0    :1
:     :      :     :     :      :     :
F     B      C     A     G      D     E
66    64     32    29    23     12    9
```

The codes that result are exactly the same as would be obtained with Huffman coding. Of course, Huffman codes can be used on any type of data, such as bytes in a graphics file. They are not very effective if the data in the file is strongly random, as the frequencies of different characters or bytes would be close to the same, but random data files are hard to compress by any technique. Fax machines, which transmit a graphics image, use a combination of RLE and Huffman compression to achieve compression ratios of about 10:1. Huffman coding techniques are also used in conjunction with other compression schemes to improve compression ratios, and are used in this fashion with the popular LHA and PKZIP archivers. As the average frequencies of letters in every language are distinctive and well known, a fact which is incidentally of great importance to codebreakers, it is possible to devise a Huffman code table for text files written in a specific language. A receiver can use the same "default" or "canonical" Huffman codes to decompress the message. However, average frequencies are just that, averages, and individual messages will differ from that average to a greater or lesser degree. That means that better compressions can be obtained by analyzing a specific file and building a "custom" Huffman code for that file. This requires that the custom table of Huffman codes be output to the file along with the compressed data stream, to allow the decoder to uncompress the data. Another variation on this idea is known as "adaptive" Huffman coding. In this approach, both the coder and decoder start with a predefined Huffman table, but both keep statistics on the frequency of different characters in the compressed data stream, and modify the Huffman codes on a continuous basis by modifying the Huffman tree as needed. As both the coder and decoder use the same modification algorithm, the decoder will adjust its Huffman table in the same way as the coder, and the two will remain in sync.

2.5.2.3 Arithmetic Coding

Huffman coding looks pretty slick, and it is, but there's a way to improve on it, known as "arithmetic coding". The idea is subtle and best explained by example. Suppose we have a message that only contains the characters A, B, and C, with the following frequencies, expressed as fractions:

A:	0.5
B:	0.2
C:	0.3

To show how arithmetic compression works, we first set up a table, listing characters with their probabilities along with the cumulative sum of those probabilities. The cumulative sum defines "intervals", ranging from the bottom value to less than, but not equal to, the top value. The order in which characters are listed in the table does not seem to be important, except to the extent that both the coder and decoder have to know what the order is.

Table 2.1: Arithmetic Encoding Step 1

Letter	Probability	Interval
C	0.3	0.0 : 0.3
B	0.2	0.3 : 0.5
A	0.5	0.5 : 1.0

Now each character can be coded by the shortest binary fraction whose value falls in the character's probability interval:

Table 2.1: Arithmetic Encoding Step 2

Letter	Probability	Interval	Binary Fraction
C	0.3	0.0 : 0.3	0
B	0.2	0.3 : 0.5	0.375
A	0.5	0.5 : 1.0	0.5

This shows how single characters can be assigned minimum-length binary codes. However, arithmetic coding doesn't stop there and simply translate the individual characters in a message as these binary codes. It takes a subtler approach, assigning binary fractions to complete messages. To start, let's consider sending messages consisting of all possible permutations of two of these three characters. We determine the probability of the two-character strings by multiplying the probabilities of the two characters, and then set up a series of intervals using those probabilities.

Table 2.3: Arithmetic Encoding Step 3

String	Probability	Interval	Binary fraction
CC	0.09	0.00 : 0.09	0.0001 = 1/16 = 0.0625
CB	0.06	0.09 : 0.15	0.001 = 1/8 = 0.125
CA	0.15	0.15 : 0.30	0.01 = 1/4 = 0.25
BC	0.06	0.30 : 0.36	0.0101 = 5/16 = 0.3125
BB	0.04	0.36 : 0.40	0.011 = 3/8 = 0.375
BA	0.1	0.40 : 0.50	0.0111 = 7/16 = 0.4375
AC	0.15	0.50 : 0.65	0.1 = 1/2 = 0.5
AB	0.1	0.65 : 0.75	0.1011 = 11/16 = 0.6875
AA	0.25	0.75 : 1.00	0.11 = 3/4 = 0.75

The higher the probability of the string, in general the shorter the binary fraction needed to represent it.

Obviously, this same procedure can be followed for more characters, resulting in a longer binary fractional value. What arithmetic coding does is find the probability value of an entire message, and arrange it as part of a numerical order that allows its unique identification. Let's stop here and send one of the binary strings defined in the table above to a decoder. We'll arbitrarily select the binary string with the decimal value of 0.21875 from the table above. This value was obtained using the probability values and intervals defined earlier:

The value 0.21875 clearly falls into the interval for "C", so "C" must be the first character. We can then "zoom in" on the characters that follow the "C" by subtracting the bottom value of the interval for "C", which happens to be 0, and dividing the result by the width of the probability interval for "C", which is 0.3:

(0.21875 - 0) / 0.3 = 0.72917

This is a simple shift and scaling operation.

The result falls into the probability interval for "A", and so the second character must be "A". We can then zoom in on the next character by the same approach as before, subtracting the bottom value of the interval for "A", which is 0.5, and dividing the result by the width of the probability interval for "A", which is also 0.5:

$(0.72917 - 0.5) / 0.5 = 0.4583$

This clearly falls into the probability interval for "B", and so the string has been correctly uncompressed to "CAB", which is the correct answer. Unfortunately, this leaves behind a remainder that can be decoded into an indefinitely long string of bogus characters. This appears to be an artifact of using decimal floating-point math to perform the calculations in this example. In practice, arithmetic coding is based on binary fixed-point math, which avoids this problem. One other problem is the fact that the binary fraction that is output by the arithmetic coder is of indefinite length, and the decoder has no idea of where the string ends if it's not told. In practice, a length header can be sent to indicate how long the fraction is, or an end-of-transmission symbol of some sort can be used to tell the decoder where the end of the fraction is. As with Huffman coding, arithmetic coding can also be performed using an adaptive algorithm, with the coder and decoder starting with a predetermined character probability interval table, tallying characters for their actual frequencies as they are encoded or decoded, and then adjusting the probability interval table accordingly. The neat thing about arithmetic coding is that by "smushing" a complete message into a single probability interval value, individual characters can be encoded with the equivalent of fractional values of bits. Huffman coding requires an integer number of bits for each character, and so this is one of the reasons that arithmetic coding is in general more efficient than Huffman coding. Another reason for the efficiency of arithmetic coding is that more "probable" messages compress to shorter binary strings. By the way, at the risk of belaboring the obvious, the probability only depends on the numbers of different characters in the uncompressed file. Their order is not important, since the result of a multiplication does not depend on the order of its factors. The problem with arithmetic coding is that it is very computation-intensive and so it is slow. Huffman and arithmetic coding are sometimes referred to as forms of "statistical coding" or "entropy coding". The term "VLW (Variable Length Word)" also seems to pop up on occasion when discussing such coding techniques, though its exact definition seems a bit unclear.

2.5.2.4 LZ-77 Encoding

Good as they are, Huffman and arithmetic coding are not perfect for encoding text because they don't capture the higher-order relationships between words and phrases. There is a simple, clever, and effective approach to compressing text known as "LZ-77", which uses the redundant nature of text to provide compression. This technique was invented by two Israeli computer scientists, Abraham Lempel and Jacob Ziv, in 1977. LZ-77 exploits the fact that words and phrases within a text file are likely to be repeated. When they do repeat, they can be encoded as a pointer to an earlier occurrence, with the pointer accompanied by the number of characters to be matched. Pointers and uncompressed characters are distinguished by a leading flag bit, with a "0" indicating a pointer and a "1" indicating an uncompressed character. This means that uncompressed characters are extended from 8 to 9 bits, working against compression a little. Key to the operation of LZ-77 is a sliding history buffer, also known as a "sliding window", which stores the text most recently transmitted. When the buffer fills up, its oldest contents are discarded. The size of the buffer is important. If it is too small, finding string matches will be less likely. If it is too large, the pointers will be larger, working against compression. For an example, consider the phrase:
the_rain_in_Spain_falls_mainly_in_the_plain -- where the underscores ("_") indicate spaces. This uncompressed message is 43 bytes, or 344 bits, long.

At first, LZ-77 simply outputs uncompressed characters, since there are no previous occurrences of any strings to refer back to. In our example, these characters will not be compressed:
the_rain_

The next chunk of the message:

in_ -- has occurred earlier in the message, and can be represented as a pointer back to that earlier text, along with a length field. This gives:

the_rain_<3,3> -- where the pointer syntax means "look back three characters and take three characters from that point." There are two different binary formats for the pointer: An 8-bit pointer plus 4-bit length, which assumes a maximum offset of 255 and a maximum length of 15. A 12-bit pointer plus 6-bit length, which assumes a maximum offset size of 4096, implying a 4 kilobyte buffer, and a maximum length of 63. As noted, a flag bit with a value of 0 indicates a pointer. This is followed by a second flag bit giving the size of the pointer, with a 0 indicating an 8-bit pointer, and a 1 indicating a 12-bit pointer. So, in binary, the pointer <3,3> would look like this:

00 00000011 0011

The first two bits are the flag bits, indicating a pointer that is 8 bits long. The next 8 bits are the pointer value, while the last four bits are the length value.
After this comes:
Sp -- which has to be output uncompressed:
the_rain_<3,3>Sp

However, the characters "ain_" have already been sent, so they are encoded with a pointer:

the_rain_<3,3>Sp<9,4>

Notice here, once again at the risk of belaboring the obvious, that the pointers refer to offsets in the uncompressed message. As the decoder receives the compressed data, it uncompresses it, so it has access to the parts of the uncompressed message that the pointers reference. The characters "falls_m" are output uncompressed, but "ain" has been used before in "rain" and "Spain", so once again it is encoded with a pointer:

the_rain_<3,3>Sp<9,4>falls_m<11,3>

Notice that this refers back to the "ain" in "Spain", and not the earlier "rain". This ensures a smaller pointer. The characters "ly" are output uncompressed, but "in_" and "the_" were output earlier, and so they are sent as pointers:

the_rain_<3,3>Sp<9,4>falls_m<11,3>ly_<16,3><34,4>

Finally, the characters "pl" are output uncompressed, followed by another pointer to "ain". Our original message: the_rain_in_Spain_falls_mainly_in_the_plain -- has now been compressed into this form:

the_rain_<3,3>Sp<9,4>falls_m<11,3>ly_<16,3><34,4>pl<15,3>

This gives 23 uncompressed characters at 9 bits apiece, plus six 14-bit pointers, for a total of 291 bits as compared to the uncompressed text of 344 bits. This is not bad compression for such a short message, and of course compression gets better as the buffer fills up, allowing more matches. LZ-77 will typically compress text to a third or less of its original size. The hardest part to implement is the search for matches in the buffer. Implementations use binary trees or hash tables to ensure a fast match. There are a several variations on the LZ-77, the best known being "LZSS", which was published by Storer and Symanski in 1982. The differences between the two are unclear from the sources I have access to. More drastic modifications of LZ-77 include a second level of compression on the output of the LZ-77 coder. "LZH", for example, performs the second level of compression using Huffman coding, and is used in the popular LHA archiver. The "ZIP" algorithm, which is probably the standard archiver and is available in a number of implementations, uses Shannon-Fano coding for the second level of compression

2.10 Win zip

WinZip is a compression utility for Microsoft Windows users, developed by WinZip Computing. It uses PKWARE's PKZIP format, and can also handle a number of other archive formats. It is a commercial product with a free evaluation version.WinZip is also capable of running under Wine on Linux and other UNIX-compatible systems.WinZip began life around the early '90s as a shareware GUI frontend for PKZip. Somewhere around 1996 the creators of WinZip created their own version of PKZip2's deflate algorithm, thus dispensing with the need for PKZip.exe to be present

2.11 G zip

gzip is short for GNU ZIP, a GNU open-source replacement for the Unix compress program.Gzip is based on the deflate algorithm, which is a combination of LZ77 and Huffman coding. 'Deflate' was developed in response to patents that covered LZW and other compression algorithms and limited the usability of 'compress' and other popular archivers.Gzip should not be confused with ZIP, with which it is not compatible. Gzip doesn't archive files, it only compresses them, which is why it is often seen in conjunction with a separate archiving tool (most popularly tar).In order to make it easier to develop software that uses compression, the zlib library was created. It supports the gzip file format and 'deflate' compression. The library is widely used, because of its small size, efficiency and versatility. Both gzip and zlib were written by Jean-Loup Gailly and Mark Adler. Since the late-1990s there has been some movement from gzip to bzip2 which produces considerably smaller files under many circumstances but is also considerably slower.The zlib compressed data format, the 'deflate' algorithm and the Gzip file format were standardised respectively as RFC 1950, RFC 1951 and RFC 1952.The usual file extension for gzipped files is `.gz`. Unix software is often distributed as files ending with `.tar.gz` or `.tgz`, called tarballs. They are files first packaged with tar and then compressed with gzip. They can be decompressed with *gzip -d file.tar.gz* or unpacked with *tar xzf file.tar.gz*. Nowadays more and more software is also distributed as `.tar.bz2` archives because of the advantages of bzip2 compression.

2.12 How Many Words in English Language

The second edition of Oxford English Dictionary contains full entries 171,146 words in current use, and 47,156 obsolete words. To may be added around 9,500 derivatives words include subentries. Over half of those words are nouns, about a quarter adjectives, and about a seventh verbs; the rest is made up of interjection, conjunction, preposition, suffixes etc. These figures take no account of entries with senses different parts of speech.

This suggests that there are, at the very least, a quarter of million distinct English words excluding inflections, and words from technical and regional vocabulary not covered by the OED, or words not yet added to publish dictionary, of which perhaps 20 per cent no longer in current use.

Webster's Third New International Dictionary, Unabridged together with its Addenda Section, 470,000 entries. The Oxford English Dictionary, second Edition, reports that it's include similar number. It is possibly reasonable to take those dictionaries as a rough of the size the general, commonly used English vocabulary.

Chapter-3

Proposed Persistent Memory Requirement Reduction, Required in Text Data Storage Using Look-up Method

To reduce the size of the text segment we will use a word lookup lookup table, which will be acting like a word store and each particular word will be assigned an address value and any particular word will be determined by that value. For that reason, during the time of storing the word, the word will not be stored as a Byte stream based on each of the character's ASCII value. Rather it will be stored as a fixed bits size long bits stream that will form the address value which will reference the address of the text in the word table.

3.1 Word Look up Table

A word lookup table is a special tabular data file containing the text dimension of a word as an attribute of an address, which is used to pop up text to display the possible text data for a field [3, 6]. For our purpose we decide to use a 19-bit word lookup table. A 19-bit word lookup table can contain a number of,

2^19 = 524,288 entries.

That is if we use a 19-bit lookup table then it can index a number of 0.524 millions of different words. Where is in current English language there is an approximation of having a total of 0.470 million of words. It shows that by using a 19-bit lookup table we can easily index any English text [1, 2, 4, 5]. As there is only 0.470 millions of different word in English language that is there are still about (0.524-0.470) = 0.054 Millions of empty entries. That is after deducting some entries for special situation handling we have approximately 54,000 entries empty. Now let us suppose that each year a total of 1,000 new English words are added in English language. That is to fill the empty entries (54,000 / 1,000) = 54 Years time needed. Though in current world each only 50 to 200 words are generally added in the dictionary.

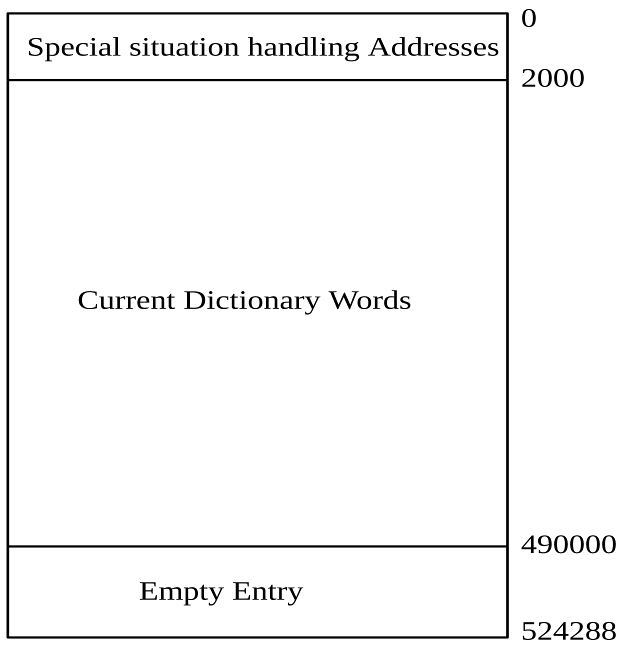

Figure 3.1: Block diagram of the word lookup table

3.2 Word Storing Architecture

In the word lookup table, we supposed that the size of any word is 8 characters long in an average. The storing architecture of the word will be shown below in Figure 1:

Index address	**Word stored in details**	**Ending signal**

19-bit address	8-Byte long word (avg.)	Terminate Byte
19 bit	8 * 8 = 80 bit	8 bit

Total 91 bits

Figure 3.2: The architecture of the stored word in lookup table

According to the proposed architecture in the word lookup table the first 19 bits is used to determine the address of the word in the word lookup table and the next couple of Bytes is for the particular word to be stored in details. Here it is taken 8 Byte as an average. Finally the last portion is a terminate Byte, which is usually a 0 valued Byte. The example of an entry in the word lookup table is shown below in Table 1,

Table 3.1: Words in word lookup table

Index Address	**Word stored in details**	**Ending Signal**
19-bit address	a	00000000
…	…	…
19-bit address	and	00000000
…	…	…
19-bit address	computer	00000000
…	…	…
19-bit address	intelligent	00000000
…	…	…
19-bit address	paper	00000000

The word lookup table has some special situation handling address in the lookup table for many reasons. In the text data there may be any name or a constant word or a spelling mistake. As the proposed compression is a lossless compression this word will needed to be represented too. But those words will not be found in the word lookup table. In that case a termination signal which is certain valued 19-bit address will be placed in the file. This value will tell the compression machine that from now until a word from lookup table is not encountered all the data will be considered as an ASCII character which is 8-bit long. Then after the ASCII values if any lookup table word is encountered it will add a zero valued Byte which will represent the termination of ASCII values and restart of address values. The address pattern of the special situation handling address is shown in the following table, Table 2.

Table 3.2: Special situation handling addresses

Index address	Word stored in details	Ending signal
01---10	Termination of address	00000000
01---01	Single Upper case	00000000
01---10	Multiple Upper case	00000000
01---11	Multile upper casetermination	00000000
01---00	Title case	00000000
01---01	SingletOGGLE cASE	00000000
01---10	Multiple tOGGLE cASE	00000000
01---10	multiple tOGGLE cASE termination	00000000

Different case words is also needed to represent in as the same as in the text segment. That is for different type of cases there should be different representation. By default there will lower case used in general and after punctuating a sentence case will default. But for other cases "Upper case", "Title Case", "Toggle Case" a lookup table address will be added before that word according to the amount of word of that case. But if any word whose case pattern is not supported by any case symbol then it will be considered as a name or special constant word.

One of the main exceptions in the word lookup table is that, in the word lookup table for each punctuation sign there is two different entries. One entry consists of only the punctuation sign and the other consists of the punctuations and with a backspace. This format is shown below in Table 2:

Table 3.3: Entry of different punctuation signs

Index address	Word stored in details	Ending signal
…	…	00000000
01---00	.	00000000
01---01	.+Backspace	00000000
…	…	00000000
11---10	,	00000000
11---11	,+Backspace	00000000

In the case to determine a word from the word lookup table firstly the memory will be searched through linear search algorithm. Let at the first position the following 19 bits will be determined to check the index number. Then the next locations will be determined as a collection 8 bits that is 1 Byte. The Byte stream will be used to form a word where each Byte determines respective alphabets until a 0 valued Byte is encountered. The next words will be determined in the same way.

3.3 Memory Allocation Method

The text segment will be indexed as a form of 19 bit long addresses consecutively. The white space between each respective word is excluded here. The reason is that, in the binary text stream each 19 bit represents a word and after each word a white space will be automatically added. Here are some examples shown in Table 3,4,5,6 where LUT stands for lookup table:-

Table 3.4: Example 1

General	He		is		a		very		good		boy	.	176 bits
LUT	21		21		21		21		21		21	21	147 bits

Table 3.5: Example 2

General	Sometimes		I		need		some		help		too	.	248 bits
LUT	21		21		21		21		21		21	21	147 bits

Table 3.6: Example 3

General	Although		computers		may		have		basic		similarities	,	376 bits
LUT	21		21		21		21		21		21	21	147 bits

Table 3.7: Example 4

General	Several		systematic		tabular		methods		for		machine		reduction		exists	.	512 bits
LUT	21		21		21		21		21		21		21		21	21	189 bits

The word lookup table will be a continuous bit stream like the following pattern. Now as the word lookup table will be a very huge database that's why it is a better solution to arrange the address from some particular locations which will be maintained by a table of index. This table of index will provide faster searching of data and also help to maintain the order of data.

Table 3.8: Primary address indexing table

Index address	**Starting Character**
01---00	a
01---01	b
…	…
11---10	**y**
11---11	z

The new entries that will be added in the word lookup table will added in the empty entries of the starting character blocks. The word lookup table pattern is shown in Figure 2:-

11---01	and	00000000	11---10	andrion	00000000	…….	01---11	daylight	00000000	………
00---01	sun	00000000	00---10	sun-bath	00000000	…….	01---11	train	00000000	………

Figure 3.3: How the word lookup table will be stored

3.4 Memory Space Requirement

Now, to build up a word lookup table of 19-bit and with 91 bit (average) we need a memory space of about,

Space = 2^19 * 91 bits
= 524288 * 109 bits
= 47710208 bits
= 5963776 Bytes
= 5824 Kilo Bytes
= 5.68 Mega Bytes
= 6 Mega Bytes

Here in 19-bit word lookup table needs only 6 MB memory space to generate and store the lookup table for 54 years.

3.5 Data Manipulation Algorithm

For the compression of the text data a fixed algorithm will be carried out over the text. General algorithm for converting uncompressed text to compressed text is shown in the below:

```
Algorithm UnCopmToComp( )
{
Read the word.
Find the starting address of the word block.
Find the address of the word in word lookup table.
Write the address in the file.
}
```

General algorithm for converting compressed text to uncompressed text is shown in the below:

```
Algorithm CopmToUnComp( )
{
Read the address.
Find the starting address of the address block.
Find the word addressed according to the word lookup table.
Write the word in the file.
}
```

3.6 Compatibility

From the compatibility view point it will have the only disadvantage that the compressed file may contain some unknown words if the conversion machine that is handling the compressed data is the earlier version of that conversion machine or else there will be no chance of incompatibility.

Chapter 4

Experimental Result

From the Example 1 we see that the general size of the text is 176 bits where is by using 19-bit word lookup table the size becomes 133 bits and the percentage of reduction is 24.43%. For Example 2, the result is in general condition 248 bits, but in 19-bit Word Lookup table is 133 bits and the percentage of reduction is 46.37%. Then the result for Example 3, is in general is 376 bits, in 19-bit word lookup table is 133 bits and percentage of reduction is 64.62%. Consequently, according to Example 4 the result is in general is 512 bits, in 19-bit word lookup table is 171 bits and percentage of reduction is 66.60%. Here is another example of the experimental data; we copied the following text segment in Example 5, 24 times, to make a large text segment. The experimental data Example 5 is shown below:-

"Although computers may have basic similarities, performance will differ markedly between them, and just the same as it does with cars. The PC contains several processes running at the same time, often at different speeds, so a fair amount of coordination is required to ensure that they don't work against each other. Most performance problems arise from bottlenecks between components that are not necessarily the best for a particular job, but a result of compromise between price and performance. Usually, price wins out and you have to work around the problems this creates. The trick to getting the most out of any machine is to make sure that each component is giving of its best, and then eliminate potential bottlenecks between them. You can get a bottleneck simply by having an old piece of equipment that is not designed to work at modern high speed - a computer is only as fast as its slowest component, but bottlenecks can also be due to badly written software."

We created another experimental data; coping the following text segment in Example 6, 32 times, to make a large text segment. The experimental data Example 6 is shown below:-

"In the current world we have high powerful processors and high capability storages devices not only in the micro computer but also in PDA's. That's why it not difficult to store or to manipulate a file. But it is still difficult to transfer file or data through communication medium. The reason is that the signal capacities of the carriers are not sufficient enough. And this problem is deeply felt in internet communication. In the case of text transfer if we can minimize the text size it will increase the faster portability of the text files. This can be done by indexing the text and by generating a lookup table which will be used to index the text and that will decrease the number of Bytes needed to define a particular text."

The compression result for the Examples 5 and 6 is shown below in Table 4.1,

Table 4.1: Experimental result

In General Situation		
	Example 5	**Example 6**
Words	3984	4192
Characters	19392	19328
Characters with white space	23378	23519
Text Size	Bytes 23378	Bytes 23519
In 21-bit Word Lookup-Table		
Words	3984	4320
Punctuation	361	224
Words with punctuation	4345	4544
Per word text size	bits 19	bits 19
Text size	Bytes 10320	Bytes 10792
Text Size Reduction Status		
General situation	23378	23519
21-bit word lookup-table	10320	10792
Size reduced	13058	12727
Reducing percentage	55.86%	54.11%

We also experimented the text size for ten stories of Leo Tolstoy; and some couple of writings published in local English newspapers. The results for the stories of Tolstoy were 55.91%, 57.32%, 54.76%, 53.01%, 51.55%, 54.72%, 44.07%, 52.73%, 54.98% and 51.83%. And for articles in daily newspaper, the results were 40.24%, 46.82%, 55.64%, 64.36%, 52.75%, 55.41%, 58.16%, 49.53%, 51.96% and 62.97%. That is, finally we got an average of 53.4186% reduction rate.

Comparison with other compression method

In general the all compression methods have there compression rate from 12% to highest 50%. But in our method we have found 53% compression in the starting of the approach. That further improvement of the approach will increase the compression rate.

Comparison with other zip software

In comparison with currently available zip software we found that the following outputs in the case of a same file. The comparison is shown in the table 4.2.

Table 4.2: Camprison between zip software

Compression Type	**Size**
Normal	78.53 KB
According to Proposed Method	36.90 KB
Gzip	47.11 KB
Winzip	49.27 KB

Chapter 5

Conclusion

In this paper we wanted to provide a whole new compression method. As the world is moving towards the goal of proving highest service at a lowest expense, this method of word lookup table will make any text segment able to use lesser memory space but will not decrease its features rather will increase its usability and portability. It will decrease the memory area occupied by text segment in any type of file, which will make a huge amount of memory area free. And also decrease its transfer time through FTP or SMTP too.

Future work

This method may be applied more efficiently if suitable algorithms are applied for determining the address value and doing memory management. We have intention to use the combination of Huffman decoding and LZ-77 combination to decrease the index address memory requirement as well the constant words memory requirements.

References

[1] http://dictionary.reference.com/help/faq/language/h/howmanywords.html

[2] http://www.askoxford.com/asktheexperts/faq/aboutwords/numberwords

[3] http://www.esri.com/library/glossary/i_l.html

[4] http://www.msexchange.org/tutorials/Full-Text-Index-Information-Store.html

[5] http://www.m-w.com/help/faq/total_words.htm

[6] http://www.novell.com/documentation/gw52/lb100001.html

[7] http://en.wikipedia.org/

[8] Debra A. Lelewer, Daniel S. Hirschberg, "Data compression" in ACM Computing Surveys (CSUR).

[9] Janet Fabri, "Automatic storage optimization" in ACM SIGPLAN Notices, Proceedings of the 1979 SIGPLAN symposium on Compiler construction.

[10] Roberto Grossi, Ankur Gupta, Jeffrey Scott Vitter, "High-Order Entropy-Compressed Text Indexes" on Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms.

[11] Bruce Hahn, "A new technique for compression and storage of data" in Communications of the ACM.

[12] Gordon Linoff, Craig Stanfill, "Compression of indexes with full positional information in very large text databases" on Proceedings of the 16th annual international ACM SIGIR conference on Research and development in information retrieval.

[13] Madnick, “Operating System”, McGraw-Hill Publishing Company.

[14] Michael S. Mahoney, "The UNIX System Oral History Project", AT&T Bell Laboratories.

[15] Morgan, Rachel and Henry McGilton, "Introducing UNIX System V", McGraw-Hill Publishing Company.

[16] Frank Rubin, "Experiments in text file compression" on Communications of the ACM.

[17] James Shaw, "Conciseness through aggregation in text generation" on Proceedings of the 33rd conference on Association for Computational Linguistics.

[18] Silberschatz Abraham, Peter Baer Galvin, Gerg Gagne, "Operating System Concept", John Wiley & Sons, Inc.