Lossless Text Compression using Dictionaries

Umesh S. Bhadade
G.H. Raisoni Institute of Engineering & Management
Gat No. 57, Shirsoli Road
Jalgaon (MS) India - 425001

Prof. A.I. Trivedi
Faculty of Technology & Engineering
M.S. University
Vadodara (GJ) India

ABSTRACT

Compression is used just about everywhere. Reduction of both compression ratio and retrieval of data from large collection is important in today's era. We propose a pre-compression technique that can be applied to text files. The output of our technique can be further applied to standard compression techniques available, such as arithmetic coding and BZIP2, which yields in better compression ratio. The algorithm suggested here uses the dynamic dictionary created at run-time and is also suitable for searching the phrases from the compressed file.

General Terms

Compression, Searching, lossless text compression

Keywords

Text Compression, Dynamic dictionary

1. INTRODUCTION

The task of compression consists of two components, an *encoding* algorithm that takes a message and generates a "compressed" representation and a *decoding* algorithm that reconstructs the original message or some approximation of it from the compressed representation. These two components are typically intricately tied together since they both have to understand the shared compressed representation.

The compression algorithms can be classified broadly in two categories viz. *lossless algorithms*, which can reconstruct the original message exactly from the compressed message, and *lossy algorithms*, which can only reconstruct an approximation of the original message. Lossless algorithms are typically used for text, and lossy for images and sound where a little bit of loss in resolution is often undetectable, or at least acceptable. Lossy is used in an abstract sense, however, and does not mean random lost pixels, but instead means loss of a quantity such as a frequency component, or perhaps loss of noise.

It is not possible to compress everything, all compression algorithms must assume that there is some bias on the input messages so that some inputs are more likely than others, *i.e.* there is some unbalanced probability distribution over the possible messages. Most compression algorithms base this "bias" on the structure of the messages – *i.e.*, an assumption that repeated characters are more likely than random characters, or that large white patches occur in "typical" images. Compression is therefore all about probability.

When discussing compression algorithms it is important to make a distinction between two components: the model and the coder. The *model* component somehow captures the probability distribution of the messages by knowing or discovering something about the structure of the input. The *coder* component then takes advantage of the probability biases generated in the model to generate codes. It does this by effectively lengthening low probability messages and shortening high-probability messages. The models in most of the current real-world compression algorithms, however, are not so sophisticated, and use more mundane measures such as repeated patterns in text. Although there are many different ways to design the component of compression algorithms and a huge range of levels of sophistication, the coder components tend to be quite generic. Current algorithms are almost exclusively based on either Huffman or arithmetic codes.

Another question about compression algorithms is how to judge the quality of one versus another. In the case of lossless compression there are several criteria such as, the time to compress, the time to reconstruct, the size of the compressed messages, memory requirements and the generality. In the case of lossy compression the judgment is further complicated since we also have to worry about how good the lossy approximation is. There are typically tradeoffs between the amount of compression, the runtime, and the quality of the reconstruction. Depending on application one might be more important than another and one would want to pick algorithm appropriately.

For most applications, a compression scheme must allow random access to data within the large data files, so that selected documents or records can be retrieved and presented to a user. Only a small class of compression schemes permits such random access and decompression. Moreover, these' schemes only work on collections where symbols - such as words - can be extracted from the data to be compressed. Hence, effective methods are to be developed which allows random access on text from large data files, with better compression ratio and improved decompression time.

There are several methods existing in the area of lossless data compression.

Some of the basic methods are:

- Huffman coding and its variants.
- Arithmetic coding and its variants.
- Dictionary based methods such as Lempel Ziv Welch (LZW)
- Run Length Encoding (RLE).
- Move-to-Front coding (MTF).
- Burrows Wheeler Transform Method (BWT).

The paper is organized in the following way. Section 2 gives an overview of few existing compression techniques. Section 3 gives brief overview of searching pattern in compressed file. Section 4 gives detailed description of the proposed new compression techniques, including dictionary creation, compression and decompression algorithms, searching algorithms. In Section 5 the quick searching algorithm used for searching the text in the compressed file is explained. Section 6 discusses the implementation details of the proposed compression techniques. Section 7 gives the comparison and evaluation of the results of the proposed compression techniques v/s existing compression techniques, followed by, conclusion in section 8.

2. BASIC COMPRESSION TECHNIQUES

2.1 Huffman Coding

Huffman codes[1] work by replacing each alphabet symbol by a variable-length code string. ASCII uses eight bits per symbol in English text, which is wasteful, since certain characters (such as `e') occur far more often than others (such as `q'). Huffman codes compress text by assigning `e' a short code word and `q' a longer one.

Optimal Huffman codes can be constructed using an efficient greedy algorithm. Sort the symbols in increasing order by frequency. We will merge the two least frequently used symbols x and y into a new symbol m, whose frequency is the sum of the frequencies of its two child symbols. By replacing x and y by m, we now have a smaller set of symbols, and we can repeat this operation n-1 times until all symbols have been merged. Each merging operation defines a node in a binary tree, and the left or right choices on the path from root-to-leaf define the bit of the binary code word for each symbol. Although they are widely used, Huffman codes have three primary disadvantages. Two passes are required over the document on encoding, the first to gather statistics and build the coding table and the second to actually encode the document. The coding table is than stored along with the document order to reconstruct it, which eats into your space savings on short documents. Finally, Huffman codes exploit only no uniformity in symbol distribution, while adaptive algorithms can recognize the higher-order redundancy in strings such as 0101010101....

2.2. Arithmetic Coding Method:

For few years the arithmetic coding[5, 6] had replaced Huffman coding. It completely bypasses the idea of replacing an input symbol with a specific code. Instead, it takes a stream of input symbols and replaces it with a single floating-point output number. Longer the message, the more bits are needed in the output number.

Arithmetic coding is especially useful when dealing with alphabets with high probabilities. It is also very useful that output from an arithmetic coding process is a single number less than 1 and greater than or equal to 0. This single number can be uniquely decoded to create the exact stream of symbols that went into its construction. In order to construct the output number, the symbols being encoded have to have a set of probabilities assigned to them.

Once the character probabilities are known, the individual symbols need to be assigned a range along a "probability line", which is nominally 0 to 1. It doesn't matter which characters are assigned to which segment of the range, as long as it is done in the same manner by both the encoder and the decoder.

Each character is assigned the portion of the 0-1 range that corresponds to its probability of appearance. Note also that the character "owns" everything up to, but not including the higher number. So the last letter has the range 0.90 - 0.9999.... and not 1.

The most significant portion of an arithmetic coded message belongs to the first symbol to be encoded. In order for the first character to be decoded properly, the final coded message has to be a number greater than or equal to the range of the first character of the actual stream. To encode this number, keep track of the range that this number could fall in. So after the first character is encoded the algorithm must continue with the next character in actual stream.

After the first character is encoded, the low and the high of the first character now bound the range for the output number. What happens during the rest of the encoding process is that each new symbol to be encoded will further restrict the possible range of the output number. If it was the first number in the message, then low and high ranges values are set directly to those values.

2.3 Lempel-Ziv algorithms (*LZ*)

Lempel-Ziv algorithms [2, 3] including the popular LZW variant [4], compress text by building the coding table on the fly as the document is read. The coding table available for compression changes at each position in the text. A clever protocol between the encoding program and the decoding program ensures that both sides of the channel are always working with the exact same code table, so no information is lost.

Lempel-Ziv algorithms build coding tables of recently-used text strings, which can get arbitrarily long. Thus it can exploit frequently-used syllables, words, and even phrases to build better encodings. Further, since the coding table alters with position, it adapts to local changes in the text distribution, which is important because most documents exhibit significant locality of reference.

The truly amazing thing about the Lempel-Ziv algorithm is how robust it is on different types of files. Even when you know that the text you are compressing comes from a special restricted vocabulary or is all lowercase, it is very difficult to beat Lempel-Ziv by using an application-specific algorithm.

2.4 Bzip2

Bzip2 compresses files using the Burrows-Wheeler block sorting text compression algorithm, and Huffman coding. Compression is generally considerably better than that achieved by more conventional LZ77/LZ78-based compressors, and approaches the performance of the PPM family of statistical compressors.

2.4.1. Borrows Wheeler Transform (BWT)

The Burrows-Wheeler transform [7, 11] is a block-sorting, lossless data compression algorithm that works by applying a reversible transformation to a block of input data. The transform does not perform any compression but modifies the data in a way to make it easy to compress with a secondary algorithm such as “move-to-front” coding and then Huffman, or arithmetic coding. The BWT algorithm achieves compression performance within a few percent of statistical compressors but at speeds comparable to the LZ based algorithms.

If the file size is large then BWT algorithm does not process data in one time slot, but takes blocks of data at a time unit, and process it sequentially. The transformed block contains the same characters, plus a little overhead, as the original block but in a form that is easy to compress by simple algorithms. Same characters in the original block are often grouped together in the transformed block.

The algorithm works by transforming a string S of N characters by forming the N rotations (cyclic shifts) of S, sorting them lexicographically, and extracting the last character of each of the rotations. A string L is then formed from these extracted characters, where the ith character of L is the last character of the ith sorted rotation. The algorithm also computes the index I of the original string S in the sorted list of rotations. With only L and I there is an efficient algorithm to compute the original string S when undoing the transformation for decompression. Burrows and Wheeler explain that much of the time, the algorithm is performing sorts which may be another area where using a parallel sorting algorithm may increase the speed of BWT. They also explain that to achieve good compression, a block size of sufficient value must be chosen, at least 2 kilobytes. Increasing the block size also increases the effectiveness of the algorithm at least up size of several megabytes. If enough memory is present, the block size could also equal the size of the file to be compressed (i.e., one large block) which would be good for compression but reduce the parallelism of the algorithm.

The BWT can be seen as a sequence of three stages: the initial sorting stage which permutes the input text so similar contexts are grouped together, the Move-To-Front stage which converts the local symbol groups into a single global structure, and the final compression stage which takes advantage of the transformed data to produce efficient compressed output [6].

Before the data in BWT is compressed, it is run through the Move-To-Front coder. The choice of MTF coder is important and can affect the compression rate of the BWT algorithm. The order of sorting determines which contexts are close to each other in the output and thus the sort order and ordering of source alphabet can be important. Many people consider the MTF coder to be fixed, but any reversible transformation can be used for that phase. The dependence of BWT on input alphabet encoding is a relatively unique characteristic for general lossless data compression algorithms. Algorithms such as PPM and LZ are based on pattern matching which is independent of source alphabet encoding. Using different methods of ordering for the input alphabet can result in smaller compressed files for some data and larger compressed sizes for others so the ordering must be carefully selected and may not be appropriate for all data.

2.5. Move to Front Technique

The output obtained from the BWT is a series of blocks which are lexicographically arranged. To take advantage of this arrangement, another coding scheme is used called as Move-To-Front technique (MTF)[8]. This is used as a sub-step in several other algorithms including the Burrows-Wheeler algorithm. This is a technique that is ideal for sequences with the property that the occurrence of a character indicates it is more likely to occur immediately afterwards. The sequence of characters is converted to a list of numbers as follows: The list of characters maintained, represent characters by their position in the list. On encoding a character, it is moved to the front of the list. Thus smaller numbers are more likely to occur than larger numbers. The MTF algorithm is not very exciting, but it does a really nice job of compressing streams that have been put through the BWT transform. The output of this technique is then pass to either Huffman Coder or Arithmetic Coding.

3. SEARCHING IN COMPRESSED FILES

With compressed files becoming more commonplace, the problem of how to search them is becoming increasingly important. There are two options to consider when deciding how to approach compressed pattern matching. The first is a `decompress-then-search' approach, where the compressed file is first decompressed, and then a traditional pattern-matching algorithm applied. This approach has the advantage of simplicity, but brings with it tremendous overheads, in terms of both computation time and storage requirements. Firstly, the entire file must be decompressed often a lengthy process, especially when considering files several megabytes in size. Additionally, the decompressed file must be stored somewhere once decompressed, so that pattern matching may occur.

The second alternative is to search the compressed file without decompressing it, or at least only partially decompress it. This approach is known as compressed-domain pattern matching, and offers several enticing advantages. The file is smaller, so a pattern matching algorithm should take less time to search the full text. It also avoids the work that would be needed to completely decompress the file.

The main difficulty in compressed-domain pattern matching[9] is that the compression process may have removed a great deal of the structure of the file. Greater the structure removed, better the compression likely to be achieved. There is therefore a subtly-balanced tension between obtaining good compression and leaving enough `hints' to allow pattern-matching to proceed. It would appear that these two goals are in constant opposition, but in fact compression is very closely related to pattern matching, in that many compression systems use some sort of pattern matching technique to find repetitions in the input, which can be exploited to give better compression. The effect of this is that these patterns are coded in a special manner, which, if suitably represented, may actually aid in pattern matching.

4. PROPOSED NEW COMPRESSION TECHNIQUES

- **Word Based Text Compression Technique using Dynamic Dictionary (Method – A)**
- **Character Based Text Compression Technique using Dynamic Dictionary**
- **Word Based Text Compression Technique using Dynamic Dictionary (Method – B)**

The first method gives better compression ratio, but is not useful for direct searching. The second method proposed here works as pre-compression stage to arithmetic coding, yields better compression ratio than arithmetic coding when used alone, and is also useful for direct searching. The third method proposed here works as pre-compression stage to Bzip2, yields better compression ratio than Bzip2 when used alone, and is also useful for direct searching.

In the next section new compression techniques developed by authors are discussed, which are based on static and dynamic dictionaries.

Static Dictionary: The static dictionary of the ASCII characters is first created from the set of the corpus. The ASCII characters frequency is calculated and the characters are arranged in the descending order. The characters are stored in the dictionary in two-dimensional matrix form. To reduce the length of the code of the character, in one row, number of characters stored must be less than 256. If the length of the code is to keep of 5-bits, then 32 characters will be stored in one row. The novel approach used here in the dictionary is to repeat the 16 characters of highest frequency in each row, and the other characters will fill up remaining 16 characters in the row.

Dynamic Dictionary: The dynamic dictionary is created separately for each file to be compressed. The dynamic dictionary is created for both the words and 2-3-4 character pairs.

In this dictionary also the above novel approach is used to create the dictionary in two-dimensional matrix form, and some of the words or 2-3-4 characters pair will be repeated in each row and the remaining portion of the row is filled by remaining words or 2-3-4 character pairs.

4.1 Word Based Text Compression Technique using Dynamic Dictionary (Method – A).

In this method three different dictionaries are created for words and sub-words (prefix and suffix part of the word). In word dictionary the words, which appear twice or more are included and in sub-word dictionary the prefix or suffix part of the words, which occur only once, but in them the prefix or suffix part occur twice or more are included. For e.g. if word 'coming' and 'going' is appearing only once. In these words the suffix string 'ing' is appearing, therefore the sub-word 'ing' will be added to the suffix sub-word dictionary. In the similar way the prefix words are added to the prefix sub-word dictionary.

Also the dictionary of non-words is also created, which includes words of non-alphabets. For e.g. say after the word 'going' there is full stop and carriage return, then both the symbols full stop and carriage return will be considered as one non-word and will be added to dictionary of non-words. All the words in the dictionary are arranged in the descending order, so that the most probable words are at the starting of the dictionary. Here the most frequent 128 words will be treated in each row, so that probability of occurrence of words in the same row increase i.e. out of 256 words, first 128 words will be the most frequent words of the dictionary.

4.1.1 Compression

In first pass Word Based Dictionary is created for words, sub-words and non-words. In Second pass, the words are scanned from the source file and is searched first in the word dictionary and if found the index value of the corresponding word is stored in the compressed file, else the sub-word dictionary is searched for finding the presence of the prefix or suffix part of the word read from the source file, if found then the index value will be stored in the compressed file, else the word is stored as it is in the compressed file. Similar process is adopted for non-words. The searching of the words and non-words is done alternatively, as in any file after word there will be a non-word and after every non-word, there will be word.

4.1.2 Making of the index value

Whenever the word is found in the dictionary, the index value is converted into two-dimensional value viz. row and column. Here we are considering the two-dimensional matrix of N rows by 256 Columns. For example, if the index value of word is say 356, then the row = 2 and column = 100. If the current index value points to the same row as that of previous, then only the column value i.e. 100 is written in the compressed file, otherwise row value 2 preceding with change in row will be written in the compressed file.

Example of Prefix Searching

Assume the current word to be compressed is 'singing'. Prefix sub-word dictionary will be used to find the occurrence of first few characters of 'singing'. In the prefix sub-word dictionary, the word 'sing' is added because of another word 'singer'. 'sing' of 'singing' will be replaced by the index value of 'sing'

Example of Suffix Searching

Assume the current word to be compressed is 'welcome' Suffix sub-word dictionary will be used to find the occurrence of last few characters of 'welcome'. In the suffix sub-word dictionary, the word 'come' is added because of another word 'become'. 'come' of 'welcome' will be replaced by the index value of 'come'

4.2 Character Based Text Compression Technique using Dynamic Dictionary

This method is similar to the above mentioned character based method, the only difference is that instead of writing 5-bit code, the codes written are in multiples of 8-bits, and instead of limited number of 4-Char, 3-Char, here all possible 4-Char pair and 3-Char pair are considered.

The frequency of all possible 4-Char pair, 3-Char pair and 2-Char pair is computed.

After counting the frequency of all possible pairs, all the pairs are sorted in descending order so that most probable pairs will have index values in the lower range.

4.2.1 Dictionary Creation

Create the dictionary of character pair in the following way:

2-Char Dictionary: Store only first 32 double character pair in the dictionary. As it is well known that normally it requires two byte to store the 2-Characters, so if 16-bit index value is used then compression is not achieved. Hence in this method only 32 most frequent 2-Char pairs are considered and they will be coded as 8-bit.

3-Char Dictionary: For achieving compression, it is wise to store all triple character pair except those whose frequency count > 3. In this dictionary the maximum triple char pair, which can be stored, is 8192 and it will be coded as 16-bit.

4-Char Dictionary: For achieving compression, it is wise to store all quad character pair except those whose frequency count > 2. In this dictionary the maximum quad char pair, which can be stored is 16384 and it will be coded as 16-bit.

4.2.2 Compression

Scan the entire file (read at least 4-Char at a time). Search 4Char pair in the dictionary, If found construct code value and store it in compressed file, else search 3-Char pair in the dictionary, if found construct code value and store it in compressed file, else search 2-Char pair in the dictionary, If found construct code value and store it in compressed file, else store the character as it is in the compressed file.

4.2.3 Construction of code value

2-Char pair

The code is of 8-bit only, because if we use 16-bit code, then we won't get compression as normally it requires 16-bit to store 2-Char. MSB bit of 8-bit code is set to '1', to distinguish it from normal character. Next two bits are kept to '00', to indicate 2-Char pair code. Remaining 5bits are used to store index value of 2-Char pair.

1	**0**	**0**	**5-bit index value of 2Char pair**

3-Char pair

Code is constructed in this way: MSB set to '1' next two bits to '01' for 3Char pair. The range of the code value varies from 40960 to 49151.

1	**0**	**1**	**13-bit index value of 3Char pair**

4-Char pair

Code is constructed in this way: MSB set to '1' next bit is set to '1' for 4Char pair. The range of the code value varies from 49152 to 65535.

As the frequency of 4Char pair is large the index value is of 14-bit i.e. total pairs will be 16384.

1	**1**	**14-bit index value of 4Char pair**

4.2.4 Decompression Technique

Read dictionaries of 4-Char pair, 3-Char pair and 2-Char pair. Read 1 byte from compressed file. Check MSB bit, if 0 then store that byte as it is in the decompressed file. If 1 then check next two bits are 00 or not, if yes the next five bits will be the index value of the double pair dictionary. Store two characters from the double pair dictionary in the decompressed file stored at that index value in the dictionary.

If next two bits are 01 then read another byte to form an index value for triple character pair, and store the triple character in the decompressed file. Else if next bit is 1 then also read another byte to form an index value for quad character pair, and store the quad character in the decompressed file. Repeat the process till all the bytes are read from the compressed file.

1.1.1.1.1.1.1.1   Example

If the byte read is say '01000101' i.e. 65, then in this case the MSB is '0' so store value 65 directly in the compressed file.

If the byte read is say '10000010' i.e.130, then in this case the MSB is '1', check another two bits, i.e. '00', hence the next five bits ('00010') will indicate the index value in the 2-char dictionary.

If the byte read is say '10100000' i.e. 160, then in this case the MSB is '1', another two bits are '01', so read another byte say '00000100' combine both bytes to form 16-bit data '10100000 00000100' the lower 13-bit value is 4, indicating the index value of the 3-char dictionary.

If the byte read is say '11000000' i.e. 1192, then in this case the MSB is '1', another bit is '1', so read another byte say '00001111' combine both bytes to form 16-bit data '11000000 00001111' the lower 14-bit value is 15, indicating the index value of the 4-char dictionary.

This method is used as a preprocessing compression stage to arithmetic coding, which yields a better compression ratio as compared to arithmetic coding when used as alone. As the codes stored in this file are byte boundary, this method is useful for direct searching in the compressed form.

4.3. Word Based Text Compression Technique using Dynamic Dictionary (Method – B).

4.3.1 Dictionary Creation

In this method, instead of character pairs, the whole word is stored in the dictionary of one dimension. The length of the word is not stored; instead separator character '#' is stored in between the words to distinguish it. The word scanned is first searched in the single array, if not found the word is added to the dictionary. The length of the word is checked, if greater than two, then only it is added to the dictionary. If the size of the dictionary goes beyond the limit of 64K, then the dictionary is converted into two-dimensional matrix and the index value is created in the same manner as it was created in the section 4.1.2, explained above for two-dimensional matrix.

4.3.2 Compression

The entire file is scanned word by word. The scanned word is searched in the dictionary. The separator character '#' helps in identifying the boundaries of the words. The searching process goes on counting the number of '#' it encounters till it finds the word to be searched. If found then the index value of that word is stored, else that word is stored in the compressed file as it is. The algorithm of searching the word in the dictionary is given below:

```
Search(char *string) {
     hash = dtrack = 0
     while(dtrack < track ) {
        if(dictionary[dtrack++] == '#') {
           hash++;
          for(i=0;i<len;i++) {
              if(dictionary[dtrack] != string[i])
                break;
              dtrack++;
         }
         if(i==len) {
            if(dictionary[dtrack] == '#')
                return hash;
         }
    }
   return 0;
}
```

4.3.3 Decompression Technique

Read the first byte, if it is normal character then store as it is in the decompressed file. If it is index value of word from the dictionary, then the word is fetched from the dictionary and written to the decompressed file. The algorithm of retrieving the word from the dictionary of an index value and writing the word to the decompressed file is given below:

```
hash = 0;
for(i=0;i<sizeofdict;i++){
    if(dictionary[i]=='#')
        hash++;
    if(hash == read)   {
      while(1){
         i++;
         ch = dictionary[i];
         if(ch == '#') break;
         fprintf(ptr,"%c",ch);
       }
       break;
   }
}
```

5. PATTERN MATCHING IN COMPRESSED FORM

For searching the pattern of the text directly in the compressed form, we are compressing the pattern with the same compression technique, by which the text is compressed and then the compressed pattern can be searched directly in compressed file in a conventional way by using any String-matching algorithm. The different string-matching algorithms are Karp-Rabin Algorithm, Brute Force Algorithm, Knuth-Morris-Pratt Algorithm, Boyer-Moore Algorithm, Quick Search Algorithm
Here the QS algorithm[10] is implemented, as it is better than other algorithms for English text.

The QS algorithm is given below which searches the pattern X from the text Y.

```
int qs(char *x,char *y,int m,long int n) {
    int i,j, bc[ASIZE];
   for(j=0;j<ASIZE;j++)
      bc[j] = m;        // Preprocessing
   for(j=0;j< m ;j++)
      bc[x[j]] = m-j-1;
    i=0;                // Searching
    while (i <= n-m)  {
        j=0;
        while(j < m && x[j] == y[i+j])
              j++;
        if(j>=m)
              return i;
        i+=bc[y[i+m]]+1;
      }
     return 0;
}
```

Algorithm 5.1 Quick Search Algorithm

6. IMPLEMENTATION OF PROPOSED TECHNIQUES

6.1 Word Based Text Compression Technique using Dynamic Dictionary (Method – A)

The method is implemented using VC++6.0 language and the input is tested mainly on the sample files in the corpus. The performance of the algorithm is shown in table 2. A large text file gives better compression than the files with smaller size. During compression process, the words are scanned from the source file and are searched in the dictionaries of words and sub-words and if found the appropriate index values of them are stored in the compressed file. The algorithms for creating dictionaries, compression, and decompression are given below.

1. Read words and non-words from source file
2. Search the word in the word-dictionary, if not exist, then add it to the dictionary of the words
3. Search the non-word in the .non-word dictionary, if not exist then add it to the dictionary of the non-words.
4. Sort the words and non-words in the descending order in words and non-words dictionary.
5. If the word is occurring only once in the source file, then it is removed from the dictionary.
6. The words which are removed are then further scanned and two more dictionaries of prefix and suffix sub-words is created. The example of creating sub-words is explained in 4.1

Algorithm 6.1 Creating dictionaries of words, non-words and sub-words

1. Read word and non-word alternatively from the source file.
2. Search the word from the word dictionary.
3. If found then create the index value of the word and store it in the compressed file.
4. If not found then, further search the sub-word of the same word in the prefix and suffix dictionary, if found then store the index value of that sub-word in the compressed file, else store the character of the words as it is in the compressed file.
5. Similar process is adopted for the non-words also.

Algorithm 6.2 Compression process

1. Read the compressed file byte by byte.
2. Check if the read byte is the normal character or an index value.
3. If normal character, then store it in the decompressed file as it is.
4. If index value, then decode the index value and retrieve the word, non-word or sub-word from the appropriate dictionaries and read those words character by character and store it in the decompressed file.

Algorithm 6.3 Decompression process

6.2 Character Based Text Compression Technique using Dynamic Dictionary

The method is implemented using VC++6.0 language and the input is tested mainly on the sample files in the corpus. The performance of the algorithm is shown in table 3. A large text file gives better compression than the files with smaller size. During compression process, the characters are scanned from the source file and the pair of 4-character, 3-character and 2-characters is stored in the dictionary. The algorithms for creating dictionaries, compression, and decompression are given below.

1. Read 4-characters from the source file at a time.
2. Update the dictionary of 4-character pair, 3-character pair and 2-character pair.
3. Arrange the dictionary of character pairs in descending order.
4. Remove the 4-character pairs whose frequency is less than three.

5. Remove the 3-character pairs whose frequency is less than four.
6. Remove all 2-characters pairs except first 32.

Algorithm 6.4 Creating dictionaries of character pairs.

1. Read 4-characters from the source file.
2. Search those 4-characters from the 4-character pair dictionary, if found store the index value of it in the compressed file.
3. Else search 3-characters from the 3-character pair dictionary, if found store the index value of it the compressed file.
4. Else search 2-characters from the 2-character pair dictionary, if found store the index value of it in the compressed file.
5. Else store the characters as it is in the compressed file.

Algorithm 6.5 Compression process

1. Read the compressed file byte by byte
2. If the byte read is normal character then store as it is in the decompressed file.
3. Else if the index value is read, then check whether the index value belongs to which character pair dictionary.
4. If it belongs to 2-character pair dictionary then read two characters from the dictionary pointed by the index value and store it in the decompressed file.
5. If the index value belongs to 3-character or 4-character pair, then read one more byte from the compressed file, so as to create the appropriate offset in the dictionary. Read the characters from the dictionary and store it in the decompressed file.

Algorithm 6.6 Decompression process

6.3 Word Based Text Compression Technique using Dynamic Dictionary (Method – B)

The method is implemented using VC++ 6.0 language and the input is tested mainly on the sample files in the corpus. The performance of the algorithm is shown in table 3. A large text file gives better compression than the files with smaller size. The words are scanned from the source file and are added to the dictionary in single dimension with '#' as a separator in between. The algorithms for compression and decompression are given below.

1. Read the words from the source file
2. Search the word in the dictionary (explained in section 4.3) if not found add the word in the dictionary with a separator in between.
3. Else store the index value of the word in the compressed file.
4. Store the non-words as it is in the compressed file.

Algorithm 6.7 Compression process

1. Read the compressed file byte by byte
2. If the byte read is normal character then store as it is in the decompressed file.
3. Else construct index value by reading one more byte from the compressed file and retrieve the word from the word dictionary (explained in section 4.3) and write that word character by character in the decompressed file.

Algorithm 6.8 Decompression process

Table 1 shows that the better compression can be achieved if this method is used as a preprocessing stage to Bzip2, WinRar and WinZip.

From the table 2 and table 3, it is seen that we get better compression ratio for large file size greater than 1 MB.

From table 4 it is seen that the time required to search in the compressed file is less than the time required to search in the normal file.

7. RESULTS AND EVALUATION OF PROPOSED TECHNIQUES

7.1 Word Based Text Compression Technique using Dynamic Dictionary (Method – A)

Table 1: Statistics of text files and their evaluated compression ratios.

FileName	Size	Bzip2	WinRar	Only WinZip
bible.txt	4,047,392	845,635	979,584	1,223,507
world192 .txt	2,473,400	489,583	531,597	731,808
book1	768,771	232,651	278,873	321,707
book2	610,856	157,443	181,093	209,336
news	377,109	118,600	126,011	145,456
harry.txt	5,216,411	1,367,284	1,550,125	1,960,193
crist.txt	183,007	57,667	67,524	72,470
devid.txt	1,987,593	543,564	631,605	767,131
great.txt	1,062,030	284,518	338,179	401,469
hunted.txt	48,115	16,447	19,223	19,747
oliver.txt	917,293	252,135	306,558	362,125
seldon.txt	2,506,369	669,947	763,866	936,514
Total : 12	**20,198,346**	**5,035,474**	**5,774,238**	**7,151,463**
FileName	**Preprocessed Size**	**Pre + Bzip2**	**Pre+ WinRar**	**Pre + WinZip**
bible.txt	3,156,598	787,504	949,806	1,144,065
world192.txt	1,671,599	440,005	527,803	694,637
book1	626,145	224,062	286,956	312,534
book2	457,887	152,091	182,496	204,220
news	310,989	110,995	129,658	146,151
harry.txt	4,009,803	1,294,385	1,520,311	1,833,721
crist.txt	148,848	54,162	68,086	72,748
devid.txt	1,565,867	510,216	625,153	723,827
great.txt	833,834	267,119	336,164	382,510
hunted.txt	40,147	15,527	19,424	20,144
oliver.txt	725,100	240,632	302,945	344,329
seldon.txt	1,987,675	631,746	762,876	894,725
Total : 12	**15,534,492**	**4,728,444**	**5,711,678**	**6,773,611**

7.2 Character Based Text Compression Technique using Dynamic Dictionary

Table 2: Statistics of some sample text files and their evaluated compression ratios.

File Name	Size	Only Arithmetic Coding	Character Based Method with Arithmetic Coding method	**Saving %**
Bible	**4077775**	**2233937**	**1959613**	**12.28**
World192	**2473400**	**2233937**	**1415228**	**36.65**
Oliver	**917293**	**526967**	**485087**	**7.95**
Crist	**183007**	**105769**	**104800**	**0.92**
Devid	**1987593**	**1130623**	**1024177**	**9.41**
Harry	**5216411**	**2989652**	**2679334**	**10.38**
Hunted	48115	27499	28812	-4.77
Paper1	53161	33120	33411	-0.87

7.3 Word Based Text Compression Technique using Dynamic Dictionary (Method – B)

Table 3: Statistics of some sample text files and their evaluated compression ratios.

File Name	Org size	Only Bzip	Word Based Method with Bzip	**Saving %**
Bible	**4077775**	**846235**	**804162**	**4.97**
Devid	**1987593**	**543564**	**538105**	**1.00**
Harry	**5216411**	**1367284**	**1311426**	**4.09**
World192	**2473400**	**489583**	**475588**	**2.86**
white-history-554	**1602075**	**436973**	**435486**	**0.34**
Pge0112	**8441343**	**2498170**	**2395251**	**4.12**
fielding-history-243	**1942195**	**520215**	**509729**	**2.02**
kjv10	**4432803**	**992968**	**960833**	**3.24**
oliver.txt	917293	252135	267062	**-5.92**
crist.txt	183007	57667	62078	**-7.65**
great.txt	1062030	284506	290754	**-2.20**
Hunted	48115	16447	18578	**-12.96**
book1	768771	232598	252096	**-8.38**
book2	610856	157443	164774	**-4.66**
twain-tramp-41	857812	258099	276010	**-6.94**

7.4 Pattern Searching (Quick Searching)

Table 5: Searching Time of phrase in normal and compressed files

File Name	String Search	Time Req. From Original file	Time Req. From Compressed file	Saving in Time
Bible	Abundant	0.078 s	0.047 s	0.031 s
	Creature	0.078 s	0.047 s	0.031 s
	punishment	0.078 s	0.047 s	0.031 s
	Establish	0.093 s	0.047 s	0.046 s
Harry	Remember	0.093 s	0.062 s	0.031 s
	amazement	0.094 s	0.079 s	0.015 s
	Parchment	0.093 s	0.078 s	0.015 s
	compartment	0.094 s	0.063 s	0.031 s

8. CONCLUSION

Paper proposes and confirms by verification, three compression techniques (based on dynamic dictionaries), and giving better compression ratio, if used as a pre-compression stage to the Arithmetic Coding and Bzip2 for large collections. Proposed Compression techniques are useful for direct searching in the compressed form.

9. REFERENCES

[1] Huffman D. A., “A method for the construction of minimum-redundancy codes” Proc. Inst. Radio Eng. 40(9):1098−1101, 1952.

[2] Ziv J. and Lempel A., ‘A universal algorithm for sequential data compression,’ IEEE Transactions on Information Theory, 23(3):337−343, 1977.

[3] Ziv J. and Lempel A., ‘Compression of individual sequences via variable-rate coding,’ IEEE Transactions on Information Theory, 24(5):530−536, 1978.

[4] Welch, T.A. ‘A Technique for High-Performance Data Compression.’ IEEE Computer 17, 6(June 1984), pp. 8-19.

[5] Rissanen J. J. and Langdon G. G., Jr., “Arithmetic Coding,” IBM J. Res. Develop. 23, 149-162 (1979).

[6] J.Rissanen J. J., “Arithmetic Coding as Number Representations,” Acta Polyt. Scandinavica Math. 34, 44-51 (December 1979).

[7] Burrows, M. & Wheeler, D. ‘A block-sorting lossless data compression algorithm’, Technical report, Digital Equipment Corporation, 1994.

[8] Bentley J.L., Sleator D.D., Tarjan R.E., and Wei V.K.. A locally adaptive data compression algorithm. Communications of the ACM, Vol. 29, No. 4, April 1986, pp. 320–330.

[9] Amis A. and Benson G., Efficient two-dimensional compressed matching. In J. Storer and M. Cohn, editors, Proceedings of the IEEE Data Compression Conference, pages 279-288, Los Alamitos, CA, Mar. 1992. IEEE Computer Society Press.

[10] Tucker An B. Jr.’The Computer Science and Engineering Handbook’ Second Edition, Chapman & Hall/CRC, 2004

[11] Mark Nelson, “Data Compression with Burrows Wheeler Transform”, Dr. Dobbs Journal, September 1996.