CHAPTER 4

PROPOSED COMPRESSION METHODS
PROPOSED COMPRESSION METHODS

4.0 OUTLINE OF THIS CHAPTER

This chapter describes the character and word based compression methods proposed and investigated by us. Two character based methods and five word based methods are described. These methods are based on dictionaries created statically, semi-dynamically and dynamically. The concept of two-dimensional dictionary is the novel idea used by us in different methods proposed here. The first character method is based on static dictionary, and uses the two-dimensional static dictionary. The method does not give an effective compression ratio by itself, but forms the basis for other methods developed by us. The second character based method is using semi-dynamic dictionary wherein instead of full words and partial words, groups of characters such as 4, 3 and 2 character groups are stored in the dictionary. This method gives improved compression when it is used as pre-compression stage to Arithmetic Coding. The first word based method uses a semi-dynamic dictionary wherein words and partial words are stored. This method gives better compression when used as pre-compression stage to methods such as Bzip2, PPM variants (PPMd and PPMII), and LZMA. The second word based method is using single dimensional semi-dynamic dictionary and the third word based method is using the two-dimensional semi-dynamic dictionary. This method outperforms over other methods when used with Bzip2 and PPMd. The fourth word based method illustrates the dynamic dictionary approach while the fifth one illustrates the use of static dictionary approach. All the methods are giving an improved compression ratio, when they are used as pre-compression stage to methods such as Bzip2, PPMd, PPMII and LZMA. All the proposed methods except the fourth word based method are useful for direct searching the phrases in the compressed file. The comparison of methods is given at the end.

4.1 INTRODUCTION

There are two distinct approaches to text compression. One is to design a “text aware” compressor; the other is to write a text preprocessor / precompressor (filter) which transforms the original input into a representation having greater redundancy for general-
purpose compressors. The first approach of specialized text compressors are potentially more powerful, both from the viewpoint of compression performance and the compression speed at a given compression rate, as there are virtually no restrictions imposed on the implemented ideas, as in the case of precompressor (second approach), one has to take into consideration how the compression takes place in the subsequent methods. Nevertheless, text preprocessing / pre-compressing is more flexible, as the filtered text can be better compressed with most existing (and hopefully future) general-purpose compressors, so with relatively little programming effort various compression speed / decompression speed / compression ratio compromises can be achieved. One of the attractive ways to increase text compression is to replace words with references to a predefined text dictionary. In our thesis, we are focusing on text pre-compression approach using dictionary based methods.

In some of the dictionary based methods phrases consisting of sub-strings are used, whereas in our methods we are using words i.e. group of alphabetic characters, instead of phrases.

Word-based compression methods parse a document into “words” (typically, contiguous alphanumeric characters) and “non-words” (typically, punctuation and white-space characters) between the words. The words and non-words become the symbols to be compressed. There are various ways to compress them. Generally, the most effective approach is to form a zero-order model for words and another for non-words. It is assumed that the text consists of strictly alternating words and non-words (the parsing method needs to ensure this, and so the two models are used alternately. If the models are adaptive, a means of transmitting previously unseen words and non-words is required. Usually, some escape symbol is transmitted, and then the novel word is spelled out character by character. The explicit characters can be compressed using a simple model, typically a zero-order model of the characters.

There are many different ways to break English text into words and the intervening non-words. One scheme is to treat any string of contiguous alphabetic characters as a word and anything else as a non-word. More sophisticated schemes could take into account punctuation that is part of a word, such as apostrophes and hyphens, and even
accommodate some likely sequences, such as a capital letter following a period. This kind of improvement does not have much effect on compression but may make the resulting list of words more useful for indexing purposes in a full-text retrieval system.

One aspect of parsing that deserves attention is the processing of numbers. If digits are treated in the same way as letters, a sequence of digits will be parsed as a word. This can cause problems if a document contains many numbers – such as tables of financial figures. The same situation occurs, and can easily be overlooked, when a large document contains page numbers – with 100,000 pages, the page number will generate 100,000 “words”, each of which occurs only once. Such a host of unique words can have a serious impact on operation: in an adaptive system, each one must be spelled out explicitly, and in static system, each will be stored in the compression model. In both cases, this is grossly inefficient because the frequency distribution of these numbers is quite different from the frequency distribution of normal words for which the system is designed. One solution is to limit the length of numbers to just a few digits. Longer numbers are broken up into shorter ones, with a null punctuation marker in between. The other is to treat these digits as non-words. The later one is adopted in our methods.

4.1.1. Dictionary Models

Dictionary-based compression methods use the principles of replacing sub-strings in a text with a codeword that identifies that sub-string in a dictionary, or codebook. The dictionary contains a list of sub-string and a codeword for each sub-string. This type of substitution is used naturally in everyday life, for example, in the substitution of the number 12 for the word December, or representing “the chord of B minor with seventh added” as Bm7. Unlike symbol based methods, dictionary methods often used fixed codewords rather than explicit probability distributions because reasonable compression can be obtained even if little attention is paid to the coding component.

The simplest dictionary compression methods use small codebooks. For example, in digram coding, selected groups of letters are replaced with codewords. A codebook for the ASCII character set might contain the 128 ASCII characters, as well as 128 common letter pairs. The output codewords are eight bits each, and the presence of the full ASCII
character set in the codebook ensures that any input can be represented. At best, every
group of characters is replaced with a codeword, reducing the input from seven bits per
careracter to four bits per character. At worst, each seven-bit character will be expanded to
eight bits. Furthermore, a straightforward extension caters to files that might contain
some non-ASCII bytes – one codeword is reserved as an escape, to indicate that the next
byte should be interpreted as a single eight-bit character rather than as a codeword for a
group of ASCII characters. Of course, a file consisting of mainly binary data will be
expanded significantly by this approach; this is the inevitable price that must be paid for
use of a static model.

Another natural extension of this system is to put even larger entries in the codebook –
perhaps common words like and and the, or common components of words, such as pre
and tion. Strings like these that appear in the dictionary are sometimes called phrases. A
phrase may sometimes be as short as one or two characters, or it may include several
words. Unfortunately, having a dictionary with a predetermined set of phrases does not
give very good compression because the entries must usually be quite short if input
independence is to be achieved. In fact, the more suitable the dictionary is for one sort of
test, the less suitable it is for others. For example, if this thesis were to be compressed,
then we would do well if the codebook contained phrases like compress, dictionary, and
even arithmetic coding, but such a codebook would be unsuitable for a text on, say,
business management.

One way to avoid the problem of the dictionary being unsuitable for the text at hands is to
use semi-static dictionary scheme, constructing a new codebook for each text that is to be
compressed. However, the overhead of transmitting or storing the dictionary is
significant, and deciding which phrases should be put in the codebook to maximize
compression is a surprisingly difficult problem. In our methods, we decide to keep the
words, instead of phrases, having frequency count greater than 2.

The concept of replacing words with shorter codewords from a given static dictionary has
at least two shortcomings. First, the dictionary must be quite large—at least tens of
thousands words—and is appropriate for a single language only (our experiments
described in this thesis concern English text only). Second, no “higher level”, e.g., related to grammar, correlations are implicitly taken into account. In spite of those drawbacks, such an approach to text compression turns out to be an attractive one, and has not been given as much attention as it deserves. The benefits of dictionary-based text compression schemes are the ease of producing the dictionary (assuming enough training text in a given language), clarity of ideas, high processing speed, cooperation with a wide range of existing compressors, and—last but not least—competitive compression ratios.

**Why Transformation is beneficial**

There are three considerations that lead us to our transform algorithm. First, we gathered data of word frequency and length of words information from our collected corpora. It is clear that almost more than 60% of the words in English text have the lengths greater than three and more than 80% of the words in English text have the lengths greater than two [14]. There exists a list of the 1000 most frequently used words in the English language. The second consideration is that the transformed output should be compressible to the backend compression algorithm. In other words, the transformed intermediate output should maintain some of the original context information as well as provide some kind of “artificial” but strong context. The reason behind this is that we choose BWT and PPM algorithms as our backend compression tools. Both of them predict symbols based on context information.

Finally, the transformed code words can be treated as the offset of words in the transform dictionary. Thus, in the transform decoding phase we can directly search the word with $O(1)$ time complexity in the dictionary. Based on this consideration, we use a continuously addressed dictionary in our algorithm.

**4.1.2. Related Work for Preprocessing Texts**

The preprocessing of textual data is a subject of many publications. In some articles, the treatment of textual data is embedded within the compression scheme itself but could easily be separated into two independent parts: a preprocessing algorithm and a standard compression algorithm, which are processed sequentially one after the other.
Bentley et al. [60] describe a word based compression scheme, where words are replaced by an index into an MTF list. The dictionary of the words is transmitted implicitly by transmitting the word during its first occurrence. This scheme can be divided into a parsing preprocessing part and a standard MTF ranking scheme. A word based variation of the PPM scheme is presented by Moffat [70]. He uses order-0, order-1 and order-2 word models to achieve better compression than the MTF scheme from Bentley et al. Similar schemes, which differentiate between alphanumeric strings and punctuation strings, and which also use an implicit dictionary, are presented by Horspool and Cormack [71]. Again, these schemes can be divided into a parsing part and a coding part using Huffman codes.

Teahan and Cleary describe several methods for enlarging the alphabet of the textual data [72]. Besides the replacement of common bigrams by a one symbol token, they propose methods for encoding special forms of bigrams called digrams (two letters representing a single sound as *ea* in "bread" or *ng* in "sing"). The replacements are processed using a fixed set of the frequently used bigrams in the English language, which makes this attempt language dependent. Teahan and Cleary [73] describe a word based compression scheme where the word dictionary is adaptively built from the already processed input data. This can also be achieved by a preprocessing stage if the words are replaced by corresponding tokens. Teahan presents a further comparison between two different word based compression schemes in his PhD thesis [74]. The first scheme uses function words, which include articles, prepositions, pronouns, numbers, conjunctions, auxiliary verbs and certain irregular forms. The second scheme uses the most frequently used words in the English language. Both schemes require external dictionaries and are language dependent.

A special case of word encoding is the star encoding method from Kruse and Mukherjee [24]. This method replaces words by a symbols sequence that mostly consist of repetitions of the single symbol "*". This requires the use of an external dictionary that must be known by the receiver as well as the sender. Inside the dictionary, the words are first sorted by their length and second by their frequency in the English language using information obtained from Horspool and Cormack [71]. All sorted words of the same
length are then encoded by sequences "*…*", "A*…*", …, "Z*…*", "a*…*", …, "z*…*", "*A*…*", … where the length of the encoded sequence is equal to the length of the word being encoded. The requirement of an external dictionary makes this method again language dependent.

Preprocessing methods, specialized for a specific compression scheme, are presented by Chapin and Tate [75] and later by Chapin [76]. They describe several methods for alphabet reordering prior to using the BWCA in order to place letters with similar contexts close to one another. Since the Burrows-Wheeler transformation (BWT) is a permutation of the input symbols based on a lexicographic sorting of the suffices, this reordering places areas of similar contexts at the BWT output stage closer together, and these can be exploited by the latter stages of the BWCA. The paper compares several heuristic and computed reorderings where the heuristic approaches always achieve a better result on text files than the computed approaches. Balkenhol and Shtarkov use a very similar heuristic alphabet reordering for preprocessing with BWCA [77]. A different alphabet reordering for BWCA is used in the paper from Kruse and Mukherjee [78]. It also describes a bigram encoding method and a word encoding method which is based on their star encoding.

Grabowski proposes several text preprocessing methods in his publication [79], which focuses on improvements for BWCA but some techniques can also be used for other compression schemes. Besides the already mentioned techniques like alphabet reordering, bigram-, trigram- and quadgram replacement, Grabowski suggests three new algorithms.

The first one is capital conversion. An escape symbol and the corresponding lower letter replace capital letters at the beginning of a word. If the second letter of the word is capitalized too, the replacement is omitted. This technique increases context dependencies and similarities between words, which can be exploited by standard compression schemes. The second algorithm is space stuffing, where a space symbol is placed at the beginning of each line in order to change the context that follows the end of line symbol (EOL) to one space instead of various symbols. The last algorithm is EOL
coding, which replaces EOL symbols by space symbols and separately encodes the former EOL positions, which is represented by the number of blanks since the previous EOL symbol. These numbers are encoded either within the symbol stream itself or in a separate data stream. Grabowski suggests using either space stuffing or EOL coding for preprocessing text files, but because of unstable side effects, he decides to omit EOL coding in his comparisons.

Franceschini et al. extend the star encoding method by using different schemes for the indices into the dictionary [80], called Length-Preserving Transform (LPT), Reverse Length-Preserving Transform (RLPT) and Shortened-Context Length-Preserving Transform (SCLPT). All of these require an external dictionary and are language dependent. A further improvement of the star encoding method, presented by Awan et al. [81], is called Length Index Preserving Transform (LIPT). LIPT encodes a word as a string that can be interpreted as an index into a dictionary. The string consists of three parts: a single symbol '*', a symbol between 'a' and 'z', and a sequence of symbol from the set 'a'…'z', 'A'…'Z'. The second part of the string, the single symbol, represents the length \( l \) of the word, where 'a' stands for length 1 and 'z' for length 26. The third part is the encoded index inside the set of words with length \( l \). They are encoded as a number representation of base 52 decremented by 1, where 'a' represents 0, …, 'z' represents 25, 'A' represents 26, …, and 'Z' represents 51. An empty substring represents the number 0. Therefore, a word of length 3 with index 0 is encoded as "*c", a word of length 3 with index 1 as "*ca", a word of length 3 with index 27 as "*cA" and so on.

Isal and Moffat present different text preprocessing schemes for bigrams and words [82] using internal and external dictionaries. In their paper, tokens are used with values above 255, so they can be used together with normal symbols, as the compression scheme needs to handle alphabets with more than 8 bits. For text files, the word based schemes with internal dictionaries give the highest compression gain. Later Isal et al. combine the word preprocessing scheme with different global structure transformations and entropy coding schemes [83]. Because of the use of an internal dictionary, where each word is spelt out the first time it occurred, the schemes of Isal and Moffat are all language independent.
Teahan and Harper propose a switching algorithm for combining both dynamic and static PPM models that also involves an initial text preprocessing step [84]. In this step that occurs prior to the encoding step, the text is essentially marked up by additional switch symbols to indicate when the compression algorithm should switch to another model. A greedy search algorithm which minimizes the overall code length of the encoded stream (of both the original symbols and additional switch symbols) is used to determine the positions of the markup symbols. This scheme is only relevant to context based schemes such as PPM, and it requires a modification of the subsequent PPM compression scheme.

In all the above methods, the dictionary is considered as a single dimension. We propose an alternative approach here to develop a reversible transformation that can be applied to a source text that improves existing algorithm’s ability to compress with two dimension dictionary. The basic idea behind our approach is to encode every word in the input text file, whose length is greater than 2, as a word in our transformed static/semi-dynamic/dynamic dictionary. These transformed words give shorter length for the input words and also retain some context and redundancy. Thus we achieve some compression at the preprocessing stage as well as retain enough context and redundancy for the compression algorithms to give better results.

Our main focus is to develop a method based on words replacement, which can be used as pre-compression stage to several standard compression methods such Bzip2, PPMd, PPMII and LZMA. All these methods are explained in chapter 2 in detail. This pre-compressed file is then given as an input to existing methods which yields in better compression ratio. The experimental results are given in chapter 6. It has been found that the compression ratio is being improved comparatively by 2.89% in case of Bzip2, 2.56% in case of PPMd, 3.68% in case of PPMII and 1.26 % in case of LZMA.

4.2. IDEA OF OUR METHOD

The main objective is to reduce the total number of possible byte values used in a text file. The idea used in our methods is to use two-dimension dictionary instead of using one-dimension dictionary. Consider for example, if there are 16K words in the dictionary then every individual word will require 14-bits ($2^{14} = 16K$) for encoding it if one
dimension dictionary is used. But if a two-dimension matrix is used then it is possible to encode the individual word in 8-bits only. Thus there is a saving of 6-bits per word. How this can be achieved is explained here.

If all the 16K words are stored in one dimension (i.e. single dimension array), then the dictionary will look like

\[
\text{word0, word1, word2, \ldots, word16381, word16382, word16383}
\]

But if the 16K words are stored in the two-dimension (row X column) with few most probable words in each row, then the dictionary will look like

\[
\begin{array}{cccccc}
\text{Col 0} & \text{Col1} & \text{Col 62} & \text{Col63} & \text{Col64} & \text{Col126} \\
\hline
\text{Row 0} & \text{word0, word1, \ldots, word62}, \text{word63, word64, \ldots \ldots \ldots \ldots}, \text{word126} \\
\text{Row 1} & \text{word0, word1, \ldots, word62}, \text{word127, word128, \ldots \ldots \ldots \ldots}, \text{word190} \\
\text{Row 254} & \text{word0, word1, \ldots, word62}, \text{word16319, word16320, \ldots \ldots \ldots}, \text{word16382} \\
\text{Row 255} & \text{word0, word1, \ldots, word62}, \text{word16383, word16384, \ldots \ldots \ldots}, \text{word16446}
\end{array}
\]

Figure 4.1 Structure of two-dimension dictionary

Thus the above dictionary is of 256 * 127 where number of rows are 256 and number of columns are 127. Here even though the column number can be encoded in 7-bits still we are using 8-bits, 1-extra bit to indicate that the code is from the dictionary. This extra bit will always be kept to 1. Normally in text files, the ASCII character are having code value in between 0 to 127, and they are coded in 8-bits instead of 7-bits, there most significant bit is always 0. To take advantage of this, our coding methods use this extra bit to differentiate between the normal ASCII character and the code of column number. Hence instead of 256 columns we are taking only 127 columns, one less than 128, because the 128th column code will be used as an escape symbol for indicating change in row number.
The idea behind using two-dimensional dictionary is to code the dictionary with the row number and column number. The most frequent words are stored in each row along with some other unique words, therefore the probability of finding the consecutive words in same row increases and we will be able to code the word with 8-bit only without storing the row number, because row number is same and hence will not be stored. This assumption will be taken into consideration by decoder while decompressing the file. We will need to specify the row number only when two consecutive words are not found in the same row. In this case, the escape symbol is to be stored to indicate the change in row number and then followed by the row number in which the word is found, along with the column number. Thus compression is achieved when the consecutive word are found in the same row, because only 8-bit code is needed instead of 14-bit code.

The total number of possible byte values is reduced to 128 only, wherein if the single dimension dictionary is used then the possible combination will be 16384. Our objective of reducing the possible number of bytes is thus achieved by using two dimension dictionary. Experimental results show that two-dimension method works better than single dimension method.

The different methods proposed here, are using static dictionary, semi-dynamic dictionary, and dynamic dictionary.

4.3. PROPOSED TEXT COMPRESSION METHODS

4.3.1. Character Based Text Compression Method using Static Dictionary (CBTC-A)

We had tried here to reduce the number of bits assigned to a normal ASCII character. Ordinary text files, at least English ones, consist solely of ASCII symbols not exceeding 127 in total. Therefore, an ASCII character requires 7-bits to encode it, but instead of 7-bit, in our method we had assigned only 5-bits to ASCII character thereby restricting the number of characters to 32. Now question is how to assign codes to 128 different ASCII characters with just 32 codes. The solution which we have found is to assign same code to multiple ASCII characters in such a way that whenever they will be decoded, we will
exactly come to know the original ASCII character. The idea is to use a two-dimension array as explained in section 4.2., wherein we will store 32 characters in each row. To accommodate all 128 characters we will require 4 rows, but instead of storing all 32 characters in a single row, we decided to repeat some characters in each row for getting effective compression. Along with single characters, we had kept one row each for 4-characters group, 3-characters group and 2-characters group in the dictionary to improve compression. These character groups will be kept in separate rows. 3 different escape symbols will be required to differentiate between the character and character groups. The structure of dictionary is shown in Figure 4.2.

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C12</th>
<th>C13</th>
<th>C14</th>
<th>C27</th>
</tr>
</thead>
<tbody>
<tr>
<td>↓</td>
<td>↓</td>
<td>.</td>
<td>.</td>
<td>↓</td>
<td>↓</td>
</tr>
<tr>
<td>R0</td>
<td>ch0, ch1, . . . , ch12, ch13, ch14 . . , ch27</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>ch0, ch1, . . . , ch12, ch28, ch29, . . . , ch42</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>ch0, ch1, . . . , ch12, ch43, ch44, . . . , ch57</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td>ch0, ch1, . . . , ch12, ch58, ch59, . . . , ch72</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td>ch0, ch1, . . . , ch12, ch73, ch74, . . . , ch87</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td>ch0, ch1, . . . , ch12, ch88, ch89, . . . , ch102</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td>ch0, ch1, . . . , ch12, ch103, ch104, . . . , ch117</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td>ch0, ch1, . . . , ch12, ch118, ch119, . . . , ch132</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.2 Structure of two dimension character dictionary.

The idea used here is that a character will be encoded by 5-bits only i.e. only column number is stored in the compressed file. From the figure 4.2 it can be easily seen that characters such as ch13, ch28, . . . , ch103 and ch118 are having same column number i.e. they will be encode by same code. Thus the same code is allotted to different characters, but with different row numbers. Let us consider an example to elaborate this idea. If we are having a sequence of characters in this way

```
ch1 ch3 ch28 ch0 ch27 ch0 ch104
```

Then to encode this sequence we will follow the procedure as given below:
Initially we will assume row number to be 0. To first encode ch1, we see that ch1 is found in row 0 at position 2 (i.e. at offset of 1). Therefore we will encode it in 5-bits as ‘00001’. The next character is ch3, it is found in row 0 at position 4 (i.e. at offset of 3), and so we will encode it in 5-bits as ‘00011’. Here we had seen that two consecutive characters ch1 and ch3 are found in the same row, so we had encoded them with column numbers only. Next character to encode is ch28, it is found in row 1 which is different from previous row number, and hence we have to now encode row number also. First an escape symbol will be stored to indicate a change in row and then new row number will be stored followed by column number of ch28 i.e. 14. The coding sequence for encoding ch28 will be ‘11111’, ‘00001’, ‘01110’. In this case instead of compression, expansion has occurred i.e. 15-bits are required to encode a single character. But this won’t happen always. Next character to encode is ch0, which is present at position 0 in every row, so this time only column number is stored i.e. ‘00000’. The searching of the character in the dictionary will start from the same row in which previous character was found and we had store most probable 13 characters in every row. Therefore the probability of getting the characters in the same row increases, thereby achieving compression. Thus we are succeeded in encoding the characters in 5-bits instead of 7-bits i.e. we had reduce the number of symbols from 128 to 32.

In order to further improve compression, the dictionary will also contains 256 most probable 2-character, 3-character and 4-character groups from the set of corpus. Before searching the single characters in the dictionary first the characters will be searched in 4-character, 3-character and 2-character groups respectively. If it is not found, then single character dictionary will be searched. If a 4-character group is found then to encode it will require 5-bits escape symbol and 8-bit code to indicate the position of 4-character group, i.e. 13-bits will be required, a saving of 15-bits (4 characters will require 28-bits to save it normally). Thus a saving of 9-bits is achieved in case of 3-character group and that of 1-bit in case of 2-character groups which is negligible but yet helps in compression.

The character groups such as ‘tion’, ‘ing’, ‘th’, and many more normally appears in every text files. This property of text file is taken into consideration for creating the dictionary.
of 4-characters, 3-characters and 2-characters dictionary. Also the frequency of some characters is very high as compared to frequency of others, for e.g., the frequency of ‘e’ is much more than frequency of ‘z’. The same static dictionary will be used both by encoder and decoder, therefore the overhead which occurs in case of semi-dynamic dictionary is reduced and the process of decompression will be fast enough.

**Dictionary creation**

At first, the files are selected from the corpus. Every file is scanned and the frequency of each character is counted. The characters are then arranged in descending order with respect to frequency counts. Thus in general for plain ASCII text files we will get maximum 128 characters with different frequency counts. The characters are then divided into rows and columns as explained in figure 4.2. The dictionary of single character is shown in Figure 4.3 (Source file – bible.txt)

```
be tha ons ir d l u m , w y c g b p v . k A l ;
be tha ons ir d l u f L O D T R G J S B ? H M E j
be tha ons ir d l u f W F ' z N P C x q Z Y K ! U
be tha ons ir d l u f ( ) V - Q @ " X # $ [ ] ^
be tha ons ir d l u _ ` % & * + / 0 1 2 3 4 5 6
be tha ons ir d l u 7 8 9 <= > { | } ~
```

**Figure 4.3 First six rows of single character**

In the Figure 4.3, it is seen that *be tha ons ir d l u* are repeated in each row. This repetition of single characters helps in achieving compression because their probability is more as compared to other characters and hence the probability of getting characters in the same row increased.

Then the most probable four character groups are found from the same corpus. The first 256 four character groups are stored in the separate row as shown below in Figure 4.4. Whenever the 4-character groups match occurs, the index position of the 4-character groups is stored in the compressed file. For e.g. if text ‘agai’ is in the source file, the 4-character groups ‘agai’ is at 2nd position in the column, therefore 2 will be stored in the compressed file along with special symbol ‘11110’. The above process is repeated for 3-characters groups and 2- characters groups. The dictionary for 3-character group and 2-character group is shown in figure 4.5 and figure 4.6.
In order to encode the dictionary symbols, the following strategy is used. To encode four character groups, the column position will be preceded by a unique symbol ‘11110’. For example to encode the four character groups ‘athe’, the code will be ‘11110’ and ‘00000000’ (8-bit code), where unique code ‘11110’ indicates that following code relates to column position of four character group. Similarly, to encode two character groups, the column position will be preceded by a special symbol ‘11100’. For example to encode the two character groups ‘to’, the code will be ‘11100’ and ‘000’, where ‘11100’ indicates that the next code means for column position of two character groups.

Before encoding starts, initially the row number is assumed to be zero both for encoding and for decoding also. To encode, the single character, the character is first searched...
among the rows. After a match is found, it is first checked whether the new row number matches with the row number in which previous character was found, if previous and new row numbers are equal then only the column position is stored and if the row number differs an escape symbol ‘11111’ is generated to indicate the change in row and then new row number is stored followed by column position. Compression is achieved when the groups of four, characters, three characters, two characters is found, and also when the single characters are found in the same row.

**Decompression**

The decompression process is very simple and fast. The same dictionary is used for decompression. The row number is assumed to be first by default as in the case of compression. First the code is read and then compared with special symbol for 4-character, 3-character groups or 2-character groups; if it is then the next code read is the column position for that groups. The appropriate character groups from the dictionary is then read and stored in the uncompressed file. If the special symbol indicates change in row then the next code is treated as row number followed by column code. The character is thus retrieved from the appropriate row and column from the dictionary and stored in the uncompressed file. Thus decompression process is very fast and the only overhead, which it requires, is the dictionary, the size of which is negligible as compared to large files.

**Searching**

To search a phrase of words (P) in the compressed file directly, first we have to compress the P using the same method explained above, and then search the compressed pattern directly in the compressed file without decompressing it. The standard searching algorithms explained in chapter 3 can be used directly to search the compressed pattern in the compressed file. Thus the searching pattern in the compressed file will be faster as there is no need to decompress the original file and then perform a search operation. Thus we can say that number of comparison to be made for searching in compressed file as compared to normal file will be less enough, thereby saving the time for searching.
This method if used as pre-compression stage to other standards methods such Bzip2, PPM, PPMII and LZMA does not give improved results because there is no redundancy left in the pre-compressed file. In this method 5-bit coding is used and normally the text compressor such as Bzip2, PPM, PPMII and LZMA works on byte boundary. Therefore, normally when this method is used as pre-compression stage then it expands instead of compressing. Hence we drop the idea of using the 5-bit code mechanism for a single character instead we proposed another method in which instead of 5-bit coding, an 8-bit multiple coding is used to encode the words and partial words. This method is explained in section 4.3.2.

4.3.2. Character Based Text Compression Method Using Semi Dynamic Dictionary (CBTC-B)

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 group, 3-Char group, here all possible 4-Char groups and 3-Char groups are considered. The frequency of all possible 4-Char groups, 3-Char groups and 2-Char groups is computed. After counting the frequency of all possible groups, all the groups are sorted in descending order so that most probable groups will have index values in the lower range.

Dictionary Creation

Create the dictionary of character groups in the following way:

2-Character Dictionary: Store only first 32 double character groups in the dictionary. As in normal case to store the 2Characters we require 2 bytes, so if we use index value of 16-bit, then we won’t get compression. Hence in our method we decided to use only 32 most frequent 2Char groups and it will be coded as 8-bit, as explained later in this section.

3-Character Dictionary: For achieving compression, it is wise to store all triple character groups having frequency count > 3. In this dictionary the maximum triple character group, which we can store, is 8192 and it will be coded as 16-bit.
4-Character Dictionary: For achieving compression, it is wise to store all quad character groups having frequency count > 2. In this dictionary the maximum quad char group, which we can store is 16384 and it will be coded as 16-bit.

Compression

Scan the entire file (read at least 4Char at a time). Search 4Char group in the dictionary, If found construct code value and store it in compressed file, else search 3Char group in the dictionary, if found construct code value and store it in compressed file, else search 2Char group 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. Thus certain context of redundancy is provided by storing the single character as it is in the compressed file, for achieving the improved compression ratio when the compressed output of this method is applied to standard method such as Arithmetic Coding. The Arithmetic Coding has been explained in detail in chapter 2. The experimental results are given in chapter 6.

Construction of code value

2-Character group
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 characters. MSB bit of 8-bit code is set to ‘1’, to distinguish it from normal ASCII character. Next two bits are kept to ’00’, to indicate 2Char group code. Remaining 5-bits are used to store index value of 2Char group. Since only 5-bits are used to indicate the index value, 32 – 2Char group can be stored in the dictionary.

1 0 0 5-bit index value of 2Char group
3-Character group

Code is constructed in this way: MSB set to ‘1’, to distinguish it from normal ASCII character. Next two bits to ‘01’ to indicate 3Char group code. The range of the code value varies from 40960 to 49151 i.e. we can store 8192 – 3Char groups in the dictionary.

| 1 | 0 | 1 | 13-bit index value of 3Char group |

4-Character group

Code is constructed in this way: MSB set to ‘1’, to distinguish it from normal ASCII character. Next bit is set to ‘1’ to indicate 4Char group code. The range of the code value varies from 49152 to 65535 i.e. we can store 16384 – 4Char groups in the dictionary.

| 1 | 1 | 14-bit index value of 4Char group |

Decompression

Read dictionaries of double character group, triple character group and quad character group. 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 group dictionary. Store two characters from the double character group 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 group. Store three character from the triple character group dictionary in the decompressed file stored at that index value in the dictionary.

Else if next bit is 1, then read another byte to form an index value for quad character group. Store four character from the quadruple group dictionary in the decompressed file stored at that index value in the dictionary.

Repeat the process till all the bytes are read from the compressed file.
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 decompressed 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 double 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 ‘0000100’ combine both bytes to form 16-bit data ‘10100000 0000100’ the lower 13-bit value is 4, indicating the index value of the triple 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 ‘0001111’ combine both bytes to form 16-bit data ‘11000000 0001111’ the lower 14-bit value is 15, indicating the index value of the quad char dictionary.

This method is used as a precompression 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.

4.3.3. Word Based Text Compression Method Using Semi Dynamic Dictionary (WBTC-A)

The algorithm is based on the idea that most of the words repeat in text. The repetition arises from the structure of the natural language. This is similar to LZW compression where compression is based on the assumption that repetitions of sequences of characters occurs in text. [85,86].

The dictionary of the WBTC-A consists of words and non-words. Horspool and Cormack [71] implemented the word based LZW algorithms using only the single pass through the text, whereas we are implementing in two pass.
Definition of words and non-words
A word is defined as maximal string of alphabetic characters (letters) and non-word is defined as maximal string of other characters (punctuations, spaces and digits). For example sentence

\textit{In\textasciitilde the\textasciitilde beginning\textasciitilde God\textasciitilde created\textasciitilde the\textasciitilde heaven\textasciitilde and\textasciitilde the\textasciitilde earth.}

can be divided into word, non-word sequence: “In”, “\textasciitilde”, “the”, “\textasciitilde”, “beginning”, “\textasciitilde”, “God”, “\textasciitilde”, “God”, “\textasciitilde”, “created”, “\textasciitilde”, “the”, “\textasciitilde”, “heaven”, “\textasciitilde”, “and”, “\textasciitilde”, “the”, “\textasciitilde”, “earth”, “\textasciitilde”, “.” (where \textasciitilde represents space). It is clear that words and non-words from input strictly alternate. The alternating of words and non-words is important piece of information. With this knowledge kind of next word or non-word can be predicted.

When using two passes variant it is necessary to store the dictionary of words and non-words together with the compressed text.

The file to be compressed is scan first to accumulate the statistics of words to form four dictionaries. The first dictionary is for storing words with frequency greater than 1. The second dictionary is for storing the prefix part of the words, which occurs only once, but then in those words some part of word is appearing twice or more. Third for storing the suffix part of the words, which occurs only once, but then in those words some part of word is appearing twice or more. The fourth dictionary is for storing the non-words.

Let us say that word ‘\textit{coming}’ and ‘\textit{going}’ is appearing only once in the source file. In both of the words the suffix string ‘\textit{ing}’ is appearing, therefore the sub-word ‘\textit{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 ‘\textit{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.

After creating all four dictionaries, the words in the dictionaries are arranged in descending order, so that the most probable words will appear in the start of the dictionary. The same idea of creating two dimensional arrays as explained in 4.3.1 is used here.
**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 of the word 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.

**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**

Let us 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 replace by the index value of ‘sing’

**Example of Suffix Searching**

Let us 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 replace by the index value of ‘come’.
In the remaining methods developed and discussed, there is variation in the creation of
dictionary and encoding the words in the dictionary.

**Decompression**

In decompression, the bytes are read from the compressed one by one. If the byte is seem
to be a normal ASCII character then it is stored as it is in the decompressed file. Else the
code is checked for the word, prefix word or suffix word and accordingly the dictionary
is read and the corresponding word is written in the decompressed file. As in the case of
compression it is assumed that words and non-words are alternate, the same assumption
is done while decompression is in progress.

4.3.4. Word Based Text Compression Method Using Semi Dynamic
Dictionary (WBTC-B).

This method is developed only for comparison purpose, to show the effect of two-
dimension dictionary over one-dimension dictionary and the experimental results given in
chapter 6, shows that the compression ratio is improved when two-dimension dictionary
is used instead of one-dimension.
In this method, the dictionary is created of words in single dimension array. The words
are separated by symbol ‘#’ in the dictionary. In this method, simultaneously the
dictionary is created and the file is compressed. This method is simply introduced here to
compare it with other methods proposed by us, which is using the two-dimensional
dictionary. This method is also used as pre-compression stage to standard methods such
Bzip2, PPM giving better result as compared to Bzip2, PPM, when used alone.

**Dictionary Creation**

In this method, instead of character groups, 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. For comparison purpose, the numbers of words kept in the dictionary are restricted to 64K only. The dictionary created will be integrated in the compressed file.

**Compression**

The compression is done in single pass. 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 founds the word to be searched. If found then the index value of that word is stored, else that word is added to the dictionary and then the corresponding index value is stored in the compressed file. Thus the dictionary consists of all the words appearing in the file irrespective of its frequency counts. In the previous methods the words having frequency count greater than 2 were included in the dictionary, but here even if the word occurs once, still it is added to the dictionary, thereby sacrificing the compression. The time required will be less as the compression is done in single pass as compared to two pass in previous method.

**Decompression**

The dictionary of the words is first read from the compressed file. The decompression process is very simple and fast. The compressed file is read byte by byte, if the read byte 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.

**4.3.5 Word Based Text Compression Method using Two-Dimension Semi-Dynamic Dictionary (WBTC-C)**

In WBTC-B method the word stored in dictionary was encoded with 16-bit value. In this method we are reducing the length from 16-bit to 8-bit by converting the dictionary from one dimension to two dimensions. The number of words kept in each row is restricted to
128 only. If we are using only 8-bit code, then the MSB is used to differentiate between the normal ASCII character and encoded value of the words. Therefore, only 7-bit remains to point to the word in the dictionary, hence \(2^7 - 1\) i.e. 127 words are kept in one row. Out of these 127 words, half of the words (i.e. 63) are repeated in each row and remaining 64 words are unique to the dictionary. Thus if we keep row size to 256 for ensuring again an 8-bit code to row number, than the total number of words which can be kept in dictionary are \(64 \times 256 + 63 = 16447\), which is plenty enough, as we had seen that the number of words (having frequency of 2 or more) which we found normally in the files, of different corpus, of size varying from 2 MB to 10 MB is ranging in between 10,000 to 22,000. So the average value comes to be around 16000. The structure of the dictionary will look like as shown in Figure 4.7 below.

![Figure 4.7 Structure of two-dimension word dictionary (WBTC-C)](image)

Even though it seems that word0, word1, . . . , word62 are repeated in each row, but actually they are stored only once and are assume logically to be present in every row. The idea behind using two-dimension dictionary is to code the dictionary with the row number and column number. The most frequent words are stored in each row along with some other unique words, therefore the probability of finding the consecutive words in same row increases and we will be able to code the word with 8-bit only. We will need to specify the row number only when two consecutive words are not found in the same row. In this case, the escape symbol is to be stored to indicate the change in row and then
followed by the row number in which the word is found, along with the column number. Thus more compression is achieved when the consecutive word are found in the same row, because only 8-bit code is needed instead of 16-bit code.

The dictionary is created in the same way as explained in previous method 4.3.4, the only difference is in the way it is now interpreted as two-dimension instead of single dimension in this method.

Compression

The source file is scanned word by word. The scanned word is searched in the dictionary, and if found the index value will be computed by the equation given below:

\[
row \ number \ = \ \frac{position - 63}{64} \\
column \ number \ = \ (position - 63) \mod 64
\]

where \( position \), is the location of word in the dictionary from starting.

If the newly computed row number is equal to previous row number (initially the row number is zero), then only the column number is converted to codeword by making its MSB to 1 (i.e. by adding 128 to it) and is stored in the compressed file or else if there is a mismatch in previous and current row number, then an escape symbol ‘11111111B’ is stored followed by new row and column number. This new row number now becomes the old row number or previous row number.

If the word is not found in the dictionary, then it is stored as it is in the compressed file. Similarly, all non-words are also stored as it is in the compressed file. The only part which is compressed is the word found in the dictionary. Thus, we achieve compression upto certain extent and also keeping the redundancy by storing some words as it is in the compressed file.

Decompression

The decompression process is very simple. The word dictionary is read from the dictionary file. The bytes are read from the compressed file. If it is plain ASCII character then it is stored in the decompressed file as it is. If it is an escape symbol for change in row, then new row number is read from the compressed file followed by column number.
If it is not escape symbol, then the byte value is treated as column number, and the new row number is equal to the previous row number. The index value (i.e. position) of the word in the dictionary is calculated by the equation given below:

\[
\text{index value} = (\text{row} \times 64) + 63 + \text{column number}
\]

The entire word of the dictionary located at the index is stored in the decompressed file. Thus, the file is decompressed after reading every byte.

4.3.6. Word Based Text Compression Method using Dynamic Dictionary (WBTC-D)

In the above two methods, the dictionary is built explicitly and is stored along with the compressed file. But in this method the dictionary is built on-the-fly and in the similar way the dictionary is to be built during decompression process. The overhead of external dictionary is reduced, but then we won’t be able to search the phrase in the compressed file, which was possible in above methods.

The file is scanned only once. Initially the dictionary is null. The first word read from the source file is stored as it is in the compressed file and at the same time it is stored in the dictionary. From the next word, the word is first search in the dictionary, and if found the index value of that word is stored in the compressed file, else that word is written as it is in the compressed file, and then added to the dictionary. The similar process is adopted in the decompression program, where the dictionary is created in the similar way it is created in the compression program. Hence in this method, we can say that there is no overhead of the dictionary.

4.3.7. Word Based Text Compression Method using Static Dictionary (WBTC-E)

A static dictionary method uses the same dictionary for all files to be compressed, thus such dictionaries are used only in specific applications where the files to be compressed contain many common words. A static dictionary is simply a set of words from the input alphabet with corresponding codewords. Ideally the dictionary should consist of words common to input strings which are typically encountered in the application domain.
Clearly the dictionary used need to be available to both the compression algorithm and its corresponding decompression algorithm. The static dictionary is created from the set of different corpus. This method is equivalent to method WBTC-C, but the only difference here is that in this method the dictionary is static and will not be considered as overhead to the compressed file, but will be an integral part of compression program, whereas in method WBTC-C the dictionary is created for a particular file and is considered as an integral part of compressed file, thereby increasing the overhead of the dictionary created.

In WBTC-C method, the word stored in dictionary was encoded with 8-bit value. The numbers of words used in WBTC-C method are 16447 and that is justifiable because the dictionary belongs to a single file. But in the case where static dictionary is to be build up from multiple files the number of words will be far more than 16447. Hence we decide to encode the word by 16-bit instead of 8-bits. The number of words kept in each row is restricted to 32768 only. If we are using only 16-bit code, then the MSB is used to differentiate between the normal ASCII character and encoded value of the words. Therefore, only 15-bit remains to point to the word in the dictionary, hence \(2^{15} - 1\) i.e. 32767 words are kept in one row. But to indicate a change in row an escape symbol 0xFF (i.e. 11111111B) is used and the corresponding 256 combinations are omitted Therefore number of words which can be kept in dictionary are 32767 – 256 = 32511. Out of these 32511 words, 32000 words are repeated in each row and remaining 511 words are unique to the dictionary. Thus if we keep row size to 256 for ensuring again an 8-bit code to row number, than the total number of words which can be kept in dictionary are 511 * 256 + 32000 = 162816, which is plenty enough. We had collected words of frequency greater than 2 from 45 files of different corpus and the number of words found is maximum 130000. The structure of the dictionary will look like as shown in Figure 4.8 below.
Even though it seems that \textit{word0, word1, \ldots, word31999} are repeated in each row, but actually they are stored only once and are assume logically to be present in every row.

The idea behind using two-dimension dictionary is to code the dictionary with the row number and column number. The most frequent words are stored in each row along with some other unique words, therefore the probability of finding the consecutive words in same row increases and we will be able to code the word with 16-bit only. We will need to specify the row number only when two consecutive words are not found in the same row. In this case, the escape symbol is to be stored to indicate the change in row and then followed by the row number in which the word is found, along with the column number. Thus more compression is achieved when the consecutive word are found in the same row, because only 16-bit code is needed instead of 18-bit code.

\textit{Dictionary Creation}

The files are selected from the different set of the corpus. Every file is scanned and the number of words having frequency count $> 2$ is stored in corresponding dictionary of that file. Thus all possible words are stored in the dictionaries of respective file. Now again all those dictionaries are scanned and the frequency of common words from different dictionaries is added. The new formed dictionary is sorted with respect to frequency in descending order so most probable words will appear in the front of the dictionary. The
dictionary created will be part of the compression program and will be available to the decompression program.

The compression and decompression process is similar to that of method WBTC-C.

4.4. COMPARISON AMONG PROPOSED METHODS

In all the methods except CBTC-A, we are getting improved compression ratio when they are used as pre-compression stage to several standard existing compression methods such Arithmetic Coding, Bzip2, PPMD, PPMII and LZMA.

In CBTC-A, 5-bit coding is used to encode a character. Every character is encoded by 5-bits. Although some character groups were encoded by 8-bits, but then it was again preceded by 5-bit escape symbol. If another program read this stream of 5-bits, it will read byte by byte, therefore the numbers of symbols are thus not minimized but are maximized to full extent i.e. 256. All combinations of bytes from 0 through 255 are generated because of continuous stream of 5-bits. In Arithmetic coding, the probability of occurrence of symbol (byte) is considered and therefore we can say that this method is not suitable to use as a pre-compression stage to arithmetic coding method. The experimental results shows that compression ratio achieved is very poor than Bzip2, Arithmetic Coding, PPMD, PPMII and LZMA. The dictionary used here is static and hence there is no overhead of dictionary in this method. Because of static dictionary, it becomes useful for searching the pattern directly in the compressed file.

In CBTC-B, 8-bit coding was used to encode 2-character groups, where as 16-bit coding was used to encode 3 & 4 character groups. The single characters were not encoded but were stored as it is in the compressed file. Thus creating some sort of context redundancy in the compressed file. This redundancy is exploited in Arithmetic Coding method giving improved compression ratio. In this method all possible 4-character and 3-character groups are stored in the dictionary and only 32 most probable 2-character groups are stored. The dictionary is overhead to the compress file and is integral part of the compressed file. This method is also suitable for direct searching the pattern in the
compressed file without decompressing it. The method is used as pre-compression stage to Arithmetic Coding technique and gives 5.38% of improvement in compression ratio.

In WBTC-A, 16-bit coding is used to encode the words, partial words and non-words. The dictionary is created for words, prefix words, suffix words and non-words. The words with frequency greater than 2 are stored in the dictionary. Similarly, the prefix and suffix words with frequency greater than 2 are stored in the dictionary. This method is used as pre-compression stage to standard methods such as Bzip2, PPMd, PPMII and LZMA etc. This method gives 1% of improvement in compression ratio when used as pre-compression stage to Bzip2, 0.67% of improvement in compression ratio when used as pre-compression stage to PPMd, 1.92% of improvement in compression ratio when used as pre-compression stage to PPMII, and 0.16% of improvement in compression ratio when used as pre-compression stage to LZMA. This method is also suitable for direct searching the pattern in the compressed file without decompressing it and the experimental results shows that time required to search the phrase in compressed form is 49% less than that of normal searching.

In WBTC-B, again 16-bit coding is used similar to that of method WBTC-A. The only difference with WBTC-A is that here all the words occurring in the source file are stored in the dictionary with maximum limit of 32768 words (i.e. $2^{15}$ only). The dictionary is assumed to be single dimension with index value ranging from 0 to 32767. This method gives 1.22% of improvement in compression ratio when used as pre-compression stage to Bzip2 and PPMd, 2.39% of improvement in compression ratio when used as pre-compression stage to PPMII, and 0.55% of improvement in compression ratio when used as pre-compression stage to LZMA. This method is also suitable for direct searching the pattern in the compressed file without decompressing it and the experimental results shows that time required to search the phrase in compressed form is 40% less than that of normal searching.

In WBTC-C, an 8-bit coding is used to encode the words in the dictionary. The dictionary is assumed to be of two dimensions instead of one dimension as in the case of method WBTC-A and method WBTC-B. The numbers of words are restricted to 16447 whereas
in method WBTC-A and WBTC-B they were up to 32768. This method gives 1.32% of improvement in compression ratio when used as pre-compression stage to Bzip2, 1.23% of improvement in compression ratio when used as pre-compression stage to PPMd, 2.12% of improvement in compression ratio when used as pre-compression stage to PPMII, and 0.46% of improvement in compression ratio when used as pre-compression stage to LZMA. This method is also suitable for direct searching the pattern in the compressed file and the experimental results shows that time required to search the phrase in compressed form is 39% less than that of normal searching.

In WBTC-D, the dictionary is built dynamically i.e. when the file is parse for compression at that time itself the dictionary is created. Again the number of words is restricted to 32768 words. The dictionary is assumed to be of single dimension. This method gives 1.72% of improvement in compression ratio when used as pre-compression stage to Bzip2, 1.75% of improvement in compression ratio when used as pre-compression stage to PPMd, 2.75% of improvement in compression ratio when used as pre-compression stage to PPMII, and 3.49% of deterioration in compression ratio when used as pre-compression stage to LZMA. But as the dictionary is built on the fly, this method is not suitable for searching the pattern directly in the compressed form.

In WBTC-E, the dictionary is static. The static dictionary is build separately from the particular application domain. All the files are scanned and the statistics of words are collected and a common dictionary is build from all the files. This dictionary is then used for compressing all the files from that application domain. There is no overhead of the dictionary in this method as compared to CBTC-B, WBTC-A, WBTC-B, and WBTC-C. This method gives 9.18% of improvement in compression ratio when used as pre-compression stage to Bzip2, 7.93% of improvement in compression ratio when used as pre-compression stage to PPMd, 9.24% of improvement in compression ratio when used as pre-compression stage to PPMII, and 8.62% of improvement in compression ratio when used as pre-compression stage to LZMA. This method is also suitable for direct searching the pattern in the compressed file without decompressing it and the experimental results shows that time required to search the phrase in compressed form is 41% less than that of normal searching. The limitation of this method is that it can
perform well only when the source file to be compressed is from the same application domain. As compared to other methods, this method outperforms over all other methods, if at all the file to be compressed is from the particular application domain.

In next chapter the implementations issues of the proposed methods are discussed, whereas in chapter 6, the experimental results and comparison of the results with other standard methods are given.