CHAPTER 3

TEXT PREPROCESSING FOR DATA COMPRESSION

3.1 PRELIMINARIES

The last ten years has witnessed an unprecedented explosion in the volume of digital data transmitted over the Internet, representing text, video, sound, images, computer programs etc. Hopefully, this trend will last, heralding an era of much greater explosion of higher magnitude wherein it will become possible to develop algorithms that can most effectively use available network bandwidth by maximally compressing data.

One approach for trying to attain better compression ratios is to develop different compression algorithms. A number of sophisticated algorithms have been proposed for lossless text compression such as Huffman encoding[46], Arithmetic encoding [2,53,59,62,63], Lempel Ziv family [99,100], Dynamic Markov Compression (DMC), Prediction by Partial Matching (PPM) [18] and Burrows Wheeler Transform (BWT) [15] based algorithms. It seems unlikely that major new progress will be made in this area.

An alternative approach, which is explored in this chapter, is to develop intelligent, reversible transformations called preprocessing algorithms that can be applied to the source text that improve an existing, or backend, algorithm’s ability to compress and also offer a sufficient level of security of the transmitted information.
3.2 MODERN PARADIGM OF TEXT PREPROCESSING

The preprocessing of the text and the transformation of it into some intermediate form forms the basic philosophy of secure compression. Moreover, this intermediate form lends itself for compression with better efficiency. It is also capable of exploiting the natural redundancy of the language to make the transformation. Text preprocessing algorithms are reversible transformations which are performed before the actual compression scheme prior to encoding and behind the decompression scheme after decoding.

Figure 3.1 illustrates the modern paradigm of text preprocessing.

![Text Compression Paradigm](image)

**Figure 3.1: Text Compression Paradigm Incorporating a Lossless, Reversible Transformation**

The transformation is designed to make it easier to compress the source file. The original text file is provided as input to the transformation, which generates the transformed text as the output. This output is provided to an existing, unmodified data compression algorithm, which compresses the transformed text. To decompress, one merely reverses this process, by first invoking the appropriate decompression algorithm, and then providing the resulting text to the inverse transform. The transformation must be exactly reversible, so that the overall lossless text compression paradigm is not compromised. The data compression and decompression algorithms are unmodified, so they do not exploit information about the transformation while
compressing. The intent is to use the paradigm to improve the overall compression ratio of the text in comparison with what could have been achieved by using only the compression algorithm.

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 can easily be separated into two independent parts: a preprocessing algorithm and a standard compression algorithm, which are processed sequentially one after the other.

Textual preprocessing ideas have been described by, among others, Kruse and Mukherjee [51,52], Teahan [89], Grabowski [35], Franceschini et al. [31], Sun et al. [87,88], and Abel and Teahan [1].

### 3.3 NEED FOR PREPROCESSING ALGORITHMS

There is an inherent redundancy in the English language that inhibits every combination of letters from representing a proper English word. A compression algorithm, on an ideal level, is construed to be capable of representing such redundancies in its adaptive compression model, inclusive of flexibility of its model in tune with the input text, exploiting the existing redundancy.

Two practical problems that handicap many compression algorithms that use adaptive compression models and render them impractical in the use of many applications are:

- The methods used by them are very limited in scope, not making provision for representation of the full redundancy of the English language.
- The amounts of memory and/or CPU time required by them are very large.
Thus, there is a need to develop effective preprocessing algorithms which can fully exploit the natural redundancy of the language in a much better way and it has been observed that the preprocessing of the text prior to conventional compression will improve the compression efficiency much better.

### 3.4 DICTIONARY-BASED PREPROCESSING ALGORITHMS

The Dictionary-based preprocessing is a technique for compressing texts in a natural language. The basis of the technique is replacing whole words with shorter codes. Here, the exploitation of redundancy in English can be experimented through the use of a sophisticated preprocessing algorithm that operationalizes a fixed dictionary of English words and transforms the input text by matching words in the input text with words in the dictionary, and replacing such words with a pointer into the dictionary.

This can be done in two basic ways.

- The words in the dictionary can be replaced with some (compact) binary encoding of a pointer into the dictionary by the preprocessing algorithm. The advantage of the method is that the length of the document is sizably reduced during preprocessing. However, the output text of the preprocessor requires further compression by a standard data compression algorithm resulting in a much better and more improved compression ratio. This is made possible because the algorithm affects actual words and not the character combinations.

- The preprocessing algorithm is capable of replacing words in the dictionary with some other character sequences, which are approximately of the same length as the original word. However, they
are found to be easier for compression for the back end compression algorithm since they are endowed with a higher degree of visible redundancy, accessible to the model used by the backend data compression algorithm.

It is observed from [52] that the results of the second approach were much better than the results of the first approach discussed above.

The main objective of the preprocessing algorithm is to bring about a replacement of English words from the input text with a character combination of merely the same length. Another objective is to increase the overall compression ratio with that character combination which contains a higher degree of redundancy than the original word.

This scheme is dependent upon a static dictionary to be mutually shared by the sender and the receiver. This dictionary is kept separately from the compressed text and shared by the compressor and decompressor.

It is not necessary that the dictionary should contain all the words that appear in the input text. Words in the input text that are not found in the dictionary are not encoded by the preprocessor but copied verbatim.

3.5 BURROWS WHEELER COMPRESSION ALGORITHM (BWCA)

In 1994, Burrows and Wheeler developed a new data compression algorithm [15], based on the Burrows–Wheeler transform called Burrows Wheeler Compression Algorithm (BWCA) or Block-sorting Compression Algorithm. There are several variations of block-sorting compression algorithm [7,22,23,24,28].
The block diagram of BWCA is shown in Figure 3.2.

![Block Diagram of Burrows – Wheeler Compression Algorithm (BWCA)](image)

**Figure 3.2: Block Diagram of Burrows – Wheeler Compression Algorithm (BWCA)**

It is composed of four stages. The first step performs the Burrows–Wheeler Transform (BWT). The second step implements the Move-to-Front (MTF) [3] transform converting the output of the BWT to a form that can be better compressed by an entropy coder. The third step is a specialized version of a Run Length Encoding (RLE) called RLE-0 (in some versions of the BWCA this step does not occur). The last step is an entropy coder, which typically is an arithmetic coder. In this algorithm, the resulting text can be easily compressed with fast locally adaptive algorithms, such as Move-to-Front coding combined with Huffman or arithmetic coding preceded by a Run Length Encoding (RLE).

Given a text file, BWCA works like this. First Burrows-Wheeler Transform is applied on it which produces a new text, which is suitable for Move-To-Front encoding as it has a great number of sequences with identical letters. The result is another new text which is more suitable for standard statistical compression algorithm, usually preceded by Run Length Encoding (RLE) because RLE produces many small numerical values. The actual compression is being done in the last step as shown in the Figure 3.2 i.e. entropy encoder. The other steps are meant to ensure that the Arithmetic encoding / Huffman encoding are able to compress the data efficiently.

The most distinguishing feature of the block-sorting compression algorithm is that it is an offline algorithm as the processing of the input data does not take place sequentially here. Division of the input data into blocks is done by the
encoder since all the data and additional data structures required for sorting it, might not fit in a memory. The compression for each block is performed separately by the encoder. Better compression effectiveness is usually achieved by larger blocks at the expense of compression speed and memory requirements. The size of the block for widely used bzip2 [75] is 900 KB.

The compression effectiveness achieved by BWT-based achievers is markedly better than that of compressors from Lempel-Ziv family. More significantly, the compression speed, and particularly the decompression speed, is lower than archivers from Lempel–Ziv family. BWT class is between dictionary-based compressors from Lempel–Ziv family and predictive compressors.

Chapin and Tate [17] and later Chapin [16] 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 later stages of the BWCA. They compare several heuristic and computed reorderings where the heuristic approaches always achieve a better result on text files than the computed approaches. The average gain for BWCA using heuristic reorderings over the normal alphabetic order was 0.4% on the text files of the Calgary Corpus. Balkenhol and Shtarkov [7] use a very similar heuristic alphabet reordering for preprocessing with BWCA. Kruse and Mukherjee [52] describe a different alphabet reordering for BWCA. It also describes a bigram encoding method and a word encoding method which is based on their star encoding[30,51,52].

Grabowski comes forward with proposals for several text preprocessing methods in his publication [35] which are a step forward for improvement for BWCA but there is a scope for the use of some techniques also for other compression schemes. In addition to the already existing techniques like alphabet
reordering, bigram, trigram and quadgram replacement, three new algorithms are suggested by Grabowski. The first one is known as capital conversion. At the beginning of the word capital letters are replaced by an escape symbol and the corresponding lower character. Omission of the replacement takes place if the second letter of the word is also capitalized. This technique helps in a large measure to increase context dependencies and similarities between words, which can be exploited by standard compression schemes. The second algorithm is space stuffing where there is a space symbol 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 brings in space symbols in the place of EOL symbols. It also performs separate encoding of the former EOL positions which are represented by the number of blanks occurring after the previous EOL symbol. These numbers are encoded either within the symbol stream itself or in a separate data stream. The reason for omission of EOL coding in his comparisons is that it suffers from unstable side effects. An average gain for BWCA of 2.64% on the 10 text files of the Calgary corpus is achieved when compression is performed using his preprocessing algorithm without EOL coding. As Grabowski employs a set of fixed bigrams, trigrams and quadgrams, his proposal requires an external dictionary and is language-dependent.

**Burrows Wheeler Transformation (BWT)**

Burrows Wheeler Transformation was developed by David J. Wheeler in 1983. Michael Burrows and David J. Wheeler presented it publicly in 1994 as a part of block-sorting compression algorithm [15]. BWT is not actually a compression scheme. BWT is a reversible transform that converts the data into a format that is generally more compressible. It takes a block of data and reorders it using a sorting algorithm. It is also called Block sorting. The most complex and time-consuming task here is sorting the rows, but present implementations can do this with an acceptable complexity. The resulting block of text contains the same
symbols as the original, but in a different order. The transformation groups similar symbols, so the probability of finding a character close to another instance of the same character increases substantially. Widely used implementations of BWT include the freely available bzip and bzip2 compression tools by Julian Seward distributed under the GNU (General Public License).

During the encoding process, the complete source symbols are treated by the algorithm in such a way that only the order of occurrence of the symbols is entirely modified. In this stage the entire source sequence of symbols undergoes permutations and another fresh symbol sequence takes shape which possesses some auspicious attributes that enhance compression. The original source sequence is then retrieved through the decoding process.

The main purpose of this transformation is to shuffle the symbols in the source sequence S, in order to derive L, a new sequence which gives better compression. Since L is actually a permutation of S, the length of the original array S and resulting array L are of the same size. Here, order of the symbols in the original array S is changed to get a new array L which can be compressed more efficiently.

In order to get L, perform the following steps:

- First shift the string S one symbol to the left in a circular way. What is meant by circular is that the leftmost symbol in the array is shifted out of the array, and then added back from the right and becomes the rightmost element in the array.

- When the circular shift is repeated n - 1 times, the n x n matrix can be generated where n is the number of symbols in the array, and each row and the column is a particular permutation of S.
Example 3.1

The Burrows Wheeler Transformation for string S = ‘MAXIMUM’ can be done as follows:

Step 1: Shift the string one symbol to the left in a circular way.

\[
\begin{array}{cccccc}
M & A & X & I & M & U \\
A & X & I & M & U & M \\
X & I & M & U & M & M \\
I & M & U & M & M & A \\
M & U & M & M & A & X \\
M & U & M & M & A & X \\
U & M & M & A & X & I \\
M & M & A & X & I & M \\
M & M & A & X & I & M \\
\end{array}
\]

Step 2: Sort the rows of the matrix in lexicographic order so the matrix becomes

\[
\begin{array}{cccccc}
A & X & I & M & U & M \\
I & M & U & M & M & A \\
M & A & X & I & M & U \\
M & M & A & X & I & M \\
M & U & M & M & A & X \\
M & U & M & M & A & X \\
U & M & M & A & X & I \\
X & I & M & U & M & M \\
\end{array}
\]

Step 3: The last column is named L, which is what is needed in the BWT for encoding, where \( S_i \) indicates the first symbol of the given array \( S \).
The last bold column of this matrix marks the BWT-transformed version of the string MAXIMUM.

The recovery of the original string S from L can be said to be the goal of the reverse process of the transform. It becomes necessary to find the order relationship in which the symbols occurred in the original string S as L is a permutation of S. Given any symbol in L, say the symbol $s_i$ at k location, i.e. $L[k] = s_i$, the next symbol in the original string S is $S_{i+1} = F[k]$. If the position of $S_{i+1}$ is known in L, it becomes clear to know the $s_{i+2}$. In this way, every symbol in S can be retrieved.

Since the first symbol in S (i.e. 'M') is also transmitted in BWT to the decoder for decoding purposes which results in the transformed data becoming even larger than its original form, but the transformed data is more compressible.

**Move-To-Front Encoding (MTF)**

Move-to-Front [3] is a self-organization heuristic for lists. In the field of data compression, MTF is used to convert the data into a sequence of integers, with hope that values of integers are low and could be effectively encoded using a statistical coding.
A list of symbols, known as a MTF list, is maintained by the MTF encoder which is initialized with all the symbols that occur in the data to be compressed. The updation of the list takes place when the encoder outputs its position on MTF list (as an integer) for each symbol from the data. What follows then is moving of a currently encoded symbol from its current position in the list to the front of the list. The most distinguishing feature of this technique is that recently used symbols are near the front of the list. Equal symbols can be hopefully expected to appear often close to each other in the data and as a result these symbols will be converted to small integers. Usually small integers appear more frequently so they are encoded in fewer bits than larger integers using a statistical coding, for example, the Huffman or the arithmetic coding.

**Example 3.2**

Encoding the sequence CAABABBCA from a source alphabet (A,B,C) using MTF is shown in Figure 3.3.

As the first step, the MTF list is initialized with the input alphabet (A,B,C).

<table>
<thead>
<tr>
<th>An MTF List</th>
<th>Data to be encoded</th>
<th>Encoded data</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C</td>
<td>CAABABBCA</td>
<td>2</td>
</tr>
<tr>
<td>C A B</td>
<td>AABABBCA</td>
<td>1</td>
</tr>
<tr>
<td>A C B</td>
<td>ABABBCA</td>
<td>0</td>
</tr>
<tr>
<td>A C B</td>
<td>BABBBCA</td>
<td>2</td>
</tr>
<tr>
<td>B A C</td>
<td>ABBCA</td>
<td>1</td>
</tr>
<tr>
<td>A B C</td>
<td>BBCA</td>
<td>1</td>
</tr>
<tr>
<td>B A C</td>
<td>BCA</td>
<td>0</td>
</tr>
<tr>
<td>B A C</td>
<td>CA</td>
<td>2</td>
</tr>
<tr>
<td>C B A</td>
<td>A</td>
<td>2</td>
</tr>
</tbody>
</table>

**Figure 3.3: MTF Encoding Example**
As a next step, the encoder processes ‘C’ by checking its position (counting from 0) on the list and outputs 2. The MTF list is updated, and ‘C’ is moved to the front of the list. The remaining symbols are moved one position further from the front of the list. Then, the encoder processes ‘A’, which is at the position number 1. The list is updated to <A, C, B> and the encoder processes the second ‘A’ and so on.

The use of MTF is a part of several other compression algorithms including the Burrows–Wheeler Transform (BWT). There are many MTF variations, for example MTF1 [7]. When MTF1 encounters a symbol that is at the position 1 (counting from 0) of the list, then it moves the symbol to the front. Otherwise, it moves encountered symbol to the position 1. Of course, if the symbol is on the front of the list, its position is not changed.

3.6 STAR TRANSFORMATION

Holger Kruse and Amar Mukherjee developed Star Encoding [30,51,52], a unique case of word encoding. In this method the words are replaced by sequence of symbols. This method replaces words by a symbol sequence that frequently consists of the single symbol ’*’ that is replicated. This transformation starts with an English language dictionary D. This dictionary D is then partitioned into a set of disjoint dictionaries Ds each containing words of length s, s = 1,2,3, ..., n. Each sub dictionaries Ds is then sorted based on the frequency of words in English language using information obtained from Horspool and Cormack[39]. 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.

98
For the words in the dictionary $D$s the following encoding scheme is used:

- Series of stars are used to represent the word that appears first in the dictionary.
- For the next following 52 words, sequence of $s-I$ stars are employed which are then followed by a particular letter from the alphabet $\Sigma=(a,b...,z,A,B...Z)$.
- Likewise, the same process of encoding is applied to the next following 52 words also, with the exception of the single letter appearing in the last but one position.
- The same procedure is carried on repeatedly until all the letters occupy the first position in the sequence.
- Then comes the part of $s-2$ *’s for the next immediately appearing group of words and the left behind two positions are occupied by unique pairs of letters from the alphabet.
- This course of action can be continued to obtain a total of $53^s$ unique encodings that are more than satisfactory for English words.

Thus, as seen from the above mentioned steps of Star encoding process, it is quite obvious that the result of encoding process is to bring into being in the transformed text a great many number of ‘*’ characters. Thus the ‘*’ becomes the most recurrently appearing character which increases the redundant nature of the data. Correspondingly, if a new word, which is not in the English dictionary, appears in the original input text, then it will be delivered to the text as it is, without any modification. There should also be provisions by the transformed text for handling special characters, punctuation marks and capitalization. The space character indicates word separator. The ‘~’ character indicates that the first letter of the input text word is capitalized. The character ‘`’ indicates that all the letters of input text word are capitalized. The ‘^’ character followed by capitalization
mask denotes capitalization of all the characters except the first letter. The ‘\’ character acts as an escape character to be used with the characters ‘*’, ‘~’, ‘\’, ‘^’, and ‘\’ that appears in the input text.

Thus, by applying all the needed encoding schemes as mentioned above, the transformed text can be delivered as an input to any of the existing lossless text compression algorithms such as bzip2. When applied to bzip2, the original text experience two transformations,

- *-transform
- BWT transform

### 3.7 LENGTH INDEX PRESERVING TRANSFORM (LIPT)

Fauzia Awan and Amar Mukherjee [5] proposed a dictionary-based reversible lossless text transformation scheme called Length Index Preserving Transformation (LIPT). They have created a dictionary ‘D’ of words in the corpus which is partitioned into disjoint dictionaries $D_i$, each containing words of length $i$, where $i = 1, 2, ..., n$. According to the frequency of words, each dictionary $D_i$ is partially sorted. Here only mapping of codewords has been changed compared to that of Star encoding. Here the codewords usually have a different length from that of the original words. Here they use the character ‘*’ at the beginning of each code word which distinguishes original and codewords. The length of a word ($C_{len}$) is encoded right after ‘*’ which refers to a character in the alphabet [a-z, A-Z] each denoting a corresponding length (1-26, 27-52). After the length, an index into the sub dictionary is encoded. The index (c) which cycles through [a-z, A-Z] is encoded as a number representation of base 52. So, for the first 52 words, the encoding is denoted as $*c_{len}c$, for the next 2,704 words the encoding is represented as $*c_{len}cc$ and for the next 1,37,852 words the encoding is represented as $*c_{len}ccc$. The dictionary of words containing the transformed words are represented as
$D_{LIPT}$. This scheme allows for a total of 1,40,608 encodings for each word length. They have used an English dictionary that has about 60,000 words and takes about 0.5 MB in its uncompressed form. The dictionary needs to be the same at the encoding and decoding ends. Here, both the compressor and the decompressor have access to the same dictionary D and its $D_{LIPT}$ correspondent. If the word in the input text is not in the English dictionary (viz. a new word in the lexicon) it will be passed to the transformed text unaltered. They have also handled special characters, punctuation marks and capitalization in their transform.

The encoding process can be explained as follows:

• Words in the input file are extracted and are searched in the dictionary D using two level index search method.

• If found, its position and block number are noted and the corresponding transformation at the same position and block is noted in $D_{LIPT}$. This is the encoding for the respective input word. If the input word is not found in the dictionary D, it is transferred as it is.

• Once all the input text has been transformed as mentioned above, the transformed text is then fed to a compressor (e.g. bzip2, PPM, etc.)

The decoding process can be explained as follows:

• Using the same compressor as was used at the sending end, the transformed LIPT text is recovered.

• Reverse transformation is applied on this decompressed transformed text. Words with ‘*’ represent transformed words and without ‘*’ represent non-transformed words and do not need any reverse transformation. The length character in the transformed words gives the length block and the next three characters give the offset in the respective block and then there might be a capitalization mask. The words are looked up in the original dictionary $D$ in the respective
length block and at the respective position in that block as given by the offset characters. The transformed words are replaced with the respective English dictionary $D$ words.

- The capitalization mask is applied.

This algorithm achieves some compression at the preprocessing stage itself and retains enough context and redundancy for the compression algorithms to give better results. The average bpc using bzip2 is 2.28, and using bzip2 with LIPT gives an average bpc of 2.16 which is 5.24% improvement. The average bpc using PPMD (order 5) is 2.14 and using PPMD with LIPT gives an average bpc of 2.04, registering an improvement of 4.46% over original PPMD. The average bpc using gzip is 2.71 and using gzip with LIPT the average bpc is 2.52 showing an improvement of 6.78% over the original gzip-9. bzip2 with LIPT shows 13.44% improvement over the word based Huffman. Another important result found was that bzip2 with LIPT has bpc much closer to original PPMD (within 1%) but is much faster in time performance. It has been observed that for normal text files, the bpc decreases as the file size increases. This observation is important for the efforts going on to achieve the PPM compression performance with higher speed.

Average compression time, using LIPT with bzip2 -9, gzip -9, and PPMD is 1.79 times slower, 3.23 times slower and fractionally (1.01 times) faster compared to original bzip2, gzip and PPMD respectively. The corresponding results for decompression times are 2.31 times slower, 6.56 times slower and almost the same compared to original bzip2, gzip and PPMD respectively. Compression using bzip2 with LIPT is 1.92 times faster and decompression is 1.98 times faster than original PPMD (order 5). The increase in time over standard methods is due to time spent in preprocessing the input file. gzip uses –9 option to achieve maximum compression and, therefore, it is found that the times for compression using bzip2 are less than gzip. When maximum compression option is not used, gzip runs much faster than bzip2.
3.8 THREE NEW TRANSFORMS: ILPT, NIT, LIT

In [6] Fauzia Awan describes three new lossless reversible text transforms based on LIPT. The first transform is Initial Letter Preserving transform (ILPT) which is similar to LIPT except that based on the lexicographic order of starting letter of the words, the dictionary is sorted into blocks. The words are sorted in each letter block according to the descending order of frequency of use. In ILPT the transformed word is denoted as $C_{\text{init}}[c][c][c]$, where $C_{\text{init}}$ denotes the initial (starting) letter of the word where as in LIPT it is denoted as $C_{\text{len}}[c][c][c]$ where $C_{\text{len}}$ denotes the length of the word. All other steps are similar to that of LIPT. bzip2 -9 with ILPT shows an improvement of 6.83% over the original bzip2 -9 and 1.68% improvement over bzip2 with LIPT.

The second transform is Number Index Transform scheme (NIT) which uses linear addressing scheme with numbers ranging from 0 – 59,950 to the 59,951 words in the dictionary. In this scheme the English dictionary $D$, is sorted according to length of words and then sorted according to frequency within each length block, which gave deteriorated performance compared to LIPT. So the dictionary is sorted globally according to descending order of word usage frequency. The transformed words are still denoted by character "*". The first word in the dictionary is encoded as "*0", the 1000th word is encoded as "*999", and so on. Special character handling is done here which is similar to that of LIPT. bzip2 -9 with NIT shows 6.83% improvement over the original bzip2 -9 and 1.68% improvement over bzip2 with LIPT.

Combining the approach taken in NIT and using letters to denote offset, a new transformation called Letter Index Transform (LIT) is arrived which is the same as NIT except that the letters of the alphabet [a-z; A-Z] are used to denote
the index or the linear address of the words in the dictionary, instead of numbers. bzip2 -9 with LIT shows 7.47% improvement over the original bzip2 -9 and 2.36% improvement over bzip2 with LIPT. The sizes of the transform dictionary vary with the transform. The size of the original dictionary is 557,537 bytes. The sizes of the transformed dictionaries for LIPT, ILPT, NIT, and LIT are 330,636 bytes, 311,927 bytes, 408,547 bytes and 296,947 bytes respectively. It has been observed that LIT has the smallest size dictionary and shows uniform compression improvement over other transforms for almost all the files.

Since the performance of LIT is better than that of other transforms, it is further compared with PPMD, YBS, RK, and PPMonstr. PPMD with LIT shows an improvement of 6.88% and 2.53% over original PPMD (order 5) and PPMD with LIPT respectively. RK with LIT has an average bpc value lower than that of RK with LIPT. LIT performs well with YBS, RK, and PPMonstr, giving an improvement of 7.47%, 5.84%, and 7.0%, respectively, over the original methods. It is also important to note that LIT with YBS outperforms bzip2 by 10.7% and LIT with PPMonstr outperforms PPMD by 10%.

In LIPT, words in each length block in English dictionary D are sorted in descending order according to frequency. For ILPT, there is a sorting based on descending order of frequency inside each initial letter block of English dictionary D. For NIT, there is no blocking of words. The whole dictionary is one block. The words in the dictionary are sorted in descending order of frequency. LIT uses the same structure of dictionary as NIT. Sorting of words according to frequency plays a vital role in the size of the transformed file and also its entropy. Arranging the words in descending order of usage frequency results in shorter codes for more frequently occurring words and longer codes for less frequently occurring words.
3.9 STAR NEW TRANSFORM (StarNT)

In [87,88], Sun et al. presented a dictionary-based fast lossless text transform algorithm called StarNT. This algorithm is based on LIPT. But it differs from LIPT by removing of the length of the original word from the codeword. Here the meaning of ‘*’ has also been changed. Originally the character ‘*’ denotes the beginning of a codeword. But here it means that the following word does not exist in the transform dictionary D. Here, the encoding/decoding time of the back end compression algorithm can be minimized by reducing the size of the transformed intermediate file which is the main reason for this change. Here the dictionary is no more divided into subdictionaries according to the word lengths because of the removal of length of an original word from the codeword. This algorithm stores 312 most frequently used words in the beginning of the dictionary in the decreasing order of their frequency of occurrence. The remaining words are sorted according to their ascending lengths in the dictionary. Words of the same length are sorted in the decreasing order of their frequency of occurrence. In LIPT, codeword consists of three parts: a single symbol ‘*’, the length of the original word and an index into the sub dictionary. But in StarNT, they have got rid of two parts and here the codeword consists only of an index into the dictionary. To achieve better compression performance of backend compression algorithm, only letters [a-z,A-Z] are used to represent codeword. So the transform dictionary D consists of totally 143364 entries but the authors use a dictionary only of about 60,000 entries. StarNT uses the ternary search tree to search the words in the dictionary.

The encoding scheme works in the following manner:

1. The Transformer is initiated.
2. The input text is read from the source file.
3. If Transform Dictionary contains the word then
   - The corresponding codeword replaces the word
   - Any special symbol, if needed, is also appended

Else
   - Prefix the word with character ‘*’

4. Continue steps 2 and 3 till the end of file is reached.

The transform decoding module performs the inverse operation of the transform encoding module.

As mentioned in step 3, the need for some special symbol arises in special cases.

- The character ‘~’ - denotes that the initial letter of the corresponding word in the original text file is capitalized.
- The appended character ‘\’ - denotes that all letters of the corresponding word in the original text file are capitalized.
- The character ‘\’ is used as escape character for encoding the occurrence of ‘*’, ‘~’, ‘\’, and ‘\’ in the input text file.

The authors have proved that StarNT works better than LIPT when both are applied with backend compression algorithm, not only in the compression phase but also in the decompression phase. They have reported that StarNT achieves an average improvement in compression ratio of 11.2% over bzip2 -9, 16.4% over gzip -9 and 10.2% over PPMD. Results show that, for the test corpus that they have considered, the average compression time using the new transform with bzip2 -9, gzip -9 and PPMD is 28.1% slower, 50.4% slower and 21.2% faster compared with the original bzip2 -9, gzip -9 and PPMD, respectively. The average decompression time using StarNT with bzip2 -9, gzip -9 and PPMD is one and six times slower and 18.6% faster compared with the original bzip2 -9, gzip -9 and PPMD respectively. They have concluded that bzip2 in conjunction with StarNT is better than PPMD both in time complexity and in compression performance.
3.10 INTELLIGENT DICTIONARY-BASED ENCODING (IDBE)

Shajeemohan and Govindan [76] propose an encoding strategy called Intelligent Dictionary Based Encoding (IDBE) which offers higher compression ratios and rate of compression. According to them, IDBE involves two stages. The first is the creation of an intelligent dictionary and the next one is encoding the input text data. While creating a dictionary, words are extracted from the input files and for the first 218 words ASCII characters 33-250 are assigned as the code. For the remaining words permutation of two of the ASCII characters in the range of 33-250 is assigned. For the left out words, if any, permutation of three of the ASCII characters for each word and, if required, permutation of four of the ASCII characters is assigned. During encoding, the length of the token is determined and the length is concatenated with the code and is represented by the ASCII characters 251-254 with 251 for a code of length 1, 252 for length 2 and so on. While decoding, the length proves to be the end marker. A better compression is achieved by using IDBE as the preprocessing stage for the BWT-based compressor. The time for transmission of files has also been reduced to a greater extent. They have measured their performance in terms of bpc for the three cases i.e., simple BWT, BWT with Star encoding and BWT with their algorithm IDBE. From the results it is clear that the average bpc using Simple BWT is 2.77, and with BWT with IDBE the average bpc is 2.40, which is 13.35% improvement. The average bpc using BWT with *-encoding is 2.52 and with BWT with IDBE the average bpc is 2.40, and so the overall improvement is 4.76%.

They conducted experiments to see the difference in data transmission time between normal, uncompressed file and the file compressed using their algorithm. They have concluded that the total time for transmission of the files has been greatly reduced.
3.11 DICTIONARY-BASED TRANSFORMATION (DBT)

Robert and Nadarajan [65] propose a reversible, preprocessing algorithm called Dictionary-Based Transformation (DBT). The reduction of total number of possible byte values used in a text file is projected as the primary objective of this transformation. A text file uses byte values from 0 to 127. This transformation will result a file that uses only few combinations of byte values. For instance, transformation of a text file that has possible byte values from 0 to 127 into a file that has only possible byte values from 0 to 31 becomes possible. A welcome feature of this transformation is that it can make a sequence of bytes or a single byte to occur repeatedly. As a result of this, the back-end compressor may compress the transformed file with better compression ratio. The transform dictionary, to be jointly shared by both the transform encoding and the decoding modules, is prepared in advance. The following rules govern the attachment of each byte from 0 to 127 to a list of bytes in the dictionary.

- For every byte, collect all bytes that may appear after it, and sort them in decreasing order of their frequency. This sorted list of bytes is attached with a byte considered.

In the encoding process, a transformer, initially, begins to read the input text, byte by byte. For each byte ‘a’, the dictionary is referred to. Let ‘b’ be the byte that follows byte ‘a’. The function of the transformation is to find the position of byte ‘b’ in the attached list of ‘a’ in the dictionary. This position is written into the output file. This process is repeated for every byte in the input file.

It has been observed that, when DBT is combined with Huffman an average of 1.2 bpc is saved. At the same time when it is combined with Arithmetic coding an average of 1.17 bpc is saved and when it is combined with LZW an average of 0.58 bpc is saved.
3.12 DYNAMIC REVERSIBLE TRANSFORMATION (DRT)

Yet another preprocessing algorithm called Dynamic Reversible Transformation (DRT) has been proposed by Robert and Nadarajan [65]. Before applying the dynamic transformation they have used the pre-processing techniques like Byte-Insertion technique (BIT) and Byte-Substitution Technique (BST) to handle space and eoln.

What is required here is two passes over the source file: one for building the header and another for actual transformation. The header contains an attached list for every byte that appears in the given input file. For example, the byte ‘a’ will have an attached list that contains the sequence of bytes that follows ‘a’. This list is sorted in the decreasing order of its frequencies. The input file is transformed like a dictionary-based method after the header creation. For each byte ‘a’ in the source file, the respective header entry is referred. Let ‘b’ be the byte that follows the byte ‘a’. The method should find the position of byte ‘b’ in the attached list of ‘a’ in the header. This position is written into the output file. This process is repeated for every byte in the input file. As this technique requires two passes over the input file, it consumes more time for transformation compared with dictionary-based transformation (DBT).

It has been observed that when DRT is combined with Huffman, a significant reduction of 0.94 bpc is observed. A significant saving of 0.93 bpc is noticed, when it is combined with Arithmetic coding. On the contrary, combination with LZW does not provide the best bpc for some of the test files because LZW is an adaptive higher order data compression algorithm.
3.13 WORD REPLACING TRANSFORMATION (WRT)

Przemysław Skibiński et al. [82,84] proposed a dictionary-based preprocessing technique called Word Replacing Transformation (WRT), which is a successor of StarNT. They used the idea of replacing words in input text file with shorter codewords and used several other techniques to improve the performance of the latter compression. WRT uses an English dictionary of about 80,000 words. A word in WRT dictionary is a sequence of at least one symbol over the alphabet [a..z]. Each word in D has assigned a corresponding codeword. The length of codewords is variable and spans from one to four symbols. Ordinary text files, at least English ones, consist solely of ASCII symbols not exceeding 127, so the alphabet of codewords has 128 symbols (ASCII values from 128 – 255). As the scheme makes use of “capital conversion” technique, there is no need to use uppercase letters in the dictionary. The alphabet of codewords alphabet (128 symbols) is divided into four parts. They have used the mapping <101,9,9,9> for codewords and thus 82,820 codewords are available which is enough for 80,000 words in WRT dictionary.

They have used several additional techniques to improve the compression performance. First is q-gram replacement, which substitutes frequent sequences of q consecutive characters with single symbols. Second is End-of-Line (EOL) coding which replaces EOL symbols with spaces and to encode information enabling the reverse operation in a separate stream. The last one is surrounding words with spaces, which converts all words to be surrounded by space characters. Results show that WRT brings an improvement on the order of 8–12% in comparison with ordinary PPM or BWCA compressors and even about 21–27% with LZ77 (Lempel Ziv) compressors.
3.14 TWO-LEVEL WORD REPLACING TRANSFORMATION (TWRT)

Przemysław Skibiński [83,84] proposed a new dictionary-based preprocessing technique and its implementation called Two-level Word Replacing Transformation, an improvement of Word Replacing Transformation (WRT). Here, up to two dictionaries can be used, which are dynamically selected before the actual preprocessing starts. The first level dictionaries (small dictionaries) are specific for some kind of data (e.g., programming language, references). For natural languages (e.g., English, Russian, French), the second level dictionaries (large dictionaries) are specific. If no appropriate first level dictionary is found, then the first level dictionary is not used. Selection of the second level dictionary is analogous.

When TWRT has selected only one (the first or the second level) dictionary, it works like WRT. If TWRT has selected both the first and the second level dictionary, then the dictionaries are automatically joined (the second level dictionary is appended after the first level dictionary). If the same word exists in the first and the second level dictionary, then the second appearance of word is ignored to minimize length of codewords. On the Calgary Corpus, TWRT improves the compression performance of PPMonstr, by about 6% on average. Even for the top compressor nowadays, PAQ6, the gain is significant – 5%. On the multilingual text files, the gain is even bigger. TWRT improves the compression performance of bzip2, a popular BWT-based compressor, by about 8%. With PPMonstr and PAQ6 the gain is 8% on average. Moreover, it improves the compression speed with PAQ6 and on larger files with PPMonstr.