A General Purpose Lossless Data Compression Scheme with Improved Compression Ratio and Decompression Time and Optimized for Searching and Retrieval of String Randomly from Large Collection

A thesis submitted
for award of the degree of

Doctor of Philosophy

in

Electrical Engineering

By:
Bhadade Umesh Shantilal

सत्यं शिवं सुन्दरम्

DEPARTMENT OF ELECTRICAL ENGINEERING

FACULTY OF TECHNOLOGY & ENGINEERING

THE MAHARAJA SAYAJIRAO UNIVERSITY OF BARODA

VADODARA – 390 001 GUJARAT, INDIA

DECEMBER 2010

Acknowledgements

First and foremost, my utmost gratitude to the omnipresent God, for providing me inspiration, strength, energy and patience to start and accomplish my goal.

I sincerely thanks with deep sense of gratitude to my guide **Prof. A. I. Trivedi,** who has been the main source of inspiration. His guidance and motivation has been of the greatest help to me in bringing out this work in its present shape. The direction, advice, discussions and constant encouragement given by him have been so helpful that it enabled me to complete work successfully. He created and pointed to the path and helped in every way to reach this final destination.

I would like to show my gratitude towards **Prof. S. K. Shah**, Head of Electrical Engineering Department, Faculty of Technology & Engineering, M.S. University of Baroda, for extending his support and guidance. I also like to express my sincere thanks to **Dr. K.B. Pai**, Dean Faculty of Technology & Engineering, M.S. University of Baroda, who was remote, yet effective source of inspiration.

My **parents** showered love and affection on me. They guided me through my first steps to this milestone in my life and always supported me. They taught me to read, think and analyze.

I owe my deepest gratitude to my beloved brother **Dnyanesh** who constantly support me during my entire time-span of Ph.D. work. His mental and moral support encouraged me to all the time to do my work. I am also thankful to my sisters **Babita** and **Suvarana** for their constant moral support. I express my sincere thanks to my friend **Vinod** who was one of the motivator from Ph.D. registration to thesis submission.

I would also like to thank my **institution**, **colleagues** and other **faculty** members for their direct and indirect support.

I express my thanks to my wife **Swati** and sweet daughter **Minakshi** and nephew **Mohit** for all the support, inspiration and love given to me despite of all inconvenience caused to them due to my preoccupation with Ph.D. work.

Finally, last but not least, I thank all persons, who helped me in my Ph.D. work, but whose name may have been missed.

December 2010, **Bhadade Umesh Shantilal**

CERTIFICATE

This is to certify that the thesis titled, “A General Purpose Lossless Data Compression Scheme with Improved Compression Ratio and Decompression Time and Optimized for Searching and Retrieval of String Randomly from Large Collection” submitted by **BHADADE UMESH SHANTILAL** in fulfillment of the degree of **DOCTOR OF PHILOSOPHY** in the Department of Electrical Engineering, Faculty of Technology & Engineering, The Maharaja Sayajirao University of Baroda, Vadodara is a bonafide record of investigations carried out by him in the Department of Electrical Engineering, Faculty of Technology & Engineering, The Maharaja Sayajirao University of Baroda, Vadodara under my guidance and supervision. In my opinion this work has attained the standard fulfilling the requirements of the Ph.D. Degree as prescribed in the regulations of the University.

Decemeber, 2010

Guide:

Prof. A. I. Trivedi
Department of Electrical Engineering,
Faculty of Technology & Engineering,
The Maharaja Sayajirao University of
Baroda, Vadodara – 390 001

Head:

Prof. S.K. Shah
Department of Electrical Engineering,
Faculty of Technology & Engineering,
The Maharaja Sayajirao University of
Baroda, Vadodara – 390 001

Dean:

Dr. K.B. Pai
Faculty of Technology & Engineering,
The Maharaja Sayajirao University of
Baroda, Vadodara – 390 001

DECLARATION

I, **Bhadade Umesh Shantilal** hereby declare that the work reported in this thesis titled, "A General Purpose Lossless Data Compression Scheme with Improved Compression Ratio and Decompression Time and optimized for Searching and Retrieval of String Randomly from Large Collection" submitted for the award of the degree of **DOCTOR OF PHILOSOPHY** in Department of Electrical Engineering, Faculty of Technology & Engineering, The Maharaja Sayajirao University of Baroda, Vadodara, is original and was carried out in the Department of Electrical Engineering, Faculty of Technology & Engineering, The Maharaja Sayajirao University of Baroda, Vadodara. I further declare that this thesis is not substantially the same as one, which has already been submitted in part or in full for the award of any degree or academic qualification of this University or any other Institution or examining body in India or abroad.

December, 2010 **Bhadade Umesh Shantilal**

Dedicated
To
My Parents
Late Shri. Shantilal Nandlal Bhadade
Smt. Sarla Shantilal Bhadade

ABSTRACT

This thesis involves comprehensive study and implementation of text compression techniques useful for direct searching the phrases in compressed form.

In the initial part of this thesis, we summarize our comprehensive study of different types of compression methods including Arithmetic Coding method, Bzip2, Prediction by Partial Match and Lempel-Ziv Markov-chain Algorithm.

In subsequent part, two categories of text compression techniques are implemented with an objective of improved compression ratio and optimized for searching and retrieval of strings randomly from compressed file.

We implement text compression techniques using three different types of dictionaries viz. static, semi-dynamic and dynamic. We also study the string-matching algorithms such as Karp-Rabin, Knuth-Morris-Pratt, Brute-Force, Boyer-Moore and Quick-Search Algorithms.

Major contribution of the thesis is to propose pre-text compression technique Word based Text Compression Technique using semi-dynamic dictionary (WBTC-C). This method gives a better compression ratio when used as a pre-stage compression to standards methods such as Bzip2, PPMd, PPMII and LZMA, and is also useful for searching the strings directly from the compressed files. The decompression time is also improved in WBTC-C method as compared to Bzip2 and PPMd. Other methods such as CBTC-A, CBTC-B, WBTC-A, WBTC-B, WBTC-D and WBTC-E are also implemented, which differ from WBTC-C. Those methods are implemented using single dimension dictionary, double dimension dictionary and using static dictionary and dynamic dictionary.

The techniques are useful for direct searching the pattern in the compressed form. The text compression techniques implemented by us uses single and double dimension dictionary. The text compression techniques are used as pre-stage compression to existing standard methods such as Arithmetic Coding, Bzip2, PPMd, PPMII and LZMA. The compression ratio is improved when our techniques are used as pre-stage to those methods.

All techniques are implemented in VC++ 6.0 version.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1

INTRODUCTION

1.0 OUTLINE OF THIS CHAPTER

This chapter discusses the origin and the usefulness of the data compression. Compression techniques based on statistical methods, dictionary based, transform based are briefly discussed. It includes the discussion of the problem of pattern searching in compressed form. It also focuses on our contribution to the field of text compression. Finally, the organization of the thesis is given at the end.

1.1 ORIGIN OF DATA COMPRESSION

Giambattista della Porta, a Renaissance scientist, was the author in 1558 of *Magia Naturalis* (Natural Magic), a book in which he discusses many subjects, including demonology, magnetism, and the camera obscura. The book mentions an imaginary device that has since become known as the “sympathetic telegraph”. This device was to have consisted of two circular boxes, similar to compasses, each with a magnetic needle. Each box was to be labeled with the 26 letters, instead of the usual directions, and the main point was that the two needles were supposed be magnetized by the *same lodestone*. Porta assumed that this would somehow coordinate the needles such that when a letter was dialed in one box, the needle in the other box would swing to point to the same letter. Needles to say, such a device does not work (this, after all, was about 300 years before Samuel Morse), but in 1711 a worried wife wrote to the *Spectator*, a London periodical, asking for advice on how to bear the long absences of her beloved husband. The adviser, Joseph Addison, offered some practical ideas, then mentioned Porta’s device, adding that a pair of such boxes might enable her and her husband to communicate with each other even when they “were guarded by spies and watches, or separated by castles and adventures.” Mr. Addison then added that in addition to the 26 letters, the sympathetic telegraph dials should contain, when used by lovers, “several entire words which always have a place in passionate epistles.” The message “I Love You,” for example, would, in such a case, require sending just three symbols instead of ten. This advice is an early example of *text compression* achieved by using short codes for common messages and longer codes for other messages. Even more importantly, this shows how the concept of data compression comes naturally to people who are interested in communications.

1.2 INTRODUCTION OF DATA COMPRESSION

In the modern digital age, information is mostly processed by the machine automatically. Hence the need for compact, precise, and efficient representation of the information is also applicable to the computers. With tremendous amount of information accumulated especially in the last few decades, data compression schemes are playing an increasingly significant role in developing compact representation of information. Moreover, finding the useful information from the mass storage has emerged as another major problem today.

People are good at producing data. In recent times, the growth of textual information via the Internet, digital libraries and archival text data in many applications is unprecedented. The estimation of the growth rate is reflected by the Parkinson's Law on data that "data expands to fill the space available for storage". The TREC **[1]** database holds around 800 million static pages having 6 trillion bytes of plain text equal to the size of a million books. The Google system routinely accumulates millions of pages of new text information every week. The web site Alexa.com is collecting over 1,000GB of information each day from the web and had collected over 35 billion web pages. There have been extensive needs to deal with the overwhelming data.

It is estimated that the memory usage of the computer systems tends to double roughly once every 18 months **[2]**.

Text compression provides a transformed representation of the text data that is understandable only by the computer (in this sense, it relates to cryptography to some extent.) The higher the compression ratio, the less disk space is needed to store the data. The advantage of the idea is twofold. First, we use less space to store the information. For example, English text can be compressed to about 20 to 30% of the original size, and images may be compressed by a factor of several hundred times. Normally, lossless compression must be used for text because we expect the full text to be recovered from the compressed form, unlike audio/video and images which have a much higher degree of redundancy and can be compressed with lossy compression algorithms. Second, we require less bandwidth in the internet transmission compared with transmitting raw data. Obviously, it takes less time to download the text in its compressed form. It will be a considerable saving for the network traffic if the data are transmitted with a much smaller size.

Storage is not the only purpose of keeping the data because we also need to find useful information hidden in the data for different purposes. For example, data mining is a new area catering to the need for extracting the information from the sleeping data. The initial step of mining the knowledge is to retrieve the portion of the text by sending a query, typically using keywords. Then algorithms will be performed on the raw or preprocessed text. Pattern matching is the most popularly used method to search the text using keywords. Although there have been comprehensive studies on text information retrieval **[3, 4, 5]**, not much work has been done on searching directly on compressed text. The compact representation of text is unreadable for human beings. In order to read the data we need to reproduce the original text from the compressed text. Therefore, it is an extra overhead of decompression process rather than mining directly from the original form. Current research on compression shows little consideration for the relationship between the compression algorithm and searching algorithm. We will be focusing on minimizing the overhead by considering the optimal combination of compression and searching schemes. Pattern matching is a typical starting point for knowledge discovering in large databases. There have been various exact and approximate pattern matching algorithms available in the literature. Boyer-Moore (BM) **[6]** and Knuth-Morris-Pratt (KMP) **[7]** pattern matching algorithms are among the best of them. However, pattern matching on compressed text has not been thoroughly explored with the known compression methods. Efficient storage, transmission, searching, and mining the knowledge have become critical and difficult problems to deal with the tremendous data flow. In this thesis, we will deal with the problems related to the lossless text compression and compressed pattern matching.

1.3 DATA COMPRESSION

Data compression is perhaps the fundamental expression of Information Theory. Information Theory is a branch of mathematics that had its genesis in the late 1940s with the work of Claude Shannon at Bell Labs. It concerns itself with various questions about information, including different ways of storing and communicating messages.

Data compression enters into the field of Information Theory because of its concern with redundancy. Redundant information in a message takes extra bits to encode, and if we can get rid of that extra information, we will have reduced the size of the message. Information Theory uses the term Entropy as a measure of how much information is encoded in a message. The word entropy was borrowed from thermodynamics, and it has a similar meaning. The higher the entropy of a message, the more information it contains. The entropy of a symbol is defined as the negative logarithm of its probability. To determine the information content of a message in bits, the entropy is expressed using the base_2 logarithm:

$$\textit{Number of bits} = -\textit{Log base}_2\ \textit{(probability)}$$

The entropy of an entire message is simply the sum of the entropy of all individual symbols. Entropy fits with data compression in its determination of how many bits of information are actually present in a message. If the probability of the character 'e' appearing in this manuscript is 1/16, for example, the information content of the character is four bits. So the character string "eeeee" has a total content of 20 bits. If we are using standard 8-bit ASCII characters to encode this message, we are actually using 40 bits. The difference between the 20 bits of entropy and the 40 bits used to encode the message is where the potential for data compression arises. One important fact to note about entropy is that, unlike the thermodynamic measure of entropy, we can not use an absolute number for the information content of a given message. The problem is that when we calculate Entropy, we use a number that gives us the probability of a given symbol. The probability figure we use is actually the probability for a given model, not an absolute number. If we change the model, the probability will change with it.

How probabilities changes can be seen clearly when using different orders with a statistical model. A statistical model tracks the probability of a symbol based on what symbols appeared previously in the input stream. The order of the model determines how many previous symbols are taken into account. An order-0 model, for example, will not look at previous characters. An order-1 model looks at the one previous character, and so on.

The different order models can yield drastically different probabilities for a character. The letter ‘u’ under an order-0 model, for example, may have only a 1 percent probability of occurrence. But under an order-1 model, if the previous character was ‘q,’ the ‘u’ may have a 95 percent probability. This seemingly unstable notion of a character’s probability proves troublesome for many people. They prefer that a character have a fixed “true” probability which tells them what are the chances of its “really” occurring. Claude Shannon attempted to determine the true information content of the English language with a “party game” experiment. He would uncover a message concealed from his audience a single character at a time. The audience guessed what the next character would be, one guess at a time, until they got it right. Shannon could then determine the entropy of the message as a whole by taking the logarithm of the guess count. Other researchers have done more experiments using similar techniques.

In order to compress data well, it needs to select models that predict symbols with high probabilities. A symbol that has a high probability has low information content and will need fewer bits to encode. Once the model is producing high probabilities, the next step is to encode the symbols using an appropriate number of bits.

Data compression is the process of converting an input data stream (the source stream or the original raw data) into another data stream (the output, or the compressed stream) that has a smaller size. A stream is either a file or buffer in memory.

There are many methods for data compression. They are based on different ideas, are suitable for different types of data, and produce different results, but they are all based on the same principle, namely, they compress the data by removing redundancy from the original data in the source file. Any nonrandom collection data has some structure, and this structure can be exploited to achieve a smaller representation of the data, a representation where no structure is discernible. Thus, redundancy is an important concept in any discussion of data compression. **[8]**

Lossless compression or text compression refers to a class of *reversible* compression algorithms that allow the compressed text to be decompressed into a message identical to the original. They are particularly tailored to use a linear data stream. These properties make text compression applicable to computer programs, which are linear sequences of instructions. Surveys of text compression techniques have been written by Lelewer and Hirschberg **[9]** and Witten et al. **[10]**. Compression algorithms that are not lossless are called lossy. These algorithms are used for compressing data (typically images) that can tolerate some data loss in the decompressed message in exchange for a smaller compressed representation. Since computer programs must be executed without ambiguity, lossy compression is not suitable for them.

1.4 LOSSLESS COMPRESSION ALGORITHMS

No compression algorithm has yet been discovered that consistently attain the predictions of lower bound of data compression **[12]** over wide classes of text files. The goal in the lossless text compression area is to find better algorithms to explore the redundancy of the context and achieve a better compression ratio with a good time complexity. Besides the basic techniques such as Run Length Coding (RLC) and Move-to-Front (MTF), etc. the lossless algorithms can be classified into three broad categories: ***statistical methods, dictionary methods*** **and** ***transform based methods*****.** There are several criteria used to select between using dictionary and statistical compression techniques. Two very important factors are the *decode efficiency* and the overall *compression ratio*. The decode efficiency is a measure of the work required to re-expand a compressed text. The compression ratio is defined by the formula:

compression ratio = compressed size / original size

1.4.1 STATISTICAL METHODS

In the late 1940s, the early years of Information Theory, the idea of developing efficient new coding techniques was just starting to be fleshed out. Researchers were exploring the ideas of entropy, information content, and redundancy. One popular notion held that if the probability of symbols in a message were known, there ought to be a way to code the symbols so that the message would take up less space.

This early work in data compression was being done before the advent of the modern digital computer. Today it seems natural that information theory goes hand in hand with computer programming, but just after World War II, for all practical purposes, there were no digital computers. So the idea of developing algorithms using base 2 arithmetic for coding symbols was really a great leap forward.

The first well-known method for effectively coding symbols is now known as **Shannon-Fano coding**. Claude Shannon at Bell Labs and R.M. Fano at MIT developed this method nearly simultaneously.

In Shannon-Fano coding, the symbols are arranged in order from most probable to least probable, and then divided into two sets whose total probabilities are as close as possible to being equal. All symbols then have the first digits of their codes assigned; symbols in the first set receive "0" and symbols in the second set receive "1". As long as any sets with more than one member remain, the same process is repeated on those sets, to determine successive digits of their codes. When a set has been reduced to one symbol, of course, this means the symbol's code is complete and will not form the prefix of any other symbol's code.

The algorithm works, and it produces fairly efficient variable-length encodings; when the two smaller sets produced by a partitioning are in fact of equal probability, the one bit of information used to distinguish them is used most efficiently. The Shannon-Fano tree is built from the top down, starting by assigning the most significant bits to each code and working down the tree until finished. Shannon-Fano does not always produce optimal prefix codes. For this reason, Shannon-Fano is almost never used.

The other method of statistical coding is **Huffman coding [13]**. Huffman coding shares most characteristics of Shannon-Fano coding. It creates variable length codes that are an integral number of bits. Symbols with higher probabilities get shorter codes. Huffman codes have the unique prefix attribute, which means they can be correctly decoded despite being variable length. Decoding a stream of Huffman codes is generally done by following a binary decoder tree.

Building the Huffman decoding tree is done using a completely different algorithm from that of the Shannon-Fano method. Huffman codes are built from the bottom up, starting with the leaves of the tree and working progressively closer to the root. The procedure for building the tree is simple and elegant. The individual symbols are laid out as a string of leaf nodes that are going to be connected by a binary tree. Each node has a weight, which is simply the frequency or probability of the symbol's appearance. The tree is then built with the following steps:

• The two free nodes with the lowest weights are located.

• A parent node for these two nodes is created. It is assigned a weight equal to the sum of the two child nodes.

• The parent node is added to the list of free nodes, and the two child nodes are removed from the list.

• One of the child nodes is designated as the path taken from the parent node when decoding a 0 bit. The other is arbitrarily set to the 1 bit.

• The previous steps are repeated until only one free node is left. This free node is designated the root of the tree.

The codes have the unique prefix property. Since no code is a prefix to another code, Huffman codes can be unambiguously decoded as they arrive in a stream.

Note, however, that the Huffman codes differ in length from Shannon-Fano codes.

In general, Shannon-Fano and Huffman coding are close in performance. But Huffman coding will always at least equal the efficiency of Shannon-Fano coding, so it has become the predominant coding method of its type. Since both algorithms take a similar amount of processing power, it seems sensible to take the one that gives slightly better performance. And Huffman was able to prove that this coding method cannot be improved on with any other integral bit-width coding stream.

In effect, the tree behaves like a dictionary that has to be transmitted once from the sender to receiver and this constitutes an initial overhead of the algorithm. This overhead is usually ignored in publishing the BPC results for Huffman code in literature. There are also Huffman codes called canonical Huffman codes which uses a look up table or dictionary rather than a binary tree for fast encoding and decoding **[8]**.

Another respectable candidate to replace Huffman coding been successfully demonstrated: **arithmetic coding [15]**. Arithmetic coding bypasses the idea of replacing an input symbol with a specific code. It replaces a stream of input symbols with a single floating-point output number. More bits are needed in the output number for longer, complex messages. The output from an arithmetic coding process is a single number less than 1 and greater than or equal to 0. This single number can be uniquely decoded to create the exact stream of symbols that went into its construction.

Arithmetic code is inherently adaptive, does not use any lookup table or dictionary and in theory can be optimal for a machine with unlimited precision of arithmetic computation. The basic idea can be explained as follows: at the beginning the semi-closed interval [0; 1) is partitioned into $|A|$ equal sized semi-closed intervals under the equiprobability assumption and each symbol is assigned one of these intervals. The first symbol, say a_1 of the message can be represented by a point in the real number interval assigned to it. To encode the next symbol a_2 in the message, the new probabilities of all symbols are calculated recognizing that the first symbol has occurred one extra time and then the interval assigned to a_1 is partitioned (as if it were the entire interval) into $|A|$ sub-intervals in accordance with the new probability distribution. The sequence a_1a_2 can now be represented without ambiguity by any real number in the new sub-interval for a_2. The process can be continued for succeeding symbols in the message as long as the intervals are within the specified arithmetic precision of the computer. The number generated at the final iteration is then a code for the message received so far. The machine returns to its initial state and the process is repeated for the next block of symbol. A simpler version of this algorithm could use the same static distribution of probability at each iteration avoiding re-computation of probabilities. The arithmetic coding method is explained in detail in chapter 2.

The Huffman and arithmetic coders are sometimes referred to as the *entropy coders*. These methods normally use an order (0) model. If a good model with low entropy can be built external to the algorithms, these algorithms can generate the binary codes very efficiently.

One of the most well known modeler is “**Prediction by Partial Match**" (PPM) **[17, 18]**. It is capable of very good compression on a wide variety of source data.

The main idea of PPM (Prediction by Partial Matching) is to take advantage of the previous *k* characters to generate a conditional probability of the current character. The simplest way to do this would be to keep a dictionary for every possible string *s* of *k* characters, and for each string have counts for every character *x* that follows *s*. The conditional probability of *x* in the context *s* is then *C(x|s) / C(s)*, where *C(x/s)* is the number of times *x* follows *s* and *C(s)* is the number of times *s* appears. The probability distributions can then be used by a Huffman or Arithmetic coder to generate a bit sequence. For example, we might have a dictionary with *qu* appearing 100 times and *e* appearing 45 times after *qu*. The conditional probability of the *e* is then .45 and the coder should use about 1 bit to encode it. Note that the probability distribution will change from character to character since each context has its own distribution. In terms of decoding, as long as the context precedes the character being coded, the decoder will know the context and therefore know which probability distribution to use. Because the probabilities tend to be high, arithmetic codes work much better than Huffman codes for this approach.

There are two problems with the basic dictionary method described in the previous paragraph. First, the dictionaries can become very large. There is no solution to this problem other than to keep *k* small, typically 3 or 4. A second problem is what happens if the count is zero. We cannot use zero probabilities in any of the coding methods (they would imply infinitely long strings). One way to get around this is to assume a probability of not having seen a sequence before and evenly distribute this probability among the possible following characters that have not been seen. Unfortunately this gives a completely even distribution, when in reality we might know that *a* is more likely than *b*, even without knowing its context.

The PPM algorithm has a clever way to deal with the case when a context has not been seen before, and is based on the idea of partial matching. The algorithm builds the dictionary on the fly starting with an empty dictionary, and every time the algorithm comes across a string it has not seen before it tries to match a string of one shorter length. This is repeated for shorter and shorter lengths until a match is found. For each length *0,1,. . .,k*. the algorithm keeps statistics of patterns it has seen before and counts of the following characters. In practice this can all be implemented in a single trial. In the case of the length-1 contexts the counts are just counts of each character seen assuming no context. The method is explained in more detail with example in chapter 2.

Dynamic Markov Compression (DMC) **[19]** is another modeling scheme that is equivalent to finite context model but uses finite state machine to estimate the probabilities of the input symbols which are bits rather than bytes as in PPM. The model starts with a single state machine with only one count of `0' and `1' transitions into itself (the zero frequency state) and then the machine adopts to future inputs by accumulating the transitions with 0's and 1's with revised estimates of probabilities. If a state is used heavily for input transitions (caused either by 1 or 0 input), it is cloned into two states by introducing a new state in which some of the transitions are directed and duplicating the output transitions from the original states for the cloned state in the same ratio of 0 and 1 transitions as the original state. The bit-wise encoding takes longer time and therefore DMC is very slow but the implementation is much simpler than PPM and it has been shown that the PPM and DMC models are equivalent **[20]**.

1.4.2 DICTIONARY METHODS

Dictionary decompression uses a codeword as an index into the dictionary table, and then inserts the dictionary entry into the decompressed text stream. If codewords are aligned with machine words, the dictionary lookup is a constant time operation. Statistical compression, on the other hand, uses codewords that have different bit sizes, so they do not align to machine word boundaries. Since codewords are not aligned, the statistical decompression stage must first establish the range of bits comprising a codeword before text expansion can proceed. It can be shown that for every dictionary method there is an equivalent statistical method which achieves equal compression and can be improved upon to give better compression **[11]**. Thus statistical methods can always achieve better compression than dictionary methods albeit at the expense of additional computation requirements for decompression. It should be noted, however, that dictionary compression yields good results in systems with memory and time constraints because one entry expands to several characters. In general, dictionary compression provides for faster (and simpler) decoding, while statistical compression yields a better compression ratio.

A dictionary-based compression scheme uses a different concept. It reads in input data and looks for groups of symbols that appear in a dictionary. If a string match is found, a pointer or index into the dictionary can be output instead of the code for the symbol. The longer the match, the better the compression ratio. This method of encoding changes the focus of dictionary compression. Simple coding methods are generally used, and the focus of the program is on the modeling.

A static dictionary is used like the list of references in an academic paper. Through the text of a paper, the author may simply substitute a number that points to a list of references instead of writing out the full title of a referenced work. The dictionary is static because it is built up and transmitted with the text of work—the reader does not have to build it on the fly. The first time a number is seen in the text like this—[2] — it points to the static dictionary. The problem with a static dictionary is identical to the problem the user of a statistical model faces: The dictionary needs to be transmitted along with the text, resulting in a certain amount of overhead added to the compressed text. An adaptive dictionary scheme helps avoid this problem.

Generally, a type of adaptive dictionary is used when performing acronym replacements in technical literature. The standard way to use this adaptive dictionary is to spell out the acronym, then put its abbreviated substitution in parentheses. So the first time if Maharaja Sayajirao University (MSU) is mentioned then, both the dictionary string and its substitution are defined. From then on, referring to MSU in the text should automatically invoke a mental substitution.

The most widely used compression algorithms (Gzip and Gif) are based on Ziv-Lempel or LZ77 coding **[21]** in which the text prior to the current symbol constitute the dictionary and a greedy search is initiated to determine whether the characters following the current character have already been encountered in the text before, and if yes, they are replaced by a reference giving its relative starting position in the text. Because of the pattern matching operation the encoding takes longer time but the process has been fine tuned with the use of hashing techniques and special data structures. The decoding process is straightforward and fast because it involves a random access of an array to retrieve the character string. A variation of the LZ77 theme, called the LZ78 coding **[22]**, includes one extra character to a previously coded string in the encoding scheme. A more popular variant of LZ78 family is the so-called LZW algorithm **[23]** which leads to widely used *Compress* utility. This method uses a suffix tree to store the strings previously encountered and the text is encoded as a sequence of node numbers in this tree. To encode a string the algorithm will traverse the existing tree as far as possible and a new node is created when the last character in the string fails to traverse a path any more. At this point the last encountered node number is used to compress the string up to that node and a new node is created appending the character that did not lead to a valid path to traverse. In other words, at every step of the process the length of the recognizable strings in the dictionary gets incrementally stretched and is made available to future steps. Many other variants of LZ77 and LZ78 compression family have been reported in the literature. **[8, 16]**.

Kruse and Mukherjee **[24]** devised a dictionary-based scheme called Star encoding. In this method the words are replaced with sequences of * symbols accompanied with references to an external dictionary. The dictionary is arranged according to word lengths, and the proper sub-dictionary is selected by the length of the sequence of "stars". There have been several minor variations of such a scheme from the same authors, most popular of which is a length index preserving transformation (LIPT). In LIPT, the word-length-related sub-dictionary is pointed by a single byte value (as opposed to a sequence of "stars").

Smirnov **[25]** proposed two modifications to LIPT. One is to use non-intersecting alphabet ranges for word lengths, word indices, and letters in words absent from the dictionary. The other idea is more complex: apart from non-intersecting alphabets, also more sub-dictionaries are considered, determined now not only by word lengths but also part-of-speech tags. For an LZ77 compressor, the original LIPT performed best.

StarNT **[26]** is the most recent algorithm from the presented family. A word in StarNT dictionary is a sequence of symbols over the alphabet [a..z]. There is no need to use uppercase letters in the dictionary, as there are two one-byte flags (reserved symbols), f_{cl} and f_{uw}, in the output alphabet to indicate that either a given word starts with a capital letter while the following letters are all lowercase, or a given word consists of capitals only. Another introduced flag, for, prepends an unknown word. Finally, there is yet a collision-handling flag, f_{esc}, used for encoding occurrences of flags f_{cl}, f_{uw}, f_{or}, and f_{esc} in the text.

The ordering of words in the dictionary D, as well as mapping the words to unique codewords, is important for the compression effectiveness. StarNT uses the following rules:

• The most popular words are stored at the beginning of the dictionary. This group has 312 words.

• The remaining words are stored in D according to their increasing lengths. Words of same length are sorted according to their frequency of occurrence in some training corpus.

• Only letters [a..zA..Z] are used to represent the codeword (with the intention to achieve better compression performance with the backend compressor).

Each word in D has assigned a corresponding codeword. Codewords' length varies from one to three bytes. As only the range [a. . z, A . . Z] for codeword bytes is used, there can be up to [52 + (52 x 52) + (52 x 52 x 52)] = 143, 364 entries in the dictionary. The first 52 words have codewords: a, b, . . . , z, A, B, . . . , Z. Words from the 53rd to the 2756th have codewords of length 2: aa, ab, . . . , ZY, ZZ; and so on.

1.4.3 TRANSFORM BASED METHODS

The Burrows-Wheeler transform **[27]** is a block-sorting, lossless data compression algorithm that works by applying a reversible transformation to a block of input data. The transform does not perform any compression but modifies the data in a way to make it easy to compress with a secondary algorithm such as "move-to-front" coding and then Huffman, or arithmetic coding. The BWT algorithm achieves compression performance within a few percent of statistical compressors but at speeds comparable to the LZ based algorithms. The BWT algorithm does not process data sequentially but takes blocks of data as a single unit, which may lend itself to parallel processing. The transformed block contains the same characters as the original block but in a form that is easy to compress by simple algorithms. Same characters in the original block are often grouped together in the transformed block.

Several authors have presented improvements to the original algorithm. Andersson and Nilsson have published several papers about Radix Sort, which can be used as a first sorting step during the BWT **[28, 29]**. In his final BWT research report, Fenwick described some BWT sort improvements including sorting long words instead of single bytes **[30]**. Kurtz presented several papers about BWT sorting stages with suffix trees, which needed less space than other suffix tree implementations and are linear in time **[31,**

32]. Sadakane described a fast suffix array sorting scheme in **[33]** and **[34]**. In **[35]**, Larsson presented an extended suffix array sorting scheme. Based on already sorted suffices, Seward developed two fast suffix sorting algorithms called "copy" and "cache" **[36]**. Itoh and Tanaka presented a fast sorting algorithm called the two stage suffix sort **[37]**. Kao improved the two stage suffix sort by some new techniques which are very fast for sequences of repeat symbols **[38]**. Manzini and Ferragina published some improved suffix array sorting techniques based on the results of Seward and of Itoh and Tanaka **[39]**. Several techniques for the post BWT stages have been also published. Besides the MTF improvements from Schindler **[40]**, and from Balkenhol and Shtarkov **[41]**, an MTF replacement, called Inversion Frequencies, was introduced by Arnavut and Magliveras **[42]** and Deorowicz **[43]** presented another MTF replacement, named Weighted Frequency Count. Various modeling techniques for the entropy coding at the end of the compression process were presented by Fenwick **[44,45]**, Balkenhol and Shtarkov **[46]**. The Burrows-Wheeler-Transform method is explained in detail in Chapter 2.

1.5 SEARCHING IN COMPRESSED FILES

With compressed files becoming more commonplace, the problem of how to search within them is becoming increasingly important **[47]**. There are two options to consider when deciding how to approach compressed pattern matching. The first is a `decompress-then-search' approach, where the compressed file is first decompressed, and then a traditional pattern-matching algorithm applied. This approach has the advantage of simplicity, but brings with it tremendous overheads, in terms of both computation time and storage requirements. Firstly, the entire file must be decompressed – often a lengthy process, especially when considering files several megabytes in size. Additionally, the decompressed file must be stored somewhere once decompressed, so that pattern matching may occur.

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

The main difficulty in compressed-domain pattern matching is that the compression process may have removed a great deal of the structure of the file. The more structure removed, the better the compression likely to be achieved. There is therefore a subtly balanced compromise between obtaining good compression and leaving enough `hints' to allow pattern-matching to proceed. It would appear that these two goals are in constant opposition, but in fact compression is very closely related to pattern matching, in that many compression systems use some sort of pattern matching technique to find repetitions in the input, which can be exploited to give better compression. The effect of this is that these patterns are coded in a special manner, which, if suitably represented, may actually aid in pattern matching. Searching techniques are discussed in detail in chapter 3.

1.6 OUR CONTRIBUTION

We have proposed text compression techniques which are used as a pre-compression stage to well known standard methods such as Arithmetic Coding, Bzip2, PPM variants (PPMd and PPMII), and LZMA. The compression ratio is improved when we use our technique as pre-compression stage. The proposed compression techniques use the concept of static, semi-dynamic and dynamic dictionary, which is arranged in the form of one-dimension and two-dimension. The idea behind using the two-dimension instead of one-dimension is that elements in the two-dimension matrix can be represented in shorter code as compared to elements in one-dimension. Also the number of possible codes reduces in case of two-dimension. The compression techniques proposed here are also suitable for searching the phrase directly in the compressed form, instead of decompress and then search.

1.7 ORGANIZATION OF THESIS

The remaining thesis is organized as given below:

Chapter 2

This chapter discusses compression technique such as arithmetic coding, BWT, PPM variants (PPMd and PPMII) and LZMA in detail.

Chapter 3

This chapter describes various searching algorithms useful for searching the pattern from the file.

Chapter 4

This chapter includes the detailed description of the proposed compression techniques,

Character Based Text Compression method using Static Dictionary (CBTC-A)

Character Based Text Compression Technique using Semi Dynamic Dictionary(CBTC-B)

Word Based Text Compression Technique using Semi Dynamic Dictionary (WBTC-A)

Word Based Text Compression Technique using Semi Dynamic Dictionary (WBTC-B)

Word Based Text Compression Technique using Two-Dimension Semi-Dynamic Dictionary (WBTC-C)

Word Based Text Compression Technique using Dynamic Dictionary (WBTC-D)

Word Based Text Compression Technique using Static Dictionary (WBTC-E)

This chapter includes the process of dictionary creation, and discusses the compression and decompression algorithms along with searching algorithms.

Chapter 5

This chapter describes the implementation of the proposed compression techniques explained in chapter 4. The algorithms are implemented in VC++.

Chapter 6

This chapter includes the comparison of the results of proposed compression techniques. Different set of corpus namely Gutenberg, Enronsent, European Parliament, E-Text, are taken for comparing the results with the existing compression techniques. Also it includes the comparison of searching the string from the compressed file and normal file using different string-matching algorithms described in chapter 3. The experimental results of decompression time of proposed method WBTC-C and Bzip2 is given at the end of this chapter

Chapter 7

This chapter concludes the thesis and discusses about probable future work.

CHAPTER 2

COMPRESSION TECHNIQUES

2

COMPRESSION TECHNIQUES

2.0 OUTLINE OF THIS CHAPTER

This chapter describes various methods of compression techniques. Generating variable-length codes: arithmetic coding, followed by transform based method BWT which is used by BZIP2. The context based method Prediction by Partial Match (PPM) and its variants are described in section 2.4 followed by in improved version of LZ77 method – Lempel Ziv Markov-chain Algorithm (LZMA) in section 2.5

2.1 ORIGINS OF ARITHMETIC CODING

The first step toward arithmetic coding was taken by Shannon **[48],** who observed in a 1948 paper that messages ***N*** symbols long could be encoded by first sorting the messages in order of their probabilities and then sending the cumulative probability of the preceding messages in the ordering. The code string was a binary fraction and was decoded by magnitude comparison. The next step was taken by Peter Elias in an unpublished result; Abramson [**49]** described Elias' improvement in **1963.** Elias observed that Shannon's scheme worked without sorting the messages, and that the cumulative probability of a message of ***N*** symbols could be recursively calculated from individual symbol probabilities and the cumulative probability of the message of ***N*** - 1 symbols. Elias' code was studied by Jelinek **[50]**. The codes of Shannon and Elias suffered from a serious problem: As the message increased in length the arithmetic involved required *increasing precision.* By using fixed-width arithmetic units for these codes, the time to encode each symbol is increased linearly with the length of the code string.

Meanwhile, another approach to coding was having a similar problem with precision. In 1972, Schalkwijk **[51]** studied coding from the standpoint of providing an index to the encoded string within a set of, possible strings. A**s** symbols were added to the string, the index increased in size. This is a *last-in-first-out* (LIFO) code, because the last symbol encoded was the first symbol decoded. Cover **[52]** made improvements to this scheme, which is now called *enumerative coding.* These codes suffered from the same precision problem.

Both Shannon's code and the Schalkwijk-Cover code can be viewed as a mapping of strings to a number, forming two branches of pre-arithmetic codes, called FIFO *(first-in-first-out)* and LIFO. Both branches use a double recursion, and both have a precision problem. Rissanen **[53]** alleviated the precision problem by suitable approximations in designing a LIFO arithmetic code. Code strings of any length could be generated with a fixed calculation time per data symbol using fixed-precision arithmetic.

Pasco **[54]** discovered a FIFO arithmetic code, discussed earlier, which controlled the precision problem by essentially the same idea proposed by Rissanen. In Pasco's work, the code string was kept in computer memory until the last symbol was encoded. This strategy allowed a carry-over to be propagated over a long carry chain. Pasco **[54]** also conjectured on the family of arithmetic codes based on their mechanization.

In Rissanen **[53]** and Pasco **[54]**, the original (given, or presumed) symbol probabilities were used. (In practice, we use estimates of the relative frequencies. However, the notion of an imaginary "source" emitting symbols according to given probabilities is commonly found in the coding literature.) In **[15]** and **[55]**, Rissanen and Langdon introduced the notion of coding parameters "based" on the symbol probabilities. The uncoupling of the coding parameters from the symbol probabilities simplifies the implementation of the code at very little compression loss, and gives the code designer some tradeoff possibilities. In **[15]** it was stated that there were ways to block the carry-over, and in **[55]** bit-stuffing was presented. In **[56]** F. Rubin also improved Pasco's code by preventing carry-overs. The result was called a "stream" code. Jones **[57]** and Martin **[58]** have independently discovered P-based FIFO arithmetic codes.

Rissanen and Langdon **[15]** successfully generalized and characterized the family of arithmetic codes through the notion of the decodability criterion which applies to all such codes, be they LIFO or FIFO, L-based or P-based. The arithmetic coding family is seen to be a practical generalization of many pre-arithmetic coding algorithms, including Elias' code, Schalkwijk **[51]**, and Cover **[52]**. In **[59]**, Rissanen presents an interesting view of an arithmetic code as a number-representation system, and shows that Elias' code and enumerative codes are duals.

2.2 ARITHMETIC CODING

In comparison to the well-known Huffman Coding algorithm, Arithmetic Coding overcomes the constraint that the symbol to be encoded has to be coded by a whole number of bits. This leads to higher efficiency and a better compression ratio in general. Indeed Arithmetic Coding can be proven to almost reach the best compression ratio possible, which is bounded by the entropy of the data being encoded. Though during encoding the algorithm generates one code for the whole input stream, this is done in a fully sequential manner, symbol after symbol.

Compared to other fields of Computer Science, Arithmetic Coding is still very young, however already mature and efficient principle for lossless data encoding, which satisfies all the requirements of what people understand of a modern compression algorithm: Data input streams can be compressed symbol wise, enabling on-the-fly data compression. Also Arithmetic Coding works in linear time with only constant use of memory. As mentioned above, finite precision integer arithmetic suffices for all calculations. These and other properties make it straightforward to derive hardware-based solutions. Arithmetic Coding is also known to reach a best-possible compression ratio, provided the single symbols of the input stream are statistically independent, which should be the case for most data streams. Also it can be enhanced very simple by allowing simple plug-in of optimized statistical models. The decoder uses almost the same source code as the encoder which also makes the implementation straightforward.

2.2.1 EXAMPLE OF ARITHMETIC CODING

Arithmetic coding bypasses the idea of replacing an input symbol with a specific code. It replaces a stream of input symbols with a single floating-point output number. More bits are needed in the output number for longer, complex messages. This concept has been known for some time, but only recently were practical methods found to implement arithmetic coding on computers with fixed sized registers.

The output from an arithmetic coding process is a single number less than 1 and greater than or equal to 0. This single number can be uniquely decoded to create the exact stream of symbols that went into its construction. To construct the output number, the symbols are assigned set probabilities. The message “BILL GATES,” for example, would have a probability distribution as shown in Table 2.1

Table 2.1 Probability Distribution of message “BILL GATES”

Character	**Probability**
SPACE	**1/10**
A	**1/10**
B	**1/10**
E	**1/10**
G	**1/10**
I	**1/10**
L	**2/10**
S	**1/10**
T	**1/10**

Once character probabilities are known, individual symbols need to be assigned a range along a “probability line,” nominally 0 to 1. It doesn’t matter which characters are assigned which segment of the range, as long as it is done in the same manner by both the encoder and the decoder. The nine-character symbol set used here would look like as shown in Table 2.2.

Table 2.2 Probability range of message “BILL GATES”

Character	**Probability**	**Range**
SPACE	1/10	0.00 - 0.10
A	1/10	0.10 - 0.20
B	1/10	0.20 - 0.30
E	1/10	0.30 - 0.40
G	1/10	0.40 - 0.50
I	1/10	0.50 - 0.60
L	2/10	0.60 - 0.80
S	1/10	0.80 - 0.90
T	1/10	0.90 - 1.00

Each character is assigned the portion of the 0 to 1 range that corresponds to its probability of appearance. The character “owns” everything up to, but not including, the higher number. So the letter T in fact has the range .90 to .9999… The most significant portion of an arithmetic-coded message belongs to the first symbols—or B, in the message “BILL GATES.” To decode the first character properly, the final coded message has to be a number greater than or equal to .20 and less than .30. To encode this number, track the range it could fall in. After the first character is encoded, the low end for this range is .20 and the high end is .30. During the rest of the encoding process, each new symbol will further restrict the possible range of the output number. The next character to be encoded, the letter I, owns the range .50 to .60 in the new subrange of .2 to .3. So the new encoded number will fall somewhere in the 50th to 60th percentile of the currently established range. Applying this logic will further restrict the number to .25 to .26. The algorithm to accomplish this for a message of any length is shown in figure 2.1

```
low = 0.0;
high = 1.0;
while ( ( c = getc( input ) ) != EOF ) {
range = high - low;
high = low + range * high_range( c );
low = low + range * low_range( c );
}
output ( low );
```

Figure 2.1 Algorithm for encoding symbols (Arithmetic Coding)

Following this process to its natural conclusion with message results in the following table 2.3.

So the final low value, .2572167752, will uniquely encode the message “BILL GATES” using coding scheme explained above.

Given this encoding scheme, it is relatively easy to see how the decoding process operates. Find the first symbol in the message by seeing which symbol owns the space our encoded message falls in. Since .2572167752 falls between .2 and .3, the first character must be B. Then remove B from the encoded number. Since the low and high range of B is known, remove their effects by reversing the process that put them in. First, subtract the low value of B, giving .0572167752. Then divide by the width of the range of B, or .1. This gives a value of .572167752. Then calculate where that lands, which is in the range of the next letter, I. The algorithm for decoding the incoming number is shown figure 2.2

Table 2.3 Encoded values of Characters

New Character	Low Value	High Value
	0.0	1.0
B	0.2	0.3
I	0.25	0.26
L	0.256	0.258
L	0.2572	0.2576
SPACE	0.25720	0.25724
G	0.257216	0.257220
A	0.2572164	0.2572168
T	0.25721676	0.2572168
E	0.257216772	0.257216776
S	0.2572167752	0.2572167756

```
number = input_code();
for ( ; ; ) {
symbol = find_symbol_straddling_this_range( number );
putc( symbol );
range = high_range( symbol ) - low_range( symbol );
number = number - low_range( symbol );
number = number / range;
}
```

Figure 2.2 Algorithm for decoding symbols (Arithmetic Coding)

The problem of how to decide when there are no more symbols left to decode can be handled either by encoding a special end-of-file symbol or by carrying the stream length with the encoded message. The decoding algorithm for the “BILL GATES” message will proceed as shown in Table 2.4

In summary, the encoding process is simply one of narrowing the range of possible numbers with every new symbol. The new range is proportional to the predefined probability attached to that symbol. Decoding is the inverse procedure, in which the range is expanded in proportion to the probability of each symbol as it is extracted.

Table 2.4 Decoded values of characters

Encoded Number	Output Symbol	Low	High	Range
0.2572167752	B	0.2	0.3	0.1
0.572167752	I	0.5	0.6	0.1
0.72167752	L	0.6	0.8	0.2
0.6083876	L	0.6	0.8	0.2
0.041938	SPACE	0.0	0.1	0.1
0.41938	G	0.4	0.5	0.1
0.1938	A	0.1	0.2	0.1
0.938	T	0.9	1.0	0.1
0.38	E	0.3	0.4	0.1
0.8	S	0.8	0.9	0.1
0.0				

2.2.2 PRACTICAL MATTERS

Encoding and decoding a stream of symbols using arithmetic coding is not too complicated. But at first glance it seems completely impractical. Most computers support floating-point numbers of around 80 bits. So is it necessary to start over every time you encode ten or fifteen symbols? Whether floating-point processor is needed? Can machines with different floating-point formats communicate through arithmetic coding?

As it turns out, arithmetic coding is best accomplished using standard 16-bit and 32-bit integer math. Floating-point math is neither required nor helpful. What is required is an incremental transmission scheme in which fixed-size integer state variables receive new bits at the low end and shift them out at the high end, forming a single number that can be as long as necessary, conceivably millions or billions of bits.

Earlier, it has been shown that the algorithm works by keeping track of a high and low number that brackets the range of the possible output number. When the algorithm first starts, the low number is set to 0 and the high number is set to 1. The first simplification made to work with integer math is to change the 1 to .999 …, or .111… in binary.

Mathematicians agree that .111… binary is exactly the same as 1 binary, and this assurance is taken at face value. It simplifies encoding and decoding. To store these numbers in integer registers, first justify them so the implied decimal point is on the left side of the word. Then load as much of the initial high and low values as will fit into the word size we are working with. If the implementation is done using 16-bit unsigned math, then initial value of high will be 0xFFFF, and low will be 0. It is known that the high value continues with Fs forever, and the low continues with zeros forever; so those extra bits can be shifted in with impunity when needed.

Consider the example of the message "BILL GATES" in a five-decimal digit register, the decimal equivalent would look like as shown below:

HIGH: 99999 implied digits => 999999999...
LOW: 00000 implied digits => 000000000...

To find the new range of numbers, apply the encoding algorithm shown in figure 2.1. First, calculate the range between the low and high values. The difference between the two registers will be 100000, not 99999. This is because the high register has an infinite number of 9s added to it, so it is needed to increment the calculated difference. Then compute the new high value using the formula

high = low + high_range(symbol)

In this case, the high range was .30, which gives a new value for high of 30000. Before storing the new value of high, it is needed to decrement it, once again because of the implied digits appended to the integer value. So the new value of high is 29999. The calculation of low follows the same procedure, with a resulting new value of 20000. So now high and low look like this:

high: 29999 (999...)
low: 20000 (000...)

At this point, the most significant digits of high and low match. Due to the nature of algorithm, high and low can continue to grow closer together without quite ever matching.

So once they match in the most significant digit, that digit will never change. That digit can be now output as the first digit of the encoded number. This is done by shifting both high and low left by one digit and shifting in a 9 in the least significant digit of high. As this process continues, high and low continually grow closer together, shifting digits out into the coded word. The process for message “BILL GATES” is shown in Table 2.5

After all the letters are accounted for, two extra digits need to be shifted out of either the high or low value to finish the output word. This is so the decoder can properly track the input data. Part of the information about the data stream is still in the high and low registers, and we need to shift that information to the file for the decoder to use later.

2.2.3 COMPLICATION IN ARITHMETIC CODING

This scheme works well for incrementally encoding a message. Enough accuracy is retained during the double-precision integer calculations to ensure that the message is accurately encoded. But there is potential for a loss of precision under certain circumstances. If the encoded word has a string of 0s or 9s in it, the high and low values will slowly converge on a value, but they may not see their most significant digits match immediately. High may be 700004, and low may be 699995. At this point, the calculated range will be only a single digit long, which means the output word will not have enough precision to be accurately encoded. Worse, after a few more iterations, high could be 70000, and low could be 69999. At this point, the values are permanently stuck. The range between high and low has become so small that any iteration through another symbol will leave high and low at their same values. But since the most significant digits of both words are not equal, the algorithm can’t output the digit and shift. It seems to have reached an impasse.

This underflow problem can be solved by preventing things from ever getting bad. The original algorithm said something like, “If the most significant digit of high and low match, shift it out.” If the two digits don’t match, but are now on adjacent numbers, a second test needs to be applied. If high and low are one apart, then the second most

Table 2.5 Cumulative output of message

	High	**Low**	**Range**	**Cumulative Output**
Initial State	99999	00000	100000	
Encode B (0.2 – 0.3)	29999	20000		
Shift out 2	99999	00000	10000	.2
Encode I (0.5 – 0.6)	59999	50000		.2
Shift out 5	99999	00000	10000	.25
Encode L (0.6 – 0.8)	79999	60000	20000	.25
Encode L (0.6 – 0.8)	75999	72000		.25
Shift out 7	59999	20000	40000	.257
Encode SPACE (0.0 – 0.1)	23999	20000		0.257
Shift out 2	39999	00000	40000	.2572
Encode G (0.4 – 0.5)	19999	16000		.2572
Shift out 1	99999	60000	40000	.25721
Encode A (0.1 – 0.2)	67999	64000		.25721
Shift out 6	79999	40000	40000	.257216
Encode T (0.9 – 1.0)	79999	76000		.257216
Shift out 7	99999	60000	40000	.2572167
Encode E (0.3 – 0.4)	75999	72000		.2572167
Shift out 7	59999	20000	40000	.25721677
Encode S (0.8 – 0.9)	55999	52000		.25721677
Shift out 5	59999	20000		.257216775
Shift out 2				.2572167752
Shift out 0				.25721677520

significant digit in high is tested for 0 and the second digit in low is tested for 0. If so, it means that underflow problem has occurred and an action is needed. Head off underflow with a slightly different shift operation. Instead of shifting the most significant digit out of the word, delete the second digits from high and low and shift the rest of the digits left to fill the space. The most significant digit stays in place. Then set an underflow counter to remember that a digit is threw away and it is not quite sure whether it was going to be a 0 or a 9. This process is shown in Table 2.6.

Table 2.6 Underflow Situation

	Before	After
High:	40344	43449
Low:	39810	38100
Underflow:	0	1

After every recalculation, check for underflow digits again if the most significant digit doesn't match. If underflow digits are present, shift them out and increment the counter. When the most significant digits do finally converge to a single value, output that value. Then output the underflow digits previously discarded. The underflow digits will all be 9s or 0s, depending on whether high and low converged on the higher or lower value.

2.2.4 DECODING

In the "ideal" decoding process, the entire input number is to be work with, the entire number is available to work with, and the algorithm had to do things like "divide the encoded number by the symbol probability." In practice, it is not possible to perform an operation like that on a number that could be billions of bytes long. As in the encoding process, however, the decoder can operate using 16- and 32-bit integers for calculations. Instead of using just two numbers, high and low, the decoder has to use three numbers. The first two, high and low, correspond exactly to the high and low values maintained by the encoder. The third number, code, contains the current bits being read in from the input bit stream. The code value always falls between the high and low values. As they come closer and closer to it, new shift operations will take place, and high and low will move back away from code.

The high and low values in the decoder will be updated after every symbol, just as they were in the encoder, and they should have exactly the same values. By performing the same comparison test on the upper digit of high and low, the decoder knows when it is time to shift a new digit into the incoming code. The same underflow tests are performed as well.

In the ideal algorithm, it was possible to determine what the current encoded symbol was just by finding the symbol whose probabilities enclosed the present value of the code. In the integer math algorithm, things are somewhat more complicated. In this case, the probability scale is determined by the difference between high and low. So instead of the range being between .0 and 1.0, the range will be between two positive 16-bit integer counts. Where the present code value falls along that range determines current probability. Divide (value - low) by (high - low + 1) to get the actual probability for the present symbol.

2.2.5 COMPARISON WITH HUFFMAN CODING

It is not immediately obvious why this encoding process is an improvement over Huffman coding. It becomes clear when we examine a case in which the probabilities are a little different. If we have to encode the stream “AAAAAAA,” and the probability of A is known to be .9, there is a 90 percent chance that any incoming character will be the letter A. The Probability table is setup so that A occupies the 0.0 to 0.9 range, and the end of-message symbol occupies the 0.9 to 1.0 range. The encoding process is shown in Table 2.7.

Table 2.7 Encoding process of message “AAAAAAA”

New Character	Low Value	High Value
	0.0	0.1
A	0.0	0.9
A	0.0	0.81
A	0.0	0.729
A	0.0	0.6561
A	0.0	0.59049
A	0.0	0.531441
A	0.0	0.4782969
END	0.43046721	0.4782969

Now that the range of high and low values is known, all that remains is to pick a number to encode this message. The number .45 will make this message uniquely decode to

"AAAAAAA." Those two decimal digits take slightly less than seven bits to specify, which means that eight symbols are encoded in less than eight bits. An optimal Huffman message would have taken a minimum of nine bits.

To take this point to an even further extreme, consider a example that had only two symbols. In it, 0 had a 16382/16383 probability, and an end-of-file symbol had a 1/16383 probability. Create a file filled with 100,000 0s. After compression using arithmetic coding, the output file was only three bytes long! The minimum size using Huffman coding would have been 12,501 bytes. This is obviously a contrived example, but it shows that arithmetic coding compresses data at rates much better than one bit per byte when the symbol probabilities are right.

2.3. BURROWS WHEELER TRANSFORMATION

The BWT algorithm does not process its input sequentially, but instead processes a block of text as a single unit. The idea is to apply a reversible transformation to a block of text to form a new block that contains the same characters, but is easier to compress by simple compression algorithms. The transformation tends to group characters together so that the probability of finding a character close to another instance of the same character is increased substantially. Text of this kind can easily be compressed with fast locally-adaptive algorithms, such as move-to-front coding **[60]** in combination with Huffman or arithmetic coding.

Briefly, the algorithm transforms a string S of N characters by forming the N rotations (cyclic shifts) of S, sorting them lexicographically, and extracting the last character of each of the rotations. A string L is formed from these characters, where the ith character of L is the last character of the ith sorted rotation. In addition to L, the algorithm computes the index I of the original string S in the sorted list of rotations. Surprisingly, there is an efficient algorithm to compute the original string S given only L and I .

The sorting operation brings together rotations with the same initial characters. Since the initial characters of the rotations are adjacent to the final characters, consecutive characters in L are adjacent to similar strings in S. If the context of a character is a good predictor for the character, L will be easy to compress with a simple locally-adaptive compression algorithm.

2.3.1 THE REVERSIBLE TRANSFORMATION

Two sub-algorithms are described here. Algorithm 2.3.1 performs the reversible transformation that is applied to a block of text before compressing it, and Algorithm 2.3.2 performs the inverse operation. In the description below, strings is treated as vectors whose elements are characters.

ALGORITHM 2.3.1: COMPRESSION TRANSFORMATION

This algorithm takes as input a string *S* of *N* characters *S*[0],,,,,,, *S*[*N*-1] selected from an ordered alphabet *X* of characters. To illustrate the technique, consider a example, using the string *S* = ‘abraca’, *N* = 6, and the alphabet *X* = {‘a’,’b’,’c’,’r’}

Step 1: Sort rotations

Form a conceptual *N* x *N* matrix *M* whose elements are characters, and whose rows are the rotations (cyclic shifts) of *S*, sorted in lexicographical order. At least one of the rows of *M* contains the original string *S*. Let *I* be the index of the first such row, numbering from zero. In this example, the index *I* = 1 and the matrix *M* is row

0	*aabrac*
1	*abraca*
2	*acaabr*
3	*bracaa*
4	*caabra*
5	*racaab*

Step 2: Find last characters of rotations

Let the string *L* be the last column of *M*, with characters *L*[0],,,,,,*L*[*N*-1] (equal to *M*[0, *N* - 1], , , , ,M[*N* – 1, *N* - 1]). The output of the transformation is the pair (*L*, *I*). In this example, *L* = ‘*caraab*’ and *I* = 1 (from step C1).

ALGORITHM 2.3.2: DECOMPRESSION TRANSFORMATION

The same example and notation used in Algorithm 2.3.1 is considered here. Algorithm 2.3.2 uses the output (L, I) of Algorithm 2.3.1 to reconstruct its input, the string *S* of length *N*.

Step 1: Find first characters of rotations

This step calculates the first column *F* of the matrix *M* of Algorithm 2.3.1. This is done by sorting the characters of *L* to form *F*. It is observed that any column of the matrix *M* is a permutation of the original string *S*. Thus, *L* and *F* are both permutations of *S*, and therefore of one another. Furthermore, because the rows of *M* are sorted, and *F* is the first column of *M*, the characters in *F* are also sorted.

In this example, *F* = ‘*aaabcr*’.

Step 2: Build list of predecessor characters

To explain in detail, this step is described in terms of the contents of the matrix *M*. It should be remember that the complete matrix is not available to the decompressor; only the strings *F*, *L*, and the index *I* (from the input) are needed by this step.

Consider the rows of the matrix *M* that start with some given character *ch*. Algorithm C ensured that the rows of matrix *M* are sorted lexicographically, so the rows that start with *ch* are ordered lexicographically.

Let us define the matrix *M’* formed by rotating each row of *M* one character to the right, so for each *i* = 0, , , , ,*N* - 1, and each *j* = 0, , , , ,*N* -1,

$$M' [i; j] = M[i, (j-1) \bmod N]$$

In this example, *M* and *M’* are:

row	*M*	*M’*
0	*aabrac*	*caabra*
1	*abraca*	*aabrac*
2	*acaabr*	*racaab*
3	*bracaa*	*abraca*
4	*caabra*	*acaabr*
5	*racaab*	*bracaa*

Like *M*, each row of *M*' is a rotation of *S* and for each row of *M* there is a corresponding row in *M*'. *M*' is constructed from M, so that the rows of *M*' are sorted lexicographically starting with their *second* character. So, if only those rows in *M*' are considered that start with a character *ch*, they must appear in lexicographical order relative to one another; they have been sorted lexicographically starting with their second characters, and their first characters are all the same and so do not affect the sort order. Therefore, for any given character *ch*, the rows in *M* that begin with *ch* appear in the *same order* as the rows in *M*' that begin with *ch*.

In this example, this is demonstrated by the rows that begin with 'a'. The rows 'aabrac', 'abraca', and 'acaabr' are rows 0, 1, 2 in *M* and correspond to rows 1, 3, 4 in *M*'.

Using *F* and *L*, the first columns of *M* and *M*' respectively, a vector T is calculated that indicates the correspondence between the rows of the two matrices, in the sense that for each $j = 0, , , , , ,N\text{-}1$, row *j* of *M*' corresponds to row $T[j]$ of *M*.

If $L[j]$ is the *k*th instance of *ch* in *L*, then $T[j] = i$ where $F[i]$ is the *k*th instance of *ch* in *F*. Note that *T* represents a one-to-one correspondence between elements of *F* and elements of *L*, and $F[T[j]] = L[j]$.

In this example, *T* is: (4 0 5 1 2 3).

Step 3: Form output *S*

Now, for each $i = 0, , , , , N - 1$, the characters $L[i]$ and $F[i]$ are the last and first characters of the row *i* of *M*. Since each row is a rotation of *S*, the character $L[i]$ cyclicly precedes the character $F[i]$ in *S*. From the construction of *T*, we have $F[T[j]] = L[j]$. Substituting $i = T[j]$, we have $L[T[j]]$ cyclicly precedes $L[j]$ in *S*.

The index *I* is defined by Algorithm 2.3.1 such that row *I* of *M* is *S*. Thus, the last character of *S* is $L[I]$. The vector *T* is used to give the predecessors of each character:

for each $i = 0, , , , , , N - 1$: $S[N - 1 - i] = L[T^i[I]]$.

where $T^0[x] = x$, and $T^{i+1}[x] = T[T^i[x]]$. This yields *S*, the original input to the compressor.

In this example, *S* = '*abraca*'.

Let us define *T* such that the string *S* would be generated from front to back, rather than the other way around.

The sequence $T^i[I]$ for $i = 0, , , , , N - 1$ is not necessarily a permutation of the numbers 0, , , , , $N - 1$. If the original string *S* is of the form Z^p for some substring *Z* and some $p > 1$, then the sequence *Ti* [*I*] for *i* = 0, , , , , *N* - 1 will also be of the form Z'^{p} for some subsequence *Z'*. That is, the repetitions in *S* will be generated by visiting the same elements of *T* repeatedly. For example, if *S* = 'cancan', *Z* = 'can' and *p* = 2, the sequence *Ti* [*I*] for *i* = 0, , , , , *N* - 1 will be [2, 4, 0, 2, 4, 0].

2.3.2 WHY THE TRANSFORMED STRING COMPRESSES WELL

Algorithm 2.3.1 sorts the rotations of an input string *S*, and generates the string *L* consisting of the last character of each rotation.

To see why this might lead to effective compression, consider the effect on a single letter in a common word in a block of English text. Consider the example of the letter 't' in the word 'the', and assume an input string containing many instances of 'the'.

When the list of rotations of the input is sorted, all the rotations starting with 'he ' will sort together; a large proportion of them are likely to end in 't'. One region of the string *L* will therefore contain a disproportionately large number of 't' characters, intermingled with other characters that can proceed 'he ' in English, such as space, 's', 'T', and 'S'.

The same argument can be applied to all characters in all words, so any localized region of the string *L* is likely to contain a large number of a few distinct characters. The overall effect is that the probability that given character *ch* will occur at a given point in *L* is very high if *ch* occurs near that point in *L*, and is low otherwise. This property is exactly the one needed for effective compression by a move-to-front coder, which encodes an instance of character *ch* by the count of distinct characters seen since the next previous occurrence of *ch*. When applied to the string *L*, the output of a move-to-front coder will be dominated by low numbers, which can be efficiently encoded with a Huffman or arithmetic coder.

For completeness, one of the possible ways is to use Move-to-Front coding technique to encode the output of Algorithm 2.3.1 and the corresponding inverse operation. A complete compression algorithm is created by combining these encoding and decoding operations with Algorithms 2.3.1 and 2.3.2.

2.3.3 MOVE-TO-FRONT CODING

This is a technique that is ideal for sequences with the property that the occurrence of a character indicates it is more likely to occur immediately afterwards. The sequence of characters is converted to a list of numbers as follows: The list of characters maintained, represent characters by their position in the list. On encoding a character, it is moved to the front of the list. Thus smaller numbers are more likely to occur than larger numbers.

ALGORITHM 2.3.3: MOVE-TO-FRONT CODING

This algorithm encodes the output (*L, I*) of Algorithm C, where *L* is a string of length *N* and *I* is an index. It encodes *L* using a move-to-front algorithm and a Huffman or arithmetic coder.

The example used in Algorithm 2.3.1 is continued here.

Step 1: Move-to-front coding

This step encodes each of the characters in *L* by applying the move-to-front technique to the individual characters. Let us define a vector of integers *R*[0], , , , , *R*[*N*-1], which are the codes for the characters *L*[0], , , , , *L*[*N*- 1].

Initialize a list *Y* of characters to contain each character in the alphabet *X* exactly once.

For each *i* = 0, , , , , ,N-1 in turn, set *R*[*i*] to the number of characters preceding character *L*[*i*] in the list *Y*, then move character *L*[*i*] to the front of *Y*.

Taking *Y* = [‘a’,’b’,’c’,’r’] initially, and *L* = ‘caraab’, compute the vector *R*: (2 1 3 1 0 3).

Step 2: Encode

Apply Huffman or arithmetic coding to the elements of *R*, treating each element as a separate token to be coded. Any coding technique can be applied as long as the decompressor can perform the inverse operation. Call the output of this coding process *OUT*. The output of Algorithm 2.3.1 is the pair (OUT ,I) where *I* is the value computed in step 1 of algorithm 2.3.1.

ALGORITHM 2.3.4: MOVE-TO-FRONT DECODING

This algorithm is the inverse of Algorithm 2.3.3. It computes the pair (L,I) from the pair . (OUT ,I).

Here it is assumed that the initial value of the list *Y* used in step 1 of algorithm 2.3.3 is available to the decompressor, and that the coding scheme used in step 2 of algorithm 2.3.3 has an inverse operation.

Step 1: Decode

Decode the coded stream *OUT* using the inverse of the coding scheme used in step 2 of algorithm 2.3.3. The result is a vector *R* of *N* integers.

In our example, *R* is: (2 1 3 1 0 3).

Step 2: Inverse move-to-front coding

The goal of this step is to calculate a string *L* of *N* characters, given the move-to-front codes *R*[0], , , , ,*R*[*N* -1].

Initialize a list *Y* of characters to contain the characters of the alphabet *X* in the same order as in step 1 of Algorithm 2.3.3.

For each *i* = 0, , , , , *N* - 1 in turn, set *L*[*i*] to be the character at position *R*[*i*] in list *Y* (numbering from 0), then move that character to the front of *Y*. The resulting string *L* is the last column of matrix *M* of Algorithm 2.3.1. The output of this algorithm is the pair (L,I), which is the input to Algorithm 2.3.2.

Taking Y = [‘a’,’b’,’c’,’r’] initially (as in Algorithm M), we compute the string *L* = ‘*caraab*’.

2.4. PREDICTION BY PARTIAL MATCH

The best known context-based algorithm is the *ppm* algorithm, first proposed by Cleary and Witten in 1984 [17]. It has not been popular as the various Ziv-Lempel based algorithms mainly because of the faster execution speeds of the latter algorithms. Lately with the development of more efficient variants, *ppm*-based algorithms are becoming increasingly more popular.

The idea of the *ppm* algorithm is elegantly simple. We would like to use large contexts to determine the probability of the symbol being encoded.

The basic algorithm initially attempts to use the largest context. The size of the largest context is predetermined. If the symbol to be encoded has not previously been encountered in this context, an escape symbol is encoded and the algorithm attempts to use the next smaller context. If the symbol has not occurred in this context earlier, the size of the context is further reduced. This process, continues until either we obtain a context that has previously been encountered with this symbol, or we arrive at the conclusion that the symbol has not been encountered previously in any context. In this case, we use a probability of *1/X* to encode the symbol, where *X* is the size of the source alphabet. For example, when coding the *a* of *probability*, we would first attempt to see if the string *proba* has previously occurred – that is, if *a* had previously occurred in the context of *prob*. If not, we would encode an escape and see if *a* had occurred in the context of *rob*. If the string *roba* had not occurred previously, we would again send an escape symbol and try the context *ob*. Continuing in this manner, we would try the context *b*, and failing that we would see if the letter *a* (with zero-order context) had occurred previously. If *a* was being encountered for the first time, we would use a model in which all letters occur with equal probability to encode *a*. This equiprobable model is sometimes referred to as the context of order -1.

As the development of the probabilities with respect to each context is an adaptive process each time a symbol is encountered, the count corresponding to that symbol is updated. The number of counts to be assigned to the escape symbol is not obvious, and a number of different approaches have been used. One approach used by Cleary and Witten is to give the escape symbol a count of one, thus inflating the total count by one. Cleary and Witten call this method – A, and the resulting algorithm *ppma*.

Example

Lets encode the sequence

thisƀisƀtheƀtithe

Assuming we have already encoded the initial seven characters *thisƀis*, the various counts and Cum_Counts arrays to be used in the arithmetic coding of the symbols are shown in Tables 2.8 – 2.11. In this example, we are assuming that the longest context length is two. This is a rather small value and is used here to keep the size of the example reasonable small. A more common value for the longest context length is five.

We will assume that the word length for arithmetic coding is four. Thus, l = 0000 and u = 1111. As *this~~b~~is*, has already been encoded, the next letter to be encoded is ~~b~~. The second order context for this letter is *is*. Looking at Table 2.11, we can see that the letter ~~b~~ is the first letter in this context with a *Cum_Count* value of 1. As the *Total_Count* in this case is 2, the update equations for the lower and upper limits are

l = 0 + [(15-0+1) * 0/2] = 0 = 0000

u - 0 +[(15 – 0 + 1) * 1/2] – 1 = 7 = 0111.

As the MSBs of both l and u are the same, we shift that bit our, shift a 0 into the LSB of l, and a 1 into the LSB of u. The transmitted sequence, lower limit, and upper limit after the update are:

Transmitted sequence: 0

l:0000

u:1111

Table 2.8 Count array for -1 order context

Letter	Count	*Cum_Count*
t	1	1
h	1	2
i	1	3
s	1	4
e	1	5
~~b~~	1	6
Total Count		6

Table 2.9 Count array for zero-order context.

Letter	Count	*Cum_Count*
t	1	1
h	1	2
i	2	4
s	1	5
~~*b*~~	1	6
<*esc*>	1	7
Total Count		7

We also update the counts in Tables 2.9 – 2.11.

The next letter to be encoded in the sequence is *t*. The second-order context is *s*~~*b*~~. Looking at Table 2.11, we can see that *t* has not appeared before in this context. We therefore encode an escape symbol. Using the counts listed in Table 2.11, we update the lower and upper limits:

l = 0 + [(15 – 0 + 1) * 1/2] = 8 = 1000

u – 0 + [(15 – 0 + 1) * 2/2] -1 = 15 = 1111.

Again, the MSBs of *l* and *u* are the same, so we shift the bit out and shift 0 into the LSB of *l*, and 1 into *u*, restoring *l* to a value of 0 and *u* to a value of 15. The transmitted sequence is now 01. After transmitting the escape, we look at the first order context of *t*, which is ~~*b*~~. Looking at Table 2.10, we can see that *t* has not previously occurred in this context. To let the decoder know this, we transmit another escape. Updating the limits, we get

l = 0 + [(15 – 0 + 1) * 1/2] = 8 = 1000

u – 0 + [(15 – 0 + 1) * 2/2] – 1 = 15 = 1111

As the MSBs of *l* and *u* are the same, we shift the MSB out and shift 0 into the LSB of *l* and 1 into the LSB of *u*. The transmitted sequence is now 011. Having escaped out of the first-order contexts, we examine Table 2.9 to see if we can encode *t* using zero-order context. Indeed we can, and using the *Cum-Count* array, we can update *l* and *u*:

l = 0 + [(15 – 0 + 1) * 0/2] = 0 = 0000

u – 0 + [(15 – 0 + 1) * 1/7] – 1 = 1 = 0001.

Table 2.10 Count array for first-order contexts.

Context	Letter	Count	*Cum_Count*
t	*h*	1	1
	<Esc>	1	2
Total Count			2
h	*i*	1	1
	<Esc>	1	2
Total Count			2
i	*s*	2	2
	<Esc>	1	3
Total Count			3
ƀ	*i*	1	1
	<Esc>	1	2
Total Count			2
s	*ƀ*	1	1
	<Esc>	1	2
Total Count			2

Table 2.11 Count array for second-order contexts.

Context	Letter	Count	*Cum_Count*
th	*i*	1	1
	<*Esc*>	1	2
Total Count			2
hi	*s*	1	1
	<*Esc*>	1	2
Total Count			2
is	*ƀ*	1	1
	<*Esc*>	1	2
Total Count			2
sƀ	*i*	1	1
	<*Esc*>	1	2
Total Count			2
ƀi	*s*	1	1
	<*Esc*>	1	2
Total Count			2

Table 2.12 Count array for zero-order context.

Letter	Count	*Cum_Count*
t	2	2
h	1	3
i	2	5
s	1	6
ƀ	1	7
<*esc*>	1	8
Total Count		8

Table 2.13 Count array for first-order contexts.

Context	Letter	Count	*Cum_Count*
t	*h*	2	2
	<Esc>	1	3
Total Count			3
h	*i*	1	1
	<Esc>	1	2
Total Count			2
i	*s*	2	2
	<Esc>	1	3
Total Count			3
ƀ	*i*	1	1
	i	1	2
	<Esc>	1	3
Total Count			3
s	*ƀ*	1	1
	<Esc>	1	2
Total Count			2

Table 2.14 Count array for second-order contexts.

Context	Letter	Count	*Cum_Count*
th	*i*	1	1
	<Esc>	1	2
Total Count			2
hi	*s*	1	1
	<Esc>	1	2
Total Count			2
is	*ƀ*	2	2
	<Esc>	1	3
Total Count			3
sƀ	*i*	1	1
	i	1	2
	<Esc>	1	3
Total Count			3
ƀi	*s*	1	1
	<Esc>	1	2
Total Count			2
ƀi	*h*	1	1
	<Esc>	1	2
Total Count			2

The three most significant bits of both *l* and *u* are the same, so we shift them out. After the update we get

Transmitted sequence: 011000

l:0000

u:1111

The next letter to be encoded is *h*. The second-order context *ƀt* has not occurred previously, so we move directly to the first-order context *t*. the letter *h* has occurred previously in this context, so we update *l* and *u* and obtain

Transmitted sequence: 0110000

l:0000

u:1111

The method of encoding should now be clear. At this point the various counts are shown in Tables 2.12 – 2.14.

The Escape Symbol

In our example we used a count of one for the escape symbol, thus inflating the total count in each context by one. Cleary and Witten call this Method A, and the corresponding algorithm is referred to as PPMA. There is really no obvious justification for assigning a count of one to the escape symbol. For that matter, there is no obvious method of assigning counts to the escape symbol. There have been various methods reported in the literature.

Another method described by Cleary and Witten is to reduce the counts of each symbol by one and assign these counts to the escape symbol. For example, suppose in a given sequence *a* occurs 10 times in the context of *prob*, *l* occurs 9 times, and o occurs 3 times in the same context (e.g., *problem*, *proboscis* etc.). In method A we assign a count of one to the escape symbol, resulting in a total count of 23, which is one more than the number of times *prob* has occurred. The situation is shown in Table 2.15

In this second method, known as Method – B, we reduce the count of each of the symbols a, *l*, and o by one and give the escape symbol a count of three, resulting in the counts shown in Table 2.16 The reasoning behind this approach is that if in a particular context more symbols can occur, there is a greater likelihood that there is a symbol in this context that has not occurred before. This increase the likelihood that the escape symbol will be used. Therefore, we should assign a higher probability to the escape symbol.

Table 2.15 Counts using Method – A.

Context	Symbol	Count
prob	*a*	10
	l	9
	o	3
	<*Esc*>	1
Total Count		23

Table 2.16 Counts using Method – B.

Context	Symbol	Count
prob	*a*	9
	l	8
	o	2
	<Esc>	3
Total Count		22

A variant of method B, appropriately named Method C, was proposed by Moffat [61]. In method C, the count assigned to the escape symbol is the number of symbols that have occurred in that context. In this respect, Method C, is similar to the Method B. The difference comes in the fact that instead of "robbing" this from the counts of individual symbols, the total count is inflated by this amount. This situation is shown in Table 2.17. While there is some variation in the performance depending on the characteristics of the data being encoded of the three methods for assigning counts to the escape symbols, on the average, Method C seems to provide the best performance.

Table 2.17 Counts using Method C

Context	Symbol	Count
prob	*a*	10
	l	9
	o	3
	<Esc>	3
Total		25

Another variant of PPM is PPMD method proposed by Paul Glor Howard [62] is slightly improved method for estimating the escape probability. Moffat's PPMC method is widely considered to be the best method of estimating escape probabilities. In the PPMC, each symbol's count in a context is taken to be number of times it has occurred so far in the context. The escape "event," that is, the occurrence of a symbol for the first time in the context, is also treated as a "symbol," with its own count. When a letter occurs for the first time, its count becomes 1; the escape count is incremented by 1, so the total count increases by 2, and at all other times the total count increases by 1.

PPMD is similar to PPMC except that it makes the treatment of new symbols more consistent by adding ½ instead of 1 to both the escape count and the new symbol’s count when a new symbol occurs; hence the total count always increases by 1.

In 2002, Shkarin proposed a variation of PPM algorithm – PPM with Information Inheritance [63]. This algorithm sets a new standard on the compression performance. The estimation of probability for the escape symbol is a very important and difficult task. As the higher order causes deterioration in the compression performance, the most often applied order for widely used PPMD is five. This is caused by frequent occurrence of the escape symbol and its bad estimation. However, estimation of the escape symbol’s frequency for PPMII is much better. PPMII uses orders even up to 64, but the main reason allowing to use such high orders is much better estimation of ordinary symbols’ probability. Escape estimation in PPMII is adaptive. It uses a secondary escape model (SEE [64]). SEE is a special, separate model used for better evaluation of probability for escape symbols only. Most PPM models use statistics from the longest matching context. PPMII inherits the statistics of shorter contexts when a longer context is encountered for the first time. The shorter (the last longest matching) context’s statistics are used to estimate the statistics of the longer context. The executable version of PPMII method is implemented as *PPMII.exe*

2.5 LEMPEL ZIV MARKOV CHAIN ALGORITHM

LZMA (Lempel-Ziv-Markov chain-Algorithm) is an optimized version of LZ77 [65]. LZMA uses a dictionary compression algorithm (a variant of LZ77), whose output is then encoded with a range encoder. It raises the compression ratio dramatically while maintaining high decompression speed and low memory requirements for decompression. The dictionary compressor produces a stream of literal symbols and phrase references, which is encoded one bit at a time by the range encoder, using a model to make a probability prediction of each bit. Prior to LZMA, most encoder models were byte-based (i.e. they coded each bit using a cascade of contexts to represent the dependencies on previous bits from the same byte). The main innovation of LZMA is that instead of a generic byte-based model, LZMA's model uses contexts specific to the bitfields in each representation of a literal or phrase. This is nearly as simple as a generic byte-based model, but gives much better compression because it avoids mixing unrelated bits together in the same context.

2.5.1 LZMA Algorithm

In LZMA compression, the compressed stream is a stream of bits, encoded using adaptive binary range coder. The stream is divided into packets, each packet describing either a single byte, or an LZ77 sequence with its length and distance implicitly or explicitly encoded. Each part of each packet is modeled with independent contexts, so the probability predictions for each bit are correlated with the values of that bit (and related bits from the same field) in previous packets of the same type.

There are 7 types of packets as shown in Table 2.18

Table 2.18 List of Packets used in LZMA

Packed code (bit sequence)	Packet description
0 + byteCode	A single byte encoded using an adaptive binary range coder. The range coder uses context based on some number of the most significant bits of the previous byte. Depending on the state machine, this can also be a single byte encoded as a difference from the byte at the last used LZ77 distance.
1+0 + len + dist	A typical LZ77 sequence describing sequence length and distance.
1+1+0+0	A one-byte LZ77 sequence. Distance is equal to the last used LZ77 distance.
1+1+0+1 + len	An LZ77 sequence. Distance is equal to the last used LZ77 distance.
1+1+1+0 + len	An LZ77 sequence. Distance is equal to the second last used LZ77 distance.
1+1+1+1+0 + len	An LZ77 sequence. Distance is equal to the third last used LZ77 distance.
1+1+1+1+1 + len	An LZ77 sequence. Distance is equal to the fourth last used LZ77 distance.

The length is encoded as shown in Table 2.19

Table 2.19 Encoding of Length

Length code (bit sequence)	Description
0+ 3 bits	The length encoded using 3 bits, gives the lengths range from 2 to 9.
1+0+ 3 bits	The length encoded using 3 bits, gives the lengths range from 10 to 17.
1+1+ 8 bits	The length encoded using 8 bits, gives the lengths range from 18 to 273.

The distance is encoded as follows:

First a distance class is encoded using 6 bits. The 5 other bits of the distance code encode the information about how many direct distance bits need to be extracted from the stream.

2.5.2 RANGE ENCODING

Range encoding conceptually encodes all the symbols of the message into one number, unlike Huffman coding which assigns each symbol a bit-pattern and concatenates all the bit-patterns together. Thus range encoding can achieve greater compression ratios than the one-bit-per-symbol upper bound on Huffman encoding and it does not suffer the inefficiencies that Huffman does when dealing with probabilities that are not exact powers of two.

The central concept behind range encoding is this: given a large-enough range of integers, and probability estimation for the symbols, the initial range can easily be divided into sub-ranges whose sizes are proportional to the probability of the symbol they represent. Each symbol of the message can then be encoded in turn, by reducing the current range down to just that sub-range which corresponds to the next symbol to be encoded. The decoder must have the same probability estimation the encoder used, which can either be sent in advance, derived from already transferred data or be part of the compressor and decompressor.

When all symbols have been encoded, merely identifying the sub-range is enough to communicate the entire message (presuming of course that the decoder is somehow notified when it has extracted the entire message). A single integer is actually sufficient to identify the sub-range, and it may not even be necessary to transmit the entire integer; if there is a sequence of digits such that every integer beginning with that prefix falls within the sub-range, then the prefix alone is all that's needed to identify the sub-range and thus transmit the message.

2.5.3 RELATIONSHIP WITH ARITHMETIC CODING

Arithmetic coding is the same as range encoding, but with the integers taken as being the numerators of fractions. These fractions have an implicit, common denominator, such that all the fractions fall in the range (0,1). Accordingly, the resulting arithmetic code is interpreted as beginning with an implicit "0.". As these are just different interpretations of the same coding methods, and as the resulting arithmetic and range codes are identical, each arithmetic coder is its corresponding range encoder, and vice-versa. In other words, arithmetic coding and range encoding are just two, slightly different ways of understanding the same thing.

An often noted feature of such range encoders is the tendency to perform renormalization a byte at a time, rather than one bit at a time (as is usually the case). In other words, range encoders tend to use bytes as encoding digits, rather than bits. While this does reduce the amount of compression that can be achieved by a very small amount, it is faster than when performing renormalization for each bit.

CHAPTER 3

STRING-MATCHING ALGORITHMS

3
STRING-MATCHING ALGORITHMS

3.0 OUTLINE OF THIS CHAPTER

This chapter is concerned with string matching methods for locating patterns occurring as a sub-string of a particular string. Such keywords searches are a common requirement in, for example, word processing and information retrieval applications. This chapter discusses the most popular string matching algorithms.

- *Brute-Force Algorithm*
- *Karp-Rabin Algorithm*
- *Knuth-Morris-Pratt Algorithm*
- *Boyer-Moore Algorithm*
- *Quick Search Algorithm*

3.1 INTRODUCTION

The general approach for looking for a pattern in a file that is stored in its compressed form is first decompressing and then applying one of the known pattern matching algorithms in the decoded file. In many cases, however, in particular on the Internet, files are stored in their original form, for if they were compressed, the host computer would have to provide memory space for each user in order to store the decoded file. This requirement is not reasonable, as many user scan simultaneously quest the same information reservoir which will demand an astronomical quantity of free memory. Another possibility is transferring the compressed files to the personal computer of the user, and then decoding the files. However, we then assume that the user knows the exact location of the information he is looking for; if this is not the case, much unneeded information will be transferred.

There is therefore a need to develop methods for directly searching within a compressed file. For a given text S and pattern P and complementary encoding and decoding functions E and D, our aim is to search $E(P)$ in $E(S)$, rather than the usual approach which searches for the pattern P in the decompressed text $D(E(S))$. But this is not always straightforward, since an instance of *E(P)* in the compressed text is not necessarily the encoding of instance of *P* in the original text *S*. This so-called compressed matching problem has been introduced by Amir and Benson [47]. The algorithms proposed in chapter 4 are useful for searching *E(P)* in *E(S)*, with any conventional string-matching algorithm discussed in this chapter.

3.2 STRING-MATCHING ALGORITHMS

String matching consists of finding one, or more generally, all the **occurrences** of a pattern in a text. The pattern and the text are both strings built over a finite alphabet (a finite set of symbols). Each algorithm describe here outputs all occurrences of the pattern in the text. The pattern is denoted by *P = P[0…m-1]*; its length is equal to *m*. The text is denoted by *S = S[0…n-1]*; its length is equal to *n*. The alphabet is denoted by $\sum$ and its size equal to σ .

String-matching algorithms work as follows: they first align the left ends of the pattern and the text, then compare the aligned symbols of the text and the pattern — this specific work is called an attempt or a scan — and after a whole match of the pattern or after a mismatch they shift the pattern to the right. They repeat the same procedure again until the right end of the pattern goes beyond the right end of the text. This is called the scan and shift mechanism. Each attempt is associated with the position *i* in the text when the pattern is aligned with *S[i…i+m-1]*.

The brute force algorithm consists of checking, at all positions in the text between *0* and *n-m*, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern exactly one position to the right. This is the simplest algorithm, which is described in Figure 3.1.

The time complexity of the brute force algorithm is *O(mn)* in the worst case but its behavior in practice is often linear on specific data.

Four categories arise: the most natural way to perform the comparisons is from left to right, which is the reading direction; performing the comparisons from right to left generally leads to the best algorithms in practice; the best theoretical bounds are reached when comparisons are done in a specific order; finally there exist some algorithms for which the order in which the comparisons are done is not relevant (such is the brute force algorithm)

```
void BF(char *s, char *p, int n, int m)
{
      int i, j;
      /* Searching */
      for (i=0; i <= n-m; i++)
      {
            j=0;
            while (j < m && s[i+j] == p[j])
                  j++;
            if (j >= m)
                  OUTPUT(i);
      }
}
```

Figure 3.1 The Brute Force string-matching algorithm.

3.2.1 From left to right

Hashing provides a simple method that avoids the quadratic number of character comparisons in most practical situations_ and that runs in linear time under reasonable probabilistic assumptions. It has been introduced by Harrison and later fully analyzed by Karp and Rabin[66].

Assuming that the pattern length is no longer than the memory-word size of the machine, the Shift-Or algorithm is an efficient algorithm to solve the exact string-matching problem and it adapts easily to a wide range of approximate string-matching problems.

The first linear-time string matching algorithm is from Morris and Pratt [67]. It has been improved by Knuth, Morris, and Pratt [7]. The search behaves like a recognition process by automaton and a character of the text is compared to a character of the pattern no more than $log_{\phi}(m+1)$ (ϕ is the golden ratio $(1+\sqrt{5})/2$). Hancart proved that this delay of a related algorithm discovered by Simon makes no more than $1+log_2 m$ comparisons per text character. Those three algorithms perform at most $2n-1$ text character comparisons in the worst case.

The search with a Deterministic Finite Automaton performs exactly n text character inspections but it requires an extra space in $O(m \text{ x } \sigma)$. The Forward Dawg Matching algorithm performs exactly the same number of text character inspections using the suffix automaton of the pattern.

The Apostolico-Crochemore algorithm is a simple algorithm which performs *3/2n* text character comparisons in the worst case.
The Not So Naive algorithm is a very simple algorithm with a quadratic worst case time complexity but it requires a preprocessing phase in constant time and space and is slightly sub-linear in the average case.

3.2.2 From right to left

The Boyer-Moore algorithm[6] is considered as the most efficient string matching algorithm in usual applications. A simplified version of it (or the entire algorithm) is often implemented in text editors for the “search” and “substitute” commands. Cole proved that the maximum number of character comparisons is tightly bounded by 3n after the preprocessing for non-periodic patterns. It has a quadratic worst case time for periodic patterns.
Several variants of the Boyer-Moore algorithm avoid its quadratic behavior. The most efficient solutions in term of number of character comparisons have been designed by Apostolico and Giancarlo, Crochemore et alii (TurboBM) and Colussi (Reverse Colussi).
Empirical results show that variations of the Boyer-Moore algorithm and algorithms based on the suffix automaton by Crochemore et alii (Reverse Factor and Turbo Reverse Factor) or the suffix oracle by Crochemore et alii (Backward Oracle Matching) are the most efficient in practice.
The Zhu-Takaoka and Berry-Ravindran algorithms are variants of the Boyer-Moore algorithm which require an extra space in $O(\sigma^2)$

3.2.3 In a specific order

The two first linear optimal space string-matching algorithms are due to Galil-Seiferas and Crochemore-Perrin (Two Way). They partition the pattern in two part, they first search for the right part of the pattern from left to right and then if no mismatch occurs they search for the left part.
The algorithms of Colussi and Galil-Giancarlo partition the set of pattern positions into two subsets. They first search for the pattern characters which positions are in the first subset from left to right and then if no mismatch occurs they search for the remaining

characters from left to right. The Colussi algorithm is an improvement over the Knuth-Morris-Pratt algorithm and performs at most *3/2n* text character comparisons in the worst case. The Galil-Giancarlo algorithm improves the Colussi algorithm in one special case which enables it to perform at most *4/3n* text character comparisons in the worst case.
Sunday’s Optimal Mismatch and Maximal Shift algorithms sort the pattern positions according their character frequency and their leading shift respectively.
Skip Search, KmPSkip Search and Alpha Skip Search algorithms by Charras et alii use buckets to determine starting positions on the pattern in the text.

3.2.4 In any order

The Horspool algorithm is a variant of the Boyer-Moore algorithm. It uses only one of its shift functions and the order in which the text character comparisons are performed is irrelevant. This is also true for other variants such as the Quick Search algorithm of Sunday[68], Tuned Boyer Moore of Hume and Sunday, the Smith algorithm and the Raita algorithm.

3.3 KARP-RABIN ALGORITHM

Hashing provides a simple method for avoiding a quadratic number of symbol comparisons in most practical situations. Instead of checking at each position of the text whether the pattern occurs, it seems to be more efficient to check only if the portion of the text aligned with the pattern “looks like” the pattern. In order to check the resemblance between these portions a hashing function is used. To be helpful for the string-matching problem the hashing function should have the following properties:

- efficiently computable,
- highly discriminating for strings,
- *hash (s[i+1. . . i+m])* must easily computable from *hash(s[i+1. . .i+m-1])*:
  hash (s[i+1. . . i+m]) = rehash(s[i],s[i+m],hash(s[i. . .i+m-1])).

For a word *w* of length *k*, its symbols can be considered as digits, and we define *hash(w)* by:

$$hash(w[0...k-1]) = (w[0] * 2^{k-1} + w[1] * 2^{k-2} + ... + w[k-1] \bmod q,$$

where *q* is a large number.

Then, rehash has a simple expression

$$rehash(a,b,h) = ((h-a*d) *2+b) \bmod q,$$

where d = 2^{k-1}.

During the search for the pattern *P*, it is enough to compare *hash(p)* with *hash(s[i. . .i+m-1])* for $0 \le i \le n-m$. If an equality is found, it is still necessary to check the equality *P = s[i. . .i+m-1]* symbol by symbol.

In the algorithm of Figure 3.2 all the multiplications by 2 are implemented by shifts. Furthermore, the computation of the modulus function is avoided by using the implicit modular arithmetic given by the hardware that forgets carries in integer operations. So, *q* is chosen as the maximum value of an integer.

```
#define REHASH(a, b, h) (((h-a*d)<<1)+b)
void KR(char *s, char *p, int n, int m)
{
      int hs, hp, d, i;
      /* Preprocessing */
      /* computes d = 2^(m-1) with the left-shift operator */
      d=1;
      for (i=1; i < m; i++)
            d<<=1;
      hs=hp=0;
      for (i=0; i < m; i++)
      {
            hp=((hp<<1)+p[i]);
            hs=((hs<<1)+s[i]);
      }
      /* Searching */
      for (i=m; i <= n; i++)
      {
            if (hs == hp && strncmp(s+i-m, p, m) == 0) OUTPUT(i-m);
            hs=REHASH(s[i-m], s[i], hs);
      }
}
```

Figure 3.2 The Karp-Rabin string-matching algorithm.

The worst-case time complexity of the Karp-Rabin algorithm is quadratic in the worst case (as it is for the brute force algorithm) but its expected running time is *O(m+n)*

Example 3.1:

Let *P* = ing.

Then *hash(p) = 105 * 22 + 110*2 + 103 = 743* (symbols are assimilated with their ASCII codes).

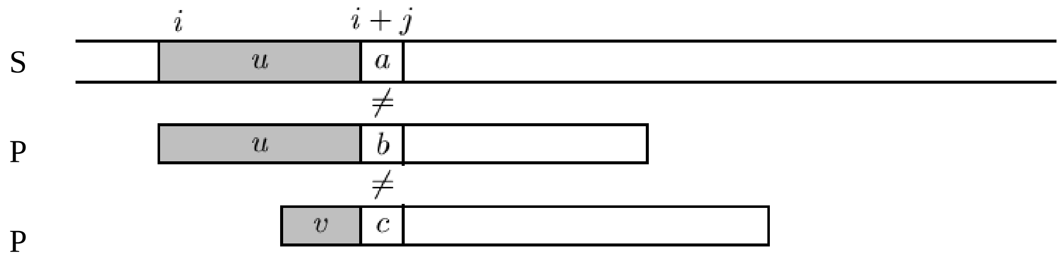

Figure 3.3 Shift in the Knuth-Morris-Pratt algorithm (*v* suffix of *u*).

S = *s t r* *i n g* *m a t c h* *i n g*
hash = *806 797 776 743 678 585 443 746 719 766 709 736 743*

3.4 KNUTH-MORRIS-PRATT ALGORITHM

This section presents the first discovered linear-time string-matching algorithm. Its design follows a tight analysis of the brute force algorithm, and especially on the way this latter algorithm wastes the information gathered during the scan of the text.

Let us look more closely at the brute force algorithm. It is possible to improve the length of shifts and simultaneously remember some portions of the text that match the pattern. This saves comparisons between characters of the text and of the pattern, and consequently increases the speed of the search.

Consider an attempt at position *i*, that is, when the pattern *P[0. . .m-1]*]is aligned with the window *S[i. . .i+m-1]* on the text. Assume that the first mismatch occurs between symbols *S[i+j]* and *P[j]* for $1 < j < m$. Then, *S[i. . .i+j-1] = P[0. . .j-1] = u* and *a = S[i+j]* ≠ *P[j] = b*. When shifting, it is reasonable to expect that a **prefix** *v* of the pattern matches some **suffix** of the portion *u* of the text. Moreover, to avoid another immediate mismatch, the letter following the prefix *v* in the pattern must be different from *b*. The longest such prefix *v* is called the **border** *u* (it occurs at both ends of *u*). This introduces the notation: let *next[j]* be the length of the longest (proper) border of *P[0. . .j-1]* followed by a character *c* different from *P[j]*. Then, after a shift, the comparisons can resume between characters *S[i+j]* and *P[next[j]]* without missing any occurrence *P* in *S*, and avoiding a backtrack on the text (see Figure 3.3).

Example 3.2:
S = . . . *a b a b a a* . . .
P = $\underline{a}\ \underline{b}\ \underline{a}\ \underline{b}\ \underline{a}\ \underline{b}\ a$
P = $\underline{a}\ \underline{b}\ a\ b\ a\ b\ a$

Compared symbols are underlined. Note that the empty string is the suitable border of *ababa*. Other borders of *ababa* are *aba* and *a*.

The Knuth-Morris-Pratt algorithm is displayed in Figure 3.4. The table *next* it uses is computed in *O(m)* time before the search phase, applying the same searching algorithm to the pattern itself, as if *(S=P)* (see Figure 3.5). The worst-case running time of the algorithm is *O(m+n)* and it requires *O(m)* extra space. These quantities are independent of the size of the underlying alphabet.

```
void KMP(char *s, char *p, int n, int m)
{
      /* XSIZE is the maximum size of a pattern */
      int i, j, next[XSIZE];
      /* Preprocessing */
      PRE_KMP(p, m, next);
      /* Searching */
      i=j=0;
      while (i < n)
      {
            while (j > -1 && p[j] != s[i]) j=next[j];
            i++; j++;
            if (j >= m)
            {
                  OUTPUT(i-j);
                  j=next[m];
            }
      }
}
```

Figure 3.4 The Knuth-Morris-Pratt string-matching algorithm.

3.5 BOYER-MOORE ALGORITHM

The Boyer-Moore algorithm is considered the most efficient string-matching algorithm in usual applications. A simplified version of it, or the entire algorithm, is often implemented in text editors for the “search” and “substitute” commands.

The algorithm scans the characters of the pattern from right to left beginning with the rightmost symbol. In case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the pattern to the right. These two shift functions are called the *bad-character shift* and the *good-suffix shift*. They are based on the following observations.

```
void PRE_KMP(char *p, int m, int next[])
{
      int i, j;
      i=0; j=next[0]=-1;
      while (i < m)
      {
            while (j > -1 && p[i] != p[j])
                  j=next[j];
            i++; j++;
            if (i < m && p[i] == p[j])
                  next[i]=next[j];
            else
                  next[i]=j;
      }
}
```

Figure 3.5 Preprocessing phase of the Knuth-Morris-Pratt algorithm: computing *next*.

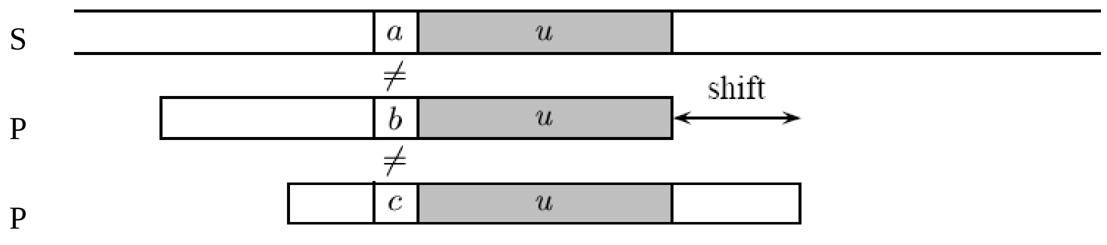

Figure 3.6 Good-suffix shift, u reappears preceded by a character different from b.

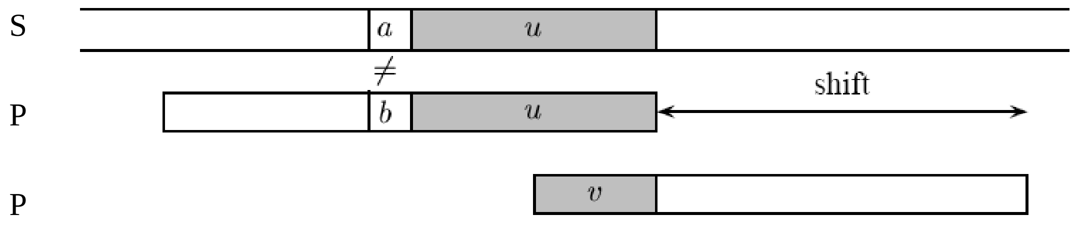

Figure 3.7 Good-suffix shift, only a suffix of u reappears as a prefix of P.

Assume that a mismatch occurs between the character *P[j] = b* of the patter and the character *S[i+j] = a* of the text during an attempt at position *i*. Then, *S[i+j+1. . .i+m-1] P[j+1. . .m-1] = u* and *S[i+j]* ≠ *P[j]*.

The good-suffix shift consists of aligning the segment *S[i+j+1. . .i+m- 1] = P[j+1. . .m-1]* with its rightmost occurrence in *P* that is preceded by a character different from *P[j]* (see figure 3.6) if there exists no such segment, the shift consists of aligning the longest suffix *v* of *S[i+j+1. . .i+m-1]* with a matching prefix of *P* (see figure 3.7).

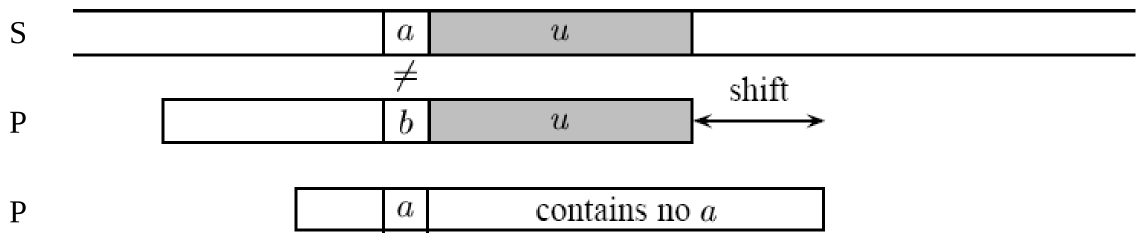

Figure 3.8 Bad-character shift, *a* appears in *P*.

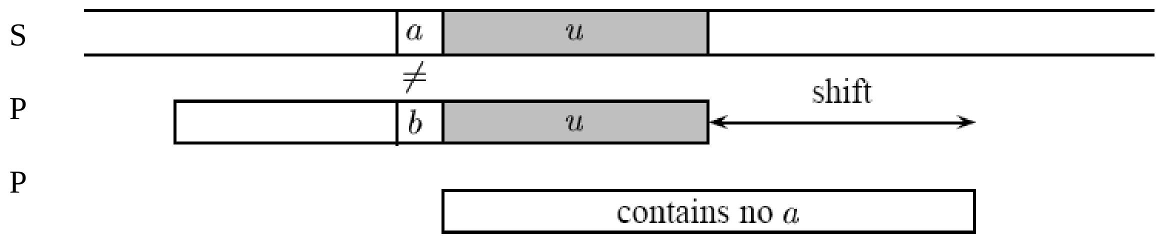

Figure 3.9 Bad-character shift, *a* does not appears in *P*.

Example 3.3:
S = . . .   *a b b a a b b a b b a . . .*
P = *a b b a a b b a b b a*
P =   *a b b a a b b a b b a*

The shift is driven by the suffix *abba* of *P* found in the text. After the shift, the segment *abba* in the middle of *S* matches a segment of *P* as in figure 3.6. The same mismatch does not reoccur.

Example 3.4:
S = . . . *a b b a a b b a b b a b b a . . .*
P =   *b b a b* $\underline{b}\ \underline{a}\ \underline{b}\ \underline{b}\ \underline{a}$
P = $\underline{b}\ \underline{b}\ \underline{a}\ \underline{b}\ \underline{b}\ \underline{a}\ \underline{b}\ \underline{b}\ \underline{a}$

The segment *abba* found in *S* partially matches a prefix of *P* after the shift, like in Figure 3.7.

The bad-character shift consists of aligning the text character *S[i+j]* with its rightmost occurrence in *P[0. . .m-2]* (see figure 3.8) If *S[i+j]* does not appear in the pattern *P*, no occurrence of *P* in *S* can overlap the symbol *S[i+j]*, then, the left end of the pattern is aligned with the character at position *i+j+1* (see figure 3.9)

Example 3.5:
S = ... *a b c d* . . . .
P = *c d a h g f* <u>*e*</u> <u>*b*</u> <u>*c*</u> <u>*d*</u>
P = *c d a h g f e b c* <u>*d*</u>
The shift aligns the symbol *a* in *P* with the mismatch symbol *a* in the text *S* (Figure 3.8).

Example 3.6:
```
S =  . . . . . a b c d . . . . . .
P = c d h g f e b c d
P =              c d h g f e b c d
```
The shift positions the left end of P right after the symbol *a* of *S* (Figure 3.9).

The Boyer-Moore algorithm is shown in Figure 3.10. For shifting the pattern, it applies the maximum between the bad-character shift and the good-suffix shift. More formally, the two shift functions are defined as follows. The bad-character shift is stored in a table *bc* of size σ and the good-suffix shift is stored in a table *gs* of size *m+1*. For a $\in \sum$:

$$bc[a] = \begin{cases} \min\{ j / 1 \le m \text{ and x[m - 1 - j]} = \text{a }\}\text{if a appears in x} \\ \text{m} \qquad\qquad\qquad\qquad\qquad \text{otherwise} \end{cases}$$

```
void BM(char *s, char *p, int n, int m)
{
      /* XSIZE is the maximum size of a pattern */
      /* ASIZE is the size of the alphabet */
      int i, j, gs[XSIZE], bc[ASIZE];
      /* Preprocessing */
      PRE_GS(p, m, gs);
      PRE_BC(p, m, bc);
      /* Searching */
      i=0;
      while (i <= n-m)
      {
            j=m-1;
            while (j >= 0 && p[j] == s[i+j])
                  j--;
            if (j < 0)
                  OUTPUT(i);
            i+=MAX(gs[j+1];
            bc[s[i+j]]-m+j+1); /* shift */
      }
}
```
Figure 3.10 The Boyer-Moore string-matching algorithm.

Let us define two conditions:

$cond_1(j;s)$: for each k such that $j<k<m$, $s \ge k$ or $p[k - s]=p[k]$
$cond_2(j,s)$: if $s<j$ then $p[j - s] \neq p[j]$
Then, for $0 \le j< m$:

$gs[j+1]=\min\{s>0 / cond_1(j, s) \text{and } cond_2(j, s) \text{ hold}\}$

and define *gs[0]*as the length of the smallest period of *p*.

```
void PRE_BC(char *p, int m, int bc[])
{
      /* ASIZE is the size of the alphabet */
      int j;
      for (j=0; j < ASIZE; j++) bc[j]=m;
      for (j=0; j < m-1; j++) bc[p[j]]=m-j-1;
}
```

Figure 3.11 Computation of the bad-character shift.

Tables *bc* and *gs* can be precomputed in time $O(m+\sigma)$before the search phase and require an extra-space in $O(m+\sigma)$(see Figures 3.12 and 3.11). The worst-case running time of the algorithm is quadratic. However, on large alphabets (relative to the length of the pattern) the algorithm is extremely fast. Slight modifications of the strategy yield linear-time algorithms

When searching for $a^{m-1}b$ in a^n the algorithm makes only $O(n/m)$ comparisons, which is the absolute minimum for any string-matching algorithm in the model where the pattern only is preprocessed.

```
void PRE_GS(char *p, int m, int gs[])
{
      /* XSIZE is the maximum size of a pattern */
      int i, j, p, f[XSIZE];
      for (i=0; i <= m; i++) gs[i]=0;
            f[m]=j=m+1;
      for (i=m; i > 0; i--)
      {
            while (j <= m && p[i-1] != p[j-1])
            {
                  if (!gs[j]) gs[j]=j-i;
                  j=f[j];
            }
            f[i-1]=--j;
      }
      p=f[0];
      for (j=0; j <= m; j++)
      {
            if (!gs[j]) gs[j]=p;
            if (j == p) p=f[p];
      }
}
```

Figure 3.12 Computation of the good-suffix shift.

3.6 QUICK SEARCH ALGORITHM

The bad-character shift used in the Boyer-Moore algorithm is not very efficient for small alphabets, but when the alphabet is large compared with the length of the pattern, as it is often the case with the ASCII table and ordinary searches made under a text editor, it becomes very useful. Using it only produces a very efficient algorithm in practice that is described now.

After an attempt where P is aligned with *S[i. . .i+m-1]*, the length of the shift is at least equal to one. So, the character *S[i+m]* is necessarily involved in the next attempt, and thus can be used for the bad-character shift of the current attempt. In the present algorithm, the bad-character shift is slightly modified to take into account the observation as follows ($a \in \sum$):

$$bc[a] = \begin{cases} \min\{ j / 0 \leq m \text{ and } \mathrm{x[m-1-j] = a} \} & \text{if a appears in x} \\ \mathrm{m} & \text{otherwise} \end{cases}$$

Indeed, the comparisons between text and pattern characters during each attempt can be done in any order. The algorithm of Figure 3.13 performs the comparisons from left to right. It is called Quick Search after its inventor and has a quadratic worst-case time complexity but a good practical behavior.

Example 3.7:

$S = s\ t\ r\ i\ n\ g - m\ a\ t\ c\ h\ i\ n\ g$

$P = \underline{i}\ n\ g$

$P = \qquad \underline{i}\ \underline{n}\ \underline{g}$

$P = \qquad\qquad\quad \underline{i}\ n\ g$

$P = \qquad\qquad\qquad\qquad \underline{i}\ n\ g$

$P = \qquad\qquad\qquad\qquad\quad \underline{i}\ \underline{n}\ \underline{g}$

Quick Search algorithm makes 9 comparisons to find the two occurrences of `ing` inside the text of length 15.

For direct searching with simple text, the linear BF algorithm is a proper choice because it produces relatively good running time results despite its striking simplicity. In addition, the BF algorithm has no special memory requirements and needs no preprocessing or complex coding and thus can be surprisingly fast. But this algorithm shouldn't use for the binary alphabet in applications such as image processing or software systems.

```
void QS(char *s, char *p, int n, int m)
{
      /* ASIZE is the size of the alphabet */
      int i, j, bc[ASIZE];
      /* Preprocessing */
      for (j=0; j < ASIZE; j++)
            bc[j]=m;
      for (j=0; j < m; j++)
            bc[p[j]]=m-j-1;
      /* Searching */
      i=0;
      while (i <= n-m)
      {
            j=0;
            while (j < m && p[j] == s[i+j])
                  j++;
            if (j >= m)
                  OUTPUT(i);
            i+=bc[s[i+m]]+1; /* shift */
      }
}
```

Figure 3.13 The Quick Search string-matching algorithm.

From the empirical evidence it can be concluded that the KR algorithm is linear in the number character comparisons but it has higher running time and it shouldn't be used for pattern matching in strings. However, the main advantage of this algorithm lies in its extension to higher dimensional string matching. It may be used for pattern recognition and image processing and thus in the expanding field of computer graphics.

Despite its theoretical elegance, the KMP algorithm provides no significant speedup advantage over the BF algorithm in practice unless the pattern has highly repetitive subpatterns. However the KMP algorithm guarantees a linear bound and it is well suited to extensions for more difficult problems. It may be a good choice when the alphabet size is near the text size or when dealing with the binary alphabet.

Based on empirical results, it is clear that the QS algorithm is proved to be much faster algorithm in practice than the rest BM-like, suffix automata and bit-parallelism algorithms for large alphabets and short patterns. Therefore it is typically suited for search in the English alphabet. In addition, the BM algorithm is faster than its variations (such as BMH, QS, BMS and TBM) for small alphabets and long patterns. However, in theory BMS and QS are better algorithms than BM-like and suffix automata algorithms for short patterns and large alphabets [69].

CHAPTER 4

PROPOSED COMPRESSION METHODS

4

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 dynmamic 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 character 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 *0(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

word0, word1, word2, . . ., 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

Col 0 Col1 Col 62 Col63 Col64 Col126

Row 0 ***word0, word1, . . . , word62,*** *word63, word64, ..., word126*

Row 1 ***word0, word1, . . . , word62****, word127, word128, ..., word190*

.

.

.

Row 254 ***word0, word1, . . . , word62,*** *word16319, word16320, . . ., word16382*

Row 255 ***word0, word1, . . . , word62,*** *word16383, word16384, . . ., word16446*

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.

C0 C1 C12 C13 C14 C27

$\downarrow$ $\downarrow$ . . . $\downarrow$ $\downarrow$ $\downarrow$ . . . $\downarrow$

R0 $\rightarrow$ *ch0, ch1, ..., ch12, ch13, ch14 . . ., ch27*

R1 $\rightarrow$ *ch0, ch1, ..., ch12, ch28, ch29, . . ., ch42*

R2 $\rightarrow$ *ch0, ch1, ..., ch12, ch43, ch44, . . ., ch57*

R3 $\rightarrow$ *ch0, ch1, ..., ch12, ch58, ch59, . . ., ch72*

R4 $\rightarrow$ *ch0, ch1, ..., ch12, ch73, ch74, . . ., ch87*

R5 $\rightarrow$ *ch0, ch1, ..., ch12, ch88, ch89, . . ., ch102*

R6 $\rightarrow$ *ch0, ch1, . . ., ch12, ch103, ch104, . . ., ch117*

R7 $\rightarrow$ *ch0, ch1, . . ., ch12, ch118, ch119, . . ., ch132*

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)

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

Figure 4.3 First six rows of single character

In the Figure 4.3, it is seen that *~~b~~ethaonsirdlu* 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.

athe atio agai ains ause also aith ayin afte arth avid aven ange alle amon away acco augh ance ater abou alem ants befo beca brou brea brin beho burn come came chil caus comm call cord city ccor cove dren ding down days dest dwel deli dred efor ever eopl even eart ered eref ehol erin ecau eave ereo esus erva esse eith ence elve east erus ethe evil ears ents eard from fore ffer fath fter fort feri gain give ghte grea good gypt geth hall here hich heir have hous hear hath hing hand hild hold halt hese hast hine hose houl heav hers hund ight ings into ildr inst ithe ions iver iest ites irst ince judg king know ldre land live lled lace like less lves ment made make msel mong mand mman migh mine more mber ness name nder ning neit nger nati nded noth ndre nswe ough ouse ople over othe ould offe ound orth ount omma oses ordi oice outh ophe peop pass plac prie part peak rael ring roug refo reat righ reth ries rvan reof read rdin rusa shal said srae serv sayi sait sons self shou side sent stro sale stan ssed spea selv seve swer that ther they them thou thei thee thin tion this thre then take thes tain turn thro than tter time ters took tand unto upon ught unde urne usal udah very vant with whic will were when word went ward wher what work well whom wate your ying year

Figure 4.4 4-character groups

the and all hat ing her tha for sha hal ere his nto unt hou ith not hey him hem wit tho eth ear thi ave ver ath ent ght our hen sai ter ill man you eve ore thy out was ich whi ain est ord aid ive wil are hic igh one ame ion com hea ven hee ess hav hei use ake ers eir ous wer red ast rom ove son ine hin kin men whe fro rea rin efo han eat ugh art oth wor ple tio ome oun old ren und nes int hil pon dre nce upo ons ild ose chi day ins she ael ati rth hol but oug rae sra rou ies say les ate alt str cam gai ngs led aga ong who ace had thr der own sel eri now ard ead eop mon nde peo opl ant oul urn off ted con ass eas ood ret rie ise hes lan res way ned see ite als sse nst ort gre hos ait ldr pri ell bef ser min lso pro sed ref ffe bro cau ses yin ese ble aus eho tes wen fer rit lle lea erv ity ice ery giv tte ade let any nge des uld fat ide tre nts ris din fte sen ves ten ayi gat ree sta ist bre pas ans ure kno pla war tan mad eca ook hte ene avi sin rne har cor usa pea liv ken pre ste has tur ale abo dow ish hre

Figure 4.5 3-character groups

th he nd an in er ha re of hi at ou en or to al ll on es is it se nt ve ed ar ea ng sh st ho

Figure 4.6 2-character groups

Compression

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 ‘00000100’ combine both bytes to form 16-bit data ‘10100000 00000100’ 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 ‘00001111’ combine both bytes to form 16-bit data ‘11000000 00001111’ 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

InƀtheƀbeginningƀGodƀcreatedƀtheƀheavenƀandƀtheƀearth.

can be divided into word, non-word sequence: “In”, “ƀ”, “the”, “ƀ”, “beginning”, “ƀ”, “God”, “ƀ”, “God”, “ƀ”, “created”, “ƀ”, “the”, “ƀ”, “heaven”, “ƀ”, “and”, “ƀ”, “the”, “ƀ”, “earth”, ”.” (where ƀ 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 ‘*coming*’ and ‘*going*’ is appearing only once in the source file. In both of the words the suffix string ‘*ing*’ is appearing, therefore the sub-word ‘*ing*’ will be added to the suffix sub-word dictionary. In the similar way the prefix words are added to the prefix sub-word dictionary.

Also the dictionary of non-words is also created, which includes words of non-alphabets. For e.g. say after the word ‘going’ there is full stop and carriage return, then both the symbols full stop and carriage return will be considered as one non-word and will be added to dictionary of non-words.

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 * 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.

Col 0 ↓ **Col1** ↓ **Col 62** ↓ **Col63** ↓ **Col64** ↓ **Col126** ↓

Row 0 → ***word0, word1, . . . , word62,*** *word63, word64, ..., word126*

Row 1 → ***word0, word1, . . . , word62****, word127, word128, ..., word190*

.

.

.

Row 254 → ***word0, word1, . . . , word62,*** *word16319, word16320, . . ., word16382*

Row 255 → ***word0, word1, . . . , word62,*** *word16383, word16384, . . ., word16446*

Figure 4.7 Structure of two-dimension word dictionary (WBTC-C)

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:

$$\textit{row number} = (\textit{position} - 63) / 64$$

$$\textit{column number} = (\textit{position} - 63) \bmod 64$$

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:

$$\textit{index value} = (\textit{row} * 64) + 63 + \textit{column number}$$

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 a 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.

Col 0 Col1 Col 62 Col63 Col64 Col126

↓ ↓ ↓ ↓ ↓ ↓

Row 0 → ***word0, word1, . . , word31999,*** *word32000, ..., word32510*

Row 1 → ***word0, word1, . . , word31999****, word32511, ..., word33021*

.

.

.

Row 254 → ***word0, word1, . . , word31999****, word161794, ..., word162304*

Row 255 → ***word0, word1, . . , word31999****, word162305, ..., word162815*

Figure 4.8 Structure of two-dimension word dictionary (WBTC-E)

Even though it seems that *word0, word1,. . . ., 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.

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 continous 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.

CHAPTER 5

IMPLEMENTATION OF PROPOSED METHODS

5
IMPLEMENTATION OF PROPOSED METHODS

5.0 OUTLINE OF THIS CHAPTER

This chapter describes the implementation of different compression methods proposed by us. The methods are already discussed in chapter 4. In this chapter certain issues related to implementation part is discussed and flow charts of each methods are drawn. All these methods are implemented in VC++ 6.0.

5.1 Implementation of CBTC-A

This method is based on static dictionary created from the set of corpus (Large Corpus, E-Text, Enronsent, European Parliament and Gutenberg). The probability of all single characters is computed and the characters are arranged in the descending order. In the similar way, the probability of all possible 4-character group, 3-character group and 2 character group is computed and are arranged in descending order. The first 256 groups are taken into consideration. The static dictionary of characters will be as shown in figure 4.3 and that of 4-character group, 3-character group and 2 character group will be as shown in fig. 4.4., 4.5, and 4.6 respectively. (All these figures are shown in chapter 4).

Compression

During the compression process, we have to specify the change in row number, or the encoded sequence is pointing to group of 4-character, group of 3-character or group of 2-character. This we will do by sending special escape symbols. The number of characters kept in one row is 28 and we are using 5-bit coding sequence i.e. we can point to $2^5 = 32$ total combinations. Now, 28 combinations (0 – through – 27) are used for pointing to characters in row, and remaining 4 combinations are used for pointing the presence of 2-character group (28 i.e. binary ‘11100’), 3-character group (29 i.e. binary ‘11101’), 4-character group (30 i.e. binary ‘11110’) and change in row number (31 i.e. ‘11111’). Thus we can say that the 28, 29, 30 and 31 are special unique escape symbols. The escape symbols will be followed by the position of 2-character group, or by the position of 3-character group or by the position of 4-character group in the dictionary.

Step 1: Read the static dictionary (The dictionary is stored in the form of two-dimension dictionary).

Code for reading single characters from the file in the form of two-dimension dictionary is given below:

```
fptr = fopen("dict121.dat","r");
while((ch = fgetc(fptr))!= EOF) {
        dict[i][j] = ch;
        j++;
        if(j>27) {
                j=0; i++;
        }
}
```

The static dictionary file *dict121.dat* is opened in read mode. Since the number of characters in the dictionary is not known in advance, *while* loop is used to read the characters from the file. The array *dict[][]* is used to store the dictionary in two-dimension. One-by-one, the characters are read from the file (using *fgetc()* function) and are stored in *dict[i][j]*, where *i* is pointing to the row index and *j* is pointing to the column index. After every 28 characters the row index *i* is incremented by 1 and column index *j* is reset to zero. The process continues till all the characters are read from the dictionary file. Now the static dictionary is available in the array *dict[][]*.

Code for reading the dictionary of multiple characters (2, 3 and 4 characters group) from the file is given below:

```
fptr = fopen("dictionary432.dat","r");
for(i=0;i<1024;i++) {
        ch = fgetc(fptr);
        dictl4[i]=ch;
}
for(i=0;i<768;i++) {
        ch = fgetc(fptr);
        dictl3[i] = ch;
}
```

```
for(i=0;i<512;i++) {
        ch = fgetc(fptr);
        dictl2[i] = ch;
}
fclose(fptr);
```

As the number of character groups are fixed in the dictionary, we are using here *for* loop to read the character groups from the dictionary. First *for* loop will iterate for 1024 times as we want to read 256, 4-character groups. Second *for* loop will iterate for 768 times to read 256, 3-character groups and third *for* loop will iterate for 512 times to read 256, 2-character group.

Step 2: First read 4 characters from the source file in ch1, ch2, ch3 and ch4.

Step 3: Search the presence of these 4 characters in the dictionary of 4-character group. If found, store the escape symbol ‘11110’ and then store the index value of that position in the compressed file. The *OutputBit*() function is used to store the bits in the compressed file. The Step 2 is repeated.

Step 4: If those 4 characters are not found in the 4-character group dictionary, then first three characters ch1, ch2 and ch3 are searched in 3-character group dictionary. If found store the escape symbol ‘11101’ and then store the index value of that position in the compressed file. The ch4 is now stored in ch1 and next 3 characters are read in ch2, ch3 and ch4 and Step 3 is repeated.

Step 5: If those 3 characters are not found in the 3-character group dictionary, then first two characters ch1 and ch2 are searched in the 2-character group dictionary. If found store the escape symbol ‘11100’ and then store the index value of that position in the compressed file. The ch3 and ch4 are now stored in ch1 and ch2 respectively and next two characters are read from the source file in ch3 and ch4 and Step 2 is repeated.

Step 6: If those 2 characters are not found in the 2-character group dictionary, then the first character ch1 is searched in the single dictionary and its column position is stored in the compressed file. If the character is not found in the same row, then the escape symbol ‘11111’ is stored first which is followed by the new row number and column number.

The ch2, ch3 and ch4 are now stored in ch1, ch2 and ch3 respectively and next one character is read from the source file in ch4 and Step 3 is repeated.

The presence of the 4-character group is done in the following way.

```
int occur[];
void find4char() {
        int i,j;
        for(i=0;i<=len-4;i++) {
                for(j=0;j<256;j++) {
                        if (strcmp (str, dict4[j]) == 0) {
                                occur[i]=4;
                                i+=3;
                                break;
                        }
                }
        }
}
```

In the above code, the word is scanned and 4-consecutive character groups of a word is compared with 4-char group in dictionary, if found the presence of 4-character group in the word is marked in *occur*[] array. In the similar way, the presence of 3-character group and 2-character group is marked in *occur*[]. Thus while compressing the entire word the *occur*[] array is checked first for the presence of the character groups and then their respective index value is stored in the compressed file.

Decompression

Step 1: Read the dictionary of single character and multiple characters (2, 3 and 4 characters group)

Step 2: Read 5-bits from the compressed file.

Step 3: Compare these five bits with the escape symbols of 4-character, 3-character and 2-character group, if matched, then read next 8-bit code as an index value which points to the position in the dictionary, to retrieve the characters from the respective dictionary and store it in the decompressed file and repeat step 2.

Step 4: Compare these five bits with the escape symbol of change in row, if matched then read next 3-bits to read the new row number and 5-bits to read the column number. If those five bits are not matched with escape symbol of change in row, then those 5-bits code is assumed to be the column number of the previous row in the two-dimension single character dictionary. Retrieve the character from the two-dimension single character dictionary from respective row and column number and store it in the decompressed file. Thus the entire file is scanned and decompressed.

5.2 IMPLEMENTATION OF CBTC-B

In this method main task is to accumulate the statistics of groups of 2, 3 and 4 ASCII characters from the source file. The difference between CBTC – A and CBTC – B is that in later, 16384 – 4-character and 8192 – 3-character groups are taken into consideration whose frequency count is greater than certain threshold, whereas in CBTC – A only first 256 groups are taken into consideration.

Dictionary Creation

The arrays are declared for all possible combinations of groups of 2, 3 and 4, ASCII characters. For e.g. the possible group of 2 ASCII characters are *aa, ab, ac,. . ., za, zb,. . . zz.* The possible group of 3 ASCII characters are *aaa, aab,. . ., aza, azb, . . ., azz, . . ., zaa, zab, . . ., zzz.* The possible group of 4 ASCII characters are *aaaa, aaab, . . ., azaa, azab, . . ., azzz, zaaa, zaab, . . ., zzzz.* The frequency of such groups of ASCII characters is initially set to zero. The source file is then scanned in the following manner.

Step 1: First four characters are read from the source file in *ch1*, *ch2*, *ch3* and *ch4*. If all the four characters are ASCII characters, then the frequency count of that group of 4-character in array is incremented by 1.

If all four characters are not ASCII, then check whether first 3 characters are ASCII or not, if yes then the frequency count of that group of 3-character in array is incremented by 1, or else first two characters are checked for ASCII and if yes, then the frequency count of that group is incremented by 1.

In this way, all the characters from the file are scanned and the frequency count of the 4-character array, 3-character array and 2-character array is updated.

Step 2: Next, the groups of ASCII characters with frequency count zero is removed.

Example: Suppose 4 character group say ‘zzzz’ does not exist in the source file, then that group will be removed from the array.

Step 3: Thus after removing the zero frequency character groups, the remaining character groups will be sorted according to their frequency counts in descending order.

Step 4: In order to achieve compression, we have to take decision which character groups shall be kept in the dictionary. For example, if 4-character group say '*zing*' appears only twice or less, then it should be removed from the dictionary. Because, even if we keep in the dictionary, we won't get compression, as 4 bytes are required to store the 4-character group in the dictionary, and to encode it twice each time 2 bytes will be required so overall 8 bytes are required for compression. In original file, it will consume 8 bytes, since the 4-character group occurs twice. Therefore it won't be wise to keep the 4-character group whose frequency count is less than or equal to 2 to keep it in the dictionary. Similarly, the 3-character group whose frequency count is less than or equal to 3 shall not be kept in the dictionary. Thus, now we will have the 4-character and 3-character groups in the dictionary in the descending order with respect to frequency of their occurrence in the source file.

Step 5: Similarly, we have to think about 2-character group. If we consider all possible 2-character group, it will take 2 bytes to represent it. Thus we won't achieve compression in case of 2-character groups. In order to get compression for 2-character group, if we can encode any how in 1 byte then we will get compression. To encode it in one byte, we have to explicitly specify that this byte contains the index value pointing to the 2-character dictionary. In order to do this some bits will have to be kept reserved for that. Already 1-bit is kept reserved for indicating a normal ASCII character and an encoded value. There are three possibilities in encoded value, one is either it can be a value pointing to 4-character group dictionary, or 3-character group dictionary or else 2-character group dictionary. So, further 2-bits will required for specifying three possible values. Hence, only 5-bits remain to encode the 2-character group. Thus, we will be able to keep only first 32 2-character group in the dictionary.

Compression

Step 1: First read 4 characters from the source file in ch1, ch2, ch3 and ch4.

Step 2: Search the presence of these 4 characters in the dictionary of 4-character group. If found store the index value of that position in the compressed file. The OutputBit() function is used to store the bits in the compressed file. The Step 1 is repeated.

Step 3: If those 4 characters are not found in the 4-character group dictionary, then first three characters ch1, ch2 and ch3 are searched in 3-character group dictionary. If found store the index value of that position in the compressed file. The ch4 is now stored in ch1 and next 3 characters are read in ch2, ch3 and ch4 and Step 2 is repeated.

Step 4: If those 3 characters are not found in the 3-character group dictionary, then first two characters ch1 and ch2 are searched in the 2-character group dictionary. If found store the index value of that position in the compressed file. The ch3 and ch4 are now stored in ch1 and ch2 respectively and next two characters are read from the source file in ch3 and ch4 and Step 2 is repeated.

Step 5: If those 2 characters are not found in the 2-character group dictionary, then the first character ch1 is stored as it is in the compressed file. The ch2, ch3 and ch4 are now stored in ch1, ch2 and ch3 respectively and next one character is read from the source file in ch4 and Step 2 is repeated.

In this way the entire file is scanned and compressed. The flowchart for compressing the character groups is shown in figure 5.1.

Decompression

Step 1: Read the dictionary of 4-character group, 3-character group and 2-character group.

Step 2: Read code of one byte from the compressed file.

Step 3: If the code value is less than 128, it means that a normal ASCII character was stored during compression process. Store that code value in the decompressed file as it is and repeat Step 2.

Step 4: If the code value is greater than or equal to 128 and less than or equal to 159, then it indicates the index value of 2-character group dictionary. Subtract the biased value 128 from it and get the two characters from the 2-character group dictionary and repeat Step 2.

Step 5: Read another byte of code and combine it previous read code byte to form 16-bit index value. If this index value is in between 40960 and 41951, then it indicates the index value of 3-character group dictionary. Subtract the biased value of 40960 from it and get the 3 characters from the 3-character group dictionary and repeat Step 2.

Step 5: If the value of 16-bit code is greater than or equal to 49152, then it indicates the index value of 4-character group dictionary. Subtract the biased value of 49152 from it and get the 4 characters from the 4-character group dictionary and repeat Step 2.

Thus the entire file is scanned byte by byte and is decompressed. The flow chart for decompression is shown in figure 5.2.

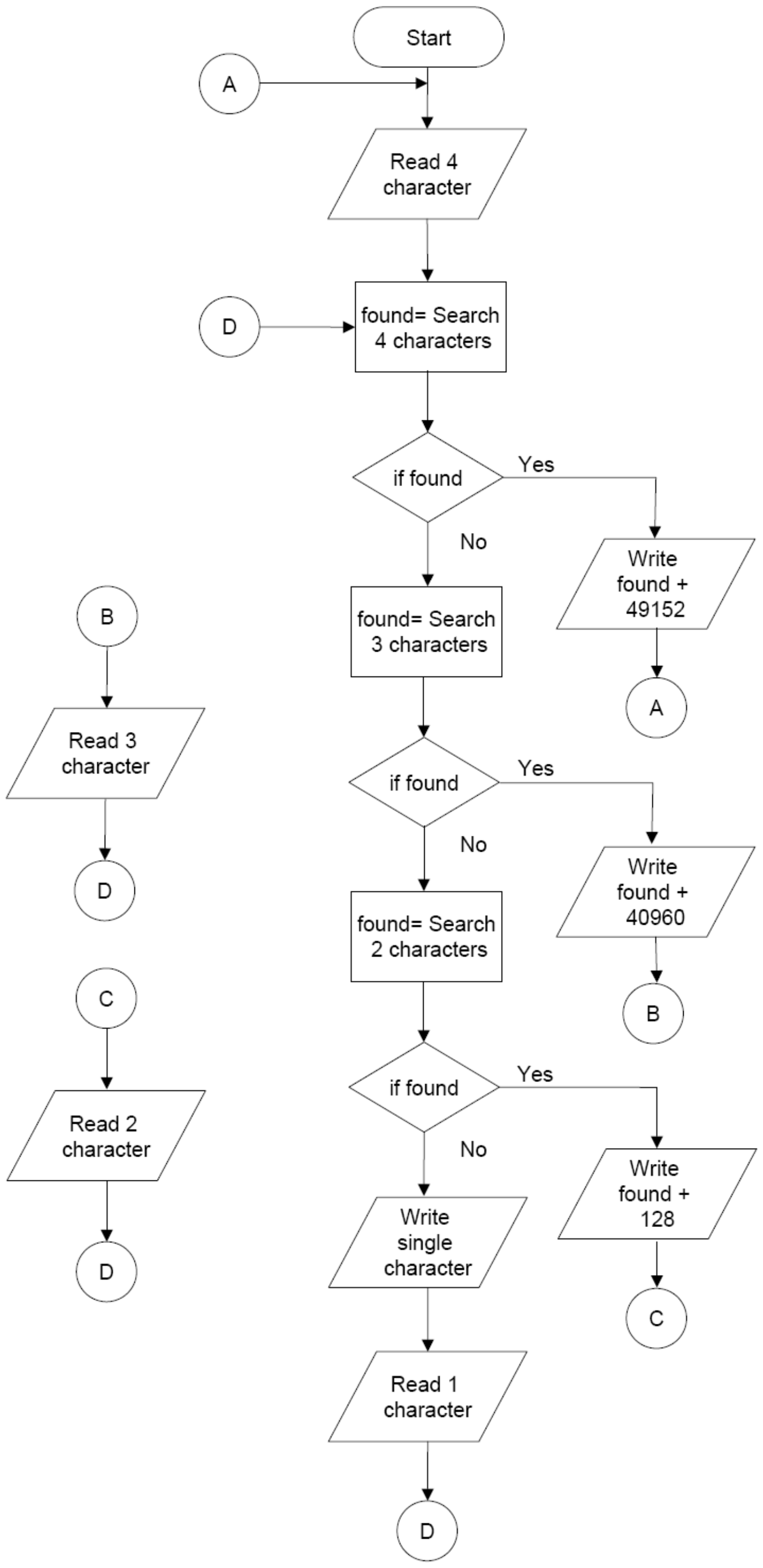

Figure 5.1 Flowchart for compression (CBTC-B)

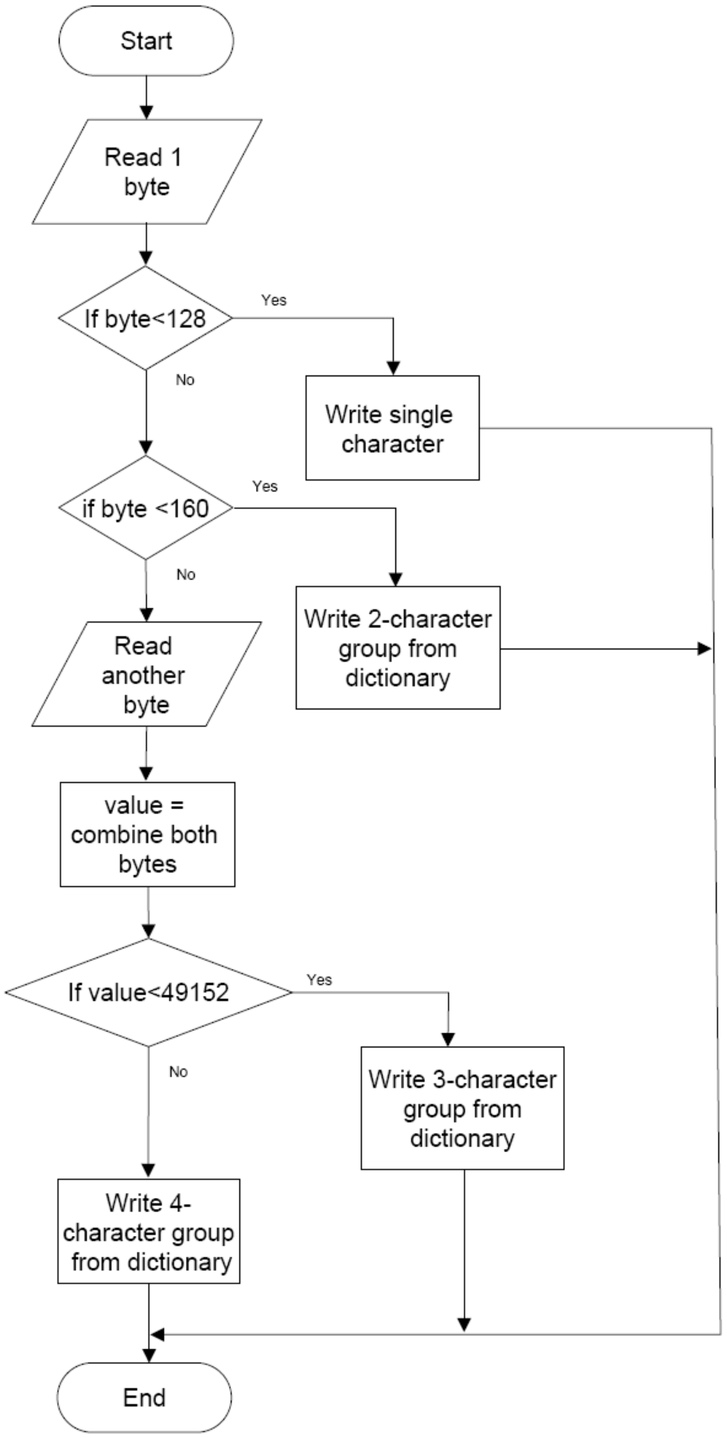
Start
Read 1 byte
If byte<128
Yes
No
Write single character
if byte <160
Yes
No
Write 2-character group from dictionary
Read another byte
value = combine both bytes
If value<49152
Yes
No
Write 3-character group from dictionary
Write 4-character group from dictionary
End

Figure 5.2 Flow chart for decompression (CBTC-B)

5.3 IMPLEMENTATION OF WBTC-A

In this method, the dictionaries of full words and partial words are created from the source file to be compressed. The words having frequency greater than 1 are considered. The remaining words with frequency of 1 then are used to create dictionaries of partial words.

Creation of Word Dictionary

At the outset, we have to decide how many words shall be stored in the dictionary. As pointed in the algorithm 4.3.3, the words stored in the dictionary will be encoded by their position in the dictionary. The ASCII character comprises the code value from 0 through 127. Therefore it requires only 7-bits to store it in memory. After studying various corpuses (for e.g. Gutenberg, Enronsent, European Parliament, Large and Etext files), it is found that at the most there are 20000 to 25000 words, 3000 to 5000 prefix and suffix partial words, in file of size varying from 2 MB to 10 MB. Hence to encode the 25000 words it requires bits in between 8 to 16. Therefore we decided to encode the words in the dictionary with 16-bit, including 1-bit to differentiate between the normal ASCII character and the encoded index value of the words position in the dictionary. Considering the above situation we have only 15-bits left to store the index value, hence we can store only 32K words i.e. 2^{15} words in the dictionary. Therefore we decide to keep 4000 Prefix partial words, 4000 Suffix partial words, and 24000 full words to store in the dictionary. Remaining 768 codewords are kept reserved for non-words.

The words of length greater than 2 are only taken into consideration for storing in the dictionary, because if we consider words of length 2 in the dictionary, then we won’t achieve any compression, as to store words of 2 characters, it will require 2 bytes and also to encode the word in the dictionary, it will require 2 bytes. The same thing applies to partial prefix and suffix words. The length of partial prefix and suffix words is kept to minimum of 3 characters.

Step 1: Declare a constant DICTCONSTANT with value 24000. Allocate the memory for the *dictionary* using statement

char dictionary[DICTCONSTANT][50];

The size of the *dictionary* considered here is having maximum 32768 words, comprising of 24000 full words and each of 4000 prefix and suffix partial words.

Step 2: Read the words from the source file and add it to the dictionary.

After reading the word from the source file, the length of the word is checked and if greater than 2, then the *addword*() function is invoked to check whether the word exists in the *dictionary*. If exist, then the count value of the word is increased by 1 else the word is added to the dictionary and the count value is set to 1. This process is shown in Flowchart in fig. 5.3.

Step 3: After scanning all the words from the source file to be compressed, sort the *dictionary* with respect to the count values of the word in the descending order by calling the *sort*() function. The *sort*() functions is implemented by linear sorting technique.

Step 4: Keep the words with frequency more than or equal to 2 in the *dictionary*.

The entire *dictionary* created in Step 3 is scanned till the first word with frequency of 1 is encountered. If the total number of words with frequency 2 or more exceeds DICTCONSTANT, then only DICTCONSTANT words are kept in the dictionary and remaining words are used for creating partial words. Any how the words stored in the dictionary are less than or equal to DICTCONSTANT. The index value of the first word with frequency 1 is stored in variable *startofonelengthword*, or else if the total number of words with frequency 2 or more exceeds DICTCONSTANT, then the value DICTCONSTANT is stored in *startofonelengthword*. The size of the dictionary is stored in variable *trackdictionary*.

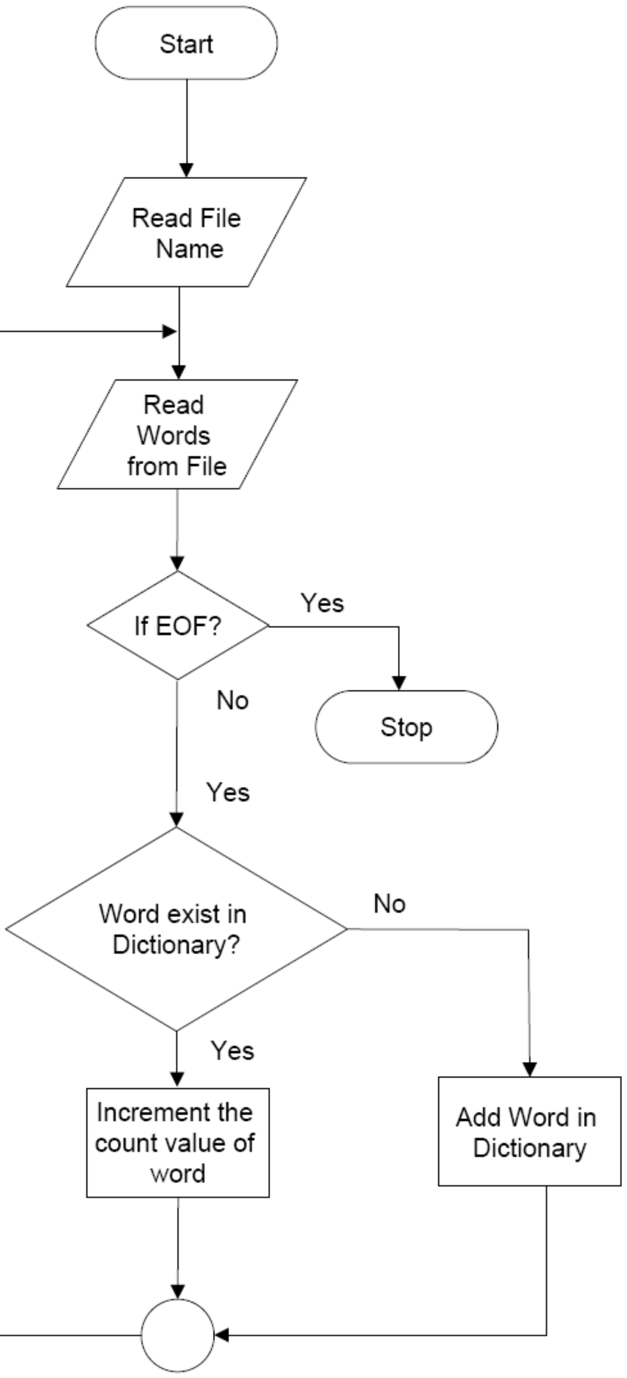

Figure 5.3 Flow chart of creating word dictionary (WBTC-A)

Creation of Partial words (Prefix and Suffix) Dictionary

Here the dictionaries of partial words either from starting (Prefix) or from end (Suffix) of the words are taken into consideration.

Step 1: Declare a constant PREFIXCONSTANT with value 4000. Allocate the memory for the dictionary using statement. Maximum 4000 words are stored in the prefix dictionary.

Step 2: The words are scanned in the reverse order i.e. from last word of the dictionary towards the first words of the *dictionary* of frequency 1. The characters from the words are compared with characters from every other word. The maximum number of characters matched is considered and that sub-word is stored in the prefix dictionary. The flowchart of creating the prefix dictionary is shown in figure 5.4.

***Example*:** Consider that words '*complicate*', '*complications*', '*complicated*' are having frequency count 1. Then at first, the last word *complicated* will be compared with word *complications*, it is seen that first 9 characters *complicat* are matching, then again the word '*complicated*' is compared with word 'complicate', now it is seen that first 10 characters of *complicated* are matched with *complicate*, so instead of *complicat*, the word *complicate* will be store in the partial word prefix dictionary.

Step 3: The maximum prefix words stored in the prefix dictionary are checked with PREFIXCONSTANT, if less then that count value is kept, otherwise PREFIXCONSTANT value is stored at the maximum count of partial words stored in the dictionary.

Similarly, the suffix dictionary is created. The only difference is that the characters are compared from first character to last character in the prefix dictionary, but in suffix dictionary, the characters will be compared from last character to first character.
Example: Consider the words '*welcome*', '*awesome*', '*outcome*', are having frequency count 1. Then at first, the last word *outcome* will be compared with word *awesome,* it is seen that last 3 characters *ome* are matching, then again the word '*outcome*' is compared with word '*welcome*', now it is seen that last 4 characters of *welcome* are matched with *outcome*, so instead of *ome*, the word *come* will be store in the partial word suffix dictionary.

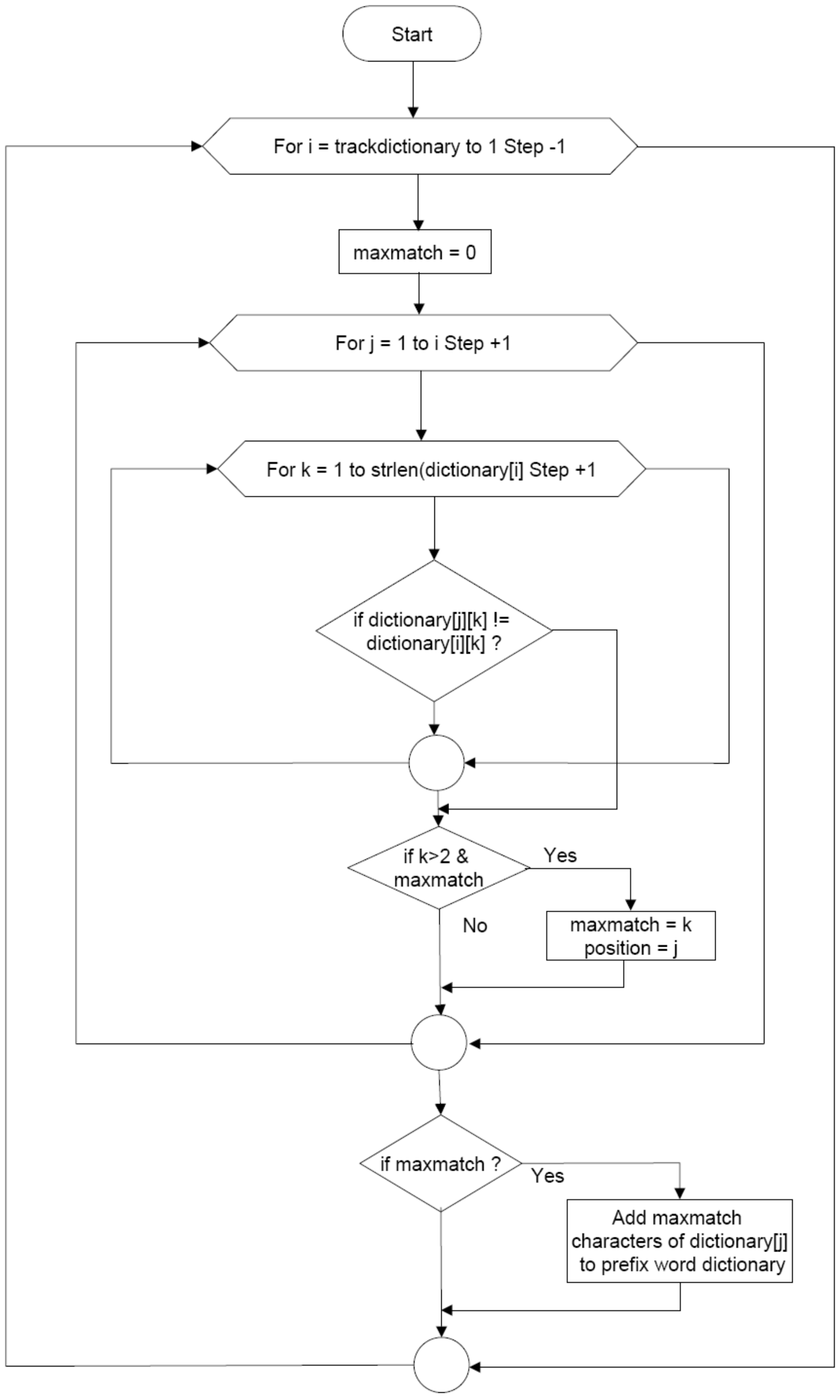
Start
For i = trackdictionary to 1 Step -1
maxmatch = 0
For j = 1 to i Step +1
For k = 1 to strlen(dictionary[i] Step +1
if dictionary[j][k] != dictionary[i][k] ?
if k>2 & maxmatch
Yes
No
maxmatch = k
position = j
if maxmatch ?
Yes
Add maxmatch characters of dictionary[j] to prefix word dictionary

Figure 5.4 Flowchart of creating the prefix word dictionary (WBTC-A)

Compression

The compression process starts with reading the source file again. Only the words which are found in the word dictionary and in the prefix and suffix dictionary are coded with index value of the dictionary and the remaining characters are stored as it is in the compressed file.

Step 1: The words read one by one are stored in the string variable *str*. If the length of *str* is greater than two than only it is search in the dictionary otherwise it is written as it is in the compressed file.

Step 2: If the word is found in the word dictionary, then the index value of the word in the dictionary is store in the compressed file after adding the constant 32768 to it. The function *OutputBits*() is used to write the bits in the compressed file, and the process repeats for next word. The value 32768 is added to the index value, so that the MSB of 16 bit index value will be ‘1’, which will differentiate between the normal ASCII character and the index value of the word placed in the dictionary.

Step 3: If the word is not found in the dictionary, then first the word is scanned in the prefix dictionary. The word is compared with every word of the prefix dictionary and the longest match of prefix word found is considered. Then the index value of the prefix part is store in the compressed file after adding the constant 32768+24000 i.e., 56768. Again the same function *OutputBits*() is used to store the bits in the compressed file. Now the prefix part is removed from the *str*. Now the *str* contains only remaining portion of the word.

Step 4: Now the *str* is scanned for the suffix part of the word in the suffix dictionary. If not found then the entire *str* is written as it is in the compressed file. Else characters other than suffix part are first stored as it is and then the index value of the suffix word is stored in the compressed file after adding 32768+24000+4000 i.e. 60768. The function *OutputBits*() is used to store the bits in the compressed file. The flowchart of writing the word is shown in figure 5.5.

Example: Suppose the prefix word dictionary contains words such as '*seme*', '*sing*', '*singer*'. And the suffix dictionary contains words such as '*ing*', '*ter*' '*one*'.

Now the word, say '*semester*' is to be compressed, then first the word *semester* will be compared with word '*seme*', in this case there is match of first 4 characters. Again the word '*semester*' is compared with next word 'sing', here there is a match of only first character of the word '*semester*'. Hence the index value of '*seme*' will be written to the compressed file. The remaining *str* of the word '*semester*', after removing the partial word '*seme*' now becomes the string '*ster*'. This word 'ster' is now searched in the suffix dictionary. It is clear that only 3 characters are matched when compared with all the words of the suffix dictionary and it is '*ter*'. Now before storing the index value of the suffix word '*ter*' in the compressed file, we have to check whether all characters from the *str* is taken or some characters are left. In our example, after writing *seme* and before writing '*ter*', the character which is left is 's', so first this 's' will be written in the compressed file as it is. And then the index value of the '*ter*' will be written to the compressed file. Thus the word semester is compressed by first writing the index value of the prefix partial word '*seme*' and then writing 's' as it is, and finally the index value of the '*ter*' is written to the compressed file.

Thus, compression is achieved when the word is found in the dictionary, and when the sub-words are found in prefix and/or suffix dictionary.

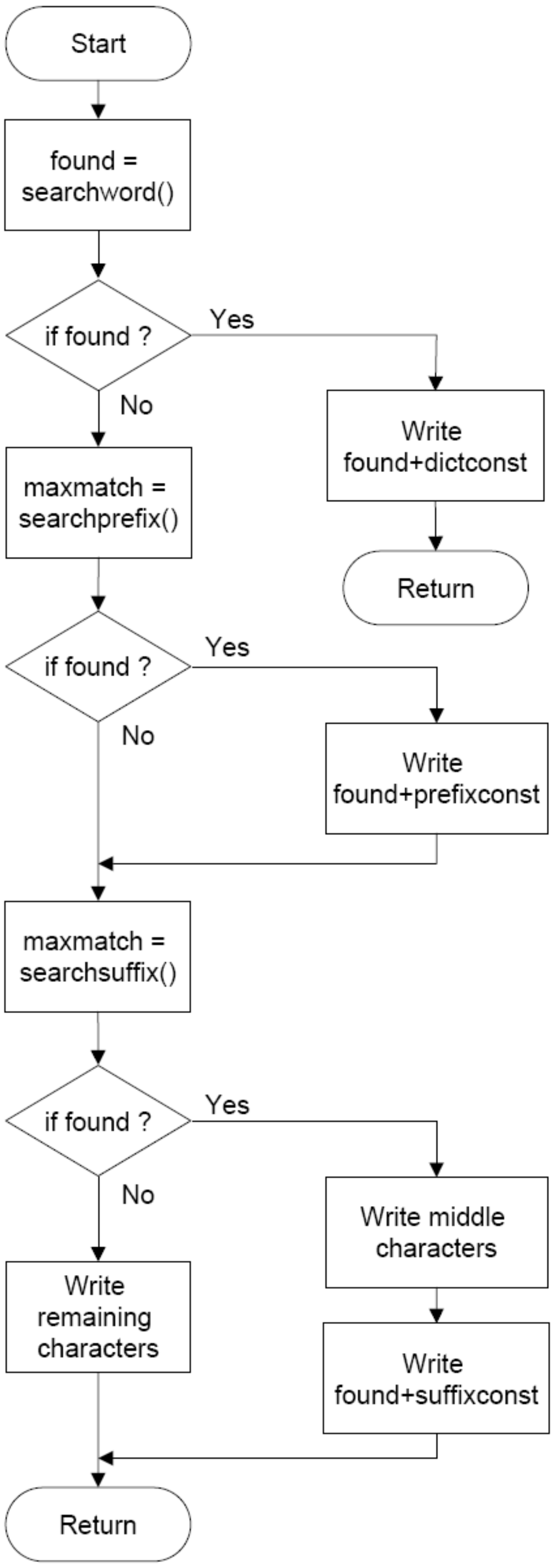

Figure 5.5 Flowchart for compressing a word (WBTC-A)

Decompression

The process of decompression is simple and fast. The flowchart of the decompressing a single normal character and decompressing a single word is shown in figure 5.6. The file to be decompressed is open in binary read mode. At first, one byte is read and is compared with value 128. If it is less than 128, then the read byte is normal character and it is written as it is in the decompressed file. If the value of that byte is greater than or equal to 128 then the read byte is a part of the encoded word of 16-bit. To form the complete value of encoded word another byte is read from the file and a 16-bit value is formed which indicates the offset of the word in the dictionary. This 16-bit value is first compared with the range of offset of suffix dictionary i.e. in between 60768 and 64768. if it is in that range, then the suffix word is retrieved from the suffix dictionary and written to the decompressed file. If the range is in between 56768 and 60768 then the prefix word is retrieved from the prefix dictionary and written to the decompressed file, otherwise the value indicates the offset of the full word of the word dictionary. Then that word is retrieved and written to the decompressed file. In this way the entire file is scanned and the file is decompressed.

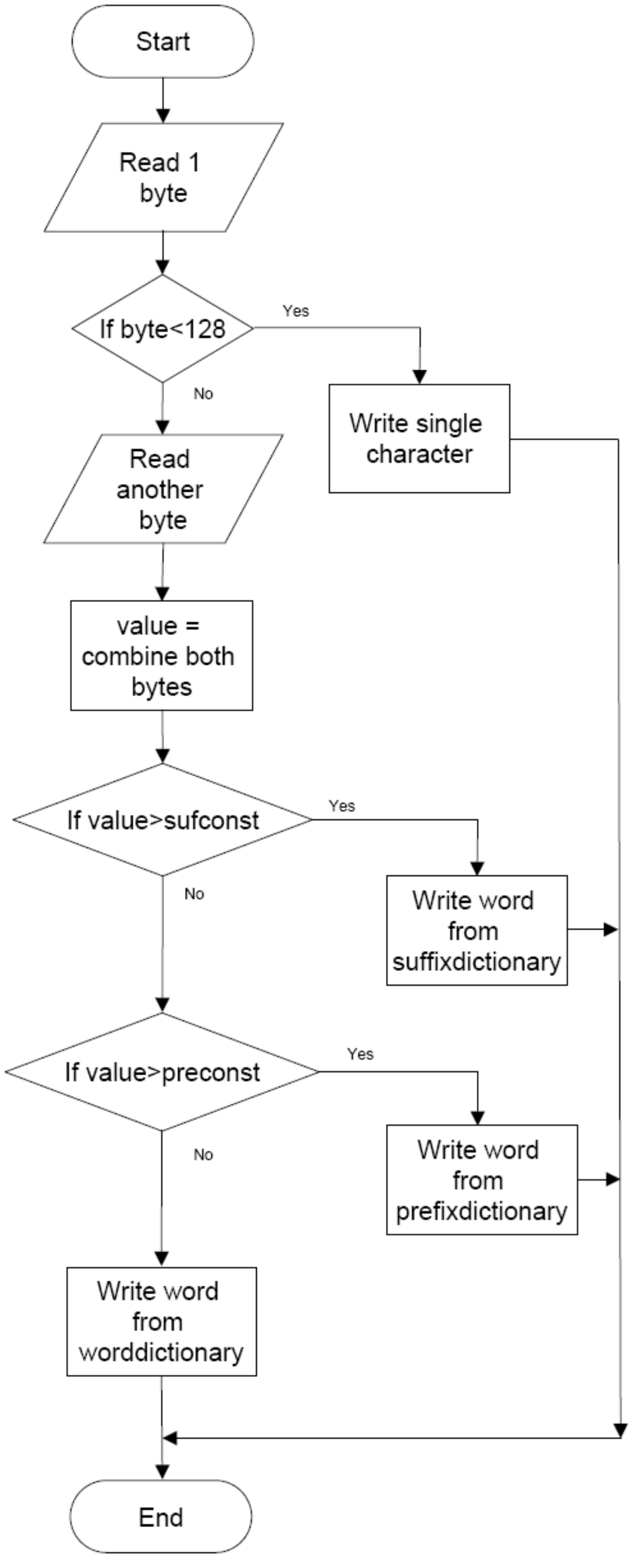
Start
Read 1 byte
If byte<128
Yes
No
Write single character
Read another byte
value = combine both bytes
If value>sufconst
Yes
No
Write word from suffixdictionary
If value>preconst
Yes
No
Write word from prefixdictionary
Write word from worddictionary
End

Figure 5.6 Flowchart of decompressing a normal character or word (WBTC-A)

5.4 IMPLEMENTATION OF WBTC-B

In this method, the dictionary is created of words in single dimension array. The words are separated by symbol '#' in the dictionary. We have used here symbol '#' as a separator because we had defined word as sequence of ASCII characters only, therefore symbol '#' won't occur in word, and if '#' symbol is occurring in the source file then it will be stored as it is. The '#' symbol is used in the dictionary only and not in the compressed file. It is used simply to distinguish between the words in the dictionary only. In this method, simultaneously the dictionary is created and the file is compressed. This method is very simple, but introduced here to compare it with another method WBTC-C, which is using the two-dimensional dictionary.

Dictionary Creation and Compression

Step 1: The word is read from the source file.

Step 2: The word read from the source file is searched in the dictionary, if found the index value of that word is written in the compressed file. Searching of the word is explained next.

Step 3: If the word read is not found then, first the word is added to the dictionary and then the index value of that word is written in the compressed file.

Searching Word in Dictionary

Step 1: Calculate the length of the word to be searched.

Step 2: Scan for the word separator character '#'.

Step 3: Compare all characters of the entire word with the characters succeeding the symbol '#". If all the characters are matched then check the next character in the dictionary, if it is '#' (i.e. the full word is matched), then the word is said to be found. If the next character is not '#' than it means that the partial word of the dictionary is matched with the word read from the source file and hence the index value of the word in the dictionary cannot be stored in the compressed file. In this case the word read from the source file is added to the dictionary and the separator symbol '#' is also added to the end of the dictionary to indicate the end of the word.

Example

If the words in the dictionary are say '*welcome#become#*' etc., and the word read from the file is '*be*'. In word '*become*' the first two characters are matching with the word '*be*', so one can say that the word is found in the dictionary, but that is not the case, because after ‘*be*’ instead of terminating symbol ‘#’, there are some characters in the word ‘*become*’ from the dictionary, hence we cannot say that complete word ‘*be*’ is found as it is in the dictionary. If word ‘*become*’ is read form the source file, then only we can say that the word is found in the dictionary.

So if word is not found in the dictionary then that word is added to the end of the dictionary with the ‘#’ separator in between. The flowchart of compressing the word is shown in figure 5.7.

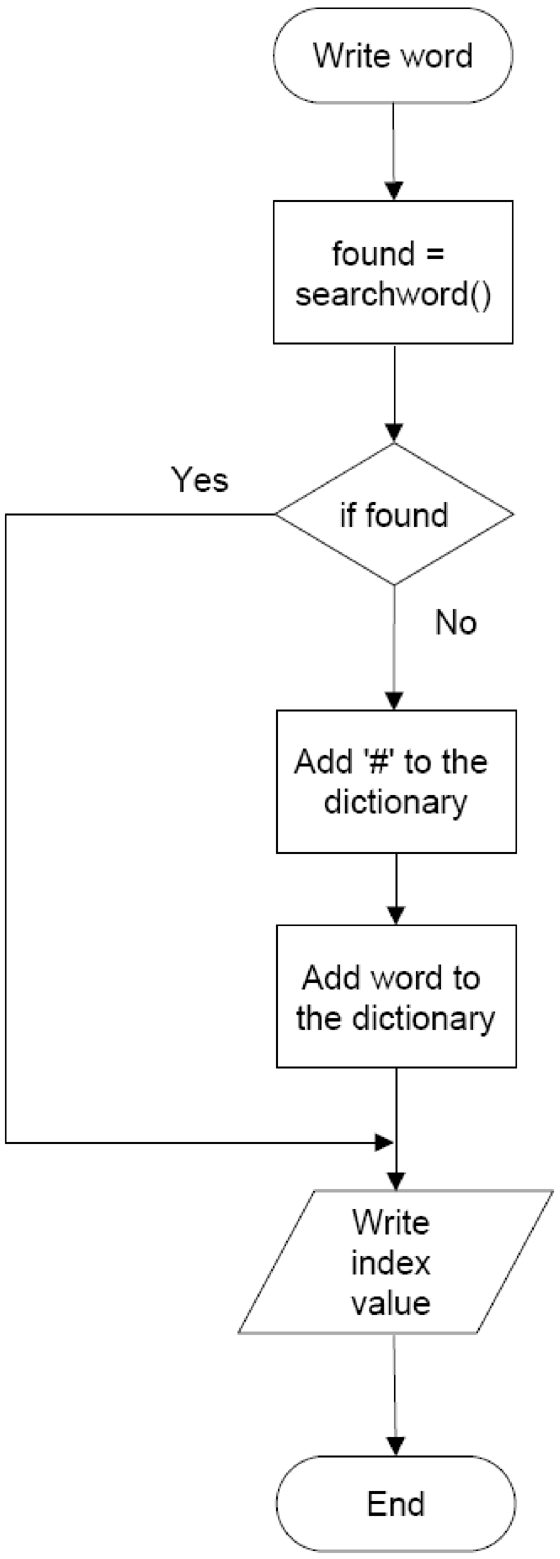

Figure 5.7 Flowchart for compressing a word (WBTC-B)

Decompression

Step 1: Read the dictionary of words from the dictionary file.

Step 2: Read code of one byte from the compressed file.

Step 3: If the code value is less than 128, it means that a normal ASCII character was stored during compression process. Store it in the decompressed file as it is and repeat Step 2.

Step 4: If the code value is greater than or equal to 128 then read another byte of code and combine it with previous read code byte to form 16-bit index value. Subtract the bias value 32768 form it. This 16-bit index value is now pointing to the corresponding word in the dictionary.

Step 5: Scan the dictionary, increment the counter of the '#' symbol every time it encounters till it matches the 16-bit index value. Read the characters from the dictionary from that point till another '#' symbol encounter to indicate the end of the word.

Thus the entire file is scanned byte by byte and is decompressed. The flowchart for decompressing the word is shown in figure 5.8.

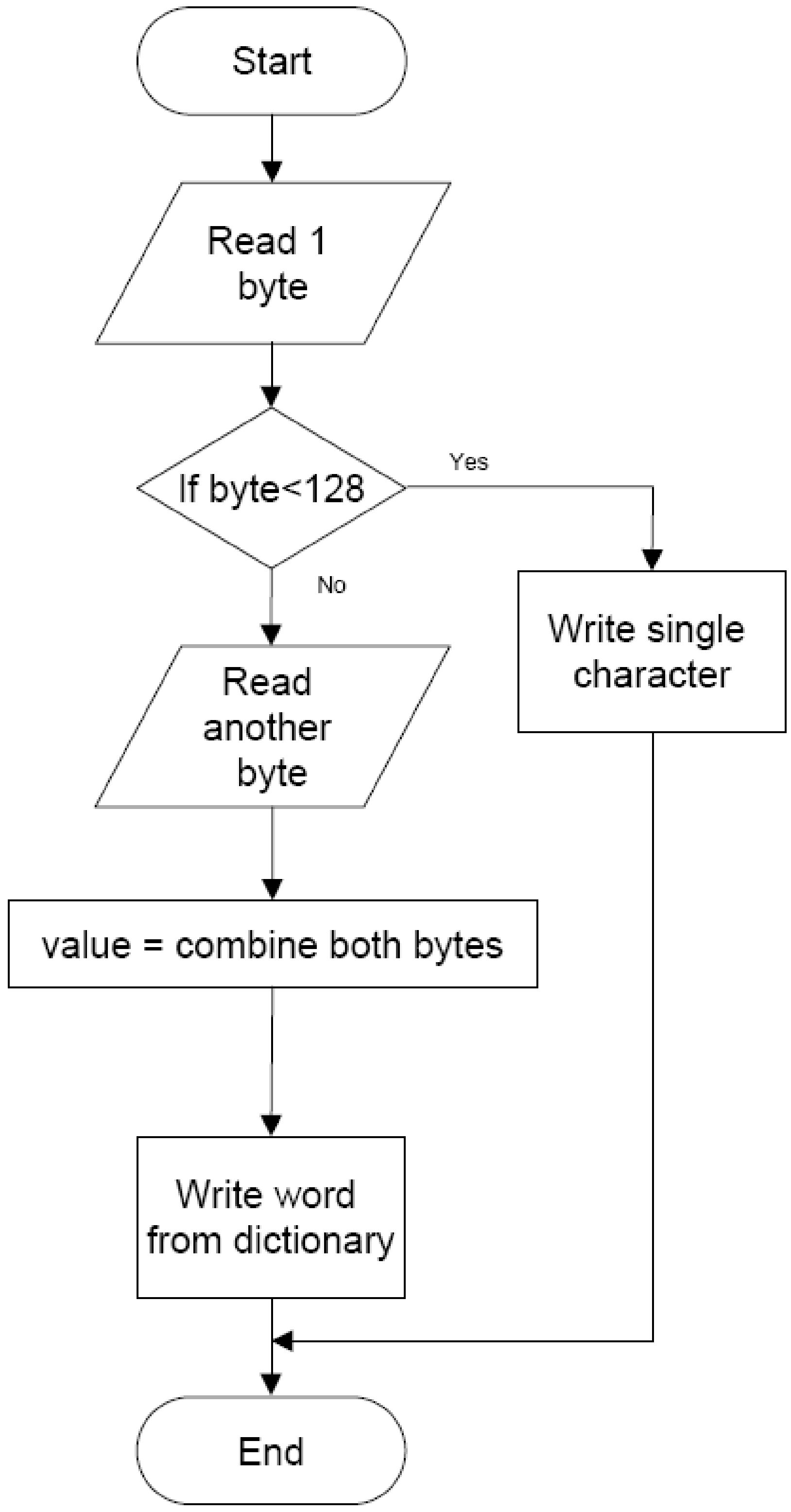

Figure 5.8 Flowchart of decompressing a normal character or word (WBTC-B)

5.5 IMPLEMENTATION OF WBTC-C

In this method two-dimension semi-dynamic dictionary is created. Thus, as explained in chapter 4, the length of the code assigned to word reduces from 16-bit to 8-bit. The dictionary structure is shown in figure 4.7 (Chapter 4). The total numbers of words to be stored in the dictionary are 16447.

Compression

Step 1: The word is read from the source file. The single characters or non-alphabetic characters, or words with length less than 3 are stored in the compressed file as it is.

Step 2: The word read from the source file is searched in the dictionary, if found the index value of that word is computed and written in the compressed file.

Computing the index value

If the word is found in first 63 words, then the index value is simply stored in the compressed file. If the word is found at a position greater than or equal to 63, then the row number and column number are computed as below:

$$\textit{rownumber} = (\textit{found} - 63) \; / \; 64$$

$$\textit{columnnumber} = (\textit{found} - 63) \bmod 64$$

If the previous row number and the current row number are same, then only the column number is written in the compressed file, else the escape symbol (0xFF) is written in the compressed file followed by the new row number and the column number.

Step 3: If the word is not found in the dictionary, then it is stored as it is in the compressed file.

Thus the entire file is scanned and compressed. The flowchart of compressing a single word is shown in figure 5.9.

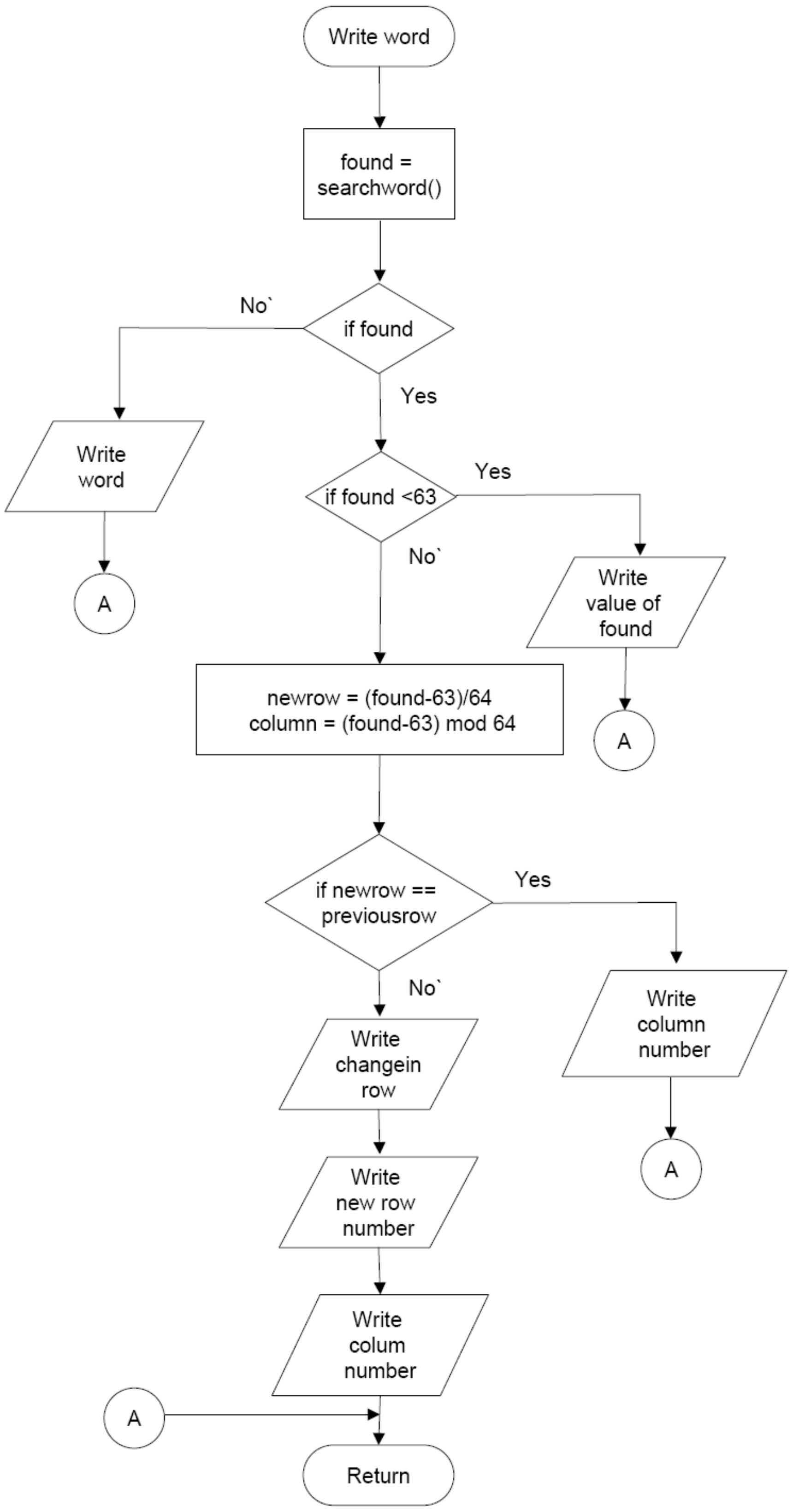

Figure 5.9 Flowchart for compressing a word (WBTC-C)

Decompression

Step 1: Read the dictionary of words from the dictionary file.

Step 2: Read code of one byte from the compressed file.

Step 3: If the code value is less than 128, it means that a normal ASCII character was stored during compression process. Store it in the decompressed file as it is and repeat Step 2.

Step 4: If the code value is greater than or equal to 128, check if the escape symbol is there for change in row. If yes, then read another two consecutive bytes for getting new row number and column number respectively. Subtract bias value 128 from the code to get actual row number and column number.

Step 5: Now calculate the position of the word in the dictionary from these row number and column number by the equation

$$\textit{Position} = (\textit{row number} * 64) + 63 + \textit{column number}$$

Thus word at that position is read from the dictionary and stored in the decompressed file. Thus the entire file is scanned byte by byte and is decompressed. The flowchart of decompressing a normal character or compressed word is shown in figure 5.10.

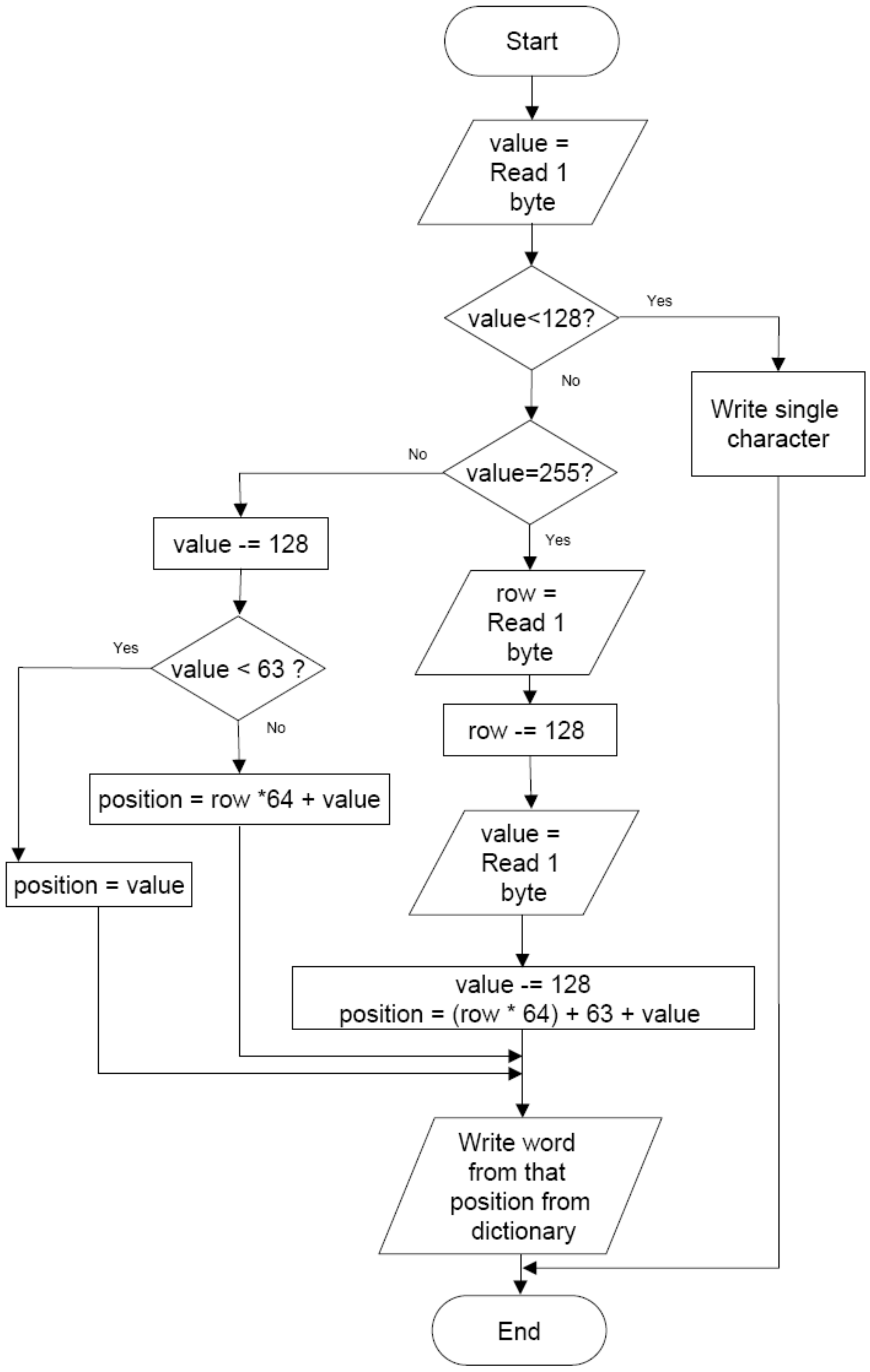

Figure 5.10 Flowchart of decompressing a normal character or word (WBTC-C)

5.6. IMPLEMENTATION OF WBTC-D

In the above methods described, the dictionary is built explicitly and is stored external to 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.

Compression

Step 1: Create a null dictionary

Step 2: Read the words from the source file. The single characters or non-alphabetic characters, or words with length of 2 are stored in the compressed file as it is.

Step 3: Search the word in the dictionary, if not found write the word as it is in the compressed file and add that word to the dictionary.

Step 4: If word is found in the dictionary, then write the index value of the word in the compressed file after adding 32768 to it i.e., making the MSB bit of 16-bit index value to 1. This MSB will differentiate it from the normal ASCII code.

Thus the entire file is scanned and compressed. The flowchart for compressing a word is shown in figure 5.11

Decompression

Step 1: Create the null dictionary.

Step 2: Read code of one byte from the compressed file.

Step 3: If the code value is less than 128, it means that a normal ASCII character was stored during compression process. Store it in the decompressed file as it is and repeat Step 2.

Step 4: If the code value is greater than or equal to 128. If yes, then read another byte for getting the complete two byte index value. Subtract bias value 32768 from the code to get actual position of the word in the dictionary. Store that word from the dictionary to the decompressed file.

Thus the entire file is scanned byte by byte and is decompressed. The flowchart for decompressing a normal character or compressed word is shown in figure 5.12.

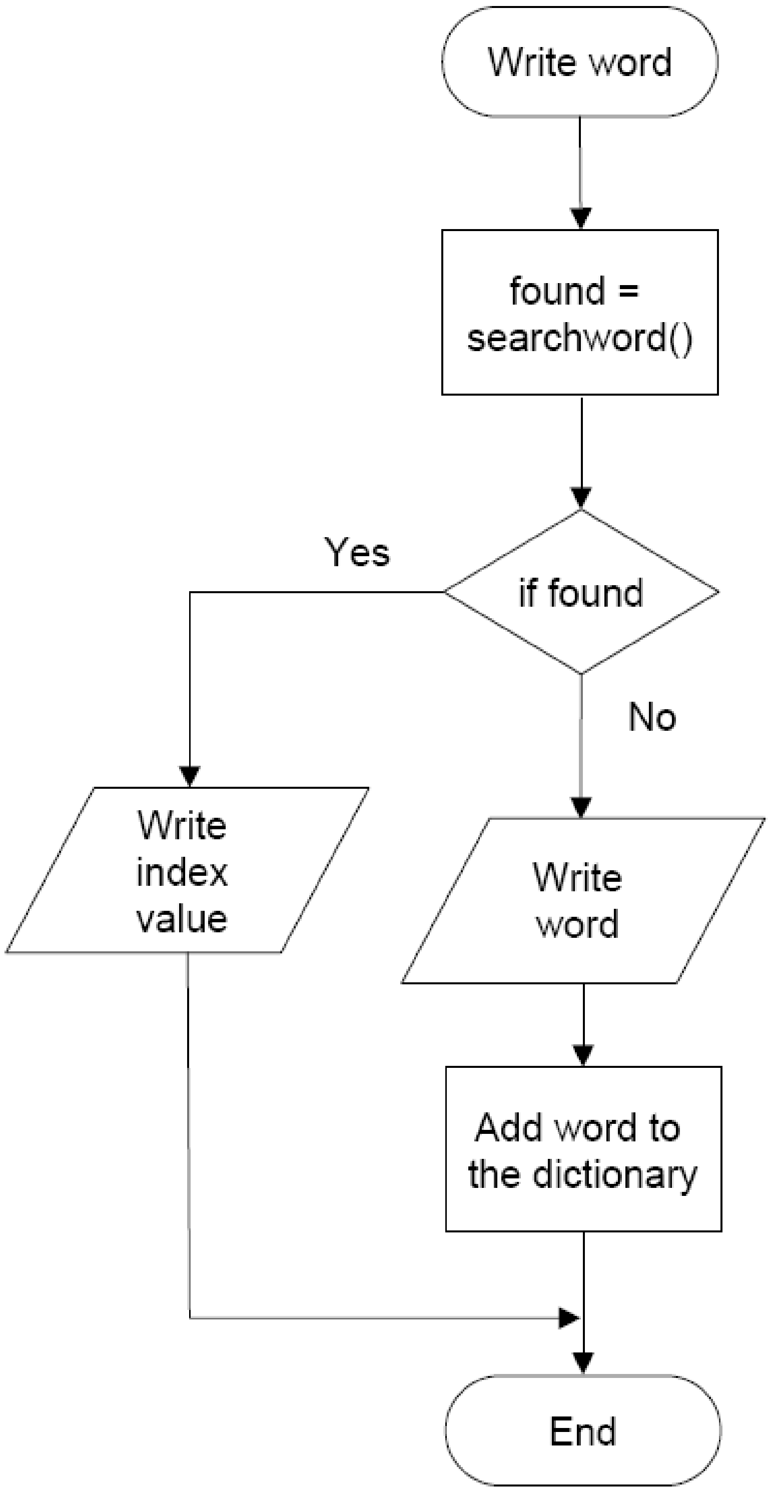

Figure 5.11 **Flowchart for compressing a word (WBTC-D)**

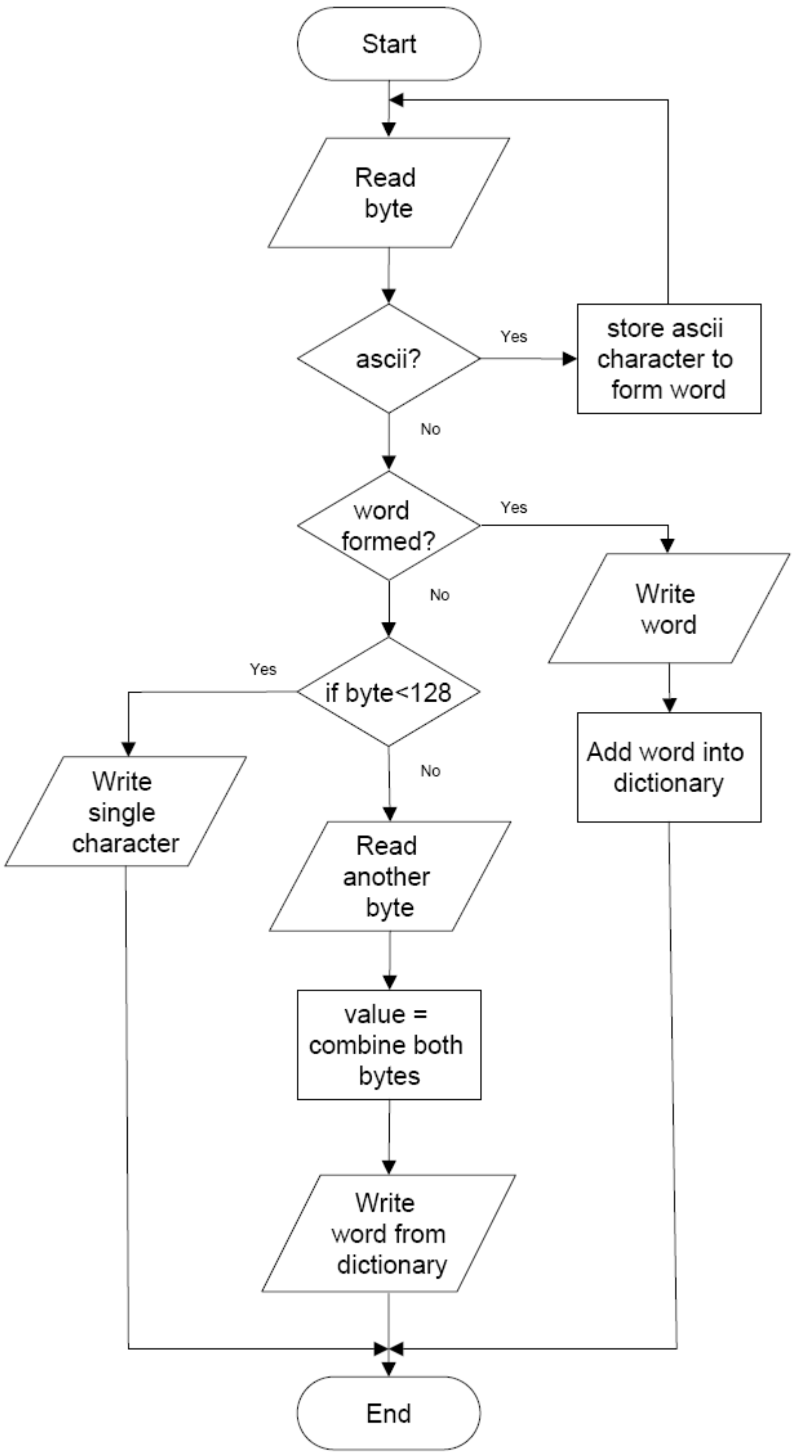

Figure 5.12 Flowchart of decompressing a normal character or word (WBTC-D)

5.7. IMPLEMENTATION OF WBTC-E

In this method the static dictionary is created from the set of different corpus. This method is equivalent to method 5.5, but the only difference here is that in this method the dictionary is static and will not be considered as overhead but as a part of compression program, whereas in method 5.5 the dictionary is created for that particular file and is considered as a part of compressed file, thereby increasing the overhead.

Creating Dictionary

Step 1: Prepare a list of files from the different corpus. Here we are considering files from the Gutenberg corpus, Enronsent Corpus, Etextfile corpus and large corpus.

Step 2: First of all the dictionary of all the words of frequency greater than 2 from each individual file is created.

Step 3: Then all the dictionary are merged into single dictionary, which consists of all words from the entire dictionaries of all files and their frequency count is also added.

Step 4: The dictionary is then sorted according to the frequency count in the descending order.

Here again we consider to code the index value of the position of the word in the dictionary by 16-bit, i.e., MSB is reserved for indicating the difference between the normal ASCII character and the index value of the word. We are getting only 15-bits to encode the index value of the word from the dictionary. The same idea used in method 5.4.2 is adopted here with a difference in the number of bits required to encode the index value of the word. In method 5.4.2 only 8-bits were used to encode the index value, whereas in this method 16-bit index value is used and therefore the capacity of dictionary storing the words increases from 16,477 to 1,62,816. How this figures come is explained below:

The dictionary is considered as two dimensional dictionary. There are 256 rows and in each row we can have 32768 words. If we decide to keep 0xFF as an escape symbol for change in row of the dictionary, then we cannot use combination of 0xFF XX as a 16-bit combination for storing the index value, where XX varies in between 0x00 to 0xFF. Therefore, the total combination which can be store in 16-bit index value is 0x80 00 through 0xFEFF i.e. total word which we can store in a row is 32511. We decide here to keep most frequent 32000 words common in each row and remaining 511 words to be unique in 256 rows. So the total number of words will be 32000 + (256 * 511) = 162816.

Step 1: The word is read from the source file. The single characters or non-alphabetic characters, or words with length of 2 are stored in the compressed file as it is.
Step 2: The word read from the source file is searched in the dictionary, if found the index value of that word is computed and written in the compressed file.

Computing the index value

If the word is found in first 32000 words, then the index value is simply stored in the compressed file. If the word is found at a position greater then or equal to 32000, then the row number and column number are computed as below:

rownumber = (found – 32000) / 511

columnnumber = (found – 32000) mod 511

If the previous row number and the current row number are same, then only the column number is written in the compressed file, else the escape symbol (0xFF) is written in the compressed file followed by the new row number and the column number.
Step 3: If the word is not found in the dictionary, then it is stored as it is in the compressed file.
Thus the entire file is scanned and compressed. The flowchart for compressing the single word is shown in figure 5.13.

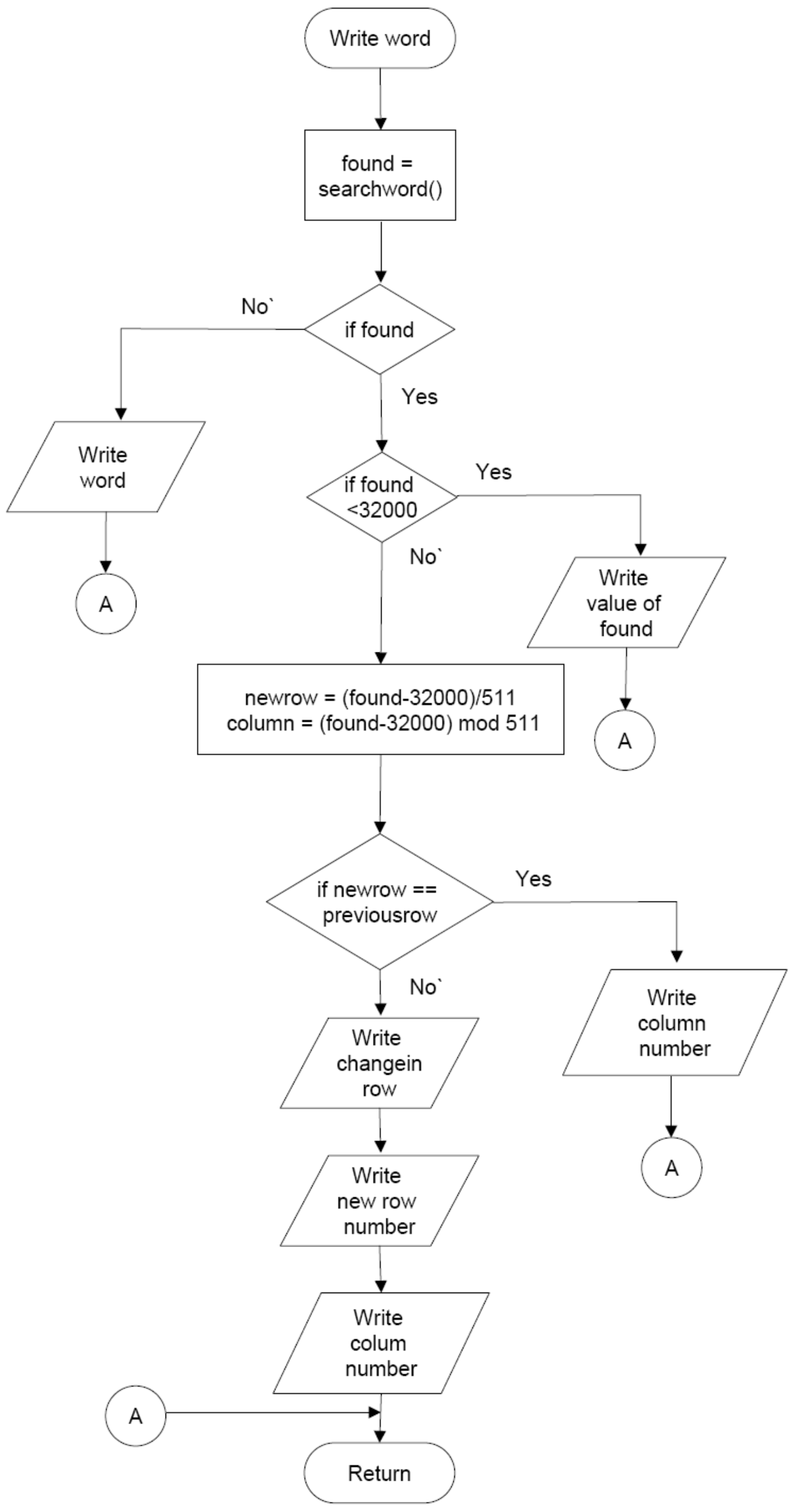

Figure 5.13 Flowchart for compressing a word (WBTC-E)

Decompression

Step 1: Read the dictionary of words from the dictionary file.

Step 2: Read code of one byte from the compressed file.

Step 3: If the code value is less than 128, it means that a normal ASCII character was stored during compression process. Store it in the decompressed file as it is and repeat Step 2.

Step 4: If the code value is greater than or equal to 128, check if the escape symbol is there for change in row. If yes, then read another three consecutive bytes for getting new row number and column number respectively.

Step 5: Now calculate the position of the word in the dictionary from these row number and column number by the equation

$$\text{Position} = (\text{row number} * 511) + 32000 + \text{column number}$$

Thus word at that position is read from the dictionary and stored in the decompressed file. Thus the entire file is scanned byte by byte and is decompressed. The flowchart for decompressing a normal character or compressed word is shown in figure 5.14.

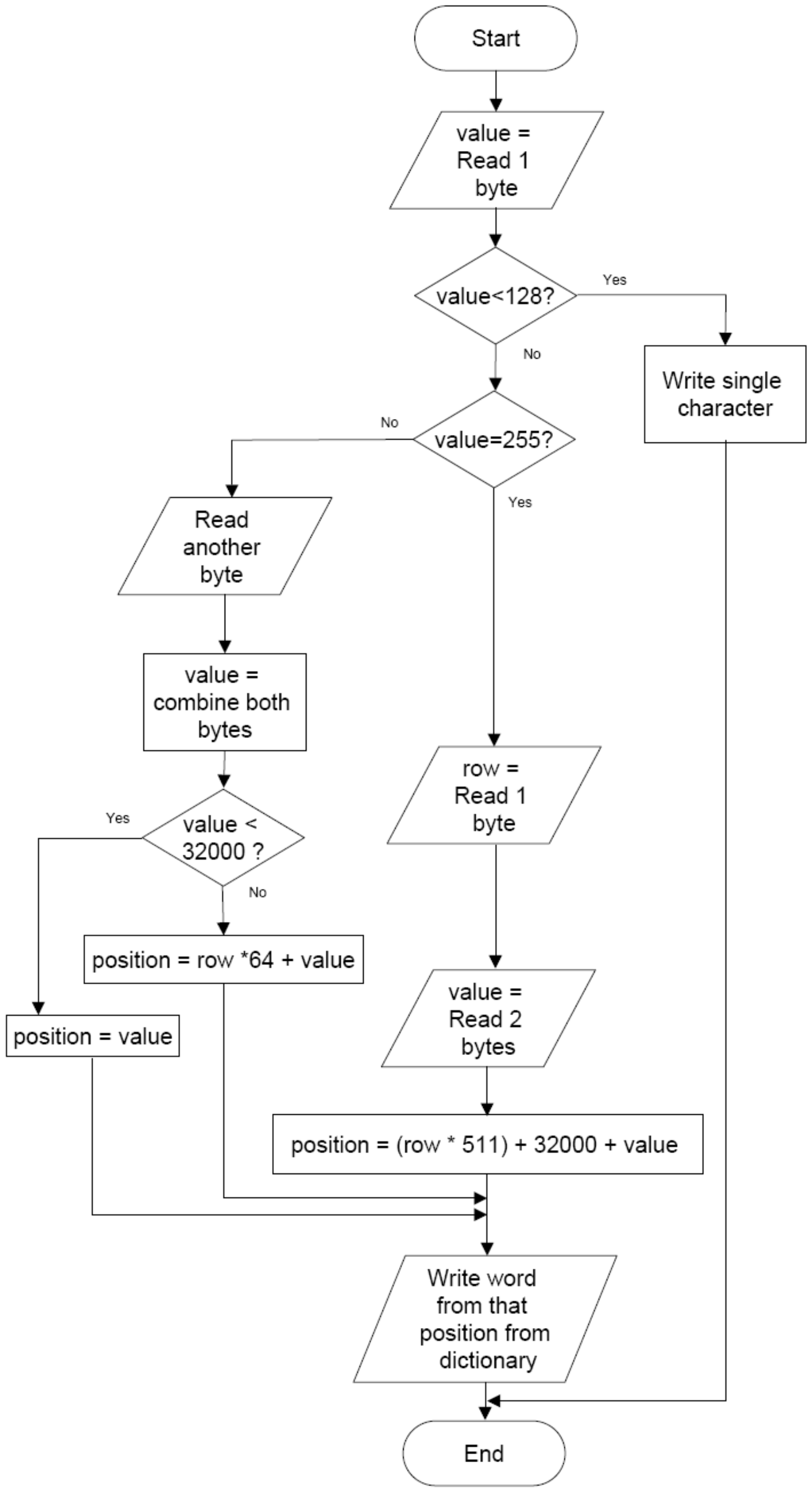

Figure 5.14 Flowchart of decompressing a normal character or word (WBTC-E)

5.8. IMPLEMENTING SEARCHING IN COMPRESSED FORM

Besides improving the compression ratio, our intention was to develop a technique in such a way so that it can be useful for searching the phrases from the compressed file directly without decompressing it. Thus, if the phrases are to be searched in the compressed files (relatively smaller than normal files), then number of comparison to be done will be comparatively less and thus time taken will be less. In the proposed methods only WBTC – D is not suitable for searching the phrases directly in the compressed form, because it is using the dynamic dictionary and hence it is not possible to search the phrases without decompressing it. All other methods are useful for searching the phrases without decompressing the compressed file.

In Chapter 3, we have seen different string matching algorithms. We have used Karp – Rabin, Knuth – Morris – Pratt, Brute – Force , Boyer – Moore and Quick Search algorithm for searching the phrases directly in the compressed form. The results are shown in Chapter 6.

Following are the steps required to search the phrases directly in the compressed form irrespective of any of the above methods.

Step 1: Read the phrase to be searched.

Step 2: Compress the phrase use the compression technique which is used to compress the file in which we have to search the phrase. This compression process is little different from the compression process of the source file. Here, there is no need to create the dictionaries as in the normal case to compress the file, instead the source dictionaries of that compressed files are read and accordingly the compression is done. In all the methods above the dictionaries required are different in different cases.

Step 3: Perform any standard phrase – matching algorithm (Karp – Rabin, Knuth – Morris – Pratt, Brute – Force, Boyer – Moore and Quick Search Algorithm).

The searching of phrase directly in the compressed form is possible because of the encoding technique we have adopted in our methods. We have used always either 8-bit encoding or 16-bit encoding, like in some other techniques such as Huffman wherein the number of bits are different for different characters. The byte boundary is always maintained.

CHAPTER 6

EXPERIMENTAL RESULTS

6
EXPERIMENTAL RESULTS

6.0 OUTLINE OF THIS CHAPTER

The character based algorithm and word based algorithm described in this thesis have been implemented and tested on several text files. For comparison purposes, we were primarily concerned with Bzip2 version 1.0.2, a version of PPM called PPMD and PPMII, LZMA as a benchmark. PPMd and PPMII were run in order-4 with 10 MB memory limit. LZMA was run with dictionary size of 8MB. The compression ratios are expressed in percentage (%). All tests were carried out on Microsoft Windows XP, Intel Pentium processor 1.60 GHz and 256 MB RAM. The overall result is that performance is consistently better than the benchmark excluding where the size of the file is small (few hundred Kilobytes) in some cases. At the end of this chapter the testing of searching the phrases directly in the compressed form is done using Karp-Rabin algorithm, Knuth-Morris-Pratt algorithm and Brute-Force algorithm. The overall number of comparisons required to search the phrase in the compressed form is much less than that if searched in the normal (decompressed) file. The time taken for searching is also comparatively less than in the case of normal file. The results shows that average decompression time of word based method (WBTC-C) is less than time taken by Bzip2 to decompress.

6.1 LIST OF FILES USED FOR TESTING

6.1.1. E-Text Corpus

In this corpus the E-Text files are taken from the corpus from the internet. There are total 9 files taken in this corpus. The list of the files and their size is given in Table 6.1.

Table 6.1 File Information of E-Text Corpus

Sr.No	Name of File	Size (Bytes)
1	Burroughst.txt	9,923,450
2	Dickens.txt	4,186,334
3	Doyle.txt	4,273,166
4	Emerson.txt	2,654,470
5	Hawthorne.txt	2,534,557
6	Irving.txt	3,159,365
7	Kant.txt	4,966,366
8	Milton.txt	2,392,226
9	Plato.txt	3,044,145
	Total Size in Bytes	**37,134,079**

6.1.2. European Parliament Corpus

In this corpus 10 files are taken of equal size. The basic corpus was of 3 GB size. The corpus was broken into files each of 5MB size. Out of those files 10 files are selected. The list of the files and their size is given in Table 6.2.

Table 6.2 File Information of European Parliament Corpus

Sr.No	Name of File	Size (Bytes)
1	europarl1	5,242,880
2	europarl2	5,242,880
3	europarl3	5,242,880
4	europarl4	5,242,880
5	europarl5	5,242,880
6	europarl6	5,242,880
7	europarl7	5,242,880
8	europarl8	5,242,880
9	europarl9	5,242,880
10	europarl10	5,242,880
	Total Size in Bytes	**52,428,800**

6.1.3. Enronsent Corpus

19 different files are taken from this corpus. The list of the files and their size is given in Table 6.3.

Table 6.3 File Information of Enronsent Corpus

Sr.No	Name of File	Size (Bytes)
1	enronsent00	1,977,255
2	enronsent01	1,747,204
3	enronsent02	2,556,530
4	enronsent03	2,128,069
5	enronsent04	2,450,083
6	enronsent05	2,049,856
7	enronsent06	1,978,373
8	enronsent07	2,566,926
9	enronsent08	2,064,851
10	enronsent09	2,124,877
11	enronsent10	2,246,081
12	enronsent11	1,826,427
13	enronsent12	2,035,599
14	enronsent13	1,991,442
15	enronsent14	1,962,961
16	enronsent15	1,673,901
17	enronsent16	1,716,634
18	enronsent17	1,610,800
19	enronsent18	1,646,888
Total Size in Bytes		**38,354,757**

6.1.4. Project Gutenberg Corpus

8 different files are taken from this corpus. The list of the files and their size is given in Table 6.4.

Table 6.4 File Information of Project Gutenberg Corpus

Sr.No	Name of File	Size (Bytes)
1	leonard	1,423,740
2	pg1342	704,139
3	pg1399	2,039,729
4	pg2981	6,840,209
5	pg3207	1,254,848
6	pg33	517,294
7	pg514	1,053,432
8	pg76	597,586
Total Size in Bytes		**14,430,977**

6.1.5 Mixed Corpus

5 files are taken from different corpus viz., Large Corpus, Gutenberg Corpus, Enronsent Corpus. The list of the files and their size is given in Table 6.5.

Table 6.5 File Information of Mixed Corpus

Sr.No	Name of File	Size (Bytes)
1	bible.txt	4,047,392
2	world192.txt	2,473,400
3	anne11.txt	258,420
4	enronsent02	2,556,530
5	pg10.txt	4,445,256
Total Size in Bytes		**13,780,998**

6.1.6 Summary of Corpus

The summary of all corpus is given in Table 6.6.

Table 6.6 Summary of all Corpus.

Sr.No	Corpus	Org Size
1	E-Text	37,134,079
2	Europarl	52,428,800
3	Enronsent	38,354,757
4	Gutenberg	14,430,977
5	Mixed	13,780,998
Total Size in Bytes		**156,129,611**

6.2 COMPARISON OF WORD BASED METHODS WITH BZIP2

6.2.1. Compression Statistics of E-Text Corpus

Table 6.7 Compression ratios for E-Text Corpus (Bzip2)

		Compression ratio in %					
Sr. No	Name of file	Bzip2	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	burroughst	26.15	23.73	23.74	24.64	23.54	23.46
2	dickens	28.05	27.56	27.33	27.13	26.62	25.98
3	doyle	27.17	25.55	25.41	25.37	25.05	24.26
4	emerson	28.85	28.57	28.16	27.43	26.93	25.88
5	hawthorne	26.72	25.92	25.71	25.51	25.16	24.12
6	irving	27.02	18.86	18.77	20.36	19.82	17.66
7	Kant	20.31	18.33	18.40	18.62	18.38	17.92
8	milton	15.06	15.21	14.91	14.45	15.01	13.56
9	Plato	24.57	19.66	19.66	19.75	20.29	18.96
	Total	25.16	22.85	22.75	23.04	22.59	21.81

The test was executed on E-Text corpus and the results are shown in Table 6.7. For all the files our all methods outperforms Bzip2. WBTC-A achieves average gain of 2.31%.

WBTC-B achieves average gain of 2.41%. WBTC-C achieves average gain of 2.12%. WBTC-D achieves average gain of 2.57%. WBTC-E achieves average gain of 3.35%.

6.2.2. Compression Statistics of European Parliament Corpus

Table 6.8 Compression ratios for European Parliament Corpus (Bzip2)

		Compression ratio in %				
Sr. No	Name of file	Bzip2	WBTC-A	WBTC-B	WBTC-C	WBTC-D
1	europarl1	22.17	21.18	20.92	20.83	20.21
2	europarl2	21.75	20.87	20.63	20.49	19.93
3	europarl3	22.23	21.28	21.05	20.90	20.39
4	europarl4	22.05	21.12	20.86	20.76	20.18
5	europarl5	21.99	21.02	20.75	20.72	20.13
6	europarl6	22.18	21.13	20.88	20.80	20.20
7	europarl7	22.33	21.32	21.02	20.99	20.38
8	europarl8	21.65	20.65	20.37	20.30	19.70
9	europarl9	22.18	21.19	20.93	20.83	20.22
10	europarl10	21.91	20.94	20.68	20.59	19.99
	Total	22.04	21.07	20.81	20.72	20.13

The test was executed on European Parliament corpus and the results are shown in Table 6.8. For all the files our all methods outperforms Bzip2. WBTC-A achieves average gain of 0.97%. WBTC-B achieves average gain of 1.23%. WBTC-C achieves average gain of 1.32%. WBTC-D achieves average gain of 1.91%.

6.2.3. Compression Statistics of Enronsent Corpus

Table 6.9 Compression ratios for Enronsent Corpus (Bzip2)

		Compression ratio in %					
Sr. No	Name of file	Bzip2	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	enronsent00	26.45	26.12	25.79	25.34	25.28	23.65
2	enronsent01	26.50	27.04	26.80	26.13	26.01	24.32
3	enronsent02	24.39	23.70	23.44	23.07	23.16	21.74
4	enronsent03	25.56	25.27	24.99	24.40	24.30	22.84
5	enronsent04	23.32	23.00	22.79	22.17	22.18	20.74
6	enronsent05	24.31	23.51	23.19	23.10	22.78	21.30
7	enronsent06	25.78	25.22	24.95	24.52	24.38	22.80
8	enronsent07	23.34	22.41	22.17	21.95	22.05	20.34
9	enronsent08	24.35	24.06	23.75	23.28	23.19	21.55
10	enronsent09	11.86	12.28	12.19	11.99	12.15	10.98
11	enronsent10	21.89	21.64	21.39	21.03	21.03	19.66
12	enronsent11	23.53	23.25	23.06	22.93	22.83	21.37
13	enronsent12	25.40	25.63	25.32	24.64	24.70	22.99
14	enronsent13	25.04	24.81	24.58	23.93	23.72	22.38
15	enronsent14	25.88	26.22	25.85	25.19	25.14	23.59
16	enronsent15	21.31	20.53	20.39	20.38	20.33	18.74
17	enronsent16	24.59	25.02	24.68	24.16	24.26	22.17
18	enronsent17	26.19	26.00	25.50	25.18	25.00	22.99
19	enronsent18	25.54	25.22	24.91	24.61	24.73	22.18
	Total	23.87	23.62	23.35	22.94	22.90	21.30

The test was executed on Enronsent corpus and the results are shown in Table 6.9. For all the files our all methods outperforms Bzip2. WBTC-A achieves average gain of 0.25%. WBTC-B achieves average gain of 0.52%. WBTC-C achieves average gain of 0.93%. WBTC-D achieves average gain of 0.97%. WBTC-E achieves average gain of 2.57%.

6.2.4. Compression Statistics of Project Gutenberg Corpus

Table 6.10 Compression ratios for Project Gutenberg Corpus (Bzip2)

		Compression ratio in %					
Sr.No	Name of file	Bzip2	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	leonard	28.59	29.93	29.49	28.48	27.61	26.33
2	pg1342	25.89	27.06	26.65	26.35	25.53	24.17
3	pg1399	26.33	26.02	25.78	25.54	25.02	24.19
4	pg2981	26.24	24.77	24.62	24.65	24.22	23.86
5	pg3207	25.79	26.76	26.27	25.72	24.89	24.01
6	pg33	28.95	32.09	31.28	30.08	28.78	26.24
7	pg514	28.41	29.35	28.97	28.38	27.40	26.15
8	pg76	27.88	29.64	29.25	28.73	27.77	26.74
	Total	26.75	26.54	26.26	25.97	25.33	24.55

The test was executed on Project Gutenberg corpus and the results are shown in Table 6.10. For all the files our all methods outperforms Bzip2. WBTC-A achieves average gain of 0.21%. WBTC-B achieves average gain of 0.49%. WBTC-C achieves average gain of 0.78%. WBTC-D achieves average gain of 1.42%. WBTC-E achieves average gain of 2.20%. From Table 6.10, it is seen the compression ratio deteriorates for files of smaller size as compared to files of larger size. This is because of the overhead of the dictionary associated with the compressed file.

6.2.5. Compression Statistics of Mixed Corpus

Table 6.11 Compression ratios for Mixed Corpus (Bzip2)

		Compression ratio in %					
Sr. No	Name of file	Bzip2	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	bible.txt	20.89	20.02	19.82	20.05	19.66	19.36
2	world192.txt	19.79	19.56	19.23	18.55	18.19	17.42
3	anne11.txt	29.71	32.93	32.36	30.87	29.58	27.20
4	enronsent02	24.39	23.70	23.44	23.07	23.16	21.74
5	pg10.txt	22.52	21.91	21.77	21.88	21.48	21.24
	Total	22.04	21.47	21.25	21.13	20.82	20.20

The test was executed on Mixed Corpus and the results are shown in Table 6.11. For all the files our all methods outperforms Bzip2. WBTC-A achieves average gain of 0.57%. WBTC-B achieves average gain of 0.79%. WBTC-C achieves average gain of 0.91%. WBTC-D achieves average gain of 1.22%. WBTC-E achieves average gain of 1.84%. From Table 6.11, it is seen the compression ratio deteriorates for files of smaller size as compared to files of larger size. This is because of the overhead of the dictionary associated with the compressed file.

6.2.6. Compression Statistics of all Corpus

Table 6.12 Compression ratios for all Corpus (Bzip2)

		Compression ratio in %					
Sr. No	Name of Corpus	Bzip2	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	E-Text	25.16	22.87	22.76	23.05	22.60	21.82
2	Europarl	22.04	21.07	20.81	20.72	20.13	-----*
3	Enronsent	23.87	23.62	23.35	22.94	22.90	21.30
4	Gutenberg	26.75	26.54	26.26	25.97	25.33	24.55
5	Mixed	22.04	21.47	21.25	21.13	20.82	20.20
	Total	23.67	22.66	22.44	22.34	21.94	21.79

The overall compression ratio for the corpus is summarized in Table 6.12 and the results are shown in Table 6.12. For all the files our all methods outperforms Bzip2. WBTC-A achieves average gain of 1.01%. WBTC-B achieves average gain of 1.23%. WBTC-C achieves average gain of 1.33%. WBTC-D achieves average gain of 1.73%. The average compression ratio of Bzip2 method without European Parliament corpus is 24.49% therefore, WBTC-E achieves average gain of 2.70%.

* - For WBTC-E method, the European parliament corpus was not considered, as it was not in the same domain as compared to other corpus

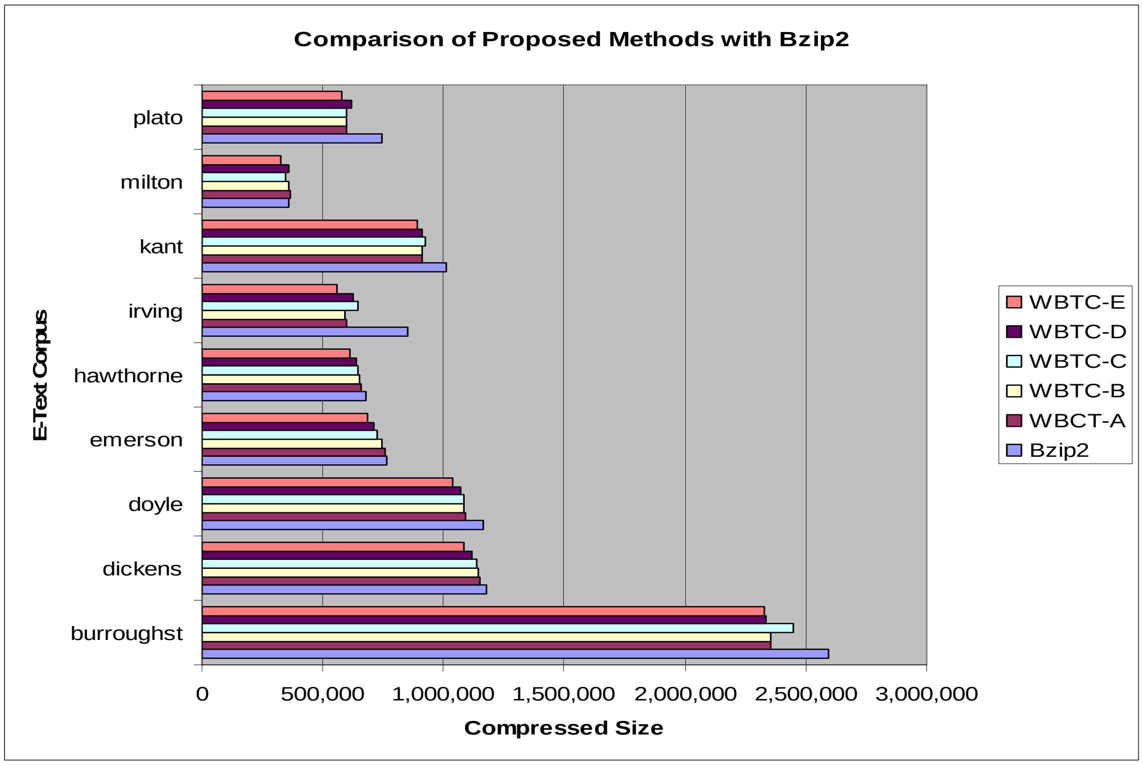

Figure 6.1 Compression ratios of E-Text Corpus (Bzip2)

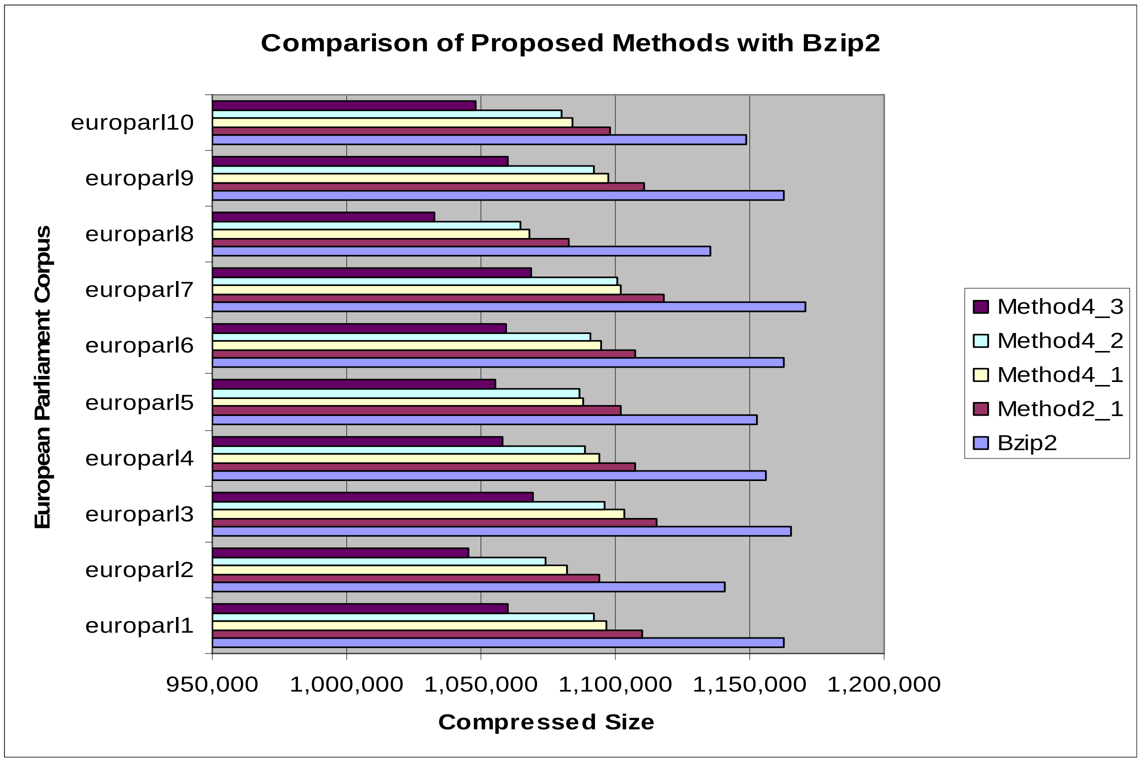

Figure 6.2 Compression ratios of European Parliament Corpus (Bzip2)

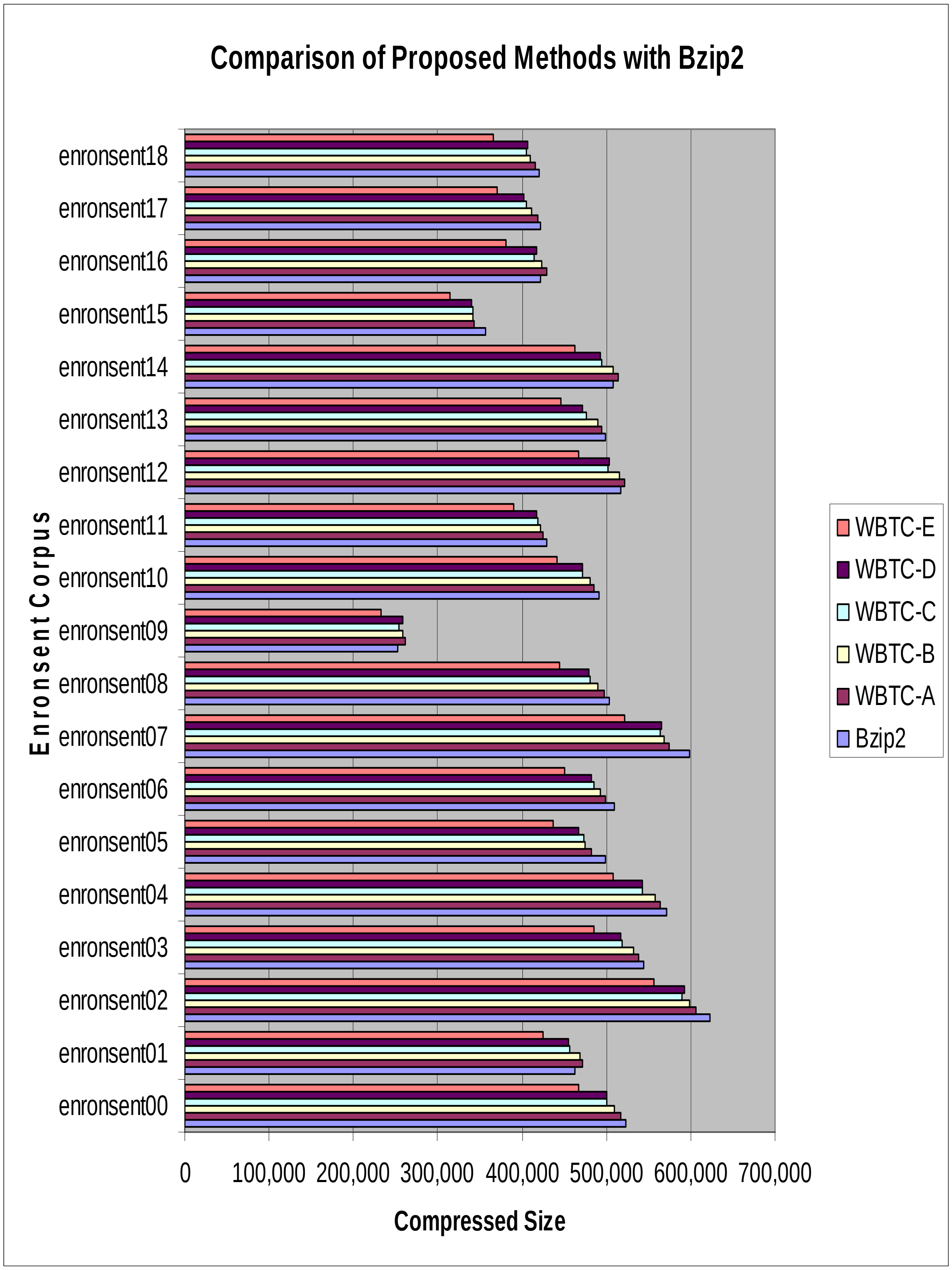

Figure 6.3 Compression ratios for Enronsent Corpus (Bzip2)

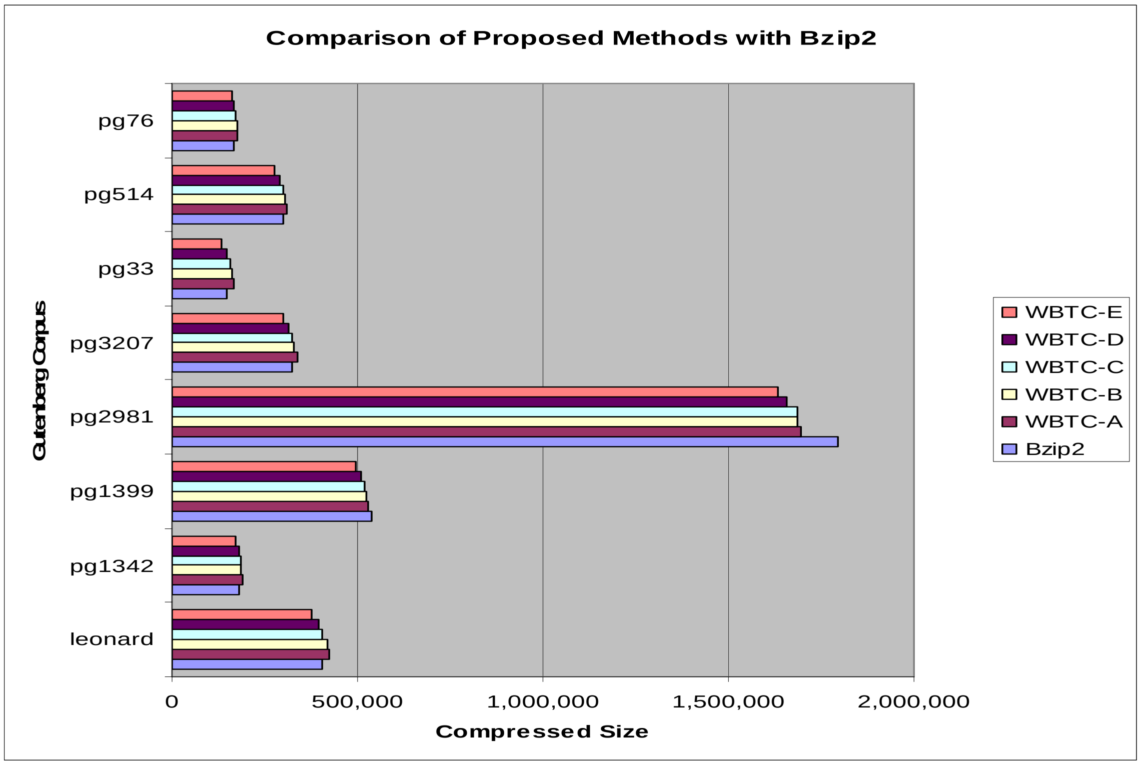

Figure 6.4 Compression ratios for Project Gutenberg Corpus (Bzip2)

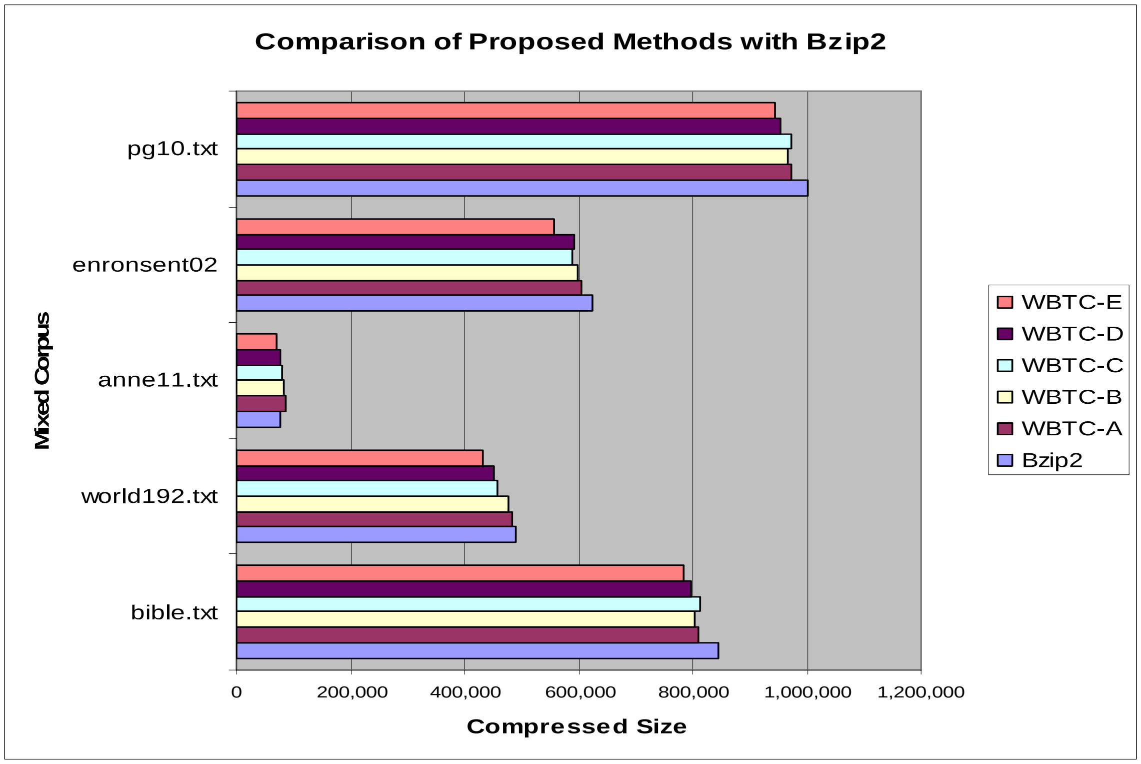

Figure 6.5 Compression ratios for Mixed Corpus (Bzip2)

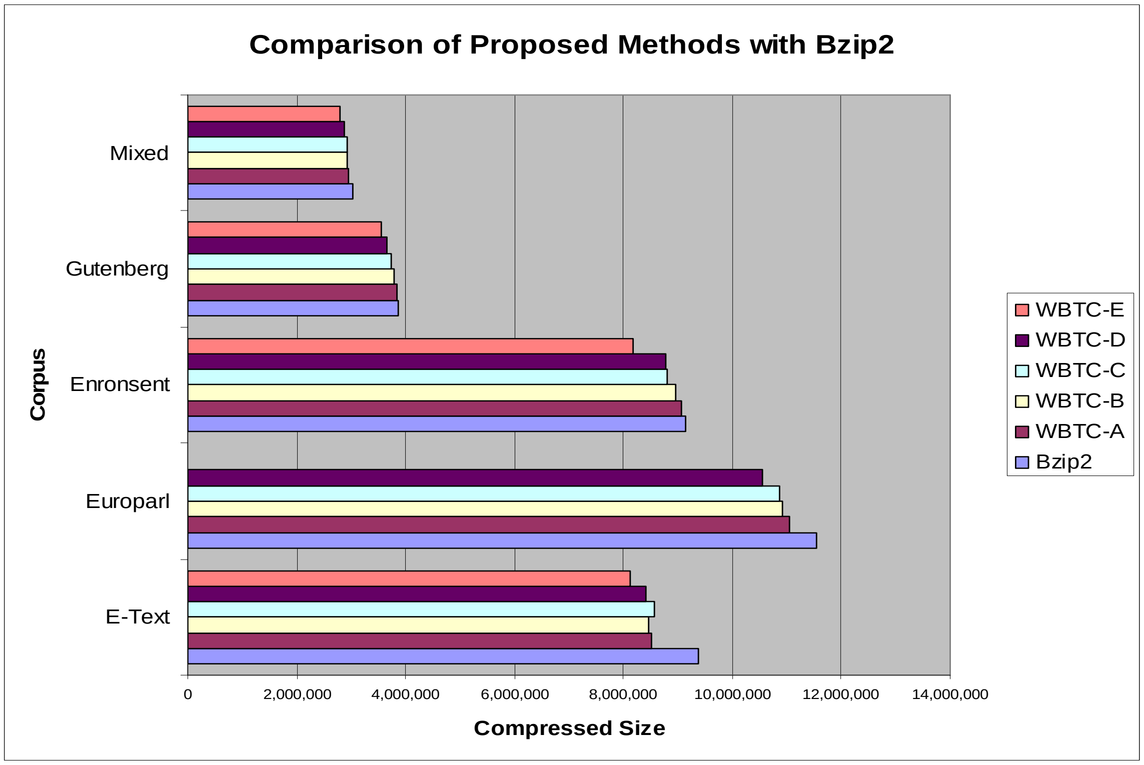

Figure 6.6 Compression ratios for all Corpus (Bzip2)

6.3 COMPARISON OF WORD BASED METHODS WITH PPMd

6.3.1. Compression Statistics of E-Text Corpus

Table 6.13 Compression ratios for E-Text Corpus (PPMd)

		Compression ration in %					
Sr.No	Name of File	PPMd	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	burroughst	24.05	23.33	22.96	23.60	22.80	22.80
2	dickens	25.67	26.53	26.03	25.59	25.40	25.04
3	doyle	24.67	24.71	24.14	24.28	23.77	23.42
4	emerson	26.52	28.34	27.38	26.92	26.28	25.65
5	hawthorne	25.51	25.60	25.10	23.79	24.54	24.00
6	irving	24.47	18.95	18.57	19.86	18.79	17.81
7	kant	18.67	16.50	16.42	17.05	16.49	16.05
8	milton	23.28	18.64	18.05	18.45	17.80	17.05
9	plato	22.65	19.36	19.08	19.70	19.18	18.54
	Total	23.73	22.45	22.02	22.31	21.77	21.36

The test was executed on E-Text Corpus and the results are shown in Table 6.13. For the entire corpus our all methods outperforms PPMd. WBTC-A achieves average gain of 1.28%. WBTC-B achieves average gain of 1.71%. WBTC-C achieves average gain of 1.42%. WBTC-D achieves average gain of 1.96%. WBTC-E achieves average gain of 2.37%.

6.3.2. Compression Statistics of European Parliament Corpus

Table 6.14 Compression ratios for European Parliament Corpus (PPMd)

		Compression ration in %				
Sr.No	Name of File	PPMd	WBTC-A	WBTC-B	WBTC-C	WBTC-D
1	europarl1	21.39	20.70	20.29	19.76	19.55
2	europarl2	21.00	20.35	19.89	19.33	19.02
3	europarl3	21.44	20.80	20.34	19.80	19.56
4	europarl4	21.32	20.68	20.16	19.68	19.41
5	europarl5	21.27	20.53	20.02	19.62	19.30
6	europarl6	21.34	20.66	20.20	19.70	19.44
7	europarl7	21.46	20.86	20.37	19.89	19.64
8	europarl8	20.98	20.20	19.64	19.19	18.89
9	europarl9	21.39	20.70	20.24	19.70	19.46
10	europarl10	21.12	20.49	20.00	19.52	19.24
	Total	21.27	20.60	20.12	19.62	19.35

The test was executed on European Parliament Corpus and the results are shown in Table 6.14. For the entire corpus, our all methods outperforms PPMd. WBTC-A achieves average gain of 0.67%. WBTC-B achieves average gain of 1.15%. WBTC-C achieves average gain of 1.65%. WBTC-D achieves average gain of 1.77%.

6.3.3. Compression Statistics of Enronsent Corpus

Table 6.15 Compression ratios for Enronsent Corpus (PPMd)

		Compression ration in %					
Sr.No	Name of File	PPMd	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	enronsent00	26.45	26.32	25.51	25.36	24.98	23.80
2	enronsent01	26.77	26.99	26.11	26.24	25.59	24.38
3	enronsent02	24.80	23.91	23.38	23.57	23.09	22.10
4	enronsent03	25.30	25.17	24.36	24.38	23.87	22.81
5	enronsent04	23.91	23.46	22.94	23.02	22.48	21.50
6	enronsent05	24.51	23.68	22.81	23.26	22.55	21.45
7	enronsent06	25.47	25.24	24.39	24.57	24.04	22.90
8	enronsent07	24.53	23.23	22.73	22.99	22.27	21.20
9	enronsent08	25.01	24.35	23.64	23.91	23.14	21.98
10	enronsent09	14.13	13.67	13.37	13.54	13.26	12.41
11	enronsent10	22.24	21.81	21.20	21.51	20.91	19.91
12	enronsent11	24.46	23.13	22.67	23.23	22.39	21.21
13	enronsent12	26.01	25.93	25.12	25.09	24.70	23.46
14	enronsent13	25.01	24.70	23.82	24.08	23.33	22.41
15	enronsent14	25.80	26.15	25.36	25.29	24.81	23.64
16	enronsent15	22.65	20.91	20.52	21.26	20.30	19.16
17	enronsent16	25.48	25.09	24.21	24.56	23.72	22.25
18	enronsent17	26.15	26.21	25.13	25.34	24.50	22.99
19	enronsent18	25.90	25.61	24.53	24.86	24.09	22.39
	Total	24.37	23.88	23.17	23.38	22.76	21.62

The test was executed on Enronsent Corpus and the results are shown in Table 6.15. For the entire corpus our all methods outperforms PPMd. WBTC-A achieves average gain of 0.49%. WBTC-B achieves average gain of 0.49%. WBTC-C achieves average gain of 0.99%. WBTC-D achieves average gain of 1.61%. WBTC-E achieves average gain of 2.75%.

6.3.4. Compression Statistics of Project Gutenberg Corpus

Table 6.16 Compression ratios for Project Gutenberg Corpus (PPMd)

		Compression ration in %					
Sr.No	Name of File	PPMd	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	leonard	26.64	28.76	27.64	27.41	26.02	25.27
2	pg1342	25.05	26.20	25.44	25.80	24.42	23.51
3	pg1399	24.41	24.71	23.92	24.22	23.20	22.66
4	pg2981	23.97	23.59	23.05	23.09	22.70	22.56
5	pg3207	24.15	25.74	24.83	24.95	23.65	23.19
6	pg33	27.73	31.40	30.05	29.74	27.66	25.89
7	pg514	26.60	28.31	27.52	27.64	26.16	25.33
8	pg76	26.69	28.61	27.79	28.03	26.49	25.80
	Total	24.80	25.41	24.67	24.75	23.85	23.40

The test was executed on Project Gutenberg corpus and the results are shown in Table 6.16. For the entire corpus our all methods outperforms PPMd except WBTC-A. WBTC-A ratio deteriorates by average gain of -0.61%. WBTC-B achieves average gain of 0.13%. WBTC-C achieves average gain of 0.05%. WBTC-D achieves average gain of 0.95%. WBTC-E achieves average gain of 1.4%

6.3.5. Compression Statistics of Mixed Corpus

Table 6.17 Compression ratios for Mixed Corpus (PPMd)

		Compression ration in %					
Sr.No	Name of File	PPMd	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	bible.txt	20.67	19.46	18.98	19.55	18.73	18.50
2	world192.txt	20.69	19.55	18.83	19.05	17.87	17.15
3	anne11.txt	28.45	32.34	31.06	30.67	28.62	26.72
4	enronsent02	24.80	23.91	23.38	23.57	23.09	22.10
5	pg10.txt	21.45	20.59	20.20	20.52	19.95	19.80
	Total	21.84	20.91	20.39	20.73	19.96	19.50

The test was executed on Mixed Corpus and the results are shown in Table 6.17. For all the files our all methods outperforms PPMd. WBTC-A achieves average gain of 0.93%. WBTC-B achieves average gain of 1.45%. WBTC-C achieves average gain of 1.11%.

WBTC-D achieves average gain of 1.88%. WBTC-E achieves average gain of 2.34%. From Table 6.17, it is seen the compression ratio deteriorates for files of smaller size as compared to files of larger size. This is because of the overhead of the dictionary associated with the compressed file.

6.3.6. Compression Statistics of all Corpus

Table 6.18 Compression ratios for all Corpus (PPMd)

		Compression ration in %					
Sr.No	Name of File	PPMd	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	E-Text	23.73	22.45	22.02	22.31	21.77	21.36
2	Europarl	21.27	20.60	20.12	19.62	19.35	-------
3	Enronsent	24.37	23.88	23.17	23.38	22.76	21.62
4	Gutenberg	24.80	25.41	24.67	24.75	23.85	23.40
5	Mixed	21.84	20.91	20.39	20.73	19.96	19.50
	Total	22.99	22.31	21.76	21.76	21.23	21.49

The overall compression ratio for the corpus is summarized in Table 6.18 For the entire corpus our all methods outperforms PPMd. WBTC-A achieves average gain of 0.68%. WBTC-B achieves average gain of 1.23%. WBTC-C achieves average gain of 1.23%. WBTC-D achieves average gain of 1.76%. The average compression ratio of PPMd method without European Parliament corpus is 23.86% therefore, WBTC-E achieves average gain of 2.37%.

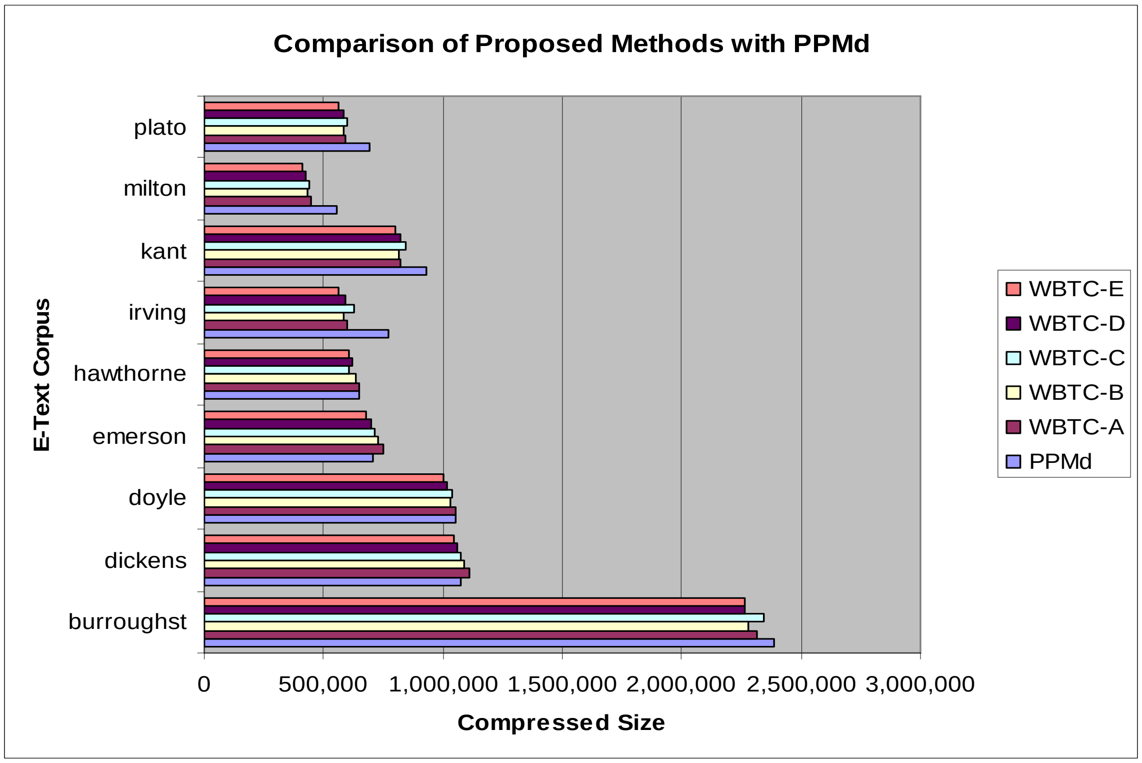

Figure 6.7 Compression ratios for E-Text Corpus (PPMd)

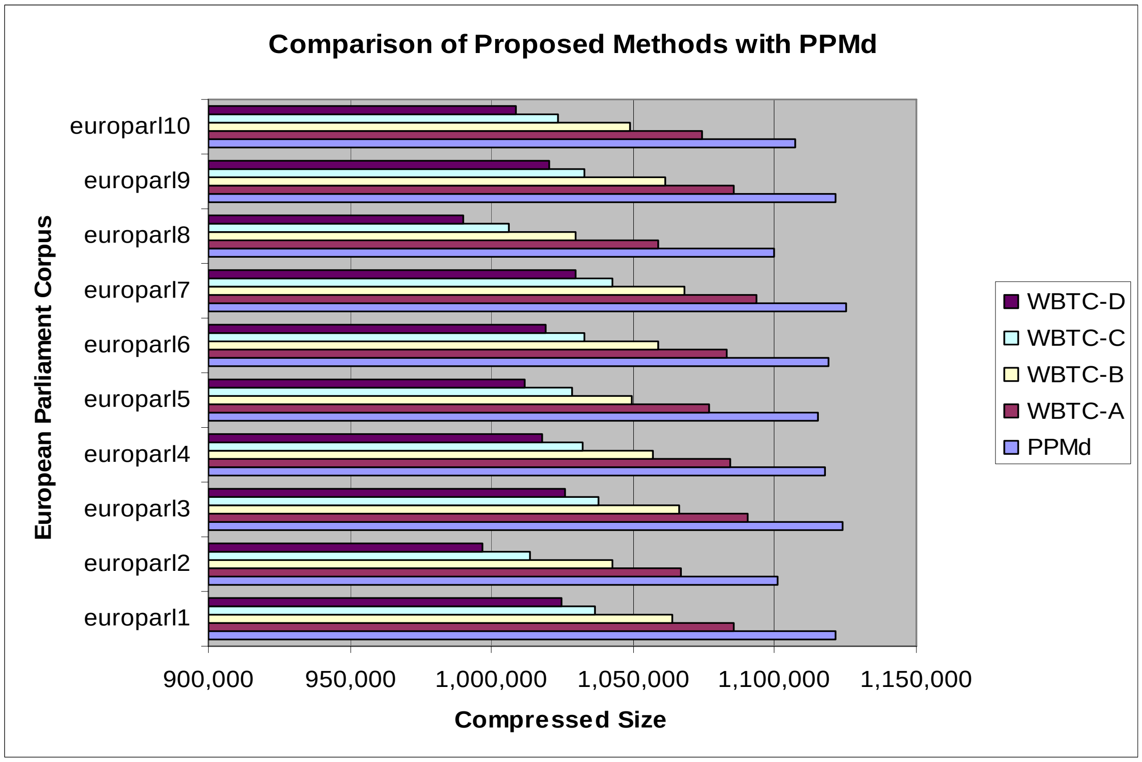

Figure 6.8 Compression ratios for European Parliament Corpus (PPMd)

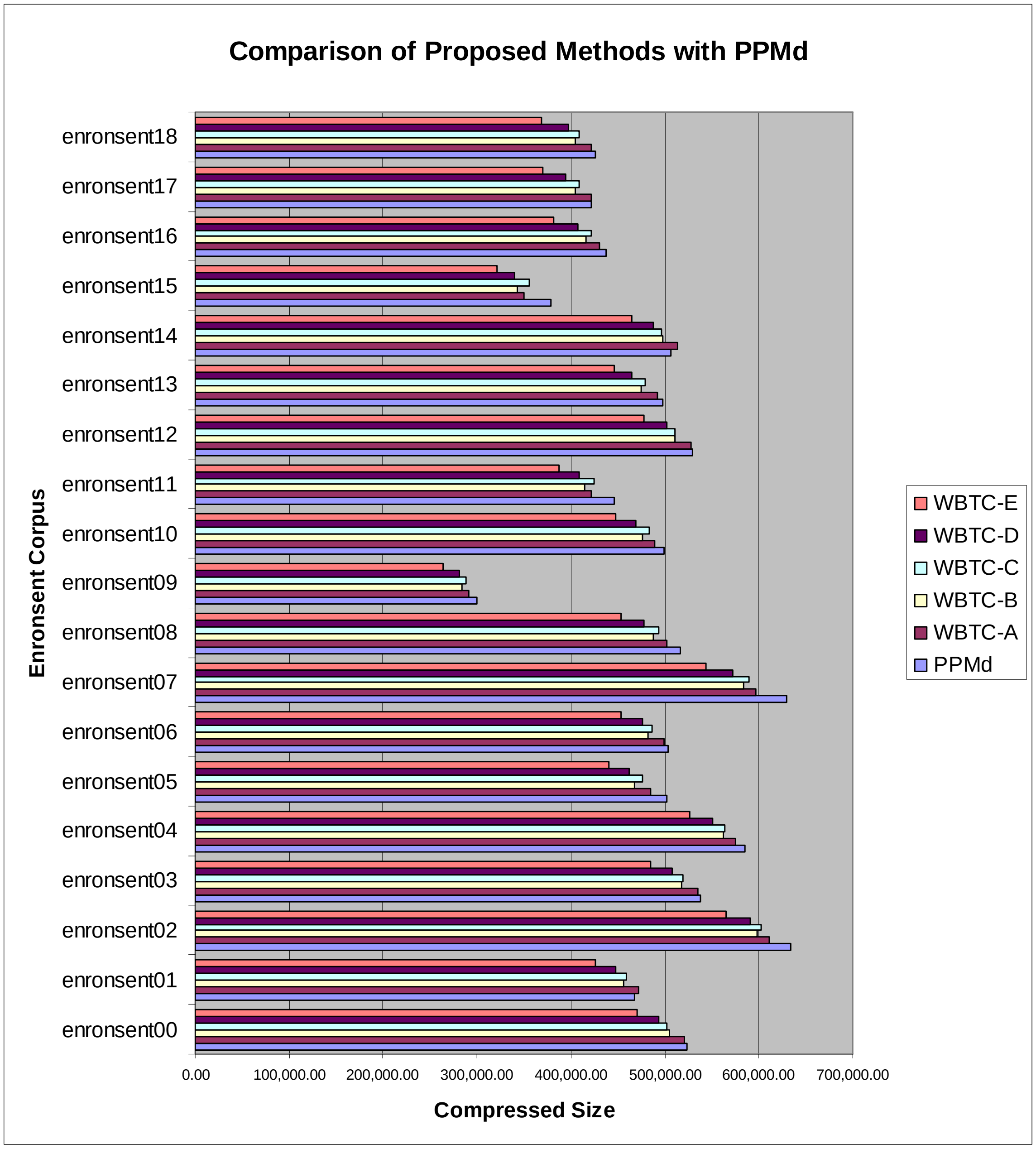

Figure 6.9 Compression ratios for Enronsent Corpus (PPMd)

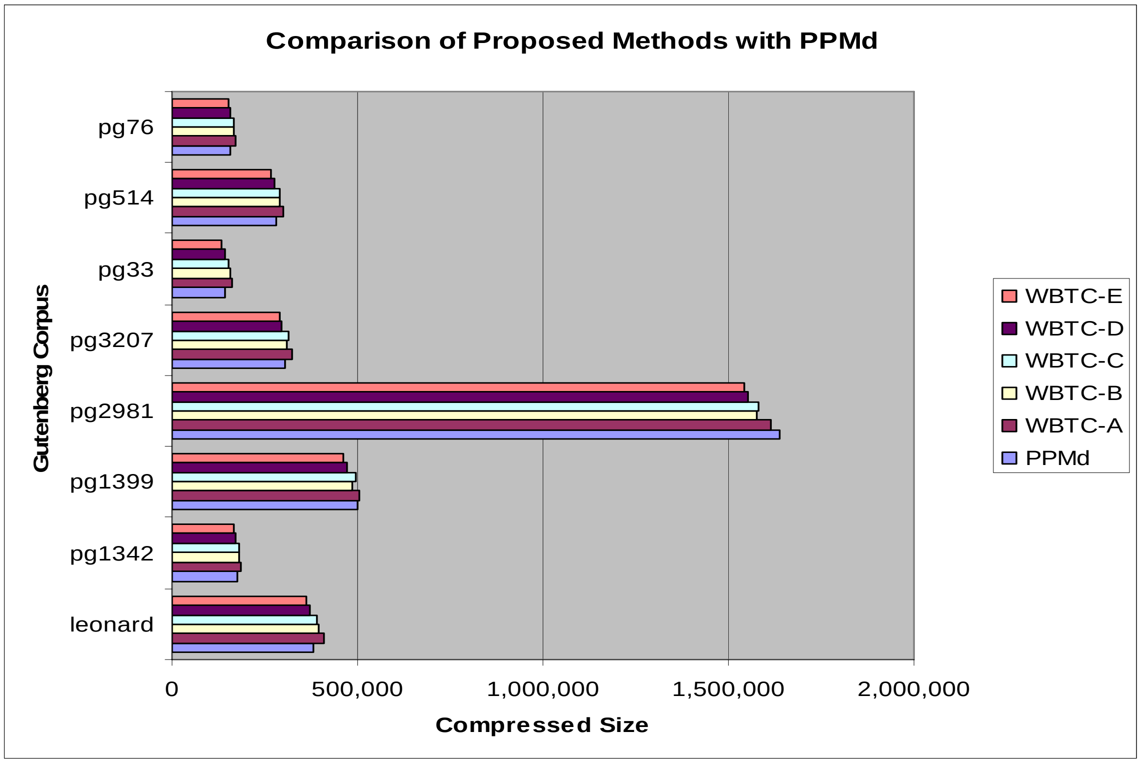

Figure 6.10 Compression ratios for Project Gutenberg Corpus (PPMd)

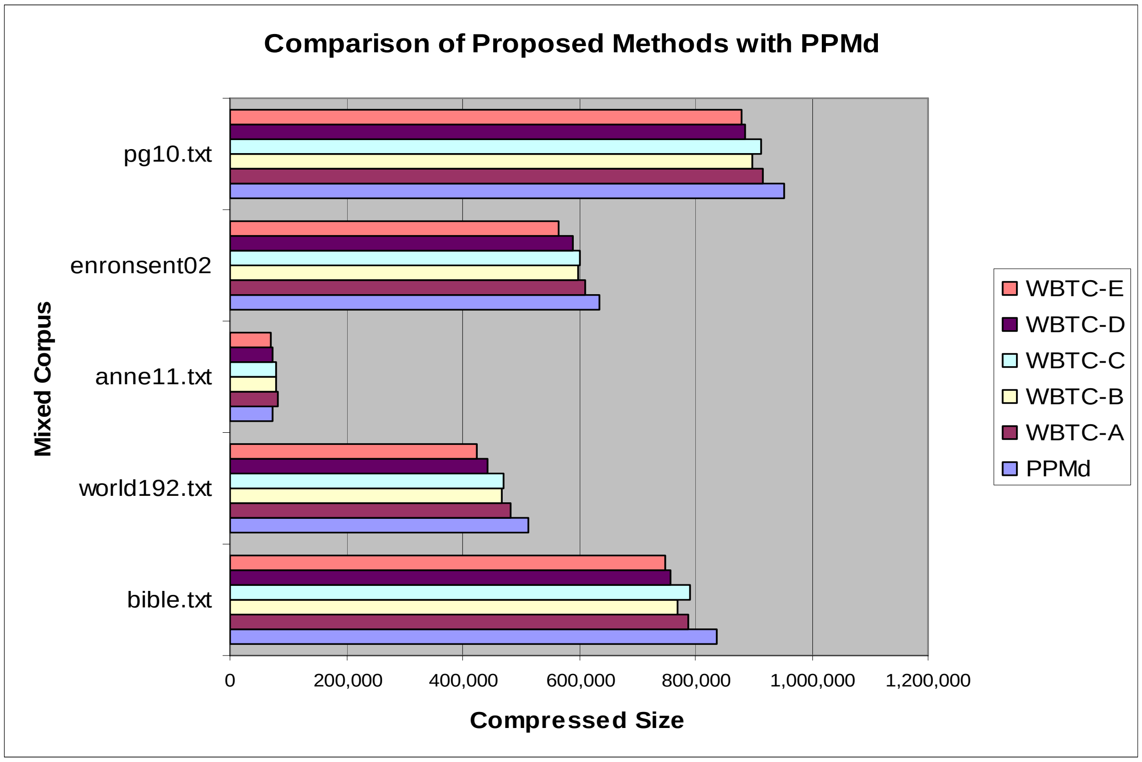

Figure 6.11 Compression ratios for Mixed Corpus (PPMd)

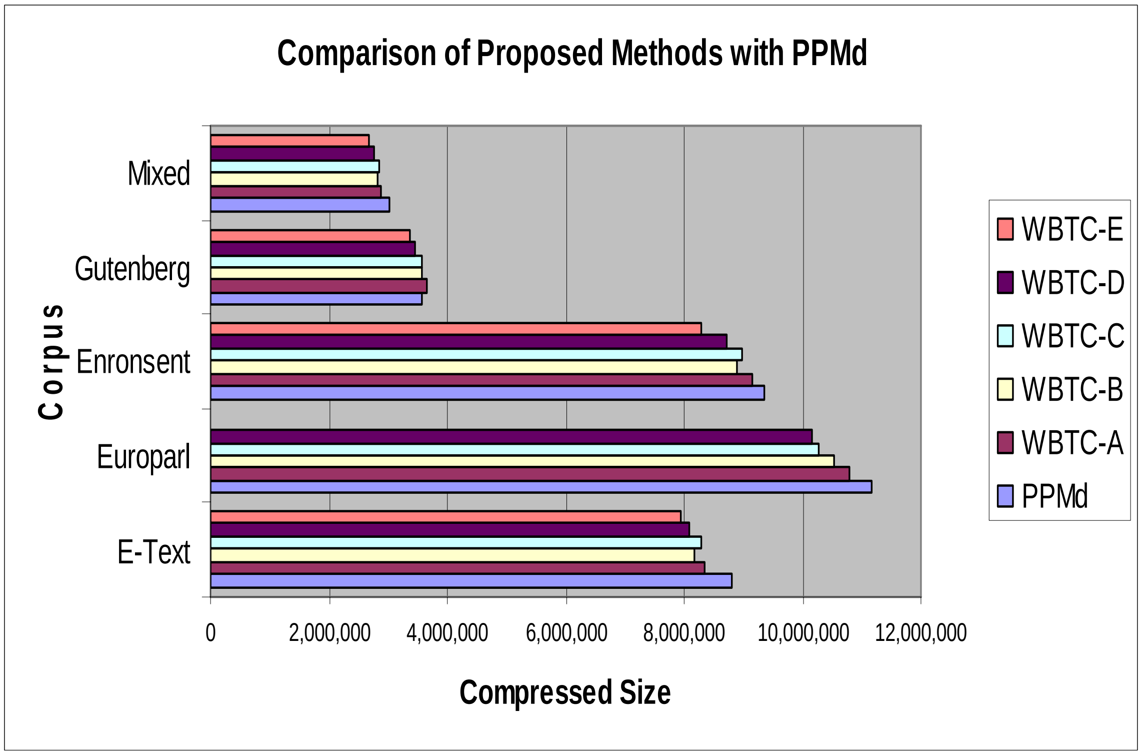

Figure 6.12 Compression ratios for all Corpus (PPMd)

6.4 COMPARISON OF WORD BASED METHODS WITH PPMII

6.4.1. Compression Statistics of E-Text Corpus

Table 6.19 Compression ratio for E-Text Corpus (PPMII)

		Compression ration in %					
Sr.No	Name of File	PPMII	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	burroughst	23.80	20.36	20.48	20.99	21.01	20.75
2	dickens	25.27	24.84	24.42	24.16	23.91	23.40
3	doyle	24.32	22.70	22.31	22.57	22.13	21.45
4	emerson	26.07	26.16	25.49	25.00	24.44	23.67
5	hawthorne	24.95	22.37	21.87	22.40	21.98	20.70
6	irving	24.01	17.79	17.31	18.61	17.71	16.69
7	kant	18.36	15.84	15.73	16.32	15.89	15.39
8	milton	22.24	17.14	16.44	16.83	16.36	15.60
9	plato	22.28	18.48	18.15	18.64	18.38	17.68
	Total	23.33	20.50	20.23	20.63	20.30	19.69

The test was executed on E-Text Corpus and the results are shown in Table 6.19. For all the files, our all methods outperforms PPMII. WBTC-A achieves average gain of 2.83%. WBTC-B achieves average gain of 3.1%. WBTC-C achieves average gain of 2.7%. WBTC-D achieves average gain of 3.03%. WBTC-E achieves average gain of 3.64%.

6.4.2. Compression Statistics of European Parliament Corpus

Table 6.20 Compression ratio for European Parliament Corpus (PPMII)

		Compression ration in %				
Sr.No	Name of File	PPMII	WBTC-A	WBTC-B	WBTC-C	WBTC-D
1	europarl1	21.14	18.97	18.57	18.59	17.98
2	europarl2	20.76	18.57	18.12	18.28	17.45
3	europarl3	21.21	19.04	18.58	18.67	17.95
4	europarl4	21.09	18.90	18.43	18.55	17.82
5	europarl5	21.03	18.81	18.34	18.50	17.75
6	europarl6	21.11	18.87	18.41	18.58	17.79
7	europarl7	21.22	19.15	18.67	18.73	18.07
8	europarl8	20.74	18.39	17.87	18.17	17.30
9	europarl9	21.17	18.92	18.48	18.59	17.85
10	europarl10	20.88	18.74	18.26	18.40	17.62
	Total	21.04	18.84	18.37	18.51	17.76

The test was executed on European Parliament Corpus and the results are shown in Table 6.20. For all the files, our all methods outperform PPMII. WBTC-A achieves average gain of 2.20%. WBTC-B achieves average gain of 2.67%. WBTC-C achieves average gain of 2.53%. WBTC-D achieves average gain of 3.28%.

6.4.3. Compression Statistics of Enronsent Corpus

Table 6.21 Compression ratio for Enronsent Corpus (PPMII)

		Compression ration in %					
Sr.No	Name of File	PPMII	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	enronsent00	25.60	24.21	23.51	23.82	23.14	21.84
2	enronsent01	25.83	24.98	24.31	24.58	23.90	22.47
3	enronsent02	24.07	22.24	21.69	22.17	21.56	20.45
4	enronsent03	24.56	23.41	22.71	22.99	22.28	21.16
5	enronsent04	23.12	21.79	21.19	21.60	20.88	19.74
6	enronsent05	23.73	21.87	21.30	21.80	21.07	19.84
7	enronsent06	24.66	23.25	22.62	23.02	22.32	21.02
8	enronsent07	23.70	21.52	20.94	21.42	20.77	19.55
9	enronsent08	24.19	22.72	21.99	22.40	21.70	20.42
10	enronsent09	13.57	12.84	12.51	12.74	12.56	11.68
11	enronsent10	21.58	20.33	19.79	20.23	19.56	18.51
12	enronsent11	23.49	21.61	21.09	21.63	21.09	19.77
13	enronsent12	25.08	24.02	23.32	23.53	22.99	21.63
14	enronsent13	24.30	23.01	22.38	22.73	21.90	20.80
15	enronsent14	24.95	24.21	23.50	23.77	23.10	21.78
16	enronsent15	21.87	19.64	19.18	19.89	19.17	17.93
17	enronsent16	24.48	23.23	22.51	22.87	22.30	20.65
18	enronsent17	25.18	24.20	23.41	23.66	22.98	21.43
19	enronsent18	24.88	23.53	22.77	23.12	22.63	20.80
	Total	23.55	22.15	21.53	21.92	21.28	20.01

The test was executed on Enronsent Corpus and the results are shown in Table 6.21. For all the files, our all methods outperform PPMII. WBTC-A achieves average gain of 1.40%. WBTC-B achieves average gain of 2.02%. WBTC-C achieves average gain of 1.63%. WBTC-D achieves average gain of 2.27%. WBTC-E achieves average gain of 3.54%.

6.4.4. Compression Statistics of Project Gutenberg Corpus

Table 6.22 Compression ratio for Project Gutenberg Corpus (PPMII)

		Compression ration in %					
Sr.No	Name of File	PPMII	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	leonard	26.00	27.19	26.16	25.98	24.84	24.03
2	pg1342	24.36	24.74	23.99	24.28	23.17	22.25
3	pg1399	23.98	23.38	22.88	23.11	22.25	21.68
4	pg2981	23.73	22.05	21.77	21.69	21.47	21.18
5	pg3207	23.62	24.44	23.53	23.61	22.55	22.04
6	pg33	26.81	29.38	28.03	27.70	26.06	24.12
7	pg514	25.96	26.80	26.04	26.10	24.88	24.02
8	pg76	25.92	27.07	26.26	26.40	25.19	24.53
	Total	24.38	23.90	23.34	23.34	22.66	22.11

The test was executed on Project Gutenberg Corpus and the results are shown in Table 6.22. For all the files, our all methods outperform PPMII. WBTC-A achieves average gain of 0.48%. WBTC-B achieves average gain of 1.04%. WBTC-C achieves average gain of 1.04%. WBTC-D achieves average gain of 1.72%. WBTC-E achieves average gain of 2.27%.

6.4.5. Compression Statistics of Mixed Corpus

Table 6.23 Compression ratio for Mixed Corpus (PPMII)

		Compression ration in %					
Sr.No	FileName	PPMII	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	bible.txt	20.39	18.66	18.33	18.84	18.12	17.87
2	world192.txt	20.21	18.54	17.96	18.22	17.12	16.39
3	anne11.txt	27.29	30.15	28.85	28.57	26.88	24.96
4	enronsent02	24.07	22.24	21.69	22.17	21.56	20.45
5	pg10.txt	21.11	19.63	19.34	19.75	19.10	18.90
	Total	21.40	19.83	19.41	19.82	19.06	18.55

The test was executed on Mixed Corpus and the results are shown in Table 6.23. For all the files, our all methods outperform PPMII. WBTC-A achieves average gain of 1.57%. WBTC-B achieves average gain of 1.99%. WBTC-C achieves average gain of 1.58%. WBTC-D achieves average gain of 2.34%. WBTC-E achieves average gain of 2.85%.

6.4.6. Compression Statistics of all Corpus

Table 6.24 Compression ratio for all Corpus (PPMII)

Sr.No	Name of File	PPMII	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	E-Text	23.33	20.50	20.23	20.63	20.30	19.69
2	Europarl	21.04	18.84	18.37	18.51	17.76	0.00
3	Enronsent	23.55	22.15	21.53	21.92	21.28	20.01
4	Gutenberg	24.38	23.90	23.34	23.34	22.66	22.11
5	Mixed	21.40	19.83	19.41	19.82	19.06	18.55
	Total	22.54	20.60	20.14	20.41	19.79	19.99

The overall compression ratio for the corpus is summarized in Table 6.24 For the entire corpus our all methods outperforms PPMII. WBTC-A achieves average gain of 1.94%. WBTC-B achieves average gain of 2.4%. WBTC-C achieves average gain of 2.13%. WBTC-D achieves average gain of 2.75%. The average compression ratio of PPMd method without European Parliament corpus is 23.30% therefore, WBTC-E achieves average gain of 3.31%.

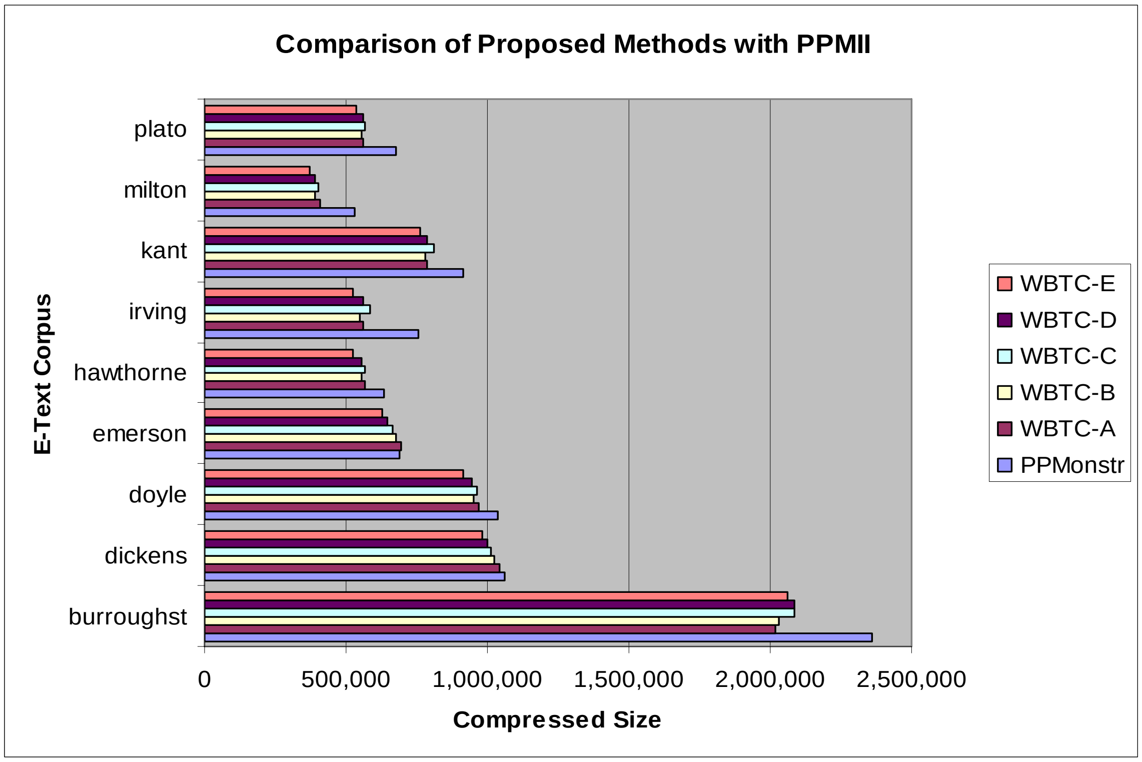

Figure 6.13 Compression ratios for E-Text Corpus (PPMII)

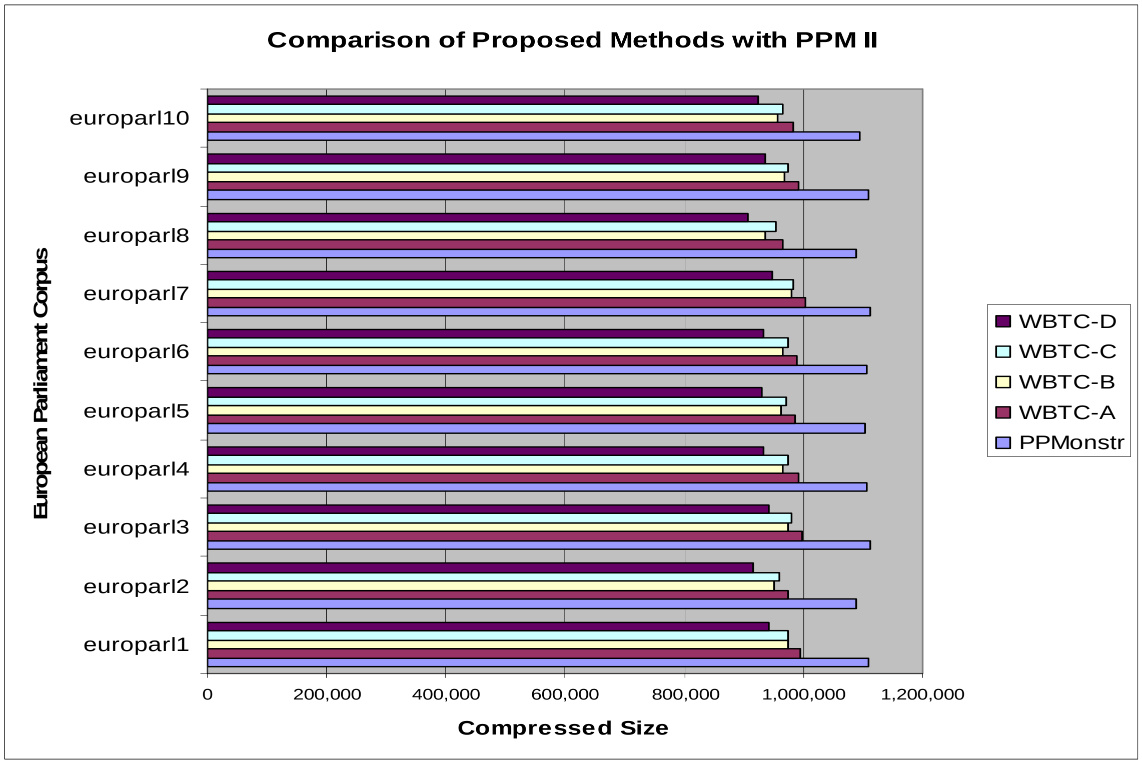

Figure 6.14 Compression ratios for European Parliament Corpus (PPMII)

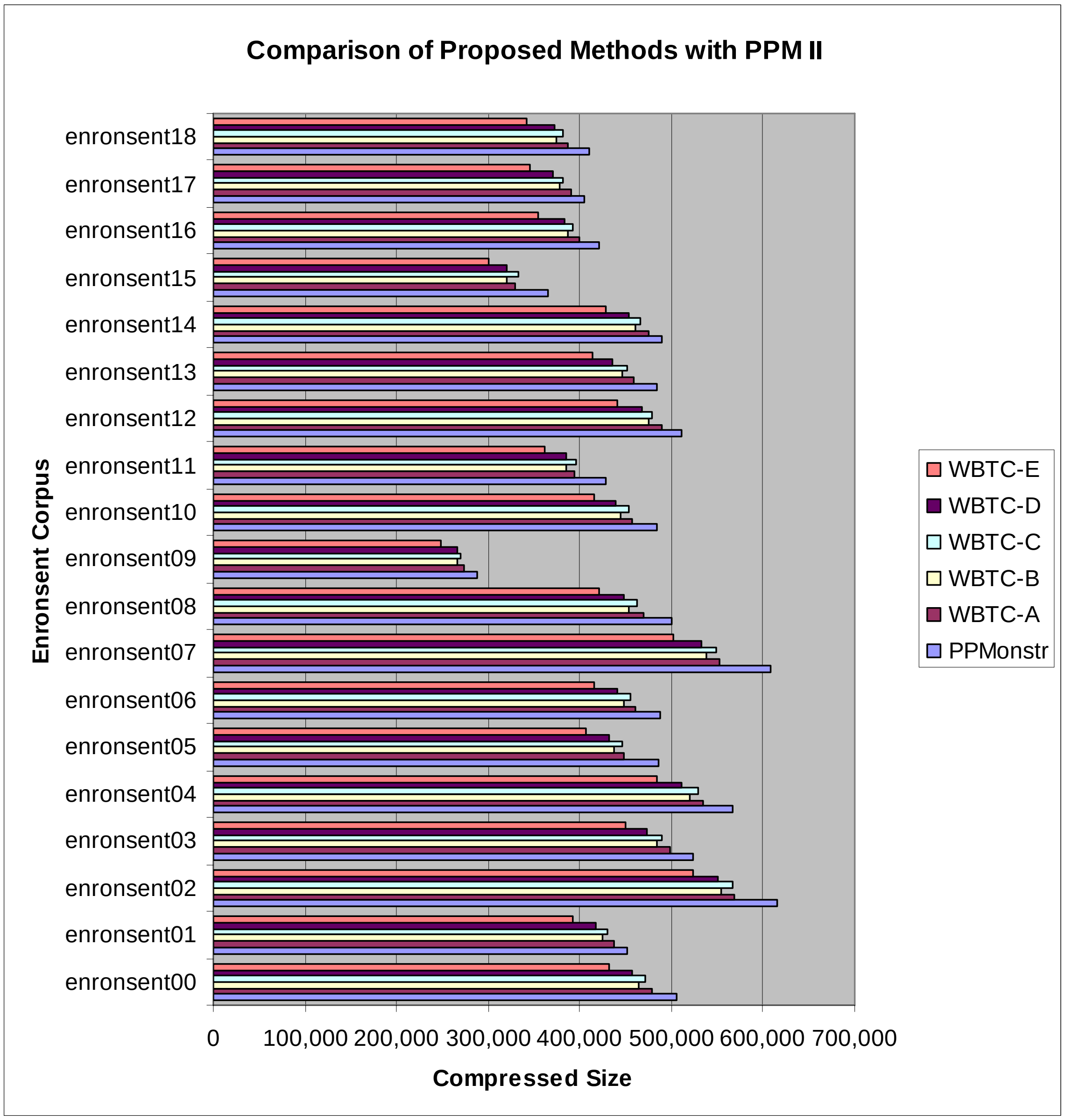

Figure 6.15 Compression ratios for Enronsent Corpus (PPMII)

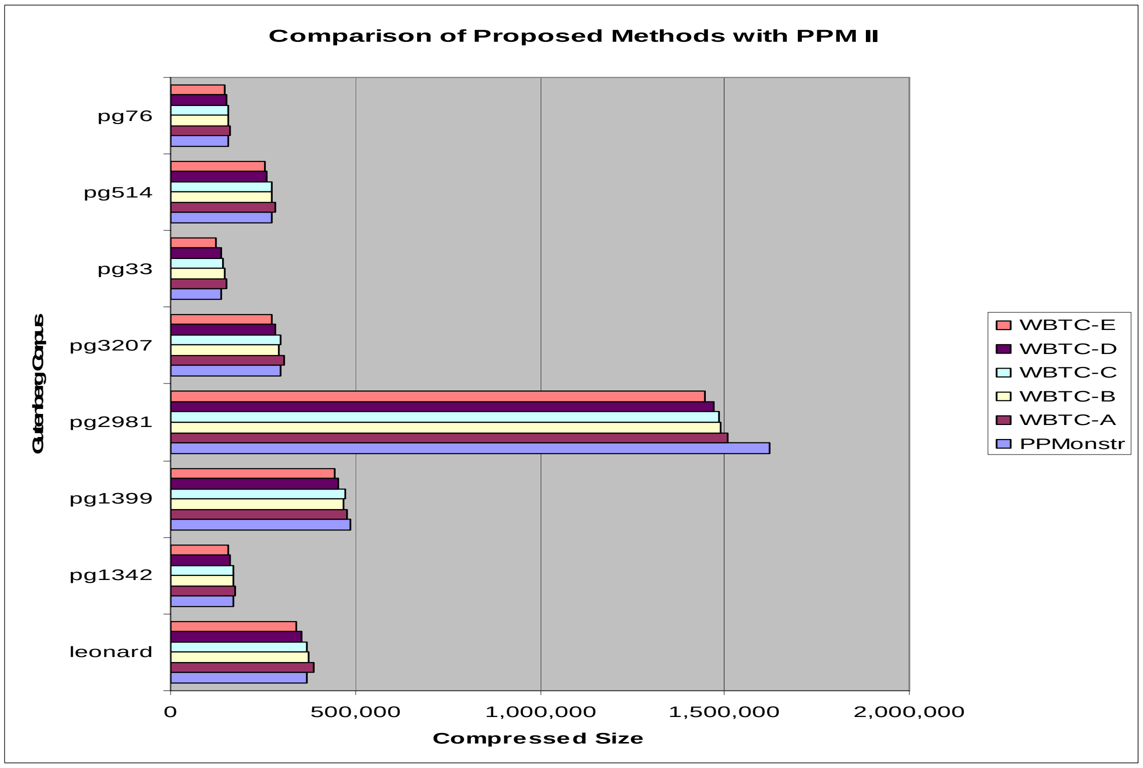

Figure 6.16 Compression ratios for Project Gutenberg Corpus (PPMII)

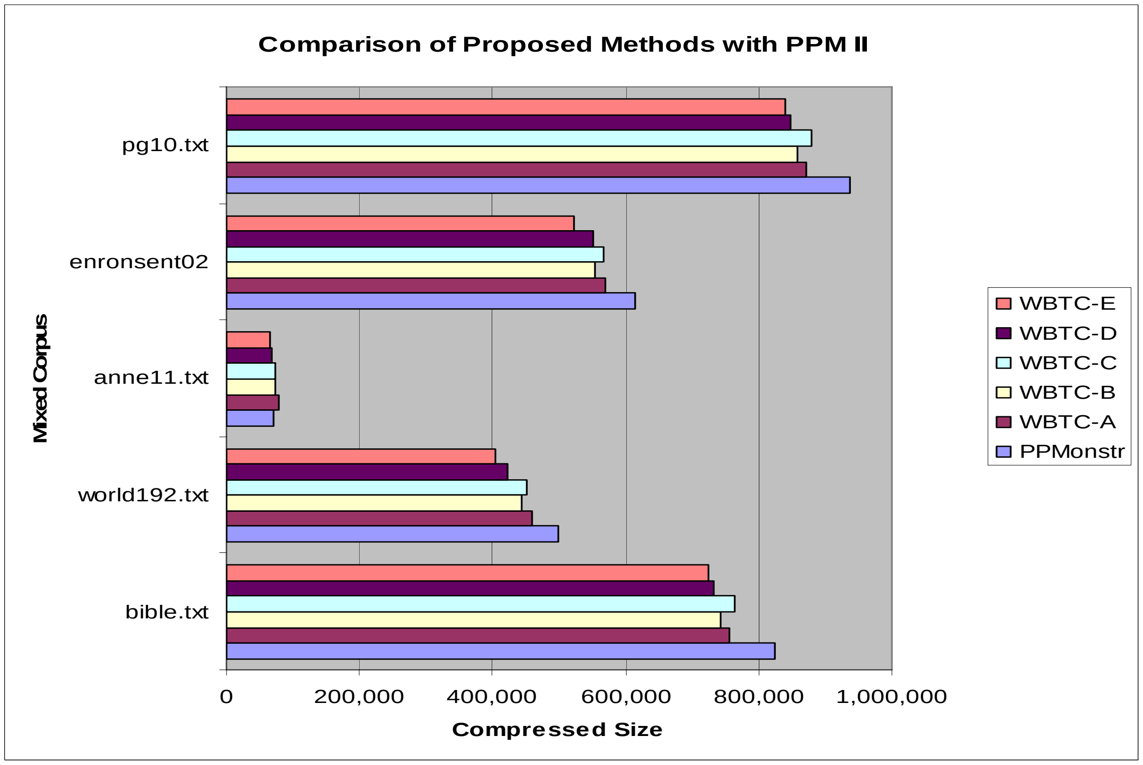

Figure 6.17 Compression ratios for Mixed Corpus (PPMII)

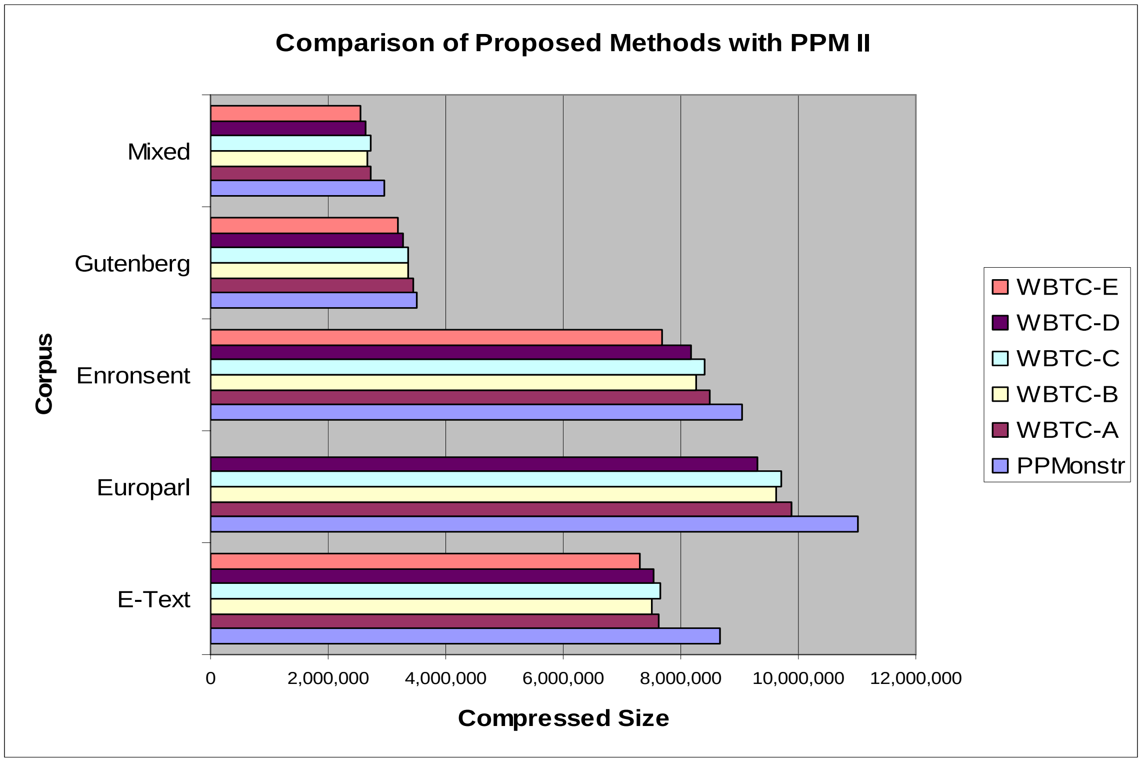

Figure 6.18 Compression ratios for all Corpus (PPMII)

6.5 COMPARISON OF WORD BASED METHODS WITH LZMA

6.5.1. Compression Statistics of E-Text Corpus

Table 6.25 Compression ratio for E-Text Corpus (LZMA)

		Compression ration in %					
Sr.No	Name of File	LZMA	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	burroughst	16.50	15.78	15.74	15.91	16.37	15.27
2	dickens	29.31	28.54	28.33	28.26	27.95	27.06
3	doyle	24.89	24.36	24.11	24.17	24.51	23.09
4	emerson	29.56	29.51	28.92	28.70	28.12	27.01
5	hawthorne	21.92	21.59	21.29	21.33	22.32	19.76
6	irving	10.53	10.71	10.52	10.55	11.70	9.38
7	kant	14.10	13.54	13.58	13.82	14.27	13.15
8	milton	11.36	12.18	11.79	11.67	12.09	10.40
9	plato	13.24	12.86	12.74	12.96	13.54	12.00
	Total	18.79	18.38	18.21	18.29	18.67	17.28

The test was executed on E-Text Corpus and the results are shown in Table 6.25. For the entire corpus our all methods outperforms LZMA. WBTC-A achieves average gain of 0.41%. WBTC-B achieves average gain of 0.58%. WBTC-C achieves average gain of 0.50%. WBTC-D achieves average gain of 0.12%. WBTC-E achieves average gain of 1.51%.

6.5.2. Compression Statistics of European Parliament Corpus

Table 6.26 Compression ratio for European Parliament Corpus (LZMA)

		Compression ration in %				
Sr.No	Name of File	LZMA	WBTC-A	WBTC-B	WBTC-C	WBTC-D
1	europarl1	22.20	21.97	21.58	21.56	21.22
2	europarl2	21.95	21.76	21.38	21.35	20.97
3	europarl3	22.45	22.19	21.78	21.79	21.39
4	europarl4	22.25	21.98	21.56	21.60	21.21
5	europarl5	22.23	21.95	21.55	21.57	21.19
6	europarl6	22.33	22.05	21.63	21.66	21.26
7	europarl7	22.46	22.22	21.80	21.81	21.41
8	europarl8	21.84	21.57	21.14	21.19	20.79
9	europarl9	22.37	22.07	21.69	21.69	21.32
10	europarl10	22.12	21.86	21.45	21.47	21.08
	Total	22.22	21.96	21.56	21.57	21.18

The test was executed on European Parliament Corpus and the results are shown in Table 6.26. For the entire corpus our all methods outperforms LZMA. WBTC-A achieves average gain of 0.26%. WBTC-B achieves average gain of 0.66%. WBTC-C achieves average gain of 0.65%. WBTC-D achieves average gain of 1.04%.

6.5.3. Compression Statistics of Enronsent Corpus

Table 6.27 Compression ratios for Enronsent Corpus (LZMA)

		Compression ration in %					
Sr.No	Name of File	LZMA	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	enronsent00	23.72	24.29	23.64	23.73	23.73	21.79
2	enronsent01	25.39	25.94	25.32	25.43	25.33	23.29
3	enronsent02	22.29	22.32	21.81	21.96	22.10	20.46
4	enronsent03	23.82	24.22	23.59	23.67	23.57	21.86
5	enronsent04	21.05	21.44	20.87	20.93	20.99	19.23
6	enronsent05	22.17	22.21	21.69	21.90	21.80	20.08
7	enronsent06	23.96	24.16	23.56	23.71	23.65	21.78
8	enronsent07	20.71	20.99	20.48	20.50	20.72	18.89
9	enronsent08	22.56	22.98	22.33	22.45	22.41	20.51
10	enronsent09	11.52	11.77	11.48	11.69	11.70	10.49
11	enronsent10	20.43	20.53	20.09	20.25	20.26	18.62
12	enronsent11	20.07	20.40	19.96	20.25	20.42	18.47
13	enronsent12	23.58	24.23	23.53	23.63	23.64	21.66
14	enronsent13	23.82	24.01	23.42	23.57	23.32	21.67
15	enronsent14	24.26	24.80	24.16	24.22	24.22	22.18
16	enronsent15	19.61	19.59	19.19	19.61	19.50	17.83
17	enronsent16	22.82	23.48	22.75	22.90	23.03	20.67
18	enronsent17	23.97	24.79	23.97	24.04	23.99	21.82
19	enronsent18	23.15	23.88	23.14	23.24	23.45	20.90
	Total	21.95	22.31	21.74	21.87	21.88	20.03

The test was executed on Enronsent Corpus and the results are shown in Table 6.27. For the entire corpus our all methods outperforms LZMA except WBTC-A. WBTC-A ratio deteriorates by average gain of -0.36%. WBTC-B achieves average gain of 0.21%. WBTC-C achieves average gain of 0.08%. WBTC-D achieves average gain of 0.07%. WBTC-E achieves average gain of 1.92%.

6.5.4. Compression Statistics of Project Gutenberg Corpus

Table 6.28 Compression ratios for Project Gutenberg Corpus (LZMA)

		Compression ration in %					
Sr.No	Name of File	LZMA	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	leonard	30.19	31.26	30.26	30.28	29.28	28.02
2	pg1342	29.76	29.39	28.75	29.10	28.03	26.52
3	pg1399	28.34	27.87	27.44	27.75	26.98	26.07
4	pg2981	26.75	25.70	25.66	25.70	25.46	24.90
5	pg3207	28.12	28.58	27.80	27.98	27.00	26.21
6	pg33	32.67	34.69	33.33	33.25	31.54	28.56
7	pg514	31.39	31.90	31.22	31.45	30.26	28.88
8	pg76	31.13	32.13	31.38	31.60	30.51	29.30
	Total	28.31	28.03	27.62	27.74	27.09	26.17

The test was executed on Project Gutenberg Corpus and the results are shown in Table 6.28. For the entire corpus our all methods outperforms LZMA. WBTC-A achieves average gain of 0.28%. WBTC-B achieves average gain of 0.69%. WBTC-C achieves average gain of 0.57%. WBTC-D achieves average gain of 1.22%. WBTC-E achieves average gain of 2.14%.

6.5.5. Compression Statistics of Mixed Corpus

Table 6.29 Compression ratios for Mixed Corpus (LZMA)

		Compression ration in %					
Sr.No	Name of File	LZMA	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	bible.txt	21.87	20.86	20.68	21.07	20.66	20.30
2	world192.txt	19.60	20.00	19.31	19.43	18.95	18.06
3	anne11.txt	33.17	35.76	34.45	34.11	32.65	29.50
4	enronsent02	22.29	22.32	21.81	21.97	22.10	20.46
5	pg10.txt	23.79	22.72	22.64	22.92	22.61	22.22
	Total	22.37	21.85	21.54	21.78	21.48	20.72

The test was executed on Mixed Corpus and the results are shown in Table 6.29. For all the files our all methods outperforms LZMA. WBTC-A achieves average gain of 0.52%. WBTC-B achieves average gain of 0.83%. WBTC-C achieves average gain of 0.59%. WBTC-D achieves average gain of 0.89%. WBTC-E achieves average gain of 1.65%.

From Table 6.17, it is seen the compression ratio deteriorates for files of smaller size as compared to files of larger size. This is because of the overhead of the dictionary associated with the compressed file.

6.5.6. Compression Statistics of all Corpus

Table 6.30 Compression ratios all Corpus (LZMA)

		Compression ration in %					
Sr.No	Name of File	LZMA	WBTC-A	WBTC-B	WBTC-C	WBTC-D	WBTC-E
1	E-Text	18.79	18.38	18.21	18.29	18.67	17.28
2	Europarl	22.22	21.96	21.56	21.57	21.18	-------
3	Enronsent	21.95	22.31	21.74	21.87	21.88	20.03
4	Gutenberg	28.31	28.03	27.62	27.74	27.09	26.17
5	Mixed	22.37	21.85	21.54	21.78	21.48	20.72
	Total	21.91	21.75	21.36	21.45	21.33	19.99

The overall compression ratio for the corpus is summarized in Table 6.30. For the entire corpus our all methods outperforms LZMA. WBTC-A achieves average gain of 0.16%. WBTC-B achieves average gain of 0.55%. WBTC-C achieves average gain of 0.46%. WBTC-D achieves average gain of 0.58%. The average compression ratio of LZMA method without European Parliament corpus is 21.76% therefore, WBTC-E achieves average gain of 1.77%.

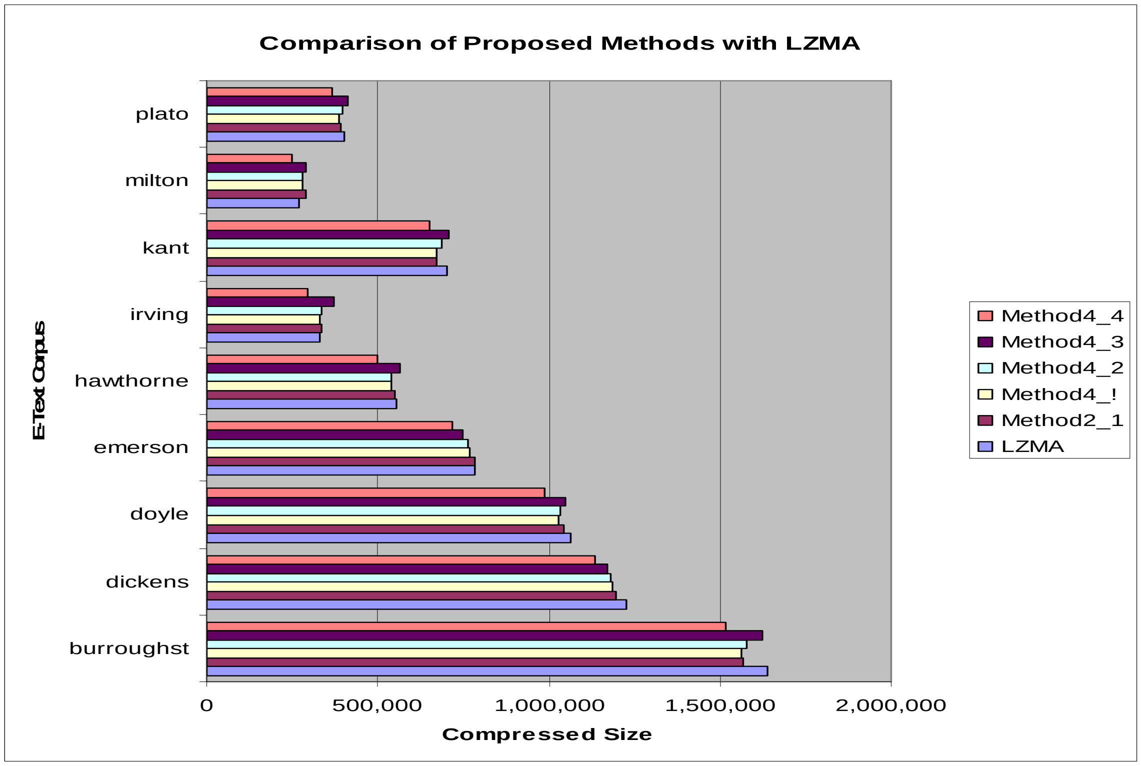

Figure 6.19 Compression ratios for E-Text Corpus (LZMA)

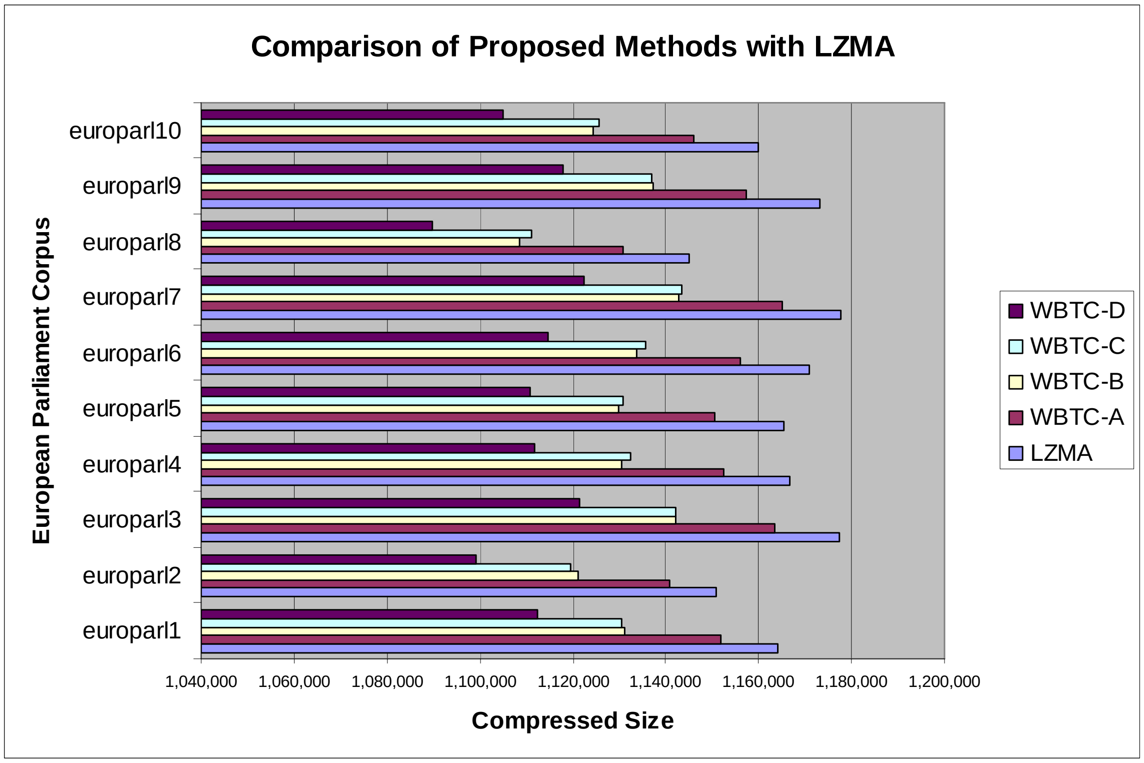

Figure 6.20 Compression ratios for European Parliament Corpus (LZMA)

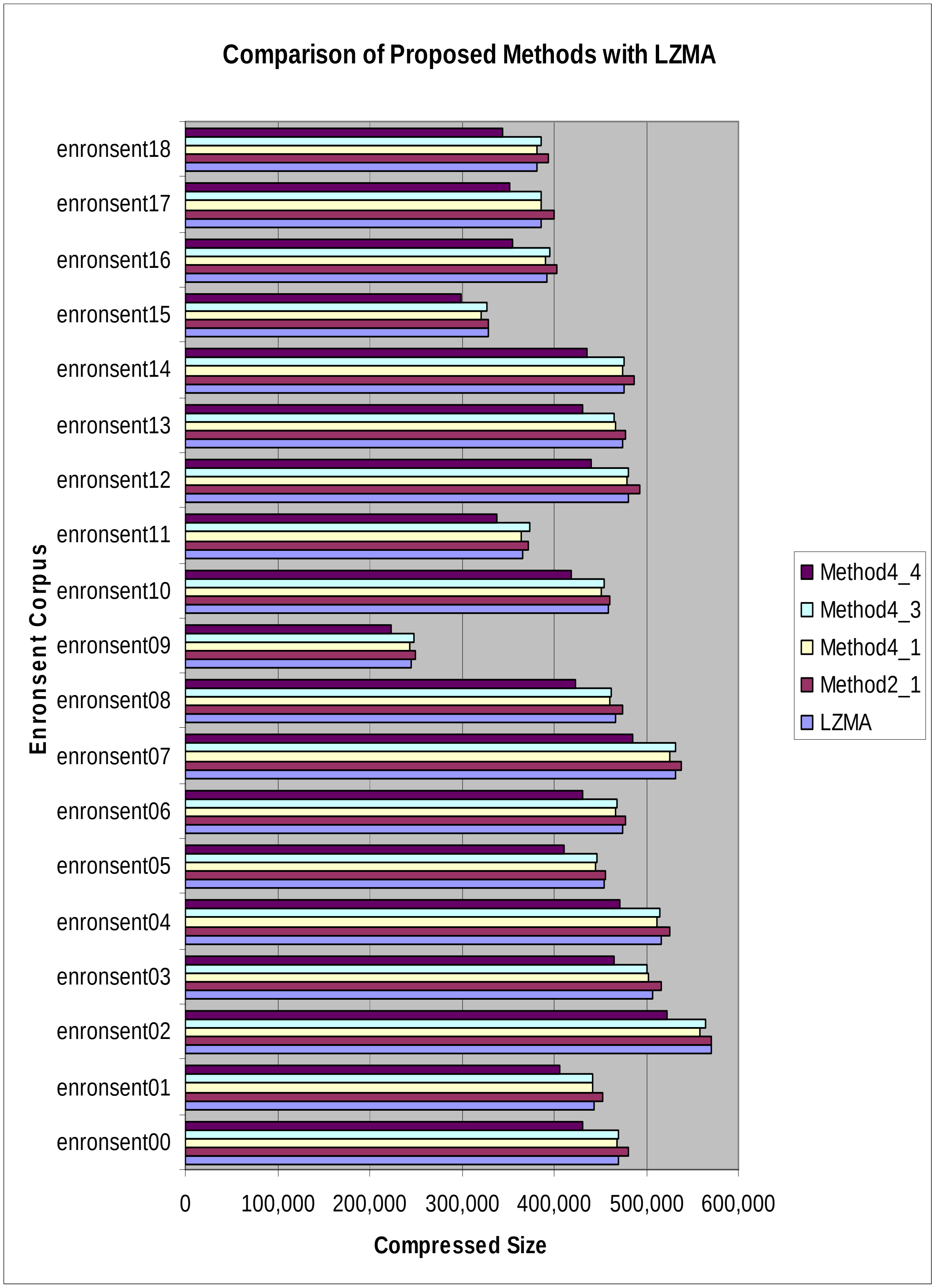

Figure 6.21 Compression ratios for Enronsent Corpus (LZMA)

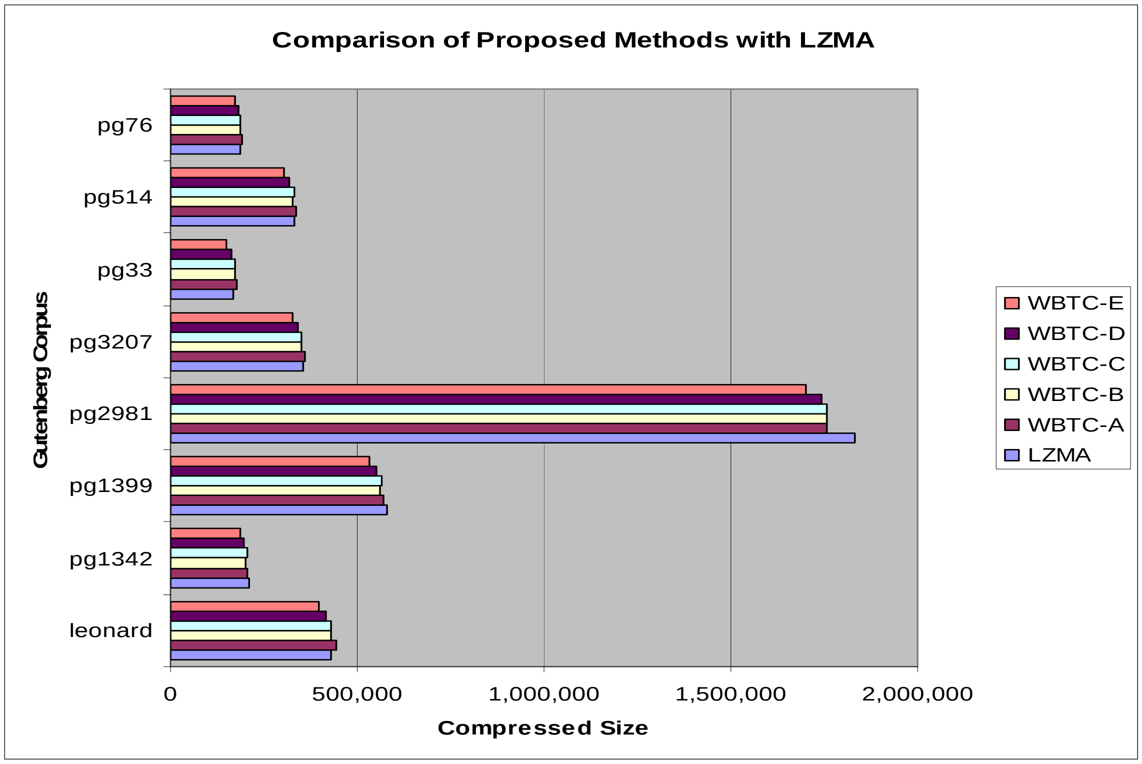

Figure 6.22 Compression ratios for Project Gutenberg Corpus (LZMA)

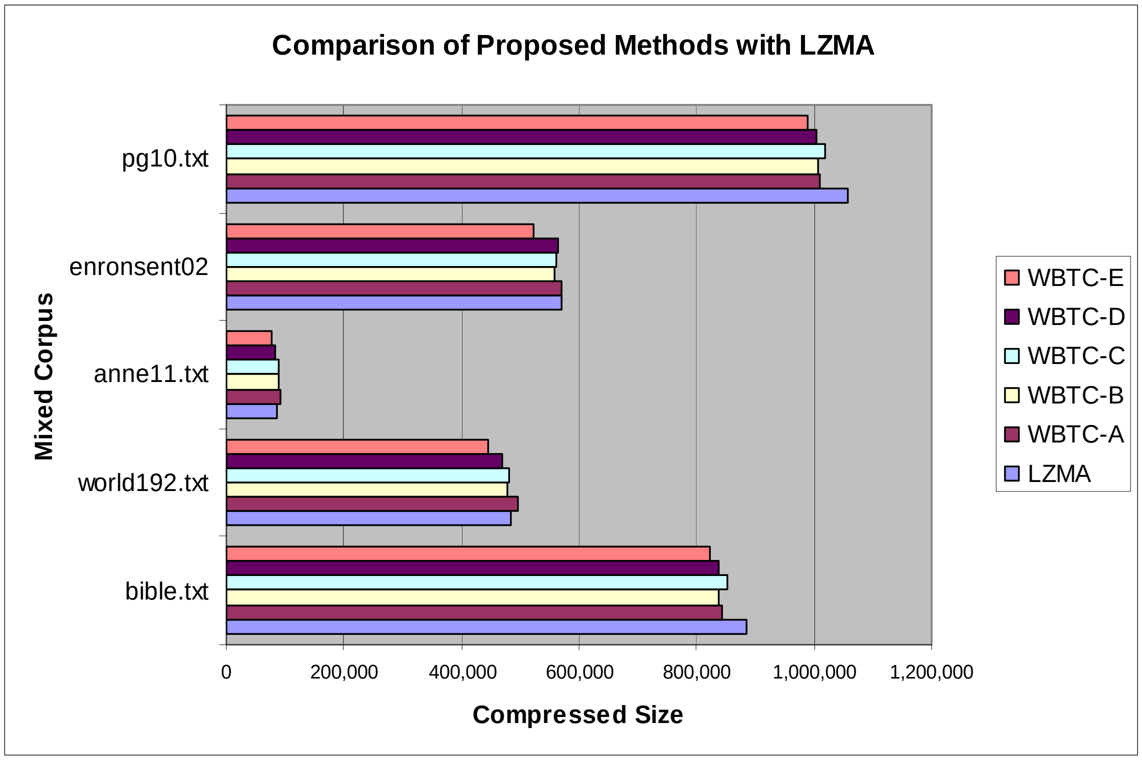

Figure 6.23 Compression ratios for Mixed Corpus (LZMA)

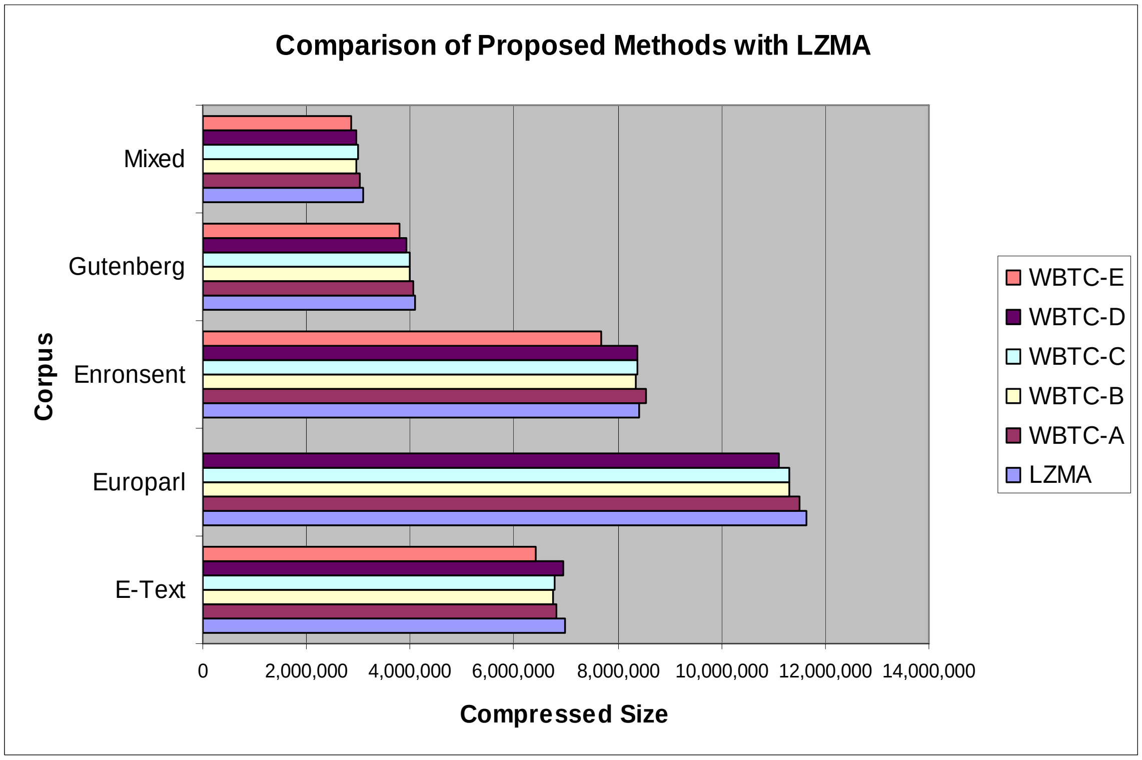

Figure 6.24 Compression ratios for all Corpus (LZMA)

6.6 COMPARISON OF CBTC-B WITH ARITHMETIC CODING

6.6.1. Compression Statistics of E-Text Corpus

Table 6.31 Compression ratios of E-Text Corpus (Arithmetic Coding)

		Compression ratio in %	
Sr.No	Name of File	Arithmetic Coding	CBTC-B
1	burroughst	55.77	49.78
2	dickens	56.02	50.53
3	doyle	54.93	49.95
4	emerson	54.85	50.67
5	hawthorne	55.43	50.74
6	irving	55.13	50.28
7	kant	51.76	44.71
8	milton	54.24	49.89
9	plato	54.56	48.40
	Total	54.83	49.38

The test was executed on E-Text Corpus and the results are shown in Table 6.31. For all the files our all methods outperforms Arithmetic Coding. CBTC-B achieves average gain of 5.45%.

6.6.2. Compression Statistics of European Parliament Corpus

Table 6.32 Compression ratios of European Parliament Corpus (Arithmetic Coding)

		Compression ratio in %	
Sr.No	FileName	Arithmetic Coding	CBTC-B
1	europarl1	54.67	49.26
2	europarl2	54.65	49.52
3	europarl3	54.27	49.17
4	europarl4	54.37	49.30
5	europarl5	54.51	49.29
6	europarl6	54.35	49.30
7	europarl7	54.52	49.55
8	europarl8	54.33	48.99
9	europarl9	54.21	49.11
10	europarl10	54.25	48.89
	Total	54.41	49.24

The test was executed on European Parliament Corpus and the results are shown in Table 6.32. For all the files our all methods outperforms Arithmetic Coding. CBTC-B achieves average gain of 5.17%.

6.6.3. Compression Statistics of Enronsent Corpus

Table 6.33 Compression ratios of Enronsent Corpus (Arithmetic Coding)

		Compression ratio in %	
Sr.No	FileName	Arithmetic Coding	CBTC-B
1	enronsent00	59.81	55.15
2	enronsent01	60.03	55.30
3	enronsent02	59.07	53.54
4	enronsent04	59.90	55.30
5	enronsent05	59.63	54.42
6	enronsent06	60.30	55.38
7	enronsent07	59.50	54.16
8	enronsent08	59.27	54.43
9	enronsent09	62.75	55.60
10	enronsent10	59.93	54.35
11	enronsent11	61.50	55.99
12	enronsent12	60.98	56.04
13	enronsent13	59.91	54.87
14	enronsent14	59.65	54.72
15	enronsent15	61.32	55.64
16	enronsent16	61.15	56.35
17	enronsent17	61.16	56.51
18	enronsent18	60.90	56.44
19	enronsent19	61.02	56.55
	Total	60.34	55.21

The test was executed on Enronsent Corpus and the results are shown in Table 6.33. For all the files our all methods outperforms Arithmetic Coding. CBTC-B achieves average gain of 5.13%.

6.6.4. Compression Statistics of Project Gutenberg Corpus

Table 6.34 Compression ratios of Project Gutenberg Corpus (Arithmetic Coding)

		Compression ratio in %	
Sr.No	FileName	Arithmetic Coding	CBTC-B
1	leonard	58.40	53.57
2	pg514	56.78	52.79
3	pg3207	57.31	51.89
4	pg33	56.32	53.87
5	pg1342	56.17	51.22
6	pg1399	56.82	51.15
7	pg76	57.21	52.87
8	pg2981	56.25	50.84
	Total	56.71	52.90

The test was executed on Project Gutenberg Corpus and the results are shown in Table 6.34. For all the files our all methods outperforms Arithmetic Coding. CBTC-B achieves average gain of 3.81%.

6.6.5. Compression Statistics of Mixed Corpus

Table 6.35 Compression ratios of Mixed Corpus (Arithmetic Coding)

		Compression ratio in %	
Sr.No	FileName	Arithmetic Coding	CBTC-B
1	bible.txt	54.40	47.73
2	world192.txt	62.48	57.22
3	anne11.txt	57.68	56.16
4	enronsent02	59.07	53.54
5	pg10.txt	57.50	50.66
	Total	57.78	51.62

The test was executed on Mixed Corpus and the results are shown in Table 6.35. For all the files our all methods outperforms Arithmetic Coding. CBTC-B achieves average gain of 6.16%.

6.6.6. Compression Statistics of all Corpus

Table 6.36 Compression ratios of all Corpus (Arithmetic Coding)

		Compression ratio in %	
Sr.No	Corpus	Arithmetic Coding	CBTC-B
1	E-Text	54.83	49.38
2	Europarl	56.30	50.94
3	Enronsent	60.34	55.21
4	Gutenberg	56.71	52.90
5	Mixed	57.78	51.62
	Total	57.10	51.85

The overall compression ratio for the corpus is summarized in Table 6.36. For the entire corpus our all methods outperforms Arithmetic Coding. WBTC-A achieves average gain of 5.25%.

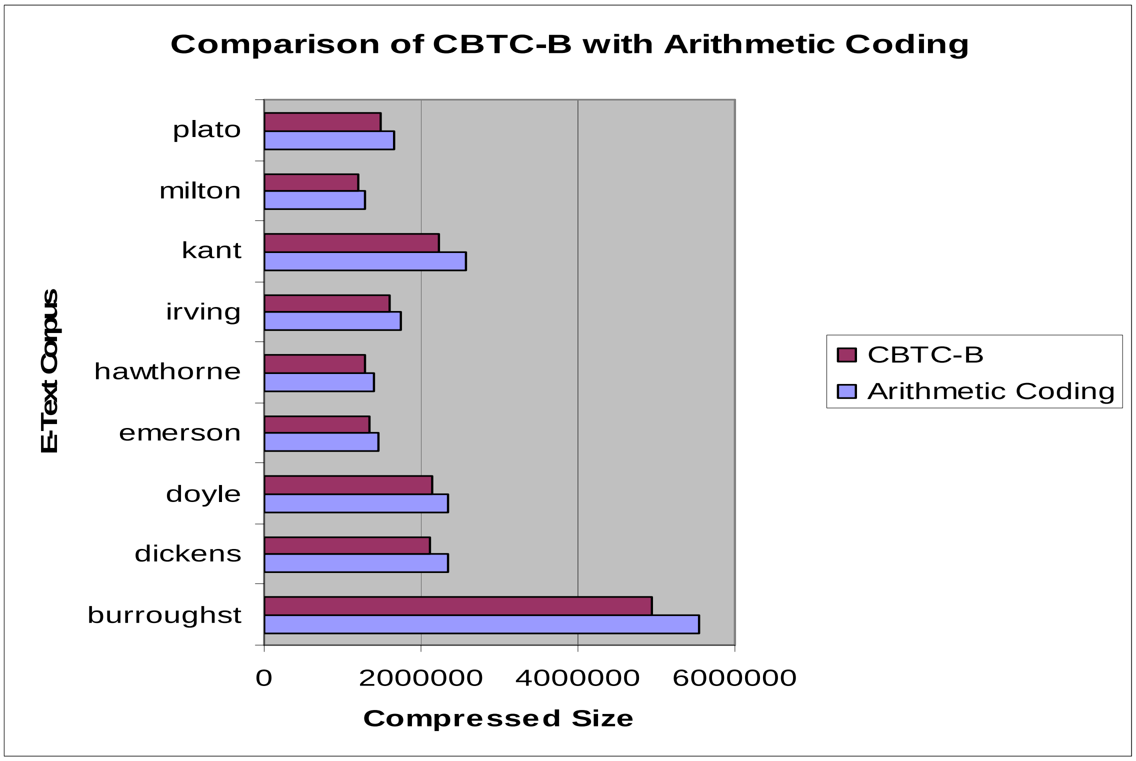

Figure 6.25 Compression ratios for E-Text Corpus (Arithmetic Coding)

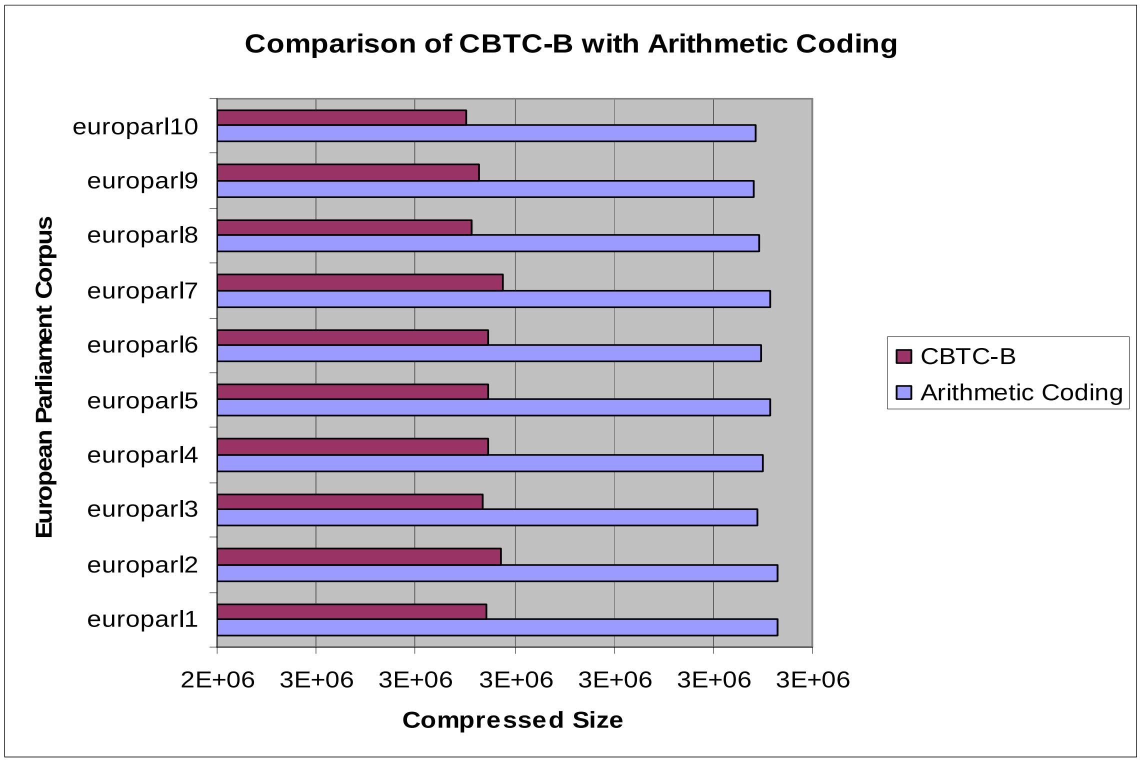

Figure 6.26 Compression ratios for European Parliament Corpus (Arithmetic Coding)

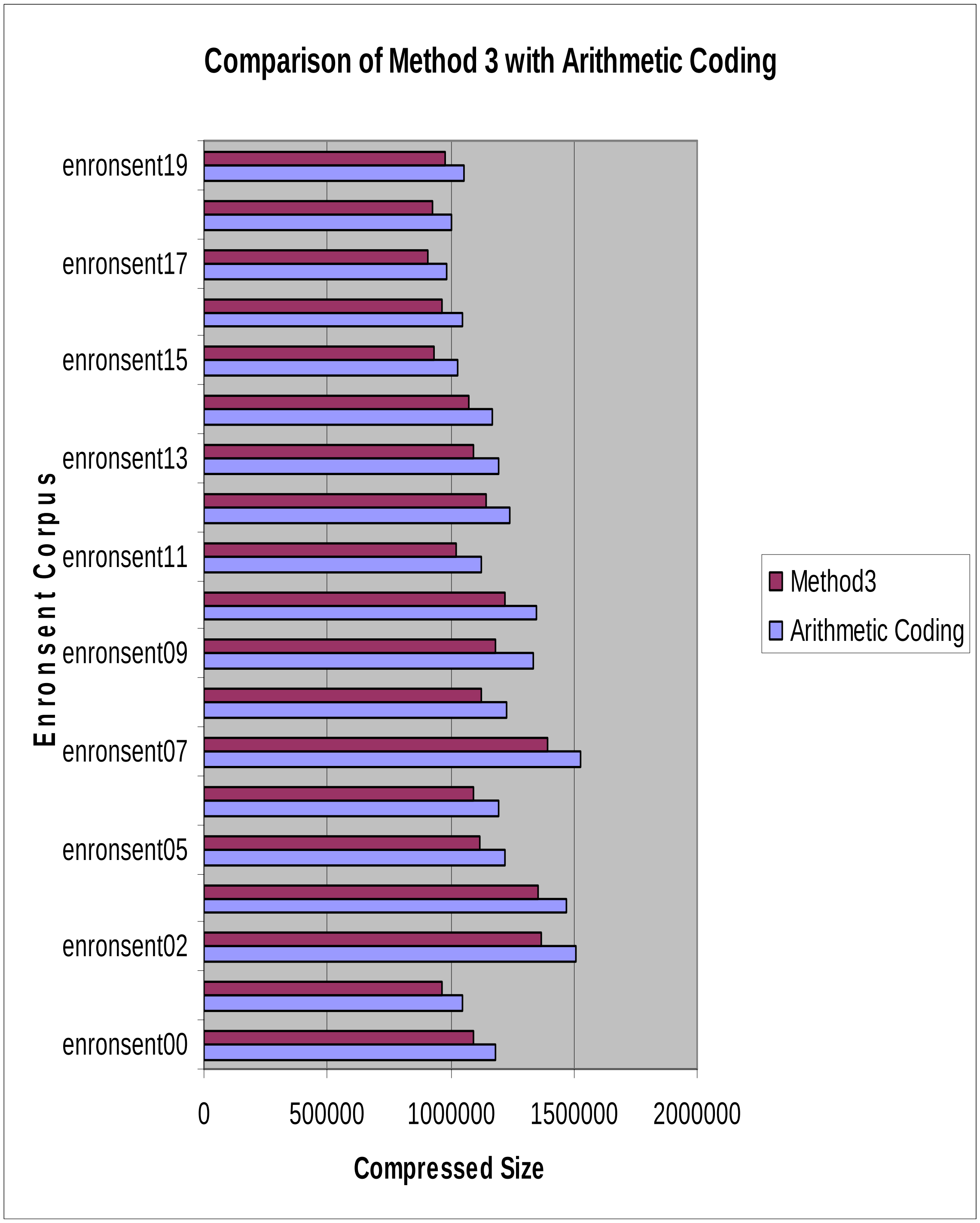

Figure 6.27 Compression ratios for Enronsent Corpus (Arithmetic Coding)

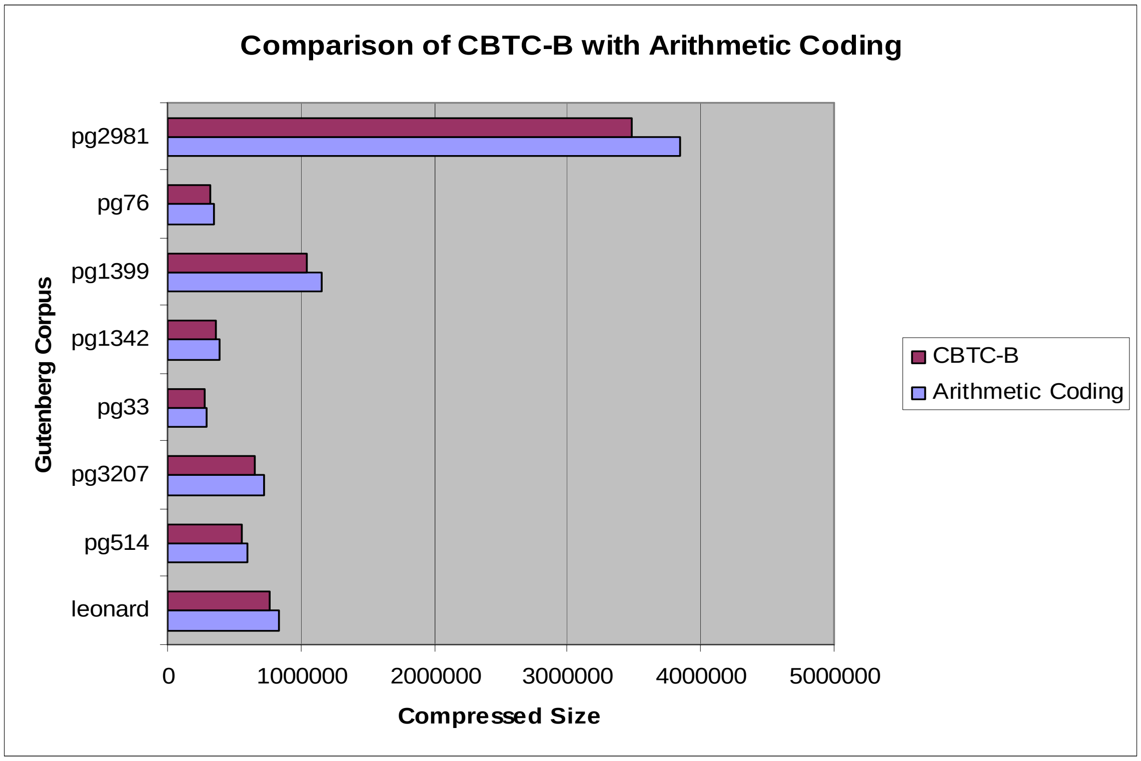

Figure 6.28 Compression ratios for Project Gutenberg Corpus (Arithmetic Coding)

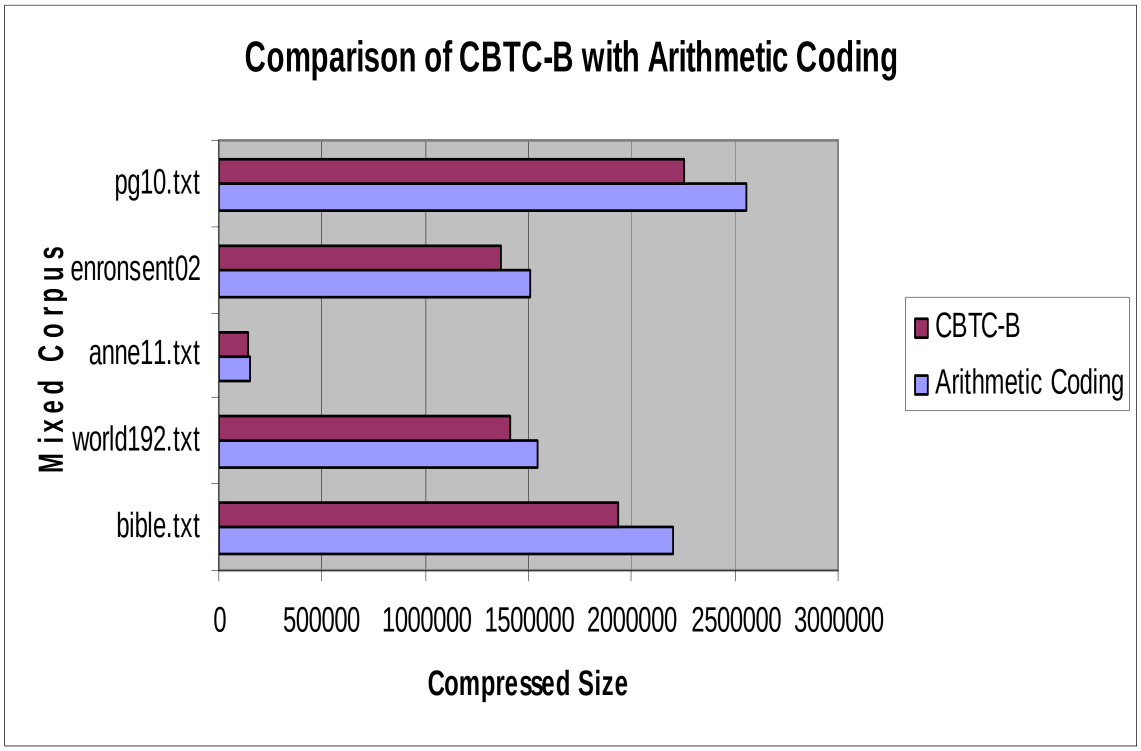

Figure 6.29 Compression ratios for Mixed Corpus (Arithmetic Coding)

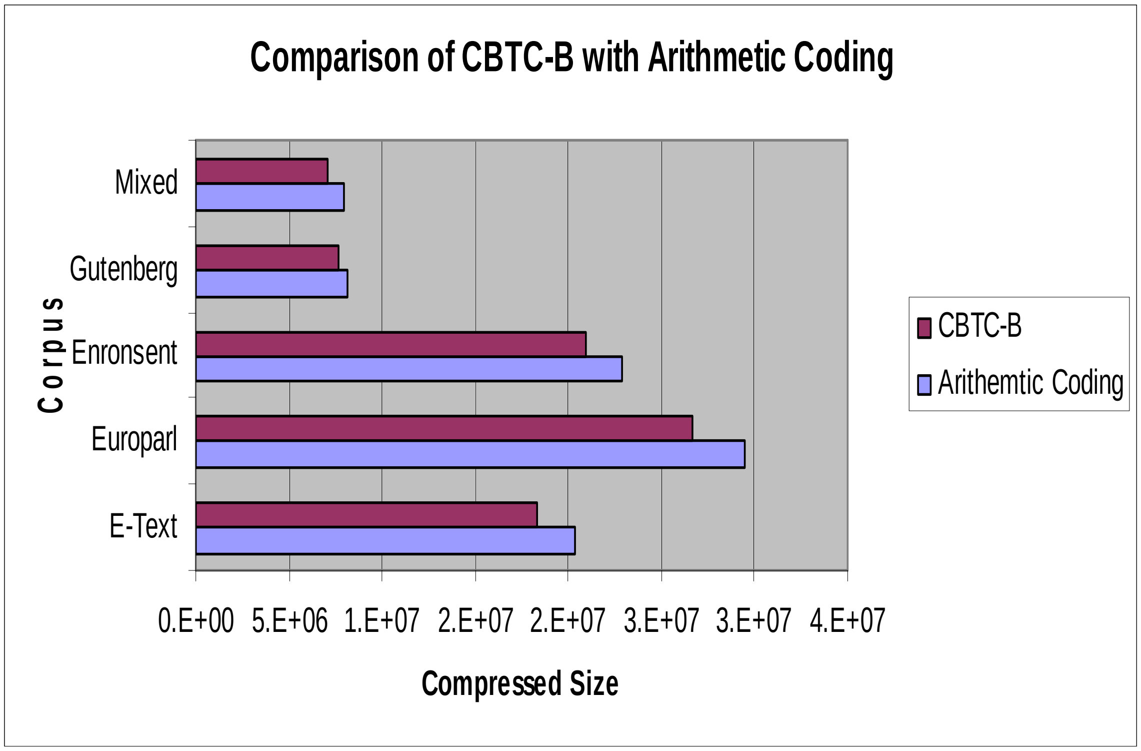

Figure 6.30 Compression ratios for all Corpus (Arithmetic Coding)

6.7 SEARCHING PHRASE IN COMPRESSED FILE

The phrase can be searched in the compressed file directly without decompressing it. The only method which can't search the phrase in the compressed is WBTC-D, because the method is compressing the file on-the-fly i.e. it uses the dynamic dictionary. All other methods such CBTC – B, WBTC – A, WBTC – C and WBTC – E are useful for directly searching the pattern in the compressed file. The test was carried out on *Bible.txt* file for five different phrases listed in Table 6.37, using Karp-Rabin algorithm, Knuth-Morris-Pratt algorithm, Brute-Force algorithm, Boyer-Moore algorithm and Quick Search algorithm. The number of comparison and time required to search the phrase is given in the following tables.

Table 6.37 Phrases for searching directly in the compressed file.

Sr.No	Phrase (From Bible.txt)
1	*“that I will not overthrow this city”*
2	*“and he begat sons”*
3	*“And Cush begat Nimrod: he began to be a mighty one in the earth.”*
4	*“Then saith he unto me, See thou do it not:”*
5	*“LORD which exercise lovingkindness”*

6.7.1 Searching phrase using Karp-Rabin Algorithm

Table 6.38 Comparison of Number of *Comparison* of searching phrases (K-R)

	Number of Comparison					
Phrase	Normal	CBTC-B	WBTC-A	WBTC-B	WBTC-C	WBTC-E
1	4047359	2784540	2447965	2434818	2407200	2462877
2	4047379	2784553	2447975	2434828	2407207	2462887
3	4047330	2784515	2447942	2434795	2407172	2462850
4	4047352	2784531	2447953	2434806	2407190	2462865
5	4047360	2784540	2447973	2434826	2407206	2462885

The test was executed on five different phrases from Bible.txt using Karp – Rabin searching algorithm and the results are shown in Table 6.38. For all the phrases, the number of comparisons required to search the phrase from the source file is comparatively very less. The graph of the same is shown in figure 6.31.

6.7.2 Searching phrase using Knuth-Morris-Pratt Algorithm

Table 6.39 Comparison of Number of *Comparison* of searching phrases (KMP)

	Number of Comparison					
Phrase	Normal	CBTC-B	WBTC-A	WBTC-B	WBTC-C	WBTC-E
1	7905121	5542617	4869766	4843244	4796515	4899401
2	7943371	5491520	4803674	4772659	4769219	4847474
3	8057761	5544582	4871440	4844926	4794244	4901148
4	8080285	5562606	4892857	4866882	4810293	4922962
5	8071498	5544543	4885220	4857977	4807202	4913838

The test was executed on five different phrases from Bible.txt using Knuth – Morris – Pratt searching algorithm and the results are shown in Table 6.39. For all the phrases, the number of comparisons required to search the phrase from the source file is comparatively very less. The graph of the same is shown in figure 6.32.

6.7.3 Searching phrase using Brute-Force Algorithm

Table 6.40 Comparison of Number of *Comparison* of searching phrases (B-F)

	Number of Comparison					
Phrase	Normal	CBTC-B	WBTC-A	WBTC-B	WBTC-C	WBTC-E
1	4535611	3045398	3020528	2876617	2459295	2875561
2	4447503	3071600	3094501	2946085	2524441	2927344
3	4101388	2828826	3018339	2874877	2466465	2873629
4	4069273	2800899	2997572	2474158	2416981	2851831
5	4079503	2819080	3007334	2862699	2430703	2861322

The test was executed on five different phrases from Bible.txt using Brute – Force searching algorithm and the results are shown in Table 6.40. For all the phrases, the number of comparisons required to search the phrase from the source file is comparatively very less. The graph of the same is shown in figure 6.33.

6.7.4 Searching phrase using Boyer-Moore Algorithm

Table 6.41 Comparison of Number of *Comparison* of searching phrases (B-M)

	Number of Comparison					
Phrase	Normal	CBTC-B	WBTC-A	WBTC-B	WBTC-C	WBTC-E
1	273094	181791	186317	175407	226627	182444
2	454241	346205	351386	316786	309123	292934
3	202058	117681	192259	181939	124314	169281
4	231102	178385	252056	245504	155997	242782
5	231727	163871	303440	297061	316005	300961

The test was executed on five different phrases from Bible.txt using Boyer - Moore searching algorithm and the results are shown in Table 6.41. For all the phrases except phrase 5, the number of comparisons required to search the phrase from the source file is comparatively very less. The graph of the same is shown in figure 6.34.

6.7.5 Searching phrase using Quick Search Algorithm

Table 6.42 Comparison of Number of *Comparison* of searching phrases (QS)

	Number of Comparison					
Phrase	Normal	CBTC-B	WBTC-A	WBTC-B	WBTC-C	WBTC-E
1	286370	213096	330333	291865	244510	307122
2	463223	420863	641785	535412	329022	487222
3	196110	113781	221497	156797	115798	151438
4	217390	149461	229106	181978	144097	185361
5	221810	163649	527188	482589	307317	474608

The test was executed on five different phrases from Bible.txt using Quick Search searching algorithm and the results are shown in Table 6.42. For CBTC-B and WBTC-C the numbers of comparisons are less. The results are not outperforming for other methods. The graph of the same is shown in figure 6.35.

6.7.6 Overall Comparison of Searching Algorithms

Table 6.43 Overall comparison of searching algorithms for proposed methods

		Number of Comparison					
Sr. No	Searching Methods	Normal	CBTC-B	WBTC-A	WBTC-B	WBTC-C	WBTC-E
1	KR	20236780	13922679	12239808	12174073	12035975	12314364
2	KMP	40058036	27685868	24322957	24185688	23977473	24484823
3	BF	21233278	14565803	15138274	14034436	12297885	14389687
4	BM	1392222	987933	1285458	1216697	**1132066**	1188402
5	QS	1384903	1060850	1949909	1648641	1140744	1605751

The summary of the number of comparisons of searching algorithms for different proposed methods is given in Table 6.43. It is seen that the most effective method for searching and retrieval of phrases from the compressed form is WBTC-C by using Boyer-Moore algorithm.

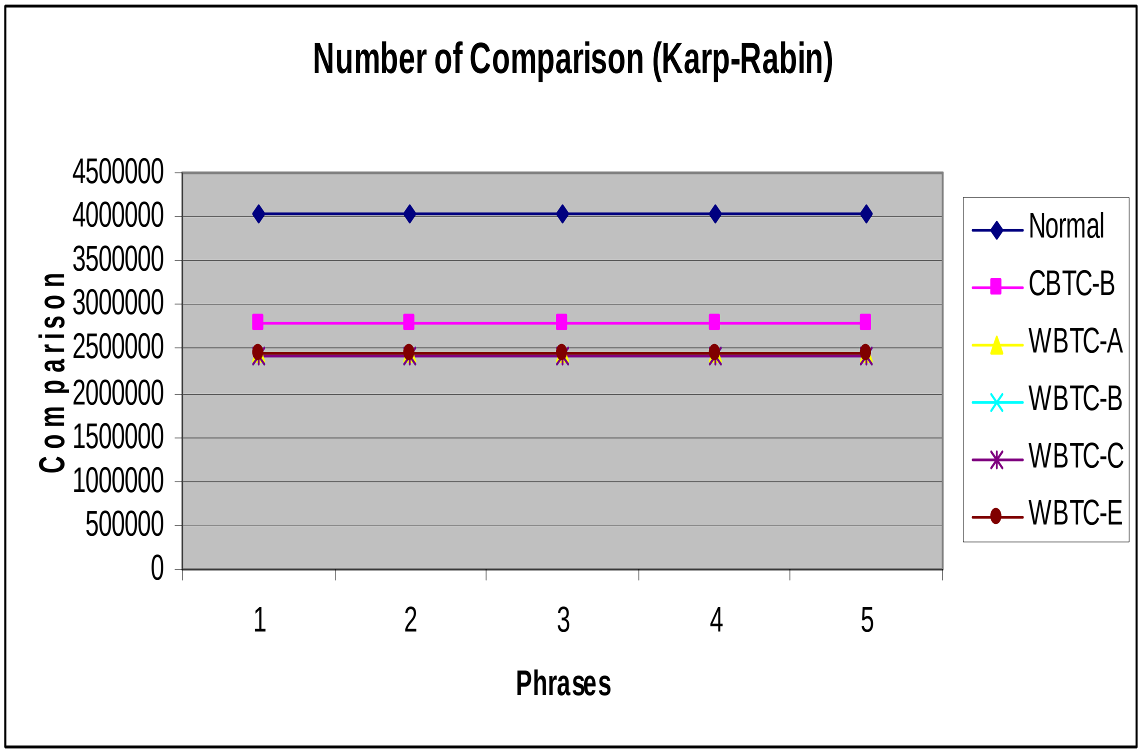

Figure 6.31 Number of Comparisons for Normal and Proposed Methods (K-R)

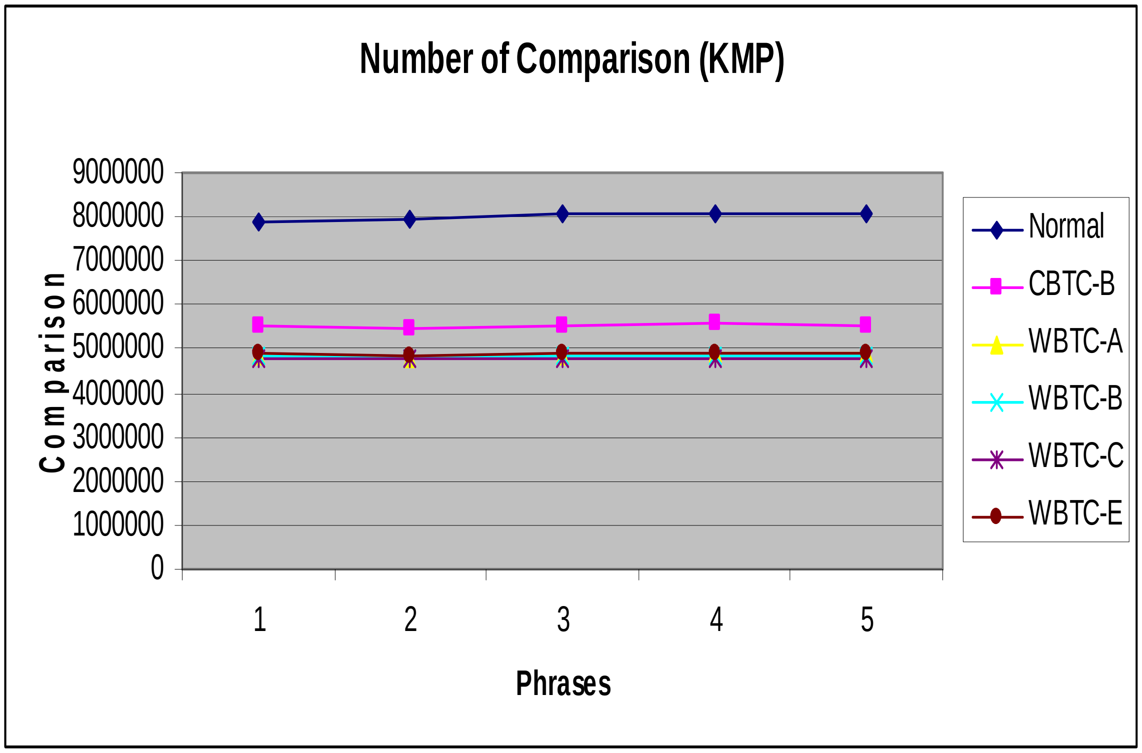

Figure 6.32 Number of Comparisons for Normal and Proposed Methods (KMP)

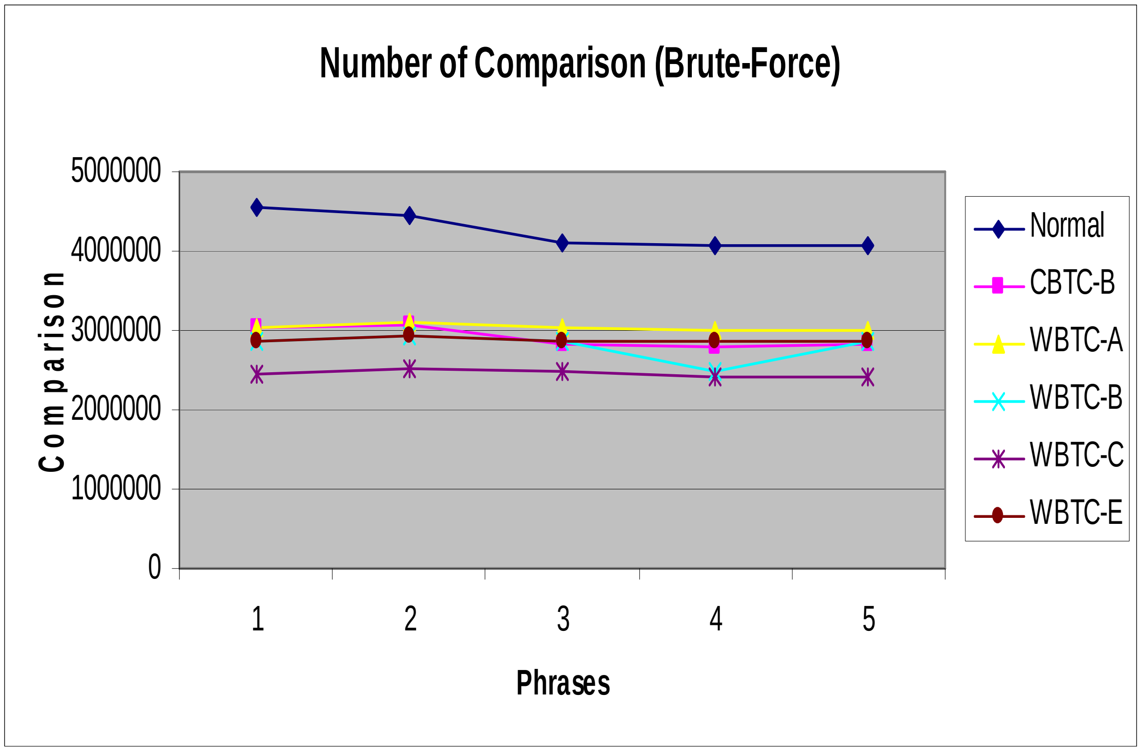

Figure 6.33 Number of Comparisons for Normal and Proposed Methods (BF)

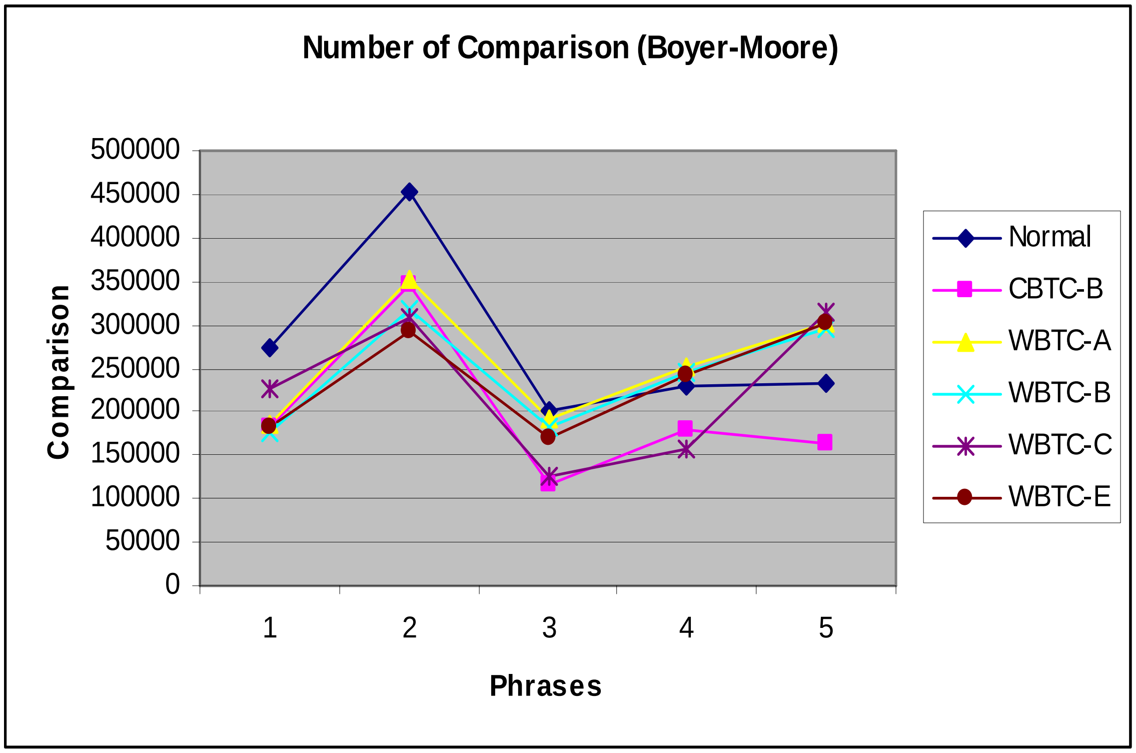

Figure 6.34 Number of Comparisons for Normal and Proposed Methods (BM)

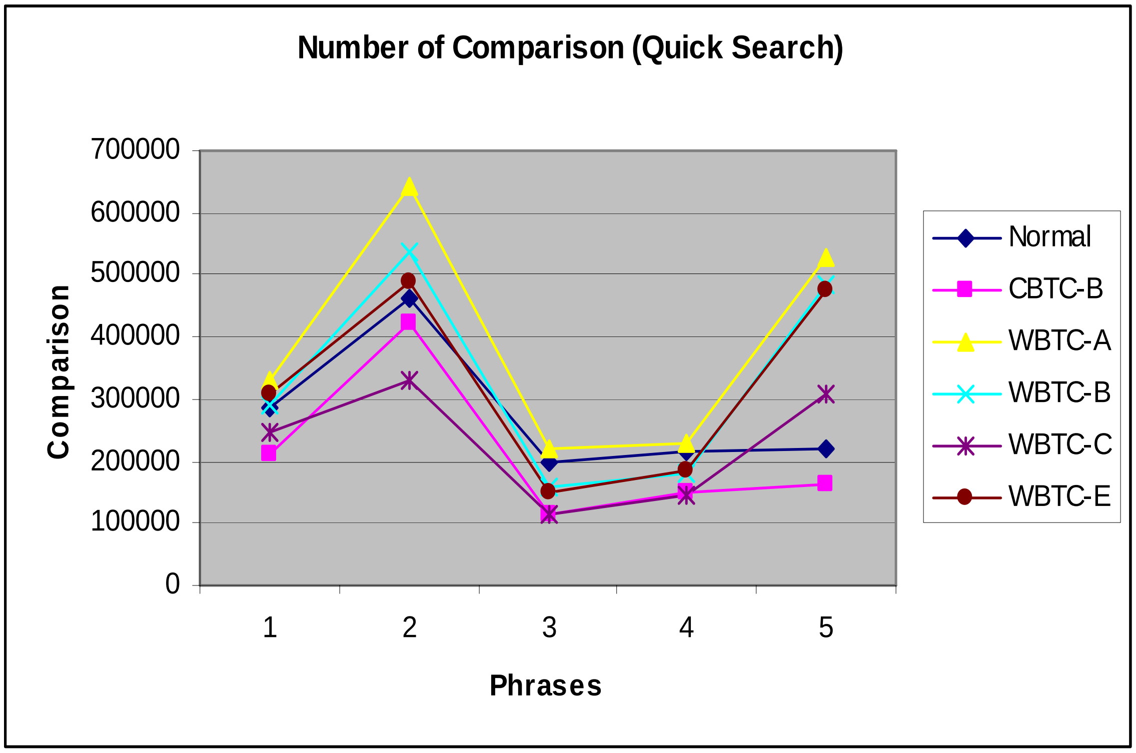

Figure 6.35 Number of Comparisons for Normal and Proposed Methods (QS)

6.8 DECOMPRESSION TIME

The time required for decompressing the file using Bzip method and using WBTC-C method is computed for all corpuses. Five times the files were decompressed and the average decompression time is computed. The results are shown in following tables.

6.8.1 Decompression Time for E-Text Corpus

Table 6.44 Decompression Time for E-Text Corpus (Bzip method)

Bzip		Decompression Time (DT) in milliseconds					
Sr.No	File Name	DT1	DT2	DT3	DT4	DT5	Average
1	burroughst	1594	1610	1594	1594	1593	1597.00
2	dickens	734	703	719	719	704	715.80
3	doyle	828	750	703	703	703	737.40
4	emerson	547	500	500	500	500	509.40
5	hawthorne	469	469	469	469	468	468.80
6	irving	578	609	547	546	547	565.40
7	kant	735	734	734	750	766	743.80
8	milton	390	391	391	407	391	394.00
9	plato	516	516	515	531	515	518.60
	Total	6391	6282	6172	6219	6187	6250.20

Table 6.45 Decompression Time for E-Text Corpus (WBTC-C method)

WBTC-C		Decompression Time (DT) in milliseconds					
Sr.No	File Name	DT1	DT2	DT3	DT4	DT5	Average
1	burroughst	1219	1203	1203	1218	1203	1209.20
2	dickens	656	640	657	641	625	643.80
3	doyle	672	641	625	609	641	637.60
4	emerson	469	453	453	484	453	462.40
5	hawthorne	422	437	421	437	407	424.80
6	irving	469	485	469	469	484	475.20
7	kant	579	594	594	593	594	590.80
8	milton	375	360	344	344	345	353.60
9	plato	468	453	469	453	453	459.20
	Total	5329	5266	5235	5248	5205	5256.60

The test for calculating decompression time was executed on E-Text Corpus for both methods Bzip2 and WBTC-C, and the results are shown in Table 6.44 and Table 6.45. For the entire corpus, the total decompression time required for WBTC-C is less than Bzip2 method. The time required to decompress WBTC-C compressed file is less than 15.90% by the time required to decompress Bzip2 compressed file.

6.8.2 Decompression Time for European Parliament Corpus

Table 6.46 Decompression Time for European Parliament Corpus (Bzip method)

Bzip		Decompression Time (DT) in milliseconds					
Sr.No	File Name	DT1	DT2	DT3	DT4	DT5	Average
1	europarl1	812	797	812	797	813	806.20
2	europarl2	813	859	828	875	797	834.40
3	europarl3	812	828	860	812	937	849.80
4	europarl4	813	813	906	813	828	834.60
5	europarl5	844	812	812	860	813	828.20
6	europarl6	812	828	828	828	812	821.60
7	europarl7	859	875	829	875	828	853.20
8	europarl8	813	813	796	797	938	831.40
9	europarl9	812	828	954	828	828	850.00
10	europarl10	829	812	796	797	906	828.00
	Total	8219	8265	8421	8282	8500	8337.40

Table 6.47 Decompression Time for European Parliament Corpus (WBTC-C method)

WBTC-C		Decompression Time (DT) in milliseconds					
Sr.No	File Name	DT1	DT2	DT3	DT4	DT5	Average
1	europarl1	641	640	625	625	625	631.20
2	europarl2	625	625	750	626	641	653.40
3	europarl3	657	641	625	655	625	640.60
4	europarl4	797	625	641	641	782	697.20
5	europarl5	625	641	640	641	624	634.20
6	europarl6	656	639	625	625	641	637.20
7	europarl7	656	625	672	641	641	647.00
8	europarl8	626	657	625	625	625	631.60
9	europarl9	639	625	640	703	641	649.60
10	europarl10	641	641	626	641	640	637.80
	Total	6563	6359	6469	6423	6485	6459.80

The test for calculating decompression time was executed on European Parliament Corpus for both methods Bzip2 and WBTC-C, and the results are shown in Table 6.46 and Table 6.47. For the entire corpus, the total decompression time required for WBTC-C is less than Bzip2 method. The time required to decompress WBTC-C compressed file is less than 22.52% by the time required to decompress Bzip2 compressed file.

6.8.3 Decompression Time for Enronsent Corpus

Table 6.48 Decompression Time for Enronsent Corpus (Bzip2 method)

Bzip		Decompression Time (DT) in milliseconds					
Sr.No	File Name	DT1	DT2	DT3	DT4	DT5	Average
1	enronsent00	359	359	360	360	359	359.40
2	enronsent01	328	329	328	328	329	328.40
3	enronsent02	438	468	609	437	546	499.60
4	enronsent03	375	375	375	375	375	375.00
5	enronsent04	391	422	391	406	391	400.20
6	enronsent05	375	360	359	532	359	397.00
7	enronsent06	437	359	359	359	375	377.80
8	enronsent07	422	421	563	469	610	497.00
9	enronsent08	359	360	359	359	360	359.40
10	enronsent09	297	297	297	281	281	290.60
11	enronsent10	360	343	344	344	359	350.00
12	enronsent11	360	329	312	328	329	331.60
13	enronsent12	421	359	484	359	421	408.80
14	enronsent13	360	359	344	407	360	366.00
15	enronsent14	484	344	344	359	344	375.00
16	enronsent15	313	313	296	297	296	303.00
17	enronsent16	312	328	329	312	313	318.80
18	enronsent17	313	297	296	297	312	303.00
19	enronsent18	328	312	1454	312	406	562.40
	Total	7032	6734	8203	6921	7125	7203.00

Table 6.49 Decompression Time for Enronsent Corpus (WBTC-C method)

WBTC-C		Decompression Time (DT) in milliseconds					
Sr.No	File Name	DT1	DT2	DT3	DT4	DT5	Average
1	enronsent00	375	344	343	345	391	359.60
2	enronsent01	343	344	344	343	328	340.40
3	enronsent02	406	391	422	421	422	412.40
4	enronsent03	438	344	359	376	360	375.40
5	enronsent04	406	1390	390	374	390	590.00
6	enronsent05	344	344	345	391	344	353.60
7	enronsent06	359	358	530	437	358	408.40
8	enronsent07	406	392	391	407	407	400.60
9	enronsent08	360	374	360	359	344	359.40
10	enronsent09	297	281	297	297	312	296.80
11	enronsent10	375	392	343	360	438	381.60
12	enronsent11	344	343	344	344	343	343.60
13	enronsent12	359	343	360	359	391	362.40
14	enronsent13	360	376	359	453	344	378.40
15	enronsent14	390	1343	360	344	360	559.40
16	enronsent15	311	329	312	328	1296	515.20
17	enronsent16	345	328	328	328	327	331.20
18	enronsent17	327	1329	359	344	329	537.60
19	enronsent18	312	328	329	328	328	325.00
	Total	6857	9673	6875	6938	7812	7631.00

The test for calculating decompression time was executed on Enronsent Corpus for both methods Bzip2 and WBTC-C, and the results are shown in Table 6.48 and Table 6.49. In the case of Enronsent corpus only, the total decompression time required for WBTC-C is slightly more than Bzip2 method. The time required to decompress WBTC-C compressed file is more than5.94% by the time required to decompress Bzip2 compressed file.

6.8.4 Decompression Time for Project Gutenberg Corpus

Table 6.50 Decompression Time for Project Gutenberg Corpus (Bzip method)

Bzip		Decompression Time (DT) in milliseconds					
Sr.No	File Name	DT1	DT2	DT3	DT4	DT5	Average
1	leonard	282	282	281	281	1281	481.40
2	pg1342	218	172	172	172	172	181.20
3	pg1399	500	390	375	391	1375	606.20
4	pg2981	1125	1125	1110	1110	1125	1119.00
5	pg3207	250	250	250	250	250	250.00
6	pg33	250	141	140	140	140	162.20
7	pg514	266	281	250	250	235	256.40
8	pg76	156	156	250	172	156	178.00
	Total	3047	2797	2828	2766	4734	3234.40

Table 6.51 Decompression Time for Project Gutenberg Corpus (WBTC-C method)

WBTC-C		Decompression Time (DT) in milliseconds					
Sr.No	File Name	DT1	DT2	DT3	DT4	DT5	Average
1	leonard	297	312	312	313	312	309.20
2	pg1342	220	281	219	203	203	225.20
3	pg1399	374	359	359	360	375	365.40
4	pg2981	891	890	922	906	890	899.80
5	pg3207	265	297	312	282	297	290.60
6	pg33	204	203	219	251	203	216.00
7	pg514	265	265	265	266	313	274.80
8	pg76	218	204	219	312	218	234.20
	Total	2734	2811	2827	2893	2811	2815.20

The test for calculating decompression time was executed on Project Gutenberg Corpus for both methods Bzip2 and WBTC-C, and the results are shown in Table 6.50 and Table 6.51. For the entire corpus, the total decompression time required for WBTC-C is less than Bzip2 method. The time required to decompress WBTC-C compressed file is less than 12.96% by the time required to decompress Bzip2 compressed file.

6.8.5 Decompression Time for Mixed Corpus

Table 6.52 Decompression Time for Mixed Corpus (Bzip method)

Bzip		Decompression Time (DT) in milliseconds					
Sr.No	File Name	DT1	DT2	DT3	DT4	DT5	Average
1	bible.txt	671	625	625	687	609	643.40
2	world192.txt	422	421	421	422	422	421.60
3	anne11.txt	110	110	110	109	109	109.60
4	enronsent02	438	468	609	437	546	499.60
5	pg10.txt	687	687	687	672	688	684.20
	Total	2328	2311	2452	2327	2374	2358.40

Table 6.53 Decompression Time for Mixed Corpus (WBTC-C method)

WBTC-C		Decompression Time (DT)					
Sr.No	File Name	DT1	DT2	DT3	DT4	DT5	Average
1	bible.txt	516	517	531	516	501	516.20
2	world192.txt	453	422	438	406	422	428.20
3	anne11.txt	187	186	187	173	187	184.00
4	enronsent02	406	391	422	421	422	412.40
5	pg10.txt	594	578	578	609	594	590.60
	Total	2156	2094	3173	2125	2126	2334.80

The test for calculating decompression time was executed on Mixed Corpus for both methods Bzip2 and WBTC-C, and the results are shown in Table 6.52 and Table 6.53. For the entire corpus, the total decompression time required for WBTC-C is slightly less than Bzip2 method. The time required to decompress WBTC-C compressed file is less than 1% by the time required to decompress Bzip2 compressed file.

6.8.6 Decompression Time for All Corpus

Table 6.54 Decompression Time for All corpuses (Bzip method)

Bzip		Decompression Time (DT) in milliseconds					
Sr.No	File Name	DT1	DT2	DT3	DT4	DT5	Average
1	Common	2328	2311	2452	2327	2374	2358.40
2	Authors	6391	6282	6172	6219	6187	6250.20
3	Europarl	8219	8265	8421	8282	8500	8337.40
4	Enronsent	7032	6734	8203	6921	7125	7203.00
5	Gutenberg	3047	2797	2828	2766	4734	3234.40
	Total	27017	26389	28076	26515	28920	27383.40

Table 6.55 Decompression Time for All corpuses (WBTC-C method)

WBTC-C		Decompression Time (DT) in milliseconds					
Sr.No	File Name	DT1	DT2	DT3	DT4	DT5	Average
1	Common	2156	2094	3173	2125	2126	2334.80
2	Authors	5329	5266	5235	5248	5205	5256.60
3	Europarl	6563	6359	6469	6423	6485	6459.80
4	Enronsent	6857	9673	6875	6938	7812	7631.00
5	Gutenberg	2734	2811	2827	2893	2811	2815.20
	Total	23639	26203	24579	23627	24439	24497.40

The overall decompression time required for all corpuses for both methods Bzip2 and WBTC-C are shown in Table 6.54 and Table 6.55. For all corpuses, the total decompression time required for WBTC-C is less than Bzip2 method. The time required to decompress WBTC-C compressed file is less than 10.55% by the time required to decompress Bzip2 compressed file. The graphs of decompression time for Bzip2 and WBTC-C methods are shown in Figure 6.36 to Figure 6.41 for all corpuses.

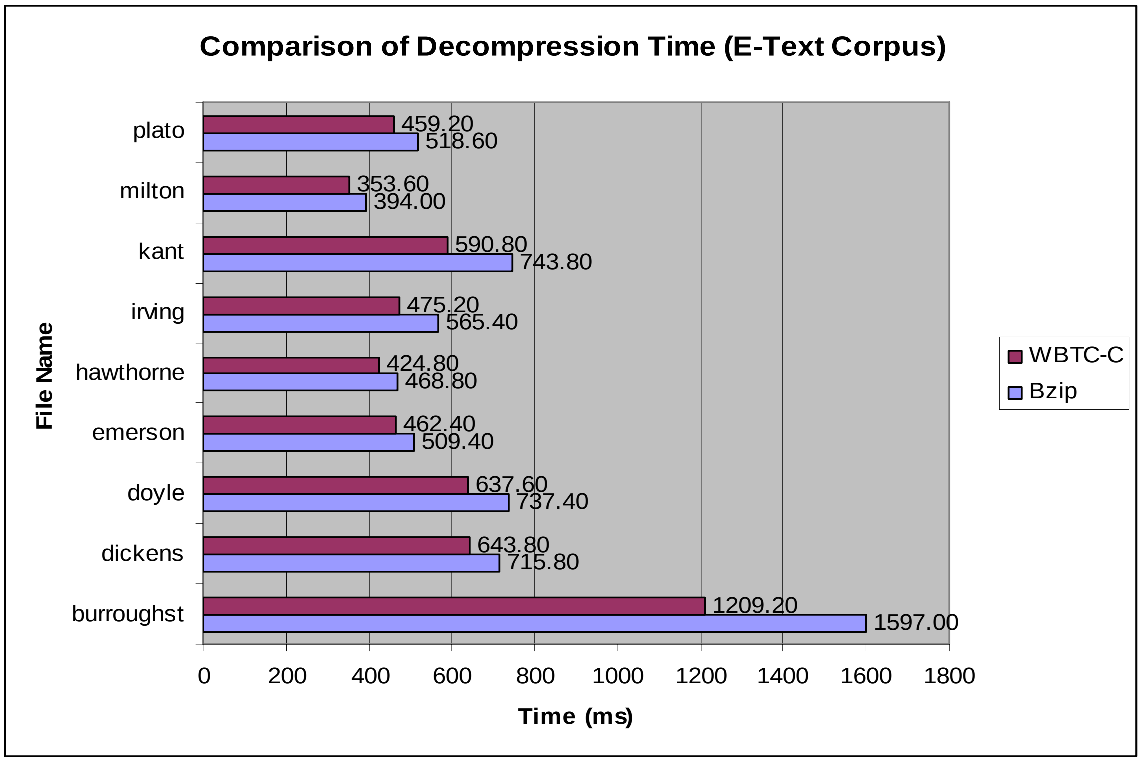

Figure 6.36 Decompression time for E-Text Corpus

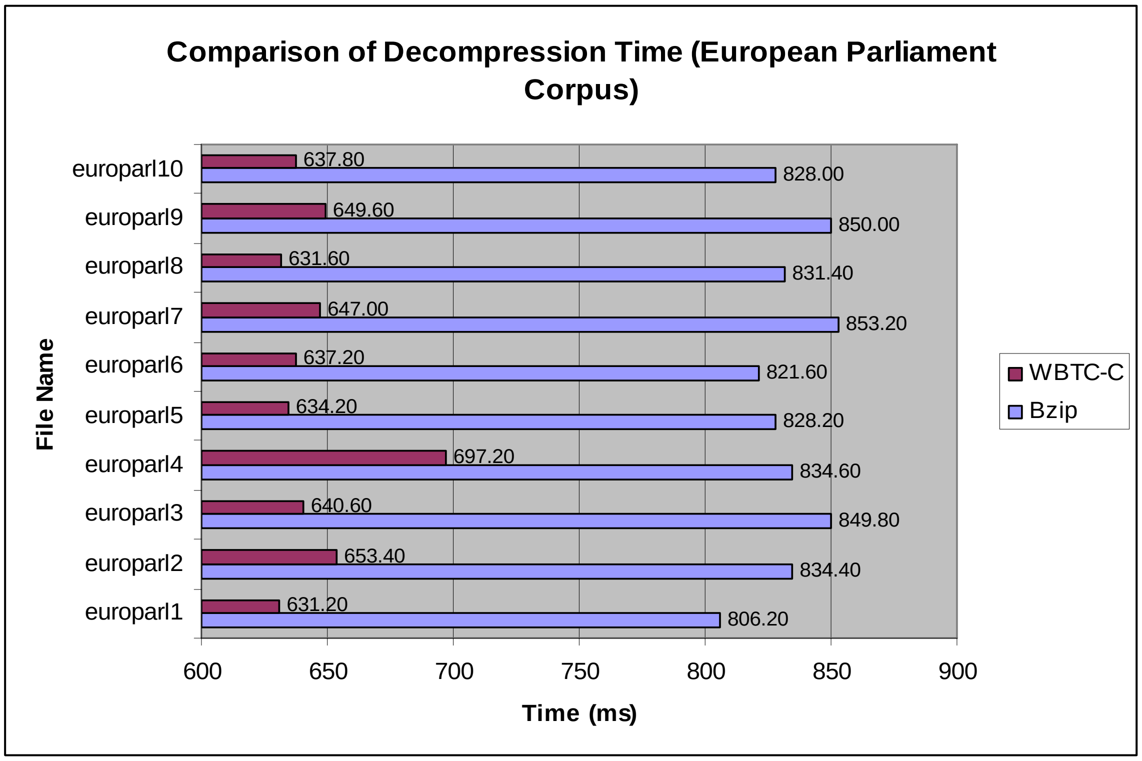

Figure 6.37 Decompression time for European Parliament Corpus

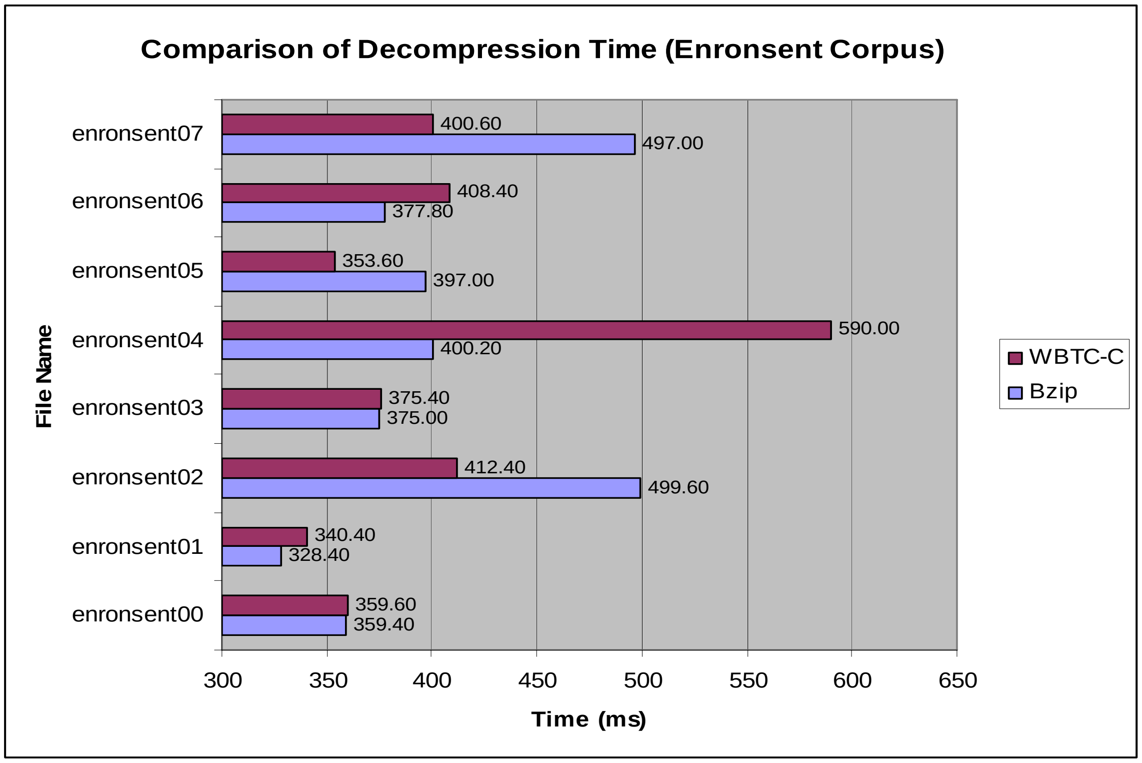

Figure 6.38 Decompression time for Enronsent Corpus

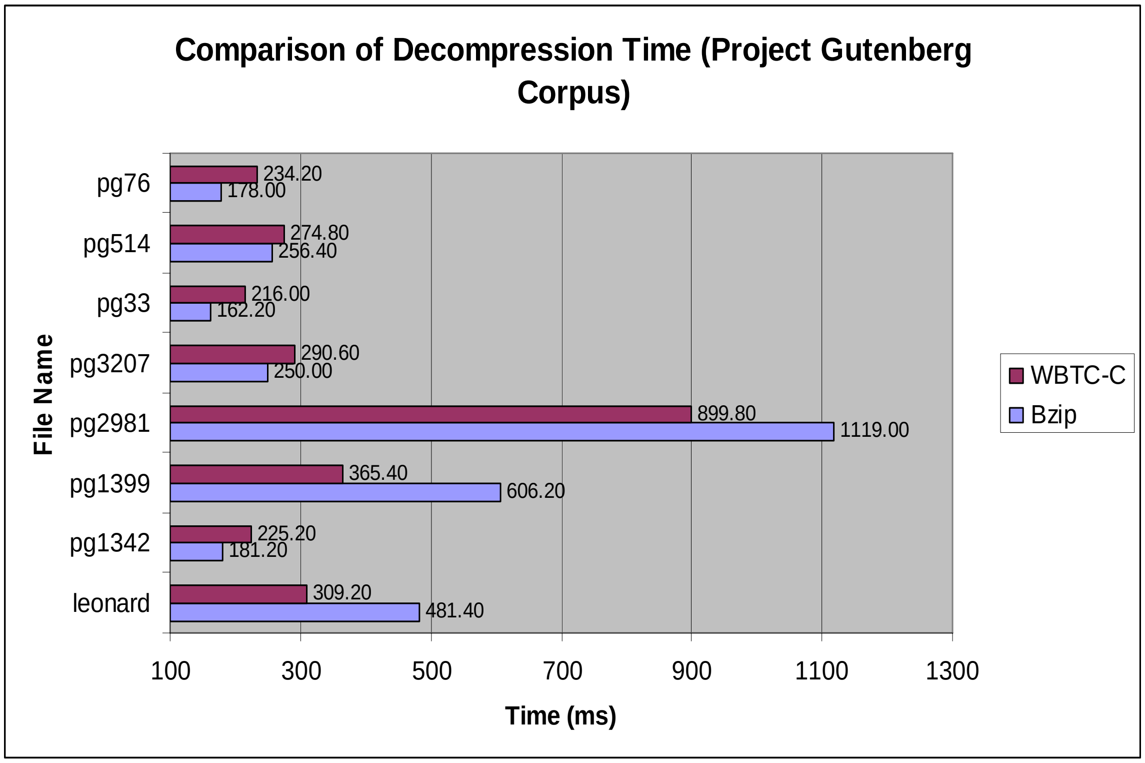

Figure 6.39 Decompression time for Project Gutenberg Corpus

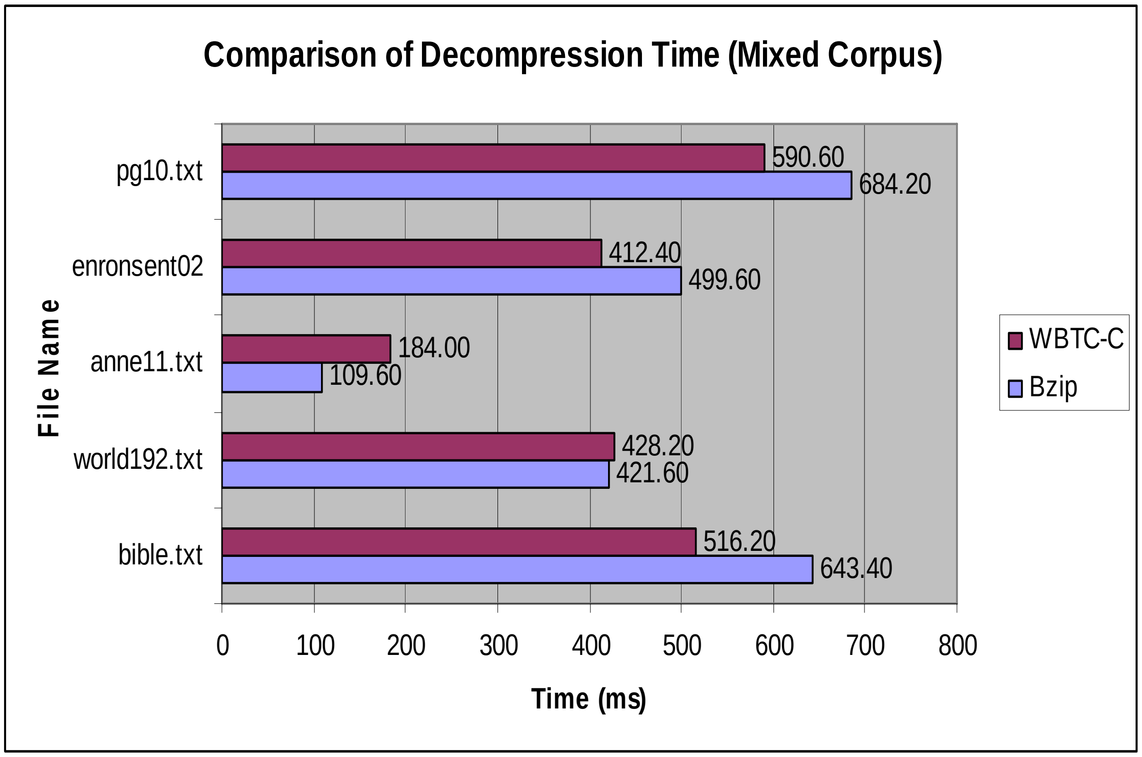

Figure 6.40 Decompression time for Mixed Corpus

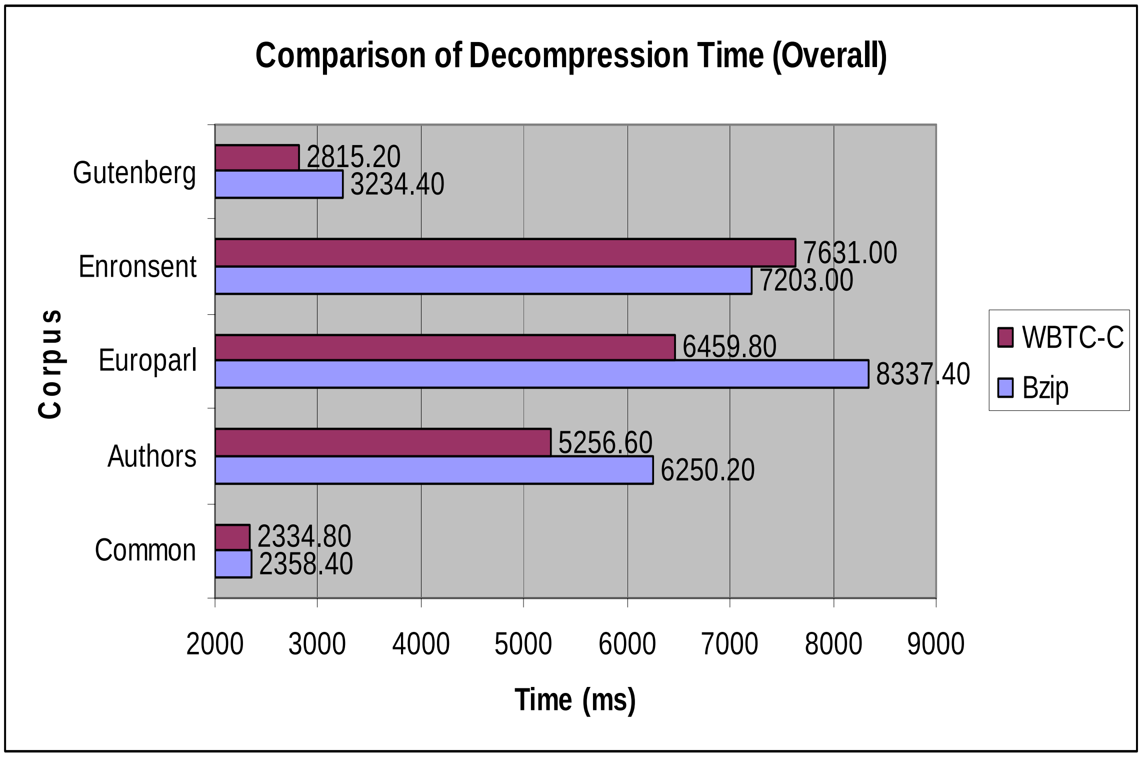

Figure 6.41 Decompression time for All Corpus

CHAPTER 7

CONCLUSION AND FUTURE WORK

7

CONCLUSION AND FUTURE WORK

We had compared the performance of our algorithms to other text compression algorithms, including standard compression algorithms such as Arithmetic Coding, Bzip2, PPMd, PPMII and LZMA. We have tested our algorithms on 51 different text files of different corpus. Our results show that in most cases our preprocessing algorithms lead to significant improvement in compression ratio. The compression ratio of our methods combined with a standard compression algorithm is typically 0.21% to 6.16% higher than that of the same standard compression algorithm when used alone. All methods proposed by us are language dependent and are useful for text files.

For large files the compression ratio improves where as for small files the compression ratio bit deteriorates. Three types of dictionaries viz., static, semi-dynamic and dynamic dictionary are used by us in proposed techniques. Except method WBTC-E all other methods proposed by us are useful for direct searching the phrases in the compressed file. The phrase to be searched is to be compressed first by respective method and then using standard matching algorithm, we can search and retrieve the phrase directly from the compressed file. From experimental results it is seen that the number of comparisons normally required to search the phrases from a compressed file is less than that of comparison required to search the phrase from a normal decompressed file. We have used five methods to check this viz. Karp-Rabin algorithm, Knuth-Morris-Pratt algorithm, Brute-Force algorithm, Boyer – Moore algorithm and Quick Search algorithm. In all five algorithms it is seen that the number of comparisons to search the phrase in compressed file are less than that searching the phrase from normal file.

We had compared among themselves all methods proposed by us and after considering the different parameters such as compression ratio, generalness and suitability for searching, we come to conclusion that method WBTC-C is optimum choice for compressing the text file.

The features of WBTC-C are that it uses two-dimensional dictionary. The total numbers of codes representing the words are less than the method using the single dimension dictionary. The dictionary is semi-dynamic i.e. created for a particular file and is a part of compressed file and hence useful for searching the phrase directly from the compressed file. The compression ratio is improved by 0.46% to 2.12% (on an average of 1.28%) when combined with standard compression techniques such as Bzip2, PPMd, PPMII and LZMA. We can retrieve the data randomly from any point in the compressed file. The only limitation is that the retrieval can be done from the point where there is a change in row, so that we will know the row number and then from that point onward the data can be retrieved.

During compression process we store a unique symbol 0xFF (i.e. 255) to indicate change in row followed by row number. If we try to retrieve from any random location, then at that point we are not knowing the current row number, and without current row number we cannot retrieve the words from the dictionary, therefore first we have to scan the compressed file for code of change in row, because after that the new row number is stored in compressed file. Once we get the new row number, then from that point onwards we will keep track of row number so that we will be able to decompress the words from the dictionary with the column number reads from the compressed file. The method will fail in the case if there is no change in row number found in the compressed file from the point from where we want to decompressed and retrieve the data. The probability of such case will occur only if we will try to retrieve the data near to the end of file. The decompression time required is also improved in WBTC-C method.

If we compare WBTC-C with other methods WBTC-A and WBTC-B, where the semi-dynamic dictionary used is of single dimension, we find that the compression ratio of former is better than the later ones when used with Bzip2 and PPMd. The compression ratio of WBTC-D is more than that of WBTC-C, but it is not useful for direct searching the phrases in the compressed form, as it is using the dynamic dictionary which is implicitly build during the compression process and is not stored along with compressed file, instead while decompressing the same kind of dynamic dictionary is build up during the decompression process.

The last method WBTC-E proposed by us is giving an average improvement of 8.74% when used as pre-stage compression technique to standard compression techniques such as Bzip2, PPMd, PPMII and LZMA. This technique is also using two-dimensional static dictionary similar to that of WBTC-C. The searching of phrase directly in the compressed file is also feasible in this technique. The only drawback of this technique is that it is suitable for text files from a particular application domain, because in this technique, an already created static dictionary is used instead of semi-dynamic dictionary.

For example, if we are having a history of medical records of patients in text files, then in all the records the words related to medical fields will exists. Most of the words will be repeated in all the text files. Static dictionary will be created from all the records and then if we compress a single file by using this static dictionary, the compression ratio will be outperforming. But at the same time, if we try to compress an another file not belonging to the medical category, then even if we compress that file using static dictionary, we won't be able to find the words in the static dictionary and hence compression ratio won't be effective. That's why we have said that method WBTC-E is suitable only to a particular application domain, whereas in method WBTC-C an dictionary is created for that particular file only which is to be compressed.

If we compare the compression ratio of WBTC-C and WBTC-E, then no doubt the compression ratio of WBTC-E is comparatively more, but WBTC-C is applicable to any kind of text file of any particular application domain, and also the compression ratio of WBTC-C is better than CBTC-B, WBTC-A, WBTC-B. Therefore, we can say that the method WBTC-C is the most versatile and robust among other methods proposed by us.

Another method CBTC-B proposed by us is giving better compression ratio when used as pre-stage compression to Arithmetic Coding. The compression ratio achieved by this method is improved by 5.38% as compared to Arithmetic Coding method when used alone.

In WBTC-C method we have used 8-bit length to encode the words whereas in WBTC-E method we have used 16-bit length to encode the words. Therefore the number of words we had kept in the dictionary are 16,446 and 1,62,815 respectively. The maximum file size from our corpus is up to 10MB. After analyzing the word statistics of the corpus, we come to conclusion that the numbers of words we have considered are sufficient for the corpus taken and hence the length of encoding bits.

A number of areas for further development remain. It will be interesting to see the effect of compression ratio if the dimension of the dictionary is increased from two-dimension to three-dimension. The encoding length of word from the dictionary can be increased from 16-bit to 32-bit thereby increasing the number of words to be kept in the dictionary.
The present implementation, keeps the dictionaries separately from the compressed file. What will be the effect on compression ratio if dictionaries are merged in the compressed file? Our implementation is executed as a separate process from the standard compression techniques. It will be interesting to see the effect on the compression ratio if our method implementations can be embedded with the implementation of Bzip2, PPMd, PPMII and LZMA, so that an integrated compression can be performed on text files.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] TREC. “Official webpage for TREC - Text REtrieval Conference series. http://trec.nist.gov." 2000.

[2] P. Lyman, H. R. Varian, Kirsten Swearingen, Peter Charles, Nathan Good, Laheem Lamar Jordan, and Joyojeet Pal. “How Much Information? 2003, http://www.sims.berkeley.edu/research/projects/how-much-info-2003." Technical report, School of Information Management and Systems at the University of California at Berkeley, 2003.

[3] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[4] William B. Franks and Ricardo Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice Hall PTR, 1992.

[5] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufman, second edition, 1999.

[6] R.S. Boyer and J.S. Moore. “A fast string searching algorithm.” *Communications of the ACM*, 20(10):762-772, October 1977.

[7] D.E. Knuth, J.H. Morris, and V.R. Pratt. “Fast pattern matching in strings." *SIAM Journal of Computing*, 6(2):323-350, June 1977.

[8] David Salomon. “*Data Compression: The Complete Reference*.” Springer-Verlag, 2nd edition, 2000.

[9] D. A. Lelewer and D. S. Hirschberg, “Data Compression”, ACM Computing Surveys, 19(3):261-296, September 1987.

[10] Witten, R. Neal, and J. Cleary, “Arithmetic Coding for Data Compression”, *Communications of the ACM*, Vol. 30(6), pp. 520-540, June 1987.

[11] T. Bell, J. Cleary, I. Witten, “*Text Compression*”, Prentice Hall, 1990

[12] C.E. Shannon. “Prediction and Entropy of Printed English." *Bell Systems Technical Journal*, 30:55, 1951.

[13] D.A. Huffman. A Method for the Construction of Minimum Redundancy Codes." *Proc. IRE*, 40(9):1098-1101, September 1952.

[14] Md. Ziaul Karim Zia, Dewan Md. Fayzur Rahman, and Chowdhury Mofizur Rahman “Two-Level Dictionary Based Text Compression Scheme”, Proceedings of 11th

International Conference on Computer and Information Technology (ICCIT 2008), Khulna, Bangladesh December, 2008, pp. 13-18

[15] J. Rissanen and G. G. Langdon. “Arithmetic Coding”, IBM *Journal of Research and Development*, 23(2):149-162, 1979.

[16] Khalid Sayood. “*Introduction to Data Compression”*. Morgan Kaufmann, 2nd edition, 2000.

[17] J.G. Cleary and I.H. Witten. “Data Compression using Adaptive Coding and Partial String Matching." *IEEE Transactions on Communications*, COM-32:396-402, April 1984.

[18] A. Moffat. “Linear Time Adaptive Arithmetic Coding." *IEEE Transactions on Information Theory*, 36(2):401-406, March 1990.

[19] Linda A Curl and Brent J Hussing. “Introductory Computing: a New Approach." In *Proc. SIGCSE 93*, pp. 131-135, March 1993.

[20] T.C. Bell and A. Moffat. “A Note on the DMC Data Compression Scheme." *Computer Journal*, 32(1):16-20, February 1989.

[21] J. Ziv and A. Lempel. “A Universal Algorithm for Sequential Data Compression." *IEEE Transactions on Information Theory*, IT-23:337-343, 1977.

[22] J. Ziv and A. Lempel. “Compression of Individual Sequences via Variable Rate Coding." *IEEE Transactions on Information Theory*, IT-24:530-536, 1978.

[23] T.A. Welch. “A Technique for High Performance Data Compression." *IEEE Computer*, 17:8-20, June 1984.

[24] Kruse H, Mukherjee A. Preprocessing Text to Improve Compression Ratios. In Storer JA, Cohn M, editors, Proceedings of the 1998 IEEE Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, 1998; 556.

[25] Smirnov MA. Techniques to enhance compression of texts on natural languages for lossless data compression methods. Proceedings of V session of post-graduate students and young scientists of St. Petersburg, State University of Aerospace Instrumentation, Saint-Petersburg, Russia, 2002.

[26] Sun W, Mukherjee A, Zhang N. A Dictionary-based Multi-Corpora Text Compression System. In Storer JA, Cohn M, editors, Proceedings of the 2003 IEEE Data Compression Conference, IEEE Computer Society Press, Los Alamitos, California, 2003; 448.

[27] M. Burrows and D.J. Wheeler. “A Block-Sorting Lossless Data Compression Algorithm." Technical Report, Digital Equipment Corporation, Palo Alto, California, 1994.

[28] Andersson, A. and Nilsson, S. A New Efficient Radix Sort. In 35th Symposium on Foundations of Computer Science, 1994, pp. 714-721

[29] Andersson, A. and Nilsson, S.. Implementing Radixsort. The ACM Journal of Experimental Algorithmics. Volume 3, Article 7, 1998.

[30] Fenwick, P. Improvements to the Block Sorting Text Compression Algorithm. Technical Report 120, University of Auckland, New Zealand, Department of Computer Science. 1995.

[31] Kurtz, S. Reducing the Space Requirement of Suffix Trees. Report 98-03, Technische Fakultat, Universitat Bielefeld. 1998.

[32] Kurtz, S. and Balkenhol, B. Space Efficient Linear Time Computation of the Burrows and Wheeler-Transformation. ALTHÖFER, I. ET AL. Eds. Numbers, Information and complexity, Festschrift in honour of Rudolf Ahlswede's 60th Birthday, 1999, pp. 375-384.

[33] Sadakane, K.. Improvements of Speed and Performance of Data Compression Based on Dictionary and Context Similarity. Master's thesis, Department of Information Science, Faculty of Science, University of Tokyo, Japan. 1997

[34] Sadakane, K. Unifying Text Search And Compression -Suffix Sorting, Block Sorting and Suffix Arrays. PhD thesis, University of Tokyo, Japan. 2000.

[35] Larsson, N.J. Structures of String Matching and Data Compression. PhD thesis, Department of Computer Science, Lund University, Sweden. 1999.

[36] Seward, J. On the performance of BWT sorting algorithms. In Proceedings of the IEEE Data Compression Conference 2000, Snowbird, Utah, STORER, J.A. AND COHN, M. Eds. 2000, pp. 173-182.

[37] Itoh, H. and Tanaka, H. 1999. An Efficient Method for Construction of Suffix Arrays. IPSJ Transactions on Databases, Abstract Vol.41, No. SIG01 – 004.

[38] Kao, T.-H. Improving Suffix-Array Construction Algorithms with Applications. Master's thesis, Department of Computer Science, Gunma University, Japan. 2001.

[39] Manzini, G. and Ferragina, P. Engineering a Lightweight Suffix Array Construction Algorithm. Lecture Notes in Computer Science, Springer Verlag, Volume 2461, 2002, pp. 698-710.

[40] Schindler, M. A Fast Block-sorting Algorithm for lossless Data Compression. In Proceedings of the IEEE Data Compression Conference 1997, Snowbird, Utah, STORER, J.A. AND COHN, M. Eds. 1997, pp. 469.

[41] Balkenhol, B., Kurtz, S. and Shtarkov, Y.M. Modifications of the Burrows and Wheeler Data Compression Algorithm. In Proceedings of the IEEE Data Compression Conference 1999, Snowbird, Utah, STORER, J.A. AND COHN, M. Eds. 1999, pp. 188-197.

[42] Arnavut, Z. and Magliveras, S.S. Block Sorting and Compression. In Proceedings of the IEEE Data Compression Conference 1997, Snowbird, Utah, STORER, J.A. AND COHN, M. Eds. 1997, pp. 181-190.

[43] Deorowicz, S. Improvements to Burrows-Wheeler Compression Algorithm. Software - Practice and Experience 2000, 30(13), 2000, pp. 1465-1483.

[44] Fenwick, P. Improvements to the Block Sorting Text Compression Algorithm. Technical Report 120, University of Auckland, New Zealand, Department of Computer Science. 1995.

[45] Fenwick, P. Block Sorting Text Compression - Final Report. Technical Report 130, University of Auckland, New Zealand, Department of Computer Science. 1996.

[46] Balkenhol, B., Kurtz, S. And Shtarkov, Y.M.. Modifications of the Burrows and Wheeler Data Compression Algorithm. In Proceedings of the IEEE Data Compression Conference 1999, Snowbird, Utah, STORER, J.A. AND COHN, M. Eds. 1999, pp. 188-197.

[47] Amir A., Benson G., ‘Efficient two-dimensional compressed matching,’ Proc. 2nd IEEE Data Compression Conference, 1992, pp. 279-288.

[48] C. Shannon, “A Mathematical Theory of Communication,” Bell System. Tech. J. 27, 379-423 (July 1948).

[49] N. Abramson, Information Theory and Coding, McGraw-Hill Book Co., Inc., New York, 1963.

[50] F. Jelinek, Probabilistic Information Theory, McGraw-Hill Book Co., Inc., New York, 1968.

[51] J. Schalkwijk, “An Algorithm for Source Coding,” IEEE Trans. Info. Theory IT-18, 395 (1972).

[52] T. M. Cover, “Enumerative Source Coding,” IEEE Trans. Info. Theory IT-19, 73 (1973).

[53] J. J. Rissanen, “Generalized Kraft Inequality and Arithmetic Coding,” IBMJ. Res. Develop. 20, 198-203 (1976).

[54] R. Pasco, “Source Coding Algorithms for Fast Data Compression,” Ph.D. Thesis, Department of Electrical Engineering, Stanford University, CA, 1976.

[55] Glen G. Langdon, Jr. and Jorma Rissanen, “Compression of Black-White Images with Arithmetic Coding,’’ IEEE Trans. Commun. COM-29,858-867 (June 1981).

[56] Frank Rubin, “Arithmetic Stream Coding Using Fixed Precision Registers,” IEEE Trans. Info. Theory lT-25,672-675 (November 1979).

[57] C. B. Jones, “An Efficient Coding System for Long Source Sequences,” IEEE Trans. Info. TheoryIT-27,280-291 (May 1981).

[58] G. N. N. Martin, “Range Encoding: an Algorithm for Removing Redundancy from a Digitized Message,’’ presented at the Video and Data Recording Conference, Southampton, England, July 1979.

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

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

[61] A. Moffat. Implementing the PPM Data Compression Scheme. IEEE Transactions on Communications, COM-38:1917 – 1921, November 1990

[62] Paul Glor Howard. “The Design and Analysis of Efficient Lossless Data Compression Systems”, PhD Thesis, Department of Computer Science, Brown University 1993.

[63] Shkarin, D. PPM: one step to practicality. In Proceedings of Data Compression Conference 2002, pp 202 – 211

[64] Bloom, C. Solving the problems of context modeling. http://www.cbloom.com/papers/ppmz.zip, 1998

[65] Igor Pavlov, “7z format”, http://www.7-zip.org/7z.html.

[66] Karp, R.M., Rabin, M.O.. Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. 31(2):249–260, 1987.

[67] J.H. Morris Jr. and V.R. Pratt, ‘A linear pattern-matching algorithm’, Report 40, University of California, Berkeley, 1970.

[68] D.M. Sunday, ‘A very fast substring search algorithm’, Comm. ACM, 33, 132–142 (1990).

[69] P. D. Michailidis; K. G. Margaritis, “On-line string matching algorithms: survey and experimental results”, International Journal of Computer Mathematics, 1029-0265, Volume 76, Issue 4, 2001, Pages 411 – 434.

[70] A. Moffat, "Word–based Text Compression," Software – Practice and Experience, pp. 185–198, 1989.

[71] N. Horspool and G. Cormack, "Constructing Word–Based Text Compression Algorithms," Proceedings of the IEEE Data Compression Conference 1992, Snowbird, pp. 62–71.

[72] W. Teahan and J. Cleary, "The Entropy of English using PPM–Based Models," Proceedings of the IEEE Data Compression Conference 1996, Snowbird, pp. 53–62.

[73] W. Teahan and J. Cleary, "Models of English Text," Proceedings of the IEEE Data Compression Conference 1997, Snowbird, pp. 12–21.

[74] W. Teahan, "Modelling English text," Ph.D. dissertation, Department of Computer Science, University of Waikato, New Zealand, 1998.

[75] B. Chapin and S. Tate, "Preprocessing Text to Improve Compression Ratios," Proceedings of the IEEE Data Compression Conference 1998, Snowbird, p. 532.

[76] B. Chapin, "Higher Compression from the Burrows–Wheeler Transform with new Algorithms for the List Update Problem," Ph.D. dissertation, Department of Computer Science, University of North Texas, 2001.

[77] B. Balkenhol and Y. Shtarkov, "One attempt of a compression algorithm using the BWT," SFB343: Discrete Structures in Mathematics, Falculty of Mathematics, University of Bielefeld, Germany, 1999.

[78] H. Kruse and A. Mukherjee, "Improving Text Compression Ratios with the Burrows–Wheeler Transform," Proceedings of the IEEE Data Compression Conference 1999, Snowbird, p. 536.

[79] S. Grabowski, "Text Preprocessing for Burrows–Wheeler Block–Sorting Compression," VII Konferencja Sieci i Systemy Informatyczne – Teoria, Projekty, Wdrozenia, Lodz, Poland, 1999.

[80] R. Franceschini, H. Kruse, N. Zhang, R. Iqbal and A. Mukherjee, "Lossless, Reversible Transformations that Improve Text Compression Ratios," Preprint of the M5 Lab, University of Central Florida, 2000.

[81] F. Awan. N. Zhang, N. Motgi, R. Iqbal and A. Mukherjee, "LIPT: A Reversible Lossless Text Transform to Improve Compression Performance," Proceedings of the IEEE Data Compression Conference 2001, Snowbird, pp. 481–210.

[82] R. Isal and A. Moffat, "Parsing Strategies for BWT Compression," Proceedings of the IEEE Data Compression Conference 2001, Snowbird, pp. 429 - 438.

[83] R. Isal, A. Moffat and A. Ngai, "Enhanced Word–Based Block–Sorting Text Compression," Proceedings of the twenty–fifth Australasian conference on Computer science, pp. 129 - 138, 2002.

[84] W. Teahan and D. Harper, "Combining PPM Models Using a Text Mining Approach," Proceedings of the IEEE Data Compression Conference 2001, Snowbird, pp. 153–162.

[85] W. F. Frakes, R. B. Yates Ed. Information Retrieval, Data Structures & Algorithms. Prentics Hall 1992.

[86] G. H. Gonnet, R. Beaza-Yates. Handbook of Algorithms and Data Structures. Addison-Wesley Publishing, 1991.

<u>APPENDIX A</u> – <u>Coding of CBTC-B</u>

```
// Source code for Method CBTC-B
// Program for Compression

// Including header files

#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<ctype.h>
#include<time.h>
#include<dos.h>
#include<string.h>
#include"bitio.h"
#include"bitio.c"

// Declaration of functions

void update_str(char,char,char,char);
void readstatistics();
void sort_char_arrays();
void remove_zero_count();
void assign_filename();
int ascii4char(char c1,char c2,char c3,char c4);
int ascii3char(char c1,char c2,char c3);
int search4char(char c1,char c2,char c3,char c4);
int search3char(char c1,char c2,char c3);
int search2char(char c1,char c2);
void search1char(char c1);

// Declaratin of variables

long single_str[256] = {0};
long double_str[26][26] = {0};
long tripple_str[26][26][26] = {0};
long quad_str[26][26][26][26] = {0};
long single_count = 0;
long double_count = 0;
long tripple_count = 0;
long quad_count = 0;
long char_count = 0;
char single_char[256][2];
char double_char[500][3];
char tripple_char[5000][4];
char quad_char[50000][5];

long int single_char_count[256];
long int double_char_count[500];
long int tripple_char_count[5000];
long int quad_char_count[50000];


char sfilename[25];
char singlefilename1[50];
char doublefilename1[50];
char tripplefilename1[50];
char quadfilename1[50];
char singlefilename2[50];
char doublefilename2[50];
char tripplefilename2[50];
char quadfilename2[50];
char tfilename[25];

long int double_match = 0;
long int tripple_match = 0;
long int quad_match = 0;

FILE *fptr,*ptr;
BIT_FILE *fout;
char str[50];
long i,j,k,l;
int len;
int iflower = 1;

void main()
{
      char ch1,ch2,ch3,ch4;
      printf("Enter source file name : ");
      scanf("%s",sfilename);
// Assigning name for target file

      assign_filename();

// Finding the size of the source file

      fptr = fopen(sfilename,"rb");
      long lenoffile;
      fseek(fptr,0L,2);
      lenoffile = ftell(fptr);
      fclose(fptr);

// Creating the dictionaries for 4,3 and 2 characters group
      fptr = fopen(sfilename,"rb");
      ch1 = fgetc(fptr);
      ch2 = fgetc(fptr);
      ch3 = fgetc(fptr);
      ch4 = fgetc(fptr);


      if(ch1>=97 && ch1<=97+25)
            single_str[ch1-97]++;
      if(ch2>=97 && ch2<=97+25)
            single_str[ch2-97]++;
      if(ch3>=97 && ch3<=97+25)
            single_str[ch3-97]++;
      if(ch4>=97 && ch4<=97+25)
            single_str[ch4-97]++;

      if((ch1>=97 && ch1<=97+25) && (ch2>=97 && ch2<=97+25))
            double_str[ch1-97][ch2-97]++;
      if((ch2>=97 && ch2<=97+25) && (ch3>=97 && ch3<=97+25))
```

```
            double_str[ch2-97][ch3-97]++;
      if((ch3>=97 && ch3<=97+25) && (ch4>=97 && ch4<=97+25))
            double_str[ch3-97][ch4-97]++;

      if((ch1>=97 && ch1<=97+25) && (ch2>=97 && ch2<=97+25) && (ch3>=97
&& ch3<=97+25))
            tripple_str[ch1-97][ch2-97][ch3-97]++;
      if((ch2>=97 && ch2<=97+25) && (ch3>=97 && ch3<=97+25) && (ch4>=97
&& ch4<=97+25))
            tripple_str[ch2-97][ch3-97][ch4-97]++;

      if((ch1>=97 && ch1<=97+25) && (ch2>=97 && ch2<=97+25) && (ch3>=97
&& ch3<=97+25) && (ch4>=97 && ch4<=97+25))
            quad_str[ch1-97][ch2-97][ch3-97][ch4-97]++;

      for(i=0;i<lenoffile-4;i++)
      {
            ch1 = ch2;
            ch2 = ch3;
            ch3 = ch4;
            ch4 = fgetc(fptr);
            update_str(ch1,ch2,ch3,ch4);
      }
      fclose(fptr);
// Removing 4,3 and 2-character groups having zero count
// and writing the non-zero count groups in the file
      remove_zero_count();

// Reading the dictionaries from the file
      readstatistics();

// Sorting the chracter groups as per their counts

      sort_char_arrays();

// Compression begins here

      fptr = fopen(sfilename,"rb");
      fout = OpenOutputBitFile(tfilename);
      int tlen=0;

// Reading 4-character group and searching in the dictionaries
      ch1 = fgetc(fptr);
      ch2 = fgetc(fptr);
      ch3 = fgetc(fptr);
      ch4 = fgetc(fptr);
      for(i=0;i<lenoffile-4;i++)
      {
// Search 4-characters group
            if(search4char(ch1,ch2,ch3,ch4))
            {
                  ch1 = fgetc(fptr);
                  ch2 = fgetc(fptr);
                  ch3 = fgetc(fptr);
                  ch4 = fgetc(fptr);
                  i+=3;
                  quad_match++;
                  continue;
            }
// Search 3-characters group
            if(search3char(ch1,ch2,ch3))
            {
                  ch1 = ch4;
                  ch2 = fgetc(fptr);
                  ch3 = fgetc(fptr);
                  ch4 = fgetc(fptr);
                  i+=2;
                  tripple_match++;
                  continue;
            }
// Search 2-characters group
            if(search2char(ch1,ch2))
            {
                  ch1 = ch3;
                  ch2 = ch4;
                  ch3 = fgetc(fptr);
                  ch4 = fgetc(fptr);
                  i+=1;
                  double_match++;
                  continue;
            }
// Search single character
            search1char(ch1);
            ch1 = ch2;
            ch2 = ch3;
            ch3 = ch4;
            ch4 = fgetc(fptr);
      }

      fclose(fptr);
      CloseOutputBitFile(fout);
}


int search4char(char c1,char c2,char c3,char c4)
{
      char tstr[5];
      long ti;
      tstr[0] = c1; tstr[1] = c2; tstr[2] = c3; tstr[3] = c4; tstr[4] =
'\0';
      if(ascii4char(c1,c2,c3,c4))
      {
            if(quad_str[c1-97][c2-97][c3-97][c4-97]==0)
                  return 0;
      }
      unsigned long int indexvalue;
      for(ti=0; ti<quad_count; ti++)
      {
            if(strcmp(quad_char[ti],tstr) == 0)
            {
                  indexvalue = ti + 49152;
                  OutputBits(fout,indexvalue,16);
                  return 1;
            }
      }
      return 0;

}

int search3char(char c1,char c2,char c3)
{
      char tstr[4];
      long ti;
      tstr[0] = c1; tstr[1] = c2; tstr[2] = c3; tstr[3] = '\0';
      if(ascii3char(c1,c2,c3))
      {
            if(tripple_str[c1-97][c2-97][c3-97]==0)
                  return 0;
      }

      unsigned long int indexvalue;
      for(ti=0; ti<tripple_count; ti++)
      {
            if(strcmp(tripple_char[ti],tstr) == 0)
            {
                  indexvalue = ti + 40960;
                  OutputBits(fout,indexvalue,16);
                  return 1;
            }
      }
      return 0;

}
int search2char(char c1,char c2)
{
      char tstr[3];
      long ti;
      tstr[0] = c1; tstr[1] = c2; tstr[2] = '\0';
      unsigned long int indexvalue;
      for(ti=0; ti<32; ti++)
      {
            if(strcmp(double_char[ti],tstr) == 0)
            {
                  indexvalue = ti + 128;
                  OutputBits(fout,indexvalue,8);
                  return 1;
            }
      }
      return 0;
}

void search1char(char c1)
{
      unsigned long int indexvalue = c1;
      OutputBits(fout,indexvalue,8);
}

// Updating the counts of character groups
void update_str(char ch1,char ch2,char ch3,char ch4)
{
      if(ch4>=97 && ch4<=97+25)
```

```
            single_str[ch4-97]++;
      if((ch3>=97 && ch3<=97+25) && (ch4>=97 && ch4<=97+25))
            double_str[ch3-97][ch4-97]++;
      if((ch2>=97 && ch2<=97+25) && (ch3>=97 && ch3<=97+25) && (ch4>=97
&& ch4<=97+25))
            tripple_str[ch2-97][ch3-97][ch4-97]++;
      if((ch1>=97 && ch1<=97+25) && (ch2>=97 && ch2<=97+25) && (ch3>=97
&& ch3<=97+25) && (ch4>=97 && ch4<=97+25))
            quad_str[ch1-97][ch2-97][ch3-97][ch4-97]++;
}

// Reading the dictionaries of character groups

void readstatistics()
{
      long int i=0;
      ptr = fopen(doublefilename1,"r");
      i=0;
      fscanf(ptr,"%s%ld", double_char[i],&double_char_count[i]);
      while(!feof(ptr))
            fscanf(ptr,"%s%ld",
double_char[i],&double_char_count[++i]);
      fclose(ptr);
      ptr = fopen(tripplefilename1,"r");
      i=0;
      fscanf(ptr,"%s%ld", tripple_char[i],&tripple_char_count[i]);
      while(!feof(ptr))
            fscanf(ptr,"%s%ld",
tripple_char[i],&tripple_char_count[++i]);
      fclose(ptr);
      ptr = fopen(quadfilename1,"r");
      i=0;
      fscanf(ptr,"%s%ld", quad_char[i],&quad_char_count[i]);
      while(!feof(ptr))
            fscanf(ptr,"%s%ld", quad_char[i],&quad_char_count[++i]);
      fclose(ptr);
}

// Sorting the character groups as per their counts

void sort_char_arrays()
{
      long int i,j;
      char temp_double_str[3];
      char temp_tripple_str[4];
      char temp_quad_str[5];
      long temp_double;
      long temp_tripple;
      long temp_quad;


      for(i=0;i<double_count-1;i++)
      {
            for(j=i+1;j<double_count;j++)
            {
                  if(double_char_count[i] < double_char_count[j])
                  {
                        strcpy(temp_double_str,double_char[i]);
                        strcpy(double_char[i],double_char[j]);
                        strcpy(double_char[j],temp_double_str);
                        temp_double = double_char_count[i];
                        double_char_count[i] = double_char_count[j];
                        double_char_count[j] = temp_double;
                  }
            }
      }

      ptr = fopen(doublefilename2,"w");
      for(i=0;i<32;i++)
            fprintf(ptr,"%s",double_char[i]);
      fclose(ptr);

      for(i=0;i<tripple_count-1;i++)
      {
            for(j=i+1;j<tripple_count;j++)
            {
                  if(tripple_char_count[i] < tripple_char_count[j])
                  {
                        strcpy(temp_tripple_str,tripple_char[i]);
                        strcpy(tripple_char[i],tripple_char[j]);
                        strcpy(tripple_char[j],temp_tripple_str);
                        temp_tripple = tripple_char_count[i];
                        tripple_char_count[i] = tripple_char_count[j];
                        tripple_char_count[j] = temp_tripple;
                  }
            }
      }

      ptr = fopen(tripplefilename2,"w");
      for(i=0;i<tripple_count;i++)
            fprintf(ptr,"%s",tripple_char[i]);
      fclose(ptr);

      for(i=0;i<quad_count-1;i++)
      {
            for(j=i+1;j<quad_count;j++)
            {
                  if(quad_char_count[i] < quad_char_count[j])
                  {
                        strcpy(temp_quad_str,quad_char[i]);
                        strcpy(quad_char[i],quad_char[j]);
                        strcpy(quad_char[j],temp_quad_str);
                        temp_quad = quad_char_count[i];
                        quad_char_count[i] = quad_char_count[j];
                        quad_char_count[j] = temp_quad;
                  }
            }
      }

      ptr = fopen(quadfilename2,"w");
      for(i=0;i<quad_count;i++)
            fprintf(ptr,"%s",quad_char[i]);
      fclose(ptr);
}
```

```
// Removing the groups having zero count

void remove_zero_count()
{
      ptr = fopen(doublefilename1,"w");
      for(i=0;i<26;i++)
            for(j=0;j<26;j++)
                  if(double_str[i][j] !=0 )
                        fprintf(ptr,"%c%c
%ld\n",i+97,j+97,double_str[i][j]);
      fclose(ptr);

      ptr = fopen(tripplefilename1,"w");
      for(i=0;i<26;i++)
            for(j=0;j<26;j++)
                  for(k=0;k<26;k++)
                        if(tripple_str[i][j][k] > 3 )
                              fprintf(ptr,"%c%c%c
%ld\n",i+97,j+97,k+97,tripple_str[i][j][k]);
      fclose(ptr);

      ptr = fopen(quadfilename1,"w");
      for(i=0;i<26;i++)
            for(j=0;j<26;j++)
                  for(k=0;k<26;k++)
                        for(l=0;l<26;l++)
                              if(quad_str[i][j][k][l] > 2 )
                                    fprintf(ptr,"%c%c%c%c
%ld\n",i+97,j+97,k+97,l+97,quad_str[i][j][k][l]);
      fclose(ptr);
}

void assign_filename()
{
      len = strlen(sfilename);

      for(i=0;i<len;i++)
      {
            if(sfilename[i]=='.')
                  break;
            singlefilename1[i] = sfilename[i];
      }
      if(i>3)
            singlefilename1[4] ='\0';
      else
            singlefilename1[i] ='\0';
      strcpy(tfilename,singlefilename1);
      strcat(tfilename,"cbtcb.usb");

      strcpy(doublefilename1,singlefilename1);
      strcpy(tripplefilename1,singlefilename1);
      strcpy(quadfilename1,singlefilename1);

      strcpy(doublefilename2,singlefilename1);
      strcpy(tripplefilename2,singlefilename1);
      strcpy(quadfilename2,singlefilename1);

      strcat(singlefilename1,"_single.dat");
      strcat(doublefilename1,"_double.dat");
      strcat(tripplefilename1,"_tripple.dat");
      strcat(quadfilename1,"_quad.dat");

      strcat(singlefilename2,"_single.txt");
      strcat(doublefilename2,"cbtcbd.txt");
      strcat(tripplefilename2," cbtcbt.txt");
      strcat(quadfilename2," cbtcbq.txt");
}

int ascii4char(char c1,char c2,char c3,char c4)
{
      if((c1>='a' && c1<='z') && (c2>='a' && c2<='z') && (c3>='a' &&
c3<='z') && (c4>='a' && c4<='z'))
            return 1;
      else
            return 0;
}

int ascii3char(char c1,char c2,char c3)
{
      if((c1>='a' && c1<='z') && (c2>='a' && c2<='z') && (c3>='a' &&
c3<='z'))
            return 1;
      else
            return 0;
}
```

```
// Program for Decompression

// Including header files

#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<ctype.h>
#include<time.h>
#include<dos.h>
#include<string.h>
#include"bitio.h"
#include"bitio.c"


long double_count = 0;
long tripple_count = 0;
long quad_count = 0;

void readstatistics();
void sort_char_arrays();
void assign_filename();



char double_char[500][3];
char tripple_char[5000][4];
char quad_char[50000][5];

long int double_char_count;
long int tripple_char_count;
long int quad_char_count;

char sfilename[25];
char doublefilename2[50];
char tripplefilename2[50];
char quadfilename2[50];
char tfilename[25];
char singlefilename1[25];

FILE *ptr,*fptr;
BIT_FILE *fin;
long i,j,k,l;
int len;
int iflower = 0;
int readbits;
unsigned long read;
unsigned long anotherread;
void main()
{
      printf("Enter file name to decompress : ");
      scanf("%s",sfilename);
// Assigning name for target file

      assign_filename();
```

```
// Reading the character groups dictionaries
      readstatistics();

// Decompression begins here
      ptr = fopen(tfilename,"wb");
      fptr = fopen(sfilename,"rb");

      readbits = 8;
      int r;
      int ch;
      while(1)
      {
            read = getc(fptr);
            if(read == EOF)
                  break;
// If normal ascii character then store as it is in the decompressed
file
            if(read < 128)
            {
                  ch = (int)read;
                  if(ch == 10)
                        fprintf(ptr,"\n");
                  else
                        fprintf(ptr,"%c",ch);
                  continue;
            }
// if code is from 2-character group store those 2-characters
// from 2-character group dictionary

            if(read >=128 && read <=159)
            {
                  read -= 128;
                  ch = double_char[read][0];
                  fprintf(ptr,"%c",ch);
                  ch = double_char[read][1];
                  fprintf(ptr,"%c",ch);
                  continue;
            }
// if code is from 3-character group store those 2-characters
// from 3-character group dictionary

            anotherread = getc(fptr);
            read = read << readbits;
            read = read | anotherread;
            if(read >= 40960 && read <= 49151)
            {
                  read -= 40960;
                  ch = tripple_char[read][0];
                  fprintf(ptr,"%c",ch);
                  ch = tripple_char[read][1];
                  fprintf(ptr,"%c",ch);
                  ch = tripple_char[read][2];
                  fprintf(ptr,"%c",ch);
                  continue;
            }
// if code is from 4-character group store those 2-characters
// from 4-character group dictionary
```

```
            read -= 49152;
            ch = quad_char[read][0];
            fprintf(ptr,"%c",ch);
            ch = quad_char[read][1];
            fprintf(ptr,"%c",ch);
            ch = quad_char[read][2];
            fprintf(ptr,"%c",ch);
            ch = quad_char[read][3];
            fprintf(ptr,"%c",ch);

      }
      fclose(fptr);
      fclose(ptr);

}

void readstatistics()
{
      long int i=0;
      printf("\nDOuble dictionary :\n");
      ptr = fopen(doublefilename2,"r");
      i=0;
      fscanf(ptr,"%c%c", &double_char[i][0],&double_char[i][1]);
      double_char[i][2]='\0';
      while(!feof(ptr))
      {
            i++;
            fscanf(ptr,"%c%c",&double_char[i][0],&double_char[i][1]);
            double_char[i][2]='\0';
      }
      fclose(ptr);
      double_count = i;
      printf("\nTripple DIctionary : \n");
      ptr = fopen(tripplefilename2,"r");
      i=0;
      fscanf(ptr,"%c%c%c",
&tripple_char[i][0],&tripple_char[i][1],&tripple_char[i][2]);
      tripple_char[i][3]='\0';
      while(!feof(ptr))
      {
            i++;
            fscanf(ptr,"%c%c%c",
&tripple_char[i][0],&tripple_char[i][1],&tripple_char[i][2]);
            tripple_char[i][3]='\0';
      }
      fclose(ptr);
      tripple_count = i;
      printf("\nQuad DIctionary : \n");
      ptr = fopen(quadfilename2,"r");
      i=0;
      fscanf(ptr,"%c%c%c%c",
&quad_char[i][0],&quad_char[i][1],&quad_char[i][2],&quad_char[i][3]);
      quad_char[i][4]='\0';
      while(!feof(ptr))
      {
            i++;
            fscanf(ptr,"%c%c%c%c",
&quad_char[i][0],&quad_char[i][1],&quad_char[i][2],&quad_char[i][3]);
            quad_char[i][4]='\0';
      }
      fclose(ptr);
      quad_count = i;
}

void assign_filename()
{
      len = strlen(sfilename);
      char tempfile[50];
      for(i=0;i<len;i++)
      {
            if(sfilename[i]=='.')
                  break;
            singlefilename1[i] = sfilename[i];
      }
      singlefilename1[i] ='\0';
      strcpy(tfilename,singlefilename1);
      strcat(tfilename,".out");



      strcpy(doublefilename2,singlefilename1);
      strcpy(tripplefilename2,singlefilename1);
      strcpy(quadfilename2,singlefilename1);


      strcat(doublefilename2,"d.txt");
      strcat(tripplefilename2,"t.txt");
      strcat(quadfilename2,"q.txt");
}
```

APPENDIX B – Coding of WBTC-A

```
// Source code for Method WBTC-A
// Program for Compression

// Including header files

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
#include"bitio.h"
#include"bitio.c"

// Declaring constants

#define MAXPREFIX 5000
#define MAXSUFFIX 5000
#define MAX 65536
#define DICTCONST 32768
#define PRECONST 56768
#define SUFCONST 60768

// Declaration of functions

void assign_filename();
void sort();
void createprefix();
void createsuffix();
void addword();
void writeseparator();
void writeword();
int ascii(unsigned long);
long int searchstr(char*);
long int searchstrfromdict(char*);
int searchprefixstr(char*);
int searchsuffixstr(char*);

// Declaration of variables

long int found;
char dictionary[MAX][50];
char prefixdictionary[MAXPREFIX][20];
char suffixdictionary[MAXSUFFIX][20];
long int dictionarycount[MAX];
long int prefixcount[MAXPREFIX];
long int suffixcount[MAXSUFFIX];
unsigned int trackdictionary;
unsigned int trackprefix;
unsigned int tracksuffix;
unsigned int tracknonword;
int suffixposition;
int startofonelengthword;
int position;
int pstr;
```

```
char str[100];
char sfilename[50],tfilename[50],wordfilename[50];
char prefixfilename[50],suffixfilename[50];
FILE *fptr;
BIT_FILE *bfin;
BIT_FILE *bfout;

void main()
{
      int i,j,c;
      unsigned long value;
      printf("\nEnter File Name : ");
      scanf("%s",sfilename);
// Assigning name for target file

      assign_filename();

// Finding the size of the source file

      fptr = fopen(sfilename,"rb");
      fseek(fptr,0L,2);
      long lenoffile = ftell(fptr);
      fclose(fptr);
      printf("\nLen of file = %ld",lenoffile);

// Creation of semi-dynamic dictionary begins

      bfin = OpenInputBitFile(sfilename);
      pstr = 0; trackdictionary = 0; trackprefix = 0; tracksuffix = 0;
tracknonword = 0;
      for(c=0;c<lenoffile;c++)
      {
            value = InputBits(bfin,8);
// Forming the words by checking the read characters are ascii or not

            if(ascii(value))
            {
                  str[pstr++]=value;
                  continue;
            }
// Adding the word to the dictionary

            addword();
            if(trackdictionary>=MAX)
            {
                  printf("Dictionary Full ");
                  break;
            }
      }
// Add the last word to the dictionary

      addword();
// Sort the words in the dictionary according to their counts

      sort();
// Check for the first occurrence of the word whose frequency is
// one and store that value in startofonelengthword
// and storing the dictionary in the file.

      fptr = fopen(wordfilename,"w");
      for(i=0;i<trackdictionary;i++)
      {
            if( (dictionarycount[i]==1) || (i == DICTCONST) )
                  break;
            fprintf(fptr,"%s\n",dictionary[i]);
      }
      fclose(fptr);
      startofonelengthword = i;
// Creation of Prefix sub-word dictionary and storing it in the file

      createprefix();
      fptr = fopen(prefixfilename,"w");
      for(i=0;i<trackprefix;i++)
            fprintf(fptr,"%s\n",prefixdictionary[i]);
      fclose(fptr);
// Creation of Suffix sub-word dictionary and storing it in the file

      createsuffix();
      fptr = fopen(suffixfilename,"w");
      for(i=0;i<tracksuffix;i++)
            fprintf(fptr,"%s\n",suffixdictionary[i]);
      fclose(fptr);

// Compression begins here

      pstr = 0;
      bfin = OpenInputBitFile(sfilename);
      bfout = OpenOutputBitFile(tfilename);
      while(1)
      {
            value = InputBits(bfin,8);
            if(value == 0xffff)
                  break;
// Forming the words by checking the read characters are ascii or not

            if(ascii(value))
            {
                  str[pstr++]=value;
                  continue;
            }
// Terminating the word

            str[pstr]='\0';
// If length of word > 2 then processing for compression
            if(pstr > 1)
            {
                  writeword();
                  pstr = 0;
            }
// Else writing the word as it is in the compressed file.

            else if(pstr == 1)
            {
                  OutputBits(bfout,str[0],8);
                  pstr = 0;
            }
// Writing the non-ascii character in the compressed file.

            OutputBits(bfout,value,8);

            pstr = 0;
      }
// Compressing the remaining words or characters left

      str[pstr]='\0';
      if(pstr > 1)
      {
            writeword();
            pstr = 0;
      }
      else if(pstr == 1)
      {
            OutputBits(bfout,str[0],8);
            pstr = 0;
      }

}
void writeword()
{
      int maxmatch;
      int slen;
      int i,j,remainingchar;
      char remstr[100];
      slen = strlen(str);
// Searching the word in the semi-dynamic dictionary

      found = searchstrfromdict(str);
// If found then writing the index value in the compressed file

      if(found)
      {
            found--;
            OutputBits(bfout,found+DICTCONST,16);
      }
      else
// Else search for the prefix and suffix sub-word in the dictionary
      {
            maxmatch = searchprefixstr(str); //Call search prefix if
match write index value in function itself
            if(maxmatch)
            {
                  j=0;
                  for(i=maxmatch;i<slen;i++)
                        remstr[j++] = str[i];
                  remstr[j]='\0';
                  strcpy(str,remstr);
                  slen = strlen(str);
            }
            maxmatch = searchsuffixstr(str);
            if(maxmatch)
            {
                  if(slen>=maxmatch)
                  {
                        remainingchar = slen - maxmatch;
                        for(j=0;j<remainingchar;j++)
                              OutputBits(bfout,str[j],8);
                  }
                  OutputBits(bfout,suffixposition+SUFCONST,16);
            }
// If word not found in the dictionary, then write it as it is in the
// compressed file.

            else
            {
                  for(j=0;j<slen;j++)
                        OutputBits(bfout,str[j],8);
            }
      }
}

// Adding the word to the
void addword()
{
      str[pstr]='\0';
      if(pstr > 1)
      {
            found = searchstr(str);
            if(found)
            {
                  found--;
                  dictionarycount[found]++;
            }
            else
            {
                  strcpy(dictionary[trackdictionary],str);
                  dictionarycount[trackdictionary]++;
                  trackdictionary++;
            }
      }
      pstr = 0;
}

void createprefix()
{
      int i,j,k,maxmatch,position;
      trackprefix = 0;
      char tempprefix[20];
      for(i=trackdictionary-1; i >= startofonelengthword;i--)
      {
            maxmatch = 0;
            for(j=startofonelengthword;j<i;j++)
            {
                  for(k=0;k<strlen(dictionary[i]);k++)
                  {
                        if(dictionary[j][k] != dictionary[i][k])
                              break;
                  }
                  if(k>2 && k>maxmatch)
```

```
                        {
                              maxmatch = k;
                              position = j;
                        }
                  }
                  if(maxmatch)
                  {
                        for(k=0;k<maxmatch;k++)
                              tempprefix[k] = dictionary[j][k];
                        tempprefix[k] ='\0';
                        for(k=0;k<trackprefix;k++)
                              if(strcmp(prefixdictionary[k],tempprefix) == 0)
                                    break;
                        if(k==trackprefix)

            strcpy(prefixdictionary[trackprefix++],tempprefix);
                        if(trackprefix>=MAXPREFIX)
                        {
                              printf("\nPrefix dictionary full ");
                              return;
                        }
                  }
            }
      }
      void createsuffix()
      {
            int i,j,k,l,m,maxmatch,position,lensrc,lendest,minlen;
            tracksuffix = 0;
            char tempsuffix[20];
            for(i=trackdictionary-1; i >= startofonelengthword;i--)
            {
                  maxmatch = 0;
                  for(j=startofonelengthword;j<i;j++)
                  {
                        lensrc = strlen(dictionary[i]);
                        lendest = strlen(dictionary[j]);
                        if(lensrc<lendest)minlen = lensrc; else minlen =
      lendest;
                        for(k=0;k<minlen;k++)
                        {
                              if(dictionary[j][lendest-1] !=
      dictionary[i][lensrc-1])
                                    break;
                              lendest--;lensrc--;
                        }
                        if(k>2 && k>maxmatch)
                        {
                              maxmatch = k;
                              position = j;
                        }
                  }
                  if(maxmatch)
                  {
                        l = strlen(dictionary[position]);
                        m=0;
                        for(k=l-maxmatch;k<l;k++)
                              tempsuffix[m++] = dictionary[position][k];
                        tempsuffix[m]='\0';
                        for(k=0;k<tracksuffix;k++)
                              if(strcmp(suffixdictionary[k],tempsuffix) == 0)
                                    break;
                        if(k==tracksuffix)

            strcpy(suffixdictionary[tracksuffix++],tempsuffix);
                        if(tracksuffix>=MAXSUFFIX)
                        {
                              printf("\nSuffix dictionary full ");
                              return;
                        }
                  }
            }
      }

      // Sorting the words in the dictionary according to their counts
      void sort()
      {
            unsigned long l,m;
            unsigned long t;
            char str[50];
            for(l=0;l<trackdictionary-1;l++)
            {
                  for(m=l+1;m<trackdictionary;m++)
                  {
                        if(dictionarycount[l]<dictionarycount[m])
                        {
                              t = dictionarycount[l];
                              dictionarycount[l] = dictionarycount[m];
                              dictionarycount[m] = t;
                              strcpy(str,dictionary[l]);
                              strcpy(dictionary[l],dictionary[m]);
                              strcpy(dictionary[m],str);
                        }
                  }
            }
      }
      // Function for searching the string in the semi-dynamic dictionary
      // Return 0 if not found, else return position of the word in
      // the dictionary.
      long int searchstr(char *str)
      {
            long int i,track = 0;
            for(i=0;i<trackdictionary;i++)
                  if(strcmp(str,dictionary[i]) == 0)
                        break;
            if(i != trackdictionary)
                  return i+1;
            else
                  return 0;

      }

      long int searchstrfromdict(char *str)
      {
            long int i,track = 0;
            for(i=0;i<startofonelengthword;i++)
                  if(strcmp(str,dictionary[i]) == 0)
                        break;
            if(i != startofonelengthword)
                  return i+1;
            else
                  return 0;

      }
      // Checking the character is ascii or not.
      int ascii(unsigned long value)
      {
            if((value >='a' && value <='z') || (value >='A' && value <='Z'))
                  return 1;
            else
                  return 0;
      }

      // Searching the sub-word from the prefix dictionary
      int searchprefixstr(char *prestr)
      {
            int plen,slen,maxmatch=0,position;
            slen = strlen(prestr);
            int i,j;
            for(i=0;i<trackprefix;i++)
            {
                  plen = strlen(prefixdictionary[i]);
                  if(slen >= plen)
                  {
                        for(j=0;j<plen;j++)
                              if(prestr[j] != prefixdictionary[i][j])
                                    break;
                        if(j==plen)
                        {
                              if(j>maxmatch)
                              {
                                    maxmatch = j;
                                    position = i;
                              }
                        }
                  }
            }
            if(maxmatch)
            {
                  OutputBits(bfout,position+PRECONST,16);
                  return maxmatch;
            }
            return 0;
      }

      // Searching the sub-word from the suffix dictionary
      int searchsuffixstr(char *sufstr)
      {
            int plen,slen,maxmatch=0,sufptr;
            slen = strlen(sufstr);
            int i,j;
            for(i=0;i<tracksuffix;i++)
            {
                  plen = strlen(suffixdictionary[i]);
                  if(slen >= plen)
                  {
                        sufptr = slen-1;
                        for(j=plen-1;j>=0;j--)
                              if(sufstr[j] != suffixdictionary[i][sufptr--])
                                    break;
                        if(j<0)
                        {
                              if(plen>maxmatch)
                              {
                                    maxmatch = plen;
                                    suffixposition = i;
                              }
                        }
                  }
            }
            return maxmatch;
      }

      void assign_filename()
      {
            int len,i;
            len = strlen(sfilename);
            char tempfilename[50];
            for(i=0;i<len;i++)
            {
                  if(sfilename[i]=='.')
                        break;
                  tempfilename[i] = sfilename[i];
            }
            tempfilename[i] ='\0';
            strcpy(tfilename,tempfilename);
            strcpy(prefixfilename,tempfilename);
            strcpy(suffixfilename,tempfilename);
            strcpy(wordfilename,tempfilename);
            strcat(tfilename,"wbtca.usb");
            strcat(prefixfilename,".pre");
            strcat(suffixfilename,".suf");
            strcat(wordfilename,".wrd");
      }
```

```
// Program for Decompression

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
#include"bitio.h"
#include"bitio.c"



#define MAXPREFIX 5000
#define MAXSUFFIX 5000
#define MAX 65536
#define DICTCONST 32768
#define PRECONST 56768
#define SUFCONST 60768


void assign_filename();

char dictionary[MAX][50];
char prefixdictionary[MAXPREFIX][20];
char suffixdictionary[MAXSUFFIX][20];
unsigned int trackdictionary;
unsigned int trackprefix;
unsigned int tracksuffix;

char sfilename[50],tfilename[50],wordfilename[50];
char prefixfilename[50],suffixfilename[50];

FILE *fptr;
BIT_FILE *bfin;
BIT_FILE *bfout;

void main()
{

      int i;
      unsigned long value,anothervalue;
      printf("\nEnter File Name for Decompressing : ");
      scanf("%s",sfilename);
// Assigning name for target file

      assign_filename();
// Finding the size of the compressed file

      fptr = fopen(sfilename,"rb");
      fseek(fptr,0L,2);
      long lenoffile = ftell(fptr);
      fclose(fptr);
      printf("\nLen of file = %ld",lenoffile);

// Reading dictionaries for words and sub-words (Prefix and Suffix)

      trackdictionary = 0;
      trackprefix = 0;
      tracksuffix = 0;
      fptr = fopen(wordfilename,"r");
      fscanf(fptr,"%s",dictionary[trackdictionary++]);
      while (!feof(fptr))
            fscanf(fptr,"%s",dictionary[trackdictionary++]);
      trackdictionary--;
      fclose(fptr);

      fptr = fopen(prefixfilename,"r");
      fscanf(fptr,"%s",prefixdictionary[trackprefix++]);
      while (!feof(fptr))
            fscanf(fptr,"%s",prefixdictionary[trackprefix++]);
      trackprefix--;
      fclose(fptr);

      fptr = fopen(suffixfilename,"r");
      fscanf(fptr,"%s",suffixdictionary[tracksuffix++]);
      while (!feof(fptr))
            fscanf(fptr,"%s",suffixdictionary[tracksuffix++]);
      tracksuffix--;
      fclose(fptr);


// Decompression begins here


      bfin = OpenInputBitFile(sfilename);
      bfout = OpenOutputBitFile(tfilename);
      while(1)
      {
            value = InputBits(bfin,8);
            if(value == 0xffff)
                  break;
// If normal ascii character then write as it is in the
// decompressed file.
            if(value<128)
            {
                  OutputBits(bfout,value,8);
            }
// Else if the read is an index value of suffix, prefix or word
dictionary
// then write the corresponding word from the respective dictionary

            else
            {
                  anothervalue = InputBits(bfin,8);
                  value = value << 8;
                  value = value | anothervalue;
                  if(value>=SUFCONST)
                  {
                        value = value-SUFCONST;
                        for(i=0;i<strlen(suffixdictionary[value]);i++)

      OutputBits(bfout,suffixdictionary[value][i],8);
                  }
                  else if(value>=PRECONST)
                  {
                        value-=PRECONST;
                        for(i=0;i<strlen(prefixdictionary[value]);i++)

      OutputBits(bfout,prefixdictionary[value][i],8);
                  }
                  else if(value>=DICTCONST)
                  {
                        value-=DICTCONST;
                        for(i=0;i<strlen(dictionary[value]);i++)
                              OutputBits(bfout,dictionary[value][i],8);
                  }

            }
      }
      CloseOutputBitFile(bfout);
}


void assign_filename()
{
      int len,i;
      len = strlen(sfilename);
      char tempfilename[50];
      for(i=0;i<len;i++)
      {
            if(sfilename[i]=='.')
                  break;
            tempfilename[i] = sfilename[i];
      }
      tempfilename[i-3] ='\0';

      strcpy(tfilename,tempfilename);
      strcpy(prefixfilename,tempfilename);
      strcpy(suffixfilename,tempfilename);
      strcpy(wordfilename,tempfilename);
      strcat(tfilename,".out");
      strcat(prefixfilename,".pre");
      strcat(suffixfilename,".suf");
      strcat(wordfilename,".wrd");
}
```

APPENDIX C – Coding of WBTC-B

```
// Source code for Method WBTC-B
// Program for Compression

// Including header files

#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<ctype.h>
#include<time.h>
#include<dos.h>
#include<string.h>
#include"bitio.h"
#include"bitio.c"

// Declaring constant

#define MAX 600000

// Declaration of functions

void assign_filename();
long int search_str(char*,int);

// Declaration of variables

char dictionary[MAX];
long int noofwords;
long int track=0;
char sfilename[50],tfilename[50],dictfilename[50];
FILE *fptr,*dptr;
BIT_FILE *fout;
BIT_FILE *fin;
void main()
{
    char ch1;
    char str[80];
    long int found;
    printf("Enter source file name : ");
    scanf("%s",sfilename);
// Assigning name for target file

    assign_filename();
// Finding the size of the source file

    fptr = fopen(sfilename,"rb");
    long lenoffile;
    fseek(fptr,0L,2);
    lenoffile = ftell(fptr);
    fclose(fptr);

// Compression begins here and simultaneously the
// semi-dynamic dictionary is created.
```

```
    fin = OpenInputBitFile(sfilename);
    fout = OpenOutputBitFile(tfilename);
    track = 0;
    dptr = fopen(dictfilename,"w");
    int pstr = 0;
    dictionary[track++] = '#';
    dictionary[track] = '\0';
    noofwords = 0;
    unsigned long value;
    while(1)
    {
        value = InputBits(fin,8);
        if(value == 0xffff)
            break;
        ch1 = value;
// Forming the words by checking the read characters are ascii or not

        if((ch1 >= 'a'  && ch1 <= 'z') || (ch1 >= 'A'  && ch1 <= 'Z') )
        {
            str[pstr] = ch1;
            pstr++;
            continue;
        }
// Terminating the word

        str[pstr]='\0';
// If the length of the word > 1 then compress the word

        if(pstr>1)
        {
// If word is found in the dictionary then store the index
// value of the word in the compressed file
// Else write the word in the dictionary and then store
// the index value in the compressed file.

            found = search_str(str,pstr);
            if(!found)
            {
                fprintf(dptr,"#");
                fprintf(dptr,"%s",str);
                strcat(dictionary,str);
                track = strlen(dictionary);
                dictionary[track++]='#';
                dictionary[track]='\0';
                noofwords++;
                OutputBits(fout,noofwords+32768,16);
            }
            else
                OutputBits(fout,found+32768,16);
        }
        else if(pstr == 1)
            OutputBits(fout,str[0],8);
        OutputBits(fout,ch1,8);
        pstr = 0;
    }
    fclose(fptr);
    CloseOutputBitFile(fout);
    fclose(dptr);
}

// Function for searching the string in the
// semi-dynamic dictionary. return 0 if not found
// else return the index value.

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

void assign_filename()
{
    int len = strlen(sfilename);

    for(int i=0;i<len;i++)
    {
        if(sfilename[i]=='.')
            break;
        tfilename[i] = sfilename[i];
    }
    tfilename[i] ='\0';
    strcpy(dictfilename,tfilename);
    strcat(tfilename,"wbtcb.usb");

    strcat(dictfilename,"wbtcbdict.txt");

}
```

```
// Program for Decompression

// Including header files

#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<ctype.h>
#include<time.h>
#include<dos.h>
#include<string.h>
#include"bitio.h"
#include"bitio.c"

// Declaring constant

#define MAX 600000

// Declaration of functions

void assign_filename();

// Declaration of variables

char dictionary[MAX];
char str[50];
long i,j,k,l;
long sizeofdict;
unsigned long hash;
unsigned long read;
unsigned long anotherread;
char sfilename[50],tfilename[50],dictfilename[50];
FILE *ptr,*fptr;
BIT_FILE *fin;
void main()
{
      printf("Enter file name to decompress : ");
      scanf("%s",sfilename);
// Assigning name for target file.

      assign_filename();
// Finding the size of the compressed file

      fptr = fopen(sfilename,"rb");
      long lenoffile;
      fseek(fptr,0L,2);
      lenoffile = ftell(fptr);
      printf("Length of file = %ld",lenoffile);
      fclose(fptr);

// Reading the dictionary

      ptr = fopen(dictfilename,"r");
      fscanf(ptr,"%s", dictionary);
      sizeofdict = strlen(dictionary);
      fclose(ptr);
```

```
// Decompression begins here

      ptr = fopen(tfilename,"w");
      fptr = fopen(sfilename,"rb");

      int ch;
      for(j=0;j<lenoffile;j++)
      {
            read = getc(fptr);
// If normal ascii character then store as it is in the decompressed
file

            if(read < 128)
            {
                  ch = (int)read;
                  if(ch == 10)
                        putc(ch,ptr);
                  else
                        fprintf(ptr,"%c",ch);
                  continue;
            }
// Else read the value of index position of encoded word
// and retrieve it from the dictionary

            anotherread = getc(fptr);j++;
            read = read << 8;
            read = read | anotherread;
            read -= 32768;
            hash = 0;
            for(i=0;i<sizeofdict;i++)
            {
                  if(dictionary[i]=='#')
                        hash++;
                  if(hash == read)
                  {
                        while(1)
                        {
                              i++;
                              if(i==sizeofdict)
                                    break;
                              ch = dictionary[i];
                              if(ch == '#')
                                    break;
                              fprintf(ptr,"%c",ch);
                        }
                        break;
                  }
            }
      }
      fclose(fptr);
      fclose(ptr);
}


void assign_filename()
{
	int len = strlen(sfilename);

	for(int i=0;i<len;i++)
	{
		if(sfilename[i]=='.')
			break;
		tfilename[i] = sfilename[i];
	}
	tfilename[i] ='\0';
	strcpy(dictfilename,tfilename);
	strcat(tfilename,".out");

	strcat(dictfilename,"dict.txt");

}
```

APPENDIX D – Coding of WBTC-C

```
// Source code for Method WBTC-C
// Program for Compression

// Including header files
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
#include"bitio.h"
#include"bitio.c"

// Declaraing constants

#define MAX 65536
#define MAXSIZE 16447

// Declaration of functions

void assign_filename();
void sort();
void addword();
void writeword();
int ascii(unsigned long);

// Declaration of variables

long int found;
long int searchstr(char*);
char dictionary[MAX][50];
long int dictionarycount[MAX];
unsigned int trackdictionary;
int startofonelengthword;
char substr[20];
int position;
int pstr;
char str[100];
FILE *fptr,*tptr;;
int currentrow,newrow,column;
char sfilename[50],tfilename[50],wordfilename[50];
BIT_FILE *bfin;
BIT_FILE *bfout;

void main()
{

    int i,j,c;  //pointer to string
    long int value;
    printf("\nEnter File Name : ");
    scanf("%s",sfilename);
// Assigning name for target file

    assign_filename();
// Finding the size of the source file
```

```
    fptr = fopen(sfilename,"rb");
    fseek(fptr,0L,2);
    long lenoffile = ftell(fptr);
    fclose(fptr);
    printf("\nLen of file = %ld",lenoffile);

// Creating the semi-dynamic dictionary in first pass

    bfin = OpenInputBitFile(sfilename);
    pstr = 0; trackdictionary = 0;
    while(1)
    {
        value = InputBits(bfin,8);
        if(value == 0xffff)
            break;
// Forming the words by checking the read characters are ascii or not
        if(ascii(value))
        {
            str[pstr++]=value;
            continue;
        }
// Adding the word to the dictionary
        addword();
        if(trackdictionary>=MAX)
        {
            printf("Dictionary Full ");
            break;
        }

    }
// Add the last word to the dictionary

    addword();
// Sort the words in the dictionary according to their counts

    sort();

// Check for the first occurrence of the word whose frequency is
// one and store that value in startofonelengthword

    for(i=0;i<trackdictionary;i++)
        if(dictionarycount[i]==1)
            break;
    startofonelengthword = i;
    if(startofonelengthword < MAXSIZE)
        trackdictionary = startofonelengthword;
    else
        trackdictionary = MAXSIZE;
// Write the dictionary in the file

    fptr = fopen(wordfilename,"w");
    for(i=0;i<trackdictionary;i++)
            fprintf(fptr,"%s\n",dictionary[i]);
    fclose(fptr);
```

```
// Compression begins here

    currentrow = 0;
    pstr = 0;
    bfin = OpenInputBitFile(sfilename);
    bfout = OpenOutputBitFile(tfilename);
    while(1)
    {
        value = InputBits(bfin,8);
        if(value == 0xffff)
            break;
// Forming the words by checking the read characters are ascii or not

        if(ascii(value))
        {
            str[pstr++]=value;
            continue;
        }
// Terminating the word

        str[pstr]='\0';
// If length of word > 2 then processing for compression
        if(pstr > 2)
        {
            writeword();
            pstr = 0;
        }
// Else writing the word as it is in the compressed file.

        else if(pstr == 1)
        {
            OutputBits(bfout,str[0],8);
            pstr = 0;
        }
        else if(pstr == 2)
        {
            OutputBits(bfout,str[0],8);
            OutputBits(bfout,str[1],8);
            pstr = 0;
        }
// Writing the non-ascii character in the compressed file.

        OutputBits(bfout,value,8);
        pstr = 0;
    }
// Compressing the remaining words or characters left
    str[pstr]='\0';
    if(pstr > 1)
    {
        writeword();
        pstr = 0;
    }
    else if(pstr == 1)
    {
        OutputBits(bfout,str[0],8);
        pstr = 0;
    }
    fclose(tptr);
}


void writeword()
{
    int maxmatch;
    int slen;
    int changeincolumn = 15;
    int i,j,remainingchar;
    char remstr[100];
    slen = strlen(str);
// Searching the word in the static dictionary

    found = searchstr(str);
// If found then writing the index value in the compressed file

    if(found)
    {
        found--;
// If word is found in first 63 positions i.e. words repeated in
// each row write the position of the column in the compressed file.

        if(found<63)
        {
            column = found;
            OutputBits(bfout,column+128,8);
        }
// Check whether the word found is in the same row as that of previous
// row

        else
        {
            newrow = (found - 63) / 64;
            column = (found - 63) % 64 ;
// If yes then write the position of the column in the compressed file.

            if(newrow == currentrow)
                OutputBits(bfout,column+128+63,8);
// Else write the escape symbol for change in row i.e. 255 (0xFF)
// and write the new row number followed by the position of the column
in
// the compressed file.

            else
            {
                OutputBits(bfout,255,8);
                OutputBits(bfout,newrow+128,8);
                OutputBits(bfout,column+128,8);
                currentrow = newrow;
            }
        }
    }
// If word not found in the dictionary, then write it as it is in the
// compressed file.

    else
```

```
        for(j=0;j<slen;j++)
            OutputBits(bfout,str[j],8);

}

// Adding the word to the semi-dynamic dictionary
void addword()
{
    str[pstr]='\0';
    if(pstr > 2)
    {
        found = searchstr(str);
        if(found)
        {
            found--;
            dictionarycount[found]++;
        }
        else
        {
            strcpy(dictionary[trackdictionary],str);
            dictionarycount[trackdictionary]++;
            trackdictionary++;

        }
    }
    pstr = 0;
}

// Sorting the words in the dictionary according to their counts
void sort()
{
    unsigned long l,m;
    unsigned long t;
    char str[50];
    for(l=0;l<trackdictionary-1;l++)
    {
        for(m=l+1;m<trackdictionary;m++)
        {
            if(dictionarycount[l]<dictionarycount[m])
            {
                t = dictionarycount[l];
                dictionarycount[l] = dictionarycount[m];
                dictionarycount[m] = t;
                strcpy(str,dictionary[l]);
                strcpy(dictionary[l],dictionary[m]);
                strcpy(dictionary[m],str);
            }
        }
    }
}

// Function for searching the string in the static dictionary
// Return 0 if not found, else return position of the word in
// the dictionary.
long int searchstr(char *str)
{
    long int i,track = 0;
    for(i=0;i<trackdictionary;i++)
        if(strcmp(str,dictionary[i]) == 0)
            break;
    if(i != trackdictionary)
        return i+1;
    else
        return 0;

}

// Checking the character is ascii or not.

int ascii(unsigned long value)
{
    if((value >='a' && value <='z') || (value >='A' && value <='Z'))
        return 1;
    else
        return 0;
}

// Assigning default file names for target file name
void assign_filename()
{
    int len,i;
    len = strlen(sfilename);
    char tempfilename[50];
    for(i=0;i<len;i++)
    {
        if(sfilename[i]=='.')
            break;
        tempfilename[i] = sfilename[i];
    }
    tempfilename[i] ='\0';
    strcpy(tfilename,tempfilename);
    strcpy(wordfilename,tempfilename);
    strcat(tfilename,"_wbtcc.usb");
    strcat(wordfilename,"wbtccwrd.wrd");
}
```

```
// Program for Decompression

// Including header files

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
#include<time.h>
#include<ctype.h>
#include<dos.h>
#include"bitio.h"
#include"bitio.c"

// Declaring constants

#define MAX 65536
#define MAXSIZE 16447

// Declaration of functions

void assign_filename();

// Declaration of variables

long int found;
char dictionary[MAX][50];
long int dictionarycount[MAX];
unsigned int trackdictionary;
int position;
int row,column,index;
long int track;
char sfilename[50],tfilename[50],wordfilename[50];
BIT_FILE *bfin;
BIT_FILE *bfout;
void main()
{
    FILE *fptr;
    int i;  //pointer to string
    long int value;
    printf("\nEnter File Name : ");
    scanf("%s",sfilename);
// Assigning name for target file

    assign_filename();

// Finding the size of the compressed file

    fptr = fopen(sfilename,"rb");
    fseek(fptr,0L,2);
    long lenoffile = ftell(fptr);
    fclose(fptr);

    trackdictionary=0;
    fptr = fopen(wordfilename,"r");
    fscanf(fptr,"%s",dictionary[trackdictionary++]);
    while (!feof(fptr))
        fscanf(fptr,"%s",dictionary[trackdictionary++]);
    trackdictionary--;
    fclose(fptr);

// Compression begins here

    bfin = OpenInputBitFile(sfilename);
    bfout = OpenOutputBitFile(tfilename);
    row = 0;
    track = 0;
    while(track<lenoffile )
    {
        value = InputBits(bfin,8);
        track++;
// If normal ascii character then write as it is in the
// decompressed file.

        if(value<128)
        {
            OutputBits(bfout,value,8);
        }
// If change in row then read the new row number and
// column number and compute the position of the word in the
// dictionary and retrieve the word from the dictionary.

        else
        {
            if(value == 255)
            {
                row = InputBits(bfin,8);
                row-=128;
                value = InputBits(bfin,8);
                value -= 128;
                position = (row * 64) +63+ value ;
                for(i=0;i<strlen(dictionary[position]);i++)
                    OutputBits(bfout,dictionary[position][i],8);
                track+=2;
            }
            else
            {
                value -= 128;
                if(value < 63)
                {
                    for(i=0;i<strlen(dictionary[value]);i++)
                        OutputBits(bfout,dictionary[value][i],8);
                }
                else
                {
                    value -= 63;
                    position = (row * 64) +63+ value ;
                    for(i=0;i<strlen(dictionary[position]);i++)
                        OutputBits(bfout,dictionary[position][i],8);
                }
            }

        }
    }
    CloseInputBitFile(bfin);
    CloseOutputBitFile(bfout);
}


void assign_filename()
{
    int len,i;
    len = strlen(sfilename);
    char tempfilename[50];
    for(i=0;i<len;i++)
    {
        if(sfilename[i]=='.')
            break;
        tempfilename[i] = sfilename[i];
    }
    tempfilename[i-3] ='\0';
    strcpy(tfilename,tempfilename);
    strcpy(wordfilename,tempfilename);
    strcat(tfilename,".out");
    strcat(wordfilename,".wrd");
}
```

APPENDIX E – Coding of WBTC-D

```
// Source code for Method WBTC-D
// Program for Compression

// Including header files

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
#include"bitio.h"
#include"bitio.c"

// Declaring constant

#define MAX 65536

// Declaration of functions
void assign_filename();
int ascii(unsigned long);
long int searchstr(char*);

// Declarations of variables

long int found;
char dictionary[MAX][50];
unsigned int trackdictionary;
char sfilename[50],tfilename[50];
BIT_FILE *bfin;
BIT_FILE *bfout;

void main()
{
    char str[100];
    int pstr,i;
    unsigned long value;
    printf("\nEnter File Name : ");
    scanf("%s",sfilename);
// Assigning name for target file

    assign_filename();

// Compression begins here

    bfin  = OpenInputBitFile(sfilename);
    bfout = OpenOutputBitFile(tfilename);
    pstr = 0; trackdictionary = 0;
    while(1)
    {
        value = InputBits(bfin,8);
        if(value == 0xffff)
            break;
// Forming the words by checking the read characters are ascii or not
```

```
        if(ascii(value))
        {
            str[pstr++]=value;
            continue;
        }
// Terminating the word

        str[pstr]='\0';
// If length of word > 1 then processing for compression

        if(pstr > 1)
        {
// Search the word in the dynamic dictionary

            found = searchstr(str);
// If found then writing the index value in the compressed file

            if(found)
            {
                found--;
                OutputBits(bfout,found+32768,16);
            }
// Else write the word as it is in the compressed file
// and add the word to the dynamic dictionary

            else
            {
                for(i=0;i<pstr;i++)
                    OutputBits(bfout,str[i],8);
                strcpy(dictionary[trackdictionary++],str);
            }
            pstr = 0;
// Writing the non-ascii character in the compressed file.

            OutputBits(bfout,value,8);
        }
// Else writing the character as it is in the compressed file.
        else if(pstr == 1)
        {
            OutputBits(bfout,str[0],8);
            pstr = 0;
            OutputBits(bfout,value,8);
        }
// Writing the non-ascii character in the compressed file.
        else
            OutputBits(bfout,value,8);

    }
// Compressing the remaining words or characters left

    str[pstr]='\0';
    if(pstr > 1)
    {
        found = searchstr(str);
        if(found)
        {
            found--;
            OutputBits(bfout,found+32768,16);
        }
        else
        {
            for(i=0;i<pstr;i++)
                OutputBits(bfout,str[i],8);
            strcpy(dictionary[trackdictionary++],str);
        }

    }
    else if(pstr == 1)
        OutputBits(bfout,str[0],8);
    printf("\nLength of dictionary = %ld",trackdictionary);
}

// Function for searching the string in the dynamic dictionary
// Return 0 if not found, else return position of the word in
// the dictionary.

long int searchstr(char *str)
{
    long int i,track = 0;
    for(i=0;i<trackdictionary;i++)
        if(strcmp(str,dictionary[i]) == 0)
            break;
    if(i!=trackdictionary)
        return i+1;
    else
        return 0;

}

int ascii(unsigned long value)
{
    if((value >='a' && value <='z') || (value >='A' && value <='Z'))
        return 1;
    else
        return 0;
}

void assign_filename()
{
    int len,i;
    len = strlen(sfilename);
    char tempfilename[50];
    for(i=0;i<len;i++)
    {
        if(sfilename[i]=='.')
            break;
        tempfilename[i] = sfilename[i];
    }
    tempfilename[i] ='\0';
    strcpy(tfilename,tempfilename);
    strcat(tfilename,"wbtcd.usb");
}
```

```
// Program for Decompression

// Including header files

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
#include"bitio.h"
#include"bitio.c"

// Declaring constant

#define MAX 65536

// Declaration of functions

void assign_filename();
int ascii(unsigned long);

// Declaration of variables

long int found;
char dictionary[MAX][50];
unsigned int trackdictionary;
char sfilename[50],tfilename[50];
BIT_FILE *bfin;
BIT_FILE *bfout;

void main()
{
    char str[100];
    int pstr,i,len;
    unsigned long value,anothervalue;
    printf("\nEnter File Name : ");
    scanf("%s",sfilename);
// Assigning name for target file

    assign_filename();

// Decompression begins here

    bfin  = OpenInputBitFile(sfilename);
    bfout = OpenOutputBitFile(tfilename);
    pstr = 0; trackdictionary = 0;
    while(1)
    {
        value = InputBits(bfin,8);
        if(value == 0xffff)
            break;
// Forming the words by checking the read characters are ascii or not

        if(ascii(value))
        {
            str[pstr++]=value;
            continue;
        }
        str[pstr]='\0';
// If length of the word is greater than 1, then adding
// the word to the dynamic dictionary

        if(pstr > 1)
        {
            for(i=0;i<pstr;i++)
                OutputBits(bfout,str[i],8);
            strcpy(dictionary[trackdictionary++],str);
            pstr = 0;
        }
        else if(pstr == 1)
        {
            OutputBits(bfout,str[0],8);
            pstr = 0;
        }
// If normal ascii character then write it as it is in the
// compressed file.

        if(value < 128)
            OutputBits(bfout,value,8);
        else
        {
// Read another byte to form 16-bit index value and retrieve
// the word from that index position from the dictionary and
// write it in the decompressed file.

            anothervalue = InputBits(bfin,8);
            value = value << 8;
            value = value | anothervalue;
            value -= 32768;
            len = strlen(dictionary[value]);
            for(i=0;i<len;i++)
                OutputBits(bfout,dictionary[value][i],8);
        }
    }
// Checking for last word any non-ascii symbol is not store at the end
of file.
    str[pstr]='\0';
    if(pstr > 1)
    {
        for(i=0;i<pstr;i++)
            OutputBits(bfout,str[i],8);
    }
    else if(pstr == 1)
        OutputBits(bfout,str[0],8);

}

// Checking the character is ascii or not.

int ascii(unsigned long value)
{
    if((value >='a' && value <='z') || (value >='A' && value <='Z'))
        return 1;
    else
```

```
        return 0;
}

// Assigning default file names for target file name
void assign_filename()
{
    int len,i;
    len = strlen(sfilename);
    char tempfilename[50];
    for(i=0;i<len;i++)
    {
        if(sfilename[i]=='.')
            break;
        tempfilename[i] = sfilename[i];
    }
    tempfilename[i] ='\0';
    strcpy(tfilename,tempfilename);
    strcat(tfilename,".out");
}
```

APPENDIX F – Coding of WBTC-E

```
// Source code for Method WBTC-E
// Program for Compression

// Including header files
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
#include"bitio.h"
#include"bitio.c"

//Declaring constant

#define MAX 200000

//Declaration of functions

void assign_filename();
void writeword();
int ascii(unsigned long);
long int searchstr(char*);

//Declaration of variables

char dictionary[MAX][50];
long count;
long int trackdictionary;
long int found;
int maxlen;
int pstr;
char str[100];
int currentrow,newrow,column;
char sfilename[50],tfilename[50];
BIT_FILE *bfin;
BIT_FILE *bfout;

void main(int argc, char *argv[])
{
      FILE *fptr;
      long int value,i;
      strcpy(sfilename,argv[1]);

// Assigning name for target file

      assign_filename();

// Finding the size of the source file

      fptr = fopen(sfilename,"rb");
      fseek(fptr,0L,2);
      long lenoffile = ftell(fptr);
      fclose(fptr);
```

```
//PROCESS FOR READING THE STATIC DICTIONARY

      fptr = fopen("dictionary.dct","r");
      trackdictionary = 0;
      fscanf(fptr,"%s %ld",dictionary[trackdictionary],&count);
      while(!feof(fptr))
      {
            trackdictionary++;
            fscanf(fptr,"%s %ld",dictionary[trackdictionary],&count);
      }
      fclose(fptr);

//Compression begins here

      currentrow = 0;
      pstr = 0;
      bfin = OpenInputBitFile(sfilename);
      bfout = OpenOutputBitFile(tfilename);
      for(i=0;i<lenoffile;i++)
      {
            value = InputBits(bfin,8);
// Forming the words by checking the read characters are ascii or not

            if(ascii(value))
            {
                  str[pstr++]=value;
                  continue;
            }
// Terminating the word

            str[pstr]='\0';
// If length of word > 2 then processing for compression

            if(pstr > 2)
            {
                  writeword();
                  pstr = 0;
            }
// Else writing the word as it is in the compressed file.

            else if(pstr == 1)
            {
                  OutputBits(bfout,str[0],8);
                  pstr = 0;
            }
            else if(pstr == 2)
            {
                  OutputBits(bfout,str[0],8);
                  OutputBits(bfout,str[1],8);
                  pstr = 0;
            }
// Writing the non-ascii character in the compressed file.

            OutputBits(bfout,value,8);
            pstr = 0;
      }
// Compressing the remaining words or characters left
```

```
      if(pstr > 2)
      {
            writeword();
            pstr = 0;
      }
      else if(pstr == 2)
      {
            OutputBits(bfout,str[0],8);
            OutputBits(bfout,str[1],8);
            pstr = 0;
      }
      else if(pstr == 1)
      {
            OutputBits(bfout,str[0],8);
            pstr = 0;
      }
      CloseOutputBitFile(bfout);
}

void writeword()
{

      int slen;
      int j;
      slen = strlen(str);
// Searching the word in the static dictionary

      found = searchstr(str);
// If found then writing the index value in the compressed file

      if(found)
      {
            found--;
// If word is found in first 32000 positions i.e. words repeated in
// each row write the position of the column in the compressed file.

            if(found<32000)
            {
                  column = found;
                  OutputBits(bfout,column+32768,16);
            }
// Check whether the word found is in the same row as that of previous
// row

            else
            {
                  newrow = (found - 32000) / 511;
                  column = (found - 32000) % 511 ;
// If yes then write the position of the column in the compressed file.

                  if(newrow == currentrow)
                        OutputBits(bfout,column+32768+32000,16);
// Else write the escape symbol for change in row i.e. 255 (0xFF)
// and write the new row number followed by the position of the column
// in the compressed file.
```

```
                  else
                  {
                        OutputBits(bfout,255,8);
                        OutputBits(bfout,newrow,8);
                        OutputBits(bfout,column+32768,16);
                        currentrow = newrow;
                  }
            }
      }
// If word not found in the dictionary, then write it as it is in the
// compressed file.

      else
      {
            for(j=0;j<slen;j++)
                  OutputBits(bfout,str[j],8);
      }
}


// Function for searching the string in the static dictionary
// Return 0 if not found, else return position of the word in
// the dictionary.

long int searchstr(char *str)
{
      long int i,track = 0;
      for(i=0;i<trackdictionary;i++)
            if(strcmp(str,dictionary[i]) == 0)
                  break;
      if(i != trackdictionary)
            return i+1;
      else
            return 0;

}

// Checking the character is ascii or not.

int ascii(unsigned long value)
{
      if((value >='a' && value <='z') || (value >='A' && value <='Z'))
            return 1;
      else
            return 0;
}

// Assigning default file names for target file name

void assign_filename()
{
      int len,i;
      len = strlen(sfilename);
      char tempfilename[50];
      for(i=0;i<len;i++)
      {
            if(sfilename[i]=='.')
```

```
                        break;
                tempfilename[i] = sfilename[i];
        }
        tempfilename[i] ='\0';
        strcpy(tfilename,tempfilename);
        strcat(tfilename,"wbtce.usb");
}
```

```
// Source Code for Method WBTC-E
// Program for Decompression

// Including header files

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
#include"bitio.h"
#include"bitio.c"

// Declaring constant

#define MAX 200000

// Declaration of funciton

void assign_filename();

// Declaration of variables

long int found;
char dictionary[MAX][50];
unsigned int trackdictionary;
int position;
int pstr;
char str[100];
int row,column,index;
long int count;
char sfilename[50],tfilename[50];
BIT_FILE *bfin;
BIT_FILE *bfout;

void main()
{
    FILE *fptr;
    int i;
    long int value,anothervalue,j;
    printf("\nEnter File Name : ");
    scanf("%s",sfilename);
// Assigning name for target file

    assign_filename();

// Finding the size of the compressed file.

    fptr = fopen(sfilename,"rb");
    fseek(fptr,0L,2);
    long lenoffile = ftell(fptr);
    fclose(fptr);

// Reading the static dictionary

    fptr = fopen("dictionary.dct","r");
    trackdictionary = 0;
    fscanf(fptr,"%s %ld",dictionary[trackdictionary],&count);
    while(!feof(fptr))
    {
        trackdictionary++;
        fscanf(fptr,"%s %ld",dictionary[trackdictionary],&count);
    }
    fclose(fptr);

// Decompression begins here

    bfin = OpenInputBitFile(sfilename);
    bfout = OpenOutputBitFile(tfilename);
    row = 0;
    for(j=0;j<lenoffile;j++)
    {
        value = InputBits(bfin,8);
// If normal ascii character then write as it is in the
// decompressed file.
        if(value<128)
            OutputBits(bfout,value,8);
        else
        {
// If change in row then read the new row number and
// column number and compute the position of the word in the
// dictionary and retrieve the word from the dictionary.
            if(value == 255)
            {
                row = InputBits(bfin,8);
                j++;
                value = InputBits(bfin,16);
                j+=2;
                value -= 32768;
                position = (row * 511) +32000+ value ;
                for(i=0;i<strlen(dictionary[position]);i++)
                    OutputBits(bfout,dictionary[position][i],8);
            }
            else
            {
                anothervalue = InputBits(bfin,8);
                j++;
                value = value << 8;
                value = value | anothervalue;
                value -= 32768;
                if(value < 32000)
                    for(i=0;i<strlen(dictionary[value]);i++)
                        OutputBits(bfout,dictionary[value][i],8);
                else
                {
                    value -= 32000;
                    position = (row * 511) + 32000 + value ;
                    for(i=0;i<strlen(dictionary[position]);i++)
                        OutputBits(bfout,dictionary[position][i],8);
                }
            }

        }
    }
    CloseOutputBitFile(bfout);
    CloseInputBitFile(bfin);
}

void assign_filename()
{
    int len,i;
    len = strlen(sfilename);
    char tempfilename[50];
    for(i=0;i<len;i++)
    {
        if(sfilename[i]=='.')
            break;
        tempfilename[i] = sfilename[i];
    }
    tempfilename[i] ='\0';
    strcpy(tfilename,tempfilename);
    strcat(tfilename,".out");
}
```

APPENDIX G – Coding of Searching Phrase

```
//  Source code for Searching
// Program for Searchin using KMP, K-R and B-F Algorithms

// Including header files
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<time.h>
#include<dos.h>

#include "bitio.h"
#include "bitio.c"

// Declaring constants

#define XSIZE 20000 // Knuth Morris Pratt
#define REHASH(a,b,h) (((h-a*d)<<1)+b)   // Karp Rabin

// Declaration of functions

void prekmp(char *x, long int m, int next[]); // Knuth Morris Pratt
int kmp(char*,char*,long int,long int); // Knuth Morris Pratt
int bruteforce(char*,char*,long int,long int);  // Brute Force
int kr(char*,char*,long int,long int); // Karp Rabin
long int readNsourcefile(char*);
long int readNpatternfile(char*);
long int krmatch,kmpmatch,bfmatch;

// Declaration of variables

int count;
char y[5000000];
char x[2000];
long int krnoofcomp,kmpnoofcomp,bfnoofcomp;

void main()
{
      long int xlen,found;
      char sfile[50];
      char pfile[50];
      long int ylen;
      FILE *fptr;
      char che;
      printf("\nEnter source file name :");
      scanf("%s",sfile);
      printf("Enter pattern to be searched (file name) : ");
      scanf("%s",pfile);

// Read the source and the pattern file.

      ylen = readNsourcefile(sfile);
      xlen = readNpatternfile(pfile);

      fptr = fopen("Statistics41.txt","a");
```

```
// Searching using KR Algorithm

      found = kr(x,y,xlen,ylen);
      if(krmatch)
            printf("\n KR Number of occurences  %d ",krmatch);
      else
            printf("\nKR String Not found ");
      printf("\nNo of comparison : %ld",krnoofcomp);

// Searching using KMP Algorithm

      found = kmp(x,y,xlen,ylen);
      if(kmpmatch)
            printf("\n KMP Number of occurences  %d ",kmpmatch);
      else
            printf("\nKMP String Not found ");
      printf("\nNo of comparison : %ld",kmpnoofcomp);

// Searching using BF Algorithm

      found = bruteforce(x,y,xlen,ylen);
      if(bfmatch)
            printf("\n BF Number of occurences  %d ",bfmatch);
      else
            printf("\nBF String Not found ");
      printf("\nNo of comparison : %ld",bfnoofcomp);

// Storing Statistics of searching in file

      printf("\nStore statistics in file  Y/N ? : ");
      che = getche();
      if(che =='y' || che == 'Y')
      {
            printf("\nSearching from Normal file ? (y/n) : ");
            che = getche();

            if(che == 'y' || che == 'Y')
            {
                  fprintf(fptr,"\n\nFILE NAME: %s and pattern to search
: %s",sfile,x);
                  fprintf(fptr,"\nNormal Searching ...........\n");
            }
            else
                  fprintf(fptr,"\nCompressed Searching.........\n");
            if(krmatch)
                  fprintf(fptr,"\nKR Number of occurences  %d
",krmatch);
            else
                  fprintf(fptr,"\nKR Not Found ");
            if(kmpmatch)
                  fprintf(fptr,"\nKMP Number of occurences  %d
",kmpmatch);
            else
                  fprintf(fptr,"\nKMP Not Found ");
            if(bfmatch)
```

```
                  fprintf(fptr,"\nBF Number of occurences  %d
",bfmatch);
            else
                  fprintf(fptr,"\nBF Not Found ");
            fclose(fptr);
      }
}

// Reading source file.
long int readNsourcefile(char *sfile)
{
      unsigned long int size=0;
      BIT_FILE *fin;
      fin = OpenInputBitFile(sfile);
      unsigned long read;
      int readbits = 8;
      while(1)
      {
            read = InputBits(fin,readbits);
            if(read==0xffff)
                  break;
            y[size++] = read;
      }
      return size;
}

// Reading pattern file
long int readNpatternfile(char *sfile)
{
      unsigned long int size=0;
      BIT_FILE *fin;
      fin = OpenInputBitFile(sfile);
      unsigned long read;
      int readbits = 8;
      while(1)
      {
            read = InputBits(fin,readbits);
            if(read==0xffff)
                  break;
            x[size++] = read;
      }
      return size;
}

// KR Algorithm
int kr(char *x,char *y,long int m,long int n)
{
      long int hy,hx,d,i;
      count = 0;
      krnoofcomp = 0;
      krmatch = 0;
      d = 1;
      for(i=1;i<m;i++)
            d = (d<<1);
      hx=hy=0;
      for(i=0;i<m;i++)
      {
            hx = ((hx<<1)+x[i]);
            hy=((hy<<1)+y[i]);
      }
      i=m;
      while (i < n)
      {
            krnoofcomp++;
            if(hy == hx && strncmp(y+i-m,x,m) ==0)
            {
                  krmatch++;
                  krnoofcomp+=2;
            }
            hy = REHASH(y[i-m],y[i],hy);
            i++;
            count++;
      }
      return 0;
}



//KMP Algorithm
int kmp(char *x,char *y,long int m,long int n)
{
      long int i,j;
      kmpmatch=0;
      int next[XSIZE];
      prekmp(x,m,next);
      i=j=0;
      count = 0;
      kmpnoofcomp = 0;
      while(i < n)
      {
            kmpnoofcomp++;
            while(j > -1 && x[j] != y[i])
            {
                  j = next[j];
                  kmpnoofcomp++;
            }
            i++;
            j++;
            if(j >= m)
            {
                  j = next[m];
                  kmpmatch++;
             }
            count++;
      }
      return 0;
}

void prekmp(char *x, long int m, int next[])
{
      long int i,j;
      i=0;j=next[0] = -1;
      while(i < m)
      {
            while(j> -1 && x[i] != x[j]) j = next[j];
            i++;j++;
            if(x[i] == x[j] )
                  next[i] = next[j];
            else
                  next[i] = j;
      }
}

// BF Algorithm
int bruteforce(char *x,char *y,long int m,long int n)
{
      long int i,j;
      i=0;
      count = 0;
      bfnoofcomp = 0;
      bfmatch = 0;
      while (i <= n-m)
      {
            bfnoofcomp++;
            j=0;
            while(j < m && y[i+j] == x[j])
            {
                  j++;
                  bfnoofcomp++;
            }
            if( j >= m)
                  bfmatch++;
                  //return i;
            i++;
            count++;
      }
      return 0;
}
```

```
// Source code for Searching
// Program for Searchin using BM and QS Algorithm

// Including header files

#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<time.h>
#include<dos.h>
#include "bitio.h"
#include "bitio.c"

// Declaring constants

#define XSIZE 50000                 // Boyer Moore
#define ASIZE 50000         // Boyer Moore
#define MAX(x,y) x>y?x:y   // Boyer Moore

// Declaration of functions

void prebc(unsigned long *x, int m, int bc[]); // Boyer Moore
void pregs(unsigned long *x, int m, int gs[]); // Boyer Moore
int bm(unsigned long*,unsigned long*,int,long int);          // Boyer
Moore
int qs(unsigned long *,unsigned long *,int,long int); // Quick Search
long int readNsourcefile(char*);
long int readNpatternfile(char*);

// Declaration of variables

int count;
long int bmmatch,qsmatch;
unsigned long  y[5000000];
unsigned long  x[2000];
long int bmnoofcomp,qsnoofcomp;

void main()
{
      long int xlen,found;
      char sfile[50];
      char pfile[50];
      long int ylen;
      double bmttime,qsttime;
      FILE *fptr;
      char che;

      printf("\nEnter source file name :");
      scanf("%s",sfile);
      printf("Enter pattern to be searched (file name) : ");
      scanf("%s",pfile);

// Read the source and the pattern file.

      ylen = readNsourcefile(sfile);
      xlen = readNpatternfile(pfile);
      printf("\nLength of Y =%d",ylen);
      printf("\nLength of X = %d \n",xlen);

      fptr = fopen("Statbmqs41.txt","a");

// Searching using Boyer-Moore Algorithm
      found = bm(x,y,xlen,ylen);
      if(bmmatch)
            printf("\n BM Number of occurences  %d ",bmmatch);
      else
            printf("\nBM String Not found ");
      printf("\nNo of comparison : %ld",bmnoofcomp);

// Searching using Quick Search Algorithm
      start = clock();
      found = qs(x,y,xlen,ylen);
      end = clock();
      if(qsmatch)
            printf("\n QS Number of occurences  %d ",qsmatch);
      else
            printf("\nQS String Not found ");
      printf("\nNo of comparison : %ld",qsnoofcomp);

// Storing Statistics of searching in file

      printf("\nStore statistics in file  Y/N ? : ");
      che = getche();
      if(che =='y' || che == 'Y')
      {
            printf("\nSearching from Normal file ? (y/n) : ");
            che = getche();

            if(che == 'y' || che == 'Y')
            {
                  fprintf(fptr,"\n\nFILE NAME: %s and pattern to search
: %s",sfile,x);
                  fprintf(fptr,"\nNormal Searching ............\n");
            }
            else
                  fprintf(fptr,"\n\nCompressed Searching.........\n");
            if(bmmatch)
            {
                  fprintf(fptr,"\nBM Number of occurences  %d
",bmmatch);
                  fprintf(fptr,"\nNo of comparison : %ld",bmnoofcomp);
            }
            else
                  fprintf(fptr,"\nBM Not Found ");
            if(qsmatch)
            {
                  fprintf(fptr,"\nQS Number of occurences  %d
",qsmatch);
                  fprintf(fptr,"\nNo of comparison : %ld",qsnoofcomp);
            }
            else
                  fprintf(fptr,"\nQS Not Found ");
            fclose(fptr);
      }
}


// Reading source file.
long int readNsourcefile(char *sfile)
{
      unsigned long int size=0;
      BIT_FILE *fin;
      fin = OpenInputBitFile(sfile);
      unsigned long read;
      int readbits = 8;
      while(1)
      {
            read = InputBits(fin,readbits);
            if(read==0xffff)
                  break;
            y[size++] = read;
      }
      return size;
}

// Reading pattern file
long int readNpatternfile(char *sfile)
{
      unsigned long int size=0;
      BIT_FILE *fin;
      fin = OpenInputBitFile(sfile);
      unsigned long read;
      int readbits = 8;
      while(1)
      {
            read = InputBits(fin,readbits);
            if(read==0xffff)
                  break;
            x[size++] = read;
      }
      return size;
}

// B-M Algorithm

int bm(unsigned long *x,unsigned long *y,int m,long int n)
{
      long int i,j;
      int gs[XSIZE],bc[ASIZE];

      pregs(x,m,gs);
      prebc(x,m,bc);
      bmnoofcomp = 0;
      bmmatch = 0;
      i=0;
      while(i <= n-m)
      {
            j = m-1;
            bmnoofcomp++;
            while (j>=0 && x[j] == y[i+j])
            {
                  j--;
                  bmnoofcomp++;
            }
            if(j < 0)
                  bmmatch++;
            i+=MAX(gs[j+1],bc[y[i+j]]-m+j+1);
      }
      return 0;
}

void prebc(unsigned long *x, int m, int bc[])
{
      int j;
      for(j=0;j < ASIZE; j++)
            bc[j] = m;
      for(j=0;j< m-1; j++ )
            bc[x[j]] = m-j-1;

}

void pregs(unsigned long *x, int m, int gs[])
{
      int i,j,p,f[XSIZE];
      for(i=0;i<=m;i++)
            gs[i] = 0;
      f[m] = j = m+1;
      for(i=m; i> 0; i--)
      {
            while(j <= m && x[i-1] != x[j-1])
            {
                  if(!gs[j]) gs[j] = j-i;
                  j = f[j];
            }
            f[i-1] = --j;
      }
      p = f[0];
      for(j=0;j<=m;j++)
      {
            if(!gs[j]) gs[j] = p;
            if( j == p) p = f[p];
      }
}

// Q-S Algorithm

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