CHAPTER 10

Text Compression

AMAR MUKHERJEE
FAUZIA AWAN[1]

10.1 INTRODUCTION

In recent times, we have seen an unprecedented explosion of textual information through the use of the Internet, digital libraries, and information retrieval systems. The advent of office automation systems and newspaper, journal, and magazine repositories has brought the issue of maintaining archival storage for search and retrieval to the forefront of research. As an example, the TREC [33] database holds around 800 million static pages having 6 trillion bytes of plain text equal to the size of a million books. Text compression is concerned with techniques for representing the digital text data in alternate representations that take less space. It not only helps conserve the storage space for archival and online data, it also helps system performance by requiring less secondary storage (disk or CD Rom) access and improves network transmission bandwidth utilization by reducing the transmission time.

Data compression methods are generally classified as *lossless* or *lossy*. Lossless compression allows the original data to be recovered exactly. Although used primarily for text data, lossless compression algorithms are useful in special classes of images such as medical imaging, fingerprint data, astronomical images, and databases containing mostly vital numerical data, tables, and text information. In contrast, lossy compression schemes allow some deterioration and are generally used for video, audio, and still-image applications. The deterioration of the quality of lossy images is usually not detectable by the human perceptual system, and the compression systems exploit this by a process called "*quantization*" to achieve compression by a factor of 10 to a factor of a couple of hundreds. Many lossy algorithms use lossless methods at the final stage of the

[1] The opinions expressed are those of the author and not those of Microsoft.

encoding, underscoring the importance of lossless methods for both lossy and lossless compression applications. This chapter will be concerned with lossless algorithms for textual information.

We will first review in Section 10.2 the basic concepts of information theory applicable in the context of lossless text compression. We will then briefly describe the well-known compression algorithms in Section 10.3. A detailed description of these and other methods is now available in several excellent recent books [16, 28, 29, 35, and others]. In Sections 10.4 and 10.5, we present our own research on text compression. In particular, we present a number of preprocessing techniques that transform the text in some intermediate forms that produce better compression performance. We give test results of compression and timing performance with respect to text files in three corpora: Canterbury, Calgary [8], and Gutenberg [17] corpora. We conclude our chapter with a discussion of the compression utility web site that is now integrated with the Canterbury web site for availability via the Internet.

10.2 INFORMATION THEORY BACKGROUND

The general approach to text compression is to find a representation of the text requiring fewer binary digits. In its uncompressed form each character in the text is represented by an 8-bit ASCII code. It is common knowledge that such a representation is not very efficient because it treats frequent and less frequent characters equally. It makes intuitive sense to encode frequent characters with a smaller number of bits (less than 8) and less frequent characters with a larger number of bits (possibly more than 8 bits) in order to reduce the *average* number of *bits per character* (BPC). In fact this principle was the basis of the invention of the so-called Morse code and the famous Huffman code developed in the early 1950s. Huffman code typically reduces the size of the text file by about 50–60% or provides a compression rate of 4–5 BPC [35] based on the statistics of frequency of characters. In the late 1940s, Claude E. Shannon laid the foundation of information theory and modeled the text as the output of a source that generates a sequence of symbols from a finite alphabet A according to certain probabilities. Such a process is known as a *stochastic process* and in the special case when the probability of occurrence of the next symbol in the text depends on the previous symbols or its context it is called a *Markov process*. Furthermore, if the probability distribution of a typical sample represents the distribution of the text it is called an *ergodic process* [31, 32]. The information content of the text source can then be quantified by the entity called *entropy* H given by

$$H = -\sum_i p_i \log p_i, \tag{10.1}$$

where p_i denotes the probability of occurrence of the ith symbol of the alphabet in the text, the sum of all symbol probabilities is unity, the logarithm is with respect to base 2, and $-\log p_i$ is the amount of *information* in bits for the event (occurrence of the ith symbol). The expression of H is simply the sum of the number of bits required to represent the symbols multiplied by their respective probabilities. Thus the entropy H can be looked upon as defining the *average number of BPC* required to represent or encode the symbols of the alphabet. Depending on how the probabilities are computed or modeled, the value of entropy may vary. If the probability of a symbol is computed as the ratio of the number of times it appears in the text to the total number of symbols in the text, the so-called *static* probability, it is called an Order (0) model. Under this model, it is also possible to compute the *dynamic* probabilities, which can be roughly described as follows. At the beginning when no text symbol has emerged out of the source, assume that every symbol is equiprobable. As new symbols of the text emerge from the source, revise the probability values according to the actual frequency distribution of the symbols at that time. In general, an Order (k) model can be defined where the probabilities are computed based on the probability of distribution of the $(k+1)$-grams of symbols or equivalently, by taking into account the context of the preceding k symbols. A value

of $k = -1$ is allowed and is reserved for the situation when all symbols are considered equiprobable; that is, $p_i = 1/|A|$, where $|A|$ is the size of the alphabet A. When $k = 1$, the probabilities are based on *bigram* statistics or equivalently on the context of just one preceding symbol and similarly for higher values of k. For each value of k, there are two possibilities: the static model and the dynamic model as explained above. For practical reasons, a static model is usually built by collecting statistics over a test *corpus*, which is a collection of text samples representing a particular domain of application (viz., English literature, physical sciences, life sciences). If one is interested in a more precise static model for a given text, a *semistatic* model is developed in a two-pass process; in the first pass the text is read to collect statistics to compute the model and in the second pass an encoding scheme is developed. Another variation of the model is to use a specific text to *prime* or seed the model at the beginning and then build the model on top of it as new text files come in.

Independent of the model, there is entropy associated with each file under that model. Shannon's fundamental noiseless source coding theorem says that entropy defines a lower limit of the average number of bits needed to encode the source symbols [31]. The "worst" model from an information theoretic point of view is the Order (-1) model, the equiprobable model, giving the maximum value H_m of the entropy. Thus, for the 8-bit ASCII code, the value of this entropy is 8 bits. The redundancy R is defined to be the difference between the maximum entropy H_m and the actual entropy H. As we build better and better models by going to higher order k, the value of entropy will be lower, yielding a higher value of redundancy. The crux of lossless compression research boils down to developing compression algorithms that can find an encoding of the source using a model with minimum possible entropy and exploiting a maximum amount of redundancy. But incorporating a higher order model is computationally expensive and the designer must be aware of other performance metrics such as decoding or decompression complexity (the process of decoding is the reverse of the encoding process in which the redundancy is restored so that the text is again readable), speed of execution of compression and decompression algorithms, and use of additional memory.

Good compression means less storage space to store or archive the data, and it also means a reduced bandwidth requirement to transmit data from source to destination. This is achieved with the use of a *channel* that may be a simple point-to-point connection or a complex entity like the Internet. For the purpose of discussion, assume that the channel is noiseless; that is, it does not introduce error during transmission and it has a *channel capacity* C that is the maximum number of bits that can be transmitted per second. Since entropy H denotes the average number of bits required to encode a symbol, C/H denotes the average number of symbols that can be transmitted over the channel per second [31]. A second fundamental theorem of Shannon says that however clever you may get developing a compression scheme, you will never be able to transmit on average more than C/H symbols per second [31]. In other words, to use the available bandwidth effectively, H should be as low as possible, which means employing a compression scheme that yields minimum BPC.

10.3 CLASSIFICATION OF LOSSLESS COMPRESSION ALGORITHMS

The lossless algorithms can be classified into three broad categories: *statistical methods*, *dictionary methods*, and *transform-based methods*. We will give a very brief review of these methods in this section.

10.3.1 Statistical Methods

The classic method of statistical coding is *Huffman* coding [19]. It formalizes the intuitive notion of assigning shorter codes to more frequent symbols and longer codes to infrequent symbols. It is built bottom-up as a binary tree as follows: given the model or the probability distribution of

the list of symbols, the probability values are sorted in ascending order. The symbols are then assigned to the leaf nodes of the tree. Two symbols having the two lowest probability values are then combined to form a parent node representing a composite symbol that replaces the two child symbols in the list and whose probability equals the sum of the probabilities of the child symbols. The parent node is then connected to the child nodes by two edges with labels "0" and "1" in any arbitrary order. The process is then repeated with the new list (in which the composite node has replaced the child nodes) until the composite node is the only node remaining in the list. This node is called the root of the tree. The unique sequence of 0's and 1's in the path from the root to a leaf node is the Huffman code for the symbol represented by the leaf node. At the decoding end the same binary tree must be used to decode the symbols from the compressed code. In effect, the tree behaves like a dictionary that must 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 the literature.* The Huffman codes for all the symbols have what is called the *prefix property*, which is that no code of a symbol is the prefix of the code for another symbol, which makes the code *uniquely decipherable* (UD). This allows forming a code for a sequence of symbols by concatenating just the codes of the individual symbols and the decoding process can retrieve the original sequence of symbols without ambiguity. Note that a prefix code is not necessarily a Huffman code and may not obey the Morse principle and that a uniquely decipherable code does not have to be a prefix code, but the beauty of Huffman code is that it is UD, is prefix, and is also optimum within 1 bit of the entropy H. Huffman code is indeed optimum if the probabilities are $1/2^k$, where k is a positive integer. There are also Huffman codes called *canonical* Huffman codes, which use a lookup table or dictionary rather than a binary tree for fast encoding and decoding [28, 35].

Note that in the construction of the Huffman code, we started with a model. The efficiency of the code will depend on how good this model is. If we use higher order models, the entropy will be smaller, resulting in a shorter average code length. As an example, a word-based Huffman code is constructed by collecting the statistics of words in the text and building a Huffman tree based on the distribution of probabilities of words rather than the letters of the alphabet. It gives very good results but the overhead to store and transmit the tree is considerable. Since the leaf nodes contain all the distinct words in the text, the storage overhead is equal to having an English word dictionary shared between the sender and the receiver. We will return to this point later when we discuss our transforms. Adaptive Huffman codes take longer for both encoding and decoding because the Huffman tree must be modified at each step of the process. Finally, Huffman code is sometimes referred to as a variable-length code because a message of a fixed length may have variable-length representations depending on what letters of the alphabet are in the message.

In contrast, the *arithmetic code* encodes a variable-size message into fixed-length binary sequence. 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 semiclosed interval [0, 1) is partitioned into $|A|$ equal sized semiclosed intervals under the equiprobabilty 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 $|A|$ subintervals 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 subinterval 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 symbols. A simpler version of this algorithm

could use the same static distribution of probability at each iteration, avoiding recomputation of probabilities. The literature on arithmetic coding is vast and the reader is referred to the texts cited above [28, 29, 35] for further study.

The Huffman and arithmetic coders are sometimes referred to as the *entropy coder*. 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 best known modelers is "*prediction by partial match*" (PPM) [10, 23]. PPM uses a finite context Order (k) model, where k is the maximum context that is specified ahead of execution of the algorithm. The program maintains all the previous occurrences of context at each level of k in a table or a trie-like data structure with associated probability values for each context. If a context at a lower level is a suffix of a context at a higher level, this context is excluded at the lower level. At each level and for each distinct context at that level (except the level with $k = -1$), an *escape character* is defined whose frequency of occurrence is assumed to be equal to the number of distinct contexts encountered at that context level for the purpose of calculating its probability. The escape character is required to handle the situation when the encoder encounters a new context never encountered before at any context level to give the decoder a signal that the context length must be reduced by 1. During the encoding process, the algorithm estimates the probability of the occurrence of some given *next character* in the text stream as follows: The algorithm tries to find the current context of maximum length k for this character in the context table or trie. If the context is not found, it passes the probability of the escape character at this level for this context and goes down one level to the $k - 1$ context table to find the current context of length $k - 1$ and the process is repeated. If it continues to fail to find the context, it may go down ultimately to the $k = -1$ level corresponding to the equiprobable level for which the probability of any next character is $1/|A|$. If, on the other hand, a context of length q, $0 \leq q \leq k$, is found, then the probability of this next character is estimated to be the product of probabilities of escape characters at levels $k, k-1, \ldots, q+1$ multiplied by the probability of the context found at the qth level.

This probability value is then passed to the backend entropy coder (arithmetic coder) to obtain the encoding. Note that at the beginning there is no context available so the algorithm assumes a model with $k = -1$. The context lengths are shorter at the early stage of the encoding when only a few contexts have been seen. As the encoding proceeds, longer and longer contexts become available. The method to assign probability to the escape character is called method C and is as follows: At any level, with the current context, let the total number of symbols seen previously be n_t and let n_d be the total number of *distinct* context. Then the probability of the escape character is given by $n_d/(n_d + n_t)$. Any character other than that which appeared in this context n_c times will have a probability $n_c/(n_d + n_t)$. The intuitive explanation of this method, based on experimental evidence, is that if many distinct contexts are encountered, then the escape character will have higher probability, but if these distinct contexts tend to appear too many times, then the probability of the escape character decreases. The PPM method using method C for probability estimation is called the PPMC algorithm. There are a few other variations: PPMA uses method A, which simply assigns a count of 1 to the escape character, yielding its probability to be $1/(n_t + 1)$ and the probability of the character which appeared n_c times is $n_c/(n_t + 1)$. PPMB is very similar to PPMC except that the probability of a symbol is $(n_c - 1)/(n_d + n_t)$; that is, 1 is subtracted from the count n_c. If n_c is 1, since $n_c - 1$ becomes 0, no probability is assigned to the symbol and the count for the escape character is increased by 1. In PPMD, the escape character gets a probability of $n_d/2n_t$ and the symbol gets a probability $(2n_c - 1)/2n_t$. All of these methods have been proposed based on practical experience and have only some intuitive explanation but no theoretical basis. PPMD performs better than PPMC, which is better than either PPMA or PPMB. In one version of PPM, called PPM*, an arbitrary-length context is allowed which should give the optimal minimum entropy. In practice a model with $k = 5$ behaves as well as PPM* [11]. Although the PPM family of algorithms performs better than other compression algorithms in

terms of high compression ratio or low BPC, it is very computation intensive and slow due to the enormous amount of computation that is needed as each character is processed for maintaining the context information and updating their probabilities.

Dynamic Markov Compression (DMC) is another modeling scheme that is equivalent to the finite context model but uses a finite-state machine to estimate the probabilities of the input symbols which are bits rather than bytes or symbols as in PPM [12]. The model is adaptive and starts with a single state fsa (finite-state automaton) with transitions into itself. In general, the transitions from a given state are marked by $0/p$ or $1/q$, where p and q denote non-zero counts of the number of transitions from the given state to two other states (which may include the given state), respectively. Initially, the values of p and q are set to 1 to handle the zero-frequency problem. The estimate of probability of a 0 being the next input is $p/(p+q)$ and 1 being the next input is $q/(p+q)$. These probability values are used by the entropy coder (such as an arithmetic coder) to estimate the interval in the cumulative probability space for the input. If the next symbol is 0, it will then change the count p to $p+1$, make the appropriate state change, and continue. Similarly, for a 1 input it will change the value of q to $q+1$. The machine adapts to future inputs by accumulating the transitions with 0's and 1's with revised estimates of probabilities. If a state s is used heavily for input transitions from another state t (caused by either 1 or 0 input), s is *cloned* into two states by introducing a new state s_c in which some of the transitions from state t are directed. The threshold value of either p or q at which cloning must be triggered is agreed upon by the encoder and the decoder ahead of time. The ratio p/q of output transitions from the original state s is kept the same for the cloned state s_c, but the count values p and q from t to s and from t to s_c are adjusted for both the cloned state and the original state so that their total sum does not change. This is necessary to preserve the total number of transitions encountered by the fsa up to the time prior to cloning. The decoder performs the reverse starting with a single-state fsa and mimicking the cloning operation of the encoder whenever necessary. For further details, refer to [29, 35].

The bit-wise encoding and decoding take longer 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 [5].

10.3.2 Dictionary Methods

The dictionary methods, as the name implies, maintain a *dictionary* or *codebook* of words or text strings previously encountered in the text input, and data compression is achieved by replacing strings in the text with a reference to the string in the dictionary. The dictionary is *dynamic* or *adaptive* in the sense that it is constructed by adding new strings that are being read and it allows deletion of less frequently used strings if the size of the dictionary exceeds some limit. It is also possible to use a *static* dictionary, like the word dictionary, to compress the text. The most widely used compression algorithms (Gzip and Gif) are based on Ziv–Lempel or LZ77 coding [37] in which the text prior to the current symbol constitutes 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 so, they are replaced by a reference giving its relative starting position in the text. Because of the pattern-matching operation the encoding takes longer 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, includes one extra character in a previously coded string in the encoding scheme. A more popular variant of the LZ78 family is the so-called LZW algorithm, which led to the widely used *compress* algorithm. 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 the LZ77 and LZ78 compression family have been reported in the literature (see [28, 29] for further references).

10.3.3 Transform-Based Methods: The Burrows–Wheeler Transform (BWT)

The word "transform" has been used to describe this method because the text undergoes a transformation, which performs a permutation of the characters in the text so that characters having similar lexical context will cluster together in the output. Given the text input, the forward Burrows–Wheeler transform [6] forms all cyclic rotations of the characters in the text in the form of a matrix M whose rows are lexicographically sorted (with a specified ordering of the symbols in the alphabet). The last column L of this sorted matrix and an index r of the row where the original text appears in this matrix are the output of the transform. The text could be divided into blocks or the entire text could be considered as one block. The transformation is applied to individual blocks separately, and for this reason the method is referred to as the *block sorting* transform [14]. The repetition of the same character in the block might slow the sorting process; to avoid this, a run-length encoding step could precede the transform step. The Bzip2 compression algorithm based on the BWT transform uses this step and other steps as follows: The output of the BWT transform stage undergoes a final transformation using either move-to-front (MTF) encoding or distance coding (DC) [1], which exploits the clustering of characters in the BWT output to generate a sequence of numbers dominated by small values (viz., 0, 1, or 2) out of possible maximum value of $|A|$. This sequence of numbers is then sent to an entropy coder (Huffman or arithmetic) to obtain the final compressed form. The inverse operation of recovering the original text from the compressed output proceeds by decoding the inverse of the entropy decoder, then the inverse of MTF or DC, and then an inverse of BWT. The inverse of BWT obtains the original text given (L, r). This is done easily by noting that the first column of M, denoted F, is simply a sorted version of L. Define an index vector Tr of size $|L|$ such that $Tr[j] = i$ if and only if both $L[j]$ and $F[i]$ denote the kth occurrence of a symbol from A. Since the rows of M are cyclic rotations of the text, the elements of L precede the respective elements of F in the text. Thus $F[Tr[j]]$ cyclically precedes $L[j]$ in the text, which leads to a simple algorithm to reconstruct the original text.

10.3.4 Comparison of Performance of Compression Algorithms

An excellent discussion of the performance comparison of the important compression algorithms can be found in [35]. In general, the performance of compression methods depends on the type of data being compressed and there is a trade-off between compression performance, speed, and the use of additional memory resources. The authors report the following results with respect to the Canterbury corpus: In order of increasing compression performance (decreasing BPC), the algorithms can be listed as follows: order zero arithmetic; order zero Huffman giving over 4 BPC; the LZ family of algorithms, whose performance ranges from 4 BPC to around 2.5 BPC (gzip) depending on whether the algorithm is tuned for compression or speed. Order zero word-based Huffman (2.95 BPC) is a good contender for this group in terms of compression performance but it is two to three times slower and needs a word dictionary to be shared between the compressor and decompressor. The best performing compression algorithms are bzip2 (based on BWT), DMC, and PPM, all giving BPC ranging from 2.1 to 2.4. PPM is theoretically the best but is extremely

slow as is DMC; bzip2 strikes a middle ground—it gives better results than Gzip but is not an on-line algorithm because it needs the entire text or blocks of text in memory to perform the BWT transform. LZ77 methods (Gzip) are fastest for decompression, then the LZ78 technique, and then Huffman coders, and the methods using arithmetic coding are the slowest. Huffman coding is better for static applications, whereas arithmetic coding is preferable in adaptive and on-line coding. Bzip2 decodes faster than most other methods and it achieves good compression as well. A lot of new research on Bzip2 (see Section 10.4) has been carried out recently to push the performance envelope of Bzip2 in terms of both compression ratio and speed, and as a result bzip2 has become a strong contender to replace the popularity of Gzip and compress.

New research is under way to improve the compression performance of many of the algorithms. However, these efforts seem to have come to a point of saturation regarding lowering the compression ratio. To get a significant further improvement in compression, other means such as transforming the text before actual compression and the use of grammatical and semantic information to improve prediction models should be looked into. Shannon made some experiments with native speakers of the English language and estimated that the English language has an entropy of around 1.3 BPC. Thus, it seems that lossless text compression research is now confronted with the challenge of bridging a gap of about 0.8 BPC in terms of compression ratio. Of course, combining compression performance with other performance metrics like speed, memory overhead, and on-line capabilities seems to pose even a bigger challenge.

10.4 TRANSFORM-BASED METHODS: STAR (*) TRANSFORM AND LENGTH-INDEX PRESERVING TRANSFORM

In this section we present our research on new transformation techniques that can be used as preprocessing steps for the compression algorithms described in the previous section. The basic idea is to transform the text into some intermediate form, which can be compressed with better efficiency. The transformation is designed to exploit the natural redundancy of the language. We have developed a class of such transformations, each giving better compression performance over the previous ones and most of them giving better compression over current and classical compression algorithms discussed in the previous section. We first present a brief description of the first transform called the Star Transform (also denoted *-encoding). We then present four new transforms called length-index preserving transform (LIPT), initial letter preserving transform (ILPT), number index transform (NIT), and letter index transform (LIT), which produce better results.

The algorithms use a fixed amount of storage overhead in the form of a word dictionary for the particular corpus of interest and must be shared by the sender and receiver of the compressed files. The typical size of a dictionary for the English language is about 0.5 MB and can be downloaded along with application programs. If the compression algorithms are going to be used over and over again, which is true in all practical applications, the amortized storage overhead for the dictionary is negligibly small. We will present experimental results measuring the performance (compression ratio, compression times, and decompression times) of our proposed preprocessing techniques using three corpora: the Calgary, Canterbury, and Gutenberg corpora.

10.4.1 Star (*) Transformation

The basic idea underlying the star transformations is to define a unique signature of a word by replacing letters in a word with a special placeholder character (*) and keeping a minimum number of characters to identify the word uniquely [15, 20, 21]. For an English language dictionary D of size 60,000 words, we observed that we needed at most two characters of the original words

to keep their identity intact. In fact, it is not necessary to keep any letters of the original word as long as a unique representation can be defined. The dictionary is divided into subdictionaries D_s containing words of length, $1 \leq s \leq 22$, because the maximum length of a word in English dictionary is 22 and there are two words of length 1, viz., "a" and "I".

The following encoding scheme is used for the words in D_s: The first word is represented as sequence of s stars. The next 52 words are represented by a sequence of $s - 1$ stars followed by a single letter from the alphabet: $= (a, b, \ldots, z, A, B, \ldots, Z)$. The next 52 words have a similar encoding except that the single letter appears in the next to last position. This will continue until all the letters occupy the first position in the sequence. The following group of words has $s - 2$ *'s and the remaining two positions are taken by unique pairs of letters from the alphabet. This process can be continued to obtain a total of 53^s unique encodings, which is more than sufficient for English words. A large fraction of these combinations are never used; for example, for $s = 2$, there are only 17 words and for $s = 8$, there are about 9000 words in the English dictionary. As an example of star encoding the sentence "*Our philosophy of compression is to transform the text into some intermediate form which can be compressed with better efficiency and which exploits the natural redundancy of the language in making this transformation*" can be *-encoded as

a******** ** ********* *a *b ******** *** ***a ***c
b ******* ***d ***** **b *c *********a ***e ***
*** **********a **c ***** ******* *** ******a *********
b ** *** ******* * d *****a **** **********

There are exactly five 2-letter words in the sentence (of, is, to, be, in) which can be uniquely encoded as (**, *a, *b, *c, *d) and the other groups of words can be uniquely encoded in a similar manner. Given such an encoding, the original word can be retrieved from the dictionary that contains a one-to-one mapping between encoded words and original words. The encoding produces an abundance of * characters in the transformed text, making it the most frequently occurring character. If the word in the input text is not in the English dictionary (viz., a new word in the lexicon), it will be passed to the transformed text unaltered. The transformed text must also be able to handle special characters, punctuation marks, and capitalization. The space character is used as word separator. The character "~" at the end of an encoded word indicates that the first letter of the input text word is capitalized. The character "`" indicates that all the characters in the input word are capitalized. A capitalization mask, preceded by the character "^", is placed at the end of an encoded word to denote capitalization of characters other than the first letter and all capital letters. The character "\" is used as an escape character for encoding the occurrences of *, ~, `, ^, and \ in the input text. The transformed text can now be the input to any available losslesss text compression algorithm, including Bzip2, where the text undergoes two transformation: first the *-transform and then a BWT transform.

10.4.2 Length-Index Preserving Transform (LIPT)

A different twist to our transformation comes from the observation that the frequency of occurrence of words in the corpus as well as the predominance of certain lengths of words in the English language might play an important role in revealing additional redundancy to be exploited by the backend algorithm. The frequency of occurrence of symbols, k-grams, and words in the form of probability models, of course, forms the cornerstone of all compression algorithms but none of these algorithms considered the distribution of the length of words directly in the models. We were motivated to consider the length of words as an important factor in English text as we gathered word frequency data according to lengths for the Calgary, Canterbury [8], and Gutenberg

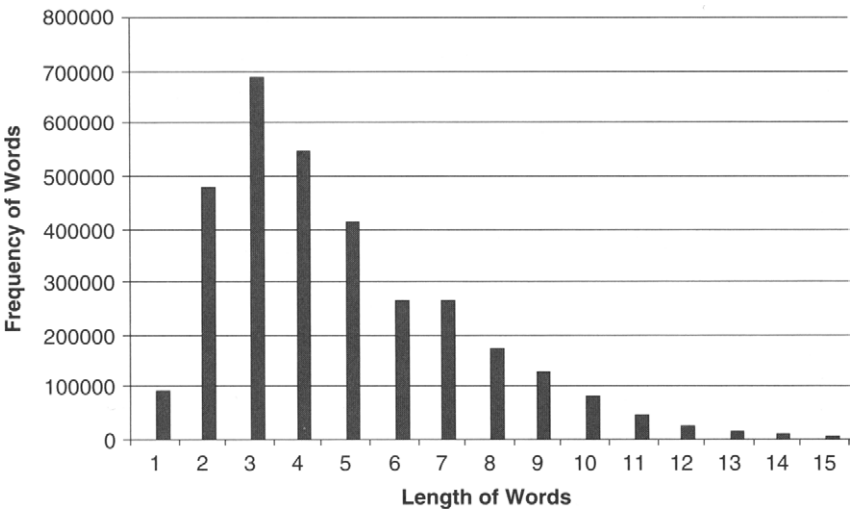

FIGURE 10.1
Frequency of English words versus length of words in the test corpus.

corpora [17]. A plot showing the total word frequency versus the word length results for all the text files in our test corpus (combined) is shown in Fig. 10.1.

It can be seen that most words lie in the range of length 1 to 10. Most words have length 2 to 5. The word length and word frequency results provided a basis to build context in the transformed text. LIPT can be regarded as the first step of a multistep compression algorithm such as Bzip2, which includes run-length encoding, BWT, move-to-front encoding, and Huffman coding. LIPT can be used as an additional component in Bzip2 before run-length encoding or it can simply replace it. Compared to the *-transform, we also made a couple of modifications to improve the timing performance of LIPT. For *-transform, searching for a transformed word for a given word in the dictionary during compression and doing the reverse during decompression takes time, which degrades the execution times. The situation can be improved by presorting the words lexicographically and doing a binary search on the sorted dictionary during both the compression and decompression stages. The other new idea that we introduce is to be able to access the words during the decompression phase in a random access manner so as to obtain fast decoding. This is achieved by generating the address of the words in the dictionary by using, not numbers, but the letters of the alphabet. We need a maximum of three letters to denote an address and these letters introduce an artificial but useful context for the backend algorithms to further exploit the redundancy in the intermediate transformed form of the text. The LIPT encoding scheme makes use of the recurrence of same-length words in the English language to create context in the transformed text that the entropy coders can exploit.

LIPT uses a static English language dictionary of 59,951 words, having a size of around 0.5 MB. LIPT uses a transform dictionary of around 0.3 MB. There is one-to-one mapping of a word from the English to the transform dictionary. The words not found in the dictionary are passed as they are. To generate the LIPT dictionary (which is done off-line), we need the source English dictionary to be sorted on blocks of lengths, and words in each block should be sorted according to their frequency of use.

A dictionary D of words in the corpus is partitioned into disjoint dictionaries D_i, each containing words of length i, where $i = 1, 2, \ldots, n$. Each dictionary D_i is partially sorted according to the frequency of words in the corpus. Then a mapping is used to generate the encoding for

all words in each dictionary D_i. $D_i[j]$ denotes the jth word in the dictionary D_i. In LIPT, the word $D_i[j]$ in the dictionary D is transformed as $*c_{len}[c][c][c]$ (the square brackets denote the optional occurrence of a letter of the alphabet enclosed and are not part of the transformed representation) in the transform dictionary D_{LIPT}, where c_{len} stands for a letter in the alphabet [a–z, A–Z] each denoting a corresponding length [1–26, 27–52] and each c is in [a–z, A–Z]. If $j = 0$, then the encoding is $*c_{len}$. For $j > 0$, the encoding is $*c_{len}\ c[c][c]$. Thus, for $1 \leq j \leq 52$ the encoding is $*c_{len}c$; for $53 \leq j \leq 2756$ it is $*c_{len}cc$, and for $2757 \leq j \leq 140{,}608$ it is $*c_{len}ccc$. Thus, the 0th word of length 10 in the dictionary D will be encoded as "*j" in D_{LIPT}, $D_{10}[1]$ as "*ja", $D_{10}[27]$ as "*jA", $D_{10}[53]$ as "*jaa", $D_{10}[79]$ as "*jaA", $D_{10}[105]$ as "*jba", $D_{10}[2757]$ as "*jaaa", $D_{10}[2809]$ as "*jaba", and so on.

The transform must also be able to handle special characters, punctuation marks, and capitalization. The character "*" is used to denote the beginning of an encoded word. The handling of capitalization and special characters is the same as in *-encoding. Our scheme allows for a total of 140,608 encodings for each word length. Since the maximum length of English words is around 22 and the maximum number of words in any D_i in our English dictionary is less than 10,000, our scheme covers all English words in our dictionary and leaves enough room for future expansion. If the word in the input text is not in the English dictionary (viz., a new word in the lexicon), it will be passed to the transformed text unaltered.

The decoding steps are as follows: The received encoded text is first decoded using the same backend compressor used at the sending end and the transformed LIPT text is recovered. The words with "*" represent transformed words and those without "*" represent non-transformed words and do not need any reverse transformation. The length character in the transformed words gives the length block and the next three characters give the offset in the respective block. The words are looked up in the original dictionary D in the respective length block and at the respective position in that block as given by the offset characters. The transformed words are replaced with the respective words from dictionary D. The capitalization mask is then applied.

10.4.3 Experimental Results

The performance of LIPT is measured using Bzip2 -9 [6, 9, 22, 30], PPMD (order 5) [11, 23, 28], and Gzip -9 [28, 35] as the backend algorithms in terms of average BPC. Note that these results include some amount of precompression because the size of the LIPT text is smaller than the size of the original text file. By average BPC we mean the unweighted average (simply taking the average of the BPC of all files) over the entire text corpus. The test corpus is shown in Table 10.1. Note that all the files given in Table 10.1 are text files extracted from the corpora.

We used SunOS Ultra-5 to run all our programs and to obtain results. We are using a 60,000-word English dictionary that takes 557,537 bytes. The LIPT dictionary takes only 330,636 bytes compared to the *-encoded dictionary, which takes as much storage as that of the original dictionary. LIPT achieves a bit of compression in addition to preprocessing the text before application to any compressor.

The results can be summarized as follows: The average BPC using the original Bzip2 is 2.28, and using Bzip2 with LIPT gives an average BPC of 2.16, a 5.24% improvement. The average BPC using the original PPMD (order 5) is 2.14, and using PPMD with LIPT gives an average BPC of 2.04, an overall improvement of 4.46%. The average BPC using the original Gzip -9 is 2.71, and using Gzip -9 with LIPT the average BPC is 2.52, a 6.78% improvement.

Figure 10.2 gives a bar chart comparing the BPC of the original Bzip2, PPMD, and the compressors in conjunction with LIPT for a few text files extracted from our test corpus. From Fig. 10.2 it can be seen that Bzip2 with LIPT (the second bar in Fig. 10.2) is close to the original

Table 10.1 Text Files and Their Sizes (a) from Calgary Corpus and (b) from Canterbury Corpus and Project Gutenberg Corpus That Were Used in Our Tests

File Name	Actual Size
(a)	
Calgary	
Bib	111,261
book1	768,771
book2	610,856
News	377,109
paper1	53,161
paper2	82,199
paper3	46,526
paper4	13,286
paper5	11,954
paper6	38,105
Progc	39,611
Progl	71,646
Progp	49,379
Trans	93,695
(b)	
Canterbury	
alice29.txt	152,089
asyoulik.txt	125,179
cp.html	24,603
fields.c	11,150
grammar.lsp	3,721
lcet10.txt	426,754
plrabn12.txt	481,861
xargs.1	4,227
bible.txt	4,047,392
kjv.Gutenberg	4,846,137
world192.txt	2,473,400
Project Gutenberg	
1musk10.txt	1,344,739
anne11.txt	586,960
world95.txt	2,988,578

PPMD (the third bar in Fig. 10.2) in bits per character. From Fig. 10.2 it can also be seen that in instances like paper5, paper4, progl, paper2, asyoulik.txt, and alice29.txt, Bzip2 with LIPT beats the original PPMD in terms of BPC. The difference between the average BPC for Bzip2 with LIPT (2.16) and that for the original PPMD (2.1384) is only around 0.02 bits; i.e., the average BPC for Bzip2 with LIPT is only around 1% more than that for the original PPMD. Table 10.2 gives a summary comparison of BPC for the original Bzip2 -9, PPMD (order 5), Gzip -9, Huffman (character based), word-based arithmetic coding, and these compressors with Star-encoding and LIPT. The data in Table 10.2 show that LIPT performs much better than Star-encoding and original algorithms except for character-based Huffman and Gzip -9.

Table 10.2 also shows that Star-encoding gives a better average BPC performance for character-based Huffman, Gzip, and Bzip2 but gives a worse average BPC performance for word-based

Table 10.2 Summary of BPC Results

	Original (BPC)	*-encoded (BPC)	LIPT (BPC)
Huffman (character based)	4.87	**4.12**	4.49
Arithmetic (word based)	2.71	2.90	**2.61**
Gzip-9	2.70	**2.52**	**2.52**
Bzip2	2.28	2.24	**2.16**
PPMD	2.13	2.13	**2.04**

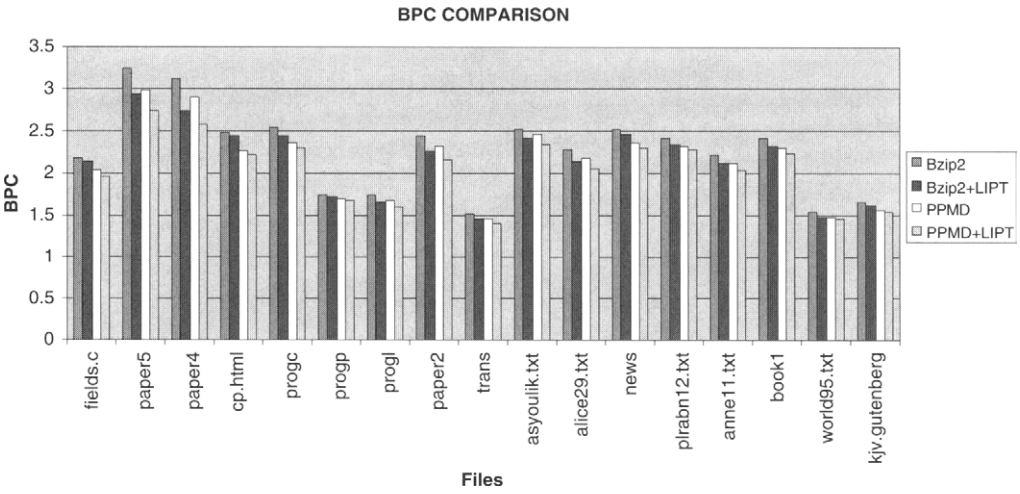

FIGURE 10.2
Bar chart giving comparison of Bzip2, Bzip2 with LIPT, PPMD, and PPPMD with LIPT.

arithmetic coding and PPMD. This is due to the presence of the non-English words and special symbols in the text. For a pure text file, for example, the dictionary itself, the star dictionary has a BPC of 1.88 and the original BPC is 2.63 for PPMD. The improvement is 28.5% in this case. Although the average BPC for Star-encoding is worse than that for the original PPMD, there are 16 files that show improved BPC and 12 files that show worse BPC. The number of words in the input text that are also found in English dictionary D is an important factor for the final compression ratio. For character-based Huffman, Star-encoding performs better than the original Huffman and LIPT with Huffman. This is because in Star-encoding there are repeated occurrences of the character "*", which gets the highest frequency in the Huffman codebook and is thus encoded with the lowest number of bits, resulting in better compression results than for the original and the LIPT files.

We focus our attention on LIPT over Bzip2 (which uses BWT), Gzip, and PPM algorithms because Bzip2 and PPM outperform other compression methods and Gzip is commercially available and commonly used. Of these, a BWT-based approach has proved to be the most efficient and a number of efforts have been made to improve its efficiency. The latest efforts include works by Balkenhol *et al.* [4], Seward [30], Chapin [9], and Arnavut [1]. PPM, on the other hand, gives a better compression ratio than BWT but is very slow in terms of execution time. A number of efforts have been made to reduce the time for PPM and also to improve the compression ratio.

Table 10.3 BPC Comparison of Approaches Based on BWT

File [Reference]	Average BPC
MBSWIC [1]	2.21
BKS98 [4]	2.105
Best x of $2x - 1$ [9]	2.11
Bzip2 with LIPT	**2.07**

Table 10.4 BPC Comparison of New Approaches Based on Prediction Models

File [Reference]	Average BPC
Multialphabet CTW order 16 [27]	2.021
NEW Effros [13]	2.026
PPMD (order 5) with LIPT	**1.98**

Sadakane *et al.* [27] have given a method where they have combined PPM and CTW [35] to get better compression. Effros [13] has given a new implementation of PPM* with the complexity of BWT. Tables 10.3 and 10.4 give a comparison of compression performance of our proposed transform, which shows that LIPT has a better average BPC than all the other methods cited.

The Huffman compression method also needs the same static dictionary to be shared at both the sender and the receiver ends, as does our method. The canonical Huffman [35] method assigns variable-length addresses to words using bits, and LIPT assigns variable-length offset in each length block using letters of the alphabet. Due to these similarities we compare the word-based Huffman with LIPT (we used Bzip2 as the compressor). Huffman and LIPT both sort the dictionary according to frequency of use of words. Canonical Huffman assigns a variable address to the input word, building a tree of locations of words in the dictionary and assigning 0 or 1 to each branch of the path. LIPT exploits the structural information of the input text by including the length of the word in encoding. LIPT also achieves a precompression due to the variable offset scheme. In Huffman, if new words are added, the whole frequency distribution must be recomputed as do the Huffman codes for them. Comparing the average BPC, the Managing Gigabyte [35] word-based Huffman model has 2.506 BPC for our test corpus. LIPT with Bzip2 has a BPC value of 2.17. The gain is 13.44%. LIPT does not give an improvement over word-based Huffman for files with mixed text such as source files for programming languages. For files with more English words, LIPT shows a consistent gain.

LIPT gives better results with some of the newer compression methods reported in the literature and web sites such as YBS [36], RK [25, 26], and PPMonstr [24]. The average BPC using LIPT along with these methods is around 2.00, which is better than any of these algorithms as well as the original Bzip2 and PPMD. For details, the reader is referred to [2, 3].

10.4.4 Timing Performance Measurements

The improved compression performance of our proposed transform comes with a penalty of degraded timing performance. For off-line and archival storage applications, such penalties in timing are quite acceptable if a substantial savings in storage space can be achieved. The increased

compression/decompression times are due to frequent access to a dictionary and its transform dictionary. To alleviate the situation, we have developed efficient data structures to expedite access to the dictionaries and memory management techniques using caching. Realizing that certain on-line algorithms might prefer not to use a preassigned dictionary, we also have been working on a new family of algorithms, called M5zip, to obtain the transforms dynamically with no dictionary and with small dictionaries (7947 words and 10,000 words), which will be reported in future papers.

The experiments were carried out on a Sun Microsystems Ultra Sparc-IIi 360-MHz machine housing SunOS 5.7. In our experiments we compare compression times of Bzip2, Gzip, and PPMD against Bzip2 with LIPT, Gzip with LIPT, and PPMD with LIPT. During the experiments we used the -9 option for Gzip. This option supports better compression. Average compression time, for our test corpus, using LIPT with Bzip2 -9, Gzip -9, and PPMD is 1.79 times slower, 3.23 times slower, and fractionally (1.01 times) faster than original Bzip2, Gzip, and PPMD, respectively. The corresponding results for decompression times are 2.31 times slower, 6.56 times slower, and almost the same compared to original Bzip2, Gzip, and PPMD, respectively. Compression using Bzip2 with LIPT is 1.92 times faster and decompression is 1.98 times faster than original PPMD (order 5). The increase in time over standard methods is due to time spent in preprocessing the input file. Gzip uses the -9 option to achieve maximum compression; therefore we find that the times for compression using Bzip2 are less than those for Gzip. When maximum compression option is not used, Gzip runs much faster than Bzip2.

Decompression time for methods using LIPT includes decompression using compression techniques plus reverse transformation time. Bzip2 with LIPT decompresses 2.31 times slower than original Bzip2, Gzip with LIPT decompresses 6.56 times slower than Gzip, and PPMD with LIPT is almost the same.

10.5 THREE NEW TRANSFORMS—ILPT, NIT, AND LIT

We will briefly describe our three new lossless reversible text transforms based on LIPT. We give experimental results for the new transforms and discuss them briefly. For details on these transformations and experimental results the reader is referred to [3]. Note that there is no significant effect on the time performance as the dictionary loading method remains the same and the number of words also remains the same in the static English dictionary D and transform dictionaries. Hence we will give only the BPC results obtained with different approaches for the corpus.

The *Initial Letter Preserving transform* is similar to LIPT except that we sort the dictionary into blocks based on the lexicographic order of starting letter of the words. We sort the words in each letter block according to descending order of frequency of use. The character denoting length in LIPT (character after "*") is replaced by the starting letter of the input word; i.e., instead of $*c_{len}[c][c][c]$, for ILPT, it becomes $*c_{init}[c][c][c]$, where c_{init} denotes the initial (staring) letter of the word. Everything else is handled the same way as in LIPT. Bzip2 with ILPT has an average BPC of 2.12 for all the files in all the corpora combined. This means that Bzip2 -9 with ILPT shows 6.83% improvement over the original Bzip2 -9. Bzip2 -9 with ILPT shows 1.68% improvement over Bzip2 with LIPT.

The *Number Index Transform* scheme uses variable addresses based on letters of the alphabet instead of numbers. We wanted to compare this using a simple linear addressing scheme with numbers, i.e., giving addresses 0–59,950 to the 59,951 words in our dictionary. Using this scheme on our English dictionary D, sorted according to length of words and then sorted according to frequency within each length block, gave deteriorated performance compared to LIPT. So we sorted the dictionary globally according to descending order of word usage frequency. No blocking

was used in the new frequency-sorted dictionary. The transformed words are still denoted by character "*". The first word in the dictionary is encoded as "*0", the 1000th word is encoded as "*999", and so on. Special character handling is same as in LIPT. We compare the BPC results for Bzip2 -9 with the new transform NIT with Bzip2 -9 with LIPT. Bzip2 with NIT has an average BPC of 2.12 for all the files in all the corpora combined. This means that Bzip2 -9 with NIT shows 6.83% improvement over the original Bzip2 -9. Bzip2 -9 with NIT shows 1.68% improvement over Bzip2 with LIPT.

Combining the approach taken in NIT and using letters to denote offset, we arrive at another transform which is same as NIT except that now we use letters of the alphabet [a–z; A–Z] to denote the index or the linear address of the words in the dictionary, instead of numbers. This scheme is called the *Letter Index Transform*. Bzip2 with LIT has an average BPC of 2.11 for all the files in all corpora combined. This means that Bzip2 -9 with LIT shows 7.47% improvement over the original Bzip2 -9. Bzip2 -9 with LIT shows 2.36% improvement over Bzip2 with LIPT.

The transform dictionary sizes vary with the transform. The original dictionary takes 557,537 bytes. The transformed dictionaries for LIPT, ILPT, NIT, and LIT are 330,636, 311,927, 408,547, and 296,947 bytes, respectively. Note that LIT has the smallest size dictionary and shows uniform compression improvement over other transforms for almost all the files.

Because of its performance is better than that of other transforms, we compared LIT with PPMD, YBS, RK, and PPMonstr. PPMD (order 5) with LIT has an average BPC of 1.99 for all the files in all corpora combined. This means that PPMD (order 5) with LIT shows 6.88% improvement over the original PPMD (order 5). PPMD with LIT shows 2.53% improvement over PPMD with LIPT. RK with LIT has an average BPC value lower than that of RK with LIPT. LIT performs well with YBS, RK, and PPMonstr, giving 7.47, 5.84, and 7.0% improvement, respectively, over the original methods. It is also important to note that LIT with YBS outperforms Bzip2 by 10.7% and LIT with PPPMonstr outperforms PPMD by 10%.

LIPT introduces frequent occurrences of common characters for BWT and good context for PPM also as it compresses the original text. Cleary *et al.* [11] and Larsson [22] have discussed the similarity between PPM and Bzip2. PPM uses a probabilistic model based on the context depth and uses the context information explicitly. On the other hand, the frequency of similar patterns and the local context affect the performance of BWT implicitly. Fenwick [14] also explains how BWT exploits the structure in the input text. LIPT introduces added structure along with smaller file size, leading to better compression after Bzip2 or PPMD is applied. LIPT exploits the distribution of words in the English language based on the length of the words as given in Fig. 10.1. The sequence of letters to denote the address also has some inherent context depending on how many words are in a single group, which also opens another opportunity to be exploited by the backend algorithm at the entropy level.

There are repeated occurrences of words with the same length in a usual text file. This factor contributes to introducing good and frequent context and thus higher probability of occurrence of same characters (space, "*", and characters denoting length of words) that enhance the performance of Bzip2 (which uses BWT) and PPM as proved by results given earlier in this report. LIPT generates an encoded file, which is smaller in size than the original text file. Because of the small input file along with a set of artificial but well-defined deterministic contexts, both BWT and PPM can exploit the context information very effectively, producing a compressed file that is smaller than the file without using LIPT.

Our research has shown that the compression given by compression factor C is inversely proportional to the product of file size reduction factor F achieved by a transform and entropy S_t. Smaller F and proportionally smaller entropy of the transformed file mean higher compression. For theoretical details, the reader is referred to [3].

Our transforms keep the word level context of the original text file but adopt a new context structure at the character level. The frequency of the repeated words remains the same in both the

original and the transformed files. The frequency of characters is different. Both of these factors along with the reduction in file size contribute toward the better compression achieved with our transforms. The context structure affects the entropy S or S_t and reduction in file size affects F. We have already discussed the finding that compression is inversely proportional to the product of these two variables. In all our transforms, to generate our transformed dictionary, we have sorted the words according to frequency of usage in our English dictionary D. For LIPT, words in each length block in English dictionary D are sorted in descending order according to frequency. For ILPT, there is a sorting based on descending order of frequency inside each initial letter block of English dictionary D. For NIT, there is no blocking of words. The whole dictionary is one block. The words in the dictionary are sorted in descending order of frequency. LIT uses the same structure of dictionary as NIT. Sorting of words according to frequency plays a vital role in the size of the transformed file and also its entropy. Arranging the words in descending order of usage frequency results in shorter codes for more frequently occurring words and longer codes for less frequently occurring words. This fact leads to smaller file sizes.

10.6 CONCLUSIONS

In this chapter, we have given an overview of classical and recent lossless text compression algorithms and then presented our research on text compression based on transformation of text as a preprocessing step for use by the available compression algorithms. For comparison purposes, we were primarily concerned with Gzip, Bzip2, a version of PPM called PPMD, and word-based Huffman. We gave a theoretical explanation of why our transforms improved the compression performance of the algorithms. We have developed a web site (http://vlsi.cs.ucf.edu) as a test bed for all compression algorithms. To use this, one must go to the "on-line compression utility" and the client could then submit any text file for compression using all the classical compression algorithms, some of the most recent algorithms including Bzip2, PPMD, YBS, RK, and PPMonstr, and, of course, all the transform-based algorithms that we developed and reported in this chapter. The site is still under construction and is evolving. One nice feature is that the client can submit a text file and obtain statistics of all compression algorithms presented in the form of tables and bar charts. The site is being integrated with the Canterbury web site.

ACKNOWLEDGMENTS

The research reported in this chapter is based on a research grant supported by NSF Award IIS-9977336. Several members of the M5 Research Group at the School of Electrical Engineering and Computer Science of University of Central Florida participated in this research. The contributions of Dr. Robert Franceschini and Mr. Holger Kruse with respect to Star transform work are acknowledged. The collaboration of Mr. Nitin Motgi and Mr. Nan Zhang in obtaining some of the experimental results is also gratefully acknowledged. Mr. Nitin Motgi's help in developing the compression utility web site is also acknowledged.

10.7 REFERENCES

1. Arnavut, Z., 2000. Move-to-front and inversion coding. *Proceedings of Data Compression Conference*, Snowbird, UT. pp. 193–202.
2. Awan, F., and A. Mukherjee, 2001. LIPT: A lossless text transform to improve compression. *International Conference on Information Theory: Coding and Computing*, Las Vegas, NV. IEEE Comput. Soc., Los Alamitos, CA. pp. 452–460, April 2001.

3. Awan, F., 2001. *Lossless Reversible Text Transforms*, M.S. thesis, University of Central Florida, Orlando, July 2001.
4. Balkenhol, B., S. Kurtz, and Y. M. Shtarkov, 1999. Modifications of the Burrows Wheeler data compression algorithm. In *Proceedings of Data Compression Conference*, Snowbird, UT. pp. 188–197.
5. Bell, T. C., and A. Moffat, 1989. A note on the DMC data compression scheme. *The British Computer Journal*, Vol. 32, No. 1, pp. 16–20.
6. Burrows, M., and D. J. Wheeler, 1994. A Block-Sorting Lossless Data Compression Algorithm. SRC Research Report 124, Digital Systems Research Center, Palo Alto, CA.
7. Bzip2 Memory Usage. Available at http://krypton.mnsu.edu/krypton/software/bzip2.html.
8. Calgary and Canterbury Corpi. Available at http://corpus.canterbury.ac.nz.
9. Chapin, B., 2000. Switching between two on-line list update algorithms for higher compression of Burrows–Wheeler transformed data. In *Proceedings of Data Compression Conference*, Snowbird, UT. pp. 183–191.
10. Cleary, J. G., and I. H. Witten, 1984. Data compression using adaptive coding and partial string matching. *IEEE Transactions and Communications*, Vol. COM-32, No. 4, pp. 396–402.
11. Cleary, J. G., W. J. Teahan, and I. H. Witten, 1995. Unbounded length contexts for PPM. In *Proceedings of Data Compression Conference*, pp. 52–61, March 1995.
12. Cormack, G. V., and R. N. Horspool, 1987. Data compression using dynamic Markov modeling. *Computer Journal*, Vol. 30, No. 6, pp. 541–550.
13. Effros, M., 2000. PPM Performance with BWT complexity: A new method for lossless data compression. *Proceedings of Data Compression Conference*, Snowbird, UT. pp. 203–212.
14. Fenwick, P., 1996. Block sorting text compression. In *Proceedings of the 19th Australian Computer Science Conference, Melbourne, Australia, January 31–February 2, 1996.*
15. Franceschini, R., and A. Mukherjee, 1996. Data compression using encrypted text. In *Proceedings of the Third Forum on Research and Technology*, Advances on Digital Libraries, ADL 96, pp. 130–138.
16. Gibson, J. D., T. Berger, T. Lookabaugh, D. Lindbergh, and R. L. Baker, 1998. Digital Compression for Multimedia: Principles and Standards. Morgan Kaufmann, San Mateo, CA.
17. Available at http://www.promo.net/pg/.
18. Howard, P. G., 1993. *The Design and Analysis of Efficient Lossless Data Compression Systems*, Ph.D. thesis. Brown University, Providence, RI.
19. Huffman, D. A., 1952. A method for the construction of minimum redundancy codes. *Proceedings of the Institute of Radio Engineers*, Vol. 40, pp. 1098–1101.
20. Kruse, H., and A. Mukherjee, 1997. Data compression using text encryption. In *Proceedings of Data Compression Conference*, p. 447, IEEE Comput. Soc., Los Alamitos, CA.
21. Nelson, M. R., "Star Encoding", *Dr. Dobb's Journal*, August, 2002, pp. 94–96.
22. Larsson, N. J., 1998. The Context Trees of Block Sorting Compression. N. Jesper Larsson: The context trees of block sorting compression. In *Proceedings of Data Compression Conference*, pp. 189–198.
23. Moffat, A., 1990. Implementing the PPM data compression scheme, *IEEE Transactions on Communications*, Vol. 38, No. 11, pp. 1917–1921.
24. PPMDH. Available at ftp://ftp.elf.stuba.sk/pub/pc/pack/.
25. RK archiver. Available at http://rksoft.virtualave.net/.
26. RK archiver. Available at http://www.geocities.com/SiliconValley/Lakes/1401/compress.html.
27. Sadakane, K., T. Okazaki, and H. Imai, 2000. Implementing the context tree weighting method for text compression. In *Proceedings of Data Compression Conference*, Snowbird, UT. pp. 123–132.
28. Salomon, D., 2000. *Data Compression: The Complete Reference*, 2nd ed. Springer-Verlag, Berlin/New York.
29. Sayood, K., 1996. *Introduction to Data Compression.* Morgan Kaufman, San Mateo, CA.
30. Seward, J., 2000. On the performance of BWT sorting algorithms. In *Proceedings of Data Compression Conference*, Snowbird, UT. pp. 173–182.
31. Shannon, C. E., and W. Weaver, 1998. *The Mathematical Theory of Communication.* Univ. of Illinois Press, Champaign.
32. Shannon, C. E., 1948. A mathematical theory of communication. *Bell System Technical Journal*, Vol. 27, pp. 379–423, 623–656.

33. Available at http://trec.nist.gov/data.html.
34. Willems, F., Y. M. Shtarkov, and T. J. Tjalkens, 1995. The context-tree weighting method: Basic properties. *IEEE Transactions on Information Theory*, Vol. IT-41, No. 3, pp. 653–664.
35. Witten, I. H., A. Moffat, and T. Bell, 1999. *Managing Gigabyte, Compressing and Indexing Documents and Images*, 2nd ed. Morgan Kaufmann, San Mateo, CA.
36. YBS Compression Algorithm. Available at http://artest1.tripod.com/texts18.html.
37. Ziv, J., and A. Lempel, 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, pp. 337–343, May 1977.