Journal of Information Technology and Engineering
Vol. 1 No. 1 (June, 2016)

Evolutionary Approach for Compressing Textual Documents

Ashutosh GUPTA[1] and Suneeta AGARWAL[2]

[1] CSED, IERT Allahabad, UP 211002, India, *E-mail:ashutosh333@rediffmail.com*

[2] CSED, MNNITAllahabad, UP 211002, India, *E-mail: suneeta@mnnit.ac.in*

Received: 03rd March 2016 Revised: 14th May 2016 Accepted: 01st June 2016

Abstract: *In the last twenty years, we have seen a vast explosion of textual information flow over web through electronic mail, web browsing and information retrieval systems etc. In this paper we present a dictionary based lossless transform algorithm. The approach for transforming a text is based on model of natural distribution for words. The transformation generates a static dictionary, which is small in size and helpful for increasing the redundancy in the text. The overhead of storing the dictionaries with backend compressors are minimal. Initial experimental results show that the facilitated with Transform Module, bzip2, gzip and compress achieves an average improvement in compression ratio of 29.28% over bzip2, 28.71% over gzip, and 19.74% over compress. Experimental results shows that the compression ratio obtained by compressing the transformed text is much better than compression ratio obtained by directly compressing the text using some of the well known existing algorithms.*

Keywords: *Text Compression, Heap's Law, Stop-word Frequency Distribution, Transform.*

1. INTRODUCTION

In the last twenty years, we have seen a vast explosion of textual information flow over web through electronic mail, web browsing and information retrieval systems etc. The importance of data compression is likely to be enhancing in the future, as there is continuous increase in amount of data that need to be transformed or archived. The aim of data compression is to exploit the redundancies in the data to reduce its space usage. The most widely used data compression algorithms are based on the sequential data compressors of Lempel and Ziv [22, 23]. Statistical modeling techniques may produce superior compression [39], but are significantly slower.

Text compression is about finding ways to represent the text in less space. This is accomplished by substituting the symbols in the text by equivalent ones that are represented using a smaller number of bits or bytes. For large text collection, text compression appears as an attractive option for reducing costs. The gain obtained from compressing text is that it requires less storage space, it takes less time to be read from disk or transmitted over a communication link, and it takes less time to search. The savings of space obtained by a compression method is measured by the compression ratio, defined as the size of compressed file as a percentage of the uncompressed file. There are other important aspects to be considered, such as compression and decompression speed. In some situations, decompression speed is more important than compression speed. For instance, this is the case with textual databases and documentation systems in which it is common to compress the text once and to read it many times from disk.

In the field of data compression, Researchers developed various approaches such as Huffman encoding [4], arithmetic encoding [20, 13], Ziv-Lempel family [22, 23, 38], Dynamic Markov compression, Prediction with partial matching [3] and Burrows Wheeler Transform [27, 25, 34, 36] based algorithms, etc. BWT permutes the symbol of a data sequence that share the same unbounded context by cyclic rotation followed by lexicographic sort operations. BWT uses move-to-front and an entropy coder as the backend compressor. PPM is slow and also consumes large amount of memory to store context information but PPM achieves better compression that almost all existing compression algorithms.

In the recent past, Awan and Mukherjee [10], Franceschini and Mukherjee [32] developed a family of reversible Star-transformations which applied to a source text along with a backend compression algorithm. The basic idea of the transform module is to transform the text into some intermediate form, which can be compressed with better efficiency. The transformed text is provided to a backend compression module, which compresses the transformed text. However, execution time performance and runtime memory expenditure of these compression systems have remained high compared with the backend compression algorithms such as bzip2 and gzip.

The compression ratio achieved with compressing the transformed text is much better than compressing the text

directly with well known compression algorithms. Heap's Law [18, 5, 12], an empirical law widely accepted in information retrieval, establishes that a natural language text of O(u) words has a vocabulary of size $v = O(u^{\beta})$, for $0 < \beta < 1$. Typically, β is between 0.4 and 0.6 [28, 8, 32], and therefore v is close to $O(\sqrt{u})$. An important conclusion of this law is that for a text of O(u) words, the total number of stop-words are u* 40%. Stop-words are those words which occur frequently in the text documents. Articles, prepositions, and conjunctions are natural candidates for a list of stop-words. We have identified a list of 664 stop-words.

Our work is based on idea that a natural language text of O(u) words have u*0.4 stop-words and remaining words in a text are vocabulary. As volume of stop-words is less as compared to volume of vocabulary, we can transform all the stop-words in a text to some intermediate form and all the vocabulary in a text remains as it is.

In this paper, we used a algorithm given in [32] for transforming text into an intermediate form suitable for compression. Compared with compress, gzip and bzip2, the transform module achieves improvement in compression performance. Experimental results show that, for our test corpus, the average compression time using the transform module with bzip2, gzip and compress is 11.2% slower, 36.35% faster, 9.43% faster as compared with the original bzip2, gzip and compress respectively. The average decompression time using transform module with bzip2, gzip and compress is 42.21% slower, 15.63% faster and 7.2% faster compared with original bzip2, gzip and compress respectively.

We conducted experiments on our own test corpora. Results show that, using transform module, the average BPC (Bits per character) improved 29.28% over bzip2, 28.71% over gzip, and 14.58% over compress.

2. BASICS AND RELATED WORK

The aim of Text compression is to exploit the redundancy in the text to reduce its space requirement [40]. In this paper we denote the uncompressed file as *T* and its length in bytes as *u*. The compressed file is denoted as *Z* and its length in bytes as *n*. *Compression ratio* is used in this article to denote the size of the compressed file as a percentage of the uncompressed file (i.e., $100 \times n / u$).

Text compression is usually divided into two categories. Statistical compression is based on estimating source character probabilities and assigning them codes according to the probabilities. Dictionary methods consist in replacing text substrings by identifiers, so as to take advantage of repetitions in the text. Semi-static compression requires two passes over the text, as well as storing the model together with the compressed file. On the other hand, adaptive compression cannot start decompression at arbitrary file positions, because all the previous text must be processed so as to learn the model that permits decompressing the text that follows.

Lempel-Ziv compression is a dictionary method based on replacing text substrings by previous occurrences thereof. The two most famous algorithms of this family are called LZ77 [22] and LZ78 [23]. A well-known variant of the latter is called LZW [38]. Well-known representatives of LZ77 compression are Info-ZIP's *zip* and GNU's *gzip*. A well-known representative of LZW is Unix's *compress*. The Lempel-Ziv family is the most popular to compress text because it combines compression ratios around 35% on plain English text with fast compression and decompression. However, Lempel-Ziv compressed text cannot be decompressed at random positions, because one must process all the text from the beginning in order to learn the window that is used to decompress the desired portion.

Huffman coding [4] is designed for statistical compression. It assigns a variable-length code to each source symbol, trying to give shorter codes to more probable symbols. Huffman algorithm guarantees that the code assignment minimizes the length of the compressed file under the probabilities given by the model. A common usage of Huffman coding is to couple it with semi-static zero-order modeling, taking text characters as the source symbols and bits as the target symbols. That is, on a first pass over the text, character frequencies are collected, then Huffman codes (variable-length bit sequences) are assigned to the characters, and finally each character occurrence is replaced by its codeword in a second pass over the text. This combination, that we call "Huffman compression" for shortness, reaches the zero-order entropy of the text up to one extra bit per symbol. Being semi-static, Huffman compression permits easy decompression of the text starting at any position. Huffman compression is not very popular on natural language text because it achieves poor compression ratios compared to other techniques. However, the situation changes drastically when one uses the text *words*, rather than the characters, as the source symbols [1, 29]. The distribution of words is much more skewed than that of symbols, and this permits obtaining much better compression ratios than character-based Huffman compressors. On English text, character based Huffman obtains around 60% compression ratio, while word-based Huffman is around 25% [30]. Actually, similar compression ratios can be obtained by using Lempel-Ziv on words [15, 17, 33].

The text in natural language is not only made up of words. The text also contains punctuation, separator, and other special characters. The sequence of characters between every pair of consecutive words is called a separator. Separators must also be considered to be symbols of the source alphabet. In [1] they use the so-called separate alphabets model, where words and separators are modeled separately. As every word is followed by a separator and vice-versa, once it is known whether the text starts with a word or a separator, no further information is necessary to decode the stream of codes from the two different alphabets. Word-based Huffman compression has other advantages. Not

only the text can be compressed and decompressed efficiently, as a whole or in parts, but it is also possible to search it without decompressing, faster than when searching the uncompressed text [30]. Also, this type of compression integrates very well with information retrieval systems, because the source alphabet is equivalent to the vocabulary of the inverted index [2, 11, 14]. One of the best known systems relying on word-based Huffman in the public domain is MG system [14].

The K-th models assign a probability to each source symbol as a function of the context of *k* source symbols that precede it. They are used to build very effective compressors such as Prediction by Partial Matching (PPM) and those based on the Burrows-Wheeler Transform (BWT). PPM [16] is a statistical compressor that models the character frequencies according to the context given by the *k* characters preceding it in the text, and codes the characters according to those frequencies using arithmetic coding [13]. PPM is adaptive, so the statistics are updated as compression progresses. The larger *k*, the more accurate is the statistical model and the better the compression, but more memory and time is necessary to compress and decompress. More precisely, PPM uses *k*+1 models, of order 0 to *k*, in parallel. It usually compresses using the *k*-th order model, unless the character to compress has never been seen in that model. In this cases it switches to a lower-order model until the character is found. The BWT [27 is a reversible permutation of the text that puts together characters having the same *k*-th order context (for any *k*). Local optimization over the permuted text obtains results similar to *k*-th order compression (for example, by applying move-to-front followed by Huffman or arithmetic coding). PPM and BWT usually achieve better compression ratios than other families (around 20% on English text), yet they are much slower to compress and decompress, and cannot decompress arbitrary portions of the text collection. The representatives of this family are bzip2[21], based on the BWT, ppmdi [37] and ppmz.

Byte Pair encoding scheme [26, 31] is a universal compression algorithm that supports random access for all types of data. The global substitution process of BPE produces a uniform data format that allows decompression to begin anywhere in the data. Using BPE, data from anywhere in a compressed block can be immediately decompressed without having to start at the beginning of the block.

In [7] it was presented a new idea of spaceless words. If a word is followed by a space, we just encode the word. If not, we encode the word and then the separator. At decoding time, we decode a word and assume that a space follows, except if the next symbol corresponds to a separator. In this case the alternating property does not hold, and a single alphabet is used. They show that the spaceless word model achieves slightly better compression ratios. The number of Huffman trees for a given probability distribution is quite large. The preferred choice for most applications is the canonical trees, defined by [9]. It allows more efficiency at decoding time with less memory requirement. Many properties of canonical codes are mentioned in [6, 14, 24].

Sun *et al.* [41] introduces a transform algorithm, called Star New Transform (StarNT). Ternary Search Tree [19] is used in the transform to expedite the transform encoding. Compared with LIPT [10], the new transform achieves improvement not only in compression performance, but also in time complexity. Our work can be regarded as being an elaboration of above observation.

3. TRANSFORMATION OF TEXT

The basic initiative of the transform module is to transform the text into some intermediate form, which can be compressed with better effectiveness. The transformed text is provided to a backend data compression module which compresses the transformed text. In this section, we will first discuss stop-word-frequency distribution. After that a brief discussion about Heaps Law in Information Retrieval is presented which is followed by detailed description of the transformation algorithm.

3.1 Stop-word Frequency Distribution

A natural language text consists of vocabulary and stop-words. Stop-words are those words, which occur frequently in a text and do not contain valuable information during searching a pattern. The example of stop-words are *a, i, an, an, so, by, and, but, the, from etc*. The detailed list of stop-words is given in *Appendix A*. Our list of such words contains 664 stop-words. The stop-word frequency distribution is based on our observation that English language text has high percentage of stop-words whose length varies from four to seven. We show the stop-word frequency and length of stop-words information, as given in Fig. 1. It is clear from the figure that words of length four and five have higher frequency as compared to other words of the English language. In natural language text, the size of all different stop-words is very less but the actual volume of the space occupied by these words in text is large. This is the basis of

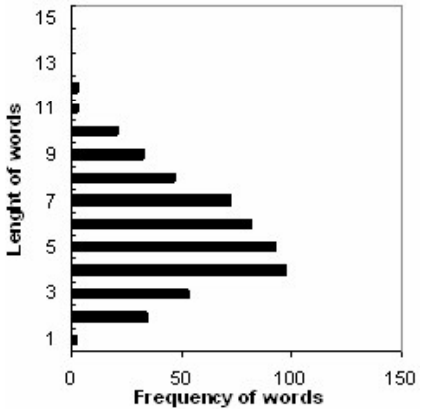

Figure 1: Stop-word Frequency Distribution

our work. We transformed the stop-word into an intermediate form which helps the backend compressor to compress well. This reason for this effectiveness is that, we are introducing an extra redundancy in the stop-words which helps the backend compressor.

3.2 Heap's Law

A natural language text consists of vocabulary and stop-words. Stop-words are frequently occurring words in a text and do not contain valuable information during searching a pattern. Our list of such words contains 664 stop-words given in Appendix A. The example of stop-words are *a, i, an, an, so, by, and, but, the, from etc*. The size of all different stop-words is very less but the actual volume of the space occupied by these words in text is large. This is an important conclusion of the Heaps law [18]. Heap's Law [18], an empirical law widely accepted in information retrieval, establishes that a natural language text of O(u) words has a vocabulary of size $v = O(u^{\beta})$, for $0< \beta <1$. Typically, β is taken as 0.5 and therefore v is close to $O(\sqrt{u})$. An important conclusion from this law is that, for a text of O (u) words, the total number of stop-words are u*40%. Stop-word includes, articles, prepositions.

Our work is based on idea that a natural language text of O (u) words has u*0.4 stop-words and remaining words in a text are vocabulary. As the total number of stop-words is less as compared to volume of vocabulary, we can transform all the stop-words in a text to some intermediate. This will lead to conversion of original text into an intermediate form which contains u*40% (transformed stop-words) and remaining u*60% of vocabulary. In this way we introduce an extra redundancy in the transformed text. In the next subsection, we will briefly discuss the transformation algorithm used [32].

3.3 Transformation Algorithm

We are using the transformation algorithm as given by [32]. Here, we briefly discuss the algorithm. "Let **A** denote a finite string (or sequence) of characters (or symbols) $a_1 a_2 a_3 \ldots\ldots a_n$ over an alphabet Σ where $a_i = \mathbf{A[i]}$ is the i^{th} character of **A**, and n is the length of the sequence **A**. **S** is a subsequence of **A** if there exists integer $1 \le r_1 < r_{2.....} < r_s \le n$ such that S[i] = $A[r_i]$, $1 \le i \le n$. Let D denote a dictionary of a set of distinct words. A transformed word corresponding to A, denoted as *A, is a sequence of n characters in which *A[i] = * if $i \ne r_i$ and for all other i, $A[i] = A[r_i]$ as in S." The set of stop-words first arranged with respect to length and with in a set of words of same length, we arrange them in lexicographic order. This makes distinct dictionaries of stop-words. For each stop-word dictionary of length one to fifteen, we make a transformed stop-word dictionary. The dictionaries used in the experiment have been prepared in advance, and shared by both the encoder and decoder. The size of dictionaries is typically around 5KB. This is a small and one-time overhead to prepare the dictionaries. Currently the transform dictionaries only contain lower-case words. Dedicated operations were designed to handle the initial letter capitalized words and all letter capitalized words. The character '~' appended to the transformed word denotes that the initial letter of the corresponding word in the original text file is capitalized. The appended character '`' denotes that all letters of the corresponding word in the original text file are capitalized. The character '\' is used as escape character for encoding the occurrence of '*', '~', '`', and '\' in the input text file.

The transformer reads a word from the input text file and checks the word into the corresponding stop-word dictionary. If the word is in stop-word dictionary, then transformer read transformed stop-word dictionary and emit the corresponding transformed word. Continuing in this way, we lead to an transformed output which contains redundant data in it. This introduced redundancy is helpful for compression algorithms.

The transform decoding module performs the inverse operations of the transform-encoding module. The escape character and special symbols ('*', '~', '`', and '\') are recognized, and transformed stop-words are replaced with their original stop-words.

4. PERFORMANCE EVALUATION

We evaluated the compression performance as well as the compression time improvement using the own test corpus, which consists of 22 files. All these test files are listed in Table 1. The experiment was carried out on a 1.6GHZ Pentium IV 256 RAM machine housing Linux 9. We have chosen bzip2, gzip and compress as a backend compression tool.

4.1 Timing Performance of Transformation

In the transform encoding module, we create the fourteen dictionaries of stop-word of length one to fifteen excluding stop-word of length fourteen as there is no word of length fourteen. These fourteen dictionaries are fixed to encoding end as well as to decoding end and since the number of stop-words are very less, so there is no need to generate it every time. The strings in the fourteen dictionaries are sorted lexicographically. The time complexity of searching a word of length *m* in an appropriate dictionary as a stop-word with *n* strings will require at most $O(\log n + m)$ time.

4.2 Timing performance with backend Compression Algorithm

The encoding/decoding time when Transform Module is combined with the backend data compression algorithms, i.e. bzip2, gzip, compress is given in Table 2 and Table 3. Following conclusions can be drawn from table 2 and table 3:

- The average compression time using the transform algorithm with bzip2, gzip and compress is 11.2% slower, 36.35 % faster, 9.43 % faster compared with the original bzip2, gzip and compress respectively.

- The average decompression time using the transform algorithm with bzip2, gzip and compress is 42.21% slower, 15.63% faster, 7.2% faster compared with the original bzip2, gzip and compress respectively

Table 1
Test Corpus

File	*Size (bytes)*
File1.txt	8833
File2.txt	48808
File3.txt	9432
File4.txt	11375
File5.txt	63826
File6.txt	18263
File7.txt	18988
File8.txt	56448
File9.txt	43460
File10.txt	39172
File11.txt	11025
File12.txt	29706
File13.txt	30726
File14.txt	2289
File15.txt	2932
File16.txt	2032
File17.txt	2968
brief.rtf	52557
project.rtf	7405686
Copying.txt	32874
Genesis	219118

Table 2
Comparison of Encoding Speed (in. ms)

	AVRG
gzip	63.16
gzip + NT	40.2
compress	103.48
compress + NT	93.72
bzip 2	550.44
bzip 2 + NT	619.8

Table 3
Comparison of Decoding Speed (in. ms)

	AVRG
gzip	45.8
gzip + NT	38.64
compress	95.36
compress + NT	88.48
bzip 2	54.8
bzip 2 + NT	94.84

4.3 Compression Performance of Transform Module

We show the compression performance (in terms of BPC) of the transform, given in figure 2. In our implementation the original stop-word dictionaries and transformed stop-word dictionaries are shared by both transform encoder and by transform decoder. These dictionaries are generated independently. The dictionaries contain 664 entries of stop-word and 664 entries of transform stop-words. The size of dictionaries is nearly 5KB. The figure 2 illustrates the comparison of average compression performance for our test corpus. The results are very clear:

- In our test corpus, facilitated with Transform Module, bzip2, gzip and compress achieves an average improvement in compression ratio of 29.28% over bzip2, 28.71% over gzip, and 19.74% over compress.
- The compression performance of bzip2 powered by Transform Module is superior to the original bzip2.

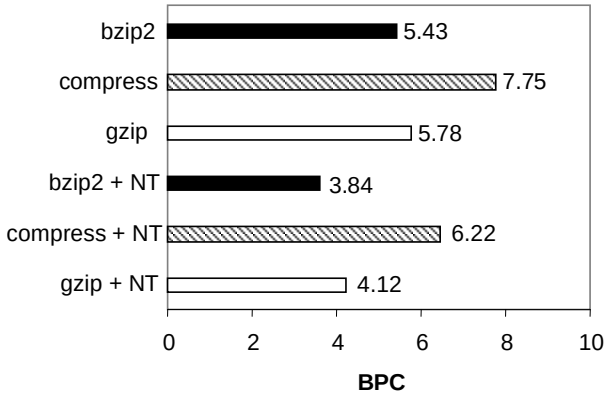

Figure 2: Compression Performance with/without Transform

5. CONCLUSIONS

In this paper, we proposed a new method for transforming the natural language text into an intermediate form which is well suited to backend compressors. We showed that the transform for stop-words are helpful for compression of natural language text documents. The percentage of stop-words is very large as compared to the vocabulary in the natural language text documents. The overhead of storing the stop-word dictionaries are very minimal and can be tolerated. The proposed scheme is helpful for better compression of natural language text documents. Initial experimental results show that the facilitated with Transform Module, bzip2, gzip and compress achieves an average improvement in compression ratio of 29.28% over bzip2, 28.71% over gzip, and 19.74% over compress.

REFERENCES

[1] A. M. Moffat. Word Based Text Compression. *Software Practice and Experience*, 19(2): 185-198, February 1989.

[2] A. Moffat and R. Wan. RE-Store: A System for Compressing, Browsing and Searching Large Documents. In *Proc. 8th Intl. Symp. On String Processing and Information Retrieval (SPIRE'01)*, pages 162-174. IEEE CS Press.

[3] A. Moffat. Implementing the PPM Data Compression Scheme. *IEEE Transactions on Communications*, 38(11): 1917-1921, 1990.

[4] D. A. Huffman. A Method for the Construction of Minimum Redundancy Codes. *Proc. of IRE*, 40: 1098-1101, 1951.

[5] D. C. van Leijenhorst and Th. P. van der Weide. A Formal Derivation of Heap's Law. *Journal of Information Science-Informatics and Computer Science*. Vol. 170:2-4, pages 263-272, 2005.

[6] D. S. Hirschberg and D. A. Lelewer. Efficient Decoding of Prefix Codes. *Commun. ACM* 33, 4 (Apr. 1990), 449-459.

[7] E. S. Moura, G. Navarro and N. Ziviani. Indexing Compressed Text. In *Proceedings of the 4th South American Workshop on String Processing*, 95-111, 1997.

[8] E. S. Moura, G. Navarro and N. Ziviani. Indexing Compressed Text. In *Proc of the 4th South American workshop on String Processing. R. Baeza-Yates*, Ed. Carleton University Press International Informatics Series, Vol. 8. Carleton University Press, Ottawa, Canada, 95-111.

[9] E. S. Schwartz and B. Kallick. Generating a Canonical Prefix Encoding. *Commu. of the ACM 7*, 166-169, 1964.

[10] F. S. Awan and A. Mukherjee. LIPT: A Lossless Text Transform to Improve Compression. In *Proceedings of International Conference on Information and Theory: Coding and Computing*, Las Vegas, Nevada, 2001, IEEE Computer Society.

[11] G. Navarro, E. Moura, M. Neubert, N. Ziviani and R. Baeza-Yates. Adding Compression to Block Addressing Inverted Indexes. *Information Retrieval*, 3(1): 49-77, 2000.

[12] H. S. Heaps. *Information Retrieval: Computational and Theoretical Aspects*, 206-208, 1978.

[13] I. H. Witten, A. Moffat and J. G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30(6): 520-541, 1987.

[14] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, Inc., second edition, 1999.

[15] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A Locally Adaptive Data Compression Scheme. *Communications of the ACM*, 29: 320–330, 1986.

[16] J. Clearly and I. Witten. Data Compression using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, COM-32(4): 396–402, April 1984.

[17] J. Dvorský, J. Pokorný, and V. Snásel. Word Based Compression Methods and Indexing for Text Retrieval Systems. *In ADBIS'99*, LNCS 1691, 75–84. Springer, 1999.

[18] J. Heaps. *Information Retrieval-Computational and Theoretical Aspects*. Academic Press, Inc., New York, NY, 1978.

[19] J. L. Bentley and R. Sedgewick. Fast Algorithms for Sorting and Searching Strings. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, New Orleans, January 1997.

[20] J. Rissannen and G. G. Langdon. Arithmetic Coding. *IBM Journal of Research and Development*, 2 3: 149-162, 1979.

[21] J. Seward. Bzip2 Program and Documentation, 1999. *http://sourceware.cygnus.com/bzip2/*

[22] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*. Vol. IT-23, No. 3, May 1977, pp. 337–343.

[23] J. Ziv and A. Lempel. Compression of Individual Sequences Via Variable Rate Coding. *IEEE Transactions on Information Theory*. Vol. IT-24, No. 5, September 1978, pp. 530–535.

[24] J. Zobel and A. Moffat. Adding Compression to a Full-Text Retrieval System. *Softw. Pract. Exper.* 25, 8 (Aug. 1995), 891-903.

[25] K. Sadakane. *Unifying Text Search and Compresion: Suffix Sorting, Block Sorting and Suffix Array*. PhD. Thesis, The university of Tokyo, December 1999.

[26] L. Robert and R. Nadarajan. New Algorithm for Random Access Text Compression. In *Proc. 3rd Intl. conf. on Information technology: New Generation (ITNG'06)*. IEEE Computer Society. 2006.

[27] M. Burrows and D. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Technical Report, SRC Research Report 124, Digital Systems Research Center, Palo Alto, CA, 1994.

[28] M. D. Araujo, G. Navaaro and N. Ziviani. Large Text Searching Allowing Errors. In *Proceedings of the 4th South American workshop on String Processing*. R. Baeza-Yates, Ed. Carleton University Press International Informatics Series, Vol. 8. Carleton University Press, Ottawa, Canada, 2-20.

[29] Moura, E., Navarro, G., Ziviani, N., and Baeza-Yates, R. (2000). Fast and Exible Word Searching on Compressed Text. *ACM Transactions on Information Systems*, 18(2): 113–139.

[30] N. Ziviani, E. Moura, G. Navarro, and R. Baeza-Yates. Compression: A Key for Next-Generation Text Retrieval Systems. *IEEE Computer*, 33(11): 37–44, November 2000.

[31] Philip Gage, "Random Access Data Compression", *The C/C++ Users Journal*, Sep. 1997.

[32] R. Franceschini and A. Mukherjee. Data Compression using Encrypted Text. In *Proceedings of the third Forum on Research and Technology*, Advances on Digital Libraries, Pages 130-138. ADL, 1996.

[33] R. Horspool and G. Cormack. Constructing Word-based Text Compression Algorithms. In *Proc. 2nd IEEE Data Compression Conference* (DCC'92), pages 62-71.

[34] R. Yugo Kartono Isal and Alistair Moffat and Alwin C. H. Ngai. Enhanced Word-Based Block-Sorting Text Compression. *Twenty-Fifth Australasian Computer Science Conference* (ACSC2002). Melbourne, Australia, 2002.

[35] Ricardo A. Baeza-Yates and Gonzalo Navarro. Block Addressing Indices for Approximate Text Retrieval. *Journal of the American Society of Information Science*, 51(1): 69-82, 2000.

[36] S. Grabowski. Text Preprocessing for Burrows-Wheeler Block Sorting Compression. VII Konferencja "Sieci i Systemy Informatyczne–teoria, projekty, wdrożenia" Łódź, październik 1999.

[37] Shkarin, D. (2002). PPM: One Step to Practicality. In Proc. *12th IEEE Data Compression Conference* (DCC 2002), p. 202–211.

[38] T. A. Welch. A Technique for High Performance Data Compression. *IEEE Computing* 17(6): 8-19, 1984.

[39] T. Bell, I. H. Witten, and J. G. Cleary. Modeling for Text Compression. *ACM Computing Surveys*, 21(4): 557–589, 1989.

[40] T. C. Bell, J. G. Cleary and I. H. Witten. *Text Compression*. Prentice Hall Inc., Upper Saddle, river, NJ, 1990.

[41] W. Sun, N. Zhang and A. Mukherjee. Dictionary Based Fast Transform for Text Compression. In *Proc. Intl Conf. on IT: Computers and Communications*. (ITCC'03). IEEE Computer Society, 2003.

Appendix A
List of StopWords

a	besides	everyone	i	miss	past	still	undoing	won't
able	best	everything	i'd	more	per	sub	unfortunately	would
about	better	everywhere	ie	moreover	perhaps	such	unless	wouldn't
above	between	ex	if	most	placed	sup	unlike	x
abroad	beyond	exactly	ignored	mostly	please	sure	unlikely	y
according	both	example	i'll	mr	plus	t	until	yes
accordingly	brief	except	i'm	mrs	possible	take	unto	yet
across	but	f	immediate	much	presumably	taken	up	you
actually	by	fairly	in	must	probably	taking	upon	you'd
adj	c	far	inasmuch	mustn't	provided	tell	upwards	you'll
after	came can	farther	inc	my	provides	tends	us	your
afterwards	cannot	few	inc.	myself	q	th	use	you're
again	cant	fewer	indeed	n	que	than	used	yours
against	can't	fifth	indicate	name	quite	thank	useful	yourself
ago	caption	first	indicated	namely	qv	thanks	uses	yourselves
ahead	cause	five	indicates	nd	r	thanx	using	you've
ain't	causes	followed	inner	near	rather	that	usually	z
all	certain	following	inside	nearly	rd	that'll	v	zero
allow	certainly	follows	insofar	necessary	re	thats	value	
allows	changes	for	instead	need	really	that's	various	
almost	clearly	forever	into	needn't	reasonably	that've	versus	
alone	c'mon	former	inward	needs	recent	the	very	
along	co	formerly	is	neither	recently	their	via	
alongside	co.	forth	isn't	never	regarding	theirs	viz	
already	com	forward	it	neverf	regardless	them	vs	
also	come	found	it'd	neverless	regards	themselves	w	
although	comes	four	it'll	nevertheless	relatively	then	want	
always	concerning	from	its	new	respectively	thence	wants	
am	consequently	further	it's	next	right	there	was	
amid	consider	furthermore	itself	nine	round	thereafter	wasn't	
amidst	considering	g	i've	ninety	s	thereby	way	
among	contain	get	j	no	said	there'd	we	
amongst	containing	gets	just	nobody	same	therefore	we'd	
an	contains	getting	k	non	saw	therein	welcome	
and	corresponding	given	keep	none	say	there'll	well	
another	could	gives	keeps	nonetheless	saying	there're	we'll	
any	couldn't	go	kept	noon	says	theres	went	
anybody	course	goes	know	no-one	second	there's	were	
anyhow	c's	going	known	nor	secondly	thereupon	we're	
anyone	currently	gone	knows	normally	see	there've	weren't	
anything	d	got	l	not	seeing	these	we've	
anyway	dare	gotten	last	nothing	seem	they	what	
anyways	daren't	greetings	lately	notwithstanding	seemed	they'd	whatever	
anywhere	definitely	h	later	novel	seeming	they'll	what'll	
apart	described	had	latter	now	seems	they're	what's	

appear	despite	hadn’t	latterly	nowhere	seen	they’ve	what’ve
appreciate	did	half	least	o	self	thing	when
appropriate	didn’t	happens	less	obviously	selves	things	whence
are	different	hardly	lest	of	sensible	think	whenever
aren’t	directly	has	let	off	sent	third	where
around	do	hasn’t	let’s	often	serious	thirty	whereafter
as	does	have	like	oh	seriously	this	whereas
a’s	doesn’t	haven’t	liked	ok	seven	thorough	whereby
aside	doing	having	likely	okay	several	thoroughly	wherein
ask	done	he	likewise	old	shall	those	where’s
asking	don’t	he’d	little	on	shan’t	though	whereupon
associated	down	he’ll	look	once	she	three	wherever
at	downwards	hello	looking	one	she’d	through	whether
available	during	help	looks	ones	she’ll	throughout	which
away	e	hence	low	one’s	she’s	thru	whichever
awfully	each	her	lower	only	should	thus	while
b	edu	here	ltd	onto	shouldn’t	till	whilst
back	eg	hereafter	m	opposite	since	to	whither
backward	eight	hereby	made	or	six	together	who
backwards	eighty	herein	mainly	other	so	too	who’d
be	either	here’s	make	others	some	took	whoever
became	else	hereupon	makes	otherwise	somebody	toward	whole
because	elsewhere	hers	many	ought	someday	towards	who’ll
become	end	herself	may	oughtn’t	somehow	tried	whom
becomes	ending	he’s	maybe	our	someone	tries	whomever
becoming	enough	hi	mayn’t	ours	something	truly	who’s
been	entirely	him	me	ourselves	sometime	try	whose
before	especially	himself	mean	out	sometimes	trying	why
beforehand	et	his	meantime	outside	somewhat	t’s	will
begin	etc	hither	meanwhile	over	somewhere	twice	willing
behind	even	hopefully	merely	overall	soon	two	wish
being	ever	how	might	own	sorry	u	with
believe	evermore	howbeit	mightn’t	p	specified	un	within
below	every	however	mine	particular	specify	under	without
beside	everybody	hundred	minus	particularly	specifying	underneath	wonder