Annual Report for Period: 06/2002 - 06/2003 **Submitted on:** 05/06/2003

Principal Investigator: Mukherjee, Amar . **Award ID:** 9977336

Organization: U of Central Florida

Title:

Algorithms to Improve the Efficiency of Data Compression and Caching on Wide-Area Networks

Project Participants

Senior Personnel

Name: Mukherjee, Amar

Worked for more than 160 Hours: Yes

Contribution to Project:

Professor Amar Mukherjee is the Principal Investigator of this project and is in charge of all the reserach and targetted activities and guidance of research assistants working under this project.

Post-doc

Graduate Student

Name: Zhang, Nan

Worked for more than 160 Hours: Yes

Contribution to Project:

Nan Zhang is working as a Graduate Research Assistant with the project. He is working on developing compression algorithms and a theory of transforms developed under this project. He is also reading literature on compressed domain search problem to come up with a formulation of a problem area for doctoral dissertation. He has been supported by this grant in the past.

Name: Motgi, Nitin

Worked for more than 160 Hours: Yes

Contribution to Project:

Nitin Motgi has been involved in the networking and infrastructure development aspects of the project. He worked on setting up an online compression utility webpage as a test bench for various comnpression algorithms and also worked on compressed data transmission infrasturucture tools. Nitin also worked on the development of new lossless compression algorithms for text. He has been supported
in this research grant during Fall of 2000. He has now accepted a job and is not working on this project.

Name: Awan, Fauzia

Worked for more than 160 Hours: Yes

Contribution to Project:

Ms. Fauzia Awan was a student in the gaduate level Multimedia Data Compression course that I taught Spring of 2000 and did a term project related to this project. Since then she got intereted doing a MS thesis under this project and has been working as a Reserach Assistant in the prtoject for one year. She defended her thesis this summer of 2001.

Name: Satya, Ravi

Worked for more than 160 Hours: No

Contribution to Project:

Mr. Ravi Vijya Satya is a graduate student who participates in this reserach project. Heis working on a paper jointly with Mr. Nan Zhang and doing 6 hrs of reserach study on data compression and pattern matching during the summer of 2002. His major interest is now Bioinformatics and DNA sequence compression and analysis.

Name: Tao, Tao

Worked for more than 160 Hours: No

Contribution to Project:

Mr Tao Tao is interested in compressed domain pattern matching for images. Since his work also involves sorted context and BWT and other transforms, he interacts with other members of the M5 Reserach group and has contributed in many discussion seminars directly related to the current project.

Name: Sun, Weifeng

Worked for more than 160 Hours: Yes

Contribution to Project:

Mr. Weifeng Sun is a new Ph. D. student supported by the Department of Computer Science here at UCF. He has joined our group and is conducting research on lossles text compression algorithms.

Undergraduate Student

Technician, Programmer

Other Participant

Name: Franceschini, Robert

Worked for more than 160 Hours: Yes

Contribution to Project:

Dr. Franceschini worked for a period of approximately three months during summer of 2000 as a Research Associate (post-doctoral) with support from matching funds under this project from the University of Central Florida. He was assisting the Graduate Reserach Assistants and was involved in producing the draft of a paper along with multiple authors for the project. In the second year, he was not involved in the project work at all, except that he serves as a member of the faculty committee for thesis done under this project. He has left UCF and accepted a job with a company.

Research Experience for Undergraduates

Organizational Partners

Other Collaborators or Contacts

I have been in touch with two well-known researchers in the data compression field: Tim Bell of Computer Science Department, University of Canterbury, New Zealand and Don Adjeroh of Departnent of Computer Science and Electrical Engineering, West Virginia University. We have been working on several joint papers on compressed domain pattern matching and submitted a new joint SMALL ITR reserach proposal to NSF for funding, and I acknowledge the partial support from this grant for these efforts. Also, we are discussing the possibility of linking up our online compression utility website vlsi.cs.ucf.edu with the Canterbury website.

Activities and Findings

Research and Education Activities:

Project Summary

The goal of this research project is to develop new lossless text compression algorithms and software tools to incorporate compression for archival storage and transmission over the Internet. The approach consists of pre-processing the text to exploit the natural redundancy of English language to obtain an intermediate transformed form via the use of a dictionary and then compressing it using existing compression algorithms. Several classical compression algorithms such as Huffman, arithmetic, LZ-family (gzip and compress) as well as some of the recent algorithms such as Bzip2, PPM family, DMC, YBS, DC, RK, PPMonstr and recent versions of Bzip2 are used as the backend compression algorithms. The performance of our transforms in combination with these algorithms are compared with the original set of algorithms, taking into account both compression, computation and storage overhead. Information theoretic explanation of experimental results are given. The impact of the research on the future of information technology is to develop data delivery systems with efficient utilization of communication bandwidth and conservation of archival storage. We also develop infrastructure software for rapid delivery of compressed data over the Internet and an online compression utility website as a test bench for comparing various kinds of compression algorithms. The site (vlsi.cs.ucf.edu) will be linked to a very well known compression website which contains the Canterbury and Calgary text corpus. The experimental research is linked to educational goals by rapid dissemination of results via reports, conference and journal papers and doctoral dissertation and master's

thesis, and transferring the research knowledge into the graduate curriculum via teaching of a graduate level course on data compression.

Goals and Objectives

The goal of this research project is to develop new lossless text compression algorithms and software tools to incorporate compression for archival storage and transmission over the Internet. Specific objectives for this period were:

À Development of new lossless text compression algorithms.

À Development of software tools to incorporate compression in text transmission over the Internet and on-line compression utility for a compression test bench.

À Measurement of performance of the algorithms taking into account both compression and communication metrics.

À Development of a theory to explain the experimental results based on information theoretic approach.

Executive Summary

The basic philosophy of our compression algorithm 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 all of them giving better compression over most of the current and classical compression algorithms (viz. Huffman, Arithmetic and Gzip (based on LZ77), Bzip2 (based on Burrows –Wheeler Transform), the class of PPM (Partial Predicate Match) algorithms (such as PPMD), RK, DC, YBS and PPMonstr). We also measured the execution times needed to produce the pre-processing and its impact on the total execution time. We developed several transforms (Star(*) and LPT) and two variations of LPT called RLPT and SCLPT. We developed four new transforms called LIPT, ILPT, LIT and NIT, which produce better results in terms of both compression ratio and execution times. 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. Typical size of dictionary for the English language is about 1 MB and can be downloaded once 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 is negligibly small. We also develop efficient data structures to expedite access to the dictionaries and propose memory management techniques using caching for use in the context of the Internet technologies. Realizing that certain on-line algorithms might prefer not to use a pre-assigned dictionary, we have been developing new algorithms to obtain the transforms dynamically with no dictionary, with small dictionaries (7947 words and 10000 words) and studying the effect of the size of the dictionaries on compression performance. We call this family of algorithms M5zip.

During this reporting period, we developed a new transform engine StarNT for a multi-corpora text compression system. StarNT achieves a superior compression ratio than almost all the other recent efforts based on BWT and PPM. StarNT is a dictionary-based fast lossless text transform. The main idea is to recode each English word with a representation of no more than three symbols. This transform not only maintains most of the original context information at the word level, but also provides some kind of 'artificial' but strong context. The transform also exploits the distribution of lengths of the words and frequency to reduce the size of the transformed text which is provided to a backend compressor. Ternary search tree is used to store the transform dictionary in the transform encoder. This data structure provides a very fast transform encoding with a low storage overhead. Another novel idea of StarNT is to treat the transformed codewords as the offset of words in the transform dictionary. Thus the time complexity of O(1) for searching a word in the dictionary is achieved in the transform decoder.

The transform encoder and transform decoder share the same dictionary, which is prepared off-line according to the following rules: 1) Most frequently used words (i.e. 312 words) are listed in the beginning of the dictionary according to their frequencies. 2) Other words are sorted according to their lengths and frequencies. Words with longer lengths are stored after words with shorter lengths. If two words have the same length, word with higher frequency of occurrence is listed after word with lower frequency. 3) To achieve better compression performance for backend data compressor, only letters [a..zA..Z] are used to represent the codeword.

One other angle of study is to adapt dynamically to domain-specific corpus (viz. biological, physics, computer science, XML documents, html documents). We experimentally measure the performance of our proposed algorithms and compare with all other algorithms using three corpuses: Calgary, Canterbury and Gutenberg corpus. Finally, we develop an information theory based explanation of the performance of our algorithms.

We make the following contributions during this phase of our work:

1. We develop a family of new lossless text compression algorithms called M5Zip which obtains the transformed version of the text

dynamically with no dictionary, with small dictionaries (7947 words and 10000 words).The transformed text is passed through a pipe of BWT transform,inversion frequency vector, run length encoding and arithmetic coding. Our results indicate that the algorithm achieves 11.65% improvement over Bzip2 and 5.95% improvement over LIPT plus Bzip2. The investigation on this class of algorithms will continue through next year. For details of the LIPT compression method, please see our annual report of last year. The details of the M5zip algorithm is given in the attachment.

2. We continue to develop our internet site (vlsi.cs.ucf.edu) as a test bed for all compression algorithms. We are now in the process of installing our new family of M5zip compression algorithms into the 'online compressiuon utility'. To use this, one has to simply clique the 'online 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 transformed based algorithms that we developed and reported in this report. 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 will be integrated with the Canterbuery website.

3. We have developed two new algorithms on compressed domain pattern matching problems directly on BWT compressed text. The BWT provides a lexicographic ordering of the input text as part of its inverse transformation process. Based on this approach, pattern matching is performed using a binary serach and then a q-gram intersection search method. The results are reprted in conference papers cited in the report.

In the 'Activities Attached File' we present detail descriptions of of the StarNT transform and experimental results.

Findings: (See PDF version submitted by PI at the end of the report)

Major Findings

The major findings for this reporting period are as follows.

We introduce StarNT, a multi-corpora text compression system, together with its transform engine StarNT. StarNT achieves a superior compression ratio than almost all the other recent efforts based on BWT and PPM. StarNT is a dictionary-based fast lossless text transform. The main idea is to recode each English word with a representation of no more than three symbols. This transform not only maintains most of the original context information at the word level, but also provides some kind of 'artificial' but strong context. The transform also exploits the distribution of lengths of the words and frequency to reduce the size of the transformed text which is provided to a backend compressor. Ternary search tree is used to store the transform dictionary in the transform encoder. This data structure provides a very fast transform encoding with a low storage overhead. Another novel idea of StarNT is to treat the transformed codewords as the offset of words in the transform dictionary. Thus the time complexity of O(1) for searching a word in the dictionary is achieved in the transform decoder.

The transform encoder and transform decoder share the same dictionary, which is prepared off-line according to the following rules: 1) Most frequently used words (i.e. 312 words) are listed in the beginning of the dictionary according to their frequencies. 2) Other words are sorted according to their lengths and frequencies. Words with longer lengths are stored after words with shorter lengths. If two words have the same length, word with higher frequency of occurrence is listed after word with lower frequency. 3) To achieve better compression performance for backend data compressor, only letters [a..zA..Z] are used to represent the codeword.

Experimental results show that the average compression time has improved by orders of magnitude compared to our previous dictionary based transform LIPT and for large files,viz. 400Kbytes or more, the compression time is no worse than those obtained by bzip2 and gzip, and is much faster than PPMD. Meanwhile, the overhead in the decompression phase is negligible. We draw a significant conclusion that bzip2 in conjunction with this transform is better than both gzip and PPMD both in time complexity and compression performance.

One of the key features of StarNT compression system is to develop domain specific dictionaries and provide tools to develop such dictionaries. Results from five corpora show that StarZip achieves an average improvement in compression performance (in terms of BPC) of 13% over bzip2 -9, 19% over gzip -9, and 10% over PPMD. Further details about the StarNT transform and experimental results can be found in the attached document.

We developed compressed domain pattern matching algorithms based on sorted context and laid the foundation of a future research proposals in this field. The papers are concerned with searching for pattern in the compressed text without or only partial decompression. The sorted context of the BWT transform gives rise to very efficient binary and q-gram based search strategy for performing exact and approximate pattern matching operations. A SMALL ITR proposal to NSF based on this reserach foundation is currently under review.

Training and Development:

Six Ph.D. students and three Masters students have participated and contributed in this research project, but not all of them received direct support from the grant. Dr. Robert Franceschini and Mr. Holger Kruse acquired valuable reseach experience working on this project and

making some early contributions. A Masters student Ms. Fauzia Awan has defended her thesis and has graduated summer of 2001. One Masters student Mr. Raja Iqbal briefly collaborated with Ms. Awan in her research. Mr. Nitin Motgi worked for about a year on the M5Zip project and then accepted a job outside. Currently, four Ph. D. students (Mr. Nan Zhang and Ravi Vijaya Satya, Mr. Tao Tao and Mr. Weifeng Sun) are working on the project. We have formed a research group called M5 Research Group which has been meeting weekly or bi-weekly to discuss reserach problems and make presentations on their work. This gives the students experience of teaching graduate level courses and seminars. The overall effect of these activities is to train graduate students with the current research on the forefront of technology. Each one of them acquired valuable experience in undertaking significant programming tasks.

Outreach Activities:

Journal Publications

Tim Bell, Don Adjeroh and Amar Mukherjee, "Pattern Matching in Compressed Text and Images", ACM Computing Survey, p. , vol. , ( ). Submitted

F. Awan and Amar Mukherjee, "LIPT: A Lossless Text Transform to Improve Compression", Proceedinds of the International Conference on Information Technology:Coding and Communication (ITCC2000), p. 452, vol. , (2001). Published

N. Motgi and Amar Mukherjee, "Network Conscious Text Compression System (NCTCSys)", Proceedings of the International Conference on Information Technology:Coding and Computing (ITCC2001), p. 440, vol. , (2001). Published

Fauzia Awan, Ron Zhang, Nitin Motgi,Raja Iqbal and Amar Mukherjee
, "LIPT: A Reversible Lossless Text Transform to Improve Compression Performance", Proc. Data Compression Conferemce, p. 311, vol. , (2001). Published

M. Powell, Tim Bell, Amar Mukherjee and Don Adjeroh, "Searching BWT Compressed Text with the Boyer-Moore Algorithm and Binary Search", Proc. Data Compression Conference (Eds. J.A.Storer and M. Cohn), p. 112, vol. , (2002). Published

D. Adjeroh, A. Mukherjee, T. Bell, M. Powell and N.Zhang, "Pattern Matching in BWT-transformed Text", Proc. Data Compression Conference, p. 445, vol. , (2002). Published

Weifeng Sun, Nan Zhang and Amar Mukherjee, "A Dictionary-Based Multi-corpora Text Compression System", Proceedings of the Data Compression Conference, p. 448, vol. , (2003). Published

Weifeng Sun, Nan Zhang and Amar Mukherjee, "Dictionary-based Fast Text Transform for Text Compresion", Proceedings International Conference on Information Technology: Coding and Computing, p. 176, vol. , (2003). Published

Books or Other One-time Publications

Amar Mukherjee and Fauzia Awan, "Text Compression", (2003). Book, Published
Editor(s): Khalid Sayood
Collection: Lossless Compression Handbook
Bibliography: 0-12-620861-1 Elsvier Science/Academic Press

Web/Internet Site

URL(s):

http://vlsi.cs.ucf.edu/

Description:

This site is for the M5 Reserach Group and the VLSI System Reserach Laboratory under the direction of Professor Amar Mukherjee. A pointer from this site leads to a site relevant to this reserach grant. There is also a pointer to our new "online compression utility".

Other Specific Products

Contributions

Contributions within Discipline:

We expect that our research will impact the future status of information technology by developing data delivery systems with efficient utilization of communication bandwidths and archival storage. We have developed new lossless text compression algorithms that have improved compression ratio over the best known existing compression algorithms which might translate into a reduction of 75% text traffic on the Internet. We have developed an online compression utility software that will allow an user to submit any text file and obtain compression statistics of all the classical and new compression algorithms. The URL for this is: vlsi.cs.ucf.edu.

Contributions to Other Disciplines:

Contributions to Human Resource Development:

So far six Ph.D. students and three Masters students have participated and contributed in this research project, but not all of them received direct support from the grant. Dr. Robert Franceschini and Mr. Holger Kruse made contributions in the project before it was officially funded by NSF. A Masters student Ms. Fauzia Awan made significant contributions and successfully defended her thesis. A Masters student Mr. Raja Iqbal worked on this project for a brief period of time and collaborated with Ms. Awan in her reserach. Mr. Nitin Motgi worked on the M5Zip project Currently, four Ph. D. students (Nan Zhang, Ravi Vijaya, Tao Tao and Weifeng Sun) are working on the project. Mr. Tao Tao who finished his Masters thesis three years ago has joined our reserach team. Other members of the M5 Research Group at the School of Electrical Engineering and Computer Science, Dr. Kunal Mukherjee and Mr.Piyush Jamkhandi, made critical comments and observation during the course of this work. The overall effect of these activities is to train graduate students with the current research on the forefront of technology.

Contributions to Resources for Research and Education:

We have taught (in the spring 2000 semester) a new course graduate level course entitled 'CAP5937:Multimedia Compression on the Internet'. The course was taught taught again spring of 2001 with a new number CAP5015. This has a new URL location: http://www.cs.ucf.edu/courses/cap5015/. This is a graduate level course and 14 students enrolled in the Spring 2000 semester and 11 students enrolled in the Spring 2001. The course was again offered Fall 2002 with 25 students. This particular topic has grown directly out of the research that we have been conducting for the last couple of years on data compression. Lecture topics have included both text and image compression,including topics from the research on the current NSF grant. The course has now being revised for next offering in Fall of 2003. The PI also delivered invited talks on research supported by this grant and in general on lossles text compression at universities in U.S. (University of California at Santa Barbara, San Diego, Riverside, Santa Cruz and Oregon State University) and abroad (Indian Institute of Technology, Kharagpur and Indian Statistical Institue , Kolkata during 2001). The PI also gave a demonstration of his work on data compression and the online compression utility web site at the IDM Workshop, 2001, Ft. Worth, Texas (April 29-30) sponsored by NSF.

Contributions Beyond Science and Engineering:

Special Requirements

Special reporting requirements: None

Change in Objectives or Scope: None

Unobligated funds: less than 20 percent of current funds

Animal, Human Subjects, Biohazards: None

Categories for which nothing is reported:

Organizational Partners

Activities and Findings: Any Outreach Activities

Any Product

Contributions: To Any Other Disciplines

Contributions: To Any Beyond Science and Engineering

Dictionary-Based Fast Transform for Text Compression

Weifeng Sun Nan Zhang Amar Mukherjee
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL. 32816
{wsun, nzhang, amar}@cs.ucf.edu

Abstract

In this paper we introduce a dictionary-based fast lossless text transform algorithm. This algorithm utilizes ternary search tree to expedite transform encoding operation. Based on an efficient dictionary mapping model, the algorithm use a fast hash function to achieve very high speed in the transform decoding phase. The experimental results show that the average compression time has improved by orders of magnitude compared to our previous dictionary based transform LIPT[4] and is only28.1% and 50.4% slower compared to bzip2 -9 , gzip –9, respectively and 21.2% faster compared to PPMD. We also achieve a superior compression ratio compared to bzip2 and its latest improvements , gzip2 and PPMD when the transform is used in conjunction with bzip2 or PPMD as a backend compressor. Meanwhile, the overhead in the decompression phase is negligible. We draw a significant conclusion that Bzip2 in conjunction with this transform is better than both gzip and PPMD both in time complexity and compression performance.

This transform is especially suitable for domain-specific lossless text compression used for archival storage and retrieval.

Key Words: Text compression, lossless transform, Ternary Search Tree

1. Introduction

In the last decade, we have seen an unprecedented explosion of textual information flow over Internet through electronic mails, web browsing, digital library and information retrieval systems. Given the continued increase in the amount of data that needs to be transferred or archived, the importance of data compression is likely to increase in the foreseeable future. It is well known that there are theoretical predictions on how far a source file can be losslessly compressed, but no existing compression approaches consistently attain these bounds over wide classes of text files.

Researchers in the field of lossless data compression have developed several sophisticated approaches, such as Huffman encoding, arithmetic encoding, the Ziv-Lempel family, Dynamic Markov Compression (DMC), Prediction by Partial Matching (PPM [1]), and Burrow-Wheeler Transform (BWT [14]) based algorithms, etc. LZ-family is dictionary based compression algorithm. Variants of LZ algorithm form the basis of UNIX compress, gzip and pkzip tools. PPM encodes each symbol by considering the preceding k (k=5) symbols (an "order k" context). PPM achieves better compression than any existing compression algorithms, but it is intolerably slow and also consumes large amount of precious memory to store context information. The family of PPM algorithms includes PPMC, PPMD, PPMZ and PPMD+ and a few others. Several efforts have been made to improve PPM [5, 11, 15, 17]. BWT arranges the symbols of a data sequence that share the same (unbounded) context by cyclic rotation followe by lexicographic sort operations. BWT utilizes move-to-front and an entropy coder at the backend to achieve compression. A number of efforts have been made to improve its efficiency [2, 3, 12, 22]

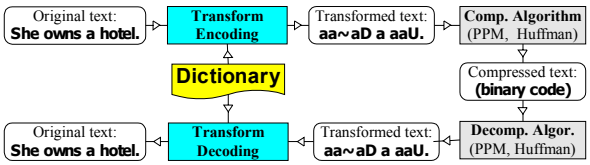
In the recent past, the M5 Data compression group, University of Central Florida (www.cs.ucf.edu) have developed a family of reversible Star transformations[19, 4] which applied to a source text along with a backend compression algorithm, achieves better compression. The transformation is designed to make it easier to compress the source file. *Figure 1* illustrates the paradigm. 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 then provided to a backend data compression module which compresses the transformed text. The decompress process is just the reverse of the compression process. First the appropriate decompression algorithm is invoked, after that the resulting text is fed into the inverse transform. In the image/video compression domains, there is an analogous paradigm that is used to compress images and video using the Fourier transform, Discrete Cosine Transform (DCT) or wavelet transforms. However, the transforms in the later case are usually lossy, some information are discarded in the transformed output

without compromising the interpretation of the image by a human.

However, execution time performance and runtime memory expenditure of our compression systems have remained high compared with other the backend compression algorithms such as bzip2 and gzip.

In this paper we introduce a fast algorithm for transform encoding module and decoding module, called *DBFT*. This algorithm utilizes *Ternary Search Tree* in the encoding module. To achieve a better

Figure 1. Text transform paradigm



performance in the transform decoding module, we also make several improvements in the dictionary mapping mechanism. Results shows that for all corpora, the average compression time using the transform algorithm with bzip2 –9 , gzip –9 and PPMD is 28.1% slower, 50.4% slower and 21.2% faster compared to the original bzip2 –9 , gzip –9 and PPMD respectively. Meanwhile, the average decompression time using the new transform algorithm with bzip2 –9 , gzip –9 and PPMD is 100% slower, 600% slower and 18.6% faster compared to the original bzip2 –9 , gzip –9 and PPMD respectively. However, since the decoding process is fairly fast, this increase is negligible.

Our compression results on text files are derived from the Canterbury, Calgary and Gutenburg corpus.

Using our transform algorithm, a better compression performance can be achieved. Facilitated with our proposed transform algorithm, bzip2 -9, gzip –9 and PPMD all achieve a better compression performance in comparison to most of the improved versions of the PPM and BWT based compression algorithms without our transform. Especially, in conjunction with bzip2, our transform algorithm achieves a better compression performance than the original PPMD. We draw a significant conclusion that Bzip2 in conjunction with DBFT is better than both gzip and PPMD both in time complexity and compression performance. It is clear that the new transform is better than LIPT not only in compression performance, but also in time complexity.

2. The Dictionary-Based Fast Transform

In this section, we will first discuss three considerations from which this algorithm is derived. After that a brief discussion about ternary search tree is presented, followed by detailed description about how the transform works.

2.1. Why Another New Transform?

There are three considerations that lead us to this transform algorithm.

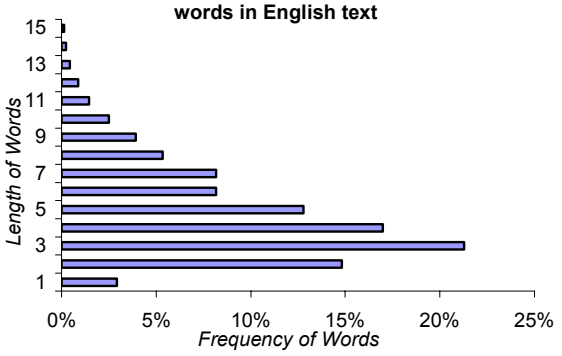
First, we gathered the data of word frequency and length of words information from Calgary, Canterbury, and Gutenburg Corpus. *Figure 2* shows the total word frequency versus the word length results for all the text files in our combined test corpus. It is clear that almost more than 82% of the words in our corpus have the lengths greater than three. If we can recode each English word with a representation with no more than three symbols, then we can achieve a certain kind of "*pre-compression*". This consideration can be implemented with a fine-tuned transform encoding algorithm, as is described later.

The second consideration is that the transformed immediate output should be more "*delicious*" to the backend compression algorithm. In other words, the transformed immediate output should maintain some of the original context information as well as providing some kind of "artificial" but strong context. The reason behind this is that BWT and PPM algorithms both predict symbols based on context, either provided by a suffix or a prefix.

In fact, the transformed codewords can be treated as offsets of words in the transform dictionary. Thus, in the transform decoding phrase we can use a hash function to achieve a fast transform. Based on this consideration, we use a continuously addressed dictionary in our algorithm to expedite the transform encoding and decoding modules. In contrast, the dictionary is split into 27 sub-blocks in LIPT. Results shows that the new transform is better than LIPT not only in time complexity but also in compression performance. This is the third consideration.

The performance of search operation in the dictionary is the key for fast transform encoding. Here there are several approaches: the first one is hash table, which is really fast, but designing a find-skewed hash function is very difficult. And unsuccessful searches using hash table are disproportionately slow. Another option is digital search tries**[Ref]**. Digital search tries are also very fast, however, they have exorbitant space requirements: suppose each node has 52-way

Figure 2: Frequency of words versus length of words in English text



branching, then one node will typically occupy 213 bytes (includes one split char and one pointer points to the transformed codeword). A three-level digital search tries with this kind of nodes will consume (1+52+52*52)*213=587,241 bytes! Or, we can use binary search tree (LIPT take this approach), which is really space efficient. Unfortunately, the search cost is usually very expensive for binary search. In this transform, we utilize ternary search tree [24] to store the dictionary. This data structure provides a very fast transform encoding speed with a low storage overhead.

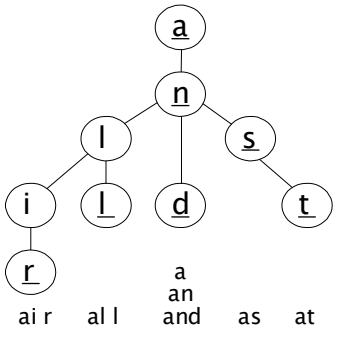
2.2. Ternary Search Tree

Ternary search trees**[Ref]** are similar with digital search tries in that strings are split in the trees with each character stored in a single node as *split char*. Besides, three pointers are included in a node: left, middle and right. All elements less than the split char are stored in the left child, those greater stored in the right child, while the middle child contains all elements with the same character.

Search operations in ternary search trees are quite straightforward: current character in the search string is compared with the *split char* at the node. If the search character is less, then go to the left child; if the search character is greater, go to the right child; otherwise, if the search character equals, just go to the middle child, and proceed to the next character in the search string. Searching for a string of length k in a ternary search tree with n strings will require at most $O(log\ n+k)$ comparisons.

Furthermore, ternary search trees are quite space-efficient. In *Figure 3*, seven strings are stored in this ternary search tree. Only nine nodes are created. If several strings have same prefix, then the corresponding nodes to these prefixes can be reused, thus memory requirements is reduced in scenarios with large amounts of data.

Figure 3: a Ternary Search Tree for seven words



In the transform encoding module, words in the dictionary are stored in the ternary search trees with the address of corresponding codewords. The ternary search tree is split into 27 distinct ternary search trees. An array is used to store the root address of these sub-ternary search trees in the root node. Words have the same starting character are stored in same sub-tree, i.e. all words starting with character 'a' in the dictionary exist in the first sub-tree, while all words start with 'b' in second sub-tree, and so on.

In the node of the ternary search tree, there is a pointer that points to the corresponding codeword. All the codeword are stored in a heap that is prepared in advance. Using this heap we can avoid storing the codeword in the node, which enables a lot of flexibility as well as space-efficiency.

There are two approaches to insert the words in the dictionary into the ternary search tree. One has the advantage of fast tree-build operation, which is especially suitable for the scenario when the size of the input text file is small; the other can offer fast search operation for transform encoding module, which is suitable for large input text file. We will discuss these issues later.

To expedite the tree-build operation, we allocate a big pool of nodes before the insert operation. Result shows that this operation is very useful.

2.3. Dictionary Mapping

The dictionary used in this experiment is prepared in advance, and shared by both the transform encoding module and the transform decoding module. Words in the dictionary ***D*** are sorted through following rules:

- Most frequently used words are listed at the head of the dictionary. Totally there are 306 words in this group.
- The leftover words are stored in ***D*** according to their lengths. Words with longer lengths are stored after words with shorter lengths.
- Words with same length are sorted according to their frequency of occurrence.

With the ordering specified as above, all words in ***D*** are assigned a corresponding codeword. The first 26 words in ***D*** are assigned "a", "b", … , "c" as their codewords. The next 26 words are assigned "A", "B", … , "Z". The 53rd word is assigned "aa", 54th "ab". Following this order, "ZZ" is assigned to the 2756th word in ***D***. The 2757th word in ***D*** is assigned "aaa" as its codeword, the following are 2758th is assigned "aab", and so on.

Using this mapping mechanism, totally 52+52*52+52*52*52 = 143,364 words can be included in ***D***. However, if the dictionary ***D*** is too large, the performance of the transform encoding/decoding module will be degraded. Currently the dictionary contains 55,951 entries.

To achieve better compression performance for backend data compression algorithm, only letters [a..zA..Z] are used to represent the codeword. This is the result of the second consideration we mentioned before.

The proposed new transform differs from our earlier Star-family transforms with respect to the meaning of the character '*'. Originally it indicated the beginning of a codeword. In our new transform, it denotes that the following word does not exist in the dictionary ***D***. The main reason for this change is to minimize the size of the transformed intermediate text file, because smaller size can decrease the encoding/decoding runtime of the backend data compression algorithm.

2.4. Transform Encoding

In the transform encoding module, the shared static dictionary is read into main memory and at the same time the corresponding ternary search tree is created. Since the length of the codewords is not uniform, we allocate a global heap to store these codewords (which could be calculated dynamically, or read with the dictionary) to minimize the memory usage.

A *replacer* is initiated to read in the input text character by character, which performs the replace operation when it recognize that the string of a certain length of input symbols (or the lower case form of this string sequence) exists in the dictionary. Then it outputs the corresponding codeword (appended with a special symbol) and continues. If the input symbol sequence does not exist in the dictionary, it will be output with a prefix escape character "*". Currently only words are stored in the dictionary, so we can use blank symbol to denote the end of word in the input text file. For future implementation, the transform module will contain sequence of all kinds of symbols to denote upper case letter, lower case letters, blank symbols, punctuations, etc..

The dictionary only stores lower-case words. So special operations were designed to handle the first-letter capitalized words and all-letter capitalized words. The character '~' at the end of an encoded word denotes that the first letter of the input text word is capitalized. The appended character '`' denotes that all the letters in the input word are capitalized. The character '\' is used as escape character for encoding the occurrences of '*', '~', '`', and '\' in the input text.

2.5. Transform Decoding

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

As in the transform encoding module, the original words in the dictionary are stored collectively in a pre-allocated heap to minimize the memory usage. And their addresses are stored in an array sequentially.

There is a very important property of the dictionary mapping mechanism that can be used to accelerate the transform decoding operation. Since the codeword in the dictionary are assigned sequentially, and the codewords contain only letters [a..zA..Z], we can think of these codewords as the addresses. In our implementation, we use a very simple hash function (with no conflict) to calculate the index of the corresponding original words in the array that stores the address of the dictionary words.

3. Performance Evaluation

Our experiments were carried out on a 360MHz Ultra Sparc-IIi Sun Microsystems machine housing SunOS 5.7 Generic_106541-04. We choose Bzip2 (-9), PPMD (order 5) and gzip (-9) as the backend compression tool. All compression results are derived from the Canterbury-Calgary and Gutenburg Corpus. The reason why we choose bzip2, gzip and PPMD as our backend compression algorithm is that bzip2 and PPM outperform other compression algorithms, and gzip is one of the most widely used compression tool.

Table 1: Comparison of Transform Encoding Speed and Transform Decoding of DBFT versus LIPT

Corpora	New Transform		LIPT	
	Transform Encoding	Transform Decoding	Transform Encoding	Transform Decoding
Calgary	0.42	0.18	1.66	1.45
Canterbury	1.26	0.85	5.7	5.56
Gutenburg	1.68	1.12	6.89	6.22
AVERAGE	0.89	0.54	3.75	3.58

Table 2: Encoding speed comparison of various compression with/without transform

File	bzip2	bzip2+DBFT	bzip2+LIPT	gzip	gzip+DBFT	Gzip+ LIPT	PPMD	PPMD+DBFT	PPMD+IPT
Calgary	0.36	0.76	1.33	0.23	0.86	1.7	9.58	7.94	9.98
Canterbury	2.73	3.04	5.22	2.46	3.36	6.59	68.3	55.7	69.2
Gutenburg	4.09	4.4	7.01	2.28	3.78	9.67	95.4	75.2	90.9
AVERAGE	1.69	2.05	3.47	1.33	2.06	4.47	41.9	33.9	41.9

3.1. Timing Performance

Table 1 illustrates the transform encoding/decoding time of our transform algorithm[I] **[Why this acronym?]** against LIPT without any backend compression

[I] From now on, we will use DBFT to denote the transform algorithm introduced in this paper.

algorithm involved. All these data are average value of 10 runs. By average time we mean the un-weighted average (simply taking the average of the transform encoding/decoding time) over the entire text corpora. Combining the three corpora and taking the average transform encoding/decoding time for all the text files, the results can be summarized as follows.

1. For all corpora, the average transform encoding time using new transform decreases nearly 76.3%. Moreover, the encoding time decreases uniformly over all files in these corpora.
2. For all corpora, the average transform decoding time using new transform decreases nearly 84.9%. Moreover, the encoding time decreases uniformly over all files in these corpora.
3. Especially, the transform encoding speed and transform decoding speed of the transform algorithm is asymmetric. The decoding module runs faster than encoding module by 39.3%. The main reason is that the hash function used in the decoding phase is more efficient than the ternary search tree in the encoding module. In LIPT the transform decoding module runs slightly faster than the encoding module because of the smaller file size processed in the decoding module.

3.2. Performance with Backend Compression Algorithm

When backend data compression algorithms such as bzip2 (-9), gzip (-9) and PPMD (order 5) are used along with the new transform, the compression system also provides a favorable timing performance. Following conclusions can be drawn form *Table 2* and *Table 3*:

1. For all corpora, the average compression time using the transform algorithm with bzip2 –9, gzip –9 and PPMD is 28.1% slower, 50.4% slower and 21.2% faster compared to the original bzip2 –9 , gzip –9 and PPMD respectively.
2. For all corpora, the average decompression time using the new transform algorithm with bzip2 –9 , gzip –9 is 2 and 6 times slower respectively, and is 18.6% faster compared to the original PPMD. However, since the decoding process is fairly fast, this increase is negligible.

It must be pointed out that when the new transform is run with bzip2, the compression time increase is imperceptible to human users, if the size of the input text file is small (less than 100Kbytes). As the size of the file increases, the relative impact the new transform introduced to bzip2 decreases proportionally.

Table 3: Decoding speed comparison of various compression with/without transform

File	bzip2	bzip2+DBFT	bzip2+ LIPT	gzip	gzip+DBFT	Gzip+ LIPT	PPMD	PPMD+ DBFT	PPMD+ LIPT
Calgary	0.13	0.33	1.66	0.04	0.27	1.64	9.65	8.07	10.9
Canterbury	0.82	1.53	6.77	0.22	1.16	9.15	71.2	57.8	77.2
Gutenburg	1.15	2.22	8.46	0.29	1.44	7.99	95.4	76.9	98.7
AVERAGE	0.51	1	4.4	0.14	0.72	5.27	43	35	46.4

Table 4: BPC comparison of new approaches based-on BWT

File	Bzip2	Mbswic [1]	bks98 [2]	best x of 2x-1[5]	Bzip2 +LIPT	Bzip2 +DBFT
bib	1.97	2.05	1.94	1.94	1.93	**1.91**
book1	2.42	2.29	2.33	**2.29**	2.31	**2.29**
book2	2.06	2.02	2.00	2.00	1.99	**1.97**
news	2.52	2.55	2.47	2.48	2.45	**2.43**
paper1	2.49	2.59	2.44	2.45	2.33	**2.30**
paper2	2.44	2.49	2.39	2.39	2.26	**2.22**
progc	2.53	2.68	2.47	2.51	**2.44**	**2.44**
progl	1.74	1.86	1.70	1.71	1.66	**1.65**
progp	1.74	1.85	**1.69**	1.71	1.72	1.70
trans	1.53	1.63	1.47	1.48	1.47	**1.24**
Average	2.14	2.20	2.09	2.10	2.06	**2.02**

Table 5: BPC comparison of new approaches based-on PPM

File	PPMD	Multi-alphabet CTW order 16 [17]	NEW [7]	PPMD +LIPT	PPMD +DBFT
Bib	1.86	1.86	1.84	1.83	**1.81**
book1	2.30	**2.22**	2.39	2.23	2.25
book2	1.96	1.92	1.97	1.91	**1.90**
news	2.35	2.36	2.37	2.31	**2.29**
paper1	2.33	2.33	2.32	2.21	**2.18**
paper2	2.32	2.27	2.33	2.17	**2.15**
progc	2.36	2.38	2.34	2.30	**2.27**
progl	1.68	1.66	1.59	1.61	**1.57**
progp	1.70	1.64	1.56	1.68	**1.65**
trans	1.47	1.43	1.38	1.41	**1.16**
Average	2.03	2.01	2.01	1.97	**1.92**

Especially, if text size is very large, the compression time using the transform for bzip2 will be faster than that without the transform. For example, for bible.txt(size 4,047,392), the runtime decreases when the transform is applied on bzip2. **[Question]**

The data in *Table 2* and *Table 3* show that the new transform works better than LIPT when they are applied with backend compression algorithms. For all corpora, the average compression time using the new transform algorithm with bzip2 –9 , gzip –9 and PPMD is 39.7%, 54.5% and 19.5% faster compared to that when LIPT is applied. Similarly, for all corpora, the average decompression time using the new transform algorithm

with bzip2 –9 , gzip –9 and PPMD is 77.3%, 86.5% and 23.9% faster respectively.

Especially, if the size of the input text file is small (less than 100Kbytes), the new transform in conjunction with bzip2 outperforms LIPT with bzip2 wonderfully. For example, for Calgary corpus (that is mainly comprised of small files), bzip2 facilitated with the new transform runs faster than bzip2 with LIPT by nearly 4 times.

4. Compression Performance of the Implementation

We compared the compression performance (in terms of BPC, bits per character) of our proposed transform with the results of the recent improvements on BWT and PPM as listed on *Table 4* and *Table 5*.

From *Table 4* and *Table 5*, it can be seen that our transform algorithm outperforms almost all the other improvements. The data in *Table 4* and *Table 5* has been taken from the references given in the respective columns.

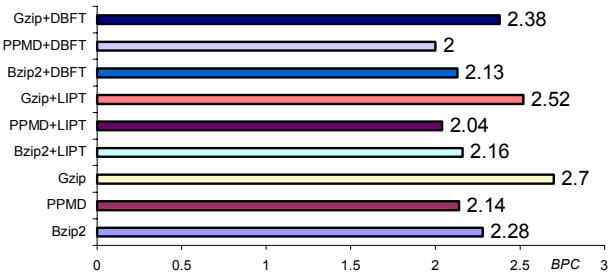
Figure 4 illustrates the comparison of average compression performance for Calgary, Canterbury and Gutenburg corpora. The results is very clear:

1. Facilitated with our proposed transform algorithm, bzip2 -9, gzip –9 and ppmd all achieve a better compression performance improvement than the original bzip2 –9 , gzip –9 and PPMD uniformly.
2. The new transform works better than LIPT when they are applied with backend compression algorithm.
3. In conjunction with Bzip2, our transform algorithm achieve a better compression performance than the original PPMD. Combined with the timing performance, we conclude that Bzip2+DBFT is better than PPMD both in time complexity and compression performance.

5. Space Complexity

In our implementation, the transform dictionary is a static dictionary shared by both transform encoder and transform decoder. The size of the off-line dictionary is nearly 0.5MB (uncompressed) and 197KB when compressed with bzip2.

Figure 4: Compression Performance with/without Transform



About the run-time overhead, since in our implementation there are 197892 nodes in the ternary search tree, the memory requirement of the dictionary in the transform encoding phase is 197892*sizeof (node)=977K bytes. In the transform decoding phase, less memory is needed.

Figure 5: Compression effectiveness versus compression speed

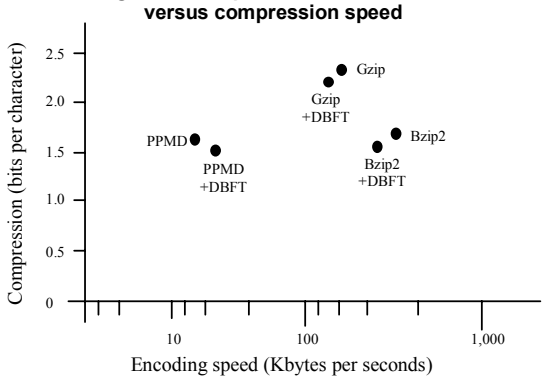
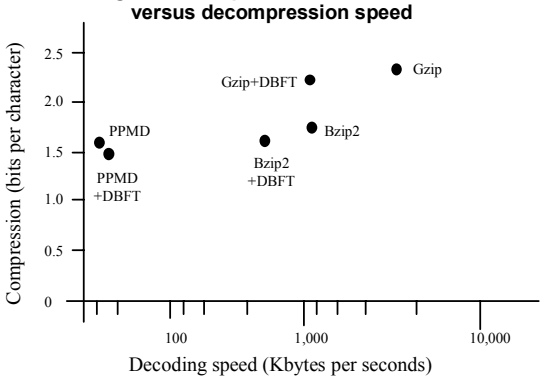


Figure 6: Compression effectiveness versus decompression speed



6. Conclusions

The order in which we insert the nodes into the ternary search tree has a lot of performance impact in the transform encoding phase. First, this order determines the time needed to construct the ternary search tree of the dictionary before performing the transform on the input text. Second, it also determines the performance of the search operation which is the key factor of the transform efficiency.

Ternary search tree is sensitive to insertion order: if we insert nodes in a good order (middle element first), we end up with a balanced tree for which the construct time is the shortest; if we insert nodes in a the order of the frequency of the words in the dictionary, then the

result would be a skinny tree that is very costly to build but efficient to search. In our experiment, results show that if we follow the first approach, the cost of inserting all words in the dictionary to the ternary search tree would be 0.19 seconds; while the second approach only needs 0.13 seconds. However, when combined with the transform replace operation, there is little difference in the total transform time for ordinary sized file. In fact, this difference is only obvious for huge files, such as bible.txt (size 4,047,392), the transform times are 4.65 seconds and 5.11 seconds for the two approaches. Our approach in the proposed transform algorithm is an eclectic one: just follow the *natural* order of the words in the dictionary. Result shows that this approach works very well.

In this paper, we have proposed a new transform algorithm which utilizes ternary search tree to expedite transform encoding operation. This transform algorithm also includes an efficient dictionary mapping mechanism based on which a fast hash function can be applied in the transform decoding phase.

We compared the compression effectiveness versus compression/decompression speed when bzip2 –9, gzip –9 and PPMD are used as backend compressor with our transform algorithm, as illustrated in *Figure 5* and *Figure 6*. It is very clear that Bzip2 + DBFT could provide a better compression performance which maintains an appealing compression and decompression speed.

Acknowledgement

We would like to thank Nan Zhang for his enormous help in preparation of this work. This research is partially supported by NSF grant number: IIS-9977336 and IIS-0207819.

.

References

[1] A. Moffat, "Implementing the PPM data Compression Scheme", *IEEE Transaction on Communications*, 38(11), pp.1917-1921, 1990

[2] B. Balkenhol, S. Kurtz , and Y. M. Shtarkov, "Modifications of the Burrows Wheeler Data Compression Algorithm", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, March 1999,pp. 188-197.

[3] B. Chapin, "Switching Between Two On-line List Update Algorithms for Higher Compression of Burrows-Wheeler Transformed Data", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, March 2000, pp. 183-191.

[4] F. Awan and A. Mukherjee, "LIPT: A Lossless Text Transform to improve compression", Proceedings of International Conference on Information and Theory : Coding and Computing, IEEE Computer Society, Las Vegas Nevada, 2001.

[5] F. Willems, Y.M. Shtarkov, and T.J.Tjalkens, "The Context-Tree Weighting Method: Basic Properties", *IEEE Transaction on Information Theory*, IT-41(3), 1995, pp. 653-664.

[6] Gutenburg Corpus: http://www.promo.net/pg/

[7] H. Kruse and A. Mukherjee, "Preprocessing Text to Improve Compression Ratios", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, 1998, pp. 556.

[8] Canterbury Corpus: http://corpus.canterbury.ac.nz

[9] I.H.Witten, A. Moffat, T. Bell, "Managing Gigabyte, Compressing and Indexing Documents and Images", 2nd Edition, Morgan Kaufmann Publishers, 1999.

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

[11] J.G. Cleary, W.J. Teahan, and Ian H. Witten, "Unbounded Length Contexts for PPM", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, March 1995, pp. 52-61.

[12] J. Seward, "On the Performance of BWT Sorting Algorithms", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, March 2000, pp. 173-182.

[13] K. Sadakane, T. Okazaki, and H. Imai, "Implementing the Context Tree Weighting Method for Text Compression", *Proceedings of Data Compression Conference,* IEEE Computer Society, Snowbird Utah, March 2000, pp. 123-132.

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

[15] M. Effros, "PPM Performance with BWT Complexity: A New Method for Lossless Data Compression", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, March 2000, pp. 203-212.

[16] N. Motgi and A. Mukherjee, "Network Conscious Text Compression System (NCTCSys)", *Proceedings of International Conference on Information and Theory: Coding and Computing*, IEEE Computer Society, Las Vegas Nevada, 2001.

[17] N.J. Larsson, "The Context Trees of Block Sorting Compression", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, 1998, pp 189- 198.

[18] P. Fenwick, "Block Sorting Text Compression", *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australia, January 31 – February 2, 1996.

[19] R. Franceschini and A. Mukherjee, "Data Compression Using Encrypted Text", *Proceedings of the third Forum on Research and Technology*, Advances on Digital Libraries, ADL 96, pp. 130-138.

[20] P.G.Howard, "The Design and Analysis of Efficient Lossless Data Compression Systems", Ph.D. thesis. Providence, RI:Brown University, 1993.

[21] R. Yugo Kartono Isal, "Enhanced Word-Based Block-Sorting Text Compression", Proc. 25th Australasian Computer Science Conference, Melbourne, January 2002, pages 129-138.

[22] Z. Arnavut, "Move-to-Front and Inversion Coding", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird, Utah, March 2000, pp. 193-202.

Dictionary-Based Fast Transform for Text Compression

Weifeng Sun Nan Zhang Amar Mukherjee
School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL. 32816
{wsun, nzhang, amar}@cs.ucf.edu

Abstract

In this paper we introduce a dictionary-based fast lossless text transform algorithm. This algorithm utilizes ternary search tree to expedite transform encoding operation. Based on an efficient dictionary mapping model, the algorithm use a fast hash function to achieve very high speed in the transform decoding phase. The experimental results show that the average compression time has improved by orders of magnitude compared to our previous dictionary based transform LIPT[4] and is only28.1% and 50.4% slower compared to bzip2 -9 , gzip –9, respectively and 21.2% faster compared to PPMD. We also achieve a superior compression ratio compared to bzip2 and its latest improvements , gzip2 and PPMD when the transform is used in conjunction with bzip2 or PPMD as a backend compressor. Meanwhile, the overhead in the decompression phase is negligible. We draw a significant conclusion that Bzip2 in conjunction with this transform is better than both gzip and PPMD both in time complexity and compression performance.

This transform is especially suitable for domain-specific lossless text compression used for archival storage and retrieval.

Key Words: Text compression, lossless transform, Ternary Search Tree

1. Introduction

In the last decade, we have seen an unprecedented explosion of textual information flow over Internet through electronic mails, web browsing, digital library and information retrieval systems. Given the continued increase in the amount of data that needs to be transferred or archived, the importance of data compression is likely to increase in the foreseeable future. It is well known that there are theoretical predictions on how far a source file can be losslessly compressed, but no existing compression approaches consistently attain these bounds over wide classes of text files.

Researchers in the field of lossless data compression have developed several sophisticated approaches, such as Huffman encoding, arithmetic encoding, the Ziv-Lempel family, Dynamic Markov Compression (DMC), Prediction by Partial Matching (PPM [1]), and Burrow-Wheeler Transform (BWT [14]) based algorithms, etc. LZ-family is dictionary based compression algorithm. Variants of LZ algorithm form the basis of UNIX compress, gzip and pkzip tools. PPM encodes each symbol by considering the preceding k (k=5) symbols (an "order k" context). PPM achieves better compression than any existing compression algorithms, but it is intolerably slow and also consumes large amount of precious memory to store context information. The family of PPM algorithms includes PPMC, PPMD, PPMZ and PPMD+ and a few others. Several efforts have been made to improve PPM [5, 11, 15, 17]. BWT arranges the symbols of a data sequence that share the same (unbounded) context by cyclic rotation followe by lexicographic sort operations. BWT utilizes move-to-front and an entropy coder at the backend to achieve compression. A number of efforts have been made to improve its efficiency [2, 3, 12, 22]

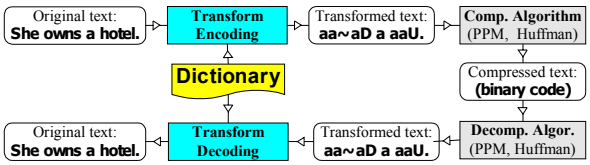
In the recent past, the M5 Data compression group, University of Central Florida (www.cs.ucf.edu) have developed a family of reversible Star transformations[19, 4] which applied to a source text along with a backend compression algorithm, achieves better compression. The transformation is designed to make it easier to compress the source file. *Figure 1* illustrates the paradigm. 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 then provided to a backend data compression module which compresses the transformed text. The decompress process is just the reverse of the compression process. First the appropriate decompression algorithm is invoked, after that the resulting text is fed into the inverse transform. In the image/video compression domains, there is an analogous paradigm that is used to compress images and video using the Fourier transform, Discrete Cosine Transform (DCT) or wavelet transforms. However, the transforms in the later case are usually lossy, some information are discarded in the transformed output

without compromising the interpretation of the image by a human.

However, execution time performance and runtime memory expenditure of our compression systems have remained high compared with other the backend compression algorithms such as bzip2 and gzip.

In this paper we introduce a fast algorithm for transform encoding module and decoding module, called *DBFT*. This algorithm utilizes *Ternary Search Tree* in the encoding module. To achieve a better

Figure 1. Text transform paradigm



performance in the transform decoding module, we also make several improvements in the dictionary mapping mechanism. Results shows that for all corpora, the average compression time using the transform algorithm with bzip2 –9 , gzip –9 and PPMD is 28.1% slower, 50.4% slower and 21.2% faster compared to the original bzip2 –9 , gzip –9 and PPMD respectively. Meanwhile, the average decompression time using the new transform algorithm with bzip2 –9 , gzip –9 and PPMD is 100% slower, 600% slower and 18.6% faster compared to the original bzip2 –9 , gzip –9 and PPMD respectively. However, since the decoding process is fairly fast, this increase is negligible.

Our compression results on text files are derived from the Canterbury, Calgary and Gutenburg corpus.

Using our transform algorithm, a better compression performance can be achieved. Facilitated with our proposed transform algorithm, bzip2 -9, gzip –9 and PPMD all achieve a better compression performance in comparison to most of the improved versions of the PPM and BWT based compression algorithms without our transform. Especially, in conjunction with bzip2, our transform algorithm achieves a better compression performance than the original PPMD. We draw a significant conclusion that Bzip2 in conjunction with DBFT is better than both gzip and PPMD both in time complexity and compression performance. It is clear that the new transform is better than LIPT not only in compression performance, but also in time complexity.

2. The Dictionary-Based Fast Transform

In this section, we will first discuss three considerations from which this algorithm is derived. After that a brief discussion about ternary search tree is presented, followed by detailed description about how the transform works.

2.1. Why Another New Transform?

There are three considerations that lead us to this transform algorithm.

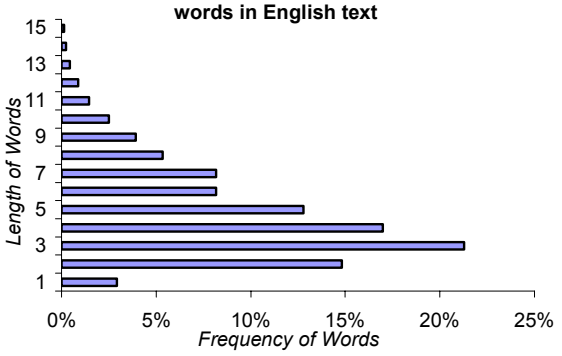
First, we gathered the data of word frequency and length of words information from Calgary, Canterbury, and Gutenburg Corpus. *Figure 2* shows the total word frequency versus the word length results for all the text files in our combined test corpus. It is clear that almost more than 82% of the words in our corpus have the lengths greater than three. If we can recode each English word with a representation with no more than three symbols, then we can achieve a certain kind of "*pre-compression*". This consideration can be implemented with a fine-tuned transform encoding algorithm, as is described later.

The second consideration is that the transformed immediate output should be more "*delicious*" to the backend compression algorithm. In other words, the transformed immediate output should maintain some of the original context information as well as providing some kind of "artificial" but strong context. The reason behind this is that BWT and PPM algorithms both predict symbols based on context, either provided by a suffix or a prefix.

In fact, the transformed codewords can be treated as offsets of words in the transform dictionary. Thus, in the transform decoding phrase we can use a hash function to achieve a fast transform. Based on this consideration, we use a continuously addressed dictionary in our algorithm to expedite the transform encoding and decoding modules. In contrast, the dictionary is split into 27 sub-blocks in LIPT. Results shows that the new transform is better than LIPT not only in time complexity but also in compression performance. This is the third consideration.

The performance of search operation in the dictionary is the key for fast transform encoding. Here there are several approaches: the first one is hash table, which is really fast, but designing a find-skewed hash function is very difficult. And unsuccessful searches using hash table are disproportionately slow. Another option is digital search tries**[Ref]**. Digital search tries are also very fast, however, they have exorbitant space requirements: suppose each node has 52-way

Figure 2: Frequency of words versus length of words in English text



branching, then one node will typically occupy 213 bytes (includes one split char and one pointer points to the transformed codeword). A three-level digital search tries with this kind of nodes will consume (1+52+52*52)*213=587,241 bytes! Or, we can use binary search tree (LIPT take this approach), which is really space efficient. Unfortunately, the search cost is usually very expensive for binary search. In this transform, we utilize ternary search tree [24] to store the dictionary. This data structure provides a very fast transform encoding speed with a low storage overhead.

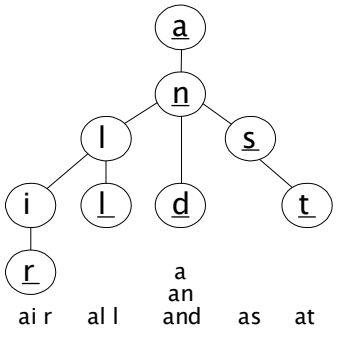
2.2. Ternary Search Tree

Ternary search trees**[Ref]** are similar with digital search tries in that strings are split in the trees with each character stored in a single node as *split char*. Besides, three pointers are included in a node: left, middle and right. All elements less than the split char are stored in the left child, those greater stored in the right child, while the middle child contains all elements with the same character.

Search operations in ternary search trees are quite straightforward: current character in the search string is compared with the *split char* at the node. If the search character is less, then go to the left child; if the search character is greater, go to the right child; otherwise, if the search character equals, just go to the middle child, and proceed to the next character in the search string. Searching for a string of length k in a ternary search tree with n strings will require at most $O(log\ n+k)$ comparisons.

Furthermore, ternary search trees are quite space-efficient. In *Figure 3*, seven strings are stored in this ternary search tree. Only nine nodes are created. If several strings have same prefix, then the corresponding nodes to these prefixes can be reused, thus memory requirements is reduced in scenarios with large amounts of data.

Figure 3: a Ternary Search Tree for seven words



In the transform encoding module, words in the dictionary are stored in the ternary search trees with the address of corresponding codewords. The ternary search tree is split into 27 distinct ternary search trees. An array is used to store the root address of these sub-ternary search trees in the root node. Words have the same starting character are stored in same sub-tree, i.e. all words starting with character 'a' in the dictionary exist in the first sub-tree, while all words start with 'b' in second sub-tree, and so on.

In the node of the ternary search tree, there is a pointer that points to the corresponding codeword. All the codeword are stored in a heap that is prepared in advance. Using this heap we can avoid storing the codeword in the node, which enables a lot of flexibility as well as space-efficiency.

There are two approaches to insert the words in the dictionary into the ternary search tree. One has the advantage of fast tree-build operation, which is especially suitable for the scenario when the size of the input text file is small; the other can offer fast search operation for transform encoding module, which is suitable for large input text file. We will discuss these issues later.

To expedite the tree-build operation, we allocate a big pool of nodes before the insert operation. Result shows that this operation is very useful.

2.3. Dictionary Mapping

The dictionary used in this experiment is prepared in advance, and shared by both the transform encoding module and the transform decoding module. Words in the dictionary ***D*** are sorted through following rules:

- Most frequently used words are listed at the head of the dictionary. Totally there are 306 words in this group.
- The leftover words are stored in ***D*** according to their lengths. Words with longer lengths are stored after words with shorter lengths.
- Words with same length are sorted according to their frequency of occurrence.

With the ordering specified as above, all words in ***D*** are assigned a corresponding codeword. The first 26 words in ***D*** are assigned "a", "b", … , "c" as their codewords. The next 26 words are assigned "A", "B", … , "Z". The 53rd word is assigned "aa", 54th "ab". Following this order, "ZZ" is assigned to the 2756th word in ***D***. The 2757th word in ***D*** is assigned "aaa" as its codeword, the following are 2758th is assigned "aab", and so on.

Using this mapping mechanism, totally 52+52*52+52*52*52 = 143,364 words can be included in ***D***. However, if the dictionary ***D*** is too large, the performance of the transform encoding/decoding module will be degraded. Currently the dictionary contains 55,951 entries.

To achieve better compression performance for backend data compression algorithm, only letters [a..zA..Z] are used to represent the codeword. This is the result of the second consideration we mentioned before.

The proposed new transform differs from our earlier Star-family transforms with respect to the meaning of the character '*'. Originally it indicated the beginning of a codeword. In our new transform, it denotes that the following word does not exist in the dictionary ***D***. The main reason for this change is to minimize the size of the transformed intermediate text file, because smaller size can decrease the encoding/decoding runtime of the backend data compression algorithm.

2.4. Transform Encoding

In the transform encoding module, the shared static dictionary is read into main memory and at the same time the corresponding ternary search tree is created. Since the length of the codewords is not uniform, we allocate a global heap to store these codewords (which could be calculated dynamically, or read with the dictionary) to minimize the memory usage.

A *replacer* is initiated to read in the input text character by character, which performs the replace operation when it recognize that the string of a certain length of input symbols (or the lower case form of this string sequence) exists in the dictionary. Then it outputs the corresponding codeword (appended with a special symbol) and continues. If the input symbol sequence does not exist in the dictionary, it will be output with a prefix escape character "*". Currently only words are stored in the dictionary, so we can use blank symbol to denote the end of word in the input text file. For future implementation, the transform module will contain sequence of all kinds of symbols to denote upper case letter, lower case letters, blank symbols, punctuations, etc..

The dictionary only stores lower-case words. So special operations were designed to handle the first-letter capitalized words and all-letter capitalized words. The character '~' at the end of an encoded word denotes that the first letter of the input text word is capitalized. The appended character '`' denotes that all the letters in the input word are capitalized. The character '\' is used as escape character for encoding the occurrences of '*', '~', '`', and '\' in the input text.

2.5. Transform Decoding

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

As in the transform encoding module, the original words in the dictionary are stored collectively in a pre-allocated heap to minimize the memory usage. And their addresses are stored in an array sequentially.

There is a very important property of the dictionary mapping mechanism that can be used to accelerate the transform decoding operation. Since the codeword in the dictionary are assigned sequentially, and the codewords contain only letters [a..zA..Z], we can think of these codewords as the addresses. In our implementation, we use a very simple hash function (with no conflict) to calculate the index of the corresponding original words in the array that stores the address of the dictionary words.

3. Performance Evaluation

Our experiments were carried out on a 360MHz Ultra Sparc-IIi Sun Microsystems machine housing SunOS 5.7 Generic_106541-04. We choose Bzip2 (-9), PPMD (order 5) and gzip (-9) as the backend compression tool. All compression results are derived from the Canterbury-Calgary and Gutenburg Corpus. The reason why we choose bzip2, gzip and PPMD as our backend compression algorithm is that bzip2 and PPM outperform other compression algorithms, and gzip is one of the most widely used compression tool.

Table 1: Comparison of Transform Encoding Speed and Transform Decoding of DBFT versus LIPT

Corpora	New Transform		LIPT	
	Transform Encoding	Transform Decoding	Transform Encoding	Transform Decoding
Calgary	0.42	0.18	1.66	1.45
Canterbury	1.26	0.85	5.7	5.56
Gutenburg	1.68	1.12	6.89	6.22
AVERAGE	0.89	0.54	3.75	3.58

Table 2: Encoding speed comparison of various compression with/without transform

File	bzip2	bzip2+DBFT	bzip2+LIPT	gzip	gzip+DBFT	Gzip+ LIPT	PPMD	PPMD+DBFT	PPMD+IPT
Calgary	0.36	0.76	1.33	0.23	0.86	1.7	9.58	7.94	9.98
Canterbury	2.73	3.04	5.22	2.46	3.36	6.59	68.3	55.7	69.2
Gutenburg	4.09	4.4	7.01	2.28	3.78	9.67	95.4	75.2	90.9
AVERAGE	1.69	2.05	3.47	1.33	2.06	4.47	41.9	33.9	41.9

3.1. Timing Performance

Table 1 illustrates the transform encoding/decoding time of our transform algorithm[I] **[Why this acronym?]** against LIPT without any backend compression

[I] From now on, we will use DBFT to denote the transform algorithm introduced in this paper.

algorithm involved. All these data are average value of 10 runs. By average time we mean the un-weighted average (simply taking the average of the transform encoding/decoding time) over the entire text corpora. Combining the three corpora and taking the average transform encoding/decoding time for all the text files, the results can be summarized as follows.

1. For all corpora, the average transform encoding time using new transform decreases nearly 76.3%. Moreover, the encoding time decreases uniformly over all files in these corpora.
2. For all corpora, the average transform decoding time using new transform decreases nearly 84.9%. Moreover, the encoding time decreases uniformly over all files in these corpora.
3. Especially, the transform encoding speed and transform decoding speed of the transform algorithm is asymmetric. The decoding module runs faster than encoding module by 39.3%. The main reason is that the hash function used in the decoding phase is more efficient than the ternary search tree in the encoding module. In LIPT the transform decoding module runs slightly faster than the encoding module because of the smaller file size processed in the decoding module.

3.2. Performance with Backend Compression Algorithm

When backend data compression algorithms such as bzip2 (-9), gzip (-9) and PPMD (order 5) are used along with the new transform, the compression system also provides a favorable timing performance. Following conclusions can be drawn form *Table 2* and *Table 3*:

1. For all corpora, the average compression time using the transform algorithm with bzip2 –9, gzip –9 and PPMD is 28.1% slower, 50.4% slower and 21.2% faster compared to the original bzip2 –9 , gzip –9 and PPMD respectively.
2. For all corpora, the average decompression time using the new transform algorithm with bzip2 –9 , gzip –9 is 2 and 6 times slower respectively, and is 18.6% faster compared to the original PPMD. However, since the decoding process is fairly fast, this increase is negligible.

It must be pointed out that when the new transform is run with bzip2, the compression time increase is imperceptible to human users, if the size of the input text file is small (less than 100Kbytes). As the size of the file increases, the relative impact the new transform introduced to bzip2 decreases proportionally.

Table 3: Decoding speed comparison of various compression with/without transform

File	bzip2	bzip2+DBFT	bzip2+ LIPT	gzip	gzip+DBFT	Gzip+ LIPT	PPMD	PPMD+ DBFT	PPMD+ LIPT
Calgary	0.13	0.33	1.66	0.04	0.27	1.64	9.65	8.07	10.9
Canterbury	0.82	1.53	6.77	0.22	1.16	9.15	71.2	57.8	77.2
Gutenburg	1.15	2.22	8.46	0.29	1.44	7.99	95.4	76.9	98.7
AVERAGE	0.51	1	4.4	0.14	0.72	5.27	43	35	46.4

Table 4: BPC comparison of new approaches based-on BWT

File	Bzip2	Mbswic [1]	bks98 [2]	best x of 2x-1[5]	Bzip2 +LIPT	Bzip2 +DBFT
bib	1.97	2.05	1.94	1.94	1.93	**1.91**
book1	2.42	2.29	2.33	**2.29**	2.31	**2.29**
book2	2.06	2.02	2.00	2.00	1.99	**1.97**
news	2.52	2.55	2.47	2.48	2.45	**2.43**
paper1	2.49	2.59	2.44	2.45	2.33	**2.30**
paper2	2.44	2.49	2.39	2.39	2.26	**2.22**
progc	2.53	2.68	2.47	2.51	**2.44**	**2.44**
progl	1.74	1.86	1.70	1.71	1.66	**1.65**
progp	1.74	1.85	**1.69**	1.71	1.72	1.70
trans	1.53	1.63	1.47	1.48	1.47	**1.24**
Average	2.14	2.20	2.09	2.10	2.06	**2.02**

Table 5: BPC comparison of new approaches based-on PPM

File	PPMD	Multi-alphabet CTW order 16 [17]	NEW [7]	PPMD +LIPT	PPMD +DBFT
Bib	1.86	1.86	1.84	1.83	**1.81**
book1	2.30	**2.22**	2.39	2.23	2.25
book2	1.96	1.92	1.97	1.91	**1.90**
news	2.35	2.36	2.37	2.31	**2.29**
paper1	2.33	2.33	2.32	2.21	**2.18**
paper2	2.32	2.27	2.33	2.17	**2.15**
progc	2.36	2.38	2.34	2.30	**2.27**
progl	1.68	1.66	1.59	1.61	**1.57**
progp	1.70	1.64	1.56	1.68	**1.65**
trans	1.47	1.43	1.38	1.41	**1.16**
Average	2.03	2.01	2.01	1.97	**1.92**

Especially, if text size is very large, the compression time using the transform for bzip2 will be faster than that without the transform. For example, for bible.txt(size 4,047,392), the runtime decreases when the transform is applied on bzip2. **[Question]**

The data in *Table 2* and *Table 3* show that the new transform works better than LIPT when they are applied with backend compression algorithms. For all corpora, the average compression time using the new transform algorithm with bzip2 –9 , gzip –9 and PPMD is 39.7%, 54.5% and 19.5% faster compared to that when LIPT is applied. Similarly, for all corpora, the average decompression time using the new transform algorithm

with bzip2 –9 , gzip –9 and PPMD is 77.3%, 86.5% and 23.9% faster respectively.

Especially, if the size of the input text file is small (less than 100Kbytes), the new transform in conjunction with bzip2 outperforms LIPT with bzip2 wonderfully. For example, for Calgary corpus (that is mainly comprised of small files), bzip2 facilitated with the new transform runs faster than bzip2 with LIPT by nearly 4 times.

4. Compression Performance of the Implementation

We compared the compression performance (in terms of BPC, bits per character) of our proposed transform with the results of the recent improvements on BWT and PPM as listed on *Table 4* and *Table 5*.

From *Table 4* and *Table 5*, it can be seen that our transform algorithm outperforms almost all the other improvements. The data in *Table 4* and *Table 5* has been taken from the references given in the respective columns.

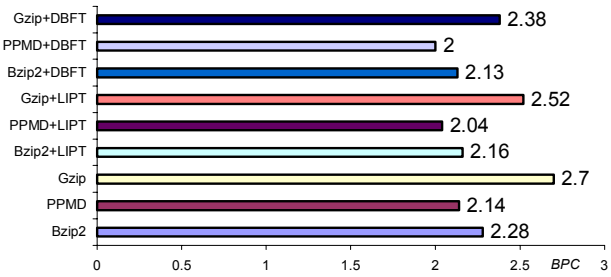
Figure 4 illustrates the comparison of average compression performance for Calgary, Canterbury and Gutenburg corpora. The results is very clear:

1. Facilitated with our proposed transform algorithm, bzip2 -9, gzip –9 and ppmd all achieve a better compression performance improvement than the original bzip2 –9 , gzip –9 and PPMD uniformly.
2. The new transform works better than LIPT when they are applied with backend compression algorithm.
3. In conjunction with Bzip2, our transform algorithm achieve a better compression performance than the original PPMD. Combined with the timing performance, we conclude that Bzip2+DBFT is better than PPMD both in time complexity and compression performance.

5. Space Complexity

In our implementation, the transform dictionary is a static dictionary shared by both transform encoder and transform decoder. The size of the off-line dictionary is nearly 0.5MB (uncompressed) and 197KB when compressed with bzip2.

Figure 4: Compression Performance with/without Transform



About the run-time overhead, since in our implementation there are 197892 nodes in the ternary search tree, the memory requirement of the dictionary in the transform encoding phase is 197892*sizeof (node)=977K bytes. In the transform decoding phase, less memory is needed.

Figure 5: Compression effectiveness versus compression speed

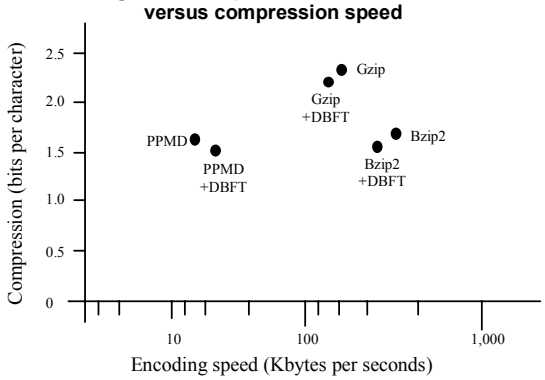
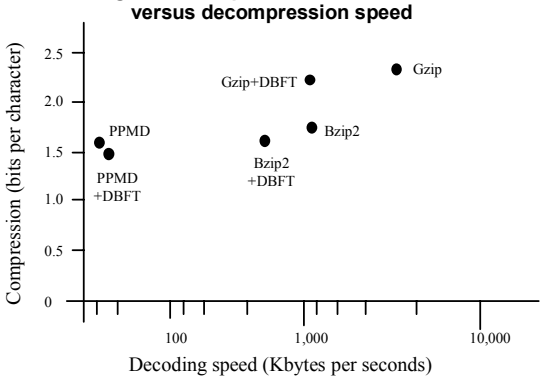


Figure 6: Compression effectiveness versus decompression speed



6. Conclusions

The order in which we insert the nodes into the ternary search tree has a lot of performance impact in the transform encoding phase. First, this order determines the time needed to construct the ternary search tree of the dictionary before performing the transform on the input text. Second, it also determines the performance of the search operation which is the key factor of the transform efficiency.

Ternary search tree is sensitive to insertion order: if we insert nodes in a good order (middle element first), we end up with a balanced tree for which the construct time is the shortest; if we insert nodes in a the order of the frequency of the words in the dictionary, then the

result would be a skinny tree that is very costly to build but efficient to search. In our experiment, results show that if we follow the first approach, the cost of inserting all words in the dictionary to the ternary search tree would be 0.19 seconds; while the second approach only needs 0.13 seconds. However, when combined with the transform replace operation, there is little difference in the total transform time for ordinary sized file. In fact, this difference is only obvious for huge files, such as bible.txt (size 4,047,392), the transform times are 4.65 seconds and 5.11 seconds for the two approaches. Our approach in the proposed transform algorithm is an eclectic one: just follow the *natural* order of the words in the dictionary. Result shows that this approach works very well.

In this paper, we have proposed a new transform algorithm which utilizes ternary search tree to expedite transform encoding operation. This transform algorithm also includes an efficient dictionary mapping mechanism based on which a fast hash function can be applied in the transform decoding phase.

We compared the compression effectiveness versus compression/decompression speed when bzip2 –9, gzip –9 and PPMD are used as backend compressor with our transform algorithm, as illustrated in *Figure 5* and *Figure 6*. It is very clear that Bzip2 + DBFT could provide a better compression performance which maintains an appealing compression and decompression speed.

Acknowledgement

We would like to thank Nan Zhang for his enormous help in preparation of this work. This research is partially supported by NSF grant number: IIS-9977336 and IIS-0207819.

.

References

[1] A. Moffat, "Implementing the PPM data Compression Scheme", *IEEE Transaction on Communications*, 38(11), pp.1917-1921, 1990

[2] B. Balkenhol, S. Kurtz , and Y. M. Shtarkov, "Modifications of the Burrows Wheeler Data Compression Algorithm", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, March 1999,pp. 188-197.

[3] B. Chapin, "Switching Between Two On-line List Update Algorithms for Higher Compression of Burrows-Wheeler Transformed Data", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, March 2000, pp. 183-191.

[4] F. Awan and A. Mukherjee, "LIPT: A Lossless Text Transform to improve compression", Proceedings of International Conference on Information and Theory : Coding and Computing, IEEE Computer Society, Las Vegas Nevada, 2001.

[5] F. Willems, Y.M. Shtarkov, and T.J.Tjalkens, "The Context-Tree Weighting Method: Basic Properties", *IEEE Transaction on Information Theory*, IT-41(3), 1995, pp. 653-664.

[6] Gutenburg Corpus: http://www.promo.net/pg/

[7] H. Kruse and A. Mukherjee, "Preprocessing Text to Improve Compression Ratios", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, 1998, pp. 556.

[8] Canterbury Corpus: http://corpus.canterbury.ac.nz

[9] I.H.Witten, A. Moffat, T. Bell, "Managing Gigabyte, Compressing and Indexing Documents and Images", 2nd Edition, Morgan Kaufmann Publishers, 1999.

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

[11] J.G. Cleary, W.J. Teahan, and Ian H. Witten, "Unbounded Length Contexts for PPM", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, March 1995, pp. 52-61.

[12] J. Seward, "On the Performance of BWT Sorting Algorithms", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, March 2000, pp. 173-182.

[13] K. Sadakane, T. Okazaki, and H. Imai, "Implementing the Context Tree Weighting Method for Text Compression", *Proceedings of Data Compression Conference,* IEEE Computer Society, Snowbird Utah, March 2000, pp. 123-132.

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

[15] M. Effros, "PPM Performance with BWT Complexity: A New Method for Lossless Data Compression", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, March 2000, pp. 203-212.

[16] N. Motgi and A. Mukherjee, "Network Conscious Text Compression System (NCTCSys)", *Proceedings of International Conference on Information and Theory: Coding and Computing*, IEEE Computer Society, Las Vegas Nevada, 2001.

[17] N.J. Larsson, "The Context Trees of Block Sorting Compression", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird Utah, 1998, pp 189- 198.

[18] P. Fenwick, "Block Sorting Text Compression", *Proceedings of the 19th Australian Computer Science Conference*, Melbourne, Australia, January 31 – February 2, 1996.

[19] R. Franceschini and A. Mukherjee, "Data Compression Using Encrypted Text", *Proceedings of the third Forum on Research and Technology*, Advances on Digital Libraries, ADL 96, pp. 130-138.

[20] P.G.Howard, "The Design and Analysis of Efficient Lossless Data Compression Systems", Ph.D. thesis. Providence, RI:Brown University, 1993.

[21] R. Yugo Kartono Isal, "Enhanced Word-Based Block-Sorting Text Compression", Proc. 25th Australasian Computer Science Conference, Melbourne, January 2002, pages 129-138.

[22] Z. Arnavut, "Move-to-Front and Inversion Coding", *Proceedings of Data Compression Conference*, IEEE Computer Society, Snowbird, Utah, March 2000, pp. 193-202.