Data Compression Using Encrypted Text

Robert Franceschini

Department of Computer Science
University of Central Florida
Orlando, FL 32816
francesc@cs.ucf.edu

Amar Mukherjee

Department of Computer Science
University of Central Florida
Orlando, FL 32816
amar@cs.ucf.edu

Abstract

In this paper, we present a new algorithm for text compression. The basic idea of our algorithm is to define a unique *encryption* or *signature* of each word in the dictionary by replacing certain characters in the words by a special character '*' and retaining a few characters so that the word is still retrievable. For any encrypted text the most frequently used character is '*' and the standard compression algorithms can exploit this redundancy in an effective way. We advocate the following compression paradigm in this paper: Given a compression algorithm A and a text T, we apply the same algorithm A on an encrypted text $*T$ and retrieve the original text via a dictionary which maps the decompressed text $*T$ to the original text T. We report better results for most widely used compression algorithms such as Huffman, LZW, arithmetic, unix compress, gnu-zip with respect to a text corpus. The compression rates using these algorithms are much better than the dictionary based methods reported in the literature.

One basic assumption of our algorithm is that the system has access to a dictionary of words used in all the texts along with a corresponding "cryptic" dictionary. The cost of this dictionary is amortized over the compression savings for all the text files handled by the organization. If two organizations wish to exchange information using our compression algorithm, they must share a common dictionary. We compare our methods with other dictionary based methods and present future research problems.

Keywords: compression, decompression, encryption, dictionary methods.

1 Introduction

The primary objective of data compression algorithms is to reduce the redundancy in data representation in order to decrease data storage requirement. Data compression also offers an attractive approach to reduce the communication cost by effectively utilizing the available bandwidth in the data links. In the nineties, we have seen an unprecedented explosion of digital data on the nation's information superhighways. This data represents a variety of objects from the multimedia spectrum such as text, images, video, sound, computer programs, graphs, charts, maps, tables, mathematical equations etc. NSF, ARPA and NASA, under the grand challenge Digital Libraries Initiative(DLI), have funded several research projects whose goal is to "advance the means to collect, store, and organize information in digital forms, and make it available for searching, retrieval and processing via communication networks" [NC95].

This paper presents a new algorithm for lossless compression of textual objects which constitute a significant portion of this information. We propose a new approach towards compression of text data based on a cryptic representation of text.

One basic assumption of our algorithm is that the system has access to a dictionary of words used in all the texts along with a corresponding "cryptic" dictionary to be defined soon. The cost of this dictionary has to be amortized over the compression savings for all the text files handled by the organization. For example, if a library or a newspaper organization or a publishing house were to use our algorithm, the availability of a in-house dictionary like the Webster's English dictionary will be a one-time investment in storage overhead. If two organizations wish to exchange information using our compression algorithm, they must share a common dictionary. This gives rise to an interesting question of developing standard word dictionaries for different application domains, which we do not address in this paper. The size of typical English dictionary is about 64,000 words requiring about 1.1 Mbytes of storage for both the real and the cryptic dictionaries combined. This a small investment considering the fact that most workstations and PC's of today have of the order of 2 to 4 gigabytes of disk space. The amortization cost will indeed be a negligible fraction if the dictionaries are placed in a centralized server for a given organization. Furthermore, the encryption scheme could be kept secret enhancing document security.

1.1 Technical Approach

Neuroscientists have identified at least two activities involved in recognizing an object by a human brain [Ro62]:

1. *Encoding* the salient features of the object in memory. This may use a new representation of the object, possibly a *signature* of the object in some compressed form in the memory.

2. *Retrieving* the encoded data when presented with the same object or when a contextual reference is made to the object.

Any search and retrieval system must embody the above two components. We may attempt to emulate these activities in an automated text search and retrieval system by compressing the text using any one of the lossless compression algorithms (viz. Huffman [Hu52], LZ [LZ77], arithmetic [RiLa79] and others [BCW90, WMB94, St88]) and by doing the search using content-based pattern search. The question is whether we can develop a better signature of the text before compression so that the compressed signature uses less storage than the original compressed text. This indeed is possible as our experimental results confirm. The English language has 52 characters (counting capital letters) and about a dozen special characters for punctuation marks etc. The word length range from 1 to 22 letters. Thus, potentially there are trillions and trillions of word combinations of which human civilization has used only a tiny fraction of about 100,000 words. Our compression paradigm attempts to exploit this redundancy by advocating the following approach: Given a compression algorithm A and a text T, we will apply the same algorithm A on an encrypted text $*T$ and retrieve the original text via a dictionary which maps the decompressed text $*T$ to the original text T. The intermediate text $*T$ compresses better because its encryption exploits the redundancy of the language. We call this algorithm $*A$. The main result of this paper is to show that there exists $*A$ algorithm(s) that gives better compression rate compared to all well-known compression algorithms such as Huffman, adaptive arithmetic, LZW, unix compress, gnu-zip-1, gnu-zip-9 and dictionary methods reported earlier in the literature [1]. We show this experimentally with respect to a text corpus. A detailed discussion of the performance of our algorithm is presented in Section 2.3.

1.2 Encoding Based on Cryptic Signatures

When text is presented to an experienced human reader, she or he reads it not by sequentially spelling each word but by recognizing each word as an unique symbol representing a collection of juxtaposed phonemes. This recognition process is robust in the sense that a human reader can allow for a lot of spelling errors or approximations. Thus, given some prior knowledge, the meaning of the following badly spelled sentence is clear: *We cell no oine before ids tyme.* Making a computer model of this kind of fuzzy association is difficult. A deterministic situation holds if it is possible to replace certain characters in a word by a special place holder character and retain a few key characters so that the word is still retrievable. Consider the set of 6-letter words starting with the letter *p* and ending with the letter *t* in English: *packet, palest, pallet, papist, parent, parrot, patent, peanut, pellet, penult, picket, pignut, piquet, pocket, precut.* Denoting an arbitrary character by a special symbol '$*$', the above set of words can be unambiguously spelled as *pa*k*t, p***st, pal**t, p*p**t, p*re*t, p***ot, p*t**t, p*a*ut, pel**t, p**u*t, pic**t, p*g**t, p*q**t, poc**t, pr***t.* An unambiguous representation of a word by a partial sequence of letters from the original sequence of letters in the word interposed by special characters '$*$' as place holders will be called a *signature of the word.* We use the place holders to retain the length information in the words. The collection of English words in a dictionary in the form of a lexicographic listing of signatures will be called a *cryptic dictionary* and an English text completely transformed using signatures from the cryptic dictionary will be called a *cryptic text.* It is important to note that for any cryptic text, the most frequently used character will likely be '$*$' and any kind of tree-based encoding scheme such as Huffman code will assign a short code for '$*$'. Other encoding schemes are also able to use such redundancy.

Some of the research issues that arise are theoretical while others relate to algorithm development. These include: How can one obtain an *optimal* cryptic dictionary for English? By optimal we mean a dictionary

[1]At the time of writing this paper, we have not been able to obtain source code for ppm and dmc algorithms to make comparative measurements

that maximizes the use of the special character '*' for a given set of application texts. The optimal solution will of course depend on the frequency of usage of letters and words in the text. Assuming static uniform probabilities of all words (as a first approximation to the optimal solution), the problem can be identified as the *Distinct Shortest Subsequence (DSS)* problem defined as follows.

Let A denote a finite string (or sequence) of characters (or symbols) $a_1a_2\ldots a_n$ over an alphabet Σ where $a_i = A[i]$ is the *ith* character of A, and n is the length of the sequence A. S is a subsequence of A if there exists integers $1 \leq r_1 < r_2 \ldots < r_s \leq n$ such that $S[i] = A[r_i], 1 \leq i \leq n$. Let D denote a *dictionary* of a set of distinct words. A *cryptic* word corresponding to A, denoted as $*A$, is a sequence of n characters in which $*A[i] = *$ if $i \neq r_i$ and for all other i, $*A[i] = A[r_i]$ as in S. Given the dictionary D, the *Distinct Shortest Subsequence* problem is the problem of determining a *cryptic dictionary*, $*D$ such that each word in $*D$ has maximal number of '*' symbols, has a distinct spelling, and is a cryptic word in one-to-one correspondence to a word in the original dictionary D. Several variations of the *DSS* problem can be defined such as maximize the number of $*$'s in D but the words themselves may not have maximal number of $*$'s; maximize the *weighted* number of $*$'s, that is, the more frequently occurring words in the language should be given more weights to have more $*$'s. Another variation is to decompose the dictionary D into a set of disjoint dictionaries D_i each containing words of length $i, i = 1, 2, \ldots, n$ and solve the DSS problem for each D_i.

1.3 Compression of Encrypted Text

The compression process consists of two steps:

1. Encrypt the text file T using the dictionary D and the cryptic dictionary $*D$, producing the cryptic text $*T$.

2. Compress the cryptic text file using the encoder of an algorithm A. Different models for the text can be used. Call the output compressed file $C(*T)$.

The decompression process consists of

1. Decompress $C(*T)$ using the decoder part of algorithm A to retrieve the encrypted text $*T$.

2. Decrypt the cryptic text file using the dictionaries, producing the original text.

We will now describe details of each of the steps outlined above.

1.3.1 Construction of the Cryptic Dictionary

There are many ways to construct a cryptic dictionary. We investigated two variations: a cryptic dictionary based on the *DSS* approach, and a cryptic dictionary based on the frequency of words in the dictionary.

For the first variation, we partitioned the dictionary D into n dictionaries $D_i, (1 \leq i \leq n)$, where n is the length of the longest word in the dictionary such that D_i contains the words of length i. Within each D_i, we computed a signature for each word as follows. For each word w in D_i, we determine whether w has a single letter (say a) that is unique in its position (say j) throughout the words of D_i. If so, then the signature of w is computed as $*w = **\ldots*a**\ldots*$ where there are $j-1$ *'s before a and $|w|-j+1$ *'s after a. Once the signature of w is computed, w is removed from continued processing for $*D_i$. Processing continues by considering pairs of letters, triples of letters, etc., of words from D_i to find unique signatures for the words until signatures have been found for all words. Note that this process terminates, since the word itself is a valid signature (however, this signature will not improve the compression of files with such words). Our implementation used a modification of a recurrence relation for generating combinations of n things taken r at a time, viz., $C(n,r) = C(n-1,r-1)+C(n-1,r)$; the modification terminated the algorithm when all signatures had been found.

The second variation was motivated by observations of the cryptic dictionary and information about our experimental text domain. As described below, one characteristic of our experimental text domain is that the most frequently occurring words in texts have lengths less than five. The first variation of the cryptic dictionary has few if any $*$'s in the signatures of such words; for example, there were 12 $*$'s among the 35 words of length two, and many words of lengths two, three, and four used no $*$'s. The second variation was an attempt to optimize the cryptic dictionary by ensuring that many more $*$'s were used in the small length words. We exploited information about the number of words of a given length and the frequency of words in text as follows. As with the first variation, we partitioned D into $D_i, (1 \leq i \leq n)$. Within each D_i we sorted the words in descending order based on available frequency information. We then assigned a signature to each word based on its location in the ordering. The first word received a signature consisting of i $*$'s. The next $52i$ words received a signature consisting of a single letter (either lower case or upper case) in a unique position, surrounded by $i-1$ $*$'s. For example, the second word of length five received the signature $a****$. The next $52 \times 52 \times C(i, 2)$ words received a signature consisting of two letters in unique positions as a pair surrounded by $i - 2$ $*$'s (where $C(i, 2)$ represents the number of ways of choosing two positions from i positions). For example, one five-letter word received a signature of $b*D**$. It was never necessary to use more than two letters for any signature in the dictionary using this scheme, although it should be clear how to continue the pattern for three, four, etc., letters. In this cryptic dictionary, there were 36 $*$'s among the 35 words of length two, a significant improvement over the first variation.

The Table 1 illustrates the encryption of several words from the dictionary.

1.4 Encryption/Decryption Process

To encrypt a text file, we read a word from the file, find it in the dictionary, obtain its signature in the cryptic dictionary, and then output the signature. Other characters from the text file (for example, spaces) are not changed.

There are a few cases that need special attention when using the above algorithm. Punctuation marks and spaces are handled by copying them from the input to the output directly; these non-letter characters are used as word-delimiters. Further improvements in this algorithm might address encryption of the spaces in the input file, especially when more than one space separates words in the file.

Capital letters within words are trickier. Since the dictionary contains only lowercase letters, we will not automatically recognize words with capitals that are in the dictionary. A naive algorithm simply copies the unrecognized word to the encrypted file. A better approach is to append the capitalization information to the end of the encrypted word in the encrypted file. We do this by appending a special character (‘) to the end of the word and then appending a bit mask in which bit i is set if and only if position i is a capital in the original word. Since we are dealing with English text, we can make an optimization to improve performance, as follows. The most likely capitalization patterns are initial capitalization of the word (for example, at the beginning of a sentence) and all capitals. Instead of appending the bit-pattern, we append ˜ or ˆ to the end of the word to handle these cases. This saves us the storage of the bit patterns in the most common cases, which reduces the requirements by one to three bytes for a word with capital letters.

Finally, we used several special characters in our encryption: *, ‘, ˜, and ˆ. If these characters appear in the input file, we prepend an escape character (\) to them in the encrypted file. Note that this adds one final special character to our encryption (namely \), which we handle in the same way as the other special characters.

To decrypt a text file, we read a signature from the encrypted file, look it up in the cryptic dictionary, obtain the corresponding word in the dictionary, and output the word. Again, other characters from the text file are not changed.

Our implementation of this process uses the same program for encryption and decryption; the only difference between the two processes is the order in which the dictionaries are specified on the command line. To encrypt a file, the dictionary is given, followed by the cryptic dictionary. To decrypt a file, the cryptic dictionary is given, followed by the dictionary. The processing is the same in both cases.

2 Experiments

This section summarizes the experiments that we have performed so far using the encryption approach. These are preliminary experiments, but we believe that they demonstrate the significant potential of this approach. We used ten text files as listed in Table 2 along with a brief description of their contents and their sizes to test our algorithm. These ten text files were based on publicly available electronic English novels, obtained from the World Wide Web. We would like to acknowledge the individuals and organizations who collected these electronic versions online, including Professor Eugene F. Irey (University of Colorado at Boulder) and Project Gutenberg and individuals responsible for the Calgary corpus.

2.1 Dictionary

We used an electronic version of an English dictionary for our work. This dictionary contained nearly 60,000 words of up to 21 letters long. For frequencies of words in English text, we referred to [HoCo92] and used information about the most frequent 100 words. In English, the most frequent words are less than five letters long.

2.2 Implementation Results

In this section, we present the results of implementation of compression algorithms on several benchmark text databases. Since the encrypted text has '*' as the most frequently occurring character and occupies approximately 60 to 70% of the information,

Proceedings of Advanced Digital Libraries 1996 (ADL '96)
0-8186-7402-4/96 $10.00 © 1996 IEEE

Dictionary Word	have	join	ounce	modify	interdenominational
Signature	e***	qC**	Tb***	q*j***	e******************

Table 1: Examples of encrypted words

Name	*Size (bytes)*	*Description*
book1	768771	*Calgary Corpus book1*
book2	610856	*Calgary Corpus book2*
news	377109	*Calgary Corpus news*
paper1	53161	*Calgary Corpus paper1*
paper2	82199	*Calgary Corpus paper2*
twocity	760697	*A Tale of Two Cities*
dracula	863326	*Dracula*
ivanhoe	1135308	*Ivanhoe*
mobydick	987597	*Moby Dick*
franken	427990	*Frankenstein*

Table 2: Test corpus

we expect the standard compression algorithms should produce much better compression. It is of interest to note that none of the words of length greater than 4 in the dictionary required more than two letters of the alphabet for encryption.

We ran experiments on our test corpus. Each of our experiments considered a different compression algorithm augmented with our encryption approach. The compression algorithms used were Unix compress (compress), GNU zip with minimal compression (gzip -1), GNU zip with maximal compression (gzip -9), and arithmetic (arithmetic) using a character based model. The results of these experiments are summarized in the following tables. It is well known that these algorithms beat the Huffman code in compression performance, so we do not report results for Huffman code in this paper. The compression is expressed as *BPC* (bits per character) and also as a percentage remaining with respect to the original size of the file. To begin, we compared the compression obtained on the cryptic dictionaries to the compression obtained on the original dictionary. These results are shown in Table 3.

Note that this implies that there is at least one English sentence that will compress to 23% of its original size requiring only 1.87 *BPC*; namely, the sentence that begins *The English words are ...* and proceeds to list the contents of the dictionary.

Tables given below show the comparative compression results for the different experiments we ran. Notice that all of the compression methods cluster around the same percentage of the file size, as compared to the original file size. However, when we examine the data for the compressed files more closely we see some interesting trends. First, notice that all of the encrypted compressions yield uniformly better results than the unaided compressions. Second, notice that the encryption based on the second variation cryptic dictionary dramatically, and consistently, outperforms all other methods.

2.3 Performance Comparison

Our method uses a full dictionary and amortizes its cost over all files handled. A question naturally arises whether our approach is better than a straight word substitution (i.e. replace each word with a unique number or Huffman code). Does our algorithm compare with other static dictionary based algorithms? Will the compression rate of LZ-algorithms be better if these algorithms had the facility of using a static dictionary?

Let us consider the idea of replacing each word in the dictionary by a unique number. If our dictionary size is 64,000 words (which is typical of most commonly available dictionaries), it will take a 16-bit address for each word. This will require 16, 8, 5.1, 4, 3.2, 2.7 and 2.3 *BPC* for 1,2,3,4,5,6 and 7 character words, respectively. Since most commonly used words use 3

	File Size (bytes)	Compressed file size (as % of original)	*BPC*
Original file	557537		
compress	250941	45%	3.60
gzip-1	249128	45%	3.57
gzip-9	223893	40%	3.21
arithmetic	299289	54%	4.29
∗compress	175061	31%	2.51
∗gzip-1	149857	27%	2.15
∗gzip-9	130402	23%	1.87
∗arithmetic	167550	30%	2.40

Table 3: Compression of the Dictionary

Name	*compress*	%	*BPC*	∗-*compress*	%	*BPC*
book1	332056	43%	3.46	289351	38%	3.01
book2	250759	41%	3.28	226025	37%	2.96
news	182121	48%	3.86	173750	46%	3.69
paper1	25077	47%	3.77	22387	42%	3.37
paper2	36161	44%	3.52	31140	38%	3.03
twocity	312035	41%	3.28	271531	36%	2.86
dracula	361093	42%	3.35	322468	37%	2.99
ivanhoe	474106	42%	3.34	416997	37%	2.94
mobydick	427160	43%	3.46	372697	38%	3.02
franken	170423	40%	3.19	148235	35%	2.77

Table 4: Comparison of compress and ∗-compress

Name	*gzip -1*	%	*BPC*	∗-*gzip -1*	%	*BPC*
book1	365005	47%	3.80	344424	45%	3.58
book2	248846	41%	3.26	240475	39%	3.15
news	164199	44%	3.48	160037	42%	3.40
paper1	21612	41%	3.25	21016	40%	3.16
paper2	35078	43%	3.41	33560	41%	3.27
twocity	344184	45%	3.62	323192	42%	3.40
dracula	399816	46%	3.70	383370	44%	3.55
ivanhoe	522367	46%	3.68	493530	43%	3.48
mobydick	466308	47%	3.78	437708	44%	3.55
franken	194794	46%	3.64	182814	43%	3.42

Table 5: Comparison of gzip -1 and ∗-gzip -1

Name	*gzip -9*	%	*BPC*	∗-*gzip -9*	%	*BPC*
book1	312281	41%	3.25	282783	37%	2.94
book2	206158	34%	2.70	191494	31%	2.51
news	144400	38%	3.06	138012	37%	2.93
paper1	18543	35%	2.79	17165	32%	2.58
paper2	29667	36%	2.89	26751	33%	2.60
twocity	290231	38%	3.05	260350	34%	2.74
dracula	337400	39%	3.13	310162	36%	2.87
ivanhoe	441053	39%	3.11	400063	35%	2.82
mobydick	398657	40%	3.23	358523	36%	2.90
franken	162829	38%	3.04	146022	34%	2.73

Table 6: Comparison of gzip -9 and ∗-gzip -9

Name	*arithmetic*	%	*BPC*	∗-*arithmetic*	%	*BPC*
book1	440007	57%	4.58	335451	44%	3.49
book2	369989	61%	4.85	290427	48%	3.80
news	247188	66%	5.24	224917	60%	4.77
paper1	33676	63%	5.07	27546	52%	4.15
paper2	48035	58%	4.67	36239	44%	3.53
twocity	429222	56%	4.51	318759	42%	3.35
dracula	477288	55%	4.42	365973	42%	3.39
ivanhoe	646114	57%	4.55	500921	44%	3.53
mobydick	555678	56%	4.50	414426	42%	3.36
franken	238357	56%	4.46	171348	40%	3.20

Table 7: Comparison of arithmetic and ∗-arithmetic

to 6 characters, the average for English text is going to be much higher than 2.5 *BPC*. Of course, it is always possible find an exceptional piece of text with long and bombastic words to give some advantage. The results reported in [[St88]] on static dictionary based algorithm confirm this observation by showing that the compression rates for the text corpus are not as good as reported in this paper. If we use unique Huffman code for each word, there will be large gaps in the address space if we are using the code itself as an index to the dictionary. The alternative is to use another intermediate dictionary that will translate the variable length Huffman codes to fixed size addresses for the words in the dictionary which doubles the storage requirements. The other problem is the construction of the Huffman codes which needs statistics of frequency of use of all the words, not necessarily the most frequently used word, for the entire corpus.

An interesting dictionary based method has been proposed by Hirschberg and Lelewer [HL90] based on the *numerical sequence property* of canonic Huffman codes [SK64]. Their method, however, assumes the transmission of the entire dictionary as a stream of characters which is not a practical idea. Even if we assume a copy of the dictionary is available to the decoder there is still a problem. Addressing is done at the byte level requiring more than 16 bits per address making the situation worse for the transmission of the dictionary. Their method, however, does not produce compression better than that achievable by the Huffman method. Their emphasis was to produce a fast decoder using as few resources as possible; it does not attempt to reduce the compression rate below what is possible using the Huffman method.

The LZ algorithms are also dictionary based but the difference with our algorithm is that it implicitly transmits the information about the dynamically growing dictionary specific with respect to the given text by sending the 'next' character along with the pointer address. The question naturally arises whether if LZ algorithms had access to the entire static dictionary, will it compress better? By the very nature of the class of LZ algorithms, the dictionary may have to be dynamically updated since LZ may create references to not only words but groups of words representing previously encountered strings. This will necessitate more address bits. On the other hand, the *LZ algorithms are very efficient in the size of the dynamic dictionary that it will build since its text stream consists mainly of multiples of '*', single letters and bigrams resulting in smaller size of the address pointers (as we noted earlier, most words in the cryptic dictionary need not use more than two letters). Further experimentation is needed to resolve this issue.

3 Conclusion

We have experimentally established that an encrypted representation of text leads to substantial savings of storage space. The encryption is designed to exploit properties of compression algorithms and has produced compression ratios much better than those produced on the original un-encrypted text by several well known compression algorithms. As soon as we have access to source codes to ppm and dmc algorithms, we will report comparative results in a future paper.

4 Acknowledgment

The authors would like to thank R. Nigel Horspool and Timothy C. Bell for helpful comments and criticism on this work.

References

[BCW90] T.C. Bell, J.G. Cleary, and I.H. Witten. *Text Compression*. Prentice-Hall, 1990.

[CDR71] J.B. Carroll, P. Davies, B. Richman. *The American Heritage Word Frequency Book*.

[FiGr89] E.R. Fiala and D.H. Greene. *Data Compression with Finite Windows*, Comm. ACM, 32(4), pp. 490-505, April, 1989.

[HL90] D.S. Hirschberg and D.A. Lelewer. *Efficient Decoding of Prefix Codes*, Communications of the ACM, Vol.23,No.4,pp.449-458, April,1990.

[HoCo92] R.N. Horspool and G.V. Cormack. *Constructing Word-based Text Compression Algorithms*, Proc. Data Compression Conference 1992, (Eds. J.A. Storer and M. Cohn), IEEE Computer Society Press, 1992, pp. 62-71.

[Hu52] D.A.Huffman. *A Method for the Construction of Minimum Redundancy Codes*. Proc.IRE, 40(9),pp.1098-1101,1952.

[LZ77] J. Ziv and A.Lempel. *A Universal Algorithm for Sequential Data Compression.* IEEE Trans on Information Theory,IT-23,pp.337-243,1977.Also, by the same authors *Compression of Individual Sequences via Variable Rate Coding,* IT-24,pp.530-536,1978.

[NC95] National Science and Technology Council. *High Performance Computing and Communications: Foundation for America's Information Future,* A report by the Committee on Information and Communications, (Supplement to the President's FY 1996 Budget, submitted to Congress), Washington DC, 1995.

[RiLa79] J. Rissanen and G.G. Langdon, *Arithmetic Coding.* IBM Journal of Research and Development.Vol.23, pp.149-162, 1979.

[Ro62] F. Rosenblatt. *Principles of Neurodynamics.* Spartan, New York, 1962.

[SK64] E.S. Schwartz and B. Kallick. *Generating a Canonical Prefix Encoding,* Communications of the ACM, Vol.7, No.3, pp.166-169,March,1964.

[St88] J.A. Storer. *Data Compression: Methods and Theory.* Computer Science Press,1988.

[WMB94] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes.* Van Nostrand Reinhold, New York,1994.