Ramdomized Huffman codes

Ruy Luiz Milidiu
e-mail: milidiu@inf.puc-rio.br

Claudio Gomes de Mello
e-mail: cgmello@inf.puc-rio.br

PUC-RioInf.MCC49/04 December, 2004

Abstract : Huffman codes have widespread use in information retrieval systems. Besides its compressing power, it also enables the implementation of both indexing and searching schema in the compressed file. In this work, we propose a randomized variant of the Huffman data compression algorithm. The algorithm results in a small loss in coding, decoding and compression performances. It uses homophonic substitution and canonical Huffman codes.

Keywords : Security, compression, prefix codes, homophonic substitution.

Resumo : Códigos homofônicos são largamente utilizados em sistemas de recuperação de informação. Além de seu poder de compressão, também possibilita a implementação de esquemas de indexação e busca no texto comprimido. Neste trabalho, propomos uma variante aleatorizada do algoritmo de compressão de dados de Huffman. O algoritmo resulta em uma pequena perda na codificação, decodificação e compressão. Utiliza substituição homofônica e códigos canônicos de Huffman.

Palavras – chave : Segurança, compressão, códigos de prefixo, substituição homofônica.

1 Introduction

The request for electronic documents and services is increasing with the widespread use of digital data. Usually, electronic documents go through two separate processes: data compression to achieve low transmission cost, and ciphering to provide security. In this work, we propose a randomized variant of the Huffman data compression algorithm. For testing our proposal, we use the statistical test suite for random and pseudorandom number generators for cryptographic applications from NIST [11]. The algorithm shows a small loss in coding, decoding and compression performances. It uses homophonic substitution, canonical Huffman codes and a secret key for enciphering. The main goal is to avoid statistical analisys and generate a random output that can be used with other security strategies to harden the ciphered data as the ones described by Milidiu et al. in [7, 8, 9].

By using Huffman codes [2] we intend to mantain information retrieval system functions like indexing and searching in the compressed file, not so easily possible with adaptative data compression algorithms. Huffman codes have some advantages: simplicity; speed; auto synchronization, that is, it is possible to decode symbols in the middle of the coded text; in search, it is possible to code the keyword and search this coded keyword in the coded text, etc.

Klein et al. [4] analysed the cryptographic aspects of Huffman codes used to encode a large natural language on CD-ROM. And in [3], they show that this problem is NP-complete for several variants of the encoding process. Rivest et al. [13] cryptoanalysed a Huffman encoded text assuming that the cryptoanalyst does not know the codebook. According to them, cryptoanalysis in this situation is surprisingly difficult and even impossible in some cases due to the ambiguity of the resulting encoded data.

Data compression algorithms have been considered by cryptographers as a ciphering scheme [16]. To protect data against statistical analysis, Shannon [14] suggested that the language redundancy should be reduced before encryption. We use Huffman codes to achieve this. Wayner [17] proposed a simple scheme for assigning a secret key to a Huffman tree. First, suppose we have a Huffman tree with n leaves. It is well known that for a strict binary tree with n leaves we have: (n-1) internal nodes; and, the depth h of the tree satisfies: $\lceil \log n \rceil \leq h \leq n-1$. Wayner proposed that we obtain a new optimal tree by operating a XOR between each (n-1) branch of the tree with a secret key with size (n-1). A simple version of this approach is to assign one bit of a secret key for each level of the tree. In this case, we XOR each Huffman code with the corresponding position of the key. The size of the key can be too small in this last case, says O(log n). Milidiu et al. [6] show that one can efficiently implement prefix-free codes with length restriction, obtaining also very effective codes with small compression loss.

In this work we use homophonic substitution techniques to produce ran-

domness and achieve security against statistical attacks.

In section II, we describe canonical Huffman codes. These are a variant of the Huffman codes with the benefit of a more efficient decoding rate plus a convenient way of generating Huffman codes. In section III, we show that canonical codes for a Markovian source with a dyadic distribution generates a random stream of bits. This new result generalizes a previous result by Klein on Huffman codes [4]. In section IV, we describe variable-length homophonic substitution. Sections V, VI and VII are used to describe the algorithm, the results of performed experiments and our conclusions.

2 Canonical Huffman Codes

A Huffman tree is an optimal tree, but we also have several other optimal trees. Some of them are easily obtained by exchanging the symbol's place at the same tree level.

Example 1 *In this example, let us assume that we have a dictionary of 4 symbols* $\{a, b, c, d\}$ *and the plaintext* $T = bcbaadaacaadaaba$ *with 16 symbols.*

The frequency of each symbol is calculated and a Huffman tree labeled with 0 or 1, shown in figure 1, is created for these symbols. A traversal of the tree generates the model of table 1. The plaintext is then encoded with 27 bits as 111001100101001000010100110. In a fixed size text coding, we would have 2 bits per symbol, and hence 32 bits for the above plaintext. With the model of table 1, we achieve 27 bits. This is due to the fact that the most frequent symbols in the plaintext have the smallest codeword lengths.

The canonical Huffman tree is very similar to the standard Huffman tree, the only difference is that the leaves at the same level are shifted-left. Hence, a canonical Huffman tree is also optimal. Figure 2 illustrates the canonical tree of example 1. The codes are altered, but the code lengths are not. The corresponding 27 bits long coded file is 010000111001110001100111011. This is a variant of Huffman coding that provides a much higher decoding speed.

To generate the canonical Huffman codes one only needs Huffman code lengths of the symbols. The algorithm is described by Moffat et al. [10].

3 Dyadic Distribution

A dyadic probability distribution is a distribution in which each probability is a negative integer power of 2. For example, $(2^{-2}, 2^{-2}, 2^{-2}, 2^{-3}, 2^{-4}, 2^{-4})$ is a dyadic distribution.

Proposition 1 *In any optimal tree, for a dyadic distribution, each node at level* l *has probability* 2^{-l}*. Proof. Immediate by induction in the number of leaves.*

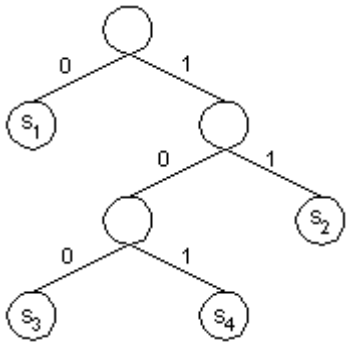

Figure 1: Huffman tree.

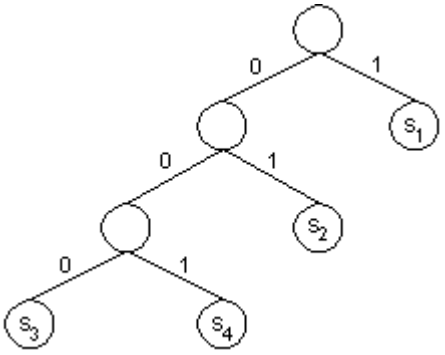

Figure 2: Canonical Huffman tree of example 1.

Table 1: Model.

Symbol	Frequency	Codeword
s_1	9	0
s_2	3	11
s_3	2	100
s_4	2	101

From proposition 1, it follows that canonical Huffman codes can be easily constructed in O(n) time. Next, we show a result that generalizes a previous finding by Klein [4] on Huffman codes.

Proposition 2 *Optimal prefix coding of a dyadic distribution source leads to a random stream of bits. Proof. The probability of receiving a giving new bit corresponds to a move from one internal node q at level l of the tree to either one of its two descendants u and v at level l + 1. These are conditional probabilities, that is,* $P(u|q) = \frac{P(u \cap q)}{P(q)} = \frac{P(u)}{P(q)}$ *as we suppose we have random input. Similarly,* $P(v|q) = \frac{P(v)}{P(q)}$. *From proposition 1, we get that* $P(u|q) = \frac{2^{-(l+1)}}{2^{-l}} = 2^{-1} = \frac{1}{2}$ *and* $P(v|q) = \frac{1}{2}$.

Theorem 1 *Canonical Huffman coding for a dyadic distributed Markovian source leads to a random stream of bits. Proof. Immediate from proposition 2.*

4 Variable-Length Homophonic Substitution

Homophonic substitution uniformizes the probability distribution of symbols in a plaintext, what is a desirable cryptographic property. Conventional homophonic substitution associates one symbol to a set of possible symbols, called homophones, such that the new symbols become equally probable. The main goal of the homophonic substitution is to convert a plaintext into a completely random sequence. Gunther [1] and Massey et al. [5] generalized homophonic substitution by using variable-length codes.

From theorem 1, if a plaintext has a dyadic distribution we can code it as a random stream of bits. Nevertheless, the probability of a symbol that ocurs in a plaintext is not necessarily a negative power of two. Hence, we artificially generate it by decomposing each probability value into a sum of negative powers of two. For example, a probability of 0.75 can be decomposed into a sum of $0.50+0.25 = 2^{-1}+2^{-2}$. We use this approach to achieve homophonic substitution. Each term of the decomposed probability corresponds to a new symbol that is called a homophone of the original symbol. So, for each probability p_i of a symbol s_i in the text we decompose it as $p_i = 2^{-a_1} + 2^{-a_2} + 2^{-a_3} + 2^{-a_4} + ...$, with $a_1 < a_2 < a_3 < ...$ in N*.

Now, we face the problem of infinite decomposition. One possible solution is to limit the number of decomposed probability terms, but truncating the decomposition changes the theoretical randomness of the bit generation. In section IV, we show that by increasing the textsize we can guarantee that the probabilities have finite decomposition.

In figure 3 we show the canonical Huffman tree for the homophones of example 1. The homophones have a dyadic distribution and are described in tabel 2. The corresponding coded file can be 000101000100001011 11010110111100011. Observe that it uses just 35 bits.

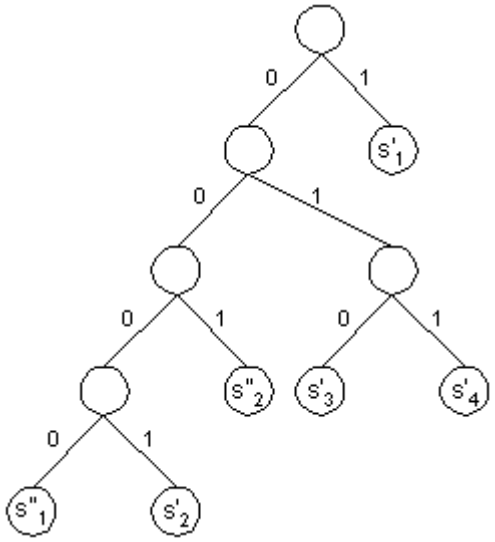

Figure 3: Canonical Huffman tree of homophones from example 1.

Table 2: Homophones of S from example 1.

Symbol	Homophones	Dyadic distribution
s_1	s_1' e s_1''	2^{-1} e 2^{-4}
s_2	s_2' e s_2''	2^{-3} e 2^{-4}
s_3	s_3'	2-3
s_4	s_4'	2-3

5 Algorithm description

Now, we describe our statistical random output compression algorithm. The new feature is the addition of randomness to the compressed data.

We use the canonical Huffman codes of variable-length homophones to encode a plaintext. The algorithm has the following steps:

5.1 Parse

Word-parse the plaintext to calculate the total number N of symbols and to create the alphabet of distinct symbols.

5.2 Break in blocks

In order to avoid infinite decomposition, we artificially increase the size of the plaintext to the next power of two, that is, it is desirable that the size of the plaintext be $N = 2^x$, where x is some positive integer. In this case, we have that each symbol has probability $p_i = \frac{f_i}{2^x}$, where f_i is the symbol frequency, that leads to a finite decomposition.

To guarantee that the plaintext's size is a power of two we insert dummy symbols to complete the size. If N is not a power of two, then inserting $2^{\lceil logN \rceil} - N$ dummy symbols can even double the original size of the plaintext, so we choose to break the original plaintext in three smaller blocks.

The steps are

- break the original plaintext T in three blocks T_1, T_2 and T_3 of sizes $n_1 = 2^{\lfloor logN \rfloor}$, $n_2 = 2^{\lfloor log(N-n_1) \rfloor}$ and $n_3 = N - (n_1 + n_2)$;
- complete block T_3 with $2^{\lceil logn_3 \rceil} - n_3$ dummy symbols;

With this approach we only have dummies in block T_3 minimizing text expansion.

For example, if $N = \mid T \mid = 23$ we have that $n_1 = 2^{\lfloor log23 \rfloor} = 16$, $n_2 = 2^{\lfloor log(23-16) \rfloor} = 4$ and $n_3 = 23 - (16 + 4) = 3$. So, $2^{\lceil log3 \rceil} - 3 = 1$ dummy symbol must be inserted in block T_3.

The number of dummies inserted is such that

$$0 \leq T_3 < 2.(N - 2^{\lfloor logN \rfloor} - 2^{\lfloor log(N-2^{\lfloor logN \rfloor}) \rfloor})$$

5.3 Parse blocks

For each block, parse again to recalculate the new frequencies related to the sizes n_1, n_2 and n_3.

Table 3: frequencies related to $T = bcbaabaacaabaababcaacac$.

Block	Size	f_a	f_b	f_c	f_{dummy}
1	16	9	5	2	0
2	4	2	1	1	0
3	3+1	1	0	2	1

5.4 Generate dyadic distribution

For each block, for each symbol in the block, generate its dyadic distribution by decomposing its probability into negative powers of two. Each component of the dyadic distribution is a homophone for the symbol.

5.5 Generate canonical Huffman codes

For each block, for each homophone, generate the Canonical Huffman codes. Use the fact that the length of the code is the absolute power of two index for each probability symbol.

Now we have one single dictionary but three possible frequencies for each symbol. One symbol can appear in one or more blocks having distinct relative frequencies to each block. Suppose $T = bcbaabaacaabaababcaacac$ with 23 symbols, Table 3 shows an example of frequencies in the blocks.

5.6 Code the plaintext

For each symbol choose one of its homophones and outcome its canonical Huffman code to the coded text. This choosing must be at random. We must use some random number generator to achieve this.

Moreover, we must break the correlaction that exists in natural language input text to guarantee the randomness of input to use proposition 2. We achieve this by permutating the input plaintext. We have a permutation vector $P = (p_1, p_2, ..., p_k)$ to permutate the input in k blocks. For example, the plaintext T of example 1 permutated with $P = (4, 3, 1, 2)$ becomes $T = abbcaaaddacaabaa$.

5.7 Cipher the model

After coding the plaintext we have a compressed text and a model. Then, we hide this model by any simmetric encryption standard like DES [15] or AES [12].

Next, we show our experiments with the compressed stream of bits using the statistical test suite for random and pseudorandom number generators for cryptographic applications from NIST [11].

We also avoid statistical attacks with the use of homophonic substitution [1, 5] and on the ambiguity of Huffman codes pointed out in [13], but other security strategies to harden the ciphered data shall be used, like the ones described by Milidiu et al. in [7, 8, 9].

6 Experiments

We use texts extracted from the Brazilian Constitution. These texts are encoded with our algorithm and then tested with the statistical test suite for random and pseudorandom number generators [11]. The statistical tests apllied are: Frequency Monobit Test, Frequency Test within a Block, Cumulative Sums Test, Runs Test, Longest-Run-of-Ones in a Block and the Lempel-Ziv Compression test. The compression rates achieved achieved were about 50% of the original size of the plaintext. And all passed the NIST suite for the statistical tests described.

7 Conclusions and future work

The algorithm shows a small loss in coding, decoding and compression performances. The main goal is to avoid statistical analisys and generate a random output that can be used with other security strategies to harden the ciphered data. From the experiments we conclude that this schema is useful to ramdomize the output stream bits of a natural language source. More experiments must be done to enforce the results. We intend to use, in future work, other collections such as the Trek and the Gutenberg Project.

References

[1] Gunter, C.G. *An Universal Algorithm for Homophonic Coding in Advances in Cryptology*, Eurocrypt-88, LNCS, vol. 330, 1988.

[2] Huffman, D. *A Method for the Construction of Maximum of Minimum Redundancy Codes*, Proc. IRE, 1098-1101, 1952.

[3] Klein, S.T., Fraenkel, A.S. *Complexity Aspects of Guessing Prefix Codes*, Algorithmica 12 409-419, 1989.

[4] Klein, S. T., Bookstein, A., Deerwester, S. *Storing Text Retrieval Systems on CD-ROM: Compression and Encryption Considerations*, ACM Transactions on Information Systems, vol. 7, no. 3, 1989.

[5] Massey, J.L., Kuhn, Y.J.B., Jendal, H.N. *An Information-Theoretic Treatment of Homophonic Substitution*, In Advances in Cryptology Eurocrypt-89, LNCS, vol. 434, 1989.

[6] Milidiu, R.L., Laber, E.S. *Improved Bounds on the Inefficient of Length-Restricted Prefix Codes*, SPIRE - String Processing and Information Retrieval, 1997.

[7] Milidiu, R.L., Mello, C.G, Fernandes J.R. *Adding security to compressed information retrieval systems*, SPIRE - String Processing and Information Retrieval, 2001.

[8] Milidiu, R.L., Mello, C.G, Fernandes J.R. *Substituio Homofnica Rpida via Cdigos de Huffman Cannicos*, Wseg - Workshop on Computer Systems Security, 2001.

[9] Milidiu, R.L., Mello, C.G., Fernandes J.R. *A Huffman-based text encryption algorithm*, SSI - Computer Security Symposium, 2000.

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

[11] NIST. *A statistical test suite for random and pseudorandom number generators for cryptographic applications*, NIST Special Publication 800-22, 2001.

[12] NIST. *Advanced Encryption Standard (AES)*, Web page: http:csrc.nist.gov/encryption/aes/aes_home.htm

[13] Rivest, R.L., Mohtashemi, M., Gillman, D.W. *On Breaking a Huffman Code*, IEEE Transactions on Information Theory, vol. 42, no. 3, 1996.

[14] Shannon, C. *Communication Theory of Secrecy Systems*, Bell Syst. Tech., vol. 28, no. 4, pp. 656-715, 1949.

[15] Schneier, B. *Applied Cryptography Second Edition: Protocols, Algorithms, and Source Code in C*, John Wiley & Sons, 1996.

[16] Simmons, G. *Contemporary Cryptology - The Science of Information Integrity*, IEEE Press, 1991.

[17] Wayner, P. *A Redundancy Reducing Cipher*, Cryptologia, 107-112, 1988.

[18] Zobel, J., Williams, H.E. *Compact In-Memory Models for Compression of Large Text Databases*, SPIRE - String Processing and Information Retrieval, 1999.