Universidade da Coruña

Departamento de Computación

New Compression Codes for Text Databases

Tese Doutoral

Doctorando: Antonio Fariña Martínez
Directores: Nieves R. Brisaboa e Gonzalo Navarro

A Coruña, Outubro de 2004

Ph. D. Thesis supervised by
Tese doutoral dirixida por

Nieves Rodríguez Brisaboa
Departamento de Computación
Facultade de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 981 167000 ext. 1243
Fax: +34 981 167160
brisaboa@udc.es

Gonzalo Navarro
Centro de Investigación de la Web
Departamento de Ciencias de la Computación
Universidad de Chile
Blanco Encalada 2120 Santiago (Chile)
Tel: +56-2-6892736
Fax: +56-2-6895531
gnavarro@dcc.uchile.cl

Abstract

This page will contain a small Abstract of the contents of the thesis

Resumo

Neste páxina vai ir un pequeno resumo dos contidos da tese en galego.

Acknowledgements

I owe a great part of this thesis to my directors for their tireless support and their help during the whole work. They always told me the way to follow.

This thesis is dedicated to my family, who was always there whenever I need them. Above all, I want to dedicate it to the most combative and strong people I have ever met. Thank you mum and dad.

Do neither I want myself to forget my brother and sisters: Arosa, Manu and 'María', nor the latest ones: Marina e Lexo.

CAMBIAR !!! CAMBIAR !!! I cannot omit my LBD colleague (so, I am not going to give names). They did easy the difficult moments and brought the light of our friendship.

Thanks also for you, Luisa, for the last months we share and the following that I hope will come.

And all the others I did not name, but you know you have your part in this work.

This is all for you.

Agradecementos

Grande parte desta tese débollela aos meus directores de tese polo seu apoio incansable, e porque sempre me serviron de apoio e me marcaron o camiño que debía de seguir.

A tese vai adicada á miña familia, que sempre estivo alí cando eu os necesitaba, pero sobre todo quérollela adicar ás dúas persoas máis loitadoras e fortes que coñezo desde hai tanto tempo. Gracias mamá e papá.

Tampouco me quero olvidar dos meus irmáns de toda a vida: Arosa, Manu e 'María', nin dos máis novos: Marina e Lexo.

CAMBIAR !!! CAMBIAR !!! Non me quero olvidar de ningún dos meus compañeiros do LBD (así que non vou dar nomes). Eles fixeron máis doados os momentos difíciles e deron luz propia á nosa amizade.

Gracias tamén a ti, Luisa, polos últimos meses que compartimos, e polos que espero virán.

E a tódolos demais que non citei, pero sabedes que tamén tedes a vosa parte neste traballo.

Vai por vós.

Aos meus pais e irmáns
Aos meus sobriños: María, Lexo e Paula

Contents

List of Tables

List of Figures

1

Introduction

1.1 Text Compression

Text compression techniques exploit redundancies in the text to represent it using less space [9]. The amount of text collections has grown rapidly in the last years, mainly due to the widespread use of digital libraries, documental databases, office automation systems and the Web. Current text databases contain hundreds of gigabytes and the Web is measured in terabytes. Although the capacity of new devices to store data grows fast and the associated costs decrease, the size of text collections increases faster. Moreover, CPU speed grows much faster than that of secondary memory devices and networks, so storing data in compressed form reduces I/O time, which is more and more convenient, even at the expense of some extra CPU time.

For these reasons, compression techniques have become attractive methods to save both space and transmission time. This two purposes of compression techniques correspond to two distinct compression scenarios that are described next.

1.1.1 Compression to space saving

Decreasing the space needed to store data is important. However, if the compression scheme does not allow us to search directly the compressed

text, text retrieval over documents will be less efficient due to the need of decompression before the search. Even if the search is done via an index, especially in compressed indexes, some text scanning is needed in the search process [31, 40]. Basically, compression techniques will be well-suited for *Text Retrieval* systems if they achieve good compression ratios and if they also maintain good search capabilities.

Classic compression techniques, like the well-known algorithms of Ziv and Lempel [60, 61] or classic Huffman [27], permit searching for words directly on the compressed text [41, 32] and empirical results showed that searching those texts can take half the time of decompressing that text and then searching it. However, the compressed search is twice as slow as just searching the uncompressed version of the text. Moreover, classic Huffman yields poor compression ratio (over 60%). Other techniques such as Byte-Pair Encoding compression [21] obtain competitive search performance [51, 46] but still poor compression on natural language texts.

Classic Huffman techniques are character-based statistical *two-pass* techniques. They use a *semi-static* model. A first pass over the text to compress gathers symbols and their frequencies, which are used to compress the text in a second pass. Then the compressed text is stored along with a header where the correspondence between the source symbols and codewords is represented. This header will be needed at decompression time.

An excellent idea to compress text is given by Moffat in [34], where it is suggested that words, rather than characters, should be the source symbols to compress. The compression scheme using a semi-static word-based model and Huffman coding achieves very good compression (compression ratio is about 25-30%). This improvement in compression ratio is due to the more biased distribution of the frequencies of words with respect to the distribution of characters. Moreover, since in Information Retrieval (IR) words are the atoms of the search, these compression schemes are particularly suitable for IR.

In [49], Moura et al presented *Plain Huffman Code* a word-based byte-oriented optimal prefix code. They also showed how to search, for either a word or phrase, into a text compressed with a word-based Huffman code without decompressing it, in such a way that the search can be up to eight

times faster than searching the plain uncompressed text. One of the keys of the efficiency is that the codes are sequences of bytes rather than bits.

The fastest and simplest search algorithms presented in [49] work over a coding variant called *Tagged Huffman Code*, where a bit of each byte is used to signal the beginning of a codeword. Hence, only 7 bits of each byte are used for the Huffman code. Note that the use of a Huffman code over the remaining 7 bits is mandatory, as the flag is not useful by itself to make the code a prefix code.

Over Tagged Huffman, *direct searching* [49] is possible, that is, the pattern can be compressed and searched using any classical string matching algorithm. In Plain Huffman this is not possible, as the pattern could occur in the text and yet not correspond to our codeword. The problem is that the concatenation of two codewords may contain the codeword of another source symbol. This cannot happen in Tagged Huffman Code because of the bit that distinguishes the first byte of each codeword. For this reason, searching with Plain Huffman requires inspecting all the bytes of the compressed text, while Boyer-Moore type searching [12] (that is, skipping bytes) is possible over Tagged Huffman Code.

Tagged Huffman Code has a price in terms of compression performance: The compressed file grows approximately by 11%. Although Huffman is the optimal prefix code, Tagged Huffman Code largely underutilizes the representation. In [13] it is shown that, by signaling the last byte instead of the first, the rest of the bits can be used in all their 128 combinations and the code is still a prefix code. Hence there are 128^i codewords of length i. The resulting code, called End-Tagged Dense Code, becomes closer to the compression ratio obtained by Plain Huffman Code (1 percentage point overhead). This code not only retains the ability of being searchable with any string matching algorithm, but it is also extremely simple to build (using a sequential assignment of codewords) and permits a more compact representation of the vocabulary (there is no need to store anything except vocabulary words ordered by frequency). Thus, the advantages over Tagged Huffman Code are (*i*) better compression ratios, (*ii*) same searching possibilities, (*iii*) simpler and faster coding and (*iv*) simpler and smaller vocabulary representation.

However, it is even possible to improve End-Tagged Dense Code compression ratio while maintaining all its good searchable features. In this work, (s, c)-Dense Code, a generalization of End-Tagged Dense Code compression ratio is presented. It overtakes its compression ratio by adapting the number of terminal and non terminal symbols to the distribution of frequencies of the words in the corpus to be compressed. As a result, (s, c)-Dense Coding compresses strictly better than End-Tagged Dense Code and Tagged Huffman Code, reaching compression ratios directly comparable with Plain Huffman Code (only 0.3 percentage points worse). At the same time, (s, c)-Dense Codes retain all the simplicity and direct search capabilities of End-Tagged Dense Codes and Tagged Huffman Codes.

1.1.2 Compression to file transmission

File transmission is another interesting scenario where compression techniques are very suitable.

In general, transmission of compressed data is usually composed of four processes: *compression*, *transmission*, *reception*, and *decompression*. The first two are carried out by a *sender* process and the last two by a *receiver*.

There are several interesting *real-time* transmission scenarios, however, where compression and transmission, as well as reception and decompression processes should take place concurrently. That is, the sender should be able to start the transmission of compressed data without preprocessing the whole text, and simultaneously the receiver should start reception and decompress the text as it arrives.

Real-time transmission is handled with so-called *dynamic* or *adaptive* compression techniques. Such techniques perform a single pass over the text (so they are also called *one-pass*) and begin compression and transmission as they read the data. Note that this is not possible in *two-pass* techniques since compression cannot start until the first pass over the whole text has been completed. Unfortunately, this reason makes *two-pass* codes unsuitable for real-time transmission.

In the case of dynamic codes, searching capabilities are not crucial

as happened in the semi-static methods used in *IR* systems. Instead of that, dynamic techniques should achieve good compression ratios and good compression/decompression time.

The first interesting adaptive techniques were presented by Faller and Gallager in [19, 22]. Such techniques consisted of dynamic Huffman codes. Those methods were later improved in [29, 52]. Since they are *one-pass* techniques, the frequency of symbols and the codeword assignment is computed and updated on-the-fly both by sender and receiver, during the whole transmission process. However, those methods were character- rather than word-oriented, and thus their compression ratios on natural language were poor (around 60%).

Currently, the most widely used adaptive compression techniques belong to the Ziv-Lempel family [9]. They are fast compression and decompression techniques, however, when applied to natural language text, the compression ratios achieved by Ziv-Lempel are not that good (around 40%).

Developing an adaptive compression technique with good compression ratio for natural language texts is a relevant problem. This work presents how to extend both End-Tagged Dense Code and (s, c)-Dense Code to build two new adaptive techniques. Moreover, their compression efficiency and processing cost is also evaluated. It is shown that the loss of compression is negligible with respect to the *semi-static* version (compression ratio is about 30-34%) and that compression speed is good. This makes up an excellent alternative for adaptive natural language text compression. A dynamic word-based Huffman method was also build to compare it against both Dynamic End-Tagged Dense code and Dynamic (s, c)-Dense Code. This Dynamic word-based Huffman technique is also described in detail because it resulted to be an interesting contribution. Since it is a Huffman method, it gets a slightly better compression than Dynamic (s, c)-Dense Code (0.3% off) and Dynamic End-Tagged Dense Code (1.0% off), but is is much slower. Dynamic word-based Huffman is about 20% slower in compression and more than 35% slower in decompression.

1.2 Objectives and contributions of the thesis

The first objective of this work was to improve the End-Tagged Dense Code taking advantage of the distribution of frequencies of the text words. This objective was accomplished by developing the (s, c)-Dense Code.

The second objective was to adapt End-Tagged Dense Code, as well as our improvement of it, the (s, c)-Dense Code, to real-time transmission scenarios. Therefore the contributions of this work are:

- The development of the (s, c)-Dense Code, a powerful generalization of End-Tagged Dense Code. It reduces the compressed text size in 2 percentage points with respect to End-Tagged Dense Code, while maintaining its good features: *i)* easy and fast decompression of any portion of compressed text, *ii)* direct searching in the compressed text for any kind of pattern with a Boyer-Moore approach. Empirical results comparing (s, c)-Dense Code with another well-known and powerful codes such as Tagged Huffman and Plain Huffman are also presented.
- The adaptation of End-Tagged Dense Code to real-time transmission by developing the Dynamic End-Tagged Dense Code. It has only a 0.1% compression ratio overhead with respect to the semi-static End-Tagged Dense Code.
- The adaptation of (s, c)-Dense Code to real-time transmission by developing the Dynamic (s, c)-Dense Code. It has at most a 0.04% overhead in compression ratio with respect the semi-static version of (s, c)-Dense Code.
- The empirical prove of the efficiency of all these methods comparing them against well-known compression techniques such as *gzip* and *bzip2*. Specifically, to have a good dynamic statistical method to compare with, we developed a word-based byte-oriented Dynamic Huffman method, which had never been implemented before (in the better of our knowledge). It is also a powerful and fast dynamic compression alternative, therefore it is a minor contribution of our work.

1.3 Outline

First, in Chapter 2, some basic concepts about compression, as well as a taxonomy of compression techniques are presented. After that, following the classification of compression techniques into both well-suited to Text Retrieval and well-suited to transmission, the remainder of this thesis is organized into two parts.

Part one is focused in *semi-static* or *two-pass* compression techniques. In Chapter 3, Compressed Text Databases, as well as the Text Retrieval systems that allow recovering documents from a Text Database are presented. We also present how compression can be integrated into those Systems.

In Chapter 4, a review of some classical text compression techniques is presented. In particular, classic Huffman code [27] operation is shown. Then Plain Huffman and Tagged Huffman codes [49] are focused, since this codes are the main competitors of End-Tagged Dense Code and also of (s, c)-Dense Code. Next the way Huffman compressed text can be searched for is described. Finally, other techniques such as Byte Pair Encoding and the Burrows-Wheeler Transform are shown.

In Chapter 5, End-Tagged Dense Code [13], the basis of the (s, c)-Dense Code is fully described. Chapter 6 presents (s, c)-Dense Code, the main contribution of this thesis. Empirical results regarding to compression ratio and also to encoding and decompression time are presented. (s, c)-Dense Code is compared with both End-Tagged Dense Code and Huffman based techniques.

Part two, once the word-based *semi-static* techniques have been explained, considers dynamic compression codes. An introduction of classic dynamic techniques, paying special attention to dynamic Huffman codes, is addressed in Chapter 7. That Chapter also includes a review of arithmetic codes, Dictionary based techniques and a predictive approach such as PPM.

Chapter 8 describes dynamic word-based byte-oriented Huffman code. Empirical results comparing this code with a character-based dynamic

Huffman code are also shown.

In Chapter 9, the dynamic version of End-Tagged Dense Code is considered. Its compression/decompression processes are described and compared with the Huffman based ones.

Chapter 10 consider dynamic (s, c)-Dense Code. The operation of this new method, and the way parameters s and c are adapted, are described. Empirical results of systematic experiments over real corpora, comparing all the presented techniques against well-known and used compression methods such as *gzip*, *bzip2* and an arithmetic compressor are presented.

Finally, Chapter 11 presenting the conclusions and future lines of work, complete this thesis.

2

Basic concepts

2.1 Chapter overview

This Chapter presents some basic concepts that are needed for a better understanding of this thesis. A brief description about several concepts related to Information Theory are shown first. Then a taxonomy of compression techniques is provided in Section 2.5. Finally, some measure units that can be used to compare compression techniques are presented in Section 2.6.

2.2 Concepts on Information Theory

Text compression techniques divide the source text into small portions that are then represented using less space. It is called *source symbols* to the basic units into which the text to be compressed can be partitioned, and it is called *vocabulary* to the set of all the distinct source symbols that appear in the text. We also consider that the size of the vocabulary is denoted by n.

A *encoding scheme* defines how a source symbol is encoded. That is, how it is mapped to a *codeword*. This codeword is composed by one or more *target symbols* from a *target alphabet* β. The number of elements of the target

alphabet is commonly $b = 2$ (binary code, β= {0,1}). *Compression* consists of substituting all the source symbols that appear in the input text by the codeword associated by the *encoding scheme*. The process of recovering the source symbol that corresponds to a given codeword is called *decoding*.

Compression techniques can be classified depending on the size of the codewords that are generated into two big groups: Fixed length and variable length methods. The most common variable length codes are the *statistical compression techniques*, which assign shorter codes to the most frequent source symbols (ie. Huffman [27] and arithmetic codes [1, 57]). Dictionary techniques, however usually generate fixed length codewords [60, 61, 54].

A code is a *distinct code* if each codeword is distinguishable from every other. A codification is said to be *uniquely decodable* if every codeword is identifiable when immersed in a sequence of codewords. Let consider that a vocabulary of three symbols A, B, C is used, and that the encoding scheme maps: $A \mapsto 0$, $B \mapsto 1$, $C \mapsto 11$. Then it is a *distinct code* since the mapping from source messages to codewords is one to one, but it is not *uniquely decodable* because the sequence 11 can be decoded as BB or as C. For example, the mapping $A \mapsto 1$, $B \mapsto 10$, $C \mapsto 100$ is *uniquely decodable*. However, a *lookahead* is needed during decoding. A bit 1 is decoded as A if it is followed by another 1. The sequence 10 is decoded as B, if it is followed by another 1. Finally the sequence 100 is always decoded as C.

A uniquely decodable code is called *prefix code* (or prefix-free code) if not codeword is a proper prefix of any other codeword. Given a vocabulary with source symbols A, B, C, the mapping: $A \mapsto 0$, $B \mapsto 10$, $C \mapsto 110$ produces a *prefix code*.

Prefix codes are *instantaneously decodable*. That is, a encoded message can be partitioned in codewords without the need of using a lookahead. This property is really important, since it permits to decode a codeword without having to inspect the following codewords in the encoded message. Therefore, it improves decompression speed. For example, the encoded message 010110010010 is decoded univocally as $ABCABAB$.

A *prefix code* is said to be a *minimal prefix code* if being x a proper prefix of some codeword, then $x\alpha$ is either a codeword or a proper prefix

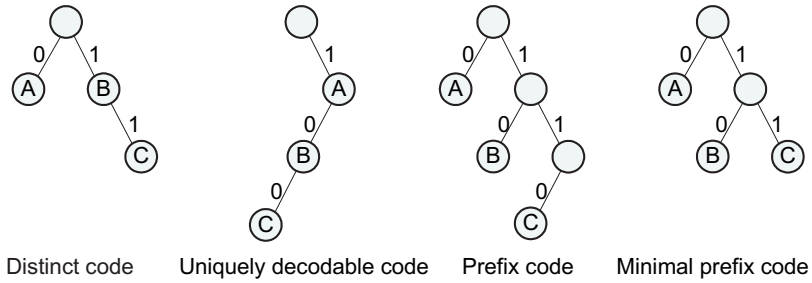


Figure 2.1: Definitions of distinct types of codes

of a codeword for each target symbol α in the target alphabet *beta*. For example, the code that maps: $A \mapsto 0$, $B \mapsto 10$ and $C \mapsto 110$ is not a minimal prefix code because 1 is a proper prefix of 10, but neither codeword 11 nor both the codewords 110 and 111 occur. If the map $C \mapsto 110$ is replaced by $C \mapsto 11$ then the code becomes a *minimal prefix code*. The minimality property avoids the use of codewords longer than needed.

In Figure 2.1 the distinct codes just described are shown.

2.2.1 The Kraft Inequality

When one tries to find an optimal prefix code, it is important to know in which situations it is possible to find such a code. Kraft's theorem [30] presented in 1949, gives some information about that possibility.

Theorem 2.2.1 *There exists a prefix binary code with codewords* $\{c_1, c_2, ..., c_n\}$ *and with corresponding codeword lengths* $\{l_1, l_2, ..., l_n\}$ *iff* $\sum_{i=1}^{n} 2^{-l_i} \leq 1$.

That is, Kraft's theorem guarantees that given a set of codewords, it is possible to find a prefix code without modifying their lengths. However, if the inequality does not hold, it is not possible to find a prefix code with the current codeword lengths, so it will be necessary to increase some of them until the inequality is verified.

Note that when the inequality turns a equality it occurs that the codeword length is minimal, therefore a minimal prefix code can be obtained.

Kraft's theorem says that for some given codeword lengths, a prefix code can be obtained. However, note that it is also possible that a non-prefix code can also be obtained with those codeword lengths. Therefore, even if a code satisfies Kraft's inequality, it does not imply that the code has to be a prefix code.

2.3 Redundancy and compression

Compression techniques are based on reducing the redundancy in the source messages, while maintaining the source information. In [2] a measure for the information content in a message x_i was defined as $I(x_i) = -log_b p(x_i)$, where b is the number of symbols of the target alphabet ($b = 2$ if a bit-oriented technique is used). This definition suppose that the probability of occurrence of a message x_i does not depend on the messages that appeared previously. From the definition of $I(x_i)$, it can be seen that:

- If $p(x_i)$ is high ($p(x_i) \to 1$) then the information content of x_i is almost *zero* since the occurrence of x_i give very little information.
- If $p(x_i)$ is low ($p(x_i) \to 0$) then x_i is a source message which does not usually appear. In this situation the occurrence of x_i makes information content maximal.

In association with the information content of a symbol x_i, the *average information content* of the source vocabulary can be computed by weighting the information content of each source message x_i by its probability of occurrence $p(x_i)$. Next expression yields $\sum_{i=1}^{n} -p(x_i)\ log_2\ p(x_i)$. This expression is referred to as the *entrophy* of the source [45] and is denoted by H. That is,

$$H = \sum_{i=1}^{n} -p(x_i)\ log_2\ p(x_i)$$

Since the length of the codeword associated to the message x_i has to be enough to represent $I(x_i)$, entrophy gives a lower bound to the number of

bits that will be required to encode the whole source text.

As shown, compression techniques try to reduce the redundancy of the source messages. Having $l(x_i)$ as the length of the codeword assigned to symbol x_i during the *encoding* phase, *redundancy* can be defined as follows:

$$R = \sum_{i=1}^{n} p(x_i)l(x_i) - H = \sum_{i=1}^{n} p(x_i)l(x_i) - \sum_{i=1}^{n} -p(x_i)\ log_b\ p(x_i)$$

Therefore, *redundancy* is a measure of the difference between the average codeword length and the *entrophy*. Since *entrophy* takes a fixed value for a given distribution of probabilities, a good code has to reduce the *average codeword length*. A code is said to be a *minimum redundancy code* if it has minimum codeword length.

2.4 Entrophy in Context dependent messages

Definitions in previous section treat source messages assuming independency in their occurrences. However it is usually possible to *model* the probability of the next source symbol x_i in a more precise way, by using those source symbols that have appeared before x_i.

The *context of a source symbol* x_i is defined as a fixed length sequence of source symbols that precede x_i.

Depending on the length of the *context* used, different models of the source text can be made. When that *context* is formed by m symbols, it is said that a *m-order model* is used.

In general, the probability of a source symbol x_i is obtained from its number of occurrences (assuming a *first-order model*). However when a *m-order model* is used to obtain that probability, the obtained probability is better than when a *low-order model* is used.

Several distinct *k-order models* can be combined to estimate the probability of the next input symbol. In this situation, it is mandatory to chose a method which describe how the frequencies estimation is done. In [18, 33, 9], it is shown a technique, called *Prediction by Pattern Matching*

(PPM), which combines several finite-context models of order k, such as k takes the values from 0 to m (m is the maximum context length). For each model, it takes account of all $k - length$ sequences S_i that have appeared previously. For a complete description of PPM see Section 7.4.

Depending on the order of the model, the *entrophy* expression varies:

- *base-order models.* In this case, it is considered that all the source symbols are independent and their frequency is uniform. Then $H = log_2\ n$.
- *Zero-order models.* In this case, all the source symbols are independent and their frequency consist of their number of occurrences. Therefore, $H = -\sum_{i=1}^{n} p(x_i) log_2\ (x_i)$.
- *First-order models.* The probability of occurrence of the symbol x_i conditioned by the occurrence of the symbol x_j is denoted by $P_{x_j|x_i}$ and the *entrophy* is computed as: $H = -\sum_{i=1}^{n} p(x_i) \sum_{j=1}^{n} P_{x_j|x_i} log_2\ P_{x_j|x_i}$.
- *Second-order models.* The probability of occurrence of the symbol x_k conditioned by the occurrence of the sequence $x_i x_j$ is denoted by $P_{x_k|x_j,x_i}$ and the *entrophy* is computed as:
  $H = -\sum_{i=1}^{n} p(x_i) \sum_{j=1}^{n} P_{x_j|x_i} \sum_{k=1}^{n} P_{x_k|x_j,x_i} log_2\ P_{x_k|x_j,x_i}$.
- *higher-order models* follow the same idea.

Given H, the redundancy can be calculated as $R = 1 - \frac{H}{log_2\ n}$.

2.5 Classification of Text Compression Techniques

Text compression is based on processing the source data to represent it in such a way that space requirements decrease. As result, the source information is maintained, but its redundancy is reduced. Decompressors act over the compressed data and permit to recover the original version of the data.

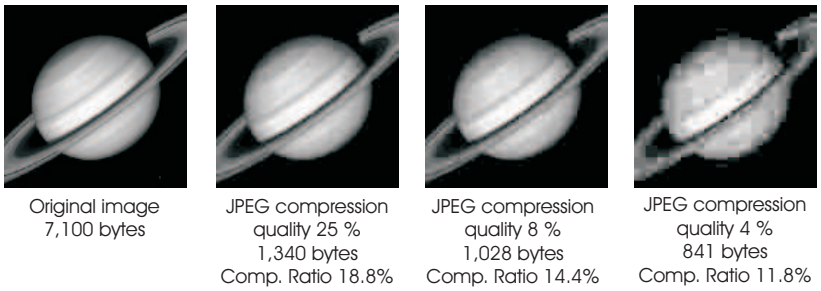


Figure 2.2: JPEG as an example of a lossy compressor

In some scenarios, some loss of source information can be allowed during compression. Depending on the possibility of recovering the whole source information during the decompression process, compression techniques can be classified into two large groups:

- *Lossy compressors*: In general, they are used to represent analogical data in a digital format. Compression of images (*jpeg, mpeg) or sound (*mp3) are typical scenarios. In this situations, some loss of the original analogical data can be permitted because human ocular/auditive sensibility cannot detect big differences between both the original and the decompressed data. In Figure 2.2 it is shown how *jpeg* image format can be used as a lossy compressor. Compression ratio can be reduced up to 14.4% (third image from the left) with not much loss of information.

  Since they have not to permit to recover the original data, but another with negligible differences, the compression ratios obtained are clearly better than in the following group of compressors.

- *Lossless methods*: In this kind of techniques it is mandatory to recover the same original data after decompression. Examples of typical scenarios where lossless compression techniques are required are text compression, filesystem compression, database records, etc.

Compression techniques *model* the source data to obtain an internal representation of the input symbols (which is used, for example, to obtain the probability of any source symbol) and then apply a compression scheme

that permits encoding those symbols. The correspondence between symbols and codewords has to be known by the decompressor, in order it to be able of recovering the original source data. Depending on the model used, compression techniques can be classified as using:

- *Static or non adaptive models.* The assignment of frequencies to each source symbol is fixed. They have probability tables previously computed (usually based on experience) that are used during the encoding process. Being fixed, provokes that those probabilities can match badly with source data in general, so these techniques are usually suitable only in specific scenarios. An example of application is *JPEG* image standard or the *Morse* code.

- *Semi-static models.* They are also commonly called *probabilistic* or *two-pass* techniques. This methods perform a first pass over the source text in order to obtain the probabilities of all the distinct source symbols that compose the *vocabulary*. Then those probabilities remain fixed and are used during the second pass, where the source text is processed again and each source symbol is assigned a code whose length depends on its probability.

  These techniques present two main disadvantages. The first one is that the source text has to be processed twice, and therefore encoding cannot start before the whole first pass has completed. This makes it impossible to apply *semi-static* techniques to compress text streams. The second problem relies on the necessity of providing the probabilities obtained by the compressor during the first pass, to the decompressor. Even when it is not a problem when large texts are compressed (as Heaps' Law [23] shows, in this case the size of the vocabulary is negligible compared to the compressed text size), it provokes an important loss of compression ratio when *semi-static* techniques are applied to small texts.

  Huffman based codes [27] are the main representatives of compressors that use a *two-pass* model.

- *Dynamic model.* They are also known as *dynamic*, *one-pass* or *adaptive* techniques because they do not perform a initial pass to obtain the

probabilities of the source symbols. This techniques commonly start with an initial empty vocabulary. Then the source text is read one symbol at a time. Each time a symbol is read, it is encoded using the current frequencies and its number of occurrences is increased. Moreover, each time a *new symbol* is input, it is also appended to the vocabulary. Note that the compression process *adapts* the frequency of each symbol as compression progresses.

Other advantages of *one-pass* techniques are their ability to compress streams of text, and the fact that the decompressor adapts the mapping between symbols and codewords in the same way the compressor does. This adaptation can be done by just taking account of the sequence of symbols that were already decoded. Therefore it is not needed to explicitly include that mapping as a *extra-part* of the compressed data.

Techniques such as the Ziv-Lempel family [60, 61, 54], and arithmetic encoding[1] [1, 57, 38] are well-known dynamic compression techniques.

Another classification of compression techniques can be done depending on how the encoding process takes place. Two families are defined: statistical and dictionary based techniques.

- *Statistical methods* assign to each source symbol a code whose length depends on the probability of the source symbol. Shorter codes are assigned to the more frequent symbols. Some typical statistical techniques are the *Huffman based codes* and the *arithmetic methods*.

- *Dictionary techniques* build a dictionary during the compression, in such a way that the last appeared phrases in the text are stored there. Encoding is performed by substituting those phrases by small pointers to their position in the dictionary. Compression methods from the Ziv-Lempel family are the most commonly used dictionary techniques.

[1] Arithmetic encoding is typically used along with a dynamic model, even when static and semi-static models are also applicable.

2.6 Measuring the efficiency of compression techniques

In order to measure the efficiency of a compression technique two basic concerns have to be taken into account. The complexity (both temporal and spatial) of the algorithms involved and the compression achieved.

The complexity gives an idea of how the technique will behave, but it is also necessary to obtain empirical results that permit to directly compare such technique against other methods in real scenarios.

Two common measures to compare the efficiency of some methods are: compression and decompression speed, and compression and decompression time.

- *Compression and decompression time* are usually measured in seconds or milliseconds.
- *Compression and decompression speed* measure the throughput achieved. Common speed units are: Kbytes per second and megaBytes per second.

Assuming that the i is the size of the compressed text, that the source text occupies o bytes, and that b is the number of bits used to represent a symbol in the source text, there are several ways to express the compression achieved by a compression technique. The most usual measures are:

- *Compression ratio* represents the percentage that the compressed text occupies with respect to the original text size. It is computed as:

$$compression\ ratio = \frac{o}{i} \times 100$$

- *Compression rate.* Indicates the decrease of space needed by the compressed text with respect to the source text. It is computed as:

$$compression\ rate = 1 - compression\ ratio = \frac{i-o}{i} \times 100$$

- *Bits per symbol (bps).* It compares the number of bits that are needed to represent a source symbol against the number of bits used to represent its codeword. It is computed as:

$$bps = \frac{o}{i} \times s$$

  Where s is the number of bits needed to represent a symbol of the source alphabet. Since the source alphabet contains typically characters, it often holds $s = 8$.

Part I

Semi-static Compression

3

Compressed Text Databases

3.1 Chapter overview

This Chapter presents the existence and diffusion of large Text Databases, and the importance of efficient Text Retrieval Systems in order to recover relevant data stored inside them. The main advantages that compression techniques provide to Text Retrieval Systems, as well as the way that techniques can be integrated into those Systems and their usually used inverted indexes, are also shown.

Finally, the problem of text searching is introduced, and some commonly used pattern matching algorithms are reviewed.

3.2 Motivation

A Document Database can be seen as a very large collection of documents and a set of tools that permit managing it and of course, searching inside it efficiently.

Libraries can be used as an analogy of a Document Database. They store lots of books, and a user can go there to find information relating to one issue of interest. However, this user has not usually time enough to review

all books in the library in order to find that information. Therefore, it is necessary to give him some kind of tool (i.e a catalog) which enables him to find rapidly the most relevant books.

A Digital Library, the set of articles published by a digital newspaper, or the Web in general, are examples of Document Databases. Since the number of documents stored is really large, the amount of space needed to store them is also enormous (and usually increases along time). The use of compression techniques reduces the size of the documents, and consequently the amount of space needed to store them in a storage device, at the expense of some CPU time needed for both compression and specially decompression.

However, not only saving storage space is important. A Text Retrieval System working against a Text Database has to provide efficient searches and retrieval of documents. In this way, *indexes* are the more commonly retrieval structures used.

3.3 Inverted indexes

An *inverted index* is a data structure that permits fast retrieval of all the positions in a Text Database where a *searched term* appears.

Basically an inverted index maintains a *terms vector* or *vocabulary* in which all terms (usually words) are stored. For each one of those terms t_i, a *list of occurrences* that keeps all the *positions* where t_i appears, is also stored.

Note that depending on the granularity used, the length of the *list of occurrences* can vary. If the level of granularity used is high (*a document*) then the list of occurrences is much smaller than when that level is low (a document and an offset inside it). Note that a term can appear lots of times in the same document, and that with low granularity, each occurrence has to be stored into the inverted index. Therefore, using higher granularity reduces the size of the whole index. This improves some searches and increases the possibilities of maintaining the whole index completely in memory.

For example, in Figure 3.1 the *list of occurrences* store 4 pointers. Each

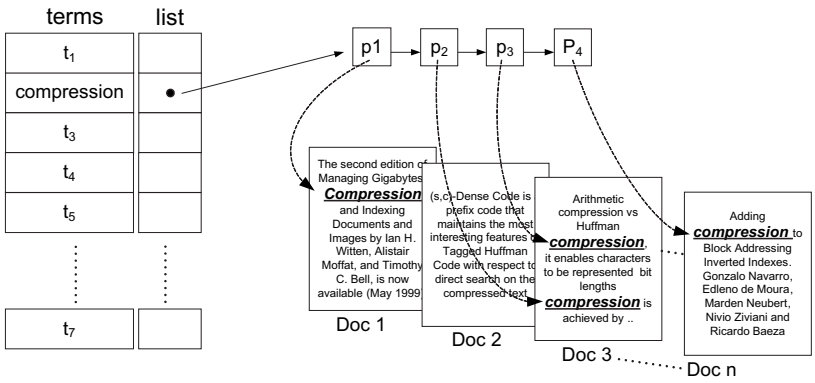


Figure 3.1: Structure of an inverted index

pointer references to the exact location (document and offset inside it) of term '*compression*'. If the level of granularity used for the *list of occurrences* had been the *document*, only 3 pointers would be needed, since '*compression*' appears only inside three documents.

Searches in the inverted index starts finding in the vocabulary the searched term. If the list of terms is maintained ordered, this process takes $O(log\ n)$ time since a binary search is possible. Then the *list of occurrences* is used to access documents.

The list of occurrences is used in a distinct way depending on the granularity of the index:

1. level of word (*word addressing index*): In this case, all terms that match the searched pattern are located in the vocabulary. Then the lists of occurrences indicates not only the documents, but also the offset inside documents where those terms appear.

2. level of document (*document index*): The *list of occurrences* points directly to documents. However, to find the exact position where a term appear in a document it is necessary to sequentially search for it in all the pointed documents.

3. level of block (*block addressing indexes*) [31, 40]. In this case the *list of*

occurrences points to *blocks*, and a block can hold several documents. Hence all searches require inspecting all the bytes in those pointed blocks, in order to know in which documents the searched pattern appears.

Block addressing indexes take special advantage of compression. Since a compressed document requires less space, more documents can be hold in a same block. This reduces considerably the size of the inverted index.

In the case of *block addressing indexes* (and also sometimes in *document addressing indexes*) a sequential pattern matching algorithm has to be used in order to sequentially search for a term through all documents in a block (or document). In Section 3.5 several string matching techniques are explained.

Compression has been used along with *inverted indexes* with good results [40, 63]. Text compression techniques have to poses some characteristics in order this symbiosis to be productive. This characteristics are shown in Section 3.4.

3.4 Compression schemes for Text Databases

Not all compression techniques are suitable for its use along with inverted indexes. Compression schemes have to permit two main operations: *Direct access* and *direct search* into the compressed text.

3.4.1 Direct access

The *direct access* property deals with the possibility of decompressing a random snippet of a compressed text without having to process it from the beginning.

In some situations we are interested in retrieving only a small context of the searched term, not the whole document. For example, a web search engine presents to the user a list of relevant links and a small *snippet* of the document as the context where the searched term appear.

If the compression scheme does not permit direct access, once a document has been located via the index, it is necessary to start decompressing the document from the beginning, and proceed until the searched term is found. Finally a small part (snippet) of the document is retrieved around the searched term.

However if the compression scheme permits direct access it is possible to have an *index over the compressed text*. In such case, the index returns the position inside the compressed document, and only a small portion of the compressed text, around the searched term need to be decompressed.

3.4.2 Direct search

It is clear that, it is not interesting to have to decompress the text before searching on it.

When a compression scheme supports *direct search*, given a pattern, it is possible to compress it and then to search for the compressed pattern directly into the compressed text with any sequential string matching algorithm [12, 28, 50]. Direct search, as it was proven by Navarro et al. [48, 49], clearly improves searching performance. It is up to 8 times faster for approximate searches than the search on uncompressed text.

The use of a compression scheme allowing direct search, permits adding compression to *block addressing indexes*, as it is shown in [40].

Therefore, using compression techniques which permit direct search and direct access results very useful. It enables maintaining the text in a compressed form all the time (what saves space), but it also improves the efficiency of Text Retrieval techniques.

3.5 String matching

The string matching problem consists basically on finding all the occurrences of a given pattern p in a larger text t [12, 28, 50].

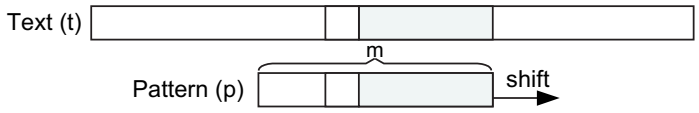


Figure 3.2: Boyer-Moore elements description

Patterns can be classified as follows: *i) words* (car, dog), *ii) prefixes and suffixes* (under-, -ation), *iii) substrings* (-ompressi–), *iv) alphabetical ranges* ([a, b) returns words starting with an 'a'), *v) allowing errors* (color "-distance=1", returns words with a edition distance less or equal to '1' with respect to color, i.e. color), and *vi) regular expressions* ($(car|bike)(0|1)^*$ returns several terms, i.e.: car, car0, car1, car01,bike,...).

In this Section some commonly used pattern matching techniques are presented. For more information about other pattern matching techniques and about how to search for several patterns (*multiple string matching*) see [42], where a good survey is presented.

3.5.1 Boyer-Moore algorithm

This algorithm uses a search window that corresponds to the searched pattern and is moved along the text. The algorithm starts aligning the pattern p (we define m = size of p) with the leftmost part of the text t, and then it searches for suffixes, right-to-left, of the pattern in the text until the pattern matches the whole search window. In each step, the longest possible safe-shift to the right of the pattern is performed.

Let suppose a suffix μ of the search window which is also a suffix of the pattern, and that the character σ from the text does not match with character α in the pattern. Figure 3.2 shows those elements. Then three shift functions are computed:

δ_1 : If the suffix μ appears in another position in p, then δ_1 is associated the distance from α to the next occurrence of μ which is not preceded by α backwards in the pattern.

δ_2 : If the suffix μ does not appear in any text position, then suffixes ν

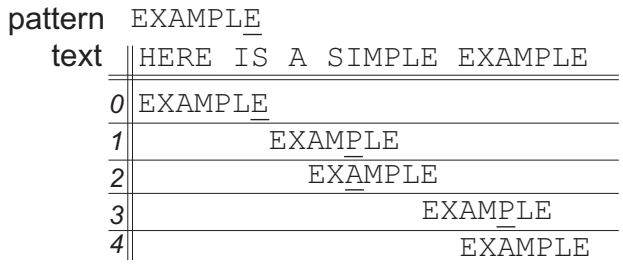


Figure 3.3: Example of Boyer-Moore searching

of μ are taken into account, since they could also be a prefix of the pattern in next step. In such case δ_2 is assigned the length of the longest prefix ν of p that is also a suffix of μ.

δ_3 : It is associated to the distance from the rightmost occurrence of σ in the pattern. If σ does not appear in p then $\delta_3 = m$.

Having computed δ_i the algorithm computes two values: $M = max(\delta_1, \delta_3)$ and $min(M, m - \delta_2)$. The last value is used to slide the search window before next iteration. However, note that when a full match of the pattern is found inside the text, only the δ_2 function is used.

An example of the way Boyer-Moore type searching works, is shown in Example 3.5.1 and can also be followed in Figure 3.3.

Example 3.5.1 Lets search for pattern `'EXAMPLE'` inside the text `'THIS IS A SIMPLE EXAMPLE'`. In step 0, the search window contains `'HERE IS'`. Since `'E'`$\neq$`'S'` then $\sigma =$`'S'`, $\alpha =$`'E'` and $\mu = \emptyset$. We achieve $\delta_1 = \delta_2 = 0$ and $\delta_3 = m = 7$, hence the pattern is shifted 7 positions to the right.

In step 1, the search window contains `' A SIMP'`. Since `'E'`$\neq$`'P'` then $\sigma =$`'P'`, $\alpha =$`'E'` and $\mu = \emptyset$. We achieve $\delta_1 = \delta_2 = 0$ and $\delta_3 = 2$ (`'P'` appears 2 positions backwards in the pattern), hence the pattern is shifted 2 positions to the right.

In step 2, the search window contains ‘ SIMPLE’. Then σ =‘I’, α =‘A’ and μ =‘MPLE’. We achieve $\delta_1 = 0$ $\delta_2 = 1$ (since the longest prefix of the pattern that is a suffix in μ is ‘E’) and $\delta_3 = 0$. Therefore the pattern is shifted $min(0, 7 - 1) = 6$ positions to the right.

In step 3, the search window contains ‘E EXAMP’. Since ‘E’$\neq$‘P’ then σ =‘P’, α =‘E’ and $\mu = \emptyset$. We achieve $\delta_1 = \delta_2 = 0$ and $\delta_3 = 2$ (‘P’ appears 2 positions backwards in the pattern), hence the pattern is shifted 2 positions to the right.

Finally, the pattern and the search window match in step 4. Then the pattern is shifted $m = 7$ positions to the right and the process ends. □

The Boyer-Moore search is $O(mn)$ in the worst case. In average, its complexity is sublinear.

Variations of the initial Boyer-Moore algorithm as the *Horspool* algorithm and the *Sunday variation* are described in [25, 50, 42].

3.5.2 Shift-Or algorithm

The Shift-Or algorithm [58, 4] uses bitwise techniques. The algorithm acts as a nondeterministic automaton that searches for the pattern in the text. Its key idea is to maintain a set with all prefixes of the pattern p that match a suffix of the read text. This set is represented via a bit mask D, such as $D = d_m, d_{m-1}, \ldots, d_1$ (note that d_i represents the i^{th} less significative bit in D). If the searched pattern is not larger than the word size of the used computer w (current architectures have 32 or 64 bits), then the bit mask fits completely in a CPU register and Shift-Or algorithm becomes really efficient.

Being m the size of the pattern and n the number of characters in the used alphabet (Σ), the algorithm starts building a table B of size $(n \times m)$, as follows: Columns are labelled from right to left with the letters of the pattern. Rows are labelled with all symbols in $\sum$. Then, each table entry is filled in as follows:

$$B[i,j] = \begin{cases} 0 & \text{if } columnLabel[j] = rowLabel[i] \\ 1 & otherwise \end{cases}$$

Example 3.5.2 Filling the B table, when the pattern '`bcab`' is being searched into the text '`abcabx`'.

		pattern			
		b	a	c	b
Σ	a	1	0	1	1
	b	0	1	1	0
	c	1	1	0	1
	x	1	1	1	1

To fill in the table B, it is needed to look at the pattern.

Zeros are set in those positions where the row and the column are labelled with the same letter.

For example the letter `'a'` appears in position 3 (starting numbering from right to left) in the pattern, so the element $B(\texttt{'a'}, 3)$ will have a 0 as its entry. After filling in the whole table with zeros in the appropriate rows and columns, the remainder entries of the table B are set with ones. □

Once the B table has been filled, the searching algorithm proceeds. A *match register* D of size m is initialized with ones. And a match of the current character input, with the pattern, will be represented with a zero. Next, letters from the text are input. Each time a letter from the text is read, D is updated as follows: $D \leftarrow (D << 1) \mid B[t]$, where '|' indicates a bitwise OR. A match is detected when $D[0] = 0$. See the example in Figure 3.4 for a better understanding of the whole process.

Note that, if a letter t, which does not belong to the pattern, is input, then the row $B[t]$ has only ones, so $D \leftarrow' 111..1'$. Hence the algorithm goes again to the initial state.

Figure 3.4 shows how Shift-Or algorithm searches for the pattern '`example`' inside the text '`this-is-an-example`'.

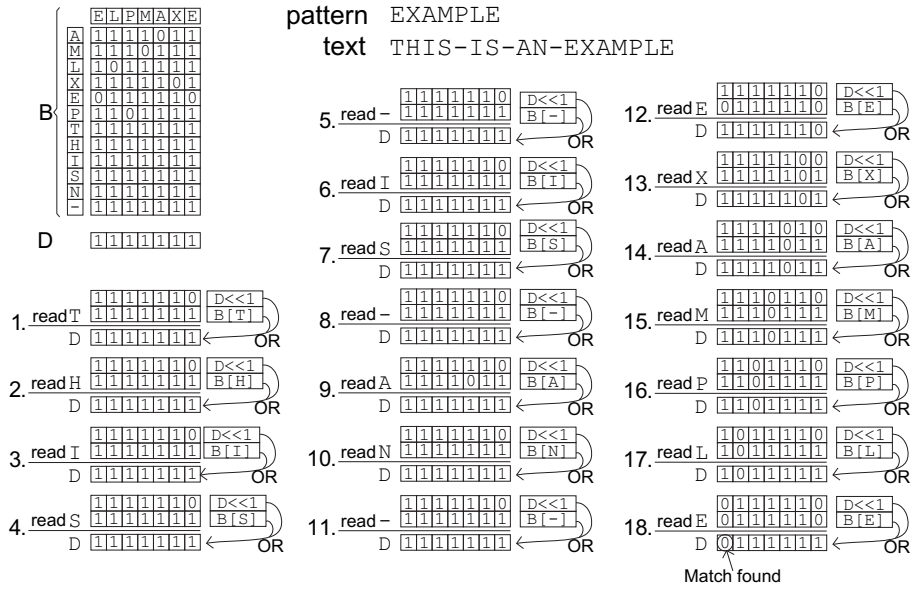


Figure 3.4: Example of Shift-Or searching

Assuming that the pattern length is no longer than the memory-word size of the machine, the space and time complexity of the preprocessing phase (building B table) is $O(m + \sigma)$.

The time complexity of the searching phase is $O(n)$, Therefore, it is independent from the alphabet size and the pattern length.

3.6 Summary

In this Chapter the necessity of maintaining Text Retrieval Systems which permit efficient access to words inside documents in a Text Database was shown. Some advantages that compression techniques offer to those systems, as well as the characteristics that they have to poses in order to be really suitable for those Text Retrieval Systems, were also presented.

Since the use of compression usually requires searching the compressed

text, we also presented the string matching problem. Finally Boyer-Moore and Shift-Or algorithms, two commonly used and fast pattern matching techniques, were also described.

4

Semi-static text compression techniques

4.1 Chapter overview

This Chapter presents a classic compression technique such as the character oriented Huffman [27], and the variations of it that appeared in the last years.

The Chapter starts presenting the classic Huffman algorithm, the basis of a new word-based generation of codes that appeared in the nineties. In Section 4.3, it is presented a brief description of some of the most interesting natural language text compression codes that are used nowadays. Special attention is paid to describe the techniques called *Plain Huffman* and *Tagged Huffman*. In Section 4.4, it is shown how searches can be performed directly into the compressed text when text is compressed via Plain Huffman or Tagged Huffman code. Finally, the Chapter ends with the description of two techniques such as the *Byte Pair Encoding* (BPE) and the *Burrows-Wheeler Transform* (BWT). BPE is a compression technique that enables efficient searches and decoding. BWT is an algorithm which transforms one text to another more compressible and can be used along with a Huffman based or an arithmetic encoder [57], improving their compression ratio.

4.2 Classic Huffman Code

The definition of the Classic Huffman code appeared in 1952 [27]. It is a character-based bit-oriented statistical semi-static technique. The codes that it generates are *prefix codes*. This method originated an important break-point in the field of the compression techniques. Several compression Huffman-based techniques where developed since that date [22, 29, 52, 48].

As a semi-static technique, two passes over the input text are performed: In the first, the number of occurrences of the distinct symbols are computed. Once the frequencies of all input symbols are known, Huffman algorithm builds a tree, that is used in the encoding process, as it is explained next. This tree is the basis of the encoding schema. In the second pass, each input symbol is encoded and output. Using the Huffman tree, shorter codes are assigned to the more frequent input symbols.

4.2.1 Building a Huffman Tree

A classic Huffman tree is a binary tree built as follows: Symbols are assigned to the leaf nodes of the tree, and their position in the tree (level) depends on the probability of each symbol. Moreover one condition has always to be keep: the number of occurrences of a node in a higher level can never be smaller than the number of occurrences of a node placed in a lower level.

A Huffman tree is built through the following process:

1. A list of leaf nodes is created, one node for each distinct input symbol. Each node stores a symbol and its frequency. This list is then sorted by frequency.
2. The b least frequent nodes are picked from the list. ($b = 2$ in a binary tree)
3. A new internal node is added to the tree. This internal node is assigned the sum of the frequencies of the nodes picked in the previous step. Then the b nodes are removed from the list of nodes (they are now in the tree), being substituted by the internal node just created.

4. Steps 2) and 3) are again executed while more than b symbols remain in the list of blocks. If b or less nodes remain in the tree, then those nodes are join into a common parent. This new internal node is the root of the Huffman tree (and its frequency will be the sum of occurrences of all the input symbols).

To understand the whole process let see Example 4.2.1 and Figure 4.1.

Example 4.2.1 Building a Huffman tree from 5 input symbols: $\{a, b, c, d, e\}$ that have frequency $0,40$, $0,25$, $0,18$, $0,12$ and $0,05$ respectively.

Figure 4.1 shows the process step by step. In the first step, a list of nodes associated to the input symbols: $\{a, b, c, d, e\}$ is created. In step two, the two least frequent nodes are chosen, and they are joined into a new internal node whose frequency is $0,20$, that is, the sum of frequencies of both d and e. In step three, the current least frequent nodes b, c and the internal node just created could be taken. In this case, we chose the internal node, and c and join them into a new internal node of frequency $0,40$ which is added to the three. Note that if b had been chosen, a distinct Huffman tree would be generated (more that one Huffman tree exist usually). Next, b and the previous internal node are joined into a new internal node, its frequency is $0,60$. In the last step, only two nodes remained to be chosen. These two nodes are set as children of the root node. □

The cost of building a character oriented huffman tree is $O(n)$ where n is the number of symbols in the tree. In general the number of iterations needed to build a huffman tree can be bounded as $\lceil \frac{n}{b-1} \rceil$.

Once the Huffman tree has been build it is possible to begin the compression process. All branches in the tree are numbered as follows: The left-branch of a node is set a 0 and the right-branch is given a 1. The path from the root of the tree to the leaf node where a symbol appears gives the code of that symbol.

Codes assigned to the symbols, using the tree in Example 4.2.1 as encoding schema, are shown in Table 4.1.

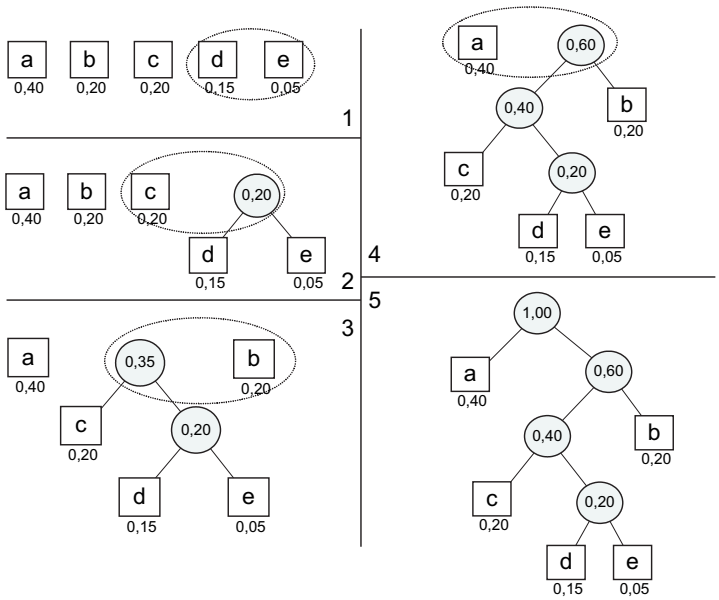


Figure 4.1: Building a classic Huffman tree

input symbol	→	code assigned
a	→	0
b	→	11
c	→	100
e	→	1010
f	→	1011

Table 4.1: Codewords assigned to each symbol in Example 4.2.1

To decompress a compressed text, the shape of the Huffman tree used during compression has to be known. Then it is input one bit at a time. The value of such a bit permits to chose the right or left branch of an internal node. When a leaf is reached, a symbol has been recognized and is output. Then the decompressor goes back to the root of the tree and continues the process.

4.2.2 Canonical Huffman tree

In 1964, Schwartz and Kallick [44] defined the concept of a canonical Huffman code. In essence the idea pointed there is that Huffman's algorithm is only needed to compute the length of the codewords that will be mapped to each one of the symbols in the dictionary. Once those lengths are known, codewords can be assigned in several ways, the only condition that has to be taken into account is to produce *preffix codes*.

Intuitively a canonical code builds the prefix code tree from left to right in increasing order of depth. At each level, leaves are placed in the first position available (from left to right). Figure 4.2 shows the canonical Huffman tree from Example 4.2.1.

The canonical code possesses some mathematical properties.

- Codewords are assigned in increasingly size order, depending on the lengths computed with Huffman's algorithm.
- The codewords of a given length are consecutive binary numbers.
- The first codeword c_l of length l is related to the last codeword of length $l-1$ byte the equation $c_l = 2(c_{l-1} + 1)$.

Given Figure 4.2, where codeword lengths are respectively 1,2,3,4 and 4, the codewords obtained are: 0, 10, 110, 1110 and 1111.

The main advantage of the canonical representation consists of that it is possible to represent the Huffman tree, by only using the lengths of the codewords. Therefore the vocabulary needed for decompression will only

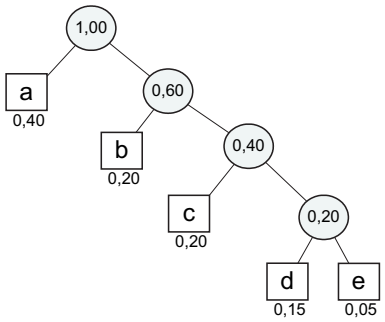


Figure 4.2: Example of canonical Huffman tree

require storing: *i)* the list of symbols of the vocabulary, *ii)* the *minimum* and *maximum* codeword length values and *iii)* the number of codes of each length. For example, the shape of the Huffman tree in Figure 4.2 can be saved as: <a, b, c, d, e><1, 4><1, 1, 1, 2>.

As result keeping a the shape of the canonical Huffman tree of n words in a canonical Huffman code takes $O(log_2 n)$.

Moreover canonical codes reduce also the memory requirements needed during compression and decompression, since the Huffman tree has not to be kept into memory.

Implementation details of a canonical Huffman code can be seen in [24]. Even when the canonical representation was defined for bit-oriented Huffman approach, it can be also defined to a byte-oriented approach. More details about how a byte-oriented canonical Huffman code can be built, appear in [35, 39].

4.3 Word-Based Huffman Compression

The traditional implementations of the Huffman code are character based, i.e., they use the characters as the symbols of the alphabet. Therefore compression is poor because the distribution of frequencies of characters in natural language texts is not much biased. In [34], Moffat uses the words in the text as the symbols to be compressed. This idea joins the requirements

of compression algorithms and of IR systems, as words are the basic atoms for most IR systems. The basic point is that a text is more compressible when regarded as a sequence of words rather than characters.

In [49, 62], two compression schemes that uses this strategy combined with a Huffman code is presented. From a compression viewpoint, character-based Huffman methods are able to reduce English texts to approximately 60% of their original size, while word-based Huffman methods are able to reduce them to $25\% - 30\%$ of their original size, because the distribution of words is much more biased than the distribution of characters.

The compression schemes presented in [49, 62] use a semi-static model, that is, the encoder makes a first pass over the text to obtain the frequency of all the words in the text and then the text is coded in the second pass. During the coding phase, original symbols (words) are replaced by codewords. For each word in the text there is a unique codeword, whose length varies depending on the frequency of the word in the text. Using the Huffman algorithm, shorter codewords are assigned to more frequent words.

The basic method proposed by Huffman is mostly used as a binary code as shown in Section 4.2, that is, each word in the original text is coded as a sequence of bits. In [49] they modified the code assignment such that a sequence of bytes instead of bits is associated with each word in the text.

The tree building of a 256-ary Huffman tree is similar to the construction of a binary tree. The main difference is the start point of the process, in the number of nodes R that have to be chosen in the first iteration of the process. That is:

$$R = \begin{cases} 1 + ((n - 2^b) \ mod \ (2^b - 1)) & \text{if } ((n - 2^b) \ mod \ (2^b - 1) > 0 \\ n & \text{if } ((n - 2^b) \ mod \ (2^b - 1) = 0 \end{cases}$$

Then, in the following iterations of the process, 256 nodes are taken and set as children of a new internal node, until only 256 available nodes remain to be chosen.

Experimental results have shown that, on natural language, there is no significant degradation in the compression ratio (less than 5%) by using bytes instead of bits. In addition, decompression and searching are faster

with byte-oriented Huffman code because no bit manipulations are necessary.

4.3.1 Plain Huffman and Tagged Huffman Codes

In [49] two Huffman codes following a word-based approach are presented. These codes are called *Plain Huffman* and *Tagged Huffman*.

Both Plain Huffman and Tagged Huffman codes allow to search for a word in the compressed text without decompressing it, in such a way that the search can be up to eight times faster for certain queries [49]. The key idea of this work (and others like that of Moffat and Turpin [39]) is the consideration of the text words as the symbols that compose the text (and therefore the symbols that should be compressed). Since in Information Retrieval (IR) text words are the atoms of searches, these compression schemes are particularly suitable for IR. This idea has been carried on further up to a full integration between inverted indexes and word-based compression schemes, opening the door to a brand new family of low-overhead indexing methods for natural language texts [56, 40, 62].

In [49], they call *Plain Huffman Code* to the one we have already described, that is, a word-based byte-oriented Huffman code. Plain Huffman obtains compression ratios about 30%-35% when applied to compression of English texts.

Plain Huffman Code does not permit direct searching the compressed text by simply compressing the pattern and then using any classical string matching algorithm. This does not work, as the pattern could occur in the text and yet not correspond to our codeword. The problem is that the concatenation of parts of two codewords may form the codeword of another vocabulary word.

The second code proposed in [49] is called Tagged Huffman Code, which avoids that problem in searches. This technique is just like the previous one differing only in that the first bit of each byte is reserved to flag whether the byte is the first byte of a codeword. Hence, only 7 bits of each byte are used for the Huffman code. Note that the use of a Huffman code over the remaining 7 bits is mandatory, as the flag bit is not useful by itself to make

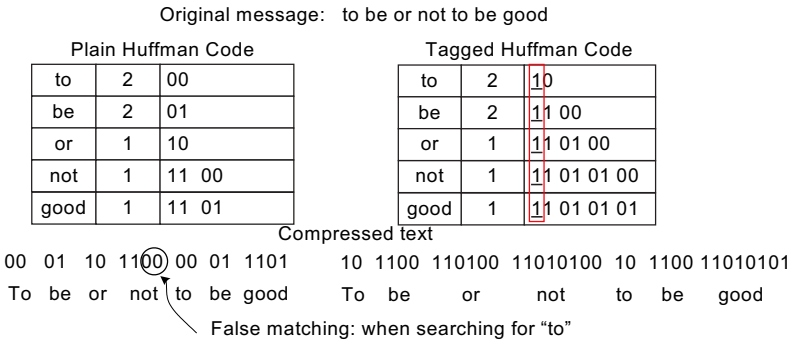


Figure 4.3: Example of False Matchings in Plain Huffman

the code a prefix code.

Therefore, due to the use of the flag bit in each byte which determines if the byte is the first byte of a codeword or not, no spurious matchings can happen in Tagged Huffman Code. See Figure 4.3 for an example of how false matchings can happen in Plain Huffman Code.

For this reason, searching with Plain Huffman requires inspecting all the bytes of the compressed text from the beginning, while Boyer-Moore type searching [12, 50] (that is, skipping bytes) is possible over Tagged Huffman Code.

Tagged Huffman Code has a price in terms of compression performance: full bytes are stored, but only 7 bits are used for coding. Hence, the compressed file grows approximately by 11%. As compensation, Tagged Huffman searches compressed text much faster than Plain Huffman.

The differences among the codes generated by the Plain Huffman Code and Tagged Huffman Code are shown in the following example.

Example 4.3.1 Assuming that the vocabulary has 17 words, with uniform distribution ($p_i = 1/17$) in Table 4.2 and with exponential distribution ($p_i = 1/2^i$) in Table 4.3. The resulting canonical Plain Huffman and Tagged Huffman trees are shown in Figure 4.5 and Figure 4.4.

For the sake of simplicity, from this example, we consider that the *bytes*

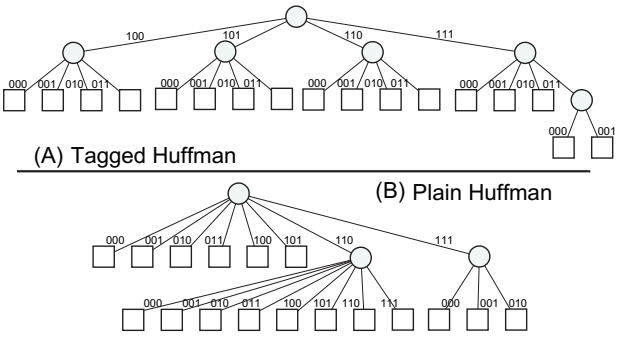


Figure 4.4: Plain and Tagged Huffman trees for an uniform distribution

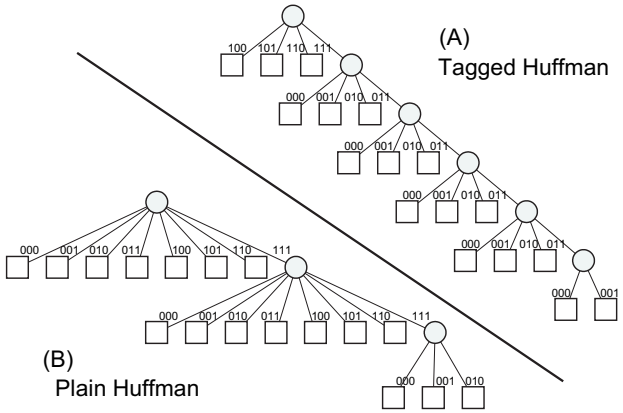


Figure 4.5: Plain and Tagged Huffman trees for an exponential distribution

used are formed by only three bits. Hence, Tagged Huffman Code uses one bit for the flag and two for the code (this makes it look worse than it is). Flag bits are shown underlined. □

4.4 Searching Huffman Compressed Text

Direct text searching takes much importance in *block addressing indexes*, a new family of low-overhead indexing methods for natural language texts [56, 40, 62] which came up in the last years. Basically, in order to reduce index space, the index does not point to exact word positions but to text

Word	Probab.	Plain Huffman	Tagged Huffman
A	1/17	000	100 000
B	1/17	001	100 001
C	1/17	010	100 010
D	1/17	011	100 011
E	1/17	100	101 000
F	1/17	101	101 001
G	1/17	110 000	101 010
H	1/17	110 001	101 011
I	1/17	110 010	110 000
J	1/17	110 011	110 001
K	1/17	110 100	110 010
L	1/17	110 101	110 011
M	1/17	110 110	111 000
N	1/17	110 111	111 001
O	1/17	111 000	111 010
P	1/17	111 001	111 011 000
Q	1/17	111 010	111 011 001

Table 4.2: Codes for an uniform distribution.

Word	Probab.	Plain Huffman	Tagged Huffman
A	1/2	000	100
B	1/4	001	101
C	1/8	010	110
D	1/16	011	111 000
E	1/32	100	111 001
F	1/64	101	111 010
G	1/128	110	111 011 000
H	1/256	111 000	111 011 001
I	1/512	111 001	111 011 010
J	1/1024	111 010	111 011 011 000
K	1/2048	111 011	111 011 011 001
L	1/4096	111 100	111 011 011 010
M	1/8192	111 101	111 011 011 011 000
N	1/16384	111 110	111 011 011 011 001
O	1/32768	111 111 000	111 011 011 011 010
P	1/65536	111 111 001	111 011 011 011 011 000
Q	1/65536	111 111 010	111 011 011 011 011 001

Table 4.3: Codes for an exponential distribution.

blocks (which can be documents or logical blocks independent of documents). A space-time tradeoff is obtained by varying the block size. The price is that searches in the index may have to be complemented with sequential scanning. If blocks do not match documents, even single word searches have to be complemented with sequential scanning of the candidate blocks. It is also possible to perform phrase queries. In this case, the index can point to blocks where all the words appear, but only a sequential search can tell whether the phrase actually appears. As result, in the case of *block addressing indexes*, it is essential to be able of keeping the text blocks in compressed form and searching them without decompressing.

As it was introduced in previous Section, both Plain Huffman and Tagged Huffman techniques enable searching the compressed text without decompressing it. However, as shown no false matchings can occur in Tagged Huffman compressed text, but they can take place with Plain Huffman, therefore searches over Tagged Huffman codes are simpler and faster that those over Plain Huffman.

4.4.1 Searching Plain Huffman Code

Two basic search methods were proposed in [49].

The first technique is known as *plain filterless*. It handles a Huffman tree, such as the leaves are the words of the vocabulary. A preprocessing phase starts marking in the vocabulary those words that are being searched for (exact and complex searches are treated in the same way). In order to handle phrase patterns, a bit mask is also associated to each marked word. This bit mask indicates which element of the pattern matches that word (that is, the order of the word inside the pattern), and permit building a non-deterministic automaton which will enable recognizing a pattern.

After the preprocessing phase, the compressed text is explored one byte at a time. Each byte enables traversing the Huffman tree downwards. When a marked leaf is reached, its bit mask is sent to the automaton, which will move from a state to one another depending on the bit mask received. Figure 4.6 shows the pattern “red hot” can be found. (It is supposed that the codeword of “red” is [255], and the codeword of “hot” is [127][201][0].

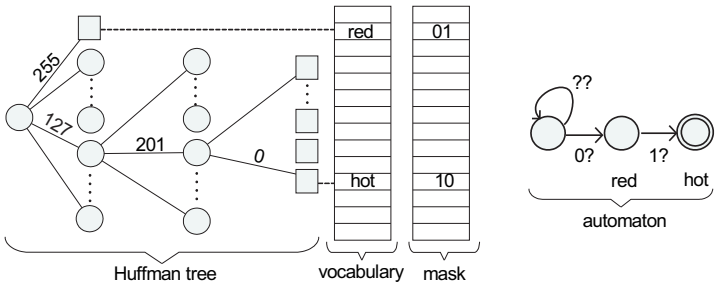


Figure 4.6: Searching Plain Huffman compressed text for pattern `"red hot"`

This is quite efficient, but not as much as the second choice, named *plain filter*, which compresses the pattern and uses any classical string matching strategy, such as Boyer-Moore [42]. For this second, faster, choice to be of use, one has to ensure that no spurious occurrences are found, so the *filterless algorithm* is applied in a region where the possible match was found. To avoid processing the text from the beginning, texts are divided in small blocks during compression, in such a way that no codeword crosses a block boundary. Therefore the filterless algorithm can start the validation of a match from the beginning of the block (blocks act as synchronization marks) where that match takes place. This algorithm also support complex patterns, that are searched by using a multi-pattern matching technique. However, its efficiency may be reduced because a large number of matches may have to be checked.

4.4.2 Searching Tagged Huffman Code

The algorithm to search for a pattern (a word, a phrase, etc.) under Tagged Huffman Code consists basically of two phases: compressing the pattern and then searching for it in the compressed text.

The first phase finds in the vocabulary all the words that compose the searched pattern. Then the compressed codeword/s for the pattern are created. If the pattern is not found in the vocabulary, then it cannot appear in the compressed text.

In the case of approximated or extended searches, each element of the

pattern (if there is more than one word) can be associated with several codewords from the vocabulary. To find all the words that match with the pattern in the vocabulary, a sequential search is performed, and a list of codewords is hold for each of the elements of the pattern.

The searching phase depends also on the type of search being performed. For an exact search, the codeword of the pattern is searched in the compressed text using any classical string matching algorithm with no modifications (i.e. the Boyer-Moore algorithm presented in Section 3.5.1). This technique is called *direct searching* [49, 62]. In the case of approximated or extended searches of one pattern, the problem of finding the codewords in the compressed text can be treated with a multi-pattern matching technique (as Baeza and Navarro described in [3]).

In the case of a phrase pattern two situations are possible: The simplest case consists of searching for simple words that compose a phrase, so their codewords are obtained and concatenated. Finally this large concatenated-codeword is searched as if it were a single codeword. The second situation takes place when elements of the phrase pattern are not single words, therefore more that one codeword can be associated to each element of that phrase pattern. The algorithm needs creating, for each one of the elements in the phrase pattern, a list with all their codewords. Then the search into the compressed text is performed to search for the elements of one of the lists. As an heuristic, it is chosen the list L_i associated to element i of the pattern, such as the minimal length of the codewords in L_i are maximized [6]. Each time a match occurs, the rest of lists are used to *validate* if that match belongs to an occurrence of the entire pattern. Note that the heuristic used, is based on the idea of choosing the list whose codewords are larger, hence those codewords correspond to low-frequency words, so the number of needed *validations* will be small.

Note that the efficiency of previous technique get worse when searching for frequent words, because in this situation the number of matches that occur is high and hence, lots of validations against the other lists have to be performed. In this case, Plain Huffman searching methods can result to be more suitable and fast.

In [49] they compare searching over Tagged Huffman and Plain Huffman

compressed text against searching the uncompressed text. Results show that searching a compressed text can be up to eight times faster than searching the uncompressed text for certain queries. Hence compression not only obtains space savings, but also improves the search speed over text collections. As result, keeping documents in compressed form and decompressing them only during the presentation process seems really interesting.

4.5 Other techniques

In this Section, two more techniques are presented. The first one is a simple compression technique, denominated *Byte Pair Encoding*, which offers competitive decompression speed (at the expense of its bad compression ratio). The second technique is *Burrows-Wheeler Transform*, an algorithm to transform a original text (or string) to another which is easier to compress.

4.5.1 Byte Pair Encoding

Byte Pair Encoding (BPE) [21], is a simple text compression technique based on pattern substitution. It is based on finding the most frequently occurring pairs of letters that stand adjacent in the text and then substituting them for a unique character which has not appeared previously in the text. This operation is repeated until: *i)* Either all the 256 possible values of a byte are used (hence no more substitutions can be done), or *ii)* no more frequent adjacent pairs can be found (that is, all pairs have the same number of occurrences).

Each time a substitution of a pair takes place, a hash table (also known as *substitution table*) is also modified either to add the new substitution pair or to increase is number of occurrences. The *substitution table* is necessary during decompression, hence the compressed file is composed of two parts: the *substitution table* and the packed data. An example of the compression process is explained in Figure 4.7.

This algorithm can be considered as multi-pass, since the number of passes over the text is not fixed, and depends on the two previous conditions.

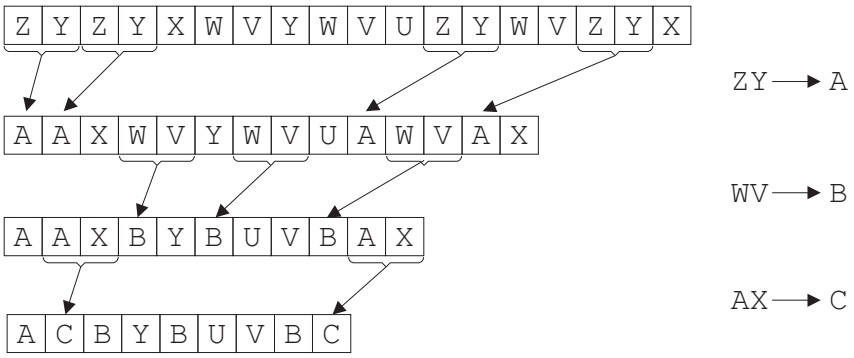


Figure 4.7: Compression process in Byte Pair Encoding

It requires that all the text being compressed, to be kept in memory during compression (so it can present memory problems with large texts). This technique could also present problems (bad compression ratio) when compressing large binary files. This is due to that in large binary files, the number of unused byte values could be small, and therefore the substitution process would finish prematurely.

Those two problems (memory usage and not enough free byte values) can be avoided by partitioning the text into blocks and then compressing each block separately. This presents the advantage that the *substitution table* can be locally adapted to the text in the block. However, for each compressed block, *substitution table* has to be included along with the packed data, what makes compression ratio to get worse.

The three main advantages of BPE stand in the high decompression speed reached (competitive with respect to *gzip*), in the possibility of partial decompression (which only needs to know the substitutions made during compression) and the fact that it is byte-oriented. These three properties make BPE very interesting when performing compressed pattern matching. In [47] two different approaches to search BPE compressed text are presented. Those are a brute force and a Shift-Or based technique.

The main disadvantages of BPE are: *i)* very slow compression, and bad compression ratio (it is worse than *compress* and *gzip*).

To improve compression speed, modifications of the initial algorithm [21] where proposed in [47], achieving better compression speed than *gzip* and *compress* at the expense of a small lose in compression ratio. Basically, they compute the *substitution table* for the first block of the text, and then they use it for the remainder blocks.

4.5.2 Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) was discovered by Wheeler in 1983, though it was not published until 1984 in [15]. It is not a compression technique, but an algorithm to transform a original text (or string) to another more compressible when using a compression technique.

This algorithm is reversible. Given a string S, BWT transforms it to a new string TS such as TS contains the same data that S, but in a different order, and from TS (and some extra information) an Inverse BWT enables recovering the original S source string.

Computing BWT

Let S be a string of length N. The algorithm starts building a N-order matrix $M_{N\times N}$, such as S appears in the first row, the second row contains $S >> 1$ (S circularly shifted one position to the right), and so on.

The second step consists of sorting alphabetically all the rows of the matrix, keeping track of the original position of the final rows. Note that one of the rows of the sorted matrix corresponds to the initial S string. Let call I the row containing the S string.

After the sorting, the first column of M if labelled F and the last one is labelled L. Two properties arise: *1)* F contains all characters in S, but now they are sorted alphabetically, and *2)* character j in L is the prefix (referring to S) of the string contained at row j.

The result of the BWT consists of the string composed of all characters in column L of M, and the I value. That is, $BWT(S) \rightarrow (L, I)$. An example of how applying BWT over a string 'abraca' is shown in Figure 4.8.

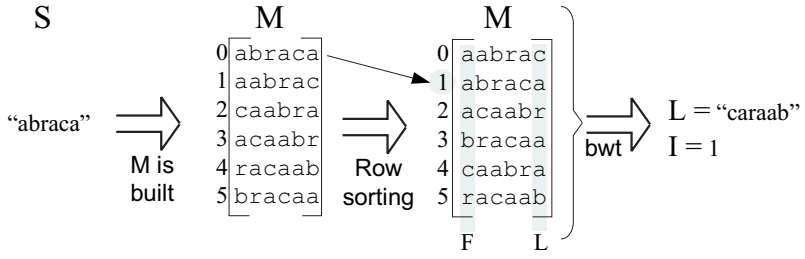


Figure 4.8: Direct Burrows-Wheeler Transform

Computing Inverse BWT

The inverse BWT (IBWT) algorithm uses the output (L, I) of the BWT algorithm to reconstruct its input, that is, the string S of length N. IBWT consists of three phases.

In the first phase, the column F of the matrix M is rebuilt. This is done by sorting alphabetically the L string. In the example shown in Figure 4.8, $L =$ `'caraab'`, hence, after sorting L it is obtained $F =$ `'aaabcr'`.

In the second phase, the strings F and L are used in order to calculate a vector T which indicates the correspondence between the characters of both the two strings. In such way, if L[j] is the kth occurrence of the character `'c'` in L, then $T[j] = i$, such as $F[i]$ is the kth occurrence of `'c'` in F. Therefore vector T represents a correspondence between the elements of F and the elements of L.

Example 4.5.1 Note that the first occurrence of 'c'$= L[0]$ in F happens in position 4, hence $T[0] = 4$. The first occurrence of 'a'$= L[1]$ is $F[0]$, therefore $T[1] = 0$, etc.

Position	0	1	2	3	4	5
L=	c	a	r	a	a	b
F=	a	a	a	b	c	r
T=	4	0	5	1	2	3

□

From the definition of T, it can also be seen that $F[T[j]] = L[j]$. This property is interesting because it will enable recovering the source string S.

In the last phase, the source text S is obtained. Using the index I, as well as vectors L and T, the process that permits to recover the text S starts performing:

$$\begin{aligned} p &\leftarrow I; \\ i &\leftarrow 0; \end{aligned}$$

Then N iterations are performed in order to recover the N elements of S as follows:

$$\begin{aligned} S_{N-i-1} &\leftarrow L[p]; \\ p &\leftarrow T[p]; \\ i &\leftarrow i+1; \end{aligned}$$

Example 4.5.2 Having L=‘`caraab`’, $T = [4, 0, 5, 1, 2, 3]$ and $I = 1$. The process starts with $p \leftarrow 1$ and $i \leftarrow 0$. Hence the first iteration makes $S_5 \leftarrow L[1] =$ ‘`a`’; $p \leftarrow T[1] = 0$; $i \leftarrow 1$. The second iteration makes $S_4 \leftarrow L[0] =$ ‘`c`’; $p \leftarrow T[0] = 4$; $i \leftarrow 2$. So the string S is built right-to-left. □

Using BWT in text compression

Once BWT has been applied over a source text S, and the transformed string L and the I value have been obtained, it is possible to compress L just applying a compression technique such as arithmetic encoding [57] or Huffman [27].

In decompression, the process starts decompressing the compressed data (using the same technique applied in compression) and then using the IBWT process to obtain the plain text.

However, even when it is possible to compress L with either an arithmetic or a Huffman based technique, results are better if a more specific encoder is

used. In [15], Burrows and Wheeler also showed how a "Move-to-Front" (MTF) encoder [11] could be used along with BWT to achieve better compression ratios. The idea consists of transforming the pair (L,I) to another pair (R,I) such as R is a vector of numbers.

To see why this might lead to effective compression, let consider the example of the letter `'t'` in the word `'the'`, and let assume an input English text S, containing many instances of `'the'`. When the rows in M are sorted, all those rows starting with `'he '` will appear together. A large proportion of them are likely to end in `'t'`. Hence, one region of the text L will contain a very large number of `'t'` characters, along with other characters that can precede `'he'` in English, such as space, `'s'`, `'T'`, and `'S'`. The same argument can be applied to all characters in all words, so any localized region of the string L will contain a large number of a few distinct characters.

The overall effect is that the probability that a character *ch* will occur at a given point in L is very high if *ch* occurs near that point in L, and is low otherwise. This property is exactly the one needed for effective compression by a MTF encoder [11], which encodes an instance of character *ch* by the count of distinct characters seen since the nearest previous occurrence of *ch*[1].

Summarizing, consecutive repetitions of a character will set consecutive zeros in the vector R, and consecutive repetitions of a small set of characters will produce an output dominated by low numbers. For example, for some sequences taken from Calgary corpus the percentage of zeros in the sequence R may reach 90%. On average, this sequence contains 60% zeros [10].

As result, the R sequence will be more compressible than the initial S text. It is usually compressed using either a Huffman based or an arithmetic encoder. However, since *zero* is the dominant symbol in R, there are many long runs in the sequence R consisting of zeros, so called *zero-runs*. This lead to Wheeler to propose another transform called *Zero-Run Transform* which is also known as RLE-0 to treat *0-runs* in a special way. It was not

[1]Basically, a MTF encoder keeps a list Z with all characters in the alphabet used. Each time a character *ch* is processed, *ch* is searched in Z, and the position j such as $Z_j = ch$ is output. Then Z_j is moved to the front of Z (by shifting the characters Z_i, $i = 0..j-1$ to the positions $1..j$ and setting $Z_0 = ch$).

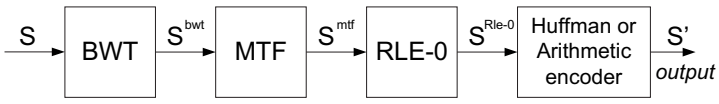


Figure 4.9: Whole compression using BWT, MTF and RLE-0

published by Wheeler but reported by Fenwick in [20]. Experimental results indicate [7] that the application of the *RLE-0 transform* indeed improves the compression ratio.

Figure 4.9 summarizes the whole compression process of a text S when applying BWT, MTF and RLE-0 transforms.

4.6 Summary

In this Chapter, an introduction to classical Huffman technique as well as a description of the mechanism to build a Huffman tree, and its representation via a canonical tree were shown. Special emphasis in the description of the two word-based Huffman techniques appeared in [48] was paid, since those techniques use the same approach that the codes we develop in this thesis and are presented in Chapters 5 and 6. In Section 4.4, the way Tagged Huffman and Plain Huffman compressed text can be direct searched was described.

Finally, two more techniques: Byte Pair Encoding, a compression scheme based on pattern substitution, and the Burrows-Wheeler Transform (a technique that is able to transform a text into another more compressible one) were presented.

5

End-Tagged Dense Code

5.1 Chapter overview

This Chapter explains in detail the End-Tagged Dense Code, the precursor of our work. This technique was developed in the Database Laboratory of the University of A Coruña in collaboration with Gonzalo Navarro in 2003. Even it is not a contribution of this thesis, it is deeply reviewed here because it is the basis of the technique presented in Chapter 6, the (s, c)-Dense Code, which constitutes the first real contribution presented in this thesis.

In Section 5.2, the motivation of the End-Tagged Dense Code is presented. Next, the way End-Tagged Dense Code works is described in Section 5.3. Later, encoding and decoding mechanisms are explained and pseudocode is given in Section 5.4. Finally analytical and empirical results, as well as some conclusions about End-Tagged Dense Code, are shown.

5.2 Introduction

End-Tagged Dense Code was first presented in [13]. This technique uses, as Plain and Tagged Huffman do, a semi-static word-based model, but it is not based on Huffman at all. It maintains the good features of Tagged Huffman

code:

- It is a *prefix code*[1].
- As Tagged Huffman code, it enables fast decompression of arbitrary portions of text, by using a flag bit in all the bytes that compose a codeword.
- It permits using Boyer-Moore type searching algorithms [12] directly on the compressed text[1].

End-Tagged Dense Code improves Tagged Huffman compression ratio. Besides, encoding and decoding processes are simpler and faster.

The remainder of this Chapter is organized as follows: First, End-Tagged Dense Code is defined. Next, encoding and decoding processes are explained. At last, empirical results which compare End-Tagged Dense Code against Plain Huffman and Tagged Huffman are also shown.

5.3 End-Tagged Dense Code

End-Tagged Dense Code [13] starts with a seemingly dull change to Tagged Huffman Code. Instead of using the flag bit to signal the *beginning* of a codeword, the flag bit is used to signal the *end* of a codeword. That is, the flag bit is *0* for the first bit of any byte of a codeword except for the last one, which has a *1* in its more significative bit.

Table 5.1 describes the differences of codewords in End-Tagged Dense Code and Tagged Huffman Code.

This change has surprising consequences. Now the flag bit is enough to ensure that the code is a prefix code regardless of the contents of the other 7 bits of each byte. To see this, consider two codewords X and Y, being X shorter than Y ($|X| < |Y|$). X cannot be a prefix of Y because the last byte

[1]Even when it is a *prefix code*, it is not a *suffix code* as Tagged Huffman is. Therefore, even when it permits Boyer-Moore type searching algorithms, when a match is found, a check over the previous byte is needed. A full explanation is presented in Page 90.

#Bytes	Tagged Huffman Code	End-Tagged Dense Code
1	**1**xxxxxxx	**1**xxxxxxx
2	**1**xxxxxxx **0**xxxxxxx	**0**xxxxxxx **1**xxxxxxx
3	**1**xxxxxxx **0**xxxxxxx **0**xxxxxxx	**0**xxxxxxx **0**xxxxxxx **1**xxxxxxx
...	...	...
n	**1**xxxxxxx **0**xxxxxxx ... **0**xxxxxxx	**0**xxxxxxx ... **0**xxxxxxx **1**xxxxxxx

Table 5.1: Format of codeword in both Tagged Huffman and End-Tagged Dense Code

of X has its flag bit in 1, while the $|X|$-th byte of Y has its flag bit in 0. This fact can be easily saw in Table 5.1, since a two-bytes codeword cannot start with a **1**xxxxxxx byte.

At this point, there is no need at all to use Huffman coding over the remaining 7 bits to get a *prefix code*. Therefore it is possible to use *all* the possible combinations of 7 bits in all the bytes, as long as the flag bit is used to mark the last byte of the codeword.

Encoding process is simpler and faster than Huffman, since no tree has to be built. Notice that it is not restricted to use symbols of 8 bits to form the codewords. It is possible to use symbols of b bits. In such a way End-Tagged Dense Code is defined as follows:

Definition 5.3.1 *Given source symbols with decreasing probabilities* $\{p_i\}_{0 \leq i < n}$ *the corresponding codeword using the End-Tagged Dense Code is formed by a sequence of symbols of b bits, all of them representing base-*(2^{b-1}) *digits (that is, from 0 to* $2^{b-1} - 1$*), except the last one which has a value between* 2^{b-1} *and* $2^b - 1$*, and the assignment is done in a completely sequential fashion.*

That is, using symbols of 8 bits, the encoding process can be described as follows:

- Words in the vocabulary are decreasingly ranked by number of occurrences.

Word rank	codeword assigned	# Bytes	# words
0	**1**0000000	1	
1	**1**0000001	1	
2	**1**0000010	1	2^7
...	...	...	
$2^7 - 1 = 127$	**1**1111111	1	
$2^7 = 128$	**0**0000000:**1**0000000	2	
129	**0**0000000:**1**0000001	2	
130	**0**0000000:**1**0000010	2	
...	...	...	
255	**0**0000000:**1**1111111	2	
256	**0**0000001:**1**0000000	2	$2^7 2^7$
257	**0**0000001:**1**0000001	2	
258	**0**0000001:**1**0000010	2	
...	...	...	
$2^7 2^7 + 2^7 - 1 = 16511$	**0**1111111:**1**1111111	2	
$2^7 2^7 + 2^7 = 16512$	**0**0000000:**0**0000000:**1**0000000	3	
16513	**0**0000000:**0**0000000:**1**0000001	3	
16514	**0**0000000:**0**0000000:**1**0000010	3	$(2^7)^3$
...	...	...	
$(2^7)^3 + (2^7)^2 + 2^7 - 1$	**0**1111111:**0**1111111:**1**1111111	3	
...	...	...	

Table 5.2: Code assignment in End-Tagged Dense Code

- Codewords 128 to 255 (**1**0000000 to **1**1111111) are given to the first 128 words in the vocabulary.
- Words ranked from 128 to 16511 are assigned sequentially to two-byte codewords. The first byte of each codeword has a value in the range $[0, 127]$ and the second in range $[128, 255]$.
- Word 16512 is assigned a tree-byte codeword, and so on, just as if we had a 21-bit number.

As it can be seen in Table 5.2, the computation of codes is extremely simple: It is only necessary to order the vocabulary words by frequency and then sequentially assign the codewords. Hence the coding phase will be faster than using Huffman because obtaining the codes is simpler.

What is perhaps less obvious is that *the code depends on the rank of the*

words, not on their actual frequency. That is, if we have four words *A, B, C, D* (ranked $i \ldots i+3$) with frequencies 0.24, 0.22, 0.22 and 0.20, respectively, then the code will be the same as if their frequencies were 0.9, 0.09, 0.009 and 0.001.

In Example 5.3.1, the differences among the codes generated by the Plain Huffman Code, Tagged Huffman Code and End-Tagged Dense Code, are shown. For the sake of simplicity we consider that the "*bytes*" used are formed by only three bits ($b = 3$).

Example 5.3.1 A vocabulary which has 17 words us used. Table 5.3 shows the codeword assigned to each word, assuming that the frequencies of those words in the text follow an uniform distribution ($p_i = 1/17$). In Table 5.4 an exponential distribution ($p_i = 1/2^i$) is assumed.

Assuming "*bytes*" of three bits, Tagged Huffman and End-Tagged Dense Code use one bit for the flag and two for the code (this makes it look worse than it is). Flag bits are shown underlined. Moreover, it can be seen that Plain Huffman uses the whole three bits to the code.

Note that in the case of End-Tagged Dense Code, the code assignment is the same independently of the distribution of frequencies of the words in the vocabulary. □

5.4 Encoding and Decoding algorithms

5.4.1 Encoding algorithm

As seen, Encoding process is really simple. In fact, it is not necessary to physically store the results of these computations: With a few operations we can obtain on-the-fly, given a word rank i, its ℓ-byte codeword, in $O(\ell) = O(\log i)$ time. However, in practice, it is faster to sequentially compute the codewords of the whole words in the vocabulary (in $O(n)$ time) in such a way that the assignment word-codeword is precomputed before the second pass of the compression starts.

Word	Probab.	Plain Huffman	Tagged Huffman	End-Tagged Dense Code
A	1/17	000	100 000	100
B	1/17	001	100 001	101
C	1/17	010	100 010	110
D	1/17	011	100 011	111
E	1/17	100	101 000	000 100
F	1/17	101	101 001	000 101
G	1/17	110 000	101 010	000 110
H	1/17	110 001	101 011	000 111
I	1/17	110 010	110 000	001 100
J	1/17	110 011	110 001	001 101
K	1/17	110 100	110 010	001 110
L	1/17	110 101	110 011	001 111
M	1/17	110 110	111 000	010 100
N	1/17	110 111	111 001	010 101
O	1/17	111 000	111 010	010 110
P	1/17	111 001	111 011 000	010 111
Q	1/17	111 010	111 011 001	011 100

Table 5.3: Codes for an uniform distribution.

Word	Probab.	Plain Huffman	Tagged Huffman	End-Tagged Dense Code
A	1/2	000	100	100
B	1/4	001	101	101
C	1/8	010	110	110
D	1/16	011	111 000	111
E	1/32	100	111 001	000 100
F	1/64	101	111 010	000 101
G	1/128	110	111 011 000	000 110
H	1/256	111 000	111 011 001	000 111
I	1/512	111 001	111 011 010	001 100
J	1/1024	111 010	111 011 011 000	001 101
K	1/2048	111 011	111 011 011 001	001 110
L	1/4096	111 100	111 011 011 010	001 111
M	1/8192	111 101	111 011 011 011 000	010 100
N	1/16384	111 110	111 011 011 011 001	010 101
O	1/32768	111 111 000	111 011 011 011 010	010 110
P	1/65536	111 111 001	111 011 011 011 011 000	010 111
Q	1/65536	111 111 010	111 011 011 011 011 001	011 100

Table 5.4: Codes for an exponential distribution.

Notice that there is no need to store the codewords (in any form such as a tree) nor the frequencies in the compressed file. It is enough to store the plain words sorted by frequency. Therefore, the vocabulary will be slightly smaller than in the case of the Huffman code, where some information about the shape of the tree must be stored (even when a canonical Huffman tree is used).

Even the codes are assigned in a sequential way, once words have been sorted by frequency in the vocabulary, a on-the-fly encode algorithm can also be used. Next, the pseudocode for the on-the-fly encode algorithm is shown. The algorithm outputs the bytes of each codeword one at a time from right to left. That is, it begins outputting the less significative bytes first.

```
Encode (i)
(1)  //input i: rank of the word being encoded 0 ≤ i ≤ n − 1
(2)  output ((i mod 128) + 128); //rightmost byte
(3)  i ← i div 128;
(4)  while i > 0 // remainder bytes
(5)       i ← i − 1;
(6)       output (i mod 128);
(7)       i ← i div 128;
```

5.4.2 Decoding algorithm

The first step to decompress a compressed text is to store the words that compose the vocabulary in a vector. Since these words were stored ranked by frequency along with the compressed text during compression, the vocabulary of the decompressor is already recovered ranked by frequency. Once the sorted vocabulary is obtained the decoding of codewords can begin.

In order to obtain the word w_i which corresponds to a given codeword c_i, the decoder can run a simple computation to obtain the rank of the word i from the codeword c_i. Then, using the value i, the corresponding word can be obtained from $vocabulary[i]$. A code c_i of ℓ bytes can be decoded in $O(\ell) = O(\log i)$ time.

The algorithm inputs a codeword x, and it iterates over each byte of

x. The last byte of the codeword has the tag byte to 1, so its value is greater than 127. This permits to distinguish the end of the codeword. After the iteration, a position i is returned, and the decoded word obtained from $vocabulary[i]$. Note that decoding a four bytes codeword $x = x_0x_1x_2x_3$ basically performs:

$$i = (((x_0 \times 128) + x_1) \times 128) + x_2) \times 128) + (x_3 - 128)$$

Decode (x)
(1) //input x: the codeword to be decoded
(2) //output i: the position of the decoded word in the ranked vocabulary
(3) $i \leftarrow 0$;
(4) $k \leftarrow 0$; // byte of the codeword
(5) **while** $x[k] < 128$
(6) $\quad i \leftarrow i \times 128 + x[k]$;
(7) $\quad k \leftarrow k + 1$;
(8) $i \leftarrow i \times 128 + x[k] - 128$;
(9) **return** i;

5.5 Using ETDC to bound Plain Huffman

An interesting property of this code is that it can be used as a bound for the compression that can be obtained with a Huffman code. It is clear that the End-Tagged Dense Code uses all the possible combinations of all bits, except the first one, that is used as a flag as in the Tagged Huffman Code. Therefore, calling D_b the code length of an End-Tagged Dense Code that uses symbols of b bits we have

$$D_{b+1} \leq H_b \leq D_b \leq T_b \leq D_{b-1}$$

Note that Figure 5.1 shows the shape of a Tagged Huffman tree. It is also shown how would be a tree represented by the codewords generated by End-Tagged Dense Code. As presented there, it can be seen that End-Tagged Dense Code achieves a smaller compressed text size than Tagged Huffman, since a internal node will have 256 children: 128 internal nodes and also 128

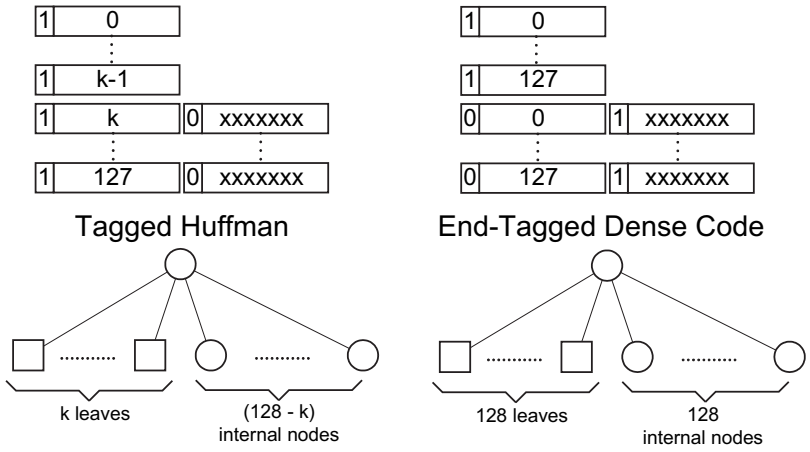


Figure 5.1: Comparison of Tagged Huffman and End-Tagged Dense Code

leaf nodes. However in Tagged Huffman a internal node cannot have more than 128 children, some of them k will be leaves and the other $(128 - k)$ will be internal nodes. Therefore End-Tagged Dense Code have, at least, the same number of codewords of a given length l (where l corresponds to a level in the tree).

In [13] a more precise comparison on these three codes is presented. There, it is shown how the End-Tagged Dense Code provides lower and upper bounds to the compression that can be obtained by Huffman with texts in natural language where the Zipf's Law is assumed [59]. We present a generalization of this bounds in the next Chapter. Using our compression method, the (s, c)-Dense Code even closer bound were obtained.

5.6 Empirical Results

Some large text collections from TREC-2 (AP Newswire 1988 and Ziff Data 1989-1990) and from TREC-4 (Congressional Record 1993, Financial Times 1991, 1992, 1993 and 1994) were used in the tests. Corpora were compressed using Plain Huffman, End-Tagged Dense Code and Tagged Huffman. The

spaceless word model [48] was used to create the vocabulary; that is, if a word was followed by a space, we just encoded the word, otherwise both the word and the separator were encoded.

The size of the compressed vocabulary was excluded from the results (this size is negligible and similar in all cases, although a bit smaller End-Tagged Dense Code because only the ranking of words is needed).

Table 5.5 shows the compression ratio obtained by the codes mentioned. The second column contains the original size of the processed corpus and the following columns indicate the number of words in the vocabulary, the θ parameter of the Zipf's Law [59, 6] and the compression ratio for each method.

Corpus	Original Size	Voc. Words	θ	P.H.	ETDC	T.H.
AP Newswire 1988	250,994,525	268,896	1.852045	31.18	32.00	34.57
Ziff Data 1989-1990	185,417,980	237,607	1.744346	31.71	32.60	35.13
Congress Record	51,085,545	117,713	1.634076	27.28	28.11	30.29
Financial Times 1991	14,749,355	75,687	1.449878	30.19	31.06	33.44
Financial Times 1992	175,449,248	284,878	1.630996	30.49	31.31	33.88
Financial Times 1993	197,586,334	291,404	1.647456	30.60	31.48	34.10
Financial Times 1994	203,783,923	294,490	1.649428	30.57	31.46	34.08

Table 5.5: Comparison of compression ratios.

As it can be seen, Plain Huffman gets the best compression ratio (as expected since it is the optimal prefix code) and End-Tagged Dense Codes always obtain better results than Tagged Huffman, with an improvement of up to 2.5 points. In fact, it is worse than the optimal Plain Huffman only by less than 1 point on average.

5.7 Conclusions

End-Tagged Dense Code, a simple and fast technique for compressing natural language texts databases, was presented. Its compression scheme is not based on Huffman at all, however it generates *prefix codes* by using a tag bit that indicates if a byte is the last byte of the codeword or not. Being a Tagged code, direct searching the compressed text, and also direct decompression are possible.

Empirical results (in compression ratio) show that the compression achieved is good. It achieves less than one percentage point of compression ratio overhead with respect to Plain Huffman, and improves Tagged Huffman's about 2,5 percentage points. Moreover, compression and decompression processes are faster, because the encode and decode algorithm are simpler. Empirical results about efficiency are provided in the next Chapter, using our generalized version of ETDC.

6

The new technique: (s, c)-Dense Code

6.1 Chapter overview

The first contribution of this thesis is presented in this Chapter. It consists of a new word-based byte-oriented *two-pass* technique denominated (s, c)-Dense Code, which generalizes the End-Tagged Dense Code technique described in the previous Chapter.

This Chapter is structured as follows: First, the key idea and the motivation of the (s, c)-Dense Code is shown. In Section 6.3 the new technique is defined and described. Next, procedures to obtain optimal s and c parameters are presented and encoding and decoding processes are explained via pseudocodigo. In Section 6.6, empirical results comparing (s, c)-Dense Code against End-Tagged Dense Code, Plain Huffman and Tagged Huffman are also shown. Finally some conclusions end this Chapter.

6.2 Introduction

End-Tagged Dense Code [13] has been described in Chapter 5. As shown this technique uses also a semi-static word-based model as the compression

techniques presented in [48, 49], but it is not based on Huffman at all.

It maintains the good features of Tagged Huffman code:

- It is a *prefix code*[1].
- It enables fast decompression of arbitrary portions of text.
- It permits to use use Boyer-Moore type searching algorithms [42], that is, skipping bytes, directly on the compressed text[1].

In Section 5.6 it is shown how End-Tagged Dense Code improves Tagged Huffman compression ratio in more than 2.5 % points. However, its difference with respect to Plain Huffman is about 1 point (in percentage).

As shown in Chapter 5, End-Tagged Dense Code uses 2^{b-1} digits, from 0 to $2^{b-1} - 1$, for the bytes at the beginning of a codeword, and it uses the other 2^{b-1} digits, from 2^{b-1} to $2^b - 1$, for the last byte of the codeword. The question that arises now is whether that proportion between the number of *non terminal* and *terminal* digits is the optimal one; that is, for a given corpus with a specific distribution of word frequencies, it might be that a different number of non terminal digits (*continuers*) and terminal digits (*stoppers*) could compress better than just using 2^{b-1}. This idea has been previously pointed out in [43], and it is the basic difference between End-Tagged Dense Code and (s, c)-Dense Code, as it will be presented later.

Example 6.2.1 introduces the advantages of using a variable -rather a fixed- number of stoppers and continuers.

Example 6.2.1 Let suppose that 5,000 distinct words compose the vocabulary of the text to compress. Suppose also that bytes are used to form the codewords, so $b = 8$.

If End-Tagged Dense Code is used, that is, if the number of stoppers and continuers is $2^7 = 128$, the maximum codeword length will be 2, since

[1]Even when it is a *prefix code*, it is not a *suffix code* as Tagged Huffman is. Therefore, even when it permits Boyer-Moore type searching algorithms, when a match is found, a check over the previous byte is needed. A full explanation is presented in Page 90.

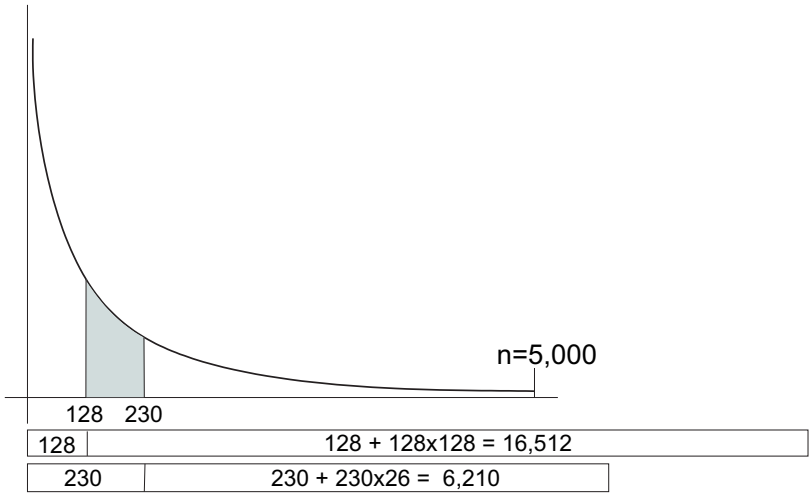


Figure 6.1: 128 versus 230 *stoppers* with a vocabulary of $5,000$ words

$128 + 128^2 = 16,512$ is the number of words that can be encoded with codewords of one or two bytes. Therefore, there are $16,512 - 5,000 = 11,512$ unused codewords of two bytes. In this situation, all the 128 one-byte codewords and $5,000 - 128 = 4,872$ two-bytes codewords are used.

If the number of stoppers chosen is 230 (so the number of continuers is $256 - 230 = 26$), then $230 + 230 \times 26 = 6,210$ words can be encoded with codewords of only one or two bytes. The first 230 words can be encoded with one-byte codewords, and the remainder $230 \times 26 = 5,980$ words with two-bytes codewords. Therefore the whole $5,000$ words can be assigned codewords of 1 or 2 bytes in the following way: the 230 most frequent words are assigned one-byte codewords and the remainder $5,000 - 230 = 4,770$ words are assigned two-bytes codewords.

It can be seen that words from 0 to 128 and words ranked from 231 to $5,000$ are assigned codewords of the same length in both schemes. However words from 129 to 230 are assigned shorter codewords when using 230 stoppers instead of only 128. Figure 6.1 describes this scenario. □

As result, it seems appropriated to adapt the number of stoppers and continuers to:

- The size of the vocabulary (n). It is possible to maximize the available number of short codewords depending on n.

- The distribution of frequencies of the words of the vocabulary. Intuitively, if that distribution presents a very steep slope (that is, if it is very biased), it can be desirable to increase the number of words that can be encoded with small codewords (so a high number of stoppers should be chosen). However this will imply that the less frequent words are encoded with larger codewords, what does not matter since the gain in the most frequent words compensate the loss of compression in the least frequent words.
  When that distribution is quite plain, it is preferred to reduce the largest-codeword size, in order not to loss compression in the least frequent words.

6.3 (s, c)-Dense Code

We define (s, c)- stop-cont codes as follows.

Definition 6.3.1 *Given source symbols with probabilities* $\{p_i\}_{0 \leq i < n}$ *an* (s, c) *stop-cont code (where* c *and* s *are integers larger than zero) assigns to each source symbol* i *a unique target code formed by a* s *plus base-*c *digit (from* s *to* $s + c - 1$*) sequence terminated by a base-*s *digit (between* 0 *and* $s - 1$*).*

It should be clear that a stop-cont coding is just a base-c numerical representation, but adding s to each digit, with the exception that the last digit is between 0 and $s - 1$. Digits between s and $s + c - 1$ are called "continuers" and those between 0 and $s - 1$ are called "stoppers" . The next property clearly follows.

Property 6.3.1 *Any* (s, c) *stop-cont code is a prefix code.*

Proof 6.3.1 *If one code were a prefix of the other, since the shorter code must have a final digit of value lower than* s*, then the longer code must have*

an intermediate digit which is not in base c plus s. This is a contradiction. □

Among all the possible (s, c) stop-cont codes for a given probability distribution, the *dense code* is one that minimizes the average codeword length. This is because a dense code uses all the possible combinations of bits in each byte. That is, codes can be assigned sequentially to the ranked symbols.

Definition 6.3.2 *Given source symbols with* decreasing *probabilities* $\{p_i\}_{0 \leq i < n}$*, the corresponding* (s, c)*-Dense Code (*(s, c)*-DC) is an* (s, c) *stop-cont code where the codewords are assigned as follows: Let* k *be the number of bytes in each codeword, which is always* ≥ 1*, then* k *will be such that*

$$s\frac{c^{k-1} - 1}{c - 1} \leq i < s\frac{c^k - 1}{c - 1}$$

Thus, the code corresponding to source symbol i is formed by $k - 1$ s plus base-c digits and a final base-s digit. If $k = 1$ then the code is simply the stopper i. Otherwise the code is formed by the number $\lfloor x/s \rfloor$ written in base c, and adding s to each digit (they are base-c digits which are then increased by s), followed by $x \; mod \; s$, where $x = i - \frac{sc^{k-1} - s}{c-1}$.

That is, using symbols of 8 bits ($b = 8$), the encoding process can be described as follows:

- One-byte codewords from 0 to $s - 1$ are given to the first s words in the vocabulary.
- Words ranked from s to $s + sc - 1$ are assigned sequentially two-bytes codewords. The first byte of each codeword has a value in the range $[s, s + c - 1]$ and the second in range $[0, s - 1]$.
- Words from $s + sc$ to $s + sc + sc^2 - 1$ are assigned tree-bytes codewords, and so on. Table 6.1 presents this process.

Lets see another example of how codewords are assigned.

Word rank	codeword assigned	# Bytes	# words
0	[0]	1	
1	[1]	1	
2	[2]	1	s
...	...	...	
$s-1$	[s-1]	1	
s	[s][0]	2	
$s+1$	[s][1]	2	
$s+2$	[s][2]	2	
...	...	...	
$s+s-1$	[s][s-1]	2	sc
$s+s$	[s+1][0]	2	
$s+s+1$	[s+1][1]	2	
...	...	...	
$s+sc-1$	[s+c-1][s-1]	2	
$s+sc$	[s][s][0]	3	
$s+sc+1$	[s][s][1]	3	
$s+sc+2$	[s][s][2]	3	sc^2
...	...	...	
$s+sc+sc^2-1$	[s+c-1][s+c-1][s-1]	3	
...	...	...	

Table 6.1: Code assignment in (s, c)-Dense Code

Example 6.3.1 The codes assigned to symbols $i \in 0 \ldots 15$ by a (2,3)-DC are as follows: $\langle 0 \rangle$, $\langle 1 \rangle$, $\langle 2,0 \rangle$, $\langle 2,1 \rangle$, $\langle 3,0 \rangle$, $\langle 3,1 \rangle$, $\langle 4,0 \rangle$, $\langle 4,1 \rangle$, $\langle 2,2,0 \rangle$, $\langle 2,2,1 \rangle$, $\langle 2,3,0 \rangle$, $\langle 2,3,1 \rangle$, $\langle 2,4,0 \rangle$, $\langle 2,4,1 \rangle$, $\langle 3,2,0 \rangle$ and $\langle 3,2,1 \rangle$. □

Note that the code does not depend on the exact symbol probabilities, but just on their ordering by frequency. We now prove that the dense coding is an optimal stop-cont coding.

Property 6.3.2 *The average length of a (s, c)-dense code is minimal with respect to any other (s, c) stop-cont code.*

Proof 6.3.2 *Let us consider an arbitrary (s, c) stop-cont code, and let us write all the possible codewords in numerical order, as in Example 6.3.1, together with the symbol they encode, if any. Then it is clear that (i) any unused code in the middle could be used to represent the source symbol with*

longest codeword, hence a compact assignment of target symbols is optimal; and (ii) if a less probable symbol with a shorter code is swapped with a more probable symbol with a longer code then the average code length decreases, and hence sorting the symbols by decreasing frequency is optimal. □

Since sc^{k-1} different codewords can be coded using k digits, let us call

$$W_k^s = \sum_{j=1}^{k} sc^{j-1} = s\frac{c^k - 1}{c - 1}$$

(where $W_0^s = 0$) the number of source symbols that can be coded with up to k digits. Let us also call

$$f_k^s = \sum_{j=W_{k-1}^s+1}^{W_k^s} p_j$$

the sum of probabilities of source symbols coded with k digits by an (s, c)-DC.

Then, the average codeword length, $LD_{(s,c)}$, for the (s, c)-DC is

$$\begin{aligned} LD_{(s,c)} &= \sum_{k=1}^{K^s} k f_k^s = \sum_{k=1}^{K^s} k \sum_{j=W_{k-1}^s+1}^{W_k^s} p_j \\ &= 1 + \sum_{k=1}^{K^s-1} k \sum_{j=W_k^s+1}^{W_{k+1}^s} p_j = 1 + \sum_{k=1}^{K^s-1} \sum_{j=W_k^s+1}^{W_k^s} p_j \end{aligned}$$

where $K^x = \lceil \log_{(2^b-x)} \left(1 + \frac{n(2^b-x-1)}{x}\right) \rceil$, and n is the number of symbols in the vocabulary.

It is clear from Definition 6.3 that the End-Tagged Dense Code is a $(2^{b-1},2^{b-1})$-DC and therefore *(s,c)*-DC can be seen as a generalization of the End-Tagged Dense Code where s and c are adjusted to optimize the compression for the distribution of frequencies and the size of the vocabulary.

As it is shown in [13] it is proved that $(2^{b-1},2^{b-1})$-DC is more efficient than Tagged Huffman. This is because Tagged Huffman is a $(2^{b-1},2^{b-1})$ *(non dense) stop-cont* code, while the End-Tagged Dense Code is a $(2^{b-1},2^{b-1})$-Dense Code.

Rank	Word	Freq	PH	(6,2)-DC	ETDC	TH	Freq × bytes			
							PH	(6,2)-DC	ETDC	TH
1	A	0.200	[0]	[0]	[4]	[4]	0.20	0.20	0.20	0.20
2	B	0.200	[1]	[1]	[5]	[5]	0.20	0.20	0.20	0.20
3	C	0.150	[2]	[2]	[6]	[6]	0.15	0.15	0.15	0.15
4	D	0.150	[3]	[3]	[7]	[7][0]	0.15	0.15	0.15	0.30
5	E	0.140	[4]	[4]	[0][4]	[7][1]	0.14	0.14	0.28	0.28
6	F	0.090	[5]	[5]	[0][5]	[7][2]	0.09	0.09	0.18	0.18
7	G	0.040	[6]	[6][0]	[0][6]	[7][3][0]	0.04	0.08	0.08	0.12
8	H	0.020	[7][0]	[6][1]	[0][7]	[7][3][1]	0.04	0.04	0.04	0.06
9	I	0.005	[7][1]	[6][2]	[1][4]	[7][3][2]	0.01	0.01	0.01	0.015
10	J	0.005	[7][2]	[6][3]	[1][5]	[7][3][3]	0.01	0.01	0.01	0.015
					average codeword length		**1.03**	**1.07**	**1.30**	**1.52**

Table 6.2: Comparative example among compression methods, for b=3

Example 6.3.2 Table 6.2 shows the codewords assigned to a small set of words ordered by their frequency when using Plain Huffman (P.H.), *(6,2)*-DC, End-Tagged Dense Code (ETDC) which is a *(4,4)*-DC, and Tagged Huffman (TH). Digits of three bits (instead of bytes) are used for simplicity (*b=3*), and therefore $s + c = 8$. The last four columns present the products of the number of bytes by the frequency for each word, and its addition, the average codeword length, is shown in the last row.

It is easy to see that, for this example, Plain Huffman and the (6,2)-Dense Code are better than the (4,4)-Dense Code (ETDC) and therefore they are also better than Tagged Huffman. Notice that (6,2)-Dense Code is clearly better than (4,4)-Dense Code because it takes advantage of the distribution of frequencies and of the number of words in the vocabulary. However the values (6,2) for s and c are not the optimal ones since a (7,1)-Dense Code obtains an optimal compressed text having, in this example, the same result than Plain Huffman. □

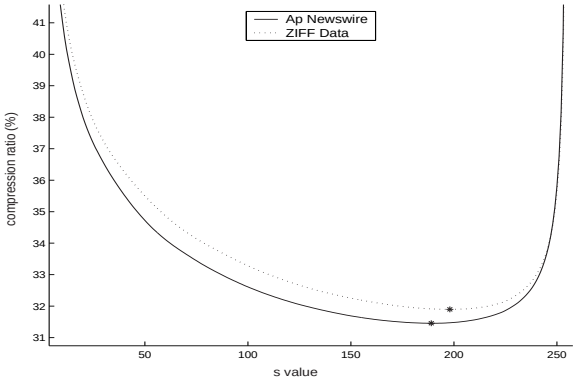
The problem now consists of finding the s and c values (assuming a fixed b where $2^b = s + c$) that minimize the size of the compressed text.

6.4 Optimal s and c values

The key advantage of this method with respect to End-Tagged Dense Code is the ability to use the optimal s,c values. In all the real text corpora from

TREC[2] used in our experiments, the size of the compressed text, as a function of s, has only one local minimum. See Figures 6.2 and 6.3, where optimal s values are shown for some real corpora, as well as the curves where it can be seen that a unique minimum exists.

In Figure 6.2 the size of the compressed texts and the compression ratios are shown as a function of the s values, for Ziff and Ap-Newswire texts. As it is also shown, in the Table of Figure 6.2, the optimal s value for Ziff corpus is 198, while for the AP-Newswire corpus the maximum compression ratio is achieved with $s = 189$. In the bottom table, we show sizes and compression ratios when s values close to the optimum are used over these two corpus.



s value	Ap Newswire Corpus		ZIFF Corpus	
	ratio(%)	size(bytes)	ratio(%)	size(bytes)
186	31.4575	78,956,616	31.9234	59,191,693
187	31.4565	78,954,060	31.9192	59,183,930
188	31.4558	78,952,303	31.9156	59,177,207
189	**31.4554**	**78,951,377**	31.9122	59,171,003
190	31.4555	78,951,674	31.9092	59,165,319
191	31.4560	78,952,893	31.9064	59,160,290
192	31.4569	78,955,045	31.9041	59,155,863
195	31.4622	78,968,414	31.8990	59,146,488
196	31.4650	78,975,466	31.8981	59,144,824
197	31.4684	78,983,892	31.8976	59,143,905
198	31.4722	78,993,433	**31.8976**	**59,143,837**
199	31.4764	79,004,086	31.8980	59,144,550
200	31.4813	79,016,243	31.8987	59,146,012
201	31.4866	79,029,758	31.9000	59,148,335

Figure 6.2: Compressed text sizes and compression ratios for different s values.

[2]Text REtrieval Conference (TREC) is an international conference where standard and well-known real corpora are used to test Text Retrieval Systems.

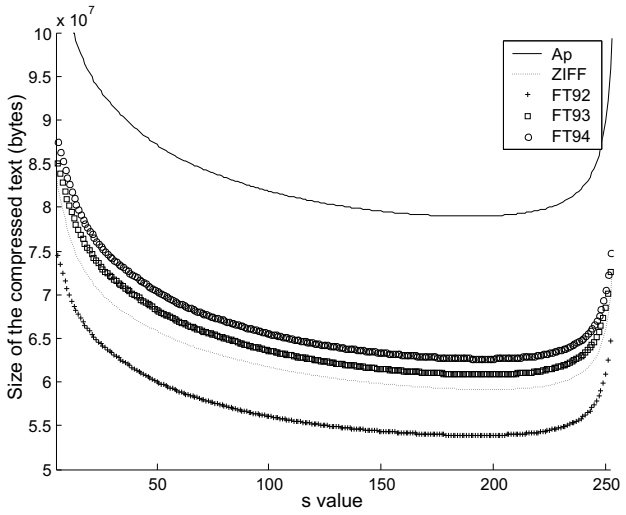


Figure 6.3: Size of the compressed text for different s values

This property (that is used as a heuristic) permits a binary search for the optimal s value (and $c = 2^b - s$). We have produced, however, artificial distributions where more than one local minima exist. Hence, although a sequential search for the optimal s,c values is theoretically necessary, a binary search will, in real cases, find the best s and c values.

When a unique minimum exists, the size of the compressed text decreases when s is increased, until reaching the *unique* optimal s value. After that optimal s value, increments of s produce a loss in the compression ratio. This is shown in Figure 6.3. Of course the value of c depends on the value of s because $c = 2^b - s$ always holds.

Intuitively, it is easy to see that when s is very small the number of high frequency words encoded with very few bytes (that is, one or two bytes) is also very small (s words are encoded with just one byte and $s \cdot c$ with two bytes) but in this case c is large and therefore words with low frequency will be encoded with few bytes ($s \cdot c^2$ words will be encoded with 3 bytes, $s \cdot c^3$ with 4 bytes and so on, but if c is so large, probably 3 bytes will be enough to encode the last word of the ranked vocabulary).

It is clear that, as s grows, highest frequency words will be encoded with

less bytes, so we improve the compression of high frequency words. But at the same time, as s grows, lowest frequency words will need more bytes to be encoded, so we loss compression in those words.

As consequence, if we try all the possible values of s starting at $s = 1$, we will see (as in Figure 6.3) that, in the beginning, compression improves a lot because each increment of s produce that words with high frequency become encoded by a codeword that is one byte shorter.

When s becomes larger, for each increment of s the number of words encoded with less bytes is smaller in proportion and these words have lower frequency. Therefore, with each increment of s, we gain less and less compression in the highest frequency words. At the same time, we lose more and more compression in the lowest frequency words, because with each increment of s they will need more bytes to be encoded. At some point, the compression lost in the last words is larger than the compression gained in words at the beginning, and therefore the global compression ratio decreases. That point gives us the optimal s value. It is easy to see in Figure 6.3 that, around of the optimal value, the compression is relatively insensitive to the exact value of s. This fact causes the smooth bottom of the curve.

The binary search algorithm takes advantage of this property. It is not necessary to check all the values of s because the shape of the distribution of compression ratios as a function of s is known. Therefore it leads the search towards the area where compression ratio improves.

6.4.1 Algorithm to get the optimal s and c values

This Section present both the binary and sequential algorithms developed to obtain the optimal c and c values for a given vocabulary.

Both algorithms *BinaryFindBestS()* and *SequentialFinBestS()* need to know the size (in bytes) of the compressed text for any given s and c values, in order to chose the best. This size is computed by another algorithm called *computeSizeS()*.

ComputeSizeS() uses a list of accumulated frequencies *acc*() previously computed to efficiently get the size of the compressed text. The size

of the compressed text can be computed for any given s and c values, as follows: $size_{(bytes)} = acc(s) + \sum_{k \geq 2} \; k * \big(acc(sc^{k-1}) - acc(sc^{k-2})\big) = acc(n) + \sum_{i \geq 0} \big(acc(n) - acc(sc^{i})\big)$. The pseudocode of *computeSizeS()* is next addressed:

---

computeSizeS (s, c, acc)
(1) //inputs: s, c and acc, the vector of accumulated frequencies
(2) //output: the length of the compressed text using s and c
(3) $k \leftarrow 1$; $n \leftarrow$ *number ofpositions in vector* $'acc'$;
(4) $Right \leftarrow min(s, n)$;
(5) $total \leftarrow acc[Right - 1]$;
(6) **while** $Right < n$
(7) $\quad Left \leftarrow Right$;
(8) $\quad Right \leftarrow Right + sc^{k}$;
(9) $\quad k \leftarrow k + 1$;
(10) $\quad$ **if** $Right > n$ **then**
(11) $\quad\quad Right \leftarrow n$;
(12) $\quad total \leftarrow total + k * (acc[Right - 1] - acc[Left])$;
(13) **return** $total$;

Notice that computing the size of the compressed text for a specific value of s costs $O(\log_c n)$, except for $c = 1$, in which case it costs $O(n/s) = O(n/2^b)$.

Sequential search

Sequentially searching the best s and c values consist on computing the size of the compressed text for each possible s value and then choosing the s value that minimizes the compressed text size.

SequentialFindBestS (b, acc)
(1) //inputs: b value ($2^b = c + s$) and acc, the vector of accumulated frequencies
(2) //output: The best s and c values
(3) $sizeBestS \leftarrow \infty$;
(4) **for** $i = 1$ **to** $2^b - 1$
(5) $\quad sizeS \leftarrow computeSizeS(i, 2^b - i, acc)$;

```
(6)        if sizeS < sizeBestS then
(7)            bestS ← i;
(8)            sizeBestS ← sizeS;
(9)  bestC ← 2^b − bestS;
(10) return bestS, bestC;
```

Since this algorithm call *computeSizeS()* for each $s \in \{1 .. 2^b - 1\}$, sequential search cost is

$$O\left(\frac{n}{2^b} + \sum_{i=2}^{2^b-1} \log_i n\right) = O\left(\frac{n}{2^b} + \log(n) \sum_{i=2}^{2^b-1} \frac{1}{log(i)}\right) = O\left(\frac{n}{2^b} + 2^b \log(n)\right)$$

The other operations of the sequential search are constant, and we have also an extra $O(n)$ cost to compute the accumulated frequencies. Hence the overall cost to find s and c is $O(n + \frac{n}{2^b} + 2^b \log(n)) = \mathrm{O}(n)$, supposing a previously sorted vocabulary.

Binary search

It consists of a binary search algorithm that, using the *ComputeSizeS()* function, computes the size of the compressed text for two consecutive values of s in the middle of the interval that is checked in each iteration. Initially these two points are: $(\lfloor 2^{b-1} \rfloor - 1$ and $\lfloor 2^{b-1} \rfloor)$. Then the algorithm (using the heuristic of the existence of a unique minimum) can lead the search to the point that reaches the best compression ratio. In each new iteration, the search space is reduced by half and a new computation of the compression that is obtained with the two central points of the new interval is performed. Finally, the s and c values that minimize the length are returned.

```
BinaryFindBestS (b, acc)
(1) //inputs: b value (2^b = c + s) and acc, the vector of accumulated frequencies
(2) //output: The best s and c values
(3) Lp ← 1;              //Lp and Up the lower and upper
```

```
(4)  Up ← 2^b − 1;        //points of the interval being checked
(5)  while Lp + 1 < Up
(6)       M ← ⌊(Lp+Up)/2⌋;
(7)       sizePp ← computeSizeS(M − 1, 2^b − (M − 1), acc); //size with M − 1
(8)       sizeM ← computeSizeS(M, 2^b − M, acc);            //size with M
(9)       if sizePp < sizeM then
(10)           Up ← M − 1;
(11)      else Lp ← M;
(12) if Lp < Up then    //Lp = Up − 1 and M = Lp
(13)      sizeNp ← computeSizeS(Up, 2^b − Up, acc); //size with M + 1
(14)      if sizeM < sizeNp then
(15)           bestS ← M;
(16)      else bestS ← Up;
(17) else bestS ← Lp;    //Lp = Up = M − 1
(18) bestC ← 2^b − bestS;
(19) return bestS, bestC;
```

Notice that computing the size of the compressed text for a specific value of s costs $O(\log_c n)$, except for $c = 1$, in which case it costs $O(n/s) = O(n/2^b)$. Hence the most expensive possible sequence of calls to **computeSizeS** in a binary search is that for values $c = 2^{b-1}$, $c = 2^{b-2}$, $c = 2^{b-3}$, ..., $c = 1$. The total cost of **computeSizeS** over that sequence of c values is

$$\frac{n}{2^b} + \sum_{i=1}^{b-1} \log_{2^{b-i}} n \;=\; \frac{n}{2^b} + \log_2 n \sum_{i=1}^{b-1} \frac{1}{b-i} \;=\; O\left(\frac{n}{2^b} + \log n \log b\right)$$

The other operations of the binary search are constant, and we have also an extra $O(n)$ cost to compute the accumulated frequencies. Hence the overall cost to find s and c is $O(n + \log(n)\log(b))$. Since the maximum b of interest is such that $b = \lceil \log_2 n \rceil$ (as at this point we can code each symbol using a single stopper), the optimization algorithm costs at most $O(n + \log(n)\log\log(n)) = O(n)$, assuming the vocabulary is already sorted. We have succeeded in making the part that computes the optimal s and c values totally negligible.

Comparing with Huffman algorithm is also linear once the vocabulary is sorted, but the constant is in practice larger because it involves more operations than just adding up frequencies.

Comparing the cost of computing the optimal s and c values against the process of building a Plain Huffman tree optimally [35], it can be seen that both processes run in linear time $O(n)$ once the vocabulary is sorted. However, building a Huffman tree[3] takes $O([n + n/2^b] + [n/2^b] + [n/2^b + n])$, while computing the optimal s and c values takes only $O(n + \log(n) \log\log(n))$. As result, (s, c)-Dense Code encoding is faster than Plain Huffman encoding process.

6.5 Encoding and Decoding algorithms

Once the optimal s and c values for a given vocabulary are known, it is possible to perform the code generation process. This encoding is usually done in a sequential fashion as shown in Table 6.1. However an on-the-fly encoding process is also available as happened in End-Tagged Dense Code. Given a word rank i, its ℓ-byte codeword, can be easily computed in $O(\ell) = O(\log i)$ time.

Hence, there is no need to store the codewords (in any form such as a tree) nor the frequencies in the compressed file. It is enough to store the plain words sorted by frequency and the s value used in the compression process. Therefore, the vocabulary will be slightly smaller than in the case of the Huffman code, where some information about the shape of the tree must be stored (even when a canonical Huffman tree is used).

6.5.1 Encoding algorithm

Even though the encoding process which assigns a codeword to each word in the ordered vocabulary, is performed in a sequential way, a on-the-fly encoding is also possible.

The following pseudocode presents the on-the-fly encode algorithm. The

[3] In order to be able to perform a canonical Huffman encoding, it is not needed to maintain a Huffman tree in memory. It is only needed to calculate the number of leaves that will appear in each level of the tree. That is, the number of codewords of 1, 2, 3, ... bytes that will be generated.

algorithm outputs the bytes of each codeword one at a time from right to left. That is, it begins outputting the less significative bytes first.

```
Encode (i)
(1) //input: i, the rank of the word in the vocabulary
(2) //outputs: the codeword C_i from right to left
(3) output i mod s;
(4) x ← i/s;
(5) while x > 0
(6)       x ← x − 1;
(7)       output(x mod c) + s;
(8)       x ← x/c;
```

6.5.2 Decoding algorithm

The first step of decompression is to store the words that compose the vocabulary in a vector. Since these words were saved ranked by frequency along with the s value and the compressed text during compression, the vocabulary of the decompressor is already recovered ranked by frequency. Once the sorted vocabulary is loaded, the decoding of codewords can begin.

In order to obtain the word w_i which corresponds to a given codeword c_i the decoder can run a simple computation to obtain, from the codeword, the rank of the word i. Then, using the value i, it obtains the word from the vocabulary sorted by frequency. A code c_i of ℓ bytes can be decoded in $O(\ell) = O(\log i)$ time as follows:

The decoder uses a *base* table. This table indicates the rank of the first word of the vocabulary that is encoded with k bytes ($k \geq 1$). Therefore $base[1] = 0$, $base[2] = s$, $base[3] = s + sc$,... $base[k] = base[k-1] + sc^{k-2}$.

The *decode* algorithm inputs a codeword x, and it iterates over each byte of x. The end of the codeword can be easily recognized because its value is smaller than s. After the iteration, the value i holds the relative position of the word w_i among all the words of k bytes. Then the *base* table is used, and $i \leftarrow i + base[k]$ is performed. As result, a position i is returned, and the decoded word w_i is obtained from $vocabulary[i]$.

```
Decode (base,x)
(1)  //inputs: x, the codeword to be decoded and the base table
(2)  //output: i, the position of the decoded word in the ranked vocabulary
(3)  i ← 0;
(4)  k ← 1; // number of bytes of the codeword
(5)  while x[k] ≥ s
(6)       i ← i × c + (x[k] − s);
(7)       k ← k + 1;
(8)  i ← i × s + x[k];
(9)  i ← i + base[k];
(10) return i;
```

6.6 Empirical Results

We used some large text collections from TREC-2 (AP Newswire 1988 and Ziff Data 1989-1990) and from TREC-4 (Congressional Record 1993, Financial Times 1991, 1992, 1993 and 1994). We compressed them applying Plain Huffman (P.H.), *(s,c)*-Dense Code ((s,c)-DC), End-Tagged Dense Code (ETDC) and Tagged Huffman (T.H.), using bytes ($b = 8$) as the symbols of the codewords. We used the spaceless word model [49] to create the vocabulary, that is, if a word was followed by a space, we just encoded the word, otherwise both the word and the separator were encoded.

6.6.1 Compression Ratio

We excluded the size of the compressed vocabulary in the results (this size is negligible and similar in all cases, although a bit smaller in (s,c)-DC and ETDC because only the sorted words are needed).

Table 6.3 shows the compression ratio obtained by the different codes. The second column contains the original size of the processed corpus, the third the number of words in the vocabulary, and the following columns give the compression ratio for each method. The fifth column, which refers to (s,c)-DC, also gives the optimal (s,c) values.

As it can be seen in Table 6.3, Plain Huffman gets the best compression

Corpus	Original Size	n	P.H.	(s,c)-DC	ETDC	T.H.
AP Newswire 1988	250,994,525	268,896	31.18	(189,67) 31.46	32.00	34.57
Ziff Data 1989-1990	185,417,980	237,607	31.71	(198,58) 31.90	32.60	35.13
Congress Record	51,085,545	117,713	27.28	(195,61) 27.50	28.11	30.29
Financial Times 1991	14,749,355	75,687	30.19	(193,63) 30.44	31.06	33.44
Financial Times 1992	175,449,248	284,878	30.49	(193,63) 30.71	31.31	33.88
Financial Times 1993	197,586,334	291,404	30.60	(195,61) 30.79	31.48	34.10
Financial Times 1994	203,783,923	294,490	30.57	(195,61) 30.77	31.46	34.08

Table 6.3: Comparison of compression ratios.

ratio (as expected since it is the optimal prefix code) and End-Tagged Dense Codes always obtain better results than Tagged Huffman, with an improvement of up to 2.5%. As expected, (s, c)-DC improves the results reached by ETDC ($(128, 128)$-DC), and it is worse than the optimal Plain Huffman only by less than 0.5% on average.

6.6.2 Encoding Time

As shown in the previous section, (s, c)-DC compression ratios are very close to Plain Huffman ones. In this section we compare (s, c)-DC and Plain Huffman encoding phase and measure code generation time.

The model used for compressing a corpus in our experiments is described in Figure 6.4. Three main phases arise.

1. The first phase is *vocabulary extraction*. The corpus is processed once in order to obtain all distinct words in it (n) and their number of occurrences. The result is a list of pairs $(word, frequency)$, which is then sorted by $frequency$. This phase is identical for both Plain Huffman and (s, c)-Dense Code.

2. In the second phase, *encoding*, each word in the vocabulary is assigned a codeword that minimizes average codeword length. This process is done in a different way for each method:

   - Plain Huffman encoding phase is split into two main parts: *Creating the Huffman tree* uses the Huffman algorithm to build

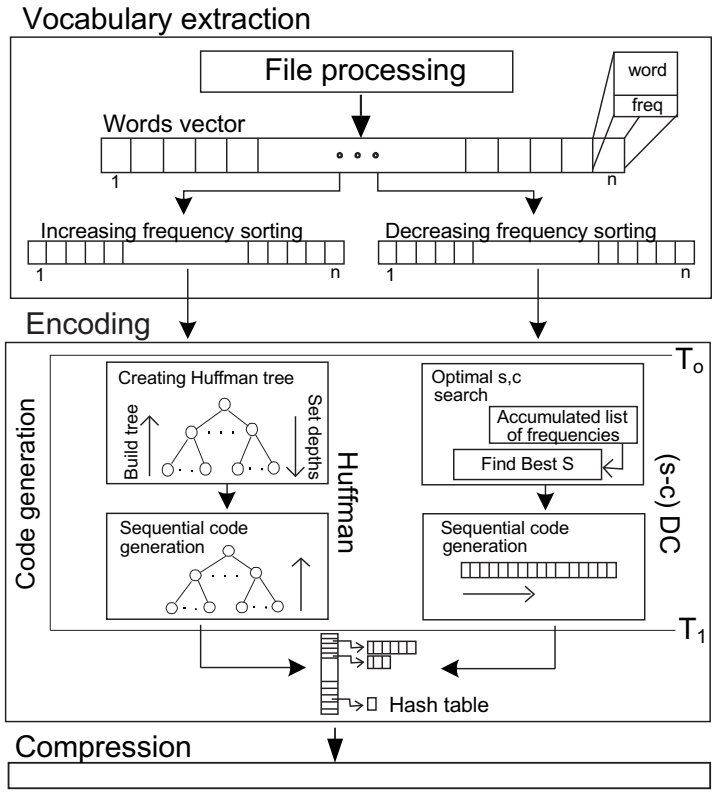


Figure 6.4: Vocabulary extraction and encoding phases

a tree where each leaf corresponds to one of the n words in the vocabulary, and the number of internal nodes is at most $\lceil \frac{n}{2^b-1} \rceil$. Then, starting from the root of the tree, the depth of each leaf is computed. Further details of this first part can be found in [35, 39]. *Code assignment* starts at the bottom of the tree (longest codewords) and goes through all leaf nodes. Nodes in the same level are given codewords sequentially, and a jump of level is determined by using the previously computed leaf depths. During this process, two vectors $base$ and $first$, needed for fast decompression, are also set: $base[l] = x$ if x is the first node of the l^{th} level, and $first[l] = y$ if y is the first codeword of l bytes. Encoding takes $O(n)$ time overall.

- (s, c)-DC encoding phase has also two parts: The first computes

the list of accumulated frequencies and searches for the optimal s and c values. Its cost is $O(n + 2^b \log(n)) = O(n)$. After getting the optimal s and c values, (s, c) *sequential encoding* is performed. The overall cost is $O(n)$.

In both cases, the result of the encoding section is a *hash table* of pairs $(word, codeword)$.

3. The third phase, *compression*, processes again the whole source text. For each word input, compression looks for it inside the *hash table* and outputs its corresponding codeword.

Since vocabulary extraction (previous to code generation), and building the hash table of pairs and compression (after code generation) are common phases for Plain Huffman and (s, c)-DC methods, we only measured code generation time ($T_1 - T_0$ in Figure 6.4), to compare both methods.

We also included other text collections such as the Calgary corpus (a very small one), and two larger collections: ALL_FT aggregates corpuses FT91, FT92, FT93 and FT94, and ALL is composed by Calgary corpus and all texts from TREC-2 and TREC-4. We compressed them applying Plain Huffman and *(s,c)*-Dense Code.

A dual Intel®pentium®-III 800 Mhz system, with 768 SDRAM-100Mhz was used in our tests. It ran Debian GNU/Linux (kernel version 2.2.19). The compiler used was gcc version 2.95.2 20000220 and *-O9* compiler optimizations were used. Time results measure encoding "user time".

CORPUS	**#words**	**n**	**(s.c)-DC** (msec)	**Plain** (msec)	DIFF (%)
CALGARY	528,611	31,000	6.150	11.133	44.76
FT91	3,135,383	75,687	15.350	26.500	42.08
CR	10,230,907	117,713	25.750	49.833	48.33
ZIFF	40,866,492	237,607	56.650	105.900	46.51
AP	53,349,620	268,896	64.700	121.900	46.92
FT92	36,803,204	284,878	69.000	129.817	46.85
FT93	42,063,804	291,404	69.725	133.350	47.71
FT94	43,335,126	294,490	71.600	134.367	46.71
ALL_FT	124,971,944	577,290	142.875	260.800	45.22
ALL	229,596,845	885,873	216.225	402.875	46.33

Table 6.4: Code generation time comparison

Table 6.4 shows the results obtained. The first column indicates the corpus processed, the second the number of words in the corpus, and the third the number of distinct words in the vocabulary. The fourth and fifth columns give the encoding time (in milliseconds) for (s, c)-DC and Plain Huffman. The last column shows the gain (in percentage) of (s, c)-DC over Plain Huffman.

(s, c)-DC code generation process is always about 45% faster than Plain Huffman. Although the encoding is in both methods $O(n)$, (s, c)-DC performs simpler operations. Computing the list of accumulated frequencies and searching for the best (s, c) pair only involve elemental operations, while the process of building a canonical Huffman tree has to deal with the tree structure.

6.6.3 Decompression Time

The decompression process is almost identical for Plain Huffman and (s, c)-DC. The process starts by loading the words of the vocabulary into a vector V. For decoding a codeword, (s, c)-DC also needs the s value used in compression, while Plain Huffman needs to load two vectors: *base* and $first$. Next, the compressed text is read and each codeword is replaced by its corresponding uncompressed word. Since it is possible to detect the end of a codeword by using either the s value (in (s, c)-DC) or the $first$ vector (in Plain Huffman), decompression is performed codeword-wise. Given a codeword C, a simple decoding algorithm obtains the position i of the word in the vocabulary, such that $V[i]$ is the uncompressed word that corresponds to codeword C. Decompression takes $O(n)$ time, being n the size of the compressed text.

Each corpus described in Section 6.6.2 was decompressed using both Plain Huffman and (s, c)-DC. Results are shown in Table 6.5. The size of the compressed text is shown in columns two and three. Next two columns present decompression "user-time" (in seconds). The sixth and the seventh columns show decompression speed (in Kbytes per second) and the latter shows the gain (in percentage) of decompression speed of (s, c)-DC over Plain Huffman. Results give (s, c)-DC a small advantage in decompression

speed in most corpora, although differences are usually negligible.

	Compressed Text Size		Decompression time (sec)		Decompression speed (Kbytes/sec)		
CORPUS	**(s,c)-DC**	**PH**	**(s,c)-DC**	**PH**	**(s,c)-DC**	**PH**	**DIFF%**
CALGARY	748,657	740,698	0.165	0.158	4,530.45	4,687.96	-3.477
FT91	4,489,787	4,453,239	1.073	1.006	4,186.28	4,426.68	-5.743
CR	14,049,996	13,937,879	3.429	3.346	4,097.40	4,165.53	-1.663
ZIFF	59,143,837	58,793,833	13.064	12.994	4,527.14	4,524.59	0.056
AP	78,951,674	78,266,031	18.091	17.985	4,364.16	4,351.74	0.285
FT92	53,884,292	53,501,281	12.541	12.509	4,296.55	4,277.17	0.451
FT93	60,846,348	60,460,320	13.752	13.676	4,424.65	4,420.91	0.085
FT94	62,704,466	62,305,648	14.315	14.392	4,380.33	4,329.19	1.168
ALL_FT	182,657,196	181,810,383	42.070	41.882	4,341.74	4,341.01	0.017
ALL	348,361,693	346,337,592	77.627	77.756	4,487.63	4,454.16	0.746

Table 6.5: Decompression speed comparison

6.7 Searching (s, c)-Dense Code

As shown, (s, c)-Dense Code is a prefix code. For this reason, the concatenation of bytes from two consecutive codewords cannot produce a false matching. Therefore, it possible to use a Boyer-Moore type searching which permits to skip exploring some bytes of a codeword. however, it has to be taken into account that both the End-Tagged Dense Code and the (s, c)-Dense Code, are not *suffix free codes* (a codeword can be the suffix of another codeword). Hence, each time that a matching occurs it is mandatory to check if the previous byte to the byte where the text starts to match the pattern, is or not a *stopper*. If the previous byte is a *stopper*, then a right matching has to be reported. However, if it is a *continuer*, then the match is not with the searched codeword, but with the suffix of a larger codeword, and the process continues. An example of how false matchings can be detected is presented in Figure 6.5.

This overload in searches is negligible because checking the previous byte is only needed when a matching is detected, and it is not necessary during the search phase. Moreover, this small disadvantage with respect to Tagged Huffman (which is a suffix code) is compensated because the size of the compressed text is smaller in (s, c)-Dense Code and End-Tagged Dense Code than in Tagged Huffman.

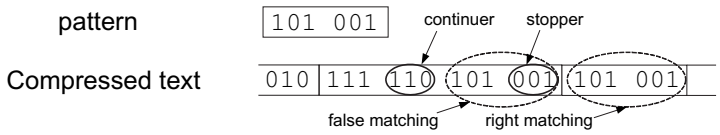


Figure 6.5: Searching $(5, 3)$-Dense Code

6.8 Bounding Plain Huffman with (s, c)-Dense Code

Gonzalo, cuando revise lo que hicimos con la Ziff-MandelBrot y le des el visto bueno (eso espero), lo incluiremos aquí.

6.9 Conclusions

(s, c)-Dense Code, a simple method for compressing natural language text databases with several advantages over the existing techniques was presented. This technique is a generalization of the previous End-Tagged Dense Code, and improves its compression ratio by adjusting the parameters s and c to the distribution of frequencies of the text to be compressed. Some

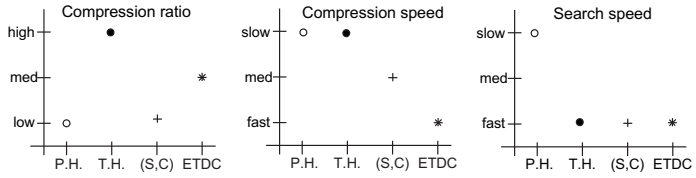


Figure 6.6: Compression ratio, compression speed and search speed comparison

empirical results comparing (s, c)-DC against Huffman codes are shown. In compression ratio, our new code is strictly better than Tagged Huffman Code (by 3.5% in practice), reaching only 0.5% excess over the optimal Plain Huffman Code. The new code is simpler to build than Huffman Codes and can be built in around half the time. This makes (s, c)-DC a real

alternative to Plain Huffman in situations where a simple, fast and good compression method is required. Moreover, (s, c)-DC enables, as Tagged Huffman, fast direct search on the compressed text, which improves Plain Huffman searching efficiency. Only Tagged Huffman Code can be searched so efficiently, but it produces about 11% larger compressed texts.

Figure 6.6 summarizes compression ratio, compression speed and search speed of the different methods.

Part II

Adaptive Compression

7

Dynamic Text Compression Techniques

7.1 Overview

In this Chapter a brief description of the State of Art of *dynamic* text compression techniques is presented. First the motivation of adaptive techniques versus static and semi-static ones is shown. In Section 7.3 the operation of statistical dynamic codes is explained. Dynamic character-based Huffman techniques, which are the basis of the methods presented in Chapters 8, 9 and 10 are described in Subsection 7.3.1. Arithmetic Coding, another typical statistical dynamic technique, is shown in Subsection 7.3.2. Section 7.4 is dedicated to *PPM*, an interesting statistical compression scheme. Finally, Section 7.5 describes dictionary techniques and the variations of Ziv-Lempel, one of the most widespread compression families.

7.2 Introduction

Transmission of compressed data is usually composed of four processes: *compression*, *transmission*, *reception*, and *decompression*. The first two are carried out by a *sender* process and the last two by a *receiver*. This structure

can be shown for example when a user download a zip file from a website. The file, which was previously compressed in the server, is transmitted in compressed form when the user downloads it. Finally, once the transmission has finished, it can be decompressed.

There are several interesting *real-time* transmission scenarios, however, where *compression*, *transmission*, *reception*, and *decompression* processes should take place concurrently. That is, the sender should be able to start the transmission of compressed data without preprocessing the whole text, and simultaneously the receiver should start reception and decompression of the text as it arrives.

Real-time transmission is usually of interest when communicating over a network. This kind of compression can be applied, for example, in the following scenarios:

- Interactive services such as remote talk/chat protocols, where small messages are exchanged during the whole communication process.
- Transmission of Web pages. Installing a browser plug-in to handle decompression, enables the exchange of compressed (relatively small) pages between a server and a client along the time in a more efficient way.
- Wireless communication with hand-held devices with little bandwidth and processing power (therefore the decompressor must be simple).

Real-time transmission is handled by *dynamic* compression techniques. Currently, the most widely used adaptive compression techniques belong to the Ziv-Lempel family [60, 61, 54]. When applied to natural language text, however, the compression ratios achieved by Ziv-Lempel are not that good (around 40%). Their advantages are compression speed and mainly decompression speed.

Other adaptive techniques as the Arithmetic Encoding [1, 57, 38] or the Prediction by Partial Matching (PPM) technique [8] has been proven as competitive adaptive techniques regarding to compression ratio. However they are not time-efficient.

Classic Huffman code [27] is a well-known two-pass method. Making it dynamic was first proposed in [19, 22]. This method was later improved in [29, 52]. However being character-based the compression ratios achieved where not good. It is interesting to point out that the operation of the adaptive Huffman based techniques can be extrapolated to a word-based approach. They are the basis of the dynamic techniques that are presented in Chapters 8, 9 and 10.

This Chapter shows the most common adaptive compression alternatives which are used in Chapter 10 to empirically compare them against the dynamic compression techniques developed in this work.

7.3 Statistical Dynamic Codes

Statistical dynamic compression techniques are one-pass. Symbol frequency is collected as the text is read, and consequently, the mapping between symbols and codewords is updated as compression progresses. The receiver acts in the same way that the sender. It computes symbol frequencies and updates the correspondence between codewords and symbols each time a codeword is received.

In particular, dynamic statistical compressors model the text using only the information about source symbol frequencies, that is, $f(s_i)$ is the number of times that the source symbol s_i appears in the text (read up to now).

In order to maintain the vocabulary up-to-date, dynamic techniques need a data structure to keep all symbols s_i and their frequencies $f(s_i)$ up to now. This data structure is used by the encoding/decoding scheme, and is continuously updated during compression/decompression. For each new source symbol, if it is already in the vocabulary, its frequency is increased by 1. If it is not, it is inserted in the vocabulary and its frequency is set to 1. After each change, the vocabulary is reordered and therefore the codeword assigned to any source symbol may be different than before.

To permit the sender to inform the receiver of new source symbols that appear in the text, a special source symbol *new-Symbol* (whose frequency

is zero by definition to keep it always in the last position) is always held in the vocabulary. The sender transmits *new-Symbol* each time a new symbol arises in the source text. Then, the sender encodes the source symbol in plain form (e.g., using ASCII code for words) so that the receiver can insert it in its vocabulary.

Figure 7.1 depicts the sender and receiver processes, highlighting the symmetry of the scheme. *CodeBook* stands for the mapping between symbols and codes, which is used to assign codes to source symbols or vice versa. Note that *new-Symbol* is always the least frequent symbol of the *CodeBook*, and n is the number of symbols in the source text.

7.3.1 Dynamic Huffman Codes

In [19, 22] an adaptive character-oriented Huffman coding algorithm was presented. It was later improved in [29], being named *FGK* algorithm. *FGK* is the basis of the UNIX *compact* command.

FGK maintains a Huffman tree for the source text already read. The tree is adapted each time a symbol is read to keep it optimal. It is maintained both by the sender, to determine the code corresponding to a given source symbol, and by the receiver, to do the opposite.

Thus, the Huffman tree acts as the *CodeBook* of Figure 7.1. Consequently, it is initialized with a unique special node called *zeroNode* (corresponding to *new-Symbol*), and it is updated every time a new source symbol is inserted in the vocabulary or when a frequency is increased. The codeword for a source symbol corresponds to the path from the tree root to the leaf corresponding to that symbol. Any leaf insertion or frequency change may require reorganizing the tree to restore its optimality.

The main challenge of Dynamic Huffman is how to reorganize the Huffman tree efficiently upon leaf insertions and frequency increments. This is a complex and potentially time-consuming process that must be carried out both by the sender and the receiver.

The basis of the *FGK* algorithm is the *sibling property* defined by Gallager in [22].

```
Sender ( )
(1)  Vocabulary ← {C_new-Symbol};
(2)  Initialize CodeBook;
(3)  for i ∈ 1...n do
(4)       read s_i from the text;
(5)       if s_i ∉ Vocabulary then
(6)            send C_new-Symbol;
(7)            send s_i in plain form;
(8)            Vocabulary ← Vocabulary ∪ {s_i};
(9)            f(s_i) ← 1;
(10)      else
(11)           send CodeBook(s_i);
(12)           f(s_i) ← f(s_i) + 1;
(13)      Update CodeBook;
```

```
Receiver ( )
(1)  Vocabulary ← {C_new-Symbol};
(2)  Initialize CodeBook;
(3)  for i ∈ 1...n do
(4)       receive C_i;
(5)       if C_i = C_new-Symbol then
(6)            receive s_i in plain form;
(7)            Vocabulary ← Vocabulary ∪ {s_i};
(8)            f(s_i) ← 1;
(9)       else
(10)           s_i ← CodeBook^-1(C_i);
(11)           f(s_i) ← f(s_i) + 1;
(12)      output s_i;
(13)      Update CodeBook;
```

Figure 7.1: Sender and receiver processes in statistical dynamic text compression.

Property 7.3.1 *A binary code tree has the sibling property if each node (except the root) has a sibling and if all nodes can be listed in decreasing weight order, with each node adjacent to its sibling.*

Gallager proved also that a binary prefix code is a Huffman code iff the code tree has the sibling property.

Using the *sibling property*, the main achievement of *FGK* is to ensure that the tree can be updated by doing only a constant amount of work per node in the path from the affected leaf to the tree root. Calling $l(s_i)$ the path length from the leaf of source symbol s_i to the root, and $f(s_i)$ its frequency, then the overall cost of algorithm *FGK* is $\sum f(s_i)l(s_i)$, which is exactly the length of the compressed text, measured in number of target symbols.

In [52] an improvement upon *FGK* denominated Λ *algorithm* was presented. An implementation of Λ can be found in [53]. The main difference with respect to *FGK* is that Λ algorithm uses a different method to update the tree, which not only minimizes $\sum f(s_i)l(s_i)$ (compressed text length) but also the external path length, $\sum l(s_i)$, and the height of the tree, $\max l(s_i)$. Moreover, Λ *algorithm* reduces to 1 the number of interchanges in which a node is moved upwards in the tree during an update of the tree. Although these changes do not modify the complexity of the whole algorithm, they give Λ algorithm advantage in compression ratio over *FGK* and even over static Huffman for small messages. Results shown in [52] show that adaptive Huffman methods are directly comparable to classic Huffman in compression ratio.

7.3.2 Arithmetic Codes

Arithmetic coding was first presented in the sixties in [1]. Using the same idea of Huffman it uses the probabilities of the input symbols in order to achieve compression.

Distinct models can be used to calculate, for a given context the probability of the next input symbol, so that static, semi-static and also adaptive arithmetic codes are available. The key-idea of this technique is to represent an input sequence of symbols using a **unique** real number in

the range [0,1). As the message to be encoded becomes larger, the interval needed to represent it becomes narrower, and therefore, the number of bits needed to represent it grows.

Basically an arithmetic encoder starts with a list of the n symbols of the vocabulary and their probabilities. The initial interval is [0,1). When a new input symbol is processed, the interval is reduced in accordance with the symbol probability provided by the model used, and the interval becomes a narrower range which represents the input sequence of symbols already processed.

We present in Example 7.3.1, a semi-static arithmetic compressor to explain how arithmetic compression works. Note that making it dynamic consists only of adapting the frequency of the source symbols each time one of them is input.

Example 7.3.1 Let compress the message "AABC! " using an semi-static model.
In the first phase the compressor creates the vocabulary that is composed of four symbols: 'A', 'B', 'C' and '!'. Their frequencies are: 0,4, 0,2, 0,2 and 0,2 respectively. Therefore, in the initial state, any number in the interval [0, 0,4) represents symbol 'A', and intervals [0,4, 0,6), [0,6, 0,8) and [0,8, 1) represent symbols 'B', 'C', and '!' respectively.
Since the first symbol to encode is 'A', the interval [0,1) is reduced to [0, 0,4). Next possible sub-intervals are [0, 0,16), [0,16, 0,24), [0,24, 0,32) and [0,32, 0,4). They would represent the sequences 'AA', 'AB', 'AC' or 'A!'. Figure 7.2 represents graphically the intervals of the whole process. Since next symbol is again 'A', the current working-interval is reduced to [0, 0,16). Note that the size of this interval depends on the probability of the sequence encoded; that is: $0,4 \times 0,4 = 0,16$.
To encode next input symbol 'B' the new interval is reduced to [0,064, 0,096), such as the sequence 'AAB' has probability of $0,032 = 0,096 - 0,064$.
After processing 'C' the range becomes [0,0832, 0,0896), and the probability associated to 'AABC' is $0,0064$.
Finally the possible sub-intervals are [0,0832, 0,08576), [0,08576, 0,08704), [0,08704, 0,08832) and [0,08832, 0,096). Since '!' was the last symbol of the vocabulary, each number in the interval [0,08832, 0,096) represents the

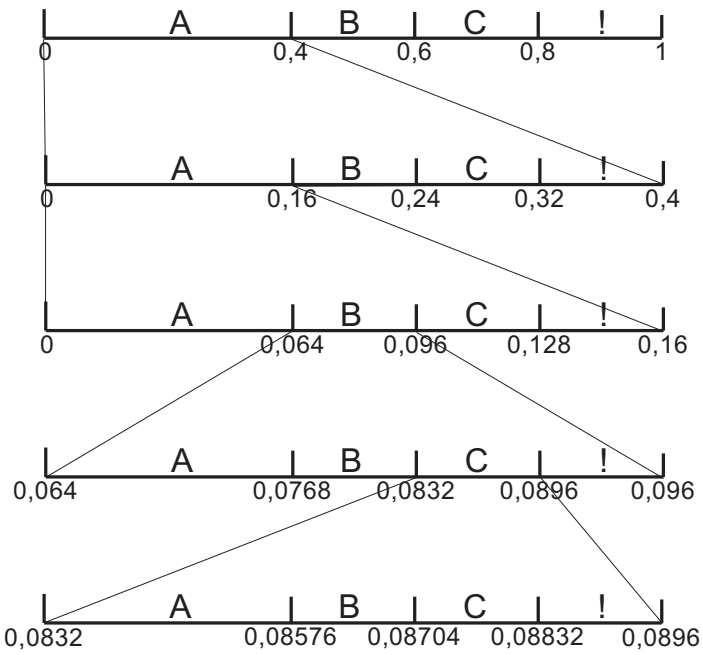


Figure 7.2: Operation of an arithmetic compressor

message 'AABC!'. So the encoder generates the number in that interval that can be encoded with less bits. □

The decompressor only has to know the the vocabulary used, the probabilities of the source symbols and the number of symbols transmitted. From the compressed data it can detect the intervals used in the encoding phase and from these intervals, it recovers the source symbols.

In general, arithmetic compression overtakes Huffman compression ratios. When static or semi-static models are used, compression and decompression speed are not competitive with respect to Huffman based techniques, and moreover they have the disadvantage that decompression and searches cannot be performed directly in the compressed text because it is represented by a simple number, so decompression is mandatory before a search can be performed. As result, arithmetic coding is not useful in

TR environments. However, it becomes a good alternative to adaptive Huffman codes. In [57] an adaptive character based arithmetic technique is compared against the *compact* algorithm (which is based on FGK dynamic Huffman). In this case arithmetic encoding is faster in both compression and decompression processes, and also it achieves best compression ratio.

Several modifications of the basic arithmetic algorithm improving its performance and/or using distinct models were made. In [57], Witten, Neal and Cleary explained an arithmetic coder based on the use of integer arithmetic. In [38] Moffat et all made improvements focused in avoiding using multiplications and divisions that could be replaced with faster shift/add operations. Moreover the code from [16] (based on [38]) is of public domain and has been used in our tests.

7.4 Prediction by Partial Matching

Prediction by Partial Matching technique (PPM), was first presented in 1984 by Cleary and Witten [18].

PPM is an adaptive statistical data compression technique based on context modelling and prediction. Basically, PPM uses sequences of previous symbols in the uncompressed symbol stream to predict the frequency of the next symbol in the input stream, and then it applies an arithmetic technique [57, 36] to encode that symbol using the predicted frequency.

PPM is based on using the last k characters from the input stream to predict the probability of the following one. This is the reason for it to be a *finite-context model of order k*. That is, a finite-context model of order 2 will use only two previous symbols to predict the frequency of the next incoming symbol.

The maximum context length (that is, the length of a sequence of symbols that are considered through the highest-order model) is usually 5. It was shown [18, 33] that increasing the context length beyond $5 - 6$ does not usually improve compression ratio.

PPM combines several finite-context models of order k, such as k takes

the values from 0 to m (m is the maximum context length). For each model, PPM takes account of all $k-length$ sequences S_i that have appeared previously. Moreover, for each of those sequences S_i, all characters that have followed previously S_i, as well as the number of times they have appeared, are kept. The number of times that a character followed a k-length sequence is used to predict the probability of the incoming character in the model of order k. Therefore, for each model, a separate predicted probability distribution is achieved.

The $m + 1$ probability distributions are blended into a single one, and arithmetic coding is used to encode the current character using that distribution. In general, the probability predicted by the highest-order model (the m-order model) is used. However, if a novel character is found in this context (no m-length sequence precedes the new character), then it is not possible to encode the new character using the given m-order model, and it is needed to try the $(m-1)$-order model. In this case, a *escape symbol* is transmitted to warn the decoder that a change from a m to $(m-1)$-order model occurs. The process continues until reaching a model where the incoming symbol is not novel (so that symbol can be encoded with the frequency predicted by the model). To ensure that the process always finishes, a (-1)-order model is assumed to exist. This bottom-level model predicts all characters s_i from the source alphabet (Σ) with the same probability, that is $p(s_i) = \frac{1}{|\Sigma|}$.

Note that, each time a model shift (from k to $k-1$) happens due to a novel symbol , the probability given to the *escape symbol* in the k-order model needs to be combined with the probability that the $(k-1)$-order model assigns to the novel symbol.

We call ω the symbol whose probability is being predicted, $p_k(\omega)$ the probability that a k-order model assigns to ω and e_k the probability assigned to the *escape symbol* by a k-order model. Two situations can happen:

1. ω can be predicted by the m-order model. In this case ω is encoded using the probability $p(\omega) = p_m(\omega)$.

2. ω can be predicted by a k-order model ($k < m$). Then the probability used to encode ω is $p(\omega) = p_k(\omega) \times \prod_{l=k+1}^{m} e_l$.

Distinct methods can be used to assign probabilities to both the *escape symbol* and also to a novel input symbol. For example, PPMA [18] uses a method called *method A*, which makes $p_k(\omega) = \frac{c_k(\omega)}{1+c_k}$ and $e_k = \frac{1}{1+c_k}$, where $c_k(\omega)$ is the number of times that the character ω appeared in the k-order model, and c_k is the total count of characters that were first predicted by a k-order model. PPMC [33], the first popular PPM variant (because PPM algorithms require a significant amount of memory and previous computers were not powerful enough), uses *method C* to assign probabilities. It sets $e_k = \frac{d_k}{c_k}$, and $p_k(\omega) = \frac{(c_k - d_k) \times c_k(\omega)}{c_k}$, where d_k is the number of distinct characters that were first predicted by a k-order model.

Choosing a method to assign probabilities to the *escape symbol* constitutes an interesting problem called *the zero-order frequency problem*. Distinct methods have been proposed: *Methods A and B* [18], *method C* [33], *method D* [26] and *method X* [55]. A description and a comparison of the those available methods can be found in [37].

Recent PPM implementations obtains good compression results. As shown in [17], PPMC uses 2.48 bits per character (bpc) to encode Calgary corpus and another variant PPM* (or PPMX)[17] needs only 2.34 bpc (less than 30% in compression ratio). However, compression and decompression speed are not so good (it is 5 times slower than *gzip* in compression).

Nowadays, the main competitor of PPM in compression ratio is *bzip2*[1]. In [56], it is shown that PPM and *bzip2* compression ratio and compression speed are quite similar (with a small advantage to PPM). However *bzip2* is 3 times faster than PPM in decompression. These reasons (similar compression, but worse decompression) and the fact that *bzip2* has become a well-known and widespread compression technique, led us to use *bzip2* instead of PPM in our tests.

[1] *bzip2* is a is a freely available, patent free data compressor. It uses the Burrows-Wheeler Transform along with a Huffman code

7.5 Dictionary techniques

Dictionary techniques do not take account of statistics about the number of occurrences of symbols in a text. They are based on building a dictionary where substrings are stored, in such a way that each substring is assigned a codeword (usually the codeword consist of its position in the dictionary). Using that dictionary, each time a substring is read from the source stream, it is searched in the dictionary, and substituted by its codeword. Compression is achieved because of the length of the substituted substrings is usually bigger than the number of bits needed to represent their codeword.

Dictionary techniques are the most commonly used compression techniques. In fact, Ziv-Lempel family [60] is the most representative technique, and its two main variations *LZ77* [60] and *LZ78* [61] are the basis of the *gzip* and *compress* programs respectively.

7.5.1 LZ77

This technique, presented in 1977 [60], is the first technique that Abraham Lempel and Jacob Ziv presented. LZ77 uses the dictionary strategy. It has a dictionary that is composed of the n last characters already processed (n is a fixed value) and is commonly called *sliding window*.

Compression starts with an empty *sliding window*. In each step of the compression process, the largest substring w that already appears in the *window* is read. That is, characters $w_0, w_1, ..., w_k$ after the *window* are read while the substring $w = w_0, w_1, ..., w_k$ can be found in the *window*. Suppose that U is the next character after w. Then the compressor outputs the triplet $< position,\ length,\ U >$, and the *window* is slid $k + 1$ positions. If $w = \emptyset$ then the triplet $< 0,\ 0,\ U >$ is output. The *position* element of the triplet represents the backwards offset with respect to the **end** of the *window* where w appears, and the *length* element corresponds to the size of the w substring (that is, k).

Example 7.5.1 Let compress the text "`abbabcabbbbc`" using *LZ77* technique and supposing a *sliding window* of only 5 bytes. The process,

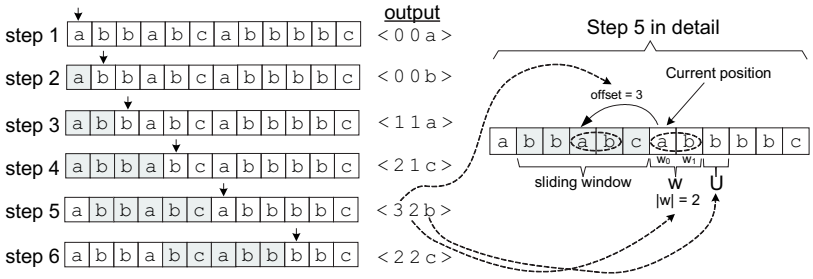


Figure 7.3: Compression using LZ77

which is summarized in Figure 7.3, consists of the following six steps:

1. The *window* starts with no characters, so `'a'` is not in the dictionary and $< 0, 0, a >$ is output. Moreover, the window is slid 1 character to the right, therefore *window* will contain ⟨-,-,-,-,'a'⟩.

2. Next `'b'` is read, it is not in the dictionary and $< 0, 0, b >$ is output. Finally the window is slid 1 position. Hence the new *window* contains ⟨-,-,-,'a','b'⟩.

3. A new `'b'` is read, and it was in the last position of the *window*, but the prefix 'ba' is not in the vocabulary yet. Therefore $w =$ `'b'`, $U =$ `'a'` and $< 1, 1, a >$ is output. Finally the window is slid $|w| + 1 = 2$ positions, so *window* contains ⟨-,'a','b','b','a'⟩.

4. The next substring is `'b'`, but `'bc'` does not appear in the *sliding window*, so $w =$ `'b'`, $U =$ `'c'` and $< 2, 1, c >$ is output. After sliding the *window* $|w| + 1 = 2$ positions, it contains ⟨'b','b','a','b','c'⟩.

5. The process continue reading `'abb'`, such as `'ab'` is a prefix in the *window*, and $< 3, 2, b >$ is output. The new *window* contains ⟨'b','c','a','b','b'⟩.

6. Finally, since `'bb'` is a new recognized substring but `'bbc'` does not appear in the *sliding window*, $< 2, 2, c >$ is output. □

Decompression is similar to compression, since the *window* holds the last decoded elements. Given a triplet $< position, length, nextChar >$, the decompressor outputs *length* characters starting in *position* elements before the end of the *window* and finally it also outputs *nextchar*. Note that the vocabulary is quite small (it can be kept in cache in an actual processor), and decoding process implies only one array lookup, therefore decompression is very fast.

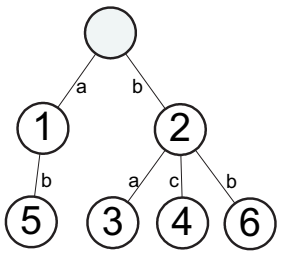
In general, a minimum substring size (usually 3 characters) is used to avoid the substitution of small prefixes. Moreover, the length of the *sliding window* has a fixed value. A higher value provokes that larger substrings can be found in the *sliding window*, however that implies that a larger pointer will also be needed to represent the *position* element of the triplet. In general, the *position* element is represented by 12 bits (hence the *sliding window* has 4096 bytes at most), and 4 bits are used for the *length* element. That is, 2 bytes are needed to represent both *position* and *length*.

7.5.2 LZ78

Instead of a window that contains the last processed text, *LZ78* technique [61] builds a dictionary that holds all phrases recognized in the text. This dictionary is efficiently searched via a *trie* data structure in such a way that a leaf l_i in the *trie* stores a pointer i to a dictionary entry (entry_i), and the path from the root of the *trie* to the leaf node l_i refers to the letters of entry_i.

Encoding and decoding in *LZ78* are simple. Basically it consists of: *i)* From the current position in the text, the longest entry in the vocabulary (entry_i) that matches with the following characters in the text is found. *ii)* The pair (i, U) is output (where U is the character that follows entry_i in the text). *iii)* The new phrase (entry_i+U) is appended to the dictionary. Figure 7.4 explains how to compress the text "`abbabcabbbbc`" using *LZ78* technique.

Compression in *LZ78* is faster than in *LZ77*, however decompression is slower because the decompressor needs to store the phrases. However this technique results interesting, and a variation of *LZ78* called *LZW* is widely used, as shown next.



Trie structure

step	input	output	Dictionary
1	a	(0,a)	entry_1 = "a"
2	b	(0,b)	entry_2 = "b"
3	ba	(2,a)	entry_3 = "ba"
4	bc	(2,c)	entry_4 = "bc"
5	ab	(1,b)	entry_5 = "ab"
6	bb	(2,b)	entry_6 = "bb"
7	bc	(4,∅)	–

Figure 7.4: Compression of the text "abbabcabbbbc" using LZ78

LZW

It is a variation of *LZ78* which was proposed by Welch in 1984 [54]. *LZW* is widely used, the Unix *compress* program and the *GIF* image format use this technique to achieve compression. The main difference of *LZW* with respect to *LZ78* is that it only produces a list of pointers in the output, while *LZ78* outputs also characters explicitly. This can be avoided in *LZW* by initializing the vocabulary with phrases that include all the characters from the source alphabet (for example, the 128 ASCII values). Another difference consists of that fast searching for a word in the vocabulary is made through a hash table instead of a trie data structure.

The encoding process starts with the initial vocabulary. From the input, the largest phrase that exists in the vocabulary is searched for (note that since all characters belong to the vocabulary, phrases are always found in the vocabulary). Suppose that the entry in the vocabulary which corresponds to that phrase is i, then i is output, and the a new phrase, formed as the concatenation of $phrase_i$ and the next character in the input, is added to the vocabulary. Then the process iterates reading next characters from the input and searching again for the largest phrase in the vocabulary. An example describing the whole compression process is shown in Table 7.1.

input	next character	output	Dictionary
initial	initial	initial	$entry_0$ = "a"
initial	initial	initial	$entry_1$ = "b"
initial	initial	initial	$entry_2$ = "c"
a	b	0	$entry_3$ = "ab"
b	b	1	$entry_4$ = "bb"
b	a	1	$entry_5$ = "ba"
ab	c	3	$entry_6$ = "abc"
c	a	2	$entry_7$ = "ca"
ab	b	3	$entry_8$ = "abb"
bb	b	4	$entry_9$ = "bbb"
b	c	1	$entry_{10}$ = "bc"
c	$\emptyset$	2	–

Table 7.1: Compression of "abbabcabbbbc", ($\sum$={a, b, c}), using LZW

Since a maximum dictionary size has to be defined, some choice has to be taken when all the available entries in the vocabulary are completed:

- Do not permit more entries and continue compression with current vocabulary.
- To use a least recently used policy to discard entries when a new entry has to be added.
- Delete the whole dictionary continuing with an empty one (which has only the initial entries, that is,the symbols of the alphabet).
- Continue monitoring the compression ratio, and drop the dictionary when compression decreases.

Two Ziv-Lempel based techniques are widely used. The first one is *gzip* (it is based on LZ77) and the second is *compress* (based on LZW). As it is shown in [56], *gzip* achieves better compression ratio (about 35-40%) and compression speed than *compress* (compression ratio about a 40%). However, *compress* is a bit faster in compression. *Gzip* was used in our

tests, as a representative of dictionary based techniques. We decided to use *gzip* instead of *compress* because of its better compression ratio.

7.6 Summary

In this Chapter, the State of Art of dynamic text compression techniques has been revisited. Special attention was paid to statistical methods such as Huffman-based ones. Arithmetic compression, as well as a symbolwise model as the Prediction by Partial Matching technique, were also shown. Finally a brief review of Ziv-Lempel based compressors, the most commonly used compression strategy, was also presented.

8

Dynamic Byte-oriented word-based Huffman code

8.1 Chapter overview

In this Chapter, a dynamic byte-oriented word-based Huffman code is presented. This is a minor contribution of this thesis. Even though we developed and implemented this technique in order to compare it against the Dynamic End-Tagged Dense Code and Dynamic (s, c)-Dense Code, presented in Chapters 9 and 10 respectively, the results achieved both in compression ratio, and also in compression/decompression speed, make it a competitive adaptive technique, and hence a valuable contribution. Therefore it is presented in this Chapter in detail.

We start with a brief introduction which explains the motivation of developing a word-based byte-oriented Huffman code. Next, some properties and definitions about the code are shown. In Section 8.4 the technique is described. Later, the data structures used and the update algorithm that enables maintaining a well-formed Huffman tree dynamically are shown. Finally, some empirical results compare the our Dynamic word-based Huffman Code against the well-known character-based approach.

8.2 Introduction

However, those methods are character- rather than word-oriented, and thus their compression ratio on natural language is poor (around 60%).

In recent years, however, new Huffman-based compression techniques for natural language have appeared, based on the idea of taking the words, not the characters, as the source symbols to be compressed [34]. Since in natural language texts the frequency distribution of words is much more biased than that of characters, the gain in compression is enormous, achieving compression ratios around 25%-30%. Additionally, since in Information Retrieval (IR) words are the atoms searched for, these compression schemes are well suited to IR tasks. Word-based Huffman variants focused on fast retrieval are presented in [49], where a byte- rather than bit-oriented coding alphabet speeds up decompression and search.

Two-pass codes, unfortunately, are not suitable for real-time transmission. Hence, developing an adaptive compression technique with good compression ratio for natural language texts is a relevant problem.

In this Chapter a dynamic word-based byte-oriented Huffman method is presented. It yields the advantages of the semi-static word-based techniques (compression ratio about 30%-35% in large texts) and the real-time facilities of the dynamic character-based bit-oriented classic Huffman techniques. Moreover, being byte- rather than bit-oriented, compression and decompression speed are clearly competitive.

8.3 Word-based Dynamic Huffman Codes

We implemented a word-based byte-oriented version of algorithm *FGK*. This is by itself a contribution because no existing adaptive technique obtains similar compression ratio on natural language.

As the number of text words is much larger than the number of characters, several challenges arised to manage such a large vocabulary. The original *FGK* algorithm pays little attention to these issues because of its

underlying assumption that the source alphabet is not very large.

For example, the sender must maintain a hash table that permits fast searching for a word s_i, in order to obtain its corresponding tree leaf and its current frequency. In a character-oriented approach, this can be simply an array indexed by character.

However, the most important difference between our word-based version and the original *FGK* is that we chose the code to be byte- rather than bit-oriented. Although this necessarily implies some loss in compression ratio, it gives a decisive advantage in efficiency. Recall that the algorithm complexity corresponds to the number of target symbols in the compressed text. A bit-oriented approach requires time proportional to the number of bits in the compressed text, while ours requires time proportional to the number of bytes. Hence byte-coding is almost 8 times faster.

Being byte-oriented implies that each internal node can have up to 256 children in the resulting Huffman tree, instead of 2 as in a binary tree. This required extending *FGK* algorithm in several aspects. In particular, some parent/child information that is made implicit by an appropriate node numbering, must be made explicit when the tree arity exceeds 2. So each node must store pointers to its first and last child. Also, the process of restructuring the tree each time an input symbol is processed, is more complex in general.

The *sibling property* (see Property 7.3.1) can be modified for its use in a byte-oriented Huffman code as follows:

Property 8.3.1 *A 2^b-ary code tree has the sibling property if each node (except the root and the zeroNode and its siblings), have $2^b - 1$ siblings, as well as if all nodes can be listed in decreasingly frequency order, with each node adjacent to its siblings.*

Note that the *zeroNode* is a special zero-frequency node that is used in dynamic codes to introduce those symbols that have not appeared yet.

Intuitively, it can be seen that in a 2^b-ary well-formed Huffman tree all internal nodes have 2^b children. However the parent of the *zeroNode* can

have less than 2^b children. In fact, if n is the number of symbols (leaves) in a 2^b-ary Huffman tree (including the *zeroNode*), the parent of the *zeroNode* has exactly R child-nodes, such as:

$$R = \begin{cases} 1 + ((n - 2^b) \ mod \ (2^b - 1)) & \text{if } ((n - 2^b) \ mod \ (2^b - 1) > 0 \\ n & \text{if } ((n - 2^b) \ mod \ (2^b - 1) = 0 \end{cases}$$

Note that the *zeroNode* and its $R-1$ siblings are the least frequent leaf nodes of the tree.

Definition 8.3.1 *To achieve a dynamic word-based, byte-oriented Huffman code it is only needed to maintain two conditions:*

- *All nodes of the Huffman tree (both internal and leaf nodes) remain sorted by frequency. In this ranking the root is the most frequent node, and the zeroNode is the least weighted node.*
- *All the siblings of a node remain adjacent.*

□

To maintain the conditions needed by Definition 8.3.1, a node-numbering method was considered. It ranks nodes in following way:

- Nodes in the top levels of the tree precede nodes in the bottom levels.
- For a given level of the tree, nodes are ranked left-to-right. Therefore the left-most node of a level precede all nodes in that level.

8.4 Method Overview

The compressor/sender and decompressor/receiver algorithms use the general guidelines shown in Figure 7.1. Considering that general scheme, the main issue that has to be taken into account is how to maintain the *Codebook* up to date, after insertions of new words, or when the frequency of a word is increased.

In this case, the *Codebook* is a 256-ary Huffman tree that stores all words procesed up to a given moment of the compression/decompression process.

This Huffman tree is used by both the encoder and the decoder to handle the *encoding* of a word and also the *decoding* of a codeword. In the encoding process, a codeword is given by the path from the root of the tree to the leaf node where that word is placed. The process of decoding a codeword, starts in the root of the tree, and each byte value in the codeword specifies the child node in the next step in a top-down traversal of the tree. A word is recognized each time a leaf node is reached.

Once a encoding/decoding of a word s_i has been performed, the tree has to be updated. If s_i was already in the tree, the frequency of the leaf node q representing s_i has to be increased, having into account that the order by frequency of words must be maintained. To achieve this, the first node t in the node-numbering such as $frequency_q = frequency_t$ is located. Next step involves an interchange of nodes q and t. After that, q's frequency can be increased (without altering the order of words). Finally the increase of frequency has to be promoted to q's parent. The process continues until the root of the tree is reached.

If s_i was not in the Huffman tree a new node q representing s_i has to be added (with $frequency = 1$). Two situations can take place: *i)* if $zeroNode$ has less than 255 siblings then q is added as a new sibling of the $zeroNode$. *ii)* If the $zeroNode$ has 255 siblings then a new internal node p is created in the position of $zeroNode$ and both q and $zeroNode$ are set as children of p. In both cases the increase of frequency has to be promoted to the ancestors of q.

Figure 8.1 presents the way a 4-ary Huffman tree is maintained via the adaptive process when words "aaaaaaaaabbbbbbbbcddddddccccceeefg" are transmitted.

The first box shows the initial state of the Huffman tree. The only nodes in the tree are the $zeroNode$ and the *root* node. In step 2, a new node 'a' is inserted into the tree as a sibling of the $zeroNode$ (as the parent of the zeroNode has enough space to hold 'a' no extra internal nodes have to be added). In step 7, node 'd' is added. As the zeroNode has already

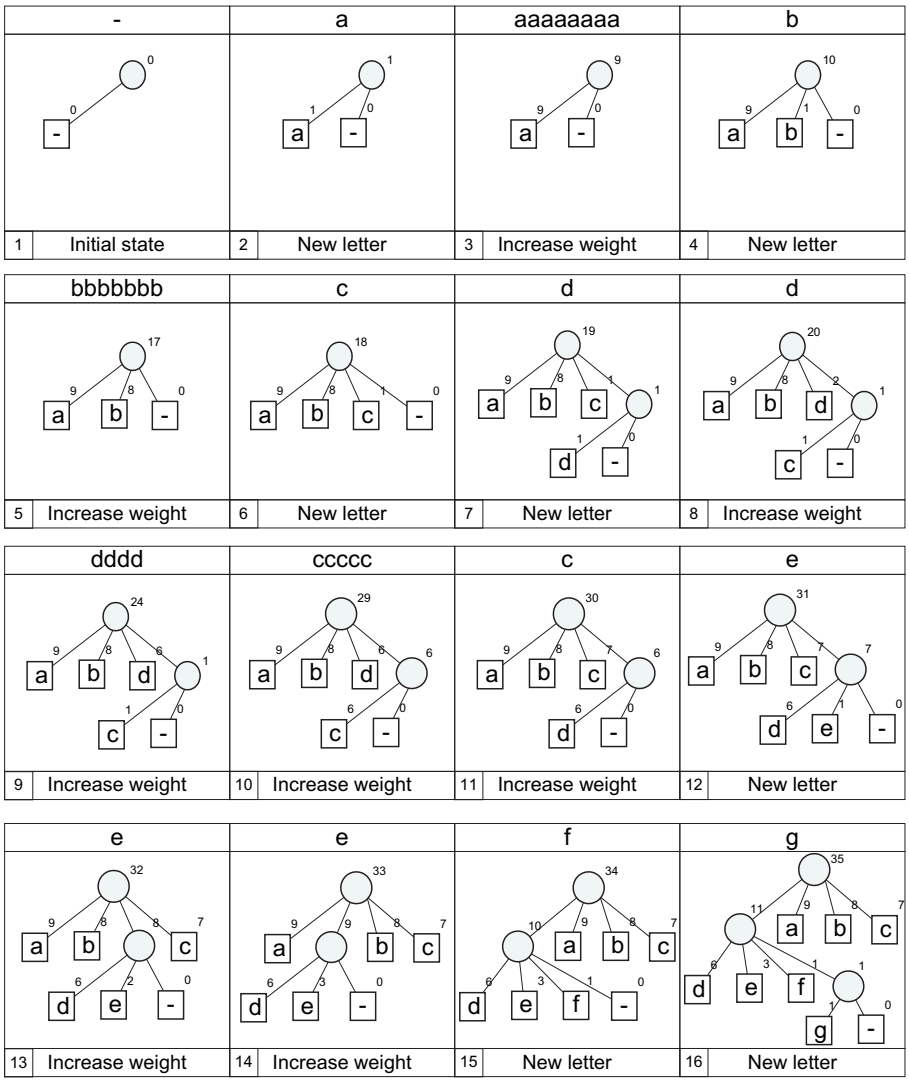


Figure 8.1: Dynamic process to maintain a well-formed 4-ary Huffman tree

$2^b - 1 = 3$ siblings, a new internal node is created. In the eighth box, increasing the frequency of node 'd' implies and interchange with node 'c' (needed to maintain the ordered list of nodes). In step 13, the increase of frequency of node 'e' cause and interchange between the parent of 'e' and node 'c'.

8.5 Data Structures

In this Section, the data structures that support this code are presented. Two main parts arise: *i)* A data structure that supports the Huffman tree, and enables fast lexicographical searches (given a word it is needed to find the node representing it in the tree). And *ii)* A *blocks* data structure that, given a frequency value x, enables finding efficiently the first node q whose frequency is x in the ordered list of nodes.

8.5.1 Definition of the tree data structures

Let suppose that the maximum number of distinct words that will be encoded is n. That is, n is the maximum number of leaf nodes of the Huffman tree and the number of internal nodes is at most $\lceil n/255 \rceil$. Let call $N = n + \lceil n/255 \rceil$, the maximum number of nodes in the tree.

The Huffman tree is hold through three different data structures: *i)* a *node index* that enables the management of both leaves and internal nodes independently of their type, *ii)* an *internal nodes* data structure, and *iii)* a *hash table* that holds the words of the vocabulary (and hence the leaf nodes of the tree). Two more variables $zeroNode$ and $maxInternalNode$ handle respectively the first free position in the *node index* and the first free position in the *internal nodes* data structure. Figure 8.2 presents all the data structures and their relationship.

Node Index

As shown, this structure keeps up to N nodes (both internal and leaf nodes). It consists of four arrays, all of them of N elements:

- nodeType [q] = $'I'$, $1 \leq q \leq N$ iff the q^{th} most frequent node of the Huffman tree is an internal node. If it is a leaf node then nodeType [q] = $'L'$.
- freq[q] = w, $1 \leq q \leq N$. Depending on the value nodeType[q]:
  - If nodeType[q] = $'L'$ then freq[q] = w indicates that the number of occurrences of the q^{th} node in the tree is w.
  - If nodeType[q] = $'I'$ then freq[q] = w represents the summation of the frequencies of all the leaf nodes located in the subtree whose root is the q^{th} node in the tree.
- parent[q] = j, $1 \leq q \leq N$ and $1 \leq j \leq \lceil n/255 \rceil$, iff the j^{th} internal node is the parent of node q. In the case of the root of the tree, that is, the first node in the node index, $parent[1] = 1$.
- relPos[q] = j, $1 \leq q \leq N$. Depending on the value nodeType[q]:
  - If nodeType[q] = 'I' then relPos[q] = j, $1 \leq j \leq \lceil n/255 \rceil$ means that the q^{th} node of the tree corresponds to the internal node that is stored in the j^{th} position of the *internal nodes* data structure.
  - If nodeType[q] = 'L' then relPos[q] = j, $1 \leq j \leq nextPrime(2n)$ means that the q^{th} node of the tree corresponds to the leaf node stored in the hash table in the j^{th} position.

Internal nodes data structure

It stores the data that the algorithms need for internal nodes. As shown, the maximum number of internal nodes is $\lceil n/255 \rceil$. Tree vectors are used:

- minChild [i] = q, $1 \leq i \leq \lceil n/255 \rceil$, $1 \leq q \leq N$, iff the first child of internal node i is held in the q^{th} position of the *node index*.

- maxChild [i] = q, $1 \leq i \leq \lceil n/255 \rceil$, $1 \leq q \leq N$, iff the last child (less frequent) of the internal node i is located in the q^{th} position of the *node index*. Note that it always holds that $maxChild[i] - minChild[i] = 255$ (except if i is the parent of the $zeroNode$. In such case the number of children of i can be smaller than 256).

- inodePos[i] = q, $1 \leq i \leq \lceil n/255 \rceil$, $1 \leq q \leq N$. It means that the i^{th} internal node is the q^{th} node of the *node index*.

Hash table to hold leaf nodes

- word[q] = w_i, $1 \leq q \leq nextPrime(2n)$. That is, a word w_i, is in position q in the hash table. This happens if and only if: *i)* $q = f_{hash}(w_i)$ and $words[q] = Null$ before adding w_i into the tree. *ii)* $q' = f_{hash}(w_i)$, $words[q'] \neq w_i$ and $q = solveCollision(q', w_i)$.

- lnodePos[i] = q, $1 \leq i \leq nextPrime(2n)$, $1 \leq q \leq N$. It means that the word in the position i in the hash table corresponds to the q^{th} node of the *node index*.

Figure 8.2 shows how the tree data structure is used supposing a 4-ary tree. In the right part, the tree that is represented by the data structures previously defined is shown. In the left, those vectors are filled in order to represent the tree of step 12 in Figure 8.1.

Note that the tree data structures just defined are enough to enable maintaining sorted by frequency all nodes in the Huffman tree. In fact, the first position in the node index stores the root of the tree (the most frequent node), the second position holds the most frequent nodes among the remainder ones (except the root), and so on. However with only those data structures it is time-expensive to find the *first* node of a given weight in the *node index*, as a sequential search should be done (remember that finding the *first* node is needed, at least once, when updating the tree). In order to improve performance next structure is also used.

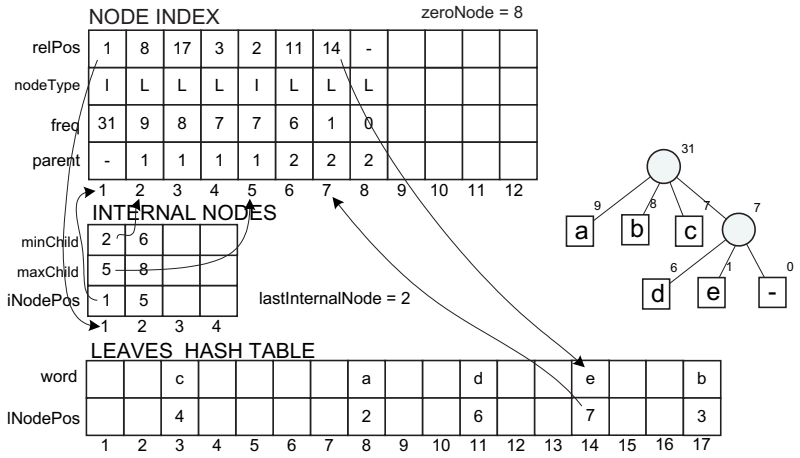


Figure 8.2: Use of the data structure to represent a Huffman tree

8.5.2 List of blocks

Using the idea pointed in [29, 52] *nodes with the same weight* are joined into *blocks* and a list of *blocks* is maintained sorted in decreasing order of weight. We define $block_j$ as the block that groups all nodes whose number of occurrences is j. We also define top_j and $last_j$ nodes respectively as the first and the last nodes in the *node index* that are members of $block_j$.

The main advantage of maintaining a list of *blocks*, is that it can be easily maintained sorted in constant time. Lets suppose that a word w_i, such as w_i already belongs to block b_j, is been input. The number of occurrences of the leaf node that represents to w_i needs to be increased (if the word was not in the tree yet, it is added at the end of the tree by using the $zeroNode$ value). This is done by changing w_i from block b_j to block b_{j+1}, what basically implies two operations:

- Interchange w_i with top_j, such as top_j is the first node in the *node index* whose frequency is j by definition. That is, freq[top_j] = j by definition and j >= freq[x] $\forall$ *node* $x > top_j$.
- Set w_i as $last_{j+1}$.

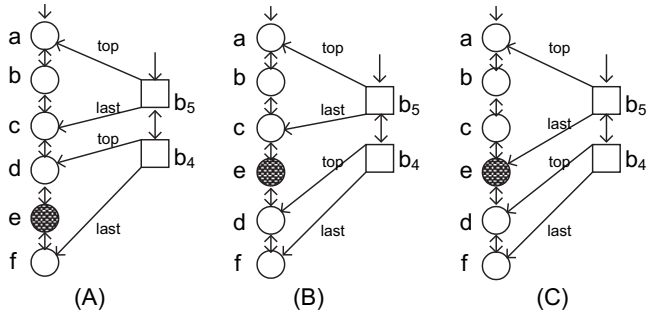


Figure 8.3: Increasing of frequency of word **e**

Figure 8.3 describes the process of changing a given node e from block b_4 to block b_5. In Figure 8.3(B) it is shown how e is interchanged with d, the *top* of its block b_4. Finally, Figure 8.3(C), e is set as the *last* node of b_5.

When a node w_i is processed, several situations can take place depending on the state of the tree and the state of sorted list of blocks:

A. w_i was not in the tree: In this case, w_i is added to the tree, and set as the last node of b_1. See Figure 8.4(A). To add w_i to the tree, both the *node index* and the *hash table* are updated in the following way: *i)* if *zeroNode* has less than 255 siblings then w_i is added as a new sibling of the *zeroNode*. *ii)* If the *zeroNode* has already 255 siblings then a new internal node q is created (in the address of the *zeroNode*) and both w_i and *zeroNode* are set as children of q.

B. w_i is the unique node in b_j and b_{j+1} exits. Since w_i is the only node in b_j, then when w_i pass to block b_{j+1}, hence b_j remains empty and has to be removed from the list of blocks. This scenario is shown in Figure 8.4(B).

C. w_i belongs to b_j and b_{j+1} does not exist. If there are two or more nodes in b_j then block b_{j+1} has to be added to the list of blocks (see Figure 8.4(C).

D. w_i is the unique node of b_j and b_{j+1} does not exist. Therefore b_j is

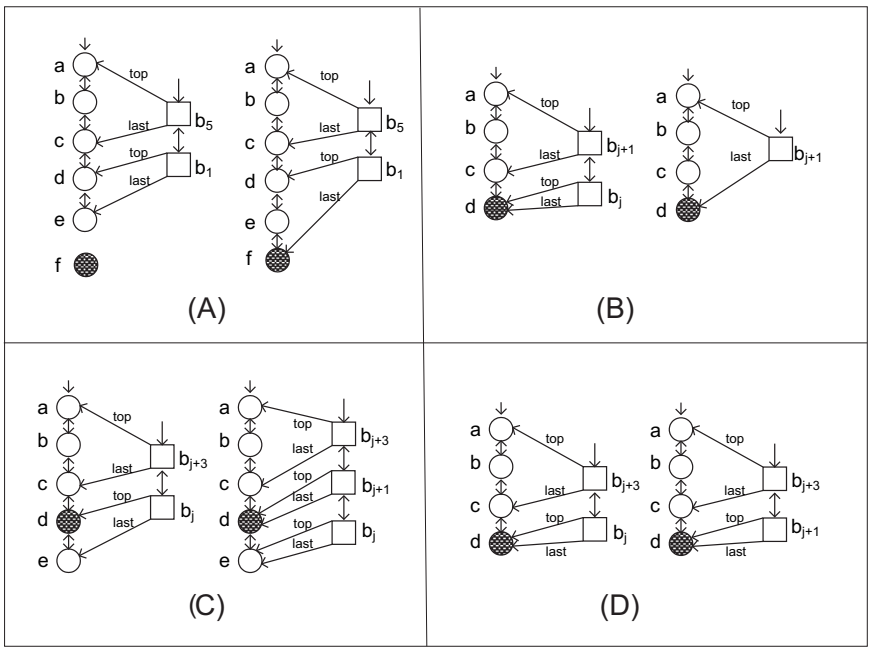


Figure 8.4: Distinct situations of increasing the frequency of a node

turned into block b_{j+1}, and no insertions or deletions of blocks are needed, as it is shown in Figure 8.4(D).

Several arrays and a variable denominated *availableBlock* define the list of blocks data structure:

- nodeInBlock[q] = i, $1 \leq q \leq N$, if and only if the node ranked q in the *node index* belongs to block i.
- topBlock[q] = i, $1 \leq q \leq N$, if and only if the node ranked i in *node index* is the *top* of block b_q.
- lastBlock[q] = i, $1 \leq q \leq N$, if and only if the node ranked i in the *node index* is the *last* node of block b_q.
- nextBlock[q] = nb, $1 \leq q \leq N$, if and only if the block b_{nb} is the next block of b_q in the list of blocks.
- previousBlock[q] = nb, $1 \leq q \leq N$, if and only if the block b_{nb} is the previous block of b_q in the list of blocks.
- freq[q] = w, $1 \leq q \leq N$, if and only if the frequency of all words in block b_q is w. Note that this vector is used instead of the one described in the *node Index* data structure.
- availableBlock indicates the first block that is unused. All unused blocks are linked together in a list using *nextBlock* array, and *availableBlock* is the header of that list or 1 if the available blocks list is empty.

8.6 Huffman tree update algorithm

The compressor/sender and decompressor/receiver algorithms use the general guidelines shown in Figure 7.1.

The update algorithm is presented next. It supposes that q is the rank in the *node index* of the node whose frequency has to be increased (or $q = zeroNode$ if a new word is been added to the tree).

```
update(q)
(1)  findNode(q); //sets q as the node to be incremented
(2)  while q ≠ 0 do; //bottom-up traversal of the tree
(3)       top = topBlock[nodeInBlock[q]];
(4)       if q ≠ top then;
(5)            if nodeType[q] =′ L′ then;  //q is a leaf node
(6)                 if nodeType[top] =′ L′ then;
(7)                      interchangeNodesLL (q, top);
(8)                 else  interchangeNodesIL (q, top);
(9)            else   //q is an internal node
(10)                if nodeType[top] =′ L′ then;
(11)                     interchangeNodesIL (top, q);
(12)                else  interchangeNodesII (top, q);
(13)           q = top;
(14)      increaseBlock(q);
(15) frequency[0] = frequency[0] + 1; //root's frequency ;
```

Interchanging two nodes depends on their type. Next three algorithms enable interchanging either two leaves, two internal nodes or both a leaf and an internal node.

```
interchangeNodesLL(leaf_i, leaf_j)
(1)  lNodePos[relPos[leaf_i]] ↔ lNodePos[relPos[leaf_j]];
(2)  relPos[leaf_i] ↔ relPos[leaf_j];
```

```
interchangeNodesIL(leaf, internal)
(1)  lNodePos[relPos[leaf]] ↔ iNodePos[relPos[internal]];
(2)  relPos[leaf] ↔ relPos[internal];
(3)  nodeType[leaf] ↔ nodeType[internal];
```

```
interchangeNodesII(i, j)
(1)  iNodePos[relPos[i]].nodePos ↔ iNodePos[relPos[j]].nodePos];
(2)  relPos[i] ↔ relPos[j];
```

Next, procedure *findNode*() is shown, it sets current q node, as the node to be incremented.

findNode(q)
(1) $bq = nodeInBlock[q]$;
(2) if $q = zeroNode$ then
(3) $pq = parent[q]$;
(4) if $(maxChild[pq] - minChild[pq]) < 255$ then //pq has less than 256
(5) $nbq = nextBlock[bq]$; //children, so it has space to hold "q".
(6) if $frequency[nbq] = 1$then //block "1" already exists
(7) $nodeInBlock[q] = 1$;
(8) $nodeInBlock[q + 1] = 0$;
(9) $lastBlock[1] = q$;
(10) $lastBlock[0] = q + 1$;
(11) $topBlock[0] = q + 1$;
(12) else //block "1" did not exists so it is created
(13) $bqI = availableBlock$;
(14) $availableBlock = nextBlock[availableBlock]$;
(15) $nextBlock[bqI] = nextBlock[bq]$;
(16) $previousBlock[bqI] = previousBlock[nbq]$;
(17) $nextBlock[bq] = bqI$;
(18) $previousBlock[nbq] = bqI$;
(19) $frequency[bqI] = 1$;
(20) $nodeInBlock[q] = 1$;
(21) $nodeInBlock[q + 1] = 0$;
(22) $topBlock[bqI] = q$;
(23) $lastBlock[bqI] = q$;
(24) $topBlock[bq] = q + 1$;
(25) $lastBlock[bq] = q + 1$;
(26) //q is added to the tree
(27) $addr = f_{hash}(s_i)$;
(28) $word[addr] = s_i$;
(29) $lNodePos[addr] = q$;
(30) $relPos[q] = addr$;
(31) $nodeType[q] =' L'$;
(32) $parent[q] = pq$;
(33) $zeroNode = zeroNode + 1$;
(34) $nodeType[zeroNode] =' L'$;
(35) $parent[zeroNode] = pq$;
(36) $intNodes[pq].maxChild = intNodes[pq].maxChild + 1$;
(37) else //a new internal node has to be created in position $zeroNode$
(38) $lastIntNode = lastIntNode + 1$;
(39) $npq = lastIntNode$;
(40) $relPos[q] = npq$;
(41) $nodeType[q] =' I'$;
(42) $parent[q] = pq$;

(43) $minChild[npq] = q + 1;$
(44) $maxChild[npq] = q + 2;$
(45) $iNodePos[npq] = q;$
(46) $nbq = nextBlock[bq];$
(47) if $frequency[nbq] = 1$ then //block "1" already exists
(48) $nodeInBlock[q] = 1;$
(49) $nodeInBlock[q + 1] = 1;$
(50) $nodeInBlock[q + 2] = 0;$
(51) $lastBlock[1] = q + 1;$
(52) $lastBlock[0] = q + 2;$
(53) $topBlock[0] = q + 2;$
(54) else //block "1" does not exist yet
(55) $bqI = availableBlock;$
(56) $availableBlock = nextBlock[availableBlock];$
(57) $nextBlock[bqI] = nextBlock[bq];$
(58) $previousBlock[bqI] = previousBlock[nbq];$
(59) $nextBlock[bq] = bqI;$
(60) $previousBlock[nbq] = bqI;$
(61) $frequency[bqI] = 1;$
(62) $nodeInBlock[q] = 1;$
(63) $nodeInBlock[q + 1] = 1;$
(64) $nodeInBlock[q + 2] = 0;$
(65) $topBlock[bqI] = q;$
(66) $lastBlock[bqI] = q + 1;$
(67) $topBlock[bq] = q + 2;$
(68) $lastBlock[bq] = q + 2;$
(69) $q = q + 1;$
(70) $addr = f_{hash}(currentWord);$
(71) $word[addr] = currentWord;$
(72) $lNodePos[addr] = q;$
(73) $relPos[q] = addr;$
(74) $nodeType[q] =' L';$
(75) $parent[q] = npq;$
(76) $zeroNode = zeroNode + 2;$
(77) $nodeType[zeroNode] =' L';$
(78) $parent[zeroNode] = npq;$
(79) $q = intNodes[pq].nodePos;$
(80) else //q is not the zeroNode, so it is already the node to be increased

Note that the function $f_{hash}(currentWord)$ used in lines 27 and 70 returns the position inside the hash table where $currentWord$ should be placed. Note also that in the case of the decompressor, as it needs only

an array of words, rather than a hash table, this lines should be replaced by next two instructions: $addr = nextFreeWord; nextFreeWord = nextFreeWord + 1$; where $nextFreeWord$ stores the next available free position in $words$ vector.

The last procedure used in $update()$ is called $increaseBlock$(q). This procedure is called in each step of the bottom-up traversal of the Huffman tree. It increases the frequency of the node q and sets q to point to its parent, next level upward in the tree.

---

increaseBlock(q)
(1) $bq = nodeInBlock[q]$;
(2) $nbq = nextBlock[bq]$;
(3) if $frequency[nbq] == (frequency[bq] + 1)$ then
(4) $nodeInBlock[q] = nbq$;
(5) $lastBlock[nbq] = q$;
(6) if $lastBlock[bq] == q$ then //q's old block disappears
(7) $previousBlock[nbq] = previousBlock[bq]$;
(8) $nextBlock[previousBlock[bq]] = nbq$;
(9) $nextBlock[bq] = availableBlock$;
(10) $availableBlock = bq$;
(11) else $topBlock[bq] = q + 1$;
(12) else //Next block does not exist
(13) if $lastBlock[bq] == q$ then
(14) $frequency[bq] = frequency[bq] + 1$;
(15) else
(16) $b = availableBlock$;
(17) $availableBlock = nextBlock[availableBlock]$;
(18) $nextBlock[bq] = b$;
(19) $previousBlock[b] = bq$;
(20) $nextBlock[b] = nbq$;
(21) $previousBlock[nbq] = b$;
(22) $topBlock[bq] = q + 1$;
(23) $topBlock[b] = q$;
(24) $lastBlock[b] = q$;
(25) $frequency[b] = frequency[bq] + 1$;
(26) $nodeInBlock[q] = b$;
(27) $q = iNodePos[parent[q]]$;

8.7 Empirical Results: Character- versus word-oriented Huffman

We compressed the real texts used in Section 10.5 to test the compression ratio and time performance of the character-based *FGK algorithm* (using the *compact* command) against the dynamic word-based byte-oriented Huffman code described in this Chapter.

Table 8.1 compares the compression ratio and compression speed of both techniques. As expected, for being a word-based technique, it reduces the compression ratio of the character-based dynamic Huffman technique to the half. Compression speed is also increased. The dynamic word-based byte-oriented Huffman (DynPH) reduces *FGK algorithm* compression time in about 6 times.

CORPUS	O SIZE	Dyn PH			*FGK* algorithm		
	bytes	bytes	ratio	time (sec)	Bytes	ratio	time(sec)
CALGARY	2,131,045	991,911	46.546	0.520	1,315,774	61.743	3.270
FT91	14,749,355	5,123,739	34.739	3.428	9,109,215	61.760	20.020
CR	51,085,545	15,888,830	31.102	11.450	31,398,726	61.463	71.010
FT92	175,449,235	56,185,629	32.024	41.330	108,420,660	61.796	259.125
ZIFF	185,220,215	60,928,765	32.895	44.628	155,010,899	83.690	264.570
FT93	197,586,294	63,238,059	32.005	47.118	124,501,969	63.011	272.375
FT94	203,783,923	65,126,566	31.959	48.260	128,274,442	62.946	278.210
AP	250,714,271	80,964,800	32.294	60.702	155,010,899	61.828	361.120
ALL_FT	591,568,807	187,586,995	31.710	143.050	370,551,380	62.639	832.134
ALL	1,080,719,883	355,005,679	32.849	268.983	678,287,443	62.763	1,523.890

Table 8.1: Comparison of word-based and character-based dynamic approaches

In Section 10.5, more empirical results, comparing the Dynamic word-based byte-oriented Huffman technique explained in this Chapter against several well-known compression techniques, are also presented.

8.8 Conclusions

Our dynamic word-based byte-oriented Huffman code and the data structures that support efficiently the update process of the Huffman tree in each step of both compression and decompression processes were presented.

Word-based Huffman codes, are known to obtain good compression. For this sake, we adapted an existing algorithm so as to handle very large sets of source words and byte-oriented output. The latter decision sacrifices some compression ratio in exchange for an 8-fold improvement in time performance. The resulting algorithm obtains compression ratio very similar to its static version (only 0.06% off) and compresses about 4 megabytes per second on a standard PC.

Some empirical results comparing the new technique with the *FGK algorithm*, a good character-based dynamic Huffman method, were presented. Those results show that our implementation clearly improves the compression ratio and reduces the compression time achieved by the character-based *FGK*.

As a result, we have obtained an adaptive natural language text compressor that obtain 30%-32% compression ratio for large texts, and compresses more than 4 megabytes per second. Empirical results also show its good performance when it is compared against other compressors such as *gzip*, *bzip2* and arithmetic encoding (see Section 10.5 for more details).

Having set the efficiency of this algorithm, we will use it as a competitive compression technique to prove that dynamic versions of both End-Tagged Dense Code and (s, c)-Dense Code are, as their *two-pass* versions, good alternatives to Huffman compression techniques for their competitive compression ratio, compression and decompression speed and mainly for the ease of their implementation.

9

Dynamic End-Tagged Dense Code

9.1 Chapter overview

This Chapter presents our first contribution to dynamic compression, a dynamic version of the End-Tagged Dense Code. First the motivation of this new technique is explained. Next, its basis are presented in Section 9.3. The data structures needed to make it an efficient adaptive technique, are explained in Section 9.4. Section 9.5 presents pseudocode for both compressor and decompressor processes. It includes the algorithms to adapt the model used, whenever a symbol is input. Empirical results, comparing the compression ratio of both the dynamic version against the semi-static approach are given in Section 9.6. Finally, some conclusions regarding to this new technique are shown.

9.2 Introduction

In Chapter 8 a good word-based byte-oriented dynamic Huffman code was presented. That code joins the good compression ratios achieved by the word-based semi-static statistical Huffman methods (in this case,

Plain Huffman method[48]) with the advantages of adaptive compression techniques in file transmission scenarios. The resulting code permits real-time transmission of data and achieves compression ratios really close to the semi-statical version of the code.

However, this algorithm is rather complex to implement and the update of the Huffman tree is expensive. Each time a symbol s_i is input, it is required to perform a full bottom-up traversal of the tree to update the weights of all the ancestors of s_i until reaching the *root* (and at each level, reorganizations can happen).

In Chapter 5 we describe End-Tagged Dense Code, a statistical word-based compression technique (not using Huffman) which we first presented in [13]. End-Tagged Dense Code is shown as a good alternative to Huffman due to several features:

- It is really simple to build since only a list of words ranked by frequency is needed by both the encoding and decoding processes.
- End-Tagged Dense Codes are faster than Huffman based codes.
- The lose of compression ratio compared against the Plain Huffman code is small (about 1 percentage point).
- It enables direct searching the compressed text.

For the first three reasons it seemed interesting to develop a Dynamic End-Tagged Dense Code in this thesis. This new code is shown in the remainder sections of this Chapter, and improves the compression/decompression speed of the dynamic Huffman-based technique (about 22%-26% faster in compression and 35% in decompression), at the expense of loosing a little bit of compression ratio (one percentage point).

9.3 Method overview

In this Section it is shown how ETDC can be made dynamic. Considering again the general scheme of Figure 7.1, the main issue is how to maintain the

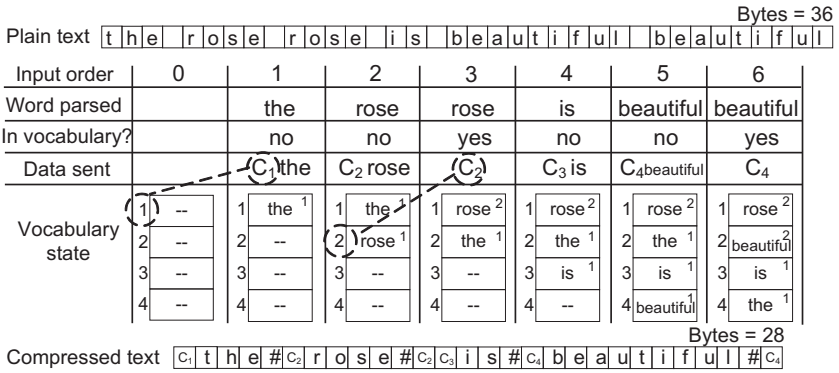


Figure 9.1: Transmission of message `"the rose rose is beautiful beautiful"`

CodeBook up to date upon insertions of new source symbols and frequency increments.

In the case of ETDC, the *CodeBook* consists essentially on one structure that keeps the vocabulary ordered by frequency. Since we maintain such sorted vocabulary upon insertions and frequency changes, we can encode any source symbol or decode any target symbol by using the on-the-fly *encode* and *decode* procedures explained in Section 5.4.

Figure 9.1 shows how should the compressor operate. At first (step 0), no words have been read so *new-Symbol* is the only word in the vocabulary (it is implicitly placed at position 1). In step 1, a new symbol `"the"` is read. Since it is not in the vocabulary, C_1 (the codeword of *new-Symbol*) is sent, followed by `"the"` in plain form (bytes `'t'`, `'h'`, `'e'` and some terminator `'#'`). Next, `"the"` is added to the vocabulary with frequency 1, at position 1. Implicitly, *new-Symbol* has been displaced to position 2. Step 2 shows the transmission of `"rose"`, which is not yet in the vocabulary. In step 3, `"rose"` is read again. As it was in the vocabulary at position 2, only the codeword C_2 is sent. Now, `"rose"` becomes more frequent than `"the"`, so it moves upwards in the ordered vocabulary. Note that a hypothetical new occurrence of `"rose"` would be transmitted as C_1, while it was sent as C_2 in step 3. In steps 4 and 5, two more new words, `"is"` and `"beautiful"`, are

transmitted and added to the vocabulary. Finally, in step 6, `"beautiful"` is read again, and it becomes more frequent than `"is"` and `"the"`. Therefore, it moves upwards in the vocabulary by means of an exchange with `"the"`.

The receiver works similarly to the sender. It receives a codeword C_i, and decodes it. As result of decoding C_i, a symbol S_i is obtained. If C_i corresponds to the *new-Symbol*, then the receiver knows that a new word s_i will be received in plain form next, so s_i is received and added to the vocabulary. When C_i corresponds to a word s_i that is already in the vocabulary, the receiver only has to increase its frequency.

The main challenge is how to efficiently maintain the vocabulary sorted. In Section 9.4 it is shown how to do it with a complexity equal to the number of source symbols transmitted. This is always lower than *FGK* complexity, because at least one target symbol must be transmitted for each source symbol, and usually several more.

Essentially, we must be able to identify *groups* of words with the same frequency in the ordered vocabulary, and be able to fast promote a word to the next group when its frequency increases. Promoting a word w_i with frequency f to next frequency group $f+1$ consists of:

- Sliding w_i over all words whose frequency is f. This operation implies two operations:
  - Locating the first ranked word in the ordered vocabulary whose frequency is f. This word is called top_f.
  - Interchanging w_i with top_f
- Increasing the frequency of w_i

Example 9.3.1 Suppose that the ranked vocabulary is composed of words {'A', 'B', 'C', 'D', 'E', 'F', 'G'} with frequencies {3, 3, 2, 1, 1, 1, 1}, and a new 'G' comes, so its frequency has to be increased.
The frequency of 'G' is 1, and the first word in the vocabulary with frequency 1 is 'D'. Therefore, 'D' and 'G' are interchanged. Now it is possible to increase the frequency of 'G', and the vocabulary remains sorted. So the

resulting vocabulary contains words {'A', 'B', 'C', 'G', 'E', 'F', 'D'} and their frequencies are {3, 3, 2, 2, 1, 1, 1} respectively. □

The whole data structures and the way they are used to maintain the sorted vocabulary efficiently are shown next.

9.4 Implementation Data structures

As with word-based dynamic Huffman (Chapter 8), the sender maintains a hash table that permits fast searching for a source word s_i, this time to obtain its rank i in the vocabulary vector (remember that to encode a word s_i only its rank i is needed), as well as its current frequency f_i (which is used to rapidly find the position top_{f_i}).

The receiver does not need to maintain a hash table to hold words. It only needs to use a *word vector*, because the decoding process generates directly a rank value i that can be used to index the *word vector*. Therefore, it never has to find a word lexicographically.

Lets n be the vocabulary size, F the maximum frequency value expected for any word in the vocabulary and $N = max\{n, F\}$. The data structures used by both the sender and the receiver, as well as their functionality are shown in Subsections 9.4.1 and 9.4.2 respectively.

9.4.1 Sender's Data Structures

The following three main data structures are needed:

- A *hash table* of words keeps in *word* the source word characters, in *posInVoc* the rank (or position) of the word in the ordered vocabulary, and in *freq* its frequency. Both *word*, *posInVoc* and *freq* are defined as H-elements arrays (having $H = nextPrime(2n)$).
- In the *posInHT* array, each position represents a specific word of the vocabulary. Words are not explicitly represented, but a pointer to

word vector in the hash table is stored. *posInHT* is defined as a n-elements vector. This vector keeps words ordered by frequency. That is, $posInHT[1]$ points to the most frequent word, $posInHT[2]$ to the second most frequent word, and so on.

- Array *top* is defined as a N-elements array. Each position represents implicitly a value of frequency. That is, *top*[1] represents words whose frequency is 1, *top*[2] represents words with frequency 2, an so on. For each possible frequency, *top* vector keeps a pointer to the hash table entry where the first word with that frequency is stored.

  gives, for each possible frequency, the vocabulary array position of the first word with that frequency. It is defined as a N-elements array.

Two more variables *new-Symbol* and *maxFreq* hold the first free position in the vocabulary (in *posInHT*) and the position "*higher frequency of current words* $+1$" (associated to *top* vector) respectively.

9.4.2 Receiver's Data Structures

The following three vectors are needed:

- A *word* vector that keeps the source word characters. It consists of a n-elements array.
- A *freq* vector that keeps the frequency of each word. That is, $freq[i] = f$, if the number of occurrences of the word stored in $word[i]$ is f. As the word array, this vector can keep up to n elements.
- Array *top*. As it happened in the sender, this array gives, for each possible frequency, the word position of the first word with that frequency. It is also defined here, as a N-elements array.

Variables *new-Symbol* and *maxFreq* are also needed by the receiver.

Next Section explains the way both sender and receiver work, and how they use the data structures just presented to make sending/compression and reception/decompression processes efficient.

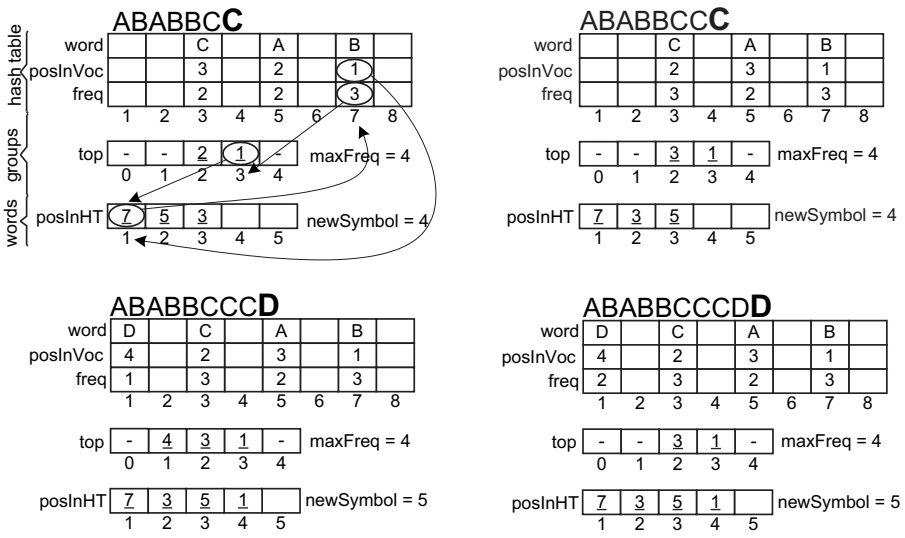


Figure 9.2: Transmission of words C, C, D and D having transmitted ABABB earlier.

9.5 Sender's and Receiver's pseudocode

When the sender reads word s_i, it uses the hash function to obtain its position p in the hash table, so that $f_{hash}(s_i) = p$ and therefore $word[p] = s_i$. After reading $f = freq[p]$, it increments $freq[p]$. The position of s_i in the vocabulary array is also obtained as $i = posInVoc[p]$ (so it will send code C_i). Now, word s_i must be promoted to its next group. For this sake, sender's algorithm finds the head of its group $j = top[f]$ and the corresponding word position $h = posInHT[j]$, so as to swap words i and j in the vector $posInHT$. The swapping requires exchanging $posInHT[j]$ with $posInHT[i] = p$, setting $posInVoc[p] = j$ and setting $posInVoc[h] = i$. Once the swapping is done, we promote j to the next group by setting $top[f] = j + 1$.

If s_i turns out to be a new word, we set $word[p] = s_i$, $freq[p] = 0$, and $posInVoc[p]$ =*new-Symbol*. Then exactly the above procedure is followed with $f = 0$ and i =*new-Symbol*. Finally also *new-Symbol* is increased.

Figure 9.2 explains the way the sender works and how its data structures are used.

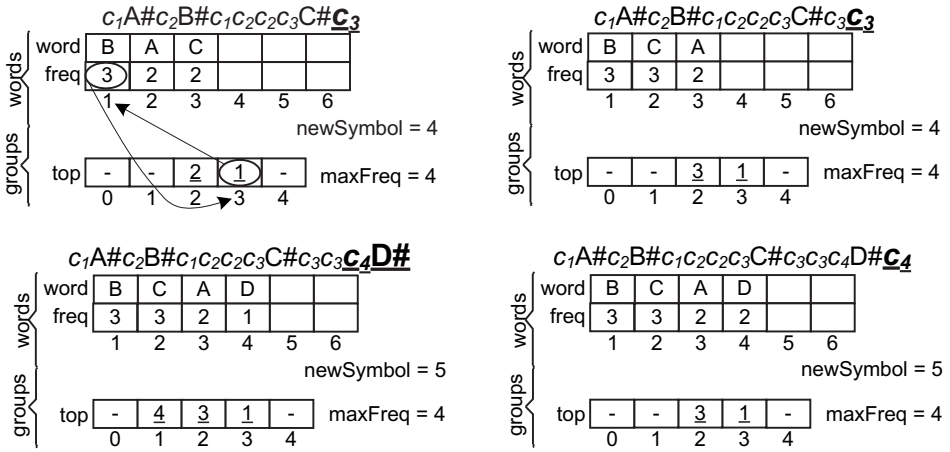


Figure 9.3: Reception of c_3, c_3, $c_4D\#$ and c_4 having received $c_1\mathrm{A}\#c_2\mathrm{B}\#c_1c_2c_2c_3\mathrm{C}\#$ previously.

The receiver works very similarly to the sender, but it is even simpler. Given a codeword C_i, the receiver decodes it using the decode algorithm in Page 64, achieving the position i such as decode$(C_i) = i$ and $word[i]$ contains the word s_i that corresponds to C_i. Therefore $word[i]$ can be output. Next it performs $f = freq[i]$ and then increases $freq[i]$. In order to promote s_i in the vocabulary, $j = top[i]$ is located. In next step $word[i]$ and $word[j]$ as well as freq[i] and $freq[j]$ are swapped. Finally j is promoted to the group of frequency $f + 1$ by setting $top[f] = j + 1$.

If $i = new\text{-}Symbol$ then a new word s_i is being transmitted in plain form. We set $word[i] = s_i$, $freq[i] = 0$, and again the previous process is performed with $f = 0$ and $i =$*new-Symbol*. Finally *new-Symbol* is also increased.

Figure 9.3 gives an example of how the receiver works.

Pseudocode for both sender and receiver processes is shown in Figures 9.4 and 9.5. Notice that implementing dynamic ETDC is simpler than building dynamic word-based Huffman. In fact, our implementation of the Huffman tree update (Section 8.6) takes about 120 C source code lines, while the update procedure takes less than 20 lines in dynamic ETDC.

```
Sender main algorithm ()
(1)  new-Symbol ← 1;
(2)  top[0] ← new-Symbol;
(3)  maxFreq ← 1;
(4)  while more words left
(5)       read s_i from text;
(6)       if (s_i ∉ word) then
(7)            p ← f_hash(s_i);
(8)            i ← new-Symbol;
(9)            send (encode(i));
(10)           send s_i in plain form;
(11)      else
(12)           i ← posInVoc[p];
(13)           send (encode(i));
(14)      update();
```

```
Sender update ()
(1)  if i =new-Symbol then // new word
(2)       word[p] ← s_i;
(3)       freq[p] ← 0;
(4)       posInVoc[p] ← new-Symbol ;
(5)       posInHT[new-Symbol] ← p;
(6)       new-Symbol ← new-Symbol + 1;
(7)  f ← freq[p];
(8)  freq[p] ← freq[p] + 1;
(9)  j ← top[f];
(10) h ← posInHT[j];
(11) posInHT[i] ↔ posInHT[j];
(12) posInVoc[p] ← j;
(13) posInVoc[h] ← i;
(14) top[f] ← j + 1;
(15) if maxFreq = f + 1 then
(16)      top[f + 1] ← 0;
(17)      maxFreq ← maxFreq + 1;
```

Figure 9.4: Dynamic ETDC sender pseudocode

```
Receiver main algorithm ()
(1)  new-Symbol ← 1;
(2)  top[0] ← new-Symbol;
(3)  maxFreq ← 1;
(4)  while more codewords remain
(5)       i ← decode(c_i);
(6)       if i =new-Symbol then
(7)            receive s_i in plain form;
(8)            output s_i;
(9)       else
(10)           output word[i];
(11)      update();
```

```
Receiver update ()
(1)  if i =new-Symbol then // new word
(2)       word[i] ← s_i;
(3)       freq[i] ← 0;
(4)       new-Symbol ← new-Symbol + 1;
(5)  f ← freq[i];
(6)  freq[i] ← freq[i] + 1;
(7)  j ← top[f];
(8)  freq[i] ↔ freq[j];
(9)  word[i] ↔ word[j];
(10) top[f] ← j + 1;
(11) if maxFreq = f + 1 then
(12)      top[f + 1] ← 0;
(13)      maxFreq ← maxFreq + 1;
```

Figure 9.5: Dynamic ETDC sender pseudocode

9.6 Empirical Results

We compressed the real texts used in Section 10.5 to test the compression ratio of the one- and two-pass versions of End-Tagged Dense Code (ETDC) and Plain Huffman (PH).

Table 9.1 compares the compression ratio of two-pass versus one-pass techniques. Columns labelled **diff** measure the increase, in percentage points, in the compression ratio of the dynamic codes compared to their semi-static version. The last column compares those differences between Plain Huffman and ETDC.

CORPUS	TEXT SIZE	Plain Huffman			End-Tagged Dense Code			diff
		2-pass	1-pass	diff	2-pass	1-pass	diff	ETDC
	bytes	ratio%	ratio%	PH	ratio%	ratio%	ETDC	-PH
CALGARY	2,131,045	46.238	46.546	0.308	47.397	47.730	0.332	0.024
FT91	14,749,355	34.628	34.739	0.111	35.521	35.638	0.116	0.005
CR	51,085,545	31.057	31.102	0.046	31.941	31.985	0.045	-0.001
FT92	175,449,235	32.000	32.024	0.024	32.815	32.838	0.023	-0.001
ZIFF	185,220,215	32.876	32.895	0.019	33.770	33.787	0.017	-0.002
FT93	197,586,294	31.983	32.005	0.022	32.866	32.887	0.021	-0.001
FT94	203,783,923	31.937	31.959	0.022	32.825	32.845	0.020	-0.002
AP	250,714,271	32.272	32.294	0.021	33.087	33.106	0.018	-0.003
ALL_FT	591,568,807	31.696	31.710	0.014	32.527	32.537	0.011	-0.003
ALL	1,080,719,883	32.830	32.849	0.019	33.656	33.664	0.008	-0.011

Table 9.1: Compression ratios of dynamic versus semi-static techniques

Compression ratios are around 31-34% in the larger texts. In the smaller, compression is poor because the size of the vocabulary is proportionally too large with respect to the compressed text size.

From the experimental results, it can also be seen that the cost of dynamism in terms of compression ratio is negligible. The dynamic versions lose very little in compression (about 0.02 percentage points, 0.06%) compared to their semi-static versions. Moreover, in most texts (the negative values in the last column) dynamic ETDC loses even less compression than the dynamic Plain Huffman.

9.7 Conclusions

In this Chapter it was considered the problem of providing adaptive compression for natural language text, with the combined aim of competitive compression ratios and good time performance.

We adapted a simpler alternative to Huffman, End-Tagged Dense Code (ETDC) to make it *one-pass*. The resulting dynamic version is much simpler than the Huffman-based one (proposed in Chapter 8). This is because maintaining ordered a list of words is much simpler than adapting a Huffman tree (and less operations are performed).

As result, Dynamic ETDC is 22%-26% faster, compressing typically 5.5 megabytes per second. Moreover, the compressed text is only 0.06% larger than with semi-static ETDC and 3% larger than with Huffman. In Section 10.5 empirical results comparing both the semi-statical against the *one-pass* version of ETDC, as well as the comparative against other well-known compression techniques are presented.

10

Dynamic (s, c)-Dense Code SIN TERMINAR !!

!! ESTE CAPÍTULO NO ESTÁ FINALIZADO!!! pero los resultados empíricos son VÁLIDOS, a falta de mejorar el Dyn-SCDC.

...AÚN TENGO QUE :

- REVISAR LA IMPLEMENTACIÓN preliminar QUE HICIMOS DEL Dyn-SCDC sobre las ideas que hablamos cuando estuviste aquí en A Coruña.

  Actualmente tengo una versión preliminar y (con las pocas pruebas que hice con la versión nuevo) mejoramos un poco (sobre todo en textos pequeños) los tiempos del Dyn-SCDC respecto a los tiempos que se incluyen en la sección de resultados empíricos (con la versión vieja)

  Quiero revisar un poco la implementación, y después ¡tomar más tiempos para que las medias sean fiables!

- CONTAR COMO FUNCIONA ESA NUEVA VERSIÓN. Secciones 10.3 y 10.4

10.1 Chapter overview

This Chapter presents the last contribution of this thesis. It consists of the development of a new adaptive compression technique named *Dynamic (s,c)-Dense Code* (D-SCDC). D-SCDC is a generalization of the Dynamic End-Tagged Dense Code, in which not only the vocabulary is dynamically maintained sorted, but also the s and c parameters can vary along the compression/decompression processes in order to achieve better compression.

The Chapter is structured as follows: First the motivation of this new technique is introduced. In Section 10.3, Dynamic (s, c)-Dense Code is described. It treats the similarities between Dynamic End-Tagged Dense Code and Dynamic (s, c)-Dense Code data structures, as well as those differences that arise due to the necessity of adapting the s and c parameters. Then, the way the s and c parameters are maintained optimal is presented, and its pseudocode is given in Section 10.4. Section 10.5 shows empirical results where the three new dynamic techniques developed are compared with the *two-pass* versions, as well as with other well-known compressors such as *gzip*, *gzip2* and an adaptive arithmetic compressor [38]. Finally some conclusions are presented in Section 10.6.

10.2 Introduction

In the previous two chapters, two dynamic word-based byte-oriented techniques were defined.

The first one is based on Plain Huffman. It achieves better compression than the second one, Dynamic End-Tagged Dense Code (ETDC) (about 3 points in percentage of the compressed text). However it is slower in compression and decompression (about 20 % in large texts).

As shown in Chapter 6, (s, c)-Dense Code [14] is a compression technique that generalizes End-Tagged Dense Code and obtains compression ratios very close to those of the optimal Plain Huffman Code.

The work being presented in the current Chapter involves building an

adaptive version of (s, c)-Dense Code [14]. It consists of an extension to ETDC where the number of byte values that signal the end of a codeword can be adapted to optimize compression, instead of being fixed at 128 as in ETDC.

The remainder of this Chapter treats the similarities between Dynamic ETDC and Dynamic (s, c)-Dense Code data structures, as well as those differences appeared due to the necessity of adapting s and c. Finally empirical results and some conclusions are presented.

10.3 Dynamic (s, c)-Dense Codes

In this Section it is shown how to generalize Dynamic ETDC to build Dynamic (s, c)-Dense Code (Dynamic SCDC).

Based on Dynamic ETDC, Dynamic (s, c)-DC also uses the scheme presented in Figure 7.1. The *CodeBook* is essentially the array of source symbols sorted by frequencies, and the on-the-fly *encode* and *decode* procedures explained in Section 6.5, are used.

The main difference with respect to Dynamic ETDC is that, in each step of the compression/decompression processes, it is mandatory not only to maintain the sorted vocabulary, but also to check if current s value remains optimal or if it should change.

10.3.1 Maintaining the s and c parameters optimal

Both encoder and decoder starts with $s = 256$. This s value is optimal while the first 255 words of the vocabulary are input. When word 256 arrives, s has to be decreased by 1 since a two-bytes codeword is needed. From this point on, the optimal s and c values are estimated depending on the number of bytes needed to encode, using *current* s *and* c values, the last processed input word, and comparing that value with the number of bytes that would be need if the *current* s value would had been $prev_s = s - 1$ or $next_s = s + 1$ (remember that $c = 256 - s$).

Three variables are needed: *prev*, s_0 and *next*. *prev* holds an estimation of the size of the compressed text when the value $s-1$ is used as the number of *stoppers* in the encoding/decoding process. s_0 and *next* accumulates the size of the compressed text, supposing that the last word input was encoded using the current s value or $s+1$ respectively.

First, the tree variables are initialized to zero. Each time a symbol s_i arrives, *prev*, s_0 and *next* are increased. Lets *countBytes(s_value, i)* be a function that computes the number of bytes needed to encode the word ranked i having $s = s\_value$.

Therefore the three variables are increased as follows:

- $prev \leftarrow prev + countBytes(s-1, i)$
- $s_0 \leftarrow s_0 + countBytes(s, i)$
- $next \leftarrow prev + countBytes(s+1, i)$

A change of the s value takes place when $prev < s_0$ or when $next < s_0$. If $prev < s_0$ then the $s-1$ becomes the new s value. When $next < s_0$ holds, then $s+1$ is set as the new s value.

Each time the parameter s changes, s_0, *next* and *prev* are initialized again, and the process continues. This initialization depends on the change of s that takes place.

- If s is increased then $prev \leftarrow s_0$; and $s_0 \leftarrow next$ (*next* does not change).
- If s is decreased then $next \leftarrow s_0$; and $s_0 \leftarrow prev$ (*prev* does not change).

The update algorithm to maintain the list of words sorted is the same used in the case of Dynamic End-Tagged Dense Code, in Figures 9.4 and 9.5. However the test that indicates a possible change of the s value, has to be performed once after each call to the update process.

10.4 Pseudocode for parameters check and change

Two main procedures are used to check if s and c values have to change after having processed a new input symbol. The first one is *countBytes()*. This algorithm calculates, given a s_i value and a word ranked i_{pos} in the vocabulary, the number of bytes needed to encode the word i_{pos} using $s = s_i$.

The second algorithm, *TakeIntoAccount()* is invoked after each call to the *update()* procedure. This algorithm checks either if $prev < s_0$ or $next < s_0$, and changes the s value used in compression or decompression if needed.

The pseucode to both algorithms is shown in Figure 10.1.

10.5 Empirical Results

We tested the different compressors over several texts. As representative of short texts, we used the Calgary corpus. We also used some large text collections from TREC-2 (AP Newswire 1988 and Ziff Data 1989-1990) and from TREC-4 (Congressional Record 1993, Financial Times 1991 to 1994). Finally, two larger collections, ALL_FT and ALL, were used. ALL_FT aggregates all texts from Financial Times collection. ALL collection is composed by Calgary corpus and all texts from TREC-2 and TREC-4.

We first compressed the texts to test the compression ratio and time performance of the one- and two-pass versions of End-Tagged Dense Code (ETDC), (s, c)-Dense Code (SCDC) and Plain Huffman (PH). These results are shown in Subsection 10.5.1. Next, corpora were also compressed with two well-known techniques such as *gzip* and *bzip2* and a word-based arithmetic compressor [16]. Results showing compression and decompression time and compression ratio are presented in Tables 10.3, 10.4 and 10.5.

The spaceless word model [48] was used to model the separators. A separator is the text between two contiguous words, and it must be coded too. In the spaceless word model, if the separator following a word is a single whitespace, we just encode the word, otherwise both the word and the separator are encoded. Hence, the vocabulary is formed by all the different

```
countBytesAlgorithm (s_i, i_pos)
    //Calculates the size of the encoding of word ranked i_pos
    //when using s = s_i in the compression process
(1) k ← 1;
(2) last ← s_i;
(3) pot ← s_i;
(4) c_i ← 256 − s_i;
(5) while last <= i_pos
(6)     pot ← pot ∗ c_i;
(7)     last ← last + pot;
(8)     k ← k + 1;
(9) return k;
```

```
TakeIntoAccount (s,i)
    //It changes, if needed, the s value than will be used in
    //both compression and decompression processes.
(1) prev ← countBytes (s − 1, i);
(2) s_0 ← countBytes (s, i);
(3) if prev < s_0 then
(4)     s ← s − 1; //s is decreased
(5)     c ← c + 1;
(6)     next ← s_0;
(7)     s_0 ← prev;
(8) else
(9)     next ← countBytes (s + 1, i);
(10)    if next < s_0 then
(11)        s ← s + 1; //s is increased
(12)        c ← c − 1;
(13)        prev ← s_0;
(14)        s_0 ← next;
```

Figure 10.1: Algorithm to change s and c parameters if needed

words and all the different separators, excluding the single whitespace.

A dual Intel®Pentium®-III 800 Mhz system, with 768 MB SDRAM-100Mhz was used in our tests. It ran Debian GNU/Linux (kernel version 2.2.19). The compiler used was gcc version 3.3.3 20040429 and `-O9` compiler optimizations were used. Time results measure CPU user-time.

10.5.1 Semi-static Vs dynamic approach: Compression ratio

Table 10.1 compares the compression ratios of the two-pass versus the one-pass versions of ETDC, SCDC and PH. Columns labeled $\mathbf{diff}_{PH}$, $\mathbf{diff}_{ETDC}$ and $\mathbf{diff}_{SCDC}$ measure the increase, in percentage points, in the compression ratio of the dynamic codes compared against their semi-static version. The last columns show those differences between PH and ETDC, and between PH and SCDC respectively.

CORPUS	TEXT SIZE	Plain Huffman			End-Tagged Dense Code		
		2-pass	dynamic	Increase	2-pass	dynamic	Increase
	bytes	ratio %	ratio %	$\mathbf{diff}_{PH}$	ratio %	ratio %	$\mathbf{diff}_{ETDC}$
CALGARY	2,131,045	46.238	46.546	0.308	47.397	47.730	0.332
FT91	14,749,355	34.628	34.739	0.111	35.521	35.638	0.116
CR	51,085,545	31.057	31.102	0.046	31.941	31.985	0.045
FT92	175,449,235	32.000	32.024	0.024	32.815	32.838	0.023
ZIFF	185,220,215	32.876	32.895	0.019	33.770	33.787	0.017
FT93	197,586,294	31.983	32.005	0.022	32.866	32.887	0.021
FT94	203,783,923	31.937	31.959	0.022	32.825	32.845	0.020
AP	250,714,271	32.272	32.294	0.021	33.087	33.106	0.018
ALL_FT	591,568,807	31.696	31.710	0.014	32.527	32.537	0.011
ALL	1,080,719,883	32.830	32.849	0.019	33.656	33.664	0.008

CORPUS	(s,c)-Dense Code			$\mathbf{diff}_{ETDC}$	$\mathbf{diff}_{SCDC}$
	2-pass	dynamic	Increase	$-$ $\mathbf{diff}_{PH}$	$-$ $\mathbf{diff}_{PH}$
	ratio %	ratio %	$\mathbf{diff}_{SCDC}$		
CALGARY	46.611	46.809	0.198	0.024	-0.110
FT91	34.875	34.962	0.086	0.005	-0.025
CR	31.291	31.332	0.041	-0.001	-0.005
FT92	32.218	32.237	0.020	-0.001	-0.004
ZIFF	33.062	33.078	0.016	-0.002	-0.003
FT93	32.178	32.202	0.024	-0.001	0.002
FT94	32.132	32.154	0.021	-0.002	-0.001
AP	32.542	32.557	0.015	-0.003	-0.006
ALL_FT	31.839	31.849	0.010	-0.003	-0.004
ALL	33.018	33.029	0.011	-0.011	-0.008

Table 10.1: Compression ratio of dynamic versus semi-static techniques

As it can be seen, a very small lose of compression occurs when converting the techniques into dynamic codes. To understand this increase of size of dynamic versus semi-static codes, two issues have to be considered: (i)

each new word s_i parsed during dynamic compression is represented in the compressed text (or sent to the receiver) as a pair $\langle C_{new\text{-}Symbol}, s_i \rangle$, while in two-pass compression only the word s_i needs to be stored/transmitted in the vocabulary; (ii) on the other hand, some low-frequency words can be encoded with shorter codewords by dynamic techniques, since by the time they appear the vocabulary may still be small.

Compression ratios are around 31%-33% for large texts. For the smallest one (Calgary collection), compression is poor because the size of the vocabulary is proportionally too large with respect to the compressed text size (as expected from Heaps' law [23]). This means that proportionally too many words are transmitted in plain form. The increase of compression ratio in ETDC compared against PH is always under 1 percentage point, in the larger texts.

On the other hand, the dynamic versions lose very little compression (not more than 0.05 percentage points in general) compared to their semi-static versions. This shows that the price paid by dynamism in terms of compression ratio is practically negligible. Note also that in most cases, dynamic ETDC loses even less compression than dynamic Plain Huffman, while adding dynamism to SCDC only produces more loss of compression ratio than PH, in the FT93 corpus.

10.5.2 Comparative of Dynamic PH, Dynamic ETDC and Dynamic SCDC: compression speed and compression ratio

Table 10.2 compares the time performance and the compression ratio of our three dynamic compressors. The first three columns indicate the corpora, their size and the size of the vocabulary (number of distinct words parsed).

The latter four columns (columns "Dyn ETDC Vs Dyn PH" and "Dyn SCDC Vs Dyn PH") measure the increase of compression ratio and the reduction of compression time (in percentage) of Dynamic ETDC and Dynamic SCDC, with respect to Dynamic PH.

As it can be seen, Dynamic ETDC loses less than 1 percentage point

CORPUS	TEXT SIZE bytes	n	Dyn PH		Dyn ETDC	
			time (sec)	ratio%	time (sec)	ratio %
CALGARY	2,131,045	30.995	0.520	46.546	0.384	47.730
FT91	14,749,355	75.681	3.428	34.739	2.488	35.638
CR	51,085,545	117.713	11.450	31.102	8.418	31.985
FT92	175,449,235	284.892	41.330	32.024	31.440	32.838
ZIFF	185,220,215	237.622	44.628	32.895	33.394	33.787
FT93	197,586,294	291.427	47.118	32.005	36.306	32.887
FT94	203,783,923	295.018	48.260	31.959	36.718	32.845
AP	250,714,271	269.141	60.702	32.294	47.048	33.106
ALL_FT	591,568,807	577.352	143.050	31.710	111.068	32.537
ALL	1,080,719,883	886.190	268.983	32.849	213.068	33.664

CORPUS	Dyn SCDC		Dyn ETDC Vs Dyn PH		Dyn SCDC Vs Dyn PH	
	time (sec)	ratio	Inc. size %	Decr. time	Inc. size %	Decr. time
CALGARY	0.410	46.809	2.543	22.892	0.565	17.671
FT91	2.660	34.962	2.588	22.685	0.642	17.340
CR	9.010	31.332	2.839	22.629	0.737	17.188
FT92	33.980	32.237	2.542	26.404	0.667	20.459
ZIFF	36.197	33.078	2.710	22.559	0.557	16.060
FT93	37.923	32.202	2.755	20.840	0.614	17.314
FT94	39.240	32.154	2.774	22.006	0.610	16.649
AP	50.610	32.557	2.514	22.796	0.816	16.951
ALL_FT	120.073	31.849	2.609	23.796	0.438	17.617
ALL	227.747	33.029	2.481	25.927	0.548	21.134

Table 10.2: Comparison of dynamic ETDC, dynamic SCDC and dynamic PH

of compression ratio (about 3% increase of size) with respect to Dynamic Plain Huffman, in the larger texts. In exchange, it is 22%-26% faster and considerably simpler to implement.

In the case of Dynamic SCDC the loss of compression ratio with respect to Dynamic Plain Huffman is about 0.25 percentage points what implies an increase of the size of the compressed text of less that 1%. Moreover, the compression speed is improved about 17%.

Dynamic Plain Huffman compresses 4 megabytes per second, while Dynamic SCDC reaches 5, and Dynamic ETDC compression speed is about 5.5 megabytes per second.

10.5.3 Comparative against *gzip*, *bzip2* and arithmetic compressors

Tables 10.3, 10.4 and 10.5 compare Dynamic PH, Dynamic SCDC and Dynamic ETDC against *gzip* (Ziv-Lempel family), *bzip2* (Burrows-Wheeler [15] type technique) and an adaptive arithmetic compressor (arith) [38].

Experiments were run setting *gzip* and *bzip2* parameters to both "best" (-b) and "fast" (-f) compression.

CORPUS	compression ratio %							
	D-PH	D-SCDC	D-ETDC	arith	gzip -f	gzip -b	bzip2 -f	bzip2 -b
CALGARY	46.546	46.809	47.730	34.679	43.530	36.840	32.827	28.924
FT91	34.739	34.962	35.638	28.331	42.566	36.330	32.305	27.060
CR	31.102	31.332	31.985	26.301	39.506	33.176	29.507	24.142
FT92	32.024	32.237	32.838	29.817	42.585	36.377	32.369	27.088
ZIFF	32.895	33.078	33.787	26.362	39.656	32.975	29.642	25.106
FT93	32.005	32.202	32.887	27.889	40.230	34.122	30.624	25.322
FT94	31.959	32.154	32.845	27.860	40.236	34.122	30.535	25.267
AP	32.294	32.557	33.106	28.002	43.651	37.225	33.260	27.219
ALL_FT	31.710	31.849	32.537	27.852	40.988	34.845	31.152	25.865
ALL	32.849	33.029	33.664	27.982	41.312	35.001	31.304	25.981

Table 10.3: Comparison of compression ratio against *gzip*, *bzip2* and arithmetic compression

As expected "bzip2 -b" achieves the best compression ratio. It is about 5-7 percentage points better than Dynamic PH (and hence a little bit more with respect to Dynamic SCDC and Dynamic ETDC). However, it is much slower than the other techniques tested in both compression and decompression processes. Using the "fast" *bzip2* option seems to be undesirable, since compression ratio gets worse (it becomes closer to Dynamic PH) and compression and decompression speeds remain poor.

On the other hand, "gzip -f" is shown to achieve good compression speed, at the expense of compression ratio (about 40%). It is shown that Dynamic ETDC is also a fast technique. It is able to beat "gzip -f" in compression speed (except in the ALL corpus). Dynamic SCDC is also really close to "gzip -f" in compression speed (an even sometimes better than it). Regarding to compression ratio, Dynamic ETDC achieves also best results than "gzip -b" (except in CALGARY and ZIFF corpora). However, *gzip* is clearly the fastest method in decompression. Note that "gzip -f" is slower than "gzip -b" in decompression. This is because of decompression has to performed over a smaller compressed file (because "gzip -best" compresses more than "gzip -fast").

Regarding to decompression speed, Dynamic PH decompresses about 6 megabytes per second, while Dynamic SCDC and Dynamic ETDC achieves about 8 and 9 respectively. Therefore, Dynamic SCDC decompresses 35% faster than Dynamic PH, and Dynamic ETDC is 50% faster in

decompression.

Hence, Dynamic ETDC is either much faster or compresses much better than *gzip*, and it is by far faster than *bzip2*. Dynamic SCDC is also a good alternative to gzip. It is almost so fast as "gzip -f" in compression, and its compression ratio is better. Regarding to Dynamic PH, it is a well-balanced technique. It is about 22%-26% slower than Dynamic ETDC (17% with respect to Dynamic SCDC) and its compression ratios are even more competitive.

With respect to the arithmetic compressor used (it also uses a word-based approach, but it is bit-oriented rather than byte-oriented), it is possible to note the advantages in compression ratio of bit-oriented word-based techniques. However it can be seen how compression and decompression speed is about the half of Dynamic PH and about 3 times slower than the byte-oriented techniques presented.

CORPUS	compression time (sec)							
	D-PH	D-SCDC	D-ETDC	arith	gzip -f	gzip -b	bzip2 -f	bzip2 -b
CALGARY	0.498	0.410	0.384	1.030	0.360	1.095	2.180	2.660
FT91	3.218	2.660	2.488	6.347	2.720	7.065	14.380	18.200
CR	10.880	9.010	8.418	21.930	8.875	25.155	48.210	65.170
FT92	42.720	33.980	31.440	80.390	34.465	84.955	166.310	221.460
ZIFF	43.122	36.197	33.394	82.720	33.550	82.470	174.670	233.250
FT93	45.864	37.923	36.306	91.057	36.805	93.135	181.720	237.750
FT94	47.078	39.240	36.718	93.467	37.500	96.115	185.107	255.220
AP	60.940	50.610	47.048	116.983	50.330	124.775	231.785	310.620
ALL_FT	145.750	120.073	91.068	274.310	117.255	293.565	558.530	718.250
ALL	288.778	227.747	213.905	509.710	188.310	532.645	996.530	1,342.430

Table 10.4: Comparison of compression speed against *gzip*, *bzip2* and arithmetic compression

CORPUS	Decompression time (sec)							
	D-PH	D-SCDC	D-ETDC	arith	gzip -f	gzip -b	bzip2 -f	bzip2 -b
CALGARY	0.330	0.275	0.240	0.973	0.090	0.110	0.775	0.830
FT91	2.350	1.790	1.545	5.527	0.900	0.825	4.655	5.890
CR	7.745	5.940	5.265	18.053	3.010	2.425	15.910	19.890
FT92	30.690	21.630	19.415	65.680	8.735	7.390	57.815	71.050
ZIFF	30.440	23.405	20.690	67.120	9.070	8.020	58.790	72.340
FT93	32.780	24.498	21.935	71.233	10.040	9.345	62.565	77.860
FT94	33.550	25.320	22.213	75.925	10.845	10.020	62.795	80.370
AP	43.660	31.745	27.233	88.823	15.990	13.200	81.875	103.010
ALL_FT	104.395	75.535	66.238	214.180	36.295	30.430	189.905	235.370
ALL	218.745	144.168	126.938	394.067	62.485	56.510	328.240	432.390

Table 10.5: Comparison of decompression time against *gzip*, *bzip2* and arithmetic technique

10.6 Conclusions

In this Chapter, Dynamic ETDC was extended to make variable the number of stoppers and continuers that can be used during the encoding/decoding processes.

The resulting code achieves compression ratios really close to the Huffman based tecnique presented in Chapter 8 while it is faster to build.

With respect to Dynamic ETDC, Dynamic SCDC worsen a little in compression and decompression speed, however compression ratios are better and really close to Huffman values.

Empirical results comparing our three contributions to the State of Art in dynamic text compression techniques, one against each other, as well a comparison against well-known compressors have been presented. As a result, we have obtained adaptive natural language text compressors that obtain about 30%-35% compression ratio, compress more than 4 megabytes per second and decompresses over 6 megabytes per second.

11

Conclusions and Future Work

Through the work contained in this thesis, four compression codes have been developed.

The first is a semi-statical code is denominated (s, c)-Dense Code. It is well-suited for its integration into a Text Retrieval system. It almost achieves the same compression ratio that the optimal Plain Huffman Code [49, 48] and maintains the good features of other similar compression schemas as the Tagged Huffman Code:

- It is a word-based technique.
- It generates a prefix code encoding.
- It enables decompression of random portions of a compressed text, for using a tag condition that allows to distinguish codes inside the compressed text.
- It improves searches with respect to the search over the uncompressed text (about 8 times faster) and enables direct searching.

This encoding incorporates new advantages over the previous ones:

- Its compression ratios are better than Tagged Huffman and also than End-Tagged Dense Code.
- Encoding and Decoding is faster and simpler than Huffman based Methods.

Summarizing, an almost optimal, fast and simple compression technique was developed.

The other three techniques developed are: *i*) a Dynamic word-based byte-oriented Huffman code, a Dynamic End-Tagged Dense Code and an Adaptive version of the (s, c)-Dense Code. These techniques are based in the semi-statical word based Plain Huffman Code, End-Tagged Dense Code and in the (s, c)-Dense Code respectively.

They are appropriated for both file compression and network transmission. Their main contributions are the following ones:

- They join the real-time capabilities of the Adaptive Huffman character-oriented existing techniques with the good compression ratios achieved the the statistical two-pass word-based compression techniques.
- Being byte-oriented, compression and decompression speed is high.

As future work, we are now interested in the application of (s, c)-Dense Code to indexing scenarios, for example in the implementation of block addressing indexes, and also to use it with suffix arrays [5], to test (s, c)-Dense Code in compressed suffix arrays.

In the case of the dynamic techniques their are directly applicable to file compression, and would enable for example the transmission of compressed web pages, or even to introduce compression in other real-time scenarios as chat/talk services. Therefore implementing a web-browser plug-in would enable comparing the transmission of plain web pages with the compressed ones.

Appendix A

Publications and Other Research Results Related to the Thesis

A.1 !!!FALTA !!!

A.2 Publications

A.2.1 International Conferences

A.2.2 National Conferences

A.2.3 Journals and Book Chapters

A.3 Research Stays

Appendix B

Descripción del Trabajo Realizado

B.1 !!!FALTA !!!

B.2 Introducción

B.3 Metodología Utilizada

B.4 Conclusiones y Trabajo Futuro

Bibliography

[1] N Abramson. *Information Theory and Coding*. McGraw-Hill, 1963.

[2] N. Abramson. *Information Theory and Coding*. McGraw-Hill, New York, 1963.

[3] R. A. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[4] Ricardo Baeza-Yates and Gaston H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, October 1992.

[5] Ricardo A. Baeza-Yates, R. Baeza-Yates, and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[6] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman, May 1999.

[7] Bernhard Balkenhol and Yuri M. Shtarkov. One attempt of a compression algorithm using the BWT, 1999.

[8] T. Bell, J. Cleary, , and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

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

[10] Timothy Bell, Ian H. Witten, and John G. Cleary. Modeling for text compression. *ACM Comput. Surv.*, 21(4):557–591, 1989.

[11] Jon Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Commun. ACM*, 29(4):320–330, 1986.

[12] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

[13] Nieves R. Brisaboa, Eva L. Iglesias, Gonzalo Navarro, and José R. Paramá. An efficient compression code for text databases. In *25th European Conference on IR Research, ECIR 2003; LNCS 2633*, pages 468–481, Pisa, Italy, 2003.

[14] N.R. Brisaboa, A. Fariña, G. Navarro, and M.F. Esteller. (s,c)-dense coding: An optimized compression code for natural language text databases. In *Proc. 10th International Symposium on String Processing and Information Retrieval (SPIRE 2003)*, LNCS 2857, pages 122–136, 2003.

[15] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, 1994.

[16] John Carpinelli, Alistair Moffat, Radford Neal, Wayne Salamonsen, Lang Stuiver, Andrew Turpin, and Ian Witten. Word, character, integer, and bit based compression using arithmetic coding. Available at http:www.cs.mu.oz.auãlistairarith_coder, 1999.

[17] J. G. Cleary, W. J. Teahan, and I. H. Witten. Unbounded length contexts for PPM. In *Data Compression Conference*, pages 52–61, 1995.

[18] John G. Cleary and Ian H. Witten. Data compression using ADaptive coding and partial string matching. IEEE *Trans. Comm.*, 32(4):396–402, 1984.

[19] N Faller. An adaptive system for data compression. In *In Record of the 7th Asilomar Conference on Circuits, Systems, and Computers*, pages 593–597, 1973.

[20] Peter Fenwick. Block sorting text compression - final report. Technical report, April 23 1996.

[21] Philip Gage. A new algorithm for data compression. *C Users Journal*, 12(2):23–??, February 1994.

[22] R.G Gallager. Variations on a theme by Huffman. *IEEE Trans. on Inf. Theory*, 24(6):668–674, 1978.

[23] H. S. Heaps. *Information Retrieval: Computational and Theoretical Aspects*. Academic Press, New York, 1978.

[24] Daniel S. Hirschberg and Debra A. Lelewer. Efficient decoding of prefix codes. *Commun. ACM*, 33(4):449–459, 1990.

[25] R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10:501–506, 1980.

[26] Paul Glor Howard. The design and analysis of efficient lossless data compression systems. Technical Report CS-93-28, 1993.

[27] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. Inst. Radio Eng.*, pages 1098–1101, September 1952. Published as Proc. Inst. Radio Eng., volume 40, number 9.

[28] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[29] Donald E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6(2):163–180, June 1985.

[30] L. G. Kraft. A device for quanitizing, grouping and coding amplitude modulated pulses. Master's thesis, Mater's Thesis, Department of Electrical Engineering, MIT, Cambridge, MA, 1949.

[31] Udi Manber and Sun Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. of the Winter 1994 USENIX Technical Conference*, pages 23–32, 1994.

[32] M. Miyazaki, S. Fukamachi, M. Takeda, and T. Shinohara. Speeding up the pattern matching machine for compressed texts. *Transactions of Information Processing Society of Japan*, 39(9):2638–2648, 1998.

[33] Moffat. Implementing the PPM data compression scheme. *IEEETCOMM: IEEE Transactions on Communications*, 38, 1990.

[34] A. Moffat. Word-based text compression. *Software - Practice and Experience*, 19(2):185–198, 1989.

[35] A. Moffat and J. Katajainen. In-place calculation of minimum-redundancy codes. In S.G. Akl, F. Dehne, and J.-R. Sack, editors, *Proc. Workshop on Algorithms and Data Structures (WADS'95)*, LNCS 955, pages 393–402, 1995.

[36] A. Moffat, R. Neal, and I. H. Witten. Arithmetic coding revisited. In J. A. Storer and M. Cohn, editors, *Proc. IEEE Data Compression Conference*, pages 202–211, Snowbird, Utah, March 1995. IEEE Computer Society Press, Los Alamitos, California.

[37] A. Moffat and A. Turpin. *Compression and Coding Algorithms.* Kluwer_Academic Publ., March 2002.

[38] Alistair Moffat, Radford M. Neal, and Ian H. Witten. Arithmetic coding revisited. *ACM Trans. Inf. Syst.*, 16(3):256–294, 1998.

[39] Alistair Moffat and Turpin. On the implementation of minimum redundancy prefix codes. *IEEETCOMM: IEEE Transactions on Communications*, 45:170–179, 1996.

[40] Navarro, Edleno Silva de Moura, M. Neubert, Nivio Ziviani, and Ricardo Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.

[41] Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In R. Giancarlo and D. Sankoff, editors, *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, number 1848 in Lecture Notes in Computer Science, pages 166–180, Montréal, Canada, 2000. Springer-Verlag, Berlin.

[42] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences.* Cambridge University Press, 2002. ISBN 0-521-81307-7. 280 pages.

[43] J. Rautio, J. Tanninen, and J. Tarhio. String matching with stopper encoding and code splitting. In *Proc. 13th Annual Symposium on Combinatorial Pattern Matching (CPM 2002)*, LNCS 2373, pages 42–52, 2002.

[44] Eugene S. Schwartz and Bruce Kallick. Generating a canonical prefix encoding. *Commun. ACM*, 7(3):166–169, 1964.

[45] C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, Illinois, 1949.

[46] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, and S. Arikawa. A Boyer-Moore type algorithm for compressed pattern matching. In *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching (CPM'00)*, LNCS 1848, pages 181–194, 2000.

[47] Ayumi Shinohara, Masayuki Takeda, Shuichi Fukamachi, Takeshi Shinohara, Takuya Kida, and Yusuke Shibata. Byte pair encoding: A text compression scheme that accelerates pattern matching. April 19 1999.

[48] Edleno Silva de Moura, G. Navarro, Nivio Ziviani, and Ricardo Baeza-Yates. Fast searching on compressed text allowing errors. In W. Bruce Croft, Alistair Moffat, C. J. van Rijsbergen, Ross Wilkinson, and Justin Zobel, editors, *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR-98)*, pages 298–306, New York City, August 24–28 1998. ACM Press.

[49] Edleno Silva de Moura, G. Navarro, Nivio Ziviani, and Ricardo Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, April 2000.

[50] Daniel M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, August 1990.

[51] M. Takeda, Y. Shibata, T. Matsumoto, T. Kida, A. Shinohara, S. Fukamachi, T. Shinohara, and S. Arikawa. Speeding up string pattern matching by text compression: The dawn of a new era. *Transactions of Information Processing Society of Japan*, 42(3):370–384, 2001.

[52] J.S. Vitter. Design and analysis of dynamic Huffman codes. *Journal of the ACM (JACM)*, 34(4):825–845, 1987.

[53] J.S. Vitter. Algorithm 673: dynamic Huffman coding. *ACM Transactions on Mathematical Software (TOMS)*, 15(2):158–167, 1989.

[54] Terry A. Welch. A technique for high performance data compression. *Computer*, 17(6):8–20, June 1984.

[55] Ian H. Witten and T. C. Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4):1085–1094, 1991.

[56] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers, USA, 1999.

[57] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, 1987.

[58] Sun Wu and Udi Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, October 1992.

[59] George K. Zipf. *Human Behavior and the Principle of Least Effort*. Addison-Wesley (Reading MA), 1949.

[60] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[61] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[62] Nivio Ziviani, Edleno Silva de Moura, G. Navarro, and Ricardo Baeza-Yates. Compression: A key for next-generation text retrieval systems. *IEEE Computer*, 33(11):37–44, 2000.

[63] Justin Zobel and Alistair Moffat. Adding compression to a full-text retrieval system. *Software Practice and Experience*, 25(8):891–903, August 1995.