Fast and Flexible Word Searching on Compressed Text

EDLENO SILVA DE MOURA
Universidade Federal de Minas Gerais, Brazil
GONZALO NAVARRO
Universidad de Chile
NIVIO ZIVIANI
Universidade Federal de Minas Gerais, Brazil
and
RICARDO BAEZA-YATES
Universidad de Chile

We present a fast compression and decompression technique for natural language texts. The novelties are that (i) decompression of arbitrary portions of the text can be done very efficiently, (ii) exact search for words and phrases can be done on the compressed text directly, using any known sequential pattern matching algorithm and (iii) word-based approximate and extended search can also be done efficiently without any decoding. The compression scheme uses a semi-static word-based model and a Huffman code where the coding alphabet is byte-oriented rather than bit-oriented. We compress typical English texts to about 30% of their original size, against 40% and 35% for *Compress* and *Gzip*, respectively. Compression time is close to that of *Compress* and approximately half the time of *Gzip*, and decompression time is lower than that of *Gzip* and one third of that of *Compress*.

We present three algorithms to search the compressed text. They allow a large number of variations over the basic word and phrase search capability, such as sets of characters, arbitrary regular expressions and approximate matching. Separators and stopwords can be discarded at search time without significantly increasing the cost. When searching for simple words, the experiments show that running our algorithms on a compressed text is twice as fast as running the best existing software on the uncompressed version of the same text. When searching complex or approximate patterns, our algorithms are up to 8 times faster than the search on uncompressed text. We also discuss the impact of our technique in inverted files pointing to logical blocks and argue for the possibility of keeping the text compressed all the time, decompressing only for displaying purposes.

Categories and Subject Descriptors: E.4 [**Coding and Information Theory**]: Data Compaction

This work has been partially supported by SIAM Project, grant MCT/FINEP/PRONEX 76.97.1016.00, AMYRI/CYTED Project, CAPES scholarship (E. S. de Moura), Fondecyt grant 1990627 (G. Navarro and R. Baeza-Yates) and CNPq grant 520916/94-8 (N. Ziviani).
Authors' addresses: E. S. de Moura and N. Ziviani, Dept. of Computer Science, Univ. Federal de Minas Gerais, Av. Antonio Carlos 6627, Belo Horizonte, Brazil; G. Navarro and R. Baeza-Yates, Dept. of Computer Science, Univ. de Chile, Av. Blanco Encalada 2120, Santiago, Chile.

and Compression; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*search process*

General Terms: Text Compression, Searching Compressed Texts

Additional Key Words and Phrases: Compressed pattern matching, Natural language text compression, Word searching, Word-based huffman coding

1. INTRODUCTION

In this paper we present an efficient compression technique for natural language texts that allows fast and flexible searching of words and phrases. To search for simple words and phrases, the patterns are compressed and the search proceeds without any decoding of the compressed text. Searching words and phrases that match complex expressions and/or allowing errors can be done on the compressed text at almost the same cost of simple searches. The reduced size of the compressed text makes the overall searching time much smaller than on plain uncompressed text. The compression and decompression speeds and the amount of compression achieved are very good when compared to well known algorithms in the literature [Ziv and Lempel 1977; Ziv and Lempel 1978].

The compression scheme presented in this paper is a variant of the word-based Huffman code [Bentley et al. 1986; Moffat 1989; Witten et al. 1999]. The Huffman codeword assigned to each text word is a sequence of whole bytes and the Huffman tree has degree either 128 (which we call "tagged Huffman code") or 256 (which we call "plain Huffman code"), instead of 2. In tagged Huffman coding each byte uses 7 bits for the Huffman code and 1 bit to signal the beginning of a codeword. As we show later, using bytes instead of bits does not significantly degrade the amount of compression. In practice, byte processing is much faster than bit processing because bit shifts and masking operations are not necessary at compression, decompression and search times. The decompression can start at any point in the compressed file. In particular, the compression scheme allows fast decompression of fragments that contain the search results, which is an important feature in information retrieval systems.

Notice that our compression scheme is designed for large natural language texts containing at least 1 megabyte to achieve an attractive amount of compression. Also, the search algorithms are *word oriented* as the pattern is a sequence of elements to be matched to a sequence of text *words*. Each pattern element can be a simple word or a complex expression, and the search can be exact or allowing errors in the match. In this context, we present three search algorithms.

The first algorithm, based on tagged Huffman coding, compresses the pattern and then searches for the compressed pattern directly in the compressed text. The search can start from any point in the compressed text because all the bytes that start a codeword are marked with their highest bit set in 1. Any conventional pattern matching algorithm can be used for exact searching and a multi-pattern matching algorithm is used for searching allowing errors, as explained later on.

The second algorithm searches on a plain Huffman code and is based on a word-oriented Shift-Or algorithm [Baeza-Yates and Gonnet 1992]. In this case the com-

pression obtained is better than with tagged Huffman code because the search algorithm does not need any special marks on the compressed text.

The third algorithm is a combination of the previous ones, where the pattern is compressed and directly searched in the text as in the first algorithm based on tagged Huffman coding. However, it works on plain Huffman code, where there is no signal of codeword beginnings, and therefore the second algorithm is used to check a surrounding area in order to verify the validity of the matches found.

The three algorithms allow a large number of variations over the basic word and phrase searching capability, which we group under the generic name of *extended patterns*. As a result, classes of characters including character ranges and complements, wild cards, and arbitrary regular expressions can be efficiently searched exactly or allowing errors in the occurrences. Separators and very common words (stopwords) can be discarded without significantly increasing the search cost.

The algorithms also allow "approximate phrase matching". They are able to search in the compressed text for approximate occurrences of a phrase pattern allowing insertions, deletions or replacements of words. Approximate phrase matching can capture different writing styles and therefore improve the quality of the answers to the query. Our algorithms are able to perform this type of search at the same cost of the other cases, which is extremely difficult on uncompressed search.

Our technique is not only useful to speed up sequential search. It can also be used to improve indexed schemes that combine inverted files and sequential search, like *Glimpse* [Manber and Wu 1993]. In fact, the techniques that we present here can nicely be integrated to the inverted file technology to obtain lower space-overhead indexes. Moreover, we argue in favor of keeping the text compressed all the time, so the text compression cannot be considered an extra effort anymore.

The algorithms presented in this paper are being used in a software package called *Cgrep*. *Cgrep* is an exact and approximate compressed matching tool for large text collections. The software is available from `ftp://dcc.ufmg.br/latin/cgrep`, as a prototype. Preliminary partial versions of this article appeared in [Moura et al. 1998a; Moura et al. 1998b].

This paper is organized as follows. In Section 2 we discuss the basic concepts and present the related work found in the literature. In Section 3 we present our compression and decompression method, followed by analytical and experimental results. In Section 4 we show how to perform exact and extended searching on tagged Huffman compressed texts. In Section 5 we show how to perform exact and extended searching on plain Huffman compressed texts. In Section 6 we present experimental results about the search performance. Finally, in Section 7 we present conclusions and suggestions for future work.

2. BASICS AND RELATED WORK

Text compression is about exploiting redundancies in the text to represent it in less space [Bell et al. 1990]. In this paper we denote the uncompressed file as T and its length in bytes as u. The compressed file is denoted as Z and its length in bytes as n. *Compression ratio* is used in this paper to denote the size of the compressed file as a percentage of the uncompressed file (i.e. $100 \times n/u$).

From the many existing compression techniques known in the literature we emphasize only the two that are relevant for this paper. A first technique of interest

is the Ziv-Lempel family of compression algorithms, where repeated substrings of arbitrary length are identified in the text and the repetitions are replaced by pointers to their previous occurrences. In these methods it is possible that $n = o(u)$, achieving $n = O(\sqrt{u})$ and even $n = O(\log u)$ in the best cases.

A second technique is what we call "zero-order substitution" methods. The text is split into *symbols* and each symbol is represented by a unique *codeword*. Compression is achieved by assigning shorter codewords to more frequent symbols. The best known technique of this kind is the minimum redundancy code, also called Huffman code [Huffman 1952]. In Huffman coding, the codeword for each symbol is a sequence of bits so that no codeword is a prefix of another codeword and the total length of the compressed file is minimized. In zero-order substitution methods we have $n = \Theta(u)$, even though the constant can be smaller than 1. Moreover, there are $\Theta(u)$ symbols in a text of u characters (bytes) and $\Theta(n)$ codewords in a compressed text of n bytes. In this work, for example, we use $O(u)$ to denote the number of words in T.

The *compressed matching problem* was first defined in the work of Amir and Benson [Amir and Benson 1992] as the task of performing string matching in a compressed text without decompressing it. Given a text T, a corresponding compressed string Z, and an (uncompressed) pattern P of length m, the compressed matching problem consists in finding all occurrences of P in T, using only P and Z. A naive algorithm, which first decompresses the string Z and then performs standard string matching, takes time $O(u+m)$. An optimal algorithm takes worst-case time $O(n + m)$. In [Amir et al. 1996], a new criterion, called *extra space*, for evaluating compressed matching algorithms, was introduced. According to the extra space criterion, algorithms should use at most $O(n)$ extra space, optimally $O(m)$ in addition to the n-length compressed file.

The first compressed pattern matching algorithms dealt with Ziv-Lempel compressed text. In [Farach and Thorup 1995] was presented a compressed matching algorithm for the LZ1 classic compression scheme [Ziv and Lempel 1976] that runs in $O(n \log^2(u/n)+m)$ time. In [Amir et al. 1996], a compressed matching algorithm for the LZ78 compression scheme was presented, which finds the first occurrence in $O(n + m^2)$ time and space, or in $O(n \log m + m)$ time and in $O(n + m)$ space. An extension of [Amir et al. 1996] to multipattern searching was presented in [Kida et al. 1998], together with the first experimental results in this area. New practical results appeared in [Navarro and Raffinot 1999], which presented a general scheme to search on Ziv-Lempel compressed texts (simple and extended patterns) and implemented it for the particular cases of LZ77, LZ78 and a new variant proposed which was competitive and convenient for search purposes. A similar result, restricted to the LZW format, was independently found and presented in [Kida et al. 1999]. Finally, [Kida et al. 1999] generalized the existing algorithms and nicely unified the concepts in a general framework.

All the empirical results obtained roughly coincide in a general figure: searching on a Ziv-Lempel compressed text 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. That is, the search algorithms are useful if the text has to be kept compressed anyway, but they do not give an extra reason to compress. The compression ratios are about 30% to 40% in practice when

a text is compressed using Ziv-Lempel.

A second paradigm is zero-order substitution methods. As explained, $n = \Theta(u)$ in this model, and therefore the theoretical definition of compressed pattern matching makes little sense because it is based in distinguishing $O(u)$ from $O(n)$ time. The goals here, as well as the existing approaches, are more practical: search directly the compressed text faster than the uncompressed text, taking advantage of its smaller size.

A first text compression scheme that allowed direct searching on compressed text was proposed by Manber [Manber 1997]. This approach packs pairs of frequent characters in a single byte, leading to a compression ratio of approximately 70% for typical text files.

A particularly successful trend inside zero-order substitution methods has been Huffman coding where the text words are considered the symbols that compose the text. The *semi-static* version of the model is used, that is, the frequencies of the text symbols is learned in a first pass over the text and the text is coded in a second pass. The table of codewords assigned to each symbol is stored together with the compressed file. This model is better suited to typical information retrieval scenarios on large text databases, mainly because the data structures can be shared (the vocabulary of the text is almost the same as the symbol table of the compressor), local decompression is efficient, and better compression and faster search algorithms are obtained (it is possible to search *faster* on the compressed than on the uncompressed text). The need for two passes over the text is normally already present when indexing text in information retrieval applications, and the overhead of storing the text vocabulary is negligible for large texts. On the other hand, the approach is limited to word-based searching on large natural language texts, unlike the Ziv-Lempel approach.

To this paradigm belongs [Turpin and Moffat 1997], a work developed independently of our work. The paper presents an algorithm to search on texts compressed by a word-based Huffman method, allowing only exact searching for one-word patterns. The idea is to search for the compressed pattern codeword in the compressed text.

Our work is based on a similar idea, but uses bytes instead of bits for the coding alphabet. The use of bytes presents a small loss in the compression ratio and the gains in decompression and search efficiency are large. We also extend the search capabilities to phrases, classes of characters, wild cards, regular expressions, exactly or allowing errors (also called "approximate string matching").

The *approximate string matching problem* is to find all substrings in a text database that are at a given "distance" k or less from a pattern P. The *distance* between two strings is the minimum number of insertions, deletions or substitutions of single characters in the strings that are needed to make them equal. The case in which $k = 0$ corresponds to the classical exact matching problem.

Approximate string matching is a particularly interesting case of extended pattern searching. The technique is useful to recover from typing, spelling and optical character recognition errors. The problem of searching a pattern in a compressed text allowing errors is an open problem in [Amir et al. 1996]. We partially solve this problem, since we allow approximate word searching. That is, we can find text *words* that match a pattern word with at most k errors. Note the limitations of this

statement: if a single error inserts a space in the middle of `"flower"`, the result is a sequence of two words, `"flo"` and `"wer"`, none of which can be retrieved by the pattern `"flowers"` allowing one error. A similar problem appears if a space deletion converts `"many flowers"` into a single word.

The best known software to search uncompressed text with or without errors is *Agrep* [Wu and Manber 1992]. We show that our compressed pattern matching algorithms compare favorably against *Agrep*, being up to 8 times faster depending on the type of search pattern. Of course *Agrep* is not limited to word searching and does not need to compress the file prior to searching. However, this last argument can in fact be used in the other direction: we argue that thanks to our search algorithms and to new techniques to update the compressed text, the text files can be kept compressed all the time and be decompressed only for displaying purposes. This leads to an economy of space and improved overall efficiency.

For all the experimental results of this paper we used natural language texts from the TREC collection [Harman 1995]. We have chosen the following texts: AP - Newswire (1989), DOE - Short abstracts from DOE publications, FR - Federal Register (1989), WSJ - Wall Street Journal (1987, 1988, 1989) and ZIFF - articles from *Computer Selected* disks (Ziff-Davis Publishing). Table 1 presents some statistics about the five text files. We considered a word as a contiguous maximal string of characters in the set {A...Z, a...z, 0...9}. All tests were run on a SUN SparcStation 4 with 96 megabytes of RAM running Solaris 2.5.1.

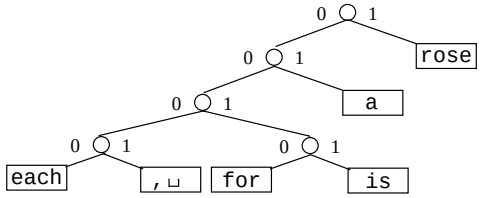
Files	Text		Vocabulary		Vocab./Text	
	Size (bytes)	#Words	Size (bytes)	#Words	Size	#Words
AP	237,766,005	38,977,670	1,564,050	209,272	0.65%	0.53%
DOE	181,871,525	28,505,125	1,949,140	235,133	1.07%	0.82%
FR	219,987,476	34,455,982	1,284,092	181,965	0.58%	0.52%
WSJ	262,757,554	42,710,250	1,549,131	208,005	0.59%	0.48%
ZIFF	242,660,178	39,675,248	1,826,349	255,107	0.75%	0.64%

Table 1. Some statistics of the text files used from the TREC collection.

3. THE COMPRESSION SCHEME

General compression methods are typically adaptive as they allow the compression to be carried out in one pass and there is no need to keep separately the parameters to be used at decompression time. However, for natural language texts used in a full-text retrieval context, adaptive modeling is not the most effective compression technique.

Following [Moffat 1989; Witten et al. 1999], we chose to use word-based semi-static modeling and Huffman coding [Huffman 1952]. In a semi-static model the encoder makes a first pass over the text to obtain the frequency of each different text word and performs the actual compression in a second pass. There is one strong reason for using this combination of modeling and coding. The data structures associated with them include the list of words that compose the vocabulary of the text, which we use to derive our compressed matching algorithm. Other important



Original Text: `for each rose, a rose is a rose`

Compressed Text: `0010 0000 1 0001 01 1 0011 01 1`

Fig. 1. A canonical tree and a compression example using binary Huffman coding for spaceless words.

reasons in text retrieval applications are that decompression is faster on semi-static models, and that the compressed text can be accessed randomly without having to decompress the whole text as in adaptive methods. Furthermore, previous experiments have shown that word-based methods give good compression ratios for natural language texts [Bentley et al. 1986; Moffat 1989; Horspool and Cormack 1992].

Since the text is not only composed of words but also of separators, a model must also be chosen for them. In [Moffat 1989; Bell et al. 1993] two different alphabets are used: one for words and one for separators. Since a strict alternating property holds, there is no confusion about which alphabet to use once it is known that the text starts with word or separator.

We use a variant of this method to deal with words and separators that we call *spaceless words*. If a word is followed by a space, we just encode the word. If not, we encode the word and then the separator. At decoding time, we decode a word and assume that a space follows, except if the next symbol corresponds to a separator. In this case the alternating property does not hold and a single coding alphabet is used. This idea was firstly presented in [Moura et al. 1997], where it is shown that the spaceless word model achieves slightly better compression ratios. Figure 1 presents an example of compression using Huffman coding for spaceless words method. The set of symbols in this case is {`"a"`, `"each"`, `"is"`, `"for"`, `"rose"`, `",␣"`}, whose frequencies are 2, 1, 1, 1, 3, 1, respectively.

The number of Huffman trees for a given probability distribution is quite large. The preferred choice for most applications is the *canonical tree*, defined by Schwartz and Kallick [Schwartz and Kallick 1964]. The Huffman tree of Figure 1 is a canonical tree. It allows more efficiency at decoding time with less memory requirement. Many properties of the canonical codes are mentioned in [Hirschberg and Lelewer 1990; Zobel and Moffat 1995; Witten et al. 1999].

3.1 Byte-Oriented Huffman Code

The original method proposed by Huffman [Huffman 1952] is mostly used as a binary code. That is, each symbol of the input stream is coded as a sequence of bits. In this work the Huffman codeword assigned to each text word is a sequence of whole bytes and the Huffman tree has degree either 128 (in this case the eighth

bit is used as a special mark to aid the search) or 256, instead of 2. In *all* cases from now on, except otherwise stated, we consider that

—the words and separators of the text are the symbols,

—the separators are codified using the spaceless word model,

—canonical trees are used,

—and the symbol table, which is the vocabulary of the different text words and separators, is kept compressed using the classical binary Huffman coding on characters.

We now define the different types of Huffman codes used in this work, all of which adhere to the above points.

Binary Huffman Code A sequence of bits is assigned to each word or separator.

Byte Huffman Code A sequence of bytes is assigned to each word or separator. This encompasses the two coding schemes that follow.

Plain Huffman Code A byte Huffman coding where all the bits of the bytes are used. That is, the Huffman tree has degree 256.

Tagged Huffman Code A byte Huffman coding where only the 7 lower order bits of each byte are used. That is, the Huffman tree has degree 128. The highest bit of each byte is used as follows: the first byte of each codeword has the highest bit in 1, while the other bytes have their highest bit in 0. This is useful for direct searching on the compressed text, as explained later.

All the techniques for efficient encoding and decoding mentioned in [Zobel and Moffat 1995] can easily be extended to our case. As we show later in the experimental results section no significant degradation of the compression ratio is experienced by using bytes instead of bits. On the other hand, decompression of byte Huffman code is faster than decompression of binary Huffman code. In practice, byte processing is much faster than bit processing because bit shifts and masking operations are not necessary at decoding time or at searching time.

3.2 Compression Ratio

In this section we consider the compression ratios achieved with this scheme. A first concern is that Huffman coding needs to store, together with the compressed file, a table with all the text symbols. As we use word compression, this table is precisely the vocabulary of the text, that is, the set of all different text words. This table can in principle be very large and ruin the overall compression ratio.

However, this is not the case on large texts. Heaps' Law [Heaps 1978], an empirical law widely accepted in information retrieval, establishes that a natural language text of $O(u)$ words has a vocabulary of size $v = O(u^\beta)$, for $0 < \beta < 1$. Typically, β is between 0.4 and 0.6 [Araújo et al. 1997; Moura et al. 1997], and therefore v is close to $O(\sqrt{u})$.

Hence, for large texts the overhead of storing the vocabulary is minimal. On the other hand, storing the vocabulary represents an important overhead when the text is small. This is why we chose to compress the vocabulary (that is, the symbol table) using classical binary Huffman on characters. As shown in Figure 2, this fact makes our compressor better than *Gzip* for files of at least 1 megabyte instead

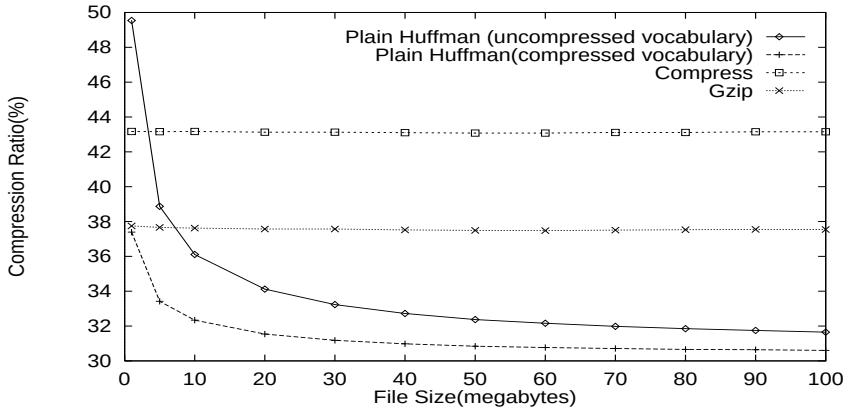


Fig. 2. Compression ratios for the WSJ file compressed by *Gzip*, *Compress*, and plain Huffman with and without compressing the vocabulary.

of 10 megabytes[1]. The need to decompress the vocabulary at search time poses a minimal processing overhead which can even be completely compensated by the reduced I/O.

A second concern is whether the compression ratio can or cannot worsen as the text grows. Since in our model the number of symbols v grows (albeit sublinearly) as the text grows, it could be possible that the average length to code a symbol grows too. The key to prove that this does not happen is to show that the distribution of words in the text is biased enough for the entropy[2] to be $O(1)$, and then to show that Huffman codes put only a constant overhead over this entropy. This final step will be done for d-ary Huffman codes, which includes our 7-bit (tagged) and 8-bit (plain) cases.

We use the Zipf's Law [Zipf 1949] as our model of the frequency of the words appearing in natural language texts. This law, widely accepted in information retrieval, states that if we order the v words of a natural language text in decreasing order of probability, then the probability of the first word is i^θ times the probability of the i-th word, for every i. This means that the probability of the i-th word is $p_i = 1/(i^\theta H)$, where $H = H_v^{(\theta)} = \sum_{j=1}^{v} 1/j^\theta$. The constant θ depends on the text.

Zipf's Law comes in two flavors. A simplified form assumes that $\theta = 1$. In this case, $H = O(\log v)$. Although this simplified form is popular because it is simpler to handle mathematically, it does not follow well the real distribution of natural language texts. There is strong evidence that most real texts have in fact a more biased vocabulary. We performed in [Araújo et al. 1997] a thorough set of experiments on the TREC collection, finding out that the θ values are roughly between 1.5 and 2.0 depending on the text, which gives experimental evidence in favor of the "generalized Zipf's Law" (i.e. $\theta > 1$). Under this assumption,

[1] The reason why both Ziv-Lempel compressors do not improve for larger texts is in part because they search for repetitions only in a relatively short window of the text already seen. Hence, they are prevented from exploiting most of the already processed part of the text.

[2] We estimate the zero-order word-based binary entropy of a text as $-\sum_{i=1..v} p_i \log_2 p_i$, where p_i is the relative frequency of the i-th vocabulary word. For simplicity we call this measure just "entropy" in this paper.

$H = O(1)$.

We have tested the distribution of the separators as well, finding that they also follow reasonably well a Zipf's distribution. Moreover, their distribution is even more biased than that of words, being θ closer to 1.9. We therefore assume that $\theta > 1$ and consider only words, since an analogous proof will hold for separators.

On the other hand, more refined versions of Zipf's Law exist, such as the Mandelbrot distribution [Gonnet and Baeza-Yates 1991]. This law tries to improve the fit of Zipf's Law for the most frequent values. However, it is mathematically harder to handle and it does not alter the asymptotic results that follow.

We analyze the entropy $E(d)$ of such distribution for a vocabulary of v words when d digits are used in the coding alphabet, as follows:

$$E(d) = \sum_{i=1}^{v} p_i \log_d \frac{1}{p_i} = \frac{1}{\ln d} \sum_{i=1}^{v} \frac{\ln H + \theta \ln i}{i^\theta H}$$

$$= \frac{1}{H \ln d} \left(\ln H \sum_{i=1}^{v} \frac{1}{i^\theta} + \theta \sum_{i=1}^{v} \frac{\ln i}{i^\theta} \right) = \log_d H + \frac{\theta}{H \ln d} \sum_{i=1}^{v} \frac{\ln i}{i^\theta}$$

Bounding the summation with an integral, we have that

$$\sum_{i=1}^{v} \frac{\ln i}{i^\theta} \leq \frac{\ln 2}{2^\theta} + \frac{(\theta - 1)\ln 2 + 1}{2^{\theta-1}(\theta-1)^2} + O(\log v / v^{\theta-1}) = O(1)$$

which allows us to conclude that $E(d) = O(1)$, as $\log_d H$ is also $O(1)$.

If we used the simple Zipf's Law instead, the result would be that $E(d) = O(\log v)$, i.e., the average codeword length would grow as the text grows. The fact that this does not happen for 1 gigabyte of text is an independent experimental confirmation of the validity of the generalized Zipf's Law against its simple version.

We consider the overhead of Huffman coding over the entropy. Huffman coding is not optimal because of its inability to represent fractional parts of bits. That is, if a symbol has probability p_i, it should use exactly $\log_2(1/p_i)$ bits to represent the symbol, which is not possible if p_i is not a power of $1/2$. This effect gets worse if instead of bits we use numbers in base d. We give now an upper bound on the compression inefficiency involved.

In the worst case, Huffman will encode each symbol with probability p_i using $\lceil \log_d(1/p_i) \rceil$ digits. This is a worst case because some symbols are encoded using $\lfloor \log_d(1/p_i) \rfloor$ digits. Therefore, in the worst case the average length of a codeword in the compressed text is

$$\sum_{i=1}^{v} p_i \lceil \log_d(1/p_i) \rceil \leq 1 + \sum_{i=1}^{v} p_i \log_d(1/p_i)$$

which shows that, regardless of the probability distribution, we cannot spend more than one extra digit per codeword due to rounding overheads. For instance, if we use bytes we spend at most one more byte per word.

This proves that the compression ratio will not degrade as the text grows, even when the number of different words and separators increases.

Table 2 shows the entropy and compression ratios achieved for binary Huffman, plain Huffman, tagged Huffman, Gnu *Gzip* and Unix *Compress* for the files of the TREC collection. As can be seen, the compression ratio degrades only slightly by using bytes instead of bits and, in that case, we are still below *Gzip*. The exception is the FR collection, which includes a large part of non-natural language such as chemical formulas. The compression ratio of the tagged Huffman code is approximately 3 points (i.e. 3% of u) over that of plain Huffman, which comes from the extra space allocated for the tag bit in each byte.

Method	Files				
	AP	WSJ	DOE	ZIFF	FR
Entropy	26.20	26.00	24.60	27.50	25.30
Binary Huffman	27.41	27.13	26.25	28.93	26.88
Plain Huffman	31.16	30.60	30.19	32.90	30.14
Tagged Huffman	34.12	33.70	32.74	36.08	33.53
Gzip	38.56	37.53	34.94	34.12	27.75
Compress	43.80	42.94	41.08	41.56	38.54

Table 2. Compression ratios achieved by different compression schemes, where "entropy" refers to optimal coding. The space used to store the vocabulary is included in the Huffman compression ratios.

3.3 Compression and Decompression Performance

Finally, we consider in this section the time taken to compress and decompress the text.

To compress the text, a first pass is performed in order to collect the vocabulary and its frequencies. By storing it in a trie data structure, $O(u)$ total worst case time can be achieved. Since a trie requires non practical amounts of memory, we use a hash table to perform this step in our implementation. The average time to collect the vocabulary using a hash table is $O(u)$. The vocabulary is then sorted by the word frequencies at $O(v \log v)$ cost, which in our case is $O(u^\beta \log u) = o(u)$.

After the sorting, we generate a canonical Huffman code of the vocabulary words. The advantage of using canonical trees is that they are space economic. A canonical tree can be represented by using only two small tables with size $O(\log v)$. Further, previous work has shown that decoding using canonical codes reduces decompression times [Hirschberg and Lelewer 1990; Zobel and Moffat 1995; Turpin and Moffat 1997]. The canonical code construction can be done at $O(v)$ cost, without using any extra space by using the algorithm described in [Moffat and Katajainen 1995]. Finally, the file is compressed by generating the codeword of each text word, which is again $O(u)$.

Decompression starts by reading the vocabulary into memory at $O(v)$ cost, as well as the canonical Huffman tree at $O(\log v)$ cost. Then each word in the compressed text is decoded and its output written on disk, for a total time of $O(u)$.

Table 3 shows the compression and decompression times achieved for binary Huffman, plain Huffman, tagged Huffman, *Compress* and *Gzip* for files of the TREC collection. In compression, we are 2-3 times faster than *Gzip* and only 17% slower

than *Compress* (which achieves much worse compression ratios). In decompression, there is a significant improvement when using bytes instead of bits. This is because no bit shifts nor masking are necessary. Using bytes, we are more than 20% faster than *Gzip* and three times faster than *Compress*.

Method	Compression					Decompression				
	AP	WSJ	DOE	ZIFF	FR	AP	WSJ	DOE	ZIFF	FR
Binary Huff.	490	526	360	518	440	170	185	121	174	151
Plain Huff.	487	520	356	515	435	106	117	81	112	96
Tagged Huff.	491	534	364	527	446	112	121	85	116	99
Compress	422	456	308	417	375	367	407	273	373	331
Gzip	1333	1526	970	1339	1048	147	161	105	139	111

Table 3. Compression and decompression times (in elapsed seconds for the whole collections) achieved by different compression schemes.

The main disadvantage of word-based Huffman methods are the space requirements to both compress and decompress the text. At compression time they need the vocabulary and a look up table with the codewords that is used to speed up the compression. The Huffman tree is constructed without any extra space by using an in-place algorithm [Moffat and Katajainen 1995; Milidiu et al. 1998]. At decompression time we need to store the vocabulary in main memory. Therefore the space complexities of our methods are $O(u^{\beta})$. The methods used by *Gzip* and *Compress* have constant space complexity and the amount of memory used can be configured. So, our methods are more memory-demanding than *Compress* and *Gzip*, which constitutes a drawback for some applications. For example, our methods need 4.7 megabytes of memory to compress and 3.7 megabytes of memory to decompress the WSJ file, while *Gzip* and *Compress* need only about 1 megabyte to either compress or decompress this same file. However, for the text searching systems we are interested in, the advantages of our methods (i.e. allowing efficient exact and approximate searching on the compressed text and fast decompression of fragments) are more important than the space requirements.

4. SEARCHING ON TAGGED HUFFMAN COMPRESSED TEXT

Our first searching scheme works on tagged Huffman compressed texts. We recall that the tagged Huffman compression uses one bit of each byte in the compressed text to mark the beginning of each codeword.

General Huffman codes are prefix free codes, which means that no codeword is a prefix of another codeword. This feature is sufficient to decode the compressed text, but it is not sufficient to allow direct searching for compressed words, due to the possibility of false matches. To see this problem, consider the word `"ghost"` in the example presented in Figure 3. Although the word is not present on the compressed text, its codeword is.

The false matches are avoided if in the compressed text no codeword prefix is a suffix of another codeword. We add this feature to the tagged Huffman coding scheme by setting to 1 the highest bit of the first byte of each codeword (this bit is

Fig. 3. An example where the codeword of a word is present in the compressed text but the word is not present in the original text. Codewords are shown in decimal notation.

the "tag"). Since a compressed pattern can now only match its first byte against the first byte of a codeword in the text, we know that any possible match is correctly aligned. This permits the use of any conventional text searching algorithm directly on the compressed text, provided we search for whole words.

In general we are able to search phrase patterns. A *phrase pattern* is a sequence of *elements*, where each element is either a simple word or an extended pattern. Extended patterns, which are to be matched against a single text word, include the ability to have any set of characters at each position, unbounded number of wild cards, arbitrary regular expressions, approximate searching, and combinations. The Appendix gives a detailed description of the patterns supported by our system.

The search for a pattern on a compressed text is made in two phases. In the first phase we compress the pattern using the same structures used to compress the text. In the second phase we search for the compressed pattern. In an exact pattern search, the first phase generates a unique pattern that can be searched with any conventional searching algorithm. In an approximate or extended pattern search, the first phase generates all the possibilities of compressed codewords that match with the original pattern in the vocabulary of the compressed text. In this last case we use a multi-pattern algorithm to search the text. We now explain this method in more detail and show how to extend it for phrases.

4.1 Preprocessing Phase

Compressing the pattern when we are performing an exact search is similar to the coding phase of the Huffman compression. We search for each element of the pattern in the Huffman vocabulary and generate the compressed codeword for it. If there is an element in the pattern that is not in the vocabulary then there are no occurrences of the pattern in the text.

If we are doing approximate or extended search then we need to generate compressed codewords for all symbols in the Huffman vocabulary that match with the element in the pattern. For each element in the pattern we make a list of the compressed codewords of the vocabulary symbols that match with it. This is done by sequentially traversing the vocabulary and collecting all the words that match the pattern. This technique has been already used in block addressing indices on uncompressed texts [Manber and Wu 1993; Araújo et al. 1997; Baeza-Yates and Navarro 1997]. Since the vocabulary is very small compared to the text size, the sequential search time on the vocabulary is negligible, and there is no other additional cost to allow complex queries. This is very difficult to achieve with online plain text searching, since we take advantage of the knowledge of the vocabulary stored as part of the Huffman tree.

Depending on the pattern complexity we use two different algorithms to search the vocabulary. For phrase patterns allowing k errors ($k \geq 0$) that contain sets of characters at any position we use the algorithm presented in [Baeza-Yates and Navarro 1999]. If v is the size of the vocabulary and w is the length of a word W the algorithm runs in $O(v + w)$ time to search W. For more complicated patterns allowing k errors ($k \geq 0$) that contain unions, wild cards or regular expressions we use the algorithm presented in [Wu and Manber 1992], which runs in $O(kv + w)$ time to search W. A simple word is searched in $O(w)$ time using, e.g., a hash table.

4.2 Searching Phase

For exact search, after obtaining the compressed codeword (a sequence of bytes) we can choose any known algorithm to process the search. In the experimental results presented in this paper we used the Sunday [Sunday 1990] algorithm, from the Boyer-Moore family, which has good practical performance. In the case of approximate or extended searching we convert the problem to the exact multipattern searching problem. We just obtain a set of codewords that match the pattern and use a multipattern search algorithm proposed by Baeza-Yates and Navarro [Baeza-Yates and Navarro 1999]. This algorithm is an extension of the Sunday algorithm, and works well when the number of patterns to search is not very large. In case of a large number of patterns to search, the best option would be Aho-Corasick [Aho and Corasick 1975], which allows to search in $O(n)$ time independently of the number of patterns.

If we assume that the compressed codeword of a pattern of length m is c, then Boyer-Moore type algorithms inspect about n/c bytes of the compressed text in the best case. This best case is very close to the average case because the alphabet is large (of size 128 or 256) and uniformly distributed, as compared to the small pattern length c (typically 3 or 4). On the other hand, the best case in uncompressed text searching is to inspect u/m characters. Since the compression ratio n/u should roughly hold for the pattern on average, we have that $n/u \approx c/m$ and therefore the number of inspected bytes in compressed and uncompressed text is roughly the same.

There are, however, three reasons that make compressed search faster. First, the number of bytes read from disk is n, which is smaller than u. Second, in compressed search the best case is very close to the average case, while this is not true when searching uncompressed text. Third, the argument that says that c/m is close to n/u assumes that the search pattern is taken randomly from the text, while in practice a model of selecting it randomly from the vocabulary matches reality much better. This model yields a larger c value on average, which improves the search time on compressed text.

Searching a phrase pattern is more complicated. A simple case arises when the phrase is a sequence of simple words that is to be found as is (even with the same separators). In this case we can concatenate the codewords of all the words and separators of the phrase and search for the resulting (single) pattern.

If, on the other hand, we want to disregard the exact separators between phrase elements or they are not simple words, we apply a different technique. In the general case, the original pattern is represented by the sequence of lists $L_1, .., L_j$, where L_i has the compressed codewords that match the i-th element of the original

pattern. To start the search in the compressed text we choose one of these lists and use the algorithm for one-word patterns to find the occurrences in the text. When an occurrence of one element of the first list searched is found, we use the other lists to verify if there is an occurrence of the entire pattern at this text position.

The choice of the first list searched is fundamental for the performance of the algorithm. We heuristically choose the element i of the phrase that maximizes the minimal length of the codewords in L_i. This choice comes directly from the cost to search a list of patterns. Longer codewords have less probability of occurrence in the text, which translates into less verifications for occurrences of elements of the other lists. Moreover, most text searching algorithms work faster on longer patterns. This type of heuristic is also of common use in inverted files when solving conjunctive queries [Baeza-Yates and Ribeiro-Neto 1999; Witten et al. 1999].

A particularly bad case for this filter arises when searching a long phrase formed by very common words, such as `"to be or not to be"`. The problem gets worse if errors are allowed in the matches or we search for even less stringent patterns. A general and uniform cost solution to all these types of searches is depicted in the next section.

5. SEARCHING ON PLAIN HUFFMAN COMPRESSED TEXT

A disadvantage of our first searching scheme described before is the loss in compression due to the extra bit used to allow direct searching. A second disadvantage is that the filter may not be effective for some types of queries. We show now how to search in the plain Huffman compressed text, a code that has no special marks and gives a better compression ratio than the tagged Huffman scheme. We also show that much more flexible searching can be carried out in an elegant and uniform way.

We present two distinct searching algorithms. The first one, called *plain filterless*, is an automaton-based algorithm that elegantly handles all possible complex cases that may arise, albeit slower than the previous scheme. The second, called *plain filter*, is a combination of both algorithms, trying to do direct pattern matching on plain Huffman compressed text and using the automaton-based algorithm as a verification engine for false matches.

5.1 The Automaton-Based Algorithm

As in the previous scheme, we make heavy use of the vocabulary of the text, which is available as part of the Huffman coding data. The Huffman tree can be regarded as a trie where the leaves are the words of the vocabulary and the path from the root to a leaf spells out its compressed codeword, as shown in the left part of Figure 4 for the word `"rose"`.

We first explain how to solve exact words and phrases and then extend the idea for extended and approximate searching. The pattern preprocessing consists on searching it in the vocabulary as before and *marking* the corresponding entry. In general, however, the patterns are phrases. To preprocess phrase patterns we simply perform this procedure for each word of the pattern. For each word of the vocabulary we set up a bit mask that indicates which elements of the pattern does the word match. Figure 4 shows the marks for the phrase pattern `"rose is"`, where `01` indicates that the word `"is"` matches the second element in the pattern and `10`

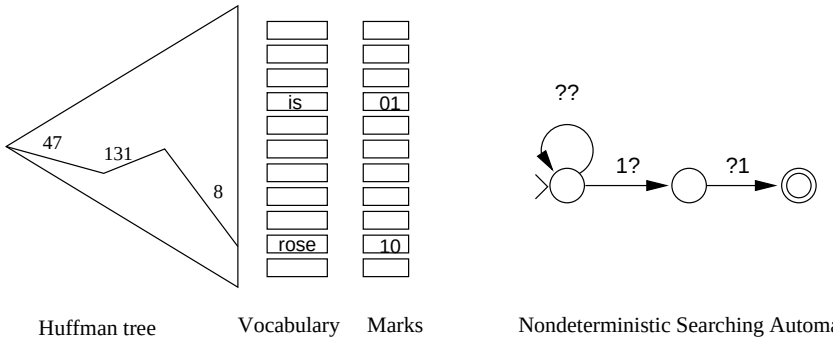


Fig. 4. The searching scheme for the pattern `"rose is"`. In this example the word `"rose"` has a three-byte codeword `47 131 8`. In the nondeterministic finite automaton, '`?`' stands for 0 and 1.

indicates that the word `"rose"` matches the first element in the pattern (all the other words have `00` since they match nowhere). If any word of the pattern is not found in the vocabulary we immediately know that the pattern is not in the text.

Next, we scan the compressed text, byte by byte, and at the same time traverse the Huffman tree downwards, as if we were decompressing the text[3]. A new symbol occurs whenever we reach a leaf of the Huffman tree. At each word symbol obtained we send the corresponding bit mask to a nondeterministic automaton, as illustrated in Figure 4. This automaton allows moving from state i to state $i+1$ whenever the i-th word of the pattern is recognized. Notice that this automaton depends only on the number of words in the phrase query. After reaching a leaf we return to the root of the tree and proceed in the compressed text.

The automaton is simulated with the Shift-Or algorithm [Baeza-Yates and Gonnet 1992]. We perform one transition in the automaton for each text word. The Shift-Or algorithm simulates efficiently the nondeterministic automaton using only two operations per transition. In a 32-bit architecture it can search a phrase of up to 32 elements using a single computer word as the bit mask. For longer phrases we use as many computer words as needed.

For complex patterns the preprocessing phase corresponds to a sequential search in the vocabulary to mark all the words that match the pattern. To search the symbols in the vocabulary we use the same algorithms described in Section 4.1. The corresponding mask bits of each matched word in the vocabulary are set to indicate its position in the pattern. Figure 5 illustrates this phase for the pattern `"ro# rose is"` with $k = 1$ (i.e. allowing 1 error per word, where `"ro#"` means any word starting with `"ro"`). For instance, the word `"rose"` in the vocabulary matches the pattern at positions 1 and 2. The compressed text scanning phase does not change.

The cost of the preprocessing phase is as in Section 4.1. The only difference is that we mark bit masks instead of collecting matching words. The search phase takes $O(n)$ time.

Finally, we show how to deal with separators and stopwords. Most online search-

[3]However, this is much faster than decompression because we do not generate the uncompressed text.

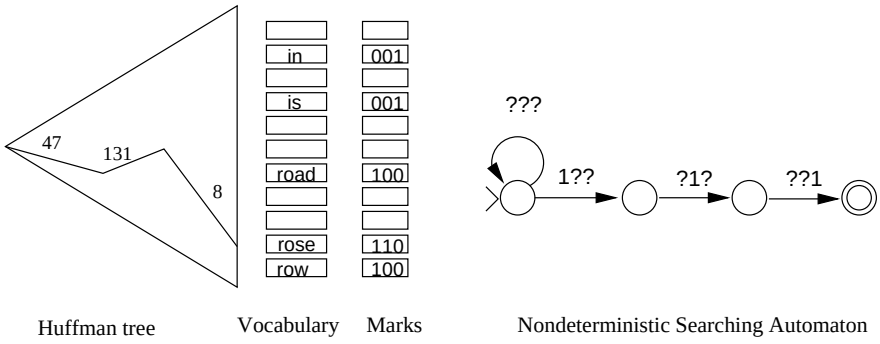


Fig. 5. General searching scheme for the phrase `"ro# rose is"` allowing 1 error. In the nondeterministic finite automaton, '`?`' stands for 0 and 1.

ing algorithms cannot efficiently deal with the problem of matching a phrase disregarding the separators among words (e.g. two spaces between words instead of one). The same happens with the stopwords, which usually can be disregarded when searching indexed text but are difficult to disregard in online searching. In our compression scheme we know which elements of the vocabulary correspond in fact to separators, and the user can define (at compression or even at search time) which correspond to stopwords. We can therefore have marked the leaves of the Huffman tree corresponding to separators and stopwords, so that the searching algorithm can ignore them by not producing a symbol when arriving at such leaves. Therefore, we disregard separators and stopwords from the sequence and from the search pattern at negligible cost. Of course they cannot be just removed from the sequence at compression time if we want to be able to recover the original text.

5.2 A Filtering Algorithm

We show in this section how the search on the plain Huffman compressed text is improved upon the automaton-based algorithm described in the previous section. The central idea is to search the compressed pattern directly in the text, as was done with the tagged Huffman code scheme presented in Section 4.

Every time a match is found in the compressed text we must verify whether this match indeed corresponds to a word. This is mandatory due to the possibility of false matches, as illustrated in Figure 3 of Section 4. The verification process consists of applying the automaton-based algorithm to the region where the possible match was found. To avoid processing the text from the very beginning to make this verification we divide the text in small *blocks* of the same size at compression time. The codewords are aligned to the beginning of blocks, so that no codeword crosses a block boundary. Therefore, we only need to run the basic algorithm from the beginning of the block that contains the match.

The block size must be small enough so that the slower basic algorithm is used only on small areas, and large enough so that the extra space lost at block boundaries is not significant. We ran a number of experiments on the WSJ file, arriving to 256-byte blocks as a good time-space tradeoff.

The extension of the algorithm for complex queries and phrases follows the same idea: search as in Section 4 and then use the automaton-based algorithm to check

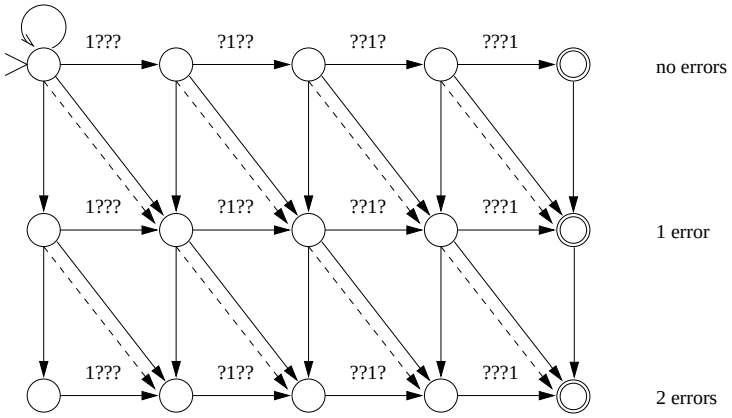


Fig. 6. A nondeterministic automaton for approximate phrase searching (4 words, 2 errors) in the compressed text. Dashed transitions flow without consuming any text input. The other vertical and diagonal (unlabeled) transitions accept any bit mask. The '?' stands for 0 and 1.

the matches. In this case, however, we use multipattern searching, and the performance may be degraded not only for the same reasons as in Section 4, but also because of the possibility of verifying too many text blocks. If the number of matching words in the vocabulary is too large, the efficiency of the filter may be degraded, and the use of the scheme with no filter might be preferable.

5.3 Even More Flexible Pattern Matching

The Shift-Or algorithm can do much more than just searching for a simple sequence of elements. For instance, it has been enhanced to search for regular expressions, to allow errors in the matches and other flexible patterns [Wu and Manber 1992; Baeza-Yates and Navarro 1999]. This powerful type of search is the basis of the software Agrep [Wu and Manber 1992].

A new handful of choices appear when we use these abilities in our word-based compressed text scenario. Consider the automaton of Figure 6. It can search in the compressed text for a phrase of four words allowing up to two insertions, deletions or replacements of *words*. Apart from the well known horizontal transitions that match words, there are vertical transitions that insert new words in the pattern, diagonal transitions that replace words, and dashed diagonal transitions that delete words from the pattern.

This automaton can be efficiently simulated using extensions of the Shift-Or algorithm to search in the compressed text for *approximate* occurrences of the phrase. For instance, the search of `"identifying potentially relevant matches"` could find the occurrence of `"identifying a number of relevant matches"` in the text with one replacement error, assuming that the stop words `"a"` and `"of"` are disregarded as explained before. Moreover, if we allow three errors at the character level as well we could find the occurrence of `"who identified a number of relevant matches"` in the text, since for the algorithm there is an occurrence of `"identifying"` in `"identified"`. Other efficiently implementable setups can be insensitive to the order of the words in the phrase. The same phrase query could be

found in `"matches considered potentially relevant were identified"` with one deletion error for `"considered"`. Finally, *proximity searching* is of interest in IR and can be efficiently solved. The goal is to give a phrase and find its words relatively close to each other in the text. This would permit to find out the occurrence of `"identifying and tagging potentially relevant matches"` in the text.

Approximate searching has traditionally operated at the character level, where it aims at recovering the correct *syntax* from typing or spelling mistakes, errors coming from optical character recognition software, misspelling of foreign names, and so on. Approximate searching at the word level, on the other hand, aims at recovering the correct *semantics* from concepts that are written with a different wording. This is quite usual in most languages and is a common factor that prevents finding the relevant documents.

This kind of search is very difficult for a sequential algorithm. Some indexed schemes permit proximity searching by operating on the list of exact word positions, but this is all. In the scheme described above, this is simple to program, elegant and extremely efficient (more than on characters). This is an exclusive feature of this compression method that opens new possibilities aimed at recovering the intended semantics, rather than the syntax, of the query. Such capability may improve the retrieval effectiveness of IR systems.

6. SEARCHING PERFORMANCE

The performance evaluation of the three algorithms presented in previous sections was obtained by considering 40 randomly chosen patterns containing 1 word, 40 containing 2 words, and 40 containing 3 words. The same patterns were used by the three search algorithms. All experiments were run on the WSJ text file and the results were obtained with a 99% confidence interval. The size of the uncompressed WSJ is 262.8 megabytes, while its compressed versions are 80.4 megabytes with the plain Huffman method and 88.6 megabytes with tagged Huffman.

Table 4 presents exact ($k = 0$) and approximate ($k = 1, 2, 3$) searching times using *Agrep* [Wu and Manber 1992], *tagged* (direct search on tagged Huffman), *plain filterless* (the basic algorithm on plain Huffman), and *plain filter* (the filter on plain Huffman, with Sunday filtering for blocks of 256 bytes). It can be seen from this table that our three algorithms are almost insensitive to the number of errors allowed in the pattern while *Agrep* is not. The *plain filterless* algorithm is really insensitive because it maps all the queries to the same automaton that does not depend on k. The filters start taking about 2/3 of the filterless version, and become closer to it as k grows. The experiments also shows that both *tagged* and *plain filter* are faster than *Agrep*, almost twice as fast for exact searching and nearly 8 times faster for approximate searching. For all times presented, there is a constant I/O time factor of approximately 8 seconds for our algorithms to read the WSJ compressed file and approximately 20 seconds for *Agrep* to read the WSJ uncompressed file. These times are already included on all tables.

The following test was for more complex patterns. This time we experimented with *specific* patterns instead of selecting a number of them at random. The reason is that there is no established model for what is a "random" complex pattern. Instead, we focused on showing the effect of different pattern features, as follows:

Algorithm	$k = 0$	$k = 1$	$k = 2$	$k = 3$
Agrep	23.8 ± 0.38	117.9 ± 0.14	146.1 ± 0.13	174.6 ± 0.16
tagged	14.1 ± 0.18	15.0 ± 0.33	17.0 ± 0.71	22.7 ± 2.23
plain filterless	22.1 ± 0.09	23.1 ± 0.14	24.7 ± 0.21	25.0 ± 0.49
plain filter	15.1 ± 0.30	16.2 ± 0.52	19.4 ± 1.21	23.4 ± 1.79

Table 4. Searching times (in elapsed seconds) for the WSJ text file using different search techniques and different number of errors k. Simple random patterns were searched.

(1) `prob#` (where `#` means any character considered zero or more times, one possible answer being `"problematic"`): an example of pattern that matches with lot of words on the vocabulary;

(2) `local television stations`, a phrase pattern composed of common words;

(3) `hydraulic forging`, a phrase pattern composed of uncommon words;

(4) `Bra[sz]il# and Ecua#`, a phrase pattern composed of a complex expression.

Table 4 presents exact ($k = 0$) and approximate ($k = 1$ and $k = 2$) searching times for the patterns presented above.

Algorithm	Pattern 1			Pattern 2		
	$k = 0$	$k = 1$	$k = 2$	$k = 0$	$k = 1$	$k = 2$
Agrep	74.3	117.7	146.0	23.0	117.6	145.1
tagged	18.4	20.6	21.1	16.5	19.0	26.0
plain filterless	22.8	23.5	23.6	21.1	23.3	25.5
plain filter	21.4	21.4	22.1	15.2	17.1	22.3
Algorithm	Pattern 3			Pattern 4		
	$k = 0$	$k = 1$	$k = 2$	$k = 0$	$k = 1$	$k = 2$
Agrep	21.9	117.1	145.1	74.3	117.6	145.8
tagged	14.5	15.0	16.0	18.2	18.3	18.7
plain filterless	21.7	21.5	21.6	24.2	24.2	24.6
plain filter	15.0	15.7	16.5	17.6	17.6	18.0

Table 5. Searching times (in elapsed seconds) for the WSJ text file using different search techniques and different number of errors k.

Note that, in any case, the results on complex patterns do not differ much from those for simple patterns. *Agrep*, on the other hand, takes much more time on complex patterns such as pattern (1) and pattern (4).

7. CONCLUSIONS AND FUTURE WORK

In this paper we investigated a fast compression and decompression scheme for natural language texts and also presented algorithms which allow efficient search for exact and extended word and phrase patterns. We showed that we achieve about 30% compression ratio, against 40% and 35% for *Compress* and *Gzip*, respectively.

For typical texts, compression times are close to the times of *Compress* and approximately half the times of *Gzip*, and decompression times are lower than those of *Gzip* and one third of those of *Compress*.

Search times are *better* on the compressed text than on the original text (about twice as fast). Moreover, a lot of flexibility is provided in the search patterns. Complex patterns are searched much faster than on uncompressed text (8 times faster is typical) by making heavy use of the vocabulary information kept by the compressor.

The algorithms presented in this paper have been implemented in a software system called *Cgrep*, which is publicly available. An example of the power of *Cgrep* is the search of a pattern containing 3 words and allowing 1 error, in a compressed file of approximately 80.4 megabytes (corresponding to the WSJ file of 262.8 megabytes). *Cgrep* runs at 5.4 megabytes per second, which is equivalent to searching the original text at 17.5 megabytes per second. As *Agrep* searches the original text at 2.25 megabytes per second, *Cgrep* is 7.8 times faster than *Agrep*.

These results are so good that they encourage keeping the text compressed *all the time*. That is, all the textual documents of a user or a database can be kept permanently compressed as a single text collection. Searching of interesting documents can be done without decompressing the collection, and fast decompression of relevant files for presentation purposes can be done efficiently. To complete this picture and convert it into a viable alternative, a mechanism to *update* a compressed text collection must be provided, so documents can be added, removed and altered efficiently. Some techniques have been studied in [Moura 1999], where it is shown that efficient updating of compressed text is possible and viable.

Finally, we remark that sequential searching is not a viable solution when the text collections are very large, in which case indexed schemes have to be considered. Our technique is not only useful to speed up sequential search. In fact, it can be used with any indexed scheme. Retrieved text is usually scanned to find the byte position of indexed terms and our algorithms will be of value for this task [Witten et al. 1999]. In particular, it can also be used to improve indexed schemes that combine inverted files and sequential search, like *Glimpse* [Manber and Wu 1993]. *Glimpse* divides the text space into logical blocks and builds an inverted file where each list of word occurrences points to the corresponding blocks. Searching is done by first searching in the vocabulary of the inverted file and then sequentially searching in all the selected blocks. By using blocks, indices of only 2%-4% of space overhead can significantly speed up the search. We have combined our compression scheme with block addressing inverted files, obtaining much better results than those that work on uncompressed text [Navarro et al. 2000].

ACKNOWLEDGMENTS

We wish to acknowledge the many fruitful discussions with Marcio D. Araújo, who helped particularly with the algorithms for approximate searching in the text vocabulary. We also thank the many comments of the referees that helped us to improve this work.

APPENDIX

A. COMPLEX PATTERNS

We present the types of phrase patterns supported by our system. For each word of a pattern it allows to have not only single letters in the pattern, but any set of letters or digits (called just "characters" here) at each position, exactly or allowing errors, as follows:

—range of characters (e.g. `t[a-z]xt`, where `[a-z]` means any letter between `a` and `z`);

—arbitrary sets of characters (e.g. `t[aei]xt` meaning the words `taxt`, `text` and `tixt`);

—complements (e.g. `t[~ab]xt`, where `~ab` means any single character except `a` or `b`; `t[~a-d]xt`, where `~a-d` means any single character except `a`, `b`, `c` or `d`);

—arbitrary characters (e.g. `t·xt` means any character as the second character of the word);

—case insensitive patterns (e.g. `Text` and `text` are considered as the same words).

In addition to single strings of arbitrary size and classes of characters described above the system supports patterns combining exact matching of some of their parts and approximate matching of other parts, unbounded number of wild cards, arbitrary regular expressions, and combinations, exactly or allowing errors, as follows:

—unions (e.g. `t(e|ai)xt` means the words `text` and `taixt`; `t(e|ai)*xt` means the words beginning with `t` followed by `e` or `ai` zero or more times followed by `xt`). In this case the word is seen as a regular expression;

—arbitrary number of repetitions (e.g. `t(ab)*xt` means that `ab` will be considered zero or more times). In this case the word is seen as a regular expression;

—arbitrary number of characters in the middle of the pattern (e.g. `t#xt`, where `#` means any character considered zero or more times). In this case the word is not considered as a regular expression for efficiency. Note that `#` is equivalent to `·*` (e.g. `t#xt` and `t·*xt` obtain the same matchings but the latter is considered as a regular expression);

—combining exact matching of some of their parts and approximate matching of other parts (`<te>xt`, with $k = 1$, meaning exact occurrence of `te` followed by any occurrence of `xt` with 1 error);

—matching with nonuniform costs (e.g. the cost of insertions can be defined to be twice the cost of deletions).

We emphasize that the system performs whole-word matching only. That is, the pattern is a sequence of words or complex expressions that are to be matched against whole text words. It is not possible to write a single regular expression that returns a phrase. Also, the extension described in Section 5.3 is not yet implemented.

REFERENCES

Aho, A. and Corasick, M. 1975. Efficient string matching: an aid to bibliographic search. *Communications of the ACM 18*, 6, 333–340.

Amir, A. and Benson, G. 1992. Efficient two-dimensional compressed matching. In *Proc. Second IEEE Data Compression Conference* (March 1992), pp. 279–288.

Amir, A., Benson, G., and Farach, M. 1996. Let sleeping files lie: pattern matching in z-compressed files. *Journal of Computer and Systems Sciences 52*, 2, 299–307.

Araújo, M. D., Navarro, G., and Ziviani, N. 1997. Large text searching allowing errors. In R. Baeza-Yates Ed., *Proc. of the Fourth South American Workshop on String Processing*, Volume 8 (1997), pp. 2–20. Carleton University Press International Informatics Series.

Baeza-Yates, R. and Gonnet, G. 1992. A new approach to text searching. *Communications of the ACM 35*, 10, 74–82.

Baeza-Yates, R. and Navarro, G. 1997. Block addressing indices for approximate text retrieval. In *Proc. of Sixth ACM International Conference on Information and Knowledge Management (CIKM'97)* (1997), pp. 1–8. Extended version to appear in *JASIS*.

Baeza-Yates, R. and Navarro, G. 1999. Faster approximate string matching. *Algorithmica 23*, 2, 127–158.

Baeza-Yates, R. and Ribeiro-Neto, B. 1999. *Modern Information Retrieval*. Addison-Wesley.

Bell, T. C., Cleary, J. G., and Witten, I. H. 1990. *Text compression*. Prentice Hall.

Bell, T. C., Moffat, A., Nevill-Manning, C., Witten, I. H., and Zobel, J. 1993. Data compression in full-text retrieval systems. *Journal of the American Society for Information Science 44*, 508–53.

Bentley, J., Sleator, D., Tarjan, R., and Wei, V. 1986. A locally adaptive data compression scheme. *Communications of the ACM 29*, 320–330.

Farach, M. and Thorup, M. 1995. String matching in lempel-ziv compressed strings. In *Proc. 27th ACM Annual Symposium on the Theory of Computing* (1995), pp. 703–712.

Gonnet, G. H. and Baeza-Yates, R. 1991. *Handbook of Algorithms and Data Structures*. Addison-Wesley.

Harman, D. K. 1995. Overview of the third text retrieval conference. In *Proc. Third Text REtrieval Conference (TREC-3)* (Gaithersburg, Maryland, 1995), pp. 1–19. National Institute of Standards and Technology Special Publication.

Heaps, J. 1978. *Information Retrieval - Computational and Theoretical Aspects*. Academic Press.

Hirschberg, D. S. and Lelewer, D. A. 1990. Efficient decoding of prefix codes. *Communications of the ACM 33*, 4, 449–459.

Horspool, R. N. and Cormack, G. V. 1992. Constructing word-based text compression algorithms. In *Proc. of IEEE Second Data Compression Conference* (1992), pp. 62–81.

Huffman, D. A. 1952. A method for the construction of minimum-redundancy codes. In *Proc. of the Institute of Electrical and Radio Engineers*, Volume 40 (1952), pp. 1090–1101.

Kida, T., Shibata, Y., Takeda, M., Shinohara, A., and Arikawa, S. 1999. A unifying framework for compressed pattern matching. In *Proc. 6th International Symposium on String Processing and Information Retrieval (SPIRE'99)* (1999), pp. 89–96.

Kida, T., Takeda, M., Shinohara, A., Miyazaki, M., and Arikawa, S. 1998. Multiple pattern matching in lzw compressed text. In *Proc. Eighth IEEE Data Compression Conference* (1998).

Kida, T., Takeda, M., Shinohara, A., Miyazaki, M., and Arikawa, S. 1999. Shift-And approach to pattern matching in LZW compressed text. In *Proc. 10th Annual Symposium on Combinatorial Pattern Matching (CPM'99)*, LNCS 1645 (1999), pp. 1–13.

Manber, U. 1997. A text compression scheme that allows fast searching directly in the compressed file. *ACM Transactions on Information Systems 15*, 2, 124–136.

Manber, U. and Wu, S. 1993. Glimpse: a tool to search through entire file systems. Technical Report 93-34 (October), Dept. of Computer Science, Univ. of Arizona.

Milidiu, R., Pessoa, A., and Laber, E. 1998. In-place, simple, and fast length-restricted prefix coding. In *Proc. 5th International Symposium on String Processing and Information Retrieval (SPIRE'98)* (September 1998), pp. 50–59. IEEE Computer Society.

MOFFAT, A. 1989. Word-based text compression. *Software Practice and Experience 19*, 2, 185–198.

MOFFAT, A. AND KATAJAINEN, J. 1995. In-place calculation of minimum-redundancy codes. In S. AKL, F. DEHNE, AND J.-R. SACK Eds., *Proc. Workshop on Algorithms and Data Structures* (Queen's University, Kingston, Ontario, August 1995), pp. 393–402. LNCS 955, Springer-Verlag.

MOURA, E. S. 1999. *Aplicações de Compressão de Dados a Sistemas de Recuperação de Informação*. Ph. D. thesis, Dept. of Computer Science, Univ. Federal de Minas Gerais, Brazil. In Portuguese.

MOURA, E. S., NAVARRO, G., AND ZIVIANI, N. 1997. Indexing compressed text. In R. BAEZA-YATES Ed., *Proc. of the Fourth South American Workshop on String Processing*, Volume 8 (1997), pp. 95–111. Carleton University Press International Informatics Series.

MOURA, E. S., NAVARRO, G., ZIVIANI, N., AND BAEZA-YATES, R. 1998b. Direct pattern matching on compressed text. In *Proc. 5th International Symposium on String Processing and Information Retrieval (SPIRE'98)* (September 1998), pp. 90–95. IEEE Computer Society.

MOURA, E. S., NAVARRO, G., ZIVIANI, N., AND BAEZA-YATES, R. 1998a. Fast searching on compressed text allowing errors. In *Proc. 21st International ACM Conference on Research and Development in Information Retrieval (SIGIR'98)* (August 1998), pp. 298–306.

NAVARRO, G., MOURA, E., NEUBERT, M., ZIVIANI, N., AND BAEZA-YATES, R. 2000. Adding compression to block addressing inverted indices. *Information Retrieval*. To appear.

NAVARRO, G. AND RAFFINOT, M. 1999. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proc. 10th Annual Symposium on Combinatorial Pattern Matching (CPM'99)*, LNCS 1645 (1999), pp. 14–36.

SCHWARTZ, E. S. AND KALLICK, B. 1964. Generating a canonical prefix encoding. *Communications of the ACM 7*, 166–169.

SUNDAY, D. 1990. A very fast substring search algorithm. *Communications of the ACM 33*, 8, 133–142.

TURPIN, A. AND MOFFAT, A. 1997. Fast file search using text compression. In *Proc. 20th Australian Computer Science Conference* (1997), pp. 1–8.

WITTEN, I., MOFFAT, A., AND BELL, T. 1999. *Managing Gigabytes* (second ed.). Morgan Kaufmann Publishers, New York.

WU, S. AND MANBER, U. 1992. Fast text searching allowing errors. *Communications of the ACM 35*, 10, 83–91.

ZIPF, G. 1949. *Human Behaviour and the Principle of Least Effort*. Addison-Wesley.

ZIV, J. AND LEMPEL, A. 1976. On the complexity of finite sequences. *IEEE Transactions on Information Theory 22*, 75–81.

ZIV, J. AND LEMPEL, A. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 3, 337–343.

ZIV, J. AND LEMPEL, A. 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory 24*, 5, 530–536.

ZOBEL, J. AND MOFFAT, A. 1995. Adding compression to a full-text retrieval system. *Software Practice and Experience 25*, 8, 891–903.