Fast Searching on Compressed Text Allowing Errors

Edleno Silva de Moura *
Depto. de Ciência da Computação
Univ. Federal de Minas Gerais, Brazil
www.dcc.ufmg.br/~edleno

Gonzalo Navarro †
Depto. de Ciencias de la Computación
Univ. de Chile, Chile
www.dcc.uchile.cl/~gnavarro

Nivio Ziviani ‡
Depto. de Ciência da Computação
Univ. Federal de Minas Gerais - Brazil
www.dcc.ufmg.br/~nivio

Ricardo Baeza-Yates †
Depto. de Ciencias de la Computación
Univ. de Chile, Chile
www.dcc.uchile.cl/~rbaeza

Abstract We present a fast compression and decompression scheme for natural language texts that allows efficient and flexible string matching by searching the compressed text directly. The compression scheme uses a word-based Huffman encoding and 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 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*.

The searching algorithm allows a large number of variations of the exact and approximate compressed string matching problem, such as phrases, ranges, complements, wild cards and arbitrary regular expressions. Separators and stopwords can be discarded at search time without significantly increasing the cost. The algorithm is based on a word-oriented shift-or algorithm and a fast Boyer-Moore-type filter. It concomitantly uses the vocabulary of the text available as part of the Huffman coding data. When searching for simple patterns, our experiments show that running our algorithm on a compressed text is twice as fast as running *Agrep* on the uncompressed version of the same text. When searching complex or approximate patterns, our algorithm is up to 8 times faster than *Agrep*. We also mention the impact of our technique in inverted files pointing to documents or logical blocks as *Glimpse*.

1 Introduction

In this paper we discuss an efficient compression scheme and also present an algorithm which allows to search for exact and approximate patterns directly in the compressed text. To the best of our knowledge this is the first attempt to do approximate string matching on compressed text, an open problem in [ABF96].

The *compressed matching problem* was first defined in the work of Amir and Benson [AB92] as the task of performing string matching in a compressed text without decompressing it. Given a text T, a corresponding compressed string Z, and a pattern P, 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 $O(u+m)$, where $u = |T|$ and $m = |P|$. An optimal algorithm takes worst-case time $O(n+m)$, where $n = |Z|$. In [ABF96], 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. Most likely an optimal run-time algorithm that takes $O(n)$ additional extra space may not be feasible in practice.

Apart from efficient searching, the compression methods try to minimize the *compression ratio*, which is the size of the compressed file as a percentage of the uncompressed file (i.e. n/u).

In our compression scheme we propose a variant of the word-based Huffman code [BSTW86, Mof89]. In our work the Huffman code assigned to each text word is a sequence of whole bytes and the Huffman tree has degree 256 instead of 2. As we show later, using bytes instead of bits does not significantly degrade the compression ratios. In practice, byte processing is much faster than bit processing because bit shifts and masking operations are not necessary at search time. Further, compression and decompression are very fast and compression ratios achieved are better than those of the Lempel-Ziv family [ZL77, ZL78].

Our searching algorithm is based on a word-oriented shift-or algorithm [BYG92] and uses a fast Boyer-Moore-type filter [Sun90] to speed up the scanning of the compressed text. The vocabulary available as part of the Huffman coding data is used concomitantly at this point. This vocabulary imposes a negligible space overhead when the text collection is large. The algorithm allows a large number of variations of the exact and approximate compressed string matching problem. As a result, phrases, ranges, complements, wild cards, and arbitrary regular expressions can be efficiently searched. Our algorithm can also discard separators and stopwords without

*This work has been partially supported by CAPES scholarship.
†This work has been partially supported by Fondecyt grant 1-950622 and AMYRI Project.
‡This work has been partially supported by CNPq grant 520916/94-8, PRONEX grant 76.97.1016.00 and AMYRI Project.

significantly increasing the search cost.

The *approximate text searching 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.

Let u, n and m be as defined above. For exact searching, our approach finds all pattern occurrences in $O(n + m)$ time (which is optimal) and near $O(\sqrt{u})$ extra space. For approximate searching our algorithms find all pattern occurrences in near $O(n + m\sqrt{u})$ time and near $O(\sqrt{u})$ extra space.

Our technique is not only useful to speed up sequential search. In fact, it can also be used to improve indexed schemes that combine inverted files and sequential search, like *Glimpse* [MW93]. *Glimpse* divides the text space in logical blocks and builds an inverted file where each list of word occurrences points to the corresponding blocks. Searching is done by first doing a search in the inverted file and then a sequential search in all the selected blocks. By using our compression scheme for the whole text, direct search can be done over each block improving the search time by a factor of 8. Notice that in this context, the alphabet size (number of different words) is very large, which is one of our working assumptions.

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 package is available from `ftp://dcc.ufmg.br/pub/research/~nivio/cgrep`, as a prototype in its version 1.0.

This paper is organized as follows. In Section 2 we present 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 approximate compressed string matching, followed by analytical and experimental results. In Section 5 we present some conclusions and future work directions.

2 Related Work

In [FT95] it was presented a compressed matching algorithm for the LZ1 classic compression scheme [ZL76] that runs in time $O(n \log^2(u/n) + m)$. In [ABF96] it was presented a compressed matching algorithm for the LZ78 compression scheme that finds the first occurrence in time $O(n+m^2)$ and space $O(n+m^2)$ or in time $O(n \log m+m)$ and in space $O(n + m)$. Our approach differs from the work in [FT95, ABF96] in the following aspects. First, we use a distinct theoretical framework. Second, while their work includes no implementation of the proposed algorithms, we implement and thoroughly evaluate our algorithms. Third, our empirical evaluation considers both the compression scheme and the compressed matching (exact and approximate) problem.

Another text compression scheme that allows direct searching was proposed by [Man97]. His scheme packs pairs of frequent characters in a single byte, leading to a compression ratio of approximately 70% for typical text files. Like this work we want also to keep the search at byte level for efficiency. However, our approach leads to a better compression ratio of less than half (30% against 70%) the compression ratio in [Man97]. Moreover, our searching algorithm can deal efficiently with approximate compressed matching, comparing favorably against *Agrep* [WM92], the fastest known software to search (exactly and approximately) uncompressed text.

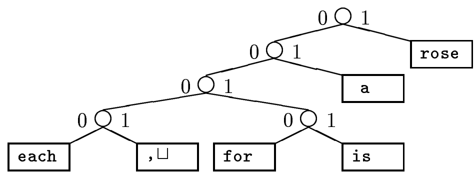
3 The Compression Scheme

Modern 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.

We chose to use word-based semi-static modeling and Huffman coding [Huf52]. In the semi-static modeling the encoder makes a first pass over the text to obtain the parameters (in this case the frequency of each different text word) and perform 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 reasons important 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 [BSTW86, Mof89, HC92].

Since the text is not only composed of words but also of separators, a model must also be chosen for them. In [Mof89, BMN+93] two different alphabets are used: one for words and one for separators. Since a strict alternating property holds, there is no confusion on 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, which 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. This idea was firstly presented in [MNZ97]. They show 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.



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

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

Figure 1: Compression using Huffman coding for spaceless words

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 Kallich [SK64]. 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 [HL90, ZM95].

3.1 Byte-Oriented Huffman Code

The original method proposed by Huffman [Huf52] is mostly used as a binary code. In our work the Huffman code assigned to each text word is a sequence of whole bytes and the Huffman tree has degree 256 instead of 2. All techniques for efficient encoding and decoding mentioned in [ZM95] 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.

The construction of byte Huffman trees involves some details to deal with. As explained in [Huf52], care must be exercised to ensure that the first levels of the tree have no empty nodes when the code is not binary. Figure 2(a) illustrates a case where a naive extension of the binary Huffman tree construction algorithm might generate a non-optimal byte tree. In this example the alphabet has 512 symbols, all with the same probability. The root node has 254 empty spaces that could be ocupied by symbols from the second level of the tree, changing their code lengths from 2 bytes to 1 byte.

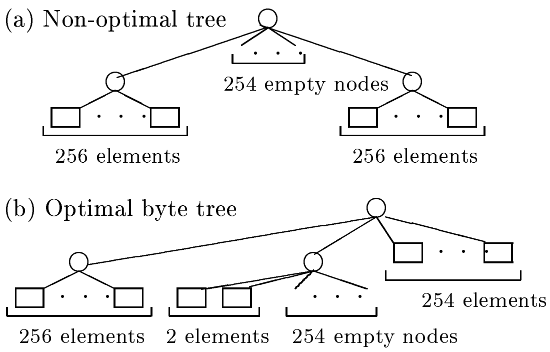


Figure 2: Example of byte Huffman tree

A way to ensure that the empty nodes always go to the lowest level of the tree follows. We calculate previously the number of empty nodes that will arise. We then compose these empty nodes with symbols of smallest probabilities. This step is sufficient to guarantee that the empty nodes will end up at the deepest level of the final tree. The remaining steps are similar to the binary Huffman tree construction algorithm.

In fact, this first coupling step must consider $1+((v-256) \bmod 255)$ symbols, where v is the total number of symbols (i.e. the size of the vocabulary). Applying this formula to our example we have that 2 elements must be coupled with 254 empty nodes in the first step ($1+((512-256) \bmod 255)=2$), as shown in Figure 2(b).

In the remaining part of this section we show that the length of byte Huffman codes does not grow as the text grows, even while the vocabulary does. The key to prove this is to show that the distribution of words in the text is biased enough for the entropy to be $O(1)$, and then to show that byte Huffman code has only a constant overhead over the entropy.

We use the Zipf Law [Zip49] as our model of the frequency of the words appearing in natural language texts. This law is widely accepted in information retrieval. Zipf's law 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. The constant θ depends on the text. 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 Zipf 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 [ANZ97] 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 Law" (i.e. $\theta > 1$). Under this assumption, $H = O(1)$.

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

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:

$$
\begin{aligned}
E(d) &= \sum_{i=1}^{v} p_i \log_d \frac{1}{p_i} \\
&= \frac{1}{\ln d} \sum_{i=1}^{v} \frac{\ln H + \theta \ln i}{i^\theta H} \\
&= \frac{1}{H \ln d} \left(\ln H \sum_{i=1}^{v} \frac{1}{i^\theta} + \sum_{i=1}^{v} \frac{\ln i}{i^\theta} \right) \\
&= \log_d H + \frac{\theta}{H \ln d} \sum_{i=1}^{v} \frac{\ln i}{i^\theta}
\end{aligned}
$$

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 to conclude that $E(d) = O(1)$.

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 in

$\lfloor \log_d(1/p_i) \rfloor$ digits. Therefore, in the worst case the average length of a code in the compressed text is

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

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

This proves that the entropy remains constant as the text grows and therefore our compression ratio will not degrade as the number of different words and separators increases.

If we used the simple Zipf Law instead, the result would be that $E(d) = O(\log n)$, i.e. the average code length grows 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 Law against its simple version.

On the other hand, more refined versions of the Zipf Law exist, such as the Mandelbrot distribution [GBY91]. This law tries to improve the fitting of the Zipf Law for the more frequent values. However, it is mathematically harder to handle and it should not alter our asymptotic results.

3.2 Compression and Decompression Performance

For the experimental results we used literary texts from the TREC collection [Har95]. 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 string of characters in the set {A...Z, a...z, 0...9} separated by other characters not 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.

Table 2 shows the entropy and compression ratios achieved for Huffman, byte Huffman, Unix *Compress* and *gnu Gzip* for the files of the TREC collection. The space used to store the vocabulary is included in the Huffman compression. As it 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.

Method	Files				
	ap	*wsj*	*doe*	*ziff*	*fr*
Entropy	26.20	26.00	24.60	27.50	25.30
Huffman (bits)	27.41	27.13	26.25	28.93	26.88
Byte Huffman	31.16	30.60	30.19	32.90	30.14
Compress	43.80	42.94	41.08	41.56	38.54
Gzip	38.56	37.53	34.94	34.12	27.75

Table 2: Entropy and compression ratios achieved by Huffman, byte Huffman, Compress and Gzip.

It is empirically known that the vocabulary of a text with u words grows sublinearly [Hea78], and hence for large texts the overhead of storing the vocabulary is minimal. However, storing the vocabulary represents an important overhead when the text is small (say, less than 10 megabytes). We therefore compress the vocabulary using standard Huffman on characters. As shown in Figure 3, this makes our compressor better than Gzip for files of at least 1 megabyte. 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.

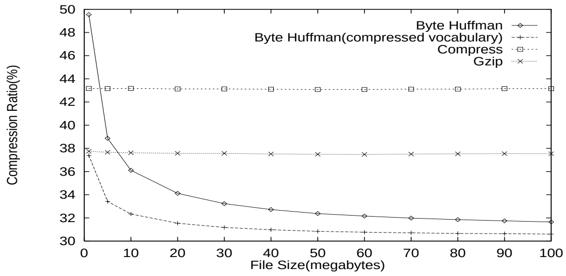


Figure 3: Compression ratios for the *wsj* file compressed by Gzip, Compress, byte Huffman and byte Huffman coding with compressed vocabulary

Table 3 shows the compression and decompression times achieved for Huffman, byte 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.

Our method is more memory-demanding than *Compress* and *Gzip*, which constitutes a drawback. The byte-Huffman algorithm has near $O(\sqrt{u})$ space complexity while the methods used by *Gzip* and *Compress* have constant space complexity. For example, our method needs 10 megabytes of memory to compress and 3.7 megabytes of memory to decompress the file *wsj*, 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 method (i.e. allowing efficient search on the compressed text and fast decompression of fragments) are more important than the space requirements.

4 Searching on Huffman Compressed Text

We show now how we search in the compressed text. We first explain exact matching, then complex patterns, and finally present a filter to speed up the search.

4.1 The Basic Algorithm

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 code, as shown in the left part of Figure 4 for the word `"rose"` (in this example the word `"rose"` has a three-byte codeword `47 131 8`).

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: Text files from the TREC collection

Method	Compression					Decompression				
	ap	*wsj*	*doe*	*ziff*	*fr*	*ap*	*wsj*	*doe*	*ziff*	*fr*
Huffman (bits)	490	526	360	518	440	170	185	121	174	151
Byte Huffman	487	520	356	515	435	106	117	81	112	96
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 seconds) achieved by Huffman, byte Huffman, Compress and Gzip

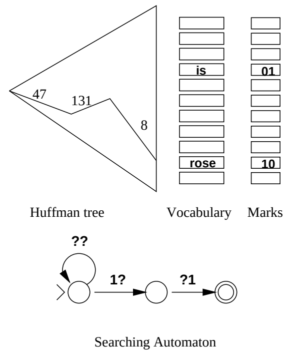


Figure 4: The searching scheme for the pattern `"rose is"`

To search for a pattern we first preprocess it. The preprocessing consists on searching it in the vocabulary and marking the corresponding entry. This search can be very efficient, for instance binary search or hashing. 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"` is the second in the pattern and `10` indicates that the word `"rose"` is the first in the pattern. If any word of the pattern is not found in the vocabulary we immediately know that it 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. We report an occurrence of a symbol whenever we reach a leaf of the Huffman tree. At each word symbol obtained we send the corresponding bit mask to an automaton, as illustrated in Figure 4. This nondeterministic automaton allows to move 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 by the shift-or algorithm [BYG92]. 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 words using a single computer word as the bit mask. For longer phrases we use as many computer words as needed.

Finally, we show how to deal with separators and stopwords. Most online searching 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 to the elimination of stopwords, which are usually disregarded in indexing schemes and are difficult to disregard in online searching. In our compression scheme, we know which elements of the vocabulary correspond in fact to separators, and which correspond to stopwords: at compression time we mark them so that the searching algorithm ignores them. Therefore, we eliminate separators and stopwords from the sequence (and from the search pattern) at negligible cost.

4.2 Extending the Basic Algorithm for Complex Patterns

Before entering into details of the searching algorithms for complex patterns we mention 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 characters at each position. In addition, 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. In the Appendix we present in detail each type of query supported by our system.

For complex patterns the preprocessing phase corresponds to a sequential search in the vocabulary to mark all the words that match the pattern. This technique has been already used in block oriented indexing schemes for searching allowing errors in uncompressed texts [MW93, ANZ97]. 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.

Each word of the pattern is searched separately in the vocabulary using a sequential pattern matching algorithm. 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 $k = 1$ (allowing 1 error). For instance, the word `"rose"` in the vocabulary matches the pattern in positions 1 and 2.

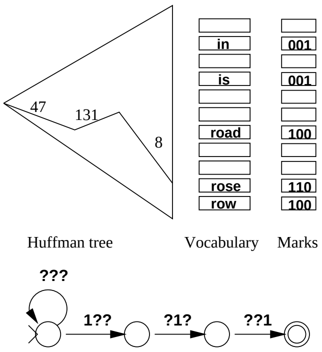


Figure 5: General searching scheme for the phrase `"ro* rose is"` allowing 1 error

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 [BYN96]. 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 [WM92], which runs in $O(kv + w)$ time to search W.

The compressed text scanning phase is similar to the one described above in Section 4.1.

4.3 Improving the Search with Boyer-Moore Filtering

We show in this section how the search on the compressed text is improved even more. The central idea is to search the compressed pattern directly in the text, with a fast Boyer-Moore-Horspool-Sunday (BMHS) [Sun90] algorithm. This avoids inspecting all the bytes in the compressed text, as the BMHS algorithm can be as good as $O(n/c)$, where n is the size in bytes of the compressed text and c is the length of the compressed pattern.

To search for a single word we first look for the word in the vocabulary and obtain its compressed code. Following Figure 6, to look for `"rose"` we search for the three-byte string `47 131 8`. This code is searched directly in the compressed text. Every time a match is found in the compressed text we must verify whether this match indeed corresponds to a word. To see that this verification is necessary, consider the word `"ghost"` in the example presented in Figure 6.

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 codes are aligned to the beginning of blocks, so that no code crosses a block boundary. Therefore, we only need to run the basic algorithm from the beginning of the block that contains the match.

Figure 6: An example where the code of a word is present in the compressed text but the word is not present in the original text

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 run a number of experiments on the *wsj* file for blocks of sizes 64, 128, 256, and 512 bytes. The worst search performance was for blocks of 64 bytes, due to the overhead of treating flags at the end of each block. The performance improved for blocks of 128 and 256 bytes and decreased for blocks of 512 bytes, the latter due to the overhead of performing more verification work at each potential match. Therefore, a good time-space tradeoff for the block size is 256 bytes.

In order to search for complex queries, we first search the vocabulary as explained in Section 4.2. Once the set of all words is obtained, we search for *all* the codes in the text using an extension of the BMHS algorithm to handle multiple patterns [BYN96, BYN97a].

In order to search a phrase pattern, we simply take one of the words of the phrase as its representative. Once the code of that element is found in the compressed text, the verification phase searches the whole phrase. Since we are free to use any representative, we take the one with longest code (i.e. the most infrequent word), as BMHS searching improves with longer patterns.

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.

4.4 Analytical Results

We analyze the performance of our searching algorithm. The analysis considers a random text, which is very appropriate because the compressed text is mainly random.

It is empirically known that the vocabulary of a text with u words grows as u^β [Hea78, ANZ97, MNZ97]. For the analysis we consider that: the vocabulary has $v = O(u^\beta) \simeq O(\sqrt{u})$ words (typically $\beta = 0.4..0.6$), the compressed search patterns are of length c (typically c is equal to 3 or 4 bytes), the original text has u characters, the compressed text has n characters, a complex query matches p words in the vocabulary (typically $p = O(u^{0.1..0.2})$ [BYN97b]), k is the number of errors allowed, σ is the coding alphabet ($\sigma = 256$ symbols), the pattern has m characters and j different words of length $w_1, ..., w_j$ ($\sum_{i=1}^{j} w_i = m$). Finally, we align the codes at the boundaries of blocks of b bytes.

We first consider the preprocessing phase. Looking exactly for a word of length w in the vocabulary can be done in $O(w)$ in the worst case by using a trie or on average by using hashing. Therefore, looking exactly for all words in the pattern has a cost of $O(\sum_{i=1}^{j} w_i) = O(m)$. On the other hand, if we search a complex pattern we preprocess all the words at a cost $O(ju^\beta + \sum_{i=1}^{j} w_i) = O(ju^\beta + m)$ or $O(jku^\beta + \sum_{i=1}^{j} w_i) = O(jku^\beta + m)$ depending on the algorithm used. In all reasonable cases the preprocessing phase is sublinear in the text size and negligible in cost. Since k is taken as a constant, the preprocessing cost is $O(ju^\beta + m) = O(mu^\beta)$, which is close to $O(m\sqrt{u})$.

We consider now text searching for natural language texts. The basic algorithm works $O(1)$ per compressed byte and therefore the search time is $O(n+t)$ in the worst case, where t is the preprocessing costs presented above. This worst-case complexity is independent on the complexity of the search.

On the other hand, the algorithm using BMHS filter does not inspect all the characters. If the pattern is a phrase we just look for one element of the phrase, so we restrict our attention to single-word queries.

If the query matches a single word in the vocabulary, the search time is very close to $O(n/c+t)$, where c is the length of the compressed word. This is because the alphabet size (256) is much larger than the length of the codes (3 or 4 at most). Even in the case of a query matching a few words in the vocabulary, the above formula holds for the multipattern BMHS algorithm, provided c is the length of the shortest code among the vocabulary words matching the query (since the shortest code is the maximum length that the multipattern BMHS algorithm can use). Notice that we refer to the shortest code among those that match the selected word in the phrase, regardless of the fact that the algorithm selects the word whose shortest matching code is maximized.

In case of a complex query which generates many different patterns to search for, the effectiveness of the filter can be degraded, being $O(n+t)$ (albeit the constant is smaller than in the basic algorithm). The exact constant is an open problem [BYR92].

However, even when using a filter, some blocks must be traversed with the basic algorithm to verify matches found by the filter. When searching p codes of length c in parallel, the probability of finding anyone in a given block of length b is $(1-(1-p/\sigma^c)^b)$, and therefore the total cost for verifications is

$$n\left(1-\left(1-\frac{p}{\sigma^c}\right)^b\right) \leq n\left(1-e^{-bp/\sigma^c}\right)$$

To give an idea of the real numbers involved, consider that we search for a single word whose code has 3 bytes, and that we use $b = 64$. In this case, the BMHS filter inspects on average 2 bytes to advance 4 positions, therefore inspecting a total of $n/2$ bytes. The proportion of bytes inspected due to verifications is smaller than 4×10^{-6}. Since n is close to $u/3$, we inspect $u/6$ bytes, which is close to a BMHS algorithm run over the original text. Therefore, in this case our CPU costs are similar. However, we perform only one third of the I/O required by the uncompressed searching, which makes our search significantly faster.

A well-accepted rule in Information Retrieval is that queries are uniformly distributed across the vocabulary. This makes queries with large c much more probable than those with code length 1 or 2. Moreover, very short code lengths correspond to stopwords and may therefore be forbidden.

To search for a very complex pattern that makes the filter unsuitable, we simply run the shift-or algorithm, inspecting all $n \approx u/3$ bytes. However, in the uncompressed version we would work at the very best $O(1)$ per original text character, which is three times our cost. The I/O times are also in our case one third of those of uncompressed searching.

4.5 Searching Performance

The performance evaluation of the algorithms presented in the previous sections was obtained using 120 randomly chosen patterns. In fact we considered 40 patterns containing 1 word, 40 patterns containing 2 words, 40 patterns containing 3 words, and submitted each one to the searching algorithms. All experiments were run on the *wsj* text file and the results were obtained with 99% confidence. The sizes of the *wsj* uncompressed and compressed files were 262.8 and 80.4 megabytes, respectively.

Table 4 presents exact ($k = 0$) and approximate ($k = 1, 2, 3$) searching times using *Agrep* [WM92], *Cgrep filterless*, and *Cgrep* with Boyer-Moore filtering for blocks of 256 bytes. It can be seen from this table that *Cgrep filterless* is almost insensitive to the number of errors allowed in the pattern while *Agrep* is not. This happens because the filterless version maps all the queries to the same automaton that does not depend on k. It also shows that for exact searching *Cgrep filter* is almost twice as fast as *Agrep* 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 *Cgrep* to read the *wsj* compressed file and approximately 20 seconds for *Agrep* to read the *wsj* uncompressed file.

The following test was for three different types of patterns, as follows:

1. `prob#atic sign#ance`: where `#` means any character considered zero or more times (one possible answer is `"problematic significance"`)
2. `petroleum services lines`
3. `Brasil`

Table 5 presents exact ($k = 0$) and approximate ($k = 1, 2$) searching times using *Agrep*, *Cgrep filterless*, and *Cgrep* with Boyer-Moore filtering for blocks of 256 bytes.

5 Conclusion and Future Work

In this paper we investigated a fast compression and decompression scheme for natural language texts and also presented an algorithm which allows to search for exact and approximate compressed matches. We analyzed our algorithms and presented experimental results on their performance for natural language texts. 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*.

For exact searching our algorithm is $O(n+m)$ time (which is optimal), using $O(\sqrt{u})$ extra space, where n is the size of the compressed text, m is the size of the pattern and u is the size of the uncompressed text. For approximate searching or complex queries our algorithm is near $O(n+m\sqrt{u})$ time using $O(\sqrt{u})$ extra space. We also presented a fast Boyer-Moore-type filter to speed up the search which is close to $O(n/c+m\sqrt{u})$ on average and uses $O(\sqrt{u})$ extra space, where c is the length of the shortest code among the words matching the pattern.

An example of the power of our compressed matching algorithm 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). It runs at 5.4 megabytes per second, which is equivalent to searching the original text at 17.5

Algorithm	$k=0$	$k=1$	$k=2$	$k=3$
Agrep	23.8 $\pm$ 0.38	117.9 $\pm$ 0.14	146.1 $\pm$ 0.13	174.6 $\pm$ 0.16
Cgrep filterless	22.1 $\pm$ 0.09	23.1 $\pm$ 0.14	24.7 $\pm$ 0.21	25.0 $\pm$ 0.49
Cgrep filter	15.1 $\pm$ 0.30	16.2 $\pm$ 0.52	17.0 $\pm$ 0.71	22.7 $\pm$ 2.23

Table 4: Searching times (in seconds) for the *wsj* text file.

Pattern	$k=0$			$k=1$			$k=2$		
	Agrep	*Cgrep filterless*	*Cgrep filter*	*Agrep*	*Cgrep filterless*	*Cgrep filter*	*Agrep*	*Cgrep filterless*	*Cgrep filter*
1	73	26	15	117	26	17	145	29	22
2	22	24	15	117	25	15	145	25	17
3	24	23	14	117	23	15	145	23	20

Table 5: Searching times (in seconds) for the *wsj* text file.

megabytes per second. As *Agrep* searches the original text at 2.25 megabytes per second, *Cgrep* is 7.8 times faster than *Agrep*.

Currently, we are integrating our results in block oriented indexing schemes similar to *Glimpse* [MW93, ANZ97], where this work can be nicely applied. We are also working in a new compression scheme where the compressed text does not need to be decoded at search time, and any known sequential pattern matching algorithm can be used for exact search [MNZB98].

Acknowledgements

We wish to acknowledge the helpful comments of Berthier Ribeiro-Neto and the many fruitful discussions with Marcio D. Araújo, who helped particularly with the algorithms for approximate searching in the text vocabulary.

References

[AB92] A. Amir and G. Benson. Efficient two-dimensional compressed matching. *Proc. Second IEEE Data Compression Conference*, pages 279–288, Mar. 1992.

[ABF96] A. Amir, G. Benson and M. Farach. Let sleeping files lie: pattern matching in z-compressed files. *Journal of Computer and Systems Sciences*, 52(2):299–307, 1996.

[ANZ97] M. D. Araújo, G. Navarro and N. Ziviani. Large text searching allowing errors. In R. Baeza-Yates, editor, *Proc. of the Fourth South American Workshop on String Processing*, Carleton University Press International Informatics Series, v. 8, pages 2–20, 1997.

[BYG92] R. Baeza-Yates and G.H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10): 74–82, 1992.

[BYN96] R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Proc. of Combinatorial Pattern Matching (CPM'96)*, Springer-Verlag LNCS, v. 1075, pages 1–13, 1996.

[BYN97a] R. Baeza-Yates and G. Navarro. Multiple approximate string matching. In *Proc. of Workshop on Algorithms and Data Structures (WADS'97)*, Springer-Verlag LNCS, v. 1272, pages 174–184. 1997.

[BYN97b] R. Baeza-Yates and G. Navarro. Block addressing indices for approximate text retrieval. In *Proc. of Sixth ACM International Conference on Information and Knowledge Management CIKM'97*, pages 1–8, Las Vegas, Nevada, 1997.

[BYR92] R. Baeza-Yates and M. Régnier. Average running time of the Boyer-Moore-Horspool algorithm. *Theoretical Computer Science* 92(1): 19–31, Jan 1992. Elsevier Science Publishers.

[BMN+93] T. C. Bell, A. Moffat, C. Nevill-Manning, I. H. Witten and J. Zobel. Data compression in full-text retrieval systems. *Journal of the American Society for Information Science*, 44: 508–531, 1993.

[BSTW86] J. Bentley, D. Sleator, R. Tarjan and V. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29: 320–330, 1986.

[FT95] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. In *Proc. 27th ACM Annual Symposium on the Theory of Computing*, pages 703–712, 1995.

[GBY91] G. H. Gonnet and R. Baeza-Yates. Handbook of Algorithms and Data Structures. Addison-Wesley, 1991.

[Har95] D. K. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, National Institute of Standards and Technology Special Publication 500-207, Gaithersburg, Maryland, 1995.

[Hea78] J. Heaps. *Information Retrieval - Computational and Theoretical Aspects.* Academic Press, 1978.

[HL90] D. S. Hirschberg and D. A. Lelewer. Efficient Decoding of Prefix Codes. *Communications of the ACM*, 33(4): 449–459, 1990.

[HC92] R. N. Horspool and G. V. Cormack. Constructing Word-Based Text Compression Algorithms. *IEEE Proc. Second Data Compression Conference*, pages 62–81, 1992.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the Institute of Electrical and Radio Engineers*, 40(9): 1090–1101, 1952.

[Man97] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM Transactions on Information Systems*, 15(2): 124–136, 1997.

[MW93] U. Manber and S. Wu. Glimpse: a tool to search through entire file systems. Tech. Report 93-34, Dept. of Computer Science, Univ. of Arizona, Oct 1993.

[Mof89] A. Moffat. Word-based text compression. *Software Practice and Experience*, 19(2): 185–198, 1989.

[MNZ97] E. de Moura, G. Navarro and N. Ziviani. Indexing compressed text. In R. Baeza-Yates, editor, *Proc. of the Fourth South American Workshop on String Processing*, Carleton University Press International Informatics Series, v.8, pages 95–111, 1997.

[MNZB98] E. de Moura, G. Navarro, N. Ziviani and R. Baeza-Yates. Direct Pattern Matching on Compressed Text. Tech. Report 03-98, Dept. of Computer Science, Univ. Federal de Minas Gerais, Brazil, Apr 1998.

[SK64] E. S. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Communications of the ACM* 7: 166–169, 1964.

[Sun90] D. Sunday. A very fast substring search algorithm. *Communications of the ACM* 33(8): 133–142, 1990.

[WM92] S. Wu, U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10): 83–91, 1992.

[Zip49] G. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison-Wesley, 1949.

[ZL76] J. Ziv and A. Lempel. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 22: 75–81, 1976.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3): 337–343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5): 530–536, 1978.

[ZM95] J. Zobel and A. Moffat. Adding compression to a full-text retrieval system. *Software Practice and Experience*, 25(8): 891–903, 1995.

Appendix: 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 characters 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).