Direct Pattern Matching on Compressed Text

Edleno S. de Moura*
Depto. de Ciência da Computação
Univ. Federal de Minas Gerais
Belo Horizonte, Brasil
edleno@dcc.ufmg.br

Gonzalo Navarro†
Depto. de Ciencias de la Computación
Univ. de Chile
Santiago, Chile
gnavarro@dcc.uchile.cl

Nivio Ziviani‡
Depto. de Ciência da Computação
Univ. Federal de Minas Gerais
Belo Horizonte, Brasil
nivio@dcc.ufmg.br

Ricardo Baeza-Yates†
Depto. de Ciencias de la Computación
Univ. de Chile
Santiago, Chile
rbaeza@dcc.uchile.cl

Abstract

We present a fast compression and decompression technique for natural language texts. The novelty is that the exact search can be done on the compressed text directly, using any known sequential pattern matching algorithm. Approximate search can also be done efficiently without any decoding. The compression scheme uses a semi-static word-based modeling and a Huffman coding where the coding alphabet is byte-oriented rather than bit-oriented. We use the first bit of each byte to mark the beginning of a word, which allows the searching of the compressed pattern directly on the compressed text. We achieve about 33% compression ratio for typical English texts. When searching for simple patterns, our experiments show that running our algorithm on a compressed text is almost 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.

1. Introduction

In this paper we present a new technique to search for patterns on compressed texts, where the patterns are compressed and the search is processed without any decoding of the compressed text. The main advantages of this scheme are simplicity, efficiency and flexibility, since complex searches can be performed almost at the same search cost of simple searches. Moreover, the reduced I/O to read the compressed text makes this algorithm even faster than those that work on plain uncompressed text.

The *compressed matching problem* was first defined in the work of Amir and Benson [1] as the task of performing string matching in a compressed text without decompressing it. Giving 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 [2], 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.

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.

In our compression scheme we use a word-based Huffman code where the code assigned to each text word is a sequence of whole bytes and the Huffman tree has degree

*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 and AMYRI Project.

128 instead of 2. Consequently, in each byte we use 7 bits for the Huffman code and 1 bit to signal the beginning of a codeword. As shown in [8], using bytes instead of bits does not significantly degrade the compression ratios. The positive aspects are that byte processing is much faster than bit processing, and compression and decompression are very fast.

Our searching algorithm 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 on. For exact searching any conventional pattern matching algorithm can be used and for approximate searching we use a multi-pattern matching algorithm, as explained later on.

Let u, n and m be as defined above. Let j be the number of text words of the pattern. 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 $O(n + j\sqrt{u} + m)$ time and near $O(\sqrt{u})$ extra space.

The algorithms presented in this paper are being used in a software package called *cgrepboyer*. *cgrepboyer* 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/cgrepboyer`, 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 the byte word based Huffman compression, the method used as a base for our method . In Section 4 we show our compression method and presents experimental compression performance results. In Section 5 we present a method to search for patterns on compressed texts. In this section we also show some analytical and experimental results. In Section 6 we present some conclusions and future work directions.

2. Related Work

In [9] it was presented a compressed matching algorithm for the LZ1 classic compression scheme [16] that runs in time $O(n \log^2(u/n) + m)$. In [2] 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 $O(n + m)$. The works in [9, 2] were presented in a theoretical framework, with no implementation of their algorithms, while we implement and experimentally evaluate our algorithms. Moreover, our implementation considers both the compression/decompression schemes and the exact and approximate compressed matching problems.

Another text compression scheme that allows direct searching was proposed by [13]. His scheme packs pairs of frequent characters in a single byte, leading to a compression ratio of approximately 70% for typical text files. Like that 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 (33% against 70%) the compression ratio of [13]. Moreover, our searching algorithm can deal efficiently with approximate compressed matching, comparing favorably against *agrep* [15], the fastest known software to search uncompressed text.

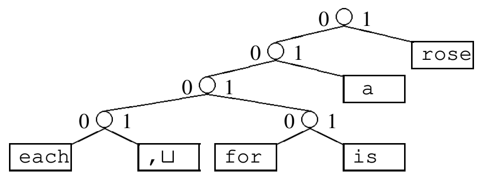
The searching algorithm presented in this paper is a variant of the scheme presented in [8], which provides good compression ratio and also permits fast searching on the compressed text. However, they concomitantly traverse the Huffman tree downwards, as if they were decompressing the text. In our new scheme, we do not need to decode the text at search time. Consequently, any known sequential pattern matching algorithm can be used for exact search. Approximate search can also be done efficiently by generating all the possible compressed patterns that matches with the query and making a multi-pattern search over the compressed text.

3. The Compression Scheme

We use a zero-order semi-static word-based modeling and Huffman coding [12]. 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 performs the actual compression in a second pass. In a word-based model [6, 14] the words of the text are taken as the alphabet to be used during the coding phase. Since the text is not only composed of words but also of separators, a model must also be chosen for them. We parse the text into *spaceless words* [8]. In this method words and separators are placed in the same alphabet. 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 [7].

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 1, 2, 1, 1, 1, 3, respectively (␣ represents a space).

The original method proposed by Huffman [12] uses a binary code. In [8] the Huffman code assigned to each text word is a sequence of whole bytes and the Huffman tree has degree 256 instead of 2. They use 8 bits instead of 1 bit at each level of the Huffman tree. Because of this we call



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

this compression scheme *Byte Huffman*. In practice byte processing is much faster than bit processing because bit shifts and masking operations are not necessary at decoding or search time.

As pointed out in Moura et al. [8], searching in a text compressed by the Byte-Huffman scheme cannot be done by compressing the pattern and applying a conventional search algorithm. This is due to the possibility of finding false matches during the search. The problem is exemplified in Figure 2, where the words `real, word, ghost` have received the 8-bit codes (or byte-codes) `85 32, 29 12 99, 32 29 12`, respectively. If we are searching for the word `ghost`, which does not exist in the original text, we find a false occurrence.

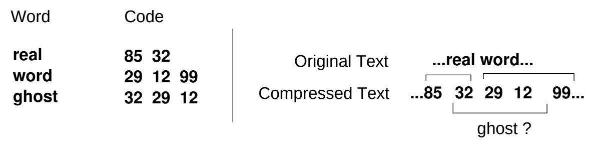


Figure 2. An example where the code of a word is present in the Byte Huffman compressed text but the word is not present in the original text.

We solve this problem by identifying the point where each word starts in the compressed text. If we knew that byte `32` in the compressed text of the Figure 2 is not the first byte of a word, we would simply not report a match. This can be done by using a Huffman tree of degree 128, instead of 256, and using the first bit of the byte-code as a flag bit. This bit has the value 1 only when a digit in the compressed text is the first digit of a word and 0 otherwise.

In the example, if we search `"ghost"`, we do not search the byte string `32 29 12` but `160 29 12` instead (where $160 = 32 + 128$). Therefore, our final code is also a byte oriented code, which permits the application of conventional text search algorithms directly on the compressed text.

As we show below in the experimental results section, no significant decrease 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. All techniques for efficient encoding and decoding mentioned in [17] can easily be extended to this case.

4. Compression and Decompression Performance

For the experimental results we used literary texts from the TREC collection [10]. 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 $\{A \ldots Z, a \ldots z, 0 \ldots 9\}$ separated by other characters not in the set $\{A \ldots Z, a \ldots z, 0 \ldots 9\}$. 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	#Word
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

Table 2 shows the entropy and compression ratios achieved for Huffman, byte Huffman, 7-bits Huffman, Compress and Gzip for files of the TREC collection. The space used to store the vocabulary is included in all Huffman compression methods. The overhead of our method compared with the Byte Huffman is near 3%. As it can be seen, the increase in the compression ratio with the 7-bits method is not so significant. In most cases we are still below Gzip.

Table 3 shows the compression and decompression times achieved for Huffman, byte Huffman, 7-bits Huffman, Compress and Gzip for files of the TREC collection. Our method has almost the same performance of the Byte Huffman method. In compression, we are 2-3 times faster than Gzip and only 17% slower than Compress (whose compression ratios are worse than those of Gzip). In decompression we are more than 20% faster than Gzip and three times faster than Compress.

Method	Files				
	AP	WSJ	DOE	ZIFF	FR
Entropy	26.20	26.00	24.60	27.50	25.30
Huffman (1 bit)	27.41	27.13	26.25	28.93	26.88
Byte Huffman	31.16	30.60	30.19	32.90	30.14
7-bits Huffman	34.12	33.70	32.74	36.08	33.53
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, 7-bits Huffman, Compress and Gzip.

Method	Compression					Decompression				
	AP	WSJ	DOE	ZIFF	FR	AP	WSJ	DOE	ZIFF	FR
Huffman (1 bit)	490	526	360	518	440	170	185	121	174	151
Byte Huffman	487	520	356	515	435	106	117	81	112	96
7-bits Huffman	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 seconds) achieved by Huffman, byte Huffman, 7-bits Huffman, Compress and Gzip.

5. Searching on Compressed Text

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 search algorithm. In an approximate search, the first phase generate all the possibilities of compressed codes 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.

5.1. Compressing the Pattern

Compressing the pattern when we are performing an exact search is similar to the coding phase of the Huffman compression. We search for each symbol (word or separator) of the pattern in the Huffman vocabulary and then generate the compressed code for it. If there is a symbol 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 search, then we need to generate compressed codes for all symbols in the Huffman vocabulary that match with the symbols in the pattern. For each symbol in the pattern we make a list of the compressed codes of the vocabulary symbols that matches with it.

To search the symbols in the vocabulary we use the same algorithms proposed by Moura et al [8]. For phrase patterns allowing k errors ($k \geq 0$) that contains sets of characters at any position we use the algorithm presented in [4]. If v is the size of the vocabulary and w is the length of a symbol W, this runs in $O(v + w)$ time to search W. For more complicated patterns allowing k errors ($k \geq 0$) containing unions, wild cards or regular expressions we use the algorithm presented in [15], which runs in $O(kv + w)$ time to search W.

5.2. Searching Phase

For exact search, after obtaining the compressed code (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 Boyer-Moore-Horspool-Sunday (BMHS) [?] algorithm, which has good practical performance.

If we are doing approximate search then the original pattern is represented by the set of lists $L_1, .., L_j$, where L_i has the compressed codes that matches the i-th symbol of the original pattern. To start the search in the compressed text we choose one of these lists and use any multi-pattern search algorithm to find the occurrences of their elements in the text. When an occurrence of one element of the first list searched is found, we look the other lists to verify if there is an occurrence of the entire pattern at this position.

The choice of the first list searched is fundamental for the performance of the algorithm. An heuristic to make this choice is to verify the size of the smallest code in each list and choose the list that has the largest value among these. This choice comes directly from the cost to search a list of patterns. Longer code words 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.

The multi-pattern search algorithm chosen to search the elements of each list is an efficient technique proposed by Baeza-Yates and Navarro [4, 5] to handle multiple patterns. This algorithm is an extension of the Boyer-Moore-Horspool-Sunday (BMHS) [?] algorithm, which has a cost of $O(n \log(c)/c)$ on average, where n is the size in bytes of the compressed text and c is the length of the smaller compressed pattern searched.

5.3. 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.

For the analysis we consider that: the vocabulary has $v = O(u^\beta)$ words, for $0 < \beta < 1$ [11]. Typically, β is between 0.4 and 0.6 [3, 7], and therefore v is close to $O(\sqrt{u})$ (although this is not essential to our analysis). The compressed search patterns are of length c (typically $c =$ 3 or 4 bytes, but this is not essential for the analysis), the original text has u characters, the compressed text has n characters, k is the number of errors allowed, the pattern has m characters and j different words of length $m_1, ..., m_j$ ($\sum_{i=1}^{j} m_i = m$).

We first consider the compression phase, where the main cost is searching in the vocabulary. Looking exactly for a word of length w in the vocabulary can be done in $O(m)$ 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 $O(\sum_{i=1}^{j} m_i) = O(m)$. On the other hand, if we search a complex pattern we preprocess all the words at a cost $O(ju^\beta + \sum_{i=1}^{j} m_i) = O(ju^\beta + m)$ or $O(jku^\beta + \sum_{i=1}^{j} m_i) = O(jku^\beta + m)$ depending on the algorithm used. In all reasonable cases the preprocessing phase is sub-linear in the text size and negligible in cost.

We consider now text searching for natural language texts. The multi pattern algorithm used in the second phase makes the search time near $O((n \log(c)/c) + t)$ on average, where c is the length of the shortest code among the words in the compressed pattern list chosen.

5.4. Searching Performance

The performance evaluation of the algorithms presented in this section was obtained by means of 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 interval. The size of the WSJ uncompressed was 262.8 megabytes and the size of the compressed files were 80.4 megabytes with the Byte Huffman method and 88.6 megabytes with the 7-bits Huffman method.

The programs used in the experiment are: agrep, a software developed by Wu and Manber [15]; cgrep [8], a software to search on compressed texts obtained with Byte Huffman; and cgrep7bits is our program to search on compressed texts using the algorithms described in this section.

The table 4 shows that our method is slightly better than cgrep for both exact and approximate search. When compared with the agrep, cgrep7bits is almost eigth times faster.

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
cgrep	15.1 ± 0.30	16.2 ± 0.52	19.4 ± 1.21	23.4 ± 1.79
cgrep7bits	14.1 ± 0.18	15.0 ± 0.33	17.0 ± 0.71	22.7 ± 2.23

Table 4. Searching times (in seconds) for the WSJ text file.

Note that agrep were run on the uncompressed version of the text and cgrep and cgrep7bits were run on the compressed version of the text. The exact search performed in the experiments of this extended abstract is done by discarding the separators during the search. We will perform experiments for exact searching without discarding the separators at the final version of this article.

6. Conclusions

We have presented a new compression scheme that has good compression and decompression performance with good compression ratio. We also presented a new algorithm to perform compressed pattern matching over the texts generated by the compression algorithm. Our scheme permits direct search on the compressed text by using any conventional algorithm for exact search and also permits approximate search by converting the search into a multi-pattern exact search. The method does not decode the text at search time. Our searching scheme is close in performance to the fastest algorithm to search on compressed text known in the literature [8].

References

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

[2] 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.

[3] 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*, volume 8, pages 2–20. Carleton University Press International Informatics Series, 1997.

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

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

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

[7] 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*, volume 8, pages 95–111. Carleton University Press International Informatics Series, 1997.

[8] E. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast text searching allowing errors. In *Proc. of the ACM Sigir'98*. to appear.

[9] M. Farach and M. Thorup. String matching in lempel-ziv compressed strings. In *27th ACM Annual Symposium on the Theory of Computing*, pages 703–712.

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

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

[12] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Proc. of the I.R.E.*, volume 40, pages 1090–1101, 1952.

[13] 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.

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

[15] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10), 1992.

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

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