Speeding Up Pattern Matching by Text Compression

Yusuke Shibata[1], Takuya Kida[1], Shuichi Fukamachi[2], Masayuki Takeda[1], Ayumi Shinohara[1], Takeshi Shinohara[2], and Setsuo Arikawa[1]

[1] Dept. of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan
{yusuke, kida, takeda, ayumi, arikawa}@i.kyushu-u.ac.jp
[2] Dept. of Artificial Intelligence, Kyushu Institute of Technology, Iizuka 320-8520, Japan
{fukamati, shino}@ai.kyutech.ac.jp

Abstract. Byte pair encoding (BPE) is a simple universal text compression scheme. Decompression is very fast and requires small work space. Moreover, it is easy to decompress an arbitrary part of the original text. However, it has not been so popular since the compression is rather slow and the compression ratio is not as good as other methods such as Lempel-Ziv type compression.
In this paper, we bring out a potential advantage of BPE compression. We show that it is very suitable from a practical view point of *compressed pattern matching*, where the goal is to find a pattern directly in compressed text without decompressing it explicitly. We compare running times to find a pattern in (1) BPE compressed files, (2) Lempel-Ziv-Welch compressed files, and (3) original text files, in various situations. Experimental results show that pattern matching in BPE compressed text is even faster than matching in the original text. Thus the BPE compression reduces not only the disk space but also the searching time.

1 Introduction

Pattern matching is one of the most fundamental operations in string processing. The problem is to find all occurrences of a given pattern in a given text. A lot of classical or advanced pattern matching algorithms have been proposed (see [8,1]). The time complexity of pattern matching algorithm is measured by the number of symbol comparisons between pattern and text symbols. The Knuth-Morris-Pratt (KMP) algorithm [19] is the first one which runs in linear time proportional to the sum of the pattern length m and the text length n. The algorithm requires additional memory proportional to the pattern length m. One interesting research direction is to develop an algorithm which uses only constant amount of memory, preserving the linear time complexity (see [11,7,5,13,12]). Another important direction is to develop an algorithm which makes a sublinear number of comparisons on the average, as in the Boyer-Moore (BM) algorithm [4] and its variants (see [24]). The lower bound of the average case time complexity is known to be $O(n \log m/m)$ [27], and this bound is achieved by the algorithm presented in [6].

G. Bongiovanni, G. Gambosi, R. Petreschi (Eds.): CIAC 2000, LNCS 1767, pp. 306–315, 2000.
© Springer-Verlag Berlin Heidelberg 2000

From a practical viewpoint, the constant hidden behind O-notation plays an important role. Horspool's variant [14] and Sunday's variant [22] of the BM algorithm are widely known to be very fast in practice. In fact, the former is incorporated into a software package `Agrep`, which is understood as the fastest pattern matching tool developed by Wu and Manber [25].

Recently, a new trend for accelerating pattern matching has emerged: *speeding up pattern matching by text compression*. It was first introduced by Manber [20]. Contrary to the traditional aim of text compression — to reduce space requirement of text files on secondary disk storage devices —, text is compressed in order to speed up the pattern matching process.

It should be mentioned that the problem of pattern matching in compressed text without decoding, which is often referred to as *compressed pattern matching*, has been studied extensively in this decade. The motivation is to investigate the complexity of this problem for various compression methods from the viewpoint of combinatorial pattern matching. It is theoretically interesting, and in practice some algorithms proposed are indeed faster than a regular decompression followed by a simple search. In fact, Kida et al. [18,17] and Navarro et al. [21] independently presented compressed pattern matching algorithms for the Lempel-Ziv-Welch (LZW) compression which run faster than a decompression followed by a search. However, the algorithms are slow in comparison with pattern matching in uncompressed text if we compare the CPU time. In other words, the LZW compression did not speed up the pattern matching.

When searching text files stored in secondary disk storage, the running time is the sum of file I/O time and CPU time. Obviously, text compression yields a reduction in the file I/O time at nearly the same rate as the compression ratio. However, in the case of an adaptive compression method, such as Lempel-Ziv family (LZ77, LZSS, LZ78, LZW), a considerable amount of CPU time is devoted to an extra effort to keep track of the compression mechanism. In order to reduce both of file I/O time and CPU time, we have to find out a compression scheme that requires no such extra effort. Thus we must re-estimate the performance of existing compression methods or develop a new compression method in the light of the new criterion: *the time for finding a pattern in compressed text directly*.

As an effective tool for such re-estimation, we introduced in [16] a unifying framework, named *collage system*, which abstracts various dictionary-based compression methods, such as Lempel-Ziv family, and the static dictionary methods. We developed a general compressed pattern matching algorithm for strings described in terms of collage system. Therefore, any of the compression methods that can be described in the framework has a compressed pattern matching algorithm as an instance.

Byte pair encoding (BPE, in short) [10], included in the framework of collage systems, is a simple universal text compression scheme based on the pattern-substitution [15]. The basic operation of the compression is to substitute a single character which did not appear in the text for a pair of consecutive two characters which frequently appears in the text. This operation will be repeated until either all characters are used up or no pair of consecutive two characters

appears frequently. Thus the compressed text consists of two parts: the substitution table, and the substituted text. Decompression is very fast and requires small work space. Moreover, partial decompression is possible, since the compression depends only on the substitution. This is a big advantage of BPE in comparison with adaptive dictionary based methods. Despite such advantages, the BPE method has received little attention, until now. The reason for this is mainly the following two disadvantages: the compression is terribly slow, and the compression ratio is not as good as other methods such as Lempel-Ziv type compression.

In this paper, we pull out a potential advantage of BPE compression, that is, we show that BPE is very suitable for speeding up pattern matching. Manber [20] also introduced a little simpler compression method. However since its compression ratio is not so good and is about 70% for typical English texts, the improvement of the searching time cannot be better than this rate. The compression ratio of BPE is about 60% for typical English texts, and is near 30% for biological sequences. We propose a compressed pattern matching algorithm which is basically an instance of the general one mentioned above. Experimental results show that, in CPU time comparison, the performance of the proposed algorithm running on BPE compressed files of biological sequences is better than that of `Agrep` running on uncompressed file of the same sequences. This is not the case for English text files. Moreover, the results show that, in elapsed time comparison, the algorithm drastically defeats `Agrep` even for English text files.

It should be stated that Moura et al. [9] proposed a compression scheme that uses a word-based Huffman encoding with a byte-oriented code. The compression ratio for typical English texts is about 30%. They presented a compressed pattern matching algorithm and showed that it is twice faster than `Agrep` on uncompressed text in the case of exact match. However, the compression method is not applicable to biological sequences because they cannot be segmented into words. For the same reason, it cannot be used for natural language texts written in Japanese in which we have no blank symbols between words.

Recall that the key idea of the Boyer-Moore type algorithms is to skip symbols of text, so that they do not read all the text symbols on the average. The algorithms are intended to avoid 'redundunt' symbol comparisons. Analogously, our algorithm also skips symbols of text in the sense that more than one symbol is encoded as one character code. In other words, our algorithm avoids processing of redundant information about text. Note that the redundancy varies depending on the pattern in the case of the Boyer-Moore type algorithms, whereas it depends only on the text in the case of speeding up by compression.

The rest of the paper is organized as follows. In Section 2, we introduce the byte pair encoding scheme, discuss its implementation, and estimate its performance in comparison with `Compress` and `Gzip`. Section 3 is devoted to compressed pattern matching in BPE compressed files, where we have two implementations using the automata and the bit-parallel approaches. In Section 4, we report our experimental results to compare practical behaviors of these al-

gorithms performed. Section 5 concludes the discussion and explains some of future works.

2 Byte Pair Encoding

In this section we describe the byte pair encoding scheme, discuss its implementation, and then estimate the performance of this compression scheme in comparison with widely-known compression tools `Compress` and `Gzip`.

2.1 Compression Algorithm

The BPE compression is a simple version of pattern-substitution method [10]. It utilizes the character codes which did not appear in the text to represent frequently occurring strings, namely, strings of which frequencies are greater than some threshold. The compression algorithm repeats the following task until all character codes are used up or no frequent pairs appear in the text:

> *Find the most frequent pair of consecutive two character codes in the text, and then substitute an unused code for the occurrences of the pair.*

For example, suppose that the text to be compressed is

$$T_0 = \texttt{ABABCDEBDEFABDEABC}.$$

Since the most frequent pair is `AB`, we substitute a code `G` for `AB`, and obtain the new text

$$T_1 = \texttt{GGCDEBDEFGDEGC}.$$

Then the most frequent pair is `DE`, and we substitute a code `H` for it to obtain

$$T_2 = \texttt{GGCHBHFGHGC}.$$

By substituting a code `I` for `GC`, we obtain

$$T_3 = \texttt{GIHBHFGHI}.$$

The text length is shorten from $|T_0| = 18$ to $|T_3| = 9$. Instead we have to encode the substitution pairs $\texttt{AB} \rightarrow \texttt{G}$, $\texttt{DE} \rightarrow \texttt{H}$, and $\texttt{GC} \rightarrow \texttt{I}$.

More precisely, we encode a table which stores for every character code what it represents. Note that a character code can represent either (1) the character itself, (2) a code-pair, or (3) nothing. Let us call such table *substitution table*. In practical implementations, an original text file is split into a number of fixed-size blocks, and the compression algorithm is then applied to each block. Therefore a substitution table is encoded for each block.

2.2 Speeding Up of Compression

In [10] an implementation of BPE compression is presented, which seems quite simple. It requires $O(\ell N)$ time, where N is the original text length and ℓ is the number of character codes. The time complexity can be improved into $O(\ell + N)$ by using a relatively simple technique, but this improvement did not reduce the compression time in practice. Thus, we decided to reduce the compression time with sacrifices in the compression ratio.

The idea is to use a substitution table obtained from a small part of the text (e.g. the first block) for encoding the whole text. The disadvantage is that the compression ratio decreases when the frequency distribution of character pairs varies depending on parts of the text. The advantage is that a substitution table is encoded only once. This is a desirable property from a practical viewpoint of compressed pattern matching in the sense that we have to perform only once any task which depends on the substitution table as a preprocessing since it never changes.

Fast execution of the substitutions according to the table is achieved by an efficient multiple key replacement technique [2,23], in which a one-way sequential transducer is built from a given collection of replacement pairs which performs the task in only one pass through a text. When the keys have overlaps, it replaces the longest possible first occurring key. The running time is linear in the total length of the original and the substituted text.

2.3 Comparison with `Compress` and `Gzip`

We compared the performance of BPE compression with those of `Compress` and `Gzip`. We implemented the BPE compression algorithm both in the standard way described in [10] and in the modified way stated in Section 2.2. The `Compress` program has an option to specify in bits the upper bound to the number of strings in a dictionary, and we used `Compress` with specification of 12 bits and 16 bits. Thus we tested five compression programs.

We estimated the compression ratios of the five compression programs for the four texts shown in Table 1. The results are shown in Table 2. We can see that the compression ratios of BPE are worse than those of `Compress` and `Gzip`,

Table 1. Four Text Files.

file	annotation
Brown corpus (6.4 Mbyte)	A well-known collection of English sentences, which was compiled in the early 1960s at Brown University, USA.
Medline (60.3 Mbyte)	A clinically-oriented subset of Medline, consisting of 348,566 references.
Genbank1 (43.3 Mbyte)	A subset of the GenBank database, an annotated collection of all publicly available DNA sequences.
Genbank2 (17.1 Mbyte)	The file obtained by removing all fields other than accession number and nucleotide sequence from the above one.

especially for English texts. We also estimated the CPU times for compression and decompression. Although we omit here the results because of lack of space, we observed that the BPE compression was originally very slow, and it is drastically accelerated by the modification stated in Section 2.2. In fact, the original BPE compression is $4 \sim 5$ times slower than `Gzip`, whereas the modified one is $4 \sim 5$ times faster than `Gzip` and is competitive with `Compress` with 12 bit option.

Thus, BPE is not so good from the traditional criteria. This is the reason why it has received little attentions, until now. However, it has the following properties which are quite attractive from the practical viewpoint of compressed pattern matching: (1) No bit-wise operations are required since all the codes are of 8 bits; (2) Decompression requires very small amount of memory; and (3) Partial decompression is possible, that is, we can decompress any portion of compressed text.

In the next section, we will show how we can perform compressed pattern matching efficiently in the case of BPE compression.

Table 2. Compression Ratios (%).

	BPE		Compress		Gzip
	standard	modified	12bit	16bit	
Brown corpus (6.8Mb)	51.08	59.02	51.67	43.75	39.04
Medline (60.3Mb)	56.20	59.07	54.32	42.34	33.35
Genbank1 (43.3Mb)	46.79	51.36	43.73	32.55	24.84
Genbank2 (17.1Mb)	30.80	32.50	29.63	26.80	23.15

3 Pattern Matching in BPE Compressed Texts

For searching a compressed text, the most naive approach would be the one which applies any string matching routine with expanding the original text on the fly. Another approach is to encode a given pattern and apply any string matching routine in order to find the encoded pattern directly in the compressed text. The problem in this approach is that the encoded pattern is not unique. A solution due to Manber [20] was to devise a way to restrict the number of possible encodings for any string.

The approach we take here is basically an instance of the general compressed pattern matching algorithm for strings described in terms of collage system [16]. As stated in Introduction, collage system is a unifying framework that abstracts most of existing dictionary-based compression methods. In the framework, a string is described by a pair of a dictionary $\mathcal{D}$ and a sequence $\mathcal{S}$ of tokens representing phrases in $\mathcal{D}$. A dictionary $\mathcal{D}$ is a sequence of assignments where

the basic operations are concatenation, repetition, and prefix (suffix) truncation. A text compressed by BPE is described by a collage system with no truncation operations. For a collage system with no truncation, the general compressed pattern matching algorithm runs in $O(\|\mathcal{D}\|+|\mathcal{S}|+m^2+r)$ time using $O(\|\mathcal{D}\|+m^2)$ space, where $\|\mathcal{D}\|$ denotes the size of the dictionary $\mathcal{D}$ and $|\mathcal{S}|$ is the length of the sequence $\mathcal{S}$.

The basic idea of the general algorithm is to simulate the move of the KMP automaton for input $\mathcal{D}$ and $\mathcal{S}$. Note that one token of sequence $\mathcal{S}$ may represent a string of length more than one, which causes a series of state transitions. The idea is to substitute just one state transition for each such consecutive state transitions. More formally, let $\delta : Q \times \Sigma \to Q$ be the state transition function of the KMP automaton, where Σ is the alphabet and Q is the set of states. Extend δ into the function $\hat{\delta} : Q \times \Sigma^* \to Q$ by

$$\hat{\delta}(q, \varepsilon) = q \qquad \text{and} \qquad \hat{\delta}(q, ua) = \delta(\hat{\delta}(q, u), a),$$

where $q \in Q$, $u \in \Sigma^*$, and $a \in \Sigma$. Let D be the set of phrases in dictionary. Let *Jump* be the limitation of $\hat{\delta}$ to the domain $Q \times D$.

By identifying a token with the phrase it represents, we can define the new automaton which takes as input a sequence of tokens and makes state transition by using *Jump*. The state transition of the new machine caused by a token corresponds to the consecutive state transitions of the KMP automaton caused by the phrase represented by the token. Thus, we can simulate the state transitions of the KMP automaton by using the new machine. However, the KMP automaton may pass through the final state during the consecutive transitions. Hence the new machine should be a Mealy type sequential machine with output function $\mathit{Output} : Q \times D \to 2^N$ defined by

$$\mathit{Output}(q, u) = \{i \in N | 1 \leq i \leq |u| \text{ and } \hat{\delta}(q, u[1..i]) \text{ is the final state}\},$$

where N denotes the set of natural numbers, and $u[1..i]$ denotes the length i prefix of string u.

In [16] efficient realizations of the functions *Jump* and *Output* were discussed for general case. In the case of BPE compression, a simpler implementation is possible. We take two implementations. One is to realize the state transition function *Jump* defined on $Q \times D$ as a two-dimensional array of size $|Q| \times |D|$. The array size is not critical since the number of phrases in D is at most 256 in BPE compression. This is not the case with LZW, in which $|D|$ can be the compressed text size.

Another implementation is the one utilizing the bit parallel paradigm in a similar way that we did for LZW compression [17]. Technical details are omitted because of lack of space.

4 Experimental Results

We estimated the running time of the proposed algorithms running on BPE compressed files. We tested the two implementations mentioned in the previous section. For comparisons, we tested the algorithm [17] in searching LZW

compressed files. We also tested the KMP algorithm, the Shift-Or algorithm [26,3], and `Agrep` (the Boyer-Moore-Horpspool algorithm) in searching uncompressed files. The performance of the BM type algorithm strongly depends upon the pattern length m, and therefore the running time of `Agrep` was tested for $m = 4, 8, 16$. The performance of each algorithm other than `Agrep` is independent of the pattern length. The text files we used are the same as the four text files mentioned in Section 2. The machine used is a PC with a Pentium III processor at 500MHz running TurboLinux 4.0 operating system. The data transfer speed was about 7.7 Mbyte/sec.

The results are shown in Table 3, where we included the preprocessing time. In this table, (a) and (b) stand for the automata and the bit-parallel implementations stated in the previous section, respectively.

Table 3. Performance Comparisons.

		BPE		LZW	uncompressed				
		(a)	(b)	[17]	KMP	Shift-Or	`Agrep`		
							$m = 4$	$m = 8$	$m = 16$
CPU time (sec)	Brown Corpus	0.09	0.16	0.94	0.13	0.11	0.09	0.07	0.07
	Medline	1.03	1.43	6.98	1.48	1.28	0.85	0.69	0.63
	Genbank1	0.52	0.89	4.17	0.81	0.76	0.72	0.58	0.53
	Genbank2	0.13	0.22	1.33	0.32	0.29	0.27	0.32	0.32
elapsed time (sec)	Brown Corpus	0.59	0.54	1.17	0.91	1.01	0.91	0.90	0.90
	Medline	4.98	4.95	7.53	8.38	8.26	8.01	7.89	7.99
	Genbank1	3.04	2.95	4.48	6.26	6.32	6.08	5.67	5.64
	Genbank2	0.76	0.73	1.46	2.28	2.33	2.19	2.18	2.14

First of all, it is observed that, in CPU time comparison, the automata-based implementation of the proposed algorithm in searching BPE compressed file is faster than each of the routines except `Agrep`. Comparing with `Agrep`, it is good for Genbank1 and Genbank2, but not so for other two files. The reason for this is that the performance of the proposed algorithm depends on compression ratio. Recall that the compression ratios for Genbank1 and Genbank2 are relatively high in comparison with those of Brown corpus and Medline.

From a practical viewpoint, the running speed in elapsed time is also important, although it is not easy to measure accurate values of elapsed time. Table 3 implies that the proposed algorithm is the fastest in the elapsed time comparison.

5 Conclusion

We have shown potential advantages of BPE compression from a viewpoint of compressed pattern matching.

The number of tokens in BPE is limited to 256 so that all the tokens are encoded in 8 bits. The compression ratio can be improved if we raise the limitation to the number of tokens. A further improvement is possible by using variable-length codewords. However, it is preferable to use fixed-length codewords with 8 bits from the viewpoint of compressed pattern matching since we want to keep the search on a byte level for efficiency.

One future direction of this study will be to develop approximate pattern matching algorithms for BPE compressed text.

References

1. A. Apostolico and Z. Galil. *Pattern Matching Algorithm*. Oxford University Press, New York, 1997.
2. S. Arikawa and S. Shiraishi. Pattern matching machines for replacing several character strings. *Bulletin of Informatics and Cybernetics*, 21(1–2):101–111, 1984.
3. R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Comm. ACM*, 35(10):74–82, 1992.
4. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):62–72, 1977.
5. D. Breslauer. Saving comparisons in the Crochemore-Perrin string matching algorithm. In *Proc. of 1st European Symp. on Algorithms*, pages 61–72, 1993.
6. M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithm. *Algorithmica*, 12(4/5):247–267, 1994.
7. M. Crochemore and D. Perrin. Two-way string-matching. *J. ACM*, 38(3):651–675, 1991.
8. M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, New York, 1994.
9. E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Direct pattern matching on compressed text. In *Proc. 5th International Symp. on String Processing and Information Retrieval*, pages 90–95. IEEE Computer Society, 1998.
10. P. Gage. A new algorithm for data compression. *The C Users Journal*, 12(2), 1994.
11. Z. Galil and J. Seiferas. Time-space-optimal string matching. *J. Comput. System Sci.*, 26(3):280–294, 1983.
12. L. Gasieniec, W. Plandowski, and W. Rytter. Constant-space string matching with smaller number of comparisons: Sequential sampling. In *Proc. 6th Ann. Symp. on Combinatorial Pattern Matching*, pages 78–89. Springer-Verlag, 1995.
13. L. Gasieniec, W. Plandowski, and W. Rytter. The zooming method: a recursive approach to time-space efficient string-matching. *Theoret. Comput. Sci*, 147(1/2):19–30, 1995.
14. R. N. Horspool. Practical fast searching in strings. *Software-Practice and Experience*, 10:501–506, 1980.
15. G. C. Jewell. Text compaction for information retrieval. *IEEE SMC Newsletter*, 5, 1976.
16. T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. A unifying framework for compressed pattern matching. In *Proc. 6th International Symp. on String Processing and Information Retrieval*, pages 89–96. IEEE Computer Society, 1999.

17. T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Shift-And approach to pattern matching in LZW compressed text. In *Proc. 10th Ann. Symp. on Combinatorial Pattern Matching*, pages 1–13. Springer-Verlag, 1999.
18. T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In J. A. Atorer and M. Cohn, editors, *Proc. Data Compression Conference '98*, pages 103–112. IEEE Computer Society, 1998.
19. D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput*, 6(2):323–350, 1977.
20. U. Manber. A text compression scheme that allows fast searching directly in the compressed file. In *Proc. Combinatorial Pattern Matching*, volume 807 of *Lecture Notes in Computer Science*, pages 113–124. Springer-Verlag, 1994.
21. G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proc. 10th Ann. Symp. on Combinatorial Pattern Matching*, pages 14–36. Springer-Verlag, 1999.
22. D. M. Sunday. A very fast substring search algorithm. *Comm. ACM*, 33(8):132–142, 1990.
23. M. Takeda. An efficient multiple string replacing algorithm using patterns with pictures. *Advances in Software Science and Technology*, 2:131–151, 1990.
24. B. W. Watson and G. Zwaan. A taxonomy of sublinear multiple keyword pattern matching algorithms. *Sci. of Comput. Programing.*, 27(2):85–118, 1996.
25. S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Usenix Winter 1992 Technical Conference*, pages 153–162, 1992.
26. S. Wu and U. Manber. Fast text searching allowing errors. *Comm. ACM*, 35(10):83–91, October 1992.
27. A. C.-C. Yao. The complexity of pattern matching for a random string. *SIAM J. Comput.*, 8(3):368–387, 1979.