Compressed Pattern Matching for Sequitur

Mitarai, Shuichi
Department of Informatics, Kyushu University

Hirao, Masahiro
Department of Informatics, Kyushu University

Matsumoto, Tetsuya
Department of Informatics, Kyushu University

Shinohara, Ayumi
Department of Informatics, Kyushu University

他

出版情報：DOI Technical Report. 180, 2000-11-28. Department of Informatics, Kyushu University
バージョン：
権利関係：

DOI-TR-180

𝔻𝕆𝕀 Technical Report

Compressed Pattern Matching for SEQUITUR

by

Shuichi Mitarai,Masahiro Hirao,Tetsuya Matsumoto,Ayumi Shinohara,Masayuki Takeda,Setsuo Arikawa

November 28, 2000

Department of Informatics
Kyushu University
Fukuoka 812-81, Japan

Email: s-mita@i.kyushu-u.ac.jp Phone: +81-92-+81-92-642-2697

Compressed Pattern Matching for SEQUITUR

Shuichi Mitarai Masahiro Hirao Tetsuya Matsumoto
Ayumi Shinohara Masayuki Takeda Setsuo Arikawa

Department of Informatics, Kyushu University 33, Fukuoka 812-8581, Japan
{ s-mita, hirao, tetsuya, ayumi, takeda, arikawa } @i.kyushu-u.ac.jp
Phone number:+81-92-642-2697

Abstract

SEQUITUR due to Nevill-Manning and Witten. [18] is a powerful program to infer a phrase hierarchy from the input text, that also provides extremely effective compression of large quantities of semi-structured text [17]. In this paper, we address the problem of searching in SEQUITUR compressed text directly. We show a compressed pattern matching algorithm that finds a pattern in compressed text without explicit decompression. We show that our algorithm is approximately 1.27 times faster than a decompression followed by an ordinal search.

1 Introduction

Nevill-Manning and Witten [18] developed a powerful program SEQUITUR that infers a phrase hierarchy from the input text. It successfully extracts some comprehensible account of the structure of the input text, that are very useful for phrase browsing in digital libraries. At the same time, SEQUITUR is very attractive as a compression tool. The compression ratio is better than those of other dictionary-based compression methods [17, 19, 21].

Amir *et al.* [1] proposed an exciting problem related to data compression. It is referred to as *compressed pattern matching* problem, in which the aim is to find pattern occurrences in compressed text without decompression. Here, the performance of the pattern matching algorithm is measured with respect not to the size N of *original (uncompressed) text*, but to the size n of *compressed text*. They showed a compressed pattern matching algorithm for LZW compression that runs in $O(n + m^2)$, where m is the length of pattern. It implies that the compressed pattern matching algorithm for LZW compression will be asymptotically faster than the decompression followed by an ordinary pattern matching, since $n = \sqrt{N}$ at the best case. Their algorithm simulates the Knuth-Morris-Pratt (KMP) automaton [9] over a sequence of *phrases* instead of *characters*. The work stimulated a considerable amount of subsequent

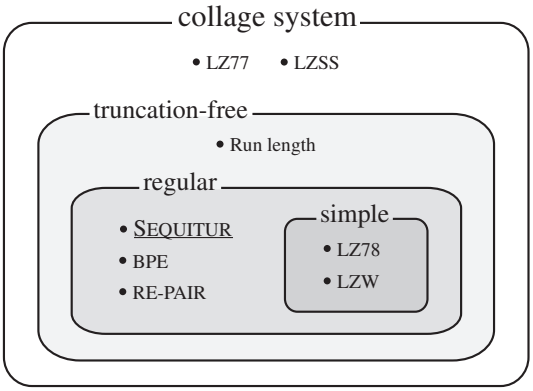

Figure 1: Hierarchy of collage system

theoretical studies on compressed pattern matching that runs in $o(N)$ time for various compression methods.

From a practical point of view, Kida *et al.* [8] showed the first experimental result that compressed pattern matching in LZW compressed files is indeed faster than a decompression followed by an ordinary search. They generalized the algorithm due to Amir *et al.* so that it can report *all* occurrences of *multiple* patterns for LZW compressed files. Followed this work, a good deal of practical effort has been made on compressed pattern matching for LZW [16], LZ77 [16], pattern substitution method [22], and so on.

From a theoretical viewpoint, on the other hand, Kida *et al.* [6] introduced a *collage system* as a unifying framework which abstracts various dictionary-based compression methods. Through the collage systems, many dictionary-based compression methods can be categorized into some classes (Fig. 1) and we can capture the essence of each compression method in the matter of compressed pattern matching. They also showed a general pattern matching algorithm for text strings in terms of collage system, that generalizes the algorithm due to Amir *et al.* considerably. Shibata *et al.* [23] showed a general pattern matching algorithm based on the Boyer-Moore (BM) [2] for text strings in terms of collage system. KMP and BM are known as the most important basic algorithms for searching in usual (uncompressed) texts.

The main purpose of this paper is to establish a compressed pattern matching algorithm for SEQUITUR and estimate its practical behavior. Since SEQUITUR is categorized as dictionary-based compression methods and we can treat it naturally as a collage system (Fig. 1), technically, we have only to adjust the general algorithm to SEQUITUR. Our experiments show that our algorithm is approximately 1.27 times faster than a decompression followed by an ordinary search. These results give us a strong evidence that the collage system is quite powerful and useful in compressed pattern matching.

We introduce some notation. Let Σ be a finite alphabet. An element of Σ^* is called a *string*. Strings x, y, and z are said to be a *prefix*, *factor*, and *suffix* of the string

$u = xyz$, respectively. A prefix, factor, and suffix of a string u is said to be *proper* if it is not u. The length of a string u is denoted by $|u|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. The ith symbol of a string u is denoted by $u[i]$ for $1 \leq i \leq |u|$, and the factor of a string u that begins at position i and ends at position j is denoted by $u[i:j]$ for $1 \leq i \leq j \leq |u|$. For convenience, let $u[i:j] = \varepsilon$ for $j < i$. Denote by u^R the reversed string of a string u. For a string u and a non-negative integer i, the string obtained by removing the length i prefix (resp. suffix) from u is denoted by ${}^{[i]}u$ (resp. $u^{[i]}$). That is, ${}^{[i]}u = u[i+1:|u|]$ and $u^{[i]} = u[1:|u|-i]$.

2 Sequitur

Sequitur is an algorithm that forms a grammar from a sequence of discrete symbols and it is also utilized for an effective compression model. The basic idea of Sequitur is to replace phrases which appear more than once by nonterminal symbols. Each repetition gives rise to a rule in the grammar. In the process of the algorithm, the following two properties should be hold in order to reduce the whole size of the grammar.

(p_1) : no pair of adjacent symbols appears more than once in the grammar.

(p_2) : every rule is used more than once.

Property p_1 assures the uniqueness of all rules in the grammar, while property p_2 ensures that each rule is useful. In [18], Nevill-Manning *et al.* proposed a linear time algorithm to infer rules from given text, by effectively using data structures based on doubly linked lists. For example, a text '`abcdbcabcd`' will be transformed into the following three rules, where S is the start symbol, and A and C are nonterminals.

$$\begin{aligned} S &\rightarrow CAC \\ A &\rightarrow \texttt{bc} \\ C &\rightarrow \texttt{a}A\texttt{d} \end{aligned}$$

You can easily recover the original text from these rules by substitutions.

As a compression tool, these rules have to be encoded into a sequence of bits compactly. At first, Sequitur converts a set of rules into a single sequence as follows. Started from the start symbol S, repeat the procedure recursively: when a new nonterminal appears, embed it with its definition. For example, the above set of rules will be the sequence

$$C(:= \texttt{a}A(:= \texttt{bc})\texttt{d})AC.$$

Then the sequence is encoded by using the arithmetic coder.

3 A unifying framework for compressed pattern matching

In a dictionary-based compression, a text string is described by a pair of a *dictionary* and a sequence of *tokens*, each of which represents a phrase defined in the dictionary. Kida *et al.* [6] introduced a unifying framework, named collage system, which abstracts various dictionary-based methods, such as the Lempel-Ziv family, SEQUITUR, RE-PAIR [10], and static dictionary methods. They presented a general compressed pattern matching algorithm for the framework, which is based on the simulation of the KMP automaton[6]. This implies that *any* compression method covered by the framework has a compressed pattern matching algorithm as an instance.

3.1 Collage system

A *collage system* is a pair $\langle \mathcal{D}, \mathcal{S} \rangle$ defined as follows: $\mathcal{D}$ is a sequence of assignments $X_1 = expr_1;\ X_2 = expr_2;\ \cdots\ ; X_\ell = expr_\ell$, where each X_k is a token (or a variable) and $expr_k$ is any of the forms:

a	for $a \in \Sigma \cup \{\varepsilon\}$,	(*primitive assignment*)
$X_i X_j$	for $i, j < k$,	(*concatenation*)
${}^{[j]}X_i$	for $i < k$ and an integer j,	(*prefix truncation*)
$X_i^{[j]}$	for $i < k$ and an integer j,	(*suffix truncation*)
$(X_i)^j$	for $i < k$ and an integer j.	(j *times repetition*)

Each token represents a string obtained by evaluating the expression as it implies. The strings represented by tokens are called *phrases*. As we want to distinguish a token from the phrase it represents, we denote by $X.u$ the phrase represented by a token X. The *size* of $\mathcal{D}$ is the number ℓ of assignments and denoted by $\|\mathcal{D}\|$. Define the *height* of a token X to be the height of the syntax tree whose root is X. The *height* of $\mathcal{D}$ is defined by $height(\mathcal{D}) = \max\{height(X) \mid X \text{ in } \mathcal{D}\}$. It expresses the maximum dependency of the tokens in $\mathcal{D}$.

On the other hand, $\mathcal{S} = X_{i_1}, X_{i_2}, \ldots, X_{i_n}$ is a sequence of tokens defined in $\mathcal{D}$. We denote by $|\mathcal{S}|$ the number n of tokens in $\mathcal{S}$. The collage system represents a string obtained by concatenating the phrases represented by $X_{i_1}, X_{i_2}, \ldots, X_{i_n}$.

According to the framework, text strings compressed by LZW, SEQUITUR, and RE-PAIR can be represented as follows. Notice that both of these collage systems are truncation-free.

LZW [24]. $\mathcal{S} = X_{i_1}, X_{i_2}, \ldots, X_{i_n}$ and $\mathcal{D}$ is as follows:

$$
\begin{aligned}
&X_1 = a_1; \quad X_2 = a_2; \quad \cdots; \quad X_q = a_q; \\
&X_{q+1} = X_{i_1} X_{\sigma(i_2)}; \quad X_{q+2} = X_{i_2} X_{\sigma(i_3)}; \quad \cdots; \\
&X_{q+n-1} = X_{i_{n-1}} X_{\sigma(i_n)},
\end{aligned}
$$

```
Input:   Pattern π and collage system consisting of D and S = S[1 : n].
Output:  All occurrences of π in the original text.
begin
/* Construction of Jump and Output */
    Construct Jump and Output from pattern π and dictionary D
/* Scanning S */
    state := 0;    ℓ := 0;
    for i := 1 to n do begin
        for each d ∈ Output(state, S[i]) do
            Report a pattern occurrence that ends at position ℓ + d;
        state := Jump(state, S[i]);   ℓ := ℓ + |S[i].u|
    end
end.
```

Figure 2: General algorithm for searching in a collage system.

where the alphabet is $\Sigma = \{a_1, \ldots, a_q\}$, $1 \leq i_1 \leq q$, and $\sigma(\ell)$ denotes the integer k, $1 \leq k \leq q$, such that a_k is the first symbol of the phrase $X_\ell.u$. $\mathcal{S}$ is encoded as a sequence of integers $i_1, i_2, \ldots, i_n$ in which an integer i_j is represented in $\lceil \log_2(q+j) \rceil$ bits, while $\mathcal{D}$ is not encoded since it can be obtained from $\mathcal{S}$.

SEQUITUR **[18] and** RE-PAIR **[10].** $\mathcal{S} = X_{i_1}, X_{i_2}, \ldots, X_{i_n}$, and $\mathcal{D}$ is as follows:

$$
\begin{array}{l}
X_1 = a_1; \quad X_2 = a_2; \quad \cdots; \quad X_q = a_q; \\
X_{q+1} = X_{\ell(1)}X_{r(1)}; \quad X_{q+2} = X_{\ell(2)}X_{r(2)}; \cdots; \\
X_{q+s} = X_{\ell(s)}X_{r(s)},
\end{array}
$$

where $\Sigma = \{a_1, \cdots, a_q\}$. $\mathcal{D}$ and $\mathcal{S}$ are encoded using some appropriate encoding. Concerning with the previous example, the rule $S \rightarrow CAC$ directly corresponds to $\mathcal{S}$, and the other rules correspond to $\mathcal{D}$. Remark that $\mathcal{S}$ is clearly separated from $\mathcal{D}$ here, while they are converted into an interleaved sequence when it is encoded.

3.2 Pattern matching in collage systems

Our problem is defined as follows.

> Given a pattern $\pi = \pi[1 : m]$ and a collage system $\langle \mathcal{D}, \mathcal{S} \rangle$ with $\mathcal{S} = \mathcal{S}[1 : n]$, find all locations at which π occurs within the original text $\mathcal{S}[1].u \cdot \mathcal{S}[2].u \cdots \mathcal{S}[n].u$.

Figure 2 gives an overview of the algorithm due to Kida *et al.* [6], which processes $\mathcal{S}$ token-by-token. The algorithm simulates the move of the KMP automaton running on the original text, by using two functions *Jump* and *Output*, both take as input a state and a token. The former is used to substitute just one state transition for the consecutive state transitions of the KMP automaton caused by each of the phrases, and

the latter is used to report all pattern occurrences found during the state transitions. Thus the definitions of the two functions are as follows.

$$\begin{aligned} \mathit{Jump}(j,t) &= \delta(j,t.u), \\ \mathit{Output}(j,t) &= \left\{ |v| \;\middle|\; \begin{array}{l} v \text{ is a non-empty prefix of } t.u \\ \text{such that } \delta(j,v) \text{ is the final state} \end{array} \right\}, \end{aligned}$$

where δ is the state transition function of the KMP automaton. This idea is a generalization of the algorithm due to Amir et al. [1], which was restricted to LZW compressed texts.

Theorem 1 (Kida et al. [6]) *In the algorithm in Fig. 2, Jump and Output can be constructed in $O(\mathit{height}(\mathcal{D}) \cdot \|\mathcal{D}\| + m^2)$ time using $O(\|\mathcal{D}\| + m^2)$ space. The scanning part consumes $O(\mathit{height}(\mathcal{D}) \cdot n + r)$ time, where r is the number of pattern occurrences. The factor $\mathit{height}(\mathcal{D})$ can be dropped if the dictionary $\mathcal{D}$ is truncation-free.*

The above theorem implies that if the dictionary $\mathcal{D}$ is truncation-free, the total running time is linear with respect to the size of $\langle \mathcal{D}, \mathcal{S} \rangle$, the compressed text. As we have shown, the collage systems for SEQUITUR and LZW are truncation-free. On the other hand, the collage systems for LZ77 and LZSS compression requires truncations [8]. These facts correspond with the observations that LZW is suitable for compressed pattern matching, while LZSS is not [7, 8, 16]. As we will show below, SEQUITUR is also suitable for compressed pattern matching.

3.3 Searching in SEQUITUR compressed files

We briefly state on the modification of the general pattern matching algorithm that are specific to SEQUITUR. In SEQUITUR, the encoding of the dictionary $\mathcal{D}$ is interleaved with that of $\mathcal{S}$, as we explained in Section 2. Therefore, we have to extract $\mathcal{D}$ incrementally in the token-by-token processing of $\mathcal{S}$, and the constructions of *Jump* and *Output* are merged into the scanning part (see Fig. 2 again).

4 Experimental results

In this section, we discuss the compression ratio, compression/decompression time and searching time. Our experiments were carried out on an AlphaStation XP1000 with an Alpha21264 processor at 667MHz running Tru64 UNIX operating system V4.0F and used the following texts:

- **Brown corpus**: A well-known collection of English sentences, which was compiled in the early 1960s at Brown university, USA. The file size is about 6.8 Mbyte.

- **Genbank**: A subset of the Genbank database, an annotated collection of all publicly available DNA sequences. The file size is about 17Mbyte.

First of all, we estimated the compression ratio and the compression and decompression time of SEQUITUR in comparison with well-known compression tools **Compress** and **Gzip**. The results are shown in Table 1.

We verified that the compression ratio of SEQUITUR outperforms Gzip as well as Compress. On the other hand, however, the compression and decompression are very slow compared to Gzip and Compress, because SEQUITUR utilizes the arithmetic coding that is time consuming, and the program might not be fully optimized. From our view point of compressed pattern matching, compression time is not a serious matter, while the decompression time is critical. In the original program of SEQUITUR, decompression routine borrows the same data structures, such as doubly linked list, that are unnecessary for decompression only. Thus we simply rewrote the decompression routine using a standard array. The improved time of decompression is shown in the last rows in Table 1.

We now consider the performance of our compressed pattern matching algorithm. We compared the running time of the following three methods:

(A) *Original decompression followed by KMP matching.*

We did not combine the decompression and the search programs using the UNIX 'pipe' because this is slow. Instead, we incorporated the KMP algorithm into these decompression programs, so that the KMP automaton processes the decoded characters directly.

(B) *Improved decompression followed by KMP matching.*

The same as (A) except that the decompression routine is improved as we stated.

(C) *Compressed pattern matching.*

Table 1: Performance of SEQUITUR compared with Compress and Gzip. The compression/decompression times are CPU times in sec.

		SEQUITUR	Compress	Gzip	Gzip -9
Brown Corpus (6.83Mbyte)	compression ratio	0.34	0.44	0.39	0.39
	comp. time	35.93	1.02	3.54	5.07
	decomp. time	8.93	0.61	0.33	0.33
	improved decomp. time	7.45	—	—	—
Genbank (17.11Mbyte)	compression ratio	0.21	0.27	0.23	0.22
	comp. time	197.14	1.91	13.79	62.03
	decomp. time	17.47	1.30	0.57	0.55
	improved decomp. time	14.87	—	—	—

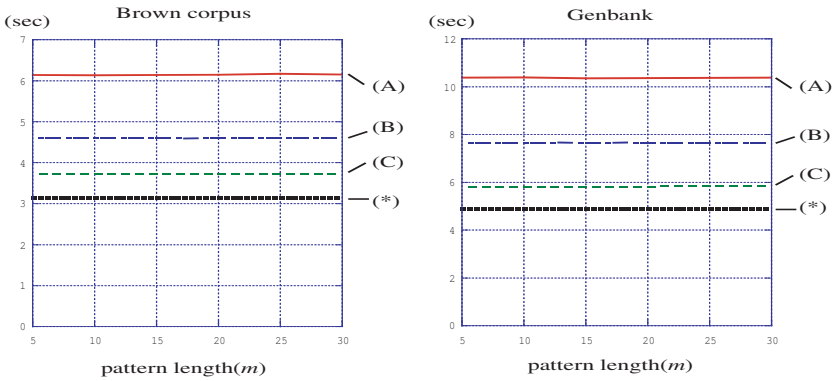

Figure 3: Searching time, where (A): decompression followed by KMP matching, (B): improved decompression followed by KMP matching, and (C): compressed pattern matching. We show the time to decode by the arithmetic coder as (*).

> We implemented the algorithm presented in the previous section in searching SEQUITUR compressed text directly. Here, we do not explicitly recover the original text file.

We performed the experiments as follows. The length of target pattern varied from 5 to 30. For each length, we randomly chose 10 substrings from the original text as target patterns. For each pattern, we ran these programs 10 times, and we calculated the average running time (CPU time). Fig. 3 shows the results. We can observe that (C) is faster than (B) as well as (A). In fact the running time of (C) is even reduced to about 25% of (B), regardless the pattern length.

Moreover, we suspected that the most expensive part in decompression would be the arithmetic coder routine. In order to estimate the time consumed this routine, we constructed a skeleton program from the decompression program, that consists of only the lines related to the arithmetic coding. We plotted the estimated time as (*) in Fig 3. It turned out that the arithmetic coder wastes 84% of compressed pattern matching (C). It should be noticed that the routine is common to all of (A), (B), and (C). If the decoder runs faster, the running time of (C) relative to (B) and (A) will be increased. One choice is to use Huffman coding instead of arithmetic coding since it is faster, while the compression ratio will be worse.

5 Conclusion

We showed a KMP type algorithm for compressed pattern matching for SEQUITUR compressed texts, and it is about 1.27 times faster than a decompression followed by KMP matching. The results make SEQUITUR more attractive as a compression tool. Through a series of our experiments, we have realized some room for further improvement in the current implementation of SEQUITUR in order to enable faster compressed matching.

- Dictionary $\mathcal{D}$ should be encoded separately from the sequential part $\mathcal{S}$. More-

over, since the dependency of all rules produced by SEQUITUR forms a directed acyclic graph, we should encode them in the topological order so that each rule is defined before it appears at other rules. For example, when encoding the rules $\{S \to CAC,\ A \to \mathtt{bc},\ C \to \mathtt{a}A\mathtt{d}\}$, we should encode A, C, and then S. Then the construction of *Jump* and *Output* is clearly separated from scanning part in our algorithm, and we can avoid recursions when reconstructing the rules.

- For faster compressed pattern matching, the dictionary part should not be too large, since the size of automaton depends on the dictionary. One extreme example is Byte Pair Encoding methods [4], where the number of rules is limited to 256, that is verified to be quite suitable for compressed pattern matching [22, 23]. In this sense, the approach to limit the memory requirement in [20] will be promising.

- Arithmetic coder may not be appropriate for our purpose, since it wastes most of the decompression time.

Unfortunately, the above discussion is beyond the scope of collage systems, in which we have ignored in the process of abstraction. Nothing to say, however, the practical behavior heavily depends on these constant factors. We hope to establish another better framework to capture these factors.

We finally note some trends on the compressed pattern matching shortly. A more ambitious goal (Goal 2) is to perform a faster search in compressed files *in comparison with an ordinary search in the original files*. In this case, the aim of compression is not only to reduce disk storage requirement but also to speed up string searching task. For some compression methods, in fact this goal has been achieved [3, 11, 14, 22, 23]. Moreover, approximate string matching is also aimed for LZ78/LZW format [5, 12], although practically even the first goal has not established yet. In their recent work [15], however, Navarro *et al.* took a practical approach, which reduces the problem to multipattern searching of pattern pieces plus local decompression and direct verification of candidate text areas, and showed experimentally that this solution achieves the first goal for moderate error level. Since the general pattern matching algorithm used in this paper can be extended to multipattern searching for regular collage systems [13], and SEQUITUR belongs to this class, we are sure that approximate string matching for SEQUITUR is also efficiently possible.

Acknowledgements

The authors would like to thank Ian H. Witten and Craig G. Nevill-Manning for fruitful discussions and helpful suggestions.

References

[1] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52:299–307, 1996.

[2] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. ACM*, 20(10):62–72, 1977.

[3] S. Fukamachi, T. Shinohara, and M. Takeda. String pattern matching for compressed data using variable length codes. In *Proc. Symposium on Informatics 1992*, pages 95–103, 1992. (in Japanese).

[4] P. Gage. A new algorithm for data compression. *The C Users Journal*, 12(2), 1994.

[5] J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching over Ziv-Lempel compressed text. In *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching*, pages 195–209. Springer-Verlag, 2000.

[6] 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.

[7] 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.

[8] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In *Proc. Data Compression Conference (DCC'98)*, pages 103–112. IEEE Computer Society, 1998.

[9] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[10] N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proc. Data Compression Conference (DCC'99)*, pages 296–305. IEEE Computer Society, 1999.

[11] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM Trans. Information Systems*, 15(2):124–136, 1997. Previous version in: *CPM'94*.

[12] T. Matsumoto, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Bit-parallel approach to approximate string matching in compressed texts. In *Proc. 7th International Symp. on String Processing and Information Retrieval*, pages 221–228. IEEE Computer Society, 2000.

[13] T. Matsumoto, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. General multiple pattern matching algorithms over compressed text. Manuscript, 2000.

[14] M. Miyazaki, S. Fukamachi, M. Takeda, and T. Shinohara. Speeding up the pattern matching machine for compressed texts. *Transactions of Information Processing Society of Japan*, 39(9):2638–2648, 1998. (in Japanese).

[15] G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Faster approximate string matching over compressed text. Manuscript.

[16] 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.

[17] C. Nevill-Manning and I. Witten. Compression and explanation using hierarchical grammars". *Computer Journal*, 40(2/3):103–116, 1997.

[18] C. Nevill-Manning and I. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research*, 7:67–82, 1997.

[19] C. Nevill-Manning and I. Witten. Linear-time, incremental hierarchy inference for compression. In *Proc. Data Compression Conference (DCC'97)*. IEEE Press, 1997.

[20] C. Nevill-Manning and I. Witten. Phrase hierarchy inference and compression in bounded space. In *Proc. Data Compression Conference (DCC'98)*, pages 179–187. IEEE Press, 1998.

[21] C. G. Nevill-Manning, I. H. Witten, and D. R. Olsen. Compressing semi-structured text using hierarchical phrase identification. In *Proc. Data Compression Conference (DCC'96)*. IEEE Press, 1996.

[22] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa. Speeding up pattern matching by text compression. In *Proc. 4th Italian Conference on Algorithms and Complexity*, pages 306–315. Springer-Verlag, 2000.

[23] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, and S. Arikawa. A Boyer-Moore type algorithm for compressed pattern matching. In *Proc. 11th Annual Symposium on Combinatorial Pattern Matching (LNCS 1848)*, pages 181–194. Springer-Verlag, 2000.

[24] T. A. Welch. A technique for high performance data compression. *IEEE Comput.*, 17:8–19, June 1984.