Searching in Compressed Dictionaries

EXTENDED ABSTRACT

Shmuel T. Klein
Department of Computer Science
Bar Ilan University
Ramat Gan, Israel
tomi@cs.biu.ac.il

Dana Shapira
Department of Computer Science
Brandeis University
Waltham, MA
shapird@cs.brandeis.edu

1. Introduction

The problem of *Compressed Pattern Matching*, introduced by Amir and Benson [1], is of performing pattern matching directly in a compressed text without any decompressing. More formally, for a given text T, pattern P and complementary encoding and decoding functions $\mathcal{E}$ and $\mathcal{D}$, respectively, our aim is to search for $\mathcal{E}(P)$ in $\mathcal{E}(T)$, rather than the usual approach which searches for the pattern P in the decompressed text $\mathcal{D}(\mathcal{E}(T))$.

Most research efforts in compressed matching were invested in what could be called "classical" texts. These are texts written generally in some natural language, and which have been compressed by one of a variety of known compression techniques, such as Huffman coding [8] or various variants of the Lempel and Ziv methods, including LZW [2, 4, 11], gzip, DoubleSpace and many others [7, 9, 10].

We suggest to extend the problem to the search of patterns in the compressed form of *structured files*. The idea is that the raw texts form only a (sometimes, small) part of what needs to be stored in an Information Retrieval system to allow also efficient access to the data. Since the search on large scale systems is not performed by a linear scan, auxiliary files are adjoined, which are generally built in a preprocessing stage, but then permit very fast access during the production stage. These files include dictionaries, concordances, thesauri, bitmaps, signature files, grammatical files and many others, and their combined sizes are of the order of magnitude of the text they are based on. Obviously, these auxiliary files create a storage problem on their own, and thus are kept in compressed form. However, due to their special internal structure, which is known in advance, custom tailored compression techniques may be more effective than general purpose compressors.

In this paper we deal with dictionaries which were compressed by the *prefix omission method* (POM) so that pattern matching could be done directly on these files. Prefix omission is a very simple, yet effective, dictionary compression technique, but for compressed matching, it raises problems that are reminiscent of the problems for a compressed match in LZ coded files: the pattern we are looking for, if it appears in the text, does not necessarily appear there contiguously [8].

The following section recalls the details of POM and presents an algorithm for searching in POM encoded files. Section 3 deals more specifically with Huffman coding and in section 4, a new variant based on the use of Fibonacci codes is suggested. The final section brings some experimental results.

2. Pattern matching in POM encoded dictionaries

The prefix omission method was apparently first mentioned by Bratley and Choueka [3]. It is based on the observation that in a dictionary of a natural language text, two consecutive entries will usually have a few leading letters in common. Therefore we eliminate these letters from the second entry, while adding to it the number of letters eliminated and to be copied from previous entry. Since the entries have now variable length, their boundaries have to be identified, for example by adding a field representing the number of characters of the suffix string in the current entry, that is, the number of characters remaining after eliminating the prefix characters. More formally, we relate to the i-th compressed entry X_i as an ordered triple (ℓ_i,n_i,σ_i), where ℓ_i is the number of characters copied from the previous entry X_{i-1}, σ_i is the remaining suffix and n_i is the length of this suffix, i.e., $n_i = |\sigma_i|$.

dictionary	POM	Pom+Huff (in characters)	idem (in bits)
`compress`	(0,8,`compress`)	(0,35,11001-0101-11011-111100-1001-000-1000-1000)	(0,35,$\cdots$)
`compression`	(8,3,`ion`)	(35,12,0110-0101-0111)	(35,12,$\cdots$)
`comprise`	(5,3,`ise`)	(24,11,0110-1000-000)	(25,10,110 $\cdots$)
`compromise`	(5,5,`omise`)	(24,20,0101-11011-0110-1000-000)	(26,18,01 $\cdots$)
`compulsion`	(4,6,`ulsion`)	(20,26,11000-10110-1000-0110-0101-0111)	(21,25,1000 $\cdots$)
`compulsive`	(8,2,`ve`)	(38,9,111110-000)	(38,9,$\cdots$)
`compulsory`	(7,3,`ory`)	(34,14,0101-1001-111010)	(36,12,01 $\cdots$)
`compunction`	(5,6,`nction`)	(25,24,0111-11001-001-0110-0101-0111)	(25,24,$\cdots$)
`computation`	(5,6,`tation`)	(25,22,001-0100-001-0110-0101-0111)	(26,21,01$\cdots$)
`compute`	(6,1,`e`)	(28,3,000)	(29,2,00)
`computer`	(7,1,`r`)	(31,4,1001)	(31,4,1001)

FIGURE 1: *Example of the prefix omission method*

Consider the example given in Figure 1. The first column is a list of some consecutive words which were taken from the Oxford Dictionary of current English. The following column gives the compressed form of these words using the prefix omission method.

Often the POM files are further compressed by some general method $\mathcal{E}$, such as `gzip`, in order to reduce space. Accessing the dictionary itself is then done in two stages: first decompressing the file T by the corresponding decompression function $\mathcal{D}$ and then reversing the POM file, or more formally, searching for the pattern P in $\text{POM}^{-1}(\mathcal{D}(T))$. The following algorithm, adapted from [3], is more direct.

Let $P = p_1 \cdots p_m$ denote the pattern of length m to be searched for, and $P[i,j]$ the sub-pattern $p_i \cdots p_j$, i.e, $P[i,j]$ is the sub-pattern of P, starting at position i and ending at position j, both included. Let $\mathcal{E}$ and $\mathcal{D}$ denote two complementary encoding and decoding functions. Given two strings $S = s_1 \cdots s_k$ and $T = t_1 \cdots t_\ell$ the function $pre(S,T)$ returns the length of the longest common prefix of the two strings (or zero, if this prefix is empty), that is

$$s_i = t_i \ \text{ for } \ 1 \le i \le pre(S,T), \qquad \text{and} \qquad s_j \ne t_j \ \text{ for } \ j = pre(S,T) + 1.$$

In particular, $pre(S,T) = |S|$ if S is a prefix of T. Denote by $\succ$ the lexicographic order relation, i.e., $S \succ T$ if the string S follows T in lexicographic order.

```
1       $i \longleftarrow 2$;   $j \longleftarrow pre(P, \mathcal{D}(\sigma_1))$;
2       **while** $(j < m)$    // Pattern not found
        {
2.1         **while**   $\mathcal{D}(\ell_i) > j$
2.1.1           $i \longleftarrow i + 1$
2.2         **if**   $\mathcal{D}(\ell_i) < j$    // The closest lexicographically preceding word
2.2.1           return $i - 1$
2.3         **else**   // $\mathcal{D}(\ell_i) = j$
            {
2.3.1           $tmp \longleftarrow pre(P[j+1,m], \mathcal{D}(\sigma_i))$
2.3.2           **if** $tmp = 0$ **and** $P[j+1,m] \succ \mathcal{D}(\sigma_i)$    return $i - 1$
2.3.3           $j \longleftarrow j + tmp$
2.3.4           $i \longleftarrow i + 1$
            }
        }
3       return $i - 1$
```

FIGURE 2: *Searching for P in $\mathcal{D}(T)$*

The algorithm for searching in compressed POM dictionaries, based on decompressing each entry, is given in Figure 2. It returns the index of the closest lexicographically preceding word to the word we are searching for. As long as the component ℓ_i, indicating the number of characters copied from the previous entry, is larger than the current longest match, we simply move to the following entry (line 2.1) by skipping over $\mathcal{D}(\sigma_i)$. This is done by decoding the following n_i codewords. The correctness here is based on the fact that in this case there is at least one character following the characters of the longest match that is not part of an extended match. If the component ℓ_i is less than the current longest match, we have already found the closest lexicographically preceding word in the previous entry, and we return its index (line 2.2). When ℓ_i is exactly equal to the current length of the longest match, we try to extend the match to the following characters (line 2.3).

Line 2.3.2 deals with the special case when several consecutive words share a common prefix. Relying here only on ℓ_i without lexicographically comparing the suffixes could yield errors, as can be seen in the following example. If the sequence of dictionary entries is {aba, abb, abd, abe, aca} and we are looking for abc, the algorithm without line 2.3.2 would return abe instead of abb.

Note that it might be that the three fields of each POM entry are encoded in different ways. This would then imply that instead of using one decoding function $\mathcal{D}$, we use several different ones, e.g., $\mathcal{D}_1$ in lines 2.1, 2.2 and 2.3, $\mathcal{D}_2$ in lines 1 and 2.3.1 and $\mathcal{D}_3$ for n_i.

3. Combining POM with Huffman coding

To perform the pattern matching directly in the Huffman compressed dictionary, we need to identify the codeword boundaries in order to skip to the beginning of the following dictionary entry by counting the number of characters left in the current entry. If the field n_i represents the number of *codewords* to the following entry, we have to decode each one to know where the next one starts. By using Skeleton trees [6], we could skip over a part of the bits, to the beginning of the following codeword, but still each codeword has to be processed on its own. However, defining n_i as the number of *bits* to the following entry, provides a way to jump directly to the beginning of the following entry, without any processing of the bits. But this way we increase the storage requirements, since larger numbers need be stored.

The third column of Figure 1 is an example of the dictionary obtained by using a Huffman code based on empirical statistics. Note that ℓ_i and n_i are now given in bits, but their values still refer to the lengths of one or more whole codewords. In the last column of Figure 1, the definition of ℓ_i is extended to be the maximal number of *bits* copied from the previous entry, regardless of codeword boundaries. Though the number of copied bits is only occasionally increased and only by a small number of bits, the extension frees the function $pre(S, T)$ of the need of checking for codewords. One can thus apply *pre* on bitstrings regardless of their interpretation as codewords, which can be done efficiently with a few assembly commands.

There is, however, a drawback when moving to perform the pattern matching directly on Huffman encoded dictionaries. In the algorithm of Figure 2, when the pattern word does not appear in the dictionary, we are able to locate the closest lexicographically preceding word, basing ourselves on the lexicographic order of the dictionary entries. The problem here stems from the fact that Huffman coding does not necessarily preserve the lexicographic order. Even if canonical Huffman codes are used which do preserve such order in its consecutive codewords, this order does not always coincide with the lexicographic order of the represented characters. For example, refer to the alphabet {t, c, b, a, q} encoded by the canonical code {00, 01, 10, 110, 111}. The the string qt precedes tq, but for their encodings, 11100 follows 00111. We can therefore only either locate the pattern, or announce a mismatch.

The compressed matching algorithm in POM files which were compressed by using

```
1.1    $i \longleftarrow 2$;   $j \longleftarrow pre(\mathcal{E}(P), \sigma_1)$;
1.2    $security \longleftarrow \max_{c \in P}\{|\mathcal{E}(c)|\}$
2      while $(j < |\mathcal{E}(P)|)$   // Pattern not found
       {
2.1        while   $\mathcal{D}(\ell_i) > j$
2.1.1          skip $n_i$ bits to the following entry
2.1.2          $i \longleftarrow i + 1$
2.2        if   $\mathcal{D}(\ell_i) + security < j$   return FALSE
2.3        else   // $j - security \le \mathcal{D}(\ell_i) \le j$
           {
2.3.1          $tmp \longleftarrow pre(\mathcal{E}(P)[\mathcal{D}(\ell_i) + 1, |\mathcal{E}(P)|], \sigma_i)$
2.3.2          $j \longleftarrow \mathcal{D}(\ell_i) + tmp$
2.3.3          skip $n_i - tmp$ bits to the following entry
2.3.4          $i \longleftarrow i + 1$
           }
       }
3      return $i - 1$
```

Figure 3: *Searching for $\mathcal{E}(P)$ in T for Huffman coding*

Huffman coding is given in Figure 3, with *pre* now working on bit strings. Note that instead of decompressing the σ_i components, as done in the previous approach, we compress the pattern P and refer to bits instead of characters.

i	dictionary entry	POM (ℓ_i, n_i, σ_i)	POM & Huffman ℓ_i, n_i refer to bits
1	abc	(0, 3, abc)	(0, 7, 110-10-01)
2	abqt	(2, 2, qt)	(5, 5, 111-00)
3	abtq	(2, 2, tq)	(5, 5, 00-111)

Figure 4: *Example for the need of a security number*

An additional complication for this variant is the need for a *security* number to assure correctness. In the algorithm of Figure 2, the closest lexicographically preceding word is found once ℓ_i is smaller than the longest common prefix we have already detected. Here, to guarantee that the word does really not appear, the condition has to be reinforced and we check that $\mathcal{D}(\ell_i) + security$ is still less than j. To illustrate the need for that change, refer to the above mentioned canonical Huffman code and the dictionary of Figure 4. Suppose we are searching for the pattern abtq, the encoded form of which is 110-10-00-111. Performing line 1 of Figure 3 we get that $j = 6$. As $j < |\mathcal{E}(P)| = 10$, we perform line 2. But as $\mathcal{D}(\ell_2) = 5 < j$, we would return FALSE, which is wrong. The security number gives us a security margin, forcing a closer analysis in the else clause.

If we detect an entry the ℓ_i component of which is less than the current longest match, we can be sure the word we are looking for is missing only if the difference is more than the length of the encoding of one character. Therefore, the *security* number could be chosen as the maximum number of bits which are used to encode the characters of the alphabet, i.e, *security*= $\max\limits_{c\in\Sigma}\{|\mathcal{E}(c)|\}$. As we deal only with the characters of the pattern, we can choose the security number to be the maximum number of bits needed to encode one of the characters of P, i.e, *security*= $\max\limits_{c\in P}\{|\mathcal{E}(c)|\}$.

4. Combining POM with Fibonacci coding

In the previous section we used Huffman codes in order to perform compressed pattern matching on POM files. This way we could skip to the following entry by counting the bits with no need of decompressing the σ_i coordinates. We still had to decompress the ℓ_i components for arithmetic comparison. In this section we present a pattern matching algorithm working on a POM file which has been compressed using a binary *Fibonacci code*. This is a universal variable length encoding of the integers based on the Fibonacci sequence rather than on powers of 2, and a subset of these encodings can be used as a fixed alternative to Huffman codes, giving obviously less compression, but adding simplicity (there is no need to generate a new code every time), robustness and speed [5]. The particular property of the Fibonacci encoding is that there are no adjacent 1's, so that the string 11 can act like a *comma* between codewords, yielding the following sequence: $\{11, 011, 0011, 1011, 00011, 10011, 01011, 000011, \ldots\}$.

In our case, we wish to encode dictionary entries, each consisting of several codewords. We know already how to parse an encoded string into its constituting codewords, what still is needed is a separator between adjacent dictionary entries. At first sight it seems that just an additional 1-bit would be enough, since the pattern 111 never appears within a codeword. However, a sequence of 3 consecutive ones can appear *between* adjacent codewords, as in 011-1011. Therefore we must add *two* 1-bits as separators between dictionary entries. The additional expense is alleviated by the fact that the n_i component becomes redundant and can be omitted.

There is, however, a problem with the first codeword 11, which is exceptional, being the only one which does not have the suffix 011. Our goal is to be able to jump to the beginning of the following dictionary entry without having to decode the current one completely. If the first codeword 11 were to be omitted, one could then simply search for the next occurrence of the string 01111, but if 11 is permitted, a sequence of 1's of any length could appear, so no separator would be possible. Our first solution is thus simply omitting 11 from the Fibonacci code, which comes at the price of adding one bit to each codeword which is the last one of a block of codewords of the same length.

Another solution is using the first codeword 11, but making sure that two such codewords cannot appear adjacently. This can be achieved by adding a new codeword for encoding the sequence of two occurrences of the most popular character. For Example, if `e` is the most popular character in a given text file, we use the codeword `11` to encode a single occurrence of `e`. But if the sequence `ee` occurs in the text, it will be encoded by a special codeword (taking the probability occurrence of `ee` into account). In other words, if Σ denotes the alphabet, the new alphabet to be encoded by the Fibonacci code is $\Sigma \cup \{\mathsf{ee}\}$. If, e.g., the string eeeee occurs, we can use the special codeword twice and follow it by 11, the codeword for e. The longest sequence of 1-bits would thus consist of 5 1's, as in 10**11-11-1**011. Therefore, to identify a new entry in the POM file, a sequence of *six* 1-bits is needed, that is, our separator consists of four 1-bits, rather than just two in the previous solution. Comparisons between the compression performance of these two solutions are given in the following section, showing, at least on our data, that the first solution (omission of 11) is preferable to the second. The rest of our discussion therefore assumes this setting.

```
1       $i \longleftarrow 2$;   $j \longleftarrow$ $\textit{fib-pre}(\mathcal{E}(P), \sigma_1)$;
2       while $(j < m)$    // Pattern not found
        {
2.1         while   $\ell_i > Fib(j)$
2.1.1           skip to the following occurrence of the string '11110'
2.1.2           $i \longleftarrow i + 1$
2.2         if   $\ell_i < Fib(j)$    return FALSE
2.3         else    // $\ell_i = Fib(j)$
            {
2.3.1           $tmp \longleftarrow \textit{fib-pre}(\mathcal{E}(P[j+1, m]), \sigma_i)$
2.3.2           $j \longleftarrow j + tmp$
2.3.3           skip to the following occurrence of the string '11110'
2.3.4           $i \longleftarrow i + 1$
            }
        }
3       return $i - 1$
```

FIGURE 5: *Searching for $\mathcal{E}(P)$ in T for Fibonacci coding*

The reason for defining the codewords with the string 11 at their end is to obtain a prefix code, which is instantaneously decodable. If we add the 11 separator between dictionary entries at the end of the ℓ_i field, the appearance of the sequence 01111 can tell us that we have just read the ℓ_i part of the following entry. If we reverse the order of the bits in all the codewords, the string 11110 will physically separate two consecutive entries. Moreover, the codewords are then in numerical order, i.e., if $i > j$, then the Fibonacci encoding of i, when regarded as a number represented in the standard binary encoding, will be larger than the corresponding encoding of j. The compressed search in a dictionary using both POM and Fibonacci coding is given in Figure 5, where $Fib(i)$ stands for the above Fibonacci representation of the integer i. There is no need in decompressing the Fibonacci encoded field ℓ_i, so that the comparisons in lines 2.1 and 2.3 can be done directly with the encoded binary strings.

Given the pattern to be searched for, we can compute, as before, the longest common prefix of σ_i and $\mathcal{E}(P)$. However, it might be that this common prefix is *not* the encoding of the longest common prefix of $\mathcal{D}(\sigma_i)$ and P. For example, if $\mathcal{E}(P) =$ 1100-1101 and $\sigma_1 =$ 1100-110101, then the longest common prefix in characters is of length 1, (i.e. the decoding of 1100), but the longest common prefix in bits is the binary string 1100-1101, which could be wrongly interpreted as *two* codewords. This can be corrected by checking whether the string which follows the longest common binary prefix in both $\mathcal{E}(P)$ and σ_i is at the begining of a codeword, i.e., starts with 11. The function *fib-pre* in Figure 5 refers to this corrected version; though getting binary strings as parameters, it returns the number of codewords, rather than the number of bits, in the common prefix.

5. Experimental results

The experiments were performed on small POM files of several K bytes because of the following particular application: POM is often used to store dictionaries in B-trees; since the B-tree structure supports an efficient access to memory pages, each node is limited to a page size, and each page has to be compressed on its own, that is, for the first entry of each page, $\ell_1 = 0$.

File	size	Huffman (bit)	Fibonacci	Huffman (char)	POM
bib1	2044	775	716	616	1171
bib2	4095	1709	1666	1413	2754
bib3	8067	2769	2749	2253	4663
bib4	16199	5242	5379	4276	9217
xml1	2047	1097	999	905	1481
xml2	4093	1640	1527	1327	2457
xml3	8190	2427	2350	1957	4079
xml4	16383	3898	4001	3156	7336
Hebbib	253230	72514	80079	55149	148890

TABLE 1: *Comparative chart of compression performance*

For our experiments, we have chosen files of different nature: the English Bible *bib*, and a large XML file *xml*. Their dictionaries were built from all the words that occur in these files. We then considered different prefixes of these dictionaries, so that we get sub-dictionaries of approximate sizes $2K$, $4K$, $8K$ and $16K$. To see how the methods scale up, we have also included as last line the dictionary of all the words in the Hebrew Bible. Table 1 gives the compression performance: the second column gives the sizes of the original sub-dictionaries, the third column gives the size of the POM file after using Huffman coding, when the values for n_i and ℓ_i are expressed in bits, the fourth column contains the corresponding values for the Fibonacci variant, ℓ_i being expressed in characters, the fifth column corresponds to a Huffman encoded POM file, for which n_i and ℓ_i represented character counts, and the last column is the performance of POM alone. The POM-Huffman methods use three Huffman trees, one for each of the components σ_i, ℓ_i and n_i. The POM-Fibonacci method uses only two components σ_i and ℓ_i. As can be seen, Fibonacci performs better for small files. This advantage could be explained by the use of two fields instead of three, and the fact that Huffman coding requires more additional space for the alphabet and its distribution.

size	Fibonacci without 11	Fibonacci with 11
8002	1812	1836
16496	3811	3855
23985	5558	5585

Table 2: *Memory storage of the two Fibonacci methods*

Table 2 compares the storage performance of the two different Fibonacci encodings, discussed in the previous section, on three sub-dictionaries of different sizes. The first column gives the size, in bytes, of the uncompressed dictionaries, the second and third columns the sizes of the POM-Fibonacci compressed dictionaries, without and with the use of the first codeword 11, respectively. As can be seen, it is worth eliminating the 11 codeword, though the difference is small.

File	size	Huffman	Fibonacci	Huffman decode
bib1	2044	6.7	2.8	7.7
bib2	4095	7.5	3.7	8.8
bib3	8067	8.4	4.9	8.9
bib4	16199	9.9	6.8	10.1
xml1	2047	7.3	3.2	7.0
xml2	4093	7.9	3.8	7.6
xml3	8190	8.5	4.7	8.3
xml4	16383	9.7	6.1	9.7
Hebbib	253230	50	65	64

Table 3: *Empirical comparison of processing time*

To empirically compare the processing times, we considered all of the words which occur in the dictionary. We thus considered one pattern for each entry in the dictionary, and averaged the search times. The results in milliseconds are given in Table 3. The last column is added for comparison: it corresponds to the character oriented Huffman coded POM file which is decoded and then scanned with the algorithm of Fig. 2. For the smaller files, there is a clear advantage of the Fibonacci approach since a part of the encoded file is not scanned. For the larger files, the Huffman variant is better, which could be explained by the smaller file to be processed. Both compressed matching techniques are generally better than decompressing and searching afterwards.

6. Conclusion

We introduced two new methods to represent a POM file so that direct search could be done in these compressed dictionaries. The processing time is typically twice as fast for the Fibonacci variant than for the Huffman based algorithm, and also compared to decoding a Huffman encoded POM file and searching on the uncompressed version. We see that in the case of small files, which is the important application since dictionaries are usually kept in small chunks, the Fibonacci variant is much faster than decoding and searching or than the POM–Huffman method. Even though the compression performance might be slightly inferior to the character version of Huffman (but is still generally better than the bit version), this might well be a price worth to pay for getting the faster processing.

References

[1] Amir A., Benson G., Efficient two-dimensional compressed matching, *Proc. Data Compression Conference DCC–92*, Snowbird, Utah (1992) 279–288.

[2] Amir A., Benson G., Farach M., Let Sleeping Files Lie: Pattern Matching in Z-compressed Files, *Journal of Computer and System Sciences* **52** (1996) 299–307.

[3] Bratley.P, Choueka.Y, Processing Truncated Terms in Document Retrieval Systems, *Information Processing & Management***18(5)**, (1982) 257–266.

[4] Farach M., Thorup M., String Matching in Lempel-Ziv Compressed Strings, *Proc. 27th Annual ACM Symposium on the Theory of Computing*, (1995) 703–712.

[5] Fraenkel A.S, Klein S.T., Robust universal complete codes for transmission and compression, *Discrete Applied Mathematics* **64** (1996) 31–55.

[6] Klein S.T., Skeleton Trees for efficient Decoding of Huffman Encoded Texts, *Information Retrieval* **3** (2000) 7-23.

[7] Klein S.T., Shapira D., A new compression method for compressed matching, *Proc. Data Compression Conference DCC–2000*, Snowbird, Utah (2000) 400–409.

[8] Klein S.T., Shapira D., Pattern Matching in Huffman Encoded Texts, *Proc. Data Compression Conference DCC–2001*, Snowbird, Utah (2001) 449–458.

[9] Manber U., A Text Compression Scheme That allows Fast Searching Directly in the compressed File, *ACM Trans. on Inf. Sys.* **15** (1997) 124–136.

[10] Navarro G., Kida T., Takeda M., Shinohara A., and Arikawa S., Faster Approximate String Matching over Compressed Text, *Proc. Data Compression Conference DCC–2001*, Snowbird, Utah (2001) 459–468.

[11] Navarro G., Raffinot M., A general practical approach to pattern matching over Ziv-Lempel compressed text, *Proc. 10th Annual Symposium on Combinatorial Pattern Matching CPM–99*, *LNCS* **1645**, Springer Verlag, (1999) 14–36.