Submitted to *Algorithms*. Pages 1 - 13.

OPEN ACCESS

Article

Compressed Matching in Dictionaries*

Shmuel T. Klein [1,⋆] and Dana Shapira [2]

[1] Dept. of Computer Science, Bar Ilan University, Ramat-Gan 52900, Israel
[2] Dept. of Computer Science, Ashkelon Academic College, Ashkelon, Israel

E-mail: tomi@cs.biu.ac.il; shapird@ash-college.ac.il

⋆ Author to whom correspondence should be addressed; Tel: (972–3) 531 8865, Fax: (972–3) 736 0498.

Version November 23, 2009 submitted to Algorithms. *Typesetted by LaTeX using class file mdpi.cls*

Abstract: The problem of compressed pattern matching, which has recently been treated in many papers dealing with free text, is extended to structured files, specifically to dictionaries, which appear in any full-text retrieval system. The prefix-omission method is combined with Huffman coding and a new variant based on Fibonacci codes is presented. Experimental results suggest that the new methods are often preferable to earlier ones, in particular for small files which are typical for dictionaries, since these are usually kept in small chunks.

Keywords: Dictionaries, IR systems, pattern matching, compressed matching, Huffman codes, Fibonacci codes.

---

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 complementing 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

*This is an extended version of a paper that has been presented at the *Data Compression Conference DCC'02*, Snowbird, Utah (2002) and appeared in its Proceedings, pp. 142–151.

variety of known compression techniques, such as Huffman coding [13] or various variants of the Lempel and Ziv (LZ) methods, including LZW [2, 8, 18], gzip, DoubleSpace and many others [12, 16, 17].

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.

Compression of structured data has been suggested in the past, though not in the context of compressed matching. For example, database compression techniques require the ability to retrieve arbitrary records even when compression is used. Ng and Ravishankar [19] explore the compression of large statistical databases and propose techniques for organizing the compressed data so that standard database operations such as retrievals, inserts, deletes and modifications are supported.

Another example is compressing XML files, as suggested in Levene and Wood [14], by separating the document structure from its data. Various XML compression methods exist, such as the offline compressor `XMill` [15], so called container-based compression, that is, the data is partitioned into containers depending on the element names, and the containers are then compressed using `Gzip`. The structure is encoded as a sequence of numeric tokens that represent both the XML markup (start-tags, end-tags, etc.) and the references to data containers. `XMill` achieves better compression than `Gzip` and runs at about the same speed. Another known online compressor is `XMLPPM` [7], based on a modification of the PPM2 compression scheme. It compresses better than `XMill`, but runs considerably slower, in part because of the use of arithmetic coding. `XGrind` [21] is another XML compressor which makes it possible to query compressed data by using Huffman coding. `XMLZip` [22] breaks the structural tree of the document at a specified depth, and compresses the resulting components using traditional dictionary compression techniques.

A further example of structured files compression is *table compression*, i.e., collections of fixed length records (unlike databases that might contain intermixed fixed and variable length fields). Table compression was introduced in the work of Buchsbaum et al. [5], where it was empirically shown that partitioning the table into contiguous intervals of columns, compressing each interval separately and grouping dependent columns, can achieve significant compression improvements. Buchsbaum et al. [6] provide theoretical evidence to a generalized partitioning approach of [5], and design new algorithms for continuous partitioning. Experimental results suggest that these algorithms yield better compression than optimal contiguous partitioning without reordering.

The structured files we deal with in this paper are *dictionaries*, and we assume more specifically that they were compressed by the *prefix omission method* (POM); the goal is to enable pattern matching that could be done directly on these files. Prefix omission is a very simple, yet effective, dictionary compression technique, and is therefore widely used. For compressed matching, however, 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 [13].

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. In Section 5 we mention an alternative solution to the problem, based on tries, and 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 the 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 compressed form of the i-th entry X_i as an ordered triple (ℓ_i,n_i,σ_i), where ℓ_i is the number of characters copied from the beginning of the previous (uncompressed) 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 + Huffman coding	
		with preserving codeword boundaries	without . . .
`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: the first column is a list of some consecutive words, the second column gives the compressed form of these words using POM. In the third column ℓ_i and n_i represent the number of bits in the binary representation of the lengths of the strings to be copied and remaining, resp., while preserving codeword boundaries, and σ_i is shown in its compressed form using Huffman coding. The last column extends ℓ_i to be the maximal number of* **bits** *copied from the previous entry, regardless of codeword boundaries.*

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 (the last two columns are referred to below, in Section 3).

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 $\mathcal{D} = \mathcal{E}^{-1}$ 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 (Bratley and Choueka, 1982), 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 complementing 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

$$\begin{aligned} & & s_i = t_i & \quad \text{for } 1 \le i \le pre(S,T), \\ \text{and} \quad & \min(k,\ell) > pre(S,T) \longrightarrow & s_j \ne t_j & \quad \text{for } j = pre(S,T)+1. \end{aligned}$$

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 $\quad i \longleftarrow 2; \quad j \longleftarrow pre(P, \mathcal{D}(\sigma_1));$

2 $\quad$ while $(j < m)$ // Pattern not found

$\quad$ {

2.1 $\quad\quad$ while $\mathcal{D}(\ell_i) > j$

2.1.1 $\quad\quad\quad i \longleftarrow i+1$

2.2 $\quad\quad$ if $\mathcal{D}(\ell_i) < j$ // The closest lexicographically preceding word

2.2.1 $\quad\quad\quad$ return $i-1$

2.3 $\quad\quad$ else // $\mathcal{D}(\ell_i) = j$

$\quad\quad$ {

2.3.1 $\quad\quad\quad tmp \longleftarrow pre(P[j+1,m], \mathcal{D}(\sigma_i))$

2.3.2 $\quad\quad\quad$ if $tmp = 0$ and $P[j+1,m] \succ \mathcal{D}(\sigma_i)$ return $i-1$

2.3.3 $\quad\quad\quad j \longleftarrow j + tmp$

2.3.4 $\quad\quad\quad i \longleftarrow i+1$

$\quad\quad$ }

$\quad$ }

3 $\quad$ return $i-1$

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

The algorithm for searching for a pattern P in a dictionary compressed by POM, based on decompressing each entry, is given in Figure 2. It assumes as input a POM compressed dictionary that has been further encoded by some function $\mathcal{E}$, hence the use of the decoding function $\mathcal{D}$ for each of the components. The algorithm returns the index of the closest lexicographically preceding word to the word we are searching for. We start with $\mathcal{D}(\sigma_1)$, which is the first element, since ℓ_1 is always zero. As long as the component $\mathcal{D}(\ell_i)$, 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 $\mathcal{D}(\ell_i)$ is less than the length of 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 $\mathcal{D}(\ell_i)$ 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 $\mathcal{D}(\ell_i)$ 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 [10], 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 canonical codes, for which the codewords are lexicographically ordered, induce the order from the frequencies of the encoded items, not from their own lexicographic order. For example, refer to the alphabet {`t`, `c`, `b`, `a`, `q`} encoded by the canonical code {00, 01, 10, 110, 111}. The string `qt` precedes `tq`, but considering their encodings, 11100 follows 00111. We can therefore only either locate the pattern, or announce a mismatch.

1.1 $i \longleftarrow 2; \quad j \longleftarrow pre(\mathcal{E}(P), \sigma_1);$
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 // $j - security \leq \mathcal{D}(\ell_i) \leq j$
{
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*

The compressed matching algorithm in POM files which were compressed by using 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 1.1 of the algorithm in Figure 3 we get that $j = 6$. As $j < |\mathcal{E}(P)| = 10$, we perform line 2.1. 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* $= \max_{c \in \Sigma}\{|\mathcal{E}(c)|\}$. 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* $= \max_{c \in P}\{|\mathcal{E}(c)|\}$.

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. A part of the processing time might be saved by using alternatives to Huffman codes which have recently been suggested, such as (s, c)-dense codes [4] or Fibonacci codes [11], trading the optimality of Huffman's compression performance against improved decoding and search capabilities.

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 [9, 11].

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, 100011, 010011, 001011, 101011, 0000011, $\ldots\}$. More precisely, the codeword set consists of all the binary strings for which the substring 11 appears exactly once, at the right end of the string. The specific order of the sequence above, which is used in the coding algorithms, is obtained as follows: just as any integer k has a standard binary representation, that is, it can be uniquely represented as a sum of powers of 2, $k = \sum_{i \geq 0} b_i 2^i$, with $b_i \in \{0, 1\}$, there is another possible binary representation based on Fibonacci numbers, $k = \sum_{i \geq 0} f_i F(i)$, with $f_i \in \{0, 1\}$, where it is convenient to define the Fibonacci sequence here by $F(0) = 1$, $F(1) = 2$ and $F(i) = F(i-1) + F(i-2)$ for $i \geq 2$. This Fibonacci representation will be unique if, when encoding an integer, one repeatedly tries to fit in the largest possible Fibonacci number.

For example, the largest Fibonacci number fitting into 19 is 13, for the remainder 6 one can use the Fibonacci number 5, and the remainder 1 is a Fibonacci number itself. So one would represent 19 as $19 = 13 + 5 + 1$, yielding the binary string 101001. Note that the bit positions correspond to $F(i)$ for increasing values of i from right to left, just as for the standard binary representation, in which $19 = 16 + 2 + 1$ would be represented by 10011. Each such Fibonacci representation starts with a 1, so by preceding it with an additional 1, one gets a set of uniquely decipherable codewords. Decoding, however, would not be instantaneous, because the set lacks the prefix property, but this can be overcome by simply reversing each of the codewords, which yields the sequence above. The adjacent 1s are then at the right instead of at the left end of each codeword, e.g., the codeword corresponding to 19 would be 1001011.

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, so that the compressed dictionary will contain only the Fibonacci compressed forms of the ℓ_i and σ_i fields.

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 frequent 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 $\Sigma \cup \{$`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.

As mentioned, 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. It turns out that for our current application, it is convenient to reverse the codewords back to their original form: by doing so, 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 for decompressing the Fibonacci encoded field ℓ_i, so that the comparisons in lines 21

1 $i \longleftarrow 2$; $j \longleftarrow$ *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 $tmp \longleftarrow$ *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*

and 2.2 can be done directly with the encoded binary strings. For example, the Fibonacci codewords for 19 and 12 are, respectively, 1101001 and 110101; when being compared without decoding, they would be considered as the (standard) binary representations of 105 and 53, but for the algorithm to be correct, it is only their relative order that matters, not their exact values.

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 beginning of a codeword, i.e., starts with 11. The function *fib-pre* in Figure 5 refers to this corrected version: it calculates the number of codewords, rather than the number of bits, in the common prefix, and returns the number of bits in these codewords. For the example above, *fib-pre*$(\mathcal{E}(P), \sigma_1) = 4$.

5. Alternative method

Ristov and Laporte [20] introduce a data structure called an LZ-trie for compressing static dictionaries which is a generic Lempel-Ziv compression of a linked list trie. This compressed trie reduces the size of the dictionary beyond that of a minimal finite automaton and allows the incorporation of the additional data in the trie itself. They perform it by sharing not only common prefixes or suffixes, but also internal patterns. In order to speed up the quadratic time compressing procedure, they use suffix arrays for looking for repeated substrings in the trie.

LZ linked list tries perform best for dictionaries of inflected languages (e.g. Romanic, Slavic) and are less efficient for English. The compression performance of the LZ-trie improves over larger dictionaries.

Since the strings are represented by a compressed trie structure, the look up is not performed sequentially but by following the trie links. The search done by the LZ linked list trie is therefore significantly faster than that of other methods, that use sequential searching.

6. 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 encoding)	Fibonacci	Huffman (char encoding)	POM	LZ trie
bib1	2044	775	716	616	1171	1000
bib2	4095	1709	1666	1413	2754	2369
bib3	8067	2769	2749	2253	4663	3496
bib4	16199	5242	5379	4276	9217	6002
xml1	2047	1097	999	905	1481	1614
xml2	4093	1640	1527	1327	2457	2138
xml3	8190	2427	2350	1957	4079	2696
xml4	16383	3898	4001	3156	7336	3785
Hebbib	253230	72514	80079	55149	148890	74363

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 represent character counts (as in the third column of the table in Figure 1), rather than the number of bits (as in the fourth column): the encoded numbers are thus smaller, yielding a reduced size of the file. The sixth column is the performance of POM alone. The last column gives the sizes of the LZ tries of [20]. All the sizes are given in bytes and include the overhead caused by storing the Huffman trees. 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, the Fibonacci variant 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. The LZ trie improves on POM, but is inferior to the Huffman encoded POM files, and on the small files also to the Fibonacci encoded ones.

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	decode + search	LZ trie
bib1	2044	6.7	2.8	7.7	.02
bib2	4095	7.5	3.7	8.8	.01
bib3	8067	8.4	4.9	8.9	.01
bib4	16199	9.9	6.8	10.1	.01
xml1	2047	7.3	3.2	7.0	.01
xml2	4093	7.9	3.8	7.6	.01
xml3	8190	8.5	4.7	8.3	.02
xml4	16383	9.7	6.1	9.7	.02
Hebbib	253230	50	65	64	.02

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. For comparison, we added also the search times with the LZ trie and a column headed "decode + search", corresponding to the character oriented Huffman coded POM file which is decoded and then scanned with the algorithm of Fig. 2. The direct access of the trie approach, in comparison with the sequential access of the other methods, makes the LZ trie several orders of magnitude faster. Among the other methods, 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 largest file 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.

7. 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. On the other hand, one can get much faster processing using tries, rather than sequential search, but for small dictionaries, compression will then be inferior.

Acknowledgment The authors wish to thank S. Ristov for his help with the implementation of the experiments presented.

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. BRISABOA, N.R., FARIÑA, A., NAVARRO, G. AND ESTELLER M.F., (S,C)-dense coding: an optimized compression code for natural language text databases, Proc. Symposium on String Processing and Information Retrieval SPIRE'03, Manaus, Brazil, October 8–10, LNCS 2857 (2003) 122–136. Springer Verlag, Berlin.
5. BUCHSBAUM A.L., CALDWELL D.F., CHURCH K.W., FOWLER G.S., MUTHUKRISHNAN S., Engineering the compression of massive tables: An experimental approach, *Proc. 10th ACM SIAM Symp. on Discrete Algorithms SODA'00* (2000) 175–184.
6. BUCHSBAUM A.L., FOWLER G.S., GIANCARLO R., Improving table compression with combinatorial optimization, *Journal of the ACM* **50**(6) (2003) 825–851.
7. CHENEY J., Compressing XML with Multiplexed Hierarchical PPM Models, *Proc. IEEE Data Compression Conference DCC'01*, Snowbird, Utah (2001), 163–172.
8. FARACH M., THORUP M., String Matching in Lempel-Ziv Compressed Strings, *Proc. 27th Annual ACM Symposium on the Theory of Computing*, (1995) 703–712.
9. FRAENKEL A.S, KLEIN S.T., Robust universal complete codes for transmission and compression, *Discrete Applied Mathematics* **64** (1996) 31–55.
10. KLEIN S.T., Skeleton Trees for efficient Decoding of Huffman Encoded Texts, *Information Retrieval* **3** (2000) 7-23.
11. KLEIN S.T., KOPEL BEN-NISSAN M., On the Usefulness of Fibonacci Compression Codes, *The Computer Journal* **52** (2009).
12. KLEIN S.T., SHAPIRA D., A new compression method for compressed matching, *Proc. Data Compression Conference DCC'00*, Snowbird, Utah (2000) 400–409.
13. KLEIN S.T., SHAPIRA D., Pattern matching in Huffman encoded texts, *Information Processing &*

Management **41**(4) (2005) 829–841.

14. LEVENE M., WOOD P., XML Structure Compression, Technical Report BBKCS-02-05, School of Computer Science and Information Systems, Birkbeck College, University of London (2002)
15. LIEFKE H., SUCIU D., XMill: an Efficient Compressor for XML Data, *Proc. ACM SIGMOD Conference*, (2000), 153–164.
16. MANBER U., A Text Compression Scheme That allows Fast Searching Directly in the compressed File, *ACM Trans. on Inf. Sys.* **15** (1997) 124–136.
17. NAVARRO G., KIDA T., TAKEDA M., SHINOHARA A., AND ARIKAWA S., Faster Approximate String Matching over Compressed Text, *Proc. Data Compression Conference DCC'01*, Snowbird, Utah (2001) 459–468.
18. 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.
19. NG W.K., RAVISHANKAR C.V., Block-Oriented Compression Techniques for Large Statistical Databases, *IEEE Transactions on Knowledge and Data Engineering*, **9**(2) (1997) 314–328.
20. RISTOV S., LAPORTE E., Ziv Lempel compression of huge natural language data tries using suffix arrays, *Proc. 10th Annual Symposium on Combinatorial Pattern Matching CPM'99, LNCS* **1645**, Springer Verlag, (1999) 196–211.
21. TOLANI P.M., HARITSA J.R., XGRIND: A Query-friendly XML Compressor, *18th International Conference on Data Engineering ICDE'02* (2002), 225–234.
22. XMLSOLUTIONS, XMLZip, URL: `http://www.xmls.com`.