Available online at www.sciencedirect.com

SCIENCE DIRECT®

Information Processing and Management 42 (2006) 429–439

INFORMATION PROCESSING & MANAGEMENT

www.elsevier.com/locate/infoproman

Adapting the Knuth–Morris–Pratt algorithm for pattern matching in Huffman encoded texts

Dana Shapira *, Ajay Daptardar

Computer Science Department, Brandeis University, Waltham MA 02454, United States

Received 8 February 2004; accepted 23 February 2005
Available online 10 May 2005

Abstract

In the present work we perform *compressed pattern matching* in binary Huffman encoded texts [Huffman, D. (1952). A method for the construction of minimum redundancy codes, *Proc. of the IRE, 40*, 1098–1101]. A modified Knuth–Morris–Pratt algorithm is used in order to overcome the problem of *false matches*, i.e., an occurrence of the encoded pattern in the encoded text that does not correspond to an occurrence of the pattern itself in the original text. We propose a bitwise KMP algorithm that can move one extra bit in the case of a mismatch since the alphabet is binary. To avoid processing any bit of the encoded text more than once, a preprocessed table is used to determine how far to back up when a mismatch is detected, and is defined so that we are always able to align the start of the encoded pattern with the start of a codeword in the encoded text. We combine our KMP algorithm with two practical Huffman decoding schemes which handle more than a single bit per machine operation; skeleton trees defined by Klein [Klein, S. T. (2000). Skeleton trees for efficient decoding of huffman encoded texts. *Information Retrieval*, *3*, 7–23], and numerical comparisons between special canonical values and portions of a sliding window presented in Moffat and Turpin [Moffat, A., & Turpin, A. (1997). On the implementation of minimum redundancy prefix codes. *IEEE Transactions on Communications, 45*, 1200–1207]. Experiments show rapid search times of our algorithms compared to the "decompress then search" method, therefore, files can be kept in their compressed form, saving memory space. When compression gain is important, these algorithms are better than *cgrep* [Ferragina, P., Tommasi, A., & Manzini, G. (2004). C Library to search over compressed texts, http://roquefort.di.unipi.it/~ferrax/CompressedSearch], which is only slightly faster than ours.

Keywords: Data compression; Compressed pattern matching; Huffman codes; Knuth–Morris–Pratt's algorithm; Skeleton trees

* Corresponding author.
E-mail addresses: shapird@cs.brandeis.edu, amax@cs.brandeis.edu (D. Shapira).

0306-4573/$ - see front matter © 2005 Elsevier Ltd. All rights reserved.
doi:10.1016/j.ipm.2005.02.003

1. Introduction

The *compressed pattern matching problem* which was first introduced by Amir and Benson (1992) is of searching for a pattern directly in the compressed text without decompressing it. It is a variant of the classical pattern matching problem, in which one is given a pattern P and a text T, and the problem is to locate the first or all occurrences of P in T. In the compressed version of this problem, the text is supposed to be stored in some compressed form. More formally, given a pattern P and text T, and complementary encoding and decoding functions $\mathscr{E}$ and $\mathscr{D}$, respectively, our goal is to search for the encoded pattern, $\mathscr{E}(P)$, in the encoded text $\mathscr{E}(T)$, rather than searching for the pattern, P, in the decompressed text, $\mathscr{D}(\mathscr{E}(T))$. Searching for the encoded pattern $\mathscr{E}(P)$ assumes that the encoded pattern is compressed in the same way throughout the text. This assumption is not always true, especially with dynamic compressions, such as Lempel-Ziv variants, where the compression changes as one proceeds. In these cases the encoding of the pattern is also dynamic, and is computed at each point of the file (Amir, Benson, & Farach, 1996; Klein & Shapira, 2000). Here we deal with static Huffman files (Huffman, 1952), and search for $\mathscr{E}(P)$ in it using a modified KMP algorithm. In order to speed-up the search we use the skeleton trees introduced in the work of Klein (2000) and the Huffman implementation of Moffat and Turpin (1997).

Searching for encoded patterns in encoded texts raises the problem of *false matches*, i.e., finding an occurrence of the encoded pattern in the encoded text which does not correspond to an occurrence of the pattern in the original text, due to crossing codeword boundaries. Consider for example the Huffman code $\{00, 01, 100, 101, 110, 1110, 1111\}$ for the characters t, a, g, d, b, o and c, respectively. The binary string 101-1110-100 is the encoding of the string dog. Suppose, however, that we are searching for the pattern cat: we could find $\mathscr{E}$(cat) starting at the third bit and extending to the end of $\mathscr{E}$(dog). The problem is thus one of verifying that the occurrence detected by the pattern matching algorithm is aligned on codeword boundaries. False matches can be avoided by using a code where no codeword is a prefix or suffix of any other codeword. This type of code is called *Affix* or a *Fixfree code*, and is extremely rare (Fraenkel & Klein, 1990). In this particular example, the code is not an affix code, since the codeword for t is a suffix of the codeword for g.

One approach for performing direct pattern matching in encoded texts is to generate a compression method which is especially suitable for compressed pattern matching. Manber (1997), for example, presents a static compression technique which is based on packing pairs of frequent characters in a single byte. Another work was done by Klein and Shapira (2000), which modifies the LZSS algorithm by moving the pointers backwards in the file, so that pointers point forwards to the reoccurring string, rather than backwards to strings that have already occurred. The dynamic nature of this compression requires a more powerful pattern matching algorithm.

Compressed pattern matching in Huffman encoded text has already been studied. Klein and Shapira (2001) present a probabilistic algorithm for searching Huffman encoded texts. They use the tendency of the Huffman code to re-synchronize quickly after errors. After an occurrence of the compressed pattern in the compressed text has been detected, the search continues by jumping back in the compressed text by a fixed number of bits, and starting decompressing from there up to the point of the occurrence. If the decoding synchronizes with the occurrence, a match with high probability is announced. Although the probability of finding wrong matches is low, in this work we present a deterministic algorithm.

Turpin and Moffat (1997) present an algorithm to directly search texts which were compressed using word-based Huffman codes, allowing only one word patterns. They construct an index of all words that occur in all files in a given directory, and apply a known pattern matching technique (such as *agrep* by Wu & Manber, 1992) on the index, in order to search for a pattern in this set of files. The index, which lists all words and their occurrences, is stored as part of the compressed file. We adapt the way the Huffman file is decoded and extend this work by allowing more than a single word pattern to be searched. Using their implementation, and our modified KMP algorithm, the file is searched directly, and more than a single bit can be handled at a time.

Another word-based Huffman method is studied in the work of deMoura, Navarro, Ziviani, and Baeza-Yates (2000). They present a compression and decompression technique where arbitrary portions of the compressed text can be decompressed, without the need for decompressing the entire file. Moreover, both exact and approximate pattern matching can be done directly on the compressed text. Their compression uses a word-based model and is based on byte oriented Huffman coding rather than bit oriented. Instead of using the original binary Huffman coding, their tree's degree is either 128 in what they call a *tagged Huffman code*, or 256 in their *plain Huffman code*. Thus the atomic unit of each codeword is a byte, and traditional byte oriented algorithms can be employed for searching through the compressed text. Although, they achieve fast searching times, the disadvantage of this approach is not only the need to re-compress Huffman encoded files in order to apply their pattern matching algorithm on the encoded file, but also the compression effectiveness using this method as opposed to binary Huffman coding.

The work of Takeda et al. (2002) suggests to perform compressed pattern matching over multi-byte character texts using the Aho-Corasick pattern matching machine. This technique processes each bit of the encoded file exactly once and is applicable to any prefix code including Huffman encoded texts, and is extended to cope with XML documents, too. In this paper we use the same idea of Takeda et al. which merges the pattern matching and synchronization tasks into one. However, the algorithm we present here is able to process more than a single bit in one machine operation. Moreover, the processing time for building the Aho-Corasick machine takes $O(m \cdot |\Sigma|)$, while the KMP preprocess stage takes only $O(m)$ processing time.

The remainder of this paper is organized as follows. Section 2 presents our modified KMP algorithm. Section 3 shows how we can combine the modified KMP algorithm with Moffat and Turpin's (1997) decoding technique or Klein's skeleton trees (2000), in order to perform compressed pattern matching. Section 4 presents experiments that compare both processing time and compression performance of our implementations against the traditional "decompress then search", and *cgrep* of deMoura et al. (2000).

2. Modifying the KMP algorithm

While searching for a given pattern in the compressed text, we first compress the pattern with the same canonical code that was used for generating the compressed text. We then use a modified Knuth–Morris–Pratt algorithm (1977) to search for the compressed pattern $\mathscr{E}(P)$ directly in the compressed text $\mathscr{E}(T)$. Note that the Boyer-Moore algorithm which searches the pattern from its right end, is not suitable for our purpose since one can not determine the codeword boundaries in the compressed text unless the text is decoded from left to right. Moreover, Boyer-Moore's algorithm, even with its sub-linear performance, is suitable for large alphabets rather than a binary alphabet as in Huffman encoded texts.

The basic idea behind the original KMP algorithm is that each time a mismatch is detected, we know exactly how far to back-up the pointer in the pattern since this relies only on the characters in the pattern and not in the text. Consequently, the pointer in the text is never decremented. To accomplish this, the pattern is preprocessed to obtain a table that gives the index in the pattern of the character to be used for the next comparison with the character that caused the mismatch in the text. We use this idea for searching for the encoded pattern in the encoded text by preprocessing the encoded pattern.

For a given pattern $P = p[0..m-1]$ of length m, during the preprocessing stage of the original KMP algorithm, a $next[0..m-1]$ table is used to determine how far to back up when a mismatch is detected. The original preprocessing algorithm slides a copy of the first i characters of the pattern over itself, from left to right, starting with the first character of the copy over the second character of the pattern, and stopping when all overlapping characters match. These overlapping characters define the next possible position the pattern could match, if a mismatch is detected at *p*[*i*]. The distance to back up in the pattern (*next*[*i*]) is exactly the number of overlapping characters. Conventionally, *next*[0] = −1, which means that there is no overlap, and one must slide the pattern all the way to its beginning. The formal definition of the original *next*[] table is as follows:

$$\begin{array}{ll} next[i] = -1 & i = 0 \\ next[i] = \max\{j - 1 : j < i \text{ and } p[1..j] \text{ is a suffix of } p[1..i-1](1 - p[i])\} & i > 0 \end{array}$$

For our purposes we cannot apply the same algorithm to generate the KMP next table for individual bits of the compressed pattern, as the table might instruct us to perform comparisons in between codewords boundaries. These comparisons must be omitted since they would result in a false match.

This situation is illustrated in the following example and presented in Fig. 1. Let a = 00, b = 01, c = 100, d = 111 and e = 110, and let P = bace, and assume that the encoded pattern is aligned on some codeword boundary. The original KMP table assigns *next*[8] = 5, i.e., if a mismatch at the bit indexed 8 is detected, the next bit to be processed is the one indexed five. Even though there might be a match of $\mathscr{E}(P)$ at the shifted position of $\mathscr{E}(T)$, this shifting will result in a mismatch, since the first bit of $\mathscr{E}(P)$ will be aligned on the second bit of the character a and thus not aligned on the first codeword boundary of $\mathscr{E}(T)$.

Another problematic solution is using the *next* table generated by the original KMP on the characters of the uncompressed pattern, and applying it to the underlying bits. The following example illustrates the drawback of this solution. Assume the same codewords as in the previous example, and let P = bacbaebad. Then $\mathscr{E}(P)$ = 01-00-100-01-00-110-01-00-111. Suppose there is a mismatch when we are located on the last bit of $\mathscr{E}(P)$. The original KMP table generated from the characters, will direct us to the character c indexed 2, while we already know that no occurrence occurs at the current position, since the last three bits read from $\mathscr{E}(T)$ are 110 and the codeword of c is 100. In this example one can slide the pattern all the way so that the current position is aligned on the first bit of $\mathscr{E}(T)$. Note that shifting the pattern as instructed in the original KMP table, will force us to recheck bits.

The KMP next table for the encoded pattern, *next_bit*[0..|$\mathscr{E}(P)$| − 1], therefore must be such that each bit of the encoded text is processed exactly once while at the same time keeping codeword boundaries aligned. Preprocessing the compressed pattern can then be done in the following way. For each bit in the compressed pattern look for the longest prefix that matches the current suffix, so that they are both aligned on the first codeword boundary (and therefore on all codeword boundaries). Also, since we are dealing with the binary alphabet, we can improve on this algorithm by taking into account the bit that caused the mismatch. If the pattern's bit that caused the mismatch is a 1, one can use this information and slide it to a point where the corresponding bit in the encoded pattern is a 0.

P	b		a		c			e		
$\mathcal{E}(P)$	0	1	0	0	1	0	0	1	1	0
j	0	1	2	3	4	5	6	7	8	9
$next[j]$	-1	0	-1	1	0	-1	1	0	5	-1
after shifting				0	1	0	0	1	0	...

Fig. 1. Preprocessing the encoded pattern using the original KMP preprocessing algorithm performed on bits, might result in false matches.

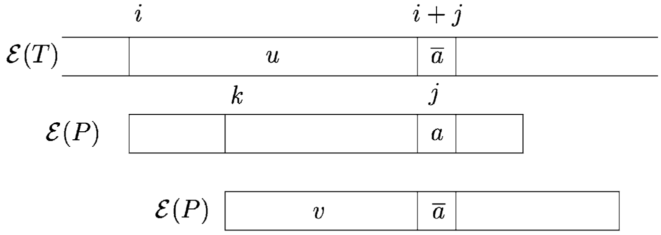

Fig. 2. A mismatch between bits $\mathscr{E}(P)[j]$ and $\mathscr{E}(T)[i+j]$.

Given an encoded pattern $\mathscr{E}(P)$ of length m and an encoded text $\mathscr{E}(T)$ of length n, we consider an attempt to match the encoded pattern with the encoded text at position i, that is, the encoded pattern $\mathscr{E}(P)$ is aligned with the sub-pattern $\mathscr{E}(T)[i..i+m-1]$ of the encoded text. Assume that the first mismatch occurs between bits $\mathscr{E}(P)[j]$ and $\mathscr{E}(T)[i+j]$ for some $0 \leqslant j < |\mathscr{E}(P)|$, as shown in Fig. 2. Then $\mathscr{E}(P)[0..j-1] = \mathscr{E}(T)[i..i+j-1]$. The *next_bit*[$i$], $0 \leqslant i < |\mathscr{E}(P)|$, gives the index of the longest prefix v of $\mathscr{E}(P)$ that matches some suffix u of $\mathscr{E}(P)[0..j-1]$, and they are both aligned on codeword boundaries. In addition, the bit following v is equal to $\overline{a} = 1 - a$, where a is the bit following u.

We define a mapping I between indices of the characters of the pattern and indices of the bits of the encoded pattern. Using a zero based index, the ith character, $0 \leqslant i < |P|$, is mapped to the index of the last bit corresponding the ith codeword of $\mathscr{E}(P)$. More formally, $I(i) = \sum_{j=0}^{i}|\mathscr{E}(p[j])| - 1$. For example, consider the codewords and pattern of the last example, and let us refer to the character c with index 2 (starting from 0), then $I(2) = 6$, since the index of the last bit of c is 6.

In order to compute the *next_bit* table, we match the pattern against itself, as in the original KMP algorithm, but at the same time we also handle the codeword boundaries. The algorithm is presented in Fig. 3. All entries of the first codeword are set to −1, since a mismatch in the first codeword enforces a mismatch of the entire pattern. In order to slide the pattern against itself, we use two indices i and j. If j is equal to −1 and i is not pointing to the last bit in a codeword, we advance i to the last bit of the current codeword, while setting each entry to −1. Otherwise, we are pointing on a codeword boundary, and we can use the original logic of the KMP algorithm to compute the entries for the *next_bit* table, based on the already computed ones.

The algorithm for searching Huffman encoded texts is given in Fig. 4. Each bit in the encoded file is processed only once, using the modified KMP algorithm. Unlike the original KMP algorithm which uses a while statement in the following code:

```
while($j \geqslant 0$) and ($b \neq \mathscr{E}(P)[j]$) do
    $j \leftarrow next\_bit[j]$
```

here an if statement suffices, since one can take advantage of the fact that the alphabet is binary, i.e., if a mismatch is detected, and one skips to some location pointed by the next_bit table, then the bits in the new location match. Note that using the original KMP algorithm on the characters (symbols of the uncompressed text), effectively decompresses the file, which is something we wanted to avoid.

3. Fast pattern matching on Huffman texts

The number of Huffman trees for a given probability distribution is quite large. Many applications prefer to use the data structure defined by Schwartz and Kallick (1964) known as a *canonical tree*. A tree is called canonical if when scanning its leaves from left to right, they appear in non-decreasing order of their

```
Preprocess($\mathcal{E}(P)$)
begin
    /* Set all entries of first codeword to -1 */
    $i \leftarrow 0$
    while ($i \leq I(0)$) do
        $next\_bit[i] \leftarrow -1$
        $i \leftarrow i + 1$
    end while
    $i \leftarrow I(0)$ /* Set $i$ to last bit in first codeword*/
    $j \leftarrow -1$
    while ($i \leq |\mathcal{E}(P)|$) do
        while (($j \geq 0$) and ($\mathcal{E}(P)[i] \neq \mathcal{E}(P)[j]$)) do
            $j \leftarrow next\_bit[j]$
        end while
         if($j \neq -1$)
            while($i \neq I(i)$) do/*not last bit in the current codeword*/
                $i \leftarrow i + 1$
                $next\_bit[i] \leftarrow -1$
            end while
        end if
        $i \leftarrow i + 1$
        $j \leftarrow j + 1$
        if($\mathcal{E}(P)[i] = \mathcal{E}(P)[j]$)
            $next\_bit[i] \leftarrow next\_bit[j]$
        else
            $next\_bit[i] \leftarrow j + 1$
        end if
    end while
end
```

Fig. 3. Computation of KMP *next_bit* table adapted for Huffman encoded patterns.

depth. An equivalent way for defining it is that when the codewords are sorted by the frequency of their corresponding symbols, they are ordered lexicographically. When using canonical trees, decoding can be done in a more efficient manner, requiring less memory space and fewer bitwise operations. Moffat and Turpin (1997) and Klein (2000) use the canonical codes for fast decoding of Huffman texts, so that more than a single bit can be processed in one machine operation. In this section we show how to combine the modified KMP algorithm with these methods.

```
KMP_search($\mathcal{E}(P), \mathcal{E}(T)$)
begin
    Preprocess $\mathcal{E}(P)$ to obtain the $next\_bit$ table
    while (not end of input)do
        get next bit $b$;
        if($j \geq 0$) and ($b \neq \mathcal{E}(P)[j]$) do
            $j \leftarrow next\_bit[j]$
        end if
        if($j = |\mathcal{E}(P)|$)
            announce a match
            $j \leftarrow -1$
        end if
        $j \leftarrow j + 1$
    end while
end
```

Fig. 4. The modified *KMP-search* algorithm for Huffman encoded texts.

3.1. KMP with skeleton trees

In the *KMP-search* algorithm, Fig. 4, we are processing each bit of the encoded text in order to locate the encoded pattern. We present the *sk-KMP* algorithm for searching Huffman texts, which improves it by processing more than a single bit per machine operation, using *skeleton trees* (2000). This data structure represents canonical Huffman codes in a space efficient way and speeds up the decoding by handling more than a single bit at a time. A skeleton tree is a binary tree which is induced by the underlying Huffman tree, where all full subtrees of depth at least 1 have been pruned. That is, the nodes of the Huffman tree that remain in the skeleton tree are those up to the depth necessary to identify the length of the codeword with the prefix corresponding to the path from the root to that node. The leaves v of the skeleton tree then contain the length of the corresponding codewords $\ell(v)$. To search for the encoded pattern in the Huffman encoded text using a skeleton tree, we use a pointer to point to the root of the skeleton tree, a pointer to point to the first bit of the encoded text, and another pointer to point to the current index in our pattern (and, therefore, in the *next_bit* table). The encoded text is scanned, while simultaneously traversing the skeleton tree and keeping track of the position j in the *next_bit* table. After having read a 0 from the encoded text, we proceed to the left child of the current node in the skeleton tree, otherwise we proceed to the right. When we reach a leaf of the skeleton tree, the length of the current codeword is identified, and the $\ell(v)$ remaining bits are fetched in order to complete reading the current codeword, in addition to updating the position j of the *next_bit* table at the end of the current codeword.

The algorithm in Fig. 5 shows how to combine the skeleton tree with the modified KMP algorithm. We use v to point to the nodes in the skeleton tree, and j to point to bits of the encoded pattern. Each leaf, v, of the skeleton tree, that is also on a path of a codeword of $\mathscr{E}(P)$, has a *leaf_next* table, with 2^{ℓ} entries, where ℓ is the remaining bits from v to a Huffman leaf rooted at v. A single lookup in *next_leaf*(v,j)[y] will give us the position in $\mathscr{E}(P)$ after reading into y, the remaining ℓ bits of the codeword. The *sk-KMP* algorithm performs a single operation per node in the skeleton tree, and not per node in the Huffman tree, saving in practice, about 50% of bit operations (2000).

```
sk_KMP_search($\mathcal{E}(P), \mathcal{E}(T)$)
begin
    Preprocess ($\mathcal{E}(P)$) to obtain the $next\_bit$ table
    Initialize the *leaf_next* tables
    $v \leftarrow$ root(sk_tree)
    $j \leftarrow 0$
    while (not end of input) do
        Let $b$ be the next bit
        if ($b = 0$)
            $v \leftarrow left(v)$
        else
            $v \leftarrow right(v)$
        end if
        if ($v$ is a leaf of the skeleton tree )
            read next $\ell(v)$ bits from input into $y$
            $j \leftarrow T(v, j)[y]$
            $v \leftarrow root$
        else if ($j \geq 0$ and $\mathcal{E}(P)(j) \neq b$)
                $j \leftarrow next\_bit[j]$
        end else
        if($j = -1$)
            skip to the next codeword
            $v \leftarrow root(sk\_tree)$
        end if    if($j = |\mathcal{E}(P)|$)
            announce a match at the current position
            $j \leftarrow -1$
        end if    $j \leftarrow j + 1$
    end while
end
```

Fig. 5. The *sk-KMP search* algorithm using skeleton trees for Huffman encoded texts.

4. Experiments

The data files considered for the experiments were all natural language texts; these are as follows: *world192.txt*—CIA 1992 World Fact-book, *bible.txt*—King James Bible, *books.txt*—Random selection of texts from the Gutenberg Archives which is an archive of over 1000 documents in the English language

Table 1
Compression performance

Files	Size (bytes)	Compression ratio		
		cgrep	*Huff*	*gzip*
world192.txt	2,473,400	50.88	32.20	**29.29**
bible.txt	4,047,392	49.70	**26.18**	29.42
books.txt	12,582,090	52.10	**30.30**	37.04
95-03-erp.txt	23,976,547	34.49	25.14	**22.53**

in computer text format (Gutenberg Archives, 2004), *95-03-erp.txt*—US Economic Reports[1] from 1995 to 2003. All experiments were performed on an Intel PC with a 900 MHz AMD Athlon CPU with 256 KB cache memory and 256 MB of main memory running RedHat Linux-7.3.

The compression parsing model used on the input files uses a word-based alphabet together with the *spaceless words* method presented in the work of deMoura et al. (2000). Every word is assumed to be followed by a space, in which case only the word is encoded, otherwise if the next symbol corresponds to a separator, in which case the separator must also be encoded. For compressing the files, we used Moffat and Katajainen's algorithm (1995) to first compute the lengths of the codewords and then proceed with the canonical code construction.

To speed-up the KMP algorithm we use Moffat and Turpin's implementation of Huffman decoding (1997) who use the structure of canonical codes, so that all codewords of a given length are consecutive binary integers. Only the first codeword of each length is stored, which also gives a sorted list of integers. A window, at least as long as the longest codeword, slides through the compressed stream, and its numerical value is compared against each one of these integers. The length of the next codeword is then determined, and the translation of the symbols number to the output string is done by a table look-up. We combine this with our modified KMP algorithm and call it *win-KMP*.

We compare our algorithms, *sk-KMP* and *win-KMP*, against *cgrep* of deMoura et al. (2000), and *agrep* of Wu and Manber (1992) which searches the uncompressed text for patterns with or without errors. The alphabet of the *cgrep*, includes all words and separators of the text, and each element is assigned a sequence of bytes using only the 7 lower order bits of each byte. The most significant bit (MSB) in the first byte of each codeword is used as a separator between codewords by setting it to 1, while all other byte's MSB is set to 0. This is done so that they can make sure that a reported match is not a false match. An implementation of *cgrep* can be downloaded from http://roquefort.di.unipi.it/~ferrax/CompressedSearch (Ferragina, Tommasi, & Manzini, 2004). This particular implementation searches for single words with or without errors.

Table 1 compares the compression performance of the original Huffman compression (*Huff*) against the compression achieved by *cgrep* and *gzip*. The compression ratio presented in this table is the size of the compressed text as a percentage of the uncompressed text.

Table 2 compares the processing time of pattern matching of these algorithms. The figures are given in seconds. The "decompress then search" methods, first decode using skeleton trees (*sk-d*) or Moffat and Turpin's sliding window algorithm (*win-d*) and then perform the search using *agrep*.

As can be seen from these tables, using our modified KMP algorithm implemented with either skeleton trees or Moffat and Turpin's decoding, is faster than the "decompress then search" algorithms using the corresponding methods. When comparing the processing times of the KMP variants to *cgrep*, *cgrep* is faster, but its compression is less effective than Huffman. Not only should files be re-compress in order to use

[1] The original files were in Adobe's pdf file format from which they were converted to text using the UNIX utility (`pdftotext` (Noonburg, 2004)).

Table 2
Search performance

Files	Size (bytes)	Search times (s)				
		cgrep	*sk-KMP*	*win-KMP*	*agrep*	
					sk-d	*win-d*
world192.txt	2,473,400	0.07	0.13	0.08	0.21	0.13
bible.txt	4,047,392	0.05	0.22	0.13	0.36	0.22
books.txt	12,582,090	0.21	0.69	0.39	1.21	0.74
95-03-erp.txt	23,976,547	0.18	1.10	0.65	1.80	1.11

cgrep, but it also harms the compression. Moreover, compressed files that can fit into the main memory, might exceed the memory space using the *cgrep* compression. Therefore, pattern matching that could have been performed in main memory would now have to include the time spent to transfer the file from secondary storage into main memory.

5. Conclusion

We have modified the Knuth–Morris–Pratt algorithm to perform *compressed pattern matching* in Huffman encoded texts. Our bitwise algorithm processes each bit of the encoded text exactly once. By combining it with the skeleton trees defined by Klein, or the Huffman decoding implementation presented in Moffat and Turpin we are able to handle more than a single bit per machine operation. The processing times are better than the "decompress then search" method and slower than *cgrep*. However, when compression performance is important or when one does not want to re-compress Huffman encoded files in order to use *cgrep*, the proposed algorithms are the better choice.

References

Amir, A., & Benson, G. (1992). Efficient two-dimensional compressed matching. In *Proceedings of the data compression conference DCC-92* (pp. 279–288). Utah: Snowbird.

Amir, A., Benson, G., & Farach, M. (1996). Let sleeping files lie: pattern matching in Z-compressed files. *Journal of Computer and System Sciences, 52*, 299–307.

deMoura, E. S., Navarro, G., Ziviani, N., & Baeza-Yates, R. (2000). Fast and flexible word searching on compressed text. *ACM TOIS, 18*(2), 113–139.

Ferragina, P., Tommasi, A., & Manzini, G. (2004). C Library to search over compressed texts, http://roquefort.di.unipi.it/~ferrax/CompressedSearch.

Fraenkel, A. S., & Klein, S. T. (1990). Bidirectional Huffman coding. *The Computer Journal, 33*, 296–307.

Gutenberg Archives. (2004). http://www.nuc.edu.ng/egranary.

Huffman, D. (1952). A method for the construction of minimum redundancy codes. In *Proceedings of the IRE* (Vol. 40, pp. 1098–1101).

Klein, S. T. (2000). Skeleton trees for efficient decoding of huffman encoded texts. *Information Retrieval, 3*, 7–23.

Klein, S. T., & Shapira, D. (2000). A new compression method for compressed matching. In *Proceedings of the data compression conference DCC-2000* (pp. 400–409). Utah: Snowbird.

Klein, S. T., & Shapira, D. (2001). Pattern matching in Huffman encoded texts. In *Proceedings of the data compression conference DCC-2001* (pp. 449–458). Utah: Snowbird.

Knuth, D. E., Morris, J. H., & Pratt, V. R. (1977). Fast pattern matching in strings. *SIAM Journal on Computing, 6*(2), 323–350.

Manber, U. (1997). A text compression scheme that allows fast searching directly in the compressed file. *ACM Transactions on Information System, 15*, 124–136.

Moffat, A., & Katajainen, J. (1995). In-place calculation of minimum-redundancy codes. In *Proceedings of the workshop on algorithms and data structures* (pp. 393–402).

Moffat, A., & Turpin, A. (1997). On the implementation of minimum redundancy prefix codes. *IEEE Transactions on Communications, 45*, 1200–1207.

Noonburg, D. B. (2004). pdftotext—Portable Document Format (PDF) to text converter (version 1.00), http://www.foolabs.com/xpdf/.

Schwartz, E. S., & Kallick, B. (1964). Generating a canonical prefix encoding. *Communications of the ACM, 7*, 166–169.

Takeda, M., Miyamoto, S., Kida, T., Shinohara, A., Fukamachi, S., Shinohara, T., et al. (2002). In *Processing text files as is: pattern matching over compressed texts, multi-bytes, character texts, and semi-structured texts*. *SPIRE-2002 LNCS 2476* (pp. 170–186). Springer Verlag.

Turpin, A., & Moffat, A. (1997). Fast file search using text compression. *Australian Computer Science Conference*, 1–8.

Wu, S., & Manber, U. (1992). Fast text searching allowing errors. *Communications of the ACM, 35*(10), 83–91.