Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information Processing and Management 41 (2005) 829–841
INFORMATION PROCESSING & MANAGEMENT
www.elsevier.com/locate/infoproman

# Pattern matching in Huffman encoded texts ☆

Shmuel T. Klein [a,*], Dana Shapira [b,1]

[a] Department of Math. & CS, Bar Ilan University, Ramat-Gan 52900, Israel
[b] Department of Computer Science, Brandeis University, Waltham, MA 02254, USA

Received 25 April 2003; accepted 25 August 2003
Available online 25 March 2004

**Abstract**

For a given text which has been encoded by a static Huffman code, the possibility of locating a given pattern directly in the compressed text is investigated. The main problem is one of synchronization, as an occurrence of the encoded pattern in the encoded text does not necessarily correspond to an occurrence of the pattern in the text. A simple algorithm is suggested which reduces the number of erroneously declared matches. The probability of such false matches is analyzed and empirically tested.


*Keywords:* Data compression; Huffman codes; Pattern matching; Compressed matching

## 1. Introduction

The general approach for looking for a pattern in a file that is stored in its compressed form, is first decompressing and then applying one of the known pattern matching algorithms to the decoded file. In many cases, however, in particular on the Internet, files are stored in their original form, for if they were compressed, the host computer would have to provide either memory space for each user in order to store the decoded file, or appropriate software to support on the fly decoding and matching. Both requirements are not reasonable, as many users can simultaneously quest the same information reservoir which will either demand a large quantity of free memory, or put a great burden on the host CPU. Another possibility is transferring the compressed files to the personal computer of the user, and then decoding the files. However, we then assume that the user

☆ This is an extended version of a paper that has been presented at the *Data Compression Conference DCC'01*, Snowbird, Utah, 2001 and appeared in its Proceedings, pp. 449–458.

* Corresponding author. Tel.: +972-3-531-8865; fax: +972-3-736-0498.
*E-mail addresses:* tomi@cs.biu.ac.il (S.T. Klein), shapird@cs.brandeis.edu (D. Shapira).
[1] Tel.: +781-736-2707; fax: +781-736-2741.

0306-4573/$ - see front matter 
doi:10.1016/j.ipm.2003.08.008

knows the exact location of the information she or he is looking for; if this is not the case, much unneeded information will be transferred.

There is therefore a need to develop methods for directly searching within a compressed file. This so-called *compressed matching problem* has been introduced by Amir and Benson (1992), and has recently got a lot of attention (Amir, Benson, & Farach, 1996; Gąsieniec & Rytter, 1999; Farach & Thorup, 1995; Kärkkäinen, Navarro, & Ukkonen, 2000; Kida, Takeda, Shinohara, & Arikawa, 1999; DeMoura, Navarro, Ziviani, & Baeza-Yates, 1998; Navarro & Raffinot, 1999; Shibata, Kida, Takeda, Shinohara, & Arikawa, 2000). It is a variant of the classical pattern matching problem, in which one is given a pattern $P$ and a (usually much larger) text $T$, and one tries 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.

For complementary encoding and decoding functions $\mathscr{E}$ and $\mathscr{D}$, that is, functions such that for any text $T$, one gets $\mathscr{D}(\mathscr{E}(T)) = T$, our aim is to search for $\mathscr{E}(P)$ in $\mathscr{E}(T)$, rather than the usual approach which searches for the pattern $P$ in the decompressed text $\mathscr{D}(\mathscr{E}(T))$. A necessary condition is then that the pattern $P$ should be encoded in the same way throughout the text, which is not the case for arithmetic coding and for dynamic methods such as adaptive Huffman coding. The various Lempel–Ziv variants are also dynamic methods, but for them compressed matching is possible: all of the fragments of the pattern $P$ appear in the compressed text, though not necessarily contiguously and not necessarily in the same order as in the pattern, since parts of the compressed text are pointers to an external dictionary or to previous locations in the given text itself. Much of the previous work on compressed pattern matching concentrates on Lempel–Ziv encodings. A different approach is not to adhere to a known compression scheme, but to devise a new one that is specially adapted to allow efficient searches directly in the compressed file (Manber, 1997; Klein & Shapira, 2000).

Fukamachi, Shinohara, and Takeda (1992) propose a pattern matching algorithm for Huffman encoded strings, based on the Aho–Corasick algorithm. In order to reduce the processing time due to bit per bit state transitions, they use a special code in which the lengths of the codewords are multiples of four bits and present an algorithm for pattern matching in this kind of compressed files. Shibata, Matsumoto, Takeda, Shinohara, and Arikawa (2000) present an efficient realization of pattern matching for Huffman encoded text, substituting $t$ consecutive state transitions of the original machine by a single one. When $t$ is a multiple of 4, this results in a speedup. Takeda et al. (2002) build a pattern matching machine by embedding a DFA that recognizes a set of codewords into an ordinary Aho–Corasick machine, and then make it run over a text byte after byte. Their technique can handle any prefix code including Huffman codes.

DeMoura, Navarro, Ziviani, and Baeza-Yates (2000) propose a compression scheme that uses a word based byte oriented Huffman encoding. The first bit of each byte is used to mark the beginning of a word. Exact and approximate pattern matching can be done on the encoded text without decoding. Their algorithm runs twice as fast as `agrep`, but compression performance is slightly hurt. Moreover, the compression method is not applicable to texts like DNA sequences, which cannot be segmented into words.

In the present work, we are interested in searching within the original Huffman encoded text without any modification. We concentrate on static Huffman coding, for which the problem might at first sight seem trivial. It is, however, not always straightforward, since an instance of $\mathscr{E}(P)$ in the compressed text is not necessarily the encoding of an instance of $P$ in the original text $T$, and

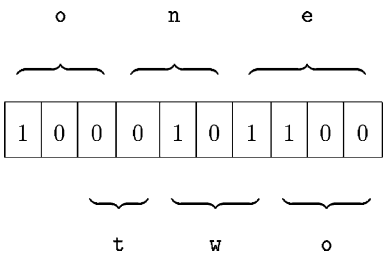


Fig. 1. Example of a false match.

might be crossing codeword boundaries. Consider for example the Huffman code $\{00, 010, 011, 100, 101, 1100, 1101, 111\}$ for the characters T, N, A, O, W, E, B and C respectively. The binary string 1000101100 is the encoding of the string one. Suppose, however, that we are searching for the pattern two: we could find $\mathscr{E}$(two) starting at the third bit and extending to the end of $\mathscr{E}$(one), as shown in Fig. 1.

The problem is thus one of verifying that the occurrence detected by the pattern matching algorithm is aligned on a codeword boundary. In the next section, we suggest an algorithm for compressed matching in Huffman encoded files. Section 3 analyses the probability of getting false matches and experimental results are presented in Section 4.

## 2. Compressed pattern matching for Huffman codes

For a given text $T$ over some alphabet $\Sigma$, we consider the Huffman encoded text $\mathscr{E}(T)$. In order to locate a pattern $P$ in $T$, we start by encoding the pattern and then apply one of the known pattern matching techniques to find $\mathscr{E}(P)$ in $\mathscr{E}(T)$. Note that Boyer and Moore's (1977) algorithm, with its sub-linear performance might not be the best choice here, as we deal with the binary alphabet $\{0, 1\}$. An attractive alternative in our case is Karp and Rabin's (1987) probabilistic pattern matching, specifically because our suggested solution is also probabilistic in nature.

If the algorithm does not find any occurrence, we know that $P$ does not occur in $T$. On the other hand, if an occurrence of $\mathscr{E}(P)$ is detected, we cannot be sure that it corresponds to an occurrence of $P$ in $T$, unless we scan the encoded text from its beginning to locate all the codeword boundaries. This means that we effectively decode the text, which is what we wanted to avoid.

No decoding is necessary, if we also keep the list $I$ of the indices $\{i_1, i_2, \ldots\}$ of the first bit of each codeword. Once the compressed pattern has been located, the index of its location can be searched for in $I$. This would just take $O(\log |T|)$ time using binary search, but keeping the list $I$ might sometimes more than double the size of the compressed file. Consider, for example, a file $\mathscr{E}(T)$ of one KB, consisting of about a thousand codewords of average length 8 bits. The corresponding list $I$ would have about a thousand entries, each requiring 13 bits! It would not really help to record, instead of the list $I$, the sequence $L$ of codeword *lengths*, which are the differences between the codeword starting points $\{i_2 - i_1, i_3 - i_2, \ldots\}$, rather than the absolute indices; $\log m$ bits are then necessary for each length, where $m$ is the maximum codeword length, so that the size of the list $L$ would still be $O(|T|)$. For the above example, if $m = 16$, the size of $L$ would be $\frac{1}{2}$ KB. Even though the space overhead is reduced, the required time can be just as bad as for the sequential decompression from scratch: instead of decoding the text, it is the list $L$ that has to be

processed from its beginning. In fact, if already one agrees to double the size of the file, a simpler solution avoiding the necessity for binary search would be to keep a bit-string $B$ of size identical to that of the compressed file; a bit in a position corresponding to the beginning of a codeword would be set to 1 in $B$, and all the others to 0.

A possible simple solution would be some tradeoff between recording all codeword boundaries or none of them by preparing a small list of possible entry points into the compressed text. Choose a parameter $b$ and partition the compressed file into blocks of $b$ bits; then move, when necessary, each partition point to the closest preceding codeword boundary, and record the index of the first bit in each such block in a list $D$. Once the pattern $\mathscr{E}(P)$ is found in $\mathscr{E}(T)$ at location $\ell$, the list $D$ provides the starting point of the block containing the compressed pattern, so this block can be decompressed. The additional required space is thus $O(|T|/b)$ and decoding time is reduced to $O(b)$. No binary search within $D$ is needed, as the required starting point is stored in the $\lfloor \ell/b \rfloor$th entry of $D$. For certain values of $b$, this may be a recommendable solution, with small storage overhead and fast performance. However, the more we wish to reduce the size of $D$, the larger $b$ will be, implying longer processing. If one agrees to change the encoded file slightly, one can get rid of the list $D$ and force alignment on block boundaries. The total number of additional bits, less than one codeword per block, could be kept very low.

As alternative we suggest a solution that does not alter the compressed file by exploiting the tendency of Huffman codes to resynchronize quickly after errors (Klein & Wiseman, 2000): if the pattern has been found at index $i$, jump back by some constant number of bits $K$ and start decoding from there. It might well be that the bit indexed $i - K$ is not the beginning of a codeword in $\mathscr{E}(T)$, so that the decoding will be erroneous at the beginning. However, if $K$ is chosen large enough, the decoding of the last bits preceding bit $i$ will generally be correct, regardless of possible errors before. One can therefore decide, with a small error probability, whether to announce a match at location $i$ or not, depending on whether bit $i$ is the beginning of a new codeword in the decoding that started at $i - K$. The formal algorithm for finding the occurrences of pattern $P$ in $T$ is given in Fig. 2. It uses the Huffman tree of the given alphabet $\Sigma$, and refers to its root as root. It also uses a procedure $\mathsf{search}(x, y)$, which returns the smallest index $i$ such that the string $x$ matches the substring of $y$ that starts at its $i$th position. If no such index exists ($x$ does not occur as substring of $y$), the procedure returns $\infty$. The decoding then starts at position $i - K$, or at the beginning of the string in case $K > i - 1$. The procedure search can be implemented using any of the known pattern matching algorithms—we shall refer specifically to the Karp Rabin algorithm in Section 3.2 below—but the details have been omitted here to keep the focus on the solution of the synchronization problem.

There are codes for which this algorithm does not work better than without the backward jump of $K$ bits. Indeed, suppose we start the decoding of a given compressed string at two different points, yet according to the same Huffman tree, and suppose that at some point, these two decodings synchronize. Let $x$ and $y$ denote the last codewords for the two decodings before reaching the synchronization point. Then either $x$ is a suffix of $y$ or $y$ is a suffix of $x$. In any case, the underlying Huffman code cannot have the so-called *suffix-property*, asserting that no codeword can be the suffix of any other, similarly to the well-known *prefix-property* of all Huffman codes. Accordingly, codes having both the prefix and the suffix property have been called *never-self-synchronizing* in Gilbert and Moore (1959); they are called *affix* codes in Fraenkel, Mor, and Perl (1983). There are infinitely many different complete variable-length affix codes, e.g., {01, 000,

```
encode P and generate the vector E(P)
while E(T) is not empty
        i ← search(E(P), E(T))
        if i = ∞ STOP
        node ← root
        for j ← max(1, i − K) to i − 1
                if j-th bit of E(T) = 1
                        node ← left (node)
                else
                        node ← right (node)
                if current node is a leaf
                        node ← root
        if node = root
                declare match at address i
        delete the first i bits of E(T)
```

Fig. 2. The compressed matching algorithm.

100, 110, 111, 0010, 0011, 1010, 1011}, but they are nonetheless extremely rare (Fraenkel & Klein, 1990). In particular, the code used in Fig. 1 is not affix, since the codeword for o is a suffix of the codeword for e. Returning now to our compressed matching algorithm, if the code is an affix code and bit $i - K$ does not happen to be the first of a codeword, the erroneous decoding will extend to the end of the file, for any size of $K$.

In practice, however, synchronization is often achieved after a small number of bits, typically less than 100. It seems therefore that by choosing $K$ as a few hundred should generally be enough to avoid errors like declaring a match when in fact there is none, or failing to declare a match even though there actually is one. We bring some experimental results below.

## 3. Estimating the number of errors

### *3.1. False matches in the pattern matching process*

We shall compute an estimated number of false matches using two different models of the probabilistic process underlying the text creation. Both models assume that the text has been

generated by choosing repeatedly, and *independently* from each other, characters from $\Sigma$ according to their probability of occurrence $p_1, \ldots, p_n$ in the text. Such an independence assumption is of course an approximation in many cases, in particular for natural language texts which generally exhibit many dependencies. One could even argue that due to the independence of symbols, regular patterns should not exist and therefore there is no basis for any pattern matching. We consider, however, also very large alphabets, the elements of which are not necessarily single characters, but rather words or even phrases. Such models are frequently used in large information retrieval systems (Witten, Moffat, & Bell, 1994).

We refer in this section to the number of false matches caused by the search function only, as if the algorithm of Fig. 2 were used with backskip parameter $K = 0$. The experiments below suggest that for large enough $K$, the number of false matches generally decreases to zero.

The first model relies on the fact that the string $\mathscr{E}(T)$ is the result of a Huffman encoding process, but ignores the specific probabilities $p_1, \ldots, p_n$. Rather, it uses the corresponding codeword lengths $\ell_1, \ldots, \ell_n$, respectively, and assumes that the probabilities of the occurrence of the characters in the text are $2^{-\ell_1}, \ldots, 2^{-\ell_n}$, and that the characters occur independently of each other. Such a distribution is called *dyadic*. The resulting approximation may be justified by the fact that since the original and the corresponding dyadic distributions yield the same Huffman code, they must be quite similar. A formal definition of this similarity can be found in Longo and Galasso (1982), in which the set of probability distributions is given a pseudo-metric, and an upper bound is derived for the distance of any probability distribution to the dyadic distribution giving the same Huffman tree.

One can then use a theorem shown in Klein, Bookstein, and Deerwester (1989), stating that with these assumptions, the output of Huffman decoding is indistinguishable from a random binary string with probability of occurrence of a 1-bit being equal to $\frac{1}{2}$. For an infinite sequence this would then imply that any binary pattern of length $k$, with $k \geqslant 1$, occurs with probability $2^{-k}$. We shall use this approximation even though $\mathscr{E}(T)$ is finite and the occurrences of characters are not really independent of each other.

To estimate the number of false matches, we proceed as follows: let $m = |\mathscr{E}(P)|$ be the length in bits of the encoding of the pattern, and assume $P$ occurs $t$ times in $T$. Consider a text string $T'$, obtained from $T$ by purging all occurrences of $P$. The Huffman encoding of $T'$, $\mathscr{E}(T')$, is a binary sequence of length $|\mathscr{E}(T)| - tm$ (assuming that there are no overlaps of suffixes of $P$ with prefixes of $P$). Since this too is a Huffman encoded string, the probability of occurrence of $\mathscr{E}(P)$ is $2^{-m}$. No occurrence of $\mathscr{E}(P)$ in $\mathscr{E}(T')$ corresponds to a true match of $P$ in $T'$, so we get as estimate for the number of false matches

$$2^{-m}(|\mathscr{E}(T)| - tm). \tag{1}$$

In fact, we have used here two more approximations: by eliminating all the occurrences of $P$, the original probabilities may have been changed, which could affect the lengths of the corresponding Huffman codewords. If $t$ and $m$ are small relative to the size of the encoded text, the change in the probabilities might be small enough to yield the same Huffman tree (Longo & Galasso, 1982), and even if the tree is altered, the change of the average codeword length will often be negligible. The second approximation is that by removing a true match, a new false match might appear that spans over the gap.

The second model takes the probabilities $p_1, \ldots, p_n$ into account and assumes a complete prefix code, though not necessarily one derived from Huffman's algorithm. For convenience, we shall

still use the terminology of Huffman codes, but the analysis is also valid for any other complete prefix code with associated probabilities. The following notations will be used below. Let $\mathscr{T}$ denote the Huffman tree corresponding to a given Huffman code. The elements which are encoded appear with probabilities $p_1, \ldots, p_n$ in the text, and the lengths of the corresponding Huffman codewords are $\ell_1, \ldots, \ell_n$, respectively. We shall also use the notation $p_y$ for the probability of the element corresponding to the leaf $y$. Denote by $\mathscr{L}$ the set of the leaves of $\mathscr{T}$, and by $\mathscr{I}$ the set of its internal nodes. For each $x \in \mathscr{I}$, we define $\mathscr{T}_x$ as the subtree of $\mathscr{T}$ rooted at $x$, and we denote by $\mathscr{L}_x = \mathscr{L} \cap \mathscr{T}_x$ the set of its leaves. The internal nodes $\mathscr{I}$ correspond to the possible positions within a codeword at which a match of the pattern $P$ can be found. In particular, the root $r$ of the tree, which belongs to $\mathscr{I}$, corresponds to the special case where position $i$, returned by the procedure search in the algorithm, is the beginning of a codeword, i.e., a true match has been found.

Consider the fact of having a possible match in a certain position as if it were generated by the following random process: the compressed text consisting of a given sequence of zeros and ones, we pick randomly bit positions which shall act as the starting position of the matches. In this sense, we can speak about the probability of having a possible match in a certain position.

We thus assume that the position $i$ returned by the procedure search occurs at random in any possible location, that is, at any internal node of $\mathscr{T}$. For a given internal node $x \in \mathscr{I}$, the probability $P(x)$ of the position corresponding to $x$ being returned by the algorithm will be proportional to $p_i\ell_i$, and not just to $p_i$, since we deal with a random process on the compressed text and not on the original one. Each leaf of the Huffman tree is associated with a probability $p_i$, and the probability associated with an internal node $y$ is the sum of the probabilities associated with the two children of $y$. Thus, when adding the probabilities associated with all the internal nodes, we get $W = \sum_{i=1}^{n} p_i\ell_i$, the weighted average codeword length, and the probability $P(x)$ is given by

$$P(x) = \frac{\sum_{y \in \mathscr{L}_x} p_y}{W}.$$

This is indeed a probability distribution, as $\sum_{x \in \mathscr{I}} P(x) = 1$.

Similarly, for a leaf $y \in \mathscr{L}$, the probability of seeing the codeword corresponding to $y$ in the compressed text, which we shall denote by $\mathscr{P}(y)$ to differentiate it from the above probabilities defined for internal nodes, will be proportional to $p_i\ell_i$, rather than just to $p_i$, so that this probability will be

$$\mathscr{P}(y) = \frac{p_y\ell_y}{W}$$

and again $\sum_{y \in \mathscr{L}} \mathscr{P}(y) = 1$.

As an example for these definitions, consider again the Huffman code mentioned in Section 1 for the characters T, N, A, O, W, E, B and C, and suppose they occur with probabilities 0.28, 0.19, 0.12, 0.11, 0.11, 0.06, 0.05 and 0.08, respectively. The corresponding Huffman tree is depicted in Fig. 3. The probability associated with any node $v$ of $\mathscr{L} \cup \mathscr{I}$ appears underneath $v$, the probabilities $P(x)$ for $x \in \mathscr{I}$ appear in grey ellipses to the right of the internal (black) nodes, and the probabilities $\mathscr{P}(y)$ for $y \in \mathscr{L}$ appear in white boxes to the left of the leaves.

Let $\mathscr{F}$ denote the event of getting a false match at a given position. We evaluate the probability $P(\mathscr{F})$ by conditioning on the position $x \in \mathscr{I}$ returned by the algorithm:

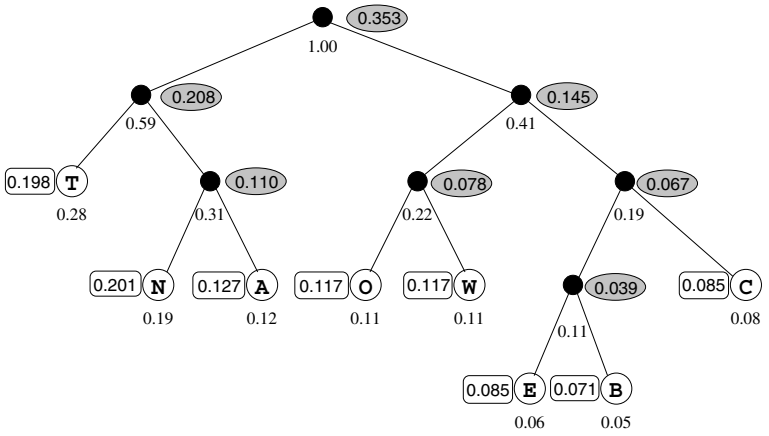


Fig. 3. Probabilities $P(x)$ and $\mathscr{P}(y)$ for the nodes of the example Huffman tree.

$$P(\mathscr{F}) = \sum_{x \in \mathscr{I}} P(\mathscr{F}|\text{algorithm returned}x)P(x).$$

If $x$ is the root, $P(\mathscr{F}|$ algorithm returned $x) = 0$, because of the prefix property of the Huffman codes. If $x$ is some other internal node, we have to consider several possibilities, which are schematically displayed in Fig. 4.

Since we deal with a complete code, any binary sequence such as $\mathscr{E}(P)$ we try to locate, can be decoded (i.e., mapped into a sequence of codewords), even if the traversal of the tree $\mathscr{T}$ does not start at its root. One possibility is that $\mathscr{E}(P)$ is a substring of a codeword, without being its prefix or suffix. This corresponds to a path in $\mathscr{T}$ starting at an internal node $x$ and ending at another internal node $y$ (Fig. 4(a)). Another possibility is that $\mathscr{E}(P)$ is a suffix of a codeword (Fig. 4(b)), or it could be such a suffix followed by several other codewords. The most general case is given in Fig. 4(c): $\mathscr{E}(P)$ consists of the suffix of some codeword, followed by (zero or more) codewords and ending with the proper prefix of some codeword.

Denote by $y(x,1),\ldots,y(x,t-1)$ the sequence of leaves encountered when traversing the tree $\mathscr{T}$, starting at the internal node $x$, and proceeding to left or right children as directed by the binary string $\mathscr{E}(P)$. Let $y'(x,t)$ be the internal node at which this traversal finishes. The case of Fig. 4(a) corresponds to $t = 1$, and if the decoding happens to finish at the end of a codeword (as in the case

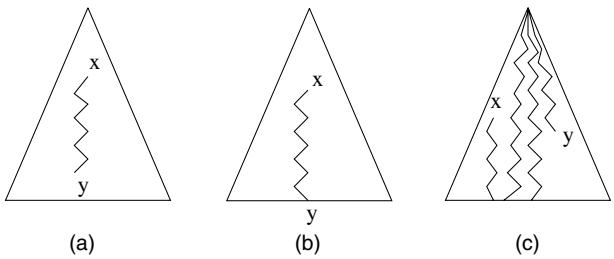


Fig. 4. Schematic view of Huffman tree traversal with $\mathscr{E}(P)$.

of Fig. 4(b)), we define $y'(x,t)$ as being the root. For each of the leaves $y(x,i)$, the probability of seeing it in the encoded text is $\mathscr{P}(y(x,i))$, and the probability of seeing the prefix corresponding to $y'(x,t)$ is the sum of the probabilities of the leaves in the subtree rooted by $y'(x,t)$. For the special case $y'(x,t) = \text{root}$, this sum is 1. Assuming independence of the events, we get

$$P(\mathscr{F} \mid \text{algorithm returned } x) = \prod_{i=1}^{t-1} \mathscr{P}(y(x,i)) \left( \sum_{j \in \mathscr{L}_{y'(x,t)}} \mathscr{P}(j) \right).$$

We can therefore derive the estimated number of false matches $|\mathscr{E}(T)|P(\mathscr{F})$ as

$$|\mathscr{E}(T)| \sum_{x \in \mathscr{I}} \left( \prod_{i=1}^{t-1} \frac{p_{y(x,i)} \ell_{y(x,i)}}{W} \left( \sum_{j \in \mathscr{L}_{y'(x,t)}} \frac{p_j \ell_j}{W} \right) \right) \left( \frac{\sum_{z \in \mathscr{L}_x} p_z}{W} \right). \tag{2}$$

In any case, we see that the probability of a false match decreases sharply when the length $m = |\mathscr{E}(P)|$ increases. We bring below experimental results comparing the formulas with empiric data. It should be noted that one can argue that similarly, the expected number of *true* matches can be evaluated; but true matches of $\mathscr{E}(P)$ in $\mathscr{E}(T)$ correspond to matches of $P$ in $T$, and these are given since $P$ and $T$ are fixed. There is therefore no probabilistic scenario on which calculating this probability could be based. For the *false* matches, however, our assumption of random occurrence seems reasonable, yielding the above analysis.

### 3.2. False matches resulting from probabilistic pattern matching

We stated above that if $x$, the node of the Huffman tree corresponding to the position returned by the algorithm, is the root, then $P(\mathscr{F} \mid \text{algorithm returned } x) = 0$, i.e., there cannot be a false match, because the encoded pattern has been found at a codeword boundary and a false match would imply a violation of the prefix property. However, this assumes that we can assure that if the pattern matching algorithm declares a match at position $i$, there is indeed a match at that position. This is true for deterministic algorithms, but not necessarily for probabilistic ones. For instance, Karp and Rabin's (1987) algorithm searches for $\mathscr{E}(P)$ in $\mathscr{E}(T)$ by scanning substrings $Z_i$ of $\mathscr{E}(T)$, each of the same length $m$ as the encoded pattern, and instead of comparing $Z_i$ with $\mathscr{E}(P)$, it compares $Z_i \bmod Q$ with $\mathscr{E}(P) \bmod Q$, where $Q$ is a large randomly chosen prime number. If the moduli are equal, a match is declared, even though obviously there are many numbers $a$ and $b$ such that $a \neq b$ but $a \bmod Q = b \bmod Q$. Two probabilities have thus to be dealt with:

1. the probability $R_1$ that the match declared by the probabilistic pattern matcher might be an erroneous one;
2. the probability $R_2$ that even if there is a true match of $\mathscr{E}(P)$ at the declared position within $\mathscr{E}(T)$, it might not correspond to a match of $P$ in $T$.

The second probability $R_2$ has been evaluated in the previous section. As to $R_1$, note that one can easily turn the probabilistic algorithm into a deterministic one, by checking at the declared position if it indeed holds a match. Moreover, such a check is generally not really needed. The probability to get a false match by the Karp Rabin algorithm is bounded by $mn/2^q$, where $m$ and $n$

are the sizes of the pattern and the scanned text, respectively, and $q = \lceil \log_2 Q \rceil$ is the number of bits of the prime number $Q$. One can therefore choose $q$ large enough (since $m$ and $n$ are given) to make this probability negligible relative to $R_2$. Summarizing, we may safely ignore $R_1$: if the pattern $\mathscr{E}(P)$ is shorter than $q$, then working modulo $Q$ is in fact a real comparison and not a probabilistic one, so $R_1 = 0$; if on the other hand, $m$ is larger than $q$, then $R_2$ will probably be extremely small.

### 3.3. Erroneous decisions of the algorithm

When running the pattern matching algorithm with the backskips, a correct performance identifies the true and false matches for each of the occurrences of $\mathscr{E}(P)$ in $\mathscr{E}(T)$. These matches are called below true positives and negatives. The algorithm can, however, also fail in two quite different ways:

1. It could announce a match, while in reality the occurrence of $\mathscr{E}(P)$ in $\mathscr{E}(T)$ does not correspond to an occurrence of $P$ in $T$ (false positives);
2. It could fail to announce a match, while in reality the occurrence of $\mathscr{E}(P)$ in $\mathscr{E}(T)$ does correspond to an occurrence of $P$ in $T$ (false negatives).

In an analogy to information retrieval literature, we wish to retrieve all, and only, occurrences of $P$ in $T$. The first type of error reduces then *precision* (the ratio of relevant retrieved items to all retrieved items), since it retrieves also elements which are *not* occurrences of $P$ in $T$; the second type of error reduces *recall* (the ratio of relevant retrieved items to all relevant items), since it does not retrieve *all* occurrences of $P$ in $T$. The following table summarizes the four possibilities for a given occurrence of $\mathscr{E}(P)$ in $\mathscr{E}(T)$: the columns correspond to the actual situation (match or non-match of $P$ in $T$), the rows to what is announced by the algorithm.

| | actual match | actual non-match |
|---|---|---|
| declare match | True positives | False positives |
| declare non-match | False negatives | True negatives |

If the algorithm does not skip backwards ($K = 0$), every occurrence of $\mathscr{E}(P)$ in $\mathscr{E}(T)$ would be declared as a match, so there are no false negatives, but possibly many false positives. For small values of $K$, it is probable that synchronization with the true decoding is not achieved before the $K$ bits are used up, which could imply a large number of false decisions; but by increasing $K$, the probability of synchronization, regardless of the starting point, increases, and the number of false positives or negatives will decrease.

## 4. Experimental results

The algorithm was tested on several files of different nature. The first one was `paper1` from the Calgary corpus, an English text with editing instructions. The second was a DNA file (of the

tobacco chloroplast genome), including also blanks and newlines for clarity; the alphabet thus consisted of six characters. Finally, to cancel the bias introduced by the independence assumption, a new text was created based on the distribution of characters in `paper1`, but with each character generated independently from the others.

The first set of tests was performed on `paper1`, searching for arbitrary patterns which were chosen randomly within the file. We used a canonical Huffman encoding and considered patterns of lengths 3–50, four of each. The compressed forms of these 192 patterns occurred in total 1077 times in the compressed text. Of these occurrences, 1040 corresponded to appearances of $P$ in $T$, and 37 were false matches, all of which occurred for the shorter patterns up to length 5. For example, when searching for $P = \texttt{fro}$, for which $\mathscr{E}(P) = 110010\text{-}10001\text{-}0011$, the pattern $P' = \texttt{k}\sqcup\texttt{h}$ was retrieved ($\sqcup$ denoting a blank), for which $\mathscr{E}(P') = 111100101\text{-}000\text{-}10011$, including $\mathscr{E}(P)$ as suffix.

The algorithm was then applied several times, each time with a different size of the backward skip, which was chosen as an integral number of bytes. The first column of Table 1 gives the size $K$ of the backward skip in bits, the following columns list the number of occurrences of true and false positives and negatives.

Obviously, true positives and false negatives add up to the number of actual matches, while true negatives and false positives add up to the number of non-matches. One sees that false positives (wrongly announced matches) are rare, independently of the algorithm, and that already for small backskips (less than 12 bytes for all our examples), both types of errors may be corrected.

Table 2 brings some example patterns $P$, comparing for each the actual number of wrong matches (occurrences of $\mathscr{E}(P)$ in $\mathscr{E}(T)$ which do not correspond to occurrences of $P$ in $T$) with the expected number on the basis of formulas (1) and (2).

We see that there is generally a good fit, with no formula consistently outperforming the other. Interestingly, on the random file, the number of wrong matches was much higher than the number of true matches for many of the examples.

Table 1
Number of matches and false matches as a function of backward skip

| $K$ in bits | True positives | True negatives | False positives | False negatives |
|---|---|---|---|---|
| 8 | 415 | 35 | 2 | 625 |
| 16 | 670 | 33 | 4 | 370 |
| 24 | 825 | 36 | 1 | 215 |
| 32 | 917 | 35 | 2 | 123 |
| 40 | 974 | 35 | 2 | 66 |
| 48 | 1013 | 37 | 0 | 27 |
| 56 | 1018 | 36 | 1 | 22 |
| 64 | 1036 | 37 | 0 | 4 |
| 72 | 1038 | 36 | 1 | 2 |
| 80 | 1036 | 36 | 1 | 4 |
| 88 | 1039 | 37 | 0 | 1 |
| 96 | 1040 | 37 | 0 | 0 |
| ... | | | | |
| Start of file | 1040 | 37 | 0 | 0 |

Table 2
Empiric and expected number of false matches

| File | Pattern | $\lvert\mathscr{E}(P)\rvert$ | # Wrong matches | Estimate (1) | Estimate (2) |
|---|---|---|---|---|---|
| paper1 | in | 8 | 621 | 1018 | 624 |
| | cl | 10 | 186 | 260 | 192 |
| | ies | 13 | 36 | 32 | 33 |
| | lose | 18 | 0 | 1 | 1 |
| | Incre | 26 | 0 | 0.004 | 0.004 |
| dna | tcg | 8 | 1783 | 1797 | 2483 |
| | atct | 9 | 1205 | 909 | 1094 |
| | aaag␣a | 14 | 35 | 29 | 34 |
| | gatactc | 17 | 5 | 4 | 4 |
| random | et | 8 | 858 | 1030 | 603 |
| | u␣ | 9 | 426 | 518 | 547 |
| | f␣o | 13 | 31 | 32 | 14 |
| | eo, e | 19 | 0 | 0.5 | 0.6 |

## 5. Concluding remarks

Searching for a pattern directly in a Huffman encoded file seems to be an easy task because of the static nature of the compression scheme. There are however synchronization problems, which we tried to overcome in this work. If the pattern is long enough, the probability of finding a wrong match is often very low, independently of the algorithm. For the other patterns, a proper choice of the backskip parameter lets us control the error.

## Acknowledgements

The authors would like to thank two anonymous referees for their helpful comments.

## References

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

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.

Boyer, R., & Moore, S. (1977). A fast string searching algorithm. *Communications of the ACM, 20*, 762–772.

DeMoura, E. S., Navarro, G., Ziviani, N., & Baeza-Yates, R. (1998). Direct pattern matching on compressed text. In *Proceedings of string processing and information retrieval (SPIRE-98)* (pp. 90–95). IEEE CS Press.

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

Farach, M., & Thorup, M. (1995). String matching in Lempel–Ziv compressed strings. In *Proceedings of 27th annual ACM symposium on the theory of computing* (pp. 703–712).

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

Fraenkel, A. S., Mor, M., & Perl, Y. (1983). Is text compression by prefixes and suffixes practical? *Acta Informatica, 20*, 371–389.

Fukamachi, S., Shinohara, T., & Takeda, M. (1992). String pattern matching for compressed data using variable length code. Efficient retrieval of Genome information. In *Proceedings of symposium on informatics* (pp. 95–103).

Gąsieniec, L., & Rytter, W. (1999). Almost optimal fully LZW-compressed pattern matching. In *Proceedings of data compression conference DCC-99, Snowbird, Utah* (pp. 316–325).

Gilbert, E. N., & Moore, E. F. (1959). Variable-length binary encodings. *The Bell System Technical Journal, 38*, 933–968.

Kärkkäinen, J., Navarro, G., & Ukkonen, E. (2000). Approximate string matching over Ziv–Lempel compressed text. In *Lecture Notes in Computer Series: 1848. Proceedings of 11th annual symposium on combinatorial pattern matching CPM-00* (pp. 195–209). Springer Verlag.

Karp, R., & Rabin, M. (1987). Efficient randomized pattern matching algorithms. *IBM Journal of Research and Development, 31*, 249–260.

Kida, T., Takeda, M., Shinohara, A., & Arikawa, S. (1999). Shift-And approach to pattern matching in LZW compressed text. In *Lecture Notes in Computer Science: 1645. Proceedings of 10th annual symposium on combinatorial pattern matching CPM-99* (pp. 1–13). Springer Verlag.

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

Klein, S. T., & Wiseman, Y. (2000). Parallel Huffman decoding. In *Proceedings of data compression conference DCC-2000, Snowbird, Utah* (pp. 383–392).

Klein, S. T., Bookstein, A., & Deerwester, S. (1989). Storing text retrieval systems on CD-ROM: compression and encryption considerations. *ACM Transactions on Information Systems, 7*, 230–245.

Longo, G., & Galasso, G. (1982). An application of informational divergence to Huffman codes. *IEEE Transactions on Information Theory, IT-28*, 36–43.

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

Navarro, G., & Raffinot, M. (1999). A general practical approach to pattern matching over Ziv–Lempel compressed text. In *Lecture Notes in Computer Science: 1645. Proceedings of 10th annual symposium on combinatorial pattern matching CPM-99* (pp. 14–36). Springer Verlag.

Shibata, Y., Kida, T., Takeda, M., Shinohara, T., & Arikawa, S. (2000). Speeding up pattern matching by text compression. In *Lecture Notes in Computer Science: 1767. Proceedings of the 4th Italian conference on algorithms and complexity* (pp. 306–315). Springer Verlag.

Shibata, Y., Matsumoto, T., Takeda, M., Shinohara, A., & Arikawa, S. (2000). A Boyer–Moore type algorithm for compressed pattern matching. In *Lecture Notes in Computer Science: 1848. Proceedings of 11th annual symposium on combinatorial pattern matching CPM-00* (pp. 181–194). Springer Verlag.

Takeda, M., Miyamoto, S., Kida, T., Shinohara, A., Fukamachi, S., Shinohara, T., & Arikawa, S. (2002). Processing text files as is: pattern matching over compressed texts, multi-byte character texts, and semi-structured texts. In *Proceedings of string processing and information retrieval (SPIRE-02)* (pp. 170–186).

Witten, I. H., Moffat, A., & Bell, T. C. (1994). *Managing gigabytes, compressing and indexing documents and images*. London: International Thomson Publishing.