Compressed Pattern Matching in JPEG Images

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

[1] Dept. of Computer Science
Bar Ilan University
Ramat-Gan 52900, Israel
e-mail: tomi@cs.biu.ac.il

[2] Dept. of Computer Science
Ashkelon Academic College
Ashkelon, Israel
e-mail: shapird@acad.ash-college.ac.il

Keywords: Data Compression, JPEG, Huffman

Abstract. The possibility of applying compressed matching in JPEG encoded images is investigated and the problems raised by the scheme are discussed. A part of the problems can be solved by the use of some auxiliary data which yields various time/space tradeoffs. Finally, approaches to deal with extensions such as allowing scaling or rotations are suggested.

1 Introduction

The paradigm of *compressed pattern matching* has recently gotten a lot of attention. The idea of the compressed matching was first introduced in the work of Amir and Benson [1] as the task of performing pattern matching in a compressed text without decompressing it. For a given text T and pattern P and complementary encoding and decoding functions, $\mathcal{E}$ and $\mathcal{D}$ respectively, our aim is to search for $\mathcal{E}(P)$ in $\mathcal{E}(T)$, rather than the usual approach which searches for the pattern P in the decompressed text $\mathcal{D}(\mathcal{E}(T))$. Amir and Benson deal with a run-length encoded two-dimensional pattern, but most works address the problem of finding one-dimensional patterns in files compressed by various methods, such as Huffman coding [9], Lempel-Ziv [13], or specially adapted methods [11, 8].

We concentrate here on two-dimensional compressed matching in which the given text is an image encoded by the standard JPEG baseline scheme [6] and the pattern consists of a given image fragment we are looking for. In a more general setting, a collection of images could be given, and the subset of those including at least one copy of the pattern is sought. An example for the former could be an aerial photograph of a city in which a certain building is to be located, an example for the more general case could be a set of pictures of faces of potential suspects, which have to be matched against some known identifying feature like a nose or an eyebrow.

Baseline JPEG uses a static Huffman code, without which compressed matching would not always be possible, since our underlying assumption is that all occurrences

of the pattern are encoded by the same binary sequence. This is not the case for dynamic Huffman coding or for arithmetic coding. Lempel-Ziv methods are also adaptive, but for them compressed matching is possible because all the fragments of the pattern appear in the text, though not necessarily in the same order as in the pattern.

In a first approach, we accept as simplifying assumption that only exact copies of the pattern are to be found. Returning to the example of the aerial picture, it would of course also be interesting to locate the given building if the pattern presents it in a different angle than it appears in the larger image, or at another scale, or even only partially, because it could have been occluded by a cloud when the picture has been taken. The corresponding pattern matching problems, allowing rotations, scaling and occlusions, are more difficult and have been treated in [2, 3].

In the next section, we review the basic ingredients of the JPEG algorithm, then turn in Section 3 to the method we suggest for compressed matching in JPEG files. The main problem to be dealt with is one of synchronization and alignment, so we explore in Section 4 the possibility of using auxiliary files to solve such alignment problems. The last section deals with extensions to rotations and scaling.

2 The JPEG standard

JPEG [6] is a lossy image compression method. In a first step, the picture is split into a sequence of blocks of size 8×8 pixels. Each block is then compressed by the following sequence of transformations:

1. Applying a *Discrete Cosine Transform* (DCT) [14] to the set of 64 values of the pixels in the block;

2. Applying *Quantization* to the DCT coefficients, thereby producing a set of 64 smaller integers. This step causes a loss of information but makes the data more compressible;

3. Applying an *entropy encoder* to the quantized DCT coefficients. Baseline JPEG uses Huffman coding in this step, but the JPEG standard specifies also arithmetic coding as possible alternative.

The decompression process just reverses the actions and their order. It first applies Huffman decoding, then dequantizes the coefficients, and finally uses an inverse DCT to obtain a set of values. Because of the quantization step, the reconstructed set includes only approximated values.

The coefficient in position (0,0) (left upper corner) is called the DC coefficient and the 63 remaining values are called the AC coefficients. In principle, the DC coefficient should store a measure of the average of the 64 pixel values of the given block, but since there is usually a strong correlation between the DC coefficients of adjacent blocks, what is actually stored is the difference between the average in this block and the average in the previous one.

Baseline JPEG uses two different Huffman trees to encode the data. The first encodes the lengths in bits (1 to 11) of the binary representations of the values in the DC fields. The second tree encodes information about the sequence of AC coefficients.

As many of them are zero, and most of the non-zero values are often concentrated in the upper left part of the 8×8 block, the AC coefficients are scanned in a fixed zig-zag order, processing elements on a diagonal close to the upper left corner before those on such diagonals further away from that corner; that is, the order is given by (0,1), (1,0), (2,0), (1,1), (0,2), (0,3), (1,2), etc. The second Huffman tree encodes pairs of the form (n, ℓ), where n (limited to the range 0 to 15) is the number of elements that are 0, preceding a non-zero element in the given order, and ℓ is the length in bits (1 to 10) of the binary representation of the non-zero quantized AC value. The second tree includes also codewords for End of Block (EOB), which is used when no non-zero elements are left in the scanning order, and for a sequence of 16 consecutive 0s in the AC sequence (ZRL). The Huffman trees used in baseline JPEG are static, and can be found in [15].

Each 8×8 block is then encoded by an alternating sequence of Huffman codewords and binary integers (except that the codewords for EOB and ZRL are not followed by any integer), the first codeword belonging to the first tree and relating to the DC value, the other codewords encoding the (n, ℓ) pairs for the AC values, with the last codeword in each block representing EOB. Figure 1(a) brings an example block of quantized values, with the DC value in boldface in the upper left corner. The upper part of Figure 1(b) shows the encoding of this block, with elements to be Huffman encoded appearing in parentheses, and the elements corresponding to DC (the value of which we assume to be 5) bold faced; the binary translation of the encoding, with framed Huffman codewords, is shown underneath.

20	1	0	0	0	0	0	0
0	3	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-2	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

(a) Typical JPEG block

(3) **5** (0,1) 1 (2,2) 3 (4,2) -2 (EOB)

100 101 00 1 11111001 11 1111111000 01 1010

(b) Encoding of JPEG block

FIGURE 1: *Example of JPEG block and its encoding*

JPEG encodes the blocks row by row, from left to right, and concatenates the encoded blocks. A small header encodes the number of rows and columns, so there is no need to encode an end-of-row indicator specifically. Actually, to simplify the discussion and the examples, our description refers to only one component, the *luminance*, of JPEG encoding, which corresponds to black and white images. JPEG also supports color images, where each color pixel is split into several components (RGB or YUV).

3 Pure compressed matching

We are given an image T of $n \times k$ pixels, in which a two-dimensional pattern P of size $m \times \ell$ pixels should be found. Since JPEG works with 8×8 pixel blocks, we

assume that n and k are multiples of 8. The compressed matching starts by encoding the pattern using the same JPEG algorithm as the one used for the original image. Even then we cannot assure that a pattern can be located, as the 8×8 blocks of the pattern are not necessarily aligned with those of the image. The search process has therefore to be repeated 64 times, positioning, for each matching attempt, the leftmost uppermost pixel of the first 8×8 block in the pattern at the ith pixel in the jth row, $1 \leq i, j \leq 8$. Figure 2 is an example of how the pattern could be partitioned: there will usually be a frame at the border of the pattern (the darker area in Figure 2) corresponding to 8×8 blocks that fit only partially. The pixels in this frame will not participate in the matching process, so the pattern is actually restricted to an area of full contiguous 8×8 blocks (the white area in Figure 2). For the rest of this paper, let m and ℓ then represent the dimensions of the restricted pattern, that is, m and ℓ are also multiples of 8.

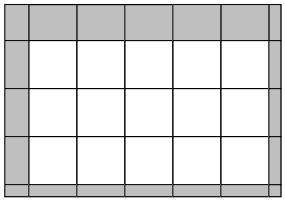


FIGURE 2: *Example of partition of the pattern into* 8×8 *blocks*

The first block is JPEG encoded, yielding one DC value and a sequence of AC values. Since DC elements are encoded relative to preceding blocks, the DC value of this first block cannot be located, so the matching starts only from the beginning of the sequence of AC values. These are calculated for each block independently, therefore if the pattern-block does appear as a block in the image T, the encoded sequence of AC values will appear in the encoded image $\mathcal{E}(T)$. The DC values of the second and subsequent blocks in the first row of 8×8 blocks of P can be evaluated based on the DC values of the preceding blocks, hence the first part of the encoded pattern to be searched for in $\mathcal{E}(T)$ consists of the sequence of AC values followed by the $\ell/8 - 1$ encoded 8×8 blocks of the first row.

The compressed matching paradigm raises then several problems. First, suppose that the binary sequence encoding the first part of the pattern is indeed located. This does not necessarily mean that an occurrence of the encoded elements is found, as the beginning of the Huffman codewords might not be synchronized. Consider, for example, the block **(2) 3** (0,2) −3 (EOB), to be located in a sequence of several blocks identical to those of the example in Figure 1(b). Figure 3 shows that the pattern (after having stripped the DC values) will be found erroneously crossing the block boundaries in $\mathcal{E}(T)$.

The same problem was noted in [9], and in [10] in an application to parallel decoding of a JPEG file when several processors are available. For long enough patterns, the tendency of Huffman codes to resynchronize after errors may suggest that false alarms as those in the above example might be rare, but in our application, the rows of the pattern may be short. Moreover, the problem in the JPEG case is more severe than for plain Huffman decoding. For the latter, once synchronization has been regained, the remainder of the encoded file is correct. In JPEG files, on

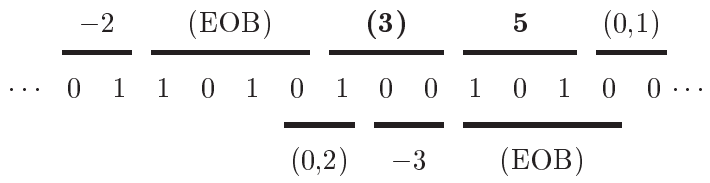


FIGURE 3: *Example of false alignment*

the other hand, consisting of both Huffman codewords and integer encodings, the fact that a given bit is the last in a codeword for both the correct and the erroneous decoding does not imply that both decodings will continue identically. Referring again to Figure 3, the codewords for **(3)** and −3 end at the same bit, which is nevertheless not a synchronization point.

The second problem is that the encoded pattern does not appear consecutively in the encoded image (unless $k = \ell$), but with gaps corresponding to the encoding of $(k - \ell)/8$ blocks. The pattern is therefore partitioned into $m/8$ sub-patterns, each corresponding to a row of $\ell/8$ blocks, and with the first DC value of each sub-pattern eliminated. If the sub-patterns are located using some pattern matching algorithm, we cannot conclude with certainty that the pattern has been found. In addition to the above problem of possible false alignments, one cannot know if each of the gaps are indeed the encoding of $(k - \ell)/8$ blocks, even if the sub-patterns are found in the required order and even if all of them are true matches.

One of the possible solutions could be, once the first row of the pattern has been found, to continue decompressing the image, keeping a count of the decoded blocks. In other words, pattern matching would only be used for the first row of the pattern, then the image would be decoded sequentially. In fact, one does not really need full decoding: the Huffman codewords in the JPEG file indicate the length in bits of the integers following the codewords, and for our purpose, these integers can be simply skipped. This solution could, however, not really be considered as compressed matching, since, depending on the position of the first occurrence of the pattern in the encoded file, large parts of it, possibly almost the whole original file, are decompressed.

The third problem relates to the fact that there are possibly many occurrences of the pattern, perhaps even overlapping ones. In images this might be more frequent than for plain texts, because large areas could represent some uniform background (a blue sky, dark parts in the shadow, etc.), and therefore consist of many identical blocks. If each of the rows of the pattern is located several times, we need to match somehow their occurrences to check whether indeed we have an occurrence of the whole pattern. This might be a difficult task even if we ignore the problem of certain occurrences being false matches.

We therefore conclude that compressed pattern matching in JPEG files is hard to achieve, unless we keep some auxiliary data, as suggested in the following section.

4 Compressed matching with auxiliary data

The task would be much easier if one would know, for a given position in the JPEG encoded file, the index of the corresponding 8×8 block in the original file. A step in this direction would be using synchronizing codewords (see [7]), for example at the

end of each encoded row, but this would require a change in the encoding standard, for example to JPEG-2000 [12] which has synchronizing codewords built-in. In fact, the code used in baseline JPEG is not really a Huffman code, because it is not complete: there is, e.g., no codeword consisting only of 1's. This can be exploited to devise a *synchronizing sequence*: the longest sequence of 1's that can appear is of length 29, in the encoding of (10,10) 1023 (15,10), which is translated into 1111111111001111 1111111111 1111111111111110. Therefore a sequence of 30 consecutive 1's is synchronizing. This synchronizing sequence could be inserted at the end of each row, which could therefore be detected without decoding. Alternatively, instead of wasting 30 bits for synchronization, one of the codewords could be replaced by this string of 1's, for example the codeword 1010 for EOB. This would increase each encoded block by 26 bits, but false matches are then easily detected. Nevertheless, 26 bits for each 8×8 block, which are generally encoded by a few hundred bits or less, might be too high a price to pay.

4.1 Building an index

Instead of modifying the JPEG file, one could construct a table S, acting as an index, that would be stored in addition to the original compressed file. $S(i)$ would be the bit-position, within the JPEG image, of the beginning of the encoding of the AC sequence in the ith block, that is the index of the bit following the DC value. The size of each entry in S would be $\lceil \log_2 |\mathcal{E}(T)| \rceil$, where $|x|$ refers to the size of x in bits, so that a 3 byte entry could accommodate a compressed image of size up to 2MB. The number of entries is S is $nk/64$, the number of 8×8 blocks in T.

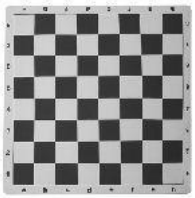
The construction of such an index has to be done in a preprocessing stage, and it could be argued that this contradicts the main idea of compressed matching, since while building the table S one actually decompresses the whole image. Nevertheless, the preprocessing can be justified in certain applications, for example when one large image will be used many times for searches with different patterns. This is similar to regular pattern matching with a fixed large text of size n and possibly many patterns to be looked for. Some of the fastest algorithms are then based on constructing a *suffix tree* [16, 4], the size of which may often exceed that of the text itself. Construction time is linear in n, but once the suffix tree is ready, the time to locate a pattern is independent of the size of the text.

The index S can be used to solve some of the problems mentioned above. Once the encoding of the first row of the pattern image has been located in $\mathcal{E}(T)$ at bit offset y, a binary search for y in S can decide in $\lceil \log n + \log k \rceil - 6$ comparisons whether the match is a true one. Similar searches for the following rows of the pattern can locate all the rows, without decoding.

To get a feeling about the size of the required indices, we have applied this idea on the three grayscale sample JPEG files in Figure 4: the classical Lenna picture, a chessboard with many identical sub-parts, and a rose. Table 1 shows the details, giving the number of rows and columns, $r \times c$, the size in bytes, s, of the compressed file, and the absolute (in bytes) and relative size (in percent) of the index S. The size is given by $((\lceil r/8 \rceil \cdot \lceil c/8 \rceil)(\log_2(s) + 3))/8$.

If the size of S is too large, a time/space tradeoff can be obtained by fixing an integer parameter d and storing only every dth entry of S. The storage overhead is

Lenna



Chess

Rose

FIGURE 4: *Examples of JPEG files*

File	pixels	jpeg size	index size	%
Lenna	256 × 256	30,763	2304	7.5
Chess board	150 × 150	14,112	768	5.4
Rose	227 × 149	12,089	1171	9.7

TABLE 1: *Details on sample files*

reduced by a factor of d, at the cost of increased search time: the binary search for the bit offset y now locates the largest value is S that is still smaller or equal to y; from there, up to d blocks have to be decoded. For example, the index for the Lenna picture can be reduced to less than 1% if only every eighth block is indexed, and if one records only the beginning of every row, the index reduces to 72 bytes.

4.2 Dealing with multiple matches

We now turn to the possibility of having found many matches for each of the rows of the pattern. Using the table S, each of the found offsets is checked to correspond to a true match and then translated to a block index. Since the dimensions of the image T are known, each index can be translated into an (r, c) pair, denoting the indices of the corresponding row and column. Let (R_i, C_i) be the sequence of n_i (true) occurrences of the ith row of the pattern,

$$(R_i, C_i) = \{(r_{i1}, c_{i1}), (r_{i2}, c_{i2}), \ldots, (r_{in_i}, c_{in_i})\}, \quad 1 \leq i \leq m.$$

The sequences can be kept in lexicographically increasing order. We need to check whether consecutive rows of the pattern have appearances in consecutive rows and identical columns of the image. Formally, we seek

$$\bigcap_{i=1}^{m} (R_i - i + 1, C_i),$$

where we use the notation $A - x$ for a set of integers $A = \{a_1, \ldots, a_n\}$ and an integer x to stand for the set $\{a_1 - x, \ldots, a_n - x\}$.

The following algorithm uses m pointers, one for each of the sequences, to find all the occurrences:

Repeat until one of the sequences is exhausted
    find the smallest element (r, c) in $(R_1, C_1) \cap (R_2 - 1, C_2)$ by sequential search
    for $i \leftarrow 3$ to m
        search for an occurrence of (r, c) in $(R_i - i + 1, C_i)$
    if (r, c) is common to all m sequences, increase all m pointers by 1

The search in the iterative step can be done by binary search, since the sequences are ordered, but this is not necessarily the best solution. Consider the special case in which all n_i are equal to n_1, and h elements are found in the intersection $(R_1, C_1) \cap (R_2 - 1, C_2)$. Assume also that $h > n_1/\log n_1$ and that all h elements of the intersection belong to the first halves of both (R_1, C_1) and $(R_2 - 1, C_2)$. Then performing the intersection takes $2n_1$ comparisons, and each of the h searches in each of the $m-2$ remaining sequences requires $\log n_1$ comparisons. To reduce this number even by 1, the length of the sequence has to be cut at least to half, so even reducing the search to the remaining sequence after each located element wouldn't help in our case. The total search time would thus be $2n_1 + h\,(m-2)\log n_1 > n_1 m$. On the other hand, scanning the m lists sequentially can be done in time $n_1 m$.

Note that it would pay to start the process by intersecting the two shortest lists, rather than the two first, which would tend to reduce h. Moreover, the intersection could be done by binary merge [5] rather than linearly.

In an experiment run on each of the images of Figure 4, a random 15 byte long fragment of the encoded file was taken as pattern, corresponding to a part of a row of the image, and occurrences of this pattern were sought. In each case, only a single occurrence was found, corresponding to the true match. This suggests that in many real life JPEG files, multiple matches will not cause a problem. On the other hand, we repeated the test with a pure black bitmap file, and found there many matches, as expected.

5 Matching with scaling and rotations

Consider the problem of locating the pattern P after having scaled it by a factor α and/or rotated it by an angle γ. The one to one correspondence between 8×8 blocks of pattern and image might be lost, but since the DCT transforms the full block as one indivisible entity, there is no way to detect the encoding of parts of the block in the JPEG file. So instead of trying to transform the encoded pattern, one has to transform the pattern first, and then apply the encoding.

For $\alpha < 1$, both height and width of the occurrence of pattern P in the image T should be α times smaller than in P. Since it is the pattern that is encoded, we get the requested effect by *enlarging* the pattern by a factor of $\beta = 1/\alpha$ before applying JPEG. If β is an integer, one could duplicate each pixel in each row, as well as the such enlarged rows β times. The resulting pattern is of lower quality than a possible occurrence in the given image, so some smoothing, taking neighboring pixels into account, could improve the search, but the DCT will take care, at least partially, of the smoothing anyway. If β is not an integer, certain rational factors can be obtained by a process similar to the one depicted in Figure 5(a). For $\beta = 1.5$, transform each 2×2 block into a 3×3 block, inserting the missing values (in grey) by interpolation.

If $\alpha > 1$, the pattern has to be reduced by a factor of $\beta = 1/\alpha$. If α is an integer, the simplest way to proceed is taking every αth pixel in both dimensions. A more precise way would be to consider some or all translations of such subsets of the pattern having their pixels α positions apart, and averaging among them the value for each pixel. For certain non-integer values of α, one could proceed similarly to the above non-integer case for β.

As to rotations, if γ is a multiple of a right angle, say 90°, 180° or 270°, each

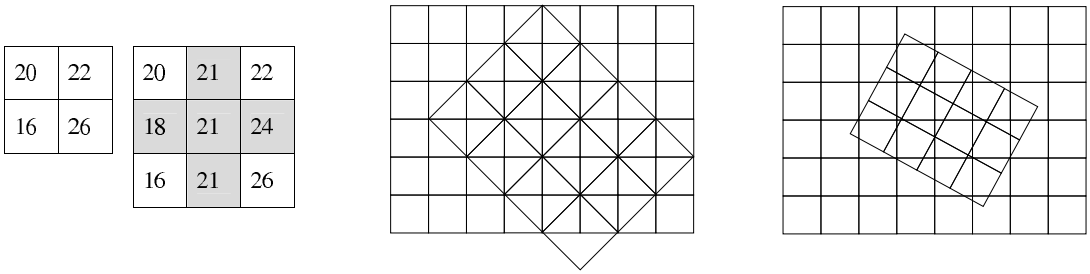


FIGURE 5: *Examples of possible rotations*

8×8 matrix can be transposed or reversed accordingly, thereby redefining the rows and columns of the pattern. If $\gamma = 45°$ after a scaling of $\alpha = \sqrt{2}$, as in Figure 5(b), each pattern block would have to match four halves of image blocks, but even if there is no such regularity and the pattern blocks might intersect a varying number of image blocks in various layouts, as for example in Figure 5(c), one can deal with it by rotating first the pattern by $-\gamma$, then partitioning into blocks and encoding.

Conclusion

Searching directly in JPEG encoded images seems to be a difficult task because the blocking used, as well as the DCT applied to the blocks, does not allow any interaction between adjacent blocks. Using an index, the size of which can be controlled in a time/space tradeoff, may alleviate some of the problems.

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., BUTMAN A., CROCHEMORE M., LANDAU G.M., SCHAPS M., Two dimensional pattern matching with rotations, *Theoretical Computer Science*, **314**(1–2) (2004) 173–187.

[3] AMIR A., BUTMAN A., LEWENSTEIN M., PORAT E., Real two dimensional scaled matching, *Proc. WADS* (2003) 353–364.

[4] APOSTOLICO A., The myriad virtues of subword trees, *Combinatorial Algorithms on Words,* NATO ASI Series Vol **F12**, Springer Verlag, Berlin (1985) 85–96.

[5] HWANG F.K., LIN S., A simple algorithm for merging two disjoint linearly-ordered sets, *SIAM Journal of Computing* **1** (1972) 31–39.

[6] ISO/IEC 10918-1 Information Technology - Digital Compression and Coding of Continuous–Tone Still Images Requirements and Guidelines, *International Standard ISO/IEC*, Geneva, Switzerland (1993).

[7] Ferguson T.J., Rabinowitz J.H., Self-synchronizing Huffman codes, *IEEE Trans. on Inf. Th.* **IT–30** (1984) 687–693.

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

[9] Klein S.T., Shapira D., Pattern Matching in Huffman Encoded Texts, *Information Processing and Management* **41** (2005) 829–841.

[10] Klein S.T., Wiseman Y., Parallel Huffman Decoding with Applications to JPEG Files, *The Computer Journal* **46**(5) (2003) 487–497.

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

[12] Marcellin M.W., Gormish M.J., Bilgin A., Boliek M.P., An Overview of JPEG-2000, *Proc. Data Compression Conference DCC-2000*, Snowbird, Utah (2000) 523–541.

[13] Navarro G., Raffinot M., A general practical approach to pattern matching over Ziv-Lempel compressed text, *Proc. 10th Symp. on Combinatorial Pattern Matching,* Warwick, UK, July 22–24 1999, *LNCS* **1645**, Springer Verlag, Berlin(1999) 14–36.

[14] Rao K.R., Yip P., *Discrete Cosine Transform Algorithms, Advatages, Applications*, Academic Press Inc., London (1990).

[15] Wallace G.K., The JPEG Still Picture Compression Standard, *Communication of the ACM* **34** (1991) 30–44.

[16] Weiner P., Linear pattern matching algorithms, *Proc. 14th Annual IEEE Symposium on Switching and Automata Theory*, Washington, DC, (1973) (1–11).