Compressed Matching for Feature Vectors*

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

[a]*Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel*
tomi@cs.biu.ac.il

[b]*Department of Computer Science and Mathematics, Ariel University, Ariel 40700, Israel*
shapird@ariel.ac.il

Abstract

The problem of compressing a large collection of feature vectors is investigated, so that object identification can be processed on the compressed form of the features. The idea is to perform matching of a query image against an image database, using directly the compressed form of the descriptor vectors, without decompression. Specifically, we concentrate on the Scale Invariant Feature Transform (SIFT), a known object detection method, as well as on Dense SIFT and PHOW features, that contain, for each image, about 300 times as many vectors as the original SIFT. Given two feature vectors, we suggest achieving our goal by compressing them using a lossless encoding by means of a Fibonacci code, for which the pairwise matching can be done directly on the compressed files. In our experiments, this approach improves the processing time and incurs only a small loss in compression efficiency relative to standard compressors requiring a decoding phase.

Keywords: Data Compression; Feature vectors; SIFT; Fibonacci code.

1. Introduction

One of the topics on which Amihood Amir has done pioneering work is known as *Compressed Matching*. This is an extension of the classical Pattern Matching paradigm, in which the match has to be performed in the compressed domain, without first decompressing. Given a pattern P, a text T and complementing encoding and decoding functions $\mathcal{E}$ and $\mathcal{D}$, the Compressed Matching problem is to locate

*This is an extended version of papers that have been presented at the Information Mining and Management Conference (IMMM 2014) and the Prague Stringology Conference (PSC'14) in 2014, and appeared in their Proceedings.

P in the compressed text $\mathcal{E}(T)$. While the traditional approach searches for the pattern in the decompressed text, i.e., searching for P in $\mathcal{D}(\mathcal{E}(T))$, compressed matching calls for rather compressing the pattern too, and looking for $\mathcal{E}(P)$ in $\mathcal{E}(T)$, with the necessary adaptations. This has first been mentioned in Amir's paper with Benson dealing with two-dimensional run-length coding at the Data Compression Conference in 1992 [1, 2], and has since then triggered a myriad of related investigations. As a tribute to Amir, we present here yet another application of the compressed matching idea, this time to compressed feature vectors used in Image Processing.

The tremendous storage requirements and ever increasing resolutions of digital images, necessitate automated analysis and compression tools for information processing and extraction. A main challenge is detecting patterns even if they were rotated or scaled, working directly on the compressed form of the image. In a more general setting, a collection of images could be given, and the subset of those including at least one object, which is a rotated or scaled copy of the original object, 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.

There are several methods for transforming an image into a set of feature vectors for the purpose of object detection, such as SIFT (Scale Invariant Feature Transform) by Lowe [29], GLOH (Gradient Location and Orientation Histogram) [31], and SURF (Speed-up-Robust Features) [5], to mention only a few.

SIFT is a high probability object detection and identification method, which is done by matching the query image against a large database of local image features. Lowe's object recognition method transforms an image into a set of feature vectors, each of which is invariant to image translation, scaling, and rotation, partially invariant to illumination changes and robust to local geometric distortion. Feature descriptor vectors are computed for the extracted key points of objects from a set of reference images, which are then stored in a database. An object in a new image is identified after matching its features against this database using the Euclidean L_2 distance.

The matching process consists in a first stage of comparing each feature vector of the query image with each feature vector in the database. In a second stage, the best matching candidates are filtered out, and a clustering process is applied. Finally, each cluster passes a further more detailed model verification [29]. While the first stage could be done in a single sequential scan of the database, the latter stages require the possibility of direct access to the individual feature vectors.

The main idea of SIFT is to carefully choose a subset of the features so that this reduced set will be representative of the original image and will be processed

instead. Obviously, there are applications in which working on a dense set of features, rather than the sparse subsets mentioned above, is much better, since a larger set of local image descriptors provides more information than the corresponding descriptors evaluated only at selected interest points. In the case of object category or scene classification, experimental evaluations show that better classification results are often obtained by computing the so-called Dense SIFT descriptors (or DSIFT for short) as opposed to SIFT on a sparse set of interest points [7]. The dense sets may contain about 300 times more vectors than the sparse sets.

Query feature compression can contribute to faster retrieval, for example, when the query data is transmitted over a network, as in the case when mobile visual applications are used for identifying products in comparison shopping. Moreover, since the memory space on the mobile device is restricted, working directly on the compressed form of the data is sometimes required. A device with restricted memory is also an example showing that one still may need the space saving implications of compression, even though the time savings are often emphasized.

In this paper we suggest to apply metric preserving compression methods on the features of an image so that they can be processed in their compressed form. There are two ways to interpret the expected gains: on the one hand, one may consider the reduction of the required space; on the other hand, assuming that the space to be used is fixed in advance, compression allows the storage of more vectors, so that, instead of choosing a representative set of interest points, possibly reducing the object detection accuracy, one can increase the number of key points that can be processed using the same amount of memory storage.

2. Related Work

A feature descriptor encoder is presented in Chandrasekhar et al. [15]. They transfer the compressed features over the network and *decompress* them once data is received for further pairwise image matching. Chen et al. [16] perform tree-based retrieval, using a scalable vocabulary tree. Since the tree histogram suffices for accurate classification, the histogram is transmitted instead of individual feature descriptors. Also [13] encode a set of feature descriptors jointly and use tree-based retrieval when the order in which data is transmitted does not matter, as in our case. Several other SIFT feature vector compressors were proposed, and we refer the reader to [12] for a comprehensive survey. DiLillo et al. [19, 20] applied compression-based tools (dimensionality reduction, vector quantization, and coding) to provide object recognition as a preprocessing step. These are *lossy* techniques, in which a part of the data cannot be recovered. We propose a special encoding, which is a *lossless* alternative to the above, and is not only compact in its representation, but can also be processed directly *without* any decompression.

Figure 1 visually represents our approach as opposed to the traditional one of feature based object detectors and previous research regarding feature descriptors compression. The client uses any feature detector for extracting key points from the image, and computes the relevant vectors. These features are then sent along a network to the server, where pairwise pattern matching is applied against the stored database, as shown in Figure 1(a). Figure 1(b) depicts the scenario assumed in previous research that deals with compressed feature descriptors: compression is applied to the vectors before transmission, and decompression is performed once the descriptors are received on the server's side. Unlike traditional work, the current suggestion omits the decompression stage, and performs pairwise matching directly on the compressed data, as shown in Figure 1(c). Similar work, using quantization, has been suggested by Chandrasekhar et al. [14]. We do not apply quantization, and rather use a lossless encoding.

Figure 1: Block diagram showing (a) the traditional image retrieval system, (b) the scenario assumed by previous research, as opposed to (c) the scenario suggested in this paper.

We thus wish to perform the matching against the query image in the compressed form of the feature descriptor vectors so that the metric is retained, i.e., vectors are close in the original distance (e.g., Euclidean distance based on nearest neighbors according to the Best-Bin-First-Search algorithm in SIFT) if and only if they are close in their compressed counterparts. This can be done either by using the same metric but requiring that the compression should not affect the metric, or by changing the distance so that the number of false matches and true mismatches does not increase under this new distance. In the present work, we stick to the first alternative and do not change the L_2 metric used in SIFT.

For the formal description of the general case, let $\{\vec{f_1}, \vec{f_2}, \ldots, \vec{f_n}\}$ be a set of feature descriptor vectors generated using some feature based object detector, and let $\| \quad \|_M$ be a metric associated with the pairwise matching of this object detector. The *Compressed Feature Based Matching Problem* (CFBM) is to find a compression encoding of the vectors, denoted $\mathcal{E}(\vec{f_i})$, and an equivalent metric m so that for every $\epsilon > 0$ there exists a $\delta > 0$ in which $\forall i, j \in \{1, \ldots, n\}$

$$\|\vec{f_i} - \vec{f_j}\|_M < \epsilon \iff \|\mathcal{E}(\vec{f_i}) - \mathcal{E}(\vec{f_j})\|_m < \delta. \tag{1}$$

The rest of the paper is organized as follows. Section 3 gives a brief description of SIFT; Section 4 presents our lossless encoding for SIFT, DSIFT and PHOW (Pyramid Histogram Of visual Words) [7] feature vectors, especially suited for CFBM; Section 5 presents the algorithm used for compressed pairwise matching; Section 6 presents results on the compression performance of our lossless encoding for these particular features, and, finally, the last section concludes.

3. Brief Description of SIFT

Matching features across different images appearing in different scales and rotations is a common problem in computer vision, and SIFT is one of the famous tools dealing with it. The SIFT algorithm first preprocesses the original image in order to construct a *scale space* to ensure scale invariance. SIFT repeatedly generates progressively blurred out images of the original image and resizes it to half the size. The Laplacian of Gaussian (LoG) operation calculates second order derivatives on the blurred images. The blur smoothes out the noise and makes the second order derivative more stable. The LoG operation locates edges and corners in the image, which are used for finding keypoints. However, since calculating the LoG is computationally intensive, it is approximated by the Difference of Gaussians (DoG), calculating the difference between two consecutive scales, resulting in scale invariant keypoints.

Each pixel of the DoG scales is compared to all 26 of its neighbors, 8 neighbors in the current scale image and 18 more in the images of the scales one above and below it. Maxima and minima pixels are chosen as keypoints, which cannot be detected in the lowest or highest scales. Edges and low contrast pixels are eliminated from the set of keypoints. An orientation is calculated for each keypoint, choosing the most dominant one(s) around the keypoint. Any further calculations are done relative to this orientation. This effectively cancels out the effect of orientation, making it rotation invariant.

Highly distinctive vectors are then created for each keypoint as follows. A 16×16 window of pixels around the keypoint is taken. The window is split into sixteen 4×4 windows, each of which used to generate a histogram of 8 bins. Each bin corresponds to a different orientation (first bin for 0-44 degrees, second for 45-89 degrees, etc.), and the gradient orientations are put into these bins. To achieve rotation independence, the keypoint's rotation is subtracted from each orientation, so that each gradient orientation is relative to that of the keypoint. Finally, the 128 values which are attained are normalized.

4. Lossless Encoding for Feature Vectors

Given two feature vectors obtained by SIFT, we suggest achieving our goal to compress them using a lossless encoding so that the pairwise matching can be done directly on the compressed form of the file, by means of a *Fibonacci code*. Note that while the encoding will be different, the metric used in SIFT does not change, or in terms of the above notation, M and m refer to the same Euclidean metric generally denoted as L_2. We also apply the same code on the dense variants of SIFT, DSIFT and PHOW.

The following reflections led, among the many possible alternatives, to the choice of the Fibonacci code. Since direct random access is requested, adaptive schemes, like those based on Ziv-Lempel algorithms, are ruled out. Static Huffman codes would be possible, and would yield better compression, but the code has to be generated for each set, and the encoded numbers are not directly comparable. Moreover, since some given feature vectors are to be compared with those of entire sets of images to find the most fitting match, the use of Huffman codes would require, at decoding time, to deal with a different code, or construct a different tree, for every image.

Families of *fixed* codeword sets have been studied by Elias [21] in what he called *universal* codes. His γ and δ-codes encode the integers by the binary sequences:

$$1, 010, 011, 00100, 00101, 00110, 00111, 0001000, \ldots \quad \text{for } \gamma,$$

$$1, 0100, 0101, 01100, 01101, 01110, 01111, 00100000, \ldots \quad \text{for } \delta.$$

The Elias codes could be used directly in their compressed forms, as the difference between integers represented by codewords of the same length can be evaluated by subtracting the representations themselves, but handling codewords of different lengths is more involved. The codes are also only efficient for quite large alphabets, whereas the number of different elements in the SIFT feature vectors is small. The same is true for several other universal sets such as ETDC [10] and (s, c)-dense codes [8] and other byte-codes, in which each codeword consists of an integral number of bytes; they are not appropriate for our application for which even a fixed length code of length one byte might suffice.

Directly accessible codes (DACs) are investigated in [11]: the standard binary representation of the integers is broken into blocks of b bits, and a flag bit is adjoined to each block to indicate whether it is the last block in the representation of a given integer. The parameter b cannot be chosen too small, otherwise the overhead caused by the flag bits might be significant, but for larger b, the shortest codewords are too long to be efficient for small sets as required in our application. Hence Fibonacci codewords are preferable here and are also easier to compare directly.

More research on direct access to variable length encoding schemes has recently been published, generally based on fast implementations of *rank* and *select* operations on bit-vectors and coupled with the use of *Wavelet trees*, as in [28] for Rice and Elias codes, [24] for Huffman trees and more generally in [32]. These data structures need additional space to enable direct access to a greater extent than is needed for our application.

Our problem of compressing vectors of integers is reminiscent of the processing of lists of indices in full text Information Retrieval Systems. In both areas, the

distribution of the numbers is skewed, with higher probability to smaller values. The compression of such lists has been addressed, among others in [34, 4, 3, 17], though here we also need the possibility of processing the numbers in their compressed form.

4.1. The Fibonacci Code

The Fibonacci code is a universal variable length encoding of the integers based on the Fibonacci sequence rather than on powers of 2. A prefix of this infinite encoding sequence can be used as a fixed alternative to Huffman codes, giving obviously less compression, but adding simplicity (there is no need to generate a new code every time), robustness and speed [22, 26]. The particular property of the binary Fibonacci encoding is that there are no adjacent 1's, so that the string 11 can act like a *comma* between codewords. More precisely, the codeword set consists of all the binary strings for which the substring 11 appears exactly once, at the left end of the string.

The connection to the Fibonacci sequence can be seen as follows: just as any integer k has a standard binary representation, that is, it can be uniquely represented as a sum of powers of 2, $k = \sum_{i \geq 0} b_i 2^i$, with $b_i \in \{0, 1\}$, there is another possible binary representation based on Fibonacci numbers, $k = \sum_{i \geq 0} f_i F(i)$, with $f_i \in \{0, 1\}$, where it is convenient to define the Fibonacci sequence here by

$$F(0) = 1, F(1) = 2; \qquad F(i) = F(i-1) + F(i-2) \text{ for } i \geq 2.$$

This Fibonacci representation will be unique if, when encoding an integer, one repeatedly tries to fit in the largest possible Fibonacci number.

For example, the largest Fibonacci number fitting into 19 is $F(5) = 13$, for the remainder 6 one can use the Fibonacci number $F(3) = 5$, and the remainder $F(0) = 1$ is a Fibonacci number itself. So one would represent 19 as $19 = 13 + 5 + 1$, yielding the binary string 101001. Note that the bit positions correspond to $F(i)$ for increasing values of i from right to left, just as for the standard binary representation, in which $19 = 16 + 2 + 1$ would be represented by 10011. Each such Fibonacci representation starts with a 1, so by preceding it with an additional 1, one gets a sequence of uniquely decipherable codewords.

Decoding, however, would not be instantaneous, because the set lacks the prefix property. For example, a first attempt to start the parsing of the encoded string 1101111111110 by 110 11 11 11 11 would fail, because the remaining suffix 10 is not the prefix of any codeword. So only after having read 5 codewords in this case (and the example can obviously be extended) would one know that the correct parsing is 1101 11 11 11 110. To overcome this problem, the Fibonacci code defined in [22] simply reverses each of the codewords. The adjacent 1s are then at

the right instead of at the left end of each codeword, thus yielding the prefix code $\{11, 011, 0011, 1011, 00011, 10011, 01011, 000011, 100011, 010011, 001011, 101011, 0000011, \ldots\}$.

A disadvantage of this reversing process is that the order preserving of the previous representation is lost, e.g., the codewords corresponding to 17 and 19 are 1010011 and 1001011, but if we compare them as if they were standard binary representations of integers, the first, with value 83, is larger than the second, with value 75. At first sight, this seems to be critical, because we want to compare numbers in order to subtract the smaller from the larger. But in fact, since we calculate the L_2 norm, the *square* of the differences of the coordinates is needed. It therefore does not matter if we calculate $x - y$ or $y - x$, and there is no problem dealing with negative numbers. The reversed representation can therefore be kept.

4.2. Using a Fibonacci Code for feature vectors

We wish to encode SIFT, DSIFT and PHOW feature vectors, each consisting of exactly 128 coordinates. Thus, in addition to the ability of parsing an encoded feature vector into its constituting coordinates, separating adjacent vectors could simply be done by counting the number of codewords, which is easily done with a prefix code.

Empirically, SIFT, DSIFT and PHOW vectors are characterized by having smaller integers appear with higher probability. To illustrate this, we considered the Lenna image (an almost standard compression benchmark) and first applied Matlab's SIFT application on it, generating 737 feature vectors. The number of occurrences of 0 was 28,182, and that of the following numbers 1 to 25 is plotted in Figure 2(a). All the numbers were between 0 and 255 and could thus be encoded in a single byte. The total raw size of the feature vectors for Lenna was thus 94,336 bytes. We then applied Matlab's DSIFT and PHOW applications on it, generating 253,009 and 237,182 feature vectors, respectively, of 128 coordinates each. The numbers (thousands of occurrences for values from 2 to 255) are plotted in Figure 2(b).

(a) SIFT (b) DSIFT and PHOW

Figure 2: Value distribution in feature vectors.

SIFT, DSIFT and PHOW feature vectors contain repeated zero-runs, as could be expected by the high number of zeros. This led to the idea of representing a *pair* of adjacent 0s by a single codeword. That is, the pair 00 is assigned the first Fibonacci codeword 11, a single 0 is encoded by the second codeword 011, and

generally, the integer k is represented by the Fibonacci codeword corresponding to the integer $k+2$, for $k \geq 0$.

The usual approach for using an universal code, such as the Fibonacci code, is first sorting the probabilities of the source alphabet symbols in decreasing order and then assigning the universal codewords by increasing codeword lengths, so that high probability alphabet symbols are given the shorter codewords. In our case, in order to be able to perform compressed pairwise matching, we omit sorting the probabilities, as already suggested in [9] for byte-codes and in [27] for Huffman coding. Figure 2 shows that the order is not strictly monotonic, but that the fluctuations are very small. Indeed, experimental results show that encoding the numbers themselves, instead of their indices in the list sorted by their decreasing values, has hardly any influence (0.1% for SIFT and less than 0.4% for DSIFT and PHOW on our test images).

As example, consider the 25th PHOW feature vector of Lenna's Image. The first 20 coordinates of this vector are

$$8, 19, 3, 1, 5, 7, 0, 0, 0, 0, 1, 1, 32, 60, 0, 0, 0, 0, 0, 0, \ldots$$

Instead of encoding it as

$$100011 \;\; 0101011 \;\; 1011 \;\; 011 \;\; 10011 \;\; 000011 \;\; 11 \;\; 11 \;\; 11 \;\; 11$$
$$011 \;\; 011 \;\; 00101011 \;\; 1001000011 \;\; 11 \;\; 11 \;\; 11 \;\; 11 \;\; 11 \;\; 11$$

in which 11 represents a single 0, we rather encode it as

$$010011 \;\; 00000011 \;\; 00011 \;\; 0011 \;\; 01011 \;\; 100011 \;\; 11 \;\; 11$$
$$0011 \;\; 0011 \;\; 000000011 \;\; 0101000011 \;\; 11 \;\; 11 \;\; 11$$

where the 11 stands for 00, and thereby reduce the size of this compressed prefix from 75 bits to 71, as opposed to 160 bits for the first 20 elements of the original uncompressed PHOW vector using one byte per integer.

Note that since all numbers are simply shifted by 2, the difference between two Fibonacci encodings is preserved, which is an essential property for computing their distance in the compressed form.

5. Compressed Pairwise Matching

We start with a general algorithm, Sub( ), for subtraction which is used in SIFT, DSIFT and PHOW L_2 norm computations. Given two encoded descriptors, one needs to compute their L_2 norm. Each component is first subtracted from the corresponding one, then the squares of these differences are summed. The

algorithm for computing the subtraction of two Fibonacci encoded coordinates A and B is given in Figure 3. We start by stripping the trailing 1s from both, and pad, if necessary, the shorter codeword with zeros at its right end so that both representations are of equal length. Note that the term first, second and next refer to the order from right to left.

Sub(A, B)
scan the bits of A and B from right to left
$a_1 \longleftarrow$ first bit of A
$a_2 \longleftarrow$ second bit of A
while next bit of A exists $\{$
  $a_3 \longleftarrow$ next bit of A
  $b_1 \longleftarrow$ next bit of B
  $a_1 \longleftarrow a_1 - b_1$
  $a_2 \longleftarrow a_1 + a_2$
  $a_3 \longleftarrow a_1 + a_3$
  $a_1 \longleftarrow a_2$
  $a_2 \longleftarrow a_3$
$\}$
$b \longleftarrow$ value of last 2 bits of B
if $b \neq 0$ then $b \longleftarrow 3 - b$
return $2 * a_1 + a_2 - b$

Figure 3: *Subtraction of Fibonacci Codewords.*

At the end of the while loop, there are 2 unread bits left in B, which can be 00, 10 or 01, with values 0, 1 or 2 in the Fibonacci representation, but when read as standard binary numbers, the values are 0, 2 and 1. This is corrected in the commands after the while loop of the algorithm. The evaluation relies on the fact that a 1 in position i of the Fibonacci representation is equivalent to, and can thus be replaced by, 1s in positions $i+1$ and $i+2$. This allows us to iteratively process the subtraction, independently of the Fibonacci number corresponding to the leading bits of the given numbers. Processing is, therefore, done in time proportional to the size of the compressed file, without any decoding.

As example, consider the numbers $A = 130$ and $B = 65$, encoded by the

strings representing 132 and 67, which are 10001001011 and 1010100011, respectively. The upper part of Figure 4 shows the results of applying the subtraction algorithm on A and B, which appear, in their reduced form (without trailing 1, but with B padded by 0 to get to the same length) in the boxed first line and last column. The last (i.e., leftmost) two bits of B do not appear in the boxed column in the figure. At the end, b is assigned the value 1, and the result is indeed $130 - 65 = 65 = 2 * 25 + 16 - 1$. Note that had we subtracted A from B, the values in columns a_1 and a_2 would be negative or 0 (except in the first row), as can be seen in the lower part of the figure, but the algorithm would still work correctly. In that case, the result is indeed $65 - 130 = -65 = 2 * (-25) - 14 - 1$.

							a_3	a_2	a_1	b_1
1	0	0	0	1	0	0	1	0	1	0
	1	0	0	0	1	0	0	2	1	1
		1	0	0	0	1	0	0	2	0
			1	0	0	0	1	2	2	0
				1	0	0	0	3	4	0
					1	0	0	4	7	1
						1	0	6	10	0
							1	10	16	1
								16	25	

							a_3	a_2	a_1	b_1
1	0	1	0	1	0	0	0	1	0	1
	1	0	1	0	1	0	0	-1	0	0
		1	0	1	0	1	0	0	-1	1
			1	0	1	0	1	-2	-2	0
				1	0	1	0	-1	-4	0
					1	0	1	-4	-5	1
						1	0	-5	-10	0
							1	-10	-15	0
								-14	-25	

Figure 4: Example of direct differencing.

To calculate the L_2 norm, the two Fibonacci encoded input vectors have to be scanned in parallel from left to right. In each iteration, the first codeword (identified as the shortest prefix ending in 11) is removed from each of the two input vectors, and each pair of coordinates is processed according to the procedure $\mathsf{Sub}(A, B)$ above. The codeword 11, representing two consecutive zeros, needs a special treatment: if one of the codewords is 11, but the other, say B, is not, the 11 should be

```
L2Norm(V1, V2)
SSQ ⟵ 0
while V1 and V2 are not empty {
    remove first codeword from V1 and assign it to A
    remove first codeword from V2 and assign it to B
    if A ≠ B then
        if A = 11 then
            S ⟵ Sub(B, 011)
            V1 ⟵ 011 ‖ V1
        else if B = 11 then
            S ⟵ Sub(A, 011)
            V2 ⟵ 011 ‖ V2
        else S ⟵ Sub(A, B)
        SSQ ⟵ SSQ + S^2
}
return √SSQ
```

Figure 5: SIFT and PHOW compressed L_2 norm computation.

replaced by two codewords 011 011, each representing a single zero. We thus perform Sub$(B, 011)$, and then concatenate the second 011 in front of the remaining input vector, to be processed in the following iteration. The details appear in Figure 5, where $\|$ denotes concatenation and the variable SSQ, accumulating the sum of the squares of the differences, is initialized to 0. At each iteration, the result, S, of the subtraction of the two given Fibonacci encoded numbers is computed by the function Sub( ); it is then squared and added to the accumulated value SSQ. By definition, the L_2 norm is the square root of the sum of the squares.

6. Compression Performance

An empirical study is supposed to apply the suggested methods to a representative set of example images. It is, however, not realistic to assume that such a set can be found, yet the compressibility and processing times are strongly related to the specific characteristics of the chosen images. We therefore chose just a small sample of what seemed to be typical images in our eyes, and we do not claim that one might infer from the results that similar performances are expected for other sets.

At first, three images are considered in our experiments: *Lenna*, *House* and *Aerial* (5.1.10), taken from the Miscellaneous set of the SIPI (Signal and Image Processing Institute) Image Data Base[1]. The last lines in the tables below correspond to the average values obtained for all the images in this set of size 256×256 or 512×512 pixels, generating between 100 and 2000 feature vectors. There are 37 images in this set.

Table 1 presents the results for the SIFT vectors. The second column shows the number of feature vectors generated by Matlab when applied on the given image. The other figures are file sizes, in KB. The third column presents the size of the uncompressed file, using a single byte for each of the 128 coordinates of each feature vector, since all values are below 256. The column entitled Fib gives the size of the file when each number is represented by its Fibonacci encoding, but reserving the first codeword for encoding 00. For comparison, the file sizes achieved by other compression methods are listed in the following columns. The column headed Huff corresponds to a Huffman code whose first element is again a pair of zeros 00 as for the Fibonacci encoding, and keeping the following elements in order of the represented values themselves, not of their frequencies, as suggested above. Then come the file sizes for Elias' γ and δ codes, and finally the last two columns give the compression performances of gzip (with parameter -9 for maximal compression) and bzip2. These are adaptive compression schemes, and as such no real competitors to Huffman or Fibonacci coding: while their performance on text files is often superior, taking advantage also of the order in which the characters appear, and not just of their frequencies, they cannot be used when direct access to a part of the compressed file is required, as in our case of feature vectors, and they require a sequential scan from their beginning for the decoding.

As can be seen, the use of Fibonacci instead of Huffman coding incurs a compression loss of about 6–10%, and in this case even gzip is 3–7% worse than Huffman. The use of an Elias-δ code would increase the file size by 18–24%.

To evaluate the compression loss due to omitting the sorting of the frequencies, we considered the compression where each symbol is encoded using the Fibonacci codeword assigned according to its position in the list of frequencies ordered by decreasing values. For Huffman coding, the elements were sorted according to their frequencies. The difference in compression was negligible in all cases, about 0.1–0.3%.

Tables 2 and 3 present the compression performance for the PHOW and DSIFT vectors, respectively, using the same format as above, only the file sizes are now given in MB. For these larger files, Fibonacci encoding increases the files by 12–

[1] `http://sipi.usc.edu/database/`

Table 1: Raw and compressed file sizes for SIFT feature vectors in KB.

Image	Vectors	Size	Fib	Huff	γ	δ	gzip	bzip2
Lenna	737	92.1	64.2	60.6	72.6	71.7	65.0	66.0
House	991	123.9	93.7	86.9	107.0	104.9	91.9	92.6
Aerial	477	59.6	50.1	45.3	57.8	56.4	46.8	47.5
average	823	108.2	86.3	78.9	99.2	97.0	82.4	82.7

Table 2: Raw and compressed file sizes for PHOW feature vectors in MB.

Image	Vectors	Size	Fib	Huff	γ	δ	gzip	bzip2
Lenna	237,182	29.0	18.3	17.5	20.4	20.2	16.5	16.5
House	237,182	29.0	20.1	18.7	22.7	22.2	17.4	17.4
Aerial	53,374	6.5	5.9	5.2	6.8	6.6	5.1	5.2
average	172,601	21.1	16.0	14.3	18.3	17.8	13.8	13.8

14% relative to Huffman coding, but gzip might sometimes compress more than Huffman by about 5%. The reason for the better performance of gzip on the larger files is the appearance of longer runs of zeros, all of which are encoded as single elements, whereas our Huffman or Fibonacci codes break 0-runs into a sequence of 00 pairs. Note that γ and δ codes yield sometimes encoded files that are larger than the original, thus giving negative compression.

Table 4 brings some more statistical data on the compression tests. To evaluate the influence of the image size, the 37 images are partitioned into two classes according to the number of pixels used: 13 images of 256×256 and 24 images of 512×512 pixels. As to the examples used above, *Aerial* belongs to the first set, and *Lenna* and *House* to the second. The average number of feature vectors generated by Matlab for SIFT was 248 for the smaller set, with standard deviation 88, and 1135 for the larger, with standard deviation 267. For PHOW and DSIFT, the number of generated feature vectors was constant within each set and can be found in the Vectors columns of Tables 2 and 3. For each of the methods SIFT, PHOW and DSIFT, and the two sets labeled 256 and 512, Table 4 displays the average compression ratio, its standard deviation in the set, as well as the minimum and maximum values obtained. The compression ratio is the size of the compressed file divided by the corresponding uncompressed file using a single byte for each of the 128 elements of each feature vector.

As can be seen, for SIFT and DSIFT, there is only a small difference of 4–9% between the two sets, but for PHOW, the set of the features of the smaller images seems to be more compressible, though with higher fluctuations. The order between the columns, representing the different compression methods, is not

Table 3: Raw and compressed file sizes for DSIFT feature vectors in MB.

Image	Vectors	Size	Fib	Huff	γ	δ	gzip	bzip2
Lenna	253,009	30.9	26.7	23.8	31.1	30.1	23.8	24.1
House	253,009	30.9	26.4	23.6	30.6	29.7	24.0	24.5
Aerial	61,009	7.4	6.4	5.7	7.4	7.2	5.8	5.9
average	177,157	21.6	19.3	16.9	22.5	21.7	17.0	17.3

affected.

The following tests were run to empirically evaluate the processing times. To simulate a large number of different L_2 norm calculations, we considered, for each of the test images, the 100 first feature vectors, and calculated their L_2 distance from each of the other vectors in the file. For example, for *Lenna*'s SIFT features, $100 \times 737 = 73{,}700$ vector pairs were processed. This was done for all the images in the set used above, and Table 5 lists the processing time, in seconds, for the three test images and the average time for all the files in the set. The columns correspond to the three scenarios described in Figure 1: (a) calculating the norms using the uncompressed feature vectors; (b) using a canonical Huffman code, decoding and calculating then the norm; finally (c) using a Fibonacci code and calculating the norm without decompressing. These test were then repeated for PHOW and DSIFT, but only with the 10 first vectors, so for *Lenna*'s PHOW features, $10 \times 237{,}182 = 2{,}371{,}820$ vector pairs were processed. All the tests have been run on an Intel Xeon CPU E5-2650 at 2.00Ghz with cache size 20MB and 16GB of RAM.

We see that using the raw data, Method (a), is obviously the fastest, but that using the Fibonacci code directly in its compressed form, Method (c), may yield time savings of about 36% over the standard approach, Method (b), of decoding (a Huffman code in our tests) and then calculating the norm.

Table 6 gives again more statistical data, including average time, its standard deviation, minimum and maximum times, after partitioning the images into the two sets labeled 256 and 512, as above. The times are given in micro-seconds and have been normalized by dividing the total times of Table 5 by the number of processed pairs for each image. We see that there are no significant differences, neither for the different image sizes, nor between the feature vector sets SIFT, PHOW and DSIFT, and that the relative order of processing methods (a), (b) and (c) is maintained.

Table 4: Statistical data on compression ratios.

			Fib	Huff	γ	δ	gzip	bzip2
SIFT	256	Avg	0.718	0.672	0.815	0.799	0.710	0.723
		Std	0.060	0.044	0.077	0.073	0.042	0.045
		min	0.637	0.605	0.714	0.691	0.635	0.625
		max	0.839	0.759	0.970	0.946	0.785	0.797
	512	Avg	0.774	0.710	0.886	0.867	0.744	0.748
		Std	0.081	0.064	0.103	0.098	0.065	0.063
		min	0.478	0.464	0.515	0.509	0.479	0.489
		max	0.852	0.766	0.987	0.960	0.790	0.789
PHOW	256	Avg	0.598	0.553	0.662	0.651	0.517	0.522
		Std	0.193	0.168	0.245	0.237	0.188	0.193
		min	0.345	0.312	0.340	0.338	0.261	0.254
		max	0.912	0.794	1.063	1.034	0.791	0.798
	512	Avg	0.778	0.695	0.892	0.868	0.672	0.674
		Std	0.171	0.134	0.219	0.206	0.150	0.153
		min	0.299	0.257	0.282	0.280	0.195	0.191
		max	0.960	0.815	1.128	1.084	0.802	0.808
DSIFT	256	Avg	0.809	0.721	0.934	0.907	0.727	0.740
		Std	0.164	0.142	0.209	0.199	0.153	0.159
		min	0.289	0.246	0.274	0.269	0.207	0.200
		max	0.980	0.814	1.158	1.104	0.812	0.831
	512	Avg	0.898	0.784	1.049	1.013	0.788	0.803
		Std	0.051	0.033	0.065	0.058	0.031	0.032
		min	0.788	0.674	0.926	0.889	0.676	0.688
		max	0.982	0.820	1.159	1.109	0.817	0.841

7. Conclusion

We have dealt with the problem of compressing sets of feature vectors known as SIFT, DSIFT and PHOW, under the constraint of processing the data directly in its compressed form. Such an approach may be advantageous not only to save storage space, but may also improve manipulation speed, and in fact the whole data handling from transmission to processing.

Our solution is based on encoding the vector elements by means of a Fibonacci code, which is generally inferior to Huffman coding from the compression point of view, but has several advantages, turning it into a preferred choice in our case: (a) simplicity – the code is fixed and need not be generated anew for different distributions; (b) the possibility to identify each individual codeword – avoiding the necessity of adding separators, and not requiring a sequential scan; (c) allowing to perform subtractions using the compressed form – and thereby calculating the L_2

Table 5: Processing times of calculating a set of L_2 norms, in seconds.

	SIFT			PHOW			DSIFT		
Image	(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)
Lenna	0.06	1.38	0.89	5.29	55.01	36.84	5.51	54.98	36.90
House	0.09	2.09	1.25	5.09	43.20	30.64	5.26	54.90	33.51
Aerial	0.04	1.10	0.64	1.07	12.09	7.73	1.23	13.42	8.78
average	0.06	1.65	1.04	3.66	33.87	21.58	3.92	39.16	25.10

Table 6: Statistical data on normalized processing times, in μ-seconds.

		SIFT			PHOW			DSIFT		
		(a)	(b)	(c)	(a)	(b)	(c)	(a)	(b)	(c)
256	Avg	0.69	19.66	11.63	2.13	15.58	9.10	2.18	20.70	12.85
	Std	0.16	1.86	0.89	0.38	4.87	3.57	0.38	4.25	2.87
	min	0.44	17.10	10.36	1.93	8.82	4.20	1.93	6.98	3.66
	max	0.99	23.06	13.42	3.35	22.65	14.48	3.39	23.95	14.59
512	Avg	0.80	20.34	12.88	2.12	20.12	12.91	2.15	21.57	14.04
	Std	0.07	0.88	0.66	0.06	3.91	2.36	0.09	4.19	1.25
	min	0.75	18.39	11.57	2.03	8.25	7.78	1.83	3.39	10.26
	max	1.03	21.60	13.66	2.23	23.66	15.53	2.22	23.75	15.16

norm, whereas a Huffman code would have to use some translation table; (d) using a lossless encoding scheme, whereas previous approaches to work with compressed data suggested the use of quantization or dimensionality reduction, in which a part of the data is lost.

On our experiments, there is only a small loss, of 6–14%, in compression efficiency relative to the optimal Huffman codes, which might be worth a price to pay for the improved processing. Other standard compressors, like gzip or bzip2, might improve even on Huffman, but do not allow random access.

The basic techniques of the present work can be extended to a different, yet related problem: the *Compressed Approximate Pattern Matching* paradigm. When searching for a pattern in a given text one may also be interested in locating strings that are not completely identical to the original pattern, but are quite similar. In the literature this problem is referred to as *Approximate Pattern Matching*, which is to find all occurrences of substrings in a given text T that are at a given "distance" k or less from a pattern P under some metric. This is yet another of the favorite research topics of Amihood Amir, who has contributed many of the seminal papers in this area.

The *Compressed Approximate Matching Problem* (CAMP) is locating *similar* patterns to the searched one working directly on the compressed form of the text.

Defining *similarity* formally necessities the existence of a metric so that if the distance between two patterns under this metric is small, searching for one of them in the compressed form of the file will be able to locate both patterns. Approximate compressed pattern matching was first introduced in [2] as an open problem. It has been solved for many cases, e.g., for byte Huffman coding of words [18], for run length encoded strings [30], for Lempel-Ziv compressed text in [33, 25], and Straight Line Programs [6, 23].

More formally, given a pattern P, a compressed text $\mathcal{E}(T)$, and a metric $\| \quad \|_M$, the CAMP is to locate all patterns Q in $\mathcal{E}(T)$ so that $\|P - Q\|_M \leq \epsilon$ for some $\epsilon \geq 0$. This is a generalization of the compressed pattern matching problem in which $\epsilon = 0$.

A tempting definition is dealing with two metrics, $\| \quad \|_M$ and $\| \quad \|_m$, so that if $\|P - Q\|_M \leq \epsilon$ for some $\epsilon \geq 0$ in T, then there exist a corresponding metric $\| \quad \|_m$ and $\delta \geq 0$ so that $\|\mathcal{E}(P) - \mathcal{E}(Q)\|_m \leq \delta$ in the compressed file $\mathcal{E}(T)$. In this paper, we dealt with the specific case in which the patterns are feature vectors and $M = m = L_2$.

References

[1] A. AMIR AND G. BENSON: *Efficient two-dimensional compressed matching*, in Proceedings of the IEEE Data Compression Conference, DCC 1992, Snowbird, Utah, March 24-27, 1992., 1992, pp. 279–288.

[2] A. AMIR, G. BENSON, AND M. FARACH: *An alphabet independent approach to two-dimensional pattern matching*. SIAM J. Comput., 23(2) 1994, pp. 313–323.

[3] V. N. ANH AND A. MOFFAT: *Compressed inverted files with reduced decoding overheads*, in SIGIR '98: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, August 24-28 1998, Melbourne, Australia, 1998, pp. 290–297.

[4] ——: *Inverted index compression using word-aligned binary codes*. Inf. Retr., 8(1) 2005, pp. 151–166.

[5] H. BAY, T. TUYTELAARS, AND L. GOOL: *SURF: Speeded Up Robust Features*, in European Conference on Computer Vision (ECCV), 2006, pp. 404–417.

[6] P. BILLE, G. M. LANDAU, R. RAMAN, K. SADAKANE, S. R. SATTI, AND O. WEIMANN: *Random access to grammar-compressed strings*, in Symposium on Discrete Algorithms (SODA), 2011, pp. 373–389.

[7] A. BOSCH, A. ZISSERMAN, AND X. MUNOZ: *Image classification using random forests and ferns*, in Proc. 11th International Conference on Computer Vision (ICCV'07), Rio de Janeiro, Brazil, 2007, pp. 1–8.

[8] N. R. BRISABOA, A. FARIÑA, G. NAVARRO, AND M. F. ESTELLER: (s, c)-*dense coding: An optimized compression code for natural language text databases*, in String Processing and Information Retrieval, 10th International Symposium, SPIRE 2003, Manaus, Brazil, October 8-10, 2003, Proceedings, 2003, pp. 122–136.

[9] N. R. BRISABOA, A. FARIÑA, G. NAVARRO, AND J. R. PARAMÁ: *Lightweight natural language text compression*. Inf. Retr., 10(1) 2007, pp. 1–33.

[10] N. R. BRISABOA, E. L. IGLESIAS, G. NAVARRO, AND J. R. PARAMÁ: *An efficient compression code for text databases*, in Advances in Information Retrieval, 25th European Conference on IR Research, ECIR 2003, Pisa, Italy, April 14-16, 2003, Proceedings, 2003, pp. 468–481.

[11] N. R. BRISABOA, S. LADRA, AND G. NAVARRO: *Dacs: Bringing direct access to variable-length codes*. Inf. Process. Manage., 49(1) 2013, pp. 392–404.

[12] V. CHANDRASEKHAR, M. MAKAR, G. TAKACS, D. CHEN, S. S. TSAI, N. M. CHEUNG, R. GRZESZCZUK, Y. A. REZNIK, AND B. GIROD: *Survey of SIFT compression schemes*, in Int. Workshop on Mobile Multimedia Processing (WMMP), 2010.

[13] V. CHANDRASEKHAR, Y. A. REZNIK, G. TAKACS, D. M. CHEN, S. S. TSAI, R. GRZESZCZUK, AND B. GIROD: *Compressing Feature Sets with Digital Search Trees*, in ICCV Workshops, 2011, pp. 32–39.

[14] V. CHANDRASEKHAR, G. TAKACS, D. M. CHEN, S. S. TSAI, Y. A. REZNIK, R. GRZESZCZUK, AND B. GIROD: *Compressed Histogram of Gradients: A Low-Bitrate Descriptor*. International Journal of Computer Vision, 96(3) 2012, pp. 384–399.

[15] V. CHANDRASEKHAR, G. TAKACS, D. M. CHEN, S. S. TSAI, J. SINGH, AND B. GIROD: *Transform Coding of Image Feature Descriptors*, in Visual Communications and Image Processing, vol. 7257 (1), 2009, pp. 725710–725710–9.

[16] D. M. CHEN, S. S. TSAI, V. CHANDRASEKHAR, G. TAKACS, J. P. SINGH, AND B. GIROD: *Tree Histogram Coding for Mobile Image Matching*, in Data Compression Conference, DCC–09, 2009, pp. 143–152.

[17] Y. CHOUEKA, A. S. FRAENKEL, AND S. T. KLEIN: *Compression of concordances in full-text retrieval systems*, in SIGIR'88, Proceedings of the 11th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Grenoble, France, June 13-15, 1988, 1988, pp. 597–612.

[18] E. DE MOURA, G. NAVARRO, N. ZIVIANI, AND R. BAEZA-YATES: *Fast and flexible word searching on compressed text*. ACM Trans. Inform. Syst. (TOIS), 18 (2) 2000, pp. 113–139.

[19] A. DILILLO, A. DAPTARDAR, G. MOTTA, K. THOMAS, AND J. STORER: *Applications of Compression to Content Based Image Retrieval and Object Recognition*, in Proceedings International Conference On Data Compression, Communication, and Processing (CPP–11), 2011, pp. 179–189.

[20] A. DILILLO, G. MOTTA, K. THOMAS, AND J. STORER: *Compression-Based Tools for Navigation with and Image Database*. Algorithms, 5 2012, pp. 1–17.

[21] P. ELIAS: *Universal codeword set and representations of the integers*. IEEE Trans. Information Theory, IT–21(2) 1975, pp. 194–203.

[22] A. S. FRAENKEL AND S. T. KLEIN: *Robust universal complete codes for transmission and compression*. Discrete Applied Mathematics, 64 1996, pp. 31–55.

[23] T. GAGIE, P. GAWRYCHOWSKI, AND S. J. PUGLISI: *Faster Approximate Pattern Matching in Compressed Repetitive Texts*, in Algorithms and Computation - 22nd International Symposium, ISAAC 2011, Yokohama, Japan, December 5-8, 2011. Proceedings, 2011, pp. 653–662.

[24] R. GROSSI, A. GUPTA, AND J. S. VITTER: *High-order entropy-compressed text indexes*, in Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA., 2003, pp. 841–850.

[25] J. KÄRKKÄINEN, G. NAVARRO, AND E. UKKONEN: *Approximate string matching on Ziv-Lempel compressed text*. Discrete Algorithms, 1 (3-4) 2003, pp. 313–338.

[26] S. T. KLEIN AND M. KOPEL BEN-NISSAN: *On the Usefulness of Fibonacci Compression Codes*. The Computer Journal, 53 2010, pp. 701–716.

[27] S. T. KLEIN AND D. SHAPIRA: *Huffman Coding with Non-Sorted Frequencies*. Mathematics in Computer Science, 5(2) 2011, pp. 171–178.

[28] M. O. KÜLEKCI: *Enhanced variable-length codes: Improved compression with efficient random access*, in Data Compression Conference, DCC 2014, Snowbird, UT, USA, 26-28 March, 2014, 2014, pp. 362–371.

[29] D. G. LOWE: *Distinctive Image Features from Scale-Invariant Keypoints*. International Journal of Computer Vision, 60 (2) 2004, pp. 91–110.

[30] V. MÄKINEN, G. NAVARRO, AND E. UKKONEN: *Approximate Matching of Run-Length Compressed Strings*. Algorithmica, 35 (4) 2003, pp. 347–369.

[31] K. MIKOLAJCZYK, T. TUYTELAARS, C. SCHMID, A. ZISSERMAN, J. MATAS, F. SCHAFFALITZKY, T. KADIR, AND L. VAN GOOL: *A Comparison of Affine Region Detectors*, in International Journal of Computer Vision, vol. 65 (1-2), 2005, pp. 43–72.

[32] G. NAVARRO: *Wavelet trees for all*. J. Discrete Algorithms, 25 2014, pp. 2–20.

[33] G. NAVARRO AND M. RAFFINOT: *A general practical approach to pattern matching over Ziv-Lempel compressed text*, in Proceedings of Combinatorial Pattern Matching (CPM), 1999, pp. 14–36.

[34] I. H. WITTEN, A. MOFFAT, AND T. C. BELL: *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*, Morgan Kaufmann, 1999.