Metric Preserving Dense SIFT Compression

Shmuel T. Klein[1] and Dana Shapira[2,3]

[1] Computer Science Department, Bar Ilan University, Israel
[2] Computer Science Department, Ashkelon Academic College, Israel
[3] Department of Computer Science and Mathematics, Ariel University, Israel
tomi@cs.biu.ac.il, shapird@gmail.com

Abstract. The problem of compressing a large collection of feature vectors so that object identification can further be processed on the compressed form of the features is investigated. The idea is to perform matching against a query image in the compressed form of the feature descriptor vectors retaining the metric. Given two SIFT feature vectors, in previous work we suggested to compress them using a lossless encoding for which the pairwise matching can be done directly on the compressed files, by means of a Fibonacci code. In this paper we extend our work to Dense SIFT and in particular to PHOW features, that contain, for each image, about 300 times as many vectors as the original SIFT.

1 Introduction

The tremendous storage requirements and ever increasing resolutions of digital images, necessitate automated analysis and compression tools for information processing and extraction. There are several methods for transforming an image into a set of feature vectors, such as SIFT (Scale Invariant Feature Transform) by Lowe [13], GLOH (Gradient Location and Orientation Histogram) [14], and SURF (Speed-up-Robust Features) [1], to mention only a few. Ideally, such descriptors are invariant to scaling, rotation, illumination changes and local geometric distortion.

The main idea 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 this paper we suggest that instead of choosing a representative set of interest points, possibly reducing the object detection accuracy, one can apply metric preserving compression methods so that a larger number of key points can be processed using the same amount of memory storage.

SIFT 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. 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 [3]. 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,

Shmuel T. Klein, Dana Shapira: Metric Preserving Dense SIFT Compression, pp. 139–147.
Proceedings of PSC 2014, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05547-2

since the memory space on the mobile device is restricted, working directly on the compressed form of the data is sometimes required.

A feature descriptor encoder is presented in Chandrasekhar et al. [7]. They transfer the compressed features over the network and *decompress* them once data is received for further pairwise image matching. Chen et al. [8] 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 Chandrasekhar et al. [5] 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 [4] for a comprehensive survey.

We propose a special encoding which is not only compact in its representation, but can also be processed directly *without* any decompression. That is, unlike traditional feature vectors compression which decompresses before applying pairwise matching, the current suggestion omits the decompression stage, and performs pairwise matching directly on the compressed data. Similar work, using quantization, has been suggested by Chandrasekhar et al. [6]. We do not apply quantization, but rather use a lossless encoding.

Working on a shorter representation and saving the decompression process may save processing time, as well as memory storage. By using a lossless compression and applying the same norm for performing the pairwise matching we make sure not to hurt the true positives and false negatives probabilities. Moreover, representing the same set of feature descriptors in less space can allow us to keep a larger set of representatives, which can result in a higher probability for object identification by reducing the number of mismatches.

The main idea is 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 [2]) 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 does 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 $\{\boldsymbol{f_1}, \boldsymbol{f_2}, \ldots, \boldsymbol{f_n}\}$ 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 $\mathcal{E}(\boldsymbol{f_i})$, and an equivalent metric m so that for every $\varepsilon > 0$ there exists a $\delta > 0$ such that $\forall i, j \in \{1, \ldots, n\}$

$$\|\boldsymbol{f_i} - \boldsymbol{f_j}\|_M < \varepsilon \iff \|\mathcal{E}(\boldsymbol{f_i}) - \mathcal{E}(\boldsymbol{f_j})\|_m < \delta.$$

The rest of the paper is organized as follows. Section 2 gives a description of our lossless encoding for DSIFT feature vectors; Section 3 suggests improving the compression for PHOW (Pyramid Histogram Of visual Words) vectors [3]; Section 4 presents the algorithm used for compressed pairwise matching the feature vectors without decompression; Section 5 presents results on the compression performance of our lossless encoding for DSIFT descriptors and PHOW features, and the last section concludes.

2 Lossless Encoding for DSIFT

Given two SIFT feature vectors, we suggest in [12] 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*. It turns out that the Fibonacci Code is also suitable for DSIFT and PHOW feature vectors.

The Fibonacci code is a universal variable length encoding of the integers based on the Fibonacci sequence rather than on powers of 2. A subset of these encodings can be used as a fixed alternative to Huffman codes, giving obviously less compression, but adding simplicity (there is no need to generate a new code every time), robustness and speed [9,10]. 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

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

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 13, for the remainder 6 one can use the Fibonacci number 5, and the remainder 1 is a Fibonacci number itself. So one would represent 19 as $19 = 13 + 5 + 1$, yielding the binary string 101001. Note that the bit positions correspond to $F(i)$ for increasing values of i from right to left, just as for the standard binary representation, in which $19 = 16 + 2 + 1$ would be represented by 10011. Each such Fibonacci representation starts with a 1, so by preceding it with an additional 1, one gets a 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 [9] 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, ...}.

A disadvantage of this reversing process is that the order preserving characteristic 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.

We wish to encode 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, 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 applied *vlfeat Matlab*'s [1] 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 1.

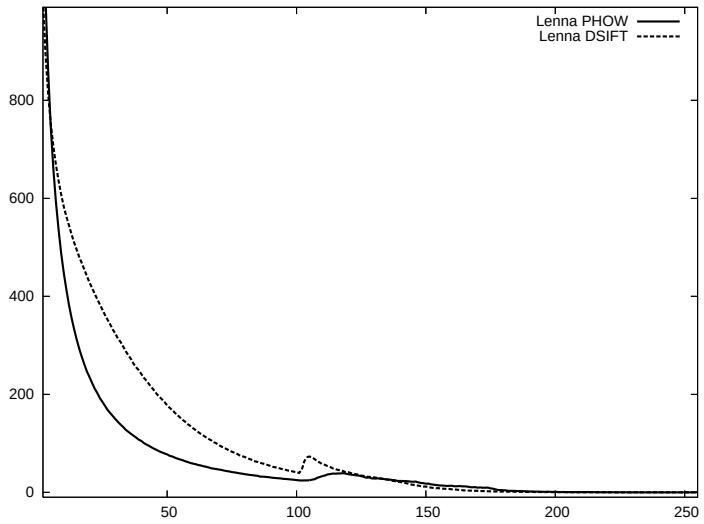

Figure 1. Value distribution in feature vectors.

The usual approach for using a 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 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 [11] for Huffman coding. Figure 1 shows that the order is not strictly monotonic, but that the fluctuations are small. Indeed, experimental results show that encoding the numbers themselves instead of their indices has hardly any influence (less than 0.4% on our test images).

3 Further compression of PHOW

DSIFT feature vectors contain repeated zero-runs, as could be expected by the high number of zeros. However, we noticed a difference between the zero-runs of DSIFT and PHOW feature vectors, leading 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 $k \geq 0$.

[1] `http://www.vlfeat.org/`

This special codeword assignment was empirically shown to be useful for PHOW but not for general DSIFT.

As example, consider the 25th PHOW feature vector of Lenna's Image, consisting of 128 coordinates, in which the first 20 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 this vector as

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

as we would do for general DSIFT, we rather encode it as

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

reducing 75 bits to 71, rather than 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 1 for the case of DSIFT and by 2 for the case of PHOW, the difference between two Fibonacci encodings is preserved, which is an essential property for computing their distance in the compressed form.

4 Compressed pairwise matching

The algorithm for computing the subtraction of two Fibonacci encoded coordinates was presented in [12] and is given here for the sake of completeness. We start with a general algorithm, Sub( ), for subtraction which is used in both 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 component, then the squares of these differences are summed. The algorithm for computing the subtraction of two corresponding Fibonacci encoded coordinates A and B is given in Figure 2. 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.

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.

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 Sub(A, B) above. As opposed to the computation of the DSIFT L_2 norm, which simply sums the squares

```
Sub(A, B)
scan the bits of A and B from right to left
a_1 ⟵ first bit of A
a_2 ⟵ second bit of A
while next bit of A exists {
    a_3 ⟵ next bit of A
    b_1 ⟵ next bit of B
    a_1 ⟵ a_1 − b_1
    a_2 ⟵ a_1 + a_2
    a_3 ⟵ a_1 + a_3
    a_1 ⟵ a_2
    a_2 ⟵ a_3
}
b ⟵ value of last 2 bits of B
if b ≠ 0 then b ⟵ 2 − b
return 2 ∗ a_1 + a_2 − b
```

Figure 2. Compressed differencing of DSIFT and PHOW encoded coordinates.

of the differences of two corresponding Fibonacci encoded coordinates, the PHOW encoding works as follows. The codeword 11, representing two consecutive zeros, needs a special treatment only if the other codeword, say B, is not 11. In this case, 11 should be replaced by two codewords 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 3, 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.

5 Compression performance

We considered three images for our experiments : *Lenna*, *Peppers* and *House*, which were taken from the SIPI (Signal and Image Processing Institute) Image Data Base[2]. We applied DSIFT and PHOW on all images getting for each 253,009 and 237,182 feature vectors, respectively. For comparison, the number of interest points on which the original SIFT features were generated for the three test images was 737, 832 and 991, respectively. Table 1 presents the compression performance of our Fibonacci encoding applied on the coordinates of the DSIFT vectors suitable for compressed matching as compared to other compressors. The second column shows the original sizes in MB. The other figures are compression ratios, defined as the original size divided by the compressed size, so that larger numbers indicate better compression. The third column presents the compression performance when each number is represented by its Fibonacci encoding. To evaluate the compression loss due to omitting the sorting of

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

```
PHOWL2Norm(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 3. PHOW compressed L_2 norm computation.

the frequencies, we considered the compression where each symbol is encoded using the Fibonacci codeword assigned according to its position in the list of decreasing order of frequencies. These values appear in the fourth column headed ordered. For comparison, the compression achieved by a Huffman code is also included as an upper bound.

Image	Original size	Fibonacci	Ordered	Huffman
Lenna	83.31	3.164	3.164	3.481
Peppers	83.88	3.113	3.114	3.455
House	82.80	3.184	3.185	3.486

Table 1. Compression efficiency of the proposed encodings for DSIFT Features.

Table 2 presents the corresponding results for the PHOW vectors. The third column presents the compression performance in which each number is represented by its Fibonacci encoding using the first codeword for encoding 00. The fourth column is ordered Fibonacci as defined in Table 1. Huffman encoding is given in the fifth column. This time it was applied on the alphabet including the symbol 00.

As can be seen, encoding the numbers themselves instead of their indices induces a negligible compression loss. The high probability for small integers also reduces the gap between the performances of Fibonacci and Huffman codes.

6 Conclusion

We have dealt with the problem of compressing a set of Dense SIFT feature vectors, and in particular on DSIFT and PHOW features, under the constraint of allowing

Image	Original size	Fibonacci	Ordered	Huffman
Lenna	70.664	3.854	3.859	4.046
Peppers	70.561	3.874	3.883	4.065
House	73.017	3.640	3.646	3.917

Table 2. Compression efficiency of the proposed encodings for PHOW Features.

processing the data directly in its compressed form. Such an approach is advantageous not only to save storage space, but also to the manipulation speed, and in fact improves 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 the 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 norm, whereas a Huffman code would have to use some translation table.

The experiments suggest that there is only a negligible loss in compression efficiency, of 1% and even less, relative to the ordered Fibonacci code, and only a small increase, of 4–10%, in the size of the compressed file relative to the size achieved by the optimal Huffman codes, which might be worth a price to pay for the improved processing.

References

1. H. Bay, T. Tuytelaars, and L. Gool: *SURF: Speeded Up Robust Features*, in European Conference on Computer Vision (ECCV), 2006, pp. 404–417.
2. J. Beis and D. G. Lowe: *Shape indexing using approximate nearest-neighbour search in high-dimensional spaces*, in Conference on Computer Vision and Pattern Recognition, 1997, pp. 1000–1006.
3. 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.
4. V. Chandrasekhar, M. Makar, G. Takacs, D. M. Chen, S. S. Tsai, N. M. Cheung, R. Grzeszczuk, Y. A. Reznik, and B. Girod: *Survey of SIFT compression schemes*, in Proc. Int. Workshop Mobile Multimedia Processing, 2010.
5. 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.
6. 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.
7. V. Chandrasekhar, G. Takacs, D. M. Chen, S. S. Tsai, J. Singh, and B. Girod: *Transform coding of image feature descriptors*, in SPIE Visual Communications and Image Processing (VCIP), 2009.
8. 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.
9. A. S. Fraenkel and S. T. Klein: *Robust universal complete codes for transmission and compression.* Discrete Applied Mathematics, 64 1996, pp. 31–55.

10. S. T. Klein and M. Kopel Ben-Nissan: *On the usefulness of Fibonacci compression codes.* The Computer Journal, 53 2010, pp. 701–716.
11. S. T. Klein and D. Shapira: *Huffman coding with non-sorted frequencies.* Mathematics in Computer Science, 5(2) 2011, pp. 171–178.
12. S. T. Klein and D. Shapira: *Compressed SIFT feature based matching.* To appear in Proc. The Fourth International Conference on Advances in Information Mining and Management, IMMM–14, 2014.
13. D. G. Lowe: *Distinctive image features from scale-invariant keypoints.* International Journal of Computer Vision, 60 (2) 2004, pp. 91–110.
14. 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.