Compressed SIFT Feature-based Matching

Shmuel Tomi Klein

Computer Science Department
Bar Ilan University, Israel
Email: tomi@cs.biu.ac.il

Dana Shapira

Computer Science Department
Ashkelon Academic College, Israel
Email: shapird@ash-college.ac.il

***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 descriptor vectors retaining the metric. Specifically, we concentrate on the Scale Invariant Feature Transform (SIFT), a known object detection method. Given two SIFT feature vectors, we suggest achieving our goal by compressing them using a lossless encoding for which the pairwise matching can be done directly on the compressed files, by means of a Fibonacci code. Experiments show that this approach incurs only a small loss in compression efficiency relative to standard compressors requiring a decoding phase.**

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

I. INTRODUCTION

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.

Invariance is obtained by using certain transforms, e.g., the one called Scale Invariant Feature Transform (SIFT) by Lowe [1], 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 large 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.

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.

The rest of the paper is organized as follows. Section 2 reviews some of the related work; Section 3 gives a brief description of SIFT; Section 4 presents our lossless encoding for SIFT feature vectors, especially suited for CFBM; Section 5 presents the algorithm used for compressed pairwise matching the feature vectors without decompression; finally, Section 6 presents results on the compression performance and the last section suggests how to extend this work.

II. RELATED WORK

Wagner et al. [2] developed object recognition algorithms especially designed for a restricted amount of available RAM, such as mobile phones. Wagner uses a fast corner detector for feature detection, and off-line preprocesses the features in different scales, while using only a fixed scale level, matching then on-line the phone's camera scale. Tackling this problem from another angle is by using good known methods for a non restricted Random Access Memory environment, but making them work in a compressed domain.

A feature descriptor encoder is presented in Chandrasekhar et al. [3]. They transfer the compressed features over the network and *decompress* them once data is received for further pairwise image matching. Chen et al. [4] 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 [6] 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.

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

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-364-3

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. [7]. We do not apply quantization, and rather use a lossless encoding.

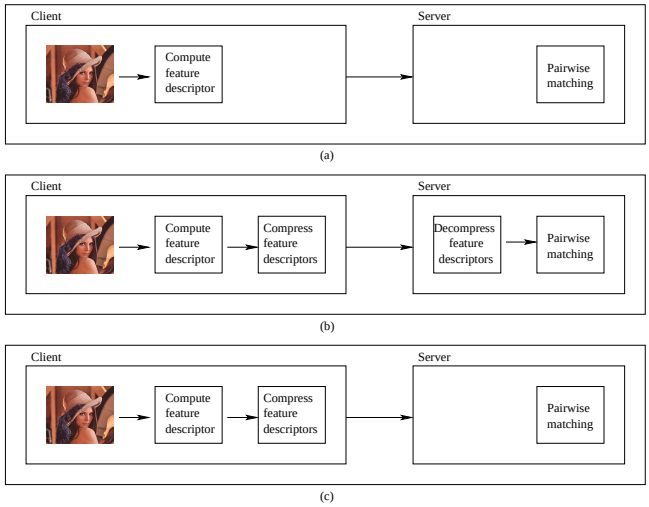

Figure 1. Block diagram showing (a) the traditional image retrieval system, (b) the scenario assumed by previous research, as opposed to (c) the compressed feature based matching problem.

Working on a shorter representation and saving the decompression process may save processing time, as well as memory storage, making 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 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) 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 $\{\vec{f_1}, \vec{f_2}, \ldots, \vec{f_n}\}$ 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 $\forall i, j \in \{1, \ldots, n\}$

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

This is an extension of the Compressed Pattern Matching paradigm introduced by Amir and Benson [8]. 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 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. In our case, previous work on feature compression would use complementary encoding and decoding functions $\mathcal{E}$ and $\mathcal{D}$, and apply decompression on the vectors, so that $\mathcal{D}(\mathcal{E}(\vec{f_i})) = \vec{f_i}$.

III. 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 blurred images are used to generate another set of images. 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.

The object detection and identification is done by pairwise matching the feature vectors of the query image against a large database of local image features using the L_2 norm.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-364-3

IV. LOSSLESS ENCODING FOR SIFT FEATURE VECTORS

Given two SIFT feature vectors, 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* [9]. 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.

A. 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 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 [10], [9]. 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 $F(0) = 1, F(1) = 2$ and $F(i) = F(i-1) + F(i-2)$, 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 [10] 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 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.

B. Using a Fibonacci Code for SIFT Vectors

We wish to encode SIFT 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 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 Matlab's SIFT application on it, generating 738 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.

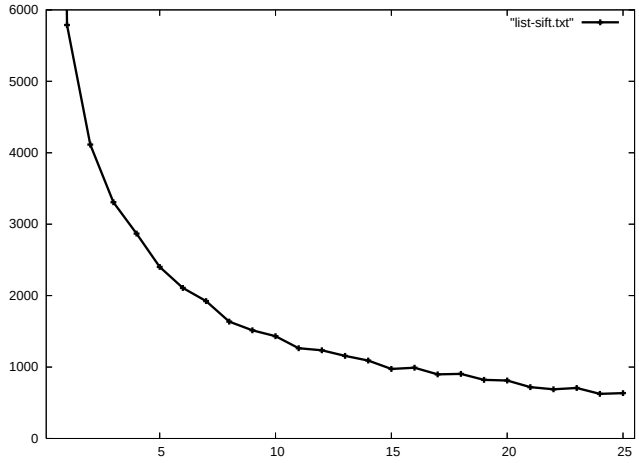

Figure 2. Value distribution in a feature vector.

Feature vectors also contain repeated zero-runs, as could be expected by the high number of zeros. We therefore chose 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 [11] 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 has hardly any influence (0.1% on our test images).

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-364-3

As an example, consider a feature vector of 128 coordinates, the first 20 of which are

$$0, 0, 0, 0, 0, 0, 0, 0, 10, 3, 6, 4, 0, 0, 2, 4, 10, 83, 69, 0, \ldots$$

corresponding to a point of interest of Lenna's Image. The Fibonacci encoding of this feature vector is

11 11 11 11 101011 00011 000011 10011 11

1011 10011 101011 1000101011 0010010011 011...

using 70 bits, rather than 160 bits for the first 20 elements of the original SIFT vector. 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.

V. Compressed Pairwise Matching

Given two compressed feature vectors 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 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 bits of A not empty
{
  $a_3 \longleftarrow$ 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 $b \longleftarrow 2 - b$
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 an example, consider the numbers $A = 130$ and $B = 65$, encoded by the strings representing 132 and 67, which are 10001001011 and 1010100011, respectively. 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. At the end, b_1 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), but the algorithm would still work correctly. In that case, the values in the last line would be -14 and -25, and 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	

Figure 4. Example of direct differencing.

L2Norm(V_1, V_2)
while V_1 and V_2 are not empty
{
  remove first codeword from V_1
    and assign it to A
  remove first codeword from V_2
    and assign it to B
  if $A \neq B$ then
    if $A = 11$ then
      $S \longleftarrow$ Sub$(B, 011)$
      $V_1 \longleftarrow 011 \parallel V_1$
    else if $B = 11$ then
      $S \longleftarrow$ Sub$(A, 011)$
      $V_2 \longleftarrow 011 \parallel V_2$
    else $S \longleftarrow$ Sub(A, B)
  SSQ $\longleftarrow$ SSQ $+ S^2$
}
return $\sqrt{\text{SSQ}}$

Figure 5. Compressed differencing of the coordinates.

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. 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 the algorithm of Figure 5, where $\parallel$ denotes concatenation and SSQ is initialized to 0.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-364-3

VI. COMPRESSION PERFORMANCE

We considered three images for our experiments: *Lenna*, *Peppers* and *House*, which were taken from the Signal and Image Processing Institute Image Data Base [12]. We first applied SIFT on all images receiving 737, 872, and 991 interest points, respectively. Table 1 presents the compression performance of our Fibonacci encoding suitable for compressed matching as compared to other compressors. The second column shows the original sizes of the SIFT feature vectors in bytes. The third column, headed Fib, presents the compression performance, as a percentage of the original size, in which each number is represented by its Fibonacci encoding, which is useful for compressed pairwise matching. 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 decreasing order of frequencies. These values appear in the 4th column headed Ordered Fib.

For comparison, the compression achieved by a Huffman code is also included in the fifth column as a lower bound. 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.

TABLE I. COMPRESSION EFFICIENCY OF THE PROPOSED ENCODINGS (IN PERCENT OF ORIGINAL SIZE).

Image	Original Size	Fib	Ordered Fib	Huffman	gzip	bzip
Lenna	236,382	27.82	27.78	26.2	34.7	30.7
Peppers	279,422	27.3	27.2	25.7	34.3	30.4
House	325,778	29.5	29.4	27.3	35.6	31.7

The last two columns give the compression performances of gzip 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 SIFT feature vectors, and they require a sequential scan from their beginning for the decoding. In this particular case, their compression is also worse than that of Huffman or Fibonacci. This can be explained by the fact that they need to encode also the separating blanks or newlines between the elements of the feature vectors, which constitute a substantial part of the files, whereas Huffman and Fibonacci encode the elements themselves, and not their representations, so the original file can be reconstructed without having to encode the separators explicitly.

VII. CONCLUSION AND FUTURE WORK

We have dealt with the problem of compressing a set of feature vectors known as SIFT, under the constraint of allowing 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 small loss, of 6–8%, in compression efficiency relative to the optimal Huffman codes, which might be worth a price to pay for the improved processing. Relative to other standard compressors, like gzip or bzip, there is even an improvement in compression, contrarily to what one might expect on text files, for example. This is due to the fact that the separators between the vector elements need not be encoded in the Fibonacci approach.

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. A common choice is the edit distance metric, in which the distance between two strings is defined as the minimum number of insertions, deletions or substitutions of single characters performed on one of the strings in order to convert it to the other. The case where $k = 0$ corresponds to the classical pattern matching problem.

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 by Amir and Benson [8] as an open problem. It has been solved for many cases, e.g., for byte Huffman coding of words [13], for run length encoded strings [14], for Lempel-Ziv compressed text in [15][16], and Straight Line Programs [17][18].

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 a corresponding metric $\| \quad \|_m$ 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 $\mathcal{E}(T)$. However, this raises some difficulties, as an occurrence of $\mathcal{E}(Q)$ in the encoded file does not necessarily correspond to an occurrence of an approximated pattern. For example, if the encoded file uses Huffman coding, an occurrence of the compressed pattern $\mathcal{E}(P)$ might appear in the encoded file, without implying that there is a corresponding occurrence of the original pattern in the original file, since

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-364-3

$\mathcal{E}(P)$ is not necessarily aligned on codeword boundaries. We intend to deal with these extensions in future work.

REFERENCES

[1] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," International Journal of Computer Vision, vol. 60 (2), 2004, pp. 91–110.

[2] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, and D. Schmalstieg, "Pose tracking from natural features on mobile phones," in Proceedings of the International Symposium on Mixed and Augmented Reality, 2008, pp. 125–134.

[3] V. Chandrasekhar et al., "Transform Coding of Image Feature Descriptors," in Visual Communications and Image Processing, vol. 7257 (1), 2009, pp. 725 710–725 710–9.

[4] D. M. Chen et al., "Tree Histogram Coding for Mobile Image Matching," in Data Compression Conference, DCC–09, 2009, pp. 143–152.

[5] V. Chandrasekhar et al., "Compressing Feature Sets with Digital Search Trees," in ICCV Workshops, 2011, pp. 32–39.

[6] V. Chandrasekhar et al., "Survey of SIFT compression schemes," in Int. Workshop on Mobile Multimedia Processing (WMMP), 2010.

[7] V. Chandrasekhar et al., "Compressed Histogram of Gradients: A Low-Bitrate Descriptor," International Journal of Computer Vision, vol. 96(3), 2012, pp. 384–399.

[8] A. Amir and G. Benson, "Efficient two-dimensional compressed matching," in Data Compression Conference DCC–92, Snowbird, Utah, 1992, pp. 279–288.

[9] S. T. Klein and M. Kopel Ben-Nissan, "On the Usefulness of Fibonacci Compression Codes," The Computer Journal, vol. 53, 2010, pp. 701–716.

[10] A. S. Fraenkel and S. T. Klein, "Robust universal complete codes for transmission and compression," Discrete Applied Mathematics, vol. 64, 1996, pp. 31–55.

[11] S. T. Klein and D. Shapira, "Huffman Coding with Non-Sorted Frequencies," Mathematics in Computer Science, vol. 5(2), 2011, pp. 171–178.

[12] [Online]. Available: http://sipi.usc.edu/database/

[13] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates, "Fast and flexible word searching on compressed text," ACM Trans. Inform. Syst. (TOIS), vol. 18 (2), 2000, pp. 113–139.

[14] V. Mäkinen, G. Navarro, and E. Ukkonen, "Approximate Matching of Run-Length Compressed Strings," Algorithmica, vol. 35 (4), 2003, pp. 347–369.

[15] 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.

[16] J. Kärkkäinen, G. Navarro, and E. Ukkonen, "Approximate string matching on Ziv-Lempel compressed text," Discrete Algorithms, vol. 1 (3-4), 2003, pp. 313–338.

[17] P. Bille et al., "Random access to grammar-compressed strings," in Symposium on Discrete Algorithms (SODA), 2011, pp. 373–389.

[18] T. Gagie, P. Gawrychowski, C. Hoobin, and S. J. Puglisi, "Faster Approximate Pattern Matching in Compressed Repetitive Texts," ISAAC, 2011, pp. 653–662.

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-364-3