IMAGE DATABASE INDEXING USING JPEG COEFFICIENTS

Sanjiv K. Bhatia
Department of Mathematics & Computer Science
University of Missouri – St. Louis
St. Louis, MO 63121
sanjiv@aryabhat.umsl.edu

Abstract

In this paper, we describe an image database system that can retrieve images that are similar to a query image. The query image is processed to extract information that is matched against an index to provide pointers to similar images. The salient feature of the system is that the index is developed from the JPEG-compressed images without first having to uncompress them. We use a similarity measure to determine the difference between the JPEG coefficients of the query image to those of the images in the database index. The index itself is developed by partitioning the JPEG-compressed image into a quad-tree structure and computing certain characteristics for each quad at each level in the tree. Each quad representation is stored in tables corresponding to the quad size in a relational database that can be queried by conventional means. Given a query image, or image segment, the corresponding quad tree-based representation is computed. Then, each of the quad representations in the query is used to search the index for the matching quads in the database. Initial experiments with the index have provided encouraging results. The system outputs a set of ranked images in the database with respect to the query using the similarity measure, and can be limited to output a specified number by changing the threshold match.

1 INTRODUCTION

An image database is used to store, retrieve, and manipulate pictorial data in an efficient manner. The pictorial data is difficult to process as the techniques to extract elements that make the picture are not yet perfected. This makes the problem of matching well-defined parts of a picture extremely difficult. Furthermore, the parts of a picture themselves may not be properly defined due to the difficulties of extracting the picture elements, for example due to the use of different edge detectors. Another problem in query processing is that the pictures are stored in compressed file formats, e.g., GIF, JPEG, and TIFF [7], and it is more desirable to compare the pictures against a query using the compressed version, if possible.

The last constraint suggests that it will be efficient to develop a technique that can compare two images in compressed domain to determine their similarity to each other. Such a solution has to compare parts of pictures without uncompressing them, and consequently, without extracting the picture elements that constitute the whole image. This assumes that the parts of the pictures are somehow compressed such that similar portions of the images contain the same kind of data. It is also desired that the output of the process can be quantified in a meaningful way so that different images in the database can be ranked by the perceived relevance to a query image. The process is even more efficient if we can develop an index that will assist in the lookup of the entire database, without having to compare the query image against every image in the database in its entirety [4].

In image databases, indexing is generally used to identify spatial constraints between different picture elements. Such indexing requires that the picture elements be extracted from each image and their mutual relationship be used to develop the index [2, 3, 9]. However, the techniques suffer from the limitation of inadequate methods available to *recognize* the individual picture elements, if they can be properly extracted. Moreover, developing an index is computationally expensive, even for an image data base of moderate size [9, 10]. In addition, a user should have the ability to dynamically update the index to reflect the addition and deletion of new images in the

[0]

database.

In this paper, we present a technique to develop an index for an image database using the JPEG coefficients of the compressed image. The use of JPEG coefficients obviates the need for uncompressing the image and can be easily achieved by decoding the run-length encoded strings of coefficients. The index is based on partitioning the JPEG-compressed image into a quad-tree structure [11]. Then, we compute some simple statistics on each node in the quad tree and include those in the index along with the identification and location information. The quads at any level are of the same size. All the quads on a given level in the tree are collected in a relation, with quads at different levels collected in separate relations.

A given query image is partitioned similarly into the quad-tree structure and statistics from different nodes are compared against the statistics from the quads of same size from the index relations. The comparison is quantified as a distance measure that can be used to determine the similarity of the query to different images in the data base. Using a threshold, some of the images can be ignored from further comparison yielding further improvements in retrieval efficiency. Finally, an adjustment of threshold yields the desired number of images that match the query image.

The need for such an index is emphasized in research reports on existing image databases such as QBIC [1, 4, 6]. The research presented in this paper is even better than that it develops the index in compressed domain, without having to uncompress the images.

In the next section, we present the basics of JPEG compression, with an emphasis on the point where we pick up the coefficients. In Section 3, we show the developmnet of quad tree corresponding to the DCT coefficients and the information to be retained at each node of the quad tree. In Section 4, we illustrate the query process, followed by the conclusion. Section 5 describes the implementation status of the system and concludes by presenting the future plans regarding the system.

2 JPEG IMAGE COMPRESSION

JPEG derives its name from Joint Photographic Experts Group and is a well-established standard for compression of color and grayscale images for the purpose of storage and transmission [5, 7, 8]. The JPEG compression results into an image that is considerably similar in quality to the original image while using far less bytes, typically from 20:1 to 25:1 bytes without a perceivable quality degradation. The compression itself is lossy which means that some information in the image may be lost depending on the desired amount of compression.

The minimal subset of the JPEG compression standard, known as the *baseline* JPEG, is based on the discrete cosine transform (DCT). First, each pixel in the image is level-shifted by 128. Then, the image is divided into fixed size blocks and DCT is applied to each block. Application of DCT to an image block results in removal of the redundant data, and the DCT coefficents for the block are obtained. These coefficents are then quantized using weighting functions optimized for the human eye. The resulting coefficients are encoded using a Huffman variable word-length algorithm to remove redundancies [7].

The compression process is started by dividing the rectangular image canvas into 8×8 blocks and the DCT is applied to each block to separate the high- and low-frequency information in the block. Application of DCT results in the average value (or DC component) in the location $(0, 0)$ while the other locations of the 8×8 block contain the AC terms. The AC terms keep the higher frequency components of the block. The maximum value for an AC term, the *dominant coefficient*, represents the highest change in the cosine wave between any two adjacent pixels in the block.

The DCT coefficients in each block are quantized to get scaled coefficients by dividing each value with a quantization coefficient from the quantization table developed by the ISO JPEG [5, 8]. The coefficients in the quantized block are rearranged using a zigzag ordering to create a vector. The zigzag pattern approximately orders the basis functions from low to high spatial frequencies [8]. The vectors resulting from the zigzag ordering contain the DC coefficient corresponding to the original image in the first location. The last several locations in the vector contain zeroes. The baseline JPEG standard requires these vectors of DCT coefficients to be run-length encoded, using Huffman encoding, for storage and transmission. However, we want to use the vectors without encoding and extract some useful data from each vector to facilitate comparison of images. In the remainder of this paper, we will use the coefficient vectors resulting from the zigzag ordering.

3 QUAD TREE STRUCTURE REPRESENTATION OF DCT COEFFICIENTS

In the previous section, we showed how to extract the DCT coefficients for each 8×8 block of the image. This effectively implies that the entire image is considered to be constructed of 8×8 pixel blocks. We consider the 8×8 DCT-encoded block to be the smallest addressable unit of the image, instead of each pixel. In this section, we show the construction of quad tree structure using the information from each 8×8 DCT-encoded block.

To develop the quad tree, we consider an image of size $r \times c$ with r rows and c columns of pixels. After the application of DCT, we obtain a two-dimensional set of vectors that can be described by r' rows and c' columns of vectors, such that

$$r' = \left\lceil \frac{r}{8} \right\rceil$$
$$c' = \left\lceil \frac{c}{8} \right\rceil$$

The two-dimensional set of vectors of size $r' \times c'$ is then superimposed on a square array of size $R \times C$, $(R = C)$, where

$$R = 2^{\lceil \lg(\max(r', c')) \rceil}$$

With R being an exponent of 2, each node in the quad tree can be subdivided into subnodes where the part of the set of two-dimensional vectors is of a side that can be described by an integer, except at the leaf node. Next, we describe the contents of different nodes in the quad tree.

Leaf nodes of the quad tree.

Each leaf node of the quad tree corresponds to an 8×8 vectors of DCT-encoded coefficients that has been computed from an 8×8 pixel block in the original image. The information of interest is the location of the original 8×8 pixel block in the image, the DC coefficient in the DCT-encoded vector (the first element in the vector), the dominant coefficient in the vector, and the location of the dominant coefficient. The location of the pixel block is taken to be the location of pixel in the top left corner in the block. In addition, we also retain the name of the image for identification purpose.

Non-leaf nodes and the quad tree.

The non-leaf nodes contain information extracted from aggregates of 8×8 DCT-encoded vectors. They will be used to make similarity comparisons between images at a *higher* level than the individual blocks. Each non-leaf node in the quad tree contains the location of an aggregate block of 8×8 DCT-encoded coefficient vectors, the location of the corresponding aggregate block in the original image, the dominant coefficient in the entire aggregate block of DCT-encoded coefficient vectors, and the location of the dominant coefficient. Just like in leaf nodes, we retain the name of the image for identification purpose. Lastly, the non-leaf nodes also contain the links to their four children, identified respectively as `ul` (upper left), `ur` (upper right), `ll` (lower left), and `lr` (lower right).

It may be noted that a leaf node uses the same structure as described above by assigning a null value to the four children. The quad tree itself is illustrated in Figure 1. In this figure, we start with the set of

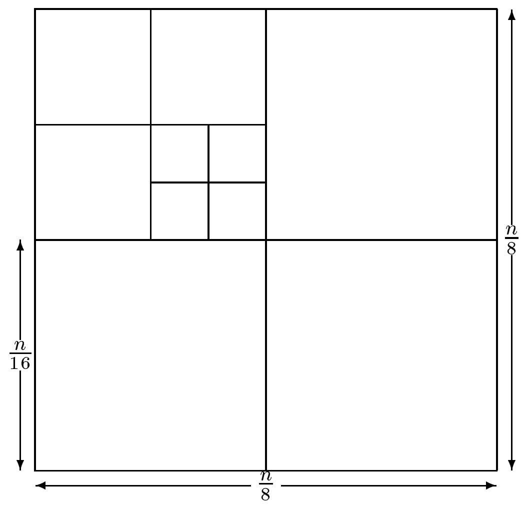

Figure 1: Quad tree of an image

8×8 DCT-encoded coefficients forming a $\frac{n}{8} \times \frac{n}{8}$ square corresponding to an $n \times n$ image at the root. The set of coefficients is divided into four sets of $\frac{n}{16} \times \frac{n}{16}$ coefficients each. We continue to subdivide each set into four subsets recursively until we get a set containing only one 8×8 DCT-encoded coefficient. At each node in the tree, we calculate the information in the node.

The nodes in the tree are at distinct levels that can be identified by the size of the image. We save the nodes depending upon their level in the tree in a file named as `indx`*nnn*`.idx` where *nnn* is indicative of the size of the *dct*-encoded segment at that level. It is easy to see that all the segments at a given level are of the same size. The nodes are saved by performing a level-order traversal of the quad tree with the following collating sequence: `ul`, `ur`, `ll`, and `lr`.

4 THE QUERY PROCESS

Once the quad tree has been developed for each image in the database and the required information saved in the index files, the query process is fairly simple. The query in this case is considered to be a JPEG image. The image is processed to develop the quad tree of its DCT-encoded coefficient blocks as described in Section sec:quad. Further steps in the query rely completely on the quad tree.

The root of the quad tree contains information about the size of the image. Since each index file contains information about the nodes of the same size, we select the index file that contains nodes with size equal to the size of the image. It may be noted that the size of the node is represented in terms of an exponent of 2 and therefore, the image is also fit into a larger *template* with size rounded to the next exponent of 2. The overall result is that the we only consider those images in the database that are larger than or equal to the query image. The root node of the quad tree constructed from the query is compared against each node in the selected index file using the expression

$$\frac{|c_q - c_n|}{\max(d_q, d_n)}$$

where c_q and c_n are the average DC coefficients for the nodes in the query and the index file, respectively, and d_q and d_n are the corresponding dominant coefficients.

It is easy to see that in the case of a perfect match, e.g., the same image in query and the database, the above comparison will evaluate to zero. The images that result in the comparison greater than a prespecified threshold can be ignored from further consideration. However, it may be the case that two images which are not similar to each other in content but are similar in average intensity (the DC component for the image), may result in the comparison evaluation of zero. These images need to be ranked using the information in the lower nodes in the quad tree.

While comparing the lower nodes, we must keep track of the relative location of each subblock at the node in the query as well as the database and must compare the corresponding nodes only. The index has been structured such that each index file contains the nodes at a given level within the quad tree. Traversing the quad tree corresponding to the query in level order, each node is compared with the corresponding node in the image database, with the root node as the reference, and the evaluation of the comparison added to the previous evaluation. At any stage, if the summation of evaluation becomes too large, the image can be removed from further consideration. We can also limit the number of images that should be kept under consideration by keeping only the top-ranked images in consideration.

If the size of the image in the index file is much larger than the image in the query, we may have more than one comparison on the same image for the query. This works fine as long as the restriction of using the root node as the reference is accounted for. At the very last stage, if we have two comparisons in the same picture in the database for the query being too small compared to the database image, we use the smallest of the evaluated comparisons.

The overall results of the query are summarized in a ranked list of images that are sorted on the basis of their distance from the query. The ranking in the list allows us to present the resulting images in the decreasing order of relevance with respect to the query. In addition, we can also limit the number of images that are to be presented to the user.

5 IMPLEMENTATION STATUS

The image database system described in this paper has been implemented on a Sun SPARC system in C. The images used in testing the system were scanned as Sun raster files using a flat bed scanner, and converted to JPEG using the public domain package `xv`. We used gray scale images (the luminance component) for testing the system.

Each raster image was processed to determine its DCT coefficients and the corresponding zigzag vectors. A quad tree was computed for each image and index files were developed using the quad tree. The query program was given an image as input with the result being a ranked list of images from the database. Currently, the program just gives the name of the image and a number to specify its distance from the query image. The results from a query are displayed in Figure 2. These images show the query image (top) and the images from the database along with their distance from the query.

5.1 Future Work

For the immediate future, we are working on creating a nice front end for the system under the X windows environment. In addition, we are working on a system that will work with color images. For the long term goal, we would like to develop a system that will examine the zigzag vectors to provide a better indication of image match than just the intensity-based approach. We also plan to validate our techniques using larger databases such as the one from Smithsonian Institute.

There are few techniques available for performing content-based retrieval. Even fewer projects with indexing the content have been reported in the literature. The QBIC team has argued for the need of "indexing, clustering, and filtering methods" but has left it for future releases [4]. In relation to that, the techniques presented in this paper go one step ahead of QBIC as it performs the comparison *without uncompressing the images*, and has indexing built into the matching method.

References

[1] R. Barber, W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and P. Yanker. Efficient query by image content for very large image databases. In *COMPCON*, pages 17–19. IEEE Computer Society Press, Los Alamitos, CA, 1993.

[2] S. K. Bhatia and C. L. Sabharwal. A fast implementation of a perfect hash function for picture objects. *Pattern Recognition*, 27(3):365–376, 1994.

[3] C.-C. Chang and S.-Y. Lee. Retrieval of similar pictures on pictorial databases. *Pattern Recognition*, 24(7):675–680, 1991.

[4] M. Flickner, et. al. Query by image and video content: The QBIC system. *IEEE Computer*, 28(9):23–32, September 1995.

[5] R. C. Gonzalez and R. E. Woods. *Digital Image Processing.* Addison Wesley, Reading, MA, 1992.

[6] B. Holt and L. Hartwick. Retrieving art images by image content: the UC Davis QBIC project. *ASLIB Proceedings*, 46(10):243–248, October 1994.

[7] J. D. Murray and W. vanRyper. *Encyclopedia of Graphics File Formats (2nd ed.).* O'Reilly, Sebastopol, CA, 1996.

[8] W. B. Pennebaker and J. L. Mitchell. *JPEG Still Image Data Compression Standard.* van Nostrand Reinhold, New York, 1993.

[9] C. L. Sabharwal and S. K. Bhatia. Perfect hash table algorithm for image databases using negative associated values. *Pattern Recognition*, 28(7):1091–1101, July 1995.

[10] C. L. Sabharwal and S. K. Bhatia. Image databases and near-perfect hash table. *Pattern Recognition*, 1997. (Accepted for publication).

[11] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

Distance = 0.00

Distance = 0.620

Distance = 0.864

Distance = 1.118

Figure 2: Results from a query, with the distance of the picture from the query