Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Navigating through a Forest of Quad Trees to Spot Images in a Database

H.G.P. Bosch, N. Nes, M.L. Kersten

Information Systems (INS)

INS-R0007 February 29, 2000

Report INS-R0007
ISSN 1386-3681

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Navigating through a Forest of Quad Trees to Spot Images in a Database

Peter Bosch, Niels Nes, Martin Kersten
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

ABSTRACT

This paper describes how we maintain color and spatial index information on more than 1,000,000 images and how we allow users to browse the spatial color feature space. We break down all our images in color-based quad trees and we store all quad trees in our main-memory database. We allow users to browse the quad trees directly, or they can pre-select images through our color bit vector, which acts as an index accelerator. A Java based GUI is used to navigate through our image indexes.

1991 ACM Computing Classification System: Data structures (E.1), Data storage representations (E.2), Information search and retrieval (H.3.3), Segmentation (I.4.6), Feature measurement (I.4.7)
Keywords and Phrases: Quad-tree, color information, large volume image storage, content based image retrieval
Note: Work carried out under project INS-1 'MIA.'

1. INTRODUCTION

We address the problem of efficient storage and quick searching through an image database with more than 1,000,000 images based on color content and spatial relations. When such large image databases are used, it is clear that the raw material cannot be analyzed on-line. For a particular sub-image, pre-calculated image features organized in an index are a prerequisite for fast responses. Also, given the amount of images, and the impatience of the average web-user, the analysis of the image features must not be a CPU intensive operation and slow devices such as disks should not be in the data path when image-index information is searched: users want an answer before they hit the 'reload' button on their web-browser. This paper describes our storage techniques that enable quick color and spatial based searches in our main-memory image database.

Given that computer visions techniques are far from generally understood, we attack the problem of content-based image retrieval in a large image database from a different angle: we break down the image set into a set of simple color-based features with spatial information and we allow the user to browse and search the color features. Unlike the predominant approach off casting features into an abstract multi-dimensional space, we make all our indexes explicit to the user: the user is aware of spatial color indexes and what precisely is stored in them. Also we try to help the user in improving its query by trying to make explicit *why* images have been selected from the database by showing the selection criteria. This provides the basis for better user articulation of image queries.

The reason for allowing the user direct access to the basic features is because we believe that the user is quite capable of formulating queries against raw image features, once the user is aware of how the index can help in trimming the candidate set. If basic features and index structures are hidden from the users, they will find browsing the image set difficult. We have started from this perspective by our frustration of using web-based image search engines. It is often difficult to formulate a query that leads to the desired result, because it is unclear how the feature index is built up. If the search index structure would be more transparent, searching the database may be more effective. Another reason for allowing users direct access to the main indexes is to learn if users appreciate such raw information and to assess that color indexing is indeed a valuable paradigm for searching large image databases.

Our main image index is a forest of spatial based color quad trees [6], much like as is done by Lu *et al.* [4]. Each image in the database is broken up into a quad tree where each node of the quad tree describes the color content of a region of the image in the hue color space. So, to learn of the color content of a particular area in

the image, one needs to find the node in the quad tree that covers the area.

All quad trees of all images are stored in the Monet main-memory database [1]. We use a 64 GB main-memory machine to store all image quad trees. Since the entire index can be stored in main-memory, we do not have to worry about secondary storage performance while browsing the index: we only have to worry about the space requirements of our index. It also implies that exhaustively searching the entire index for a particular spot is not entirely unpractical.

All nodes in the color quad trees are called color features. We currently use the COREL image set and in particular 60,000 images with a size of 384×256. Usually, 5,000-10,000 color features are required to describe an image. A pixel that is described by a color feature is said to be *covered*. Our quad trees usually obtain a coverage of approximately 70% with a moderately deep index; *i.e.* with 5,000-10,000 image features we usually describe the colors and their spatial location of 70% of an image.

On top of the forest of quad trees, a color bit vector is laid out to allow quick indexing of the large set of images. This color bit vector structure is based on color histograms, and allows retrieval of images based on just a few colors that are searched for. If, for example, a user is looking for an image spot that has a few distinct colors, the user only needs to calculate the bit vector for that point to find all images satisfying the constraint. More elaborate search mechanisms are then used on the reduced quad-tree set.

We realize that our index is based on simple features only. Our aim with this work is not to build the best possible feature set of a given image set, but rather to look into the database issues for maintaining such large image sets. Our interests lie in databases storage issues as well as how to allow the users to browse our image database intelligently, *i.e.* how do we present the image features to the users so that they can navigate through the database to find their desired image. We acknowledge that other image feature spaces can be and must be integrated.

This paper acts as a foundation paper of our work: it describes the basic index structures of our system. In particular, in Section 3 we describe in detail our quad-tree break-down algorithm and we present some basic statistics when the break-down algorithm is applied on a part of the COREL image set. In Section 4, we present our color bit vector, which is used as an accelerator for our quad-tree index. Lastly, in Section 5 we present the ImageSpotter, a Java based GUI front-end for our database.

2. RELATED WORK

The most common automatic feature extractor is based on color spaces. Usually the red, green and blue pixels are first transformed into a hue, saturation and value space before they are summarized in a color histogram. Images are then compared based on their color histogram. Swain *et al.* [7] describe an L_1 metric which is used to compare color histograms as a measure to select images from a image set. This L_1 metric is used to find similar histograms based on a sample histogram. The problem with such approaches is that they are often CPU intensive to calculate and, once the metric has been calculated over a set of images, it is difficult to explain to the user what has been calculated: all one can present is a number that somehow represents the distance between two histograms. As similar argument can be applied to QBIC's weighted Euclidean distance function [3].

Lu *et al.* [4] also describe a color-based quad tree approach: an image is segmented using the quad tree approach and for each node in the quad tree, a histogram is generated. The advantage of this approach is that spatial information is kept with the color distribution. The major disadvantage of this approach is that it requires a substantial storage space [5]. When compared to our approach: we only store the dominant color of a segment in our database to lessen the storage space effects. We can approximate the original histogram by scanning the quad tree.

Rui *et al.* [5] argue that in order to make content-based image retrieval from large sets of databases truly scalable, one needs to resort to high-dimensional (in the order of 10^2) feature vectors, which are combined in a non-Euclidean way. The reason for the high-dimensionality is that there does not exist a single indexing structure that allows a comprehensive analysis of the diverse image set. One needs to use a non-Euclidean similarity measure as Euclidean similarity measures may not effectively simulate human perception. The common approach is that a feature vector function is presented that is able to compute some sort of ordering of the feature spaces. The problem with such feature vectors is that designing such a combining function is likely to be a black art. Further, once the function presents an ordering on a set of images, it may be hard for the user

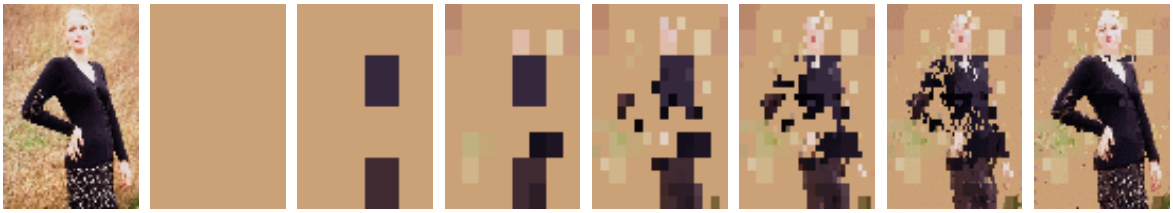

Figure 1: Left to right, top to bottom: the original image, and the image at various depths with 1, 3, 14, 56, 172, 553 and 5211 features.

to understand why images have been ordered in that order. Also, the only way to calculate the ordering is by considering all or at least most images. Given that there may be more than 1,000,000 images in a database and one may be looking for sub-images, this operation may not meet the patience of a web-user.

ExSight [8] and Blobworld [2] both provide a form of query and result visualization that can be used to articulate the image retrieval query. Both systems start by automatically segmenting the images in the database. The user can then specify the queries by selecting segments from sample images and spatially arrange them to form a spatial image query. The systems search for all similar images based on these spatial arrangements using the segments features. The results are visualized with an emphasis on matching segments and the used features.

3. IMAGE QUAD TREES

We use the hierarchical quad tree for efficient storage of the 'color contents' of an image. At each level a rectangle is split into four sub-rectangles. In each rectangle we determine the dominant color (in the hue color space based on 256 color bins), and if each of the four sub-rectangles contributes at least 10 % of this dominant color, we define a *color feature*. Next, we wipe out the dominant color pixels and proceed until all pixels have been dealt with, or a rectangle cannot be split into four sub-rectangles (when either the width or height of a rectangle consists of one pixel). We store all color features with their image location in the database.

As an example, Figure 1 graphically shows the image features at various depths into the image. While the left figure shows the original image, the pictures on the right hand side present the contents of the quad tree at various levels. Clearly, only a limited number of features is required to learn the contents of an image. We exploit this phenomenon with our search algorithms.

The advantage of breaking up an image using a quad tree is that it is an efficient storage mechanism of the color contents. Only a limited number of features are required to describe the image. Figure 2 shows a cumulative distribution of the number of features per file that are required to describe the spatial color contents of a set of images from the COREL image set. The mean number of features is 6807.6 for the 60,000 images, the standard deviation is 2536.3.[1]

Each color feature is described by a tuple with the following fields: *(ID, H, S, V, Weight)*. The *ID* field represents the x, y, width and height of a particular rectangle in an image: the upper rectangle has number zero, and each of its four sub-rectangles are numbered one to four from left to right, top to bottom. Each of the four sub-rectangles are broken up and numbered in the same fashion until all rectangles are numbered. The *H* represents the hue value of the dominant color in a rectangle, *S* and *V* represent the average saturation and intensity of a rectangle. Lastly, *Weight* represents the number of pixels in the dominant color in the selected rectangle. Note that at the deepest level in the image (*i.e.* the level at which a rectangle cannot be broken up anymore), it can happen that a rectangle does not satisfy the color feature distribution constraint. In this case we still define a feature for the dominant color, if only to be able to graphically represent an image by its features alone (see, for example, the right-hand side of Figure 1).

Monet, our database [1], is used to store all image features as is shown in Figure 3. Monet uses *Binary Association Tables* (BATs) to organize all database information. Each table only holds a *head* and a *tail* value.

[1] Note that all images we have used have a size of 384×256. When larger images are used, more features are found per image.

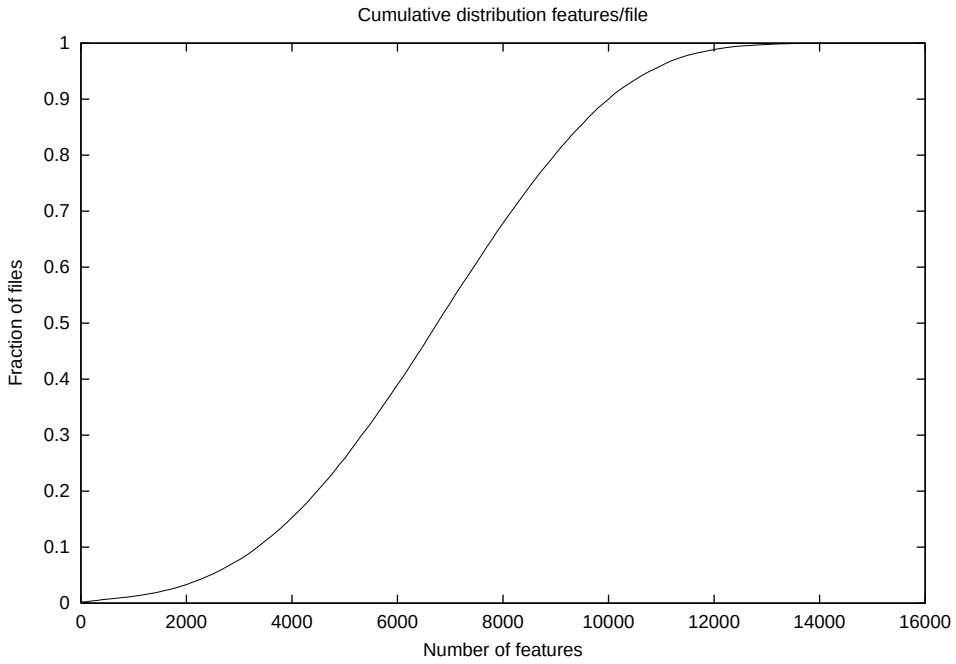

Figure 2: Cumulative distribution of number of features for all files in the COREL image set.

Each of the four feature elements is added to their table and the entire tuple can be addressed by joining the tables on their heads. These heads are Monet *ObjectID* (OID) and are assigned in a strict ascending order, which means that they do not have to be physically stored. One extra table (the *Image key* table in Figure 3) is used to link the virtual OIDs from the feature tables to the images themselves: each virtual OID maps onto an image name.

Currently we use thirteen bytes to encode a feature. The hue, saturation and value are encoded in single bytes, and the ID and the weight are encoded in three bytes. The mapping table uses four bytes per feature. The image name table size is negligible. Hence, the total memory requirements of a quad tree are on average 86.4 KB per image, which is about twice the size of the JPEG file size. Our current database can hold an index with slightly more than 776,000 images; currently we only use 10% of that amount. When more image quad trees need to be stored online, either the machine needs to be equipped with more memory, or we will need to be more careful with encoding the features. *E.g.* we are uncertain whether maintaining a weight in the index is useful, and the saturation and value can be reduced to four bits each. Finally, we may reconsider the contents of the Image key table: if instead we record the starting OID and the number of features per image, only eight bytes per image are required. So when necessary, we can compress a feature to five bytes, in which case the quad-tree size corresponds to the JPEG file size of our test set. With space optimizations, we increase the storage capacity to over 2,000,000 image quad trees of images with a size of 384×256.

As said earlier, the *coverage* of the image by its features is defined to be the fraction of the number of pixels described by the features divided by the image size. More formally:

$$C(d) = \frac{\sum_{n=1}^{j} w(d)_j}{W \times H}$$

Here, $C(d)$ represents the coverage at a certain 'depth' d in the image, j the total number of features at level d, w_j the weight of feature j, W the image width and H the image height. The *depth* represents the size of the rectangles: a depth of zero means the top rectangle, a depth of n represents all those rectangles when the top rectangle has been recursively split n times. The total coverage C is defined to be $C = \sum_{d=1}^{D} C(d)$ with D the

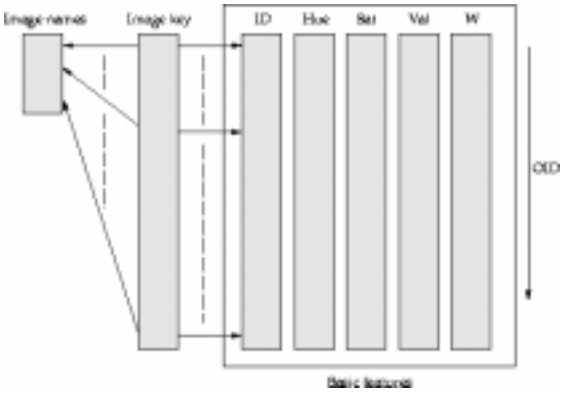

Figure 3: Monet quad-tree organization.

number of depths.

Figure 4 shows the average coverage of a set of images from the COREL image set at various depths in the images. Each one of the 49 lines of the figure represents the average coverage of 100 384x256 JPEG images from the COREL image discs (in particular: disc one). Each 100 images represents a different set of images (*e.g.* images of Africa, Plants, Rome, or the various wildlife sets). The figure shows that the coverage at level seven is between 0.66 and 0.89. More interestingly, the coverage at lower levels of the image can also be substantial. For example, at level five, there is a coverage between 0.20 and 0.50, while the number of features used is substantially less as is shown in Figure 5. So, by only analyzing the high level features of an image, one can get a good idea of the image.

The single line that starts with a coverage of 0.20 in Figure 4 is caused by the image set 'wild life eagle' in the image set. Many images in this set have a clear blue sky, and our algorithm selects the blue skies from the upper rectangle as an image feature.

With the information that is stored in our forest of quad trees we can search the images for the availability of colors, color transitions and for spatial color constraints by analyzing each of the quad trees. If, for example, we are looking for images with pyramids, a useful approach is first to select all images with blue and yellow rectangles at small depths (*i.e.* large areas) of the quad trees, possibly with the rule that only quad trees are selected which have large blue features *above* large yellow features. Once a limited number of quad trees are selected, more elaborate analysis can be executed on the reduced quad-tree set.

4. NAVIGATING THROUGH THE FOREST

The way we have organized our quad-tree tables still requires many joins over tables to find out which color information is associated to which file. If, for example, a user selects all images with a blue and a yellow component, it is likely that first the hue table as is shown in Figure 3 is used to select all features with blue and all features with yellow, before the OIDs of the features are joined with the mapping to find the unique images.

To provide quick selection of images based on color, we have implemented a color bit vector structure over the quad trees. When Monet is populated with image quad trees, we also calculate a color histogram based on 256 bins. Each bin in the histogram is allocated a bit vector with the size of the number of images. Bit number OID in the bit vector for a color bin is set, when the image corresponding to the OID has one or more pixels in that color. This is done for all bins.

To execute the earlier pyramid query, we now first select all images that contain blue and yellow. When only single color bands are used, we only need to logically-AND the bit vectors of the corresponding color bins. The resulting bit vector holds all encoded OIDs of all images satisfying the colors. When larger color bands are used (*e.g.* all blue color bins), multiple color bins are first logically OR-ed.

We also maintain a number of the color bit vectors at the various quad-tree depths of the image to enable sub quad tree searches. If, for example a user likes to select all images with large blue areas, the color bit

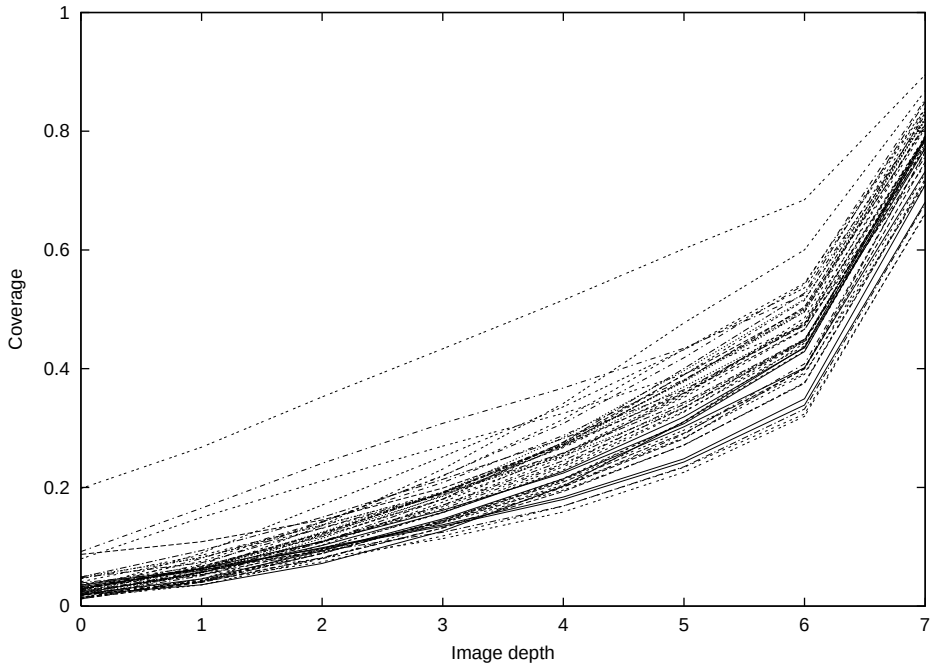

Figure 4: Average coverage per quad-tree level per image set from the COREL images

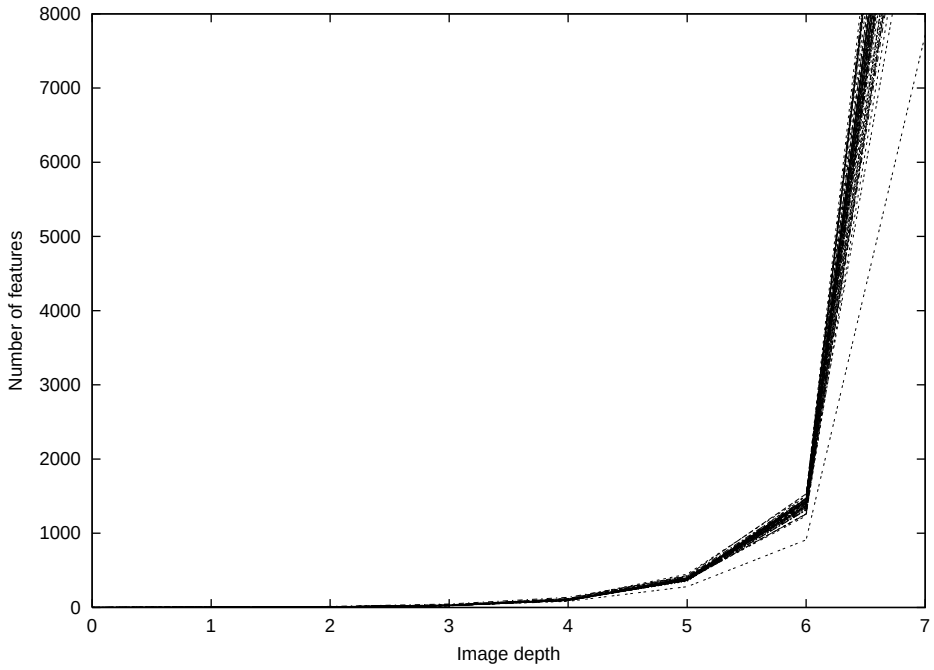

Figure 5: Average number of features per depth per image set from the COREL images

vectors for the blue bands and for small depths (*i.e.* large regions) are logically OR-ed to find such regions. The advantage of this approach is that the user can now quickly retrieve information on image regions and their colors. Secondly, it turns out that there are many images in the COREL image set where *all* 256 colors are available in the image albeit the region size of some of the colors in some images is quite low: using a multi-layer color bit vector makes sure that a color that only has a few occurrences in an image are only visible at the deepest color bit vector.

The color bit vector structure does not require much memory. For each color bit vector we need 256 times the number of images worth of bits. So, to record 2,000,000 images in color bit vectors at eight depths and the for the overall color histogram, we need less than 550 MB worth of bit vectors. Note that the color bit vector structure alone is not enough to search for spatial constraints (*e.g.* blue areas above yellow areas): it only speeds up access to those image quad trees holding the searched for colors.

5. THE IMAGESPOTTER

Our Java based user front-end allows users to interact with the image database. It enables users to formulate queries against our database, it presents the results of the queries to the users and it allows the users to re-formulate its query when they are not happy with the results of the query. In a way we see our front-end as a navigator that helps users to find their way through the color bit vector and forest of quad trees. We expect that users need help in formulating what they are looking for (if they already know what they are looking for in the first place). Our navigator has been constructed such that we make the index information explicit and show why images have been selected by database during a query.

The most important feature of our navigator is the way *spots* are selected. Figure 6 shows a screen dump of our front end with on the left side a blown up representation of the spot that is selected from the image in the upper-left corner of the right-hand side of the screen dump. A spot is a distinct portion of an image; for example, the blue sky and the brownish mountain in Figure 6. This spot can be used to articulate a query to our database.

Once one or more spots have been identified, a user can query the database for similar spots. The way we attack this problem is first find all images that have at least the same color configuration through the color bit vector structure. When candidate images in the form of quad trees are selected, further selections are performed by means of spatial constraints: *e.g.* when a user is looking for mountain rigs, it is likely that the user formulates a query with spatial information for the color distribution (*i.e.* blue must be above yellow).

The image spotter has a number of control knobs for making the queries more precise or to loosen up the query. For example, weight information may be attached to the query, deltas to the colors bands may be reduced or increased before issuing the query to the database, or color transitions may be of importance. These control knobs will be and already are integrated in the ImageSpotter.

6. SUMMARY

We have presented the foundation of our image retrieval work. Our main-memory database stores color and spatial information of, at most, 2,000,000 images, and we allow users to intelligently browse the index using the forest of quad trees and a color bit vector. Browsing and selecting images from the index can be done through our ImageSpotter – a navigator that is used to select interesting spots from images, to present the exact color information from spots, to formulate and adjust queries and to present query results.

The fundamental difference between our approach and existing approaches is the sheer volume of our image set. With such large image databases, only simple selection and indexing techniques can be used or else the storage of the index or the computational costs of performing queries are prohibitively large. This implies that CPU intensive image feature extractors are unusable for the vast majority of the images.

With this work we have finished the ground work that enables more extensive experiments and inclusion of different kinds of features. We believe that our approach provides for the scalability required for an extensive image database through which users can find their way.

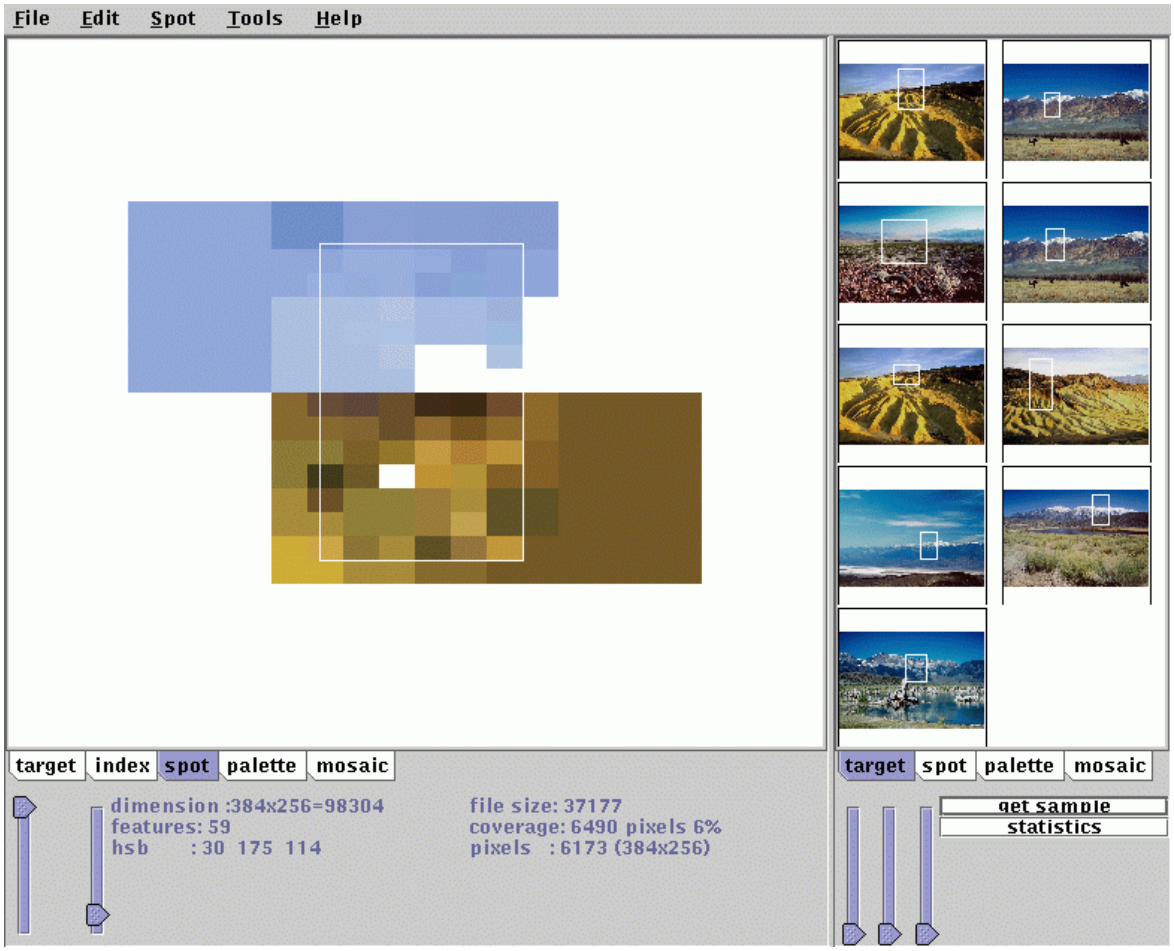

Figure 6: ImageSpotter screen dump

References

1. P. A. Boncz and M. L. Kersten. MIL Primitives for Querying a Fragmented World. *VLDB Journal*, **8**(2):101-19, 1999.
2. C. Carson, M. Thomas, S. Belongie, J. M. Hellerstein, and J. Malik. Blobworld: A System for Region-Based Image Indexing and Retrieval. In *Third International Conference Visual Information and Information Systems*, pages 509-516, 1999.
3. C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and Effective Querying by Image Content. *Intelligent Information Systems*, **3**:231–62, 1994.
4. Hongjun Lu, Beng-Chin Ooi, and Kian-Lee Tan. Efficient Image Retrieval By Color Contents. *Applications of Databases* (Vadstena, Sweden), number 819 in Lecture Notes in Computer Science, pages 95–108. Springer-Verlag, June 1994.
5. Yong Rui, T.S. Huang, and S.-F. Chang. Image Retrieval: Past, Present and Future. *Symposium on Multimedia Information Processing* (Taiwan), 1997.
6. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison Wesley, 1990.
7. Michael Swain and Dana Ballard. Color indexing. *Internation Journal of Computer Vision*, **7**(1), 1991.
8. M. Yamamuro, K. Kushima, H. Kimoto, H. Akama, S. Konya, J. Nakagawa, K. Mii, N. Taniguchi, and K. Curtis. Exsight-multimedia information retrieval system. *20th Annual Pacific Telecommunications Conference* (Honolulu, HI, USA), pages 734-9, 1998.