An Extendible Hash for Multi-Precision Similarity Querying of Image Databases*

Shu Lin†¶ M. Tamer Özsu‡¶ Vincent Oria§¶ Raymond Ng||

Abstract

We propose multi-precision similarity matching where the image is divided into a number of sub-blocks, each with its associated color histogram. We present experimental results showing that the spatial distribution information recorded by multi-precision color histograms helps to make similarity matching more precise. We also show that sub-image queries are much better supported with multi-precision color histograms. To minimize the overhead, we employ a filtering scheme based on the 3-dimensional average color vectors. We provide a formal result proving that filtering with multi-precision color histograms is complete. Finally, we develop a novel extendible hashing structure for indexing the average color vectors. We give experimental results showing that the proposed structure significantly outperforms the SR-tree.

1 Introduction

Content-based access to and querying of image repositories is of significant interest to the database and image processing communities. Most studies have focused on image retrieval based on such perceptual properties as color, texture and shape. The problem, in this context, is to find the images that are "similar" to a query image with respect to one of these properties; hence the term *similarity matching*.

*Research supported by Institute for Robotics and Intelligent Systems (IRIS), a Network of Centre of Excellence funded by the Government of Canada and by a Strategic Grant from the Natural Sciences and Engineering Research Council (NSERC) of Canada.

†IBM Toronto Laboratories; shulin@ca.ibm.com

‡University of Waterloo; tozsu@db.uwaterloo.ca

§New Jersey Institute of Technology; oria@cis.njit.edu

¶Work performed while the authors were at the University of Alberta.

||University of British Columbia; rng@cs.ubc.ca

Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001

In this paper, our focus is on color similarity matching. The research on color similarity matching has primarily been restricted to color histograms for the entire images. While a histogram is capable of summarizing the color distribution of an image, it is incapable of capturing the spatial distribution of the color values. As a simple example, consider two images, one of which has one color in its top half and a second color in its lower half, while the second reverses the placement of the two colors. These two images have exactly the same color histogram, and cannot be distinguished from one another. Furthermore, *sub-image* queries, where the user only specifies a (small) part of the image he/she remembers or cares about, cannot be supported. To address these two weaknesses, we study in this paper *multi-precision* similarity matching. The idea is to divide the image into a number of sub-blocks, each of which has an associated color histogram. These are used for more precise color similarity comparisons.

A second issue that we study in this paper is the development of a hashing structure to facilitate searching over high-dimensional spaces, which is a characteristic of similarity matching. Even when only one feature, e.g., color, is considered, high-dimensionality is evident. To facilitate color similarity matching, in their seminal work, Swain and Ballard propose histogram intersection for color histogram matching [18]. Stricker and Orengo propose cumulative color histogram indexing [17]. If color histograms are to be indexed directly, the indexing structure would be very high-dimensional (e.g., 64- or 256-dimensions). To avoid the "dimensionality curse", the standard solution is to conduct dimensionality reduction. One way is to apply principal component analysis or singular value decomposition (e.g., [19, 11]). Alternatively, a 3-dimensional *average* color vector can be used as a filter [2]. The idea is to filter the database using average color comparisons first, and then to apply full color histogram comparisons to the significantly smaller set of images returned by the filtering step. We propose an extendible hashing structure for searching in high-dimensional spaces.

We demonstrate how the extendible hashing structure can be used to improve the performance of multi-precision similarity matching. Consequently, this paper makes the following contributions. We present experimental results showing that the spatial distribution information recorded by multi-precision color histograms helps to make similarity matching more precise. We also show that sub-image queries are much better supported with multi-precision color histograms. To improve the efficiency of this added functionality, we employ a filtering scheme based on the 3-dimensional average color vectors, similar to [2], but applied to multi-precision color histograms. We provide a formal result proving that filtering with multi-precision color histograms is complete. We further develop an extendible hashing structure to improve the effectiveness and efficiency of filtering. We give experimental results showing that the proposed structure significantly outperforms the popular SR-tree [5]. Multi-precision filtering, and the hash structure supporting it, have been incorporated into an image DBMS. This provides a demonstration of the feasibility of the approach.

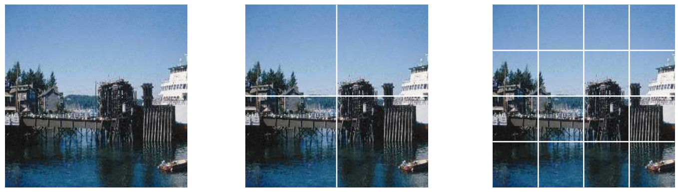
2 Background

The fundamental elements of histogram-based image retrieval include the selection of the color space, the color space quantization, and the histogram distance metric. There is no general agreement as to the most suitable color space for color histogram-based image retrieval. This is a result of the fact that color perception is highly subjective [21]. Therefore, a variety of color spaces are used in practice such as RGB, HIS, or L*u*v*. Multi-precision filtering proposed in this paper is insensitive to the choice of color space, but in our current hash design, we use the RGB model. The technique can be extended to other models, but additional work is required. We indicate what is needed in future sections.

As noted above, there are different ways of dealing with the dimensionality curse. We adopt the method proposed in [2, 3]. In this method, a 3-dimensional compact representation of color histograms—average colors— is proposed as a cheaper way to simulate full n-dimensional histogram comparisons. The idea is to filter the database using average color comparisons first, and then apply full color histogram comparisons to the significantly smaller set of images which are retrieved at the filtering step.

A color histogram is a discrete function $h(c_k) = n_k$, where c_k is the k-th color value and n_k is the number of pixels in the image with that color. In order to compare color histograms of images with different sizes, color histograms are often normalized as $H(c_k) = \frac{n_k}{n}$ where n is the total number of pixels in the image. Given that each bin of a color histogram represents a three-dimensional color value, the average color of a color histogram is defined to be the weighted average color corresponding to the normalized color histogram. Specifically, let $C = [c_1 c_2 \dots c_n]$ be a $3 \times n$ matrix whose i-th column is the color $c_i = [\alpha_i \beta_i \gamma_i]^T$, where α, β, and γ represent the magnitude along the three color dimensions. Given two normalized n-dimensional color histograms, X and Y, the 3×1 average color vector for each is $X_{ave} = CX$, $Y_{ave} = CY$. The squared average color distance is defined by $d^2_{ave} = (X_{ave} - Y_{ave})^T (X_{ave} - Y_{ave}) = (X - Y)^T C^T C (X - Y)$. If we let $W = C^T C$, the above expression can be rewritten as $d^2_{ave} = Z^T W Z = \tilde{Z}^T \tilde{W} \tilde{Z}$ where $Z = (X - Y)$, $\tilde{Z}$ is the first $n-1$ elements of Z, and $\tilde{W}$ is defined in terms of W.

While the average color comparisons are not as accurate as one between full n-dimensional histograms, they are much faster. Moreover, the images retrieved by average color comparisons are guaranteed to include all images that should be retrieved by color histogram comparisons, as proven in [3]. Accordingly, for any range query of the form $d_{hist} \leq \epsilon$ ($d^2_{hist}(X, Y) = Z^T A Z$), it is possible to use $d_{ave} \leq \epsilon / \sqrt{\lambda_1}$ to retrieve images quickly and without misses. Here λ_1 is the minimum eigenvalue of the generalized eigenvalue problem $\tilde{A}\tilde{Z} = \lambda \tilde{W} \tilde{Z}$ in which $A = [a_{ij}]$ is a *similarity matrix* and $\tilde{A}$ is defined in terms of A. d_{hist} is then applied only to the filtered set of images.



(a) 1st Level (b) 2nd Level (c) 3rd Level

Figure 1: The three-level multi-scale representation

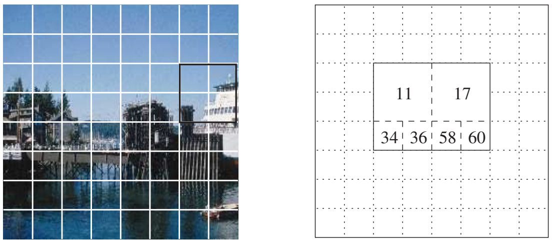
3 Multi-Precision Querying

3.1 Multi-Precision Similarity Filtering

Besides extracting color histograms from entire images, an image can also be segmented into several blocks, each of which has an associated color histogram. Figure 1 shows a 3-level multi-scale representation in which the entire image is divided into four blocks, and each block is recursively divided into four. A color histogram is computed for each of the blocks at each level. These color histograms together form *multi-scale color histograms* of an image. At the end of this section, we show how this simple, fixed grid segmentation can be generalized.

With multi-scale color histograms, image similarity queries based on color histograms can be done at several levels of precision. In a three-level decomposition, which is what we employ in the experiments reported in this paper, two images can be similar at three precision levels: At the first level, the color histogram of the entire image is compared using 1 color histogram comparison; at the second level, the color histograms corresponding to the 4 blocks ($\frac{1}{4}$ of the entire image) are compared requiring 4 color histogram comparisons; at the third level, the color histograms corresponding to the 16 blocks ($\frac{1}{16}$ of the entire images) are compared resulting in 16 color histogram comparisons.

At the first level, the image distance D_1 is defined as the distance between color histograms of entire images. At a higher level l, D_l is defined as the average of the color histogram distances computed for all the blocks. At the third level, for instance, $D_3 = \frac{1}{16} \sum_{i=1}^{16} d_i$, where d_i is the color histogram distance computed for the i-th image block. This image distance metric takes the image color composition



(a) (b)

Figure 2: Sub-image querying: (a) Example, (b) 8x8 grid

into account, and also captures the spatial distribution of color in an image. Thus, the distance metrics at the higher precision levels provide better discrimination power.

If images are similar at a higher level, they must be similar at the lower levels too. For this reason, the color histogram distance must be formulated in such a way that the image distance at the lower level is no larger than the distance at the higher level, i.e., $D_1 \leq D_2 \leq D_3$.

Theorem 1 *In a n-level multi-scale color histogram, if D_l denotes the distance between two images at level l, then $D_{l-1} \leq D_l \leq D_{l+1}$ ($2 \leq l \leq n$).*

Obviously, the distance metrics defined at higher precision levels are computationally more expensive than those defined at lower levels. Fortunately, since $D_1 \leq D_2 \leq D_3$, efficient multi-scale search strategies with the use of lower level distances as filters can be explored. As suggested in [9], filtering schemes are depth-first: Matching is first done at level 1 and then at level 2 and so on, without skipping any intermediate levels. Thus, when the second precision level query of the form $D_2 \leq \epsilon$ is processed, the following filter is applied: $d_{ave} \leq \epsilon/\sqrt{\lambda_1}$ (d_{ave} and λ_1 are described above), $D_1 \leq \epsilon$, $D_2 \leq \epsilon$. When third precision level query is processed, $D_3 \leq \epsilon$ is added to the filter.

3.2 Sub-image Similarity Queries

Multi-scale color histograms can be used to support sub-image queries. The third level of decomposition superimposes a 4-by-4 grid on an image (see Figure 1, 3rd level). Users can specify the part of image in which they are interested by choosing any of the grid cells. For example, the user may want to find all the images that have a white building along the right-hand border like the one shown in Figure 2(a); the remaining part of the image is "don't care" for the user[1]. Without multi-precision color histograms, this sub-image query can only be answered based on the histograms of the entire images. This strategy gives poor results, particularly when the size of the sub-image query is small relative to the entire image.

With multi-precision color histograms, however, a sub-image query can be readily answered. The appropriate grid cells can be chosen, as long as the resulting shape is a rectangle. The selected rectangle is composed of predefined image quadrants, each of which is uniquely identified by a number as in the quadtree as shown in Figure 2(b). The color histogram of the selected portion can be computed based on the precomputed multi-scale color histograms. For this example, the color histogram can be computed from the two third-level color histograms and the four fourth-level color histograms. Assuming that the color histograms are normalized, the computation formula is:

$$H = \frac{1}{3}H_{11} + \frac{1}{3}H_{17} + \frac{1}{12}(H_{34} + H_{36} + H_{58} + H_{60}) \quad (1)$$

where H_{11}, H_{17}, ..., H_{60} are color histograms of the corresponding numbered parts.

So far, we have assumed a fixed grid superimposed on the image. The results can be generalized. The fixed grid can have *overlapping* blocks [16]. Equation (1) can certainly be generalized to a grid of overlapping blocks. If the query sub-image is not aligned with the fixed grid (e.g., the white building given in Figure 2(a) where the query sub-image is strictly contained in but smaller than the four smallest grid cells along the right-hand border) the *padding* technique of [9] would be applicable to solve the problem. Based on a linear constraint optimization framework, this technique can pad extra pixels to the smaller query sub-image histogram in such a way that the estimated color histogram distance is minimized. The results given in [9] show that sub-image querying with padding still provides much better quality than if the sub-image query is answered based on histograms of the entire images.

4 Hashing Algorithm

As indicated above, multi-precision filtering is a depth-first process. The first operation in this process is important as it is performed on the largest set of images. We develop a hashing structure to perform this operation efficiently.

There are a number of proposals for indexing multi-dimensional data. These can be categorized into two classes: hierarchical (tree-based) methods (e.g., R*-tree [1], SS-tree [20], and SR-tree [5]), and non-hierarchical (hashing-based) methods (e.g., grid file [12]).

In addition to the above mentioned structures, different multidimensional hash strategies have been proposed mainly as extensions of the linear hashing to avoid keeping large directory information. These include multidimensional linear dynamic hashing [14], which does not support range queries, MOLPHE (Multidimensional Order-Preserving Linear Hashing with Partial Expansions) [6] that works fairly well for uniformly distributed data, but fails for non-uniform data distribution because of the hashing function used, quantile hashing [7, 8], which addresses this problem by uniformly distributing the original data during a division, and z-ordering [4] that proposes a better ordering method using the z-order.

4.1 Extendible Hash Structure

The traditional extendible hashing is capable of expanding and contracting hash address space as needed. However, because it can only handle one dimensional data, it is not suitable for multi-dimensional similarity search. We design a new n-dimensional hash structure for this purpose.

[1]To assist in better exposing the idea, we show 4 levels rather than 3, which is what we use in our experiments.

In the following, we only concentrate on its application to 3-dimensional color space. To focus on this particular instantiation that, the hash structure will be referred to as *three-dimensional extendible hash* (3DEH).

This structure is suitable for any color model defined in an Euclidean coordinate system. In this paper, we use the RGB model, but the structure can be extended to work with the L*u*v* model; that will be the topic of follow-up work.

Extendible hashing literature conflicts in how the directory growth is accommodated. Some add a bit at the least significant bit position and others add the bit at the most significant bit position. We add the bit at the most significant position (as described, for example, in [15]). One advantage of our approach is that new space is added at the end of the directory table, eliminating the need to reorganize the table after expansion (i.e., none of the keys in the original 2^k key space need to be re-assigned to the newly created directory entries). Given an address x in the original 2^k key space, the corresponding overflow bucket can be computed as $x + c$ where c is the beginning address of the new space. In the case when the new key space is contiguous to the original one, $c = 2^k$ (since the key space doubles at every expansion). Second, in regular hashing, the objective is equality search. Therefore two keys that hash to the same bucket are considered conflicts that result from a non-perfect hash function. In our case, the objective is similarity matches, not equality. Thus, two keys that hash to the same bucket are considered similar, not an anomaly. A second objective of adding a most significant bit is to facilitate putting similar images into the same bucket. Finally, we introduce a *mask track* to keep track of the split direction in the buckets, since they can split along any of the three dimensions (colors). This is important, because the splitting dimension can change from bucket to bucket. The mask track controls the growth of the directory by mapping three dimensions into one. Thus, the 3DEH directory grows slower than the grid file.

4.2 Initial Setup and Simple Insertion

The hash directory of the three-dimensional extendible hashing has three *initial depths* (d_1, d_2, d_3), one for each of the R, G, B color components; it also has a *growth depth* d_g, which is 0 at the beginning and increases with the address space. The number of bits of a hash address is $(d_1 + d_2 + d_3 + d_g)$, so the hash directory has $2^{(d_1+d_2+d_3+d_g)}$ entries. The bucket in 3DEH has three local depths (p_1, p_2, p_3), which means that all records in the bucket have common p_1, p_2, p_3 leading bits of the R, G, B values, respectively. At the beginning, the local depths of buckets are the same as the initial depths of the directory. The directory entry either points to a bucket or holds a null value if no data records are hashed to this entry.

The initial hash directory has $2^{(d_1+d_2+d_3)}$ entries since d_g is zero at the beginning. The $(d_1 + d_2 + d_3)$-bit hash address is computed by taking the leading d_1, d_2, d_3 bits from the R, G, B values, respectively, and then concatenating them. We call this $(d_1+d_2+d_3)$-bit address *initial hash address*. Figure 3 shows an initial hash directory whose initial depths are (1, 1, 1). Insertion of a value into bucket A (hash address 010), for example, is a simple insertion that proceeds as in regular hash structures.

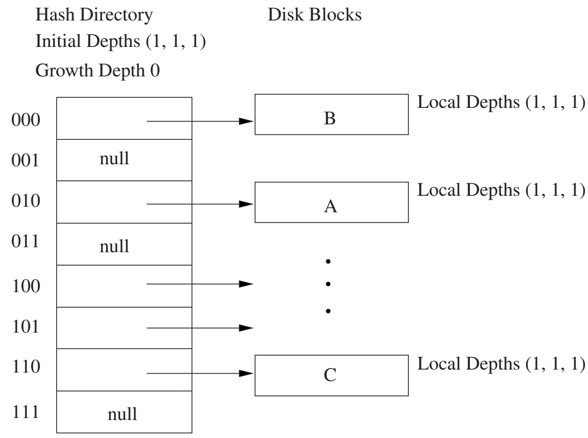


Figure 3: Initial hash directory of the three-dimensional extendible hashing

The initial depths of the hash directory can be chosen according to the applications. If green colors are dominant and red colors barely appear in an application, for example, we may extract 1 bit from R, 3 bits from G, 2 bits from B, resulting in an initial hash directory of size $2^{(1+3+2)} = 64$.

4.3 Insertion with Split and Directory Doubling

When a bucket overflows in 3DEH, like traditional extendible hashing, the hash address space increases and the bucket splits. Unlike traditional extendible hashing, in which a bucket can be split along only one dimension, a bucket in 3DEH can be split along any of the R, G, B dimensions. We split the bucket along the dimension with the highest variance so that the records can distribute as evenly as possible in the two resulting buckets. Suppose, in Figure 3, that bucket A, whose hash address is 010, overflows and R dimension has the highest variance — then the bucket is split along R dimension by putting all the records whose R values' second leading bit is 1 to a new bucket D. Since there is no more space in the hash directory to accommodate the new bucket, the address space is doubled, and now the hash address becomes a 4-bit binary number.

Since the bucket can be split along any one of the three dimensions, we need to record the dimension along which it is split. A data structure named *mask track* is maintained to keep track of the splitting history of the bucket. For example, we record the fact that only the bucket with the initial hash address 010 has split along R dimension in the mask track shown in Figure 4. Every entry in the mask track is zero, except that the 010 entry is 100. Note that space used for this purpose can be optimized by using fewer bits to record this information (e.g., 2 bits for 3 dimensions).

Now, further suppose that bucket D in Figure 4 overflows and the highest variant dimension within this bucket is B dimension—so that bucket D is split along B dimension by moving all the records whose B values start with

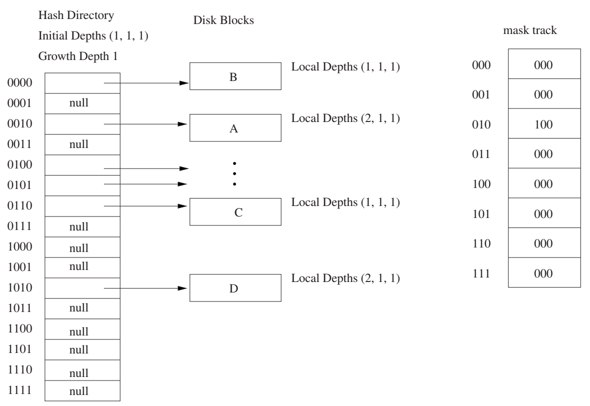


Figure 4: First expansion of the hash directory

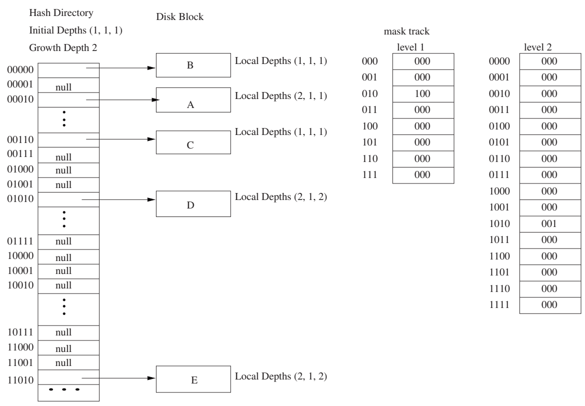


Figure 5: Second expansion of the hash directory

01 to a new bucket E. The hash address space is doubled again. The bucket E is pointed by the entry 11010. The 5-bit hash address of color (01111100, 10101000, 00011001) is 01010, which is computed by putting the second leading bit (0) of B value in front of its previous 4-bit hash address 1010. The second level of mask track is created and the value 001 is stored in entry 1010, which means the bucket whose hash address is 1010 is split along B dimension. Figure 5 illustrates this situation.

4.4 Insertion with Split but without Directory Doubling

Suppose now that the bucket C in Figure 5 overflows and is split along G dimension. A new bucket F is allocated, and all the records whose G values start with 11 are moved from bucket C to F. This time we do not expand the address space. Instead, we simply let the 01110 entry, which previously held a null value, point to bucket F. The contents of entry 110 of the first level mask track are changed from 000 to 010 to record the splitting. If bucket C overflows again, and is split along R dimension this time, a new bucket G is allocated and pointed by the directory entry 10110. The 0110 entry of the second level mask track is changed from 000 to 100 (see Figure 6).

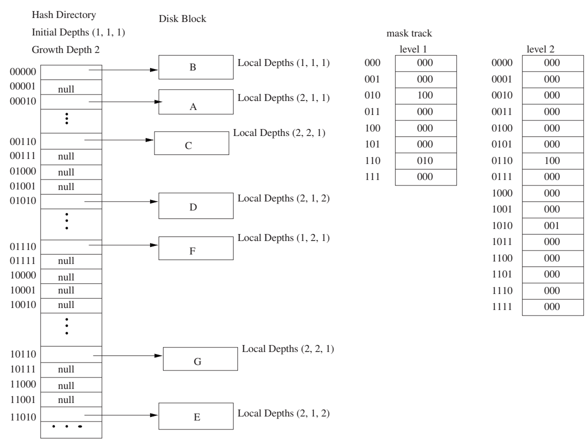


Figure 6: Third expansion of the hash directory

4.5 General Case for Insertion

In general, when a bucket overflows and needs to be split, its local depths are compared to the directory depths. If $(p_1 + p_2 + p_3) = (d_1 + d_2 + d_3 + d_g)$, then there is no address space to hold the new bucket, the directory must be doubled and d_g is increased by 1. Otherwise, the address space remains unchanged. The original bucket has the same hash address k as it had before; the hash address of the new bucket is $2^{(p_1+p_2+p_3)} + k$. The bucket is split along the highest variant dimension. Suppose the R dimension has the the highest variance. The bucket is split by moving all records whose R values' $(p_1 + 1)$-th leading bit is 1 to the new bucket. The local depths of the two split buckets are now $(p_1 + 1, p_2, p_3)$. The k-th entry of the $(p_1 + p_2 + p_3 - d_1 - d_2 - d_3 + 1)$-th level mask track is updated to indicate the splitting.

With the above hash structure, a given color can be mapped to its hash address as follows:

1. Extract d_1, d_2, d_3 leading bits from R, G, B values, respectively. Concatenate these bits to form an initial address.
2. Refer to the entry corresponding to the initial address in the first level of the mask track. If the bucket has not been split, go to Step 4; otherwise compute the new address as described above.
3. Keep looking up the next level of mask track with the new address until the mask indicates that the bucket represented by this address has not been split.
4. Put appropriate number of zeros at the front of the obtained address to make it a $(d_1 + d_2 + d_3 + d_g)$-bit hash address.

4.6 Deletion

Deletion in 3DEH is similar to traditional extensible hashing; buddy buckets are merged to collapse the directory. The mask track entries are re-set to 000 when this occurs. To avoid merged buckets from being split again soon, a merge-threshold [12] is used. The merge-threshold is the percent-occupancy, which the resulting bucket should not exceed when two buckets are merged. If the percent-occupancy of resulting bucket is above the threshold, merging shouldn't occur.

4.7 Analysis

Hash support for multi-dimensional querying introduces a new data structure: the mask track. During query processing, it is necessary to navigate through the multiple levels of the mask track, similar to tree navigation. The additional cost of maintaining and searching the mask track needs to be considered.

The size of the mask track is bound by 2^k, which is the size of the directory. Furthermore, each directory entry is only 2 bits. Thus, for example, 1,000,000 directory entries (which corresponds to much larger database due to bucketing) require only 2Mbits. With the mask track small enough to be kept in memory even for very large databases, it is guaranteed that at most 2 I/O's are needed for equality searches.

4.8 Range Search Algorithm

Similarity matching, which is more common in image DBMSs than exact matching, may require range searching for which traditional hash structures are not suitable. Given an example color value (a, b, c) and a distance threshold ϵ, a range query finds all the color values that fall into the search region, which is a sphere whose center is (a, b, c) and radius is ϵ.

In 3DEH, each bucket pointed at by the initial hash directory represents a region corresponding to one of the partitions. During the growth of the hash directory, the buckets may have been split. The regions represented by the split buckets are further partitions of the original bucket region. The range query algorithm will first locate all the buckets whose regions overlap the search region, and then examine the contents of the buckets. Since the shape of the bucket region is a rectangular solid, and it is easier to decide whether two rectangular solids overlap, a minimum bounding cube of the sphere, instead of the sphere itself, is used during the procedure of locating the buckets.

The first step of this algorithm is to compute the set of initial hash addresses of the partitions which overlap the cube. For example, given a color value (R, G, B) represented by binary numbers (01111100, 10101000, 00011001) and a distance threshold 12 (1100 in binary), the minimum bounding cube of the search region is

$$\big(\underbrace{[01110000, 10001000]}_{\text{R}}, \underbrace{[10011100, 10110100]}_{\text{G}}, \underbrace{[00001101, 00100101]}_{\text{B}}\big).$$

Take R, as an example. Since its color value is 01111100, its range is calculated as $(01111100) \pm (1100)$ which gives the interval [01110000, 10001000]. G and B ranges are calculated similarly. So, for the hash directory whose initial depths are (1, 1, 1) as shown in Figure 3, the set of computed initial hash addresses is $\{010, 110\}$, obtained as follows. Since the initial depths are (1,1,1), the R range of [01110000,10001000] is reduced to [0,1], essentially just taking the first bit of the two ends of the range.

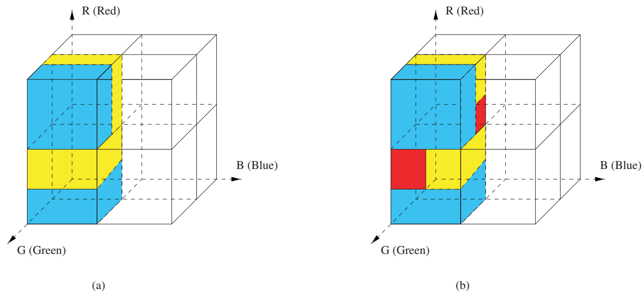


Figure 7: Further partitioning of initial partitions

Similarly, the G range is reduced to [1,1] and the B range is reduced to [0,0]. The "cross-product" of the three reduced ranges gives the set $\{010, 110\}$.

The mask track is looked up level-by-level to find out how these initial partitions are further partitioned. For example, the first level mask track in Figure 6 indicates that the 010 bucket has been split along the R dimension, resulting in two split buckets 0010 and 1010. Only the 1010 bucket still overlaps the minimum bounding cube; the 0010 bucket is outside the cube. This situation is shown in Figure 7(a). The lower cube is split into two halves. Only the upper half (lighter shading) still overlaps the minimum bounding cube.

When we look up entry 110 in the first level mask track, we know that this bucket has been split along G dimension. Only bucket 0110 still overlaps the cube. The upper cube is also cut in two halves, and only the lighter shaded half overlaps the minimum bounding cube. So the set of addresses which are still in consideration becomes $\{1010, 0110\}$, where the 010 is replaced by 1010 and 110 is replaced by 0110 (again note the addition of bits at the most significant position). The addresses 0010 and 1110 are discarded, because the regions represented by them are outside the bounding cube.

The second level mask track tells us that 1010 bucket has been split along B dimension, and 0110 bucket along R dimension. The regions represented by the addresses 11010 and 10110 are outside the bounding cube, so now the address set becomes $\{01010, 00110\}$. The lightest shaded rectangular solids (Figure 7(b)) are divided into two halves, and the darkest parts of these still overlap the minimum bounding cube.

In general, when a bucket overlapping the bounding cube is split, there are two possibilities: either only one split bucket overlaps the cube, or both of them do. Only the buckets overlapping the cube need to be further checked.

The algorithm continues looking up the next level mask track until all the addresses in the set are final; that is, the corresponding buckets do not split. The address set $\{01010, 00110\}$ is final, for instance. Then a further check is performed on the relative position between the bucket and the sphere. There are three possibilities: If the bucket is contained in the sphere, all the color values in this bucket are returned as query results; if the bucket intersects with the sphere, the color values in this bucket need to be examined one-by-one, and those qualified color values are returned; if the bucket is outside the sphere, it is discarded.

5 Experiment Results and Discussion

5.1 Experiment Setup

As stated earlier, we use RGB color model in the experiments. All possible 256^3 color values in RGB color model are grouped into 64 values, i.e., each dimension is quantized into 4 equal length intervals, which results 64-bin color histograms. Average colors are then computed from color histograms [3]. The number of entries in the initial hash directory is set to 64, requiring an initial hash address has 6 bits, with 2 bits from each of the R, G, B.

We take 6601 images from a published photo collection CD ROM. These images fall into six categories, including 919 animal, 1087 people, 1161 plant, 1510 scenic, 963 structure, and 961 transportation images. We compute a color histogram and an average color for each of the images. The average colors (three-dimensional points) constitute our experiment dataset.

Since the real images come from six different categories, we ran experiments to determine if there were significant differences, among classes, in the distribution of average colors over the 64 partitions. We determined that most of the average colors are in one partition [10]. Because of the similarity, there was no point in differentiating images into six categories, and we considered them in one large dataset.

In order to test the scalability of the hash structure, we need much larger datasets than the 6601 images. Synthetic data were generated for this purpose. To simulate the real data distribution more accurately, we also examined the distribution of the three color components which make up the average color values. The synthetic data are generated in two steps. First R, G, B values are generated separately according to the distributions that were obtained from the analysis of the 6601 real images. Then these R, G, B values are combined randomly to make up color values on the condition that the color values are distributed in the 64 color partitions in the same way as the real images.

The experiments are run on six datasets with the sizes of 5,000, 10,000, 50,000, 100,000, 500,000, 1,000,000 for the experiments. The 5,000-size dataset contains only real data. All the other datasets are combinations of 6601 data points (average color values) computed from the real images, and synthetic data. To save space, we only show the graphs for the 5,000 and 1,000,000 datasets. Dataset of 10,000 is similar to 5,000 and datasets larger than 10,000 are similar to 1,000,000 in all the experiments.

We implement the hashing algorithm in C++. The SR-tree code is provided by its authors. The disk block size is set to 4096. A hash bucket or a SR-tree leave node can hold maximum 511 data points. The fan-out of SR-tree is 88. The experiments were run on a SunUltraSparc 10 workstation under Solaris.

5.2 Query Performance

We run range queries on the hash structure and the SR-tree to compare their query performance. 500 query points are randomly generated with the same data distribution as the dataset. Queries are run for nine range thresholds: 4, 9, 13, 18, 22, 27, 31, 35, 40, 44, which are approximately 1% - 10% (in 1% increments) of the maximum distance between two color values. (The maximum distance is $\sqrt{3 * 255^2} = 441.67296$.) There is an obvious relationship between the distance threshold and the number of returned images: larger thresholds return larger sets of images at the end of the filter phase. We ran experiments with thresholds up to 44, because these return a sufficiently large set. For example, for a dataset size of 1,000,000 images, this value gives a result set of 100,000 images. It is obvious that this is already a large number of images as a result of filtering. We run 500 queries for each of the range thresholds and each of the datasets, and then compute average from the obtained 500 values.

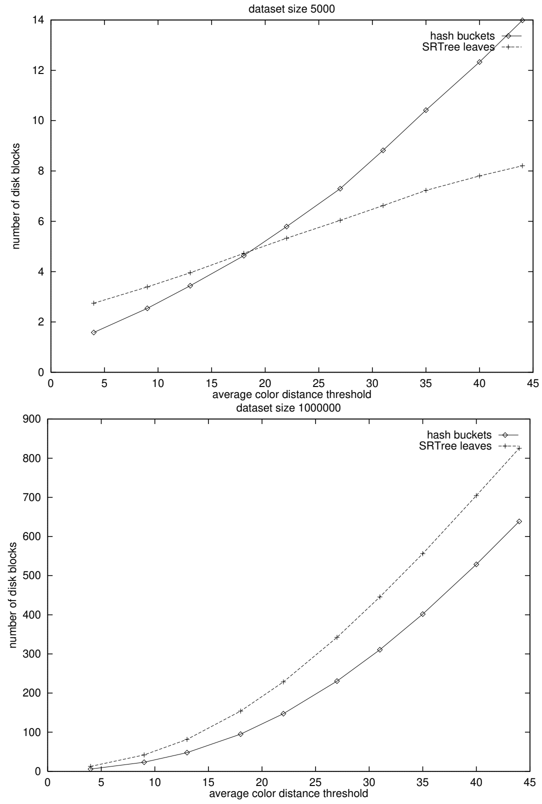


Figure 8: I/O performance of Hash and SR-tree

Figure 8 presents the experiment results about I/O performance. The hash structure outperforms the SR-tree for all the distance threshold ranges we consider, except for small datasets (5000 and 10000). The reason for the behavior in the case of small datasets is related to the way these two index structures divide the color space: SR-tree divides the space into overlapping partitions, while hash divides it into non-overlapping, but finer, partitions. This requires more reads when searching the SR-tree, because, if the search region intersects with the overlapping region of two disk blocks, then both of these disk blocks have to be read. On the other hand, the finer partitioning of the hash structure results in a larger number of buckets than the number of leaves in the SR-tree For smaller thresholds, a finer partition does not affect performance much, because the region that needs to be searched is small. In this case overlapping becomes the dominant factor. For larger thresholds, overlapping does not matter anymore, because even if the partitions were not overlapped, the search region is large enough to include them. This effect could be observed for large datasets as well, but only at very high distance thresholds.

Figure 9 presents the total time comparisons of the two structures. The difference between the total time and the I/O time depicted in Figure 8 depicts the CPU cost of managing the data structures to facilitate multi-precision and sub-image querying.

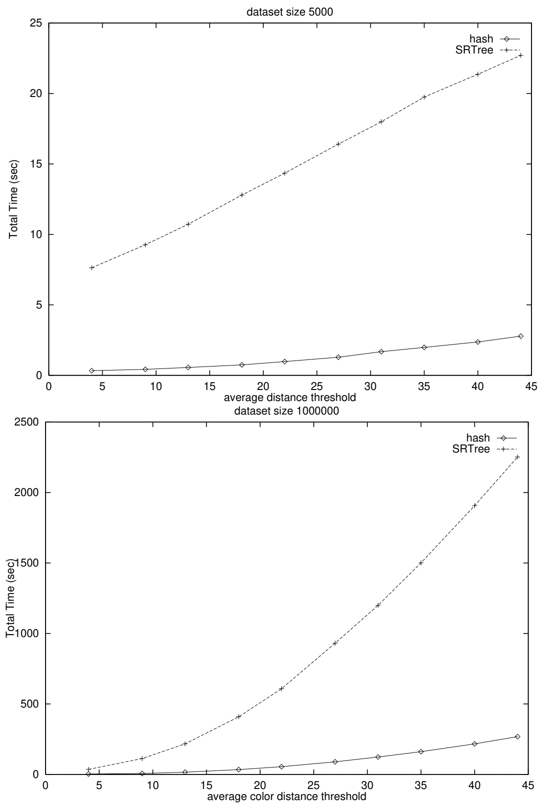


Figure 9: Total Time Comparison of Hash and SR-tree

As the above results indicate, for meaningful distance threshold ranges and for large datasets, 3DEH is an order of magnitude faster than SR-trees. The superior performance is due both to the I/O behavior described above and to the superior CPU performance of the hash. The reason for the better CPU performance is related to the search algorithms, which, of course, are based on the index structures. For 3DEH, the initial partition of the space is predetermined, so the initial hash addresses of the buckets that overlap the search region can be computed, i.e., no search operations are needed here. For the SR-tree, the root node must be searched first to decide which child node overlaps the search region, and then the child nodes are searched in turn. Searching the nodes is time-consuming, because the fan-out of the SR-tree is fairly large.

Another reason why 3DEH outperforms the SR-tree is how the two structures compare the bucket/node region to the search region. For 3DEH, the search starts from the candidate buckets obtained by computation, whcih may have been split during the growth of the hash structure. Region comparisons are needed to determine whether the split buckets overlap the search region. Only the dimension along which the bucket has been split is compared because the original bucket overlaps the search region, which results in only two possibilities: either both of the split buckets overlap the search region, or only one of them does. For the SR-tree, in order to determine whether the node region overlaps the search region, comparisons must be done for all the three dimensions since there are no hints about the relative position between the node region and the search region.

5.3 Space Utilization

We are interested in two scenarios: the continuously growing database (repeated insertions) and the steady-state database (a more balanced number of insertions and deletions).

Figure 10 presents the experimental results for the growing database case. We measure average bucket occupancy every 2000 insertions. The average bucket occupancy shows a steady state behavior with small fluctuations of around 66 percent when the number of inserted points is sufficiently large (larger than 100,000). The directory entry utilization is poor because the data distribution in our dataset is very skewed (recall that most of the average colors fall into one partition), causing the directory size to double when the dataset size grows.

For the experiments with steady-state dataset, the hash structure is first initialized with 500,000 points, then 250,000 randomly generated points are inserted into and 250,000 random points are deleted from it. Insertions and deletions are randomly interleaved. The experiments are conducted with different merge-thresholds (the percent-occupancy which the resulting bucket should not exceed when two buckets are merged): 100%, 90%, 70% and 50%. Figure 11 shows the experimental results. The average bucket occupancy changes minimally, which indicates that the space partitioning remains optimum during the deletions and insertions.

In the experiments, we also measure the number of splits and merges happening during the insertions and deletions. Comparing the two graphs in Figure 11, we can see that setting the merge-threshold at 90% saves many merge operations without degrading space utilization too much. The results for 70% and 50% merge-threshold is the same as 90% because the insertions and deletions interleave so evenly that the bucket occupancy doesn't change much: actually 90% merge-threshold already results in no merges. It is important to note that space occupancy reduces by only 2%

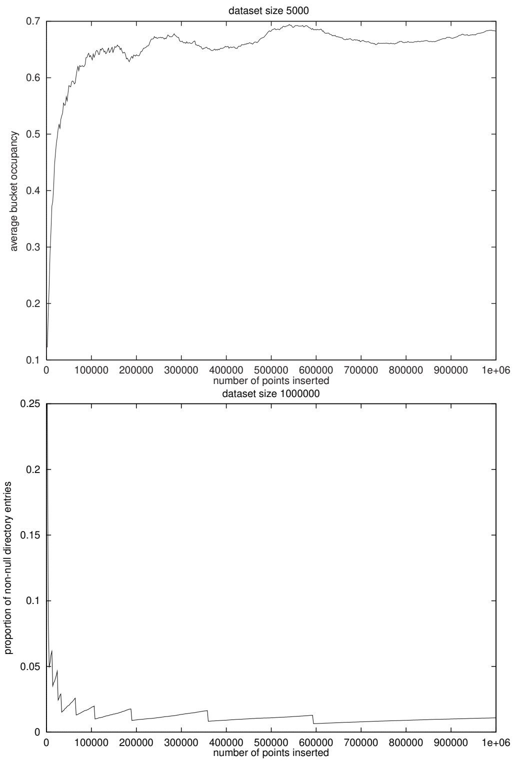


Figure 10: Space utilization of a continuously growing dataset

	Query vs right	Query vs left
Level 1	0.0110	0.0138
Level 2	0.0373	0.0767
Level 3	0.0815	0.0819

Table 1: Improvements with multi-precision filtering

after 500,000 insertion and deletion operations.

5.4 Quality and Overhead

In Section 3.1, we showed, by way of Theorem 1, that multi-precision filtering is correct. Multi-precision filtering also improves the quality of color similarity match. In order to test this claim and to quantify the improvement, we ran an experiment. From the set of 1510 scenery images we chose one image and generated from this two images that show the left and right sides of the scene. These images, which we call "left image" and "right image", are similar to the original image at level 1, but not at higher levels. The left and right images were added to the original 1510 images in the dataset (to control the result set sizes of the match queries). We then used the original image as the query image to search the entire dataset. The results are given in Table 1, in terms of the distances between the query image and the two added images. The table shows that multi-precision histograms do what they are designed for — namely, the higher the level of precision used, the stronger is the discrimination power.

One question that arises is whether doing multi-level filtering helps in the number of color histogram similarity comparisons. When executing a third precision level query,

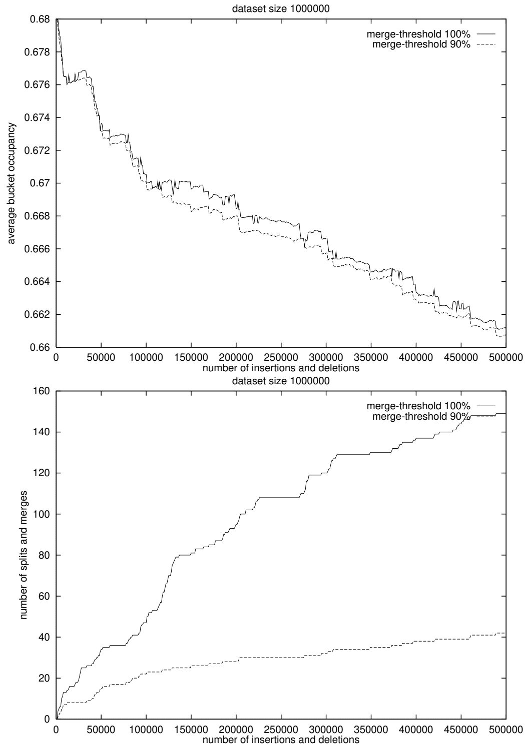


Figure 11: Space utilization of a steady-state dataset

is there an advantage to follow step-by-step filtering, or can one go directly to third level, or perform filtering at the first level and then jump directly to filtering at the third level? We have run experiments with randomly picked query images to justify our claim that step-by-step filtering is useful. We ran these images against the data set described in Section 5.1 where we compared the cost of searching using only level 3, using level 1 and level 3, and using all three levels. The results, which we omit due to space limitations, uniformly show that multi-level filtering using all three levels reduces the total number of color histogram comparisons.

A second question that arises, however, is the overhead that is incurred to obtain this precision. In order to test the overhead of multi-precision search, we conducted timing experiments using the same set-up. These results show that the overhead of multi-precision filtering is very acceptable. For large thresholds (e.g., 0.2), multi-precision matching is about 3.5 times slower than full-image matching. However, for large thresholds, the precision is relatively low. Specifically, for the threshold of 0.2, the first level returns 36 images out of 1500, whereas the third level returns only 3 images. For large datasets, users may wish to set the distance threshold significantly smaller, resulting in lower threshold. For example, for a distance threshold of 0.08, multi-precision matching takes about twice as long as full-image matching.

These results also show the power of the hash structure that supports multi-precision filtering. Without filtering, multi-precision matching at level 3 should have taken 16 times as long, since it would require the comparison of 16 color histograms instead of one. Filtering has a significant effect in reducing the overhead.

6 Conclusions and Future Work

In this paper, we propose multi-precision similarity matching to improve the quality of color similarity searchers and an n-dimensional extensible hash to support searching in high-dimensional spaces. Multi-precision color matching divides the image into a number of sub-blocks, each with its own associated color histogram that is used as the basis of finer granularity matching. The technique is supported by a particular instantiation of a novel n-dimensional extensible hash using 3 dimensions (called 3DEH). The hash structure indexes image average colors, and facilitates the filtering process. Experiments indicate that 3DEH at least an order of magnitude faster than SR-tree in total time. The space utilization of the structure is also satisfactory.

The proposed multi-precision search technique and the multi-dimensional hash structure have been incorporated into DISIMA [13], a research prototype image DBMS. Space limitations do not allow us to describe the implementation details, which will be described in subsequent papers.

There are a number of issues that we will investigate further. We will improve the hash structure in two ways. First, we will develop reinsertion algorithms to keep space division optimal. Second, we will work to improve the directory occupancy, which the experiments show as rather low.

With respect to multi-precision similarity search, there are two important areas to investigate. The first is the relaxation of the fixed grid. The second important avenue of investigation is the extension to the L*u*v* color model. The fundamental difficulty is that the visible color space in L*u*v* is conic, not rectangular as is the case in the RGB model. This requires more care in the division of the color space into cubes. It should be noted that the only part of the technique that needs to change is this part; the rest would remain unchanged. If this subdivision is done in a similar manner to what is described in this paper, then some of the space that is covered by the index structure does not have colors in it. This introduces more skew into the data distribution. However, our experiments have already taken skewed data distribution into consideration and the technique performs well in these situations. We will investigate techniques that do not introduce such skew in future work.

References

[1] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, May 1990.

[2] C. Faloutsos, W. Equitz, M. Flickner, W. Niblack, D. Petkovic, and R. Barber. Efficient and effective querying by image content. *Jour. of Intelligent Information Systems*, 3(3/4):231–262, July 1994.

[3] J. Hafner, H. S. Sawhney, W. Equitz, M. Flickner, and W. Niblack. Efficient color histogram indexing for quadratic form distance functions. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 17(7):729–736, July 1995.

[4] A. Hutflesz, H.-W. Six, and P. Widmayer. Globaly order preserving multidimensional linear hashing. In *Proc. Fourth Int. Conf. on Data Eng.*, pages 572–579, February 1988.

[5] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 369–380, May 1997.

[6] H.-P. Kriegel and B. Seeger. Multdimensional order preserving linear hashing with partial expansion. In *Proc. Int. Conf. on Database Theory*, pages 203–220, September 1986.

[7] H.-P. Kriegel and B. Seeger. Multidimensional quantile hashing is very efficient or non-uniform record distribution. In *Proc. Third Int. Conf. on Data Engineering*, pages 10–17, February 1987.

[8] H.-P. Kriegel and B. Seeger. Multidimensional quantile hashing is very efficient or non-uniform record distribution. *Informaion Science*, 48:99–117, 1989.

[9] K. Leung and R. T. Ng. Multiscale similarity matching for subimage queries of arbitrary size. In *Proc. 4th Working Conf. on Visual Database Syst.*, pages 243–264, May 1998.

[10] S. Lin. An extendible hashing structure for image similarity searches. Master's thesis, University of Alberta, Edmonton, Alberta, Canada, Fall 2000.

[11] R. T. Ng and D. Tam. Multilevel filtering for high-dimensional image data: Why and how. *IEEE Trans. on Knowledge and Data Engineering*, 11(6), November/December 1999.

[12] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. on Database Systems*, 9(1):38–71, March 1984.

[13] V. Oria, M. T. Özsu, L. Liu, X. Li, J. Z. Li, Y. Niu, and P. J. Iglinski. Modeling images for content-based queries: The DISIMA approach. In *Proc. 2nd Int. Conf. on Visual Information Systems*, pages 339–346, October 1997.

[14] M. Ouksel and P. Scheuermann. Storage mappings for multidimensional linear dynamic hashing. In *Proceedings of ACM PODS*, pages 90–105, May 1983.

[15] R. Ramakrishnan and J. Gehrke. *Database Management Systems, 2nd edition* McGraw Hill, 2000.

[16] M. Stricker and A. Dimai. Color indexing with weak spatial constraints. In *Proc. Storage and Retrieval for Image and Video Databases IV*, pages 29–40, February 1996.

[17] M. A. Stricker and M. Orengo. Similarity of color images. In *Proc. Storage and Retrieval for Image and Video Databases III*, February 1995.

[18] M. J. Swain and D. H. Ballard. Color indexing. *Int. Jour. of Computer Vision*, 7(1):11–32, 1991.

[19] M. Turk and A. Pentland. Eigenfaces for recognition. *Jour. of Cognitive Neuroscience*, 3(1):71–86, 1991.

[20] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proc. 12th Int. Conf. on Data Engineering*, pages 516–523, 1996.

[21] G. Wyszecki and W. S. Stiles. *Color science : concepts and methods, quantitative data and formulae*. Wiley and Sons, Inc., New York, NY, 2nd edition, 1982.