Adapting k-d Trees to Visual Retrieval

Rinie Egas, Nies Huijsmans, Michael Lew, Nicu Sebe

Leiden Institute of Advanced Computer Science,
Leiden University, 2333 CA Leiden, The Netherlands
{huijsman mlew nicu}@wi.leidenuniv.nl

Abstract. The most frequently occurring problem in image retrieval is find-the-similar-image, which in general is finding the nearest neighbor. From the literature, it is well known that k-d trees are efficient methods of finding nearest neighbors in high dimensional spaces. In this paper we survey the relevant k-d tree literature, and adapt the most promising solution to the problem of image retrieval by finding the best parameters for the bucket size and threshold. We also test the system on the Corel Studio photo database of 18,724 images and measure the user response times and retrieval accuracy.

1 Introduction

Searching for multimedia is one of the outstanding problems for the World Wide Web and for companies which have networked databases. For example, Philips Media uses tera-byte databases which are derived from ownership of trademarks, video, images, and audio. The real world usage of the multimedia search programs is to support and to enable the re-use and trading of assets. Content based retrieval is a rapidly growing area consisting of advances in interfaces, features, search paradigms, and indexing methods.

This paper addresses the indexing problem using the general method of k-d trees because they have been found to have moderate requirements for incremental addition of media and logarithmic access speed, which is especially important as the databases grow from tens of thousands to millions of items.

2 Feature Vector Searching

Considering a collection of feature vectors as a point set $S = p_1, p_2, \ldots, p_n$ in a d-dimensional space, the problem of retrieving images similar to a given query image is transformed to a *k-nearest neighbor searching* problem. This nearest neighbor searching problem is to find point(s) p_k in S most similar to a query point q, not necessarily in S, according to some distance measure.

2.1 Linear and Constant Time Searching

The most straightforward way to solve the nearest neighbor searching problem in high-dimensional space is to sequentially compare all points, using the given distance measure. This linear search takes $O(n)$ time. Another way is to precompute a *distance matrix* which stores all difference measures:

$$\begin{pmatrix} dist_{0,0} & dist_{0,1} & \dots & dist_{0,n-1} \\ \vdots & & \ddots & \vdots \\ dist_{n-1,0} & dist_{n-1,1} & \dots & dist_{n-1,n-1} \end{pmatrix}$$

where $dist_{i,j}$ is the distance between p_i and p_j. Note that only the lower half triangle of the matrix is necessary since the matrix is symmetric.

The main advantage of this method is that it has constant access time. A disadvantage is that when one wants to use an image from outside the image database as a query image, a new row must be added to the distance matrix. This will be an expensive operation, especially for large image collections. Furthermore when only m of the smallest distances per column are stored it will not be possible to retrieve more than the m best matches of a query image.

In summary, the linear search method requires minimal memory resources, has constant time updating of the image index, but does not scale well for large databases. The distance matrix approach requires large memory resources, has linear time updating of the image index, and is the fastest method regarding access time. In the next section, we introduce and discuss a compromise approach called *k-d* trees, which requires moderate memory resources, has moderate update times, and has logarithmic access speed.

2.2 The *k-d* Tree

The *k-d* tree ([1] [2]) is a binary tree in which each node represents a hyper-rectangle and a hyperplane orthogonal to one of the coordinate axis, which splits the hyper-rectangle into two parts. These two parts are then associated with the two child nodes. Starting with the root of the tree, which represents the entire search space, the search space is partitioned until the number of data points in the hyper-rectangle falls below some given threshold. The leaf nodes of the tree are called *buckets*, and the threshold that limits the maximum number of data points in a bucket is called the *bucket size* of the tree. Note that data points are only stored in leaf nodes, not in the internal nodes.

In the case of one-dimensional searching, the *k-d* tree is identical to a binary search tree. Each node contains some partition value by which the data points are partitioned. All data points with values less than the partition value belong to the left child, while those with a larger or equal value belong to the right child.

In k dimensions, one dimension is chosen to serve as a *discriminator*. The search space is partitioned along this discriminator dimension by some partition value. There are several possibilities of choosing both the discriminator and the partition value. We chose the discriminator as the dimension with the *largest variance*, as suggested in [3]. They found that compared with the standard maximum spread dimension, used in [1], this provided equivalent or slightly better search performance and required less CPU time to build the tree. Furthermore we chose the partition value of a node to be the *median* of the data points in

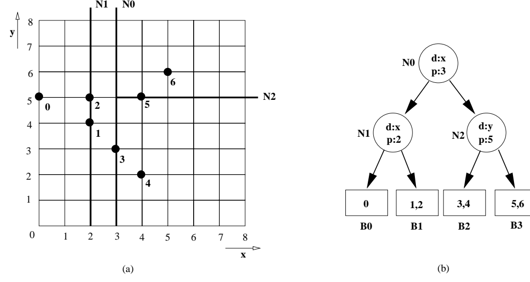

Fig. 1. (a) data points in 2 dimensions (x,y); (b) *k-d* tree; **N0**, **N1** and **N2** are the partition planes, defined by the discriminators **d** and the partition values **p**.

the hyper-rectangle represented by that node along the discriminator dimension, making the *k-d* tree balanced.

2.3 The Standard *k-d* Tree Search Algorithm

The *standard* k-d *tree search algorithm* was proposed by Arya [5]. At each leaf node visited the distance between the query point and each data point in the bucket is computed, and the nearest neighbor is updated if this is the closest point seen so far. At each internal node the subtree whose corresponding hyper-rectangle is closer to the query point is visited. Later, the farther subtree is searched if the distance between the query point and the closest point visited so far exceeds the distance between the query point and the corresponding hyper-rectangle.

To perform this test, in [1] are maintained two boundary arrays which keep track of the upper and lower boundaries of the hyper-rectangle. The distance between the query point and the closest point visited so far exceeds the distance between the query point and the corresponding hyper-rectangle if the boundaries of the hyper-rectangle overlap the ball centered at the query point with radius equal to the distance between the query point and the closest point visited so far. This test is called the *bounds-overlap-ball* test.

Arya ([6]) presents a more efficient method to perform the test, he refers to this method as the *incremental distance calculation technique*. We will use this technique in our search algorithm. A variable $CPdist_H$ is used to keep track of the distance between the query point and the hyper-rectangle H. Furthermore an array CP_H is maintained to keep track of the distance between the query point and the hyper-rectangle H along each dimension. As the *k-d* tree is traversed, $CPdist_H$ and the appropriate element of CP_H can be updated incrementally. We will use the example of Figure 1 to show how this is done.

Consider node $N2$ and its two children $B2$ and $B3$. Let H_{N2} be the hyper-rectangle corresponding to node $N2$, which is partitioned into hyper-rectangles H_{B2} and H_{B3}. Without loss of generality, assume H_{B2} is closer to query point q than H_{B3} is. Given $CPdist_{H_{N2}}$ and $CP_{H_{N2}}(i), i \in [0, d-1]$, where $CP_H(i)$ denotes the distance between the query point q and hyper-rectangle H along

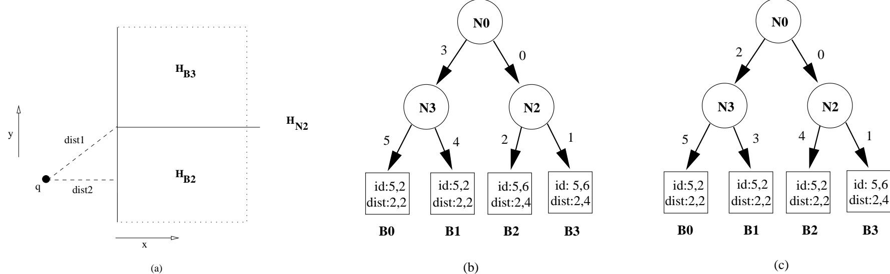

Fig. 2. (a) incremental distance calculation technique; dist1=$CPdist_{H_{B3}}$, dist2=$CPdist_{H_{B2}}$; (b) standard *k-d* tree search; (c) priority *k-d* tree search, (branch numbering shows how the tree is traversed)

dimension i, we can update these quantities to get $CPdist_H$ and $CP_H(i)$ for $i \in [0, d-1]$ and $H = H_{B2}$ or $H = H_{B3}$. Figure 2(a) shows that $CPdist_{H_{B2}} = CPdist_{H_{N2}}$and $CP_{H_{B2}}(i) = CP_{H_{N2}}(i), i \in [0, d-1]$. Also, $CP_{H_{B3}} = CP_{H_{N2}}, i \in [0, d-1], i \neq y$. The value of $CP_{H_{B3}}(y)$ is the distance between q and the plane that partitions hyper-rectangle H_{N2} and thus can be computed doing one substraction. When we use the squared distance as our distance metric, $CPdist_{H_{B3}}$ can be computed like this:

$$CPdist_{H_{B3}} = CPdist_{H_{N2}} - CP_{H_{N2}}(y) \cdot CP_{H_{N2}}(y) + CP_{H_{B3}}(y) \cdot CP_{H_{B3}}(y)$$

The above facts allow us to update CP_H and $CPdist_H$ in time independent of dimension, for each dimension. Because the efficiency of the *bounds_overlap_ball* test used by Friedman et al. ([1]) degrades with higher dimensions, the incremental distance calculation technique offers better efficiency, especially in high dimensions.

Figure 2(b) shows how the *k-d* tree is traversed when the standard search algorithm is applied to the example data points of figure 1 with query point $q = (3, 6)$. In the example the number of matches to be returned is two ($NN = 2$). The buckets show the IDs and distances of the best 2 matches after that bucket has been visited. Because the example data set contains very few data points, all buckets must be searched and no savings are made compared to linear searching.

2.4 The Priority *k-d* Tree Search Algorithm

In the search example of the previous section we saw that the standard *k-d* tree algorithm came across the nearest neighbors before the search terminated. The complexity of the search algorithm can be reduced by sacrificing this guarantee and interrupting the search before it terminates (say, when the number of visited buckets reaches a certain threshold). In this case, it is desirable to order the search so that buckets that are more likely to contain the nearest neighbor are visited first.

Arya ([5]) suggests the *priority* k-d *tree search algorithm*. This algorithm visits the buckets in increasing order of distance from the query point. This

is done by maintaining a priority queue of subtrees, where the priority of a subtree is inversely related to the distance between the query point and the hyper-rectangle corresponding to the subtree ($CPdist$). Initially the root of the *k-d* tree is inserted into the priority queue. Then the following procedure is repeatedly carried out. First the subtree with the highest priority is extracted from the queue. This subtree is descended to visit the bucket closest to the query point. In this bucket the nearest neighbor is updated if one or more data points in the buckets is the closest point seen so far. As the subtree is descended, for each node that is visited the farther subtree is inserted into the priority queue. The algorithm terminates when the priority queue is empty, or if the distance from the query point to the hyper-rectangle corresponding to the highest priority subtree is greater than the distance to the closest data point.

Figure 2(c) shows how the *k-d* tree from figure 1 is traversed when the priority search algorithm is applied to the example data points. Again, the query point is $q = (3, 6)$ and the best 2 matches must be retrieved. The priority search algorithm finds the best 2 matches after visiting 2 buckets, where the standard search algorithm needs to visit 3 buckets to find the matches.

When using the priority search algorithm White and Jain ([3]) propose the use of a threshold that limits the number of buckets that will be visited. Using such a threshold will change the *exact* nearest neighbor search to an *approximate* nearest neighbor search. Results of White and Jain ([4]) and our empirical results show that allowing a very small probability of an incorrect result can provide a significant speedup. In the previous example (Figure 2(c)) a threshold of 2 buckets would have provided a speedup of 2 and would not have effected the results.

2.5 Integration with the Karhunen-Loève transform

We saw that in each node the dimension with the largest variance is chosen as the discriminator dimension. White & Jain ([3]) suggest to transform the feature vector sets using the *Karhunen-Loève transform* (KLT) before indexing them in a *k-d* tree. Applying the KLT on the feature vectors results in a vector set whose first elements have the largest variance and whose last elements have the smallest variance. Furthermore the elements of the KLT vectors are uncorrelated.

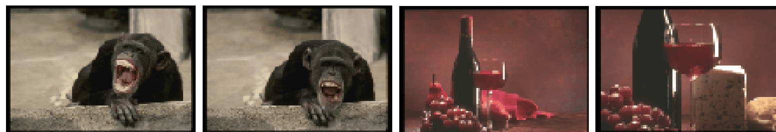

Fig. 3. Two test-pairs: monkeys (left), wine (right)

The number of buckets that must be searched to find the best matches decreases significantly when the *k-d* tree is build using the KLT vectors. We tested the trees on our high dimensional feature vectors. Those test showed that using the original feature vectors all buckets are searched (when no threshold is used).

Using the KLT vectors to build the tree, only about 50% of the buckets are searched. We therefore will use the KLT vectors to build our k-d trees.

3 Optimizing k-d trees for Image Retrieval

In this section, we used the priority KLT k-d trees for indexing the Corel Studio Photo database consisting of 18,724 color images. For testing purposes, 30 test sets were created, each test set consisting of a group of visually similar images, as shown in figure 3. As features, we used the color histogram and statistical and spectral textures. The color histogram was computed from discretizing the color space into the 512 bins. Each bin corresponds to the index computed from the most significant 3 bits from RGB, respectively. The statistical texture method (Texture 1) measures the cooccurence matrix from the gray levels ([7]), and the spectral texture (Texture 2) method uses the Fourier coefficients ([7]).

Up to this point, we have described the general theory behind the variations in k-d tree methods. In this section we find the k-d tree parameters (bucket size and threshold) which minimize the access time in the context of image retrieval.

tree height	#buckets	bucket size	buckets searched	time (sec.)
9	512	37	48%	6.5
10	1024	19	45%	6.4
11	2048	10	41%	6.8
12	4096	5	40%	7.5

Table 1. Selecting Tree Height: average results of 20 test queries

3.1 Bucket Size

From our perspective, the k-d tree with the best bucket size is the k-d tree which finds the best `k-nn` matches in the least time.

To minimize the size of the k-d tree we will choose a bucket size that fills the buckets as much as possible, given the number of images. Test results (see [8]) verified that the search time is minimal when the tree is as full as possible.

When we use maximally filled trees, the bucket size is defined as follows:

$$\text{bucket_size}(h) = \lceil \frac{N}{2^h} \rceil$$

where $N = 18724$ and h is the tree height. (recall that the binary k-d trees are complete and that the height of a binary tree equals $log_2(\#\text{buckets})$). So, we now need to choose the best k-d tree height. We tried k-d trees with different heights on our COREL image collection. For every height we performed 20 test queries using the Color and Texture 1 algorithms, and we computed the average percentage of buckets that was searched to find the best 10 matches. Table 1 shows the results.

The results show that the percentage of buckets searched decreases with the tree height, but the search time increases when the tree gets very large. This can be explained by the fact that both the tree loading time and the tree search overhead (distance computation) increase significantly for large trees.

Since the results show that the search time does not vary dramatically for different bucket sizes, we will use 19 as the bucket size of all our *k-d* trees, regardless of the number of images used.

#images	2000	4000	9000	18724
#buckets	128	256	512	1024
Color	30%	30%	25%	20%
Texture 1	30%	25%	20%	20%
Overall	30%	27.5%	22.5%	20%

Table 2. Selecting Threshold: average results of 20 test queries per tree size

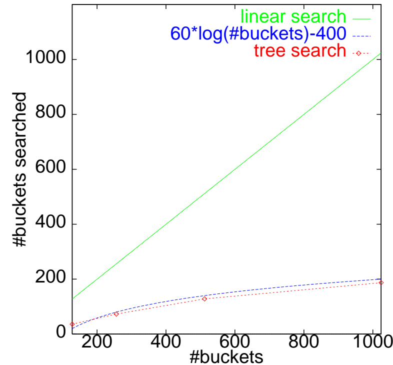

Fig 5. Linear vs. Tree search

3.2 Threshold

Having chosen a bucket size, we can now choose a threshold that limits the number of buckets visited. Given the size of the search tree we want to know how low we can choose the threshold, while the difference between the tree search results and the exact linear search results is negligible.

We created *k-d* trees with four different heights and we located the lowest threshold for those trees. For every tree size we performed 20 test queries using the Color and Texture 1 algorithms. Similar queries were performed using a linear search. We defined the difference between the tree search results and the linear results to be negligible when the average difference between the best 10, 20 and 30 matches found with the tree search and the best 10, 20 and 30 matches found with linear search was less then 0.5%. Table 2 shows the results.

Note that we choose the number of images such that the buckets of the *k-d* tree were as full as possible. The percentages indicate the thresholds that were used. The *Overall* row shows the average percentage of the buckets that is searched. This percentage decreases as the trees grow. Figure 5 shows a plot of the number of buckets that must be visited.

Because our tree search function needs to know what threshold to use with a given tree size, we choose the following function to compute the threshold:

$$\text{threshold}(\#\text{buckets}) = 60log_2(\#\text{buckets}) - 400$$

This function is also plotted in Figure 5. This plot shows that our function gives a good approximation of the threshold for trees with up to 1024 buckets (19456 images). When larger image collections are used it will be necessary to update the function.

3.3 User Search Response Times and Retrieval Accuracy

We measured the response time of the system to the user and then gave a breakdown of the time into initialization (program setup time before queries

can be made); search (time spent searching the tree); and response time (time including initialization, search, and display of results). In Table 3 the times are given using linear search and the priority KLT k-d search tree, respectively.

		initialization	search	sum	response time
Linear search	Color	0.21	7.1	7.3	7.5
	Texture 1	0.21	10.7	10.9	12.3
Tree search	Color	0.68	2.3	3.0	3.1
	Texture 1	0.68	3.0	3.7	3.9

Table 3. Search Response Time: average results of 20 test queries per algorithm

Table 4 describes the accuracy of the indexing method for each of the features. The accuracy is measured in terms of the ratio of similar images found in the "Top R" ranks, also known as normalized recall ratio.

Algorithm	Top 5	Top 10	Top 20	Top 50	Top 100	Top 500
Color	0.72	0.74	0.77	0.79	0.81	0.87
Texture 1	0.46	0.46	0.49	0.55	0.61	0.74
Texture 2	0.36	0.40	0.47	0.50	0.60	0.72

Table 4. Normalized recall ratios using 30 test-sets (78 images) embedded in 18724 images.

4 Conclusions

Solution of the find-the-similar-image problem requires solving the find-the-nearest-neighbor problem. Priority *k-d* search trees are an efficient method for finding nearest neighbors in high dimensional search problems. In this paper, we have adapted the search trees to the problem of image retrieval and found the best parameters regarding minimizing the access time. Tests on image collections with up to 18724 images showed that the threshold value can be approximated by a logarithmic function of the number of feature vectors.

References

1. J.H. Friedman, J.L. Bentley, and R.A. Finkel, *An Algorithm for Finding Best Matches in Logarithmic Expected Time*, ACM Transactions on Mathematical Software, 3(3), p. 209-226, Sep. 1977
2. J.L. Bentley, *K-d Trees for Semidymamic point sets*, in Proc. 6th Ann. ACM Sympos. Comput. Geom., p. 187-197, 1990
3. D.A. White and R. Jain, *Algorithms and Strategies for Similarity Retrieval*, Visual Computing Laboratory, University of California, San Diego, 1996
4. D.A. White and R. Jain, *Similarity Indexing: Algorithms and Performance*, Visual Computing Laboratory, University of California, San Diego, 1997
5. S. Arya, *Nearest Neighbor Searching and Applications*, PhD thesis, Computer Vision Laboratory, University of Maryland, College Park, 1995
6. S. Arya, D. Mount, *An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions*, in Proc. 5th ACM-SIAM Sympos. Discrete Algorithms, p.573-582, 1994
7. R.C. Gonzalez and R.E. Woods,*Digital Image Processing*, Addison-Wesley, 1992
8. R. Egas, Benchmarking of Visual Query Algorithms, Internal Report 97-06, Computer Science Dept., Leiden University, 1997