3-WAY-TREES: A SIMILARITY SEARCH METHOD FOR HIGH-DIMENSIONAL DESCRIPTOR MATCHING

Eduardo Valle[1], *Matthieu Cord*[2], *Sylvie Philipp-Foliguet*[1]

Équipes Traitement des Images et du Signal — ETIS UMR CNRS 8051[1]
Laboratoire d'Informatique de Paris 6 — Université Pierre et Marie Curie[2]
valle@ensea.fr, matthieu.cord@lip6.fr, philipp@ensea.fr

ABSTRACT

In this paper we look into the problem of high-dimensional local descriptor matching for image identification on cultural databases, presenting an important improvement over a classic method, the KD-Tree. Our method, the 3-Way Tree, uses redundant, overlapping sub-trees, in order to avoid the boundary effects that disrupt the KD-Tree in higher dimensionalities, achieving more precision for the same querying times.

Index Terms **—** local descriptors, image identification, nearest neighbor search, kd-tree, descriptor matching

1. INTRODUCTION

Institutions possessing large image databases, such as museums, archives, and news agencies, often face the separation of an image from its metadata: the title, authors, description and other information are missing. This arises when source references are absent or irregular. Since the meaning of a document depends on its context, the lack of metadata reduces its usefulness. The institutions are asked to retrieve all information about a document, relying only on visual information.

A related problem is the detection of copyright infringements. In this context, the users believe that a database contain images of their possession. They want to confront this suspicious dataset against their own image set and find all intersections.

Both problems are further complicated because the query images may have been distorted from the originals by translations, rotations, scale changes, changes in brightness and contrast, occlusions, etc.

Some image retrieval methods are also adequate for image identification. Local descriptors-based methods, in special, have been shown to be of great value, due to their strong robustness to occlusions, cropping and geometric distortions. [3][6][8] Instead of creating a single descriptor per image, those methods will identify a large number of PoI (Points of Interest), and compute a local descriptor around each one of those points.

In order to perform the identification, first, in an offline phase, which is done only once, the image set is prepared: each image has its PoI detected and described, and the descriptors are indexed in a large database in order to facilitate the matching. Then, in the online phase, the PoI of the query image are detected and described, and each descriptor is matched with the descriptors in the database. Each matched descriptor votes to the image to which it belongs.

This method is robust, because the descriptors are many. Even if some descriptors are matched incorrectly, giving votes for the wrong images, only a correctly identified image will receive a significant amount of votes.

Unfortunately, the multiplicity of descriptors brings also a performance penalty, since hundreds, even thousands of matches must be found in order to identify a single image. The matter is made worse by the high dimensionality of the descriptors, which makes each match operation very expensive.

In this paper we will devise a method that will allow us to match the descriptors efficiently, in order to benefit from the local descriptors advantages, without the performance drawbacks. In section 2 we will explain how the descriptor matching is performed, and why the dimensionality of the descriptors disrupts the process. In section 3 we will briefly explain the importance of reducing the number of random accesses to the disk. In section 4 we will introduce our method, the *3-Way Tree* and explain how the overlapping sub-trees minimize the damaging effects of the growing dimensionality. In section 5 we explain how we set up the experiments to compare our method to the classic KD-Tree, and in sections 6 and 7 we present our results and conclusions.

2. THE KD-TREE AND THE KNN SEARCH

The matching of the descriptors is performed by an operation known as *k nearest neighbors search* or *kNN search,* which consists in finding the *k* elements which are the most similar to a given query descriptor.

An obvious solution is the sequential search, where each element of the base is compared to the query, and the ***k*** most similar are kept. However, this brute-force solution is acceptable only for small bases, being unfeasible in our context. A better solution is to prearrange the data in an index, in order to accelerate the search.

The *KD-Tree* is a classic data structure for multidimensional indexing. Basically, it is a binary search tree where each node of the tree splits the search space along a given dimension. The sub-trees are thus implicitly associated to regions of the descriptors space. The leaves of the tree are called *buckets*, and contain a certain number of descriptors, which is decided *a priori*.

For reasons of space, we can not give an account of this structure. Fortunately, the literature is overabundant on this subject. The reader is invited to refer to the original article on [4], and the excellent tutorial on [1], which is available on the World Wide Web.

Unfortunately, the KD-Tree, like most kNN search methods, fails on high dimensional spaces. While the search time can be made to grow only logarithmically to the size of the base, it will grow exponentially to the dimensionality of the elements. For small dimensionality (between 2 and 12), it will perform efficiently enough to allow an exact or near-exact solution in reasonable time.

For higher dimensionalities, the KD-Tree can be adapted to give approximate results using a technique named *Best-Bin-First* [5] or *Priority KD-Tree Search* [7], but for very high dimensionalities (more than 30) the trade-off between precision and efficiency will become progressively more severe.

One of the worst problems for KD-Trees on higher dimensions is the aggravation of boundary effects. The probability that the query will fall near to at least one edge of the region associated to the leaf approaches 1 as the dimensionality increases, forcing the KD-Tree to explore many regions in order to find a good set of approximate neighbors.

3. INDEXES AND DISK ACCESS

Practical implementations of the naïve algorithm are surprisingly difficult to beat, mainly because of its sequentiality. Once the database becomes too big to fit the primary memory, one faces the limitations of disk access, where sequential access is up to 10 times more efficient than random access, because of the high costs involved in relocating the read/write magnetic heads.

Every time the KD-Tree accesses a bucket (which are stored in the disk), it must make a random access. Because the order in which the buckets are accessed is essentially unpredictable, there is no way to optimize the disk access. Because of the boundary effects, many buckets may need to be visited, resulting in severe disk access costs.

4. THE 3-WAY TREES

In order to avoid boundary effects, it would be desirable to choose a region where the query would be the most centralized possible. However, because the regions are built in the offline phase, for some queries it will be impossible to find a region where the query will be centralized. In fact, because of the aggravation of boundary effects on higher dimensionalities, it is almost certain that the query will fall near to the boundaries of at least one of the dimensions of the region.

Customizing the partitioning for each query would be the ideal solution, but of course, its cost prevents it from being of practical consideration. The next best solution is providing *some* redundancy in the partitioning scheme, in the form of overlapping regions. In that way, the most adequate region can be selected, accordingly to the current query. This is the idea behind the *3-Way Trees*.

The 3-Way Trees are, essentially, ternary trees, where the left and right sub-trees are equivalent to the ones of a KD-Tree. The middle subtree, however, is overlapping. It adds redundancy to the tree, containing the same points as half each other sub-trees. [Figure 1]

To build a 3-Way Tree, the following algorithm is used:

1. The dimension with maximum *interquartile range* is determined. This dimension is chosen as the splitting dimension;
2. The *median element* is chosen as the *pivot* for the partitioning, left and right sub-trees are thus balanced;
3. All elements smaller or equal than the pivot on the splitting dimension are put in the left subtree;
4. All elements greater than the pivot on the splitting dimension are put in the right subtree;
5. All elements greater than the first quartile and smaller or equal than the third quartile are put in the middle subtree;
6. The three sub-trees are built recursively from the step 1;
7. The recursion stops once the number of elements is small enough to fit in a *bucket*. A leaf is then created with all remaining elements.

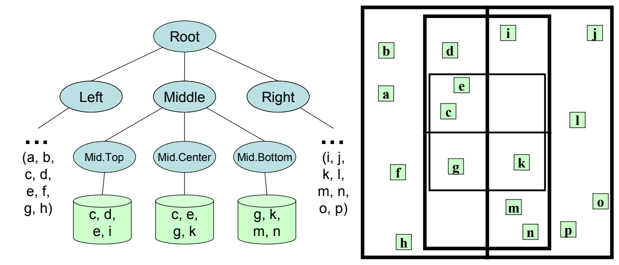


Figure 1. A 3-Way Tree and its associated space. For clarity, only the middle branch is developed.

Using the **interquartile range** (the difference between the third and first quartiles, i.e., the 75th and 25th percentiles) as the criterion of spread in the 3-Way Trees is very important, since the sensitivity to outliers of other criterions (like the **range**, which simply takes the difference between the maximum and minimum values) may spoil the results. In fact, as we are going to see, even the simple KD-Tree show benefits from using the interquartile range.

The redundancy added by the 3-Way Trees is exploited in the search operation in the following manner:

1. The tree is traversed top-down;
2. The subtree to explore is determined, by choosing, among the 3 nodes, where the query falls the most centralized;
3. If the chosen node is not a bucket, recurse to step 2;
4. Otherwise, explore sequentially all the points on the bucket, choosing the *k* nearest neighbors.

The greatest advantage of the 3-way Tree is that it keeps the query roughly centralized all the way down the search tree, avoiding the problems brought by boundary effects. In that way, only one leaf must be explored. Even for huge databases, all non-leaf nodes may be made to fit on primary memory. Therefore, only one random access must be made to the disk, resulting in savings in performance.

The 3-way Tree exchanges the disk space occupied by the overlapping nodes for the increased performance. The offline processing time is proportional to the redundancy, due to the need of propagating the elements from level to level in the tree. The amount of disk space needed to store the leaves of the 3-way Trees is proportional to:

$$N\ {}^{3}\!/_{2}\ {}^{H\ \ 1} \qquad \textbf{Eq. 1.}$$

where N is the number of descriptors in the database and H is the height of the tree.

5. EXPERIMENTAL SETUP

5.1. Database: images and descriptors

To compare the methods, we used a database consisting of 15 transformations over 100 original images, summing up to 1,500 images. Each image had its PoI detected and described using SIFT, a very robust method, but one which uses descriptors of 128 dimensions — a very high dimensionality [2]. The final base contained almost two million nine hundred thousand descriptors. The database contained photos from the XIX and early XX centuries.

The transformations were three rotations, four scale changes, two shearings, four changes in the gamma curve and two smoothings (3×3 and 5×5 grids). [Figure 2]

We've chosen those transformations because they distort to some extent the value of the descriptors, making the descriptor matching meaningful. Croppings and occlusions, even if they are very important, make some points disappear altogether, while letting others pass with their exact values unchanged. Likewise, brightness and contrast changes are invariant under SIFT, so we opted for gamma changes instead.

5.2. Compared methods

We compared the 3-Way Trees with two versions of the KD-Tree, using different criterions of spread of the data: the range and the interquartile range. We wanted to isolate which gains were due to the use of the interquartile range and which were due to the redundant segments.

In all methods, just the first bucket was explored. This was in order to put all methods in the same restriction that only one random access to the disk was allowed, and keep execution times the same.

We set the maximum bucket sizes to 1024, 2048, 4096, 8192 and 16384 descriptors. The actual bucket size varies, and was indicated in the horizontal axis of the graphs.

For the sake of comparison, we also run the KD-Tree in a more conventional scenario, allowing it to make 16 random accesses to the disk, using buckets of 1024 descriptors. This would take more than 10 times the execution time than the other methods, but it is instructive in order to see how much one would gain in precision.

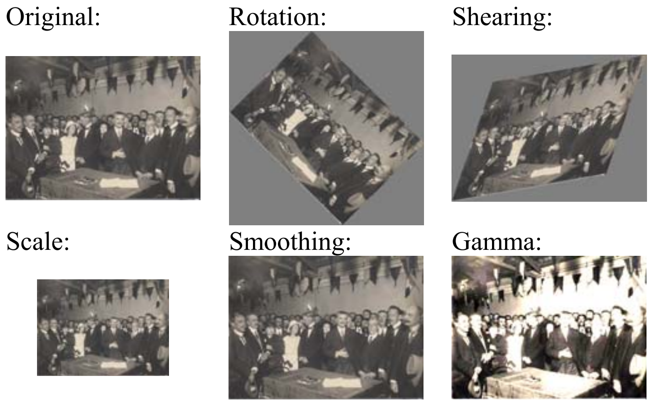


Figure 2. Original image and some transformations.

5.3. Measurements and ground truth

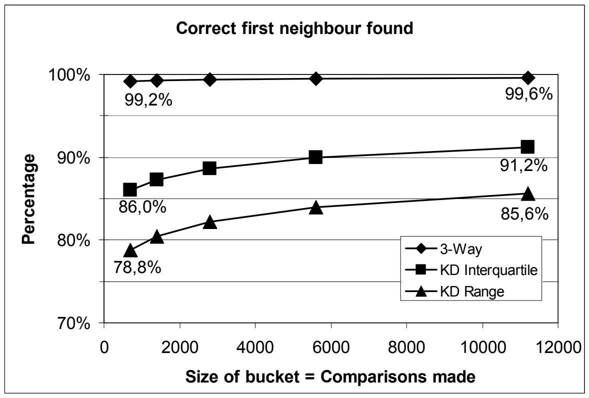
Each one of the 100 original images had its descriptors computed, resulting in 263,968 query descriptors. The experiment consisted in finding the 20 nearest neighbors of each one of those descriptors using each method.

The answers were compared with a ground truth which was computed using the sequential search. Two measurements were taken:

- The percentage of queries were the *first* nearest neighbor was correct;
- The average number of nearest neighbors among the 20 that were correct.

6. RESULTS

We compiled the results of the experiments in the three graphs of [Figure 3]. An important parameter in both methods is the maximum allowed bucket size. The actual size of the buckets may vary, and is indicated in the horizontal axis. The size of the bucket corresponds to the number of descriptors examined.



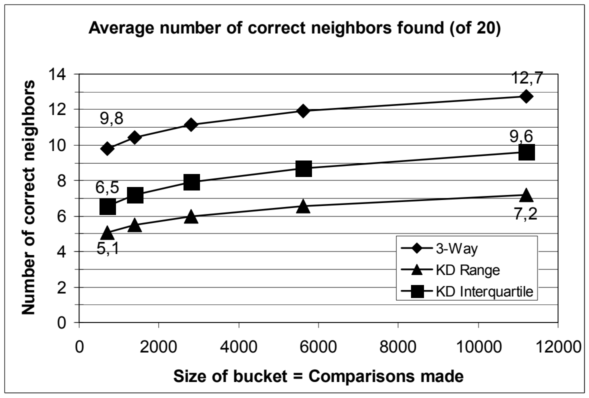


Figure 3. The results of the experiments.

The worst case of the 3-Way Tree performed better than the best case of the KD-Tree. The performance of all methods grows with the size of the bucket, which is not surprising, since more descriptors get to be examined.

Besides the results shown on the graphs of [Figure 3], we executed an independent run of the KD-Tree examining 16 buckets of a maximum of 1024 descriptors (actual size of 701 descriptors). It gave 99.8% correct first neighbors found and an average of 14.3 correct neighbors found. Being more than 10 times slower, the gains brought by a KD-Tree in this more conventional scenario did not compensate the drawbacks.

7. CONCLUSIONS

We have been investigating ways of making high-dimensional descriptor matching more efficient, by developing better heuristics for nearest neighbors search. We have added redundancy to the KD-Tree in order to boost its efficiency on disk storage while keeping a very good precision. Using redundant regions, we avoided the boundary effects that plague the KD-Tree, keeping the queries roughly centralized all the way down the tree. We trade off the storage space of the overlapping regions for the gains in precision.

Using our scheme, one can benefit from the precision brought by the use of local descriptors, while minimizing its performance drawbacks. The use of the 3-Way Trees allows us to do image identification in large cultural databases, using local descriptors, providing, at the same time, precision and timeliness.

8. ACKNOWLEDGEMENTS

Eduardo Valle is sponsored by CAPES through the CAPES/COFECUB program. The photography database was kindly provided by the Arquivo Público Mineiro, the state archive of Minas Gerais, Brazil.

9. REFERENCES

[1] A. W. Moore. “An introductory tutorial on kd-trees,” extract from *Efficient Memory-based Learning for Robot Control*, Technical Report No. 209. Computer Laboratory, University of Cambridge, 1991.

[2] D. Lowe, “Distinctive Image Features from Scale-Invariant Keypoints,” in *International Journal of Computer Vision*, Vol. 60, N. 2, pp. 91–110, 2004.

[3] E. Valle, M. Cord and S. Phillip-Foliguet, “Content-Based Retrieval of Images for Cultural Institutions Using Local Descriptors,” in *Geometric Modeling and Imaging — New Trends* (GMAI'06), 2006.

[4] J. H. Friedman, J. L. Bentley and R. A. Finkel, “An Algorithm for Finding Best Matches in Logarithmic Expected Time,” in *ACM Transactions on Mathematical Software*, Vol. 3, N. 3, pp. 209–226, 1977.

[5] J. S. Beis and D. Lowe, “Shape Indexing Using Approximate Nearest-Neighbour Search in High-Dimensional Spaces,” in *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, p.1000, June, 1997.

[6] L. Amsaleg , P. Gros , S-A. Berrani, “Robust Object Recognition,” in *Images and the Related Database Problems, Multimedia Tools and Applications*, v. 23 n. 3, p. 221–235, August 2004

[7] S. Arya. “Nearest neighbor searching and applications,” Technical Report CAR-TR-777, Center for Automation Research, University of Maryland, June 1995.

[8] Y. Maret, S. Nikolopoulos, F. Dufaux, et al. “A Novel Replica Detection System Using Binary Classifiers, R-Trees, and PCA,” in *International Conference on Image Processing*, Parallel Computing in Electrical Engineering. IEEE, 2006.