Optimised KD-trees for fast image descriptor matching

Miguel Alcón Doganoc
Universitat Politècnica de Catalunya
Barcelona, Spain
miguel.alcon@est.fib.upc.edu

1 SUMMARY

This paper focuses on improving the *KD-tree* for indexing a large number of *SIFT* (Scale invariant feature) and other types of image descriptors, which usually are large point vectors of high dimensionality. For instance, a *SIFT* descriptor is a 128-dimensional vector normalized to length one. In order to achieve their objective, the authors of the paper extended priority search to priority search among multiple trees, by creating them with the same data but with different structure. Also, they wanted the search of a point to be simultaneously independent among the trees.

They proposed 3 different types of *KD-trees* that have the properties described above.

- ***NKD-tree.*** Given n and m, a *NKD-tree* is a group of m trees constructed with the same data but, as said before, with different structure. This difference is achieved by rotating each point of the data with an arbitrary rotation matrix R, which is unique for each of the trees. Related with the search of a point, the authors propose to search the point among multiple trees in the form of a concurrent search with a pooled priority queue. After descending each of the trees to find an initial nearest-neighbor candidate, the best candidate from all the trees is selected. They then pool the node ranking by using one queue to sort the nodes from all m trees. As a result, all of them are searched in the order of their distance from the query point simultaneously.
- ***RKD-tree.*** This kind of tree is almost the same than *NKD-tree*, but it achieves the difference of structures in the construction stage by selecting a random dimension in which to subdivide the data, instead of selecting it sequentially. This dimension is selected among a few dimensions in which the data has a high variance. No rotation of the data is needed.
- ***PKD-tree.*** Again, the only difference between this kind of tree and the other two is how it achieves the difference of structure of all its trees. In this case, they align the data using *PCA* (Principal Component Analysis) and then build multiple trees using rotations that fix the space spanned by the top principal axes.

Finally, the authors test the proposed trees. They conclude that *NKD-trees* performs as well as *RKD-trees* (success rate from 75% to 88%), but that *PKD-tree* applying random *Householder*[1] transformations to the data points, in order to preserve the *PCA* subspace of appropriate dimension, leads to the highest success rate (95%).

2 IMPORTANCE

Matching image descriptors for an application like image recognition must be fast. So, increasing the speed of the query is really important in the *CV* (Computer Vision) field and the other ones that take advantage of it (like robotics).

However, this improvement is not only relevant in *CV*, since, at the end, the authors are trying to increase the speed of searching k-dimensional points within some dataset using *KD-trees*. As it is seen in the paper, this data structure performs really good in matching image descriptors quickly, which is not a surprise since it is a well-known space-partitioning data structure for organizing points in a k-dimensional space.

3 EXPERIMENTS

Since I do not know how to perform rotations in a k-dimensional space (with $k > 3$) or creating the *Householder* matrices, again, in a k-dimensional space, I was not able to implement *NKD-tree* nor *PKD-tree*. Furthermore, some formulas related to their implementation are not complete in the paper. I also made some research on how to do it, but I did not achieve anything. So, my experiments are focused on comparing the basic *KD-tree* with the *RKD-tree* proposed in the paper. Before starting with the experiments, I implemented in *C++* both *KD-tree* and *RKD-tree*.

3.1 *KD-tree* implementation

I only implemented the required operations of *KD-trees* to make the experiments, which are construction and nearest neighbor search (*NNS*). I based my implementation in [2], but I made the following changes.

- **Construction.** To select the pivot among a point list, in each recursion step, I used the *Quick Select* algorithm. Moreover, as the authors did in the paper, I implemented the random selection of the axis that is going to be used to create the splitting plane among a subset of the axis with high variance. In order to use less space, instead of saving one point in each node of the tree, I saved the index of the point's position within the main vector.
- **NNS.** To know if the other side of the splitting hyperplane crosses the hypersphere around the search point (with radius equal to the current nearest distance), I followed what is explained in [1]. I also created a new type of search, the *limited search*, which, given a positive integer n, it limits the search to n nodes.

3.2 *RKD-tree* implementation

For the *RKD-tree*, I followed what is explained in the paper. In this case I want to explain what I did in detail.

- **Construction.** Given n, m, k and a vector of points V, first the algorithm computes the top k axis with highest variance, and then it generates the vector of its indices (I will refer to this vector as HV). Finally it creates m *KD-trees* as explained

[1] Explained in the paper, not necessary for my experiments.

before. In order to create the trees without using a lot more of space, it passes to them a pointer to V and to HV. The algorithm also stores n as the maximum number of nodes that the search algorithm can visit. At this moment, we have m *KD-trees* with different structure, ready to be explored.

- **NNS.** Given a point p that we want to search, first, for the root of each tree, the algorithm computes the distance between the root's point and p. It stores each root together with the computed distance inside a *priority queue*. Then, while $n > 0$, while the queue is not empty and the algorithm does not find the exact point p, it takes the top element e of the queue, it updates the closest point and its distance if e's point is the closest one to p visited so far. After that, it computes the distance between p and each of e's children, if possible, and add them to the queue. At the end of the algorithm, we have the closest point to p that the algorithm can find visiting at most n nodes of all the trees.

4 RESULTS

5 CONCLUSIONS

REFERENCES

[1] Nguyen, T. Nearest neighbor search (pg. 15). http://andrewd.ces.clemson.edu/courses/cpsc805/references/nearest_search.pdf, 2019.

[2] Wikipedia. kd-tree. https://en.wikipedia.org/wiki/K-d_tree, 2019.