Optimizing kd-trees for scalable visual descriptor indexing*

You Jia† Jingdong Wang‡ Gang Zeng† Hongbin Zha† Xian-Sheng Hua‡
†Key Laboratory of Machine Perception, Peking University ‡Microsoft Research Asia
jiayou50@gmail.com gang.zeng@pku.edu.cn zha@cis.pku.edu.cn
{jingdw, xshua}@microsoft.com

Abstract

In this paper, we attempt to scale up the kd-tree indexing methods for large-scale vision applications, e.g., indexing a large number of SIFT features and other types of visual descriptors. To this end, we propose an effective approach to generate near-optimal binary space partitioning and need low time cost to access the nodes in the query stage. First, we relax the coordinate-axis-alignment constraint in partition axis selection used in conventional kd-trees, and form a partition axis with the great variance by combining a few coordinate axes in a binary manner for each node, which yields better space partitioning and requires almost the same time cost to visit internal nodes during the query stage thanks to cheap projection operations. Then, we introduce a simple but very effective scheme to guarantee the partition axis of each internal node is orthogonal to or parallel with those of its ancestors, which leads to efficient distance computation between a query point and the cell associated with each node and yields fast priority search. Compared with the conventional kd-trees, our approach takes a little more tree construction time, but obtains much better nearest neighbor search performance. Experimental results on large scale local patch indexing and image search with tiny images show that our approach outperforms the state-of-the-art kd-tree based indexing methods.

1. Introduction

Nearest neighbor (NN) search is a fundamental problem in the research communities of computational geometry [10] and machine learning [26]. It also plays an important role and has many applications in computer vision. For instance, NN search is adapted for fast matching to establish the correspondences of the local points between two images, which is required in many applications, such as wide-baseline matching for 3D modeling from photo databases [28], and panorama building [8]. Local feature-based recognition methods rely on NN search to search a huge database of patch descriptors [14]. Content-based image and video retrieval also depends on the NN search technique for fast indexing and searching on a large database.

In this paper, we study the kd-tree based indexing and searching technique and look at the specific problem of matching image descriptors for the applications of image retrieval and recognition. The (approximate) nearest neighbor (ANN) search algorithms [2, 18], based on kd-trees, have been applied to large scale indexing and searching. Generally, this type of search algorithm may not be effective, particularly in the high-dimensional case. However, it has been proved to work well [3, 18, 21, 23, 25, 27] in some computer vision applications, such as searching the similar patches with the SIFT descriptor.

The basic idea of kd-trees is to recursively partition the space into two subspaces to construct a binary tree with each internal node associated with a subspace (cell). At each internal node a *partition hyperplane* is generated to split the space. The common-used way to construct kd-trees finds a *partition hyperplane* only from coordinate axis-aligned hyperplanes, which limits the capability of partitioning the space. Thus, this leads to limited ability of indexing and searching. It is required to search more nodes to increase the probability of finding the true nearest neighbor.

Searching for the nearest neighbor with kd-trees, given a query point, requires a descent down the tree to a leaf node to obtain the first candidate. But, this first candidate may not necessarily be the nearest neighbor. *Priority search*, in which the cells are searched in the order of the distances from the query point, is recommended [1, 3] to first search the cells containing better candidates with larger probability. One of the keys in *priority search* is the cost of computing the distance of the query point from the cell for maintaining a *priority queue*, which is an important factor of affecting the query time.

The purpose of this paper is to scale up kd-trees, by finding the near-optimal space partitioning, which leads to searching for the true nearest neighbor with greater prob-

*This work was done when You Jia was an intern at Microsoft Research Asia

ability, but avoiding accessing more nodes. Our approach divides the space for each internal node by searching for a non-coordinate-axis-aligned hyperplane generated from a few coordinate axes in a binary combination way. This technique yields acceptable construction time. The big advantage is that the query time is almost the same with that of the conventional kd-tree to access the same number of leaf nodes and the query precision is better than that of the current state-of-the-art kd-trees given the same query time.

Our approach follows the below criteria to scale up the kd-trees for fast indexing and searching. First, the variance of the projections of the data points, along the *partition axis*, which is a direction that is perpendicular to the partition hyperplane, should be as large as possible to yield good space partitioning. Second, it should be computationally efficient to decide which child node a query point lies in for each internal node, i.e., the projection operation along the *partition axis* is not costly. Third, the partition hyperplane of each node should be perpendicular to or parallel with those of its ancestors in order to efficiently compute the distance of a query point from each cell to maintain the priority queue for *priority search* to find the nearest neighbors. Finally, we build multiple randomized kd-trees by introducing a randomized scheme to select the partition hyperplane for efficient search. We present a set of comprehensive experiments to demonstrate our approach.

2. Related work

The kd-tree, introduced in [4], aimed to generalize a binary search tree to high-dimensional data. Several variants in building a tree, including randomisation in selecting the partition axis and value, were investigated. A kd-tree with a theoretic logarithmic search-time was proposed in [12]. The kd-tree has been widely applied to search in a low-dimensional space, e.g., ray tracing in computer graphics [19]. However, these algorithms with the logarithmic search time do not apply to trees of high dimension, where the search time may become almost linear.

The reason of the efficiency diminishing in high dimensional data is that searching a kd-tree usually takes a lot of time to backtrack through the tree to find the optimal solution. By limiting the amount of backtracking, the certainty of finding the true nearest neighbors is sacrificed and replaced with a probabilistic performance. Recent research has therefore aimed at increasing the probability of finding the true nearest neighbor while keeping backtracking within reasonable limits. For instance, the best-bin-first search [2] and a priority search [3] were proposed for fast approximate nearest neighbor search.

Large scale content-based image and video retrieval and local feature based object recognition have been attracting a lot of attention in the computer vision community. The kd-tree based ANN search methods were proved very successful in matching high-dimensional visual descriptors (e.g., 128-dimensional SIFT features) [18, 25]. As reported in [25], the kd-tree yields better performance than the hierarchical vocabulary tree [24] as an aid to K-means clustering. In [23], an effort was made to automatically select the fast approximate nearest neighbor search algorithm from the kd-tree and the hierarchical K-means tree. There are several recent published works [21, 27] in computer vision. [21] aimed at learning a Mahalanobis SIFT representation for fast kd-tree indexing. Principal component analysis was adopted in [27] to preprocess the data points by rotating them using the principal eigenvectors.

There are some works relevant to ours in the tree-structure based approximate nearest neighbor search algorithms, which also relax the constraint that the partition hyperplane is required to be perpendicular to a coordinate axis. The representative works were PCA kd-trees [29] and PCA for generalized trees [20]. These two methods find the partition hyperplane from the leading eigenvectors of the covariance matrix of the data points for each node, which essentially corresponds to the axis along which the projections of the data points have the greatest variance. This type of partition hyperplane selection leads to better space partitioning. However, its scalability in tree construction and particularly querying process is very weak, because the computation of the principal eigenvectors is very expensive in the case of large scale data and large scale dimension and the querying process, which requires to perform an inner product between the partition axis and the query point to decide the order to traverse the tree, is quite time-consuming. Therefore, PCA kd-tree is less applied compared with the conventional kd-tree.

There exist many other NN search algorithms. A metric tree, e.g., ball trees [22], spill trees [17], and vantage point trees [31], is a tree structure to index data in metric spaces, which exploits the properties of metric spaces, such as the triangle inequality, to make accesses to the data more efficient. More descriptions of metric trees can be found in [5], and some experimental comparisons were presented in [16]. In this paper, we focus on scaling up kd-trees and would not investigate the comparison of our approach with other metric trees. Locality sensitive hashing (LSH) [9] is another type of ANN technique that essentially performs probabilistic dimension reduction of high-dimensional data. The basic idea is to hash the input data points so that similar points are mapped to the same buckets with high probability. LSH was also applied in the computer vision applications [15, 26]. But the performance, using LSH into computer vision problems, is not as good as kd-trees [23, 27].

The tree structure based techniques have also been applied to supervised learning in machine learning, e.g., decision tree [7], and random forests [6]. These methods are not comparable with tree structure based indexing algorithms

for unsupervised problems because they are specially for supervised problems.

3. Backgrounds

Given a set of n data points $\mathcal{X} = \{\mathbf{x}_1, \cdots, \mathbf{x}_n\}$ with $\mathbf{x}_i \in \mathcal{R}^k$ being a k-dimensional point, the goal is to build a tree structure to index these data points so that the nearest neighbors of a query vector $\mathbf{x}_q$ can be fast found. Before presenting our approach, we review the backgrounds, including the conventional kd-tree and the PCA kd-tree.

Construction A kd-tree (k-dimensional tree) is a special case of binary space partitioning trees, which is constructed in a recursive manner. At the root, the data points are split into two halves by a *partition hyperplane*. Then each half is assigned to one child node, and is recursively split in the same manner to create a balanced binary tree. The leaf node may contain a single point or more than one points in different implementations. In this way, each node in the constructed kd-tree corresponds to a cell in $\mathcal{R}^k$, bounded by a set of partition hyperplanes.

Here, a partition hyperplane is perpendicular to a *partition axis* and decided by a *partition value*. The partition axis in the conventional kd-tree is the coordinate axis with the greatest variance, and the partition value is the median of the projections of the data points along the partition axis.

Search To find the nearest neighbor of a query point, a top-down searching procedure is performed from the root to the leaf nodes. At each internal node, it is required to inspect which side of the partition hyperplane the query point lies in, then the associated child node is accordingly accessed. The descent down process requires $\log_2 n$ (the height of the kd-tree) comparisons to reach a leaf node. The data point associated with the first leaf node is the first candidate for the nearest neighbor, which is not necessarily the true nearest neighbor. It must be followed by a process of backtracking, or iterative search, in which other leaf nodes are searched for better candidates. The widely used scheme with high chance to find the true nearest neighbor early is *priority search* based on a *priority queue*, in which the cells are searched in the order of their distances from the query point. The search terminates when there are no more cells within the distance defined by the best point found so far.

The nearest neighbor search in the high-dimensional case [2] may require visiting a very large number of nodes, and even the process costs linear time. Therefore, alternatively, an approximate nearest neighbor search is usually performed, through an advanced search termination scheme, e.g., after searching a specified number of nodes, or if the distance from the closest cell to the query exceeds $\delta = d(\mathbf{x}_p, \mathbf{x}_q)/(1+\epsilon)$, where $\mathbf{x}_q$ is the query point, $\mathbf{x}_p$ is the NN found so far, and ϵ is a positive termination parameter, which guarantees that no subsequent point to be found can be closer to q than δ. In this manner, the search is guaranteed to have some certain probability to obtain the true nearest neighbor.

PCA kd-tree The basic idea of PCA kd-tree [29] is that the partition hyperplane at each node is found without the coordinate-axis-aligned constraint so that the variance of the projections along the corresponding partition axis is maximum. Such a partition axis can be easily obtained through performing PCA over the data points associated with each node. This way of selecting the partition hyperplanes leads to good space partition, but with high construction cost due to the costly principal component computation. On the other hand, in the query stage, the projection of a query point to the partition axis costs $O(k)$ time, and the computation of the distance from a query point to a cell is also expensive. These drawbacks result in that in practice PCA kd-tree is not applicable for large scale and high-dimensional data.

4. Our approach

The performance of the nearest neighbor search with the kd-tree relies on the following aspects. The first aspect is how to split the space. Better space partition leads to searching fewer nodes to get good performance. The second one is the projection cost of visiting an internal node during the process of traversing the tree. The last one is how tight the lower bound of the distance of a query point from one cell is estimated and how efficient the estimation is, which determine in which order the nodes are accessed and how many leaf nodes are accessed to find the nearest neighbors. All the above depend on the partition hyperplanes.

In the conventional kd-tree, the *partition hyperplane* is limited to be orthogonal to the coordinate axes. This limitation results in that the space is not well partitioned and hence leads to searching more nodes to get the nearest neighbors. It should be noted that the scheme of choosing the partition hyperplanes has the benefit side since it guarantees that those hyperplanes are orthogonal to or parallel with each other and that the distance estimation of a query point from a cell is very fast. The PCA kd-tree yields better space partitioning, but suffers from the high querying cost due to expensive cost of accessing internal nodes.

Therefore, we propose a novel scheme to scale up kd-tree so that the proposed kd-tree yields better space partitioning and leads to better querying performance. We find a suboptimal binary combination of coordinate axes with great variances as the partition axis, which in some sense can be viewed as the approximation of the true axis with the greatest variance, the principal axis, but avoiding the

time-consuming computation in PCA kd-tree. Moreover, we introduce an efficient way to make sure that the partition hyperplane of each internal node is orthogonal to or parallel with those of its ancestors, which helps compute the distance of a query point from the cell with lower computational cost. Further, we extend the proposed kd-tree to multiple randomized trees to expect higher performance.

4.1. Algorithm

Forming partition axes We propose to use a binary combination of coordinate axes as the partition axis with a great variance in each node. Let a set of unit vectors, $\{\mathbf{n}_1 \cdots \mathbf{n}_k\}$, represent the k coordinate axes. A candidate partition axis in our approach can be written as $\frac{1}{\sqrt{l}} \sum_{i=1}^{k} w_i \times \mathbf{n}_i, w_i \in \{-1, 0, 1\}$. Here l is the number of non-zero entries in the weight vector $[w_1 \cdots w_k]^T$. The term *binary* means whether a coordinate axis contributes the partition axis.

Since the number of candidate partition axes has grown from k to $(3^k - 1)/2$, it is obvious that the probability of finding a partition axis close to one principal component would become much higher. In addition, the projection operation of any point onto such a partition axis will only take $(l - 1)$ addition operations. As we will show later, a small value for l would suffice for most nodes, thus saving a considerable amount of time over a projection on real-valued partition axes that would take $O(k)$ multiplications.

To check all the binary combinations, it is necessary to compute the variances for each binary combination, which leads to the computation of the covariance matrix over all the dimensions. Obviously, in the high dimensional large scale cases, this computation will be too computationally expensive even for offline processing. Therefore, we introduce an approximate but effective way that only needs to compute a partial covariance matrix over a few dimensions.

We pick out the top d coordinate axes ranked by variances as dominant axes, and find a partition axis with the large variance from binary combinations of them. As a result, the covariance matrix is computed only for these d dimensions. This scheme is based on the following observations. If a binary combination has a large variance, it is very likely that the axes forming it will also have large variances. On the other hand, it is possible that a low variance axis that has very strong linear correlations with some other high variance axes does make their combination's variance even higher. However, due to its strong linear correlations with other axes, it will not be quite helpful for partitioning the space. So leaving it out may not matter much. Consistent with this intuition, experiments have shown that $d = 10$ is enough when coping with visual descriptors.

To further speed up the construction, we enumerate the binary combinations using a greedy pruning scheme to discard some combinations that tend to have small variances with high probability. This scheme is performed in a sequential manner, from single coordinate axis, two-coordinate-axis combinations, to d-coordinate-axis combinations. After finishing each combination size, only the top d ranked by variance will be kept, each to be further combined with dominant axes not yet contained in it, thereby forming binary combinations of the next larger size.

This greedy pruning scheme introduces little degradation on the overall performance, but reducing the number of checked combinations from an exponential function of d to a polynomial one. We conducted an experiment of ANN searches on the trees with d set to 5 and 7, each with five random runs. The average precision loss is only 0.11% with a small variance of 1.44×10^{-5}. It should be noted that we have used two relatively small values for d, which means that more lower ranked combinations are pruned out. Therefore, it is a strong evidence to show that the greedy pruning scheme keeps almost all the useful combinations with large variances.

Orthogonalizing partition axes In order to calculate the distance from a query point to a cell in the way used in the conventional kd-trees, the partition hyperplanes along any root-to-leaf path must be perpendicular to or parallel with each other. If this condition is violated in the tree construction stage, even a reasonable estimation of this distance could take considerably more computational efforts for high dimensional data, e.g., in [20], its lower bound is estimated in $O(k)$ time for each node instead of $O(1)$. Hence, we present a simple but effective scheme by deliberately satisfying the condition.

To do this, we keep track of which coordinate axes have been used and which ancestor node used them while building the tree. In each node, after picking out dominant axes, those already used will be replaced by their combinations that are perpendicular to or parallel with all partition axes of ancestors. These added combinations will be taken as a part in the later candidate enumerating process. After this replacement, all further combinations become safe and no combination that violates the orthogonality and parallelism condition will be generated.

In this process, for each ancestor node that has used a subset of the axes in the dominant axes list, we will need to generate combinations out of this subset that are either perpendicular to or parallel with the partition axes used by the ancestor. The inspection of orthogonality and parallelism is very cheap in our case. It is easy to check whether the partition axes, formed by the binary combination of the coordinate axes, are orthogonal or parallel. This can be conducted by computing their inner product, $v = \sum_{i=1}^{k} w_i^1 w_i^2$. Since d, the number of the dominant coordinate axes, is small (in our experiments, it is about 10), the inner product can be reduced to be performed in a sparse operation that requires to add at most $2d$ integer variables. Finally, when a dom-

inant axis is used by more than one ancestors, only the intersection of the sets of combinations generated with those ancestors concerning this dominant axis is kept.

Since the variance along a partition axis will be greatly reduced after performing the partition on this axis, the chance that a coordinate axis appears in the dominant axes list will be much smaller if it is already used in an ancestor node. Even in case that most dominant axes have been used and the dominant axes list becomes quite short after the replacement procedure, we can simply rearrange the initial dominant axes list by adding a few extra lower ranked axes and perform the candidate combination enumeration again until we get enough number of candidate combinations.

4.2. Extension to multiple trees

We extend the single tree to multiple randomized kd-trees, in order to obtain the better performance. The basic idea is that the partition axis for each node is randomly selected from several candidate combinations with top great variances instead of deterministically selecting the candidate with the greatest variance. In the query stage, we also perform the searching simultaneously in the multiple trees through a shared priority queue, similarly to [27].

4.3. Complexity analysis

In this section, we will analytically compare the complexity of our modified kd-tree with the conventional ones. The analysis will also be justified by experimental results.

For building an internal node, the extra computational cost of our approach mainly lies in the calculation of the covariances, which takes $O(\bar{n}d^2)$ time with $\bar{n}$ being the number of points associated with an internal node. Additionally, enumerating binary combinations also introduces some overheads. Due to the greedy pruning scheme, however, the number of combinations checked is reduced to polynomial time cost $O(d^3)$ rather than exponential time cost $O(3^d)$. For orthogonality inspection, the complexity depends on both d and the depth of the node h, which is logarithmic to the size of the database. Overall, these two parts of the algorithm do not affect the asymptotic time complexity of building a node and run faster than the computation of the covariances.

In the query stage, a conventional kd-tree will spend only $O(1)$ time for both the computation of projection and query-to-cell distance in an internal node. The additional priority queue operation dominates the time complexity, $O(\log n_{queue})$, where n_{queue} is the number of nodes in the priority queue. Note that in approximate nearest neighbor search, n_{queue} does not directly rely on the scale of the search database, therefore can be sometimes taken as a big constant.

The only difference between our method and the conventional kd-tree is the projection operation. The projection of any query point onto a partition axis (a binary combination of axes), costs linear time with the number of non-zero weight entries, which in turn is not larger than d. But due to the more time-consuming priority queue operation and the small value of d, the change from $O(1)$ to $O(d)$ in the projection operation does not significantly affect the query efficiency. More importantly, the major dominator of the query time cost is visiting leaf nodes to compute the distances, where the two methods work identically. The cost in a leaf is $O(k)$ in most cases. Overall, our modified kd-tree accesses internal nodes a little slower than the conventional version. Tab. 1 shows the relevant time complexities.

5. Experiment

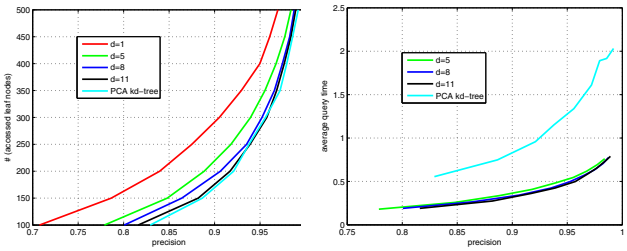
Data sets We justify our approach mainly over the Caltech 101 data set [11], which contains about 9000 images. We do not perform the nearest neighbor search directly over the small scale images. Instead, we extract a set of local features for each image, and then integrate all the sets of local features to construct the search data base. We extract the maximally stable extremal regions (MSERs) for each image, and extract a 128-dimensional SIFT feature for each MSER. On average, there are about 400 SIFT features for each image. In this way, we get a search data base containing around 4000k SIFT feature points. In our experiment, we randomly sampled 300k points from those points to build the search data base. To formulate the query, we randomly sampled 30k points from the original data points and guaranteed that these query points do not appear in the search data base.

In addition, to avoid the bias of the single data set, we also conducted the experiments over other three popular data sets: recognition benchmark images [24], tiny images [30], and the patch data set [13]. We sampled 300k SIFT features from the recognition benchmark images for the search data base, and 10k SIFT features as the queries. The tiny image data set consists of 80 million images, introduced in [30]. The sizes of all the images in this database are 32×32. Similar to [15], we use a global Gist descriptor to represent each image, which is a 384-dimensional vector describing the texture within localized grid cells. We randomly sampled 200k images from the 1.5M images downloaded from the project Web page [1] to build the search database and sampled randomly another 20k as queries from the remaining images. The patch data set [13], associated with the Photo Tourism project [28], consists of local image patches of Flickr photos of various landmarks. The goal is to compute correspondences between local features across multiple images, which can then be provided to a structure-from-motion algorithm to generate 3D reconstruc-

[1] http://people.csail.mit.edu/torralba/tinyimages/

Method	Time Complexity		
	Build an internal node	Search an internal node	Search a leaf node
Kd-Tree	$O(k \times \bar{n})$	$O(1 + \log n_{queue})$	$O(k + \log n_{queue})$
Our approach	$O((k + d^2) \times \bar{n})$	$O(d + \log n_{queue})$	$O(k + \log n_{queue})$
PCA kd-Tree	$O(k \times \bar{n})$	$O(k + \log n_{queue})$	$O(k + \log n_{queue})$

Table 1. Time complexity comparison. d is the number of dominant coordinate axes in our method and is much smaller than the number of dimensions k for visual descriptors. The theoretic time complexity of building an internal node for PCA kd-tree seems smaller than ours, but in practice the construction of PCA kd-tree is slower than ours due to its larger constant.



(a) (b)

Figure 1. Illustrating that our approach with different numbers of coordinate axes to form the partition axis can produce good space partitioning and require less time for nearest neighbor search. (a) shows the number of accessed leaf nodes against the precision. (b) shows the query time against the precision.

tions of the photographed landmark [28]. Thus, one critical sub-task is to take an input patch and retrieve its corresponding patches within any other images in the database, which is essentially a large-scale similarity search problem. We use the 400k image patches from the Notre Dame Cathedral as the search data base and 60k image patches as queries.

Measurement criterion The goal of the experiments is to show that our approach outperforms the widely-used (the state-of-the-art) kd-tree in approximate nearest neighbor search. We also want to demonstrate how proposed schemes in our approach affect the performance. To evaluate the performance, we adopt the accuracy measurement to check whether the approximate nearest neighbor for each query is exactly the ground truth, the true nearest neighbor. In our experiments, we build the ground truth through the exhaustive linear scan.

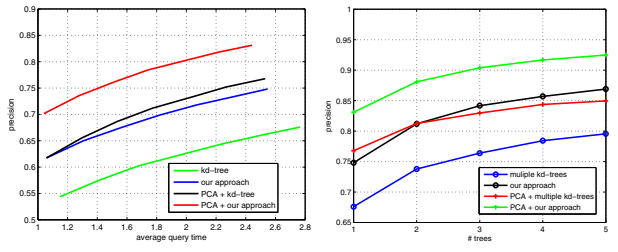
Dominant coordinate axes selection and space partitioning evaluation We conducted an experiment to show that our approach, using a few dominant coordinate axes to form the partition axis, can actually produce good space partitioning. This is done by comparing the number of accessed leaf nodes and the time cost between our approach and the PCA kd-tree, given the same query precisions. The PCA kd-tree has been shown to produce good space partitioning with good performance, given the limitation of the number of accessed leaf nodes. Due to the high complexity of building the PCA kd-tree for high dimensional data, we carried out this comparison in a small number of data points, 1K for the search data base, and 1K for the queries.

The comparison is shown in Fig. 1(a), in which the horizontal axis corresponds to the precision and the vertical axis corresponds to the number of the accessed leaf nodes. From this figure, we can have the following observations. Considering the precision reaches 0.95 (the high precision case is meaningful for the comparison here since the data set is small), our approach, with 5, 8, 11 dominant coordinate axes used, requires to access the similar number of leaf nodes with the PCA kd-tree. This result shows that our approach using a few number of dominant coordinate axes for each internal node can still produce good space partitioning. Importantly, our approach outperforms the PCA kd-tree about the query time. In Fig. 1(b), we show the curves of the average query time (milliseconds) against the query precision, where the horizontal axis represents the precision and the vertical axis represents the query time. It can be observed that our approach requires much less time to get the similar precision. Based on the above analysis, to build the partition axis for each internal node, our approach only requires a few dominant coordinate axes and can produce high precision with little query time.

5.1. Comparison with the state-of-the-art kd-trees

We compare the performance of our approach with the widely-used kd-tree for nearest neighbor search, and demonstrate the superiority of our approach, from the perspectives of query precision and query time. We first conducted the experiments on the Caltech 101 data set.

We start the comparison by considering the single tree case. The results are shown in Fig. 2(a), in which the horizontal axis is the average query time (milliseconds), and the vertical axis is the precision. From it, we can see that our approach gets significant improvement and obtains about 8% absolute improvement. Moreover, we also perform the the comparison over the data points preprocessed by rotation with PCA over the original data as [27]. The results are also shown in Fig. 2(a), from which it can be observed that they both outperform the methods without PCA as preprocessing. More importantly, our approach with PCA as preprocessing still gets 7% improvement. This is because the global principal eigenvector aligned partition axes is only helpful for the top few levels of the tree to find the



(a) (b)

Figure 2. (a) Comparison of the precision against the query time between our approach and the conventional kd-tree in the single tree case. We compare them over the original data space and the preprocessed data points, rotated with PCA. It can be observed that our approach performs better. (b) Illustration of the multiple tree version of our approach. We show the precision comparison against the number of trees between our approach and the conventional kd-tree over the original data space and the preprocessed data points, rotated with PCA. It can be observed that our approach consistently performs better in the multiple tree case.

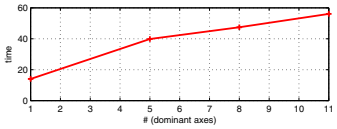


Figure 3. Comparison of construction time between our approach with different numbers of dominant axes and the conventional kd-tree (i.e., one dominant axis used).

good partition axes but less helpful for the deeper nodes because principal components of their associated subsets of data could alter dramatically from the global ones, and this is exactly where the proposed method shows its power.

We also demonstrate our approach in the multiple tree cases, which is expected to increase the performance. The comparison across different numbers of trees with and without using PCA processing is shown in Fig. 2(b), in which the horizontal axis corresponds to the number of trees, and the vertical axis corresponds to the precision. To make the comparison clear, we get the performances of those approaches with the same number of accessed leaf nodes. This is reasonable because the time cost of our approach is in reality even lower as we will show later. From this figure, it can be seen that that our approach is consistently better than the conventional kd-tree.

Fig. 3 is a comparison of time costs (seconds) between our approach and the conventional kd-tree. Consistent with the complexity analysis, our method takes more time in building a tree and the time increases linearly with the number of dominant coordinate axes used. But the tree constructions are done offline, and the amount of extra overheads is still acceptable. For the time costs of searching the trees, our method actually takes less time for visiting the same amount of leaf nodes as shown in Fig. 4. This is because a good space partition would reduce the efforts spent in backtracking, therefore making the search procedure visit fewer

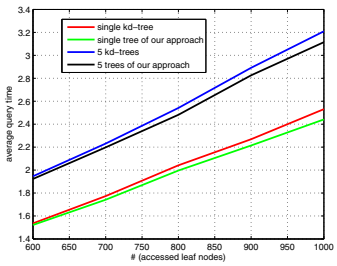


Figure 4. The query time comparison over the single tree case and the 5-tree case between our approach and the conventional kd-tree. We can see that our approach takes even less time than the conventional kd-tree when accessing the same number of leaf nodes.

internal nodes before reaching another leaf. In other words, the few extra operations for projections in our method are well compensated by better space partitions.

To avoid the bias of a single image data base, we also show the query precision under different data sets, recognition benchmark images [24], notredame [13], and tiny images [30] against the average query time (milliseconds). The results are shown in Fig. 5. We compare our approach and the conventional kd-tree under the single tree and multiple tree cases. The performance improvement over the conventional kd-tree further demonstrates that our approach outperforms the conventional kd-tree.

6. Conclusion

In this paper, we have presented a simple yet effective scheme to scale up kd-trees. We relax the coordinate-axis-alignment constraint for partition hyperplane selection in the conventional kd-tree and find a partition axis with the great variance by combining a few coordinate axes in a binary manner, in order to get better space partitioning and cheaper projection operations for deciding the order of visiting the nodes on the querying stage. On the other hand, the proposed scheme guarantees that the partition hyperplane of each internal node is orthogonal to or parallel with those of its ancestors, to compute the distance from a query point to a cell quickly for efficient maintenance of the priority queue. The analysis and experiments show that our approach is better than the state-of-the-art kd-trees.

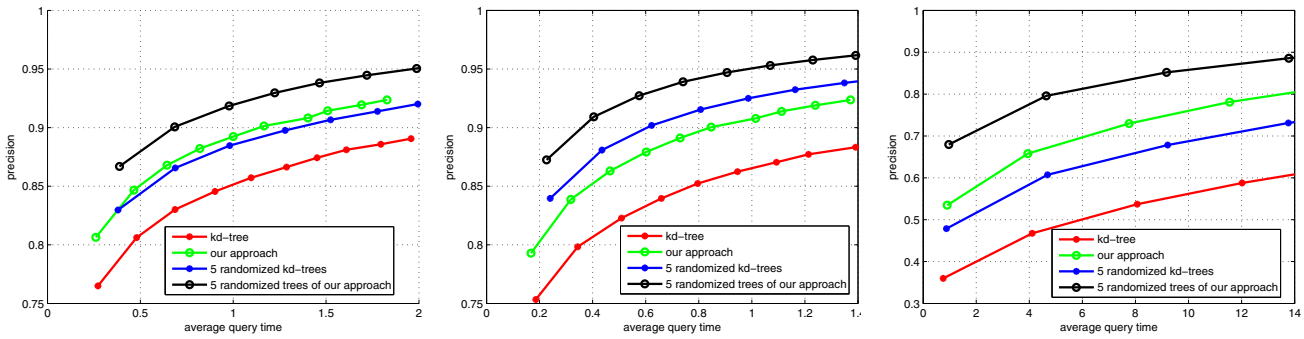
Acknowledgements

The research work of You Jia, Gang Zeng and Hongbin Zha is supported by NSFC Grant (No. 90920304) and NHTRDP 863 Grant (No. 2009AA012105).

References

[1] S. Arya and D. M. Mount. Algorithms for fast vector quantizaton. In *Data Compression Conference*, pages 381–390, 1993.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest



(a) (b) (c)

Figure 5. Performance comparison over (a) recognition benchmark images [24], (b) notredame [13], and (c) tiny images [30].

neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.

[3] J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *CVPR*, pages 1000–1006, 1997.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[5] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.

[6] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[7] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[8] M. Brown and D. G. Lowe. Recognising panoramas. In *ICCV*, pages 1218–1227, 2003.

[9] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, pages 253–262, 2004.

[10] M. de Berg, T. Eindhoven, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2008.

[11] L. Fei-Fei, R. Fergus, and P. Perona. Learning generative visual models from few training examples: an incremental bayesian approach tested on 101 object categories. In *CVPR 2004 Workshop on Generative-Model Based Vision*, 2004.

[12] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977.

[13] G. Hua, M. Brown, and S. A. J. Winder. Discriminant embedding for local image descriptors. In *ICCV*, pages 1–8, 2007.

[14] P. Jain, B. Kulis, and K. Grauman. Fast image search for learned metrics. In *CVPR*, 2008.

[15] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *ICCV*, 2009.

[16] N. Kumar, L. Zhang, and S. K. Nayar. What is a good nearest neighbors algorithm for finding similar patches in images? In *ECCV (2)*, pages 364–378, 2008.

[17] T. Liu, A. W. Moore, A. G. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *NIPS*, 2004.

[18] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.

[19] J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3):153–166, 1990.

[20] J. McNames. A fast nearest-neighbor algorithm based on a principal axis search tree. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(9):964–976, 2001.

[21] K. Mikolajczyk and J. Matas. Improving descriptors for fast tree matching by optimal linear projection. In *ICCV*, pages 1–8, 2007.

[22] A. W. Moore. The anchors hierarchy: Using the triangle inequality to survive high dimensional data. In *UAI*, pages 397–405, 2000.

[23] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISSAPP (1)*, pages 331–340, 2009.

[24] D. Nistér and H. Stewénius. Scalable recognition with a vocabulary tree. In *CVPR (2)*, pages 2161–2168, 2006.

[25] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *CVPR*, 2007.

[26] G. Shakhnarovich, T. Darrell, and P. Indyk. *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice*. The MIT press, 2006.

[27] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *CVPR*, 2008.

[28] N. Snavely, S. M. Seitz, and R. Szeliski. Photo tourism: exploring photo collections in 3D. *ACM Trans. Graph.*, 25(3):835–846, 2006.

[29] R. F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(4):579–589, 1991.

[30] A. B. Torralba, R. Fergus, and W. T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(11):1958–1970, 2008.

[31] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, pages 311–321, 1993.