Optimised KD-trees for fast image descriptor matching

Chanop Silpa-Anan
Seeing Machines, Canberra

Richard Hartley
Australian National University and NICTA.

Abstract

In this paper, we look at improving the KD*-tree for a specific usage: indexing a large number of* SIFT *and other types of image descriptors. We have extended priority search, to priority search among multiple trees. By creating multiple* KD*-trees from the same data set and simultaneously searching among these trees, we have improved the* KD*-tree's search performance significantly. We have also exploited the structure in* SIFT *descriptors (or structure in any data set) to reduce the time spent in backtracking. By using Principal Component Analysis to align the principal axes of the data with the coordinate axes, we have further increased the* KD*-tree's search performance.*

1. Introduction

Nearest neighbour search is an important component in many computer vision applications. In this paper, we look at the specific problem of matching image descriptors for an application such as image retrieval or recognition. We describe and analyze a search method based on using multiple randomized KD-trees, and achieve look-up results significantly superior to the priority KD-trees suggested by Lowe in [10] for this purpose. The randomized KD-tree approach described here has been used recently in [15] for feature lookup for a very large image recognition problem.

The SIFT descriptor (Scale invariant feature) [11] is of particular interest because it performs well compared with other types of image descriptors in the same class [13]. A SIFT descriptor is a 128-dimensional vector normalised to length one. It characterises a local image patch by capturing local gradients into a set of histograms which are then compacted into one descriptor vector. In a typical application, a large number of SIFT descriptors extracted from one or many images are stored in a database. A query usually involves finding the best matched descriptor vector(s) in the database to a SIFT descriptor extracted from a query image.

A useful data-structure for finding nearest-neighbour queries for image descriptors is the KD-tree [4], which is a form of balanced binary search tree.

Outline of KD-tree search. The general idea behind KD-trees is described now. The elements stored in the KD-tree are high-dimensional vectors in R^d. At the first level (root) of the tree, the data is split into two halves by a hyperplane orthogonal to a chosen dimension at a threshold value. Generally, this split is made at median in the dimension with the greatest variance in the data set. By comparing the query vector with the partitioning value, it is easy to determine to which half of the data the query vector belongs. Each of the two halves of the data is then recursively split in the same way to create a fully balanced binary tree. At the bottom of the tree, each tree node corresponds to a single point in the data set; though in some implementation, the leaf nodes may contain more than one point. The height of the tree will be $_2 N$ where N is the number of points in the data set.

Given a query vector, a descent down the tree requires $_2 N$ comparisons and leads to a single leaf node. The data point associated with this first node is the first candidate for the nearest neighbour. It is useful to remark, that each node in the tree corresponds to a cell in R^d, as shown in figure 1. And a search with a query point lying anywhere in a given leaf cell will lead to the same leaf node.

The first candidate will not necessarily be the nearest neighbour to the query vector; it must be followed by a process of backtracking, or iterative search, in which other cells are searched for better candidates. The recommended method is *priority search* [2, 4] in which the cells are searched in the order of their distance from the query point. This may be accomplished efficiently using a priority tree for ordering the cells; see figure 1 in which the cells are numbered in the order of their distance from the query vector. The search terminates when there are no more cells within the distance defined by the best point found so far. Note that the priority queue is a dynamic structure that is built while the tree is being searched.

In high dimensions to find the nearest neighbour may require searching a very large number of nodes. This problem may be overcome at the expense of an approximate answer by terminating the search after a specified number of nodes are searched (or earlier if the search terminates).

[1]This research was partly supported by NICTA, a research centre funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

978-1-4244-2243-2/08/$25.00 ©2008 IEEE

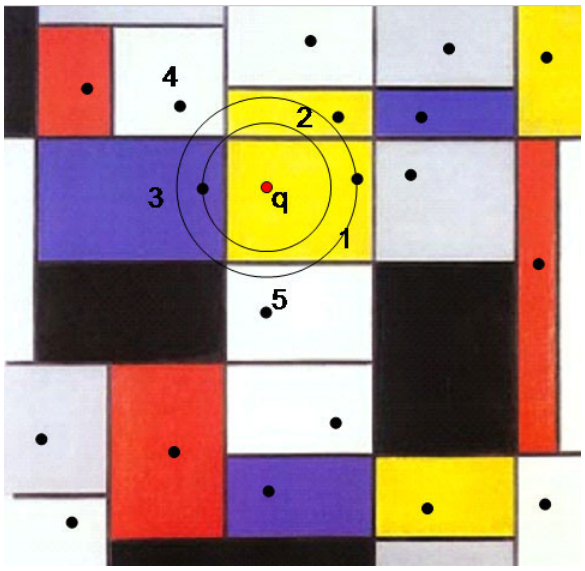

Figure 1. **[Priority search of a KD-tree]** *In this figure, a query point is represented by the red dot and its closest neighbour lies in cell 3. A priority search first descends the tree and finds the cell that contains the query point as the first candidate (label 1). However, a point contained in this cell is often not the closest neighbour. A* priority search *proceeds in order through other nodes of the tree in order of their distance from the query point – that is, in this example through the nodes labelled 2 to 5. The search is bounded by a hypersphere of radius r (the distance between the query point and the best candidate). The radius r is adjusted when a better candidate is found. When there are no more cells within this radius, the search terminates.* KD*-tree diagram thanks to P. Mondrian.*

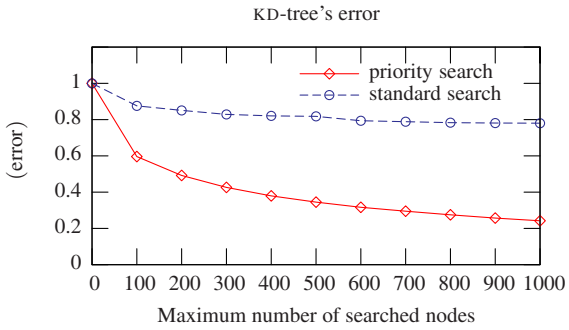

Figure 2. **[Search performance]** *When a search is restricted to some maximum number of searched nodes, the probability of finding the true nearest neighbour increases with the increasing limit. Priority search increases search performance, compared with a tree backtracking search.*

This graph and subsequence graphs are made by searching SIFT *descriptors from a data set of size approximately 500 000 points. A query is a descriptor drawn from the data set and corrupted with Gaussian noise. A nearest neighbour query result is compared with the true nearest match.*

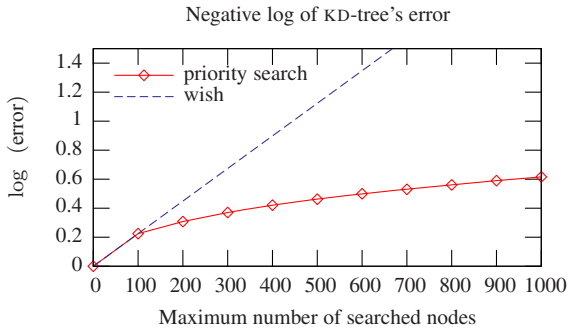

Figure 3. **[Diminished returns]** *This figure is essentially the same as figure 2 (lower curve), but on a negative logarithmic scale. Suppose each search of m nodes is independent and has a failure probability p_e. By searching n nodes the error rate reduces to ${p_e}^{n/m}$. On a logarithmic scale, this is a straight line with a slope of $(n/m)\ \log p_e$. The figure shows that increasing the number of searched nodes for* KD*-tree does not lead to independent searches, and gives diminished returns.*

The best candidate may be the exact nearst neighbour with some acceptable probability that increases as more nodes are searched (see figure 2).

Unfortunately, extending the search to more and more nodes leads to diminishing returns, in which we have to work harder and harder to increase the probability of finding the nearest neighbour. This point is illustrated in figure 3. The main purpose of this paper is to propose methods of avoiding this problem of diminishing returns by carrying out simultaneous independent searches in different trees.

The problem with diminishing returns in priority search is that searches of the individual nodes in a tree are not independent, and the more searched nodes, the further away the nodes are from the node that contain the query point. To address this problem, we investigate the following strategies.

1. We create m different KD-trees each with a different structure in such a way that searches in the different trees will be (largely) independent.
2. With a limit of n nodes to be searched, we break the search into simultaneous searches among all the m trees. On the average, n/m nodes will be searched in each of the trees.
3. We use Principal Component Analysis (PCA) to rotate the data to align its moment axes with the coordinate axes. Data will then be split up in the tree by hyperplanes perpendicular to the principal axes.

By either using multiple search-trees or by building the KD-tree from data realigned according to its principal axes, search performance improves and even improves further when both techniques are used together.

Previous work

The KD-tree was introduced in [5] as a generalisation of a binary tree to high dimensions. Several variations in building a tree, including randomisation in selecting the partitioning value, were suggested. Later on, an optimised KD-tree with a theoretic logarithmic search-time was proposed in [7]. However, this logarithmic search-time does not apply to trees of high dimension, where the search time may become almost linear.

Thus, although KD-trees are effective and efficient in low dimensions, their efficiency diminishes for high-dimensional data. This is because with high dimensional data, 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 absolute minimum is sacrificed and replaced with a probabilistic performance. Recent research has therefore aimed at increasing the probability of success while keeping backtracking within reasonable limits. Two similar approximated search methods, a best-bin-first search and a priority search were proposed in [2, 4], and these methods have been used with significant success in object recognition ([10]).

In the vision community, interest in large scale nearest-neighbour search has increased recently, because of its evident importance in object recognition as a means of looking up feature points, such as SIFT features in a database. Notable work in this area has been [14, 15]. It was reported in [15] that KD-trees gave better performance than Nister's vocabulary trees [14] as an aid to K-means clustering. In this paper therefore we concentrate on methods of improving the performance of KD-trees, and demonstrating significant improvements in this technique.

Our KD-tree method is related to randomized trees, as used in [9], based on earlier work in [1]. However, our methods are not directly comparable with theirs, since we address different problems. Randomized trees are used in [9] for direct recognition, whereas we are concerned with the more geometric problem of nearest-neighbour search. Similarly, more recent work of Grauman [8] uses random hashing techniques for matching sets of features, but their algorithm is not directly comparable with ours. Finally, locality-sensitive hashing (LSH) [6] is based on similar techniques as ours, namely projection of the data in different random directions. Whereas LSH projects the data onto different lines, in our case we consider randomly reorienting the data, which is related to projection onto differently oriented linear subspaces. Like KD-trees, LSHs also have trouble dealing with very high dimensional data, and have not been used extensively in computer vision applications.

2. Independent multiple searches

When examining priority search results in a linear or a logarithmic scale (figure 2 and 3), it is clear that increasing the number of searched nodes increases the probability of finding the true nearest neighbour. Any newly examined cell, however, depends on previously examined cells. Suppose each search is independent with a failure probability p_e. Searching independently n times reduces the failure probability to p_e^n. This is a straight line on a logarithmic scale. It is clear that searches of successive nodes of the tree are not independent, and searches of more and more nodes become less and less productive.

On the other hand, if KD-trees are built with different parameters, with different ways to select partitioning value for example, the order of search node and search results on these KD-trees may be different. This leads to the idea of using multiple search-trees to enhance independence.

Rotating the tree. Our method of doing independent multiple searches is to create multiple KD-trees with different orientations. Suppose we have a data set $\mathcal{X}$ $\{\mathbf{x}_i\}$. Creating KD-trees with different orientations simply means creating KD-trees from rotated data $\mathtt{R}\mathbf{x}_i$, where $\mathtt{R}$ is a rotation matrix. A principal (a regular) KD-tree is one created without any rotation, $\mathtt{R}$ $\mathtt{I}$. By rotating the data set, the resulting KD-tree has a different structure and covers a different set of dimensions compared with the principal tree. An algorithm for searching on a rotated KD-tree is essentially searching a rotated tree with a rotated query point $\mathtt{R}\mathbf{q}$.

Once a rotated tree is built, the rotated data set can be discarded because all the information is kept inside the tree and the rotation matrix. Only the original vector is needed in order to compute the distance between a point in the data set and a query. Under a rotation, the original distance $\mathbf{q} - \mathbf{x}_i$ is the same as the rotated distance $\mathtt{R}\mathbf{q} - \mathtt{R}\mathbf{x}_i$.

Are rotated trees independent? To test whether searches of differently rotated trees were independent, we carried out successive searches over the same data set successively on multiple trees. First, we searched the 20 closest nodes in one tree, then the closest 20 on the next tree, and so on, to a total of 200 nodes searched. The total cumulative (empirical) probability of failure was computed and plotted on a negative logarithmic scale. If the searches on different trees were independent, this would give a linear graph. The results are shown in figure 4, and support the hypothesis that the searches are independent. This test was done with randomly generated vectors. Tests done also with real SIFT features showed major improvements of using multiple search trees, compared with a single tree, as reported in figure 6. However in that case the hypothesis of independent searches was not quite so strongly sustained. Nevertheless, using multiple trees goes a long way towards enabling independent

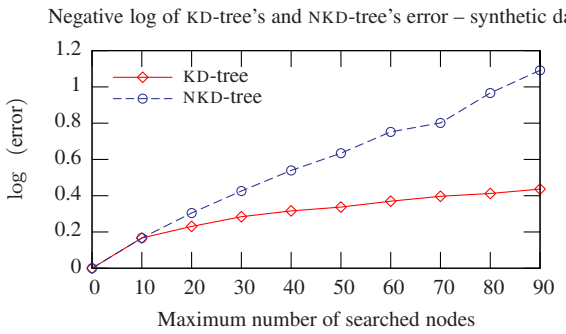

Figure 4. **[Independence of searches]** *This graph shows the result of successive independent searches of 20 nodes on each of 9 trees, showing the (empirical) negative log-likelihood of error. If the searches on individual trees are independent, then this will be a straight-line through the origin. Visibly, the graph is approximately linear, which supports a hypothesis of independence. For comparison we show the results for a single* KD*-tree as well.*

searches, even if they are not completely independent.

A saving using Householder matrices. Computation of $\mathtt{Rx}$ in d dimensions has complexity O d^2 . The underlying idea of using a rotation matrix is to transform the data set onto different bases while preserving the norm. Almost any orthogonal transformation matrix can achieve this. A Householder matrix of the form $\mathtt{H_v}$ $\mathtt{I} -$ $\mathtt{vv}$ $/\mathtt{v}$ $\mathtt{v}$ is an orthonormal transformation; it is a reflection through a plane with normal vector $\mathbf{v}$. Multiplication of a vector and $\mathtt{H_v}$ can be arranged such that it has complexity of O d instead of O d^2 . With the Householder transformation, m trees can be built in O mdN N time.

Searching multiple trees. With multiple trees, we need to expand the concept of a priority search on a single tree. Conceptually, searching m trees with a limitation of n search nodes is simply searching each tree for n/m nodes. This can be easily implemented by searching trees sequentially. However, this is not optimal, and besides does not scale to a case where we impose no limitation on the number of searched nodes (searching for the true nearest neighbour) because we would already find the best solution and would not be required to search extra trees.

We prefer to search 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-neighbour candidate, we select the best one from all the trees. We then pool the node ranking by using one queue to order the nodes from all m trees. In this way, nodes are not only ranked against other nodes within the same tree, but also ranked against other nodes in all trees. As a result, nodes from all the trees are searched in the order of their distance from the query point simultaneously.

With these modifications, we use the term NKD-tree for a data structure of multiple KD-trees with different orientations.

Space requirements. For large data sets it is important to consider the space requirements for holding large numbers of trees. We have implemented our KD-trees as pointerless trees, in which the nodes are kept in a linear array. The two children of node n are the nodes at positions n and n in the array. Only the number of the splitting dimension (one byte) and the splitting value (one byte for SIFT descriptors containing single byte data, or bytes for floating point data) need to be stored at internal nodes in the tree. The leaf nodes must contain an index to the associated point. In total this means N bytes for a tree with N elements. In addition, the actual data vectors must be stored, but only once. Thus, with one million data vectors of dimension 128, the storage requirement is 128MB for the data vectors (or 512MB if float data is used), and for each independent tree only 6MB. Thus, the storage overhead for having multiple trees is minimal.

A different randomisation: RKD-tree. The purpose of the rotation is to create KD-trees with different structures. Instead of explicitly rotating the tree, using randomness on parameters can also alter the tree structure. In fact, the partitioning value was originally selected randomly [5] while the partitioning dimension was selected in cyclical order. This approach was later on abandoned in an optimal KD-tree [7] and in other splitting rules [3].

In accordance with the principle of selecting a partitioning value in the dimension with the greatest variance, we considered creating extra search trees with the following idea. In the standard KD-tree, the dimension which the data is divided is the one in which the data has the greatest variance. In reality, data variance is quite similar in many of the dimensions, and it does not make a lot of difference in which of these dimensions the subdivision is made. We adopt the strategy of selecting at random (at each level of the tree) the dimension in which to subdivide the data. The choice is made from among a few dimensions in which the data has high variance. Multiple trees are constructed in this way, different from each other in the choice of subdivision dimensions. By using this method, we retain high probability that a query can be on either half of the node, and at the same time maintain backtracking efficiency.

In this randomisation strategy, in contrast to rotating the data explicitly, the data set stays in the original space, thus, saving some computation on data projection while building the tree. In searching SIFT descriptors, this randomised tree (RKD-tree) performs as well as NKD-tree. Also note that the RKD-tree has the same complexity level in storage as the NKD-tree.

3. Modelling KD-trees.

The following argument is meant to give an intuitive idea of why using differently rotated trees will be effective. It is difficult to model the performance of a KD-tree mathematically, because of the the complexity of the priority search process. In this section, it will be argued that the performance of a KD-tree is well modelled by projecting the data into a lower dimensional space followed by testing the data in the order of its distance from a projected query point. It will be seen that modelling a KD-tree's performance in this way gives rise to performance graphs that strongly resemble the actual performance of KD-trees on SIFT data.

Consider a KD-tree in a high-dimensional space, such as dimension for SIFT features. If the tree contains one million nodes, then it has height . During a search with a query point in the tree, no more than of the of the entries of the vector are considered. The other entries are irrelevant for the purposes of determining which cell the search ends up in. If $\mathbf{q}$ is the query vector and $\mathbf{x}$ is the vector associated with the leaf cell that the query vector leads to, then $\mathbf{q}$ and $\mathbf{x}$ are close in of their entries, but maybe no others.

The virtual projection. Typically only a small number n of the data dimensions will be used to partition the tree. The other $- n$ dimensions will be unused, and irrelevant. Exactly the same tree will be obtained if the data is first projected by a projection π onto a subspace of dimension n before building the tree. The subspace is aligned with the coordinate axes. Since no actual projection takes place, we refer to this as a *virtual projection* – but since such a projection would make no difference to the method of searching the KD-tree we may assume that this projection takes place, and the tree is built using the projected data.

Under priority search in the KD-tree, leaf nodes in the tree are searched in the order of the distance of the corresponding cells from the projected query point $\pi\ \mathbf{q}$. The points associated with the cells are tested in that order to find the one that is closest to $\mathbf{q}$. Since the structure of the KD-tree may be complex, we simplify the analysis by make a simplifying assumption that searching the cells in this order is the same as testing the points $\mathbf{x}_i$ in the order of the distance of their projection $\pi\ \mathbf{x}_i$ from the projected query point $\pi\ \mathbf{q}$. Since the tree is virtually built from the projected data, this is the best search outcome that can be achieved. Thus it provides a best-possible performance for search in the KD-tree.

To summarise this discussion: in our model, the order in which points $\mathbf{x}_i$ are tested to find the closest match to $\mathbf{q}$ is the order of the distance of $\pi\ \mathbf{x}_i$ from $\pi\ \mathbf{q}$ where π is a projection onto a lower dimensional space.

Probability analysis. We have argued that KD-tree search in high dimensions is closely related to nearest-neighbour search after projection into a lower dimensional space. The essential question here is whether the nearest neighbour to a query point in the high dimension will remain an approximate nearest neighbour after projection. Consider a large number of points $\mathbf{x}$ in a high-dimensional space, suppose $\mathbf{q}$ is a query point and $\mathbf{x}_{\text{best}}$ is the closest point to it. Now, let all points be projected to a lower dimensional space by a projection π, and denote by $p\ n$ the probability that $\pi\ \mathbf{x}_{\text{best}}$ is the n-th closest point to $\pi\ \mathbf{q}$. We would expect that the most likely outcome is that $\pi\ \mathbf{x}_{\text{best}}$ is in fact the closest point to $\pi\ \mathbf{q}$, but this is not certain. In fact, it does not happen with high probability, as will be seen. It may be possible to compute this probability directly under some assumptions. However, instead we computed the form of this probability distribution empirically; the resulting probability distribution function is shown in figure 5. The most notable feature is that this distribution has a very long tail. Also shown is the cumulative probability of failure after testing m nodes, given by $f\ m \quad - \sum_{i=1}^{m} p\ i$.

The simulation of figure 5 shows that the nearest-neighbour point to the query may be a very long way down the list of best of closest matches when projected to low dimension. Consequently the probability of failure $f\ m$ remains relatively high even for large m. This indicates why searching the list of closest neighbours in the KD-tree may require a very large number of cells to be searched before the minimum is found.

On the other hand, the strategy of using several independent projections may give better results. If the data is rotated, then the virtual projection to a new n-dimensional subspace will be in a different direction aligned with the new coordinate axes. If $\mathbf{q}$ and $\mathbf{x}$ are close to each other in the full space, then they will remain close under any projection, and hence will belong to the same or neighbouring leaf cells in any of the rotated KD-trees. On the other hand, points that are not close may belong to neighbouring leaf cells under one projection, but are unlikely to be close under a different projection. We model the probability of failure using k trees as the product of the independent probabilities of failure from searching approximately m/k cells in each of the k trees. More exactly, if $a \quad m \bmod k$ and $m_i \quad m - a\ /k$, then the probability of failure when searching m_i cells in $k - a$ trees and m_i cells in a trees (a total of m cells in all) is $f_{m,k} \quad f\ m_i\ ^{k\ a} f\ m_i \quad ^{a}$. The graphs of these probabilities are shown in figure 5 for several different values of k. They clearly show the advantage of searching in several independent projections. Using independent projections into lower dimensional spaces boosts the probability that the closest point will be among the closest points in at least one of the projections. The results obtained from this model agree quite closely in form with the results obtained using independent KD-trees. This supports our thesis that closest point search in high dimensional KD-trees is closely related to nearest neighbour search in a low dimensional projection,

and that the long tail of the probability distribution $p\ n$ is a major reason why searching in a single KD-tree will fail, whereas using several KD-trees with independently rotated data will work much better.

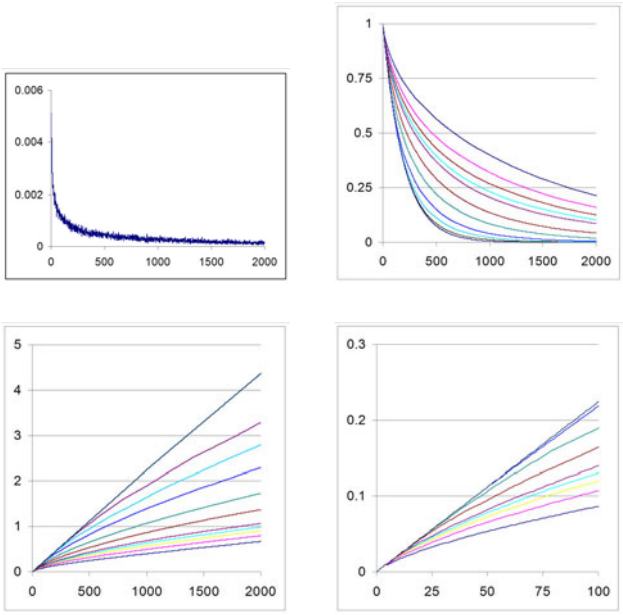

Figure 5. **Top left:** *Distribution of rank of best fit after projection to low dimensions.* $100,000$ *random points in a hypercube in* R^{128} *are projected by a projection* π *into* R^{20}*. The nearest point* $\mathbf{x}_{\text{best}}$ *to a query* $\mathbf{q}$ *in* R^{128} *may become* n*-th closest after projection onto* R^{20}*. The* x*-axis is the ranking* n *and the* y*-axis shows the probability that the projection* $\pi(\mathbf{x}_{\text{best}})$ *will be* n*-th closest point to* $\pi(\mathbf{q})$ *in* R^{20}*. The graph shows that the most likely ranking is first, but the graph has a long tail.*
Top right: *Top line shows the cumulative failure rate. It represents the probability of failure to find the best fit in* R^{128} *by examining the* n *best fits in* R^{20}*. . Subsequent graphs show the probability of finding the best fit by examining the closest* n *fits in* m *different independent projections for* $m = 2, 3, 4, 5, 10, 20, 50, 100, 200, 1000$*. As may be seen, the more independent projections are used, the better the chance of finding the best fit* $\mathbf{x}_{\text{best}}$ *in* R^{128} *among the* n *points tested.*
Bottom left: *Negative log likelihood of cumulative failure rate for the same number of projections as above. If each independent query (point tested) has independent probability of being the optimum point* $\mathbf{x}_{\text{best}}$*, the lines will be straight.* **Bottom right:** *Close up of the previous graph showing curvature of the graphs, and hence diminished returns from examining more and more projected points.*

4. Principal Component Trees

It has been suggested in the discussion of section 3 that one of the main reasons that KD-trees perform badly in high dimensions is that points far from the query point $\mathbf{q}$ may be projected (via a virtual projection) to points close to the best match $\mathbf{x}_{\text{best}}$. It makes sense therefore to minimize this effect to project the data in its narrowest direction. Since in a KD-tree the data is divided up in the directions of the coordinate axes, it makes sense to align these axes with the principal axes of the data set. Aligining data with principal axes has been considered previously in, for instance [12]. We show in this section that by combining this idea with the KD-tree data structure gives even better results.

To be precise, let $\{\mathbf{x}_i | i \quad , \ldots, N\}$ be a set of points in R^d. We begin by translating the data set so that its centroid is at the origin. This being done, we now let $\mathtt{A} \quad \sum_{i=1}^{N} \mathbf{x}_i \mathbf{x}_i \quad .$ The eigenvectors of $\mathtt{A}$ are the principal axes of the data set, and the eigenvalues are referred to as the principal moments. If $\mathtt{A} \quad \mathtt{U} \ \mathtt{U}$ is the eigenvalue decomposition of $\mathtt{A}$, such that the columns of $\mathtt{U}$ are the (orthogonal) eigenvectors, then the mapping $\mathbf{x}_i \mapsto \mathtt{U} \ \mathbf{x}_i$ maps the points onto a set for which the principal axes are aligned with the coordinate axes. Furthermore, if $\mathtt{U}_{1:k}$ is the matrix consisting of the k dominant eigenvectors, then $\mathtt{U}_{1:k}$ projects the point set into the space spanned by the k principal axes of the data.

Generally speaking if the data is aligned via the rotation $\mathtt{U}$, the dimensions along which data is divided in the KD-tree will be chosen from among the k principal axes of the data, where k is the depth of the tree (though this will not hold strictly, especially near the leaves of the tree). The effect will be the same as if the data were projected via the projection $\mathtt{U}_{1:k}$ and the projected data were used to build the tree. In a sense the data is *thinnest* in the directions of the smallest principal axes, and projection onto the space of the k dominant axes will involve the smallest possible change in the data, and will minimize the possibility of distant points projecting to points closer to $\mathbf{x}_{\text{best}}$ than $\pi \ \mathbf{q}$.

We therefore suggest the following modifications to the KD-tree algorithm.

1. Before building the tree, the data points $\mathbf{x}_i$ should be rotated via the mapping $\mathtt{U}$ to align the principal axes with the coordinate axes.
2. When using multiple trees, the rotation of the data should be chosen to preserve the subspace spanned by the k largest principal axes.

The resulting version of the KD-tree algorithm will be called the PKD-tree algorithm. The second conditions is reasonable, since it makes no sense to rotate the data to align it with the coordinate axes and then unalign it using random rotations while creating multiple trees.

Almost equivalent is to project the data to a lower dimensional subspace using the PCA projection $\mathtt{U}_{1:k}$, and then use this projected data to build multiple trees. (The only difference is that in this way, the splitting dimension is strictly constrained to the top k principal axis directions.) An important point which we emphasize is that the original unprojected data must still be used in testing the distance between the query point $\mathbf{q}$ and candidate closest neighbours. The ef-

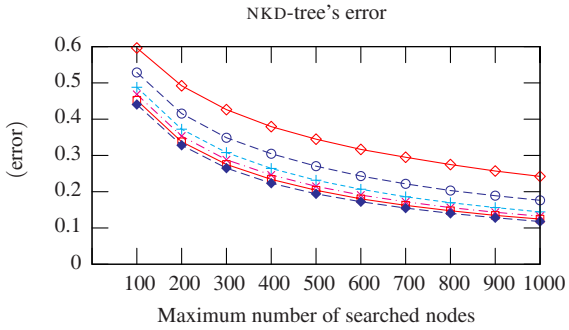

Figure 6. **[An NKD-tree]** *This figure shows some search results using an* NKD*-tree. From the top-most graph to the bottom-most graph are* NKD*-trees using from 1 to 6 search trees. The* NKD*-tree's performance increases with the increasing number of searched nodes and the increasing number of trees.*

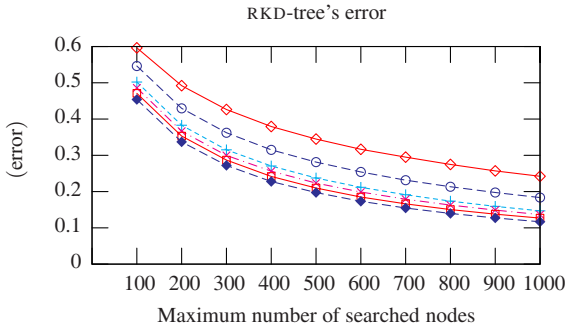

Figure 7. **[An RKD-tree]** *In a similar setting to that of figure 6, the* RKD*-tree performs as well as the* NKD*-tree in searching* SIFT *descriptors.*

fect of projecting the data is only to constrain and guide the structure and alignment of the KD-trees. The projected data is discarded once the tree is built.

5. Experimental results

For comparison, we use a data set of approximately 500 000 SIFT descriptors, computed from 600 images. Of these descriptors, 20 000 descriptors are randomly picked for nearest neighbour queries; however, they are corrupted with some small Gaussian noise with standard deviation 0.05 in all 128 dimensions. Note that SIFT descriptors are normalised to length 1 including ones used for queries; therefore the expected norm of the distance from the original to the corrupted point is 0.5.

Figure 6 shows search results with an NKD-tree. The number of search trees ranges from one to six trees; the limitation of searched nodes ranges from 100 to 1000 nodes. Clearly, the error reduces when we search more nodes. At the same number of search nodes, using more search trees also reduces the error. In a similar setting, an RKD-tree performs as well as an NKD-tree; figure 7 shows these results.

For a PKD-tree, figure 8 shows results when the data is aligned using PCA, but multiple search trees are built with unconstrained rotations. In a single search tree case, there is a significant improvement over an ordinary KD-tree (compare the top most lines between figure 6 and figure 8). The

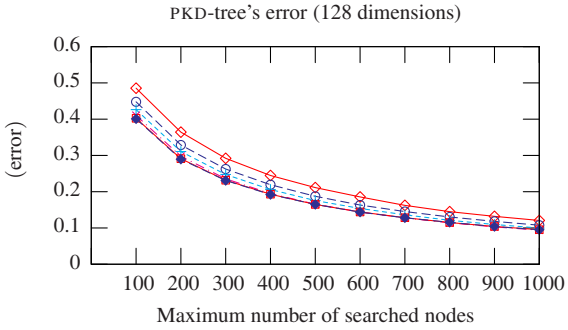

Figure 8. **[Multiple PKD-tree trees with unconstrained rotation]** *The* PKD*-tree with full 128 dimensions performs better than both* NKD*-tree and* RKD*-tree. There is a significant improvement over a* KD*-tree when a single search tree is used. For multiple search trees, we allow the rotations of the data to be arbitrary. The performance continues to improve but is not so marked, because the rotation of the data undoes the* PCA *alignment.*

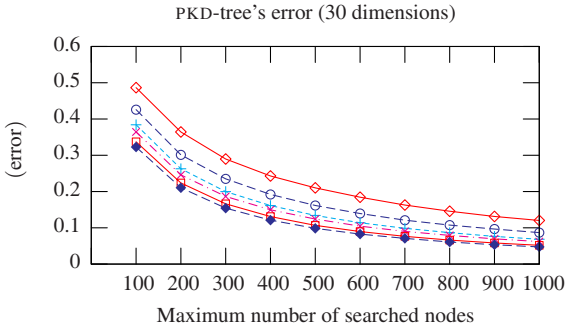

Figure 9. **[A PKD-tree created with 30 dimensions]** *The* PKD*-tree's performance is at its best when rotations are constrained to the optimal number of dimensions. This figure shows the* PKD*-tree's results when the data is projected by a projection* $U_{1:k}$ *using* PCA*onto 30 dimensions. This constrains the rotations used to build multiple trees so that they preserve the space spanned by the* $k = 30$ *principal axes. There are significant improvements compared with the results shown in figure 8.*

PKD-tree still performs better with an increasing number of trees, however with a smaller increment.

We next explored the strategy of aligning the data using PCA and then building multiple trees using rotations that fix (not pointwise) the space spanned by the top k principal axes. Our experiments showed that is the optimal number for this purpose. In figure 9, we simply project the data onto a subspace of dimension using PCA, and then allow arbitrary rotations of this projected data. It is a clear that using multiple search trees with the trees built from an optimal number of dimensions improves the result. There is a significant difference when six search trees are used.

This form of PKD-tree with multiple search trees gives a very substantial improvement over the standard single KD-tree. By comparing the top line of figure 6 (single standard KD-tree) with the bottom line of figure 9 (PKD-tree with 6 trees), we see that the same search success rate achieved in 1000 nodes with the standard tree is reached in about 150 nodes by the PKD-tree. Looked at a different way, after searching 1000 nodes, the standard KD-tree has about 75% success rate, whereas the multiple PKD-tree has over 95% success.

6. Conclusions

In the context of nearest neighbour query in a high dimensional space with a structured data set – SIFT descriptors in 128 dimensions in our application – we have demonstrated that various randomisation techniques give enormous improvements to the performance of the KD-tree algorithm.

The basic technique of randomisation is to carry out simultaneous searches using several trees, each one constructed using a randomly rotated (or more precisely, reflected) data set. This technique can lead to an improvement from about 75% to 88% in successful search rate, or a 3-times search speed-up with the same performance.

Best results are obtained by combining this technique with a rotation of the data set to align it with its principal axis directions using PCA, and then applying random Householder transformations that preserve the PCA subspace of appropriate dimension (d in our trials). This leads to a success rate in excess of 95%.

Tests with synthetic high-dimensional data (not reported in detail here) led to even more dramatic improvements, with up to 7-times diminished error rate with the NKD-tree algorithm alone.

References

[1] Y. Amit and D. Geman. Shape quantization and recognition with randomized trees. *Neural Quantization*, 9(7):1545 – 1588, 1997. 3

[2] S. Arya and D. M. Mount. Algorithms for fast vector quantization. In *Proceedings of the data compression conference*, pages 381 – 390, 1993. 1, 3

[3] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the* ACM, 45(6):891–923, 1998. 4

[4] Jeffrey S. Beis and David G. Lowe. Shape indexing using approximate nearest–neighbour search in high–dimensional spaces. In *Proceedings of computer vision and pattern recognition*, pages 1000–1006, Puerto Rico, June 1997. 1, 3

[5] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the* ACM, 18(9):509–517, 1975. 3, 4

[6] M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the symposium on computational geometry*, pages 253–262, 2004. 3

[7] Jerome H. Freidman, Jon Louis Bently, and Raphael Arifinkel. An algorithm for finding best matches in logarithmic expected time. ACM *transactions on mathematical software*, 3(3):209–206, september 1977. 3, 4

[8] K. Grauman and T. Darrell. Pyramid match hashing: Sublinear time indexing over partial correspondences. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition*, 2007. 3

[9] V. Lepetit and P. Fua. Keypoint recognition using randomized trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(9):1465–1479, 2006. 3

[10] D. Lowe. Distinctive image features from scale invariant keypoints. *IJCV*, 60(2):91 – 110, 2004. 1, 3

[11] David G. Lowe. Distictive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004. 1

[12] J. McNames. A fast nearest-neighbor algorithm based on a principal axis search tree. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23:964–976, September 201. 6

[13] Krystian Mikolajczyk and Cordelia Schmid. A performance evaluation of local descriptors. In *Proceedings of computer vision and pattern recognition*, 2003. 1

[14] D. Nister and H. Stewenius. Scalable recognition with a vocabulary tree. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition*, volume 2, pages 2161 – 2168, 2006. 3

[15] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition*, 2007. 1, 3