Approximate Nearest Neighbor Search

Yannis Kalantidis

Image, Video and Multimedia Systems Laboratory
National Technical University of Athens

September 2010
Outline

KD-Trees

Best Bin First - BBF

Randomized KD-Trees

k-Means Tree

Best Bin First k-Means Tree

Greedy N-Best Paths Search
Outline

KD-Trees

Best Bin First - BBF

Randomized KD-Trees

k-Means Tree

Best Bin First k-Means Tree

Greedy N-Best Paths Search
KD-Trees

KD-Tree - Exact Nearest Neighbors
**Construction** ($O(N \log^2 N)$):

- Cycle through data dimension to select splitting plane (modification: Split on the dimension of greatest variance)
- Split at medoid (to have a balanced tree$^1$)
- Continue recursively until both sides of the splitting plane are empty (Height: $\log_2 N$)

---

$^1$Speed-up: Select $M < N$ random points and get their median
KD-Tree – Nearest Neighbor Search Example

Algorithm/

Constructing a \(k_d\)-tree

Input/: \(\text{exset} /\), of type exemplar-set

Output/: \(\text{kd} /\), of type kdtree

Pre/: None

Post/: \(\text{exset} /\) = \(\text{exset-rep} /\) (\(\text{kd} /\))

Code/

1. If \(\text{exset} /\) is empty then return the empty kdtree

2. Call pivot-choosing procedure, which returns two values:
   - \(\text{ex} /\) = a member of \(\text{exset} /\)
   - \(\text{split} /\) = the splitting dimension

3. \(\text{d} /\) = domain vector of \(\text{ex} /\)

4. \(\text{exset}' /\) = \(\text{exset} /\) with \(\text{ex} /\) removed

5. \(\text{r} /\) = range vector of \(\text{ex} /\)

6. \(\text{exsetleft} /\) = \(\text{f} /\) (\(\text{d} /0 /;;\) \(\text{r} /0 /;;\) \(\text{split} /\) / \(\text{kdleft} /\) /)

7. \(\text{exsetright} /\) = \(\text{f} /\) (\(\text{d} /0 /;;\) \(\text{r} /0 /;;\) \(\text{split} /\) / \(\text{kdrightright} /\) /)

8. \(\text{kdleft} /\) = recursively construct \(k_d\)-tree from \(\text{exsetleft} /\)

9. \(\text{kdrightright} /\) = recursively construct \(k_d\)-tree from \(\text{exsetright} /\)

Pro of/

By induction on the length of \(\text{exset} /\) and the definitions of \(\text{exset-rep} /\) and \(\text{Is-legal-kdtree} /\).

Table 6.3: Constructing a \(k_d\)-tree from a set of exemplars.
KD-Tree – Nearest Neighbor Search Example

Before discussing its execution, I will explain how the operations on the hyperrectangles can be implemented. A hyperrectangle is represented by two arrays: one of its minimum coordinates, the other of its maximum coordinates. To cut the hyperrectangle, so that one of its edges is moved closer to its center, the appropriate array component is altered. To check to see if a hyperrectangle \( hr \) intersects with a hypersphere radius \( r \) centered at point \( t \), we find the point \( p \) in \( hr \) which is closest to \( t \). Write \( hr_{\text{min}}^i \) as the minimum extreme of \( hr \) in the \( i \)th dimension and \( hr_{\text{max}}^i \) as the maximum extreme.

\[
    p_i = \begin{cases} 
        hr_{\text{min}}^i & \text{if } t_i > hr_{\text{min}}^i \\
        t_i & \text{if } hr_{\text{min}}^i < t_i < hr_{\text{max}}^i \\
        hr_{\text{max}}^i & \text{if } t_i < hr_{\text{min}}^i 
    \end{cases}
\]

The objects intersect only if the distance between \( p \) and \( t \) is less than or equal to \( r \).

The search is depth-first, and uses the heuristic of searching first the child node which contains the target. Step 1 deals with the trivial empty tree case, and Steps 2 and 3 assign two important local variables. Step 4 cuts the current hyperrectangle into the two hyperrectangles covering the space occupied by the child nodes. Steps 5/6/7 determine which child contains the target. After Step 8, when this initial child is searched, it may be possible to prove that there cannot be any closer point in the hyperrectangle of the further child. In particular, the point at the current node must be out of range. The test is made in Steps 9 and 10. Step 9 restricts the maximum radius in which any possible closer point could lie, and then the test in Step 10 checks whether there is any...
KD-Tree – Nearest Neighbor Search Example

Generally during a nearest neighbor search only a few leaf nodes need to be inspected.

A bad distribution which forces almost all nodes to be inspected.
KD-Tree’s Major Problem

High dimensional data:

- General rule (Wikipedia): for $k$ dimensions and $N$ data, you need $N >> 2^k$, otherwise most of the points in the tree will be evaluated (there are many adjacent bins)
- Approximate solutions are needed
- The backtracking search is still inefficient because the order of examining leaf nodes is according to the tree structure, which depends only on the stored points, and does not take into account the position of the query point.
Outline

KD-Trees

Best Bin First - BBF

Randomized KD-Trees

k-Means Tree

Best Bin First k-Means Tree

Greedy N-Best Paths Search
Best Bin First - BBF

Shape Indexing Using Approximate Nearest-Neighbour Search in High-Dimensional Spaces

Beis, Lowe – CVPR 1997
Simple idea:

- Look in bins in order of increasing distance from the query point. The distance to a bin is defined to be the minimum distance between the query and any point on the bin boundary.
- Such a search can be easily implemented using a priority queue (with a small amount of overhead)
- At each internal node visited store \((position, distance)\) in the queue
- Instead of backtracking, pop the closest from the queue and continue from it
The values of the parameters when fixed are the curves, while the trends remain the same as those shown. 

**Figure 5.** Approximate NN lookup vs dimension of space: Average of ratios of approximate to actual NN distance. (Uniform distribution; 100,000 stored points; $E_{max}(BBF) = 200$; $E_{max}(restricted) = 480$; averaged over 1000 queries.)

**Figure 6.** Approximate NN lookup vs $E_{max}$. (Uniform distribution; 100,000 stored points; dimension of space = 12; averaged over 1000 queries. $E_{max}(restricted)$ was 2.4 times $E_{max}(BBF)$.)
Figure 7. Approximate NN lookup vs Number of Stored Points. (Uniform distribution; \(E_{\text{max}}(BBF) = 200\); \(E_{\text{max}}(\text{restricted}) = 480\); dimension of space = 12; averaged over 1000 queries.)

Figure 8. Timing comparison of NN lookup methods. (Uniform distribution; 30,000 stored points; \(E_{\text{max}}(BBF)\) set to recover 95% of exact neighbours; Averaged over 1000 queries.)
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>KD-Tree</th>
<th>BBF</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Construction</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spliting Dimension</td>
<td>cyclic / greatest variance</td>
<td>greatest variance</td>
</tr>
<tr>
<td>Spliting Point</td>
<td>medoid</td>
<td>medoid</td>
</tr>
<tr>
<td>Number of Trees</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Nearest Neighbor Search</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exact / Approximate</td>
<td>exactly exact</td>
<td>approximate</td>
</tr>
<tr>
<td>Number of obligatory traverses</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Number of Nodes Examined</td>
<td>as many as needed to cover the whole tree</td>
<td>Emax (leaves)</td>
</tr>
<tr>
<td>Priority Queue</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Outline

KD-Trees

Best Bin First - BBF

Randomized KD-Trees

k-Means Tree

Best Bin First k-Means Tree

Greedy N-Best Paths Search
Randomized KD-Trees

Figure 1. [Priority search of aKD-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—i.e., 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.

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 $p_e/n$. On a logarithmic scale, this is a straight line with a slope of $\log(n/m)$. 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 nearest 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 contains 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.

Silpa-Anan, Hartley – CVPR 2008

Optimised KD-trees for fast image descriptor matching
Randomized KD-Trees

Main points:

- 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).

- 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.

- After traversing all trees once, use a shared priority queue.
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 $\mathcal{X} = x_i$. Creating KD-trees with different orientations simply means creating KD-trees from rotated data $Rx_i$, where $R$ is a rotation matrix.
- A principal (a regular) KD-tree is one created without any rotation, $R = I$.
- An algorithm for searching on a rotated KD-tree is essentially searching a rotated tree with a rotated query point $Rq$. 
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.
A saving using Householder matrices. Computation of $Rx$ 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 $H_v = I - 2vv^T/v^Tv$ is an orthonormal transformation; it is a reflection through a plane with normal vector $v$. Multiplication of a vector and $H_v$ 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 $O(mdN \log N)$ time.
NKD-Trees – Space Requirements

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 $2n$ and $2n + 1$ 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 4 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 $6N$ 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.
## Comparison

<table>
<thead>
<tr>
<th>. .</th>
<th>KD-Tree</th>
<th>BBF</th>
<th>NKD-Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Construction</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Spliting Dimension</strong></td>
<td>cyclic / greatest variance</td>
<td>greatest variance</td>
<td>greatest variance</td>
</tr>
<tr>
<td><strong>Spliting Point</strong></td>
<td>medoid</td>
<td>medoid</td>
<td>medoid</td>
</tr>
<tr>
<td><strong>Number of Trees</strong></td>
<td>1</td>
<td>1</td>
<td>m</td>
</tr>
<tr>
<td><strong>Relation Between Trees</strong></td>
<td>just one tree</td>
<td>just one tree</td>
<td>rotated</td>
</tr>
<tr>
<td><strong>Nearest Neighbor Search</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Exact / Approximate</strong></td>
<td>exactly exact</td>
<td>approximate</td>
<td>approximate</td>
</tr>
<tr>
<td><strong>Number of obligatory traverses</strong></td>
<td>1</td>
<td>1</td>
<td>m</td>
</tr>
<tr>
<td><strong>Number of Nodes Examined</strong></td>
<td>as many as needed to cover the whole tree</td>
<td>Emax (leaves)</td>
<td>n (n/m per tree)</td>
</tr>
<tr>
<td><strong>Priority Queue</strong></td>
<td>No</td>
<td>Yes</td>
<td>Yes (shared)</td>
</tr>
</tbody>
</table>
Randomized KD-Trees – RKD-Tree

Using Randomness

- 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.
- Splitting plane: Select 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\(^2\)
- In searching SIFT descriptors, this randomised tree (RKD-tree) performs as well as NKD-tree and it’s construction time is lower. Also note that the RKD-tree has the same complexity level in storage as the NKD-tree.

\(^2\)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.
Object retrieval with large vocabularies and fast spatial matching

- Spliting Point: the split value is randomly chosen using a point close to the median
Randomized KD-Trees – Speedup and build time without any regard to memory overhead, while setting $w_m = 1$ will give equal weight to a given percentage increase in memory use as to the same percentage increase in search and build time.

We choose the best nearest neighbor algorithm and the optimum parameters in a two step approach: a global exploration of the parameter space followed by a local tuning of the best parameters. Initially, we sample the parameter space at multiple points and choose those values that minimize the cost of equation 1. For this step we consider using $\{1, 4, 8, 16, 32\}$ as the number of random kd-trees, $\{16, 32, 64, 128, 256\}$ as the branching factor for the k-means tree and $\{1, 5, 10, 15\}$ as the number of k-means iterations. In the second step we use the Nelder-Mead downhill simplex method to further locally explore the parameter space and fine-tune the best parameters obtained in the first step. Although this does not guarantee a global minimum, our experiments have shown that the parameter values obtained are close to optimum.

The optimization can be run on the full dataset or just on a fraction of the dataset. Using the full dataset gives the most accurate results but in the case of large datasets it may take more time than desired. An option is provided to use just a fraction of the dataset to select parameter values. We have found that using just one tenth of the dataset typically selects parameter values that still perform close to the optimum on the full dataset. The parameter selection needs only be performed once for each type of dataset, and our library allows these values to be saved and applied to all future datasets of the same type.

4 EXPERIMENTS

4.1 Randomized kd-trees

Figure 2 shows the value of searching in many randomized kd-trees at the same time. It can be seen that performance improves with the number of random trees up to a certain point (about 20 random trees in this case) and that increasing the number of random trees further leads to static or decreasing performance. The memory overhead of using multiple random trees increases linearly with the number of trees, so the cost function may choose a lower number if memory usage is assigned importance.

4.2 Hierarchical k-means tree

The hierarchical k-means tree algorithm has the highest performance for some datasets. However, one disadvantage of this algorithm is that it often has a higher tree-build time than the randomized kd-trees. The build time can be reduced significantly by doing a small number of iterations in the k-means clustering stage instead of running it until convergence. Figure 3 shows the performance of the tree constructed using a limited number of iterations in the k-means clustering step relative to the performance of the tree when the k-means clustering is run until convergence. It can be seen that using as few as 7 iterations we get more than 90% of the nearest-neighbor performance of the tree constructed using full convergence, but requiring less than 10% of the build time.

When using zero iterations in the k-means clustering we obtain the more general GNAT tree of (Brin, 1995), which assumes that the data lives in a generic metric space, not in a vector space. However figure 3(a) shows that the search performance of this tree is worse than that of the hierarchical k-means tree (by factor of 5).

4.3 Data dimensionality.

Data dimensionality is one of the factors that has a great impact on the nearest neighbor matching performance. Figure 4(a) shows how the search performance degrades as the dimensionality increases. The datasets each contain $10^5$ vectors whose values are randomly sampled from the same uniform distribution. These random datasets are one of the most difficult problems for nearest neighbor search, as no value gives any predictive information about any other value. As can be seen in figure 4(a), the nearest-neighbor searches have a low efficiency for higher dimensional data (for 68% precision the approximate search speed is no better than linear search when the number of dimensions is greater than 800). However

Figure 2: Speedup obtained by using multiple random kd-trees (100K SIFT features dataset)
## Comparison

<table>
<thead>
<tr>
<th>.</th>
<th>KD-Tree</th>
<th>BBF</th>
<th>NKD-Trees</th>
<th>RKD-Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Construction</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Spliting Dimension</strong></td>
<td>cyclic / greatest variance</td>
<td>greatest variance</td>
<td>greatest variance</td>
<td>choose randomly one among the ones with the greatest variance</td>
</tr>
<tr>
<td><strong>Spliting Point</strong></td>
<td>medoid</td>
<td>medoid</td>
<td>medoid</td>
<td>choose randomly one among the ones close to medoid</td>
</tr>
<tr>
<td><strong>Number of Trees</strong></td>
<td>1</td>
<td>1</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td><strong>Relation Between Trees</strong></td>
<td>just one tree</td>
<td>just one tree</td>
<td>rotated</td>
<td>random(?)</td>
</tr>
<tr>
<td><strong>Nearest Neighbor Search</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Exact / Approximate</strong></td>
<td>exactly exact</td>
<td>approximate</td>
<td>approximate</td>
<td>approximate</td>
</tr>
<tr>
<td><strong>Number of obligatory traverses</strong></td>
<td>1</td>
<td>1</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td><strong>Number of Nodes Examined</strong></td>
<td>as many as needed to cover the whole tree</td>
<td>Emax (leaves)</td>
<td>n (n/m per tree)</td>
<td>n (n/m per tree)</td>
</tr>
<tr>
<td><strong>Priority Queue</strong></td>
<td>No</td>
<td>Yes</td>
<td>Yes (shared)</td>
<td>Yes (shared)</td>
</tr>
</tbody>
</table>
The virtual projection:

- If $N = 1.000.000$ then the height is 20. During a search with a query point in the tree, no more than 20 of the entries of the vector are considered!
- Typically only a small number $n$ of the data dimensions will be used to partition the tree. The other $128 - n$ dimensions will be unused, and irrelevant.
- Exactly the same tree will be obtained if the data is first projected by a projection $\pi$ onto a subspace of dimension $n$ before building the tree. Since no actual projection takes place, we refer to this as a virtual projection.
Assumptions/simplifications:

- 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 $\pi(q)$.

- So, in our model, we consider the order in which points $x_i$ are tested to find the closest match to $q$ is the order of the distance of $\pi(x_i)$ from $\pi(q)$.

- 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.
Essential question:

- Will the nearest neighbour to a query point in the high dimension, remain an approximate nearest neighbour after 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. The main reason that KD-trees perform badly in high dimensions is that points far from the query point $q$ 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 $\pi(x_{best})$ will be $n$-th closest point to $\pi(q)$ in $R^20$. The graph shows that the most likely ranking is first, but the graph has a long tail.

**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 $\pi$ into $R^{20}$. The nearest point $x_{best}$ to a query $q$ 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 $\pi(x_{best})$ will be $n$-th closest point to $\pi(q)$ 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 $x_{best}$ in $R^{128}$ among the $n$ points tested.
Principal Component Trees:

- One of the main reasons that KD-trees perform badly in high dimensions is that points far from the query point $q$ may be projected (via a virtual projection) to points close to the best match $x_{best}$. 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.
Randomized KD-Trees – PKD-Tree

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

- Before building the tree, the data points $x_i$ should be rotated via the mapping $U^T$ to align the principal axes with the coordinate axes.
- 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.
- An important point which we emphasize is that the original unprojected data must still be used in testing the distance between the query point $q$ and candidate closest neighbours.
Randomized KD-Trees – Performance

![Graphs showing performance of NKD-tree and RKD-tree](image)

- **NKD-tree’s error**
  - Maximum number of searched nodes vs. $p(\text{error})$ for different numbers of search trees.
  - The graph shows a significant improvement over an ordinary KD-tree (compare the top most line between figure 6 and figure 8).

- **RKD-tree’s error**
  - Maximum number of searched nodes vs. $p(\text{error})$ for different numbers of search trees.
  - The RKD-tree performs as well as the NKD-tree in searching for nearest neighbour queries; however, they are corrupted to the same number of dimensions.

- **PKD-tree’s error**
  - Maximum number of searched nodes vs. $p(\text{error})$ for different numbers of search trees when the data is projected by a projection.

- **PKD-tree’s error (128 dimensions)**
  - Maximum number of searched nodes vs. $p(\text{error})$ for different numbers of search trees when the data is projected by a projection with full 128 dimensions.

- **PKD-tree’s error (30 dimensions)**
  - Maximum number of searched nodes vs. $p(\text{error})$ for different numbers of search trees when the data is projected by a projection with 30 dimensions.

**Experimental results**

- The expected norm of the distance from the original point is discarded once the tree is built.

- The norm is normalised to length 1 including ones used for queries; however, they are corrupted to the same number of dimensions.

- For a PKD-tree, the effect 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.

- For a PKD-tree, the error reduces when we search more nodes. At the same number of search nodes, using more search trees clearly improves the performance but is not so marked, because the error reduces with some small Gaussian noise with standard deviation.

- The performance continues to improve but is not so marked, because the error reduces with some small Gaussian noise with standard deviation.

- In a single search tree case, there is also a significant improvement over an ordinary KD-tree (compare the top most lines between figure 6 and figure 8).
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>KD-Tree</th>
<th>BBF</th>
<th>N KD-Trees</th>
<th>RKD-Trees</th>
<th>PKD-Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Construction</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Preprocess Data</td>
<td>No</td>
<td>No</td>
<td>Yes, rotate them</td>
<td>No</td>
<td>Yes, rotate to align the principal axes with the coordinate axes</td>
</tr>
<tr>
<td>Splitting Dimension</td>
<td>cyclic / greatest variance</td>
<td>greatest variance</td>
<td>greatest variance</td>
<td>choose randomly one among the ones with the greatest variance</td>
<td>choose randomly among the ones with the k largest eigenvalues</td>
</tr>
<tr>
<td>Splitting Point</td>
<td>medoid</td>
<td>medoid</td>
<td>medoid</td>
<td>choose randomly one among the ones close to medoid</td>
<td>medoid(probably)</td>
</tr>
<tr>
<td>Number of Trees</td>
<td>1</td>
<td>1</td>
<td>m</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td><strong>Nearest Neighbor Search</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exact / Approximate</td>
<td>exactly exact</td>
<td>approximate</td>
<td>approximate</td>
<td>approximate</td>
<td>approximate</td>
</tr>
<tr>
<td>Number of obligatory traverses</td>
<td>1</td>
<td>1</td>
<td>m</td>
<td>m</td>
<td>m</td>
</tr>
<tr>
<td>Number of Nodes Examined</td>
<td>as many as needed to cover the whole tree</td>
<td>Emax (leaves)</td>
<td>n (n/m per tree)</td>
<td>n (n/m per tree)</td>
<td>n (n/m per tree)</td>
</tr>
<tr>
<td>Priority Queue</td>
<td>No</td>
<td>Yes</td>
<td>Yes (shared)</td>
<td>Yes (shared)</td>
<td>Yes (shared)</td>
</tr>
</tbody>
</table>
Outline

KD-Trees

Best Bin First - BBF

Randomized KD-Trees

k-Means Tree

Best Bin First k-Means Tree

Greedy N-Best Paths Search
Scalable Recognition With a Vocabulary Tree

Nister, Stewenius – CVPR 2006
k-Means Tree

Construction:

- The vocabulary tree defines a hierarchical quantization that is built by hierarchical k-means clustering. Instead of $k$ defining the final number of clusters or quantization cells, $k$ defines the *branch factor* (number of children of each node) of the tree.
- First, an initial k-means process is run on the training data, defining $k$ cluster centers. The training data is then partitioned into $k$ groups, where each group consists of the descriptor vectors closest to a particular cluster center.
- The same process is then recursively applied to each group of descriptor vectors, recursively defining quantization cells by splitting each quantization cell into $k$ new parts.
- The tree is determined level by level, up to some maximum number of levels $L$, and each division into $k$ parts is only defined by the distribution of the descriptor vectors that belong to the parent quantization cell.
k-Means Tree – Construction

For the most part, the above approaches keep amounts of data around in the database that is on the order of magnitude as large as the image patches themselves, or at least the region descriptors. However, the compactness of the database is very important for query efficiency in a large database. With our vocabulary tree approach, the representation of an image patch is simply one or two integers, which should be contrasted to the hundreds of bytes or floats used for a descriptor vector.

Compactness is also the most important difference between our approach and the hierarchical approach used by Grauman and Darrell [5]. They use a pyramid of histograms, at each level doubling the number of bins along each axis without considering the distribution of data. By using a vocabulary adapted to the likely distribution of data, we can use a much smaller tree, resulting in better resolution while maintaining a compact representation. We also estimate that our approach is around a factor 1000 faster.

For feature extraction, we use our own implementation of Maximally Stable Extremal Regions (MSERs) [10]. They have been found to perform well in thorough performance evaluation [13, 4]. We warp an elliptical patch around each MSER region into a circular patch. The remaining portion of our feature extraction is then implemented according to the SIFT feature extraction pipeline by Lowe [9]. Canonical directions are found based on an orientation histogram formed on the image gradients. SIFT descriptors are then extracted relative to the canonical directions. The SIFT descriptors have been found highly distinctive in performance evaluation [12]. The normalized SIFT descriptors are then quantized with the vocabulary tree. Finally, a hierarchical scoring scheme is applied to retrieve images from a database.

3. Building and Using the Vocabulary Tree

The vocabulary tree defines a hierarchical quantization that is built by hierarchical k-means clustering. A large set of representative descriptor vectors are used in the unsupervised training of the tree. Instead of $k$ defining the final number of clusters or quantization cells, $k$ defines the branch factor (number of children of each node) of the tree. First, an initial $k$-means process is run on the training data, defining $k$ cluster centers. The training data is then partitioned into $k$ groups, where each group consists of the descriptor vectors closest to a particular cluster center.

The same process is then recursively applied to each group of descriptor vectors, recursively defining quantization cells by splitting each quantization cell into $k$ new parts. The tree is determined level by level, up to some maximum number of levels $L$, and each division into $k$ parts is only defined by the distribution of the descriptor vectors that belong to the parent quantization cell. The process is illustrated in Figure 2.

In the online phase, each descriptor vector is simply propagated down the tree by at each level comparing the descriptor vector to the $k$ candidate cluster centers (represented by $k$ children in the tree) and choosing the closest one. This is a simple matter of performing $k$ dot products at each level, resulting in a total of $kL$ dot products, which is very efficient if $k$ is not too large. The path down the tree can be encoded by a single integer and is then available for use in scoring.

Note that the tree directly defines the visual vocabulary and an efficient search procedure in an integrated manner. This is different from for example defining a visual vocabulary non-hierarchically, and then devising an approximate nearest neighbor search in order to find visual words efficiently. We find the seamless choice more appealing, although the latter approach also defines quantization cells in the original space if used consistently and deterministically. The hierarchical approach also gives more flexibility to the subsequent scoring procedure.

While the computational cost of increasing the size of the vocabulary in a non-hierarchical manner would be very high, the computational cost in the hierarchical approach is...
k-Means Tree – Approximate Nearest Neighbor Search

Approximate Nearest Neighbor Search:

- In the online phase, each descriptor vector is simply propagated down the tree by at each level comparing the descriptor vector to the $k$ candidate cluster centers (represented by $k$ children in the tree) and choosing the closest one.
- This is a simple matter of performing $k$ dot products at each level, resulting in a total of $kL$ dot products, which is very efficient if $k$ is not too large.
• Note that the tree directly defines the visual vocabulary and an efficient search procedure in an integrated manner. This is different from for example defining a visual vocabulary non-hierarchically, and then devising an approximate nearest neighbor search in order to find visual words efficiently.
k-Means Tree vs Approximate Nearest Neighbor Kmeans

![Graph showing comparison between AKM and HKM](image)

- **AKM = 3.45**
- **HKM = 3.29**

**Table 2.** Comparison of the performance of exact k-means to our k-means vs AKM.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># of descr.</th>
<th>Voc. size</th>
<th>mAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>800K</td>
<td>10K</td>
<td>0.355</td>
<td>0.358</td>
</tr>
<tr>
<td>1M</td>
<td>20K</td>
<td>0.384</td>
<td>0.385</td>
</tr>
<tr>
<td>5M</td>
<td>50K</td>
<td>0.464</td>
<td>0.465</td>
</tr>
</tbody>
</table>

Clearly, more work is needed to understand the HKM performance here.
While the computational cost of increasing the size of the vocabulary in a non-hierarchical manner would be very high, the computational cost in the hierarchical approach is logarithmic in the number of leaf nodes.

The memory usage is linear in the number of leaf nodes $kL$. 
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>KD-Tree</th>
<th>BBF</th>
<th>NKD-Trees</th>
<th>RKD-Trees</th>
<th>PKD-Trees</th>
<th>k-Means Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Construction</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Preprocess Data</td>
<td>No</td>
<td>No</td>
<td>Yes, rotate them</td>
<td>No</td>
<td>Yes, rotate to align the principal axes with the coordinate axes</td>
<td>No</td>
</tr>
<tr>
<td>Spliting Dimension</td>
<td>cyclic / greatest variance</td>
<td>greatest variance</td>
<td>greatest variance</td>
<td>choose randomly one among the ones with the greatest variance</td>
<td>choose randomly among the ones with the k largest eigenvalues</td>
<td>no dimension splitting</td>
</tr>
<tr>
<td>Spliting Point</td>
<td>medoid</td>
<td>medoid</td>
<td>medoid</td>
<td>choose randomly one among the ones close to medoid</td>
<td>medoid (probably)</td>
<td>split at k points, centroids of k-means clustering</td>
</tr>
<tr>
<td>Number of Trees</td>
<td>1</td>
<td>1</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>1</td>
</tr>
<tr>
<td><strong>Nearest Neighbor Search</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exact / Approximate</td>
<td>exactly exact</td>
<td>approximate</td>
<td>approximate</td>
<td>approximate</td>
<td>approximate</td>
<td>approximate</td>
</tr>
<tr>
<td>Number of obligatory traverses</td>
<td>1</td>
<td>1</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>1</td>
</tr>
<tr>
<td>Number of Nodes Examined</td>
<td>as many as needed to cover the whole tree</td>
<td>Emax (leaves)</td>
<td>n (n/m per tree)</td>
<td>n (n/m per tree)</td>
<td>n (n/m per tree)</td>
<td>logN</td>
</tr>
<tr>
<td>Priority Queue</td>
<td>No</td>
<td>Yes</td>
<td>Yes (shared)</td>
<td>Yes (shared)</td>
<td>Yes (shared)</td>
<td>No</td>
</tr>
</tbody>
</table>
Outline

KD-Trees

Best Bin First - BBF

Randomized KD-Trees

k-Means Tree

Best Bin First k-Means Tree

Greedy N-Best Paths Search
Figure 6: Search efficiency. (a) Comparison of different algorithms. (b) Search speedup for different dataset sizes. (c) Search speedup when the query points don’t have “true” matches in the dataset vs the case when they have. (d) Search speedup for the Trevi Fountain patches dataset.

Our experiments showed that the randomized kd-trees have a significantly better performance for true matches, when the query features are likely to be significantly closer than other neighbors. Similar results were reported in (Mikolajczyk and Matas, 2007).

Muja, Love – ICCVTA 2006

Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration
Best-Bin-First k-Means Tree

- We have developed an algorithm that explores the hierarchical k-means tree in a best-bin-first manner, by analogy to what has been found to improve the exploration of the kd-tree.

- The algorithm initially performs a single traversal through the tree and adds to a priority queue all unexplored branches in each node along the path.

- Next, it extracts from the priority queue the branch that has the closest center to the query point and it restarts the tree traversal from that branch. In each traversal the algorithm keeps adding to the priority queue the unexplored branches along the path.

- The degree of approximation is specified in the same way as for the randomized kd-trees, by stopping the search early after a predetermined number of leaf nodes have been examined.
Figure 3: The influence that the number of k-means iterations has on the search time efficiency of the k-means tree (a) and on the tree construction time (b) (100K SIFT features dataset)
Figure 4: Search efficiency for data of varying dimensionality. The random vectors (a) represent the hardest case in which dimensions have no correlations, while most real-world problems behave more like the image patches (b).
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>KD-Tree</th>
<th>BBF</th>
<th>NKD-Trees</th>
<th>RKD-Trees</th>
<th>PKD-Trees</th>
<th>k-Means Tree</th>
<th>BBF k-Means Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Construction</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Preprocess Data</td>
<td>No</td>
<td>No</td>
<td>Yes, rotate them</td>
<td>No</td>
<td>Yes, rotate to align the principal axes with the coordinate axes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Splitting Dimension</td>
<td>cyclic / greatest variance</td>
<td>greatest variance</td>
<td>greatest variance</td>
<td>choose randomly one among the ones with the greatest variance</td>
<td>choose randomly among the ones with the k largest eigenvalues</td>
<td>no dimension splitting</td>
<td>no dimension splitting</td>
</tr>
<tr>
<td>Splitting Point</td>
<td>medoid</td>
<td>medoid</td>
<td>medoid</td>
<td>choose randomly one among the ones close to medoid</td>
<td>medoid(probably)</td>
<td>split at k points, centroids of k-means clustering</td>
<td>split at k points, centroids of k-means clustering</td>
</tr>
<tr>
<td>Number of Trees</td>
<td>1</td>
<td>1</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>k</td>
<td>1</td>
</tr>
<tr>
<td><strong>Nearest Neighbor Search</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exact / Approximate</td>
<td>exactly exact</td>
<td>approximate</td>
<td>approximate</td>
<td>approximate</td>
<td>approximate</td>
<td>approximate</td>
<td>approximate</td>
</tr>
<tr>
<td>Number of obligatory traverses</td>
<td>1</td>
<td>1</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>k</td>
<td>1</td>
</tr>
<tr>
<td>Number of Nodes Examined</td>
<td>as many as needed to cover the whole tree</td>
<td>Emax (leaves)</td>
<td>n (n/m per tree)</td>
<td>n (n/m per tree)</td>
<td>n (n/m per tree)</td>
<td>logN</td>
<td>n</td>
</tr>
<tr>
<td>Priority Queue</td>
<td>No</td>
<td>Yes</td>
<td>Yes (shared)</td>
<td>Yes (shared)</td>
<td>Yes (shared)</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Figure 6: Search efficiency. (a) Comparison of different algorithms. (b) Search speedup for different dataset sizes. (c) Search speedup when the query points don’t have “true” matches in the dataset vs the case when they have. (d) Search speedup for the Trevi Fountain patches dataset.
In our experiments, we have found that either of two algorithms can have the best performance, depending on the dataset and desired precision.

We have demonstrated that these can speed the matching of high-dimensional vectors by up to several orders of magnitude compared to linear search.
Outline

KD-Trees

Best Bin First - BBF

Randomized KD-Trees

k-Means Tree

Best Bin First k-Means Tree

Greedy N-Best Paths Search
Greedy N-Best Paths Search


City-scale location recognition.
Greedy N-Best Paths Search on k-Means Tree

Main idea:

• We propose an algorithm similar in spirit to BBF which exploits the unique properties of metric trees to allow us to specify how much computation takes place during nearest neighbor search.

• We propose the *Greedy N-Best Paths (GNP)* algorithm, which follows multiple branches at each level rather than just the branch whose parent is closest to the query feature.
Computational Cost:

- For branching factor $k$ and depth $L$, the normal search algorithm for a metric tree performs $k$ comparisons between the query feature and the nodes of the tree at each of $L$ levels for a total of $kL$ comparisons. Our algorithm performs $k$ comparisons at the top level, and $kN$ comparisons at each of the remaining $L - 1$ levels, for a total of $k + kN(L - 1)$ comparisons.

- This allows us to specify the amount of computation per search by varying the number of paths followed $N$.

- Traditional search is just the specific case in which $N = 1$. 
Greedy N-Best Paths Search on k-Means Tree

**Algorithm 1** Greedy N-Best Paths

Given query feature $q$, and level $\ell = 1$ 
Compute distance from $q$ to all $k$ children of root node 
While $(\ell < L)$

$\ell = \ell + 1$

Candidates=children of closest $N$ nodes at level $\ell - 1$
Compute distance from $q$ to all $kN$ candidates

} 
Return all features quantized under closest candidate
Greedy N-Best Paths Search on k-Means Tree

Effect of branching factor on performance:

- In previous work on vocabulary trees, it was noted that increasing the branching factor for fixed vocabulary size tended to improve the quality of search results.
The most important case is where entropy is defined with video independent of the database. For comparison, the result of using the ground truth target subset of images is also shown.
We claim that much of this improvement is due not to the fact that increasing branching factor produces better-structured trees, but to the fact that more nodes are being considered in traversing a tree with higher branching factor.

Changing the branching factor of a vocabulary tree requires time-consuming offline re-training via hierarchical k-means. However, varying the number of nodes searched is a decision that can be made at search time based on available computational power.
Thus, we should concentrate not on the relationship between performance and branching factor, but between performance and number of comparisons per query feature, a measure which GNP allows us to optimize.
Comparisons per query feature performance: By employing a geometric consistency check on the top 10 matches, we could achieve performance of more than 80%.

Vocabulary trees of any depth and branching factor can achieve comparable performance using GNP search. While vocabulary trees throw away this information, it is tempting to think that a kd-tree will naturally perform better at location recognition tasks and that one must compromise performance when using vocabulary trees. Just as GNP can be used to vary the amount of computation in a vocabulary tree search, the BBF algorithm can consider varying numbers of nodes visited in the search, these experiments show that we can achieve close to maximum performance in only 0.2 seconds per query (excluding feature extraction) using a k-means tree.

In the second experiment, we built trees of varying depth and branching factor against the set of 278 query images using the GNP search and searched more strongly than with the branching factor of 4. In Figure 6 that performance varies with the number of nodes searched. As discussed in Section 3.1, we can see from Figure 5, as images are added to the database, the performance drops for both types of trees. However, the tree built with new data for each situation.

Because kd-trees using the Best-Bin First algorithm access descriptors of every searched database feature, while vocabulary trees throw away this information, it is tempting to think that a kd-tree will naturally perform better at location recognition tasks and that one must compromise performance when using vocabulary trees. Just as GNP can be used to vary the amount of computation in a vocabulary tree search, the BBF algorithm can consider varying numbers of nodes visited in the search, these experiments show that we can achieve close to maximum performance in only 0.2 seconds per query (excluding feature extraction) using a k-means tree.

In the second experiment, we built trees of varying depth and branching factor against the set of 278 query images using the GNP search and searched more strongly than with the branching factor of 4. In Figure 6 that performance varies with the number of nodes searched. As discussed in Section 3.1, we can see from Figure 5, as images are added to the database, the performance drops for both types of trees. However, the tree built with new data for each situation.

In addition, in Figure 6 we compare the performance of a vocabulary tree against traditional kd-tree search. These results show the performance of a vocabulary tree is largely dependent upon its structure. Note that in both trees, the contents of the vocabulary tree are exactly the same, so that the only difference is performance of a vocabulary tree is largely dependent upon its structure.
Greedy N-Best Paths Search on k-Means Tree - Performance

Figure 3. Greedy N-Best Paths Search. From left to right, we increase the number of nodes $N$ whose children are considered at each level of the tree. Cells are colored from red to green according to the depth at which they are encountered in the tree, while gray cells are never searched. By considering more nodes in the tree, recognition performance is improved at a computational cost that varies with $N$. 

3. Greedy N-Best Paths Search

A popular search heuristic for approximate nearest neighbors in kd-trees is the Best Bin First (BBF) algorithm [1]. Bounds are computed on the nearest possible feature residing in each path not followed as the search descends down the tree, and a specified number of candidate features are considered in succession. We propose an algorithm similar in spirit to BBF which exploits the unique properties of metric trees to allow us to specify how much computation takes place during nearest neighbor search. We propose the Greedy N-Best Paths (GNP) algorithm, which follows multiple branches at each level rather than just the branch whose parent is closest to the query feature. This generalization of the traditional vocabulary tree search method is described in Algorithm 1.

For branching factor $k$ and depth $L$, the normal search algorithm for a metric tree performs $kL$ comparisons. Our algorithm performs $k$ comparisons at the top level, and $kN$ comparisons at each of the remaining $L-1$ levels, for a total of $k + kN(L-1)$ comparisons. This allows us to specify the amount of computation per search by varying the number of paths followed $N$. Note that traditional search is just the specific case in which $N=1$.

3.1. Branching Factor

For a fixed vocabulary size $M$, corresponding to the number of leaf nodes in a tree, there are several ways to construct a vocabulary tree. This is accomplished by varying the branching factor $k$ and depth $L$ of the tree such that $kL \approx M$ for integer values of $k$ and $L$. In previous work on vocabulary trees [9], it was noted that increasing the branching factor for fixed vocabulary size tended to improve the quality of search results. We claim that much of this improvement is due not to the fact that increasing branching factor produces better-structured trees, but to the fact that the GNP algorithm offers a way to consider more nodes in a tree with higher branching factor. As an example, using a 1 million word vocabulary, consider that in a $10^6$ tree only 60 nodes are ever examined while in a $1000^2$ tree 2000 nodes are considered during a traditional search. The GNP algorithm offers a way to consider 2010 nodes in a $10^6$ tree with $N=40$, and we show in Section 6 that comparable performance is achieved with GNP on a tree with fewer branches. Note that changing the branching factor of a vocabulary tree requires time-consuming offline re-training via hierarchical k-means. However, varying the number of nodes searched is a decision that can be made at search time based on available computational power. Thus, we should concentrate not on the relationship between performance and branching factor, but between performance and number of comparisons per query feature, a measure which GNP allows us to optimize (see Figure 6).
## Final Comparison

<table>
<thead>
<tr>
<th></th>
<th>KD-Tree</th>
<th>BBF</th>
<th>NKD-Tree</th>
<th>RKD-Tree</th>
<th>PKD-Tree</th>
<th>k-Means Tree</th>
<th>BBF k-Means Tree</th>
<th>GNP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Construction</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Preprocess Data</td>
<td>No</td>
<td>No</td>
<td>Yes, rotate them</td>
<td>No</td>
<td>Yes, rotate to align the principal axes with the coordinate axes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Spliting Dimension</td>
<td>cyclic / greatest variance</td>
<td>greatest variance</td>
<td>greatest variance</td>
<td>choose randomly one among the ones with the greatest variance</td>
<td>choose randomly among the ones with the k largest eigenvalues</td>
<td>no dimensin splitting</td>
<td>no dimensin splitting</td>
<td>no dimensin splitting</td>
</tr>
<tr>
<td>Spliting Point</td>
<td>medoid</td>
<td>medoid</td>
<td>medoid</td>
<td>choose randomly one among the ones close to medoid</td>
<td>medoid(probably)</td>
<td>split at k points, centroids of k-means clustering</td>
<td>split at k points, centroids of k-means clustering</td>
<td>split at k points, centroids of k-means clustering</td>
</tr>
<tr>
<td>Number of Trees</td>
<td>1</td>
<td>1</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Nearest Neighbor Search</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Exact / Approximate</td>
<td>exactly exact</td>
<td>approximate</td>
<td>approximate</td>
<td>approximate</td>
<td>approximate</td>
<td>approximate</td>
<td>approximate</td>
<td>approximate</td>
</tr>
<tr>
<td>Number of obligatory traverses</td>
<td>1</td>
<td>1</td>
<td>m</td>
<td>m</td>
<td>m</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Number of Nodes Examined</td>
<td>as many as needed to cover the whole tree</td>
<td>E_max (leaves)</td>
<td>n (n/m per tree)</td>
<td>n (n/m per tree)</td>
<td>n (n/m per tree)</td>
<td>kL</td>
<td>n</td>
<td>k+kL(L-1)</td>
</tr>
<tr>
<td>Priority Queue</td>
<td>No</td>
<td>Yes</td>
<td>Yes (shared)</td>
<td>Yes (shared)</td>
<td>Yes (shared)</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>