# An Empirical Study of a New Approach to Nearest Neighbor Searching[⋆]

Songrit Maneewongvatana and David M. Mount

Department of Computer Science,
University of Maryland, College Park, Maryland
{songrit,mount}@cs.umd.edu

**Abstract.** In nearest neighbor searching we are given a set of $n$ data points in real $d$-dimensional space, $\Re^d$, and the problem is to preprocess these points into a data structure, so that given a query point, the nearest data point to the query point can be reported efficiently. Because data sets can be quite large, we are interested in data structures that use optimal $O(dn)$ storage.

In this paper we consider a novel approach to nearest neighbor searching, in which the search returns the correct nearest neighbor with a given probability assuming that the queries are drawn from some known distribution. The query distribution is represented by providing a set of *training query points* at preprocessing time.

The data structure, called the *overlapped split tree*, is an augmented BSP-tree in which each node is associated with a cover region, which is used to determine whether the search should visit this node. We use principal component analysis and support vector machines to analyze the structure of the data and training points in order to better adapt the tree structure to the data sets. We show empirically that this new approach provides improved predictability over the kd-tree in average query performance.

## 1 Introduction

Nearest neighbor searching is a fundamental problem the design of geometric data structures. Given a set $S$ of $n$ data points in some metric space, we wish to preprocess $S$ into a data structure so that given any query point $q$, the nearest point to $q$ in $S$ can be reported quickly. Nearest neighbor searching has applications in many areas, including knowledge discovery and data mining [20], pattern recognition and classification [13, 17], machine learning [12], data compression [24], multimedia databases [21], document retrieval [15], and statistics [16]. Many data structures have been proposed for nearest neighbor searching. Because many applications involve large data sets, we are interested in data structures that use only linear or nearly linear storage. Throughout we will

[⋆] The support of the National Science Foundation under grant CCR–9712379 is gratefully acknowledged.

assume that the space is real $d$-dimensional space $\Re^d$, and the metric is any Minkowski metric. For concreteness we consider Euclidean distance.

Naively, nearest neighbor queries can be answered in $O(dn)$ time. The search for more efficient data structures began with the seminal work by Friedman, Bentley, and Finkel [22], who showed that such queries could be answered efficiently in fixed dimensions through the use of kd-trees. Since then many different data structures have been proposed from the fields of computational geometry and databases. These include the $R$-tree and its variants [4, 33], the *X-tree* [5], the SR-tree [26], the *TV-tree* [29], and the BAR tree [18], not to mention numerous approaches based on Voronoi diagrams and point location [14].

Although nearest neighbor searching can be performed efficiently in low-dimension spaces, for all known linear-space data structures, search times grow exponentially as a function of dimension. Even in moderately large dimensional spaces, in the worst case, a large fraction of points in $S$ are visited in the search. One explanation for this phenomenon is that in high dimensional space, the distribution of the inter-point distances tends to concentrate around the mean value. As a consequence, it is difficult for the search algorithm to eliminate points as being candidates for the nearest neighbor. Fortunately, in many of the data sets that arise in real applications, correlations between dimensions are common, and as a consequence the points tend to cluster in lower dimensional subspaces [19]. Good search algorithms take advantage of this low-dimensional clustering to reduce search times.

A popular approach to reducing the search time is through *approximate nearest neighbor search*. Given $\epsilon > 0$ the search may return any point $p$ in $S$ whose distance to query $q$ is within a factor of $(1 + \epsilon)$ of the true nearest distance. Approximate nearest neighbor search provides a trade-off between speed and accuracy. Algorithms and data structures for approximate nearest neighbor searching have been given by Bern [6], Arya and Mount [1], Arya, et al. [2], Clarkson [11], Chan [8], Kleinberg [27], Indyk and Motwani [25], and Kushilevitz, Ostrovsky and Rabani [28].

Our experience with the ANN library for approximate nearest neighbor searching, we have observed two important phenomena. The first is that significant improvements in search times often require uncomfortably large values of $\epsilon$. The second is that the actual average error committed in the search is typically much smaller (by factors of 10 to 30) than $\epsilon$. The combination of these two effects result in an undesirable lack of predictability in the performance of the data structure. In order to achieve greater efficiency, users often run the algorithm with a high $\epsilon$, and sacrifice assurances of accuracy on each query for better speed and the hope of good average case performance [32]. One of our motivations in this research is to find data structures that provide good *efficiency* but with a higher degree of *predictability*.

In this paper we consider an alternative approach for dealing with this shortcoming. We propose a new data structure for nearest neighbor searching, which applies a different kind of approximation. In many applications of nearest neighbor searching, the performance of any one query is not as important as the aggregate results. One example is image compression based on vector quantization [24]. An error committed on a single pixel may not seriously impact the overall image quality. Our approach may be described as a *probably-correct nearest neighbor search*. Assuming that the queries are drawn from some known distribution the search returns the true nearest neighbor with a probability that user can adjust. The query distribution is described by providing a set of *training queries* as part of the preprocessing stage. By analyzing the training data we can better adapt the data structure to the distributional structure of the queries. The idea of allowing occasional failures was considered earlier by Ciaccia and Patella [10] in a more limited setting.

We introduce a new data structure for this problem called an *overlapped split tree* or *os-tree* for short. The tree is a generalization of the well known BSP-tree, which uses the concept of a covering region to control which nodes are searched. We will introduce this data structure in the next section, and discuss how it can be applied for both exact and probably-correct nearest neighbor searching. In the subsequent section we provide experimental evidence for the efficacy and efficiency of this data structure. We show empirically that it provides an enhanced level of predictability in average query performance over the kd-tree data structure.

## 2 Overlapped Split Tree

The os-tree is a generalization of the well-known *binary space partition (BSP) tree* (see, e.g., [14]). Consider a set $S$ of points in $d$-dimensional space. A BSP-tree for this set of points is based on a hierarchical subdivision of space into convex polyhedral *cells*. Each node in the BSP-tree is associated with such a cell and the subset of points lying within this cell. The root node is associated with the entire space and the entire set. A cell is split into two disjoint cells by a hyperplane, and these two cells are then associated with the children of the current node. Points are distributed among the children according the cell in which they are contained. The process is repeated until the number of points associated with a node falls below a given threshold. Leaf nodes store these associated points.

Many of the data structures used in nearest neighbor searching are special cases or generalizations of the BSP-tree, but we will augment this data structure with an additional element. In addition to its BSP-tree structure, each node of the os-tree is associated with an additional convex polyhedral cell called a *cover*. Intuitively, the cover associated with each node contains every point in space whose the nearest neighbor is a data point in the associated cell. That is, the cover contains the union of the Voronoi cells [14] of the points associated with the cell. (Later we will need to relax this condition, for purposes of computational efficiency.) The covers of the children of a node will generally overlap each other.

Fig. 1 shows an example of a parent and its two children in the plane. In each case the cell is shaded and the cover is the surrounding polygon. In typical BSP fashion, the parent cell is split by a line, which partitions the point set and cell. The Voronoi bisector between the two subsets of points is shown as a broken

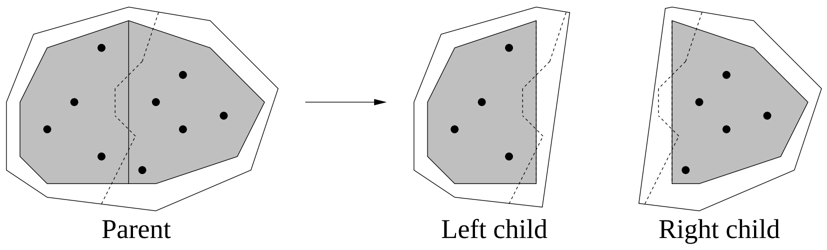

**Fig. 1.** Overlapping covers

line. Observe that the covers for the left and right children are large enough to enclose the Voronoi bisector, and hence all the points of the space whose nearest neighbor lies within the corresponding side of the partition.

Let us describe the os-tree more formally. The data point set is denoted by $S$. Each node of the tree is denoted by a string of $l$'s and $r$'s. The root node is labeled with the empty string, $\phi$. Given a node $\delta$ its left child is $\delta l$, and its right child is $\delta r$. (The terms left and right are used for convenience, and do not connote any particular orientation in space.) Give a node $\delta$, we use $c_\delta$ to denote its cell, $C_\delta$ to denote its cover, and $S_\delta$ to denote its subset of points.

The entities $c_\delta$ and $S_\delta$ together form a BSP-tree for the point set. The cover of a node $\delta$ is a convex polyhedron that has the following properties:

– The cover of the root node $\phi$ is the entire space.
– Let $V(p)$ denote the Voronoi cell of point $p$ in $S_\delta$, and let $V_\delta = \bigcup_{p \in S_\delta} V(p)$. Then
$$C_\delta \supseteq V_\delta.$$
– Let $H_\delta$ be a set of $(d-1)$-dimensional hyperplanes that bound $C_\delta$. There are two parallel hyperplanes $L$ and $R$ such that
$$H_{\delta l} \subseteq H_\delta \cup \{L\} \qquad H_{\delta r} \subseteq H_\delta \cup \{R\}.$$
(Note that these hyperplanes need not be parallel to the BSP splitting hyperplane.)

Each internal node of the tree stores the coefficients of hyperplanes $L$ and $R$. The selection of these hyperplanes will be discussed later.

To answer the nearest neighbor query $q$, the search starts at the root. At any internal node $\delta$, it determines if the query lies within either cover $C_{\delta l}$ or $C_{\delta r}$ (it may be in both) . For each cover containing $q$, the associated child is visited recursively. From the definition of the cover, if the query point does not lie within the cover then the subtree rooted at this node cannot possibly contain the nearest neighbor. When the search reaches a leaf node, distances from the query to all associated points in the leaf are computed. When all necessary nodes have been visited, the search terminates and returns the smallest distance seen and the associated nearest neighbor. It is easy to see that this search returns the correct result. Observe that the efficiency of the search is related to how "tightly" the cover surrounds the union of the Voronoi cells for the points, since looser covers require that more nodes be visited.

## 2.1 Probably Correct os-tree

One of the difficulties in constructing the os-tree as outlined in the previous section, is that it would seem to require knowledge of the Voronoi diagram. The combinatorial complexity of the diagram grows exponentially with dimension, and so explicitly computing the diagram for high dimensional nearest neighbor searching is impractical. In order to produce a more practical construction, we first introduce a variant of the os-tree in which queries are only answered correctly with some probability.

Let us assume for now that the probability density function of query points is known. (This is rather bold assumption, but later we will see that through the use of training data we can approximate what we need to know about the distribution reasonably accurately.) In particular given a region $X \subseteq \Re^d$, let $Q(X)$ to be the probability that a query lies within $X$. Given real $f$, $0 \leq f \leq 1$, let $C(f,Q)_\delta$ denote any convex polyhedron such that

$$\frac{V_\delta \setminus Q(C(f,Q)_\delta)}{Q(V_\delta)} \leq f.$$

In other word, the region $C(f,Q)_\delta$ contains all but at most a fraction $f$ of the probability mass of $V_\delta$.

The os-tree has the property that the cover of each node contains the covers of its children. Given the probability density $Q$ and a search algorithm, the *failure probability* of the algorithm is the probability (relative to $Q$) that this algorithm fails to return the correct nearest neighbor. The following is easy to prove.

**Lemma 1.** *Given queries drawn from $Q$, if the cover for each leaf $\delta$ of an os-tree satisfies the condition $C(f,Q)_\delta$ above, then the failure probability for the os-tree search is at most $f$.*

The lemma provides sufficient conditions on the covers of the os-tree to guarantee a particular failure probability. In the next section, we discuss the efficient construction of the os-tree in greater detail.

## 2.2 Building the os-tree

We do not know of a practical way to construct the ideal os-tree that we have outlined earlier. This is because complex query distributions are not generally easily representable, and the shape of Voronoi cells of a high dimensional point set can be highly complex. Finding a convex polyhedron that covers a desired fraction of the density of a set of Voronoi cells seems to require intensive computation. We incorporate the following changes to the os-tree in our implementation:

- Instead of using an abstract query distribution $Q$, we use a set $T$ of *training query points*, which are assumed to be randomly sampled from $Q$. Each training point will be labeled according to its nearest neighbor in $S$, that is, the Voronoi cell containing it.
- In the search process, we add the distance comparison (described below) as another criterion for pruning nodes from being visited.

Although the assumption of the existence of the training set is a limitation to the applicability of our approach, it is not an unreasonable one. In most applications of nearest neighbor searching there will be many queries. One mode in which to apply this algorithm is to sample many queries over time, and update the data structure periodically based on the recent history of queries.

Given the data points $S$ and training points $T$, the construction process starts by computing the nearest neighbor in $S$ for each point in $T$. The construction then starts with the root node, whose cell and cover are the entire space. In general, for node $\delta$, there will be associated subsets $S_\delta$ and $T_\delta$. If the number of points in $S_\delta$ is smaller than a predefined threshold, then these points are stored in a leaf cell. Otherwise $c_\delta$ is split into two new cells by a *separating hyperplane*, $H_\delta$, which partitions $S_\delta$ into subsets of roughly equal sizes. (The computation of $H_\delta$ is described below.) This hyperplane may have an arbitrary orientation. We partition the points among the two resulting child nodes.

We then use the training set $T_\delta$ to compute the cover of the node. Partition this into $T_{\delta l}$ and $T_{\delta r}$ depending whether its nearest neighbor is in $S_{\delta l}$ or $S_{\delta r}$, respectively. We then compute two (signed) hyperplanes, $B_{\delta l}$ and $B_{\delta r}$, called the *boundary hyperplanes.* (By a method described below.) These two hyperplanes are parallel to each other but have oppositely directed normal vectors. Define $B_{\delta l}$ so that it bounds the smallest halfspace enclosing $T_{\delta l}$. This can be done by computing the dot product between each point in $T_{\delta l}$ and $B_{\delta l}$'s normal, and taking the minimum value. Do the same for $B_{\delta r}$. These two hyperplanes are then stored with the resulting node. Define $C_{\delta l}$ to be $C_\delta \cap B'_{\delta l}$, where $B'_{\delta l}$ is the halfspace bounded by $B_{\delta l}$, and do similarly for $C_{\delta r}$. The process is then applied recursively to the children.

To answer the query, the search starts at the root node. At any internal node, we find the location of the query point relative to the separating hyperplane $H_\delta$ and the two boundary hyperplanes $B_{\delta l}$ and $B_{\delta r}$. (This can be done in $O(d)$ time.) For concreteness, suppose that we are on the left side of the separating hyperplane. (The other case is symmetrical.) We visit the left child first. If the query point does not lie within right bounding halfspace, $B'_{\delta r}$ (that is, it is not in the overlap region) then the right child will not be visited. Even if the query is in the overlap region, but if the current nearest neighbor distance is less than the distance from the query point to the separating hyperplane, then the right child is not visited. (Since no point on the other side of $H_\delta$, and hence no point in $S_{\delta r}$ can be closer.) Otherwise the right child is visited. When we reach a leaf node, distances to the data points are computed and the smallest distance is updated if needed.

## 2.3 The os-tree Splitting Rule

The only issues that are left to be explained are the choices of the splitting hyperplane and the boundary hyperplanes. We impose the requirement that roughly half of the points in the cell lie on one each side of the splitting hyperplane. This condition guarantees a balanced tree. An obvious criterion for selecting the orientation of the hyperplane is to subdivide the points orthogonal to the direction of greatest variation. To do this we use the well known *principal component analysis* (PCA) method [23]. We compute the covariance matrix for the data points $S_\delta$, and sort the points according to their projection onto the eigenvector corresponding to the largest eigenvalue. Half of the resulting sequence half of the points are placed in $S_{\delta l}$ and the larger half in $S_{\delta r}$. The use of PCA's in choosing splitting planes has been considered before (see, e.g., [35]).

It is clear that if the maximum of the projections of $l$ labeled point is lower than the minimum of the projections of $r$ labeled points, then there are infinitely many hyperplanes that can be choosen as the separating hyperplane. The optimal hyperplane (with respect to the search time) is the one that minimizes the number of training points in the final overlap region. The more training points in the overlap region, the more likely that both children of the current node will be visited. We use the *support vector machine* (SVM) method [34] to choose separating hyperplane. Even the SVM method does not give us the optimal hyperplane, but it finds a hyperplane that is good enough in the sense that it is the one that is the furthest from closest data points.

The SVM method was developed in the area of learning theory [34] to find the hyperplane that best separates two classes of points in multidimensional space. A more detailed description of SVM can be found in [7]. In this paper, we will review the basic linear SVM method using the hyperplane as the separator in the original space.

By construction the two data point sets are linearly separable. In this case SVM finds a separating hyperplane with the highest margin, where *margin* is defined as the sum of the distance from the hyperplane to the points in $S_{\delta l}$ and the distance to the points in $S_{\delta r}$. SVM formulates this problem a nonlinear optimization problem. Solving the optimization problem returns a set of points called *support vectors* as well as other coefficients. These values are used to determine the separating hyperplane. The result of SVM is the separating hyperplane $H_\delta$ of the node. Now, the training points associated with the node are used to determine the final placement of the boundary hyperplanes $H_{\delta l}$ and $H_{\delta r}$, as described earlier.

## 2.4 Implementation

We modified the ANN library [30] to accommodate the os-tree. This is a library for approximate nearest neighbor search using the kd-tree and bbd-tree [3] as the underlying data structures. We use a PCA program from [31] and the LIBSVM library [9] as the SVM module. LIBSVM is a simple and fast implement of SVM. It uses a very simple optimization subroutine to find the optimal separating hyperplane. Like most other SVM packages, the result of LIBSVM may not be the optimal hyperplane. This is due to numerical errors and the penalty cost. Based on our experiments, if we use a higher value of penalty cost, it usually produces a better separating hyperplane, but it requires more CPU time. In all of our experiments, we set the penalty cost to 1000.

Even though LIBSVM is relatively fast, it was the main bottleneck in the construction time of the data structure. Its running time is superlinear in the number of input points. In the implementation, instead of providing SVM with the entire set of data points in the cell, we divided the points into smaller batches. We call this version the *batched SVM*. Each batch has a relatively small number of points, 200. We then invoke the SVM routine for each batch, and collect the resulting support vectors. We ignore other points that are not support vectors because they are unlikely to have a significant influence on the final result of SVM. The support vectors of each batch are then combined and used as the input to SVM. This process is repeated until the size of the input is small enough to run a single SVM. We use the result from this final run as the separating hyperplane of the cell.

Obviously, the batched version of SVM would be expected to perform more poorly than the standard (unbatched) method. But we found that over a number of experiments it usually produced a hyperplane that is close to the result of the standard algorithm. The CPU time required to find the hyperplane using the batched SVM is significantly less than one using the standard SVM.

## 3 Experimental Results

### 3.1 Synthetic Data Sets

We begin by discussing our results on synthetically generated data sets. Because the os-tree requires a relatively large number of training points, one advantage of synthetic data sets is that it is easy to generate sufficiently large training sets. To emulate real data sets, we chose data distributions that are clustered into subsets having low intrinsic dimensionality. (Our subsequent experiments with real data sets bears out this choice.) The two distributions that we considered are outlined below.

**Clustered-rotated-flats:** Points are distributed in a set of clusters. The points in each cluster are first generated along an axis-aligned flat, and then this flat is rotated randomly. The axes parallel to the flat are called *fat dimensions*, and the others are called *thin dimensions*. The distribution is characterized by the following parameters:

- the number of clusters (fixed at 5 clusters in all of our experiments),
- the number of fat dimensions,
- the standard deviation for thin dimensions, $\sigma_{\text{thin}}$.
- the number of rotations applied.

The center of each cluster is sampled uniformly from the hypercube $[-1,1]^d$. Then we randomly select the fat dimensions from among axes of the full space. Points are distributed randomly and evenly among the clusters. In fat dimensions, coordinates are drawn from a uniform distribution in the interval $[-1,1]$. In thin dimensions, they are sampled from a Gaussian distribution with standard deviation $\sigma_{\mathrm{thin}}$. We compute a random rotation matrix for each cluster. This is done by repeatedly multiplying a rotation matrix (initially a $d \times d$ identity matrix) with a matrix $A$. $A$ matrix is an identity matrix except for four elements $A_{ii} = A_{jj} = \cos(\theta), A_{ij} = -A_{ji} = \sin(\theta)$, where $i$ and $j$ are randomly chosen axes and $\theta$ is randomly chosen from $-\pi/2$ to $\pi/2$. We apply the rotation matrix to all points in the cluster and then translate them by a vector from the origin to the center of the cluster.

**Clustered-rotated-ellipsoids:** This distribution is similar to the clustered-rotated-flats distribution, except that the coordinates are sampled from a Gaussian distribution on fat dimensions instead of a uniform distribution. It has the additional parameters $\sigma_{\mathrm{lo}}$ and $\sigma_{\mathrm{hi}}$, which are the minimum and maximum standard deviations of the Gaussian distribution of the fat dimensions. The actual standard deviation is randomly chosen in $[\sigma_{\mathrm{lo}}, \sigma_{\mathrm{hi}}]$. In our experiments, we used $\sigma_{\mathrm{lo}} = 0.25$ and $\sigma_{\mathrm{hi}} = 0.5$.

We compared the query performance of the os-tree against that of the kd-tree. The query performance was measured by the total number of nodes visited, the number of leaf nodes visited, and the CPU time used. We show CPU time in most of our graphs. We compiled both programs using the g++-2.95.2 compiler with the same options, and we conducted all experiments in the same class of machines (UltraSparc-5 running Solaris 2.6 for the synthetic data sets and a PC Celeron 450 running Linux for the real data sets). In the first set of experiments we also compared the total number of nodes visited as well.

There is some difficulty in a direct comparison between the os-tree and the kd-tree because the search models are different: probably-correct model in os-tree and approximately correct model in the kd-tree. To reconcile this difference, in each experiment, we adjusted the $\epsilon$ value (approximation parameter) of the kd-tree so that the resulting failure probability of the kd-tree is similar to that of the os-tree. Once the epsilon value is found, we ran the experiments with the same query set because changing it may alter the failure probability. This gives the kd-tree a slight advantage, because the query set is the same as the training set.

The results of the first set of experiments are shown in Fig. 2. The point set, training set and query set (for os-trees) are all sampled from the clustered rotated ellipsoids distribution. The number of clusters is fixed at 5 and $\sigma_{\mathrm{thin}}$ is fixed at 0.05 and 0.25. We show the results for dimensions $d = 10, 20, 30$. The number of fat dimensions is fixed, relative to $d$, at $\frac{3d}{10}$. The number of rotations applied is equal to the dimension of the space. The size of the point set varies from 2K (K = 1024) to 32K points. For all experiments, the number of training points (in os-trees) is fixed at 200 times the size of the point set and the results are average over 5 different trees, each with 10K different queries.

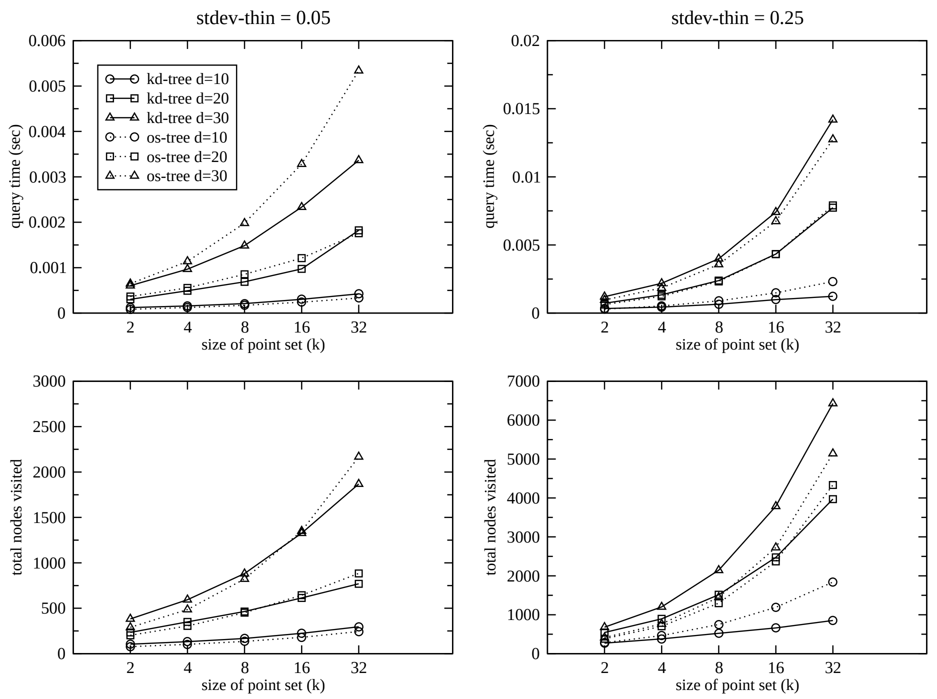

**Fig. 2.** Query time and number of nodes visited comparison for the kd-tree and os-tree. All distributions are clustered-rotated-ellipsoids

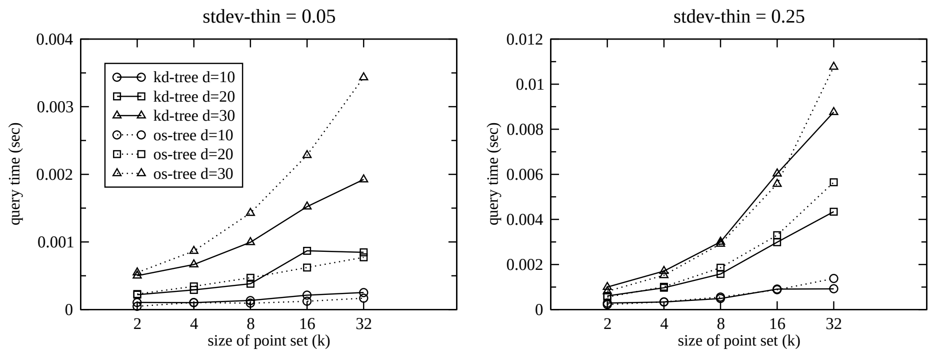

**Fig. 3.** Query time comparison for the kd-tree and os-tree. All distributions are clustered-rotated-flats

The top graphs of Fig. 2 shows comparisons of the average CPU time each query uses. The bottom graphs compare the total number of nodes visited. We can see that these two running times are remarkably similar. The kd-tree's query time is somewhat better than that of os-tree when the point set is large and $\sigma_{\mathrm{thin}}$ is high. The search in os-tree is slightly faster in low dimension ($d = 10$) and for smaller data set sizes. Overall, the differences in the query time of both trees are quite small.

The second set of experiments is quite similar to the first one except that we change the distribution from clustered rotated ellipsoids to clustered rotated flats. The query time comparison of both trees is presented in Fig. 3. The results of Fig. 3 are very similar to those of Fig. 2. The query times of both trees are comparable. Note that the search is somewhat faster for clustered rotated flat distributions for both trees.

In the next set of experiments we varied the density of the clusters. By varying $\sigma_{\mathrm{thin}}$ from 0.01 to 0.5, the clusters are less dense and the distribution is more uniform. The point set size is fixed at 4K and 16K with $d$ rotations applied. Fig. 4 shows the results. Again, the performances of both trees are quite similar.

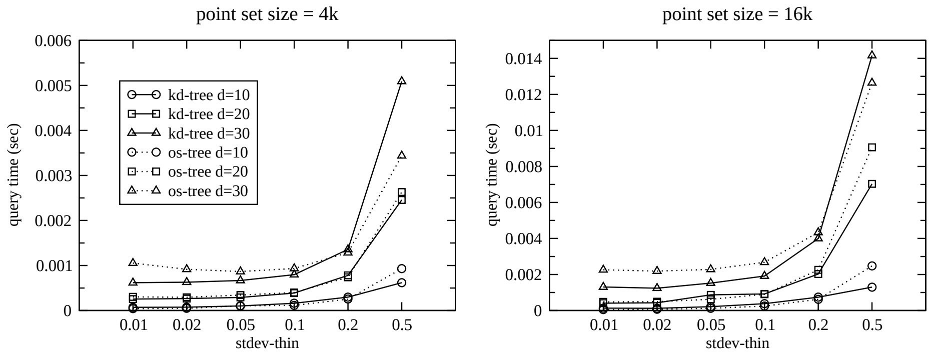

**Fig. 4.** Effect of $\sigma_{\mathrm{thin}}$ (stdev-thin, standard deviation on thin dimensions of the flats). All distributions are clustered rotated flats

## 3.2 Enhanced Predictability

We have shown that the os-tree and kd-tree are similar with respect to running time (subject to the normalizations mentioned earlier). The principal difference between them is in the degree to which it is possible to *predict* performance. In all the experiments above, we set the size of training set to be 200 times the size of the point set. Based on the ratio between the size of training set and the size of data set, the predicted failure probability of the os-tree is around 0.5%. We also computed the failure rate of similar experiment runs of the kd-tree with $\epsilon = 0.52$ (average value of matching values of $\epsilon$) and $\epsilon = 0.8$. Fig. 5 shows the failure rate sorted in increasing order of some 258 experimental runs. We can see that the variation of the failure rate of kd-tree is much higher (which is not unexpected, since the kd-tree is not designed for probably-correct nearest neighbor search.) This justifies our claim that the os-tree provides comparable efficiency as the kd-tree, but with significantly enhanced predictability.

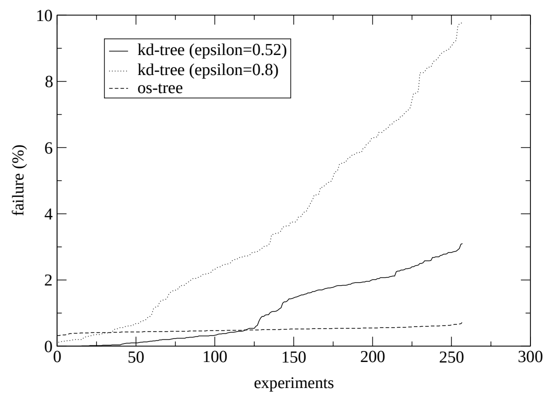

**Fig. 5.** Failure rate, sorted in increasing order from 258 runs. All distributions are clustered rotated ellipsoids

## References

[1] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 271–280, 1993.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 573–582, 1994.

[3] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. *Journal of the ACM*, 45:891–923, 1998.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R$^*$-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 322–331, 1990.

[5] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. 22nd VLDB Conference*, pages 28–39, 1996.

[6] M. Bern. Approximate closest-point queries in high dimensions. *Inform. Process. Lett.*, 45:95–99, 1993.

[7] C. J.C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

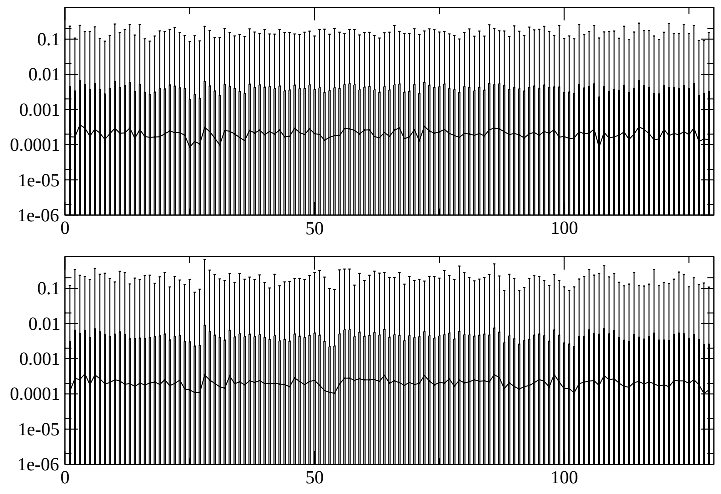

**Fig. 6.** Average error. Showing the average (connecting line), standard deviation (box), and maximum (bar line) of both kd-tree (top graph) and os-tree (bottom graph). All distributions are clustered rotated flats

[8] T. Chan. Approximate nearest neighbor queries revisited. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 352–358, 1997.

[9] C. Chang and C. Lin. LIBSVM: Introduction and benchmarks. LIBSVM can be obtained from URL: `http://www.csie.ntu.edu.tw/~cjlin/libsvm`, 1999.

[10] P. Ciaccia and M. Patella. Using the distance distribution for approximate similarity queries in high-dimensional metric spaces. In *Proc. 10th Workshop Database and Expert Systems Applications*, pages 200–205, 1999.

[11] K. L. Clarkson. An algorithm for approximate closest-point queries. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 160–164, 1994.

[12] S. Cost and S. Salzberg. A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning*, 10:57–78, 1993.

[13] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *IEEE Trans. Inform. Theory*, 13:57–67, 1967.

[14] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, 1997.

[15] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *J. Amer. Soc. Inform. Sci.*, 41(6):391–407, 1990.

[16] L. Devroye and T. J. Wagner. Nearest neighbor methods in discrimination. In P. R. Krishnaiah and L. N. Kanal, editors, *Handbook of Statistics*, volume 2. North-Holland, 1982.

[17] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* John Wiley & Sons, NY, 1973.

[18] C. Duncan, M. Goodrich, and S. Kobourov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. In *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 300–309, 1999.

[19] Christos Faloutsos and Ibrahim Kamel. Beyond uniformity and independence: Analysis of r-trees using the concept of fractal dimension. In *Proc. Annu. ACM Sympos. Principles Database Syst.*, pages 4–13, 1994.

[20] U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy. *Advances in Knowledge Discovery and Data Mining.* AAAI Press/MIT Press, 1996.

[21] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: The QBIC system. *IEEE Computer*, 28:23–32, 1995.

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

[23] K. Fukunaga. *Introduction to Statistical Pattern Recognition.* Academic Press, 2nd edition, 1990.

[24] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression.* Kluwer Academic, Boston, 1992.

[25] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. 30th Annu. ACM Sympos. Theory Comput.*, pages 604–613, 1998.

[26] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. ACM SIGMOD Conf. on Management of Data*, pages 369–380, 1997.

[27] J. M. Kleinberg. Two algorithms for nearest-neighbor search in high dimension. In *Proc. 29th Annu. ACM Sympos. Theory Comput.*, pages 599–608, 1997.

[28] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimemsional spaces. In *Proc. 30th Annu. ACM Sympos. Theory Comput.*, pages 614–623, 1998.

[29] K. I. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *VLDB Journal*, 3(4):517–542, 1994.

[30] D. M. Mount and S. Arya. ANN: A library for approximate nearest neighbor searching. Center for Geometric Computing 2nd Annual Fall Workshop on Computational Geometry, URL: `http://www.cs.umd.edu/~mount/ANN`, 1997.

[31] F. Murtagh. PCA (principal components analysis): C program. PCA program can be obtained from URL: `http://astro.u-strasbg.fr/~fmurtagh/mda-sw/pca.c` , 1989.

[32] D. Saupe, 1994. Private communication.

[33] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R$^+$-tree: A dynamic index for multi-dimensional objects. In *Proc. 13th VLDB Conference*, pages 507–517, 1987.

[34] V. Vapnik. *Statistical Learning Theory.* John Wiley & Sons, NY, 1998.

[35] K. Zatloukal, M. H. Johnson, and R. Ladner. Nearest neighbor search for data compression. (Presented at the 6th DIMACS Implementation Challenge Workshop), URL: `http://www.cs.washington.edu/homes/ladner/nns.ps`, 1999.