The ANN-tree: An index for efficient approximate nearest neighbor search

King-Ip Lin
Division of Computer Science,
Department of Mathematical Sciences
The University of Memphis,
Memphis, TN 38152, USA
linki@msci.memphis.edu

Congjun Yang
Division of Computer Science,
Department of Mathematical Sciences
The University of Memphis,
Memphis, TN 38152, USA
yangc@msci.memphis.edu

Abstract

In this paper we explore the problem of approximate nearest neighbor searches. We propose an index structure, the ANN-tree (approximate nearest neighbor tree) to solve this problem. The ANN-tree supports high accuracy nearest neighbor search. The actual nearest neighbor of a query point can usually be found in the first leaf page accessed. The accuracy increases to near 100% if a second page is accessed. This is not achievable via traditional indexes. Even if an exact nearest nearest neighbor query is desired, the ANN-tree is demonstrably more efficient than existing structures like the R-tree. This makes the ANN-tree a preferable index structure for both exact and approximate nearest neighbor searches. We present the index in detail and provide experimental results on both real and synthetic data sets.*

1 Introduction

Nowadays many database applications, such as geographic information systems and web search engines, require efficient and effective means of answering similarity queries. Many such queries come in the form of *nearest neighbor queries.* As databases are large in size, an index is usually devised to facilitate such queries. Most of them are tree-based structures similar to the B-tree, the main difference being that a multidimensional index uses multidimensional regions to divide the search space, instead of 1-D intervals used by the B-tree. While nearest neighbor search for numbers can be handled effectively by B-trees, for many applications we need to deal with multi-dimensional points or objects. There has been a lot of work on designing efficient nearest neighbor search algorithms with multi-dimensional indexes. However, most such algorithms suffer from some degree of inefficiency. For instance, even if only one nearest neighbor is required, they usually end up retrieving multiple data pages before finding the solution. This is because most search algorithms using a tree-based index are heuristic-based. If the heuristic makes a bad decision at the higher level of the tree, then the search is led down the wrong path and extra pages will be unnecessarily retrieved.

On the other hand, in many cases it is good enough to obtain a solution that is close to the actual nearest neighbor. Closeness can be measured in terms of distance or rank. This is important, for instance, in Web search engines, where a quick response time for a very good but not necessarily the best solution is preferred to a costly wait for the exact solution. Also in OLAP, an on-line algorithm can quickly display an approximate solution (which has a high chance of being correct), while continues to look for the exact solution.

With a tree based index structure, an ideal algorithm should retrieve only one page at each level and find the nearest neighbor in the first leaf page accessed. We call such an algorithm *the minimum access algorithm.* In this paper, we first explore the conditions on the index structure for the minimum access algorithm to exist. Such an index, while ideal, is impractical to implement. Hence we design the Approximate Nearest Neighbor Tree (ANN-tree), to approximate the ideal index. On the ANN-tree, the minimum access algorithm can be used to perform effective approximate nearest neighbor searches. The algorithm can be configured to access one or more leaf pages according to the requirement on accuracy. Our experiments on various data sets (both synthetic and real world) give excellent results: with the minimum access search algo-

0-7695-0996-7/01 $10.00 © 2001 IEEE

rithm configured to retrieve 1 leaf page, 94% of the time the exact nearest neighbor is found. The accuracy increases to 99% if the algorithm is configured to retrieve a maximum of two leaf pages. Moreoever, the ANN-tree supports exact nearest neighbor queries. Experiments show that the ANN-tree beats the R*-tree structure in such algorithms.

The rest of the paper is organized as follows. Section 2 outlines previous related work. Section 3 describes our proposed technique in detail. Section 4 provides experimental results and Section 5 summarizes our work and discusses future directions.

2 Related work

There has been substantial work on nearest neighbor search on multi-dimensional data. Most algorithms work with index structures like the R-tree [7, 14, 4] and follow a branch-and-bound approach to traverse the tree during the search. At each step, a heuristic is applied to choose a branch to visit next. At the same time information is collected and used to prune the search. Various algorithms differ in the order of the search. Roussopoulos et al. [12] used a depth-first approach; while Hjaltason and Samet [8] proposed a "distance-browsing" algorithm, using a priority queue to maintain all the branches that have been accessed and choose among them for the next one to visit. Other techniques modify the index structure itself to improve performance. Examples include the the SS-tree [15] (which uses spheres as bounding regions) and the SR-tree [10] (which employes the intersections of the minimum bounding rectangles and the bounding spheres).

Berchtold et al [5] proposed an alternative approach. Instead of indexing the data points, they index the Voronoi diagram [3] associated with the data set. A Voronoi diagram of a data set D is a graph that partitions the whole space into Voronoi regions. Each region corresponds to the set of points that have a certain point p as the nearest neighbor among the points in D. Hyperrectangles are used to represent an overestimate of each region. The regions are then stored in a standard index. Thus the nearest neighbor query is transformed into the point query in that index.

Both nearest neighbor and approximate nearest neighbor search have been studied in computational geometry (e.g. see [13]). Some interesting work include: the ϵ-nearest neighbor search: finding a point whose distance to the query point is at most $1+\epsilon$ times the distance of the query point to the actual nearest neighbor [1, 2]; locality-sensitive hashing techniques for nearest neighbors [9, 6].

3 The Approximate Nearest Neighbor Search Tree

As stated previously, our goal is to develop an index structure that supports nearest neighbor queries with minimum node access and high accuracy. We begin this section by defining the notion of *minimum access algorithm* for nearest neighbor queries using a tree-based index structure. We examine the conditions on a tree-based index for such an algorithm to exist. Such a structure, while ideal, is unrealistic to implement, especially in high dimensions. This motivates us to design a new structure, the ANN-tree.

In what follows, we assume that D is a data set, $\mathrm{V}_D(p)$ the Voronoi region (cf. section 2) of point p in D, and $\mathrm{NN}_D(q)$ the nearest neighbor of query point q in data set D. Each node of the index consists of k branches, $B_1, B_2, \cdots, B_k$. The bounding region of branch B_i is denoted as R_i.

3.1 Motivation

Current nearest neighbor search algorithms via tree-based index structures require traversing multiple branches of a tree. Ideally, a nearest neighbor search on an index structure should only traverse one path since we are looking for the one data point that satisfies the query, assuming no ties. In other words, the algorithm should start from the root of the tree, and at each level, choose only one branch to traverse downward, until the leaf level is reached. The critical step in such an algorithm is the branch selection. It can be viewed as a function $f()$ of the query point q and all branches of the current node: $B_1, B_2, \cdots, B_k$. It returns a value between 1 and k representing the chosen branch. Algorithm 1 describe the algorithm formally.

Algorithm 1 *MinimumAccess*
1) Starting from the root node, compute a *branch selection function* $f(q, B_1, \ldots, B_k)$ for the current node. The chosen branch is traversed downward, and the process repeated until a leaf is reached.
2) The data item in the leaf node that is closest to q is returned as the solution.

In general, $f()$ can be any function. However, for all practical purposes, $f()$ has to satisfy at least the following constraint:

DEFINITION: (well-behaved branch selection function) A branch selection function $f()$ is *well-behaved* if $\forall i, j$

such that $q \in R_i, q \notin R_j$ we have $f(q, R_1, \ldots, R_k) \neq j$.

In other words, a well-behaved branch selection function favors the branch that contains the query point. We assume that $f()$ is well-behaved in the rest of this paper. We are interested in finding conditions on tree-based index structures such that the minimum access nearest neighbor search algorithm returns the correct results. We first show a sufficiency result.

THEOREM 3.1 *For any tree-based index structure, the minimum access nearest neighbor search algorithm returns the correct result if the following holds:*

1. *In every node, any two R_i and R_j do not intersect.*
2. *At each level of the tree, the union of the bounding regions of all nodes is the entire data space.*
3. *A data point p is contained in a subtree if and only if $V_D(p)$ intersects with the bounding region of the root node of the subtree.*

Notice that a data point may appear in more than one leaf node, as $V_D(p)$ may intersect with multiple bounding regions.

Proof: Let D be the data set and q the query point. Define $f(q, B_1, B_2, \ldots, B_k) = i$ where q is contained in R_i. Condition (1) and (2) ensure that such a function is well-defined. Also assume that p is the solution of the query. That means $q \in V(p)$. Suppose L is the leaf node we reached. By the definition of $f()$, we have $q \in R_L$, where R_L denotes the bounding rectangle of L. Hence $q \in V(p) \cap R_L \neq \phi$. By condition (3), p is in L. As a result, the last step will pick p as the nearest neighbor (since p is the solution, then p must be closer to q than any other points in L). *QED*

The above theorem provides a sufficient condition for an index to have a correct minimum access nearest neighbor search algorithm. For necessary conditions, we have the following:

THEOREM 3.2 *For any tree-based index structure that satisfies condition (1) and (2) in theorem* 3.1*, a minimum access nearest neighbor search algorithm returns the correct result only if condition (3) holds.*

Proof: Notice that with condition (1) and (2) and the restriction on f, we must have $f(q, B_1, B_2, \cdots, B_k) = i$ where q is contained in R_i. Now if (3) is false, then we can find a point q in the data set such that there exists a leaf node L whose boundary region intersects with $V(q)$ but does not contain q. Pick a point in the region where $V(q)$ intersects with the boundary region of L as the query point. It can be seen that the search on this point will end up in L. However, q (the supposed solution) is not in L. *QED*

There are index structures satisfying condition (1) and/or (2). For instance, the R+-tree satisfies condition (1) while the K-D-B tree [11] satisfies (1) and (2). However, an index with condition (3) is hard to build. This is because verifying condition (3) requires knowing the Voronoi diagram, which can be hard to store or build, especially in high dimensional space.

To avoid this difficulty, we estimate the Voronoi region by a ball. Based on it, we devise an index structure, called the *ANN-tree*, satisfying conditions (1), (2), and relaxed condition (3) where the Voronoi region is replaced by a ball. More specifically, a point is contained in a subtree if and only if the ball intersects with the bounding region of the root node of the subtree. While this does not guarentee the minimum access search algorithm on the ANN-tree to find the correct nearest neighbor, our experiments show that it returns the correct nearest neighbors with very high accuracy.

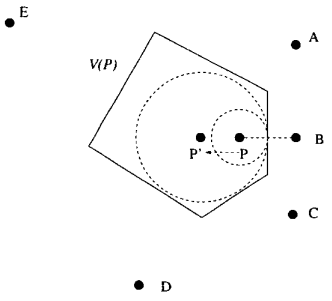


Figure 1. Estimate Voronoi region

Estimating the Voronoi Region We illustrate the process of estimating the Voronoi region by an example in 2 dimensional space. Consider a data set D, and a point p in D and its Voronoi region $V(p)$ (Figure 1). Assume b is the nearest neighbor of p. By the property of the Voronoi Diagram, it's easy to see that the ball centered at point p with radius $|pm|$ (the distance from p to m) is completely enclosed in $V(p)$ (The smaller circle in Figure 1). This is an underestimate of $V(p)$.

However, this can be a gross underestimate, especially if b is much closer to p than any other points in the data set. Thus we propose moving the center of the ball to the opposite direction of the line bp (For instance, point p' in figure 1), while using $|p'm|$ as the radius. This increases the size of the ball and results in higher accuracy. We denote the increase of the radius

by an *extension factor* f, $|p'm| = f * |pm|$. We discuss the best value for f in section 4.

3.2 ANN-tree structure and algorithms

The structure of ANN-tree is similar to the R-trees. A leaf node of the ANN-tree is of the form: $(RECT, Handle_1, \ldots, Handle_n)$. Each handle contains the following information $(ptid, BALL)$, where *ptid* is the data point, and $BALL$ is an estimate of the Voronoi region of the point referred to by *ptid*. $BALL$ can be represented by radius and center. $RECT$ is a rectangle called the cover space of the node.

A non-leaf node is of the form $(MaxR, B_1, \ldots, B_n)$ where $MaxR$ is the largest radius of all BALLs in the subtree rooted at this node. This information is used to ensure efficient insertion. Each branch B_i is in the form $(cldptr, RECT)$ where *cldptr* is a pointer to a child node and $RECT$ is a rectangle called the cover space of the child.

Each non-leaf node covers a portion of the whole space enclosed by a rectangle. To satisfy conditions (1) and (2) from the previous section, the rectangles in all branches of a node should form a partition of the cover space. This means the rectangles of different branches do not overlap with each other and the union of them is the cover space of the node.

Nearest Neighbor and Approximate Nearest Neighbor Search As the ANN-tree is an R-tree like structure, any existing branch-and-bound exact nearest neighbor search algorithms such as [12] works correctly on it.

For approximate nearest neighbor queries, we can use the minimum access algorithm discribed previously. If a higher accuracy is desired, we can follow a branch-and-bound approach to retrieve the second leaf node, the third leaf node, and so on. That is, once we reached the first leaf node, we next look for the sibling of the node that is closest to the query point, and so on. This process converges to a Nearest Neighbor Search algorithm. Our expirments show that retrieving the second leaf node yields an accuracy of about 99%. This tells us that two leaf nodes are usually all we require to locate the true nearest neighbor.

Insertion Insertion is done in two phases. First, we estimate the Voronoi region for the point to be inserted, then we insert this region into the tree. Estimating the Voronoi region requires finding the nearest neighbor of the point to be inserted.

After the estimated region is found, we create a handle to the point p. Inserting a new handle to the tree is done by examining the branches and adding it to the leaf nodes. The handle is inserted into a branch if and only if the ball intersects with the cover space of the branch. Hence, a data point can be inserted into more than one leaf node. Like the R+ tree, overflowing nodes are split and the splits are propagated to parent and possibly children nodes.

Algorithm 2 *InsertHandle*(Node N, Handle H)
1) If N is a non-leaf node, then for each branch B_i check if the R_i intersect with the $BALL$ of H. If so, call $InsertHandle()$ recursively to traverse B_i.
2) If N is a leaf node, add H to N if the cover space of N intersects with the $BALL$ of H.
3) If N overflows, call $SplitNode(N)$

Algorithm 3 *Insert* (Tree T, Point p)
1) Find the nearest neighbor p' of p in T.
2) Compute the estimated Voronoi region $BALL$ using extension factor f and create a handle H for ($BALL$, *point*).
3) Call $InsertHandle(T, H)$ to insert H.
4) Call $UpdateRegion(p)$ to update existing regions in T.

Node split Node splitting is similar to standard R-tree based indexes. The goal is to find an axis-parallel split line that partitions the group of branches into two groups. This can be done via a plane-sweep algorithm. Different R-tree varients have different ways of choosing such a line. Here since our bounding regions always cover the entire space, optimizing values such as volume of the bounding regions becomes secondary. Rather, we would like to maintain a balance between the two split nodes. Moreover, similar to R^+-tree, the split may be propagated downwards as the split line may cut through the bounding rectangle of a branch in the node. We would also like to minimize the number of such branches. Assume node N with n branches is to be split into two nodes $N1$ and $N2$ with $n1$ and $n2$ branches respectively. We have $n1 + n2 \geq n$. To balance $N1$ and $N2$, we need to minimize $|n1 - n2|$; to avoid the downward splitting, we need to minimize $n1+n2-n$. Combining the two constraints means that we would need to minimize $max(n1, n2)$.

Unlike the standard R-tree, in an ANN-tree there is a slight possibility that an internal node split may result in multiple nodes. Since a new data point may be inserted into multiple branches of a subtree, and each branch may split, it is possible for a node to be

split into multiple nodes. Thus the algorithm may have to pick multiple split lines. However, our experiments show that such splits are rare.

Algorithm 4 *FindSplitAxis* (BranchList L)
1) Collect the rectangles from all branches (handlers) in list L
2) For each axis $x[i]$, collect all the low bounds and high bounds of all rectangles, put them into an array of numbers, and then perform the following
   a. Sort the array of numbers.
   b. For each number p in the array, take $P : x[i] = p$ as a candidate partition line perpendicular to axis $x[i]$ and compute $max(n1, n2)$. If $max(n1, n2)$ of P is less than that of the candidate optimal partition line, save P as the current optimal partition line.
3) Output the current optimal partition line.

Algorithm 5 *Split* (BranchList L)
1) Call $FindSplitAxis(L)$ to get the partition line P
2) Divide L into two lists L_1, L_2. For each branch B_i in L Do
   a. If B_i is below P, add B_i to L_1
   b. If B_i is above P, add B_i to L_2
   c. If B_i intersects with P, add B_i to both L_1 and L_2 in case B_i is a leaf, or split the child nodes of B_i along P otherwise.
3) Call $Split(L_1)$ if L_1 has more branches than a node can contain.
4) Call $Split(L_2)$ if L_2 has more branches than a node can contain.
5) Create a node from each resultant list and output the nodes.

Updating Regions Inserting a new data point p results in changes in the Voronoi regions of some existing data points. Thus we need to change (usually shrink) the estimated Voronoi regions of those points.

Notice that only existing data points that have p as their nearest neighbor will need to change their regions. This is because estimations are done using only nearest neighbor information. This implies we need to traverse the tree to find nodes that contain the affected points. This can be sped up by observing the following lemma.

LEMMA 3.1 *For any point p in the index structure, let r_p be the radius of the ball estimating $V(p)$. Then a point q is the nearest neighbor of p if and only if $|pq| \leq 2 * r_p/f$, where f is the extension factor and $|pq|$ the distance between p and q.*

Proof: Refer back to figure 1. Notice that $2 * r_p/f$ is the distance from p to its nearest neighbor in D. So, for any point q to be the nearest neighbor of p, we must have $|pq| \leq 2 * r_p/f$ and vice versa. *QED*

Thus at each node, we maintain the largest radius of all the balls in its subtree, enabling us to prune the search effectively.

Algorithm 6 *UpdateRegion* (Node N, Point p)
1) If N is a non-leaf node, check if distance between p and the bounding rectangle of N is less than $2 * MaxR$. If so, recursively call UpdateRegion() for all branches.
2) If N is a leaf node, then for each point update the approximate Voronoi region if p is the nearest neighbor of it.
3) Update the $MaxR$ field for the nodes traversed on the way back up.

Algorithm 7 *Delete* (p)
1) Starting from the root go down the tree along the branches containing p. Retrieve the estimated region $BALL$ of p from the leaf node reached.
2) Go down the tree to locate all leaf nodes that intersect $BALL$ and delete all entries containing the $BALL$. Update $MaxR$ on the way back up.
3) Remove the points that take p as their nearest neighbor.
   a. Starting from the root, a branch is traversed if and only if the distance from p to the cover space of that branch is less than $2 * MaxR$ in that branch.
   b. Once a leaf node is reached, then for each point x in it, remove x from the node if $d * f \leq r$, where f is the extension factor, d the distance from p to x, and r the radius of $BALL$ for x.
4) Reinsert the points removed in previous step using the *Insert* algorithm.

Deletion Deletion in the ANN-tree is similar to insertion. First, the data point p to be deleted is located and the estimated Voronoi region $BALL$ retrieved. A range search is done to locate all nodes that intersect with $BALL$. These nodes must contain p according to property (3) in theorem 3.1 and hence p is deleted from these nodes now. After that we need to update regions

of points that have p as their nearest neighbor. Here we locate the points, simply remove them from the leaf nodes, and reinsert them using the Insert algorithm.

4 Experimental results

This section presents the results of our experiments. We measure how well our index structure performs against the R*-tree. We choose the R*-tree because the ANN-tree is a dynamic structure, so comparison with a fellow dynamic structure is fair. We measure both accuracy (how often our approximate algorithm is useful) and efficiency (how much time we save using an approximate algorithm). We also compare the results from exact nearest neighbor search algorithms.

We implemented both structures in C++, and ran our tests on a machine with 2 500-MHz Pentium II processors and 512 MB RAM under SCO UNIX. We experimented on both synthetic and real world data sets. The real data set is described in subsequent paragraphs. For synthetic data, we generated data sets of various sizes and dimensionalities with uniform and normal distributions. The results with both distributions are similar, so in the rest of the section we mainly present the results from uniformly distributed data.

Accuracy The first thing we measure is the accuracy of the approximate search. We ran experiments on the ANN-tree and recorded how often the actual nearest neighbor was retrieved. As a comparison, we modified the depth-first nearest neighbor algorithm of [12] such that the search stops whenever the first leaf node is reached. This means that we ran the same minimum access search algorithm on the R*-tree but with the branch selection function $f()$ set to return the branch with bounding region closest to the query point.

Figure 2 shows the results. We ran the experiments over various data sets. For each data set we ran 100 queries and averaged the results. The figure shows that the ANN-tree exhibits a precision of approximately 95%. The same accuracy is maintained with various data set size, meaning our algorithm scales up very well. On the other hand, the R*-tree's precision is significantly lower than ours, and is decreasing steadily when the data set size increases. Thus we can see that the ANN-tree is a much preferred structure in terms of fast approximation nearest neighbor searches.

We also examined the case of retrieving two leaf pages from the ANN-tree. The precision rises to approximately 99%. Moreover, only 30% of time a second leaf page access happened. This makes the average number of data page access to be 1.3.

Dimensionality	2	4	6	8	10
Accuracy (%)	95.1	95.3	93.7	93.3	93.1

(a) # of data: 50,000. Extension factor : 1.2

Extension factor	1.0	1.1	1.2	1.3
Accuracy (%)	93	93	94.5	93
Extension factor	1.4	1.5	1.6	1.7
Accuracy (%)	94	91.5	90.5	88.6

(b) # of data: 50,000. Dimensionality : 4

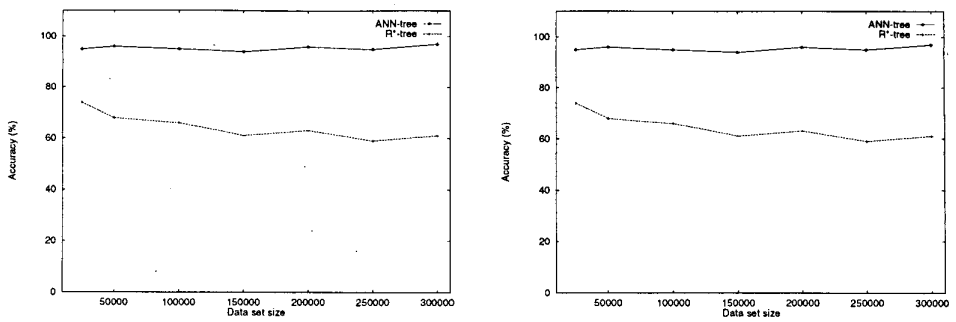
Table 1. Comparison of accuracy using different dimensionalities and extension factors

The next set of experiments tests the ANN-tree performance over various dimensionalities of data using different extension factors respectively. The results are shown in table 1. It can be seen that the ANN-tree is robust over different dimensionalities and extension factors. The extension factor is a parameter that one may tune. The optimal value may be dependent on the distribution of the data. The results show that the performance of ANN search is not very sensitive to the extension factor and 1.1 to 1.4 is a good range for it. However, a bigger extension factor produces a bigger estimated region and hence causes a higher chance for a point to be inserted into multiple nodes. This will results in an increased tree size and therefore reduces the effective branching factor of the tree. From Table 1 we can see that the accuracy does tail off when the extension factor becomes large. This suggests we are correct in underestimating the Voronoi region.

# of data(x 10^4)	5	10	15	20	25
% wrong answers	3	5	5	4	6
Rank of wrong answers	2	2-4	2-3	2	2-3
Error of distance	0.07	0.14	0.09	0.13	0.13

Table 2. Statistics for approximate anwers of the ANN-tree

We also want to measure the quality of the retrieved points if they are not the nearest neighbor of the query point. Thus we examined the queries that did not return the correct answers. Table 2 shows the statistics of such approximate solutions. The table shows that the approximate answers are in general the second or third nearest neighbors. Moreover, the average relative



Uniformly distributed points Multi-normal distributed points

Dimensionality = 4, f = 1.1, Page size = 6K

Figure 2. Comparison of accuracy of various data set size

error in terms of distance is around 10%. This seems large, but one has to remember that most of the time the points retrieved are the second nearest neighbors, so the error is inherit to the data set. Thus we can say in cases where the ANN-tree failed to find the nearest neighbor with one leaf access, it can still find an excllent approximation.

Efficiency Obviously, the approximation algorithm on ANN-tree is the fastest possible (in terms of number of pages accessed). We want to measure actual speed-up to see the trade off between the efficiency and accuracy. We also compare the efficiency of the exact nearest neighbor algorithm on both the ANN-tree and the R*-tree.

Figure 3 shows the results. We measured both the number of leaf nodes and the total number of nodes. We can see that the exact nearest neighbor algorithm for R*-tree retrieves about 3-4 leaf pages for some decent data size. That means the savings in ANN-tree can be threefold to fourfold. This shows that a high accuracy approximate algorithm is viable. Also the approximate algorithm requires half the number of node access (Figure 3 (b)).

Moreover, the ANN-tree outperforms the R*-tree in exact nearest neighbor searches by around 30%. The savings scale up well. We also tested on a data set with 1,000,000 points, and we still obtain a 30% savings. Thus the ANN-tree can be beneficial for both approximate and exact nearest neighbor queries.

Results from a real data set We also ran tests on a real data set. This was obtained from the US National Mapping Information web site (URL: `http://mappings.usgs.gov/www/gnis/`). It contains populated places in the USA, represented by latitude and longitude. We ran experiments on this set, randomly picking 10,000 to 100,000 points as the database, and picking points from the set as test queries.

# of data (x 10^4)	1	2.5	5	7.5	10
1 leaf access	95.2	96.8	96.6	97.2	97.2
2 leaf access	98.4	98.6	98.8	99.0	99.4

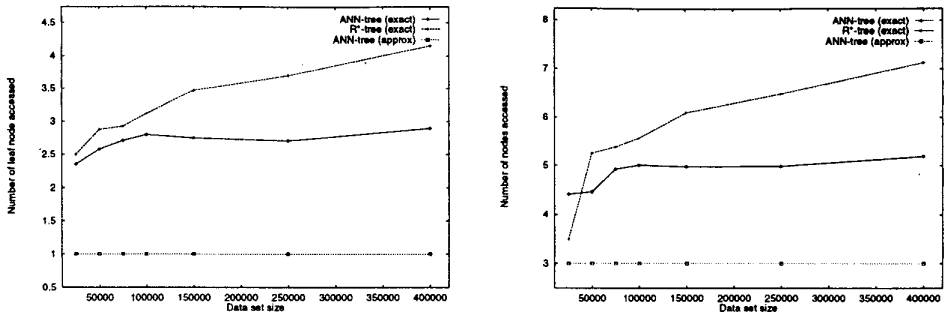
Table 3. Accuracy (in %) for the real data set

Table 3 shows the accuracy results for this data set. Once again, we achieve an accuracy of 95% if we access only one leaf node. If we allow the algorithm to access a second leaf node, the accuracy rises to 99%.

We also examined the 1-5% of the incorrect results. Again, most of the points retrieved are either the second or third nearest neighbors. However, there are a few cases where a "farther away" neighbor is found (like the 10^{th} nearest neighbor). These are the cases where the query comes from an outlying region of the data set and the nearest neighbor is very far away from the query point. However, for such query points, the ratio between the distance of the retrieved point to the query point and the distance of the nearest neighbor to the query point is very close to one. Thus the retrieved points are still representative.

5 Conclusion and future work

In this paper, we examined the nearest neighbor and approximate nearest neighbor searches using tree-based indexes. We explored the conditions for an index to perform optimally (in terms of node access). We also introduced the ANN-tree, an index structure built on those principles. Our experiments show that



(a) Leaf nodes (b) All nodes

Figure 3. Number of nodes accessed for both approx. and exact NN queries

the ANN-tree provides excellent performance on nearest neighbor queries, both in accuracy on fast approximate nearest neighbor searches, and in efficiency on exact nearest neighbor searches.

In section 3 we examined sufficient conditions for an index to achieve minimum node access nearest neighbor search. An important part of our future work is to nail down the necessary conditions for such indexes to exist. This will benefit future research on index structures and provide a framework to compare indexes.

Another direction to explore is to utilize the ANN-tree to perform k-nearest neighbor queries. Many applications require multiple nearest neighbors. We intend to look at how the ANN-tree can be generalized.

References

[1] S. Arya and D. Mount. Approximate nearest neighbor searching. In *Proc. of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 271–280, 1993.

[2] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proc. of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, 1994.

[3] F. Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, Sept. 1991.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: an efficient and robust access method for points and rectangles. *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pages 322–331, May 1990.

[5] S. Berchtold, B. Ertl, D. A. Keim, H.-P. Kriegel, and T. Seidl. Fast nearest neighbor search in high-dimensional spaces. In *Proc. of the 14th IEEE Conference on Data Engineering*, 23–27 Feb. 1998.

[6] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 1999 International Conference on Very Large Databases*, pages 518–529, 1999.

[7] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Proc. of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, June 1984.

[8] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999.

[9] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 604–613, 1998.

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

[11] J. Robinson. The K-D-B-Tree: a search structure for large multidimensional dynamic indexes. *Proc. of the 1981 ACM SIGMOD International Conference on Management of Data*, pages 10–18, 1981.

[12] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. *Proc. of 1995 ACM SIGMOD International Conference on Management of Data*, May 1995.

[13] J. Sack and J. Urrutia, editors. *Handbook on Computational Geometry*. North-Holland, 2000.

[14] T. Sellis, N. Roussopoulos, and C. Faloutsos. The $R+$ tree: a dynamic index for multi-dimensional objects. In *Proc. 13th International Conference on Very Large Databases*, pages 507–518, England, Sept. 1987.

[15] D. A. White and R. Jain. Similarity indexing with the ss-tree. In *Proceedings of the 12th International Conference on Data Engineering*, pages 516–523, Feb. 1996.