Optimizing Search Strategies in k-d Trees

Neal Sample[†], Matthew Haines[‡], Mark Arnold[‡], And Timothy Purcell[†]

[‡]Department of Computer Science
University of Wyoming
Laramie, WY 82070
{haines,marnold}@cs.uwyo.edu

[†]Department of Computer Science
Stanford University
Stanford, CA 94305
{nsample,tpurcell}@cs.stanford.edu

Abstract: While k-d trees have been widely studied and used, their theoretical advantages are often not realized due to ineffective search strategies and generally poor performance in high dimensional spaces. In this paper we outline an effective search algorithm for k-d trees that combines an optimal depth-first branch and bound (DFBB) strategy with a unique method for path ordering and pruning. Our initial method was developed for improving nearest neighbor (NN) search, but has also proven effective for k-NN search and approximate k-NN classification.

Introduction

Search is an important technique in many fields, and various search methods are continually being refined. There are several types of search that are of interest in different disciplines. Nearest neighbor (NN) search is important to many case-based reasoning (CBR) as well as various classification and matching problems [9]. Approximate nearest neighbor search is important to many AI systems, and in systems where there is an acceptable trade-off between exact answers and performance. K-NN and other multivariate range queries play critical roles in database retrieval, classification problems, and clustering problems.

Various methods have been proposed to solve search problems, including hashing and indexing, various types of trees, and many hybrid and novel approaches. Proposed tree solutions alone include k-d, B+, R+, BBD, VAMSplit k-d, and other variants [2, 3, 5, 6, 7, 8, 9]. Tree-based search strategies are popular for many reasons, including, for n cases, O(log n) search and insertion time, O(n log n) construction time, and reasonable space requirements. Tree structures also allow for dynamic insertion of additional elements and simple formulation of range queries.

In this paper, we outline essential optimizations for performing various types of searches on a k-d tree structure. The first optimization, tracking nodes, provides an efficient pruning technique that allows for effective searches of high-order trees. The second optimization, depth first branch and bound (DFBB) search, provides a tight bound on the search depth. The third optimization, path ordering, is used in conjunction with DFBB to select a search order that is optimal. The fourth optimization, the addition of an additional information structure at tree build time, allows for further pruning with a Bounds-Overlap-Ball (BOB) test.

Recent algorithms have been proposed that have reasonable performance on a subset of the searches outlined above, but have significant limitations that prevent their extension to all domains. Recently, Nene and Nayar proposed a method for effective NN search in high dimensions [1]. Their method, while providing a good search time, generates a static structure that prevents the insertion of additional elements without completely rebuilding the tree. Such a solution would not be appropriate for many problems, including growing case bases and expanding databases.

The k-d tree has been analyzed extensively, but not exhaustively, since its introduction in 1975 [6]. Friedman, Bentley, and Finkel presented good search and construction algorithms for NN searching with k-d trees as early as 1977 [2]. More recently, Arya and Mount have presented refined search tactics which have been especially effective for approximate searches [3]. The refinements we present are effective for all of the search types outlined and provide the maximum speed-up for exact NN queries.

Traditional Search

Many authors have examined the best ways to construct k-d trees. We do not purport to improve the construction method, but present our simple method for future comparisons. In fact, our search and pruning techniques work equally well with a tree that has been built using one of the bulk-loading techniques [5, 9] as with our incremental construction technique, outlined below.

We start with a randomized set of elements of dimensionality d, which are added to the tree one at a time. The only parameter passed to the tree constructor is a density value, δ, also referred to as “bucket size.” Each leaf (or "bucket") is stored as an unordered list of elements. When a leaf (or “bucket”) exceeds δ elements, it is split into two new leaves. The split process analyzes the elements in a leaf to determine the dimension with the greatest local variance. The nodes are then split based on the mean value of that dimension. Although splitting on the median might yield a more balanced tree, the mean proves to be just as effective as a determinant for splitting and has the advantage of being able to be computed in $O(\delta)$ time, compared with $O(\delta \log \delta)$ for the sort required to find the median. In short, this tree construction method is quick and dirty.

Over many insertions and deletions into the k-d tree using our simple insertion algorithm, it may be necessary to rebuild the tree structure to maintain near optimal splitting. We leave the subject of optimal k-d tree insertion techniques as a separate investigation as we are primarily concerned with optimizing searches within a given tree. However, tree imbalance is known to negatively affect search time, and we expect results to be improved with better tree building methods [10].

The underlying assumption is that the tree will be split along the most discriminating dimensions, at least for each leaf. Better bulk-loading construction methods guarantee balance and discrimination by considering the entire data set in the first split, and making an even division of that set. Each split set is then considered until no set is larger than δ. Testing against such an optimally built tree does not consider the effects of adding elements, and we believe that a primary reason to use a tree structure is the ability to add elements.

Search queries generally begin with the construction of a “clipping window,” which defines a region of the tree space to search. For example, a three-dimensional range query in a three-dimensional space uses a hyper-rectangular region. The enclosed region is partitioned by upper and lower bounds for each dimension. Such a clipping window can be seen in figure 1.

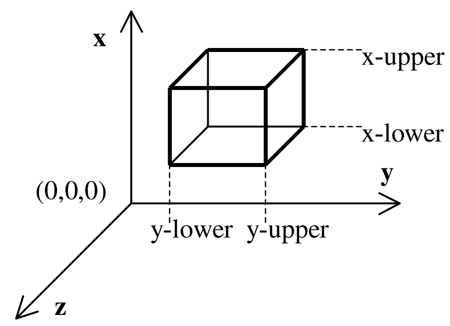

figure 1. Clipping window example

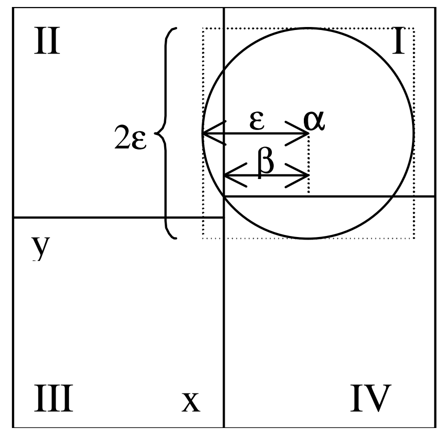

figure 2. Clipping window excess

A traversal of the tree, returning the elements in leaves that intersect the clipping window, will provide an answer to the query. Nodes that lie outside the clipping window are generally pruned as a leaf is searched. K-d trees are very effective for range queries as they are typically formed as spaces bounded by planes orthogonal to dimensional axis. Since the tree is built by dividing the search space along orthogonal planes, it is a natural structure for these queries.

Nearest neighbor searches are a different matter. When k-d trees are used for NN search in low dimensional spaces, they are quite effective. The reason for their effectiveness is that in low dimensions, very few extra cases need to be examined to find the nearest neighbor. However, in high dimensions, the search space must be carefully pruned to prevent the search from degrading to an exhaustive search. We will discuss the reasons (and solutions) to the dimensionality problem in the next section. We discuss the use of k-d trees in **k**-NN search in the Additional Findings section.

Tracking Nodes for NN Search

In a one-dimensional space, an NN search of a k-d tree works much like a typical binary tree, except that the leaves contain multiple elements. A search of a one dimensional k-d tree proceeds as follows. First, ε, an initial bound on the distance from the nearest neighbor is established. If only cases of a certain similarity are usable, ε may be user defined. Additionally, if cases have different similarity requirements along different dimensional axes, a user defined clipping window may be used.

If there are no restrictions, and an exact nearest neighbor is required, the selection of an arbitrary ε is a trivial task. Selection of a *good* ε is a different matter. To find an adequate starting ε, determine which leaf the unique node (α) would be placed in if a **tree.insert(**α**)** were performed. (We will use "α" to denote a node, not already in the tree, whose nearest neighbor we are trying to find.) Search that leaf for the closest node to α and make the distance from the closest node to α the initial ε. With varying frequency, based on dimensionality and spatial density, this initially inspected case is often the exact NN in question.

Given a unique node, α, and an initial value for ε, we form a clipping window. The simplest clipping window is for a one-dimensional space. The "upper" and "lower" values of the one-dimensional clipping window are set to α+ε and α-ε, respectively. A traversal of the tree will yield all nodes within α±ε, and a simple search of these marshaled nodes will yield the nearest neighbor. If the tree density were 1, no more than 3 nodes would ever have to be examined in a one-dimensional search. If the tree density were δ, no more than 3δ nodes would have to be examined.

The problem is harder in two dimensions. The reason is well understood. The clipping window is formed from values that lie normal to the dimensional axes. Given an arbitrary ε, the clipping window in two dimensions contains $4\varepsilon^2$ units of the two-dimensional space, similar to the enclosed dotted region in figure 2. However, the optimal search space lies within the circle of radius ε immediately around α, which covers approximately $\pi\varepsilon^2$ units of the two-dimensional space. Thus, the actual clipping window covers about ($4\varepsilon^2/\pi\varepsilon^2$) 1.27 times the *optimal* search area in a two-dimensional search.

In three dimensions, the clipping window covers $8\varepsilon^3$ units of space, while the *optimal* search area is about $4.19\varepsilon^3$ units ($4/3\pi\varepsilon^3$). This means that in three dimensions, the clipping window contains almost twice the volume of the optimal search space constrained by ε.

As dimensions grow, these extra "corners" formed by the hyper-rectangular region enclosed by the clipping window grow *exponentially* faster than the hyper-spherical region enclosed by the optimal search window. As dimensionality grows, the intersection of this extra area in the clipping window dominates the search process. The corners outside the hyper-sphere take up more space than the area within it in just 4 dimensions.

The solution to this problem comes in the form of what we refer to as a "tracking node." Because of the orthogonal planes used to build the tree, search is fast, but little information is stored at each branch of the tree. It is this absence of information that the tracking node solves for. Other complicated methods have been proposed for tracking, primarily variants of the *Bounds-Within-Ball* (BWB) test [2, 9]. Our tracking nodes achieve the same effect as the BWB test with just one simple structure and a single comparison at each branch. Also, it is our conjecture that the code for the tracking node solution is more intuitive.

The tracking node is an object of size O(d), as it only has to one distance for each dimension. The tracking node keeps track of the total distance already covered by the clipping window on a particular path in the k-d tree. With this extra information, the "corners" outside the hyper-sphere can be easily pruned from the search space.

For instance, in figure 2, the clipping window intersects all four quadrants. The circular area constrained by ε only intersects with quadrants I, II, and IV. There is no need to search the nodes in quadrant III.

As noted before, these overlapping areas grow exponentially with dimensionality. As such, the savings incurred by *not* searching these corner regions grows exponentially with δ. The tracking node is what allows the efficient pruning of the superfluous leaves.

To show how this pruning is achieved, assume that the root of the tree contains a value for the x-axis that divides the tree into two main branches, as seen in figure 2. Quadrant I is searched to find the initial value of ε. Quadrants II and IV are still likely to contain the nearest neighbor to α, but there is a constraint on the maximum distance *into* quadrants II and IV that the neighbor may lie.

In figure 2, β is the distance from the unique case α to the spatial division at **x**. By crossing the dividing line at **x**, the closest neighbor that may still be a candidate in quadrant II must be within ε-β of the division **x**. That means that the value of ε "used up" in the left sub-tree's tracking node is β. The value of ε "used up" in the right sub-tree's tracking node remains zero.

During each recursive call, the tracking node is updated to store the total distance from the unique node α that the minimum case must be. In this case, even though the clipping window intersects quadrant III of the space, the tracking node indicates that it does not need to be searched because no elements in the quadrant can be within ε of α.

The value contained in the tracking node allows rapid pruning of the search space to something more similar to the true region within ε of α. The tracking node only prunes nodes that are beyond ε. As noted previously, the amount of space pruned by the tracking node increases with dimensionality. This is expected since the excess volume of the clipping window also grows with dimensionality.

The initial values of the tracking node are set to the initial values for each dimension of the novel case, α. (Recall, α is the whose NN we are trying to find.) The values of the tracking node are only updated when a left-hand cutting plane lies to the left of the initial value of α, or a right-hand cutting plane lies to the right of the initial value of α. This ensures that the tracking node only prunes the proper nodes.

DFBB Search

While the tracking nodes prune the leaves that lie on the external boundaries of the search space defined by ε, the order in which the internal nodes are searched can be used to the advantage of the process.

Depth first branch and bound (DFBB) works best (relative to IDS and A* methods) when the branching factor of the search space is low but the solution density is high [4]. The beauty of the k-d tree is that the actual branching factor is always two because each internal node only splits on a single dimension. The *effective* branching factor generally is much lower. Like the travelling salesman problem (TSP), the axial division lines in a k-d tree are unevenly spread, like cities are spread in the TSP [4]. Also, every leaf node is a solution as each leaf node contains at least 1 NN candidate.

The basic premise of the DFBB search is to find an initial solution using a depth-first technique, then search the rest of the space in a depth-first manner, never going beyond the current best solution depth. Each time a better solution is found, a new limit is placed on the search depth and the search continues. We apply this theory to the k-d tree, noting that *tree depth* and *solution depth* (distance of closest neighbor) are not necessarily the same. (E.g., the best solution depth (NN distance) in a tree of depth 5 could be in a branch of depth 4 in that tree.)

We can look at the initial value for ε as the initial solution depth. There is no reason to consider any node "deeper" (farther from α) than this initial bound, as there is at least one node closer to α (the node that determines ε). There are two elements to consider for successful DFBB search of a k-d tree: revising ε (the solution depth) , and search order.

Epsilon revision for NN search is a simple task. Whenever a neighbor is discovered that is closer to α than the current distance ε, this new distance becomes the new ε. By changing a global ε, there can be a dynamic *contraction* of the clipping window as well. This constrains the search space, but what happens to search nodes that are several levels farther up in the recursive search procedure? Do they need to be updated too? The answer is no.

Because the tracking node contains the *minimum* distance a neighbor will be in a given tree branch, when ε is globally updated, a simple (*tracking_node_distance* > ε) test will prune the now unnecessary recursive calls from the stack.

By dynamically contracting the clipping window and pruning recursive search calls based on tracking values, considerable time can be saved. These savings are illustrated in figure 3 of the Results section.

Simple contraction of the clipping windows has dramatic results, as has been shown previously. But a critical aid do DFBB search, appropriate path ordering, has been neglected in the same studies. The critical code fragment presented in [9] demonstrates an algorithm for ordering that examines tree leaves in a static order. The algorithm operates independently of freely available information from tracking nodes. Such information can be used to intelligently order search.

Path Ordering

The other important factor in DFBB search, in addition to revision of solution depth, is the search order. In a naïve recursive search, all the "left" branches would be traversed, then all the "right" branches. This naïve approach has the effect of searching a large part of the space that is unlikely to contain promising neighbors, because it searches the outermost regions of the hyper-sphere first, whose nodes are most likely to be far from α.

A simple Monte Carlo approach, stochastically choosing a path at each opportunity, does not yield significantly better results than naïve search. To *maximize* effectiveness, the value of ε has to be revised downward as soon as possible.

How can the search be easily ordered without further information stored at each of the tree nodes or building a complicated priority structure at search time? By cleverly using the data provided by the tracking nodes, even a recursive algorithm, with no extra knowledge stored in the tree (and without expensive volumetric calculations), can quickly search the most promising leaves before moving to the outer surfaces of the search space.

Here is the simple algorithm for doing so:

```
Search-Tree (tree_node, tracking_node_left, tracking_node_right)
{
    if (tree_node == leaf)
        check-for-candidates(tree_node)
    else {
        tracking_node_left  = intersection(tree_node->left)
        tracking_node_right = intersection(tree_node->right)

        if (tracking_node_left < tracking_node_right) {
            if (tracking_node_left < epsilon)
                Search-Tree(tree-node->left, tracking_node_left,
                            tracking_node_left)
            if (tracking_node_right < epsilon)
                Search-Tree(tree-node->left, tracking_node_right,
                            tracking_node_right)
        } else {
            if (tracking_node_right < epsilon)
                Search-Tree(tree-node->left, tracking_node_right,
                            tracking_node_right)
            if (tracking_node_left < epsilon)
                Search-Tree(tree-node->left, tracking_node_left,
                            tracking_node_left)
        }
    }
}  //end Search-Tree
```

What this algorithm accomplishes is a search that is depth first, but ordered by desirability. The "internal" leaves nearest to the novel case α will be searched first. This generally allows for the earliest revisions of ε and thus the maximum possible contraction of the search space.

For example, once again consider the case in figure 2. The most logical quadrant to search first would be quadrant I. Does the `Search-Tree` algorithm guarantee that quadrant I will be searched first? It does.

Assuming once again that the first division of the k-d space was drawn along the **x**-axis, the tracking value of the left node will again be β. The tracking value of the right node will be zero, as the value of the **x** dimension is less than (or "left of") the value of the **x** dimension in α, so the tracking value is not revised.

Quadrants I and IV are now immediate candidates for search, and II and III can be considered only when we pop up the recursion stack. Given the dividing line along axis **y**, quadrant I will have a tracking value of zero still, but quadrant IV will have some positive value (>0). So, quadrant I will be searched first.

After the quadrant I search pops off the stack frame, quadrant IV will be searched. Once this is searched, we return to quadrants II and III. Since the value of dimension **y** used to split quadrants II and III is greater than α's value along the **y** dimension, quadrant II will still have a tracking value of β, but quadrant III will have a tracking value of β plus some positive amount. This means quadrant II will be searched next.

Finally, the last potential search is of quadrant III. Even without a revision of ε, the tracking value of quadrant III is greater than the initial ε and will not be considered in the search.

With this ordering by tracking value placed on the k-d tree's nodes, the most internal nodes are searched first and ε is revised downward at the earliest possible time. Of course this is a heuristic method for ordering the search, and the knowledge is not perfect, but it is superior to its non-heuristic counterpart, as can be seen in figure 5.

This algorithm simpler than [9], and provides a critical advantage. By ordering search, fewer Bounds-Overlap-Ball tests will have to be performed and more pre-case-inspection pruning occurs, as shown in the results section.

Bounds-Overlap-Ball (BOB) Test

We save the description of the BOB for last, as it is only possible to perform when additional information is embedded when the tree is constructed. As such, it may not be applicable to many trees built by other methods, but it is a simple task to add BOB information to existing trees and construction methods.

The time required to add the BOB information when the tree is built is a constant, as will be shown. Since this is true, the information is essentially free, and should be used whenever possible. If it is not possible to put such information into an tree existing structure, performance will not be optimal, but it can still be very good.

The information required to perform the BOB test consists of two objects in a leaf node, both of the same type as the data stored in the leaf. For example, if the tree held structures consisting of two real numbers, an **x** and a **y** coordinate, two such structures would be required in each leaf node. One structure contains the maximal values for the **x** and **y** fields of the elements in a given leaf. The second structure contains the minimal values for the **x** and **y** fields of the elements in a given leaf.

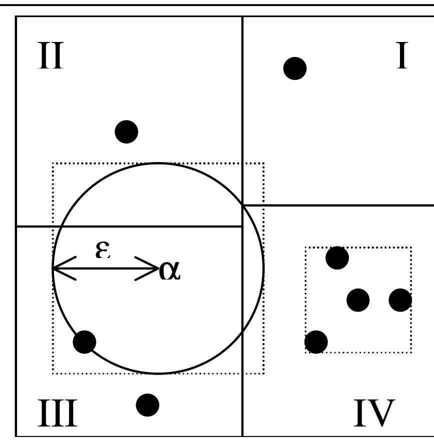

figure 3. BOB test

Before inspecting the contents of a leaf, and incurring the cost of δ comparisons (recall, there are up to δ elements in a given leaf), a single BOB test will indicate whether there are likely candidates in the leaf. Clearly, this test is more productive when the leaf density (δ) is high, and saves more time as the cost of similarity comparisons increase.

Figure 3 demonstrates how the BOB test works, in principle. The case closest to α lies quadrant III. That case defines a circle, with radius ε, around α. All other potential candidates to be the nearest neighbor to α must lie within this region.

This region extends into quadrant IV, but the cases in quadrant IV do not need to be examined. The BOB test determines whether or not there is an intersection between the region delimited by ε in quadrant III (with dashed lines) and the region around the cases in quadrant IV (also with dashed lines). If there is an intersection, at least one possible candidate lies in quadrant IV, otherwise it the cases in quadrant IV can be neglected.

Results

To isolate the contributions of our four optimization strategies (DFFB, search ordering, tracking nodes, and BOB), we performed a battery tests.

The data set used to build the k-d trees consisted of 200,000 elements. The elements of the set were uniformly distributed and normalized to the range 0.0-1.0 in each dimension. The test set was similarly constructed, and had 1000 elements that were used to calculate search times.

Each test was performed with 200,000 elements to build the k-d tree. 1000 searches were performed and the total time and number of nodes examined were recorded. The tests were also balanced for dimensionality. 200,000 points were used to build a two-dimensional tree, a three-dimensional tree, and so on, up to and including a 25-dimensional tree.

The first comparison is of all four optimizations to a wholly uniformed search. The results are shown in figure 4. At 11 dimensions, the uninformed search method examines over 50,000 nodes, or 25% of the entire search space. The optimized DFBB search can effectively consider an additional six dimensions before degenerating to that level of performance. It appears that even the best search method is still futile for high dimensions, but is somewhat better than the

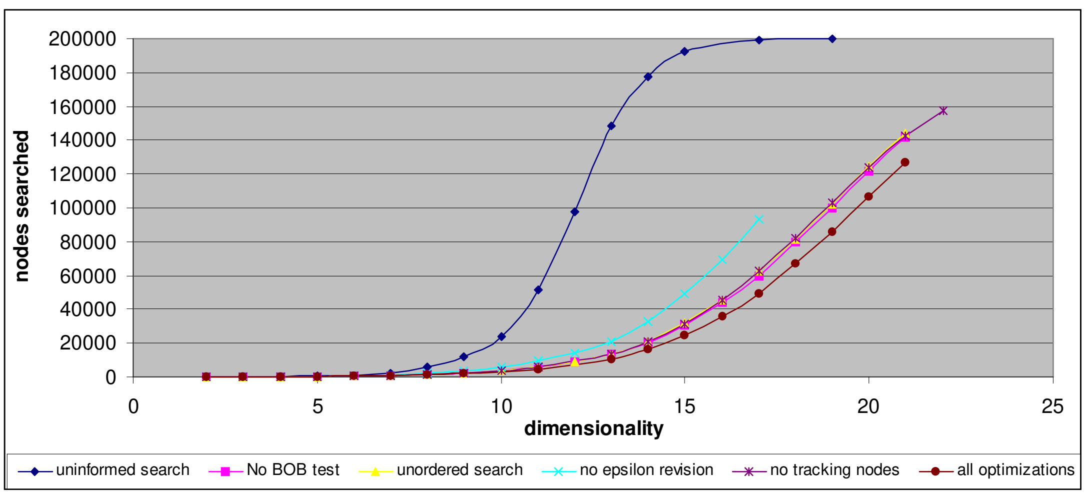

figure 4. Comparison of various search methods

wholly uninformed search strategy.

Before analyzing the other test results, a word about dimensionality. There are several reasons that the number of nodes searched increases with an increase in dimensionality. This "curse of dimensionality" can be averted with certain permutations of the initial data set that lower the dimensionality of the problem. These statistical methods include convolving and eliminating variables with high covariance, and other techniques that reduce the dimensionality of the data set.

The issue of dimensionality is not intrinsically insurmountable by the search techniques of the k-d tree. High dimensionality generally degrades k-d search times because of the depopulation of the search space (i.e., 100,000 elements in a 10 dimensional space are generally much more dense than 100,000 elements in a 15 dimensional space). A progression of the final ε values sheds light on the problem. In the two dimensional search space, the average distance of the nearest neighbor is <0.001. In twenty five dimensions, the average nearest neighbor is 0.954 away from the unique case, α. The density of 200,000 elements in two-dimensions (ranging from 0.0-1.0) is far higher than a similar 200,000 elements in a 25 dimensional space.

In practice, the problems of high dimensionality are often not as extreme as we have demonstrated using a fixed number of points over an increasing range of dimensions. In many problems, the number of points in a space increases rapidly with dimensionality. The increase in the population density of high dimensional spaces diminishes the effects of dimensionality and brings nodes searched closer to the ideal of O(log N), with a large constant that is exponential in dimension, δ [7].

Regardless of the ability to reduce the dimensionality of a data set, it is crucial to be able to effectively search the highest possible dimension before resorting to reduction techniques.

The second comparison we make is between the optimized process and the process without the tracking nodes. The search with tracking nodes is also shown in figure 4. The discerning observer will note that the graph line nearly identical to the "unordered search" and "no BOB test" lines. This result indicates that the omission of any of the three optimizations (ordering, tracking nodes and BOB test, but *not* epsilon revision) will yield a significant speedup, but that all four methods are required for maximal gain.

An omitted line, identical to the wholly uninformed search, was achieved by leaving out the tracking nodes and BOB test. Without either test, no additional pruning of leaves (by tracking nodes) or within leaves (via BOB) is possible and the search is not better than the worst method shown here. The tracking nodes facilitate a reasonable search in spaces above 10 dimensions. Without the tracking nodes, the performance degrades rapidly. As noted in the section on the Bounds-Overlap-Ball test, it may not be possible to add such information to existing trees. In such cases, omission of the tracking nodes is enough to render search as ineffective as the wholly uninformed search.

While search *without* the tracking nodes and BOB test degenerates to the performance of the wholly uninformed search, the DFBB ordering component and ε revisions both contribute to significant speedups.

Figure 4 demonstrates the effect of the ε revision. Epsilon revision is possible in almost any search of a k-d tree, but the effect can be magnified by reducing ε as early as possible.

Assuming an existing tree structures not suited to the BOB test, search without the ε revisions averages 67.4% more nodes examined, in dimensions 5-21, and 50.5% over the entire range from 2-25.

The search ordering did not provide as much gain as we had hoped, but it still provides an additional reduction in search time. As figure 4 shows, the DFBB search pattern is better than the naïve search order. We found that stochastic ordering of the search is no better than the naïve method.

The overall time reduction for DFBB search (measured in nodes examined) compared to a naïve search method is a remarkable 79% average over dimensions 6-19, with a 67% increase over dimensions 1-19, and a lower average considering the entire range (when all searches degrade to looking at almost all cases).

However, the significance of this paper comes in comparison to the *next best* methods, wherein only one optimization was omitted. The modest average speedup of 20% in dimensions

10-20 achieved by the combination of all techniques demonstrates that all components must be present to maximize search gains in high dimensional spaces. In dimensions 2-21, there was an average of almost 15% fewer cases inspected.

Almost any search strategy could employ some sort of DFBB reductions by revising ε downward, but figure 4 shows a clear savings from revising ε downward at the earliest possible times.

Additional Findings

Our search algorithm tracked various factors in addition to the number of nodes examined to discover the nearest neighbor. Two important statistics further demonstrate the necessity of a dense search space in high dimensions. We counted and tracked the percentage of cases for which the actual nearest neighbor was discovered in the first leaf searched, and further the number of revisions to ε.

First, it is interesting to note that the growth rate in number of cases examined was indeed close to $O(1.56594^{\delta})$, as predicted in [7]. In dimensions 2-19, the growth rate was $\sim 1.576^{\delta}$, concurring with previous findings for a fixed size population, uniformly distributed, in increasing dimensions. Comparing this value to the mean number of ε revisions per search, φ, we found that the successive ratios for δ^{φ} for the dimensions from 2-19 was ~1.567, even closer to limit proposed in [7].

This finding demonstrated a strong correlation between the number of ε revisions and the size of the search space, further indicating that early ε revision times are of critical importance to reducing search cost. We are planning additional examination of this correlation between number of ε revisions and search space size.

We have also extended our technique to approximate NN search and k-NN search as well. Using information about initial distributions of the data set, we can determine an initial value for ε based on the probability that at least one element will lie within the hyper-sphere around α

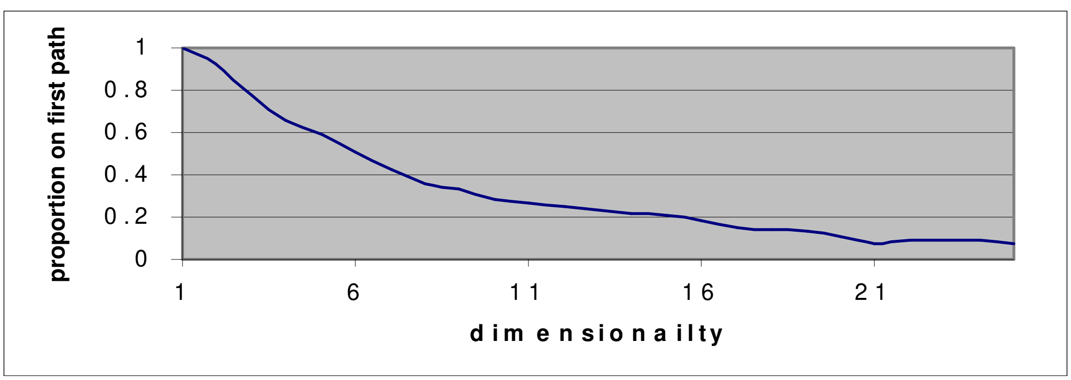

figure 5. True nearest neighbor identified on first path

bounded by ε. The method for choosing an appropriate initial ε is outlined in [1], and has proven successful in their fixed technique. While the results of those experiments are beyond the scope of this paper, we present two critical findings from our additional work on the subject [8].

First, the ability to find the actual nearest neighbor to α on the *first path* through the k-d tree is startlingly good in low dimensions. As shown in figure 5, there is 50% or better chance of hitting the exact nearest neighbor on the first path searched through six dimensions. Through fifteen dimensions, the first path averages finding no worse than the “third” nearest neighbor. A very good approximation of the true nearest neighbor to α can be found with the search of a single tree path.

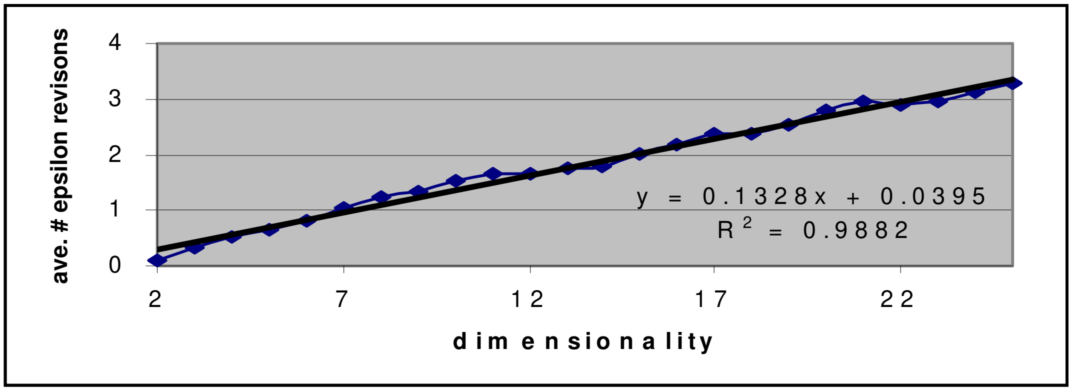

figure 6. Average number of epsilon revisions

Further, the number of ε revisions grows at a constant rate (figure 6). Again, this is partly due to the sparse population of high dimensional spaces, but has been used for good k-NN search times [8]. Point clustering is much lower in high dimensions, thus our search path encounters better candidates more often, but these better candidates are not as often *significantly* better candidates. In effect, there is a slower rate of search sphere contraction in higher dimensions. This facilitates k-NN search without much additional overhead.

It is a simple task to collect the **k** nearest neighbors from the first bin(s) examined, using the distance of the farthest neighbor from α as the initial value for ε. Whenever a case nearer to α than the **k**th case is found, it is inserted into the candidate list; ε is revised down to the distance of the new **k**th element in the candidate list from α. This technique and a similar approximate k-NN method are quite successful for classification [8]. Our method for finding the **k** nearest neighbors is an adaptation of the priority queue method described in [9].

Conclusions

It is unlikely that the growth rates for k-d tree searches will fall below the $O(1.56594^{\delta})$ barrier without a significant (and previously unimagined) break-through. In the meantime, we have proposed a method for minimizing the hidden constant in the search time.

By coupling DFBB search with an intelligent search ordering heuristic based on our tracking nodes, we have shown times that are nearly 80% better than naïve or stochastic searches, until effects of spatial density distort the results in higher dimensions.

Approximate NN search and k-NN search of k-d trees can benefit from our search method as well. A similar approach to our own was taken in [9], but we have advanced those techniques with an ordered search property previously unused.

References

[1] Nene, S. A., and Nayar, S. K. 1997. A simple algorithm for nearest neighbor search in high dimensions. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 19(9): 989-1003.

[2] Friedman, J. H., Bentley, J. H., and Finkel, R. A. 1977. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3): 209-226.

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

[4] Vempati, N. R., Kumar, V., and Korf, R. E. 1991. Depth-first vs best-first search. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, pages 434-440.

[5] White, D. A., and Jain, R. 1996. Algorithms and Strategies for Similarity Retrieval. Technical Report VCL-96-101, Visual Computing Laboratory, University of California, San Diego.

[6] Bentley, J. L. 1975. Multidimensional binary search trees used for associative searching. In *Communications of the ACM*, 18(9): 509-517.

[7] Arya, S., Mount, D. M., and Narayan, O. 1995. Accounting for Boundary Effects in Nearest Neighbor Searching. In *11th Annual Symposium on Computational Geometry*, pages 336-344.
Outline:

[8] Sample, N., and Haines, M. 1997. Rapid classification techniques for high dimensional search spaces. Technical Report. HHCL-97-100, High-speed Heterogeneous Computing Laboratory, University of Wyoming.

[9] Wess, S., Althoff, K. D., and Derwand, G. 1994. Using k-d trees to improve the retrieval step in case-based reasoning. In S. Wess, K. D. Althoff, and M. M. Richter (Eds.), *Topics in Case-Based Reasoning*. Berlin: Springer Verlag.

[10] Sproull, R. F. 1991. Refinements to Nearest-Neighbor Searching in k-Dimensional Trees. *Algorithmica* 6(4): 579-589.