Algorithms for Fast Vector Quantization[*]

Sunil Arya[†]
Department of Computer Science
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong

David M. Mount[‡]
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland, College Park, Maryland, USA

Abstract

Nearest neighbor searching is an important geometric subproblem in vector quantization. Existing studies have shown that the difficulty of solving this problem efficiently grows rapidly with dimension. Indeed, existing approaches on unstructured codebooks in dimension 16 are little better than brute-force search. We show that if one is willing to relax the requirement of finding the true nearest neighbor then dramatic improvements in running time are possible, with negligible degradation in the quality of the result. We present an empirical study of three nearest neighbor algorithms on a number of data distributions, and in dimensions varying from 8 to 16. The first algorithm is the standard *k-d* tree algorithm which has been enhanced to use incremental distance calculation, the second is a further improvement that orders search by the proximity of the *k-d* cell to the query point, and the third is based on a simple greedy search in a structure called a neighborhood graph.

Key words: Nearest neighbor searching, closest-point queries, data compression, vector quantization, *k-d* trees.

1 Introduction

The nearest neighbor problem is to find the point closest to a query point among a set of n points in d-dimensional space. We assume that the distances are measured in the Euclidean metric. Finding the nearest neighbor is a problem of significant importance in many applications. One important application is vector quantization, a technique used in the compression of speech and images [15]. Samples taken from a signal are blocked

[*]A preliminary version of this paper appeared in the *Proc. Data Compression Conference*, J. A. Storer and M. Cohn, eds., Snowbird, Utah, 1993, IEEE Computer Society Press, 381–390.

[†]Email: `arya@cs.ust.edu`. The work of this author was partially supported by HK RGC grant HKUST 736/96E. Part of this research was conducted while the author was at the Dept. of Computer Science, University of Maryland, College Park, MD, USA.

[‡]Email: `mount@cs.umd.edu`. The support of the National Science Foundation under grant CCR–9310705 is gratefully acknowledged.

into vectors of length d. Based on a training set of vectors, a set of codevectors is first precomputed. The technique then encodes each new vector by the index of its nearest neighbor among the codevectors.

The rate r of a vector quantizer is the number of bits used to encode a sample and it is related to n, the number of codevectors, by $n = 2^{rd}$. For fixed rate, the performance of vector quantization improves as dimension increases but, unfortunately, the number of codevectors grows exponentially with dimension. There have been two major approaches to deal with this increase in complexity. The first approach is to impose structure on the codebook, so that the nearest neighbor or an approximation to it can be found rapidly [6, 11, 17, 20, 22]. Some deterioration in performance occurs because the imposition of structure results in a non-optimal codebook. The second approach is to preprocess the unstructured codebook so that the complexity of nearest neighbor searching is reduced [8, 9, 14, 21, 28]. Although methods based on preprocessing an unstructured codebook appear to work well in low dimensions, their complexity increases so rapidly with dimension that their running time is little better than brute-force linear search in moderately large dimensions [23].

In this paper we show that if one is willing to relax the requirement of finding the true nearest neighbor, it is possible to achieve significant improvements in running time and at only a very small loss in the performance of the vector quantizer. Recently this approach has been studied by us and some others from a theoretical perspective [1, 2, 10, 7]. In this work, however, we are more concerned with practical aspects of the search algorithms.

We present three algorithms for nearest neighbor searching:

(1) the standard *k-d* tree search algorithm [14, 23], which has been enhanced to use incremental distance calculation,

(2) a further improvement, which we call *priority k-d tree search*, which visits the cells of the search tree in increasing order of distance from the query point, and

(3) a neighborhood graph search algorithm [1] in which a directed graph is constructed for the point set and edges join neighboring points.

The following elements characterize our *k-d* tree implementation in comparison with Sproull [23] and Friedman, et al. [14]:

(1) As in Sproull but unlike Friedman, et al., we do not store the bounds array at each node of the tree.

(2) As in Friedman, et al. but unlike Sproull, we compute exact distances to the cells of the *k-d* tree.

(3) We introduce an incremental distance calculation technique. This allows us to maintain exact distances to cells as we search the *k-d* tree in constant time independent of dimension.

We performed numerous experiments on these algorithms on point sets from various distributions, and in dimensions ranging from 8 to 16. We studied the running times of these algorithms measured in various ways (number of points visited, number of floating point operations). We also measured the performance of these algorithms in various ways (average and maximum relative error, signal to noise ratio, and probability of failing to find the true nearest neighbor). Our studies show that, for many distributions in high dimensions, the latter two algorithms provide a dramatic reduction in running time over standard approaches with very little loss in performance.

2 Standard k-d Tree Search with Incremental Distance Calculation

Bentley introduced the *k-d* tree as a generalization of the binary search tree in higher dimensions [4]. For our purposes, a *rectangle* in real d dimensional space, R^d, is the product of d closed intervals on the coordinate axes. Each internal node of the *k-d* tree is associated with a rectangle and a hyperplane orthogonal to one of the coordinate axis, which splits the rectangle into two parts. These two parts are then associated with the two child nodes in the tree. The process of partitioning space continues until the number of data points in the rectangle falls below some given threshold. The rectangles associated with the leaf nodes of the tree are called *buckets*, and they define a subdivision of space into rectangles. Data points are only stored in the leaf nodes of the tree, not in the internal nodes.

Friedman, Bentley and Finkel [14] gave an algorithm to find the nearest neighbor using optimized *k-d* trees that takes $O(\log n)$ time in the expected case, under certain assumptions on the distribution of data and query points. The internal nodes of the *optimized k-d* tree split the set of data points lying in the corresponding rectangle into two subsets of equal size, along the dimension in which the data points have maximum spread. Intuitively, the query algorithm works by first descending the tree to find the data points lying in the bucket that contains the query point. Then it recursively examines surrounding buckets if they intersect the ball B centered at the query point and having radius equal to the distance between the query point and the closest data point visited so far.

To determine whether a given subtree contains a bucket that should be visited, each internal node can store a *bounds array*, which contains the lower and upper limits of the corresponding rectangle. Sproull [23] showed that there was no need to store the bounds array explicitly, and gave an alternative version to save space and speed up the search. Sproull's version examines buckets if they intersect the smallest hypercube enclosing ball B. Although this simplifies comparisons, this results in a significantly higher number of nodes being visited in higher dimensions. This is a consequence of the large difference in the volume of a ball and the volume of its minimum enclosing hypercube in these dimensions.

We refine Sproull's implementation by examining buckets only if they actually intersect ball B. We call this the *distance refinement*. This provides a very significant savings in higher dimensions. For example, given 65,536 points in 16 dimensional space from an uncorrelated Gaussian source, the numbers of points visited with and without this refinement are 14,500 and 50,000, respectively. These averages were computed over 25,000 query points also from the same source.

The distance refinement is easy to carry out when the partitioning hyperplanes are orthogonal, by exploiting the very simple relation that exists between the distance of the query point from the rectangle corresponding to a node, and the distance of the query point from the rectangles corresponding to the two children of the node. However, we do not know how to apply this refinement in the case of partitioning hyperplanes with general orientations. The case of partitioning hyperplanes with general orientation has been studied by Sproull [23] who found that in high dimensions (e.g. 16) the added generality led to speed-ups of over two-fold for correlated data sets, compared to the case of orthogonal partitioning hyperplanes. It would be interesting to develop a scheme that allows flexibility in the choice of the orientation of the partioning hyperplane while still retaining the ease of carrying out the distance refinement.

For our experiments we used optimized *k-d* trees with one data point per bucket and we measured distances in the Euclidean norm. We shall assume this to be the case in the rest of the paper. The code we present, however, can be easily modified to work with more than one point per bucket and with other Minkowski metrics. We borrow Bentley's terminology on *k-d* trees and modify his code for nearest neighbor searching (that includes Sproull's suggestions) [5].

Using the C++ programming language [24], a node in a *k-d* tree is represented by the following structure.

```
struct kd_node {
    int     leaf_node;              // 1 if leaf, 0 if internal node
    int     cut_dim;                // cutting dimension
    float   cut_val;                // cutting value
    kd_node *lo_child, *hi_child;   // low and high children
    int     pt;                     // data point index
}
```

The Boolean variable `leaf_node` is 1 for leaf nodes and 0 for internal nodes. For internal nodes, `cut_dim` gives the dimension, 0 through $d-1$, being partitioned, and `cut_val` gives the location of the partitioning hyperplane along this dimension. Also, for internal nodes, `lo_child` and `hi_child` are pointers to the children of this node. A data point is placed in `lo_child` if its coordinate along the cutting dimension is lower than the cutting value, and in `hi_child` if it is greater. Points lying exactly on the cutting hyperplane can be placed in either child. For leaf nodes, `pt` indicates the point associated with this bucket, and is given by an index into a global array of data points `points[n]`.

The algorithm works as follows. At each leaf node visited we compute the squared distance between the query point and the data point in the bucket and update the nearest neighbor if this is the closest point seen so far. At each internal node visited we first recursively search the subtree whose corresponding rectangle is closer to the query point. Later, we search the farther subtree if the distance between the query point and the closest point visited so far exceeds the distance between the query point and the corresponding rectangle. See [14] for more details.

We can avoid computing square roots by working with squared distances instead. We facilitate the computation of the squared distance between the query point and the rectangle through a method called *incremental distance calculation*. Given a query point q and the rectangle R_u associated with a node u, for $0 \leq i \leq d-1$, define the ith *offset*, denoted $\mathit{off}_u[i]$, to be the distance from q to R_u along this axis. More precisely, letting I denote the orthogonal projection of R_u onto the ith coordinate axis and letting $q[i]$ denote the ith coordinate of q, $\mathit{off}_u[i]$ is defined to be the difference between $q[i]$ and its nearest point on I. The difference is negative if the ith coordinate of q is less than the lowest point of I, zero if it overlaps I, and positive if it is greater than the highest point in I. Clearly, the squared distance, denoted rd_u, between q and the rectangle is just the sums of the squares of these offsets.

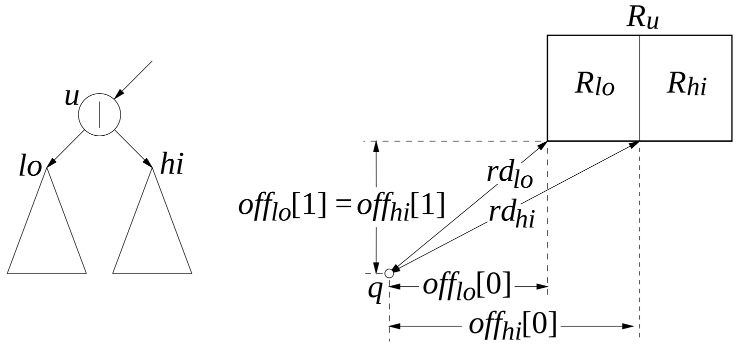


Figure 1: Incremental Distance Calculation Technique

When the algorithm descends the k-d tree from a node u to its children lo and hi, it is possible to compute the distance from q to the two children rectangles R_{lo} and R_{hi} in constant time (independent of dimension). Let cd denote the cutting dimension and let cv denote the cutting value. For concreteness, assume that R_{lo} is closer to q than R_{hi} is. The other case is handled symmetrically. (See Figure 1.) Because R_{lo} is the closer rectangle to q, its distance and offsets from q are the same as the enclosing rectangle R_u. For R_{hi}, observe that for each dimension $i \neq cd$, $off_{hi}[i] = off_u[i]$, since these coordinates are not affected by the current cut. Since this is the further of the two children, it follows that along the cutting dimension, the offset between q and R_{hi} is the distance from q to the cutting value, that is, $off_{hi}[cd] = q[cd] - cv$. To compute rd_{hi}, we simply subtract the square of the existing offset $off_u[cd]$ and add the square of the new offset, yielding

$$rd_{hi} = rd_u - off_u[cd]^2 + (q[cd] - cv)^2.$$

We can now present the entire function `kd_standard`. It is given a query point and the root of a k-d tree, and returns the squared distance to the nearest neighbor of a given query point. (It is a trivial matter to modify the code to return the actual nearest neighbor point.) After some initialization it invokes the recursive procedure `rkd` which performs the search. For each node u visited, the parameter `rd` contains rd_u. Note that this parameter is passed by value, so changes made to `rd` do not alter the value of the variable in the calling procedure. The global array `off[d]` contains the contents of off_u. The elements of this array are modified individually as we traverse the tree.

The constant `d` is the dimension of the space, `n` is the number of data points, and `HUGE` is a number larger than any squared distance, which is used to initialize the nearest neighbor distance before beginning the search. (In many applications, nearest neighbor queries demonstrate a large amount of coherence, and so a more practical choice is the distance between the query point and the previous point returned from the nearest neighbor algorithm.) The function `dist2` returns the squared distance between two points.

```
typedef float Point[d];                         // point data type
```

```
Point points[n];                           // data point storage
Point q;                                   // query point
float off[d];                              // array of offsets
float nn_dist;                             // best squared distance so far

float kd_standard(
    Point qq,                              // query point
    kd_node* root)                         // root of kd tree
{
    q = qq;                                // save query point
    nn_dist = HUGE;                        // initial distance
    for (int i = 0; i < d; i++)            // initialize offsets
        off[i] = 0.0;
    rkd(root, 0.0);                        // search the tree
    return nn_dist;
}

void rkd(                                  // recursive search procedure
    kd_node* u,                            // current node
    float rd)                              // squared dist to this rect
{
    if (u->leaf_node) {                    // at a leaf bucket
        nn_dist = min(nn_dist,             // use this point if closer
                      dist2(q, points[u->pt]));
    }
    else {                                 // internal node
        int   cd = u->cut_dim;             // cutting dimension
        float old_off = off[cd];           // save old offset
        float new_off = q[cd] - u->cut_val; // offset to further child
        if (new_off < 0) {                 // left of cutting plane
            rkd(u->lo_child, rd);          // search closer subtree first
            rd += - old_off*old_off        // distance to further child
                  + new_off*new_off;
            if (rd < nn_dist) {            // close enough to consider?
                off[cd] = new_off;         // update offset
                rkd(u->hi_child, rd);      // search further subtree
                off[cd] = old_off;         // restore offset
            }
        }
        else {                             // q is above cutting plane
            ...analogous with lo_child and hi_child interchanged...
        }
    }
}
```

3 Priority *k*-*d* Tree Search

Our experience suggests that the standard *k*-*d* tree algorithm usually comes across the nearest neighbor well before the search terminates. One may view the extra search as the price to pay to guarantee that the nearest neighbor has been found. If we are willing to sacrifice this guarantee, then the complexity can be reduced by interrupting the search before it terminates (say, after a fixed number of points have been visited). In this case, it is desirable to order the search so that buckets more likely to contain the nearest neighbor are visited early on. This suggests a variant of the standard k-d tree algorithm that visits the buckets of the k-d tree in increasing order of distance from the query point.

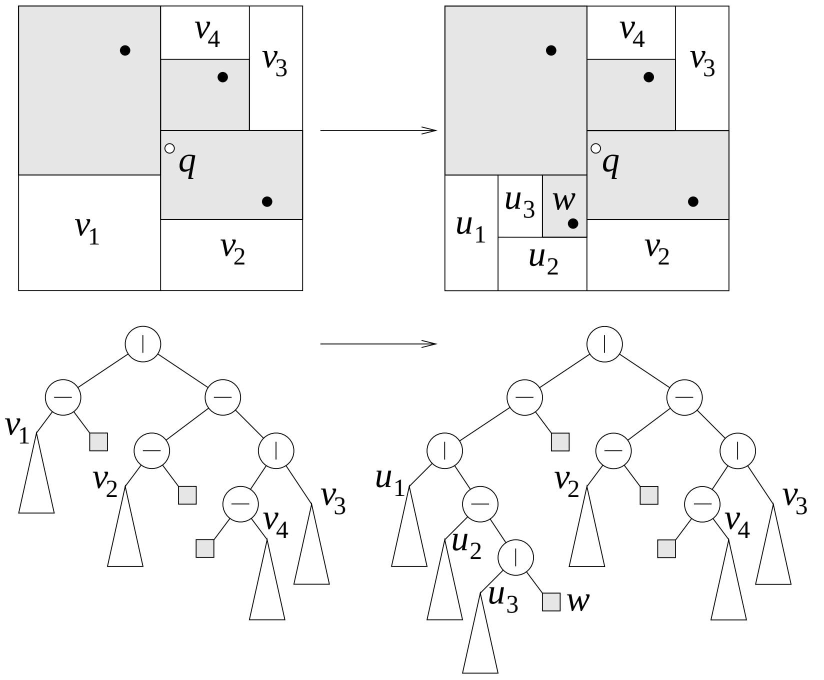


Figure 2: Priority k-d tree algorithm

The algorithm maintains a priority queue of subtrees, where the priority of a subtree is inversely related to the distance between the query point and the rectangle corresponding to the subtree. Initially, we insert the root of the k-d tree into the priority queue. Then we repeatedly carry out the following procedure. First, we extract the node v from the queue with the highest priority, that is, closest to the query point. (See Figure 2.) Then we descend the subtree rooted at this node in search of the leaf bucket that is closest to the query point. As we descend the subtree, for each node u that we visit we insert u's sibling into the priority queue. These nodes will be considered at a later time in the search. (For example, in Figure 2, nodes u_1, u_2, and u_3 are inserted into the queue). The algorithm terminates when the priority queue is empty (meaning that the entire tree has been searched), or sooner, if the distance from the query point to the rectangle corresponding to the highest priority subtree is greater than the distance to the closest data point.

To compute the time required to visit each new bucket, note first that for the binomial heap implementation of the priority queue, it takes amortized $O(1)$ time to insert a subtree and $O(\log n)$ time to extract the subtree with highest priority [18, 26]. The depth of the balanced k-d tree is $O(\log n)$; this bounds the time taken to descend the subtree as also the number of subtrees inserted into the queue at each iteration. Thus the time needed to visit each new bucket is $O(\log n)$.

For our experiments, we implemented the priority queue as a binary heap [12, 27]. Theoretically, insertions may take $O(\log n)$ time with binary heaps, but we observed that they took only $O(1)$ time on average. This suggests that the greater complexity of binomial heaps is not warranted for our application.

The following lemma establishes the correctness of the algorithm.

LEMMA 3.1 *Priority search visits the buckets in the order of increasing distance from the query point.*

PROOF:

For the sake of simplicity, we assume that no two buckets are equidistant from the query point. Let $v_1, v_2, \ldots, v_k$ be the subtrees in the priority queue, just before we extract the subtree with highest priority from the queue. We claim that the following invariant holds (here $L(t)$ denotes the set of leaves in subtree t):

$$\begin{aligned} L(v_i) \cap L(v_j) &= \emptyset \quad \text{for } 1 \leq i < j \leq k \quad \text{and} \\ \textstyle\bigcup_{1 \leq i \leq k} L(v_i) &= \text{Set of leaves not yet visited} \end{aligned}$$

Clearly, the invariant holds at the beginning of the first iteration, when the root is the only subtree in the queue and no leaf has yet been visited. To establish the induction step assume that the invariant holds before some iteration. Among leaves not yet visited, let b be the one that is closest to q. It follows from the invariant that there must be exactly one subtree v_j in the priority queue such that $b \in L(v_j)$. It is easy to see that v_j has the highest priority and is removed from the queue. As we descend the subtree v_j to visit b, we insert m subtrees ($m \geq 0$), $u_1, u_2, \ldots, u_m$, which satisfy

$$\begin{aligned} L(u_i) \cap L(u_j) &= \emptyset \quad \text{for } 1 \leq i < j \leq m \quad \text{and} \\ \textstyle\bigcup_{1 \leq i \leq m} L(u_i) &= L(v_j) - \{b\} \end{aligned}$$

From this it is a straightforward exercise to show that the invariant continues to hold after this iteration. □

The search algorithm is outlined below. Here too we employ the incremental distance calculation technique which we discussed for the standard *k-d* tree algorithm. In order to make this technique work, we stored two additional pieces of information in each node of the tree, the lower and upper limits of the node's rectangle along the node's cutting dimension (denoted `low_val` and `high_val` respectively). We let `Q` denote the priority queue. The procedure `Q.Insert` inserts a node into the queue at the specified distance, and `Q.Extr_Min` extracts the node with the minimum distance from the queue, and returns the node and its distance from the query point through its arguments.

```
float kd_priority(
    Point q,                                  // query point
    kd_node* root)                            // root of kd tree
{
    Priority_Queue Q;
    float nn_dist = HUGE;                     // initial distance
    kd_node* u;
    float rd;                                 // distance to rectangle
    Q.Insert(root, 0.0);                      // start with root of tree
    while (Q.NotEmpty()) {                       // repeat until queue is empty
        Q.Extr_Min(u, rd);                       // closest node to query point
        if (rd >= nn_dist)                       // further from nearest so far
            break;
        while (!u->leaf_node) {                  // descend until leaf found
            int   cd = u->cut_dim;               // cutting dimension
            float old_off, new_rd;
            float new_off = q[cd] - u->cut_val; // offset to further child
            if (new_off < 0) {                   // q is below cutting plane
                old_off = q[cd] - u->low_val;   // compute offset
                if (old_off > 0)                 // overlaps interval
                    old_off = 0;
                new_rd = rd - old_off*old_off  // distance to further child
                              + new_off*new_off;
                Q.Insert(u->hi_child, new_rd);  // enqueue hi_child for later
                u = u->lo_child;                 // visit lo_child next
            }
            else {                               // q is above cutting plane
                ...analogous with lo_child and hi_child interchanged...
            }
        }
        nn_dist = min(nn_dist,                   // leaf - use point if closer
                    dist2(points[u->pt], q));
    }
    return nn_dist;
}
```

4 Neighborhood Graphs

We give here a brief overview of an approach to nearest neighbor searching based on the notion of neighborhood graphs, which was introduced in [1]. A *neighborhood graph* is a connected graph (directed or undirected) whose vertices are the set of data points, such that two points are adjacent to one another if they satisfy some local criterion. For example, the Delaunay triangulation is an undirected neighborhood graph in which two points are adjacent if there is a sphere passing through the two points that contains no other point in its interior.

Given a neighborhood graph we can search for the nearest neighbor of a query point using a greedy strategy. We start the search with the data point p from the bucket of the k-d tree containing the query point. We repeatedly carry out the following steps. We *expand* the point p, by which we mean that we compute the distance to the query point for all those neighbors of point p that have not yet been expanded. Among such neighbors, we expand the point that is closest to the query point. We continue to expand points in this manner until we arrive at a point all of whose neighbors have already been expanded (the search is said to have reached an *impasse*), or the number of points visited by the algorithm exceeds some prespecified cut-off value. Then we end the search and output the closest data point visited.

The neighborhood graph we use for nearest neighbor searching is quite similar to the relative neighborhood graph (RNG) [19, 25]. In the RNG, two points p and r are adjacent if there is no point that is simultaneously closer to both points than they are to one another. The modified graph we build is equivalent to a graph presented by Jaromczyk and Kowaluk

[16] which was used as an intermediate result in their construction of the RNG. It is based on the following pruning rule. For each point p in the data set, we consider the remaining points in increasing order of distance from p. We remove the closest point x from this sequence, create a directed edge from p to x, and remove from further consideration all points s such that $dist(p, s) > dist(x, s)$. This process is repeated until all points are pruned. Figure 3 shows an example of this process applied to a point p in three stages; at each stage, a new edge is directed from p to one of its neighbors. The three neighbors of point p in the neighborhood graph are x, y and z. This variant, called the RNG*, can be computed in $O(n^2)$ time, where n is the number of points. Details on the degree of the RNG* and intuitive reasons for its appeal in nearest neighbor searching can be found in [1].

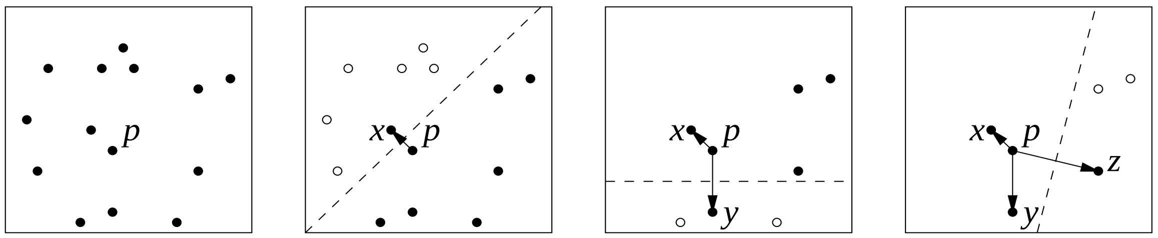


Figure 3: Construction of the *RNG** neighborhood graph.

Although the worse case behavior of the greedy algorithm can be quite bad, our experimental studies indicate that in high dimensions the search quickly zeroes in to find the nearest neighbor and only rarely reaches an *impasse* before finding the nearest neighbor.

5 Empirical Analysis

We ran a series of experiments to compare the performance of these three algorithms: *k-d* standard, *k-d* priority, and *RNG**-search. Both the *k-d* tree algorithms were enhanced to use incremental distance calculation. Before running the experiments, we optimized many aspects of the code. We mention two of these optimizations. First, the well-known *partial distance* optimization was implemented for all three algorithms: as we compute the squared distance between the query point and the data point by summing the contribution from each dimension, we exit the loop when the accumulated sum of the squares becomes too large [3, 23]. This optimization diminished the total number of floating point operations at only a small increase in the number of comparisons. Second, for the RNG*-search, we saved the results of the partial distance computations so that they could be used again if the same point was encountered on expanding several different points.

We studied how the performance of the vector quantizer changes as a function of the complexity of the algorithms. We focused on a rate of one bit per sample in dimensions ranging from 8 to 16. For each of the three algorithms the search is interrupted if the number of points visited by the algorithm reaches a certain threshold (*cut-off value*), and the closest point visited until then is taken as the output of the algorithm. By varying this cut-off value the complexity of each algorithm can be changed. We used 25,000 query points for each experiment and recorded the following for each query point at each cut-off (here

d_e is the distance between the query point and the data point output by the algorithm and d_n is the distance between the query point and its nearest neighbor):

- The number of data points visited by the algorithm. A data point is said to be *visited* if the algorithm accesses its coordinates. Each data point is counted at most once in this total. If the algorithm terminates before the cut-off is reached, then this quantity is the same as the number of points visited until termination, otherwise it is the cut-off value.
- The number of floating point operations per sample. In this we included all floating point additions, subtractions, multiplications and comparisons (except comparisons with zero) performed by the algorithm, not just those involved in computing distances between points. The total number of floating point operations is divided by the dimension to get the per sample average.
- Whether the true nearest neighbor has already been found.
- The error-factor which is defined as $(d_e - d_n)/d_n$.
- The distortion per sample which is defined as $(d_e)^2/d$, where d is the dimension.

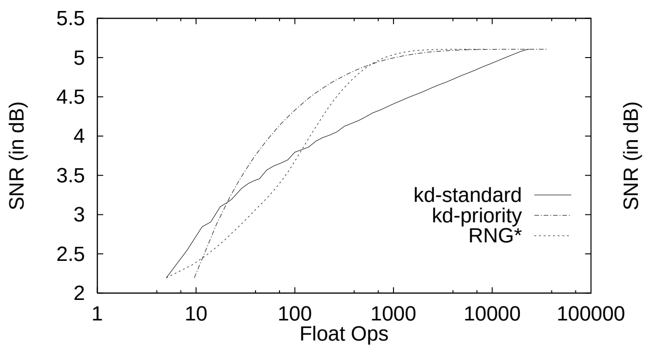
Three measures of performance were computed at each cut-off: (1) the signal-to-noise ratio (SNR), (2) the average error-factor, and (3) the miss probability (probability of failing to find the nearest neighbor). The SNR is defined as $10\log_{10}(V/D)$, where V is the variance of the samples and D is the average distortion per sample. All these averages are taken over the entire set of query points. Of these measures of performance, SNR is the most significant one for vector quantization. The other two measures are here principally to aid in a better empirical understanding of the algorithms.

Two measures of complexity were computed at each cut-off: (1) the average number of points visited and (2) the average number of floating point operations per sample. The number of points visited is a useful quantity to study, but since the algorithms have different overheads, the number of floating point operations is more directly related to the complexity of the algorithms. Our studies indicate that the number of floating point operations is a reasonable measure of the search time, and can be used to compare the algorithms.

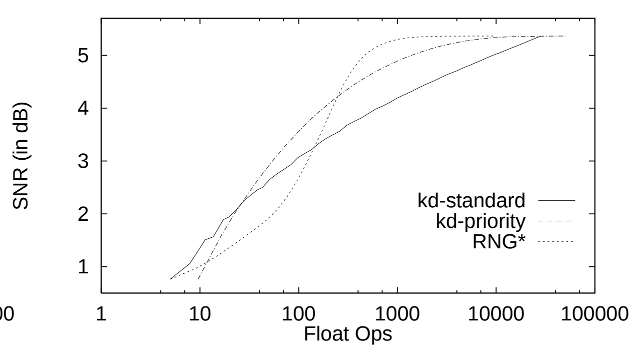
We conducted experiments using the Gaussian and the Laplacian sources. Both uncorrelated and correlated sources were used. For the correlated sources, we used 0.9 as the correlation coefficient. All the sources had zero mean and unit variance.

In dimension 16 we used codebooks consisting of 65,536 codevectors generated by the *k-d* tree based Equitz algorithm [13]. We sped up Equitz algorithm in several ways and, for uncorrelated sources, instead of building balanced *k-d* trees as is customary, we partitioned the rectangles corresponding to the internal nodes such that a random number of points were contained in each part. This led to codebooks of better quality. The size of the training set used was 32 times the size of the codebook. For our experiments the training set and test set were different.

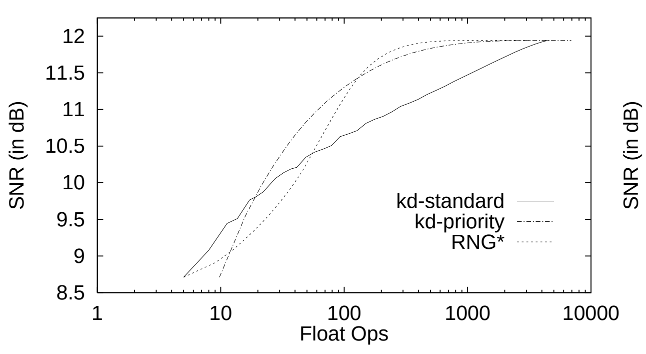
Figure 4 (a)–(d) show the variation of the SNR with the average number of floating point operations per sample, for the uncorrelated Gaussian, uncorrelated Laplacian, correlated Gaussian, and correlated Laplacian sources, respectively. For a more careful study of the



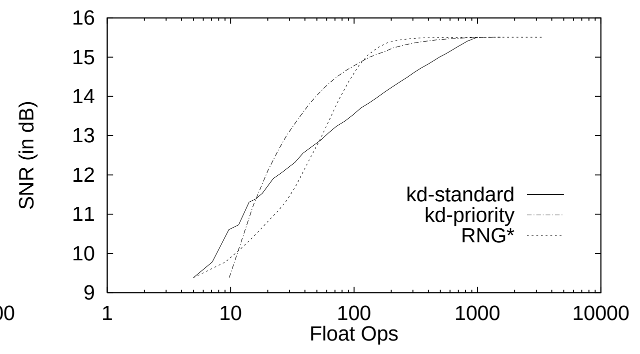
(a)



(b)



(c)



(d)

Figure 4: SNR vs. Average floating point operations per sample: (a) Uncorrelated Gaussian Source (b) Uncorrelated Laplacian Source (c) Correlated Gaussian Source (d) Correlated Laplacian Source.

DISTRIBUTION	SNR-MAX	KD-STANDARD	KD-PRIORITY	RNG*
Uncorrelated Gaussian	5.11	12000	1100	850
Uncorrelated Laplacian	5.36	18500	4500	850
Correlated Gaussian	11.94	2500	550	300
Correlated Laplacian	15.51	650	400	200

Table 1: Floating point operations per sample to achieve SNR within 0.1 dB of SNR-MAX.

DISTRIBUTION	SNR-MAX	KD-STANDARD	KD-PRIORITY	RNG*
Uncorrelated Gaussian	5.11	19000	5000	2000
Uncorrelated Laplacian	5.36	24000	15000	2000
Correlated Gaussian	11.94	3700	1700	600
Correlated Laplacian	15.51	800	950	450

Table 2: Floating point operations per sample to achieve SNR within 0.01 dB of SNR-MAX.

high performance region (say, less than 0.1 dB deterioration from the performance obtained by full exhaustive search, SNR-MAX), we make two tables. Table 1 compares the three algorithms in terms of the average number of floating point operations per sample needed to achieve SNR within 0.1 dB of SNR-MAX. Table 2 shows the same needed to achieve SNR within 0.01 dB of SNR-MAX.

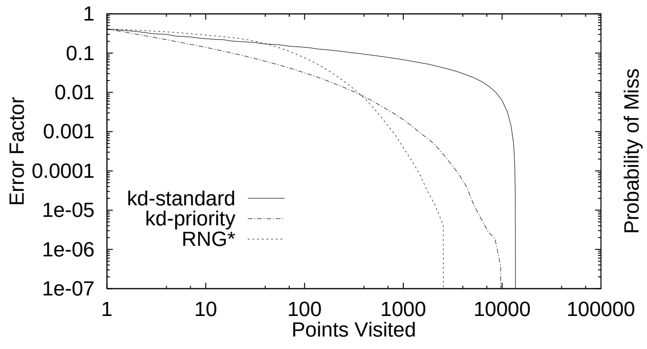
We summarize the key observations for dimension 16 in the high performance region:

- RNG*-search is the fastest of the three algorithms followed by the priority *k-d* tree algorithm. Both these algorithms often achieve very significant speed-ups, sometimes by a factor of over 10, compared to the standard *k-d* tree algorithm.

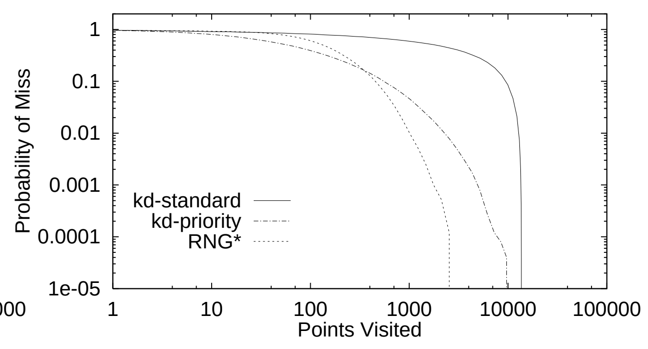
- The complexity of RNG*-search ranges from being just a little better than the priority *k-d* tree algorithm to being much better, sometimes achieving speed-ups by a factor of over 5.

- All three algorithms achieve significant speed-ups over full exhaustive search, with negligible loss in performance (less than 0.01 dB). RNG*-search achieves massive speed-ups by a factor of over 100 compared to full exhaustive search. Even the standard *k-d* tree algorithm achieves speed-ups by a factor of over 8 compared to full exhaustive search.

- The two *k-d* tree algorithms require similar storage, while the RNG*-search may require about twice as much (depending on the source and the implementation). Regarding dimension as fixed and ignoring the space needed for the data points, the storage requirements of all three algorithms is $O(n)$. The constant of proportionality for the *k-d* tree algorithm is largely independent of dimension, while for the RNG*-search it shows a moderately exponential growth. For the uniform distribution, empirical studies show that the degree of RNG* grows roughly as $2.90(1.24^k)$ in the asymptotic case, and as $1.46(1.20^k)$ when the number of points grow as 2^k [1]. The cost of building the RNG* is $O(n^2)$ while that of building the k-d tree is $O(n \log n)$. This is an enormous difference because the number of points is so large.



(a)



(b)

Figure 5: Uncorrelated Gaussian Source: (a) Average error-factor vs. Average points visited: (b) Probability of missing the nearest neighbor vs. Average points visited.

For the uncorrelated Gaussian source, Figure 5 (a) and (b) show how the average error-factor and the probability of failing to find the nearest neighbor vary with the average number of points visited. These graphs show that both these quantities fall much more rapidly for the RNG*-search than for the priority k-d tree algorithm. This suggests that the RNG*-search would enjoy very significant advantage over the priority k-d tree algorithm in any application where these quantities are critical.

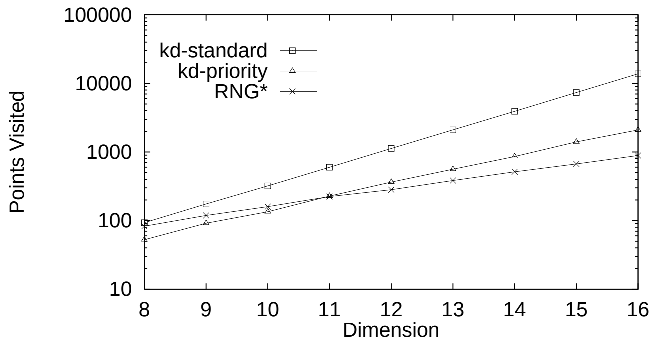


Figure 6: Average number of points visited to achieve average error-factor 0.001: Uncorrelated Gaussian Source

We also conducted experiments for the uncorrelated Gaussian source in several other dimensions ranging from 8 to 16, using random codebooks. In each case we used a rate of 1 bit per sample. In all these dimensions we found that the priority k-d tree algorithm and the RNG*-search are much faster than the standard *k-d* tree algorithm. In dimensions below 11 we found the priority *k-d* tree algorithm to be faster than the RNG*-search, while in dimensions above 11 the RNG*-search is faster in the high performance region. This can be seen in Figure 6 which shows, in various dimensions, the average number of points visited by the three algorithms to achieve an average error-factor of 0.001.

6 Conclusions

We have shown how to improve the *k-d* tree implementation using incremental distance calculation. This technique employed in conjunction with both standard and priority search greatly reduces their query time and storage requirements.

We have presented and compared three algorithms for nearest neighbor searching in high dimensions, within the framework of vector quantization. Two of the algorithms give drastic reductions in complexity with negligible deterioration in performance. There are several interesting open questions. One is to develop a theoretical understanding of the relationship between the performance and the complexity of the algorithms and establish their efficiency. Another question is whether the cost of building the RNG* can be reduced, or whether we can devise some other neighborhood graph that can be built more quickly, while still giving the same reduction in search complexity.

7 Acknowledgements

We would like to thank Nam Phamdo for many helpful discussions on the subject of generating codebooks, and Nariman Farvardin for useful suggestions.

References

[1] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, 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] C.-D. Bei and R. M. Gray. An improvement of the minimum distortion encoding algorithm for vector quantization. *IEEE Transactions on Communications*, 33(10):1132–1133, October 1985.

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

[5] J. L. Bentley. K-d trees for semidynamic point sets. In *Proc. 6th Ann. ACM Sympos. Comput. Geom.*, pages 187–197, 1990.

[6] A. Buzo, A. H. Gray, R. M. Gray, and J. D. Markel. Speech coding based upon vector quantization. *IEEE Transactions on Acoust., Speech and Signal Process.*, 28(5):562–574, October 1980.

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

[8] D. Y. Cheng and A. Gersho. A fast codebook search algorithm for nearest-neighbor pattern matching. In *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, volume 1, pages 265–268, April 1986.

[9] K. L. Clarkson. A randomized algorithm for closest-point queries. *SIAM Journal on Computing*, 17(4):830–847, 1988.

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

[11] J. H. Conway and N. J. A. Sloane. Fast quantizing and decoding algorithms for lattice quantizers and codes. *IEEE Transactions on Information Theory*, 28:227–232, March 1982.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw-Hill, New York, NY, 1990.

[13] W. H. Equitz. A new vector quantization clustering algorithm. *IEEE Transactions on Acoust., Speech and Signal Process.*, 37(10):1568–1575, October 1989.

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

[15] A. Gersho and R. M. Gray. *Vector Quantization and Signal Compression.* Kluwer Academic, 1991.

[16] J. W. Jaromczyk and M. Kowaluk. A note on relative neighborhood graphs. In *Proc. 3rd Ann. ACM Sympos. Comput. Geom.*, pages 233–241, 1987.

[17] B. H. Juang and A. H. Gray. Multiple stage vector quantization for speech coding. In *Proc. IEEE Int. Conf. Acoust., Speech, Signal Processing*, volume 1, pages 597–600, April 1982.

[18] D. C. Kozen. *The Design and Analysis of Algorithms.* Texts and Monographs in Computer Science. Springer-Verlag, 1992.

[19] P. M. Lankford. Regionalization: theory and alternative algorithms. *Geographical Analysis*, 1(2):196–212, April 1969.

[20] R. Laroia and N. Farvardin. A structured fixed-rate vector quantizer derived from variable-length encoded scalar quantizers. In *Proc. Twenty-fourth Ann. Conf. Inform. Sciences and Systems*, pages 796–801, March 1990.

[21] V. Ramasubramanian and K. K. Paliwal. Fast k-dimensional tree algorithms for nearest neighbor search with application to vector quantization encoding. *IEEE Transactions on Signal Processing*, 40(3):518–531, March 1992.

[22] M. J. Sabin and R. M. Gray. Product code vector quantizers for waveform and voice coding. *IEEE Transactions on Acoust., Speech and Signal Process.*, 32:474–488, June 1984.

[23] R. L. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6, 1991.

[24] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[25] G. T. Toussaint. The relative neighborhood graph of a finite planar set. *Pattern Recognition*, 12(4):261–268, 1980.

[26] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21:309–314, 1978.

[27] J. W. J. Williams. Algorithm 232 (heapsort). *Communications of the ACM*, 7:347–348, 1964.

[28] A. C. Yao and F. F. Yao. A general approach to d-dimensional geometric queries. In *Proc. 17th Ann. ACM Sympos. Theory Comput.*, pages 163–168, 1985.