DIMACS Series in Discrete Mathematics
and Theoretical Computer Science

Nearest Neighbor Search for Data Compression

Kevin Zatloukal, Mary Holland Johnson, and Richard E. Ladner

ABSTRACT. Vector Quantization is a lossy data compression method whose running time is dominated by a series of nearest neighbor searches. In this paper, we empirically compare six nearest neighbor search algorithms for performing vector quantization. Three of these algorithms were designed specifically for vector quantization. One is the traditional k-d tree algorithm. The other two are new algorithms based on the idea of principal component partitioning.

1. Introduction

Vector Quantization (VQ) is a standard method for lossy data compression [**10, 6, 5**]. The running time of VQ is dominated by a series of nearest neighbor searches of a set a d-dimensional vectors called a *codebook*. Hence, the choice of nearest neighbor search algorithm is of critical importance in producing a fast implementation of VQ.

In this paper, we compare six nearest neighbor search algorithms for performing VQ. Orchard's Method [**11**], the Annulus Method [**7**], and the Double Annulus Method [**9**] are algorithms that were specifically designed for VQ. The k-d tree nearest neighbor search algorithm [**3**] is a traditional, general–purpose algorithm. The PCP tree nearest neighbor search algorithm is based on a partitioning of the codebook by hyperplanes perpendicular to principal components of subsets of the codebook. In prior work the PCP tree was used for progressive transmission of compressed images [**12**] and for approximate nearest neighbor search [**16, 15**]. We use the same PCP tree for nearest neighbor search. The last algorithm is the PCP/k-d tree nearest neighbor search algorithm which combines the PCP tree with the k-d tree. The PCP and PCP/k-d tree nearest neighbor search algorithms are presented for the first time in this paper.

These six algorithms are compared empirically using data from standard images. We will see that no one algorithm is the best for all situations. Each of

1991 *Mathematics Subject Classification.* 68W05.

Key words and phrases. Nearest Neighbor Search, Vector Quantization, Principal Components.

The first author was supported in part by a Mary Gates Research Award at the University of Washington.

The third author was supported in part by NSF grant No. CCR-9732828.

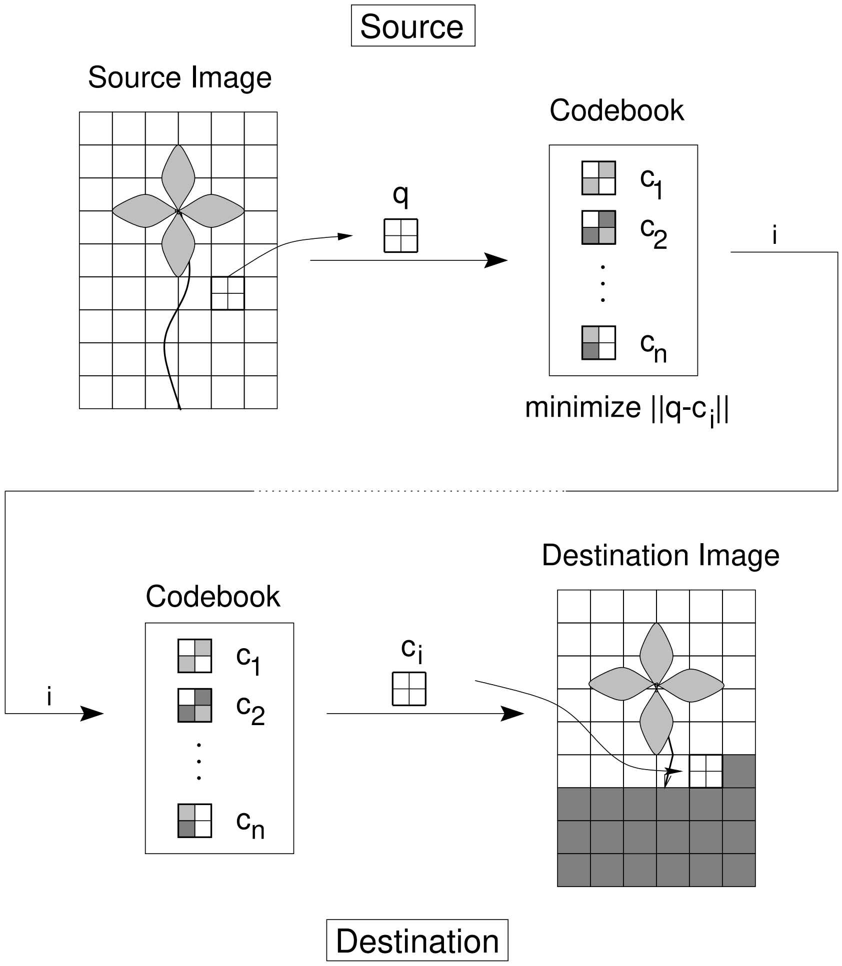

FIGURE 1. VQ algorithm

the algorithms has strengths and weaknesses and their relative running times depend on the vector dimension and size of the codebook. The paper is organized as follows. In section 2 we describe the vector quantization method for image compression. In section 3 we describe all the algorithms we will compare except for the PCP/k-d tree nearest neighbor search algorithm. In section 4 we give the results of experiments comparing the first five algorithms for various vector dimensions and codebook sizes. In section 5 we present the PCP/k-d tree nearest neighbor search algorithm and compare it with the best of the other five. Finally, in section 6 we summarize what we can conclude from the experiments.

2. Vector Quantization

The Vector Quantization algorithm can be used to compress data stored in an array of any dimension [**10, 6, 5**]. However, in this paper, we are only concerned with 2D arrays. In particular, our input is a gray–scale image, i.e., a 2D array of pixel data.

Figure 1 demonstrates how VQ is used to compress an image being sent across a communication channel. The algorithm processes the source image in small 2D blocks (typically between 2×2 and 8×8 pixels). Both the source and destination have a copy of the codebook, a collection of representative blocks. (The blocks in the codebook are called *codewords*.) For each block q of the source image, we find the codeword $\mathsf{c_i}$ that is "most similar" and transmit the index i across the channel. When the index i is received at the destination, the block $\mathsf{c_i}$ is placed into the appropriate position in the destination image. Provided that the codewords chosen are suitably similar to the blocks of the source image, this will produce a destination image very similar to the source image.

One question that remains is how to measure the similarity between two blocks. First, note that any 2D block can be considered to be a mathematical vector by enumerating the pixels in some order. For example, 2×2 blocks are 4D vectors, and 4×4 blocks are 16D vectors. In VQ blocks are not always square, but are typically a power of two on a side. For example, 2×4 blocks are 8D vectors. We can consider a block q from the source image to be a vector $\mathbf{q}$, and we can consider each codeword $\mathsf{c_i}$ to be a vector $\mathbf{c}_i$. Vector quantization defines the "most similar" codeword to be the $\mathbf{c}_i$ that minimizes

$$\|\mathbf{q} - \mathbf{c}_i\| = \sqrt{\sum_{j=1}^{k} ((\mathbf{q})_j - (\mathbf{c}_i)_j)^2},$$

the Euclidean distance from $\mathbf{c}_i$ to $\mathbf{q}$, where k is the dimension of the vectors. Finding this $\mathbf{c}_i$ is a nearest neighbor search.

Vector Quantization achieves compression because the number of bits in an index is usually much smaller than the number of bits in a block. The compression ratio is $b : \log_2 n$ where b is the number of bits in a block and n is the number of codewords. Vector Quantization commonly yields compression ratios better than $10 : 1$ while producing a destination image that is nearly indistinguishable from the source image. For example, suppose we compressed a gray-scale image using 1,024 codewords each 4×4 pixels of 8 bits each. A block would require $b = 4 \cdot 4 \cdot 8 = 128$ bits, whereas an index would require only $\log_2 1024 = 10$ bits. This gives us a compression ratio of $12.8 : 1$. We should note that we are not counting the size of the codebook because we make the reasonable assumption that many images are compressed using the same codebook.

Vector Quantization can also be used to compress data stored in files. The compressed file is simply the stream of indices that would be sent across the channel.

Another question that has not been answered is how to produce the codebook. The goal is to have the codebook contain vectors that are close to vectors that will be coded. Typically, the codebook is created by the Generalized Lloyd Algorithm (GLA) which is a variant of the k-means algorithm [**10**]. The GLA produces a codebook from a large set of training data that is similar to data to be coded. For example, if the images are photographs of people then so should be the set of training images. In this way vectors in the source image are likely to be closely approximated by vectors in the codebook. The GLA starts with some initial codebook and iteratively improves the codebook until the improvements become sufficiently small. This process can take many minutes to produce the final codebook. However, this is an offline cost since the same codebook is used to compress many images.

We see that encoding an image using VQ is simply a series of nearest neighbor searches. There are several properties of VQ codebooks and the searching process that influence the organization of codebooks for fast search. First, neighboring pixels in a vector in an image tend to be highly correlated. For example, if a pixel is dark then the pixel to its right is likely to be dark. Hence, the vectors that form an image tend to cluster on the diagonal rather than be more uniformly distributed. As the vector dimension grows this tendency to be clustered on the diagonal becomes less because the pixels in the vector are further apart in the image. Second, typically vectors are encoded one row at a time from top to bottom. This means that the nearest neighbor in the codebook to one vector is likely to be close to the nearest neighbor for the next vector. Again, this tendency is less as the dimension grows.

In all algorithms below we let n be the number of codewords in the codebook and let k be the dimension of the codewords. In order to achieve data compression it is necessary to have $\log n = O(k)$. For example, a k-dimensional codeword of 8 bit pixels uses $8k$ bits. A codeword requires $\log_2 n$ bits. Hence, VQ achieves a compression ratio of $8k : \log_2 n$. Unless $\log_2 n < 8k$ there is no compression. To be more concrete, if $k = 4$ then we should only consider codebooks of size $< 2^{32}$. Consequently asymptotic analyses of nearest neighbor search algorithms with n becoming arbitrarily large and fixed dimension may be inappropriate.

3. Nearest Neighbor Search Algorithms

As we saw above, the nearest neighbor search problem that arises in VQ is to find the codeword $\mathbf{c}_i$ with minimal Euclidean distance to the query vector $\mathbf{q}$. The "brute–force" solution to the nearest neighbor search problem is simply to compute the distance between $\mathbf{q}$ and each codeword $\mathbf{c}_i$ and keep track of which was the closest. The idea of each of the algorithms below is to try to search only a subset of the codewords. In the worst case, this subset could include every codeword; however, in the average case, this subset will include only a small fraction of them.

For all of these algorithms, including "brute–force," there are two standard optimizations that should be used [**2**]. The first is to notice that minimizing $\|\mathbf{q} - \mathbf{c}_i\|$ is equivalent to minimizing $\|\mathbf{q} - \mathbf{c}_i\|^2$, and thus we can avoid performing most square root computations by using squared distances rather than distances. The second comes from noticing that most often when we are computing the quantity $\|\mathbf{q} - \mathbf{c}_i\|^2 = \sum_{j=1}^{k}((\mathbf{q})_j - (\mathbf{c}_i)_j)^2$, we only want to know whether it is less than smallest squared distance found thus far. Thus, if at some point the partial sum becomes larger than the smallest squared distance found thus far, we can stop the computation.

We will use two metrics in measuring the performance of the algorithms. The first is the execution time of the algorithm. The second, which is a major component of the first, is the *number of codewords searched* which is the number of distance computations made.

3.1. Orchard's Method.

Orchard's Method is a nearest neighbor search algorithm designed specifically for Vector Quantization [**11**]. It is based on the geometric observation shown in Figure 2. (This picture is drawn in 2D; however, the result is true in any dimension.) Suppose that $\mathbf{c}_g$ is the closest codeword seen thus far, with $r = \|\mathbf{q} - \mathbf{c}_g\|$. Any codeword $\mathbf{c}_i$ that is closer to $\mathbf{q}$ than $\mathbf{c}_g$ must be no further than $2r$ from $\mathbf{c}_g$, or in symbols, $\|\mathbf{c}_i - \mathbf{c}_g\| \leq 2r$.

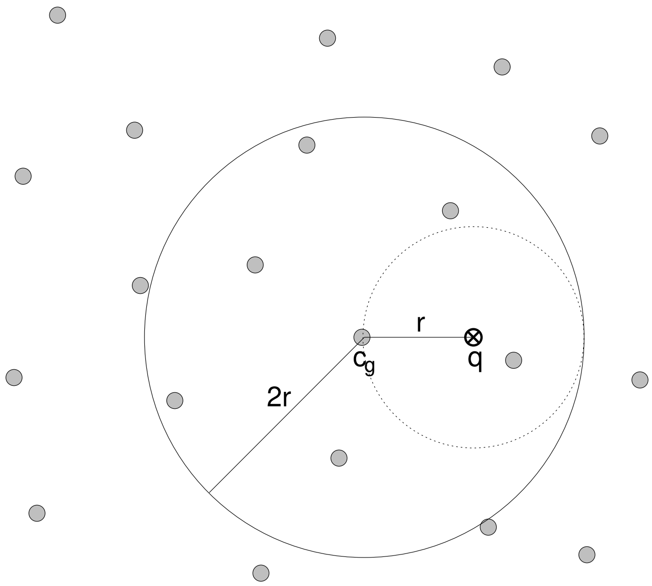

FIGURE 2. Orchard's Method

If we had a list of all the codewords sorted by their distance to $\mathbf{c}_g$ then we could simply enumerate those whose distance to $\mathbf{c}_g$ was less than $2r$ (which would be at the front of the list). Unfortunately, since $\mathbf{c}_g$ could be any codeword, we must maintain such a list for every codeword. More properly, the indices of the sorted codewords can be stored in these lists requiring $O(n^2)$ space.

Since $O(n^2)$ space is impractical for all but the smallest codebooks, a variation has been suggested which uses less space. The idea is to keep only the first m entries in each list for some number m. This requires $O(mn)$ space. If during our search we need to reference past the mth item in one of the lists, then we resort to a full search of every codeword. A smaller value of m will save space; however, it will also decrease performance.

One question that arises when implementing Orchard's Method is what to do when we encounter a codeword that is closer than the closest seen so far: do we keep enumerating through the current list or do we switch to the list of the codeword just discovered? Orchard's original paper [**11**] implemented the latter and we empirically verified that the latter is definitely the right thing to do. However, when this is done, care must be taken not to examine the same codeword more than once. This can be dealt with by marking codewords as they are examined. The marking process can be accomplished by maintaining an array of length n, indexed 1 to n for each codeword. Each member of the marking array has a bit indicating if the codeword has been searched and a pointer (index) to the last codeword that had been searched. Initially all the bits are zero and all the pointers are null. Before searching a codeword its bit is checked to see if it was already searched. When a codeword is searched its bit is switched from 0 to 1 and its pointer points to the last codeword searched. In the end, when the nearest codeword is found, the searched codewords are located by following the linked list, switching each 1 to 0 and setting each pointer to null. This returns the marking array to its initial state for the

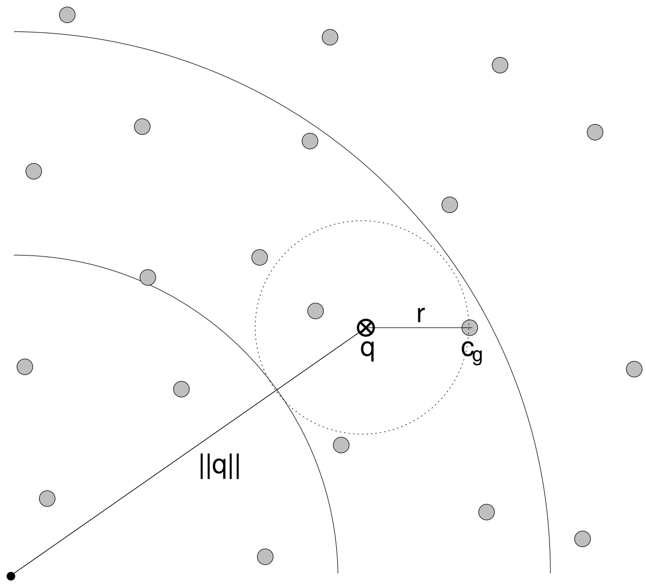

FIGURE 3. Annulus Method

next nearest neighbor search. In this way the time to find the nearest codeword is proportional to the number of codewords searched.

Another question that arises is which list to start processing initially. This choice has a large impact on the running time of this algorithm. To see this, note that the codewords enumerated are those that lie in the circle centered at $\mathbf{c}_g$ with a radius of twice the distance from $\mathbf{c}_g$ to $\mathbf{q}$. Thus, the number of codewords enumerated should grow as the distance between $\mathbf{c}_g$ and $\mathbf{q}$ grows. This means that we should try to pick an initial $\mathbf{c}_g$ that it is as close to $\mathbf{q}$ as possible. In other words, $\mathbf{c}_g$ should be a good guess of the closest codeword.

It remains for us to determine how to pick this guess codeword $\mathbf{c}_g$. It turns out that in VQ there is a very natural way to pick it: use the codeword that was closest to the last block processed. As mentioned earlier these two blocks are most likely adjacent to each other in the source image, they will probably be made up of very similar pixels. Thus, any codeword that is close to one will probably be close to the other. This prediction mechanism is very good in practice. Note that even if the prediction is correct other codewords inside the circle of radius $2r$ centered at the correct codeword must be searched to make sure it is the nearest.

3.2. Annulus Method. The Annulus Method is another nearest neighbor search algorithm designed specifically for VQ [**7**]. It is also based on a geometric observation. As is shown in Figure 3, any codeword $\mathbf{c}_i$ that is closer to $\mathbf{q}$ than $\mathbf{c}_g$ must satisfy $\|\mathbf{q}\| - r \le \|\mathbf{c}_i\| \le \|\mathbf{q}\| + r$, where $r = \|\mathbf{q} - \mathbf{c}_g\|$. This constraint defines an annular region.

To implement this algorithm, we maintain a list of the codewords sorted by their norm. We use the same guess codeword $\mathbf{c}_g$ as in Orchard's Method. To find the closest codeword, we enumerate those codewords whose norms lie in the range $[\|\mathbf{q}\| - r, \|\mathbf{q}\| + r]$. As with Orchard's Method, we can shrink the radius r

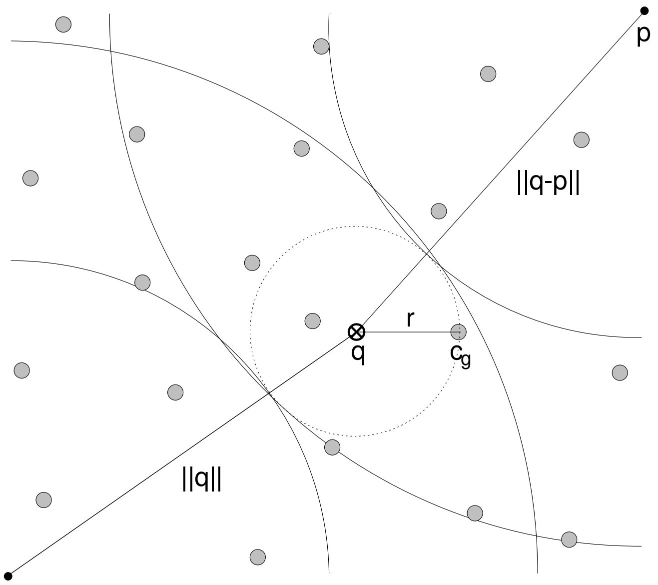

FIGURE 4. Double Annulus Method

of this search as we find closer codewords. However, we discovered as part of our experiments that this has little impact on performance.

The running time of this algorithm can be made proportional to the number of codewords which lie in the annulus. We can enumerate the codewords whose norms lie in this range without any additional work by taking advantage of the fact that we already know the index of $\mathbf{c}_g$ in the array. Starting with $\mathbf{c}_g$ we can traverse the list of codewords in both directions until the distance from the searched codewords to $\mathbf{q}$ exceeds r.

The main advantage of the Annulus Method over Orchard's Method is that the additional storage requirement is $O(n)$ rather than $O(n^2)$. The total storage requirement for the Annulus Method is $O(kn)$ which is the storage needed for the codebook itself.

3.3. Double Annulus Method. The Double Annulus Method is our third nearest neighbor search algorithm designed specifically for VQ [**9**]. The idea is to have two annular constraints and to try to search only those vectors lying in their intersection (see Figure 4). The second annulus is centered at a fixed vector $\mathbf{p}$ that is suitably chosen. Since the first annulus is centered at the origin, a good choice for $\mathbf{p}$ is where $(\mathbf{p})_j$ is the maximum pixel value for $1 \leq j \leq k$. Since the codewords tend to be clustered on the diagonal this choice of centers tends to eliminate from the intersection codewords that are in one annulus but far from the query vector. The two annuli require maintaining two sorted lists one for $\|\mathbf{c}_i\|$ and one for $\|\mathbf{c}_i - \mathbf{p}\|$ for $1 \leq i \leq n$. The the number of codewords searched by this algorithm can be made proportional to the number of codewords which lie in the intersection of the two annuli plus $\log n$. To accomplish this first do two binary searches on the first list to find the first and last codewords inside the first annulus. Similarly, find the first and last codewords in the second annulus. This requires $O(\log n)$ distance

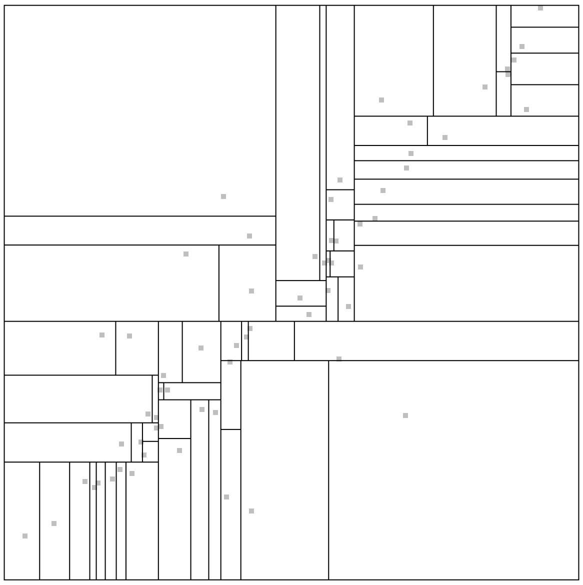

FIGURE 5. *k-d* Tree Splitting Lines

computations (searches). Next, find the intersection of the two lists, which contains at least $\mathbf{c}_g$. Finally, find the closest codeword to $\mathbf{q}$ in the intersection. Technically, the running time is proportional to the number of codewords in the union (not the intersection) of the annuli plus $\log n$, while the number of codeword searched is proportional to the number of codewords in the intersection (not the union) of the annuli plus $\log n$. The Double Annulus Method, like the Annulus Method, also uses storage $O(kn)$.

3.4. ***k-d* Tree Search.** The *k-d* tree is a multidimensional binary tree data structure that supports nearest neighbor and other spatial searches [**3, 4**]. Improvements to nearest neighbor search in *k-d* trees have been made [**13, 1**], but for this study we adhere to the basic algorithm described below.

A leaf of the *k-d* tree stores a k-dimensional vector (a codeword in our case). An internal node stores an index j and a value v that are used to define its left and right subtrees. The left subtree contains those nodes whose jth component is less than or equal to v, and the right subtree contains those nodes whose jth component is greater than v. For 2D vectors, each internal node partitions the data along a line which is perpendicular to one of the axes as shown in Figure 5. For example, an internal node with $j = 2$ and $v = 15$ partitions the vectors along the horizontal line $y = 15$. All vectors below the line have y–components less than or equal to 15, so they are in the left subtree, and all vectors above the line have y–components greater than 15, so they are in the right subtree. In higher dimension, each internal node partitions the data along a *splitting plane* that is perpendicular to some axis. A balanced *k-d* tree of height $\lceil \log_2 n \rceil$ can be built in $O(kn \log n)$ time and uses $O(kn)$ space.

The nearest neighbor search algorithm for *k-d* trees starts at the root of the tree and proceeds down toward the leaves recursively. At a leaf node, we simply compute the distance between the query vector and the vector stored in the leaf. At an internal node, we first search recursively down whichever subtree the query vector falls in. If the jth component of the query vector is less than or equal to v, then we search down the left subtree; otherwise, we search down the right subtree. When that recursive call returns, we check to see whether we must search the other subtree as well. If the distance between the query vector and the splitting plane is less than distance between the query vector and the closest codeword found thus far, then it is possible that there is a closer vector in the other subtree, so we must search there as well.

Unlike the previous algorithms that were designed specifically for VQ, the k-d tree nearest neighbor algorithm does not take advantage of the guess codeword from a previous nearest neighbor search. For this study we decided to keep to the basic algorithm as defined in the literature in order to compare a general purpose nearest neighbor search algorithm with those that were designed specifically for VQ. Even if we were to try to use the guess codeword in a new k-d tree nearest neighbor search algorithm, the new algorithm must eventually search the first codeword searched in the standard algorithm. This is because the query vector is in the leaf rectangle containing this first codeword searched, thereby creating the possibility that the distance between them is zero.

Friedman, Bentley, and Finkel [**4**] have shown that, in a reasonable model, the number of vectors searched in k-d tree nearest neighbor search is independent of n and the execution time is $O(\log n)$. In their model queries are given by a probability distribution and the codewords (keys in their terminology) are chosen from the same distribution. This situation is similar to the codebook being derived from a good training set by the GLA. They report, based on their own empirical studies, that the asymptotic result of a constant number of codewords searched per nearest neighbor search only applies to very large codebooks. Hence, their result has limited applicability to VQ because of the constraint that $\log n = O(k)$ to achieve data compression.

3.5. PCP Tree Search. The Principal Component Partitioning (PCP) tree [**12, 16, 15**] is a variation on a k-d tree that uses splitting planes which are usually not perpendicular to the axes. Each splitting plane is perpendicular to the principal component of the data it is partitioning. The principal component is an affine vector that points along the line which minimizes the sum of the squared distances to each vector (this is the "best fit" line of the vectors). Figure 6 shows the splitting planes (lines in 2D) of a PCP tree used to partition a set of codewords taken from a typical codebook.

The PCP tree was not originally designed for nearest neighbor search so why is the PCP tree a good vehicle for nearest neighbor search?

Suppose that we were allowed to choose a splitting line for a set of codewords. How would we characterize a good choice? First, we would like this line to split the codewords in half. If we do this at every internal node, then the resulting tree will be perfectly balanced. Second, we would like to choose our line so as to minimize the number of times that we have to search both sides when performing a nearest neighbor search. We are forced to search both sides when the distance from the query vector to the closest codeword is greater than the distance from the query vector to the splitting line. Thus, if we increase the expected distance from the

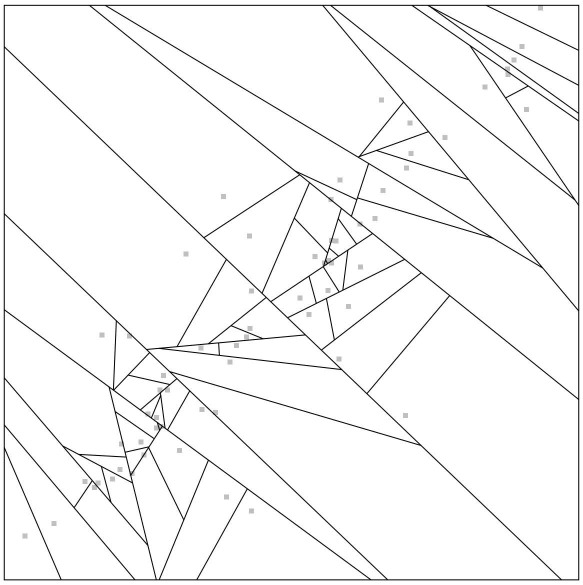

FIGURE 6. PCP Tree Splitting Lines

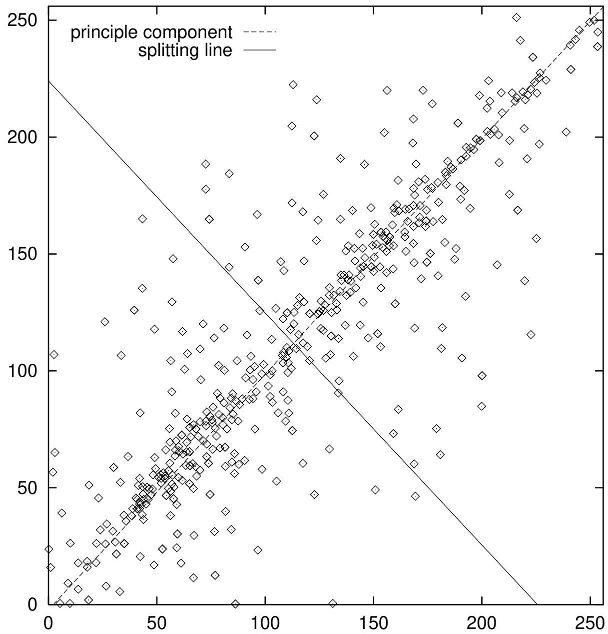

FIGURE 7. Principal Component Partitioning

query vector to the splitting line, we would expect to decrease the number of times that we are forced to search both sides.

It remains to determine how we can increase the expected distance from the query vector to the splitting line. First, we should note that we can expect the query vectors to be distributed very similarly to the codewords: the codewords are supposed to represent well the blocks that will occur in the source image. Therefore, we can increase the expected distance from the query vector to the splitting line by increasing the expected distance from a codeword to the splitting line. For example, Figure 7 shows a set of real 2×1 codewords taken from a typical codebook. The dotted line, along which the vectors are highly clustered, passes through the principal component. The solid line is perpendicular to the principal component. This is the line that principal component partitioning would choose, and as we can see, only a small number of vectors are very close to it. The *k-d* tree would choose a horizontal or vertical splitting line that partitions the vectors in half. In either case, this line would have more vectors close to it than the one chosen by principal component partitioning.

Building a PCP tree requires more time than building a *k-d* tree. To create each internal node, we must find the principal component of the codewords in that subtree. This is done by solving a small eigenvalue problem. To be specific, let $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_m\}$ be a set of k-dimensional vectors (codewords in a subtree). To compute the principal component of X we compute the dominant eigenvalue of the symmetric $k \times k$ matrix $C = A^T A$, where A is the $m \times k$ matrix with entries $a_{ij} = (\mathbf{x}_i)_j - (\overline{\mathbf{x}})_j$. The vector $\overline{\mathbf{x}}$ is the centroid of X, defined by $(\overline{\mathbf{x}})_j = \sum_{i=1}^{m} (\mathbf{x}_i)_j / m$. The principal component of X is the affine vector that is in the direction of the dominant eigenvector of C and goes through the centroid $\overline{\mathbf{x}}$ of X. Computing dominant eigenvector of a $k \times k$ matrix can be done with the power method, which used negligible execution time in our studies. The power method is an iteration method for finding eigenvectors whose number of iterations depends on properties of the matrix and the desired accuracy [**14**]. Once the principal component of the set X is found, then a splitting plane is determined by choosing a plane perpendicular to the principal component of X that partitions X in half. A balanced PCP tree can be built in $O(k^2 n \log n)$ time ignoring the cost of computing the dominant eigenvectors. This bound comes from computing $A^T A$ for 2^i different matrices at level i of the tree where A is a $n/2^i \times k$ matrix. The PCP tree requires $O(kn)$ space with all the nodes, internal and leaves, using $O(k)$ storage. This is in contrast to the *k-d* tree where the internal nodes use constant size and only the leaves have size $O(k)$.

The nearest neighbor search algorithm for the PCP tree is conceptually the same as for *k-d* trees. The only differences are in how we determine which side of the splitting plane the query vector is on and how we compute the distance from the query vector to the splitting plane. For PCP trees, both operations require $O(k)$ time, whereas with *k-d* trees, both operations require only constant time. Since these two operations are more expensive in PCP trees, the nearest neighbor search algorithm will have to examine fewer codewords if it is to be faster than the *k-d* tree algorithm. Like the *k-d* tree algorithm, the PCP tree nearest neighbor search algorithm does not use the guess vector from a previous nearest neighbor search.

4. Experimental Results

In our study, we compare the six algorithms described above using two metrics. Since our ultimate goal is to find the fastest algorithm, our first metric is execution

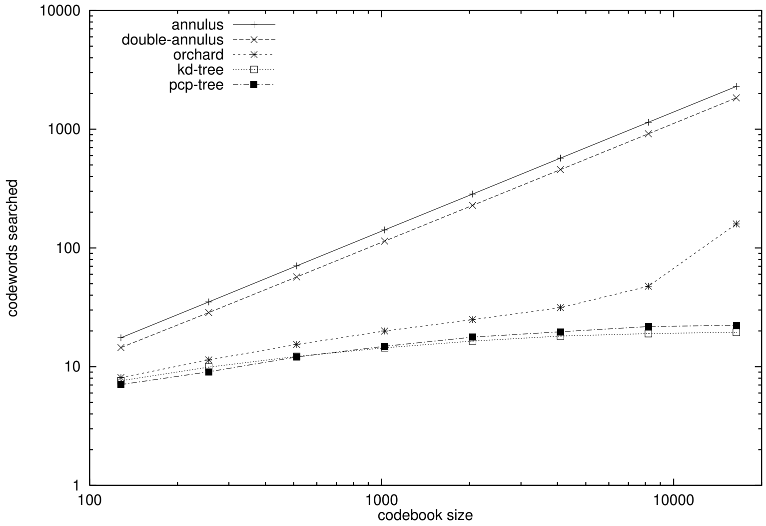

FIGURE 8. Codewords Searched in 4D

time. Our second metric is the number of codewords searched. This second metric should be a good predictor of execution time because distance computations account for the bulk of the processing done by most algorithms. In addition, this second metric will give us extra insight into the behavior of the algorithm.

The execution times and codewords searched that are presented include only the nearest neighbor searches. Preprocessing is not included. This is because the preprocessing time is greatly dominated by the GLA. In addition, as was mentioned earlier, the same codebook will be used to compress many images, so the preprocessing is an offline cost.

The algorithms were run on a total of up to 7 images (baboon, man, airport, couple, lena, and tiffany, grass) from the USC-SIPI Image Database [**8**]. All the images are 512×512 8 bit gray-scale. For the 4D and 16D we used one of the images, Lena, for training and the other 6 for testing. For 64D one image does not provide enough training data so we used all seven images for both training and testing. The experiments were performed on a 500 MHz DEC Alpha with 128MB of memory and running the Linux operating system[1]. Execution times reported in Figures 9, 11, 13, and 16 are the average execution times per nearest neighbor search for VQ encoding all the test images. The codewords searched in Figures 8, 10, and 12 are the average numbers of codewords searched per nearest neighbor search for VQ encoding all the test images.

Let's start by looking at the results in a low dimension, 4D (2×2 blocks). Figure 8 shows the number of codewords searched by the different algorithms. At the bottom are the two tree methods, *k-d* and PCP trees, which searched about the same number of codewords. Just above is Orchard's Method. At the top are Annulus Method and Double Annulus Method. Notice that all of the algorithms search far fewer codewords than would the "brute–force" algorithm. For example,

[1]The experiments were also run on the same machine using the Windows NT operating system and produced nearly identical results.

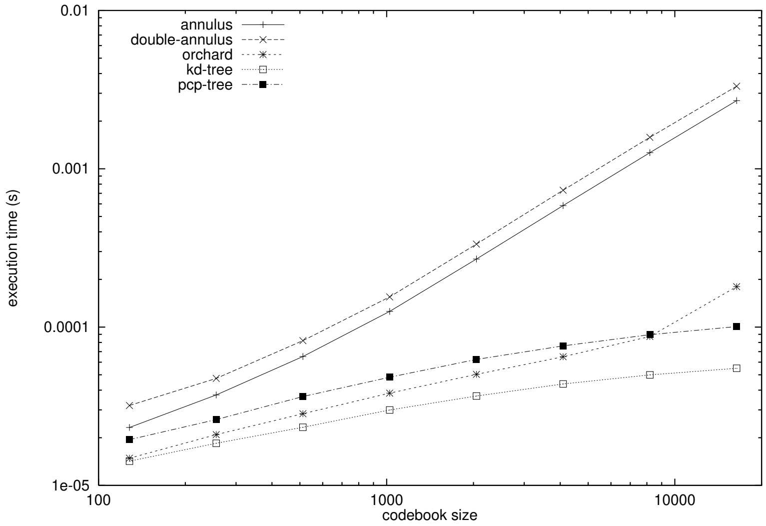

FIGURE 9. Execution Time in 4D

with a codebook containing 16,384 codewords (the last data point), these algorithms all search fewer than 10% of the codewords.

Notice that the plot of Orchard's Method bends up sharply at the larger codebook sizes. For large n a full $O(n^2)$ matrix would exceed the computer's memory causing the virtual memory system to do excessive paging. To avoid this the length of the sorted lists had to be decreased when searching larger codebooks. We restricted the list length so that the total memory usage was no more than 16 MB (still far more memory than was used by any other algorithm) to avoid paging. For all codebooks with up to 1,024 codewords, full length lists were used in all dimensions.

In Figure 8, we can see that Double Annulus Method searches only slightly fewer codewords than Annulus Method. The reason for such a mild increase is that the codewords tend to be clustered along the diagonal and the centers of the two annuli are at each end of the diagonal. The intersection primarily eliminates those few codewords in one annulus but far from the diagonal.

Figure 9 shows the execution time for the different algorithms in 4D. The results look quite similar to the those in the previous figure. This shows us that the number of codewords searched is a reasonably good predictor of execution time. However, there are two key differences between the two plots. The first difference is that PCP tree search was slower than k-d tree search while they searched about the same number of codewords. This is a result of the fact that the PCP tree algorithm did more work at each internal node: $O(k)$ instead of $O(1)$ work. The other difference is that is that Double Annulus Method was slower than Annulus Method while Double Annulus Method searched fewer codewords.

Next, let's look at the results in higher dimension, 16D (4×4 blocks). Figure 10 shows the number of codewords searched for the different algorithms. The only major difference from 4D is that the k-d tree algorithm moved up quite a bit compared to the other algorithms. When we look at Figure 11, which shows the

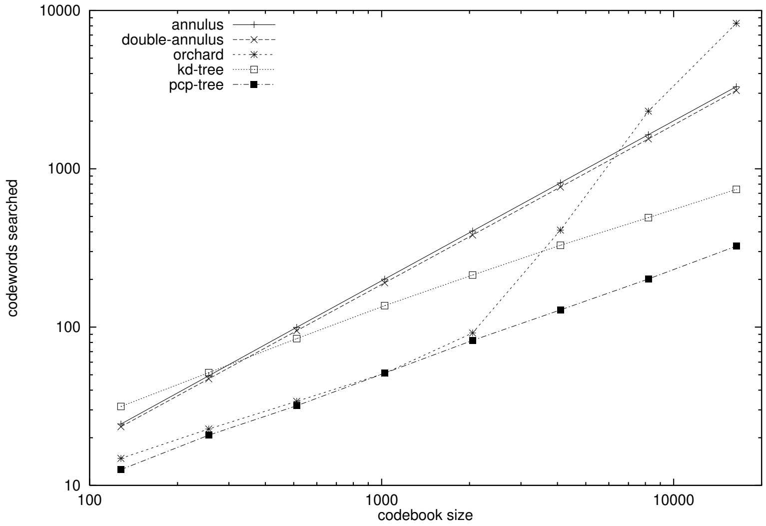

FIGURE 10. Codewords Searched in 16D

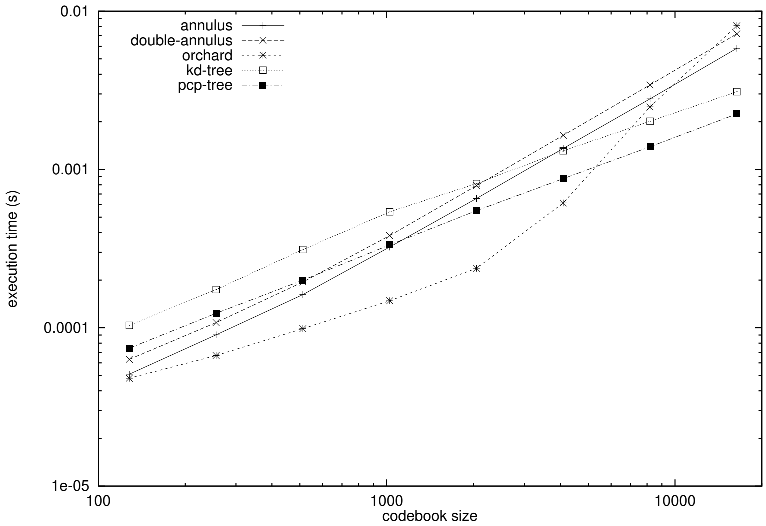

FIGURE 11. Execution Time in 16D

execution time of the different algorithms in 16D, we see that this resulted in a faster execution time for the PCP tree algorithm than the *k-d* tree algorithm. For small codebooks, both were greatly out done by Orchard's Method. However, Orchard's Method slows down considerably, once again, as the lists are shortened. Annulus Method and Double Annulus Method also gained a little ground compared to the other algorithms.

Lastly, Figure 12 and Figure 13 show the number of codewords searched and execution time, respectively, of the different algorithms in 64D (8×8 blocks). Again, it appears that the *k-d* tree algorithm is losing ground in higher dimension, while

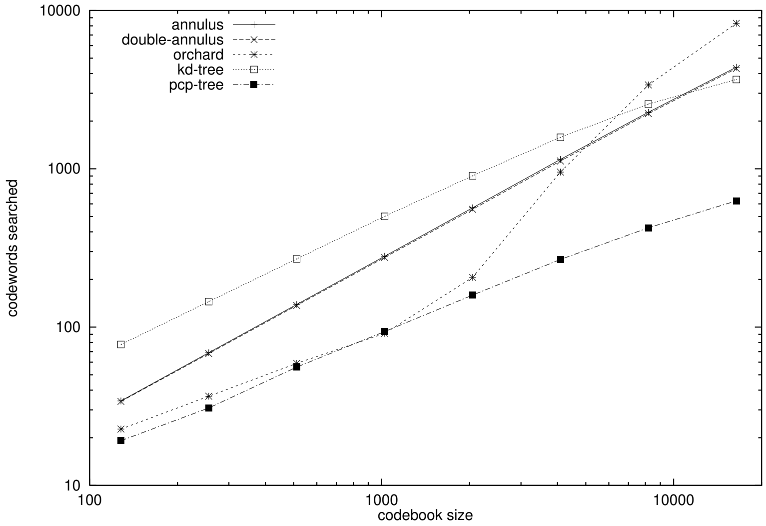

FIGURE 12. Codewords Searched in 64D

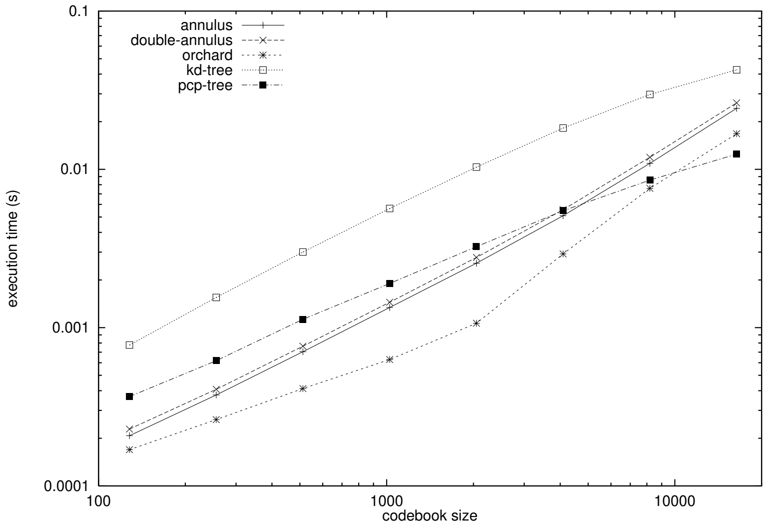

FIGURE 13. Execution Time in 64D

the Annulus Method and Double Annulus Method gain ground. Again, Orchard's Method performs extremely well for small codebooks, but slows down as the lists are shortened.

5. PCP/k-d Trees

The PCP tree was used in order to take advantage of the fact that the data are clustered along a line. Unfortunately, this fact is only true on the large scale: a small collection of neighboring vectors do not exhibit such clustering (in fact, they

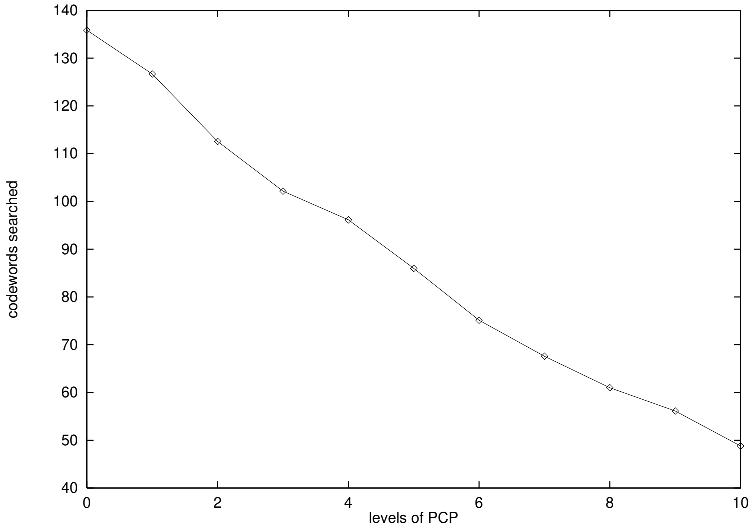

FIGURE 14. Levels of PCP v. Codewords Searched

appear to be almost uniformly random). The PCP nodes near the top of the tree are worth the extra expense, but the PCP nodes near the bottom of the tree are not earning their keep.

One solution is to use a mixed PCP/*k-d* tree. The first m levels of the tree are made up of the expensive PCP nodes and the rest of the levels are made up of lightweight *k-d* nodes. The number m is a parameter which we can vary.

Figure 14 shows the number of codewords searched as m, the number of levels of PCP nodes, is varied. (This figure is using 1024 16D codewords.) We can see that every extra level of PCP nodes causes fewer codeword to be searched. However, the levels near the bottom of the tree cause less of a decrease than those at the top. Figure 15 shows the execution time as m is varied. We can see that at some point adding extra levels of PCP nodes does not increase performance: the decrease in codewords searched does not make up for the extra expense of the PCP nodes. This is the point where the curve bottoms out.

The value of m for which the PCP/*k-d* tree performs best appears to depend on the dimension. In lower dimension, where the *k-d* tree performs well, the best value of m will be smaller. In higher dimension, where the PCP tree performs well, the best value of m will be larger. In general, the best value of m should remain strictly less than the height of the tree: it is almost never beneficial to use a PCP node to partition 2 vectors.

Figure 16 shows the running times of the algorithms in 4, 16, and 64 dimensions for a codebook of size 4,096. (Double–Annulus Method was left out as it always performed worse than Annulus Method.) We can see that PCP/*k-d* tree performs well in all cases. In 4D, the best PCP/*k-d* tree is a *k-d* tree, so its running time is equal to that of *k-d* tree which is the fastest. In 16D, the PCP/*k-d* tree is around 45% faster than *k-d* tree and around 15% faster than PCP Tree. In 64D, the PCP/*k-d* tree is around 75% faster than *k-d* tree and around 20% faster than PCP tree.

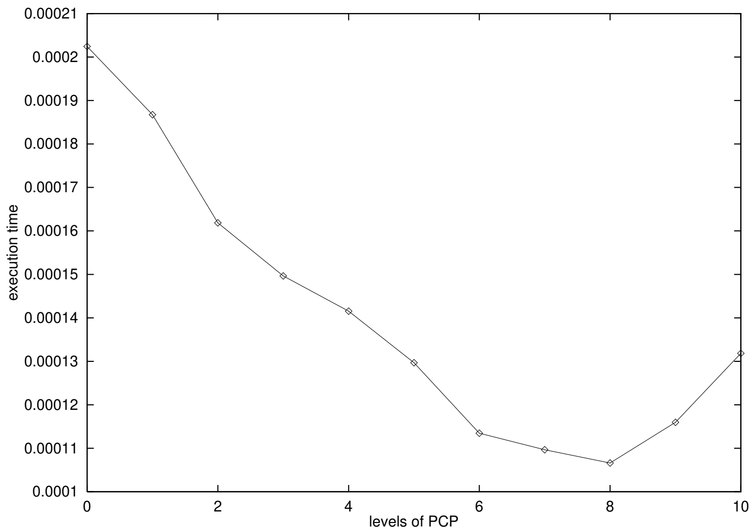

FIGURE 15. Levels of PCP v. Execution Time

	Orchard	Annulus	k-d	PCP	PCP/k-d
4D	0.065	0.585	0.044	0.076	0.044
16D	0.61	1.36	1.31	0.87	0.73
64D	2.9	5.1	18.2	5.4	4.3

FIGURE 16. Running Times of the Algorithms for 4,096 Codewords (in msec)

6. Summary

In this paper, we saw six nearest neighbor search algorithms for performing Vector Quantization, all of which greatly outperform "brute-force" search. We summarize our conclusions as follows.

- Orchard's Method has wonderful performance, especially in high dimension. Unfortunately, its $O(n^2)$ memory requirement make it impractical for large codebooks. We can lessen its memory requirement by using shorter lists; however, this quickly decreases performance.
- The Annulus Method is a simple algorithm that works well for small codebooks in high dimension.
- The Double Annulus Method is a variation on Annulus Method that does not appear to have better performance than the simpler Annulus Method in most situations.
- The k-d tree nearest neighbor search algorithm is the fastest algorithm in low dimension.
- The PCP tree nearest neighbor search algorithm is faster than the k-d tree algorithm in higher dimension. However, the nodes near the bottom of the tree provide little benefit over k-d nodes despite their extra expense.

- The PCP/k-d tree combines the strengths of PCP and k-d trees to obtain good performance in all dimensions and codebook sizes.

References

[1] S. Arya and D.M. Mount. Algorithms for fast vector quantization. In *Proceedings Data Compression Conference*, pages 381–390. IEEE, March 1993.

[2] C. D. Bei and R. M. Gray. An improvement of the minimum distortion encoding algorithm for vector quantization. *IEEE Transactions on Communications*, pages 1132–1133, October 1985.

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

[4] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[5] A. Gersho and R.M. Gray. *Vector Quantization and Signal Compression*. Kluwer, Boston, MA, July 1992.

[6] R. M. Gray. Vector quantization. *IEEE ASSP Magazine*, 1:4–29, April 1984.

[7] C. M. Huang, Q. Bi, G. S. Stiles, and R. W Harris. Fast full search equivalent encoding algorithms for image compression using vector quantization. *IEEE Transactions on Image Processing*, 1(3):413–416, July 1992.

[8] USC-SIPI Image Database. http://sipi.usc.edu/services/database/database.html.

[9] M. H. Johnson, R. Ladner, and E. A. Riskin. Fast nearest neighbor search of entropy-constrained vector quantization. *IEEE Transactions on Image Processing*, 9:1435–1437, August 2000.

[10] Y. Linde, A. Bruzo, and R. M. Gray. An algorithm for vector quantizer design. *IEEE Transactions on Communication*, 28:84–95, January 1980.

[11] Michael T. Orchard. A fast nearest-neighbor search algorithm. In *ICASSP 91: 1991 International Conference on Acoustics, Speech and Signal Processing*, volume 4, pages 2297–3000, New York, NY, USA, April 1991. IEEE.

[12] E. A. Riskin, R. Ladner, R.-Y. Wang, and L. E. Atlas. Index assignment for progressive transmission of full-search vector quantization. *IEEE Transactions on Image Processing*, 3(3):307–312, May 1994.

[13] R.L. Sproull. Refinements to nearest-neighbor searching in k dimensional trees. *Algorithmica*, 6:579–589, 1991.

[14] G.W. Stewart. *Introdution to Matrix Computations*. Academic Press, New York, NY, July 1973.

[15] R.-Y. Wang. *Organization of Fixed Rate Vector Quantization Codebooks*. Dissertation, University of Washington, July 1994.

[16] X. Wu and K. Zhang. A better tree-structured vector quantizer. In *Proceedings Data Compression Conference*, pages 392–401. IEEE, April 1991.

Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, USA

E-mail address: `kevinz@cs.washington.edu`

Department of Electrical Engineering, University of Washington, Seattle, WA 98195, USA

E-mail address: `mhjohns@cs.washington.edu`

Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, USA

E-mail address: `ladner@cs.washington.edu`