

Efficient Multidimensional Searching Routines for Music Information Retrieval

Josh Reiss, Jean-Julien Aucouturier, Mark Sandler
Department of Electrical Engineering, King's College, London
Strand, London WC2 2LR
UK

Phone: +44 020 7848 2041

{josh.reiss, jean-julien.aucouturier, mark.sandler}@kcl.ac.uk

ABSTRACT

The problem of Music Information Retrieval can often be formalized as “searching for multidimensional trajectories”. It is well known that string-matching techniques provide robust and effective theoretic solutions to this problem. However, for low dimensional searches, especially queries concerning a single vector as opposed to a series of vectors, there are a wide variety of other methods available. In this work we examine and benchmark those methods and attempt to determine if they may be useful in the field of information retrieval. Notably, we propose the use of *KD-Trees* for multidimensional near-neighbor searching. We show that a KD-Tree is optimized for multidimensional data, and is preferred over other methods that have been suggested, such as the *K-Tree*, the *box-assisted sort* and the *multidimensional quick-sort*.

1. MULTIDIMENSIONAL SEARCHING IN MUSIC IR

The generic task in Music IR is to search for a query pattern, either a few seconds of raw acoustic data, or some type of symbolic file (such as MIDI), in a database of the same format.

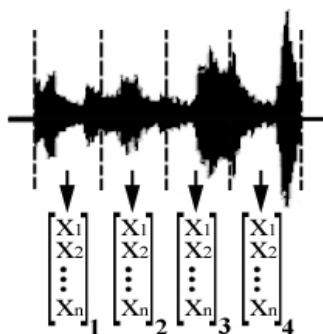


Figure 1- Feature extraction

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page.

To perform this task, we have to encode the files in a convenient way. If the files are raw acoustic data, we often resort to a *feature extraction* (fig. 1). The files are cut into M time frames and for each frame, we apply a signal-processing transform that outputs a vector of N features (e.g. psychoacoustics parameters such as pitch, loudness, brightness, etc...). If the data is symbolic, we similarly encode each symbol (e.g. each note, suppose there are M of them) with a N -dimensional vector (e.g. pitch, duration). In both cases, the files in the database are turned into a trajectory of M vectors of dimension N .

Within this framework, two search strategies can be considered:

- String-matching techniques try to align the two sequences of vectors $\{\vec{x}_1, \vec{x}_2, \vec{x}_3\}$ and $\{\vec{y}_1, \vec{y}_2, \vec{y}_3\}$ using a set of elementary operations (substitutions, insertions...). They have received much coverage in the Music IR community (see for example [1]) since they allow a context-dependent measure of similarity and thus can account for many of the high-level specificities of a musical query (i.e. replacing a note by its octave shouldn't be a mismatch). They are robust and relatively fast.

- The other approach would be to “fold” the trajectories of M vectors of dimension N into embedded vectors of higher dimension $M*N$. For example, with $M=3$ and $N=2$:

$$\{(x, y)_1, (x, y)_2, (x, y)_3\} = (x_1, y_1, x_2, y_2, x_3, y_3)$$

The search problem now consists of identifying the nearest vector in a multidimensional data set (i.e. the database) to some specified vector (i.e. the query). This approach may seem awkward, as

- We loose structure in the data that could be used to help the search routines (e.g. the knowledge that x_1 and x_2 are coordinates of the same “kind”).

- We increase the dimensionality of the search.

However, there has been a considerable amount of work in devising very efficient searching and sorting routines for such multidimensional data. A complete review of the multidimensional data structures that might be required is described by Samet, et al. [2,3]. Non-hierarchical methods, such as the use of grid files [4] and extendable hashing [5], have been applied to multidimensional searching and analyzed extensively. In many areas of research, the KD-Tree has become accepted as

one of the most efficient and versatile methods of searching. This and other techniques have been studied in great detail throughout the field of computational geometry [6,7].

Therefore, we feel that Music IR should capitalize on these well-established techniques. It is our hope that we can shed some light on the beneficial uses of *KD-Trees* in this field, and how the multi-dimensional framework can be adapted to the peculiarities of music data.

The paper is organized as follows. In the next four sections, we review four multidimensional searching routines: The KD-Tree, the K-Tree, the Multidimensional Quick-sort, which is an original algorithm proposed by the authors, and the Box-Assisted Method. We then benchmark and compare these routines, with an emphasis on the very efficient KD-Tree algorithm. Finally, we examine some properties of these algorithms as regards a multidimensional approach to Music IR.

2. THE KD-TREE

2.1 Description

The K-dimensional binary search tree (or KD-Tree) is a highly adaptable, well-researched method for searching multidimensional data. This tree was first introduced in Bentley, et al.[9], studied extensively in[10] and a highly efficient and versatile implementation was described in [11]. It is this second implementation, and variations upon it, that we will be dealing with here.

There are two types of nodes in a KD-Tree, the terminal nodes and the internal nodes. The internal nodes have two children, a left and a right son. These children represent a k-dimensional partition of the hyperplane. Records on one side of the partition are stored in the left sub-tree, and on the other side are records stored in the right sub-tree. The terminal nodes are buckets which contain up to a set amount of points. A one-dimensional KD-Tree would in effect be a simple quick-sort.

2.2 Method

The building of the KD-Tree works by first determining which dimension of the data has the largest spread, i.e. difference between the maximum and the minimum. The sorting at the first node is then performed along that dimension. A quickselect algorithm, which runs in order n time, finds the midpoint of this data. The data is then sorted along a branch depending on whether it is larger or smaller than the midpoint. This succeeds in dividing the data set into two smaller data sets of equal size. The same procedure is used at each node to determine the branching of the smaller data sets residing at each node. When the number of data points contained at a node is smaller than or equal to a specified size, then that node becomes a bucket and the data contained within is no longer sorted.

Consider the following data:

**A (7,-3); B (4,2); C (-6,7); D (2,-1); E (8,0);
F (1,-8); G (5,-6); H (-8,9); I (9,8); J (-3,-4);**

Fig. 2 depicts the partition in 2 dimensions for this data set. At each node the cut dimension and the cut value (the median of the corresponding data) are stored. The bucket size has been chosen to be one.

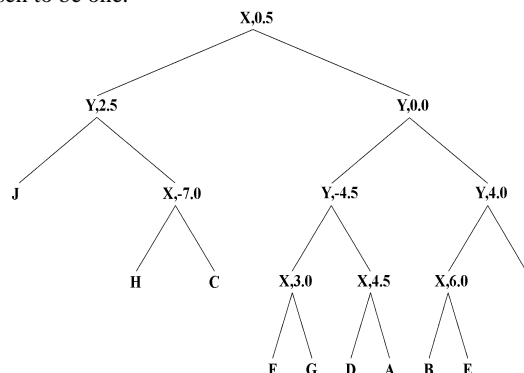


Figure 2- The KD-Tree created using the sample data.

The corresponding partitioning of the plane is given in Fig. 3. We note that this example comes from a larger data set and thus does not appear properly balanced. This data set will be used as an example in the discussion of other methods.

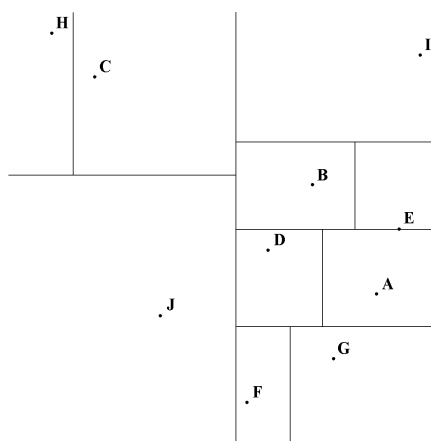


Figure 3- The sample data as partitioned using the KD-Tree method.

A nearest neighbor search may then be performed as a top-down recursive traversal of a portion of the tree. At each node, the query point is compared with the cut value along the specified cut dimension. If along the cut dimension the query point is less than the cut value, then the left branch is descended. Otherwise, the right branch is descended. When a bucket is reached, all points in the bucket are compared to see if any of them is closer than the distance to the nearest neighbor found so far. After the descent is completed, at any node encountered, if the distance to the closest neighbor found is greater than the distance to the cut value, then the other branch at that node needs to be descended as well. Searching stops when no more branches need to be descended.

Bentley recommends the use of parent nodes for each node in a tree structure. A search may then be performed using a bottom-up approach, starting with the bucket containing the search point and searching through a small number of buckets

until the appropriate neighbors have been found. For nearest neighbor searches this reduces computational time from $O(\log m)$ to $O(1)$. This however, does not immediately improve on search time for finding near neighbors of points *not* in the database. Timed trials indicated that the increased speed due to bottom-up (as opposed to top-down) searches was negligible. This is because most of the computational time is spent in distance calculations, and the reduced number of comparisons is negligible.

3. THE K-TREE

3.1 Description

K-trees are a generalization of the single-dimensional M-ary search tree. As a data comparative search tree, a K-tree stores data objects in both internal and leaf nodes. A hierarchical recursive subdivision of the k-dimensional search space is induced with the space partitions following the locality of the data. Each node in a K-tree contains $K=2^k$ child pointers. The root node of the tree represents the entire search space and each child of the root represents a *K-ant* of the parent space.

One of the disadvantages of the K-tree is its storage space requirements. In a standard implementation, as described here, a tree of N k-dimensional vectors requires a minimum of $(2^k + k) \cdot N$ fields. Only $N-1$ of the 2^k branches actually point to a node. The rest point to NULL data. For large k, this waste becomes prohibitive.

3.2 Method

Consider the case of two-dimensional data ($k=2$, $K=4$). This K-tree is known as a quad-tree, and is a 4-ary tree with each node possessing 4 child pointers. The search space is a plane and the partitioning induced by the structure is a hierarchical subdivision of the plane into disjoint quadrants. If the data consists of the 10 vectors described in Section 2.2, then the corresponding tree is depicted in Fig. 4 and the partitioning of the plane in Fig. 5.

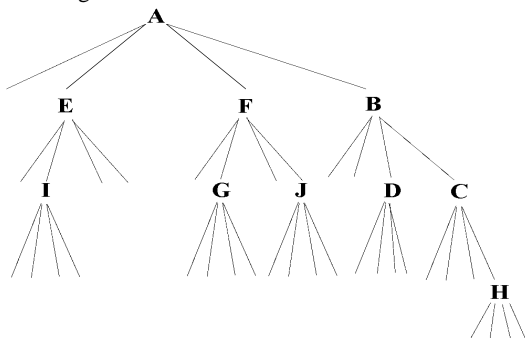


Figure 4- The KDTree created using the sample data.

Note that much of the tree is consumed by null pointers.

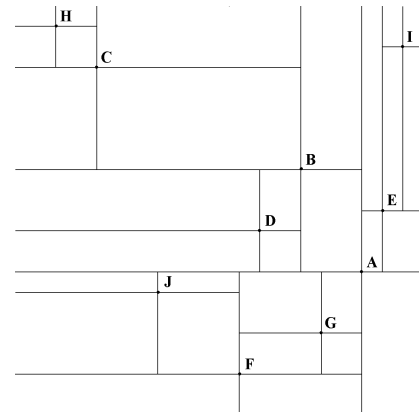


Figure 5- The sample data as partitioned using the KTree method.

Searching the tree is a recursive two-step process. A cube that corresponds to the bounding extent of the search sphere is intersected with the tree at each node encountered. The bounds of this cube are maintained in a k-dimensional range array. This array is initialized based on the search vector. At each node, the direction of search is determined based on this intersection. A search on a child is discontinued if the region represented by the child does not intersect the search cube. This same general method may be applied to weighted, radial, and nearest neighbor searches. For radial searches, the radius of the search sphere is fixed. For nearest neighbor searches it is doubled if the nearest neighbor has not been found, and for weighted searches it is doubled if enough neighbors have not been found.

4. MULTIDIMENSIONAL QUICKSORT

4.1 Description

For many analyses, one wishes to search only select dimensions of the data. A problem frequently encountered is that a different sort would need to be performed for each search based on a different dimension or subset of all the dimensions. We propose here a multidimensional generalization of the quick-sort routine.

4.2 Method

The data is described as a series of N vectors where each vector is D dimensional, $\overline{X}_k = (X_k^1, X_k^2, \dots, X_k^D)$. A quick-sort is performed on each of the D dimensions. The original array is not modified. Instead, two new arrays are created for each quick-sort. The first is the quick-sort array, an integer array where the value at position k in this array is the position in the data array of the k^{th} smallest value in this dimension. The second array is the inverted quick-sort. It is an integer array where the value at position k in the array is the position in the quick-sort array of the value k . Keeping both arrays allows one to identify both the location of a sorted value in the original array, and the location of a value in the sorted array. Thus, if \overline{X}_1 has the second smallest value in the third dimension, then it may be

represented as $\overline{X_{2,3}}$. The value stored at the second index in the quick-sort array for the third dimension will be 1, and the value stored at the first index in the inverted quick-sort array for the third dimension will be 2. Note that the additional memory overhead need not be large. For each floating-point value in the original data, two additional integer values are stored-, one from the quick-sort array and one from the inverted quick-sort array.

We begin by looking at a simple case and showing how the method can easily be generalized. We consider the case of two-dimensional data, with coordinates x and y , where we make no assumptions about delay coordinate embeddings or uniformity of data.

Suppose we wish to find the nearest neighbor of the vector $\overline{A} = (x, y)$. If this vector's position in the x -axis quick-sort is i and its position in the y -axis quick-sort is j (i and j are found using the inverted quick-sorts), then it may also be represented as $\overline{A} = \overline{A_{i,x}} = (x_{i,x}, y_{i,x}) = \overline{A_{j,y}} = (x_{j,y}, y_{j,y})$.

Using the quick-sorts, we search outward from the search vector, eliminating search directions as we go. Reasonable candidates for nearest neighbor are the nearest neighbors on either side in the x -axis quick-sort, and the nearest neighbors on either side in the y -axis quick-sort. The vector $\overline{A_{i-1,x}} = (x_{i-1,x}, y_{i-1,x})$ corresponding to position $i-1$ in the x -axis quick-sort is the vector with the closest x -coordinate such that $x_{i-1,x} < x$. Similarly, the vector $\overline{A_{i+1,x}} = (x_{i+1,x}, y_{i+1,x})$ corresponding to $i+1$ in the x -axis quick-sort is the vector with the closest x -coordinate such that $x_{i+1,x} > x$. And from the y -axis quick-sort, we have the vectors $\overline{A_{j-1,y}} = (x_{j-1,y}, y_{j-1,y})$ and $\overline{A_{j+1,y}} = (x_{j+1,y}, y_{j+1,y})$. These are the four vectors adjacent to the search vector in the two quick-sorts. Each vector's distance to the search vector is calculated and we store the minimal distance and the corresponding minimal vector. If $|x_{i-1,x} - x|$ is greater than the minimal distance, then we know that all vectors $\overline{A_{i-1,x}}, \overline{A_{i-2,x}}, \dots, \overline{A_{1,x}}$ must also be further away than the minimal vector. In that case, we will no longer search in decreasing values on the x -axis quick-sort. We would also no longer search in decreasing values on the x -axis quick-sort if $\overline{A_{i,x}}$ has been reached. Likewise, if $|x_{i+1,x} - x|$ is greater than the minimal distance, then we know that all vectors $\overline{A_{i+1,x}}, \overline{A_{i+2,x}}, \dots, \overline{A_{N,x}}$ must also be further away than the minimal vector. If either that is the case or $\overline{A_{N,x}}$ has been reached then we would no longer search in increasing values on the x -axis quick-sort. The same rule applies to $|y_{j-1,y} - y|$ and $|y_{j+1,y} - y|$.

We then look at the four vectors, $\overline{A_{i-2,x}}, \overline{A_{i+2,x}}, \overline{A_{j-2,y}}$ and $\overline{A_{j+2,y}}$. If any of these is closer than the minimal vector, then we replace the minimal vector with this one, and the minimal distance with this distance. If $|x_{i-2,x} - x|$ is greater than the minimal distance, then we no longer need to

continue searching in this direction. A similar comparison is made for $|x_{i+2,x} - x|, |y_{j-2,y} - y|$ and $|y_{j+2,y} - y|$.

This procedure is repeated for $\overline{A_{i-3,x}}, \overline{A_{i+3,x}}, \overline{A_{j-3,y}}$ and $\overline{A_{j+3,y}}$, and so on, until all search directions have been eliminated. We find which of these four vectors is closest to the search vector. We then search the next four closest points. If any of these points is further away in its direction of search than the minimal distance, then we can eliminate that direction from further search. Also, if we reach the end of the data set in any direction, then we can eliminate that direction. We find the distance of the four points from our point of interest and, if possible, replace the minimal distance. We then proceed to the next four points and proceed this way until all directions of search have been eliminated.

The minimal vector must be the nearest neighbor, since all other neighbor distances have either been calculated and found to be greater than the minimal distance, or have been shown that they must be greater than the minimal distance.

Extension of this algorithm to higher dimensions is straightforward. In n dimensions there are $2n$ possible directions. Thus $2n$ immediate neighbors are checked. A minimal distance is found, and then the next $2n$ neighbors are checked. This is continued until it can be shown that none of the $2n$ directions can contain a nearer neighbor.

It is easy to construct data sets for which this is a very inefficient search. For instance, if one is looking for the closest point to $(0,0)$ and one were to find a large quantity of points residing outside the circle of radius 1 but inside the square of side length 1 then all these points would need to be measured before the closer point at $(1,0)$ is considered. However, similar situations can be constructed for most multidimensional sort and search methods, and preventative measures can be taken.

5. THE BOX-ASSISTED METHOD

The box-assisted search method was described by Schreiber, et al.^[11] as a simple multidimensional search method for nonlinear time series analysis. A grid is created and all the vectors are sorted into boxes in the grid. Fig. 6 demonstrates a two-dimensional grid that would be created for the sample data. Searching then involves finding the box that a point is in, then searching that box and all adjacent boxes. If the nearest neighbor has not been found, then the search is expanded to the next adjacent boxes. The search is continued until all required neighbors have been found.

One of the difficulties with this method is the determination of the appropriate box size. The sort is frequently tailored to the type of search that is required, since a box size is required and the preferred box size is dependent on the type of search to be done. However, one usually has only limited a priori knowledge of the searches that may be performed. Thus the appropriate box size for one search may not be appropriate for another. If the box size is too small, then many boxes are left unfilled and many boxes will need to be searched. This results in both excessive use of memory and excessive computation.

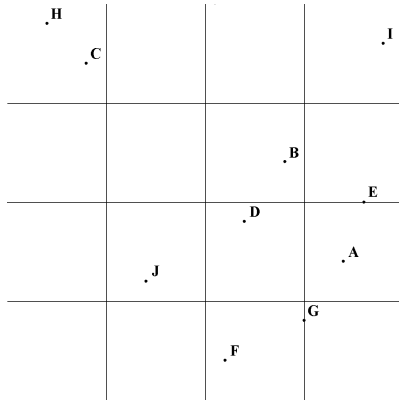


Figure 6- The sample data set as gridded into 16 boxes in two dimensions, using the box-assisted method.

The choice of box dimensionality may also be problematic. Schreiber, et al.[11] suggest 2 dimensional boxes. However, this may lead to inefficient searches for high dimensional data. Higher dimensional data may still be searched although many more boxes are often needed in order to find a nearest neighbor. On the other hand, using higher dimensional boxes will exacerbate the memory inefficiency. In the benchmarking section, we will consider both two and three-dimensional boxes.

6. BENCHMARKING AND COMPARISON OF METHODS

In this section we compare the suggested sorting and searching methods, namely the box assisted method, the KD-Tree, the K-tree, and the multidimensional quick-sort. All of these methods are preferable to a brute force search (where no sorting is done, and all data vectors are examined each time we do the searching). However, computational speed is not the only relevant factor. Complexity, memory use, and versatility of each method will also be discussed. The versatility of the method comes in two flavors- how well the method works on unusual data and how adaptable the method is to unusual searches. The multidimensional binary representation and the uniform K-Tree, described in the previous two sections, are not compared with the others because they are specialized sorts used only for exceptional circumstances.

6.1 Benchmarking of the KDTree

One benefit of the KD-Tree is its rough independence of search time on data set size. Figure 7 compares the average search time to find a nearest neighbor with the data set size. For large data set size, the search time has a roughly logarithmic dependence on the number of data points. This is due to the time it takes to determine the search point's location in the tree. If the search point were already in the tree, then the nearest neighbor search time is reduced from $O(\log n)$ to $O(1)$. This can be accomplished with the implementation of Bentley's suggested use of parent pointers for each node in the tree structure.^[10]

This is true even for higher dimensional data, although the convergence is much slower.

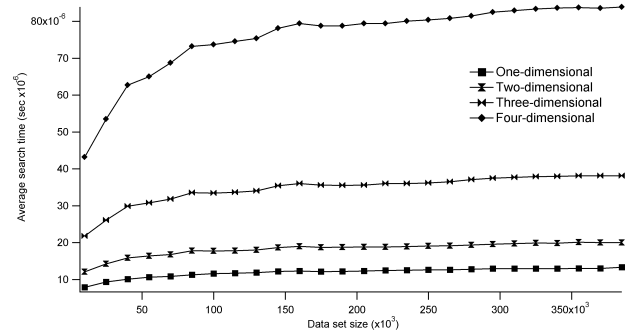


Figure 7- The dependence of average search time on data set size.

In Figure 8, the KD-Tree is shown to have an exponential dependence on the dimensionality of the data. This is an important result, not mentioned in other work providing diagnostic tests of the KD-Tree.^[10, 12] It implies that KD-Trees become inefficient for high dimensional data. It is not yet known what search method is most preferable for neighbor searching in a high dimension (greater than 8).

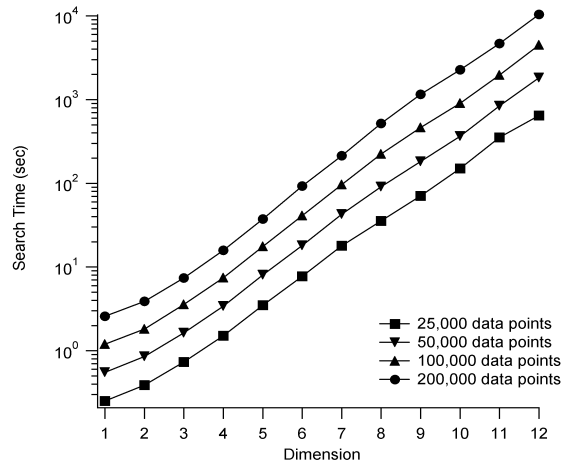


Figure 8- A log plot of search time vs dimension.

Figure 9 shows the relationship between the average search time to find n neighbors of a data point and the value n . In this plot, 10 data sets were generated with different seed values and search times were computed for each data set. The figure shows that the average search time is almost nearly linearly dependent on the number of neighbors n . Thus a variety of searches (weighted, radial, with or without exclusion) may be performed with only a linear loss in speed.

The drawbacks of the KD-Tree, while few, are transparent. First, if searching is to be done in many different dimensions, either a highly inefficient search is used, or additional search trees must be built. Also the method is somewhat memory intensive. In even the simplest KD-Tree, a number indicating the cutting value is required at each node, as well as an ordered array of data (similar to the quick-sort). If pointers to the parent node or principal cuts are used then the tree must contain even more information at each node. Although this increase may at

first seem unimportant, one should note that experimental data typically consists of 100000 floating points or more. In fact, some data analysis routines require a minimum of this many points. If the database is also on that order, then memory may prove unmanageable for many workstation computers.

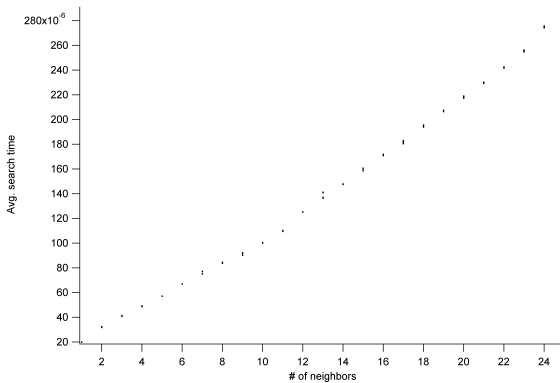


Figure 9- A plot of the average search time to find n neighbors of a data point, as a function of n .

We have implemented the KD-Tree as a dynamic linked library consisting of a set of fully functional object oriented routines. In short, they consist of the following functions

```

Create(*phTree, nCoords, nDims, nBucketSize,*aPoints);
FindNearestNeighbor(Tree,*pSearchPoint,*pFoundPoint);
FindMultipleNeighbors(Tree,*pSearchPoint,
*pnNeighbors,*aPoints);
FindRadialNeighbors(Tree,*pSearchPoint,radius,
**paPoints,*pnNeighbors);
ReleaseRadialNeighborList(*aPoints);
Release(Tree);

```

6.2 Comparison of methods

The KD-Tree implementation was tested in timed trials against the multidimensional quick-sort and the box-assisted method. In Figure 10 through Figure 13, we depict the dependence of search time on data set size for one through four dimensional data, respectively.

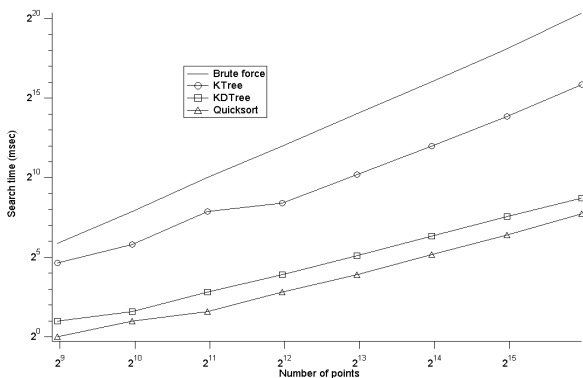


Figure 10- Comparison of search times for different methods using 1 dimensional random data.

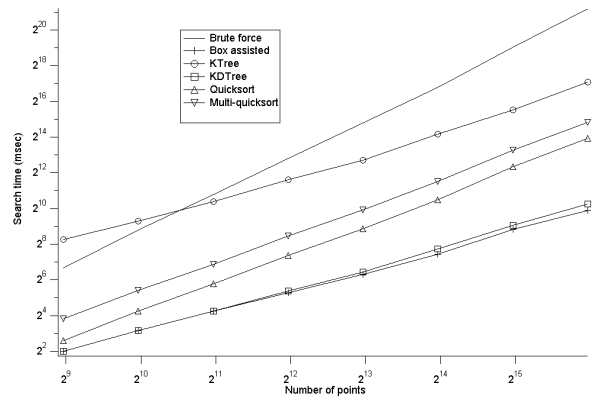


Figure 11- Comparison of search times for different methods using 2 dimensional random data.

In Figure 10, the multidimensional quick-sort reduces to a one-dimensional sort and the box assisted method as described by [12] is not feasible since it requires that the data be at least two-dimensional. We note from the slopes of these plots that the box-assisted method, the KDTree and the KTree all have an $O(n \log n)$ dependence on data set size, whereas the quick-sort based methods have approximately $O(n^{1.5})$ dependence on data set size for 2 dimensional data and $O(n^{1.8})$ dependence on data set size for 3 or 4 dimensional data. As expected, the brute force method has $O(n^2)$ dependence.

Despite its theoretical $O(n \log n)$ performance, the KTree still performs far worse than the box-assisted and KDTree methods. This is because of a large constant factor worse performance that is still significant for large data sets (64,000 points). This constant worse performance relates to the poor balancing of the KTree. Whereas for the KDTree, the data may be permuted so that cut values are always chosen at medians in the data, the KTree does not offer this option because there is no clear multidimensional median. In addition, many more branches in the tree may need to be searched in the KTree because at each cut, there are 2^k instead of 2 branches.

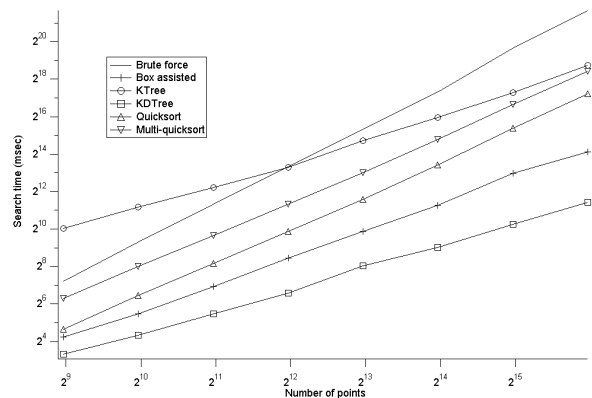


Figure 12- Comparison of search times for different methods using 3 dimensional random data.

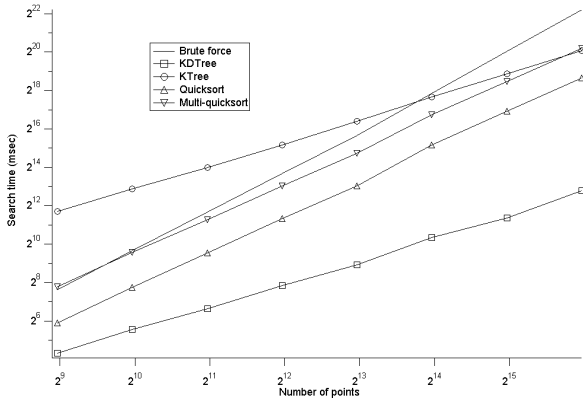


Figure 13- Comparison of search times for different methods using 4 dimensional random data.

However, all of the above trials were performed using uniform random noise. They say nothing of how these methods perform with other types of data. In order to compare the sorting and searching methods performance on other types of data, we compared their times for nearest neighbor searches on a variety of data sets. Table 1 depicts the estimated time in milliseconds to find all nearest neighbors in different 10,000 point data sets for each of the benchmarked search methods. The uniform noise data was similar to that discussed in the previous section.

Each Gaussian noise data set had a mean of 0 and standard deviation of 1 in each dimension. The identical dimensions and one valid dimension data sets were designed to test performance under unusual circumstances.

For the identical dimensions data, uniform random data was used and each coordinate of a vector was set equal, e.g.,

$$\overline{X}_i = (X_i^1, X_i^2, X_i^3) = (X_i^1, X_i^1, X_i^1).$$

For the data with only one valid dimension, uniform random data was used in only the first dimension, e.g.,

$$\overline{X}_i = (X_i^1, X_i^2, X_i^3) = (X_i^1, 0, 0).$$

In all cases the KD-Tree proved an effective method of sorting and searching the data. Only for the last two data sets did the multidimensional quick-sort method prove faster, and these data sets were constructed so that they were, in effect, one-dimensional. In addition, the box method proved particularly ineffective for high dimensional Gaussian data where the

dimensionality guaranteed that an excessive number of boxes needed to be searched, and for the Lorenz data, where the highly non-uniform distribution ensured that many boxes went unfilled. The K-tree also performed poorly for high dimensional data (four and five dimensional), due to the exponential increase in the number of searched boxes with respect to dimension.

A summary of the comparison of the four routines can be found in Table 2. The “adaptive” and “flexible” criteria refer to the next section.

Table 2- Comparison of some features of the four routines. Rating from 1=best to 4=worst.

Algorithm	Memory	Build	Search	Adapt.	Flexible
KDTree	2	3/4	1	yes	yes
KTree	3	3/4	3	no	no
Quick-sort	1	2	4	yes	yes
BoxAssisted	4	1	2	no	yes

7. INTERESTING PROPERTIES FOR MUSIC IR

The multi-dimensional search approach to Music IR, and the corresponding algorithms presented above have a number of interesting properties and conceptual advantages.

7.1 Adaptive to the distribution

A truly multi-dimensional approach enables an adaptation to the distribution of the data set. The KD-Tree algorithm, for example, focuses its discriminating power in a non-uniform way, to best fit the density of the data. This could be efficient for, say, search tasks in a database where part of the features remain quasi constant, e.g. a database of samples which are all pure tones of a given instrument, with quasi constant pitch, and a varying brightness. It is interesting to compare this adaptive behavior with a string-matching algorithm that would have to

Table 1- Nearest neighbor search times for data sets consisting of 10000 points. The brute force method, multidim. quick-sort, the box assisted method in 2 and 3 dimensions, the KDTree and the KTree were compared. An X indicates that it wasn't possible to use this search method on this type of data. The fastest method is given in bold and the second fastest method is given in italics.

Data set	Dimension	Brute	Quicksort	Box (2) method	Box (3) method	KDTree	KTree
Uniform noise	3	32567	2128	344	210	129	845
Gaussian	2	16795	280	623	X	56	581
Gaussian	4	44388	8114	54626	195401	408	3047
Identical dimensions	3	33010	19	1080	5405	42	405
One valid dimension	3	30261	31	1201	7033	37	453

compare sequences that all begin with “aaa...”. The latter can’t adapt and systematically tests the first three digits, which is an obvious waste of time.

7.2 Independent of the metric and of the alphabet

All the methods presented here are blind to the metric that is used. This is especially useful if the set of features is composite, and requires a different metric for each coordinate, e.g. pitches can be measured modulo 12. The routines are also independent of the alphabet, and work for integers as well as for floating-points. This makes them very general, as they can deal with a variety of queries on mixed low-level features and high-level meta-data such as:

Nearest neighbor (*pitch1, pitch2, pitch3, "BACH"*)

7.3 Flexibility

There are a variety of searches that are often performed on multidimensional data.^[7] Perhaps the most common type of search, and one of the simplest, is the nearest neighbor search. This search involves the identification of the nearest vector in the data set to some specified vector, known as the search vector. The search vector may or may not also be in the data set. Expansions on this type of search include the radial search, where one wishes to find all vectors within a given distance of the search vector, and the weighted search, where one wishes to find the nearest N vectors to the search vector.

Each of these searches (weighted, radial and nearest neighbor) may come with further restrictions. For instance, points or collections of points may be excluded from the search. Additional functionality may also be required. The returned data may be ordered from closest to furthest from the search vector, and the sorting and searching may be required to handle the insertion and deletion of points. That is, if points are deleted from or added to the data, these additional points should be added or deleted to the sort so that they can be removed or included in the search. Such a feature is essential if searching is to be performed with real-time analysis.

Most sorting and searching routine presented above are able to perform all the common types of searches, and are adaptable enough so that they may be made to perform any search.

7.4 A note on dimensionality

One of the restrictions shared by the multidimensional search routines presented on this paper is their dependence on the dimensionality of the data-set (not its size). This is detrimental to the sheer “folding” of the trajectory search as presented in the introduction, especially when it involves long M -sequences of high- N -dimension features (dimension $M*N$ may be too high). However, as we mentioned in the course of

this paper, there are still a variety of searches that can fit into the multidimensional framework. We notably wish to suggest:

- Searches for combinations of high-level metadata ($M=1$)
- It is possible to reduce N with classic dimensionality reduction techniques, such as Principal Component Analysis or Vector Quantization
- It is possible to reduce M by computing only 1 vector of features per audio piece. It is the approach taken in the Muscle Fish™ technology [13], where the mean, variance and correlation of the features are included in the feature vector.
- It is possible to reduce M by computing the features not on a frame-to-frame basis, but only when a significant change occurs (“event-based feature extraction”, see for example [14])
- For finite alphabets, it is always possible to reduce the dimension of a search by increasing the size of the alphabet. For example, searching for a set of 9 notes out of a 12 semi-tone alphabet can be reduced to a 3D search over an alphabet of 12^3 symbols.

8. CONCLUSION

We’ve presented and discussed four algorithms for a multidimensional approach to Music IR. The KD search tree is a highly adaptable, well-researched method for searching multidimensional data. As such it is very fast, but also can be memory intensive, and requires care in building the binary search tree. The k tree is a similar method, less versatile, more memory intensive, but easier to implement. The box-assisted method on the other hand, is used in a form designed for nonlinear time series analysis. It falls into many of the same traps that the other methods do. Finally the multidimensional quick-sort is an original method designed so that only one search tree is used regardless of how many dimensions are used.

These routines share a number of conceptual advantages over the approaches taken so far in the Music IR community, which -we believe- can be useful for a variety of musical searches. The aim of the paper is to be only a review, and the starting point of a reflection about search algorithms for music. In particular, we still have to implement specific music retrieval systems that use the results presented here.

9. REFERENCES

- [1] K. Lemstrom, String Matching Techniques for Music Retrieval. Report A-2000-4, University of Helsinki Press.
- [2] H. Samet, *Applications of Spatial Data Structures*: Addison-Wesley, 1989.
- [2] H. Samet, *The design and analysis of spatial data structures*: Addison-Wesley, 1989.
- [3] H. Hinterberger, K. C. Sevcik, and J. Nievergelt, *ACM Trans. On Database Systems*, vol. 9, pp. 38, 1984.
- [4] N. Pippenger, R. Fagin, J. Nievergelt, and H. R. Strong, *ACM Trans. On Database Systems*, vol. 4, pp. 315, 1979.

- [5] K. Mehlhorn, *Data Structures and Algorithms 3: Multidimensional Searching and Computational Geometry*: Springer-Verlag, 1984.
- [6] F. P. Preparata and M. I. Shamos, *Computational geometry: An introduction*. New York: Springer-Verlag, 1985.
- [7] J. Orenstein, *Information Processing Letters*, vol. 14, pp. 150, 1982.
- [8] J. H. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, vol. 18, pp. 509-517, 1975.
- [9] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Trans. Math. Software*, vol. 3, pp. 209, 1977.
- [10] J. L. Bentley, "K-d trees for semidynamic point sets," in *Sixth Annual ACM Symposium on Computational Geometry*, vol. 91. San Francisco, 1990.
- [11] T. Schreiber, "Efficient neighbor searching in nonlinear time series analysis," *Int. J. of Bifurcation and Chaos*, vol. 5, pp. 349-358, 1995.
- [12] R. F. Sproull, "Refinement to nearest-neighbour searching in k-d trees," *Algorithmica*, vol. 6, pp. 579-589, 1991.
- [13] E. Wold, T. Blum et al., "Content Based Classification, Search and Retrieval of Audio", in *IEEE Multimedia*, Vol.3, No. 3, Fall 1996, p.27-36.
- [14] F. Kurth, M. Clausen, "Full Text Indexing of Very Large Audio Databases", in *Proc. 110th AES Convention*, Amsterdam, May 2001.