OPTIMISED KD-TREE INDEXING OF MULTIMEDIA DATA

J. D. REISS[1], J. SELBIE[2] AND M. B. SANDLER[1]

[1]*Department of Electronic Engineering,*
Queen Mary University of London
Mile End Road, London, N52LL, United Kingdom
E-mail: josh.reiss,mark.sandler@elec.qmul.ac.uk

[2]*Microsoft Corp.*
One Microsoft Way,
Redmond, WA 98052, USA
E-mail: jselbie@microsoft.com

Near neighbor searching in image databases is a multidimensional problem. The kd-tree is one of the first methods proposed for indexing multidimensional data. We describe optimizations of this method, and determine when they are appropriate. We discuss adaptations of the tree to feature extraction and indexing problems in multimedia data. Results show increased functionality and speed using the kd-tree as the index structure on a multimedia database.

1. Introduction

For retrieval of multidimensional data, efficient indexing becomes essential. If no sorting is performed, then searching may require that each data vector be examined. To find the nearest neighbor of each point in a data set of N vectors requires the comparison of $N(N-1)/2$ distances when using a brute force method. Considerable work has been done in devising searching and sorting routines that can be run far more efficiently. In many areas of research the kd-tree[1, 2] has become accepted as one of the most efficient and versatile methods of searching.

Recent work has concentrated on multidimensional indexes that are stored in external memory, where the index construction time is of little importance.[3] For low dimensional data stored in main memory, the kd-tree remains one of the best indexing and neighbor searching methods available.[4] The kd-tree is also one of the simplest. Each internal node has two children, representing a partition along a given dimension of the n-dimensional hyperplane. The terminal nodes contain the n-dimensional records, which are typically features extracted from multimedia data. The choice of which dimension to choose to partition, and where to place the partition, is determined by the distribution of the data.

2. Multidimensional searches

A metric space is defined by the following four properties: for all *n*-dimensional vectors $\overline{x}, \overline{y}$ and $\overline{z}$ and integers *i* such that $1 \le i \le n$,

$$\text{Positivity: } d(\overline{x},\overline{y}) \ge 0 \quad (1)$$

$$\text{Definiteness: } d(\overline{x},\overline{y}) = 0 \Rightarrow \overline{x} = \overline{y} \quad (2)$$

$$\text{Symmetry: } d(\overline{x},\overline{y}) = d(\overline{y},\overline{x}) \quad (3)$$

$$\text{Triangle Inequality: } d(\overline{x},\overline{z}) \le d(\overline{x},\overline{y}) + d(\overline{y},\overline{z}) \quad (4)$$

Metrics are often defined through norms on a vector space, $d(\overline{x},\overline{y}) = \| \overline{x} - \overline{y} \|$. Most index structures should operate on the following, commonly used norms,

$$L_1(\overline{x}) = |x_1| + |x_2| \ldots + |x_n| \text{ (taxicab norm)}$$

$$L_2(\overline{x}) = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2} \text{ (Euclidean norm)}$$

$$L_\infty(\overline{x}) = \max(|x_1|, |x_1|, \ldots |x_n|) \text{ (Chebyshev norm)}$$

The kd-tree will correctly identify nearest neighbors for any metric space where $|x_i - y_i| \le d(\overline{x},\overline{y})$ [2]. However, the distance function *d* does not need to be a metric. That is, the triangle inequality is not a necessary condition on the distance measure. This implies that we may replace a metric *D* with a non-metric distance measure *d* as long as $d(\overline{x},\overline{y}) \le d(\overline{x},\overline{z}) \Rightarrow D(\overline{x},\overline{y}) \le D(\overline{x},\overline{z})$. Formally, a kd-tree can be used for indexing vectors provided that, for all *n*-dimensional vectors, $\overline{x}$, $\overline{y}$ and $\overline{z}$, a one dimensional distance d_i may be defined such that

$$x_i \le y_i \le z_i \Rightarrow d_i(x_i, y_i) \le d_i(x_i, z_i) \quad (5)$$

$$d_i(x_i, y_i) = d_i(y_i, x_i) \quad (6)$$

$$d(\overline{x},\overline{y}) = d(\overline{y},\overline{x}) \quad (7)$$

$$d_i(x_i, y_i) \le d(\overline{x},\overline{y}) \quad (8)$$

Eq.(5) guarantees that a kd-tree operates on a variety of distance measures, but it does not hold for finite commutative rings. Suppose a dimension measures hourly time stamps or the data is features from music files, and one feature represents position in the 12 tone chromatic scale. A realistic distance measure would be $d(x,y) = \min((x-y)_{\bmod n}, (y-x)_{\bmod n})$, so 1 o'clock is considered close to 12 o'clock, and on the chromatic scale, A is close to G Sharp. Although this will produce a metric on $\mathbb{Z}_n$, distance is no longer related to sequential ordering.

Eq. (8) is the justification for defining partitions along individual dimensions. The nature of the relationship between d_i and *d* is important in determining how to optimize a kd-tree. If $d_i(x_i, y_i) = |x_i - y_i|$, then Eq. (1) and Eq. (2) hold. This definition for the 1 dimensional distance was used in [2], but it is not necessary.

The kd-tree constraints hold for the entire family of *L* norms. They also hold when the one dimensional distances are not equivalent, i.e., $d_i(a,b) \ne d_i(b,a)$.

This is useful in situations where one may wish to perform multidimensional searches on a collection of features and give different weights to each feature.

In a traditional kd-tree nearest neighbour search, this fragment of pseudocode performs the recursive searching of internal nodes.

```
Search1(Node)
Dist=Distance1d(QueryPt[Node->CutDim],Node->CutValue);
if (QueryPt[Node->CutDim]<Node->CutValue) {
  Search(Node->Low);
  if (Dist<BestDist) Search(Node->High); }
else {
  Search(Node->High);
  if (Dist<BestDist) Search(Node->Low); }
```

Search checks all points in a leaf node, or calls Search1 again for internal nodes. However, this method may be searching unnecessarily many points. With the L_2 norm, all neighbors closer than a distance r are confined to an area of size pr^2 but we search an area of size $4r^2$. For the L_1 norm, we search an area of size $4r^2$, but the area in which nearer neighbors can exist is $2r^2$. This problem becomes exponentially worse with dimension, as seen in Table 1.

Table 1. Growth of unnecessarily searched space that is considered for searches under different dimensions and different norms.

Dimension (n)	**Volume Searched**	**Search Space**		***Unnecessary* Search Space**	
		L_1	L_2	L_1	L_2
2	$4r^2$	p r^2	$2r^2$	21.5%	50%
3	$8r^3$	4p $r^3/3$	$4r^3/3$	47.6%	83.3%
4	$16r^4$	$1p^2r^4/2$	$2r^4/3$	69.2%	95.8%
5	$32r^5$	$8p^2r^5/15$	$4r^5/15$	83.6%	99.2%

3. Bounding Search

As one compares successive partitions against the query, one compares against cut values further and further away from the search point, and along multiple dimensions. This is demonstrated in Figure 1. Assume one has descended the tree and found that a query point $\overline{q}$ resides in region A. In order to find neighbors of this point, one compares against points in that region. Then regions B, C and D are searched by searching the other sides of the partitions at Y_1, X_1 and Y_2 respectively. Yet points in region D are guaranteed to be separated by Y_2-q_2 in the second dimension *and* by X_1-q_1 in the first dimension.

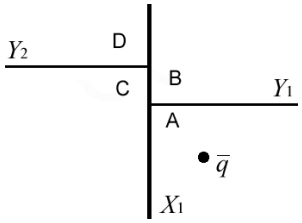

Figure 1.To find neighbors of a query $\bar{q}$, one first compares against points in A. Then B, C and D are examined by searching across partitions at Y_1, X_1 and Y_2, respectively.

It is more effective to consider the multidimensional distance from the closest corner of the hyperrectangle defined by all partitions that have been crossed. Thus we replace Search1 with the following search routine.

```
Search2(Node,Bounds)
NewBounds=Bounds;
NewBounds[Node->CutDim]=Node->CutValue;
NewMinDist=Distance(QueryPt,NewBounds);
if (QueryPt[Node->CutDim]<Node->CutValue) {
  Search(Node->Low,Bounds);
  if (NewMinDist<BestDist) Search(Node->High,NewBounds); }
else {
  Search(Node->High,Bounds);
  if (NewMinDist<BestDist) Search(Node->Low,NewBounds); }
```

The bounds array stores the locations of the closest corners of the bounding hyperrectangle to the query point. Initially the array is equal to the query point. For Figure 1, when region *B* is searched the bounds array becomes $(0,Y_1)$, then $(X_1,0)$ for region *C*, and (X_1, Y_2) for region *D*.

We allocate a new bounds array each time the routine is called. This adds computation at each node, and the number of nodes visited may be large so that large stacks are created. However, at most one element of the bounds array is changed at each node that is visited. So the bounds array may be made global and the stack only needs to incorporate changes to the array. Furthermore, the distance computation changed from 1-dimensional to *n*-dimensional. But many distance measures allow a simple computation of the distance between the query and the bounds if the previous distance and bounds are known. Assume the following holds when $\bar{y}$ and $\bar{z}$ are identical except in the i^{th} dimension

$$d_i(x_i,y_i) \le d_i(x_i,z_i) \Rightarrow d(\bar{x},\bar{z}) = F(d(\bar{x},\bar{y}), d_i(x_i,y_i), d_i(x_i,z_i)) \quad (9)$$

This is a form of weak separability between the dimensions. The bounds array now stores distances to the closest corners of the bounding hyperrectangle, and it is initially set to $(d(x_1,x_1), d(x_2,x_2), ..d(x_n,x_n))$.

In most cases, $d_i(x_i, y_i) \le d_i(x_i, z_i)$ is not necessary. L_∞ is an exception, since decreased distance along one dimension requires knowledge of all 1-dimensional distances to find the new distance. For increase in distance, Eq. (9) becomes a function on two variables, $d(\overline{x}, \overline{z}) = \max(d(\overline{x}, \overline{y}), d_i(x_i, z_i))$. If the distance measure is the square of the L_2 norm then F becomes $d(\overline{x}, \overline{y}) - (x_i - y_i)^2 + (x_i - z_i)^2$. This allows us to use a bounds array storing 1-dimensional distances and reduce the calculations at each node from n to 1.

```
Search3(Node,MinDist)
NewDist1d=SqDist1d(QueryPt[Node->CutDim],Node->CutValue);
NewMinDist=MinDist-Bounds[Node->CutDim]+NewDist1d;
if (QueryPt[Node->CutDim]<Node->CutValue) {
  Search(Node-> Low,MinDist);
  if (NewMinDist<BestDist)   {
     tmp=Bounds[Node->CutDim];
     Bounds[Node->CutDim]=NewDist1d;
     Search(Node-> High,NewMinDist);
     Bounds[Node->CutDim]=tmp;  }
}
else {
  Search (Node-> High,MinDist);
  if (NewMinDist<BestDist)   {
     tmp=Bounds[Node->CutDim];
     Bounds[Node->CutDim]=NewDist1d;
     Search (Node-> Low,NewMinDist);
     Bounds[Node->CutDim]=tmp;  } }
```

Figure 2(a) depicts the average number of Euclidean distance calculations required for a nearest neighbor search as a function of dimensionality, where an n-dimensional distance calculation is equivalent to n 1-dimensional calculations. For $n \le 6$, Search2 increases the number of distance calculations required. This is because the full n-dimensional distance is calculated at each node. For n>6 the reduced number of nodes visited compensates for the increased number of calculations at each node. However, Search3 offers a drastic reduction in the number of distance calculations required. Figure 2(b) more clearly demonstrates the improvements of a bounding search. Here we consider the average number of leaves visited. Search2 and Search3 provide the same results, since the difference between these methods only relates to how distances are calculated. Here, the bounding search is always an improvement over a traditional kd-tree search, and can lead to a five-fold reduction in the number of nodes visited.

Furthermore, the kd-tree improvements have been implemented on a large data set (275,465 60-dimensional vectors) consisting of features extracted from aerial images[5], and showed similar improvements. However, this also showed

the limits of the kd-tree. Using all 60 feature vectors caused the number of distance calculations to approach the $N(N\text{-}1)/2$ limit of a brute force search method. The kd-tree was only effective for less than 20 dimensional data, and proved less effective when smaller data sets of the same dimension were used.

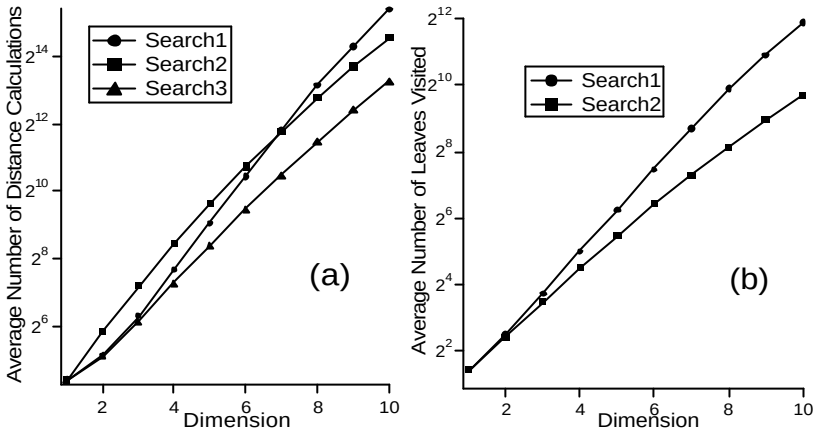

Figure 2. Average number of distance calculations and average number of leaves visited in a neighbor search as a function of dimension for 100,000 randomly distributed points.

4. Conclusion

It is important to notice that multidimensional indexing schemes often make assumptions regarding the distance measure that is used. The kd-tree constraints are surprisingly weak, since they do not require the triangle inequality to hold. This allows for alternative search methods that may give a significant (approximate factor of five) improvement in the speed of neighbor searching. This improvement is enough to justify the use of kd-trees on multimedia databases consisting of only moderately low dimensional features.

References

1. Bentley, J.H., *IEEE Trans. on Software Engineering*, 1979. **SE-5**: p. 333-340.
2. Bentley, J. *Sixth ACM Symposium on Comp. Geometry*. 1990. San Francisco.
3. Chakrabarti, K. and S. Mehrotra. *15th IEEE International Conference on Data Engineering (ICDE)*. 1999. Sydney, Australia.
4. Reiss, J.D., J.-J. Aucouturier, and M.B. Sandler. *2nd Annual International Symposium on Music Information Retrieval*. 2001. Bloomington, Indiana USA.
5. Manjunath, B.S. and W.Y. Ma, *IEEE Trans. Pattern Analysis and Machine Intelligence*, 1996. **18**(8): p. 837-842.