An intodoctory tutorial on kd-trees

Andrew W. Moore
Carnegie Mellon University
awm@cs.cmu.edu

Extract from Andrew Moore's PhD Thesis: *Efficient Memory-based Learning for Robot Control*
PhD. Thesis; Technical Report No. 209, Computer Laboratory, University of Cambridge. 1991.

Chapter 6

Kd-trees for Cheap Learning

This chapter gives a specification of the nearest neighbour algorithm. It also gives both an informal and formal introduction to the kd-tree data structure. Then there is an explicit, detailed account of how the nearest neighbour search algorithm is implemented efficiently, which is followed by an empirical investigation into the algorithm's performance. Finally, there is discussion of some other algorithms related to the nearest neighbour search.

6.1 Nearest Neighbour Specification

Given two multi-dimensional spaces **Domain** $= \Re^{k_d}$ and **Range** $= \Re^{k_r}$, let an *exemplar* be a member of **Domain** $\times$ **Range** and let an *exemplar-set* be a finite set of exemplars. Given an exemplar-set, $\mathbf{E}$, and a target domain vector, $\mathbf{d}$, then a *nearest neighbour* of $\mathbf{d}$ is any. exemplar $(\mathbf{d}', \mathbf{r}') \in \mathbf{E}$ such that *None-nearer*$(\mathbf{E}, \mathbf{d}, \mathbf{d}')$. Notice that there might be more than one suitable exemplar. This ambiguity captures the requirement that any nearest neighbour is adequate. *None-nearer* is defined thus:

$$\textit{None-nearer}(\mathbf{E}, \mathbf{d}, \mathbf{d}') \quad \Leftrightarrow \quad \forall (\mathbf{d}'', \mathbf{r}'') \in \mathbf{E} \quad |\, \mathbf{d} - \mathbf{d}' \,| \leq |\, \mathbf{d} - \mathbf{d}'' \,| \tag{6.1}$$

In Equation 6.1 the distance metric is Euclidean, though any other L_p-norm could have been used.

$$|\, \mathbf{d} - \mathbf{d}' \,| = \sqrt{\sum_{i=1}^{i=k_d} (d_i - d'_i)^2} \tag{6.2}$$

where d_i is the ith component of vector $\mathbf{d}$.

In the following sections I describe some algorithms to realize this abstract specification with the additional informal requirement that the computation time should be relatively short.

Algorithm:	Nearest Neighbour by Scanning.
Data Structures:	
domain-vector	A vector of k_d floating point numbers.
range-vector	A vector of k_r floating point numbers.
exemplar	A pair: (**domain-vector**, **range-vector**)
Input:	**exlist**, of type **list of exemplar**
	dom, of type **domain-vector**
Output:	**nearest**, of type **exemplar**
Preconditions:	**exlist** is not empty
Postconditions:	if **nearest** represents the exemplar $(\mathbf{d}', \mathbf{r}')$,
	and **exlist** represents the exemplar set $\mathbf{E}$,
	and **dom** represents the vector $\mathbf{d}$,
	then $(\mathbf{d}', \mathbf{r}') \in \mathbf{E}$ and *None-nearer*$(\mathbf{E}, \mathbf{d}, \mathbf{d}')$.
Code:	
1.	**nearest-dist** := infinity
2.	**nearest** := undefined
3.	for **ex** := each exemplar in **exlist**
3.1	**dist** := distance between **dom** and the domain of **ex**
3.2	if **dist** < **nearest-dist** then
3.2.1	**nearest-dist** := **dist**
3.2.2	**nearest** := **ex**

Table 6.1: Finding Nearest Neighbour by scanning a list.

6.2 Naive Nearest Neighbour

This operation could be achieved by representing the exemplar-set as a list of exemplars. In Table 6.1, I give the trivial nearest neighbour algorithm which scans the entire list. This algorithm has time complexity $O(N)$ where N is the size of $\mathbf{E}$. By structuring the exemplar-set more intelligently, it is possible to avoid making a distance computation for every member.

6.3 Introduction to kd-trees

A *kd-tree* is a data structure for storing a finite set of points from a k-dimensional space. It was examined in detail by J. Bentley [Bentley, 1980; Friedman *et al.*, 1977]. Recently, S. Omohundro has recommended it in a survey of possible techniques to increase the speed of neural network learning [Omohundro, 1987].

A *k*d-tree is a binary tree. The contents of each node are depicted in Table 6.2. Here I provide an informal description of the structure and meaning of the tree, and in the following subsection I

Field Name:	Field Type	Description
dom-elt	**domain-vector**	A point from k_d-d space
range-elt	**range-vector**	A point from k_r-d space
split	**integer**	The splitting dimension
left	**kd-tree**	A kd-tree representing those points to the left of the splitting plane
right	**kd-tree**	A kd-tree representing those points to the right of the splitting plane

Table 6.2: The fields of a kd-tree node

give a formal definition of the invariants and semantics.

The exemplar-set **E** is represented by the set of nodes in the kd-tree, each node representing one exemplar. The **dom-elt** field represents the domain-vector of the exemplar and the **range-elt** field represents the range-vector. The **dom-elt** component is the index for the node. It splits the space into two subspaces according to the splitting hyperplane of the node. All the points in the "left" subspace are represented by the **left** subtree, and the points in the "right" subspace by the **right** subtree. The splitting hyperplane is a plane which passes through **dom-elt** and which is perpendicular to the direction specified by the **split** field. Let i be the value of the **split** field. Then a point is to the left of **dom-elt** if and only if its ith component is less than the ith component of **dom-elt**. The complimentary definition holds for the right field. If a node has no children, then the splitting hyperplane is not required.

Figure 6.1 demonstrates a kd-tree representation of the four **dom-elt** points $(2,5)$, $(3,8)$, $(6,3)$ and $(8,9)$. The root node, with **dom-elt** $(2,5)$ splits the plane in the y-axis into two subspaces. The point $(3,8)$ lies in the lower subspace, that is $\{(x,y) \mid y < 5\}$, and so is in the left subtree. Figure 6.2 shows how the nodes partition the plane.

6.3.1 Formal Specification of a kd-tree

The reader who is content with the informal description above can omit this section. I define a mapping

$$\textit{exset-rep} : k\text{d-tree} \rightarrow \text{exemplar-set} \tag{6.3}$$

which maps the tree to the exemplar-set it represents:

$$\begin{aligned}
\textit{exset-rep}(\text{empty}) &= \phi \\
\textit{exset-rep}(<\mathbf{d}, \mathbf{r}, -, \text{empty}, \text{empty}>) &= \{(\mathbf{d}, \mathbf{r})\} \\
\textit{exset-rep}(<\mathbf{d}, \mathbf{r}, \text{split}, \text{tree}_{left}, \text{tree}_{right}>) &= \\
\quad \textit{exset-rep}(\text{tree}_{left}) \cup \{(\mathbf{d}, \mathbf{r})\} \cup \textit{exset-rep}(\text{tree}_{right}) &
\end{aligned} \tag{6.4}$$

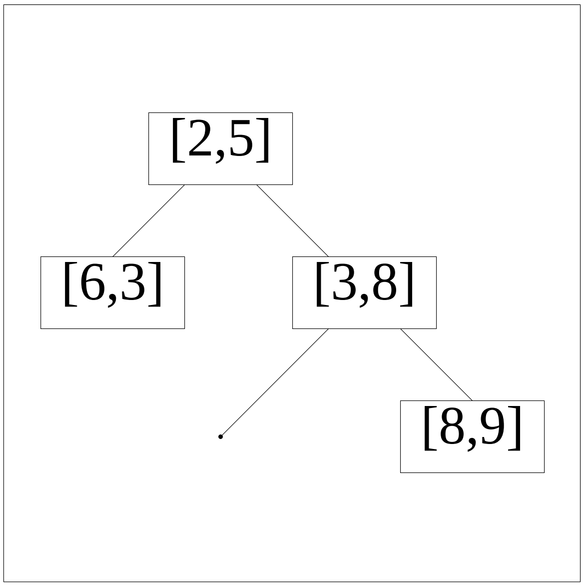

Figure 6.1

A 2d-tree of four elements. The splitting planes are not indicated. The [2,5] node splits along the y = 5 plane and the [3,8] node splits along the x = 3 plane.

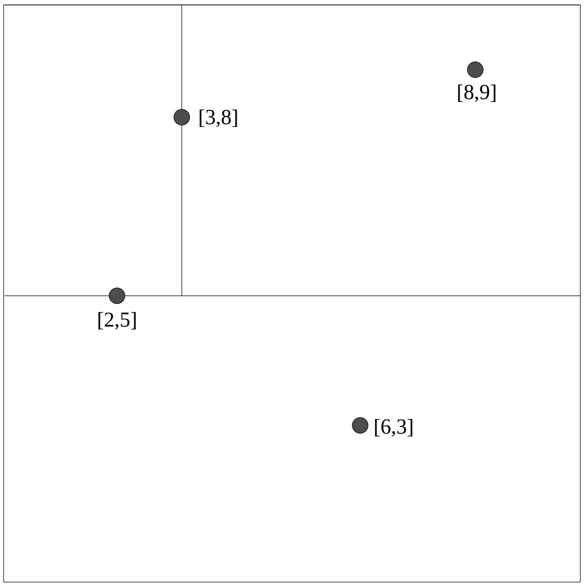

Figure 6.2

How the tree of Figure 6.1 splits up the x,y plane.

The invariant is that subtrees only ever contain **dom-elt**s which are on the correct side of all their ancestors' splitting planes.

$$
\begin{aligned}
&\textit{Is-legal-kdtree}(\text{empty}).\\
&\textit{Is-legal-kdtree}(<\mathbf{d},\mathbf{r},-,\text{empty},\text{empty}>).\\
&\textit{Is-legal-kdtree}(<\mathbf{d},\mathbf{r},\text{split},\text{tree}_{\textit{left}},\text{tree}_{\textit{right}}>) \quad \Leftrightarrow\\
&\qquad \forall(\mathbf{d}',\mathbf{r}') \in \textit{exset-rep}(\text{tree}_{\textit{left}}) \;\; d'_{\text{split}} \leq d_{\text{split}} \wedge\\
&\qquad \forall(\mathbf{d}',\mathbf{r}') \in \textit{exset-rep}(\text{tree}_{\textit{right}}) \;\; d'_{\text{split}} > d_{\text{split}} \wedge\\
&\qquad\qquad \textit{Is-legal-kdtree}(\text{tree}_{\textit{left}}) \wedge\\
&\qquad\qquad \textit{Is-legal-kdtree}(\text{tree}_{\textit{right}})
\end{aligned}
\tag{6.5}
$$

6.3.2 Constructing a kd-tree

Given an exemplar-set **E**, a *k*d-tree can be constructed by the algorithm in Table 6.3. The pivot-choosing procedure of Step 2 inspects the set and chooses a "good" domain vector from this set to use as the tree's root. The discussion of how such a root is chosen is deferred to Section 6.7. Whichever exemplar is chosen as root will not affect the correctness of the *k*d-tree, though the tree's maximum depth and the shape of the hyperregions *will* be affected.

6.4 Nearest Neighbour Search

In this section, I describe the nearest neighbour algorithm which operates on *k*d-trees. I begin with an informal description and worked example, and then give the precise algorithm.

A first approximation is initially found at the leaf node which contains the target point. In Figure 6.3 the target point is marked X and the leaf node of the region containing the target is coloured black. As is exemplified in this case, this first approximation is not necessarily the nearest neighbour, but at least we know any potential nearer neighbour must lie closer, and so must lie within the circle centred on X and passing through the leaf node. We now back up to the parent of the current node. In Figure 6.4 this parent is the black node. We compute whether it is *possible* for a closer solution to that so far found to exist in this parent's other child. Here it is not possible, because the circle does not intersect with the (shaded) space occupied by the parent's other child. If no closer neighbour can exist in the other child, the algorithm can immediately move up a further level, else it must recursively explore the other child. In this example, the next parent which is checked will need to be explored, because the area it covers (i.e. everywhere north of the central horizontal line) *does* intersect with the best circle so far.

Table 6.4 describes my actual implementation of the nearest neighbour algorithm. It is called with four parameters: the *k*d-tree, the target domain vector, a representation of a hyperrectangle in **Domain**, and a value indicating the maximum distance from the target which is worth searching. The search will only take place within those portions of the *k*d-tree which lie both in the hyper-

Algorithm:	Constructing a kd-tree
Input:	**exset**, of type **exemplar-set**
Output:	**kd**, of type **kdtree**
Pre:	None
Post:	**exset** = *exset-rep*(**kd**) $\wedge$ *Is-legal-kdtree*(**kd**)
Code:	
1.	If **exset** is empty then return the empty kdtree
2.	Call pivot-choosing procedure, which returns two values: **ex** := a member of **exset** **split** := the splitting dimension
3.	**d** := domain vector of **ex**
4.	**exset'** := **exset** with **ex** removed
5.	**r** := range vector of **ex**
6.	**exsetleft** := $\{(\mathbf{d}', \mathbf{r}') \in$ **exset'** $\mid d'_{\text{split}} \leq d_{\text{split}}\}$
7.	**exsetright** := $\{(\mathbf{d}', \mathbf{r}') \in$ **exset'** $\mid d'_{\text{split}} > d_{\text{split}}\}$
8.	**kdleft** := recursively construct kd-tree from **exsetleft**
9.	**kdright** := recursively construct kd-tree from **exsetright**
10.	**kd** := $< \mathbf{d}, \mathbf{r}, \mathbf{split}, \mathbf{kdleft}, \mathbf{kdright} >$
Proof:	By induction on the length of **exset** and the definitions of *exset-rep* and *Is-legal-kdtree*.

Table 6.3: Constructing a kd-tree from a set of exemplars.

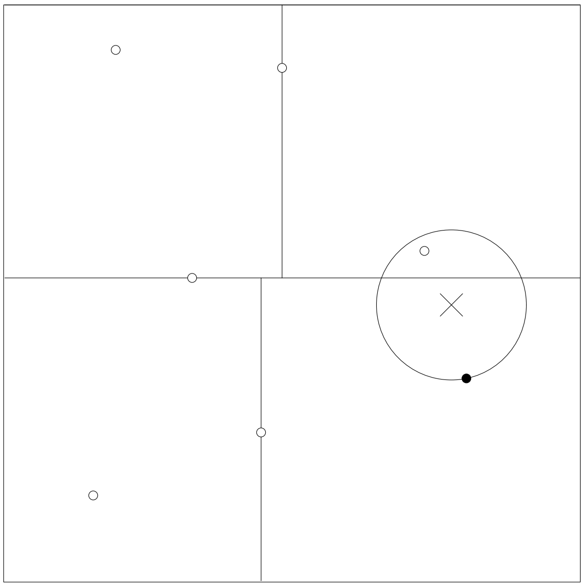

Figure 6.3

The black dot is the dot which owns the leaf node containing the target (the cross). Any nearer neighbour must lie inside this circle.

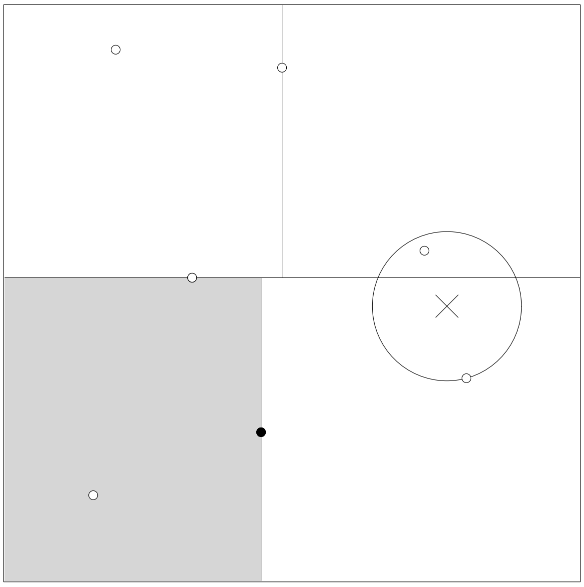

Figure 6.4

The black dot is the parent of the closest found so far. In this case the black dot's other child (shaded grey) need not be searched.

rectangle, and within the maximum distance to the target. The caller of the routine will generally specify the infinite hyperrectangle which covers the whole of **Domain**, and the infinite maximum distance.

Before discussing its execution, I will explain how the operations on the hyperrectangles can be implemented. A hyperrectangle is represented by two arrays: one of its minimum coordinates, the other of its maximum coordinates. To *cut* the hyperrectangle, so that one of its edges is moved closer to its centre, the appropriate array component is altered. To check to see if a hyperrectangle **hr** intersects with a hypersphere radius r centered at point $\mathbf{t}$, we find the point $\mathbf{p}$ in **hr** which is closest to $\mathbf{t}$. Write $hr_i^{\min}$ as the minimum extreme of **hr** in the ith dimension and $hr_i^{\max}$ as the maximum extreme. p_i, the ith component of this closest point is computed thus:

$$p_i = \begin{cases} hr_i^{\min} & \text{if } t_i \leq hr_i^{\min} \\ t_i & \text{if } hr_i^{\min} < t_i < hr_i^{\max} \\ hr_i^{\min} & \text{if } t_i \geq hr_i^{\max} \end{cases} \tag{6.6}$$

The objects intersect only if the distance between $\mathbf{p}$ and $\mathbf{t}$ is less than or equal to r.

The search is depth first, and uses the heuristic of searching first the child node which contains the target. Step 1 deals with the trivial empty tree case, and Steps 2 and 3 assign two important local variables. Step 4 cuts the current hyperrectangle into the two hyperrectangles covering the space occupied by the child nodes. Steps 5–7 determine which child contains the target. After Step 8, when this initial child is searched, it may be possible to prove that there cannot be any closer point in the hyperrectangle of the further child. In particular, the point at the current node must be out of range. The test is made in Steps 9 and 10. Step 9 restricts the maximum radius in which any possible closer point could lie, and then the test in Step 10 checks whether there is any

Algorithm:	Nearest Neighbour in a *k*d-tree
Input:	**kd**, of type **kdtree** **target**, of type **domain vector** **hr**, of type **hyperrectangle** **max-dist-sqd**, of type **float**
Output:	**nearest**, of type **exemplar** **dist-sqd**, of type **float**
Pre:	*Is-legal-kdtree*(**kd**)
Post:	Informally, the postcondition is that **nearest** is a nearest exemplar to **target** which also lies both within the hyperrectangle **hr** and within distance $\sqrt{\textbf{max-dist-sqd}}$ of **target**. $\sqrt{\textbf{dist-sqd}}$ is the distance of this nearest point. If there is no such point then **dist-sqd** contains infinity.
Code:	
1.	if **kd** is empty then set **dist-sqd** to infinity and exit.
2.	**s** := **split** field of **kd**
3.	**pivot** := **dom-elt** field of **kd**
4.	Cut **hr** into two sub-hyperrectangles **left-hr** and **right-hr**. The cut plane is through **pivot** and perpendicular to the **s** dimension.
5.	**target-in-left** := $\textbf{target}_\textbf{s} \leq \textbf{pivot}_\textbf{s}$
6.	if **target-in-left** then
6.1	**nearer-kd** := **left** field of **kd** and **nearer-hr** := **left-hr**
6.2	**further-kd** := **right** field of **kd** and **further-hr** := **right-hr**
7.	if **not target-in-left** then
7.1	**nearer-kd** := **right** field of **kd** and **nearer-hr** := **right-hr**
7.2	**further-kd** := **left** field of **kd** and **further-hr** := **left-hr**
8.	Recursively call **Nearest Neighbour** with parameters (**nearer-kd**,**target**, **nearer-hr**,**max-dist-sqd**), storing the results in **nearest** and **dist-sqd**
9.	**max-dist-sqd** := minimum of **max-dist-sqd** and **dist-sqd**
10.	A nearer point could only lie in **further-kd** if there were some part of **further-hr** within distance $\sqrt{\textbf{max-dist-sqd}}$ of **target**. if this is the case then
10.1	if $(\textbf{pivot} - \textbf{target})^2 < \textbf{dist-sqd}$ then
10.1.1	**nearest** := (**pivot**, **range-elt** field of **kd**)
10.1.2	**dist-sqd** := $(\textbf{pivot} - \textbf{target})^2$
10.1.3	**max-dist-sqd** := **dist-sqd**

10.2	Recursively call **Nearest Neighbour** with parameters
	    (**further-kd**,**target**, **further-hr**,**max-dist-sqd**),
	    storing the results in **temp-nearest** and **temp-dist-sqd**

10.3	If **temp-dist-sqd** < **dist-sqd** then
10.3.1	**nearest** := **temp-nearest** and **dist-sqd** := **temp-dist-sqd**
Proof:	Outlined in text

Table 6.4: The Nearest Neighbour Algorithm

space in the hyperrectangle of the further child which lies within this radius. If it is *not* possible then no further search is necessary. If it *is* possible, then Step 10.1 checks if the point associated with the current node of the tree is closer than the closest yet. Then, in Step 10.2, the further child is recursively searched. The maximum distance worth examining in this further search is the distance to the closest point yet discovered.

The proof that this will find the nearest neighbour within the constraints is by induction on the size of the kd-tree. If the cutoff were not made in Step 10, then the proof would be straightforward: the point returned is the closest out of (i) the closest point in the nearer child, (ii) the point at the current node and (iii) the closest point in the further child. If the cutoff *were* made in Step 10, then the point returned is the closest point in the nearest child, and we can show that neither the current point, nor any point in the further child can possibly be closer.

Many local optimizations are possible which while not altering the asymptotic performance of the algorithm will multiply the speed by a constant factor. In particular, it is in practice possible to hold almost all of the search state globally, instead of passing it as recursive parameters.

6.5 Theoretical Behaviour

Given a kd-tree with N nodes, how many nodes need to be inspected in order to find the proven nearest neighbour using the algorithm in Section 6.4?. It is clear at once that on average, at least $O(\log N)$ inspections are necessary, because any nearest neighbour search requires traversal to at least one leaf of the tree. It is also clear that no more than N nodes are searched: the algorithm visits each node at most once.

Figure 6.5 graphically shows why we might expect considerably fewer than N nodes to be visited: the shaded areas correspond to areas of the kd-tree which were cut off.

The important values are (i) the worst case number of inspections and (ii) the expected number of inspections. It is actually easy to construct worst case distributions of the points which will force nearly all the nodes to be inspected. In Figure 6.6, the tree is two-dimensional, and the points are scattered along the circumference of a circle. If we request the nearest neighbour with the target close to the centre of the circle, it will therefore be necessary for each rectangle, and hence each leaf, to be inspected (this is in order to ensure that there is no point lying inside the circle in any rectangle).

Calculation of the expected number of inspections is difficult, because the analysis depends critically on the expected distribution of the points in the kd-tree, and the expected distribution of the target points presented to the nearest neighbour algorithm.

The analysis is performed in [Friedman *et al.*, 1977]. This paper considers the expected number of hyperrectangles corresponding to leaf nodes which will provably need to be searched. Such hyperrectangles intersect the volume enclosed by a hypersphere centered on the query point whose surface passes through the nearest neighbour. For example, in Figure 6.5 the hypersphere (in this

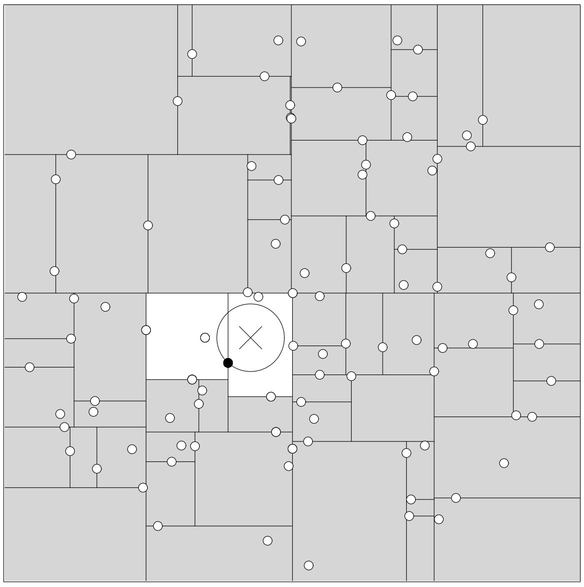

Figure 6.5

Generally during a nearest neighbour search only a few leaf nodes need to be inspected.

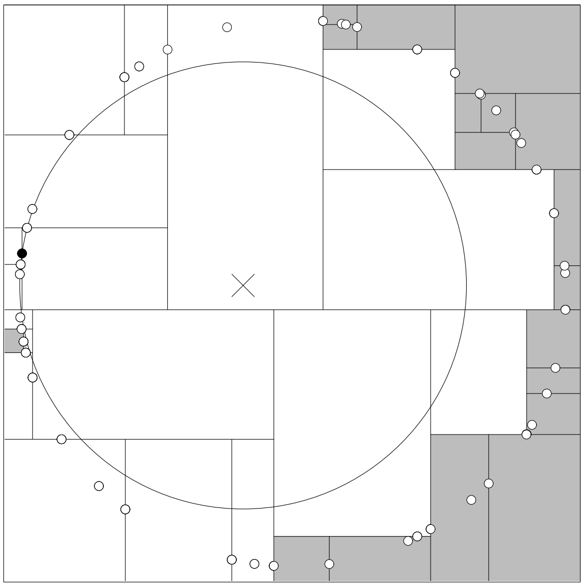

Figure 6.6

A bad distribution which forces almost all nodes to be inspected.

case a circle) is shown, and the number of intersecting hyperrectangles is two.

The paper shows that the expected number of intersecting hyperrectangles is independent of N, the number of exemplars. The asymptotic search time is thus logarithmic because the time to descend from the root of the tree to the leaves is logarithmic (in a balanced tree), and then an expected constant amount of backtracking is required.

However, this reasoning was based on the assumption that the hyperrectangles in the tree tend to be hypercubic in shape. Empirical evidence in my investigations has shown that this is not generally the case for their tree building strategy. This is discussed and demonstrated in Section 6.7.

A second danger is that the cost, while independent of N, is exponentially dependent on k, the dimensionality of the domain vectors[1].

Thus theoretical analysis provides some insight into the cost, but here, empirical investigation will be used to examine the expense of nearest neighbour in practice.

6.6 Empirical Behaviour

In this section I investigate the empirical behaviour of the nearest neighbour searching algorithm. We expect that the number of nodes inspected in the tree varies according to the following properties of the tree:

- N, the size of the tree.
- k_{dom} , the dimensionality of the domain vectors in the tree. This value is the k in kd-tree.
- d_{distrib} , the distribution of the domain vectors. This can be quantified as the "true" dimensionality of the vectors. For example, if the vectors had three components, but all lay on the surface of a sphere, then the underlying dimensionality would be 2. In general, discovery of the underlying dimensionality of a given sample of points is extremely difficult, but for these tests it is a straightforward matter to *generate* such points. To make a kd-tree with underlying dimensionality d_{distrib}, I use randomly generated k_{dom}-dimensional domain vectors which lie on a d_{distrib}-dimensional hyperelliptical surface. The random vector generation algorithm is as follows: Generate d_{distrib} random angles $\theta_i \in [0, 2\pi)$ where $0 \leq i < d_{\text{distrib}}$. Then let the jth component of the vector be $\prod_{i=0}^{i=d-1} \sin(\theta_i + \phi_{ij})$. The phase angles ϕ_{ij} are defined as $\phi_{ij} = \frac{1}{2}\pi$ if the jth bit of the binary representation of i is 1 and is zero otherwise.
- d_{target} , the probability distribution from which the search target vector will be selected. I shall assume that this distribution is the same as that which determines the domain vectors. This is indeed what will happen when the kd-tree is used for learning control.

[1]This was pointed out to the author by N. Maclaren.

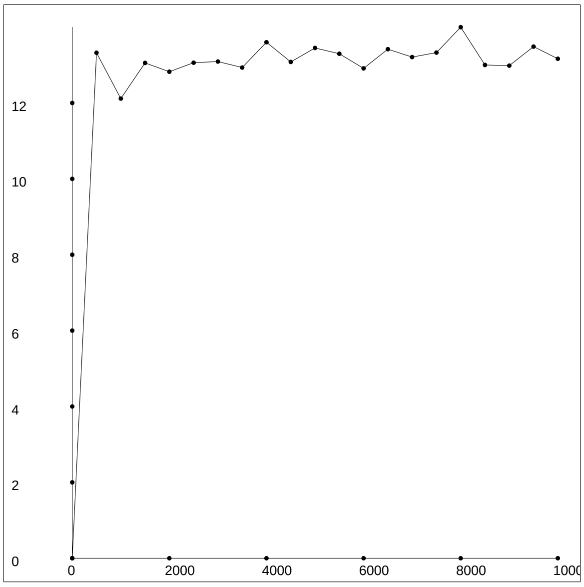

Figure 6.7

Number of inspections required during a nearest neighbour search against the size of the kd-tree. In this experiment the tree was four-dimensional and the underlying distribution of the points was three-dimensional.

In the following sections I investigate how performance depends on each of these properties.

6.6.1 Performance against Tree Size

Figures 6.7 and 6.8 graph the number of nodes inspected against the number of nodes in the entire *k*d-tree. Each value was derived by generating a random *k*d-tree, and then requesting 500 random nearest neighbour searches on the *k*d-tree. The average number of inspected nodes was recorded. A node was counted as being inspected if the distance between it and the target was computed. Figure 6.7 was obtained from a 4d-tree with an distribution distribution $d_{\text{distrib}} = 3$. Figure 6.8 used an 8d-tree with an underlying distribution $d_{\text{distrib}} = 8$.

It is immediately clear that after a certain point, the expense of a nearest neighbour search has no detectable increase with the size of the tree. This agrees with the proposed model of search cost—logarithmic with a large additive constant term.

6.6.2 Performance against the "k" in kd-tree

Figure 6.9 graphs the number of nodes inspected against k_{dom}, the number of components in the *k*d-tree's domain vectors for a 10,000 node tree. The underlying dimensionality was also k_{dom}. The number of inspections per search rises very quickly, possibly exponentially, with k_{dom}. This behaviour, the massive increase in cost with dimension, is familiar in computational geometry.

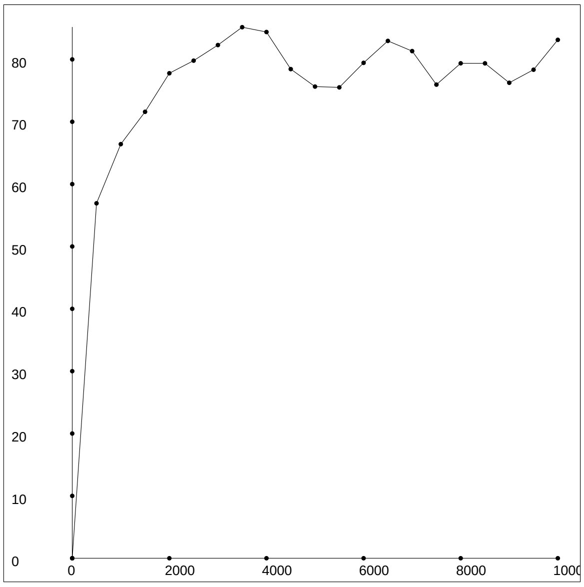

Figure 6.8

Number of inspections against kd-tree size for an eight-dimensional tree with an eight-dimensional underlying distribution.

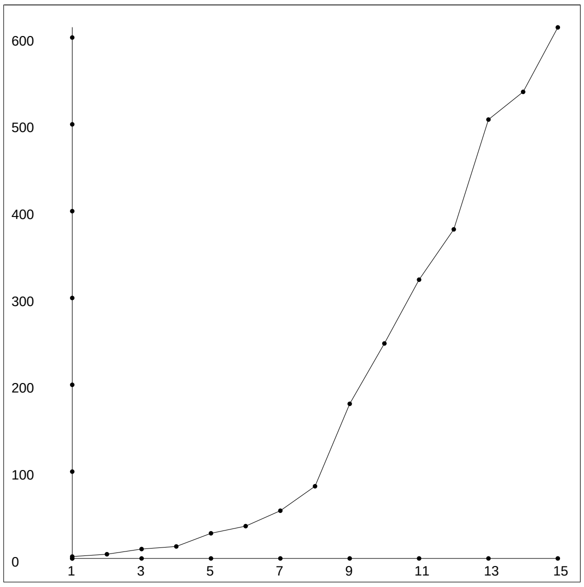

Figure 6.9

Number of inspections graphed against tree dimension. In these experiments the points had an underlying distribution with the same dimensionality as the tree.

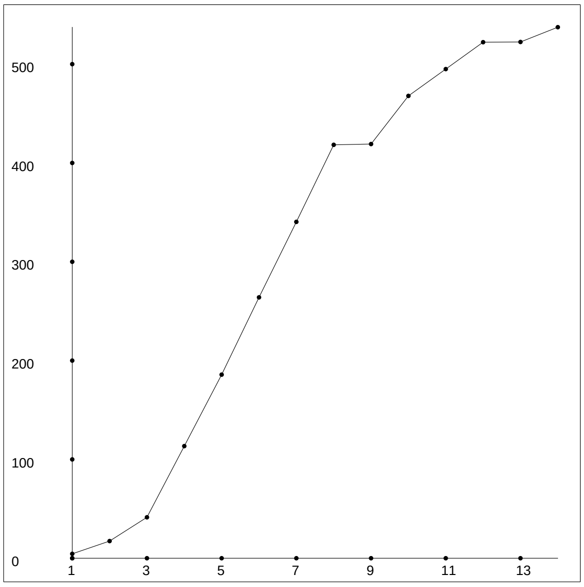

Figure 6.10

Number of inspections graphed against underlying dimensionality for a fourteen-dimensional tree.

6.6.3 Performance against the Distribution Dimensionality

This experiment confirms that it is d_{distrib}, the distribution dimension from which the points were selected, rather than k_{dom} which critically affects the search performance. The trials for Figure 6.10 used a 10,000 node kd-tree with domain dimension of 14, for various values of d_{distrib}. The important observation is that for 14d-trees, the performance does improve greatly if the underlying distribution-dimension is relatively low. Conversely, Figure 6.11 shows that for a fixed (4-d) underlying dimensionality, the search expense does not seem to increase any worse than linearly with k_{dom}.

6.6.4 When the Target is not Chosen from the kd-tree's Distribution

In this experiment the points were distributed on a three dimensional elliptical surface in ten-dimensional space. The target vector was, however, chosen at random from a ten-dimensional distribution. The kd-tree contained 10,000 points. The average number of inspections over 50 searches was found to be 8,396. This compares with another experiment in which both points and target were distributed in ten dimensions and the average number of inspections was only 248. The reason for the appalling performance was exemplified in Figure 6.6: if the target is very far from its nearest neighbour then very many leaf nodes must be checked.

6.6.5 Conclusion

The speed of the search (measured as the number of distance computations required) seems to vary . . .

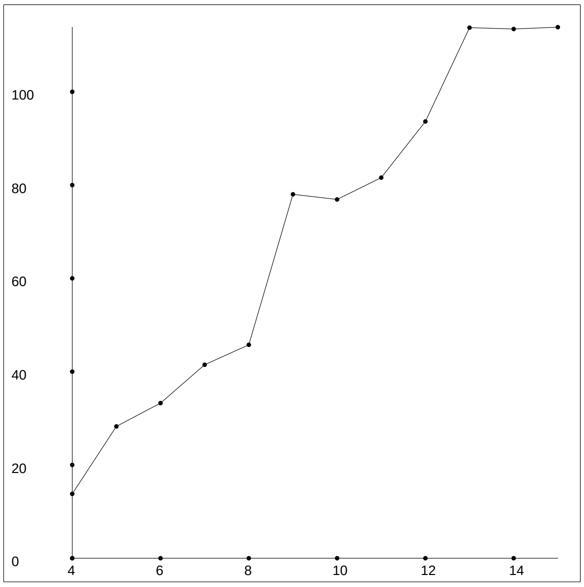

Figure 6.11

Number of inspections graphed against tree dimension, given a constant four dimensional underlying distribution.

- ...only marginally with tree size. If the tree is sufficiently large with respect to the number of dimensions, it is essentially constant.
- ...very quickly with the dimensionality of the distribution of the datapoints, d_{distrib}.
- ...linearly with the number of components in the kd-tree's domain (k_{dom}), given a fixed distribution dimension (d_{distrib}).

There is also evidence to suggest that unless the target vector is drawn from the same distribution as the kd-tree points, performance can be greatly worsened.

These results support the belief that real time searching for nearest neighbours is practical in a robotic system where we can expect the underlying dimensionality of the data points to be low, roughly less than 10. This need not mean that the vectors in the input space should have less than ten components. For data points obtained from robotic systems it will not be easy to decide what the underlying dimensionality is. However Chapter 10 will show that the data does tend to lie within a number of low dimensional subspaces.

6.7 Further kd-tree Operations

In this section I discuss some other operations on kd-trees which are required for use in the *SAB* learning system. These include incrementally adding a point to a kd-tree, range searching, and selecting a pivot point.

6.7.1 Range Searching a kd-tree

$$\textit{range-search} \quad : \quad \text{exemplar-set} \times \mathbf{Domain} \times \Re \rightarrow \text{exemplar-set}$$

The abstract range search operation on an exemplar-set finds all exemplars whose domain vectors are within a given distance of a target point:

$$\textit{range-search}(\mathbf{E}, \mathbf{d}, r) \quad = \quad \{(\mathbf{d}', \mathbf{r}') \in \mathbf{E} \mid (\mathbf{d} - \mathbf{d}')^2 < r^2\}$$

This is implemented by a modified nearest neighbour search. The modifications are that (i) the initial distance is not reduced as closer points are discovered and (ii) all discovered points within the distance are returned, not just the nearest. The complexity of this operation is shown, in [Preparata and Shamos, 1985], to still be logarithmic in N (the size of $\mathbf{E}$) for a fixed range size.

6.7.2 Choosing a Pivot from an Exemplar Set

The tree building algorithm of Section 6.3 requires that a pivot and a splitting plane be selected from which to build the root of a *k*d-tree. It is desirable for the tree to be reasonably balanced, and also for the shapes of the hyperregions corresponding to leaf nodes to be fairly equally proportioned. The first criterion is important because a badly unbalanced tree would perhaps have $O(N)$ accessing behaviour instead of $O(\log N)$. The second criterion is in order to maximize cutoff opportunities for the nearest neighbour search. This is difficult to formalize, but can be motivated by an illustration. In Figure 6.12 is a perfectly balanced *k*d-tree in which the leaf regions are very non-square. Figure 6.13 illustrates a *k*d-tree representing the same set of points, but which promotes squareness at the expense of some balance.

One pivoting strategy which would lead to a perfectly balanced tree, and which is suggested in [Omohundro, 1987], is to pick the splitting dimension as that with maximum variance, and let the pivot be the point with the median split component. This will, it is hoped, tend to promote square regions because having split in one dimension, the next level in the tree is unlikely to find that the same dimension has maximum spread, and so will choose a different dimension. For uniform distributions this tends to perform reasonably well, but for badly skewed distributions the hyperregions tend to take long thin shapes. This is exemplified in Figure 6.12 which has been balanced using this standard median pivot choice.

To avoid this bad case, I choose a pivot which splits the exemplar set in the middle of the range of the most spread dimension. As can be seen in Figure 6.13, this tends to favour squarer regions at the expense of a slight imbalance in the *k*d-tree. This means that large empty areas of space are filled with only a few hyperrectangles which are themselves large. Thus, the number of leaf nodes which need to be inspected in case they contain a nearer neighbour is smaller than for the original case, which had many small thin hyperrectangles.

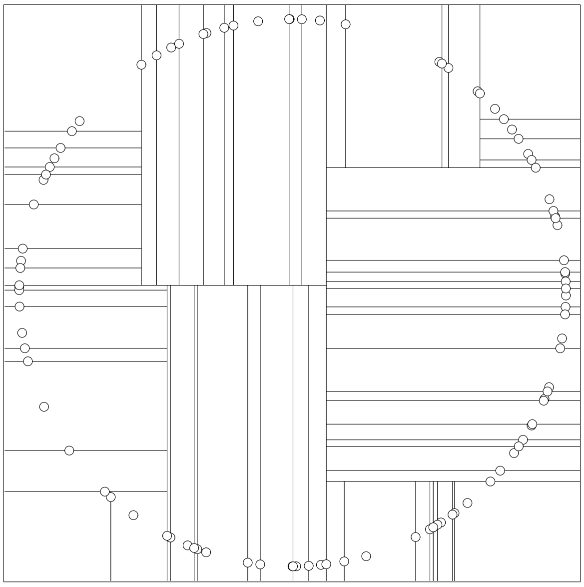

Figure 6.12

A 2d tree balanced using the 'median of the most spread dimension' pivoting strategy.

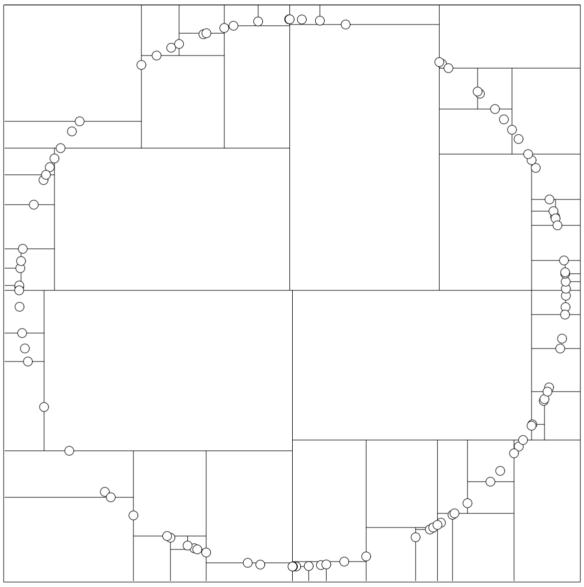

Figure 6.13

A 2d tree balanced using the 'closest to the centre of the widest dimension' pivoting strategy.

My pivot choice algorithm is to firstly choose the splitting dimension as the longest dimension of the current hyperrectangle, and then choose the pivot as the point closest to the middle of the hyperrectangle along this dimension. Occasionally, this pivot may even be an extreme point along its dimension, leading to an entirely unbalanced node. This is worth it, because it creates a large empty leaf node. It is possible but extremely unlikely that the points could be distributed in such a way as to cause the tree to have one empty child at every level. This would be unacceptable, and so above a certain depth threshold, the pivots are chosen using the standard median technique.

Selecting the median as the split and selecting the closest to the centre of the range are both $O(N)$ operations, and so either way a tree rebalance is $O(N \log N)$.

6.7.3 Incrementally Adding a Point to a kd-tree

Firstly, the leaf node which contains the new point is computed. The hyperrectangle corresponding to this leaf is also obtained. See Section 6.4 for hyperrectangle implementation. When the leaf node is found it may either be (i) empty, in which case it is simply replaced by a new singleton node, or (ii) it contains a singleton node. In case (ii) the singleton node must be given a child, and so its previously irrelevant split field must be defined. The split field should be chosen to preserve the squareness of the new subhyperrectangles. A simple heuristic is used. The split dimension is chosen as the dimension in which the hyperrectangle is longest. This heuristic is motivated by the same requirement as for tree balancing—that the regions should be as square as possible, even if this means some loss of balance.

This splitting choice is just a heuristic, and there is no guarantee that a series of points added in this way will preserve the balance of the *k*d-tree, nor that the hyperrectangles will be well shaped for nearest neighbour search. Thus, on occasion (such as when the depth exceeds a small multiple of the best possible depth) the tree is rebuilt. Incremental addition costs $O(\log N)$.

6.7.4 Q Nearest Neighbours

This uses a modified version of the nearest neighbour search. Instead of only searching within a sphere whose radius is the closest distance yet found, the search is within a sphere whose radius is the Qth closest yet found. Until Q points have been found, this distance is infinity.

6.7.5 Deleting a Point from a kd-tree

If the point is at a leaf, this is straightforward. Otherwise, it is difficult, because the structure of both trees below this node are pivoted around the point we wish to remove. One solution would be to rebuild the tree below the deleted point, but on occasion this would be very expensive. My solution is to mark the point as deleted with an extra field in the *k*d-tree node, and to ignore deletion nodes in nearest neighbour and similar searches. When the tree is next rebuilt, all deletion nodes are removed.

Bibliography

[Bentley, 1980] J. L. Bentley. Multidimensional Divide and Conquer. *Communications of the ACM*, 23(4):214—229, 1980.

[Friedman *et al.*, 1977] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. on Mathematical Software*, 3(3):209–226, September 1977.

[Omohundro, 1987] S. M. Omohundro. Efficient Algorithms with Neural Network Behaviour. *Journal of Complex Systems*, 1(2):273–347, 1987.

[Preparata and Shamos, 1985] F. P. Preparata and M. Shamos. *Computational Geometry.* Springer-Verlag, 1985.