Nearest Neighbours using kd-Trees

Steve Margetts

December 3, 1996

1 Introduction

The aim of this document is to explain the algorithm outlined in "An Algorithm for Finding Best Matches in Logarithmic Expected Time"[2] when applied to finding nearest neighbours to a set of points distributed in a k-dimensional space. *Mathematica* code will be used to illustrate the algorithm.

The algorithm works in two phases. Firstly, a binary tree (termed "kd-tree") is constructed from the data set to be searched. The tree is then used in the second phase to extract the nearest neighbours of each point. The basically premise is that to find the nearest neighbours of some query point it makes sense to search for them in increasing distance from that point. By keeping track of the region of the search space it is currently searching in, candidate points can be excluded from the search when it is known that all the points in a particular region are further away than the nearest neighbours found so far.

2 Building the kd-Tree

The first step is to build the kd-tree. It will be assumed that the points are to be referenced by an identifier (such as an array index), and that the output for each query point should be a list of point identifiers with their distances from the query point.

As Mathematica is the chosen implementation language, the input data will form a list of k-dimensional tuples, where the position of the point in the input list will be its identifier. It is necessary to preserve this index as the order of the points will be disrupted by the tree building process, so the first task is to create a new list from the data where each point is tagged with its index number. The following procedure implements this:

```
addIndex[data_] := Table[{data[[i]], i}, {i,1,Length[data]}];
```

As accessing the data is now somewhat convoluted, it is nice to define a few data-access routines. The following are used to extract a value from a vector, the index or the entire vector from a list of data points tagged with their identifiers:

```
getValueFor[in_, i_, d_] := in[[i]][[1]][[d]];
getIndex[in_, i_] := in[[i]][[2]];
getPoint[in_, i_] := in[[i]][[1]];
```

The next step is to recursively build the tree. The data points are to be distributed at the leaves of the tree. Each leaf node, or "bucket", can hold upto a set number of records, termed the "bucket size". It is this value which determines whether or not the procedure is to be called recursively.

If the number of data points being processed in this instance of the procedure is less than the bucket size, the data set is simply returned. If larger, then the dimension with the largest spread is found by searching through all the values of each dimension for each record and recording the maximum and minimum values. The median of the values for the largest spread dimension is then extracted.

The reason for this is that at each non-terminal node, it is desired to split the remaining data points as evenly as possible. By picking the dimension with the largest spread, and then taking the median value of that dimension, it is hoped that after partitioning the data around the median value, the data will be evenly distributed. (This need not be the case in point distributions with one or two outlying values - the interested reader should consult the paper by D.A White[1], where they advocate splitting the data on the dimension with the largest variance). It is this fair split which allows the tree to be constructed in time proportional to $O(nlogn)$.

It only remains to define the format of a kd-tree. At each non-terminal node, the split dimension number and the partition value (the median) must be recorded, together with the left and right branch. At terminal or leaf nodes, the set of data points must be recorded. This suggests the following structure:

kd-tree ::= `{` *dimension number*, *median value*, **kd-tree**, **kd-tree** `}`
| `[{` *vector*, *identifier* `}, ...]`

where the two **kd-tree** references in the definition are the left and right branches of the tree. The Mathematica code to build such a tree is below. The module requires two parameters: *inData* is the list of data, and k is the dimensionality of the data (eg. set $k = 2$ for (x, y) pairs).

```
buildTree[k_, inData_] :=
        Module[
        (* "b" is the number of records per bucket *)
        {bucketsize = 1,  maxspread = 0, i, dimension,
         hi, low, thisHi, thisLow, medianval, pair, temp},

        If[Length[inData] <= bucketsize,
                (* This is a terminal node, so simply return the
                 * list as it was passed *)
                Return[inData],
                (* Find the dimension with the largest spread *)
                For[i = 1, i <= k, i++,
                        hi = -Infinity; low = Infinity;
                        For [j = 1, j <= Length[inData], j++,
                                temp = getValueFor[inData, j, i];
                                If[hi < temp, hi = temp, ];
                                If[low > temp, low = temp, ]
                        ];
                        If[hi - low >= maxspread,
                                maxspread = hi - low;
                                thisHi = hi; thisLow = low;
                                dimension = i, ]
                ];

                (* Extract values at [dimension] and take median *)
                medianval = Median[Map[Part[Part[#,1],dimension]&,
                                                inData]];

                (* If the median is the same as the hi or low values
                 * for the dimension, then repeated values must exist
                 * in the data set. Return the set without splitting
                 *)
                If[thisLow == medianval || thisHi == medianval,
                        Return[inData],
                (* Else return the node in the tree with the correct
                 * subtrees *)
```

```
                    pair = split[dimension, medianval, inData];
                    Return[{dimension, medianval,
                            buildTree[k, First[pair]],
                            buildTree[k, Last[pair]]}]
                ]
        ]
];
```

The procedure **split** called by the above is used partition the data about the median value. It requires three parameters: *list* is the data to be partitioned, *d* is the dimension with the largest spread, and *v* is the median value to split around.

```
split[d_, v_, list_] :=
        Module[
        {i, hi = {}, low = {}},
        For[i = 1, i <= Length[list], i++,
                If [getValueFor[list, i, d] >= v,
                        hi = Append[hi, list[[i]]],
                        low = Append[low, list[[i]]]
                        ]
        ];
        Return[{low, hi}]
];
```

3 Searching the kd-Tree for Nearest Neighbours

Once the kd-tree has been built, it can be used to find the nearest neighbours for any single point. The search process is remarkable simple - given a query data point and the tree, the algorithm first searches through the tree to find the closest terminal node to the query, then it repeatedly searches outward from the query point to the next nearest terminal node.

The **searchTree** procedure relies on two decision procedures in order to guide the search. The first, **ballWithinBounds** is called to determine if the search can terminate. It tests to see if the remaining region to be searched is further away than the furthest nearest neighbour. This is done by comparing the distance from the query record to the boundaries: if any boundary is closer than the furthest nearest neighbour then the search must continue. The parameters *query* and *k* are the query node and the dimensionality of the data.

The several global variables used within the procedure are described below:

nearest : This is a priority queue of the best nearest neighbours found so far. It is formed from a pair of lists - one for the record identifiers and the other for their associated distances for the query point. Both lists are maintained in increasing distance from the query node, so that the furthest nearest neighbour always lies at the end of the list. Before each nearest neighbour search, each distance is initialised to ∞.

lowerBounds : This is the set of k lower bounds currently defining the edges of the search region. It is initialised to $-\infty$ before each nearest neighbour search.

upperBounds : This is the set of k upper bounds currently defining the upper edges of the search region. It is initialised to ∞ before each nearest neighbour search.

```
ballWithinBounds[query_, k_] :=
        Module[
        {d},
        For [d = 1, d <= k, d++,
                (* "Last[nearest[[2]]]" is the furthest
```

```
             * nearest neighbour *)
            If [((query[[d]] - lowerBounds[[d]])^2 <=
                        Last[nearest[[2]]]) ||
                    ((query[[d]] - upperBounds[[d]])^2 <=
                        Last[nearest[[2]]]),
                    Return[False],
            ]
        ];
        Return[True]
];
```

The **boundsOverlapBall** test works on similar lines, but is slightly more complex. Its purpose is to decide whether or not to search down the second branch of a non-terminal node on backtracking from a search down the first. It is this test which allows the search to ignore branches which do not contain any points which might become part of the final nearest neighbours list. It does this by testing to see if the geometric boundaries of the branch under consideration are closer than the furthest nearest neighbour. If so, then the branch must be search, otherwise then branch can be ignored. (The paper by Bentley from which this algorithm was taken has the true/false return values switched. This causes the algorithm to fail; I suspect a typing error crept into the pseudo-code).

Again, the function takes parameters for the query node under investigation and the dimensionality of the data.

```
boundsOverlapBall[query_, k_] :=
        Module[
        {total = 0, d},
        (* "Last[nearest[[2]]]" is the furthest
         * nearest neighbour *)
        For[d = 1, d <= k, d++,
            If[query[[d]] < lowerBounds[[d]],
                total = total + (query[[d]] - lowerBounds[[d]])^2;
                    If[total > (Last[nearest[[2]]])^2,
                    Return[False],
                    (* else nothing *)
                ],
            (* else *)
                If[query[[d]] > upperBounds[[d]],
                    total = total + (query[[d]] - upperBounds[[d]])^2;
                    If[total > (Last[nearest[[2]]])^2,
                        Return[False],
                        (* else nothing *)
                    ],
                (* else nothing *)
                ]
            ]
        ];
        Return[True]
];
```

The code for **searchTree** is in two main halves - one for the terminal node case, and the other for the non-terminal node case. Whenever a leaf node is encountered, the points within it are added to the priority queue if their distances to the query node are less than the current furthest nearest neighbour. If the query node itself is encountered, then it is ignored as it is not required that the query node be included in the list of nearest neighbours.

If the node is non-terminal, the branch closer to the query node is searched first. On backtracking, the **boundsOverlapBall** test is used to decide if the further branch must be searched.

It is this ability to ignore sections of the tree which allows the nearest neighbours for a single query point to be found in time proportional to $O(logn)$.

The procedure takes parameters for the query node, the dimensionality of the data and the previously constructed kd-tree.

```
searchTree[query_, k_, tree_] :=
    Module[
    {medianval, dimension, temp, i, distance, maxloc, maxdist},

    Print["upper = ", upperBounds, " lower = ", lowerBounds,
        " tree = ", tree];
    If[Not[NumberQ[tree[[1]]]],
        (* Node is terminal - update nearest neighbours *)
        For[i = 1, i <= Length[tree], i++,
            (* Add points in bucket if closer than furthest
             * found so far, so long as this is not the query
             * record *)
            maxdist = Max[nearest[[2]]];
            distance = norm[query - tree[[i]][[1]]];
            If [(distance < maxdist) &&
                (query != getPoint[tree, i]),
                maxloc = Flatten[Position[nearest[[2]], maxdist]][[1]];
                nearest[[1, maxloc]] = getIndex[tree, i];
                nearest[[2, maxloc]] = distance, ]
        ];
        (* Sort nearest neighbours list *)
        nearest = Transpose[Sort[Transpose[nearest],
                        (Part[#1, 2] < Part[#2, 2])&]];

        (* Check for termination and return *)
        If[ballWithinBounds[query, k],
            Print["**done**"]; Return[ ],
            Return[ ]],
    ];

    (* Node not terminal - call searchTree recursively *)
    dimension = tree[[1]];
    medianval = tree[[2]];
    (* Recursive call on closer son *)
    If[query[[dimension]] <= medianval,
        temp = upperBounds[[dimension]];
        upperBounds[[dimension]] = medianval;
        searchTree[query, k, tree[[3]]];
        upperBounds[[dimension]] = temp,
    (* else *)
        temp = lowerBounds[[dimension]];
        lowerBounds[[dimension]] = medianval;
        searchTree[query, k, tree[[4]]];
        lowerBounds[[dimension]] = temp
    ];

    (* Recursive call on further son if necessary *)
    If[query[[dimension]] <= medianval,
        temp = lowerBounds[[dimension]];
        lowerBounds[[dimension]] = medianval;
        If[boundsOverlapBall[query, k],
```

```
                searchTree[query, k, tree[[4]]],
            ];
            lowerBounds[[dimension]] = temp,
        (* else *)
            temp = upperBounds[[dimension]];
            upperBounds[[dimension]] = medianval;
            If[boundsOverlapBall[query, k],
                searchTree[query, k, tree[[3]]],
            ];
            upperBounds[[dimension]] = temp
        ];

        (* See if we should terminate *)
        If[ballWithinBounds[query, k],
            Print["**done**"]; Return[ ],
            Return[ ]]
    ];

    norm[x_] :=    N[Sqrt[x.x]]; (* vector normalisation *)
```

Now that the nearest neighbours for a single point can be found, the method can be applied to all points. It is necessary to initialise the global variables used before each search is performed. This is done by the following procedure, which takes parameters for the query point and the number of nearest neighbours desired:

```
    initialise[query_, numNearest_] :=
        Module[
        {},
        nearest = {Table[-1, {numNearest}],Table[Infinity, {numNearest}]};
        upperBounds = Table[Infinity, {Length[query]}];
        lowerBounds = Table[-Infinity, {Length[query]}]
```

The final piece of code is the top level nearest neighbours function. Given a list of k-dimensional tuples in *inData*, and the number of nearest neighbours required, it returns a list of pairs, where each pair is a list of the point identifiers (index numbers) and a list of their respective distances from the query node.

```
    nearestNeighbours[inData_, numNearest_] :=
        Module[
        {tree, query, indexedData, currentList = {}, i},
        indexedData = addIndex[inData];
        Print["Building Search Tree..."];
        tree = buildTree[Length[inData[[1]]], indexedData];
        Print["Starting Nearest Neighbour Search..."];
        For[i = 1, i <= Length[inData], i++,
            Print["Processing ", i];
            query = indexedData[[i]][[1]];
            initialise[query, numNearest];
            searchTree[query, Length[query], tree];
            currentList = Append[currentList, nearest]
        ];
        Return[currentList]
    ];
```

Recall that the search can terminate whenever the **ballWithinBounds** test returns true. If the procedure **searchTree** was converted to a function and the result of this **ballWithinBounds** test was returned as its value, then the search process could be terminated whenever a recursive call to **searchTree** returned true. This has not been done in the code above in order to keep it simple and to keep it similar to the pseudo-code provided by Bentley.

4 An Example Run

This section intends to illustrate the workings of the algorithm by means of a small example. Suppose it is required to extract three nearest neighbours from each of the eight two-dimensional data points in figure 1.

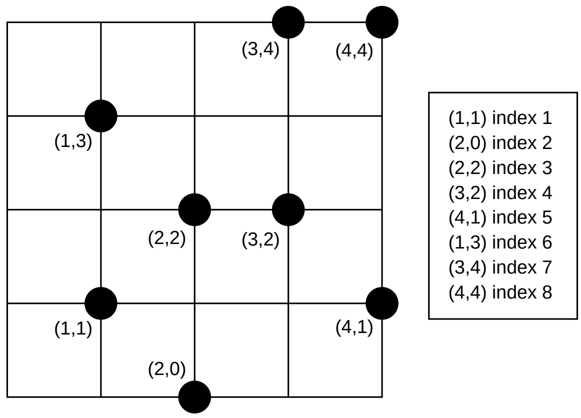

Figure 1: The Initial Data

The first step is to build the kd-tree for the points, remembering to pre-process the data points to store their indices. After the **buildTree** procedure has completed, the tree will look like the one in figure 2. Note that the values of the dimension and the median value on which the split was performed are stored in each non-terminal node of the tree. The next step is to find the nearest neighbours for each point in turn.

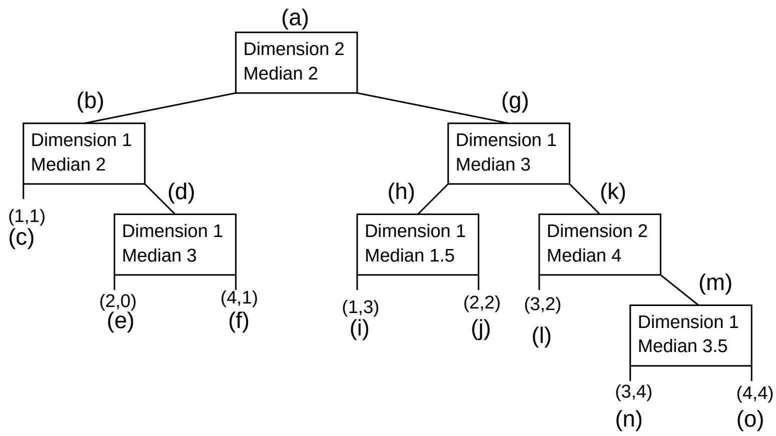

Figure 2: The completed kd-Tree

To simplify matters, only the point $(1,1)$ will be considered - this is the the first point in the data set and has an index of 1. Figure 3 shows the upper and lower bounds at each node in the tree: the hashed region shows the area currently being searched. It is these bounds which the two control procedures **ballWithinBounds** and **boundsOverlapBall** test. Table 4 shows the contents of the nearest neighbour priority queue, the radius of the current "ball" (defined to be the distance of the furthest nearest neighbour from the query point), and the values of the control procedures **ballWithinBounds** and **boundsOverlapBall** for the current node.

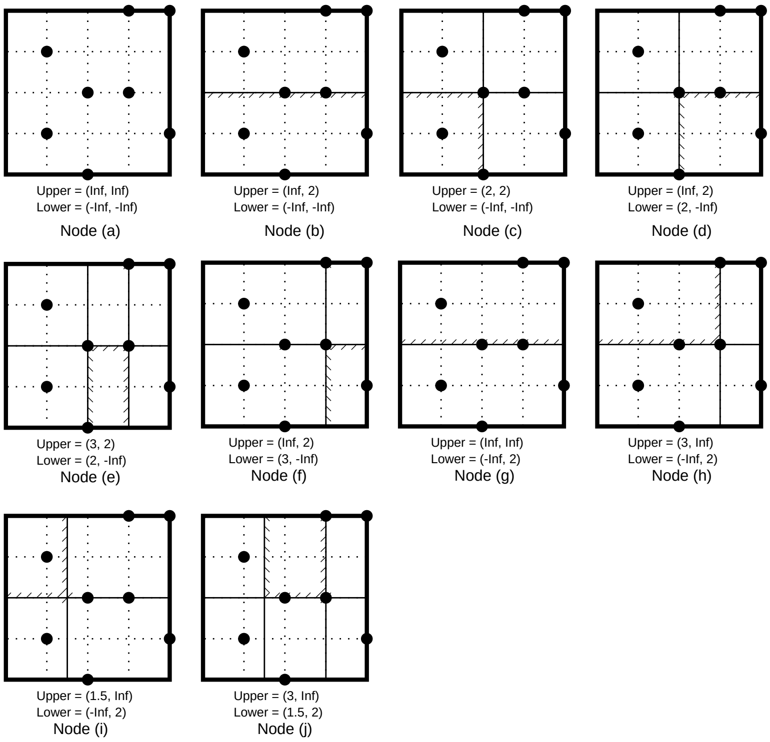

Figure 3: Upper and Lower Bounds trace for point $(1,1)$

Initially, the algorithm looks for the "closest" node to the query node $(1,1)$ - it rapidly finds the terminal node (c) in step 3, but as this is the query node, it does not record it in the nearest neighbour list. The search then proceeds down the tree to node (e), where the first nearest neighbour candidate is found. Notice that until the nearest neighbour list fills up with candidates, the distance to the furthest nearest neighbour is ∞, ensuring that the **ballWithinBounds** test will return false and that the **boundsOverlapBall** test will return true. This effectively means that until the nearest neighbour list fills up, the tree will be searched exhaustively in order of increasing distance from the query node.

All slots in the nearest neighbour list become filled step 14, but the **boundsOverlapBall** test does not return false until step 18. This can be seen from figure 3, in the bounds for node (g). It must be remembered that the bounds are modified before the test is called - in this case, the upper bounds are (∞,∞) and the lower ones are $(3,2)$, ie. the region represented by the right subtree is a rectangle with the point (2,3) as its lower left hand corner. This means that the circle of radius 2 centred on the query point $(1,1)$ is not overlapped by the bounds, so the test can return false, as can be seen in figure 4. The main effect of this is that the right branch of node (g) does not need to be searched - all points contained inside the bounded region must be further away than the furthest nearest neighbour found so far.

5 Experimental Results

As a test, the algorithm above and a naive $O(n^2)$ algorithm were both coded in C, using long double precision maths. The utility of using *Mathematica* as a prototyping language became obvious during

Step	Node Label	Furthest NN Distance	Nearest Neighbour Priority Queue			Ball Within Bounds?	Bounds Overlap Ball?
1	(a)	∞	-	-	-	-	-
2	(b)	∞	-	-	-	-	-
3	(c)	∞	-	-	-	No	-
4	(b)	∞	-	-	-	-	Yes
5	(d)	∞	-	-	-	-	-
6	(e)	∞	(2,0)	-	-	No	-
7	(d)	∞	(2,0)	-	-	-	Yes
8	(f)	∞	(2,0)	(4,1)	-	No	-
9	(d)	∞	(2,0)	(4,1)	-	No	-
10	(b)	∞	(2,0)	(4,1)	-	No	-
11	(a)	∞	(2,0)	(4,1)	-	-	Yes
12	(g)	∞	(2,0)	(4,1)	-	-	-
13	(h)	∞	(2,0)	(4,1)	-	-	-
14	(i)	3	(2,0)	(1,3)	(4,1)	No	-
15	(h)	3	(2,0)	(1,3)	(4,1)	-	Yes
16	(j)	2	(2,2)	(2,0)	(1,3)	No	-
17	(h)	2	(2,2)	(2,0)	(1,3)	No	-
18	(g)	2	(2,2)	(2,0)	(1,3)	No	No
19	(a)	2	(2,2)	(2,0)	(1,3)	Yes	-

Table 1: Trace of run for point (1,1)

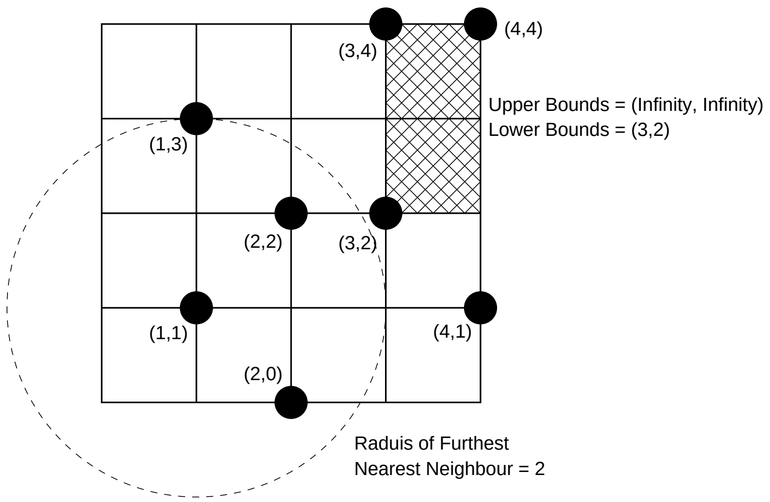

Figure 4: The *boundsOverlapBall* test at step 14

this exercise. Not only was it useful in verifying the correctness of the output, but time was also saved in being able to test each block of code as it was written.

Both algorithms were then timed on a set of sample datafiles, at sizes of 100, 200, 400, 800, 1600 and 3200 points. The tests were run three times to obtain an average time for each algorithm, and all tests were set to find the 30 nearest neighbours for all the points in the data set. The experiments were performed on a SPARC 4+ workstation; the results are displayed in figure 5.

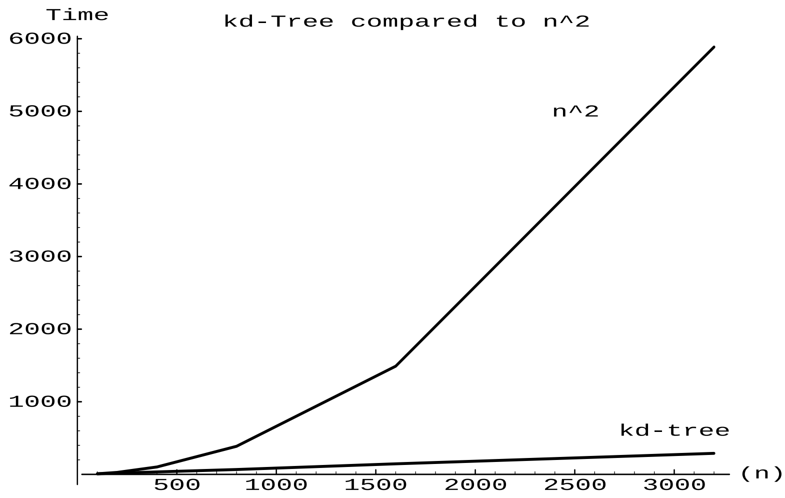

Figure 5: Comparison of $O(n^2)$ Algorithm with kd-Tree

It can be seen that the $O(n^2)$ algorithm was rapidly outstripped by the kd-tree algorithm. Indeed, at the largest sample size of 3200 points, the $O(n^2)$ algorithm required over 1.5 hours, while the kd-tree algorithm needed just under five minutes. These results agree with those in [2].

Another set of experiments was carried out to investigate the effect of increasing the bucket size (recall that this is the number of records stored at each terminal node in the kd-tree). The same data files were used on versions which stored 1, 4 and 8 records at each terminal node. The results are graphed in figure 6.

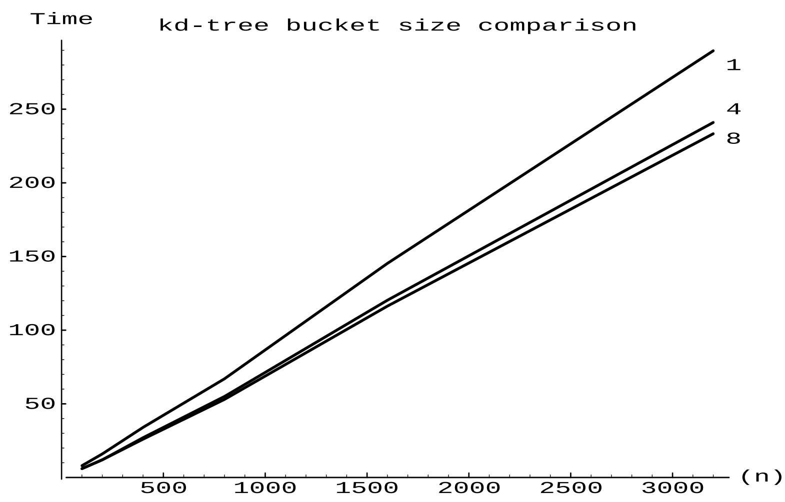

Figure 6: Bucket Size Comparison for kd-Tree

The graph shows that while increasing the bucket size from 1 does give a performance increase, the effect is reduced above a bucket size of 4. These results again agree with Bentley.

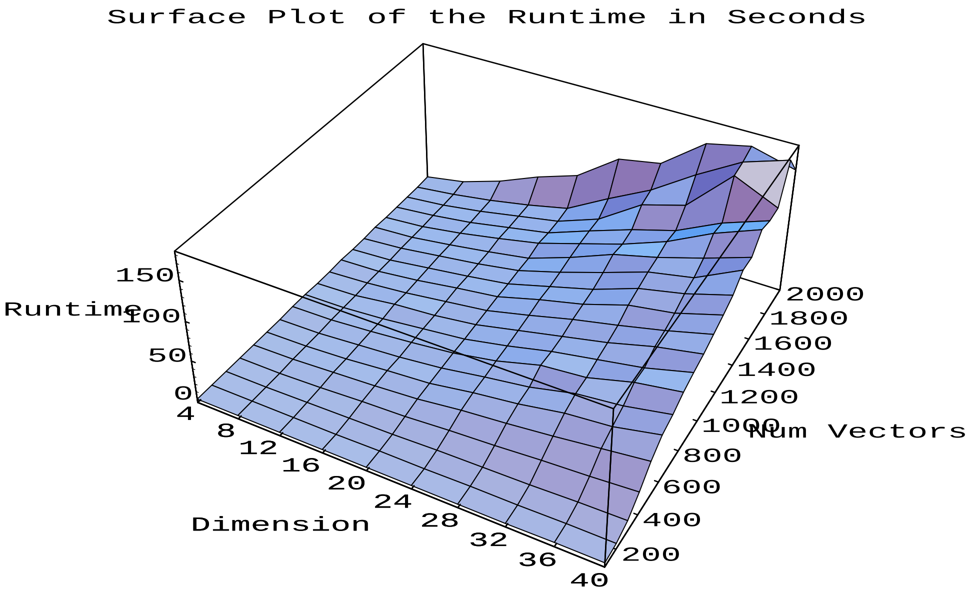

Figure 7: Effect of increasing dimension on run time

The final graph shown in figure 7 shows how the run time scales with increasing dimension and increasing data set size. The data for this graph was generated by running a series of tests on randomly generated data files where dimension sizes ranged from 2 to 40 and data set sizes ranged from 100 to 2000 points, and all timings were taken on an Alpha workstation (although using only double precision maths). The graph is a surface plot of the run times – the x-axis is dimension, the y-axis is the number of vectors and the z-axis the run time in seconds. Again, the almost linear scaling over the number of vectors can be seen, as is the slight non-linear scaling with increasing dimension.

6 Summary

The kd-tree algorithm due to Bentley[2] runs in $O(nlogn)$ time, and requires only a little more program complexity than the naive $O(n^2)$ algorithm. For this reason, the kd-tree algorithm is recommended for any nearest neighbour computation. *Mathematica* has proved invaluable as a prototyping language for this application.

References

[1] R Jain D.A White. Similarity indexing: Algorithms and performance. Technical report, Hmm, 1996. Available at `http://vision.ucsd.edu/papers`.

[2] R.A Finkel J.H Friedman, J.L Bentley. An algorithm for finding best matches in logarthimic expected time. *ACM Transactions on Mathematical Software*, 3(3):200 – 226, 1977.