Data Structures for Computer Graphics

Proximity Search and its Applications I

Lectured by Vlastimil Havran

Lectures Outline

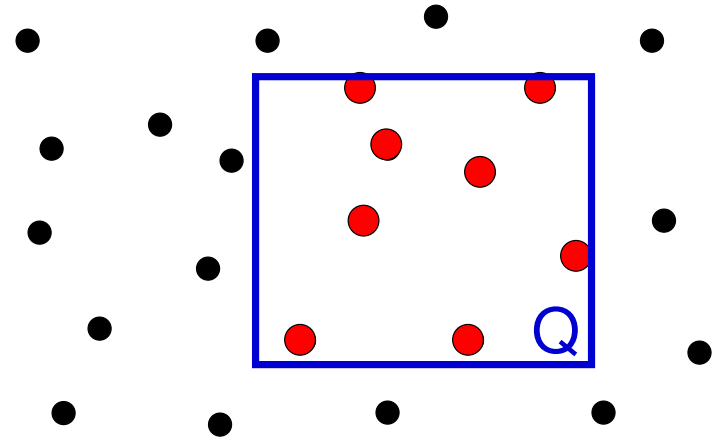
- Range search
- Nearest neighbor search (NN)
- k-nearest neighbor search (kNN)

Proximity Search

- given a fixed-size range, find all the data in the range

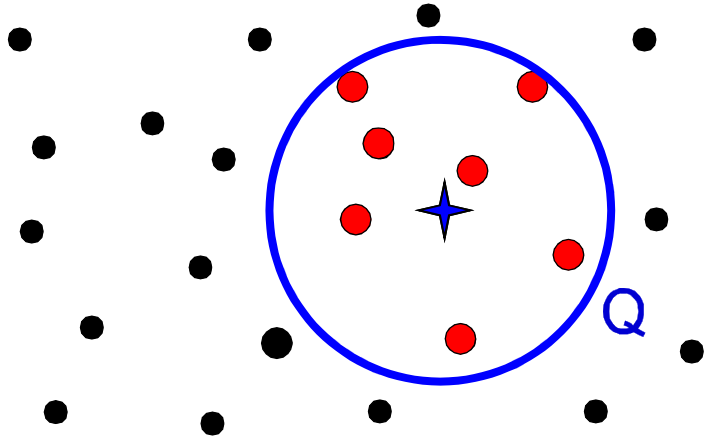
- Range search:
  - Window query:

    (box query)



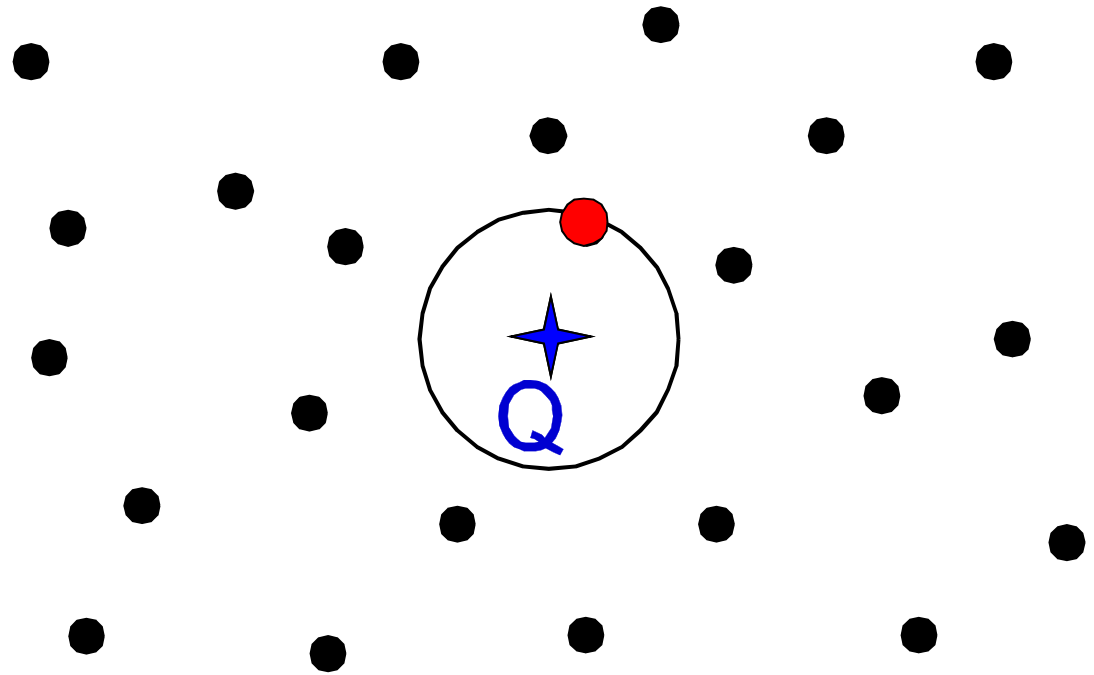
  - Circle query:

    (center + radius)



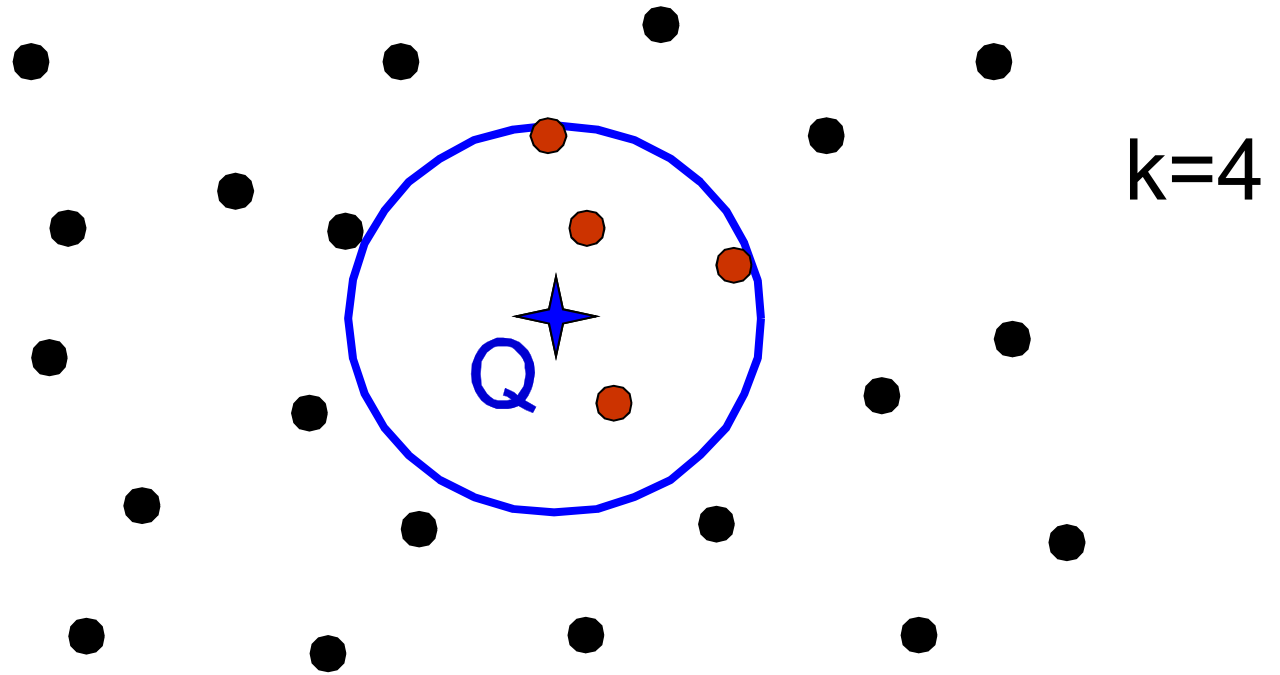
Nearest Neighbor Search (NN-search)

- Find the nearest neighbor given a query using a distance metric:



K-Nearest Neighbor Search (k-NN search)

- Find all k nearest neighbors given a query center using a distance metric



Distance Functions

- Let *X* be the universe of valid objects
- We have two objects ***x*** and ***y*** in *X*
- *Distance functions* between two objects ***x*** and ***y*** must have following properties
  - ***Positiveness***: for all x,y in X, d(x,y) >= 0
  - ***Symmetry***: for all x,y in X, d(x,y)=d(y,x)
  - ***Reflexivity***: for all x in X, d(x,x) = 0
- In most cases also ***strict positiveness*** is necessary
  - For all x,y in X, x is different to y, then d(x,y) > 0

True Metric

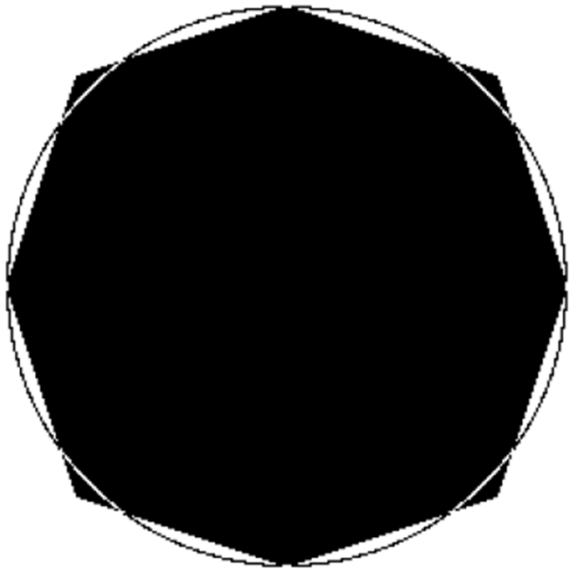
- *Metric* is such a distance function that fulfills also
  - *triangular inequality*:

  For all x,y,z in X, d(x,y) <= d(x,z) + d(z,y)

- *Metric space* is a pair *(X,d)*, where *X* is **universum** and *d* is **metric**

Euclidian Metric Approximations

- Usage: to compute results a bit faster on some computer architectures (with slow square root)
- Examples: Chamfer, Octagonal, and D-Euclidean metric. Details in the book of Samet [2005].
- Chamfer metric in 2D
  - e=| |px-qx| - |py-qy| |
  - f = min { |px-qx|, |py-qy| }
  - $d_{(a,b)}(p,q) = a * e + b * f$
  - $d_{(a,b)}(p,q)$ is close to Euclidean metric.



Example: a=3 and b=4

Vector Space

- Vector space is such a metric space (X,d), where the elements of universum X are tuples of real numbers (=vectors).
- The most widely used distance functions
  - $Lp((x1,\ldots,xn),(y1,\ldots,yn)) = (sum(|xi-yi|^p))^{1/p}$
  - L_1 … Manhattan distance (city-block metric), p=1
  - L_2 … Euclidean distance (our notion of distance), p=2
  - L_{inf} … maximum distance, p = infinity,

    L_{inf} = for all i (max_i|xi-yi|)

Algorithm Classification

- The dimensionality (of a problem)
  - low dimensions (frequent for CG)
  - high dimensions (similarity search)
- Accuracy
  - exact
  - approximate
- The data category
  - point data
  - non-point data
- Result type (for fixed-size range searching)
  - counting … give only the number points in the range
  - reporting … report all the points in the range

Algorithm Requirements (NN-search)

- O(N) space complexity
- At most O(N log N) construction time
- Almost O(log N) search time
- Low dependence on data dimensionality
- Practicality (numerical robustness, low constants behind big-Oh notation)

Curse of Dimensionality

- **Observation**

  the number of points to estimate an arbitrary function grows exponentially with the number of variables that the function comprises

- **Consequence**

  the complexity of search algorithms (with kd-trees etc.) grows exponentially with the dimensionality of data.

Notion Behind Curse of Dimensionality

- 100 values sufficient for sampling 1D interval
- 100 values completely insufficient for sampling 10-dimensional hypercube – for the same density of samples (=accuracy of function representation) you need 10^{20} samples
- Therefore increasing problem size by 9 dimensions induces the increase of problem complexity of problem by factor 10^{18}

Handling Object Data

Two options:

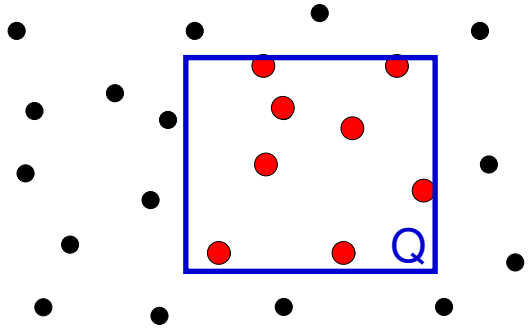
- Using distances to real object: it can be computationally expensive, it requires BVH-type hierarchies such as R-trees
- Using representative points
  - Centers of objects
  - Several points located at feature points of objects
  - Several points distributed uniformly over the objects

Lecture Content Below

- Algorithms suitable for low-dimensional data
  - Exact window range search with kd-trees
  - Exact NN-search with kd-trees
  - Exact circular range search with kd-trees
  - Exact kNN-search with kd-trees
  - Tricks and hints for the faster algorithms.

Note: NN stands for nearest neighbor.

Exact Window Search with kd-trees



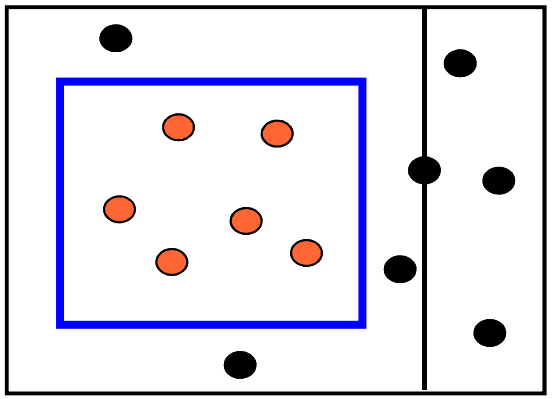
- Input point data in vector space R^N
- Preprocessing:
  - Compute the box in R^N tightly encompassing all data.
  - construct kd-tree with spatial median in top-down fashion, using sliding midpoint strategy, partitioning axis – n% round robin, 100-n% largest extent
  - Put possibly several points (10 to 20) into the leaves if you expect queries of large size
  - Use efficient packing of interior nodes, for 32-bits computers using only 64 bits per interior node.

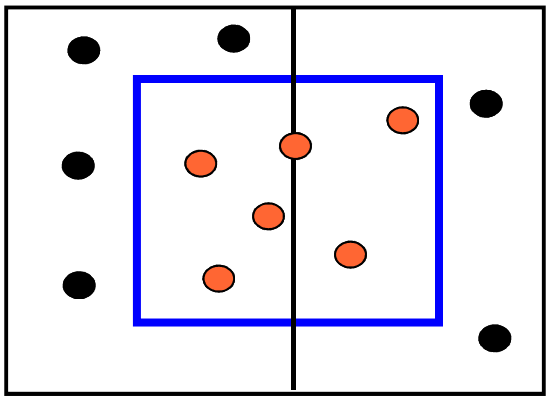
Window Search Algorithm for kd-trees

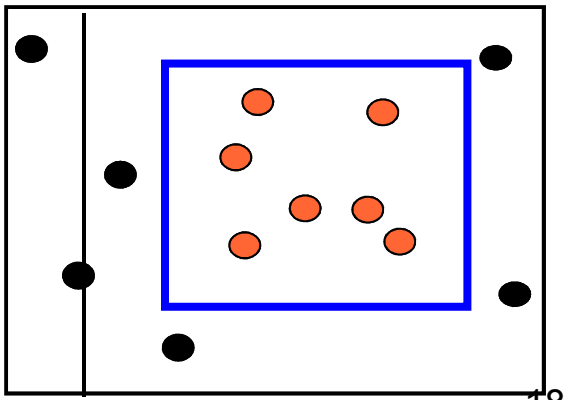
- Put the root of kd-tree to the stack.
- While loop until stack is not empty
- Pop a node “N” from the stack
  - If “N” is a leaf, check the points against the window, report all the points in the window. Continue.
  - If “N” is an interior node, decide between three cases:
    - ➔Visit only left child: push left child to the stack. Continue.
    - ➔Visit only right child: push right child to the stack. Continue.
    - ➔Visit both children: put left child to the stack and push right child to the stack. Continue.
- End of while loop

Interior Node Decision for Window Query, in R^N space

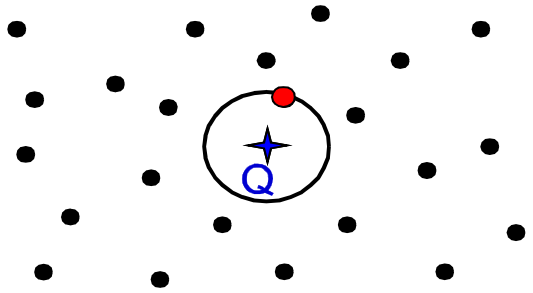
- The query window is given as <c_i - s_i,c_i + s_i> for all i in n, that is center c_i and size 2*s_i
- Kd-tree node has axis “a” and position “p”
- Algorithm:
  - If c_a+s_a < p then traverse left child
  - If c_a-s_a > p then traverse right child
  - Otherwise traverse both children







Exact NN-search



- Naïve solution without preprocessing:
  - Compute the distance between the query to all the data. Pick up a minimum. This requires O(N) time for searching.
  - It is convenient if we know that the number of queries is very limited. If the number of queries is M <= log N, then we use a naïve algorithm.

    (Proof: as homework)
- The same reasoning applies to *farthest neighbor algorithm*.

Exact NN-search with kd-trees

- Preprocessing in O(N log N), space requirement is O(N), searching algorithm works in $O(2^d * \log N)$, where d is the dimensionality of data.
- The ANSI-C structure used for the node (not optimized)

```
struct kd_node {
    int leaf_node; // 1 if leaf, 0 if internal node
    int cut_dim; // cutting dimension for internal node
    float cut_val; // cutting position
    kd_node *lo_child, *hi_child; // left and right children
    int   pt; // data point index
};
```

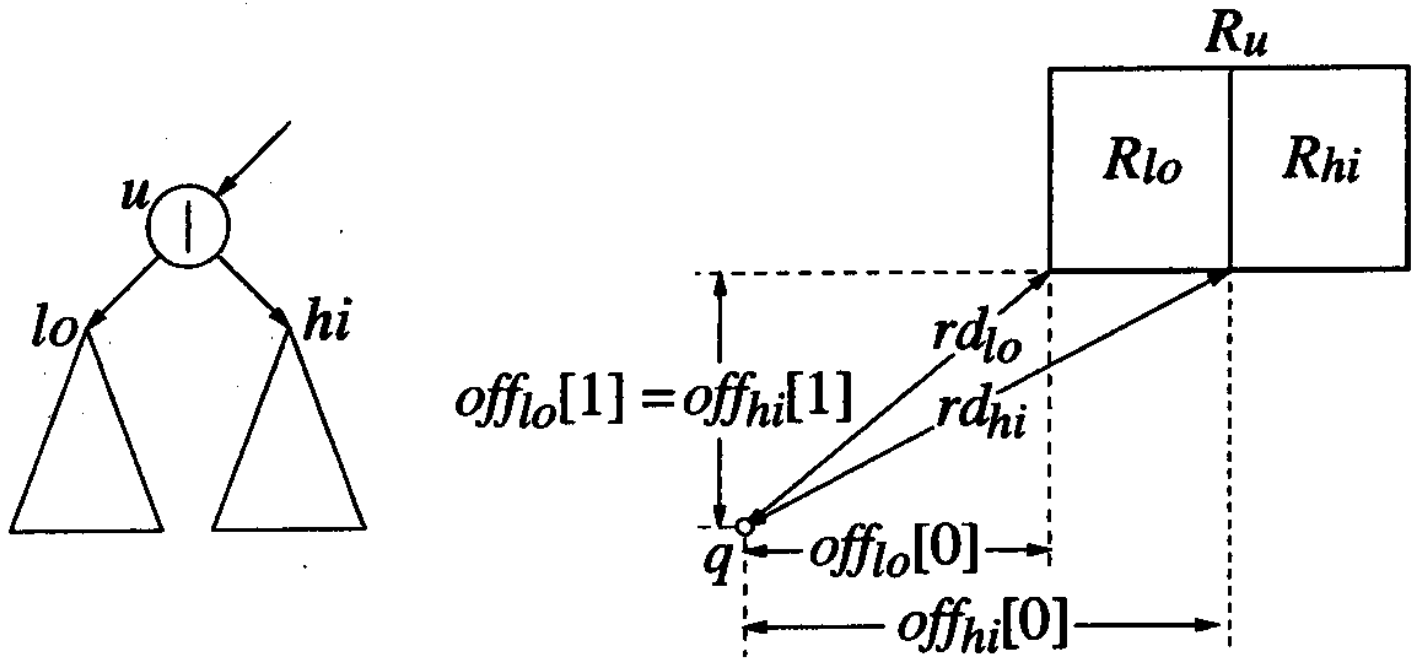
Global Variables used in Search

- d … the number of dimensions
- n … the number of points
- struct Point {

  float coor[d]; // the point data

  };
- Point points[n] … the input data
- Point q … query point
- float offset[d] … the array of offfsets
- float nn_dist2 … best squared distance so far

Distance from Query to Box Incrementally



- Knowing rd_{lo}, we can compute rd_{hi} incrementally:

$$(rd_{hi})^2 = (rd_{lo})^2 - (off[cut_{dim}])^2 + (q[cut_{dim}]\text{-}cut_{val})^2$$

Variable "off[i]" stores the distance from query to the cutting plane in dimension "**i**", it is then updated:

$$off[i] = q[cut_{dim}]\text{-}cut_{val}$$

Starting Search Procedure for NN

```
float
kd_standardNN(
    Point qq, // query point
    kd_node  *root) // root of the kd-tree
{
   q = qq;                 // save query point
   nn_dist = HUGE; // initial distance to nearest neighbor
   for (int I = 0; I < d; i++) off[i] = 0.0;
   recursiveSearchNN(root, 0.0);
   return sqrt(nn_dist2) ; // the distance to the nearest neighbor
}
```

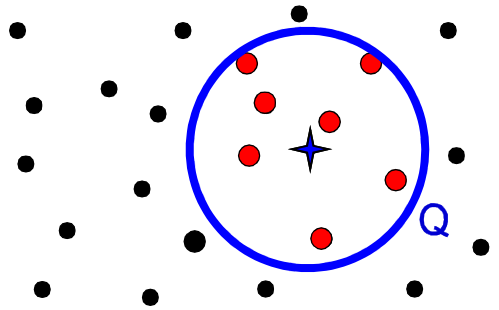
Recursive Search Procedure for NN

```
void recursiveSearchNN(
    kd_node *u, // current node
     float       rd  // squared distance to this node
{
  if (u->leaf_node) { // if this is a leaf
     for all points update nn_dist2 = Min(nn_dist,dist2(q,points[u->pt])); // update min distance
  } else { // internal node
     int cd = u->cut_dim;  // cutting node dimension
     float old_off = off[cd]; // save old offset
     float new_off = q[cd] – u->cut_val; // offset to further child
     if (new_off < 0) {  // query is on the left of the cutting plane
        recursiveSearchNN(u->lo_child, rd);   // search closer subtree first
        rd += -old_off * old_off + new_off*new_off; // distance to farther child
        if (rd < nn_dist) {  // closer than currently found NN candidate
            off[cd] = new_off;   // update offset
            recursiveSearchNN(u->hi_child, rd);   // search further subtree
            off[cd] = old_off; // restore offset
        }
     }
     else {  // query  q is on the right of the cutting plane
         .. Analogous  with lo_child and hi_child interchanged..
     }
  }
}
```

Comments to NN-Search Algorithm

- The distance to the box associated with the interior node is computed in O(1) time (independent of the number of dimensions)
- Efficient implementation requires to rewrite the recursive version to non-recursive one with a stack.
- Both interior nodes/leaves should be represented much more space-efficiently, such as 16 Bytes for the both structures, see previous lectures.
- It is convenient to rewrite the real point data pointed in leaves according to the order given by kd-tree, the spatial locality of program accesses during search is then much better.

Exact Circular Range Search with kd-trees



- Only slight modification of the algorithm above. We use **”rad2”** as the radius of the query in the pseudocode.
- We do not need to keep the smallest distance **”nn_dist”** to the nearest neighbor so far.
- We report or count the points in leaves, if they are closer than **”rad2”**

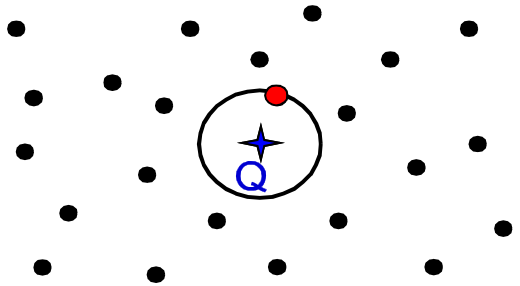
Starting Search Procedure for Circular Range Search, Counting

```
float
kd_circularRangeSearch(
    Point qq, // query point
    kd_node  *root, // root of the kd-tree
    float    rad2) // the squared distance of the search
{
   q = qq;                 // save query point
   nn_dist = rad2;     // the squared radius to global variable nn_dist
   for (int I = 0; I < d; i++) off[i] = 0.0;
   count = 0; // the number of points in the range
   recursiveCircularRangeSearch(root, 0.0);
   return count; // return the number of points found in the range
}
```

Recursive Search Procedure for Circular Range Search, Counting

```
void recursiveCircularRangeSearch (
    kd_node *u, // current node
     float       rd  // squared distance to this node
{
  if (u->leaf_node) { // if this is a leaf
     if (dist2(q,points[u->pt]) <= nn_dist)  { count++; …report point on the output…}
  } else { // internal node
     int cd = u->cut_dim;  // cutting node dimension
     float old_off = off[cd]; // save old offset
     float new_off = q[cd] – u->cut_val; // offset to further child
     if (new_off < 0) {  // query is on the left of the cutting plane
        recursiveCircularRangeSearch(u->lo_child, rd);   // search closer subtree first
        rd += -old_off * old_off + new_off*new_off; // distance to farther child
        if (rd < nn_dist) {  // node closer than specified squared radius
            off[cd] = new_off;   // update offset
            recursiveCircularRangeSearch (u->hi_child, rd);   // search further subtree
            off[cd] = old_off; // restore offset
        }
     }
     else {  // query  q is on the right of the cutting plane
         .. Analogous  with lo_child and hi_child interchanged..
     }
  }
}
```

Faster NN-search with kd-trees using Priority Queue

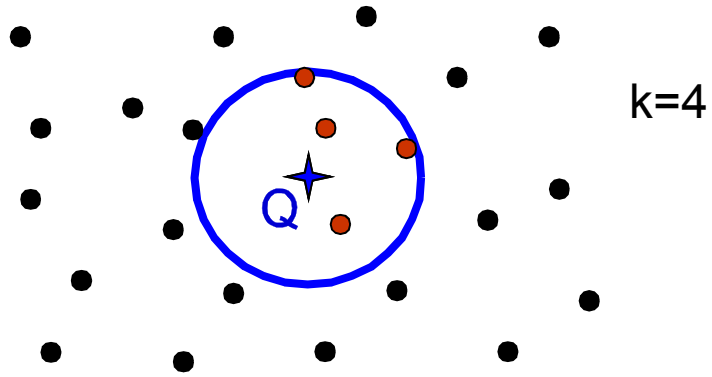


- We use priority queue **”QN”** to prioritize the search in the nodes that are likely to contain the nearest neighbor.
- The priority queue **”QN”** is organized according to the distance from the query to the boxes associated with the nodes.
- You need an efficient implementation of priority queue, such as used in STL etc.

Recursive Search Procedure for NN search

```
void kd_prioritySearchNN(
    Point qq, // query point
    kd_node  *root, // root of the kd-tree
{
   PriorityQueue Q;
   nn_dist = HUGE; // Initial distance
   kd_node *u; // current node
   float rd; // distance to the box associated with internal node
  Q.Insert(root, 0.0);
  while (Q.NotEmpty()) { // repeat until queue is empty
   ▌ Q.ExtractMin(u, rd);  // closest node to query point
   ▌ if  (rd >= nn_dist) break;    // further from nearest so far, this is end
   ▌ while (!u->leaf_node) { // descend until leaf is found
   ▌ ▌ int cd = u->cut_dim;  // cutting dimension
   ▌ ▌ float old_off, new_rd; // auxiliary variables for offset and distance
   ▌ ▌ float new_off = q[cd] – u->cut_val; // offset to further child
   ▌ ▌ if (new_off < 0) {  // query is on the left (below) of the cutting plane
   ▌ ▌      old_off = q[cd] – u->low_val;  // compute offset
   ▌ ▌      if (old_off > 0)   // overlaps interval
   ▌ ▌          old_off = 0;
   ▌ ▌      new_rd = rd - old_off * old_off + new_off * new_off; // distance to further child
   ▌ ▌      Q.Insert(u->hi_child, new_rd);  // insert the further child to priority queue for later
   ▌ ▌       u = u->lo_child; // first visit the child lo_child
   ▌ ▌ }
   ▌ ▌ else {  // query  q is on the right of the cutting plane
   ▌ ▌     .. Analogous  with lo_child and hi_child interchanged..
   ▌ ▌ }
   ▌   }  // while until leaf is found
   ▌  nn_dist = min(nn_dist, dist2(points[u->pt], q);   //  leaf, update the nearest distance neighbor if closer
    } // while
}
```

K-Nearest Neighbor Search with kd-trees



- Analogous to NN search with priority queue
- We only need the second priority queue ”QD” to organize the points according to the distance
- QD size is always kept to only **“k”** entries
- **”nn_dist”** is updated whenever the k-NN neighbor is successfully updated in the QD
- After popping node from priority queue ”QD” we always skip such nodes that are farther than **”nn_dist”**.

kNN Search based on Circular Range Search

- Motivation: maintaining the order using a priority queue is costly, each insertion and deletion in priority queue is O(log N).
- We can estimate the radius of sphere R to contain all ”**k**” entries
  - If the number of found queries is smaller than required ”**k**”, we incrementally increase the radius and start again
  - Otherwise (number of found entries < k), we sort the entries according to the distance to ”**q**” and we issue to output first ”**k**” entries

Two Methods to Estimate Radius for kNN Search

1. Use the result of the same size as the previous query for the same **”k”**. It assumes the query is similar to the previous query.
2. Estimate the number of data based on the diagonal of cells created during interior node construction. Store the density of data per volume in the interior nodes/leaves.

Please, Recall Priority Queue (PQ)

- How priority queue is implemented, check C++ documentation:

http://www.cplusplus.com/reference/stl/priority_queue/

- What are the operations for priority queues.
- Check the other implementations of PQ.

Thank you for your attention!