# Nearest-neighbor search
# Lecture Notes - 2/17/2011

The nearest-neighbor problem takes as input a query point $q$ and returns the point $p$ in some underlying set $P$ which is nearest to $q$. A common variant is the $k$-nearest neighbors problem, which returns the $k$ closest points in $P$ to $q$, where $k$ is a constant.

Many different spatial data structures can be used to answer nearest-neighbor or $k$-nearest-neighbor queries. The Voronoi diagram can be post-processed so that we can locate query points in the diagram, and thus answer nearest-neighbor queries, in $O(\lg n)$ time, but as we have seen the size of the Voronoi diagram grows exponentially with the dimension.

Octrees and kd-trees are more commonly used. In either case, the following *priority search* algorithm can be used; the pseudocode appears below.

As we have observed, the size of an octree can be unbounded, even in fixed dimension, if the distribution of points is very uneven. In practice, a depth of 15 or so is almost always sufficient in $\mathbb{R}^3$. We'll consider octrees of fixed depth, that is, of size $O(n \lg n)$, because in this case we can actually prove something about the running time of the algorithm. It is sufficient, for this purpose, for the octree representation to include only the *occupied* cells of the tree; empty leaves are not needed (unlike the mesh-generation application we saw earlier, where empty cells are needed). We store, with each cell, one of the data points which it contains; we call this the *representative* of the cell.

```
1: {find an Approximate Nearest Neighbor to query point $q$}
2: $r = 0$ {radius of ball which has been completely explored}
3: $\delta = \infty$ {distance to nearest point seen so far}
4: enqueue (bounding box) {queue is ordered by distance of nearest point in box to $q$}
5: while $\delta \geq r$ do
6:    dequeue box $B$, containing representative point $p$
7:    $r = d(q, B)$
8:    if $d(q,p) < \delta$ then
9:       $p$ becomes best choice seen so far and $\delta = d(q,p)$
10:   end if
11:   for all children $B'$ of $B$ containing points in $P$ do
12:      enqueue ($B'$)
13:   end for
14: end while
15: return p
```

This algorithm is sadly not magic. It finds an exact nearest neighbor but it might end up examining every cell in the octree (finding an example of a dataset and a query for which this is true is left as an exercise). If, however, we become a little less ambitious and seek not the exact nearest neighbor, but an *approximate* nearest neighbor, we can get a good bound on the running time. To do this, we change Line 5 to read:

**while** $\delta \geq (1 + \epsilon)r$ **do**

Here $\epsilon$ is the multiplicative approximation error we will allow; say 0.01 for one percent error.

To see that this limits the number of cells examined, let's consider the situation at the beginning of the last iteration of the algorithm, pictured in Figure **??**. The intuition is that the cells we have examined so far must be fairly large with respect to the distance to the nearest neighbor, since if we examine a very small cell we are guaranteed to discover an approximate nearest neighbor and we can stop (we stop when the representative of a cell that we open falls into the annulus between the ball of radius $r$ and the ball of radius $(1+\epsilon)r$; this always happens the cell has small diameter).

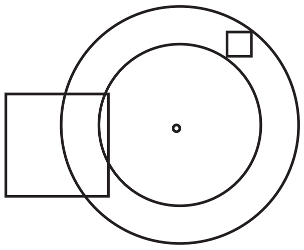

Figure 1: The situation right before the last step of the priority search algorithm. The small sphere has radius $r$ and the larger one $(1+\epsilon)r$; the point in the center is $q$. Once the small box is opened, we are guaranteed to find an approximate nearest neighbor. The large box on the left is an example of a cell which has be previously examined; it must intersect the inner sphere and also contain a representative point outside the outer sphere.

**Theorem 1** *On an octree, the priority search algorithm examines $O(\lg n)$ cells before finding an approximate nearest neighbor.*

**Proof:** Let $m$ be the number of cells examined by the algorithm in the while loop, and let $r_m$ represent the final value of $r$. Cells $1, \ldots, (m-1)$ all intersect the ball of radius $r_m$ surrounding $q$. Also, the representative points of cells $1, \ldots, (m-1)$ all must lie outside the sphere of radius $(1+\epsilon)r_m$ surrounding $q$ (otherwise the algorithm would have stopped at some earlier step). So none of cells $1, \ldots, (m-1)$ can have diameter less than $\epsilon r$.

How many such cells can there be, total? Let us first consider all cells $\sigma$ such that we examine $\sigma$ but not any child of $\sigma$. These cells will have mutually disjoint interiors. The maximum number of disjoint cells of diameter at least $\epsilon r$ that can cross the annular region between the sphere with inner radius $r$ and the sphere with outer radius $(1+\epsilon)r$ is a constant $c$, which depends on $\epsilon$ and the dimension $d$, but not on $r$.

For each of these $c$ cells, we may also examine its parent, its grandparent, and so on. But since we assume the depth of the octree is $O(\lg n)$, each smallest cell examined entails examining at most $O(\lg n)$ ancestors. So the total number of cells examined by this algorithm is $O(\lg n)$.
$\square$

This proof sadly cannot be applied to (most variants of) kd-trees.