Computational Geometry: Nearest Neighbor Search

Don Sheehy

April 15, 2010

1 Distance Queries

So far, the geometric search algorithms we have looked at so far are point location in the plane and range search. Today, we're going to look at a very important geometric search problem, called *the Post Office Problem*. It is also known as *Nearest Neighbor search* (NN). This is not the fist time we have seen this problem. You may recall that we saw it before in the context of Voronoi diagrams. Here is the set up.

Definition 1.1. *Given n points $P \subset \mathbb{R}^d$ and a query $q \in \mathbb{R}^d$, the nearest neighbor of q is*

$$NN(q) = \operatorname*{argmin}_{p \in P} |p - q| \tag{1}$$

Problem 1.1 (Nearest Neighbor Search)**.** *Given n points $P \subset \mathbb{R}^d$, preprocess them into a data structure that supports queries $q \in \mathbb{R}^d$ and returns $NN(q)$.*

2 We know how to do this in the plane

If we are only trying to solve this problem in the plane, we can combine some of the algorithms and data structures that we have already seen in order to get a nice fast solution. For example, we could start by building a Voronoi diagram. This can be done in $O(n \log n)$ time using the random incremental construction. We could then use Kirkpatrick's algorithm for doing point location in a planar subdivision to find the nearest neighbor.

3 Nearest Neighbors with kd-trees

It is not too hard to modify our range search algorithm for kd-trees to get a NN-search algorithm. We simply have to replace the query boxes with a ball. Here is the pseudocode for a range search.

To do NN-search, we just have to define the region R as a ball with center q and radius $|p - q|$, where p is our current best guess for the nearest neighbor.

Algorithm 1 RangeSearch(T, R)

```
// Base Case
if T is a leaf containing a point v then
  if v ∈ R then
    return v
  else
    return nil
  end if
end if
// Search left subtree
if region(T.left) ∩ R then
  S_left = Search(T.left, R)
end if
// Search right subtree
if region(T.right) ∩ R then
  S_right = Search(T.right, R)
end if
// Return all the points found
return S_left ∪ S_right
```

Notice that the search still has the same binary search tree flavor of the range search. The difference is that the range is not orthogonal and may change between calls. Rather than trying to analyze this algorithm in depth, and then try to patch it, we'll focus instead on a different paradigm for this problem. To motivate it, let's see where the kd-tree **fails**.

Consider an input in which all of the points are evenly-spaced arbitrarily close to a circle. After only a couple of iterations of the kd-tree, it becomes clear that we need to have some really skinny cells. We say that the *aspect ratio* of a cell is the ratio of its largest to smallest dimensions. If we go lucky and all of the cells in our kd-tree are perfect squares, then it is not hard to see that the situation is a little better. Now, a query circle will intersect only $O(\sqrt{n})$ cells rather than $O(n)$ cells. This might encourage us to think about dividing the space rather than dividing the points. That is, do our splits geometrically (by distances) rather than combinatorially (by counting). But first, we must define a slightly easier problem that we will focus on.

Definition 3.1. *Given n points $P \subset \mathbb{R}^d$ and a query $q \in \mathbb{R}^d$, $p \in P$ is a k-approximate nearest neighbor (k-ANN) of q if*

$$|p - q| \leq k|NN(q) - q|. \tag{2}$$

That is, the distance from p to q is at most k times the distance from q to its true nearest neighbor.

Problem 3.1 (Approximate Nearest Neighbor Search)**.** *Given n points $P \subset \mathbb{R}^d$, preprocess them into a data structure that supports queries $q \in \mathbb{R}^d$ and returns a k-approximate nearest neighbor of q.*

Algorithm 2 NNSEARCH($T, R = ball(q, |p - q|)$)

```
// Base Case
if T is a leaf containing a point v then
  if v ∈ R then
    return v
  else
    return nil
  end if
end if
// Search left subtree
if region(T.left) ∩ R then
  S_left = Search(T.left, R)
end if
// Update R
if S_left ≠ nil then
  p ← S_left
  R ← ball(q, |p - q|)
end if
// Search right subtree
if region(T.right) ∩ R then
  S_right = Search(T.right, R)
end if
if S_right ≠ nil then
  p ← S_right
end if
// Return the closest point found
return p
```

4 Quadtrees

Let's start with a data structure for points the plane. Later we will talk about how it generalizes to higher dimensions, though I suspect, many of you could figure it out without my telling you.

Definition 4.1. *A* Quadtree *is a* 4*-ary tree in which every node is a square and the* 4*-children decompose the parent into* 4 *equal squares.*

There is a simple construction for a quadtree that works as follows.

Algorithm 3 BUILDQT

Input: $P \subset \mathbb{R}^2$
while Some cell C contains more than 1 point **do**
  Split cell C.
end while

One major difference between the quadtree and the kd-tree is that the quadtree has cells which are empty. In fact, there could be a lot of empty cells. When I say "a lot", I actually mean "unboundedly many" as a function of n. This is a big scary cardinality that is even bigger and scarier than other big scary cardinalities we see in algorithm analysis, like "exponentially many".

4.1 The Spread of the Input

Let s denote the minimum distance between any pair of points of P. Let L denote the side length of the root node. Usually, we will assume $L = 1$. The spread of the input set P is defined to be $\Delta = L/s$.

Lemma 4.1. *The depth of a Quadtree is* $O(\log \Delta)$.

Proof. Pick a node v whose children have maximum depth k. This node is corresponds to a square that contains at least two input points, for otherwise we would not have split it. At each level of the tree, we half the side length, so at level $k-1$, the side length is $\frac{1}{2^{k-1}}$. Let p, q be two input points that are children of v. We can bound the distance between p and q as follows.

$$|p - q| \leq \frac{\sqrt{2}}{2^{k-1}}.$$

By definition,

$$\Delta = \frac{1}{\min_{a,b \in P} |a - b|},$$

and so,

$$\frac{1}{\Delta} = \min_{a,b \in P} |a - b| \leq |p - q| \leq \frac{\sqrt{2}}{2^{k-1}}.$$

It follows that

$$\Delta \geq \frac{2^{k-1}}{\sqrt{2}},$$

and thus,

$$k = O(\log \Delta).$$

□

4.2 Quadtrees as Tries

In keeping with the theme of this course, we can compare quadtrees to a standard *one-dimensional* data structure: the trie. Trie is pronounced "try" not "tree". Originally, they were pronounced "tree" because the name came from re**trie**val. This was all kinds of bad because among other things they are a special case of trees. Another name for a trie is a *prefix tree*.

Fix some alphabet $\Sigma = \{a, b, c, d\}$. If we want to store a collection of strings in this alphabet, we can save space by grouping together strings with the same common prefix. Taken to the extreme, this results in a trie, which is just a tree with edges labeled with elements of Σ. The nodes of the tree store the string in the path to the root.

In the case where the alphabet has just 4 letters, it is clear that there is a connection to quadtrees. We can think of the quadtree as associating a string to each box of the quadtree.

There is actually a much tighter connection between the geometry of the quadtree and the structure of a corresponding trie. If we use $\{00, 01, 10, 11\}$ as our alphabet, the "strings" associated with each box in the quadtree are actually the interleaved bits of the coordinates of the upperleft corner of the box. This may seem completely wild and surprising at first, but we'll walk through the construction explicitly next time. For now, we just want to borrow a little intuition from tries in order to combat the problem of $\log \Delta$ depth in our search data structure.

4.3 Compressing Quadtrees

In bad examples leading to $O(\log \Delta)$ with $\log \Delta >> n$, we observe that many boxes were left empty of input points. In fact, we saw many splits which did not divide any of the points. In the case of strings, this could be interpreted as saying that many strings contain a long, common prefix. In the quadtree, it looks like a long string of nodes with outdegree 1. We saw the same problem in the history DAG when a triangle appears in many consecutive levels. We can apply the same fix by simply compressing long paths. This will guarantee that the path to the root is only $O(n)$.

Lemma 4.2. *The depth of a compressed Quadtree is at most $n - 1$.*

Proof. Consider an path from the root to a leaf square. Each time we follow an edge from v to v', the number of points in the subtree rooted at v' must be

strictly less than the number of points in the subtree rooted at v. Otherwise, we would have compressed the edge (v, v'). So, the total number of edges we can traverse from the root to a leaf is at most $n - 1$. □