

# Lecture Notes: Nearest Neighbors

Sam Westrick

s.westrick@nyu.edu

## All Nearest Neighbors

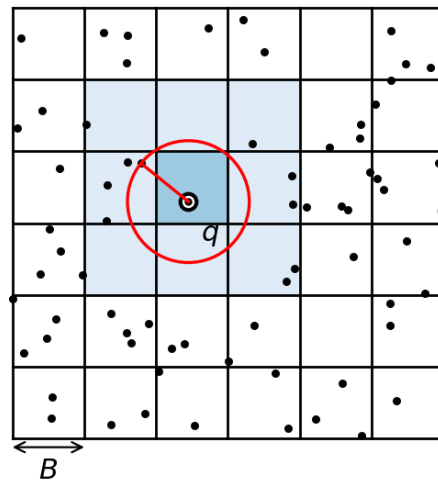
Given  $n$  (distinct) points in 2D, the **all nearest neighbors** problem is to find, for every point  $q$ , the closest other point. Checking every pair of points has plenty of parallelism, but requires  $O(n^2)$  work. We'd like to avoid comparing points that are far away from each other.

## Bucketing

A simple strategy is to divide the plane into a grid of square **buckets**, each of width  $B$ . Let  $(x_{\min}, y_{\min})$  be the bottom-left corner of the grid (the smallest  $x$  and  $y$  coordinates over all of the points). Then each point  $(x, y)$  belongs to the bucket  $(b_x, b_y) = (\lfloor (x - x_{\min})/B \rfloor, \lfloor (y - y_{\min})/B \rfloor)$ .

We number the buckets with ids  $b_x \cdot n_{by} + b_y$ , where  $n_{bx}$  and  $n_{by}$  are the number of buckets in the  $x$  and  $y$  directions, e.g.,  $n_{bx} = \lfloor (x_{\max} - x_{\min})/B \rfloor + 1$ . Sorting the points into buckets can be done with the `partition` function from last lecture (`partition: int -> ('a -> int) -> 'a seq -> 'a seq seq`), with a total of  $n_{bx} \cdot n_{by}$  output buckets. The result is a `point seq seq`, where each inner sequence holds the contents of one bucket.

```
fun bucketId (x, y) =  
  let  
    val bx = Real.floor ((x - xmin) / B)  
    val by = Real.floor ((y - ymin) / B)  
  in  
    bx * nby + by  
  end  
  
val buckets: point seq seq =  
  partition (nbx * nby) bucketId pts
```



To find the nearest neighbor of a point  $q$ , we start with the bucket containing  $q$ , and then move outwards: next we search the 8 adjacent buckets, then the next ring of 16 buckets, etc. (We can find any bucket in constant time, by computing its id.) After searching  $k$  rings, every point that we haven't looked at yet is further than  $kB$  from  $q$ . So, if the closest point found so far is at distance  $d \leq kB$ , we can stop.

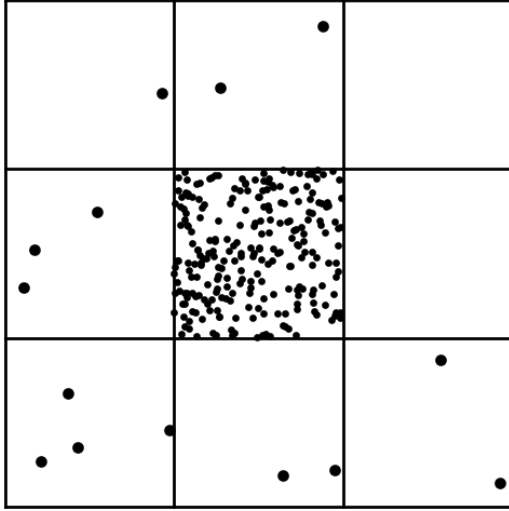
On uniformly distributed sets of points, this works reasonably well: we can choose  $B$  so that on average, every bucket holds just a few points. Queries then take  $O(1)$  time, because they only have to inspect a constant number of buckets, and altogether, we can find all nearest neighbors in only  $O(n)$  work. The queries are easily parallelized, since querying is read-only on the buckets after partitioning the points.

## An Adversarial Input

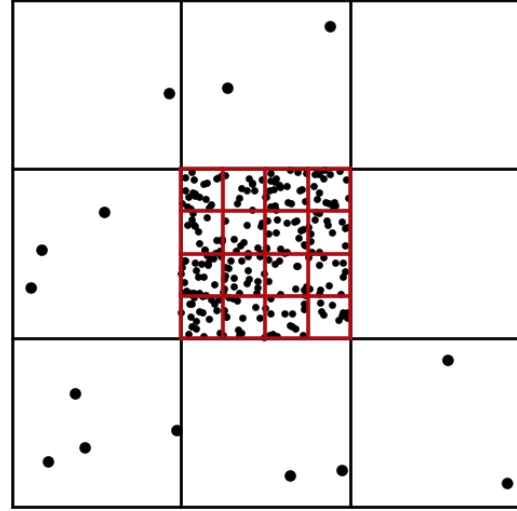
What is an adversarial input for the bucketing strategy? We can put most of the points into a single bucket (as depicted below, on the left). Then, every search for a point in that bucket has to look

at most of the points, and we're back to  $O(n^2)$  work. We can't fix this by just making  $B$  smaller everywhere: we would need a huge number of buckets, and most of them would be empty.

Instead, we can take the heavy bucket and partition it further, into smaller buckets (below, right).



Most of the points in one bucket



The heavy bucket, partitioned further

## Quadtrees

In the above example, we picked a heavy bucket and partitioned it into smaller buckets. It could be necessary to do this recursively: one of the inner buckets might still be too heavy, and might need to be partitioned itself, etc.

What is this data structure? It's not a sequence of buckets with an obvious strategy for numbering buckets. Different datasets will require different recursive partitionings.

Essentially, we have a tree: each bucket is either a sequence of inner buckets, or just a sequence of points. We can make our lives a bit simpler by placing a limit on the number of children (because a larger number of buckets can be accomplished by recursively partitioning). A natural choice in 2 dimensions is 4 children, commonly known as a **quadtree**.

A quadtree recursively divides a square (a “box”) into four equal quadrants, until each box contains only a few points. To build one, we **partition** the points into quadrants (with  $K = 4$  keys), and then recursively build each quadrant in parallel.

```

type point = real * real
type box = point * real    (* lower-left corner, and width *)

datatype tree =
  Leaf of point seq
| Node of tree seq    (* 4 children, one per quadrant *)

(* 0: top-left, 1: top-right, 2: bottom-left, 3: bottom-right *)
fun quadrant ((x, y), w) (px, py) =
  (if py >= y + w/2.0 then 0 else 2) + (if px >= x + w/2.0 then 1 else 0)

(* the box for quadrant i *)
fun subbox ((x, y), w) i =

```

```

let val h = w/2.0
in ((if i mod 2 = 0 then x else x + h, if i < 2 then y + h else y), h)
end

val leafSize = 16

fun build (b: box) (pts: point seq) : tree =
  if Seq.length pts <= leafSize then Leaf pts else
  let
    val buckets = partition 4 (quadrant b) pts
  in
    Node (Parallel.tabulate (0, 4) (fn i =>
      build (subbox b i) (Seq.nth buckets i)))
  end
end

```

Initially, we call `build` with a box  $b_0$  that contains all of the points (found with a few `reduces` over the coordinates). The tree doesn't need to store the boxes; we can recompute them with `subbox`.

## Searching for a Nearest Neighbor

To find the nearest neighbor of a point  $q$ , we search the tree while keeping track of the closest point found so far, at distance  $d$ . The key idea is **pruning**: if the box of a subtree is further than  $d$  from  $q$ , then nothing in that subtree can beat the current best, so we can skip it entirely. To make  $d$  small quickly (so that we can prune more), we first search the quadrant that contains  $q$ , and then the other three.

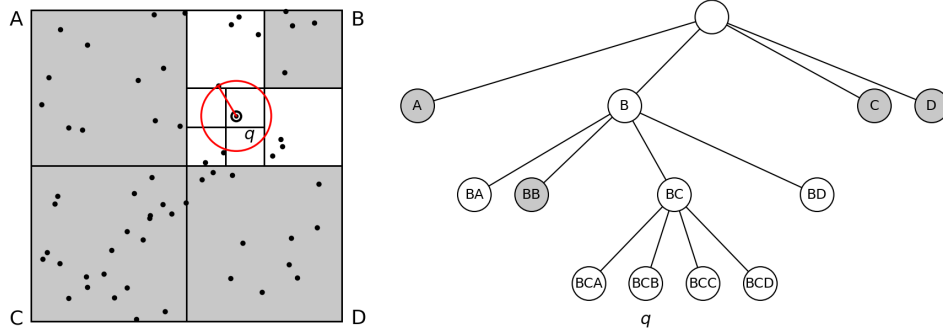


Figure 1: Searching for the nearest neighbor of  $q$ . The circle has radius  $d$ , the distance to the nearest neighbor, and only boxes that intersect it need to be searched. The gray boxes are pruned: their subtrees are never explored.

```

(* distance from p to the closest point in box b *)
fun distToBox ((x, y), w) (px, py) =
  let fun clamp (lo, hi) v = Real.max (lo, Real.min (hi, v))
  in dist ((px, py), (clamp (x, x + w) px, clamp (y, y + w) py))
  end

(* nearest neighbor of q (other than q itself) in tree t with box b *)
fun nearest (b: box, t: tree) (q: point) : point =
  let
    (* best = (closest point found so far, its distance from q) *)
    fun search (best as (_, d)) (b, t) =

```

```

if distToBox b q > d then best else
case t of
  Leaf pts =>
    Seq.iterate (fn (best as (_, d), p) =>
      let val d' = dist (p, q)
      in if 0.0 < d' andalso d' < d then (p, d') else best
    end) best pts
  | Node children =>
    let
      fun child i = (subbox b i, Seq.nth children i)
      val first = quadrant b q
      fun visit (i, best) =
        if i = first then best else search best (child i)
    in
      List.foldl visit (search best (child first)) [0, 1, 2, 3]
    end
    val (p, _) = search (q, Real.posInf) (b, t)
in
  p
end

```

Here, `dist` is the usual Euclidean distance. Each search is sequential; the parallelism comes from doing all  $n$  searches at the same time, with `Seq.map (nearest (b0, t)) pts`.

## Cost Analysis

**Building.** At each level of the tree, the calls to `partition` operate on disjoint subsets of the points, for  $O(n)$  work and  $O(\log n)$  span per level. For “nice” inputs (e.g., uniformly distributed points), there are  $O(\log n)$  levels, giving  $O(n \log n)$  work and  $O(\log^2 n)$  span. But be careful: the depth depends on the distribution, not just on  $n$ . A tight cluster of points requires many levels to separate.

**Searching.** On “nice” inputs, a search typically walks down to the leaf containing  $q$ , and then only checks a few neighboring boxes:  $O(\log n)$  work, for  $O(n \log n)$  total work and  $O(\log n)$  span. There is no good worst-case guarantee (the circle around  $q$  could intersect many boxes), but in practice, quadtrees are extremely effective, even for somewhat lopsided or uneven distributions of points.

(**Note:** there are strategies that can guarantee  $O(n \log n)$  work and  $O(\text{polylog}(n))$  span for the all-nearest-neighbor problem, even for adversarial inputs, but they are a bit more complex and not as widely used in practice.)