CHAPTER 17

Approximate Nearest Neighbor Search in Low Dimension

17.1. Introduction

Let P be a set of n points in $\mathbb{R}^{\mathsf{d}}$. We would like to preprocess it, such that given a query point q, one can determine the closest point in P to q quickly. Unfortunately, the exact problem seems to require prohibitive preprocessing time. (Namely, it requires computing the Voronoi diagram of P and preprocessing it for point-location queries. This requires (roughly) $O(n^{\lceil \mathsf{d}/2 \rceil})$ time.)

Instead, we will specify a parameter $\varepsilon > 0$ and build a data-structure that answers ***$(1+\varepsilon)$-approximate nearest neighbor*** queries.

Definition 17.1. For a set $\mathsf{P} \subseteq \mathbb{R}^{\mathsf{d}}$ and a query point q, we denote by $\mathsf{nn}(\mathsf{q}) = \mathsf{nn}(\mathsf{q}, \mathsf{P})$ the ***closest point*** (i.e., ***nearest neighbor***) in P to q. We denote by $\mathbf{d}(\mathsf{q}, \mathsf{P})$ the distances between q and its closest point in P; that is, $\mathbf{d}(\mathsf{q}, \mathsf{P}) = \|\mathsf{q} - \mathsf{nn}(\mathsf{q})\|$.

For a query point q and a set P of n points in $\mathbb{R}^{\mathsf{d}}$, a point $\mathsf{s} \in \mathsf{P}$ is a ***$(1+\varepsilon)$-approximate nearest neighbor*** (or just ***$(1+\varepsilon)$-ANN***) if $\|\mathsf{q} - \mathsf{s}\| \leq (1+\varepsilon)\mathbf{d}(\mathsf{q}, \mathsf{P})$. Alternatively, for any $\mathsf{t} \in \mathsf{P}$, we have $\|\mathsf{q} - \mathsf{s}\| \leq (1+\varepsilon)\,\|\mathsf{q} - \mathsf{t}\|$.

This is yet another instance where solving the bounded spread case is relatively easy. (We remind the reader that the ***spread*** of a point set P, denoted by $\Phi(\mathsf{P})$, is the ratio between the diameter and the distance of the closest pair of P.)

17.2. The bounded spread case

Let P be a set of n points contained inside the unit hypercube in $\mathbb{R}^{\mathsf{d}}$, and let $\mathcal{T}$ be a quadtree of P, where $\mathrm{diam}(\mathsf{P}) = \Omega(1)$. We assume that with each (internal) node u of $\mathcal{T}$ there is an associated representative point, rep_u, such that rep_u is one of the points of P stored in the subtree rooted at u.

Let q be a query point and let $\varepsilon > 0$ be a parameter. Here our purpose is to find a $(1+\varepsilon)$-ANN to q in P.

Idea of the algorithm. The algorithm would maintain a set of nodes of $\mathcal{T}$ that might contain the ANN to the query point. Each such node has a representative point associated with it, and we compute its distance to the query point and maintain the nearest neighbor found so far. At each stage, the search would be refined by replacing a node by its children (and computing the distance from the query point to all the new representatives of these new nodes).

The key observation is that we need to continue searching in a subtree of a node only if the node can contain a point that is significantly closer to the query point than the current best candidate found.

We keep only the "promising" nodes and continue this search till there are no more candidates to check. We claim that we had found the ANN to q, and furthermore the query time is fast. This idea is depicted in Figure 17.1.

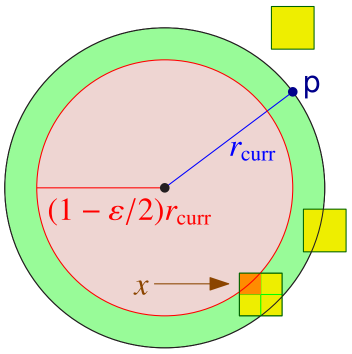


FIGURE 17.1. Out of the three nodes currently in the candidate set, only one of them (i.e., x) has the potential to contain a better ANN to q than the current one (i.e., p).

Formally, let p be the current best candidate found so far, and let its distance from q be r_{curr}. Now, consider a node w in $\mathcal{T}$, and observe that the point set P_w, stored in the subtree of w, might contain a "significantly" nearer neighbor to q only if it contains a point $s \in rg_w \cap P$ such that $\|q - s\| < (1-\varepsilon/2)r_{\text{curr}}$. A conservative lower bound on the distance of any point in rg_w to q is $\|q - rep_w\| - \text{diam}(\square_w)$. In particular, if $\|q - rep_w\| - \text{diam}(\square_w) > (1-\varepsilon/2)r_{\text{curr}}$, then we can abort the search for ANN in the subtree of w.

The algorithm. Let $A_0 = \{\text{root}(T)\}$, and let $r_{\text{curr}} = \|q - rep_{\text{root}(T)}\|$. The value of r_{curr} is the distance to the closest neighbor of q that was found so far by the algorithm.

In the ith iteration, for $i > 0$, the algorithm expands the nodes of A_{i-1} to get A_i. Formally, for $v \in A_{i-1}$, let C_v be the set of children of v in $\mathcal{T}$ and let $\square_v$ denote the cell (i.e., region) that v corresponds to. For every node $w \in C_v$, we compute

$$r_{\text{curr}} \leftarrow \min\left(r_{\text{curr}}, \|q - rep_w\|\right).$$

The algorithm checks if

$$\|q - rep_w\| - \text{diam}(\square_w) < (1-\varepsilon/2)r_{\text{curr}}, \tag{17.1}$$

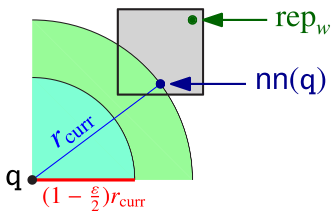
and if so, it adds w to A_i. The algorithm continues in this expansion process till all the elements of A_{i-1} are considered, and then it moves to the next iteration. The algorithm stops when the generated set A_i is empty. The algorithm returns the point realizing the value of r_{curr} as the ANN.

The set A_i is a set of nodes of depth i in the quadtree that the algorithm visits. Note that all these nodes belong to the canonical grid $G_{2^{-i}}$ of level $-i$, where every canonical square has sidelength 2^{-i}. (Thus, nodes of depth i in the quadtree are of ***level*** $-i$. This is somewhat confusing but it in fact makes the presentation simpler.)

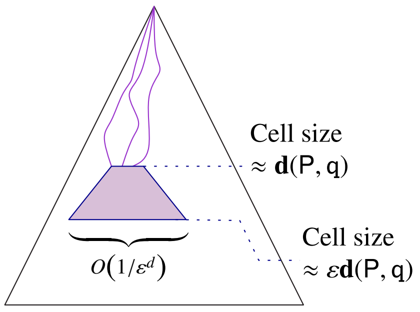
Correctness. Note that the algorithm adds a node w to A_i only if the set P_w might contain points which are closer to q than the (best) current nearest neighbor the algorithm found, where P_w is the set of points stored in the subtree of w. (More precisely, P_w might contain a point which is $1-\varepsilon/2$ closer to q than any point encountered so far.)

Consider the last node w inspected by the algorithm such that $nn(q) \in P_w$. Since the algorithm decided to throw this node away, we have, by the triangle inequality, that

$$\begin{aligned}\|q - nn(q)\| &\geq \|q - rep_w\| - \|rep_w - nn(q)\| \\ &\geq \|q - rep_w\| - \text{diam}(\square_w) \geq (1-\varepsilon/2)r_{\text{curr}}.\end{aligned}$$



Thus, $\|q - nn(q)\| / (1 - \varepsilon/2) \geq r_{curr}$. However, $1/(1 - \varepsilon/2) \leq 1 + \varepsilon$, for $1 \geq \varepsilon > 0$, as can be easily verified. Thus, $r_{curr} \leq (1 + \varepsilon)\mathbf{d}(q, P)$, and the algorithm returns $(1 + \varepsilon)$-ANN to q.


Cell size
$\approx \mathbf{d}(P, q)$
Cell size
$\approx \varepsilon\mathbf{d}(P, q)$
$O(1/\varepsilon^d)$

Running time analysis. Before barging into a formal proof of the running time of the above search procedure, it is useful to visualize the execution of the algorithm. It visits the quadtree level by level. As long as the level's grid cells are bigger than the ANN distance $r = \mathbf{d}(q, P)$, the number of nodes visited is a constant (i.e., $|A_i| = O(1)$). (This is not obvious, and at this stage the reader should take this statement with due skepticism.) This number "explodes" only when the cell size become smaller than r, but then the search stops when we reach grid size $O(\varepsilon r)$. In particular, since the number of grid cells visited (in the second stage) grows exponentially with the level, we can use the number of nodes visited in the bottom level (i.e., $O(1/\varepsilon^d)$) to bound the query running time for this part of the query.

Lemma 17.2. *Let* P *be a set of* n *points contained inside the unit hypercube in* $\mathbb{R}^d$*, and let* $\mathcal{T}$ *be a quadtree of* P*, where* $\text{diam}(P) = \Omega(1)$*. Let* q *be a query point, and let* $\varepsilon > 0$ *be a parameter. A* $(1 + \varepsilon)$*-ANN to* q *can be computed in* $O\left(\varepsilon^{-d} + \log(1/\varpi)\right)$ *time, where* $\varpi = \mathbf{d}(q, P)$.

Proof. The algorithm is described above. We are only left with the task of bounding the query time. Observe that if a node $w \in \mathcal{T}$ is considered by the algorithm and $\text{diam}(\Box_w) < (\varepsilon/4)\varpi$, then

$$\left\|q - \text{rep}_w\right\| - \text{diam}(\Box_w) \geq \left\|q - \text{rep}_w\right\| - (\varepsilon/4)\varpi \geq r_{curr} - (\varepsilon/4)r_{curr} \geq (1 - \varepsilon/4)r_{curr},$$

which implies that neither w nor any of its children would be inserted into the sets $A_1, \ldots, A_m$, where m is the depth $\mathcal{T}$, by (17.1). Thus, no nodes of depth $\geq h = \lceil -\lg(\varpi\varepsilon/4)\rceil$ are being considered by the algorithm.

Consider the node u of $\mathcal{T}$ of depth i containing nn(q, P). Clearly, the distance between q and rep_u is at most $\ell_i = \varpi + \text{diam}_u = \varpi + \sqrt{d}2^{-i}$. As such, in the end of the ith iteration, we have $r_{curr} \leq \ell_i$, since the algorithm had inspected u.

Thus, the only cells of $G_{2^{-i-1}}$ that might be considered by the algorithm are the ones in distance $\leq \ell_i$ from q. Indeed, in the end of the ith iteration, $r_{curr} \leq \ell_i$. As such, any node of $G_{2^{-i-1}}$ (i.e., nodes considered in the $(i + 1)$st iteration of the algorithm) that is within a distance larger than r_{curr} from q cannot improve the distance to the current nearest neighbor and can just be thrown away if it is in the queue. (We charge the operation of putting a node into the queue to its parent. As such, nodes that get inserted and deleted in the next iteration are paid for by their parents.)

The number of such relevant cells (i.e., cells that the algorithm dequeues and do not get immediately thrown out) is the number of grid cells of $G_{2^{-i-1}}$ that intersect a box of sidelength $2\ell_i$ centered at q; that is,

$$n_i = \left(2\left\lceil \frac{\ell_i}{2^{-i-1}} \right\rceil\right)^d = O\left(\left(1 + \frac{\varpi + \sqrt{d}2^{-i}}{2^{-i-1}}\right)^d\right) = O\left(\left(1 + \frac{\varpi}{2^{-i-1}}\right)^d\right) = O\left(1 + \left(2^i\varpi\right)^d\right),$$

since for any $a, b \geq 0$ we have $(a+b)^{\mathsf{d}} \leq (2\max(a,b))^{\mathsf{d}} \leq 2^{\mathsf{d}}\left(a^{\mathsf{d}} + b^{\mathsf{d}}\right)$. Thus, the total number of nodes visited is

$$\sum_{i=0}^{h} n_i = O\left(\sum_{i=0}^{\lceil -\lg(\varpi\varepsilon/4)\rceil} \left(1 + \left(2^i \varpi\right)^{\mathsf{d}}\right)\right) = O\left(\lg\frac{1}{\varpi\varepsilon} + \left(\frac{\varpi}{\varpi\varepsilon/4}\right)^{\mathsf{d}}\right) = O\left(\log\frac{1}{\varpi} + \frac{1}{\varepsilon^{\mathsf{d}}}\right),$$

and this also bounds the overall query time. ■

One can apply Lemma 17.2 to the case for which the input has spread bounded from above. Indeed, if the distance between the closest pair of points of P is $\mu = \mathcal{CP}(\mathsf{P})$, then the algorithm would never search in cells that have diameter $\leq \mu/8$. Indeed, no such nodes would exist in the quadtree, to begin with, since the parent node of such a node would contain only a single point of the input. As such, we can replace ϖ by μ in the above argumentation.

Lemma 17.3. *Let P be a set of n points in $\mathbb{R}^{\mathsf{d}}$, and let $\mathcal{T}$ be a quadtree of P, where* $\mathrm{diam}(\mathsf{P}) = \Omega(1)$. *Given a query point q and $1 \geq \varepsilon > 0$, one can return a $(1+\varepsilon)$-ANN to q in $O\left(1/\varepsilon^{\mathsf{d}} + \log\Phi(\mathsf{P})\right)$ time, where $\Phi(\mathsf{P})$ is the spread of P.*

A less trivial task is to adapt the algorithm, so that it uses compressed quadtrees. To this end, the algorithm would still handle the nodes by levels. This requires us to keep a heap of integers in the range $0, -1, \ldots, -\lfloor\lg\Phi(\mathsf{P})\rfloor$. This can be easily done by maintaining an array of size $O(\log\Phi(\mathsf{P}))$, where each array cell maintains a linked list of all nodes with this level. Clearly, an insertion/deletion into this heap data-structure can be handled in constant time by augmenting it with a hash table. Thus, the above algorithm would work for this case after modifying it to use this "level" heap instead of just the sets A_i.

Theorem 17.4. *Let P be a set of n points in $\mathbb{R}^{\mathsf{d}}$. One can preprocess P in $O(n\log n)$ time and using linear space, such that given a query point q and parameter $1 \geq \varepsilon > 0$, one can return a $(1+\varepsilon)$-ANN to q in $O\left(1/\varepsilon^{\mathsf{d}} + \log\Phi(\mathsf{P})\right)$ time. In fact, the query time is $O(1/\varepsilon^{\mathsf{d}} + \log(\mathrm{diam}(\mathsf{P})/\varpi))$, where $\varpi = \mathbf{d}(\mathsf{q}, \mathsf{P})$.*

17.3. ANN – the unbounded general case

The snark and the unbounded spread case (or a metaphilosophical pretentious discussion that the reader might want to skip). (The reader might consider this to be a footnote to a footnote, which finds itself inside the text because of lack of space at the bottom of the page.) We have a data-structure that supports insertions, deletions, and approximate nearest neighbor reasonably quickly. The running time for such operations is roughly $O(\log\Phi(\mathsf{P}))$ (ignoring additive terms in $1/\varepsilon$). Since the spread of P in most real world applications is going to be bounded by a constant degree polynomial in n, it seems this is sufficient for our purposes, and we should stop now, while we are ahead in the game. But the nagging question remains: If the spread of P is not bounded by something reasonable, what can be done?

The rule of thumb is that $\Phi(\mathsf{P})$ can be replaced by n (for this problem, but also in a lot of other problems). This usually requires some additional machinery, and sometimes this machinery is quite sophisticated and complicated. At times, the search for the ultimate algorithm that can work for such "strange" inputs looks like the hunting of the snark [**Car76**] – a futile waste of energy looking for some imaginary top-of-the-mountain, which has no practical importance.

Solving the bounded spread case can be acceptable in many situations, and it is the first stop in trying to solve the general case. However, solving the general case provides us with more insight into the problem and in some cases leads to more efficient solutions than the bounded spread case.

With this caveat emptor[1] warning duly given, we plunge ahead into solving the ANN for the unbounded spread case.

Plan of attack. To answer the ANN query in the general case, we will first get a fast rough approximation. Next, using a compressed quadtree, we will find a constant number of relevant nodes and apply Theorem 17.4 to those nodes. This will yield the required approximation. Before solving this problem, we need a minor extension of the compressed quadtree data-structure.

Extending a compressed quadtree to support cell queries. Let $\widehat{\square}$ be a canonical grid cell (we remind the reader that this is a cell of the grid G_{2^i}, for some integer $i \leq 0$). Given a compressed quadtree $\widehat{T}$, we would like to find the *single* node $v \in \widehat{T}$, such that $\mathsf{P} \cap \widehat{\square} = \mathsf{P}_v$. (Note that the node v might be compressed, and the square associated with it might be much larger than $\widehat{\square}$, but its only child w is such that $\square_w \subseteq \widehat{\square} \subseteq \square_v$. However, the annulus $\square_v \subseteq \square_w$ contains no input point.) We will refer to such a query as a ***cell query***.

It is not hard to see that the quadtree data-structure can be modified to support cell queries in logarithmic time (it's a glorified point-location query), and we omit the easy but tedious details. See Exercise 2.4_{p26}.

LEMMA 17.5. *One can perform a cell query in a compressed quadtree $\widehat{T}$, in $O(\log n)$ time, where n is the size of $\widehat{T}$. Namely, given a query canonical cell $\widehat{\square}$, one can find, in $O(\log n)$ time, the node $w \in \widehat{T}$ such that $\square_w \subseteq \widehat{\square}$ and $\mathsf{P} \cap \widehat{\square} = \mathsf{P}_w$.*

17.3.1. Putting things together – answering ANN queries. Let P be a set of n points in $\mathbb{R}^{\mathsf{d}}$ contained in the unit hypercube. We build the compressed quadtree $\widehat{T}$ of P, so that it supports cell queries, using Lemma 17.5. We will also need a data-structure that supports very rough ANN queries quickly. We describe one way to build such a data-structure in the next section, and in particular, we will use the following result (see Theorem 17.10). Note that a similar result can be derived by using a shifted quadtree and a simple point-location query; see Theorem 11.22_{p160}. Explicitly, we need the following (provided by Theorem 17.10_{p240}):

> Let P be a set of n points in $\mathbb{R}^{\mathsf{d}}$. One can build a data-structure $\mathcal{T}_R$, in $O(n \log n)$ time, such that given a query point $\mathsf{q} \in \mathbb{R}^{\mathsf{d}}$, one can return a $(1 + 4n)$-ANN of q in P in $O(\log n)$ time.

Given a query point q, using $\mathcal{T}_R$, we compute a point $u \in \mathsf{P}$, such that $\mathbf{d}(\mathsf{q}, \mathsf{P}) \leq \|u - \mathsf{q}\| \leq (1+4n)\mathbf{d}(\mathsf{q}, \mathsf{P})$. Let $R = \|u - \mathsf{q}\|$ and $r = \|u - \mathsf{q}\| / (4n+1)$. Clearly, $r \leq \mathbf{d}(\mathsf{q}, \mathsf{P}) \leq R$. Next, compute $\mathbb{L} = \lceil \lg R \rceil$, and let C be the set of cells of $\mathsf{G}_{2^{\mathbb{L}}}$ that are within a distance $\leq R$ from q; that is, C is the set of grid cells of $\mathsf{G}_{2^{\mathbb{L}}}$ whose union (completely) covers $\mathbf{b}(\mathsf{q}, R)$. Clearly, $\mathsf{nn}(\mathsf{q}, \mathsf{P}) \in \mathbf{b}(\mathsf{q}, R) \subseteq \bigcup_{\square \in C} \square$. Next, for each cell $\square \in C$, we compute the node $v \in \widehat{T}$ such that $\mathsf{P} \cap \square = \mathsf{P}_v$, using a cell query (i.e., Lemma 17.5). (Note that if $\square$ does not contain any point of P, this query would return a leaf or a compressed node whose region contains $\square$, and it might contain at most one point of P.) Let V be the resulting set of nodes of $\widehat{T}$.

For each node of $v \in V$, we now apply the algorithm of Theorem 17.4 to the compressed quadtree rooted at v. We return the nearest neighbor found.

[1]Buyer beware in Latin.

Since $|V| = O(1)$ and $\mathrm{diam}(\mathsf{P}_v) = O(R)$, for all $v \in V$, the query time is

$$\sum_{v \in V} O\left(\frac{1}{\varepsilon^{\mathsf{d}}} + \log \frac{\mathrm{diam}(\mathsf{P}_v)}{r}\right) = O\left(\frac{1}{\varepsilon^{\mathsf{d}}} + \sum_{v \in V} \log \frac{\mathrm{diam}(\mathsf{P}_v)}{r}\right) = O\left(\frac{1}{\varepsilon^{\mathsf{d}}} + \sum_{v \in V} \log \frac{R}{r}\right)$$
$$= O\left(\frac{1}{\varepsilon^{\mathsf{d}}} + \log n\right).$$

As for the correctness, observe that there is a node $w \in V$, such that $\mathsf{nn}(\mathsf{q}, \mathsf{P}) \in \mathsf{P}_w$. As such, when we apply the algorithm of Theorem 17.4 to w, it would return us a $(1+\varepsilon)$-ANN to q.

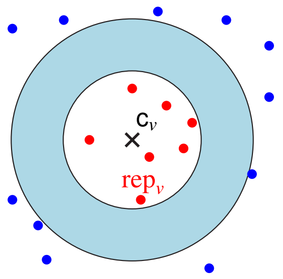
Theorem 17.6. *Let* P *be a set of* n *points in* $\mathbb{R}^{\mathsf{d}}$. *One can construct a data-structure of linear size, in* $O(n \log n)$ *time, such that given a query point* $\mathsf{q} \in \mathbb{R}^{\mathsf{d}}$ *and a parameter* $1 \geq \varepsilon > 0$, *one can compute a* $(1+\varepsilon)$*-ANN to* q *in* $O(1/\varepsilon^{\mathsf{d}} + \log n)$ *time.*

17.4. Low quality ANN search via the ring separator tree

To perform ANN in the unbounded spread case, all we need is a rough approximation (i.e., polynomial factor in n) to the distance to the nearest neighbor (note that we need only the distance). One way to achieve that was described in Theorem 11.22_{p160}, and we present another alternative construction that uses a more direct argument.

Definition 17.7. A binary tree $\mathcal{T}$ having the points of P as leaves is a ***t*-ring tree** for P if every node $v \in T$ is associated with a ring (hopefully "thick"), such that the ring separates the points into two sets (hopefully both relatively large) and the interior of the ring is empty of any point of P.

For a node v of $\mathcal{T}$, let P_v denote the subset of points of P stored in the subtree of v, and let rep_v be a point stored in v. We require that for any node v of $\mathcal{T}$, there is an associated ball $\mathbf{b}_v = \mathbf{b}(\mathsf{c}_v, r_v)$, such that all the points of $\mathsf{P}^v_{\text{in}} = \mathsf{P}_v \cap \mathbf{b}_v$ are in one child of $\mathcal{T}$. Furthermore, all the other points of P_v are outside the interior of the enlarged ball $\mathbf{b}(\mathsf{c}_v, (1+t)r_v)$ and are stored in the other child of v. (Note that c_v might not be a point of P.)



We will also store an arbitrary representative point $\mathrm{rep}_v \in \mathsf{P}^v_{\text{in}}$ in v (rep_v is not necessarily c_v).

To see what the above definition implies, consider a t-ring tree $\mathcal{T}$. For any node $v \in \mathcal{T}$, the interior of the ring associated with v (i.e., $\mathbf{b}(\mathsf{c}_v, (1+t)r_v) \setminus \mathbf{b}(\mathsf{c}_v, r_v)$) is empty of any point of P. Intuitively, the bigger t is, the better $\mathcal{T}$ clusters P. Furthermore, every internal node v of $\mathcal{T}$ has the following quantities associated with it:

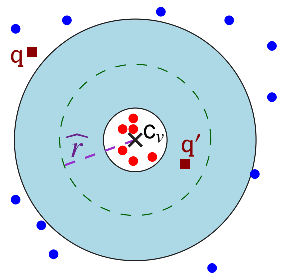
(i) r_v: radius of the inner ball of the ring of v,
(ii) c_v: center of the ring of v (the point c_v is not necessarily in P), and
(iii) $\mathrm{rep}_v \in \mathsf{P}^v_{\text{in}}$: a representative from the point set stored in the inner ball of v (note that rep_v might be distinct from c_v).

The ANN search procedure. Let q denote the query point. Initially, set v to be the root of $\mathcal{T}$ and $r_{\text{curr}} \leftarrow \infty$. The algorithm answers the ANN query by traversing down $\mathcal{T}$.

During the traversal, we first compute the distance $l = \left\|\mathsf{q} - \mathrm{rep}_v\right\|$. If this is smaller than r_{curr} (the distance to the current nearest neighbor found), then we update r_{curr} (and store the point realizing the new value of r_{curr}).

If $\|\mathsf{q} - \mathsf{c}_v\| \leq \widehat{r}$, we continue the search recursively in the child containing P^v_{in}, where $\widehat{r} = (1+t/2)r_v$ is the "middle" radius of the ring centered at c_v. Otherwise, we continue the search in the subtree containing $\mathsf{P}^v_{\mathrm{out}}$. The algorithm stops when reaching a leaf of $\mathcal{T}$ and returns the point realizing r_{curr} as the approximate nearest neighbor.

Intuition. If the query point q is outside the outer ball of a node v, it is so far from the points inside the inner ball (i.e., $\mathsf{P}^v_{\mathrm{in}}$), and we can treat all of them as a single point (i.e., rep_v). On the other hand, if the query point q' is inside the inner ball, then it must have a neighbor nearby (i.e., a point of $\mathsf{P}^v_{\mathrm{in}}$), and all the points of $\mathsf{P}^v_{\mathrm{out}}$ are far enough away that they can be ignored. Naturally, if the query point falls inside the ring, the same argumentation works (with slightly worst constants), using the middle radius as the splitting boundary in the search. See the figure on the right.



Lemma 17.8. *Given a t-ring tree $\mathcal{T}$, one can answer $(1 + 4/t)$-approximate nearest neighbor queries, in $O(\mathtt{depth}(\mathcal{T}))$ time.*

Proof. Clearly, the query time is $O(\mathtt{depth}(\mathcal{T}))$. As for the quality of approximation, let π denote the generated search path in $\mathcal{T}$ and let $\mathsf{nn}(\mathsf{q})$ denote the nearest neighbor to q in P. Furthermore, let w denote the last node in the search path π, such that $\mathsf{nn}(\mathsf{q}) \in \mathsf{P}_w$. Clearly, if $\mathsf{nn}(\mathsf{q}) \in \mathsf{P}^w_{\mathrm{in}}$ but we continued the search in $\mathsf{P}^w_{\mathrm{out}}$, then q is outside the middle sphere (i.e., $\|\mathsf{q} - \mathsf{c}_w\| \geq (1 + t/2)r_w$) and $\|\mathsf{q} - \mathsf{nn}(\mathsf{q})\| \geq (t/2)r_w$ (since this is the distance between the middle sphere and the inner sphere). Thus,

$$\left\|\mathsf{q} - \mathrm{rep}_w\right\| \leq \|\mathsf{q} - \mathsf{nn}(\mathsf{q})\| + \left\|\mathsf{nn}(\mathsf{q}) - \mathrm{rep}_w\right\| \leq \|\mathsf{q} - \mathsf{nn}(\mathsf{q})\| + 2r_w,$$

since $\mathsf{nn}(\mathsf{q}), \mathrm{rep}_w \in \mathbf{b}_w = \mathbf{b}(\mathsf{c}_w, r_w)$. In particular,

$$\frac{\left\|\mathsf{q} - \mathrm{rep}_w\right\|}{\|\mathsf{q} - \mathsf{nn}(\mathsf{q})\|} \leq \frac{\|\mathsf{q} - \mathsf{nn}(\mathsf{q})\| + 2r_w}{\|\mathsf{q} - \mathsf{nn}(\mathsf{q})\|} \leq 1 + \frac{2r_w}{\|\mathsf{q} - \mathsf{nn}(\mathsf{q})\|} \leq 1 + \frac{2r_w}{(t/2)r_w} = 1 + \frac{4}{t}.$$

Namely, rep_w is a $(1 + 4/t)$-approximate nearest neighbor to q.

Similarly, if $\mathsf{nn}(\mathsf{q}) \in \mathsf{P}^w_{\mathrm{out}}$ but we continued the search in $\mathsf{P}^w_{\mathrm{in}}$, then (i) $\|\mathsf{q} - \mathsf{nn}(\mathsf{q})\| \geq (t/2)r_w$, (ii) $\|\mathsf{q} - \mathsf{c}_w\| \leq (1 + t/2)r_w$, and (iii) $\left\|\mathsf{q} - \mathrm{rep}_w\right\| \leq \|\mathsf{q} - \mathsf{c}_w\| + \left\|\mathsf{c}_w - \mathrm{rep}_w\right\| \leq (t/2 + 2)r_w$. As such, we have that

$$\frac{\left\|\mathsf{q} - \mathrm{rep}_w\right\|}{\|\mathsf{q} - \mathsf{nn}(\mathsf{q})\|} \leq \frac{(t/2 + 2)r_w}{(t/2)r_w} \leq 1 + \frac{4}{t}.$$

Since rep_w is considered as one of the candidates to be nearest neighbor in the search, this implies that the algorithm returns a $(1 + 4/t)$-ANN. ■

In low dimensions, there is always a good separating ring (see Lemma 3.28$_{\mathrm{p40}}$), and we use it in the following construction.

Lemma 17.9. *Given a set P of n points in $\mathbb{R}^{\mathsf{d}}$, one can compute a $(1/n)$-ring tree of P in $O(n \log n)$ time.*

Proof. The construction is recursive. Setting $t = n$, by Lemma 3.28$_{\mathrm{p40}}$, one can compute, in linear time, a ball $\mathbf{b}(\mathsf{p}, r)$, such that (i) $|\mathbf{b}(\mathsf{p}, r) \cap \mathsf{P}| \geq n/c$, (ii) the ring $\mathbf{b}(\mathsf{p}, r(1 + 1/t)) \setminus \mathbf{b}(\mathsf{p}, r)$ contains no point of P, and (iii) $|\mathsf{P} \setminus \mathbf{b}(\mathsf{p}, r(1 + 1/t))| \geq n/2$.

Let v be the root of the new tree, set $\mathsf{P}^v_{\mathrm{in}}$ to be $\mathsf{P} \cap \mathbf{b}(\mathsf{p}, r)$ and $\mathsf{P}^v_{\mathrm{out}} = \mathsf{P} \setminus \mathsf{P}^v_{\mathrm{in}}$, and store $\mathbf{b}_v = \mathbf{b}(\mathsf{p}, r')$ and $\mathrm{rep}_v = \mathsf{p}$. Continue the construction recursively on those two sets. Observe that $\left|\mathsf{P}^v_{\mathrm{in}}\right|, \left|\mathsf{P}^v_{\mathrm{out}}\right| \geq n/c$, where c is a constant. It follows that the construction time of the algorithm is $T(n) = O(n) + T\left(\left|\mathsf{P}^v_{\mathrm{in}}\right|\right) + T\left(\left|\mathsf{P}^v_{\mathrm{out}}\right|\right) = O(n \log n)$, and the depth of the resulting tree is $O(\log n)$. ■

Combining the above two lemmas, we get the following result.

Theorem 17.10. *Let* P *be a set of n points in* $\mathbb{R}^{\mathsf{d}}$*. One can preprocess it in* $O(n \log n)$ *time, such that given a query point* $\mathsf{q} \in \mathbb{R}^{\mathsf{d}}$*, one can return a* $(1 + 4n)$*-ANN of* q *in* P *in* $O(\log n)$ *time.*

17.5. Bibliographical notes

The presentation of the ring tree follows the recent work of Har-Peled and Mendel [**HM06**]. Ring trees are probably an old idea. A more elaborate but similar data-structure is described by Indyk and Motwani [**IM98**]. Of course, the property that "thick" ring separators exist is inherently low dimensional, as the regular simplex in n dimensions demonstrates. One option to fix this is to allow the rings to contain points and to replicate the points inside the ring in both subtrees. As such, the size of the resulting tree is not necessarily linear. However, careful implementation yields linear (or small) size; see Exercise 17.1 for more details. This and several additional ideas are used in the construction of the cover tree of Indyk and Motwani [**IM98**].

Section 17.2 is a simplification of the work of Arya et al. [**AMN+98**]. Section 17.3 is also inspired to a certain extent by Arya et al., although it is essentially a simplification of Har-Peled and Mendel [**HM06**] data-structure to the case of compressed quadtrees. In particular, we believe that the data-structure presented is conceptually simpler than in previously published work.

There is a huge amount of literature on approximate nearest neighbor search, both in low and high dimensions in the theory, learning and database communities. The reason for this lies in the importance of this problem on special input distributions encountered in practice, different computation models (i.e., I/O-efficient algorithms), search in high dimensions, and practical efficiency.

Liner space. In low dimensions, the seminal work of Arya et al. [**AMN+98**], mentioned above, was the first to offer linear size data-structure, with logarithmic query time, such that the approximation quality is specified with the query. The query time of Arya et al. is slightly worse than the running time of Theorem 17.6, since they maintain a heap of cells, always handling the cell closest to the query point. This results in query time $O(\varepsilon^{-d} \log n)$. It can be further improved to $O(1/\varepsilon^{\mathsf{d}} \log(1/\varepsilon) + \log n)$ by observing that this heap has only very few delete-mins and many insertions. This observation is due to Duncan [**Dun99**].

Instead of having a separate ring tree, Arya et al. rebalance the compressed quadtree directly. This results in nodes which correspond to cells that have the shape of an annulus (i.e., the region formed by the difference between two canonical grid cells).

Duncan [**Dun99**] and some other authors offered a data-structure (called the ***BAR-tree***) with similar query time, but it seems to be inferior, in practice, to the work of Arya et al., for the reason that while the regions that the nodes correspond to are convex, they have higher descriptive complexity, and it is harder to compute the distance of the query point to a cell.

Faster query time. One can improve the query time if one is willing to specify ε during the construction of the data-structure, resulting in a trade-off between space and query time. In particular, Clarkson [**Cla94**] showed that one can construct a data-structure of (roughly) size $O(n/\varepsilon^{(d-1)/2})$ and query time $O(\varepsilon^{-(d-1)/2}\log n)$. Chan simplified and cleaned up this result [**Cha98**] and also presented some other results.

Details on faster query time. A set of points Q is $\sqrt{\varepsilon}$*-far* from a query point q if the $\|\mathsf{q} - c_\mathsf{Q}\| \geq \mathrm{diam}(\mathsf{Q})/\sqrt{\varepsilon}$, where c_Q is some point of Q. It is easy to verify that if we partition space around c_Q into cones with central angle $O\big(\sqrt{\varepsilon}\big)$ (this requires $O(1/\varepsilon^{(d-1)/2})$ cones), then the most extreme point of Q in such a cone ψ, furthest away from c_Q, is the $(1+\varepsilon)$-approximate nearest neighbor for any query point inside ψ which is $\sqrt{\varepsilon}$-far. Namely, we precompute the ANN inside each cone if the point is far enough away. Furthermore, by careful implementation (i.e., grid in the angles space), we can decide, in constant time, which cone the query point lies in. Thus, using $O(1/\varepsilon^{(d-1)/2})$ space, we can answer $(1+\varepsilon)$-ANN queries for q if the query point is $\sqrt{\varepsilon}$-far, in constant time.

Next, construct this data-structure for every set P_v, for $v \in \widehat{T}(\mathsf{P})$, where $\widehat{T}(\mathsf{P})$ is a compressed quadtree for P. This results in a data-structure of size $O(n/\varepsilon^{(d-1)/2})$. Given a query point q, we use the algorithm of Theorem 17.6 and stop for a node v as soon P_v is $\sqrt{\varepsilon}$-far, and then we use the secondary data-structure for P_v. It is easy to verify that the algorithm would stop as soon as $\mathrm{diam}(\square_v) = O\big(\sqrt{\varepsilon}\,\mathbf{d}(\mathsf{q},\mathsf{P})\big)$. As such, the number of nodes visited would be $O(\log n + 1/\varepsilon^{d/2})$, and the bound on the query time is identical.

Note that we omitted the construction time (which requires some additional work to be done efficiently), and our query time is slightly worse than the best known. The interested reader can check out the work by Chan [**Cha98**], which is somewhat more complicated than what is outlined here.

Even faster query time. The first to achieve $O(\log(n/\varepsilon))$ query time (using near linear space) was Har-Peled [**Har01b**], using space roughly $O(n\varepsilon^{-d}\log^2 n)$. This was later simplified and improved by Arya and Malamatos [**AM02**], who presented a data-structure with the same query time and of size $O(n/\varepsilon^d)$. These data-structure relies on the notion of computing approximate Voronoi diagrams and performing point-location queries in those diagrams. By extending the notion of approximate Voronoi diagrams, Arya, Malamatos, and Mount [**AMM02**] showed that one can answer $(1+\varepsilon)$-ANN queries in $O(\log(n/\varepsilon))$ time, using $O(n/\varepsilon^{(d-1)})$ space. On the other end of the spectrum, they showed that one can construct a data-structure of size $O(n)$ and query time $O(\log n + 1/\varepsilon^{(d-1)/2})$ (note that for this data-structure $\varepsilon > 0$ has to be specified in advance). In particular, the later result breaks a space/query time trade-off that all other results suffer from (i.e., the query time multiplied by the construction time has dependency of $1/\varepsilon^d$ on ε).

Practical considerations. Arya et al. [**AMN⁺98**] implemented their algorithm. For most inputs, it is essentially a *kd*-tree. The code of their library was carefully optimized and is very efficient. In particular, in practice, the author would expect it to beat most of the algorithms mentioned above. The code of their implementation is available online as a library [**AM98**].

Higher dimensions. All our results have exponential dependency on the dimension, in query and preprocessing time (although the space can probably be made subexponential with careful implementation). Getting a subexponential algorithm requires a completely different technique and is discussed in detail later (see Theorem 20.20_{p276}).

Stronger computation models. If one assume that the points have integer coordinates in the range $[1, U]$, then approximate nearest neighbor queries can be answered in (roughly) $O(\log\log U + 1/\varepsilon^d)$ time [**AEIS99**] or even $O(\log\log(U/\varepsilon))$ time [**Har01b**]. The algorithm of Har-Peled [**Har01b**] relies on computing a compressed quadtree of height $O(\log(U/\varepsilon))$ and performing a fast point-location query in it. This only requires using the floor function and hashing (note that the algorithm of Theorem 17.6 uses the floor function and hashing during the construction, but it is not used during the query). In fact, if one is allowed to slightly blow up the space (by a factor U^δ, where $\delta > 0$ is an arbitrary constant), the ANN query time can be improved to constant [**HM04**].

By shifting quadtrees and creating $\mathsf{d}+1$ quadtrees, one can argue that the approximate nearest neighbor must lie in the same cell (and this cell is of the "right" size) of the query point in one of those quadtrees. Next, one can map the points into a real number, by using the natural space filling curve associated with each quadtree. This results in $\mathsf{d} + 1$ lists of points. One can argue that a constant approximate neighbor must be adjacent to the query point in one of those lists. This can be later improved into $(1 + \varepsilon)$-ANN by spreading $1/\varepsilon^{\mathsf{d}}$ points. This simple algorithm is due to Chan [**Cha02**].

The reader might wonder why we bothered with a considerably more involved algorithm. There are several reasons: (i) This algorithm requires the numbers to be integers of limited length (i.e., $O(\log U)$ bits), (ii) it requires shuffling of bits on those integers (i.e., for computing the inverse of the space filling curve) in constant time, and (iii) the assumption is that one can combine d such integers into a single integer and perform exclusive-or on their bits in constant time. The last two assumptions are not reasonable when the input is made out of floating point numbers.

Further research. In low dimensions, the ANN problem seems to be essentially solved both in theory and practice (such proclamations are inherently dangerous and should be taken with a considerable amount of healthy skepticism). Indeed, for $\varepsilon > 1/\log^{1/d} n$, the current data structure of Theorem 17.6 provides logarithmic query time. Thus, ε has to be quite small for the query time to become bad enough that one would wish to speed it up.

The main directions for further research seem to be on this problem in higher dimensions and solving it in other computation models.

Surveys. A survey on the approximate nearest neighbor search problem in high dimensions is by Indyk [**Ind04**]. In low dimensions, there is a survey by Arya and Mount [**AM05**].

17.6. Exercises

Exercise 17.1 (Better ring tree). Let P be a set of n points in $\mathbb{R}^{\mathsf{d}}$. Show how to build a ring tree, of linear size, that can answer $O(\log n)$-ANN queries in $O(\log n)$ time. [**Hint:** Show that there is always a ring containing $O(n/\log n)$ points, such that it is of width w and its interior radius is $O(w \log n)$. Next, build a ring tree, replicating the points in both children of this ring node. Argue that the size of the resulting tree is linear, and prove the claimed bound on the query time and quality of approximation.**]**

Exercise 17.2 (k-ANN from ANN). In the ***k-approximate nearest neighbor*** problem one has to preprocess a point set P, such that given a query point q, one needs to return k distinct points of P such that their distance from q is at most $(1 + \varepsilon)\ell$, where ℓ is the distance from q to its kth nearest neighbor in P.

Show how to build $O(k \log n)$ ANN data-structures, such that one can answer such a k-ANN query using $O(k \log n)$ "regular" ANN queries in these data-structures and the result returned is correct with high probability. (Prove that your data-structure indeed returns k distinct points.)