MADALGO Summer School 2011
Data-Structures for Geometric Approximation[1]

Sariel Har-Peled

August 10, 2010

[2]Work on this book was partially supported by a NSF CAREER award CCR-0132901 and NSF AF award CCF-0915984.

Contents

Chapter 1

The Power of Grids - Closest Pair and Smallest Enclosing Disk

> The Peace of Olivia. How sweat and peaceful it sounds! There the great powers noticed for the first time that the land of the Poles lends itself admirably to partition.
>
> – The tin drum, Gunter Grass.

In this chapter, we are going to discuss two basic geometric algorithms. The first one, computes the closest pair among a set of n points in linear time. This is a beautiful and surprising result that exposes the computational power of using grids for geometric computation. Next, we discuss a simple algorithm for approximating the smallest enclosing ball that contains k points of the input. This at first looks like a bizarre problem, but turns out to be a key ingredient to our later discussion.

1.1 Preliminaries

For a real positive number α and a point $\mathsf{p} = (x, y)$ in $\mathbb{R}^2$, define $\mathsf{G}_\alpha(\mathsf{p})$ to be the grid point $(\lfloor x/\alpha \rfloor \alpha, \lfloor y/\alpha \rfloor \alpha)$. We call α the ***width*** or ***sidelength*** of the ***grid*** G_α. Observe that G_α partitions the plane into square regions, which we call grid ***cells***. Formally, for any $i, j \in \mathbb{Z}$, the intersection of the half-planes $x \geq \alpha i$, $x < \alpha(i+1)$, $y \geq \alpha j$ and $y < \alpha(j+1)$ is said to be a grid ***cell***. Further we define a ***grid cluster*** as a block of 3×3 contiguous grid cells.

Note, that every grid cell $\square$ of G_α, has a unique ID; indeed, let $\mathsf{p} = (x, y)$ be any point in $\square$, and consider the pair of integer numbers $\mathrm{id}_\square = \mathrm{id}(\mathsf{p}) = (\lfloor x/\alpha \rfloor, \lfloor y/\alpha \rfloor)$. Clearly, only points inside $\square$ are going to be mapped to $\mathrm{id}_\square$. We can use this to store a set P of points inside a grid efficiently. Indeed, given a point p, compute its $\mathrm{id}(\mathsf{p})$. We associate with each unique id a data-structure (e.g., a linked list) that stores all the points of P falling into this grid cell (of course, we do not maintain such data-structures for grid cells which are empty). So, once we have computed $\mathrm{id}(\mathsf{p})$, we fetch the data structure associated with this cell, by using hashing. Namely, we store pointers to all those data-structures in a hash table, where each such data-structure is indexed by its unique id. Since the ids are integer numbers, we can do the hashing in constant time.

Assumption 1.1.1 Throughout the discourse, we assume that every hashing operation takes (worst case) constant time. This is quite a reasonable assumption when true randomness is available (using for example perfect hashing [CLRS01]).

Assumption 1.1.2 Our computation model is the unit cost RAM model, where every operation on real numbers takes constant time, including log and $\lfloor \cdot \rfloor$ operations. We will (mostly) ignore numerical issues and assume exact computation.

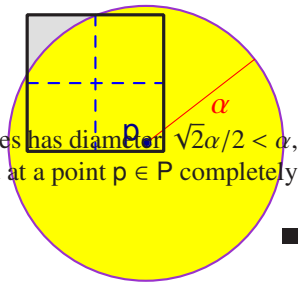
Definition 1.1.3 For a point set P and a parameter α, the ***partition*** of P into subsets by the grid G_α, is denoted by $\mathsf{G}_\alpha(\mathsf{P})$. More formally, two points $\mathsf{p}, \mathsf{q} \in \mathsf{P}$ belong to the same set in the partition $\mathsf{G}_\alpha(\mathsf{P})$, if both points are being mapped to the same grid point or equivalently belong to the same grid cell; that is, $\mathrm{id}(\mathsf{p}) = \mathrm{id}(\mathsf{q})$.

1.2 Closest Pair

We are interested in solving the following problem:

Problem 1.2.1 Given a set P of n points in the plane, find the pair of points closest to each other. Formally, return the pair of points realizing $\mathcal{CP}(\mathsf{P}) = \min_{\mathsf{p}\neq\mathsf{q},\ \mathsf{p},\mathsf{q}\in\mathsf{P}} \|\mathsf{p}-\mathsf{q}\|$.

The following is an easy standard ***packing argument*** that underlines, under various disguises, many algorithms in Computational Geometry.



Lemma 1.2.2 *Let P be a set of points contained inside a square $\square$, such that the side-length of $\square$ is $\mathcal{CP}(\mathsf{P})$. Then $|\mathsf{P}| \leq 4$.*

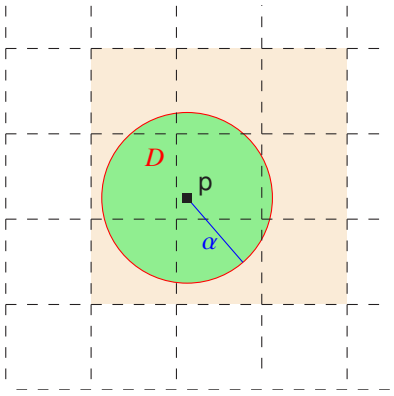
Proof: Partition P into four equal squares $\square_1, \ldots, \square_4$, and observe that each of these squares has diameter $\sqrt{2}\alpha/2 < \alpha$, and as such it can contain at most one of points of P; that is, the disk of radius α centered at a point $\mathsf{p} \in \mathsf{P}$ completely covers the subsquare containing it, see figure on the right.

Note, that the set P can have four points if it is the four corners of $\square$. ■

Lemma 1.2.3 *Given a set P of n points in the plane, and a distance α, one can verify in linear time, whether $\mathcal{CP}(\mathsf{P}) < \alpha$, $\mathcal{CP}(\mathsf{P}) = \alpha$, or $\mathcal{CP}(\mathsf{P}) > \alpha$.*

Proof: Indeed, store the points of P in the grid G_α. For every non-empty grid cell, we maintain a linked list of the points inside it. Thus, adding a new point p takes constant time. Specifically, compute $\mathrm{id}(\mathsf{p})$, check if $\mathrm{id}(\mathsf{p})$ already appears in the hash table, if not, create a new linked list for the cell with this ID number, and store p in it. If a linked list already exist for $\mathrm{id}(\mathsf{p})$, just add p to it. This takes $O(n)$ time overall.

Now, if any grid cell in $\mathsf{G}_\alpha(\mathsf{P})$ contains more than, say, 4 points of P, then it must be that the $\mathcal{CP}(\mathsf{P}) < \alpha$, by Lemma 1.2.2.



Thus, when we insert a point p, we can fetch all the points of P that were already inserted in the cell of p, and the 8 adjacent cells (i.e., all the points stored in the cluster of p); that is, these are the cells of the grid G_α that intersects the disk $D = \mathrm{disk}(\mathsf{p}, \alpha)$ centered at p with radius α, see figure on the right. If there is a point closer to p than α that was already inserted then it must be stored in one of these 9 cells (since it must be inside D). Now, each one of those cells must contain at most 4 points of P by Lemma 1.2.2 (otherwise, we would already have stopped since the $\mathcal{CP}(\cdot)$ of the inserted points is smaller than α). Let S be the set of all those points, and observe that $|S| \leq 9 \cdot 4 = O(1)$. Thus, we can compute, by brute force, the closest point to p in S. This takes $O(1)$ time. If $\mathbf{d}(\mathsf{p}, S) < \alpha$, we stop, otherwise, we continue to the next point.

Overall, this takes at most linear time.

As for correctness, observe that the algorithm returns '$\mathcal{CP}(\mathsf{P}) < \alpha$' only after finding a pair of points of P with distance smaller than α. So, assume that p and q are the pair of points of P realizing the closest pair, and $\|\mathsf{p}-\mathsf{q}\| = \mathcal{CP}(\mathsf{P}) < \alpha$. Clearly, when the later point (say p) is being inserted, the set S would contain q, and as such the algorithm would stop and return '$\mathcal{CP}(\mathsf{P}) < \alpha$'. Similar argumentation works for the case that $\mathcal{CP}(\mathsf{P}) = \alpha$. Thus if the algorithm returns '$\mathcal{CP}(\mathsf{P}) > \alpha$' it must be that $\mathcal{CP}(\mathsf{P})$ is not smaller than α or equal to it. Namely, it must be larger. Thus, the algorithm output is correct. ■

Remark 1.2.4 Assume that $\mathcal{CP}(\mathsf{P} \setminus \{\mathsf{p}\}) \geq \alpha$, but $\mathcal{CP}(\mathsf{P}) < \alpha$. Furthermore, assume that we use Lemma 1.2.3 on P, where $\mathsf{p} \in \mathsf{P}$ is the last point to be inserted. When p is being inserted, not only we discover that $\mathcal{CP}(\mathsf{P}) < \alpha$, but in fact, by checking the distance of p to all the points stored in its cluster, we can compute the closest point to p in $\mathsf{P} \setminus \{\mathsf{p}\}$, and denote this point by q. Clearly, pq is the closest pair in P, and this last insertion still takes only constant time.

Slow algorithm. Lemma 1.2.3 provides a natural way of computing $\mathcal{CP}(\mathsf{P})$. Indeed, permute the points of P in an arbitrary fashion, and let $\mathsf{P} = \langle \mathsf{p}_1, \ldots, \mathsf{p}_n \rangle$. Next, let $\alpha_{i-1} = \mathcal{CP}(\{\mathsf{p}_1, \ldots, \mathsf{p}_{i-1}\})$. We can check if $\alpha_i < \alpha_{i-1}$, by just calling the algorithm of Lemma 1.2.3 on P_i and α_{i-1}. In fact, if $\alpha_i < \alpha_{i-1}$, the algorithm of Lemma 1.2.3, would return '$\mathcal{CP}(\mathsf{P}_i) < \alpha_{i-1}$', and the two points of P_i realizing α_i.

So, consider the "good" case, where $\alpha_i = \alpha_{i-1}$; that is, the length of the shortest pair does not change when p_i is being inserted. In this case, we do not need to rebuild the data structure of Lemma 1.2.3 to store $\mathsf{P}_i = \langle \mathsf{p}_1, \ldots, \mathsf{p}_i \rangle$. We can just reuse the data-structure from the previous iteration that was used by P_{i-1} by inserting p_i into it. Thus, inserting a single point takes constant time, as long as the closest pair does not change.

Things become problematic when $\alpha_i < \alpha_{i-1}$, because then we need to rebuild the grid data structure, and reinsert all the points of $\mathsf{P}_i = \langle \mathsf{p}_1, \ldots, \mathsf{p}_i \rangle$ into the new grid $\mathsf{G}_{\alpha_i}(\mathsf{P}_i)$. This takes $O(i)$ time.

In the end of this process, we output the number α_n, together with the two points of P that realize the closest pair.

Observation 1.2.5 *If the closest pair distance, in the sequence $\alpha_1, \ldots, \alpha_n$, changes only t times, then the running time of our algorithm would be $O(nt + n)$. Naturally, t might be $\Omega(n)$, so this algorithm might take quadratic time in the worst case.*

Linear time algorithm. Surprisingly[2], we can speed up the above algorithm to have linear running time by spicing it up by using randomization.

We pick a random permutation of the points of P, and let $\langle \mathsf{p}_1, \ldots, \mathsf{p}_n \rangle$ be this permutation. Let $\alpha_2 = \|\mathsf{p}_1 - \mathsf{p}_2\|$, and start inserting the points into the data structure of Lemma 1.2.3. We will keep the invariant that α_i would be the closest pair distance in the set P_i, for $i = 2, \ldots, n$.

In the ith iteration, if $\alpha_i = \alpha_{i-1}$, then this insertion takes constant time. If $\alpha_i < \alpha_{i-1}$, then we know what is the new closest pair distance α_i (see Remark 1.2.4), rebuild the grid and reinsert the i points of P_i from scratch into the grid G_{α_i}. This rebuilding of $\mathsf{G}_{\alpha_i}(\mathsf{P}_i)$ takes $O(i)$ time.

Finally, the algorithm returns the number α_n and the two points of P_n realizing it, as the closest pair in P.

Lemma 1.2.6 *Let t be the number of different values in the sequence $\alpha_2, \alpha_3, \ldots, \alpha_n$. Then $\mathbf{E}[t] = O(\log n)$. As such, in expectation, the above algorithm rebuilds the grid $O(\log n)$ times.*

Proof: For $i \geq 3$, let X_i be an indicator variable that is one if and only if $\alpha_i < \alpha_{i-1}$. Observe that $\mathbf{E}[X_i] = \mathbf{Pr}[X_i = 1]$ (as X_i is an indicator variable), and $t = \sum_{i=3}^{n} X_i$.

To bound $\mathbf{Pr}[X_i = 1] = \mathbf{Pr}[\alpha_i < \alpha_{i-1}]$, we (conceptually) fix the points of P_i and randomly permute them. A point $\mathsf{q} \in \mathsf{P}_i$ is ***critical***, if $\mathcal{CP}(\mathsf{P}_i \setminus \{\mathsf{q}\}) > \mathcal{CP}(\mathsf{P}_i)$. If there are no critical points, then $\alpha_{i-1} = \alpha_i$ and then $\mathbf{Pr}[X_i = 1] = 0$ (this happens, for example, if there are two pairs of points realizing the closest distance in P_i). If there is one critical point, then $\mathbf{Pr}[X_i = 1] = 1/i$, as this is the probability that this critical point would be the last point in the random permutation of P_i.

Assume there are two critical points and let p, q be this unique pair of points of P_i realizing $\mathcal{CP}(\mathsf{P}_i)$. The quantity α_i is smaller than α_{i-1} only if either p or q is p_i. The probability for that is $2/i$ (i.e., the probability in a random permutation of i objects, that one of two marked objects would be the last element in the permutation).

Observe, that there can not be more than two critical points. Indeed, if p and q are two points that realize the closest distance, then if there is a third critical point s, then $\mathcal{CP}(\mathsf{P}_i \setminus \{\mathsf{s}\}) = \|\mathsf{p} - \mathsf{q}\|$, and hence the point s is not critical.

Thus, $\mathbf{Pr}[X_i = 1] = \mathbf{Pr}[\alpha_i < \alpha_{i-1}] \leq 2/i$, and by linearity of expectations, we have that $\mathbf{E}[t] = \mathbf{E}\left[\sum_{i=3}^{n} X_i\right] = \sum_{i=3}^{n} \mathbf{E}[X_i] \leq \sum_{i=3}^{n} 2/i = O(\log n)$. ■

Lemma 1.2.6 implies that, in expectation, the algorithm rebuilds the grid $O(\log n)$ times. By Observation 1.2.5, the running time of this algorithm, in expectation, is $O(n \log n)$. However, we can do better than that. Intuitively, rebuilding the grid in early iterations of the algorithm is cheap, and only late rebuilds (when $i = \Omega(n)$) are expensive, but the number of such expensive rebuilds is small (in fact, in expectation it is a constant).

Theorem 1.2.7 *For set P of n points in the plane, one can compute the closest pair of P in expected linear time.*

[2] Surprise in the eyes of the beholder. The reader might not be surprised at all, and might be mildly disgusted by the whole affair. In this case, the reader should read any occurrence of "surprisingly" in the text as being "mildly disgustingly".

Proof: The algorithm is described above. As above, let X_i be the indicator variable which is 1 if $\alpha_i \neq \alpha_{i-1}$, and 0 otherwise. Clearly, the running time is proportional to

$$R = 1 + \sum_{i=3}^{n} (1 + X_i \cdot i).$$

Thus, the expected running time is proportional to

$$\begin{aligned}\mathbf{E}[R] = \mathbf{E}\left[1 + \sum_{i=3}^{n}(1 + X_i \cdot i)\right] &\leq n + \sum_{i=3}^{n} \mathbf{E}[X_i] \cdot i \leq n + \sum_{i=3}^{n} i \cdot \mathbf{Pr}[X_i = 1] \\ &\leq n + \sum_{i=3}^{n} i \cdot \frac{2}{i} \leq 3n,\end{aligned}$$

by linearity of expectation and since $\mathbf{E}[X_i] = \mathbf{Pr}[X_i = 1]$, and since $\mathbf{Pr}[X_i = 1] \leq 2/i$ (as shown in the proof of Lemma 1.2.6). Thus, the expected running time of the algorithm is $O(\mathbf{E}[R]) = O(n)$. ■

Theorem 1.2.7 is a surprising result, since it implies that uniqueness (i.e., deciding if n real numbers are all distinct) can be solved in linear time. Indeed, compute the distance of the closest pair of the given numbers (think about the numbers as points on the x-axis). If this distance is zero, then clearly they are not all unique.

However, there is a lower bound of $\Omega(n \log n)$ on the running time to solve uniqueness, using the comparison model. This "reality dysfunction" can be easily explained once one realizes that the computation model of Theorem 1.2.7 is considerably stronger, using hashing, randomization, and the floor function.

1.3 A Slow 2-Approximation Algorithm for the k-Enclosing Disk

For a disk D, we denote by radius(D) the ***radius*** of D. Let $D_{\text{opt}}(\mathsf{P}, k)$ be a disk of minimum radius which contains k points of P, and let $r_{\text{opt}}(\mathsf{P}, k)$ denote the radius of $D_{\text{opt}}(\mathsf{P}, k)$. For $k = 2$, this is equivalent to computing the closest pair of points of P.

Lemma 1.3.1 *For any point set P and $\alpha > 0$, we have that if $\alpha \leq 2r_{\text{opt}}(\mathsf{P}, k)$ then any cell of the grid G_α contains at most $5k$ points; that is $\mathsf{gd}_\alpha(\mathsf{P}) \leq 5k$.*

Lemma 1.3.2 *Given a set P of n points in $\mathbb{R}^{\mathsf{d}}$, and parameter k, one can compute, in $O\left(n(n/k)^{\mathsf{d}}\right)$ deterministic time, a ball $\mathbf{b}$ that contains k points of P and its radius* radius($\mathbf{b}$) $\leq 2r_{\text{opt}}(\mathsf{P}, k)$*, where $r_{\text{opt}}(\mathsf{P}, k)$ is the radius of the smallest ball in $\mathbb{R}^{\mathsf{d}}$ containing k points of P.*

1.4 A Linear Time 2-Approximation for the k-Enclosing Disk

Theorem 1.4.1 *Given a set P of n points in the plane, and a parameter k, the algorithm* **algDCover** *computes, in expected linear time, a radius α, such that $r_{\text{opt}}(\mathsf{P}, k) \leq \alpha \leq 2r_{\text{opt}}(\mathsf{P}, k)$, where $r_{\text{opt}}(\mathsf{P}, k)$ is the minimum radius of a disk covering k points of P.*

1.5 Bibliographical notes

Our closest-pair algorithm follows Golin *et al.* [GRSS95]. This is in turn a simplification of a result of Rabin [Rab76]. Smid provides a survey of such algorithms [Smi00]).

The minimum disk approximation algorithm is a simplification of the work of Har-Peled and Mazumdar [HM03]. Note that this algorithm can be easily adapted to any point set in constant dimension (with the same running time).

Computing the exact minimum disk containing k points. By plugging the algorithm of Theorem 1.4.1 into the exact algorithm of Matoušek [Mat95], one gets a $O(nk)$ time algorithm that computes the minimum disk containing k points. It is conjectured that any exact algorithm for this problem requires $\Omega(nk)$ time.

Chapter 2

Quadtrees - Hierarchical Grids

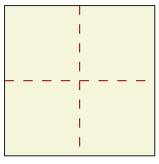
What do you know? You know just what you perceive.
What can you show? Nothin' of what you believe,
And as you grow, each thread of life that you leave
Will spin around your deeds and dictate your needs
As you sell your soul and you sow your seeds,
And you wound yourself and your loved one bleeds,
And your habits grow, and your conscience feeds
On all that you thought you should be –
I never thought this could happen to Meeeeeeeee
– Dreidel, Don McLean.

In this chapter, we discuss quadtrees which is arguably one of the simplest and most powerful geometric data-structure. We begin in Section 2.1 by giving a simple application of quadtrees and describe a clever way for performing point-location queries quickly in such a quadtree. In Section 2.2, we describe how such quadtrees can be compressed and how they can be quickly constructed and used for point-location queries. In Section 2.3 we show how to dynamically maintain a compressed quadtree under insertions and deletions of points.

2.1 Quadtrees - a simple point-location data-structure

Let P_{map} be a planar map. To be more concrete, let P_{map} be a partition of the unit square into polygons. The partition P_{map} can represent any planar map, where a region in the map might be composed of several polygons (or triangles). For the sake of simplicity, assume that every vertex in P_{map} shares a constant number of polygons.

We want to preprocess P_{map} for point-location queries, so that given a query point we can figure out which polygon contains the query (see figure on the right). Of course, there are numerous data-structures that can do this, but let us consider the following simple solution (which in the worst case, can be quite bad).



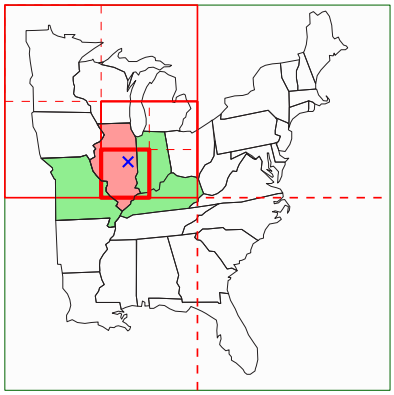
Build a tree $\mathcal{T}$, where every node $v \in T$ corresponds to a cell $\square_v$ (i.e., a square), and the root corresponds to the unit square. Each node has four children that correspond to the four equal size squares formed by splitting $\square_v$ by horizontal and vertical cuts, see figure on the right.

The construction is recursive, and we start from $v = \text{root}_T$. The ***conflict-list*** of the square $\square_v$ (i.e., the square associated with v) is a list of all the polygons of P_{map} that intersect $\square_v$. If the current node's conflict list has more than, say, nine[②] polygons, we create its children nodes, and we call recursively on each child. We compute each child's conflict-list from its parent list. As such, we stop at a leaf, if its conflict-list is of size at most nine. For each constructed leaf, we store in it its conflict-list (but we do not store the conflict list for internal nodes).

[②]The constant here is arbitrary, it just has to be at least the number of polygons of P_{map} meeting in one common corner.

Given a query point q, in the unit square, we can compute the polygon of P_{map} containing q, by traversing down $\mathcal{T}$ from the root, repeatedly going into the child of the current node, whose square contains q. We stop as soon as we reach a leaf, and then we scan the leaf's conflict-list, and check which of the polygons in this list contains q.

An example is depicted on the right, where the nodes on the search path of the point-location query (i.e., the nodes accessed during the query execution) are shown. The marked polygons are all the polygons in the conflict list of the leaf containing the query point.



Of course, in the worst case, if the polygons are long and skinny, this quadtree might have unbounded complexity. However, for reasonable inputs (say, the polygons are in fact fat triangles), then the quadtree would have linear complexity in the input size. The big advantage of quadtrees of course, is their simplicity. In a lot of cases, quadtrees would be a sufficient solution, and seeing how to solve a problem using a quadtree might be a first insight into a problem.

2.1.1 Fast point-location in a quadtree

One possible interpretation of quadtrees is that they are a multi-grid representation of a point-set.

Definition 2.1.1 (Canonical squares and canonical grids.) A square is a ***canonical square***, if it is contained inside the unit square, it is a cell in a grid G_r, and r is a power of two (i.e., it might correspond to a node in a quadtree). We will refer to such a grid G_r, as a ***canonical grid***.

Consider a node v of a quadtree of depth i (the root has depth 0), and its associated square $\square_v$. The side length of $\square_v$ is 2^{-i}, and it is a canonical square in the canonical grid $\mathsf{G}_{2^{-i}}$. In fact, we will refer to $\ell(v) = -i$ as the ***level*** of v. However, a cell in a grid has a unique ID made out of two integer numbers. Thus, a node v of a quadtree is uniquely defined by the triple $\mathrm{id}(v) = (\ell(v), \lfloor x/r \rfloor, \lfloor y/r \rfloor)$, where (x, y) is any point in $\square_v$, and $r = 2^{\ell(v)}$.

Furthermore, given a query point q, and a desired level ℓ, we can compute the ID of the quadtree cell of this level that contains q in constant time. Thus, this suggests a very natural algorithm for doing a point-location in a quadtree: Store all the IDs of nodes in the quadtree in a hash-table, and also compute the maximal depth h of the quadtree. Given a query point q, we now have access to any node along the point-location path of q in $\mathcal{T}$, in constant time. In particular, we want to find the point in $\mathcal{T}$ where this path "falls off" the quadtree (i.e., reaches the leaf). This we can find by performing a binary search for the leaf.

```
QTFastPLI(T, q, l, h).
  m ← ⌊(l + h)/2⌋
  v ← QTGetNode(T, q, m)
  if v = null then
    return QTFastPLI(T, q, l, m − 1).
  w ← Child(v, q)
      //w is the child of v containing q.
  If w = null then
    return v
  return QTFastPLI(T, q, m + 1, h)
```

Figure 2.1: One can perform point-location in a quadtree $\mathcal{T}$ by calling **QTFastPLI** $(T, q, 0, \text{height}(T))$.

Let **QTGetNode**(T, q, d) denote the procedure that, in constant time, returns the node v of depth d in the quadtree $\mathcal{T}$ such that $\square_v$ contains the point q. Given a query point q, we can perform point-location in $\mathcal{T}$ (i.e., finding the leaf containing the query) by calling **QTFastPLI**$(T, q, 0, \text{height}(T))$. See Figure 2.1 for the pseudo-code for **QTFastPLI**.

Lemma 2.1.2 *Given a quadtree $\mathcal{T}$ of size n and of height h, one can preprocess it (using hashing), in linear time, such that one can perform a point-location query in $\mathcal{T}$ in $O(\log h)$ time. In particular, if the quadtree has height $O(\log n)$ (i.e., it is "balanced"), then one can perform a point-location query in $\mathcal{T}$ in $O(\log \log n)$ time.*

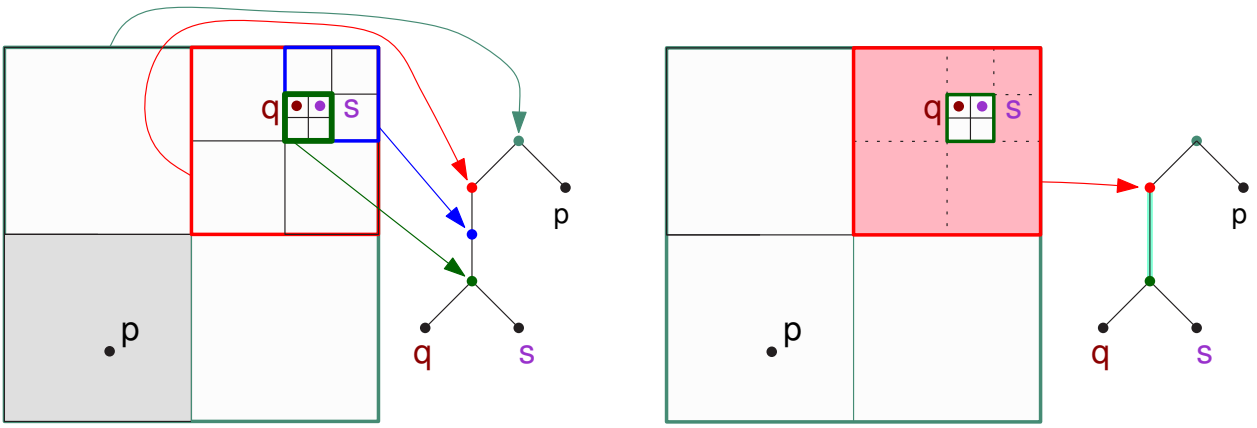


Figure 2.2: A point set and its quadtree, and its compressed quadtree. Note, that each node is associated with a canonical square. For example, the node in the above quadtree marked by p, would store the gray square showed on the left, and also would store the point p in this node.

2.2 Compressed Quadtrees

2.2.1 Definition

Definition 2.2.1 (Spread.) For a set P of n points in a metric space, let

$$\Phi(\mathsf{P}) = \frac{\max_{\mathsf{p},\mathsf{q}\in\mathsf{P}} \|\mathsf{p}-\mathsf{q}\|}{\min_{\mathsf{p},\mathsf{q}\in\mathsf{P},\mathsf{p}\neq\mathsf{q}} \|\mathsf{p}-\mathsf{q}\|} \tag{2.1}$$

be the ***spread*** of P. In words, the spread of P is the ratio between the diameter of P and the distance between the two closest points in P. Intuitively, the spread tells us the range of distances that P possesses.

One can build a quadtree $\mathcal{T}$ for P, storing the points of P in the leaves of $\mathcal{T}$, where one keeps splitting a node as long as it contains more than one point of P. During this recursive construction, if a leaf contains no points of P, we save space by not creating this leaf, and instead creating a null pointer in the parent node for this child.

Lemma 2.2.2 *Let P be a set of n points contained in the unit square, such that $\operatorname{diam}(\mathsf{P}) = \max_{\mathsf{p},\mathsf{q}\in\mathsf{P}} \|\mathsf{p}-\mathsf{q}\| \geq 1/2$. Let $\mathcal{T}$ be a quadtree of P constructed over the unit square, where no leaf contains more than one point of P. Then, the depth of $\mathcal{T}$ is bounded by $O(\log \Phi)$, it can be constructed in $O(n \log \Phi)$ time, and the total size of $\mathcal{T}$ is $O(n \log \Phi)$, where $\Phi = \Phi(\mathsf{P})$.*

Proof: The construction is done by a straightforward recursive algorithm as described above.

Let us bound the depth of $\mathcal{T}$. Consider any two points $\mathsf{p},\mathsf{q} \in \mathsf{P}$, and observe that a node v of $\mathcal{T}$ of level $u = \lfloor \lg \|\mathsf{p}-\mathsf{q}\| \rfloor - 1$ containing p must not contain q (we remind the reader that $\lg n = \log_2 n$). Indeed, the diameter of $\square_v$ is smaller than $\sqrt{1^2+1^2}2^u = \sqrt{2}2^u \leq \sqrt{2}\,\|\mathsf{p}-\mathsf{q}\|/2 < \|\mathsf{p}-\mathsf{q}\|$. Thus, $\square_v$ can not contain both p and q. In particular, any node u of $\mathcal{T}$ of level $r = -\lceil \lg \Phi \rceil - 2$ can contain at most one point of P, where $\Phi = \Phi(\mathsf{P})$. Otherwise, if u contained two distinct points, say $\mathsf{p},\mathsf{q} \in \mathsf{P}$, in its square, then

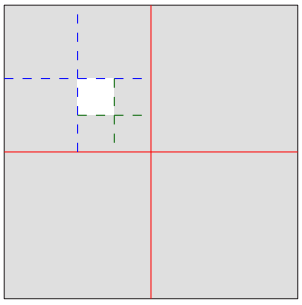
$$\|\mathsf{p}-\mathsf{q}\| \leq \sqrt{2}2^r = \frac{\sqrt{2}}{2^{\lceil \lg \Phi \rceil + 2}} \leq \frac{\sqrt{2}}{4\Phi} < \frac{\operatorname{diam}(\mathsf{P})}{\Phi},$$

since $\operatorname{diam}(\mathsf{P}) \geq 1/2$. This implies that $\Phi < \operatorname{diam}(\mathsf{P}) / \|\mathsf{p}-\mathsf{q}\|$, which is impossible by the definition of the spread, see Eq. (2.1). Thus, all the nodes of $\mathcal{T}$ are of depth $O(\log \Phi)$.

Since the construction algorithm spends $O(n)$ time at each level of $\mathcal{T}$, it follows that the construction time is $O(n \log \Phi)$, and this also bounds the size of the quadtree $\mathcal{T}$. ■

The bounds of Lemma 2.2.2 are tight, as one can easily verify. But in fact, if you inspect a quadtree generated by Lemma 2.2.2, you would realize that there are a lot of nodes of $\mathcal{T}$ which are of degree one (the degree of a node is the

number of children it has). See Figure 2.2 for an example. Indeed, a node v of $\mathcal{T}$ has more than one child, only if it has at least two children x and y, such that both $\square_x$ and $\square_y$ contains points of P. Let P_v be the subset of points of P stored in the subtree of v, and observe that $\mathsf{P}_x \cup \mathsf{P}_y \subseteq \mathsf{P}_v$ and $\mathsf{P}_x \cap \mathsf{P}_y = \emptyset$. Namely, such a node v splits P_v into, at least, two non-empty subsets and globally there can be only $n-1$ such splitting nodes. Thus, a regular quadtree might have a large number of "useless" nodes that one should be able to get rid of and get a more compact data-structure.



If you compress them, they will fit in memory. We can replace such a sequence of edges by a single edge. To this end, we will store inside each quadtree node v, its square $\square_v$, and its level $\ell(v)$. Given a path of vertices in the quadtree that are all of degree one, we will replace them with a single vertex v that corresponds to the first vertex in this path, and its only child would be the last vertex in this path (this is the first node of degree larger than one). This ***compressed node*** v has a single child, and the region rg_v that v "controls" is an annulus, seen as the gray area in the figure on the right.

Otherwise, if v is not compressed, the ***region*** that v is in charge of is a square $\mathsf{rg}_v = \square_v$. Specifically, an uncompressed node v is either a leaf or a node that splits the point set into two (or more) non-empty subsets (i.e., two or more of the children of v have points stored in their subtrees). For a compressed node v, its sole child corresponds to the inner square, which contains all the points of P_v. We call the resulting tree a ***compressed quadtree***.

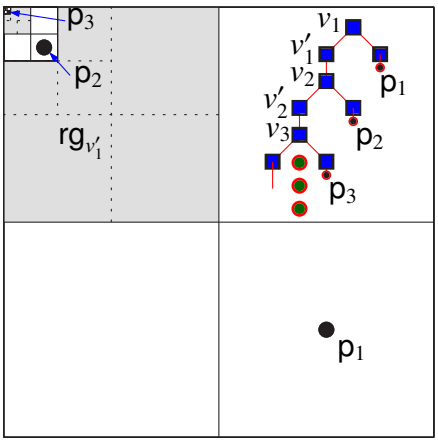
Observe, that the only nodes in a compressed quadtree with a single child are compressed (they might be trivially compressed, the child being one of the four subsquares of the parent). In particular, since any node that has only a single child is compressed, we can charge it to its parent, which has two (or more) children. There are at most $n-1$ internal nodes in the new compressed quadtree that have degree larger than one, since one can split a set of size n at most $n-1$ times till ending up with singletons. As such, a compressed quadtree has linear size (however, it still can have linear depth in the worst case).

See Figure 2.2 for an example of a compressed quadtree.

Example 2.2.3 As an application for compressed quadtrees, consider the problem of counting how many points are inside a query rectangle r. We can start from the root of the quadtree, and recursively traverse it, going down a node only if its region intersects the query rectangle. Clearly, we will report all the points contained inside r. Of course, we have no guarantee about the query time, but in practice, this might be fast enough.

Note, that one can perform the same task using a regular quadtree. However, in this case, such a quadtree might require unbounded space. Indeed, the spread of the point set might be arbitrarily large, and as such the depth (and thus size) of the quadtree can be arbitrarily large. On the other hand, a compressed quadtree would use only $O(n)$ space, where n is the number of points stored in it.

Example 2.2.4 Consider the point set $\mathsf{P} = \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}$, where $\mathsf{p}_i = (3/4, 3/4)/8^{i-1}$, for $i = 1, \ldots, n$. The compressed quadtree for this point set is depicted on the right.



Here,we have

$$\square_{v_i} = [0,1]^2/8^{i-1} \quad \text{and} \quad \square_{v'_i} = [0,1]^2/(2 \cdot 8^{i-1}).$$

As such, v'_i is a compressed node, where

$$\mathsf{rg}_{v'_i} = \square_{v'_i} \setminus \square_{v_{i+1}}.$$

Note, that this compressed quadtree has depth and size $\Theta(n)$. In particular, in this case the compressed quadtree looks like a linked list storing the points.

2.2.2 Efficient construction of compressed quadtrees

2.2.2.1 Bit twiddling and compressed quadtrees

Unfortunately, to be able to efficiently build compressed quadtrees one requires a slightly bizarre computational model. We are assuming implicitly the unit RAM model, where one can store and manipulate arbitrarily large real numbers in constant time. To work with grids efficiently we need to be able to compute quickly (i.e., constant time) $\lg(x)$ (i.e., $\log_2$), 2^x, and $\lfloor x \rfloor$. Strangely, computing a compressed quadtree efficiently is equivalent to the following operation.

Definition 2.2.5 (Bit Index.) Let $\alpha, \beta \in [0, 1)$ be two real numbers. Assume these numbers in base two are written as $\alpha = 0.\alpha_1\alpha_2 \ldots$ and $\beta = 0.\beta_1\beta_2 \ldots$. Let $\mathrm{bit}_\Delta(\alpha, \beta)$ be the index of the first bit after the period in which they differ.

For example, $\mathrm{bit}_\Delta(1/4 = 0.01_2, 3/4 = 0.11_2) = 1$ and $\mathrm{bit}_\Delta(7/8 = 0.111_2, 3/4 = 0.110_2) = 3$.

Lemma 2.2.6 *If one can compute a compressed quadtree of two points in constant time, then one can compute* $\mathrm{bit}_\Delta(\alpha, \beta)$ *in constant time.*

Proof: Given α and β we need to compute $\mathrm{bit}_\Delta(\alpha, \beta)$. Now, if $|\alpha - \beta| > 1/128$ then we can compute $\mathrm{bit}_\Delta(\alpha, \beta)$ in constant time. Otherwise, build a one dimensional compressed quadtree (i.e., a compressed trie) for the set $\{\alpha, \beta\}$. The root is a compressed node in this tree, and its only child[②] v has sidelength 2^{-i}; that is, $\ell(v) = -i$. A quadtree node stores the canonical grid cell it corresponds to, and as such $\ell(v)$ is available from the compressed quadtree. As such, α and β are identical in the first i bits of their binary representation, but clearly the differ at the $(i + 1)$th bit, as the next level of the quadtree splits them into two different subtrees. As such, if one can compute a compressed trie of two numbers in constant time, then one can compute $\mathrm{bit}_\Delta(\alpha, \beta)$ in constant time.

If the reader is uncomfortable with building a one dimensional compressed quadtree, then use the point set $\mathsf{P} = \{(\alpha, 1/3), (\beta, 1/3)\}$, and compute a compressed quadtree $\mathcal{T}$ for P having the unit square as the root. Similar argumentation would apply in this case. ■

Interestingly, once one has such an operation at hand, it is quite easy to compute a compressed quadtree efficiently via "linearization". The idea is to define an order on the nodes of a compressed quadtree and maintain the points sorted in this order, see Section 2.3 below for details. Given the points sorted in this order, one can build the compressed quadtree in constant time using, essentially, scanning.

However, the resulting algorithm is somewhat counter intuitive. As a first step, we suggest a direct construction algorithm.

2.2.2.2 A construction algorithm

Let P be a set of n points in the unit square, with unbounded spread. We are interested in computing the compressed quadtree of P. The regular algorithm for computing a quadtree when applied to P might require unbounded time (but in practice it might be fast enough). Modifying it so it requires only quadratic time is an easy exercise. Getting down to $O(n \log n)$ time requires some cleverness.

Theorem 2.2.7 *Given a set* P *of n points in the plane, one can compute a compressed quadtree of* P *in* $O(n \log n)$ *deterministic time.*

Proof: Compute, in linear time, a disk D of radius r, which contains at least $n/10$ of the points of P, such that $r \leq 2r_{\mathrm{opt}}(\mathsf{P}, n/10)$, where $r_{\mathrm{opt}}(\mathsf{P}, n/10)$ denotes the radius of the smallest disk containing $n/10$ points of P. Computing D can be done in linear time, by a rather simple algorithm (Lemma 1.3.2).

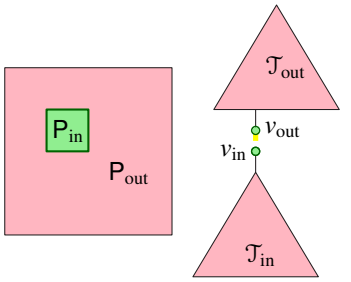
Let $\alpha = 2^{\lfloor \lg r \rfloor}$. Consider the grid G_α. It has a cell that contains at least $(n/10)/25$ points of P (since D is covered by $5 \times 5 = 25$ grid cells of G_α and $\alpha \geq r/2$), and no grid cell contains more than $5(n/10)$ points, by Lemma 1.3.1. Thus, compute $\mathsf{G}_\alpha(\mathsf{P})$, and find the cell $\square$ containing the largest number of points. Let P_{in} be the points inside this cell $\square$, and $\mathsf{P}_{\mathrm{out}}$ the points outside this cell. Specifically, we have

$$\mathsf{P}_{\mathrm{in}} = \mathsf{P} \cap \square \quad \text{and} \quad \mathsf{P}_{\mathrm{out}} = \mathsf{P} \setminus \square = \mathsf{P} \setminus \mathsf{P}_{\mathrm{in}}.$$

We know that $\left|\mathsf{P}_{\mathrm{in}}\right| \geq n/250$, and $\left|\mathsf{P}_{\mathrm{out}}\right| \geq n/2$.

[②] The root is Chinese and is not allowed to have more children at this point in time.

Next, compute (recursively) the compressed quadtrees for P_{in} and P_{out}, respectively, and let $\mathcal{T}_{\text{in}}$ and $\mathcal{T}_{\text{out}}$ denote the respective quadtrees. Create a node in both quadtrees that corresponds to $\square$. For $\mathcal{T}_{\text{in}}$ this would just be the root node v_{in}, since $\mathsf{P}_{\text{in}} \subseteq \square$. For $\mathcal{T}_{\text{out}}$ this would be a new leaf node v_{out}, since $\mathsf{P}_{\text{out}} \cap \square = \emptyset$. Note, that inserting this new node might require linear time, but it requires only changing a constant number of nodes and pointers in both quadtrees Now, hang v_{out} on v_{in} creating a compressed quadtree for P, see figure on the right. In fact, the new glued node (i.e., v_{out} and v_{in}) might be redundant because of compression – this can be done if necessary in constant time.



The overall construction time is $T(n) = O(n) + T(|\mathsf{P}_{\text{in}}|) + T(|\mathsf{P}_{\text{out}}|) = O(n \log n)$. ■

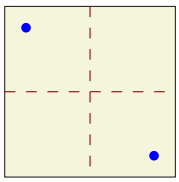
Remark 2.2.8 The reader might wonder where did the Tweedledee and Tweedledum operation (i.e., $\text{bit}_{\Delta}(\cdot,\cdot)$) get used in the algorithm of Theorem 2.2.7. Observe that the hanging stage might require such operation if we hang $\mathcal{T}_{\text{in}}$ from a compressed node of $\mathcal{T}_{\text{out}}$, as computing the new compressed node requires exactly this kind of operation.

Compressed quadtree from squares. It is sometime useful to be able to construct a compressed quadtree from a list of squares that must appear in it as nodes.

For reasons that will become clear later, we want to construct the quadtree out of a list of quadtree nodes that must appear in the quadtree. Namely, we get a list of canonical grid cells that must appear in the quadtree (i.e., the level of the node, together with its grid ID).

Lemma 2.2.9 *Given a list C of n canonical squares, all lying inside the unit square, one can construct a compressed quadtree $\mathcal{T}$ such that for any square $c \in C$, there exists a node $v \in T$, such that $\square_v = c$. The construction time is $O(n \log n)$.*

Proof: For every *canonical* square $\square \in C$, we place two points into a set P, such that any quadtree for P must have $\square$ in it. This is done by putting two points in $\square$ such that they belong to two different sub-squares of $\square$. See figure on the right.



The resulting point-set P has $2n$ points, and we can compute its compressed quadtree $\mathcal{T}$ in $O(n \log n)$ time using Theorem 2.2.7. Observe that any cell of C is an internal node of $\mathcal{T}$. Thus trimming away all the leafs of the quadtree results in a minimal quadtree that contains all the cells of C as nodes. ■

2.2.3 Fingering a Compressed Quadtree - Fast Point Location

Let $\mathcal{T}$ be a compressed quadtree of size n. We would like to preprocess it so that given a query point, we can find the lowest node of $\mathcal{T}$ whose cell contains a query point q. As before, we can perform this by traversing down the quadtree, but this might require $\Omega(n)$ time. Since the range of levels of the quadtree nodes is unbounded, we can no longer use binary search on the levels of $\mathcal{T}$ to answer the query.

Instead, we are going to use a rebalancing technique on $\mathcal{T}$. Namely, we are going to build a balanced tree $\mathcal{T}'$, which would have cross pointers (i.e., fingers) into $\mathcal{T}$. The search would be performed on $\mathcal{T}'$ instead of on $\mathcal{T}$. In the literature, the tree $\mathcal{T}'$ is known as a ***finger tree***.

Definition 2.2.10 Let $\mathcal{T}$ be a tree with n nodes. A ***separator*** in $\mathcal{T}$ is a node v, such that if we remove v from $\mathcal{T}$, we remain with a forest, such that every tree in the forest has at most $\lceil n/2 \rceil$ vertices.

Lemma 2.2.11 *Every tree has a separator, and it can be computed in linear time.*

Proof: Consider $\mathcal{T}$ to be a rooted tree. We precompute for every node in the tree the number of nodes in its subtree by a bottom-up computation that takes linear time. Set v to be the lowest node in $\mathcal{T}$ such that its subtree has $\geq \lceil n/2 \rceil$ nodes in it, where n is the number of nodes of T. This node can be found by performing a walk from the root of $\mathcal{T}$ down to the child with a sufficiently large subtree till this walk gets "stuck". Indeed, let v_1 be the root of $\mathcal{T}$, and let v_i be the child of v_{i-1} with the largest number of nodes in its subtree. Let $s(v_i)$ be the number of nodes in the subtree rooted at v_i. Clearly, there exists a k such that $s(v_k) \geq n/2$ and $s(v_{k+1}) < n/2$.

Clearly, all the subtrees of the children of v_k have size at most $n/2$. Similarly, if we remove v_k and its subtree from $\mathcal{T}$ we remain with a tree with at most $n/2$ nodes. As such v_k is the required separator. ■

This suggests a natural way for processing a compressed quadtree for point-location queries. Find a separator $v \in T$, and create a root node f_v for $\mathcal{T}'$ which has a pointer to v; now recursively build a finger tree for each tree of $T \setminus \{v\}$. Hang the resulting finger trees on f_v. The resulting tree is the required finger tree $\mathcal{T}'$.

Given a query point q, we traverse $\mathcal{T}'$, where at node $f_v \in \mathcal{T}'$, we check whether the query point $\mathsf{q} \in \square_v$, where v is the corresponding node of $\mathcal{T}$. If $\mathsf{q} \notin \square_v$, we continue the search into the child of f_v, which corresponds to the connected component outside $\square_v$ that was hung on f_v. Otherwise, we continue into the child that contains q; naturally, we have to check out the $O(1)$ children of v to decide which one we should continue the search into. This takes constant time per node. As for the depth for the finger tree $\mathcal{T}'$, observe $D(n) \leq 1 + D(\lceil n/2 \rceil) = O(\log n)$. Thus, a point-location query in $\mathcal{T}'$ takes logarithmic time.

Theorem 2.2.12 *Given a compressed quadtree $\mathcal{T}$ of size n, one can preprocess it, in time $O(n \log n)$, such that given a query point q, one can return the lowest node in $\mathcal{T}$ whose region contains q in $O(\log n)$ time.*

Proof: We just need to bound the preprocessing time. Observe that it is $T(n) = O(n) + \sum_{i=1}^{t} T(n_i)$, where $n_1, \ldots n_t$ are the sizes of the subtrees formed by removing the separator node from $\mathcal{T}$. We know that $t = O(1)$ and $n_i \leq \lceil n/2 \rceil$, for all i. As such, $T(n) = O(n \log n)$. ■

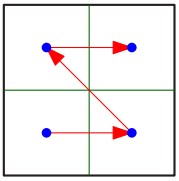
2.3 Dynamic Quadtrees

Here we show how to store compressed quadtrees using any data-structure for ordered sets. The idea is to define an order on the compressed quadtree nodes, and then store the compressed nodes in a data-structure for ordered sets. We show how to perform basic operations using this representation. In particular, we show how to perform insertions and deletions. This provides us with a simple implementation of dynamic compressed quadtrees.

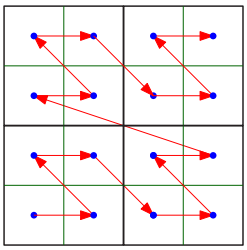
We start our discussion with regular quadtrees, and later on discuss how to handle compressed quadtrees.

2.3.1 Ordering of nodes and points

Consider a regular quadtree $\mathcal{T}$, and a **DFS** traversal of $\mathcal{T}$, where the **DFS** always traverse the children of a node in the same relative order (i.e., say, first the bottom-left child, then the bottom-right child, top-left child, and top-right child).

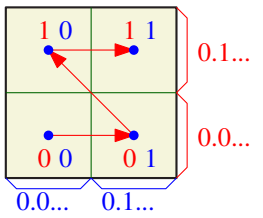


Consider any two canonical squares $\square$ and $\widehat{\square}$, and imagine a quadtree $\mathcal{T}$ that contains both squares (i.e., there are nodes in $\mathcal{T}$ with these squares as their cells). Notice, that the above **DFS** would always visit these two nodes in a specific order, independent of the structure of the rest of the quadtree. Thus, if $\square$ gets visited before $\widehat{\square}$, we denote this fact by $\square \prec \widehat{\square}$. This defines a total ordering over all canonical squares. It would be in fact useful to extend this ordering to also includes points. Thus, consider a point p and a canonical square $\square$. If $\mathsf{p} \in \square$ then we will say that $\square \prec \mathsf{p}$. Otherwise, if $\square \in \mathsf{G}_i$, let $\widehat{\square}$ be the cell in G_i that contains p. We have that $\square \prec p$ if and only if $\square \prec \widehat{\square}$. Next, consider two points p and q, and let G_i be a grid fine enough such that p and q lie in two different cells, say, $\square_{\mathsf{p}}$ and $\square_{\mathsf{q}}$, respectively. Then $\mathsf{p} \prec \mathsf{q}$ if and only if $\square_{\mathsf{p}} \prec \square_{\mathsf{q}}$.



We will refer to the ordering induced by $\prec$ as the ***Q-order***.

The ordering $\prec$ when restricted only to points, is the ordering along a space filling mapping that is induced by the quadtree **DFS**. This ordering is know as the ***Z-order***. Note, however, that since we allow comparing cells to cells, and cells to points, the Q-order no longer has this exact interpretation. Furthermore, unlike the Peano or Hilbert space filling curves, our mapping is not continuous. Also, our mapping has the advantage of being easy to define. Indeed, given a real number $\alpha \in [0, 1)$, with the binary expansion $\alpha = 0.x_1x_2x_3 \ldots$ (i.e., $\alpha = \sum_{i=1}^{\infty} x_i 2^{-i}$), our mapping will map it to the point $(0.x_2x_4x_6 \ldots, 0.x_1x_3x_5 \ldots)$.



To see that this property indeed holds, think about performing a point location query in an infinite quadtree for a point $\mathsf{p} \in [0,1)^2$. In the top level of the quadtree we have four possibilities to continue the search. These four possibilities can be encoded as a binary string of length 2, see figure on the right. Now, if $y(\mathsf{p}) \in [0, 1/2)$ then the first bit in the encoding is 0, and if $y(\mathsf{p}) \in [1/2, 1)$ the first bit output is 1.

Similarly, the second bit in the encoding is the just the first bit in the binary representation of $x(\mathsf{p})$. Now, after resolving the point-location query in the top level we continue this search in the tree. Every level in this traversal generates two bits, and these two bits corresponds to the relevant two bits in the binary representation of $y(\mathsf{p})$ and $x(\mathsf{p})$. In particular, the $2i+1$ and $2i+2$ bit in the encoding of p as a single real number (in binary), is just the ith bits of (the binary representation of) $y(\mathsf{p})$ and $x(\mathsf{p})$, respectively. In particular, for a point $\mathsf{p} \in [0,1)^d$, let $\mathrm{enc}_{\prec}(\mathsf{p})$ denote the number in the range $[0,1)$ encoded by this process.

Claim 2.3.1 *For any two points $\mathsf{p}, \mathsf{q} \in [0,1)^2$, we have that $\mathsf{p} \prec \mathsf{q}$ if and only if $\mathrm{enc}_{\prec}(\mathsf{p}) < \mathrm{enc}_{\prec}(\mathsf{q})$.*

2.3.1.1 Computing the $\mathcal{Q}$-order quickly

For our algorithmic applications, we need to be able to find the ordering according to $\prec$ between any two given cells/points quickly.

Definition 2.3.2 for any two points $\mathsf{p}, \mathsf{q} \in [0,1]^2$, let $\mathsf{lca}(\mathsf{p}, \mathsf{q})$ denote the smallest canonical square that contains both p and q. It intuitively corresponds to the node that must be in any quadtree storing p and q.

To compute $\mathsf{lca}(\mathsf{p}, \mathsf{q})$, we revisit the Tweedledee and Tweedledum operation $\mathrm{bit}_\Delta(\alpha, \beta)$ (see Definition 2.2.5), that for two real numbers $\alpha, \beta \in [0,1)$ returns the index of the first bit in which α and β differ (in base two). The level ℓ of $\square = \mathsf{lca}(\mathsf{p}, \mathsf{q})$ is equal to

$$\ell = 1 - \min(\mathrm{bit}_\Delta(x_\mathsf{p}, x_\mathsf{q}), \mathrm{bit}_\Delta(y_\mathsf{p}, y_\mathsf{q})),$$

where x_p and y_p denote the x and y coordinates of p, respectively. Thus, the side length of $\square = \mathsf{lca}(\mathsf{p}, \mathsf{q})$ is $\Delta = 2^\ell$. Let $x' = \Delta \lfloor x/\Delta \rfloor$ and $y' = \Delta \lfloor y/\Delta \rfloor$. Thus,

$$\mathsf{lca}(\mathsf{p}, \mathsf{q}) = [x', x' + \Delta) \times [y', y' + \Delta).$$

We also define the lca of two cells to be the lca of their centers.

Now, given two cells $\square$ and $\widehat{\square}$, we would like to determine their $\mathcal{Q}$-order. If $\square \subseteq \widehat{\square}$ then $\widehat{\square} \prec \square$. If $\widehat{\square} \subseteq \square$ then $\square \prec \widehat{\square}$. Otherwise, let $\widetilde{\square} = \mathsf{lca}(\square, \widehat{\square})$. We can now determine which children of $\widetilde{\square}$ contains these two cells, and since we know the traversal ordering among children of a node in a quadtree we can now resolve this query in constant time.

Corollary 2.3.3 *Assuming that the bit_Δ operation and the $\lfloor \cdot \rfloor$ operation can be performed in constant time, then one can compute lca of two points (or cells) in constant time. Similarly, their $\mathcal{Q}$-order can be resolved in constant time.*

Computing bit_Δ efficiently. It seems somewhat suspicious that one assumes that the bit_Δ operations can be done in constant time on a classical RAM machine. However it is a reasonable assumption on a real world computer. Indeed, in floating point representation, once you are given a number it is easy to access its mantissa and exponent in constant time. If the exponents are different then bit_Δ can be computed in constant time. Otherwise, we can easily $\oplus_{\mathrm{xor}}$ the mantissas of both numbers, and compute the most significant bit that is one. This can be done in constant time by converting the resulting mantissa into a floating point number, and computing its $\log_2$ (some CPUs have this command built in). Observe, that all these operations are implemented in hardware in the CPU and require only constant time.

2.3.2 Performing operations on a (regular) quadtree stored using $\mathcal{Q}$-order

Let $\mathcal{T}$ be a given (regular) quadtree, with its nodes stored in an ordered-set data-structure (think about it as a sorted list), using the $\mathcal{Q}$-order over the cells. We next describe how to implement some basic operations on this quadtree.

2.3.2.1 Performing a point-location in a quadtree

Given a query point $\mathsf{q} \in [0,1]^2$, we would like to find the leaf v of $\mathcal{T}$ such that its cell contains q.

To answer the query, we first find the two consecutive cells in this ordered list such that q lies between them. Formally, let $\square$ be the last cell in this list such that $\square \prec \mathsf{q}$. It is now easy to verify that $\square$ must be the quadtree leaf containing q. Indeed, let $\square_{\mathsf{q}}$ be the leaf of $\mathcal{T}$ whose cell contains q. By definition, we have that $\square_{\mathsf{q}} \prec \mathsf{q}$. Thus, the only bad scenario is that $\square_{\mathsf{q}} \prec \square \prec \mathsf{q}$. But this implies, by the definition of $\mathcal{Q}$-order, that $\square$ must be contained inside $\square_{\mathsf{q}}$ contradicting our assumption that $\square_{\mathsf{q}}$ is a leaf of the quadtree.

Lemma 2.3.4 *Given a quadtree $\mathcal{T}$ of size n, with its leaves stored in an ordered-set data-structure $\mathcal{D}$ according to the $\mathcal{Q}$-order, then one can perform a point-location query in $O(Q(n))$ time, where $Q(n)$ is the time to perform a search query in $\mathcal{D}$.*

2.3.2.2 Overlaying two quadtrees

Given two quadtrees $\mathcal{T}'$ and $\mathcal{T}''$ we would like to overlay them to compute their combined quadtree. This is the minimal quadtree such that every cell of $\mathcal{T}'$ and $\mathcal{T}''$ appears in it. Observe that if the two quadtrees are given as sorted lists of their cells (ordered by the $\mathcal{Q}$-order) then their overlay is just the merged list, with replication removed.

Lemma 2.3.5 *Given two quadtrees $\mathcal{T}'$ and $\mathcal{T}''$, given as sorted lists of their nodes, one can compute the merged quadtree in linear time (in the total size of the lists representing them) by merging the two sorted lists and removing duplicates.*

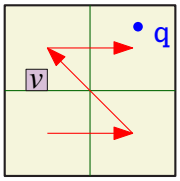
2.3.3 Performing operations on a compressed quadtree stored using $\mathcal{Q}$-order

Let $\mathcal{T}$ be a compressed quadtree whose nodes are stored in an ordered-set data-structure using the $\mathcal{Q}$-order. Note, that the nodes are sorted according to the canonical square that they corresponds to. As such, a node $v \in \mathcal{T}$ is sorted according to $\square_v$. The subtlety here is that a compressed node v, is stored according to the square $\square_v$ in this representation, but in fact it is in charge of the compressed region rg_v. This will make things slightly more complicated.

2.3.3.1 Point location in a compressed quadtree

Performing a point-location query is a bit subtle in this case. Indeed, if the query point q is contained inside a leaf of $\mathcal{T}$, then a simple binary search for the predecessor of q in the $\mathcal{Q}$-order sorted list of the cells of $\mathcal{T}$ would return this leaf.

However, if the query is contained inside the region of a compressed node, the predecessor to q in this list (sorted by the $\mathcal{Q}$-order), might be some arbitrary leaf that is contained inside the compressed node of interest. As a concrete example, consider the figure on the right of a compressed node. Because of the $\mathcal{Q}$-order, the predecessor query on q would return a leaf that is stored in the subtree of v.



In such a case, we need to find the lca of the query point and the leaf returned. This returned cell $\square$ would either be in the quadtree itself. In this case we are done as $\square$ corresponds to the compressed node that contains q. The other possibility is that $\square$ is contained inside the cell $\square_w$ of a compressed node w that is the parent of v. Again, we can now find this parent node using a single predecessor query.

The code for the point-location procedure is depicted on the right. The query point is q. The query point is not necessarily stored in the quadtree, and as such the cell that contains it might be a compressed node (as described above). As such, the required node v has $\mathsf{q} \in \mathsf{rg}_v$ and is either a leaf of the quadtree or a compressed node.

Lemma 2.3.6 *We have that: (i) if $\mathsf{q} \in \square_v$ then $\mathsf{q} \in \mathsf{rg}_v$, and (ii) if the region of $\mathcal{T}$ containing the query point q is a leaf then v is the required leaf.*

```
algPntLoc_Q-order(T, q).
  □_v = predecessor_{Q-order}(T, q).
  // □_v last node in T
  //      s.t.  □_v ≤ q.
  if q ∈ □_v then
      return v
  □ = lca(□_v, q)
  □_w = predecessor_{Q-order}(T, □).
  return w
```

Proof: (i) Otherwise, there exists a node $x \in \mathcal{T}$ such that $\mathsf{q} \in \square_x$ and $\square_x \subsetneq \square_v$, but that would imply that $\square_v \prec \square_x \prec \mathsf{q}$, contradicting how $\square_v$ was computed.

(ii) Since q is contained in a leaf, then the last square in the $\mathcal{Q}$-order that is before q is the cell of this leaf; that is, the node v. ■

Now, if $\mathsf{q} \notin \square_v$ then our search "failed" to find a node that contains the query point. So, consider the node w that should be returned. It must be a compressed node, and by the above, the search returned a node v that is a descendant of w. As a first step, we compute the square $\square = \mathsf{lca}(\square_v, \mathsf{q})$. Next, we compute the last cell $\square_u$ such that $\square_u \preceq \square$

Now, if $\square_u = \square$ then we are done as $\mathsf{q} \in \mathsf{rg}_u$. Otherwise, if $\square$ is not in the tree, then u is a compressed node, and consider its child u'. We have that $\square$ is contained inside $\square_u$ and contains $\square_{u'}$. Namely, $\mathsf{q} \in \square_u \setminus \square_{u'} = \mathsf{rg}_u$. Now, u' is the only child of u (since it is a compressed node), which implies that $\square_u$ and $\square_{u'}$ are consecutive in $\mathcal{T}$ according to the $\mathcal{Q}$-order, and $\square$ lies in between these two cells in this order. As such, $u \prec \square \prec u'$ and as such u is the required node that contains the query point.

We get the following result.

Lemma 2.3.7 *Given a compressed quadtree $\mathcal{T}$ of size n, with its leaves stored in an ordered-set data-structure $\mathcal{D}$ according to the $\mathcal{Q}$-order, then point-location queries can be performed using $\mathcal{T}$ in $O(Q(n))$ time, where $Q(n)$ is the time to perform a search query in $\mathcal{D}$.*

2.3.3.2 Insertions and deletions

Let q be a point to be inserted into the quadtree, and let w be the node of the compressed quadtree such that $\mathsf{q} \in \mathsf{rg}_w$. There are several possibilities:

1. The node w is a leaf, and there is no point associated with it. Then store p at w, and return.
2. The node w is a leaf, and there is a point p already stored in w. In this case, let $\square = \mathsf{lca}(\mathsf{p}, \mathsf{q})$, and insert $\square$ into the compressed quadtree. This is done by creating a new node z in the quadtree, with the cell of z being $\square$. We hang z below w if $\square \neq \square_w$ (this turns w into a compressed node). (If $\square = \square_w$ we do not need to introduce a new cell, and just set $z = w$.) Furthermore, split $\square$ into its children, and also insert the children into the compressed quadtree. Finally, associate p with the new leaf that contains it, and associate q with the leaf that contains it. Note, that because of the insertion w becomes a compressed node if $\square_w \neq \square$, and it becomes a regular internal node otherwise.
3. The node w is a compressed node. Let z be the child of w, and consider $\square = \mathsf{lca}(\square_z, \mathsf{q})$. Insert $\square$ into the compressed quadtree if $\square \neq \square_w$ (note that in this case w would still be a compressed node, but with a larger "hole"). Also insert all the children of $\square$ into the quadtree, and store p in the appropriate child. Hang $\square_z$ from the appropriate child, and turn this child into a compressed node.

In all three cases, the insertion requires a constant number of search/insert operations on the ordered-set data-structure.

Deletion is done in a similar fashion. We delete the point from the node that contains it, and then we trim away nodes that are no longer necessary.

Theorem 2.3.8 *Assuming one can compute the $\mathcal{Q}$-order in constant time, then one can maintain a compressed quadtree of a set of points in $O(\log n)$ time per operation, where insertions, deletions and point-location queries are supported. Furthermore, this can implemented using any data-structure for ordered-set that supports an operation in logarithmic time.*

2.3.4 Compressed quadtrees in high dimension

Naively, the constants used in the compressed quadtree are exponential in the dimension d. However, one can be more careful in the implementation. The first problem, for a node v in the compressed quadtree, is to store all the children of v in an efficient way so that we can access them efficiently. To this end, each child of v can be encoded as a binary string of length d, and we build a trie inside v for storing all the strings defining the children of v. It is easy, given a point, to figure out what the binary string encoding the child containing this point. As such, in $O(d)$ time, one can retrieve the relevant child. (One can also use hashing to this end, but it is not necessary here.)

Now, we build the compressed quadtree using the algorithm described above using $\mathcal{Q}$-order. It is not too hard to verify that all the basic operations can be computed in $O(d)$ time. Specifically, comparing two points in the $\mathcal{Q}$-order

takes $O(d)$ time. One technicality that matters when d is large (but this issue can be ignored in low dimensions) is that if a node has only a few children that are not empty, we create only these nodes, and not the other children. Putting everything together, we get the following result.

Theorem 2.3.9 *Assuming one can compute the $\mathcal{Q}$-order in $O(d)$ time for two points $\mathbb{R}^d$, then one can maintain a compressed quadtree of a set of points in $\mathbb{R}^d$, in $O(d \log n)$ time per operation. The operations insertion, deletion and point-location query are supported. Furthermore, this can implemented using any data-structure for ordered-set that supports an operation in logarithmic time.*

In particular, one can construct a compressed quadtree of a set of n points in $\mathbb{R}^d$ in $O(dn \log n)$ time.

2.4 Bibliographical notes

The authoritative text on quadtrees is the book by Samet [Sam89], he also has a more recent book that provides a comprehensive survey of various tree like data-structures [Sam05] (our treatment is naturally more theoretically oriented than his). The idea of using hashing in quadtrees is a variant of an idea due to Van Emde Boas, and is also used in performing fast lookup in IP routing (using PATRICIA tries which are one dimensional quadtrees [WVTP97]), among a lot of other applications.

The algorithm described, in Section 2.2.2, for the efficient construction of compressed quadtrees is new, as far as I know. The classical algorithms for computing compressed quadtrees efficiently achieve the same running time, but require considerably more careful implementation, and paying careful attention to details [CK95, AMN+98]. The idea of fingering a quadtree is from [AMN+98] (although their presentation is different than ours), but the idea is probably much older.

The idea of storing a quadtree in an ordered set by using the $\mathcal{Q}$-order on the nodes (or even only on the leaves) is due to Gargantini [Gar82], and it is referred to as ***linear quadtrees*** in the literature. The idea was used repeatedly for getting good performance in practice from quadtrees.

Our presentation of the dynamic quadtrees (i.e., Section 2.3) follows (very roughly) the work de Berg *et al.* [dHTT07].

It is maybe beneficial to emphasize that if one does not require the internal nodes of the compressed quadtree for the application, then one can avoid storing them in the data-structure. In fact, if one is only interested in the point themselves, then can even skip storing the leaves themselves, and then the compressed quadtree just becomes a data-structure that stores the points according to their $\mathcal{Z}$-order. This approach can be used for example to construct a data-structure for approximate nearest neighbor [Cha02] (however, this data-structure is still inferior, in practice, to the more optimized but more complicated data-structure of Arya *et al.* [AMN+98]). The author finds that thinking about such data-structures as compressed quadtrees (with the whole additional unnecessary information) is more intuitive, but the reader might disagree[③].

$\mathcal{Z}$-order and space filling curves. The idea of using $\mathcal{Z}$-order for speeding up spatial data-structures can be traced back to the above work of Gargantini [Gar82], and it is widely used in databases and seems to improve performance in practice [KF93]. The $\mathcal{Z}$-order can be viewed as a mapping from the unit interval to the unit-square, by splitting the odd bits, of a real number $\alpha \in [0, 1)$, to be the x-coordinate and the even bits of α to encode the y-coordinate of the mapped point. While this mapping is simple to define it is not continuous. Somewhat surprisingly one can find a continuous mapping that maps the unit interval to the unit-square. A large family of such mappings is known by now, see Sagan [Sag94] for an accessible book on the topic.

But is it really practical? Quadtrees seems to be widely used in practice and perform quite well. Compressed quadtrees seems to be less widely used, but they have the benefit of being much simpler than their relatives which seems to be more practical but theoretically equivalent.

Compressed quadtrees require strange operations. Lemma 2.2.6 might be new, although it seems natural to assume that it was known before. It implies that computing compressed quadtrees requires at least one "strange" operation in the computation model. Once one comes to term with this imperfect situation, the use of $\mathcal{Q}$-order seems

[③] The author reserves the right to disagree with himself on this topic in the future if the need arises.

natural and yields reasonably simple algorithm for dynamic maintenance of quadtrees. For example, if we maintain such a compressed quadtree by using skip-list on the $\mathcal{Q}$-order, we will essentially get the skip-quadtree of Eppstein *et al.* [EGS05].

Generalized Compressed quadtrees. Har-Peled and Mendel [HM06] have shown how to extend compressed quadtrees to the more general settings of doubling metrics. Note, that this variant of compressed quadtrees no longer requires strange bit operations. However, in the process, one loses the canonical grids structures that a compressed quadtree has, which is such a useful property.

Why compressed quadtrees? The reader might wonder why we are presenting compressed quadtrees when in practice people use different data-structures (that can also be analyzed). Our main motivation is that compressed quadtrees seems to be a universal data-structure, in the sense that they can be used to many different tasks, they are conceptually and algorithmically simple, and they provide a clear route to solving a problem: Solve your problem initially on a quadtree for a point set with bounded spread. If this works, try and solve it on a compressed quadtree. If you want something practical, try some more practical variants like *kd*-trees [dBCvKO08].

Good triangulations. Balanced quadtree and good triangulations are due to Bern *et al.* [BEG94]. The analysis we present uses ideas from the work of Ruppert [Rup95]. While using this approach to analyze [BEG94] is well known to people working in the field, I am unaware of a reference in the literature where it is presented in this way.

The problem of generating good triangulations had received considerable attention, as it is central to the problem of generating good meshes, which in turn are important for efficient numerical simulations of physical processes. One of the main techniques used in generating good triangulations is the method of Delaunay refinement. Here, one computes the Delaunay triangulation of the point set, and inserts circumscribed centers as new points, for "bad" triangles. Proving that this method converges and generates optimal triangulations is a non-trivial undertaking, and is due to Ruppert [Rup93]. Extending it to higher dimensions, and handling boundary conditions make it even more challenging. However, in practice, the Delaunay refinement method outperforms the (more elegant and simpler to analyze) method of Bern *et al.* [BEG94], which easily extends to higher dimensions. Namely, the Delaunay refinement method generates good meshes with fewer triangles.

Furthermore, Delaunay refinement methods are slower in theory. Getting an algorithm to perform Delaunay refinement in the same time as the algorithm of Bern *et al.* is still open, although Miller [Mil04] got an algorithm with only slightly slower running time.

Recently, Alper Üngör came up with a "Delaunay-refinement type" algorithm, which outputs better meshes than the classical Delaunay refinement algorithm [Üng09]. Furthermore, by merging the quadtree approach with the Üngör technique, one can get an optimal running time algorithm [HÜ05].

Chapter 3

Well Separated Pairs Decomposition

> The fact remains that getting people right is not what living is all about anyway. It's getting them wrong that is living, getting them wrong and wrong and wrong and then, on careful reconsideration, getting them wrong again. That's how we known we're alive: we're wrong. Maybe the best thing would be to forget being right or wrong about people and just go along for the ride. But if you can do that - well, lucky you.
>
> – American Pastoral, Philip Roth.

In this chapter, we will investigate how to represent distances between points efficiently. Naturally, an explicit description of the distances between n points requires listing all the $\binom{n}{2}$ distances. Here we will show that there is a considerably more compact representation which is sufficient if all we care about are approximate distances. This representation would have many nice applications.

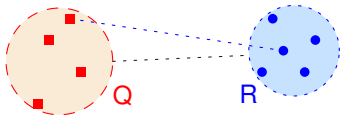
3.1 Well-separated pairs decomposition

Let P be a set of n points in $\mathbb{R}^{\mathsf{d}}$, and $1/4 > \varepsilon > 0$ a parameter. One can represent all distances between points of P by explicitly listing the $\binom{n}{2}$ pairwise distances. Of course, the listing of the coordinates of each point gives us an alternative more compact representation (of size dn), but its not a very informative representation. We are interested in a representation that will capture the structure of the distances between the points.

As a concrete example, consider the three points on the right. We would like to have a representation that captures that p has similar distance to q and s, and furthermore, q and s are close together as far as p is concerned. As such, if we are interested in the closest pair among the three points, we will only check the distance between q and s, since they are the only pair (among the three) that might realize the closest pair.

Figure 3.1:

Denote by $A \otimes B = \left\{ \{x, y\} \,\middle|\, x \in A,\ y \in B \right\}$ all the (unordered) pairs of points formed by the sets A and B. We will be informal and refer to $A \otimes B$ as a ***pair*** of the sets A and B. Here, are interested in schemes that cover all possible pairs of P by a small collection of such pairs.



Definition 3.1.1 (Pair decomposition.) For a point set P, a ***pair decomposition*** of P is a set of pairs

$$\mathcal{W} = \left\{ \{A_1, B_1\}, \ldots, \{A_s, B_s\} \right\},$$

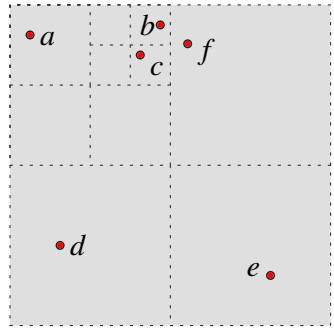
such that (I) $A_i, B_i \subset \mathsf{P}$ for every i, (II) $A_i \cap B_i = \emptyset$ for every i, and (III) $\cup_{i=1}^{s} A_i \otimes B_i = \mathsf{P} \otimes \mathsf{P}$.

Translation: For any pair of points $\mathsf{p}, \mathsf{q} \in \mathsf{P}$, there is at least one (and usually exactly one) pair $\{A_i, B_i\} \in \mathcal{W}$ such that $\mathsf{p} \in A_i$ and $\mathsf{q} \in B_i$.

Definition 3.1.2 The pair Q and R is $(1/\varepsilon)$***-separated*** if

$$\max(\mathrm{diam}(\mathsf{Q}), \mathrm{diam}(\mathsf{R})) \leq \varepsilon \cdot \mathbf{d}(\mathsf{Q}, \mathsf{R}),$$

where $\mathbf{d}(\mathsf{Q}, \mathsf{R}) = \min_{\mathsf{q} \in \mathsf{Q}, \mathsf{s} \in \mathsf{R}} \|\mathsf{q} - \mathsf{s}\|$.



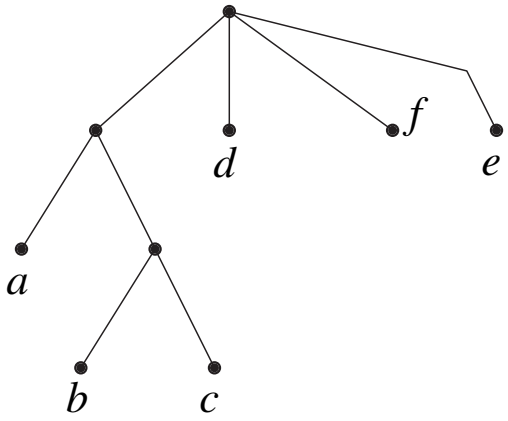
(i)

$$
\begin{aligned}
A_1 &= \{d\},\ B_1 = \{e\}\\
A_2 &= \{a,b,c\},\ B_2 = \{e\}\\
A_3 &= \{a,b,c\},\ B_3 = \{d\}\\
A_4 &= \{a\},\ B_4 = \{b,c\}\\
A_5 &= \{b\},\ B_5 = \{c\}\\
A_6 &= \{a\},\ B_6 = \{f\}\\
A_7 &= \{b\},\ B_7 = \{f\}\\
A_8 &= \{c\},\ B_8 = \{f\}\\
A_9 &= \{d\},\ B_9 = \{f\}\\
A_{10} &= \{e\},\ B_{10} = \{f\}
\end{aligned}
$$

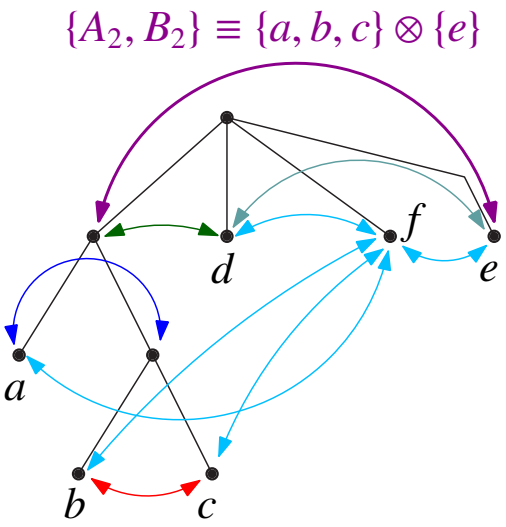
(ii)

$$
\mathcal{W} = \left\{ \begin{array}{c} \{A_1, B_1\}, \\ \{A_2, B_2\}, \\ \{A_3, B_3\}, \\ \{A_4, B_4\}, \\ \{A_5, B_5\}, \\ \{A_6, B_6\}, \\ \{A_7, B_7\}, \\ \{A_8, B_8\}, \\ \{A_9, B_9\}, \\ \{A_{10}, B_{10}\} \end{array} \right\}
$$

(iii)



(iv)



(v)

Figure 3.2: (i) A point set $\mathsf{P} = \{a, b, c, d, e, f\}$. (ii) The decomposition into pairs. (iii) The respective $(1/2)$-WSPD. For example, the pair of points b and e (and their distance) is represented by $\{A_2, B_2\}$ as $b \in A_2$ and $e \in B_2$. (iv) The quadtree $\mathcal{T}$ representing the point set P. (v) The WSPD as defined by pairs of vertices of $\mathcal{T}$.

Intuitively, the pair $\mathsf{Q} \otimes \mathsf{R}$ is $(1/\varepsilon)$-***separated*** if all the points of Q have roughly the same distance to the points of R. Alternatively, imagine covering the two point sets with two balls of minimum size, and require that the distance between the two balls is at least $2/\varepsilon$ times the radius of the larger of the two.

Thus, for the three points of Figure 3.1, the pairs $\{\mathsf{p}\} \otimes \{\mathsf{q}, \mathsf{s}\}$ and $\{\mathsf{q}\} \otimes \{\mathsf{s}\}$ are (say) 2-separated and describe all the distances among these three points. (The gain here is quite marginal, as we replaced the distance description, made out of three pairs of points, by distance between two pairs of sets. But stay tuned – exciting things are about to unfold.)

Motivated by the above example, a well-separated pair decomposition is a way to describe a metric by such "well separated" pairs of sets.

Definition 3.1.3 (WSPD) For a point set P, a ***well-separated pair decomposition*** (***WSPD***) of P with parameter $1/\varepsilon$ is a pair decomposition of P with a set of pairs

$$\mathcal{W} = \Big\{ \{A_1, B_1\}, \ldots, \{A_s, B_s\} \Big\},$$

such that, for any i, the sets A_i and B_i are ε^{-1}-separated.

For a concrete example of a WSPD, see Figure 3.2.

Instead of maintaining such a decomposition explicitly, it is convenient to construct a tree $\mathcal{T}$ having the points of P as leaves. Now every pair, (A_i, B_i) is just a pair of nodes (v_i, u_i) of $\mathcal{T}$, such that $A_i = \mathsf{P}_{v_i}$ and $B_i = \mathsf{P}_{u_i}$, where P_v denotes the points of P stored in the subtree of v (here v is a node of $\mathcal{T}$). Naturally, in our case, the tree we would use

is a compressed quadtree of P, but any tree that decomposes the points such that the diameter of a point set stored in a node drops quickly as we go down the tree might work. Naturally, even when the underlying tree is specified, there are many possible WSPDs that can be represented using this tree. Naturally, we will try find a WSPD that is "minimal".

This WSPD representation using a tree gives us a compact representation of the distances of the point set.

Corollary 3.1.4 *For a ε^{-1}-WSPD $\mathcal{W}$, it holds, for any pair $\{u, v\} \in \mathcal{W}$, that*

$$\forall \mathsf{q} \in \mathsf{P}_u, \mathsf{s} \in \mathsf{P}_v \qquad \max(\operatorname{diam}(\mathsf{P}_u), \operatorname{diam}(\mathsf{P}_v)) \leq \varepsilon \|\mathsf{q} - \mathsf{s}\|.$$

It would usually be convenient to associate with each set P_u in the WSPD, an arbitrary representative point $\operatorname{rep}_u \in \mathsf{P}$. Selecting and assigning these representative points can always be done by a simple DFS traversal of the $\mathcal{T}$ used to represent the WSPD.

3.1.1 The construction algorithm

Given the point set P in $\mathbb{R}^{\mathsf{d}}$, the algorithm first computes the compressed quadtree $\mathcal{T}$ of P. Next, the algorithm works by being greedy. It tries to put into the WSPD pairs of nodes in the tree that are as high as possible. In particular, if a pair $\{u, v\}$ would be generated than the pair formed by the parents of this pair of nodes will not be well separated. As such, the algorithm starts from the root, and tries to separate it from itself. If the current pair is not well separated, then we replace the bigger node of the pair by its children (i.e., thus replacing a single pair by several pairs). Clearly, sooner or later this refinement process would reach well-separated pairs, which it would output. Since it considers all possible distances up front (i.e., trying to separate the root from itself), it would generate a WSPD covering all pairs of points.

Let $\Delta(v)$ denote the diameter of the cell associated with a node v of the quadtree $\mathcal{T}$. We tweak this definition a bit so that if a node contains a single point or is empty then it is zero. This would make our algorithm easier to describe.

Formally, $\Delta(v) = 0$ if P_v is either empty or a single point. Otherwise, it is the diameter of the region associated with v; that is $\Delta(v) = \operatorname{diam}(\square_v)$, where $\square_v$ (we remind the reader) is the quadtree cell associated with the node v. Note, that since $\mathcal{T}$ is a compressed quadtree, we can always decide if $|\mathsf{P}_v| > 1$ by just checking if the subtree rooted at v has more than one node (since then this subtree must store more than one point).

We define the ***geometric distance*** between two nodes u and v of $\mathcal{T}$ to be

$$\mathbf{d}(u, v) = \mathbf{d}(\square_u, \square_v) = \min_{\mathsf{p} \in \square_u, \mathsf{q} \in \square_v} \|\mathsf{p} - \mathsf{q}\|.$$

We compute the compressed quadtree $\mathcal{T}$ of P in $O(n \log n)$ time. Next, we compute the WSPD by calling **algWSPD**$(u_0, u_0, \mathcal{T})$, where u_0 is the root of $\mathcal{T}$ and **algWSPD** is depicted in Figure 3.3.

algWSPD(u, v)
  if $\Delta(u) < \Delta(v)$ **then**
    Exchange u and v
  If $\Delta(u) \leq \varepsilon \cdot \mathbf{d}(u, v)$ **then**
    return $\left\{ \{u, v\} \right\}$

  // $u_1, \ldots, u_r$ - the children of u
  return $\bigcup_{i=1}^{r}$ **algWSPD**(u_i, v).

Figure 3.3: The algorithm **algWSPD** for computing well-separated pairs decomposition. The nodes u and v belong to a compressed quadtree $\mathcal{T}$ of P.

3.1.1.1 Analysis

The following lemma is implied by an easy packing argument.

Lemma 3.1.5 *Let $\square$ be a cell of a grid G of $\mathbb{R}^{\mathsf{d}}$ with cell diameter x. For $y \geq x$, the number of cells in G at distance at most y from $\square$ is $O\left((y/x)^{\mathsf{d}}\right)$. (The $O(\cdot)$ notation here, and in the rest of the chapter, hides a constant that depends exponentially on d.)*

Lemma 3.1.6 **algWSPD** *terminates and computes a valid pair-decomposition.*

Proof: By induction, it follows that every pair of points of P is covered by a pair of subsets $\{\mathsf{P}_u, \mathsf{P}_v\}$ output by the **algWSPD** algorithm. Note, that **algWSPD** always stops if both u and v are leafs, which implies that **algWSPD** always terminates.

Now, observe that if $\{u, v\}$ is in the output pair, and either P_u or P_v are not a single point, then $\alpha = \max(\operatorname{diam}(\mathsf{P}_u), \operatorname{diam}(\mathsf{P}_v)) > 0$. This implies that $\mathbf{d}(\mathsf{P}_u, \mathsf{P}_v) \geq \mathbf{d}(\square_u, \square_v) \geq (\alpha/\varepsilon) > 0$. Namely, $\mathsf{P}_u \cap \mathsf{P}_v = \emptyset$. ■

Lemma 3.1.7 *For the WSPD generated by* **algWSPD**, *we have that for any pair $\{u, v\}$ in the WSPD, it holds*

$$\max(\operatorname{diam}(\mathsf{P}_u), \operatorname{diam}(\mathsf{P}_v)) \leq \varepsilon \cdot \mathbf{d}(u, v) \quad \text{and} \quad \mathbf{d}(u, v) \leq \|\mathsf{q} - \mathsf{s}\|,$$

for any $\mathsf{q} \in \mathsf{P}_u$ and $\mathsf{s} \in \mathsf{P}_v$.

Proof: For every output pair $\{u, v\}$, we have by the design of the algorithm that

$$\max\{\operatorname{diam}(\mathsf{P}_u), \operatorname{diam}(\mathsf{P}_v)\} \quad \leq \quad \max\{\Delta(u), \Delta(v)\} \leq \varepsilon \cdot \mathbf{d}(u, v).$$

Also, for any $\mathsf{q} \in \mathsf{P}_u$ and $\mathsf{s} \in \mathsf{P}_v$, we have $\mathbf{d}(u, v) = \mathbf{d}(\square_u, \square_v) \leq \mathbf{d}(\mathsf{P}_u, \mathsf{P}_v) \leq \mathbf{d}(\mathsf{q}, \mathsf{s})$, since $\mathsf{P}_u \subseteq \square_u$ and $\mathsf{P}_v \subseteq \square_v$. ■

Note, that our algorithm will generate pairs separating every single point of P from itself. Such pairs are not interesting and can be thrown away. This is not explicitly checked for in the pseudo-code of Figure 3.3 to keep it conceptually simple.

Lemma 3.1.8 *For a pair $\{u, v\} \in \mathcal{W}$ computed by* **algWSPD**, *we have that*

$$\max\Big(\Delta(u),\ \Delta(v)\Big) \leq \min\Big(\Delta(\overline{\mathrm{p}}(u)),\ \Delta(\overline{\mathrm{p}}(v))\Big),$$

where $\overline{\mathrm{p}}(x)$ denotes the parent node of the node x in the tree $\mathcal{T}$.

Proof: We trivially have that $\Delta(u) < \Delta(\overline{\mathrm{p}}(u))$ and $\Delta(v) < \Delta(\overline{\mathrm{p}}(v))$.

A pair $\{u, v\}$ is generated because of a sequence of recursive calls **algWSPD**(u_0, u_0), **algWSPD**(u_1, v_1), ..., **algWSPD**(u_s, v_s), where $u_s = u$, $v_s = v$, and u_0 is the root of $\mathcal{T}$. Assume that $u_{s-1} = u$ and $v_{s-1} = \overline{\mathrm{p}}(v)$. Then $\Delta(u) \leq \Delta(\overline{\mathrm{p}}(v))$, since the algorithm always refines the larger cell.

Similarly, let t be the last index such that $u_{t-1} = \overline{\mathrm{p}}(u)$ (namely, $u_{t-1} \neq u_t = u$ and $v_{t-1} = v_t$). Then, since v is a descendant of v_{t-1}, it holds that

$$\Delta(v) \leq \Delta(v_t) = \Delta(v_{t-1}) \leq \Delta(u_{t-1}) = \Delta(\overline{\mathrm{p}}(u)),$$

since (again) the algorithm always refines the larger cell in the pair $\{u_{t-1}, v_{t-1}\}$. ■

Lemma 3.1.9 *The number of pairs in the computed WSPD is $O(n/\varepsilon^{\mathsf{d}})$.*

Proof: Let $\{u, v\}$ be a pair appearing in the output. Consider the sequence (i.e., stack) of recursive calls that led to this output. In particular, assume that the last recursive call to **algWSPD**(u, v) was issued by **algWSPD**(u, v'), where $v' = \overline{\mathrm{p}}(v)$ is the parent of v in $\mathcal{T}$. Then

$$\Delta(\overline{\mathrm{p}}(u)) \geq \Delta(v') \geq \Delta(u),$$

by Lemma 3.1.8.

We charge the pair $\{u, v\}$ to the node v', and claim that each node of $\mathcal{T}$ is charged at most $O(\varepsilon^{-d})$ times. To this end, fix a node $v' \in V(\mathcal{T})$, where $V(\mathcal{T})$ is the set of vertices of $\mathcal{T}$. Since the pair $\{u, v'\}$ was not output by **algWSPD** (despite being considered) we conclude that $\Delta(v') > \varepsilon \cdot \mathbf{d}(u, v')$ and as such $\mathbf{d}(u, v') < r = \Delta(v')/\varepsilon$. Now, there are several possibilities:

(i) $\Delta(v') = \Delta(u)$. But there are at most $O\Big((r/\Delta(v'))^{\mathsf{d}}\Big) = O(1/\varepsilon^{\mathsf{d}})$ nodes that have the same level (i.e., diameter) as v' such that their cells are in distance at most r from it, by Lemma 3.1.5. Thus, this type of charge can happen at most $O(2^{\mathsf{d}} \cdot (1/\varepsilon^{\mathsf{d}}))$ times, since v' has at most 2^{d} children.

(ii) $\Delta(\overline{\mathrm{p}}(u)) = \Delta(v')$. By the same argumentation as above $\mathbf{d}(\overline{\mathrm{p}}(u), v') \leq \mathbf{d}(u, v') < r$. There are at most $O(1/\varepsilon^{\mathsf{d}})$ such nodes $\overline{\mathrm{p}}(u)$. Since the node $\overline{\mathrm{p}}(u)$ has at most 2^{d} children, it follows that the number of such charges is at most $O\Big(2^{\mathsf{d}} \cdot 2^{\mathsf{d}} \cdot (1/\varepsilon^{\mathsf{d}})\Big)$.

(iii) $\Delta(\overline{\mathsf{p}}(u)) > \Delta(v') > \Delta(u)$. Consider the canonical grid G having $\square_{v'}$ as one of its cells (see Definition 2.1.1). Let $\widehat{\square}$ be the cell in G containing $\square_u$. Observe that $\square_u \subsetneq \widehat{\square} \subsetneq \square_{\overline{\mathsf{p}}(u)}$. In addition, $\mathbf{d}\left(\widehat{\square}, \square_{v'}\right) \leq \mathbf{d}(\square_u, \square_{v'}) = \mathbf{d}(u, v') < r$. It follows that are at most $O(1/\varepsilon^{\mathsf{d}})$ cells like $\widehat{\square}$ that might participate in charging v', and as such, the total number of charges is $O(2^{\mathsf{d}}/\varepsilon^{\mathsf{d}})$, as claimed.

As such, v' can be charged at most $O\left(2^{2d}/\varepsilon^{\mathsf{d}}\right) = O\left(1/\varepsilon^{\mathsf{d}}\right)$ times[4]. This implies that the total number of pairs generated by the algorithm is $O(n\varepsilon^{-d})$, since the number of nodes in $\mathcal{T}$ is $O(n)$. ■

Since the running time of **algWSPD** is clearly linear in the output size, we have the following result.

Theorem 3.1.10 *For $1 \geq \varepsilon > 0$, one can construct a ε^{-1}-WSPD of size $n\varepsilon^{-d}$, and the construction time is $O\left(n \log n + n\varepsilon^{-d}\right)$.*

3.2 Applications of WSPD

3.2.1 Spanners

It is sometime beneficial to describe distances between n points by using a graph to encode the distances.

Definition 3.2.1 For a weighted graph G, and any two vertices p and q of G, we will denote by $\mathsf{d}_{\mathsf{G}}(\mathsf{q}, \mathsf{s})$ the ***graph distance*** between q and s. Formally, $\mathsf{d}_{\mathsf{G}}(\mathsf{q}, \mathsf{s})$ is the length of the shortest path in G between q and s. It is easy to verify that d_{G} is a metric; that is, it complies with the triangle inequality. Naturally, if q and s belongs to two different connected components of G then $\mathsf{d}_{\mathsf{G}}(\mathsf{q}, \mathsf{s}) = \infty$.

Such a graph might be useful algorithmically if it captures the distances we are interested in while being sparse (i.e., having few edges). In particular, if such a graph G over n vertices has only $O(n)$ edges, then it can be manipulated efficiently, and it is a compact implicit representation of the $\binom{n}{2}$ distances between all the pairs of vertices of G.

A ***t-spanner*** of a set of points $\mathsf{P} \subset \mathbb{R}^{\mathsf{d}}$ is a weighted graph G whose vertices are the points of P, and for any $\mathsf{q}, \mathsf{s} \in \mathsf{P}$, we have

$$\|\mathsf{q} - \mathsf{s}\| \leq \mathsf{d}_{\mathsf{G}}(\mathsf{q}, \mathsf{s}) \leq t \, \|\mathsf{q} - \mathsf{s}\| ,$$

The ratio $\mathsf{d}_{\mathsf{G}}(\mathsf{q}, \mathsf{s}) / \|\mathsf{q} - \mathsf{s}\|$ is the ***stretch*** of q and s in G. The ***stretch*** of G is the maximum stretch of any pair of points of P.

3.2.1.1 Construction

We are given a set of n points set $\mathbb{R}^{\mathsf{d}}$, and parameter $1 \geq \varepsilon > 0$. We will construct a spanner as follows.

Let $c = 9$ be an arbitrary constant, and set $\delta = \varepsilon/c$. Compute a δ^{-1}-WSPD decomposition using the algorithm of Theorem 3.1.10. For any vertex u in the quadtree $\mathcal{T}$ (used in computing the WSPD), let rep_u be an arbitrary point of P_u. For every pair $\{u, v\} \in \mathcal{W}$, add an edge between $\{\mathrm{rep}_u, \mathrm{rep}_v\}$ with weight $\left\|\mathrm{rep}_u - \mathrm{rep}_v\right\|$, and let G be the resulting graph.

One can prove that the resulting graph is connected, and contains all the points of P. We will not prove this explicitly as this is implied by the analysis below.

3.2.1.2 Analysis

Observe, that by the triangle inequality, we have that $\mathsf{d}_{\mathsf{G}}(\mathsf{q}, \mathsf{s}) \geq \|\mathsf{q} - \mathsf{s}\|$, for any $\mathsf{q}, \mathsf{s} \in \mathsf{P}$.

Theorem 3.2.2 *Given a set P of n points in $\mathbb{R}^{\mathsf{d}}$, and a parameter $1 \geq \varepsilon > 0$, one can compute a $(1+\varepsilon)$-spanner of P with $O(n\varepsilon^{-d})$ edges, in $O\left(n \log n + n\varepsilon^{-d}\right)$ time.*

[4] We remind the reader that we will usually consider the dimension d to be a constant, and the O notation would happily consume any constants that depend only on d. Conceptually, you can think about the O as being a black hole for such constants, as its gravitational force tear such constants away. The shrieks of horror of these constant as they are being swallowed alive by the black hOle can be heard every time you look at the O.

Proof: The construction is described above. The upper bound on the stretch is proved by induction on the length of pairs in the WSPD. So, fix a pair $x, y \in \mathsf{P}$, and assume that by the induction hypothesis, that for any pair $z, w \in \mathsf{P}$ such that $\|z - w\| < \|x - y\|$, it holds $\mathbf{d}_{\mathsf{G}}(z, w) \leq (1 + \varepsilon)\|z - w\|$.

The pair x, y must appear in some pair $\{u, v\} \in \mathcal{W}$, where $x \in \mathsf{P}_u$, and $y \in \mathsf{P}_v$. Thus

$$\left\|\mathrm{rep}_u - \mathrm{rep}_v\right\| \leq \mathbf{d}(u, v) + \Delta(u) + \Delta(v) \leq (1 + 2\delta)\|x - y\|$$

and

$$\begin{aligned}\max\left(\left\|\mathrm{rep}_u - x\right\|, \left\|\mathrm{rep}_v - y\right\|\right) &\leq \max(\Delta(u), \Delta(v)) \leq \delta \cdot \mathbf{d}(u, v) \leq \delta\left\|\mathrm{rep}_u - \mathrm{rep}_v\right\| \\ &\leq \delta(1 + 2\delta)\|x - y\| < \frac{1}{4}\|x - y\|,\end{aligned}$$

by Theorem 3.1.10 and since $\delta \leq 1/16$. As such, we can apply the induction hypothesis to $\mathrm{rep}_u x$ and $\mathrm{rep}_v y$, implying that

$$\mathsf{d}_{\mathsf{G}}(x, \mathrm{rep}_u) \leq (1 + \varepsilon)\left\|\mathrm{rep}_u - x\right\| \quad \text{and} \quad \mathsf{d}_{\mathsf{G}}(\mathrm{rep}_v, y) \leq (1 + \varepsilon)\left\|y - \mathrm{rep}_v\right\|.$$

Now, since $\mathrm{rep}_u\mathrm{rep}_v$ is an edge of G, it holds $\mathsf{d}_{\mathsf{G}}(\mathrm{rep}_u, \mathrm{rep}_v) \leq \left\|\mathrm{rep}_u - \mathrm{rep}_v\right\|$. Thus, by the inductive hypothesis and the triangle inequality, we have that

$$\begin{aligned}\|x - y\| &\leq \mathsf{d}_{\mathsf{G}}(x, y) \leq \mathsf{d}_{\mathsf{G}}(x, \mathrm{rep}_u) + \mathsf{d}_{\mathsf{G}}(\mathrm{rep}_u, \mathrm{rep}_v) + \mathsf{d}_{\mathsf{G}}(\mathrm{rep}_v, y) \\ &\leq (1 + \varepsilon)\left\|\mathrm{rep}_u - x\right\| + \left\|\mathrm{rep}_u - \mathrm{rep}_v\right\| + (1 + \varepsilon)\left\|\mathrm{rep}_v - y\right\| \\ &\leq 2(1 + \varepsilon) \cdot \delta \cdot \left\|\mathrm{rep}_u - \mathrm{rep}_v\right\| + \left\|\mathrm{rep}_u - \mathrm{rep}_v\right\| \\ &\leq (1 + 2\delta + 2\varepsilon\delta)\left\|\mathrm{rep}_u - \mathrm{rep}_v\right\| \\ &\leq (1 + 2\delta + 2\varepsilon\delta)(1 + \delta)\|x - y\| \\ &\leq (1 + \varepsilon)\|x - y\|.\end{aligned}$$

The last step follows by an easy calculation. Indeed, since $c = 9$ and $c\delta = \varepsilon \leq 1$, we have that

$$(1 + 2\delta + 2\varepsilon\delta)(1 + \delta) \leq (1 + 4\delta)(1 + \delta) = 1 + 5\delta + 4\delta^2 \leq 1 + 9\delta \leq 1 + \varepsilon,$$

as required. ■

3.2.2 Approximating the Minimum Spanning Tree

For a graph G, let $\mathsf{G}_{\leq r}$ denote the subgraph of G resulting from removing all the edges of weight (strictly) larger than r from G.

Lemma 3.2.3 *Given a set P of n points in $\mathbb{R}^{\mathsf{d}}$, one can compute a spanning tree $\mathcal{T}$ of P, such that $\omega(\mathcal{T}) \leq (1+\varepsilon)\omega(\mathcal{M})$, where $\mathcal{M}$ is a minimum spanning tree of P, and $\omega(\mathcal{T})$ is the total weight of the edges of $\mathcal{T}$. This takes $O\left(n \log n + n\varepsilon^{-d}\right)$ time.*

In fact, for any $r \geq 0$ and a connected component C of $\mathcal{M}_{\leq r}$, the set C is contained in a connected component of $\mathcal{T}_{\leq (1+\varepsilon)r}$.

Proof: Compute a $(1 + \varepsilon)$-spanner G of P and let $\mathcal{T}$ be a minimum spanning tree of G. We output the tree $\mathcal{T}$ as the approximate minimum spanning tree. Clearly the time to compute $\mathcal{T}$ is $O(n \log n + n\varepsilon^{-d})$, since the MST of a graph, with n vertices and m edges, can be computed in $O(n \log n + m)$ time.

We remain the task of proving that $\mathcal{T}$ is the required approximation. For any $\mathsf{q}, \mathsf{s} \in \mathsf{P}$, let π_{qs} denote the shortest path between q and s in G. Since G is a $(1 + \varepsilon)$-spanner, we have that $w\left(\pi_{\mathsf{qs}}\right) \leq (1 + \varepsilon)\|\mathsf{q} - \mathsf{s}\|$, where $w\left(\pi_{\mathsf{qs}}\right)$ denote the weight of π_{qs} in G. We have that $G' = (P, E)$ is a connected subgraph of G, where

$$E = \bigcup_{\mathsf{qs} \in E(\mathcal{M})} \pi_{uv},$$

where $E(\mathcal{M})$ denotes the set of edges of the graph $\mathcal{M}$. Furthermore,

$$\omega(G') = \sum_{(\mathsf{q},\mathsf{s})\in\mathcal{M}} w\left(\pi_{\mathsf{qs}}\right) \leq \sum_{(\mathsf{q},\mathsf{s})\in\mathcal{M}} (1+\varepsilon)\,\|\mathsf{q}-\mathsf{s}\| = (1+\varepsilon)\omega(\mathcal{M})\,,$$

since G is a $(1+\varepsilon)$-spanner. It thus follows that $\omega(\mathcal{T}) \leq \omega(\mathcal{M}(G')) \leq \omega(G') \leq (1+\varepsilon)w(\mathcal{M})$, where $\mathcal{M}(G')$ is the minimum spanning tree of G'.

The second claim follows by similar argumentation. ■

3.2.3 Approximating the Diameter

Lemma 3.2.4 *Given a set P of n points in $\mathbb{R}^{\mathsf{d}}$, one can compute, in $O\left(n\log n + n\varepsilon^{-d}\right)$ time, a pair $\mathsf{p},\mathsf{q}\in\mathsf{P}$, such that $\|\mathsf{p}-\mathsf{q}\| \geq (1-\varepsilon)\mathrm{diam}(\mathsf{P})$, where $\mathrm{diam}(\mathsf{P})$ is the diameter of P.*

Proof: Compute a $(4/\varepsilon)$-WSPD $\mathcal{W}$ of P. As before, we assign for each node u of $\mathcal{T}$ an arbitrary representative point that belongs to P_u. This can be done in linear time. Next, for each pair of $\mathcal{W}$, compute the distance of its representative points (for each pair, this takes constant time). Let $\{x,y\}$ be the pair in the $\mathcal{W}$ such that distance between its two Representative points is maximal, and return $\mathsf{p} = \mathrm{rep}_x$ and $\mathsf{q} = \mathrm{rep}y$ as the two points realizing the approximation. Overall, this takes $O(n\log n + n\varepsilon^{-d})$ time, since $|\mathcal{W}| = O(n/\varepsilon^{\mathsf{d}})$..

To see why it works, consider the pair $\mathsf{q},\mathsf{s}\in\mathsf{P}$ realizing the diameter of P, and let $\{u,v\}\in\mathcal{W}$ be the pair in the WSPD that contain the two points, respectively (i.e., $\mathsf{q}\in\mathsf{P}_u$ and $\mathsf{s}\in\mathsf{P}_v$). We have that

$$\begin{aligned}\left\|\mathrm{rep}_u - \mathrm{rep}_v\right\| &\geq \mathbf{d}(u,v) \geq \|\mathsf{q}-\mathsf{s}\| - \mathrm{diam}(\mathsf{P}_u) - \mathrm{diam}(\mathsf{P}_v)\\ &\geq (1-2(\varepsilon/2))\,\|\mathsf{q}-\mathsf{s}\| = (1-\varepsilon)\mathrm{diam}(\mathsf{P})\,,\end{aligned}$$

since, by Corollary 3.1.4, $\max(\mathrm{diam}(\mathsf{P}_u),\mathrm{diam}(\mathsf{P}_v)) \leq 2(\varepsilon/4)\,\|\mathsf{q}-\mathsf{s}\|$. Namely, the distance of the two points output by the algorithm is at least $(1-\varepsilon)\mathrm{diam}(\mathsf{P})$. ■

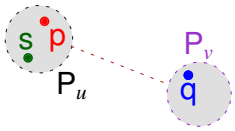
3.2.4 Closest Pair

Let P be a set of points in $\mathbb{R}^{\mathsf{d}}$. We would like to compute the ***closest pair***; namely, the two points closest to each other in P.

We need the following observation.

Lemma 3.2.5 *Let $\mathcal{W}$ be a ε^{-1}-WSPD of P, for $\varepsilon \leq 1/2$. There exists a pair $\{u,v\}\in\mathcal{W}$, such that (i) $|\mathsf{P}_u| = |\mathsf{P}_v| = 1$, and (ii) $\left\|\mathrm{rep}_u - \mathrm{rep}_v\right\|$ is the length of the closest pair, where $\mathsf{P}_u = \{\mathrm{rep}_u\}$ and $\mathsf{P}_v = \{\mathrm{rep}_v\}$.*

Proof: Consider the pair of closest points p and q in P, and consider the pair $\{u,v\}\in\mathcal{W}$, such that $\mathsf{p}\in\mathsf{P}_u$ and $\mathsf{q}\in\mathsf{P}_v$. If P_u contains an additional point $\mathsf{s}\in\mathsf{P}_u$, then we have that



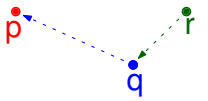
$$\|\mathsf{p}-\mathsf{s}\| \leq \mathrm{diam}(\mathsf{P}_u) \leq \varepsilon\cdot\mathbf{d}(u,v) \leq \varepsilon\,\|\mathsf{p}-\mathsf{q}\| < \|\mathsf{p}-\mathsf{q}\|\,,$$

by Theorem 3.1.10 and since $\varepsilon = 1/2$. Thus, $\|\mathsf{p}-\mathsf{s}\| < \|\mathsf{p}-\mathsf{q}\|$, a contradiction to the choice of p and q as the closest pair. Thus, $|\mathsf{P}_u| = |\mathsf{P}_v| = 1$ and $\mathrm{rep}_u = \mathsf{q}$ and $\mathrm{rep}_v = \mathsf{s}$. ■

Algorithm. Compute a ε^{-1}-WSPD $\mathcal{W}$ of P, for $\varepsilon = 1/2$. Next, scan all the pairs of $\mathcal{W}$, and compute for all the pairs $\{u,v\}$ which connect singletons (i.e., $|\mathsf{P}_u| = |\mathsf{P}_v| = 1$), the distance between their representatives rep_u and rep_v. The algorithm returns the closest pair of points encountered.

Theorem 3.2.6 *Given a set P of n points in $\mathbb{R}^{\mathsf{d}}$, one can compute the closest pair of points of P in $O(n\log n)$ time.*

We remind the reader that we already saw a linear (expected) time algorithm for this problem in Section 1.2. However, this is a deterministic algorithm, and it can be applied in more abstract settings where a small WSPD still exists, while the previous algorithm would not work.



3.2.5 All Nearest Neighbors

Given a set P of n points in $\mathbb{R}^d$, we would like to compute for each point $\mathsf{q} \in \mathsf{P}$, its ***nearest neighbor*** in P (formally, this is the closest point in $\mathsf{P} \setminus \{\mathsf{q}\}$ to q). This is harder than it might seem at first, since this is *not* a symmetrical relationship. Indeed, in the figure on the right, q might be the nearest neighbor to p, but s might be the nearest neighbor to q.

3.2.5.1 The bounded spread case

Algorithm. Assume P is contained in the unit square, and $\operatorname{diam}(\mathsf{P}) \geq 1/4$. Furthermore, let $\Phi = \Phi(P)$ denote the spread of P. Compute a ε^{-1}-WSPD $\mathcal{W}$ of P, for $\varepsilon = 1/4$.

Scan all the pairs $\{u, v\}$ with a singleton as one of their sides (i.e., $|P_u| = 1$), and for each such singleton $\mathsf{P}_u = \{\mathsf{p}\}$, record for p the closest point to it in the set P_v. Maintain for each point the closest point to it that was encountered.

We claim is that in the of this process, of every point in P its recorded nearest point encountered is indeed its nearest neighbor in P.

Analysis. The analysis of this algorithm is slightly tedious, but it reveals some additional interesting properties of WSPD. We start with a claim that shows that we will indeed find the nearest neighbor for each point.

Lemma 3.2.7 *Let p be any point of P, and let q be the nearest neighbor to p in the set $\mathsf{P} \setminus \{\mathsf{p}\}$. Then, there exists a pair $\{u, v\} \in \mathcal{W}$, such that $\mathsf{P}_u = \{\mathsf{p}\}$ and $\mathsf{q} \in \mathsf{P}_v$.*

Proof: Consider the pair $\{u, v\} \in \mathcal{W}$ such that $\mathsf{pq} \in \mathsf{P}_u \otimes \mathsf{P}_v$, where $\mathsf{p} \in \mathsf{P}_u$ and $\mathsf{q} \in \mathsf{P}_v$. However, $\operatorname{diam}(\mathsf{P}_v) \leq \varepsilon \mathbf{d}(\mathsf{P}_u, \mathsf{P}_v) \leq \varepsilon \|\mathsf{p} - \mathsf{q}\| \leq \|\mathsf{p} - \mathsf{q}\|/4$. Namely, if P_v contained any other point except p then q would not be the nearest neighbor to p. ■

Thus, the above lemma implies that the algorithm will find the nearest neighbor for each point p of P, as the appropriate pair containing only p on one side would be considered by the algorithm. We remain with the task of bounding the running time.

A pair of nodes $\{x, y\}$ of $\mathcal{T}$ is a ***generator*** of a pair $\{u, v\} \in \mathcal{W}$ if $\{u, v\}$ was computed inside a recursive call **algWSPD**(x, y).

Lemma 3.2.8 *Let $\mathcal{W}$ be a ε^{-1}-WSPD of a point set P generated by* **algWSPD**. *Consider a pair $\{u, v\} \in \mathcal{W}$, then $\Delta(\overline{\mathsf{p}}(v)) \geq (\varepsilon/2)\mathbf{d}(u, v)$ and $\Delta(\overline{\mathsf{p}}(u)) \geq (\varepsilon/2)\mathbf{d}(u, v)$, where $\mathbf{d}(u, v) = \mathbf{d}(\square_u, \square_v)$ is the distance between the cell of u and the cell of v.*

Proof: Assume, for the sake of contradiction, that $\Delta(v') < (\varepsilon/2)\ell$, where $\ell = \mathbf{d}(u, v)$ and $v' = \overline{\mathsf{p}}(v)$. By Lemma 3.1.8, we have that

$$\Delta(u) \leq \Delta(v') < \varepsilon \frac{\ell}{2}.$$

But then

$$\mathbf{d}(u, v') \geq \ell - \Delta(v') \geq \ell - \varepsilon \frac{\ell}{2} \geq \frac{\ell}{2}.$$

Thus,

$$\max(\Delta(u), \Delta(v')) < \varepsilon \frac{\ell}{2} \leq \varepsilon\, \mathbf{d}(u, v').$$

Namely, u and v' are well-separated, and as such $\{u, v'\}$ can not be a generator of $\{u, v\}$. Indeed, if $\{u, v'\}$ was considered by the algorithm than it would have added it to the WSPD, and never created the pair $\{u, v\}$.

So, the other possibility is that $\{u', v\}$ is the generator of $\{u, v\}$, where $u' = \overline{\mathsf{p}}(u)$. But then $\Delta(u') \leq \Delta(v') < \varepsilon\ell/2$, by Lemma 3.1.8. Using the same argumentation as above, we have that $\{u', v\}$ is a well-separated pair and as such it can not be a generator of $\{u, v\}$.

But this implies that $\{u, v\}$ can not be generated by **algWSPD**, since either $\{u, v'\}$ or $\{u', v\}$ must be a generator of $\{u, v\}$. A contradiction. ■

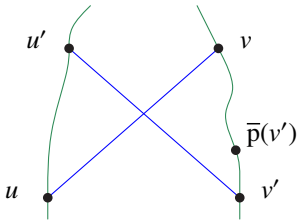
Claim 3.2.9 *For two pairs $\{u,v\},\{u',v'\} \in \mathcal{W}$ such that $\square_u \subseteq \square_{u'}$, it holds that the interiors of $\square_v$ and $\square_{v'}$ are disjoint.*

Proof: Since $\square_u \subseteq \square_{u'}$ it follows that u' is an ancestor of u.

If v' is an ancestor of v then **algWSPD** returned the pair $\{u',v'\}$ and it would have never generated the pair $\{u,v\}$.

If v is an ancestor of v' (see figure on the right) then

$$\begin{aligned}
\Delta(u) &< \Delta(u') && (u' \text{ is an ancestor of } u)\\
&\leq \Delta(\overline{\mathsf{p}}(v')) && (\text{by Lemma 3.1.8 applied to } \{u',v'\})\\
&\leq \Delta(v) && (v \text{ is an ancestor of } v')\\
&\leq \Delta(\overline{\mathsf{p}}(u)) && (\text{by Lemma 3.1.8 applied to } \{u,v\})\\
&\leq \Delta(u') && (u' \text{ is an ancestor of } u).
\end{aligned}$$



Namely, $\Delta(v) = \Delta(u')$. But then, the pair $\{u',v\}$ is a generator of both $\{u,v\}$ and $\{u',v'\}$. To see that, observe that **algWSPD** always try to consider pairs that the same diameter (it always split the bigger side of a pair). As such, for the algorithm to generate the pair $\{u,v\}$ it must have had a generator u'',v, such that $\operatorname{diam}(u'') \geq \operatorname{diam}(v)$. But the lowest ancestor of u that has this property is u'.

Now, when **algWSPD** considered the pair $\{u',v\}$ it split one of its sides (by calling on its children). In either case, it either did not create the pair $\{u,v\}$ or it did not create the pair $\{u',v'\}$. A contradiction.

The case $v = v'$ is handled in a similar fashion. ■

Lemma 3.2.10 *Let P be a set n points in $\mathbb{R}^{\mathsf{d}}$, $\mathcal{W}$ a ε^{-1}-WSPD of P, $\ell > 0$ be a distance, and W be the set of pairs $\{u,v\} \in \mathcal{W}$ such that $\ell \leq \mathbf{d}(u,v) \leq 2\ell$. Then, for any point $\mathsf{p} \in \mathsf{P}$, the number of pairs in W containing p is $O(1/\varepsilon^{\mathsf{d}})$.*

Proof: Let u be the leaf of the quadtree $\mathcal{T}$ (that is used in computing $\mathcal{W}$) storing the point p, and let π be the path between u and the root of $\mathcal{T}$. We claim that W contains at most $O(1/\varepsilon^{\mathsf{d}})$ pairs with nodes that appears along π. Let

$$T = \left\{ v \;\middle|\; v \in \pi, \{u,v\} \in W \right\}.$$

The cells of T are interior disjoint by Claim 3.2.9, and they contain all the pairs in W that covers p.

So, let r be the largest power of two which is smaller than (say) $\varepsilon\ell/(4\sqrt{d})$. Clearly, there are $O(1/\varepsilon^{\mathsf{d}})$ cells of G_r in distance at most 2ℓ from $\square_u$. We account for the nodes $v \in T$, as follows:

(i) If $\Delta(v) \geq r\sqrt{d}$ then $\square_v$ contains a cell of G_r, and there are at most $O(1/\varepsilon^{\mathsf{d}})$ such cells.

(ii) If $\Delta(v) < r\sqrt{d}$ and $\Delta(\overline{\mathsf{p}}(v)) \geq r\sqrt{d}$, then:

  (a) If $\overline{\mathsf{p}}(v)$ is a compressed node, then $\overline{\mathsf{p}}(v)$ contains a cell of G_r and it has only v as a single child. As such, there are most $O(1/\varepsilon^{\mathsf{d}})$ such charges.

  (b) Otherwise, $\overline{\mathsf{p}}(v)$ is not compressed, but then $\operatorname{diam}(\square_v) = \operatorname{diam}\left(\square_{\overline{\mathsf{p}}(v)}\right)/2$. As such $\square_v$ contains a cell of $\mathsf{G}_{r/2}$ in distance at most 2ℓ from $\square_u$, and there are $O(1/\varepsilon^{\mathsf{d}})$ such cells.

(iii) The case $\Delta(\overline{\mathsf{p}}(v)) < r\sqrt{d}$ is impossible. Indeed, by Lemma 3.1.8, we have $\Delta(\overline{\mathsf{p}}(v)) < r\sqrt{d} \leq \varepsilon\ell/4 = \frac{\varepsilon}{4}\mathbf{d}(u,v)$, a contradiction to Lemma 3.2.8.

We conclude that there are at most $O(1/\varepsilon^{\mathsf{d}})$ pairs that include p in W. ■

Lemma 3.2.11 *Let P be a set of n points in the plane, then one can solve the all nearest neighbor problem, in time $O(n(\log n + \log \Phi(P)))$ time, where Φ is the spread of P.*

Proof: The algorithm is described above. We only remain with the task of analyzing the running time. For a number $i \in \{0,-1,\ldots,-\lfloor \lg \Phi \rfloor - 4\}$, consider the set of pairs W_i, such that $\{u,v\} \in W_i$, if and only if $\{u,v\} \in \mathcal{W}$, and $2^{i-1} \leq \mathbf{d}(u,v) \leq 2^i$. Here, $\mathcal{W}$ is a $1/\varepsilon$-WSPD of P, where $\varepsilon = 1/4$. A point $\mathsf{p} \in \mathsf{P}$ can be scanned at most $O(1/\varepsilon^{\mathsf{d}}) = O(1)$ times because of pairs in W_i by Lemma 3.2.10. As such, a point gets scanned at most $O(\log \Phi)$ times overall, which implies the running time bound. ■

3.2.5.2 All nearest neighbor – the unbounded spread case

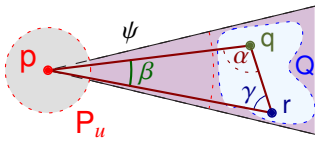
To handle the unbounded case, we need to use some additional geometric properties.

Lemma 3.2.12 *Let u be a node in the compressed $\mathcal{QT}$ of P, and partition the space around rep_u into cones of angle $\leq \pi/3$. Let ψ be such a cone, and let Q be the set of all points in P which are in distance $\geq 4\mathrm{diam}(\mathsf{P}_u)$ from rep_u, and they all lie inside ψ. Let q be the closest point in Q to rep_u. Then, q is the only point in Q that its nearest neighbor might be in P_u.*

Proof: Let $\mathsf{p} = \mathrm{rep}_u$ and consider any point $\mathsf{s} \in \mathsf{Q}$.

Since $\|\mathsf{s}-\mathsf{p}\| \geq \|\mathsf{q}-\mathsf{p}\|$, it follows that $\alpha = \angle\mathsf{sqp} \geq \angle\mathsf{qsp} = \gamma$. Now, $\alpha+\gamma = \pi-\beta$, where $\beta = \angle\mathsf{spq}$. But $\beta \leq \pi/3$, and as such

$$2\alpha \geq \alpha + \gamma = \pi - \beta \geq 3\beta - \beta = 2\beta.$$



Namely, α is the largest angle in the triangle $\triangle\mathsf{pqs}$, which implies $\|\mathsf{s}-\mathsf{p}\| \geq \|\mathsf{s}-\mathsf{q}\|$. Namely, q is closer to s than p, and as such p can not serve as the nearest neighbor to s in P.

It is now straightforward (but tedious) to show that, in fact, for any $\mathsf{t} \in \mathsf{P}_u$, we have $\|\mathsf{s}-\mathsf{t}\| \geq \|\mathsf{s}-\mathsf{q}\|$, which implies the claim.[②] ■

Lemma 3.2.12 implies that we can do a top-down traversal of $\mathcal{QT}(P)$, after computing a ε^{-1}-WSPD $\mathcal{W}$ of P, for $\varepsilon = 1/16$. For every node u, we maintain a (constant size) set R_u of candidate points such that P_u *might* contain their nearest neighbor.

So, assume we had computed $\mathsf{R}_{\overline{\mathsf{p}}(u)}$, and consider the set

$$X(u) = \mathsf{R}_{\overline{\mathsf{p}}(u)} \cup \bigcup_{\{u,v\}\in\mathcal{W},|\mathsf{P}_v|=1} \mathsf{P}_v.$$

(Note, that we do not have to consider pairs with $|\mathsf{P}_v| > 1$, since no point in P_v can have its nearest neighbor in P_u in such a scenario.) Clearly, we can compute $X(u)$ in linear time in the number of pairs in $\mathcal{W}$ involved with u. Now, we build a "grid" of cones around rep_u, and throw the points of $X(u)$ into this grid. For each such cone, we keep only the closest point p' to rep_u (because the other points in this cone would be use p' as nearest neighbor before using any point of P_u). Let R_u be the set of these closest points. Since the number of cones is $O(1)$, it follows that $|\mathsf{R}_u| = O(1)$.

We continue this top-down traversal till R_u is computed for all the nodes in the tree.

Now, for every vertex u, if P_u contains only a single point p, then we compute for any point $\mathsf{q} \in \mathsf{R}_u$ its distance to p, and if p is a better candidate to be a nearest neighbor, then we set p as the (current) nearest neighbor to q.

Correctness. Clearly, the resulting running time (ignoring the computation of the WSPD) is linear in the number of pairs of the WSPD and the size of the compressed quadtree. If p is the nearest neighbor to q, then there must be a WSPD pair $\{u, v\}$ such that $\mathsf{P}_v = \{\mathsf{q}\}$ and $\mathsf{p} \in \mathsf{P}_u$. But then, the algorithm would add q to the set R_u, and it would be in R_z, for all descendants z of u in the quadtree, such that $\mathsf{p} \in \mathsf{P}_z$. In particular, if y is the leaf of the quadtree storing p, then $\mathsf{q} \in \mathsf{R}_y$, which implies that the algorithm computes correctly the nearest neighbor to q.

This implies the correctness of the algorithm.

Theorem 3.2.13 *Given a set P of n points in $\mathbb{R}^{\mathsf{d}}$, one can solve the all nearest neighbor problem in $O(n \log n)$ time.*

[②] Here are the details for readers of little fate. By the law of sines, we have $\frac{\|\mathsf{p}-\mathsf{s}\|}{\sin\alpha} = \frac{\|\mathsf{q}-\mathsf{s}\|}{\sin\beta}$. As such, $\|\mathsf{q}-\mathsf{s}\| = \|\mathsf{p}-\mathsf{s}\|\frac{\sin\beta}{\sin\alpha}$. Now, if $\alpha \leq \pi - 3\beta$ then $\|\mathsf{q}-\mathsf{s}\| = \|\mathsf{p}-\mathsf{s}\|\frac{\sin\beta}{\sin\alpha} \leq \|\mathsf{p}-\mathsf{s}\|\frac{\sin\beta}{\sin(3\beta)} \leq \frac{\|\mathsf{p}-\mathsf{s}\|}{2} < \|\mathsf{p}-\mathsf{s}\| - \Delta(u)$, since $\|\mathsf{p}-\mathsf{s}\| \geq 4\Delta(u)$. This implies that no point of P_u can be the nearest neighbor of s.

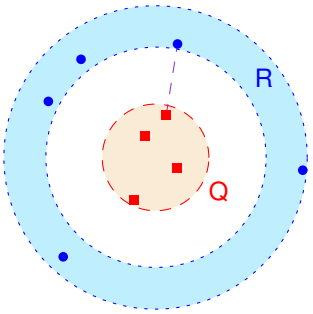
If $\alpha \geq \pi - 3\beta$ then the maximum length of qs is achieved when $\gamma = 2\beta$. The sines law then implies that $\|\mathsf{q}-\mathsf{s}\| = \|\mathsf{p}-\mathsf{q}\|\frac{\sin\beta}{\sin(2\beta)} \leq \frac{3}{4}\|\mathsf{p}-\mathsf{q}\| \leq \frac{3}{4}\|\mathsf{p}-\mathsf{s}\| < \|\mathsf{p}-\mathsf{s}\| - \Delta(u)$, which again implies the claim.

3.3 Semi-separated pairs decomposition

Here we present an interesting relaxation of WSPD, that has the advantage of having a low total weight.

Definition 3.3.1 Given a pair decomposition $\mathcal{W} = \left\{\{A_1, B_1\}, \ldots, \{A_s, B_s\}\right\}$ of a point-set P, its ***weight*** is $\omega(\mathcal{W}) = \sum_{i=1}^{s}(|A_i| + |B_i|)$.

It is easy to verify that, in the worst case, a WSPD of a set of n points in the plane might have weight $O(n^2)$. The notation of SSPD circumvent this by requiring a weaker notion of separation.



Definition 3.3.2 Two sets of points Q and R are $(1/\varepsilon)$-***semi-separated*** if

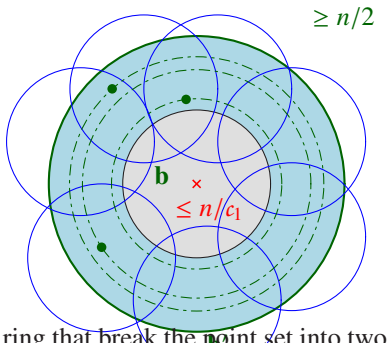
$$\min(\mathrm{diam}(\mathsf{Q}), \mathrm{diam}(\mathsf{R})) \leq \varepsilon \cdot \mathbf{d}(\mathsf{Q}, \mathsf{R}),$$

where $\mathbf{d}(\mathsf{Q}, \mathsf{R}, =) \min_{\mathsf{s}\in\mathsf{Q}, \mathsf{t}\in\mathsf{R}} \|\mathsf{s} - \mathsf{t}\|$.

See figure on the right for an example.

See Definition 3.1.2 for the original notion of sets being well-separated; in particular, in the well-separated case we demand that the separation was large relative to the diameter of both sets, while here it is sufficient that the separation is large for the smaller of the two sets. The analog of the WSPD (Definition 3.1.3) is the following.

Definition 3.3.3 (SSPD) For a point set P, a ***semi-separated pair decomposition*** (***SSPD***) of P with parameter $1/\varepsilon$, denoted by ε^{-1}-SSPD, is a pair decomposition of P formed by a set of pairs $\mathcal{W}$ such that all the pairs are $1/\varepsilon$-semi-separated.



Observation 3.3.4 *A ε^{-1}-WSPD of* P *is ε^{-1}-SSPD of* P.

3.3.1 Construction

We use the property that in low dimensions there always exists a good separating ring that break the point set into two reasonably large subsets. Indeed, consider the smallest ball $\mathbf{b} = \mathbf{b}(\mathsf{p}, r)$ that contains n/c_1 points of P, where c_1 is a sufficiently large constant. Let $\mathbf{b}'$ be the scaling of this ball by a factor of two. By a standard packing argument, the ring $\mathbf{b}' \setminus \mathbf{b}$ can be covered with $c = O(1)$ copies of $\mathbf{b}$, none of which can contain more than n/c_1 points of P. It follows, that by picking $c_1 = 3c$, we are guaranteed that at least half the points of P are outside $\mathbf{b}'$. Now, the ring can be split into $n/2$ empty rings (by taking a sphere that passes through each point inside the ring), and one of them would be of thickness at least r/n, and it would separate the inner n/c points of P from the outer $n/2$ of P. Doing this efficiently requires trading off some constants, and some tedious details, as described in the following lemma.

Lemma 3.3.5 *Let* P *be a set of n points in $\mathbb{R}^{\mathsf{d}}$, $t > 0$ be a parameter, and let c be a sufficiently large constant. Then one can compute, in linear time, a ball $\mathbf{b} = \mathbf{b}(\mathsf{p}, r)$, such that (i) $|\mathbf{b} \cap \mathsf{P}| \geq n/c$, (ii) $|\mathbf{b}(\mathsf{p}, r(1 + 1/t)) \cap \mathsf{P}| \leq n/2t + |\mathbf{b} \cap \mathsf{P}|$, and (iii) $|\mathsf{P} \setminus \mathbf{b}(\mathsf{p}, 2r)| \geq n/2$.*

We also need the following easy property.

Lemma 3.3.6 *Let* P *be a set of n points in $\mathbb{R}^{\mathsf{d}}$, with spread $\Phi = \Phi(\mathsf{P})$, and let $\varepsilon > 0$ be a parameter. Then, one can compute $(1/\varepsilon)$-WSPD (and thus a $(1/\varepsilon)$-SSPD) for* P *of total weight $O(n\varepsilon^{-2}\log\Phi)$. Furthermore, any point of* P *participates in at most $O\left(\varepsilon^{-d}\log\Phi\right)$ pairs.*

Proof: Build a regular (i.e., not compressed) quadtree for P. and observe that its depth is $O(\log\Phi)$. Now, construct a WSPD for P using this quadtree. Consider a pair of nodes (u, v) in this WSPD, and observe that the sidelength of u and v is the same up to a factor of two (since we used a non-compressed quadtree). As such, every node participates in $O(1/\varepsilon^{\mathsf{d}})$ pairs in the WSPD. We conclude that each point participates in $O\left(\varepsilon^{-d}\log\Phi\right)$ pairs, which implies that the total weight of this WSPD is as claimed. ■

Theorem 3.3.7 *Let* P *be a set of points in* $\mathbb{R}^{\mathsf{d}}$*, and let* $\varepsilon > 0$ *be a parameter. Then, one can compute a* $(1/\varepsilon)$*-SSPD for* P *of total weight* $O(n\varepsilon^{-d}\log^2 n)$*. The number of pairs in the SSPD is* $O(n\varepsilon^{-d}\log n)$*, and the computation time is* $O(n\log^2 n + n/\varepsilon^{-d}\log n)$*.*

3.3.2 Lower bound

The result of Theorem 3.3.7 can be improved so that the total weight of the SSPD is $O(n\varepsilon^{-d}\log n)$, see bibliographical notes. Interestingly, it turns out that any pair decomposition (without nay required separation property) has to be of total weight $\Omega(n\log n)$.

Lemma 3.3.8 *Let* P *be a set of* n *points, and let* $\mathcal{W} = \left\{\{A_1, B_1\}, \ldots, \{A_s, B_s\}\right\}$ *be a pair decomposition of* P *(see Definition 3.1.1). Then,* $\sum_i (|A_i| + |B_i|) = \Omega(n\log n)$*.*

3.4 Bibliographical Notes

Well separated pairs decomposition was defined by Callahan and Kosaraju [CK95]. They defined a different space decomposition tree, known as the ***fair split tree***. Here, one computes the axis parallel bounding box of the point-set, and always split along the longest edge by a perpendicular plane in the middle (or near the middle). This splits the point set into two sets, for which we construct fair split tree recursively. Implementing this in $O(n\log n)$ time requires some cleverness. See [CK95] for details.

Our presentation of WSPD (very roughly) follows [HM06]. The (easy) observation that a WSPD can be generated directly from a compressed quadtree (thus avoiding the fair split tree mess) is from there.

Callahan and Kosaraju [CK95] were inspired by the work of Vaidya [Vai86] on the all nearest neighbor problem (i.e., compute for each points in P, its nearest neighbor in P). He defined the fair split tree, and showed how to compute the all nearest neighbor in $O(n\log n)$ time. However, the first to give an $O(n\log n)$ time algorithm for the all nearest neighbor algorithm was Clarkson [Cla83] (this was part of his PhD thesis).

Diameter. The algorithm for computing the diameter in Section 3.2.3 can be improved by not constructing pairs that can not improve the (current) diameter, and constructing the underlying tree on the fly together with the diameter. This yields a simple algorithm that works quite will in practice, see [Har01a].

All nearest neighbor. Section 3.2.5 is a simplification of the solution for the all k-nearest neighbor problem. Here, one can compute for every point its k-nearest neighbor in $O(n\log n + nk)$ time. See [CK95] for details.

The all nearest neighbor algorithm for the bounded spread case (Section 3.2.5.1) is from [HM06]. Note, that unlike the unbounded case, this algorithm only use packing arguments for its correctness. Surprisingly, the usage of the Euclidean nature of the underlying space (as done in Section 3.2.5.2) seems to be crucial in getting a faster algorithm for this problem. In particular, for the case of metric spaces of low doubling dimension (that do have a small WSPD), solving this problem requires $\Omega(n^2)$ time in the worst case.

Dynamic maintenance. WSPD can be maintained in polylogarithmic time under insertions and deletions. This is quite surprising when one considers that, in the worst case, a point might participate in a linear number of pairs, and in fact, a node in the quadtree might participate in a linear number of pairs. This is described in detail in Callahan thesis [Cal95]. Interestingly, using randomization, maintaining the WSPD can be considerably simplified, see the work by Fischer and Har-Peled [FH05].

High dimension. In high dimensions, as the uniform metric demonstrates (i.e., n points, all of them in distance 1 from each other) the WSPD can have quadratic complexity. This metric is easily realizable as the vertices of a simplex in $\mathbb{R}^{n-1}$. On the other hand, doubling metrics have near linear size WSPD. Since WSPDs by themselves are so powerful, it is tempting to try and define the dimension of a point set by the size of the WSPD it posses. This seems like an interesting direction for future research, as currently little is known about it (to the best of my knowledge).

Semi-separated pairs decomposition. The notion of semi-separated pairs decomposition was introduced by Varadarajan [Var98] who used it to speed up matching algorithm for points in the plane. His SSPD construction of was of total weight $O(n \log^4 n)$ (for a constant ε). This was improved to $O(n \log n)$ by [AdBFG09].

SSPD were used to construct spanners that can survive even if large fraction of the graph disappears (for example, all the nodes inside arbitrary convex region disappear) [AdBFG09]. In [AdBF+09], SSPDs were used for computing additively weighted spanners. Further work on SSPD can be found in [ACFS09]. The simpler construction showed here is due to the author and it also works for metrics with low doubling dimension.

The elegant proof of the lower bound on the size of SSPD is from [BS07].

Euclidean minimum spanning tree. Let P be a set of points in $\mathbb{R}^d$, for which we want to compute its minimum spanning tree (MST). It is easy to verify that if an edge pq that is not a Delaunay edge of P, then its diametrical ball must contain some other point s in its interior, but then ps and qs are shorter than pq. This in turn implies that pq can not be an edge of the MST. As such, the edges of the MST are subset of the edges of the Delaunay triangulation of P. Since the Delaunay triangulation can be computed in $O(n \log n)$ time in the plane, this implies that MST can be computed in $O(n \log n)$ time in the plane. In $d \geq 3$ dimensions, the exact MST can be computed in $O\left(n^{2-2/(\lceil d/2 \rceil + 1) + \varepsilon'}\right)$ time, where $\varepsilon' > 0$ is an arbitrary small constant [AESW91]. The computation of MST is closely related to the bichromatic closest pair problem [KLN99].

There is a lot of work on MST in Euclidean settings, from estimating its weight in sub-linear time [CRT05], to doing this in the streaming model under insertions and deletions [FIS05], to implementation of a variant of the approximation algorithm described here [NZ01]. This list is by no means exhaustive.

Chapter 4

Random Partition via Shifting

> Associations of mutual interest between the university and the corporations were natural, inevitable, and widely accepted. According to the state legislature, they were to be actively pursued. The legislature, in fact, was already counting the "resources" that could be "allocated" elsewhere in state government when corporations began picking up more of the tab for higher education, so success in finding this money would certainly convince them that further experiments in driving the university into the arms of the private sector would be warranted, that actually paying for the university out of state funds was irresponsible, or even immoral, or even criminal (robbing widows and children, etc., to fatten sleek professors who couldn't find real employment, etc.).
>
> – Moo, Jane Smiley.

In this chapter, we investigate a rather simple technique for partitioning a geometric domain. We randomly shift a grid in $\mathbb{R}^{\mathsf{d}}$, and consider each grid cell resulting from this shift. The shifting is done by adding a random vector, chosen uniformly from $[0, 1]^{\mathsf{d}}$, so that the new grid has this vector as its origin.

Points that are close together have good probability to fall into the same cell in the new grid. This idea can be extended to shifting multi resolution grids over $\mathbb{R}^{\mathsf{d}}$. That is, we randomly shift a quadtree over a region of interest. This yields some simple algorithms for clustering and nearest-neighbor search.

4.1 Partition via Shifting

4.1.1 Shifted partition of the real line

Consider a real number $\Delta > 0$, and let b be a uniformly distributed number in the interval $[0, \Delta]$. This induces a natural partition of the real line into intervals, by the function

$$h_{\mathsf{b},\Delta}(x) = \left\lfloor \frac{x - \mathsf{b}}{\Delta} \right\rfloor.$$

Hence each interval has size Δ, and the origin is shifted to the right by an amount b.

Remark 4.1.1 Note, that $h_{\mathsf{b},\Delta}(x)$ induces the same partition of the real line as $h_{\mathsf{b}',\Delta}(x)$ (but it is not the same function) if $|\mathsf{b} - \mathsf{b}'| = i\Delta$, where i is an integer. In particular, this implies that we can pick b uniformly from any interval of length Δ and get the same distribution on the partitions of the real line.

Specifically, for our purposes, is enough if b is distributed uniformly in an interval of the from $[y + i\Delta, y + j\Delta]$, for two integers i and j such that $i < j$ and y is some real number. It is easy to verify that the we still get the same distribution on the partitions of the real line.

Lemma 4.1.2 *For any $x, y \in \mathbb{R}$, we have* $\mathbf{Pr}[h_{\mathsf{b},\Delta}(x) \neq h_{\mathsf{b},\Delta}(y)] = \min\left(\frac{|x-y|}{\Delta}, 1\right)$.

Proof: Assume $x < y$. Clearly, the claim trivially holds if $x - y > \Delta$, since then x and y are always assigned different values of $h_{\mathsf{b},\Delta}(\cdot)$, and the required probability is one. Now, imagine we pick b uniformly in the range $[x, x+\Delta]$. Clearly, the probability of the event we are interested in remains the same. But then, $h_{\mathsf{b},\Delta}(x) \neq h_{\mathsf{b},\Delta}(y)$ if and only if $\mathsf{b} \in [x, y]$, which implies the claim. ■

4.1.2 Shifted partition of space

Let P be a point set in $\mathbb{R}^{\mathsf{d}}$, and consider a point $\mathsf{b} = (\mathsf{b}_1, \ldots, \mathsf{b}_{\mathsf{d}}) \in \mathbb{R}^{\mathsf{d}}$ randomly and uniformly chosen from the hypercube $[0, \Delta]^{\mathsf{d}}$, and consider the grid $\mathsf{G}^{\mathsf{d}}(\mathsf{b}, \Delta)$ that has its origin at b, and with sidelength Δ. For a point $\mathsf{p} \in \mathbb{R}^{\mathsf{d}}$ the ID of the grid cell containing it is

$$\mathrm{id}(\mathsf{p}) = h_{\mathsf{b},\Delta}(\mathsf{p}) = (h_{\mathsf{b}_1,\Delta}(\mathsf{p}_1), \ldots, h_{\mathsf{b}_{\mathsf{d}},\Delta}(\mathsf{p}_{\mathsf{d}})).$$

(We used a similar concept when solving the closest pair problem, see Theorem $1.2.7_{\mathrm{p3}}$.)

Lemma 4.1.3 *Given a ball B of radius r in $\mathbb{R}^{\mathsf{d}}$ (or an axis parallel hypercube with sidelength $2r$). The probability that B is not contained in a single cell of $\mathsf{G}^{\mathsf{d}}(\mathsf{b}, \Delta)$ is bounded by* $\min(2dr/\Delta, 1)$*, where $\mathsf{G}^{\mathsf{d}}(\mathsf{b}, \Delta)$ is a randomly shifted grid.*

Proof: Project B into the ith coordinate. It becomes an interval B_i of length $2r$, and $\mathsf{G}^{\mathsf{d}}(\mathsf{b}, \Delta)$ becomes the one dimensional shifted grid $\mathsf{G}^1(b_i, \Delta)$. Clearly, B is contained in a single cell of $\mathsf{G}^{\mathsf{d}}(\mathsf{b}, \Delta)$ if and only if B_i is contained in a single cell of $\mathsf{G}^1(b_i, \Delta)$, for $i = 1, \ldots, d$.

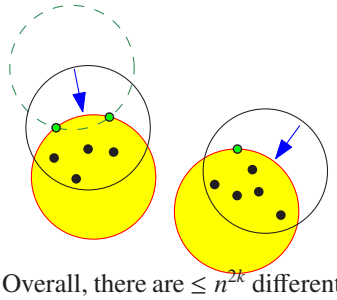
Now, B_i is not contained in an interval of $\mathsf{G}^1(\mathsf{b}_i, \Delta)$ iff its endpoints are in different cells. Let $\mathcal{E}_i$ denote this event. By Lemma 4.1.2, the probability of $\mathcal{E}_i$ is $\leq 2r/\Delta$. As such, the probability of B to not be contained in a single grid is $\mathbf{Pr}[\cup_i \mathcal{E}_i] \leq \sum_i \mathbf{Pr}[\mathcal{E}_i] \leq 2dr/\Delta$. Since a probability is always bounded by one, we have that this probability is bounded by $\min(2dr/\Delta, 1)$. ■

4.1.2.1 Application – covering by disks

Given a set P of n points in the plane we would like to cover them by a minimal number of unit disks. Here, a ***unit disk*** is a disk of radius one. Observe, that if the cover requires k disks, then we can compute it in $O\left(kn^{2k+1}\right)$ time.

Indeed, consider a unit disk D that covers a subset $\mathsf{Q} \subseteq \mathsf{P}$. We can translate D such that it still covers Q, and either its boundary circle contains two points of Q, or the top point of this circle is on an input point.

Observe that: (I) Every pair of such input points defines two possible disks, see figure on the right. (II) The same subset covered by a single disk might be coverable by several different such ***canonical*** disks. (III) If a pair of input points is at distance larger than 2, then the canonical disk they define is invalid.



As such, there are

$$2\binom{n}{2} + n \leq n^2$$

such canonical disks. We can assume the cover uses only such disks

Thus, we exhaustively check all such possible covers formed by k canonical disks. Overall, there are $\leq n^{2k}$ different covers to consider, and each such candidate cover can be verified in $O(nk)$ time. We thus get the following easy result.

Lemma 4.1.4 *Given a set P of n points in the plane one can compute, in $O\left(kn^{2k+1}\right)$ time, a cover of P by at most k unit disks, if such a cover exists.*

Proof: We use the above algorithm trying all covers of size i, for $i = 1, \ldots, k$. The algorithm returns the first cover found. Clearly, the running time is dominated by the last iteration of this algorithm. ■

One can improve the running time of Lemma 4.1.4 to $n^{O(\sqrt{k})}$.

The problem with this algorithm is that k might be quite large (say $n/4$). Fortunately, the shifting grid saves the day.

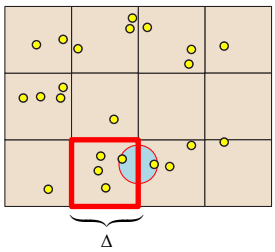
Theorem 4.1.5 *Given a set P of n points in the plane and a parameter $\varepsilon > 0$, one can compute using a randomized algorithm, in $n^{O(1/\varepsilon^2)}$ time, a cover of P by X unit disks, where $\mathbf{E}[X] \leq (1+\varepsilon)\mathrm{opt}$, where* opt *is the minimum number of disks required.*

Proof: Let $\Delta = 12/\varepsilon$, and consider a randomly shifted gird $\mathsf{G}^2(\mathsf{b}, \Delta)$. Compute all the grid cells that contain points of P. This is done by computing for each point $\mathsf{p} \in \mathsf{P}$ its id(p) and storing it in a hash table. Now, when inserting a point into the hash table, we add its grid cell to a separate list if this id is being inserted for the first time into the hash table. This clearly can be done in linear time.

Now, for each such grid cell $\square$ we now have the points of P falling into this grid cell (they are stored with the same id in the hash table), and let $\mathsf{P}_\square$ denote this set of points.

Observe, that any grid cell of $\mathsf{G}^2(\mathsf{b}, \Delta)$ can be covered by $M = (\Delta + 1)^2$ unit disks. Indeed, each unit disk contains a unit square, and clearly a grid cell of sidelength Δ can an be covered using at most M unit squares.

Thus, for each such grid cell $\square$, compute the minimum number of unit disks required to cover $\mathsf{P}_\square$. Since this number is at most M, we can compute this minimum cover in $O\left(Mn^{2M+1}\right)$ time, by Lemma 4.1.4. There are at most n non-empty grid cells, so the total running time of this stage is $O\left(Mn^{2M+2}\right) = n^{O(1/\varepsilon^2)}$. Finally, merge together the covers from each grid cell, and return this as the overall cover.



We are left with the task of bounding the expectation of X. So, consider the optimal solution $\mathcal{F} = \left\{D_1, \ldots, D_{\text{opt}}\right\}$. We generate a feasible solution $\mathcal{G}$ that is one of the possible solutions considered by our algorithm. Specifically, for each grid cell $\square$, let $\mathcal{F}_\square$ denote the set of disks of the optimal solution that intersect $\square$. Consider the multiset $\mathcal{G} = \bigcup_\square \mathcal{F}_\square$. Clearly, the algorithm returns for each grid cell $\square$ a cover that is of size at most $|\mathcal{F}_\square|$ (indeed, it returns the smallest possible cover for $\mathsf{P}_\square$, and $\mathcal{F}_\square$ is one such possible cover). As such, the cover returned by the algorithm is of size at most $|\mathcal{G}|$.

Clearly, a disk of the optimal solution can intersect at most 4 cells of the grid, and as such it can appear in $\mathcal{G}$ at most 4 times. In fact, a disk $D_i \in \mathcal{F}$ will appear in $\mathcal{G}$ more than once, if and only if it is not fully contained in a grid cell of $\mathsf{G}^2(\mathsf{b}, \Delta)$. By Lemma 4.1.3 the probability for that is bounded by $4/\Delta$ (as $r = 1$ and $d = 2$).

Specifically, let X_i be an indicator variable that is one if and only if D_i is not fully contained in a single cell of $\mathsf{G}^2(\mathsf{b}, \Delta)$. We have that

$$\begin{aligned}\mathbf{E}\left[|\mathcal{G}|\right] &\le \mathbf{E}\left[\text{opt} + \sum_{i=1}^{\text{opt}} 3X_i\right] = \text{opt} + \sum_{i=1}^{\text{opt}} 3\,\mathbf{E}[X_i] = \text{opt} + \sum_{i=1}^{\text{opt}} 3\,\mathbf{Pr}[X_i = 1] \le \text{opt} + \sum_{i=1}^{\text{opt}} 3\frac{4}{\Delta} \\ &= \left(1 + \frac{12}{\Delta}\right)\text{opt} = (1+\varepsilon)\,\text{opt},\end{aligned}$$

since $\Delta = 12/\varepsilon$. As such, in expectation, the solution returned by the algorithm is of size at most $(1+\varepsilon)$opt. ■

The running time of Theorem 4.1.5 can be improved to $n^{O(1/\varepsilon)}$.

4.1.3 Shifting quadtrees

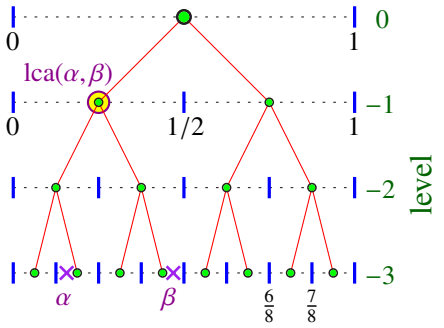
4.1.3.1 One dimensional quadtrees

Assume, that we are given a set P of n numbers contained in the interval $[1/2, 3/4]$. Randomly, and uniformly, choose a number $\mathsf{b} \in [0, 1/2]$, and consider the (one dimensional) quadtree $\mathcal{T}$ of P, using the interval $\mathsf{b} + [0, 1]$ for the root cell. Now, for two numbers $\alpha, \beta \in \mathsf{P}$ let

$$\mathbb{L}_{\mathsf{b}}(\alpha, \beta) = 1 - \text{bit}_\Delta(\alpha - \mathsf{b}, \beta - \mathsf{b}),$$

see Definition 2.2.5$_{\text{p9}}$.

This is the last ***level*** of the (one dimensional) shifted canonical grid (i.e., one dimensional quadtree) that contains α and β in the same interval. Namely, this is the level of the node of $\mathcal{T}$ that is the least common ancestor containing both numbers; that is, the level of $\mathrm{lca}(\alpha,\beta)$ is $\mathbb{L}_{\mathsf{b}}(\alpha,\beta)$. We remind the reader that a node in this quadtree that corresponds to an interval of length 2^{-i} has level $-i$. These definitions are demonstrated in the figure on the right (without shifting).



In the following, we will assume that one can compute $\mathbb{L}_{\mathsf{b}}(\alpha,\beta)$ in constant time.

Remark 4.1.6 Interestingly, the value of $\mathbb{L}_{\mathsf{b}}(\alpha,\beta)$ depends only on α, β and b, but is independent of the other points of P.

The following lemma bounds the probability that the least common ancestor of two numbers is in charge of an interval that is considerably longer than the difference between the two numbers.

Lemma 4.1.7 *Let $\alpha,\beta \in [1/2,3/4]$ be two numbers, and consider a random number $\mathsf{b} \in [0,1/2]$. Then, for any positive integer t, we have that $\mathbf{Pr}[\mathbb{L}_{\mathsf{b}}(\alpha,\beta) > \lg_2 |\alpha-\beta| + t] \le 4/2^t$.*

Proof: Let $M = \lfloor \lg|\alpha-\beta| \rfloor$ (we remind the reader that $\lg x = \log_2 x$). Consider the shifted partition of the real line by intervals of length $\Delta_{M+i} = 2^{M+i}$, and a shift b. Let X_{M+i} be an indicator variable that is one if and only if α and β are in different intervals of this shifted partition. Formally, $X_{M+i} = 1$ if and only if $h_{\mathsf{b},\Delta_{M+i}}(\alpha) \neq h_{\mathsf{b},\Delta_{M+i}}(\beta)$.

Now, if $\mathbb{L}_{\mathsf{b}}(\alpha,\beta) = M+i$ then the highest level such that both α and β lie in the same shifted interval is $M+i$. As such, going one level down, these two numbers are in different intervals, and $h_{\mathsf{b},2^{M+i-1}}(\alpha) \neq h_{\mathsf{b},2^{M+i-1}}(\beta)$. Namely, we have that $X_{M+i-1} = 1$. By Lemma 4.1.2, we have that $\mathbf{Pr}[X_{M+i} = 1] \le |\alpha-\beta|/\Delta_{M+i}$. As such, the probability we are interested in is

$$\mathbf{Pr}[\mathbb{L}_{\mathsf{b}}(\alpha,\beta) > \lg_2|\alpha-\beta| + t] \le \sum_{i=1+t}^{\infty} \mathbf{Pr}[\mathbb{L}_{\mathsf{b}}(\alpha,\beta) = M+i] \le \sum_{i=t}^{\infty} \mathbf{Pr}[X_{M+i}=1]$$
$$\le \sum_{i=t}^{\infty} \frac{|\alpha-\beta|}{\Delta_{M+i}} = \sum_{i=t}^{\infty} \frac{|\alpha-\beta|}{2^M\cdot 2^i} \le \sum_{i=t}^{\infty} \frac{|\alpha-\beta|}{(|\alpha-\beta|/2)\,2^i} \le \sum_{i=t}^{\infty} 2^{1-i} = 2^{2-t}.$$

■

Corollary 4.1.8 *Let $\alpha,\beta \in \mathsf{P} \subseteq [1/2,3/4]$ be two numbers, and consider a random number $\mathsf{b} \in [0,1/2]$. Then, for any parameters $c > 1$, we have $\mathbf{Pr}[\mathbb{L}_{\mathsf{b}}(\alpha,\beta) > \lg_2|\alpha-\beta| + c\lg n] \le 4/n^c$, where $n = |\mathsf{P}|$.*

4.1.3.2 Higher dimensional quadtrees

Let P be a set of n points in $[1/2,3/4]^{\mathsf{d}}$, and pick uniformly and randomly a point $\mathsf{b} \in [0,1/2]^{\mathsf{d}}$, and consider the shifted compressed quadtree $\mathcal{T}$ of P having $\mathsf{b}+[0,1]^{\mathsf{d}}$ as the root cell.

Consider any two points $\mathsf{p},\mathsf{q} \in \mathsf{P}$, their $\mathrm{lca}(\mathsf{p},\mathsf{q})$, and the one dimensional quadtrees $\mathcal{T}_1,\ldots,\mathcal{T}_{\mathsf{d}}$ built on each of the d coordinates of the point set. Since the quadtree $\mathcal{T}$ is the combination of these one dimensional quadtrees $\mathcal{T}_1,\ldots,\mathcal{T}_{\mathsf{d}}$, the level where p and q get separated in $\mathcal{T}$, is the first level in any of the quadtrees $\mathcal{T}_1,\ldots,\mathcal{T}_{\mathsf{d}}$ where p and q are separated. In particular, the ***level of the least common ancestor*** (i.e., the level of $\mathrm{lca}(\mathsf{p},\mathsf{q})$) is

$$\mathbb{L}_{\mathsf{b}}(\mathsf{p},\mathsf{q}) = \max_{i=1}^{\mathsf{d}} \mathbb{L}_{\mathsf{b}_i}(\mathsf{p}_i,\mathsf{q}_i)\,. \tag{4.1}$$

As in the one dimensional case (see Remark 4.1.6) the value of $\mathbb{L}_{\mathsf{b}}(\mathsf{p},\mathsf{q})$ is independent of the other points of P.

Intuitively, $\mathbb{L}_{\mathsf{b}}(\mathsf{p},\mathsf{q})$ is a well behaved random variable, as testified by the following lemma. Since we do not use this lemma anywhere directly we leave its proof as an exercise to the reader.

Lemma 4.1.9 *For any two fixed points $\mathsf{p},\mathsf{q} \in \mathsf{P}$, the following properties hold.*

(A) For any integer $t > 0$, we have that $\mathbf{Pr}\Big[\mathbb{L}_{\mathsf{b}}(\mathsf{p},\mathsf{q}) > \lg\|\mathsf{p}-\mathsf{q}\| + t\Big] \le 4\mathsf{d}/2^t$.

(B) $\mathbf{E}[\mathbb{L}_{\mathsf{b}}(\mathsf{p},\mathsf{q})] \le \lg\|\mathsf{p}-\mathsf{q}\| + \lg\mathsf{d} + 6$.

(C) $\mathbb{L}_{\mathsf{b}}(\mathsf{p},\mathsf{q}) \ge \lg\|\mathsf{p}-\mathsf{q}\| - \lg\mathsf{d} - 3$.

4.2 Hierarchical Representation of a Point Set

In the following, it will be convenient to carry out our discussion in a more general setting than low dimensional Euclidean space. In particular, we will use the notion of metric space.

Definition 4.2.1 A ***metric space*** is a pair $(\mathcal{X}, \mathbf{d})$ where $\mathcal{X}$ is a set and $\mathbf{d} : \mathcal{X} \times \mathcal{X} \to [0, \infty)$ is a ***metric***, satisfying the following axioms: (i) $\mathbf{d}_{\mathcal{M}}(x, y) = 0$ if and only if $x = y$, (ii) $\mathbf{d}_{\mathcal{M}}(x, y) = \mathbf{d}_{\mathcal{M}}(y, x)$, and (iii) $\mathbf{d}_{\mathcal{M}}(x, y) + \mathbf{d}_{\mathcal{M}}(y, z) \geq \mathbf{d}_{\mathcal{M}}(x, z)$ (triangle inequality).

Specifically, we are given a metric space $\mathcal{M}$ with a metric $\mathbf{d}$. We will slightly abuse notations by using $\mathcal{M}$ to refer to the underlying set of points.

4.2.1 Low Quality Approximation by HST

We will use the following special type of metric space.

Definition 4.2.2 Let P be a set of elements, and $\mathcal{H}$ be a tree having the elements of P as leaves. The tree $\mathcal{H}$ defines a ***hierarchically well-separated tree*** (HST) over the points of P, if for each vertex $u \in \mathcal{H}$ there is associated a label $\Delta(u) \geq 0$, such that $\Delta(u) = 0$ if and only if u is a leaf of $\mathcal{H}$. Furthermore, the labels are such that if a vertex u is a child of a vertex v then $\Delta(u) \leq \Delta(v)$. The distance between two leaves $x, y \in \mathcal{H}$ is defined as $\Delta(\mathrm{lca}(x, y))$, where $\mathrm{lca}(x, y)$ is the least common ancestor of x and y in $\mathcal{H}$.

If every internal node of $\mathcal{H}$ has exactly two children, we will refer to it as being a ***binary HST*** (BHST).

It is easy to verify that the distances defined by a HST comply with the triangle inequality, and as such it defines a metric. The usefulness of a HST is that the metric it defines has a very simple structure, and it can be easily manipulated algorithmically.

Example 4.2.3 Consider a point set $\mathsf{P} \subseteq \mathbb{R}^{\mathsf{d}}$, and a compressed quadtree $\mathcal{T}$ storing P, where for each node $v \in \mathcal{T}$, we set the diameter of $\square_v$ to be its label. It is easy to verify that this is a HST.

For convenience, from now on, we will work with BHSTs, since any HST can be converted into a binary HST in linear time while retaining the underlying distances. We will also associate with every vertex $u \in \mathcal{H}$, an arbitrary representative point $\mathrm{rep}_u \in \mathsf{P}_u$ (i.e., a point stored in the subtree rooted at u). We also require that $\mathrm{rep}_u \in \left\{ \mathrm{rep}_v \;\middle|\; v \text{ is a child of } u \right\}$.

Definition 4.2.4 A metric space $\mathcal{N}$ is said to ***t-approximate*** the metric $\mathcal{M}$, if they are defined over the same set of points P and $\mathbf{d}_{\mathcal{M}}(u, v) \leq \mathbf{d}_{\mathcal{N}}(u, v) \leq t \cdot \mathbf{d}_{\mathcal{M}}(u, v)$, for any $u, v \in \mathsf{P}$.

It is not hard to see that any n-point metric is $(n-1)$-approximated by some HST.

Lemma 4.2.5 *Given a weighted connected graph* G *on* n *vertices and* m *edges, it is possible to construct, in* $O(n \log n + m)$ *time, a binary* HST $\mathcal{H}$ *that* $(n-1)$*-approximates the shortest path metric of* G.

Proof: Compute the minimum spanning tree $\mathcal{T}$ of G in $O(n \log n + m)$ time[3].

Sort the $n-1$ edges of $\mathcal{T}$ in non-decreasing order, and add them to the graph one by one, starting with an empty graph on $V(\mathsf{G})$. The HST is built bottom up. At each stage, we have a collection of HSTs, each corresponding to a connected component of the current graph. Each added edge merges two connected components, and we merge the two corresponding HSTs into a single HST by adding a new common root v for the two HST, and labeling this root with the edge's weight times $|\mathsf{P}_v| - 1$, where P_v is the set of points stored in this subtree. This algorithm is only a slight variation on Kruskal algorithm, and hence has the same running time.

Let $\mathcal{H}$ denote the resulting HST. As for the approximation factor, let x, y be any two vertices of G, and e be the first edge added such that x and y are in the same connected component C, created by merging two connected components C_x and C_y. Observe that e must be the lightest edge in the cut between C_x and the rest of the minimum spanning tree

[3] Using, say, Prim's algorithm implemented using a Fibonacci heap.

$\mathcal{T}$, and as such, any path between x and y in G must contain an edge of weight at least $\omega(\mathsf{e})$. Now, e is the heaviest edge in C and

$$\mathbf{d}_{\mathsf{G}}(x,y) \leq (|C|-1)\,\omega(\mathsf{e}) \leq (n-1)\omega(\mathsf{e}) \leq (n-1)\mathbf{d}_{\mathsf{G}}(x,y),$$

since $\omega(\mathsf{e}) \leq \mathbf{d}_{\mathsf{G}}(x,y)$. Now, $\mathbf{d}_{\mathcal{H}}(x,y) = (|C|-1)\,\omega(\mathsf{e})$, and the claim follows. ■

Since any n-point metric $\mathsf{P} \subseteq \mathcal{M}$ can be represented using the complete graph over n vertices (with edge weights $\omega(xy) = \mathbf{d}_{\mathcal{M}}(x,y)$ for all $x, y \in \mathsf{P}$), we get the following.

Corollary 4.2.6 *For a set P of n points in a metric space $\mathcal{M}$, one can compute, in $O(n^2)$ time, a HST $\mathcal{H}$ that $(n-1)$-approximates the metric $\mathbf{d}_{\mathcal{M}}$.*

One can improve the running in low-dimensional Euclidean space (the approximation factor deteriorates slightly).

Corollary 4.2.7 *For a set P of n points in $\mathbb{R}^{\mathsf{d}}$, one can construct, in $O(n \log n)$ time (the constant in the $O(\cdot)$ depends exponentially on the dimension), a BHST $\mathcal{H}$ that $(2n-2)$-approximates the distances of points in P. That is, for any $\mathsf{p}, \mathsf{q} \in \mathsf{P}$, we have $\mathbf{d}_{\mathcal{H}}(\mathsf{p},\mathsf{q})/(2n-2) \leq \|\mathsf{p}-\mathsf{q}\| \leq \mathbf{d}_{\mathcal{H}}(\mathsf{p},\mathsf{q})$.*

Proof: We remind the reader, that in $\mathbb{R}^{\mathsf{d}}$, one can compute a 2-spanner for P of size $O(n)$, in $O(n \log n)$ time (see Theorem 3.2.2$_{\text{p21}}$). Let G be this spanner, and apply Lemma 4.2.5 to this spanner. Let $\mathcal{H}$ be the resulting HST metric. For any $\mathsf{p}, \mathsf{q} \in \mathsf{P}$, we have $\|\mathsf{p}-\mathsf{q}\| \leq \mathbf{d}_{\mathcal{H}}(\mathsf{p},\mathsf{q}) \leq (n-1)\mathbf{d}_{\mathsf{G}}(\mathsf{p},\mathsf{q}) \leq 2(n-1)\|\mathsf{p}-\mathsf{q}\|$. ■

Corollary 4.2.7 is unique to $\mathbb{R}^{\mathsf{d}}$ since for general metric spaces no HST can be computed in subquadratic time.

4.2.2 Fast and Dirty HST in High Dimensions

The above construction of HST has exponential dependency on the dimension. We next show how one can get an approximate HST of low quality but in polynomial time in the dimension.

Lemma 4.2.8 *Let P be a set of n points in $[1/2, 3/4]^{\mathsf{d}}$, and pick uniformly and randomly a point $\mathsf{b} \in [0, 1/2]^{\mathsf{d}}$, and consider the shifted compressed quadtree $\mathcal{T}$ of P having $\mathsf{b} + [0,1]^{\mathsf{d}}$ as the root cell. Then, for any constant $c > 1$, with probability $\geq 1 - 4\mathsf{d}/n^{c-2}$, we have that for all $i = 1, \ldots, \mathsf{d}$ and for all pairs of points of P, it holds*

$$\mathbb{L}_{\mathsf{b}_i}(\mathsf{p}_i, \mathsf{q}_i) \leq \lg|\mathsf{p}_i - \mathsf{q}_i| + c \lg n. \tag{4.2}$$

In particular, this property implies that the compressed quadtree $\mathcal{T}$ is a $2\sqrt{\mathsf{d}}n^c$-approximate HST for P.

Proof: Consider two points $\mathsf{p}, \mathsf{q} \in \mathsf{P}$, and a coordinate i. By Corollary 4.1.8, we have that

$$\mathbf{Pr}[\mathbb{L}_{\mathsf{b}_i}(\mathsf{p}_i, \mathsf{q}_i) > \lg|\mathsf{p}_i - \mathsf{q}_i| + c \lg n] \leq 4/n^c.$$

There are d possible coordinates, and $\binom{n}{2}$ possible pairs, and as such, by the union bound, this does not happen for any pair points of P, for any coordinate, with probability $\geq 1 - \mathsf{d}\binom{n}{2}\frac{4}{n^c} \geq 1 - 2\mathsf{d}/n^{c-2}$.

As such, with probability $\geq 1 - 2\mathsf{d}/n^c$, the level of the lca of any two points $\mathsf{p}, \mathsf{q} \in \mathsf{P}$ is at most

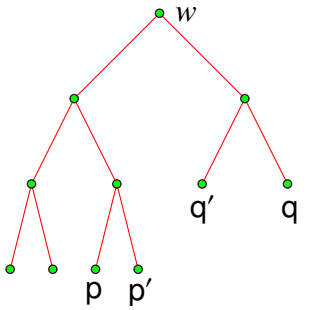
$$U = \mathbb{L}_{\mathsf{b}}(\mathsf{p},\mathsf{q}) = \max_{i=1}^{\mathsf{d}} \mathbb{L}_{\mathsf{b}_i}(\mathsf{p}_i,\mathsf{q}_i) \leq \max(\lg|\mathsf{p}_i - \mathsf{q}_i| + c\lg n) \leq \left\lceil \lg\|\mathsf{p}-\mathsf{q}\| + c\lg n \right\rceil,$$

by Eq. (4.1). The diameter of a cell at level $U(\mathsf{p},\mathsf{q})$ is at most $\sqrt{\mathsf{d}}2^U \leq 2\sqrt{\mathsf{d}}\,\|\mathsf{p}-\mathsf{q}\|\,n^c$. Namely, $\mathcal{T}$ is a $2\sqrt{\mathsf{d}}n^c$-approximate HST. ■

Verifying *quickly* that $\mathcal{T}$ is an acceptable HST (as far as the quality of approximation goes) is quite challenging in general. Fortunately for us, one can easily check that Eq. (4.2) holds.

Claim 4.2.9 *One can verify that Eq. (4.2) holds for the quadtree $\mathcal{T}$ computed by the algorithm of Lemma 4.2.8 in $O(\mathsf{d}n \log n)$ time.*

Proof: If Eq. (4.2) fails then it must be that there are two points $\mathsf{p}, \mathsf{q} \in \mathsf{P}$ and a coordinate j such that $\mathbb{L}_{\mathsf{b}_j}(\mathsf{p}_j, \mathsf{q}_j) > \lg|\mathsf{p}_j - \mathsf{q}_j| + t$, for $t = c \lg n$.



Now, observe that if there are two such bad points for the jth coordinate, then there are two bad points that are consecutive in the order along the jth coordinate. Indeed, consider $w = \mathrm{lca}(\mathsf{p}_j, \mathsf{q}_j)$ in the one dimensional compressed quadtree $\mathcal{T}_j$ of P on the jth coordinate. Let p' be the point (resp. q') with maximal (resp. minimal) value in the jth coordinate that is still in the left (resp. right) subtree of w, see figure on the right. Clearly, (i) $\mathrm{lca}(\mathsf{p}', \mathsf{q}') = w$, (ii) $\left|\mathsf{p}'_j - \mathsf{q}'_j\right| \leq \left|\mathsf{p}_j - \mathsf{q}_j\right|$, and (iii) p' and q' are consecutive in the ordering of P according to the values in the jth coordinate. As such, $\mathbb{L}_{\mathsf{b}_j}\left(\mathsf{p}'_j, \mathsf{q}'_j\right) = \mathbb{L}_{\mathsf{b}_j}\left(\mathsf{p}_j, \mathsf{q}_j\right) > \lg\left|\mathsf{p}_j - \mathsf{q}_j\right| + t \geq \lg\left|\mathsf{p}'_j - \mathsf{q}'_j\right| + t$. Namely, Eq. (4.2) fails for p' and q' on the jth coordinate.

As such, to verify Eq. (4.2) on the jth coordinate, we sort the points according to their order on the jth coordinate, and verify Eq. (4.2) for each consecutive pair. This takes $O(n \log n)$, time per coordinate, since computing $\mathbb{L}_{\mathsf{b}_j}(\cdot,\cdot)$ takes constant time. Doing this for all d coordinates takes $O(\mathsf{d}n \log n)$ time overall. ■

Theorem 4.2.10 *Given a set P of n points in $\mathbb{R}^{\mathsf{d}}$, for $\mathsf{d} \leq n$, one can compute a $2\sqrt{\mathsf{d}}n^5$-approximate HST of P in $O(\mathsf{d}n \log n)$ expected time.*

Proof: Set $c = 5$, use the algorithm of Lemma 4.2.8, and verify it, as described in Claim 4.2.9. If the compressed quadtree fails, we repeat the construction until succeeding. Computing and verifying the compressed quadtree takes $O(\mathsf{d}n \log n)$ time, by Theorem 2.3.9$_{\mathrm{p15}}$ and Claim 4.2.9. Now, since the probability of success is $\geq 1 - 4\mathsf{d}/n^{c-2} \geq 1 - 1/n$ (assuming $n \geq 3$), it follows that the algorithm would have to perform, in expectation, $1/(1 - 1/n) \leq 2$ iterations till it succeeds. ■

4.3 Low Quality ANN Search

We are interested in answering ***approximate nearest neighbor*** (***ANN***) queries in $\mathbb{R}^{\mathsf{d}}$. Namely, given a set of n points P in $\mathbb{R}^{\mathsf{d}}$, and a parameter $\varepsilon > 0$, we want to preprocess P, such that given a query point q, we can compute (quickly) a point $\mathsf{p} \in \mathsf{P}$, such that p is a $(1+\varepsilon)$-approximate nearest-neighbor to q in P. Formally, $\|\mathsf{q} - \mathsf{p}\| \leq (1+\varepsilon)\mathbf{d}(\mathsf{q}, \mathsf{P})$, where $\mathbf{d}(\mathsf{q}, \mathsf{P}) = \min_{\mathsf{p} \in \mathsf{P}} \|\mathsf{q} - \mathsf{p}\|$.

4.3.1 The data-structure and search procedure.

Let P be a set of n points in $\mathbb{R}^{\mathsf{d}}$, contained inside the cube $[1/2, 3/4]^{\mathsf{d}}$. Let b be a random vector in the cube $[0, 1/2]^{\mathsf{d}}$, and consider the compressed shifted quadtree $\mathcal{T}$ having the hypercube $\mathsf{b} + [0,1]^{\mathsf{d}}$ as its root. We choose for each node v of the quadtree a representative point $\mathrm{rep}_v \in \mathsf{P}_v$.

Given a query point $\mathsf{q} \in [1/2, 3/4]^{\mathsf{d}}$, let v be the lowest node of $\mathcal{T}$ whose region rg_v contains q.

(A) If rep_v is defined then we return it as the ANN.

(B) If rep_v is not defined, then it must be that v is an empty leaf, so let u be its parent, and return rep_u as q's ANN.

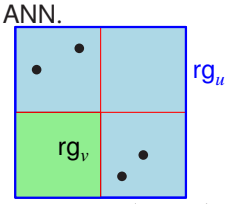
4.3.2 Analysis

Let us consider the above search procedure. When answering a query, there are several possibilities:

(A) If rg_v is a cube (i.e., v is a leaf) and v stores a point $\mathsf{p} \in \mathsf{Q}$ inside it, then we return p as the ANN.

(B) If v is a leaf but there is no point associated with it.

In this case rep_v is not defined, and we return rep_u, where $u = \overline{\mathrm{p}}(v)$. Observe, that $\left\|\mathsf{q} - \mathrm{rep}_u\right\| \leq 2\mathrm{diam}(\mathsf{rg}_v)$. This case happens if the parent u of v has two children that contains points of P, and v is one of the empty children. This situation is depicted in the figure on the right.



(C) If rg_v is an annulus, then v is a compressed node. In this case, we return rep_v as the ANN. Observe that $\mathbf{d}(\mathsf{q}, \mathrm{rep}_v) \leq \mathrm{diam}(\mathsf{rg}_v)$.

In all these cases, the distance to the ANN found is at most $2\mathrm{diam}(\square_v)$.

Lemma 4.3.1 *For any $\tau > 1$, and a query point q, the above data-structure returns a τ-approximation to the distance to the nearest neighbor of q in P, with probability $\geq 1 - 4\mathsf{d}^{3/2}/\tau$.*

Proof: Let p be q's nearest neighbor in P (i.e., $= \|\mathsf{q} - \mathsf{p}\| = \mathbf{d}(\mathsf{q}, \mathsf{P})$). Let $\mathbf{b}$ be the ball with diameter $\|\mathsf{q} - \mathsf{p}\|$ that contains q and p (i.e., this is the diametrical ball of qp). Consider the lowest node u in the compressed quadtree that contains $\mathbf{b}$ completely. By construction, the node v fetched by the point-location query for q must be either u or one of its descendants. As such, the ANN returned is of distance $\leq 2\mathrm{diam}(\square_v) \leq 2\mathrm{diam}(\square_u)$.

Let $\ell = \|\mathsf{q} - \mathsf{p}\|$. By Lemma 4.1.3, the probability that $\mathbf{b}$ is fully contained inside a single cell in the ith level of $\mathcal{T}$ is at least $1 - \mathsf{d}\ell/2^i$ (i is a non-positive integer). In such a case, let $\square$ be this grid cell, and observe that the distance to the ANN returned is at most $2\mathrm{diam}(\square) \leq 2\sqrt{\mathsf{d}}2^i$. As such, the quality of the ANN returned in such a case is bounded by $2\sqrt{\mathsf{d}}2^i/\ell$. If we want this ANN to be of quality τ, then we require that

$$2\sqrt{\mathsf{d}}2^i/\ell \leq \tau \quad \Longrightarrow \quad 2^i \leq \frac{\ell\tau}{2\sqrt{\mathsf{d}}} \quad \Longrightarrow \quad i \leq \lg \frac{\ell\tau}{2\sqrt{\mathsf{d}}}.$$

In particular, setting $i = \left\lfloor \lg\left(\ell\tau/2\sqrt{\mathsf{d}}\right) \right\rfloor$, we get that the returned point is a τ-ANN with probability at least

$$1 - \frac{\mathsf{d}\ell}{2^i} \geq 1 - \frac{4\mathsf{d}^{3/2}\ell}{\ell\tau} = 1 - \frac{4\mathsf{d}^{3/2}}{\tau},$$

implying the claim. ■

One thing we glossed over in the above is how to handle queries outside the square $[1/2, 3/4]^{\mathsf{d}}$. This can be handled by scaling the input point set P to lie inside a hypercube of diameter (say) $1/n^2$ centered at the middle of this domain $[1/2, 3/4]^{\mathsf{d}}$. Let T be the affine transformation (i.e., it is scaling and translation) that realizing this. Clearly, the ANN to q in P is the point corresponding to the ANN to $T(\mathsf{q})$ in $T(\mathsf{P})$. In particular, given a query point q we answer the ANN query on the transformed point set $T(\mathsf{P})$ using the query point $T(\mathsf{q})$.

Now, if the query point $T(\mathsf{q})$ is outside $[1/2, 3/4]^{\mathsf{d}}$ then any point of $T(\mathsf{P})$ is $(1 + 1/n)$-ANN to the query point as can be easily verified.

Combining Lemma 4.3.1 with Theorem 2.3.9, we get the following result.

Theorem 4.3.2 *For a point set $\mathsf{P} \subseteq [1/2, 3/4]^{\mathsf{d}}$, a randomly shifted compressed quadtree $\mathcal{T}$ of P can answer ANN queries in $O(\mathsf{d}\log n)$ time. The time to build this data-structure is $O(\mathsf{d}n\log n)$. Furthermore, for any $\tau > 1$, the returned point is a τ-ANN with probability $\geq 1 - 4\mathsf{d}^{3/2}/\tau$.*

Remark 4.3.3 (A) Theorem 4.3.2 is usually interesting for τ being polynomially large in n, where the probability of success is quite high.

(B) Note, that even for a large τ, this data-structure does not necessarily return the guaranteed quality of approximation for all the points in space. One can prove that this data-structure works with high probability to all the points in space (but the guarantee of approximation deteriorates, naturally).

4.4 Bibliographical notes

The approximation algorithm for covering by unit disks is due to Hochbaum and Mass [HM85], and our presentation is a variant of their algorithm. The idea of HSTs was used by Bartal [Bar98] to get a similar result for more general settings.

The idea of shifted quadtrees can be derandomized [Cha98] and still yield interesting results. This is done by picking the shift carefully and inspecting the resulting Ω-order. The idea of doing point location queries in a compressed quadtree to answer ANN is also from [Cha98] but it is probably found in much earlier work.

Chapter 5

Approximate Nearest Neighbor Search in Low Dimension

"Napoleon has not been conquered by man. He was greater than all of us. But god punished him because he relied on his own intelligence alone, until that prodigious instrument was strained to breaking point. Everything breaks in the end."
– Carl XIV Johan, King of Sweden.

5.1 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, computing the Voronoi diagram of P, and preprocessing it for point-location queries. This requires (roughly) $O(n^{\lceil d/2 \rceil})$ time.)

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

Definition 5.1.1 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 an $(1+\varepsilon)$-***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.)

5.2 The bounded spread case

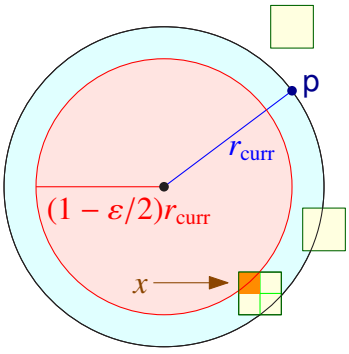
Let P be a set of n points contained inside the unit hypercube in $\mathbb{R}^{\mathsf{d}}$, and let $\mathcal{T}$ be a quadtree of P, where $\mathrm{diam}(\mathsf{P}) = \Omega(1)$. We assume that with each (internal) node u of $\mathcal{T}$, there is an associated representative point, rep_u, such that 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 algorithm. The algorithm would maintain a set of nodes of $\mathcal{T}$ that might contain the ANN to the query point. Each such node as 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 refined by replaced 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 that 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 on the right. Out the three nodes currently in the candidate set, only of child 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 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 $\mathsf{s} \in \mathsf{rg}_w \cap \mathsf{P}$ such that $\|\mathsf{q}-\mathsf{s}\| < (1-\varepsilon/2)r_{\text{curr}}$. A conservative lower bound on the distance of any point in rg_w to q is $\left\|\mathsf{q}-\mathrm{rep}_w\right\| - \mathrm{diam}(\square_w)$. In particular, if $\left\|\mathsf{q}-\mathrm{rep}_w\right\| - \mathrm{diam}(\square_w) > (1-\varepsilon/2)r_{\text{curr}}$ then we can abort the search for ANN in the subtree of w.

The algorithm. Let $A_0 = \{\mathrm{root}(T)\}$, and let $r_{\text{curr}} = \left\|\mathsf{q}-\mathrm{rep}_{\mathrm{root}(T)}\right\|$. 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 $\square_v$ denote the cell (i.e., region) v corresponds to. For every node $w \in C_v$, we compute

$$r_{\text{curr}} \leftarrow \min\Big(r_{\text{curr}}\,,\, \left\|\mathsf{q}-\mathrm{rep}_w\right\|\Big).$$

The algorithm checks if

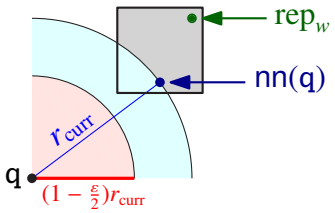
$$\left\|\mathsf{q}-\mathrm{rep}_w\right\| - \mathrm{diam}(\square_w) < (1-\varepsilon/2)r_{\text{curr}}, \tag{5.1}$$

and if so, it adds w to A_i. The algorithm continues in this expansion process till all the elements of A_{i-1} were 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, all these nodes belong to the canonical grid $\mathsf{G}_{2^{-i}}$ 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.)

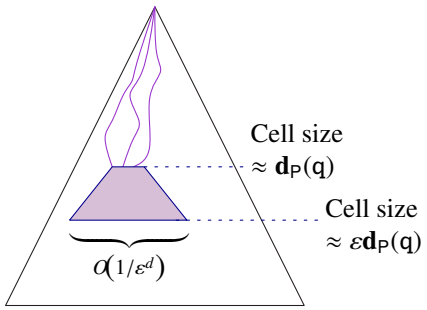
Consider the last node w inspected by the algorithm such that $\mathsf{nn}(\mathsf{q}) \in \mathsf{P}_w$. Since the algorithm decided to throw this node away, we have, by the triangle inequality, that



$$\begin{aligned}\|\mathsf{q}-\mathsf{nn}(\mathsf{q})\| &\geq \left\|\mathsf{q}-\mathrm{rep}_w\right\| - \left\|\mathrm{rep}_w-\mathsf{nn}(\mathsf{q})\right\| \\ &\geq \left\|\mathsf{q}-\mathrm{rep}_w\right\| - \mathrm{diam}(\square_w) \geq (1-\varepsilon/2)r_{\text{curr}}.\end{aligned}$$

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

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 $r = \mathbf{d}(\mathsf{q}, \mathsf{P})$, 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 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., $O(1/\varepsilon^{\mathsf{d}})$) to bound the query running time for this part of the query.



Lemma 5.2.1 *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* $\operatorname{diam}(\mathsf{P}) = \Omega(1)$*. Let* q *be a query point, and let* $\varepsilon > 0$ *be a parameter. An* $(1+\varepsilon)$*-ANN to* q *can be computed in* $O\left(\varepsilon^{-d} + \log(1/\varpi)\right)$ *time, where* $\varpi = \mathbf{d}(\mathsf{q}, \mathsf{P})$*.*

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 $\operatorname{diam}(\square_w) < (\varepsilon/4)\varpi$ then

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

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 Eq. (5.1). 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 $\mathsf{nn}(\mathsf{q}, \mathsf{P})$. Clearly, the distance between q and rep_u is at most $\ell_i = \varpi + \operatorname{diam}_u = \varpi + \sqrt{\mathsf{d}}2^{-i}$. As such, in the end of the ith iteration, we have $r_{\text{curr}} \leq \ell_i$, since the algorithm had inspected u.

Thus, the only cells of $\mathsf{G}_{2^{-i-1}}$ 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, $r_{\text{curr}} \leq \ell_i$. As such, any node of $\mathsf{G}_{2^{-i-1}}$ (i.e., nodes considered in the $(i+1)$th iteration of the algorithm) that is in distance larger than r_{curr} from q can not 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 $\mathsf{G}_{2^{-i-1}}$ that intersect a box of sidelength $2\ell_i$ centered at q; that is

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

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 5.2.1 to the case that 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 5.2.2 *Let* P *be a set of* n *points in* $\mathbb{R}^{\mathsf{d}}$*, and let* $\mathcal{T}$ *be a quadtree of* P*, where* $\operatorname{diam}(\mathsf{P}) = \Omega(1)$*. Given a query point* q *and* $1 \geq \varepsilon > 0$*, one can return an* $(1+\varepsilon)$*-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 5.2.3 *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 $O\Big(1/\varepsilon^{\mathsf{d}} + \log \Phi(\mathsf{P})\Big)$ time. In fact, the query time is $O(1/\varepsilon^{\mathsf{d}} + \log(\mathrm{diam}(\mathsf{P})/\varpi))$, where $\varpi = \mathbf{d}(\mathsf{q}, \mathsf{P})$.*

5.3 ANN – the unbounded general case

The Snark and the unbounded spread case. (Or a meta-philosophical pretentious discussion that the reader might want to skip. The reader might consider this to be a footnote of a footnote, which finds itself inside the text because of lack of space in 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 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 always 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. (At times the resulting solution is so complicated, it feels like a Boojum [Car76].)

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 insights on the problem, and in some cases leads to more efficient solutions than the bounded spread case.

With this caveat emptor[2] warning duly given, we plunge ahead into solving the ANN for the unbounded spread case.

Plan of attack. To answer 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 5.2.3 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 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.

Lemma 5.3.1 *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 $\mathsf{P} \cap \widehat{\square} = \mathsf{P}_w$.*

5.3.1 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 5.3.1. We will also need a data-structure that supports very rough ANN queries

[2] Buyer beware in Latin.

quickly. We describe one way to build such a data-structure in the next section. We will use the result we derived by using a shifted quadtree and a simple point-location query, see Theorem 4.3.2$_{\text{p37}}$.

> 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 $\mathbf{d}(\mathsf{q},\mathsf{P}) \leq \|u-q\| \leq (1+4n)\mathbf{d}(\mathsf{q},\mathsf{P})$. Let $R = \|u-q\|$ and $r = \|u-q\|/(4n+1)$. 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 in distance $\leq R$ from q; that is, C is the set of grid cells of $\mathsf{G}_{2^{\mathbb{L}}}$ that (completely) covers ball(q, R). Clearly, $\mathsf{nn}(\mathsf{q},\mathsf{P}) \in \text{ball}(\mathsf{q},R) \subseteq \bigcup_{\square\in C} \square$. 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 5.3.1). (Note, that if $\square$ does not contain any point of P this query would return a leaf or a compressed node that its 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 5.2.3 to the compressed quadtree rooted at v. We return the nearest neighbor found.

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

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

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 5.2.3 to w, it would return us a $(1+\varepsilon)$-ANN to q.

Theorem 5.3.2 *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 $q \in \mathbb{R}^{\mathsf{d}}$, 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.*

5.4 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. This and several additional ideas are used in the construction of the cover tree of Indyk and Motwani [IM98].

Section 5.2 is a simplification of Arya *et al.* [AMN+98] work. Section 5.3 is also inspired to a certain extent by Arya *et al.* work, 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 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 see 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 5.3.2, 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.* rebalances 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 is seems to be inferior, in practice, to Arya *et al.* work, for the reason that while the regions 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 presented also some other results.

Details on Faster Query Time. A set of points Q is $\sqrt{\varepsilon}$-*far* from a query point q, if the $\left\|p - c_Q\right\| \geq \operatorname{diam}(Q) / \sqrt{\varepsilon}$, 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 $O(\sqrt{\varepsilon})$ (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. 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 5.3.2, 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 $\operatorname{diam}(\square_v) = O\Big(\sqrt{\varepsilon}\mathbf{d}(\mathsf{q}, \mathsf{P})\Big)$. As such, the number of nodes visited would be $O(\log n + 1/\varepsilon^{d/2})$, and 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 $O(n/\varepsilon^{\mathsf{d}})$. Those 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, Mount and Malamatos [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 tradeoff that all other results suffer from (i.e., the query time multiplied by the construction time has dependency of $1/\varepsilon^{\mathsf{d}}$ 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, I 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 be probably be made subexponential with careful implementation). Getting a subexponential algorithm requires a completely different technique.

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) $O(\log\log U + 1/\varepsilon^{\mathsf{d}})$ 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 fast point-location query in it. This only requires using the floor function and hashing (note, that the algorithm of Theorem 5.3.2 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 blowup 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 $d + 1$ quadtrees, one can argue that the approximate nearest neighbor must lie in the same cell (and 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 $d + 1$ 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 $1/\varepsilon^{d}$ 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), and (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 XOR 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 5.3.2 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.

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 approximate nearest neighbor search in high dimensions is by Indyk [Ind04]. In low dimensions, there is a survey by Arya and Mount [AM04].

Chapter 6

Tail Inequalities

"Wir müssen wissen, wir werden wissen" (We must know, we shall know)
– David Hilbert.

6.1 Tail Inequalities

6.1.1 The Chernoff Bound — Special Case

Theorem 6.1.1 *Let $X_1, \ldots, X_n$ be n independent random variables, such that $\mathbf{Pr}[X_i = 1] = \mathbf{Pr}[X_i = -1] = \frac{1}{2}$, for $i = 1, \ldots, n$. Let $Y = \sum_{i=1}^{n} X_i$. Then, for any $\Delta > 0$, we have*

$$\mathbf{Pr}[Y \geq \Delta] \leq e^{-\Delta^2/2n}.$$

By the symmetry of Y, we get the following:

Corollary 6.1.2 *Let $X_1, \ldots, X_n$ be n independent random variables, such that $\mathbf{Pr}[X_i = 1] = 1/2$ and $\mathbf{Pr}[X_i = -1] = 1/2$, for $i = 1, \ldots, n$. Let $Y = \sum_{i=1}^{n} X_i$. Then, for any $\Delta > 0$, we have $\mathbf{Pr}\Big[|Y| \geq \Delta\Big] \leq 2\exp\Big(-\Delta^2/2n\Big)$.*

Corollary 6.1.3 *Let $X_1, \ldots, X_n$ be n independent* coin flips*, such that $\mathbf{Pr}[X_i = 0] = \mathbf{Pr}[X_i = 1] = 1/2$, for $i = 1, \ldots, n$. Let $Y = \sum_{i=1}^{n} X_i$. Then, for any $\Delta > 0$, we have $\mathbf{Pr}[|Y - n/2| \geq \Delta] \leq 2\exp\Big(-2\Delta^2/n\Big)$.*

6.1.2 The Chernoff Bound — General Case

Theorem 6.1.4 *For any $\delta > 0$, we have $\mathbf{Pr}[X \geq (1+\delta)\mu] \leq \left(\dfrac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu}$.*

Or in a more simplified form, for any $\delta \leq 2e - 1$,

$$\mathbf{Pr}[X \geq (1+\delta)\mu] \leq \exp\Big(-\mu\delta^2/4\Big), \tag{6.1}$$

and

$$\mathbf{Pr}[X \geq (1+\delta)\mu] \leq 2^{-\mu(1+\delta)}, \tag{6.2}$$

for $\delta \geq 2e - 1$.

Values	Probabilities	Inequality	Ref
$-1, +1$	$\mathbf{Pr}[X_i = -1] =$ $\mathbf{Pr}[X_i = 1] = \frac{1}{2}$	$\mathbf{Pr}[Y \geq \Delta] \leq e^{-\Delta^2/2n}$ $\mathbf{Pr}[Y \leq -\Delta] \leq e^{-\Delta^2/2n}$ $\mathbf{Pr}[\lvert Y\rvert \geq \Delta] \leq 2e^{-\Delta^2/2n}$	Theorem 6.1.1 Theorem 6.1.1 Corollary 6.1.2
$0, 1$	$\mathbf{Pr}[X_i = 0] =$ $\mathbf{Pr}[X_i = 1] = \frac{1}{2}$	$\mathbf{Pr}\left[\left\lvert Y - \frac{n}{2}\right\rvert \geq \Delta\right] \leq 2e^{-2\Delta^2/n}$	Corollary 6.1.3
0,1	$\mathbf{Pr}[X_i = 0] = 1 - p_i$ $\mathbf{Pr}[X_i = 1] = p_i$	$\mathbf{Pr}[Y > (1+\delta)\mu] < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu$	Theorem 6.1.4
	For $\delta \leq 2e - 1$ $\delta \geq 2e - 1$ For $\delta \geq 0$	$\mathbf{Pr}[Y > (1+\delta)\mu] < \exp\left(-\mu\delta^2/4\right)$ $\mathbf{Pr}[Y > (1+\delta)\mu] < 2^{-\mu(1+\delta)}$ $\mathbf{Pr}[Y < (1-\delta)\mu] < \exp\left(-\mu\delta^2/2\right)$	Theorem 6.1.4 Theorem 6.1.5
$X_i \in \left[a_i, b_i\right]$	The X_is have arbitrary independent distributions	$\mathbf{Pr}\left[\left\lvert Y - \mu\right\rvert \geq \eta\right] \leq$ $2\exp\left(-\frac{2\eta^2}{\sum_{i=1}^n (b_i - a_i)^2}\right)$	Theorem 6.1.6

Table 6.1: Summary of Chernoff type inequalities covered. Here we have n variables $X_1, \ldots, X_n$, $Y = \sum_i X_i$ and $\mu = \mathbf{E}[Y]$.

6.1.2.1 The Chernoff Bound — General Case — The Other Direction

Theorem 6.1.5 *Let $X_1, \ldots, X_n$ be independent Bernoulli trials, where $\mathbf{Pr}[X_i = 1] = p_i$, and $\mathbf{Pr}[X_i = 0] = q_i = 1 - p_i$, for $i = 1, \ldots, n$. Furthermore, let $X = \sum_{i=1}^b X_i$ and $\mu = \mathbf{E}[X] = \sum_i p_i$. We have the following*

$$\mathbf{Pr}[X < (1-\delta)\mu] < e^{-\mu\delta^2/2}.$$

Theorem 6.1.6 (Hoeffding's inequality.) *Let $X_1, \ldots, X_n$ be independent random variables, where $X_i \in \left[a_i, b_i\right]$, for $i = 1, \ldots, n$. Then, for the random variable $S = X_1 + \cdots + X_n$, and any $\eta > 0$, we have*

$$\mathbf{Pr}\left[\left\lvert S - \mathbf{E}[S]\right\rvert \geq \eta\right] \leq 2\exp\left(-\frac{2\eta^2}{\sum_{i=1}^n (b_i - a_i)^2}\right)$$

6.2 Bibliographical notes

The exposition here follows more or less the exposition in [MR95]. The special symmetric case (Theorem 6.1.1) is taken from [Cha01], although the proof is only very slightly simpler than the generalized form, it does yield a slightly better constant, and it would be useful when discussing discrepancy.

An orderly treatment of probability is outside the scope of our discussion. The standard text on the topic is the book by Feller [Fel91]. A more accessible text might be any introductory undergrad text on probability, in particular [MN98] has a nice chapter on the topic.

The proof of Hoeffding's inequality follows (roughly) the proof provided in Carlos Rodriguez's class notes (which are available online). There are generalization of Hoeffding's inequality to considerably more general setups, see Talagrand's inequality [AS00].

Bibliography

[ACFS09] M. A. Abam, P. Carmi, M. Farshi, and M. H. M. Smid. On the power of the semi-separated pair decomposition. In *Proc. 11th Workshop Algorithms Data Struct.*, pages 1–12, 2009.

[AdBF+09] M. A. Abam, M. de Berg, M. Farshi, J. Gudmundsson, and M. H. M. Smid. Geometric spanners for weighted point sets. In *Proc. 17th Annu. European Sympos. Algorithms*, pages 190–202, 2009.

[AdBFG09] M. A. Abam, M. de Berg, M. Farshi, and J. Gudmundsson. Region-fault tolerant geometric spanners. *Discrete Comput. Geom.*, 41(4):556–582, 2009.

[AEIS99] A. Amir, A. Efrat, P. Indyk, and H. Samet. Efficient algorithms and regular data structures for dilation, location and proximity problems. In *Proc. 40th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 160–170, 1999.

[AESW91] P. K. Agarwal, H. Edelsbrunner, O. Schwarzkopf, and E. Welzl. Euclidean minimum spanning trees and bichromatic closest pairs. *Discrete Comput. Geom.*, 6(5):407–422, 1991.

[AM98] S. Arya and D. Mount. ANN: library for approximate nearest neighbor searching. `http://www.cs.umd.edu/~mount/ANN/`, 1998.

[AM02] S. Arya and T. Malamatos. Linear-size approximate Voronoi diagrams. In *Proc. 13th ACM-SIAM Sympos. Discrete Algorithms*, pages 147–155, 2002.

[AM04] S. Arya and D. M. Mount. Computational geometry: Proximity and location. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, chapter 63. CRC Press LLC, Boca Raton, FL, 2004. to appear.

[AMM02] S. Arya, T. Malamatos, and D. M. Mount. Space-efficient approximate Voronoi diagrams. In *Proc. 34th Annu. ACM Sympos. Theory Comput.*, pages 721–730, 2002.

[AMN+98] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. Assoc. Comput. Mach.*, 45(6), 1998.

[AS00] N. Alon and J. H. Spencer. *The Probabilistic Method*. Wiley Inter-Science, 2nd edition, 2000.

[Bar98] Y. Bartal. On approximating arbitrary metrices by tree metrics. In *Proc. 30th Annu. ACM Sympos. Theory Comput.*, pages 161–168, 1998.

[BEG94] M. Bern, D. Eppstein, and J. Gilbert. Provably good mesh generation. *J. Comput. Syst. Sci.*, 48:384–409, 1994.

[BS07] B. Bollobás and A. Scott. On separating systems. *Eur. J. Comb.*, 28(4):1068–1071, 2007.

[Cal95] P. B. Callahan. *Dealing with higher dimensions: the well-separated pair decomposition and its applications*. Ph.D. thesis, Dept. Comput. Sci., Johns Hopkins University, Baltimore, Maryland, 1995.

[Car76] L. Carroll. The hunting of the snark, 1876.

[Cha98] T. M. Chan. Approximate nearest neighbor queries revisited. *Discrete Comput. Geom.*, 20:359–373, 1998.

[Cha01] B. Chazelle. *The Discrepancy Method: Randomness and Complexity*. Cambridge University Press, New York, 2001.

[Cha02] T. M. Chan. Closest-point problems simplified on the ram. In *Proc. 13th ACM-SIAM Sympos. Discrete Algorithms*, pages 472–473. Society for Industrial and Applied Mathematics, 2002.

[CK95] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. Assoc. Comput. Mach.*, 42:67–90, 1995.

[Cla83] K. L. Clarkson. Fast algorithms for the all nearest neighbors problem. In *Proc. 24th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 226–232, 1983.

[Cla94] K. L. Clarkson. An algorithm for approximate closest-point queries. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 160–164, 1994.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press / McGraw-Hill, Cambridge, Mass., 2001.

[CRT05] B. Chazelle, R. Rubinfeld, and L. Trevisan. Approximating the minimum spanning tree weight in sublinear time. *SIAM J. Comput.*, 34(6):1370–1379, 2005.

[dBCvKO08] M. de Berg, O. Cheong, M. van Kreveld, and M. H. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3 edition, 2008.

[dHTT07] M. de Berg, H. J. Haverkort, S. Thite, and L. Toma. I/o-efficient map overlay and point location in low-density subdivisions. In *Proc. 18th Annu. Internat. Sympos. Algorithms Comput.*, pages 500–511, 2007.

[Dun99] C. A. Duncan. *Balanced Aspect Ratio Trees*. Ph.D. thesis, Department of Computer Science, Johns Hopkins University, Baltimore, Maryland, 1999.

[EGS05] D. Eppstein, M. T. Goodrich, and J. Z. Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. In *Proc. 21st Annu. ACM Sympos. Comput. Geom.*, pages 296–305. ACM, June 2005.

[Fel91] W. Feller. *An Introduction to Probability Theory and its Applications*. John Wiley & Sons, NY, 1991.

[FH05] J. Fischer and S. Har-Peled. Dynamic well-separated pair decomposition made easy. In *CCCG*, pages 235–238, 2005.

[FIS05] G. Frahling, P. Indyk, and C. Sohler. Sampling in dynamic data streams and applications. In *Proc. 21st Annu. ACM Sympos. Comput. Geom.*, pages 142–149, 2005.

[Gar82] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, 1982.

[GRSS95] M. Golin, R. Raman, C. Schwarz, and M. Smid. Simple randomized algorithms for closest pair problems. *Nordic J. Comput.*, 2:3–27, 1995.

[Har01a] S. Har-Peled. A practical approach for computing the diameter of a point-set. In *Proc. 17th Annu. ACM Sympos. Comput. Geom.*, pages 177–186, 2001.

[Har01b] S. Har-Peled. A replacement for Voronoi diagrams of near linear size. In *Proc. 42nd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 94–103, 2001.

[HM85] D. S. Hochbaum and W. Maas. Approximation schemes for covering and packing problems in image processing and VLSI. *J. Assoc. Comput. Mach.*, 32:130–136, 1985.

[HM03] S. Har-Peled and S. Mazumdar. Fast algorithms for computing the smallest k-enclosing disc. In *Proc. 11th Annu. European Sympos. Algorithms*, volume 2832 of *Lect. Notes in Comp. Sci.*, pages 278–288. Springer-Verlag, 2003.

[HM04] S. Har-Peled and S. Mazumdar. Coresets for k-means and k-median clustering and their applications. In *Proc. 36th Annu. ACM Sympos. Theory Comput.*, pages 291–300, 2004.

[HM06] S. Har-Peled and M. Mendel. Fast construction of nets in low dimensional metrics, and their applications. *SIAM J. Comput.*, 35(5):1148–1184, 2006.

[HÜ05] S. Har-Peled and A. Üngör. A time-optimal delaunay refinement algorithm in two dimensions. In *Proc. 21st Annu. ACM Sympos. Comput. Geom.*, pages 228–236, 2005.

[IM98] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. 30th Annu. ACM Sympos. Theory Comput.*, pages 604–613, 1998.

[Ind04] P. Indyk. Nearest neighbors in high-dimensional spaces. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 39, pages 877–892. CRC Press LLC, Boca Raton, FL, 2nd edition, 2004.

[KF93] I. Kamel and C. Faloutsos. On packing r-trees. In *Proc. 2nd Intl. CConf. Info. Knowl. Mang.*, pages 490–499, 1993.

[KLN99] D. Krznaric, C. Levcopoulos, and B. J. Nilsson. Minimum spanning trees in d dimensions. *Nordic J. Comput.*, 6(4):446–461, 1999.

[Mat95] J. Matoušek. On enclosing k points by a circle. *Inform. Process. Lett.*, 53:217–221, 1995.

[Mil04] G. L. Miller. A time efficient Delaunay refinement algorithm. In *Proc. 15th ACM-SIAM Sympos. Discrete Algorithms*, pages 400–409, 2004.

[MN98] J. Matoušek and J. Nešetřil. *Invitation to Discrete Mathematics*. Oxford Univ Pr, 1998.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, 1995.

[NZ01] G. Narasimhan and M. Zachariasen. Geometric minimum spanning trees via well-separated pair decompositions. *J. Exp. Algorithmics*, 6:6, 2001.

[Rab76] M. O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 21–39. Academic Press, New York, NY, 1976.

[Rup93] J. Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 83–92, 1993.

[Rup95] J. Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1995.

[Sag94] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.

[Sam89] H. Samet. *Spatial Data Structures: Quadtrees, Octrees, and Other Hierarchical Methods*. Addison-Wesley, Reading, MA, 1989.

[Sam05] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling. Morgan Kaufmann Publishers Inc., 2005.

[Smi00] M. Smid. Closest-point problems in computational geometry. In Jörg-Rüdiger Sack and Jorge Urrutia, editors, *Handbook of Computational Geometry*, pages 877–935. Elsevier Science Publishers B. V. North-Holland, Amsterdam, 2000.

[Üng09] A. Üngör. Off-centers: A new type of steiner points for computing size-optimal quality-guaranteed delaunay triangulations. *Comput. Geom. Theory Appl.*, 42(2):109–118, 2009.

[Vai86] P. M. Vaidya. An optimal algorithm for the all-nearest-neighbors problem. In *Proc. 27th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 117–122, 1986.

[Var98] K. R. Varadarajan. A divide-and-conquer algorithm for min-cost perfect matching in the plane. In *Proc. 39th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 320–331, 1998.

[WVTP97] M. Waldvogel, G. Varghese, J. Turener, and B. Plattner. Scalable high speed ip routing lookups. In *Proc. ACM SIGCOMM 97*, Octeber 1997.