Chapter 1

The Power of Grids - Computing the Minimum Disk Containing k Points

> 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 r and a point $p = (x, y)$ in $\mathbb{R}^2$, define $\mathsf{G}_r(p)$ to be the grid point $(\lfloor x/r \rfloor r, \lfloor y/r \rfloor r)$. We call r the ***width*** of the ***grid*** G_r. Observe that G_r 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 $x \geq ri$, $x < r(i+1)$, $y \geq rj$ and $y < r(j+1)$ 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 C of G_r, has a unique ID; indeed, let $p = (x, y)$ be any point in C, and consider the pair of integer numbers $\mathrm{id}_C = \mathrm{id}(p) = (\lfloor x/r \rfloor, \lfloor y/r \rfloor)$. Clearly, only points inside C are going to be mapped to id_C. This is very useful, since we store a set P of points inside a grid efficiently. Indeed, given a point p, compute its $\mathrm{id}(p)$. We associate with each unique id a data-structure that stores all the points falling into this grid cell (of course, we do not maintain such data-structures for grid cells which are empty). So, once we computed $\mathrm{id}(p)$, 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]).*

For a point set P, and parameter r, the partition of P into subsets by the grid G_r, is denoted by $\mathsf{G}_r(\mathsf{P})$. More formally, two points $p, q \in \mathsf{P}$ belong to the same set in the partition $\mathsf{G}_r(\mathsf{P})$, if both points are being mapped to the same grid point or equivalently belong to the same grid cell.

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 $\mathcal{CP}(\mathsf{P}) = \min_{p,q \in P} \|p - q\|$.

Lemma 1.2.2 *Given a set P of n points in the plane, and a distance r, one can verify in linear time, whether $\mathcal{CP}(\mathsf{P}) < r$, $\mathcal{CP}(\mathsf{P}) = r$, or $\mathcal{CP}(\mathsf{P}) > r$.*

Proof: Indeed, store the points of P in the grid G_r. 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. Indeed, compute $\mathrm{id}(p)$, check if $\mathrm{id}(p)$ 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 data-structure already exist for $\mathrm{id}(p)$, just add p to it.

This takes $O(n)$ time. Now, if any grid cell in $\mathsf{G}_r(\mathsf{P})$ contains more than, say, 9 points of p, then it must be that the $\mathcal{CP}(\mathsf{P}) < r$. Indeed, consider a cell C containing more than nine points of P, and partition C into 3×3 equal squares. Clearly, one of those squares must contain two points of P, and let C' be this square. Clearly, the diameter of $C' = \mathrm{diam}(C)/3 = \sqrt{r^2 + r^2}/3 < r$. Thus, the two (or more) points of P in C' are at distance smaller than r from each other.

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. All those cells, must contain at most 9 points of P (otherwise, we would already have stopped since the $\mathcal{CP}(\cdot)$ of inserted points, is smaller than r). Let S be the set of all those points, and observe that $|S| \leq 9 \cdot 9 = O(1)$. Thus, we can compute by brute force the closest point to p in S. This takes $O(1)$ time. If $\mathbf{d}(p, S) < r$, we stop, otherwise, we continue to the next point.

Overall, this takes $O(n)$ time. As for correctness, first observe that if $\mathcal{CP}(\mathsf{P}) > r$ then the algorithm would never make a mistake, since it returns '$\mathcal{CP}(\mathsf{P}) < r$' only after finding a pair of points of P with distance smaller than r. Thus, assume that p, q are the pair of points of P realizing the closest pair, and $\|pq\| = \mathcal{CP}(\mathsf{P}) < r$. Clearly, when the later of them, say p, is being inserted, the set S would contain q, and as such the algorithm would stop and return '$\mathcal{CP}(\mathsf{P}) < r$'. ■

Lemma 1.2.2 provides a natural way of computing $\mathcal{CP}(\mathsf{P})$. Indeed, permute the points of P in arbitrary fashion, and let $\mathsf{P} = \langle p_1, \ldots, p_n \rangle$. Next, let $r_i = \mathcal{CP}(\{p_1, \ldots, p_i\})$. We can check if $r_{i+1} < r_i$, by just calling the algorithm for Lemma 1.2.2 on P_{i+1} and r_i. In fact, if $r_{i+1} < r_i$, the algorithm of Lemma 1.2.2, would give us back the distance r_{i+1} (with the other point realizing this distance).

So, consider the "good" case, where $r_i = r_{i-1}$; that is, the length of the shortest pair does not change when p_i is inserted. In this case, we do not need to rebuild the data structure of Lemma 1.2.2 for the ith point. We can just reuse it from the previous iteration 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 $r_i < r_{i-1}$, because then we need to rebuild the grid data structure, and reinsert all the points of $\mathsf{P}_i = \langle p_1, \ldots, p_{i+1} \rangle$ into the new grid $\mathsf{G}_{r_i}(\mathsf{P}_i)$. This takes $O(i)$ time.

Specifically, if the closest pair distance, in the sequence $r_1, \ldots, r_n$, changes only k times, then the running time of our algorithm would be $O(nk)$. In fact, we can do even better.

Theorem 1.2.3 *For set P of n points in the plane, one can compute the closest pair of P in expected linear time.*

Proof: Pick a random permutation of the points of P, let $\langle p_1, \ldots, p_n \rangle$ be this permutation. Let $r_2 = \|p_1 p_2\|$, and start inserting the points into the data structure of Lemma 1.2.2. In the ith iteration, if $r_i = r_{i-1}$, then this insertion takes constant time. If $r_i < r_{i-1}$, then we rebuild the grid and reinsert the points. Namely, we recompute $\mathsf{G}_{r_i}(\mathsf{P}_i)$.

To analyze the running time of this algorithm, let X_i be the indicator variable which is 1 if $r_i \neq r_{i-1}$, and 0 otherwise. Clearly, the running time is proportional to

$$R = 1 + \sum_{i=2}^{n} (1 + X_i \cdot i).$$

Thus, the expected running time is

$$\mathbf{E}[R] = \mathbf{E}\left[1 + \sum_{i=2}^{n} (1 + X_i \cdot i)\right] = n + \sum_{i=2}^{n} (\mathbf{E}[X_i] \cdot i) = n + \sum_{i=2}^{n} i \cdot \mathbf{Pr}[X_i = 1],$$

by linearity of expectation and since for indicator variable X_i, we have $\mathbf{E}[X_i] = \mathbf{Pr}[X_i = 1]$.

Thus, we need to bound $\mathbf{Pr}[X_i = 1] = \mathbf{Pr}[r_i < r_{i-1}]$. To bound this quantity, fix the points of P_i, and randomly permute them. A point $q \in \mathsf{P}_i$ is called ***critical***, if $\mathcal{CP}(\mathsf{P}_i \setminus \{q\}) > \mathcal{CP}(\mathsf{P}_i)$. If there are no critical points, then $r_{i-1} = r_i$ and then $\mathbf{Pr}[X_i = 1] = 0$. 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.

If 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 r_i is smaller than r_{i-1}, one if either p or q are p_i. But 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 realizing the closest distance, than if there is a third critical point r, then $\mathcal{CP}(\mathsf{P}_i \setminus \{r\}) = \|pq\|$, and r is not critical.

We conclude that

$$\mathbf{E}[R] = n + \sum_{i=2}^{n} i \cdot \mathbf{Pr}[X_i = 1] \leq n + \sum_{i=2}^{n} i \cdot \frac{2}{i} \leq 3n,$$

and the expected running time is $O(\mathbf{E}[R]) = O(n)$. ■

Theorem 1.2.3 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. However, there is a lower bound of $\Omega(n \log n)$ on uniqueness, using the comparison model. This reality dysfunction can be easily explained once one realizes that the computation of Theorem 1.2.3 is considerably stronger, using hashing, randomization, and the floor function.

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

For a circle 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 $D_{\text{opt}}(\mathsf{P}, k)$.

Let P be a set of n points in the plane. Compute a set of $m = O(n/k)$ horizontal lines $h_1, \ldots, h_m$ such that between two consecutive horizontal lines, there are at most $k/4$ points of P in the strip they define. This can be easily done in $O(n \log(n/k))$ time using deterministic median selection together with recursion.[2] Similarly, compute a set of vertical lines $v_1, \ldots, v_m$, such that between two consecutive lines, there are at most $k/4$ points of P.

[2] Indeed, compute the median in the x-order of the points of P, split P into two sets, and recurse on each set, till the number of points in a subproblem is of size $\leq k/4$. We have $T(n) = O(n) + 2T(n/2)$, and the recursion stops for $n \leq k/4$. Thus, the recursion tree has depth $O(\log(n/k))$, which implies running time $O(n \log(n/k))$.