Chapter III

Closest Pair of Points

We consider the problem finding a closest pair of points in a given set of points in the plane. More precisely, we are given a set of \( n \) points in \( \mathbb{R}^2 \) – each is a pair \( x = (x_1, x_2) \) – and we are interested in a pair of points \( p, q \in P \) such that

\[
\|p - q\| = \min \{\|p' - q'\| : p', q' \in P, p' \neq q'\}
\]

where \( \|p - q\| = \sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2} \) is the euclidean distance between \( p = (p_1, p_2) \) and \( q = (q_1, q_2) \).

The 1-dimensional version of this problem is equivalent to finding the smallest interval determined by \( n \) numbers, which is obviously determined by two consecutive numbers. To solve this problem, it suffices to sort the numbers and this results in a running time \( O(n \log n) \). Can this running time be improved? It turns out that if only certain computation primitives – which most algorithms are restricted to use – then this is the best possible running time (if there is time – which is unlikely, we may talk about this later in the course). On the other hand, if more powerful primitives are allowed, the running time can be reduced to (expected) \( O(n) \).

We first describe a natural divide-and-conquer algorithm with a running time \( O(n \log n) \), and then a randomized incremental algorithm with an expected running time \( O(n) \). For the latter, we make use of stronger primitives: random bits, a table that allows \( O(1) \) access and update time (hash tables), and the floor function.

Let \( \delta(P) \) denote the smallest distance between different points in \( P \). In general, there is no unique closest pair, so our algorithms will return a pair that realizes the smallest distance \( \delta(P) \).

III.1 Divide-And-Conquer

A natural way to divide the problem is through a line, and particularly a vertical one. This divides the set \( P \) of points into \( P_L \) and \( P_R \) for which the problem is solved recursively. It remains to specify how to merge these subproblems.
**Closest-Pair-DC** \((P)\)

Partition \(P\) with a vertical halving line \(\ell\) into points to the left \(P_L\) and to the right \(P_R\)

\[(p_L, q_L) \leftarrow \text{Closest-Pair-DC}(P_L)\]

\[(p_L, q_L) \leftarrow \text{Closest-Pair-DC}(P_R)\]

\[\delta \leftarrow \min\{\|p_L - q_L\|, \|p_R - q_R\|\}\]

\[(p_M, q_M) \leftarrow \text{CP-Merge}(P, \ell, \delta)\]

return closest pair among \((p_L, p_L), (p_R, q_R), (p_M, q_M)\)

From the recursion, we know closest pairs with both points on the left and on the right of \(\ell\). It remains to check for a closest pair with one point on the left and one on the right. In principle, we would have to check every such pair, and that would be very time consuming. We observe that:

(i) Only points within a strip of bounded by lines \(\ell^-\) and \(\ell^+\) parallel to \(\ell\) and at distance \(\delta\) on the left and right (see figure on the left): a point outside this strip would be at least distance \(\delta\) away from a point on the opposite side. Let \(P_{\text{strip}}\) denote the subset of \(P\) in this strip.

(ii) Each point in this strip only need checked against four other points, that can be easily determined: To see this, consider the lowest point \(q\) of a possible closest pair. Suppose \(q\) is on the right. Then a candidate \(p\) to be closest to \(q\) must lie in the square determined by the lines \(\ell^-\), \(\ell\), \(x_2 = q_2\) and \(x_2 = q_2 + \delta\) (see right figure). This square contains at most 4 points: if there are more than 4, then there are 2 in one of the “subsquares” (see figure) and so the distance between them is

\[
\sqrt{(\delta/2)^2 + (\delta/2)^2} = \delta/\sqrt{2} < \delta,
\]

a contradiction.

\(P_{\text{strip}}\) is computed by simply scanning \(P\) and selecting the points within distance \(\delta\) from \(\ell\). If \(P_{\text{strip}}\) is sorted in increasing order of \(x_2\)-coordinate, then (ii) implies that each point in \(P_{\text{strip}}\) only need to be considered to the next 7 points in that list (at most 4 points on the opposite side are relevant, but other 3 on the same side may appear between them in the list).
The overall algorithm first sorts $P$ according to both coordinates to obtain an $x_1$-list ordered by $x_1$-coordinate, and an $x_2$-list ordered according to the $x_2$-coordinate. Using these, both steps, divide and merge take $O(n)$ time: dividing uses the $x_1$-list to find the halving line and both subproblems inherit the $x_1$- and $x_2$- order; merging uses the $x_1$-list to get $P_{\text{strip}}$, and then the $x_2$-list as just described to determine the candidate pairs. We omit the detailed pseudocode for CP-MERGE, which should be clear by now.

Excluding the initial sorting time, then the running time $T(n)$ of the algorithm satisfies the recurrence $T(n) \leq 2T(n/2) + O(n)$, which has solution $T(n) = O(n \log n)$. Hence, the overall time, also including the initial sorting, is $O(n \log n)$.

### III.2 Randomized Incremental

Now, we consider a randomized incremental algorithm. It uses also the ability to “hash” points to its cell in a uniform grid. Let $r > 0$ be a real number $r > 0$. The grid $G_r$ is determined by the vertical lines $x_1 = k_1 \cdot r$ and the horizontal lines $x_2 = k_2 \cdot r$, where $k_1, k_2 \in \mathbb{Z}$. A point $(p_1, p_2)$ belongs to the grid cell $gc_r(p) = ([p_1/r], [p_2/r])$ in $G_r$ (a cell includes its lower and left bounding edges, but excludes its right and upper ones). The grid could have an offset rather than have its origin $(0, 0)$ coincide with the origin $(0, 0)$ of the coordinate system; we ignore this possibility in what follows. In the algorithm we will describe, it is assumed that $gc_r(p)$ can be computed in time $O(1)$ and that we have at our disposal a table data structure (hash table) that allows grid cell identifiers to be stored with $O(1)$ search, insertion and deletion time. We will see later in the course, how such data structures can be implemented.

A simple observation that will be essential in the algorithm is that given a point $p$, a point that is at distance smaller than $r$ from $p$ must be either in $gc_r(p)$ or in one of the 8 neighboring cells (see figure). We denote this neighborhood by $Nh_r(p)$ (all 9 cells, including $gc_r(p)$).

The algorithm is randomized incremental. It considers a random permutation $p_1, p_2, p_3, \ldots, p_n$ of the input set $P$ of points. Let $P_i$ be $\{p_1, p_2, \ldots, p_i\}$, the $i$-th prefix in the permutation. Let $\delta_i = \delta(P_i)$, that is, the shortest distance in $P_i$. Starting with $i = 2$, at the end of the $i$-th iteration the algorithm has considered $P_i$ and has computed $\delta_i$ (and also a closest pair), and it has stored $gc_{\delta_j}(p_j)$, for $j = 1, \ldots, i$ in a hash table $T_i$. In the $(i + 1)$-st iteration, the algorithm determines the new shortest distance $\delta_{i+1}$ as follows:
1. get the set \( N(p) \) of all points in \( N_{h_{r+1}}(p_i) \) (stored in \( T_i \))

2. \( \delta_i \leftarrow \min \{ \| p_{i+1} - q \| : q \in N(p_{i+1}) \} \).

We claim that \( N(p) \) has size \( O(1) \): by the same argument as earlier, each of the cells has at most 4 points, so in total there are at most 36 (actually less, since the maximum 4 is achieved for 4 points in the corners of a cell and these would be shared among cells). Furthermore, these can be determined in time \( O(1) \) making use of the table \( T_i \) which has access time \( O(1) \). Thus, in time \( O(1) \), the algorithm can compute \( \delta_{i+1} \) and so decide whether \( \delta_i = \delta_{i+1} \) or \( \delta_i > \delta_{i+1} \).

The case in which \( \delta_i < \delta_{i-1} \) is very expensive computationally: it takes time \( \Theta(i) \) under the assumption of a table with \( O(1) \) access/update time. It is easy to construct an example of a sequence \( p_1, p_2, \ldots, p_n \) for which this happens for every \( i \). However, we are considering a random permutation of the input: the idea is that on the average the algorithm will do well (notice that as in other cases already considered, this is an average over the randomization introduced by the algorithm, and it holds for any input—even the bad example just mentioned).

**Running Time Analysis.** Now we compute the expected running time. Let \( X_i \) be the following 0/1 random variable:

\[
X_i = [\delta_i < \delta_{i-1}].
\]

The total running time is upper bounded by

\[
O \left( \sum_{i=3}^{n} (1 + i \cdot X_i) \right).
\]

Using as usual linearity of expectation and that \( X_i \) is a 0/1 r.v., we have

\[
\mathbb{E} \left[ \sum_{i=3}^{n} (1 + i \cdot X_i) \right] \leq n + \sum_{i=1}^{n} i \cdot \mathbb{P}(X_i = 1).
\]

**Claim 1.** \( \mathbb{P}(X_i = 1) \leq 2/i \).
Proof. We want to upper bound \( \Pr \{ \delta_i < \delta_{i-1} \} \). We consider the random permutation from \( P \) formed as follows: first \( P_i \) is a subset from \( P \) of size \( i \) selected uniformly at random (among all the subsets of size \( i \)), then we permute \( P_i \) randomly, and follow this with a random permutation of \( P - P_i \).

We consider first

\[
\Pr \{ \delta_i < \delta_{i-1} \mid P_i = P_i^* \},
\]

that is, the probability conditioned to a particular outcome \( P_i^* \) during the random selection of \( P_i \). Let’s see how it could happen that \( \delta_i < \delta_{i-1} \). For this we consider the following cases (it depends on whether \( \delta_i \) is realized by one or more pairs of points):

(i) \( \delta_i \) is realized by a single pair of points: say \( p, q \), then \( \delta_i < \delta_{i-1} \) iff \( p_i \) is either \( p \) or \( q \); since \( p_1, p_2, \ldots, p_i \) is a random permutation of \( P_i^* \), then \( \Pr \{ \delta_i < \delta_{i-1} \mid P_i = P_i^* \} \) is \( 2/i \).

(ii) \( \delta_i \) is realized by two or more disjoint pairs of points: say \( p, q \) and \( p', q' \); then no matter what point \( p_i \) is \( \delta_i = \delta_{i-1} \) and so \( \Pr \{ \delta_i < \delta_{i-1} \mid P_i = P_i^* \} \) is 0.

(iii) \( \delta_i \) is realized by two or more non disjoint pairs (but not by two or more disjoint pairs): say \( p, q \) and \( p, q' \); then \( \delta_i < \delta_{i-1} \) only if \( p_i = p \) and so \( \Pr \{ \delta_i < \delta_{i-1} \mid P_i = P_i^* \} = 1/i \).

Thus, putting together all cases:

\[
\Pr \{ \delta_i < \delta_{i-1} \mid P_i = P_i^* \} \leq \frac{2}{i}.
\]

The bound is independent of the conditioning on \( P_i \), so it can be removed and we obtain:

\[
\Pr \{ \delta_i < \delta_{i-1} \} \leq \frac{2}{i}.
\]

(Note that the random permutation on \( P - P_i \) is irrelevant.)

Substituting this upper bound in our expression upper bounding the expected running time, we get

\[
n + \sum_{i=1}^{n} i \cdot \frac{2}{i} \leq 3n.
\]

So the expected running time is \( O(n) \).