CH09 – Divide & Conquer and Analysis of Recurrences (03)

Finding the closest pair of points

Closest refers to the Euclidean distance: the straight-line distance between points $P_1(x_1, x_y)$ and $P_1(x_2, 2)$ is $\sqrt{(x_1 - x_2)^2 + (y_1 - y)^2}$

Application example, in traffic control systems: identify two closest vehicles in order to detect potential collisions.

Brute force algorithm: look at all of the pairs of points: $\binom{n}{2}$ = Θ(n^2).

We develop a divide-and-conquer approach whose running time is described by the familiar recurrence:

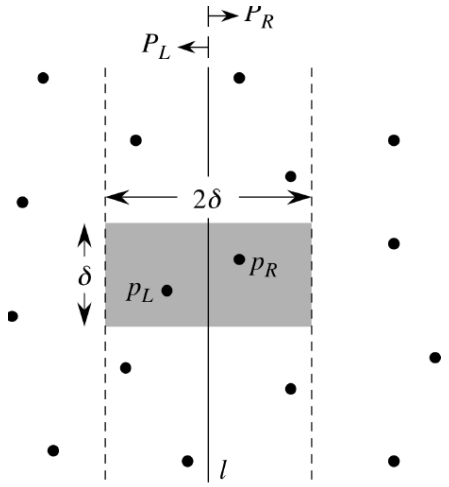
$2T(n/2) + O(n)$. Thus this algorithm uses only O(n lg n) time.

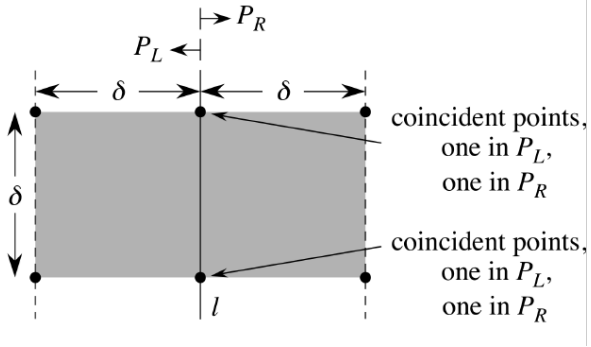
1. Sort points according to their x-coordinates. And sort the points again by their y-coordinates.
2. Split the set of points P into two equal-sized subsets P_L and P_R by a vertical line *l*.
3. Solve the problem recursively in the left and right subsets. This yields the left-side and right-side minimum distances δ_L and δ_R, respectively.
4. Determine if any pair of points with one point in P_L and the other in P_R has distance less than δ.
5. The final answer is the minimum among δ_L, δ_R, and the minimum distance found in step

The key to accomplishing the O(n lg n) time is accomplishing step 4 in linear time. A naive approach would require the calculation of distances for all left-right pairs, but this would take quadratic time.

The key observation is based on the following sparsity property of the point set. We already know that the closest pair of points is no further apart than $\delta = \min(\delta_L, \delta_R)$, where δ_L, is the shortest distance between points on the left side of *l* and is the δ_R, is the shortest distance between points on the right side of *l*.

Therefore, for each point P_L to the left of the dividing line we have to compare the distances to the points that lie in the rectangle of dimensions (δ, 2 δ) to the right of the dividing line, as shown in the figure below. Additionally, this rectangle can contain at most eight points with pairwise distances at least δ. Therefore, it is sufficient to compute at most 7 pairwise distances comparisons for each point in P_L.


P_R
P_L
2δ
δ
p_L
p_R
l


P_R
P_L
δ
δ
δ
coincident points, one in P_L, one in P_R
coincident points, one in P_L, one in P_R
l

The divide-and-conquer algorithm (details from our optional textbook)

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

Each recursive call of the algorithm takes as input a subset P of Q and arrays X and Y, each of which contains all the points of the input set P:

- X contains the elements of P sorted in increasing order of the x- coordinate
- Y contains the elements of P sorted in increasing order of the y- coordinate

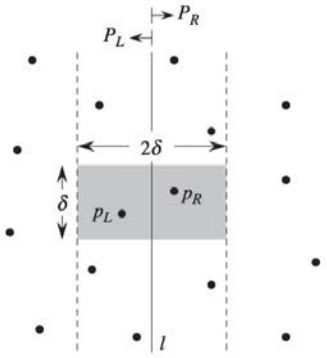
A given recursive invocation with inputs P, X, and Y first checks if the size of P is less than or equal to 3. If so, find the closest points in P with the brute-force approach, by trying all pairs of points. If $P > 3$, the recursive invocation carries out the divide-and-conquer paradigm as follows.

Divide: Find a vertical line l that bisects the point set P into two sets P_L and P_R such that $|P_L| = ceiling(|P|/2)$, $|P_R| = floor(|P|/2)$, all points in P_L are on or to the left of line l, and all points in P_R are on or to the right of l. Divide the array X into arrays X_L and X_R, which contain the points of P_L and P_R respectively, sorted by monotonically increasing x-coordinate. Similarly, divide the array Y into arrays Y_L and Y_R, which contain the points of P_L and P_R respectively, sorted by monotonically increasing y-coordinate.

Conquer: Having divided P into P_L and P_R, make two recursive calls, one to find the closest pair of points in P_L and the other to find the closest pair of points in P_R. The inputs to the first call are the subset P_L and arrays X_L and Y_L; the second call receives the inputs P_R, X_R, and Y_R. Let the closest-pair distances returned for P_L and P_R be δ_L and δ_R, respectively, and let $\delta = \min(\delta_L, \delta_R)$.

Combine: The closest pair is either the pair with distance δ found by one of the recursive calls, or it is a pair of points with one point in P_L and the other in P_R. The algorithm determines whether there is a pair with one point in P_L and the other point in P_R and whose distance is less than δ. Observe that if a pair of points has distance less than δ, both points of the pair must be within δ units of line l. Thus, they both must reside in the 2 δ-wide vertical strip centered at line l. To find such a pair, if one exists, we do the following:

1. Create an array Y', which is the array Y with all points not in the 2 δ- wide vertical strip removed. The array Y' is sorted by y-coordinate, just as Y is.

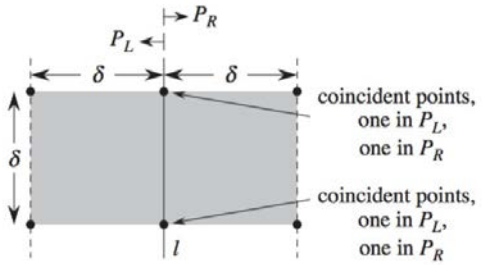


2. For each point p in the array Y', try to find points in Y' that are within δ units of p. As we shall see shortly, only the 7 points in Y' that follow p need be considered. Compute the distance from p to each of these 7 points, and keep track of the closest-pair distance δ' found over all pairs of points in Y'.
3. If $\delta' < \delta$, then the vertical strip does indeed contain a closer pair than the recursive calls found. Return this pair and its distance δ'. Otherwise, return the closest pair and its distance δ found by the recursive calls.

Why are seven points sufficient for lookup?

We shall prove that we need only check the seven points following each point p in array Y'.

$p_R \in P_R$. Thus, the distance δ' between p_L and p_R is strictly less than δ. Point p_L must be on or to the left of line l and less than δ units away. Similarly, p_R is on or to the right of l and less than δ units away. Moreover, p_L and p_R are within δ units of each other vertically. Thus, as Figure below shows, p_L and p_R are within a δ 2δ rectangle centered at line l. (There may be other points within this rectangle as well.)



We next show that at most 8 points of P can reside within this $\delta \times 2\delta$ rectangle. Consider the $\delta \times \delta$ square forming the left half of this rectangle. Since all points within P_L are at least δ units apart, at most 4 points can reside within this square; Figure 33.11(b) shows how.

Similarly, at most 4 points in P_R can reside within the $\delta \times \delta$ square forming the right half of the rectangle. Thus, at most 8 points of P can reside within the $\delta \times 2\delta$ rectangle. (Note that since points on line l may be in either P_L or P_R, there may be up to 4 points on l. This limit is achieved if there are two pairs of coincident points such that each pair consists of one point from P_L and one point from P_R, one pair is at the intersection of l and the top of the rectangle, and the other pair is where l intersects the bottom of the rectangle.)

Having shown that at most 8 points of P can reside within the rectangle, we can easily see why we need to check only the 7 points following each point in the array Y'. Still assuming that the closest pair is p_L and p_R, let us assume without loss of generality that p_L precedes p_R in array Y'. Then, even if p_L occurs as early as possible in Y' and p_R occurs as late as possible, p_R is in one of the 7 positions following p_L. Thus, we have shown the correctness of the closest-pair algorithm.

Another key implementation issue

How to ensure that the arrays X_L, X_R, Y_L, and Y_R, which are passed to recursive calls, are sorted by the proper coordinate and also that the array Y' is sorted by y-coordinate? Note that if the array X that is received by a recursive call is already sorted, then we can easily divide set P into P_L and P_R in linear time.

The following algorithm splits Y into Y_L and Y_R

```
1  let Y_L[1..Y.length] and Y_R[1..Y.length] be new arrays
2  Y_L.length = Y_R.length = 0
3  for i = 1 to Y.length
4      if Y[i] ∈ P_L
5          Y_L.length = Y_L.length + 1
6          Y_L[Y_L.length] = Y[i]
7      else Y_R.length = Y_R.length + 1
8          Y_R[Y_R.length] = Y[i]
```