CS 218 : Design and Analysis of Algorithms

Lecture 9.2: Finding a Pair of Closest Points: Algorithm 2

Lecturer: *Sundar Vishwanathan*
Computer Science & Engineering — Indian Institute of Technology, Bombay

The initial part of this lecture is the same as the previous.

1 Finding Closest Points

1.1 Problem Statement

Our next problem is from computational geometry. **Input**: A set of n points specified by their x and y co-ordinates. **Output**: The closest pair of points.

The distance between two points is the usual: $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

A naive algorithm takes $O(n^2)$ time. We, as usual, aim for something better. Why do we even suspect that this is possible for this problem?

Design Tip for Geometric Problems. First work with the same problem in smaller dimensions. In one dimension the problem reduces to finding the closest pair of points on a line. Say the x-axis. Try this before reading ahead.

The solution is to sort and check distances between adjacent points. Time is $O(n \log n)$. So, there is hope.

The obvious problem in two dimensions is that there is no one notion of sorting. In one dimension, for each point, the number of points that can be at the shortest distance away from this point is at most two. In two dimensions this can clearly be $n-1$. Draw a figure to convince yourself of this.

The algorithm we described above was construted in an ad-hoc manner. It will be useful to construct it using the principles that we have discussed so far. The simplest inductive approach is this: remove one point, recurse on the rest and construct the solution with this point put in. We remind the reader that we are still working in one dimension. The algorithm is first remove one element, then recursively find the closest pair among the rest, and now compute the minimum distance of the point removed from the other points. Compare and output the result.

As usual we notice that if we soup up the induction and return the sorted order we can place the new point ($O(\log n)$ time) and then finish in $O(1)$ time. This of course leads to insertion sort-a familiar problem with familiar solutions. One is to use a balanced binary search tree and the other is a merge sort type recursion. We focus on the first route in this lecture. Can we extend this to 2D? What are the problems we face? The key problem remains the same: no notion of order amongst the points. What should the inductive call return in addition to the closest pair? The obvious way to complete the induction is to compute the closest distance of old points from the last point quickly. This seems hopeless. Viewed in isolation we can show that this requires us to compute $O(n)$ distances. See if you can argue how before reading ahead. Put the new point at the center of a circle and the old points on the perimeter. Now arbitrarily move one of the points on the perimeter slightly inwards. It is easy to see that all $n-1$ distances have to be computed to find the minimum.

Note. Data structures for the problem of finding the closest point in a set S to a query point p is called *the nearest neighbor problem* and is well studied in the community.

To see that we still have a chance, notice that if this were the configuration then the closest point will be between two old points. This naturally leads us to the new algorithmic question to settle the inductive step. The operations we want from the data structure is **insert** and **locate**(p, d) which finds all points at a distance at most d from the point p. The previous intuition tells us that there cannot be too many of them.

We can simplify our problem further since we are given all points right at the beginning. We have a choice asto which point to remove: order makes sense here. We will sort the points by x-co-ordinate and remove the last point. We then have to only look to the left for the last point. The recursion returns the minimum distance seen so far, d. Let the last point be (x', y'). Now notice that we may only concentrate on points with x-co-ordinate at least $x' - d$. Among these points we need to isolate points with y-co-ordinates between $y' - d$ and $y' + d$. The data structure needs to support the following operations: **insert**, **delete**, and **find-in-range**. This can be done with a BBST. Note that the points will be inserted/deleted in the order of their x-coordinates. So we need to maintain the points sorted in their x-coordinates as well. The key fact is that we need to look at only a constant number of points around y'. Why?

Algorithm Summary. We maintain a BBST of y-coordinates and a sorted array of the x-coordinates. Consider the points in increasing order of their x-coordinates. For a new point (x', y'), we first delete from the BBST, all points whose x-coordinate is less than $x' - d$. Note that this is easy since we have the sorted list on the x-coordinates. Now find y' in the BBST. Consider the four points before y' and after y' using successor and predecessor functions. Compute the distances and update the minimum if needed. Finally insert y'.