Lecture 31: Divide-and-Conquer: Closest Pair

**Closest Pair**: Today, we consider another application of divide-and-conquer, which comes from the field of computational geometry. We are given a set $P$ of $n$ points in the plane, and we wish to find the closest pair of points $p, q \in P$ (see Fig. 105(a)). This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision. Recall that, given two points $p = (p_x, p_y)$ and $q = (q_x, q_y)$, their (Euclidean) distance is

$$
\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}.
$$

Clearly, this problem can be solved by brute force in $O(n^2)$ time, by computing the distance between each pair, and returning the smallest. Today, we will present an $O(n \log n)$ time algorithm, which is based on a clever use of divide-and-conquer.

Before getting into the solution, it is worth pointing out a simple strategy that fails to work. If two points are very close together, then clearly both their $x$-coordinates and their $y$-coordinates are close together. So, how about if we sort the points based on their $x$-coordinates and, for each point of the set, we’ll consider just nearby points in the list. It would seem that (subject to figuring out exactly what “nearby” means) such a strategy might be made to work. The problem is that it could fail miserably. In particular, consider
Fig. 105: (a) The closest pair problem and (b) why sorting on $x$- or $y$-alone doesn’t work.

the point set of Fig. 105(b). The points $p$ and $q$ are the closest points, but we can place an arbitrarily large number of points between them in terms of their $x$-coordinates. We need to separate these points sufficiently far in terms of their $y$-coordinates that $p$ and $q$ remain the closest pair. As a result, the positions of $p$ and $q$ can be arbitrarily far apart in the sorted order. Of course, we can do the same with respect to the $y$-coordinate. Clearly, we cannot focus on one coordinate alone.$^{16}$

**Divide-and-Conquer Algorithm:** Let us investigate how to design an $O(n \log n)$ time divide-and-conquer approach to the problem. The input consists of a set of points $P$, represented, say, as an array of $n$ elements, where each element stores the $(x, y)$ coordinates of the point. (For simplicity, let’s assume there are no duplicate $x$-coordinates.) The output will consist of a single number, being the closest distance. It is easy to modify the algorithm to also produce the pair of points that achieves this distance.

For reasons that will become clear later, in order to implement the algorithm efficiently, it will be helpful to begin by presorting the points, both with respect to their $x$- and $y$-coordinates. Let $P_x$ be an array of points sorted by $x$, and let $P_y$ be an array of points sorted by $y$. We can compute these sorted arrays in $O(n \log n)$ time. Note that this initial sorting is done only once. In particular, the recursive calls do not repeat the sorting process.

Like any divide-and-conquer algorithm, after the initial basis case, our approach involves three basic elements: divide, conquer, and combine.

**Basis:** If $|P| \leq 3$, then just solve the problem by brute force in $O(1)$ time.

**Divide:** Otherwise, partition the points into two subarrays $P_L$ and $P_R$ based on their $x$-coordinates. In particular, imagine a vertical line $\ell$ that splits the points roughly in half (see Fig. 106). Let $P_L$ be the points to the left of $\ell$ and $P_R$ be the points to the right of $\ell$.

$^{16}$While the above example shows that sorting along any one coordinate axis may fail, there is a variant of this strategy that can be used for computing nearest neighbors approximately. This approach is based on the observation that if two points are close together, their projections onto a randomly oriented vector will be close, and if they are far apart, their projections onto a randomly oriented vector will be far apart in expectation. This observation underlies a popular nearest neighbor algorithm called **locality sensitive hashing**.
In the same way that we represented $P$ using two sorted arrays, we do the same for $P_L$ and $P_R$. Since we have presorted $P_x$ by $x$-coordinates, we can determine the median element for $\ell$ in constant time. After this, we can partition each of arrays $P_x$ and $P_y$ in $O(n)$ time each.

![Fig. 106: Divide-and-conquer closest pair algorithm.]

**Conquer:** Compute the closest pair within each of the subsets $P_L$ and $P_R$ each, by invoking the algorithm recursively. Let $\delta_L$ and $\delta_R$ be the closest pair distances in each case (see Fig. 106). Let $\delta = \min(\delta_L, \delta_R)$.

**Combine:** Note that $\delta$ is not necessarily the final answer, because there may be two points that are very close to one another but are on opposite sides of $\ell$. To complete the algorithm, we want to determine the closest pair of points between the sets, that is, the closest points $p \in P_L$ and $q \in P_R$ (see Fig. 106). Since we already have an upper bound $\delta$ on the closest pair, it suffices to solve the following restricted problem: if the closest pair $(p, q)$ are within distance $\delta$, then we will return such a pair, otherwise, we may return any pair. (This restriction is very important to the algorithm’s efficiency.) In the next section, we’ll show how to solve this restricted problem in $O(n)$ time. Given the closest such pair $(p, q)$, let $\delta' = \|pq\|$. We return $\min(\delta, \delta')$ as the final result.

Assuming that we can solve the “Combine” step in $O(n)$ time, it will follow that the algorithm’s running time is given by the recurrence $T(n) = 2T(n/2) + n$, and (as in Mergesort) the overall running time is $O(n \log n)$, as desired.

**Closest Pair Between the Sets:** To finish up the algorithm, we need to compute the closest pair $p$ and $q$, where $p \in P_L$ and $q \in P_R$. As mentioned above, because we already know of the existence of two points within distance $\delta$ of each other, this algorithm is allowed to fail, if there is no such pair that is closer than $\delta$. The input to our algorithm consists of the point set $P$, the $x$-coordinate of the vertical splitting line $\ell$, and the value of $\delta = \min(\delta_L, \delta_R)$. Recall that our goal is to do this in $O(n)$ time.

This is where the real creativity of the algorithm enters. Observe that if such a pair of points exists, we may assume that both points lie within distance $\delta$ of $\ell$, for otherwise the resulting
distance would exceed $\delta$. Let $S$ denote this subset of $P$ that lies within a vertical strip of width $2\delta$ centered about $\ell$ (see Fig. 107(a)).

$$\begin{align*}
S & \subset P \\
& \text{lies within a vertical strip of width } 2\delta \text{ centered about } \ell 
\end{align*}$$

How do we find the closest pair within $S$? Sorting comes to our rescue. Let $S_y = \langle s_1, \ldots, s_m \rangle$ denote the points of $S$ sorted by their $y$-coordinates (see Fig. 107(a)). At the start of the lecture, we asserted that considering the points that are close according to their $x$- or $y$-coordinate alone is not sufficient. It is rather surprising, therefore, that this does work for the set $S_y$.

The key observation is that if $S_y$ contains two points that are within distance $\delta$ of each other, these two points must be within a constant number of positions of each other in the sorted array $S_y$. The following lemma formalizes this observation.

**Lemma:** Given any two points $s_i, s_j \in S_y$, if $\|s_i, s_j\| \leq \delta$, then $|j - i| \leq 7$.

**Proof:** Suppose that $\|s_i, s_j\| \leq \delta$. Since they are in $S$ they are each within distance $\delta$ of $\ell$. Clearly, the $y$-coordinates of these two points can differ by at most $\delta$. So they must both reside in a rectangle of width $2\delta$ and height $\delta$ centered about $\ell$ (see Fig. 107(b)). Split this rectangle into eight identical squares each of side length $\delta/2$. A square of side length $x$ has a diagonal of length $x \sqrt{2}$, and no two points within such a square can be farther away than this. Therefore, the distance between any two points lying within one of these eight squares is at most $\delta \sqrt{2}/2 = \delta < \sqrt{2}$. Since each square lies entirely on one side of $\ell$, no square can contain two or more points of $P$, since otherwise, these two points would contradict the fact that $\delta$ is the closest

You might be tempted to think that we have pruned away many of the points of $P$, and this is the source of efficiency, but this is not generally true. It might very well be that every point of $P$ lies within the strip, and so we cannot afford to apply a brute-force solution to our problem.
pair seen so far. Thus, there can be at most eight points of \( S \) in this rectangle, one for each square. Therefore, \( |j - i| \leq 7 \).

**Avoiding Repeated Sorting:** One issue that we have not yet addressed is how to compute \( S_y \). Recall that we cannot afford to sort these points explicitly, because we may have \( n \) points in \( S \), and this part of the algorithm needs to run in \( O(n) \) time. This is where presorting comes in. Recall that the points of \( P_y \) are already sorted by \( y \)-coordinates. To compute \( S_y \), we enumerate the points of \( P_y \), and each time we find a point that lies within the strip, we copy it to the next position of array \( S_y \). This runs in \( O(n) \) time, and preserves the \( y \)-ordering of the points.

By the way, it is natural to wonder whether the value “8” in the statement of the lemma is optimal. Getting the best possible value is likely to be a tricky geometric exercise. Our textbook proves a weaker bound of “16”. Of course, from the perspective of asymptotic complexity, the exact constant does not matter.

The final algorithm is presented in the code fragment below.
closestPair(P = (Px, Py)) {
    n = |P|
    if (n <= 3) solve by brute force // basis case
    else {
        Find the vertical line L through P’s median // divide
        Split P into PL and PR (split Px and Py as well)
        dL = closestPair(PL) // conquer
        dR = closestPair(PR)
        d = min(dL, dR)
        for (i = 1 to n) { // create Sy
            if (Py[i] is within distance d of L) {
                append Py[i] to Sy
            }
        }
        d’ = stripClosest(Sy) // closest in strip
        return min(d, d’) // overall closest
    }
}
stripClosest(Sy) { // closest in strip
    m = |Sy|
    d’ = infinity
    for (i = 1 to m) {
        for (j = i+1 to min(m, i+7)) { // search neighbors
            if (dist(Sy[i], Sy[j]) <= d’) {
                d’ = dist(Sy[i], Sy[j]) // new closest found
            }
        }
    }
    return d’
}