Closest Pair Problem

Given a set of n points in 2D plane, $P = \{p_1, p_2, \cdots, p_n\}$, where each point p_i is represented as its x-coordinate $p_i.x$ and y-coordinate $p_i.y$, the *closest pair* problem seeks a pair of points $p_i, p_j \in P$ such that their distance, defined as $dist(p_i, p_j) = \sqrt{(p_i.x - p_j.x)^2 + (p_i.y - p_j.y)^2}$, is minimized.

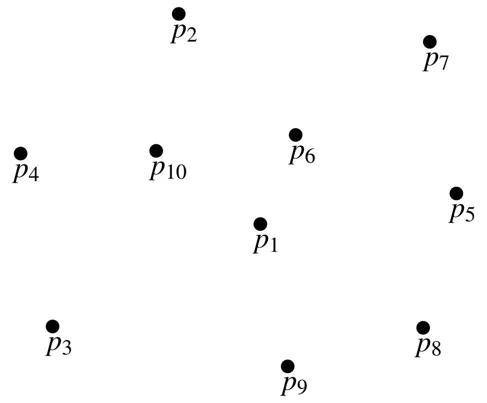

Figure 1: A set of 2D points $P = \{p_1, p_2, \cdots, p_9, p_{10}\}$. The closest pair is (p_1, p_6).

A brute-force algorithm enumerates all pairs, compute their distances, and find the smallest one:

```
Algorithm #1 ($P[1 \cdots n]$)
    use $d^*$ to store the current minimum distance, initialized as a large number $M$
    use $(p^*, q^*)$ to store the corresponding optimal pair, initialized as NULL
    for $i = 1$ to $n$
        for $j = i + 1$ to $n$
            $d = dist(P[i], P[j])$;
            if $d < d^*$: $d^* = d$, $p^* = P[i]$, $q^* = P[j]$;
        end
    end
end algorithm;
```

The above algorithm certainly takes $\Theta(n^2)$ time. We aim for designing a faster, $\Theta(n \cdot \log n)$ algorithm. To achieve this, we certainly need to avoid the above all-vs-all comparisons. How? Let's consider the 1D closest pair problem, i.e., the given n points $P = \{p_1, p_2, \cdots, p_n\}$ all locate on the x-axis (now each point is represented by a single coordinate rather than two coordinates). See the example below. To find the closest pair, we can sort all points from left to right. Let $P' = (p'_1, p'_2, \cdots, p'_n)$ be the sorted list of these 1D points. It is obvious that the closest pair of P must be adjacent in P', i.e., the closest pair must be (p'_i, p'_{i+1}) for some i. Hence, we can compute $dist(p'_i, p'_{i+1})$ for all $1 \le i \le n-1$ and pick the minimum. The whole process takes $\Theta(n \log n)$ time as the sorting step takes $\Theta(n \log n)$ time and the following step takes $\Theta(n)$ time.

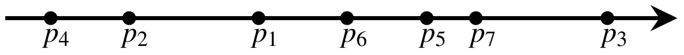

Figure 2: A set of 1D points $P = \{p_1, p_2, \cdots, p_7\}$. After sorting we get $P' = (p_4, p_2, p_1, p_6, p_5, p_7, p_3)$, and the closest pair must be next to each other in P'; in this case it is (p_5, p_7).

For 1D closest pair problem, after sorting one point p_i only needs to be compared with one other point (i.e., its successor in the sorted list P'). We will try to borrow these ideas to the 2D case. That is, we will try to sort the points (in some way), and we hope to show that it suffices to compare one point to constant number of other points. Such extension to 2D case is by no means obvious though.

We now design a divide-and-conquer algorithm, aiming for achieving a time complexity of $\Theta(n \log n)$. In a preprocessing step, we sort all points in P according to their x-coordinates in ascending order, obtaining a sorted list denoted as P_X. We will also sort all points according to their y-coordinates in ascending order, obtaining a sorted list denoted as P_Y. For example in Figure 1, $P_X = (p_4, p_3, p_{10}, p_2, p_1, p_9, p_6, p_8, p_7, p_5)$, and $P_Y = (p_9, p_8, p_3, p_1, p_5, p_4, p_{10}, p_6, p_7, p_2)$. You will see later, that we need both sorted lists.

Since we design a divide-and-conquer algorithm, we need to define the recursive function. We define DC-closest-pair (P_X, P_Y), where P_X and P_Y are the sorted lists of the same set of points P w.r.t. x- and y-coordinates, returns the closest pair (p^*, q^*) in P. We follow the framework of divide-and-conquer, that partitions the input into smaller subproblems. Naturally, we can partition the points in P into two halves, L and R, based on their x-coordinates. For example in Figure 1, we will have $L = \{p_1, p_2, p_3, p_4, p_{10}\}$ and $R = \{p_5, p_6, p_7, p_8, p_9\}$. Note that, in order to recursively call DC-closest-pair on L and R, we have to create the corresponding sorted lists for both L and R. That is, for the left partition L, we have to create L_X and L_Y, the sorted lists of L w.r.t. x- and y-coordinates, and for R we need to create R_X and R_Y. In the algorithm we will create lists L_X, L_Y, R_X, R_Y; we will not explicitly create the two halves L and R.

Since P_X is given, and L and R are partitioned based on x-coordinates, the construction and L_X and R_X is easy: L_X is simply the first half of P_X, and R_X is the remaining half, i.e., $L_X = P_X[1, 2, \cdots, n/2]$, and $R_X = P_X[n/2+1, \cdots, n]$. The construction of L_Y (resp. R_Y) can be done by sorting the points in L_X (resp. R_X), but this will take $\Theta(n \cdot \log n)$ time, which will make the final algorithm run in $\Theta(n \cdot \log^2 n)$ time. In fact, the construction of L_Y and R_Y can be done in $\Theta(n)$ time, by making use of the given P_Y, which sorts all points (i.e., P) by their y-coordinates. We can tranverse P_Y, and for each point p, put it to the end of L_Y if p locates in L, and put it to the end of R_Y if p locates in R. Since we keep the order of P_Y, certainly the resulting L_Y and R_Y are sorted by y-coordinates. The remaining question is how to determine p locates in which side. This can be determined by comparing the x-cooridnate of p, i.e., $p.x$, with the x-coordinate of the rightmost point in L, i.e., $L_X[n/2].x$. If $p.x \leq L_X[n/2].x$, then p is in L and otherwise p is in R. For the example in Figure 1, $L_X = (p_4, p_3, p_{10}, p_2, p_1)$, $R_X = (p_9, p_6, p_8, p_7, p_5)$, $L_Y = (p_3, p_1, p_4, p_{10}, p_2)$, and $R_Y = (p_9, p_8, p_5, p_6, p_7)$.

Now we recursively call DC-closest-pair (L_X, L_Y) and DC-closest-pair (R_X, R_Y), which return the closest pair in L and the closest pair in R. We denote them (p_L^*, q_L^*) and (p_R^*, q_R^*), respectively. We calculate the distances between them, and calculate the smaller of the two: $d^* := \min\{dist(p_L^*, q_L^*), dist(p_R^*, q_R^*)\}$. Clearly, d^* is the closest pair we find so far. But it may not the closest pair in P, since we have not considered pairs that span the two sides. We cannot afford to compare all pairs between L and R, as it takes $\Theta(n^2)$ time. We need a more efficient way to do it. The critical information is that we already have a pair with distance d^*, and therefore we only need to determine if there exists any pair (spanning the two sides) that is closer than d^*. Although this argument is straightforward, it is actually very powerful to avoid $\Theta(n^2)$ of comparisons.

Following above argument, we only need to consider points within a band centered at the partition line, i.e., $x = L_X[2/n].x$. See Figure 3. The width of this band is $2 \cdot d^*$. Any point outside this band must have a distance larger than d^* with any point on the other side. Hence, if a pair spanning L and R have a distance smaller than d^*, then both points must be in the band.

So we narrowed down the points to those in the band. But we still cannot afford comparing all pairs in it. Fortunately, we do not have to. In fact, for each point in the band, we only *need* to compare it to a

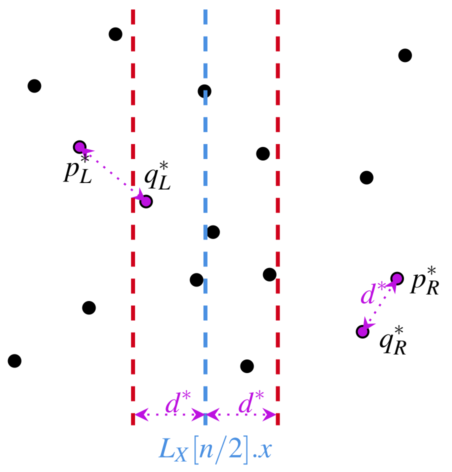

Figure 3: Illustration of the band centered at line $x = L_X[n/2].x$ with width $2 \cdot d^*$.

constant number of points. This boils down the total number of comparisons to $\Theta(n)$. Let's establish this key result. First, we need to create a sorted list for points in the band, in ascending order of y-coordinates. We denote this sorted list as B_Y. The construction of B_Y, again, can be done by using the given sorted list P_Y. Specifically, we tranverse P_Y, and for each point p in it, determine if it is in the band, by testing if $|p.x - L_X[n/2].x| \leq d^*$, and if it is so, we put p to the end of B_Y. The lemma given below states the key conclusion: if two points are 12 positions or more apart in the sorted list B_Y, then their distance is guaranteed to be larger than d^*. With this lemma, in order to determine possible pair with distance smaller than d^* in the band, we just need to examine pairs $(B_Y[i], B_Y[j])$, where $j \leq i + 12$. The number of such pairs is certainly at most $12 \cdot |B_Y|$, which is $\Theta(n)$.

Lemma. If $j \geq i + 12$, then $dist(B_Y[i], B_Y[j]) \geq d^*$.

We now prove above lemma. We partition the band into squares of size $d^*/2 \times d^*/2$. See Figure 4. We first prove that, there is at most 1 point in each box (including its boundary). Suppose conversely that there are two points, p and q, in a box. Then the distance between them $dist(p, q) \leq d^*/2 \cdot \sqrt{2} < d^*$. This is a contradiction, since each box is entirely inside either left side or the right side, and we know that the closest

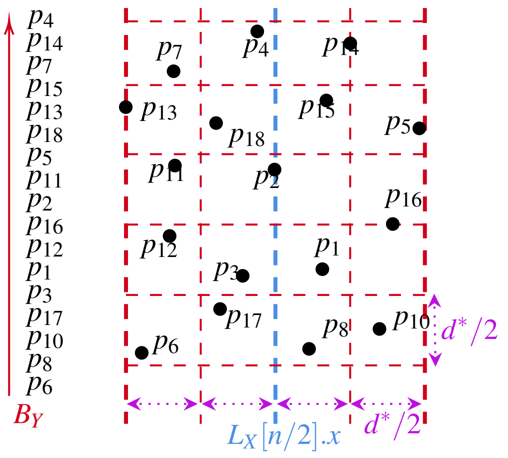

Figure 4: Illustration the proof of above lemma.

pair not spanning two sides is d^*. Next, if $j \geq i+12$, then $B_Y[j]$ is at least three layers of boxes above $B_Y[i]$. This is because, each layer contains 4 boxes, and therefore at most 4 points can be in them. Hence if $B_Y[i]$ is in layer-l, then $B_Y[j]$ must be in layer-$(l+3)$ or higher. The height of each layer is $d^*/2$. The vertical distance between $B_Y[i]$ and $B_Y[j]$ is hence larger than d^*; therefore, their distance $dist(B_Y[i], B_Y[j]) > d^*$.

The complete algorithm is given below in pseudo-code.

```
Algorithm #2 ($P[1 \cdots n]$)
    sort $P$ according to $x$-coordinates to obtain $P_X$;
    sort $P$ according to $y$-coordinates to obtain $P_Y$;
    return DC-closest-pair ($P_X$, $P_Y$);
end algorithm;

function DC-closest-pair ($P_X$, $P_Y$)
    if $|P_X| \leq 3$: enumerate all pairs to find the closest pair and return it;
    $L_X = P_X[1 \cdots n/2]$, $R_X = P_X[n/2+1 \cdots n]$;
    init $L_Y$ and $R_Y$ as empty lists;
    let $x^* = L_X[n/2].x$;
    for $i = 1$ to $|P_Y|$
        if $P_Y[i].x \leq x^*$: add $P_Y[i]$ to the end of $L_Y$;
        else: add $P_Y[i]$ to the end of $R_Y$;
    end
    $(p_L^*, q_L^*)$ = DC-closest-pair $(L_X, L_Y)$;
    $(p_R^*, q_R^*)$ = DC-closest-pair $(R_X, R_Y)$;
    $d^* = \min\{dist(p_L^*, q_L^*), dist(p_R^*, q_R^*)\}$;
    $(p^*, q^*) = \arg\min\{dist(p_L^*, q_L^*), dist(p_R^*, q_R^*)\}$, to store the closest pair found from two halves;
    init $B_Y$ as an empty list;
    for $i = 1$ to $|P_Y|$
        if $|P_Y[i].x - x^*| \leq d^*$: add $P_Y[i]$ to the end of $B_Y$;
    end
    for $i = 1$ to $|B_Y|$
        for $j = i+1$ to $i+12$
            $d = dist(B_Y[i], B_Y[j])$;
            if $d < d^*$: $d^* = d$, $p^* = B_X[i]$, $q^* = B_Y[j]$;
        end
    end
    return $(p^*, q^*)$;
end algorithm;
```

Let's analyze the time complexity. Let $T(n)$ be running time of DC-closest-pair (P_X, P_Y) when $|P_X| = |P_Y| = n$. In DC-closest-pair function, the two recursive calls takes $2 \cdot T(n/2)$ time; the rest takes $\Theta(n)$ time. Therefore, the recurrence is $T(n) = 2 \cdot T(n/2) + \Theta(n)$ which gives $T(n) = \Theta(n \cdot \log n)$. The entire algorithm #2 is therefore also takes $\Theta(n \cdot \log n)$ time since the two sorts takes $\Theta(n \cdot \log n)$ as well.