CS31 (Algorithms), Spring 2020 : Lecture 4 Supplement

Date:

Topic: Divide and Conquer

Disclaimer: These notes have not gone through scrutiny and in all probability contain errors. Please discuss in Piazza/email errors to deeparnab@dartmouth.edu

---

This is an "advanced" application of divide and conquer. This is present in the Echo 360 video, but we did not go over this in the classroom session. Read it at your own leisure

1 Closest Pair of Points on the Plane

We look at a simple geometric problem: given n points on a plane, find the pair which is closest to each other. More precisely, the n points are described as their (x, y) coordinates; point p_i will have coordinates (x_i, y_i). The distance between two points p_i and p_j is defined as

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

One could also look at other distances such as $d(p_i, p_j) = \max(|x_i - x_j|, |y_i - y_j|)$ and $d(p_i, p_j) = |x_i - x_j| + |y_i - y_j|$. What we describe below works for both these as well.

CLOSEST PAIR OF POINTS ON THE PLANE
Input: n points $P = \{p_1, \ldots, p_n\}$ where $p_i = (x_i, y_i)$.
Output: The pair p_i, p_j with smallest $d(p_i, p_j)$.
Size: The number of points, n.

Once again, as many of the examples before, there is a trivial $O(n^2)$ time algorithm: simply try all pairs and return the closest pair. This is the naive benchmark which we will try to beat using Divide-and-Conquer.

How should we divide this set of points into two halves? To do so, let us think whether there is a natural ordering of these points? A moment's thought leads us to two natural orderings: one sorted using their x-coordinates, and one using their y-coordinates. Let us use $P_x[1:n]$ to denote the permutation of the n points such that[1] $\mathsf{xcoor}(P_x[i]) < \mathsf{xcoor}(P_x[j])$ for $i < j$. Similarly we define $P_y[1:n]$. Getting these permutations from the input takes $O(n \log n)$ time.

Before moving further, we point out something which we will use later. Let $S \subseteq P$ be an arbitrary set of points of size s. Suppose we want the arrays $S_x[1:s]$ and $S_y[1:s]$ which are permutations of S ordered according to their xcoor's and ycoor's, respectively. If S is given as a "bit-array" with a 1 in position i if point $p_i \in S$, then to obtain S_x and S_y we don't need to sort again, but can obtain these from P_x and P_y. This is obtained by "masking" S with P_x; we traverse P_x from left-to-right and pick the point $p = P_x[i]$ if and only if $S[p]$ evaluates to 1. Note this is a $O(n)$ time procedure. This "dynamic sorting" was something we encountered in the Counting Inversions problem and is an useful thing to know. For more details, see UGP2, Problem 2. Let us now get back to our problem.

Given P_x, we can divide the set of points P into two halves as follows. Let $m = \lfloor n/2 \rfloor$ and $x^* := \mathsf{xcoor}(P_x[m])$ be the median of P_x. Define $Q_x := P_x[1:m]$ and $R_x := P_x[m+1:n]$, and let us use Q and R to denote the set of these point. Figure 1 illustrates this.

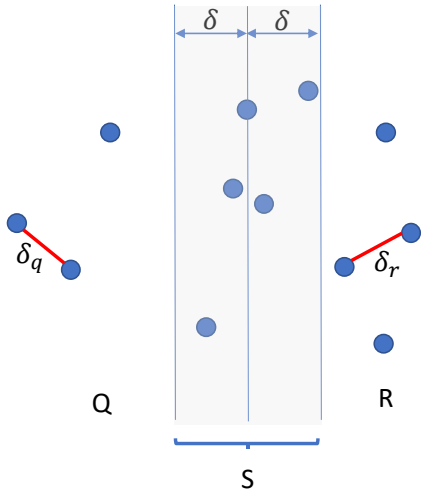

Figure 1: Closest pair in a plane

We recursively call the algorithm on the sets Q and R. Let (q_i, q_j) and (r_i, r_j) be the pairs returned. We will use[2] $\delta_q := d(q_i, q_j)$ and $\delta_r := d(r_i, r_j)$. Clearly these are candidate points for closest pair of points among P.

The other candidate pairs of P are precisely the *cross pairs*: (q_i, r_j) for $q_i \in Q$ and $r_j \in R$. Therefore, to conquer we need to find the nearest cross pair. Can we do this in time much better than $O(n^2)$? If you think for a little bit, this doesn't seem any easier at all – can we still get a win? Indeed we will, but we need to exploit the **geometry** of the problem. And this will form the bulk of the remainder of this lecture.

First let us note that we don't need to consider all pairs in $Q \times R$. Define $\delta := \min(\delta_q, \delta_r)$. Since we are looking for the closest pair of points, we don't need to look at cross-pairs which are more than δ apart.

Claim 1. Consider any point $q_i \in Q$ with $\mathsf{xcoor}(q_i) < x^* - \delta$. We don't need to consider any (q_i, r_j) point for $r_j \in R$ as a candidate. Similarly, for any point $r_j \in R$ with $\mathsf{xcoor}(r_j) > x^* + \delta$, we don't need to consider any (q_i, r_j) point for $q_i \in Q$ as a candidate.

Proof. Any candidate (q_i, r_j) we need to consider better have $d(q_i, r_j) \leq \delta$. But

$$d(q_i, r_j) \geq |\mathsf{xcoor}(q_i) - \mathsf{xcoor}(r_j)|$$

Therefore, if $\mathsf{xcoor}(q_i) < x^* - \delta$, and since $\mathsf{xcoor}(r_j) \geq x^*$ for all $r_j \in R$, we get $|\mathsf{xcoor}(q_i) - \mathsf{xcoor}(r_j)| > \delta$. Thus, we can rule out (q_i, r_j) for all $r_j \in R$. The other statement follows analogously. □

Motivated by the above, let us define $Q' := \{q_i \in Q : \mathsf{xcoor}(q_i) \geq x^* - \delta\}$ and $R' := \{r_j \in R : \mathsf{xcoor}(r_j) \leq x^* + \delta\}$. That is $S := Q' \cup R'$ lies in the band illustrated in Figure 1. To summarize, we only need to look for cross-pairs[3] in $S \times S$.

[1] Just for simplicity we assume no two points share xcoor or ycoor coordinates. Not really necessary, but let's assume anyway.

[2] We haven't discussed the base case: if $n = 2$, then we return that pair; if $n = 1$, then we actually return $\perp$ and the corresponding $\delta = \infty$.

[3] Actually, we can restrict to $Q' \times R'$, but searching more widely doesn't hurt and makes exposition easier.

Have we made progress? Note that all of Q could be sitting in Q' and all of R could be sitting in R', and it may feel we haven't moved much. But note, if that is the case, then all points are in a "narrow band". We will soon see why that is important.

Let us start with a "naive" way of going over all cross-pairs in $S \times S$. Start with a point $q \in S$. Go over all *other* points $r \in S$ evaluating $d(q,r)$ as we go and store the minimum. Then repeat this for all $q \in S$ and take the smallest of all these minimums. Again, to make sure we are on the same page, given that in the worst case $S = P$, as stated this naive algorithm is still $O(n^2)$.

Once again, we want to use the observation that pairs which are $> \delta$ far needn't be considered. In particular, if the *y-coordinates* of two points are more than δ, we don't need to consider that pair. So, for any fixed $q \in S$, we could restrict our search only on the points $r \in S$ with $|\mathsf{ycoor}(r) - \mathsf{ycoor}(q)| \leq \delta$. We can do this restriction easily using the *sorted array* S_y.

To formalize this, first note that, as mentioned before, we can use P_y (the sorted array of the original points) to find the array S_y which is the points in S sorted according to the ycoor's. To find the closest cross-pair, we consider the points in the increasing ycoor order; for a point $q \in S$ we look at the other points $r \in S$ subsequent to it in S_y having $\mathsf{ycoor}(r) \leq \mathsf{ycoor}(q) + \delta$, store the distances $d(q,r)$, and return the minimum. The following piece of pseudocode formalizes this.

```
1:  procedure ClosestCrossPairs(S, δ):
2:      ▷ Returns cross pair (q, r) ∈ S × S with d(q, r) < δ and smallest among them.
3:      ▷ If no d(q, r) < δ, then returns ⊥.
4:      Use P_y to compute S_y i.e. S sorted according to ycoor.  ▷ Can be done in O(n) time.
5:      t ← ⊥  ▷ t is a tuple which will contain the closest cross pair
6:      dmin ← δ  ▷ dmin is the current min init to δ
7:      for 1 ≤ i ≤ |S| do:
8:          p_cur ← S_y[i].
9:          ▷ Next, check if there is a point q_cur such that its distance to p_cur is < dmin.
10:         ▷ If so, then we define this pair to be t and define this distance to be the new dmin.
11:         ▷ Crucially, we don't need to check points which are δ away in the y-coordinate.
12:         j ← 1; q_cur ← S_y[i + j].
13:         while ycoor(q_cur) < ycoor(p_cur) + δ do:
14:             if d(p_cur, q_cur) < dmin then: ▷ Modify dmin and t.
15:                 dmin ← d(p_cur, q_cur);
16:                 t ← (p_cur, q_cur)
17:             j ← j + 1; q_cur ← S_y[i + j].  ▷ Move to the next point in S_y.
18:     return t  ▷ Could be ⊥ as well.
```

Remark: *One may wonder that we are not returning cross-pairs as we could return q, r both in Q'. However, for any pair (q,r) returned, we have $d(q,r) < \delta$; since $\delta = \min(\delta_q, \delta_r)$, this pair can't lie on the same side.*

Armed with the above "conquering" step, we can state the full algorithm.

```
1:  procedure ClosestPair(P):
2:      ▷ We assume n = |P|.
3:      ▷ We assume arrays P_x[1 : n] and P_y[1 : n] which are xcoor and ycoor-sorted P.
4:      if n ∈ {1, 2} then:
5:          If n = 1 return ⊥; else return P.
6:      m ← ⌊n/2⌋
7:      Q be the points in P_x[1 : m]
8:      R be the points in P_x[m + 1 : n]
9:      (q1, q2) ← ClosestPair(Q); δ_q ← d(q1, q2).
10:     (r1, r2) ← ClosestPair(R); δ_r ← d(r1, r2).
11:     δ ← min(δ_q, δ_r)
12:     x* ← xcoor(P_x[m]).
13:     Compute S ← {p_i : x* − δ ≤ xcoor(p_i) ≤ x* + δ}. ▷ Store as indicator bit-array
14:     ▷ All cross-pairs worthy of consideration lie in S
15:     (s1, s2) ← ClosestCrossPair(S)
16:     return Best of (q1, q2), (r1, r2) and (s1, s2).
```

How long does the above algorithm take? It really depends on how long ClosestCrossPair(S) takes. We now focus on the running time of this algorithm.

Note $|S|$ could be as large as $\Theta(n)$. The inner while loop, a priori, can take $O(|S|)$ time, and thus along with the for-loop, the above seems to take $O(n^2)$ time. Doesn't seem we have gained anything. Next comes the real geometric help.

Remark: *In the echo 360 lecture videos, we have a much stronger lemma than before with 72 replaced by 8. Still, I think the arguments below has a certain generality which is good to know.*

Lemma 1. Fix any point $q \in S$. Then there are at most 72 points $r \in S$ with $d(q, r) \leq \sqrt{5}\delta$.

Before we prove this, let us first see why is this lemma useful.

Corollary 1. The inner while loop always takes $O(1)$ time.

Proof. Suppose not, that is, the while loop runs for > 72 iterations for some $q = S_y[i]$. Then, there are at least 72 points $r \in S$ s.t. $\mathsf{ycoor}(r) \leq \mathsf{ycoor}(q) + \delta$, or $|\mathsf{ycoor}(r) - \mathsf{ycoor}(q)| \leq \delta$. Since $q, r \in S$, we know that $|\mathsf{xcoor}(r) - \mathsf{xcoor}(q)| \leq 2\delta$. This means that $d(q, r) \leq \sqrt{5}\delta$. But this contradicts Lemma 1. □

Proof of Lemma 1. Before going over the math, let's see the intuition. Suppose there are > 72 points of S in a circle of radius $\sqrt{5}\delta$ around a point q. Now at least 36 of these points belong to one set Q or R; let's without loss of generality this is R. What do we know about these 36 points – their pairwise distances are $\geq \delta$. How can we have so many points (if 36 doesn't sound a lot, thing 36000) which are each δ-far from each other, all sitting in a circle of radius $\sqrt{5}\delta$? We can't : try to picture it. You will see lot of congestion.

Now we do the math. Here is the formal argument. Let's take these 36 points of R and draw circles of radius $\delta/2$ around them. Since any two pair of points is $\geq \delta$, all these circles are non-overlapping.

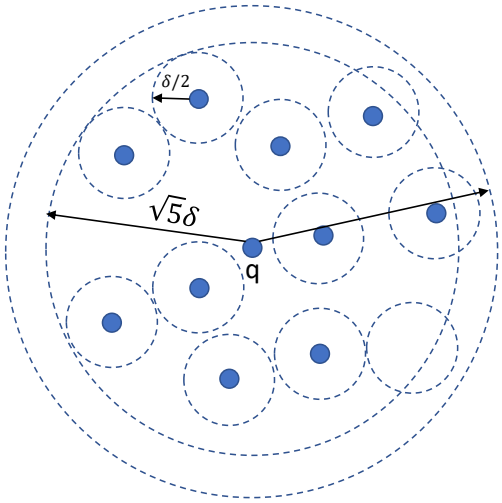

Figure 2: Small Non-overlapping circles inside another circle. Can't be many.

Furthermore, all these 36 circles lie in the bigger circle of radius $(\sqrt{5}+1/2)\delta$ around q. See Figure 2 for an illustration.

We get a contradiction by an "area" argument. The area of the big circle is $\pi \cdot \delta^2 \cdot (\sqrt{5}+1/2)^2 < 9\pi\delta^2$. The area of each small circle is $\pi \cdot \delta^2/4$. Since the 36 small circles all fit in the big circle and they are *non-overlapping*, the sum of the areas of the small circles must be ≤ the area of the big circle. This is where we reach a contradiction – the 36 small circles have area $9\pi\delta^2$. □

If $T(n)$ is the worst case running time of CLOSESTPAIR when run on point set of n points, we get the recurrence inequality which I hope we all have learned to love:

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n)$$

This evaluates to $T(n) = O(n \log n)$.

Theorem 1. The closest pair of points among n points in a plane can be found by CLOSESTPAIR in $O(n \log n)$ time.