Finding the nearest pair of points

Table of Contents

Problem statement

Given n points on the plane. Each point p_i is defined by its coordinates (x_i, y_i). It is required to find among them two such points, such that the distance between them is minimal:

$$\min_{\substack{i,j=0 \ldots n-1,\\ i \neq j}} \rho(p_i, p_j).$$

We take the usual Euclidean distances:

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

The trivial algorithm - iterating over all pairs and calculating the distance for each — works in $O(n^2)$.

The algorithm running in time $O(n \log n)$ is described below. This algorithm was proposed by Preparata in 1975. Preparata and Shamos also showed that this algorithm is optimal in the decision tree model.

Algorithm

We construct an algorithm according to the general scheme of **divide-and-conquer** algorithms: the algorithm is designed as a recursive function, to which we pass a set of points; this recursive function splits this set in half, calls itself recursively on each half, and then performs some operations to combine the answers. The operation of combining consist of detecting the cases when one point of the optimal solution fell into one half, and the other point into the other (in this case, recursive calls from each of the halves cannot detect this pair separately). The main difficulty, as always in case of divide and conquer algorithms, lies in the effective implementation of the merging stage. If a set of n points is passed to the recursive function, then the merge stage should work no more than $O(n)$, then the asymptotics of the whole algorithm $T(n)$ will be found from the equation:

$$T(n) = 2T(n/2) + O(n).$$

The solution to this equation, as is known, is $T(n) = O(n \log n)$.

So, we proceed on to the construction of the algorithm. In order to come to an effective implementation of the merge stage in the future, we will divide the set of points into two subsets, according to their x-coordinates: In fact, we draw some vertical line dividing the set of points into two subsets of approximately the same size. Is is convenient to make such a partition as follows: We sort the points in the standard way as pairs of numbers, ie.:

$$p_i < p_j \Longleftrightarrow (x_i < x_j) \lor \Big((x_i = x_j) \land (y_i < y_j) \Big)$$

Then take the middle point after sorting $p_m (m = \lfloor n/2 \rfloor)$, and all the points before it and the p_m itself are assigned to the first half, and all the points after it - to the second half:

$$A_1 = p_i \mid i = 0 \ldots m$$

$$A_2 = p_i \mid i = m + 1 \ldots n - 1.$$

Now, calling recursively on each of the sets A_1 and A_2, we will find the answers h_1 and h_2 for each of the halves. And take the best of them: $h = \min(h_1, h_2)$.

Now we need to make a **merge stage**, i.e. we try to find such pairs of points, for which the distance between which is less than h and one point is lying in A_1 and the other in A_2. It is obvious that it is sufficient to consider only those points that are separated from the vertical line by a distance less than h, i.e. the set B of the points considered at this stage is equal to:

$$B = p_i \mid |x_i - x_m| < h.$$

For each point in the set B, we try to find the points that are closer to it than h. For example, it is sufficient to consider only those points whose y-coordinate differs by no more than h. Moreover, it makes no sense to consider those points whose y-coordinate is greater than the y-coordinate of the current point. Thus, for each point p_i we define the set of considered points $C(p_i)$ as follows:

$$C(p_i) = p_j \mid p_j \in B,\ \ y_i - h < y_j \le y_i.$$

If we sort the points of the set B by y-coordinate, it will be very easy to find $C(p_i)$: these are several points in a row ahead to the point p_i.

So, in the new notation, the **merging stage** looks like this: build a set B, sort the points in it by y-coordinate, then for each point $p_i \in B$ consider all points $p_j \in C(p_i)$, and for each pair (p_i, p_j) calculate the distance and compare with the current best distance.

At first glance, this is still a non-optimal algorithm: it seems that the sizes of sets $C(p_i)$ will be of order n, and the required asymptotics will not work. However, surprisingly, it can be proved that the size of each of the sets $C(p_i)$ is a quantity $O(1)$, i.e. it does not exceed some small constant regardless of the points themselves. Proof of this fact is given in the next section.

Finally, we pay attention to the sorting, which the above algorithm contains: first,sorting by pairs (x, y), and then second, sorting the elements of the set B by y. In fact, both of these sorts inside the recursive function can be eliminated (otherwise we would not reach the $O(n)$ estimate for the **merging stage**, and the general asymptotics of the algorithm would be $O(n \log^2 n)$). It is easy to get rid of the first sort — it is enough to perform this sort before starting the recursion: after all, the elements themselves do not change inside the recursion, so there is no need to sort again. With the second sorting a little more difficult to perform, performing it previously will not work. But, remembering the merge sort, which also works on the principle of divide-and-conquer, we can simply embed this sort in our recursion. Let recursion, taking some set of points (as we remember,ordered by pairs (x, y)), return the same set, but sorted by the y-coordinate. To do this, simply merge (in $O(n)$) the two results returned by recursive calls. This will result in a set sorted by y-coordinate.

Evaluation of the asymptotics

To show that the above algorithm is actually executed in $O(n \log n)$, we need to prove the following fact: $|C(p_i)| = O(1)$.

So, let us consider some point p_i; recall that the set $C(p_i)$ is a set of points whose y-coordinate lies in the segment $[y_i - h; y_i]$, and, moreover, along the x coordinate, the point p_i itself, and all the points of the set $C(p_i)$ lie in the band width $2h$. In other words, the points we are considering p_i and $C(p_i)$ lie in a rectangle of size $2h \times h$.

Our task is to estimate the maximum number of points that can lie in this rectangle $2h \times h$; thus, we estimate the maximum size of the set $C(p_i)$. At the same time, when evaluating, we must not forget that there may be repeated points.

Remember that h was obtained from the results of two recursive calls — on sets A_1 and A_2, and A_1 contains points to the left of the partition line and partially on it, A_2 contains the remaining points of the partition line and points to the right of it. For any pair of points from A_1, as well as from A_2, the distance can not be less than h — otherwise it would mean incorrect operation of the recursive function.

To estimate the maximum number of points in the rectangle $2h \times h$ we divide it into two squares $h \times h$, the first square include all points $C(p_i) \cap A_1$, and the second contains all the others, i.e. $C(p_i) \cap A_2$. It follows from the above considerations that in each of these squares the distance between any two points is at least h.

We show that there are at most four points in each square. For example, this can be done as follows: divide the square into 4 sub-squares with sides $h/2$. Then there can be no more than one point in each of these sub-squares (since even the diagonal is equal to $h/\sqrt{2}$, which is less than h). Therefore, there can be no more than 4 points in the whole square.

So, we have proved that in a rectangle $2h \times h$ can not be more than $4 \cdot 2 = 8$ points, and, therefore, the size of the set $C(p_i)$ cannot exceed 7, as required.

Implementation

We introduce a data structure to store a point (its coordinates and a number) and comparison operators required for two types of sorting:

```
struct pt {
    int x, y, id;
};

struct cmp_x {
    bool operator()(const pt & a, const pt & b
        return a.x < b.x || (a.x == b.x && a.y
    }
};

struct cmp_y {
    bool operator()(const pt & a, const pt & b
        return a.y < b.y;
    }
};

int n;
vector<pt> a;
```

For a convenient implementation of recursion, we introduce an auxiliary function upd_ans(), which will calculate the distance between two points and check whether it is better than the current answer:

```
double mindist;
pair<int, int> best_pair;

void upd_ans(const pt & a, const pt & b) {
    double dist = sqrt((a.x - b.x)*(a.x - b.x)
    if (dist < mindist) {
        mindist = dist;
        best_pair = {a.id, b.id};
    }
}
```

Finally, the implementation of the recursion itself. It is assumed that before calling it, the array $a[]$ is already sorted by x-coordinate. In recursion we pass just two pointers l, r, which indicate that it should look for the answer for $a[l \ldots r)$. If the distance between r and l is too small, the recursion must be stopped, and perform a trivial algorithm to find the nearest pair and then sort the subarray by y-coordinate.

To merge two sets of points received from recursive calls into one (ordered by y-coordinate), we use the standard STL $merge()$ function, and create an auxiliary buffer $t[]$(one for all recursive calls). (Using inplace_merge () is impractical because it generally does not work in linear time.)

Finally, the set B is stored in the same array t.

```
vector<pt> t;

void rec(int l, int r) {
    if (r - l <= 3) {
        for (int i = l; i < r; ++i) {
            for (int j = i + 1; j < r; ++j) {
                upd_ans(a[i], a[j]);
            }
        }
        sort(a.begin() + l, a.begin() + r, cmp
        return;
    }

    int m = (l + r) >> 1;
    int midx = a[m].x;
    rec(l, m);
    rec(m, r);

    merge(a.begin() + l, a.begin() + m, a.begi
    copy(t.begin(), t.begin() + r - l, a.begin

    int tsz = 0;
    for (int i = l; i < r; ++i) {
        if (abs(a[i].x - midx) < mindist) {
            for (int j = tsz - 1; j >= 0 && a[
                upd_ans(a[i], t[j]);
            t[tsz++] = a[i];
        }
    }
}
```

By the way, if all the coordinates are integer, then at the time of the recursion you can not move to fractional values, and store in $mindist$ the square of the minimum distance.

In the main program, recursion should be called as follows:

```
t.resize(n);
sort(a.begin(), a.end(), cmp_x());
mindist = 1E20;
rec(0, n);
```

Generalization: finding a triangle with minimal perimeter

The algorithm described above is interestingly generalized to this problem: among a given set of points, choose three different points so that the sum of pairwise distances between them is the smallest.

In fact, to solve this problem, the algorithm remains the same: we divide the field into two halves of the vertical line, call the solution recursively on both halves, choose the minimum $minper$ from the found perimeters, build a strip with the thickness of $minper/2$, and iterate through all triangles that can improve the answer. (Note that the triangle with perimeter $\leq minper$ has the longest side $\leq minper/2$.)

Practice problems

- UVA 10245 "The Closest Pair Problem" [difficulty: low]
- SPOJ #8725 CLOPPAIR "Closest Point Pair" [difficulty: low]
- CODEFORCES Team Olympiad Saratov - 2011 "Minimum amount" [difficulty: medium]
- Google CodeJam 2009 Final " Min Perimeter " [difficulty: medium]
- SPOJ #7029 CLOSEST "Closest Triple" [difficulty: medium]