COMPUTATIONAL GEOMETRY

Labworks

November 11, 2020

Deadline: December 2, 2020.

Consider the java classes

```
class Point {
    public double x,y;
}
```

which we use to represent point coordinates in the Cartesian plane, and

```
class Point3D {
    public double x,y,z;
}
```

which we use to represent point coordinates in the Cartesian space.

Programming assignment 1 (30 points)

Write a java program that finds the closest pair of points from a set of $n \geq 2$ points. The program should read from the standard input the following data:

> one line which contains the value of n, followed by
>
> each of the following n lines contains two floating-point numbers separated by whitespace. These numbers represent the (x, y) coordinates of a point.

Note. This algorithm was presented in Lecture 7. More explanations can be found in the Appendix of this labwork assignment.

Programming assignment 2 (20 points)

Extend class `Point` with the static method

```
public static Point intersect(Point A,Point B,Point C,Point D);
```

which checks if the segments `AB` and `CD` intersect, and returns

- `null` if $\texttt{AB} \cap \texttt{CD} = \emptyset$.
- otherwise, the point `P` which is at the intersection of `AB` with `CD`.

Programming assignment 3 (20 points)

Suppose we represent a segment in the Cartesian space by an instance of the class

```
class Segment {
   public Point3D A,B;    // the coordinates of the endpoints of the segment
}
```

We say that a segment s_1 is **above** another segment s_2 if there is a vertical line ℓ that intersects both segments, such that the $\ell \cap s_1$ is above $\ell \cap s_2$.

Extend class `Segment` with the static method

```
public static int above(Segment s1,Segment s2);
```

which returns

0 if `s1` and `s2` intersect,

1 if `s1` is above `s2`,

2 if `s2` is `above s1`, and

3 otherwise.

Programming assignment 4 (30 points)

Heron's formula tells us that the area of a triangle whose sides have lengths a, b, c is

$$\text{area}(ABC) = \sqrt{s(s-a)(s-b)(s-c)}$$

where $s = (a+b+c)/2$.

Suppose $P_1P_2 \dots P_n$ is a convex polygon, and we want to compute its area Extend class `Point` with the static methods

```
public static double area(vector<Point> P);
public static double perimeter(vector<Point> P);
```

which compute the area and perimeter of $P_1P_2 \dots P_n$ when `P` is the vector of coordinates of points $P_1, P_2, ..., P_n$ in clockwise order.

Appendix: Finding the closest pair of points

Consider the problem of finding the closest pair of points in a set Q of $n \geq 2$ points.

- **Closest** refers to the usual euclidean distance: the distance between points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.
- Several applications, e.g., in traffic control systems: identify two closest vehicles in order to detect potential collisions.

- The simple, brute-force closest-pair algorithm: look at all the $\binom{n}{2}$ pairs of points $\Rightarrow O(n^2)$ complexity.
- In this labwork, we consider a divide-and-conquer algorithm with running time $O(n \log n)$.

The divide-and-conquer algorithm

Each recursive call of the algorithm takes as input a subset $P \subseteq Q$ and arrays X and Y, each of which contains all the points of the input set P:

- ▶ X contains the elements of P sorted in increasing order of the x-coordinate
- ▶ Y contains the elements of P sorted in increasing order of the y-coordinate

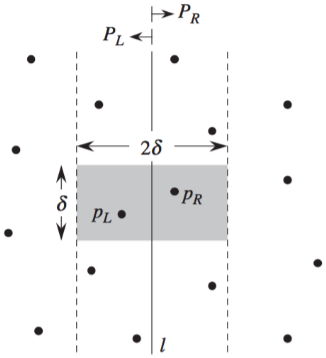
A given recursive invocation with inputs P, X, and Y first checks if $|P| \leq 3$. If so, find the closest points in P with the brute-force approach, by trying all pairs of points. If $|P| > 3$, the recursive invocation carries out the divide-and-conquer paradigm as follows.

Divide: Find a vertical line l that bisects the point set P into two sets P_L and P_R such that $|P_L| = \lceil |P|/2 \rceil$, $Q_L = \lfloor |P|/2 \rfloor$, all points in P_L are on or to the left of line l, and all points in P_R are on or to the right of l. Divide the array X into arrays X_L and X_R, which contain the points of P_L and P_R respectively, sorted by monotonically increasing x-coordinate. Similarly, divide the array Y into arrays Y_L and Y_R, which contain the points of P_L and P_R respectively, sorted by monotonically increasing y-coordinate.

Conquer: Having divided P into P_L and P_R, make two recursive calls, one to find the closest pair of points in P_L and the other to find the closest pair of points in P_R. The inputs to the first call are the subset P_L and arrays X_L and Y_L; the second call receives the inputs P_R, X_R, and Y_R. Let the closest-pair distances returned for P_L and P_R be δ_L and δ_R, respectively, and let $\delta = \min(\delta_L, \delta_R)$.

Combine: The closest pair is either the pair with distance δ found by one of the recursive calls, or it is a pair of points with one point in P_L and the other in P_R. The algorithm determines whether there is a pair with one point in P_L and the other point in P_R and whose distance is less than δ. Observe that if a pair of points has distance less than δ, both points of the pair must be within δ units of line l. Thus, they both must reside in the $2\,\delta$-wide vertical strip centered at line l. To find such a pair, if one exists, we do the following:

1. Create an array Y', which is the array Y with all points not in the $2\,\delta$-wide vertical strip removed. The array Y' is sorted by y-coordinate, just as Y is.

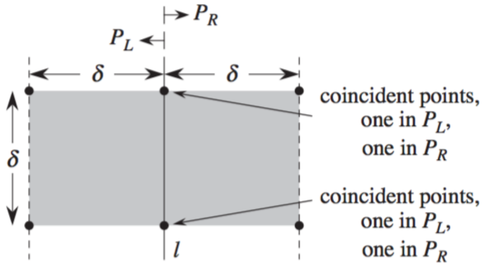


2. For each point p in the array Y', try to find points in Y' that are within δ units of p. As we shall see shortly, only the 7 points in Y' that follow p need be considered. Compute the distance from p to each of these 7 points, and keep track of the closest-pair distance δ' found over all pairs of points in Y'.
3. If $\delta' < \delta$, then the vertical strip does indeed contain a closer pair than the recursive calls found. Return this pair and its distance δ'. Otherwise, return the closest pair and its distance δ found by the recursive calls.

Why are seven points sufficient for lookup?

We shall prove that we need only check the seven points following each point p in array Y'.

Suppose that at some level of the recursion, the closest pair of points is $p_L \in P_L$ and $p_R \in P_R$. Thus, the distance δ' between p_L and p_R is strictly less than δ. Point p_L must be on or to the left of line l and less than δ units away. Similarly, p_R is on or to the right of l and less than δ units away. Moreover, p_L and p_R are within δ units of each other vertically. Thus, as Figure below shows, p_L and p_R are within a $\delta \times 2\delta$ rectangle centered at line l. (There may be other points within this rectangle as well.)



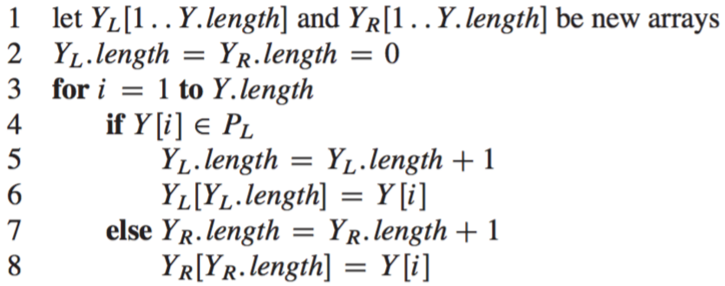
We next show that at most 8 points of P can reside within this $\delta \times 2\delta$ rectangle. Consider the $\delta \times \delta$ square forming the left half of this rectangle. Since all points within P_L are at least δ units apart, at most 4 points can reside within this square; The figure above shows how. Similarly, at most 4 points in P_R can reside within the $\delta \times \delta$ square forming the right half of the rectangle. Thus, at most 8 points of P can reside within the $\delta \times 2\delta$ rectangle. (Note that since points on line l may be in either P_L or P_R, there may be up to 4 points on l. This limit is achieved if there are two pairs of coincident points such that each pair consists of one point from P_L and one point from P_R, one pair is at the intersection of l and the top of the rectangle, and the other pair is where l intersects the bottom of the rectangle.)

Having shown that at most 8 points of P can reside within the rectangle, we can easily see why we need to check only the 7 points following each point in the array Y'. Still assuming that the closest pair is p_L and p_R, let us assume without loss of generality that p_L precedes p_R in array Y'. Then, even if p_L occurs as early as possible in Y' and p_R occurs as late as possible, p_R is in one of the 7 positions following p_L. Thus, we have shown the correctness of the closest-pair algorithm.

Another key implementation issue

How to ensure that the arrays X_L, X_R, Y_L, and Y_R, which are passed to recursive calls, are sorted by the proper coordinate and also that the array Y' is sorted by y-coordinate? Note that if the array X that is received by a recursive call is already sorted, then we can easily divide set P into P_L and P_R in linear time.

The following algorithm splits Y into Y_L and Y_R



```
let $Y_L[1..Y.length]$ and $Y_R[1..Y.length]$ be new arrays
$Y_L.length = Y_R.length = 0$
for $i = 1$ to $Y.length$
    if $Y[i] \in P_L$
        $Y_L.length = Y_L.length + 1$
        $Y_L[Y_L.length] = Y[i]$
    else $Y_R.length = Y_R.length + 1$
        $Y_R[Y_R.length] = Y[i]$
```