Lecture 23

Computational geometry

Supplemental reading in CLRS: Chapter 33 except §33.3

There are many important problems in which the relationships we wish to analyze have geometric structure. For example, computational geometry plays an important role in

- Computer-aided design
- Computer vision
- Computer animation
- Molecular modeling
- Geographic information systems,

to name just a few.

23.1 Intersection Problem

Given a set of line segments $S_1,\ldots,S_n$ in the plane, does there exist an intersection? Humans, with their sophisticated visual systems, are particularly well-equipped to solve this problem. The problem is much less natural (but still important, given the above applications) for a computer.

Input: Line segments $\mathscr{S} = \{S_1,\ldots,S_n\}$ in the plane, represented by the coordinates of their endpoints

Output: *Goal I: Detect.* Determine whether there exists an intersection.
Goal II: Report. Report all pairs of intersecting segments.

The obvious algorithm is to check each pair of line segments for intersection. This would take $\Theta\left(n^2\right)$ time. For Goal II, this can't be improved upon, as it could take $\Theta\left(n^2\right)$ time just to report all the intersections (see Figure 23.1). We will focus on Goal I and show that there exists a $\Theta(n \lg n)$ algorithm.

For ease of exposition, we will assume that our line segments are in *general position*, i.e.,

- No two endpoints have the same x-coordinate. (In particular, there are no perfectly vertical segments.)
- There are no instances of three segments intersecting at a single point.

Both of these assumptions are unnecessary and can be dropped after a couple of minor (but careful) modifications to the algorithm.

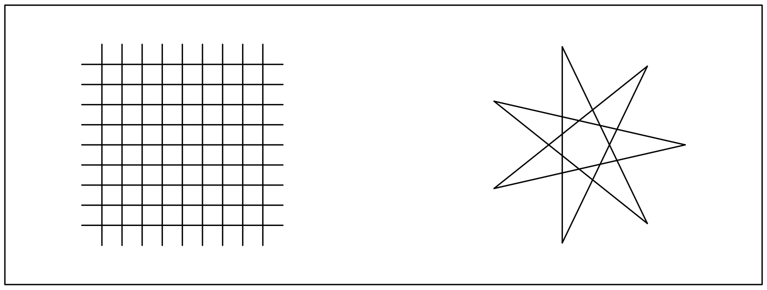

Figure 23.1. If the segments $S_1, \ldots, S_n$ form a square grid, there could be as many as $\left(\frac{n}{2}\right)\left(\frac{n}{2}\right) = \frac{n^2}{4} = \Theta\left(n^2\right)$ intersections. In other situations, it could happen that every pair of segments intersects.

23.1.1 Sweep line and preorder

Suppose you have a sheet of paper with the line segments $S_1, \ldots, S_n$ drawn on it. Imagine dragging a vertical ruler across the page from left to right (see Figure 23.2). This is exactly what we have in mind when we say,

> For each $x \in \mathbb{R}$, let L_x be the vertical line at distance x from the y-axis.[1]

We imagine x increasing over time, so that L_x represents a **sweep line** which moves across the plane from left to right.

For each $x \in \mathbb{R}$, there is a natural *preorder*[2] on the set of non-vertical line segments in the plane: for two non-vertical segments a, b, we say $a \geq_x b$ if the intersection of a (or its extension, if need be) with L_x lies *above* the intersection of b with L_x. This relation needn't be antisymmetric because it could happen that a and b intersect L_x at the same point. In that case, we say that both $a \geq_x b$ and $b \geq_x a$; but of course, this does not imply that $a = b$. Of crucial importance is the following obvious fact:

> Given $x \in \mathbb{R}$ and line segments a and b, there exists an $O(1)$-time algorithm to check whether $a \geq_x b$.[3]

Also important is the following observation:

Observation 23.1. Suppose line segments a and b intersect at the point $P = (x^*, y^*)$. Then one of the following two statements holds:

(i) For $x > x^*$, we have $a \geq_x b$. For $x < x^*$, we have $b \geq_x a$.
(ii) For $x > x^*$, we have $b \geq_x a$. For $x < x^*$, we have $a \geq_x b$.

Conversely, if a and b do *not* intersect, then one of the following two statements holds:

(i) For all x such that L_x intersects a and b, we have $a \geq_x b$.
(ii) For all x such that L_x intersects a and b, we have $b \geq_x a$.

Another way to view the second part of Observation 23.1 is as follows:

[1] It took me a while to figure out how to write down a definition of L_x that didn't include the equation "$x = x$."

[2] A **preorder** is a binary relation that is reflexive and transitive, but not necessarily symmetric or antisymmetric.

[3] It is not so obvious, however, what the best such algorithm is. Most naïve attempts involve calculating a slope, which is numerically unstable. To remedy this, the best algorithms use cross products to compute orientations without using division.

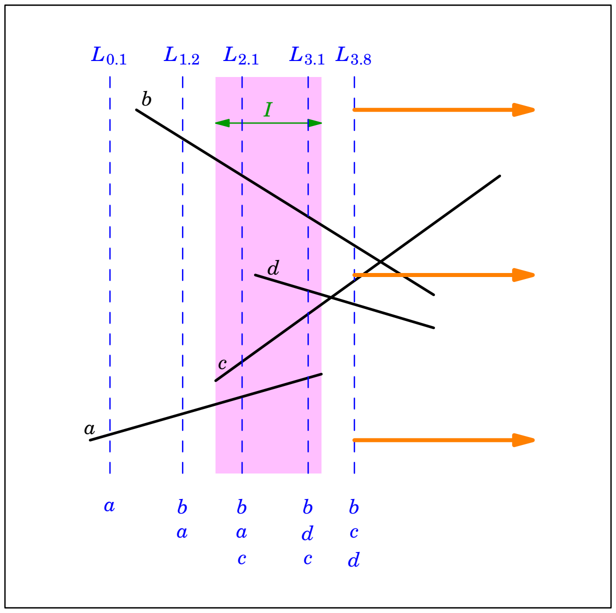

Figure 23.2. A sweep line traversing from left to right. Notice that $b \geq_x d \geq_x c \geq_x a$ for all $x \in I$, but $b \geq_{3.8} c \geq_{3.8} d$.

Thought Experiment 23.2. Consider a finite set of line segments; for concreteness let's say we are looking at the four segments $\{a,b,c,d\}$. There is some interval I (perhaps empty) such that, for each $x \in I$, L_x intersects all four segments (see Figure 23.2). For each $x \in I$, the relation $\geq_x$ induces a preorder structure on the set $\{a,b,c,d\}$. Observation 23.1 implies that, if there are no intersections among $\{a,b,c,d\}$, then this preorder structure does not depend on the choice of x.

Thought Experiment 23.2 leads to the following strategy, which is quite ingenious. To introduce the strategy, we first define the preorder data structure. The **preorder data structure** stores a dynamic collection T of objects along with a preorder on those objects, which we denote by $\geq$. The supported operations are[4]

- INSERT(a) – inserts a into the collection
- DELETE(a) – deletes a from the collection
- ABOVE(a) – returns the element immediately above a. If no such element exists, returns NIL.
- BELOW(a) – returns the element immediately below a. If no such element exists, returns NIL.

The obvious issue with using this data structure in conjunction with our plane sweep is that it only stores a single preorder relation, which we have denoted above by $\geq$ (and the usual counterparts $\leq$, $>$, $<$), whereas the relation $\geq_x$ has the potential to change as x varies. However, this needn't be a problem, as long as:

[4] For ease of exposition, we have obscured one detail from view. It is possible for T to contain two elements d, e such that $d \geq e$ and $e \geq d$. In order for the operations ABOVE(d) and BELOW(d) to work correctly, there must be some rule for breaking ties. To this end, T will also internally store an *order* $\succeq$ on its elements, such that $b \succeq a$ always implies $b \geq a$ (but not necessarily conversely). Then, "the element immediately above a" is the element b such that $b \succ a$ and, whenever $c \succ a$, we always have $c \succeq b$. (Similarly for "the element immediately below a.") The choice of order $\succeq$ does not matter, as long as it is persistent through insertions and deletions.

Condition 23.3. During the lifetime[5] of any particular element a, the ranking of a with respect to the other elements of T does not ever need to be changed.

We are now prepared to give the algorithm:

```
Algorithm: DETECT-INTERSECTION($\mathscr{S}$)
1  Sort the endpoints of segments in $\mathscr{S}$ by $x$-coordinate
2  $T \leftarrow$ new preorder structure
3  ▷ Run the sweepline from left to right
4  for each endpoint $(x, y)$, by order of $x$-coordinate do
5      if $(x, y)$ is the left endpoint of a segment $a$ then
6          Insert $a$ into $T$, using the relation $\geq_x$ to decide where in the preorder $a$ should
           belong*
7          Check $a$ for intersection with $T$.ABOVE($a$)
8          Check $a$ for intersection with $T$.BELOW($a$)
9          if there was an intersection then
10             return TRUE
11     else if $(x, y)$ is the right endpoint of a segment $a$ then
12         Check $T$.ABOVE($a$) for intersection with $T$.BELOW($a$)
13         if there was an intersection then
14             return TRUE
15         $T$.DELETE($a$)
16 return FALSE
```

* This step needs justification. See the proof of correctness.

Proof of correctness. First, note the need to justify line 6. In implementation form, T will probably store its entries in an ordered table or tree; then, when T.INSERT(a) is called, it will take advantage of T's internal order by using a binary search to insert a in $\Theta\left(\lg|T|\right)$ time. In line 6, we are asking T to use the relation $\geq_x$ to decide where to place a. This is fine, *provided that* $\geq_x$ agrees with T's internal preorder when considered as a relation on the elements that currently belong to T. (Otherwise, a binary search would not return the correct result.[6]) So, over the course of this proof, we will argue that

> Whenever line 6 is executed, the relation $\geq_x$ agrees with T's internal preorder when considered as a relation on the elements that currently belong to T.

From Thought Experiment 23.2 we see that the above statement holds as long as the sweep line has not yet passed an intersection point. Thus, it suffices to prove the following claim:

Claim. DETECT-INTERSECTION terminates before the sweep line passes an intersection point.

The claim obviously holds if there are no intersections, so assume there is an intersection. Let's say the leftmost intersection point is P, where segments a and b intersect. Assume without loss of generality that a's left endpoint lies to the right of b's left endpoint (so that b gets inserted into T

[5] By "the lifetime of a," I mean the period of time in which a is an element of T.

[6] Actually, in that situation, there would be no correct answer; the very notion of "binary search" on an unsorted list does not make sense.

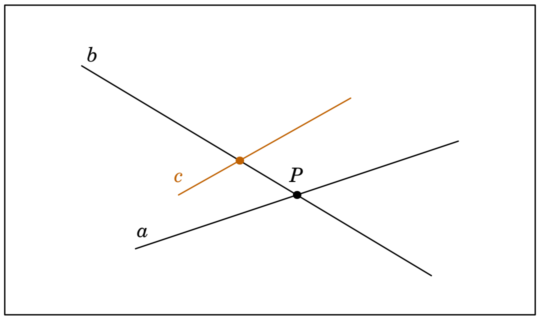

Figure 23.3. If the left endpoint of c lies between a and b to the left of P and the right endpoint of c lies to the right of P, then c must intersect either a or b.

before a does). If a ever becomes adjacent to b in T, then either lines 7–8 or line 12 will detect that a and b intersect, and the algorithm will halt.

So we are free to assume that a and b are never adjacent in T until after the sweep line has passed P. This means that there exists a line segment c such that the left endpoint of c lies between a and b[7] and to the left of P, and the right endpoint of c lies to the right of P. Geometrically, this implies that c must intersect either a or b, and that intersection point must lie to the *left* of P (see Figure 23.3). But this is impossibe, since we assumed that P was the leftmost intersection point. We conclude that the claim holds.

Actually, we have done more than prove the claim. We have shown that, if there exists an intersection, then an intersection will be reported. The converse is obvious: if an intersection is reported, then an intersection exists, since the only time we report an intersection is after directly checking for one between a specific pair of segments. □

23.1.2 Running time

The running time for this algorithm depends on the implementation of the preorder data structure. CLRS chooses to use a red–black tree[8], which has running time $O(\lg n)$ per operation. Thus, DETECT-INTERSECTION has total running time

$$\Theta(n \lg n).$$

23.2 Finding the Closest Pair of Points

Our next problem is simple:

Input: A set Q of points in the plane

Output: The two points of Q whose (Euclidean) distance from each other is shortest.

The naïve solution is to proceed by brute force, probing all $\binom{n}{2}$ pairs of points and taking $\Theta(n^2)$ time. In what follows, we will exhibit a subtle divide-and-conquer algorithm which runs in $\Theta(n \lg n)$ time.

[7] That is, if the left endpoint of c is (x, y), then either $a \le_x c \le_x b$ or $a \ge_x c \ge_x b$.
[8] For more information about red–black trees, see Chapter 13 of CLRS.

In §23.1, the pseudocode would not have made sense without a few paragraphs of motivation beforehand. By contrast, in this section we will give the pseudocode first; the proof of correctness will elucidate some of the strange-seeming choices that we make in the algorithm. This is more or less how the algorithm is presented in §33.4 of CLRS.

The algorithm begins by pre-sorting the points in Q according to their x- and y-coordinates:

```
Algorithm: CLOSEST-PAIR($Q$)
$X \leftarrow$ the points of $Q$, sorted by $x$-coordinate
$Y \leftarrow$ the points of $Q$, sorted by $y$-coordinate
return CLOSEST-PAIR-HELPER($X, Y$)
```

Most of the work is done by the helper function CLOSEST-PAIR-HELPER, which makes recursive calls to itself:

```
Algorithm: CLOSEST-PAIR-HELPER($X, Y$)
if $|X| \leq 3$ then
    Solve the problem by brute force and return
$x^* \leftarrow$ the median $x$-coordinate of $X$
Let $X_L \subseteq X$ consist of those points with $x$-coordinate $\leq x^*$
Let $X_L \subseteq X$ consist of those points with $x$-coordinate $> x^*$
Let $Y_L \subseteq Y$ consist of those points which are in $X_L$
Let $Y_R \subseteq Y$ consist of those points which are in $X_R$
▷ Find the closest two points in the left half
$\langle p_L, q_L \rangle \leftarrow$ CLOSEST-PAIR-HELPER($X_L, Y_L$)
▷ Find the closest two points in the right half
$\langle p_R, q_R \rangle \leftarrow$ CLOSEST-PAIR-HELPER($X_R, Y_R$)
$\delta_R \leftarrow$ distance from $p_L$ to $q_L$
$\delta_R \leftarrow$ distance from $p_R$ to $q_R$
$\delta \leftarrow \min\{\delta_L, \delta_R\}$
$Y' \leftarrow$ those points in $Y$ whose $x$-coordinate is within $\delta$ of $x^*$
▷ Recall that $Y'$ is already sorted by $y$-coordinate
$\epsilon \leftarrow \infty$
for $i \leftarrow 1$ to $|Y'| - 1$ do
    $p \leftarrow Y'[i]$
    for $q$ in $Y'[i+1, \ldots, \min\{i+7, |Y'|\}]$ do
        if $\epsilon >$ distance from $p$ to $q$ then
            $\epsilon \leftarrow$ distance from $p$ to $q$
            $p^* \leftarrow p$
            $q^* \leftarrow q$
if $\epsilon < \delta$ then
    return $\langle p^*, q^* \rangle$
else if $\delta_R < \delta_L$ then
    return $\langle p_R, q_R \rangle$
else
    return $\langle p_L, q_L \rangle$
```

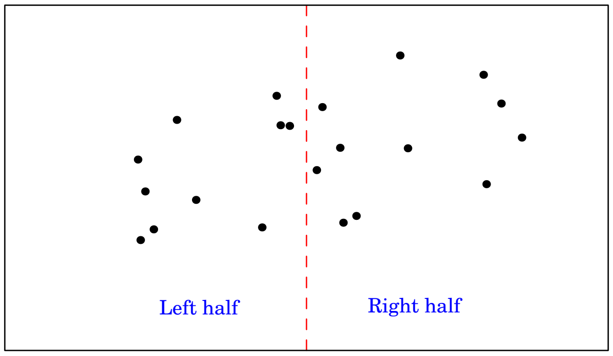

Figure 23.4. CLOSEST-PAIR-HELPER divides the set X into a left half and a right half, and recurses on each half.

23.2.1 Running time

Within the procedure CLOSEST-PAIR-HELPER, everything except the recursive calls runs in linear time. Thus the running time of CLOSEST-PAIR-HELPER satisfies the recurrence

$$T(n) = 2 \cdot T\left(\tfrac{n}{2}\right) + O(n), \tag{23.1}$$

which (by the Master Theorem in §4.5 of CLRS) has solution

$$T(n) = O(n \lg n). \tag{23.2}$$

Note that, if we had decided to sort within each recursive call to CLOSEST-PAIR-HELPER, the $O(n)$ term in (23.1) would have instead been an $O(n \lg n)$ term and the solution would have been $T(n) = O\left(n (\lg n)^2\right)$. This is the reason for creating a helper procedure to handle the recursive calls: it is important that the lists X and Y be pre-sorted so that recursive calls need only linear-time operations.

Note also that, if instead of lines 20–24 we had simply checked the distance between each pair of points in Y', the $O(n)$ term in (23.1) would have instead been an $O\left(n^2\right)$ term, and the solution would have been $T(n) = O\left(n^2\right)$.

23.2.2 Correctness

CLOSEST-PAIR-HELPER begins by recursively calling itself to find the closest pairs of points on the left and right halves. Thus, lines 15–24 are ostensibly an attempt to check whether there exists a pair of points $\langle p^*, q^* \rangle$, with one point on the left half and one point on the right half, whose distance is less than that of any two points lying on the same half. What remains to be proved is that lines 15–24 do actually achieve this objective.

By the time we reach line 15, the variable δ stores the shortest distance between any two points that lie on the same half. It is easy to see that there will be no problems if δ truly is the shortest possible distance. The case we need to worry about is that in which the closest pair of points—call it $\langle p, q \rangle$—has distance less than δ. In such a case, the x-coordinates of p and q would both have to be within δ of x^*; so it suffices to consider only points within a vertical strip V of width 2δ centered at $x = x^*$ (see Figure 23.5). These are precisely the points stored in the array Y' on line 15.

Say $p = Y'[i]$ and $q = Y'[j]$, and assume without loss of generality that $i < j$. In light of lines 20–24 we see that, in order to complete the proof, we need only show that $j - i \le 7$.

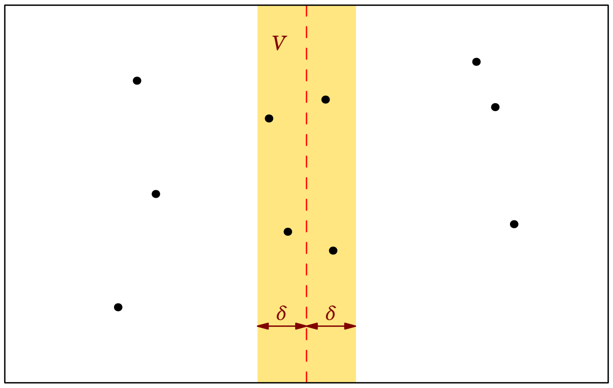

Figure 23.5. It suffices to consider a vertical strip V of width 2δ centered at $x = x^*$.

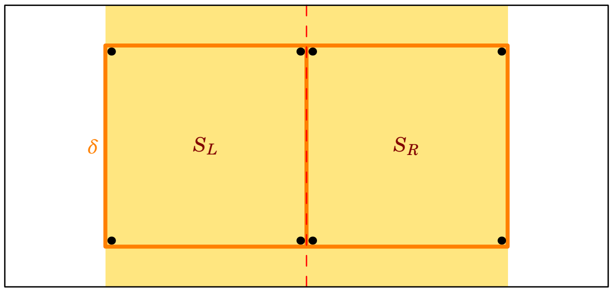

Figure 23.6. Each of S_L and S_R can hold at most 4 points. (Actually, in completely ideal geometry, S_R cannot contain 4 points because its left boundary is excluded. But since the coordinates in a computational geometry problem are typically given as floating point numbers, we are not always guaranteed correct handling of edge cases.)

Say $p = (p_x, p_y)$. Let S_L be the square (including boundaries) of side length δ whose right side lies along the vertical line $x = x^*$ and whose bottom side lies along the horizontal line $y = p_y$. Let S_R be the square (excluding the left boundary) of side length δ whose left side lies along the vertical line $x = x^*$ and whose bottom side lies along the horizontal line $y = p_y$. It is evident that q must lie within either S_L or S_R (see Figure 23.6). Moreover, any two points in the region S_L are separated by a distance of at least δ; the same is true for any two points in the region S_R. Thus, by a geometric argument[9], S_L and S_R each contain at most four points of Y'. In total, then, $S_L \cup S_R$ contains at most eight points of Y'. Two of these at-most-eight points are p and q. Moreover, since Y' consists of all points in V sorted by y-coordinate, it follows that the at-most-eight points of $S_L \cup S_R$ occur *consecutively* in Y'. Thus, $j - i \le 7$.

[9] One such argument is as follows. Divide S_L into northeast, northwest, southeast and southwest quadrants. Each quadrant contains at most one point of Y' (why?), so S_L contains at most four points of Y'.