FTP_Alg
Sweep algorithms: segment intersection and closest pair of points

jungkyu.canci@hslu.ch

28. October 2024

Introduction

- Computational geometry is a branch of computer science that studies algorithms for solving geometric problems.
- Here we study three problems:
  1. Determining whether two line segments intersect.
  2. Determining whether any pair of line segments intersects in a set of given segments.
  3. Finding the closest pair of points.
- We start by presenting some basic notions on segments.
  - Let $p_1 = (x_1, y_1), p_2 = (x_2, y_2)$ be two distinct points on the xy–plane. The set

$$\overline{p_1 p_2} = \{(x, y) = (x_1, y_1) + \alpha(x_2 - x_1, y_2 - y_1) \mid \alpha \in [0, 1]\}$$

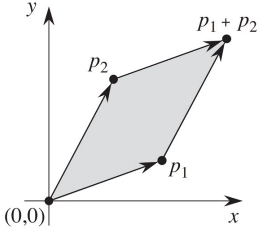
    consists of all **convex combinations** of the points p_1 and p_2.
  - $\overline{p_1 p_2}$ is also called **line segment** (or simply **segment**).

Cross products

- If in $\overline{p_1 p_2}$ the ordering of p_1 and p_2 is relevant we speak of the **directed segment** (or vector) $\overrightarrow{p_1 p_2}$.
- Let $O = (0, 0)$ the origin. Let p be a point of the plane. With abuse of notation we often denote by p the vector $\overrightarrow{Op}$ as well.
- Let $p_1 = (x_1, y_1), p_2 = (x_2, y_2)$ be two points (and so vectors). The **cross product** is the number:

$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1 .$$

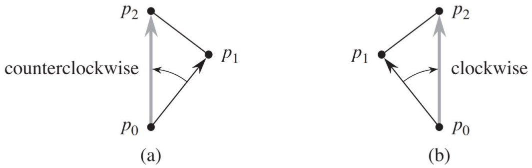
A justification from *Linear Algebra* shows that the magnitude $|p_1 \times p_2|$ is the area of the parallelogram with sides the segments p_1 and p_2.



Determining whether consecutive segments turn left or right

Let p_0, p_1, p_2 be three points on the xy–plane. We consider the two vectors $v_1 = p_1 - p_0$ and $v_2 = p_2 - p_0$. We consider their cross product

$$v_1 \times v_2 = (p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0).$$



By moving by an angle with amplitude in $[0, \pi]$ (in radiant) we are in one of the above cases. In (a) we have $v_1 \times v_2 > 0$ and in (b) we have $v_1 \times v_2 < 0$. And $v_1 \times v_2 = 0$ iff v_1 and v_2 are collinear.

Determining whether two line segments intersect

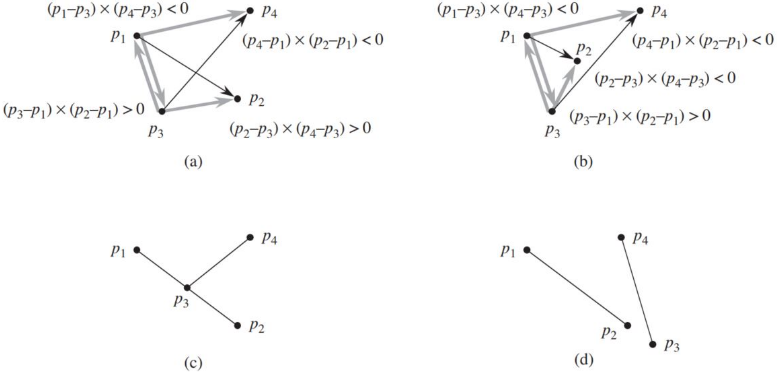
- For given four points p_1, p_2, p_3, p_4 on the xy–plane, we consider the two segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$.
- We want to check whether the two segments intersect each other. We assume that we have 4 distinct points, otherwise the corresponding segments trivially intersects.

```
SEGMENTS-INTERSECT(p1, p2, p3, p4)
d1 = DIRECTION(p3, p4, p1)
d2 = DIRECTION(p3, p4, p2)
d3 = DIRECTION(p1, p2, p3)
d4 = DIRECTION(p1, p2, p4)
if ((d1 > 0 and d2 < 0) or (d1 < 0 and d2 > 0)) and
      ((d3 > 0 and d4 < 0) or (d3 < 0 and d4 > 0))
      return TRUE
elseif d1 == 0 and ON-SEGMENT(p3, p4, p1)
      return TRUE
elseif d2 == 0 and ON-SEGMENT(p3, p4, p2)
      return TRUE
elseif d3 == 0 and ON-SEGMENT(p1, p2, p3)
      return TRUE
elseif d4 == 0 and ON-SEGMENT(p1, p2, p4)
      return TRUE
else return FALSE
```

Where we use the two procedure

```
DIRECTION(pi, pj, pk)
return (pk − pi) × (pj − pi)

ON-SEGMENT(pi, pj, pk)
if min(xi, xj) ≤ xk ≤ max(xi, xj) and min(yi, yj) ≤ yk ≤ max(yi, yj)
      return TRUE
else return FALSE
```



In (a) we have ($d_1 < 0$ and $d_2 > 0$) and ($d_3 > 0$ and $d_4 < 0$). In (b) we have ($d_1 < 0$ and $d_2 < 0$). In (c) we have $d_3 = 0$ and ON–SEGMENT(p_1, p_2, p_3) true. In (d) we have $d_3 = 0$ but ON–SEGMENT(p_1, p_2, p_3) is not true.

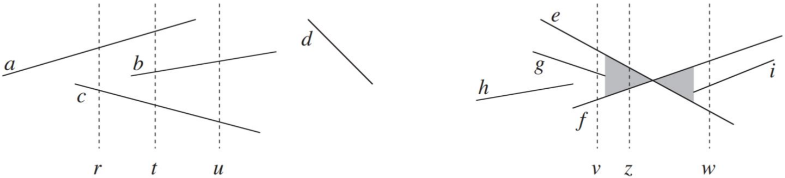
Determining whether any pair of segments intersects

- We consider a given set of segments on the plane and we present an algorithm to check whether there exists a pair of such a segments, which intersects each other.
- We will use a technique called **sweeping**, which is used in many algorithms in computational geometry. A vertical line is considered, i.e. the **sweep line**, which passes through a given set of objects (in our case points), usually from left to right.
- We consider a finite set $\mathscr{S}$ of segments of the plane. For simplicity of presentation we assume that no segment is vertical (parallel to the y–axis) and no three segments meet in a point. (These latter assumptions can be removed with a slight modification).

Ordering segments

- Here we present a partial order on the set $\mathscr{S}$ of segments.
- Let r be a real number. We consider the sweep line, i.e. the vertical line, of the point of the plane having first coordinate $x = r$.
- We say that two segments s_1, s_2 are **compatible** at r, if the both segments intersect the sweep line $x = r$. Recall that no line in $\mathscr{S}$ are vertical so $s \in \mathscr{S}$ intersects the line $x = r$ at most once.
- We give a partial order on $\mathscr{S}$. For $s_1, s_2 \in \mathscr{S}$ we say that s_1 is **above** s_2 at r if are comparable at r and the intersection of s_1 with the sweep line is higher than or equal to the one of s_2 with the sweep line. We write $s_1 \succcurlyeq_r s_2$.

Total Preorder



(a) (b)

- The relation $\succcurlyeq_r$ is clearly reflexive, transitive, but is neither symmetric nor antisymmetric.
- Thus $\succcurlyeq_r$ is a total preorder on the subset of $\mathscr{S}$ of segment intersecting the sweep line $\{x = r\}$.

Moving the sweep line

As usual in general with sweeping algorithms, we will manage two sets of data:

- The **sweep–line status**: we will consider the above described total orders associated to each sweep lines.
- The **event point schedule**: We will consider a sequence of points, called **event points**, ordered from left to right. We will let move ("continuously") the sweep line from left to right and stop at these *event points*. The process will analyze the sweep–line status at this points, and then resume.
- The *event points* will be the end points of the segments.
- The event points will be sorted according to the x–coordinate. If two or more endpoints have same x–coordinate, we breaks ties by putting before left endpoints than right endpoint. For tails we put before endpoints with smaller y–coordinate.

Operations of sweep–line status

Let T be a total preorder associated to a sweep–line status. We will consider the following operations on T.

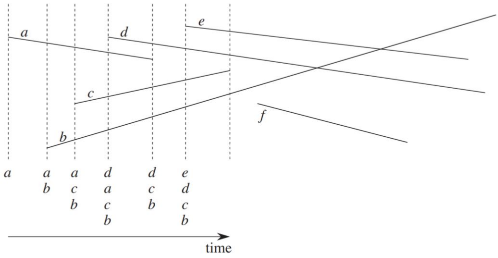
- INSERT(T, s): insert the segment s into T.
- DELETE(T, s): delete segment s from T.
- ABOVE(T, s): return the segment immediately above the segment s in T.
- BELOW(T, s): return the segment immediately below the segment s in T.

We can perform all the above operations in $O(\ln n)$ by using red–black trees. Instead of using red–black trees you could compare two elements in T by using the cross ratio (exercise).

Pseudocode

```
ANY-SEGMENTS-INTERSECT(S)
T = ∅
sort the endpoints of the segments in S from left to right,
        breaking ties by putting left endpoints before right endpoints
        and breaking further ties by putting points with lower
        y-coordinates first
for each point p in the sorted list of endpoints
        if p is the left endpoint of a segment s
                INSERT(T, s)
                if (ABOVE(T, s) exists and intersects s)
                        or (BELOW(T, s) exists and intersects s)
                        return TRUE
        if p is the right endpoint of a segment s
                if both ABOVE(T, s) and BELOW(T, s) exist
                        and ABOVE(T, s) intersects BELOW(T, s)
                        return TRUE
                DELETE(T, s)
return FALSE
```

The start is by taking an empty total preorder. Line 2 sort the end point as described in slide 10. The loop at lines 3-11 processes an event point. Line 5 adds the segment to T (after test in 4).



Line 6 and 7 needs SEGMENTS–INTERSECTS and lines 9 and 10 as well. Line 9 and 11 check if the surrounding segments of s intersects. If they do not intersect line 11 delete s. Line 1 costs $O(1)$. line 2 $O(n \log n)$ (ex. with Merge Sort). The **for** loop iterate at most for each event point, so at most $2n$. As pointed out before we have at each iteration $O(\log n)$ running time, because we have at most 6 call of the operations describe in slide 11. Thus the running time of the code is $O(n \log n)$. But, why the algorithms works? (See exercise.)

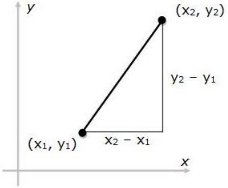
Finding the closest pair of points (Optional part)

We consider a finite set of n points Q of the xy–plane.

For given points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ we define

$$d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

the **distance between p_1 and p_2**.



We consider the problem of finding "the" closest pair of points in Q, in the sense that it minimize the above distance.
We could consider a brute–force solution that consider the distance of all pair of points in Q, there are $\binom{n}{2} = \frac{n(n-1)}{2}$. Therefore we will need a $\Theta(n^2)$ running time.

The divide–and-conquer algorithm

- We present a divide–and–conquer algorithm, whose running time $T(n)$ is described by the recurrence $T(n) = 2T(n/2) + O(n)$. As we have already seen, $T(n) = O(n \log n)$ holds.
- Each recursive call of the algorithm, it takes as inputs a subset $P \subset Q$ and arrays X and Y. Both arrays contain all points in P.
- The elements in X are so sorted so that their x–coordinate are sorted in monotonically increasing order. The same for Y with respect to the y–coordinate.
- We will consider a initial order of X and Y in the case of $P = Q$ and a recursive procedure, so that at each recursion the order of recursive subsets X and Y is maintained with a cost at most linear.

The recursion starts by checking if $|P| \leq 3$. If so the algorithm check all $\binom{|P|}{2}$ pairs. If $|P| > 3$ the recursion carries out the divide–and–conquer paradigm as follows:

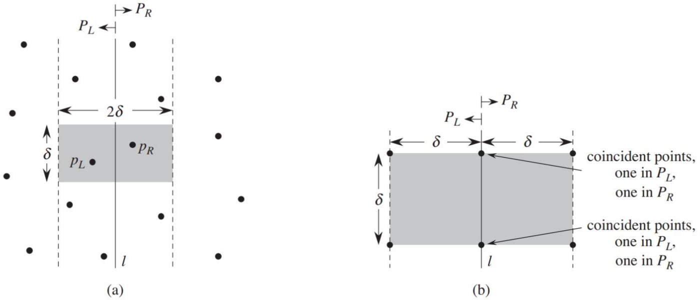
- **Divide**: We divide the set P with a vertical line in two set P_L and P_R with $|P_L| = \lceil |P|/2 \rceil$ and $|P_R| = \lfloor |P|/2 \rfloor$. By denoting X_L (and X_R) the set of points of P_L (and of P_R respectively) sorted by x–coordinates. Since X is assumed monotonically sorted, the same holds for X_L and X_R. We divide similarly Y in the subset Y_L and Y_R. The procedure that we will consider in slide 18 maintain the order of Y_L and Y_R in linear time.
- **Conquer**: We have the two subset P_L and P_R, as subdivison of P. We iterate the call on P_L and P_R, obtaining a closest pair of points in P_L having distance δ_L and similarly for P_R obtaining the minimal distance δ_R. We denote $\delta = \min\{\delta_L, \delta_R\}$.
- **Combine**: This step is subtle, because the closest pair of points in P can be composed from one point in P_L and one in P_R. To find such a pair, if it exists, we do the following:

Combine (continuation)

1. Y' is the subset of Y contained in the 2δ–wide strip. By removing the outside points, Y' inherits the order from Y.
2. For each point p in Y', we try to find in Y' points having distance less than δ. We will see in next slide that for each point $p \in Y'$ one hast to check the distance with at most other 7 points. We keep track of the closest pair distance δ' found over all pairs of points in Y'.
3. If $\delta' < \delta$, then the 2δ–wide strip contain a closer pair than the recursive calls found. Return the distance δ' and the pair realizing the distance δ'. Otherwise return the pair of the recursive call and its distance δ.

To prove that the algorithm runs in $O(n \log n)$ we have to prove that the combine step is not time expensive more than $O(n)$. Actually it is so, because at each point $p \in Y'$ we have to check the distance with at most 7 other points. We have to also show that the arrays X_L, X_R, Y_L, Y_R and Y' are at each call sorted in at most linear time.

“At most 7 points...”



In **Combine**, we have to determine whether a point p_L in P_L and p_R in P_R have distance less than δ (see (a)). These two points, if they exist, should be contained in rectangle with size $2\delta \times \delta$. Now it is enough to consider that there are at most 8 points in such a rectangle, see (b), where one is the point p of part 2. in **Combine**.

X_L, X_R, Y_L, Y_R and Y' are already sorted

The sorting of X_L and X_R is inherited from X, since we split P vertically. Dividing P needs only linear time. Similarly for Y'. For Y_L and Y_R we use an opposite MERGE procedure.

```
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] ∈ 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]
```

which only costs linear time. This shows that the running time $T_r(n)$ of the recursive part satisfy the following:

$$T_r(n) = \begin{cases} O(1) & \text{if } n \leq 3 \\ 2T_r(n/2) + O(n) & \text{if } n > 3 \end{cases}$$

Thus $T_r(n) = O(n \log n)$. We need $O(n \log n)$ for sorting X and Y at the start. Thus the running time of the algorithm is $T(n) = O(n \log n)$.

Reference

The material of these slides is taken form the book "Introduction to Algorithms" by de Cormen et al., Section 33.1, Section 33.2 and Section 33.4.