Eksamensnoter - Computational Geometry

André Oskar Andersen (wpr684)

March 31, 2020

33 Computational Geometry

- Computational geometry is the branch of computer science that studies algorithms, for solving geometric problems
- The input to a computational-geometry problem is typically a description of a set of geometric objects, such as a set of points, a set of line segments, or the vertices of a polygon in counterclockwise order
- The output is often a response to a query about the objects, such as whether any of the lines intersect, or perhaps a new geometric object, such as the convex hull of the set of points
- In this chapter, we look at a few computational-geometry algorithms in two dimensions. We represent each input object by a set of points $\{p_1, p_2, p_3, ...\}$, where $p_i = (x_i, y_i)$ and $x_i, y_i \in \mathbb{R}$

33.1 Line-segment properties

- A *convex combination* of two distinct points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is any point $p_3 = (x_3, y_3)$ such that for some α in the range $0 \leq \alpha \leq 1$, we have $x_3 = \alpha x_1 + (1 - \alpha)x_2$ and $y_3 = \alpha y_1 + (1 - \alpha)y_2$. We also write that $p_3 = \alpha p_1 + (1 - \alpha)p_2$. Intuitively, p_3 is any point that is on the line passing through p_1 and p_2 and is on or between p_1 and p_2 on the line.
- Given two distinct points p_1 and p_2, the *line segment* $p_1\bar{}p_2$ is the set of convex combinations of p_1 and p_2
- We call p_1 and p_2 the *endpoints* of segment $p_1\bar{}p_2$.
- Sometimes the ordering of p_1 and p_2 matters, and we speak of the *directed segment* $\overrightarrow{p_1 p_2}$
- If $p_1$is the origin, then we can treat the directed segment $\overrightarrow{p_1 p_2}$ as the vector p_2

Cross products

- We can interpret the *cross product* $p_1 \times p_2$ as the signed area of the parallelogram formed by the points $(0, 0), p_1, p_2$ and $p_1 + p_2 = (x_1 + x_2, y_1 + y_2)$. An equivalent definition gives the cross product as the determinant of a matrix:

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

- if $p_1 \times p_2$ is positive, then p_1 is clockwise from p_2, with respect to the origin $(0, 0)$; if this cross product is negative, then p_1 is counterclockwise from p_2. A boundary condition arises if the cross product is 0; in this case, the vectors are *colinear*, pointing in either the same or opposite directions

- To determine whether a directed segment $\overrightarrow{p_0p_1}$ is closer to a directed segment $\overrightarrow{p_0p_2}$ in a clockwise direction or in a counterclockwise direction with respect to their common endpoint p_0, we simply translate to use p_0 as the origin. That is, we let $p_1 - p_0$ denote the vector $p'_1 = (x'_1, y'_1)$, where $x'_1 = x_1 - x_0$ and $y'_1 = y_1 - y_0$, and we define $p_2 - p_0$ similarly. We then compute the cross product $(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$. If this cross product is positive, then $\overrightarrow{p_0p_1}$ is clockwise from $\overrightarrow{p_0p_2}$; if negative, it is counterclockwise.

Determining whether consecutive segments turn left or right

- Our next question is whether two line segments $\overline{p_0p_1}$ and $\overline{p_1p_2}$ turn left or right at point p_1. Equivalently, we want a method to determine which way a given angle $\angle p_0p_1p_2$ turns.

- We simply check whether directed segment $\overrightarrow{p_0p_2}$ is clockwise or counterclockwise relative to directed segment $\overrightarrow{p_0p_1}$. To do so, we compute the cross product $(p_2 - p_0) \times (p_1 - p_0)$. If the sign of this cross product is negative, then $\overrightarrow{p_0p_2}$ is counterclockwise with respect to $\overrightarrow{p_0p_1}$, and thus we make a left turn at p_1. A positive cross product indicates a clockwise orientation and a right turn. A cross product of 0 emeans that points $p_0, p_1,$ and p_2 are colinear.

Determining whether two lines segments intersect

- To determine whether two line segments intersect, we check whether each segment straddles the line containing the other. A segment $\overline{p_1p_2}$ *straddles* a line if point p_1 lies on one side of the line and point p_2 lies on the other side. A boundary case arises if p_1 or p_2 lies directly on the line. Two line segments intersect if and only if either (or both) of the following conditions holds:

  1. Each segment straddles the line containing the other
  2. An endpoint of one segment lies on the other segment (comes from the boundary case)

Determining whether any pair of segments intersects

- This section presents an algorithm for determining whether any two line segments in a set of segments intersect

- The algorithm runs in $O(n \lg n)$ time, where n is the number of segments we are given. It determines only whether or not any intersection exists; it does not print all the intersections.

- In *sweeping*, an imaginary vertical *sweep line* passes through the given set of geometric objects, usually from left to right. The line-segment-intersection algorithm in thi ssection considers all the line-segment endpoints in the left-to-right order and checks for an intersection each time it encounters an endpoint.

Ordering segments

- We can order the segments that intersect a verical sweep line according to the y-coordinates of the points of intersection

- Consider two segments s_1 and s_2. We say that these segments are *comparable* at x if the vertical sweep line with x-coordinate x intersects both of them. We say that s_1 is *above* s_2 at x, written $s_1 \succeq s_2$, if s_1 and s_2 are comparable at x and the intersection of s_1 with the sweep line at x is higher than the intersection of s_2 with the same sweep line, or if s_1 and s_2 intersect at the sweep line.

- For a given x, the relation " $\succeq_x$ " is a total preorder for all segments that intersect the sweep line at x. That is, if segments s_1 and s_2 each intersect the sweep line at x, then either $s_1 \succeq s_2$ or $s_2 \succeq s_1$, or both

Moving the sweep line

- Sweeping algorithms typically manage two sets of data:

  1. The *sweep-line status* gives the relationships among the objects that the sweep line intersects
  2. The *event-point schedule* is a sequence of points, called *event points*, which we order from left to right according to their x-coordinates. As the sweep progresses from left to right, wehnever the sweep line reaches the *x-coordinate* of an event point, the sweep halts, processes the event point, and the resumes. Changes to the sweep-line status occur only at event points

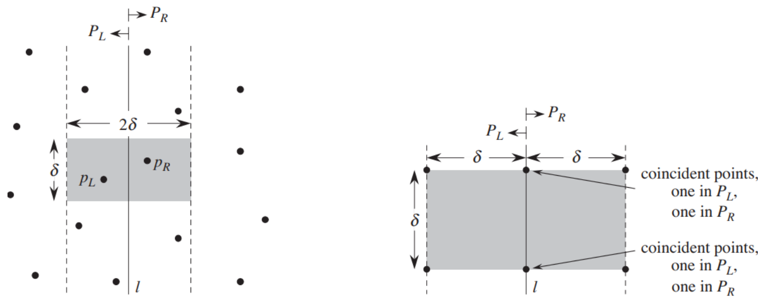
- Each segment endpoint is an event point. We sort the segment endpoints by increasing x-coordinate and proceed from left to right. When we encounter a segment's left endpoint, we insert the segment into the sweep-line status, and we delete the segment from the sweep-line status upon encountering its right endpoint. Whenever two segments first become consecutive in the total preorder, we check whether they intersect

33.4 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.

- The divide-and-conquer algorithm for this problem, has a running time described by $T(n) = 2T(n/2) + O(n)$

The divide-and-conquer algorithm



- Each recursive invocation of the algorithm takes as input a subset $P \subseteq Q$ and arrrays X and Y, each of which contains all the points of the inpput subset P. The points in array X are sorted so that their x-coordinates are monotonically increasing. Similarly, array Y is sorted by monotonically increasing y-coordinate.

- A given recursive invocation with inputs P, X, and Y first checks whether $|P| \leq 3$. If so, the invocation simply performs the brute-force method: try all $\binom{|P|}{2}$ pairs of points and return the closest pair. If $|P| > 3$, the recursive invocation carries out the divide-and-conquer paradigm as follows:

  – **Divide**: Find a line l that bisects the point set P into two equally sized sets P_L and P_R. 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 poitns in P_L and the other to find the closest pair of points in P_R. The input 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. 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 2δ-wide vertical strip centered at line l. To find such a pair, we do the following:

    1. Create an array Y', which is the array Y with all points not in the 2δ-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. Only the 7 points in Y' that follow p need to

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 verical 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.

Correctness

- By bottoming out the recursion when $|P| \leq 3$, we ensure that we never try to solve a subproblem consisting of only one point.

- We need only to check the 7 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, p_L and p_R are within a $\delta \times 2\delta$ rectangle centered at l.

- 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. Similarly, at most 4 points in P_R can reside within the δ square forming the right half of the rectangle. Thus, at most 8 points of P can reside within $\delta \times 2\delta$ rectangle.

- Now, we can 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 that p_L precedes p_R in array Y'. Then, p_R is in one of the 7 positions following p_L. Thus, we have shown the correctness of the closest-pair algorithm.