Computational Geometry (Fall 2012)
Project 2: Point Location

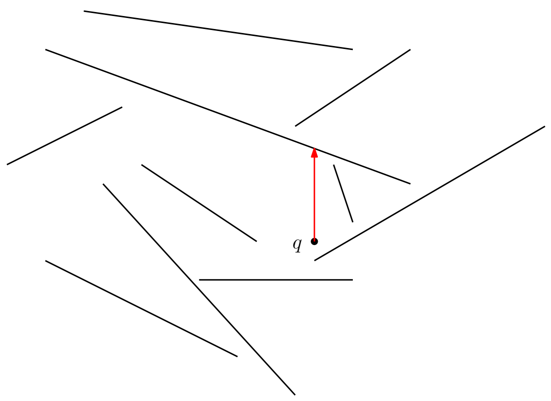
Peyman Afshani

October 10, 2012

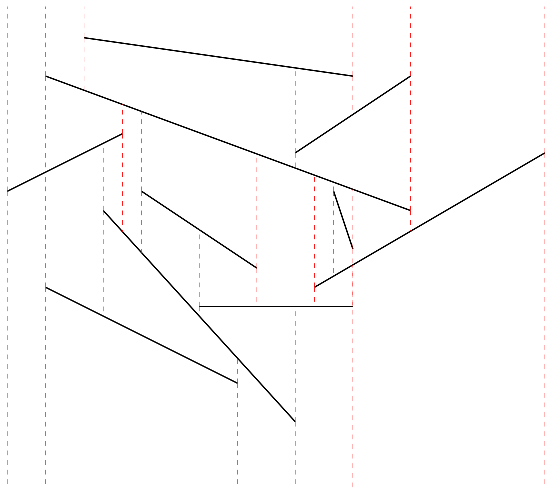
Parts (A) and (B) are mandatory (trapezoidal map decomposition). Part (C) is optional (*kd*-trees).

In this project, you are asked to implement a point location data structure to solve the following problem, known as the *vertical ray shooting problem*: for a set S of n *disjoint* line segments in the plane, build a data structure such that given a query point $q = (x_q, y_q)$, the first line segment directly above q can be found efficiently (see Figure 1(a)).

To do this, first you must build the trapezoid decomposition of the line segments (See Figure 1(b)) and then build a data structure for point location queries. This point location data structure is able to find the face of the trapezoid decomposition that contains q. If you place a pointer from each face of the decomposition to its top segment, then you can easily solve the vertical ray shooting problem using the point location data structure.



(a) The segment directly above q is the first line segment hit by a vertical ray shot from q.



(b) The trapezoid decomposition of the line segments.

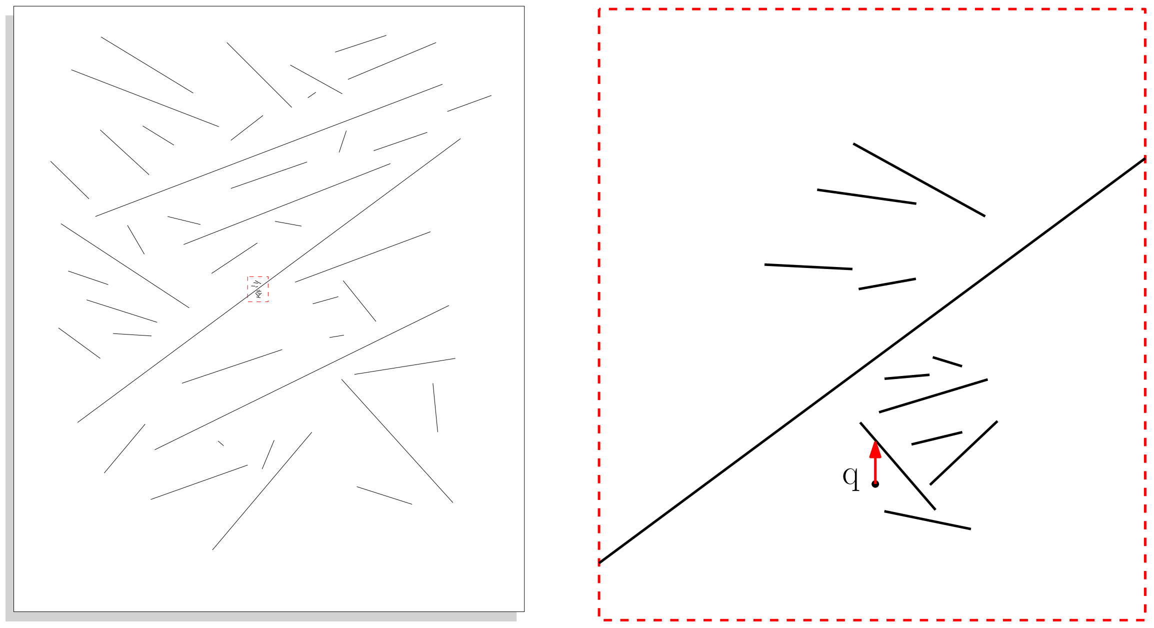
Figure 1: A set of line segments and their trapezoid decomposition.

Your implementations should accept an input file of the following format. The i-th line of the input file will be empty if it is the last line of the file. Otherwise, the i-th line will describe the segment s_i and it will contain four floating point numbers of which the first two represent the x- and y-coordinate of one end point of s_i, and the last two represent the x- and y-coordinate of the other end point of s_i. These numbers are separated by a single space.

You can assume the input contains no vertical line segments.

Part A (trapezoidal map). In this part, you can assume that the input file contains few line segments (at most 100). After building the point location data structure, graphically draw the line segments. Next, your data structure should accept queries interactively with a mouse (for example, each mouse left click can be a query point). Note that you cannot assume that the input line segments have integer coordinates so you must convert the point returned by the mouse click into the coordinate system of the line segments. Using the point location data structure, find the line segment above the query point and then highlight it.

Part B (a better user interface). In this part you can assume you have a moderately large input file (at most 10000 line segments). As before, build the point location data structure on the input line segments. In this part, you have the option of either drawing all the line segments or only a subset of them (e.g., only those longer than a certain threshold). However, you should support a zooming functionality by which the user should be able to zoom in as well as being able to zoom out. The user should also be able to issue point location queries as before by clicking on the screen (see Figures 2(a) and 2(b)).



(a) An input with many segments. The user decides to zoom inside the red rectangle.

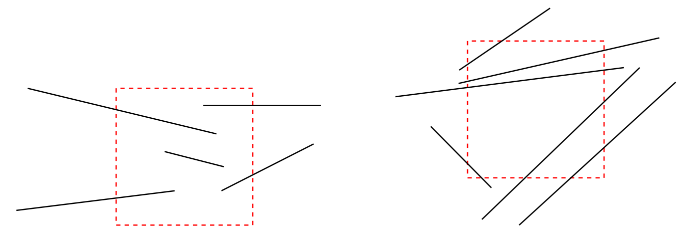
(b) The result of the zoom operation. The user also asks a query inside the zoomed region.

Figure 2: An example of zooming in functionality.

Finding which segments should be displayed can be a problem if you want to do it efficiently. In this part, we do not ask you to have an efficient visualization solution. You can simply check all the segments to see if they intersect the viewed region, and if so, display the part that falls inside the region.

Part C$^+$. In this part, you should be able to handle large files (e.g., files containing 5 or 10 million line segments). Because of this, the previous method of zooming in or zooming out will no longer be time efficient (essentially, you want to be able to zoom in or zoom out in a faction of a second).

Assume the user wants to zoom on a rectangle z with (a, b) and (c, d) as its the bottom-left and top-right corners, respectively. Consider a segment s. There are two main types of intersections between s and z. In the type I intersection, z contains at least one end point of s (see Figure 3(a)). In the type II intersection, s simply "cuts through" z without having any end points inside z (see Figure 3(b))


(a) A number of type I intersections.

(b) A number of type II intersections.

Figure 3: Different cases for the line segments intersecting the rectangle.

Type I intersections can be found using a data structure for orthogonal range reporting. However, type II intersections are more difficult to find. Technically, finding type II intersections is a bit more difficult. In Chapter 10 of the book we shall see how to do this but for now we are looking for a simpler solution.

Observe that the segments that cut through z intersect the boundary of z. Thus, to find them, it suffices to find the face of the trapezoid decomposition that contains one corner of z and then "walk" along the boundary of z. Each time we enter a new face, we check its boundary segments to see if they intersect z and if so, we add them to the list of segments that we must draw.

To summarize, every time the user wants to zoom in (or zoom out) on a rectangle z, you must find the segments that should be displayed in the following way:

- Using an orthogonal range reporting data structure, find the segments with at least one end point inside z. Build a *kd*-trees on the end points of the segments to do this.
- Walk along the boundary of z, and find the segments that cut through z (see Figure 4). You need to place the right pointers from a face to its neighbors to do this.

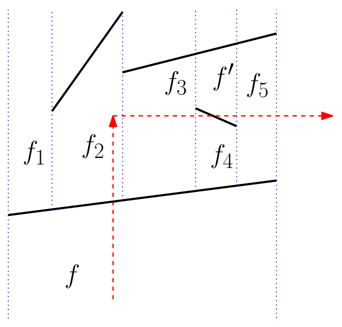


Figure 4: Starting from face f, we check all the neighbors of f (f_1 to f_5), then find out that we must traverse into f_2. Then we continue visiting f_3, f_4, f', f_5 and so on. During this process, we can find all the line segments that intersect the boundary of the region.