27 Geometric Intersection

A natural problem that arises frequently in applications involving geometric data is: "Given a set of N objects, do any two intersect?" The "objects" involved may be lines, rectangles, circles, polygons, or other types of geometric objects. For example, in a system for designing and processing integrated circuits or printed circuit boards, it is important to know that no two wires intersect to make a short circuit. In an industrial system for designing layouts to be executed by a numerically controlled cutting tool, it is important to know that no two parts of the layout intersect. In computer graphics, the problem of determining which of a set of objects is obscured from a particular viewpoint can be formulated as a geometric intersection problem on the projections of the objects onto the viewing plane. And in operations research, the mathematical formulation of many important problems leads naturally to geometric intersection problems.

The obvious solution to the intersection problem is to check each pair of objects to see if they intersect. Since there are about $N^2/2$ pairs of objects, the running time of this algorithm is proportional to N^2. For a few applications, this may not be a problem because other factors limit the number of objects to be processed. However, for many other applications, is not uncommon to deal with hundreds of thousands or even millions of objects. The brute-force N^2 algorithm is obviously inadequate for such applications. In this section, we'll study a general method for determining, in time proportional to *N log N*, whether any two out of a set of *N* objects intersect; this method is based on algorithms presented by M. Shamos and D. Hoey in a seminal 1976 paper.

First, we'll consider an algorithm for returning all intersecting pairs among a set of lines that are constrained to be horizontal or vertical. This makes the problem easier in one sense (horizontal and vertical lines are relatively simple geometric objects), more difficult in another sense (returning all intersecting pairs is more difficult than simply determining whether one such pair exists). The implementation we'll develop applies binary search trees and the interval range-searching program of the previous chapter in a doubly recursive program.

Next, we'll examine the problem of determining whether any two of a set of N lines intersect, with no constraints on the lines. The same general strategy as used for the horizontal-vertical case can be applied. In fact, the same basic idea works for detecting intersections among many other types of geometric objects. However, for lines and other objects, the extension to return all intersecting pairs is somewhat more complicated than for the horizontal-vertical case.

Horizontal and Vertical Lines

To begin, we'll assume that all lines are either horizontal or vertical: the two points defining each line have either equal x coordinates or equal y coordinates, as in the sample sets of lines shown in Figure 27.1. (This is sometimes called *Manhattan geometry* because, Broadway to the contrary notwithstanding, the Manhattan street map consists mostly of horizontal and vertical lines.) Constraining lines to be horizontal or vertical is certainly a severe restriction, but this is far from a "toy" problem. Indeed, this restriction is often imposed in a particular application: for example, very large-scale integrated circuits are typically designed under this constraint. In the figure on the right, the lines are relatively short, as is typical in many applications, though one can usually count on encountering a few very long lines.

The general plan of the algorithm to find an intersection in such sets of lines is to imagine a horizontal scan line sweeping from bottom to top. Projected onto this scan line, vertical lines are points, and horizontal lines are intervals: as the scan line proceeds from bottom to top, points (representing vertical lines) appear and disappear, and horizontal lines are encountered periodically. An intersection is found when a horizontal line is encountered representing an interval on the scan line that contains a point representing a vertical line. Meeting the point means that the vertical line intersects the scan line, and the horizontal line lies on the scan line, so the horizontal and vertical lines must intersect. In this way, the two-dimensional problem of finding an intersecting pair of lines is reduced to the one-dimensional range-searching problem of the previous chapter.

Of course, it is not necessary actually to "sweep" a horizontal line all the way up through the set of lines; since we need to take action only when endpoints of the lines are encountered, we can begin by sorting the lines according to their y coordinate, then processing the lines in that order. If the bottom endpoint of a vertical line is encountered, we add the x coordinate of that line to the binary search tree (here called the x-tree); it' the top endpoint of a vertical line is encountered, we delete that line from the tree; and if a horizontal line is encountered,

we do an interval range search using its two x coordinates. As we'll see, some care is required to handle equal coordinates among line endpoints (by now the reader should be accustomed to encountering such difficulties in geometric algorithms).

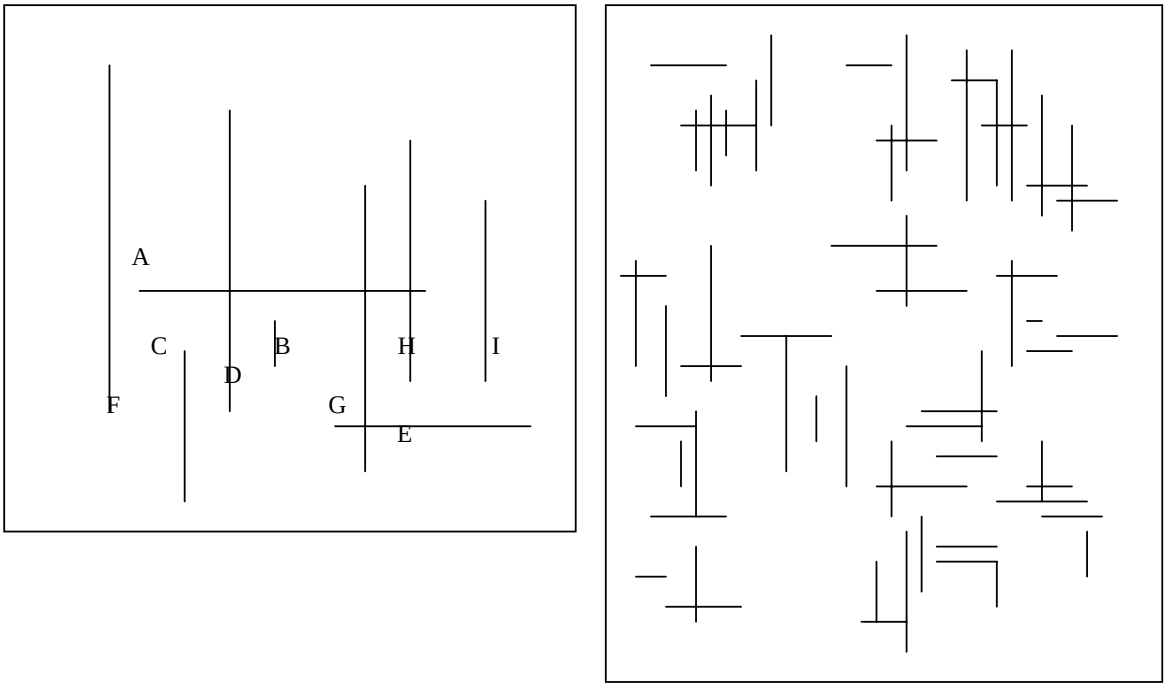


Figure 27.1 Two line intersection problems (Manhattan)

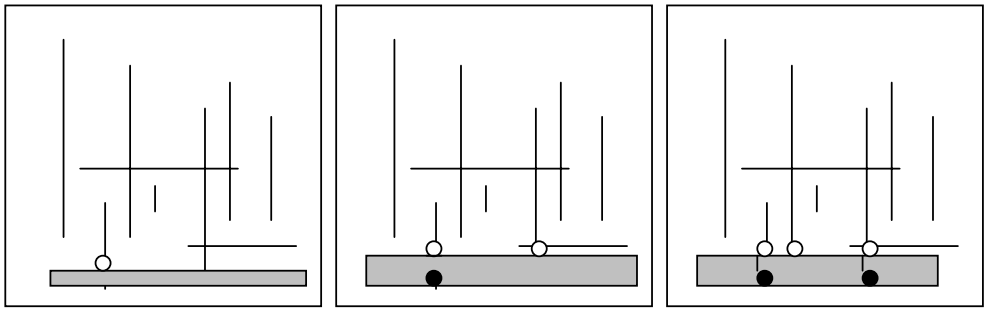


Figure 27.2 Scanning for intersections: initial steps.

Figure 27.2 shows the first few steps of scanning to find the intersections in the example on the left in Figure 27. 1. The scan starts at the point with the lowest y coordinate, the lower endpoint of C. Then E is encountered, then D. The rest of the process is shown in Figure 27.3: the next line encountered is the horizontal line G, which is tested for intersection with C, D, and E (the vertical lines that intersect the scan line).

To implement the scan, we need only sort the line endpoints by their y coordinates. For our example, this gives the list

C E D G I B F C H B A I E D H F

Each vertical line appears twice in this list, each horizontal line appears once. For the purposes of the line intersection algorithm, this sorted list can be thought of as a sequence of insert (vertical lines when the bottom endpoint is encountered), *delete* (vertical lines when the top endpoint is encountered), and r*ange* (for the endpoints of horizontal lines) commands. All of these "commands" are simply calls on the standard binary tree routines from Chapters 14 and 26, using x coordinates as keys.

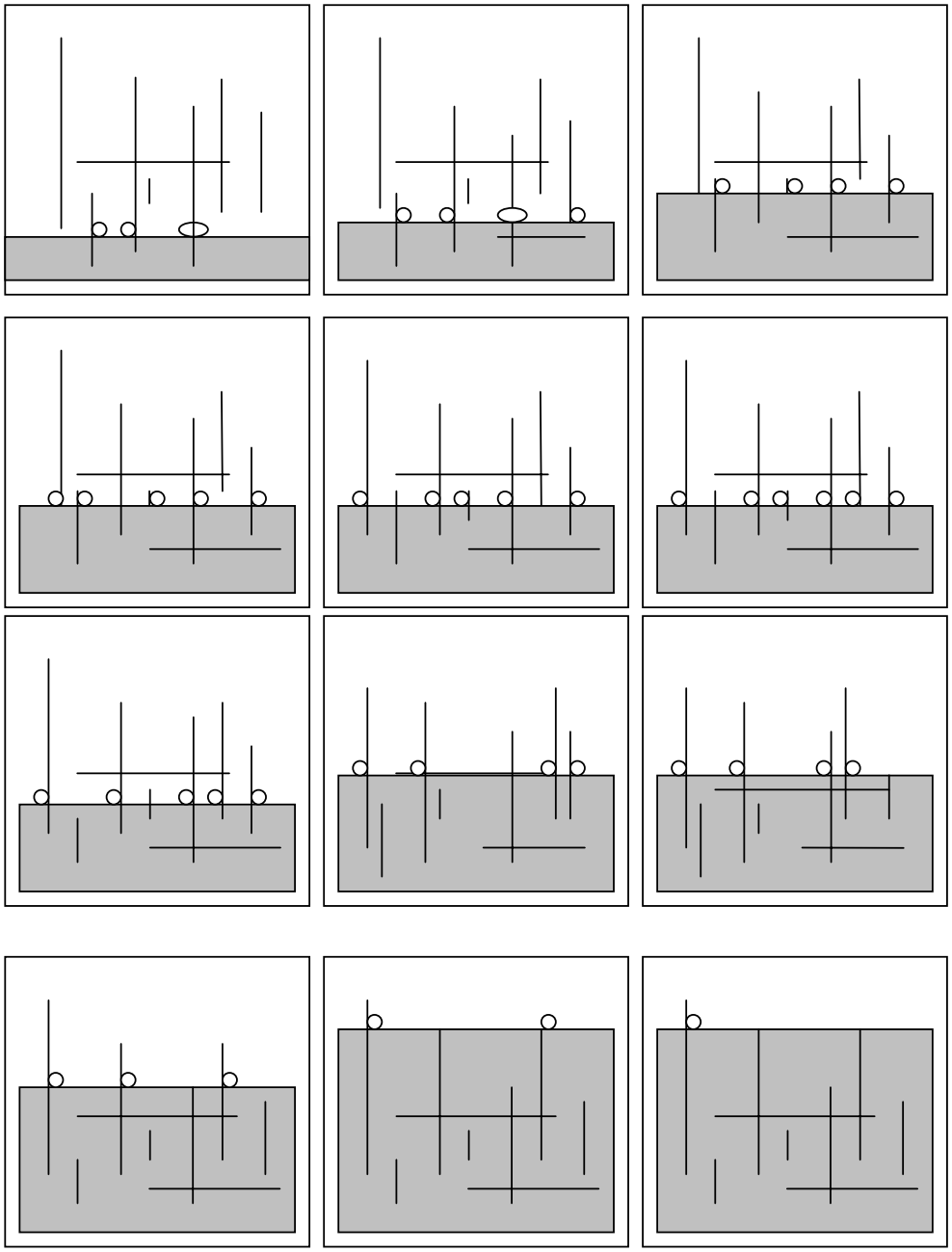
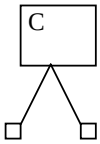
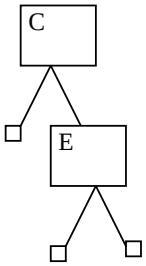
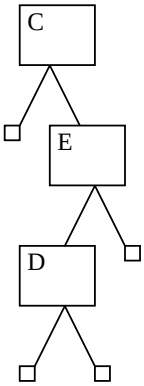
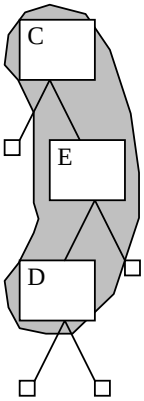


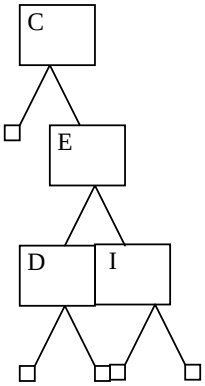
Figure 27.3 Scanning for intersections: compression of process.

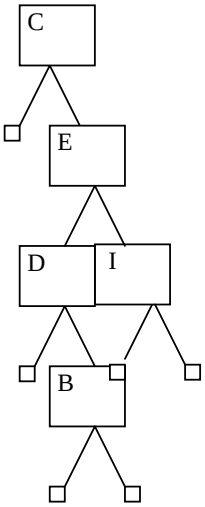

C

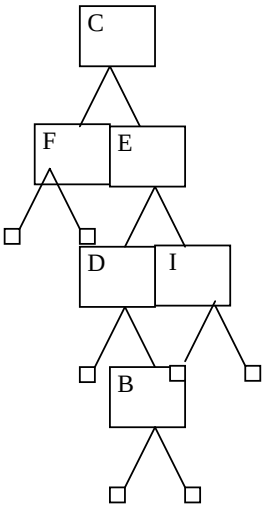

C
E

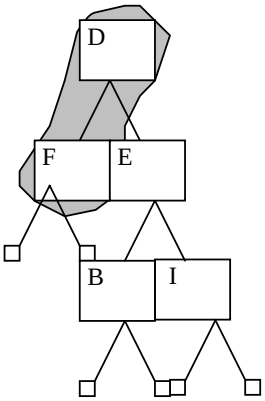

C
E
D

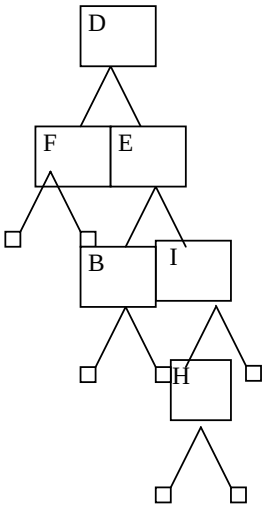

C
E
D


C
E
D
I


C
E
D
I
B


C
F
E
D
I
B


D
F
E
B
I


D
F
E
B
I
H

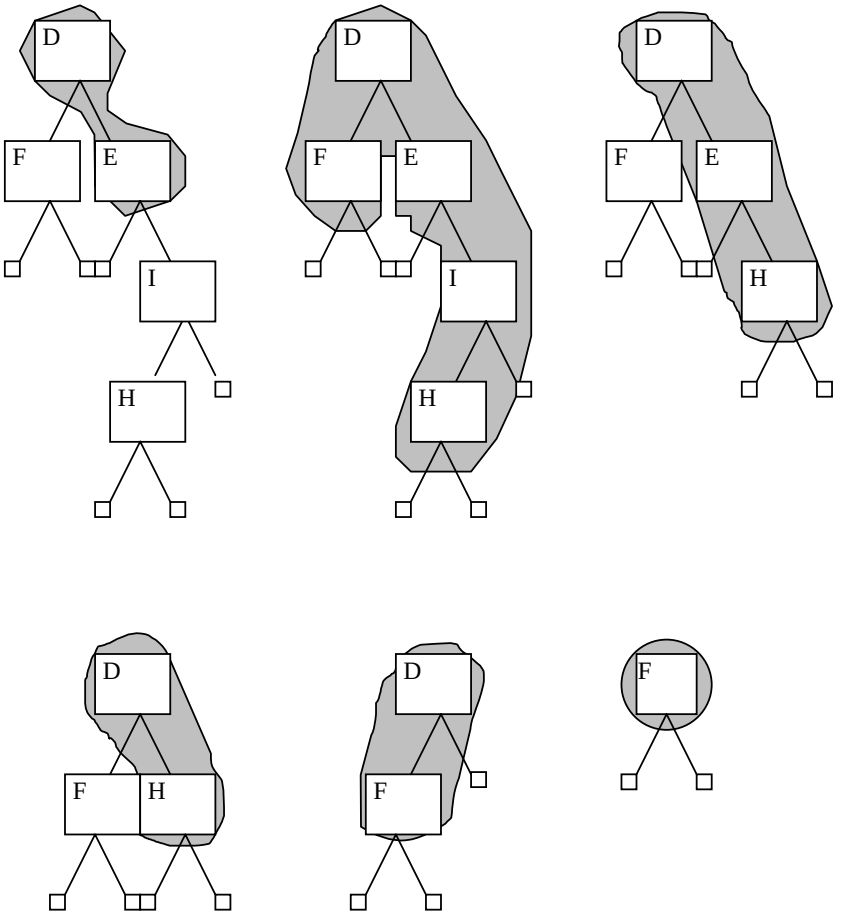


Figure 27.4 Data structure during scan: constructing the x-tree

Figure 27.4 shows the x -tree construction process during the scan. Each node in the tree corresponds to a vertical line, but the key used during the construction of the tree is the x -coordinate. Since E is to the right of C, it is in C's right subtree, etc. The first line of Figure 27.4 corresponds to Figure 27.2 and the rest to Figure 27.3.

When a horizontal line is encountered, it is used to make a range search in the tree: all vertical lines in the range represented by the horizontal line correspond to intersections. In our example, the intersection between E and G is discovered, then 1, B, and F are inserted. Then C is deleted, H inserted, and B deleted. At this point, A is encountered, and a range search for the interval defined by A is performed. This search discovers the intersections between A and D, E, and H. Next, the upper endpoints of 1, E, D, and F are encountered and deleted, leading back to the empty tree.

Implementation

The first step in the implementation is to sort the line endpoints on their y coordinates. But since binary trees will be used to maintain the status of vertical lines with respect to the horizontal scan line, they may as well be used for the initial y sort! Specifically, we will use two "indirect" binary trees on the line set, one with header node *hy* and one with header node hx. *The* y tree will contain all the line endpoints, to be processed in order one at a time; the x tree will contain the lines that intersect the current horizontal scan line. We begin by initializing both hx and hy with 0 keys and pointers to a dummy external node z, as in *treeinitialize in* Chapter 14. Then the *hy* tree is constructed by inserting both y coordinates from vertical lines and the y coordinate of horizontal lines into the binary search tree with header node *hy,* as follows:

```
-----------------------------------------------------------
  procedure buildytree;
      var xl.yl , x2,y2: integer;
      begin
      hy:=bstinitialize; N:=0;
      repeat
      N:=N+l;
          read(xl.yl,x2.y2); if eoln then readln;
          lines[N].p1.x:=xl; lines[N].pl.y:=yl;
          lines[N].p2.x:=x2; lines[N].p2.y:=y2;
           bstinsert(N.yl,hy);
           if y2 < >y 1 then bstinsert (N, y2,  hy);
      until eof;
  end;
-----------------------------------------------------------------------
```

This program reads in groups of four numbers that specify lines and puts them into the lines array and into the binary search tree on the y coordinate. The standard *bstinsert* routine from Chapter 14 is used, with the y coordinates as keys, and indices into the array of lines as the *info* field. For our example set of lines, the tree shown in Figure 27.5 is constructed.

Now, the sort on y is effected by a recursive inorder tree traversal routine (see Chapters 4 and 14). We visit the nodes in increasing y order by visiting all the nodes in the left subtree of the *hy* tree, then visiting the root, then visiting all the nodes in the right subtree of the hy tree. At the same time, we maintain a separate tree (rooted at *hx)* as described above, to simulate the operation of passing through a horizontal scan line:

```
-----------------------------------------------------------------------------------------------------------------------------------------
                    procedure scan (next: link);
                    var t, xl, x2, yl, y2: integer;
                            int: internal;
                    begin
                    if next< >z then
                    begin
                    scan (next|.l);



                        xl:=lines[next|.info].pl.x; yl:=lines[next|.info].pl.y;
                        x2:=lines[next|.info].p2.x; y2:=lines[next|.info].p2.y;
                        if x2<xl then begin t:=x2; x2:=xl; xl:=t end;
                        if y2<yl then begin t:=y2; y2:=yl; yl:=t end;
                        if next|.key=yl then bstinsert(next|.info, xl, hx);
                         if next|.key=y2 then
                              begin
                                  bstdelete (next|. info, x1, hx);
                              int.xl:=xl; int.x2:=x2;
                              bstrange (hx|.r. int);
                              end;
                             scan (next|. r)
                             end
                         end;
```

From the description above, it is rather straightforward to put together the code at the point where each node is "visited". First, the coordinates of the endpoint of the corresponding line are fetched from the lines array, indexed by the *info* field of the node. Then the *key* field in the node is compared against these coordinates to determine whether this node corresponds to the upper or the lower endpoint of the line: if it is the lower endpoint, it is inserted into the *hx* tree, and if it is the upper endpoint, it is deleted from the hx tree and a range search is performed. The implementation differs slightly from this description in that horizontal lines are actually inserted into the hx tree, then immediately deleted, and a range search for a one-point interval is performed for vertical lines. This makes the code properly handle the case of overlapping vertical lines, which are considered to "intersect."

This approach of intermixed application of recursive procedures operating on the x and y coordinates is quite important in geometric algorithms. Another example of this is the 2D tree algorithm of the previous chapter, and we'll see yet another example in the next chapter.

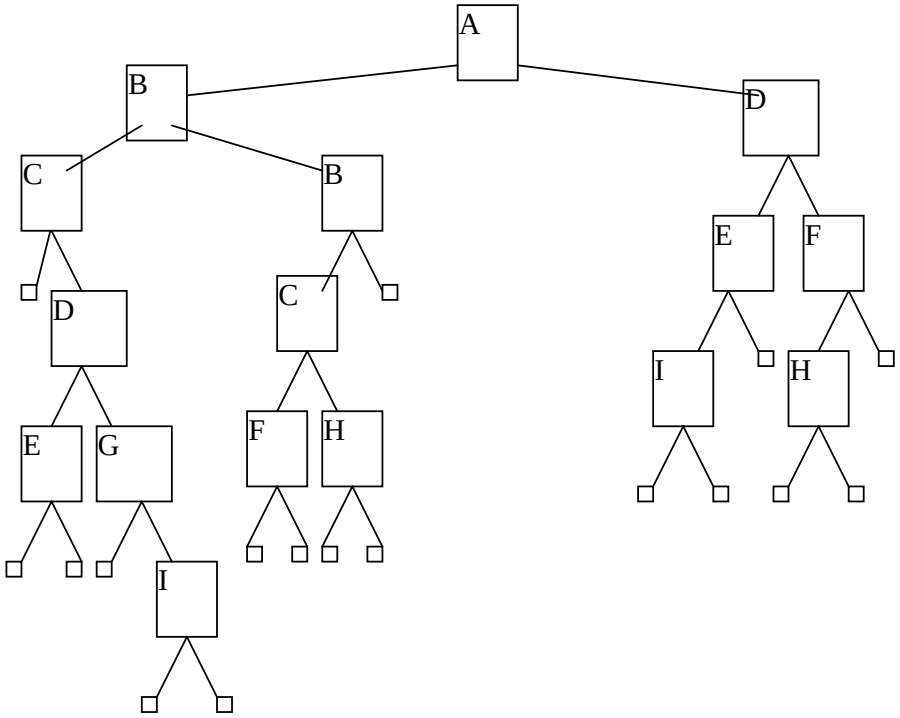


Figure 27.5 Sorting for scan using the y-tree.

Property 27.1 All *i tei-sec-tions among N horizontal and veticalr lines can be* fou*nd in time proportional to N log N +I, where I is the number of intersections.*

The tree manipulation operations take time proportional to logN on the average (if balanced trees were used, a logn worst case could be guaranteed), but the time spent in *bstrange* also depends on the total number of intersections. In general, the number of intersections can be quite large. For example, if we have N/2 horizontal lines and N12 vertical lines arranged in a crosshatch pattern, then the number of intersections is proportional to N^2.

As with range searching, if it is known in advance that the number of intersections is very large, then some brute-force approach should be used. Typically, applications involve a "needle-in-haystack" kind of situation where a large set of lines is to be checked for a few possible intersections.

General Line Intersection

When lines of arbitrary slope are allowed, the situation can become more complicated, as illustrated in Figure 27.6. First, the various line orientations possible make it necessary to test explicitly whether certain pairs of lines intersect-we can't get by with a simple interval range test. Second, the ordering relationship **between** lines for the binary tree **is** more complicated than before, since it depends on the current y range of interest. Third, any intersections that do occur add new "interesting" y values that are likely to be different from the set of y values we get from the line endpoints.

It turns out that these problems can be handled in an algorithm with the same basic structure as given above. To simplify the discussion, we'll consider an algorithm for detecting whether or not there exists an intersecting pair in a set of *N* lines, and then we'll discuss how it can be extended to return all intersections.

As before, we first sort on *y* to divide the space into strips within which no line endpoints appear. Just as before, we proceed through the sorted list of points, adding each line to a binary search tree when its bottom point is encountered and deleting it when its top point is encountered. Just as before, the binary tree gives the order in which the lines appear in the horizontal "strip" between two consecutive y values. For example, in the strip between the bottom endpoint of D and the top endpoint of B in Figure 27.6, the lines should appear in the order F

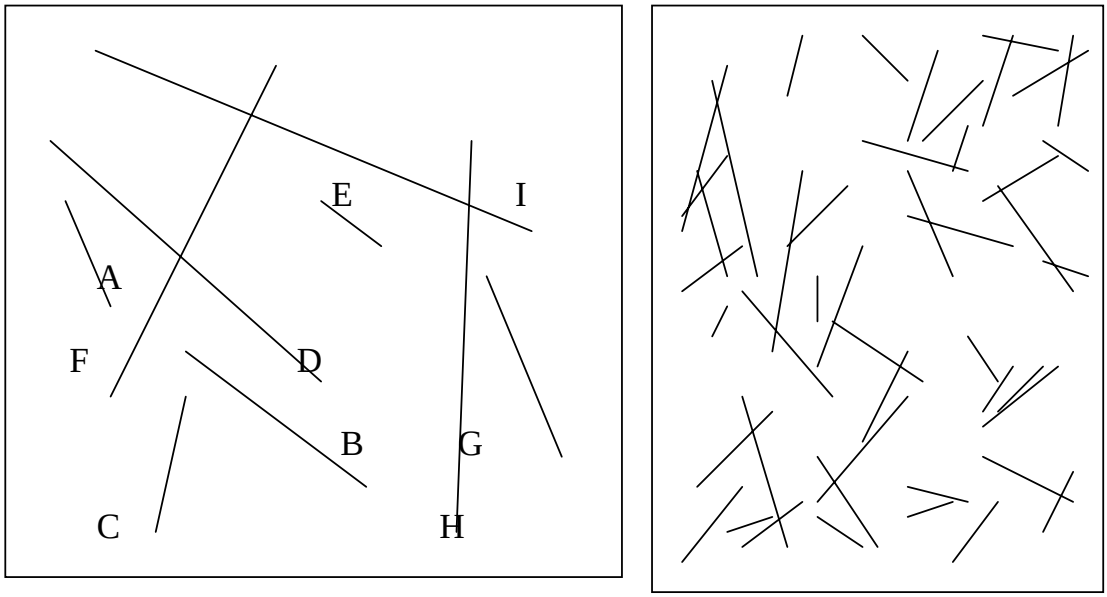


Figure 27.6 Two general line intersection problems.

B D H G. We assume that there are no intersections within the current horizontal strip of interest: our goal is to maintain this tree structure and use it to help find the first intersection.

To build the tree, we can't simply use x coordinates from line endpoints as keys (doing this would put B and D in the wrong order in the example above, for instance). Instead, we use a more general ordering relationship: a line x is defined to be to the right of a line *y* if both endpoints of x are on the same side of y as a point infinitely far to the right, *or if y* is to the left of x, with "left" defined analogously. Thus, in the diagram above, B is to the right of A and B is to the right of C (since C is to the left of B). If *x* is neither to the left nor to the right of y, then they must intersect. This generalized "line comparison" operation can be implemented using the *ccw* procedure of Chapter 24. Except for the use of this function whenever a comparison is needed, the standard binary search tree procedures (even balanced trees, if desired) can be used. Figure 27.7 shows the manipulation of the tree for our example between the time line C is encountered and the time line D is encountered. Each "comparison" performed during the treemanipulation procedures is actually a line-intersection test: if the binary search tree procedure cah't decide to go right or left, then the two lines in question must intersect, and we're finished.

But this is not the whole story, because this generalized comparison operation is not *transitive*. In the example above, F is to the left of B (because B is to the right of F) and B is to the left of D, but F is *not* to the left of D. It is essential to note this, because the binary tree deletion procedure assumes that the comparison operation is transitive: when B is deleted from the last tree in the above sequence, the tree shown in Figure 27.7 is formed without any explicit comparison of F and D. For our intersection-testing algorithm to work correctly, we must test explicitly that comparisons are valid each time we change the tree structure. Specifically, every time we make the left link of node x point to node y, we explicitly test that the line corresponding to x is to the left of the line corresponding to y, according to the above definition, and similarly for the right. Of course, this comparison could result in the detection of an intersection, as it does in our example.

In summary, to test for an intersection among a set of *N* lines, we use the program above, but we remove the call to *range* and extend the binary tree routines to use the generalized comparison as described above. If there is no intersection, we'll start with a null tree and end with a null tree without finding any incomparable lines. If there is an intersection, then the two lines that intersect must be compared against each other at some point during the scanning process and the intersection will be discovered.

Once we've found an intersection, however, we can't simply press on and hope to find others, because the two lines that intersect should swap places in the ordering directly after the point of intersection. One way to handle this issue would be to use a priority queue instead of a binary tree for the y sort: initially put lines on the priority queue according to the y coordinates of their endpoints, then work the scan line up by successively taking the smallest y coordinate from the priority queue and doing a binary tree insert or delete as above. When an intersection is found, new entries are added to the priority queue for each line, using the intersection point as the lower endpoint for each.

Another way to find all intersections, which is appropriate if not too many are expected, is simply to remove one of the intersecting lines when an intersection is found. Then after the scan is completed, we know that all

intersecting pairs must involve one of those lines, and we can use a brute-force method to enumerate all the intersections.

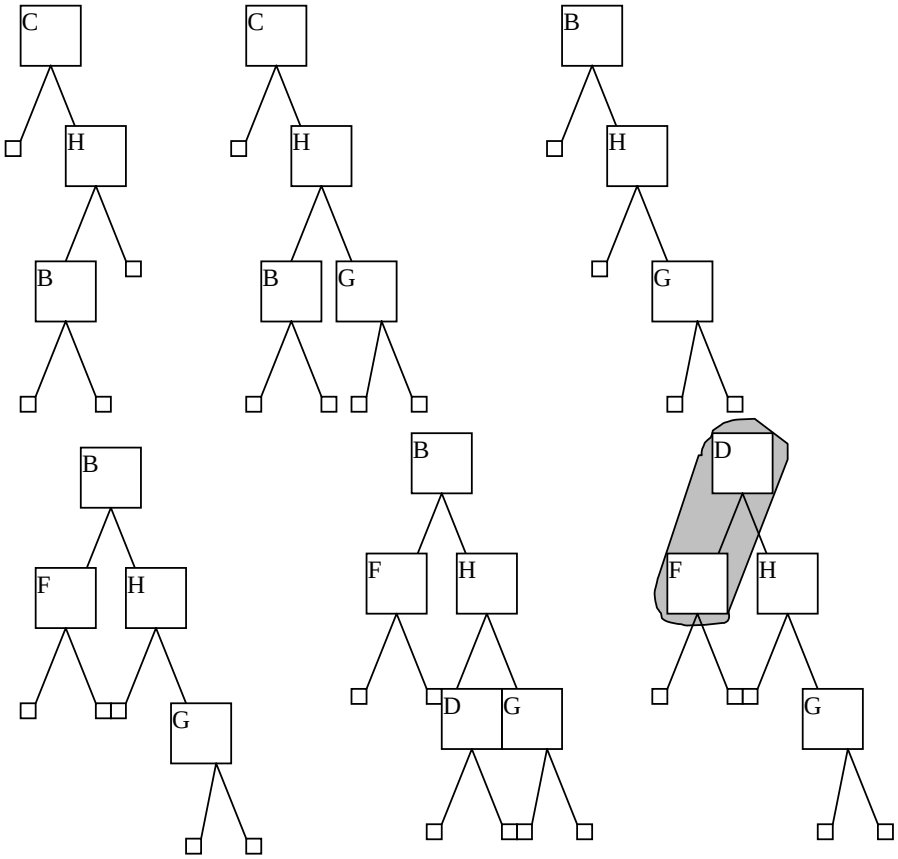


Figure 27.7 Data structure (x-tree) for general problem.

Property 27.2 A*ll intersections among N lines can be found in time proportional to N log N + I, where I is the numbet- of intersections.*

This follows directly from the discussion above.

An interesting feature of the above procedure is that it can be adapted just by changing the generalized comparison procedure to test for the existence of an intersecting pair among a set of more general geometric shapes. For example, if we implement a procedure that compares two rectangles whose edges are horizontal and vertical according to the trivial rule that rectangle x is to the left of rectangle *y* if the right edge of x is to the left of the left edge of y, then we can use the above method to test for intersection among a set of such rectangles. For circles, we can use the x coordinates of the centers for the ordering and explicitly test for intersection (for example, compare the distance between the centers to the sum of the radii). Again, if this comparison procedure is used in the above method, we have an algorithm for testing for intersection among a set of circles. The problem of returning all intersections in such cases is much more complicated, though the brute-force method mentioned in the previous paragraph will always work if few intersections are expected. Another approach that will suffice for many applications is simply to consider complicated objects as sets of lines and use the line-intersection procedure.