28 Closest-Point Problems

Geometric problems involving points on the plane usually involve implicit or explicit treatment of distances between the points. For example, a very natural problem which arises in many applications is the *nearest-neighbor problem:* find the point among a set of given points closest to a given new point. This seems to involve checking the distance from the given point to each point in the set, but much better solutions are possible. In this section we'll look at some other distance problems, a prototype algorithm, and a fundamental geometric structure called the *Voronoi diagram* that can be used effectively for a variety of such problems in the plane. Our approach will be to describe a general method for solving closest point problems through careful consideration of a prototype implementation for a simple problem.

Some of the problems that we consider in this chapter are similar to the rangesearching problems of Chapter 26, and the grid and 2D tree methods developed there are suitable for solving the nearest-neighbor and other problems. The fundamental shortcoming of those methods, however, is that they rely on randomness in the point set: they have bad worst-case performance. Our aim in this chapter is to examine another general approach that has guaranteed good performance for many problems, no matter what the input. Some of the methods are too complicated for us to examine a full implementation, and they involve sufficient overhead that the simpler methods may do better when the point set is not large or when it is sufficiently well dispersed. However, the study of methods with good worst-case performance will uncover some fundamental properties of point sets that should be understood even if simpler methods are more suitable in specific situations.

The general approach we'll examine provides yet another example of the use of doubly recursive procedures to intertwine processing along the two coordinate directions. The two previous methods we've seen of this type (kD trees and line intersection) have been based on binary search trees; here the method is a "combine and conquer" method based on mergesort.

Closest-Pair Problem

The closest-pair problem is to find the two points that are closest together among a set of points. This problem is related to the nearest-neighbor problem; though it is not as widely applicable, it will serve us well as a prototype closest-point problem in that it can be solved with an algorithm whose general recursive structure is appropriate for other problems.

It would seem necessary to examine the distances between all pairs of points to find the smallest such distance: for N points this would mean a running time proportional to N^2 . However, it turns out that we can use sorting to get by with examining only about *N logN* distances between points in the worst case (far fewer on the average) and get a worst-case running time proportional to *N logN* (far better on the average). In this section, we'll examine such an algorithm in detail.

The algorithm we'll use is based on a straightforward "divide-and-conquer" strategy. The idea is to sort the points on one coordinate, say the x coordinate, then use that ordering to divide the points in half. The closest pair in the whole set is either the closest pair in one of the halves or the closest pair with one member in each half. The interesting case, of course, is when the closest pair crosses the dividing line: the closest pair in each half can obviously be found by using recursive calls, but how can all the pairs on either side of the dividing line be checked efficiently?

Since the only information we seek is the closest pair of the point set, we need examine only points within distance min of the dividing line, where min is the smaller of the distances between the closest pairs found in the two halves. By itself, however, this observation isn't enough help in the worst case, since there could be many pairs of points very close to the dividing line; for example, all the points in each half could be lined up right next to the dividing line.

To handle such situations, it seems necessary to sort the points on y. Then we can limit the number of distance computations involving each point as follows: proceeding through the points in increasing y order, check if each point is inside the vertical strip containing all points in the plane within min of the dividing line. For each such point, compute the distance between it and any point also in the strip whose y coordinate is less than the y coordinate of the current point, but not more than min less. The fact that the distance between all pairs of points in each half is at least min means that only a few points are likely to be checked.

In the small set of points on the left in Figure 28.1, the imaginary vertical dividing line just to the right of F has eight points to the left, eight points to the right. The closest pair on the left half is AC (or AO), the closest pair on the right is JM. If the points are sorted on y, then the closest pair split by the line is found by checking the pairs HI, Cl, FK (the closest pair in the whole point set), and finally EK. For larger point sets, the band that could contain a closest pair spanning the dividing line is narrower, as shown on the right in Figure 28. 1.

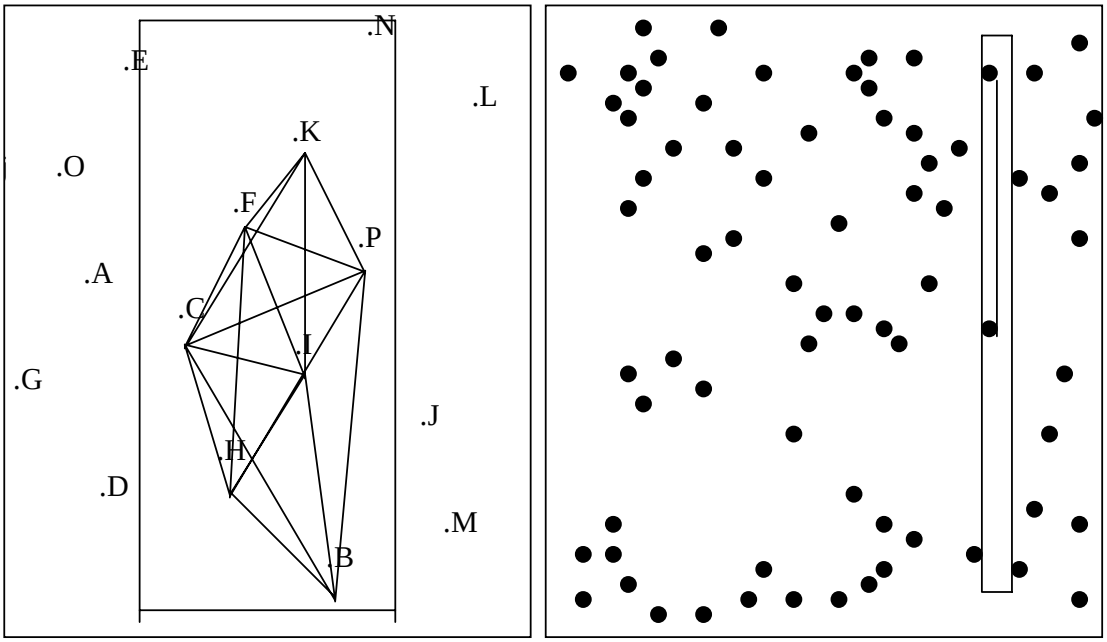


Figure 28.1 Divide-and-conquer approach to find the closest pair.

Though this algorithm is stated simply, some care is required to implement it efficiently: for example, it would be too expensive to sort the points on y within our recursive subroutine. We've seen several algorithms with running times described by the recurrence $C_N = 2C_{N/2}$ + N, which implies that C_N is proportional to *N log N;* if we were to do the full sort on y, then the recurrence would become $C_N = 2C_{N/2} + N \log$ N, which implies that C_N is proportional to $N \log^2 N$ (see Chapter 6). To avoid this, we need to avoid the sort of y.

The solution to this problem is simple, but subtle. The *mergesort* method from Chapter 12 is based on dividing the elements to be sorted exactly as the points are divided above. We have two problems to solve and the same general method to solve them, so we may as well solve them simultaneously! Specifically, we'll write one recursive routine that both sorts on *y and* finds the closest pair. It will do so by splitting the point set in half, then calling itself recursively to sort the two halves on *y* and find the closest pair in each half, then merging to complete the sort on *y* and applying the procedure above to complete the closest-pair computation. In this way, we avoid the cost of doing an extra y sort by intermixing the data movement required for the sort with the data movement required for the closest-pair computation.

For they sort, the split in half could be done in any way, but for the closest-pair computation, it's required that the points in one half all have smaller x coordinates than the points in the other half. This is easily accomplished by sorting on x before doing the division. In fact, we may as well use the same routine to sort on x! Once this general plan is accepted, the implementation is not difficult to understand.

As mentioned above, the implementation will use the recursive sort *and* merge procedures of Chapter 12. 'fhe first step is to modify the list structures to hold points instead of keys, and to modify *merge* to check a global variable *pass to* decide how to do its comparison. If *pass=1,* we should compare the x coordinates of the two points; if *pass=2* we compare the y coordinates of the two points. The implementation of this is straightforward:

```
----------------------------------------------------
function merge (a. b: link): link;
    var c: link; comp: boolean;
    begin
    c:=z;
    repeat
```

```
    if pass=l
        then comp: =a|.p.x < b|.p.x
        else comp:=a|.p.y<b|.p.y;
    if comp
  then begin c|. next:=a; c:=a; a:=a|.next end
   else begin c|.next:=b; c:=b; b:=b|.next end
 until c=z;
 merge:=z|.next; z|.next:=z;
  end;
```

The dummy node z which appears at the end of all lists is initialized to contain a "sentinel" point with artificially high x *and* y coordinates.

To compute distances, we use another simple procedure that checks that the distance between the two points given as arguments is less than the global variable min. If so, it resets min to that distance and saves the points in the global variables *cpl and cp2:*

```
procedure check (p1, p2: point);
    var dist: real;
    begin
    if (pl.y< >z |.p.y) and (p2.y< >z |.p.y) then
    begin
      dist:=sqrt((pl.x-p2.x)* (pl.x-p2.x)+(pl.y-p2.y)* (pl.y-p2.y));
      if dist<min then
          begin min:=dist; cpl:=pl; cp2:=p2 end;
       end;
    end;
```

Thus, the global min always contains the distance between cpl and ep2, the closest pair found so far.

The next step is to modify the recursive *sort* of Chapter 12 also to do the closest-point computation when pass=2, as follows:

```
function sort (c: link; N: integer): link;
        var a. b: link; i: integer;
                middle: real;
          p1, p2, p3, p4: point;
        begin
        if c|.next=z then sort:=c else
                    begin
                     a:=c;
                     for i:= 2 to N div 2 do c:=c|.next;
                      b:=c|.next; c|.next:=z;
                      if pass=2 then middle:=b|.p.x;
                       c:=merge (sort(a, N div 2), sort(b.N- (N div 2)));
                      sort:=c;
                       if pass=2 then
                     begin
                     a:=c; pl:=z|.p; p2:=z|.p; p3:=z|.p; p4:=z|.p;
                     repeat
if abs (a|.p.x - middle) < min then
        begin
         check (a|.p, p1);
         check (a|.p, p2);
          check (a|.p,p3);
          check (a|.p, p4);
         pl:=p2; p2:=p3; p3:=p4; p4:=a|.p
         end;
a: =a |.next
```

```
            until a=z
          end
        end;
      end;
```

If pass=l, this is exactly the recursive mergesort routine of Chapter 12: it returns a linked list containing the points sorted on their x coordinates (because merge has been modified as described above to compare x coordinates on *passl). The* magic of this implementation comes whenpass=2. The program not only sorts on y (because merge has been modified as described above to compare y coordinates *on pass2)* but also completes the closest-point computation, as described in detail below.

First, we sort on x; then we sort on y and find the closest pair by invoking *sort* as follows:

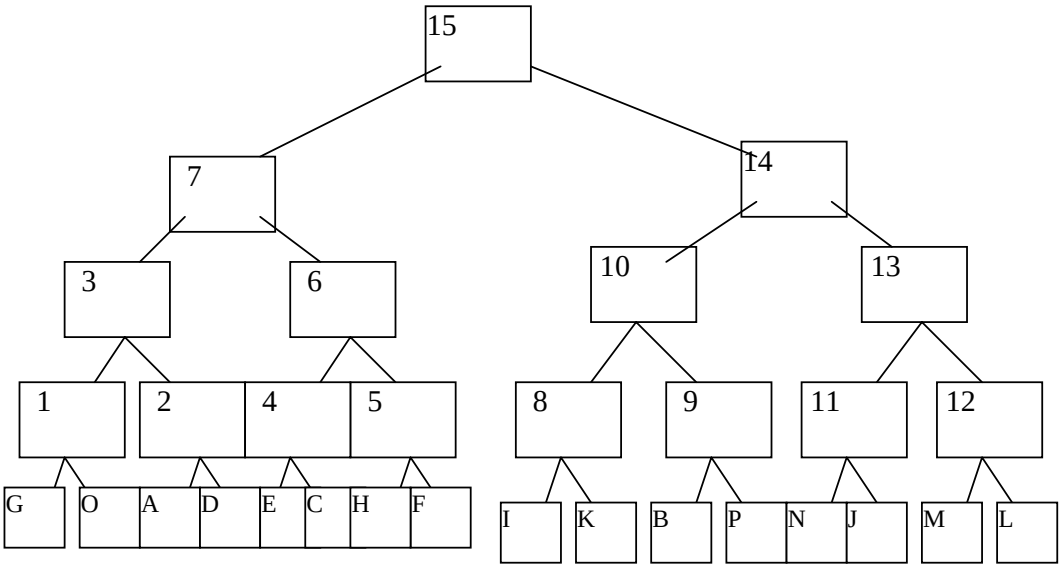


Figure 28.2 Recursive call tree for closest-pair computation

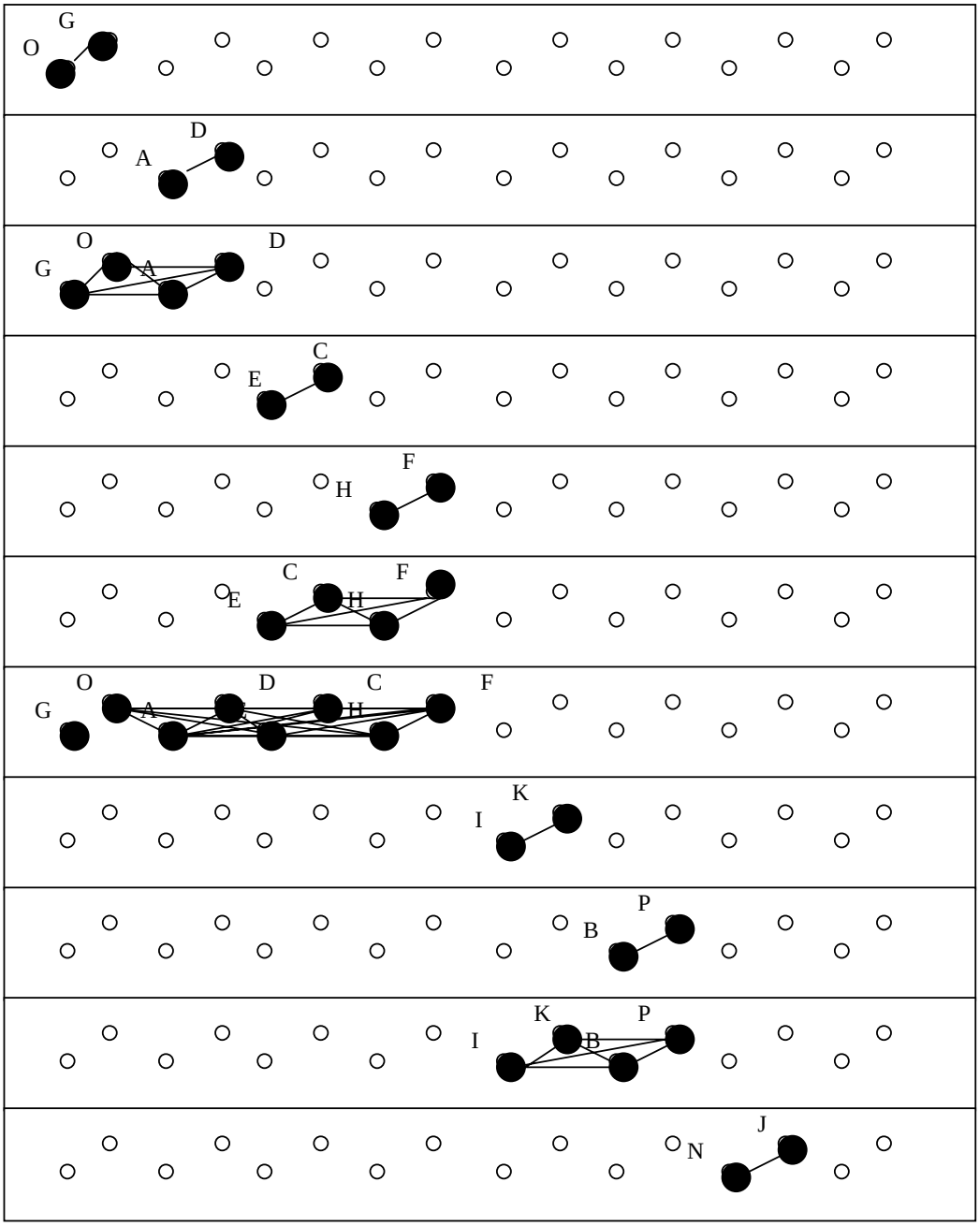
```
new(z); z|.next:=z;
z|.p.x:=maxint; z|.p.y:=maxint;
new(h); h|.next:=readlist;
min:=maxint;
pass:=l; h|.next:=sort(h|.next, N);
pass:=2; h|.next:=sort(h|.next, N);
```

After these calls, the closest pair of points is found in the global variables cpl and cp2, which are managed by the *check* "find the minimum" procedure.

The crux of the implementation is the operation of *sort when pass=2. Before* the recursive calls the points are sorted on x: this ordering is used to divide the points in half and to find the x coordinate of the dividing line. *After* the recursive calls the points are sorted on y and the distance between every pair of points in each half is known to be greater than min. The ordering on y is used to scan the points near the dividing line; the value of min is used to limit the number of points to be tested. Each point within a distance of min of the dividing line is *checked* against each of the previous four points found within a distance of min of the dividing line. This check is guaranteed to find any pair of points closer together than min with one member of the pair on either side of the dividing line. (This is an amusing geometric fact which the reader may wish to verify): We know that points that fall on the same side of the dividing line are spaced by at least min, so the number of points falling in any circle of radius min is limited.)

Figure 28.2 shows the recursive call tree describing the operation of this algorithm on our small set of points. An internal node in this tree represents a vertical line dividing the points in the left and right subtree. The nodes

are numbered in the order in which the vertical lines are tried in the algorithm. This numbering corresponds to a postorder traversal of the tree because the computation involving the dividing line comes *aftei-* the recursive calls in the program, and is simply another way of looking at the order in which merges are done during a recursive mergesort (see Chapter 12).



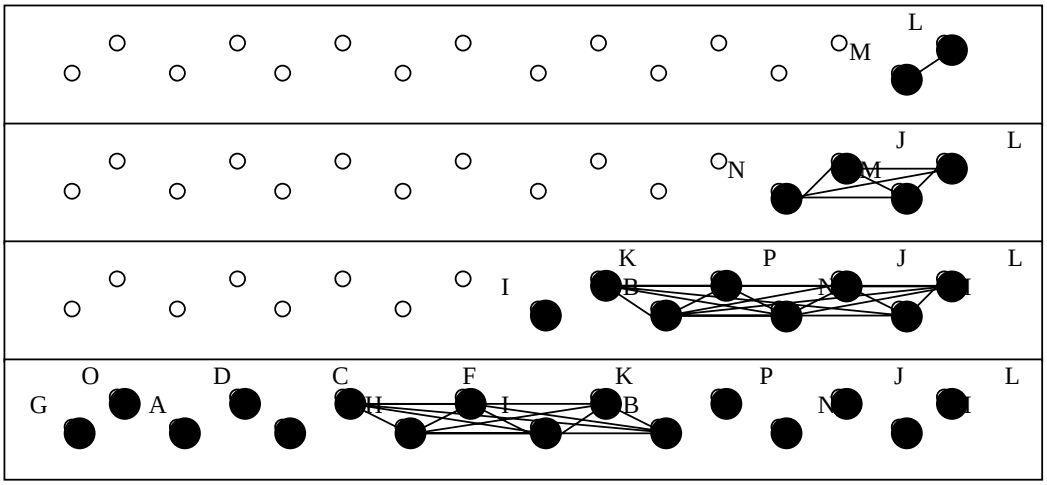


Figure 28.3 Closest-pair computation (x coordinate magnified).

Thus, first the line between G and 0 is tried and the pair GO is retained as the closest so far. Then the line between A and D is tried, but A and D are too far apart to change min. Then the line between 0 and A is tried and the pairs GD GA and OA all are successively closer pairs. It happens for this example that no closer pairs are found until FK, which is the last pair checked for the last dividing line tried.

The careful reader may have noticed that we have not implemented the pure divide-and-conquer algorithm described above-we don't actually compute the closest pair in the two halves, then take the better of the two. Instead, we get the closer of the two closest pairs simply by using a global variable for min during the recursive computation. Each time we find a closer pair, we can consider a narrower vertical strip around the current dividing line, no matter where we are in the recursive computation.

Figure 28.3 shows the process in detail. The x -coordinate in these diagrams is magnified to emphasize the x orientation of the process and to point out parallels with mergesort (see Chapter 12). We start by doing a y-sort on the four leftmost points G 0 A D, by sorting G 0, then sorting A D, then merging. After the merge, the y-sort is complete, and we find the closest pair AO spanning the dividing line. Eventually, the points are sorted on their y-coordinate and the closest pair is computed.

Property28.1 The closest pair in a set of N points can be found in O(NlogN) *steps.*

Essentially, the computation is done in the time it takes to do two mergesorts (one on the x-coordinate, one on the y-coordinate) plus the cost of looking along the dividing line. This cost is also governed by the recurrence $T_N = T_{N/2} + N$ (see Chapter 6).

The general approach we've used here for the closest-pair problem can be used to solve other geometric problems. For example, another question of interest is the *all-nearest-neighbors* problem: for each point we want to find the point nearest to it. This problem can be solved using a program like the one above with extra processing along the dividing line to find, for each point, whether there is a point on the other side closer than its closest point on its own side. Again, the "free" *y* sort is helpful for this computation.

Voronoi Diagrams

The set of all points closer to a given point in a point set than to all other points in the set is an interesting geometric structure cal led the Voronoi po4gon for the point. The union of all the Voronoi polygons for a point set is called its Voronoi diagram. This is the ultimate in closest-point computations: we'll see that most of the problems we face involving distances between points have natural and interesting solutions based on the Voronoi diagram. The diagrams for our sample point sets are shown in Figure 28.4.

The Voronoi polygon for a point is made up of the perpendicular bisectors of the segments linking the point to those points closest to it. Its actual definition is the other way around: the Voronoi polygon is defined to be perimeter of the set of all points in the plane closer to the given point than to any other point in the point set, and each edge on the Voronoi polygon separates a given point from one of the points "closest to" it.

The dual of the Voronoi diagram, shown in Figure 28.5, makes this correspon dence explicit: in the dual, a line is drawn between each point and all the points "closest to" it. This is also called the Delaunay triangulation. Put another way, x and y are connected in the Voronoi dual if their Voronoi polygons have an edge in common.

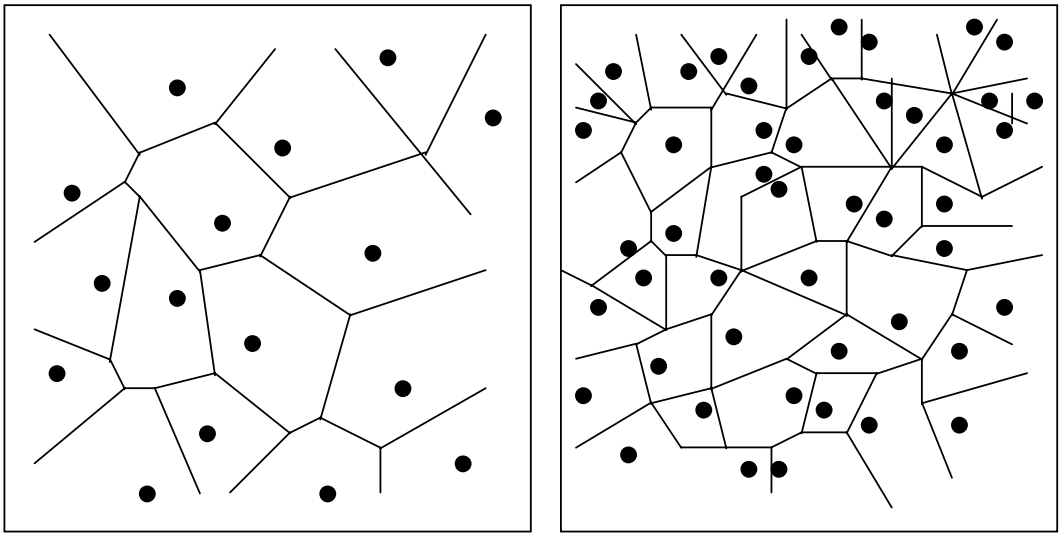


Figure 28.4 Voronoi Diagram.

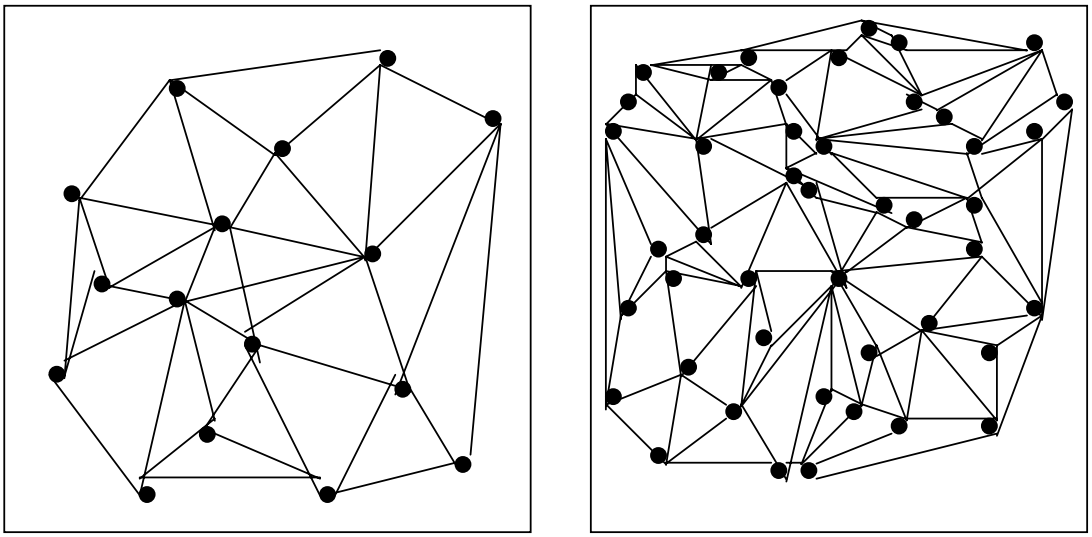


Figure 28.5 Delaunay Triangulation.

The Voronoi diagram and the Delaunay triangulation have many properties that lead to efficient algorithms for closest-point problems. The property that makes these algorithms efficient is that the number of lines in both the diagram and the dual is proportional to a small constant times N. For example, the line connecting the closest pair of points must be in the dual, so the problem of the previous section can be solved by computing the dual and then simply finding the minimum length line among the lines in the dual. Similarly, the line connecting each point to its nearest neighbor must be in the dual, so the all-nearest-neighbors problem reduces directly to finding the dual. The convex hull of the point set is part of the dual, so computing the Voronoi dual is yet another convex hull algorithm. We'll see yet another example in Chapter 31 of a problem which can be solved efficiently by first finding the Voronoi dual.

The defining property of the Voronoi diagram means that it can be used to solve the nearest-neighbor problem: to identify the nearest neighbor in a point set to a given point, we need only find out which Voronoi polygon the point falls in. It is possible to organize the Voronoi polygons in a structure like a 2D tree to allow this search to be done efficiently.

The Voronoi diagram can be computed using an algorithm with the same general structure as the closest-point algorithm above. The points are first sorted on their x coordinate. Then that ordering is used to split the points in half, leading to two recursive calls to find the Voronoi diagram of the point set for each half. At the same time, the points are sorted on y; finally, the two Voronoi diagrams for the two halves are merged together. As before, this merging (done with pass=2) can exploit the fact that the points are sorted on x before the recursive calls and that they are sorted on y and that the Voronoi diagrams for the two halves have been built after the recursive calls. However, even with these aids, the merge is quite a complicated task, and presentation of a full implementation would be beyond the scope of this book.

The Voronoi diagram is certainly the natural structure for closest-point problems, and understanding the characteristics of a problem in terms of the Voronoi diagram or its dual is certainly a worthwhile exercise. However, for many particular problems, a direct implementation based on the general schema given in this chapter may be suitable. This schema is powerful enough to compute the Voronoi diagram, so it is powerful enough for algorithms based on the Voronoi diagram, and it may admit to simpler, more efficient code, as we saw for the closest-pair problem.