25 *Finding the Convex Hull*

Often, when we have a large number of points to process, we're interested in the boundaries of the point set. People looking at a diagram of a set of points plotted in the plane, have little trouble distinguishing those on the "inside" of the point set from those lying on the edge. This distinction is a fundamental property of point sets; in this chapter we'll see how it can be precisely characterized by looking at algorithms for separating out the "natural boundary" points.

The mathematical way to describe the natural boundary of a point set depends on a geometric property called *convexity*. This is a simple concept that the reader may have encountered before: a *convex polygon* has the property that any line connecting any two points inside the polygon must itself lie entirely inside the polygon. For example, the "simple closed polygon" that we computed in the previous chapter is decidedly nonconvex; on the other hand, any triangle or rectangle is convex.

Now, the mathematical name for the natural boundary of a point set is the *convex hull*. The convex hull of a set of points in the plane is defined to be the smallest convex polygon containing them all. Equivalently, the convex hull is the shortest path surrounding the points. An obvious property of the convex hull that is easy to prove is that the vertices of the convex polygon defining the hull are points from the original point set. Given N points, some of them form a convex polygon within which all the others are contained. The problem is to find those points. Many algorithms have been developed to find the convex hull; in this chapter we'll examine some of the important ones.

Figure 25.1 shows our sample sets of points for Figure 24.1 and their convex hulls. There are 8 points on the hull of the small set and 15 points on the hull of the large set. In general, the convex hull can contain as few as three points (if the three points form a large triangle containing all the others) or as many as all the points (if they fall on a convex polygon, then the points comprise their own convex hull). The number of points on the convex hull of a "random" point set

falls somewhere in between these extremes, as we will see below. Some algorithms work well when there are many points on the convex hull; others work better when there are only a few.

A fundamental property of the convex hull is that any line outside the hull, when moved in any direction towards the hull, hits it at one of its vertex points. (This is an alternate way to define the hull: it is the subset of points from the point set that could be hit by a line moving in at some angle from infinity.) In particular, it's easy to find a few points guaranteed to be on the hull by applying this rule for horizontal and vertical lines: the points with the smallest and largest x and y coordinates are all on the convex hull. This fact is used as the starting point for the algorithms we consider.

Rules of the Game

The input to an algorithm for finding the convex hull is of course an array of points; we can use the *point* type defined in the previous chapter. The output is a polygon, also represented as an array of points with the property that tracing through the points in the order in which they appear in the array traces the outline of the polygon. On reflection, this might appear to require an extra ordering condition on the computation of the convex hull (why not just return the points on the hull in any order?), but output in the ordered form is obviously more useful, and it has been shown that the unordered computation is no easier to do. For all of the algorithms that we consider, it is convenient to do the computation *in place: the* array used for the original point set is also used to hold the output. The algorithms simply rearrange the points in the original array so that the convex hull appears in the first M positions, in order.

From the description above, it may be clear that computing the convex hull is closely related to sorting. In fact, a convex hull algorithm can be used to sort, in the following way. Given N numbers to sort, turn them into points (in polar coordinates) by treating the numbers as angles (suitably normalized) with a fixed radius for each point. The convex hull of this point set is an N-gon containing all of the points. Now, since the output must be ordered in the order in which the points appear on this polygon, it can be used to find the sorted order of the original values (remember that the input was unordered). This is not a formal proof that computing the convex hull is no easier than sorting, because, for example, the cost of the trigonometric functions required to convert the numbers into points on the polygon must be considered. Comparing convex hull algorithms (which involve trigonometric operations) to sorting algorithms (which involve comparisons between keys) is a bit like comparing apples to oranges, but even so it has been shown that any convex hull algorithm must require about *N log N* operations, the same as sorting (even though the operations allowed are likely to be quite different). It is helpful to view finding

the convex hull of a set of points as a kind of "twodimentional sort," since frequent parallels to sorting algorithms arise in the study of algorithms for finding the convex hull.

In fact, the algorithms we'll study show that finding the convex hull is no harder than sorting either: there are several algorithms that run in time proportional to *N log N* in the worst case. Many of the algorithms tend to use even less time on actual point sets, because their running time depends on how the points are distributed and on the number of points on the hull.

As with all geometric algorithms, we have to pay some attention to degenerate cases that are likely to occur in the input. For example, what is the convex hull of a set of points all of which fall on the same line segment? Depending upon the application, this could be all the points or just the two extreme points, or perhaps any set including the two extreme points would do. Though this seems an extreme example, it would not be unusual for more than two points to fall on one of the line

segments defining the hull of a set of points. In the algorithms below, we won't insist that points failing on a hull edge be included, since this generally involves more work (though we will indicate how this could be done when appropriate). On the other hand, we won't insist that they be omitted either, since this condition could be tested afterwards if desired.

Package-Wrapping

The most natural convex hull algorithm, which parallels how a human would draw the convex hull of a set of points, is a systematic way to "wrap up" the set of points. Starting with some point guaranteed to be on the convex hull (say the one with the smallest y coordinate), take a horizontal ray in the positive direction and

"sweep" it upward until hitting another point; this point must be on the hull. Then anchor at that point and continue "sweeping" until hitting another point, etc., until the "package" is fully "wrapped" (the beginning point is included again). Figure 25.2 shows how the hull is discovered in this way for our sample set of points. Point B has the minimum y coordinate and is the starting point. Then M is the first point hit by the sweeping ray, then L, etc.

Of course, we don't actually need to sweep through all possible angles; we just do a standard find-the-minimum computation to find the point that would be hit next. For each point to be included on the hull, we need to examine each point not yet included on the hull. Thus, the method is quite similar to selection sorting we successively choose the "best" of the points not yet chosen, using a brute-force search for the minimum. The actual data movement involved is depicted in Figure 25.3: the *M*th line of the table shows the situation after the *M* th point is added to the hull.

The following program finds the convex hull of an array *p [I..N]* of points, represented as described at the beginning of Chapter 24. The basis for this implementation is the function *theta (p l,p2: point)* developed in the previous chapter, which can be thought of as returning the angle between *pl, p2* and the horizontal (though it actually returns a more easily computed number with the same ordering properties). Otherwise, the implementation follows directly from the discussion above. The array position *p [N+1*] is used to hold a sentinel.

```
function wrap: integer;

  var I, min, M: integer;
        minangle, v: real;
        t: point;
  begin
  min:=l;
  for i:=2 to N do

     if p[i].y<p[min].y then min:=i;
  M:=O; p[N+l]:=p[min]; minangle:=O.O; repeat

     M:=M+l; t:=p[M]; p[M]:=p[min]; p[min]:=t; min:=N+I; v:=minangie; minangle:=360.0; for i:=M+l
     to N+l do
        if theta (p[M],p[i])>v then

           if theta(p[M].p[i])<minangle then

  begin min:=i; minangle:=theta(p[M],p[min]) end;
  until min=N+l;
```

wrap:=M;
end;

First, the point with the smallest y coordinate is found and copied into *p [N+1]* in order to stop the loop, as described below. The variable M is maintained as the number of points so far included on the hull, and v is the current value of the "sweep" angle (the angle from the horizontal to the line between *p*[M-1] and *p[M]).* The **repeat** loop puts the last point found into the hull by exchanging it with the *M*th point, and uses the *theta* function from the previous chapter to compute the angle from the horizontal made by the line between that point and each of the points not yet included on the hull, searching for the one whose angle is smallest among those with angles greater than v. The loop stops when the first point (actually the copy of the first point put into *p*[N+l]) is encountered again.

This program may or may not return points which fall on a convex hull edge. This situation is encountered when more than one point has the same *theta* value with *p [M]* during the execution of the algorithm; the implementation above returns the point first encountered among such points, even though there may be others closer to *p*[*M*]. When it is important to find points falling on convex hull edges, we can achieve this by changing *theta* to take into account the distance between the points given as its arguments and assign the closer point a smaller value when two points have the same angle.

The major disadvantage of package-wrapping is that in the worst case, when all the points fall on the convex hull, the running time is proportional to *N* 2 (like that of selection sort). On the other hand, the method has the attractive feature that it generalizes to three (or more) dimensions. The convex hull of a set of points in

k-dimensional space is the minimal convex polytope containing them all, where a convex polytope is defined by the property that any line connecting two points inside must itself lie inside. For example, the convex hull of a set of points in 3-space is a convex three-dimensional object with flat faces. It can be found by "sweeping" a plane until the hull is hit, then "folding" faces of the plane, anchoring on different lines on the boundary of the hull, until the "package" is "wrapped." (Like many geometric algorithms, it is rather easier to explain this generalization than to implement it!)

The Graham Scan

The next method we'll examine, invented by R. L. Graham in 1972, is interesting because most of the computation involved is for sorting: the algorithm includes a sort followed by a relatively inexpensive (though not immediately obvious) computation. The algorithm starts by constructing a simple closed polygon from the points using the method of the previous chapter: sort the points using as keys the *theta* function values corresponding to the angle from the horizontal made from the line connecting each point with an 'anchor' point *p*[1] (the one with the lowest y coordinate), so that tracing p [*1*], p [*2*], ..., *p* [*N*], p [1] gives a closed polygon. For our example set of points, we get the simple closed polygon of the previous chapter. Note that *p* [*N*], *p* [*1*], and *p* [*2*] are consecutive points on the hull; by sorting, we've essentially run the first iteration of the package-wrapping procedure (in both directions). Computation of the convex hull is completed by proceeding around, trying to place each point on the hull and eliminating previously placed points that couldn't possibly be on the hull. For our example, we consider the points in the order B M J L N P K F I E C 0 A H G D; the first few steps are shown in Figure 25.4. At the beginning, we know because of the sort that B and M are on the hull. When J is encountered, the algorithm includes it on the trial hull for the first three points.

Then, when L is encountered, the algorithm finds out that J couldn't be on the hull (since, for example, it falls inside the triangle BML).

In general, testing which points to eliminate is not difficult. After each point has been added, we assume that we have eliminated enough points that what we have traced out so far could be part of the convex hull on the basis of the points so far seen. As we trace around, we expect to turn left at each hull vertex. If a new point causes us to turn *right*, then the point just added must be eliminated, since there exists a convex polygon containing it. Specifically, the test for eliminating a point uses the *ccw* procedure of the previous chapter, as follows. Suppose we have determined that *p* [*l..M*] are on the partial hull determined on the basis of examining *p* [*I.. i* - I]. When we come to examine a new point *p* [*i*], we eliminate *p* [*M*] from the hull if *ccw (p* [*M*],*p* [*M- 1*]*p* [*i*]) is nonnegative. Otherwise, *p* [*M*] could still be on the hull, so we don't eliminate it.

Figure 25.5 shows the completion of this process on our sample set of points. The situation as each new point is encountered is diagrammed: each new point is added to the partial hull so far constructed and is then used as a

"witness" for the elimination of (zero or more) points previously considered. After L, N, and P are added to the hull, P is eliminated when K is considered (since NPK is a right turn), then F and I are added, leading to the consideration of E. At this point, I must be eliminated because FIE is a right turn, then F and K must be eliminated because KFE and NKE are right turns.. Thus more than one point can be eliminated during the "backup" procedure, perhaps several. Continuing in this way, the algorithm finally arrives back at B.

The initial sort guarantees that each point is considered in turn as a possible hull point, because all points considered earlier have a smaller *theta* value. Each line that survives the "eliminations" has the property that all points so far considered are on the same side of it, so that when we get back to *p* [*N*], which also must be on the hull because of the sort, we have the complete convex hull of all the points.

As with the package-wrapping method, points on a hull edge may or may not be included, though there are two distinct situations that can arise with collinear points. First, if there are two points collinear with *p* [*1*], then, as above, the sort using *theta* may or may not get them in order along their common line. Points out of order in this situation will be eliminated during the scan. Second, collinear points along the trial hull can arise (and not be eliminated).

Once the basic method is understood, the implementation is straightforward, though there are a number of details to attend to. First, the point with the maximum x value among all points with minimum y value is exchanged with *p [1*]. Next, *shellsort* is used to rearrange the points (any comparison-based sorting routine would do), modified as necessary to compare two points using their theta values with *p* [*l*]. After the sort, *p* [*N*] is copied into *p*[*O*] to serve as a sentinel in case *p*[*3*] is not on the hull. Finally, the scan described above is performed. The following program finds the convex hull of the point set *p*[*1..N*]

```
function grahamscan: integer;
    var i, j, min, M: integer;
          1: line; t: point;
    begin
    min:=l;
    for i:=2 to N do
    if p [i ].y <p [min ].y then min:=i;
    for i:=l to N do
      if (p [i ].y=p [min ].y) and (p [i ].x>p [min ].x) then min:=i; t:=p [1 ]; p [1 ]:=p [min ]; p [min ]:=t;
    shellsort;
    p [0]:=p [N];
    M:=3;
    for i:=4 to N do
      begin
      while ccw(p[M],p[M-1],p[I])>=O do M:=M-1; M:=M+l;
      t:=p[M]; p[M]:=p[i]; p[i]:=t;
      end;
  grahamscan:=M;
  end;
```

The loop maintains a partial hull in *p*[*l..M*], as described above. For each new *i* value considered, *M* is decremented if necessary to eliminate points from the partial hull and thenp[i] is exchanged with *p*[*M+l*] to (tentatively) add it to the partial hull. Figure 25.6 shows the contents of the *p* array each time a new point is considered for our example.

The reader may wish to check why it is necessary for the *min* computation to find the point with the lowest *x* coordinate among all points with the lowest *y* coordinate, the canonical form described in Chapter 24. As discussed above, another subtle point is to consider the effect of the fact that collinear points lead to equal *theta* values, and may not be sorted in the order in which they appear on the line, as one might have hoped.

One reason that this method is interesting to study is that it is a simple form of *backtracking*, the algorithm design technique of "try something, and if it doesn't work then try something else" that we'll revisit in Chapter 44.

Interior Elimination

Almost any convex hull method can be vastly improved by a simple techique which quickly disposes of most points. The general idea is simple: pick four points known to be on the hull, then throw out everything inside the quadrilateral

formed by those four points. This leaves many fewer points to be considered by, say, the Graham scan or the package wrapping technique.

The four points known to be on the hull should be chosen with an eye towards any information available about the input points. Generally, it is best to adapt the choice of points to the distribution of the input. For example, if all *x* and *y* values within certain ranges are equally likely (a rectangular distribution), then choosing four points by scanning in from the corners (find the four points with the largest and smallest sum and difference of the two coordinates) turns out to eliminate nearly all the points. Figure 25.7 shows that this technique eliminates most points not on the hull in our two example point sets.

In an implementation of the interior elimination method, the "inner loop" for random point sets is the test of whether or not a given point falls within the test quadrilateral. This can be speeded up somewhat by using a rectangle with edges parallel to the *x* and *y* axes. The largest such rectangle which fits in the quadrilateral described above is easy to find from the coordinates of the four points defining the quadrilateral. Using this rectangle will eliminate fewer points from the interior, but the speed of the test more than offsets this loss.

Performance Issues

As mentioned in the previous chapter, geometric algorithms are somewhat harder to analyze than algorithms in some of the other areas we've studied because the input (and the output) is more difficult to characterize. It often doesn't make sense to speak of "random" point sets: for example, as *N* gets large, the convex hull of points drawn from a rectangular distribution is extremely likely to be very close to the rectangle defining the distribution. The algorithms we've looked at depend on different properties of the point set distribution and are thus incomparable in practice, because to compare them analytically would require an understanding of very complicated interactions between little-understood properties of point sets. On the other hand, we can say some things about the performance of the algorithms that will help choosing one for a particular application.

Property 25.1 *After the sort, the Graham scan is a linear-time process.*

A moment's reflection is necessary to convince oneself that this is true, since there is a "loop-within-a-loop" in the program. However, it is easy to see that no point is "eliminated" more than once, so the code within that double loop is iterated fewer than *N* times. The total time required to find the convex hull using this method is *O(N logN)*, but the "inner loop" of the method is the sort itself, which can be made efficient using techniques of Chapters 8-12.

Property 25.2 *If there are M vertices on the hull, then the "package-wrapping" technique requires about MN steps.*

First, we must compute $N - 1$ angles to find the minimum, then $N - 2$ to find the next, then $N - 3$, etc., so the total number of angle computations is $(N - 1) + (N - 2) + \ldots + (N - M + 1)$, which is exactly equal to $MN - M(M - 1)/2$. To compare this analytically with the Graham scan would require a formula for *M* in terms of *N*, a difficult problem in stochastic geometry. For a circular distribution (and some others) the answer is that *M* is $O(N^{1/3})$, and for values of *N* which are not large $N^{1/3}$ is comparable to log *N* (which is the expected value for a rectangular distribution), so this method will compete very favorably with the Graham scan. Of course, the N^2 worst case should always be taken into consideration.

Property 25.3 *The interior elimination method is linear, on the average.*

Full mathematical analysis of this method would require even more sophisticated stochastic geometry than above, but the general result is the same as that given by intuition: almost all the points fall inside the quadrilateral and are discarded-the number of points left over is $O(\sqrt{N})$. This is true even if the rectangle is used as described above. This makes the average running time of the whole convex hull algorithm proportional to *N*, since most points are examined only once (when they are thrown out). On the average, it doesn't matter much which method is afterwards, since so few points are likely to be left. However, to protect against the worst case (when all points are on the hull), it is prudent to use the Graham scan. This gives an algorithm which is almost sure to run in linear time in practice and is guaranteed to run in time proportional to *N logN*.

The average-case result of Property 25.3 holds only for randomly distributed points in a rectangle, and in the worst case nothing is eliminated by the interior elimination method. However, for other distributions or for point sets with unknown properties, this method is still recommended because the cost is low (a linear scan through the

points, with a few simple tests) and the possible savings is high (most of the points can be easily eliminated). The method also extends to higher dimensions.

It is possible to devise a recursive version of the interior elimination method: find extreme points and remove points on the interior of the defined quadrilateral as above, but then consider the remaining points as partitioned into subproblems which can be solved independently, using the same method. This recursive technique is similar to the Quicksort-like *select* procedure for selection discussed in Chapter 12. Like that procedure, it is vulnerable to an N 2 worst-case running time. For example, if all the original points are on the convex hull, then no points are thrown out in the recursive step. Like *select*, the running time is linear on the average (though it is not easy to prove this). But because so many points are eliminated in the first step, it is not likely to be worth the trouble to do further recursive decomposition in any practical application.