# Chapter 18

# Geometric Sweeping

## 18.1 Introduction

In geometric algorithms, the main objects considered are usually points, line segments, polygons and others in two-dimensional, three-dimensional and higher dimensional spaces. Sometimes, a solution to a problem calls for "sweeping" over the given input objects to collect information in order to find a feasible solution. This technique is called *plane sweep* in the two-dimensional plane and *space sweep* in the three-dimensional space. In its simplest form, a vertical line sweeps from left to right in the plane stopping at each object, say a point, starting from the leftmost object to the rightmost object. We illustrate the method in connection with a simple problem in computational geometry.

**Definition 18.1** Let $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ be two points in the plane. $p_2$ is said to dominate $p_1$, denoted by $p_1 \prec p_2$, if $x_1 \leq x_2$ and $y_1 \leq y_2$.

**Definition 18.2** Let $S$ be a set of points in the plane. A point $p \in S$ is a *maximal point* or a *maximum* if there does not exist a point $q \in S$ such that $p \neq q$ and $p \prec q$.

The following problem has a simple algorithm, which is a good example of a geometric sweeping algorithm.

MAXIMAL POINTS: Given a set $S$ of $n$ points in the plane, determine the maximal points in $S$.

This problem can easily be solved as follows. First, we sort all the points in $S$ in nonincreasing order of their $x$-coordinates. The rightmost point (the

one with maximum $x$-value) is clearly a maximum. The algorithm *sweeps* the points from right to left and for each point $p$ it determines whether it is dominated on the $y$-coordinate by any of the previously scanned points. The algorithm is given as Algorithm MAXIMA.

**Algorithm 18.1** MAXIMA
**Input:** A set $S$ of $n$ points in the plane.
**Output:** The set $M$ of maximal points in $S$.

1. Let $A$ be The points in $S$ sorted in nonincreasing order of their $x$-coordinates. If two points have the same $x$-coordinate then the one with larger $y$-coordinate appears first in the ordering.
2. $M \leftarrow \{A[1]\}$
3. $maxy \leftarrow y$-coordinate of $A[1]$
4. **for** $j \leftarrow 2$ **to** $n$
5. $\quad (x, y) \leftarrow A[j]$
6. $\quad$ **if** $y > maxy$ **then**
7. $\quad\quad M \leftarrow M \cup \{A[j]\}$
8. $\quad\quad maxy \leftarrow y$
9. $\quad$ **end if**
10. **end for**

Figure 18.1 illustrates the behavior of the algorithm on a set of points. As shown in the figure, the set of maxima $\{a, b, c, d\}$ forms a staircase. Note that, for example, $e$ is dominated by $a$ only, whereas $f$ is dominated by both $a$ and $b$, and $g$ is dominated by $c$ only.

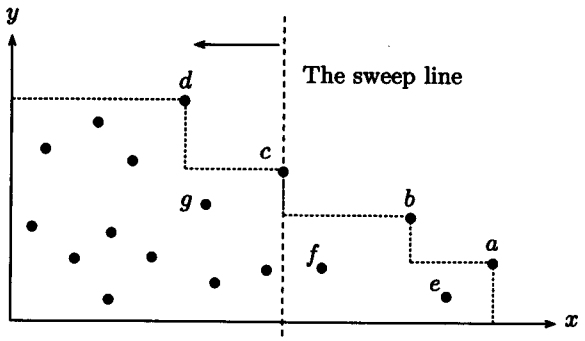


Fig. 18.1 A set of points with their maxima.



It is easy to see that the running time of the Algorithm MAXIMA is dominated by the sorting step, and hence is $O(n \log n)$.

The above example reveals the two basic components of a plane sweep algorithm. First, there is the *event point schedule*, which is a sequence of the $x$-coordinates ordered from right to left. These points define the "stopping" positions of the sweeping line, which is in this case a vertical line. Unlike the case in the previous example, in some plane sweep algorithms, the event point schedule may be updated dynamically, and thus data structures that are more complex than a simple array or a queue may be required for efficient implementation.

The other component in the plane sweep method is the *sweep line status*. This is an appropriate description of the geometric objects at the sweeping line. In the above example, the sweep line status consists of a "description" of the most recent maximal point detected. This description is simply the value of its $y$-coordinate. In other geometric algorithms, the sweep line status may require storing the relevant information needed in the form of a stack, a queue, a heap, etc.

## 18.2 Geometric Preliminaries

In this section we present the definitions of some of the fundamental concepts in computational geometry that will be used in this chapter. Most of these definitions are within the framework of the two-dimensional space; their generalization to higher dimensions is straightforward. A *point* $p$ is represented by a pair of coordinates $(x, y)$. A *line segment* is represented by two points called its *endpoints*. If $p$ and $q$ are two distinct points, we denote by $\overline{pq}$ the line segment whose endpoints are $p$ and $q$. A *polygonal path* $\pi$ is a sequence of points $p_1, p_2, \ldots, p_n$ such that $\overline{p_i p_{i+1}}$ is a line segment for $1 \leq i \leq n-1$. If $p_1 = p_n$, then $\pi$ (together with the closed region bounded by $\pi$) is called a *polygon*. In this case, the points $p_i, 1 \leq i \leq n$, are called the *vertices* of the polygon, and the line segments $\overline{p_1 p_2}, \overline{p_2 p_3}, \ldots, \overline{p_{n-1} p_n}$ are called its *edges*. A polygon can conveniently be represented using a circular linked list to store its vertices. In some algorithms, it is represented by a circular doubly linked list. As defined above, technically, a polygon refers to the closed connected region called the *interior* of the polygon plus the *boundary* that is defined by the closed polygonal path. However, we will mostly write "polygon" to mean its boundary. A polygon $P$ is called *simple*

if no two of its edges intersect except at its vertices; otherwise it is *nonsimple*. Figure 18.2 shows two polygons, one is simple and the other is not.

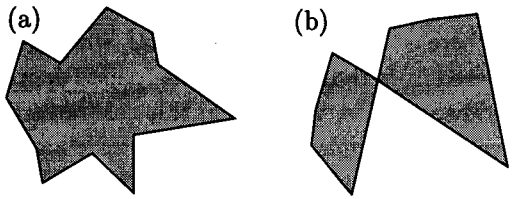


Fig. 18.2 (a) A simple polygon. (b) A nonsimple polygon.

Henceforth, it will be assumed that a polygon is simple unless otherwise stated, and hence the modifier "simple" will be dropped. A polygon $P$ is said to be *convex* if the line segment connecting any two points in $P$ lies entirely inside $P$. Figure 18.3 shows two polygons, one is convex and the other is not.

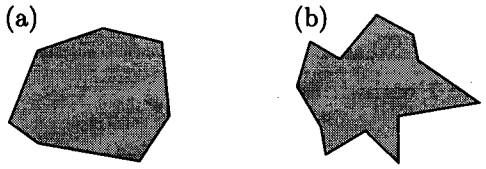


Fig. 18.3 (a) A convex polygon. (b) A nonconvex polygon.

Let $S$ be a set of points in the plane. The *convex hull* of $S$, denoted by $CH(S)$, is defined as the smallest convex polygon enclosing all the points in $S$. The vertices of $CH(S)$ are called *hull vertices* and are also referred to as the *extreme points* of $S$.

Let $u = (x_1, y_1)$, $v = (x_2, y_2)$ and $w = (x_3, y_3)$. The *signed area* of the triangle formed by these three points is half the determinant

$$D = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}.$$

$D$ is positive if $u, v, w, u$ form a counterclockwise cycle, in which case we say that the path $u, v, w$ is a *left turn*. It is negative if $u, v, w, u$ form a clockwise cycle, in which case we say that the path $u, v, w$ is a *right turn*.



(See Fig. 18.4). $D = 0$ if and only if the three points are collinear, i.e., lie on the same line.

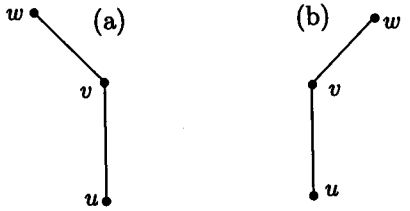


Fig. 18.4 (a) A left turn. (b) A right turn.

## 18.3 Computing the Intersections of Line Segments

In this section, we consider the following problem. Given a set $L = \{l_1, l_2, \ldots, l_n\}$ of $n$ line segments in the plane, find the set of of points at which they intersect. We will assume that no line segment is vertical and no three line segments intersect at the same point. Removing these assumptions will only make the algorithm more complicated.

Let $l_i$ and $l_j$ be any two line segments in $L$. If $l_i$ and $l_j$ intersect the vertical line with $x$-coordinate $x$ at two distinct points $p_i$ and $p_j$, respectively, then we say that $l_i$ *is above* $l_j$ *at* $x$, denoted by $l_i >_x l_j$, if $p_i$ lies above $p_j$ on the vertical line with $x$-coordinate $x$. The relation $>_x$ defines a total order on the set of all line segments intersecting the vertical line with $x$-coordinate $x$. Thus, in Fig. 18.5, we have

$$l_2 >_x l_1, l_2 >_x l_3, l_3 >_y l_2 \text{ and } l_4 >_z l_3.$$

The algorithm starts by sorting the $2n$ endpoints of the $n$ line segments in nondecreasing order of their $x$-coordinates. Throughout the algorithm, a vertical line sweeps all endpoints of the line segments and their intersections from left to right. Starting from the empty relation, each time an endpoint or an intersection point is encountered, the order relation changes. Specifically, the order relation changes whenever one of the following "events" occurs while the line is swept from left to right.

(1) When the left endpoint of a line segment is encountered.

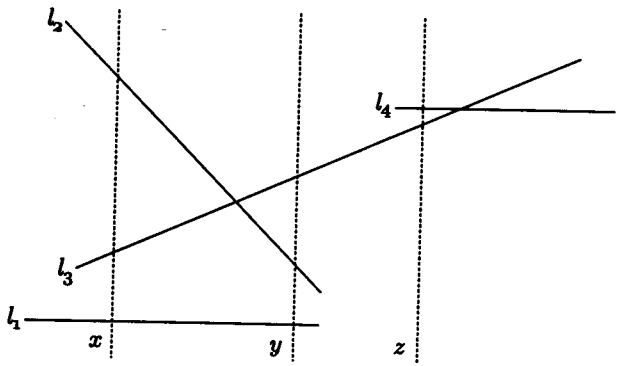


Fig. 18.5 Illustration of the relation $>_x$.

(2) When the right endpoint of a line segment is encountered.
(3) When the intersection point of two line segments is encountered.

The sweep line status $S$ is completely described by the order relation $>_x$. As to the event point schedule $E$, it includes the sorted endpoints plus the intersections of the line segments, which are added dynamically while the line is swept from left to right.

The actions taken by the algorithm on each event are as follows.

(1) When the left endpoint of a line segment $l$ is encountered, $l$ is added to the order relation. If there is a line segment $l_1$ immediately above $l$ and $l$ and $l_1$ intersect, then their intersection point is inserted into the event point schedule $E$. Similarly, if there is a line segment $l_2$ immediately below $l$ and $l$ and $l_2$ intersect, then their intersection point is inserted into $E$.

(2) When the right endpoint $p$ of a line segment $l$ is encountered, $l$ is removed from the order relation. In this case, the two line segments $l_1$ and $l_2$ immediately above and below $l$ are tested for a possible intersection at a point $q$ *to the right of* $p$. If this is the case, $q$ is inserted into $E$.

(3) When the intersection point $p$ of two line segments is encountered, their relative order in the relation is reversed. Thus, if $l_1 >_x l_2$ to the left of their intersection, the order relation is modified so that $l_2 >_x l_1$. Let $l_3$ and $l_4$ be the two line segments immediately above and below the intersection point $p$, respectively (see Fig. 18.6). In other words, $l_3$ is above $l_2$ and $l_4$ is below $l_1$ *to the right of* the intersection point (see Fig. 18.6). In this case, we



check for the possibility of $l_2$ intersecting with $l_3$ and $l_1$ intersecting with $l_4$. As before, we insert their intersection points into $E$, if any.

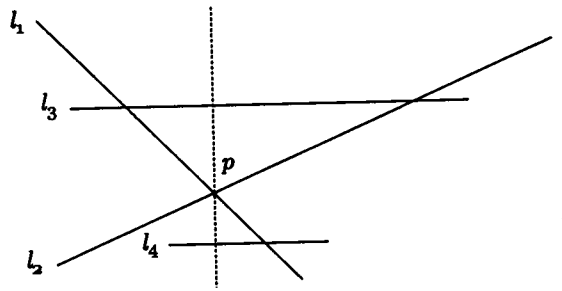


Fig. 18.6 Reversing the order of two line segments at an intersection point.

It remains to specify the data structures needed to implement the event point schedule and the sweep line status. To implement the event point schedule $E$, we need a data structure that supports the following operations:

- *insert*$(p, E)$: Insert point $p$ into $E$.
- *delete-min*$(E)$: Return the point with minimum $x$-coordinate and delete it from $E$.

These two operations are clearly supported by the heap data structure in $O(\log n)$ time. Thus, $E$ is implemented as a heap that initially contains the $2n$ sorted points. Each time the sweep line is to be moved to the right, the point with minimum $x$-coordinate is extracted. As explained above, when the algorithm detects an intersection point $p$, it inserts $p$ into $E$.

As we have seen in the description of the algorithm above, the sweep line status $S$ must support the following operations:

- *insert*$(l, S)$: Insert line segment $l$ into $S$.
- *delete*$(l, S)$: Delete line segment $l$ from $S$.
- above$(l, S)$: Return the line segment immediately above $l$.
- below$(l, S)$: Return the line segment immediately below $l$.

A data structure known as a *dictionary* supports each of the above operations in $O(\log n)$ time. Note that above$(l, S)$ or below$(l, S)$ may not exist; a simple test (which is not included in the algorithm) is needed to handle these two cases.

A more precise description of the algorithm is given in Algorithm INTERSECTIONSLS. In the algorithm, Procedure process($p$) inserts $p$ into $E$ and outputs $p$.

**Algorithm 18.2** INTERSECTIONSLS
**Input:** A set $L = \{l_1, l_2, \ldots, l_n\}$ of $n$ line segments in the plane.
**Output:** The intersection points of the line segments in $L$.

1. Sort the endpoints in nondecreasing order of their $x$-coordinates and insert them into a heap $E$ (the event point schedule).
2. **while** $E$ is not empty
3. $p \leftarrow$ *delete-min*($E$)
4. **if** $p$ is a left endpoint **then**
5. let $l$ be the line segment whose left endpoint is $p$
6. *insert*($l, S$)
7. $l_1 \leftarrow$ above($l, S$)
8. $l_2 \leftarrow$ below($l, S$)
9. **if** $l$ intersects $l_1$ at point $q_1$ **then** process($q_1$)
10. **if** $l$ intersects $l_2$ at point $q_2$ **then** process($q_2$)
11. **else if** $p$ is a right endpoint **then**
12. let $l$ be the line segment whose right endpoint is $p$
13. $l_1 \leftarrow$ above($l, S$)
14. $l_2 \leftarrow$ below($l, S$)
15. *delete*($l, S$)
16. **if** $l_1$ intersects $l_2$ at point $q$ *to the right of* $p$ **then** process($q$)
17. **else** {$p$ is an intersection point}
18. Let the two intersecting line segments at $p$ be $l_1$ and $l_2$
19. where $l_1$ is above $l_2$ to the left of $p$
20. $l_3 \leftarrow$ above($l_1, S$) {to the left of $p$}
21. $l_4 \leftarrow$ below($l_2, S$) {to the left of $p$}
22. **if** $l_2$ intersects $l_3$ at point $q_1$ **then** process($q_1$)
23. **if** $l_1$ intersects $l_4$ at point $q_2$ **then** process($q_2$)
24. interchange the ordering of $l_1$ and $l_2$ in $S$
25. **end if**
26. **end while**

As regards the running time of the algorithm, we observe the following. The sorting step takes time $O(n \log n)$. Let the number of intersections be $m$. Then, there are $2n + m$ event points to be processed. Each point requires $O(\log(2n+m))$ processing time. Hence, the total time required by the algorithm to process all intersection points is $O((2n+m)\log(2n+m))$. Since $m \leq n(n-1)/2 = O(n^2)$, the bound becomes $O((n+m)\log n)$. Since the naïve approach to find all intersections runs in time $O(n^2)$, the algorithm is not suitable to process a set of line segments whose number of intersections is known *apriori* to be $\Omega(n^2/\log n)$. On the other hand, if $m = O(n)$, then the algorithm runs in $O(n \log n)$ time.



## 18.4 The Convex Hull Problem

In this section we consider, perhaps, the most fundamental problem in computational geometry: Given a set $S$ of $n$ points in the plane, find $CH(S)$, the convex hull of $S$. We describe here a well-known geometric sweeping algorithm called "Graham scan".

In its simplest form, Graham scan uses a line centered at a certain point and makes one rotation that sweeps the whole plane stopping at each point to decide whether it should be included in the convex hull or not. First, in one scan over the list of points, the point with minimum $y$-coordinate is found, call it $p_0$. If there are two or more points with the minimum $y$-coordinate, $p_0$ is chosen as the rightmost one. Clearly, $p_0$ belongs to the convex hull. Next, the coordinates of all points are transformed so that $p_0$ is at the origin. The points in $S - \{p_0\}$ are then sorted by polar angle about the origin $p_0$. If two points $p_i$ and $p_j$ form the same angle with $p_0$, then the one that is closer to $p_0$ precedes the other in the ordering. Note that here we do not have to calculate the real distance from the origin, as it involves computing the square root which is costly; instead, we only need to compare the squares of the distances. Let the sorted list be $T = \{p_1, p_2, \cdots, p_{n-1}\}$, where $p_1$ and $p_{n-1}$ form the least and greatest angles with $p_0$, respectively. Figure 18.7 shows an example of a set of 13 points after sorting them by polar angle about $p_0$.

Now, the scan commences with the event point schedule being the sorted list $T$, and the sweep line status being implemented using a stack $St$. The stack initially contains $(p_{n-1}, p_0)$, with $p_0$ being on top of the stack. The algorithm then traverses the points starting at $p_1$ and ending at $p_{n-1}$. At any moment, let the stack content be

$$St = (p_{n-1}, p_0, \cdots, p_i, p_j)$$

(i.e. $p_i$ and $p_j$ are the most recently pushed points), and let $p_k$ be the next point to be considered. If the triplet $p_i, p_j, p_k$ forms a left turn, then $p_k$ is pushed on top of the stack and the sweep line is moved to the next point. If $p_i, p_j, p_k$ form a right turn or are collinear, then $p_j$ is popped off the stack

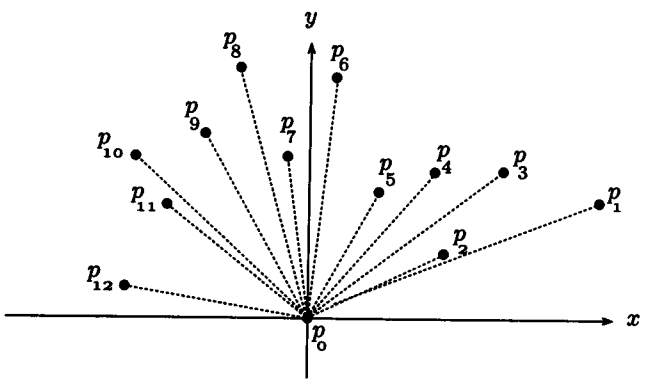


Fig. 18.7 A set of points sorted in polar angle about $p_0$.

and the sweep line is kept at point $p_k$.

Figure 18.8 shows the resulting convex hull just after $p_5$ has been processed. At this point, the stack content is

$$(p_{12}, p_0, p_1, p_3, p_4, p_5).$$

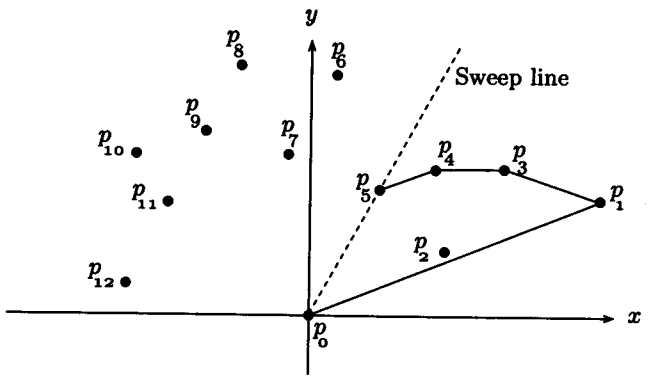


Fig. 18.8 The convex hull after processing point $p_5$.

After processing point $p_6$, the points $p_5, p_4$ and $p_3$ are successively popped off the stack, and the point $p_6$ is pushed on top of the stack (see Fig. 18.9). The final convex hull is shown in Fig. 18.10.

Given below is a more formal description of the algorithm. At the end



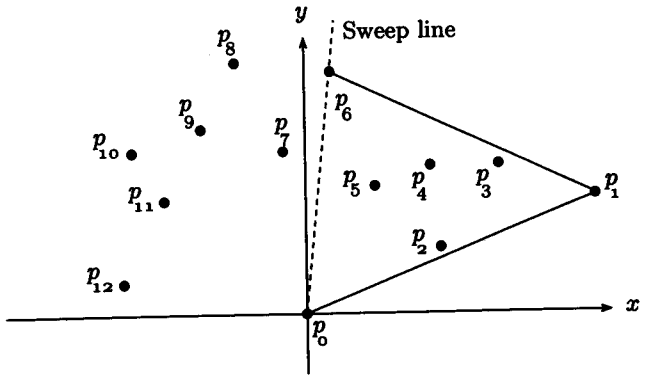


Fig. 18.9 The convex hull after processing point $p_6$.

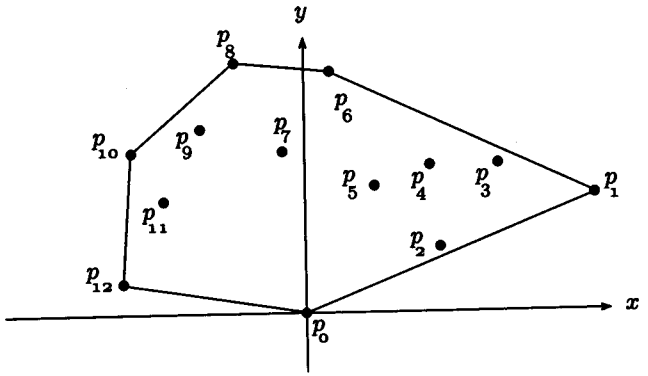


Fig. 18.10 The final convex hull.

of the algorithm, the stack $St$ contains the vertices of $CH(S)$, so it can be converted into a linked list to form a convex polygon.

The running time of Algorithm CONVEXHULL is computed as follows. The sorting step costs $O(n \log n)$ time. As to the **while** loop, we observe that each point is pushed exactly once and is popped at most once. Moreover, checking whether three points form a left turn or a right turn amounts to computing their signed area in $\Theta(1)$ time. Thus the cost of the while loop is $\Theta(n)$. It follows that the time complexity of the algorithm is $O(n \log n)$. See Exercise 18.9 for an alternative approach that avoids computing the

**Algorithm 18.3** CONVEXHULL
**Input:** A set $S$ of $n$ points in the plane.
**Output:** $CH(S)$, the convex hull of $S$ stored in a stack $St$.

1. Let $p_0$ be the rightmost point with minimum $y$-coordinate.
2. $T[0] \leftarrow p_0$
3. Let $T[1..n-1]$ be the points in $S - \{p_0\}$ sorted in increasing polar angle about $p_0$. If two points $p_i$ and $p_j$ form the same angle with $p_0$, then the one that is closer to $p_0$ precedes the other in the ordering.
4. **push** $(St, T[n-1])$; **push** $(St, T[0])$
5. $k \leftarrow 1$
6. **while** $k < n-1$
7. Let $St = (T[n-1], \ldots, T[i], T[j])$, $T[j]$ is on top of the stack.
8. **if** $T[i], T[j], T[k]$ is a left turn **then**
9. **push** $(St, T[k])$
10. $k \leftarrow k+1$
11. **else pop** $(St)$
12. **end if**
13. **end while**

polar angles. Other algorithms for computing the convex hull are outlined in Exercises 18.10 and 18.13.

## 18.5 Computing the Diameter of a Set of Points

Let $S$ be a set of points in the plane. The diameter of $S$, denoted by $Diam(S)$, is defined to be the maximum distance realized by two points in $S$. A straightforward algorithm to solve this problem compares each pair of points and returns the maximum distance realized by two points in $S$. This approach leads to a $\Theta(n^2)$ time algorithm. In this section, we study an algorithm to find the diameter of a set of points in the plane in time $O(n \log n)$.

We start with the following observation, which seems to be intuitive (see Fig. 18.11):

**Observation 18.1** The diameter of a point set $S$ is equal to the diameter of the vertices of its convex hull, i.e., $Diam(S) = Diam(CH(S))$.

Consequently, to compute the diameter of a set of points in the plane, we only need to consider the vertices on its convex hull. Therefore, in



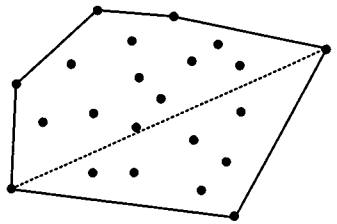

Fig. 18.11 The diameter of a set of points is the diameter of its convex hull.

what follows we will be concerned primarily with the problem of finding the diameter of a convex polygon.

**Definition 18.3** Let $P$ be a convex polygon. A *supporting line* of $P$ is a straight line $l$ passing through a vertex of $P$ such that the interior of $P$ lies entirely on one side of $l$ (see Fig. 18.12).

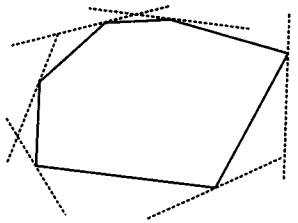

Fig. 18.12 Some supporting lines of a convex polygon.

A useful characterization of the diameter of a convex polygon is given in the following theorem (see Fig. 18.13).

**Theorem 18.1** The diameter of a convex polygon $P$ is equal to the greatest distance between any pair of parallel supporting lines of $P$.

**Definition 18.4** Any two points that admit two parallel supporting lines are called *antipodal pair.*

We have the following corollary of Theorem 18.1:

**Corollary 18.1** Any pair of vertices realizing the diameter in a convex polygon is an antipodal pair.

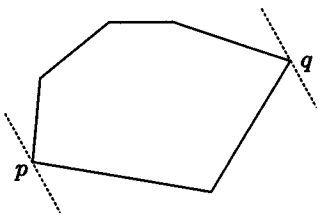


Fig. 18.13 Two parallel supporting lines with largest separation.

By the above corollary, the problem now reduces to finding *all* antipodal pairs and selecting the one with maximum separation. It turns out that we can accomplish that in optimal linear time.

**Definition 18.5** We define the distance between a point $p$ and a line segment $\overline{qr}$, denoted by $dist(q,r,p)$ to be the distance of $p$ from the straight line on which the line segment $\overline{qr}$ lies. A vertex $p$ is *farthest* from a line segment $\overline{qr}$ if $dist(q,r,p)$ is maximum.

Consider Fig. 18.14(a) in which a convex polygon $P$ is shown. From the figure, it is easy to see that $p_5$ is the farthest vertex from edge $\overline{p_{12}p_1}$. Similarly, vertex $p_9$ is the farthest from edge $\overline{p_1p_2}$.

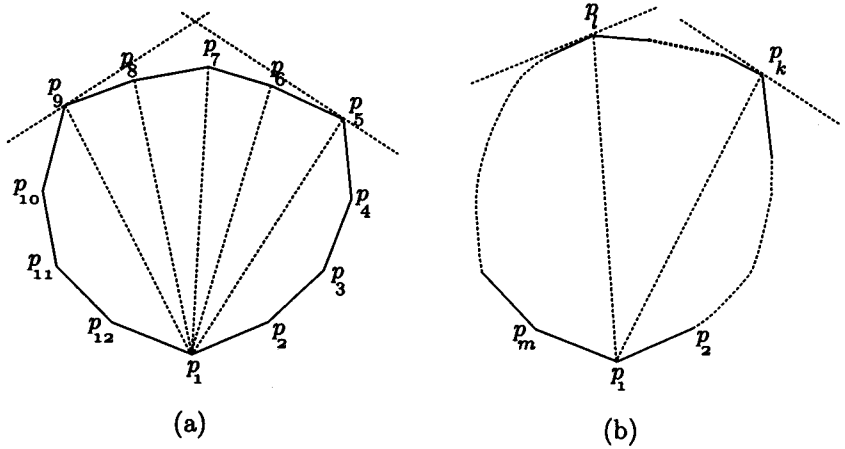


Fig. 18.14 Computing the set of antipodal pairs.

It can be shown that a vertex $p$ forms an antipodal pair with $p_1$ if and



only if it is one of the vertices $p_5, p_6, \ldots, p_9$. In general, let the vertices on the convex hull of the point set be $p_1, p_2, \ldots, p_m$ for some $m \leq n$, in counterclockwise ordering. Let $p_k$ be the first farthest vertex from edge $\overline{p_mp_1}$, and $p_l$ the first farthest vertex from edge $\overline{p_1p_2}$ when traversing the boundary of $CH(S)$ in counterclockwise order (see Fig. 18.14(b)). Then, any vertex between $p_k$ and $p_l$ (including $p_k$ and $p_l$) forms an antipodal pair with $p_1$. Moreover, all other vertices do *not* form an antipodal pair with $p_1$.

This important observation suggests the following method for finding all antipodal pairs. First, we traverse the boundary of $CH(S)$ in counterclockwise order starting at $p_2$ until we find $p_k$, the farthest vertex from $\overline{p_mp_1}$. We add the pair $(p_1, p_k)$ to an initially empty set for holding the antipodal pairs. We then keep traversing the boundary and include the pair $(p_1, p_j)$ for each vertex $p_j$ encountered until we reach $p_l$, the vertex farthest from $\overline{p_1p_2}$. It may be the case that $l = k+1$ or even $l = k$, i.e., $p_l = p_k$. Next, we advance to the edge $\overline{p_2p_3}$ to find the vertices that form antipodal pairs with $p_2$. Thus, we are simultaneously doing two counterclockwise traversals of the boundary: one from $p_1$ to $p_k$ and the other from $p_k$ to $p_m$. The traversal ends when the antipodal pair $(p_k, p_m)$ is detected. Finally, a linear scan over the set of antipodal pairs is clearly sufficient to find the diameter of the convex hull, which by Observation 18.1 is the desired diameter of the point set. This method is described more formally in Algorithm DIAMETER.

If the convex hull contains no parallel edges, the number of antipodal pairs will be exactly $m$, which is the size of the convex hull. If there are pairs of parallel edges, then their number is at most $\lfloor m/2 \rfloor$ and hence the total number of antipodal pairs is at most $\lfloor 3m/2 \rfloor$.

When comparing the distance between a vertex and a line segment, we do not compute the actual distance (which involves taking the square roots); instead, we compare the signed area since it is proportional to the actual distance (see Sec. 18.2 for the definition of the signed area). For example, the comparison

$$dist(p_i, p_{i+1}, p_{j+1}) \geq dist(p_i, p_{i+1}, p_j)$$

in the algorithm can be replaced with the comparison

$$area(p_i, p_{i+1}, p_{j+1}) \geq area(p_i, p_{i+1}, p_j)$$

where $area(q,r,p)$ is the area of the triangle formed by the line segment $\overline{qr}$

**Algorithm 18.4** DIAMETER
**Input:** A set $S$ of $n$ points in the plane.
**Output:** $Diam(S)$, the diameter of $S$.

```
1. {p_1, p_2, ..., p_m} ← CH(S)      {Compute the convex hull of S}
2. A ← {}      {Initialize the set of antipodal pairs}
3. k ← 2
4. while dist(p_m, p_1, p_{k+1}) > dist(p_m, p_1, p_k)      {Find p_k}
5.     k ← k + 1
6. end while
7. i ← 1;   j ← k
8. while i ≤ k and j ≤ m
9.     A ← A ∪ {(p_i, p_j)}
10.    while dist(p_i, p_{i+1}, p_{j+1}) ≥ dist(p_i, p_{i+1}, p_j) and j < m
11.        A ← A ∪ {(p_i, p_j)}
12.        j ← j + 1
13.    end while
14.    i ← i + 1
15. end while
16. Scan A to obtain an antipodal pair (p_r, p_s) with maximum separation.
17. return the distance between p_r and p_s.
```

and the point $p$. This area is half the magnitude of the signed area of these three points.

The running time of the algorithm is computed as follows. Finding the convex hull requires $O(n \log n)$ time. Since the two nested **while** loops consist of two concurrent sweeps of the boundary of the convex hull, the time taken by these nested **while** loops is $\Theta(m) = O(n)$, where $m$ is the size of the convex hull. It follows that the overall running time of the algorithm is $O(n \log n)$.

## 18.6 Exercises

**18.1.** Let $S$ be a set of $n$ points in the plane. Design an $O(n \log n)$ algorithm to compute for each point $p$ the number of points in $S$ dominated by $p$.

**18.2.** Let $I$ be a set of intervals on the horizontal line. Design an algorithm to report all those intervals that are contained in another interval from $I$. What is the running time of your algorithm?

**18.3.** Consider the decision problem version of the line segment intersection problem: Given $n$ line segments in the plane, determine whether two of them intersect. Give an $O(n \log n)$ time algorithm to solve this problem.



**18.4.** Give an efficient algorithm to report all intersecting pairs of a set of $n$ horizontal line segments. What is the time complexity of the algorithm?

**18.5.** Give an efficient algorithm to report all intersecting pairs of a given set of $n$ horizontal and vertical line segments. What is the time complexity of the algorithm?

**18.6.** Explain how to determine whether a given polygon is simple. Recall that a polygon is simple if and only if no two of its edges intersect except at its vertices.

**18.7.** Let $P$ and $Q$ be two simple polygons whose total number of vertices is $n$. Give an $O(n \log n)$ time algorithm to determine whether $P$ and $Q$ intersect.

**18.8.** Give an $O(n)$ time algorithm to solve the problem in Exrcise 18.7 in the case where the two polygons are convex.

**18.9.** In Graham scan for finding the convex hull of a point set, the points are sorted by their polar angles. However, computing the polar angles is costly. One alternative to computing the convex hull is to sort using the sines or cosines of the angles instead. Another alternative is to sort the points around the point $(0, -\infty)$ instead. This is equivalent to sorting the points by decreasing $x$-coordinates. Explain how to use this idea to come up with another algorithm for computing the convex hull.

**18.10.** Another algorithm for finding the convex hull is known as *Jarvis march*. In this algorithm, the edges of the convex hull are found instead of its vertices. The algorithm starts by finding the point with the least $y$-coordinate, say $p_1$, and finding the point $p_2$ with the least polar angle with respect to $p_1$. Thus, the line segment $\overline{p_1 p_2}$ defines an edge of the convex hull. The next edge is determined by finding the point $p_3$ with the least polar angle with respect to $p_2$, and so on. From its description, the algorithm resembles Algorithm SELECTIONSORT. Give the details of this method. What is its time complexity?

**18.11.** What are the merits and demerits of Jarvis march for finding the convex hull as described in Exercise 18.10?

**18.12.** Let $p$ be a point external to a convex polygon $P$. Given $CH(P)$, explain how to compute in $O(\log n)$ time the convex hull of their union, i.e., the convex hull of $P \cup \{p\}$.

**18.13.** Use the result of Exercise 18.12 to devise an incremental algorithm for computing the convex hull of a set of points. The algorithm builds the convex hull by testing one point at a time and deciding whether it belongs to the current convex hull or not. The algorithm should run in time $O(n \log n)$.

**18.14.** Design an $O(n)$ time algorithm to find the convex hull of two given convex polygons, where $n$ is the total number of vertices in both polygons.

**18.15.** Give an $O(n)$ time algorithm that decides whether a point $p$ is inside a simple polygon.

**18.16.** Prove or disprove the following statement: Given a set of points $S$ in the plane, there is only one unique simple polygon whose vertices are the points in $S$.

**18.17.** Given a set of $n$ points in the plane, show how to construct a simple polygon having them as its vertices. The algorithm should run in time $O(n \log n)$.

**18.18.** Referring to the algorithm for finding the diameter of a given point set $S$ in the plane, prove that the diameter is the distance between two points on their convex hull.

**18.19.** Prove Theorem 18.1.

**18.20.** Let $P$ be a simple polygon with $n$ vertices. $P$ is called *monotone with respect to the y-axix* if for any line $l$ perpendicular to the $y$-axix, the intersection of $l$ and $P$ is either a line segment or a point. For example, any convex polygon is monotone with respect to the $y$-axix. A *chord* in $P$ is a line segment that connects two nonadjacent vertices in $P$ and lies entirely inside $P$. The problem of *triangulating* a simple polygon is to partition the polygon into $n-2$ triangles by drawing $n-3$ nonintersecting chords inside $P$ (see Fig. 7.8 for the special case of convex polygons). Give an algorithm to triangulate $P$ in $\Theta(n)$ time.

## 18.7 Bibliographic Notes

Some books on computational geometry include de Berg *et al.* (1997), Edelsbrunner (1987), Mehlhorn (1984c), O'Rourke (1994), Preparata and Shamos(1985) and Toussaint (1984). The algorithm for computing line segment intersections is due to Shamos and Hoey (1975). The convex hull algorithm is due to Graham (1972). Theorem 18.1 is due to Yaglom and Boltyanskii (1986). The algorithm of finding the diameter can be found in Preparata and Shamos (1972). The problem of triangulating a simple polygon is fundamental in computational geometry. The solution to the problem of triangulating a monotone polygon in $\Theta(n)$ time (Exercise 18.20) can be found in Garey *et al.* (1978). In this paper, it was also shown that triangulating a simple polygon can be achieved in $O(n \log n)$ time. Later, Tarjan and Van Wyk (1988) gave an $O(n \log \log n)$ time algorithm for triangulating a simple polygon. Finally, Chazelle (1990, 1991) gave a linear time algorithm, which is quite complicated. The Voronoi diagram can also be computed using line sweeping in $O(n \log n)$ time (Fortune (1992)).