
# 6. Computational Geometry

**Pukar Karki**
**Assistant Professor**

# Introduction

- Computational geometry is the branch of computer science that studies algorithms for solving geometric problems.
- In modern engineering and mathematics, computational geometry has applications in such diverse fields as computer graphics, robotics, VLSI design, computer-aided design, molecular modeling, metallurgy, manufacturing, textile layout, forestry, and statistics.

# Introduction

- The input to a computational geometry problem is typically a description of a set of geometric objects, such as a set of points, a set of line segments, or the vertices of a polygon in counterclock wise order.
- The output is often a response to a query about the objects, such as whether any of the lines intersect, or perhaps a new geometric object, such as the convex hull (smallest enclosing convex polygon) of the set of points.

# Introduction

- In this chapter, we look at a few computational-geometry algorithms in two dimensions, that is, in the plane.
- We represent each input object by a set of points $\{p_1, p_2, p_3, \ldots\}$, where each $p_i = (x_i, y_i)$ and $x_i, y_i \in R$.
- For example, we represent an n-vertex polygon P by a sequence

$$\mathbf{\{p_0, p_1, p_2 \ldots . \ p_n\}}$$

of its vertices in order of their appearance on the boundary of P .

# Line-Segment Properties

- Several of the computational-geometry algorithms in this chapter require answers to questions about the properties of line segments.
- A convex combination of two distinct points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is any point $p_3 = (x_3, y_3)$ such that for some $\alpha$ in the range $0 \leq \alpha \leq 1$, we have $x_3 = \alpha x_1 + (1 - \alpha)x_2$ and $y_3 = \alpha y_1 + (1 - \alpha)y_2$.
- We also write that $p_3 = \alpha p_1 + (1 - \alpha)p_2$.
- Intuitively, $p_3$ is any point that is on the line passing through $p_1$ and $p_2$ and is on or between $p_1$ and $p_2$ on the line.

# Line-Segment Properties

- Given two distinct points $p_1$ and $p_2$, the line segment $\overline{p_1p_2}$ is the set of convex combinations of $p_1$ and $p_2$.
- We call $p_1$ and $p_2$ the endpoints of segment $\overline{p_1p_2}$.
- Sometimes the ordering of $p_1$ and $p_2$ matters, and we speak of the directed segment $\overrightarrow{p_1p_2}$.
- If $p_1$ is the origin (0, 0), then we can treat the directed segment $\overrightarrow{p_1p_2}$ as the vector $p_2$.

# Line-Segment Properties

- In this section, we shall explore the following questions:

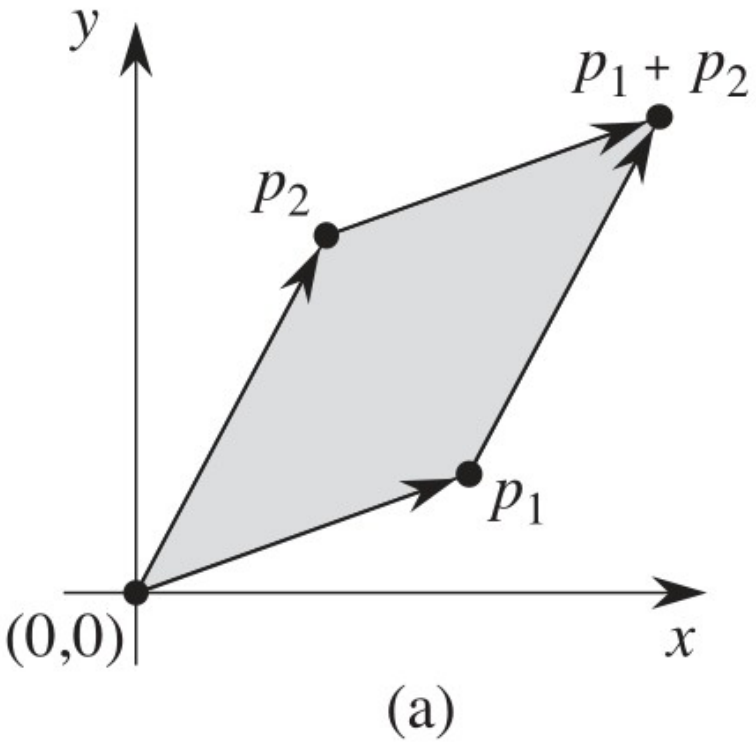
1. Given two directed segments $\overrightarrow{p_0 p_1}$ and $\overrightarrow{p_0 p_2}$, is $\overrightarrow{p_0 p_1}$ clockwise from $\overrightarrow{p_0 p_2}$ with respect to their common endpoint $p_0$?
2. Given two line segments $\overline{p_0 p_1}$ and $\overline{p_1 p_2}$, if we traverse $\overline{p_0 p_1}$ and then $\overline{p_1 p_2}$, do we make a left turn at point $p_1$?
3. Do line segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ intersect?

- We can answer each question in O(1) time.

# Line-Segment Properties

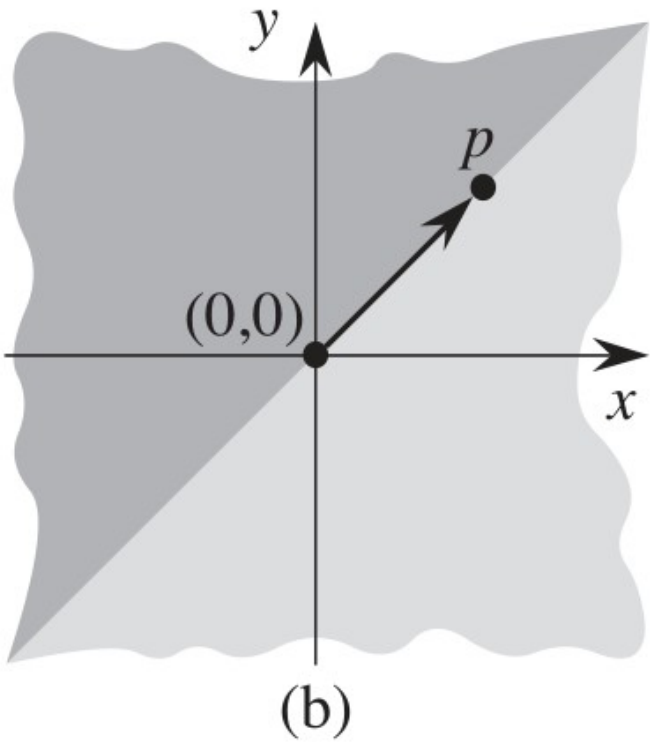
## Cross Products

- Computing cross products lies at the heart of our line-segment methods. Consider vectors $p_1$ and $p_2$, shown in figure:




(a)

(a) The cross product of vectors $p_1$ and $p_2$ is the signed area of the parallelogram.




(b)

(b) The lightly shaded region contains vectors that are clockwise from p. The darkly shaded region contains vectors that are counterclockwise from p.

# Line-Segment Properties

## Cross Products

- Cross product $p_1 \times p_2$ can be viewed as the signed area of the parallelogram formed by the points (0, 0), $p_1$, $p_2$ and $(p_1 + p_2) = (x_1 + x_2, y_1 + y_2)$.
- An equivalent more useful, definition gives the cross product as the determinant of a matrix.

$$\begin{aligned} p_1 \times p_2 &= \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} \\ &= x_1 y_2 - x_2 y_1 \\ &= -p_2 \times p_1 \end{aligned}$$

- If $p_1 \times p_2$ is positive, then $p_1$ is clockwise from $p_2$ with respect to the origin (0, 0) and if this cross product is negative, then $p_1$ is counterclockwise from $p_2$.
- A boundary condition arises if the cross product is 0; in this case, the vectors are co-linear, pointing in either the same or opposite directions.

# Line-Segment Properties

## Cross Products

- To determine whether a directed segment $\overrightarrow{p_0p_1}$ is closer to a directed segment $\overrightarrow{p_0p_2}$ in a clockwise direction or in a counterclockwise direction with respect to their common endpoint $p_0$, we simply translate to use $p_0$ as the origin.
- That is, we let $p_1 - p_0$ denote the vector $p'_1 = (x'_1, y'_1)$, where $x'_1 = x_1 - x_0$ and $y'_1 = y_1 - y_0$, and we define $p_2 - p_0$ similarly.
- We then compute the cross product

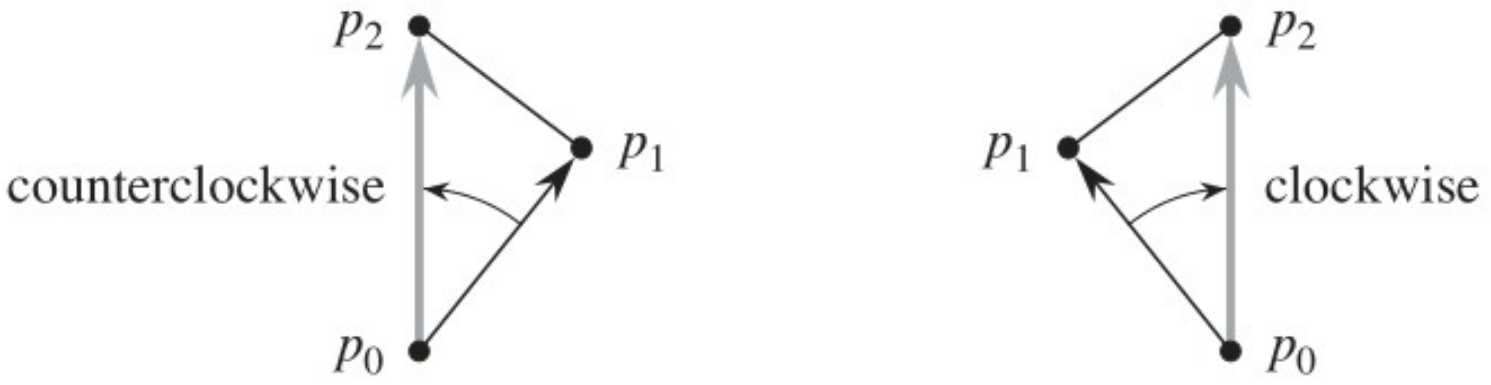
$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0)$$

- If this cross product is positive, then $\overrightarrow{p_0p_1}$ is clockwise from $\overrightarrow{p_0p_2}$; if negative, it is counterclockwise.

# Line-Segment Properties

## Determining whether consecutive segments turn left or right

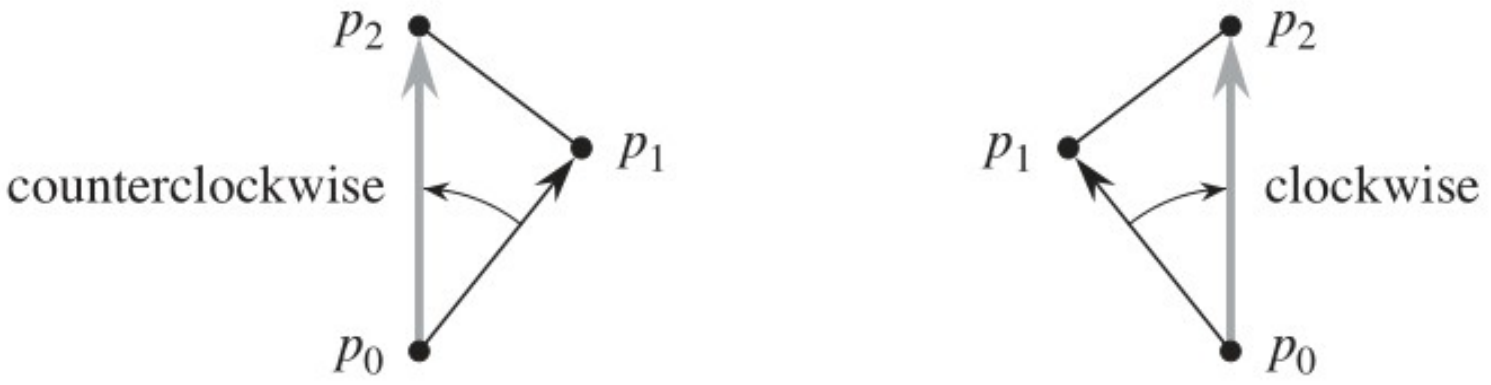
- Our next question is whether two consecutive line segments $\overline{p_0p_1}$ and $\overline{p_1p_2}$ turn left or right at point $p_1$. Equivalently, we want a method to determine which way a given angle $p_0p_1p_2$ turns.
- Cross products allow us to answer this question without computing the angle.
- As Figure below shows, we simply check whether directed segment $\overrightarrow{p_0p_2}$ is clockwise or counterclockwise relative to directed segment $\overrightarrow{p_0p_1}$.



# Line-Segment Properties

**Determining whether consecutive segments turn left or right**

- To do so, we compute the cross product $(p_2 - p_0) \times (p_1 - p_0)$.
- If the sign of this cross product is negative, the $\overrightarrow{p_0p_2}$ is counterclockwise with respect to $\overrightarrow{p_0p_1}$, and thus we make a left turn at $p_1$.
- A positive cross product indicates a clockwise orientation and a right turn.
- A cross product of 0 means that points $p_0$, $p_1$, and $p_2$ are colinear.



# Line-Segment Properties

## Determining whether two line segments intersect

- To determine whether two line segments intersect, we check whether each segment straddles the line containing the other.
- A segment $\overline{p_1p_2}$ straddles a line if point $p_1$ lies on one side of the line and point $p_2$ lies on the other side.
- A boundary case arises if $p_1$ or $p_2$ lies directly on the line.
- Two line segments intersect if and only if either (or both) of the following conditions holds:

1) Each segment straddles the line containing the other.
2) An endpoint of one segment lies on the other segment. (This condition comes from the boundary case.)

```
SEGMENTS-INTERSECT(p1, p2, p3, p4)
 1  d1 = DIRECTION(p3, p4, p1)
 2  d2 = DIRECTION(p3, p4, p2)
 3  d3 = DIRECTION(p1, p2, p3)
 4  d4 = DIRECTION(p1, p2, p4)
 5  if ((d1 > 0 and d2 < 0) or (d1 < 0 and d2 > 0)) and
        ((d3 > 0 and d4 < 0) or (d3 < 0 and d4 > 0))
 6      return TRUE
 7  elseif d1 == 0 and ON-SEGMENT(p3, p4, p1)
 8      return TRUE
 9  elseif d2 == 0 and ON-SEGMENT(p3, p4, p2)
10      return TRUE
11  elseif d3 == 0 and ON-SEGMENT(p1, p2, p3)
12      return TRUE
13  elseif d4 == 0 and ON-SEGMENT(p1, p2, p4)
14      return TRUE
15  else return FALSE
```

```
DIRECTION(pi, pj, pk)
1  return (pk − pi) × (pj − pi)
```

- **SEGMENTS-INTERSECT** returns TRUE if segments $\overline{p_1p_2}$ and $\overline{p_3p_4}$ intersect and FALSE if they do not.
- It calls the subroutines DIRECTION, which computes relative orientations using the cross-product method above, and ON-SEGMENT, which determines whether a point known to be colinear with a segment lies on that segment.

```
ON-SEGMENT(pi, pj, pk)
1  if min(xi, xj) ≤ xk ≤ max(xi, xj) and min(yi, yj) ≤ yk ≤ max(yi, yj)
2      return TRUE
3  else return FALSE
```

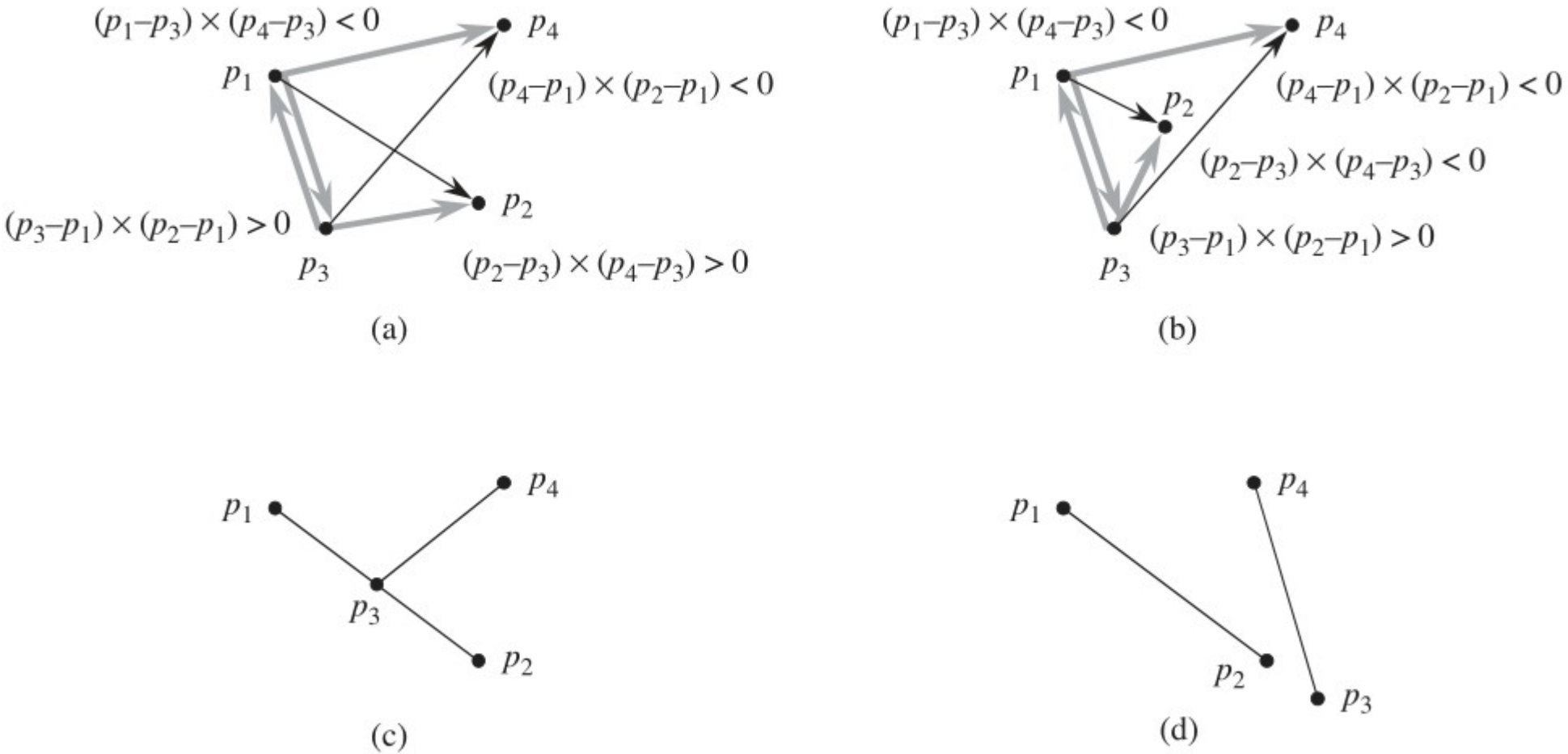



**Figure** Cases in the procedure SEGMENTS-INTERSECT. **(a)** The segments $\overline{p_1 p_2}$ and $\overline{p_3 p_4}$ straddle each other's lines. Because $\overline{p_3 p_4}$ straddles the line containing $\overline{p_1 p_2}$, the signs of the cross products $(p_3 - p_1) \times (p_2 - p_1)$ and $(p_4 - p_1) \times (p_2 - p_1)$ differ. Because $\overline{p_1 p_2}$ straddles the line containing $\overline{p_3 p_4}$, the signs of the cross products $(p_1 - p_3) \times (p_4 - p_3)$ and $(p_2 - p_3) \times (p_4 - p_3)$ differ. **(b)** Segment $\overline{p_3 p_4}$ straddles the line containing $\overline{p_1 p_2}$, but $\overline{p_1 p_2}$ does not straddle the line containing $\overline{p_3 p_4}$. The signs of the cross products $(p_1 - p_3) \times (p_4 - p_3)$ and $(p_2 - p_3) \times (p_4 - p_3)$ are the same. **(c)** Point $p_3$ is colinear with $\overline{p_1 p_2}$ and is between $p_1$ and $p_2$. **(d)** Point $p_3$ is colinear with $\overline{p_1 p_2}$, but it is not between $p_1$ and $p_2$. The segments do not intersect.

# Determining Whether Any pair of Segments Intersects

- This section presents an algorithm for determining whether any two line segments in a set of segments intersect.
- The algorithm uses a technique known as “sweeping,” which is common to many computational-geometry algorithms.
- The algorithm runs in O(n lg n) time, where n is the number of segments we are given.
- It determines only whether or not any intersection exists.

# Determining Whether Any pair of Segments Intersects

- In sweeping, an imaginary vertical sweep line passes through the given set of geometric objects, usually from left to right.
- We treat the spatial dimension that the sweep line moves across, in this case the x-dimension, as a dimension of time.

# Determining Whether Any pair of Segments Intersects

- To describe and prove correct our algorithm for determining whether any two of n line segments intersect, we shall make two simplifying assumptions.
- First, we assume that no input segment is vertical. Second, we assume that no three input segments intersect at a single point.

# Determining Whether Any pair of Segments Intersects

## Ordering segments

- Because we assume that there are no vertical segments, we know that any input segment intersecting a given vertical sweep line intersects it at a single point.
- Thus, we can order the segments that intersect a vertical sweep line according to the y-coordinates of the points of intersection.

# Determining Whether Any pair of Segments Intersects

## Ordering segments

- To be more precise, consider two segments $s_1$ and $s_2$. We say that these segments are comparable at x if the vertical sweep line with x-coordinate x intersects both of them.
- We say that $s_1$ is above $s_2$ at x, written $s_1 \succcurlyeq_x s_2$, if $s_1$ and $s_2$ are comparable at x and the intersection of $s_1$ with the sweep line at x is higher than the intersection of $s_2$ with the same sweep line, or if $s_1$ and $s_2$ intersect at the sweep line.
- For any given x, the relation $\succcurlyeq_x$ is a total preorder for all segments that intersect the sweep line at x.

# Determining Whether Any pair of Segments Intersects

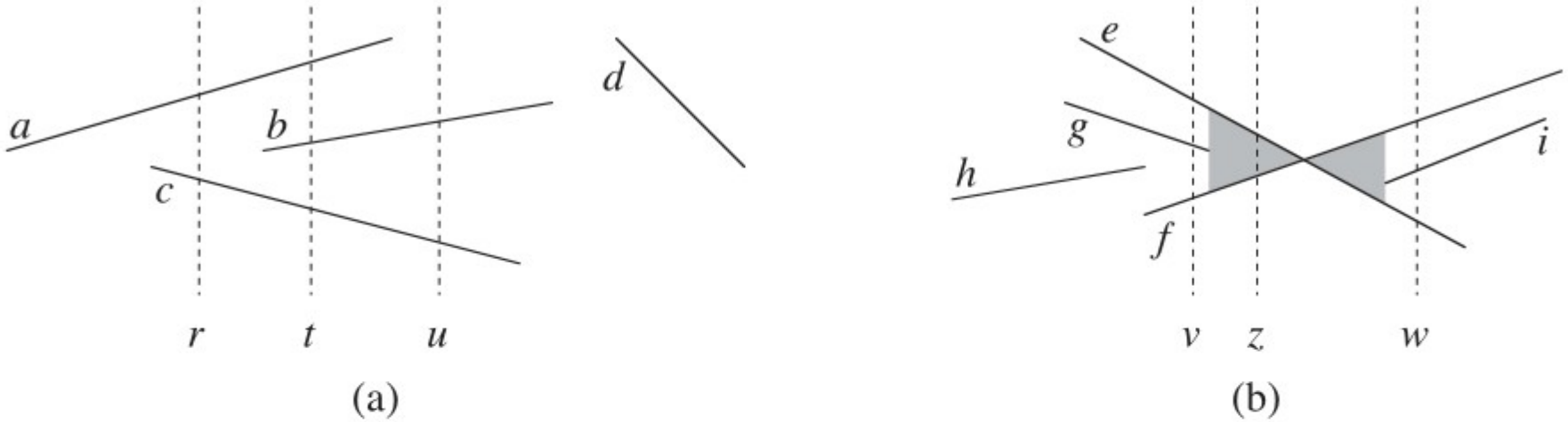



**Figure** The ordering among line segments at various vertical sweep lines. **(a)** We have $a \succcurlyeq_r c$, $a \succcurlyeq_t b$, $b \succcurlyeq_t c$, $a \succcurlyeq_t c$, and $b \succcurlyeq_u c$. Segment $d$ is comparable with no other segment shown. **(b)** When segments $e$ and $f$ intersect, they reverse their orders: we have $e \succcurlyeq_v f$ but $f \succcurlyeq_w e$. Any sweep line (such as $z$) that passes through the shaded region has $e$ and $f$ consecutive in the ordering given by the relation $\succcurlyeq_z$.

# Determining Whether Any pair of Segments Intersects

## Moving the sweep line

- Sweeping algorithms typically manage two sets of data:

1) The **sweep-line status** gives the relationships among the objects that the sweep line intersects.

2) The **event-point schedule** is a sequence of points, called **event points**, which we order from left to right according to their x-coordinates. As the sweep progresses from left to right, whenever the sweep line reaches the x-coordinate of an event point, the sweep halts, processes the event point, and then resumes. Changes to the sweep-line status occur only at event points.

# Determining Whether Any pair of Segments Intersects

- The sweep-line status is a total preorder T , for which we require the following operations:

1) INSERT(T, s): insert segment s into T.

2) DELETE(T, s): delete segment s from T.

3) ABOVE(T, s): return the segment immediately above segment s in T.

4) BELOW(T, s): return the segment immediately below segment s in T.

# Determining Whether Any pair of Segments Intersects

- If the input contains n segments, we can perform each of the operations INSERT, DELETE, ABOVE, and BELOW in O(lg n) time using red-black trees.
- Recall that the red-black-tree operations involve comparing keys.
- We can replace the key comparisons by comparisons that use cross products to determine the relative ordering of two segments.

**Refer to Chapter 13 of "Introduction to the Algorithms" book for more information regarding red-black-trees.**

# Determining Whether Any pair of Segments Intersects

- The following algorithm takes as input a set S of n line segments, returning the boolean value TRUE if any pair of segments in S intersects, and FALSE otherwise.
- A red-black tree maintains the total preorder T .

# Determining Whether Any pair of Segments Intersects

```
ANY-SEGMENTS-INTERSECT(S)
 1  T = ∅
 2  sort the endpoints of the segments in S from left to right,
        breaking ties by putting left endpoints before right endpoints
        and breaking further ties by putting points with lower
        y-coordinates first
 3  for each point p in the sorted list of endpoints
 4      if p is the left endpoint of a segment s
 5          INSERT(T, s)
 6          if (ABOVE(T, s) exists and intersects s)
                or (BELOW(T, s) exists and intersects s)
 7              return TRUE
 8      if p is the right endpoint of a segment s
 9          if both ABOVE(T, s) and BELOW(T, s) exist
                and ABOVE(T, s) intersects BELOW(T, s)
10              return TRUE
11          DELETE(T, s)
12  return FALSE
```

# Determining Whether Any pair of Segments Intersects

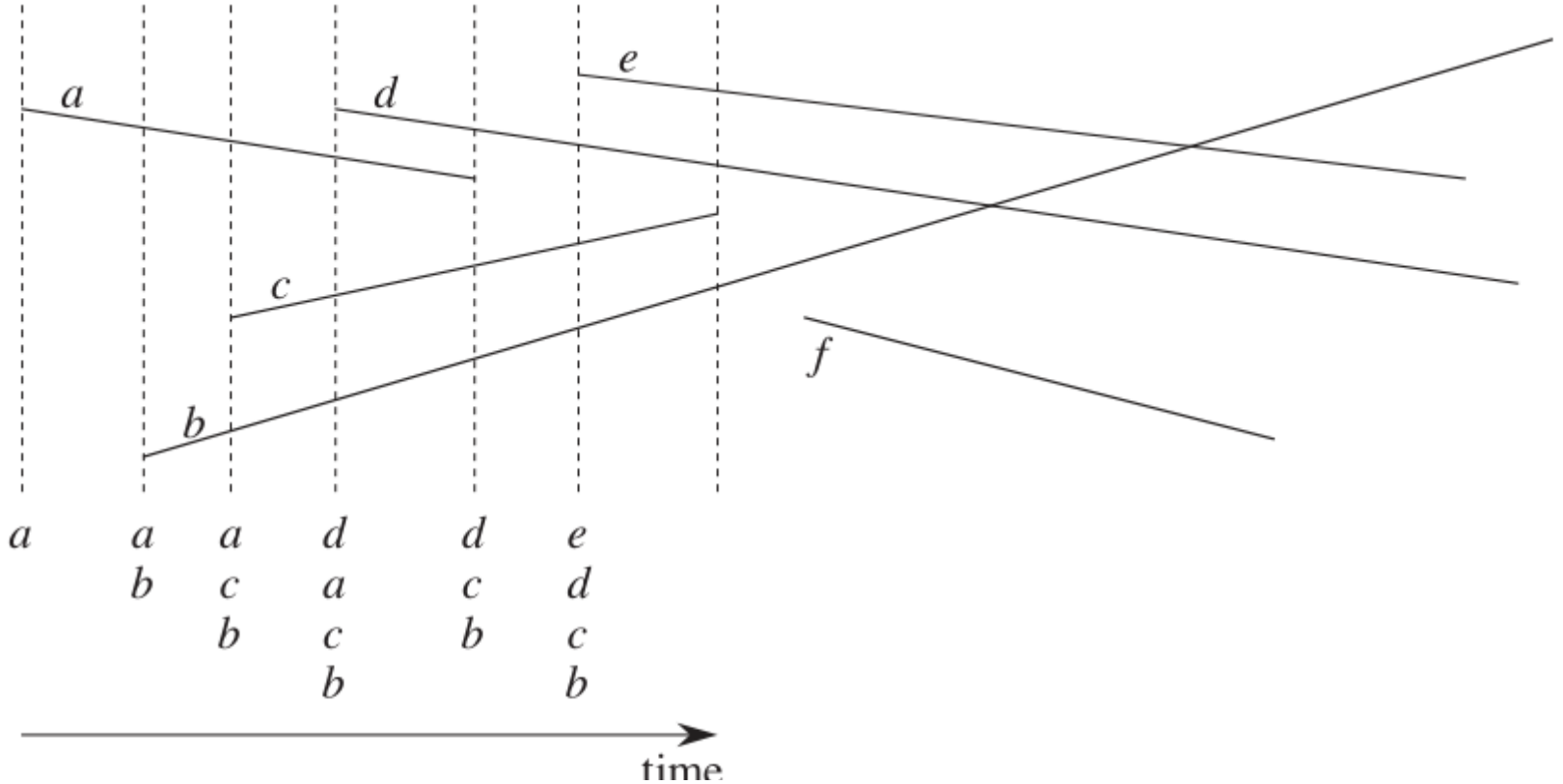



**Figure** The execution of ANY-SEGMENTS-INTERSECT. Each dashed line is the sweep line at an event point. Except for the rightmost sweep line, the ordering of segment names below each sweep line corresponds to the total preorder $T$ at the end of the **for** loop processing the corresponding event point. The rightmost sweep line occurs when processing the right endpoint of segment $c$; because segments $d$ and $b$ surround $c$ and intersect each other, the procedure returns TRUE.

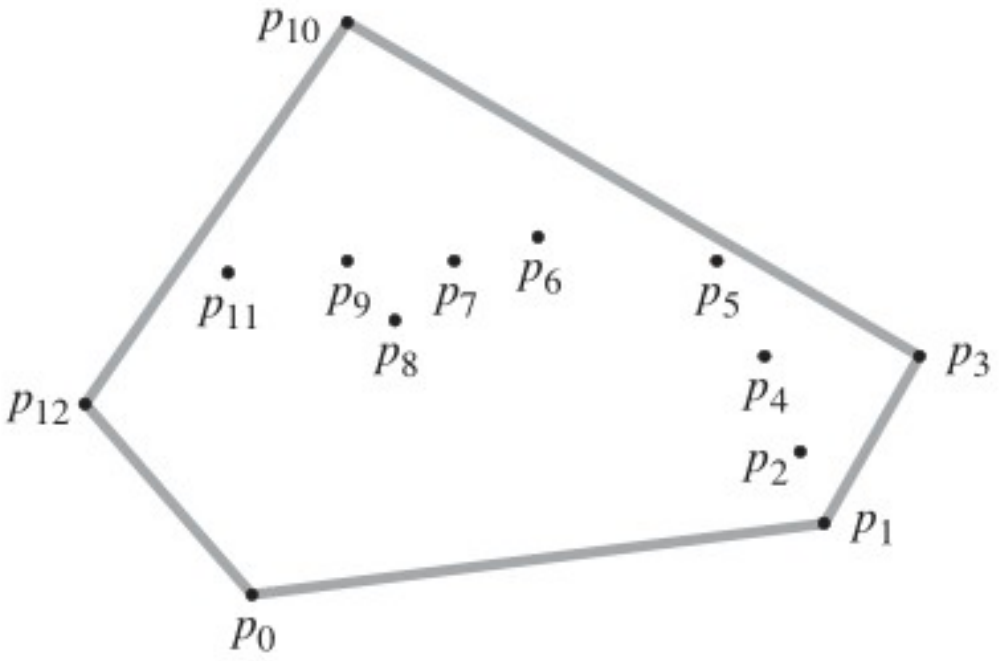
# Determining Whether Any pair of Segments Intersects

**Running time**

- If set S contains n segments, then ANY-SEGMENTS-INTERSECT runs in time O(n lg n).
- Line 1 takes O(1) time.
- Line 2 takes O(n lg n) time, using mergesort or heapsort.
- The for loop of lines 3–11 iterates at most once per event point, and so with 2n event points, the loop iterates at most 2n times.
- Each iteration takes O(lg n) time, since each red-black-tree operation takes O(lg n) time and, each intersection test takes O(1) time.
- The total time is thus O(n lg n).

# Convex Hull

- The convex hull of a set Q of points, denoted by CH(Q), is the smallest convex polygon P for which each point in Q is either on the boundary of P or in its interior.



# Convex Hull

- We implicitly assume that all points in the set Q are unique and that Q contains at least three points which are not colinear.
- Intuitively, we can think of each point in Q as being a nail sticking out from a board. The convex hull is then the shape formed by a tight rubber band that surrounds all the nails.

# Convex Hull

- We shall discuss two algorithms that compute the convex hull of a set of n points.
- Both algorithms output the vertices of the convex hull in counterclockwise order.
- The first, known as Graham's scan, runs in O(n lg n) time.
- The second, called Jarvis's march, runs in O(n h) time, where h is the number of vertices of the convex hull.

# Convex Hull

- As Figure below illustrates, every vertex of CH(Q) is a point in Q.
- Both algorithms exploit this property, deciding which vertices in Q to keep as vertices of the convex hull and which vertices in Q to reject.

# Convex Hull

- Computing the convex hull of a set of points is an interesting problem in its own right.
- Moreover, algorithms for some other computational-geometry problems start by computing a convex hull.
- Consider, for example, the two-dimensional farthest-pair problem: we are given a set of n points in the plane and wish to find the two points whose distance from each other is maximum.
- We can find the farthest pair of vertices of an n-vertex convex polygon in O(n) time.
- Thus, by computing the convex hull of the n input points in O(n lg n) time and then finding the farthest pair of the resulting convex-polygon vertices, we can find the farthest pair of points in any set of n points in O(n lg n) time.

# Graham’s Scan

- Graham’s scan solves the convex-hull problem by maintaining a stack S of candidate points.
- It pushes each point of the input set Q onto the stack one time,and it eventually pops from the stack each point that is not a vertex of CH(Q).
- When the algorithm terminates, stack S contains exactly the vertices of CH(Q), in counterclockwise order of their appearance on the boundary.

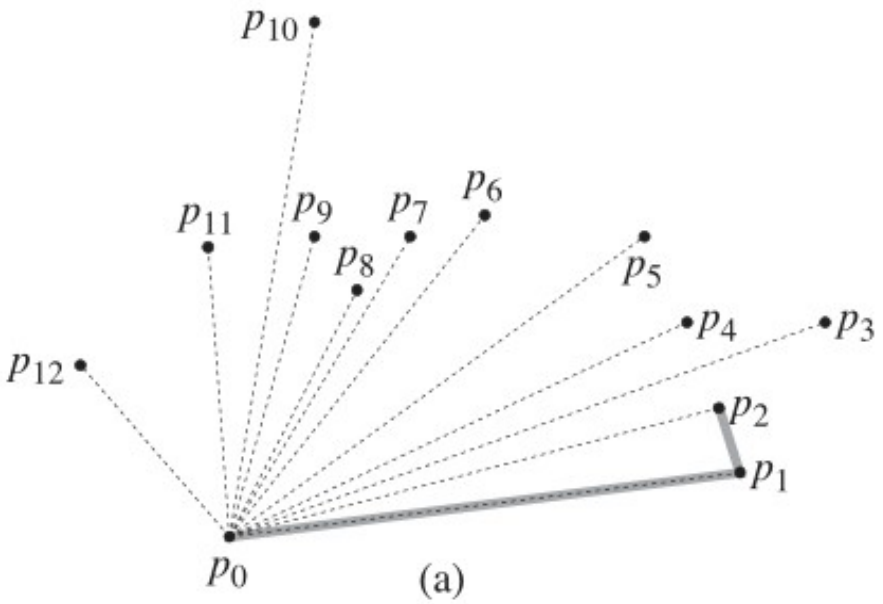
# Graham's Scan

- The procedure GRAHAM-SCAN takes as input a set Q of points, where $|Q| \geq 3$.
- It calls the functions TOP(S), which returns the point on top of stack S without changing S, and NEXT-TO-TOP(S), which returns the point one entry below the top of stack S without changing S.
- As we shall prove in a moment, the stack S returned by GRAHAM-SCAN contains, from bottom to top, exactly the vertices of CH(Q) in counterclockwise order.

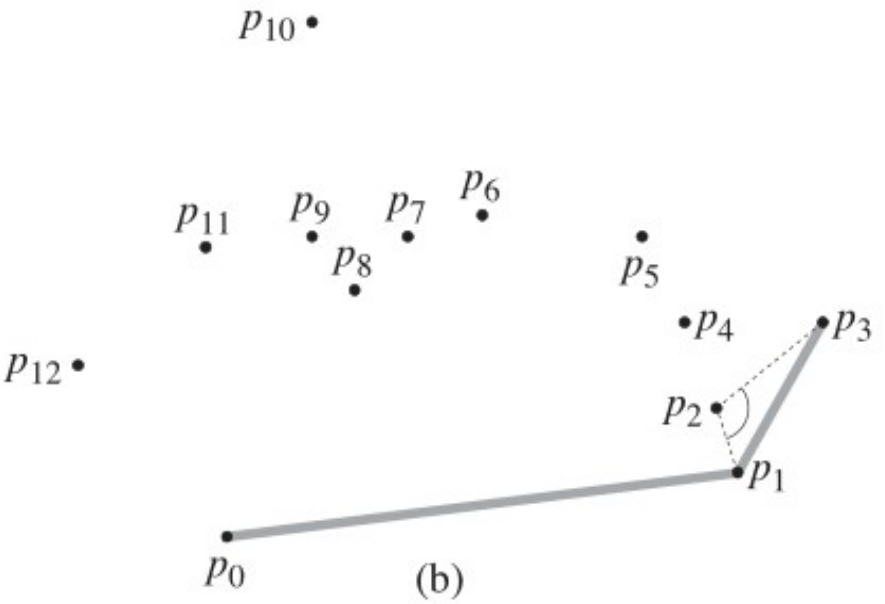
# Graham's Scan

```
GRAHAM-SCAN(Q)
 1  let p_0 be the point in Q with the minimum y-coordinate,
        or the leftmost such point in case of a tie
 2  let ⟨p_1, p_2, ..., p_m⟩ be the remaining points in Q,
        sorted by polar angle in counterclockwise order around p_0
        (if more than one point has the same angle, remove all but
        the one that is farthest from p_0)
 3  let S be an empty stack
 4  PUSH(p_0, S)
 5  PUSH(p_1, S)
 6  PUSH(p_2, S)
 7  for i = 3 to m
 8      while the angle formed by points NEXT-TO-TOP(S), TOP(S),
                and p_i makes a nonleft turn
 9          POP(S)
10      PUSH(p_i, S)
11  return S
```

# Graham's Scan

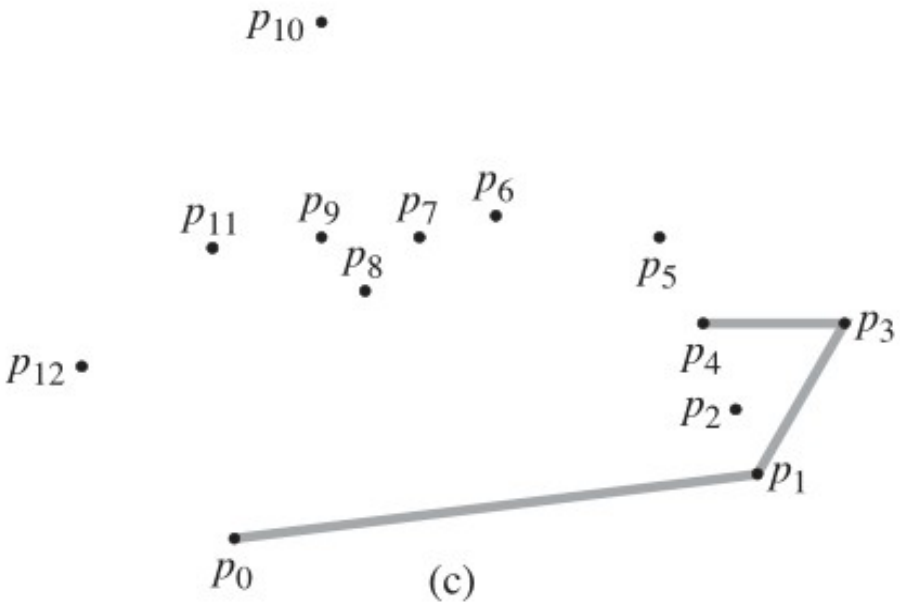


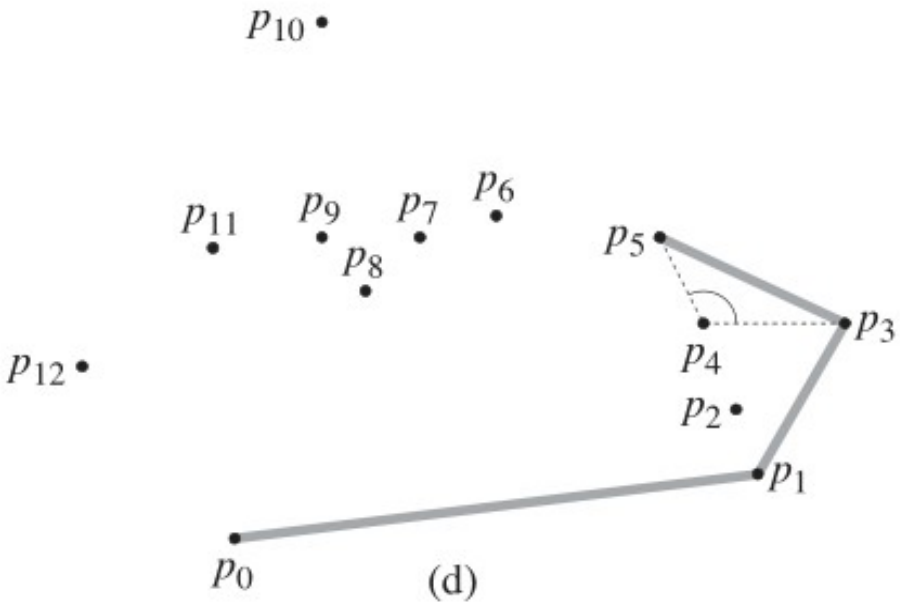

(a)The sequence ($p_1$,$p_2$, . . . $p_{12}$) of points numbered in order of increasing polar angle relative to $p_0$, and the initial stack S containing $p_0$, $p_1$, and $p_2$.




(b)–(k) Stack S after each iteration of the for loop of lines 7–10. Dashed lines show nonleft turns, which cause points to be popped from the stack.

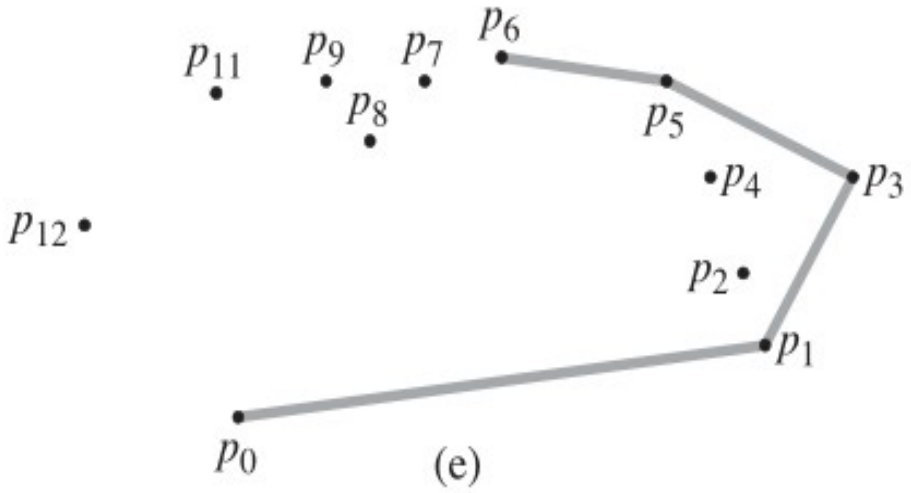
# Graham's Scan

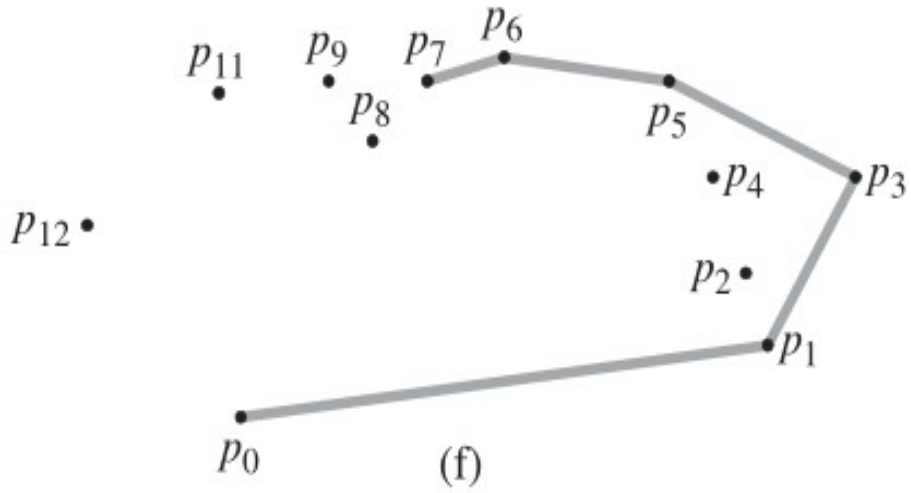



(b)–(k) Stack S after each iteration of the for loop of lines 7–10. Dashed lines show nonleft turns, which cause points to be popped from the stack.

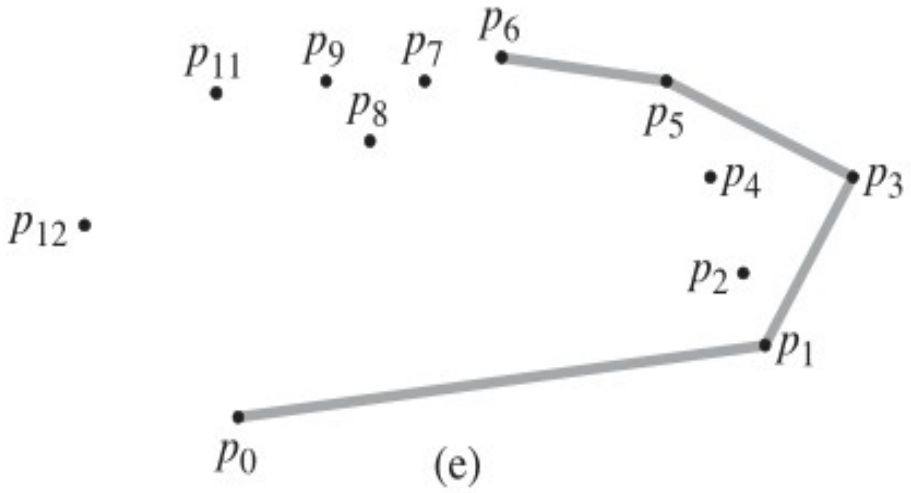
# Graham's Scan

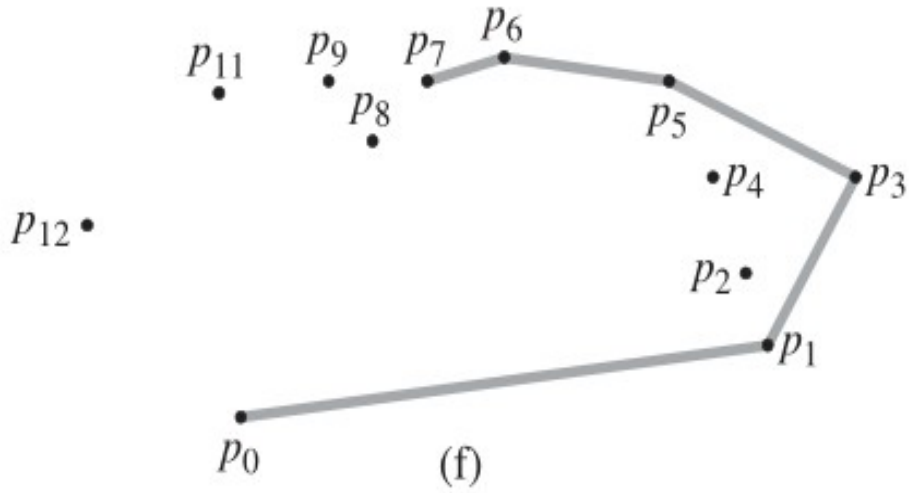



(b)–(k) Stack S after each iteration of the for loop of lines 7–10. Dashed lines show nonleft turns, which cause points to be popped from the stack.

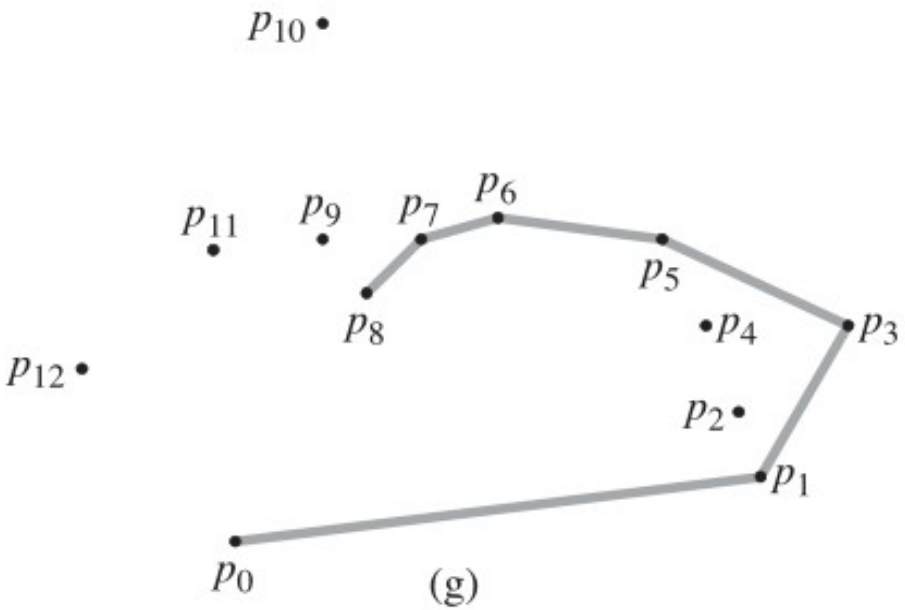
# Graham's Scan

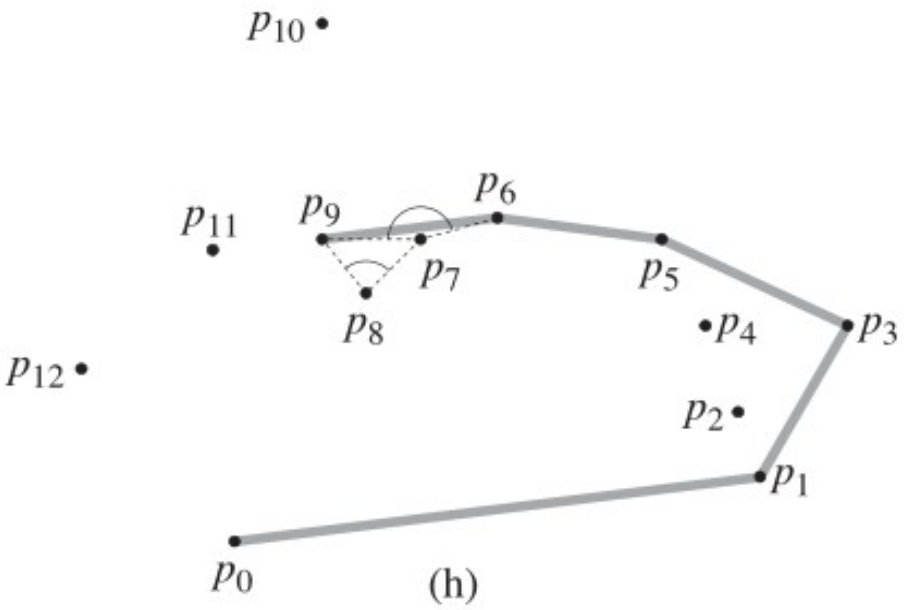



(b)–(k) Stack S after each iteration of the for loop of lines 7–10. Dashed lines show nonleft turns, which cause points to be popped from the stack.

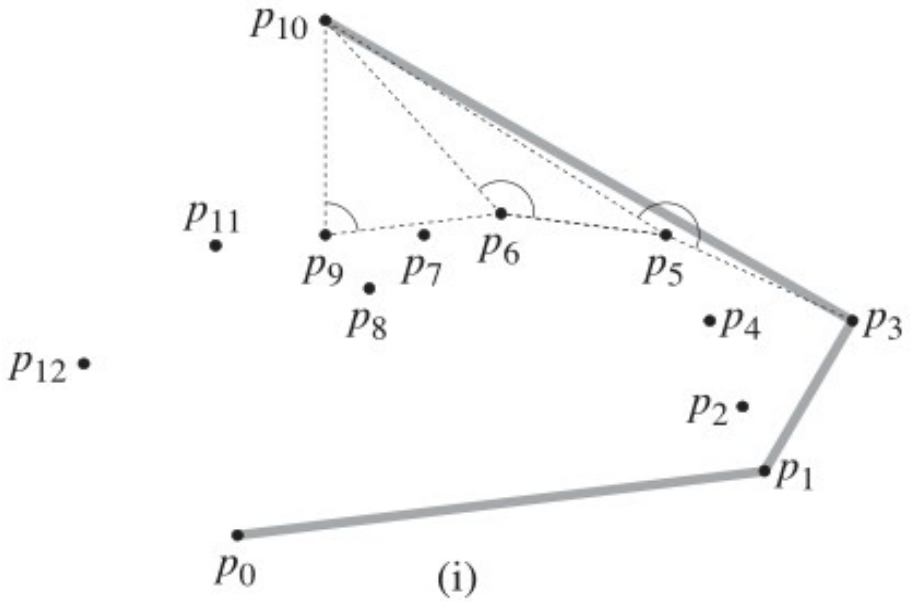
# Graham's Scan

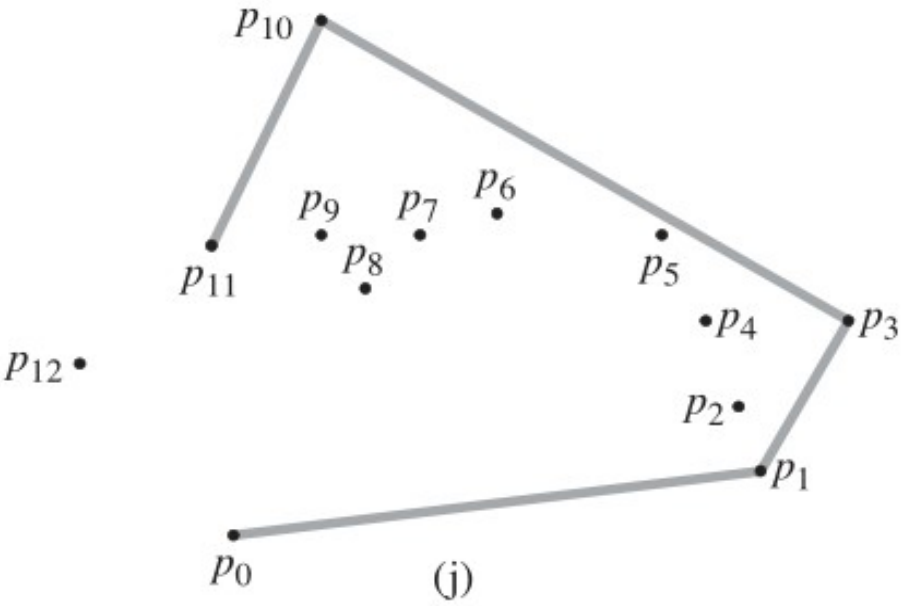



(b)–(k) Stack S after each iteration of the for loop of lines 7–10. Dashed lines show nonleft turns, which cause points to be popped from the stack.

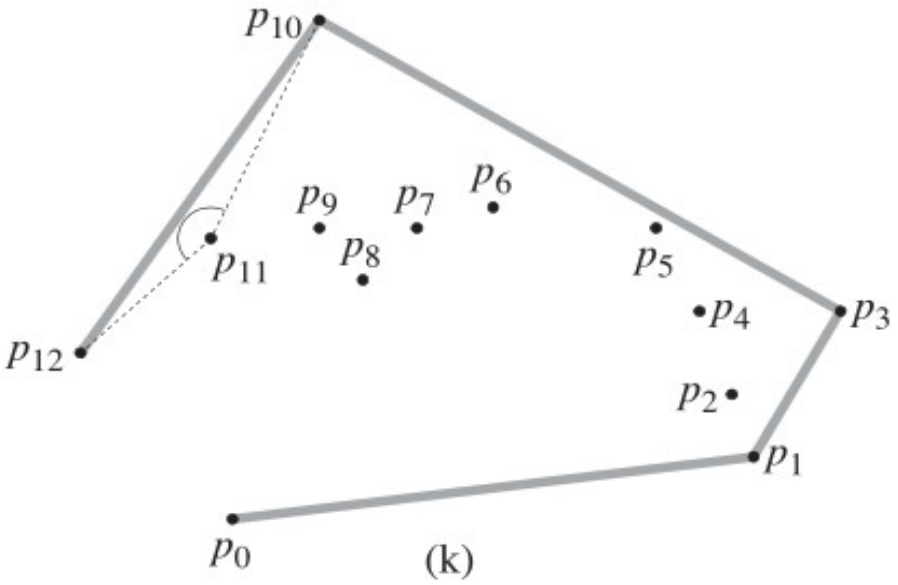
# Graham's Scan



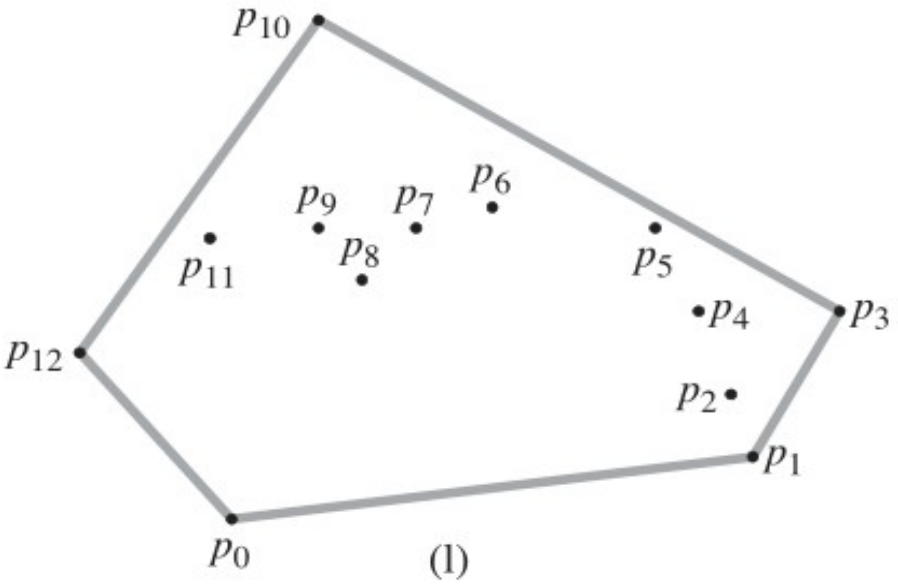




(b)–(k) Stack S after each iteration of the for loop of lines 7–10. Dashed lines show nonleft turns, which cause points to be popped from the stack.

# Graham's Scan




(b)–(k) Stack S after each iteration of the for loop of lines 7–10. Dashed lines show nonleft turns, which cause points to be popped from the stack.




(l) The convex hull returned by the procedure

# Jarvis’s march

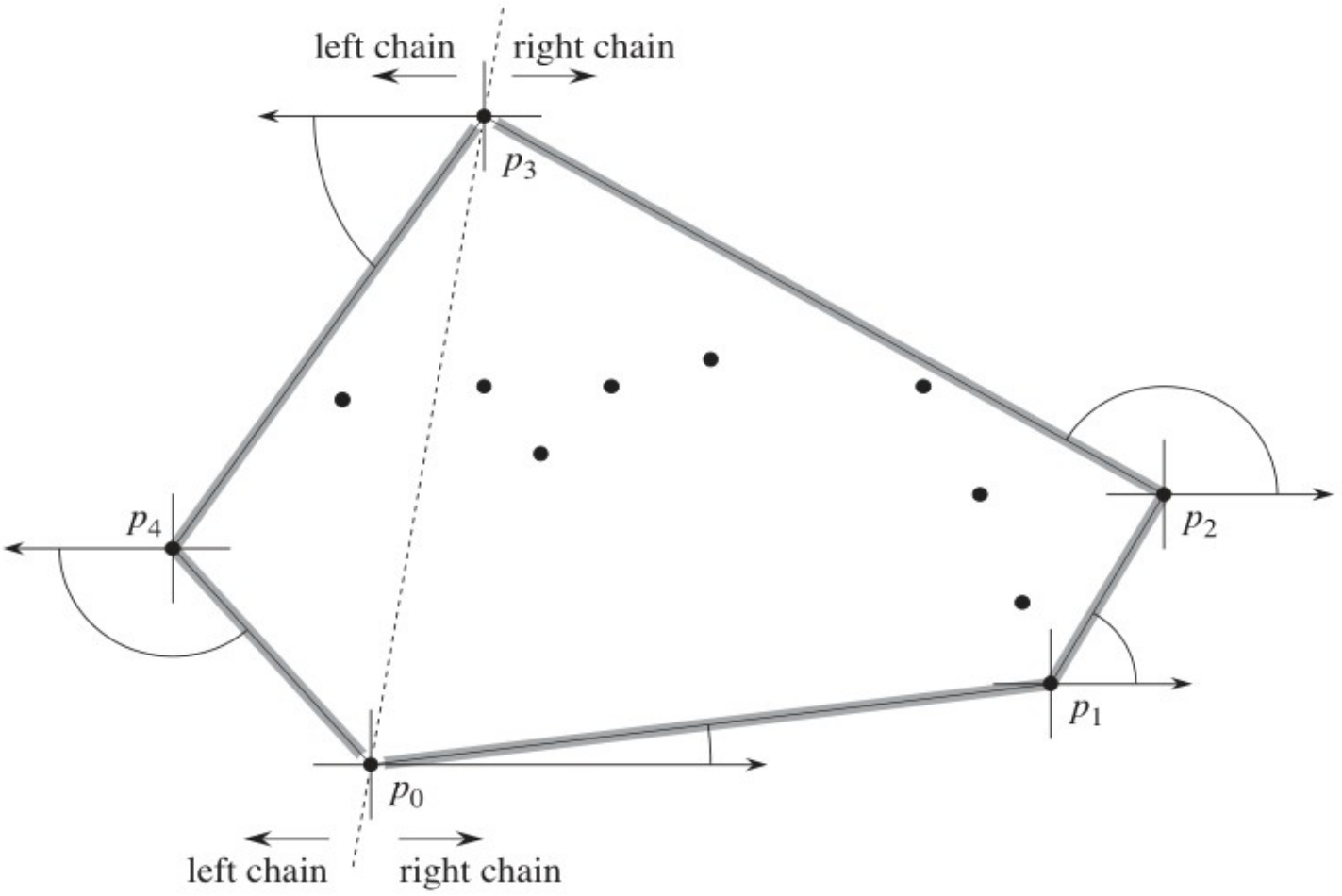
- Jarvis’s march computes the convex hull of a set Q of points by a technique known as package wrapping (or gift wrapping).
- The algorithm runs in time O(n h), where h is the number of vertices of CH(Q).

# Jarvis's march

- Intuitively, Jarvis's march simulates wrapping a taut piece of paper around the set Q.
- We start by taping the end of the paper to the lowest point in the set, that is, to the same point $p_0$ with which we start Graham's scan. We know that this point must be a vertex of the convex hull.
- We pull the paper to the right to make it taut, and then we pull it higher until it touches a point.
- This point must also be a vertex of the convex hull.
- Keeping the paper taut, we continue in this way around the set of vertices until we come back to our original point $p_0$.

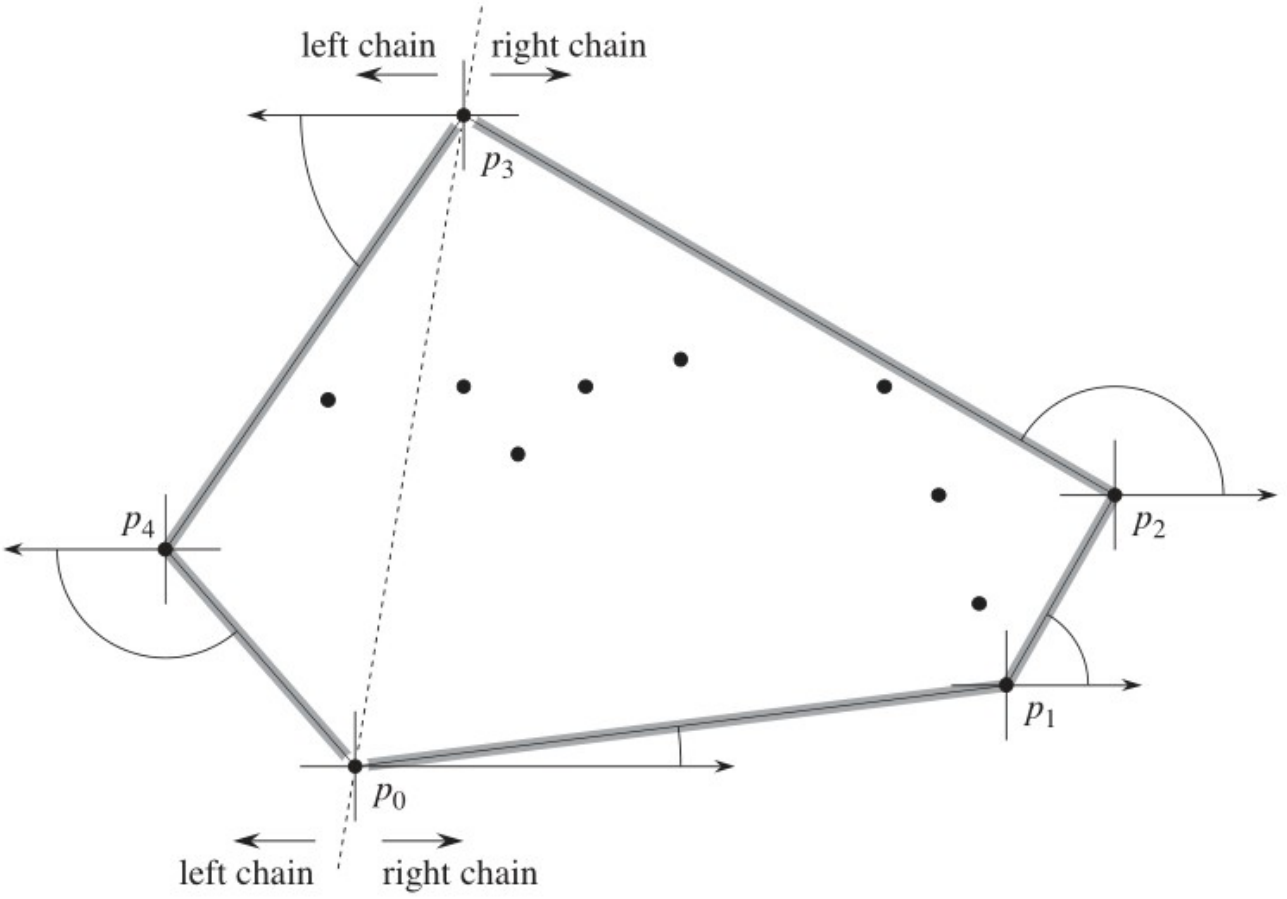
# Jarvis's march

- More formally, Jarvis's march builds a sequence $H = (p_0, p_1 \ldots p_h)$ of the vertices of CH(Q). We start with $p_0$. As Figure below shows, the next vertex $p_1$ in the convex hull has the smallest polar angle with respect to $p_0$. (In case of ties, we choose the point farthest from $p_0$.)



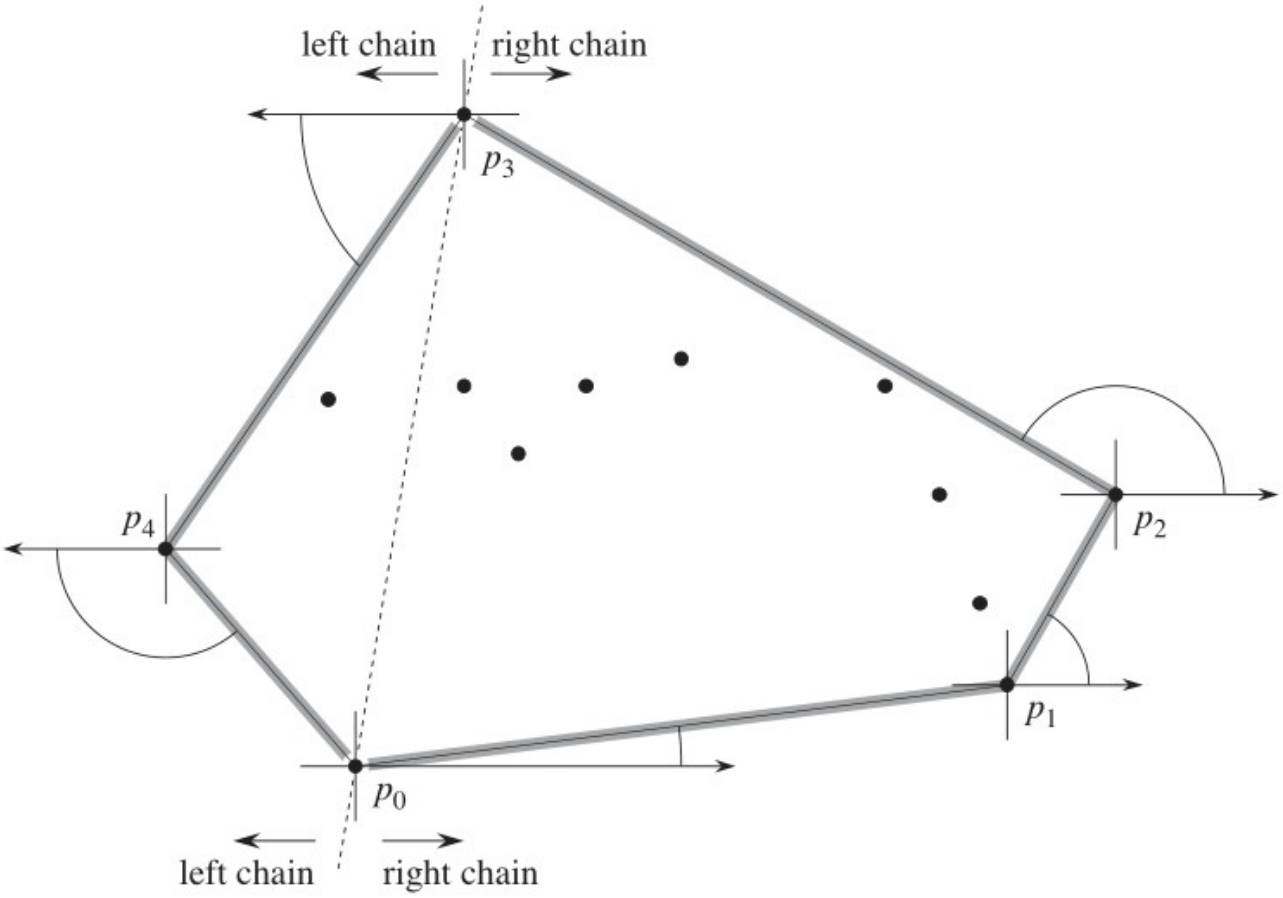
# Jarvis's march

- Similarly, $p_2$ has the smallest polar angle with respect to $p_1$, and so on.
- When we reach the highest vertex, say $p_k$ (breaking ties by choosing the farthest such vertex), we have constructed, as Figure below shows, the right chain of CH(Q).



# Jarvis's march

- To construct the left chain, we start at $p_k$ and choose $p_{k+1}$ as the point with the smallest polar angle with respect to $p_k$, but from the negative x-axis.
- We continue on, forming the left chain by taking polar angles from the negative x-axis, until we come back to our original vertex $p_0$.



# Jarvis’s march

- If implemented properly, Jarvis’s march has a running time of $O(n h)$
- For each of the $h$ vertices of $CH(Q)$, we find the vertex with the minimum polar angle.
- Each comparison between polar angles takes $O(1)$ time.
- We can compute the minimum of $n$ values in $O(n)$ time if each comparison takes $O(1)$ time. Thus, Jarvis’s march takes $O(n h)$ time.

# Voronoi Diagrams

- It’s a simple mathematical intricacy that often arises in nature, and can also be a very practical tool in science.
- It’s named after the famous Russian mathematician Georgy Voronoi.
- We can also refer to it as the Voronoi tesselation, Voronoi decomposition, or Voronoi partition.

# Voronoi Diagrams

- In the illustration below, we have a set of 40 randomly placed orange points on a 2-dimensional plane, along with its Voronoi diagram. The regions or tiles that we get as a result are called Voronoi cells:

# Voronoi Diagrams

- **Input description:** A set S of points $p_1, \ldots, p_n$.
- **Problem description:** Decompose space into regions around each point such that all points in the region around $p_i$ are closer to $p_i$ than any other point in S.
- **Discussion:** Voronoi diagrams represent the region of influence around each of a given set of sites. If these sites represent the locations of McDonald's restaurants, the Voronoi diagram partitions space into cells around each restaurant. For each person living in a particular cell, the defining McDonald's represents the closest place to get a Big Mac.

# Voronoi Diagrams

**Applications:**

- Nearest neighbor search – Finding the nearest neighbor of query point q from among a fixed set of points S is simply a matter of determining the cell in the Voronoi diagram of S that contains q.
- Facility location – Suppose McDonald's wants to open another restaurant. To minimize interference with existing McDonald's, it should be located as far away from the closest restaurant as possible. This location is always at a vertex of the Voronoi diagram, and can be found in a linear-time search through all the Voronoi vertices.

# Voronoi Diagrams

**Applications:**

- Largest empty circle – Suppose you needed to obtain a large, contiguous, undeveloped piece of land on which to build a factory. The same condition used to select McDonald's locations is appropriate for other undesirable facilities, namely that they be as far as possible from any relevant sites of interest. A Voronoi vertex defines the center of the largest empty circle among the points.
- Path planning – If the sites of S are the centers of obstacles we seek to avoid, the edges of the Voronoi diagram define the possible channels that maximize the distance to the obstacles. Thus the "safest" path among the obstacles will stick to the edges of the Voronoi diagram.

# Voronoi Diagrams

- Each edge of a Voronoi diagram is a segment of the perpendicular bisector of two points in S, since this is the line that partitions the plane between the points.
- The conceptually simplest method to construct Voronoi diagrams is randomized incremental construction.
- To add another site to the diagram, we locate the cell that contains it and add the perpendicular bisectors separating this new site from all sites defining impacted regions.
- If the sites are inserted in random order, only a small number of regions are likely to be impacted with each insertion.

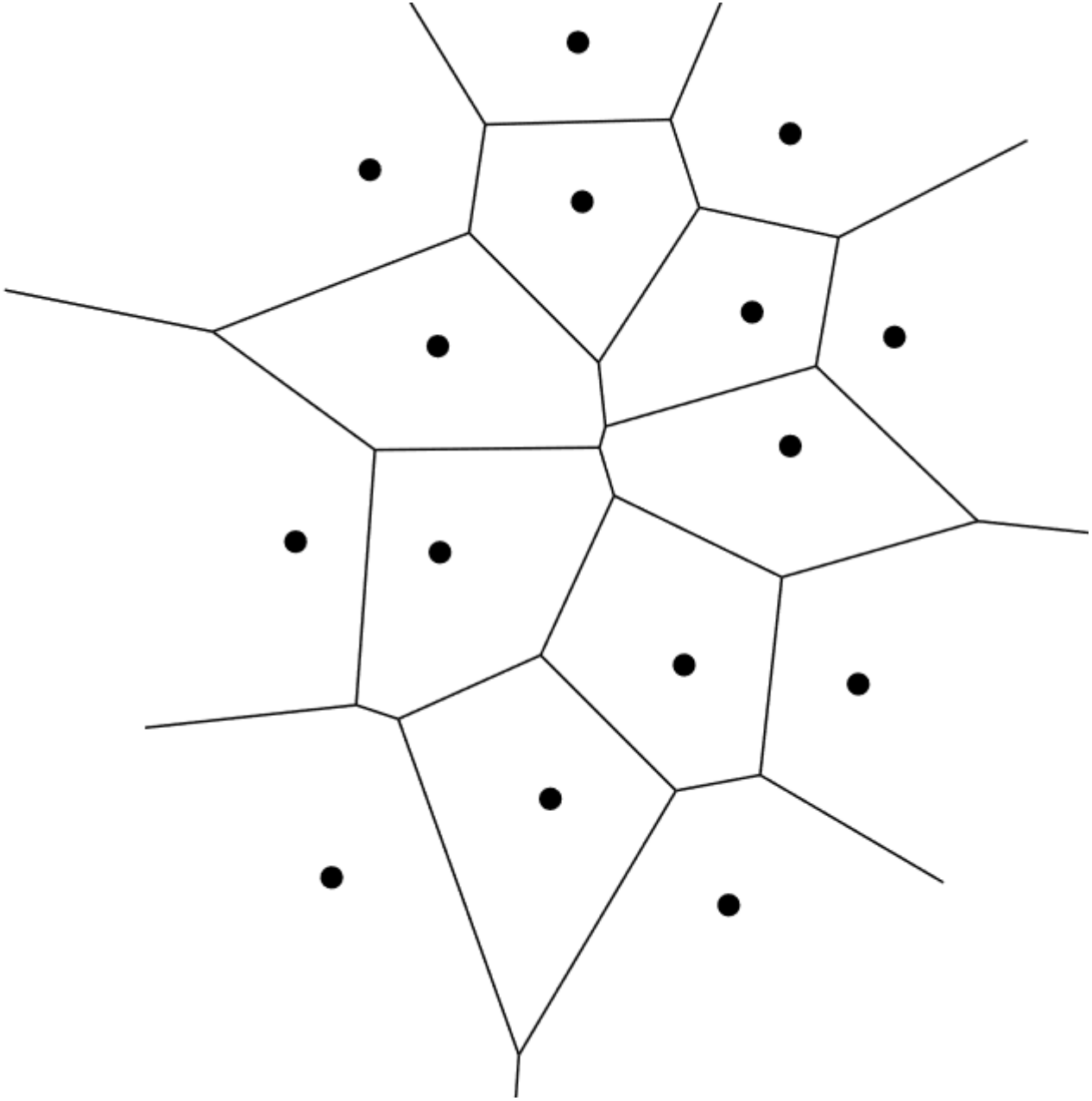
# Voronoi Diagrams

## INCREMENTAL ALGORITHM

# Voronoi Diagrams

## INCREMENTAL ALGORITHM

Starting with the Voronoi diagram of $\{p_1, \ldots, p_i\}$...

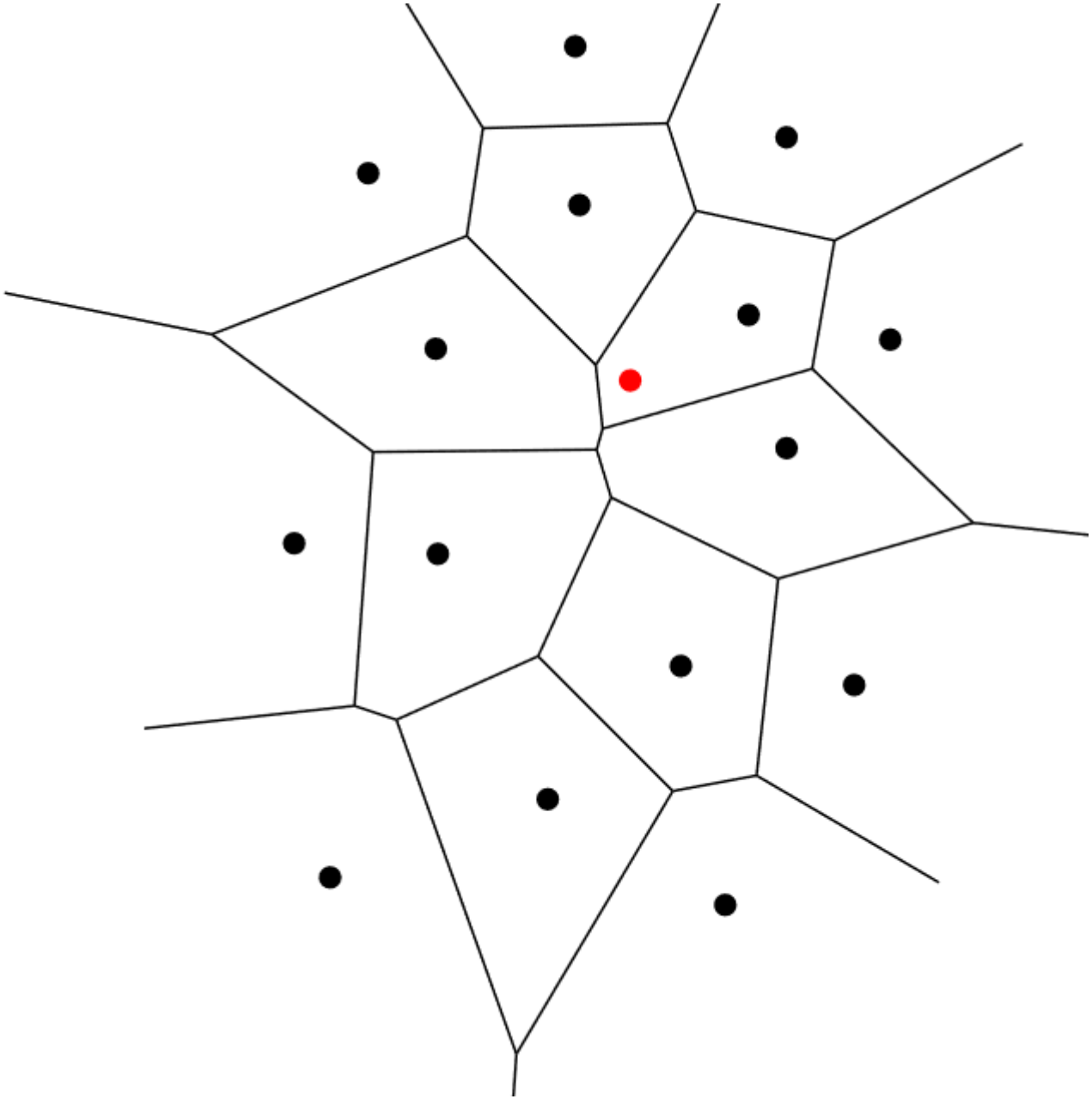


# Voronoi Diagrams

## INCREMENTAL ALGORITHM

Starting with the Voronoi diagram of $\{p_1, \ldots, p_i\}$...

... add point $p_{i+1}$



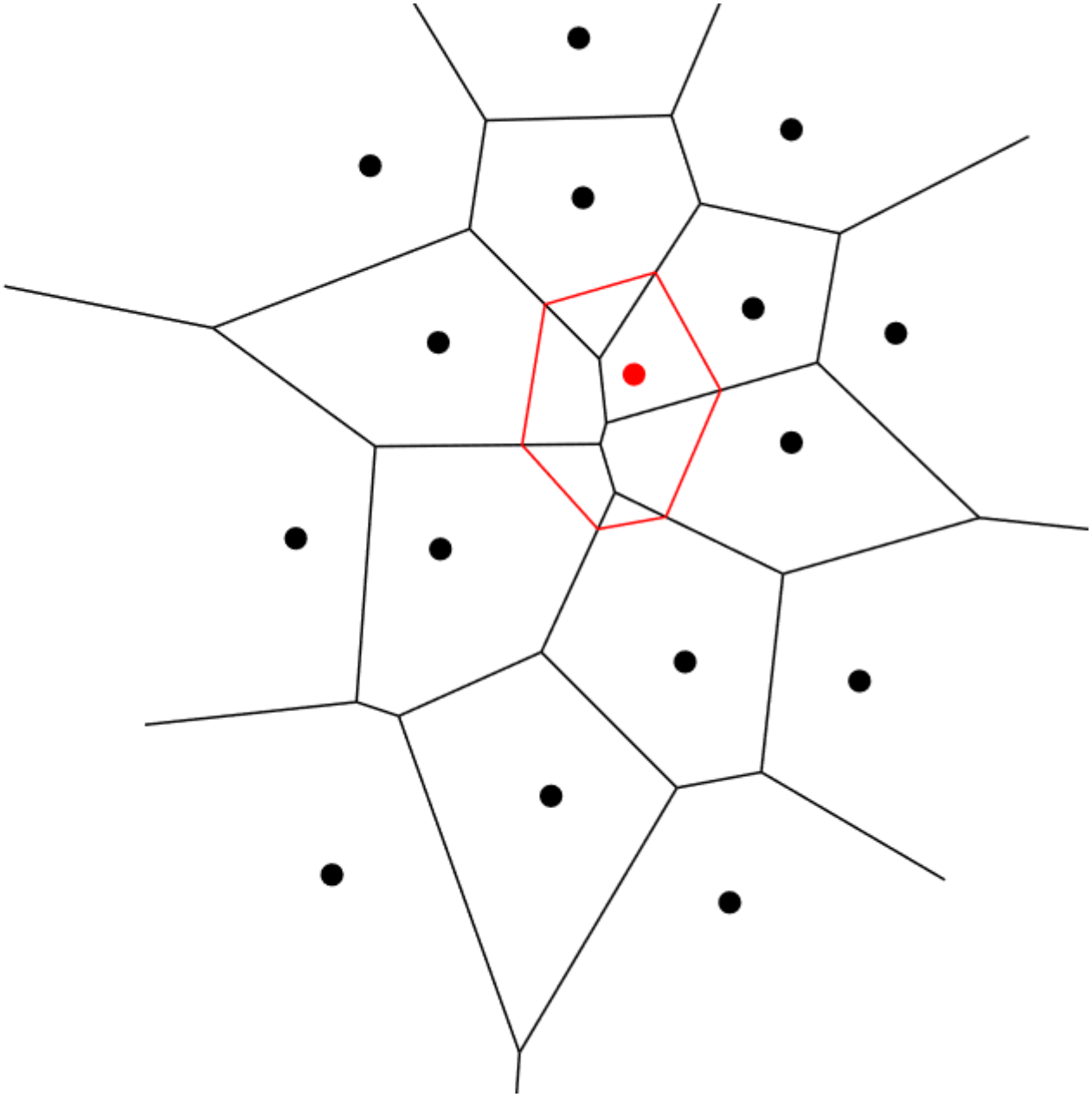
# Voronoi Diagrams

## INCREMENTAL ALGORITHM

Starting with the Voronoi diagram of $\{p_1, \ldots, p_i\}$...

... add point $p_{i+1}$

... compute its region



# Voronoi Diagrams

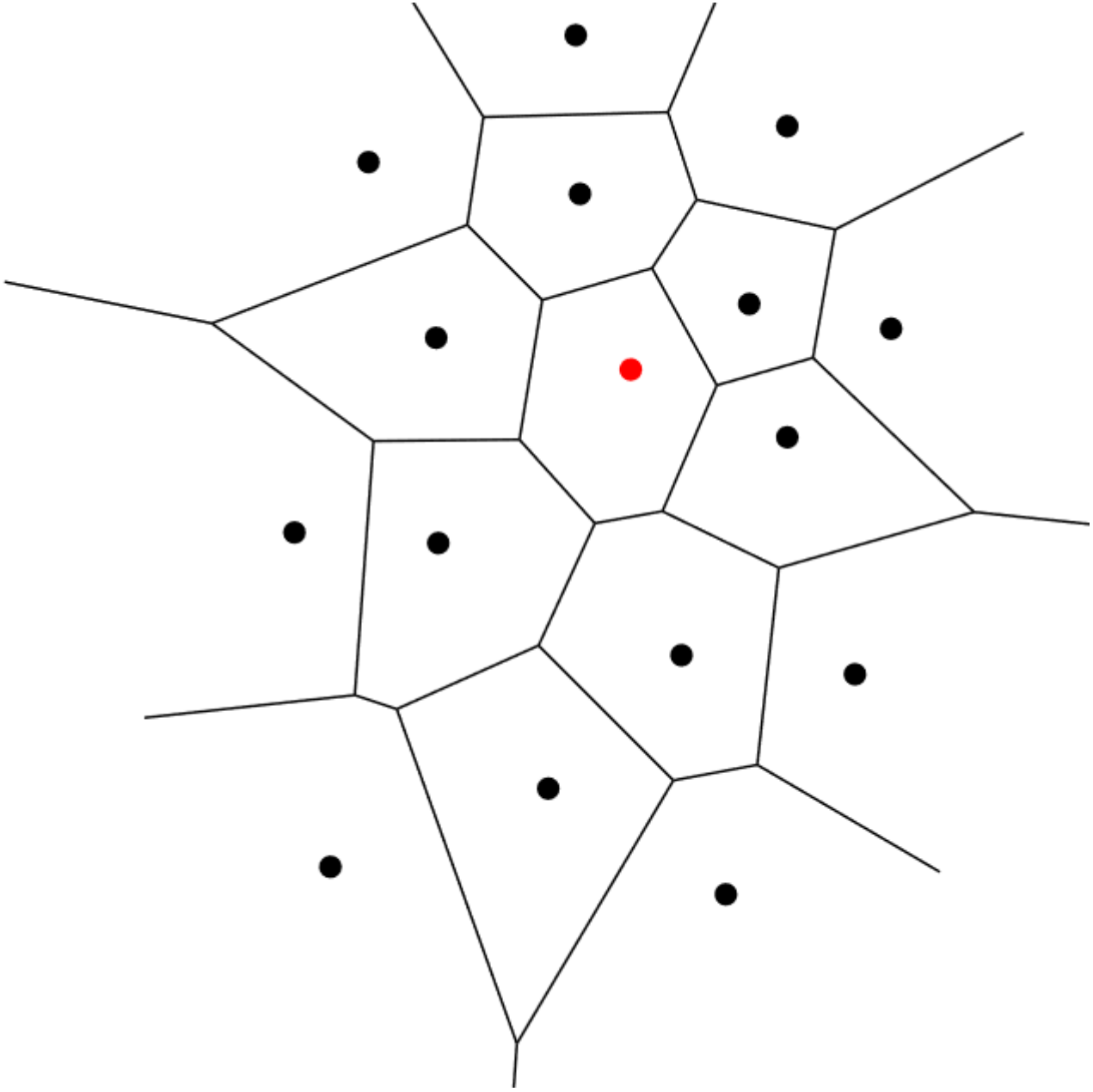
## INCREMENTAL ALGORITHM

Starting with the Voronoi diagram of $\{p_1, \ldots, p_i\}$...

... add point $p_{i+1}$

... compute its region

... and prune the initial diagram.



# Voronoi Diagrams

## INCREMENTAL ALGORITHM

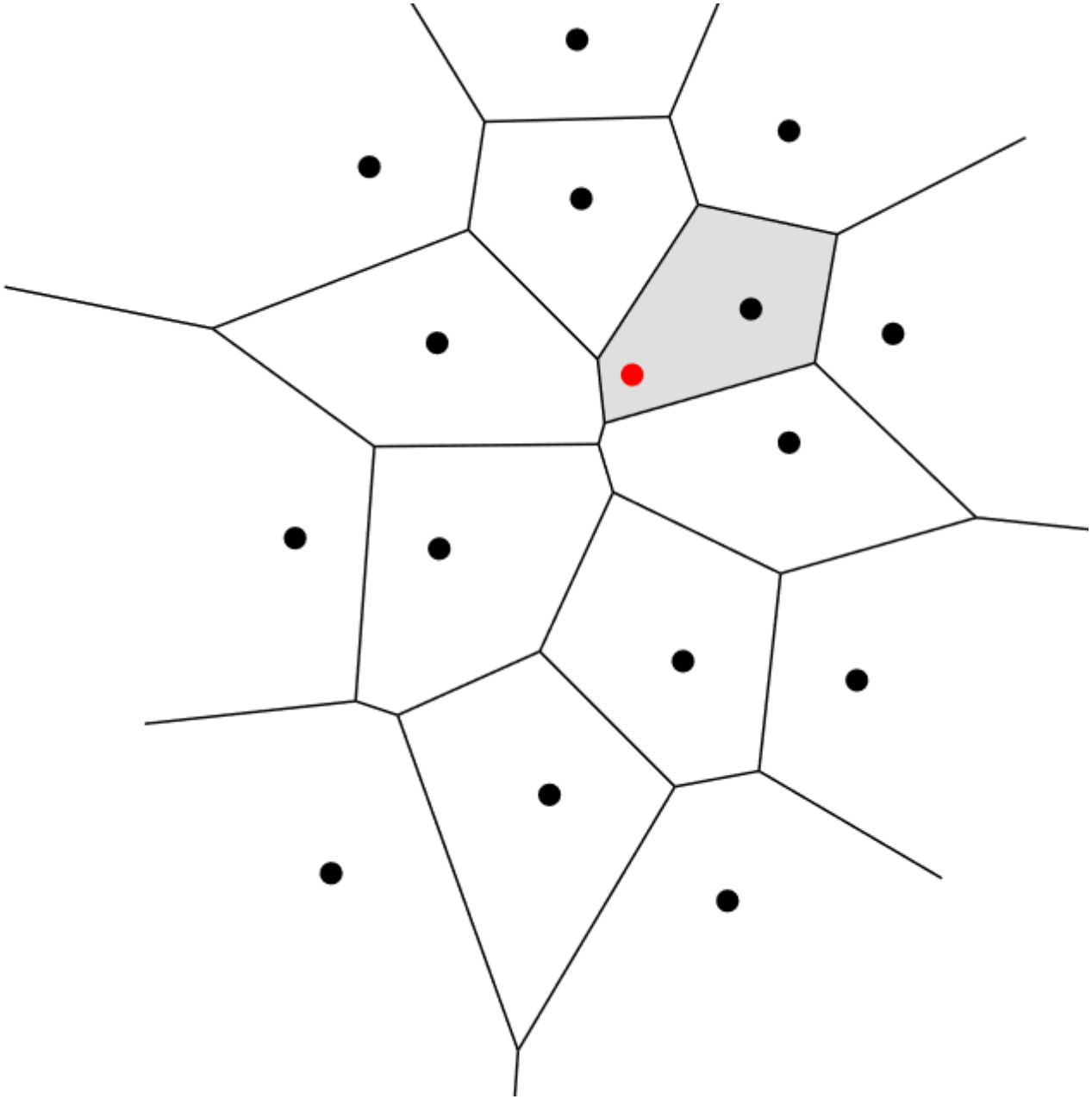
Starting with the Voronoi diagram of $\{p_1, \ldots, p_i\}$...

... add point $p_{i+1}$

Explore all candidates to find the site $p_j$ $(1 \leq j \leq i)$ closest to $p_{i+1}$.

... compute its region

... and prune the initial diagram.



# Voronoi Diagrams

## INCREMENTAL ALGORITHM

Starting with the Voronoi diagram of $\{p_1, \ldots, p_i\}$...

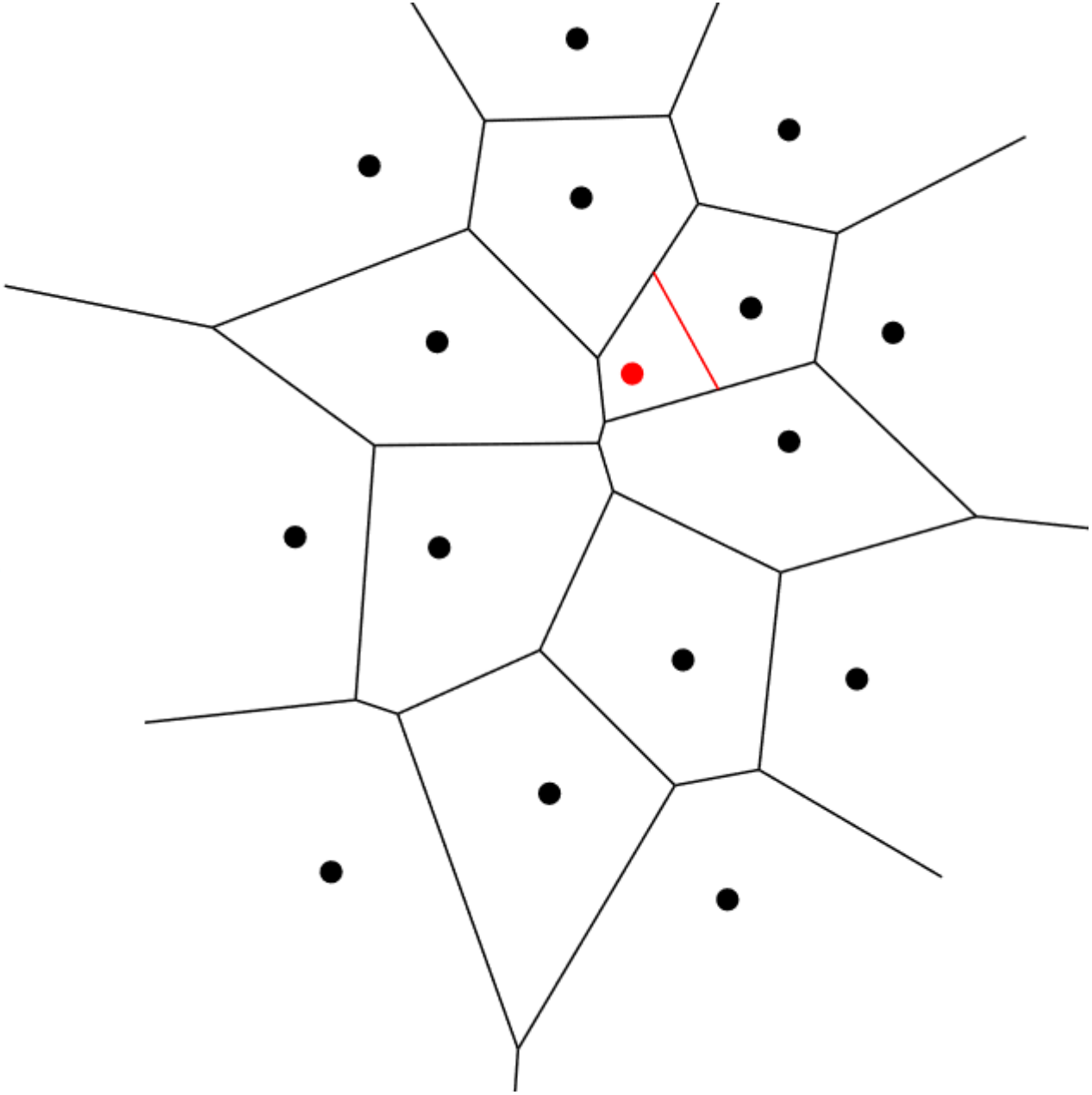
... add point $p_{i+1}$

Explore all candidates to find the site $p_j$ $(1 \leq j \leq i)$ closest to $p_{i+1}$.

... compute its region

Build its boundary starting from bisector $b_{i+1,j}$.

... and prune the initial diagram.



# Voronoi Diagrams

## INCREMENTAL ALGORITHM

Starting with the Voronoi diagram of $\{p_1, \ldots, p_i\}$...

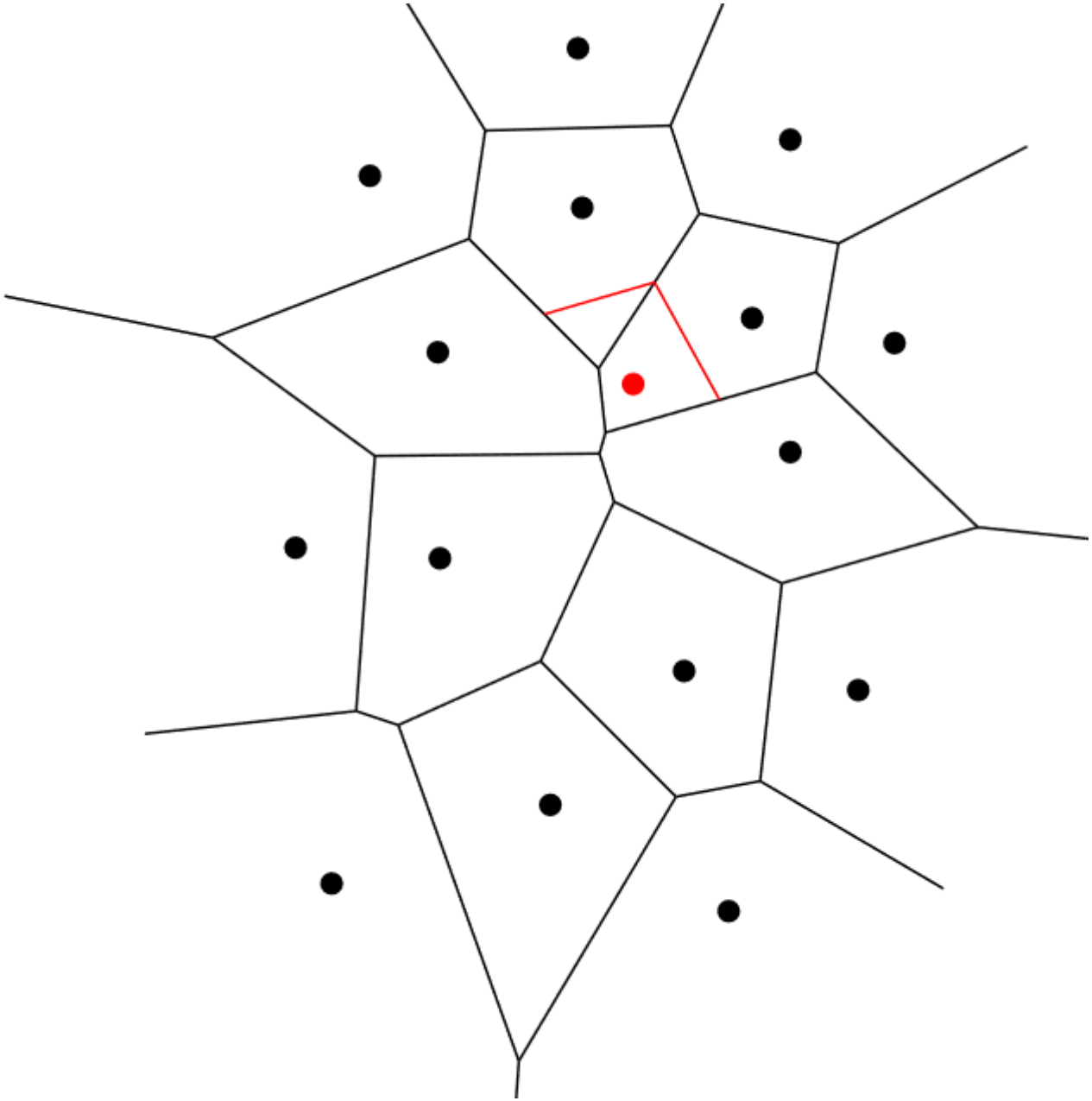
... add point $p_{i+1}$

Explore all candidates to find the site $p_j$ $(1 \leq j \leq i)$ closest to $p_{i+1}$.

... compute its region

Build its boundary starting from bisector $b_{i+1,j}$.

... and prune the initial diagram.



# Voronoi Diagrams

## INCREMENTAL ALGORITHM

Starting with the Voronoi diagram of $\{p_1, \ldots, p_i\}$...

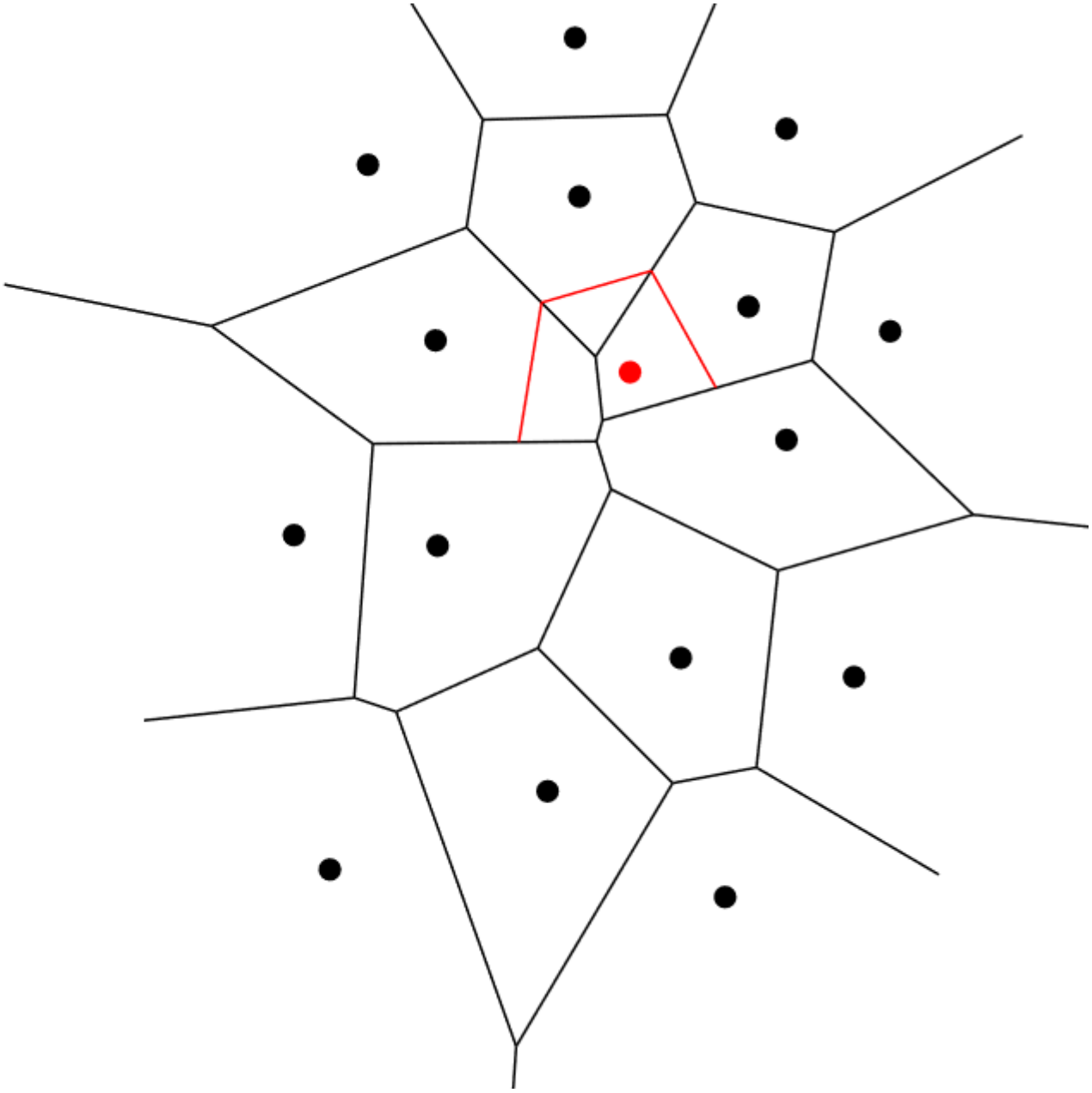
... add point $p_{i+1}$

Explore all candidates to find the site $p_j$ $(1 \leq j \leq i)$ closest to $p_{i+1}$.

... compute its region

Build its boundary starting from bisector $b_{i+1,j}$.

... and prune the initial diagram.



# Voronoi Diagrams

## INCREMENTAL ALGORITHM

Starting with the Voronoi diagram of $\{p_1, \ldots, p_i\}$...

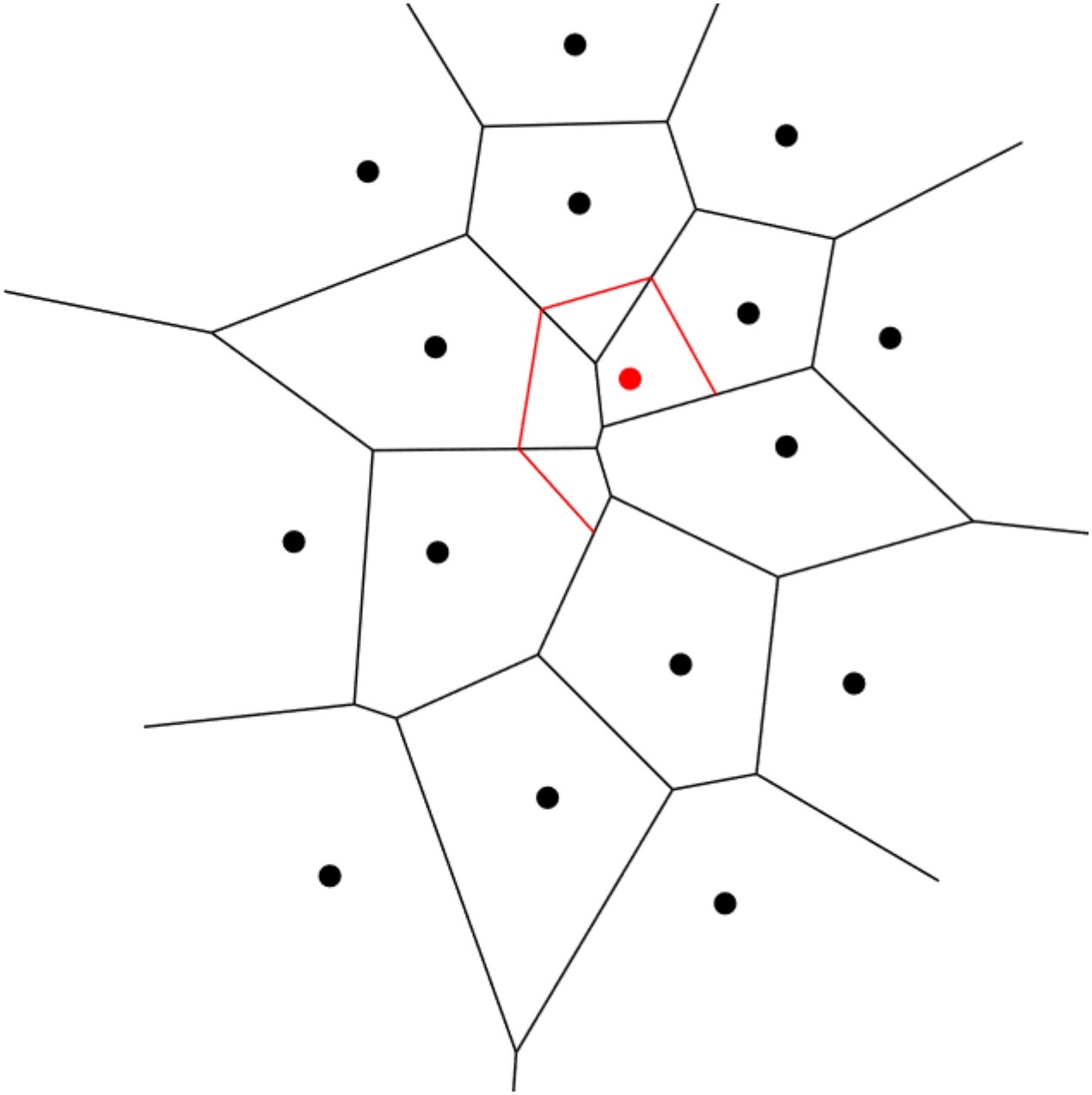
... add point $p_{i+1}$

Explore all candidates to find the site $p_j$ $(1 \leq j \leq i)$ closest to $p_{i+1}$.

... compute its region

Build its boundary starting from bisector $b_{i+1,j}$.

... and prune the initial diagram.



# Voronoi Diagrams

## INCREMENTAL ALGORITHM

Starting with the Voronoi diagram of $\{p_1, \ldots, p_i\}$...

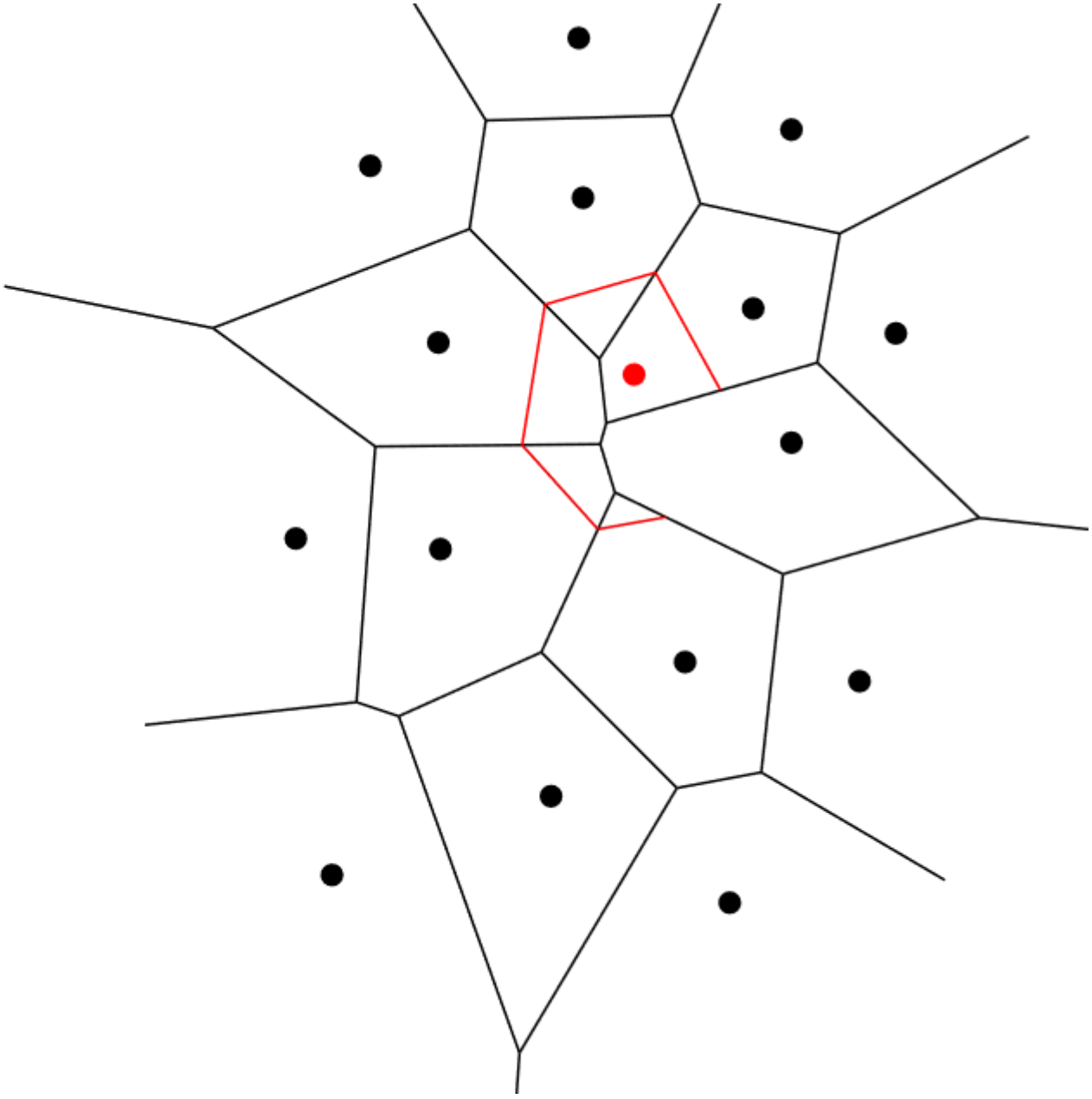
... add point $p_{i+1}$

Explore all candidates to find the site $p_j$ $(1 \leq j \leq i)$ closest to $p_{i+1}$.

... compute its region

Build its boundary starting from bisector $b_{i+1,j}$.

... and prune the initial diagram.



# Voronoi Diagrams

## INCREMENTAL ALGORITHM

Starting with the Voronoi diagram of $\{p_1, \ldots, p_i\}$...

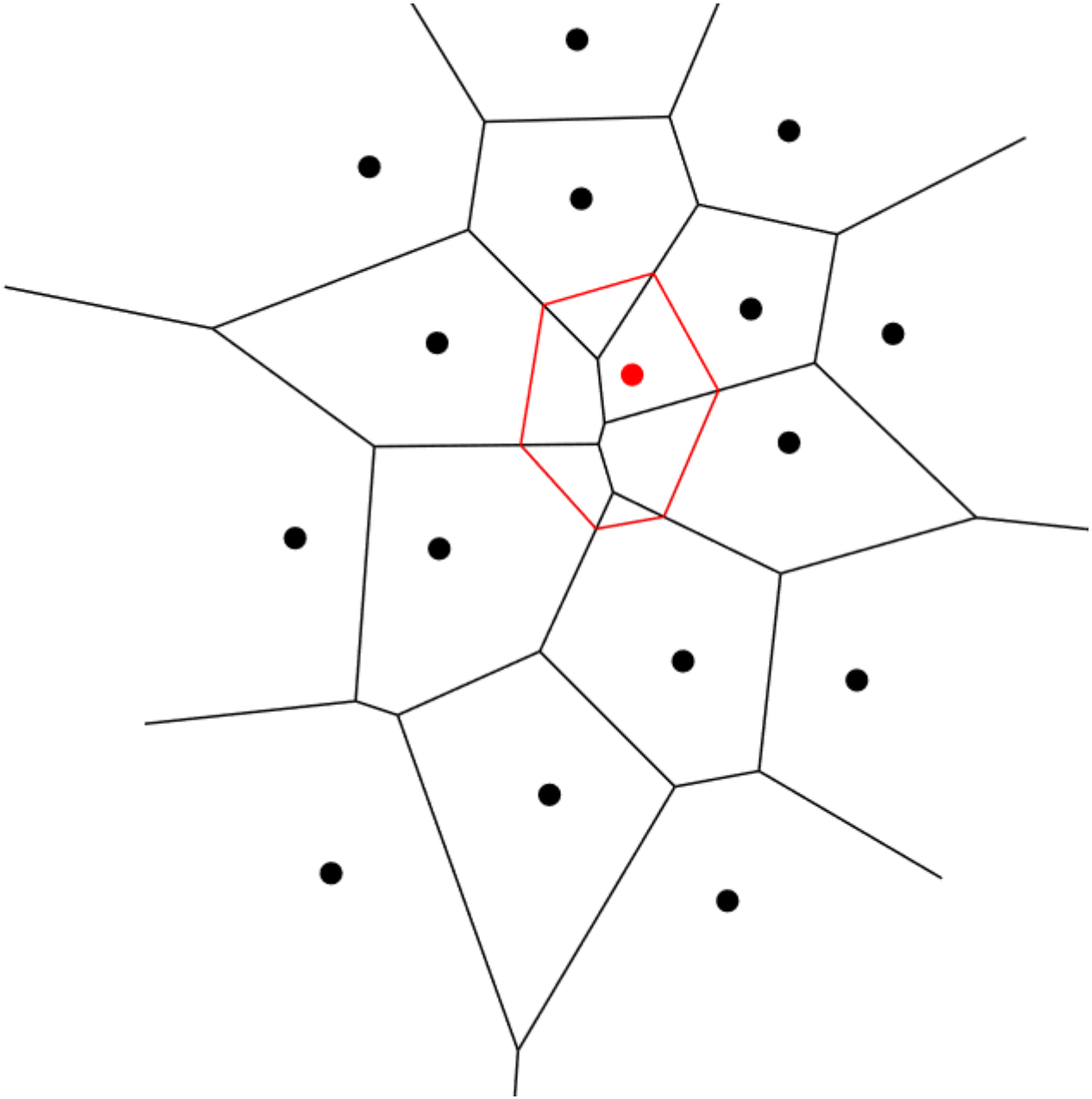
... add point $p_{i+1}$

Explore all candidates to find the site $p_j$ $(1 \leq j \leq i)$ closest to $p_{i+1}$.

... compute its region

Build its boundary starting from bisector $b_{i+1,j}$.

... and prune the initial diagram.



# Voronoi Diagrams

## INCREMENTAL ALGORITHM

Starting with the Voronoi diagram of $\{p_1, \ldots, p_i\}$...

... add point $p_{i+1}$

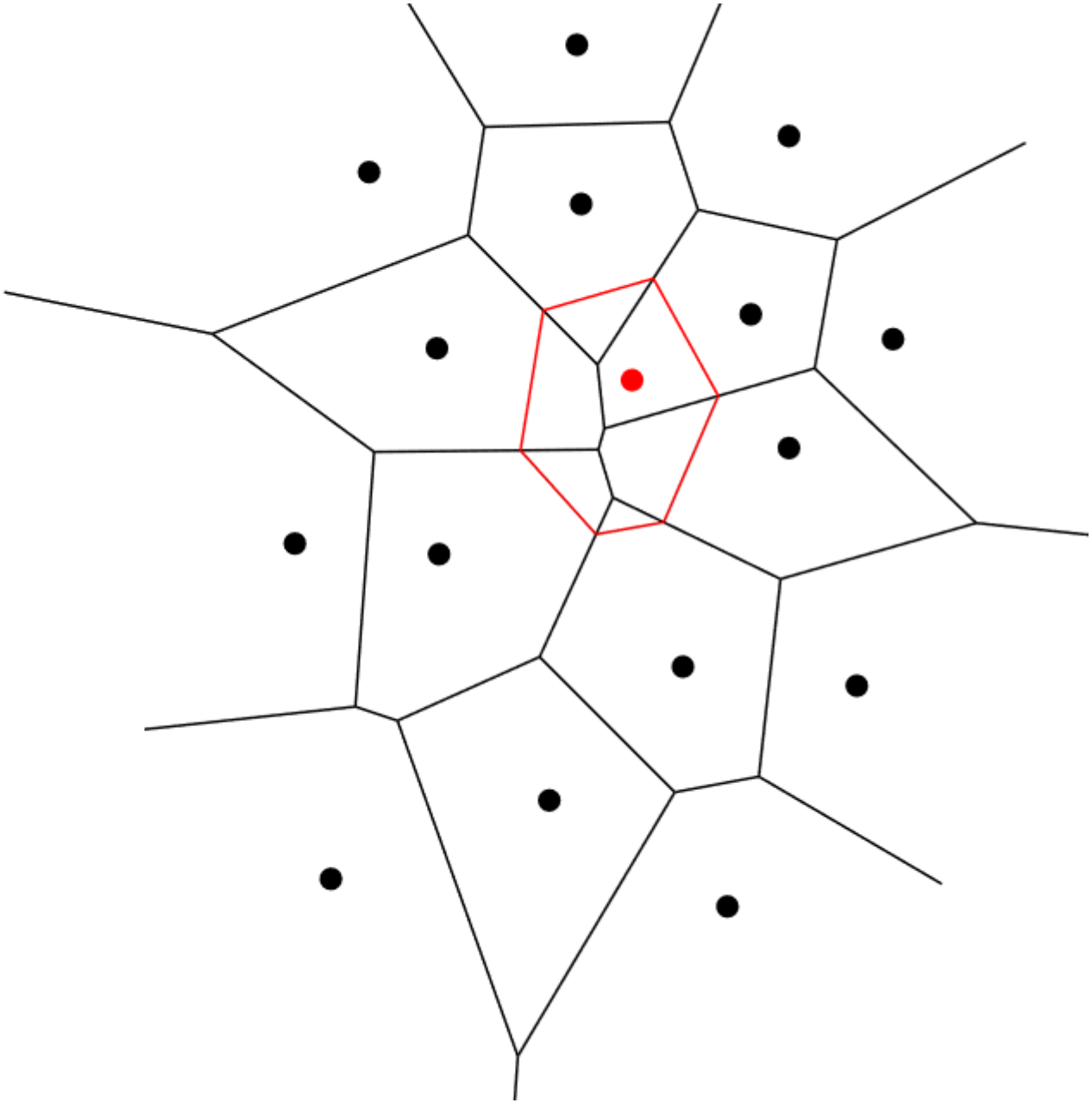
Explore all candidates to find the site $p_j$ $(1 \leq j \leq i)$ closest to $p_{i+1}$.

... compute its region

Build its boundary starting from bisector $b_{i+1,j}$.

... and prune the initial diagram.

While building the Voronoi region of $p_{i+1}$, update the DCEL.



# Voronoi Diagrams

## INCREMENTAL ALGORITHM

Starting with the Voronoi diagram of $\{p_1, \ldots, p_i\}$...

... add point $p_{i+1}$

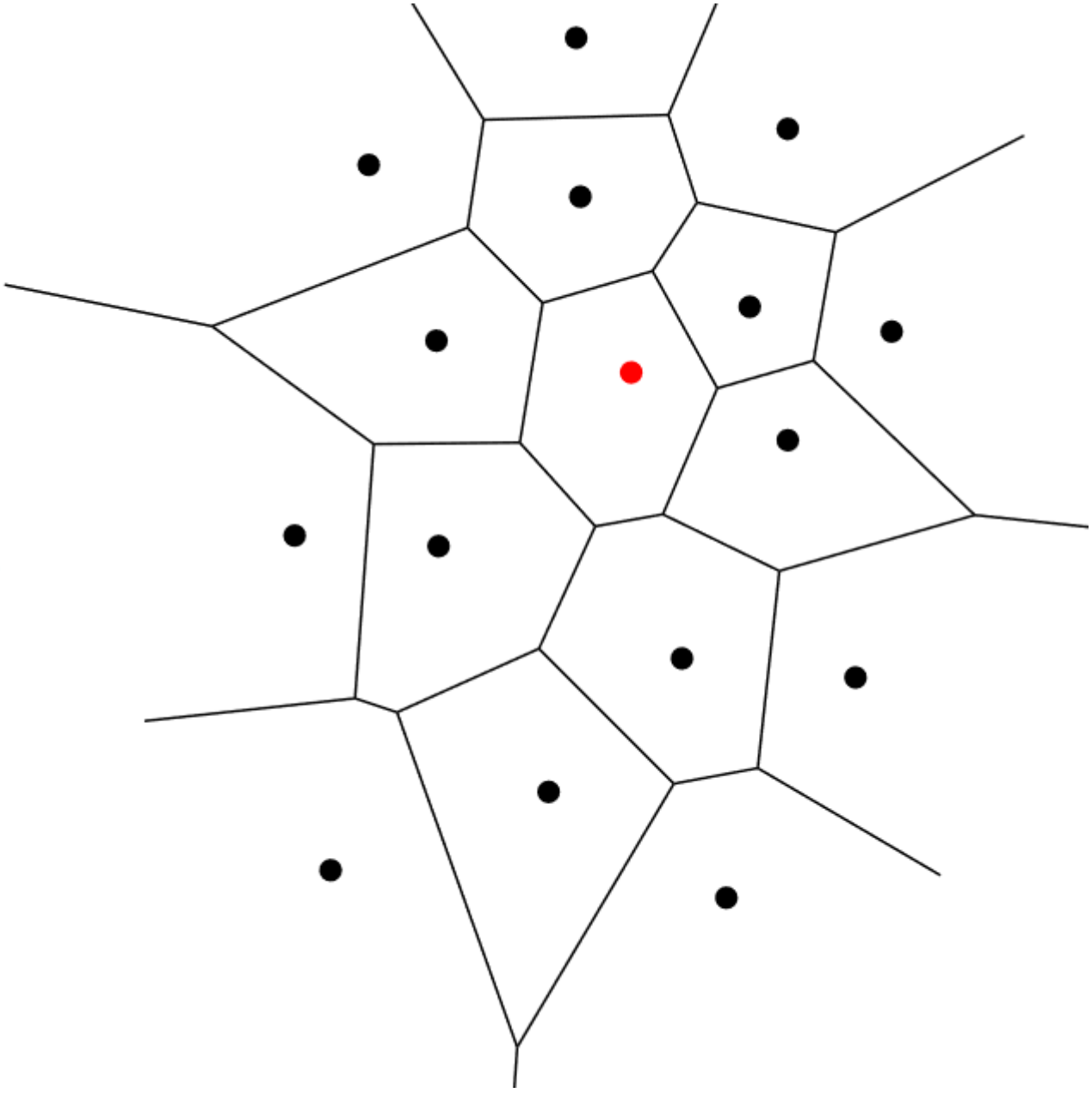
Explore all candidates to find the site $p_j$ $(1 \leq j \leq i)$ closest to $p_{i+1}$.

... compute its region

Build its boundary starting from bisector $b_{i+1,j}$.

... and prune the initial diagram.

While building the Voronoi region of $p_{i+1}$, update the DCEL.



# Voronoi Diagrams

## INCREMENTAL ALGORITHM

Starting with the Voronoi diagram of $\{p_1, \ldots, p_i\}$...

... add point $p_{i+1}$

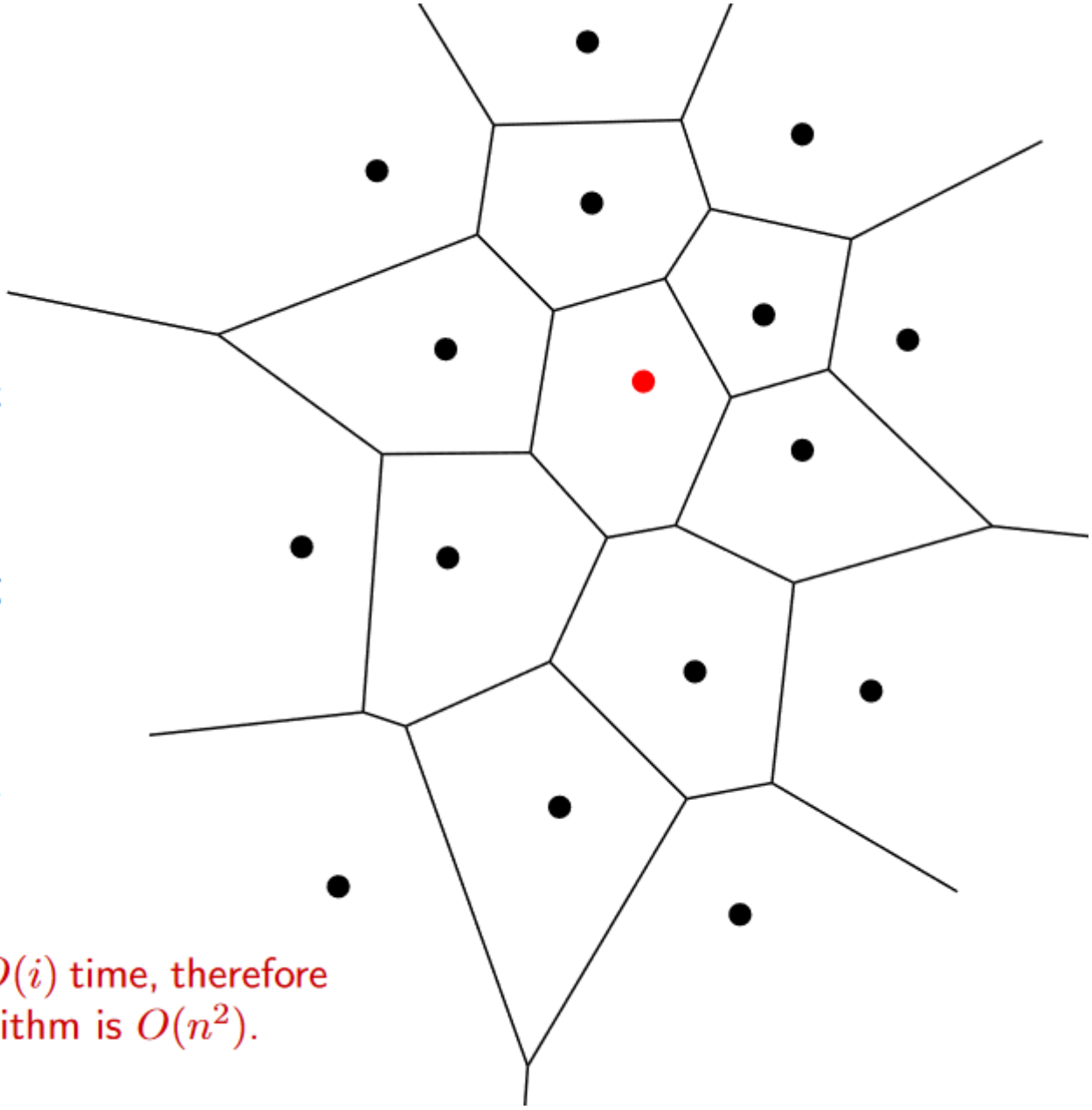
Explore all candidates to find the site $p_j$ $(1 \leq j \leq i)$ closest to $p_{i+1}$.

... compute its region

Build its boundary starting from bisector $b_{i+1,j}$.

... and prune the initial diagram.

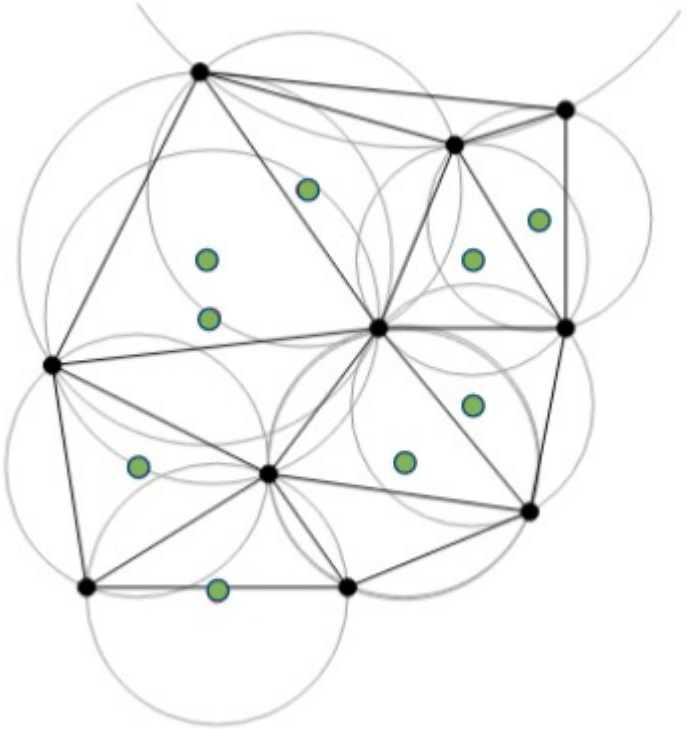
While building the Voronoi region of $p_{i+1}$, update the DCEL.



**Running time:** Each step runs in $O(i)$ time, therefore the total running time of the algorithm is $O(n^2)$.

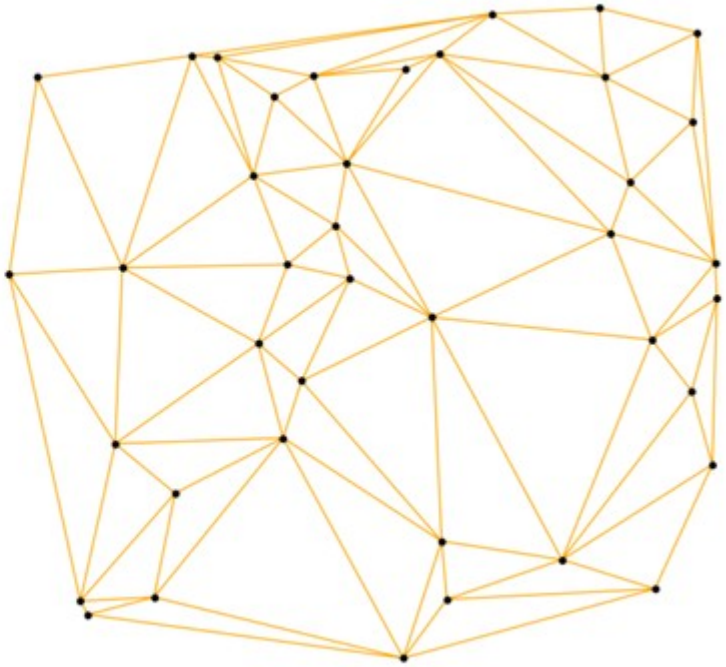
# Voronoi Diagrams

- We can also compute the Voronoi diagram by computing the **Delaunay triangulation** of our set of points.
- A Delaunay triangulation is a collection of triangles built using our original set of points as vertices.
- There is one condition though, “No triangle’s vertex should lie inside the circumcircle of other triangles in the formation”



- We have 10 points in black and it’s easy to see that no other point lies within the drawn circumcircles.
- We also denote the origin of each circumcircle with a green dot.

# Voronoi Diagrams



- By closely comparing the two graphs visually, and imaging a circle around each triangle on the right, we can see a connection with the resulting Voronoi diagram on the left. Furthermore, every triangle circumcircle that we draw over actually corresponds to a vertex in the Voronoi graph.

- That is why if we have a ready Delaunay triangulation, we just need to find the edges of our Voronoi graph by forming an edge from each neighboring triangle's circumcenter to its own circumcenter.

# Voronoi Diagrams

- However, the method of choice is **Fortune's sweepline algorithm,** especially since robust implementations of it are readily available.
- Advantages of Fortune's algorithm include that

(1) it runs in optimal Θ(n log n) time,

(2) it is reasonable to implement, and

(3) we need not store the entire diagram if we can use it as we sweep over it.

# Review Questions

1) Show with examples that if p1 x p2 is positive, then vector $p_1$ is clockwise from vector $p_2$ with respect to the origin (0, 0) and that if this cross product is negative, then $p_1$ is counterclockwise from $p_2$.
2) Give an O(1) algorithm to determine whether given two line segments intersect.
3) Define Convex Hull. Discuss Graham's scan algorithm to compute the convex hull.
4) What are Voronoi diagrams? Explain randomized incremental construction method to construct Voronoi diagrams and analyze it.