Analysis of Algorithms

Computational Geometry

Andres Mendez-Vazquez

November 30, 2015

Outline

Cinvestav

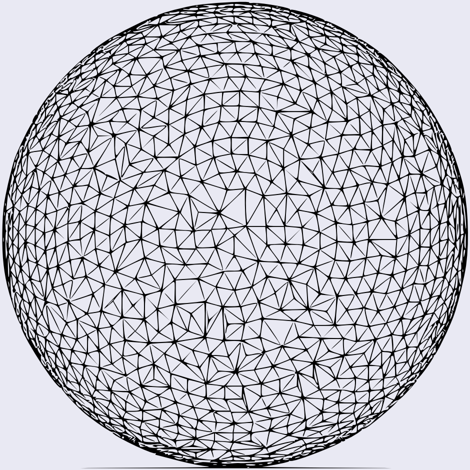
Outline

Cinvestav

Computational Geometry

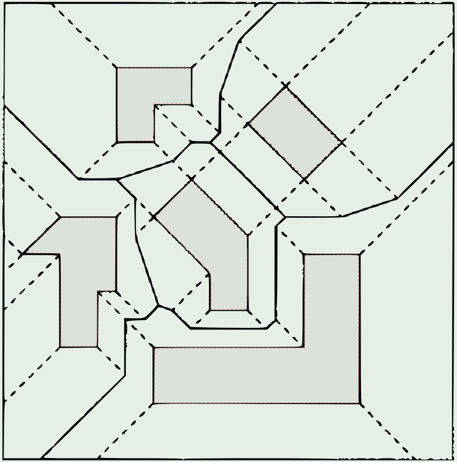
Motivation

- We want to solve geometric problems!!!



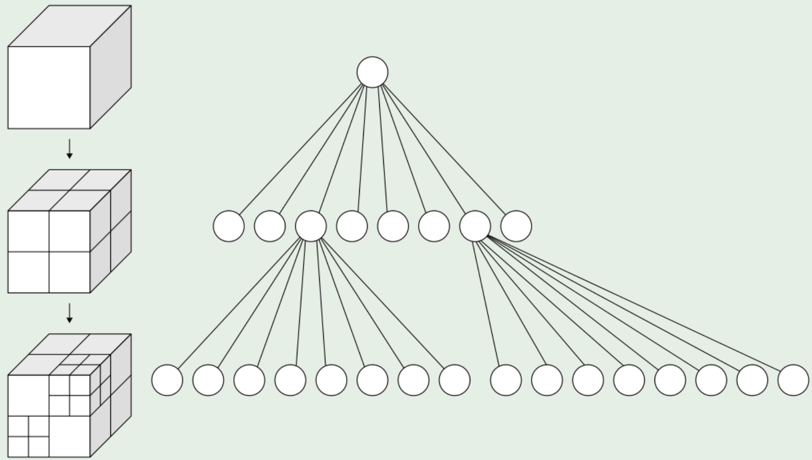
Field of Application

VLSI design - Generation for Fast Voronoi Diagrams for Massive Layouts Under Strict Distances to avoid Tunneling Effects!!



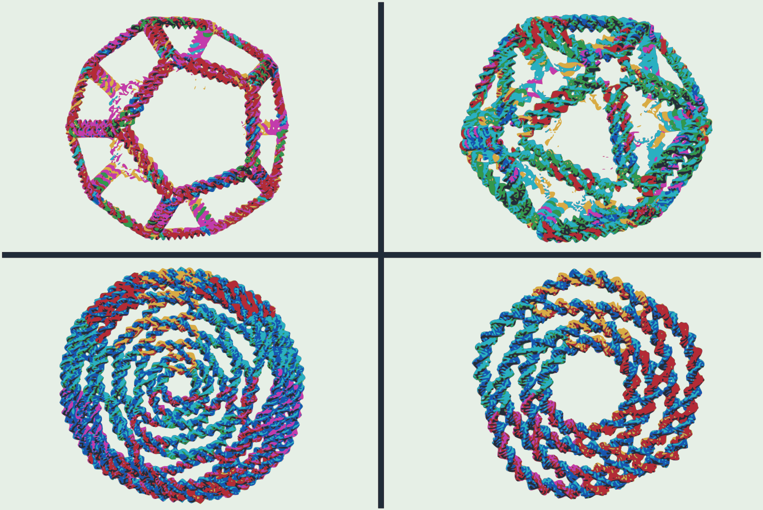
Field of Application

Databases - Octrees for fast localization of information in database tables



Field of Application

Synthetic Biology - Geometric Algorithms to Obtain new DNA configurations for Molecular Machines



Field of Application

Computer Graphics for more engaging Virtual Environments - For example: Bump Mapping!!!

Outline

Cinvestav

The Plane Representation

Although 3D algorithms exist...

- We will deal only with algorithms working in the plane.

  - $p_i = (x_i, y_i)$ and $x_i, y_i \in \mathbb{R}$

The Plane Representation

Although 3D algorithms exist...

- We will deal only with algorithms working in the plane.

Object Representation

- Each object is a set of points $\{p_1, p_2, ..., p_n\}$ where
  - $p_i = (x_i, y_i)$ and $x_i, y_i \in \mathbb{R}$.

The Plane Representation

Although 3D algorithms exist...

- We will deal only with algorithms working in the plane.

Object Representation

- Each object is a set of points $\{p_1, p_2, ..., p_n\}$ where
  - $p_i = (x_i, y_i)$ and $x_i, y_i \in \mathbb{R}$.

Example

- For example an n-vertex polygon P is the following order sequence:
  - $\langle p_0, p_2, ..., p_n \rangle$

The Plane Representation

Although 3D algorithms exist...

- We will deal only with algorithms working in the plane.

Object Representation

- Each object is a set of points $\{p_1, p_2, ..., p_n\}$ where
  - $p_i = (x_i, y_i)$ and $x_i, y_i \in \mathbb{R}$.

Example

- For example an n-vertex polygon P is the following order sequence:
  - $\langle p_0, p_2, ..., p_n \rangle$

The Plane Representation

Although 3D algorithms exist...

- We will deal only with algorithms working in the plane.

Object Representation

- Each object is a set of points $\{p_1, p_2, ..., p_n\}$ where
  - $p_i = (x_i, y_i)$ and $x_i, y_i \in \mathbb{R}$.

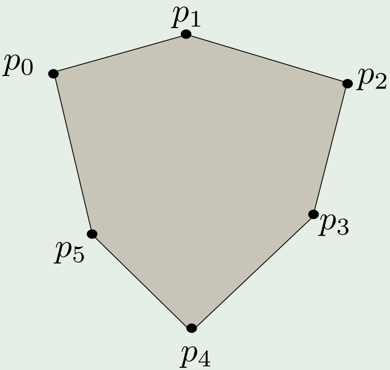
Example

- For example an n-vertex polygon P is the following order sequence:
  - $\langle p_0, p_2, ..., p_n \rangle$

Cinvestav

Example

Polygon



Outline

Cinvestav

Line-segment Properties

A convex combination

- Given two distinct points $p_1 = (x_1, y_1)^T$ and $p_2 = (x_2, y_2)^T$, a convex combination of $\{p_1, p_2\}$ is any point p_3 such that:
  - $p_3 = \alpha p_1 + (1 - \alpha) p_2$ with $0 \leq \alpha \leq 1$.

- If $p_1 = (0, 0)$ then $\overrightarrow{p_1 p_2}$ is the vector p_2.

Line-segment Properties

A convex combination

- Given two distinct points $p_1 = (x_1, y_1)^T$ and $p_2 = (x_2, y_2)^T$, a convex combination of $\{p_1, p_2\}$ is any point p_3 such that:
  - $p_3 = \alpha p_1 + (1 - \alpha)\, p_2$ with $0 \leq \alpha \leq 1$.

Line Segment as Convex Combination

- Given two points p_1 and p_2 (Known as End Points), the line segment $\overline{p_1 p_2}$ is the set of convex combinations of p_1 and p_2.

Line-segment Properties

A convex combination

- Given two distinct points $p_1 = (x_1, y_1)^T$ and $p_2 = (x_2, y_2)^T$, a convex combination of $\{p_1, p_2\}$ is any point p_3 such that:
  - $p_3 = \alpha p_1 + (1 - \alpha) p_2$ with $0 \leq \alpha \leq 1$.

Line Segment as Convex Combination

- Given two points p_1 and p_2 (Known as End Points), the line segment $\overline{p_1 p_2}$ is the set of convex combinations of p_1 and p_2.

Directed Segment

- Here, we care about the direction with initial point p_1 for the directed segment $\overrightarrow{p_1 p_2}$:
  - If $p_1 = (0, 0)$ then $\overrightarrow{p_1 p_2}$ is the vector p_2.

Line-segment Properties

A convex combination

- Given two distinct points $p_1 = (x_1, y_1)^T$ and $p_2 = (x_2, y_2)^T$, a convex combination of $\{p_1, p_2\}$ is any point p_3 such that:
  - $p_3 = \alpha p_1 + (1 - \alpha) p_2$ with $0 \leq \alpha \leq 1$.

Line Segment as Convex Combination

- Given two points p_1 and p_2 (Known as End Points), the line segment $\overline{p_1 p_2}$ is the set of convex combinations of p_1 and p_2.

Directed Segment

- Here, we care about the direction with initial point p_1 for the directed segment $\overrightarrow{p_1 p_2}$:
  - If $p_1 = (0, 0)$ then $\overrightarrow{p_1 p_2}$ is the vector p_2.

Line-segment Properties

A convex combination

- Given two distinct points $p_1 = (x_1, y_1)^T$ and $p_2 = (x_2, y_2)^T$, a convex combination of $\{p_1, p_2\}$ is any point p_3 such that:
  - $p_3 = \alpha p_1 + (1 - \alpha) p_2$ with $0 \leq \alpha \leq 1$.

Line Segment as Convex Combination

- Given two points p_1 and p_2 (Known as End Points), the line segment $\overline{p_1 p_2}$ is the set of convex combinations of p_1 and p_2.

Directed Segment

- Here, we care about the direction with initial point p_1 for the directed segment $\overrightarrow{p_1 p_2}$:
  - If $p_1 = (0, 0)$ then $\overrightarrow{p_1 p_2}$ is the vector p_2.

Outline

Cinvestav

Cross Product

Question!!!

- 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?

Cross Product

Question!!!

- 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?

Cross Product

- Cross product $p_1 \times p_2$ as the signed area of the parallelogram formed by

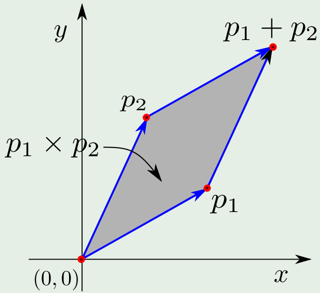
Cross Product

Question!!!

- Given two directed segments $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_0p_2}$,
  - Is $\overrightarrow{p_0p_1}$ clockwise from $\overrightarrow{p_0p_2}$ with respect to their common endpoint p_0?

Cross Product

- Cross product $p_1 \times p_2$ as the signed area of the parallelogram formed by



Cross Product

A shorter representation

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

- If $p_1 \times p_2$ is negative, then p_1 is counterclockwise from p_2.

Cross Product

A shorter representation

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

Thus

- if $p_1 \times p_2$ is positive, then p_1 is clockwise from p_2.
- if $p_1 \times p_2$ is negative, then p_1 is counterclockwise from p_2.

Cross Product

A shorter representation

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

Thus

- if $p_1 \times p_2$ is positive, then p_1 is clockwise from p_2.
- if $p_1 \times p_2$ is negative, then p_1 is counterclockwise from p_2.

Cinvestav

Regions

Clockwise and Counterclockwise Regions

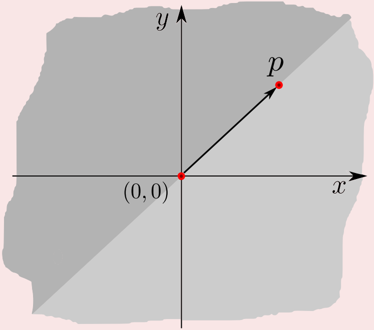


Figure: Darker counterclockwise; lighter clockwise with respect to p

Outline

Cinvestav

Turn Left or Right

Question

Given two line segments $\overrightarrow{p_0 p_1}$ and $\overrightarrow{p_1 p_2}$,

- if we traverse $\overrightarrow{p_0 p_1}$ and then $\overrightarrow{p_1 p_2}$, do we make a left turn at point p_1?

Turn Left or Right

Simply use the following idea

- Compute cross product $(p_2 - p_0) \times (p_1 - p_0)$!!!
- **This translates p_0 to the origin!!!**
- What about $(p_2 - p_0) \times (p_1 - p_0) = 0$?

Cinvestav

Turn Left or Right

Simply use the following idea

- Compute cross product $(p_2 - p_0) \times (p_1 - p_0)$!!!
- **This translates p_0 to the origin**!!!
- What about $(p_2 - p_0) \times (p_1 - p_0) = 0$?

Left Turn = counterclockwise; Right Turn = clockwise

Turn Left or Right

Simply use the following idea

- Compute cross product $(p_2 - p_0) \times (p_1 - p_0)$!!!
- **This translates p_0 to the origin**!!!
- What about $(p_2 - p_0) \times (p_1 - p_0) = 0$?

Left Turn = counterclockwise; Right Turn = clockwise

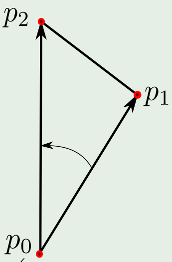
Turn Left or Right

Simply use the following idea

- Compute cross product $(p_2 - p_0) \times (p_1 - p_0)$!!!
- **This translates p_0 to the origin**!!!
- What about $(p_2 - p_0) \times (p_1 - p_0) = 0$?

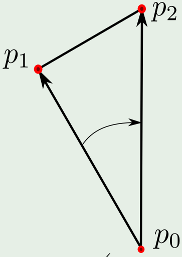
Left Turn = counterclockwise; Right Turn = clockwise

counterclockwise



$$(p_2 - p_0) \times (p_1 - p_0) = \begin{pmatrix} x_2 - x_0 & x_1 - x_0 \\ y_2 - y_0 & y_1 - y_0 \end{pmatrix} < 0$$

clockwise



$$(p_2 - p_0) \times (p_1 - p_0) = \begin{pmatrix} x_2 - x_0 & x_1 - x_0 \\ y_2 - y_0 & y_1 - y_0 \end{pmatrix} > 0$$

Code for this

We have the following code

Direction(p_i, p_j, p_k)

1. return $(p_k - p_i) \times (p_j - p_i)$

Outline

Intersection

Question

Do line segments $\overrightarrow{p_1 p_2}$ and $\overrightarrow{p_3 p_4}$ intersect?

- An endpoint of one segment lies on the other segment

Intersection

Question

Do line segments $\overrightarrow{p_1 p_2}$ and $\overrightarrow{p_3 p_4}$ intersect?

Very Simple!!! We have two possibilities

1. Each segment straddles the line containing the other.
2. An endpoint of one segment lies on the other segment.

Cinvestav

Intersection

Question

Do line segments $\overrightarrow{p_1 p_2}$ and $\overrightarrow{p_3 p_4}$ intersect?

Very Simple!!! We have two possibilities

1. Each segment straddles the line containing the other.
2. An endpoint of one segment lies on the other segment.

Case I This summarize the previous two possibilities

The segments straddle each other's lines.

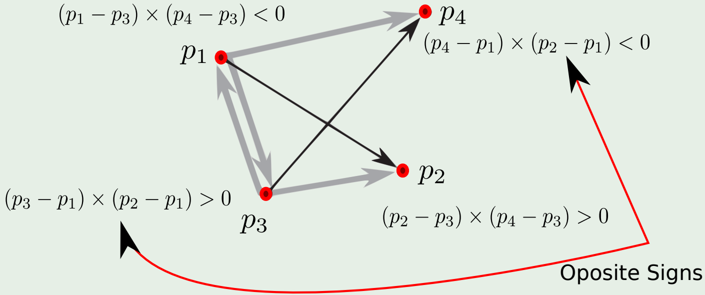


Figure: Using Cross Products to find intersections

Case II No intersection

The segment straddles the line, but the other does not straddle the other line

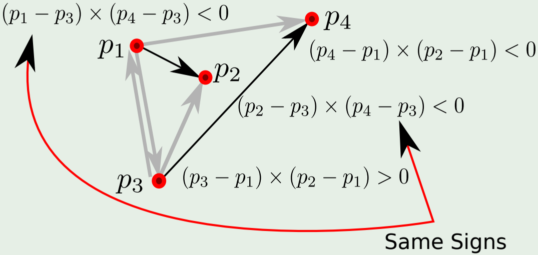


Figure: Using Cross Products to find that there is no intersection

Code

Code

Segment-Intersection(p_1, p_2, p_3, p_4)

1. $d_1 = Direction\,(p_3, p_4, p_1)$
2. $d_2 = Direction\,(p_3, p_4, p_2)$
3. $d_3 = Direction\,(p_1, p_2, p_3)$
4. $d_4 = Direction\,(p_1, p_2, p_4)$
5. **if** $((d_1 > 0$ **and** $d_2 < 0)\ or\ (d_1 < 0$ **and** $d_2 > 0)\ and$
6. $\qquad (d_3 > 0$ **and** $d_4 < 0)\ or\ (d_3 < 0$ **and** $d_4 > 0))$
7. $\qquad\qquad$ **return** $TRUE$

Figure: The Incomplete Code, You still need to test for endpoints over the segment

Code

Code

Segment-Intersection(p_1, p_2, p_3, p_4)

1. $d_1 = Direction\,(p_3, p_4, p_1)$
2. $d_2 = Direction\,(p_3, p_4, p_2)$
3. $d_3 = Direction\,(p_1, p_2, p_3)$
4. $d_4 = Direction\,(p_1, p_2, p_4)$
5. **if** $((d_1 > 0$ **and** $d_2 < 0)\ or\ (d_1 < 0$ **and** $d_2 > 0)\ and$
6. $\quad (d_3 > 0$ **and** $d_4 < 0)\ or\ (d_3 < 0$ **and** $d_4 > 0))$
7. $\qquad$ **return** $TRUE$

Figure: The Incomplete Code, You still need to test for endpoints over the segment

Outline

Cinvestav

Sweeping

Sweeping

Use an imaginary vertical line to pass through the n segments with events $x \in \{r, t, u\}$:

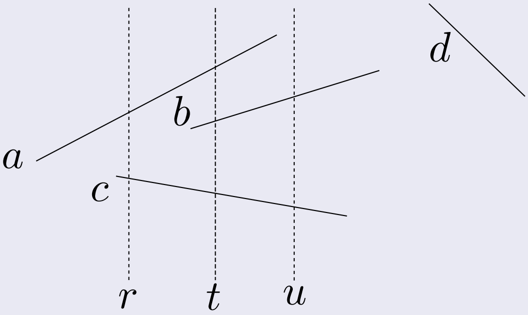


Figure: Vertical Line to Record Events

Thus

This can be used to record events given two segments s_1 and s_2

- Event I: s_1 above s_2 at x, written $s_1 \succsim_x s_2$.
  - This is a total preorder relation for segment intersecting the line at x.
  - The relation is transitive and reflexive.
- Event II: s_1 intersect s_2, then neither $s_1 \succsim_x s_2$ or $s_2 \succsim_x s_1$, or both (**if** s_1 **and** s_2 **intersect at** x)

Thus

This can be used to record events given two segments s_1 and s_2

- Event I: s_1 above s_2 at x, written $s_1 \succsim_x s_2$.
  - This is a total preorder relation for segment intersecting the line at x.
  - The relation is transitive and reflexive.
- Event II: s_1 intersect s_2, then neither $s_1 \succsim_x s_2$ or $s_2 \succsim_x s_1$, or both (**if s_1 and s_2 intersect at x**)

Thus

This can be used to record events given two segments s_1 and s_2

- Event I: s_1 above s_2 at x, written $s_1 \succsim_x s_2$.
  - This is a total preorder relation for segment intersecting the line at x.
  - The relation is transitive and reflexive.
- Event II: s_1 intersect s_2, then neither $s_1 \succsim_x s_2$ or $s_2 \succsim_x s_1$, or both (**if** s_1 **and** s_2 **intersect at** x)

Thus

This can be used to record events given two segments s_1 and s_2

- Event I: s_1 above s_2 at x, written $s_1 \succsim_x s_2$.
  - This is a total preorder relation for segment intersecting the line at x.
  - The relation is transitive and reflexive.
- Event II: s_1 intersect s_2, then neither $s_1 \succsim_x s_2$ or $s_2 \succsim_x s_1$, or both (**if** s_1 **and** s_2 **intersect at** x)

Cinvestav

Example

Example: $a \succcurlyeq_r c \;\; a \succcurlyeq_t c$

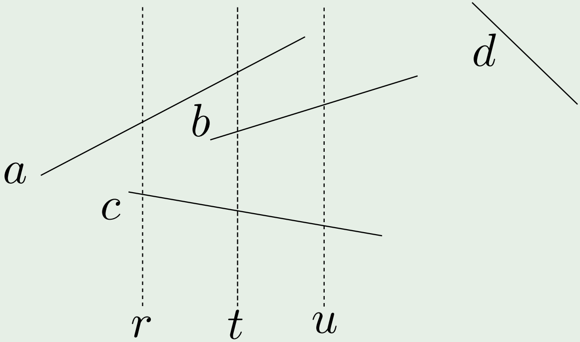


Figure: Vertical Line to Record Events

Change in direction

When e and f intersect, $e \succsim_v f$ and $f \succsim_w e$. In the Shaded Region, any sweep line will have e and f as consecutive

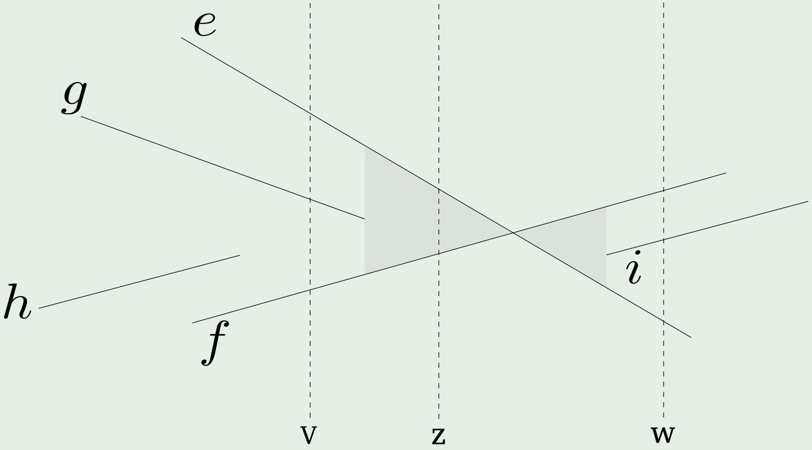


Figure: Vertical Line to Record Events

Moving the sweep line

Something Notable

Sweeping algorithms typically manage two sets of data.

Moving the sweep line

Something Notable

Sweeping algorithms typically manage two sets of data.

Sweep-line status

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

Event-point schedule

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 progress from left to right, it stops and processes each event point, then resumes.
- It is possible to use a min-priority queue to keep those event points sorted by x-coordinate.

Moving the sweep line

Something Notable

Sweeping algorithms typically manage two sets of data.

Sweep-line status

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

Event-point schedule

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 progress from left to right, it stops and processes each event point, then resumes.
- It is possible to use a min-priority queue to keep those event points sorted by x-coordinate.

Moving the sweep line

Something Notable

Sweeping algorithms typically manage two sets of data.

Sweep-line status

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

Event-point schedule

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 progress from left to right, it stops and processes each event point, then resumes.
- It is possible to use a min-priority queue to keep those event points sorted by x-coordinate.

Sweeping Process

First

- We sort the segment endpoints by increasing x-coordinate and proceed from left to right.

However, sometimes they have the same x-coordinate (Covertical)

If two or more endpoints are covertical, we break the tie by putting all the covertical left endpoints before the covertical right endpoints.

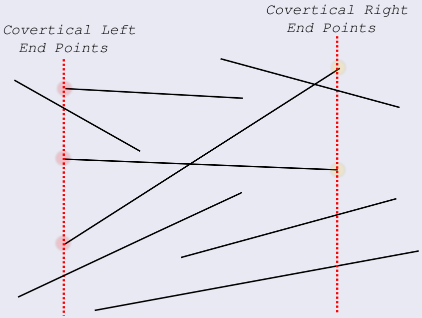
Sweeping Process

First

- We sort the segment endpoints by increasing x-coordinate and proceed from left to right.

However, sometimes they have the same x-coordinate (Covertical)

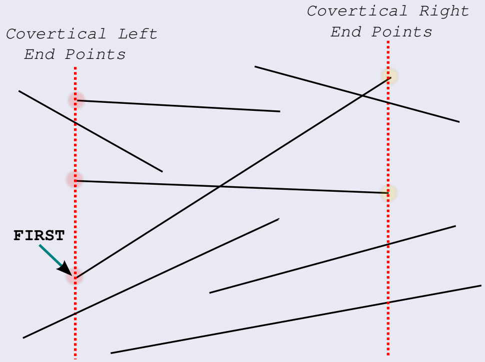
If two or more endpoints are covertical, we break the tie by putting all the covertical left endpoints before the covertical right endpoints.



Then

Second

Within a set of covertical left endpoints, we put those with lower y-coordinates first, and we do the same within a set of covertical right endpoints.



Then

Process

1. When we encounter a segment's left endpoint, we insert the segment into the sweep-line status.
2. We delete the segment from the sweep-line status upon encountering its right endpoint.

Then

Process

1. When we encounter a segment's left endpoint, we insert the segment into the sweep-line status.
2. We delete the segment from the sweep-line status upon encountering its right endpoint.

Thus

Whenever two segments first become consecutive in the total preorder, we check whether they intersect.

Then

Process

1. When we encounter a segment's left endpoint, we insert the segment into the sweep-line status.
2. We delete the segment from the sweep-line status upon encountering its right endpoint.

Thus

Whenever two segments first become consecutive in the total preorder, we check whether they intersect.

Operations

Operations to keep preorder on the events for algorithm

- INSERT(T, s): insert segment s into T.
- DELETE(T, s): delete segment s from T.
- ABOVE(T, s): return the segment immediately above segment s in T.
- BELOW(T, s): return the segment immediately below segment s in T.

Operations

Operations to keep preorder on the events for algorithm

- INSERT(T, s): insert segment s into T.
- DELETE(T, s): delete segment s from T.
- ABOVE(T, s): return the segment immediately above segment s in T.
- BELOW(T, s): return the segment immediately below segment s in T.

Note

Each operation can be performed in $O(\log_2 n)$ using a red-black-tree by using comparisons by cross product to find the above and below.

Operations

Operations to keep preorder on the events for algorithm

- INSERT(T, s): insert segment s into T.
- DELETE(T, s): delete segment s from T.
- ABOVE(T, s): return the segment immediately above segment s in T.
- BELOW(T, s): return the segment immediately below segment s in T.

Note

Each operation can be performed in $O(\log_2 n)$ using a red-black-tree by using comparisons by cross product to find the above and below.

This allows to see

The relative ordering of two segments.

Cinvestav

Operations

Operations to keep preorder on the events for algorithm

- INSERT(T, s): insert segment s into T.
- DELETE(T, s): delete segment s from T.
- ABOVE(T, s): return the segment immediately above segment s in T.
- BELOW(T, s): return the segment immediately below segment s in T.

Note

Each operation can be performed in $O(\log_2 n)$ using a red-black-tree by using comparisons by cross product to find the above and below.

This allows to see

The relative ordering of two segments.

Cinvestav

Operations

Operations to keep preorder on the events for algorithm

- INSERT(T, s): insert segment s into T.
- DELETE(T, s): delete segment s from T.
- ABOVE(T, s): return the segment immediately above segment s in T.
- BELOW(T, s): return the segment immediately below segment s in T.

Note

Each operation can be performed in $O(\log_2 n)$ using a red-black-tree by using comparisons by cross product to find the above and below.

This allows to see

The relative ordering of two segments.

Operations

Operations to keep preorder on the events for algorithm

- INSERT(T, s): insert segment s into T.
- DELETE(T, s): delete segment s from T.
- ABOVE(T, s): return the segment immediately above segment s in T.
- BELOW(T, s): return the segment immediately below segment s in T.

Note

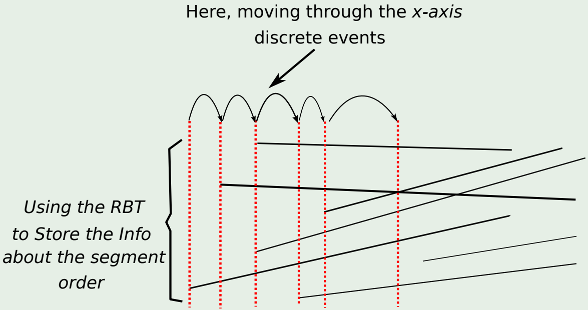
Each operation can be performed in $O(\log_2 n)$ using a red-black-tree by using comparisons by cross product to find the above and below.

This allows to see

The relative ordering of two segments.

What the algorithm does?

Moving the sweeping line discretely - Event-point schedule



Event-Point Schedule Implementation

For this

We can use a Priority Queue using lexicographic order

The interesting part is the Sweeping-Line Satus

Because the way we build the balanced tree

Event-Point Schedule Implementation

For this

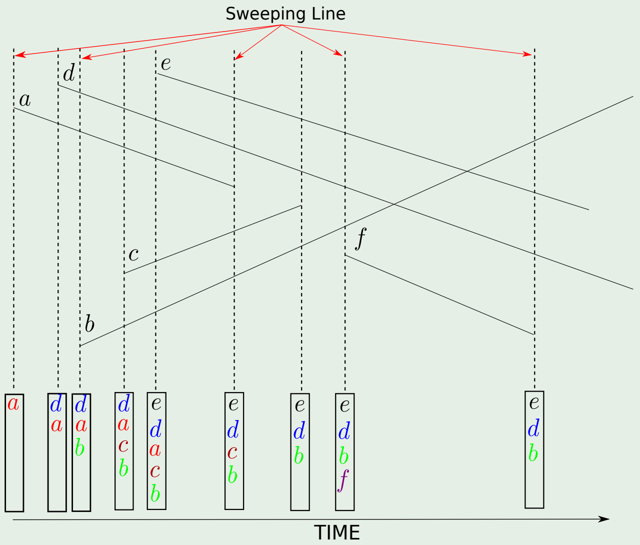
We can use a Priority Queue using lexicographic order

The interesting part is the Sweeping-Line Satus

Because the way we build the balanced tree

Sweep-Line Status

The Above and Below relation



Sweeping Line Status Implementation

Use the following relation of order to build the binary tree

Given a segment x, then you insert y

- Case I if y is counterclockwise, it is below x (Go to the left).
- Case II if y is clockwise, it is above x (Go to the Right)

In addtion

If you are at a leaf do the insertion, but also insert the leaf at the left or right given the insertion.

Sweeping Line Status Implementation

Use the following relation of order to build the binary tree

Given a segment x, then you insert y

Case I if y is counterclockwise, it is below x (Go to the left).

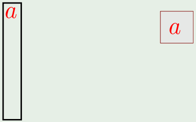
Case II if y is clockwise, it is above x (Go to the Right)

In addtion

If you are at a leaf do the insertion, but also insert the leaf at the left or right given the insertion.

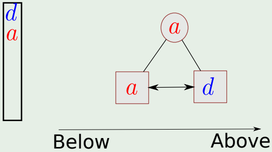
Example

We insert th first element in the circular leaves list



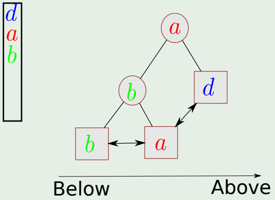
Example

We insert a inner node after binary search



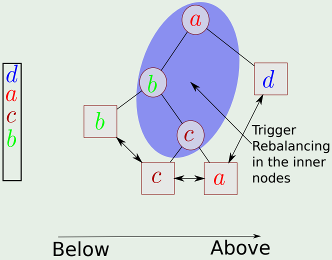
Example

Similar



Example

Etc....


d
a
c
b
a
b
d
b
c
c
a
Trigger
Rebalancing
in the inner
nodes
Below
Above

Pseudo-code with complexity $O\left(n \log_2 n\right)$

Any-Segment-Intersect(S)

1. $T = \emptyset$
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
4. **if** p is the left endpoint of a segment s
5. INSERT(T, s)
6. **if** (ABOVE(T, s) exists and intersect s)
   or (BELOW(T, s) exists and intersect 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) intersect BELOW(T, s))
10. **return** $TRUE$
11. DELETE(T, s)
12. **return** $FALSE$

Pseudo-code with complexity $O(n \log_2 n)$

Any-Segment-Intersect(S)

1. $T = \emptyset$
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
4. **if** p is the left endpoint of a segment s
5. INSERT(T, s)
6. **if** (ABOVE(T, s) exists and intersect s) or (BELOW(T, s) exists and intersect 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) intersect BELOW(T, s))
10. **return** $TRUE$
11. DELETE(T, s)
12. **return** $FALSE$

Pseudo-code with complexity $O(n \log_2 n)$

Any-Segment-Intersect(S)

1. $T = \emptyset$
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
4. if p is the left endpoint of a segment s
5. INSERT(T, s)
6. if (ABOVE(T, s) exists and intersect s)
   or (BELOW(T, s) exists and intersect 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) intersect BELOW(T, s))
10. **return** $TRUE$
11. DELETE(T, s)
12. **return** $FALSE$

Pseudo-code with complexity $O\left(n\log_2 n\right)$

Any-Segment-Intersect(S)

1. $T=\emptyset$
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
4. if p is the left endpoint of a segment s
5. INSERT(T,s)
6. **if** (ABOVE(T,s) exists and intersect s)
   or (BELOW(T,s) exists and intersect 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) intersect BELOW(T,s))
10. **return** $TRUE$
11. DELETE(T,s)
12. **return** $FALSE$

Pseudo-code with complexity $O(n \log_2 n)$

Any-Segment-Intersect(S)

1. $T = \emptyset$
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
4. if p is the left endpoint of a segment s
5. INSERT(T, s)
6. if (ABOVE(T, s) exists and intersect s)
   or (BELOW(T, s) exists and intersect 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) intersect BELOW(T, s))
10. **return** $TRUE$
11. DELETE(T, s)
12. **return** $FALSE$

Pseudo-code with complexity $O(n \log_2 n)$

Any-Segment-Intersect(S)

1. $T = \emptyset$
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
4. **if** p is the left endpoint of a segment s
5. INSERT(T, s)
6. **if** (ABOVE(T, s) exists and intersect s)
   or (BELOW(T, s) exists and intersect 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) intersect BELOW(T, s))
10. **return** $TRUE$
11. DELETE(T, s)
12. **return** $FALSE$

Pseudo-code with complexity $O\left(n \log_2 n\right)$

Any-Segment-Intersect(S)

1. $T = \emptyset$
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
4.   **if** p is the left endpoint of a segment s
5.   INSERT(T, s)
6.   **if** (ABOVE(T, s) exists and intersect s)
               or (BELOW(T, s) exists and intersect 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) intersect BELOW(T, s))
10.   **return** $TRUE$
11.   DELETE(T, s)
12. **return** $FALSE$

Pseudo-code with complexity $O(n \log_2 n)$

Any-Segment-Intersect(S)

$T = \emptyset$

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

for each point p in the sorted list

    if p is the left endpoint of a segment s

        INSERT(T, s)

        if (ABOVE(T, s) exists and intersect s)
            or (BELOW(T, s) exists and intersect s)

            return $TRUE$

    if p is the right endpoint of a segment s

        if (both ABOVE(T, s) and BELOW(T, s) exist)
            and (ABOVE(T, s) intersect BELOW(T, s))

            return $TRUE$

        DELETE(T, s)

return $FALSE$

Pseudo-code with complexity $O(n \log_2 n)$

Any-Segment-Intersect(S)

$T = \emptyset$

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

for each point p in the sorted list
    if p is the left endpoint of a segment s
        INSERT(T, s)
        if (ABOVE(T, s) exists and intersect s)
            or (BELOW(T, s) exists and intersect s)
                return $TRUE$
    if p is the right endpoint of a segment s
        if (both ABOVE(T, s) and BELOW(T, s) exist)
            and (ABOVE(T, s) intersect BELOW(T, s))
                return $TRUE$
        DELETE(T, s)
return $FALSE$

Pseudo-code with complexity $O(n \log_2 n)$

Any-Segment-Intersect(S)

```
T = ∅
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
for each point p in the sorted list
        if p is the left endpoint of a segment s
                INSERT(T, s)
                if (ABOVE(T, s) exists and intersect s)
                        or (BELOW(T, s) exists and intersect s)
                                return TRUE
        if p is the right endpoint of a segment s
                if (both ABOVE(T, s) and BELOW(T, s) exist)
                        and (ABOVE(T, s) intersect BELOW(T, s))
                                return TRUE
                DELETE(T, s)
return FALSE
```

Pseudo-code with complexity $O\left(n \log_2 n\right)$

Any-Segment-Intersect(S)

$T = \emptyset$

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

for each point p in the sorted list

  if p is the left endpoint of a segment s

    INSERT(T, s)

    if (ABOVE(T, s) exists and intersect s)
      or (BELOW(T, s) exists and intersect s)

        return $TRUE$

  if p is the right endpoint of a segment s

    if (both ABOVE(T, s) and BELOW(T, s) exist)
      and (ABOVE(T, s) intersect BELOW(T, s))

        return $TRUE$

    DELETE(T, s)

return $FALSE$

Outline

Cinvestav

Correctness

The ANY-SEGMENTS-INTERSECT returns TRUE

If it finds an intersection between two of the input segments.

Correctness

The ANY-SEGMENTS-INTERSECT returns TRUE

If it finds an intersection between two of the input segments.

Proof:

Observation: What if there is the leftmost intersection, p?

- Then, let a and b be the segments to intersect at p

Correctness

The ANY-SEGMENTS-INTERSECT returns TRUE

If it finds an intersection between two of the input segments.

Proof:

Observation: What if there is the leftmost intersection, p?

- Then, let a and b be the segments to intersect at p

Then, for a and b

- Since no intersections occur to the left of p, the order given by T (Sweeping Line Data Structure) is correct at all points to the left of p.
- Assuming that no three segments intersect at the same point, a and b become consecutive in the total preorder of some sweep line z.

Cinvestav

Correctness

The ANY-SEGMENTS-INTERSECT returns TRUE

If it finds an intersection between two of the input segments.

Proof:

Observation: What if there is the leftmost intersection, p?

- Then, let a and b be the segments to intersect at p

Then, for a and b

- Since no intersections occur to the left of p, the order given by T (Sweeping Line Data Structure) is correct at all points to the left of p.
- Assuming that no three segments intersect at the same point, a and b become consecutive in the total preorder of some sweep line z.

Correctness

The ANY-SEGMENTS-INTERSECT returns TRUE

If it finds an intersection between two of the input segments.

Proof:

Observation: What if there is the leftmost intersection, p?

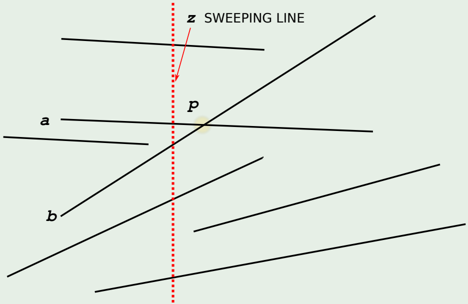
- Then, let a and b be the segments to intersect at p

Then, for a and b

- Since no intersections occur to the left of p, the order given by T (Sweeping Line Data Structure) is correct at all points to the left of p.
- Assuming that no three segments intersect at the same point, a and b become consecutive in the total preorder of some sweep line z.

Now, we have two possbibilities

Case I



Case I

Moreover

z is to the left of p or goes through p.

In addition

There is a endpoint q where a and b become consecutive.

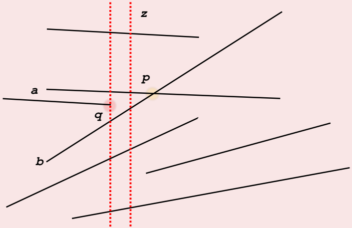
Case I

Moreover

z is to the left of p or goes through p.

In addition

There is a endpoint q where a and b become consecutive.



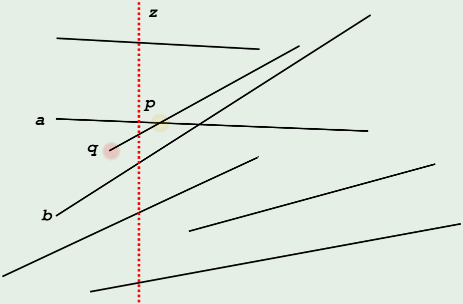
Finally

Then a and b

They become consecutive in the total pre-order of a sweep line.

Case II

We have that q is a left endpoint where a and b stop being consecutive



Correctness about the order given by T

Then, given the two following cases

1. if p is in the sweep line $\Rightarrow p == q$.
2. If q is at the left of p, and it is the nearest left one.

Correctness about the order given by T

Then, given the two following cases

1. if p is in the sweep line $\Rightarrow p == q$.
2. If q is at the left of p, and it is the nearest left one.

Do we mantain the correct preorder?

We have that given that p is first

Then, it is processed first becacuse the lexicographic order.

Therefore, two cases can happen

1. The point is processed - then the algorithm returns true
2. If the event is not processed - then the algorithm must have returned true

Do we mantain the correct preorder?

We have that given that p is first

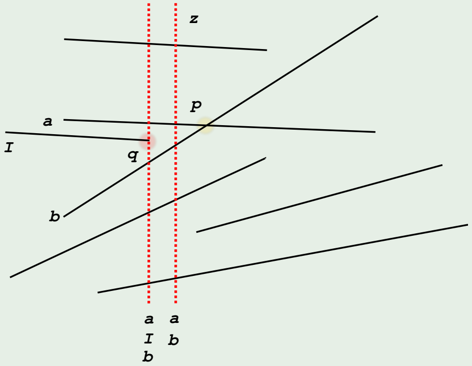
Then, it is processed first becacuse the lexicographic order.

Therefore, two cases can happen

1. The point is processed - then the algorithm returns true
2. If the event is not processed - then the algorithm must have returned true

Handling Case I

Segments a and b are already in T , and a segment between them in the total pre-order is deleted, making a and b to become consecutive



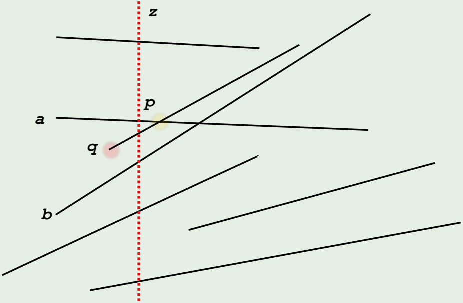
When is this detected?

In the following lines of the code

Lines 8–11 detect this case.

Handling Case II

Either a or b is inserted into T, and the other segment is above or below it in the total pre-order.



When is this detected?

In the following lines of the code

Lines 4–7 detect this case.

Finally

If event point q is not processed

It must have found an earlier intersection!!!

Therefore

If there is an intersection Any-Segment-Intersect returns true all the time

Cinvestav

Finally

If event point q is not processed

It must have found an earlier intersection!!!

Therefore

If there is an intersection Any-Segment-Intersect returns true all the time

Running Time

Something Notable

1. Line 1 takes $O(1)$ time.
2. Line 2 takes $O(n \log_2 n)$ time, using merge or heap sort
3. The for loop iterates at most $2n$ times
   1. Each iteration takes $O(\log_2 n)$ in a well balanced tree.
   2. Each intersection test takes $O(1)$

Running Time

Something Notable

1. Line 1 takes $O(1)$ time.
2. Line 2 takes $O(n \log_2 n)$ time, using merge or heap sort
3. The for loop iterates at most $2n$ times
   1. Each iteration takes $O(\log_2 n)$ in a well balanced tree.
   2. Each intersection test takes $O(1)$

Total Time

$O(n \log_2 n)$

Running Time

Something Notable

1. Line 1 takes $O(1)$ time.
2. Line 2 takes $O(n \log_2 n)$ time, using merge or heap sort
3. The for loop iterates at most $2n$ times
   1. Each iteration takes $O(\log_2 n)$ in a well balanced tree.
   2. Each intersection test takes $O(1)$

Total Time

$O(n \log_2 n)$

Running Time

Something Notable

1. Line 1 takes $O(1)$ time.
2. Line 2 takes $O(n \log_2 n)$ time, using merge or heap sort
3. The for loop iterates at most $2n$ times
   1. Each iteration takes $O(\log_2 n)$ in a well balanced tree.
   2. Each intersection test takes $O(1)$

Total Time

$O(n \log_2 n)$

Running Time

Something Notable

1. Line 1 takes $O(1)$ time.
2. Line 2 takes $O(n \log_2 n)$ time, using merge or heap sort
3. The for loop iterates at most $2n$ times
    1. Each iteration takes $O\left(\log_2 n\right)$ in a well balanced tree.
    2. Each intersection test takes $O\left(1\right)$

Total Time

$O\left(n \log_2 n\right)$

Running Time

Something Notable

1. Line 1 takes $O(1)$ time.
2. Line 2 takes $O(n \log_2 n)$ time, using merge or heap sort
3. The for loop iterates at most $2n$ times
   1. Each iteration takes $O(\log_2 n)$ in a well balanced tree.
   2. Each intersection test takes $O(1)$

Total Time

$O(n \log_2 n)$

Outline

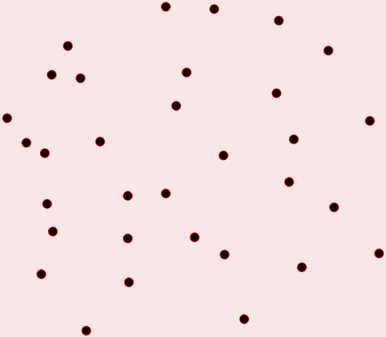
Cinvestav

Convex Hull

Convex Hull

- Given a set of points, Q, find the smallest convex polygon P such that $Q \subset P$. This is denoted by $\text{CH}(Q)$.

Bunch of Points in 2D

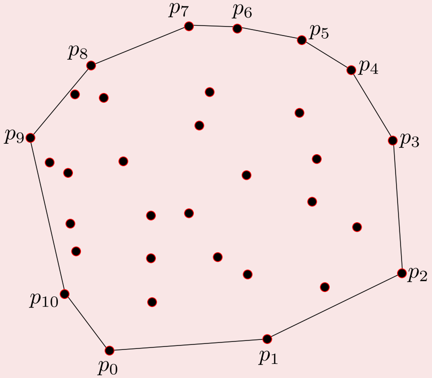


Convex Hull

Convex Hull

- Given a set of points, Q, find the smallest convex polygon P such that $Q \subset P$. This is denoted by $\text{CH}(Q)$.

Their Convex Hull



Convex Hull

The two main Algorithms (Using the "Rotational Sweep") that we are going to explore

- Graham's Scan.

Convex Hull

The two main Algorithms (Using the "Rotational Sweep") that we are going to explore

- Graham's Scan.
- Jarvis' March.

Nevertheless there are other methods

- The incremental method.
- Divide-and-conquer method.
- Prune-and-search method.

Convex Hull

The two main Algorithms (Using the "Rotational Sweep") that we are going to explore

- Graham's Scan.
- Jarvis' March.

Nevertheless there are other methods

- The incremental method
- Divide-and-conquer method.
- Prune-and-search method.

Convex Hull

The two main Algorithms (Using the "Rotational Sweep") that we are going to explore

- Graham's Scan.
- Jarvis' March.

Nevertheless there are other methods

- The incremental method
- Divide-and-conquer method.
- Prune-and-search method.

Convex Hull

The two main Algorithms (Using the "Rotational Sweep") that we are going to explore

- Graham's Scan.
- Jarvis' March.

Nevertheless there are other methods

- The incremental method
- Divide-and-conquer method.
- Prune-and-search method.

Outline

Graham's Scan

Graham's Scan Basics

- **It keeps a Stack of candidate points.**
- It Pops elements that are not part of the CH(Q).
- Whatever is left in the Stack is part of the CH(Q).

Graham's Scan Basics

- **It keeps a Stack of candidate points.**
- **It Pops elements that are not part of the CH(Q).**
- **Whatever is left in the Stack is part of the CH(Q).**

Graham's Scan

Graham's Scan Basics

- **It keeps a Stack of candidate points.**
- **It Pops elements that are not part of the CH**(Q)**.**
- **Whatever is left in the Stack is part of the CH**(Q)**.**

Graham's Scan Code

Algorithm

GRAHAM-SCAN(Q)

1. Let p_0 be the point Q with the minimum y-coordinate or the leftmost such point in case of a **tie**
2. let $\langle p_1, p_2, ..., p_n \rangle$ be the remaining points in Q, sorted by polar angle in counter clockwise order around p_0 (If more than one point has the same angle, remove all but 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** n
8.   **while** ccw(next-top(S), p_i, top (S)) ≤ 0
9.   POP(S)
10.   PUSH(S)
11. **return** S

▷ The clockwise and counter clockwise algorithm◁

ccw(p_1, p_2, p_3)

1. return $(p_3 - p_1 \times p_2 - p_1)$

Graham's Scan Code

Algorithm

GRAHAM-SCAN(Q)

1. Let p_0 be the point Q with the minimum y-coordinate or the leftmost such point in case of a **tie**
2. let $\langle p_1, p_2, ..., p_n \rangle$ be the remaining points in Q, sorted by polar angle in counter clockwise order around p_0 (If more than one point has the same angle, remove all but 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** n
8. **while** ccw(next-top(S), p_i, top (S)) ≤ 0
9. POP(S)
10. PUSH(S)
11. **return** S

▷ The clockwise and counter clockwise algorithm◁

ccw(p_1, p_2, p_3)

1. return $(p_3 - p_1 \times p_2 - p_1)$

Graham's Scan Code

Algorithm

GRAHAM-SCAN(Q)

1. Let p_0 be the point Q with the minimum y-coordinate or the leftmost such point in case of a **tie**
2. let $\langle p_1, p_2, ..., p_n \rangle$ be the remaining points in Q, sorted by polar angle in counter clockwise order around p_0 (If more than one point has the same angle, remove all but 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** n
8. **while** ccw(next-top(S), p_i, top (S)) ≤ 0
9. POP(S)
10. PUSH(S)
11. **return** S

▷ The clockwise and counter clockwise algorithm◁

ccw(p_1, p_2, p_3)

1. return ($p_3 - p_1 \times p_2 - p_1$)

Graham's Scan Code

Algorithm

GRAHAM-SCAN(Q)

1. Let p_0 be the point Q with the minimum y-coordinate or the leftmost such point in case of a **tie**
2. let $\langle p_1, p_2, ..., p_n \rangle$ be the remaining points in Q, sorted by polar angle in counter clockwise order around p_0 (If more than one point has the same angle, remove all but 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** n
8. **while** ccw(next-top(S), p_i, top (S)) ≤ 0
9. POP(S)
10. PUSH(S)
11. **return** S

▷ The clockwise and counter clockwise algorithm◁

ccw(p_1, p_2, p_3)

1. return $(p_3 - p_1 \times p_2 - p_1)$

Graham's Scan Code

Algorithm

GRAHAM-SCAN(Q)

1. Let p_0 be the point Q with the minimum y-coordinate or the leftmost such point in case of a **tie**
2. let $\langle p_1, p_2, ..., p_n \rangle$ be the remaining points in Q, sorted by polar angle in counter clockwise order around p_0 (If more than one point has the same angle, remove all but 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** n
8. **while** ccw(next-top(S), p_i, top $(S)) \leq 0$
9. POP(S)
10. PUSH(S)
11. **return** S

▷ The clockwise and counter clockwise algorithm◁

ccw(p_1, p_2, p_3)

1. return $(p_3 - p_1 \times p_2 - p_1)$

Graham's Scan Code

Algorithm

GRAHAM-SCAN(Q)

1. Let p_0 be the point Q with the minimum y-coordinate or the leftmost such point in case of a **tie**
2. let $\langle p_1, p_2, ..., p_n \rangle$ be the remaining points in Q, sorted by polar angle in counter clockwise order around p_0 (If more than one point has the same angle, remove all but 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** n
8.   **while** ccw(next-top(S), p_i, top $(S)) \leq 0$
9.   POP(S)
10.   PUSH(S)
11. **return** S

▷ The clockwise and counter clockwise algorithm◁

ccw(p_1, p_2, p_3)

1. return $(p_3 - p_1 \times p_2 - p_1)$

Graham's Scan Code

Algorithm

GRAHAM-SCAN(Q)

1. Let p_0 be the point Q with the minimum y-coordinate or the leftmost such point in case of a **tie**
2. let $\langle p_1, p_2, ..., p_n \rangle$ be the remaining points in Q, sorted by polar angle in counter clockwise order around p_0 (If more than one point has the same angle, remove all but 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** n
8. **while** ccw(next-top(S)$, p_i,$ top $(S)) \leq 0$
9. POP(S)
10. PUSH(S)
11. **return** S

▷ The clockwise and counter clockwise algorithm◁

ccw(p_1, p_2, p_3)

1. return $(p_3 - p_1 \times p_2 - p_1)$

Graham's Scan Code

Algorithm

GRAHAM-SCAN(Q)

1. Let p_0 be the point Q with the minimum y-coordinate or the leftmost such point in case of a **tie**
2. let $\langle p_1, p_2, ..., p_n \rangle$ be the remaining points in Q, sorted by polar angle in counter clockwise order around p_0 (If more than one point has the same angle, remove all but 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** n
8.   **while** ccw(next-top(S), p_i, top (S)) ≤ 0
9.   POP(S)
10.   PUSH(S)
11. **return** S

▷ The clockwise and counter clockwise algorithm◁

ccw(p_1, p_2, p_3)

1. return ($p_3 - p_1 \times p_2 - p_1$)

Graham's Scan Code

Algorithm

GRAHAM-SCAN(Q)

1. Let p_0 be the point Q with the minimum y-coordinate or the leftmost such point in case of a **tie**
2. let $\langle p_1, p_2, ..., p_n \rangle$ be the remaining points in Q, sorted by polar angle in counter clockwise order around p_0 (If more than one point has the same angle, remove all but 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** n
8.   **while** ccw(next-top(S)$, p_i,$ top $(S)) \leq 0$
9.   POP(S)
10.   PUSH(S)
11. **return** S

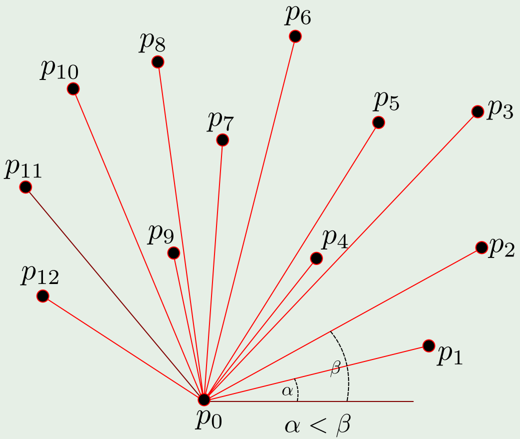
▷ The clockwise and counter clockwise algorithm◁

ccw(p_1, p_2, p_3)

1. return $(p_3 - p_1 \times p_2 - p_1)$

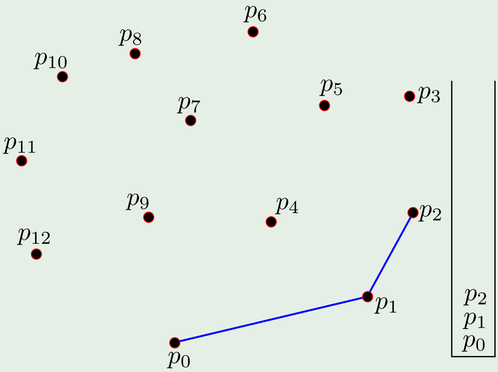
Example

Sort points using the smallest to largest polar coordinate



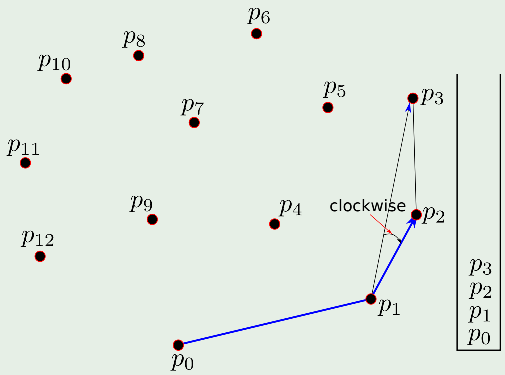
Example

Push the first points into the stack



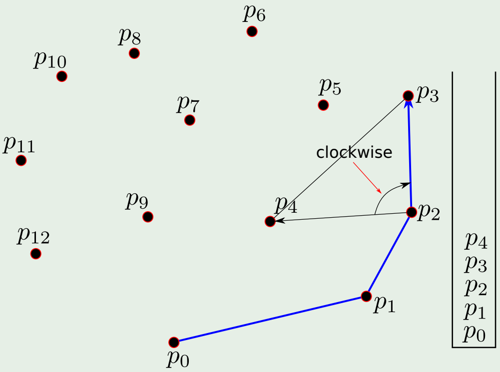
Example

ccw$(p_1, p_2, p_3) > 0$, do not get into the loop and push p_3 into S



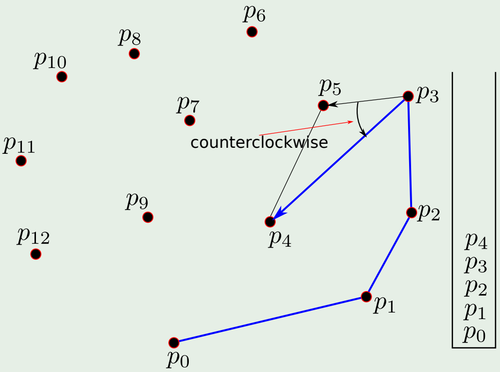
Example

ccw$(p_2, p_3, p_4) > 0$, do not get into the loop and push p_4 into S



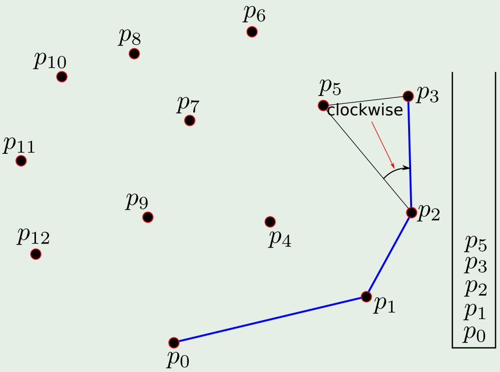
Example

Counterclockwise - Pop p_4



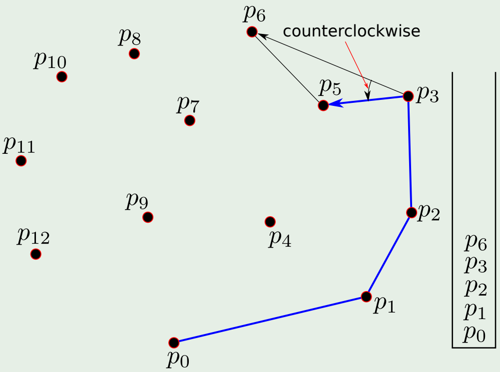
Example

Clockwise - Push p_5



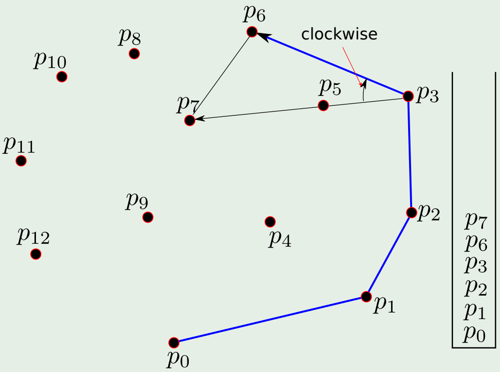
Example

Counterclockwise - Pop p_5 and push p_6



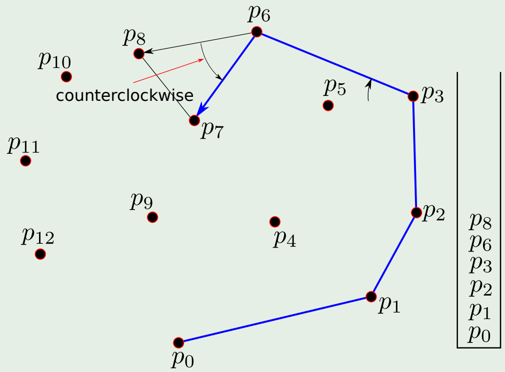
Example

Clockwise push p_7



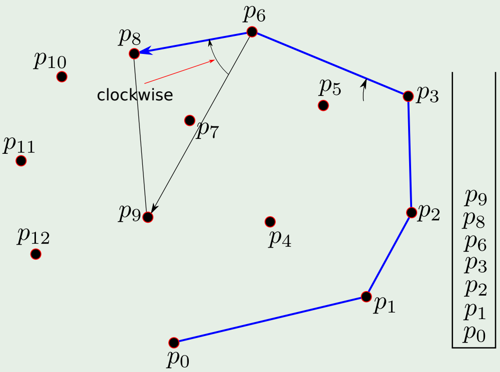
Example

Counterclockwise pop p_7 and push p_8



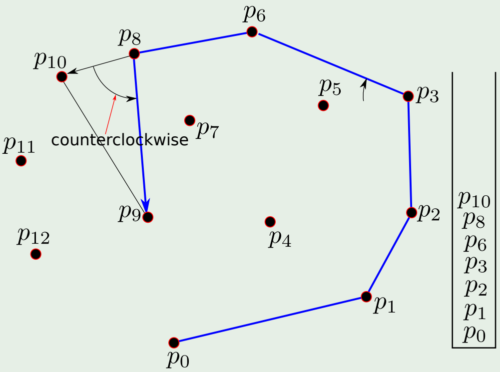
Example

Clockwise push p_9



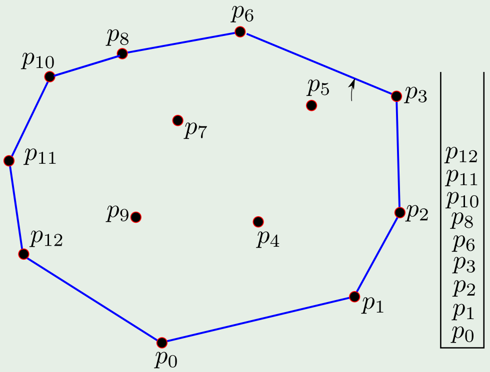
Example

Clockwise pop p_9 and push p_{10}



Example

Keep going until you finish



We have the following theorem for correctness of the algorithm

Theorem 33.1 (Correctness of Graham's scan)

If GRAHAM-SCAN executes on a set Q of points, where $|Q| \geq 3$, then at termination, the stack S consists of, from bottom to top, exactly the vertices of CH(Q) in counterclockwise order.

The proof is based in loop invariance

I leave this to you to read!!!

We have the following theorem for correctness of the algorithm

Theorem 33.1 (Correctness of Graham's scan)

If GRAHAM-SCAN executes on a set Q of points, where $|Q| \geq 3$, then at termination, the stack S consists of, from bottom to top, exactly the vertices of $\text{CH}(Q)$ in counterclockwise order.

The proof is based in loop invariance

I leave this to you to read!!!

Using Aggregate Analysis to obtain the complexity $O\left(n \log_2 n\right)$

Complexity

1. Line 2 takes $O\left(n \log_2 n\right)$ using Merge sort or Heap sort by using polar angles and cross product.
2. Lines 3-6 take $O\left(1\right)$.

Using Aggregate Analysis to obtain the complexity $O(n \log_2 n)$

Complexity

1. Line 2 takes $O(n \log_2 n)$ using Merge sort or Heap sort by using polar angles and cross product.
2. Lines 3-6 take $O(1)$.

For the Lines 7-10

The for loop executes at most $n-3$ times because we have $|Q|-3$ points left

Using Aggregate Analysis to obtain the complexity $O(n \log_2 n)$

Complexity

1. Line 2 takes $O(n \log_2 n)$ using Merge sort or Heap sort by using polar angles and cross product.
2. Lines 3-6 take $O(1)$.

For the Lines 7-10

The for loop executes at most $n-3$ times because we have $|Q|-3$ points left

Cinvestav

In addition

Given

PUSH takes $O(1)$ time:

- Each iteration takes $O\left(1\right)$ time not taking in account the time spent in the **while** loop in lines 8-9.

Then

The for loop take overall time $O(n)$ time

Here is the aggregate analysis

Here, we will prove that the overall time for all the times the **while** loop is touched by the **for** loop is going to be $O\left(n\right)$.

In addition

Given

PUSH takes $O(1)$ time:

- Each iteration takes $O(1)$ time not taking in account the time spent in the **while** loop in lines 8-9.

Then

The for loop take overall time $O(n)$ time

Here is the aggregate analysis

Here, we will prove that the overall time for all the times the **while** loop is touched by the **for** loop is going to be $O(n)$.

In addition

Given

PUSH takes $O(1)$ time:

- Each iteration takes $O(1)$ time not taking in account the time spent in the **while** loop in lines 8-9.

Then

The for loop take overall time $O(n)$ time

Here is the aggregate analysis

Here, we will prove that the overall time for all the times the **while** loop is touched by the **for** loop is going to be $O(n)$.

Aggregate Analysis

We have that

For $i = 0, 1, ..., n$, we push each point p_i into the stack S exactly once.

Remember Multipop?

We can pop at most the number of items that we push on it.

Thus

At least three points p_0, p_1 and p_m are never popped out of the stack!!!

- p_m is the last point being taken in consideration!!! With $m \leq n$

Aggregate Analysis

We have that

For $i = 0, 1, ..., n$, we push each point p_i into the stack S exactly once.

Remember Multipop?

We can pop at most the number of items that we push on it.

Thus

At least three points p_0, p_1 and p_m are never popped out of the stack!!!

- p_m is the last point being taken in consideration!!! With $m \leq n$

Aggregate Analysis

We have that

For $i = 0, 1, ..., n$, we push each point p_i into the stack S exactly once.

Remember Multipop?

We can pop at most the number of items that we push on it.

Thus

At least three points p_0, p_1 and p_m are never popped out of the stack!!!

- p_m is the last point being taken in consideration!!! With $m \leq n$

Aggregate Analysis

Thus

We have $m-2$ POP operations are performed in total!!! If we had pushed m elements into S.

Thus, each iteration of the while loop

It performs one POP, and there are at most $m-2$ iterations of the **while** loop altogether.

Now

Given that the test in line 8 takes $O(1)$ times, each call of the POP takes $O(1)$ and $m \leq n-1$.

Cinvestav

Aggregate Analysis

Thus

We have $m-2$ POP operations are performed in total!!! If we had pushed m elements into S.

Thus, each iteration of the **while** loop

It performs one POP, and there are at most $m-2$ iterations of the **while** loop altogether.

Now

Given that the test in line 8 takes $O(1)$ times, each call of the POP takes $O(1)$ and $m \leq n-1$.

Aggregate Analysis

Thus

We have $m-2$ POP operations are performed in total!!! If we had pushed m elements into S.

Thus, each iteration of the **while** loop

It performs one POP, and there are at most $m-2$ iterations of the **while** loop altogether.

Now

Given that the test in line 8 takes $O(1)$ times, each call of the POP takes $O(1)$ and $m \leq n-1$.

Cinvestav

Aggregate Analysis

We have that

The total time of the **while** loop is $O(n)$.

Finally

The Running Time of $GRAHAM-SCAN$ is $O(n \log_2 n)$

Aggregate Analysis

We have that

The total time of the **while** loop is $O(n)$.

Finally

The Running Time of $GRAHAM - SCAN$ is $O(n \log_2 n)$

Outline

Jarvis' March

Jarvis' March Basics

- It computes CH by using
  - A technique called Package Wrapping.
  - At each point calculate the minimum polar angle.
  - Create a left and right chain with the convex hull points.

Jarvis' March

Jarvis' March Basics

- It computes CH by using
  - A technique called Package Wrapping.
  - At each point calculate the minimum polar angle.
  - Create a left and right chain with the convex hull points.

Jarvis' March

Jarvis' March Basics

- It computes CH by using
  - A technique called Package Wrapping.
  - At each point calculate the minimum polar angle.
  - Create a left and right chain with the convex hull points.

Cinvestav

Jarvis' March

Jarvis' March Basics

- It computes CH by using
  - A technique called Package Wrapping.
  - At each point calculate the minimum polar angle.
  - Create a left and right chain with the convex hull points.

Cinvestav

Formally

Jarvis's march builds a sequence

$H = \langle p_0, p_1, p_2, ..., p_{h-1} \rangle$ of the vertices of CH(Q)

First

We start with p_0 the next vertex p_1 in the convex hull has the smallest polar angle with respect to p_0.

Next

- p_2 has the smallest polar angle with respect to p_1.

Formally

Jarvis's march builds a sequence

$H = \langle p_0, p_1, p_2, ..., p_{h-1} \rangle$ of the vertices of CH(Q)

First

We start with p_0 the next vertex p_1 in the convex hull has the smallest polar angle with respect to p_0.

Next

- p_2 has the smallest polar angle with respect to p_1.

Formally

Jarvis's march builds a sequence

$H = \langle p_0, p_1, p_2, ..., p_{h-1} \rangle$ of the vertices of $\mathrm{CH}(Q)$

First

We start with p_0 the next vertex p_1 in the convex hull has the smallest polar angle with respect to p_0.

Next

- p_2 has the smallest polar angle with respect to p_1.

Now

Then

When we reach the highest vertex, p_k (Breaking ties by choosing the farthest such vertex), we have constructed **the right chain** of CH(Q).

To construct **the left chain**

We start at p_k, then we choose p_{k+1} as the point with the smallest polar angle with respect to p_k negative , but from ***the negative x-axis***.

Next

The next point is selected in the same manner until we have reached p_0.

Now

Then

When we reach the highest vertex, p_k (Breaking ties by choosing the farthest such vertex), we have constructed **the right chain** of $\mathrm{CH}(Q)$.

To construct **the left chain**

We start at p_k, then we choose p_{k+1} as the point with the smallest polar angle with respect to p_k negative , but from ***the negative x-axis***.

Next

The next point is selected in the same manner until we have reached p_0.

Now

Then

When we reach the highest vertex, p_k (Breaking ties by choosing the farthest such vertex), we have constructed **the right chain** of CH(Q).

To construct **the left chain**

We start at p_k, then we choose p_{k+1} as the point with the smallest polar angle with respect to p_k negative , but from ***the negative x-axis***.

Next

The next point is selected in the same manner until we have reached p_0.

Cinvestav

Jarvis' March

Example

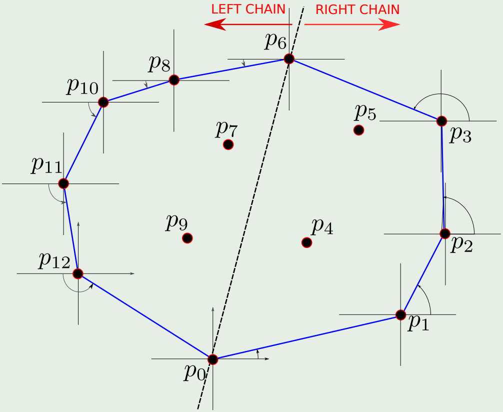


Figure: Wrapping the Gift. Here the Right Chain finishes at p_6, then the Left Chain is started

Complexity

Something Notable

Complexity $O\left(hn\right)$

- h number of points in CH.
- $O\left(n\right)$ for finding the minimum angle and the farthest point by y-axis

Complexity

Something Notable

Complexity $O\left(hn\right)$

- h number of points in CH.
- $O\left(n\right)$ for finding the minimum angle and the farthest point by y-axis

Complexity

Something Notable

Complexity $O(hn)$

- h number of points in CH.
- $O(n)$ for finding the minimum angle and the farthest point by y-axis

Complexity

Something Notable

Complexity $O(hn)$

- h number of points in CH.
- $O(n)$ for finding the minimum angle and the farthest point by y-axis