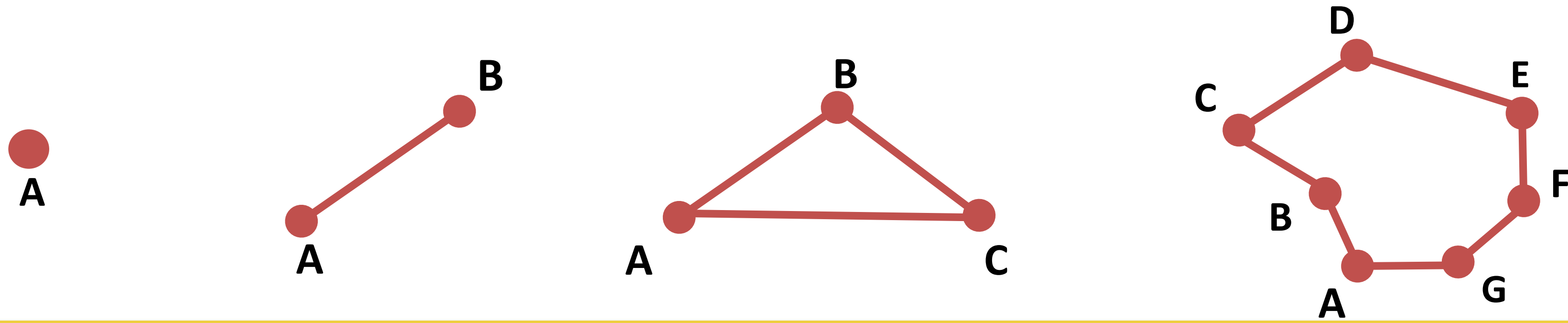
Fundamentals of Computational Geometry

CSCI 587: Lecture 2

08/28/2024

What is Computational Geometry?

- Design, Analysis and Implementation of *efficient algorithms* for solving geometric problems, e.g., problems involving points, lines, segments, triangles, polygons



Many Applications

Robotics

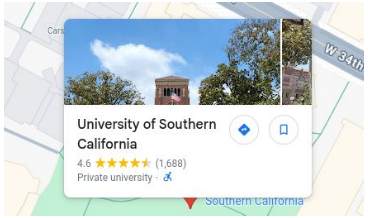
Geographic
Information Systems

Computer Graphics

Fundamental Operations

- In computational geometry, the most primitive object is a point.

A = (0, 1)

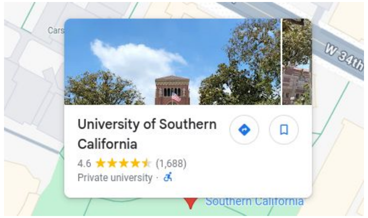


B = (34.022, -118.285)

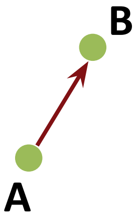
Fundamental Operations

- In computational geometry, the most primitive object is a point.
- Common Operations: *addition, subtraction*

A = (0, 1)

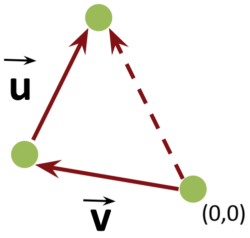


B = (34.022, -118.285)



$$(x_1, y_1) - (x_2, y_2) := (x_1 - x_2, y_1 - y_2)$$

subtraction

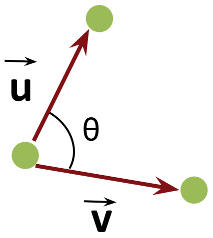


$$(x_1, y_1) + (x_2, y_2) := (x_1 + x_2, y_1 + y_2)$$

addition

Fundamental Operations

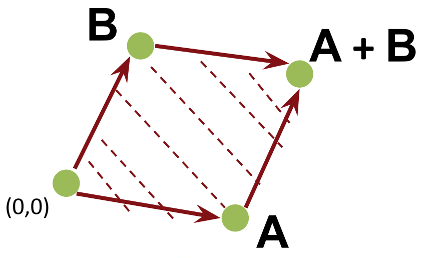
- In computational geometry, the most primitive object is a point.
- Common Operations: *addition, subtraction, dot product, cross product*



$$(x_1, y_1) \cdot (x_2, y_2) = x_1 x_2 + y_1 y_2$$

$$u \cdot v = \|u\| \|v\| \cos(\theta)$$

dot product



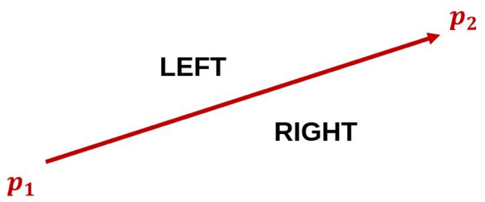
$$(x_1, y_1) \times (x_2, y_2) := \det\left(\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \end{bmatrix}\right) = x_1 y_2 - y_1 x_2$$

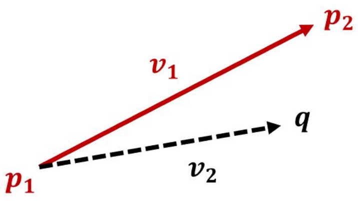
$$u \times v = \|u\| \|v\| \sin(\theta)$$

cross product

Line Side Test

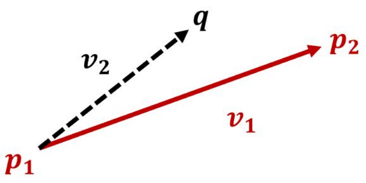
- Decide whether a point *q* is on the left or right of a line segment
  - Construct vectors: $v_1 = p_2 - p_1$ and $v_2 = q - p_1$
  - Compute the cross product v_1 and v_2
  - Compare value to 0





$v_1 \times v_2 < 0$

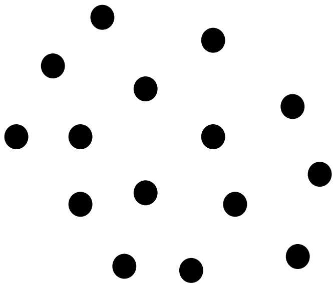
q is *right* of v_1



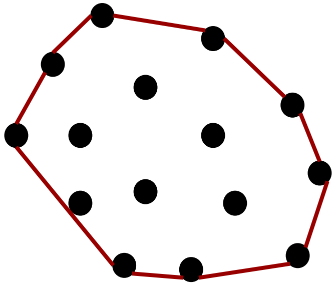
$v_1 \times v_2 > 0$

q is *left* of v_1

Convexity

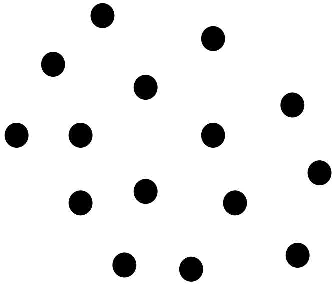


A set of points **P** in a Euclidean space

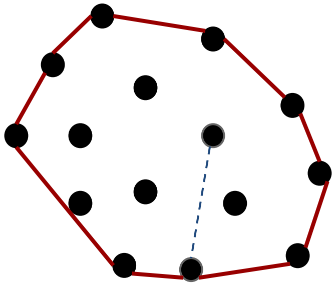


Convex Hull

Convexity

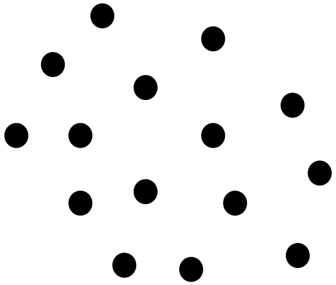


A set of points **P** in a Euclidean space

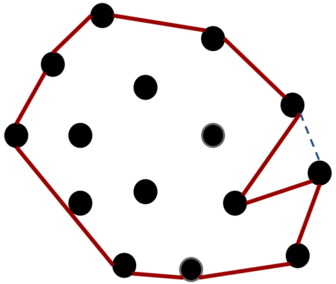


Convex Hull

Convexity

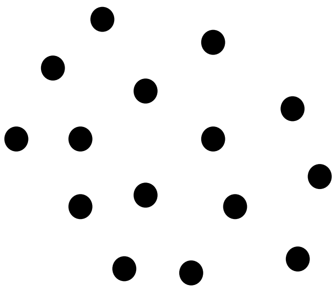


A set of points **P** in a Euclidean space

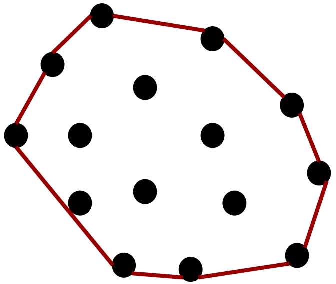


Concave

Convexity



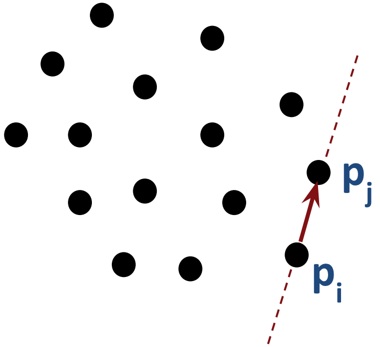
A set of points **P** in a Euclidean space



Convex Hull

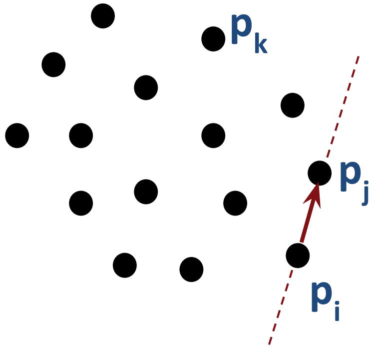
Definition: The convex hull of a set of points **P** is the boundary of the convex closure of **P** . That is, it is the *smallest convex polygon* that contains *all* of the points in **P** , either on its boundary or interior.

Algorithms for 2D Convex Hull



Claim: A directed segment between a pair of points p_i, p_j is on the convex hull if and only if all other points are to the left of the ray through p_i and p_j.

Algorithms for 2D Convex Hull

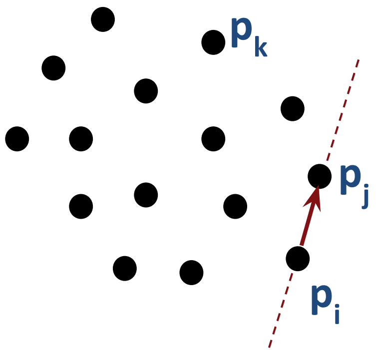


Brute Force Algorithm

1. Try every pair of points p_i, p_j
2. Perform Line Side Test on every other point p_k
3. If every p_k is on the left:
   a. Add $p_i \rightarrow p_j$ to the hull
4. Sort the final set of edges into counterclockwise order

Claim: A directed segment between a pair of points p_i, p_j is on the convex hull if and only if all other points are to the left of the ray through p_i and p_j.

Algorithms for 2D Convex Hull

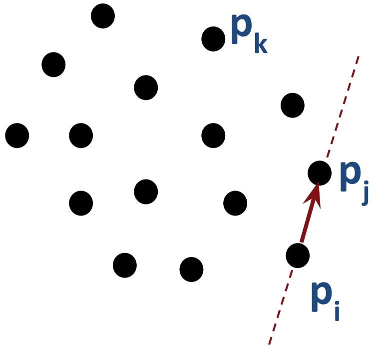


Brute Force Algorithm ⟹ **Complexity?**

1. Try every pair of points $\mathbf{p_i}$, $\mathbf{p_j}$
2. Perform Line Side Test on every other point $\mathbf{p_k}$
3. If every $\mathbf{p_k}$ is on the left:
   a. Add $\mathbf{p_i} \rightarrow \mathbf{p_j}$ to the hull
4. Sort the final set of edges into counterclockwise order

Claim: A directed segment between a pair of points $\mathbf{p_i}$, $\mathbf{p_j}$ is <u>on</u> the convex hull if and only if all other points are to the left of the ray through $\mathbf{p_i}$ and $\mathbf{p_j}$.

Algorithms for 2D Convex Hull

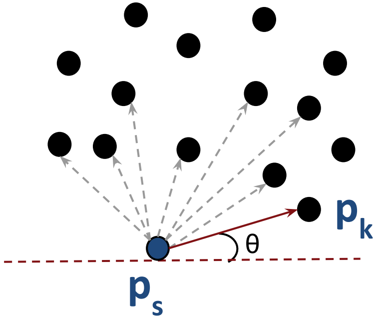


Brute Force Algorithm $\Longrightarrow$ $O(N^3)$

1. Try every pair of points p_i, p_j
2. Perform Line Side Test on every other point p_k
3. If every p_k is on the left:
   a. Add $p_i \rightarrow p_j$ to the hull
4. Sort the final set of edges into counterclockwise order

Claim: A directed segment between a pair of points p_i, p_j is <u>on</u> the convex hull if and only if all other points are to the left of the ray through p_i and p_j.

Algorithms for 2D Convex Hull

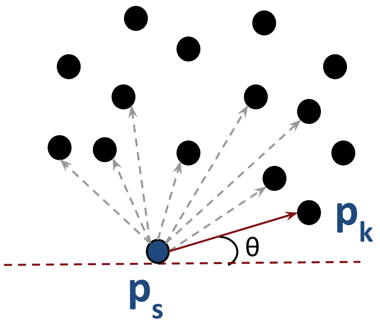


A little bit faster Algorithm

1. Take the point with the lowest y-coordinate $\mathbf{p_s}$
2. Measure the angle from $\mathbf{p_s}$ to all the other points $\mathbf{p_k}$
3. Select the point with the *smallest angle*
   a. Add $\mathbf{p_s} \rightarrow \mathbf{p_k}$ to the hull
4. Find the point $\mathbf{p_u}$ that has the smallest angle with respect to $(\mathbf{p_s}, \mathbf{p_k})$
5. Continue until all points are exhausted

Claim: The lowest y-coordinate point $\mathbf{p_s}$ is *always* in the convex hull.

Algorithms for 2D Convex Hull

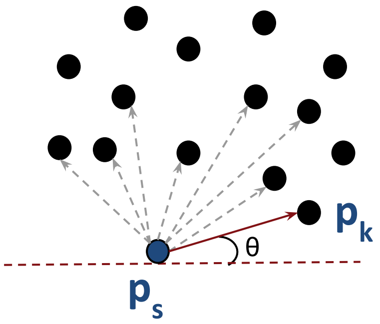


A little bit faster Algorithm ⟹ **Complexity?**

1. Take the point with the lowest y-coordinate $\mathbf{p_s}$
2. Measure the angle from $\mathbf{p_s}$ to all the other points $\mathbf{p_k}$
3. Select the point with the *smallest angle*
   a. Add $\mathbf{p_s} \rightarrow \mathbf{p_k}$ to the hull
4. Find the point $\mathbf{p_u}$ that has the smallest angle with respect to $(\mathbf{p_s}, \mathbf{p_k})$
5. Continue until all points are exhausted

Claim: The lowest y-coordinate point $\mathbf{p_s}$ is *always* in the convex hull.

Algorithms for 2D Convex Hull

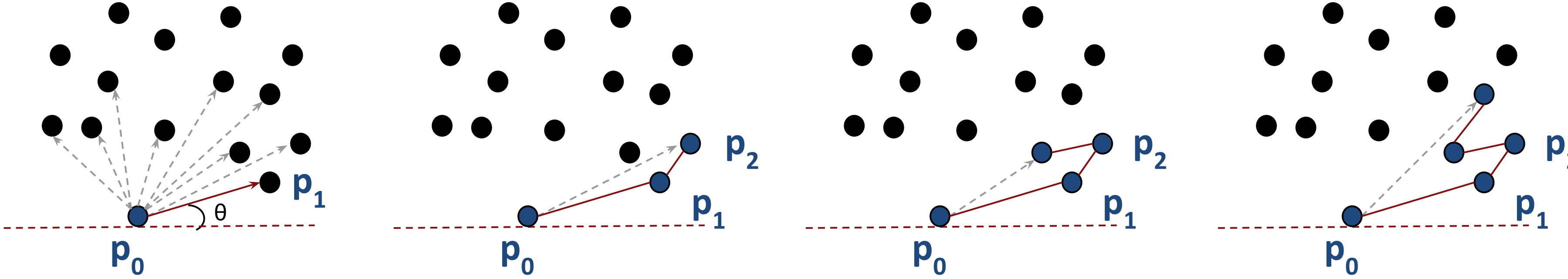


A little bit faster Algorithm ⟹ **$O(N^2)$**

1. Take the point with the lowest y-coordinate $\mathbf{p_s}$
2. Measure the angle from $\mathbf{p_s}$ to all the other points $\mathbf{p_k}$
3. Select the point with the *smallest angle*
   a. Add $\mathbf{p_s} \rightarrow \mathbf{p_k}$ to the hull
4. Find the point $\mathbf{p_u}$ that has the smallest angle with respect to $(\mathbf{p_s}, \mathbf{p_k})$
5. Continue until all points are exhausted

Claim: The lowest y-coordinate point $\mathbf{p_s}$ is *always* in the convex hull.

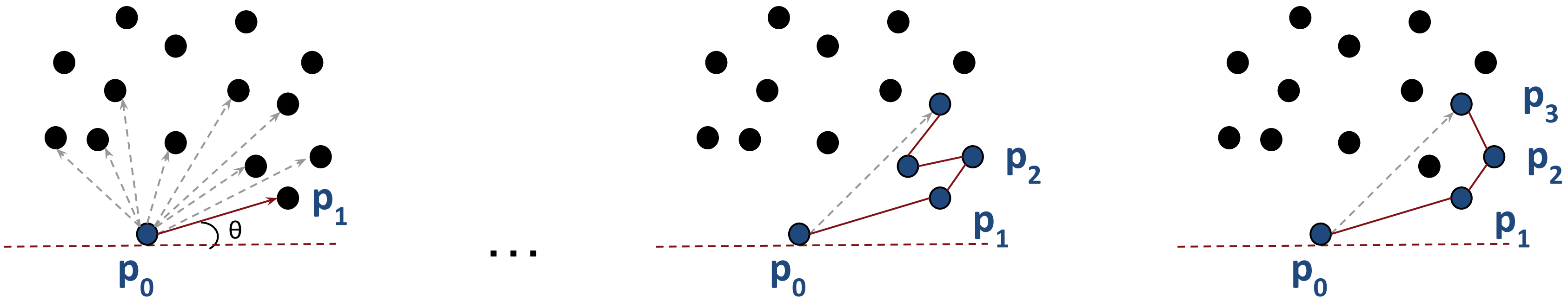
Algorithms for 2D Convex Hull



we are making a **right** turn!

Claim: The lowest y-coordinate point $\mathbf{p_s}$ is *always* in the convex hull.

Algorithms for 2D Convex Hull

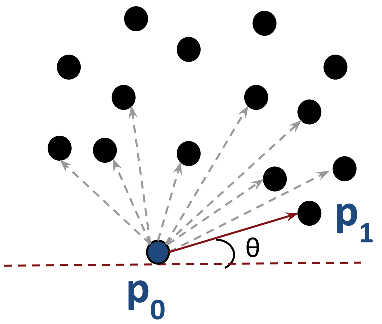


we are making a **right** turn!

Remove the last added point from Hull

Claim: The lowest y-coordinate point $\mathbf{p_s}$ is *always* in the convex hull.

Algorithms for 2D Convex Hull

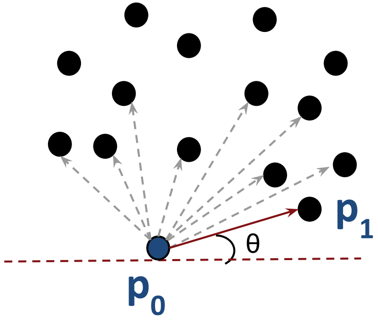


Graham Scan Algorithm

1. Find lowest y-coordinate point $\mathbf{p_0}$
2. Sort the points counterclockwise by *their angle with* $\boldsymbol{p_0}$
3. Hull = [$\mathbf{p_0}$, $\mathbf{p_1}$]
4. For each point $\mathbf{p_i}$
   a. If LineSideTest(Hull, $\mathbf{p_i}$) is RIGHT
      i. H.pop() // remove last element
   b. H.add($\mathbf{p_i}$)

Claim: The lowest y-coordinate point $\mathbf{p_s}$ is *always* in the convex hull.

Algorithms for 2D Convex Hull

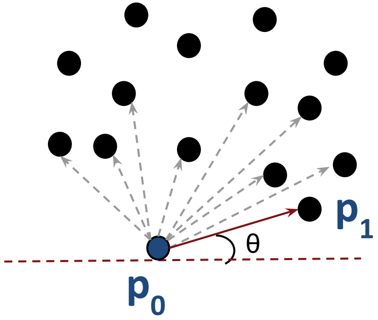


Graham Scan Algorithm ⟹ **Complexity?**

1. Find lowest y-coordinate point $\mathbf{p}_0$
2. Sort the points counterclockwise by *their angle with* $\boldsymbol{p_0}$
3. Hull = [$\mathbf{p}_0$, $\mathbf{p}_1$]
4. For each point $\mathbf{p}_i$
   a. If LineSideTest(Hull, $\mathbf{p}_i$) is RIGHT
      i. H.pop()   // remove last element
   b. H.add($\mathbf{p}_i$)

Claim: The lowest y-coordinate point $\mathbf{p}_s$ is *always* in the convex hull.

Algorithms for 2D Convex Hull

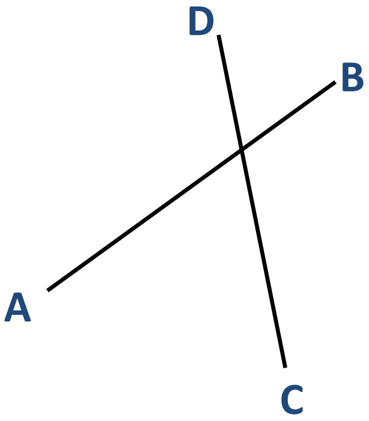


Graham Scan Algorithm ⟹ **O(NlogN)**

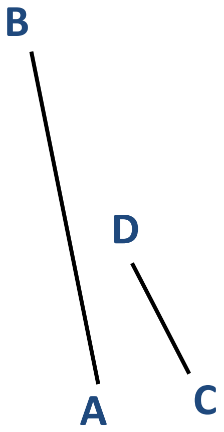
1. Find lowest y-coordinate point $\mathbf{p_0}$
2. Sort the points counterclockwise by *their angle with* $\boldsymbol{p_0}$
3. Hull = [$\mathbf{p_0}$, $\mathbf{p_1}$]
4. For each point $\mathbf{p_i}$
   a. If LineSideTest(Hull, $\mathbf{p_i}$) is RIGHT
      i. H.pop() // remove last element
   b. H.add($\mathbf{p_i}$)

Claim: The lowest y-coordinate point $\mathbf{p_s}$ is *always* in the convex hull.

Intersections



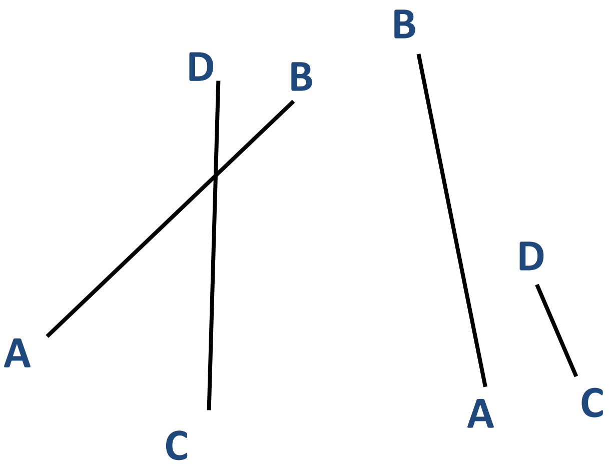
Intersecting pair



Non Intersecting pair

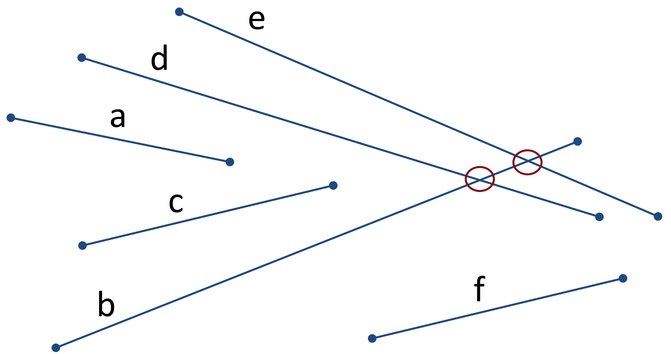
When do segments **AB** and **CD** *intersect*?

Intersections



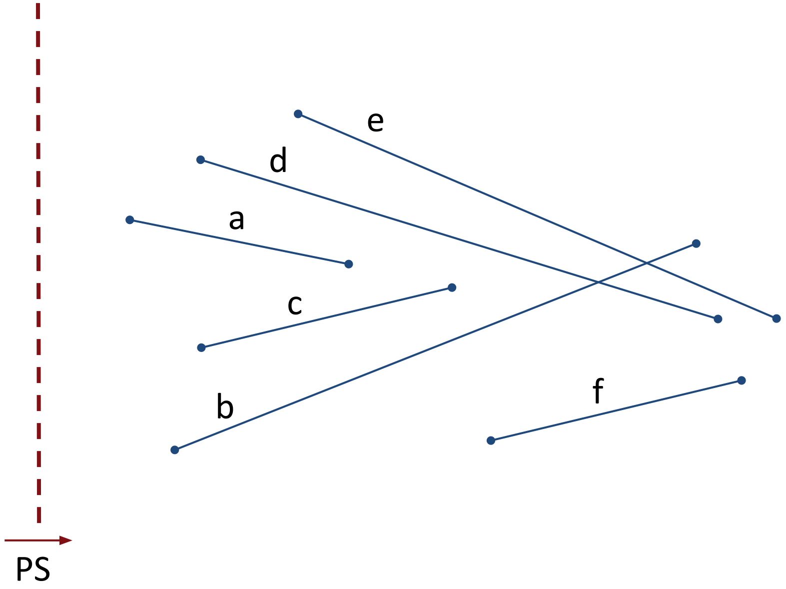
- When do segments **AB** and **CD** *intersect*?
  - We can take every point in one segment and test if it exists in the other.
  - We can check if A and B are on opposite sides of **CD** segment.
    - *eg.* LineSideTest(CD, A) = RIGHT and LineSideTest(CD, B) is LEFT

Intersections

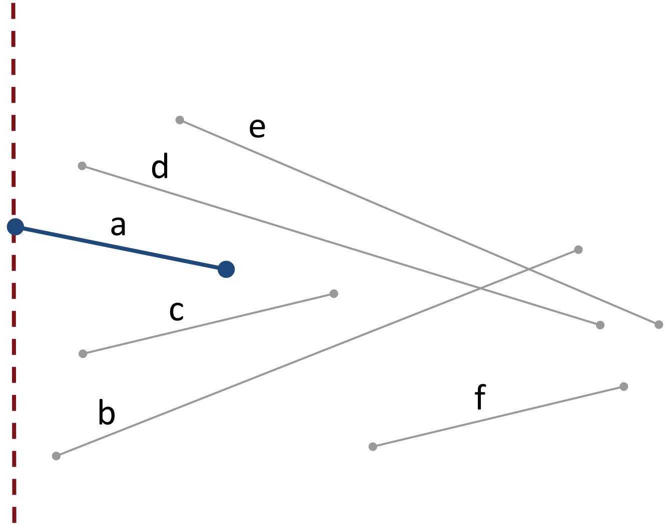


- What if we have **N** segments and want to detect **k** intersections?
  - We can perform the same checks for <u>all</u> possible pairs of segments.
  - **Complexity:** $O(N^2)$

Intersections: Plane Sweep Algorithm

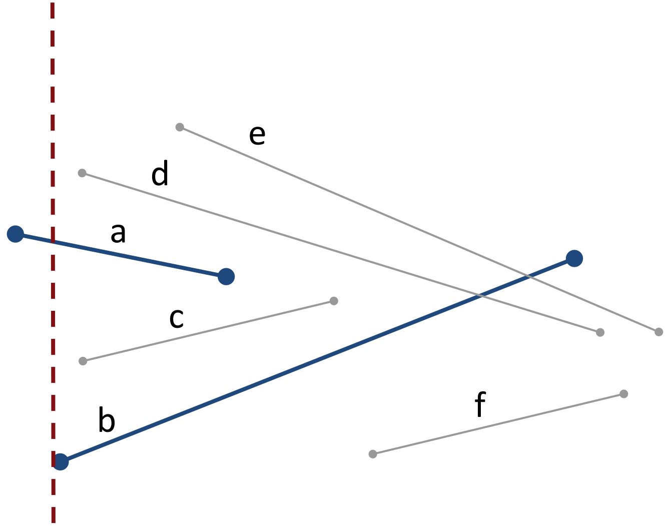


Intersections: Plane Sweep Algorithm



Currently exploring: a

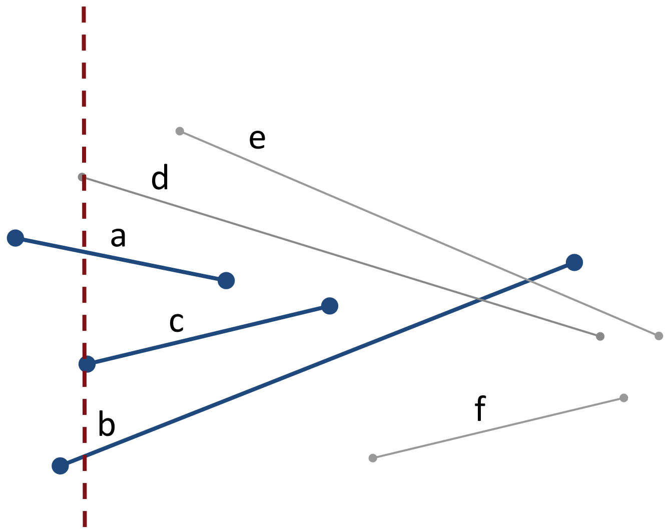
Intersections: Plane Sweep Algorithm



Currently exploring: b, a

➔ b does not *intersect* with a

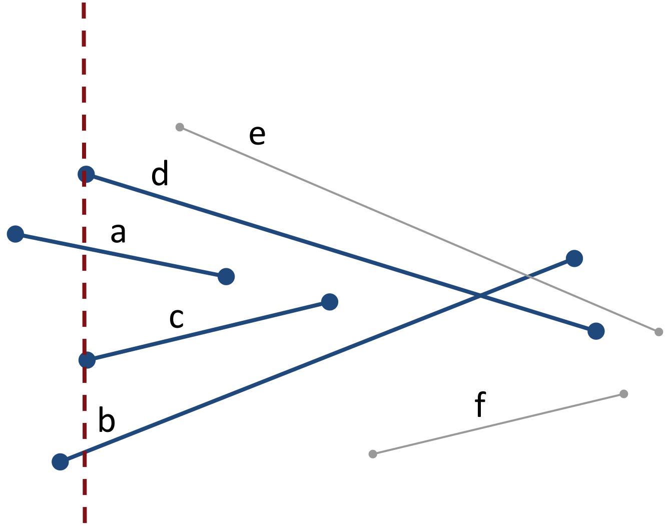
Intersections: Plane Sweep Algorithm



Currently exploring: b, c, a

➔ c does not *intersect* with b or a

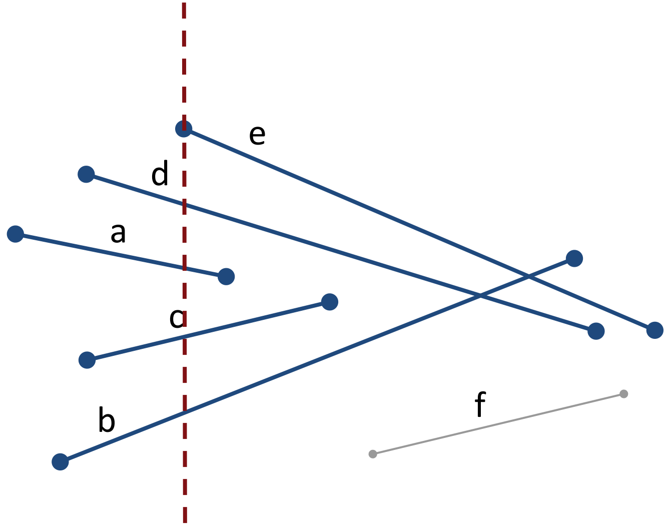
Intersections: Plane Sweep Algorithm



Currently exploring: b, c, a, d

→ d does not *intersect* with a

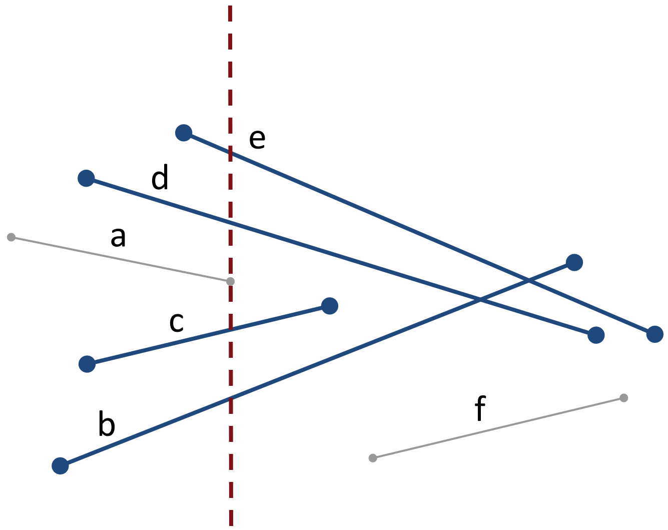
Intersections: Plane Sweep Algorithm



Currently exploring: b, c, a, d, e

➔ e does not *intersect* with d

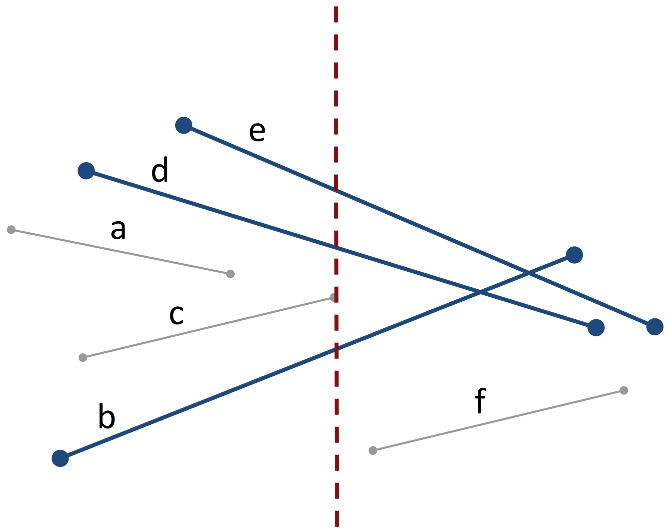
Intersections: Plane Sweep Algorithm



Currently exploring: b, c, ~~a~~, d, e

➔ c does not *intersect* with d !

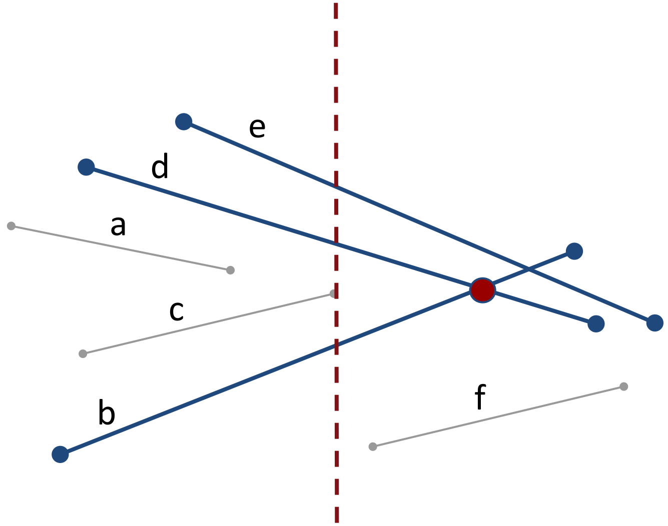
Intersections: Plane Sweep Algorithm



Currently exploring: b, ~~c~~, d, e

→ b *intersects* with d !

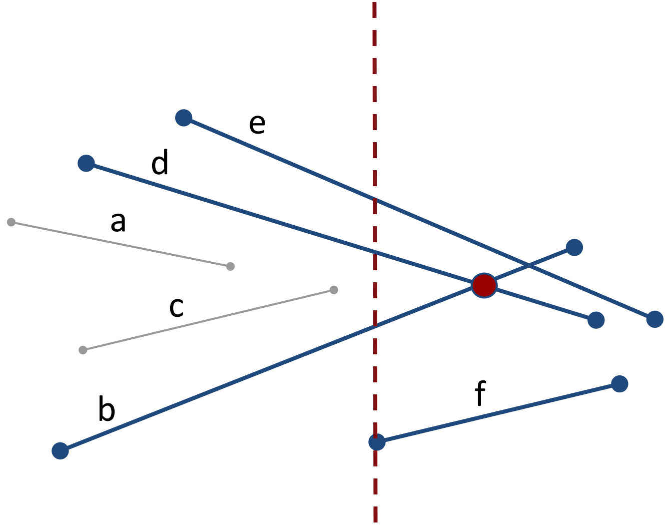
Intersections: Plane Sweep Algorithm



Currently exploring: b, d, e

→ b *intersects* with d !

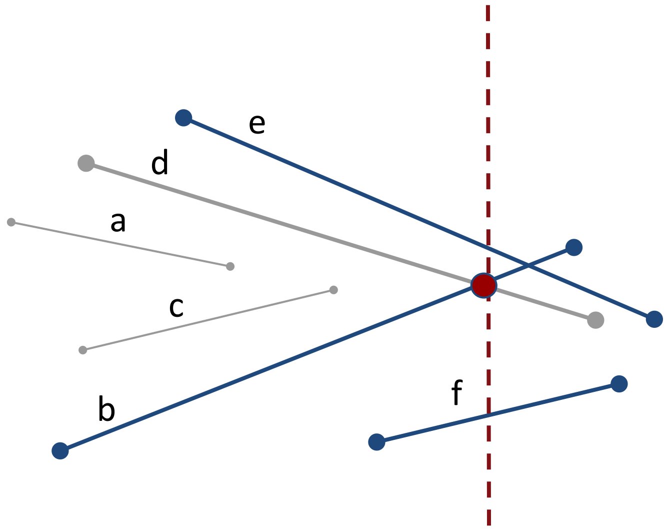
Intersections: Plane Sweep Algorithm



Currently exploring: f, b, d, e

➔ f does not *intersect* with b

Intersections: Plane Sweep Algorithm

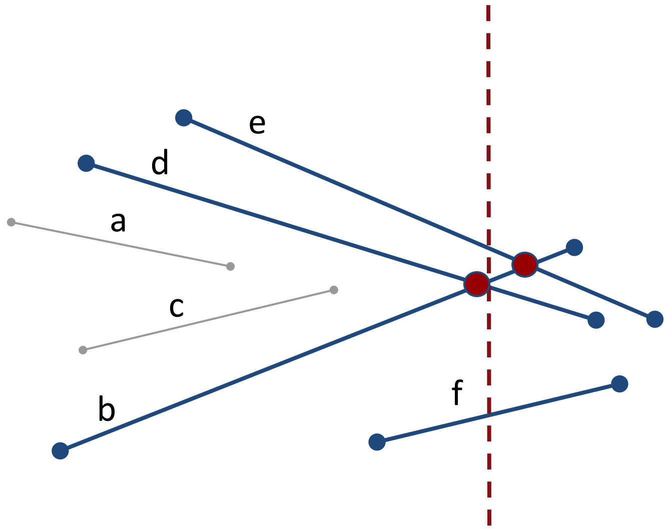


Currently exploring: f, b, ~~d~~, e

➔ e *intersects* with b !

Note: Treat intersection point as endpoint

Intersections: Plane Sweep Algorithm

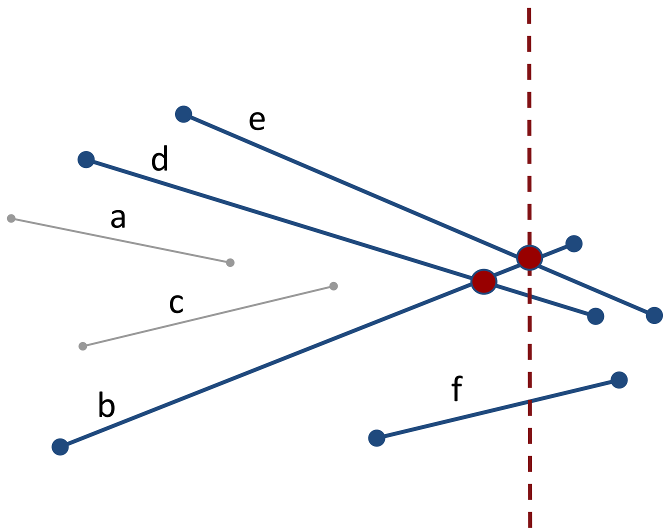


Currently exploring: f, b, d, e

→ e *intersects* with b !

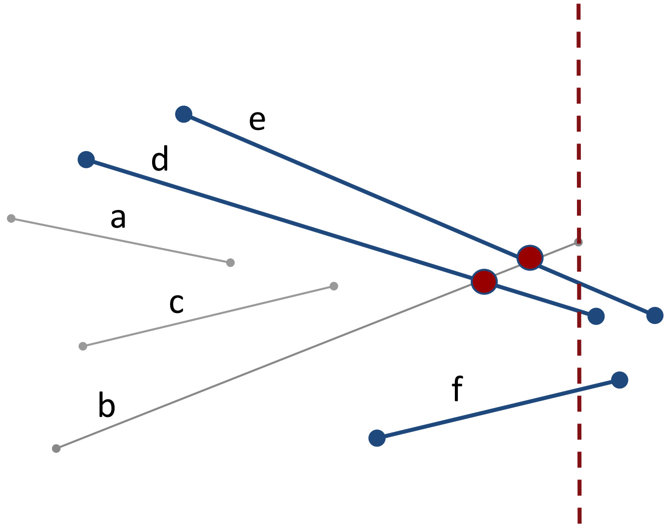
Note: Treat intersection point as endpoint

Intersections: Plane Sweep Algorithm



Currently exploring: f, b, d, e

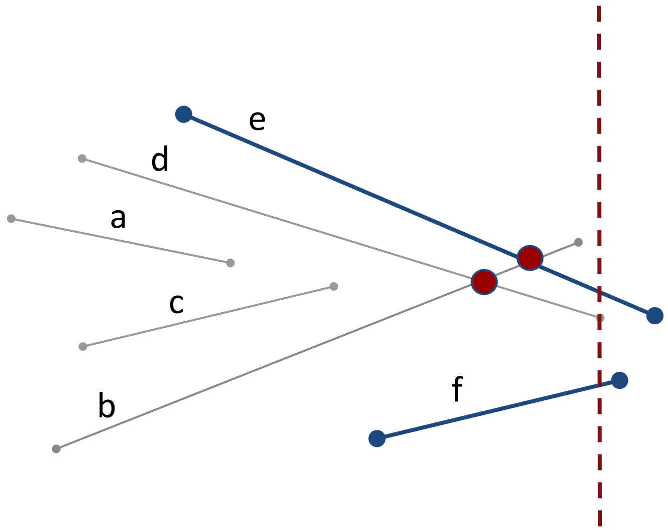
Intersections: Plane Sweep Algorithm



Currently exploring: f, d, e

➔ f does not *intersect* with d

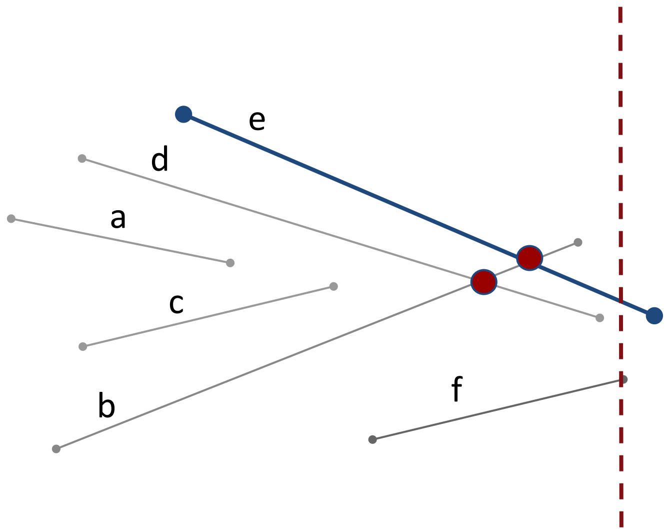
Intersections: Plane Sweep Algorithm



Currently exploring: f, e

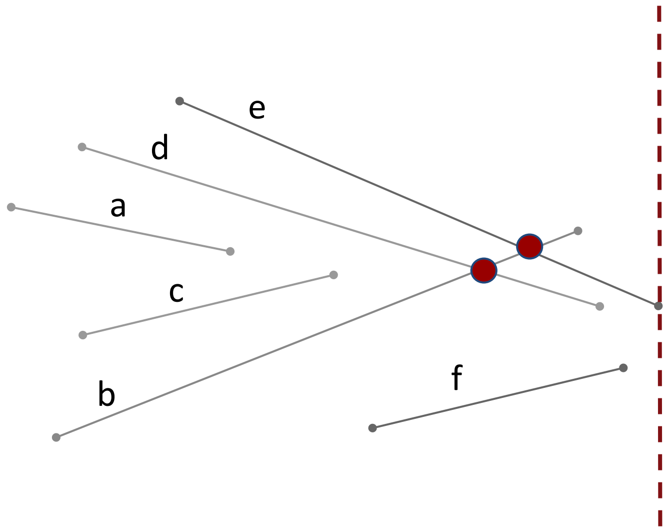
➔ f does not *intersect* with e

Intersections: Plane Sweep Algorithm



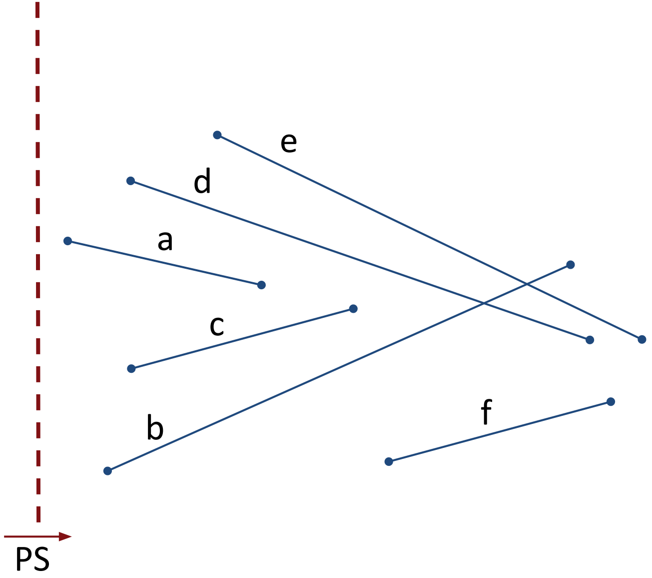
Currently exploring: e

Intersections: Plane Sweep Algorithm



Currently exploring:

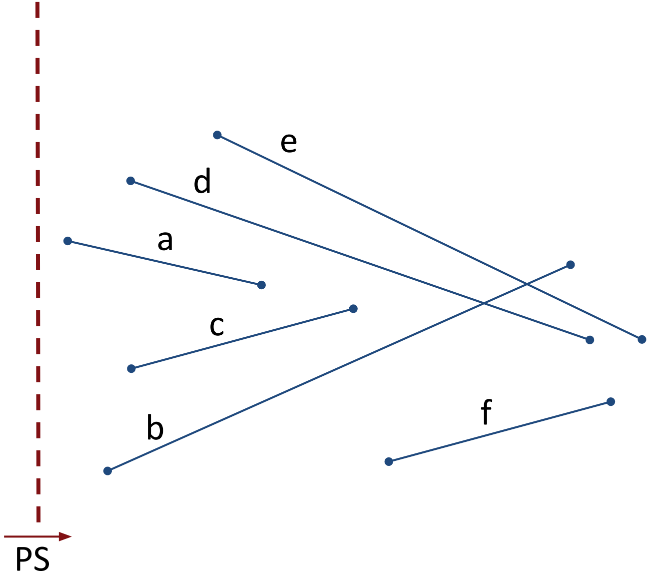
Intersections: Plane Sweep Algorithm



Plane Sweep Algorithm

1. Queue **EQ** = start and end of each segment *Si;* List **SL** = {}
2. For pi in **EQ**:
   a. If pi is *start* point:
      i. **SL**.add(pi); Intersects(Si, succ(Si)); Intersects(Si, predec(Si));
   b. If pi is *end* point:
      i. **SL**.delete(pi); Intersects(succ(Si), predec(Si));
   c. If *cross event* for Si, Sj:
      i. Remove Si from **SL**; Intersects(Sj, new neighbor);
      ii. Remove Sj from **SL**; Intersects(Si, new neighbor);

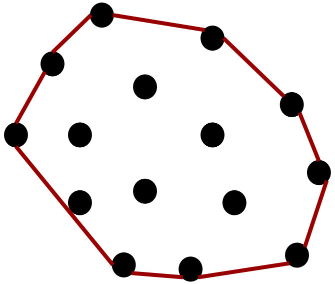
Intersections: Plane Sweep Algorithm



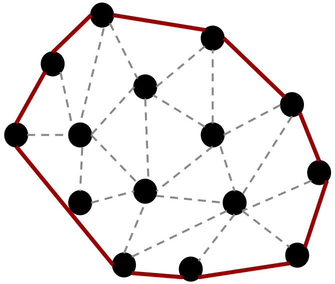
Plane Sweep Algorithm ⟹ **O(NlogN)**

1. Queue **EQ** = start and end of each segment *Si;* List **SL** = {}
2. For pi in **EQ**:
    a. If pi is *start* point:
        i. **SL**.add(pi); Intersects(Si, succ(Si)); Intersects(Si, predec(Si));
    b. If pi is *end* point:
        i. **SL**.delete(pi); Intersects(succ(Si), predec(Si));
    c. If *cross event* for Si, Sj:
        i. Remove Si from **SL**; Intersects(Sj, new neighbor);
        ii. Remove Sj from **SL**; Intersects(Si, new neighbor);

Triangulation



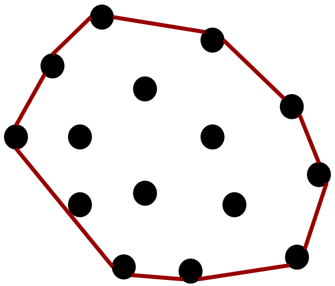
Convex Hull



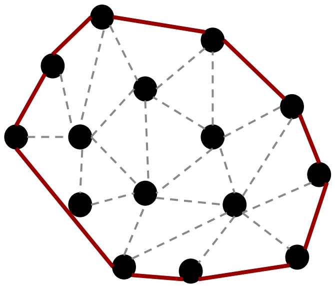
Triangulated Convex Hull

Definition: A **triangulation** is the process of subdividing a complex object (e.g. convex hull) into a disjoint collection of “simpler” objects (e.g. triangles).

Triangulation



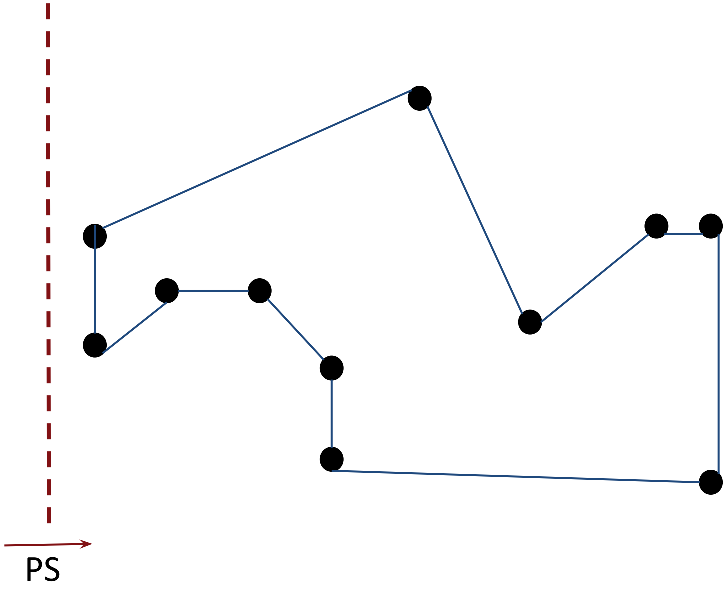
Convex Hull



Triangulated Convex Hull

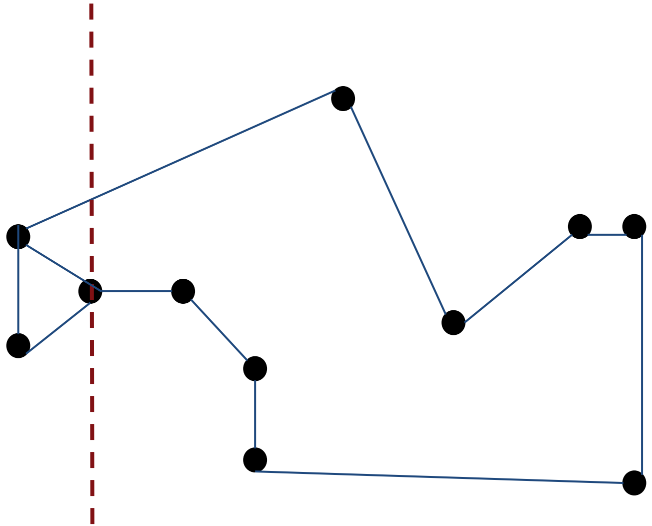
We can again use the **Plane Sweep Algorithm**!

Triangulation: Plane Sweep Algorithm

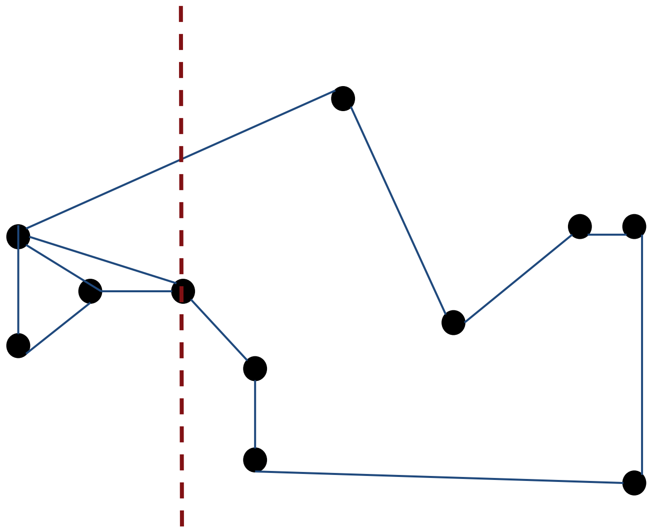


Intuition: Try to triangulate everything you can that is left from the sweep by *adding diagonals*.

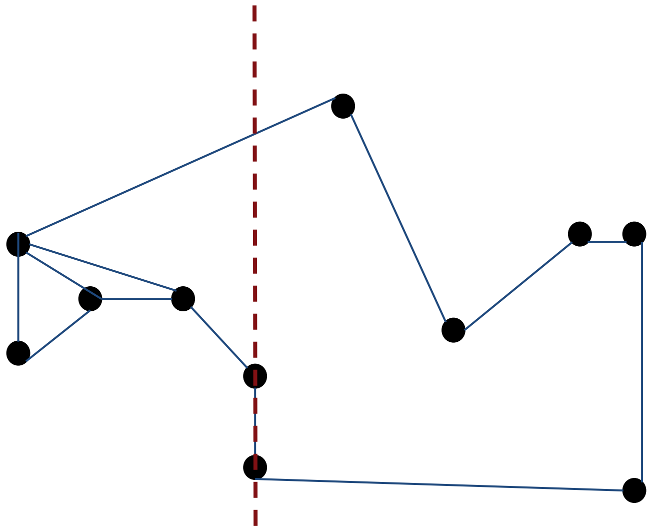
Triangulation: Plane Sweep Algorithm



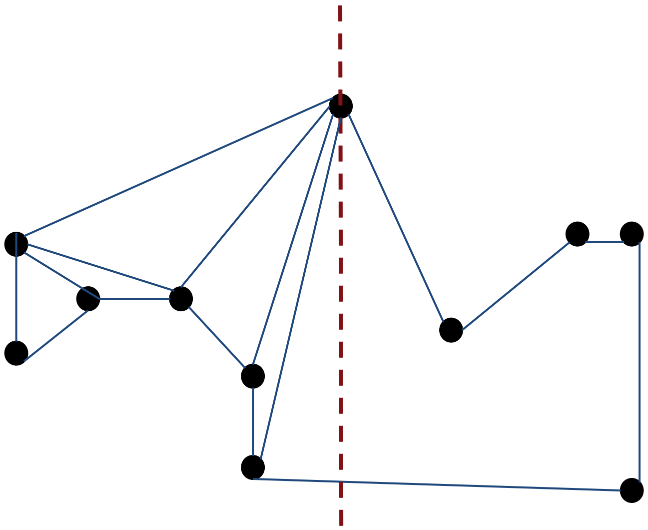
Triangulation: Plane Sweep Algorithm



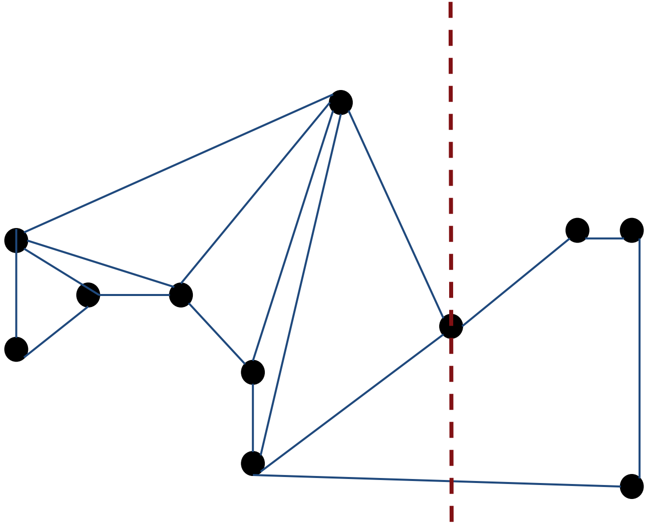
Triangulation: Plane Sweep Algorithm



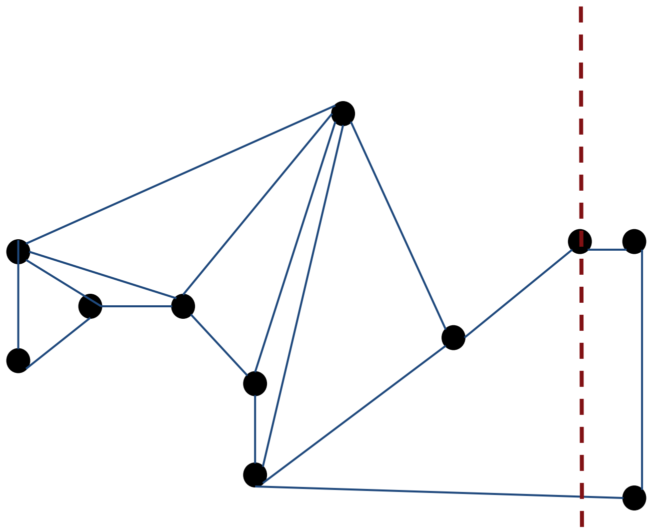
Triangulation: Plane Sweep Algorithm



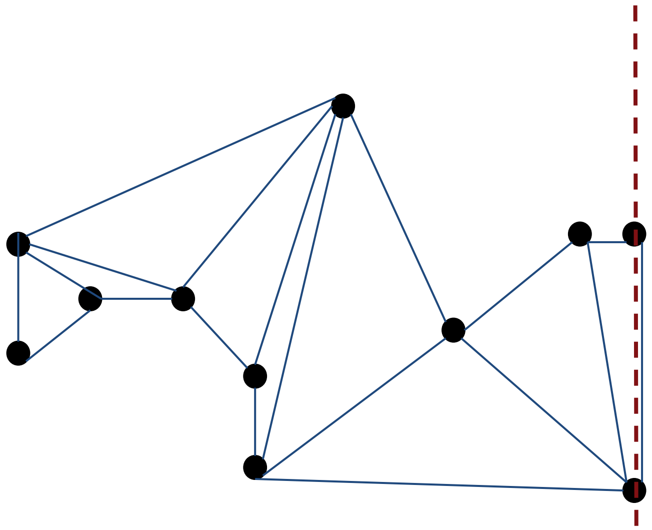
Triangulation: Plane Sweep Algorithm



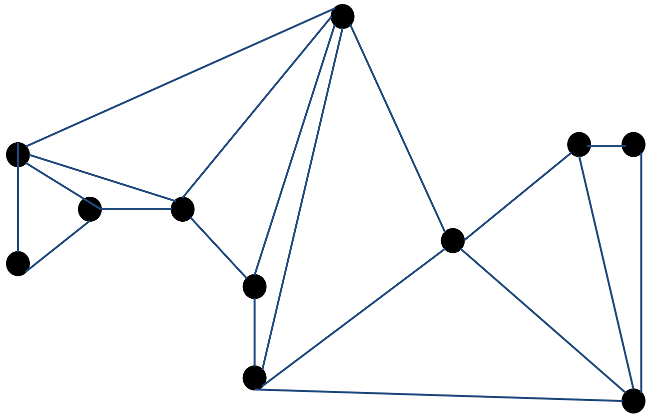
Triangulation: Plane Sweep Algorithm



Triangulation: Plane Sweep Algorithm

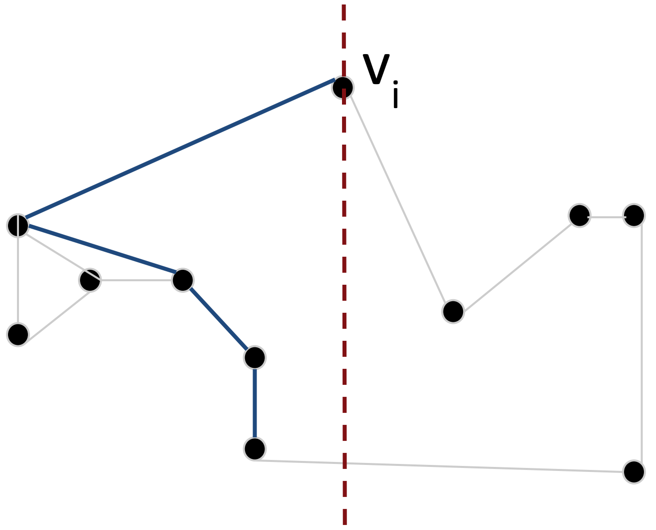


Triangulation: Plane Sweep Algorithm



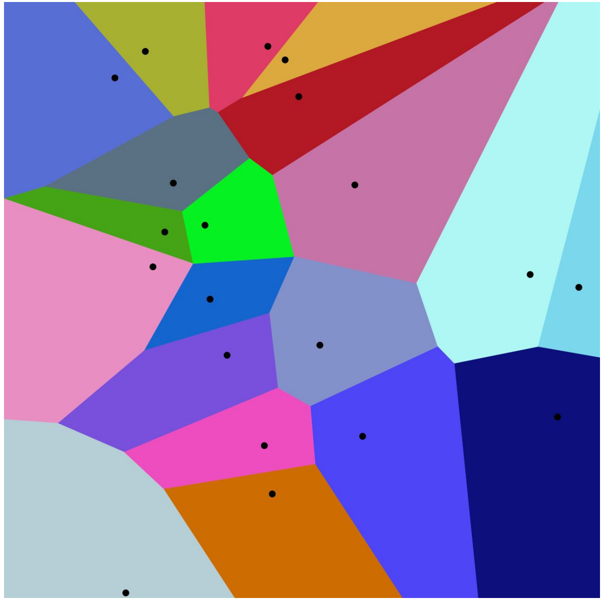
How can we determine if a region is ***un-triangulated***?

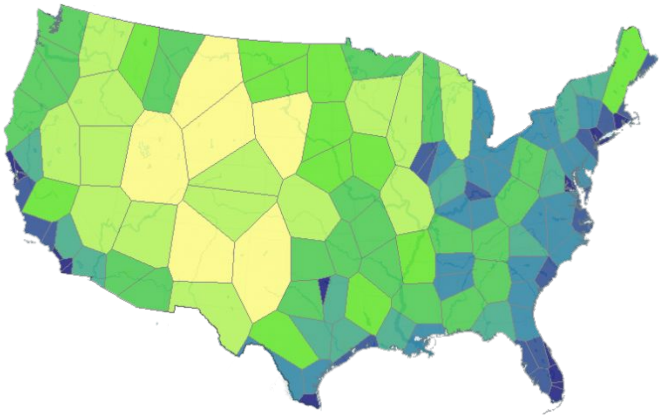
Triangulation: Plane Sweep Algorithm



Lemma: For i≥2, after processing vertex v_i, **if** there are 2 x-monotone *chains*, a lower and an upper chain, and one has multiple edges, then this is an **untriangulated region**.

Voronoi Diagrams

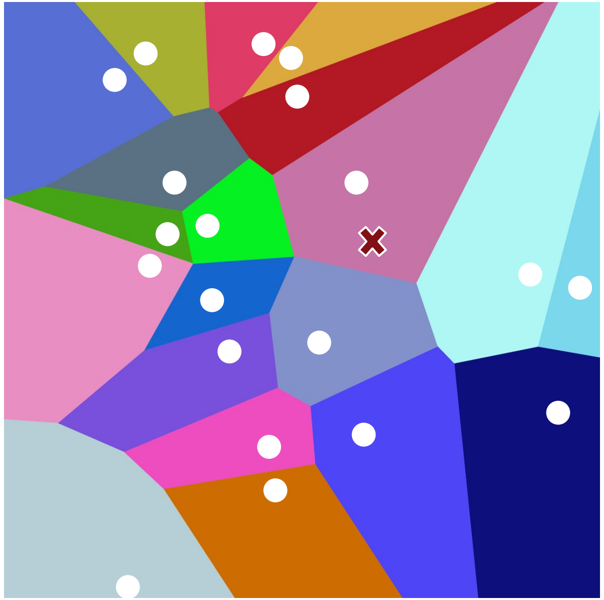




Voronoi Diagram of airports in the US

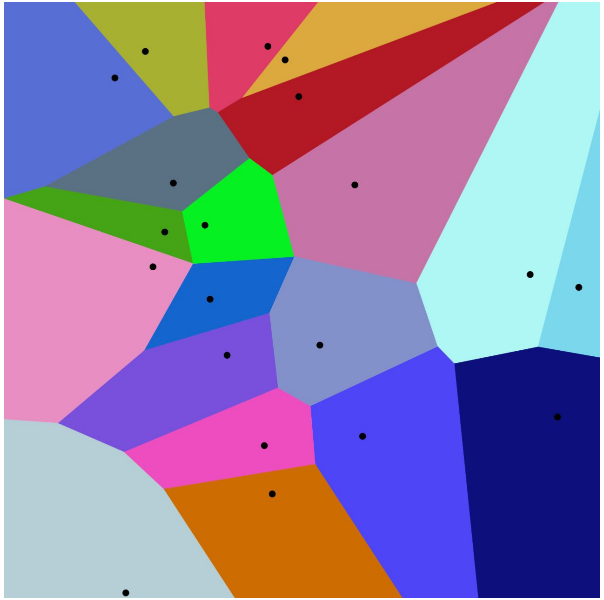
Definition: A **Voronoi diagram** is a partition of a plane into regions close to each of a given set of objects.

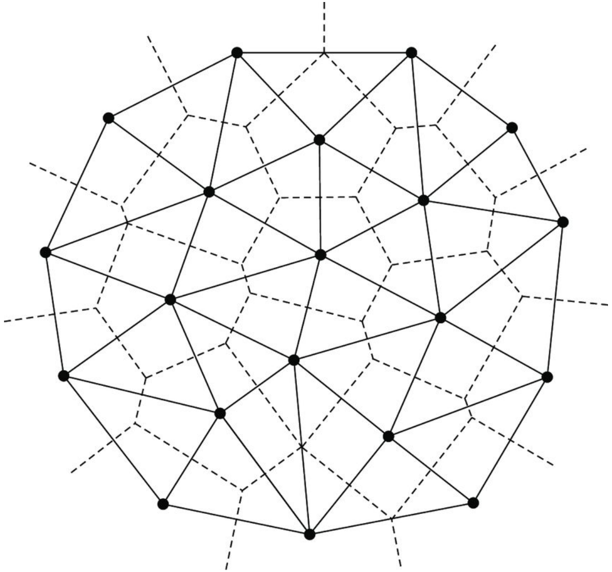
Voronoi Diagrams



Which is the closest POI to **X** ?

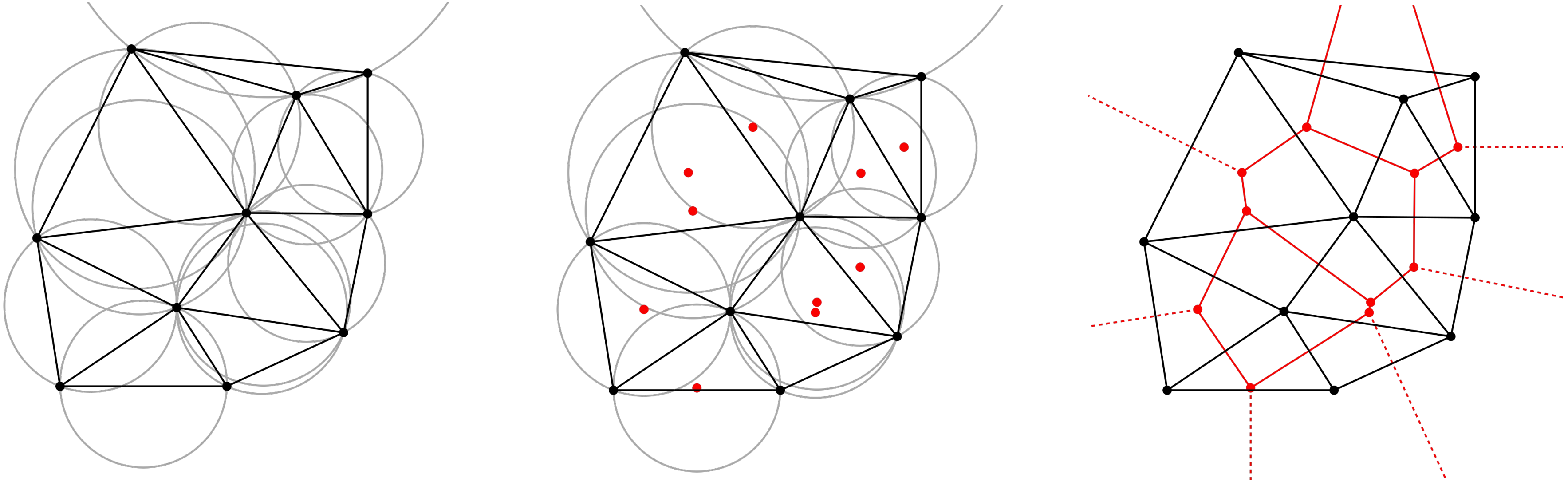
Voronoi Diagrams





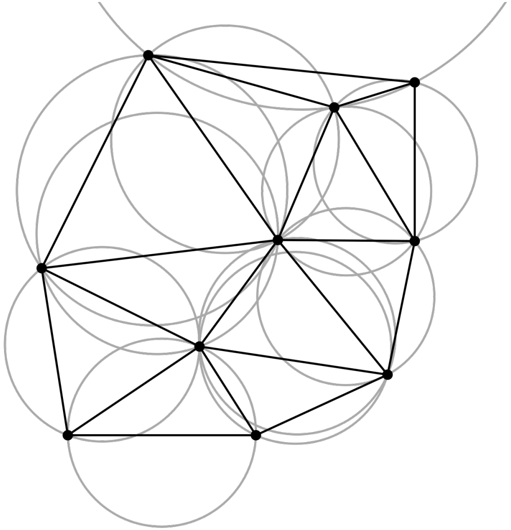
Dual with Delaunay Triangulation

Delaunay Triangulation

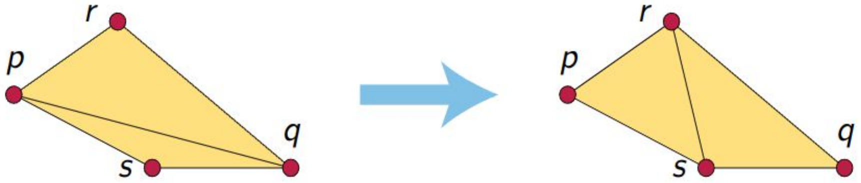


Definition: A **Delaunay triangulation** of a set of points in the plane, subdivides their *convex hull* into *triangles* whose circumcircles do not contain any of the points.

Delaunay Triangulation



- ➔ Start with a triangulation algorithm, e.g. plane sweep triangulation.
- ➔ **Fix** bad triangulations afterwards.



References

- CMU Fall 2022 Lectures on *Fundamentals of Computational Geometry* (https://www.cs.cmu.edu/~15451-f22/lectures/lec21-geometry.pdf)
- Duke Fall 2008 Lectures on Design and Analysis of Algorithms (https://courses.cs.duke.edu/fall08/cps230/Lectures/L-20.pdf)
- UCR CS133 Lectures on Computational Geometry (https://www.cs.ucr.edu/~eldawy/19SCS133/slides/CS133-05-Intersection.pdf)