CS60064 Spring 2022

Computational Geometry

Instructors

Bhargab B. Bhattacharya (BBB)
Partha Bhowmick (PB)

Lecture #13 & Lecture #14
04 February 2022

Indian Institute of Technology Kharagpur
Computer Science and Engineering

Intersection

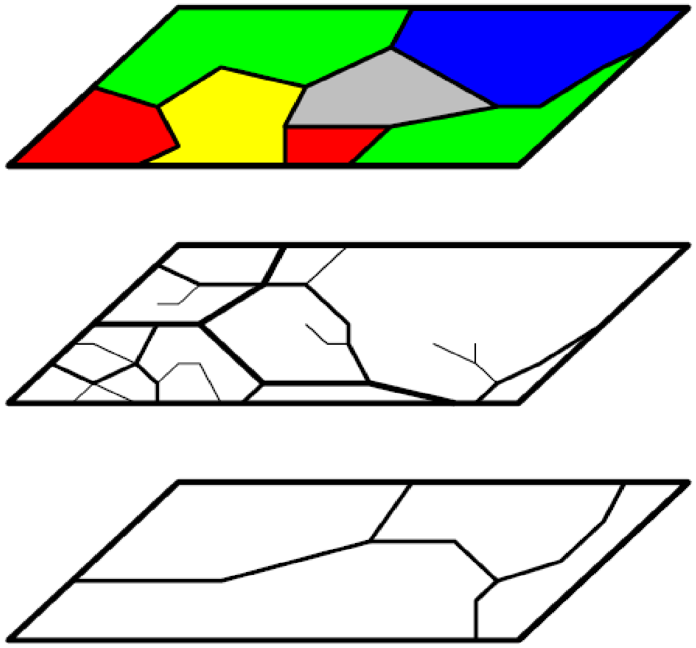
Cloverleaf non-intersecting traffic intersection (1916)

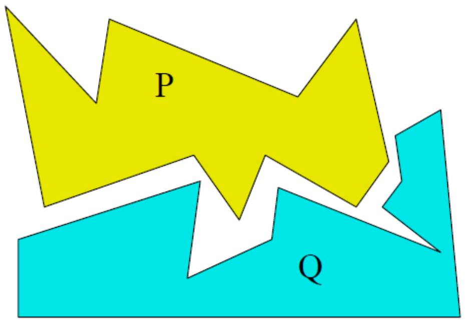
Two roads diverged in a wood, and I -
I took the one less traveled by,
And that has made all the difference.
--- Robert Frost (1916)

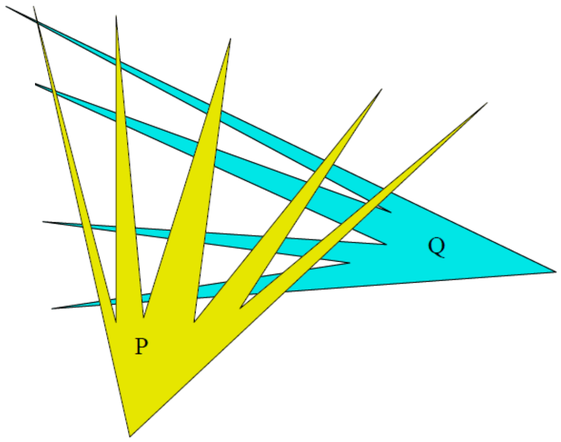
Map layers

In a geographic information system (GIS) data is stored in separate layers

A layer stores the geometric information about some theme, like land cover, road network, municipality boundaries, habitat, . . .







Compute the union and intersection of two simple polygons of *n* and *m* vertices; compute the union/ intersections of a family of rectangles

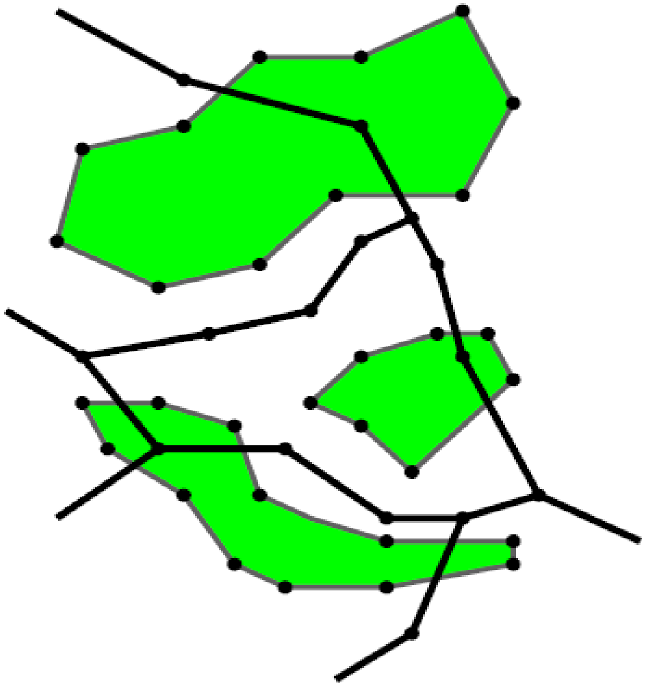
Map Overlay: Intersection Problems

Map overlay is needed to answer questions such as:

1. What is the total length of roads through forests?

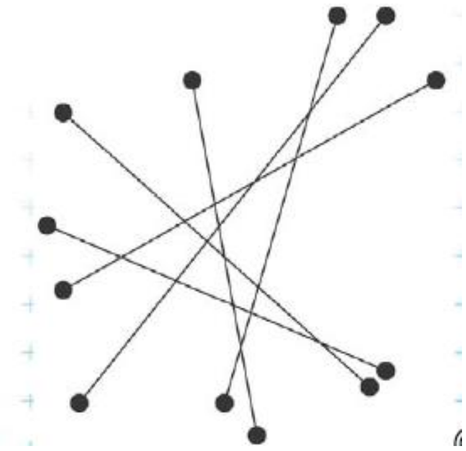
2. What is the total area of corn-fields within one km from a river?

3. What area of all lakes occurs at the geological soil type “rock"?



To solve map-overlay questions, we need information about the intersection points from two sets of line segments (possibly, boundaries of regions)

Line segment intersections

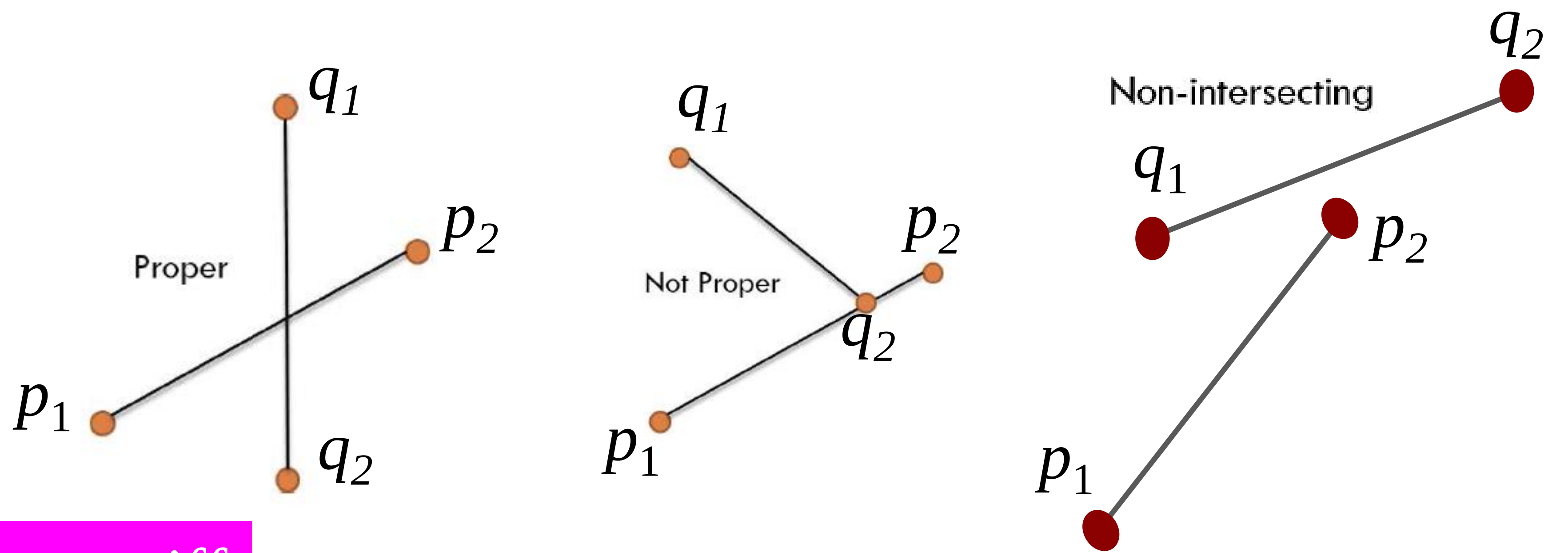


One of the most basic problems in computational geometry

- Solid modeling
  – Intersection of object boundaries
- Overlay of subdivisions, e.g. layers in GIS
  – Bridges on intersections of roads and rivers
  – Maintenance duty (road network crossing state boundaries)
- Robotics
  – Collision detection and collision avoidance
- Computer graphics
  – Rendering via ray shooting (intersection of the ray with objects)

Intersection Test

Determine whether two line segments (p_1, p_2) and (q_1, q_2) intersect properly, i.e., the point of intersection must lie strictly to the interior of both segments



intersect iff

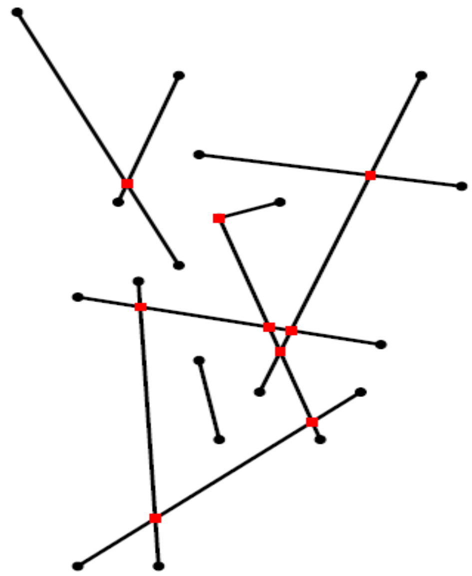
$$\text{Orient}\,(p_1, p_2, q_1) * \text{Orient}\,(p_1, p_2, q_2) < 0$$

and

$$\text{Orient}\,(q_1, q_2, p_1) * \text{Orient}\,(q_1, q_2, p_2) < 0$$

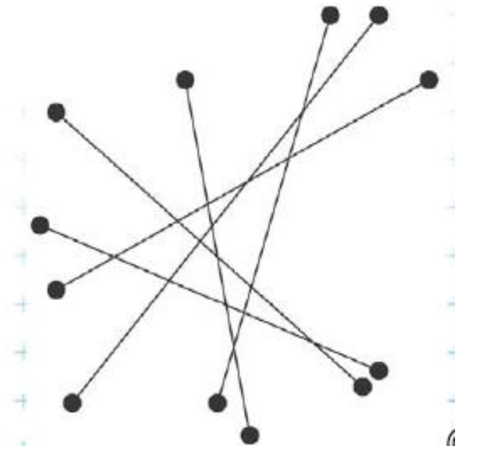
The Easy Problem

Given a set of of *n* line segments in the plane, find all intersection points efficiently



Algorithm Find_Intersections(S)
Input. A set *S* of line segments in the plane.
Output. The set of intersection points among the segments in *S*.
1. for each pair of line segments in *S*
2. If intersected, report their intersection point
Question: Can we improve this $O(n^2)$ bound?

Line segment intersections



Intersection of complex shapes $\Rightarrow$ simpler intersection problems

- Line segment intersection is the most basic intersection algorithm

- Given n line segments in the plane, report all points where a pair of line segments intersect
- Problem complexity

– Worst case # $I = O(n^2)$ intersections

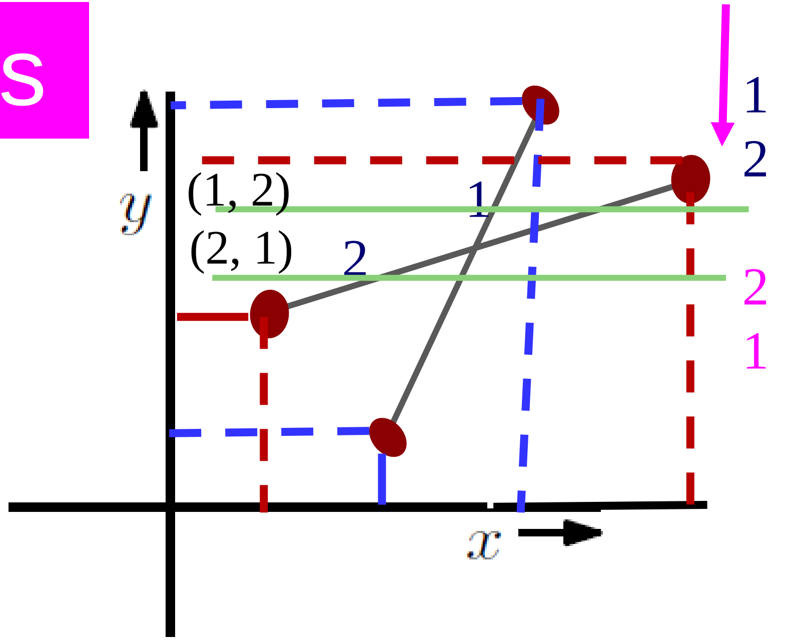
– Practical case – only some intersections

– Can we build an output-sensitive algorithm?

• $O(n \log n + I)$ optimal randomized algorithm

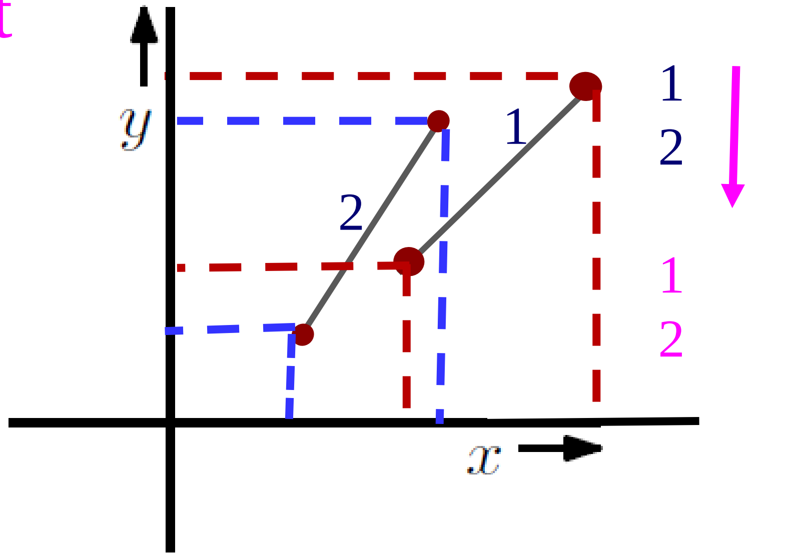
• $O(n \log n + I \log n)$ Bentley-Ottmann sweep-line algorithm, IEEE TC 1979

Line segment intersections



Two line segments can intersect **only if** their *x*- and *y*-spans have an overlap (i.e., necessary condition)

The converse may not be true; hence, the condition is not sufficient



Two major observations:

1. Intersections $\Rightarrow$ swapping of order during plane sweep
2. Intersections $\Rightarrow$ segments will be *x*-adjacent during *y*-sweep before intersection (i.e, they will be horizontal neighbors on the horizontal sweep-line)

Plane sweep

Imagine a horizontal line moving from top to bottom, solving the problem as it goes down; sort the y-coordinates of the vertices to guide the sweep-line

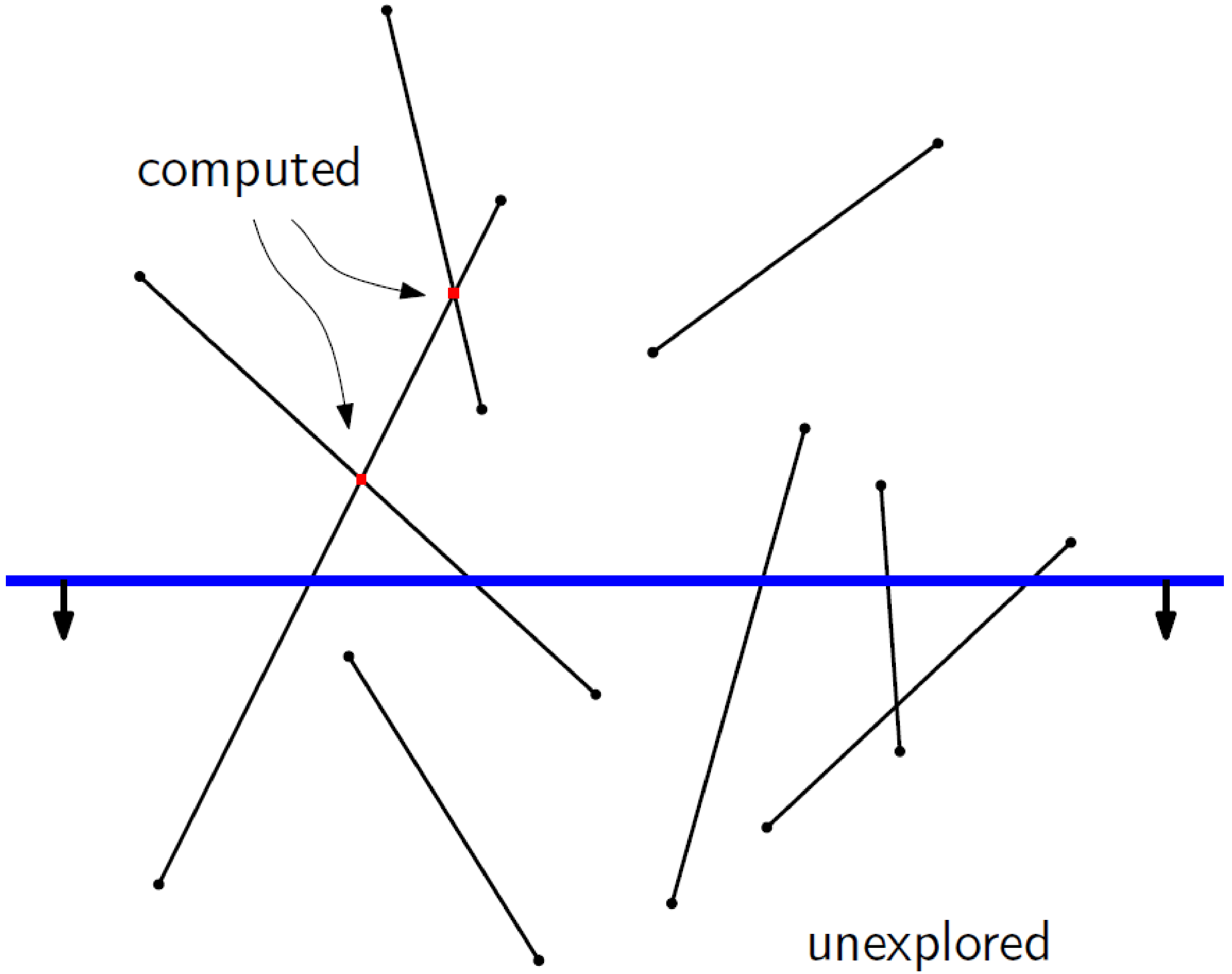
The sweep-line stops and the algorithm computes at certain positions $\Rightarrow$ events in priority queue Q (segment end-points, intersection points)

The algorithm stores the relevant situation at the current position of the sweep line $\Rightarrow$ status (in a tree *Tree*)

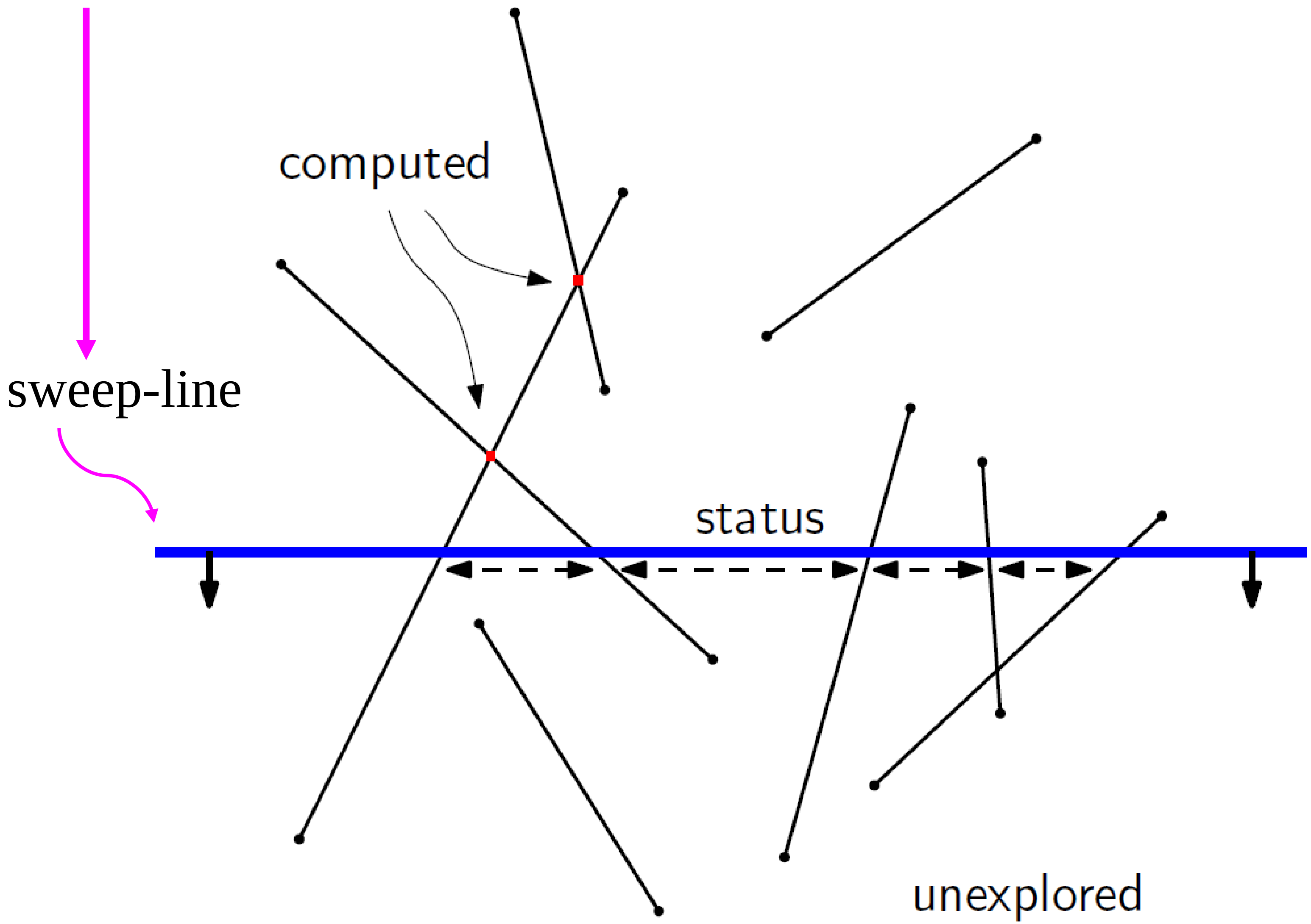
The algorithm knows everything it needs to know above the sweep line, and detects all intersection points (Segment intersections between neighboring segments along sweep-line)

Non-degeneracy assumptions: No line segment is horizontal; no two segments have the same *y*-coordinate; when two segments intersect, they intersect properly in a single point; no three line-segments are concurrent

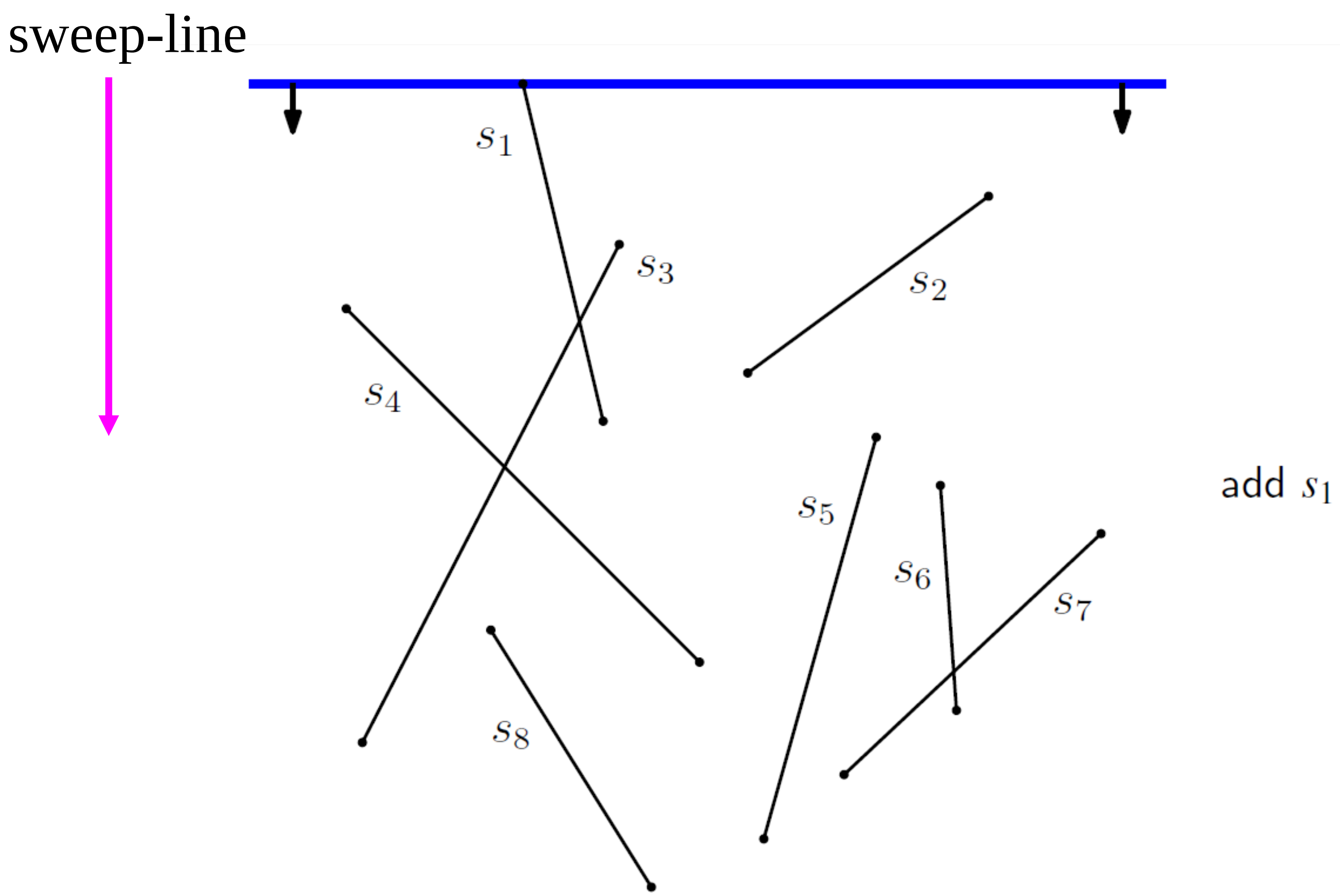
Plane sweep



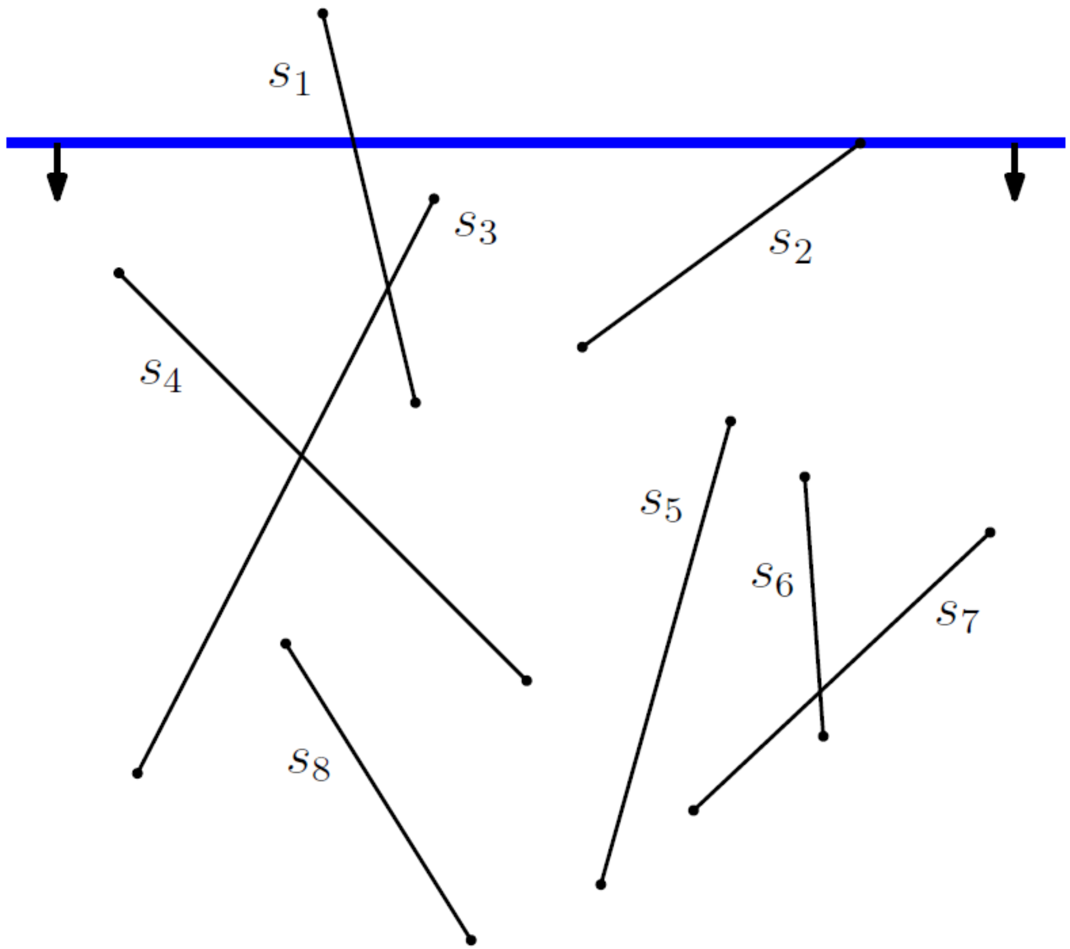
Plane sweep



Events and Sweep-Line Status

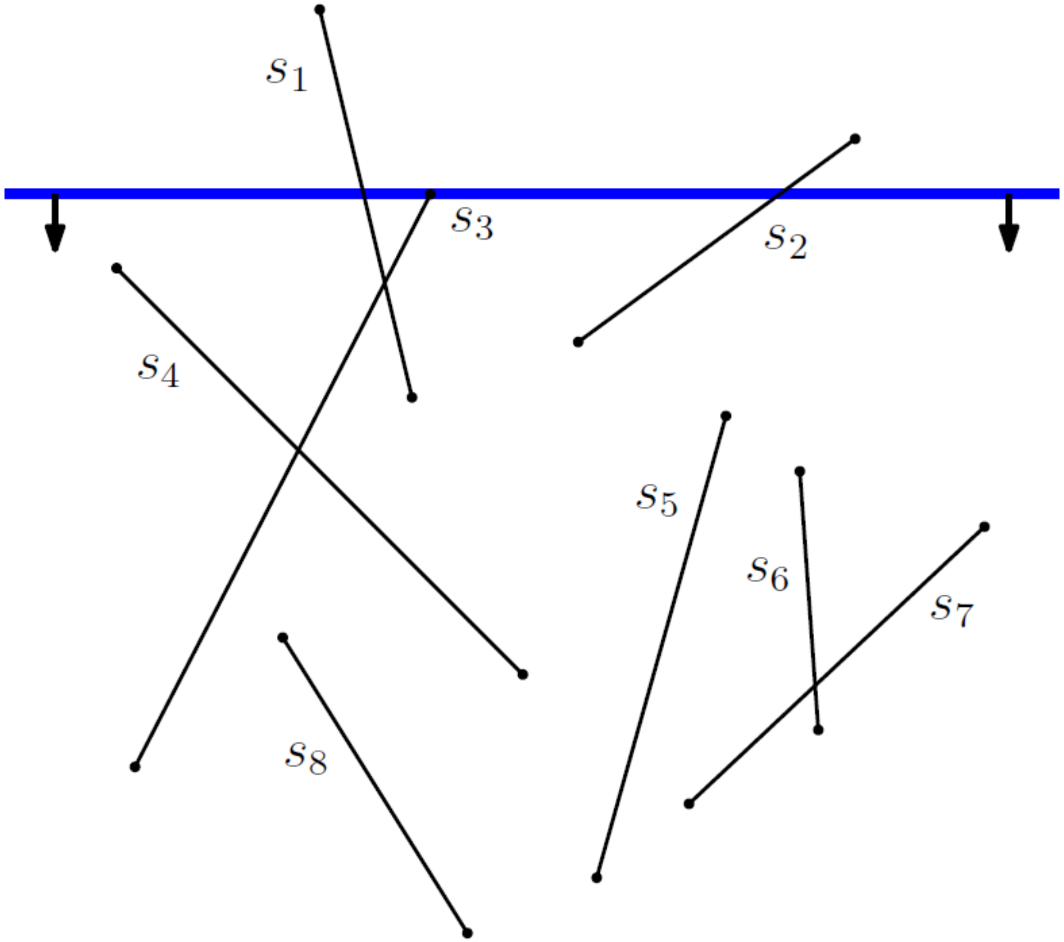


Events and Sweep-Line Status



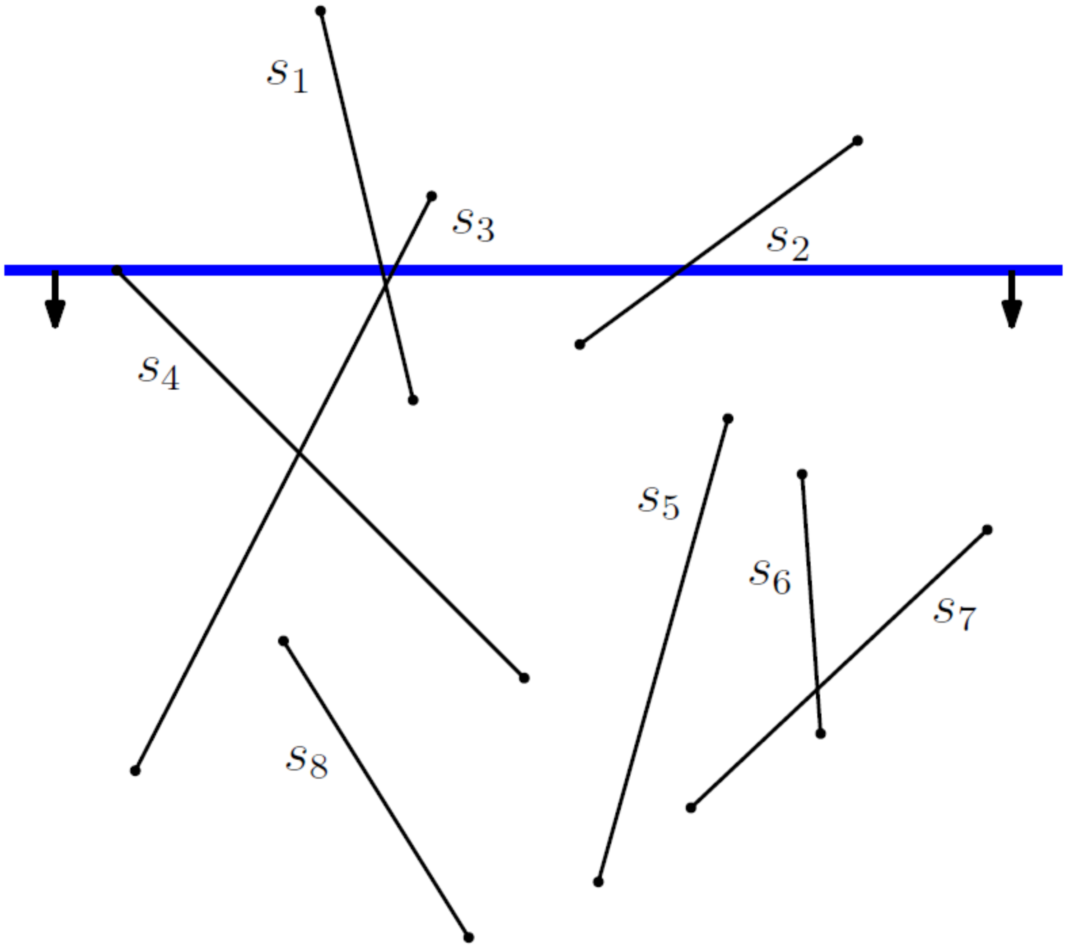
add s_2 after s_1

Events and Sweep-Line Status



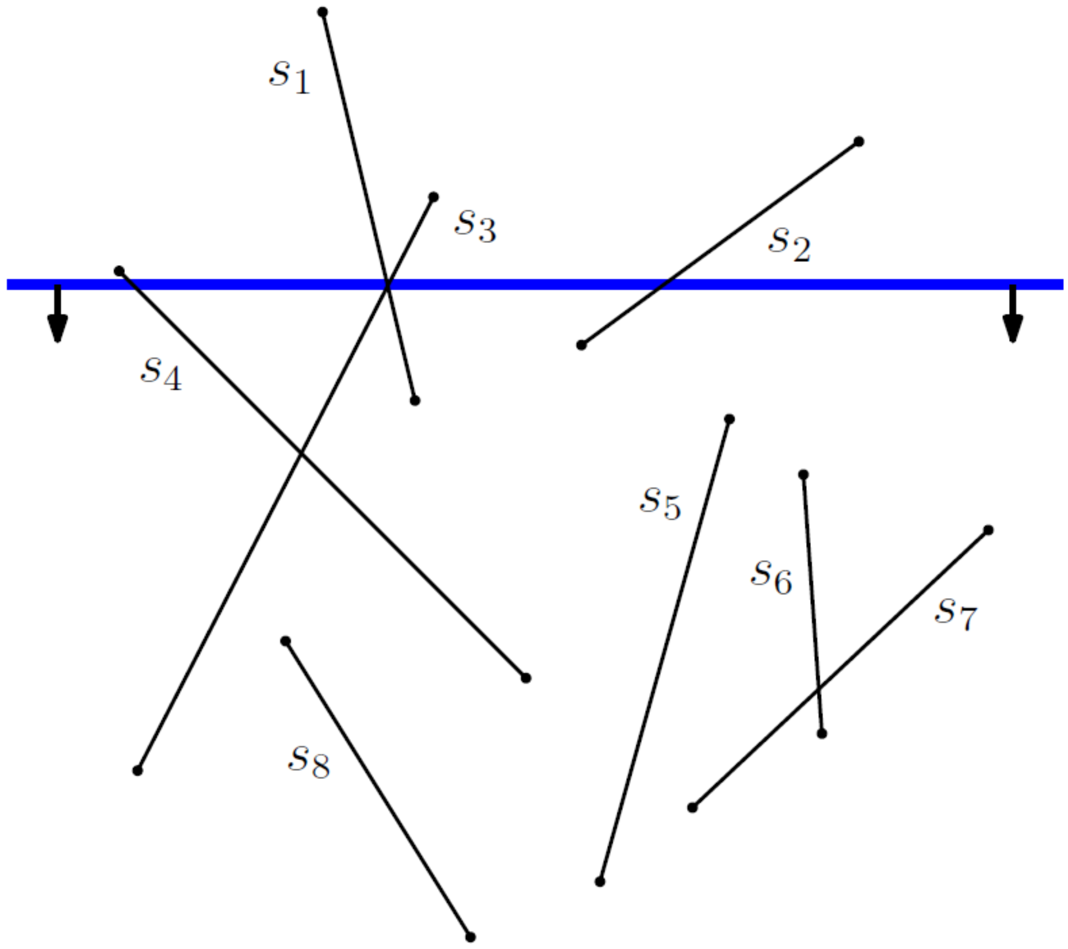
add s_3 between s_1 and s_2

Events and Sweep-Line Status



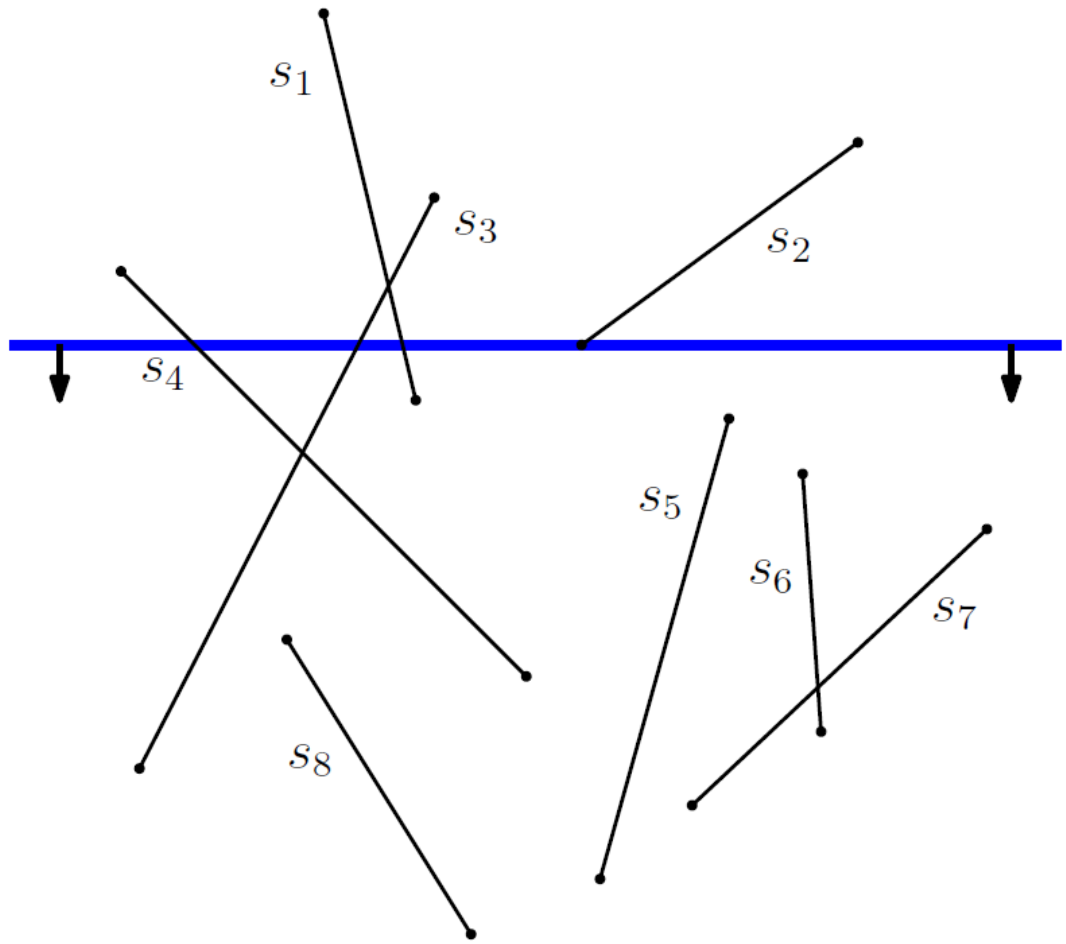
add s_4 before s_1

Events and Sweep-Line Status



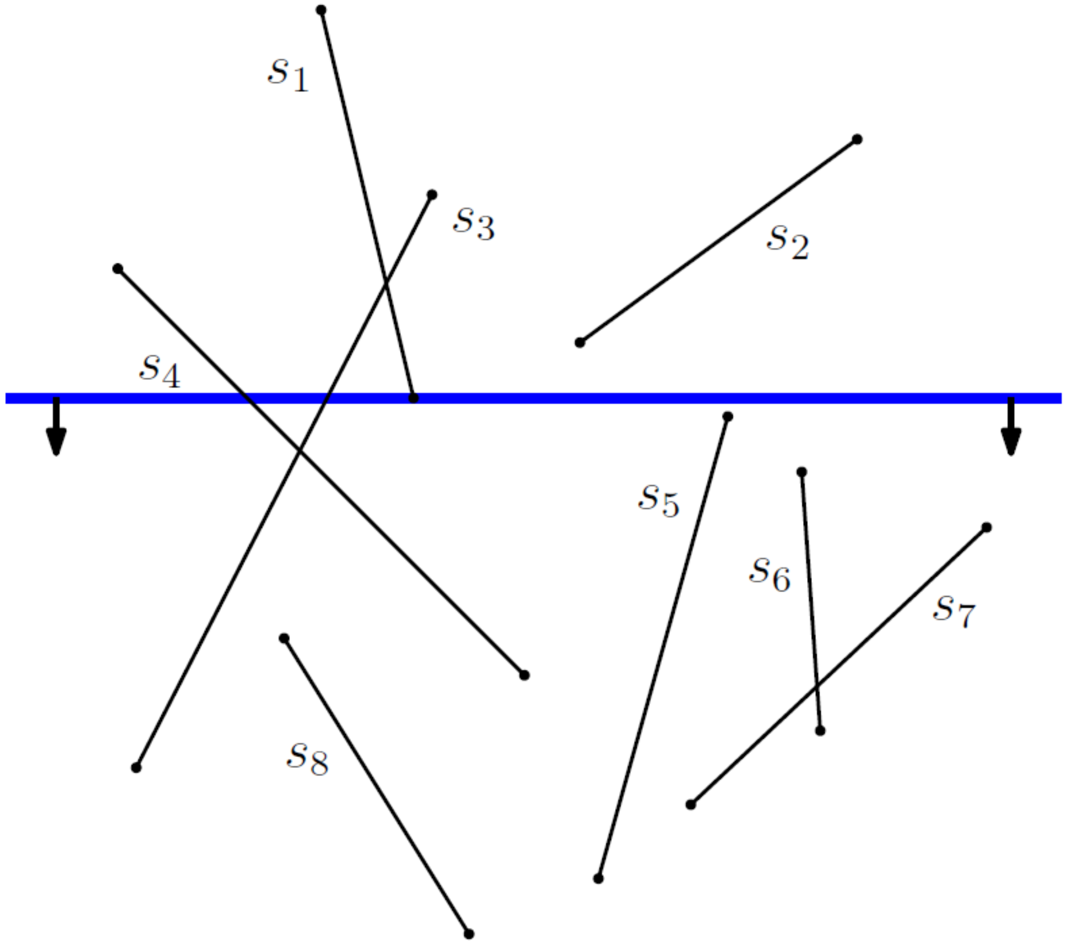
report intersection (s_1, s_3); swap s_1 and s_3

Events and Sweep-Line Status



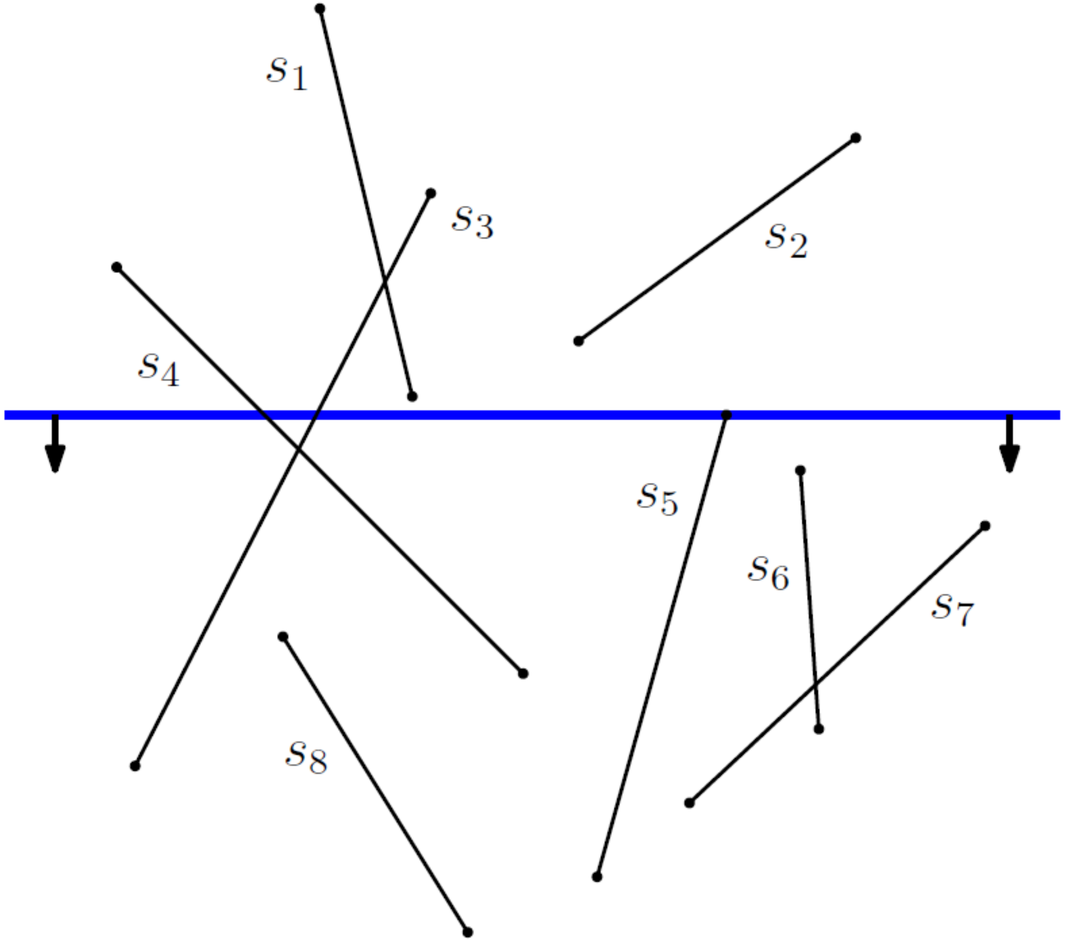
remove s_2

Events and Sweep-Line Status



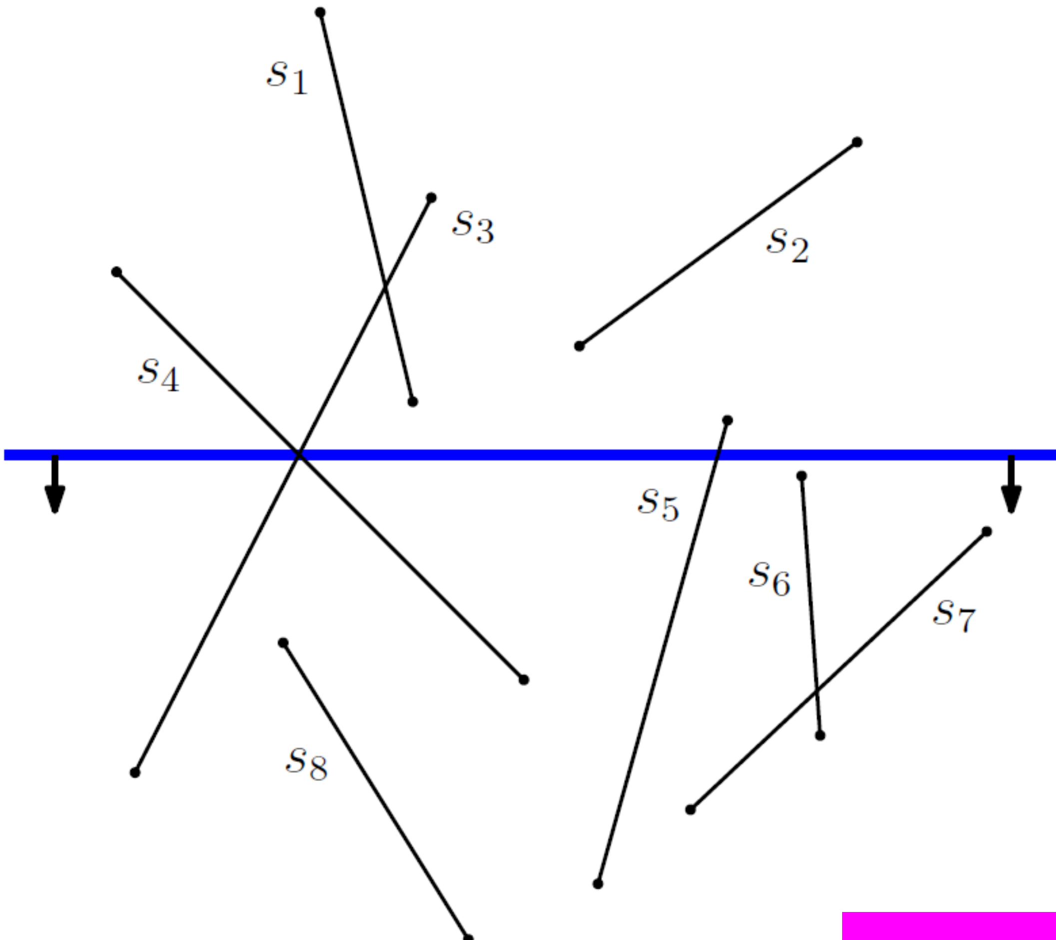
remove s_1

Events and Sweep-Line Status



add s_5 after s_3

Events and Sweep-Line Status



report intersection (s_3, s_4); swap s_3 and s_4

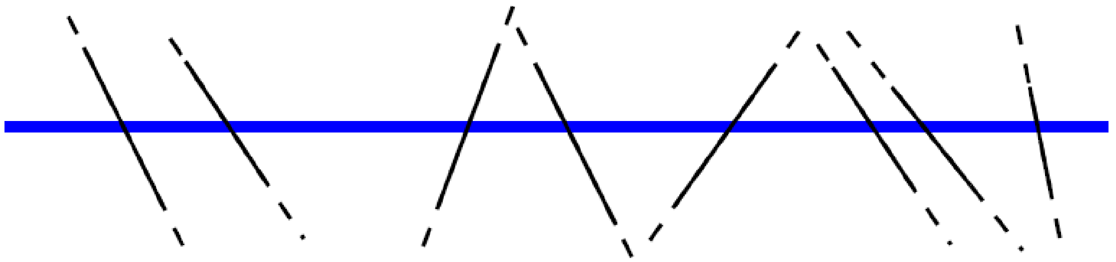
and so on …

Event list and status structure

The event list is an abstract data structure (priority queue Q) that stores all events in the order in which they occur (use a balanced binary tree to implement Q)

The status structure is an abstract data structure (*Tree*) that maintains the current sweep-line status (use a balanced binary tree to implement *Tree* as well)

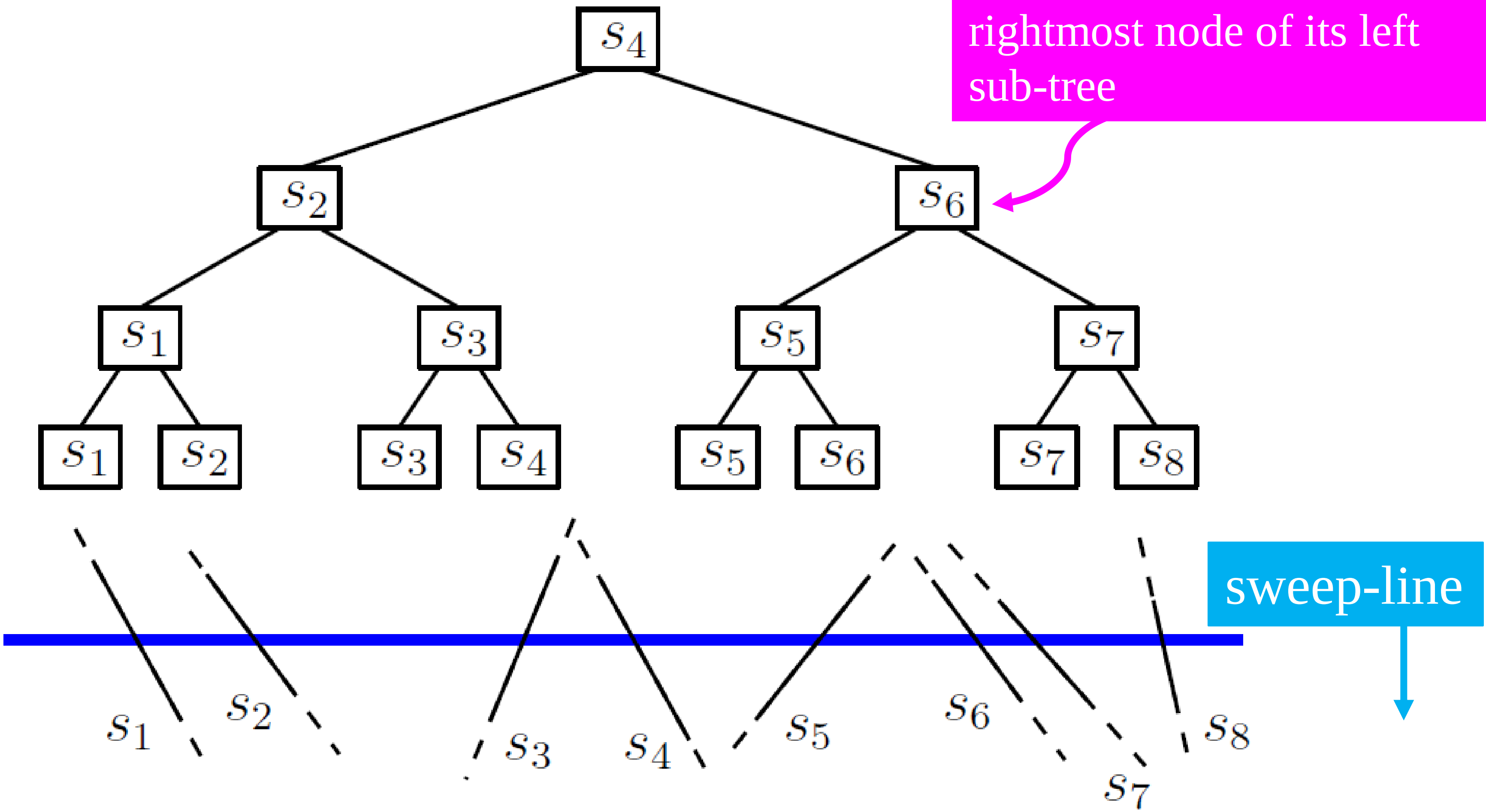
The *status* is the set of currently intersected line segments in the left-to-right order



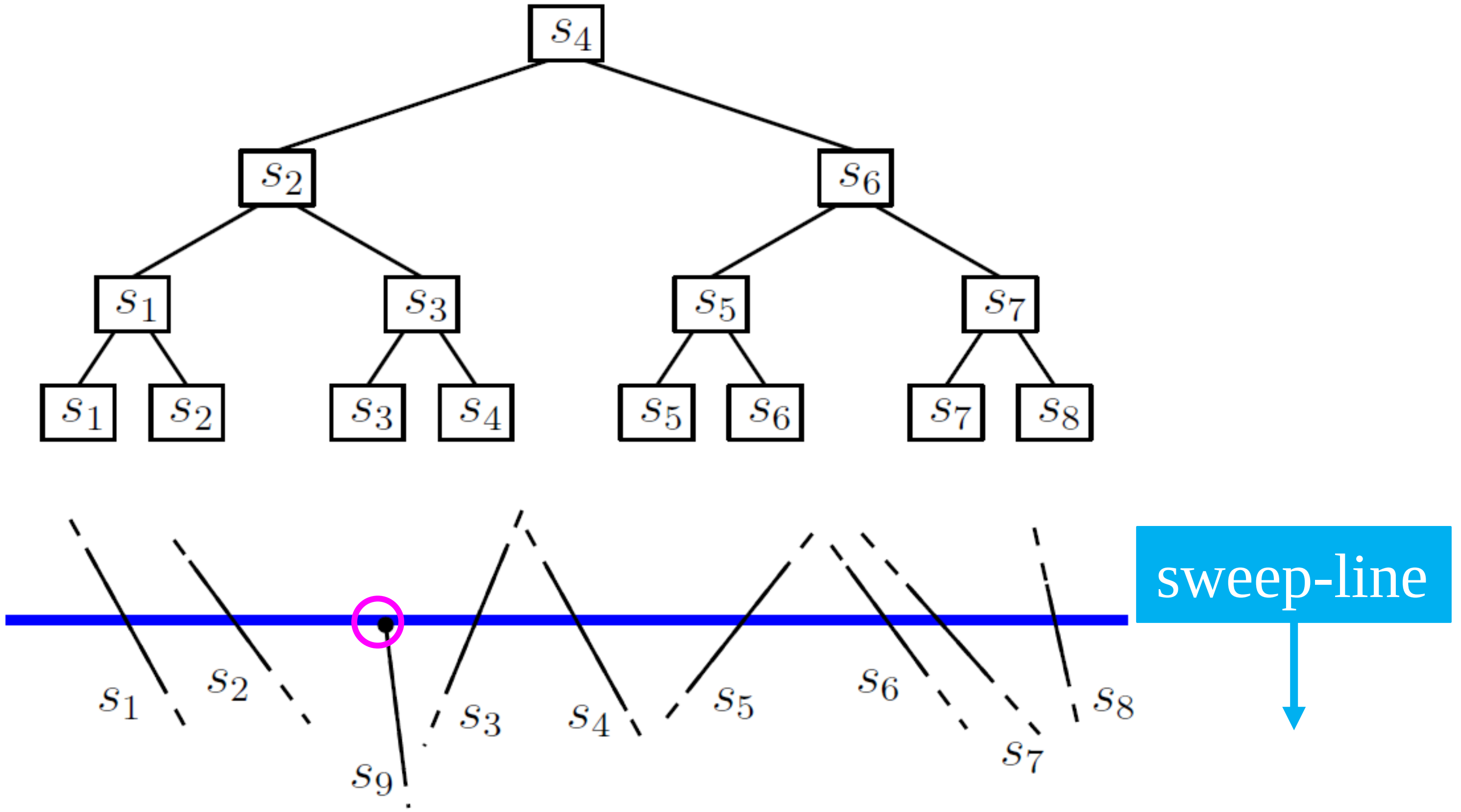
Status structure

We use a balanced binary search tree (*Tree*) with the line segments in the leaves as the status structure

each internal node is the rightmost node of its left sub-tree

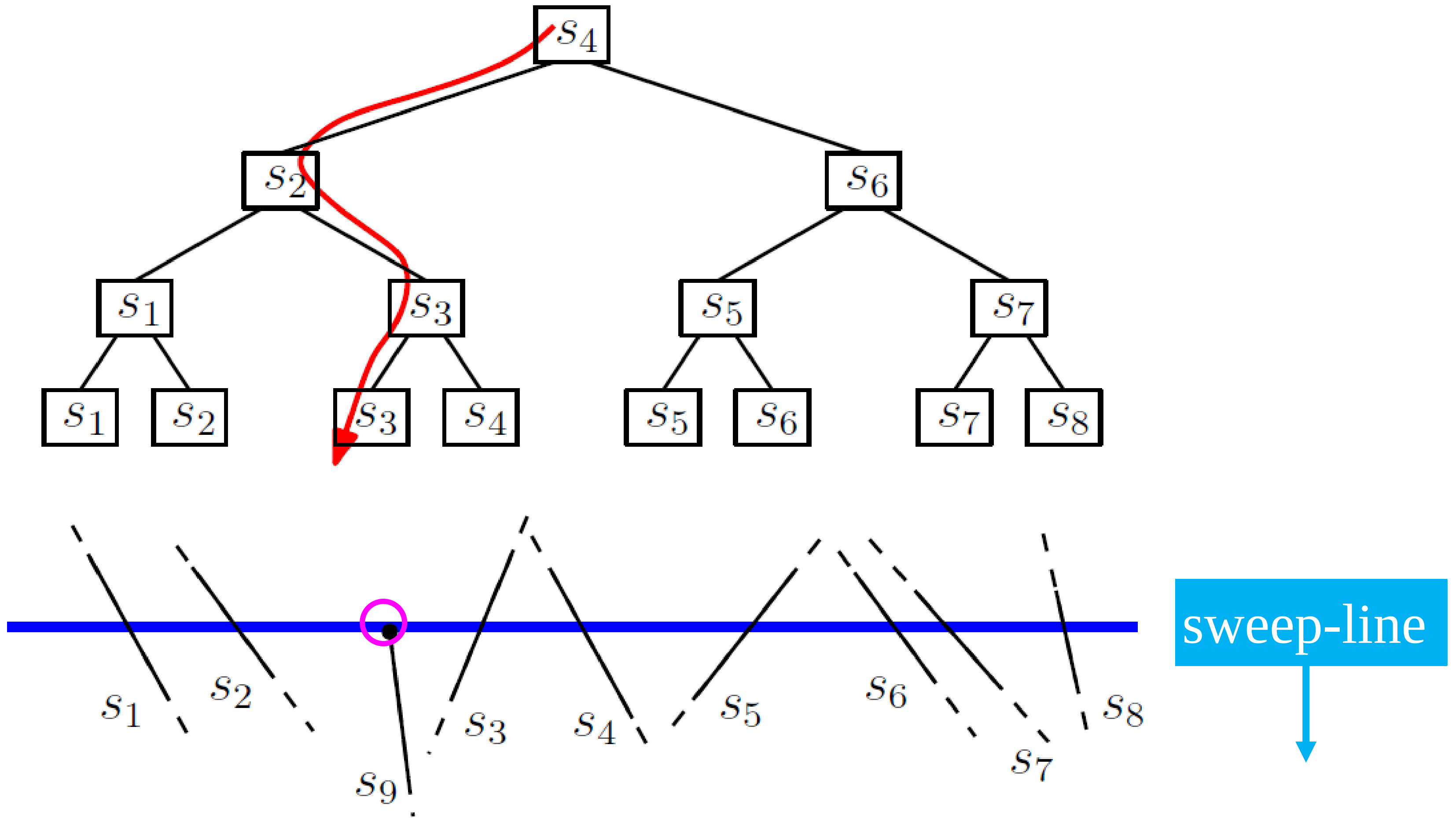


Status structure



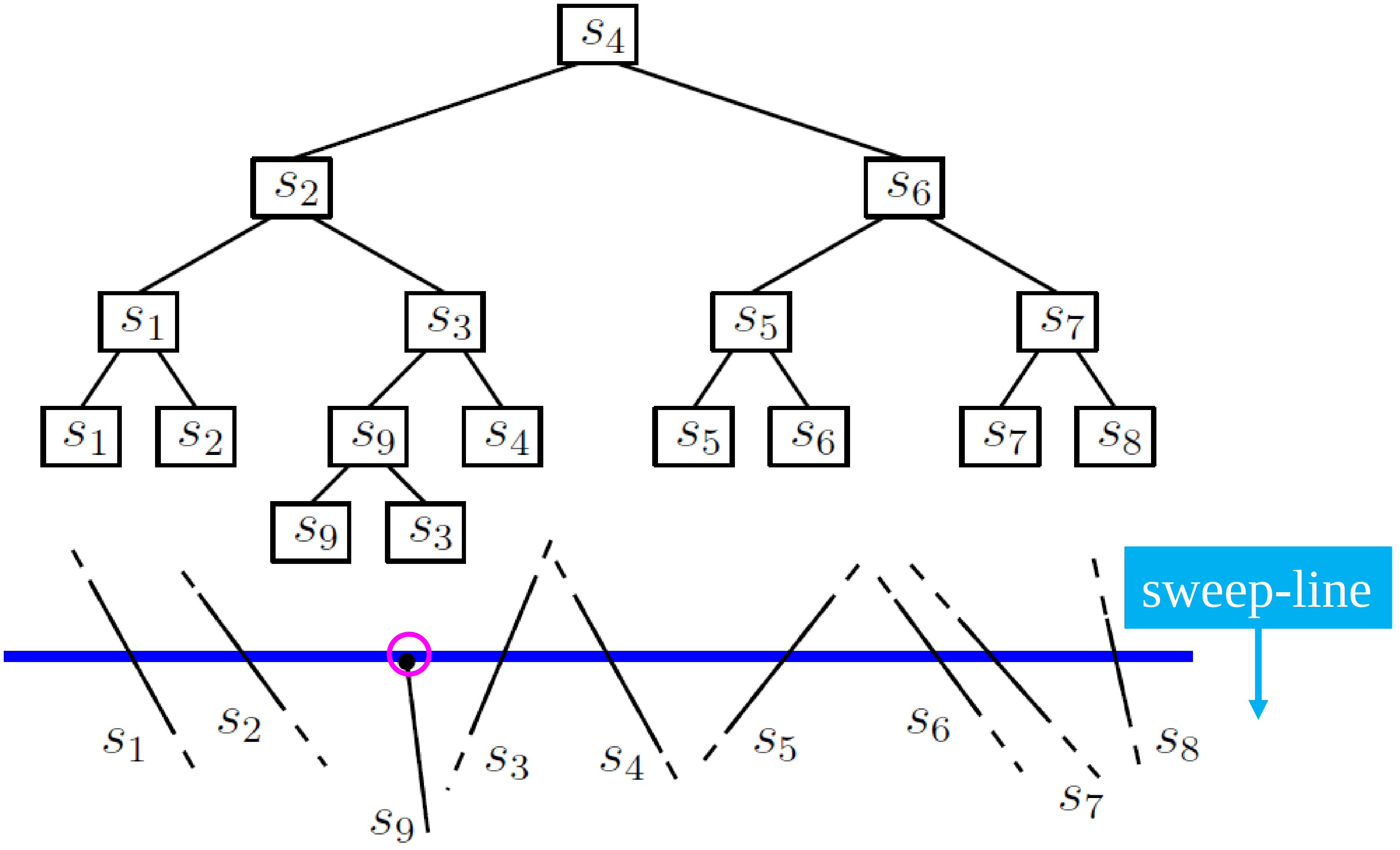
Upper endpoint: search, and insert

Status structure

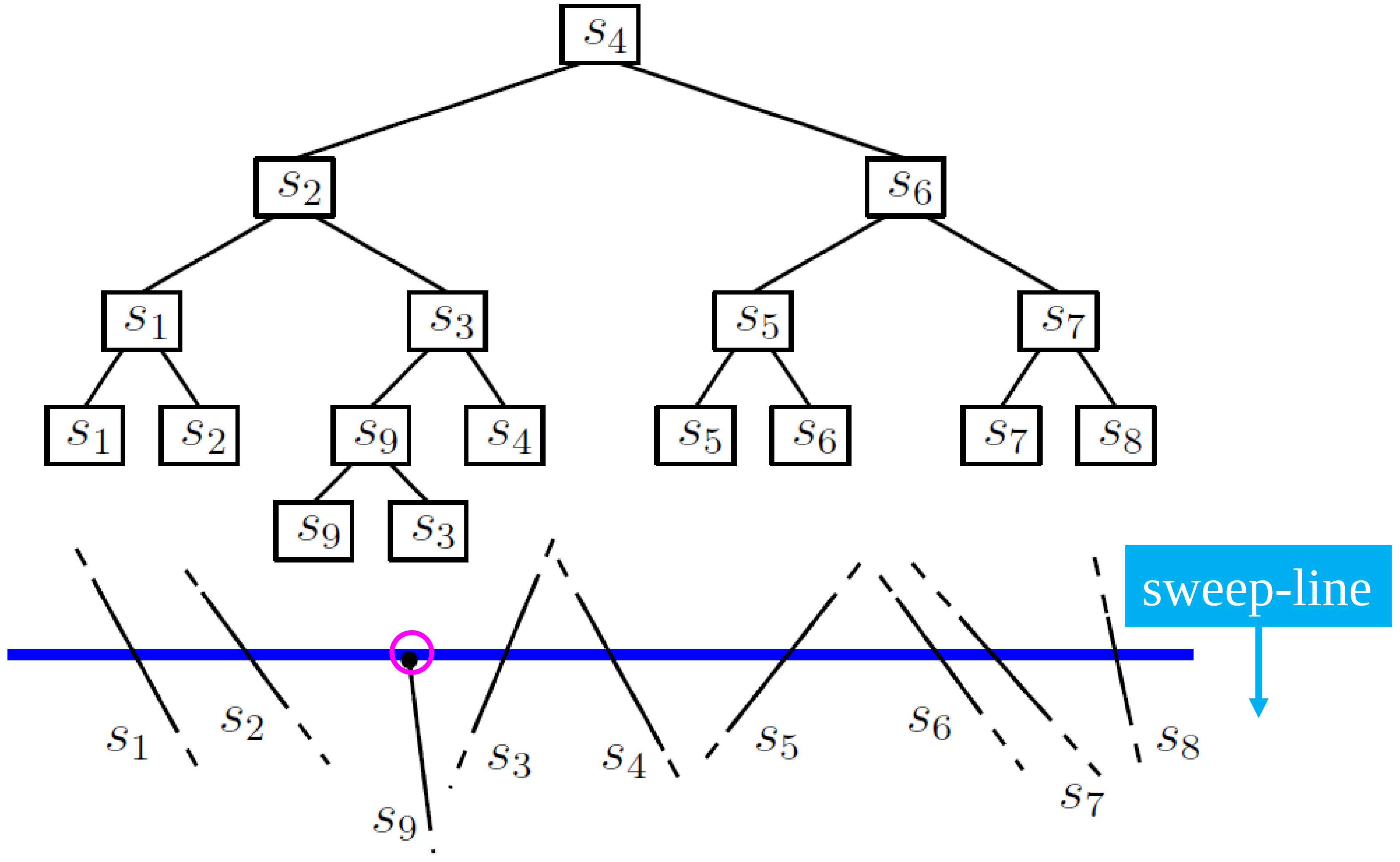


Upper endpoint: search, and insert

Status structure: Insertion of segment beginning

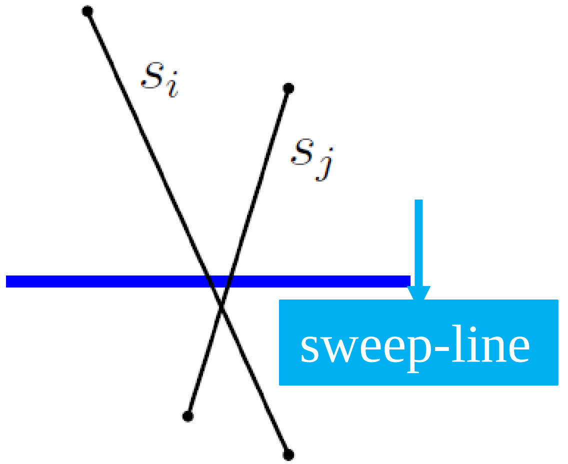


Status structure: Insertion of segment beginning

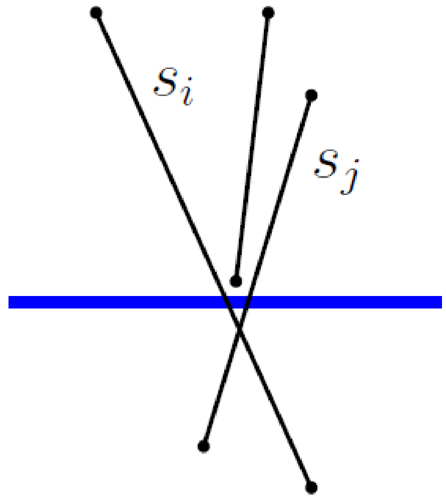


Status structure: Insertion of segment beginning

Lemma: Two line segments s_i and s_j can only intersect (= below) after they have become horizontal neighbors



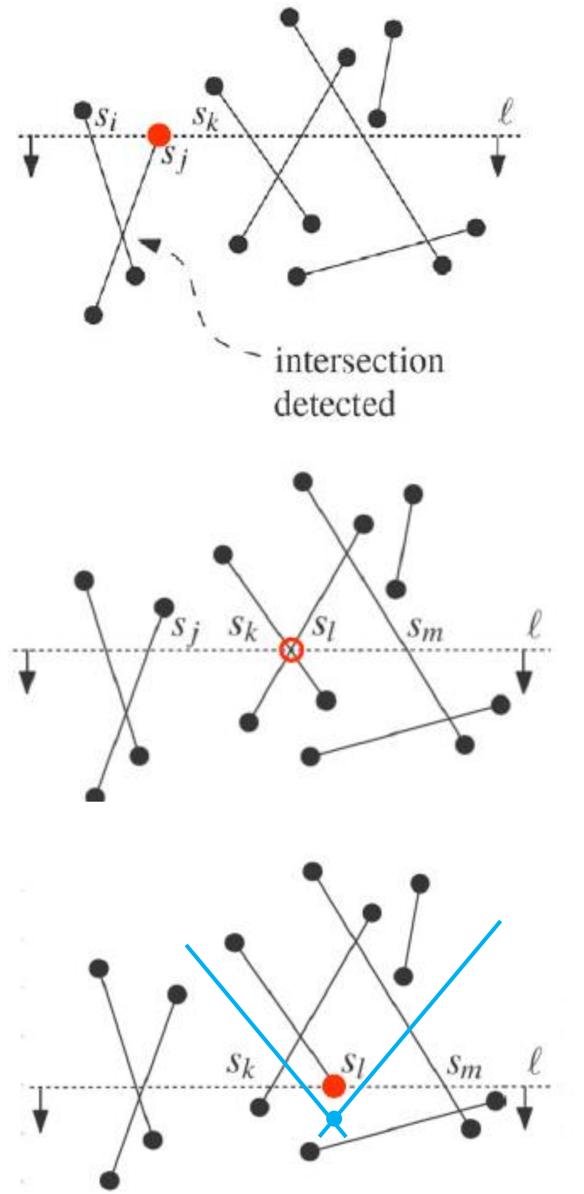
Proof: Just imagine that the sweep line is slightly above the intersection point of s_i and s_j, but below any other event □



Also: some earlier (= higher) event made s_i and s_j horizontally adjacent!

Status Structure: Event Management

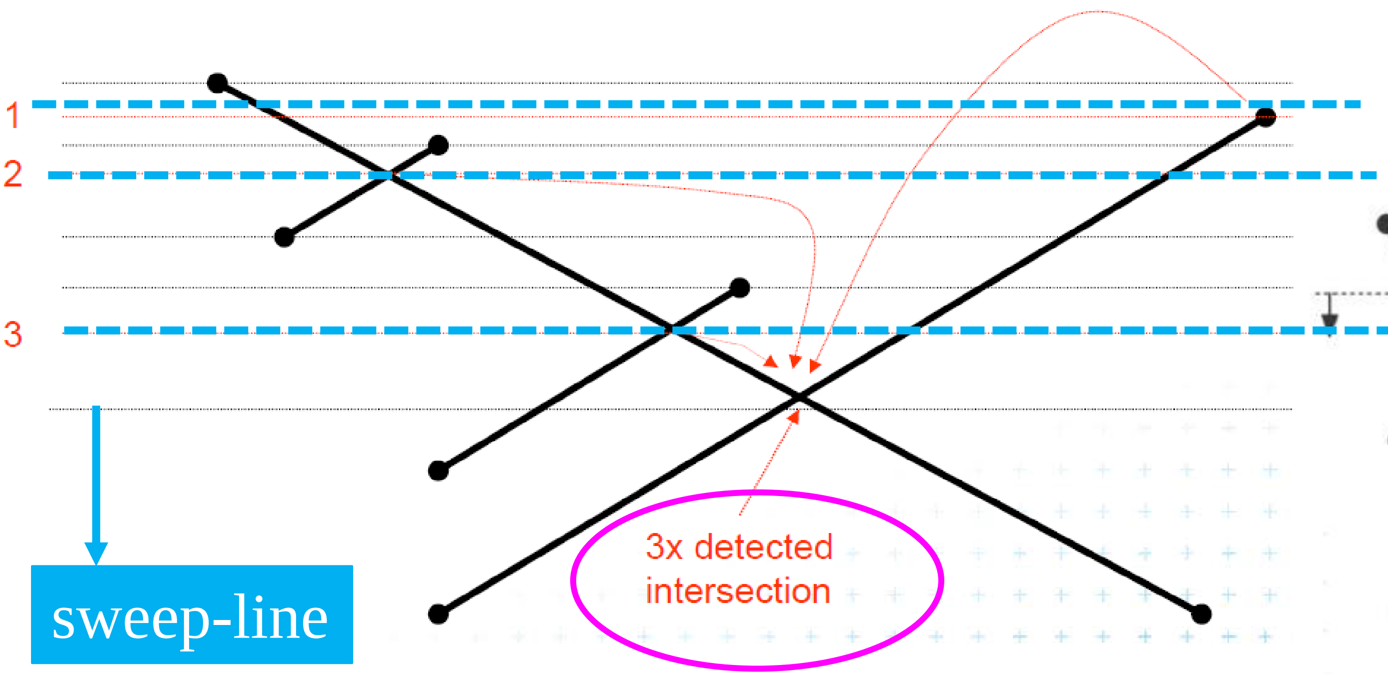
- Upper endpoint *U*(s) of segment *s*
  – insert *s* to *Tree*
  – check two immediate neighbors (left/right) of *s* for possible intersections with *s* and add intersections to *Q*, if any
- Intersection point (s_i, s_j)
  – switch order of segments (s_i, s_j) →(s_j, s_i) in *Tree*
  – for the new left (right) segment, check its intersections with nearest left (right) neighboring segment; insert intersection points, if any, to *Q*
- Lower endpoint *L*(*s*) of segment *s*
  – delete *s* from *Tree*
  – check intersections between immediate left and right neighbors of *s* and update *Q*, when any intersection is detected

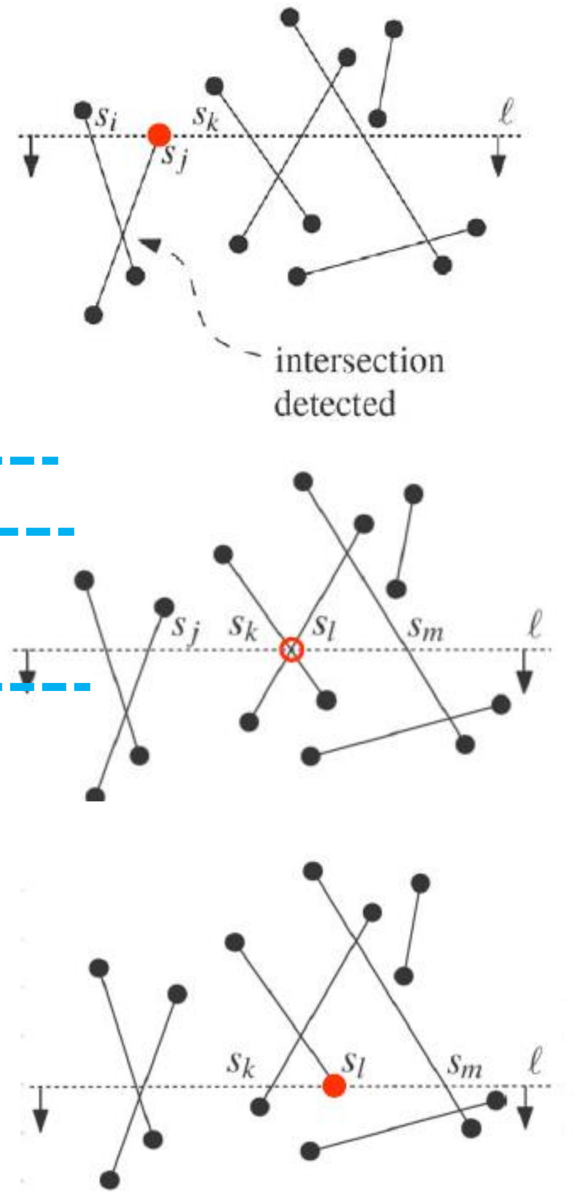


Ref: 4A Textbook

An intersection point may be detected multiple times

Avoid testing of pairs of segments far apart;
Compute intersections of neighbors on the sweep line only



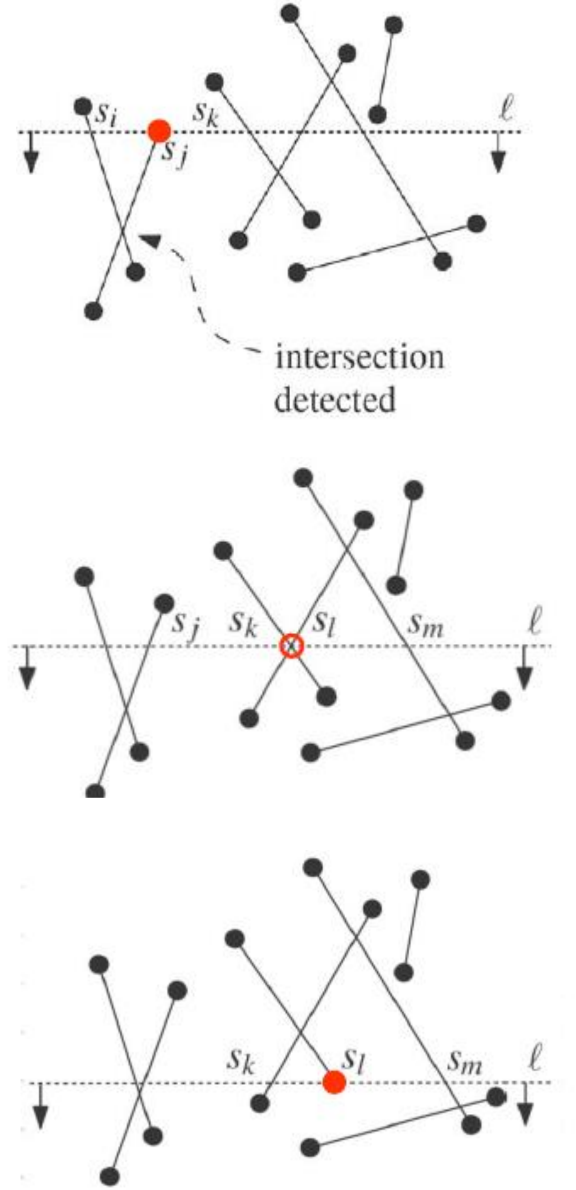


Algorithm: Line-Segment Intersections

Input: A set *S* of line segments in the plane
Output: The set of intersection points + pointers to segments

1. insert the segment end-points in the event queue *Q*;
2. status structure *Tree* $\leftarrow \emptyset$;
3. while *Q* in not empty
4. remove next event *p* from *Q*
5. handleEventPoint(p)
   upper end-point; intersection; lower end-point

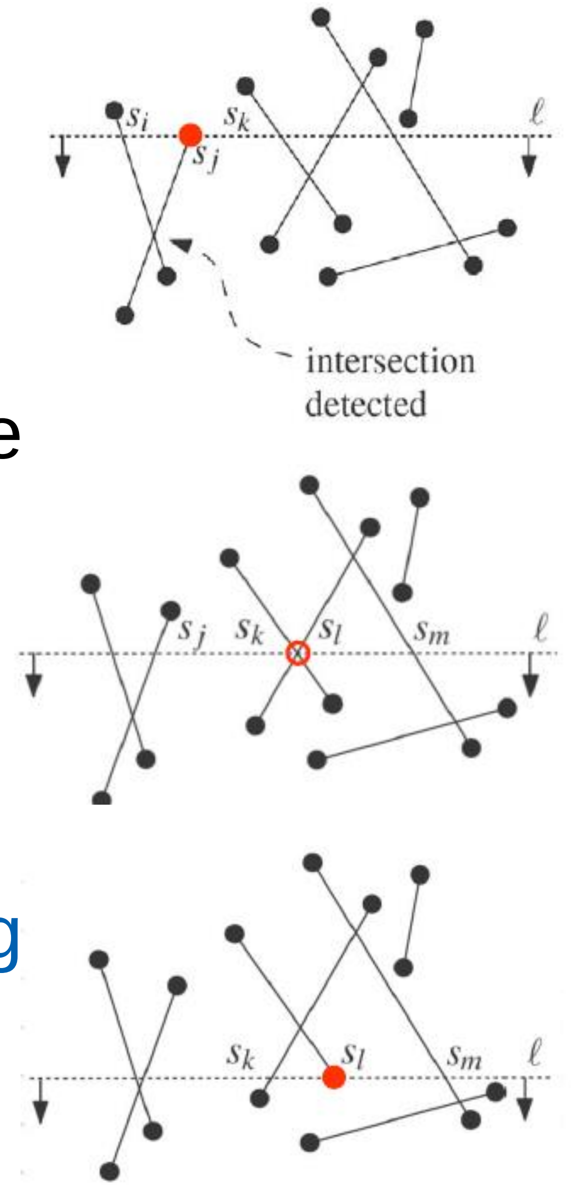
*Upper end-point is used to store the segment-ID in *Q;* both *Q* and *Tree* are implemented as balanced binary search tree



Ref: 4A Textbook

Analysis: Line-Segment Intersection Algorithm

Vertical sorting of $2n$ points: $O(n \log n)$
Sweep-line halts at:
– $2n$ steps for end points;
– I steps for intersections;
– $\log n$ search/update in the status tree



Time Complexity:
$O(n \log n + I \log n)$

Working Space: Tree: $O(n)$;
Queue Q: $O(n + I)$

Size of Q Can be made $O(n)$ by storing intersection points between adjacent segments only;
Output size: $O(I)$

Ref: 4A Textbook

Conclusion

For every sweep algorithm:

Define the status

Choose the status structure and the event list

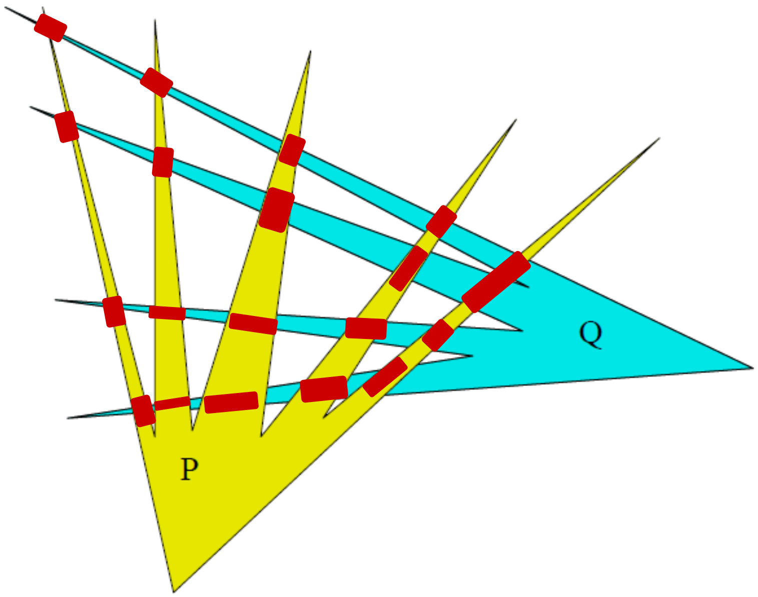
Figure out how events are handled

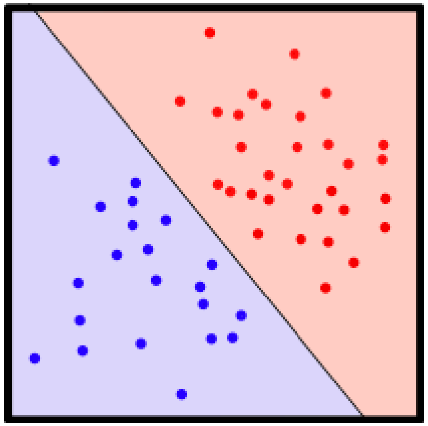
To analyze, determine the number of events and how much time they take

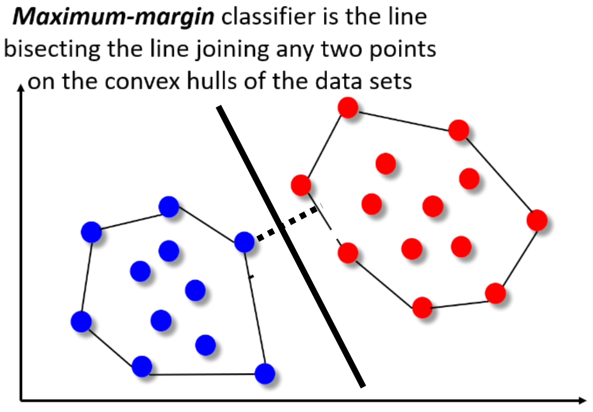
Deal with degeneracies separately

Ref: 4A Textbook

Intersections





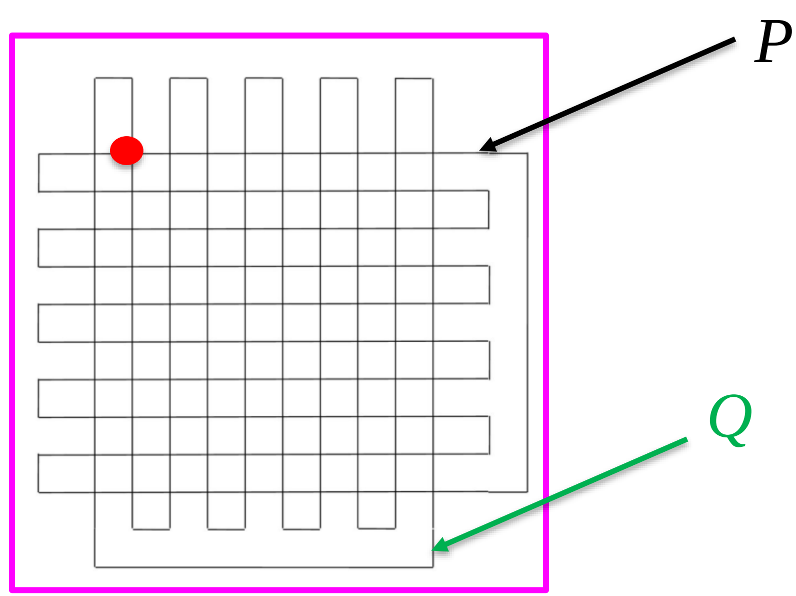


Check if the two convex hulls are disjoint to test linear separability of bichromatic data-sets

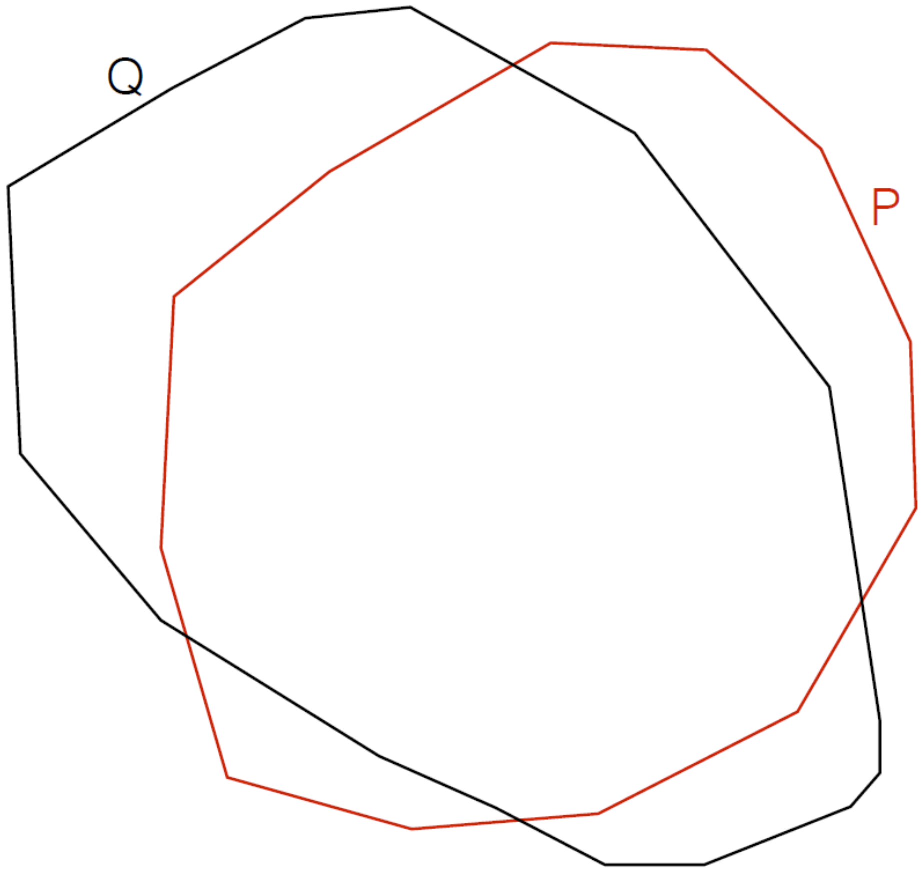
The reason that *Apple* is able to create products like *iPad* is because we have always tried to be at the **intersection** of technology and liberal arts. --- Steve Jobs

Intersection of two polygons

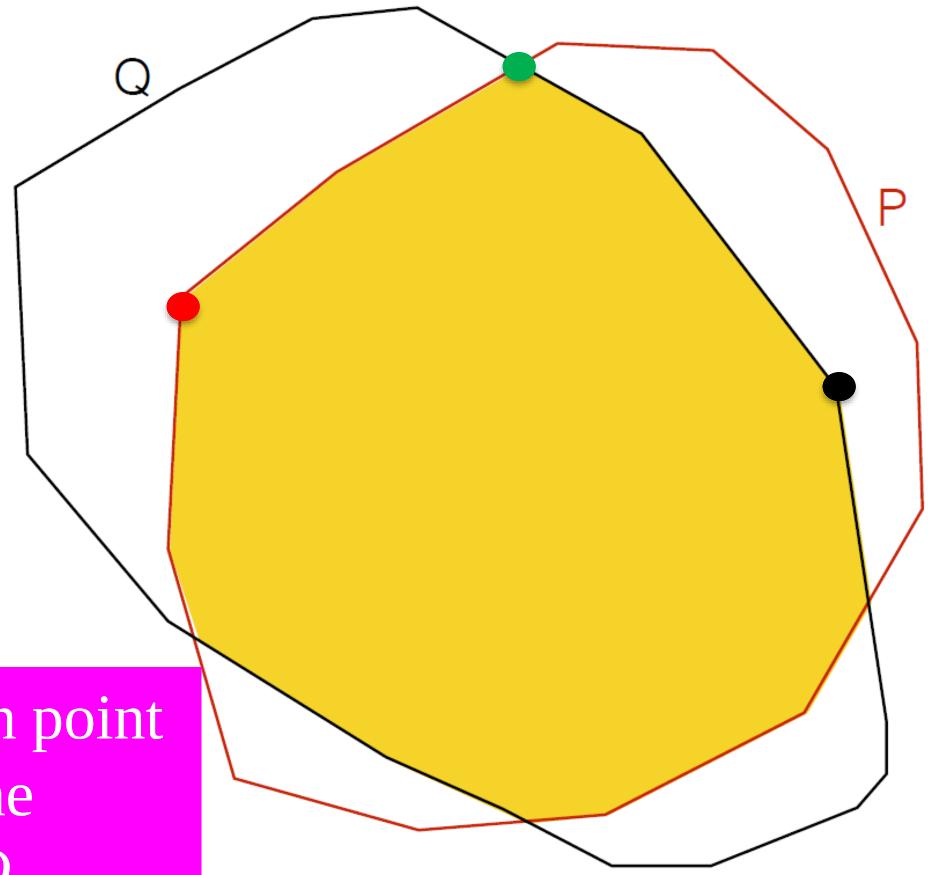
- Determination of intersections may require $O((m + n + I)\log(m + n))$ time, where m and n denote the size of the polygons and I is the number of intersections
- I could be $O(m.n)$
- Naïve time complexity by direct checking each edge-pair: $O(m.n)$



Intersection of two convex polygons



Intersection of two convex polygons

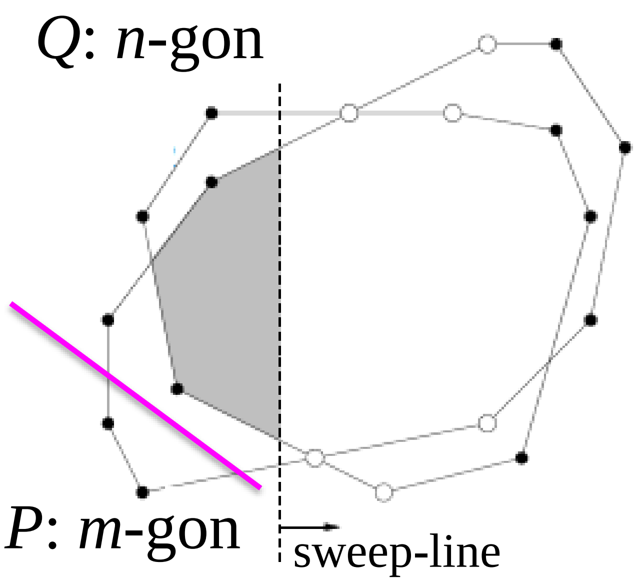


Every intersection point must appear on the boundary of $P \cap Q$

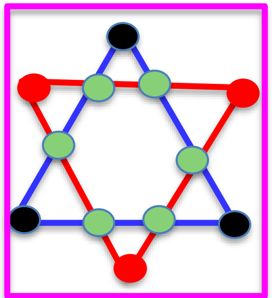
Computing intersection points is not enough; we also need to identify the "intersected" portion (golden region)

Intersection of two convex polygons

Compute $R = P \cap Q$



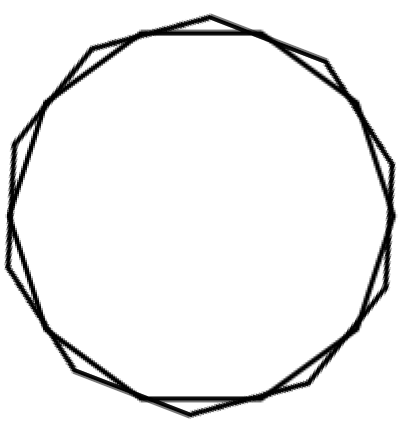
Time Complexity for computing intersections of n line segments: $O(n \log n + I \log n)$; I: # intersec.



Each edge of a convex polygon can intersect with at most two edges of the other polygon
$\Rightarrow$ # total intersections = $O(m + n)$

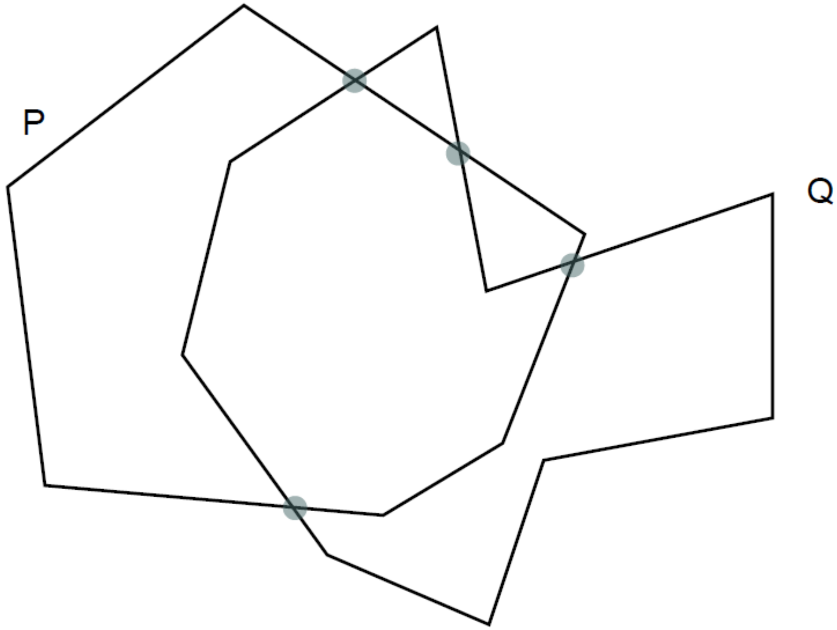
vertices in R is at most $(m + n)$

Size of the sweep-line status is *at most* 4
Size of the event queue is $O(1)$; *at most* 8
$\Rightarrow$ Time Complexity for computing intersecting points of two convex polygons: $O(m + n)$



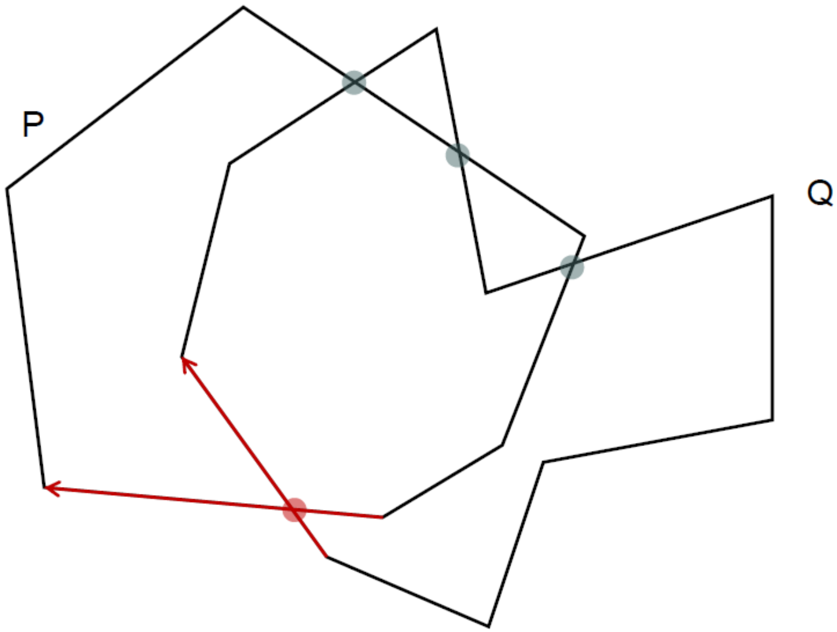
Ref: David Mount's Lecture Notes

Intersection of two simple polygons



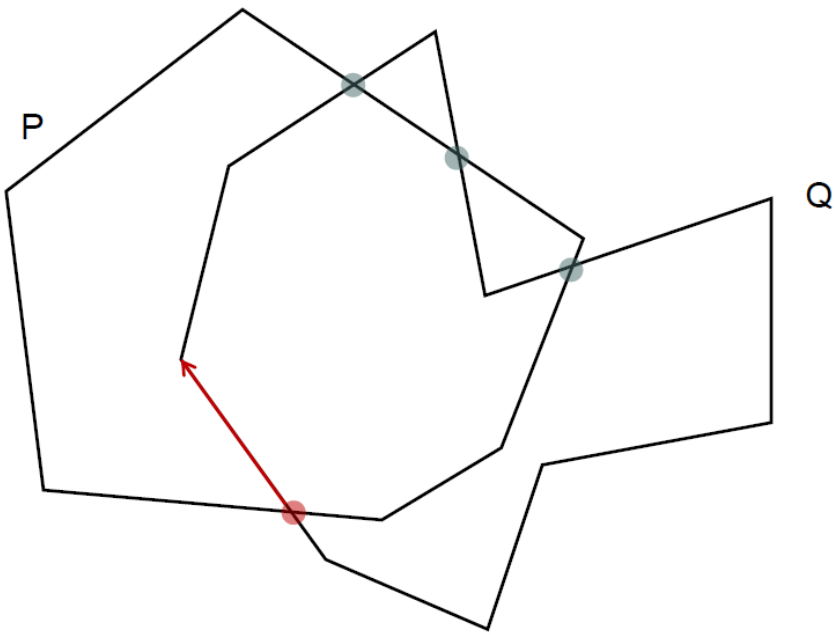
- **Assume:** intersected portion is connected (always true for convex polygons)
- Start from an intersection point
- Always take the rightmost move while traversing boundaries CW

Intersection of two simple polygons



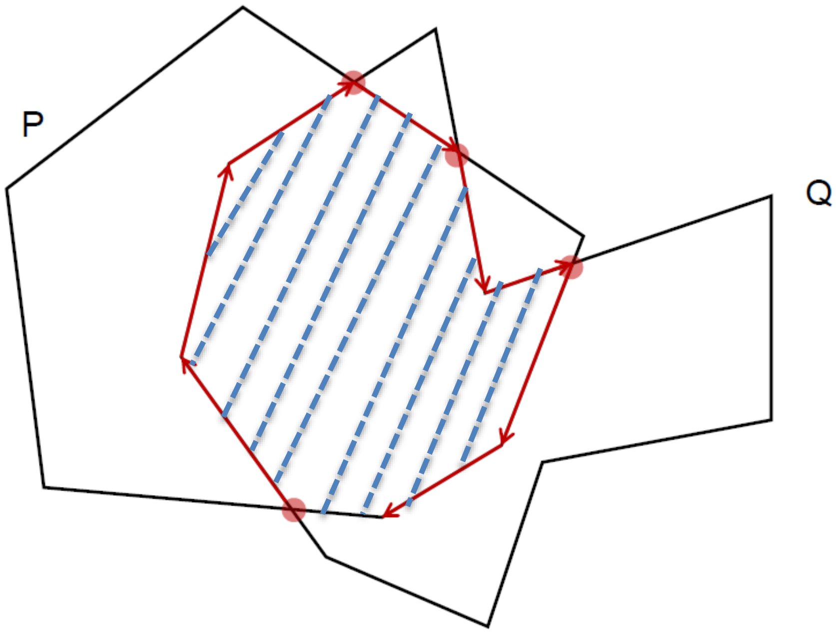
- Assume: intersected portion is connected (always true for convex polygons)
- Start from an intersection point
- Always take the rightmost move while traversing boundaries CW

Intersection of two simple polygons



- Assume: intersected portion is connected (always true for convex polygons)
- Start from an intersection point
- Always take the rightmost move while traversing boundaries CW

Intersection of two simple polygons

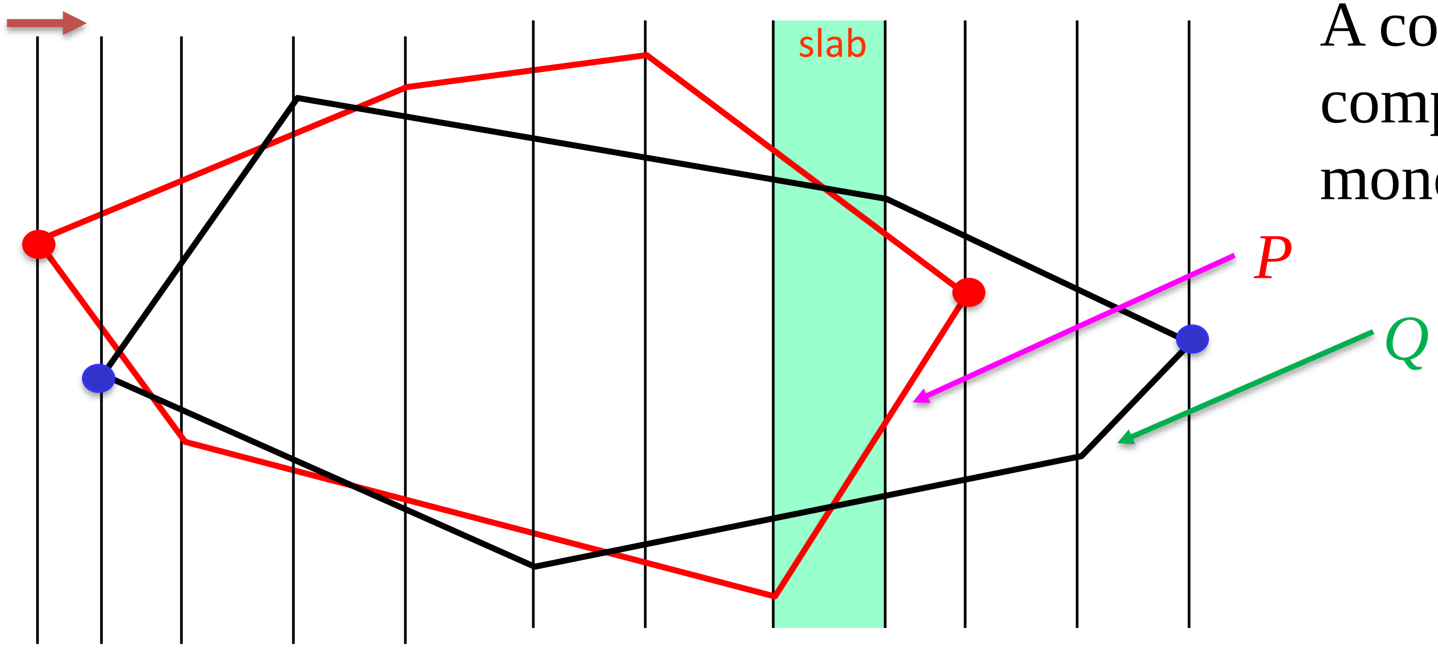


- Assume: intersected portion is connected (always true for convex polygons)
- Start from an intersection point
- Always take the rightmost move while traversing boundaries CW

Intersection of two Convex Polygons: Second Method

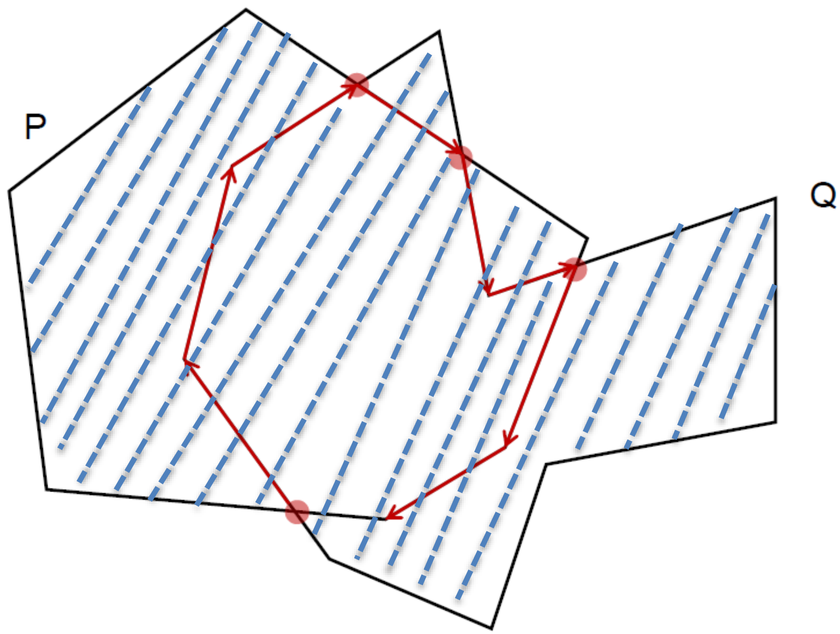
$P \cap Q$ can be computed in $O(m + n)$ time

The upper and lower chains of P and Q are x-monotone; In $O(m+n)$ time, merge four sorted vertex-lists to form an x-sorted order of $m+n$ vertices; Sweep a vertical line from L $\rightarrow$ R, thus partitioning the plane into $m+n$-1 vertical slabs; The intersection of each slab with P or Q is a trapezoid; The two trapezoids within a slab can be intersected in $O(1)$ time; Hence, we can obtain $P \cap Q$ in $O(m+n)$ time



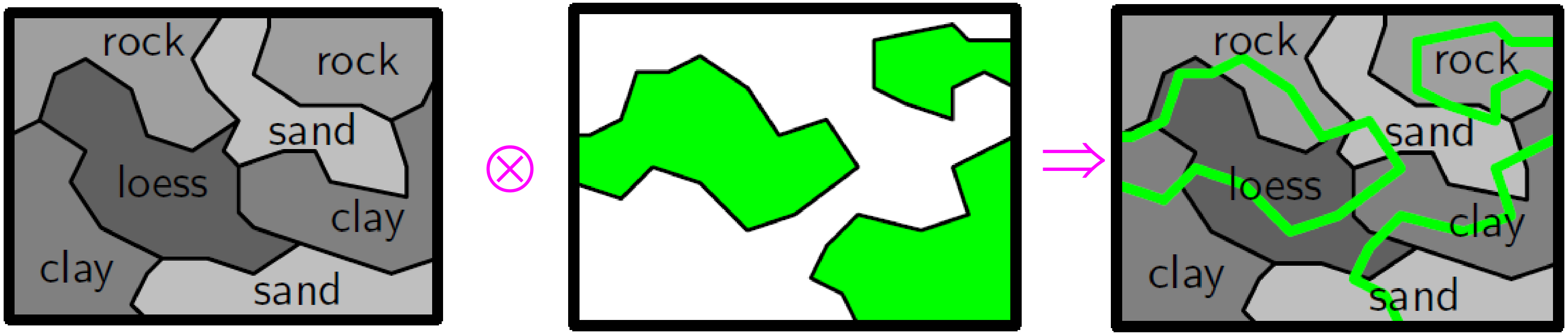
A convex polygon comprises two monotone chains

Union of two simple polygons

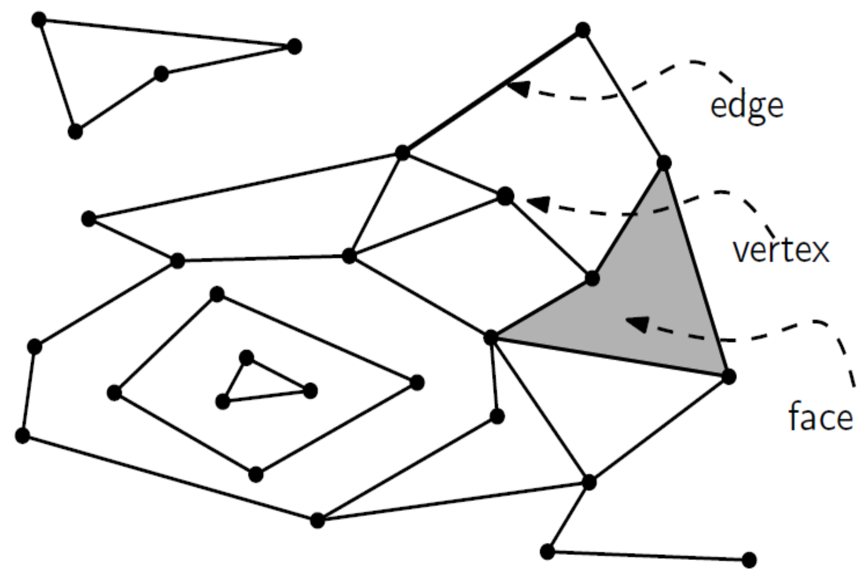


- Assume: unified portion does not contain a hole (always true for convex polygons)
- Start from an intersection point
- Always take the leftmost move while traversing boundaries CW

Map Overlay



To solve map overlay questions, we need to represent subdivisions

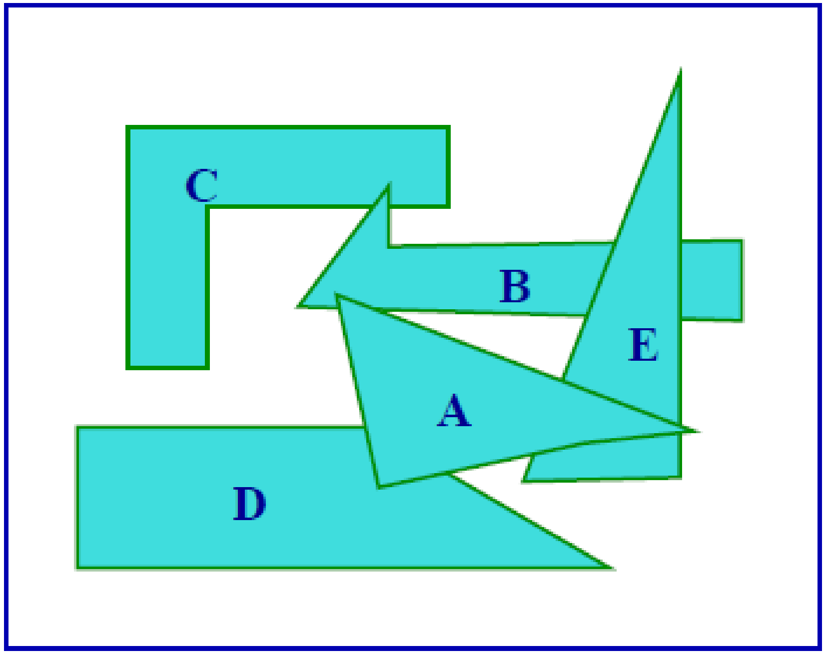


A planar subdivision is a structure induced by a set of line segments in the plane that can only intersect at common endpoints. It consists of vertices, edges, and faces

Representation: DCEL

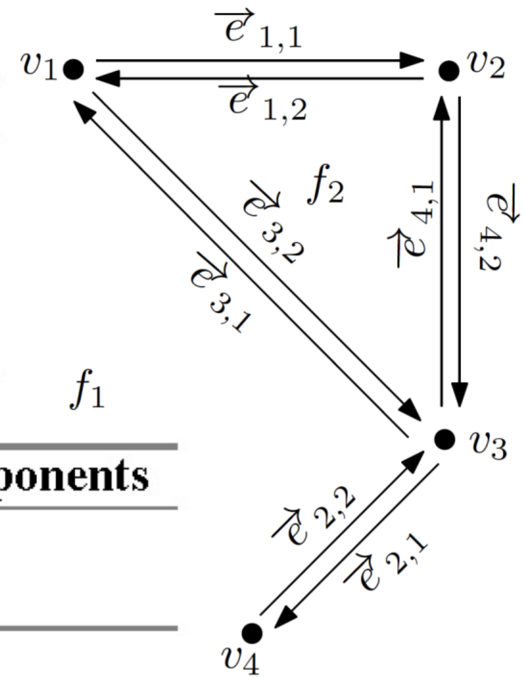
Intersections and Map Overlay

- How do we organize a planar subdivision for easy access to useful information?
- How to tell that objects *A, B, E* create a hole? Which edges bound that hole?
- The planar subdivision, or planar straight line graph (PSLG), is the embedding of a geometric graph



Example DCEL

Vertex	Coordinates	IncidentEdge
v_1	$(0,4)$	$\vec{e}_{1,1}$
v_2	$(2,4)$	$\vec{e}_{4,2}$
v_3	$(2,2)$	$\vec{e}_{2,1}$
v_4	$(1,1)$	$\vec{e}_{2,2}$



Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

Half-edge	Origin	Twin	IncidentFace	Next	Prev
$\vec{e}_{1,1}$	v_1	$\vec{e}_{1,2}$	f_1	$\vec{e}_{4,2}$	$\vec{e}_{3,1}$
$\vec{e}_{1,2}$	v_2	$\vec{e}_{1,1}$	f_2	$\vec{e}_{3,2}$	$\vec{e}_{4,1}$
$\vec{e}_{2,1}$	v_3	$\vec{e}_{2,2}$	f_1	$\vec{e}_{2,2}$	$\vec{e}_{4,2}$
$\vec{e}_{2,2}$	v_4	$\vec{e}_{2,1}$	f_1	$\vec{e}_{3,1}$	$\vec{e}_{2,1}$
$\vec{e}_{3,1}$	v_3	$\vec{e}_{3,2}$	f_1	$\vec{e}_{1,1}$	$\vec{e}_{2,2}$
$\vec{e}_{3,2}$	v_1	$\vec{e}_{3,1}$	f_2	$\vec{e}_{4,1}$	$\vec{e}_{1,2}$
$\vec{e}_{4,1}$	v_3	$\vec{e}_{4,2}$	f_2	$\vec{e}_{1,2}$	$\vec{e}_{3,2}$
$\vec{e}_{4,2}$	v_2	$\vec{e}_{4,1}$	f_1	$\vec{e}_{2,1}$	$\vec{e}_{1,1}$

Doubly-Connected Edge Lists (DCEL)

A vertex object stores:

- Coordinates
- **IncidentEdge**
- *Any attributes, mark bits*

A face object stores:

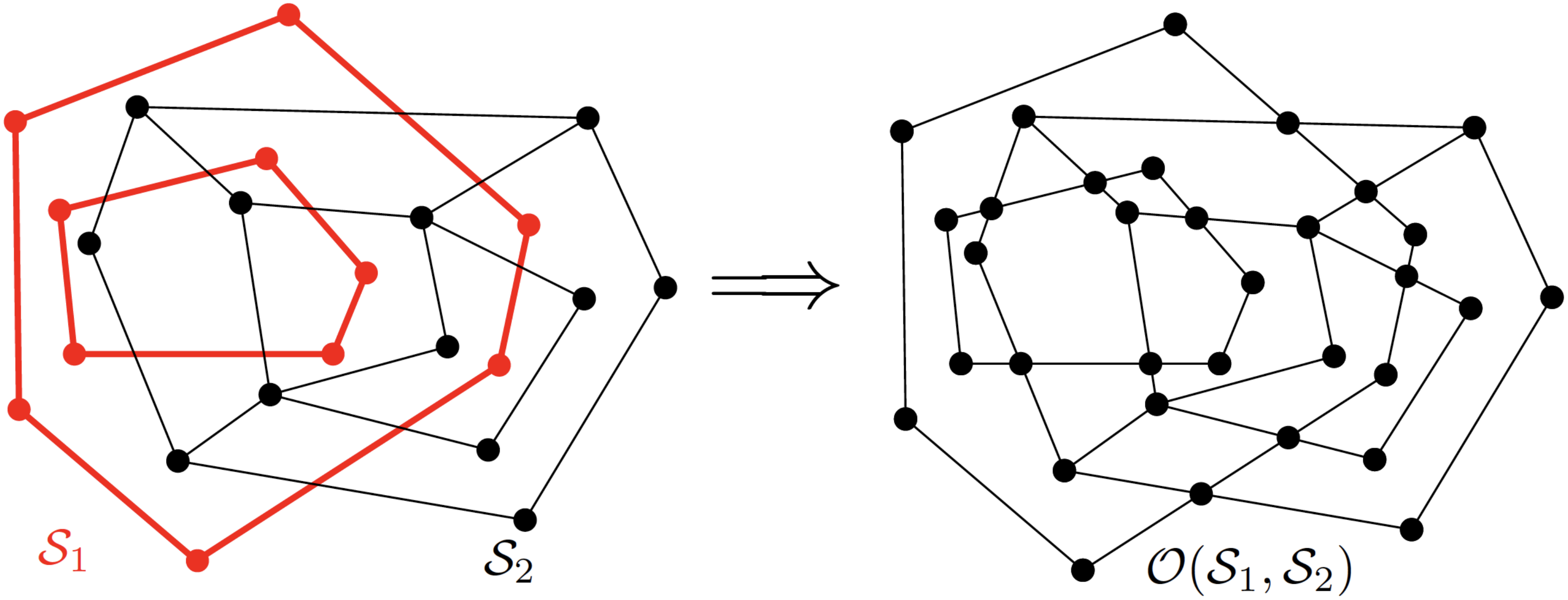
- **OuterComponent** (half-edge of outer cycle)
- **InnerComponents** (half-edges for the inner cycles)
- *Any attributes, mark bits*

A half-edge object stores:

- **Origin** (vertex)
- **Twin** (half-edge)
- **IncidentFace** (face)
- **Next** (half-edge in cycle of the incident face)
- **Prev** (half-edge in cycle of the incident face)
- *Any attributes, mark bits*

Computing the Overlay

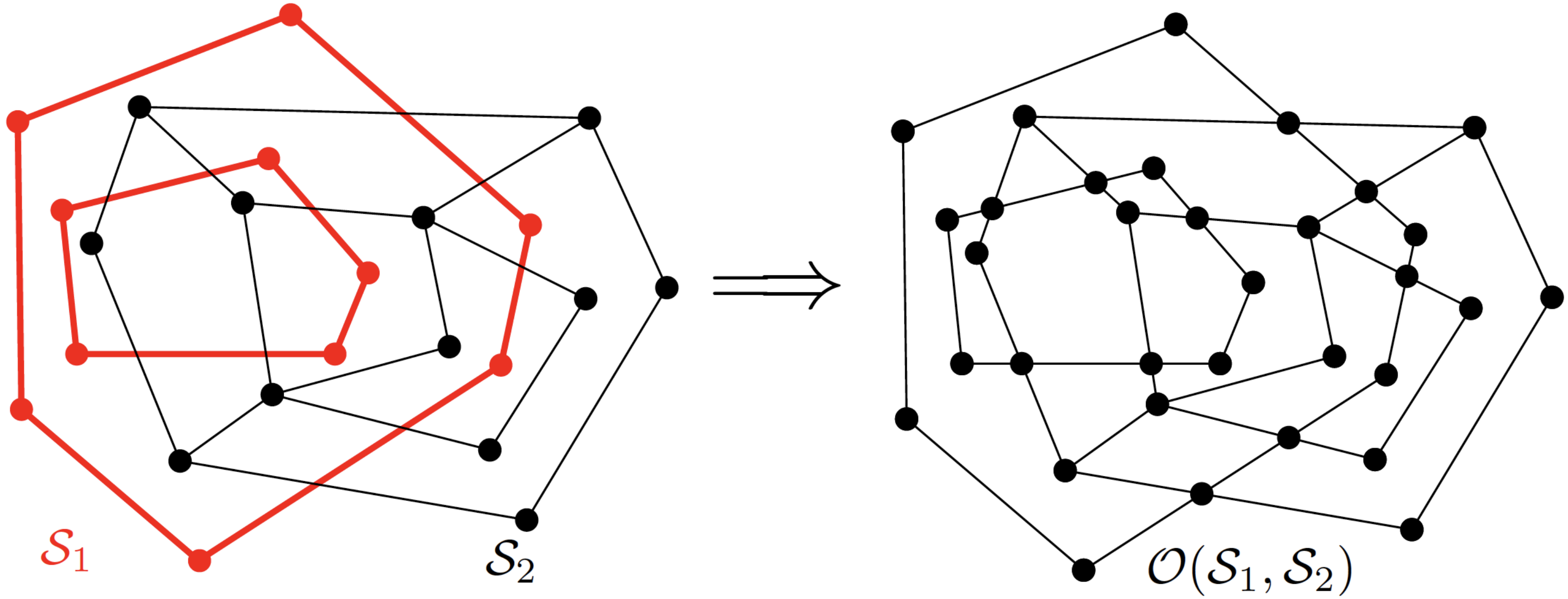
- Input: DCEL for S_1 and DCEL for S_2
- Output: DCEL for the overlay of S_1 and S_2



Ref: 4A Textbook

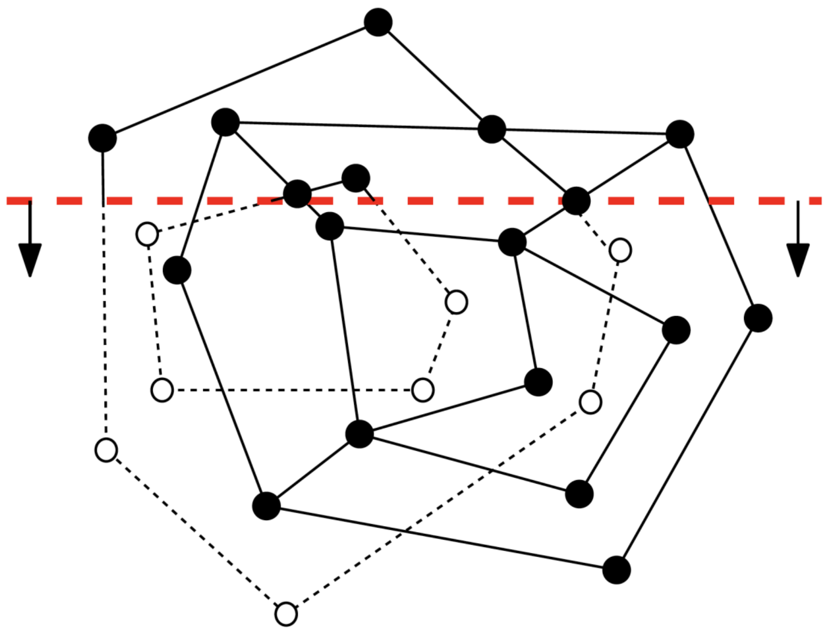
Computing the Overlay

- Initialization: copy the DCEL for S_1 and S_2
- These are then “merged” into one



Use the plane-sweep algorithm

- Plane-sweep as in line-segment intersection



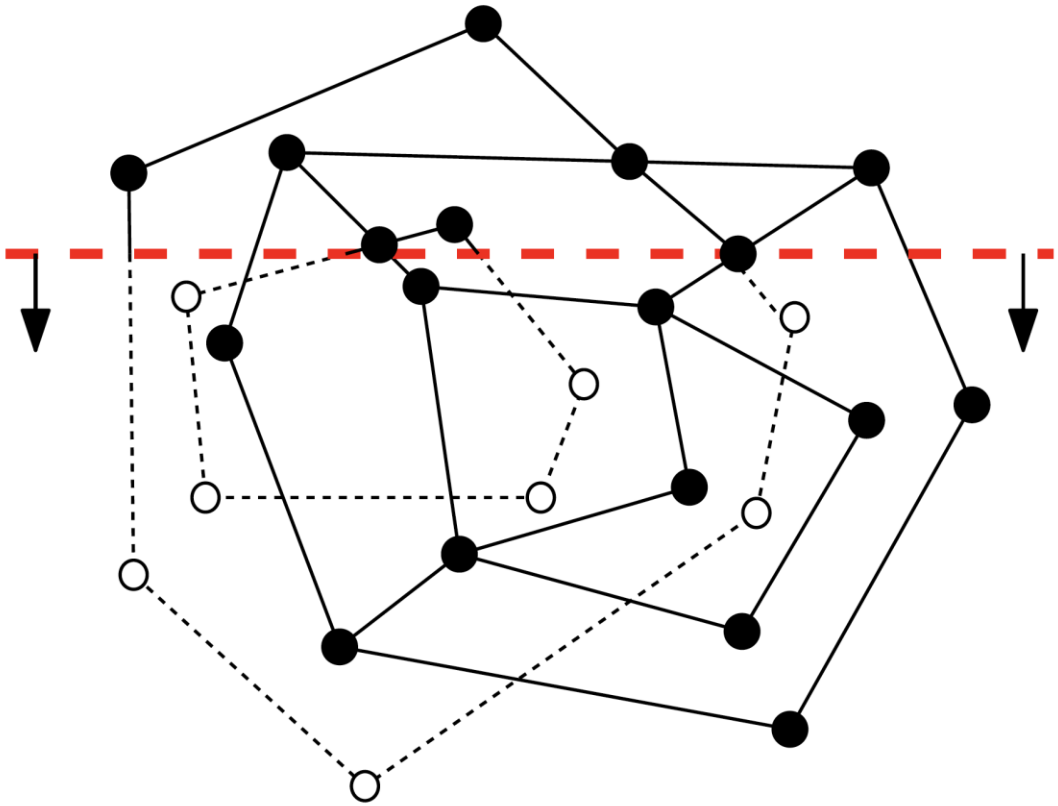
Status: the edges of S_1 and S_2 intersecting the sweep line in the left-to-right order;

Events happen:
at the vertices of S_1 and S_2;

at intersection points from S_1 and S_2

Event Management using DCEL

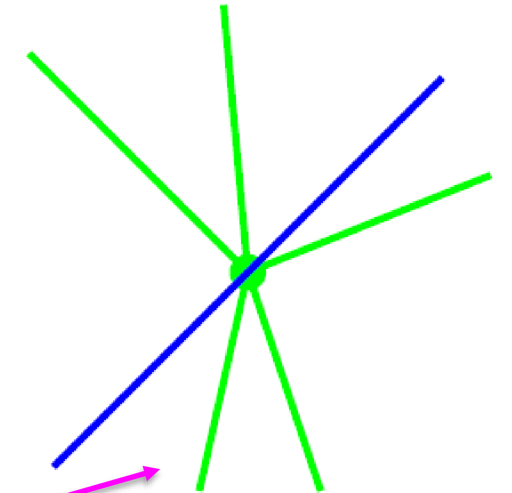
- For each intersection event, add a new vertex to the merged DCEL

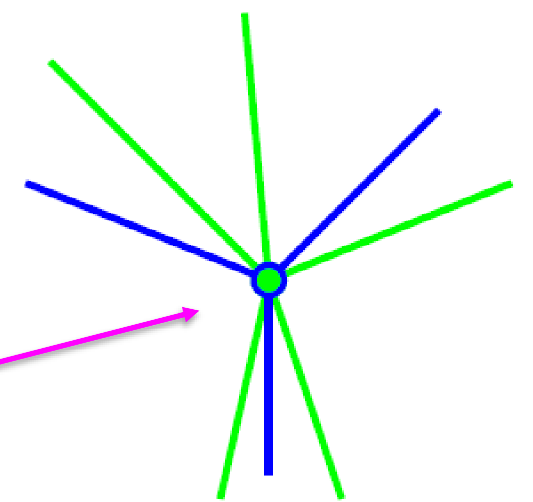


Overlay Events

Six types of events:

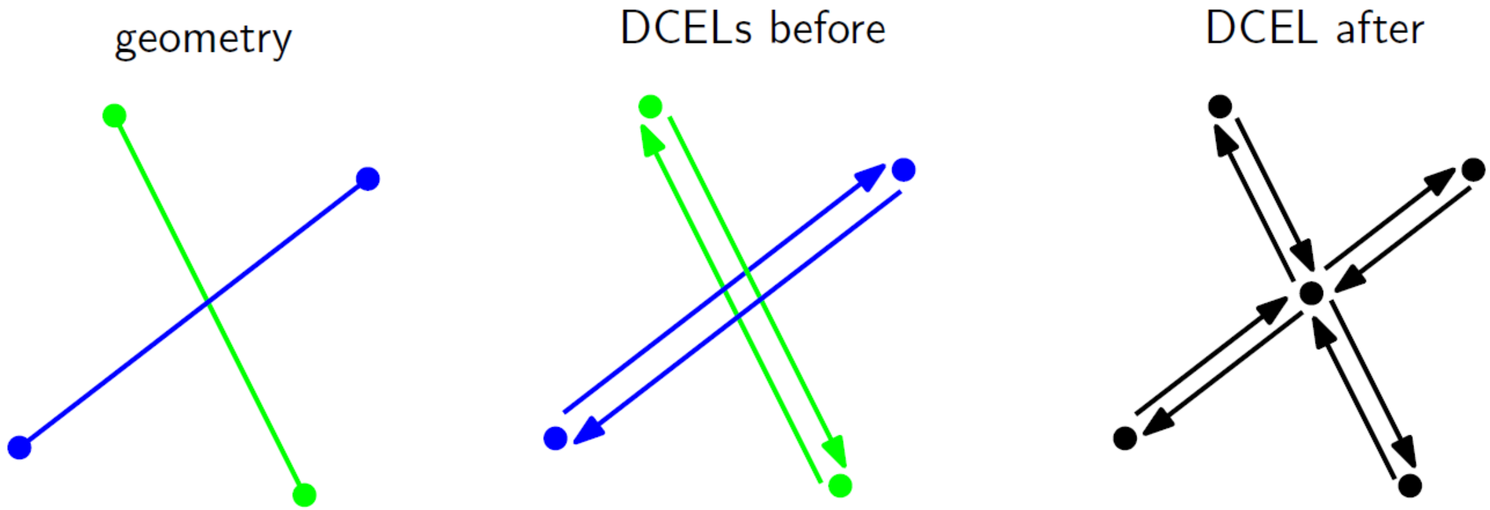
- A vertex of S_1
- A vertex of S_2
- An intersection point of one edge from S_1 and one edge from S_2
- An edge of S_1 goes through a vertex of S_2
- An edge of S_2 goes through a vertex of S_1
- A vertex of S_1 and a vertex of S_2 coincide





Overlay Events

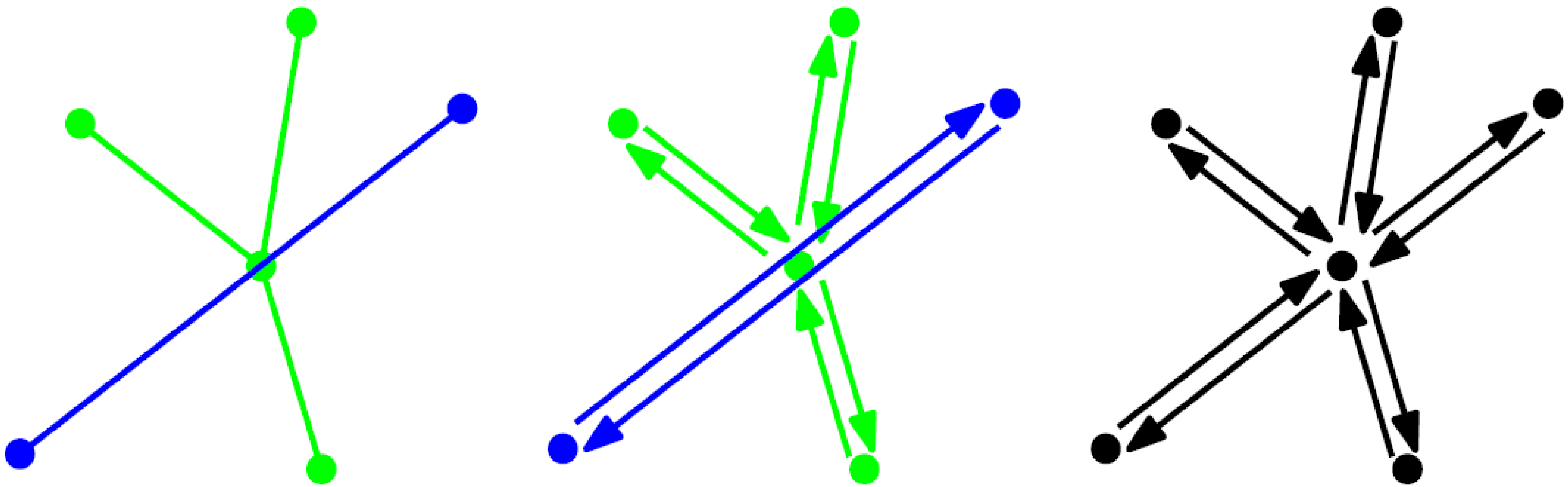
Consider the event:
an intersection point of one edge from S_1 and one edge from S_2



Overlay Events

Consider the event:
an edge from S_1 goes through a vertex of S_2

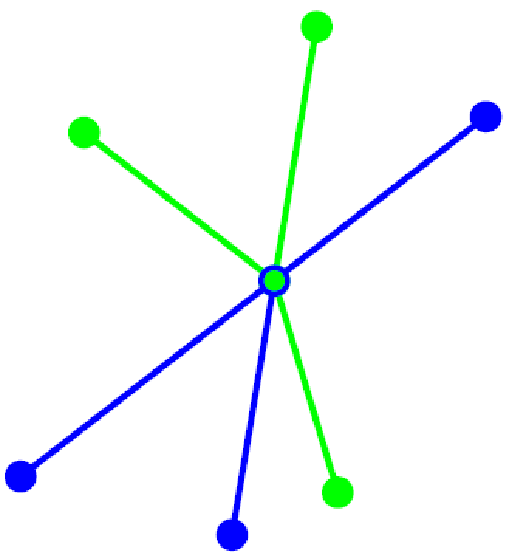
geometry　　DCELs before　　DCEL after



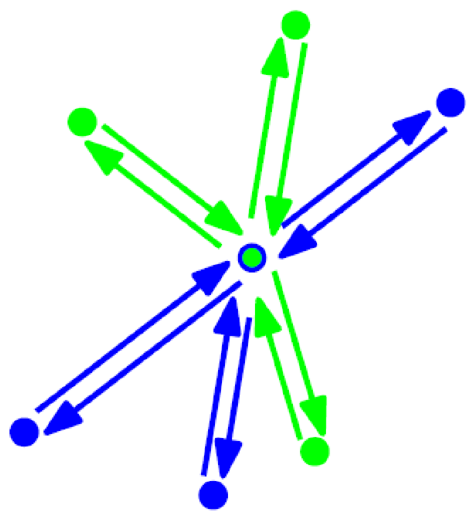
Overlay Events

Consider the event:
a vertex of S_1 and a vertex of S_2 coincide

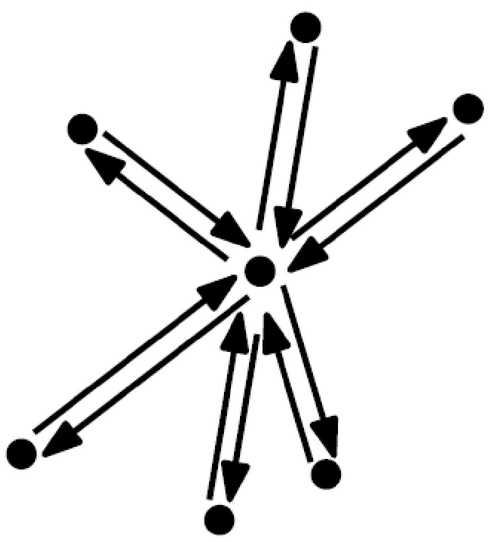
geometry



DCELs before



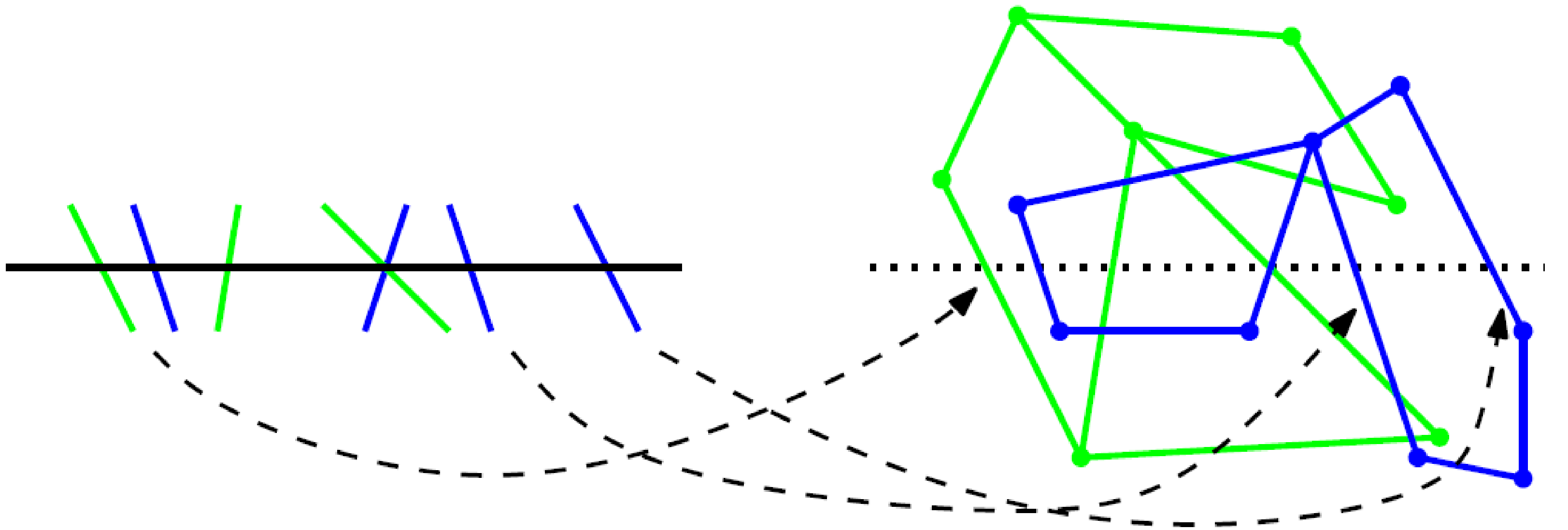
DCEL after



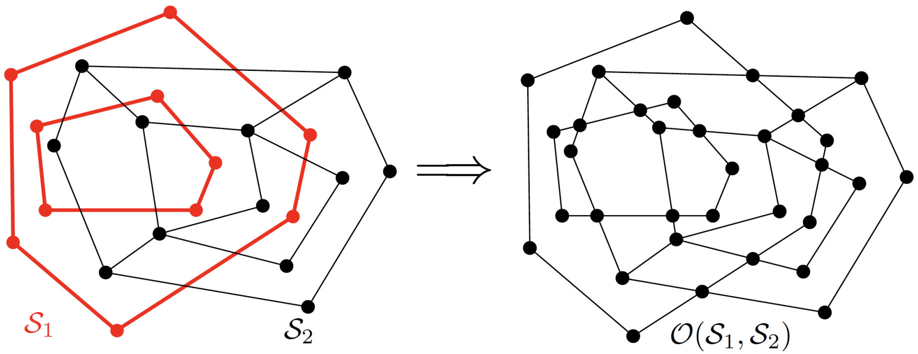
Overlay Data Structure

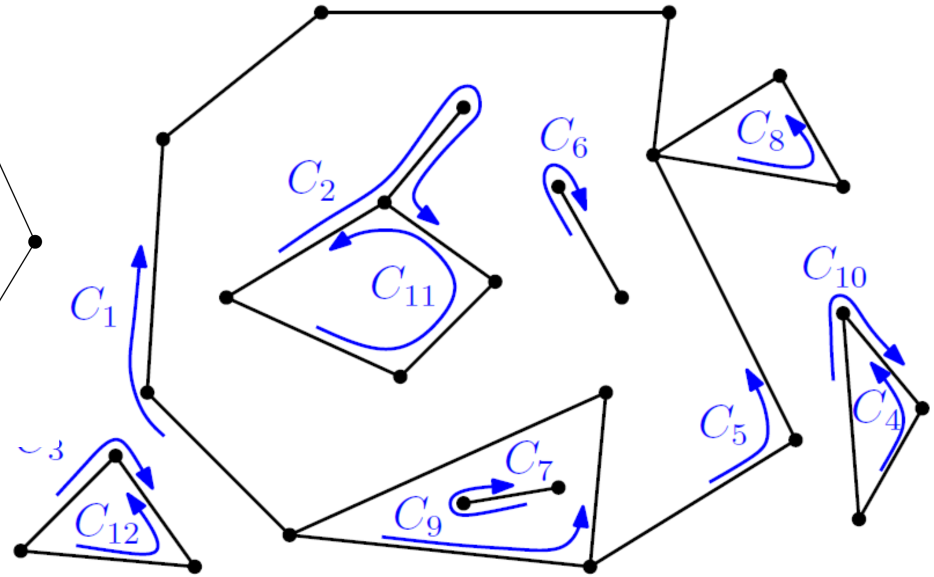
When we take an event from the event queue Q, we need quick access to the DCEL to make the necessary changes

We keep a pointer from each leaf in the status structure to one of the representing half-edges in the DCEL



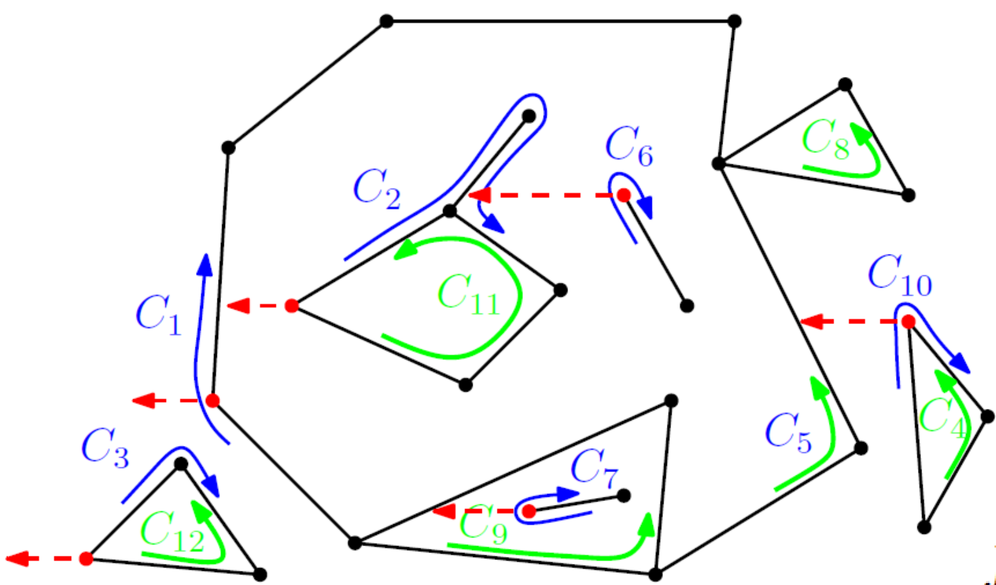
Overlay Face Updating



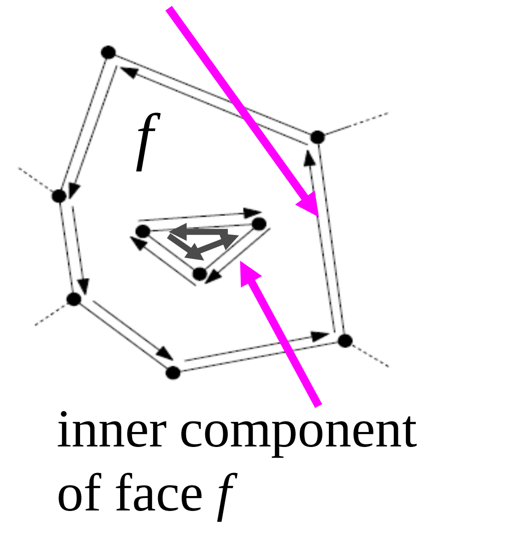


- Determine all cycles of half-edges, and whether they are inner or outer boundaries of the incident face
- Make a face object for each outer boundary, plus one for the unbounded face, and set the **OuterComponent** variable of each face. Set the **IncidentFace** variable for every half-edge in an outer boundary cycle

Overlay Face Updating



$C_1 \rightarrow$
$C_3 \rightarrow$
$C_2 \rightarrow C_5$
$C_7 \rightarrow C_9$
$C_6 \rightarrow C_2$
$C_{10} \rightarrow C_1$



f_5: outer e_5; inner e_2, e_6

Determine the leftmost vertex of each inner boundary cycle;
Determine the edge horizontally left of it, take the downward half-edge and its cycle to set **InnerComponents**;
Set **IncidentFace** for half-edges to inner-boundary cycle;

Ref: 4A Textbook

Analysis

n: sum of the complexities of input DCELs

Every event takes $O(\log n)$ or $O(m + \log n)$ time to handle, where m is the sum of the degrees of any vertex from S_1 and/or S_2 involved

The sum of the degrees of all vertices is exactly twice the number of edges

Theorem: Given two planar subdivisions S_1 and S_2, their overlay can be computed in $O(n \log n + k \log n)$ time, where k is the number of vertices of the overlay