OI-OPPA. European Social Fund
Prague & EU: We invest in your future.

GEOMETRIC SEARCHING PART 1: POINT LOCATION

PETR FELKEL

FEL CTU PRAGUE

felkel@fel.cvut.cz

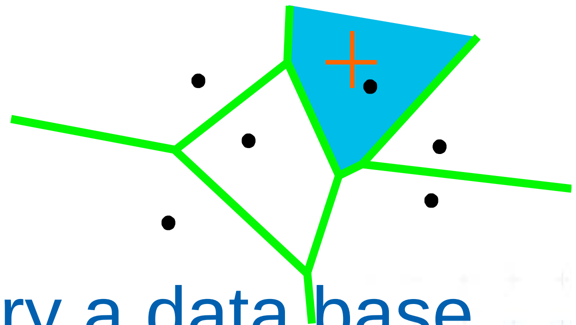
https://cw.felk.cvut.cz/doku.php/courses/a4m39vg/start

Based on [Berg] and [Mount]

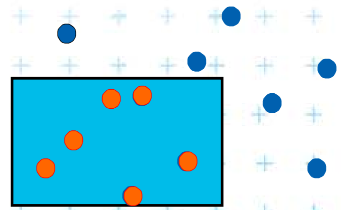
Version from 4.10.2012

Geometric searching problems

- Point location (static) – Where am I?
  - (Find the name of state, pointed by the mouse (cursor))
  - Search space S: a planar (spatial) subdivision
  - Query: point Q
  - Answer: region containing Q



- Orthogonal range searching – Query a data base
(Find points, located in d-dimensional axis-parallel box)
  - Search space S: a set of points
  - Query: set of orthogonal intervals q
  - Answer: subset of points in the box
  - (Was studied in DPG)

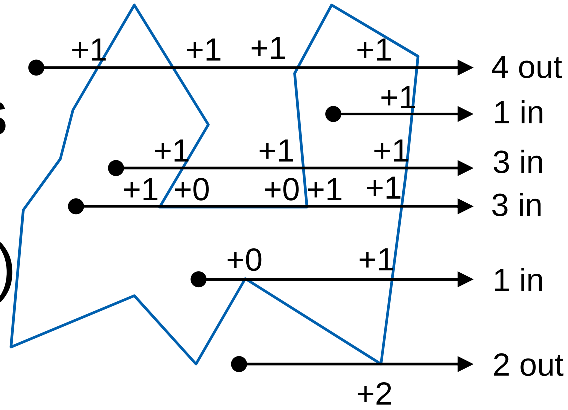


Point location

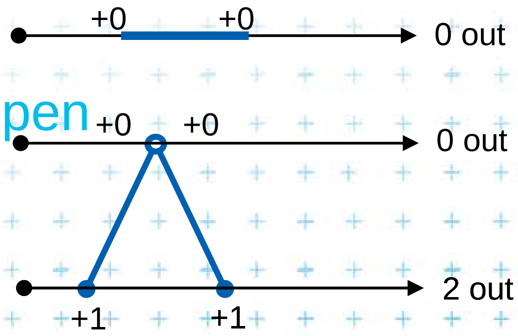
- Point location in polygon
- Planar subdivision
- DCEL data structure
- Point location in planar subdivision
  - slabs
  - monotone sequence
  - trapezoidal map

Point location in polygon by ray crossing

1. Ray crossing - O(n)
   – Compute number *t* of intersections of ray with polygon edges (e.g., X+ after point move to origin)
   – If odd(*t*) then inside
     else out

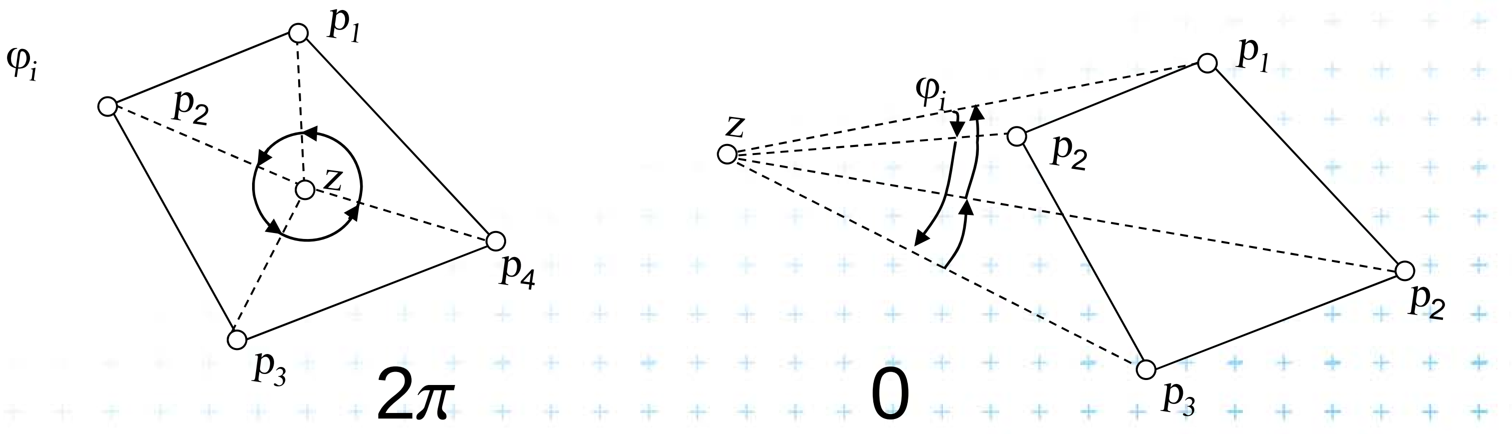


   – Singular cases must be handled!
     - Do not count horizontal line segments
     - Take non-horizontal segments as half-open (upper point not part of the segment)



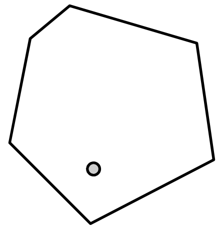
Point location in polygon

2. Winding number - O(n)
   (number of turns around the point)
   - Sum angles φi = ∠(p_i, z, p_{i+1})
   - If (sum φi = 2π) then inside (1 turn)
   - If (sum φi = 0) then outside (no turn)
   - About 20-times slower than ray crossing

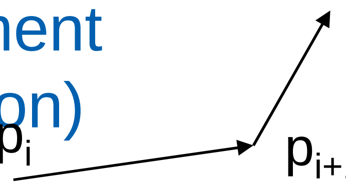


Point location in polygon

3. Position relative to all edges
   – For convex polygons
   – If (left from all edges) then inside



- Position of point in relation to the line segment (Determination of convex polygon orientation)



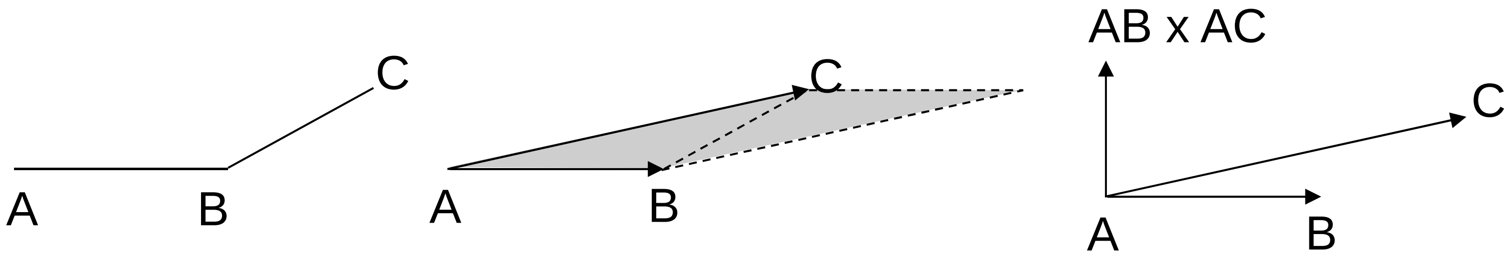
  – Convex polygon,
  noncollinear points $p_i = [x_i, y_i, 1]$, $p_{i+1} = [x_{i+1}, y_{i+1}, 1]$, $p_{i+2} = [x_{i+2}, y_{i+2}, 1]$

$$\begin{vmatrix} x_i & y_i & 1 \\ x_{i+1} & y_{i+1} & 1 \\ x_{i+2} & y_{i+2} & 1 \end{vmatrix}$$

> 0 => point left from edge (CCW polygon)

< 0 => point right from edge (CW polygon)

Area of Triangle



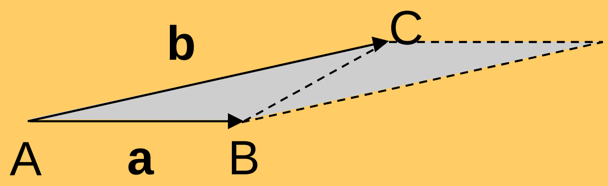
Vector product of vectors AB x AC

= Vector perpendicular to both vectors AB and AC

- For vectors in plane is perpendicular to the plane (normal)
- In 2D (plane *xy)* – has only *z*-coordinate is non-zero
- |AB x AC| = z-coordinate of the normal vector
- = area of parallelopid
- = 2x area *T* of triangle ABC

Area of Triangle

- $T = ½ |AB \times AC|$



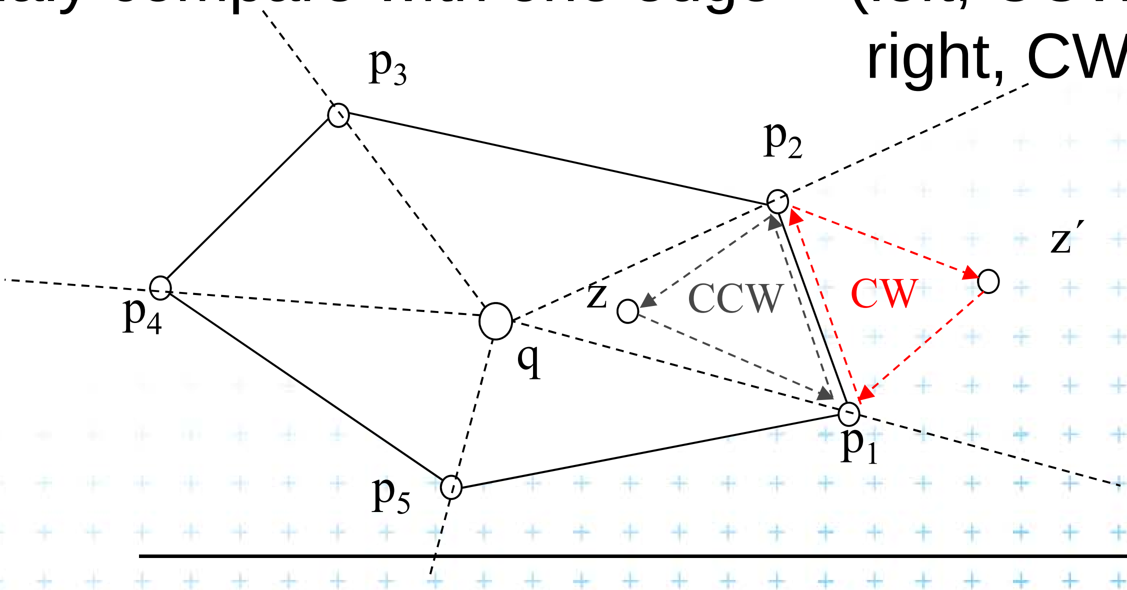
- $\mathbf{a} = B - A$
- $\mathbf{b} = C – A$
- $T = ½ (\mathbf{a}_x \mathbf{b}_y - \mathbf{a}_y \mathbf{b}_x)$

$$\Rightarrow 2T = A_x B_y + B_x C_y + C_x A_y - A_x C_y - B_x A_y - C_x B_y$$

$$2T = \begin{vmatrix} A_x & A_y & 1 \\ B_x & B_y & 1 \\ C_x & C_y & 1 \end{vmatrix} = A_x B_y + B_x C_y + C_x A_y - A_x C_y - B_x A_y - C_x B_y$$

Point location in polygon

4. Binary search in angles
   - Works for convex and star-shaped polygons
   - Choose any point *q* inside
   - *q* forms wedges with polygon edges
   - Binary search of wedge based on angle
   - Finaly compare with one edge (left, CCW => in, right, CW => out)

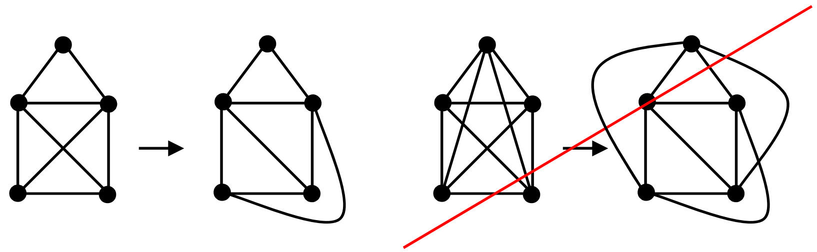


Planar graph

Planar graph

U=set of nodes, H=set of arcs

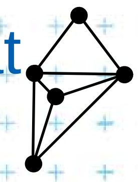
- = Graph G = (U,H) is planar, if it can be embedded into plane without crossings



Planar embedding of planar graph G = (U,H)

- = mapping of each *node in U to vertex* in the plane and each *arc in H into simple curve (edge)* between the two images of extreme nodes of the arc, so that **no** two **images of arc intersect** except at their endpoints

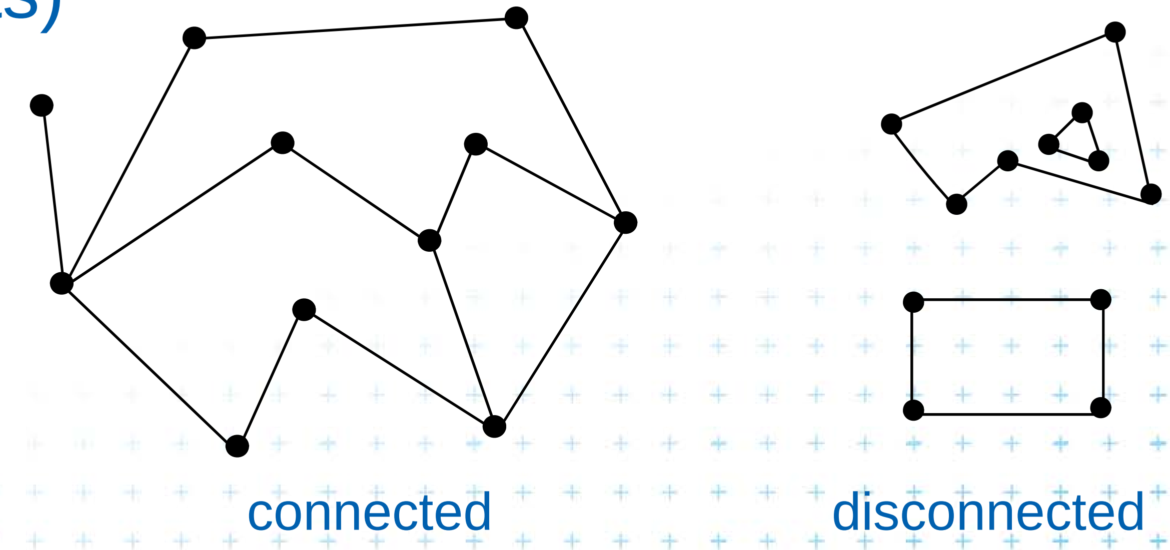
Every planar graph can be embedded in such a way that arcs map to straight line segments [Fáry 1948]



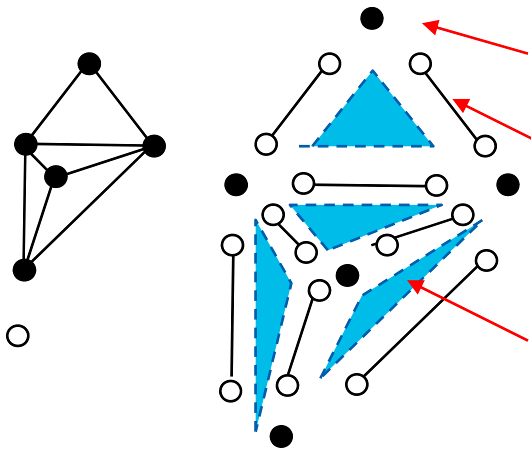
Planar subdivision

= Partition of the plane determined by straight line planar embedding of a planar graph.
Also called PSLG – Planar straight line graph

- (embedding of a planar graph in the plane such that its arcs are mapped into straight line segments)



Planar subdivision



Vertex = embedding of graph node

Edge = embedding of graph arc
(open – without vertices)

Face = maximal connected subset of a plane that doesn't contain points on edges nor vertices (open polygonal region whose boundary is formed by edges and vertices from the subdivision)

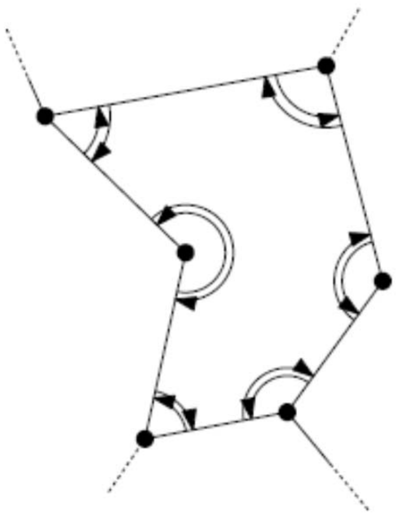
Complexity (size) of a subdivision = sum of number of vertices +
+ number of edges +
+ number of faces it consists of

Euler's formula: |V| - |E| + |F| >= 2

DCEL = Double Connected Edge List

- A structure for storage of planar subdivision
- Operations like:

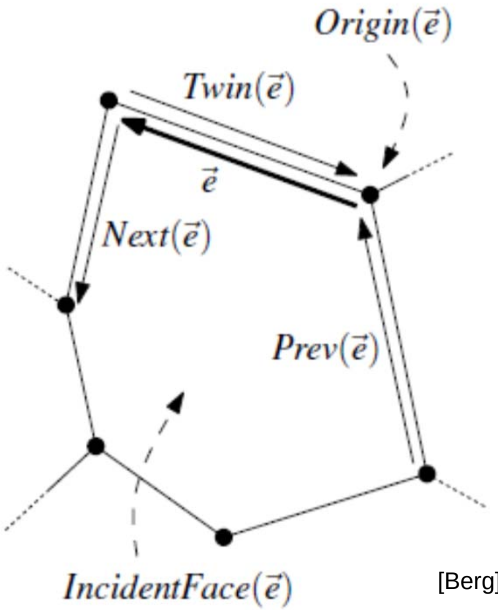
Walk around boundary of a given face



[Berg]

Pointers to next and prev edge

Get incident face

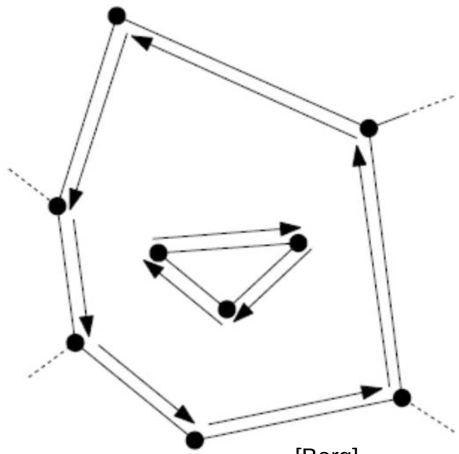


[Berg]

Half-edge, op. Twin(e), unique Next(e), Prev(e)

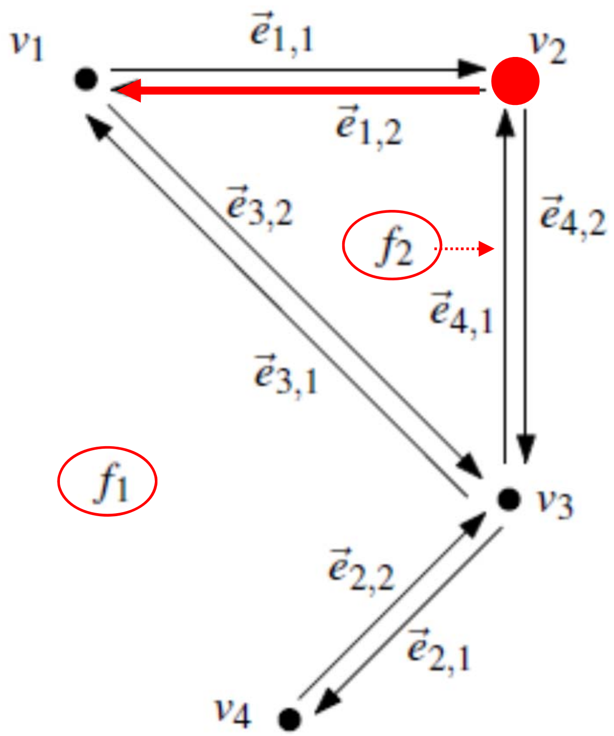
DCEL = Double Connected Edge List

- Vertex record v
  - Coordinates(v) and pointer to IncidentEdge(v)
- Face record f
  - OuterComponent(f) pointer (boundary)
  - List of holes – InnerComponent(f)
- Half-edge record e
  - Origin(e), Twin(e), IncidentFace(e)
  - Next(e), Prev(e)
  - [Dest(e) = Origin(Twin(e))]
- Possible attribute data for each



[Berg]

DCEL = Double Connected Edge List



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	$(1,1)$	$\vec{e}_{2,2}$

One of edges

List of holes

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	$\vec{e}_{2,1}$	$\vec{e}_{1,1}$

[Berg]

DCEL simplifications

- If no operations with vertices and no attributes
  - Store vertex coords in origin
  - No separate vertex record
- If no need for faces (e.g. river network)
  - No face record and no IncidentFace() field
- If only connected subdivision allowed
  - Join holes with rest by dummy edges
  - Visit all half-edges by simple graph traversal
  - No InnerComponent() list for faces

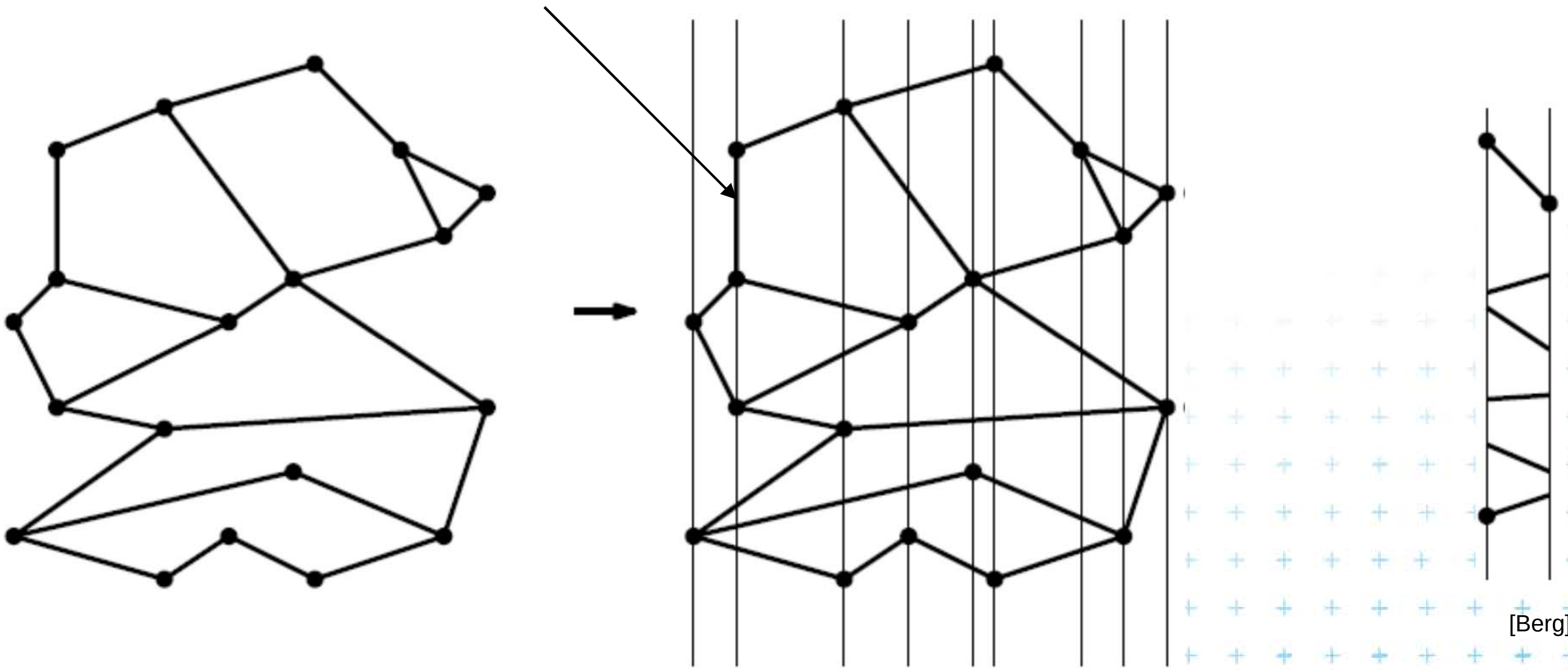
Point location in planar subdivision

- Using special search structures an optimal algorithm can be made with
  - O(n) preprocessing,
  - O(n) memory and
  - O(log n) query time.
- Simpler methods
  1. Slabs O(log n) query
  2. monotone chain tree O($\log^2$ n) query
  3. trapezoidal map O(log n) query expected time

1. Vertical (horizontal) slabs

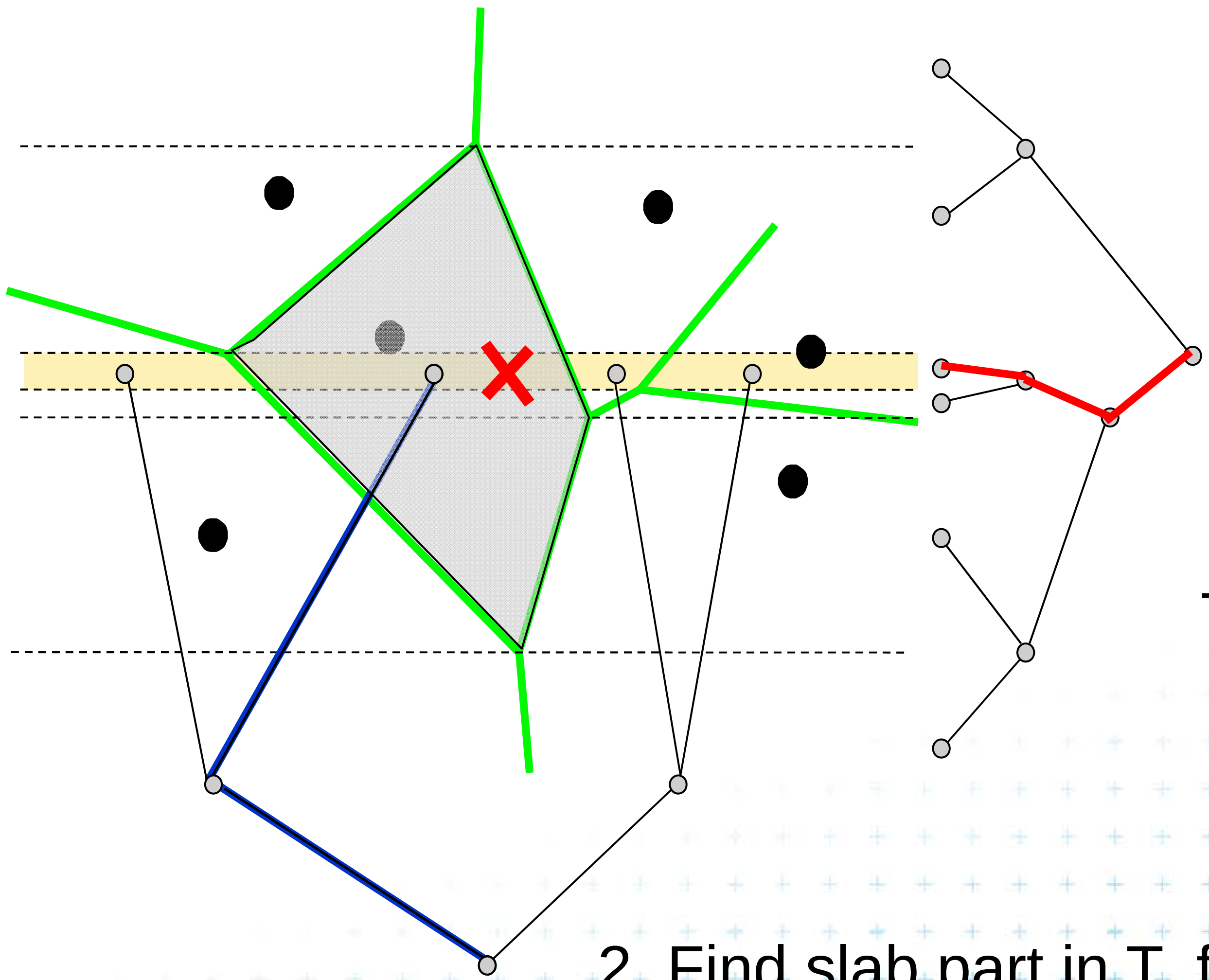
[Dobkin and Lipton, 1976]

- Draw vertical or horizontal lines through vertices
- It partitions the plane into vertical slabs
  - Avoid points with same x coordinate (to be solved later)



[Berg]

Horizontal slabs example



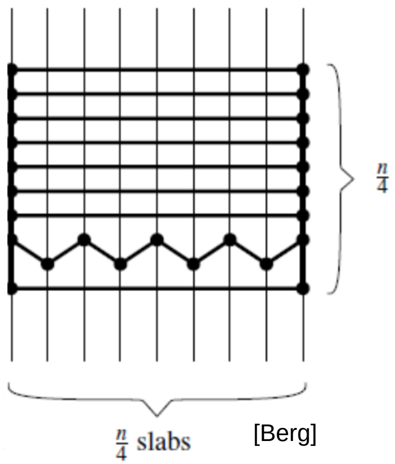
1. Find slab in T_y for y

T_x and T_y are arrays

2. Find slab part in T_x for x

Horizontal slabs complexity

- Query time $O(\log n)$
  - $O(\log n)$ time in slab array T_y (size max 2n endpoints)
  - + $O(\log n)$ time in slab array T_x (slab crossed max by n edges)
- Memory $O(n^2)$
  - Slabs: Array with y-coordinates of vertices … $O(n)$
  - For each slab O(n) edges intersecting the slab



$O(n \log n)$ construction

$O(\log n)$ query

$O(n^2)$ memory

2. Monotone chain tree

[Lee and Preparata, 1977]

- Construct monotone planar subdivision
  - The edges are monotone in the same direction
- Each separator chain
  - is monotone (can be projected to line an searched)
  - splits the plane into two parts – allows binary search
- Algorithm
  - Preprocess: Find the separators (e.g., horizontal)
  - Search:

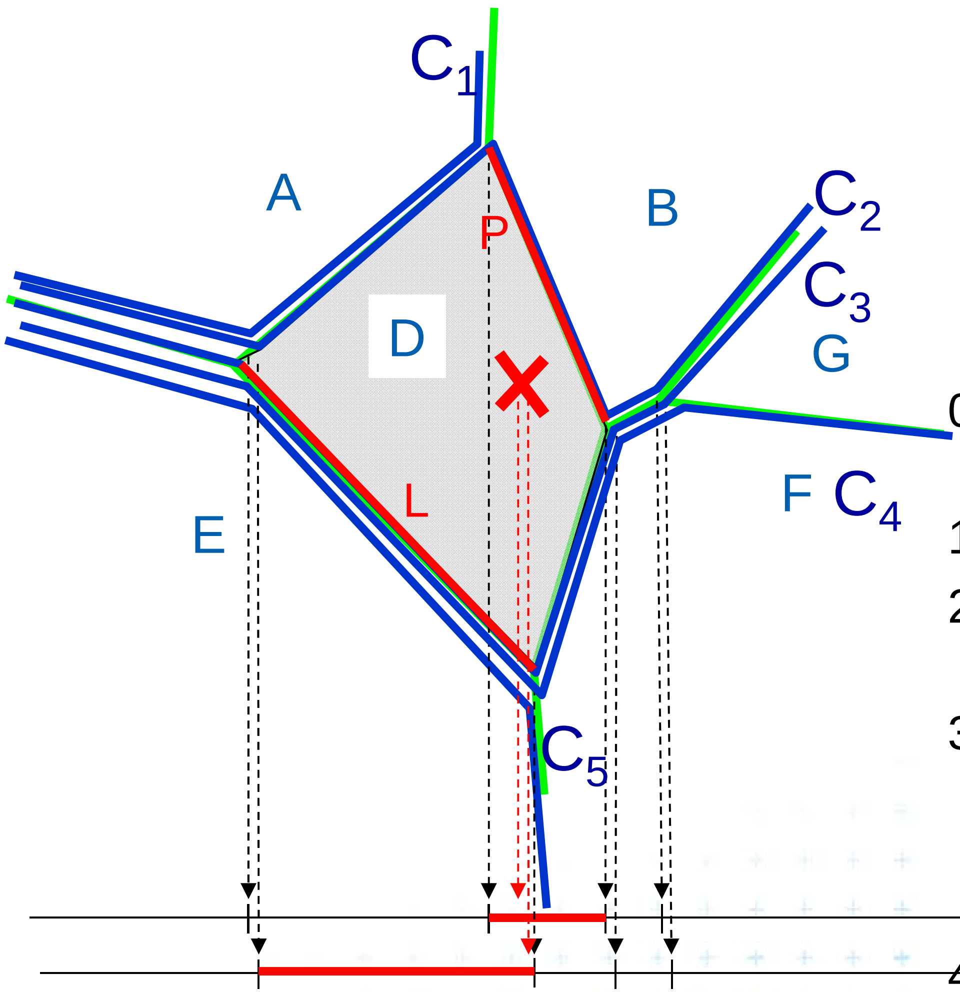
    Binary search among separators (Y) … $O(\log n)$

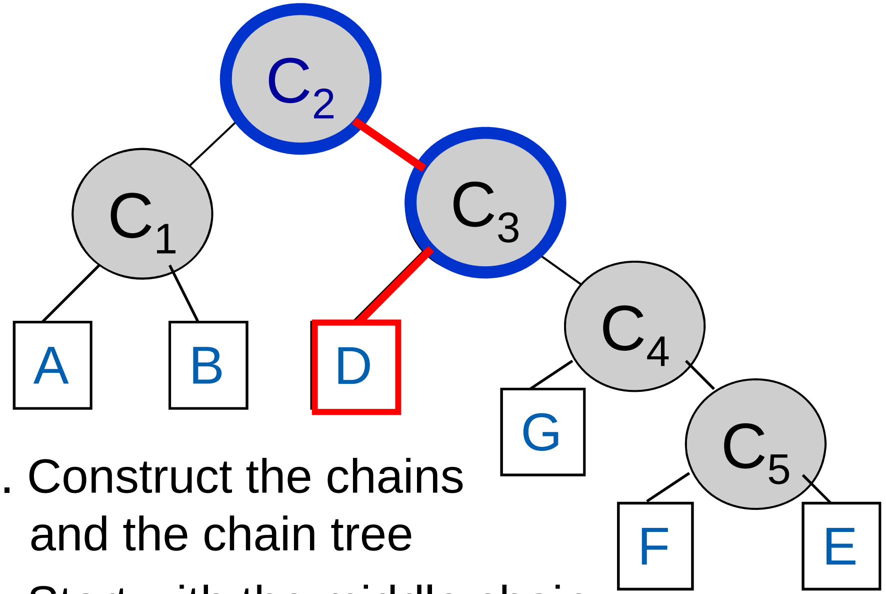
    Binary search along the separator (X) … $O(\log n)$

    $O(\log^2 n)$ query

    $O(n^2)$ memory
  - Not optimal, but simple
  - Can be made optimal, but the algorithm and data structures are complicated

Monotone chain tree example

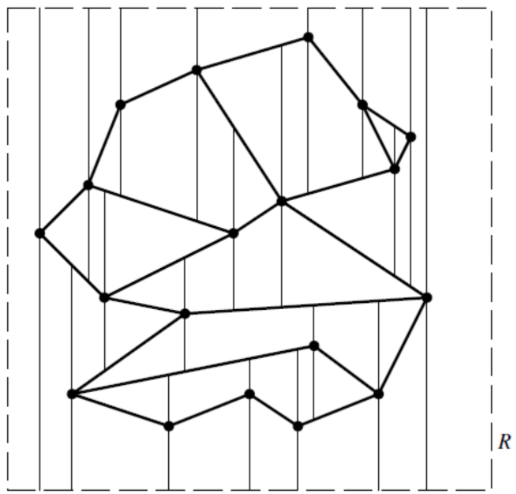




0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right (This is the position of *x* relatively to the whole chain)
4. Continue in L or R chain -> goto 2. or stop if in the leaf

3. Trapezoidal map (TM) search

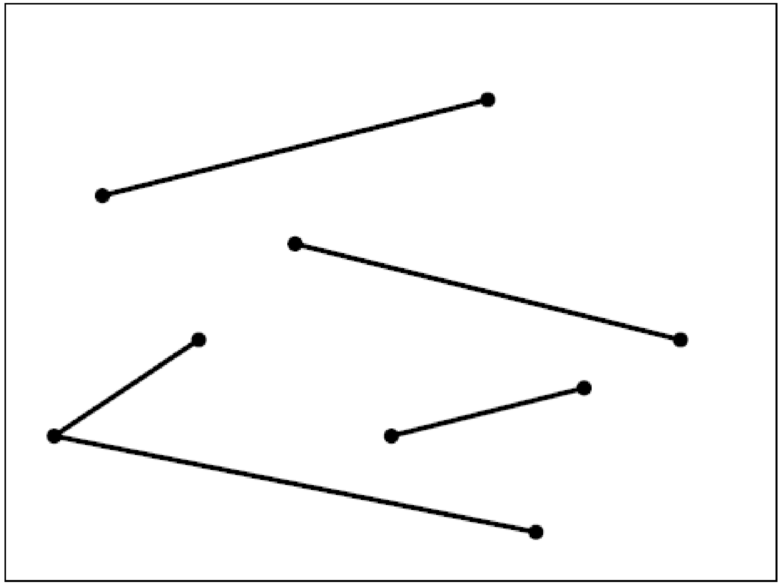
- The simplest and most practical known optimal algorithm
- Randomized algorithm with O(n) expected storage and O(log n) expected query time
- Expectation depends on the random order of segments during construction, not on the position of the segments
- TM is refinement of original subdivision
- Converts complex shapes into simple ones
- Weaker assumption on input:
  - Input individual segments, not polygons
  - $S = \{s_1, s_2, \ldots, s_n\}$
  - S_i subset of first *i* segments
  - Answer: segment below the pointed trapezoid (Δ)



[Berg]

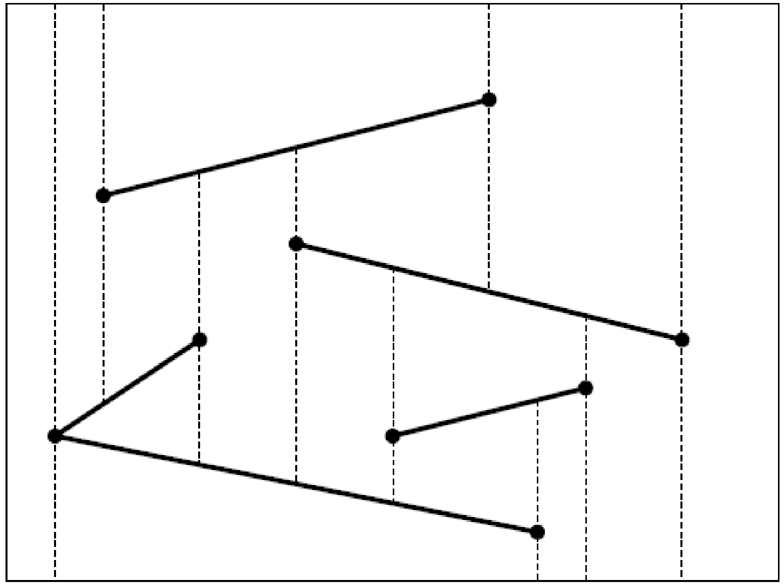
Trapezoidal map of line segments in general position

Input: individual segments *S*



Constru-
ction

Trapezoidal map *T*



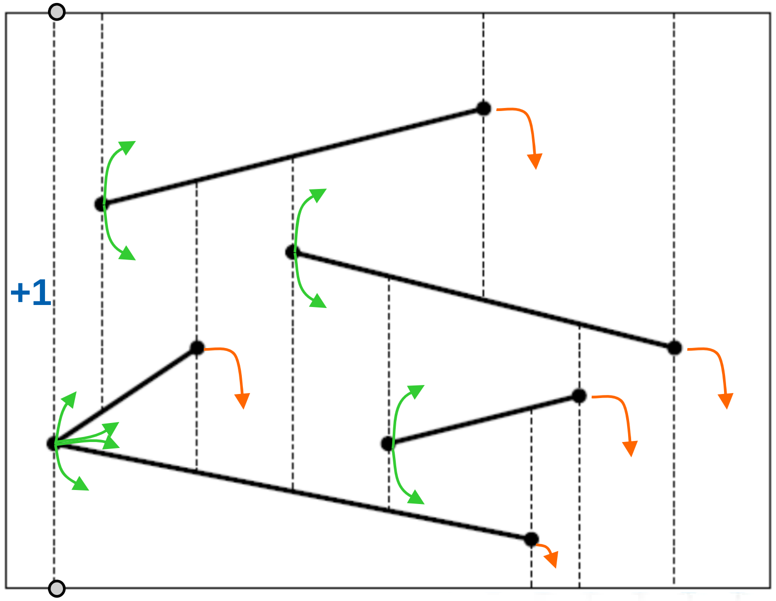
- They do not intersect, except in endpoints
- No vertical segments
- No 2 distinct endpoints with the same x-coordinate

[Mount]

- Bounding rectangle
- 4 Bullets up and down
- Stop on input segment or on bounding rectangle

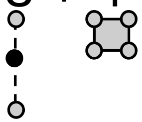
Trapezoidal map of line segments in general position

- Faces are trapezoids Δ with vertical sides
- Given n segments, TM has
  - at most 6n+4 vertices
  - at most 3n+1 trapezoids



[Mount]

- Proof:
  - each point 2 bullets -> 1+2 points
  - 2n endpoints * 3 + 4 = 6n+4 vertices

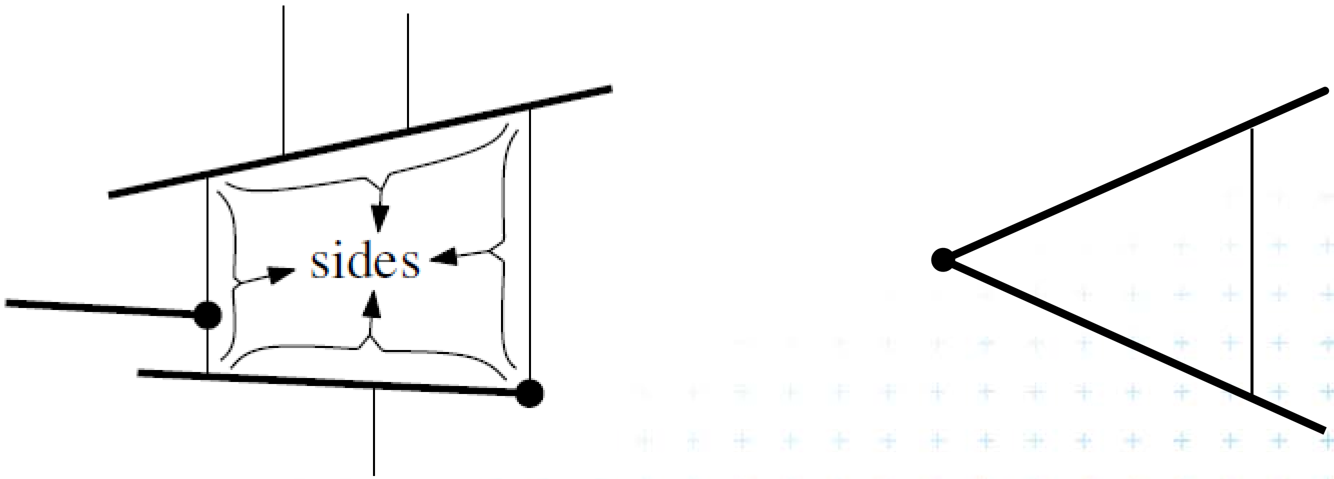


  - start point –> max 2 trapezoids
  - end point –> 1 trapezoid
  - 3 * (n segments) + 1 left Δ => max 3n+1 Δ

Trapezoidal map of line segments in general position

Each face has

- one or two vertical sides (trapezoid or triangle) and
- exactly two non-vertical sides

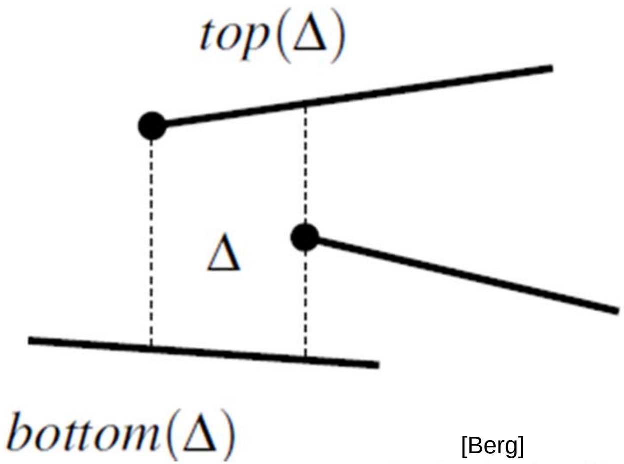


[Berg]

Two non-vertical sides

Non-vertical side

- is contained in a segment of *S*
- or in the horizontal edge of bounding rectangle *R*

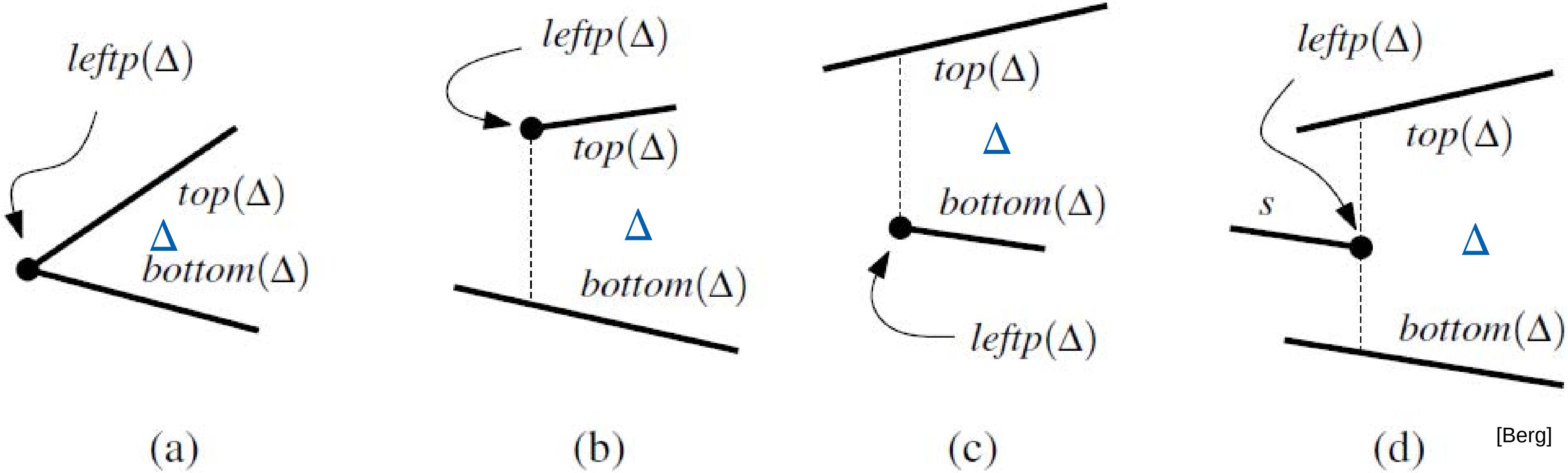


[Berg]

top(Δ) - bounds from above

bottom(Δ) - bounds from below

Vertical sides – left vertical side of Δ



[Berg]

Left vertical side is defined by the segment end-point *p*=*leftp*(Δ)

(a) common left point *p* itself

(b) by the lower vert. extension of left point *p* ending at bottom()

(c) by the upper vert. extension of left point *p* ending at top()

(d) by both vert. extensions of the right point *p*

(e) the left edge of the bounding rectangle R (leftmost Δ only)

Vertical sides - summary

Vertical edges are defined by segment endpoints

- *leftp*(Δ) = the end point defining the left edge of Δ
- *rightp*(Δ) = the end point defining the right edge of Δ

leftp(Δ) is

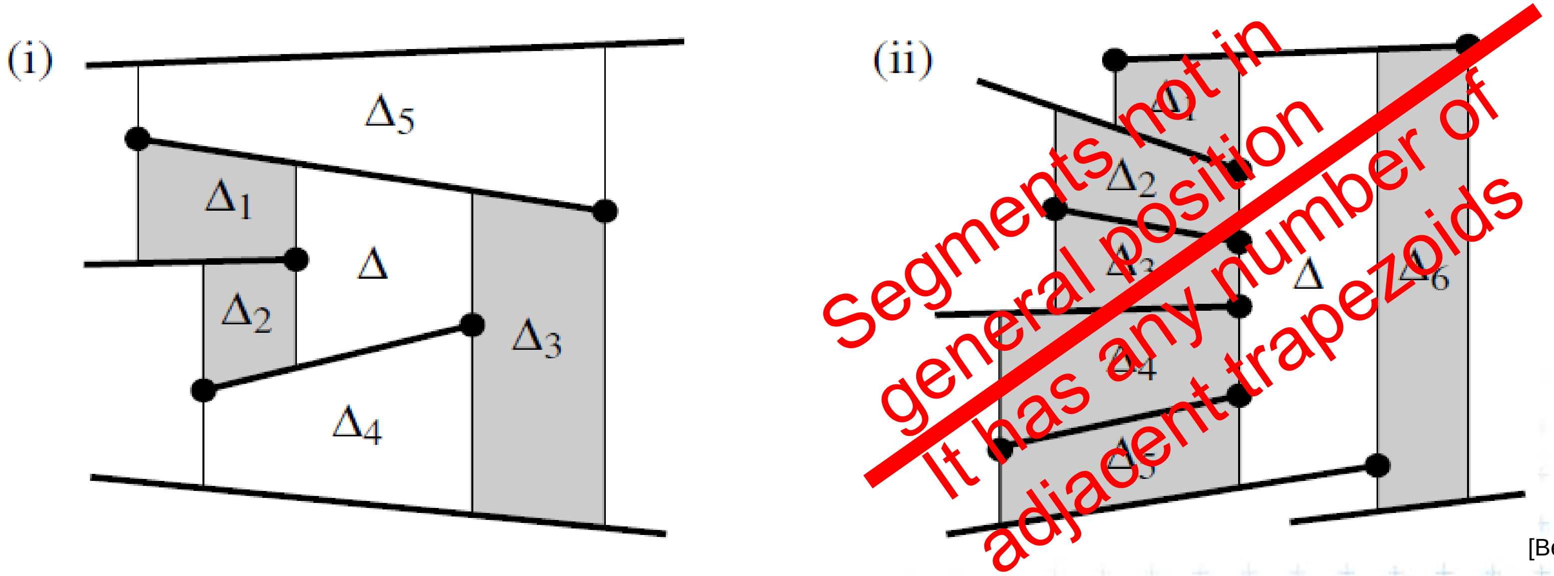
- the left endpoint of *top*() or *bottom*() (a,b,c)
- the right point of a third segment (d)
- the lower left corner of R (e)

Trapezoid Δ

- Trapezoid Δ is uniquely defined by the segments *top*(Δ), *bottom*(Δ)
- And by the endpoints *leftp*(Δ), *rightp*(Δ)

Adjacency of trapezoids segments in general position

- Trapezoids Δ and Δ’ are adjacent, if they meet along a vertical edge



[Berg]

- Δ_1= upper left neighbor of Δ (common *top*(Δ) edge)
- Δ_2 = lower left neighbor of Δ (common *bottom*(Δ))
- Δ_3 is a right neighbor of Δ (common *top*(Δ) & *bottom*(Δ))

Representation of the trapezoidal map *T*

Special trapezoidal map structure $T(S)$ stores:

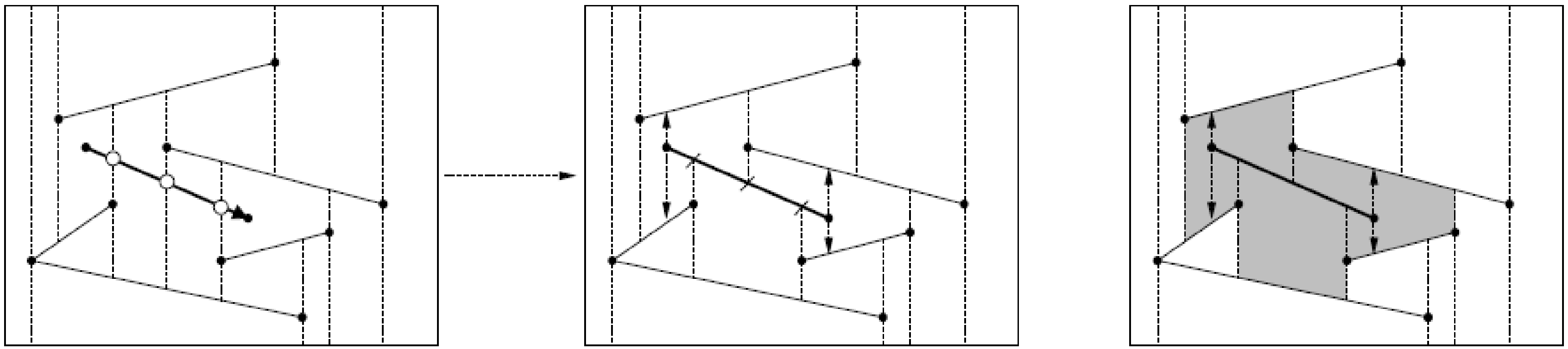
- Records for all line segments and end points
- Records for each trapezoid $\Delta \in T(S)$
  - Definition of Δ - pointers to segments *top*(Δ), *bottom*(Δ),
    - pointers to points *leftp*(Δ), *rightp*(Δ)
  - Pointers to its max four neighboring trapezoids
  - Pointer to the leaf □ in the search structure *D* (see below)

- Does not store the geometry explicitly!
- Geometry of trapezoids is computed in O(1)

Construction of trapezoidal map

- Randomized incremental algorithm
  1. Create the initial bounding rectangle (T_0 =1Δ) … O(n)
  2. Randomize the order of segments in S
  3. for i = 1 to n do
  4. Add segment S_i to trapezoidal map T_i
  5. locate left endpoint of S_i in T_{i-1}
  6. find intersected trapezoids
  7. shoot 4 bullets from endpoints of S_i
  8. trim intersected vertical bullet paths

[Mount]



Locate left endpoint and determine intersections

Shoot new bullet paths and trim intersecting rays

Newly created trapezoids

Trapezoidal map point location

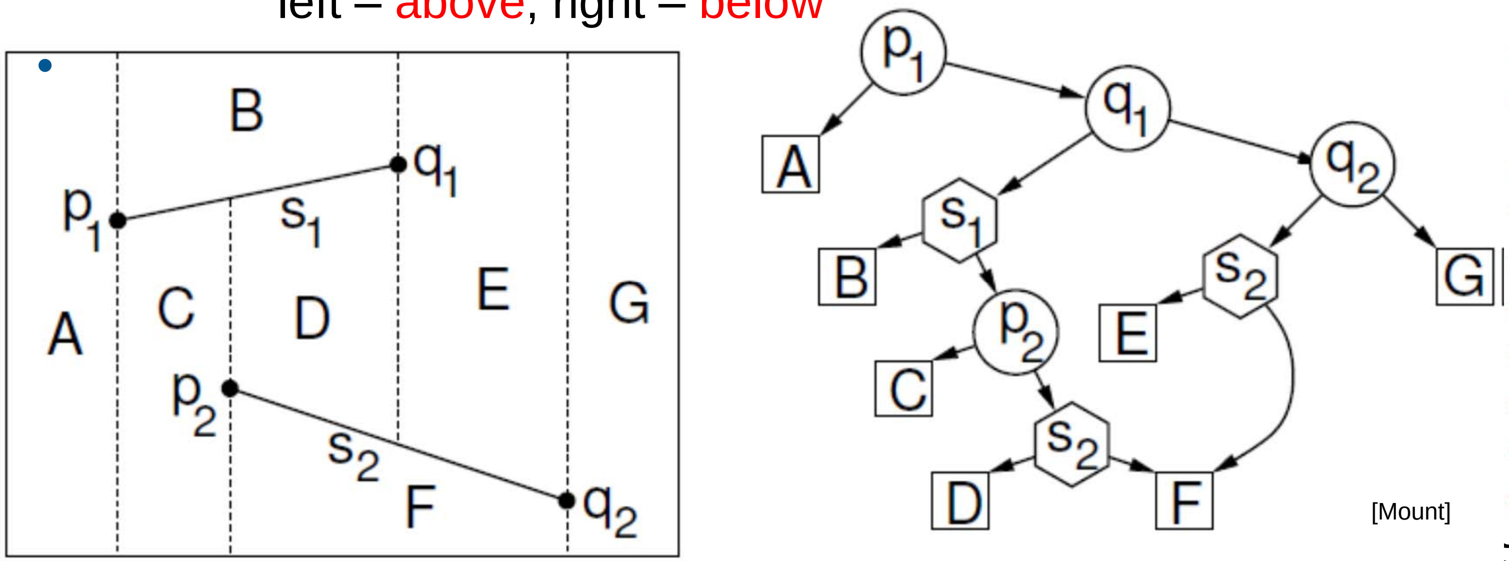
- While creating the trapezoidal map *T* construct the *Point location data structure D*
- Query this data structure

Point location data structure D

- Rooted directed acyclic graph (not a tree!!)
  - Leaves □ – trapezoids, each appears exactly once
  - Internal nodes – 2 outgoing edges, guide the search

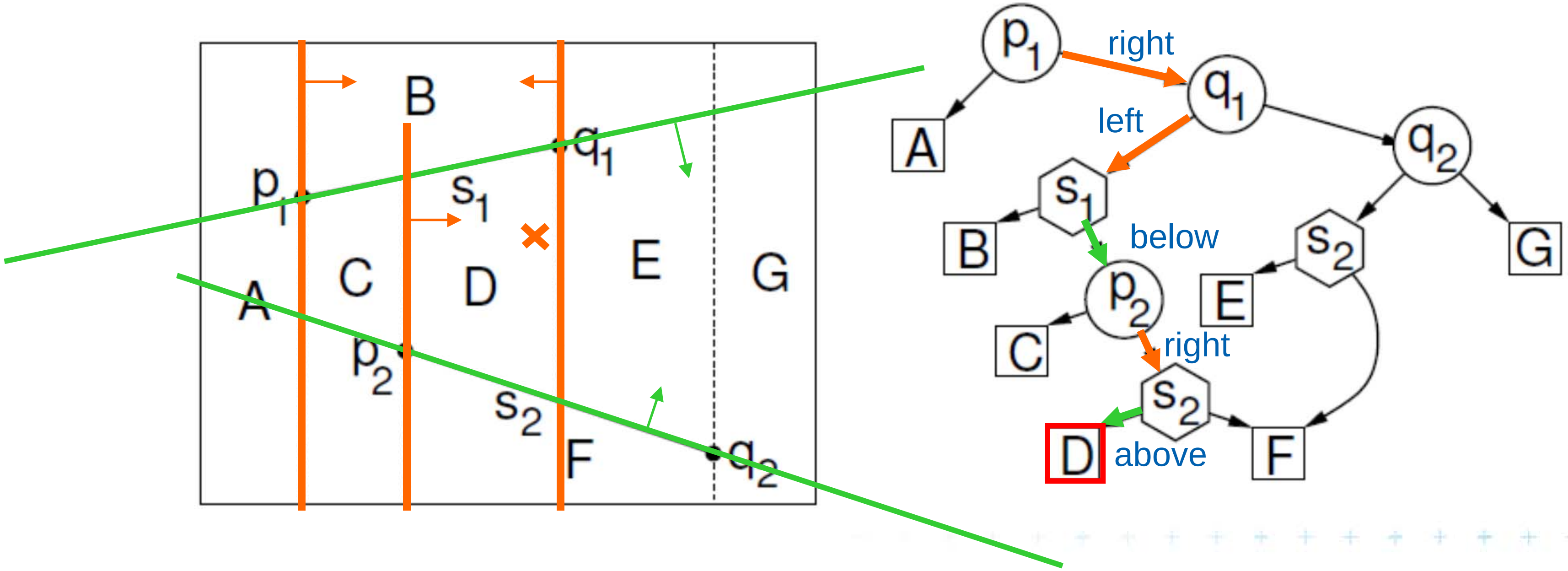
p_1 x-node – x-coord x_0 of segment start- or end-point
left child lies left of vertical line $x=x_0$
right child lies right of vertical line $x=x_0$
– used first to detect the vertical slab

s_1 y-node – pointer to the line segment of the subdivision (not only its y!!!)
left – above, right – below



[Mount]

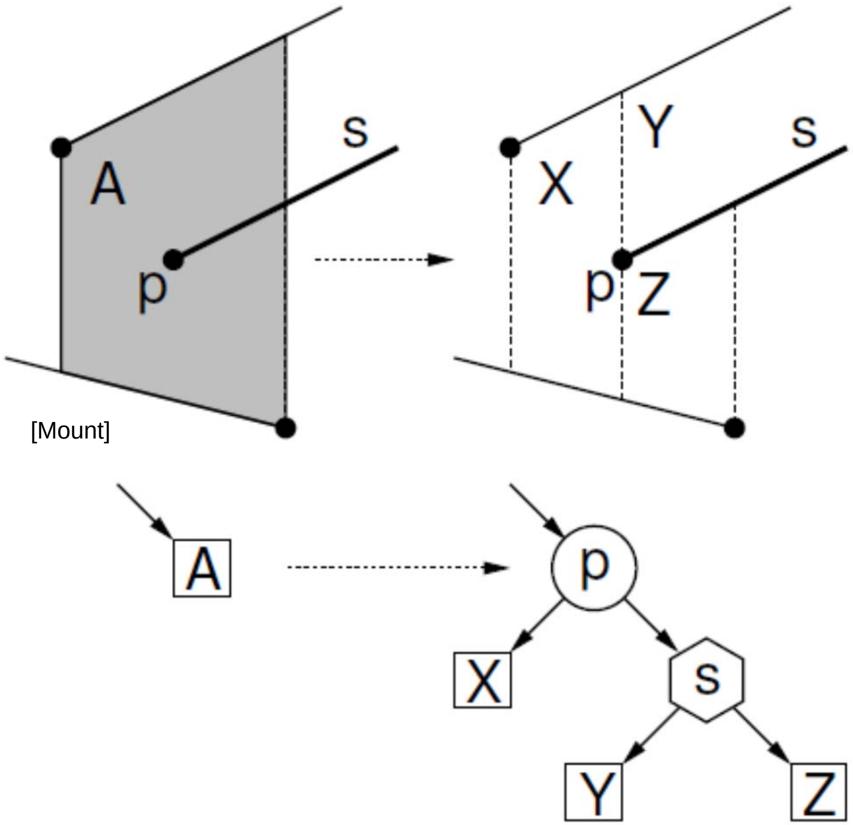
TM search example



[Mount]

Construction – addition of a segment

a) Single (left or right) endpoint - 3 new trapezoids



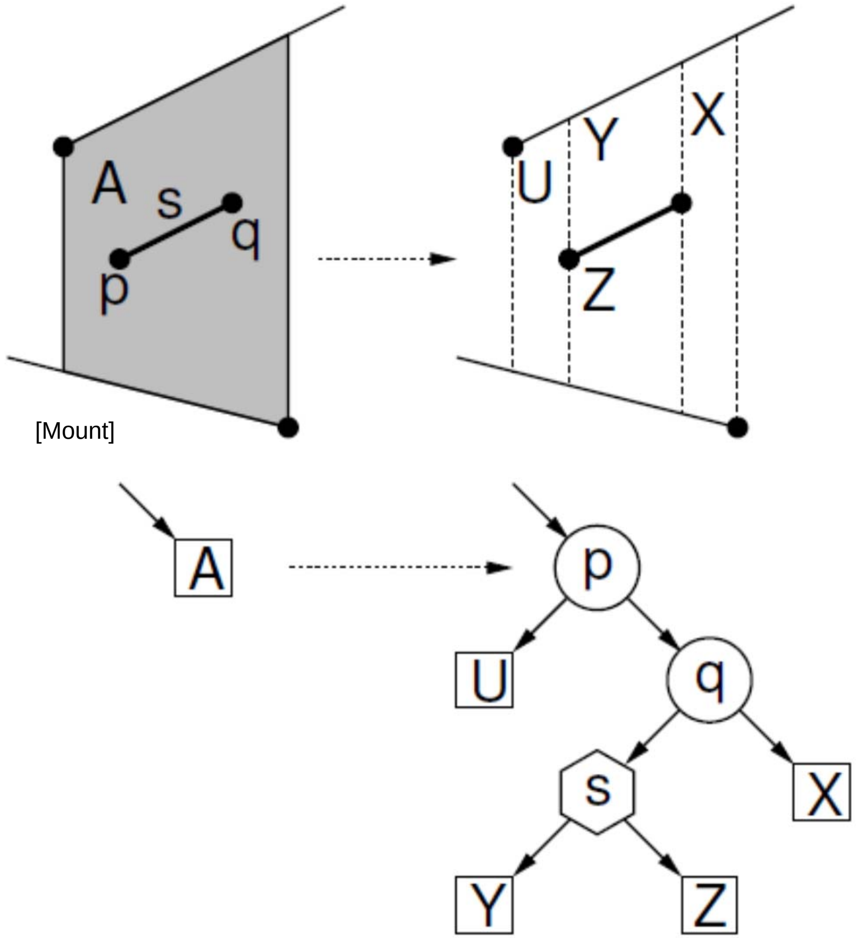
[Mount]

Trapezoid A replaced by

– * x-node for point *p*
– add left leaf for X Δ
– add right subtree
– * y-node for segment *s*
– add left leaf for Y Δ above
– add right leaf Z Δ below

Construction – addition of a segment

b) Two segment endpoints – 4 new trapezoids



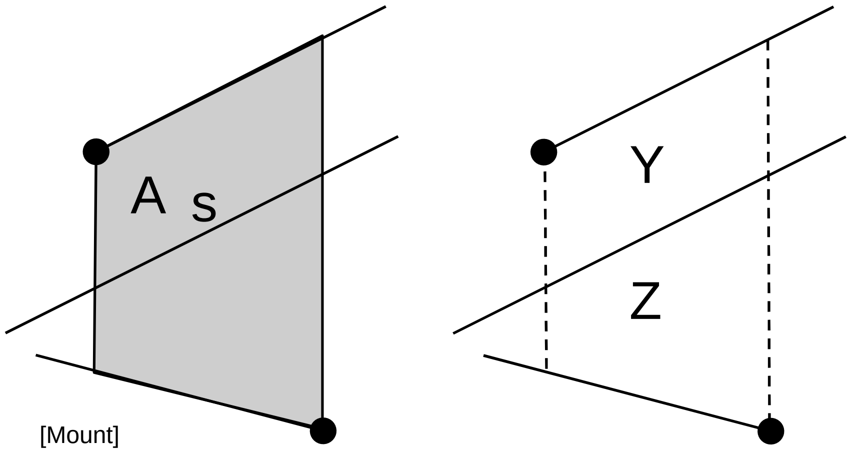
[Mount]

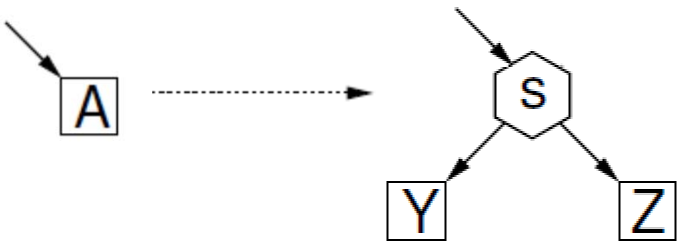
Trapezoid A replaced by

- * x-node for point *p*
- * x-node for point *q*
- * y-node for segment *s*
- add leaves for U, X, Y, Z

Construction – addition of a segment

c) No segment endpoint – create 2 trapezoids

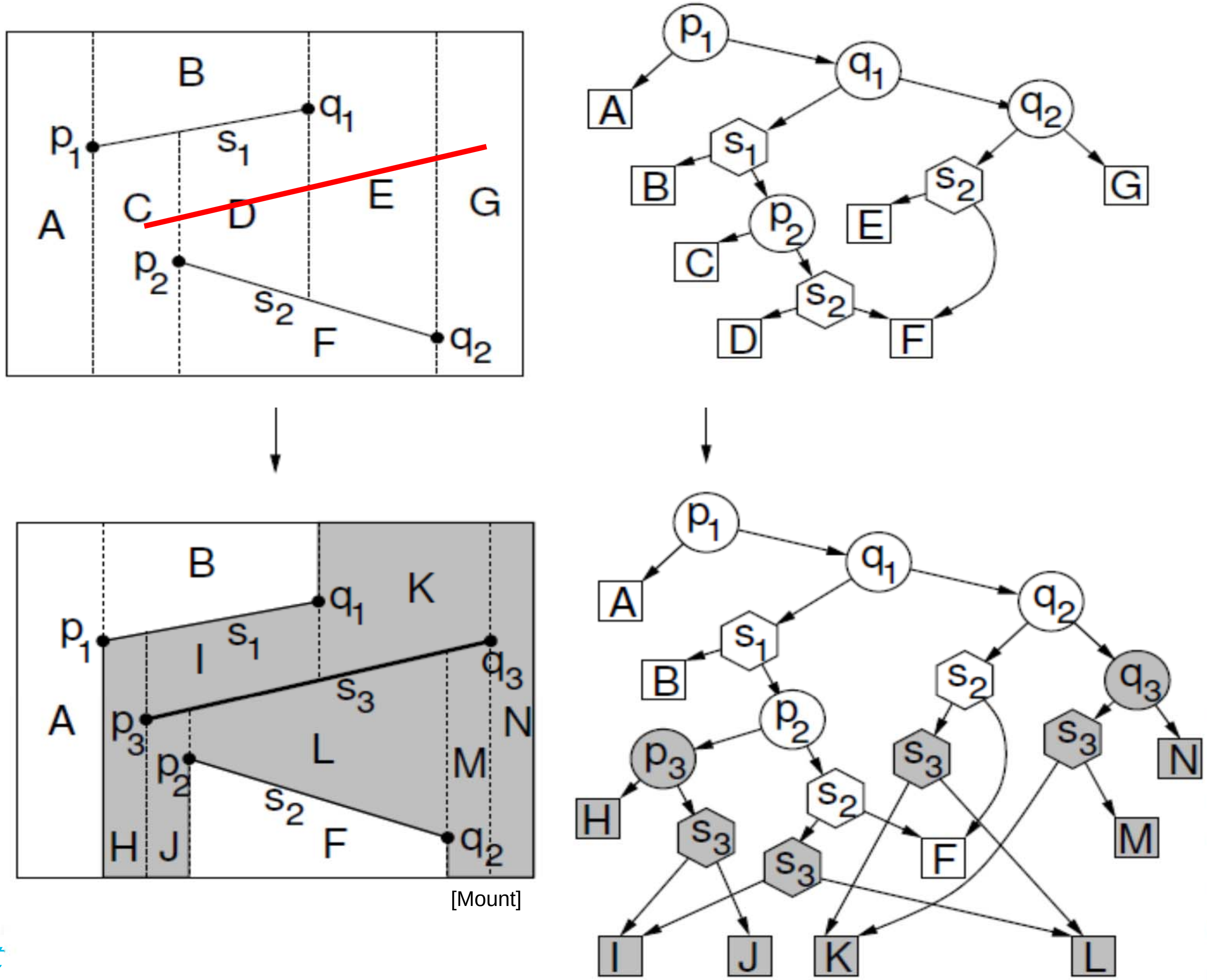




Trapezoid A replaced by

- * y-node for segment *s*
- add leaves for Y, Z

Segment insertion example

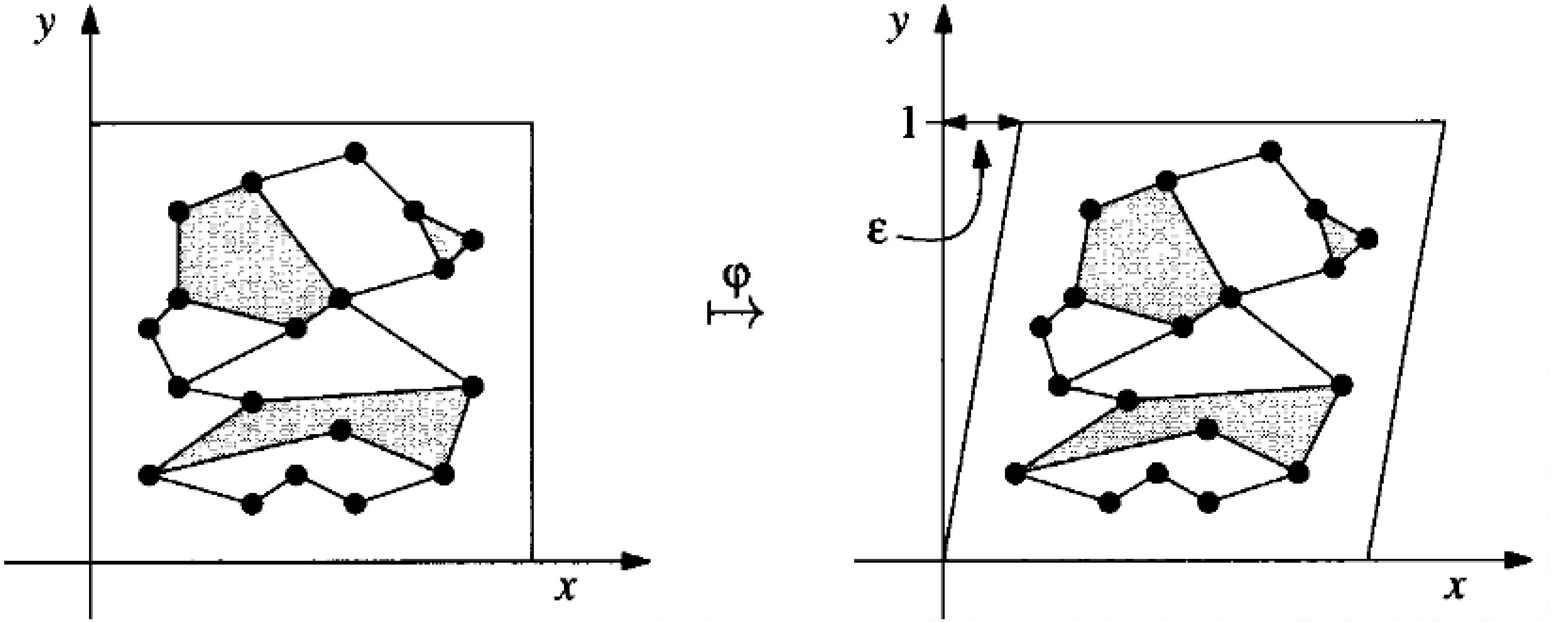


Analysis and proofs

- This holds:
  – Number of newly created Δ for inserted segment:
    k_i = K+4 => O(k_i) = O(1) for K trimmed bullet paths
  – Search point O(log *n*) in average
    => Expected construction O(*n*(1+ *log n)) = O(n log n*)
- For detailed analysis and proofs see
  – [Berg] or [Mount]

Handling of degenerate cases - principle

- No distinct endpoints lie on common vertical line
  - Rotate or shear the coordinates x'=x+εy, y'=y



[Berg]

Handling of degenerate cases - realization

- Trick
  - store original (x,y), we need to perform just 2 operations

1. For two points *p,q* determine if transformed point *q* is to the left, to the right or on vertical line through point *p*
   - If $x_p = x_q$ then compare y_p and y_q (on only for $y_p = y_q$)
   - => use the original coords (x, y) and **lexicographic order**
2. For segment given by two points decide if 3rd point *q* lies above, below or on the segment p_1 p_2
   - Mapping preserves this relation
   - => use the original coords (x, y)

Point location summary

- Slab method [Dobkin and Lipton, 1976]
  – $O(n^2)$ memory $O(\log n)$ time
- Monotone chain tree in planar subdivision [Lee and Preparata,77]
  – $O(n^2)$ memory $O(\log^2 n)$ time
- Layered directed acyclic graph (Layered DAG) in planar subdivision [Chazelle , Guibas, 1986] [Edelsbrunner, Guibas, and Stolfi, 1986]
  – $O(n)$ memory $O(\log n)$ time => optimal algorithm of planar subdivision search (optimal but complex alg. => see elsewhere)
- Trapeziodal map
  – $O(n)$ expected memory $O(\log n)$ expected time
  – $O(n \log n)$ expected preprocessing (simple alg.)

References

- [Berg] Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars: Computational Geometry: *Algorithms and Applications*, Springer-Verlag, 3rd rev. ed. 2008. 386 pages, 370 fig. ISBN: 978-3-540-77973-5 http://www.cs.uu.nl/geobook/
- [Mount] David Mount, - CMSC 754: Computational Geometry, Lecture Notes for Spring 2007, University of Maryland http://www.cs.umd.edu/class/spring2007/cmsc754/lectures.shtml

OI-OPPA. European Social Fund
Prague & EU: We invest in your future.
