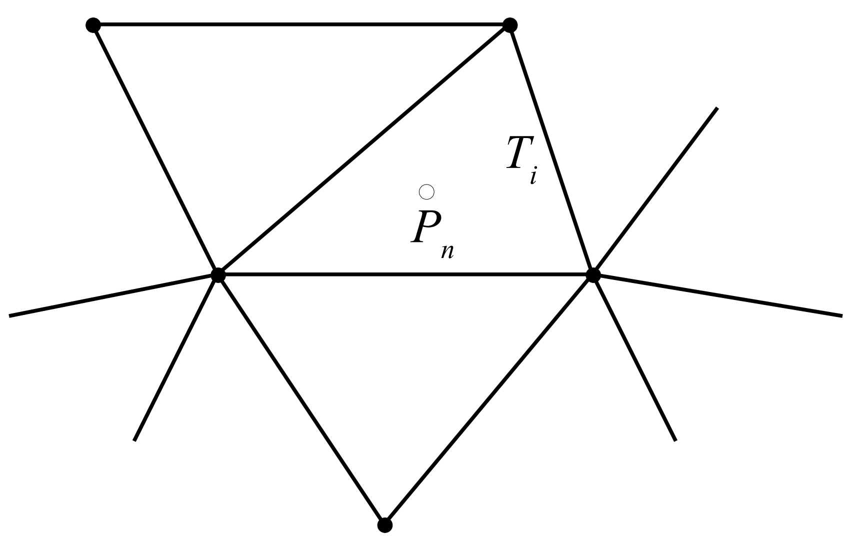
CAD & Computational Geometry

Course plan

- Introduction
- Segment-Segment intersections
- Polygon Triangulation
- Intro to Voronoï Diagrams & Delaunay Triangulations
- Geometric Search
- Sweeping algorithm for Voronoï Diagrams

Geometric Search

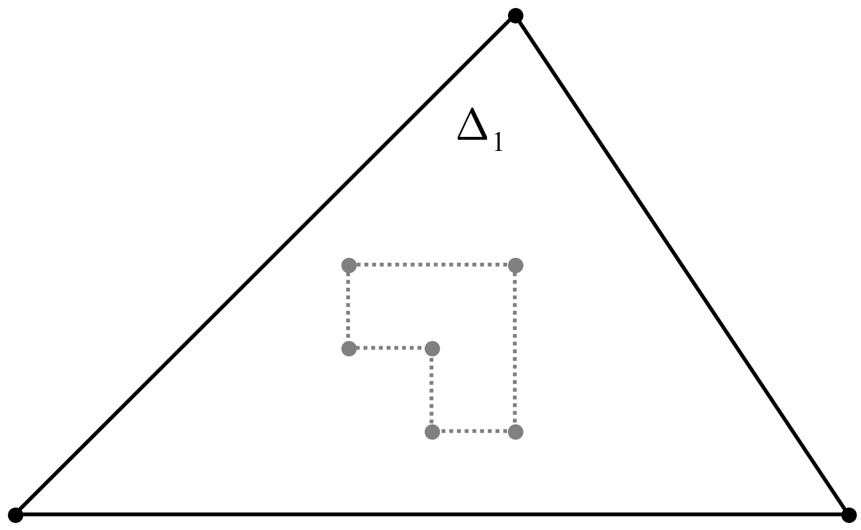
- In the incremental Delaunay triangulation algorithm, it may be necessary to find the triangle T_i (of the triangulation with n-1 points) that contains the next point to insert (p_n).

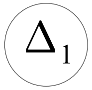


Geometric Search

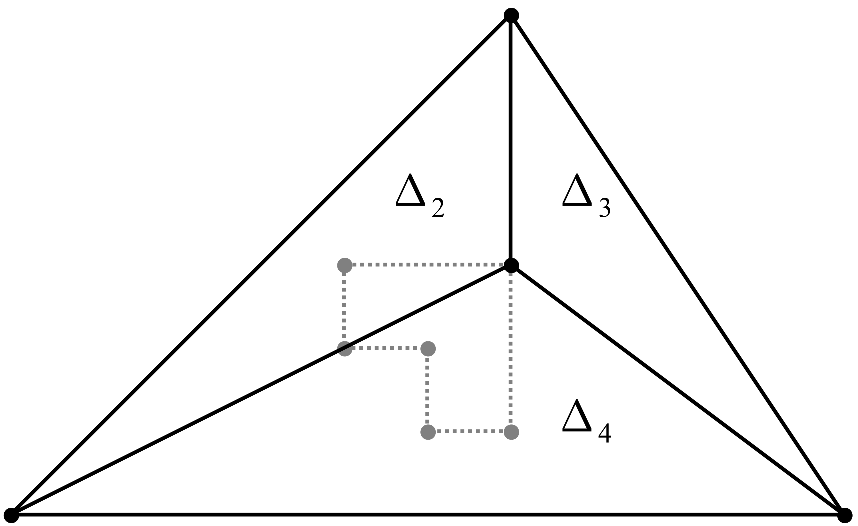
- Some geometric search techniques
  - Case of the Delaunay triangulation – a relatively straightforward solution
  - General case in 2D : the « trapezoidal map »
  - Some other approaches and extensions to 3D: octrees

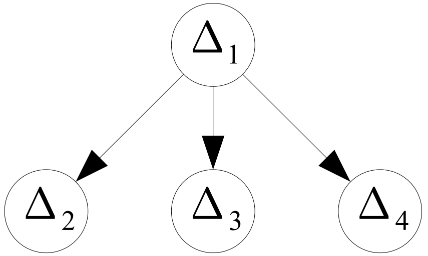
- Case of the Delaunay triangulation
  - One can build a Directed Acyclic Graph while triangulating (DAG)
    - Root : the big triangle that contains all the others vertices



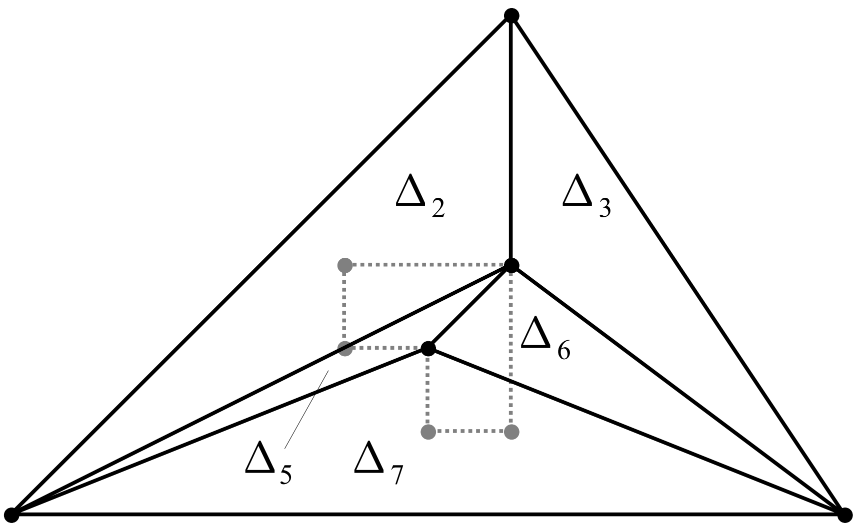


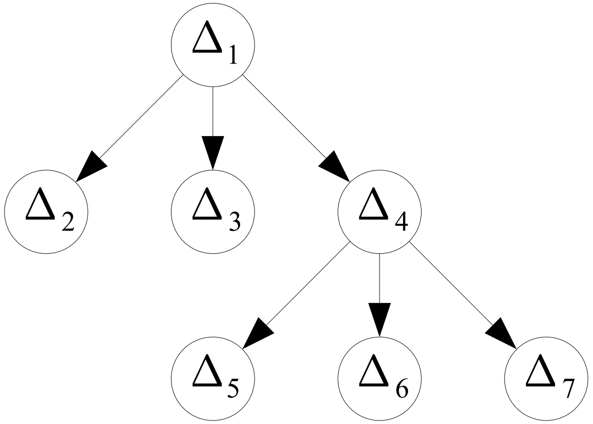
- Case of the Delaunay triangulation
  - Building a directed acyclic graph (DAG)
    - Two types of operations : 1 – insert a vertex (cuts a triangle in 3 or two triangles in 4



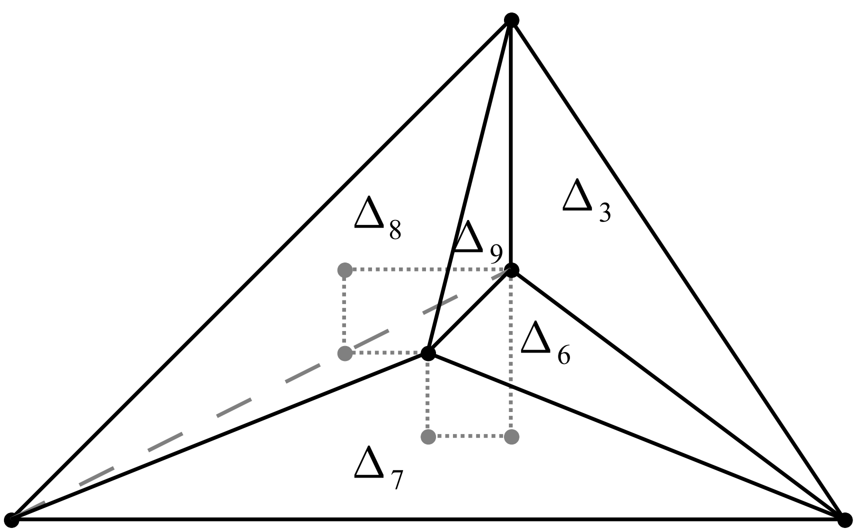


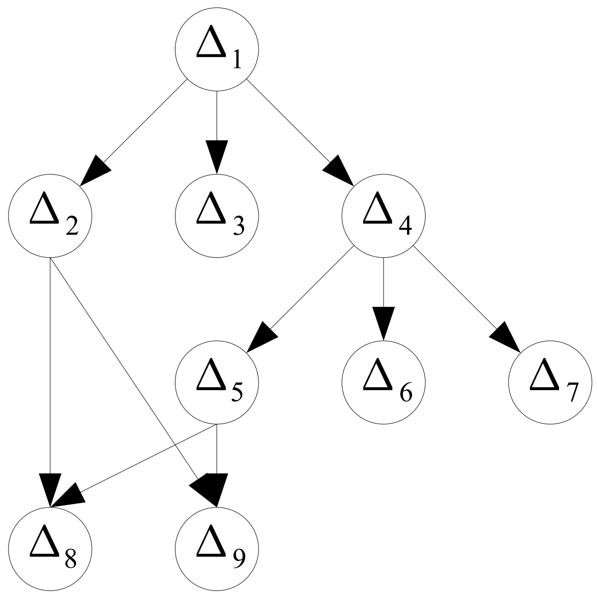
- Case of the Delaunay triangulation
  - Building a directed acyclic graph (DAG)
    - Two types of operations : 1 – insert a vertex (cuts a triangle in 3 or two triangles in 4



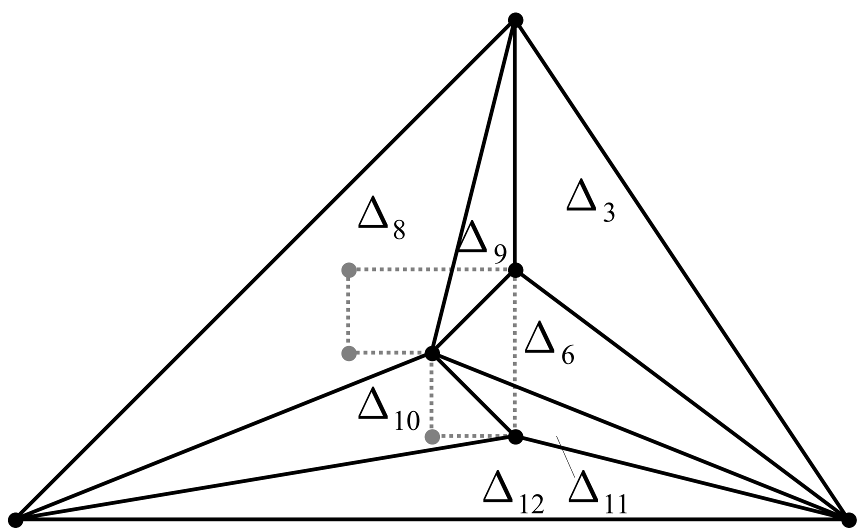


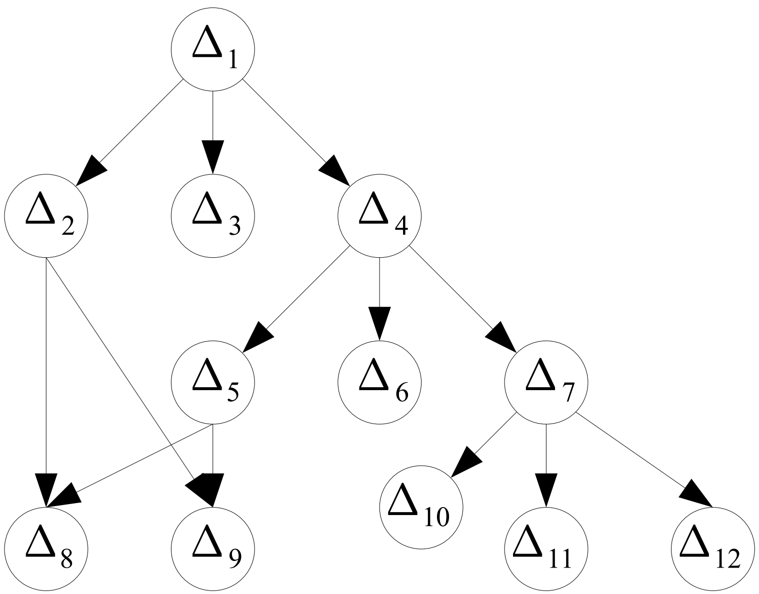
- Case of the Delaunay triangulation
  - Building a directed acyclic graph (DAG)
    - Two types of operations : 2 – edge swapping



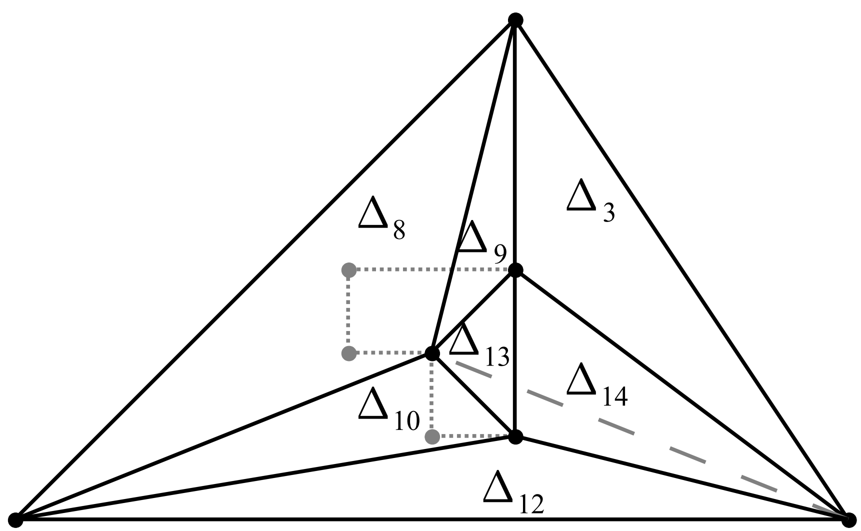


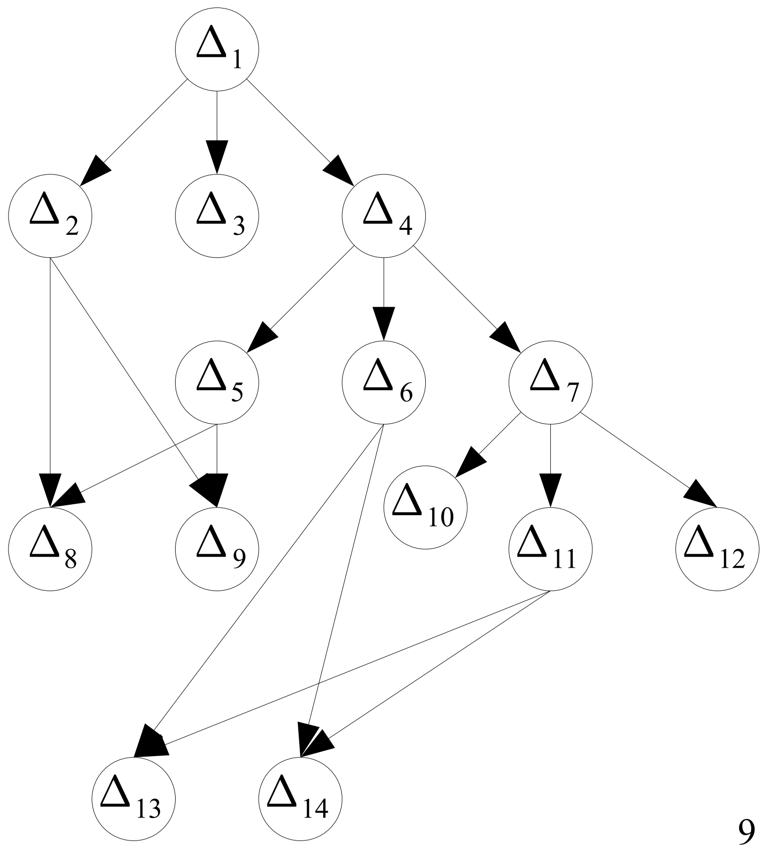
- Case of the Delaunay triangulation
  - Building a directed acyclic graph (DAG)



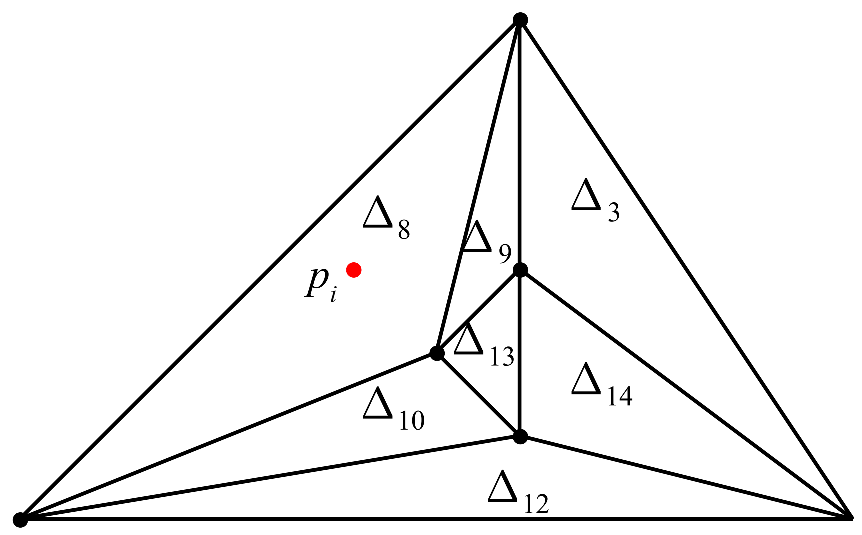


- Case of the Delaunay triangulation
  - Building a directed acyclic graph (DAG)

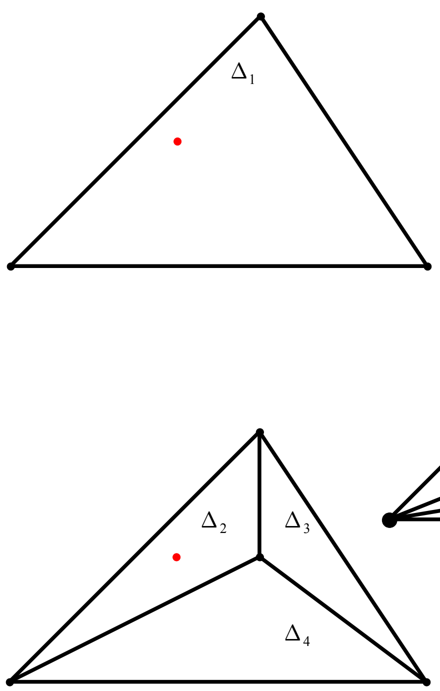


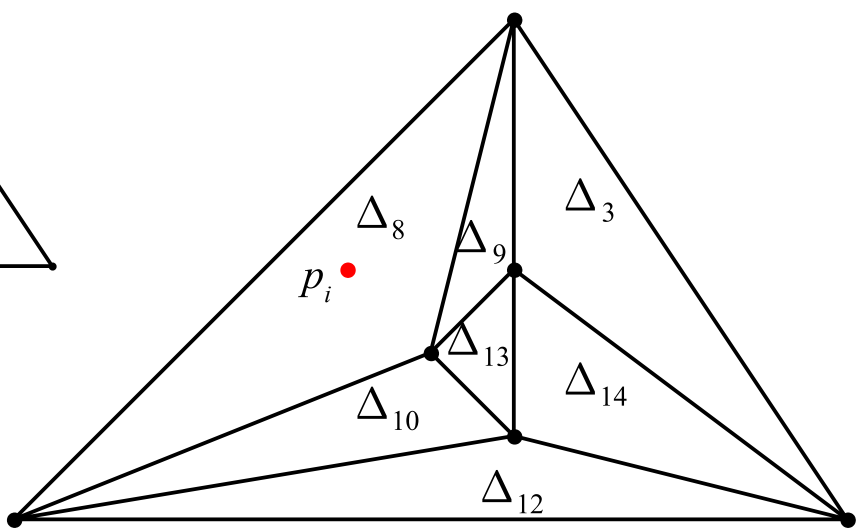


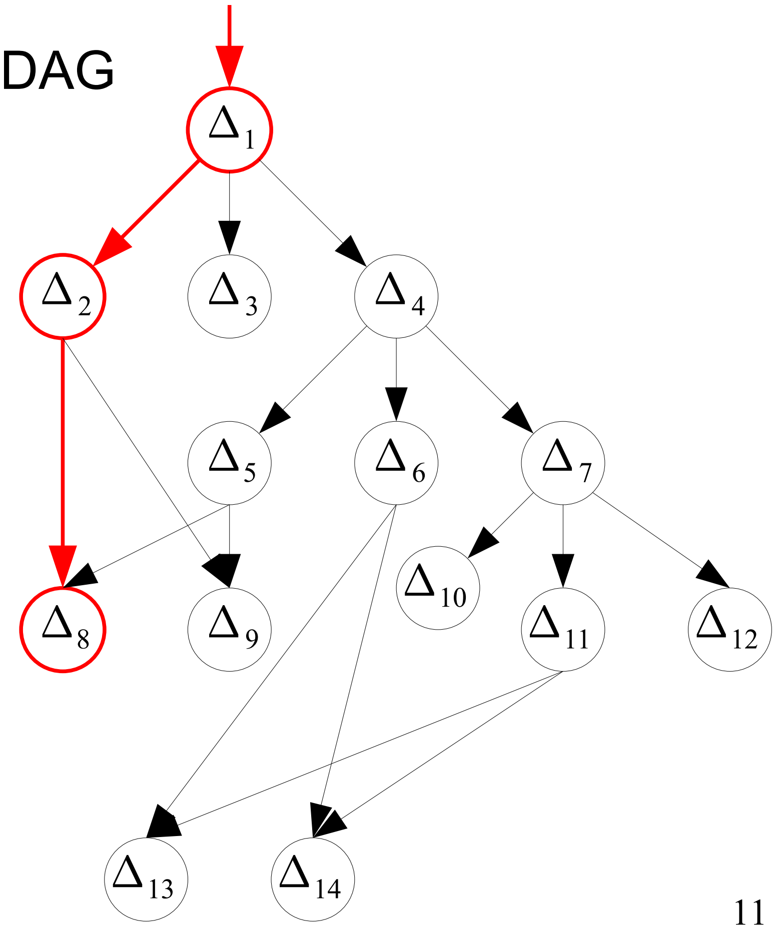
- Case of the Delaunay triangulation
  - Search of a triangle containing a given point (unknown as of now) using the DAG...



- Case of the Delaunay triangulation
  - Search of a triangle containing a given point (unknown as of now) using the DAG...
    - It is sufficient to follow the edges of the DAG







Geometric Search

- Efficiency: the cost of the search is proportional to the number of triangles containing the point (including *all* triangles that have been created since the beginning, and do not exist anymore !)
- How many such triangles (size of the DAG) ?

  At most $9n+1$ (Delaunay triangulation with n vertices)
- Proof

Geometric Search

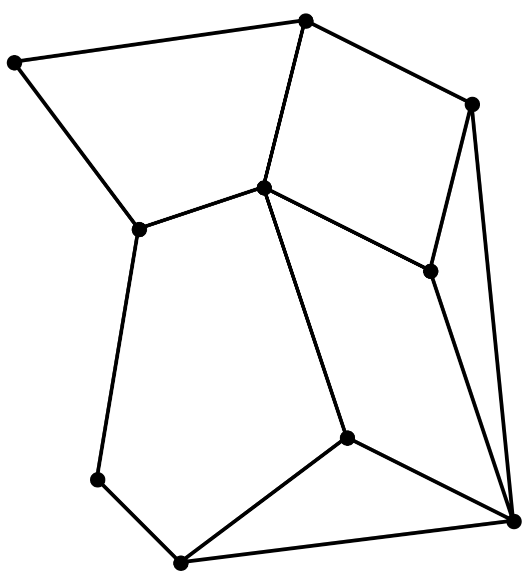
- Proof that at most $9n+1$ triangles are generated
- A step r : one inserts the point $p_r \in P$ into the current triangulation
  - Cut 1 or 2 triangles to form 3 or 4 new triangles
  - Edge legalization → 2 more new triangles at each time
  - If a vertex p_r has a valence or degree equal to k, then one creates at most $2(k\text{-}3) + 3 = 2k\text{-}3$ new triangles when inserting it. What is this valence for every possible permutation of the P_r (set of points inserted up to now) ?
    - Such a triangulation has at most $3(r+3)-6$ edges (cf Euler relations, less. 8, p. 15)
    - Three of these are from the initial triangle, thus the sum of all degrees in P_r is less than $2(3(r+3)-9)=6r$
    - The mean value of the valence of p_r is therefore 6.
    - The linearity of the expected value (mean value) allows us to say that, in average, at most $2{\cdot}6-3\ =9$ triangles are created at step r .
    - In total, $9n+1$ triangles counting the initial triangle.

Geometric Search

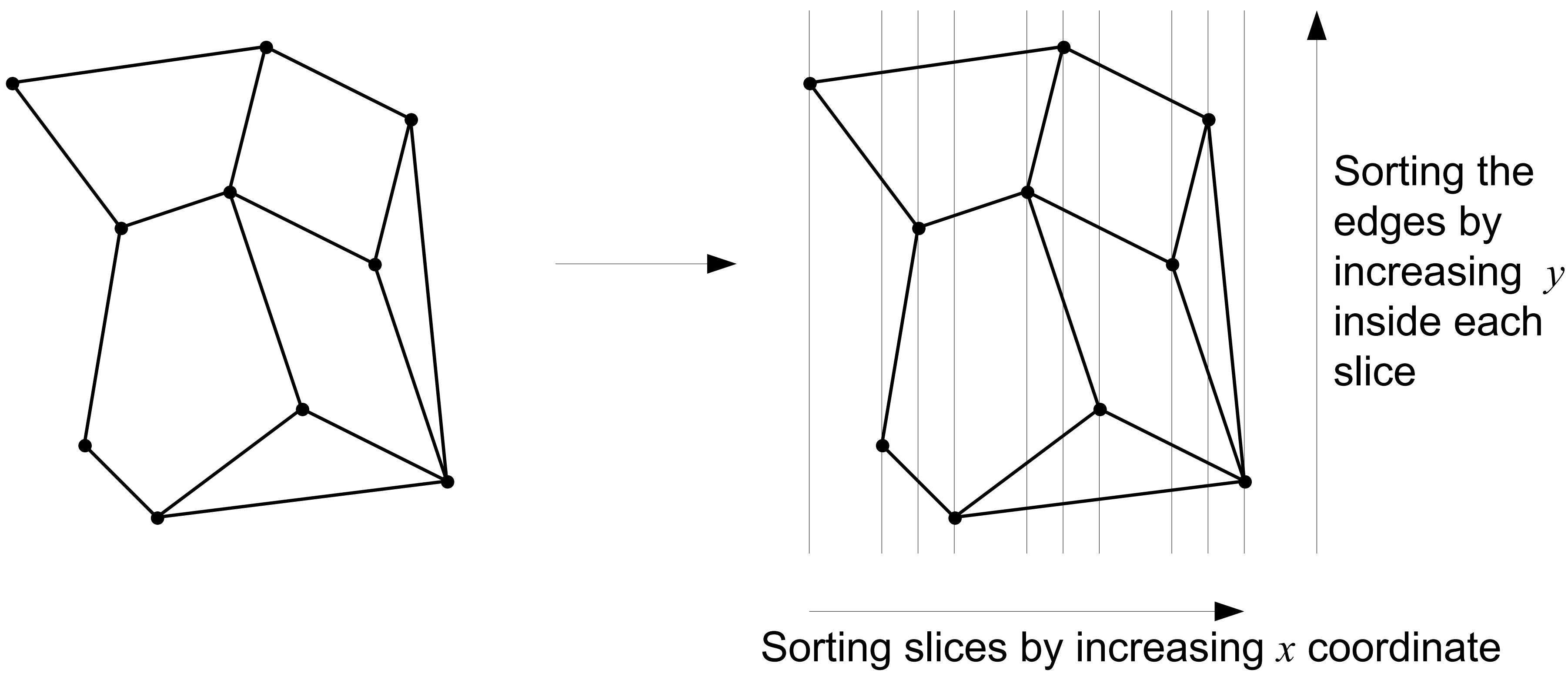
- How many triangles actually contain the point ?
  - It depends on the sequence of the operations made until now (the algorithm must therefore be **randomized**)
  - A statistical analysis in the case of **Lawson's algorithm** shows a complexity of $O(\log n)$ *in average* for this search.
  - It is not possible to generalize this for any triangulation with n points !
  - Lengthy proof : cf book p. 206
- The global complexity is therefore $n\log n$.

  (what about mesh generation, i.e. the insertion of a vertex for which the position is known ?

- General 2D case: the « trapezoidal map »
  - In any polygonal paving of the plane, allows to find, in logarithmic time, the polygon containing any given point.

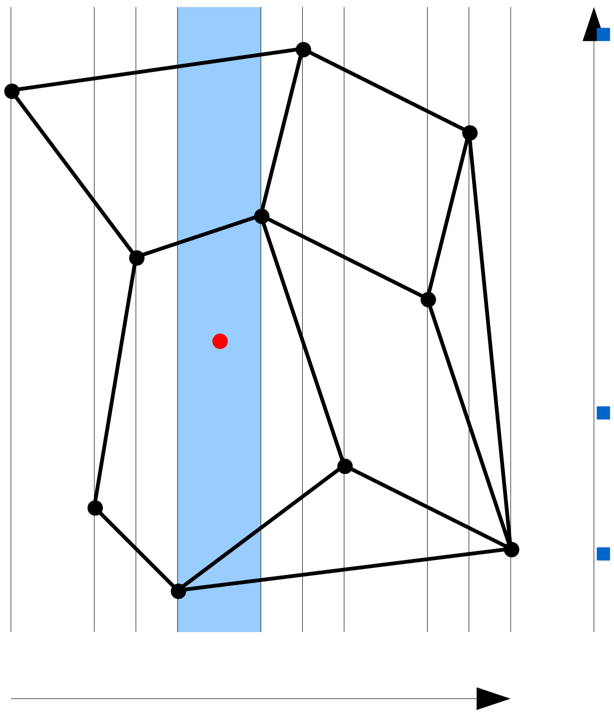


- General 2D case: the « trapezoidal map »
  - A trivial partitioning: a vertical line goes through every vertex of the paving, slicing it with additional edges which never intersects each other

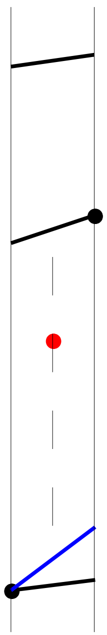


- Trivial partitioning: Looking for a polygon that contains a given point

    1 – Find the slice containing the point (in $\log n$) with a binary search, the number of slices being at most $2n$ (n = number of edges)

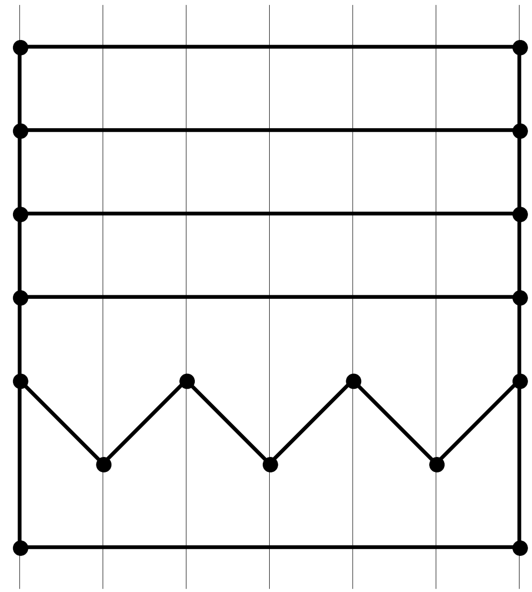


- 2- Find which edge (or half-edge) immediately below the point. It points to the adequate polygon. If there is no such polygon, then the point is outside the structure. In may also be done in $O(\log n)$ (at most, there are n edges in the slice)
- Globally, the search is in $O(\log n)$, which is perfectly fine...
- But what about the time spent to build the search structure, and the memory footprint ?



- Trivial partitioning : Memory footprint and time spent
  - Worst case : in $O(n^2)$ …
  - Usual case : in $O(n\sqrt{n})$

    In both cases, the setup of the search structure takes at least $O(n^2 \log n)$ or $O(n\sqrt{n}\log n)$.
  - In fact, this partitioning is a refinement of the paving for which the complexity worse than the complexity of the initial paving
  - One needs to find a better partitioning for which the complexity is of the same order $\mathrm{O}(n)$

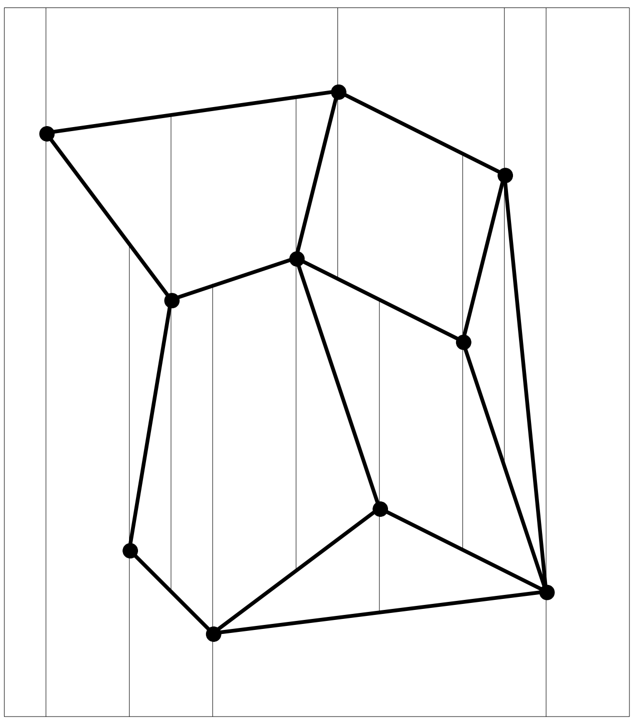


$n/4$ edges in each slice

$n/4$ slices

Geometric Search

- A “cheaper” partitioning
  - Draw vertical lines from each vertex until one meets the edges of the paving (or the edges of the bounding box)



- Properties of the cells then obtained:

  - they are convex

  - they have 1 or 2 vertical sides, and exactly two non vertical sides

  - they are either trapezoids or triangles
- We will first admit that the points are in general positions (in particular, the x coordinates are all distinct)

Geometric Search

- There are 5 different configurations for the **left** side of a trapezoid

  (a) Those with a left vertical side limited to a point

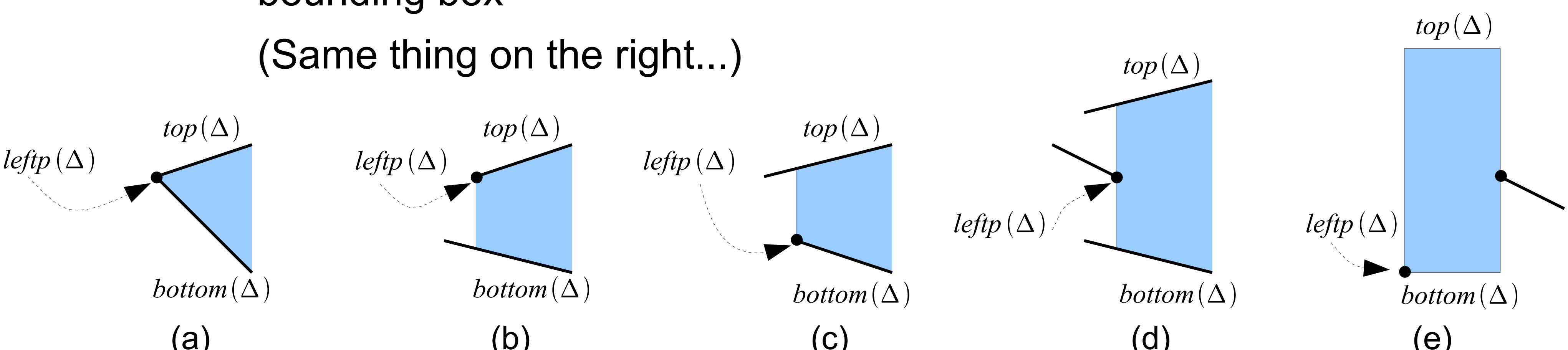
  (b) Those where the left vertical side as a prolongation of the upper edge

  (c) Those where the left vertical side as a prolongation of the bottom edge

  (d) Those where the left vertical side as a prolongation of the right vertex of another edge

  (e) The only trapezoid for which the left side is a boundary of the bounding box

  (Same thing on the right...)



Geometric Search

- There are 5 different configurations for the **right** side of a trapezoid

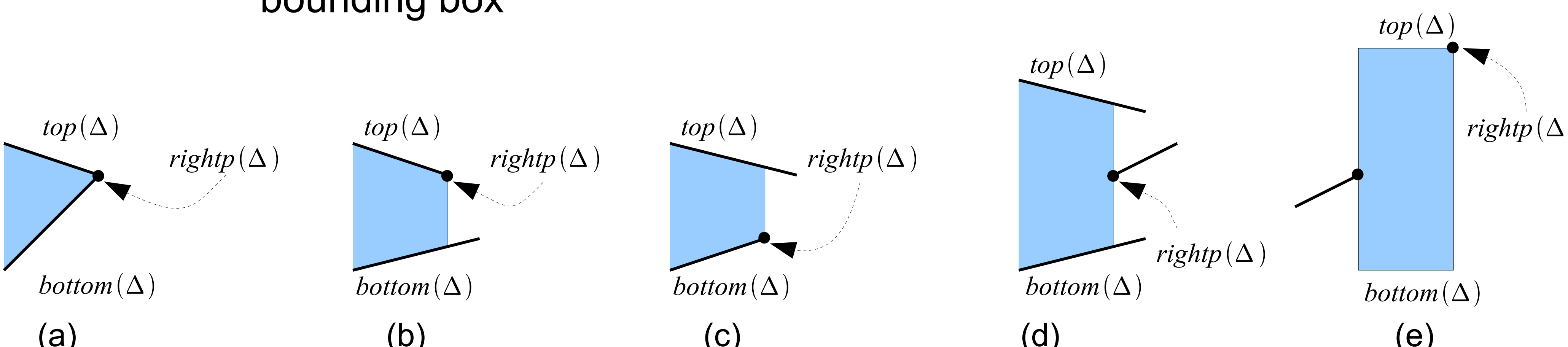
  (a) Those with a right vertical side limited to a point

  (b) Those where the right vertical side as a prolongation of the upper edge

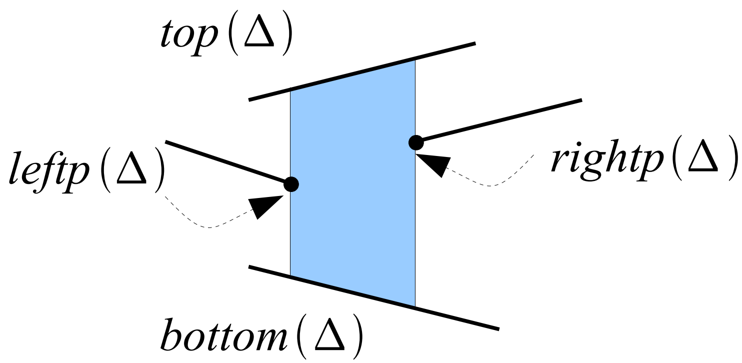
  (c) Those where the right vertical side as a prolongation of the bottom edge

  (d) Those where the right vertical side as a prolongation of the left vertex of another edge

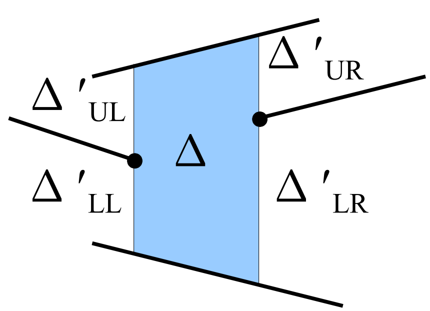
  (e) The only trapezoid for which the right side is a boundary of the bounding box

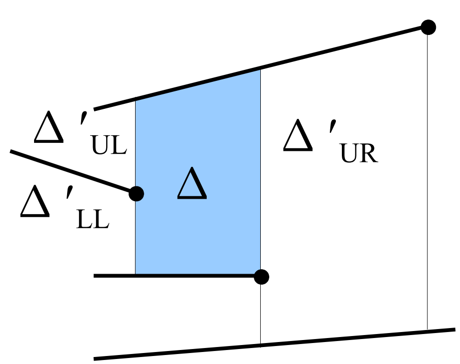


- Each trapezoid is defined by the 4 following entities, whatever the global configuration (23 different configurations all-in-all)



- One defines the neighborhood as follows : the trapezoids Δ and Δ' are adjacent if they share a vertical edge.
  - As the set of edges is in general position, there are at most 4 neighbors. For each neighbor, *top* or *bottom* is shared with Δ





- What is the complexity of this decomposition ?

  It has at most $6n+4$ vertices and $3n+1$ trapezoids.

  Proof :

  - Case of the vertices

  One vertex of the decomposition is either :

  - a vertex of the bounding box (max 4)

  - a vertex belonging to one of the edges of the initial paving (max $2n$)

  - the intersection of an edge (or the bounding box) with the vertical edges coming from extremities of the n initial edges. As there are obviously only 2 prolongations per extremity, one going up and one going down, we have at most $2(2n)$ such vertices.

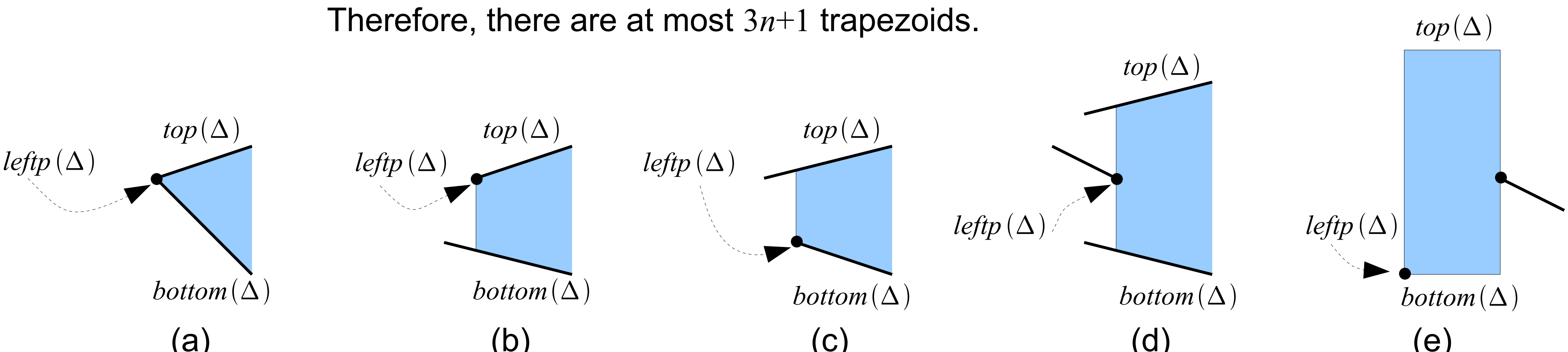
  - Globally, $4+2n+4n = 6n+4$ vertices is a maximum.

Geometric Search

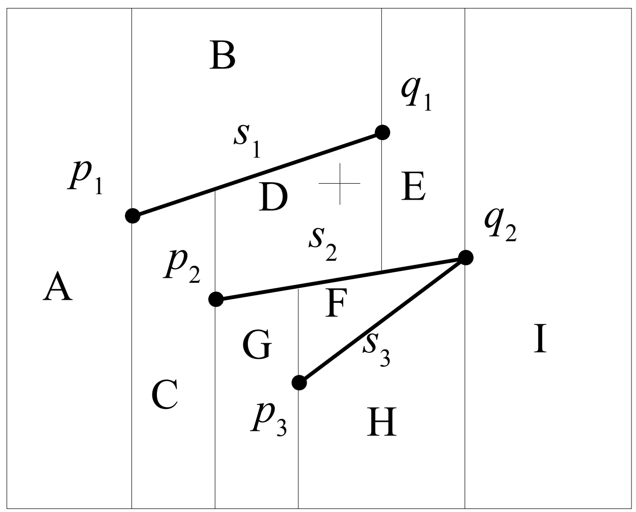
- Case of the trapezoids : each trapezoid has one point $leftp$. This point is either and extremity of the n segments ; or the lower right corner of the bounding box. By analyzing the 5 configurations for $leftp$, we get :

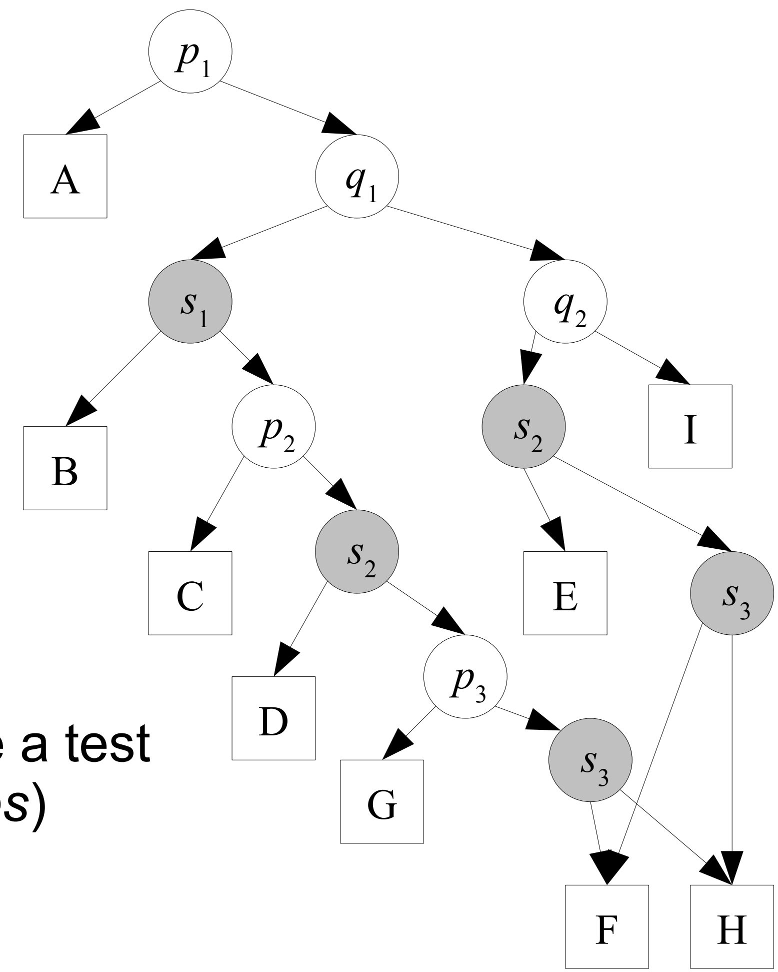
  - Case (e) cannot happen for more than one trapezoid, (1 trapezoid)
  - For a given right extremity of an initial edge, cas (d) may happen only for one trapezoid, (n trapezoids)
  - For a given left extremity of an initial edge, cases (b) and (c) apply, therefore at most for 2 trapezoids. ($2n$ trapezoids)
  - For case (a), one may consider that $leftp$ is the left extremity of $bottom$. Hence, the left vertex of an endge s may be $leftp$ of only two trapezoids, one above and one below. One therefore count some trapezoids multiple times when assimilating (a) into cases (b) or (c)...

  Therefore, there are at most $3n+1$ trapezoids.



- Search data structure in the trapezoidal map





- The white circles are nodes, where a test is made on the x coordinate (*x-nodes*)
- The grayed circles are nodes where a test is made on the y coordinate (*y-nodes*)
- Squares are leaves (the trapezoids)
- The structure is not unique !

Geometric Search

- The incremental construction of the trapezoidal map
  - The trapezoidal map is unique. However, the search structure associated to the map is not unique, it depends in which order new edges are inserted in the structure.
  - The global algorithm therefore have to be randomized, and will update, at the same time, the search structure and the map.
  - At the end of each step i, the map and the search structure is valid, coherent and allows the insertion of a new edge at step $i+1$. (let us call this the *invariant* of the incremental algorithm)

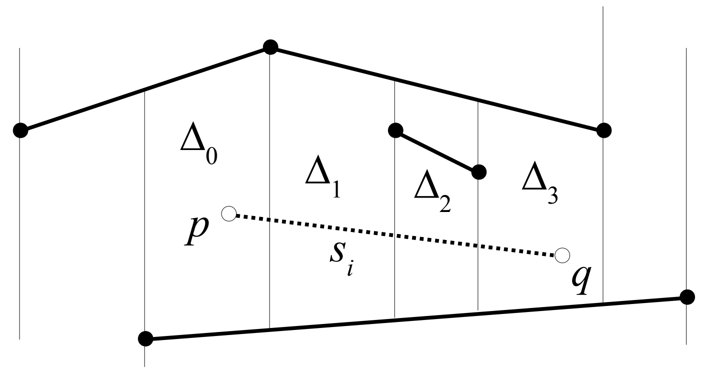
- General algorithm

TrapezoidalMap(S)
Input : a set S of n edges in the plane (non intersecting)
Output : a trapezoidal map $T(S)$ and a search structure R (in a bounding box B)
{
Compute the bounding box B that contains S, and initialize the map T and the search structure R.
Compute an arbitrary random permutation s_i of the edges in S.
For i from 1 to n
{
Find the set of trapezoids $\Delta_0 \dots \Delta_k$ intersecting s_i
Delete $\Delta_0 \dots \Delta_\mathrm{k}$ from T and replace by the new trapezoids that appear because of s_i.
Remove the leaves that correspond to $\Delta_0 \dots \Delta_k$ in R, and create new leaves for the new trapezoids.
Link the new leaves to the internal nodes R and create new comparison nodes
}
}

- Initialization

- Insertion of an edge s_i
  - First we need to find the ordered list of trapezoids intersected by the edge.

    One may notice that the list contains only neighboring trapezoids (in the meaning given previously), therefore, Δ_{j+1} is a neighbor of Δ_j.
  - More precisely,if $rightp(\Delta_j)$ is above s_i , then Δ_{j+1} is the neighboring lower right trapezoid (Δ'_{LR}), otherwise it is the upper right neighboring trapezoid (Δ'_{UR})
  - It is therefore only necessary to determine Δ_0 (starting point p) and loop over neighbors until q is on the right to or equal to $rightp(\Delta_j)$



- Algorithm used to look for the set of intersected trapezoids

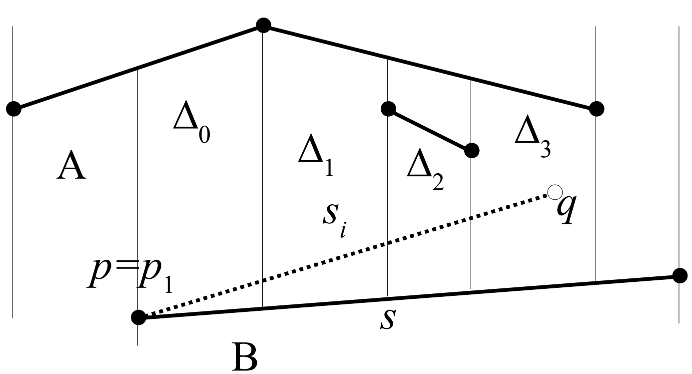
SearchTrapezoids(*T*,*R*,*si*)
Input : the map *T*, search structure *R* on *T*, the segment to intersect with
Output : A sorted list of the trapezoids intersected by s_i

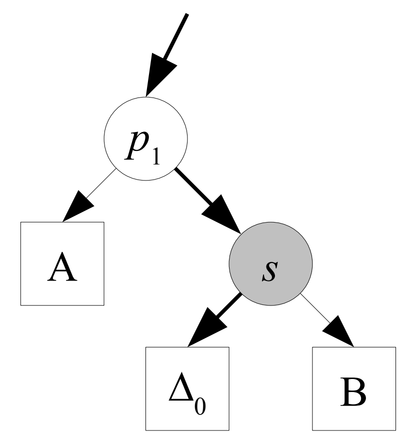
```
{
 Let p and q the left and right extremities of s_i
 Lookout  p in R to get Δ_0
 j=0
 While q is on the right or equal to rightp(Δ_j)
 {
  If rightp() is above s_i
   Δ_j is the lower right neighbor (Δ'_LR)
  Else
   Δ_j is the upper right neighbor (Δ'_UR)
  j=j+1
 }
 Return  Δ_0 … Δ_j
}
```

Geometric Search

- Algorithm used to look for the set of intersected trapezoids (follow-up)

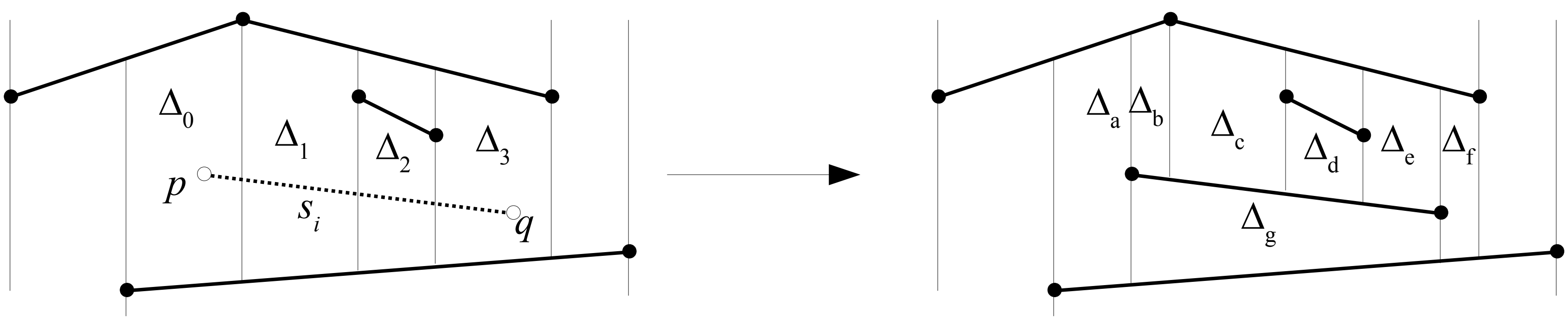
  Special case if p belongs to the points already inserted in T : In this case, at some point during the search of Δ_0, p will be exactly on the vertical line over an x*-node*, and we have to consider it is in fact slightly on the right – and continue the search for Δ_0. It amounts to consider that points located exactly over an x*-node* are in fact on the right. Thus, Δ_0 corresponds to the first trapezoid on the right of p (which is cut by s_i). The same idea applies if p is exactly on a y*-node*, the slopes must be compared to decide on which side to “go”.





- Insertion of an edge s_i
  - Modification of T

    Delete the trapezoids from T, and replace them by new ones.
  - How many ?



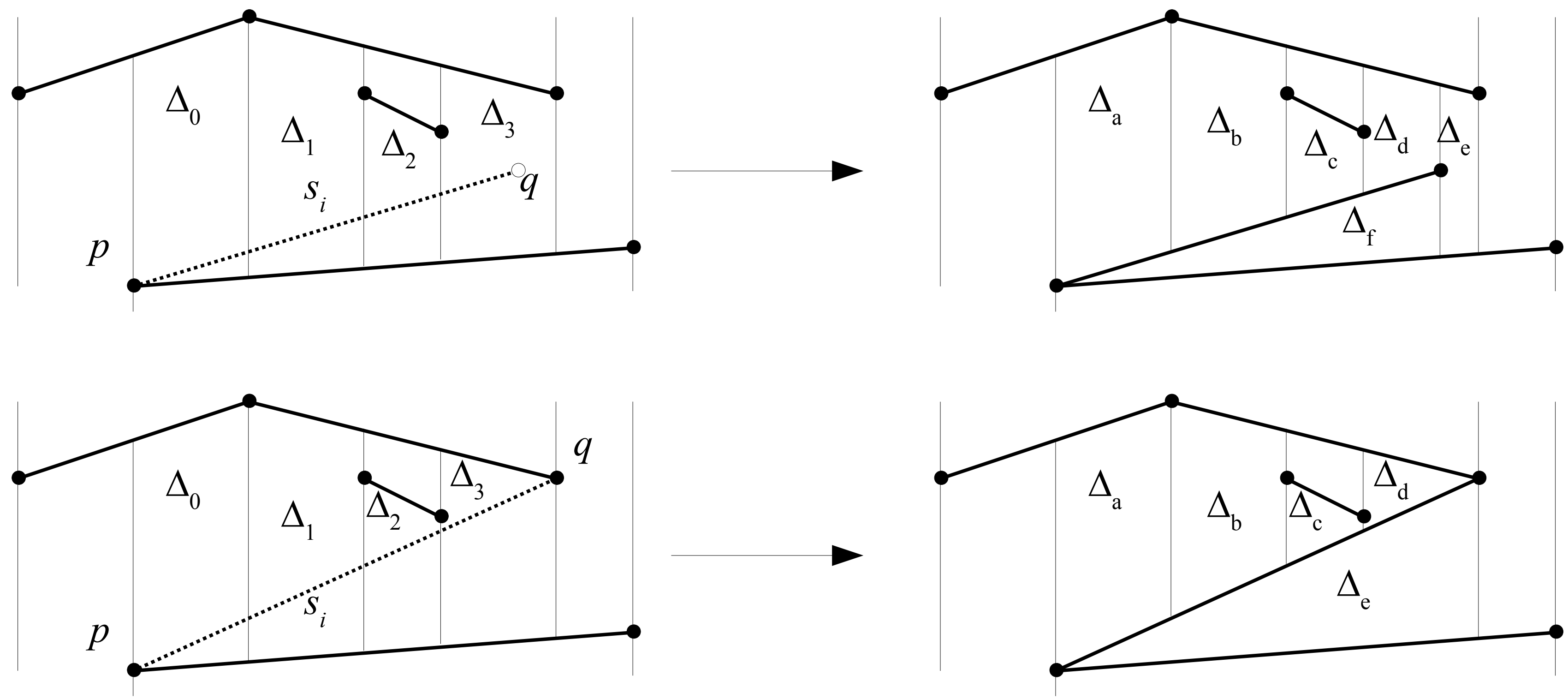
- There is a set of trapezoids for which s_i is the inferior segment (*bottom*) and for which the *leftp* and *rightp* vertices are located above s_i

- There is another set for which s_i is the superior segment (*top*) and for which *leftp* and *rightp* are located below s_i

- Then there is a pair of trapezoids to the left and right when the segment is composed of vertices not already part of T.

- Modification of T

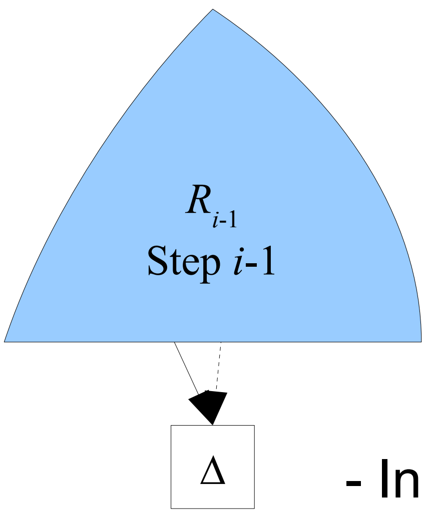
  - All these operations take a time proportional to the number of trapezoid crossed by the edge s_i.

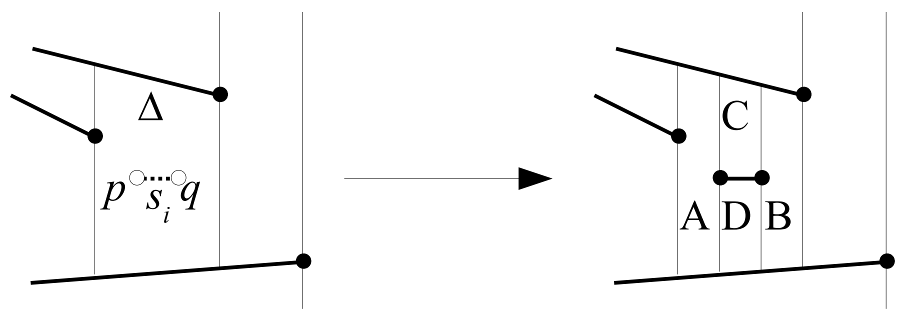


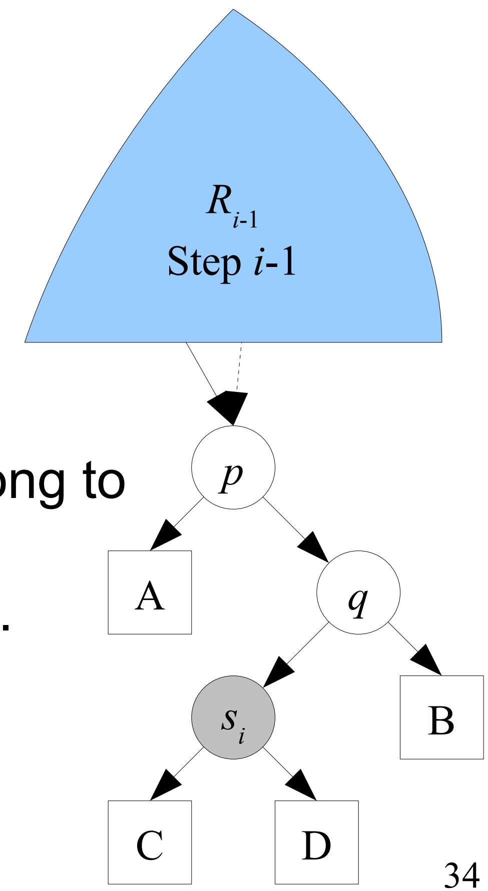
- Update of the search structure R

  The leaves corresponding to trapezoids crossed by s_i must be updated.

  - Simple example with one trapezoid :



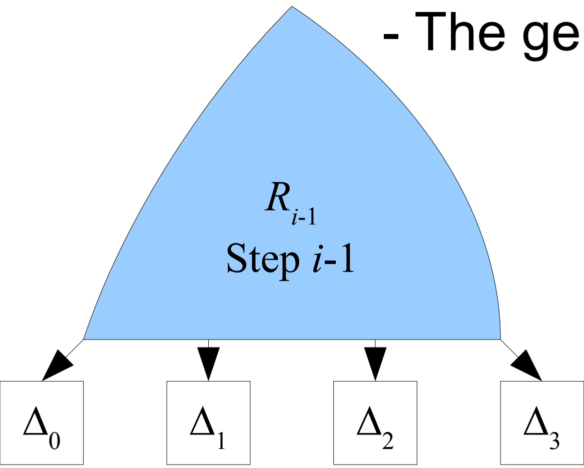


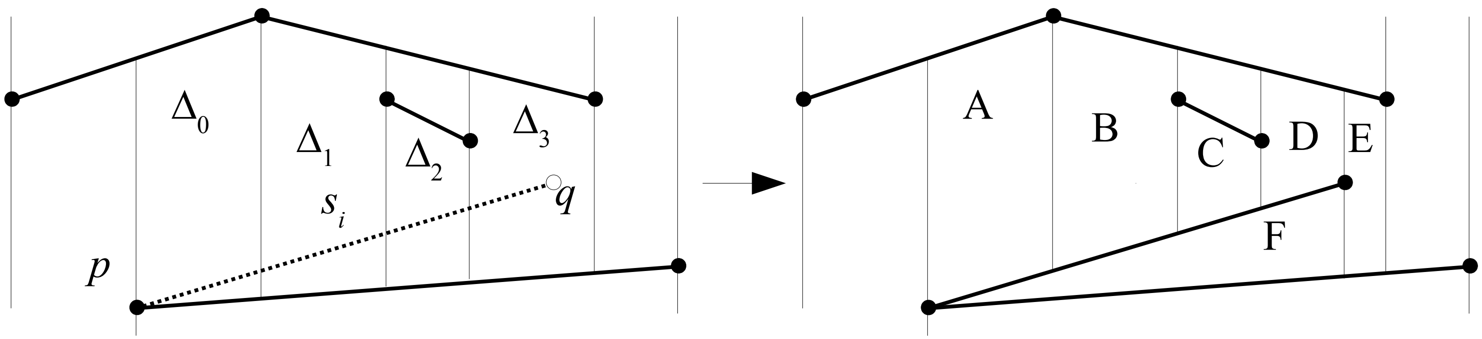


- In the case where the extremities of s_i already belong to T (*leftp* and *rightp* of Δ), only 2 or 3 trapezoids are added : the changes are made using the same idea.

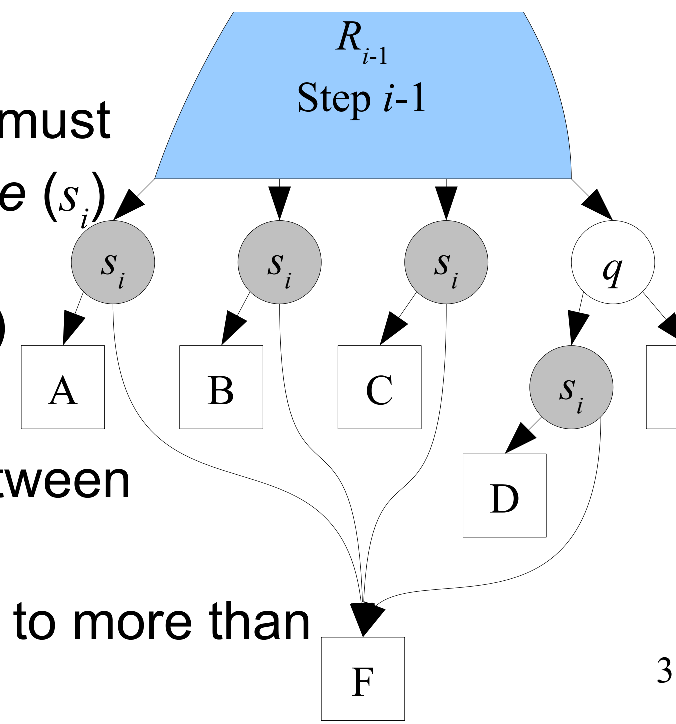
- Update of the search structure R

- The general case is somewhat more complex...





- principle : if Δ_0 has p in its interior, then if must be replaced by an x-*node* (p) and an y-*node* (s_i) the allows to choose between the three trapezoids (same on the other side with Δ_k)

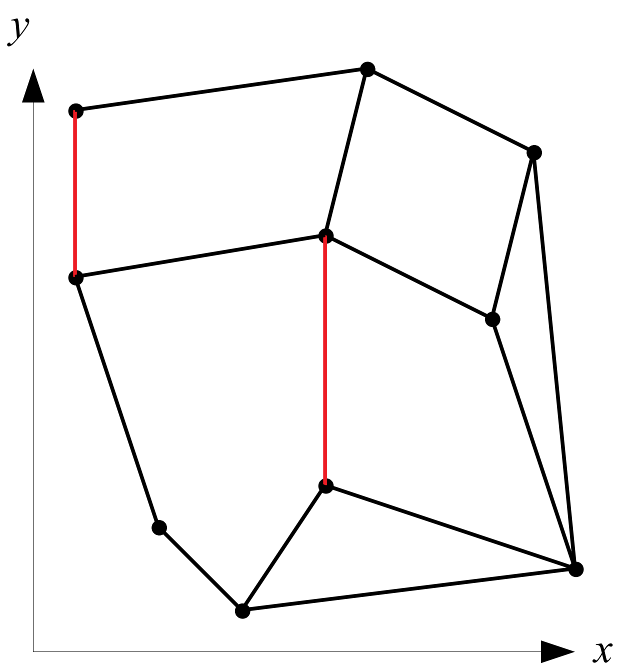


- For each trapezoid entirely cut,
one y-*node* must be inserted to choose between the superior or inferior trapezoid

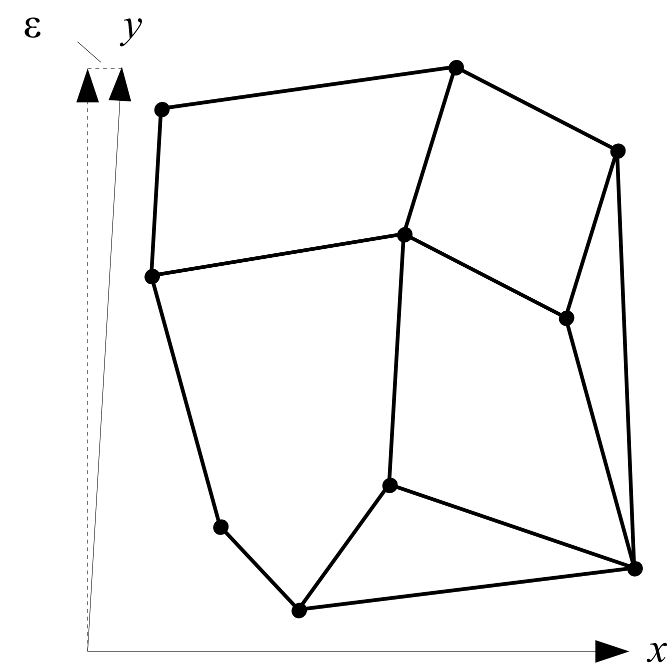
- It is possible that the new leaves connect to more than one node coming from the tree at step i-1.

- The validity of the algorithm is guaranteed by the invariant : at each step i, T_i and R_i are coherent with the i segments that have been inserted.
- Performances ?
  - Depends in which order the segments are inserted into R.
  - It can be shown that for degenerate case (e.g. when segments are sorted by position), R may be built in $O(n^2)$, and the search in R may be as worse as $O(n)$.
  - However, in average, for the $n!$ permutations in the order in which segments are inserted, the building time for R is in $O(n \log n)$ and a single search in $O(\log n)$

- Degenerate cases ...
  - It is the case where vertices may have equal x coordinates...
  - Use of a virtual shear transformation that does not change the x-ordering of nodes having distinct x coordinates.



$$\varphi : \begin{pmatrix} x \\ y \end{pmatrix} \rightarrow \begin{pmatrix} x+\epsilon\, y \\ y \end{pmatrix}$$



  - This transformation is virtual : we will just evaluate the result of applying this transformation to the result of the predicates used in the algorithms. The coordinates will not be changed in reality.

- Two predicates are used

  1 – comparison between two points p and q to know if q is on the right of p, on the left, or on a vertical line from p (at x-*nodes*)

  2 – comparison of a point q with a segment $p_1 p_2$ to know if q is above, below or on the segment (at y-*nodes*)

- Lets apply the transformation and check what happens in case 1 :

$$\varphi\, p = \begin{pmatrix} x_p + \epsilon\, y_p \\ y_p \end{pmatrix} \quad \text{et} \quad \varphi\, q = \begin{pmatrix} x_q + \epsilon\, y_q \\ y_q \end{pmatrix}$$

  - if $x_p \neq x_q$, the comparison is made on x_p and x_q and determines the result, because the ordering on x is not changed by ε (too small).

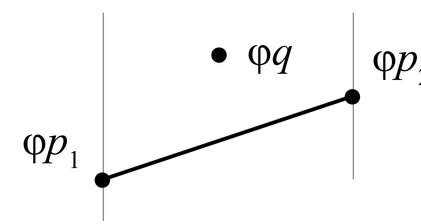
  - if $x_p = x_q$, then the relation between y_p and y_q determines the result.

- Therefore, it is sufficient to perform the test on a lexicographic order of (unchanged) nodes to simulate the shear transformation.

- Second case, comparison of a point q with a segment $s=p_1p_2$.
- We will test the following entities:

$$\varphi q=\begin{pmatrix} x_q+\epsilon y_q \\ y_q \end{pmatrix} \quad \text{and} \quad \varphi s : \left\{ \varphi p_1=\begin{pmatrix} x_1+\epsilon y_1 \\ y_1 \end{pmatrix} \;,\; \varphi p_2=\begin{pmatrix} x_2+\epsilon y_2 \\ y_2 \end{pmatrix} \right\}$$

One condition is always verified before any test in a *y-node* : the vertical through q intersects s : $x_1+\varepsilon y_1 \le x_q+\varepsilon y_q \le x_2+\varepsilon y_2$.
This implies $x_1 \le x_q \le x_2$.
If $x_q = x_1$ then $y_q \ge y_1$
If $x_q = x_2$ then $y_q \le y_2$
Let us consider two cases :

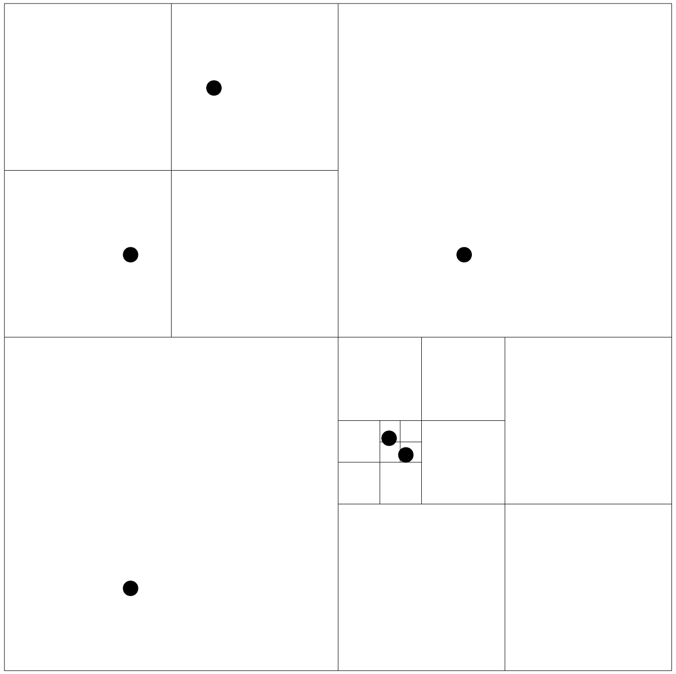


- If $x_q = x_2 = x_1$ then s is a vertical and $y_1 \le y_q \le y_2$ (in this case, q is in or on the segment s, and therefore φq is on φs)
- If $x_1 < x_2$, the transformation φ keeps the ordering, and the test on the initial coordinates gives the same result.
- Nothing to change here …

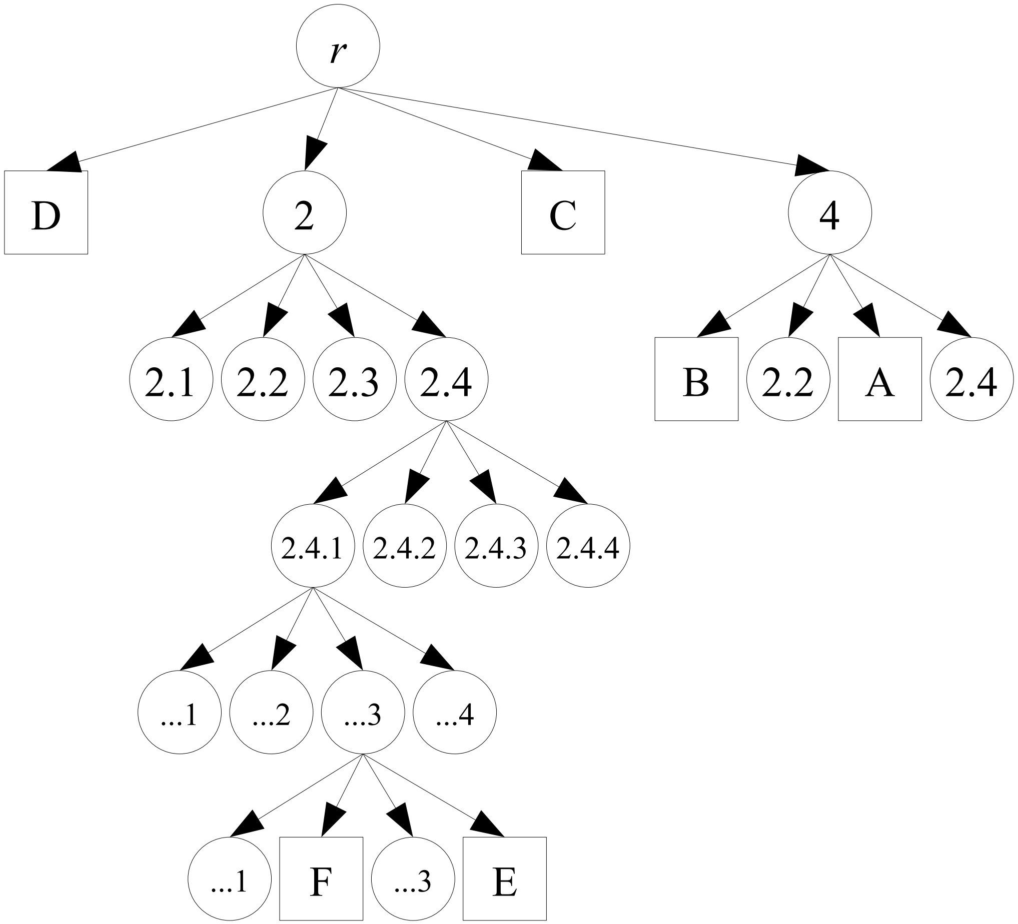
- In 3D : the case of the Octree
  - There does not (yet) exist a simple data structure allowing a search in O(log n) with a memory footprint in O(n) for any type of 3D geometries (*i.e.* independent of the position or distribution)
  - Either we lose in time, or in memory, or both...
  - The octree is a reasonable compromise here.
  - As the principle is exactly identical to quadtrees, we will show this case for the sake of clarity

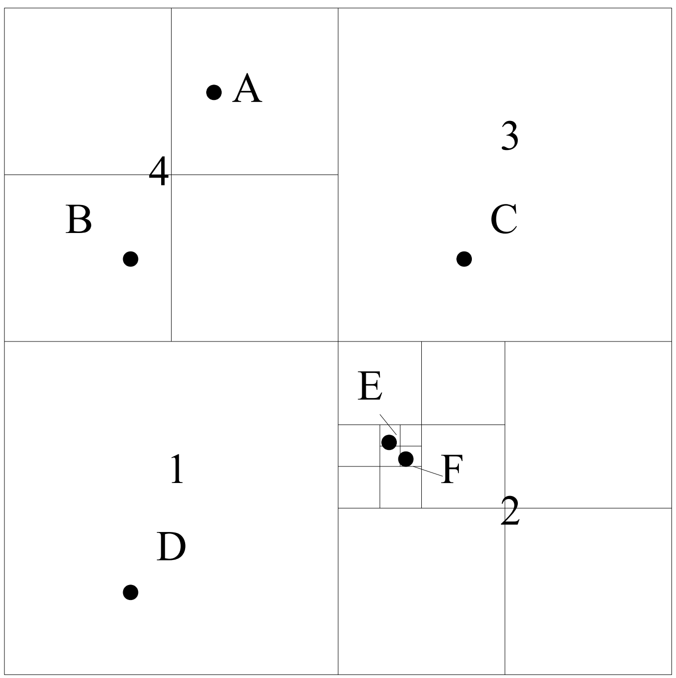
Geometric Search

- Same issue as before : quickly find in which polygon is located a given point q.
  - PM-Quadtree (PM stands for polygonal map) – there exist a huge zoology of datastructures (PR-, MX-, etc...) that we won't detail here. It is a recursive decomposition into rectangular cells.
  - The principle is shown here on a point set (point quadtree)
  - The cost of a search is proportional to the depth of the tree.
  - In turn, it depends on the spatial organization of the points, hence there are no interesting upper bound that is always valid …

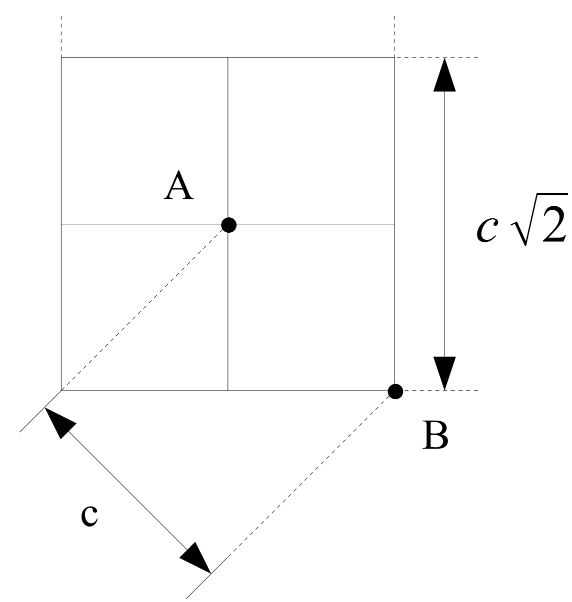


- Principle





- Complexity of the point-quadtree (depth)
  - In the worst case, if two points among the most close are separated by a tiny distance c, then the parent node of the two distinct leaves containing each of the two points must have a size that is at most $c\sqrt{2}$.
  - If the side of the initial cell of the quadtree is s ; a node at depth i will have a side equal to $s/2^i$
  - We have therefore : $\frac{s}{2^i} \geq c\sqrt{2} \Rightarrow \frac{s}{c} \geq \sqrt{2}\cdot 2^i \Rightarrow i \leq \log_2 \frac{s}{c} - \frac{1}{2}$
  - The depth of the internal node satisfies then: $p_n \leq \log_2 \frac{s}{c} + \frac{1}{2}$
  - The total depth (including leaves) is $p = p_n + 1$, and satisfies $p \leq \log_2 \frac{s}{c} + \frac{3}{2}$



- In the case of an octree :
  - Same reasoning, but the diagonal now measures $c\dfrac{2}{\sqrt{3}}$

    Therefore,

    $$\frac{s}{2^i} \geq c\frac{2}{\sqrt{3}} \quad \Rightarrow \quad \frac{s}{c} \geq \frac{2}{\sqrt{3}} \cdot 2^i \quad \Rightarrow \quad i \leq \log_2 \frac{s}{c} - 1 + \frac{1}{2}\log_2 3$$

  - The total depth is therefore limited to :

    $$p \leq \log_2 \frac{s}{c} + 1 + \frac{1}{2}\log_2 3$$

- Complexity of the point-quadtree (number of cells)
  - Lets start from an initial quatree with an internal node and 4 leaves (n_i=1, n_f=4). The construction proceeds by the addition of an internal node and 4 new leaves, that replace a leaf

    Therefore, the number of leaves is $n_f = 1 + 3$ times the number of internal nodes that have been added, so $n_f = 3\ n_i + 1$
  - How many internal nodes n_i ?

    Each internal node "contains" at least one point (inside the associated square area). The squares at a given depth are disjoint and form a paving of the initial square (there is no area missing)

    Therefore, at a given depth, the total number of internal nodes is bounded by n, the number of inserted points.
  - In the worst case, the number of internal nodes is bounded by $n\ (d+1)$
  - The size of the whole structure is therefore in $O(n(d+1))$. (same result for the Octree), and The depth may be high as the number of points … $d \sim O(n)$

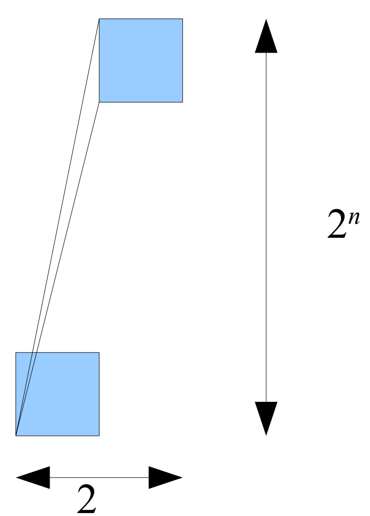
- Complexity in building time
  - The algorithm is recursive and based on sorting points in the 4 quadrants.
  - The quadtree is subdivided until there is only one 1 point in each of the quadrants of the current level (depth).
  - We have seen that the total internal nodes for a given level is at most n (number of points).
  - In the worst case, at each level until depth d, every point but one will be classified again and the complexity may be as bad as $O(n(d+1))$., with $d \sim O(n)$ in the worst case.
  - Generally, closer to $O(n(\log n))$ because in a good distribution of points $d \sim O(\log n)$
  - Again, same bound for the Octree...

- PM1 Quadtree
  - Allows to classify a polygonal decomposition (like a mesh)
  - 4 conditions :

    1. At most one vertex in a leaf of the quadtree
    2. If a leaf contains a vertex, it cannot contain an edge not emanating from this vertex
    3. If a leaf does not contain a vertex, then it can contain at most one edge
    4. Each leaf is maxima (not useful to subdivide more, and impossible to subdivide less)

- Global cost of such a structure
  - Size of the structure : $O(L \,.\, 2^D \,.\, (D+A)\,)$
    (A = average valence of the vertices)
  - Depth D of the structure

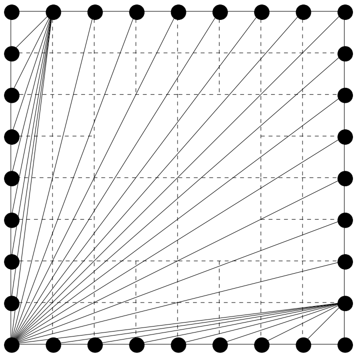
    In the case where the vertices are on a regular grid $2^n.2^n$, a superior bound is $4.n$ (n being the number of bits in the coordinates of the vertices …)
  - This is because we avoid having multiple edges in one cell (except of course if they are connected to the same vertex).



Geometric Search

- PM3 Quadtree
  - Allow also to represent a polygonal decomposition
  - Only two conditions :
  - 1. At most one vertex in a leaf of the quadtree
    2. If a leaf contains a vertex, it cannot contain an edge not emanating from this vertex
    3. If a leaf does not contain a vertex, then it can contain at most one edge
    4. Each leaf is maximal (not useful to subdivide more, and impossible to subdivide less)
  - The number of nodes in the quadtree is the same as the one based only on vertices (point-quadtree)
  - The complexity of the cells is increased (contain the complete set of edges intersecting the cell)

- Global cost of such a structure
  - size of the structure : $O(L\ .\ 2^D\ .\ (D+A)\ )$ (A = average valence of the vertices)
  - Depth D of the structure :
  - In the case where the vertices are on a regular grid $2^n.2^n$, a superior bound is n (n being the number of bits in the coordinates of the vertices …)
  - It is similar to the bound of a point - quadtree .
  - However, the cells contain a potentially high number of edges (up to n …), said otherwise, the partition of the $O(n)$ polygons by the $O(n)$ cells leads unfortunately to a complexity in $O(n^2)$, which makes it far from optimal (an optimal partition in 2D would be linear)



Geometric Search

- Octree ? Generalization in 3D.
  - PM octree
  - 6 conditions

    1- One vertex max. in one leaf

    2- If a leaf contains a vertex, it may not contain other edges and faces not connected to that vertex

    3- If a leaf does not contain a vertex, it may contain up to one edge

    4- If it does not contain a vextex, and one edge, it cannot contain any other face that is not connected to the edge

    5- If it contain no edge, it can only contain one face

    6- Each leaf is maximal

Geometric Search

- See reference book for more info

Hanan Samet, *Foundations of Multidimensional and Metric Data Structures*, 2006, Morgan-Kaufmann, San Francisco

ISBN 978-0-12-369446-1