DCGI

DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

TRIANGULATIONS

PETR FELKEL

FEL CTU PRAGUE

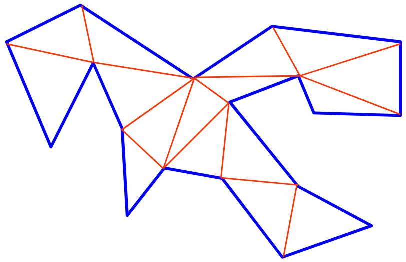
felkel@fel.cvut.cz

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

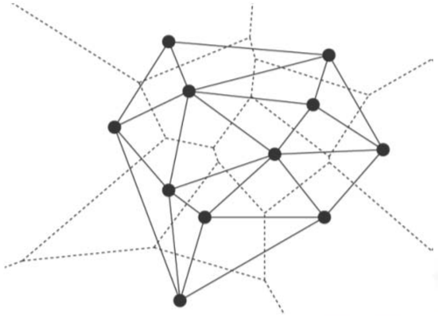
Based on [Berg] and [Mount]

Version from 10.12.2016

Talk overview



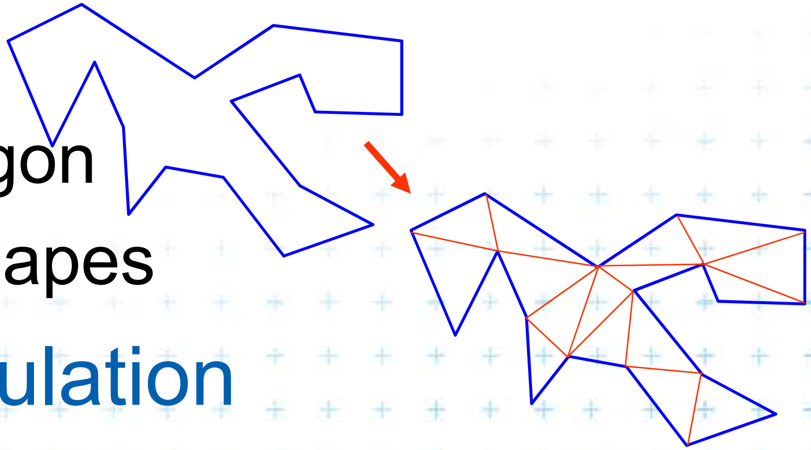
- Polygon triangulation
  - Monotone polygon triangulation
  - Monotonization of non-monotone polygon



- Delaunay triangulation (DT) of points
  - Input: set of 2D points
  - Properties
  - Incremental Algorithm
  - Relation of DT in 2D and lower envelope (CH) in 3D
    and
    relation of VD in 2D to upper envelope in 3D

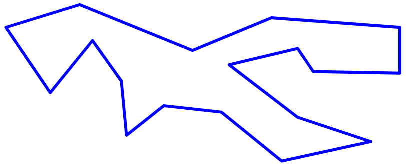
Polygon triangulation problem

- Triangulation (in general)
  = subdividing a spatial domain into simplices
- Application
  - decomposition of complex shapes into simpler shapes
  - art gallery problem (how many cameras and where)
- We will discuss
  - Triangulation of a simple polygon
  - without demand on triangle shapes
- Complexity of polygon triangulation
  - O(*n*) alg. exists [Chazelle91], but it is too complicated
  - practical algorithms run in O(*n* log *n*)



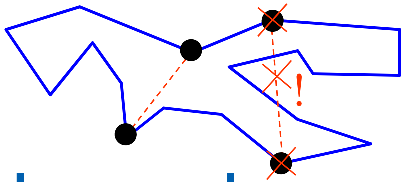
Terminology

Simple polygon



= region enclosed by a closed polygonal chain that does not intersect itself

Visible points

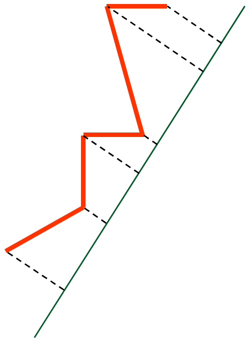


= two points on the boundary are visible if the interior of the line segment joining them lies entirely in the interior of the polygon

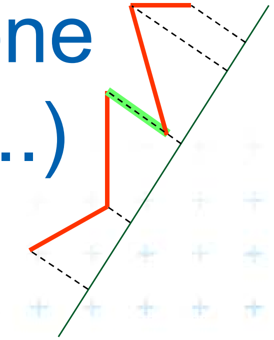
Diagonal

= line segment joining any pair of visible vertices

Terminology



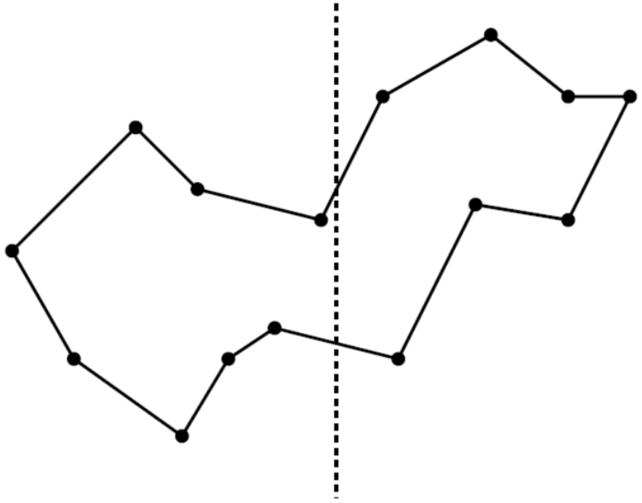
- A polygonal chain C is strictly monotone with respect to line L, if any line orthogonal to L intersects C in at most one *point*
- A chain C is monotone with respect to line L, if any line orthogonal to L intersects C in at most one *connected component* (point, line segment,...)



- Polygon P is monotone with respect to line L, if its boundary (bnd(P), ∂P) can be split into two chains, each of which is monotone with respect to L

Terminology

- Horizontally monotone polygon
  = monotone with respect to *x*-axis
  - Can be tested in *O*(*n*)
  - Find leftmost and rightmost point in *O*(*n*)
  - Split boundary to upper and lower chain
  - Walk left to right, verifying that x-coord are non-decreasing



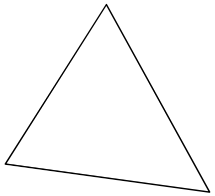
x–monotone polygon

[Mount]

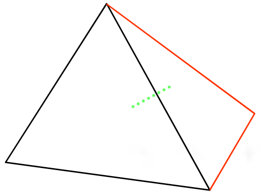
Terminology

- Every simple polygon can be triangulated
- Simple polygon with *n* vertices consists of
  - exactly n-2 triangles
  - exactly n-3 diagonals
  - Each diagonal is added once
    => O(n) sweep line algorithm exist

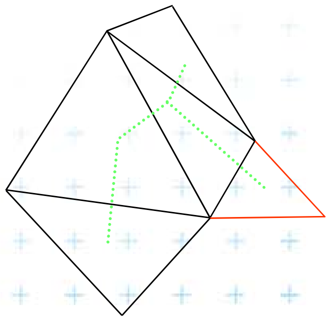
Proof by induction



n = 3 => 0 diagonal



n = 4 => 1 diagonal



n := n+1 => n + 1 – 3 diagonals

n + 1 = 7 => 4 diagonals)

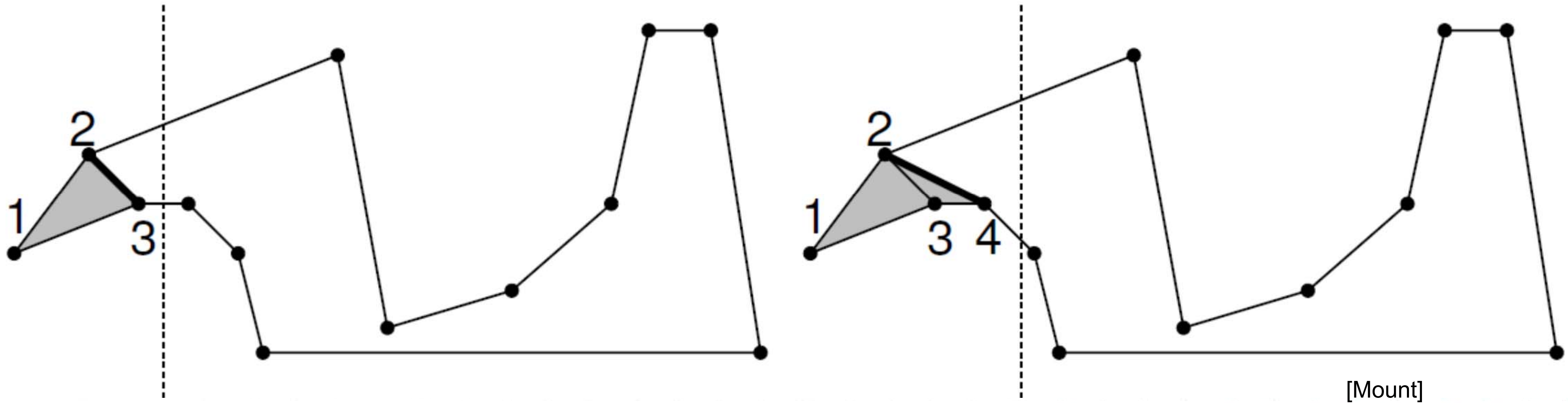
Simple polygon triangulation

- Simple polygon can be triangulated in 2 steps:
  1. Partition the polygon into x-monotone pieces
  2. Triangulate all monotone pieces

(we will discuss the steps in the reversed order)

2. Triangulation of the monotone polygon

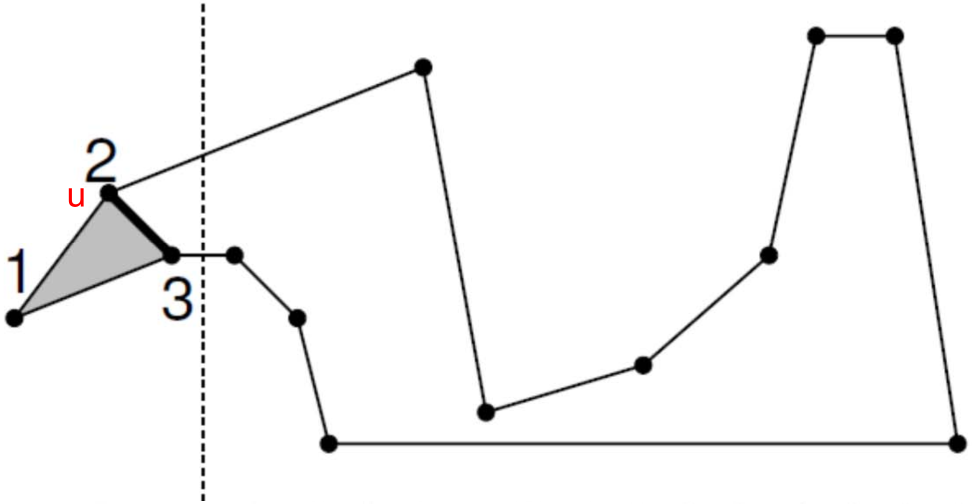
- Sweep left to right - in O(n) time
- Triangulate everything you can by adding diagonals between visible points
- Remove triangulated region from further consideration – mark as DONE

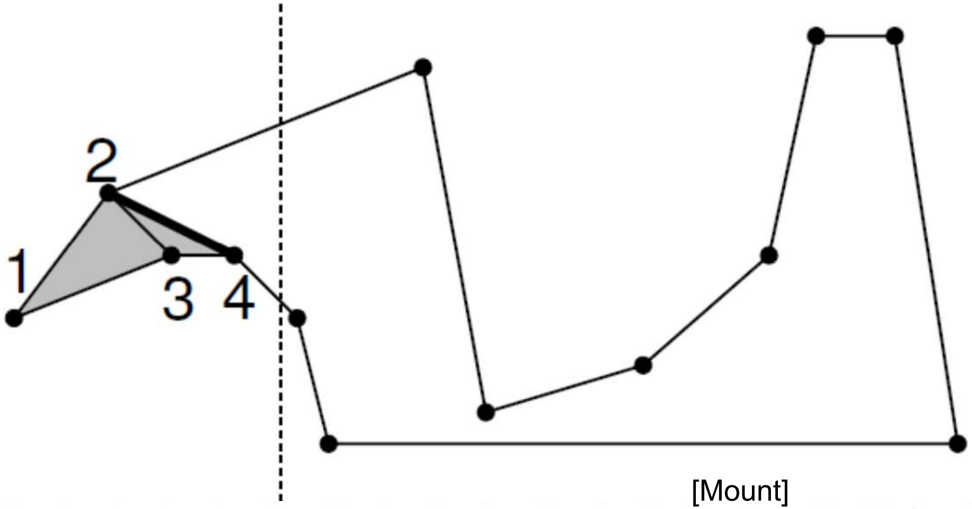


[Mount]

2. Triangulation of the monotone polygon

- Sweep left to right - in O(n) time
- Triangulate everything you can by adding diagonals between visible points
- Remove triangulated region from further consideration – mark as DONE

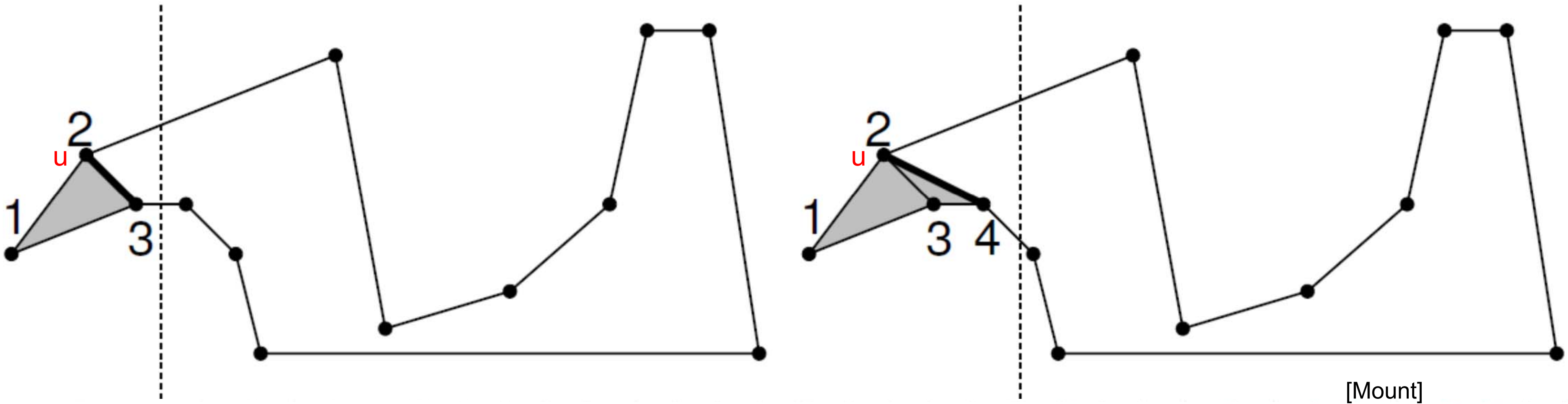




[Mount]

2. Triangulation of the monotone polygon

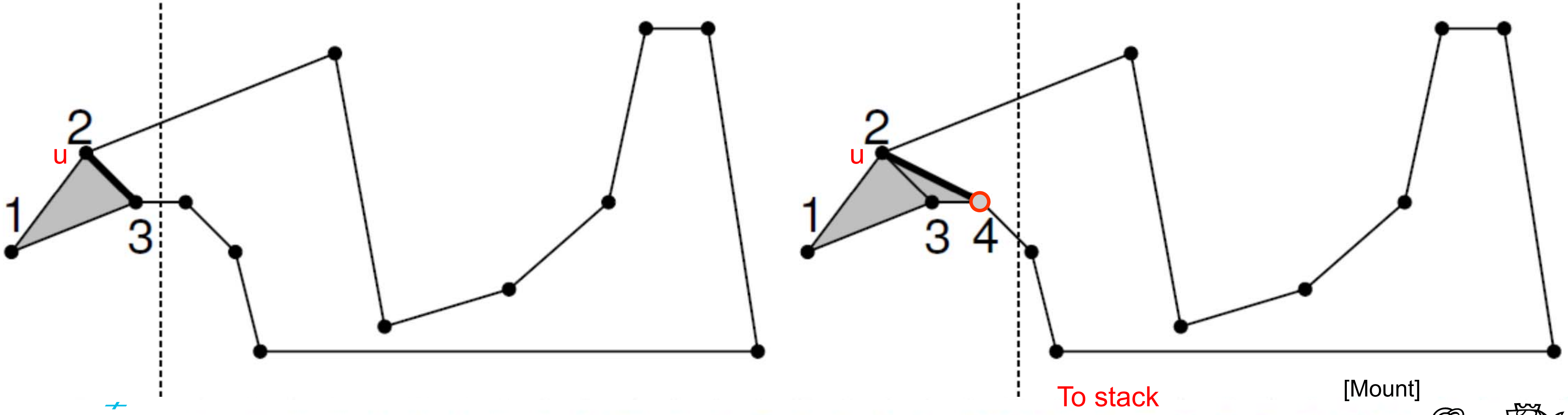
- Sweep left to right - in O(n) time
- Triangulate everything you can by adding diagonals between visible points
- Remove triangulated region from further consideration – mark as DONE



[Mount]

2. Triangulation of the monotone polygon

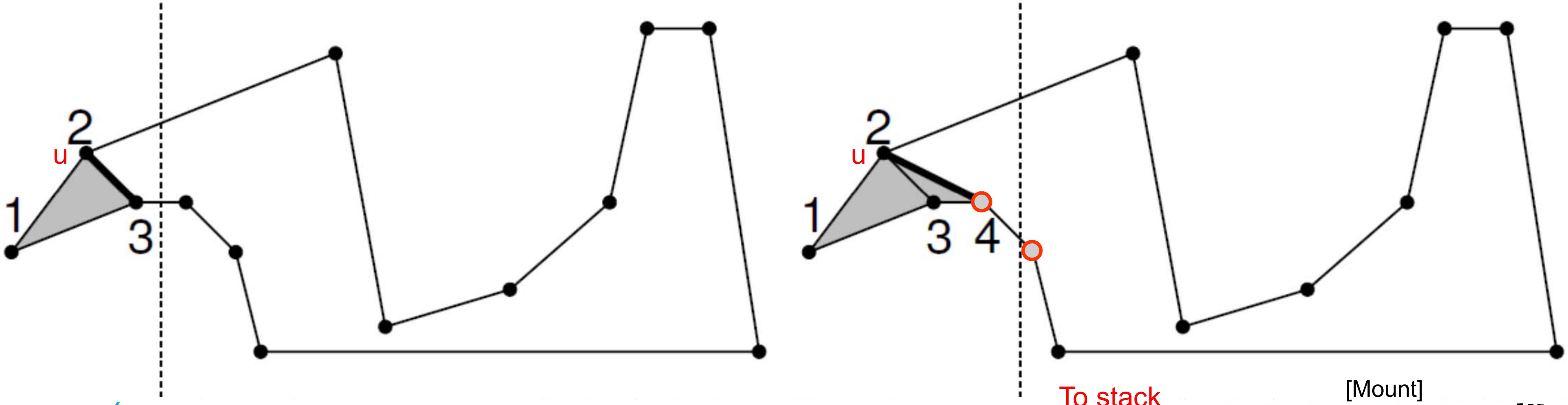
- Sweep left to right - in O(n) time
- Triangulate everything you can by adding diagonals between visible points
- Remove triangulated region from further consideration – mark as DONE



[Mount]

2. Triangulation of the monotone polygon

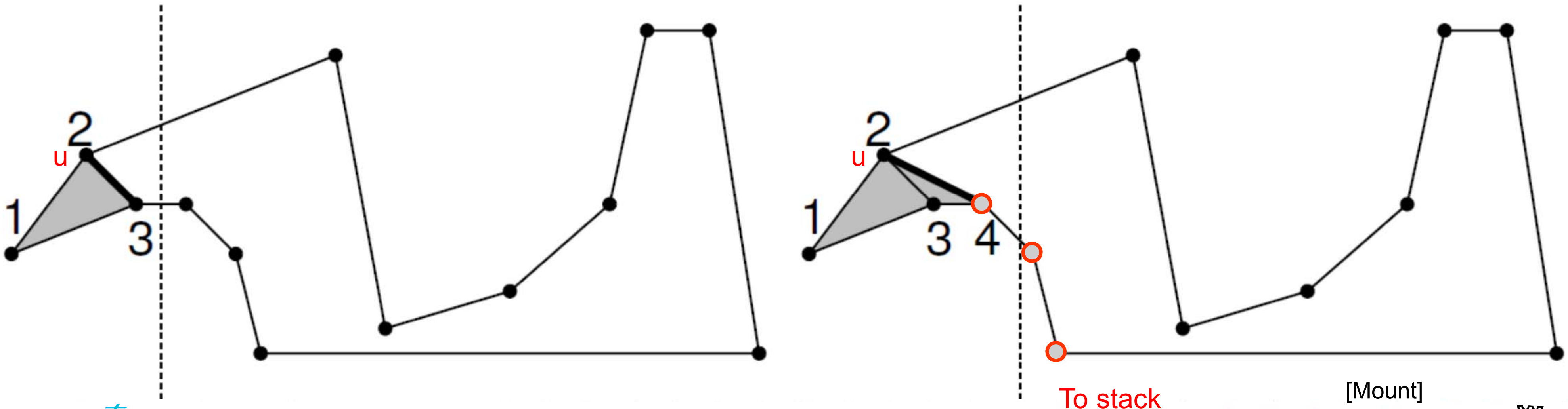
- Sweep left to right - in O(n) time
- Triangulate everything you can by adding diagonals between visible points
- Remove triangulated region from further consideration – mark as DONE



[Mount]

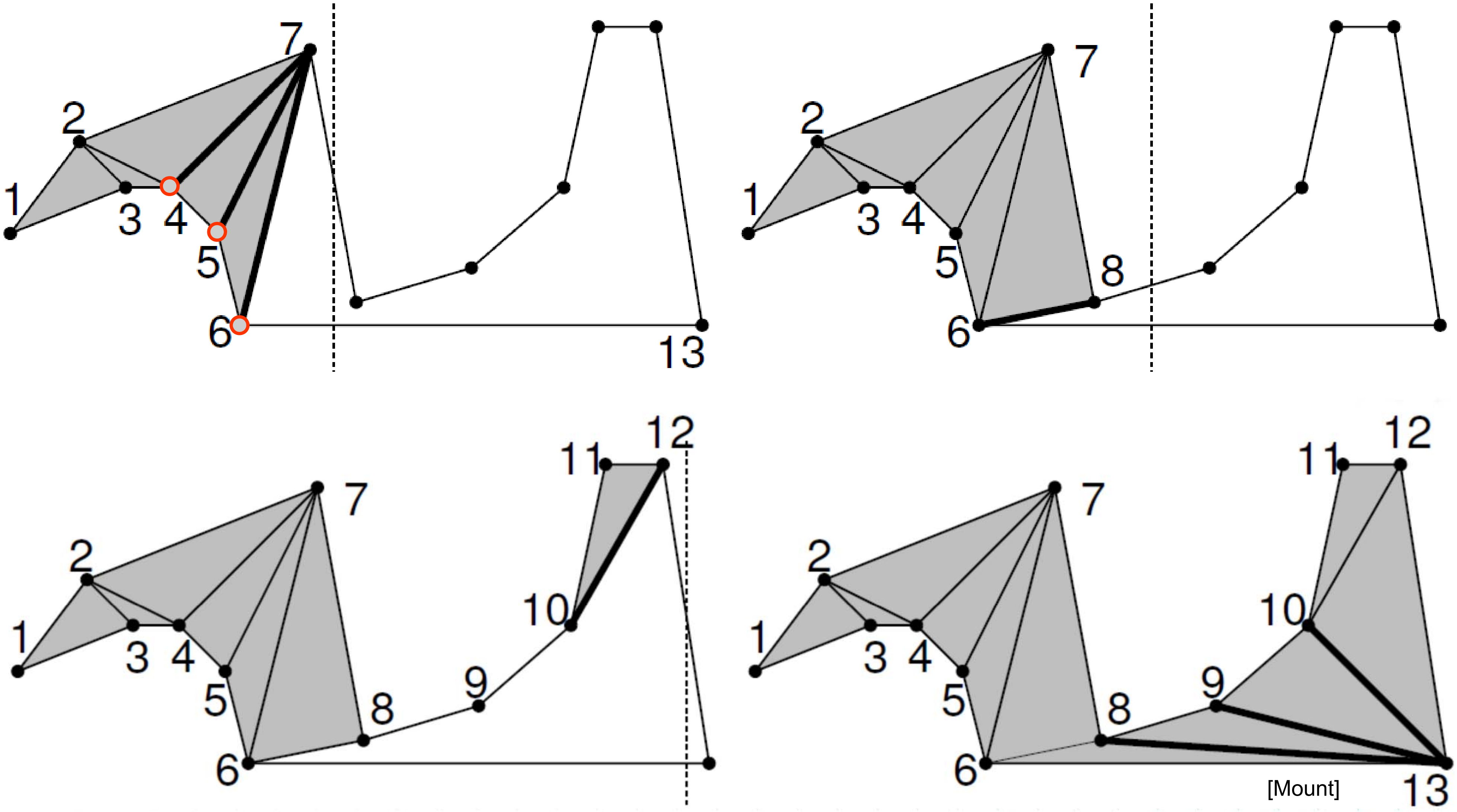
2. Triangulation of the monotone polygon

- Sweep left to right - in O(n) time
- Triangulate everything you can by adding diagonals between visible points
- Remove triangulated region from further consideration – mark as DONE



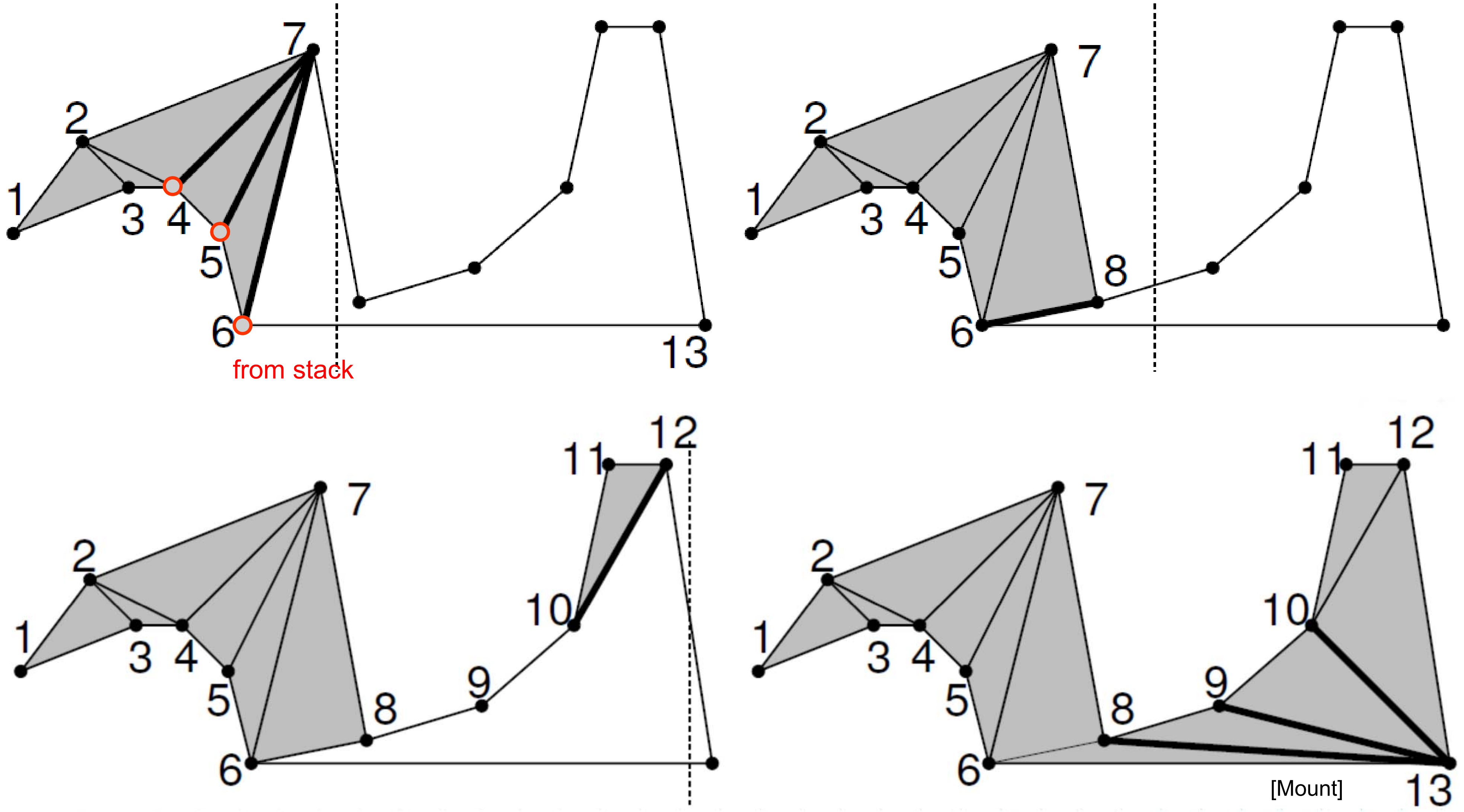
[Mount]

Triangulation of the monotone polygon



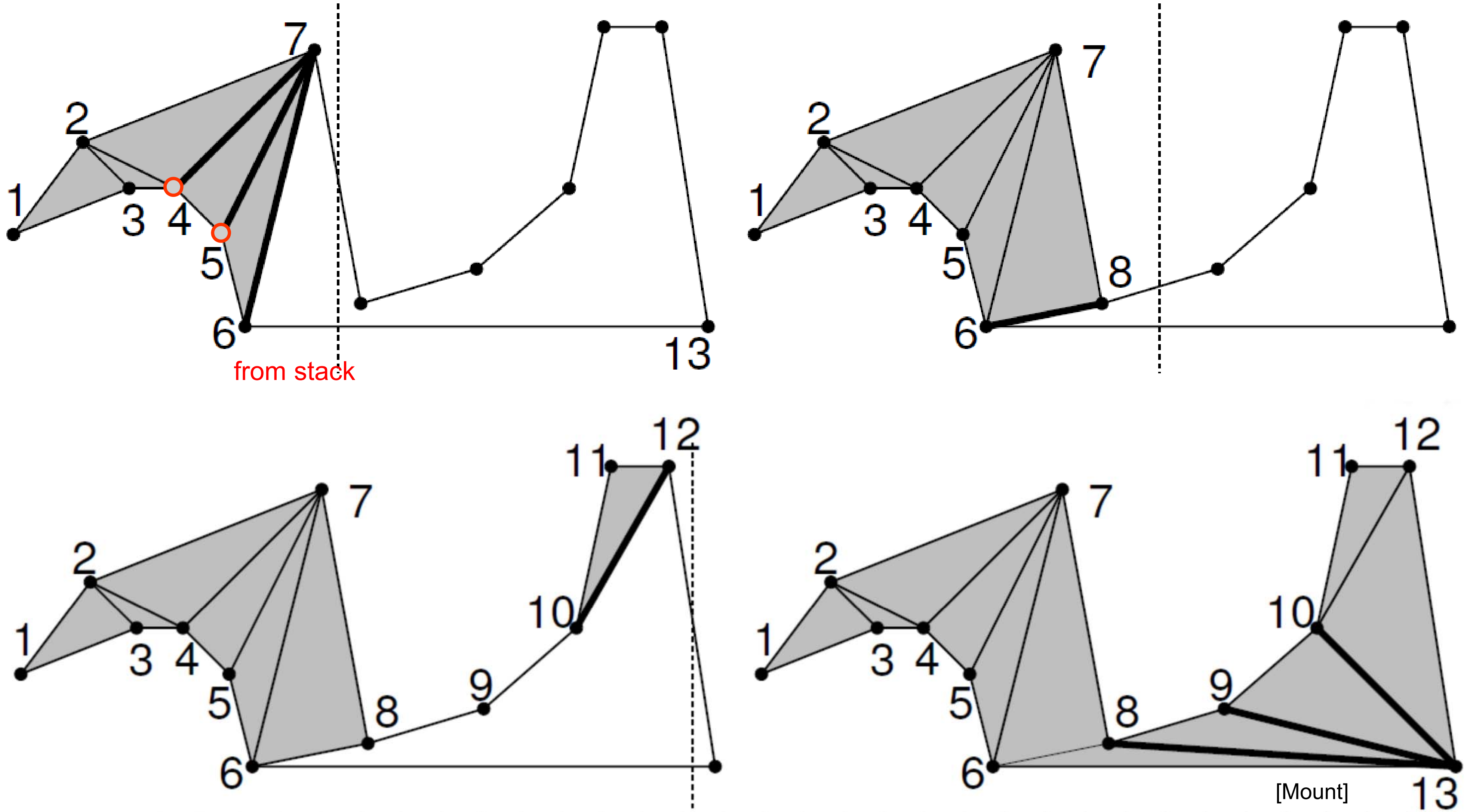
[Mount]

Triangulation of the monotone polygon



[Mount]

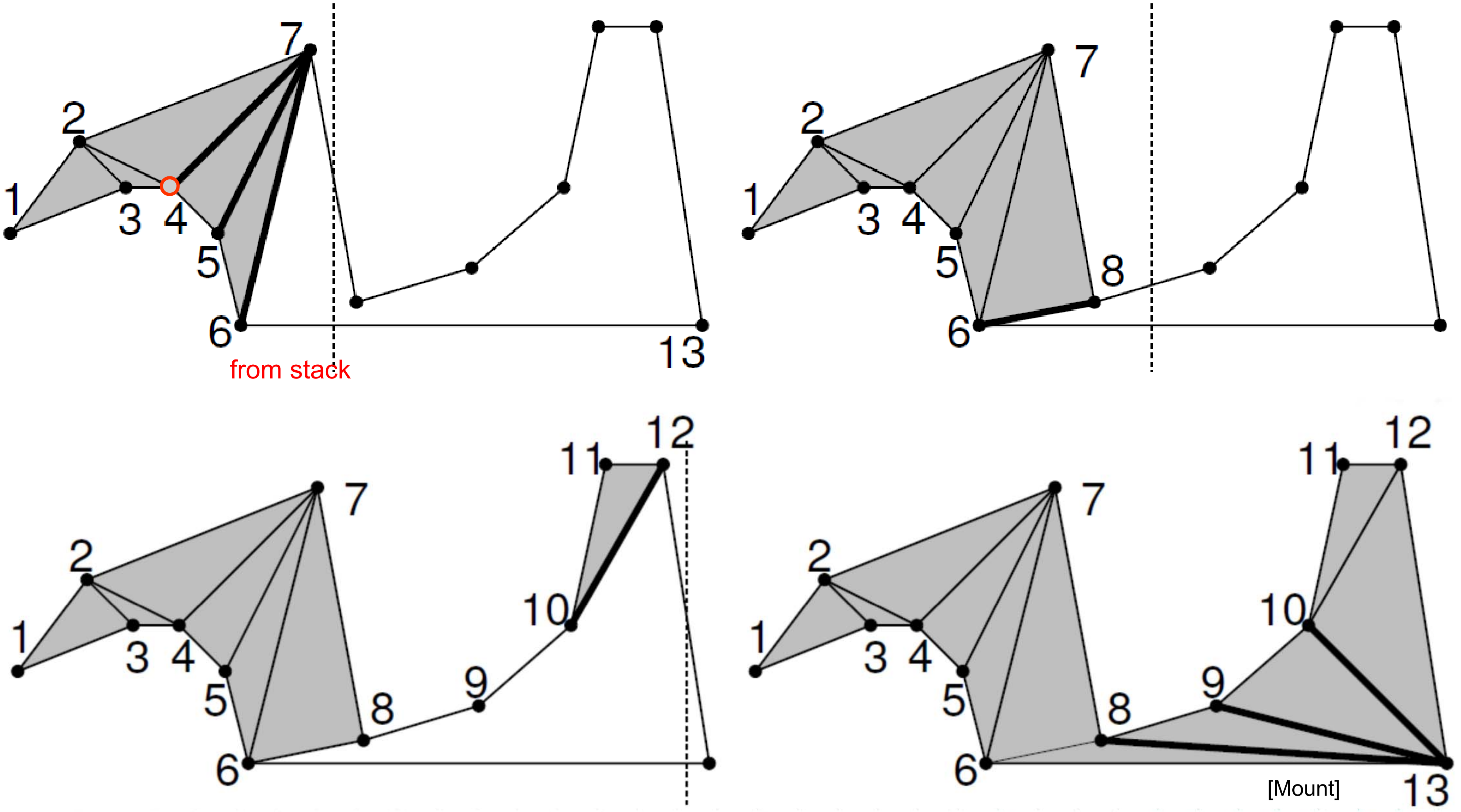
Triangulation of the monotone polygon



from stack

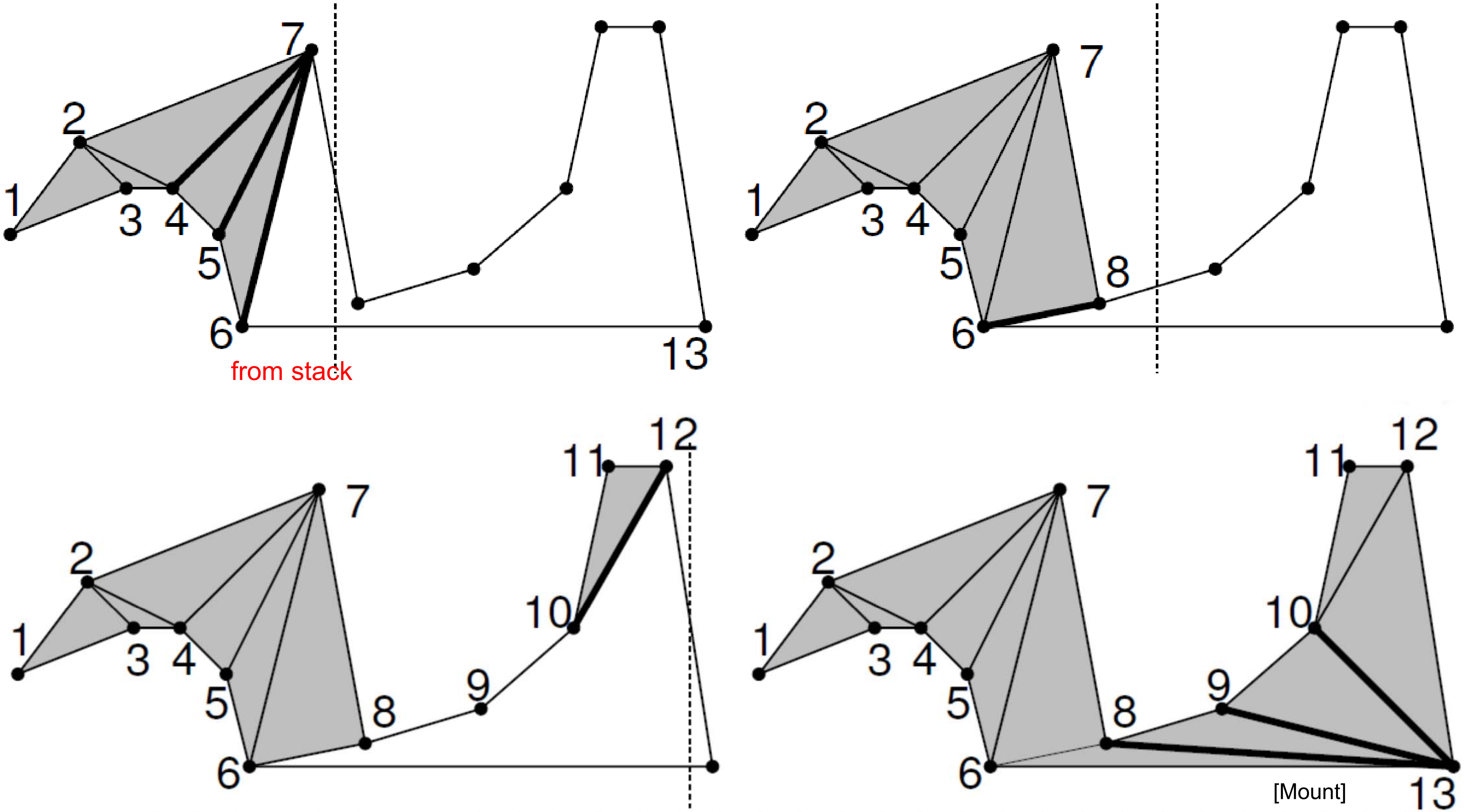
[Mount]

Triangulation of the monotone polygon



[Mount]

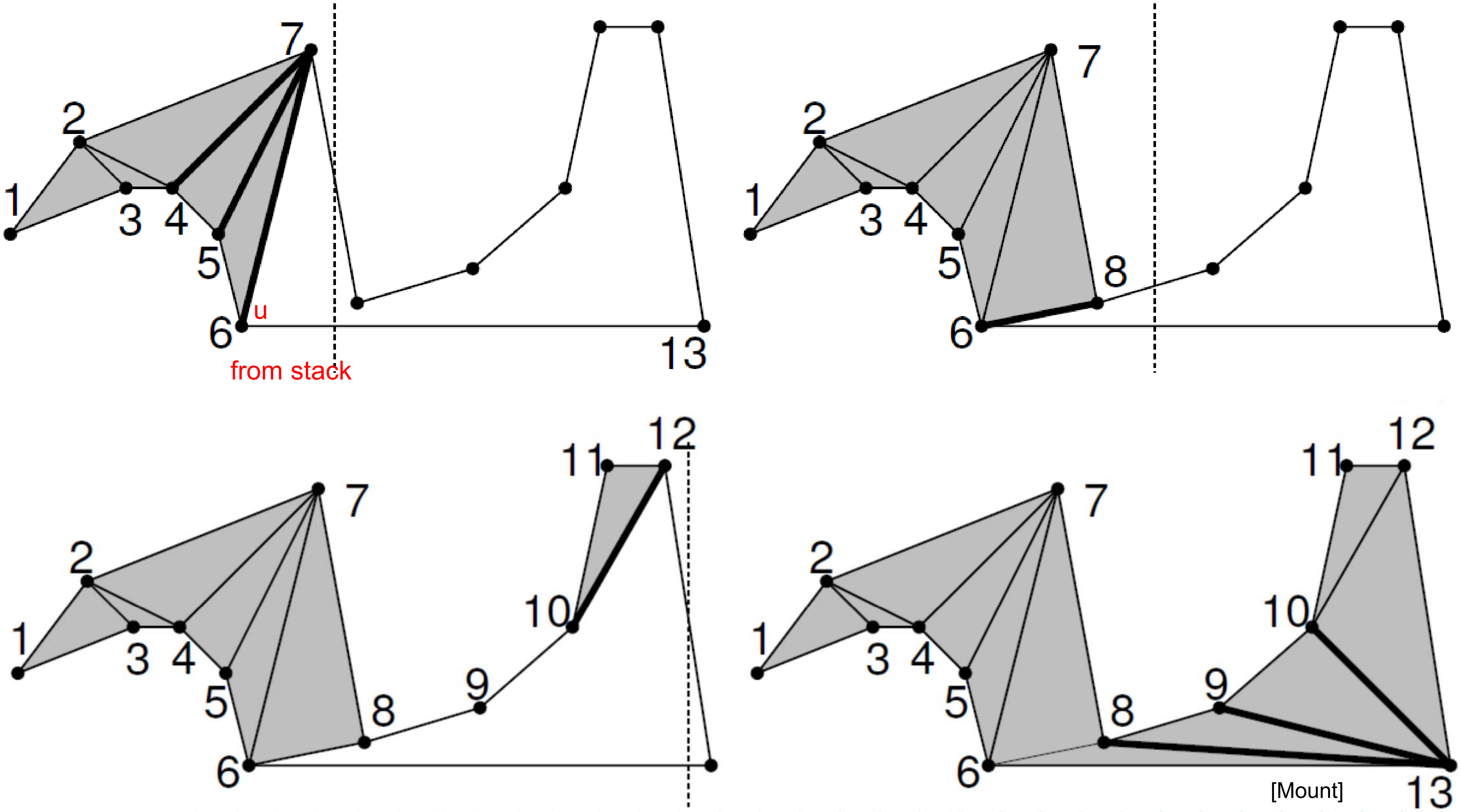
Triangulation of the monotone polygon



from stack

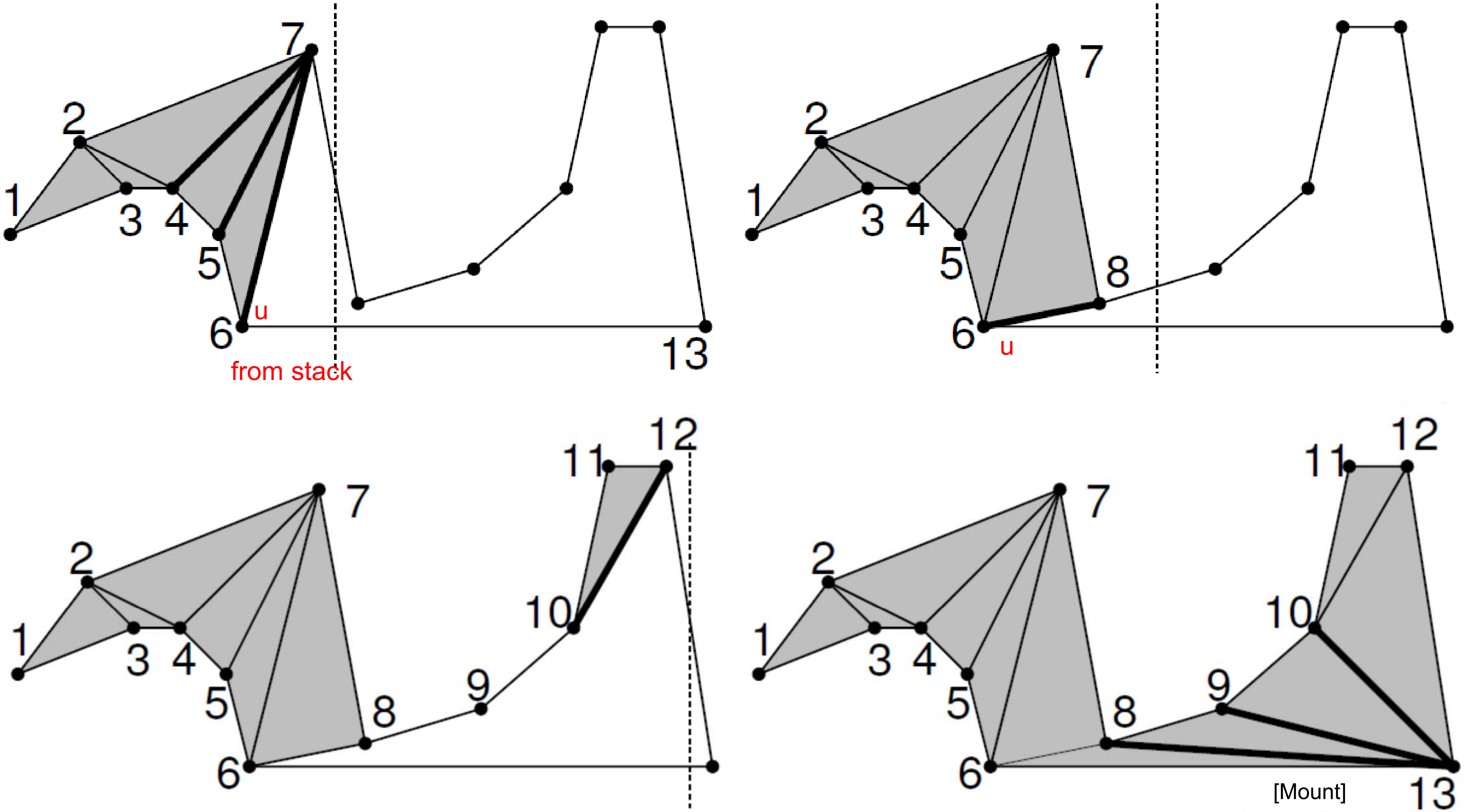
[Mount]

Triangulation of the monotone polygon

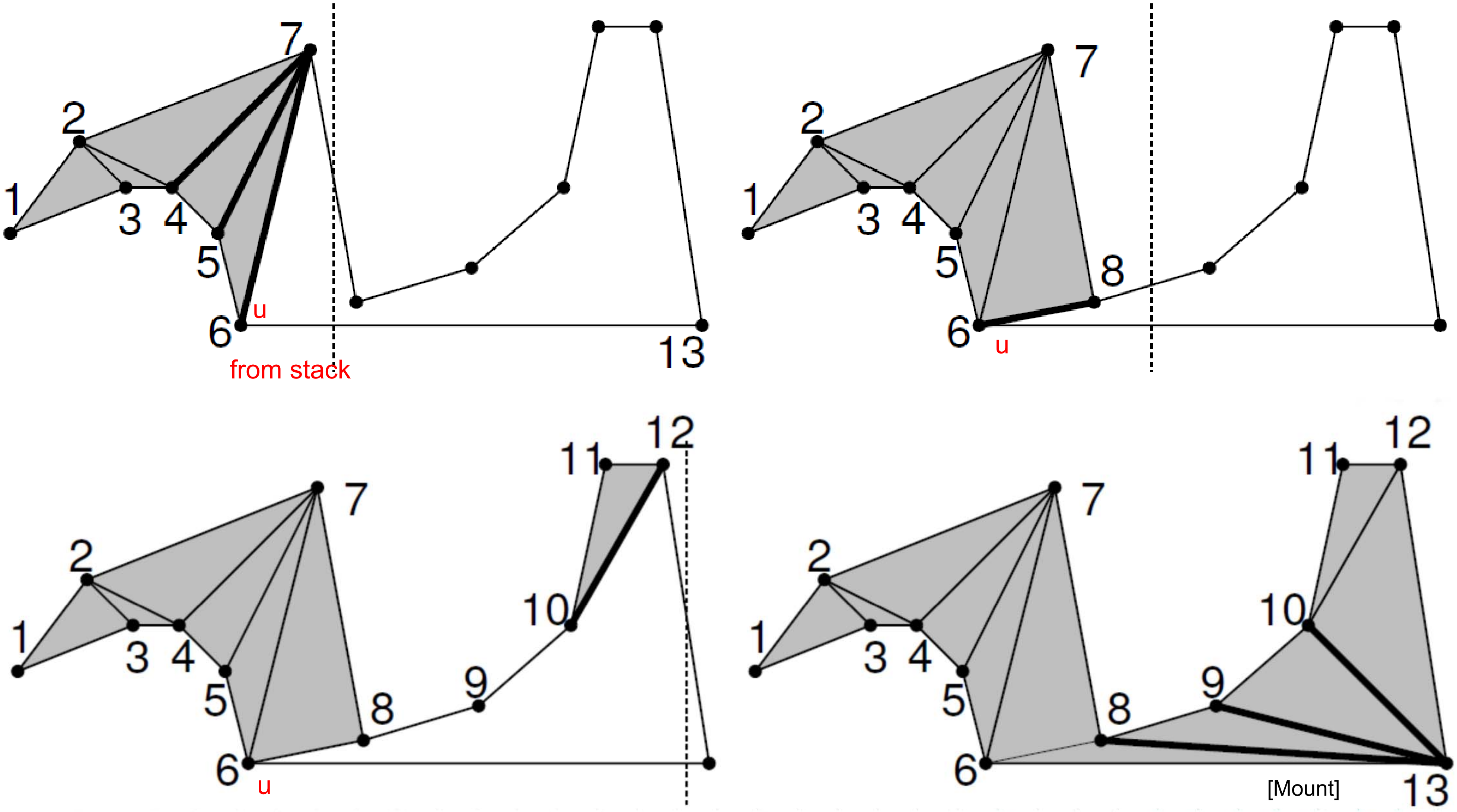


[Mount]

Triangulation of the monotone polygon

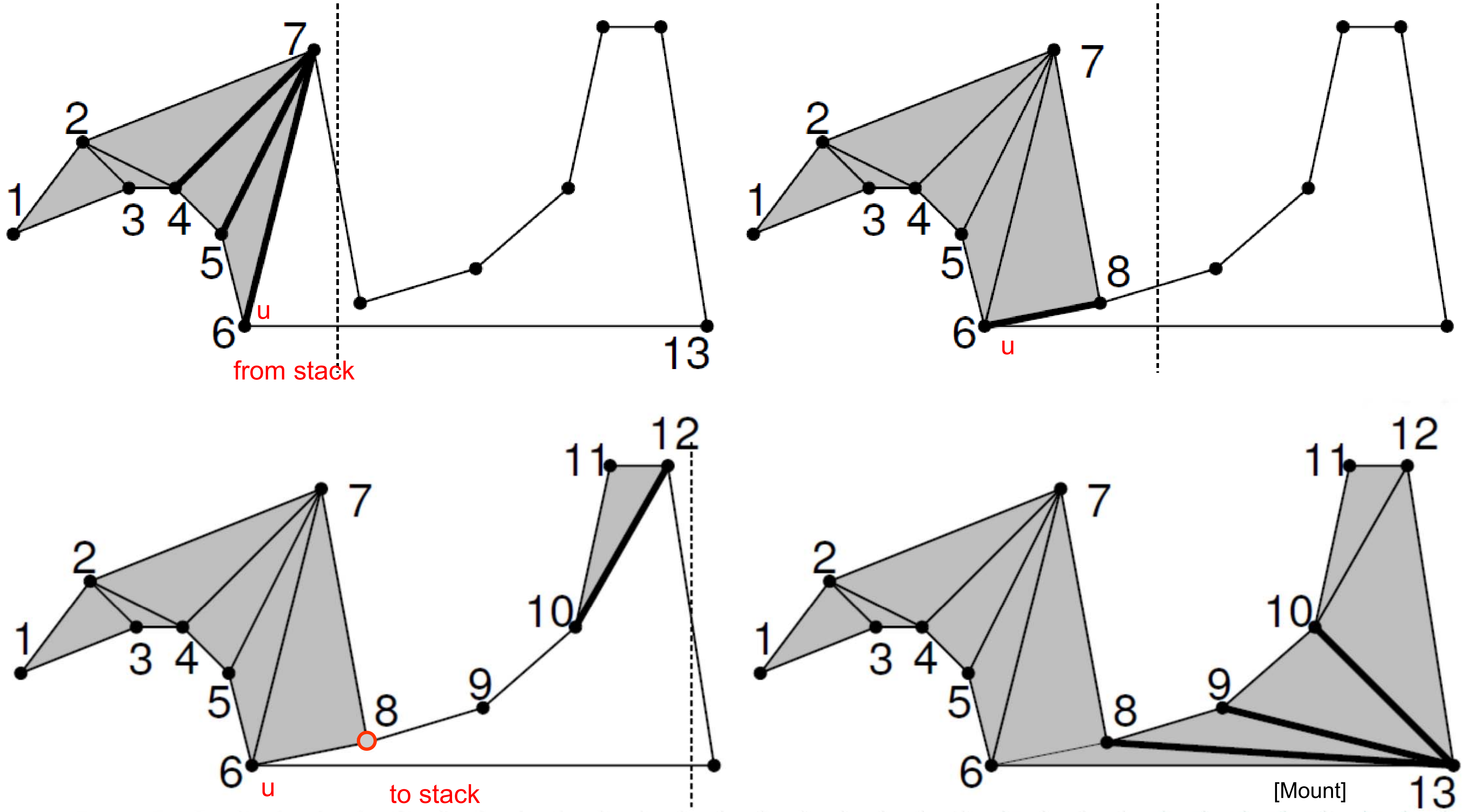


Triangulation of the monotone polygon



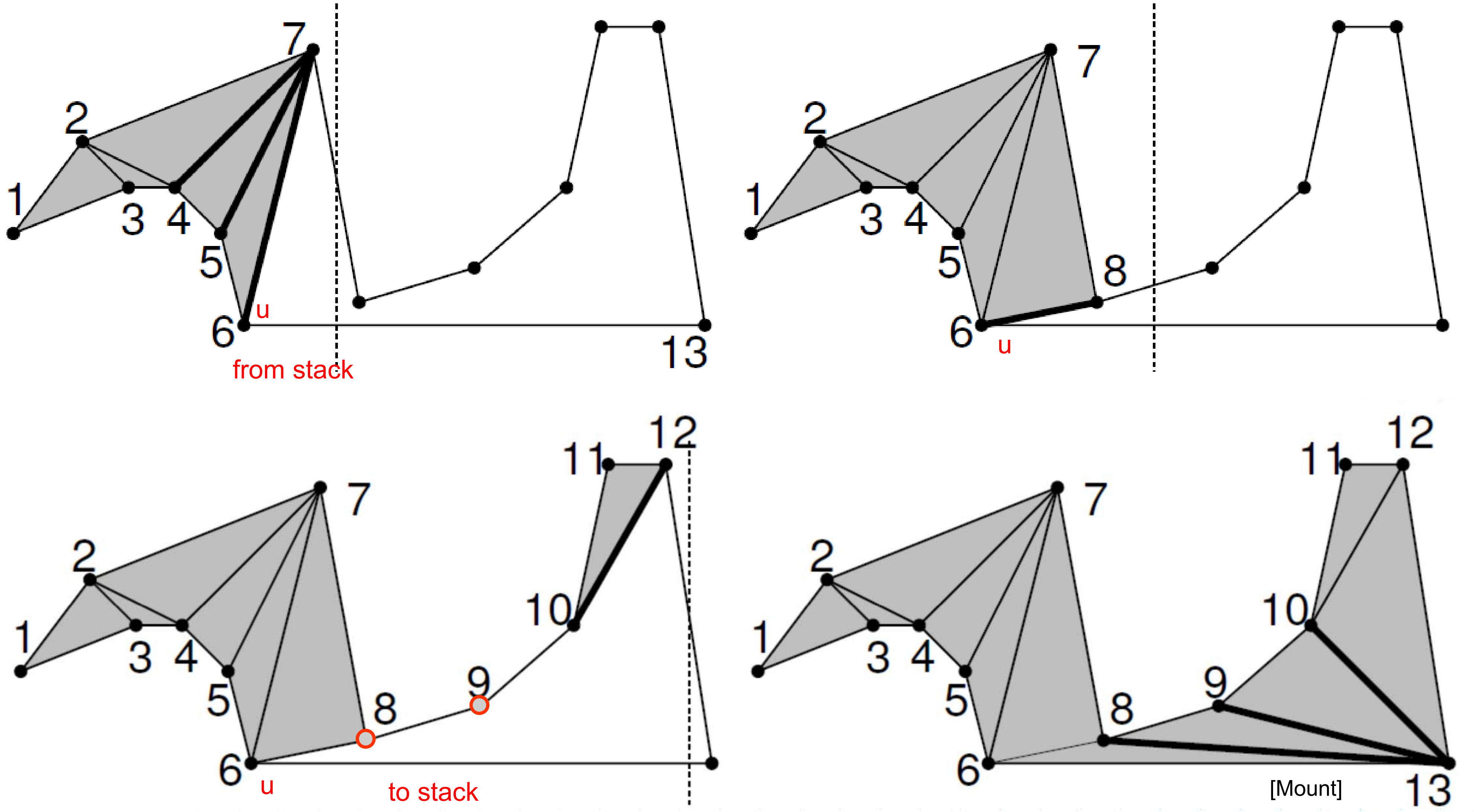
[Mount]

Triangulation of the monotone polygon



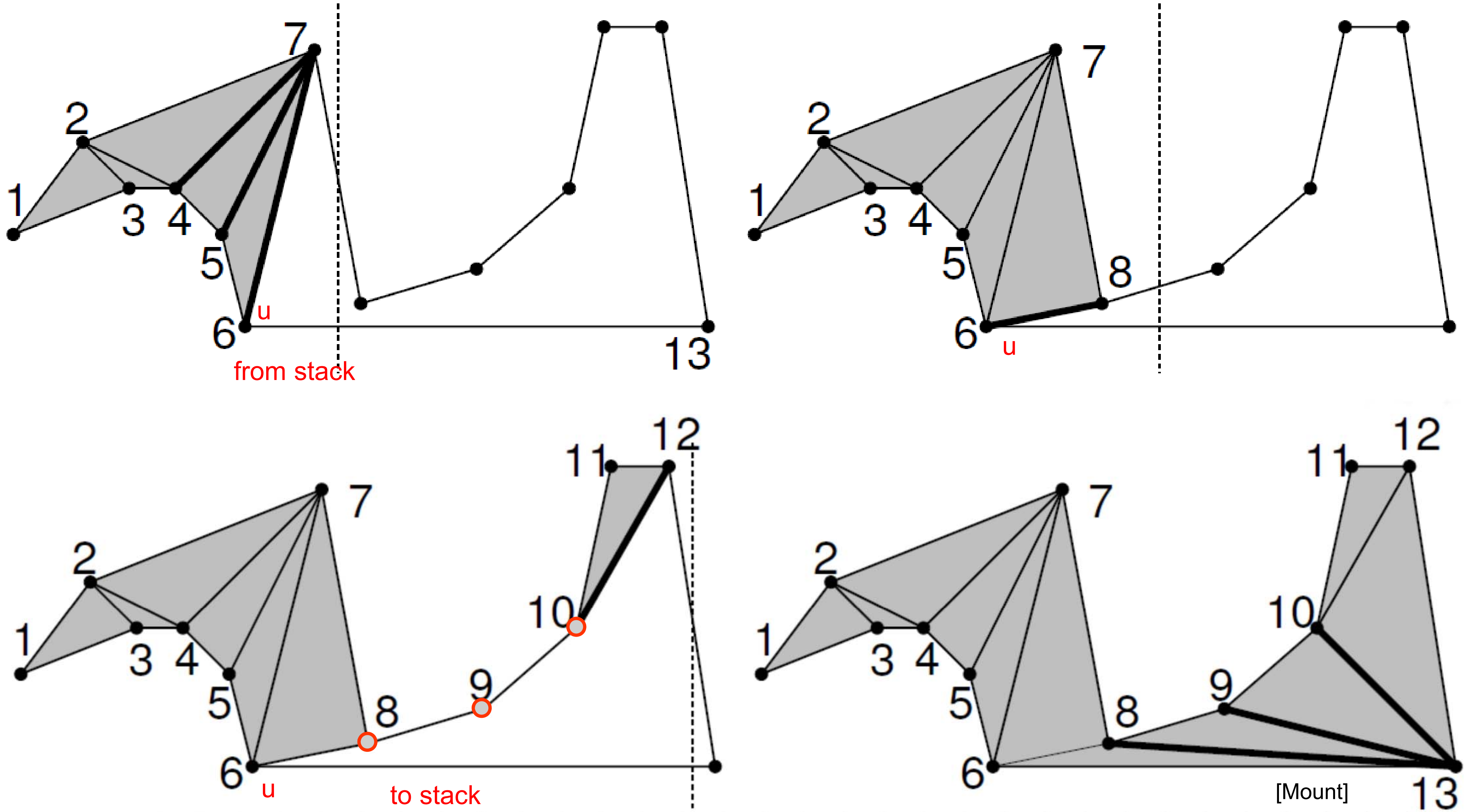
[Mount]

Triangulation of the monotone polygon

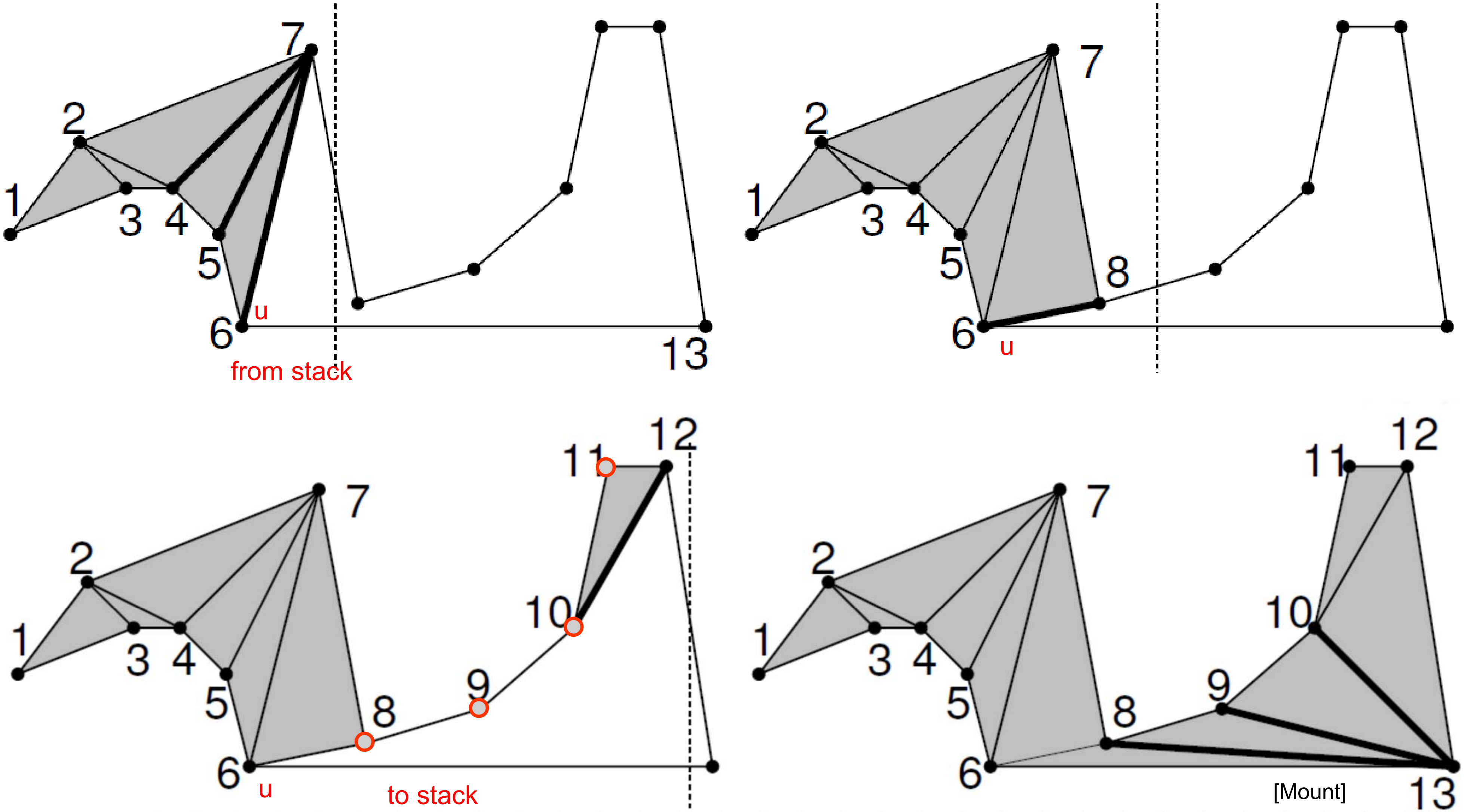


[Mount]

Triangulation of the monotone polygon

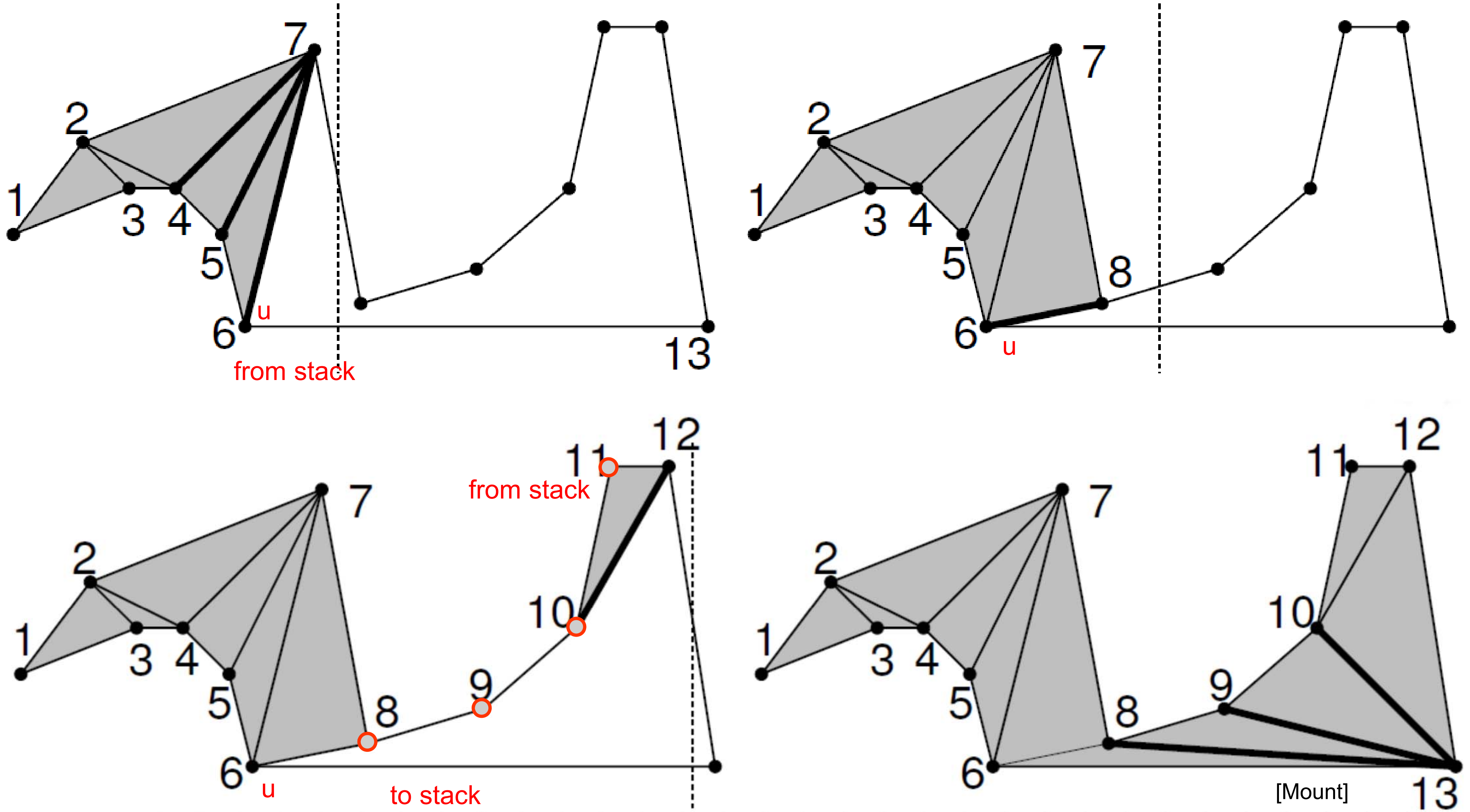


Triangulation of the monotone polygon



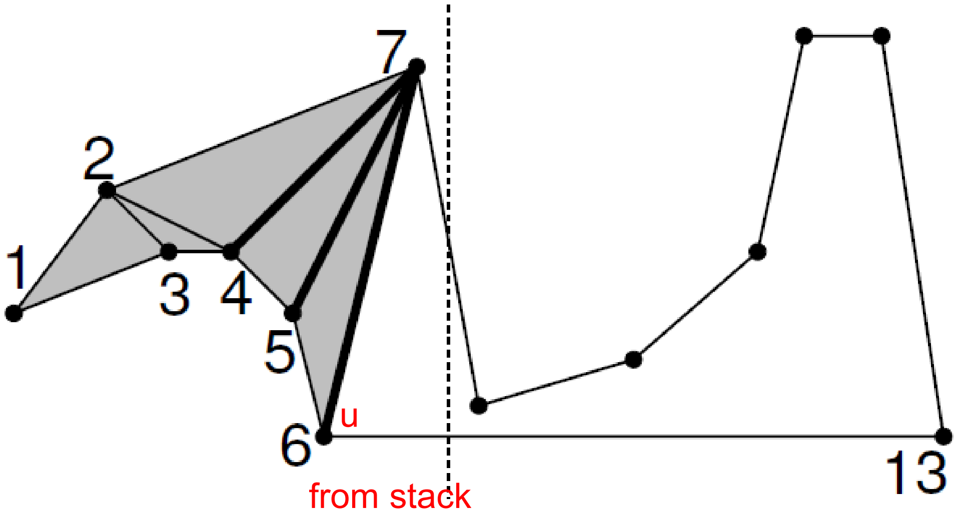
[Mount]

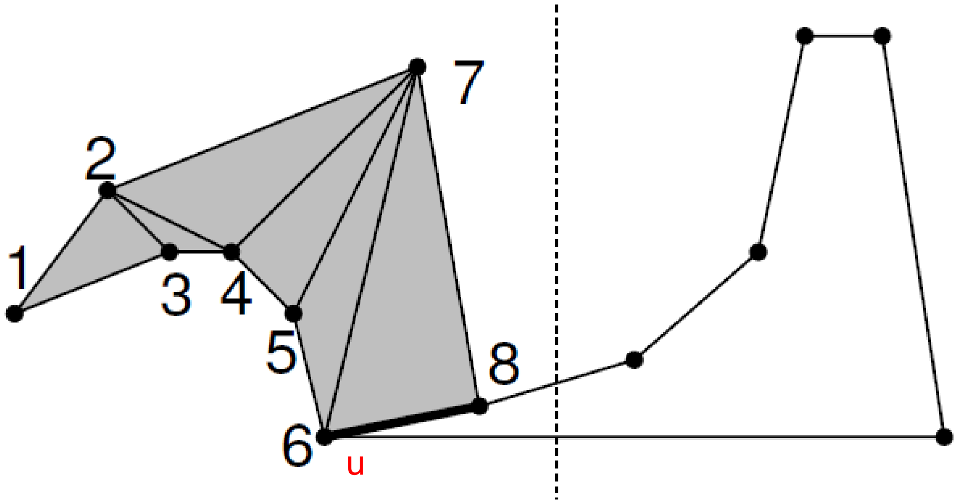
Triangulation of the monotone polygon

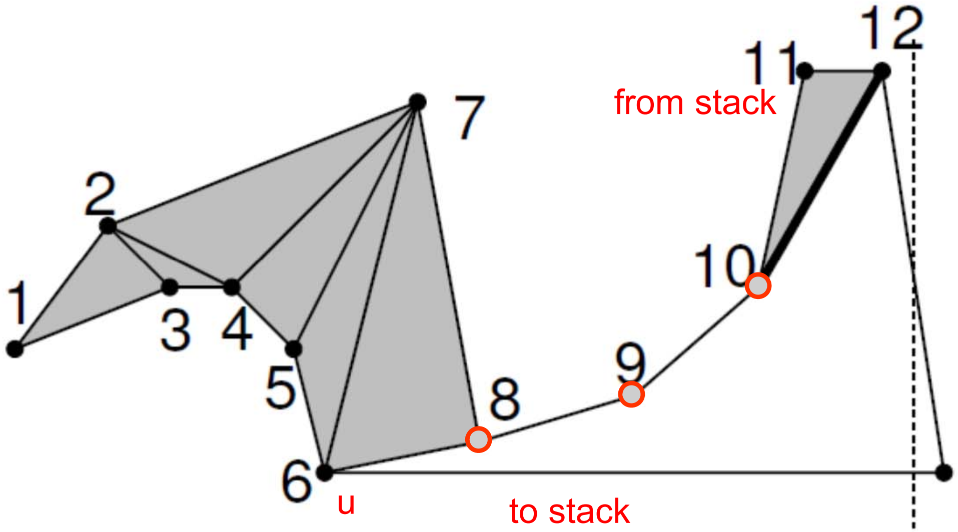


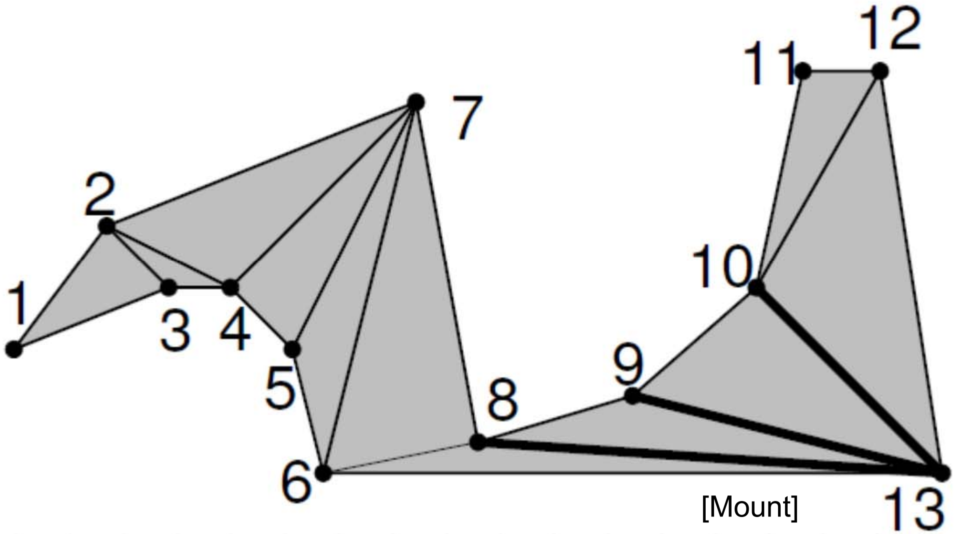
[Mount]

Triangulation of the monotone polygon



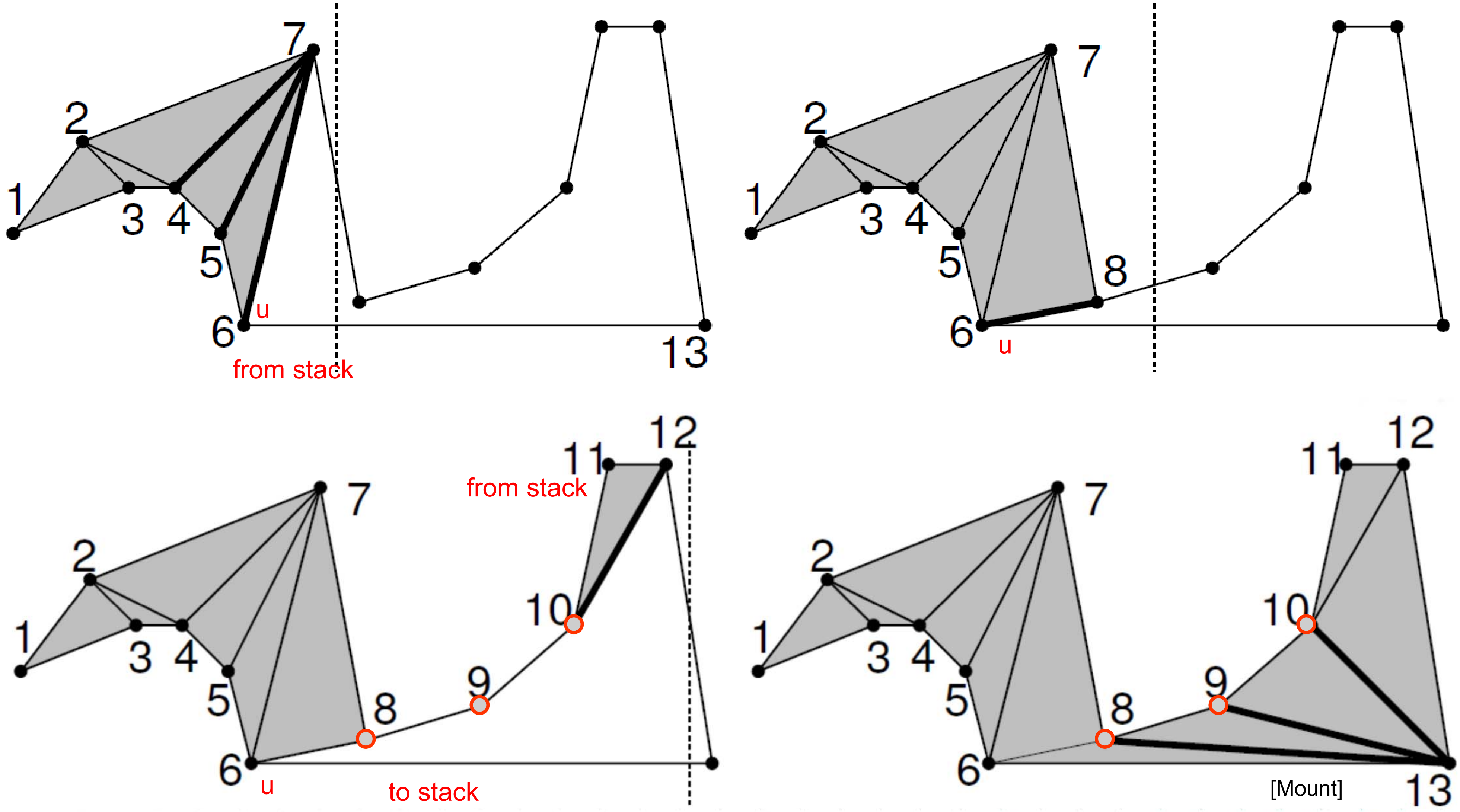




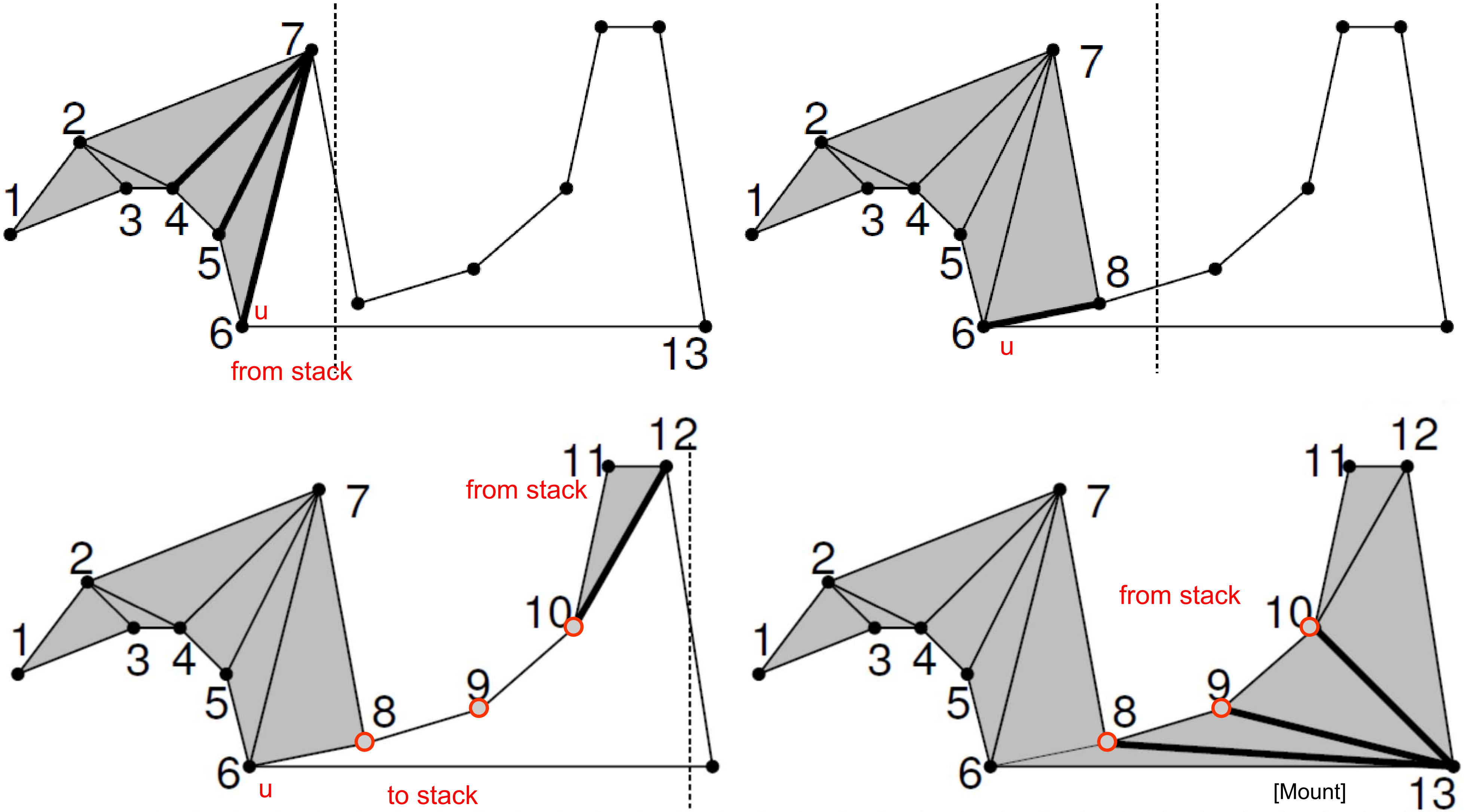


[Mount]

Triangulation of the monotone polygon

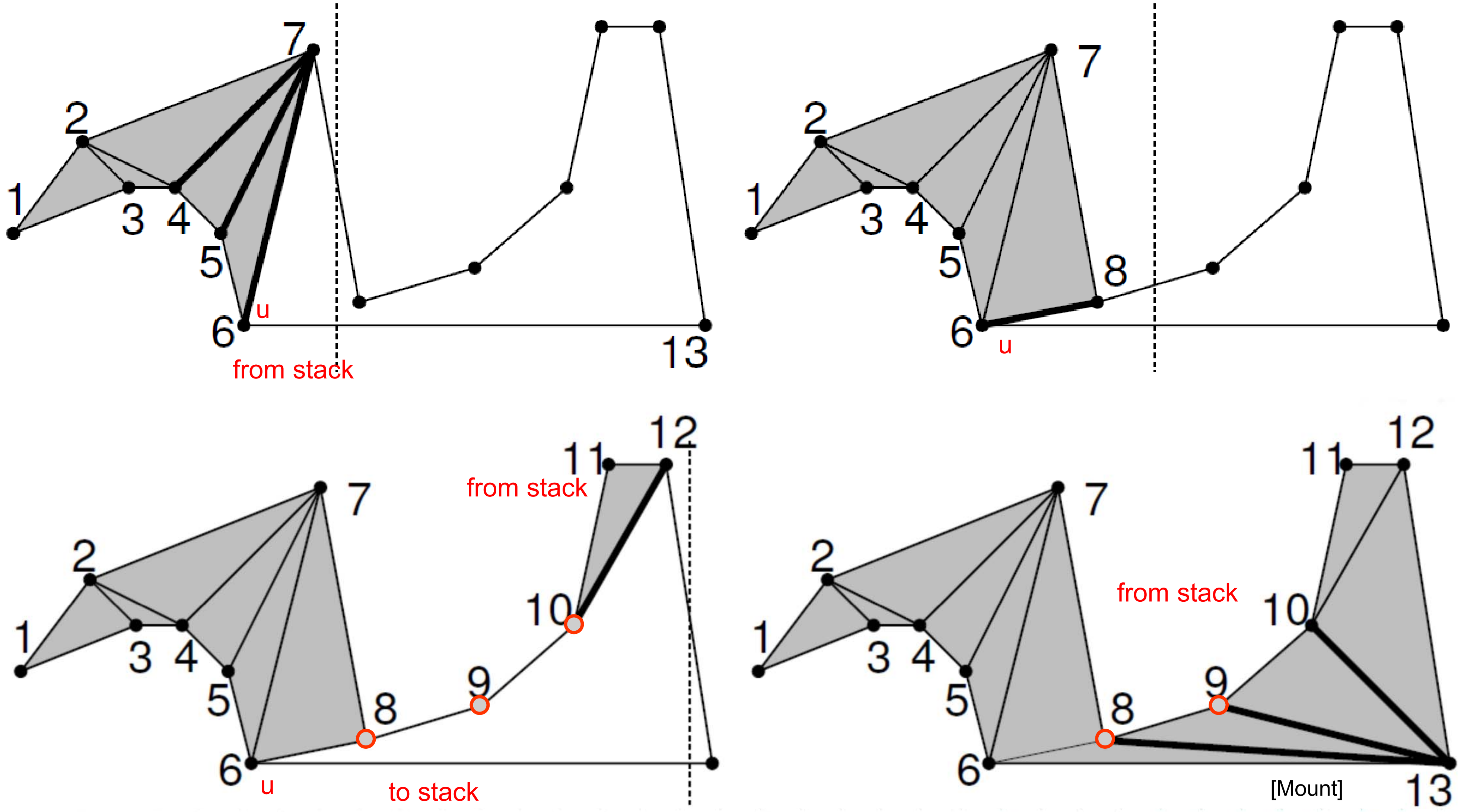


Triangulation of the monotone polygon



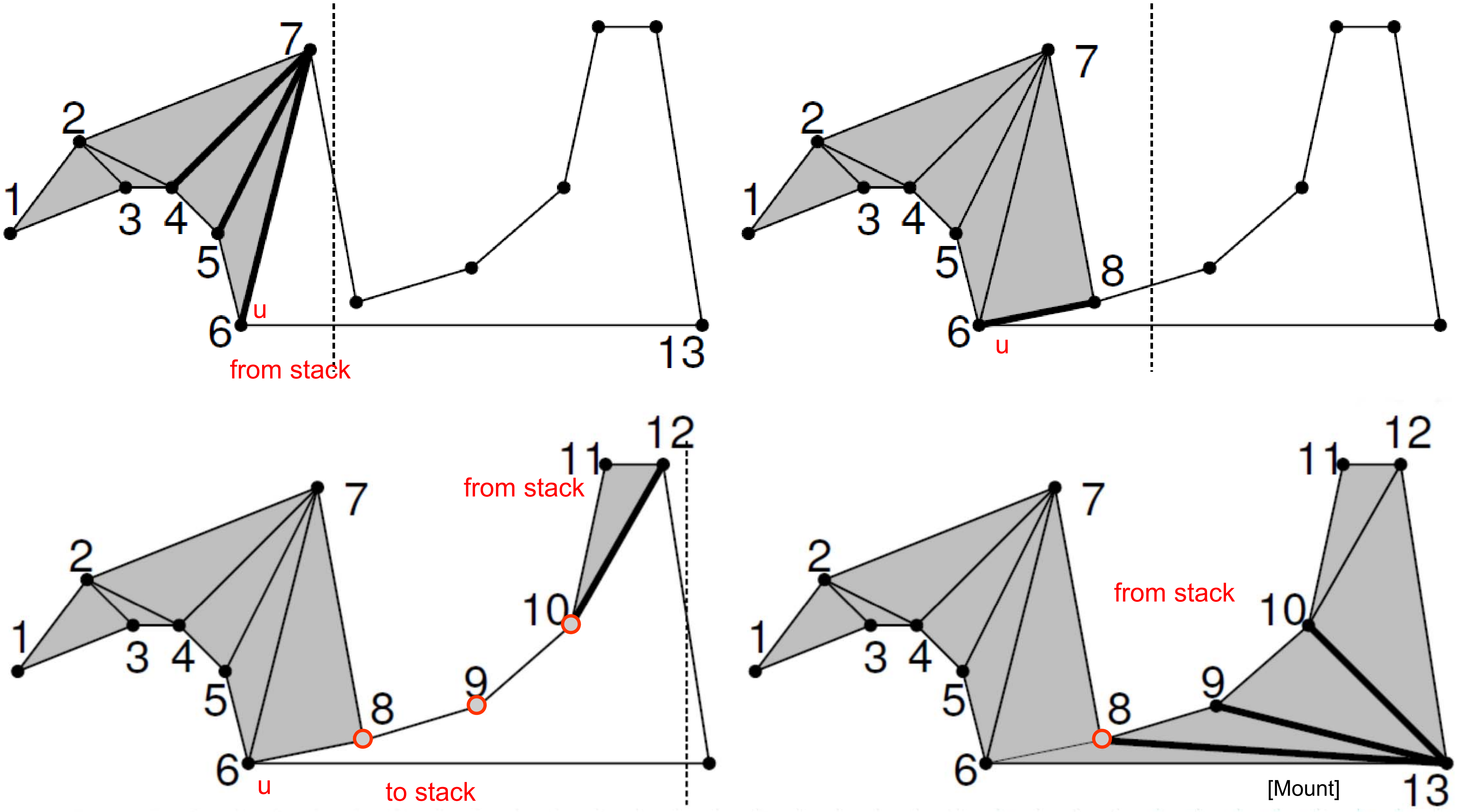
[Mount]

Triangulation of the monotone polygon



[Mount]

Triangulation of the monotone polygon



from stack

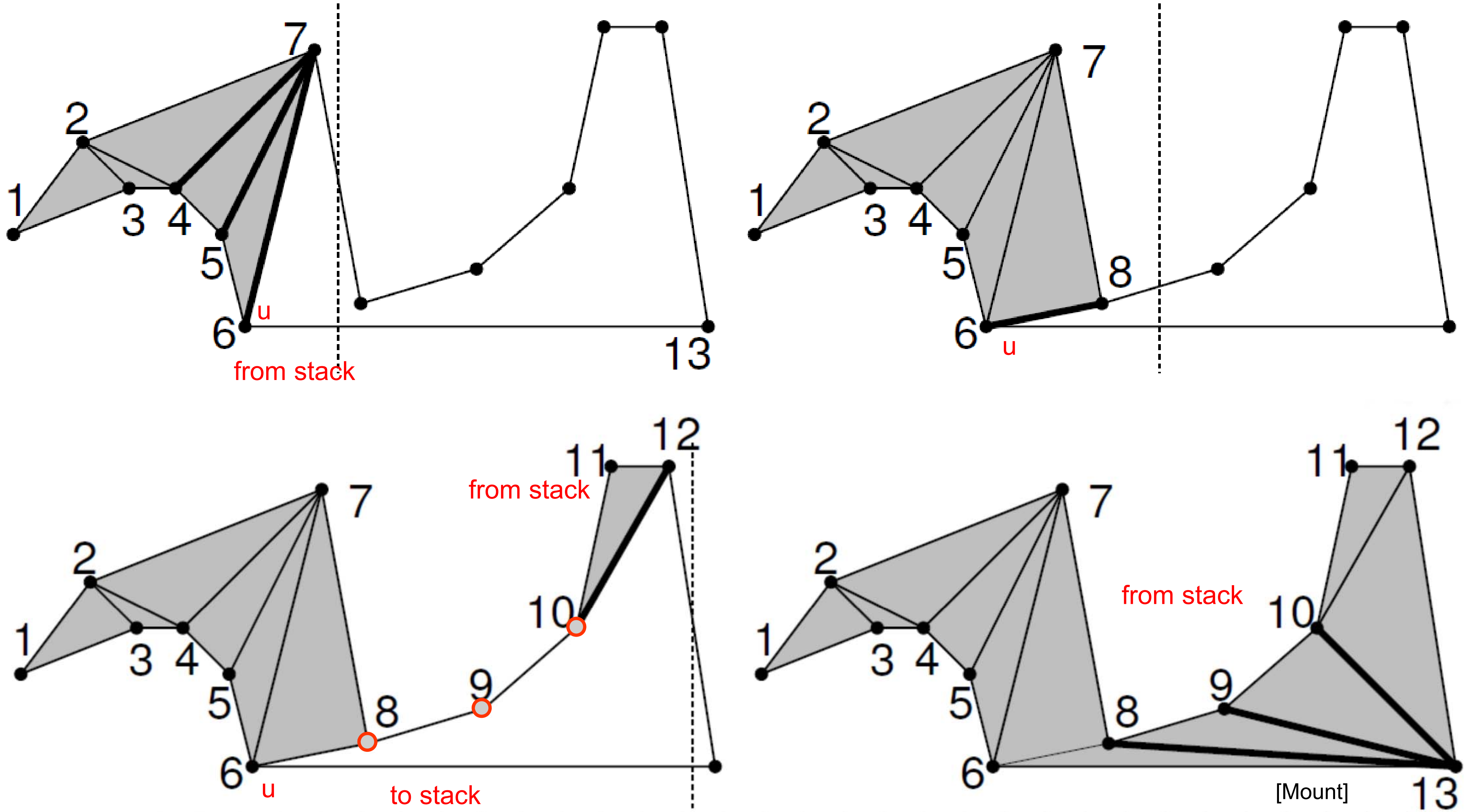
from stack

to stack

from stack

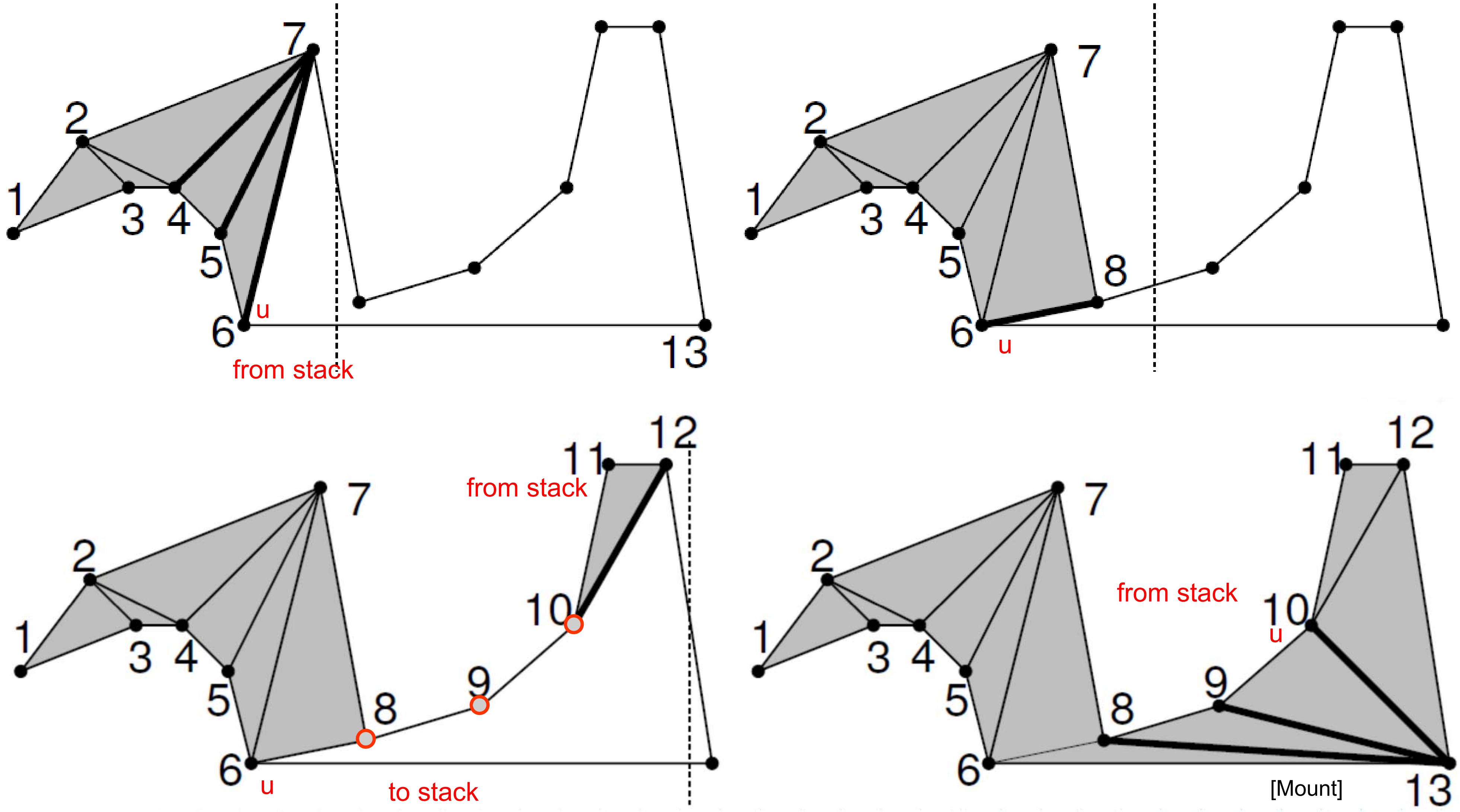
[Mount]

Triangulation of the monotone polygon



[Mount]

Triangulation of the monotone polygon

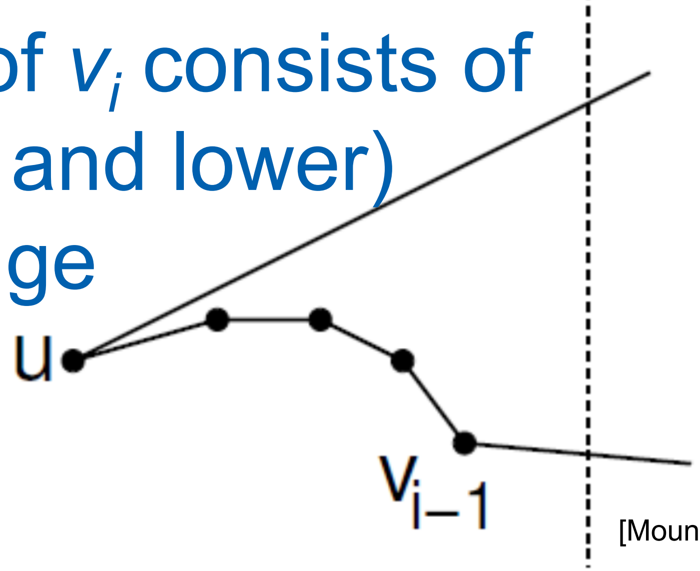


[Mount]

Main invariant of the untriangulated region

Main invariant

- Let v_i be the vertex being just processed
- The untriangulated region left of v_i consists of two x-monotone chains (upper and lower)
- Each chain has at least one edge
- If it has more than one edge
  - these edges form a reflex chain
    = sequence of vertices
    with interior angle ≥ 180°
  - the other chain consist of single edge $u\ v_i$
- Left vertex of the last added diagonal is u
- Vertices between u and v_i are waiting in the stack



[Mount]

Initial invariant

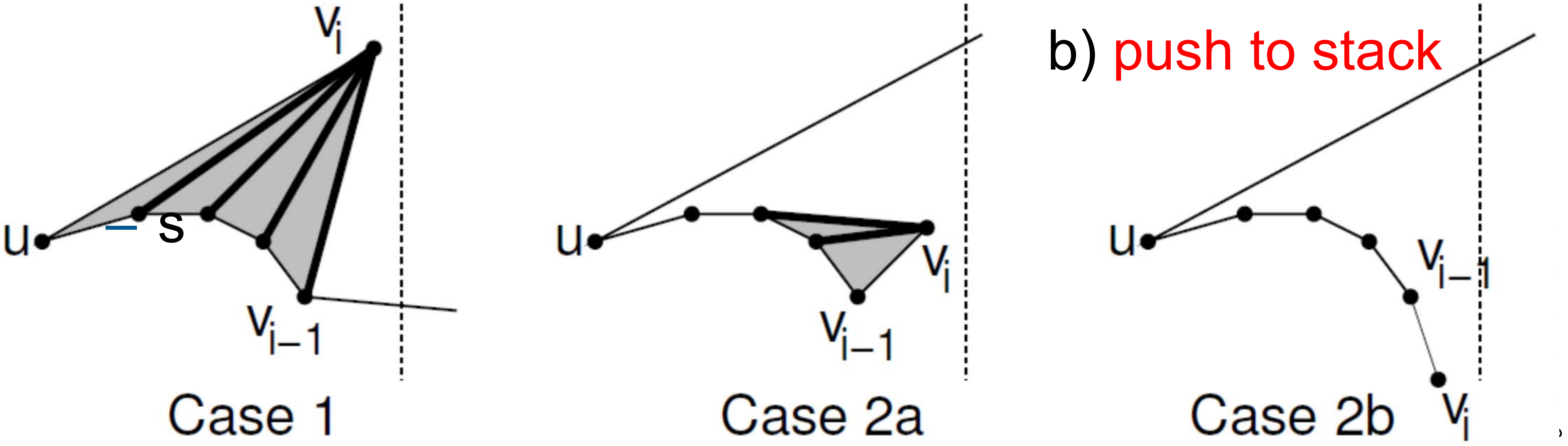
Triangulation cases

- Case 1: v_i lies on the opposite chain
  - Add diagonals from next(u) to v_{i-1} (empty the stack)
  - Set u = v_{i-1}. Last diagonal (invariant) is $v_i v_{i-1}$
- Case 2: v_i is on the same chain as v_{i-1}

  a) walk back, adding diagonals joining v_i to prior vertices until the angle becomes > 180° or *u* is reached - pop)

  b) push to stack

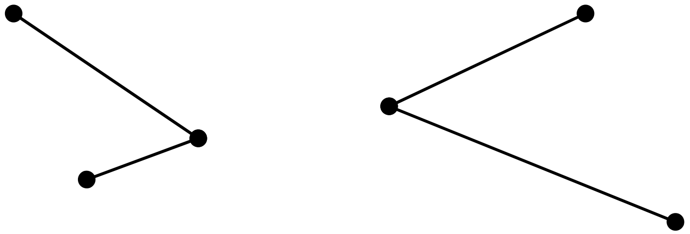
  – s



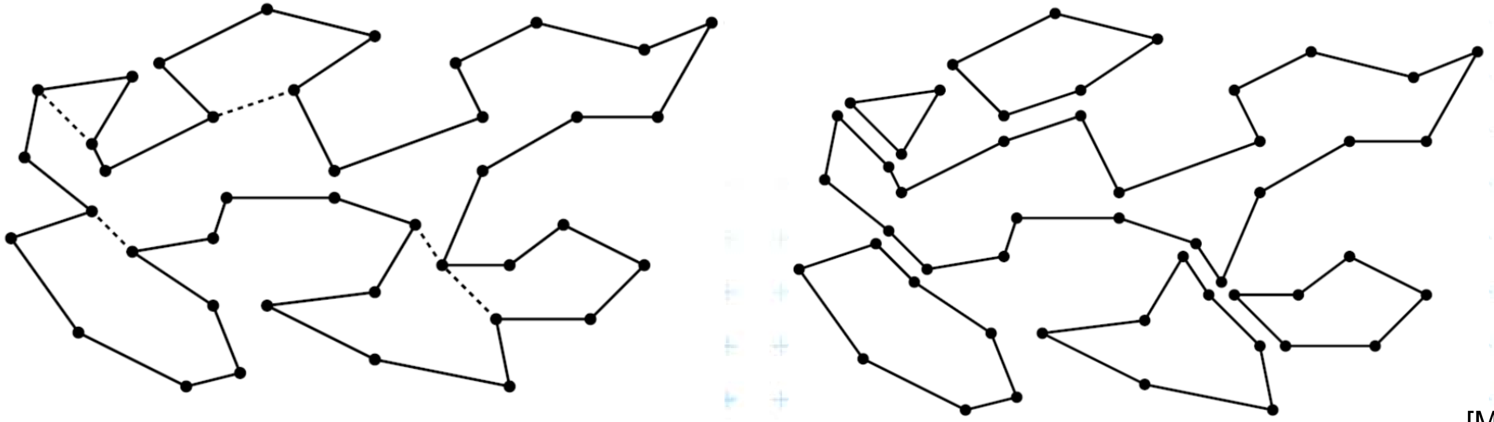
[Mount]

1. Polygon subdivision into monotone pieces

- X-monotonicity breaks the polygon in vertices with edges directed both left or both right



- The monotone polygons parts are separated by the splitting diagonals (joining vertex and helper)



Splitting diagonals

Monotone decomposition

[Mount]

Data structures for subdivision

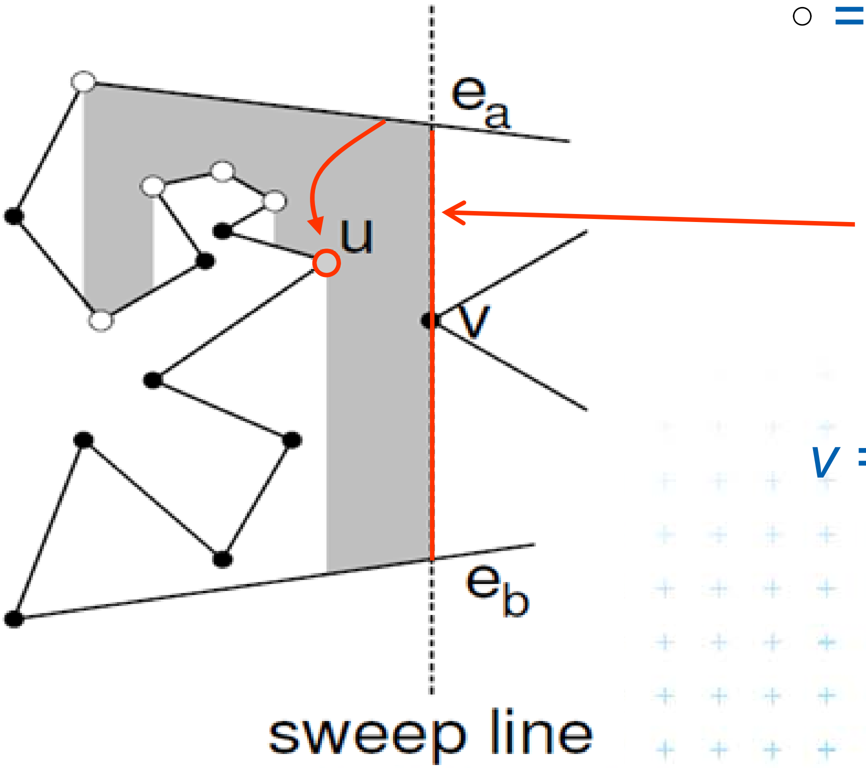
- Events
  - Endpoints of edges, known from the beginning
  - Can be stored in sorted list – no priority queue
- Sweep status
  - List of edges intersecting sweep line (top to bottom)
  - Stored in O(log n) time dictionary (like balanced tree)
- Event processing
  - Six event types based on local structure of edges around vertex *v*

Helper – definition

helper(e_a)

= the rightmost vertically visible processed vertex u below edge e_a on polygonal chain between edges e_a & e_b

is visible to every point along the sweep line between e_a & e_b



○ = vertically visible processed vertex

all these vertices

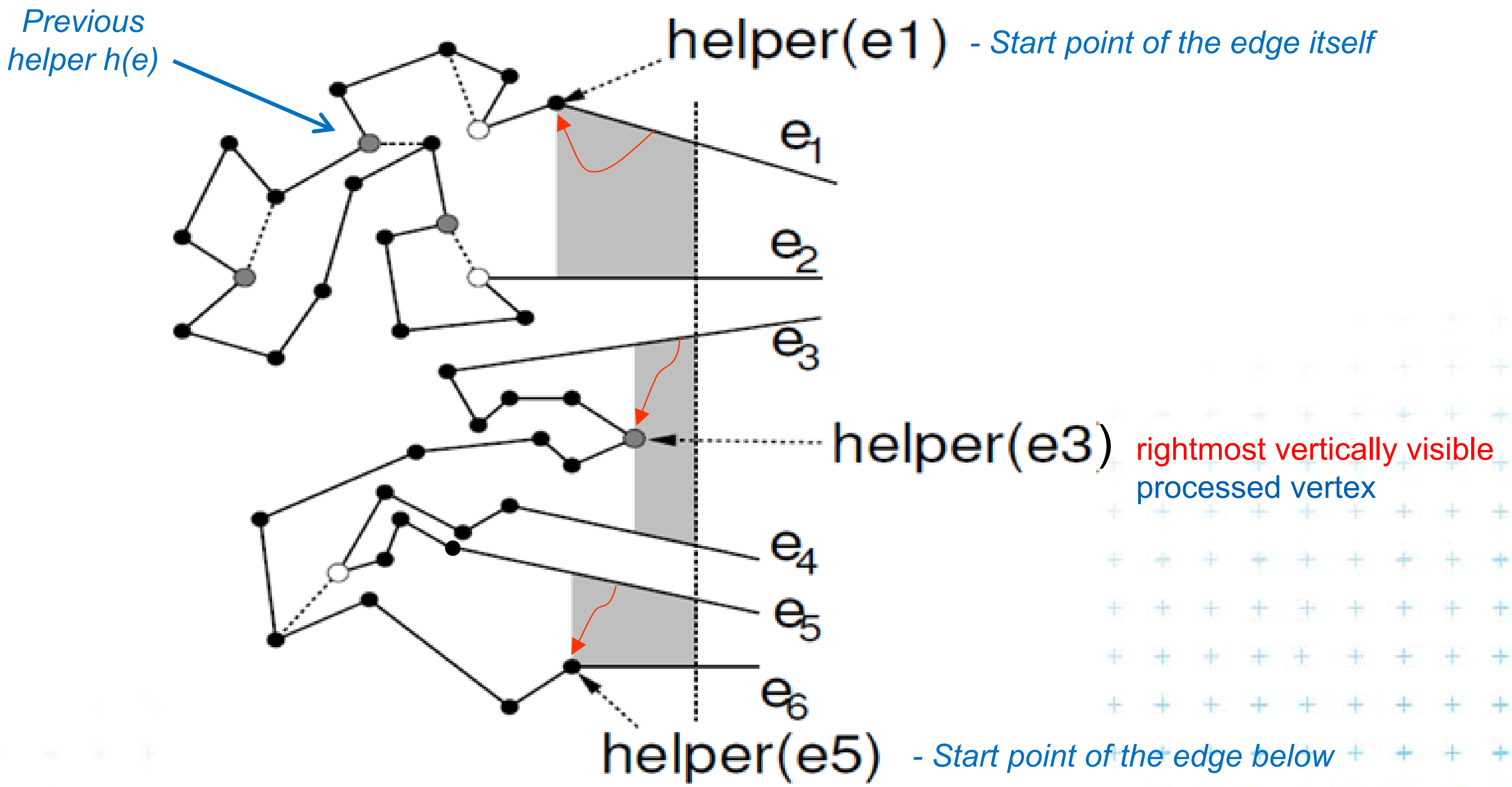
see ○ u = helper(e_a)

v = current vertex (sweep line stop)

Helper

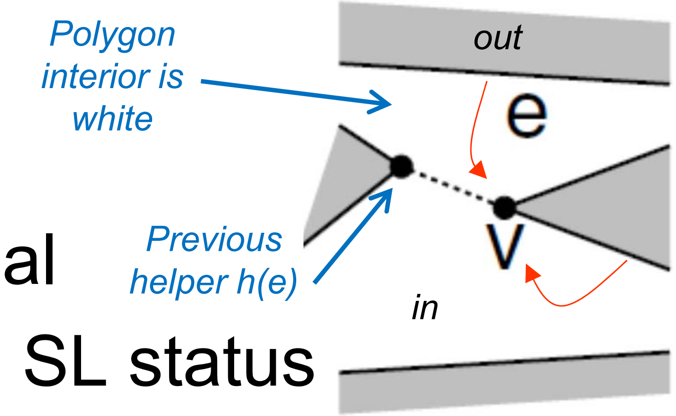
helper(e_a)

is defined only for edges intersected by the sweep line



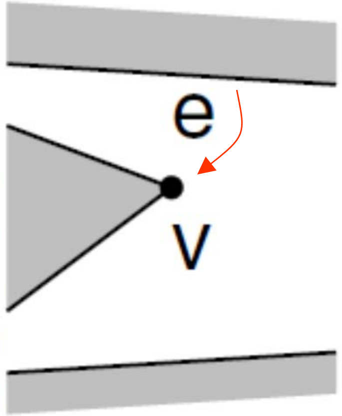
Six event types of vertex *v*

1. Split vertex



- Find edge *e* above *v*, connect *v* with helper(e) by diagonal
- Add 2 new edges incident to *v* into SL status
- Set new helper(e) = helper(lower edge of these two) = *v*

2. Merge vertex



- Find two edges incident with *v* in SL status
- Delete both from SL status
- Let *e* is edge immediately above *v*
- Make helper(e) = *v*

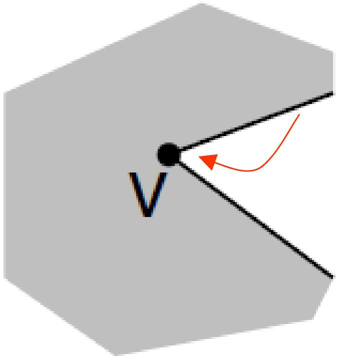
[Mount]

(Interior angle >180° for both – split & merge vertices)

Six event types of vertex *v*

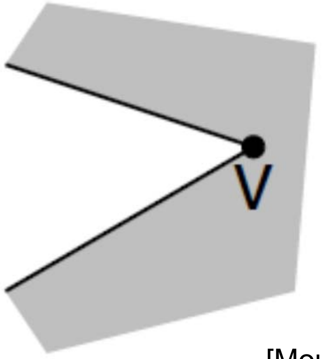
3. Start vertex

- Both incident edges lie right from *v*
- But interior angle <180°
- Insert both edges to SL status
- Set helper(upper edge) = *v*



4. End vertex

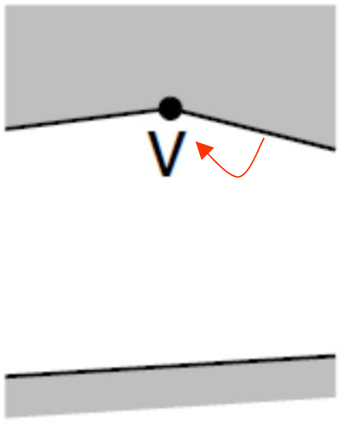
- Both incident edges lie left from *v*
- But interior angle <180°
- Delete both edges from SL status
- No helper set – we are out of the polygon



[Mount]

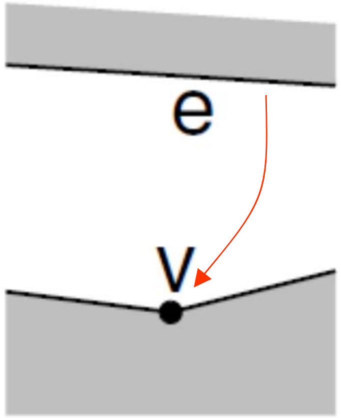
Six event types of vertex *v*

5. Upper chain-vertex



- one side is to the left, one side to the right, interior is below
- replace the left edge with the right edge in SL status
- Make *v* helper of the new (upper) edge

6. Lower chain-vertex



[Mount]

- one side is to the left, one side to the right, interior is above
- replace the left edge with the right edge in SL status
- Make *v* helper of the edge *e above*

Polygon subdivision complexity

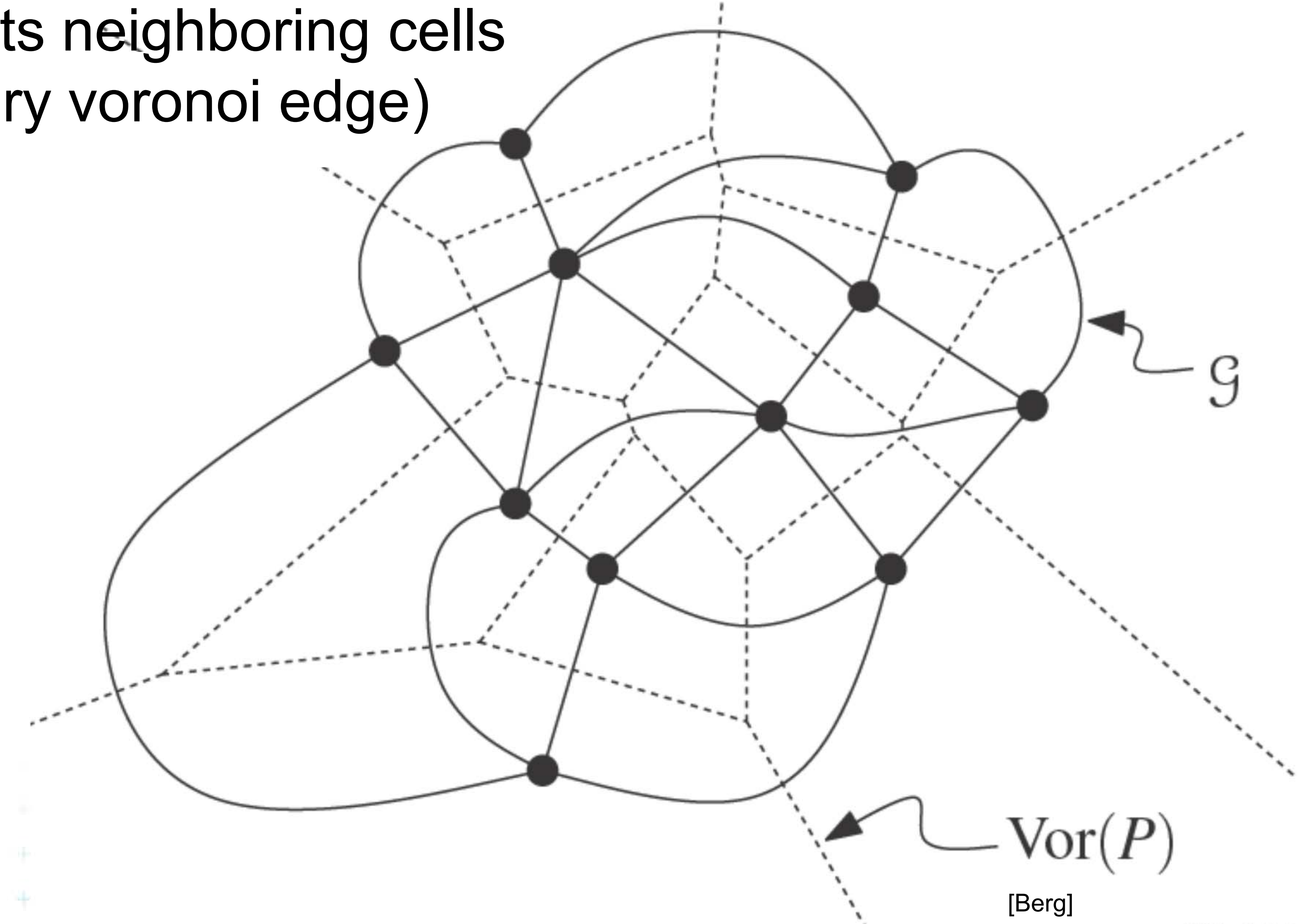
- Simple polygon with *n* vertices can be partitioned into x-monotone polygons in
  - O(*n* log *n*) time (n steps of SL, log n search each)
  - O(*n*) storage

- Complete simple polygon triangulation
  - O(*n* log *n*) time for partitioning into monotone polygons
  - O(*n*) time for triangulation
  - O(*n*) storage

Dual graph G for a Voronoi diagram

Graph G: Node for each Voronoi-diagram cell *V*(*p*) ~ VD site *p*

Arc connects neighboring cells
(arc for every voronoi edge)



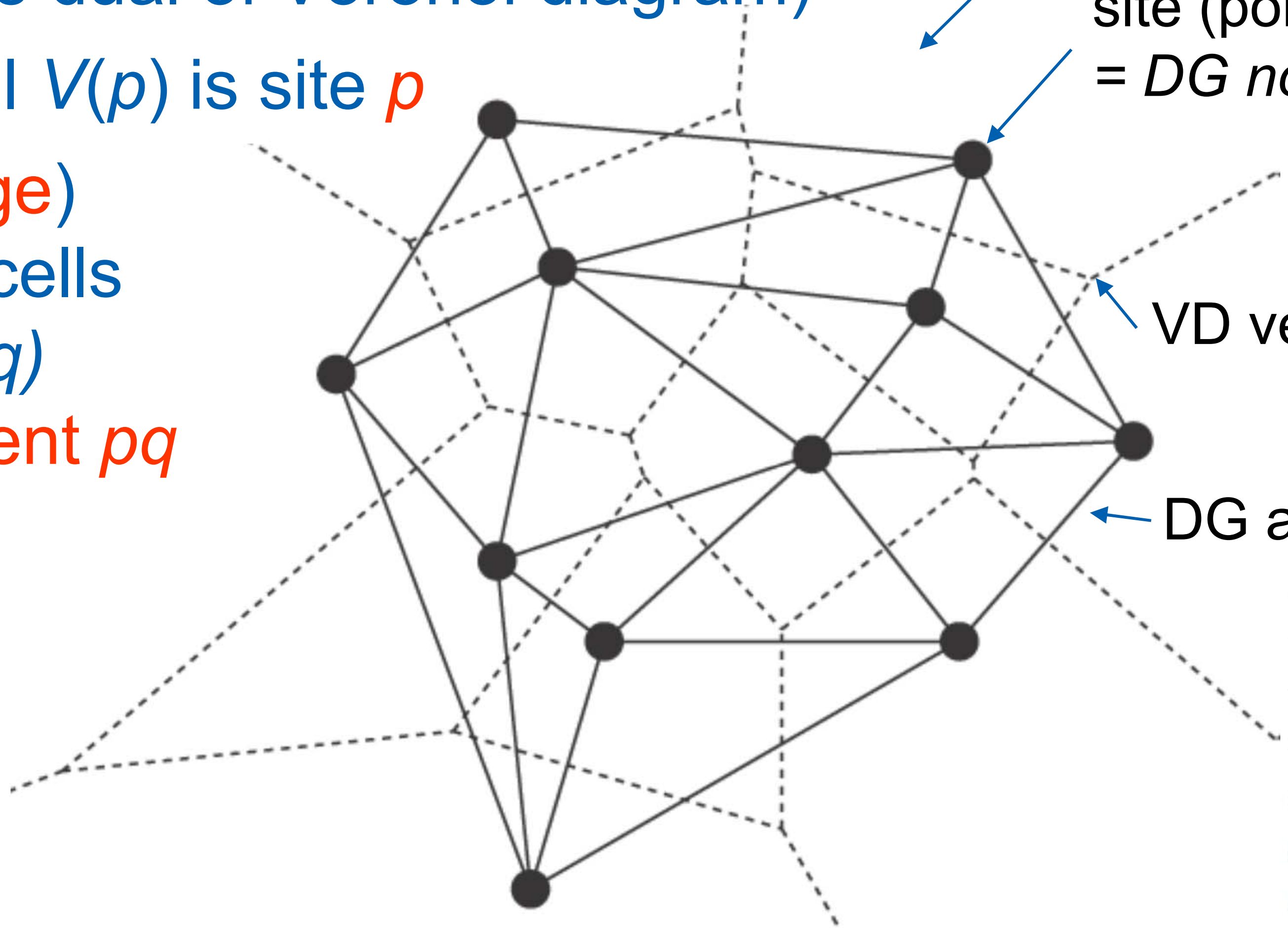
[Berg]

Delaunay graph *DG*(*P*)

[*Борис Николаевич Делоне*]

= straight line embedding of *G*
(straight-line dual of Voronoi diagram)

- Node for cell *V*(*p*) is site *p*
- Arc (DG edge) connecting cells *V(p) and V(q)* is the segment *pq*



VD cell V(*p*)

site (point) *p*
= *DG node*

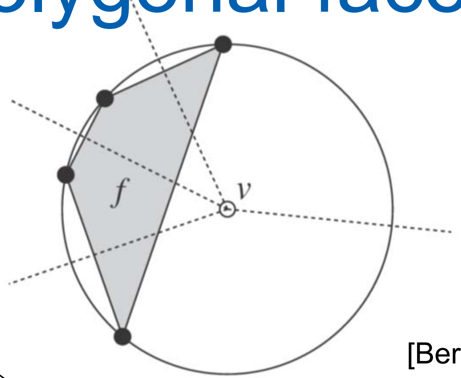
VD vertex

DG arc

[Berg]

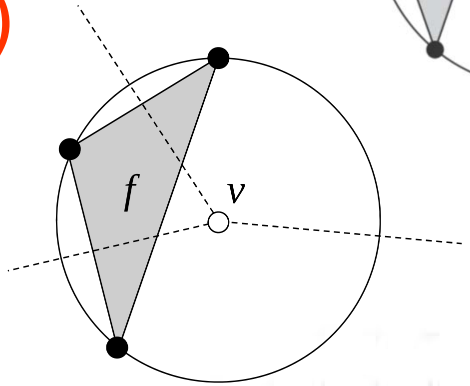
Delaunay graph and Delaunay triangulation

- Delaunay *graph DG(P)* has convex polygonal faces (with number of vertices ≥3, equal to the degree of Voronoi vertex)



[Berg]

- Delaunay *triangulation DT(P)* = Delaunay graph for sites in general position



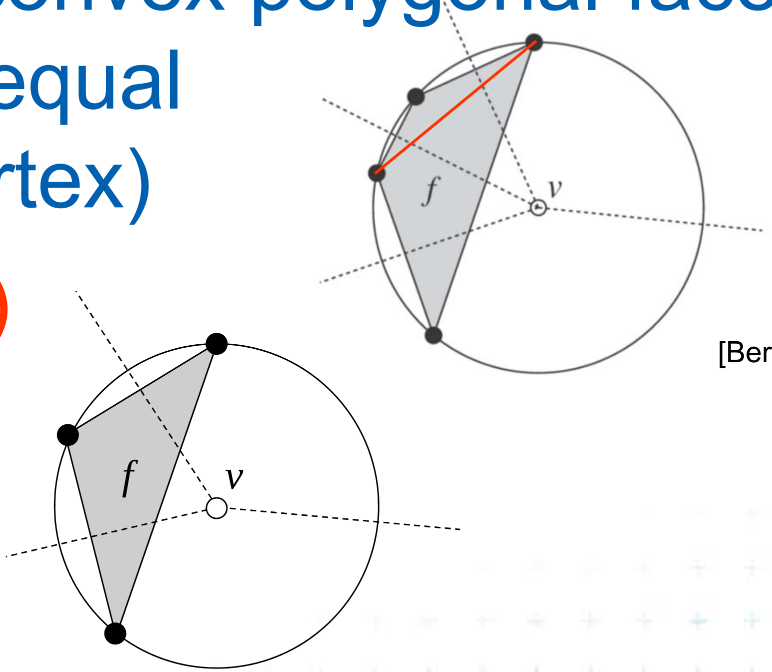
  - No four sites on a circle
  - Faces are triangles (Voronoi vertices have degree = 3)
  - DT is unique (DG not! Can be triangulated differently)

DG(P) sites not in general position

  - Triangulate larger faces – such triangulation is not unique

Delaunay graph and Delaunay triangulation

- Delaunay *graph DG(P)* has convex polygonal faces (with number of vertices ≥3, equal to the degree of Voronoi vertex)



[Berg]

- Delaunay *triangulation DT*(*P*) = Delaunay graph for sites in general position
  - No four sites on a circle
  - Faces are triangles (Voronoi vertices have degree = 3)
  - DT is unique (DG not! Can be triangulated differently)

DG(P) sites not in general position

- Triangulate larger faces – such triangulation is not unique

Delaunay triangulation properties 1/2

Circumcircle property

- The circumcircle of any triangle in DT is empty (no sites)
  Proof: It's center is the Voronoi vertex
- Three points *a*,*b*,*c* are vertices of the same face of *DG*(*P*)
  iff circle through *a*,*b*,*c* contains no point of *P* in its interior

Empty circle property and legal edge

- Two points *a*,*b* form an edge of *DG*(*P*) – it is a legal edge
  iff ∃ closed disc with *a*,*b* on its boundary that contains no other point of *P* in its interior … disc minimal diameter = dist(a,b)

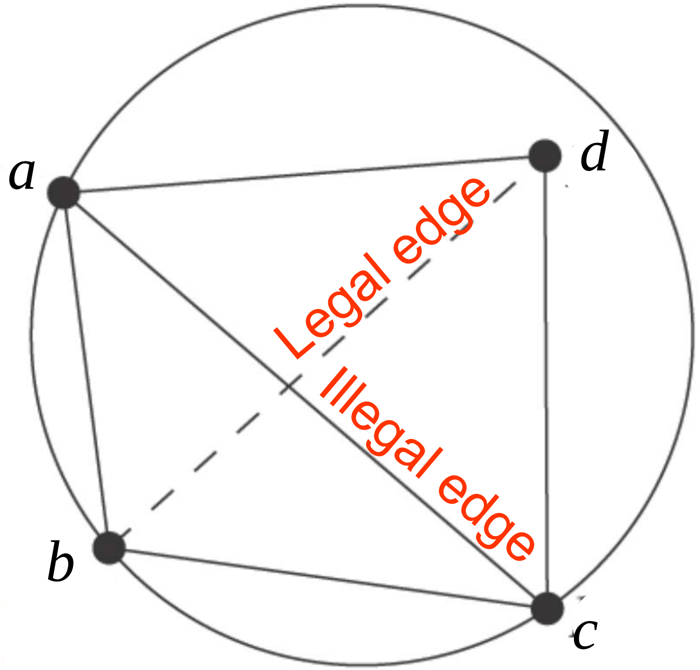
Closest pair property

- The closest pair of points in *P* are neighbors in *DT*(*P*)

Delaunay triangulation properties 2/2

- DT edges do not intersect
- Triangulation *T* is legal, **iff** *T* is a Delaunay triangulation (i.e., if it does not contain illegal edges)
- Edge that was legal before may become illegal if one of the triangles incident to it changes
- In convex quadrilateral *abcd* (*abcd* do not lie on common circle) exactly one of *ac, bd*
  - is an illegal edge
  - and the other edge is legal

= principle of edge flip operation

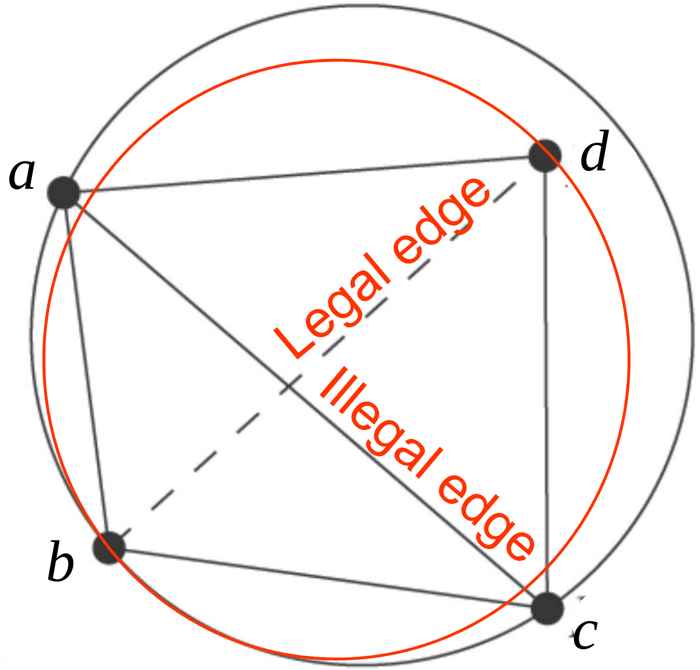


[Berg]

Delaunay triangulation properties 2/2

- DT edges do not intersect
- Triangulation *T* is legal, **iff** *T* is a Delaunay triangulation (i.e., if it does not contain illegal edges)
- Edge that was legal before may become illegal if one of the triangles incident to it changes
- In convex quadrilateral *abcd* (*abcd* do not lie on common circle) exactly one of *ac, bd*
  is an illegal edge
  and the other edge is legal

  ≡ principle of edge flip operation



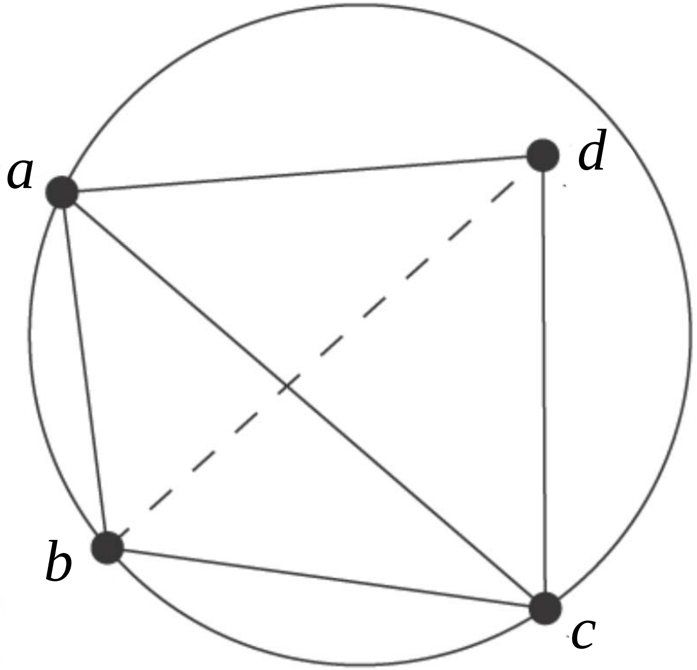
[Berg]

Edge flip operation

Edge flip

= a local operation, that increases the angle vector

- Given two adjacent triangles $\triangle abc$ and $\triangle cda$ such that their union forms a convex quadrilateral, the edge flip operation replaces the diagonal *ac* with *bd*.



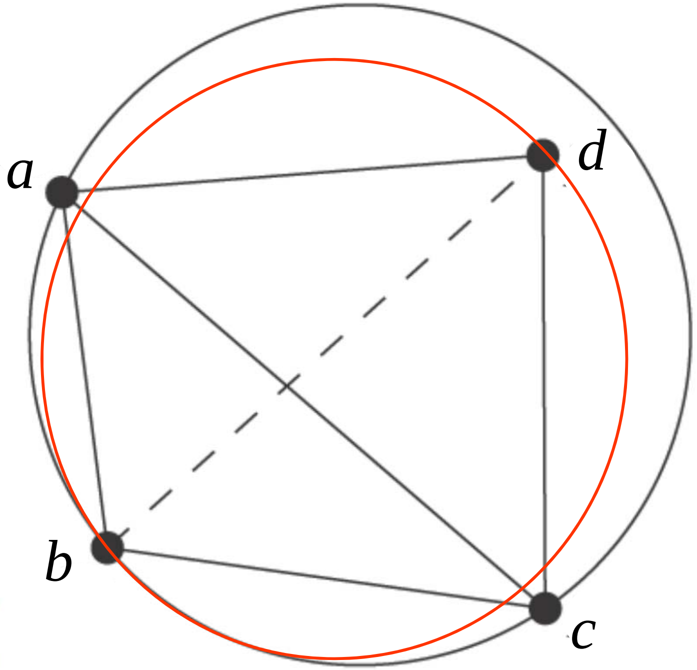
[Berg]

Edge flip operation

Edge flip

= a local operation, that increases the angle vector

- Given two adjacent triangles $\triangle abc$ and $\triangle cda$ such that their union forms a convex quadrilateral, the edge flip operation replaces the diagonal *ac* with *bd*.



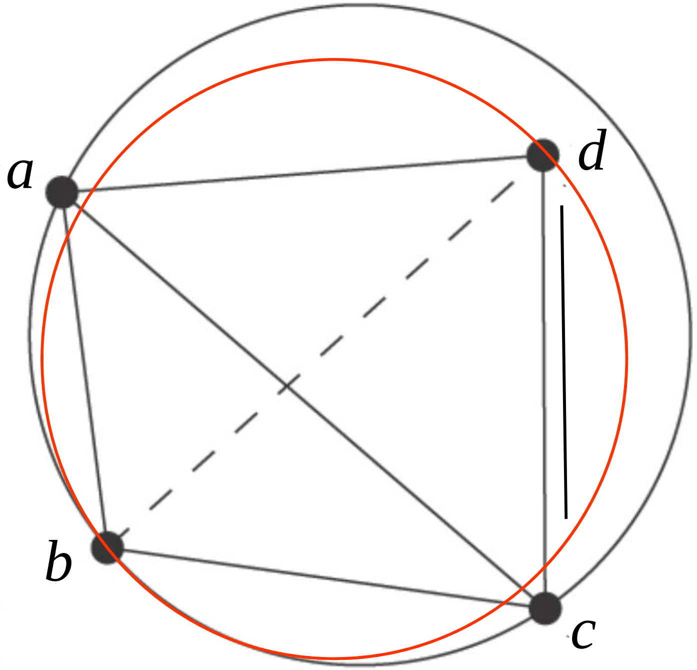
[Berg]

Edge flip operation

Edge flip

= a local operation, that increases the angle vector

- Given two adjacent triangles $\triangle abc$ and $\triangle cda$ such that their union forms a convex quadrilateral, the edge flip operation replaces the diagonal *ac* with *bd*.



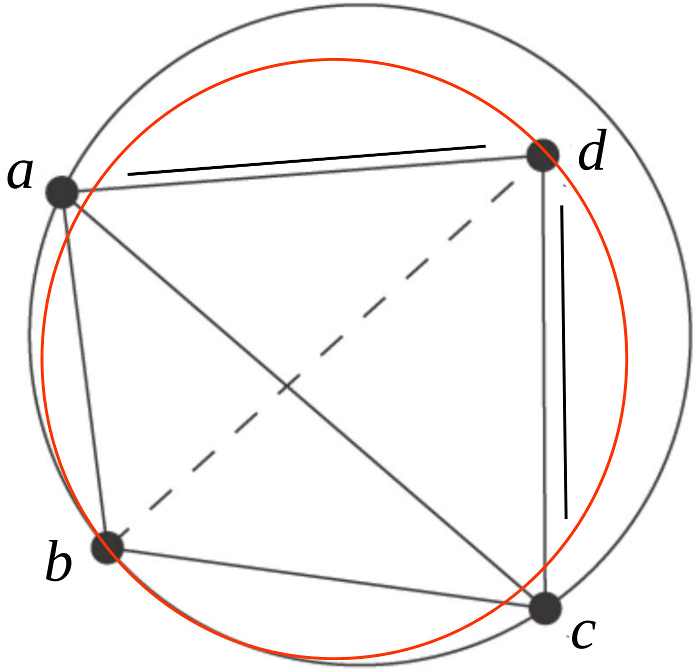
[Berg]

Edge flip operation

Edge flip

= a local operation, that increases the angle vector

- Given two adjacent triangles $\triangle abc$ and $\triangle cda$ such that their union forms a convex quadrilateral, the edge flip operation replaces the diagonal *ac* with *bd*.



[Berg]

Delaunay triangulation

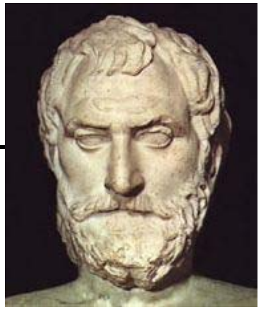
- Let *T* be a triangulation with *m* triangles (and 3*m angles*)
- Angle-vector

  = non-decreasing ordered sequence $(\alpha_1, \alpha_2, \ldots , \alpha_{3m})$ inner angles of triangles, $\alpha_i \leq \alpha_j$, for $i < j$
- In the plane, Delaunay triangulation has the lexicographically largest angle sequence
  - It maximizes the minimal angle (the first angle in angle-vector)
  - It maximizes the second minimal angle, …
  - It maximizes all angles
  - It is an angle sequence optimal triangulation

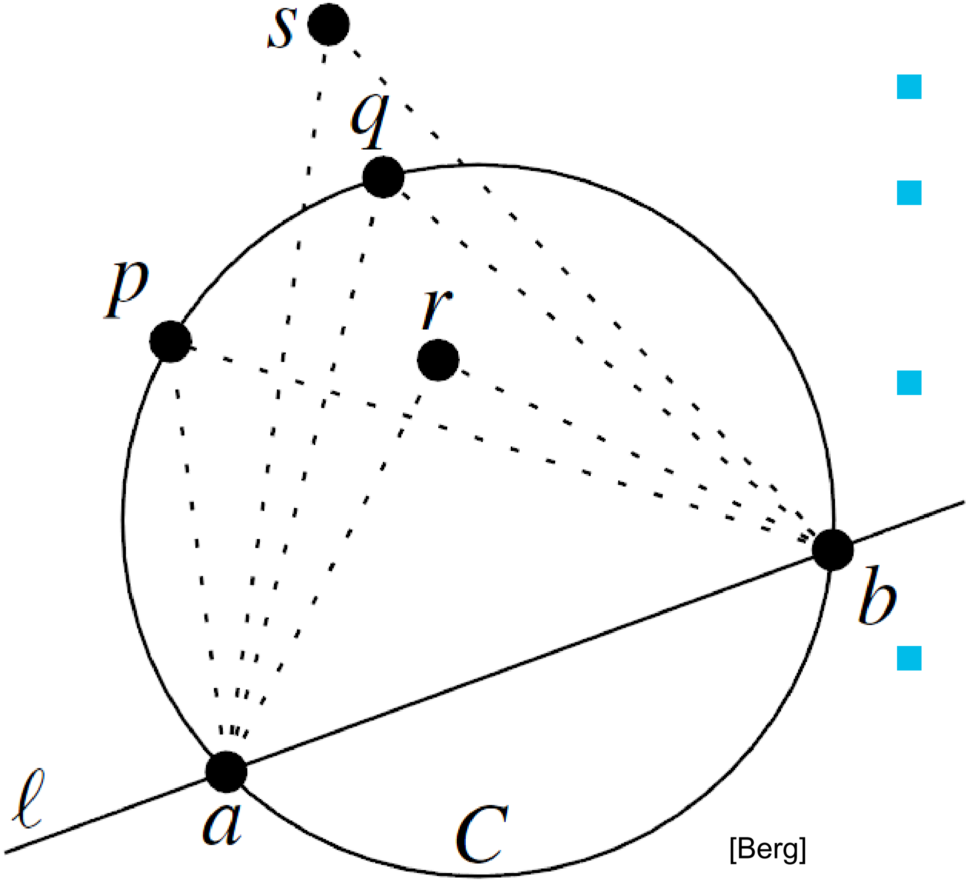
Delaunay triangulation

- It maximizes the minimal angle
  - The smallest angle in the DT is at least as large as the smallest angle in any other triangulation.
- However, the Delaunay triangulation
  - does not necessarily minimize the maximum angle.[4]
  - does not necessarily minimize the length of the edges.

Thales's theorem (624-546 BC)



Respective Central Angle Theorem



[Berg]

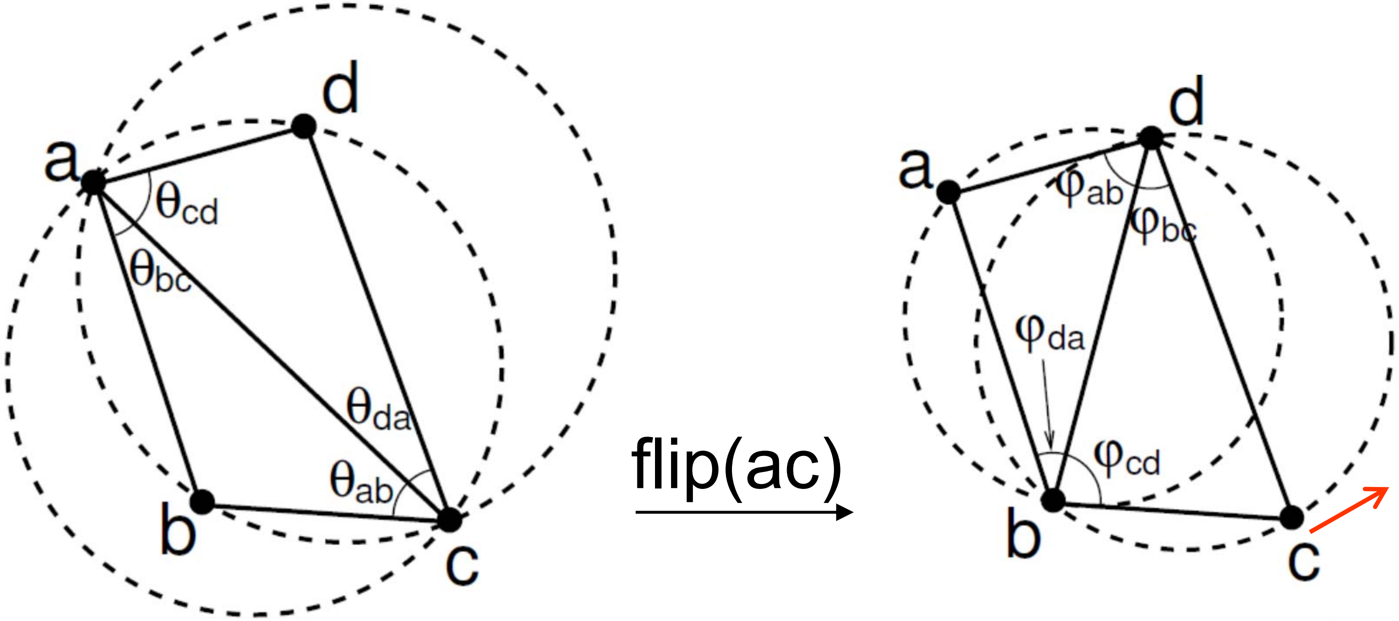
- Let C = circle,
- l =line intersecting C in points a, b
- p, q, r, s = points on the same side of l *p,q* on C , r is *in,* s is out
- Then for the angles holds: $\sphericalangle arb > \sphericalangle apb = \sphericalangle aqb > \sphericalangle asb$

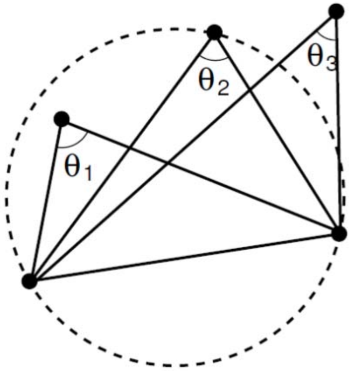
http://www.mathopenref.com/arccentralangletheorem.html

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd





$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac|$ $\varphi_{ab} > \theta_{ab}$ $\varphi_{bc} > \theta_{bc}$ $\varphi_{cd} > \theta_{cd}$ $\varphi_{da} > \theta_{da}$

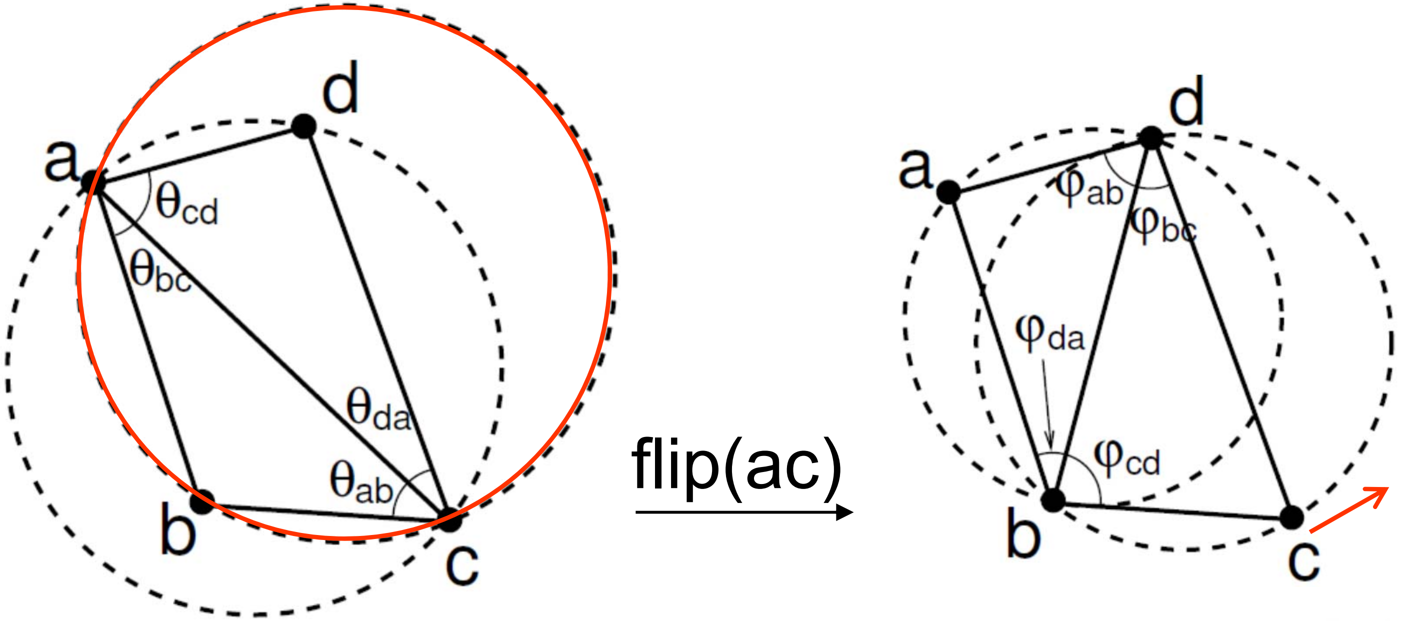
=> After limited number of edge flips

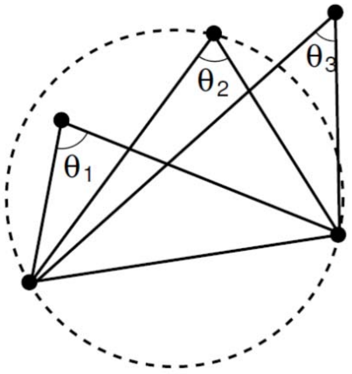
- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd





$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac| \qquad \varphi_{ab} > \theta_{ab} \qquad \varphi_{bc} > \theta_{bc} \qquad \varphi_{cd} > \theta_{cd} \qquad \varphi_{da} > \theta_{da}$

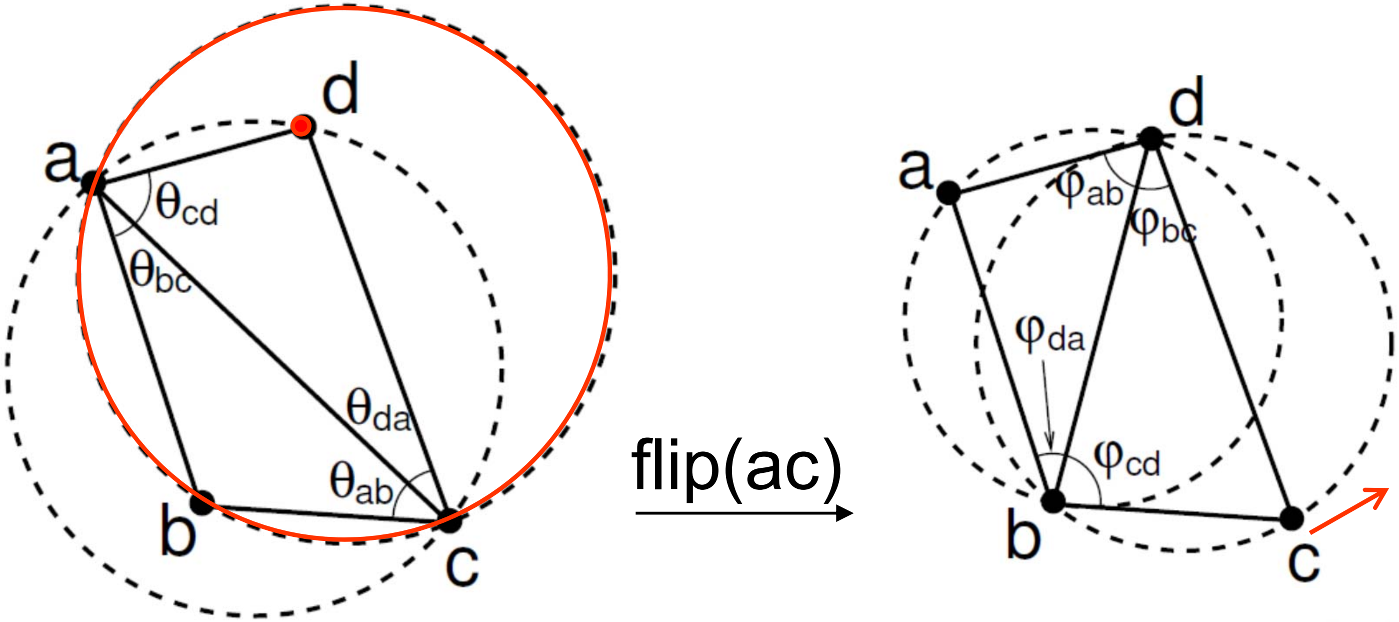
=> After limited number of edge flips

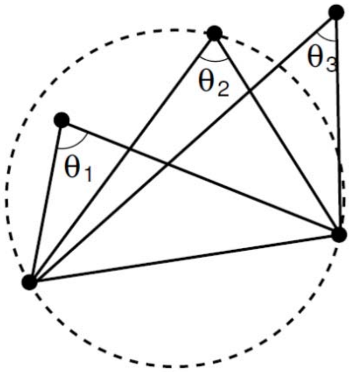
- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd





$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac|$ $\varphi_{ab} > \theta_{ab}$ $\varphi_{bc} > \theta_{bc}$ $\varphi_{cd} > \theta_{cd}$ $\varphi_{da} > \theta_{da}$

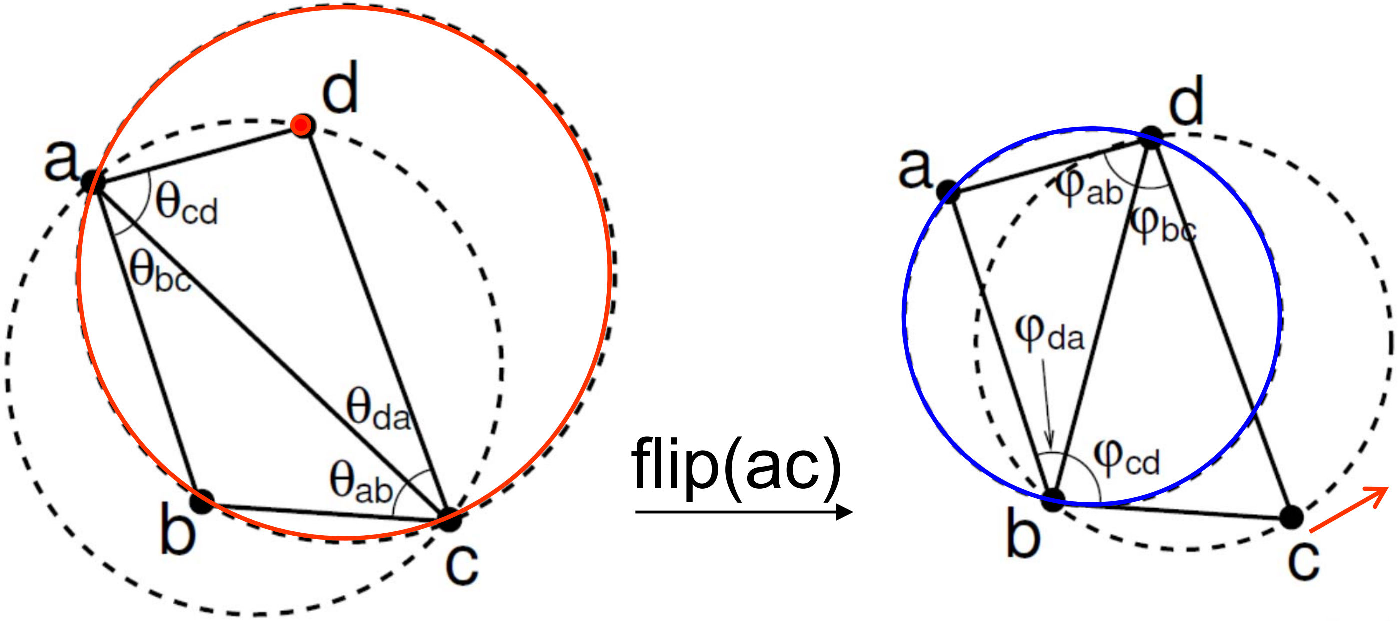
=> After limited number of edge flips

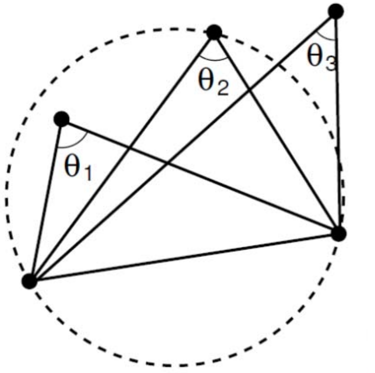
- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd





$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac| \quad \varphi_{ab} > \theta_{ab} \quad \varphi_{bc} > \theta_{bc} \quad \varphi_{cd} > \theta_{cd} \quad \varphi_{da} > \theta_{da}$

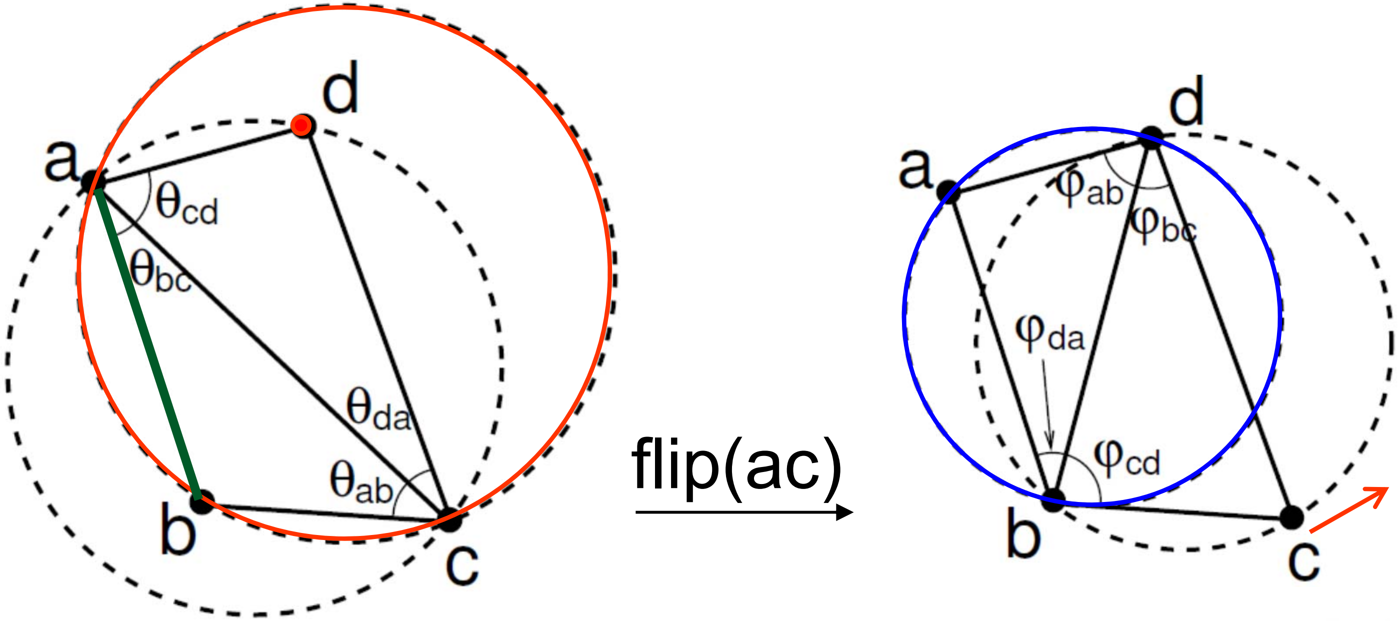
=> After limited number of edge flips

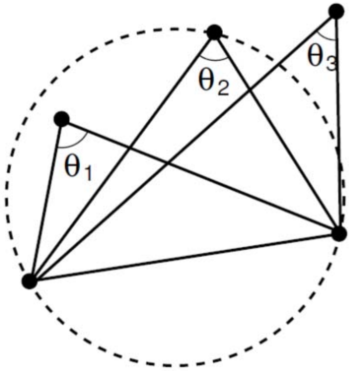
- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd





$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac|$ $\varphi_{ab} > \theta_{ab}$ $\varphi_{bc} > \theta_{bc}$ $\varphi_{cd} > \theta_{cd}$ $\varphi_{da} > \theta_{da}$

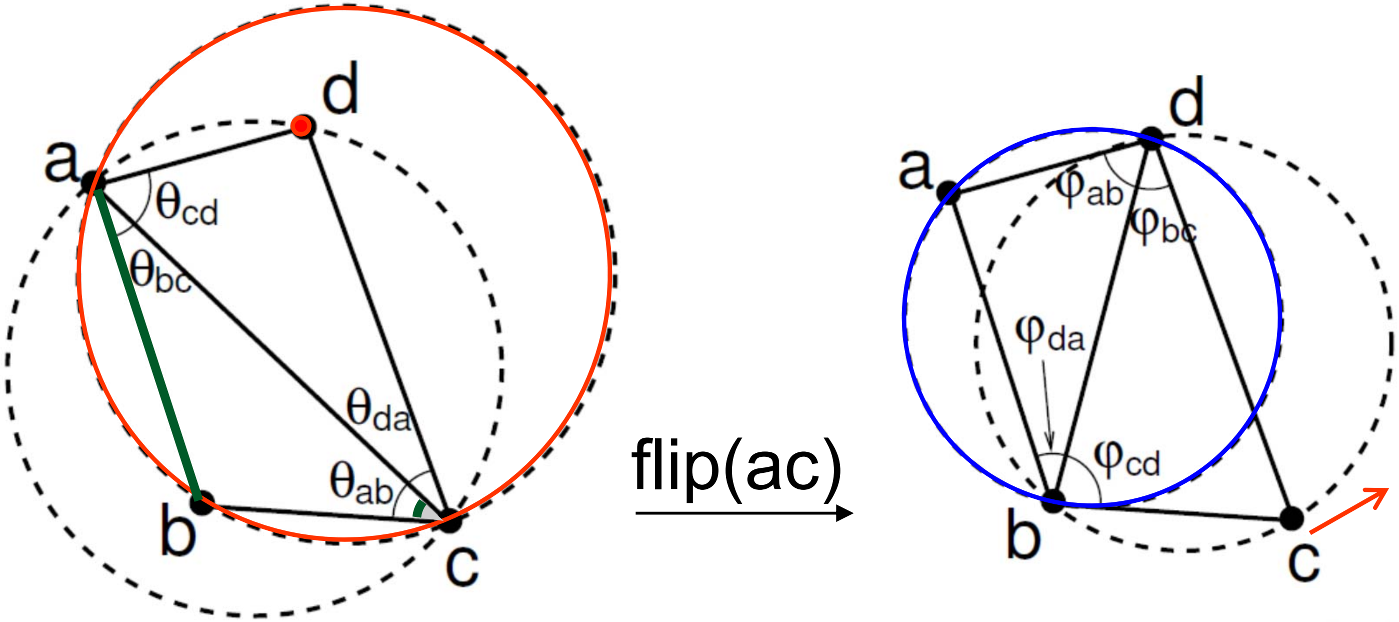
=> After limited number of edge flips

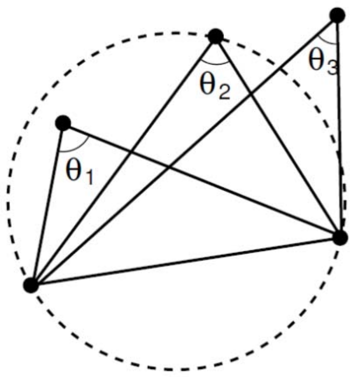
- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd





$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac|$ $\varphi_{ab} > \theta_{ab}$ $\varphi_{bc} > \theta_{bc}$ $\varphi_{cd} > \theta_{cd}$ $\varphi_{da} > \theta_{da}$

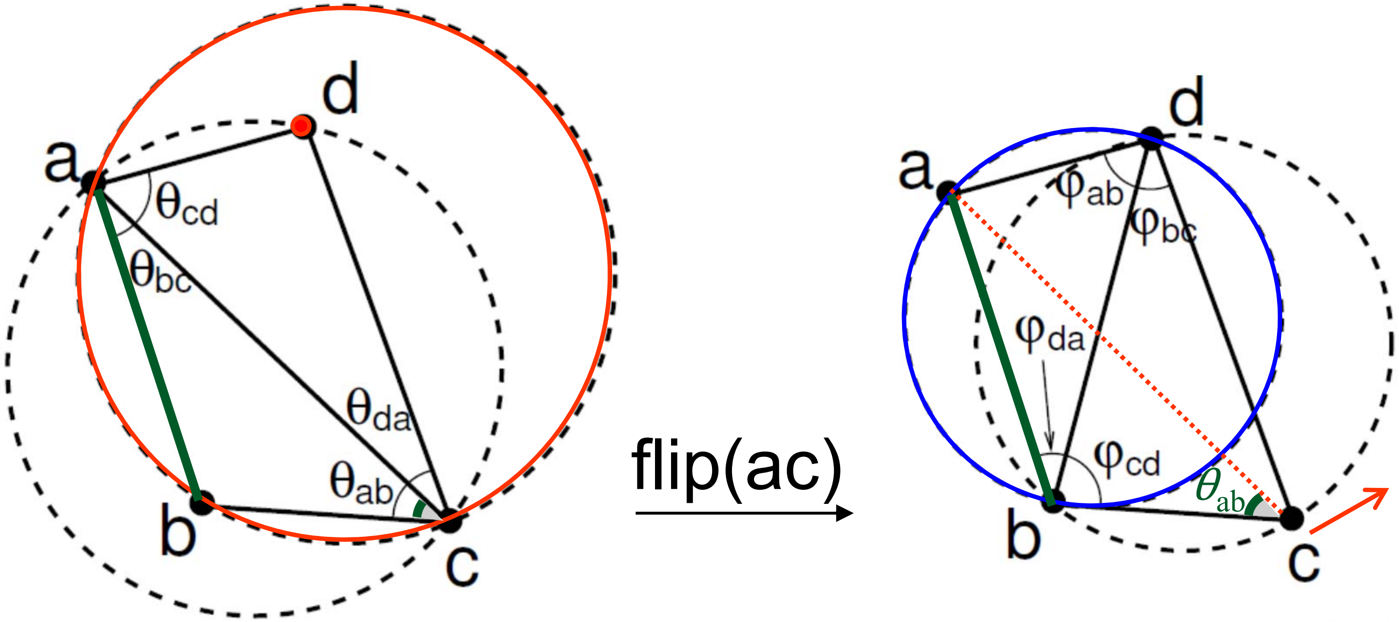
=> After limited number of edge flips

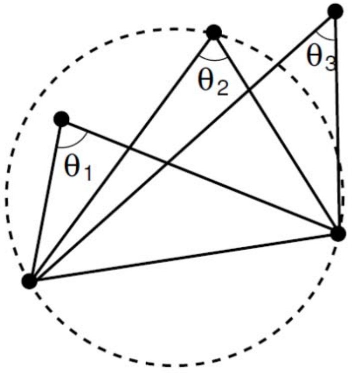
- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd





$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac|$ $\quad \varphi_{ab} > \theta_{ab} \quad \varphi_{bc} > \theta_{bc} \quad \varphi_{cd} > \theta_{cd} \quad \varphi_{da} > \theta_{da}$

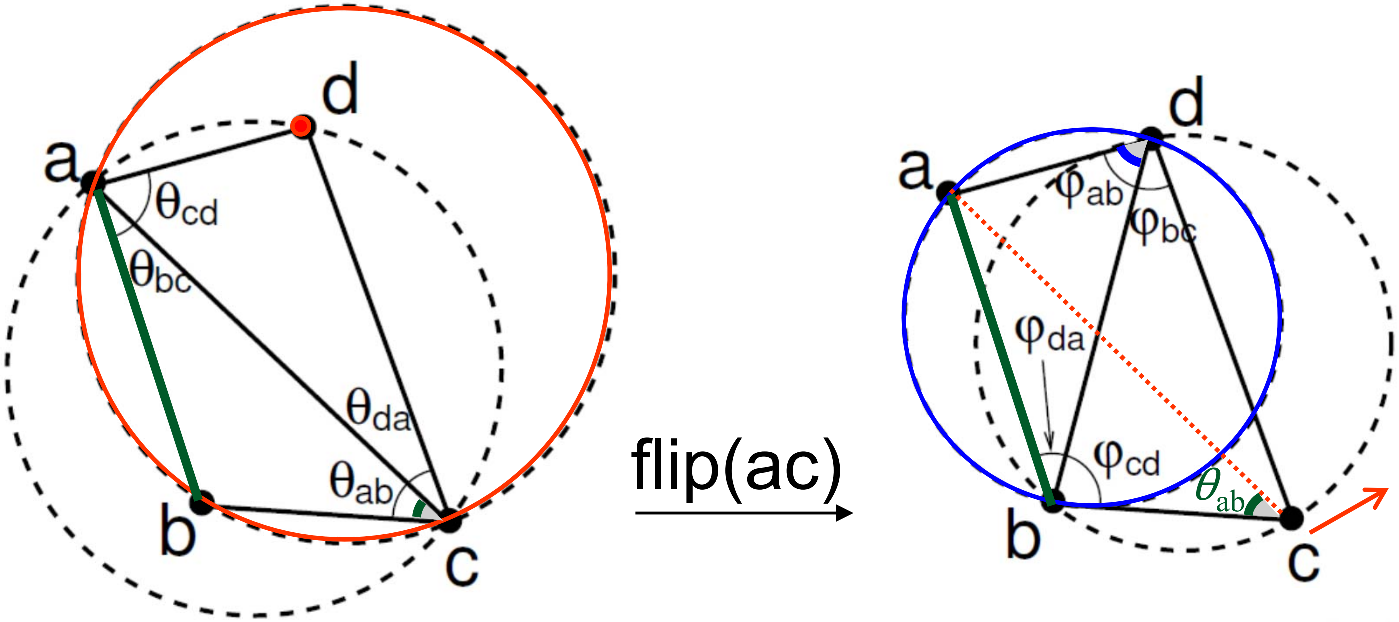
=> After limited number of edge flips

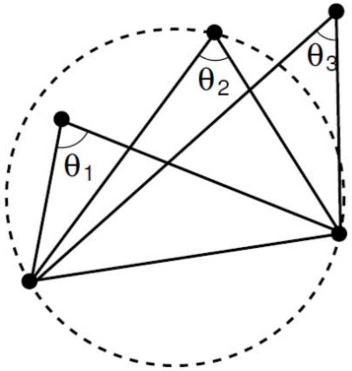
- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd





$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac|$ $\quad \varphi_{ab} > \theta_{ab} \quad \varphi_{bc} > \theta_{bc} \quad \varphi_{cd} > \theta_{cd} \quad \varphi_{da} > \theta_{da}$

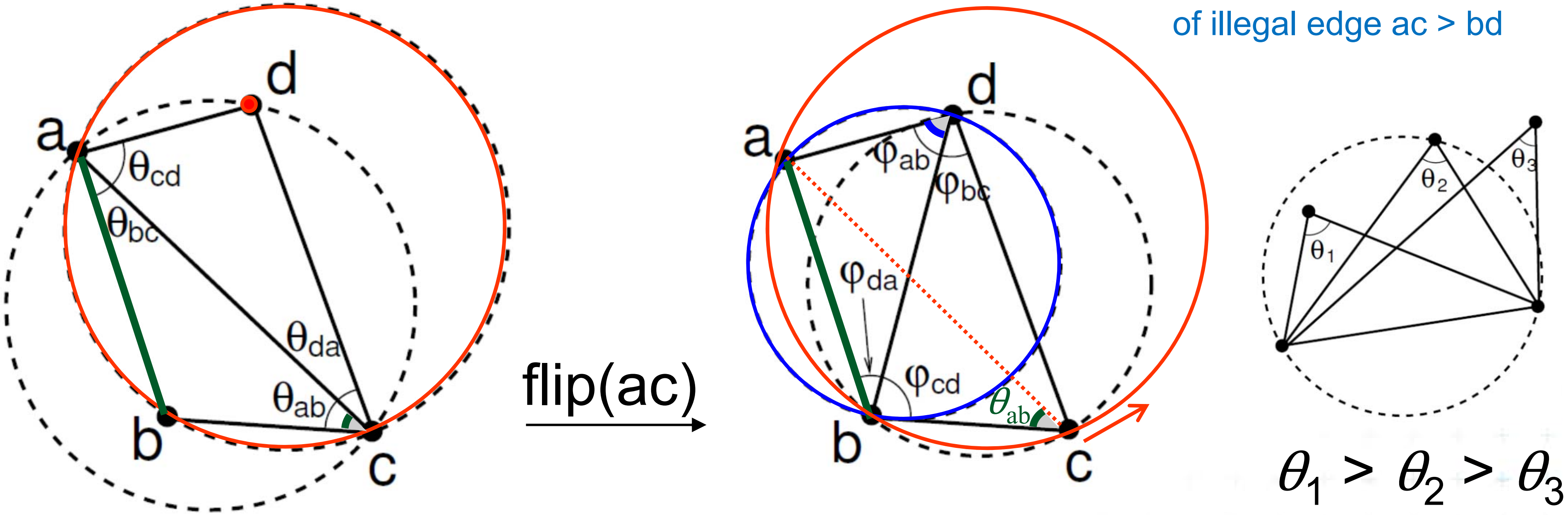
=> After limited number of edge flips

- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

Edge flip of illegal edge and angle vector

- The minimum angle increases after the edge flip

of illegal edge ac > bd



$\theta_1 > \theta_2 > \theta_3$

[Mount]

$|bd| < |ac| \quad \varphi_{ab} > \theta_{ab} \quad \varphi_{bc} > \theta_{bc} \quad \varphi_{cd} > \theta_{cd} \quad \varphi_{da} > \theta_{da}$

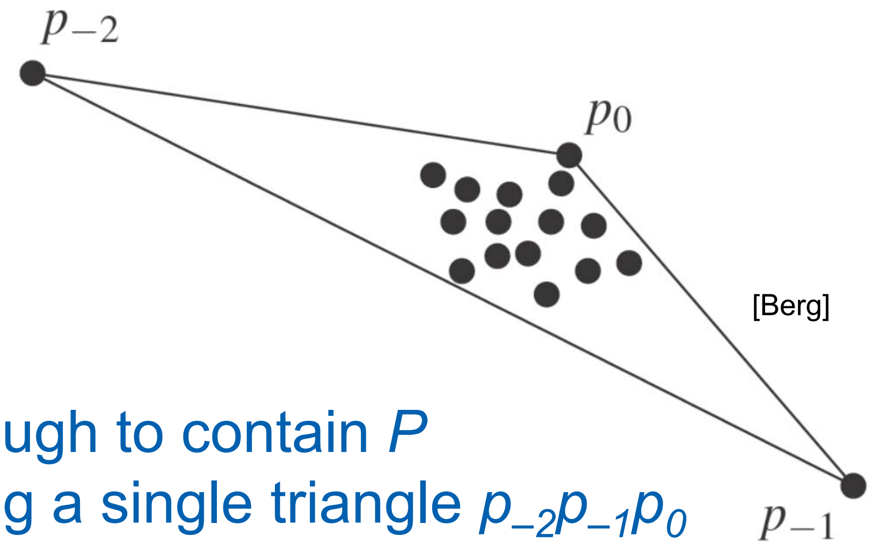
=> After limited number of edge flips

- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

Incremental algorithm principle

1. Create a large triangle containing all points (to avoid problems with unbounded cells)
   - must be larger than the largest circle through 3 points
   - will be discarded at the end
2. Insert the points in random order
   - Find triangle with inserted point *p*
   - Add edges to its vertices (these new edges are correct)
   - Check correctness of the old edges (triangles) "around *p*" and legalize (flip) potentially illegal edges
3. Discard the large triangle and incident edges

Incremental algorithm in detail



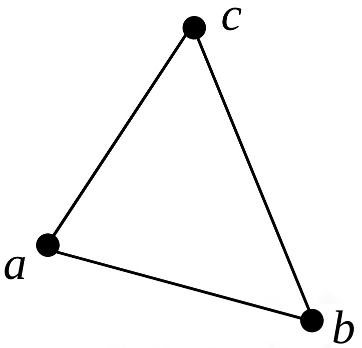
[Berg]

DelaunayTriangulation(*P*)
Input: Set P of n points in the plane
Output: A Delaunay triangulation T of P

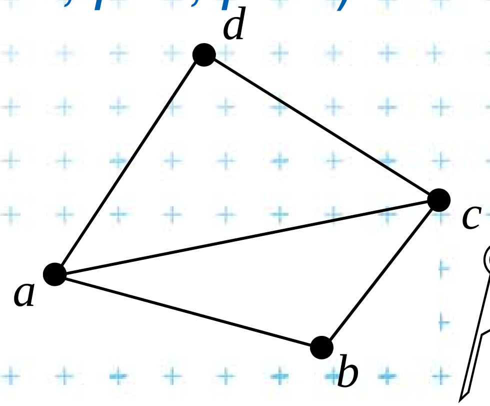
```
1. Let $p_{-2}, p_{-1}, p_0$ form a triangle large enough to contain $P$
2. Initialize $T$ as the triangulation consisting a single triangle $p_{-2}p_{-1}p_0$
3. Compute random permutation $p_1, p_2, \ldots, p_n$ of $P \setminus \{p_0\}$
4. for $r = 1$ to $n$ do
5.     $T$ = Insert( $p_r$ , $T$ )
6. Discard $p_{-1}, p_{-2}$ with all incident edges from $T$
7. return $T$
```

Incremental algorithm – insertion of a point

```
Insert( p, T )
Input:   Point p being inserted into triangulation T
Output:  Correct Delaunay triangulation after insertion of p
Find a triangle $abc \in T$ containing p
if p lies in the interior of abc then
    Insert edges pa, pb, pc into triangulation T
    (splitting abc into 3 triangles pab, pbc, pca )
    LegalizeEdge( p, ab, T)
    LegalizeEdge( p, bc, T)
    LegalizeEdge( p, ca, T)
else // p lies on the edge of abc, say ab, point d is right from edge ab
    Remove ab and insert edges pa, pb, pc, pd into triangulation T
    (splitting abc and abd into 4 triangles pad, pdb, pbc, pca )
    LegalizeEdge( p, ab, T)
    LegalizeEdge( p, bc, T)
    LegalizeEdge( p, cd, T)
    LegalizeEdge( p, da, T)
return T
```



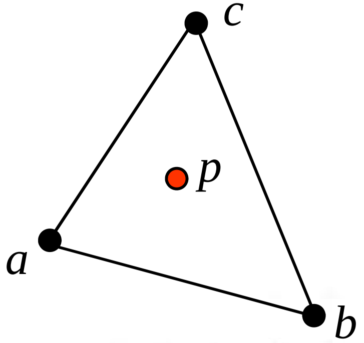
[Berg]



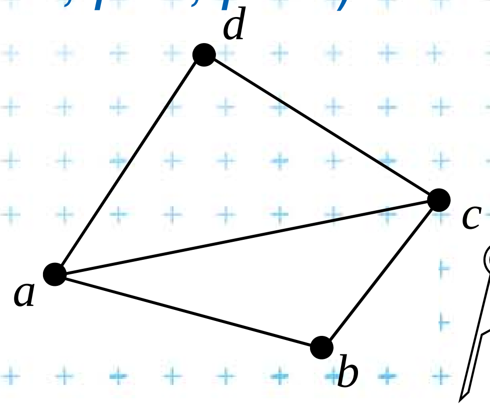
[Berg]

Incremental algorithm – insertion of a point

```
Insert( p, T )
Input:   Point p being inserted into triangulation T
Output:  Correct Delaunay triangulation after insertion of p
1.  Find a triangle abc ∈ T containing p
2.  if p lies in the interior of abc then
3.      Insert edges pa, pb, pc into triangulation T
        (splitting abc into 3 triangles pab, pbc, pca )
4.      LegalizeEdge( p, ab, T)
5.      LegalizeEdge( p, bc, T)
6.      LegalizeEdge( p, ca, T)
7.  else // p lies on the edge of abc, say ab, point d is right from edge ab
8.      Remove ab and insert edges pa, pb, pc, pd into triangulation T
        (splitting abc and abd into 4 triangles pad, pdb, pbc, pca )
9.      LegalizeEdge( p, ab, T)
10.     LegalizeEdge( p, bc, T)
11.     LegalizeEdge( p, cd, T)
12.     LegalizeEdge( p, da, T)
13. return T
```



[Berg]



[Berg]

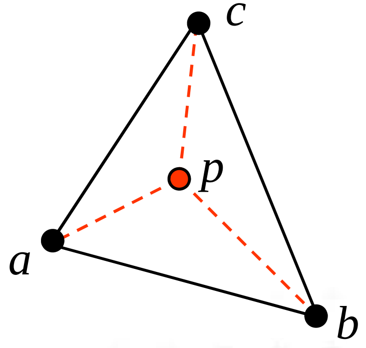
Incremental algorithm – insertion of a point

Insert(*p*, *T*)

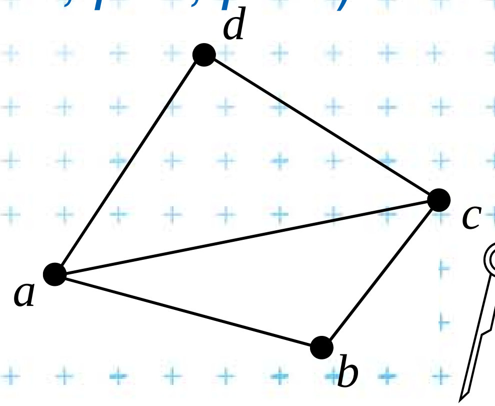
Input: Point *p* being inserted into triangulation *T*

Output: Correct Delaunay triangulation after insertion of *p*

```
Find a triangle $abc \in T$ containing p
if p lies in the interior of abc then
    Insert edges pa, pb, pc into triangulation T
    (splitting abc into 3 triangles pab, pbc, pca )
    LegalizeEdge( p, ab, T)
    LegalizeEdge( p, bc, T)
    LegalizeEdge( p, ca, T)
else // p lies on the edge of abc, say ab, point d is right from edge ab
    Remove ab and insert edges pa, pb, pc, pd into triangulation T
    (splitting abc and abd into 4 triangles pad, pdb, pbc, pca )
    LegalizeEdge( p, ab, T)
    LegalizeEdge( p, bc, T)
    LegalizeEdge( p, cd, T)
    LegalizeEdge( p, da, T)
return T
```



[Berg]



[Berg]

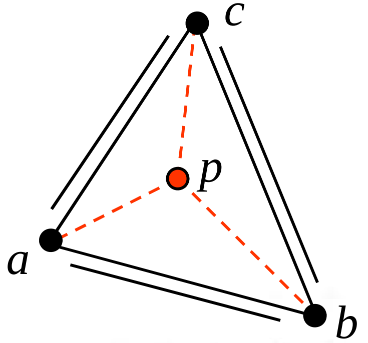
Incremental algorithm – insertion of a point

Insert(*p*, *T*)

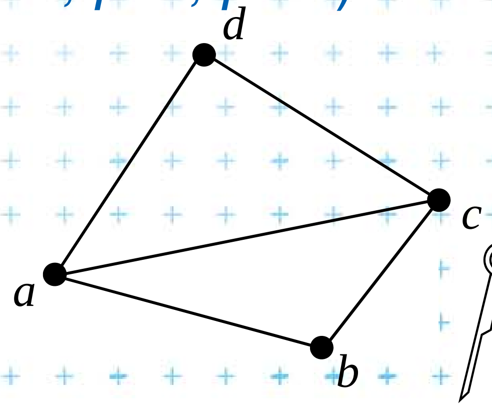
Input: Point *p* being inserted into triangulation *T*

Output: Correct Delaunay triangulation after insertion of *p*

```
Find a triangle $abc \in T$ containing $p$
if $p$ lies in the interior of $abc$ then
    Insert edges $pa, pb, pc$ into triangulation $T$
    (splitting $abc$ into 3 triangles $pab, pbc, pca$ )
    LegalizeEdge( $p, ab, T$)
    LegalizeEdge( $p, bc, T$)
    LegalizeEdge( $p, ca, T$)
else // $p$ lies on the edge of $abc$, say $ab$, point $d$ is right from edge $ab$
    Remove $ab$ and insert edges $pa, pb, pc, pd$ into triangulation $T$
    (splitting $abc$ and $abd$ into 4 triangles $pad, pdb, pbc, pca$ )
    LegalizeEdge( $p, ab, T$)
    LegalizeEdge( $p, bc, T$)
    LegalizeEdge( $p, cd, T$)
    LegalizeEdge( $p, da, T$)
return $T$
```



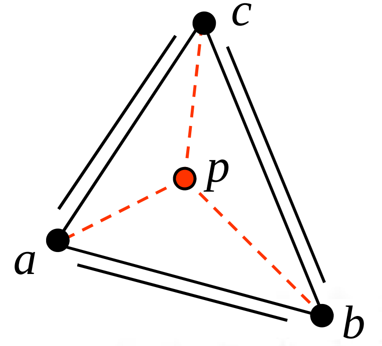
[Berg]



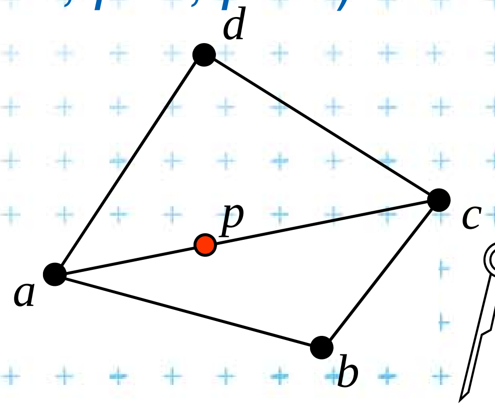
[Berg]

Incremental algorithm – insertion of a point

```
Insert( p, T )
Input:   Point p being inserted into triangulation T
Output:  Correct Delaunay triangulation after insertion of p
Find a triangle $abc \in T$ containing p
if p lies in the interior of abc then
    Insert edges pa, pb, pc into triangulation T
    (splitting abc into 3 triangles pab, pbc, pca )
    LegalizeEdge( p, ab, T)
    LegalizeEdge( p, bc, T)
    LegalizeEdge( p, ca, T)
else // p lies on the edge of abc, say ab, point d is right from edge ab
    Remove ab and insert edges pa, pb, pc, pd into triangulation T
    (splitting abc and abd into 4 triangles pad, pdb, pbc, pca )
    LegalizeEdge( p, ab, T)
    LegalizeEdge( p, bc, T)
    LegalizeEdge( p, cd, T)
    LegalizeEdge( p, da, T)
return T
```



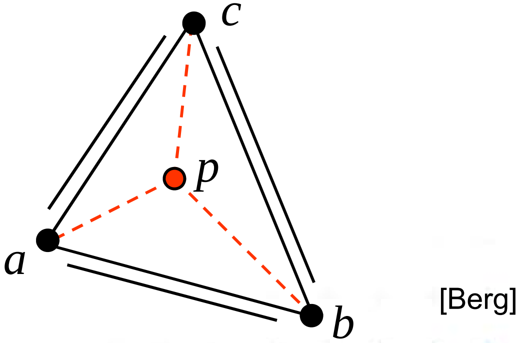
[Berg]



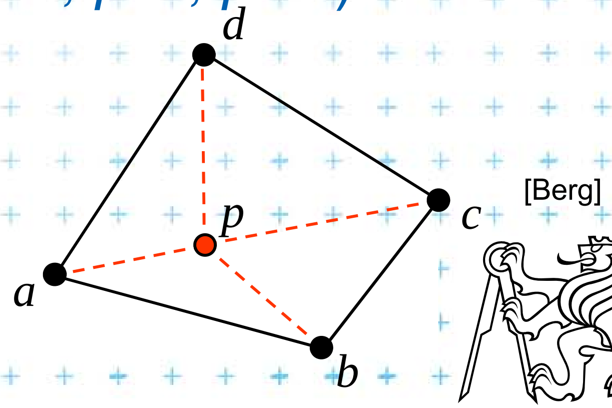
[Berg]

Incremental algorithm – insertion of a point

```
Insert( p, T )
Input:   Point p being inserted into triangulation T
Output:  Correct Delaunay triangulation after insertion of p
1.   Find a triangle $abc \in T$ containing p
2.   if p lies in the interior of abc then
3.       Insert edges pa, pb, pc into triangulation T
         (splitting abc into 3 triangles pab, pbc, pca )
4.       LegalizeEdge( p, ab, T)
5.       LegalizeEdge( p, bc, T)
6.       LegalizeEdge( p, ca, T)
7.   else // p lies on the edge of abc, say ab, point d is right from edge ab
8.       Remove ab and insert edges pa, pb, pc, pd into triangulation T
         (splitting abc and abd into 4 triangles pad, pdb, pbc, pca )
9.       LegalizeEdge( p, ab, T)
10.      LegalizeEdge( p, bc, T)
11.      LegalizeEdge( p, cd, T)
12.      LegalizeEdge( p, da, T)
13.  return  T
```



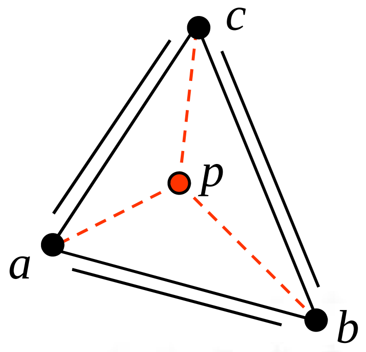
[Berg]



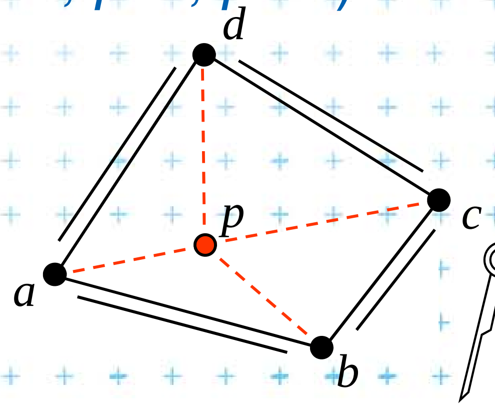
[Berg]

Incremental algorithm – insertion of a point

```
Insert( p, T )
Input:   Point p being inserted into triangulation T
Output:  Correct Delaunay triangulation after insertion of p
Find a triangle $abc \in T$ containing p
if p lies in the interior of abc then
    Insert edges pa, pb, pc into triangulation T
    (splitting abc into 3 triangles pab, pbc, pca )
    LegalizeEdge( p, ab, T)
    LegalizeEdge( p, bc, T)
    LegalizeEdge( p, ca, T)
else // p lies on the edge of abc, say ab, point d is right from edge ab
    Remove ab and insert edges pa, pb, pc, pd into triangulation T
    (splitting abc and abd into 4 triangles pad, pdb, pbc, pca )
    LegalizeEdge( p, ab, T)
    LegalizeEdge( p, bc, T)
    LegalizeEdge( p, cd, T)
    LegalizeEdge( p, da, T)
return  T
```



[Berg]



[Berg]

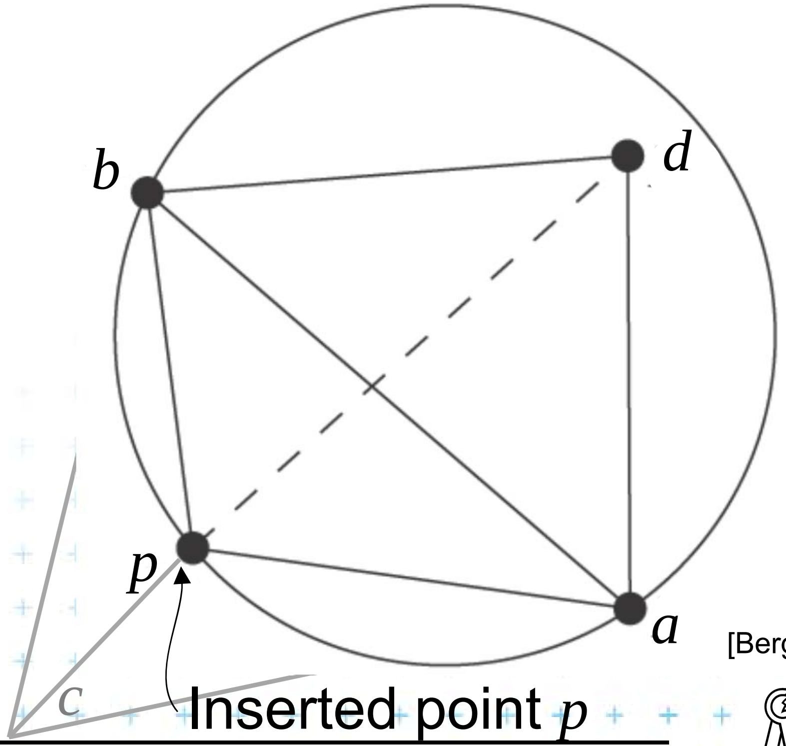
Incremental algorithm – edge legalization

LegalizeEdge(*p, ab, T*)

Input: Edge *ab* being checked after insertion of point *p* to triangulation *T*

Output: Delaunay triangulation of *p* ∪*T*

```
if( ab is edge on the exterior face ) return
let d be the vertex to the right of edge ab
if( inCircle( p, a, b, d ) )   // d is in the circle around pab => d is illegal
    Flip edge ab for pd
    LegalizeEdge( p, ad, T )
    LegalizeEdge( p, db, T )
```



[Berg]

Incremental algorithm – edge legalization

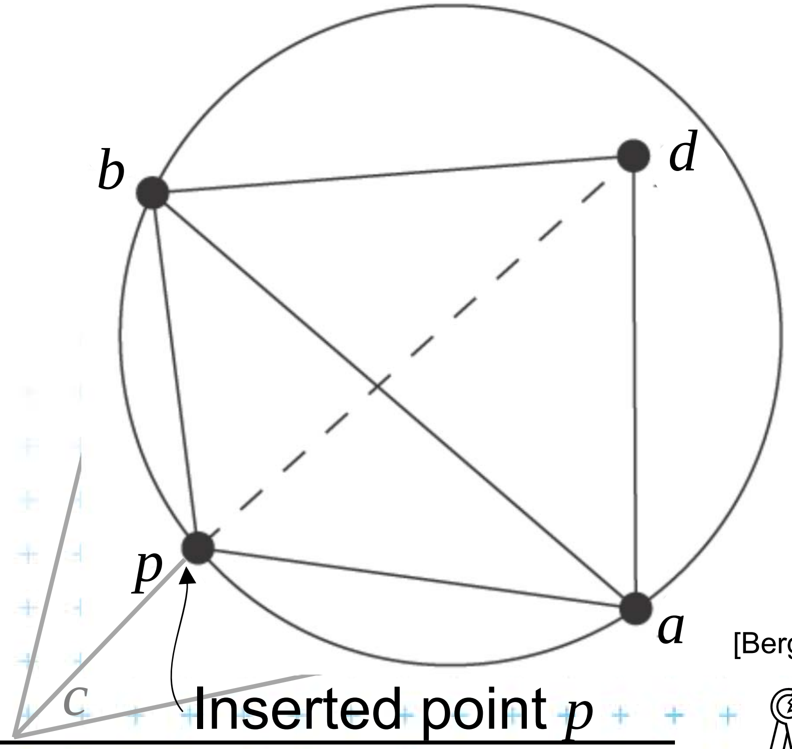
LegalizeEdge(*p, ab, T*)

Input: Edge *ab* being checked after insertion of point *p* to triangulation *T*

Output: Delaunay triangulation of *p* ∪*T*

```
1. if( ab is edge on the exterior face ) return
2. let d be the vertex to the right of edge ab
3. if( inCircle( p, a, b, d ) )   // d is in the circle around pab => d is illegal
4.     Flip edge ab for pd
5.     LegalizeEdge( p, ad, T )
6.     LegalizeEdge( p, db, T )
```

Insertion of *p* may make edges *ab*, *bc* & *ca* illegal
(circle around *pab* will contain point *d*)



[Berg]

Incremental algorithm – edge legalization

LegalizeEdge(*p, ab, T*)

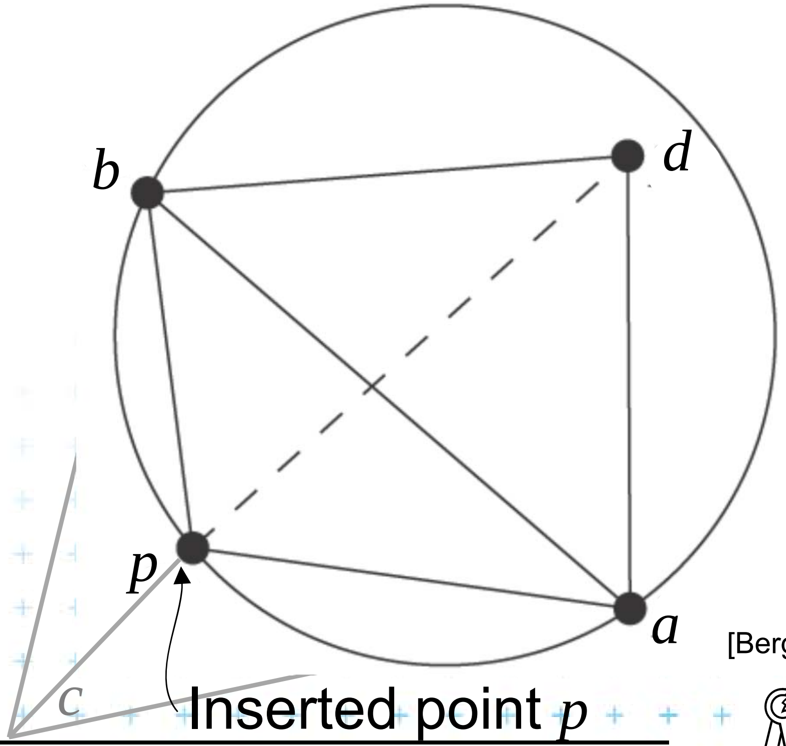
Input: Edge *ab* being checked after insertion of point *p* to triangulation *T*

Output: Delaunay triangulation of *p* ∪*T*

```
1. if( ab is edge on the exterior face ) return
2. let d be the vertex to the right of edge ab
3. if( inCircle( p, a, b, d ) )   // d is in the circle around pab => d is illegal
4.     Flip edge ab for pd
5.     LegalizeEdge( p, ad, T )
6.     LegalizeEdge( p, db, T )
```

Insertion of *p* may make edges *ab, bc* & *ca* illegal
(circle around *pab* will contain point *d*)

After edge flip, the edge *pd* will be legal
(the circumcircles of the resulting triangles
pdb, and *pad* will bee empty)



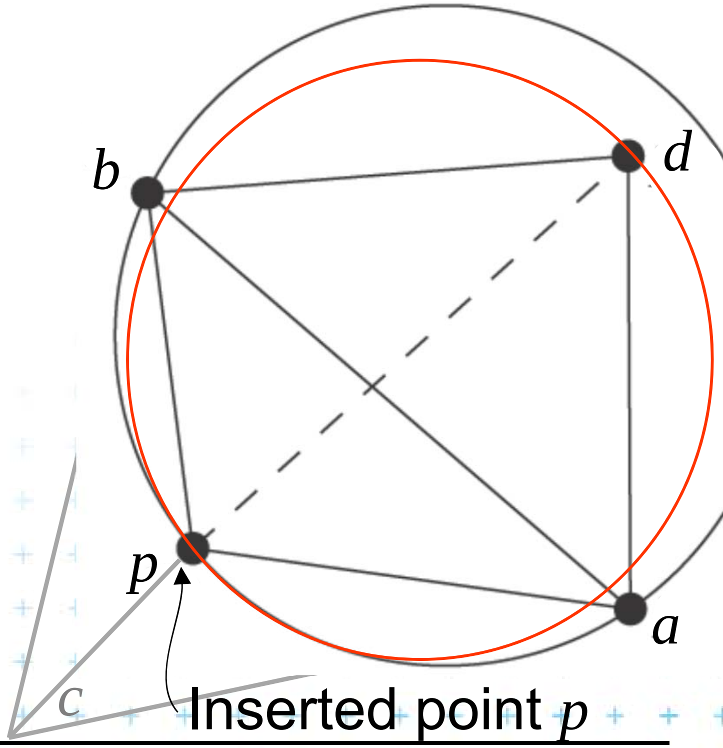
[Berg]

Incremental algorithm – edge legalization

```
LegalizeEdge( p, ab, T )
Input:   Edge ab being checked after insertion of point p to triangulation T
Output:  Delaunay triangulation of p ∪T
if( ab is edge on the exterior face ) return
let d be the vertex to the right of edge ab
if( inCircle( p, a, b, d ) )   // d is in the circle around pab => d is illegal
    Flip edge ab for pd
    LegalizeEdge( p, ad, T )
    LegalizeEdge( p, db, T )
```

Insertion of *p* may make edges *ab, bc* & *ca* illegal
(circle around *pab* will contain point *d*)

After edge flip, the edge *pd* will be legal
(the circumcircles of the resulting triangles
pdb, and *pad* will bee empty)



[Berg]

Incremental algorithm – edge legalization

LegalizeEdge(*p, ab, T*)

Input: Edge *ab* being checked after insertion of point *p* to triangulation *T*

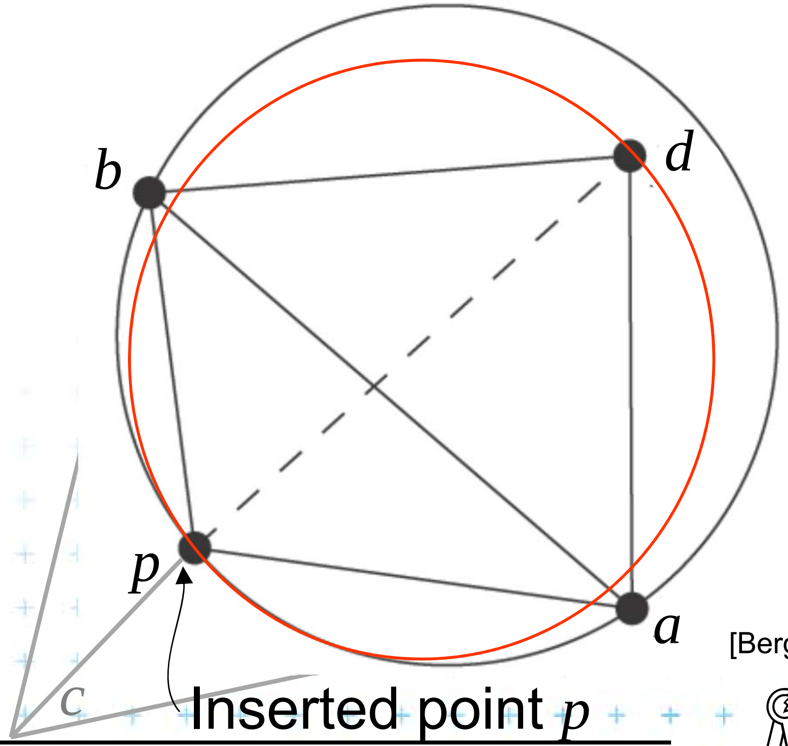
Output: Delaunay triangulation of $p \cup T$

```
if( ab is edge on the exterior face ) return
let d be the vertex to the right of edge ab
if( inCircle( p, a, b, d ) )   // d is in the circle around pab => d is illegal
    Flip edge ab for pd
    LegalizeEdge( p, ad, T )
    LegalizeEdge( p, db, T )
```

Insertion of *p* may make edges *ab*, *bc* & *ca* illegal (circle around *pab* will contain point *d*)

After edge flip, the edge *pd* will be legal (the circumcircles of the resulting triangles *pdb*, and *pad* will bee empty)

We must check and possibly flip edges *ad*, *db*



[Berg]

Incremental algorithm – edge legalization

LegalizeEdge(*p, ab, T*)

Input: Edge *ab* being checked after insertion of point *p* to triangulation *T*
Output: Delaunay triangulation of *p* ∪*T*

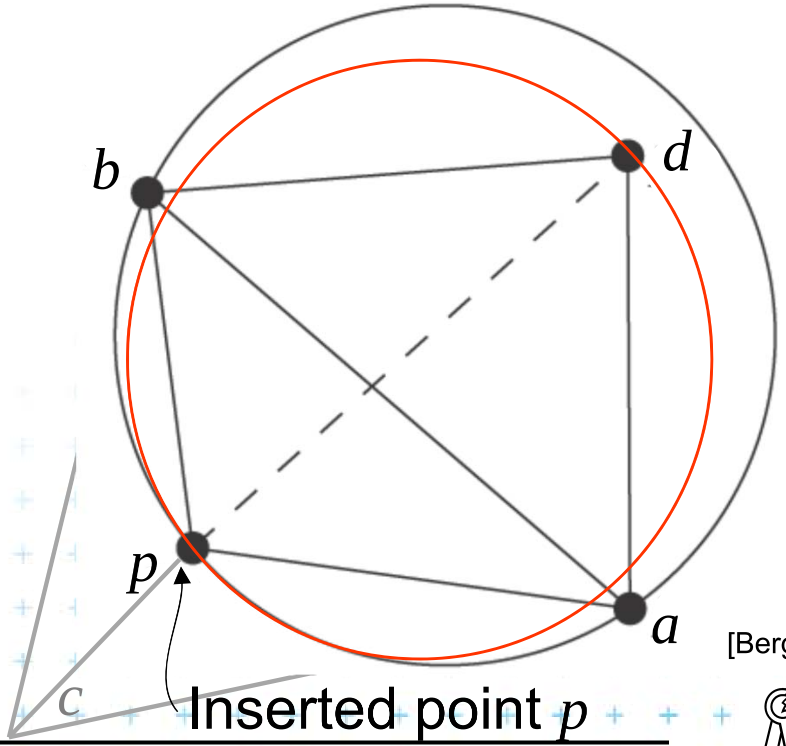
```
1. if( ab is edge on the exterior face ) return
2. let d be the vertex to the right of edge ab
3. if( inCircle( p, a, b, d ) )   // d is in the circle around pab => d is illegal
4.     Flip edge ab for pd
5.     LegalizeEdge( p, ad, T )
6.     LegalizeEdge( p, db, T )
```

Insertion of *p* may make edges *ab, bc* & *ca* illegal
(circle around *pab* will contain point *d*)

After edge flip, the edge *pd* will be legal
(the circumcircles of the resulting triangles
pdb, and *pad* will bee empty)

We must check and possibly flip edges *ad, db*

(We must check and possibly flip edges *bc* & *ca*
- lines 5,6 in Insert(p, T))



[Berg]

Incremental algorithm – edge legalization

LegalizeEdge(*p, ab, T*)

Input: Edge *ab* being checked after insertion of point *p* to triangulation *T*

Output: Delaunay triangulation of *p* ∪*T*

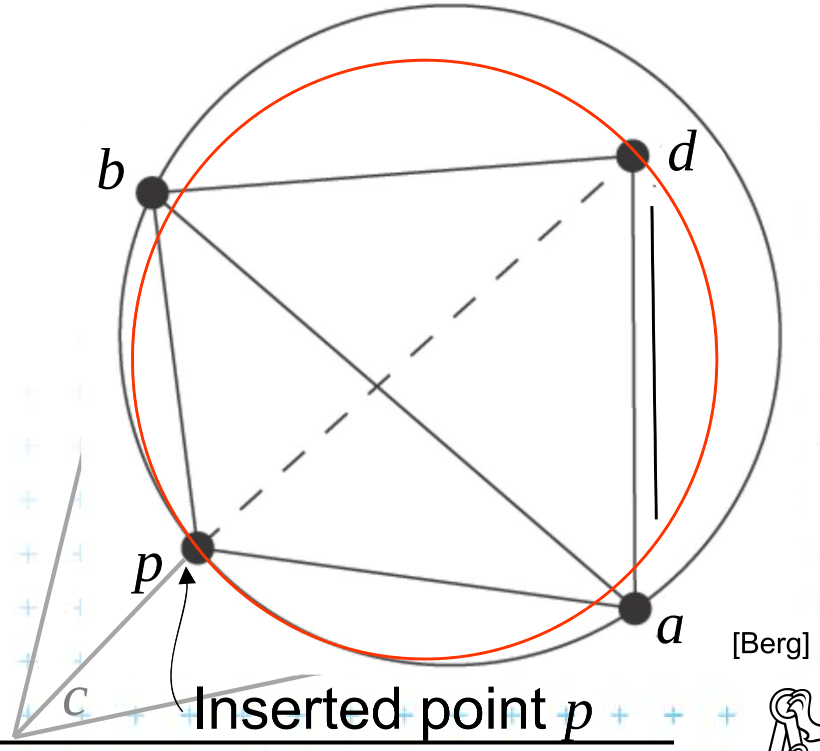
```
if( ab is edge on the exterior face ) return
let d be the vertex to the right of edge ab
if( inCircle( p, a, b, d ) )   // d is in the circle around pab => d is illegal
    Flip edge ab for pd
    LegalizeEdge( p, ad, T )
    LegalizeEdge( p, db, T )
```

Insertion of *p* may make edges *ab, bc* & *ca* illegal
(circle around *pab* will contain point *d*)

After edge flip, the edge *pd* will be legal
(the circumcircles of the resulting triangles
pdb, and *pad* will bee empty)

We must check and possibly flip edges *ad, db*

(We must check and possibly flip edges *bc* & *ca*
- lines 5,6 in Insert(p, T))



[Berg]

Incremental algorithm – edge legalization

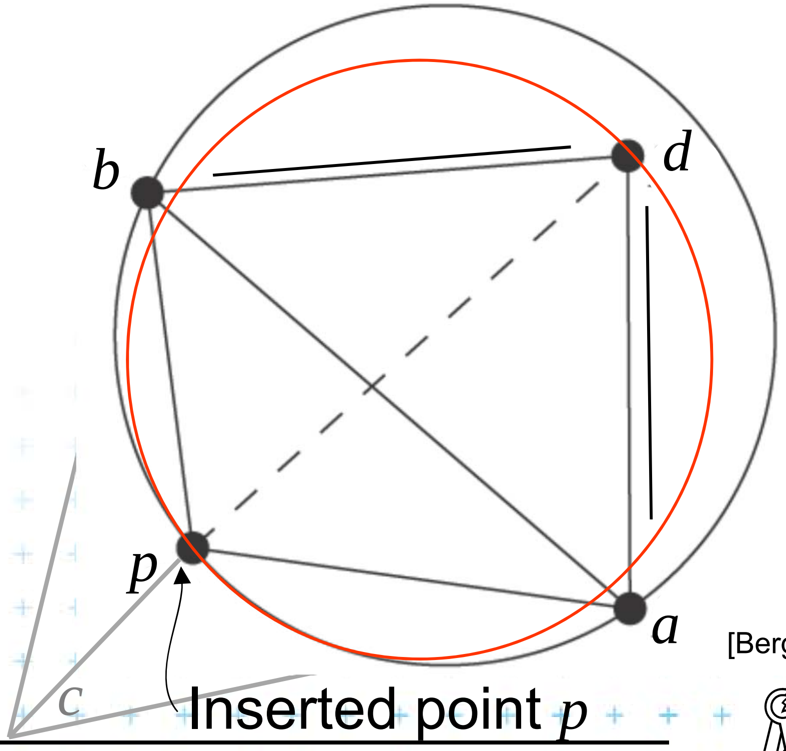
```
LegalizeEdge( p, ab, T )
Input:   Edge ab being checked after insertion of point p to triangulation T
Output:  Delaunay triangulation of p ∪T
if( ab is edge on the exterior face ) return
let d be the vertex to the right of edge ab
if( inCircle( p, a, b, d ) )   // d is in the circle around pab => d is illegal
    Flip edge ab for pd
    LegalizeEdge( p, ad, T )
    LegalizeEdge( p, db, T )
```

Insertion of *p* may make edges *ab, bc* & *ca* illegal
(circle around *pab* will contain point *d*)

After edge flip, the edge *pd* will be legal
(the circumcircles of the resulting triangles
pdb, and *pad* will bee empty)

We must check and possibly flip edges *ad, db*

(We must check and possibly flip edges *bc* & *ca*
- lines 5,6 in Insert(p, T))

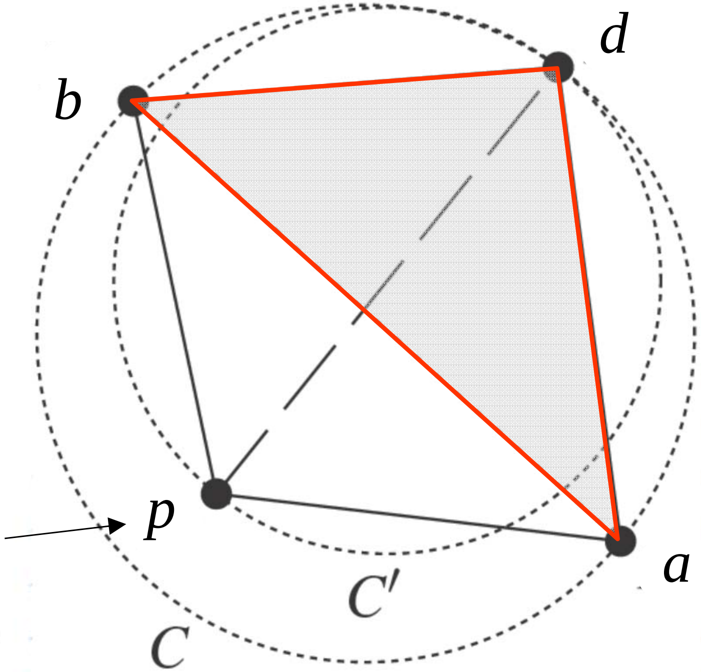


[Berg]

Correctness of edge flip of illegal edge

- Assume point p is in C (it violates DT criteria for adb)
- adb was a triangle of DT => C was an empty circle
- Create circle C' trough point p, C' is inscribed to C, $C' \subset C$
  => C' is also an empty circle ($a, b \notin C$)
  => new edge *pd* is a Delaunay edge

Inserted point p

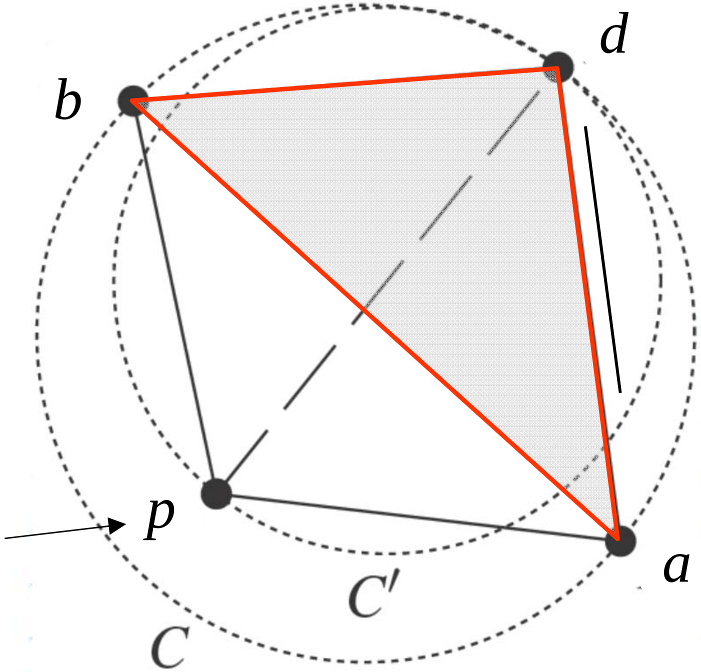


[Berg]

Correctness of edge flip of illegal edge

- Assume point p is in C (it violates DT criteria for adb)
- adb was a triangle of DT => C was an empty circle
- Create circle C' trough point p, C' is inscribed to C, $C' \subset C$
  => C' is also an empty circle ($a, b \notin C$)
  => new edge *pd* is a Delaunay edge

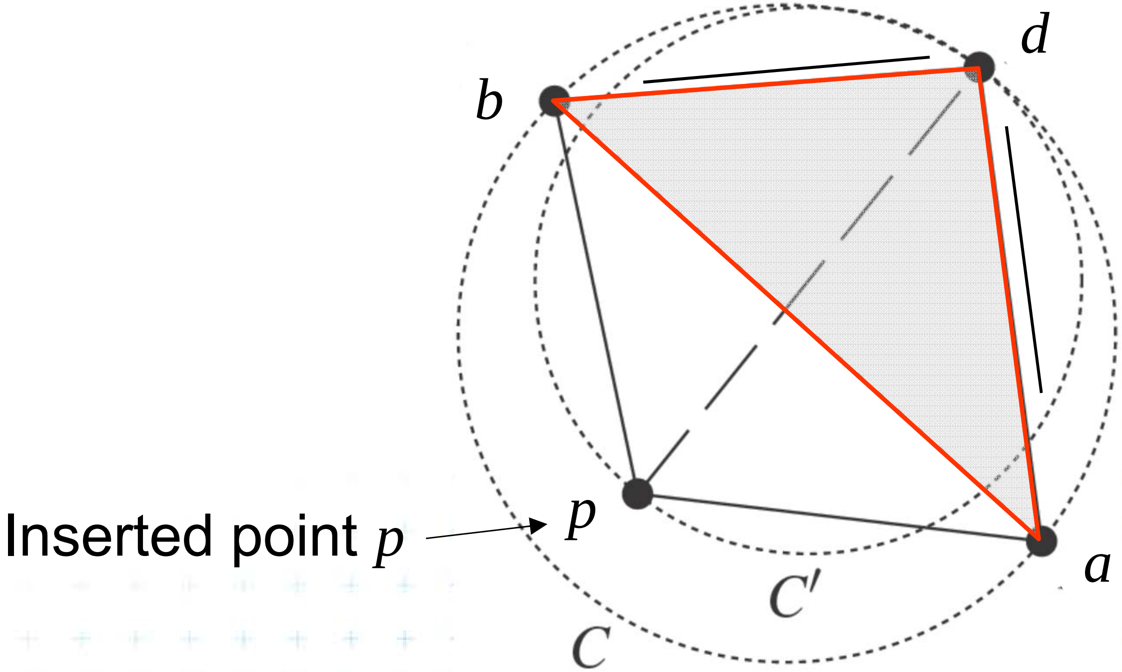
Inserted point p



[Berg]

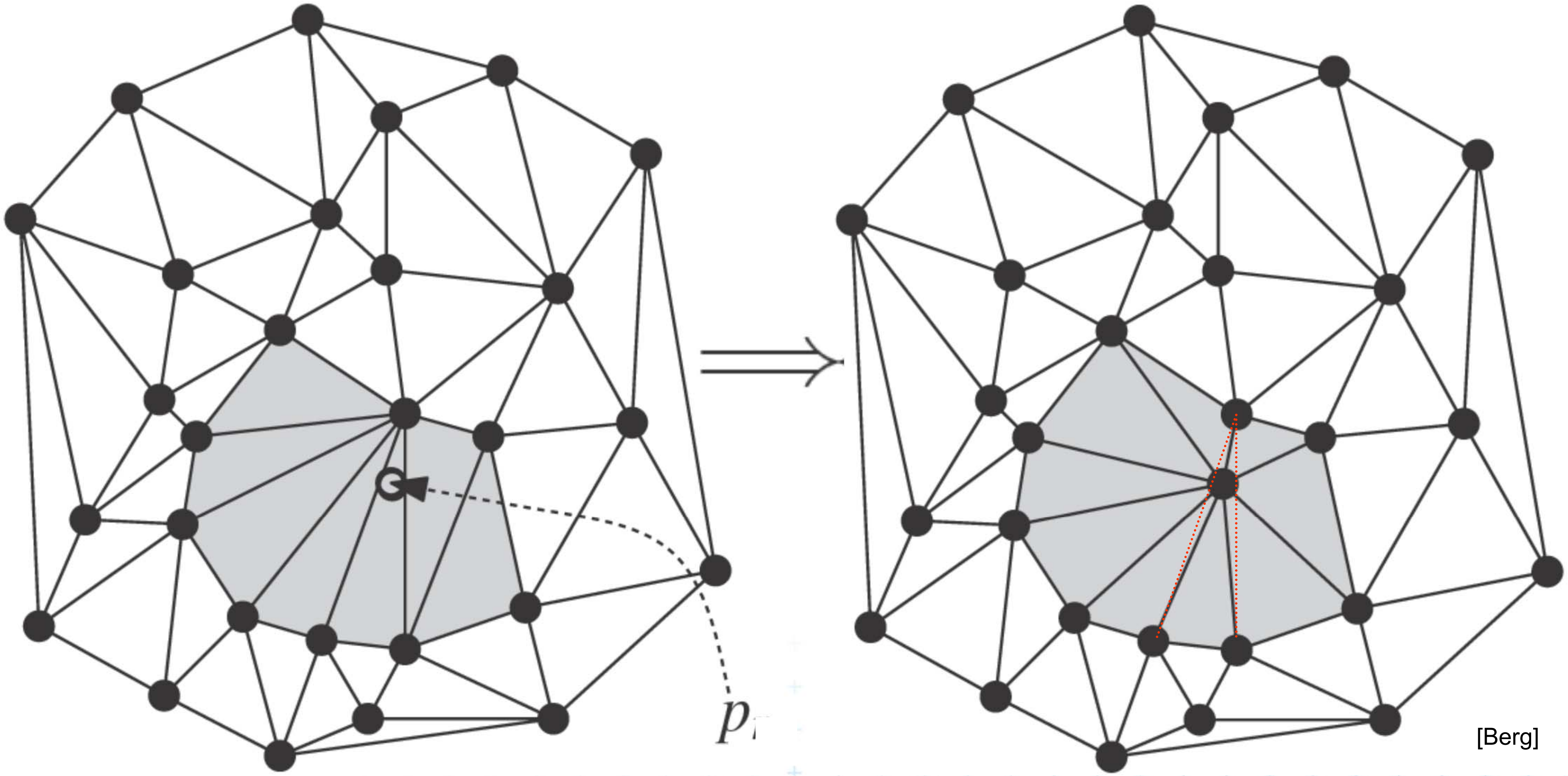
Correctness of edge flip of illegal edge

- Assume point p is in C (it violates DT criteria for adb)
- adb was a triangle of DT => C was an empty circle
- Create circle C' trough point p, C' is inscribed to C, $C' \subset C$
  => C' is also an empty circle ($a, b \notin C$)
  => new edge *pd* is a Delaunay edge



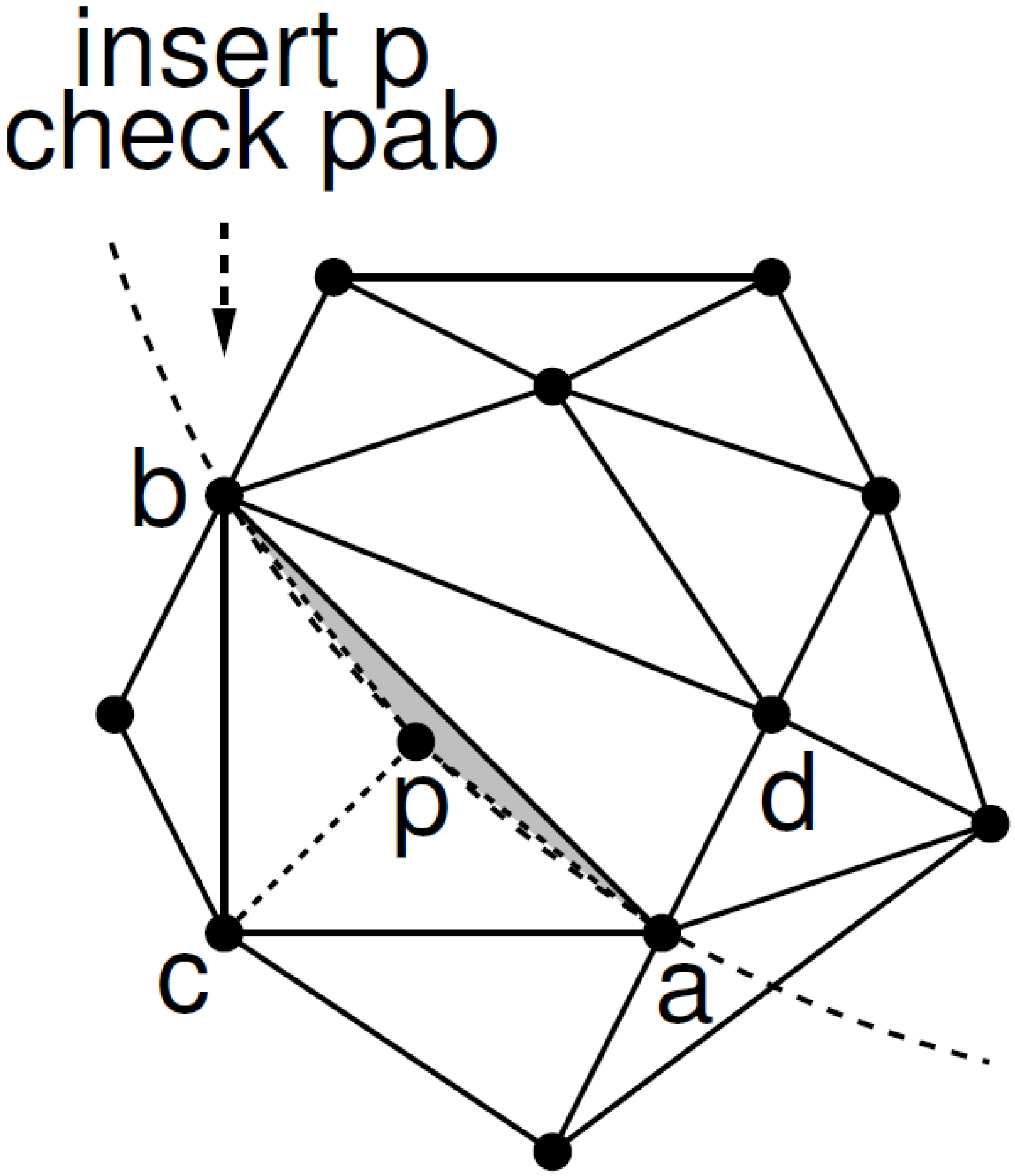
[Berg]

DT- point insert and mesh legalization



Every new edge created due to insertion of *p* will be incident to *p*

Delaunay triangulation – other point insert



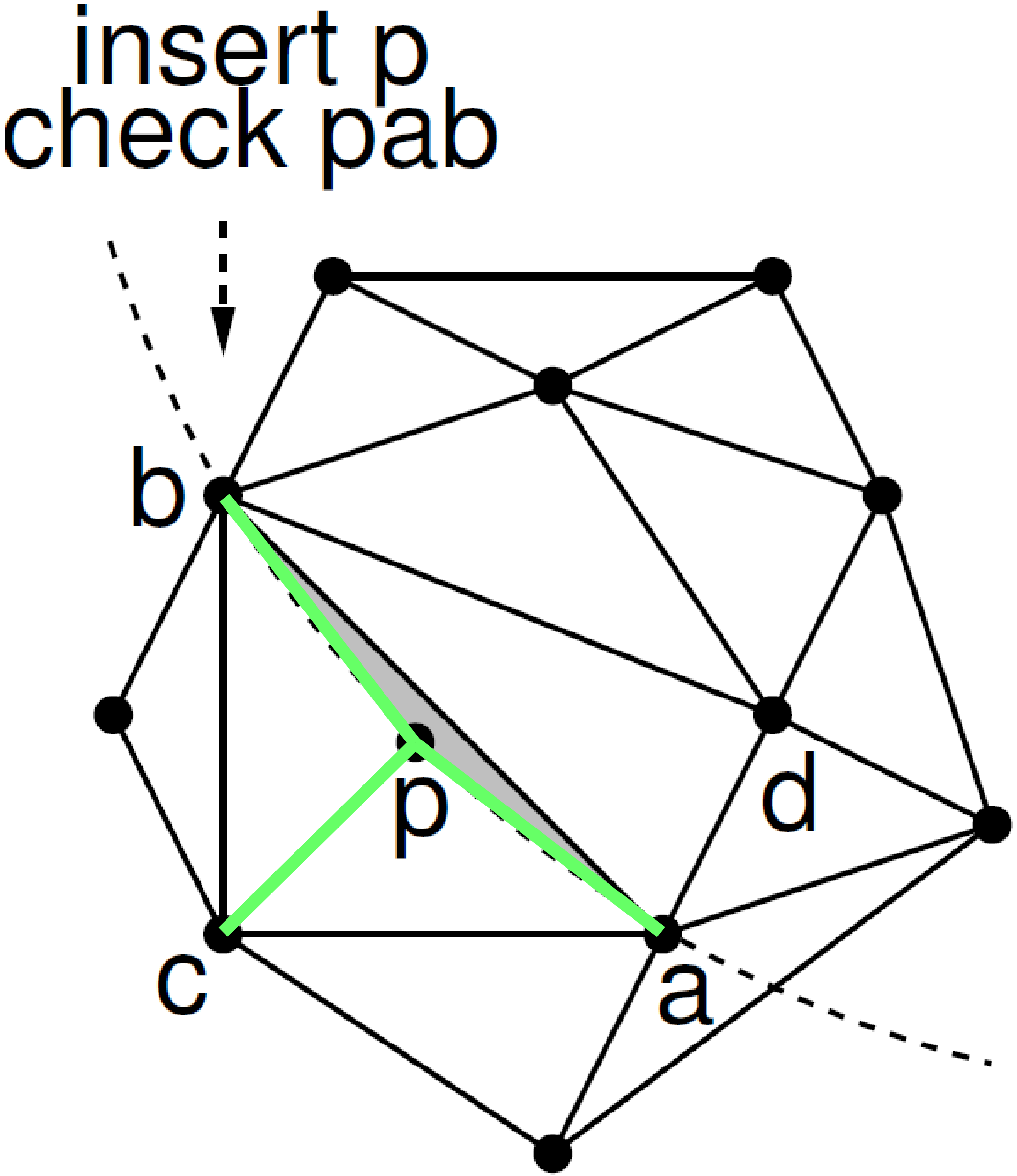
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



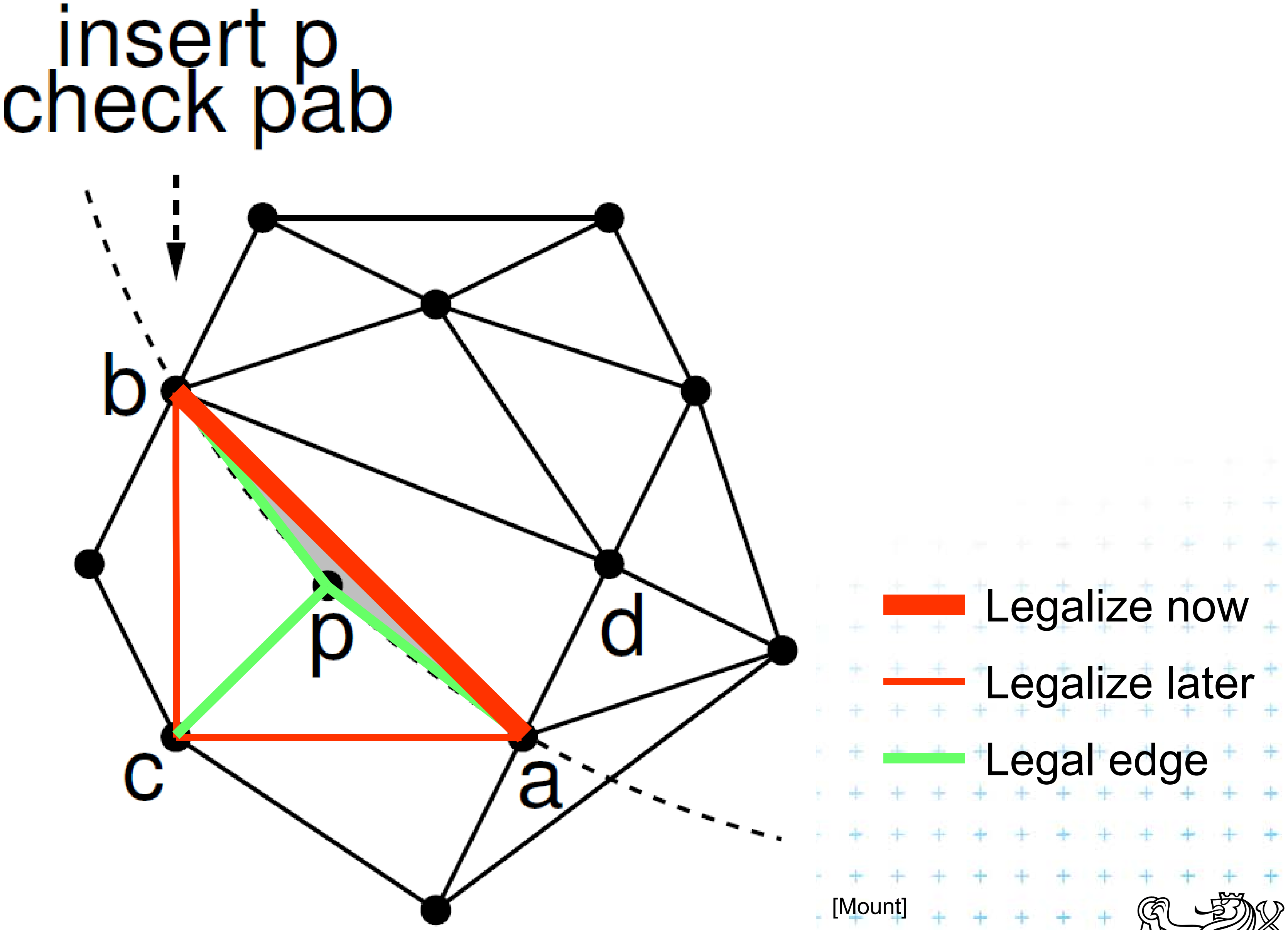
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

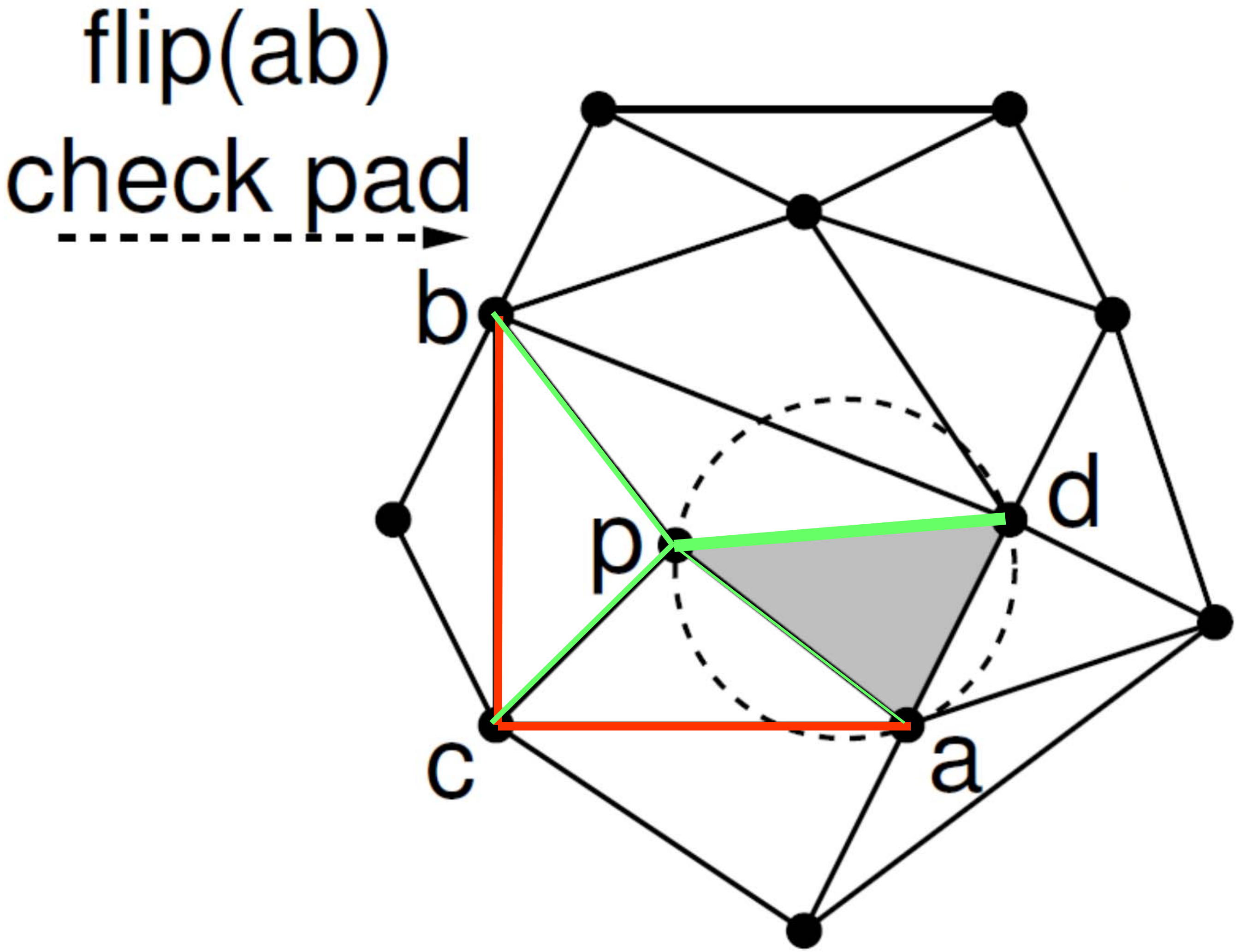
[Mount]

Delaunay triangulation – other point insert



[Mount]

Delaunay triangulation – other point insert



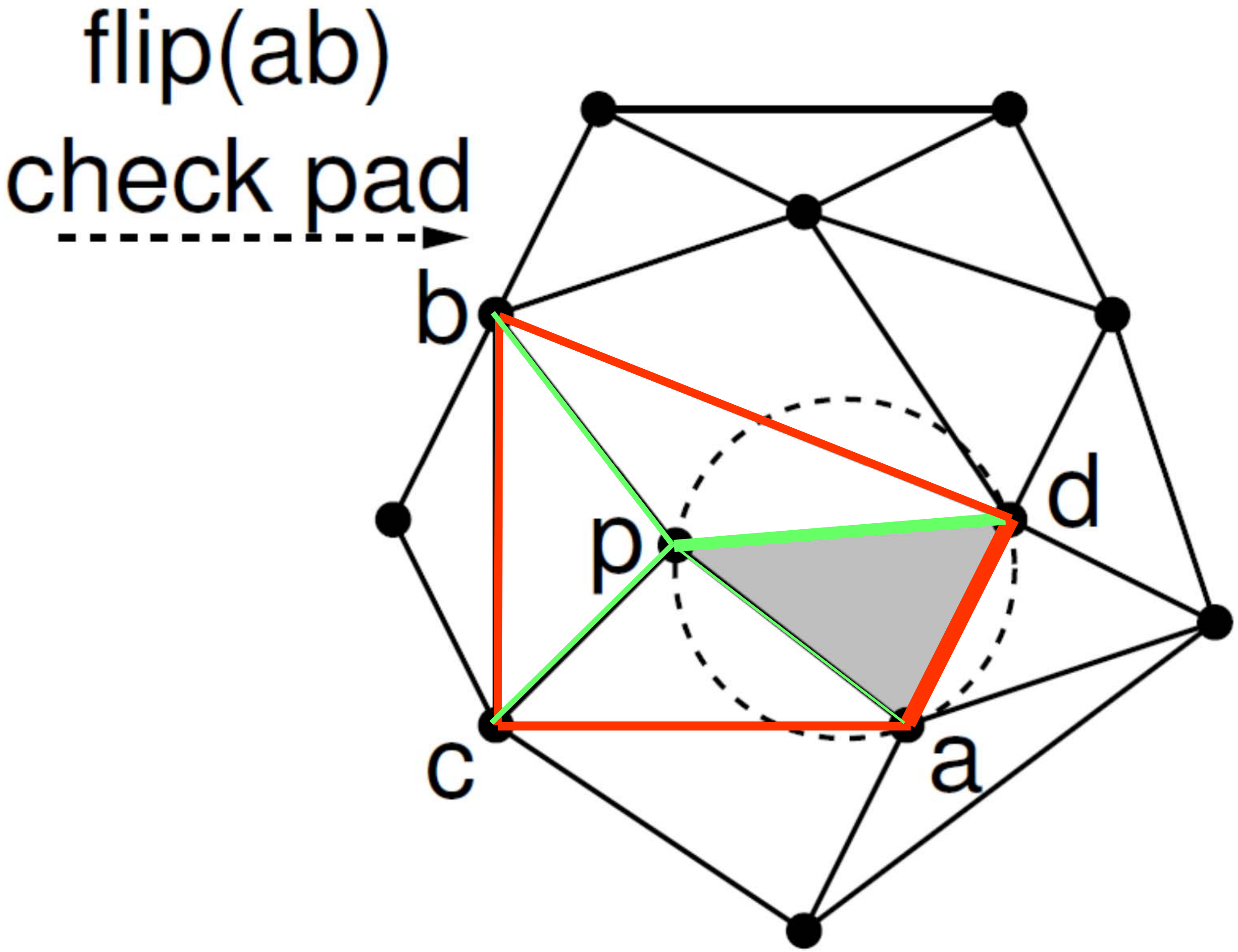
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



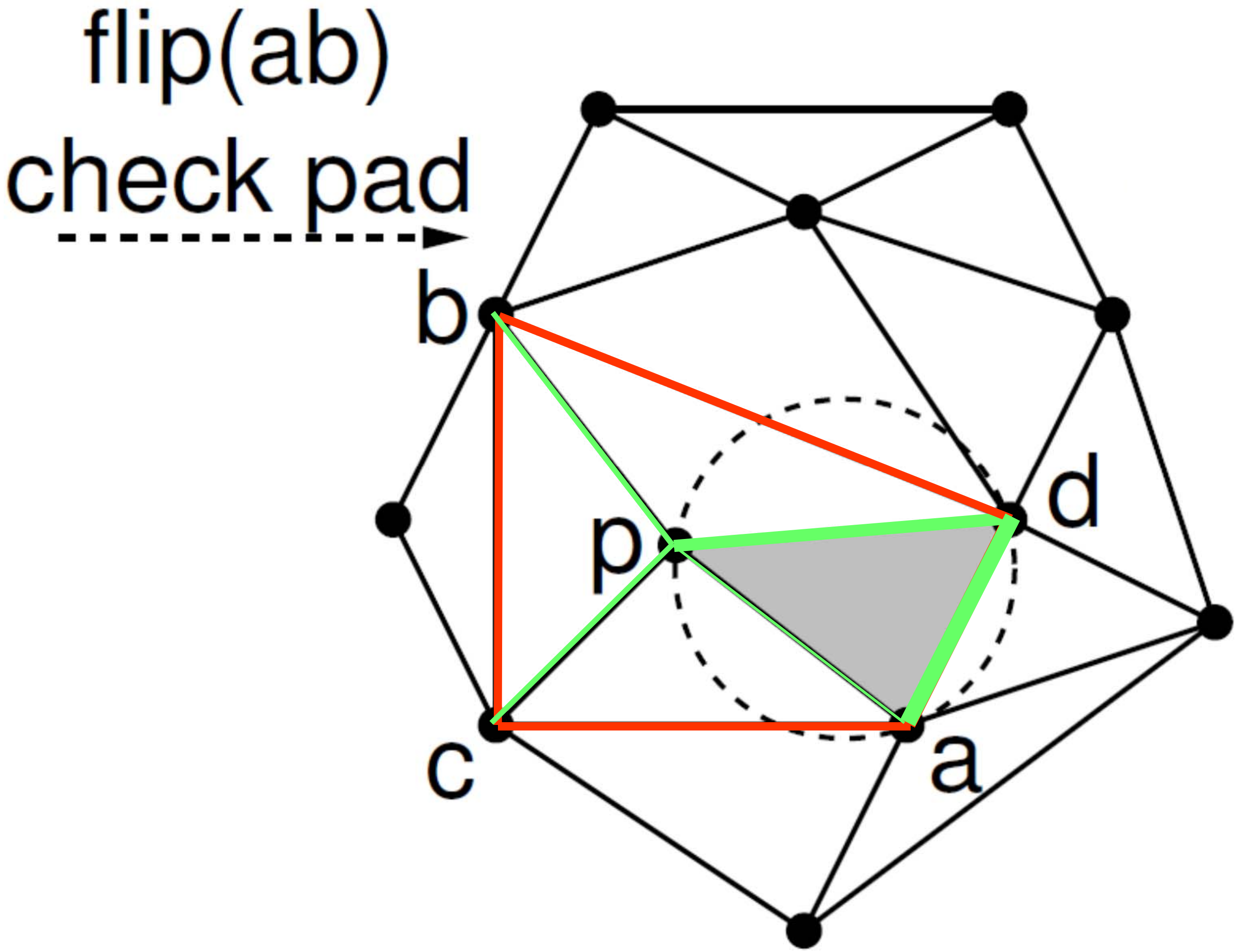
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



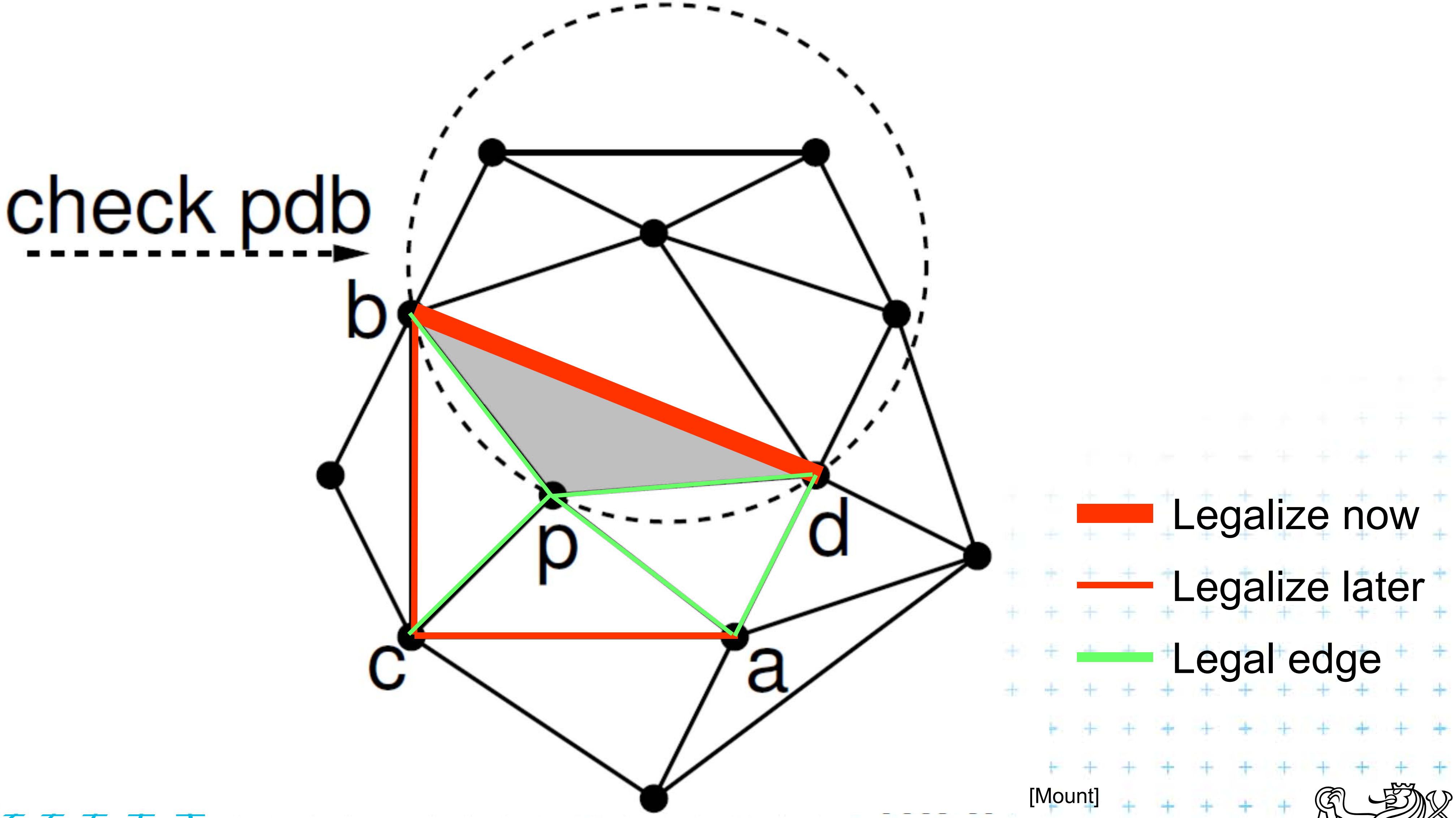
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

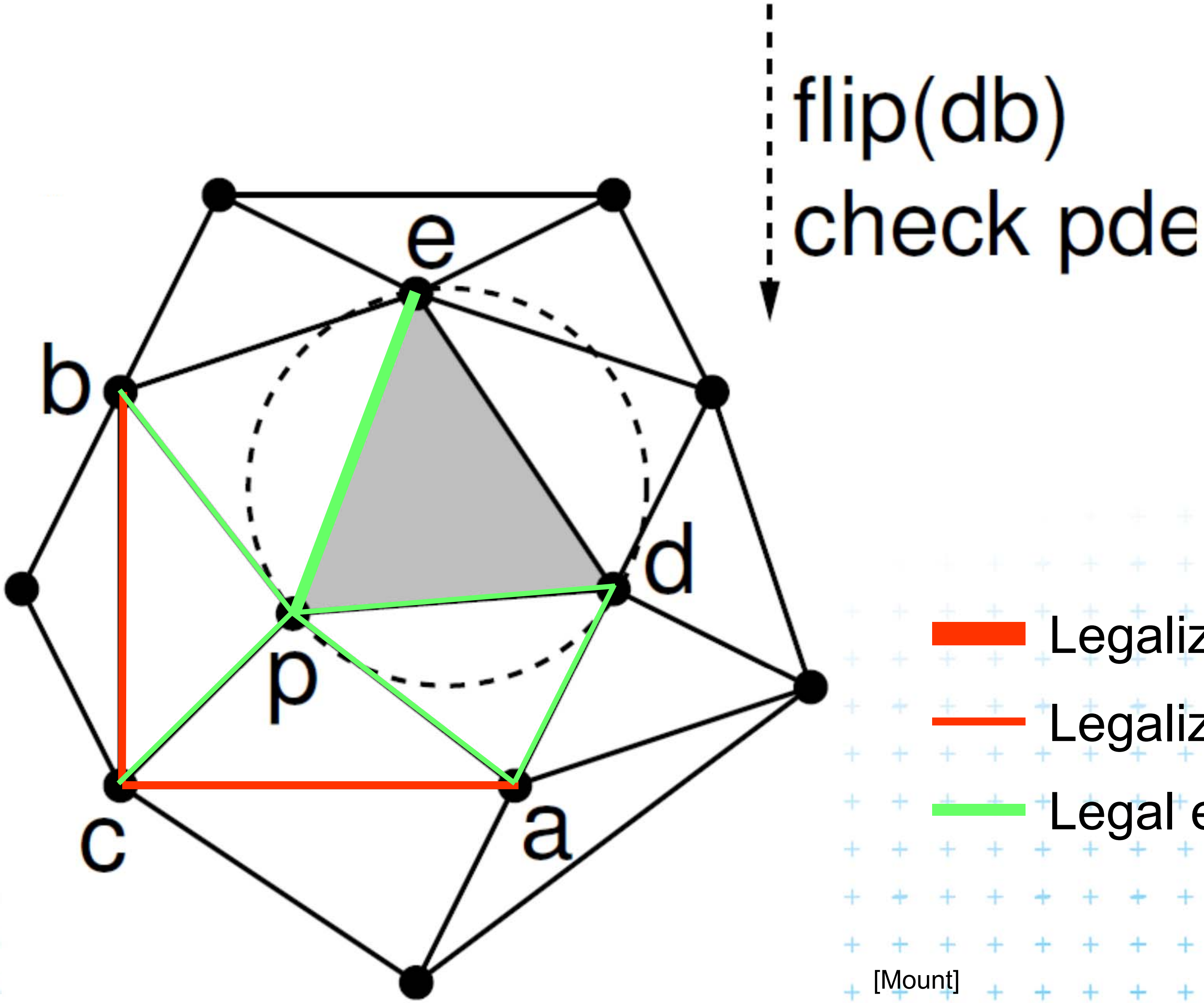
[Mount]

Delaunay triangulation – other point insert



[Mount]

Delaunay triangulation – other point insert



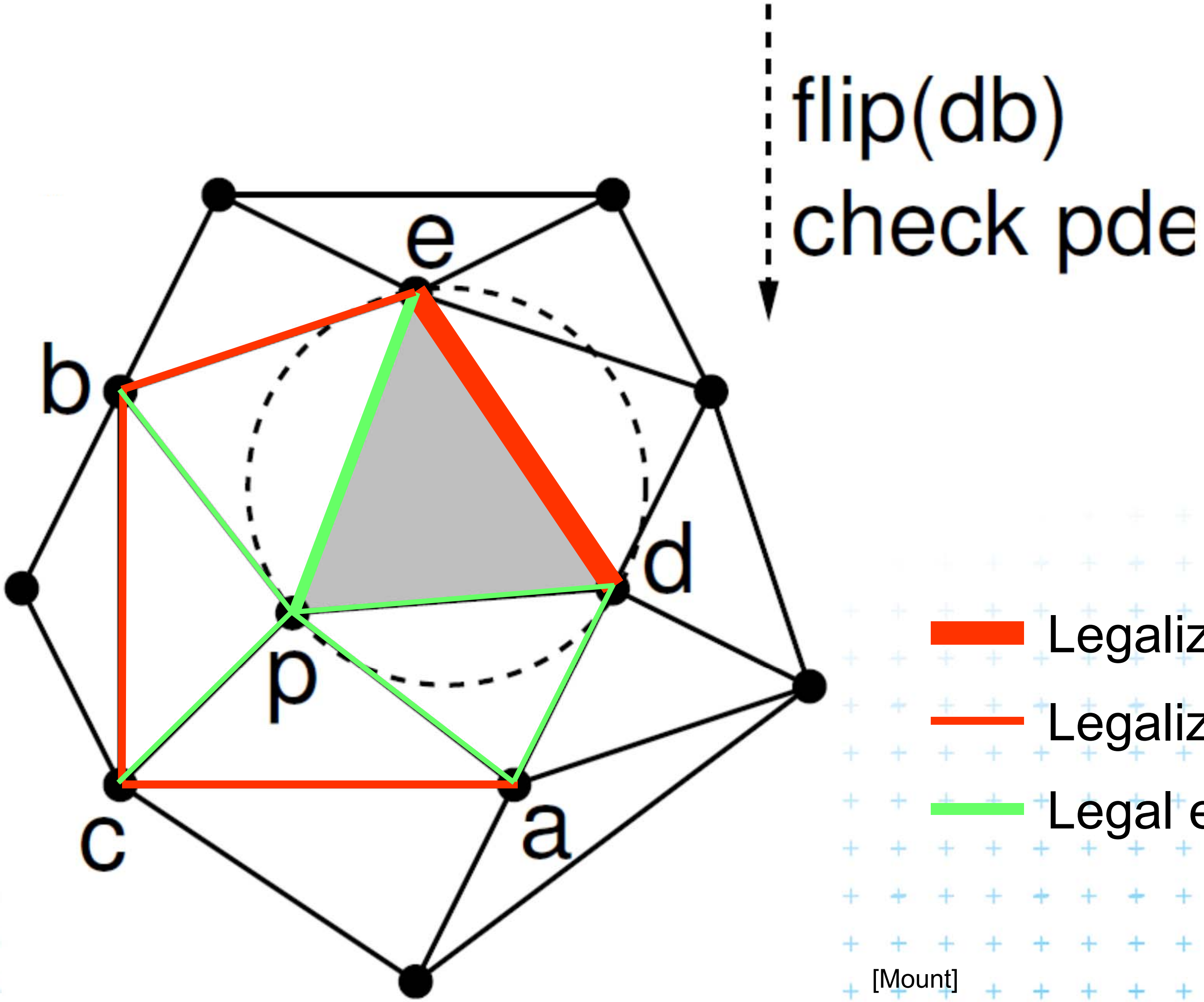
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



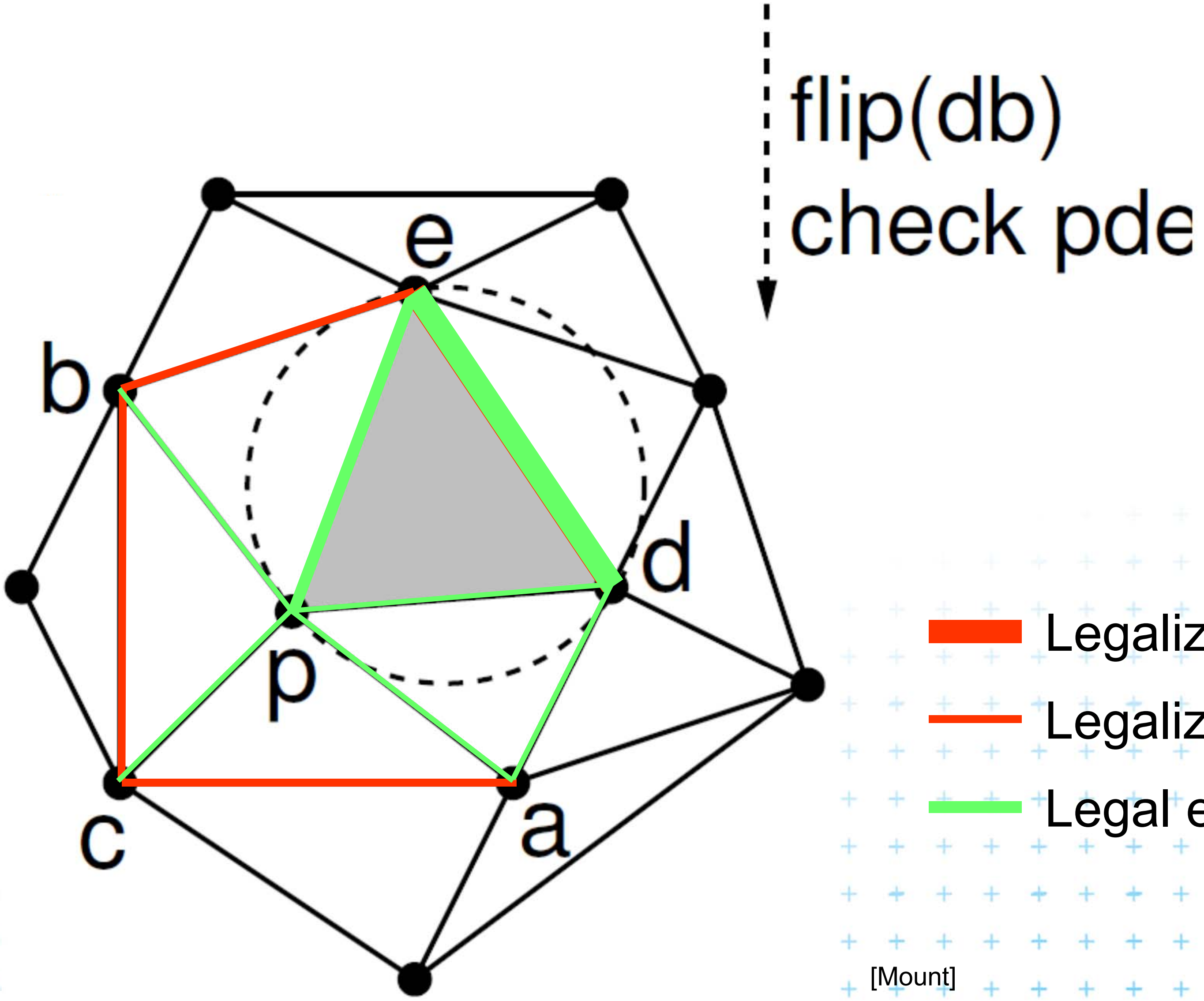
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



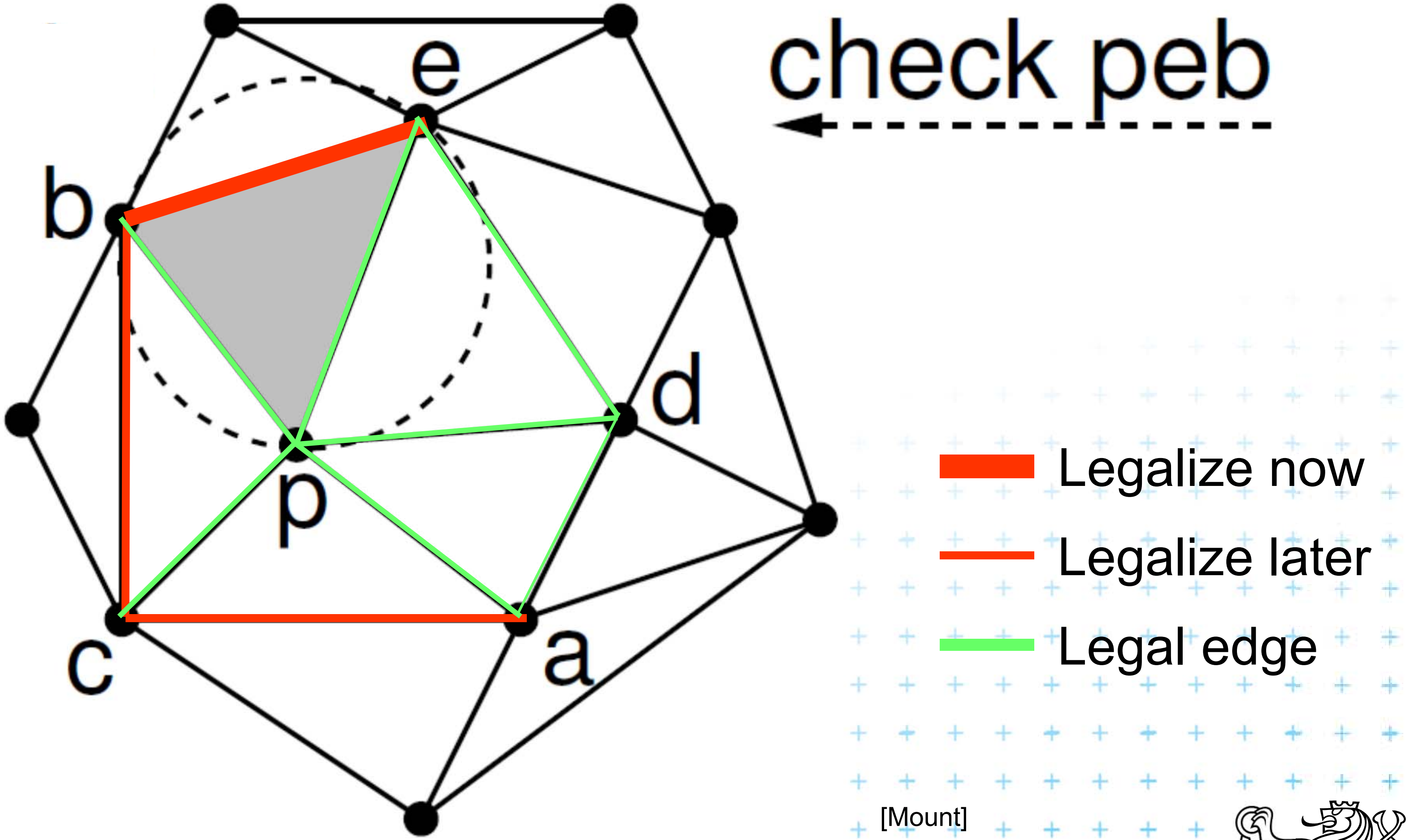
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

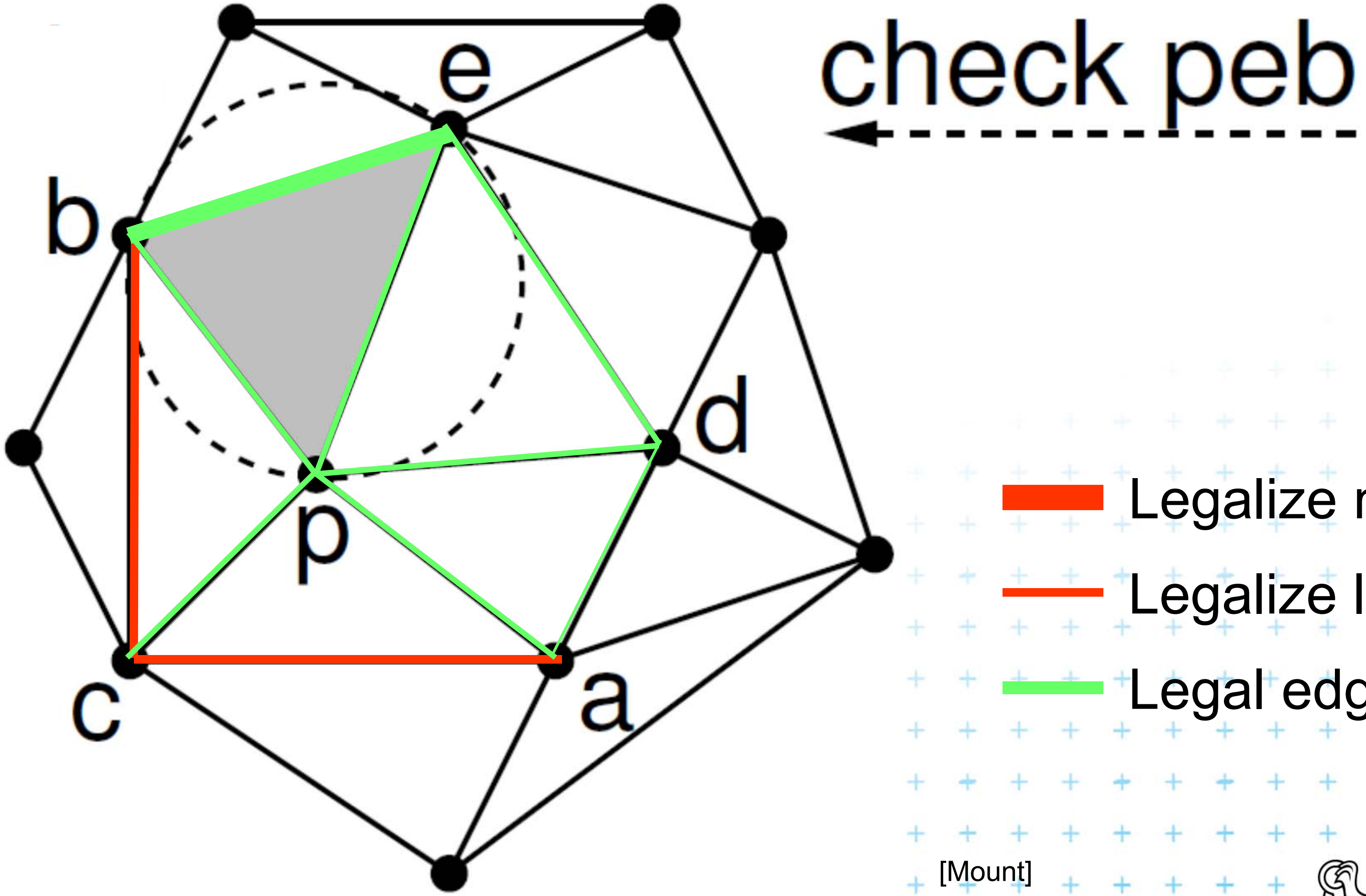
[Mount]

Delaunay triangulation – other point insert



[Mount]

Delaunay triangulation – other point insert



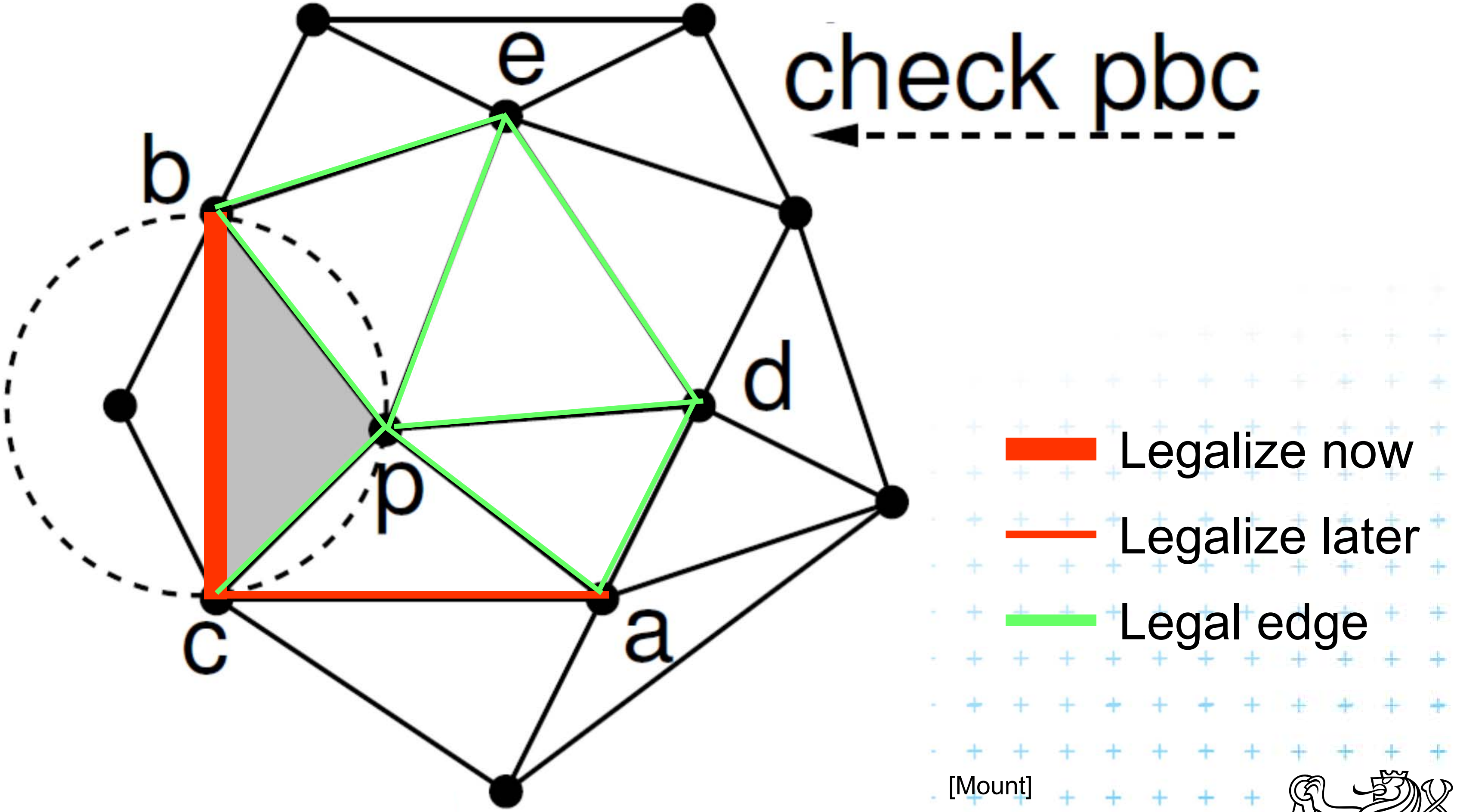
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

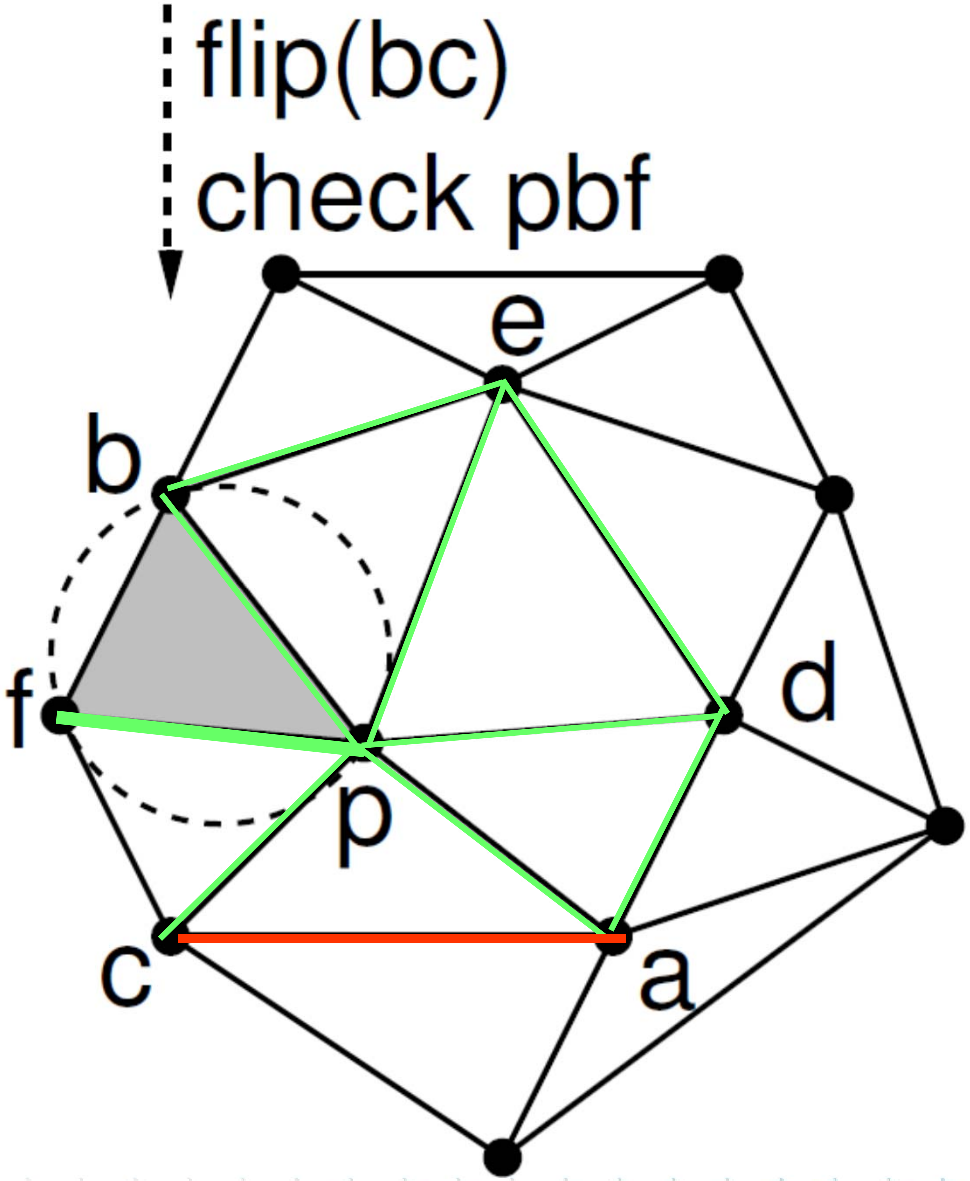
[Mount]

Delaunay triangulation – other point insert



[Mount]

Delaunay triangulation – other point insert



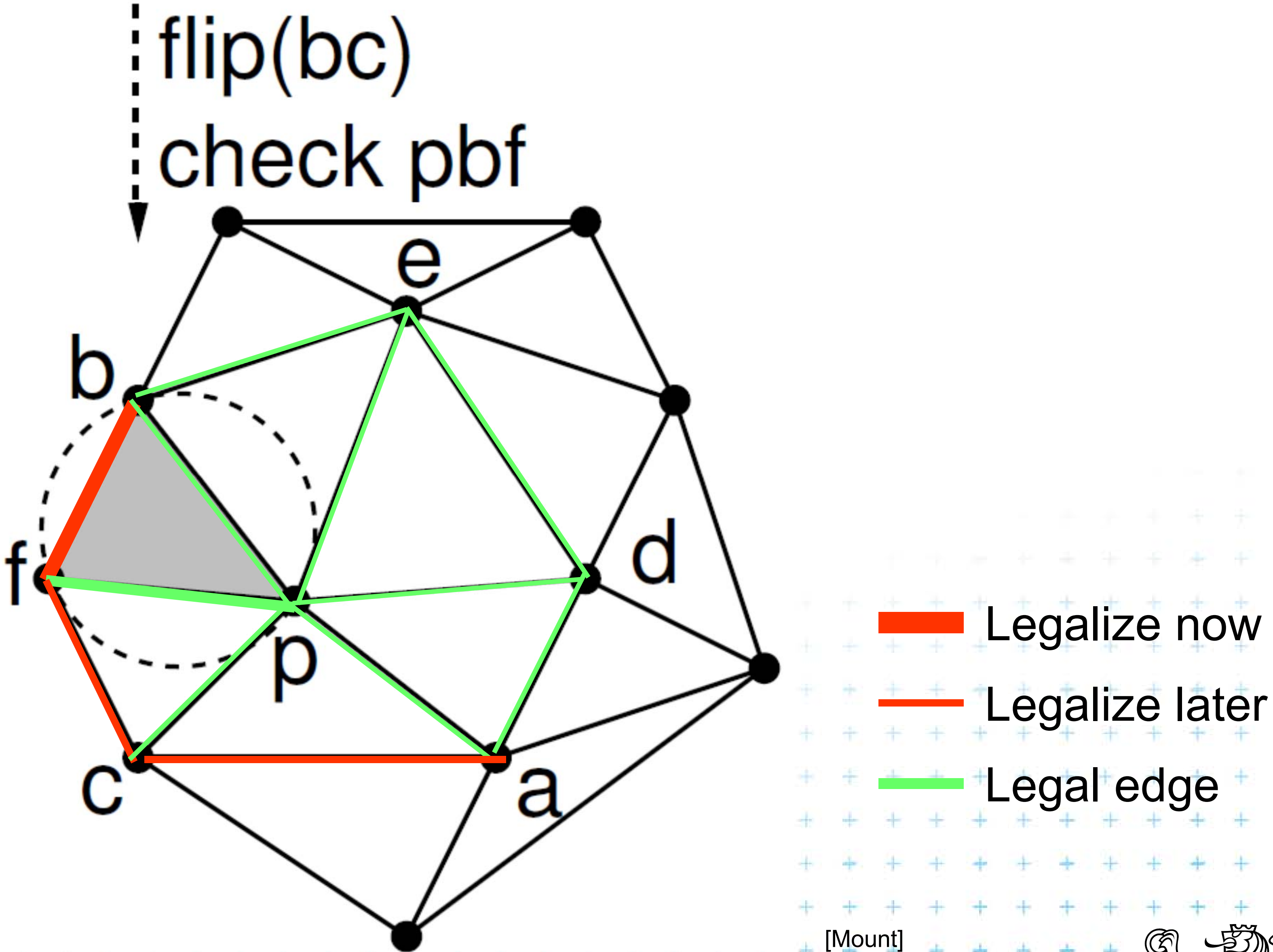
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

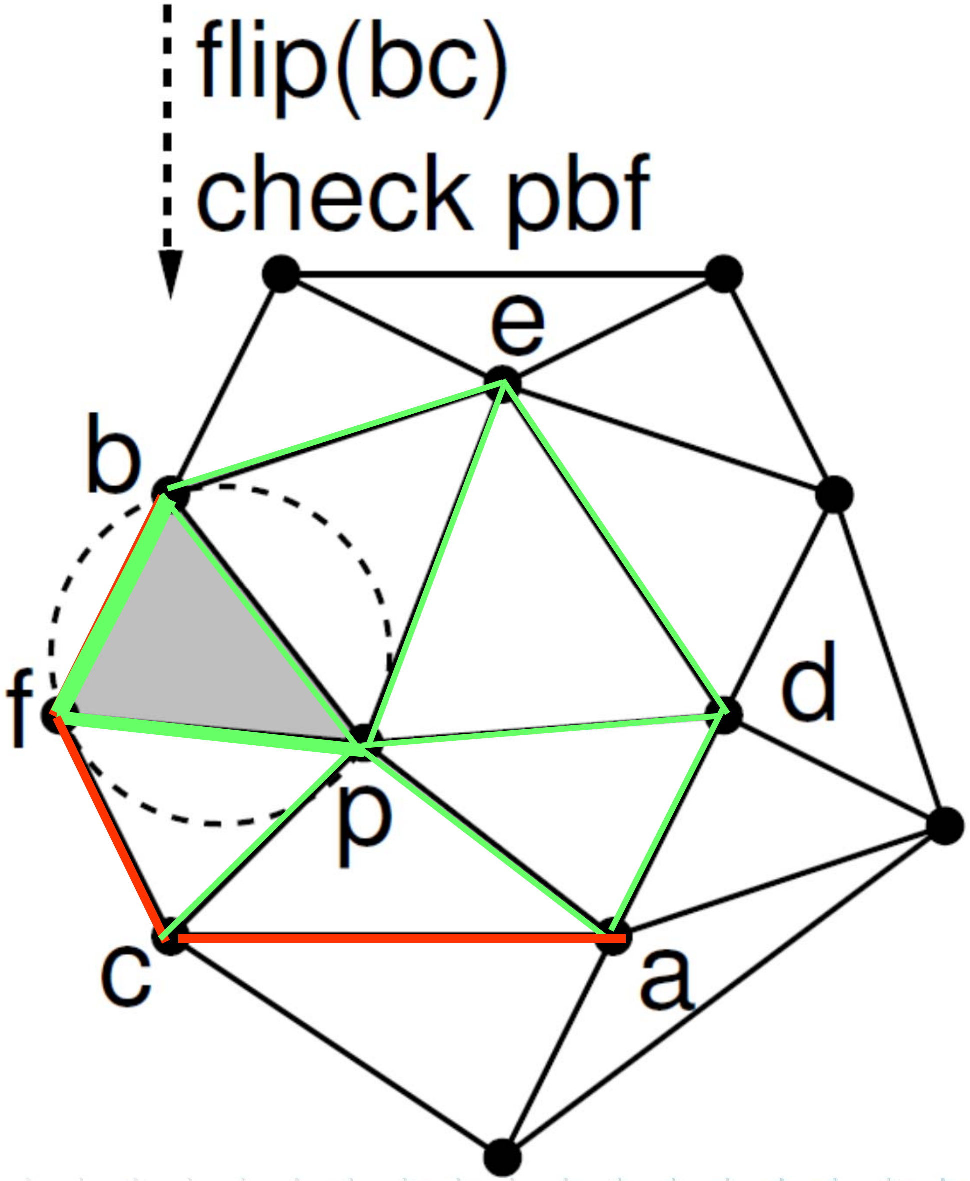
[Mount]

Delaunay triangulation – other point insert



[Mount]

Delaunay triangulation – other point insert



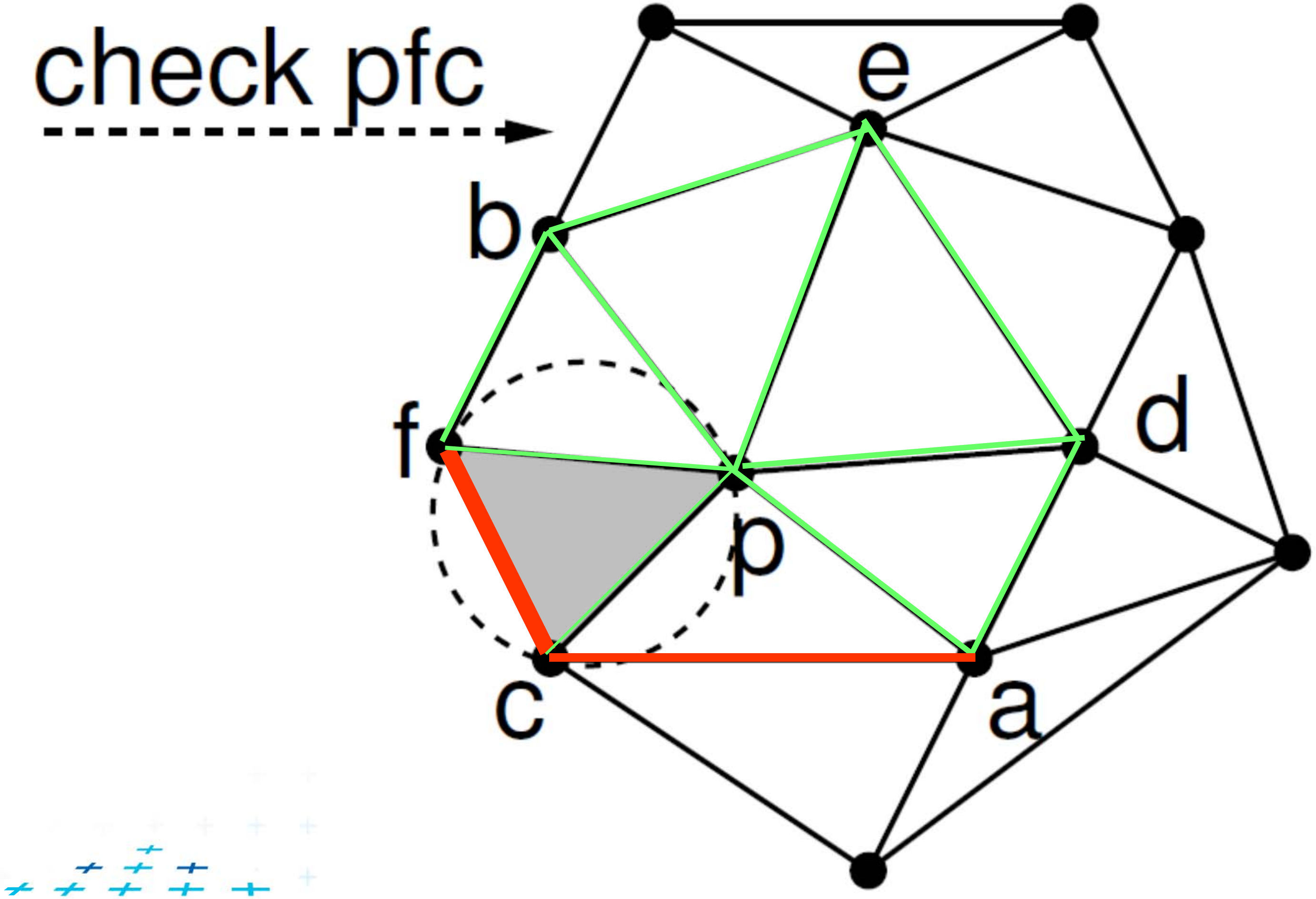
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



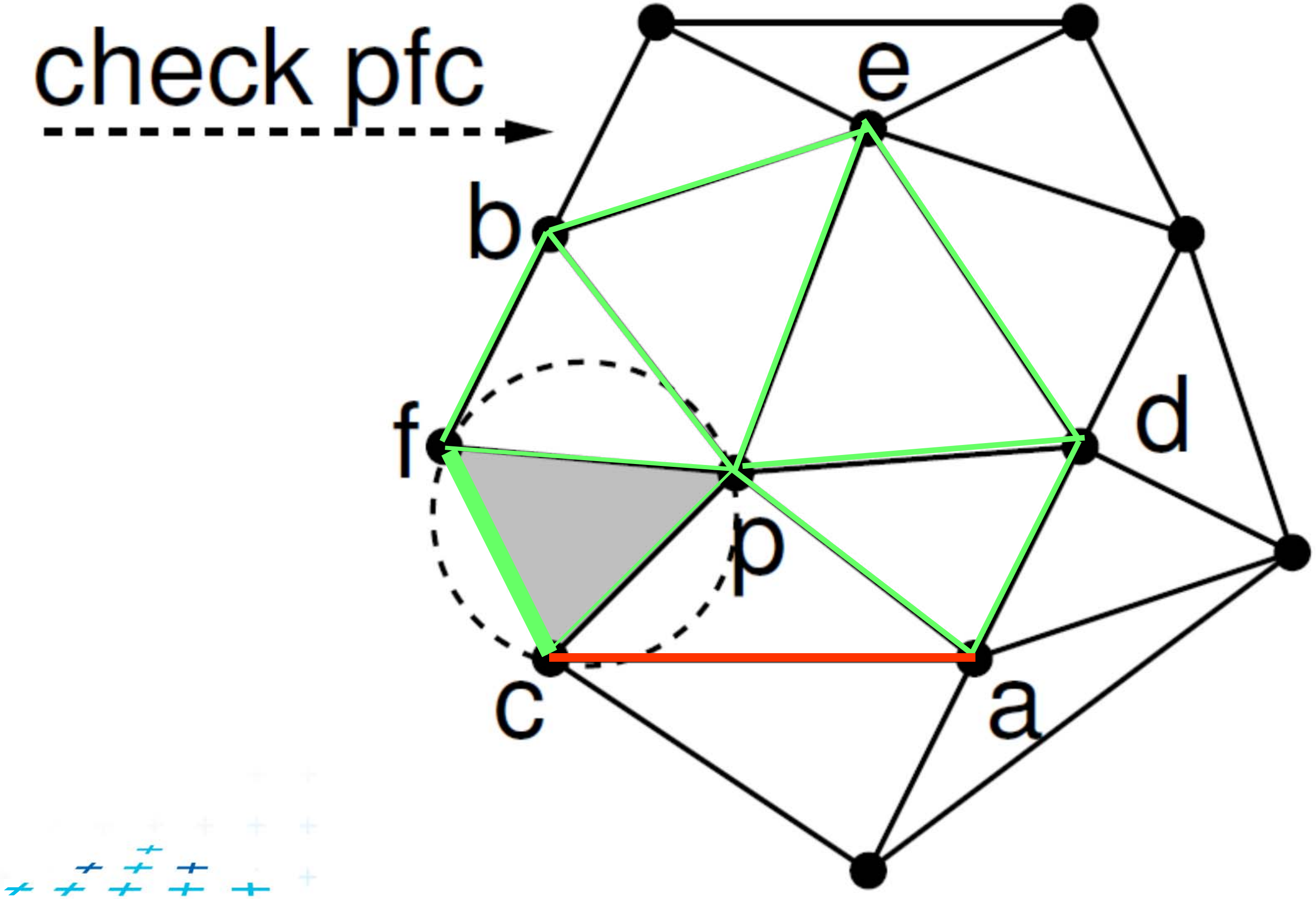
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



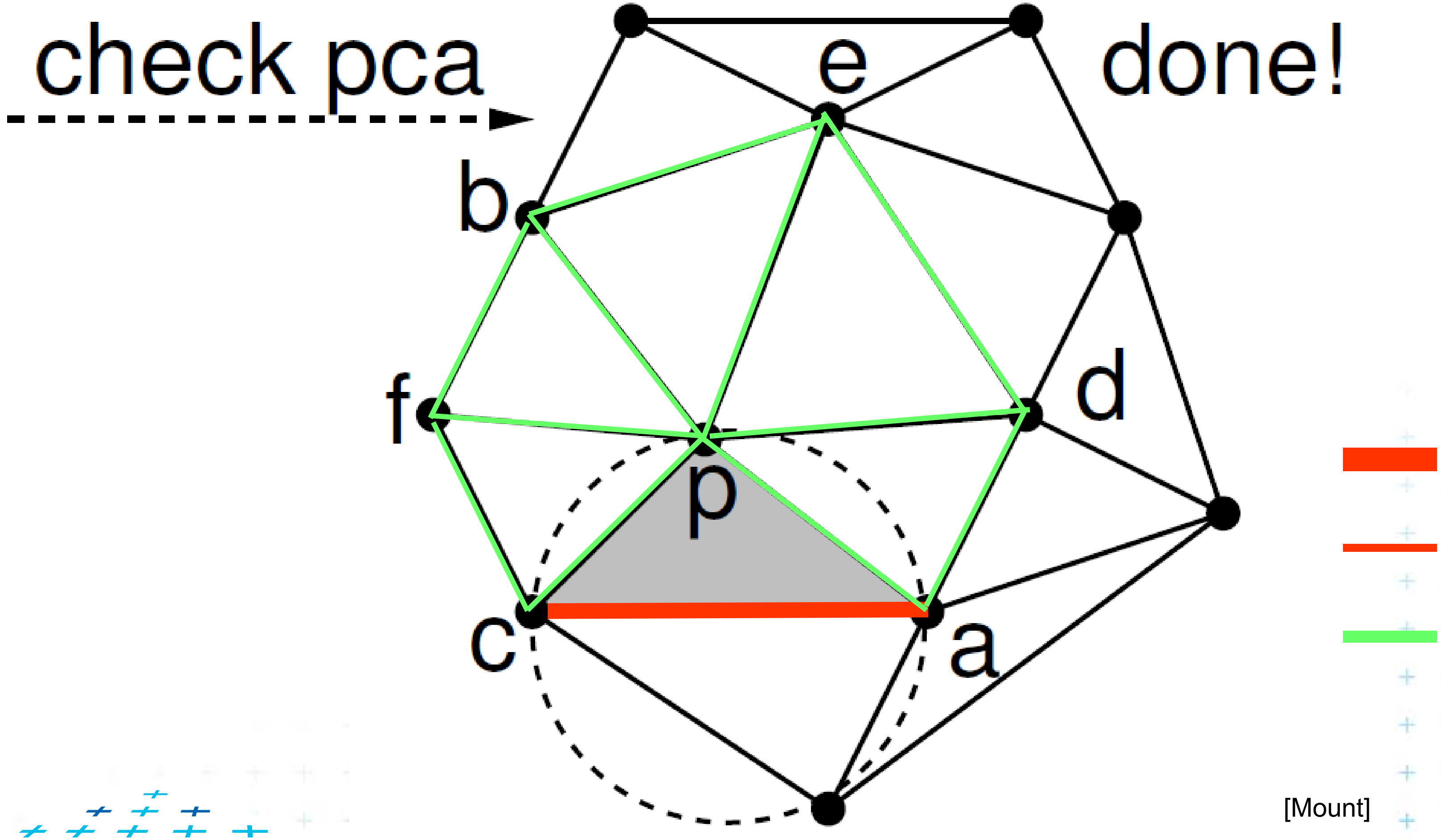
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



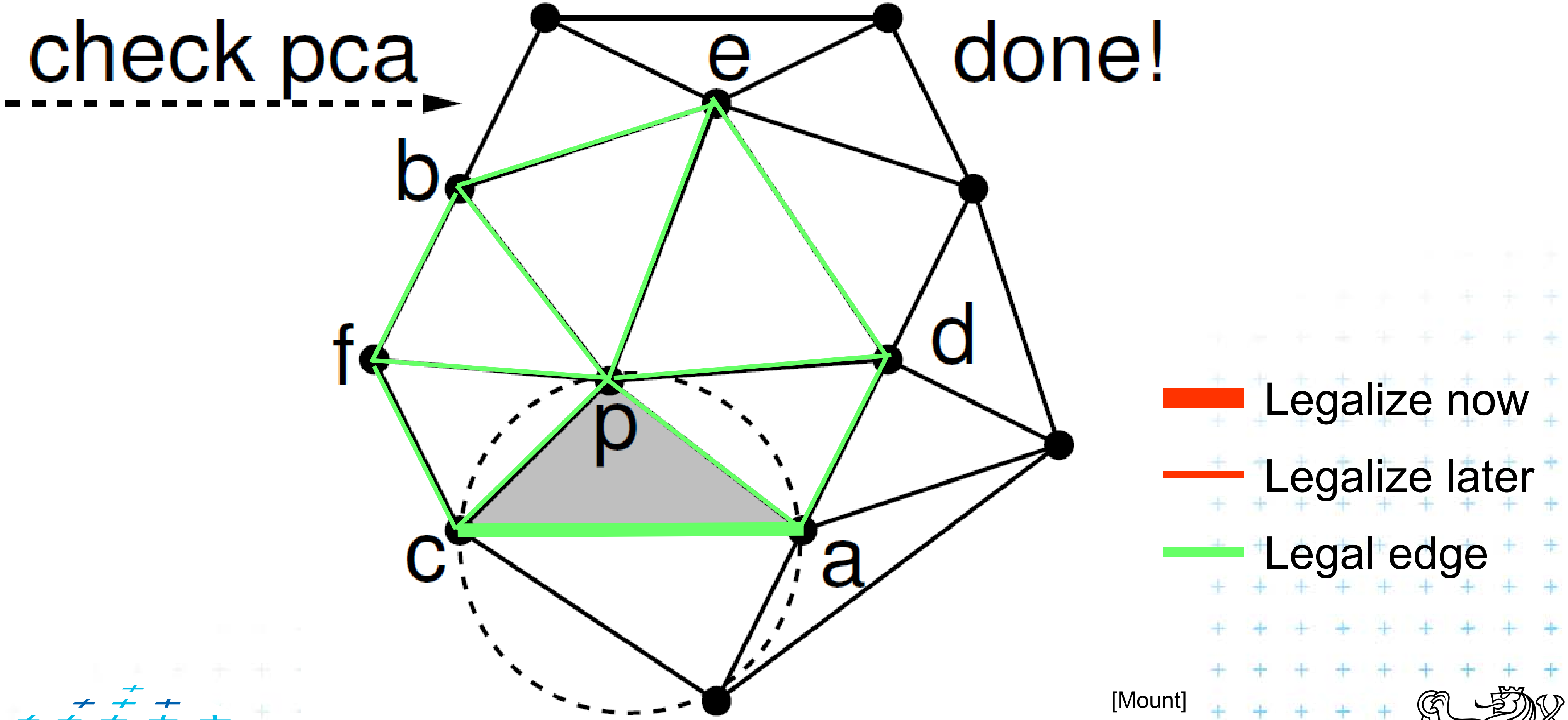
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

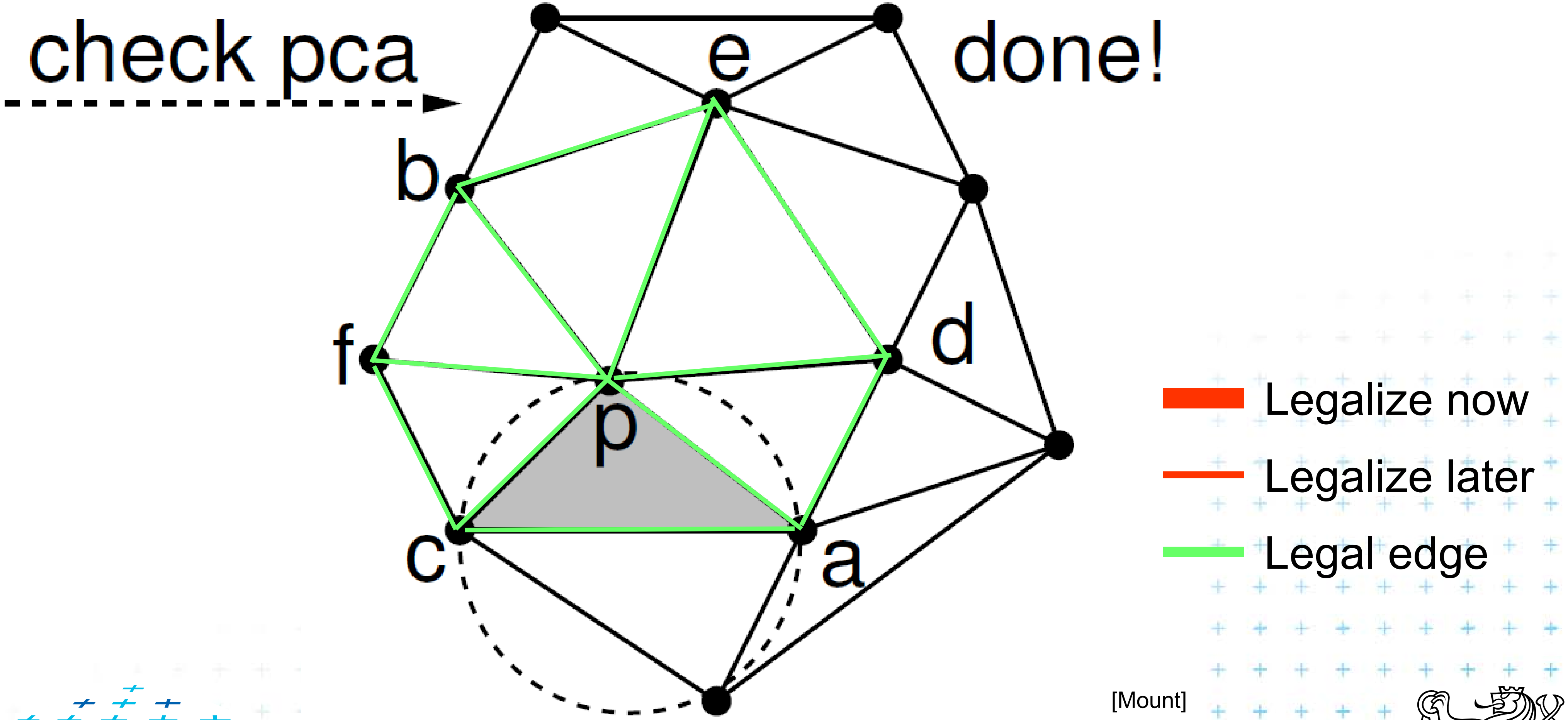
[Mount]

Delaunay triangulation – other point insert



[Mount]

Delaunay triangulation – other point insert



[Mount]

Correctness of the algorithm

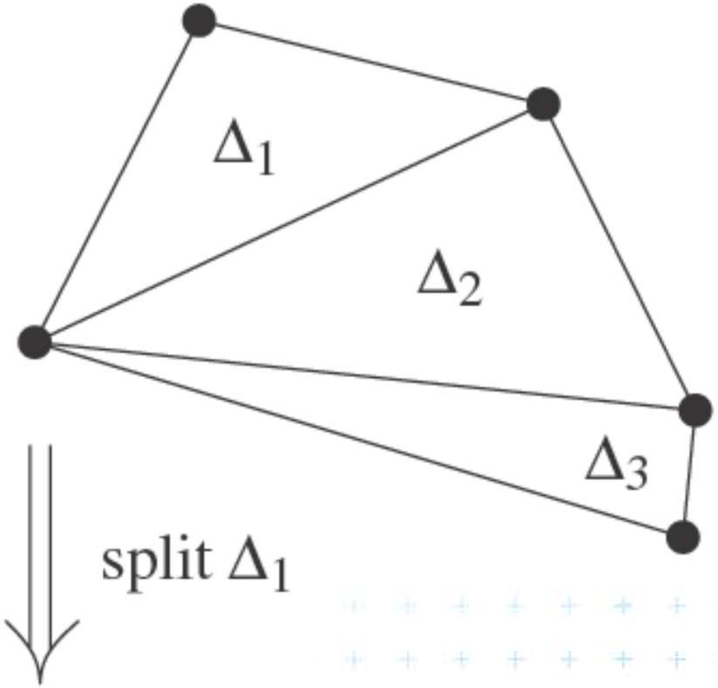
- Every new edge (created due to insertion of *p)*
  - is incident to *p*
  - must be legal
    => no need to test them
- Edge can only become illegal if one of its incident triangle changes
  - Algorithm tests any edge that may become illegal
    => the algorithm is correct
- Every edge flip makes the angle-vector larger
  => algorithm can never get into infinite loop

Point location data structure

- For finding a triangle $abc \in T$ containing p
  - Leaves for active (current) triangles
  - Internal nodes for destroyed triangles
  - Links to new triangles
- Search p: start in root (initial triangle)
  - In each inner node of T:
    - Check all children (max three)
    - Descend to child containing p

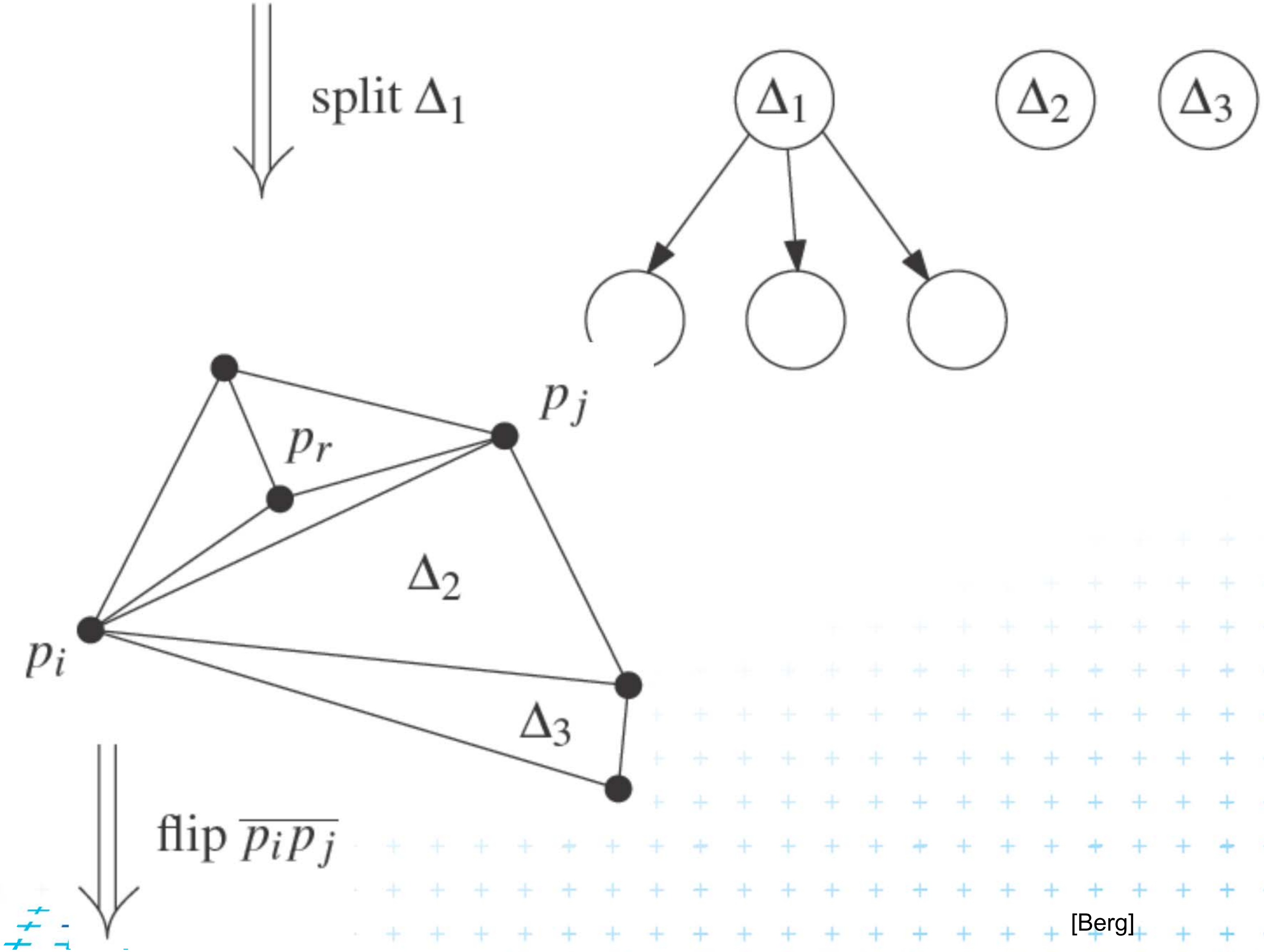
Point location data structure

Simplified
- it should also contain the root node



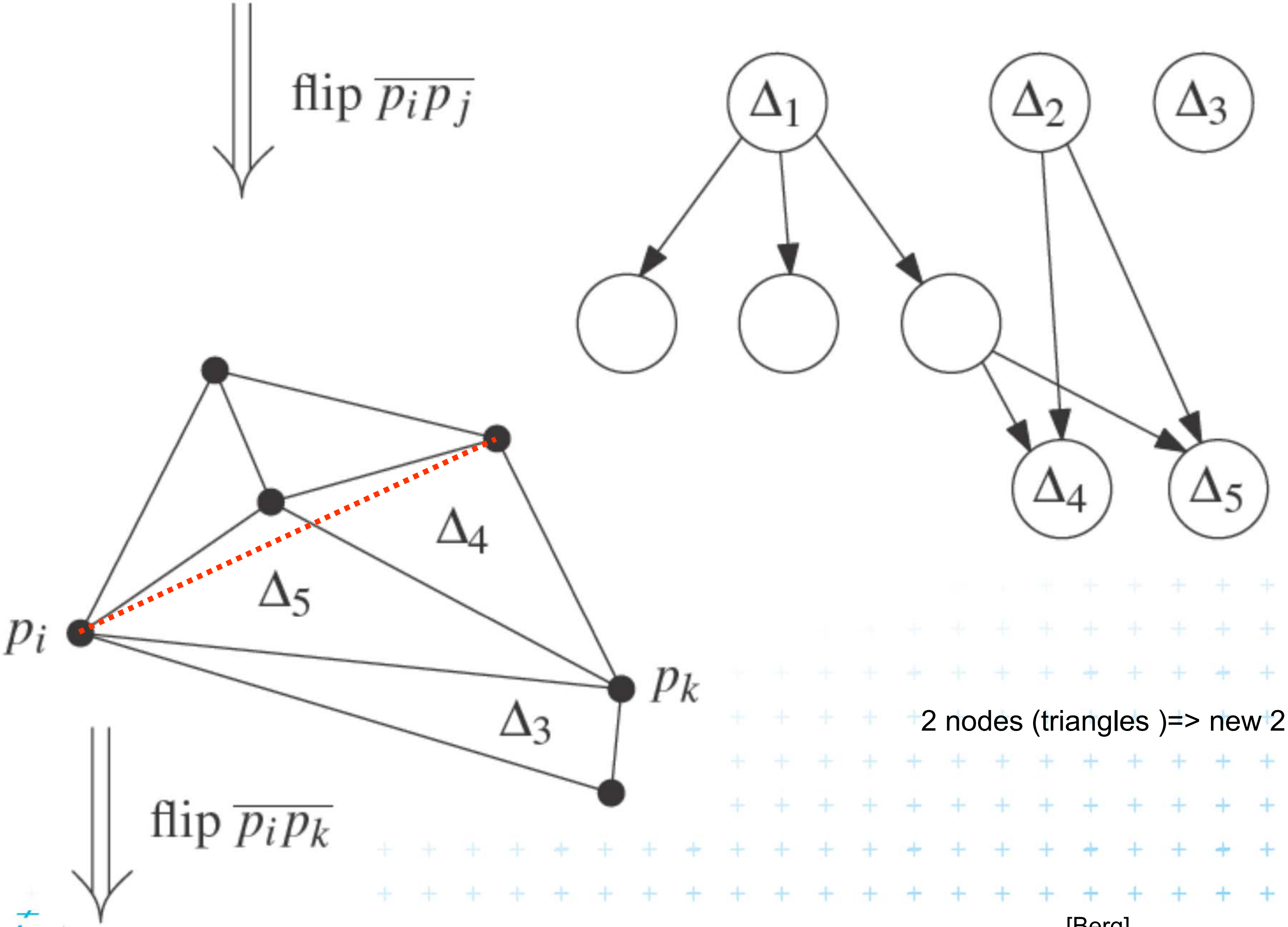
[Berg]

Point location data structure



[Berg]

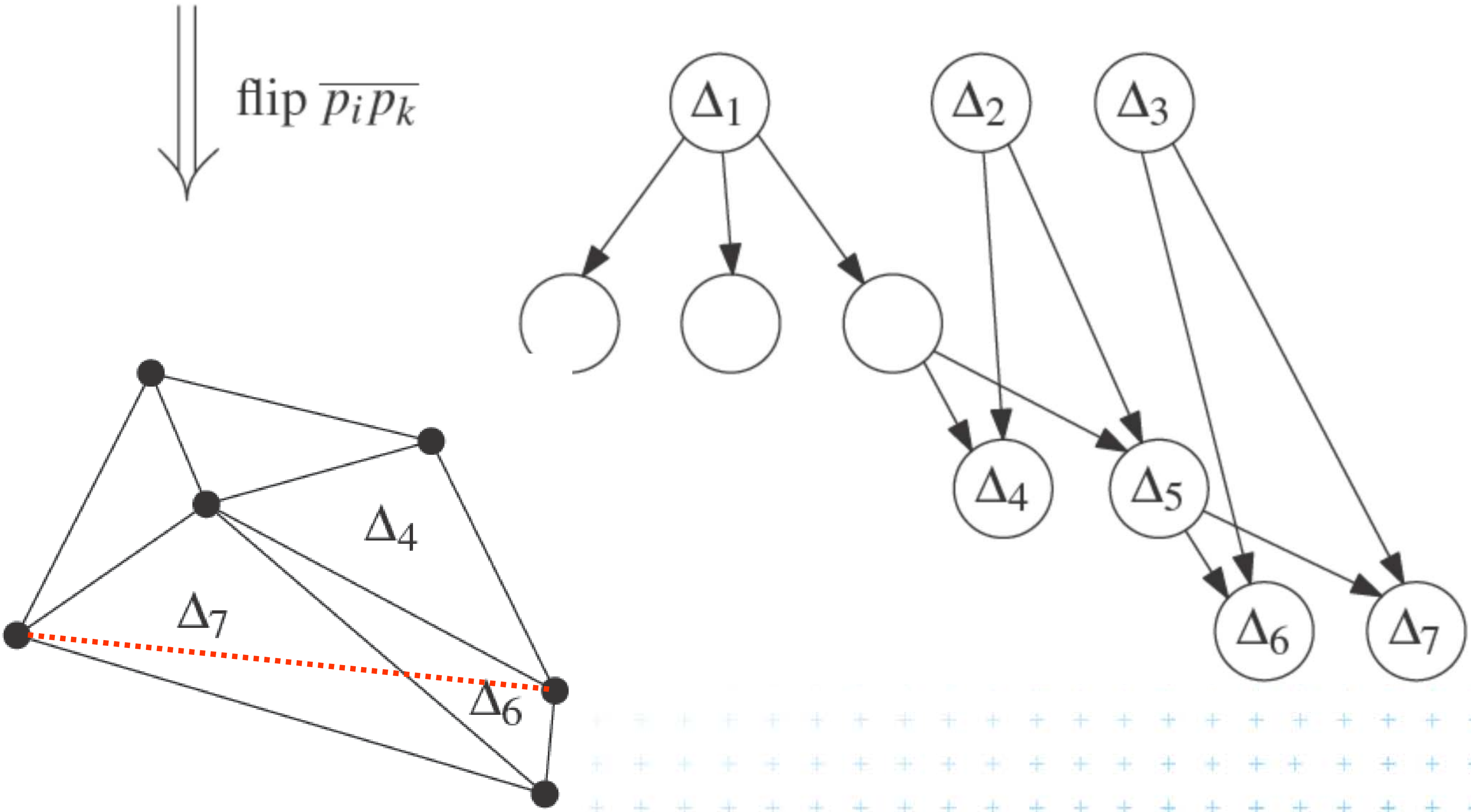
Point location data structure



2 nodes (triangles)=> new 2 nodes

[Berg]

Point location data structure

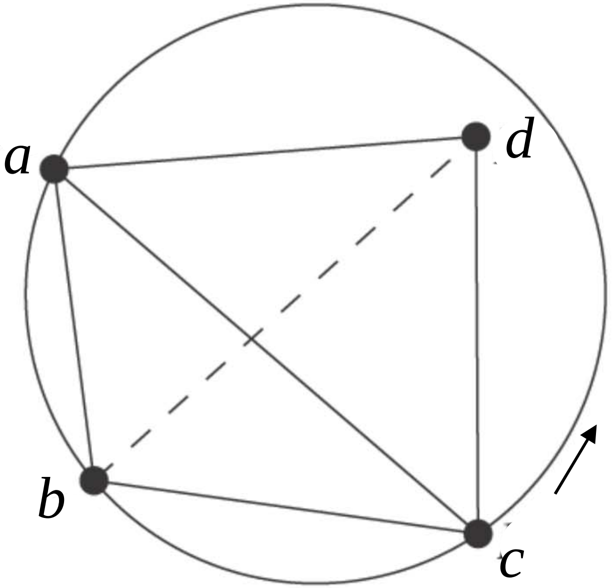


[Berg]

InCircle test

- *a,b,c* are counterclockwise in the plane
- Test, if *d* lies to the left of the oriented circle through *a,b,c*

$$\mathrm{inCircle}(a,b,c,d) = \det \begin{pmatrix} a_x & a_y & a_x^2 + a_y^2 & 1 \\ b_x & b_y & b_x^2 + b_y^2 & 1 \\ c_x & c_y & c_x^2 + c_y^2 & 1 \\ d_x & d_y & d_x^2 + d_y^2 & 1 \end{pmatrix} > 0$$

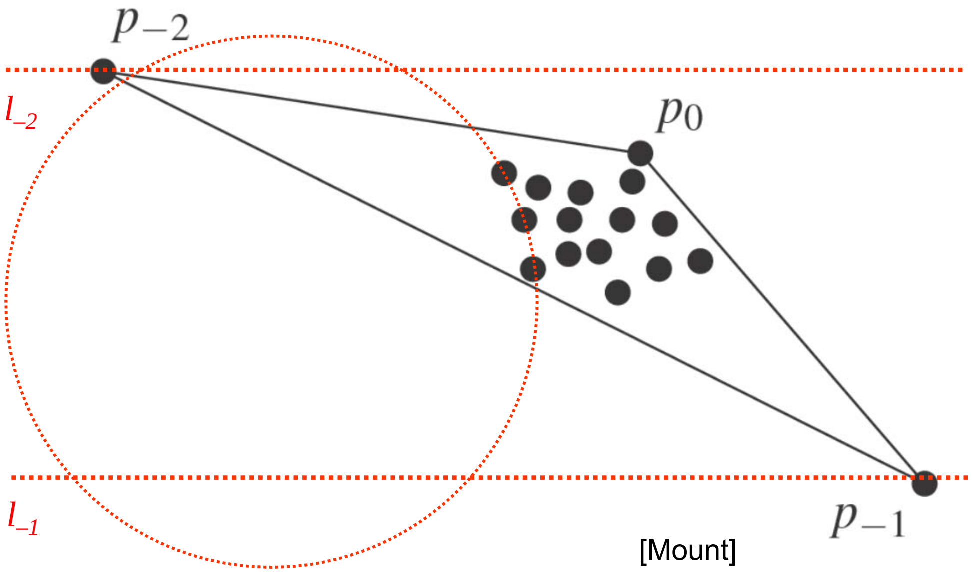


[Mount]

Creation of the initial triangle

Idea: For given points set P:

- Initial triangle $p_{-2}p_{-1}p_0$
  - Must contain all points of P
  - Must not be (none of its points) in any circle defined by non-collinear points of P
- l_{-2} = horizontal line above P
- l_{-1} = horizontal line below P
- p_{-2} = lies on l_{-2} as far left that p_{-2} lies outside every circle
- p_{-1} = lies on l_{-1} as far right that p_{-1} lies outside every circle defined by 3 non-collinear points of P



[Mount]

Symbolical tests with this triangle => p_{-1} and p_{-2} always out

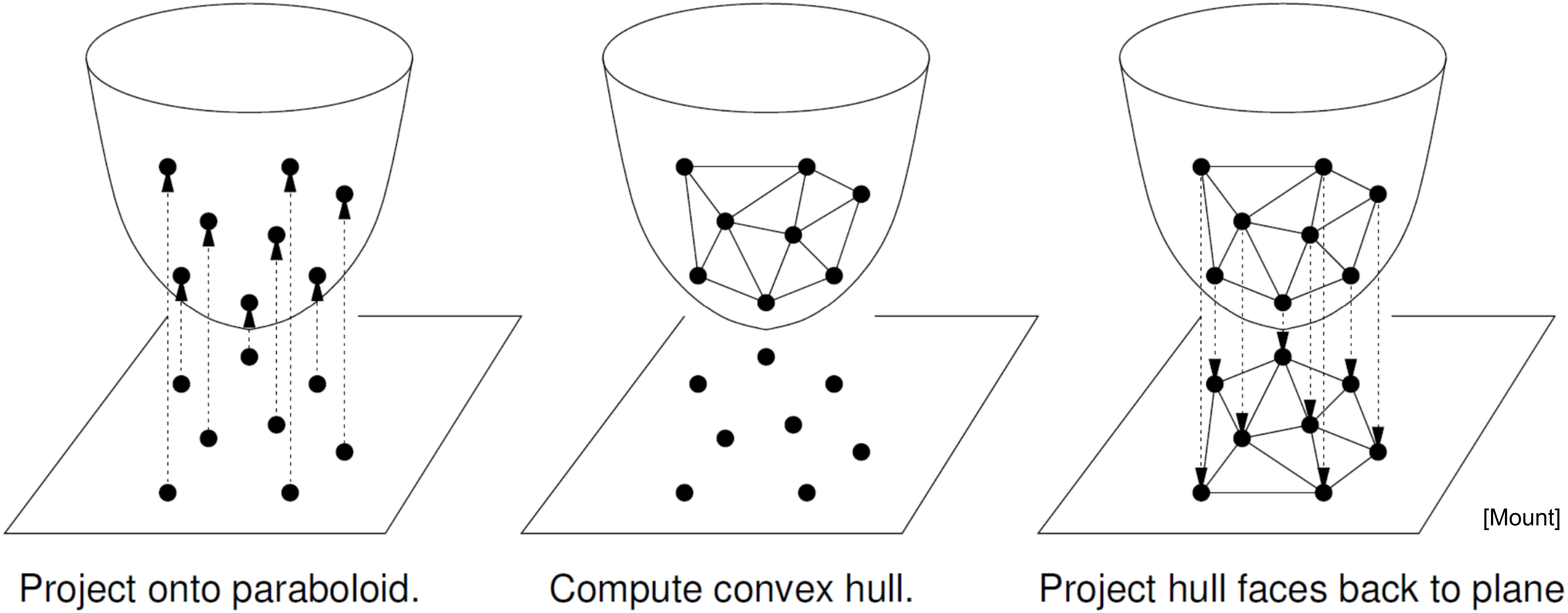
Complexity of incremental DT algorithm

- Delaunay triangulation of a set *P* in the plane can be computed in
  - O(n log n) expected time
  - using O(n) storage
- For details see [Berg, Section 9.4]

  Idea
  - expected number of created triangles is 9n+1
  - expected search O(log n) in the search structure done n times for n inserted points

Delaunay triangulations and Convex hulls

- Delaunay triangulation in R^d can be computed as part of the convex hull in R^{d+1} (lower CH)
- 2D: Connection is the paraboloid: $z = x^2 + y^2$



[Mount]

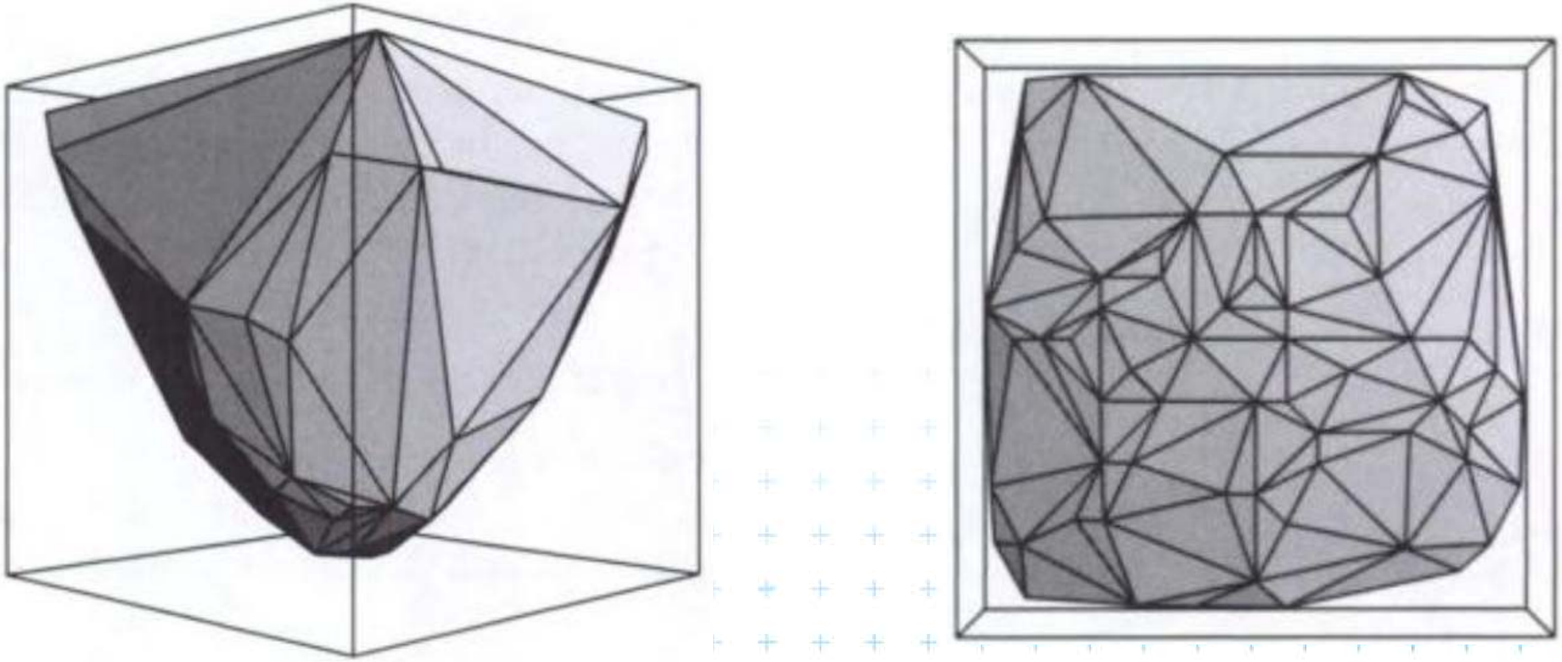
Project onto paraboloid. Compute convex hull. Project hull faces back to plane.

Vertical projection of points to paraboloid

- Vertical projection of 2D point to paraboloid in 3D

$$(x, y) \rightarrow (x, y, x^2 + y^2)$$

- Lower convex hull
= portion of CH visible from $z = -\infty$ (forms DT)

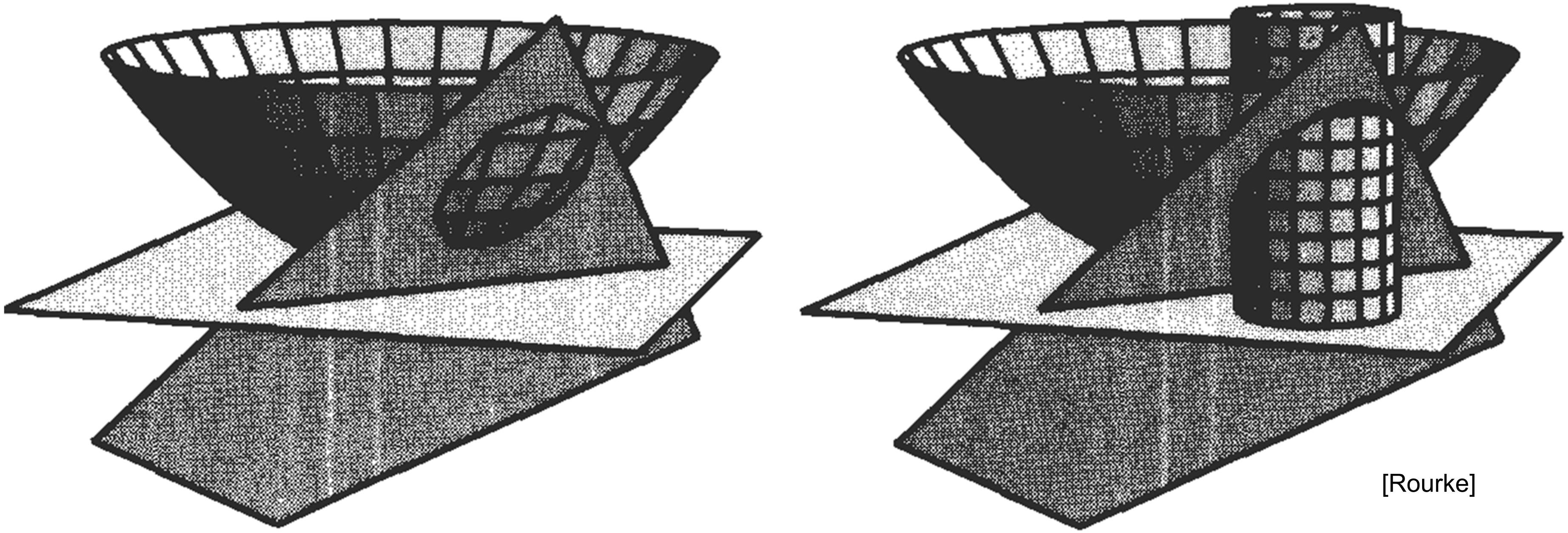


[Rourke]

Relation between CH and DT

- Delaunay condition (2D)
  Points $p,q,r \in S$ form a Delaunay triangle **iff** the circumcircle of p,q,r is empty (contains no point)
- Convex hull condition (3D)
  Points $p',q',r' \in S'$ form a face of $CH(S')$ **iff** the plane passing through p',q',r' is supporting S'
  - all other points lie to one side of the plane
  - plane passing through p',q',r' is supporting hyperplane of the convex hull CH(S')

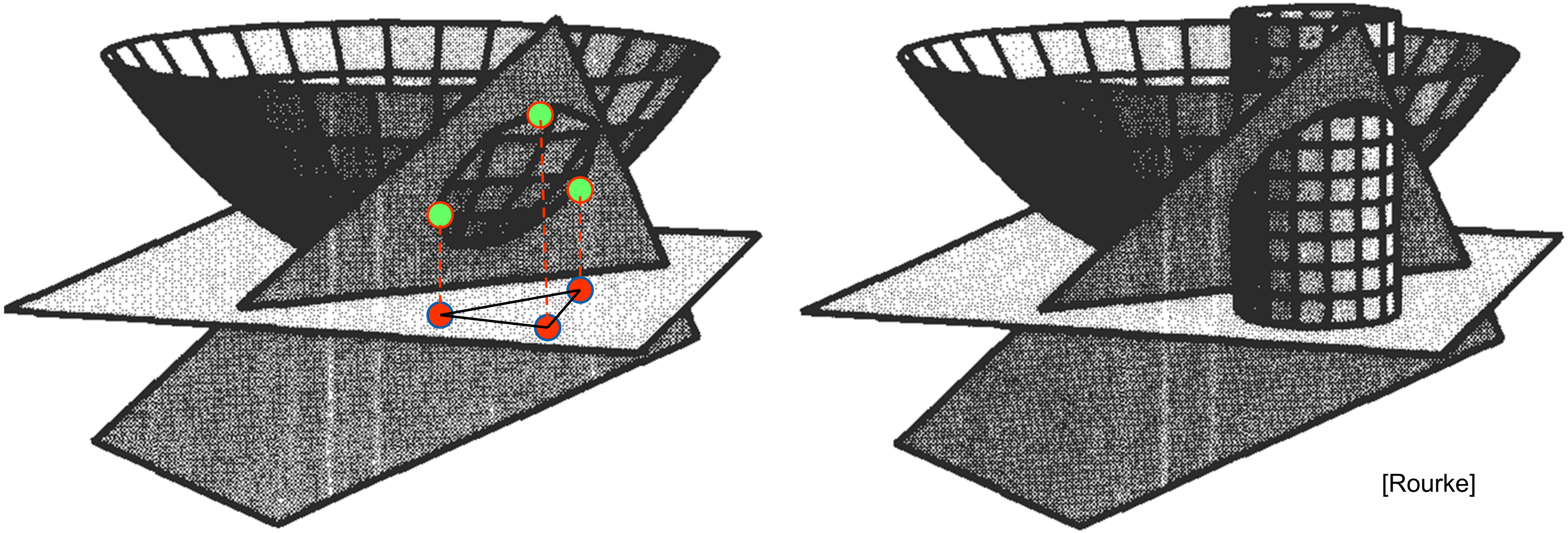
Relation between CH and DT



[Rourke]

- 4 distinct points *p,q,r,s* in the plane, and let *p', q', r', s'* be their respective projections onto the paraboloid, $z = x^2 + y^2$.
- The point *s* lies within the circumcircle of *pqr* **iff** s' lies on the lower side of the plane passing through *p', q', r'*.

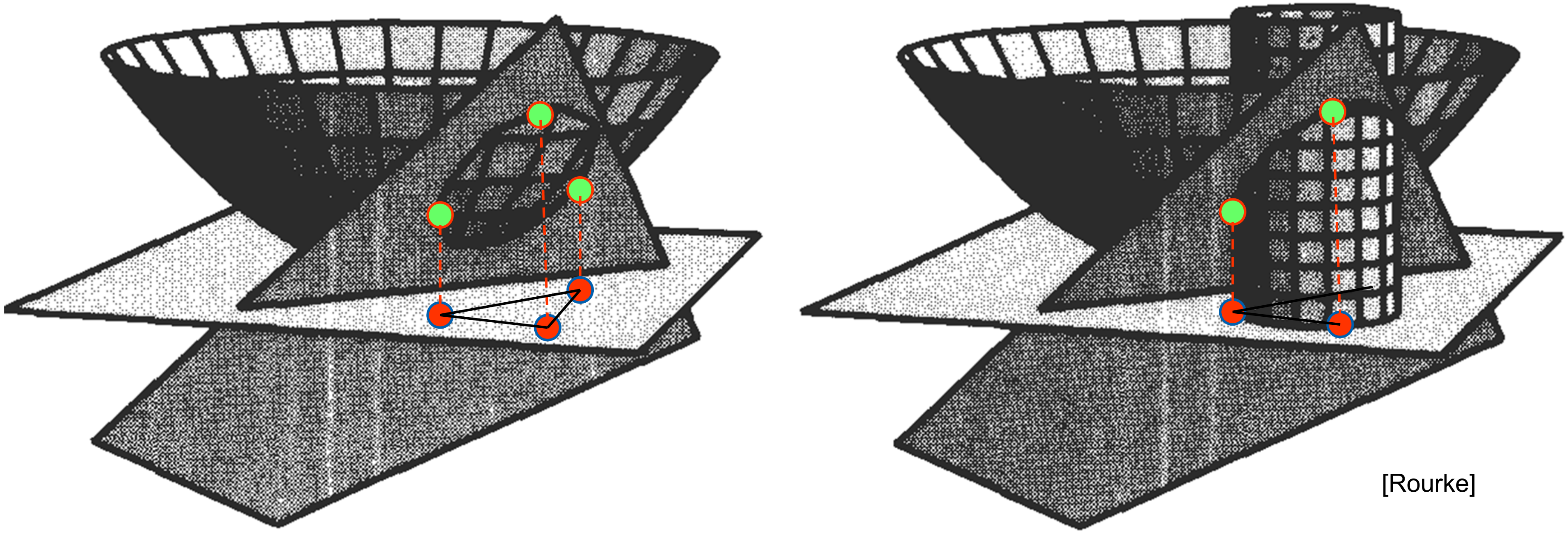
Relation between CH and DT



[Rourke]

- 4 distinct points *p,q,r,s* in the plane, and let *p', q', r', s'* be their respective projections onto the paraboloid, $z = x^2 + y^2$.
- The point *s* lies within the circumcircle of *pqr* **iff** s' lies on the lower side of the plane passing through *p', q', r'*.

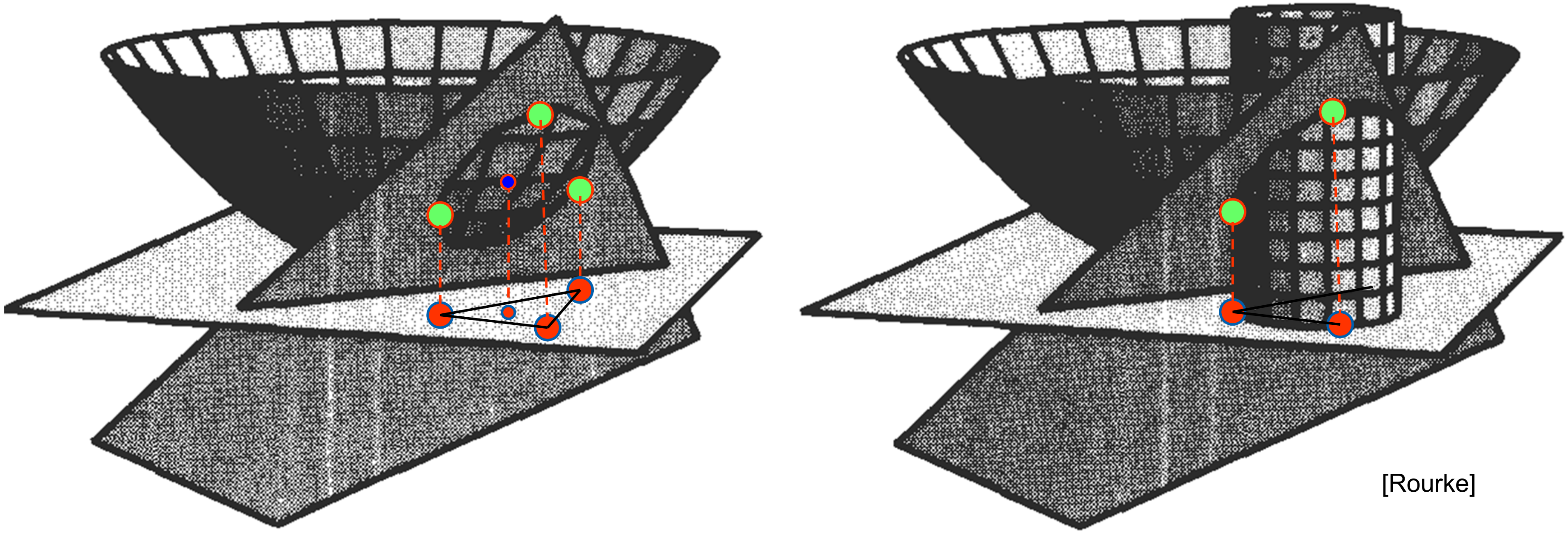
Relation between CH and DT



[Rourke]

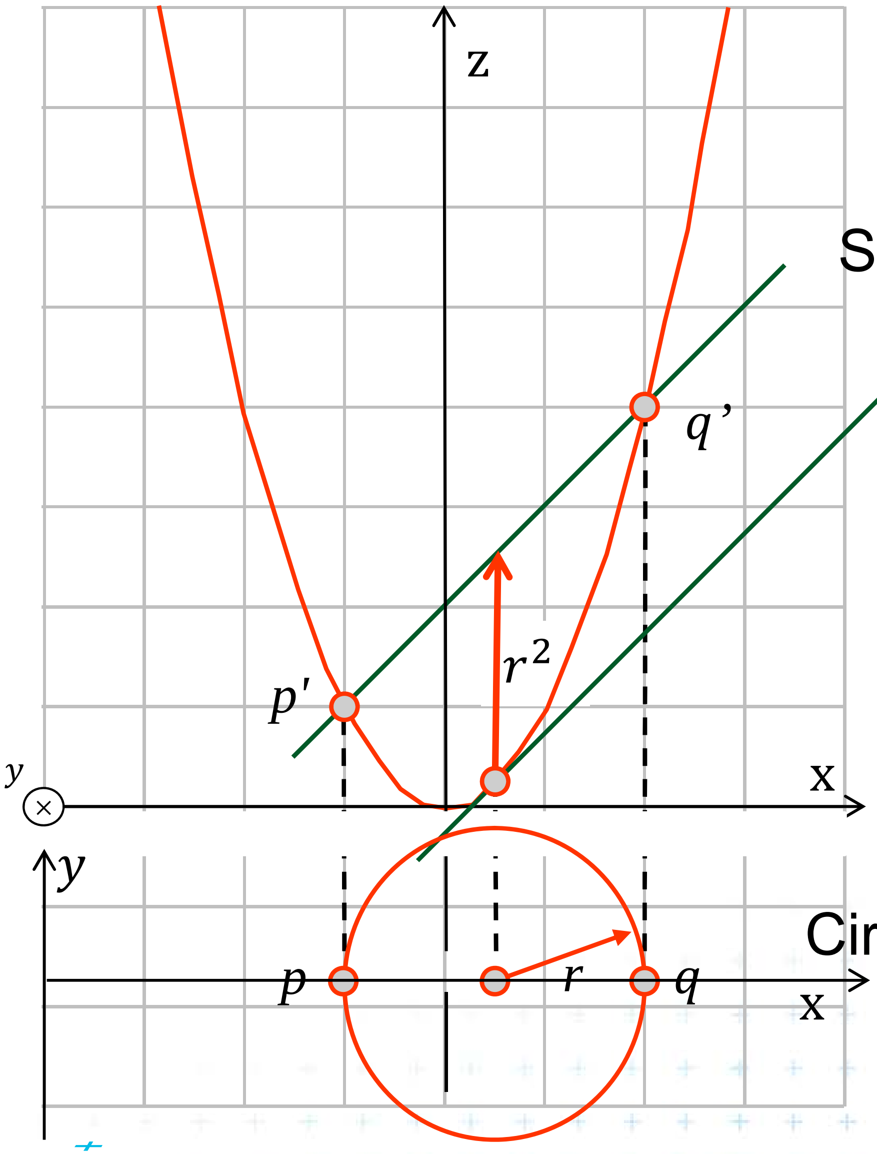
- 4 distinct points *p,q,r,s* in the plane, and let *p', q', r', s'* be their respective projections onto the paraboloid, $z = x^2 + y^2$.
- The point *s* lies within the circumcircle of *pqr* **iff** s' lies on the lower side of the plane passing through *p', q', r'*.

Relation between CH and DT


[Rourke]

- 4 distinct points *p,q,r,s* in the plane, and let *p', q', r', s'* be their respective projections onto the paraboloid, $z = x^2 + y^2$.
- The point *s* lies within the circumcircle of *pqr* **iff** s' lies on the lower side of the plane passing through *p', q', r'*.

Tangent and secant planes



Cross section of the paraboloid

Secant plane

Tangent plane

Circle in *xy* plane

(a,b)

Tangent plane to paraboloid

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$
- Paraboloid $z = x^2 + y^2$
- Derivation at this point

$$\frac{\partial z}{\partial x} = 2x \qquad \frac{\partial z}{\partial y} = 2y$$

- Evaluates to $2a$ and $2b$
- Plane: $z = 2ax + 2by + \gamma$

$$a^2 + b^2 = 2a.a + 2b.b + \gamma \qquad \gamma = -(a^2 + b^2)$$

- Tangent plane through point $(a, b, a^2 + b^2)$

$$z = 2ax + 2by - (a^2 + b^2)$$

[Mount]

Plane intersecting the paraboloid (secant plane)

- Non-vertical tangent plane through $(a, b, a^2 + b^2)$

  $z = 2ax + 2by - (a^2 + b^2)$

- Shift this plane r^2 upwards –> secant plane intersects the paraboloid in an ellipse in 3D

  $z = 2ax + 2by - (a^2 + b^2)+r^2$

- Eliminate *z* (project to 2D) $z = x^2+y^2$

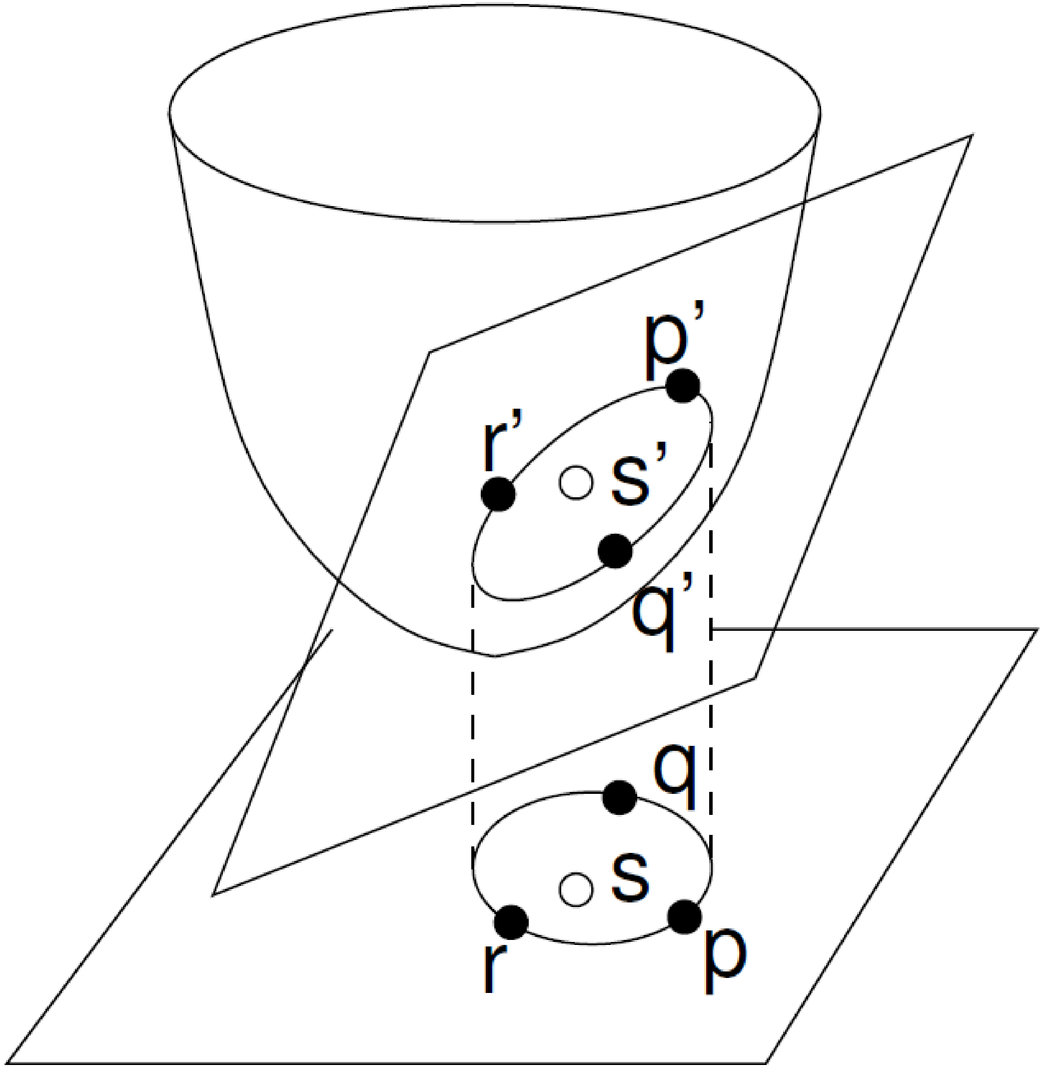
  $x^2+y^2 = 2ax + 2by - (a^2 + b^2)+r^2$

- This is a circle projected to 2D with center (a, b):

$$(x - a)^2 + (y - b)^2 = r^2$$

[Mount]

Secant plane defined by three points

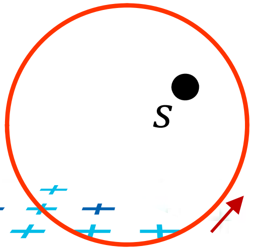


[Mount]

Test inCircle – meaning in 3D

- Points *p,q,r* are counterclockwise in the plane
- Test, if *s* lies in the circumcircle of ∆*pqr* is equal to
  - = test, weather *s'* lies within a lower half space of the plane passing through *p',q',r'* (3D)
  - = test, if quadruple *p',q',r',s'* is positively oriented (3D)
  - = test, if *s lies* to the left of the oriented circle through *pqr* (2D)

$$\mathrm{in}(p,q,r,s) = \det\begin{pmatrix} p_x & p_y & p_x^2+p_y^2 & 1 \\ q_x & q_y & q_x^2+q_y^2 & 1 \\ r_x & r_y & r_x^2+r_y^2 & 1 \\ s_x & s_y & s_x^2+s_y^2 & 1 \end{pmatrix} > 0.$$



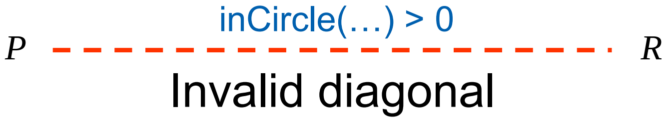
[Mount]

Delaunay triangulation and inCircle test

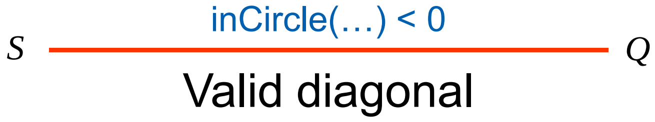
- DT splits each quadrangle by one of its two diagonals
- For a valid diagonal, the fourth point is not inCircle

  => the fourth point is right from the oriented circumcircle (outside)

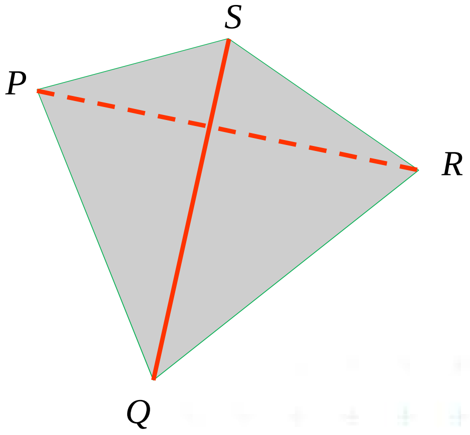
  => inCircle(….) < 0 for CCW orientation
- inCircle(P,Q,R,S) = inCircle(P,R,S,Q) = – inCircle(P,Q,S,R) = – inCircle(S,Q,R,P)



Invalid diagonal



Valid diagonal

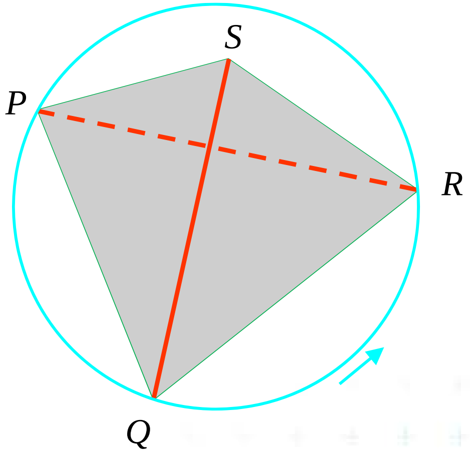


Delaunay triangulation and inCircle test

- DT splits each quadrangle by one of its two diagonals
- For a valid diagonal, the fourth point is not inCircle
  => the fourth point is right from the oriented circumcircle (outside)
  => inCircle(….) < 0 for CCW orientation
- inCircle(P,Q,R,S) = inCircle(P,R,S,Q) = – inCircle(P,Q,S,R) = – inCircle(S,Q,R,P)

Invalid diagonal

Valid diagonal

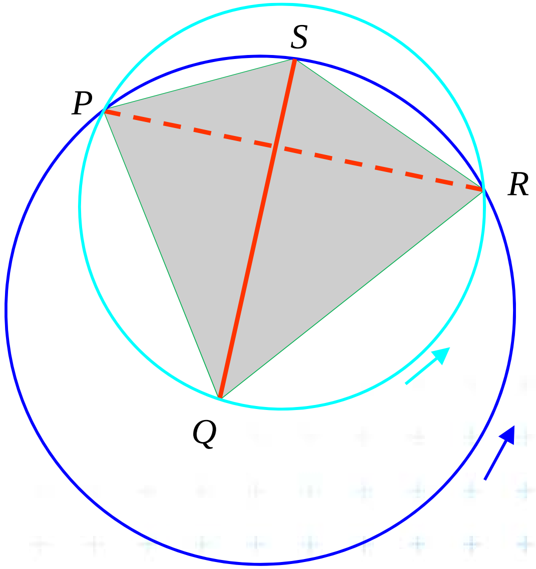


Delaunay triangulation and inCircle test

- DT splits each quadrangle by one of its two diagonals
- For a valid diagonal, the fourth point is not inCircle
  => the fourth point is right from the oriented circumcircle (outside)
  => inCircle(….) < 0 for CCW orientation
- inCircle(P,Q,R,S) = inCircle(P,R,S,Q) = – inCircle(P,Q,S,R) = – inCircle(S,Q,R,P)

Invalid diagonal

Valid diagonal



Delaunay triangulation and inCircle test

- DT splits each quadrangle by one of its two diagonals
- For a valid diagonal, the fourth point is not inCircle
  => the fourth point is right from the oriented circumcircle (outside)
  => inCircle(….) < 0 for CCW orientation
- inCircle(P,Q,R,S) = inCircle(P,R,S,Q) = – inCircle(P,Q,S,R) = – inCircle(S,Q,R,P)

inCircle(…) > 0

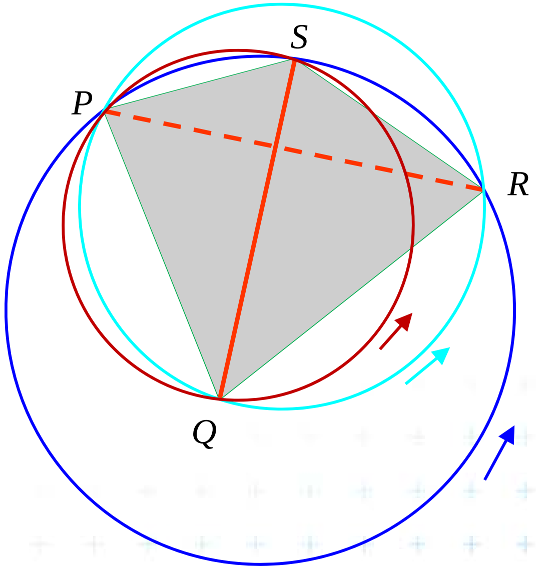
P - - - - - - - - - R

Invalid diagonal

inCircle(…) < 0

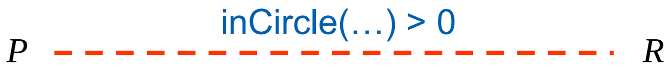
S ——— Q

Valid diagonal



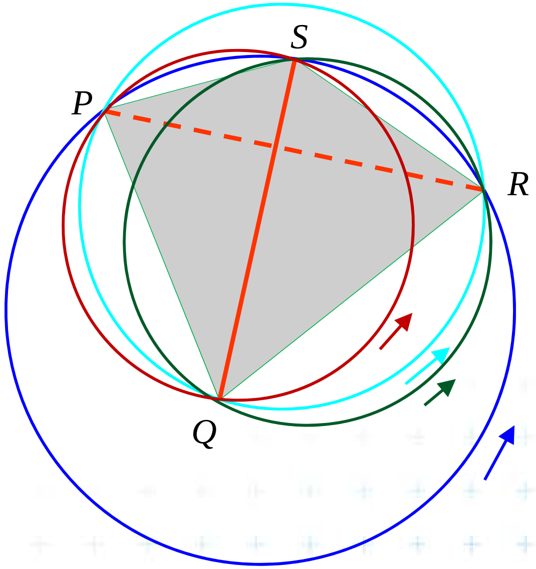
Delaunay triangulation and inCircle test

- DT splits each quadrangle by one of its two diagonals
- For a valid diagonal, the fourth point is not inCircle
  => the fourth point is right from the oriented circumcircle (outside)
  => inCircle(….) < 0 for CCW orientation
- inCircle(P,Q,R,S) = inCircle(P,R,S,Q) = – inCircle(P,Q,S,R) = – inCircle(S,Q,R,P)



Invalid diagonal

Valid diagonal

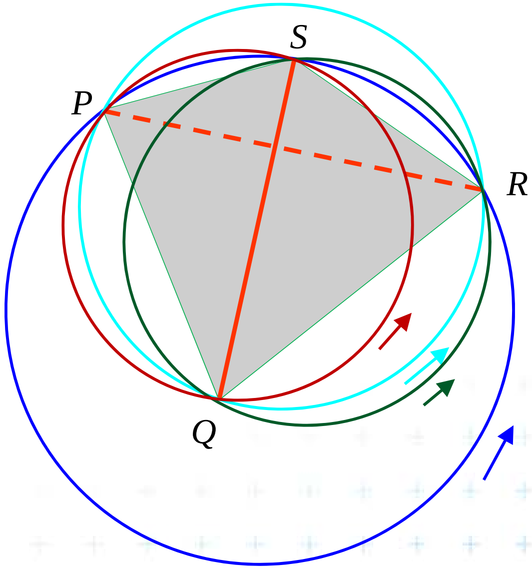


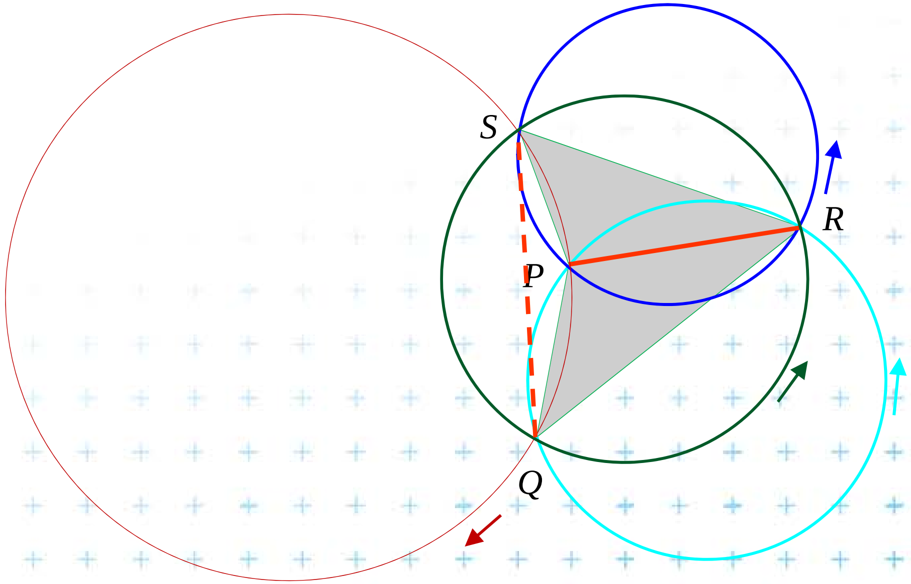
Delaunay triangulation and inCircle test

- DT splits each quadrangle by one of its two diagonals
- For a valid diagonal, the fourth point is not inCircle
  - => the fourth point is right from the oriented circumcircle (outside)
  - => inCircle(….) < 0 for CCW orientation
- inCircle(P,Q,R,S) = inCircle(P,R,S,Q) = – inCircle(P,Q,S,R) = – inCircle(S,Q,R,P)

Invalid diagonal

Valid diagonal

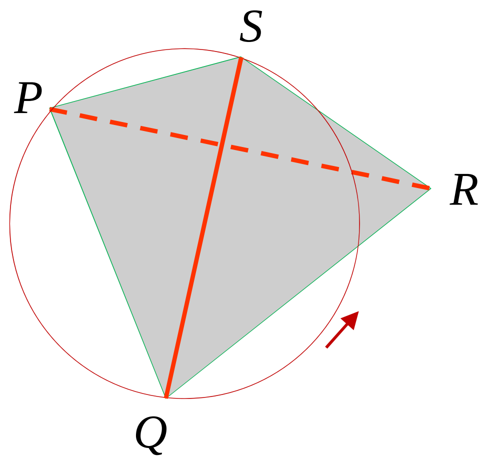




inCircle test detail

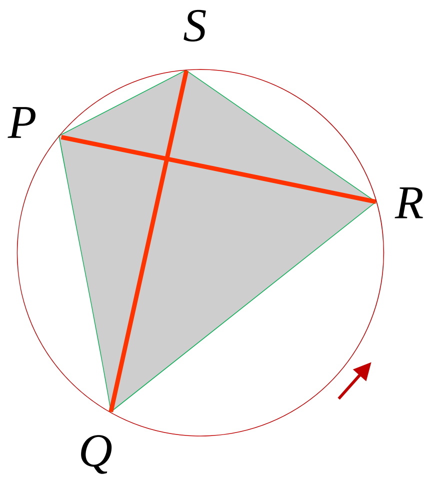
Point *P* moves right toward point *R*

We test position of *R* in relation to oriented circle (*P*,*Q*,*S*)



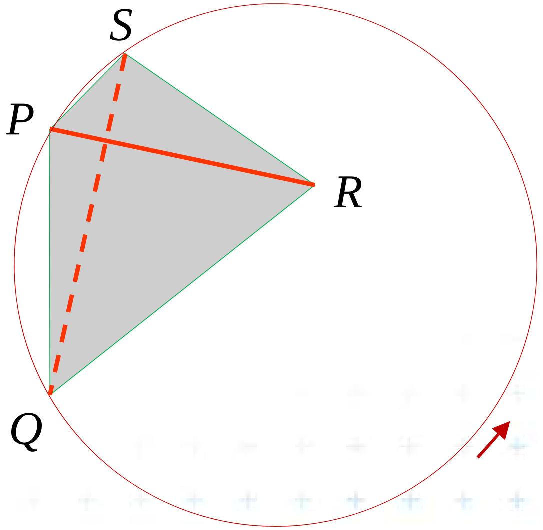
inCircle(P,Q,S,R) < 0

R is right (out)



inCircle(P,Q,S,R) = 0

R is on the circle



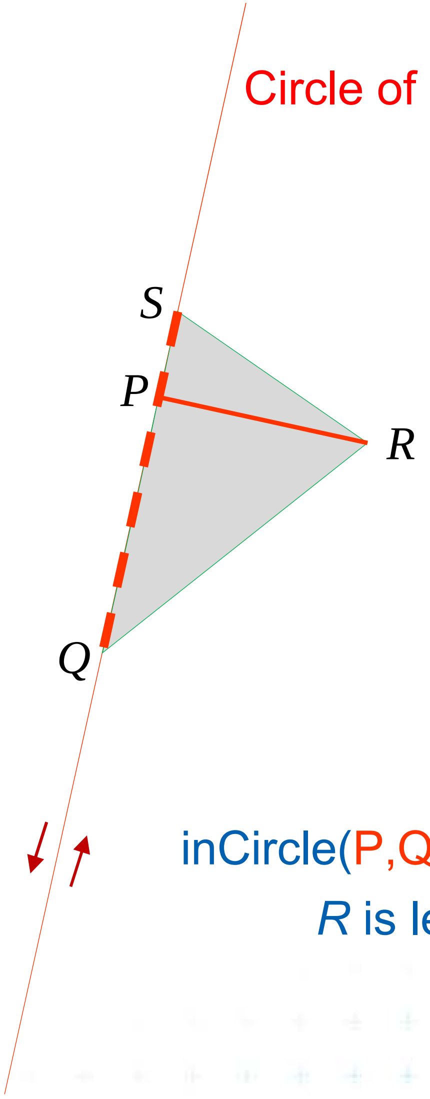
inCircle(P,Q,S,R) > 0

R is left (in)

Invalid diagonal - - - - - - - -

Valid diagonal ———

inCircle test detail

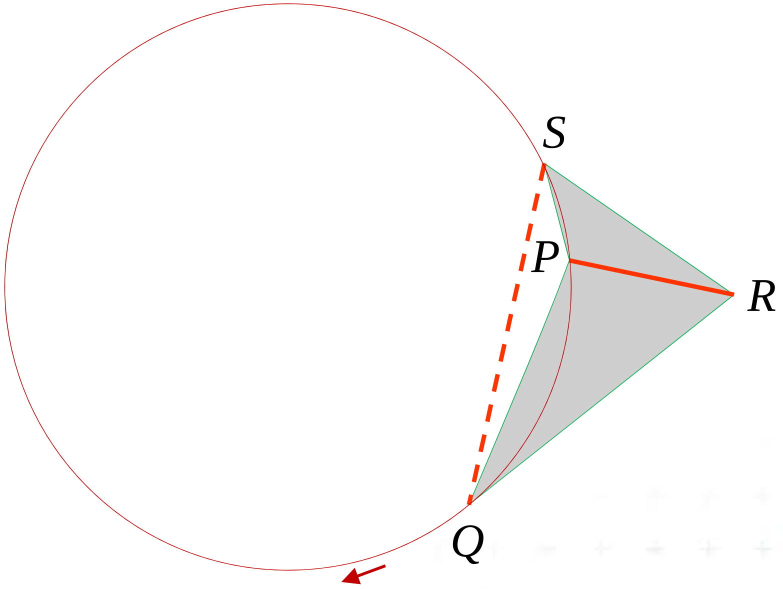


Circle of infinite diameter

inCircle(P,Q,S,R) > 0

R is left

Invalid diagonal



The circle flipped its orientation

inCircle(P,Q,S,R) > 0

R is left

Valid diagonal

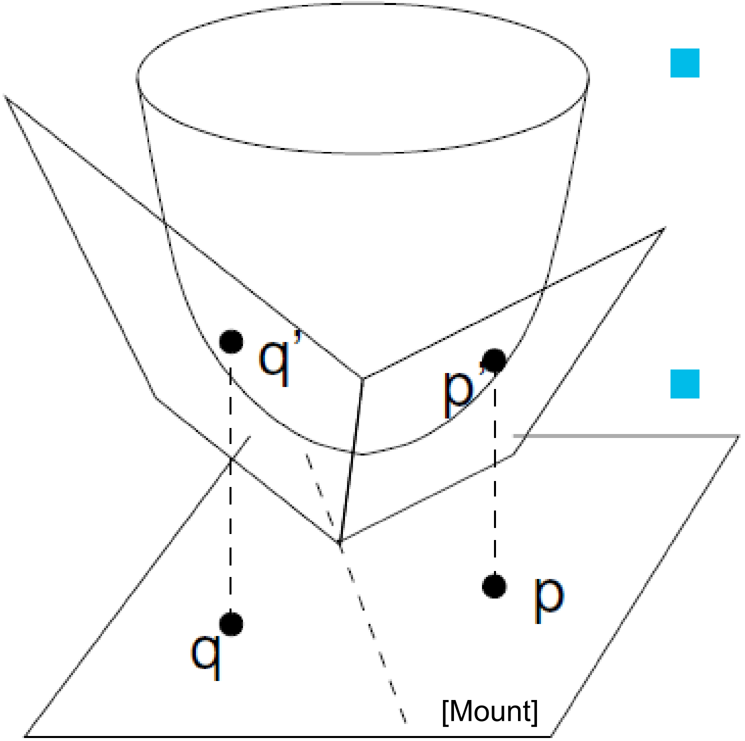
An the Voronoi diagram?

- VD and DT are dual structures
- Points and lines in the plane
  are dual to
  points and planes in 3D space
- VD of points in the plane
  can be transformed to
  intersection of halfspaces in 3D space

Voronoi diagram as upper envelope in R^{d+1}

- For each point $p = (a, b)$ a tangent plane to the paraboloid is $z = 2ax + 2by - (a^2 + b^2)$
- $H^+(p)$ is the set of points above this plane

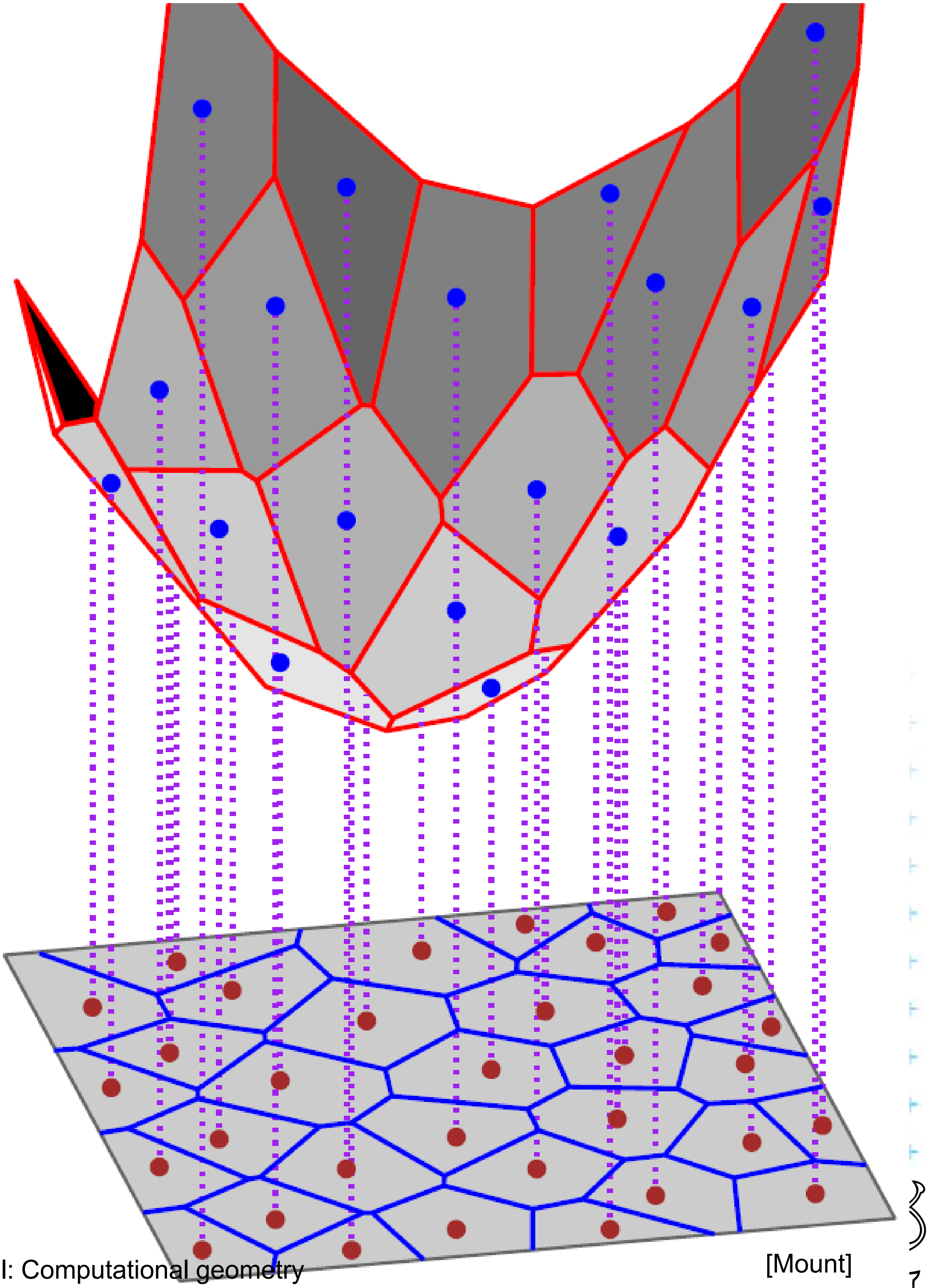
$$H^+(p) = \{(x, y, z) \mid z \geq 2ax + 2by - (a^2 + b^2)\}$$



- VD of points in the plane can be computed as intersection of halfspaces $H^+(p_i)$ in 3D
- This intersection of halfspaces
  = unbounded convex polyhedron
  = upper envelope of halfspaces $H^+(p_i)$

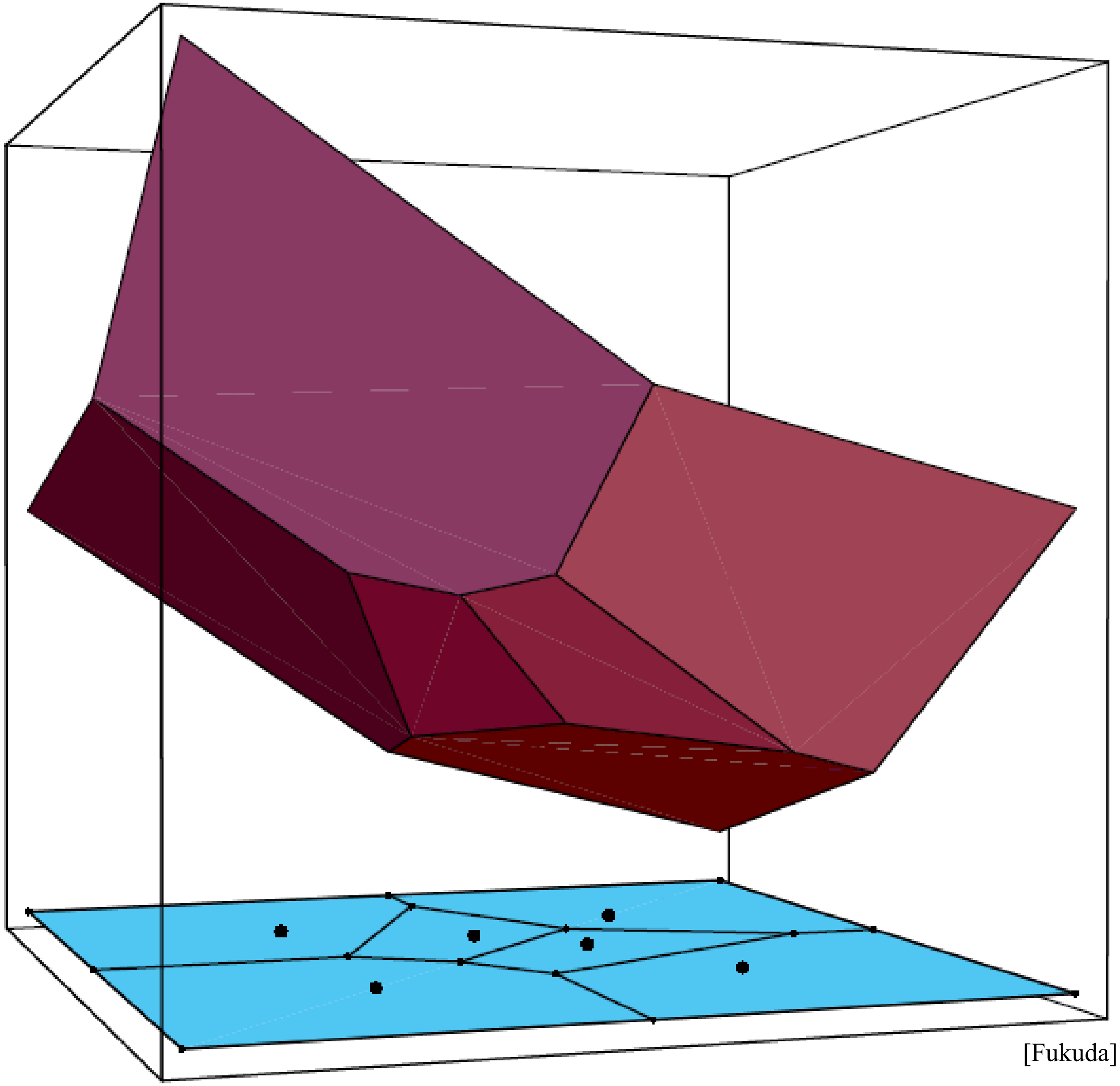
Projection to 2D

- Upper envelope of tangent hyperplanes (through sites projected upwards to the cone)
- Projected to 2D gives Voronoi diagram



[Mount]

Voronoi diagram as upper envelope in 3D



[Fukuda]

Derivation of projected Voronoi edge

- 2 points: $p = (a, b)$ and $q = (c, d)$ in the plane

  $z = 2ax + 2by - (a^2 + b^2)$

  $z = 2cx + 2dy - (c^2 + d^2)$

  Tangent planes to paraboloid

- Intersect the planes, project onto xy (eliminate *z*)

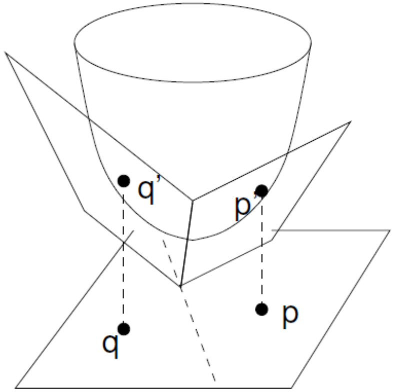
  $x(2a - 2c) + y(2b - 2d) = (a^2 - c^2) + (b^2 - d^2)$

- This line passes through midpoint between p and q

  $\frac{a+c}{2}(2a - 2c) + \frac{b+d}{2}(2b - 2d) = (a^2 - c^2) + (b^2 - d^2)$

- It is perpendicular bisector with slope

  $-(a - c)/(b - d)$



[Mount]

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, Chapters 3 and 9, http://www.cs.uu.nl/geobook/

[Mount] David Mount, - CMSC 754: Computational Geometry, Lecture Notes for Spring 2007, University of Maryland, Lectures 7,22, 13,14, and 30. http://www.cs.umd.edu/class/spring2007/cmsc754/lectures.shtml

[Rourke] Joseph O´Rourke: .: Computational Geometry in C, Cambridge University Press, 1993, ISBN 0-521- 44592-2 http://maven.smith.edu/~orourke/books/compgeom.html

[Fukuda] Komei Fukuda: Frequently Asked Questions in Polyhedral Computation. Version June 18, 2004 http://www.ifor.math.ethz.ch/~fukuda/polyfaq/polyfaq.html