DCGI

DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

COMPUTATIONAL GEOMETRY INTRODUCTION

PETR FELKEL

FEL CTU PRAGUE

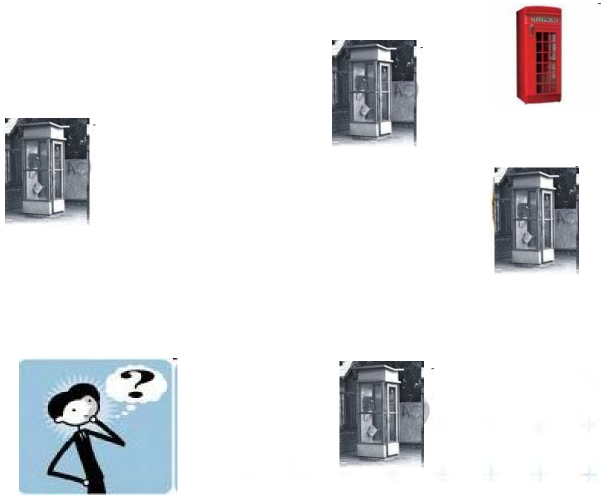
Version from 8.10.2018

Computational Geometry

1. What is Computational Geometry (CG)?
2. Why to study CG and how?
3. Typical application domains
4. Typical tasks
5. Complexity of algorithms
6. Programming techniques (paradigms) of CG
7. Robustness Issues
8. CGAL – CG algorithm library intro
9. References and resources
10. Course summary

1. What is Computational Geometry?

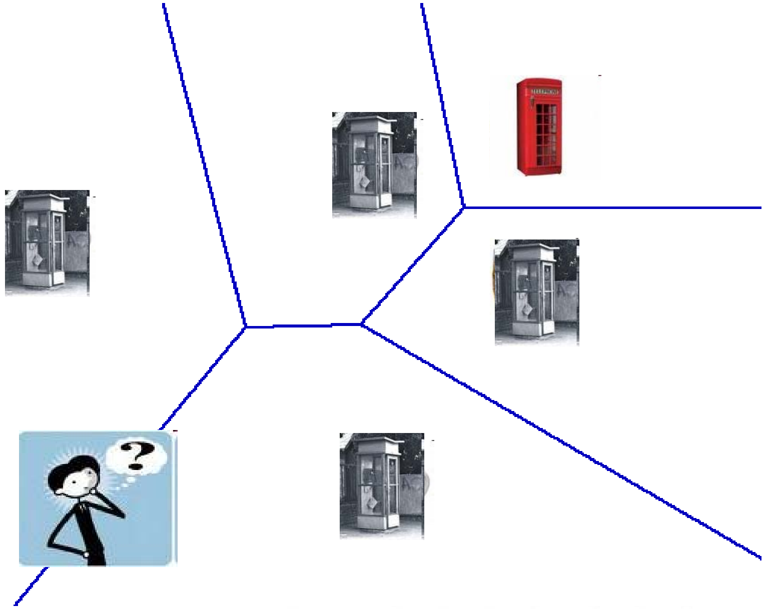
- CG Solves geometric problems that require clever geometric algorithms
- Ex 1: Where is the nearest phone, metro, pub,…?



[Berg]

1. What is Computational Geometry?

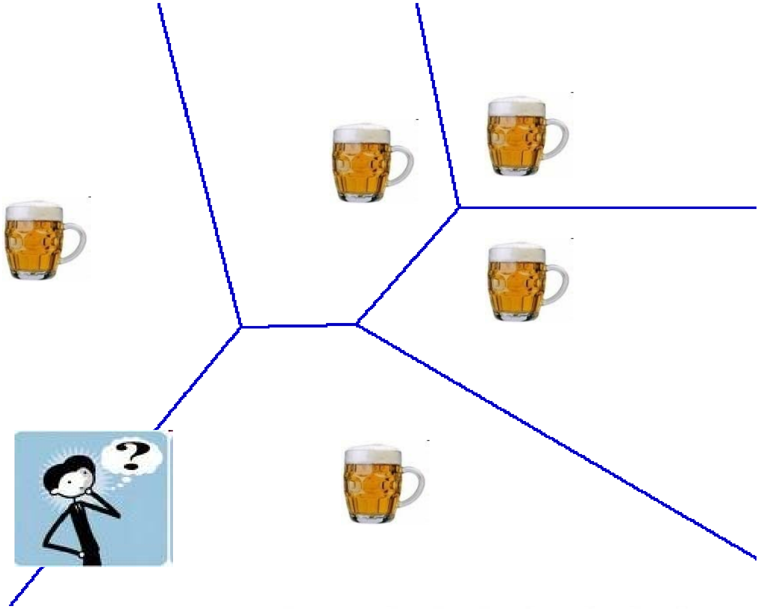
- CG Solves geometric problems that require clever geometric algorithms
- Ex 1: Where is the nearest phone, metro, pub,…?



[Berg]

1. What is Computational Geometry?

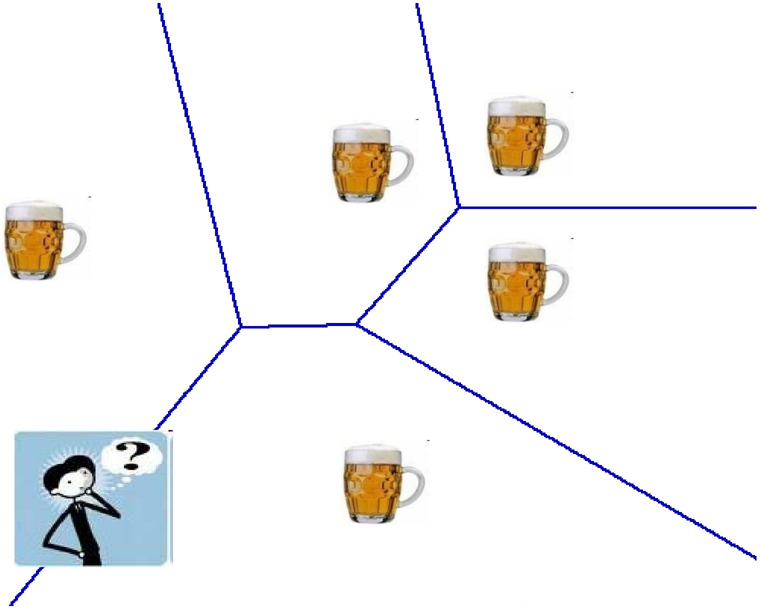
- CG Solves geometric problems that require clever geometric algorithms
- Ex 1: Where is the nearest phone, metro, pub,…?



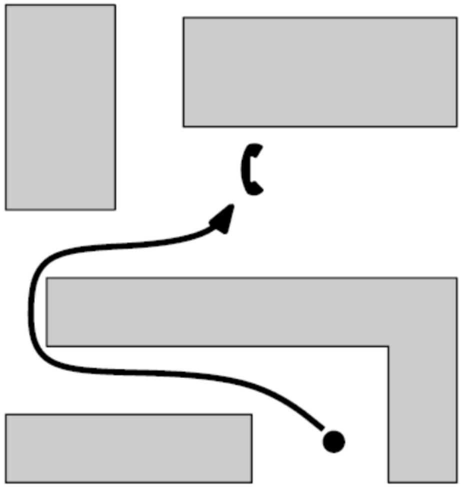
[Berg]

1. What is Computational Geometry?

- CG Solves geometric problems that require clever geometric algorithms
- Ex 1: Where is the nearest phone, metro, pub,…?



Ex 2: How to get there?

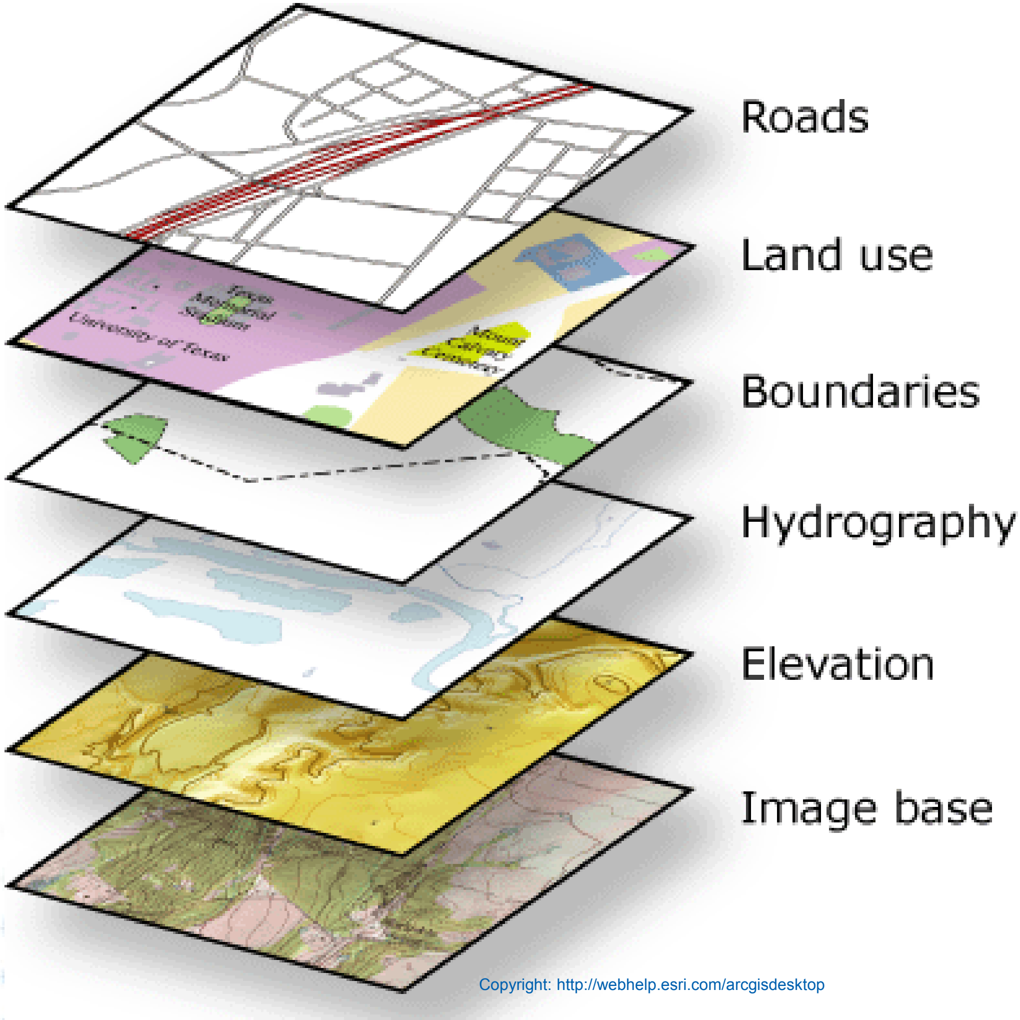


[Berg]

1.1 What is Computational Geometry? (…)

- Ex 3: Map overlay

Myslíkova
Odborů
Na Zbořenci
Na Zderaze
Karlovo nám.
Resslova



Copyright: http://webhelp.esri.com/arcgisdesktop

1.2 What is Computational Geometry? (…)

- Good solutions need both:
  - Understanding of the
    geometric properties of the problem
  - Proper applications of
    algorithmic techniques (paradigms) and data structures

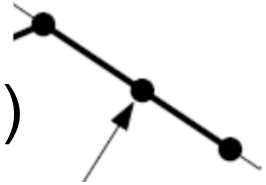
1.3 What is Computational Geometry? (…)

- Computational geometry
  = systematic study of algorithms and data structures for geometric objects (points, lines, line segments, n-gons,…) with focus on exact algorithms that are asymptotically fast

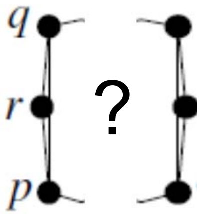
  - “Born” in 1975 (Shamos), boom of papers in 90s
    (first papers sooner: 1850 Dirichlet, 1908 Voronoi,…)
  - Many problems can be formulated geometrically
    (e.g., range queries in databases)

1.4 What is Computational Geometry? (…)

- Problems:
  - Degenerate cases (points on line, with same *x*,…)
    - Ignore them first, include later



  - Robustness - correct algorithm but not robust
    - Limited numerical precision of real arithmetic
    - Inconsistent *eps* tests (a=b, b=c, but a ≠ c)



- Nowadays:
  - focus on practical implementations, not just on asymptotically fastest algorithms
  - nearly correct result is better than nonsense or crash

2. Why to study computational geometry?

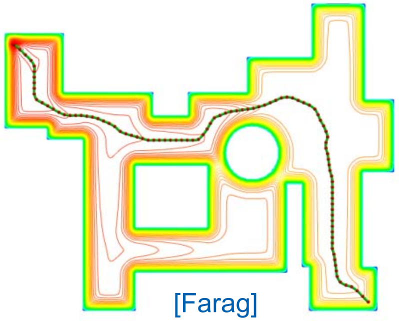
- Graphics- and Vision- Engineer should know it („Data structures and algorithms in nth-Dimension“) - DSA, PRP
- Set of ready to use tools
- You will know new approaches to choose from

2.1 How to teach computational geometry?

- Typical "mathematician" method:
  - definition-theorem-proof
- Our "practical" approach:
  - practical algorithms and their complexity
  - practical programing using a geometric library
- Is it OK for you?

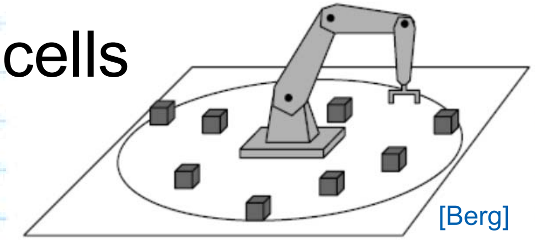
3. Typical application domains

- Computer graphics
  - Collisions of objects
  - Mouse localization
  - Selection of objects in region
  - Visibility in 3D (hidden surface removal)
  - Computation of shadows



[Farag]

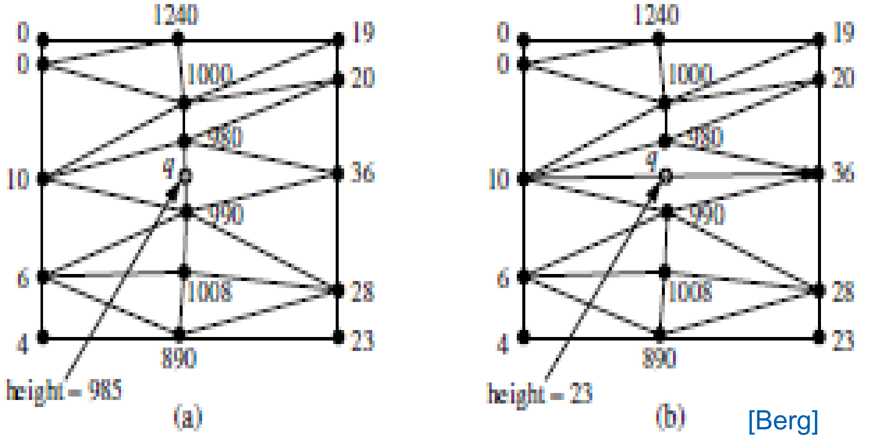
- Robotics
  - Motion planning (find path - environment with obstacles)
  - Task planning (motion + planning order of subtasks)
  - Design of robots and working cells



[Berg]

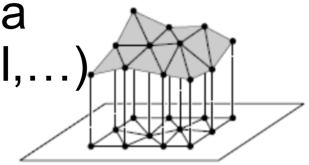
3.1 Typical application domains (…)

- GIS
  - How to store huge data and search them quickly
  - Interpolation of heights
  - Overlap of different data



[Berg]

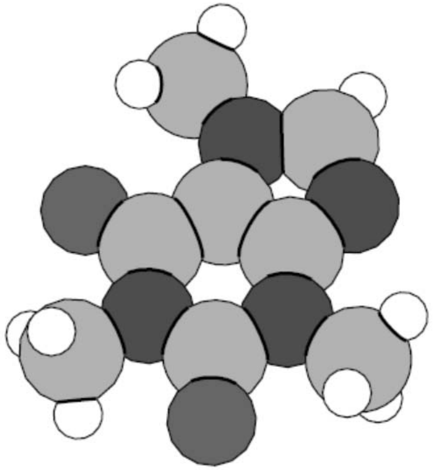
    - Extract information about regions or relations between data (pipes under the construction site, plants x average rainfall,…)
    - Detect bridges on crossings of roads and rivers…



- CAD/CAM
  - Intersections and unions of objects
  - Visualization and tests without need to build a prototype
  - Manufacturability

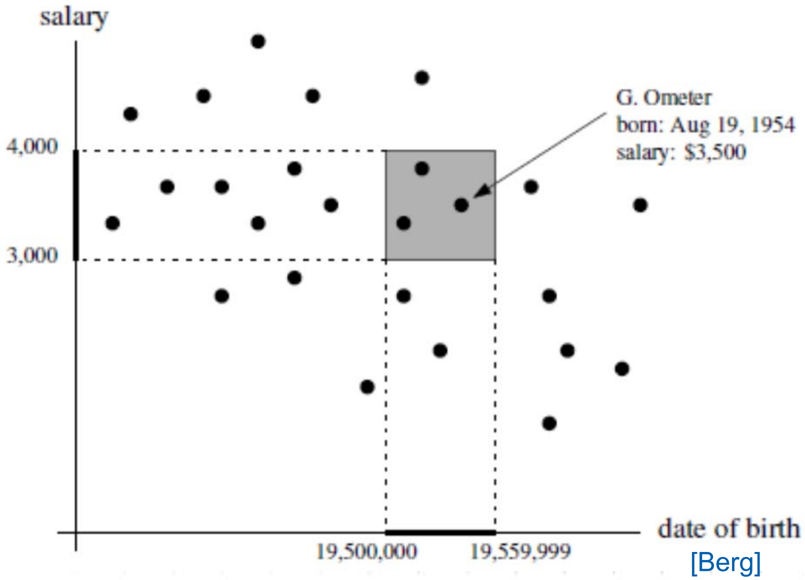
3.2 Typical application domains (…)

- Other domains
  - Molecular modeling
  - DB search
  - IC design

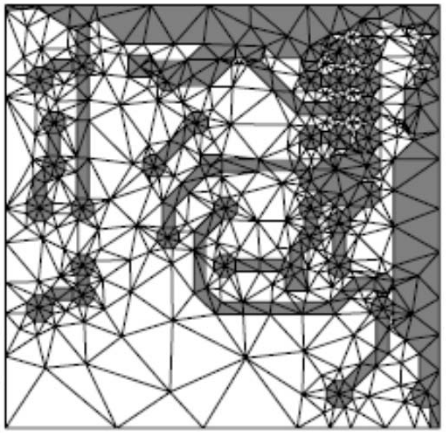


[Berg]

caffeine



[Berg]

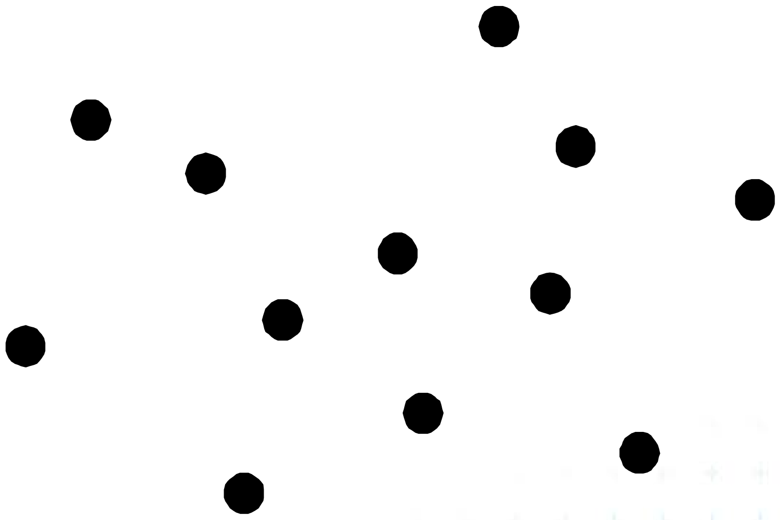


[Berg]

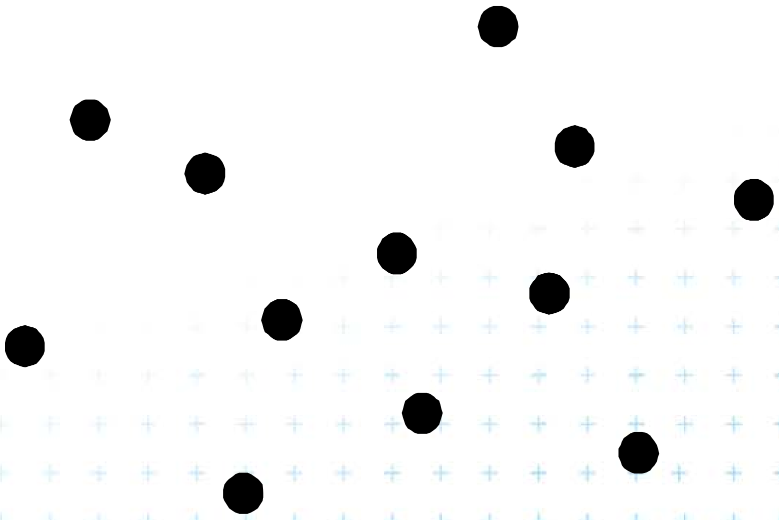
4. Typical tasks in CG

- Geometric searching - fast location of :

The nearest neighbor



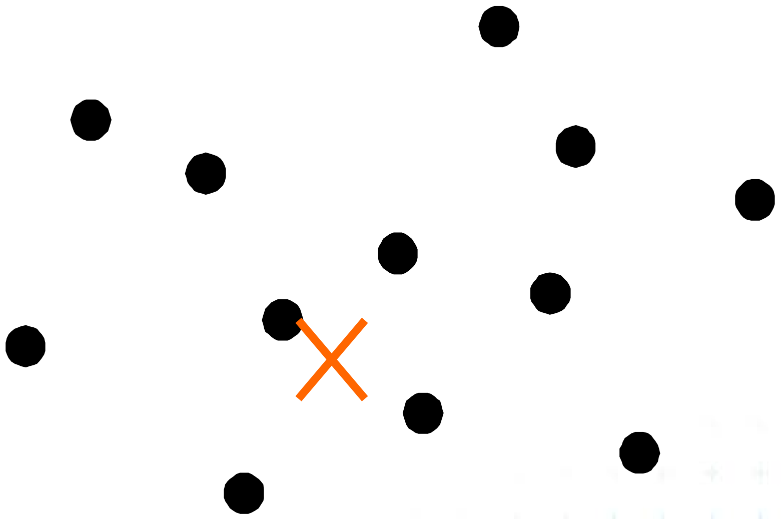
Points in given range (*range query*)



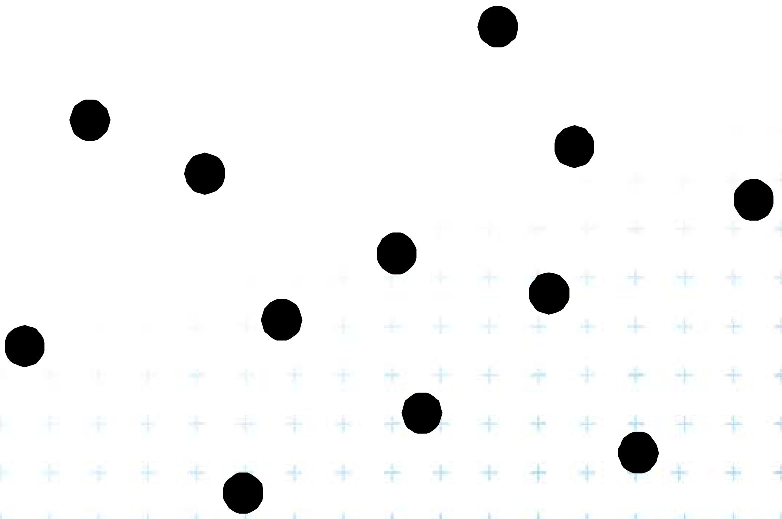
4. Typical tasks in CG

- Geometric searching - fast location of :

The nearest neighbor



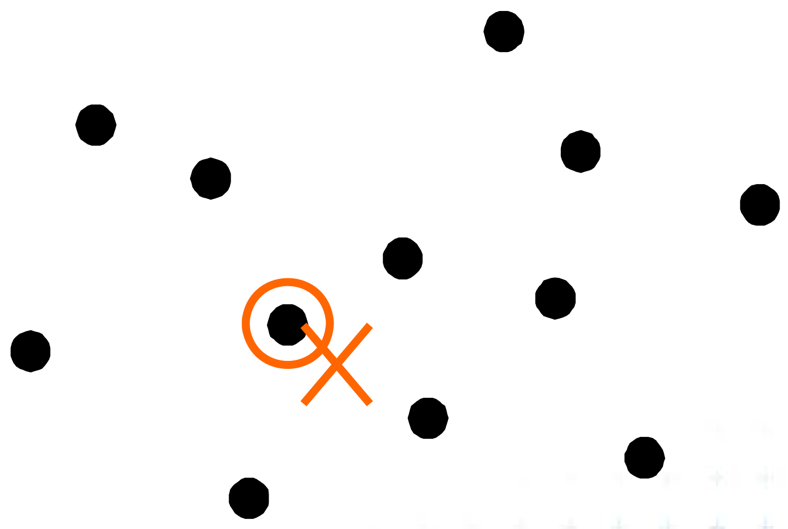
Points in given range (*range query*)



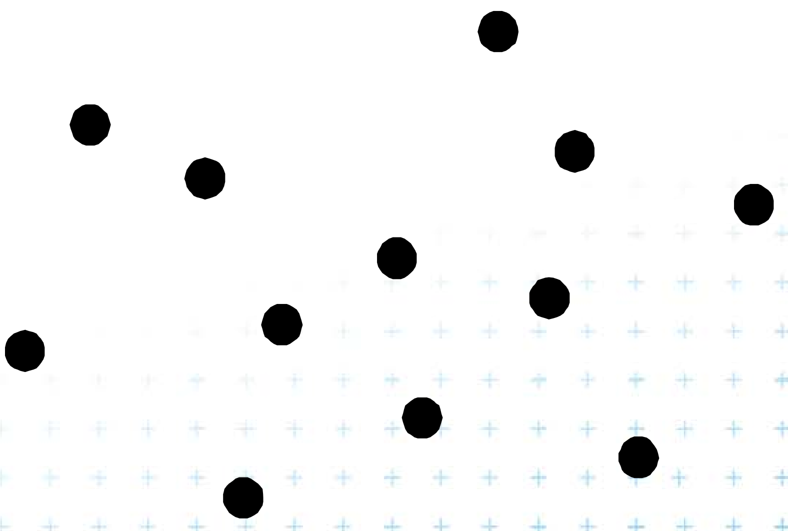
4. Typical tasks in CG

- Geometric searching - fast location of :

The nearest neighbor



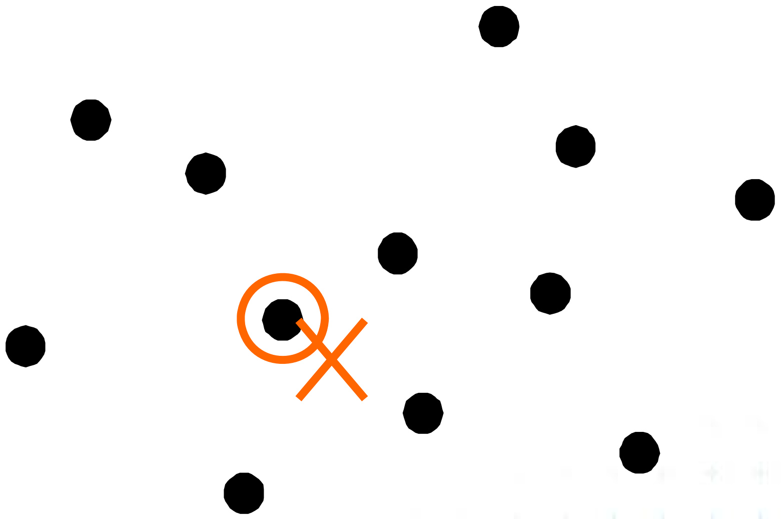
Points in given range
(*range query*)



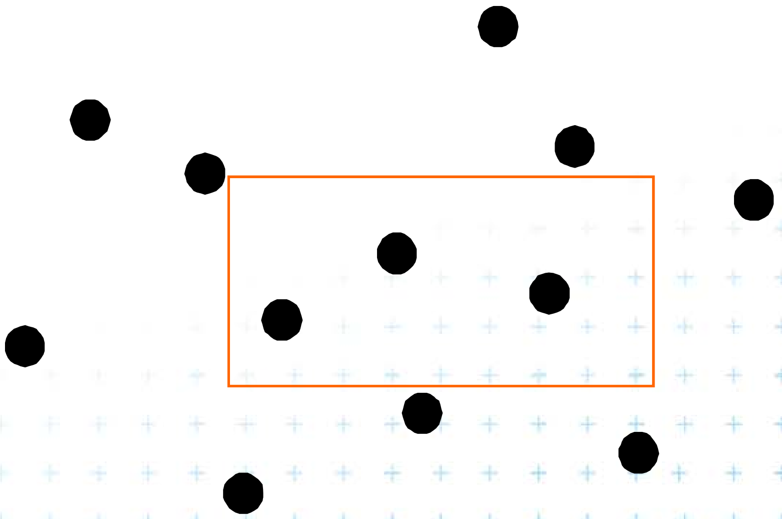
4. Typical tasks in CG

- Geometric searching - fast location of :

The nearest neighbor



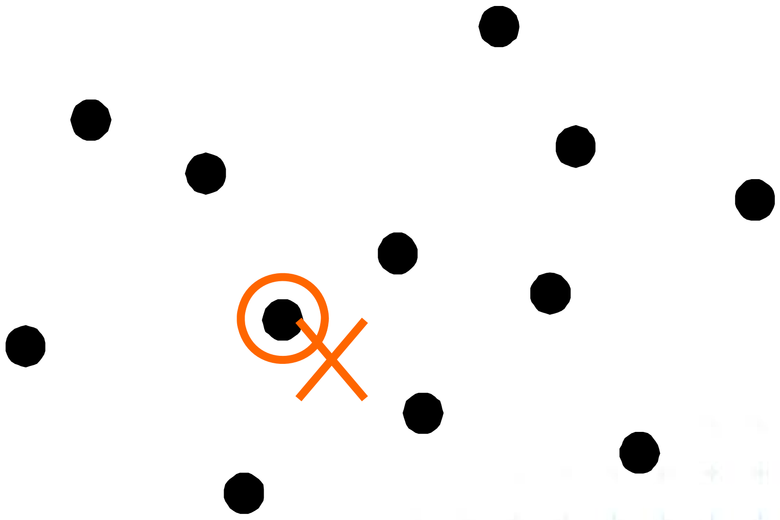
Points in given range
(*range query*)



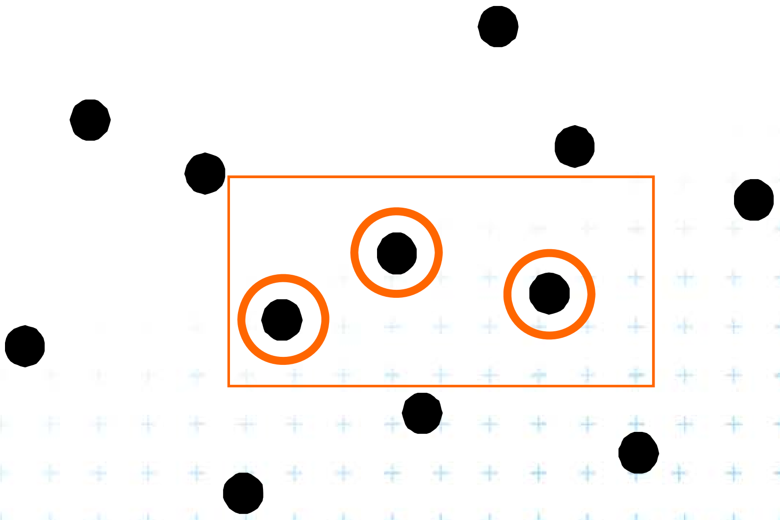
4. Typical tasks in CG

- Geometric searching - fast location of :

The nearest neighbor

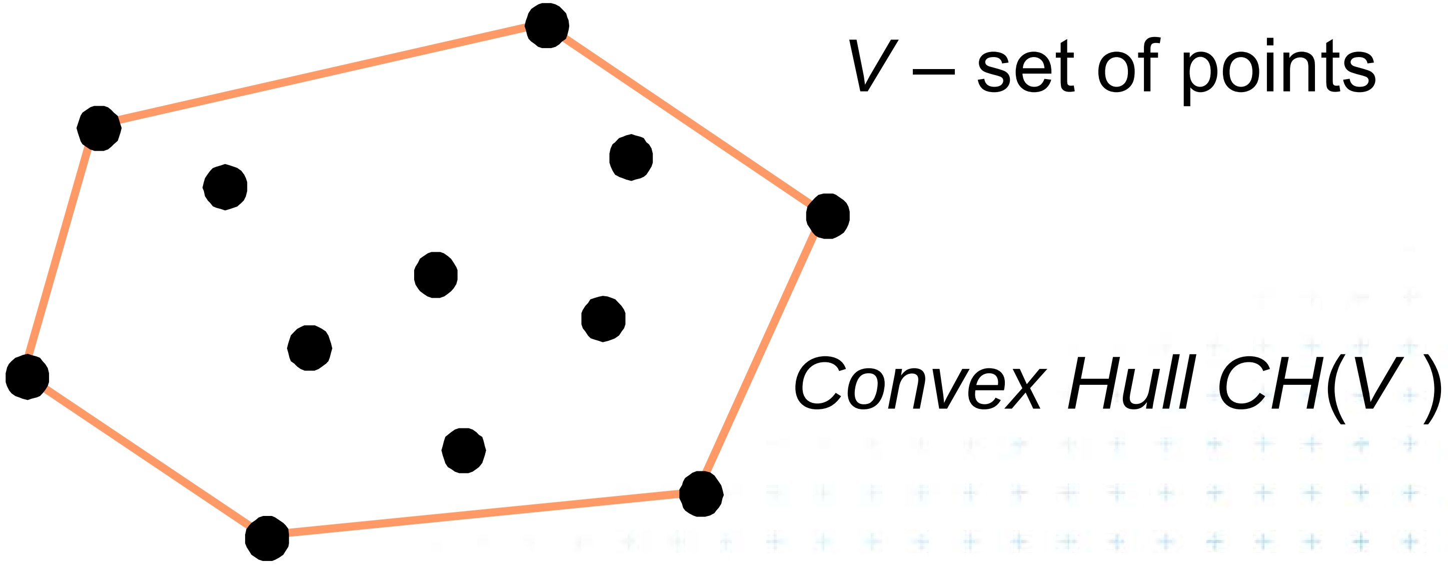


Points in given range (*range query*)



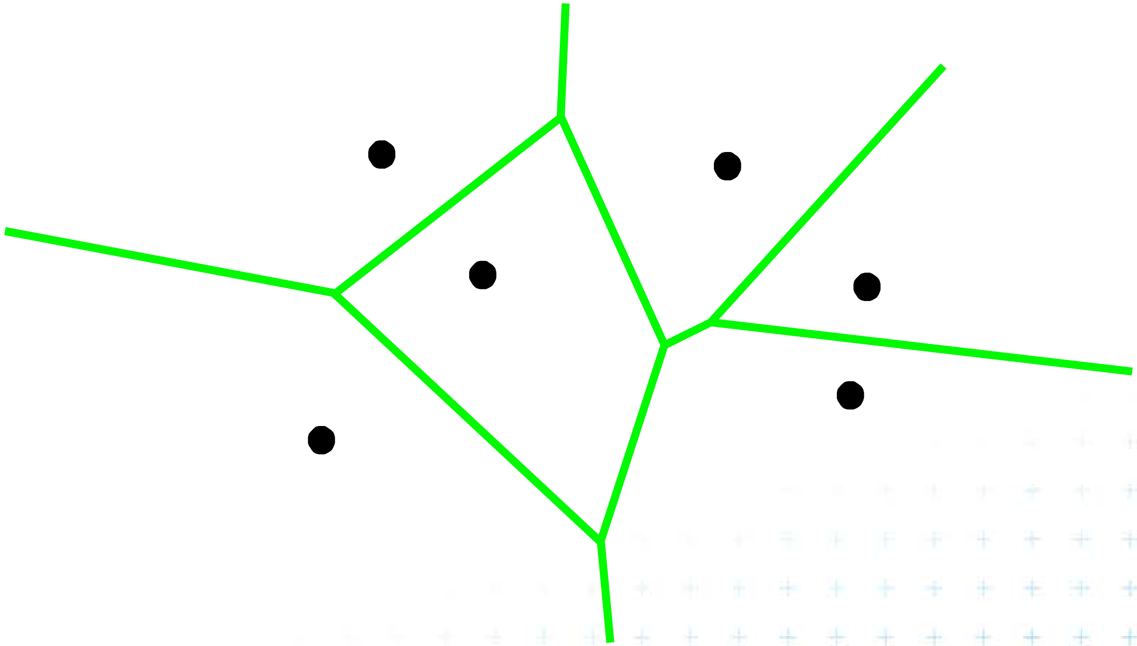
4.1 Typical tasks in CG

- Convex hull
  = smallest enclosing convex polygon in E^2 or n-gon in E^3 containing all the points



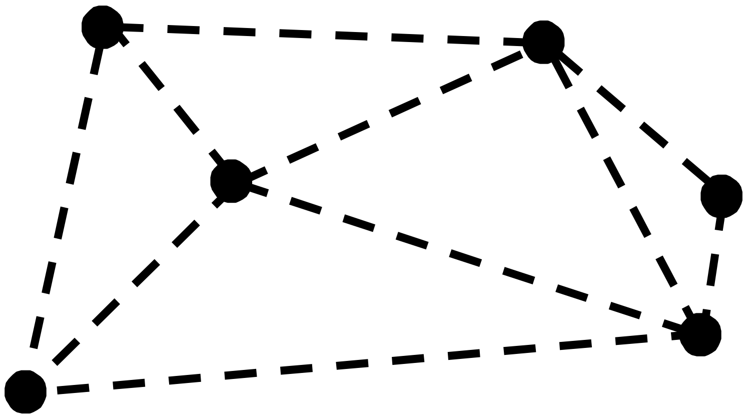
4.2 Typical tasks in CG

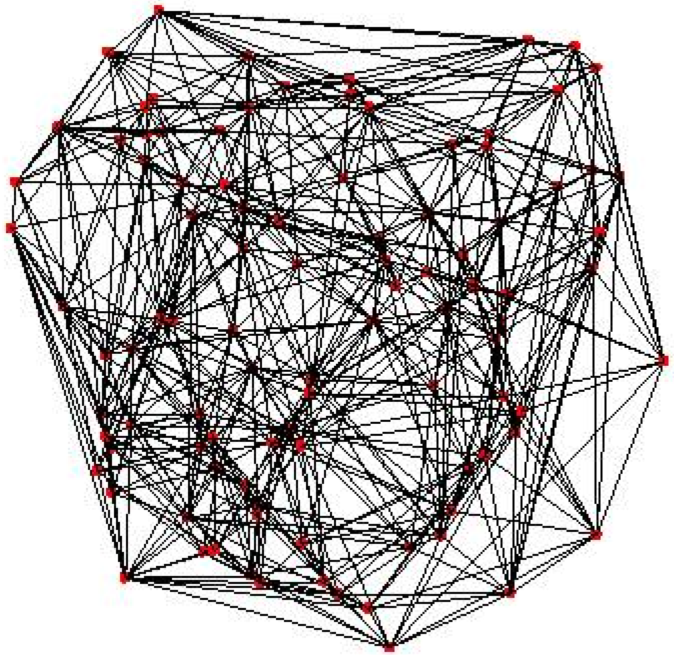
- Voronoi diagrams
  - Space (plane) partitioning into regions whose points are nearest to the given primitive (most usually a point)



4.3 Typical tasks in CG

- Planar triangulations and space tetrahedronization of given point set

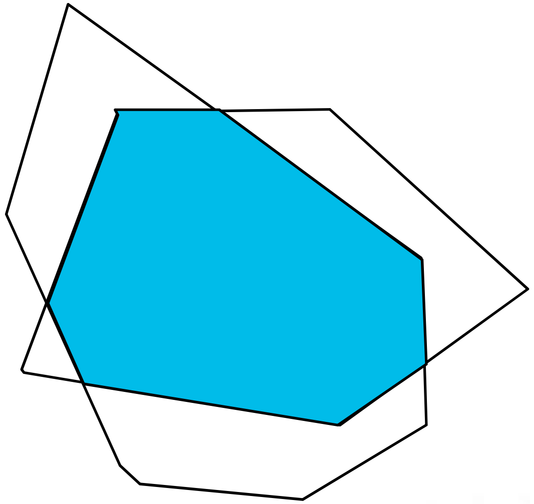


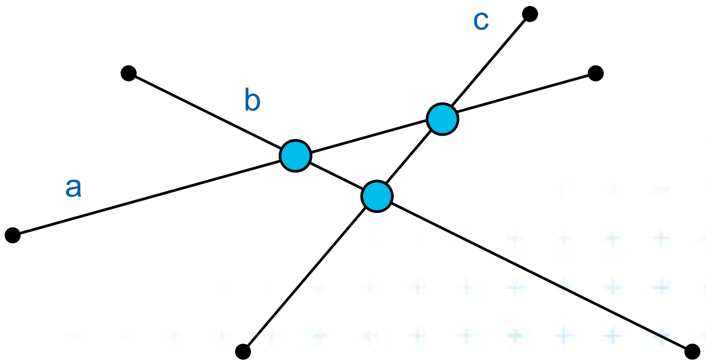


[Maur]

4.4 Typical tasks in CG

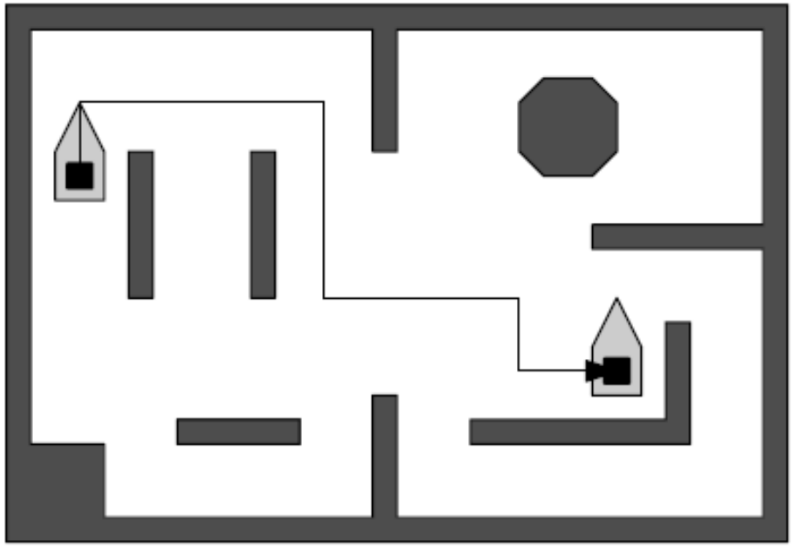
- Intersection of objects
  - Detection of common parts of objects
  - Usually linear (line segments, polygons, n-gons,…)





4.5 Typical tasks in CG

- Motion planning
  - Search for the shortest path between two points in the environment with obstacles



[Berg]

5. Complexity of algorithms and data struc.

- We need a measure for comparison of algorithms
  - Independent on computer HW and prog. language
  - Dependent on the problem size *n*
  - Describing the behavior of the algorithm for different data
- Running time, preprocessing time, memory size
  - Asymptotical analysis – **O(g(*n*))**, $\Omega(g(n))$, $\Theta(g(n))$
  - Measurement on real data
- Differentiate:
  - complexity of the algorithm (particular sort) and
  - complexity of the problem (sorting)
    – given by number of edges, vertices, faces,… = problem size
    – equal to the complexity of the best algorithm

5.1 Complexity of algorithms

- Worst case behavior
  - Running time for the “worst” data
- Expected behavior (average)
  - expectation of the running time for problems of particular size and probability distribution of input data
  - Valid only if the probability distribution is the same as expected during the analysis
  - Typically much smaller than the worst case behavior
  - Ex.: Quick sort $O(n^2)$ worst and $O(n \log n)$ expected

6. Programming techniques (paradigms) of CG

3 phases of a geometric algorithm development

1. Ignore all degeneracies and design an algorithm

2. Adjust the algorithm to be correct for degenerate cases
   - Degenerate input exists
   - Integrate special cases in general case
   - It is better than lot of case-switches (typical for beginners)
   - e.g.:
     lexicographic order for points on vertical lines
     or Symbolic perturbation schemes

3. Implement alg. 2 (use sw library)

6.1 Sorting

- A preprocessing step
- Simplifies the following processing steps
- Sort according to:
  - coordinates x, y,…, or lexicographically to [y,x],
  - angles around point
- *O(n* log*n)* time and *O(n)* space

6.2 Divide and Conquer (divide et impera)

- Split the problem until it is solvable, merge results

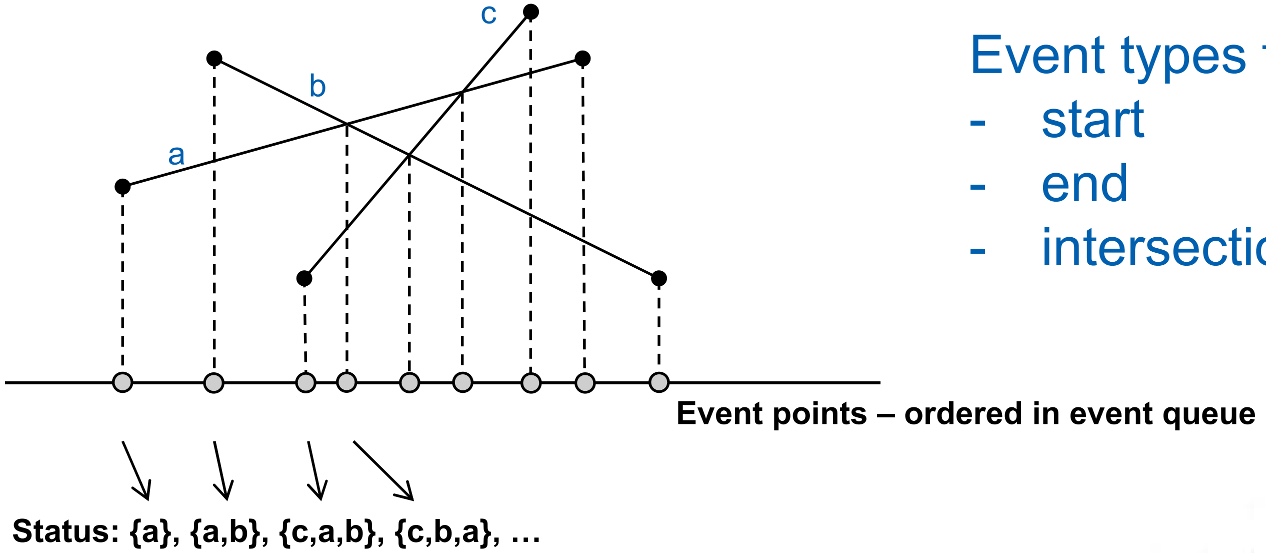
```
DivideAndConquer(S)
If known solution then return it
else
    Split input $S$ to $k$ distinct subsets $S_i$
    Foreach $i$ call DivideAndConquer($S_i$)
    Merge the results and return the solution
```

- Prerequisite
  - The input data set must be separable
  - Solutions of subsets are independent
  - The result can be obtained by merging of sub-results

6.3 Sweep algorithm

- Split the space by a hyperplane (2D: sweep line)
  - "Left" subspace – solution known
  - "Right" subspace – solution unknown
- Stop in event points and update the status
- Data structures:
  - **Event points** – points, where to stop the sweep line and update the status, sorted
  - **Status** – state of the algorithm in the current position of the sweep line
- Prerequisite:
  - Left subspace does not influence the right subspace

6.3b Sweep-line algorithm

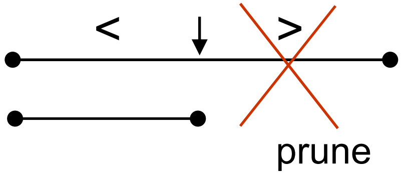

c
b
a
Event points – ordered in event queue
Status: {a}, {a,b}, {c,a,b}, {c,b,a}, …

Event types for segments:

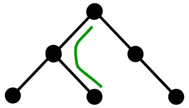
- start
- end
- intersection

6.4 Prune and search

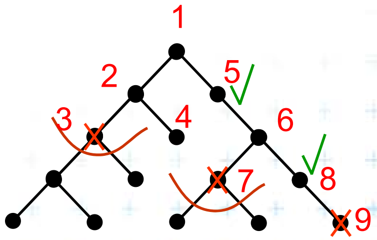
- Eliminate parts of the state space, where the solution clearly does not exist
  - Binary search



  - Search trees

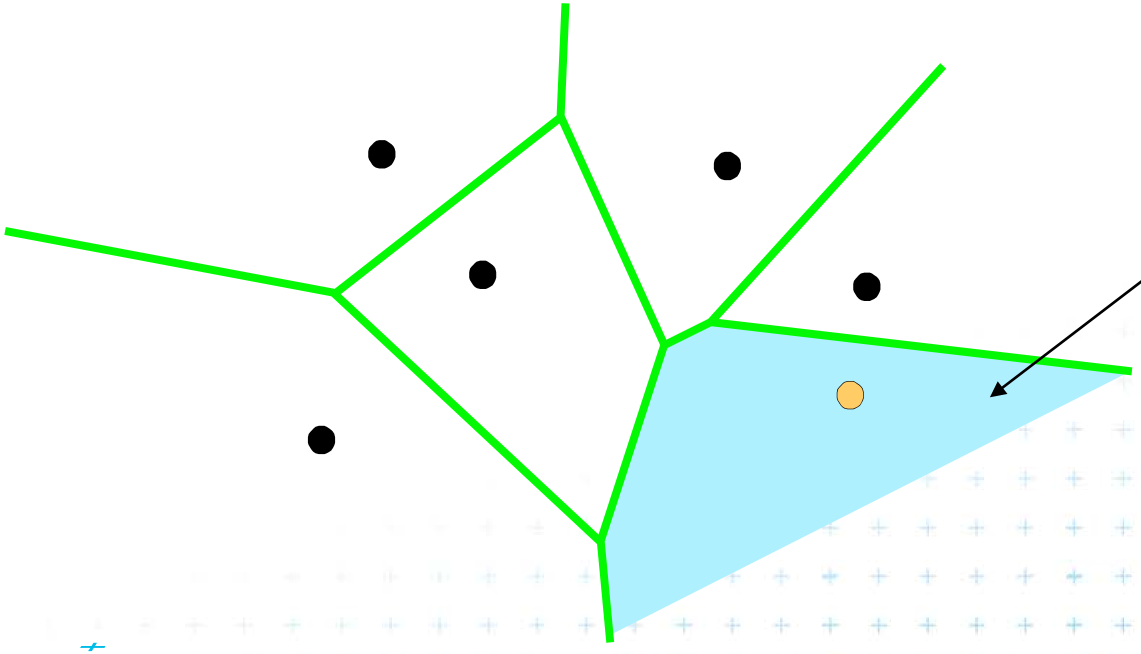


  - Back-tracking (stop if solution worse than current optimum)



6.5 Locus approach

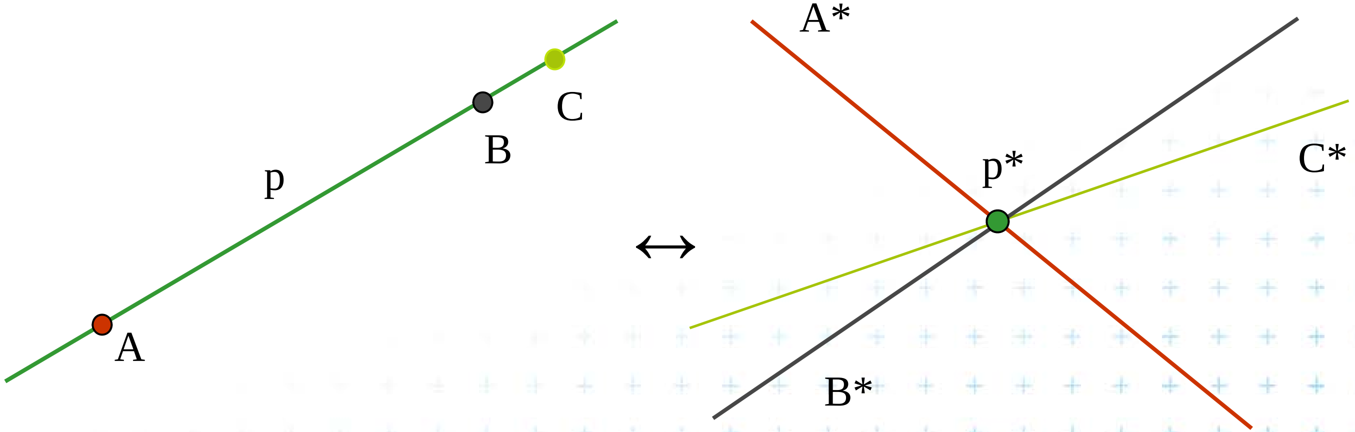
- Subdivide the search space into regions of constant answer
- Use point location to determine the region
  - Nearest neighbor search example



Region of the constant answer: All points in this region are nearest to the yellow point

6.6 Dualisation

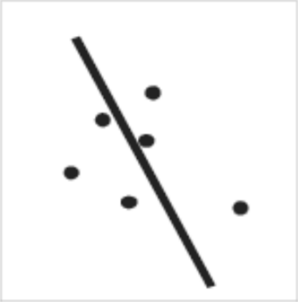
- Use geometry transform to change the problem into another that can be solved more easily
- Points ↔ hyper planes
  - Preservation of incidence ($A \in p \;\Rightarrow\; p^* \in A^*$)
- Ex. 2D: determine if 3 points lie on a common line



6.7 Combinatorial analysis

= The branch of mathematics which studies the number of different ways of arranging things

- Ex. How many subdivisions of a point set can be done by one line?



6.8 New trends in Computational geometry

- From 2D to 3D and more from mid 80s, from linear to curved objects
- Focus on line segments, triangles in E^3 and hyper planes in E^d
- Strong influence of combinatorial geometry
- Randomized algorithms
- Space effective algorithms (in place, in situ, data stream algs.)
- Robust algorithms and handling of singularities
- Practical implementation in libraries (CGAL, …)
- Approximate algorithms

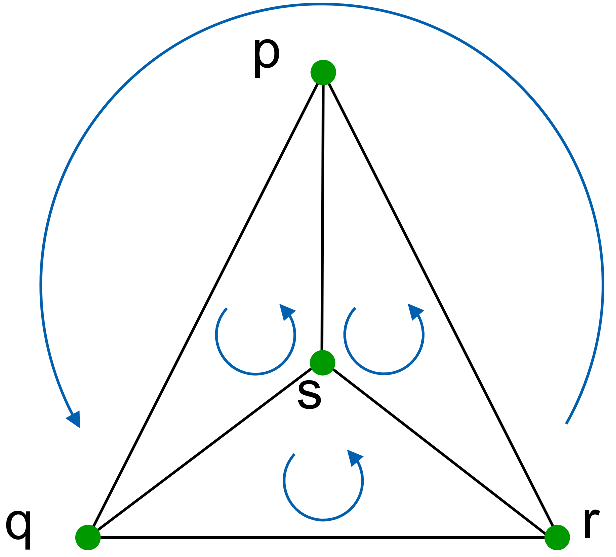
7. Robustness issues

- Geometry in theory is exact
- Geometry with floating-point arithmetic is not exact
  - Limited numerical precision of real arithmetic
  - Numbers are rounded to nearest possible representation
  - Inconsistent *epsilon* tests (a=b, b=c, but a≠c)
- Naïve use of floating point arithmetic causes geometric algorithm to
  - Produce slightly or completely wrong output
  - Crash after invariant violation
  - Infinite loop

[siggraph2008-CGAL-course]

Geometry in theory is exact

- ccw(s,q,r) & ccw(p,s,r) & ccw(p,q,s) => ccw(p,q,r)

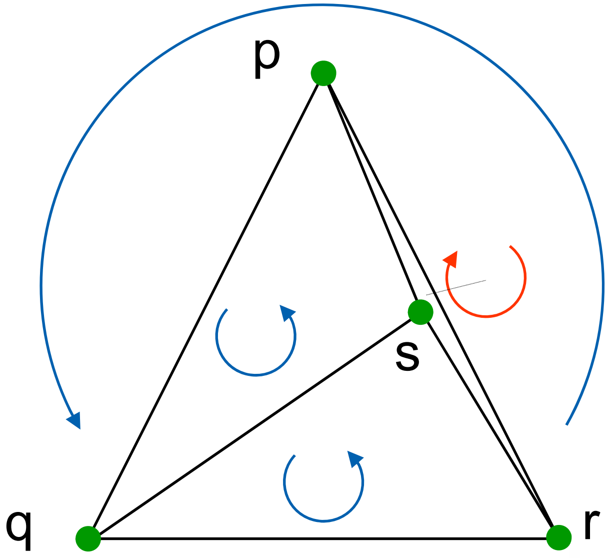


- Correctness proofs of algorithms rely on such theorems

[siggraph2008-CGAL-course]

Geometry with float. arithmetic is not exact

- ccw(s,q,r) & !ccw(p,s,r) & ccw(p,q,s) ~~=>~~ ccw(p,q,r)



wrong result of the orientation predicate

- Correctness proofs of algorithms rely on such theorems => such algorithms fail

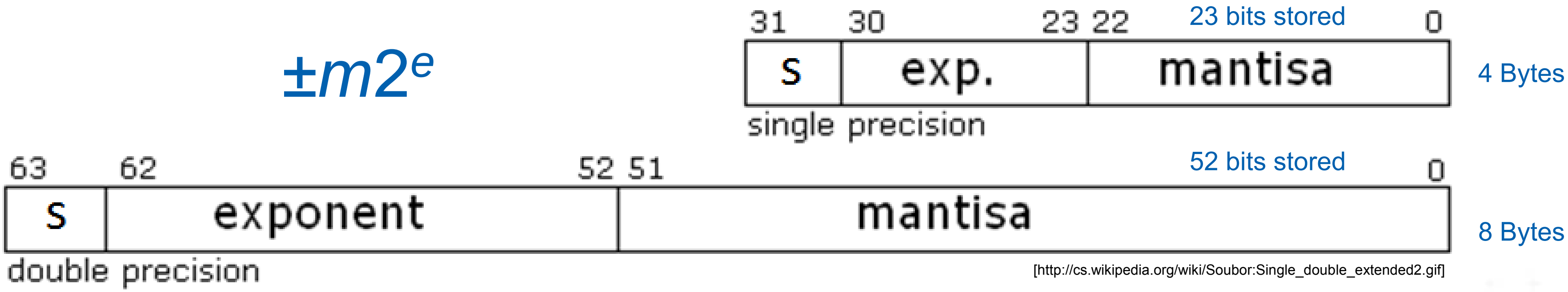
[siggraph2008-CGAL-course]

Floating-point arithmetic is not exact

a) Limited numerical precision of real numbers

- Numbers represented as normalized

$$\pm m2^e$$



[http://cs.wikipedia.org/wiki/Soubor:Single_double_extended2.gif]

- The mantissa *m* is a 24-bit (53-bit) value whose most significant bit (MSB) is always 1 and is, therefore, not stored.
- Stored numbers are rounded to 24/53 bits mantissa – lower bits are lost

Floating-point special values

+0	0	00000000	00000000000000000000000
- 0	1	00000000	00000000000000000000000
+Infinity	0	11111111	00000000000000000000000
-Infinity	1	11111111	00000000000000000000000
NaN	?	11111111	00000000000000000000001

Floating-point arithmetic is not exact

b) Smaller numbers are shifted right during additions and subtractions to align the digits of the same order

Example for float:

- 12 – *p* for *p* ~ 0.5

Invisible leading bit – not stored

2^3 1 Normalized mantisa 23 bit

  - $12_{10} = 1100_2 \quad = 0\boxed{10000010}\boxed{10000000000000000000000}_2$

2^{-1} 1

  - $p = 0.5_{10} \quad = 0\boxed{01111110}\boxed{00000000000000000000000}_2$
  - $p = 0.5000008_{10} = 0\boxed{01111110}\boxed{00000000000000000001101}_2$
  - Mantissa of *p* is shifted 4 bits right to align with 12 (to have the same exponent 2^3)

    $p = 0.5\cancel{000008}_{10} = 0\boxed{10000010}\boxed{00010000000000000000000}_2\cancel{1101}$

    –> four least significant bits (LSB) are lost
  - The result is 11.5 instead of 11.4999992

Floating-point arithmetic is not exact

b) Smaller numbers are shifted right during additions and subtractions to align the digits of the same order

Example for float:

- 12 – *p* for *p* ~ 0.5 (such as 0.5+2^(-23))
  - Mantissa of *p* is shifted 4 bits right to align with 12
    –> four least significant bits (LSB) are lost
- 24 – *p* for *p* ~ 0.5
  - Mantissa of *p* is shifted 5 bits right to align with 24 -> 5 LSB are lost

Try it on [http://www.h-schmidt.net/FloatConverter/IEEE754.html or http://babbage.cs.qc.cuny.edu/IEEE-754/index.xhtml]

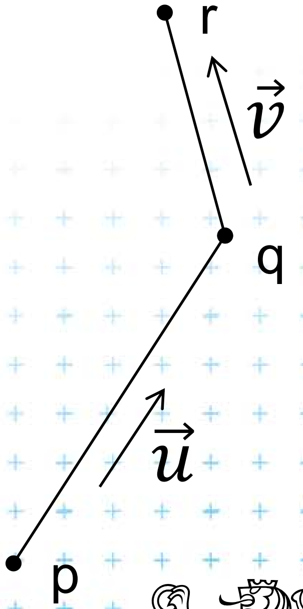
Orientation predicate - definition

$$\text{orientation}(p,q,r) = \text{sign}\left(\det\begin{bmatrix}1 & p_x & p_y\\ 1 & q_x & q_y\\ 1 & r_x & r_y\end{bmatrix}\right) =$$

$$= \text{sign}\left((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)\right),$$

$$\text{where point } p = (p_x, p_y), \ldots$$

$$= \text{third coordinate of} = (\vec{u} \times \vec{v}),$$

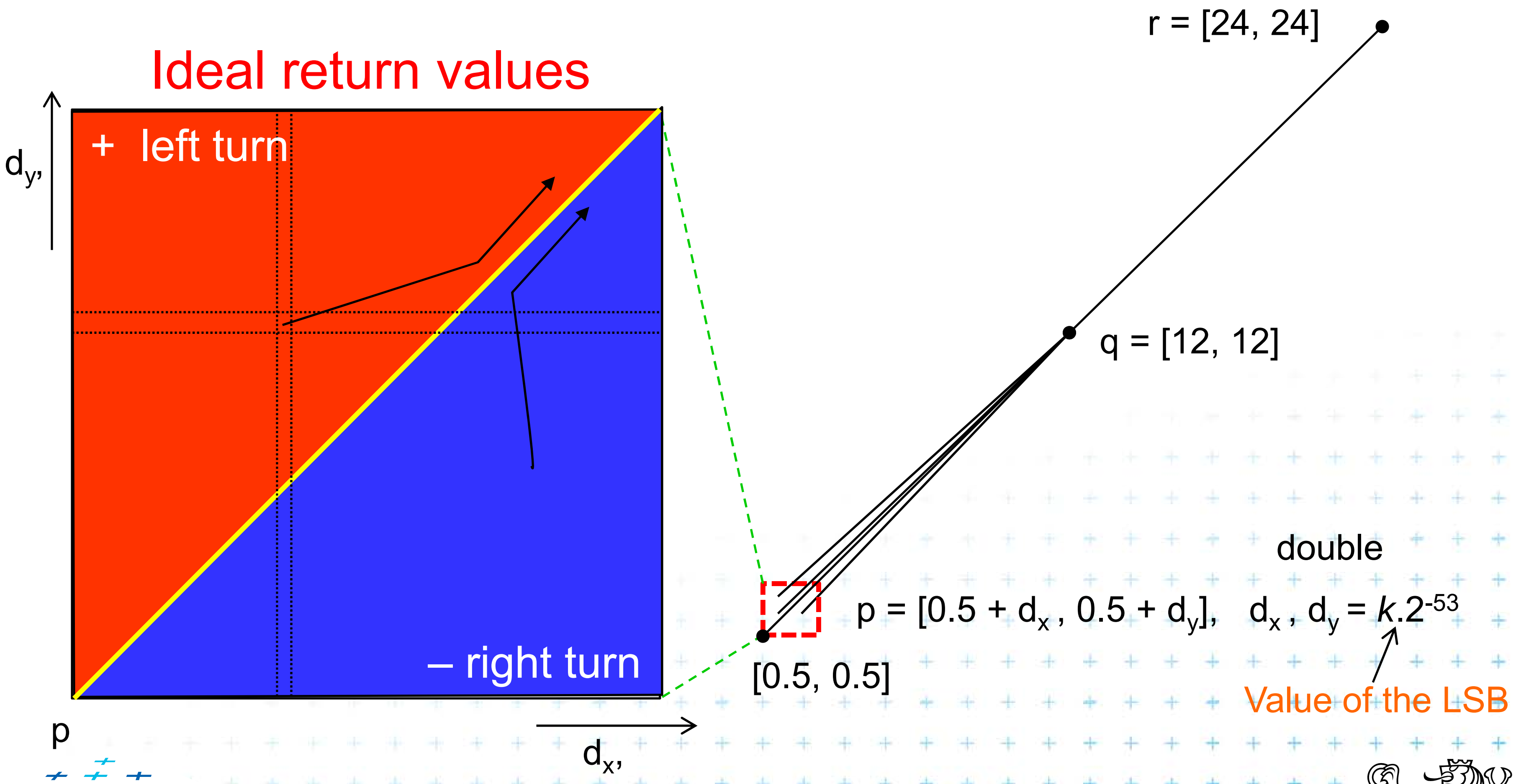


Three points $\quad \text{orientation}(p,q,r) =$

- lie on common line = 0
- form a left turn = +1 (positive)
- form a right turn = –1 (negative)

Experiment with orientation predicate

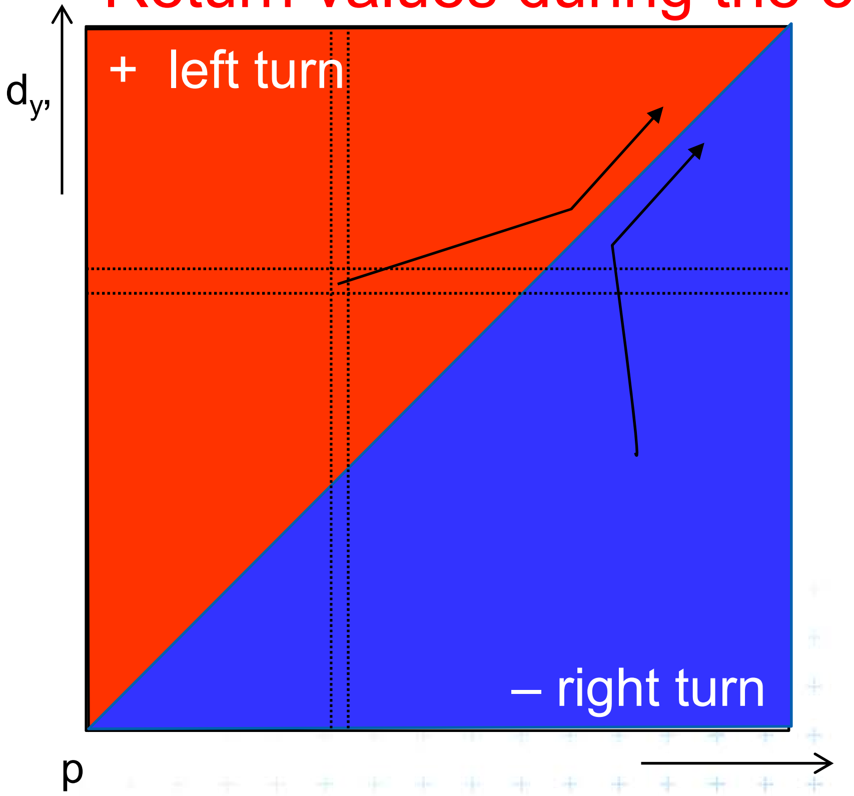
- orientation(p,q,r) = sign($(p_x-r_x)(q_y-r_y)-(p_y-r_y)(q_x-r_x)$)



Real results of orientation predicate

- orientation(p,q,r) = sign($((p_x-r_x)(q_y-r_y)-(p_y-r_y)(q_x-r_x))$)

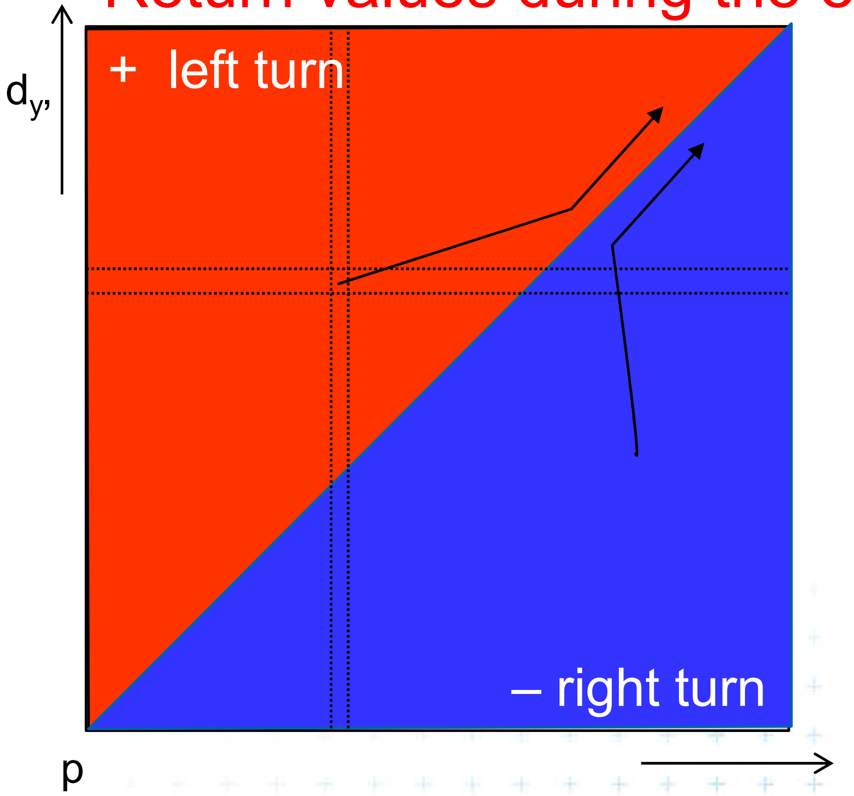
Return values during the experiment for exponent > -52



Real results of orientation predicate

- orientation(p,q,r) = sign$((p_x-r_x)(q_y-r_y)-(p_y-r_y)(q_x-r_x))$

Return values during the experiment for exponent > -52

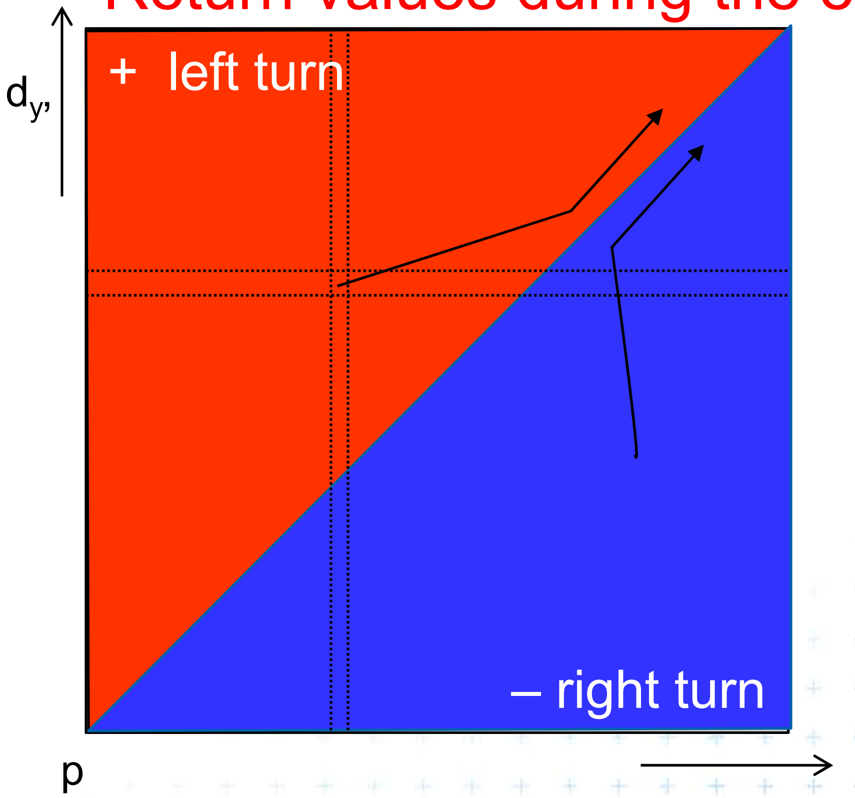


Where is the yellow line?

Real results of orientation predicate

- orientation(p,q,r) = sign($(p_x-r_x)(q_y-r_y)-(p_y-r_y)(q_x-r_x)$)

Return values during the experiment for exponent > -52



Where is the yellow line?

Robust predicate returns slightly non-zero values

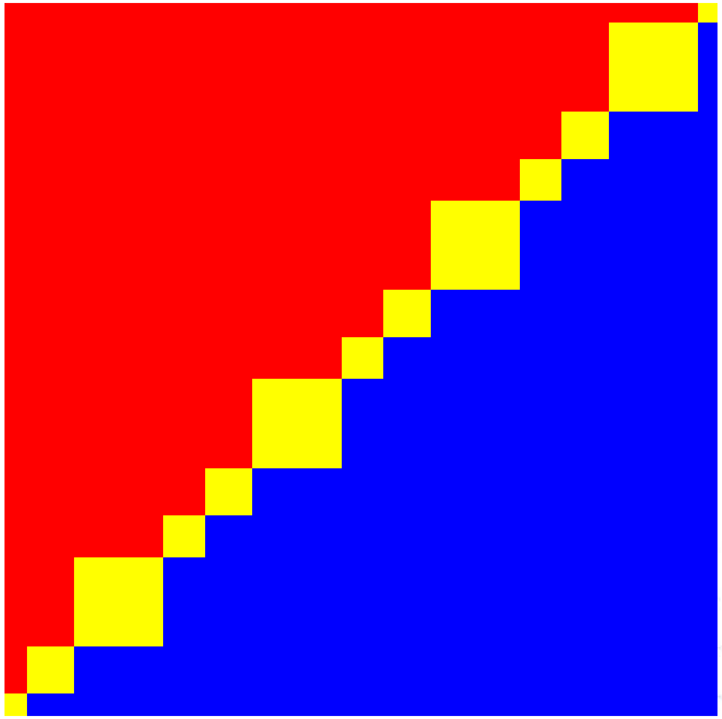
$$\mathrm{orientation}(p,q,r) \neq 0$$

Never lie on common line

Real results of orientation predicate

- orientation(p,q,r) = sign($((p_x-r_x)(q_y-r_y)-(p_y-r_y)(q_x-r_x))$)

Return values during the experiment for exponent -52



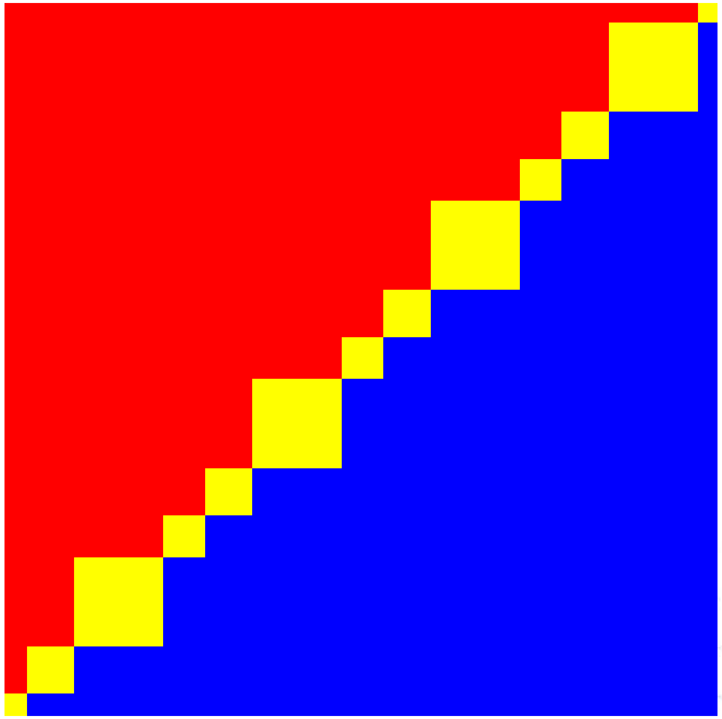
Pivot r $_{24}$

Pivot p $_{0.5}$

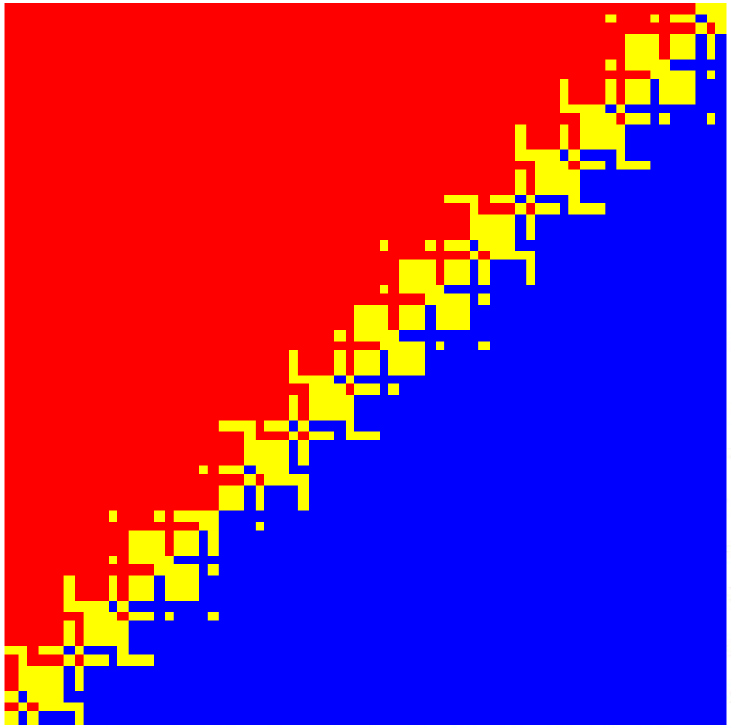
Real results of orientation predicate

- orientation(p,q,r) = sign($(p_x-r_x)(q_y-r_y)-(p_y-r_y)(q_x-r_x)$)

Return values during the experiment for exponent -52



Pivot r $_{24}$

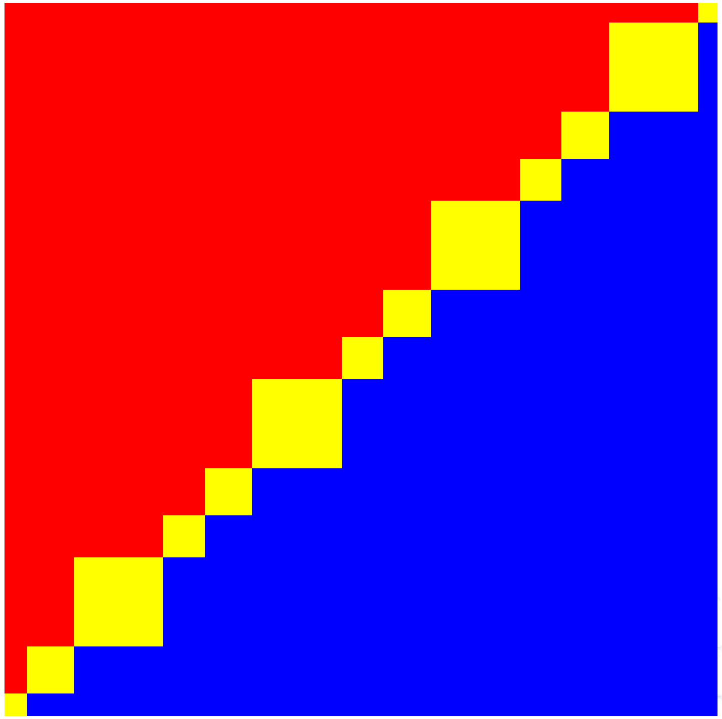


Pivot p $_{0.5}$

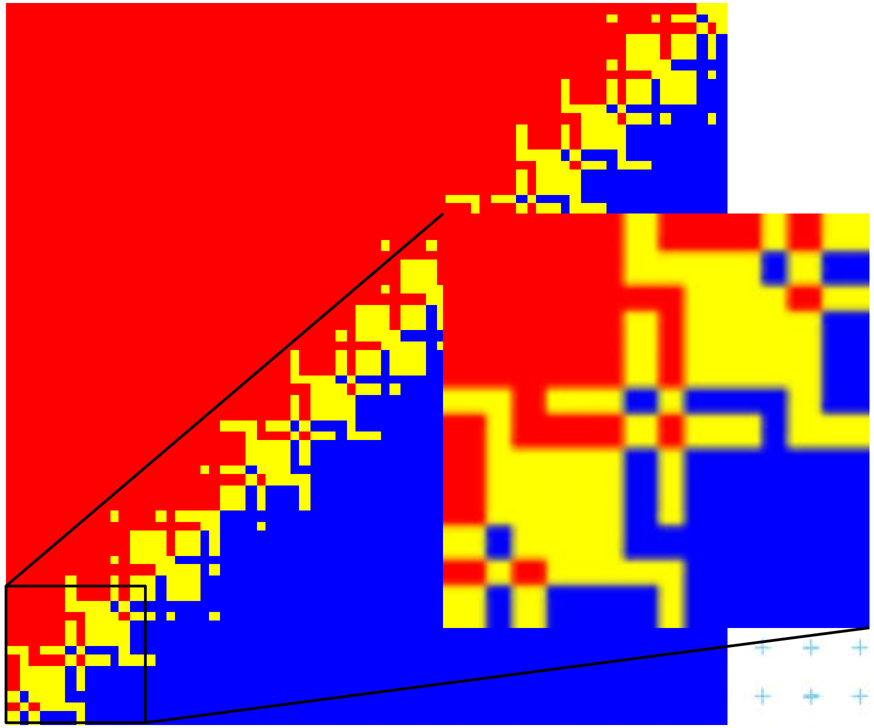
Real results of orientation predicate

- orientation(p,q,r) = sign($(p_x-r_x)(q_y-r_y)-(p_y-r_y)(q_x-r_x)$)

Return values during the experiment for exponent -52



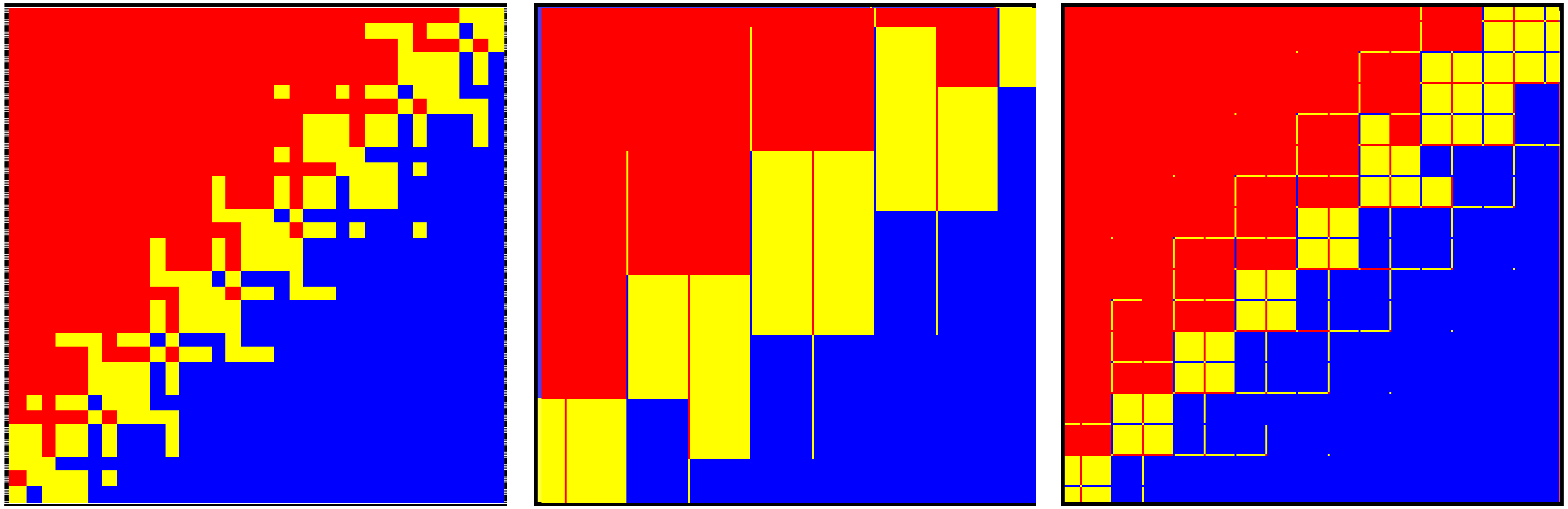
Pivot r_{24}



Pivot $p_{0.5}$

Floating point orientation predicate double exp=-53

Pivot *p*



$$p:\begin{pmatrix}0.5\\0.5\end{pmatrix}\quad q:\begin{pmatrix}12\\12\end{pmatrix}\quad r:\begin{pmatrix}24\\24\end{pmatrix}$$

(a)

$$\begin{pmatrix}0.50000000000002531\\0.5000000000000171\end{pmatrix}\quad\begin{pmatrix}17.300000000000001\\17.300000000000001\end{pmatrix}\quad\begin{pmatrix}24.00000000000005\\24.0000000000000517765\end{pmatrix}$$

(b)

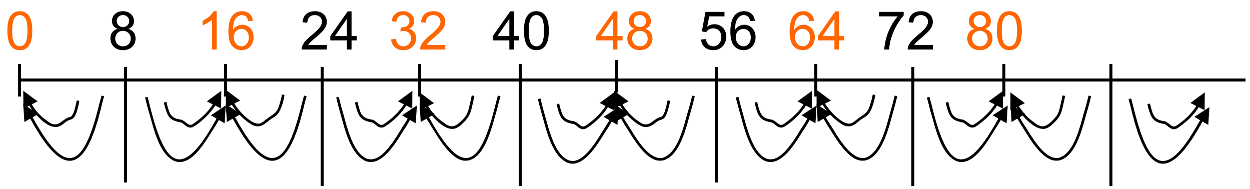
$$\begin{pmatrix}0.5\\0.5\end{pmatrix}\quad\begin{pmatrix}8.8000000000000007\\8.8000000000000007\end{pmatrix}\quad\begin{pmatrix}12.1\\12.1\end{pmatrix}$$

(c)

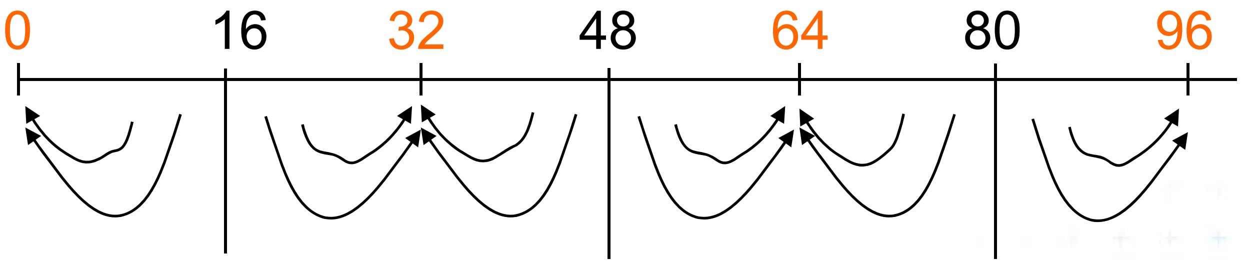
[Kettner] with correct coolors

Errors from shift ~0.5 right in subtraction

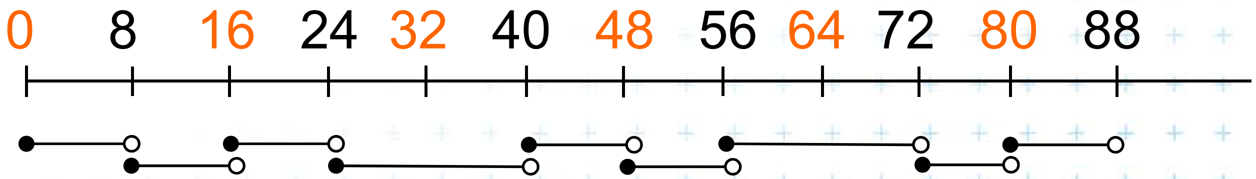
- 4 bits shift => 2^4 values rounded to the same value

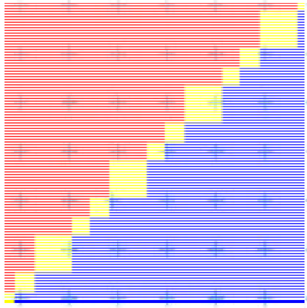


- 5 bits shift => 2^5 values rounded to the same value



- Combined intervals of size 8, 16, 24,…





These intervals match the size of rectangular areas of the same value

Orientation predicate – pivot selection

$$\text{orientation}(p,q,r) = \text{sign}\left(\det\begin{bmatrix}1 & p_x & p_y\\ 1 & q_x & q_y\\ 1 & r_x & r_y\end{bmatrix}\right) =$$

The formula depends on choose of the pivot = row to be subtracted from other rows

$$= \text{sign}\left((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)\right)$$

$$= \text{sign}\left((r_x - q_x)(p_y - q_y) - (r_y - q_y)(p_x - q_x)\right)$$

$$= \text{sign}\left((p_x - r_x)(q_y - r_y) - (p_y - r_y)(q_x - r_x)\right)$$

Orientation predicate – pivot selection

$$\text{orientation}(p,q,r) = \text{sign}\left(\det\begin{bmatrix}1 & p_x & p_y\\ 1 & q_x & q_y\\ 1 & r_x & r_y\end{bmatrix}\right) =$$

The formula depends on choose of the pivot = row to be subtracted from other rows

$$= \text{sign}\left((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)\right)$$

$$= \text{sign}\left((r_x - q_x)(p_y - q_y) - (r_y - q_y)(p_x - q_x)\right)$$

$$= \text{sign}\left((p_x - r_x)(q_y - r_y) - (p_y - r_y)(q_x - r_x)\right)$$

Which order is the worst?

Orientation predicate – pivot selection

$$\text{orientation}(p,q,r) = \text{sign}\left(\det\begin{bmatrix}1 & p_x & p_y\\ 1 & q_x & q_y\\ 1 & r_x & r_y\end{bmatrix}\right) =$$

The formula depends on choose of the pivot = row to be subtracted from other rows

$$= \text{sign}\left((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)\right)$$

$$= \text{sign}\left((r_x - q_x)(p_y - q_y) - (r_y - q_y)(p_x - q_x)\right)$$

$$= \text{sign}\left((p_x - r_x)(q_y - r_y) - (p_y - r_y)(q_x - r_x)\right)$$

$$p_x = 0.5,\ q_x = 12,\ r_x = 24$$

Orientation predicate – pivot selection

$$\text{orientation}(p, q, r) = \text{sign}\left(\det\begin{bmatrix}1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y\end{bmatrix}\right) =$$

The formula depends on choose of the pivot = row to be subtracted from other rows

$$= \text{sign}\left((q_x \overset{\text{4 bits lost}}{-} p_x)(r_y - p_y) - (q_y \overset{\text{4 bits lost}}{-} p_y)(r_x - p_x)\right)$$

$$= \text{sign}\left((r_x - q_x)(\overset{\text{4 bits lost}}{p_y - q_y}) - (r_y - q_y)(\overset{\text{4 bits lost}}{p_x - q_x})\right)$$

$$= \text{sign}\left((p_x - r_x)(q_y - r_y) - (p_y - r_y)(q_x - r_x)\right)$$

$$p_x = 0.5,\ q_x = 12,\ r_x = 24$$

Orientation predicate – pivot selection

$$\mathrm{orientation}(p, q, r) = \mathrm{sign}\left(\det\begin{bmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{bmatrix}\right) =$$

The formula depends on choose of the pivot = row to be subtracted from other rows

$$= \mathrm{sign}\left((q_x \overset{\text{4 bits lost}}{-} p_x)(r_y \overset{\text{5 bits lost}}{-} p_y) - (q_y \overset{\text{4 bits lost}}{-} p_y)(r_x \overset{\text{5 bits lost}}{-} p_x)\right)$$

$$= \mathrm{sign}\left((r_x - q_x)(\overset{\text{4 bits lost}}{p_y} - q_y) - (r_y - q_y)(\overset{\text{4 bits lost}}{p_x} - q_x)\right)$$

$$= \mathrm{sign}\left((\overset{\text{5 bits lost}}{p_x} - r_x)(q_y - r_y) - (\overset{\text{5 bits lost}}{p_y} - r_y)(q_x - r_x)\right)$$

$$p_x = 0.5, \; q_x = 12, \; r_x = 24$$

Orientation predicate – pivot selection

$$\text{orientation}(p, q, r) = \text{sign}\left(\det\begin{bmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{bmatrix}\right) =$$

The formula depends on choose of the pivot = row to be subtracted from other rows

$$= \boxed{\text{sign}\left((q_x - \overset{\text{4 bits lost}}{p_x})(r_y - \overset{\text{5 bits lost}}{p_y}) - (q_y - \overset{\text{4 bits lost}}{p_y})(r_x - \overset{\text{5 bits lost}}{p_x})\right)}$$

$$= \text{sign}\left((r_x - q_x)(\overset{\text{4 bits lost}}{p_y} - q_y) - (r_y - q_y)(\overset{\text{4 bits lost}}{p_x} - q_x)\right)$$

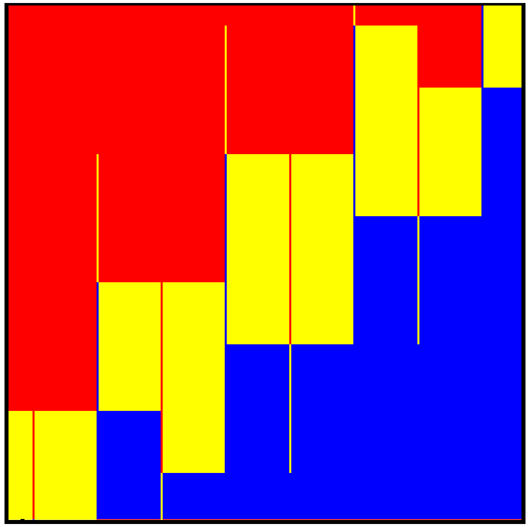
$$= \text{sign}\left((\overset{\text{5 bits lost}}{p_x} - r_x)(q_y - r_y) - (\overset{\text{5 bits lost}}{p_y} - r_y)(q_x - r_x)\right)$$

$$p_x = 0.5,\ q_x = 12,\ r_x = 24$$

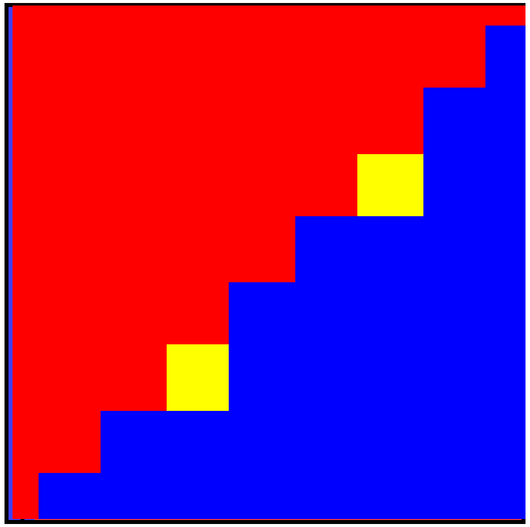
Little improvement - selection of the pivot

(b) double exp=-53

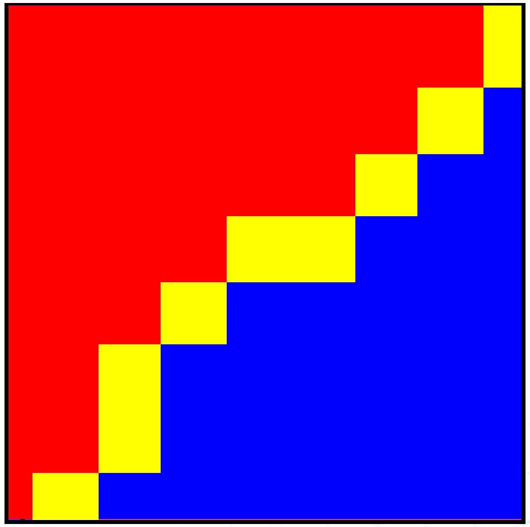
- Pivot – subtracted from the rows in the matrix



Pivot *p* 0.5



Pivot *q* 12



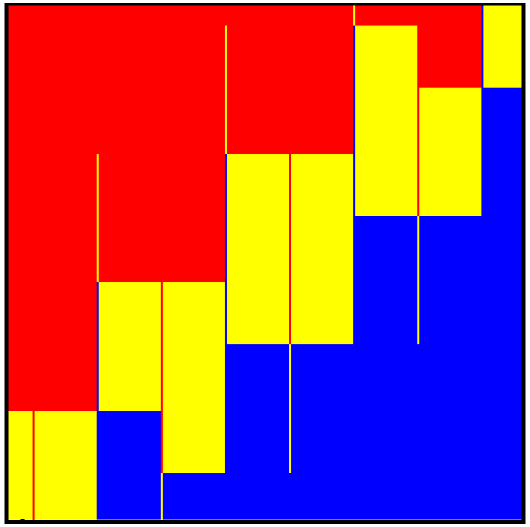
Pivot *r* 24

[Kettner]

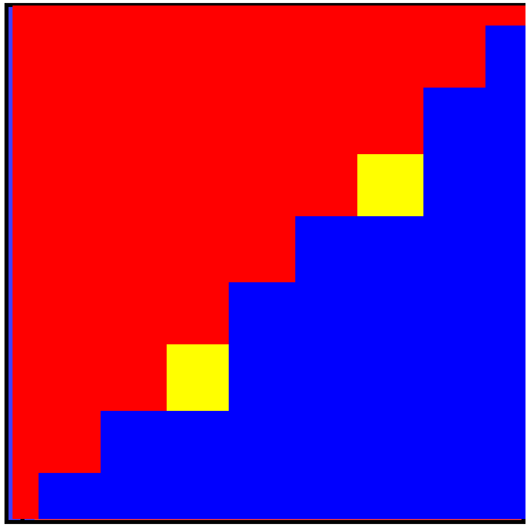
Little improvement - selection of the pivot

(b) double exp=-53

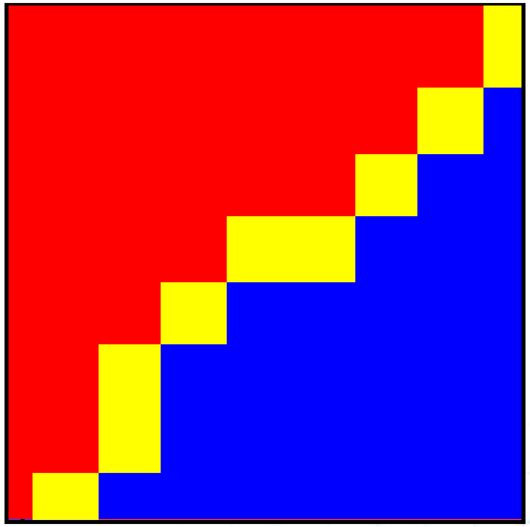
- Pivot – subtracted from the rows in the matrix



Pivot *p* 0.5



Pivot *q* 12



Pivot *r* 24

=> Pivot *q* (point with middle *x* or *y* coord.) is the best
But it is typically not used – pivot search is too complicated in comparison to the predicate itself

[Kettner]

Epsilon tweaking

[Kettner]

Epsilon tweaking

- Use tolerance ε =0.00005 to 0.0001 for float

[Kettner]

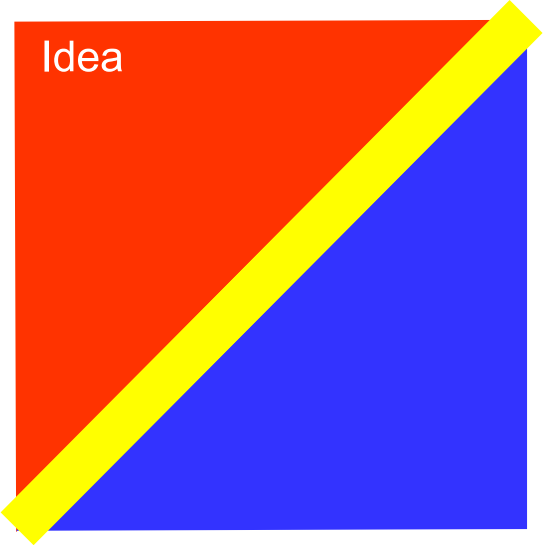
Epsilon tweaking

- Use tolerance ε =0.00005 to 0.0001 for float
- Points are declared collinear if float_orient returns a value ≤ ε   0.5+2^(-23) , the smallest repr. value 0.500 000 06

[Kettner]

Epsilon tweaking

- Use tolerance ε =0.00005 to 0.0001 for float
- Points are declared collinear if float_orient returns a value ≤ ε   0.5+2^(-23) , the smallest repr. value 0.500 000 06

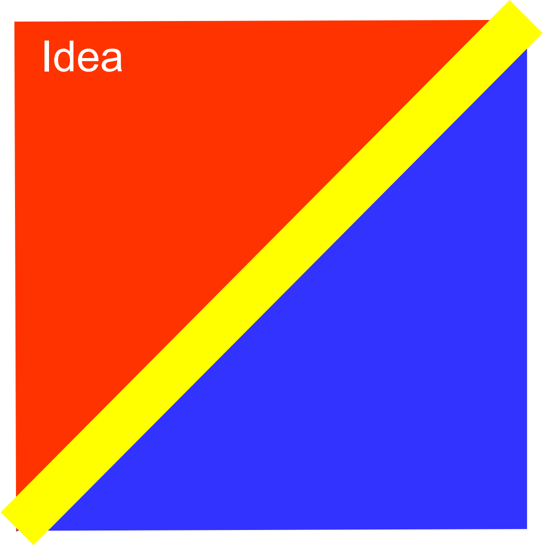


Idea: boundary for ε

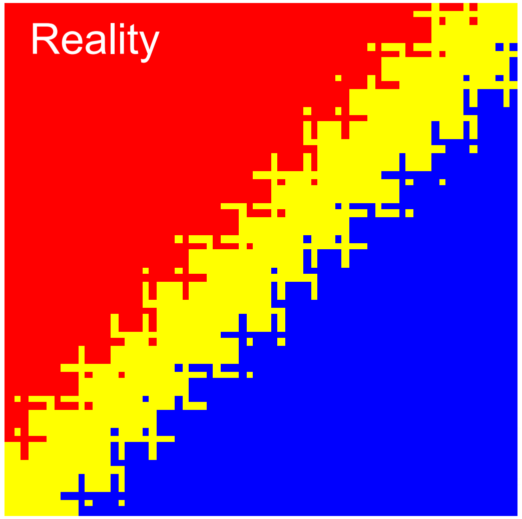
[Kettner]

Epsilon tweaking

- Use tolerance ε =0.00005 to 0.0001 for float
- Points are declared collinear if float_orient returns a value ≤ ε   0.5+2^(-23) , the smallest repr. value 0.500 000 06



Idea: boundary for ε

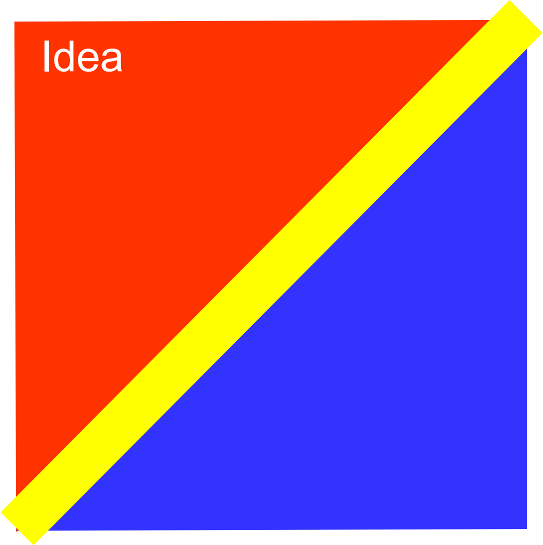


Boundary for ε= 0.00005

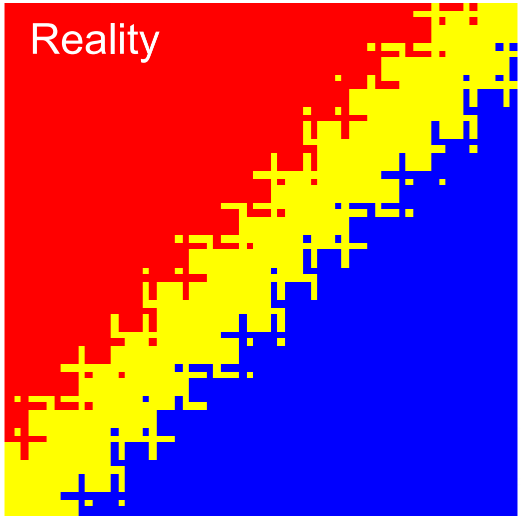
[Kettner]

Epsilon tweaking

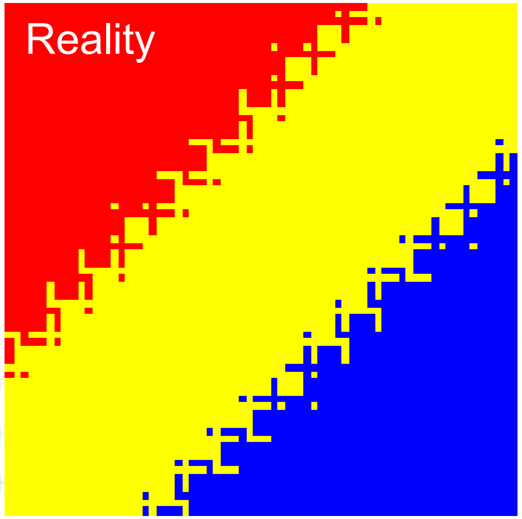
- Use tolerance ε =0.00005 to 0.0001 for float
- Points are declared collinear if float_orient returns a value ≤ ε   0.5+2^(-23) , the smallest repr. value 0.500 000 06



Idea: boundary for ε



Boundary for ε= 0.00005

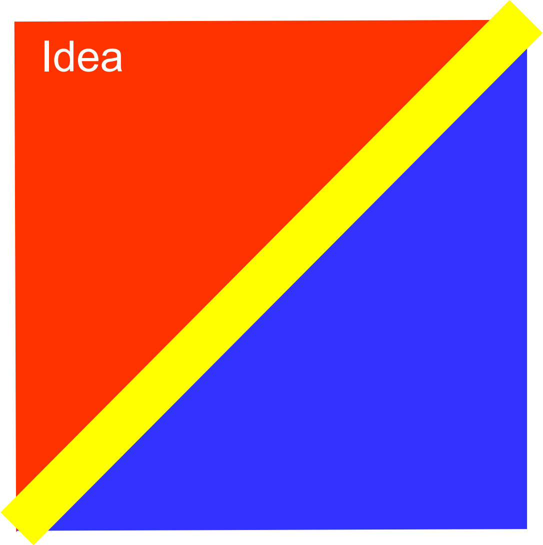


Boundary for ε= 0.0001

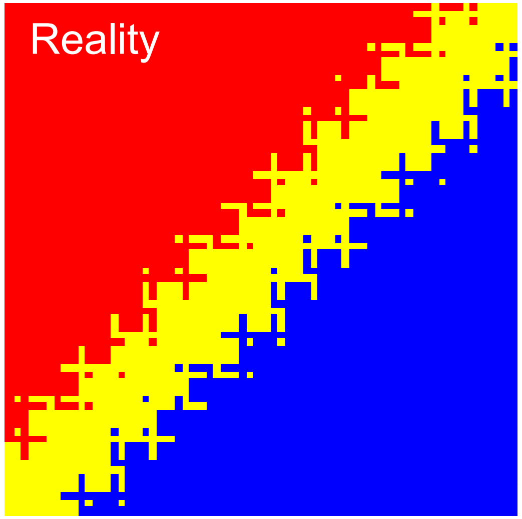
[Kettner]

Epsilon tweaking

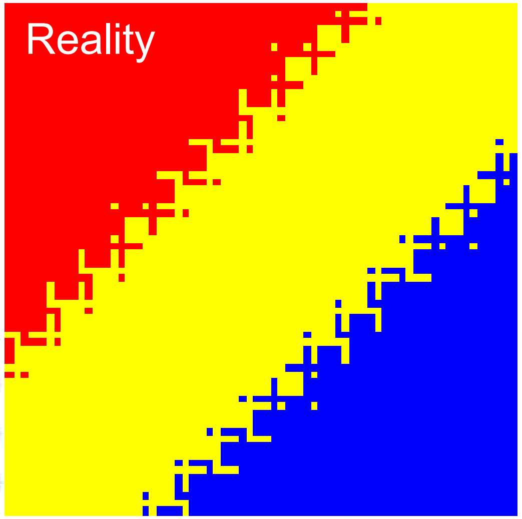
- Use tolerance ε =0.00005 to 0.0001 for float
- Points are declared collinear if float_orient returns a value ≤ ε   0.5+2^(-23) , the smallest repr. value 0.500 000 06



Idea: boundary for ε



Boundary for ε= 0.00005



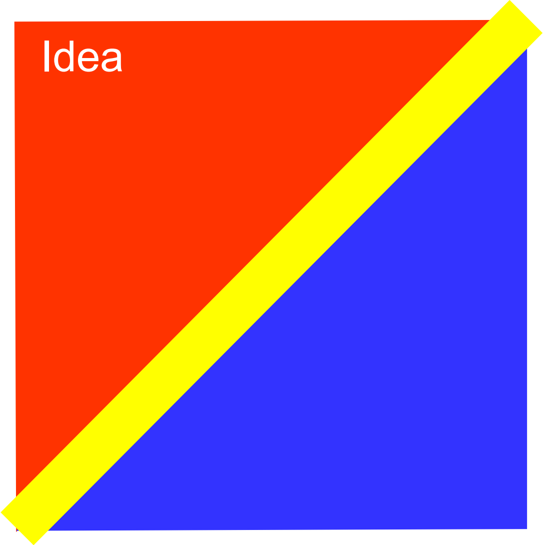
Boundary for ε= 0.0001

- Boundary is fractured as before, but brighter

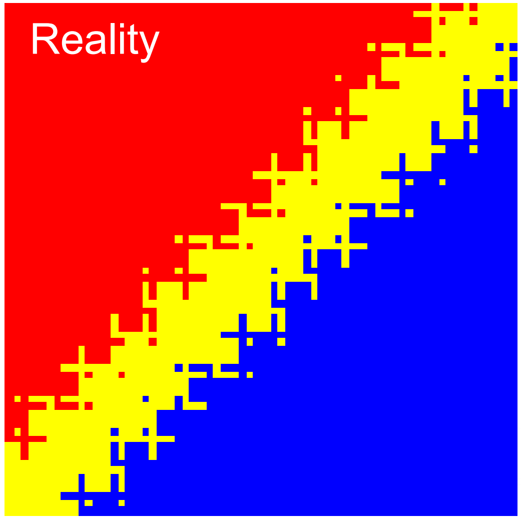
[Kettner]

Epsilon tweaking – is the wrong approach

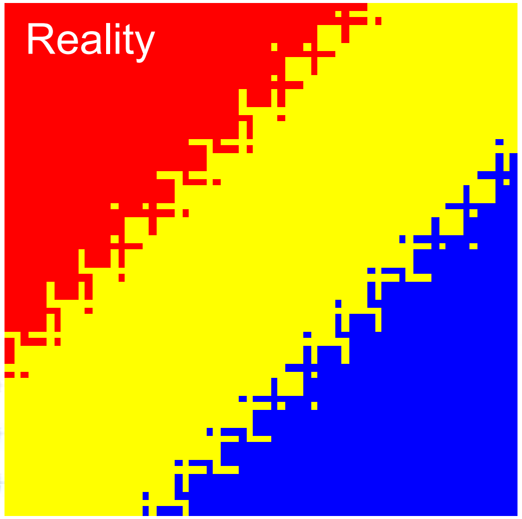
- Use tolerance ε =0.00005 to 0.0001 for float
- Points are declared collinear if float_orient returns a value ≤ ε   0.5+2^(-23) , the smallest repr. value 0.500 000 06



Idea: boundary for ε



Boundary for ε= 0.00005

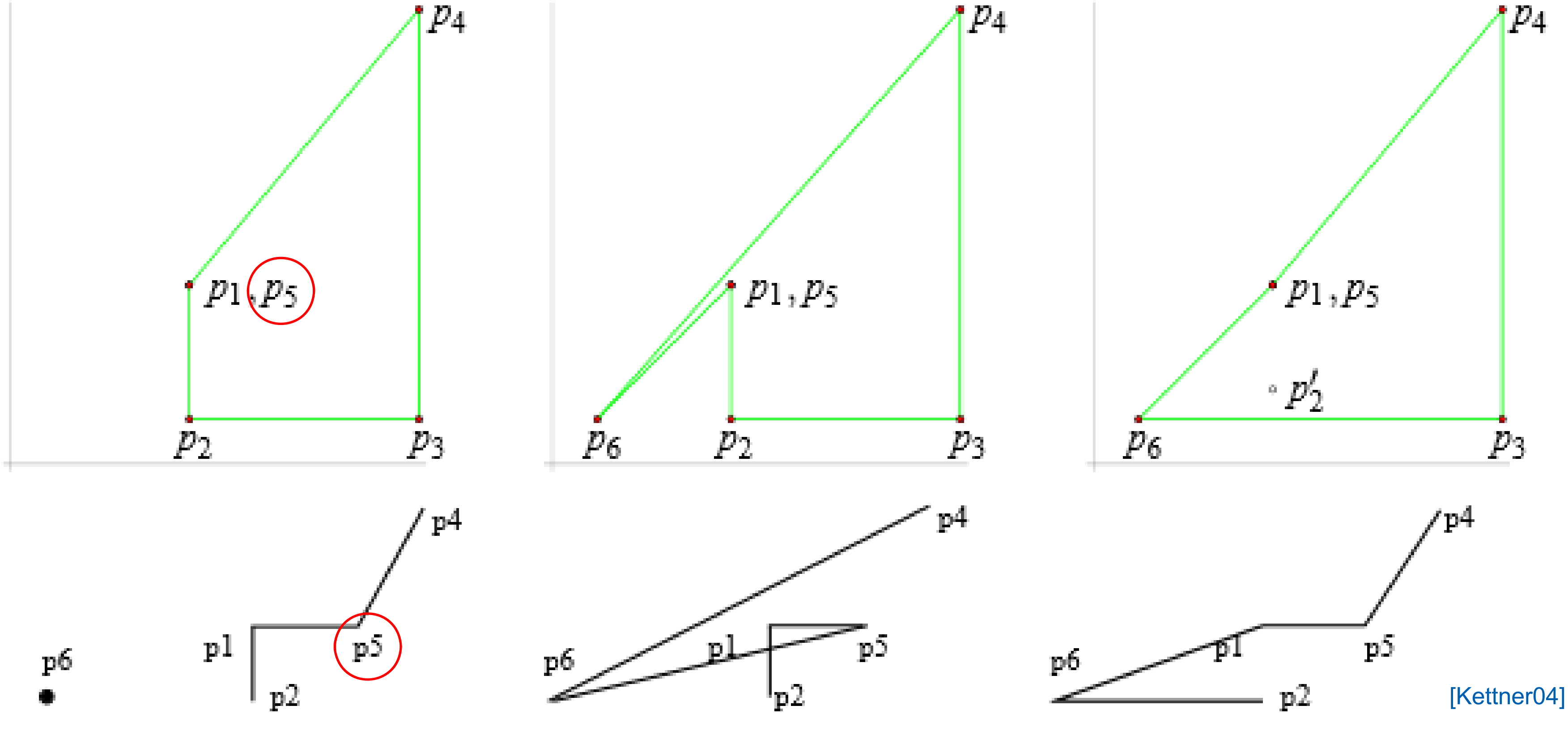


Boundary for ε= 0.0001

- Boundary is fractured as before, but brighter

[Kettner]

Consequences in convex hull algorithm



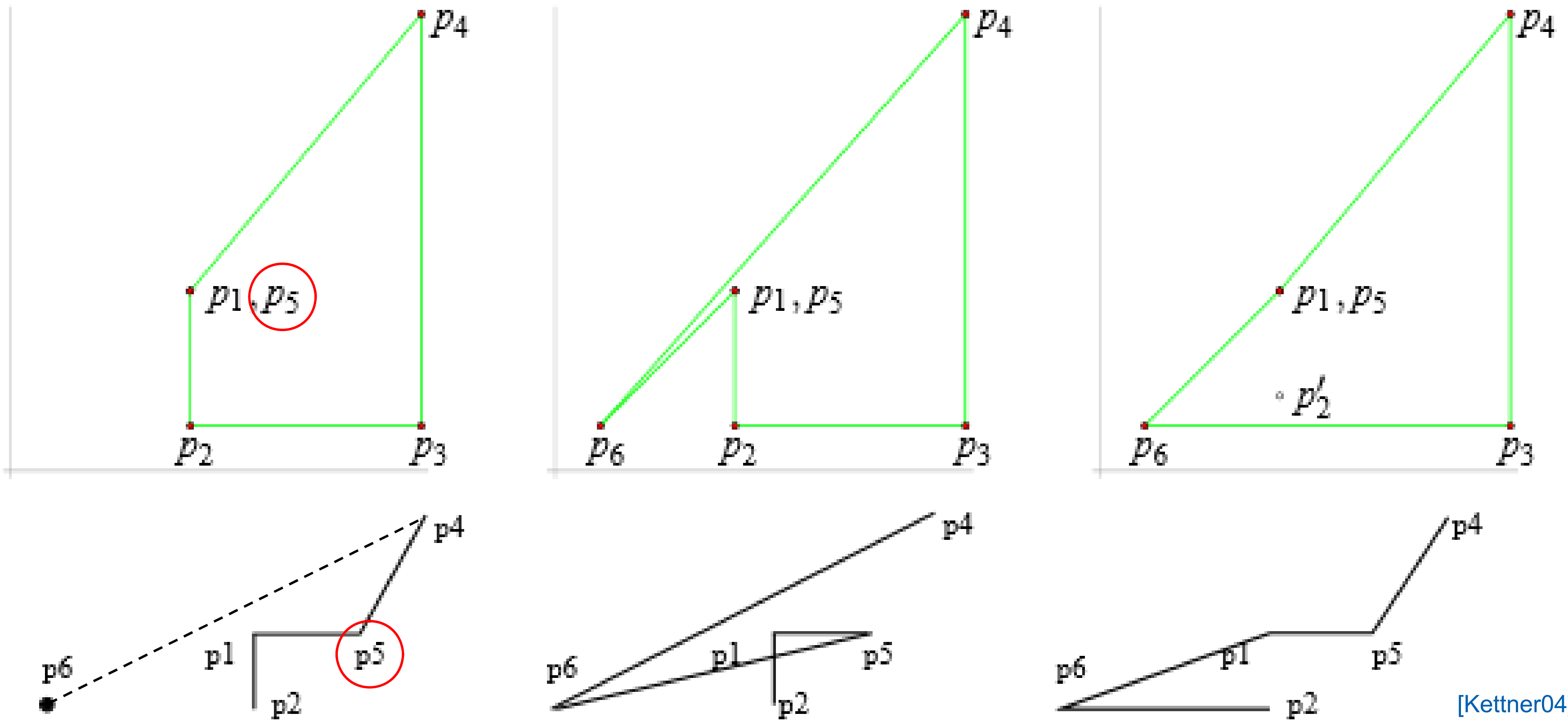
[Kettner04]

p_5 erroneously inserted
Inserting p_6 =>

a) p_6 sees $p_4 p_5$ first
=> forms p_4 p_6 p_5

b) p_6 sees $p_1 p_2$ first
=> forms p_1 p_6 p_2

Consequences in convex hull algorithm



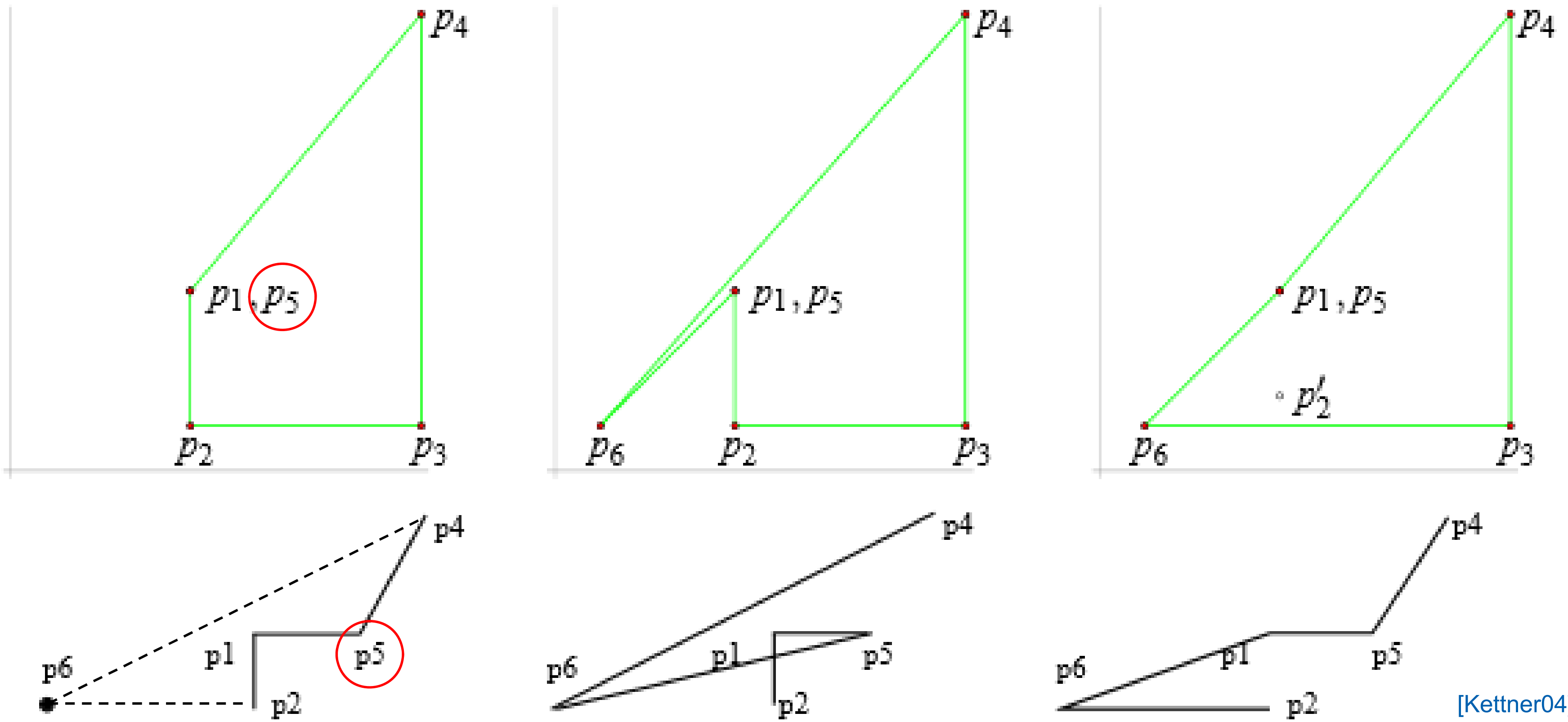
[Kettner04]

p_5 erroneously inserted
Inserting p_6 =>

a) p_6 sees p_4p_5 first
=> forms p_4 p_6 p_5

b) p_6 sees p_1p_2 first
=> forms p_1 p_6 p_2

Consequences in convex hull algorithm



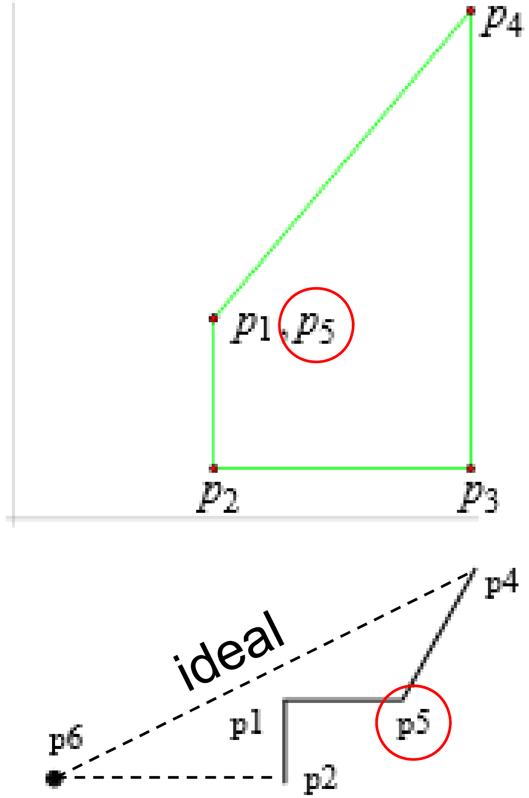
[Kettner04]

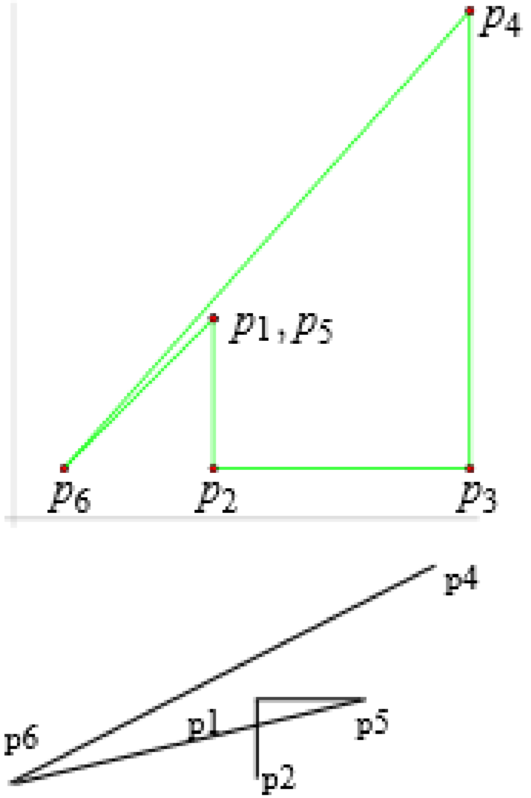
p_5 erroneously inserted
Inserting p_6 =>

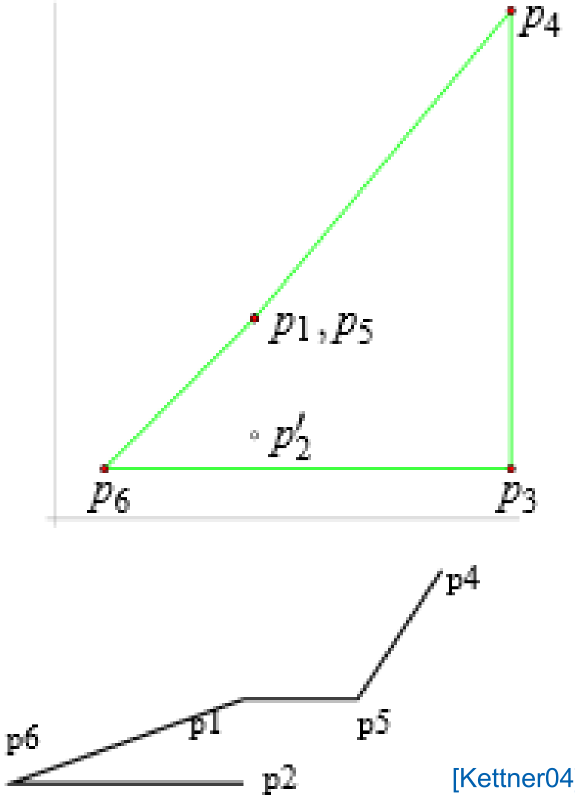
a) p_6 sees p_4p_5 first
=> forms p_4 p_6 p_5

b) p_6 sees p_1p_2 first
=> forms p_1 p_6 p_2

Consequences in convex hull algorithm







[Kettner04]

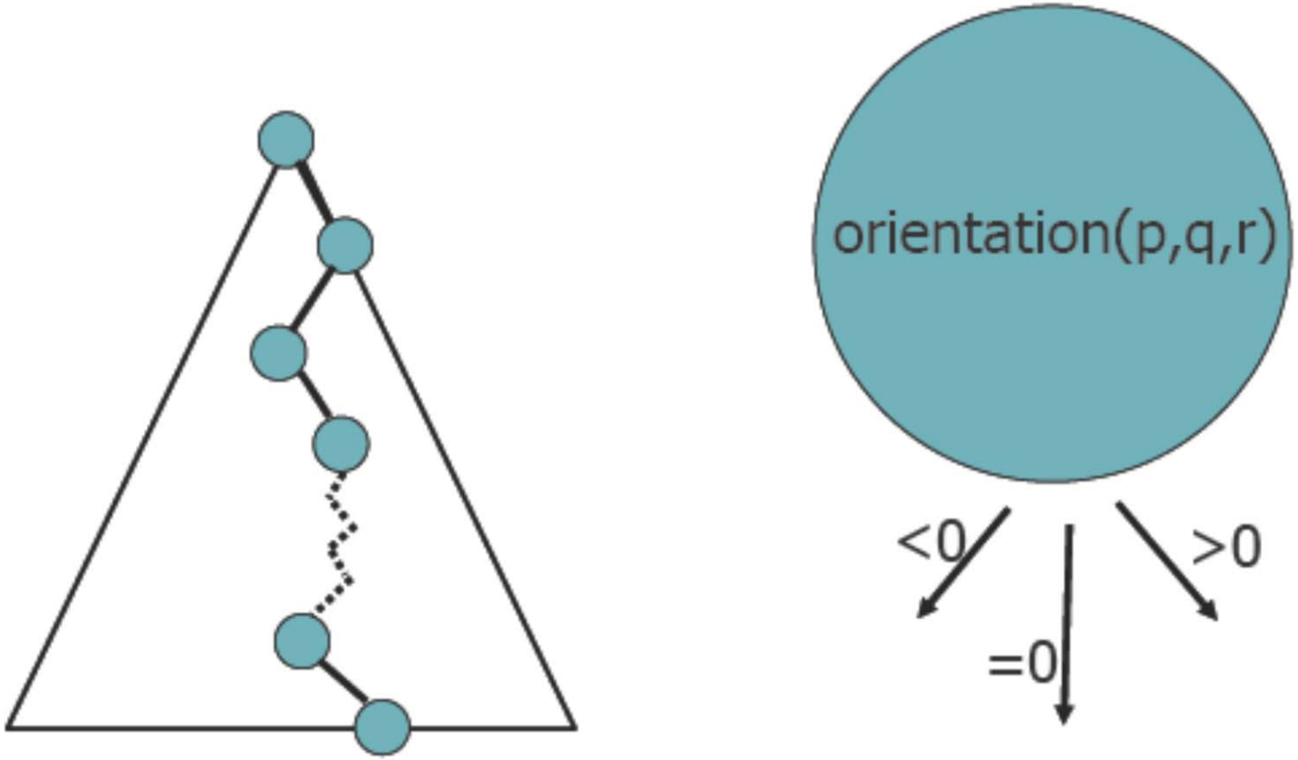
p_5 erroneously inserted
Inserting p_6 =>

a) p_6 sees p_4p_5 first
=> forms p_4 p_6 p_5

b) p_6 sees p_1p_2 first
=> forms p_1 p_6 p_2

Exact Geometric Computing [Yap]

Make sure that the control flow in the implementation corresponds to the control flow with exact real arithmetic

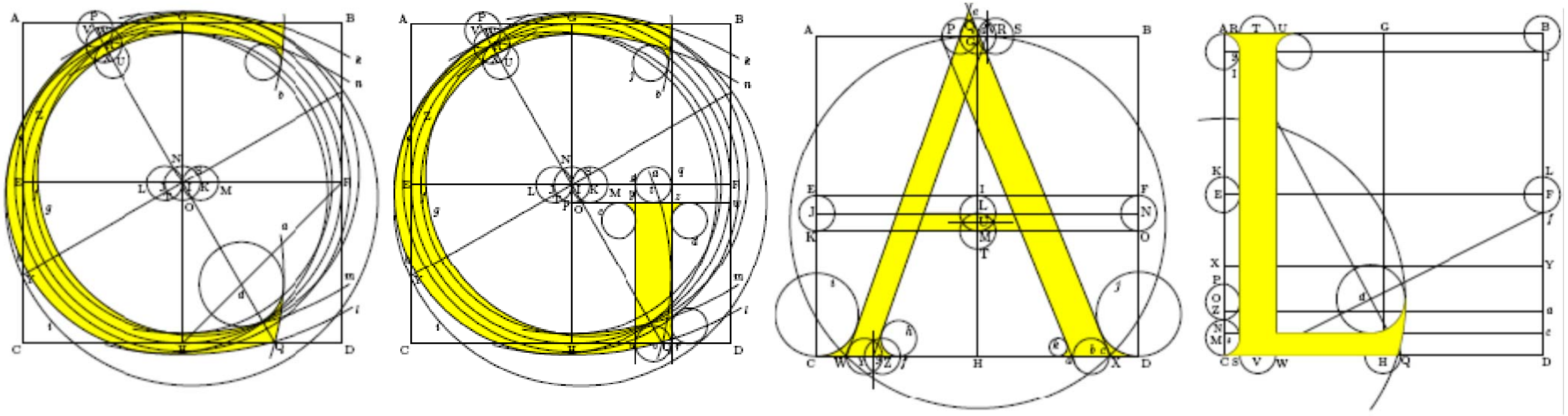


[siggraph2008-CGAL-course]

Solution

1. Use predicates, that always return the correct result -> Schewchuck, YAP, LEDA or CGAL
2. Change the algorithm to cope with floating point predicates but still return something *meaningful* (hard to define)
3. Perturb the input so that the floating point implementation gives the correct result on it

8. CGAL



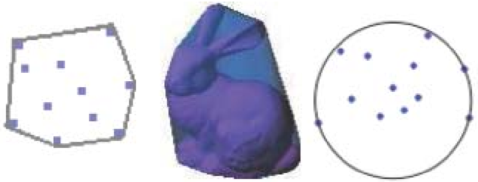
Computational Geometry Algorithms Library

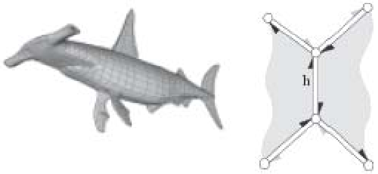
Slides from [siggraph2008-CGAL-course]

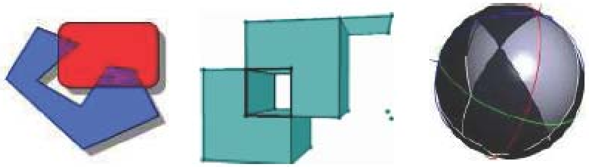
CGAL

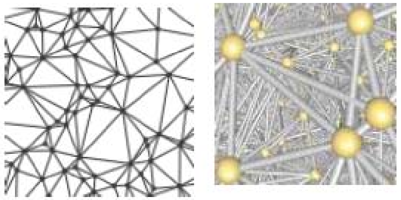
- Large library of geometric algorithms
  - Robust code, huge amount of algorithms
  - Users can concentrate on their own domain
- Open source project
  - Institutional members (Inria, MPI, Tel-Aviv U, Utrecht U, Groningen U, ETHZ, Geometry Factory, FU Berlin, Forth, U Athens)
  - 500,000 lines of C++ code
  - 10,000 downloads/year (+ Linux distributions)
  - 20 active developers
  - 12 months release cycle

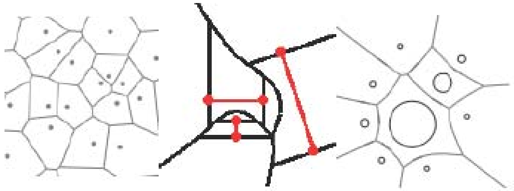
CGAL algorithms and data structures

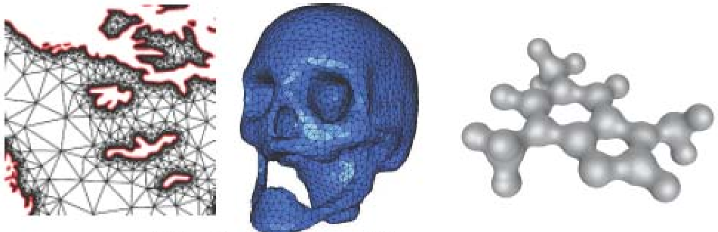

Bounding Volumes

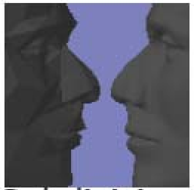

Polyhedral Surface


BooleanOperations


Triangulations

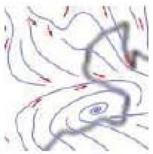

Voronoi Diagrams


Mesh Generation

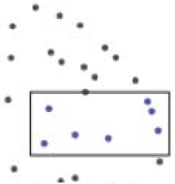

Subdivision

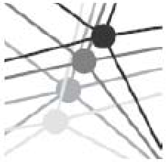
Simplification

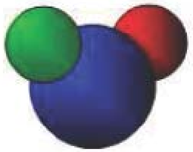
Parametrisation


Streamlines

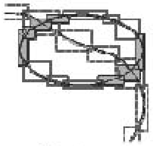
Ridge Detection


Neighbor Search

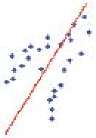

Kinetic Datastructures

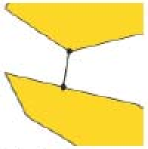

Lower Envelope

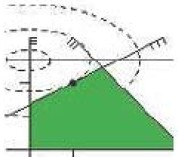
Arrangement


Intersection Detection

Minkowski Sum


PCA


Polytope distance

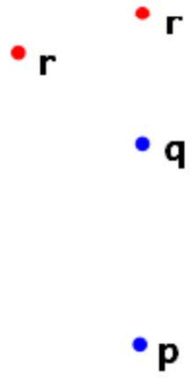

QP Sover

[siggraph2008-CGAL-course]

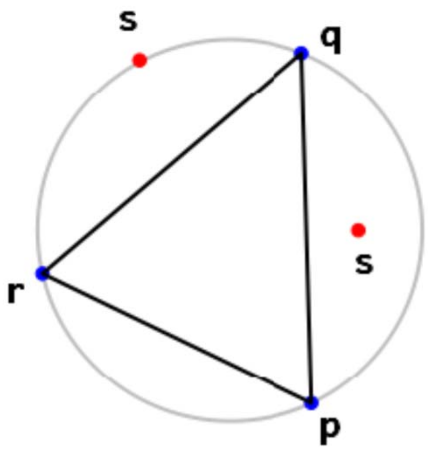
Exact geometric computing

Predicates

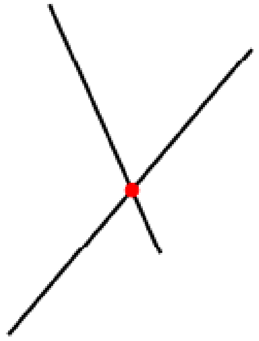
Constructions



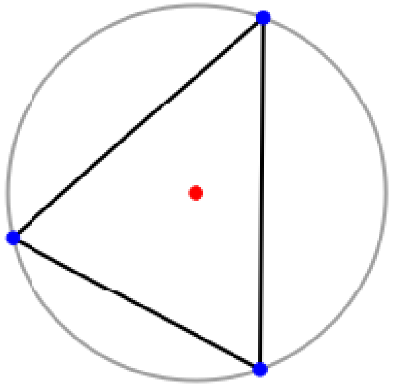
orientation



in_circle



intersection



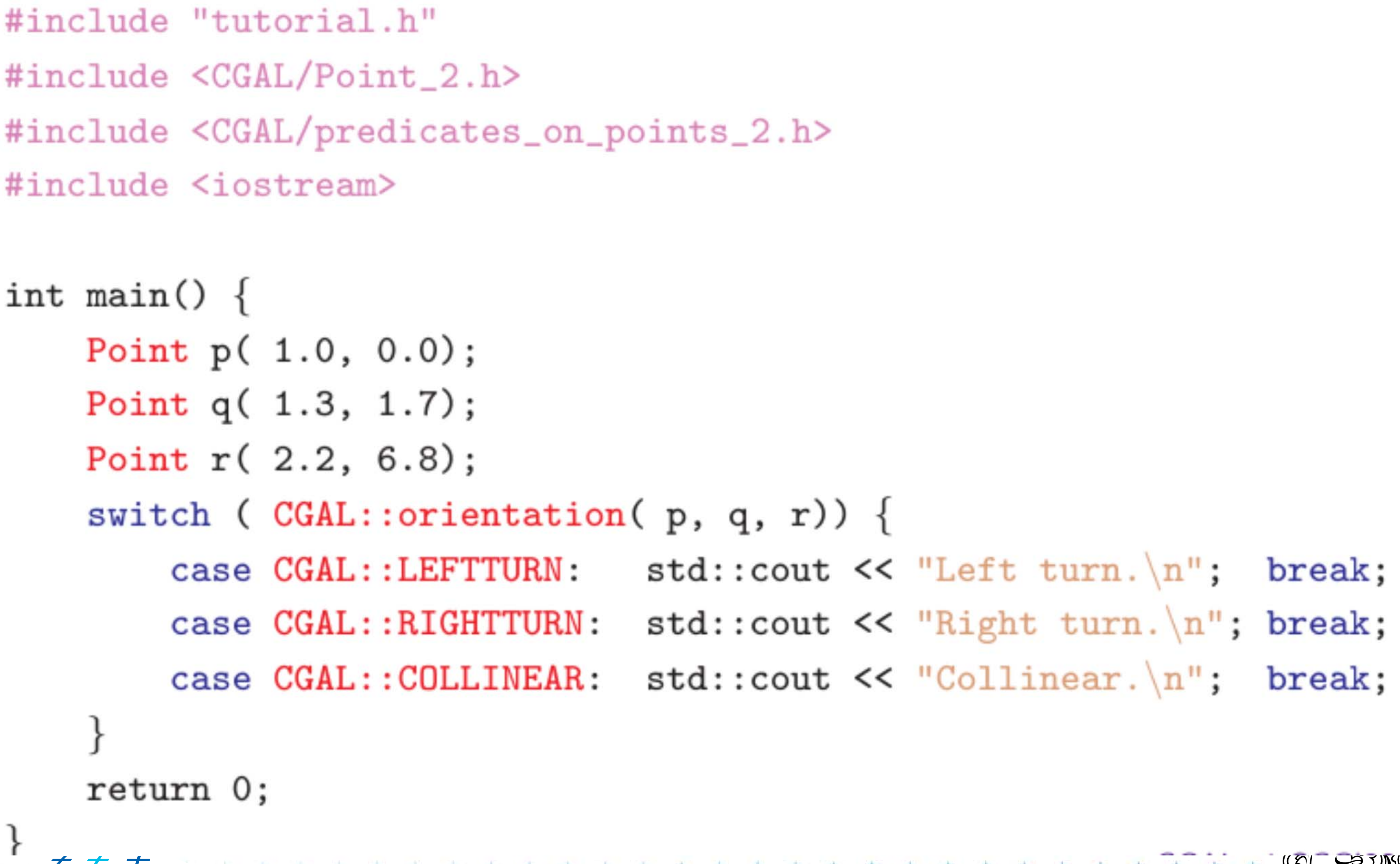
circumcenter

[siggraph2008-CGAL-course]

CGAL Geometric Kernel (see [Hert] for details)

- Encapsulates
  - the representation of geometric objects
  - and the geometric operations and predicates on these objects
- CGAL provides kernels for
  - Points, Predicates, and Exactness
  - Number Types
  - Cartesian Representation
  - Homogeneous Representation

Points, predicates, and Exactness

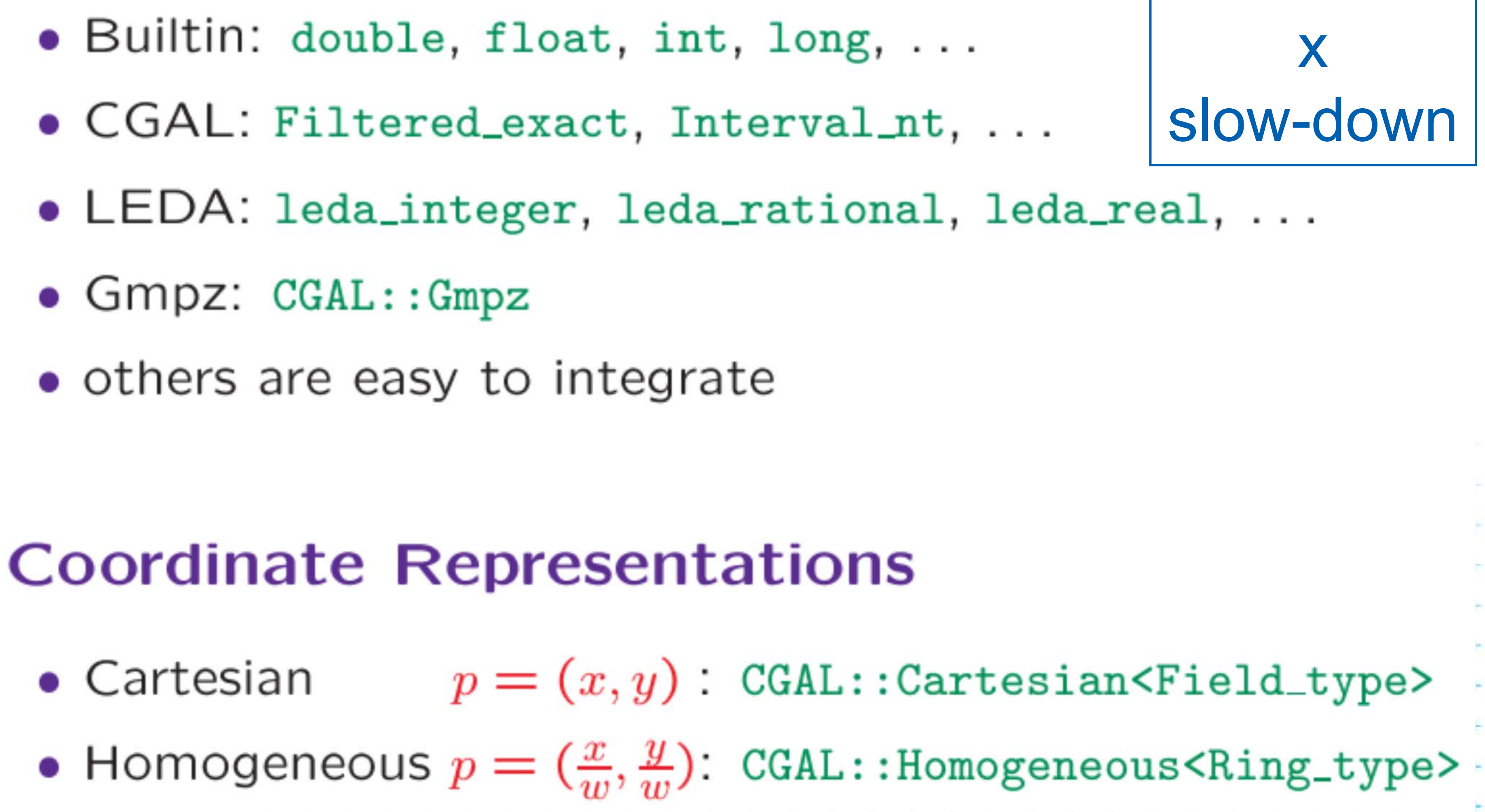


```
#include "tutorial.h"
#include <CGAL/Point_2.h>
#include <CGAL/predicates_on_points_2.h>
#include <iostream>

int main() {
    Point p( 1.0, 0.0);
    Point q( 1.3, 1.7);
    Point r( 2.2, 6.8);
    switch ( CGAL::orientation( p, q, r)) {
        case CGAL::LEFTTURN:   std::cout << "Left turn.\n";  break;
        case CGAL::RIGHTTURN:  std::cout << "Right turn.\n"; break;
        case CGAL::COLLINEAR:  std::cout << "Collinear.\n";  break;
    }
    return 0;
}
```

[CGAL at SCG ‘99]

Number Types

Precission
x
slow-down



- Builtin: `double`, `float`, `int`, `long`, . . .
- CGAL: `Filtered_exact`, `Interval_nt`, . . .
- LEDA: `leda_integer`, `leda_rational`, `leda_real`, . . .
- Gmpz: `CGAL::Gmpz`
- others are easy to integrate

Coordinate Representations

- Cartesian $p = (x, y)$: `CGAL::Cartesian<Field_type>`
- Homogeneous $p = (\frac{x}{w}, \frac{y}{w})$: `CGAL::Homogeneous<Ring_type>`

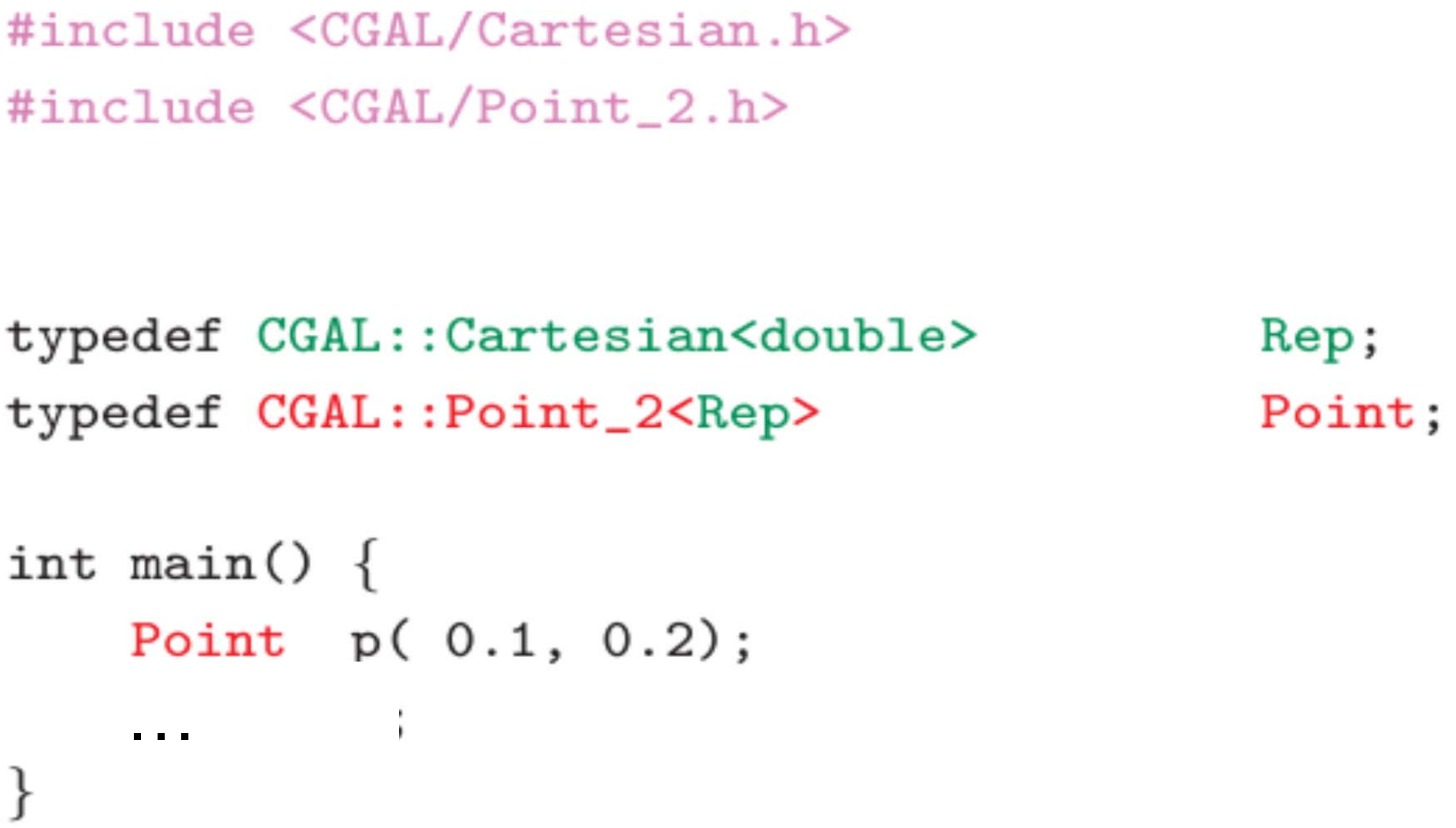
[CGAL at SCG '99]

Cartesian with double

```
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>




int main() {
    CGAL::Point_2< CGAL::Cartesian<double> >  p( 0.1, 0.2);
    ...
}
```

[CGAL at SCG '99]

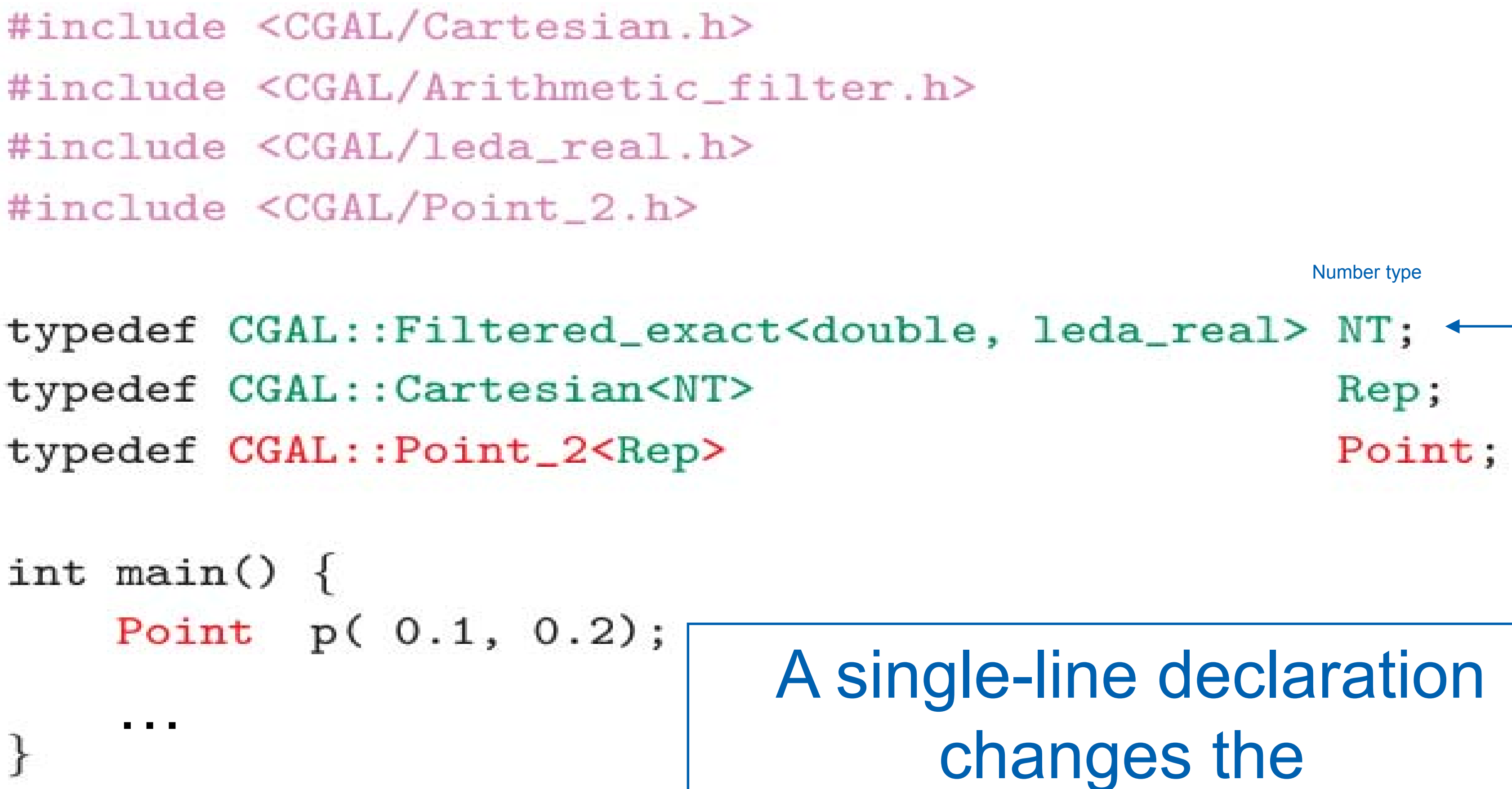
Cartesian with double



```
#include <CGAL/Cartesian.h>
#include <CGAL/Point_2.h>


typedef CGAL::Cartesian<double>           Rep;
typedef CGAL::Point_2<Rep>                Point;

int main() {
    Point  p( 0.1, 0.2);
    ...        ;
}
```

[CGAL at SCG '99]

Cartesian with Filtered_exact and leda_real



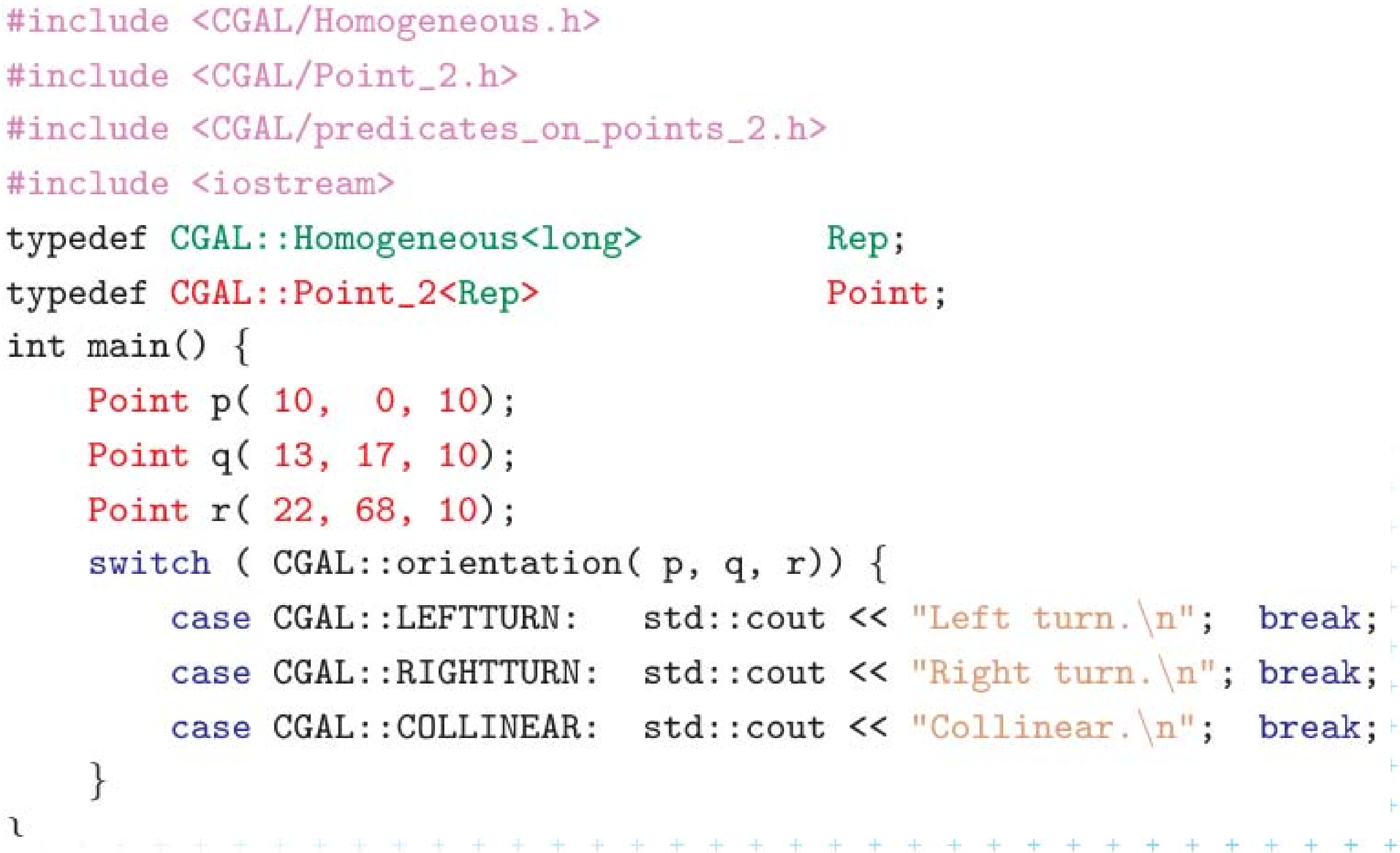
```
#include <CGAL/Cartesian.h>
#include <CGAL/Arithmetic_filter.h>
#include <CGAL/leda_real.h>
#include <CGAL/Point_2.h>

typedef CGAL::Filtered_exact<double, leda_real> NT;
typedef CGAL::Cartesian<NT>                     Rep;
typedef CGAL::Point_2<Rep>                      Point;

int main() {
    Point  p( 0.1, 0.2);
    ...
}
```

Number type

A single-line declaration changes the precision of all computations

[CGAL at SCG '99]

Exact orientation test – homogeneous rep.



```
#include <CGAL/Homogeneous.h>
#include <CGAL/Point_2.h>
#include <CGAL/predicates_on_points_2.h>
#include <iostream>
typedef CGAL::Homogeneous<long>          Rep;
typedef CGAL::Point_2<Rep>               Point;
int main() {
    Point p( 10,  0, 10);
    Point q( 13, 17, 10);
    Point r( 22, 68, 10);
    switch ( CGAL::orientation( p, q, r)) {
        case CGAL::LEFTTURN:   std::cout << "Left turn.\n";  break;
        case CGAL::RIGHTTURN:  std::cout << "Right turn.\n"; break;
        case CGAL::COLLINEAR:  std::cout << "Collinear.\n";  break;
    }
}
```

[CGAL at SCG '99]

9 References – for the lectures

- Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars: Computational Geometry: *Algorithms and Applications*, Springer-Verlag, 3rd rev. ed. 2008. 386 pages, 370 fig. ISBN: 978-3-540-77973-5 http://www.cs.uu.nl/geobook/
- [Mount] Mount, D.: *Computational Geometry Lecture Notes for Spring 2007* http://www.cs.umd.edu/class/spring2007/cmsc754/Lects/comp-geom-lects.pdf
- Franko P. Preperata, Michael Ian Shamos: *Computational Geometry*. An Introduction. Berlin, Springer-Verlag,1985
- Joseph O´Rourke: .: *Computational Geometry in C*, Cambridge University Press, 1993, ISBN 0-521- 44592-2 http://maven.smith.edu/~orourke/books/compgeom.html
- Ivana Kolingerová: *Aplikovaná výpočetní geometrie*, Přednášky, MFF UK 2008
- Kettner et al. *Classroom Examples of Robustness Problems in Geometric Computations*, CGTA 2006, http://www.mpi-inf.mpg.de/~kettner/pub/nonrobust_cgta_06.pdf

9.1 References – CGAL

CGAL

- www.cgal.org
- **Kettner, L.: Tutorial I: Programming with CGAL**
- **Alliez, Fabri, Fogel: Computational Geometry Algorithms Library, SIGGRAPH 2008**
- Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Michael Seel. **An adaptable and extensible geometry kernel**. *Computational Geometry: Theory and Applications*, 38:16-36, 2007. [doi:10.1016/j.comgeo.2006.11.004]

9.2 Useful geometric tools

- OpenSCAD - *The Programmers Solid 3D CAD Modeler*, http://www.openscad.org/
- J.R. Shewchuk - *Adaptive Precision Floating-Point Arithmetic and Fast Robust Predicates*, Effective implementation of Orientation and InCircle predicates http://www.cs.cmu.edu/~quake/robust.html
- *OpenMESH* - A generic and efficient polygon mesh data structure, https://www.openmesh.org/
- *VCG Library* - The Visualization and Computer Graphics Library, http://vcg.isti.cnr.it/vcglib/
- *MeshLab* - A processing system for 3D triangular meshes - https://sourceforge.net/projects/meshlab/?source=navbar

9.3 Collections of geometry resources

- N. Amenta, *Directory of Computational Geometry Software*, http://www.geom.umn.edu/software/cglist/.
- D. Eppstein, *Geometry in Action*, http://www.ics.uci.edu/~eppstein/geom.html.
- Jeff Erickson, *Computational Geometry Pages*, http://compgeom.cs.uiuc.edu/~jeffe/compgeom/

10. Computational geom. course summary

- Gives an overview of geometric algorithms
- Explains their complexity and limitations
- Different algorithms for different data
- We focus on
  - discrete algorithms and precise numbers and predicates
  - principles more than on precise mathematical proofs
  - practical experiences with geometric sw

DCGI

DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

GEOMETRIC SEARCHING PART 1: POINT LOCATION

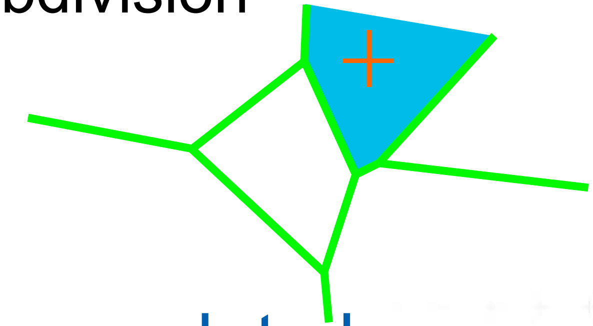
PETR FELKEL

FEL CTU PRAGUE

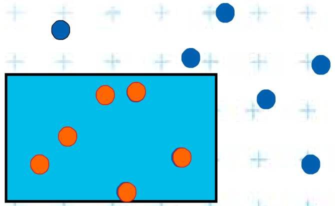
Version from 25.1.2019

Geometric searching problems

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



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

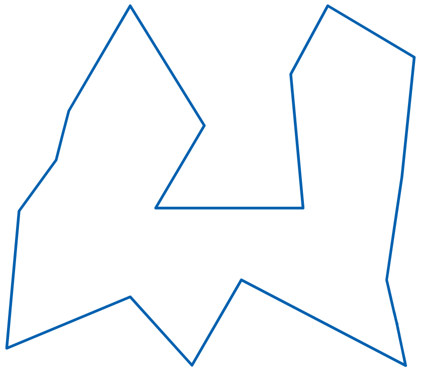


Point location

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

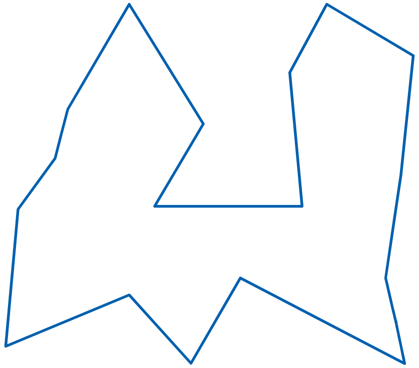
Point location in polygon by ray crossing

1. Ray crossing - O(n)



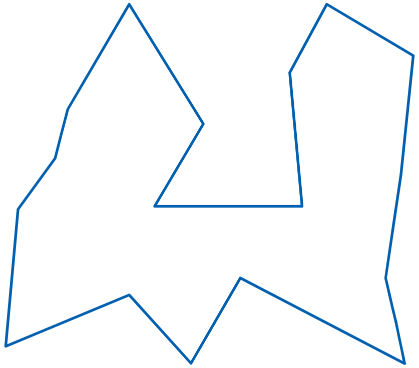
Point location in polygon by ray crossing

1. Ray crossing - O(n)
   - Compute number *t* of ray intersections with polygon edges
     (e.g., ray X+ after point moved to origin)



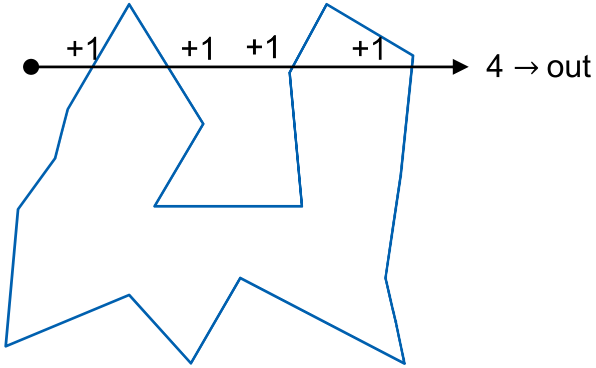
Point location in polygon by ray crossing

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



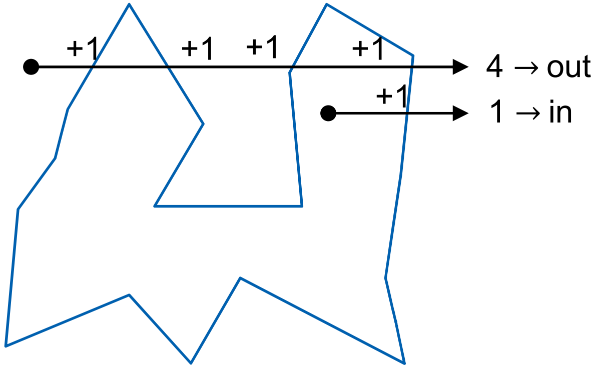
Point location in polygon by ray crossing

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



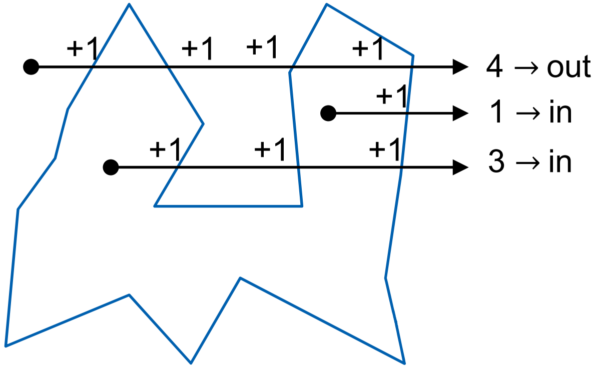
Point location in polygon by ray crossing

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



Point location in polygon by ray crossing

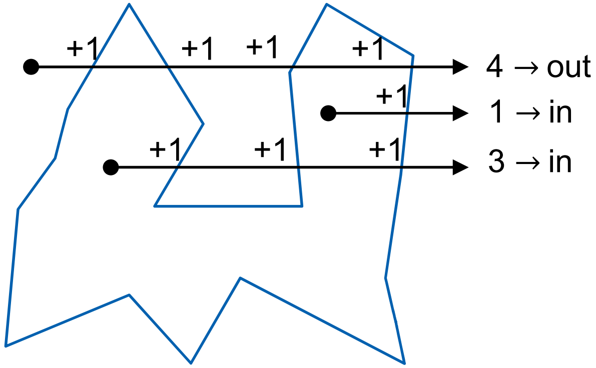
1. Ray crossing - O(n)
   - Compute number *t* of ray intersections with polygon edges (e.g., ray X+ after point moved to origin)
   - If odd(*t*) then inside
     else out



Point location in polygon by ray crossing

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

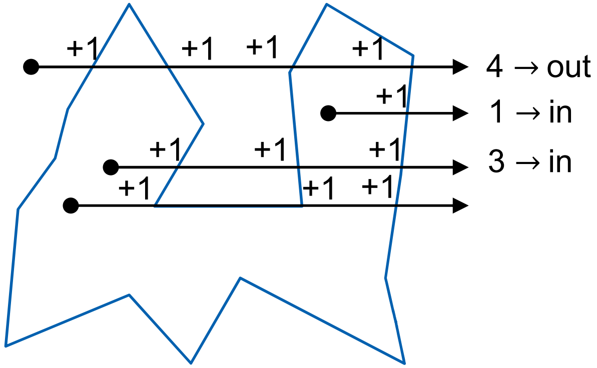
   - Singular cases must be handled!



Point location in polygon by ray crossing

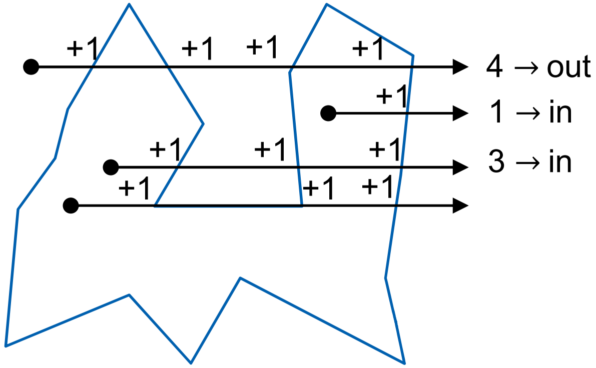
1. Ray crossing - O(n)
   - Compute number *t* of ray intersections with polygon edges
     (e.g., ray X+ after point moved to origin)
   - If odd(*t*) then inside
     else out

   - Singular cases must be handled!

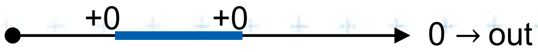


Point location in polygon by ray crossing

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

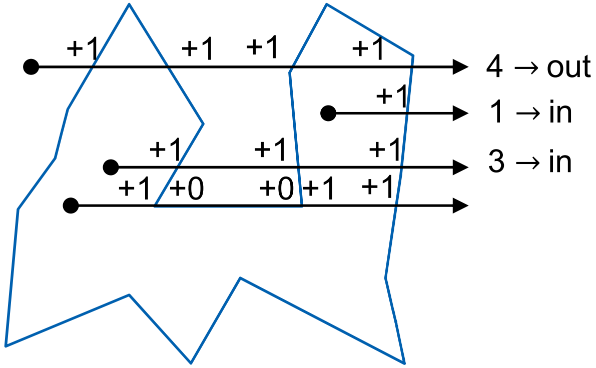


   - Singular cases must be handled!
     - Do not count horizontal line segments

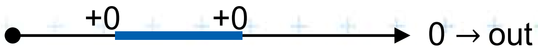


Point location in polygon by ray crossing

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



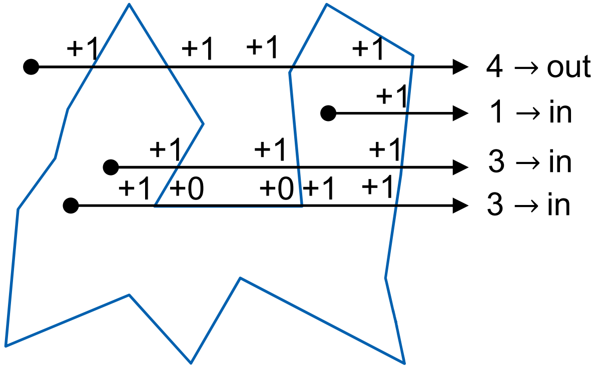
   - Singular cases must be handled!
     - Do not count horizontal line segments



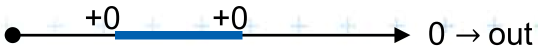
Point location in polygon by ray crossing

1. Ray crossing - O(n)

- Compute number *t* of ray intersections with polygon edges (e.g., ray X+ after point moved to origin)
- If odd(*t*) then inside
  else out

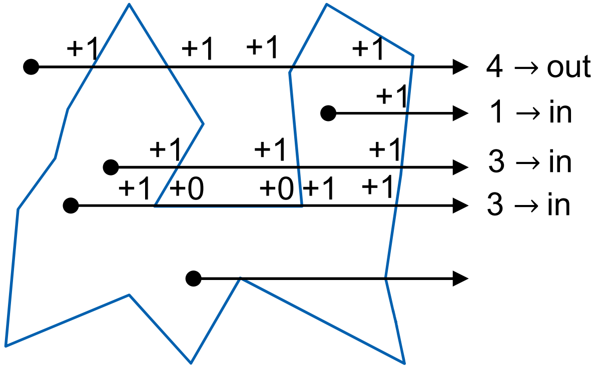


- Singular cases must be handled!
  - Do not count horizontal line segments

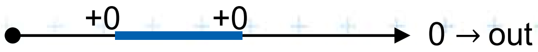


Point location in polygon by ray crossing

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



   - Singular cases must be handled!
     - Do not count horizontal line segments

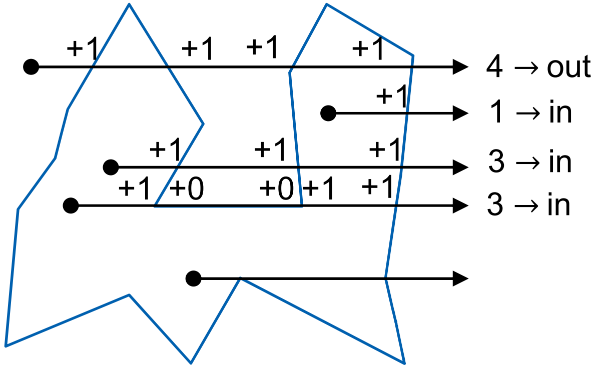


Point location in polygon by ray crossing

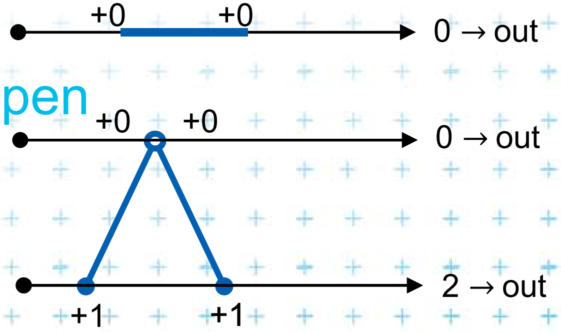
1. Ray crossing - O(n)

   – Compute number *t* of ray intersections with polygon edges
   (e.g., ray X+ after point moved to origin)

   – If odd(*t*) then inside
   else out

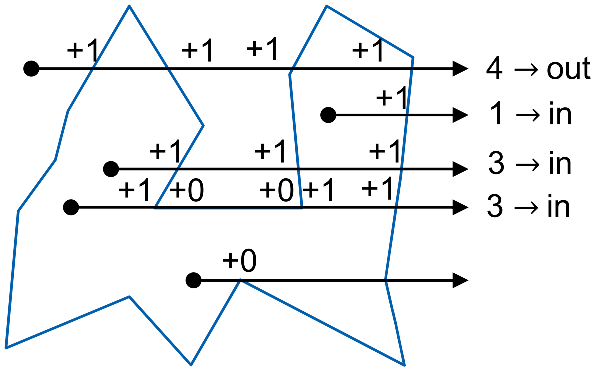


   – Singular cases must be handled!
   - Do not count horizontal line segments
   - Take non-horizontal segments as half-open (upper point not part of the segment)

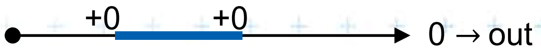


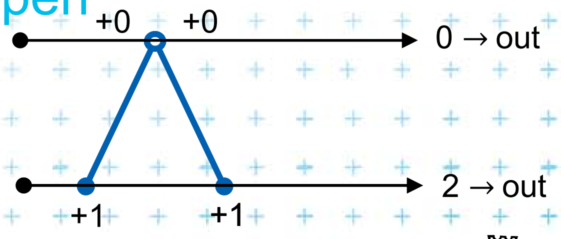
Point location in polygon by ray crossing

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



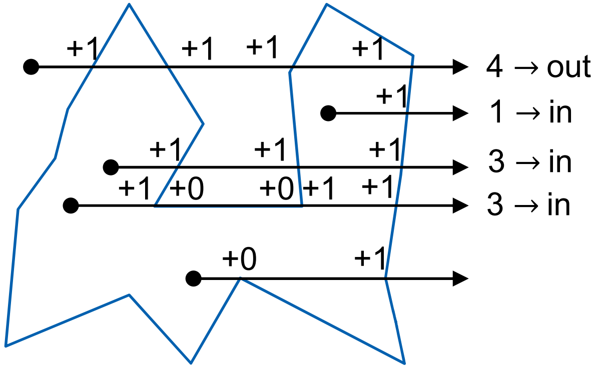
   - Singular cases must be handled!
     - Do not count horizontal line segments
     - Take non-horizontal segments as half-open
       (upper point not part of the segment)



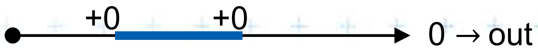


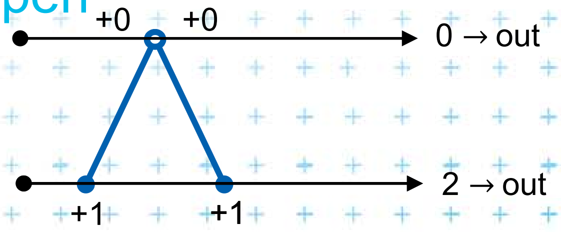
Point location in polygon by ray crossing

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



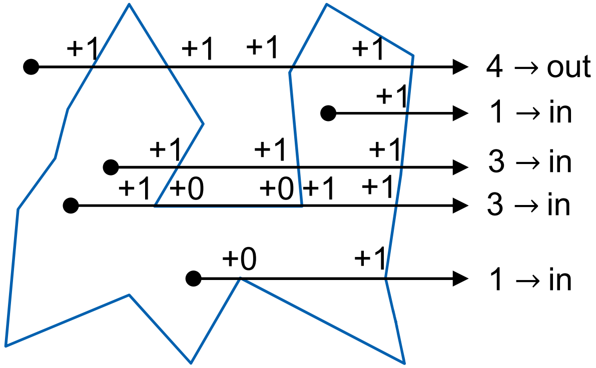
   - Singular cases must be handled!
     - Do not count horizontal line segments
     - Take non-horizontal segments as half-open (upper point not part of the segment)



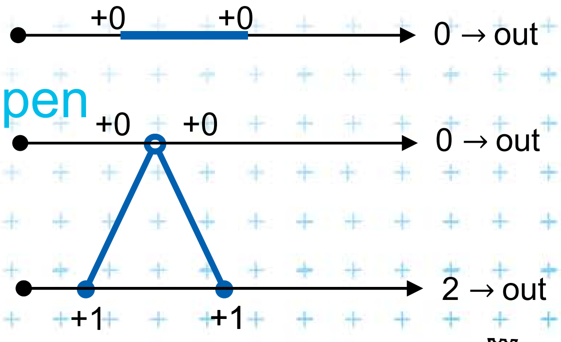


Point location in polygon by ray crossing

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

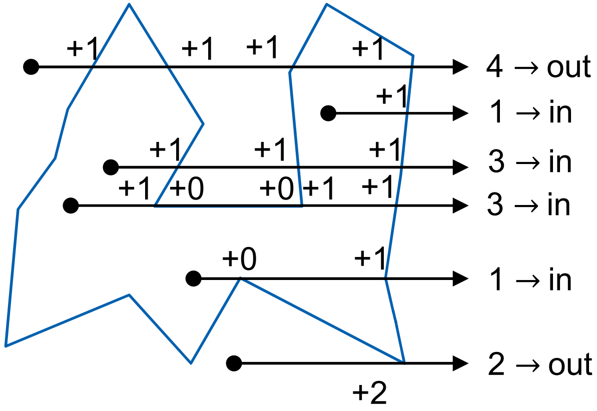


   - Singular cases must be handled!
     - Do not count horizontal line segments
     - Take non-horizontal segments as half-open (upper point not part of the segment)

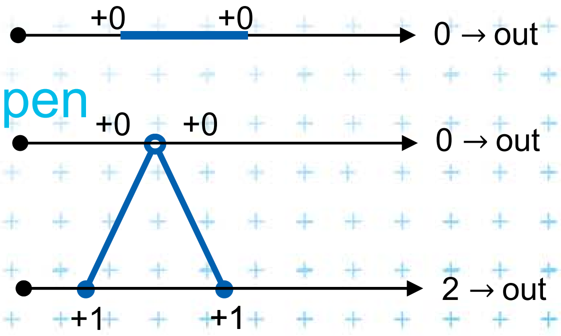


Point location in polygon by ray crossing

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

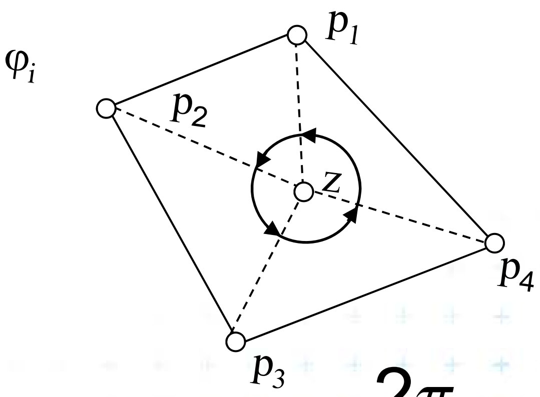


   - Singular cases must be handled!
     - Do not count horizontal line segments
     - Take non-horizontal segments as half-open (upper point not part of the segment)

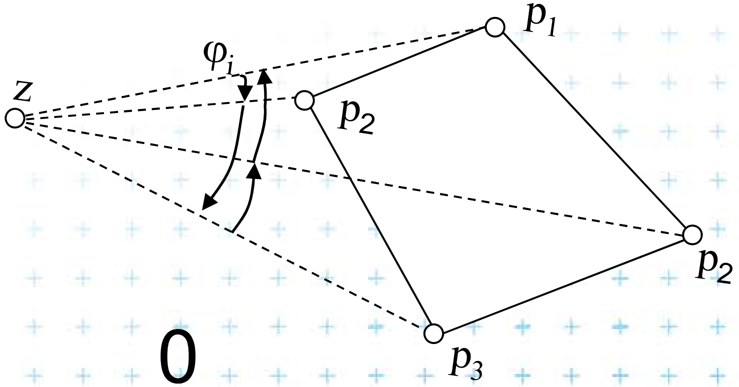


Point location in polygon

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



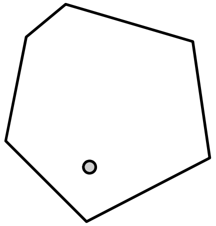
2π



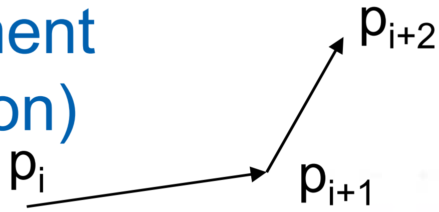
0

Point location in polygon

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



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



Convex polygon, non-collinear points

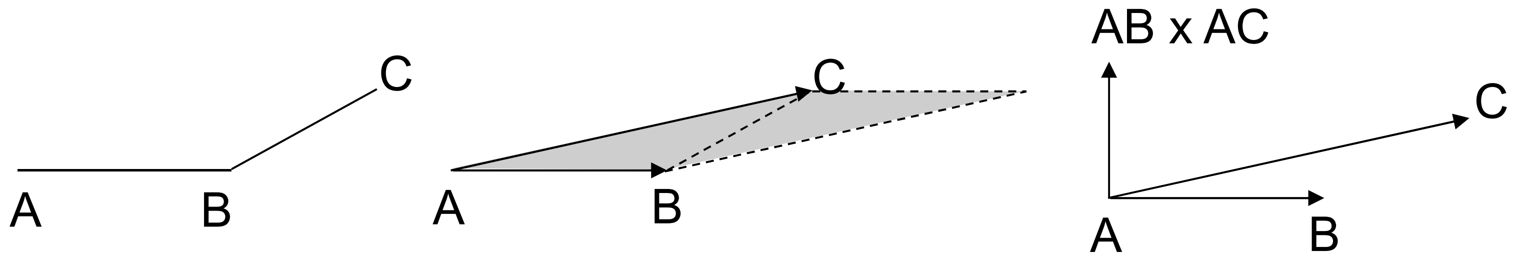
$p_i = [x_i, y_i, 1],\ \ p_{i+1} = [x_{i+1}, y_{i+1}, 1],\ \ p_{i+2} = [x_{i+2}, y_{i+2}, 1]$

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

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

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

Area of Triangle



Vector product of vectors AB x AC

= Vector perpendicular to both vectors AB and AC

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

  = area of parallelopid

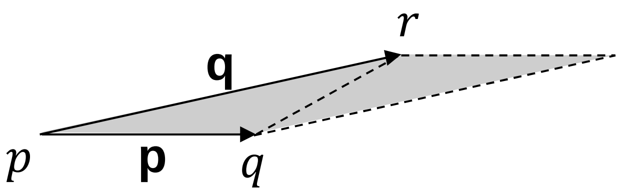
  = 2x area *T* of triangle ABC

Area of Triangle

$$T = \frac{1}{2}|\mathbf{p} \times \mathbf{q}|$$

$$\mathbf{p} = q - p$$

$$\mathbf{q} = r - p$$



$$2T = \mathbf{p}_x\mathbf{q}_y - \mathbf{p}_y\mathbf{q}_x$$ using vector product $\mathbf{p} \times \mathbf{q}$

$$2T = \begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix}$$ using coordinates of points

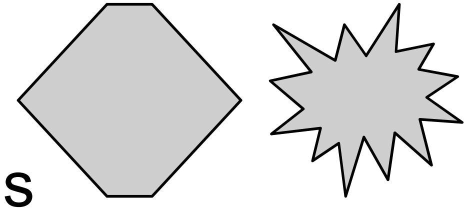
Orientation is computed as $\mathrm{sign}(2\mathrm{T}) =$

$$= \mathrm{sign}(p_xq_y + q_xr_y + r_xp_y - p_xr_y - q_xp_y - r_xq_y)$$

$$= \mathrm{sign}\left((q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)\right)$$ pivot p

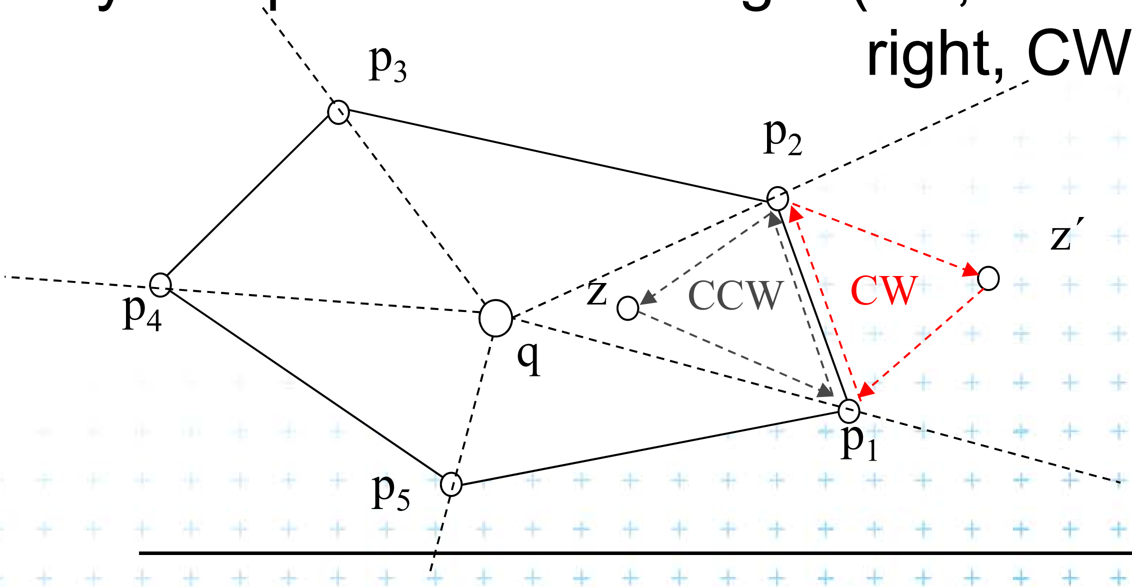
Point location in polygon

4. Binary search in angles



Works for convex and star-shaped polygons

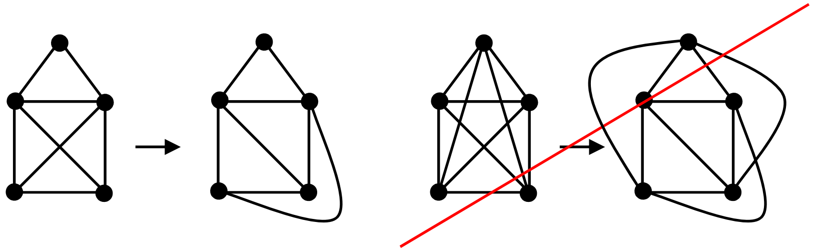
1. Choose any point *q* inside / in the polygon core
2. *q* forms wedges with polygon edges
3. Binary search of wedge výseč based on angle
4. Finaly compare with one edge (left, CCW => in, right, CW => out)



Planar graph

Planar graph U=set of nodes, H=set of arcs

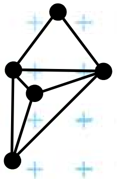
= Graph G = (U,H) is planar, if it can be embedded into plane without crossings



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

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

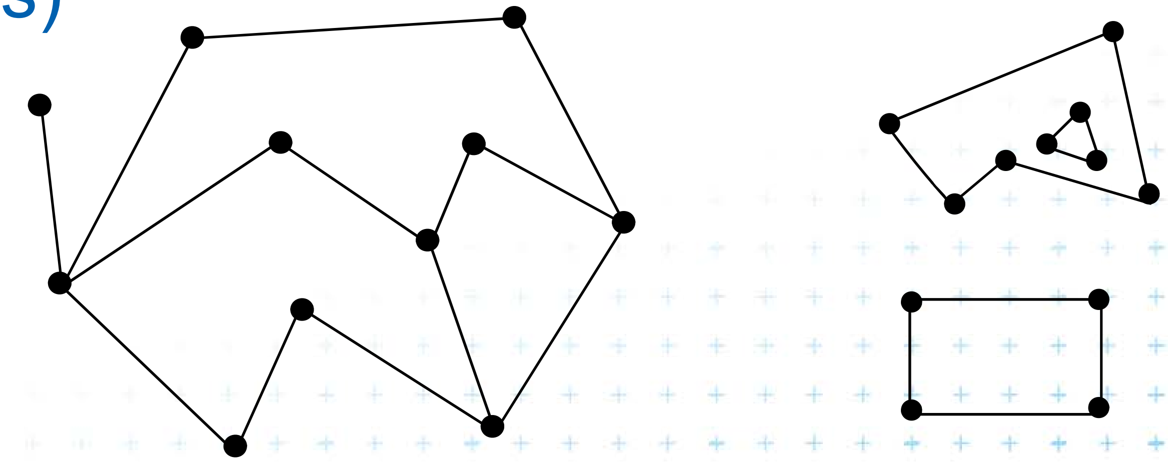
Every planar graph can be embedded in such a way that arcs map to straight line segments [Fáry 1948]



Planar subdivision

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

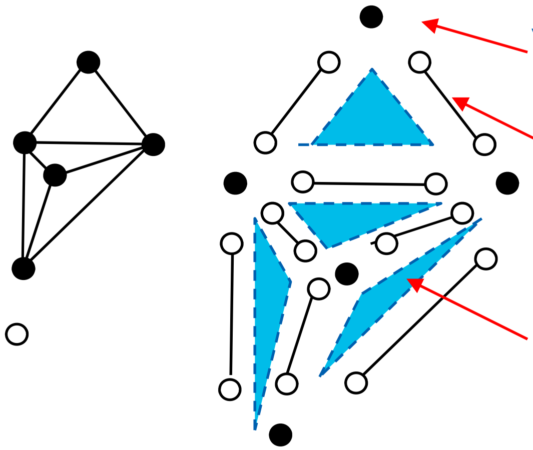
- (embedding of a planar graph in the plane such that its arcs are mapped into straight line segments)



connected

disconnected

Planar subdivision



Vertex = embedding of graph node

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

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

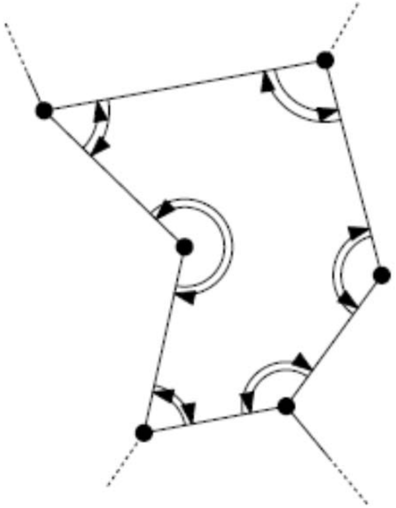
Complexity (size) of a subdivision = sum of number of vertices +
+ number of edges +
+ number of faces it consists of

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

DCEL = Double Connected Edge List

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

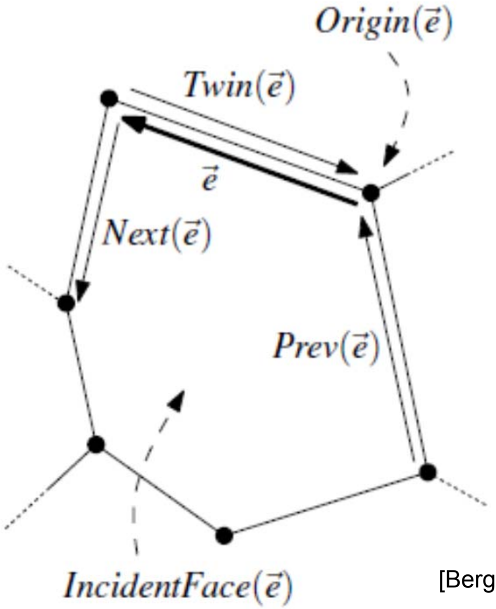
Walk around boundary of a given face



[Berg]

Pointers to next and prev edge

Get incident face

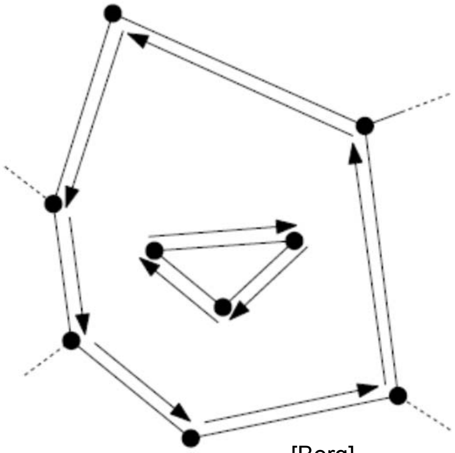


[Berg]

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

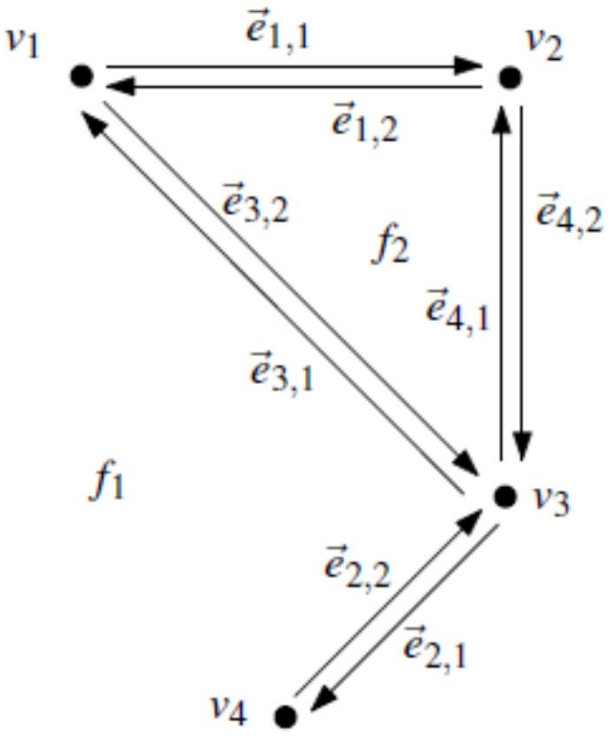
DCEL = Double Connected Edge List

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

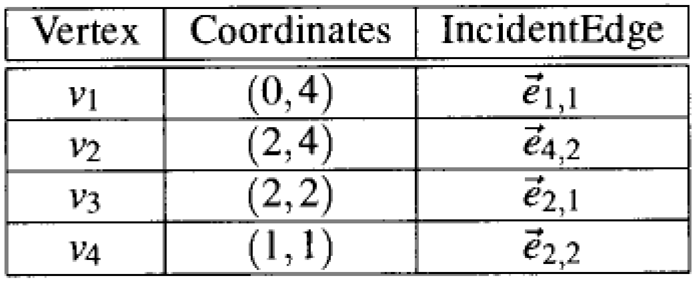


[Berg]

DCEL = Double Connected Edge List

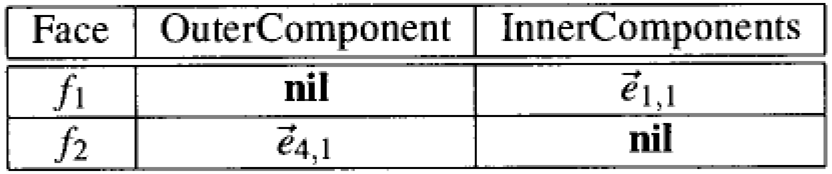


G



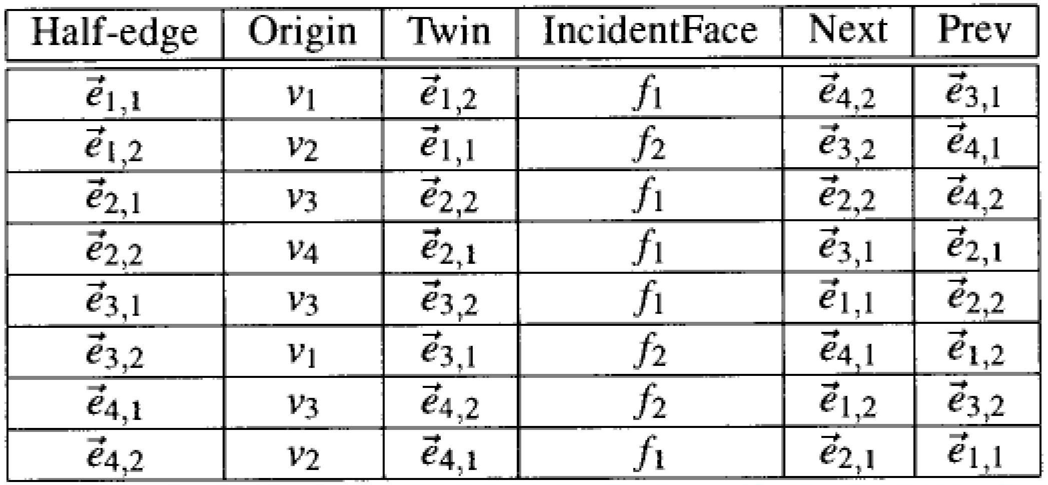
Vertex	Coordinates	IncidentEdge
v_1	$(0,4)$	$\vec{e}_{1,1}$
v_2	$(2,4)$	$\vec{e}_{4,2}$
v_3	$(2,2)$	$\vec{e}_{2,1}$
v_4	$(1,1)$	$\vec{e}_{2,2}$

T



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

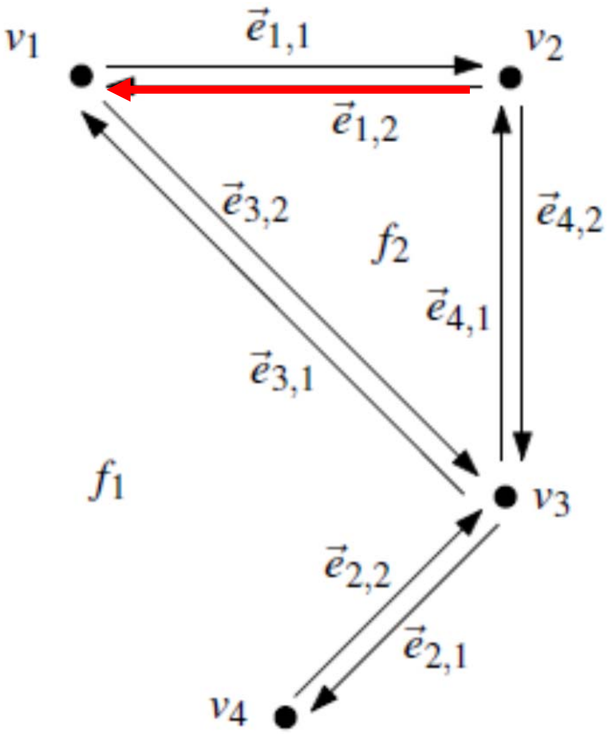
T



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

[Berg]

DCEL = Double Connected Edge List



G

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

T

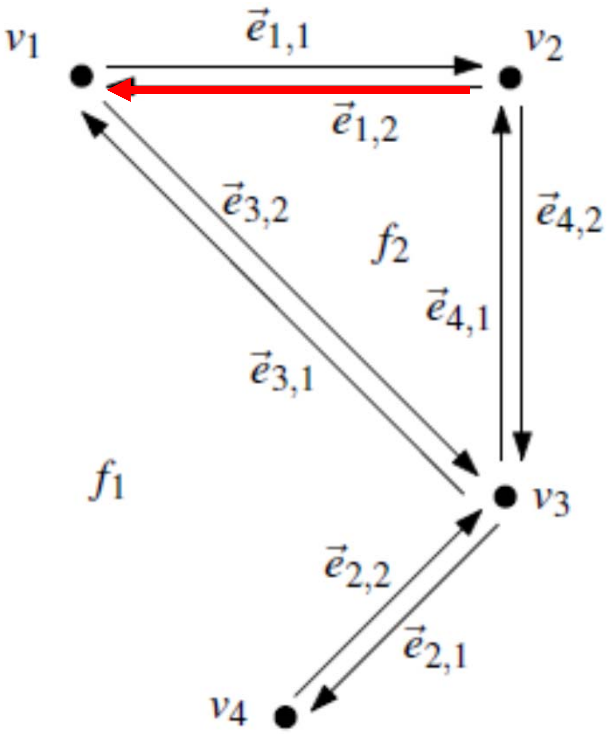
Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

T

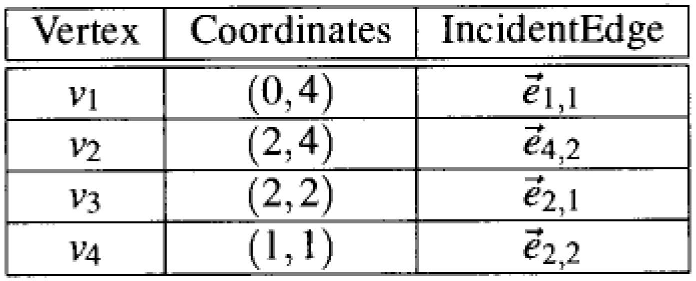
Half-edge	Origin	Twin	IncidentFace	Next	Prev
$\vec{e}_{1,1}$	v_1	$\vec{e}_{1,2}$	f_1	$\vec{e}_{4,2}$	$\vec{e}_{3,1}$
$\vec{e}_{1,2}$	v_2	$\vec{e}_{1,1}$	f_2	$\vec{e}_{3,2}$	$\vec{e}_{4,1}$
$\vec{e}_{2,1}$	v_3	$\vec{e}_{2,2}$	f_1	$\vec{e}_{2,2}$	$\vec{e}_{4,2}$
$\vec{e}_{2,2}$	v_4	$\vec{e}_{2,1}$	f_1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

[Berg]

DCEL = Double Connected Edge List

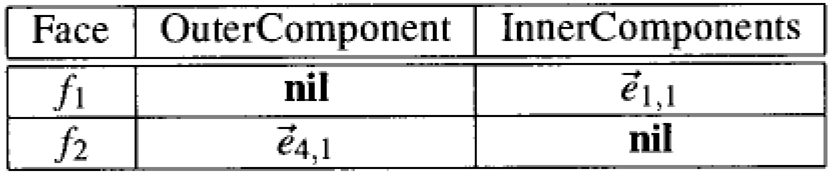


G



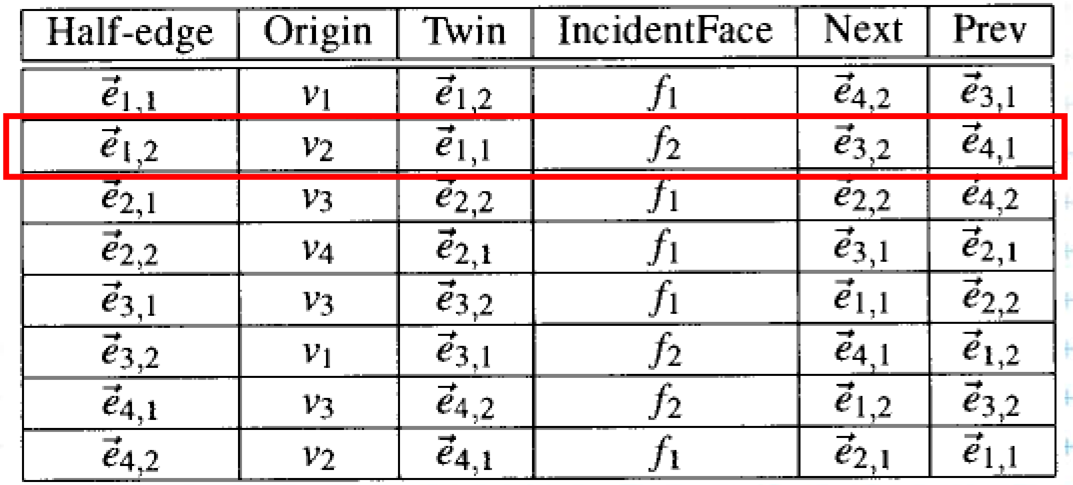
Vertex	Coordinates	IncidentEdge
v_1	$(0,4)$	$\vec{e}_{1,1}$
v_2	$(2,4)$	$\vec{e}_{4,2}$
v_3	$(2,2)$	$\vec{e}_{2,1}$
v_4	$(1,1)$	$\vec{e}_{2,2}$

T



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

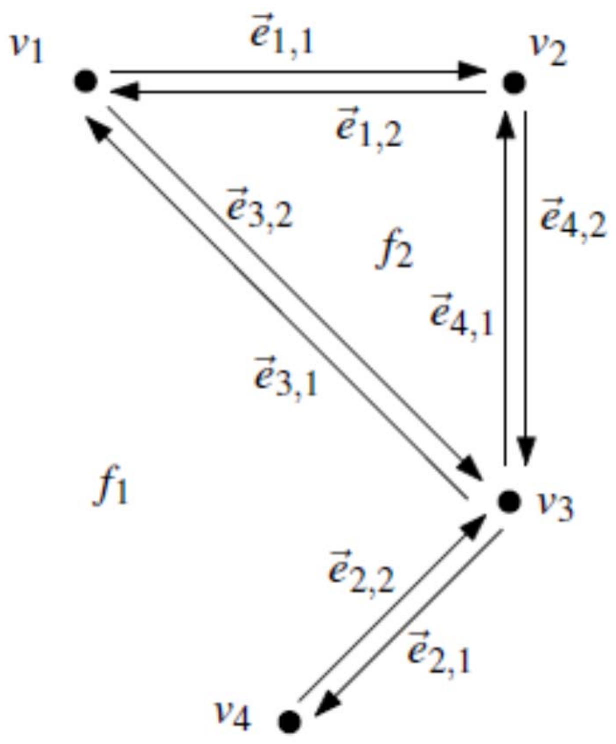
T



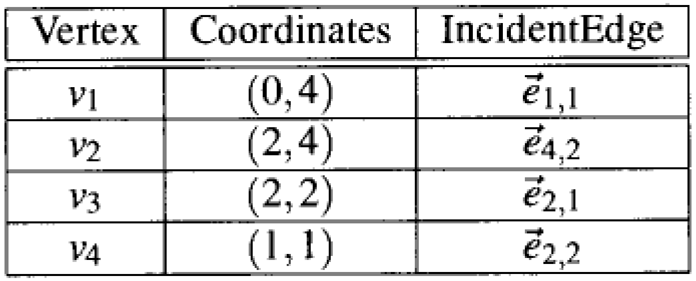
Half-edge	Origin	Twin	IncidentFace	Next	Prev
$\vec{e}_{1,1}$	v_1	$\vec{e}_{1,2}$	f_1	$\vec{e}_{4,2}$	$\vec{e}_{3,1}$
$\vec{e}_{1,2}$	v_2	$\vec{e}_{1,1}$	f_2	$\vec{e}_{3,2}$	$\vec{e}_{4,1}$
$\vec{e}_{2,1}$	v_3	$\vec{e}_{2,2}$	f_1	$\vec{e}_{2,2}$	$\vec{e}_{4,2}$
$\vec{e}_{2,2}$	v_4	$\vec{e}_{2,1}$	f_1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

[Berg]

DCEL = Double Connected Edge List

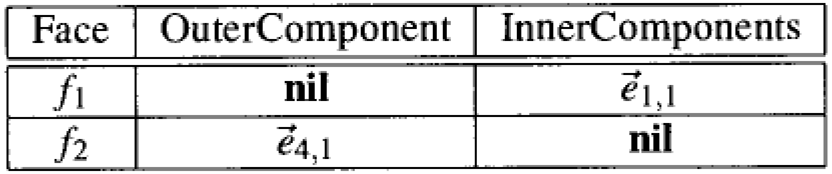


G



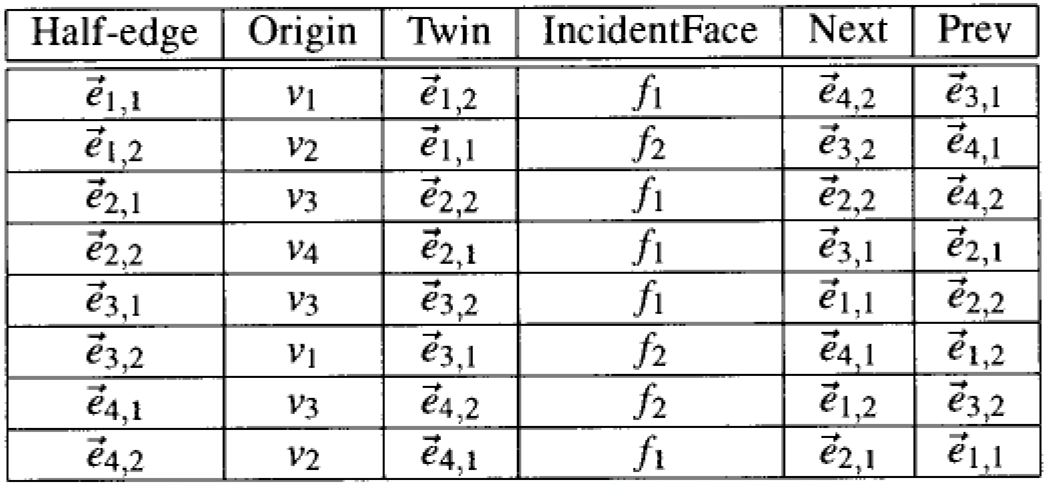
Vertex	Coordinates	IncidentEdge
v_1	$(0,4)$	$\vec{e}_{1,1}$
v_2	$(2,4)$	$\vec{e}_{4,2}$
v_3	$(2,2)$	$\vec{e}_{2,1}$
v_4	$(1,1)$	$\vec{e}_{2,2}$

T



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

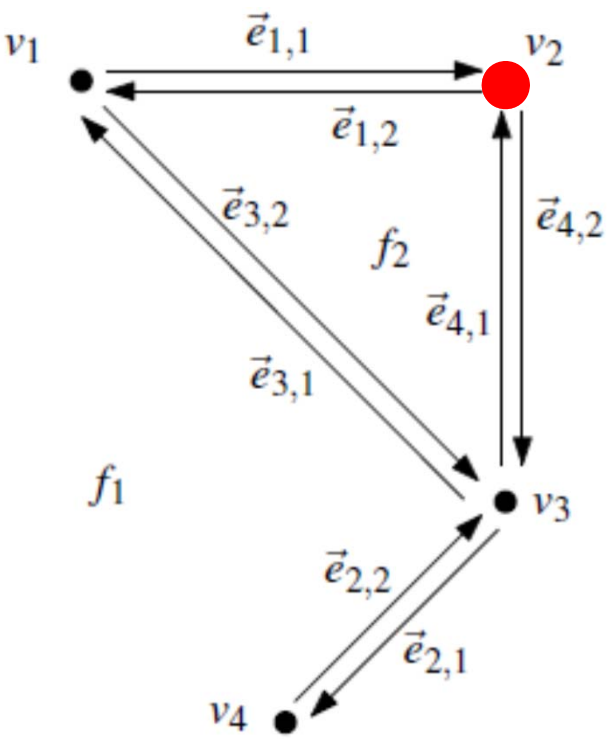
T



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

[Berg]

DCEL = Double Connected Edge List



G

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

T

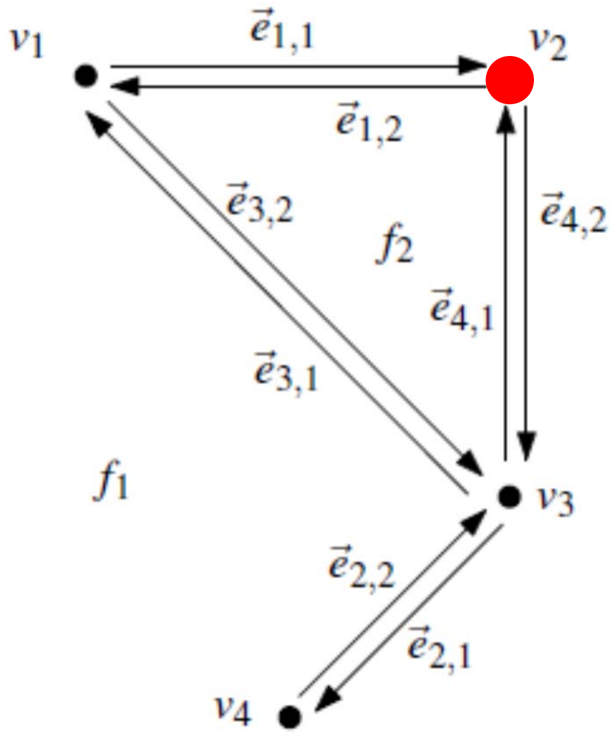
Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

T

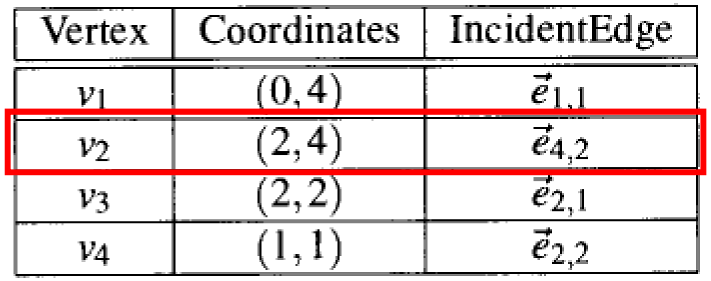
Half-edge	Origin	Twin	IncidentFace	Next	Prev
$\vec{e}_{1,1}$	v_1	$\vec{e}_{1,2}$	f_1	$\vec{e}_{4,2}$	$\vec{e}_{3,1}$
$\vec{e}_{1,2}$	v_2	$\vec{e}_{1,1}$	f_2	$\vec{e}_{3,2}$	$\vec{e}_{4,1}$
$\vec{e}_{2,1}$	v_3	$\vec{e}_{2,2}$	f_1	$\vec{e}_{2,2}$	$\vec{e}_{4,2}$
$\vec{e}_{2,2}$	v_4	$\vec{e}_{2,1}$	f_1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

[Berg]

DCEL = Double Connected Edge List

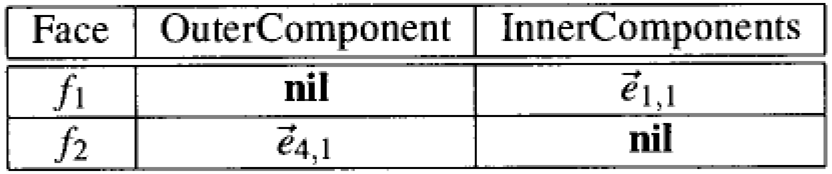


G



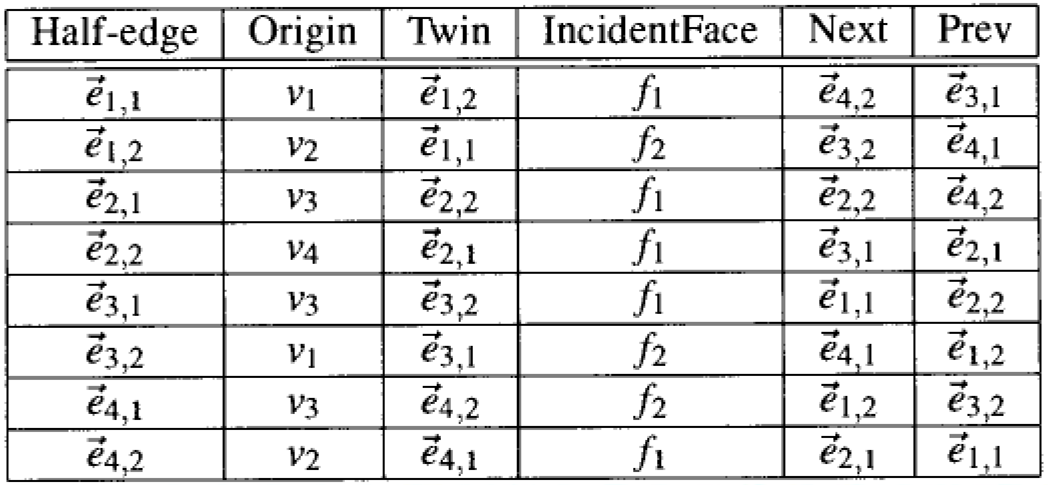
Vertex	Coordinates	IncidentEdge
v_1	$(0,4)$	$\vec{e}_{1,1}$
v_2	$(2,4)$	$\vec{e}_{4,2}$
v_3	$(2,2)$	$\vec{e}_{2,1}$
v_4	$(1,1)$	$\vec{e}_{2,2}$

T



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

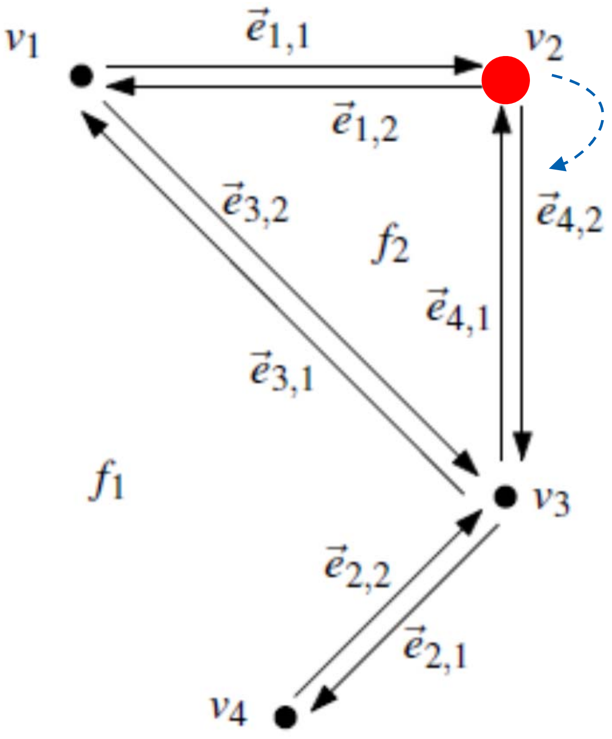
T



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

[Berg]

DCEL = Double Connected Edge List



G

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

One of edges

T

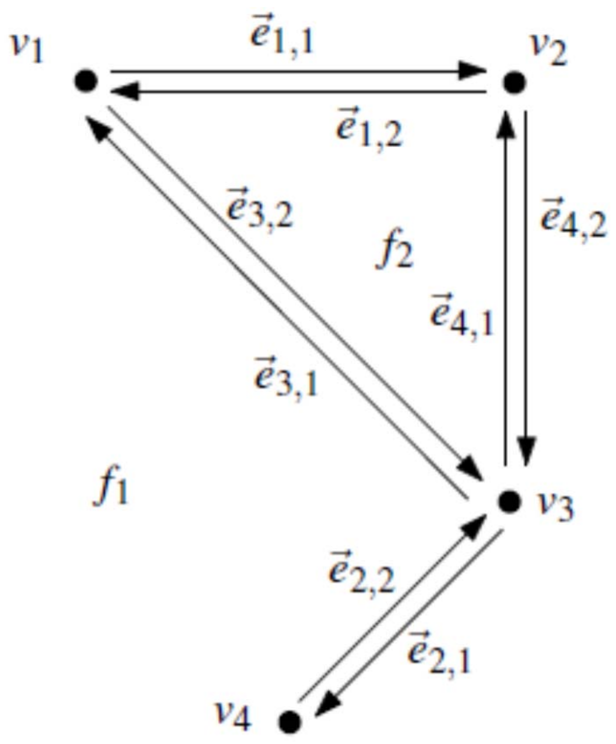
Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

T

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

[Berg]

DCEL = Double Connected Edge List



G

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

One of edges

T

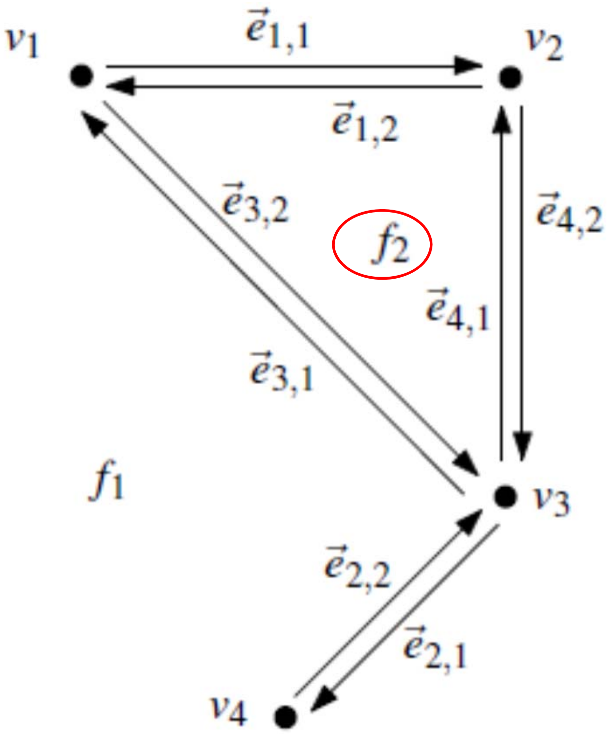
Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

T

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

[Berg]

DCEL = Double Connected Edge List



G

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

One of edges

T

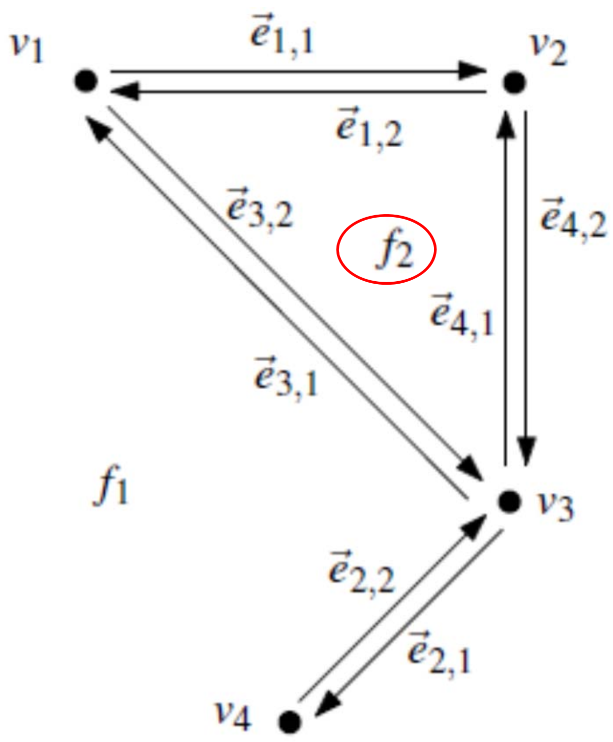
Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

T

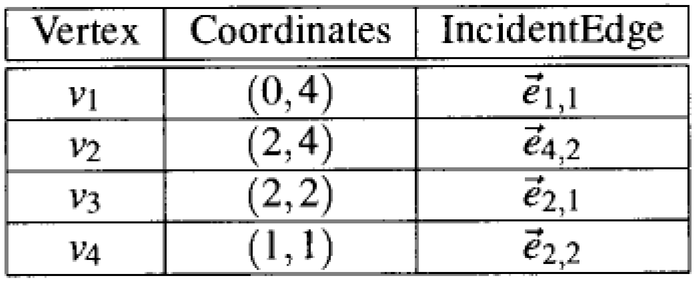
Half-edge	Origin	Twin	IncidentFace	Next	Prev
$\vec{e}_{1,1}$	v_1	$\vec{e}_{1,2}$	f_1	$\vec{e}_{4,2}$	$\vec{e}_{3,1}$
$\vec{e}_{1,2}$	v_2	$\vec{e}_{1,1}$	f_2	$\vec{e}_{3,2}$	$\vec{e}_{4,1}$
$\vec{e}_{2,1}$	v_3	$\vec{e}_{2,2}$	f_1	$\vec{e}_{2,2}$	$\vec{e}_{4,2}$
$\vec{e}_{2,2}$	v_4	$\vec{e}_{2,1}$	f_1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

[Berg]

DCEL = Double Connected Edge List



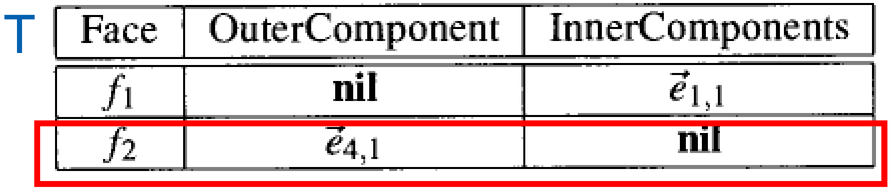
G



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

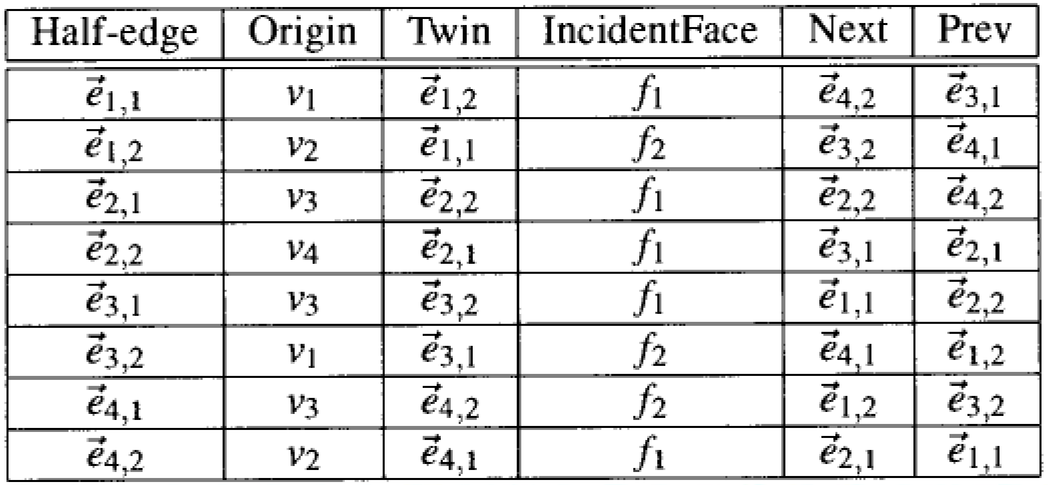
One of edges

T



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

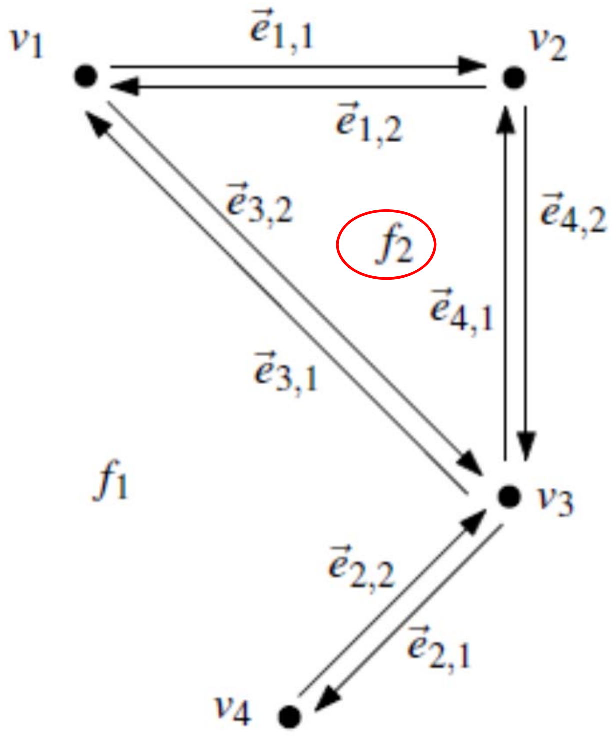
T

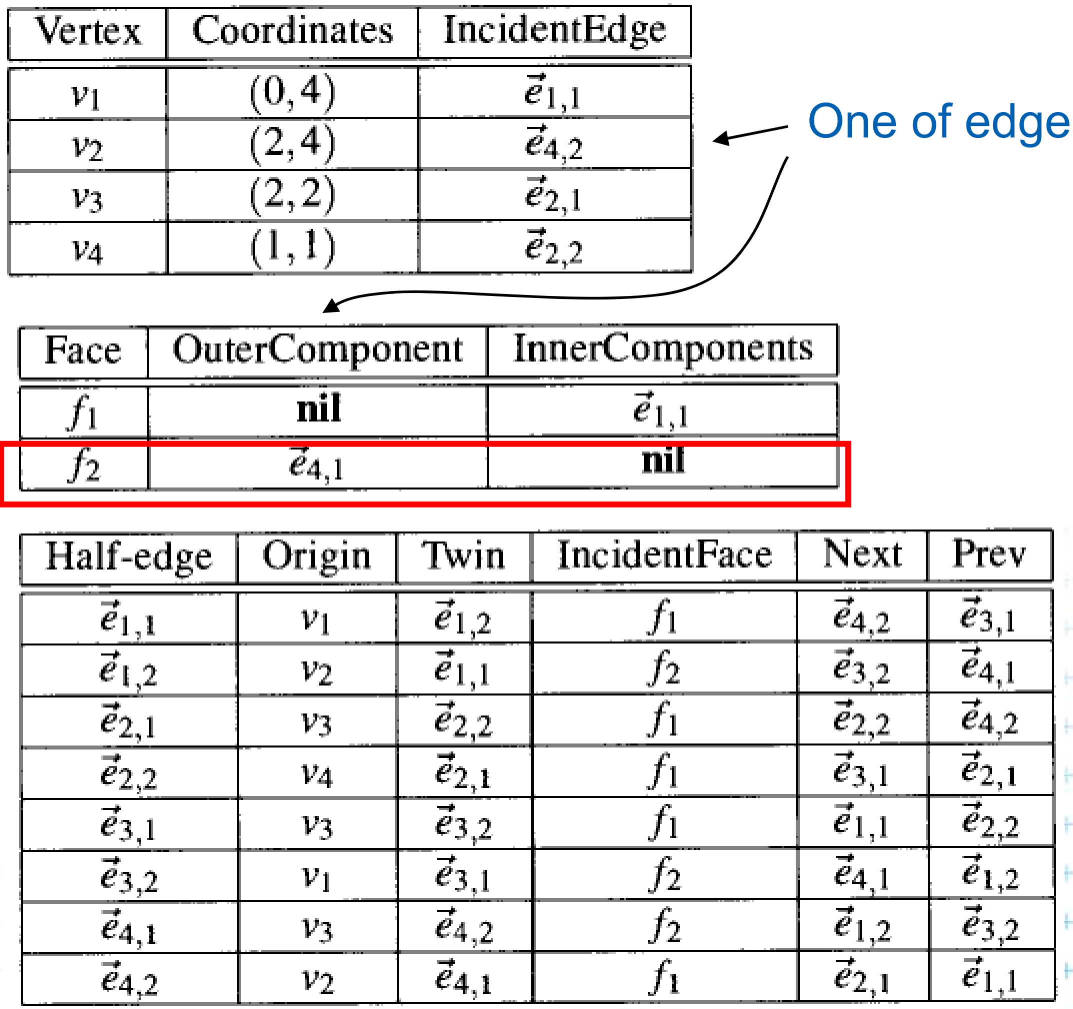


Half-edge	Origin	Twin	IncidentFace	Next	Prev
$\vec{e}_{1,1}$	v_1	$\vec{e}_{1,2}$	f_1	$\vec{e}_{4,2}$	$\vec{e}_{3,1}$
$\vec{e}_{1,2}$	v_2	$\vec{e}_{1,1}$	f_2	$\vec{e}_{3,2}$	$\vec{e}_{4,1}$
$\vec{e}_{2,1}$	v_3	$\vec{e}_{2,2}$	f_1	$\vec{e}_{2,2}$	$\vec{e}_{4,2}$
$\vec{e}_{2,2}$	v_4	$\vec{e}_{2,1}$	f_1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

[Berg]

DCEL = Double Connected Edge List





G

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

One of edges

T

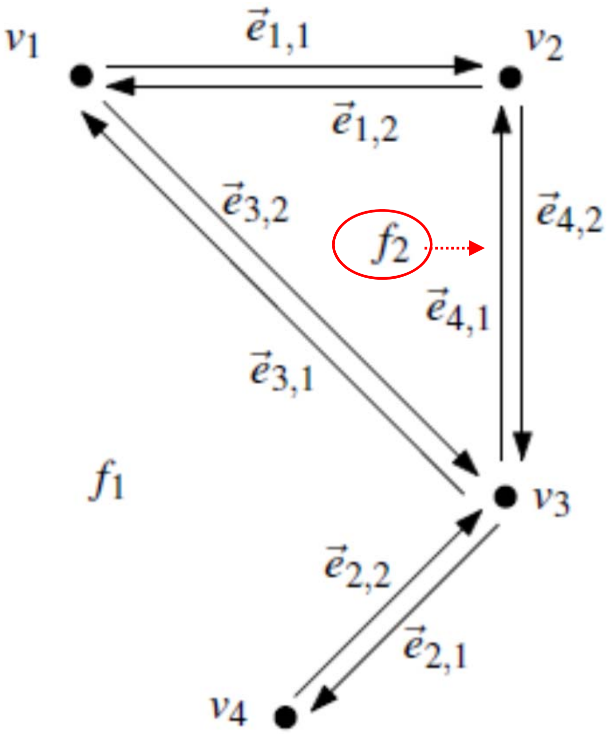
Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

T

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

[Berg]

DCEL = Double Connected Edge List



G

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

One of edges

T

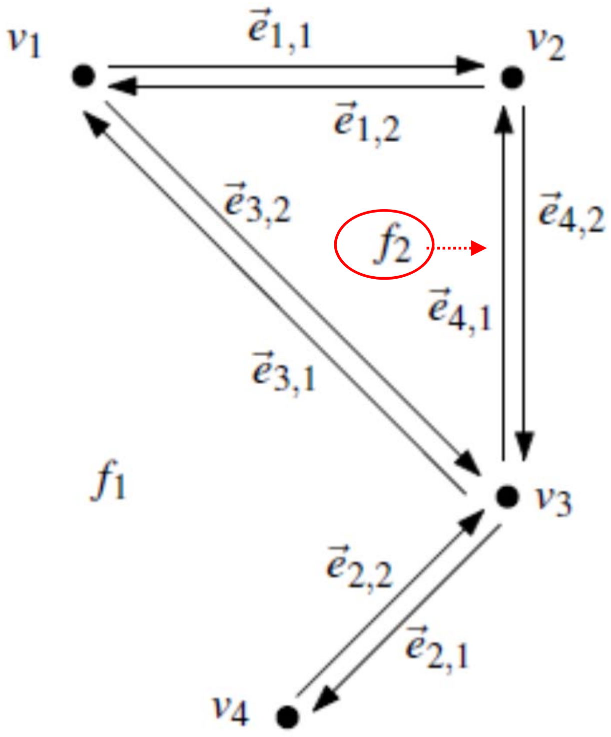
Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

T

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

[Berg]

DCEL = Double Connected Edge List



G

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

One of edges

List of holes

T

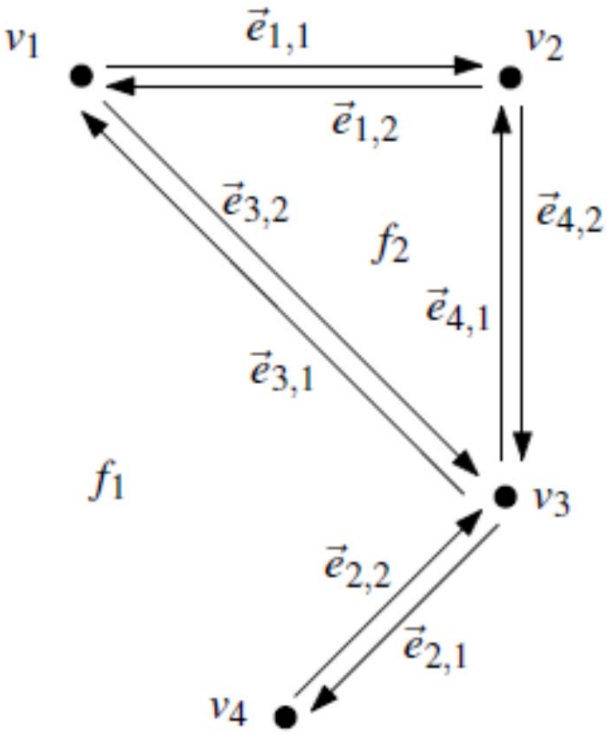
Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

T

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

[Berg]

DCEL = Double Connected Edge List



G

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

One of edges

List of holes

T

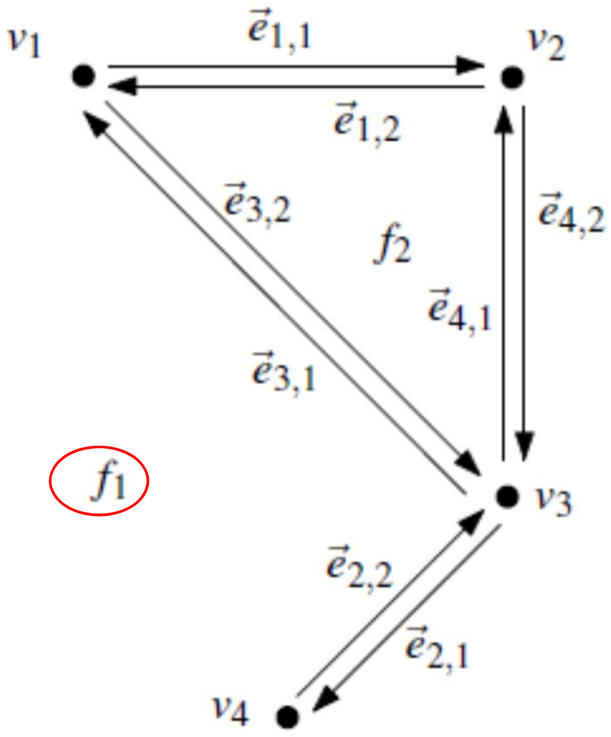
Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

T

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

[Berg]

DCEL = Double Connected Edge List



G

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

One of edges

List of holes

T

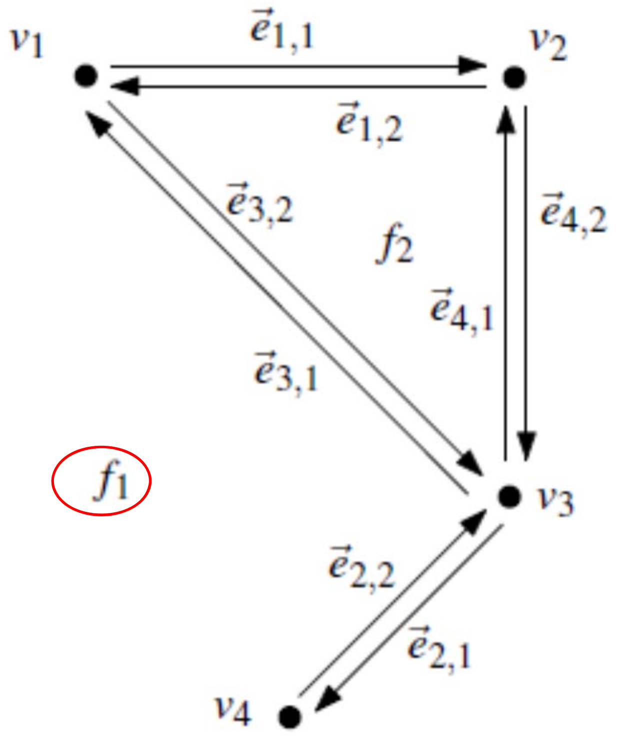
Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

T

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

[Berg]

DCEL = Double Connected Edge List



G

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

One of edges

List of holes

T

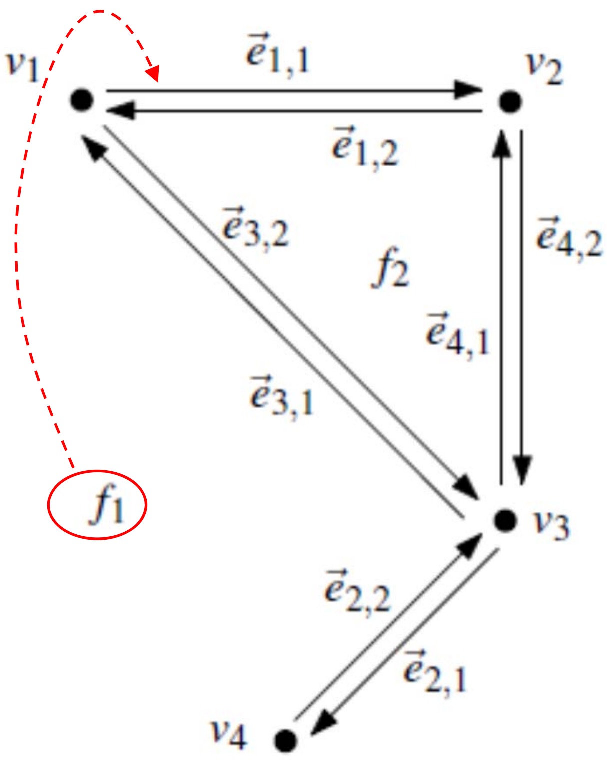
Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

T

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

[Berg]

DCEL = Double Connected Edge List



G

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

One of edges

List of holes

T

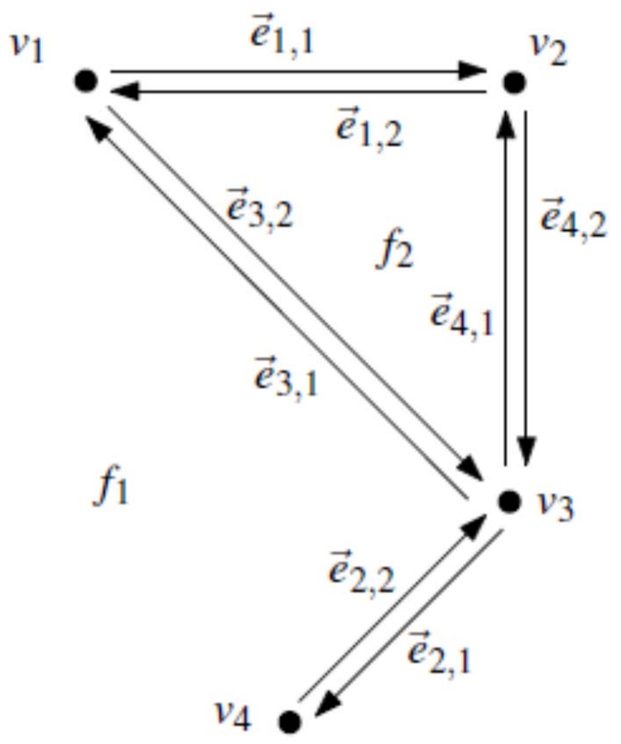
Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

T

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

[Berg]

DCEL = Double Connected Edge List



G

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

One of edges

List of holes

T

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

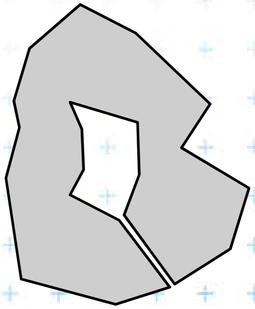
T

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

[Berg]

DCEL simplifications

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



Point location in planar subdivision

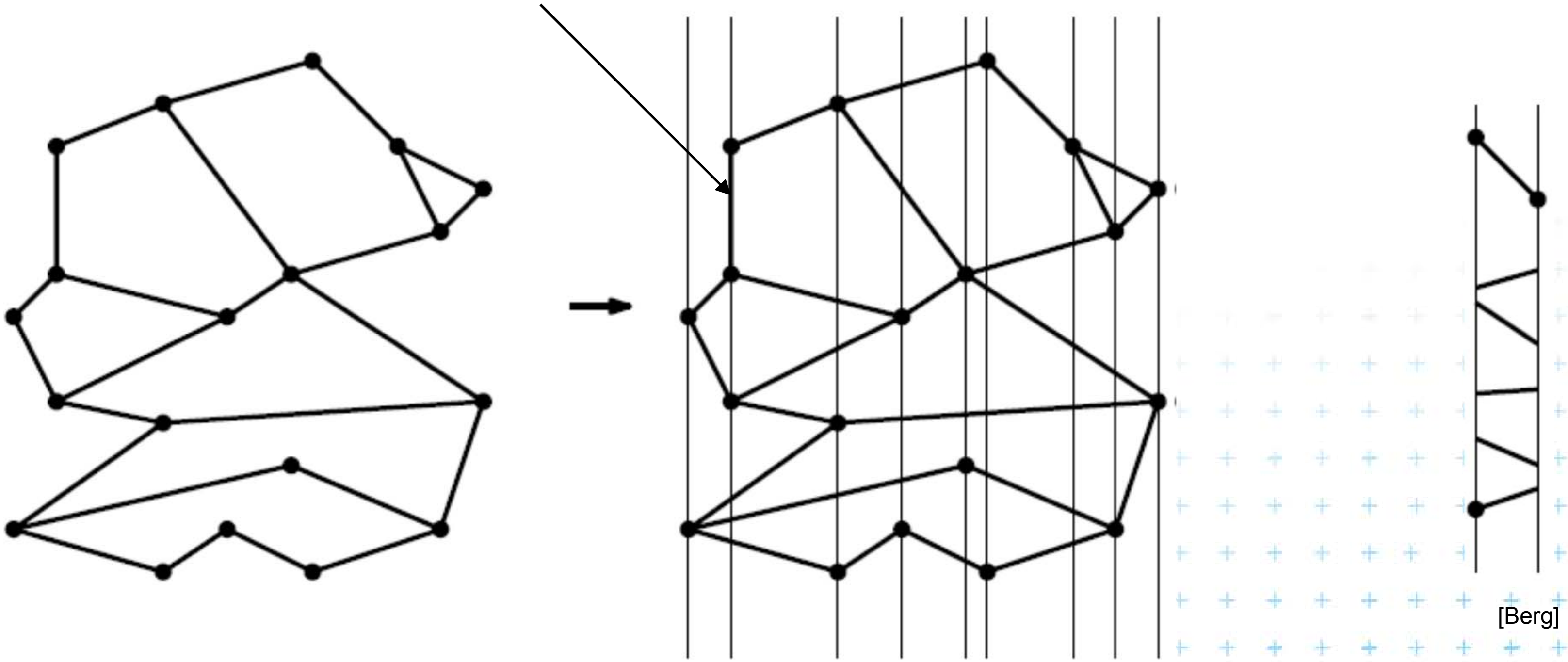
- Using special search structures
  an optimal algorithm can be made with
  - O(n) preprocessing,
  - O(n) memory and
  - O(log n) query time.
- Simpler methods

1. Slabs	O(log n) query, O(n^2) memory
2. monotone chain tree	O($\log^2$ n) query, O(n^2) memory
3. trapezoidal map	O(log n) query expected time O(n) expected memory

1. Vertical (horizontal) slabs

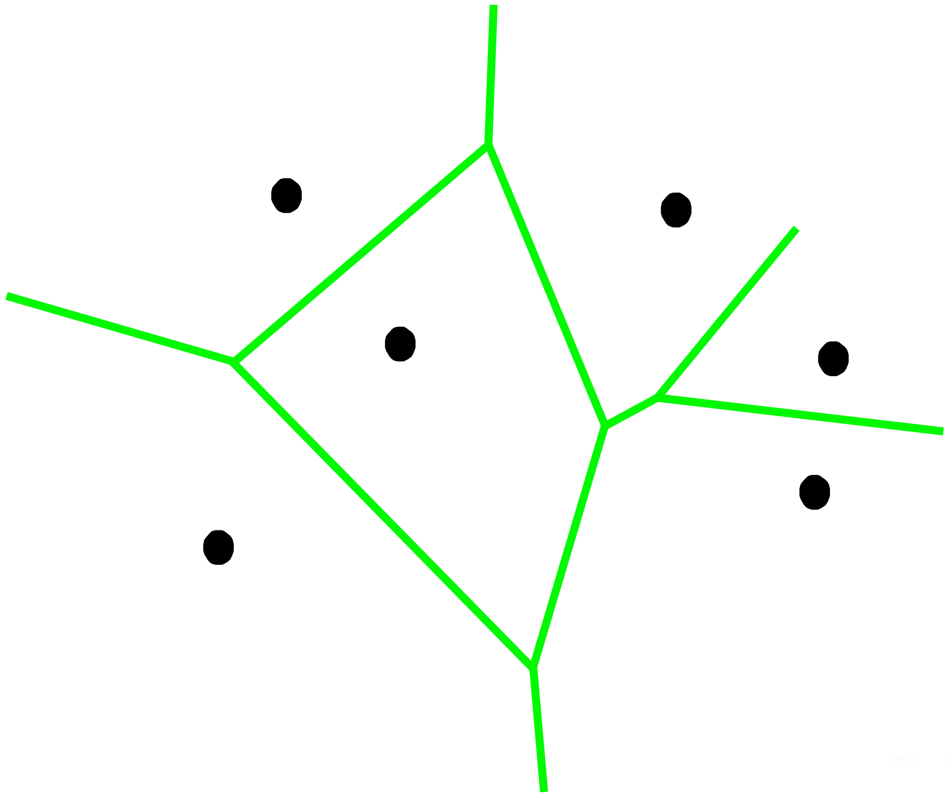
[Dobkin and Lipton, 1976]

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



[Berg]

Horizontal slabs example

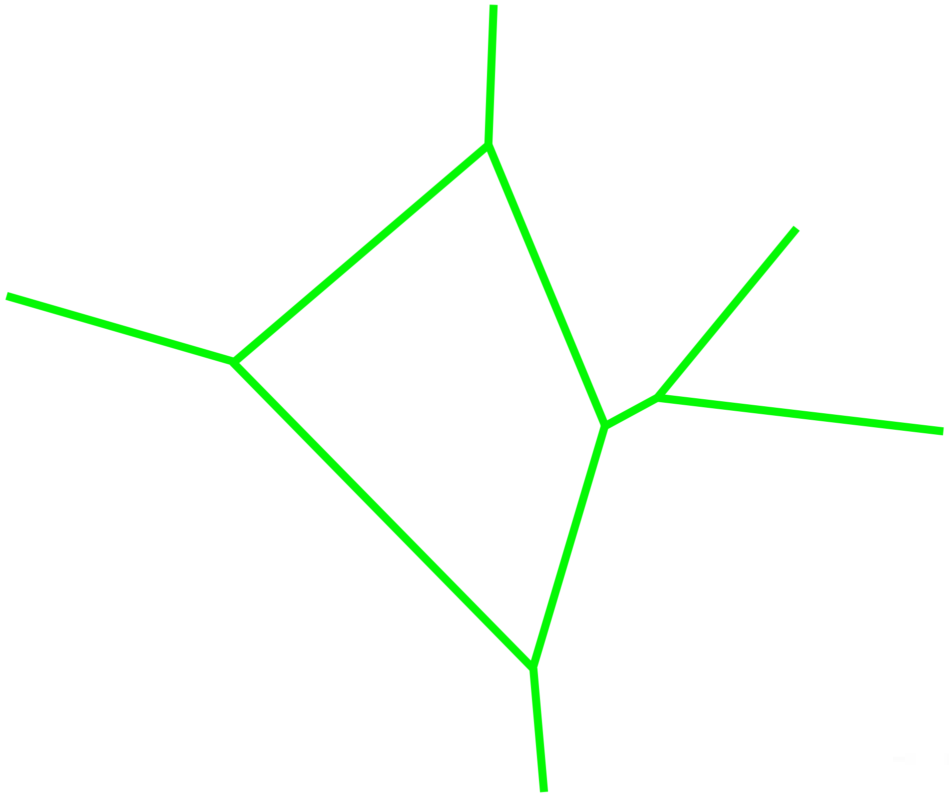


1. Find slab in T_y for y

T_x and T_y are arrays

2. Find slab part in T_x for x

Horizontal slabs example

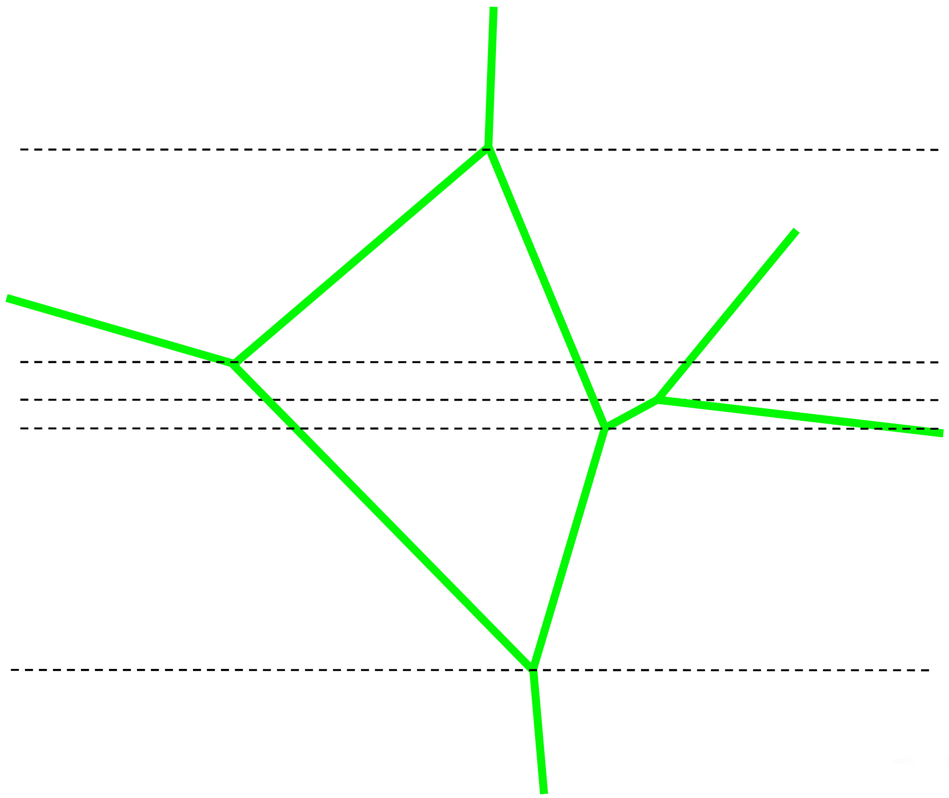


1. Find slab in T_y for y

T_x and T_y are arrays

2. Find slab part in T_x for x

Horizontal slabs example

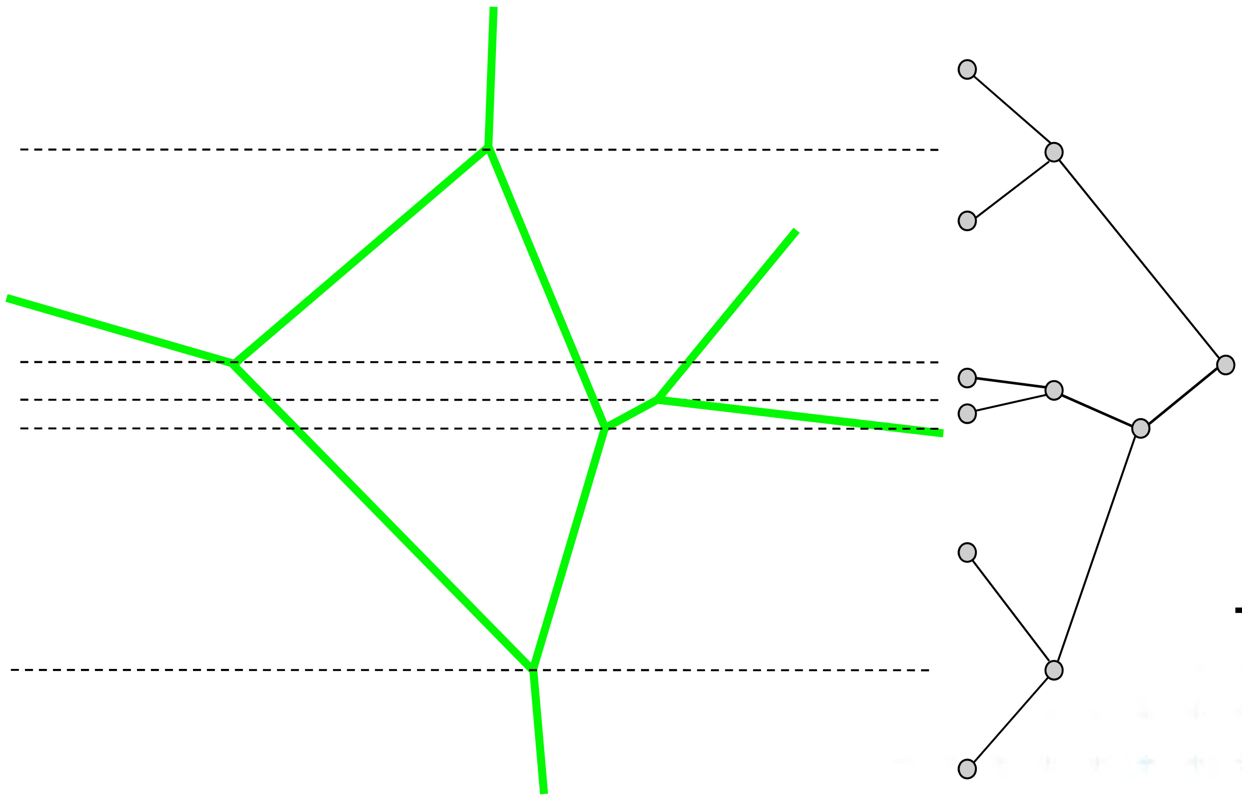


1. Find slab in T_y for y

T_x and T_y are arrays

2. Find slab part in T_x for x

Horizontal slabs example

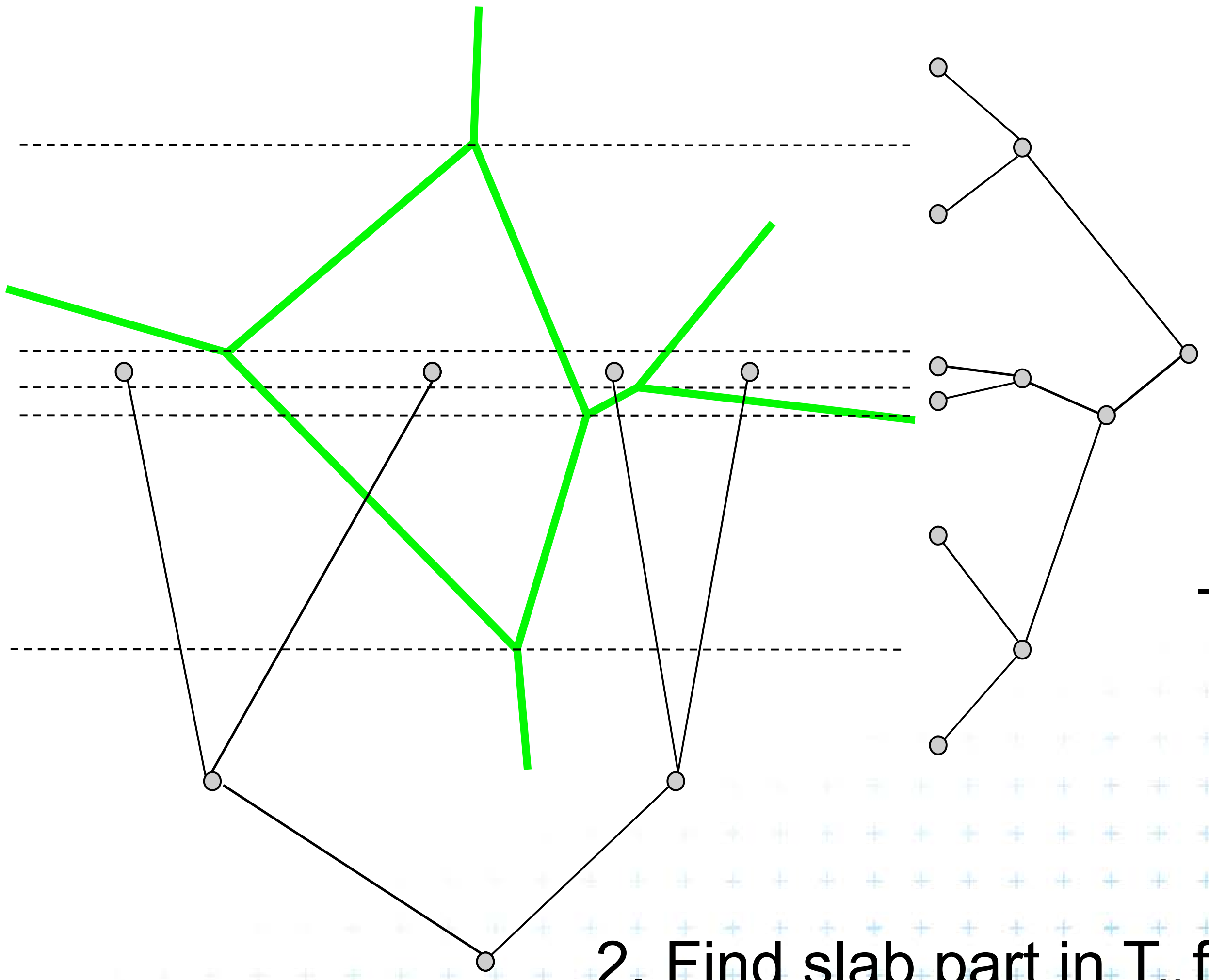


1. Find slab in T_y for y

T_x and T_y are arrays

2. Find slab part in T_x for x

Horizontal slabs example

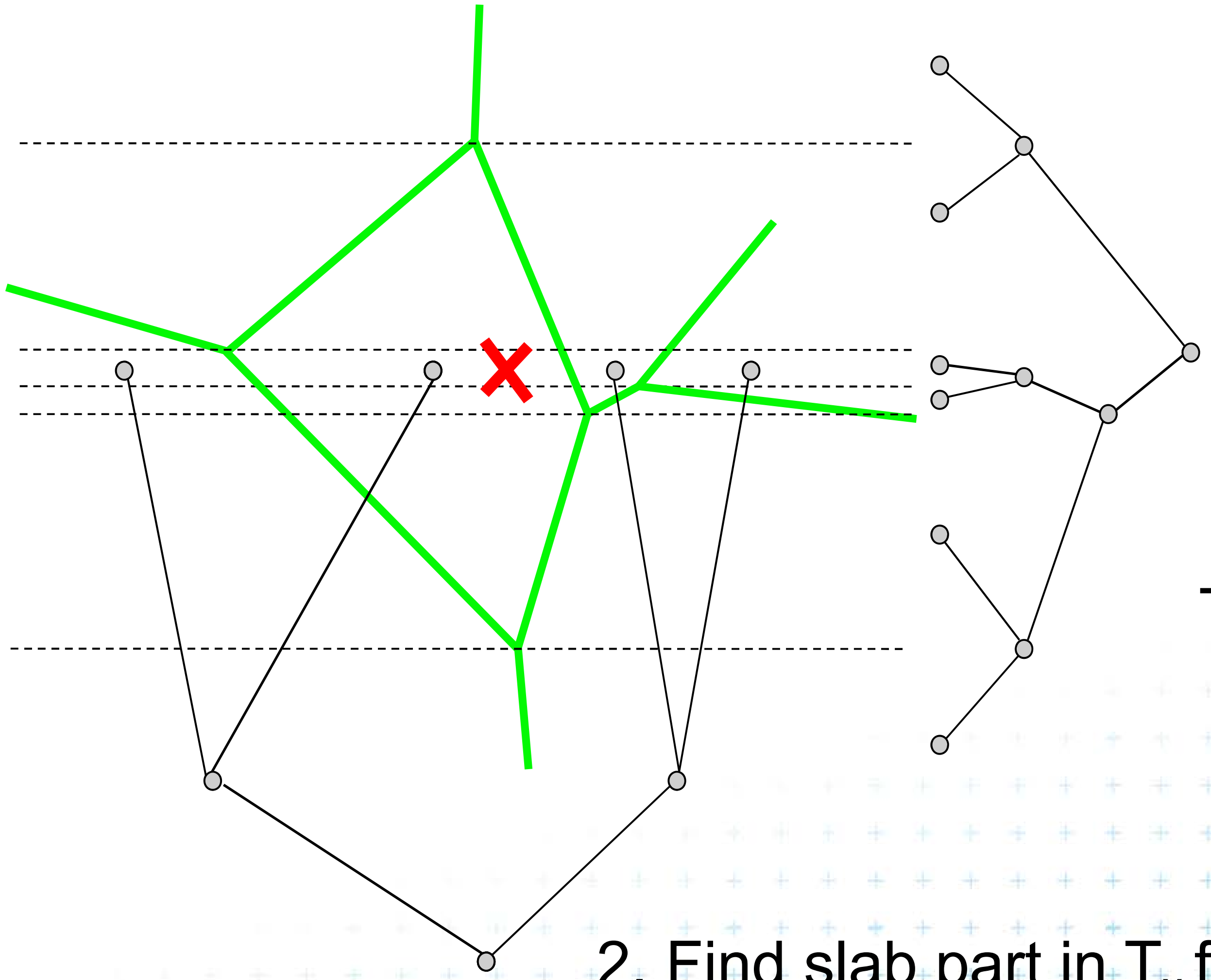


1. Find slab in T_y for y

T_x and T_y are arrays

2. Find slab part in T_x for x

Horizontal slabs example

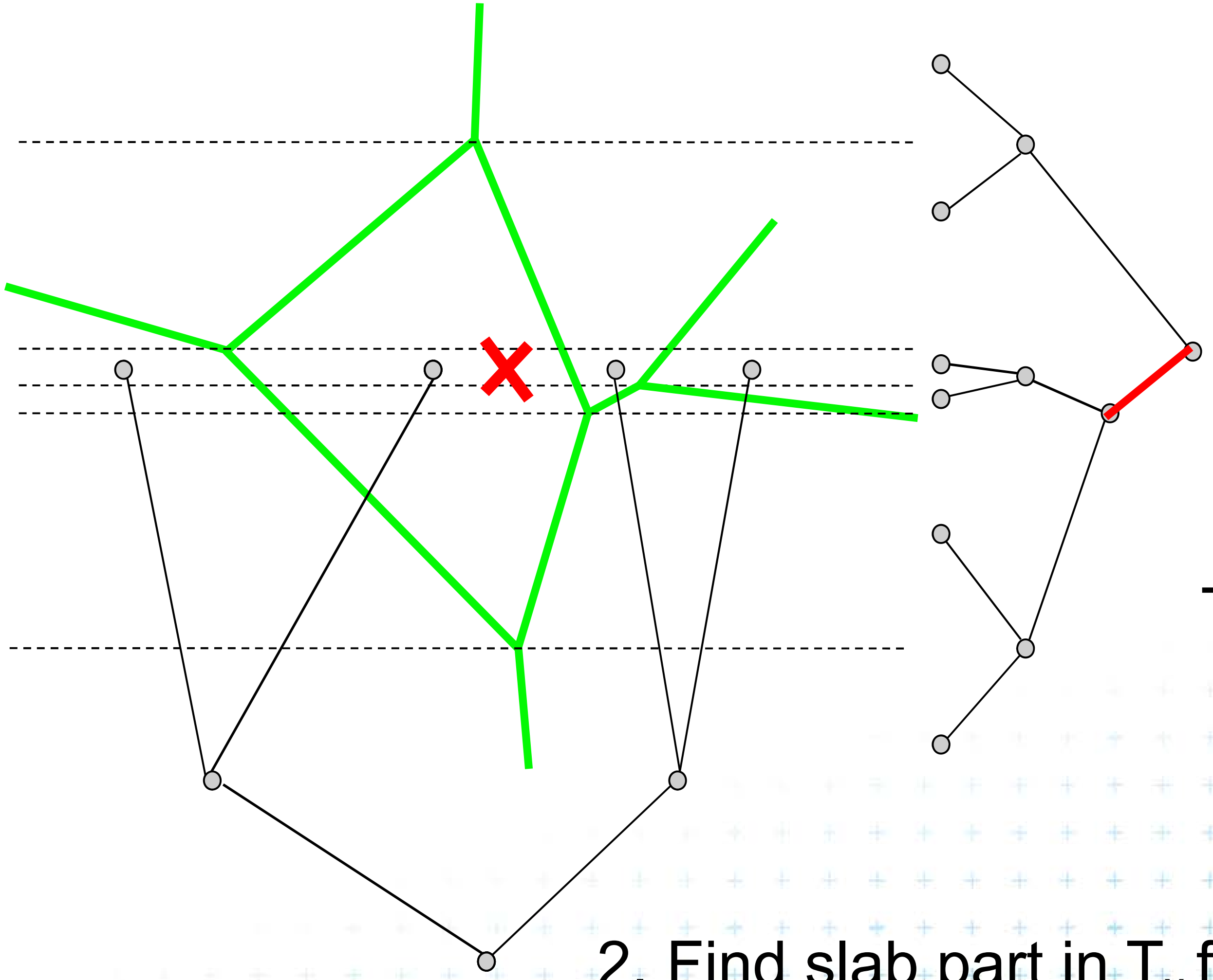


1. Find slab in T_y for y

T_x and T_y are arrays

2. Find slab part in T_x for x

Horizontal slabs example

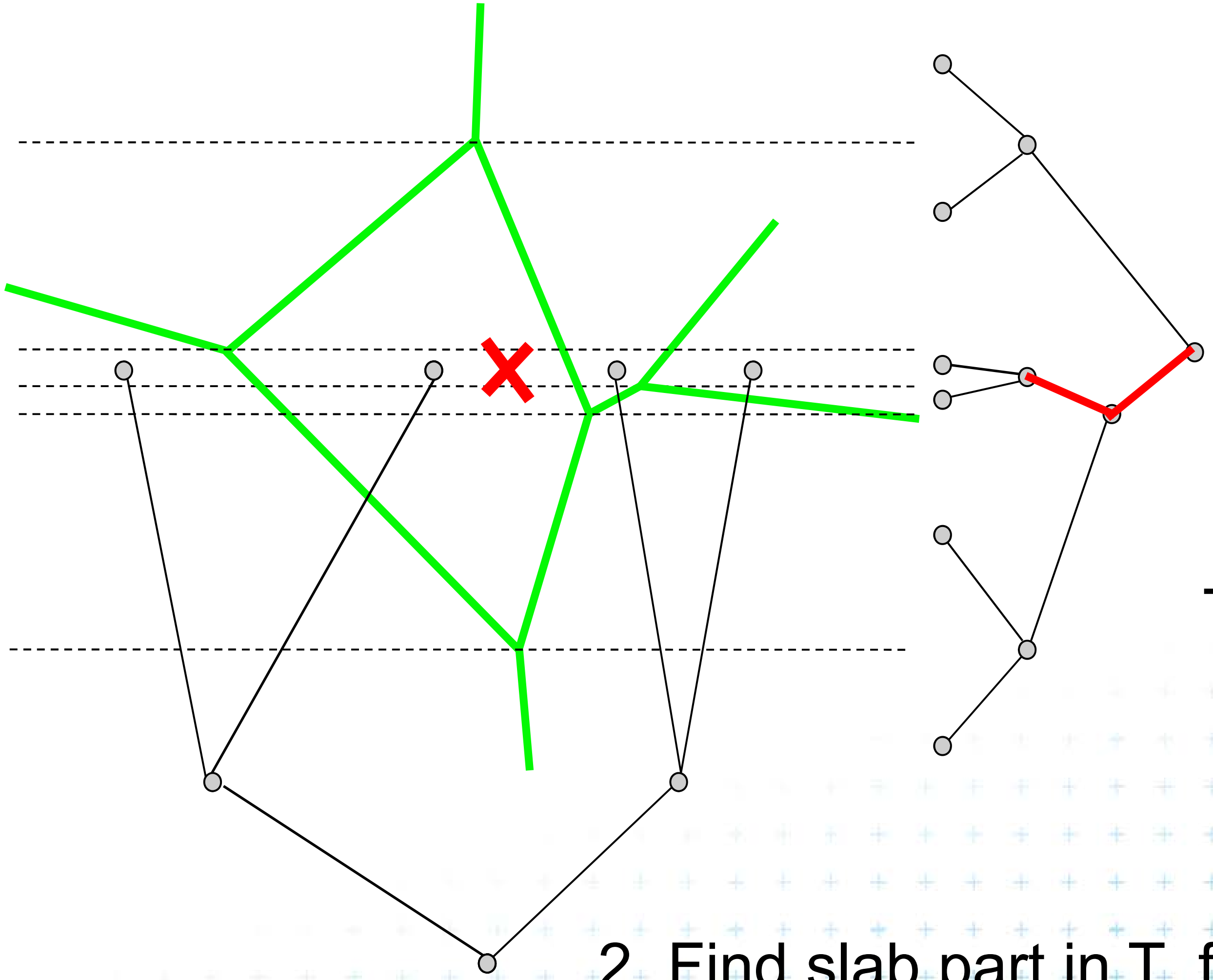


1. Find slab in T_y for y

T_x and T_y are arrays

2. Find slab part in T_x for x

Horizontal slabs example

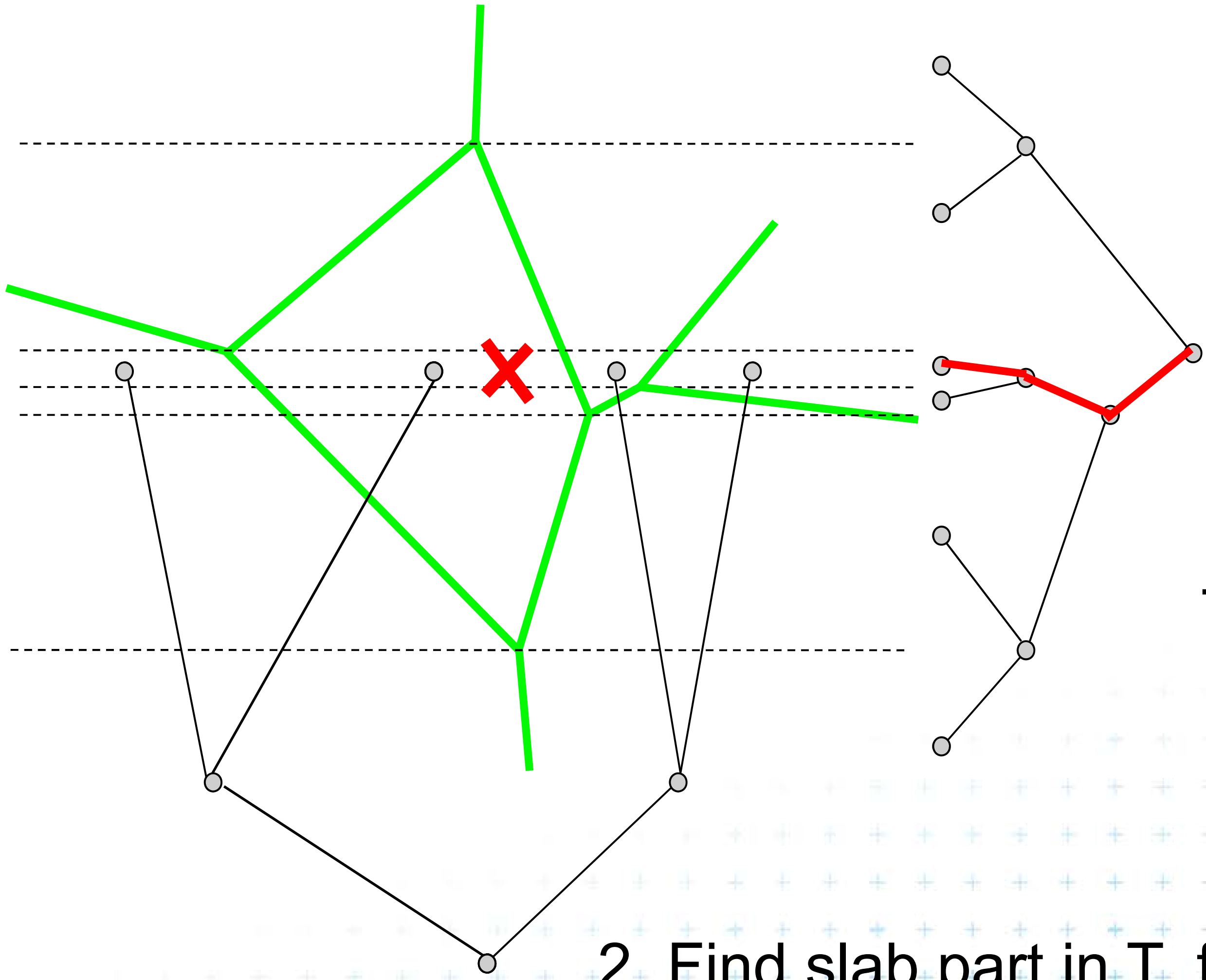


1. Find slab in T_y for y

T_x and T_y are arrays

2. Find slab part in T_x for x

Horizontal slabs example

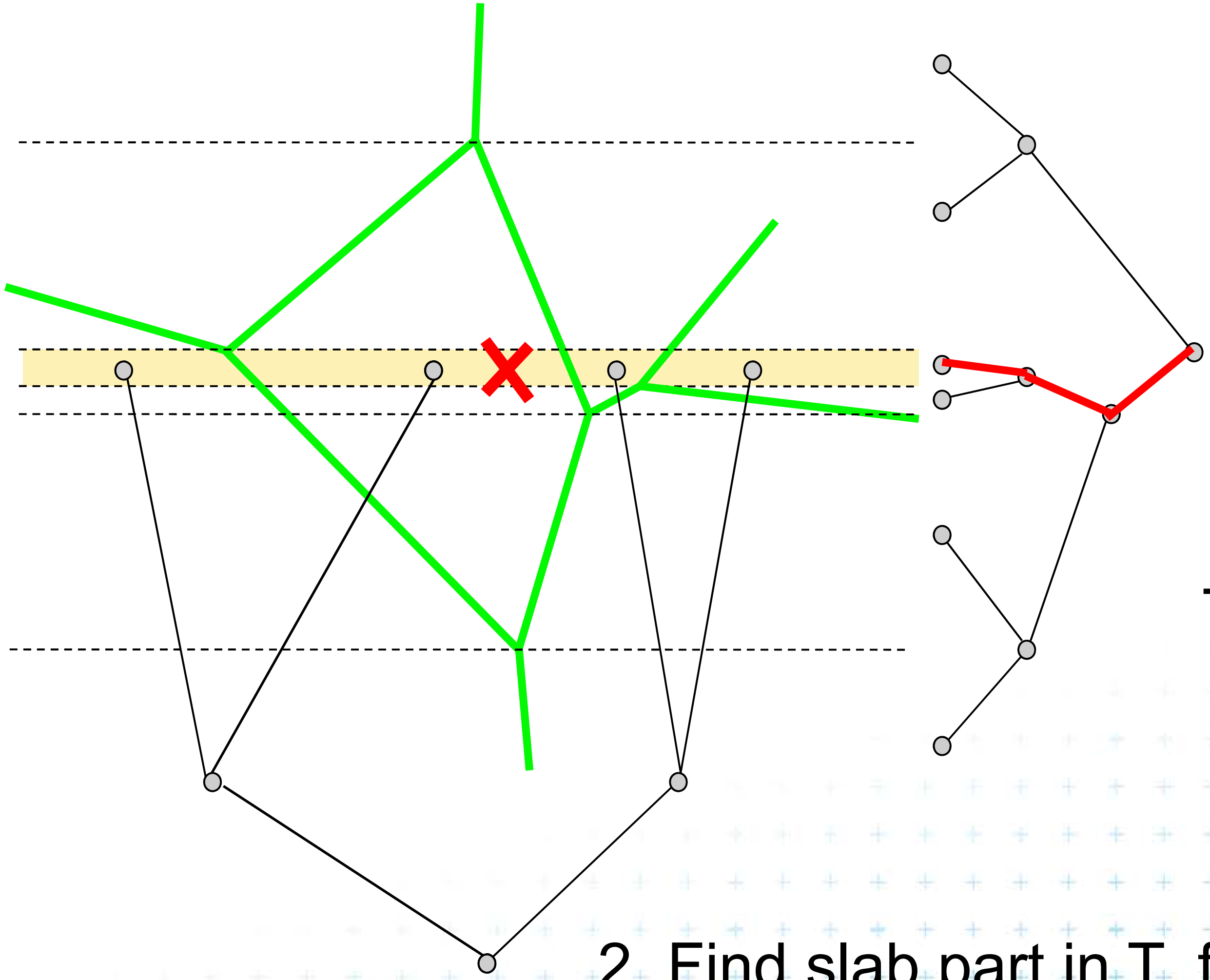


1. Find slab in T_y for y

T_x and T_y are arrays

2. Find slab part in T_x for x

Horizontal slabs example

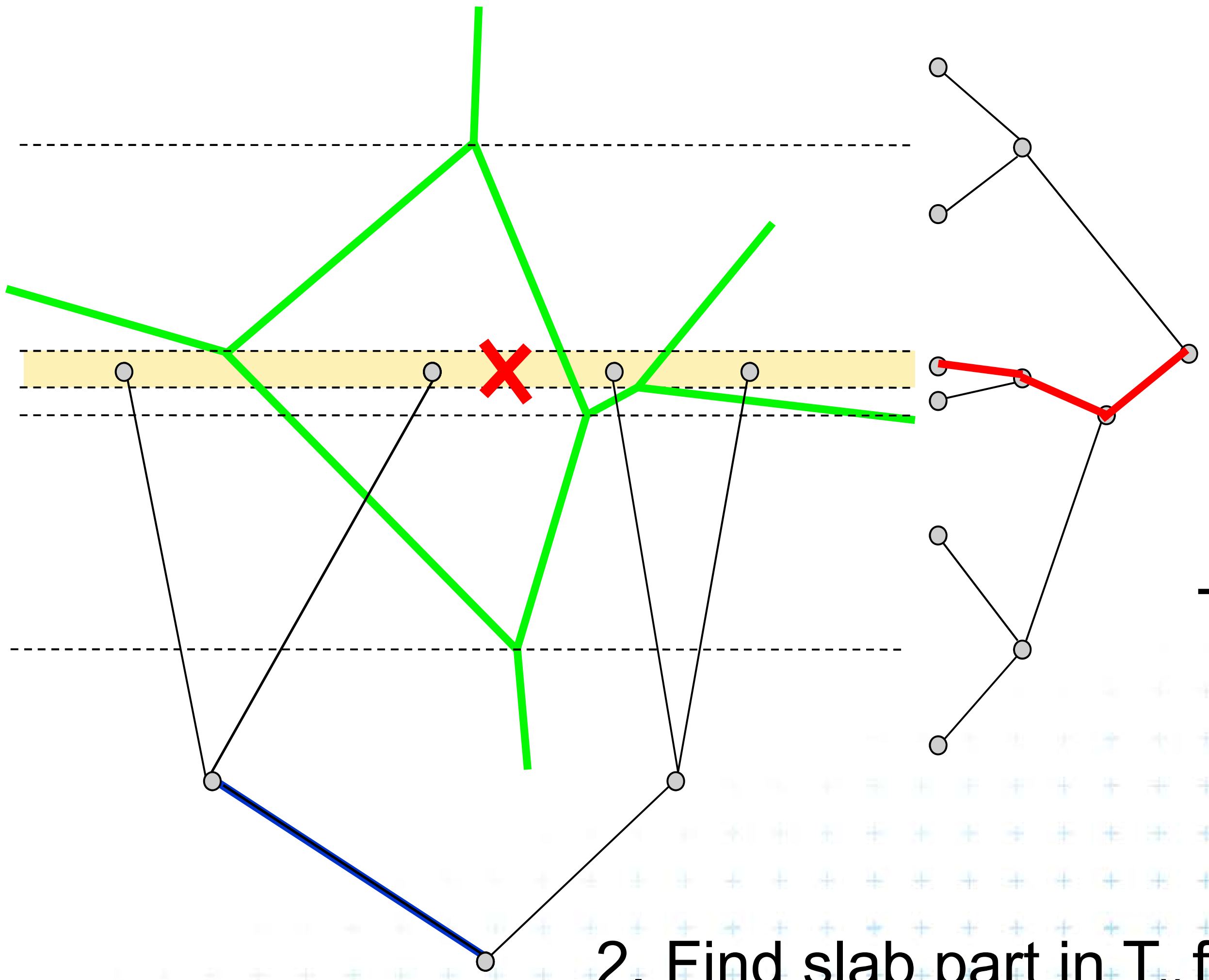


1. Find slab in T_y for y

T_x and T_y are arrays

2. Find slab part in T_x for x

Horizontal slabs example

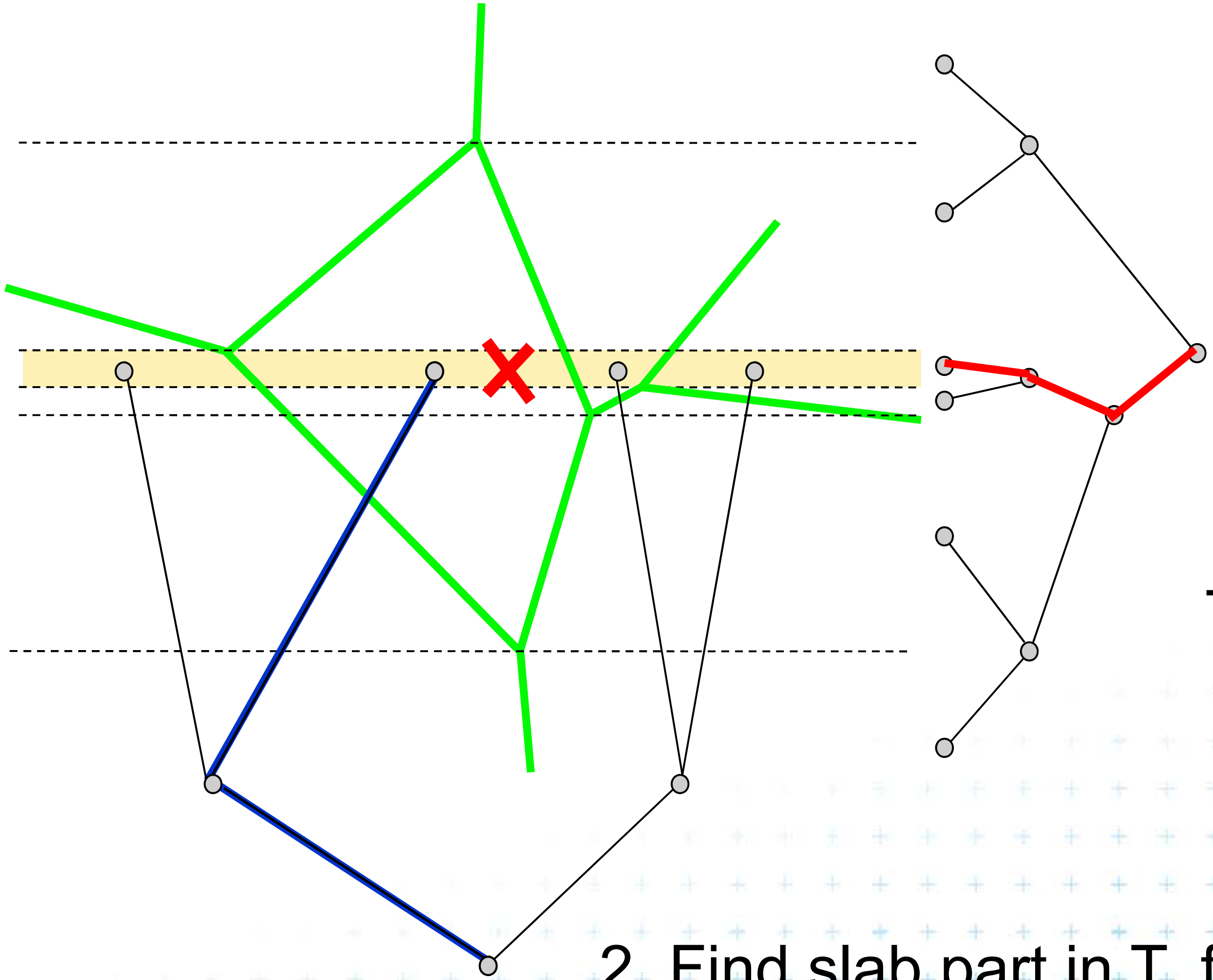


1. Find slab in T_y for y

T_x and T_y are arrays

2. Find slab part in T_x for x

Horizontal slabs example

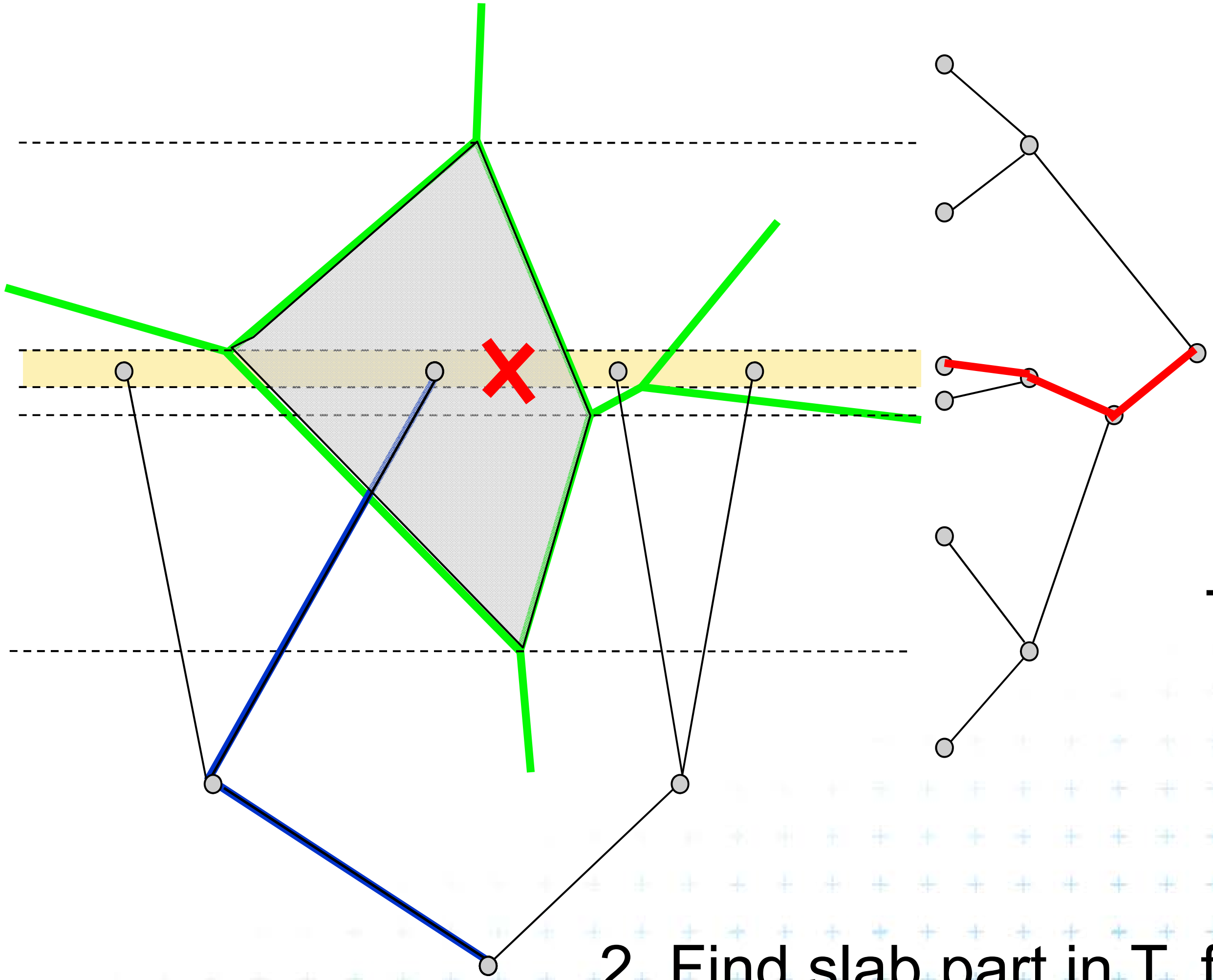


1. Find slab in T_y for y

T_x and T_y are arrays

2. Find slab part in T_x for x

Horizontal slabs example



1. Find slab in T_y for y

T_x and T_y are arrays

2. Find slab part in T_x for x

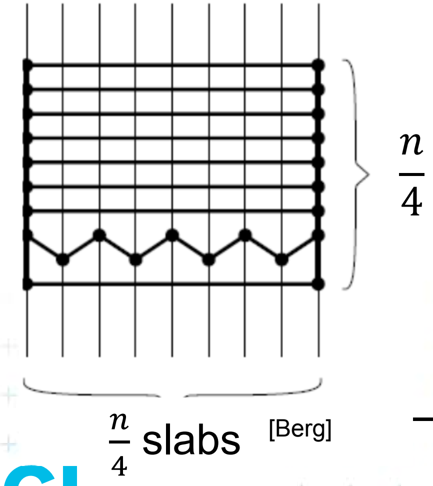
Horizontal slabs complexity

- Query time $O(\log n)$

  $O(\log n)$ time in slab array T_y (size max 2n endpoints)

  + $O(\log n)$ time in slab array T_x (slab crossed max by n edges)

- Memory $O(n^2)$
  - Slabs: Array with y-coordinates of vertices … $O(n)$
  - For each slab $O(n)$ edges intersecting the slab



[Berg]

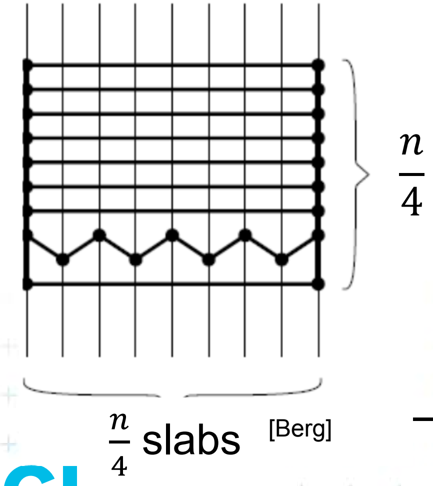
Horizontal slabs complexity

- Query time $O(\log n)$

  $O(\log n)$ time in slab array T_y (size max 2n endpoints)

  + $O(\log n)$ time in slab array T_x (slab crossed max by n edges)

- Memory $O(n^2)$
  - Slabs: Array with y-coordinates of vertices … $O(n)$
  - For each slab $O(n)$ edges intersecting the slab



[Berg]

$O(n^2)$ construction

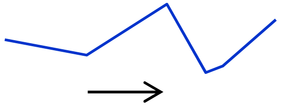
$O(\log n)$ query

$O(n^2)$ memory

2. Monotone chain tree

[Lee and Preparata, 1977]

- Construct monotone planar subdivision
  - The edges are all monotone in the same direction



- Each separator chain
  - is monotone (can be projected to line and searched)
  - splits the plane into two parts – allows binary search
- Algorithm
  - Preprocess: Find the separators (e.g., horizontal)
  - Search:

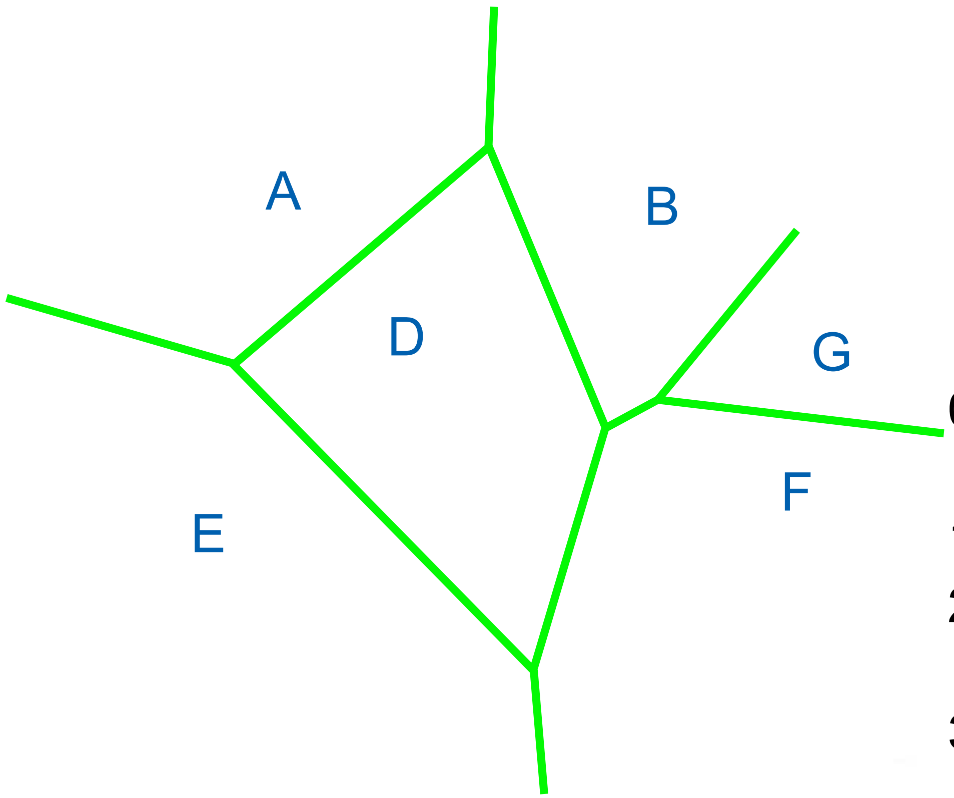
    Binary search among separators (Y) … $O(\log n)$ times

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

    $O(\log^2 n)$ query

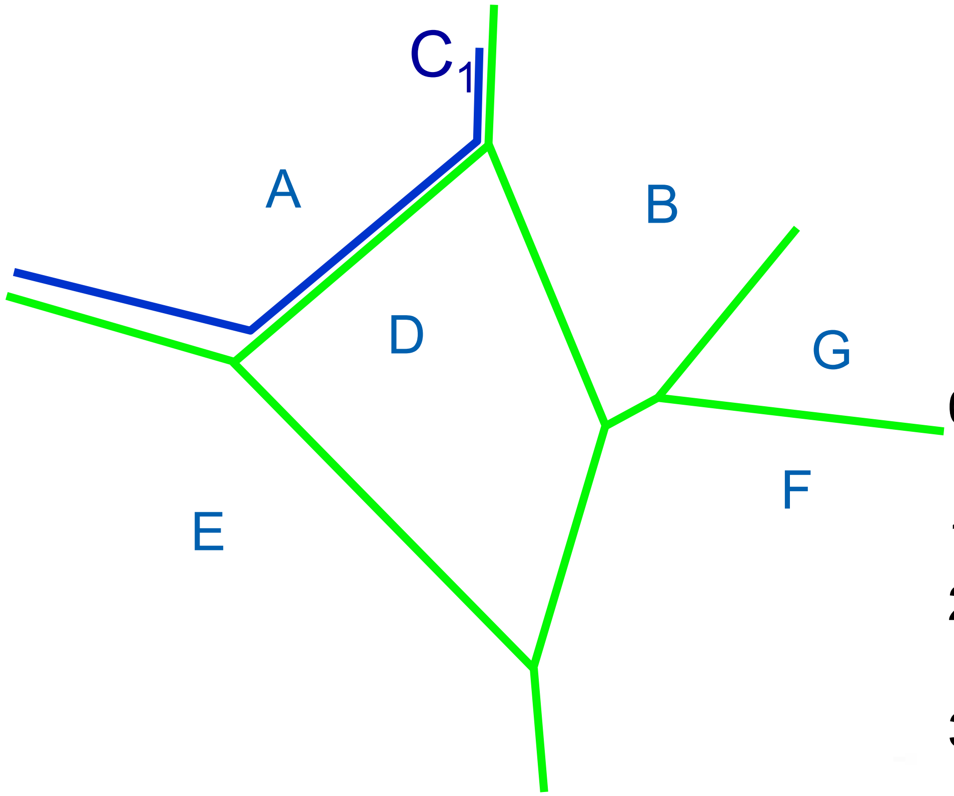
    $O(n^2)$ memory
  - Not optimal, but simple
  - Can be made optimal, but the algorithm and data structures are complicated

Monotone chain tree example



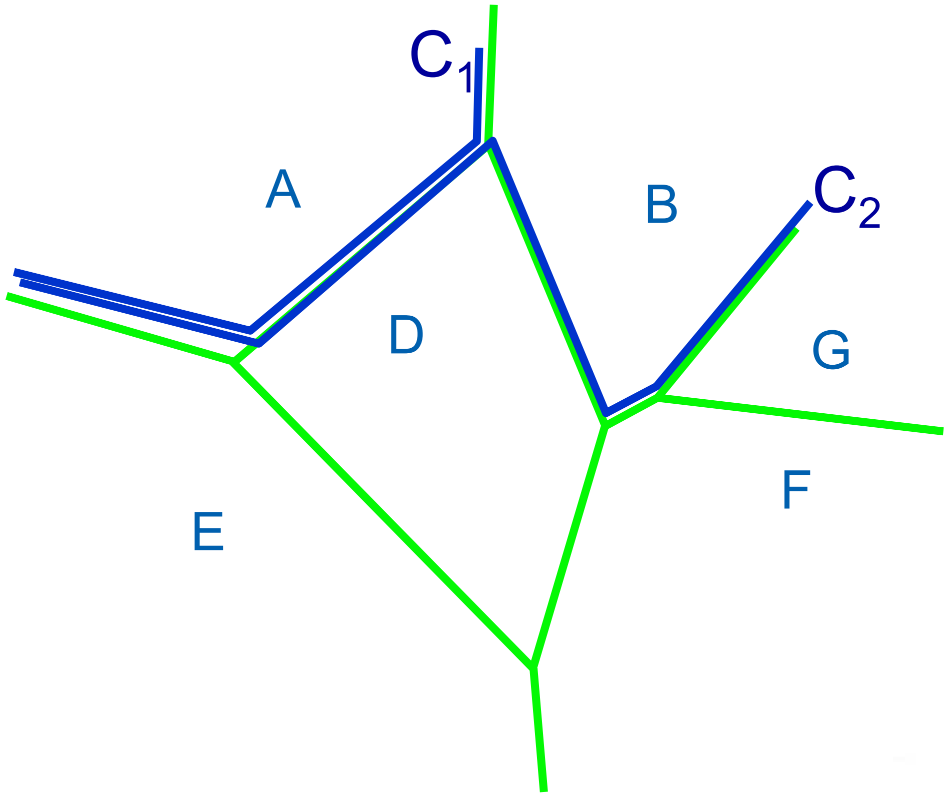
0. Construct the chains
   and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of
   the chain – determine the segment
3. Identify position of x in relation to the
   segment – Left or Right
   (This is the position of *x* relatively to the
   whole chain)
4.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

Monotone chain tree example



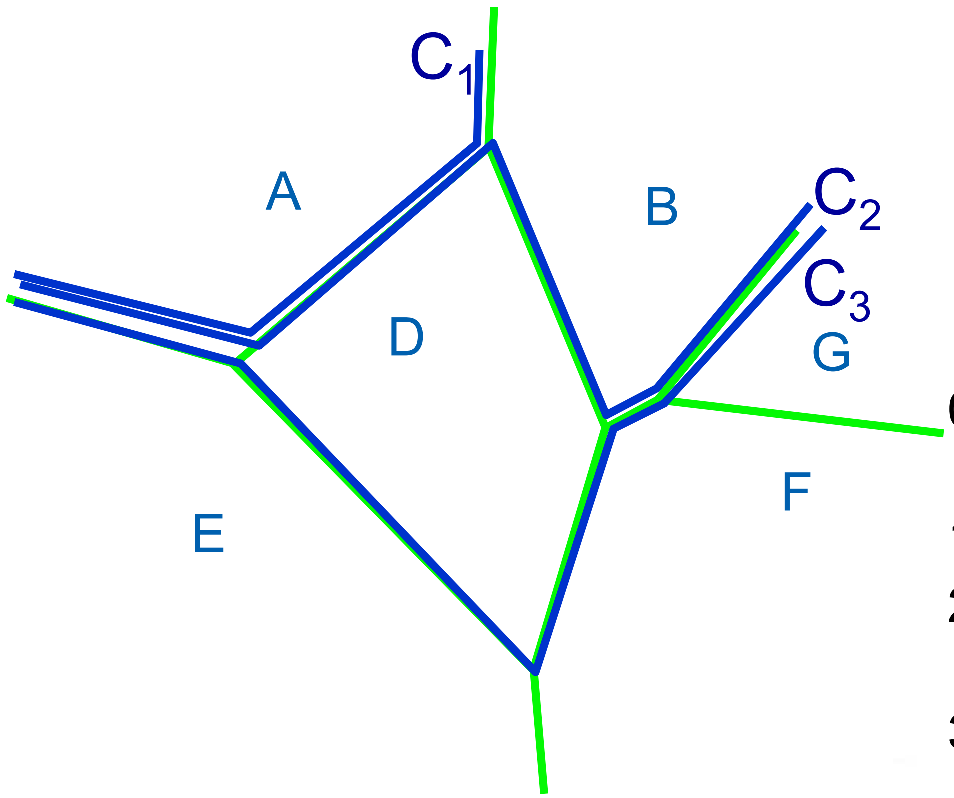
0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right
   (This is the position of *x* relatively to the whole chain)
4. Continue in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

Monotone chain tree example



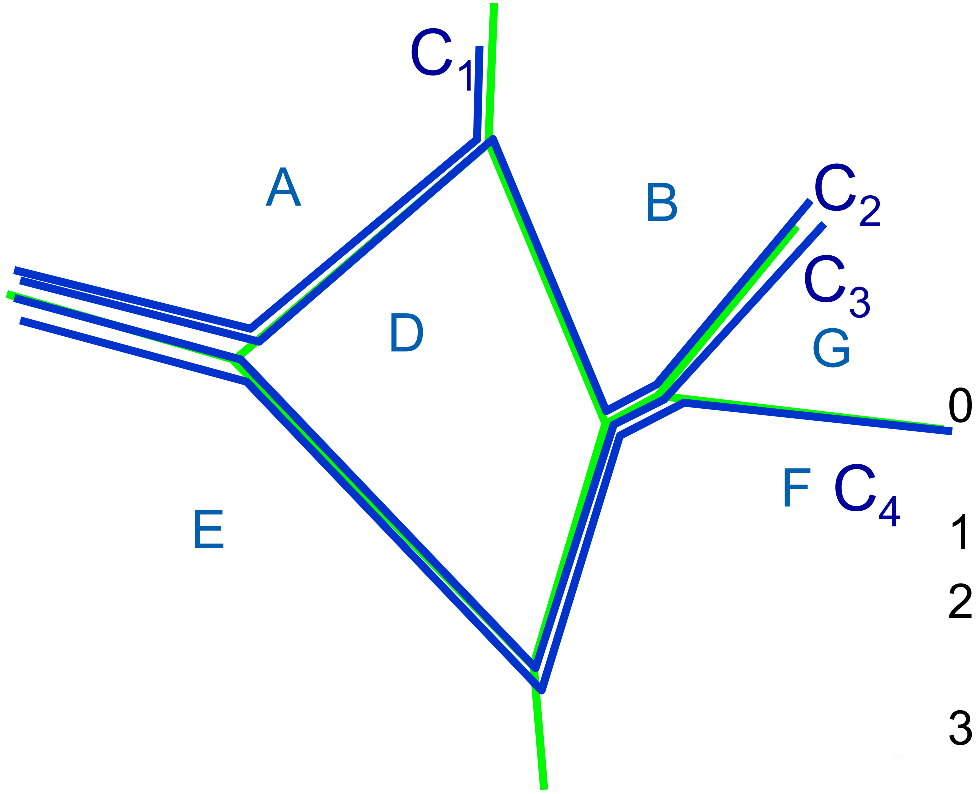
0. Construct the chains
   and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of
   the chain – determine the segment
3. Identify position of x in relation to the
   segment – Left or Right
   (This is the position of *x* relatively to the
   whole chain)
4.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

Monotone chain tree example



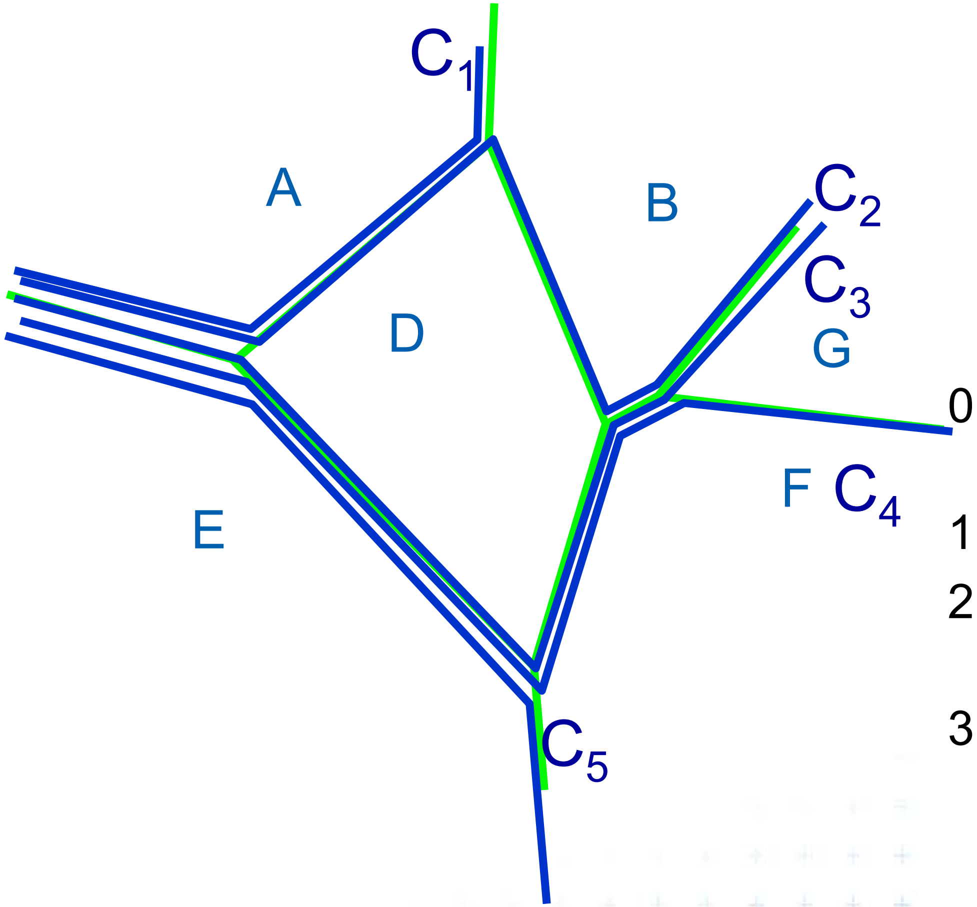
0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right (This is the position of *x* relatively to the whole chain)
4. Continue in L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

Monotone chain tree example



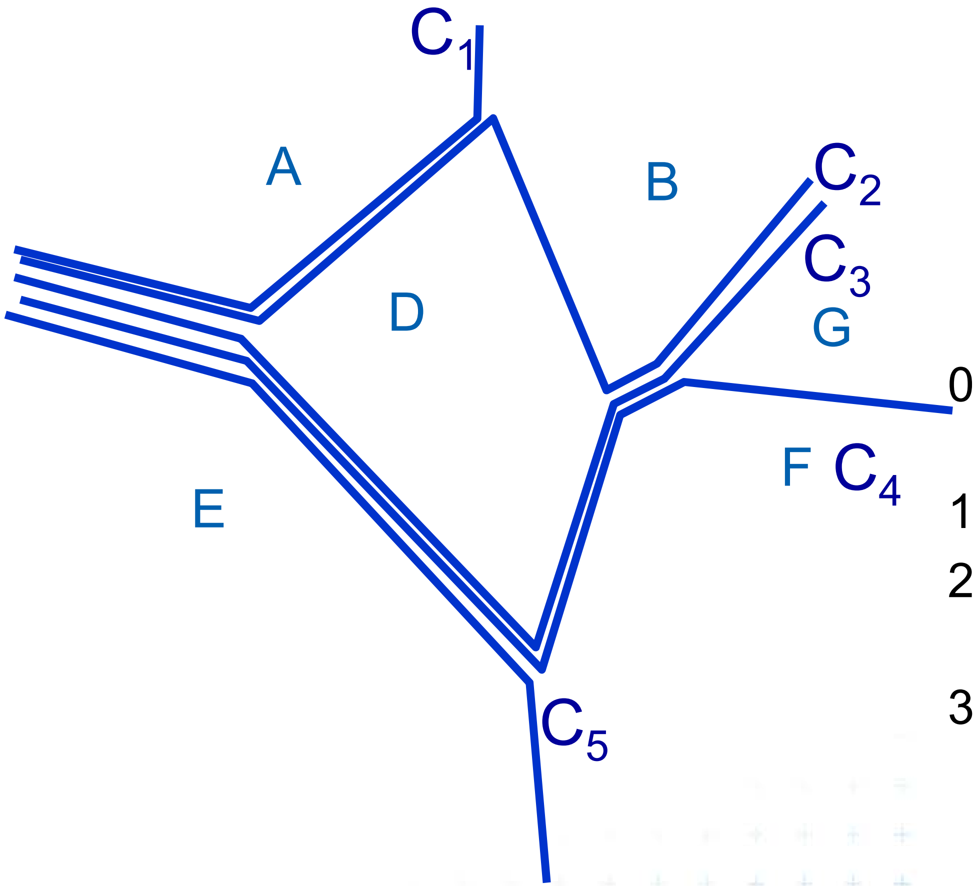
0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right (This is the position of *x* relatively to the whole chain)
4. Continue in L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

Monotone chain tree example



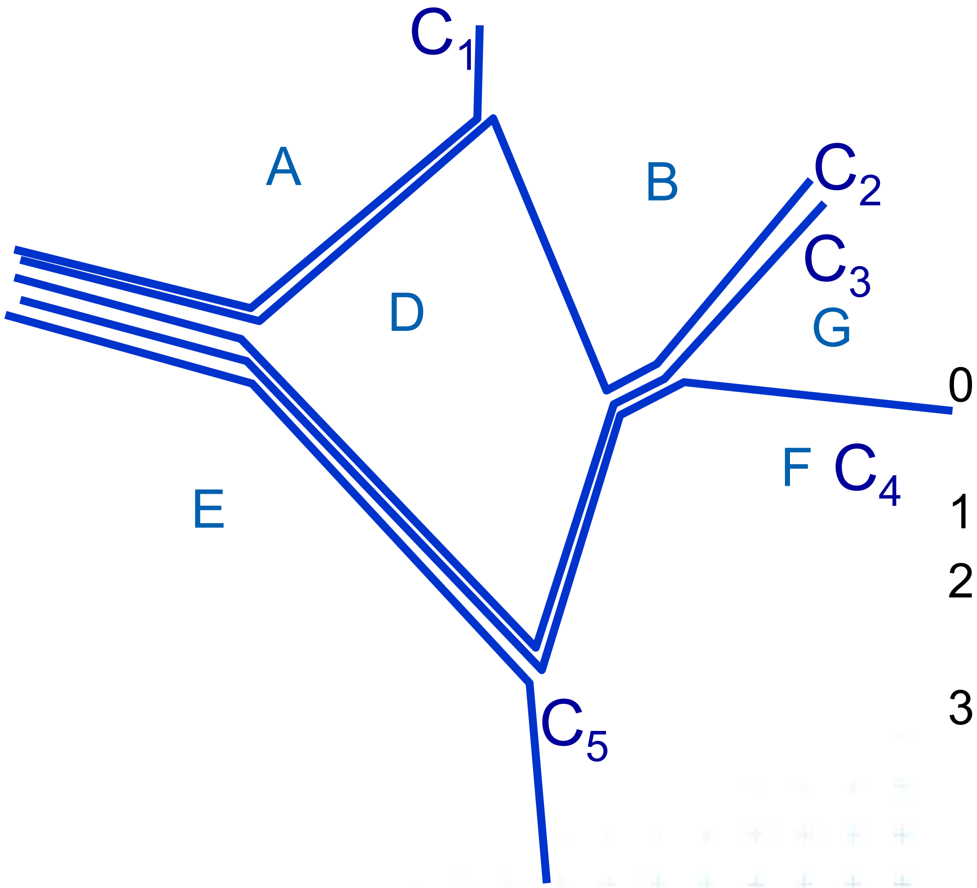
0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right
   (This is the position of *x* relatively to the whole chain)
4. Continue in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

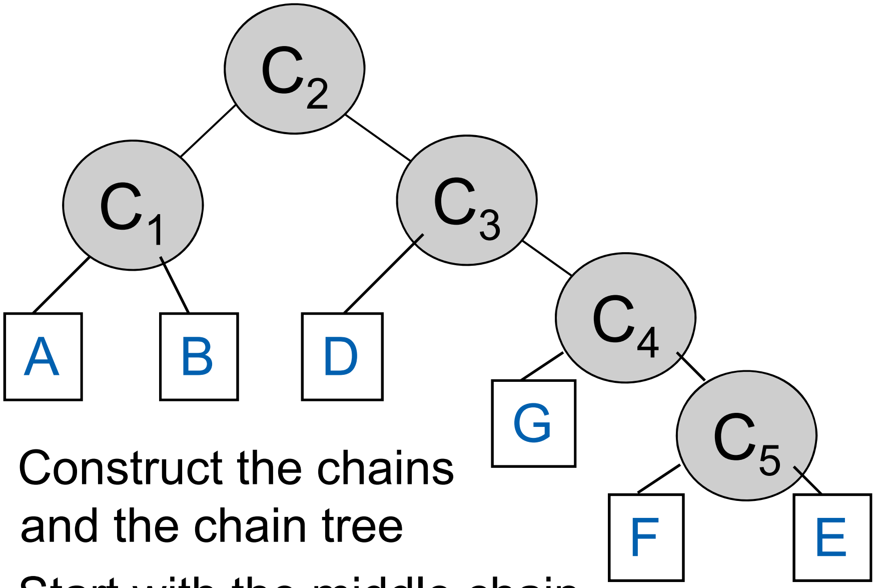
Monotone chain tree example



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

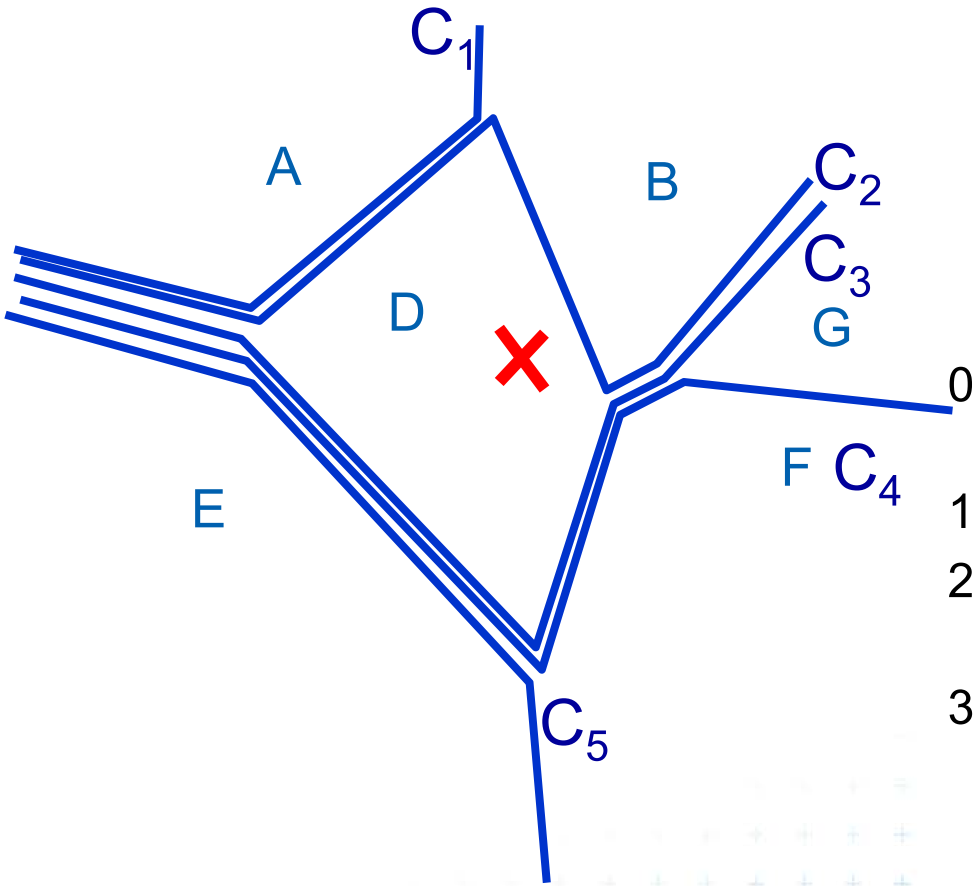
Monotone chain tree example

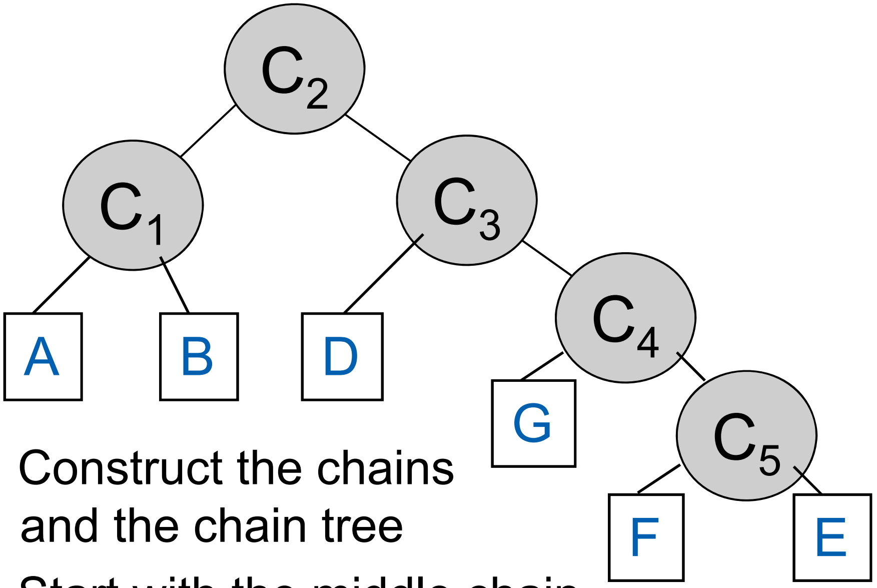




0. Construct the chains
   and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of
   the chain – determine the segment
3. Identify position of x in relation to the
   segment – Left or Right
   (This is the position of *x* relatively to the
   whole chain)
4.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

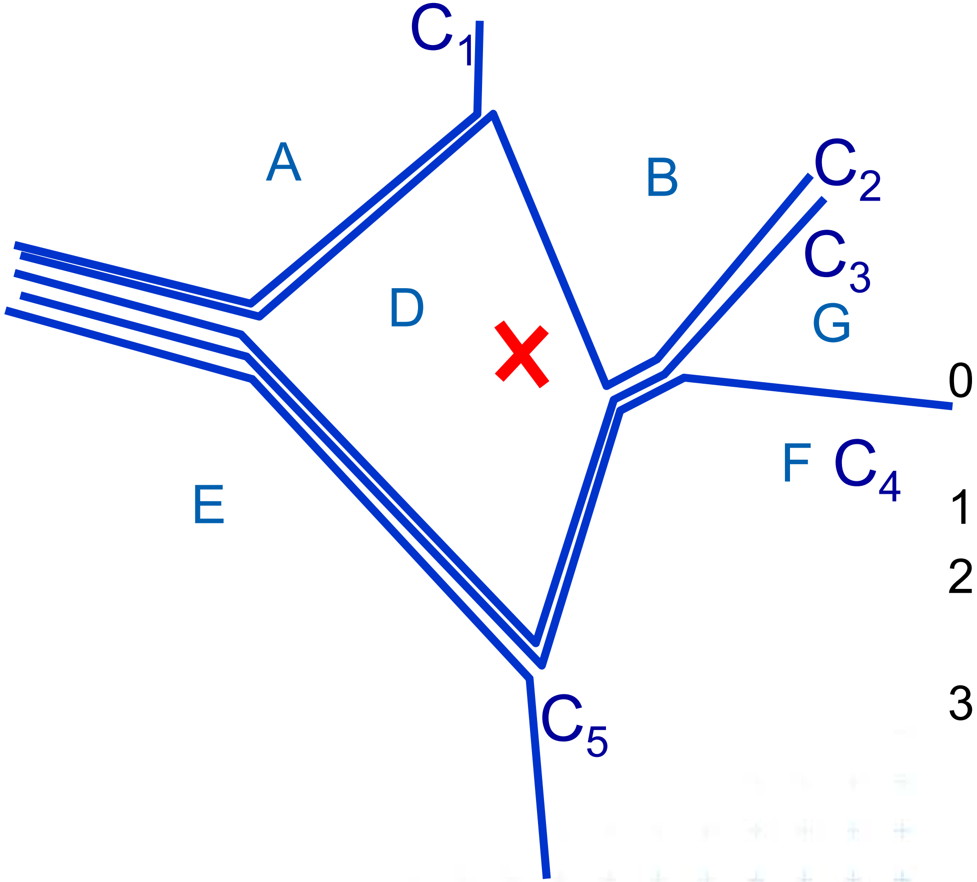
Monotone chain tree example

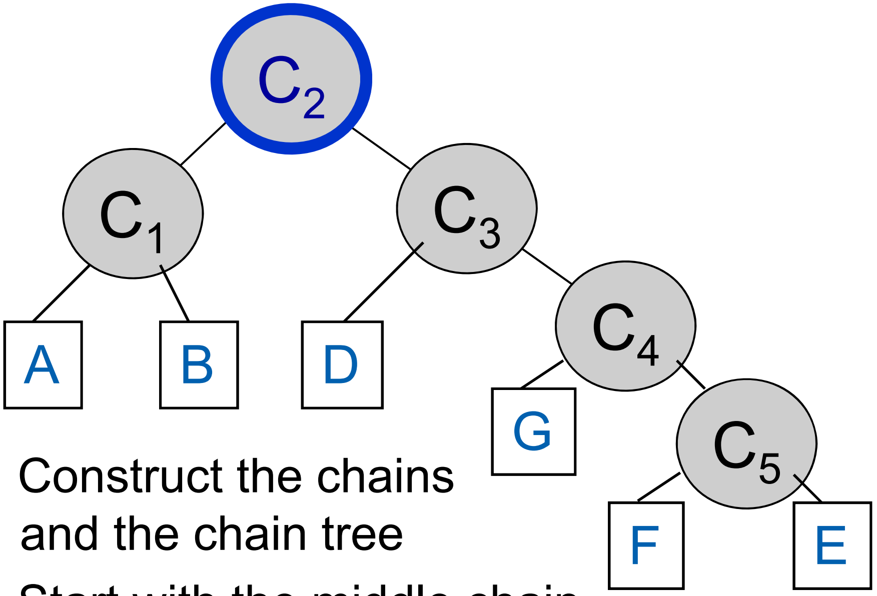




0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right
   (This is the position of *x* relatively to the whole chain)
4. Continue in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

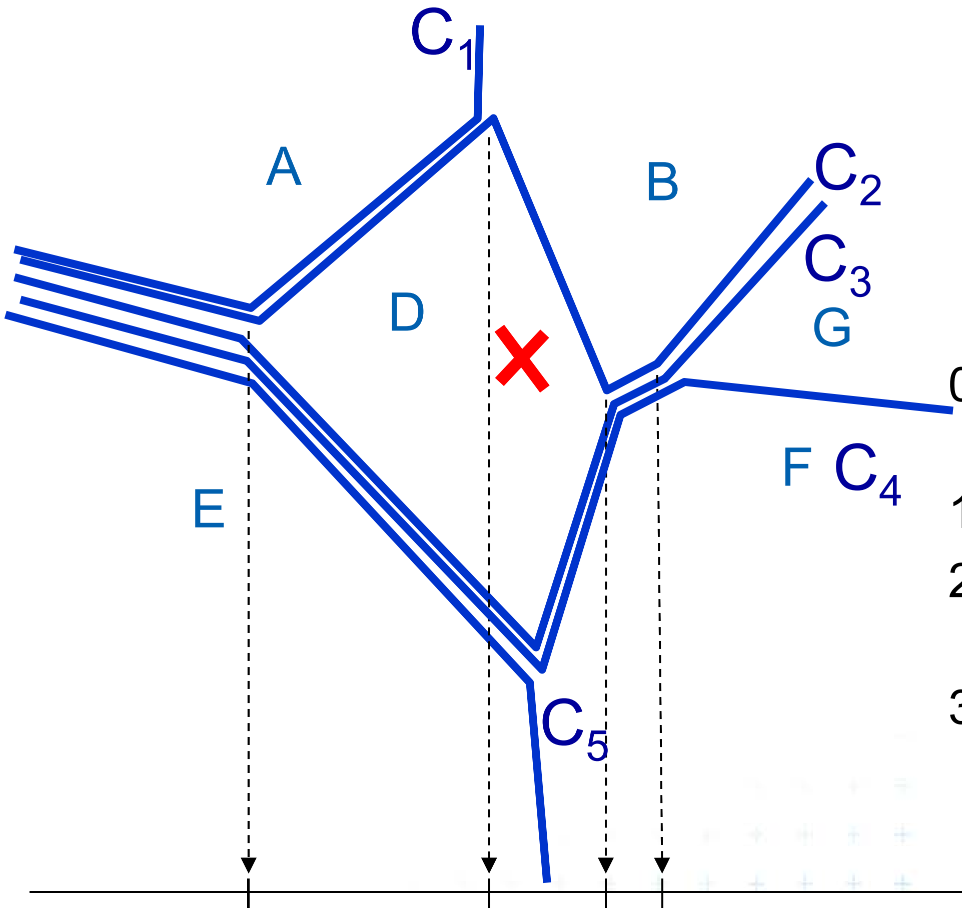
Monotone chain tree example

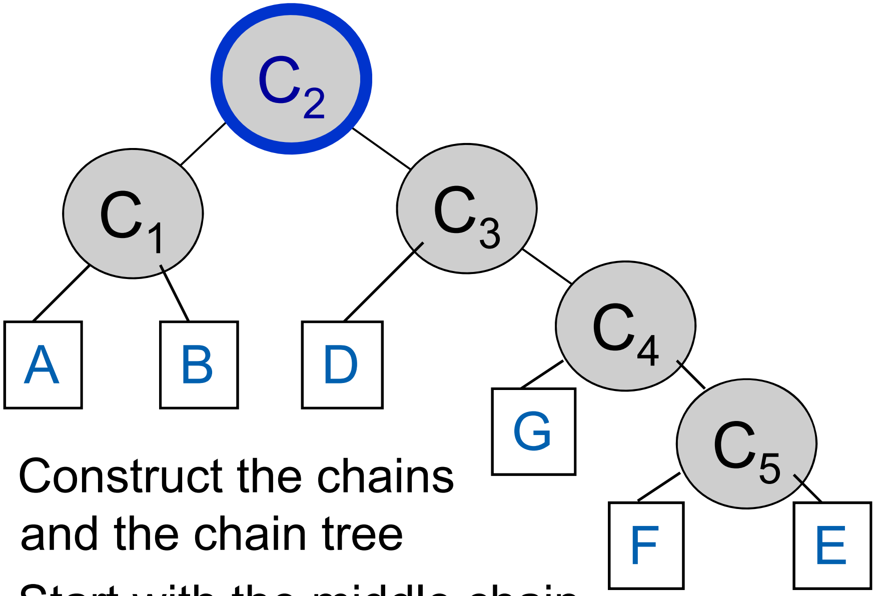




0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right (This is the position of *x* relatively to the whole chain)
4. Continue in L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

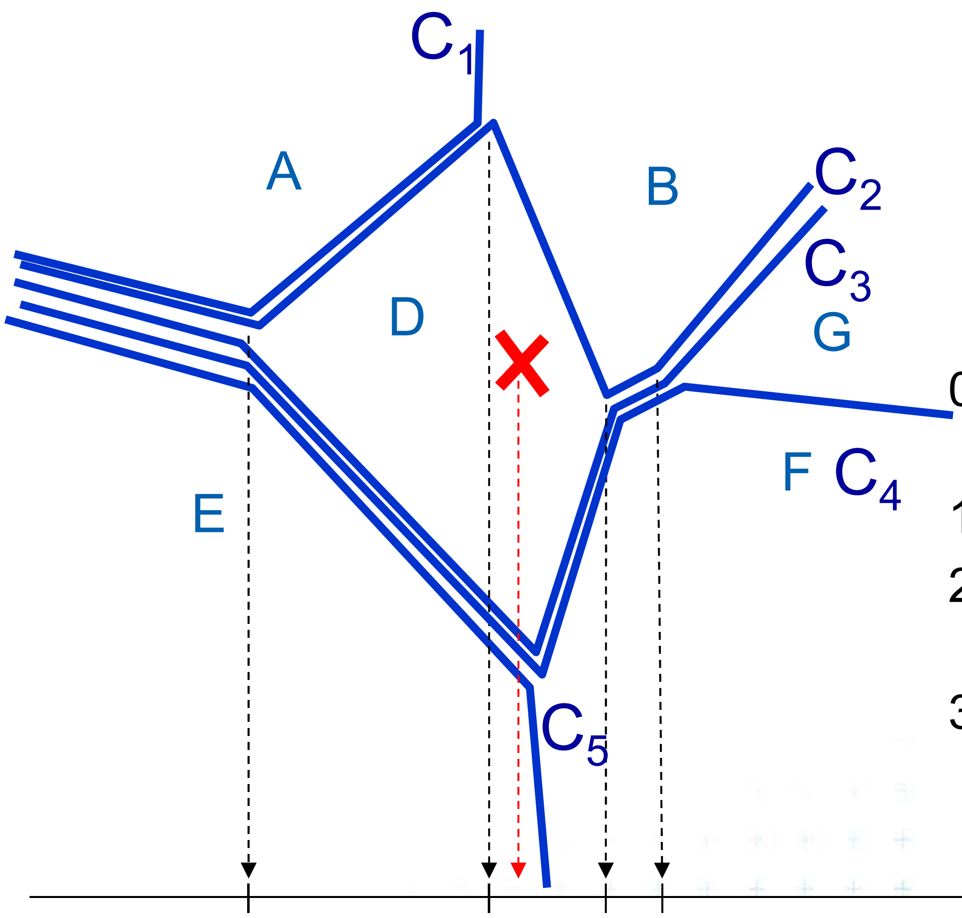
Monotone chain tree example

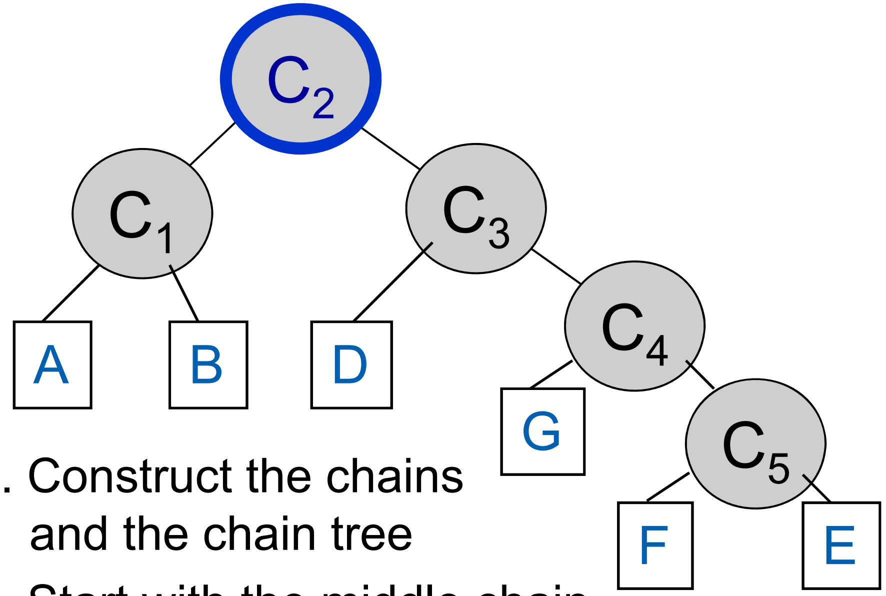




0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right
   (This is the position of *x* relatively to the whole chain)
4. Continue in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

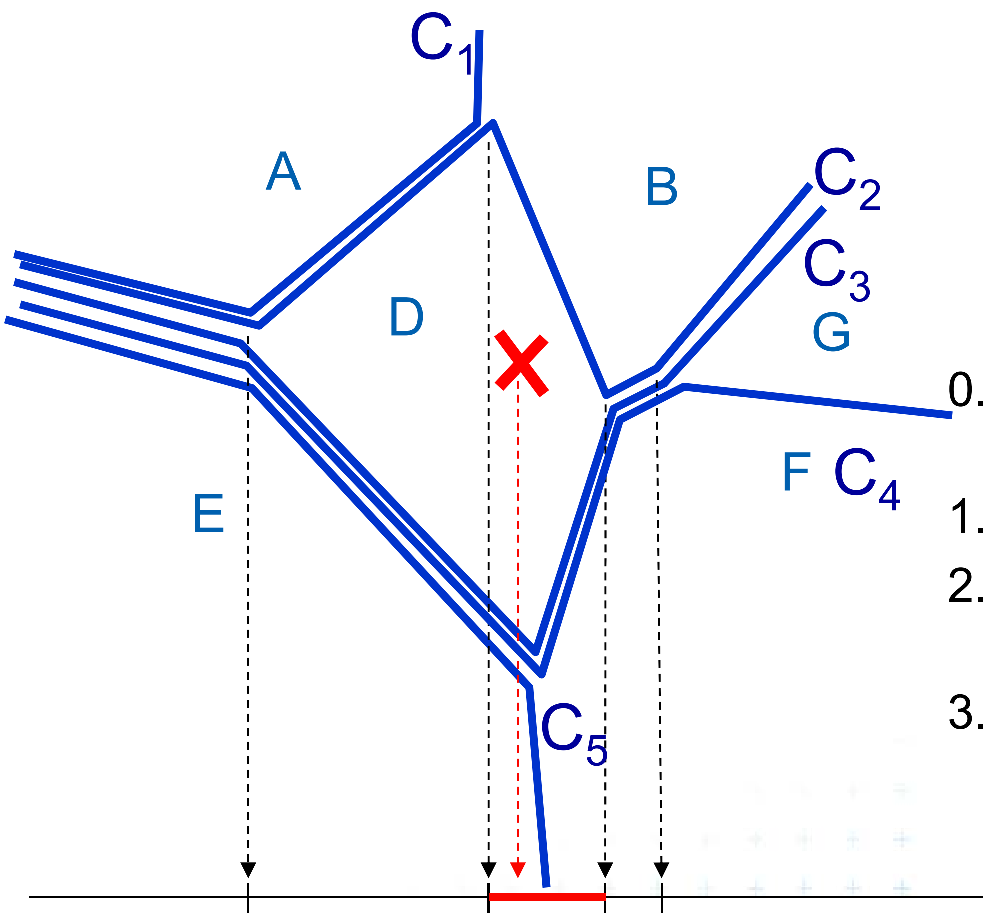
Monotone chain tree example

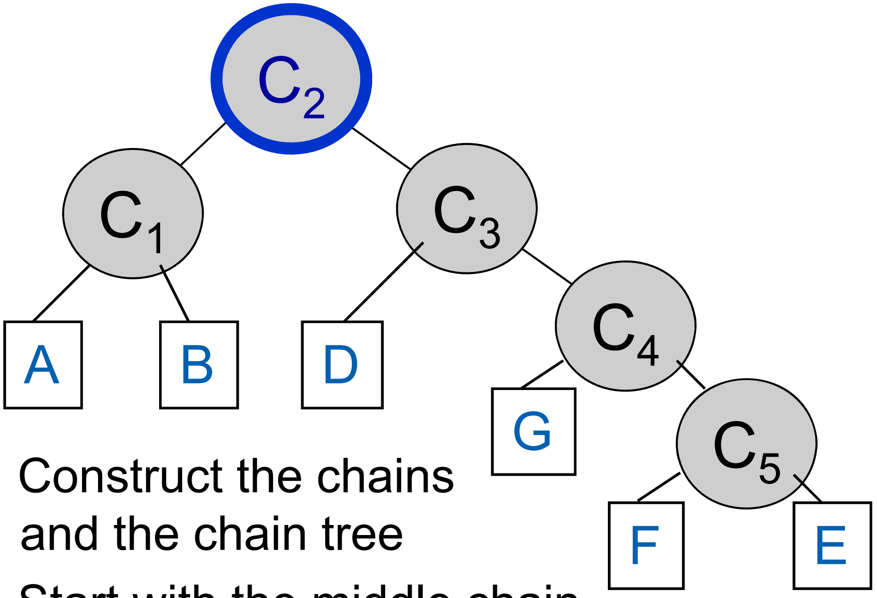




0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right (This is the position of *x* relatively to the whole chain)
4. Continue in L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

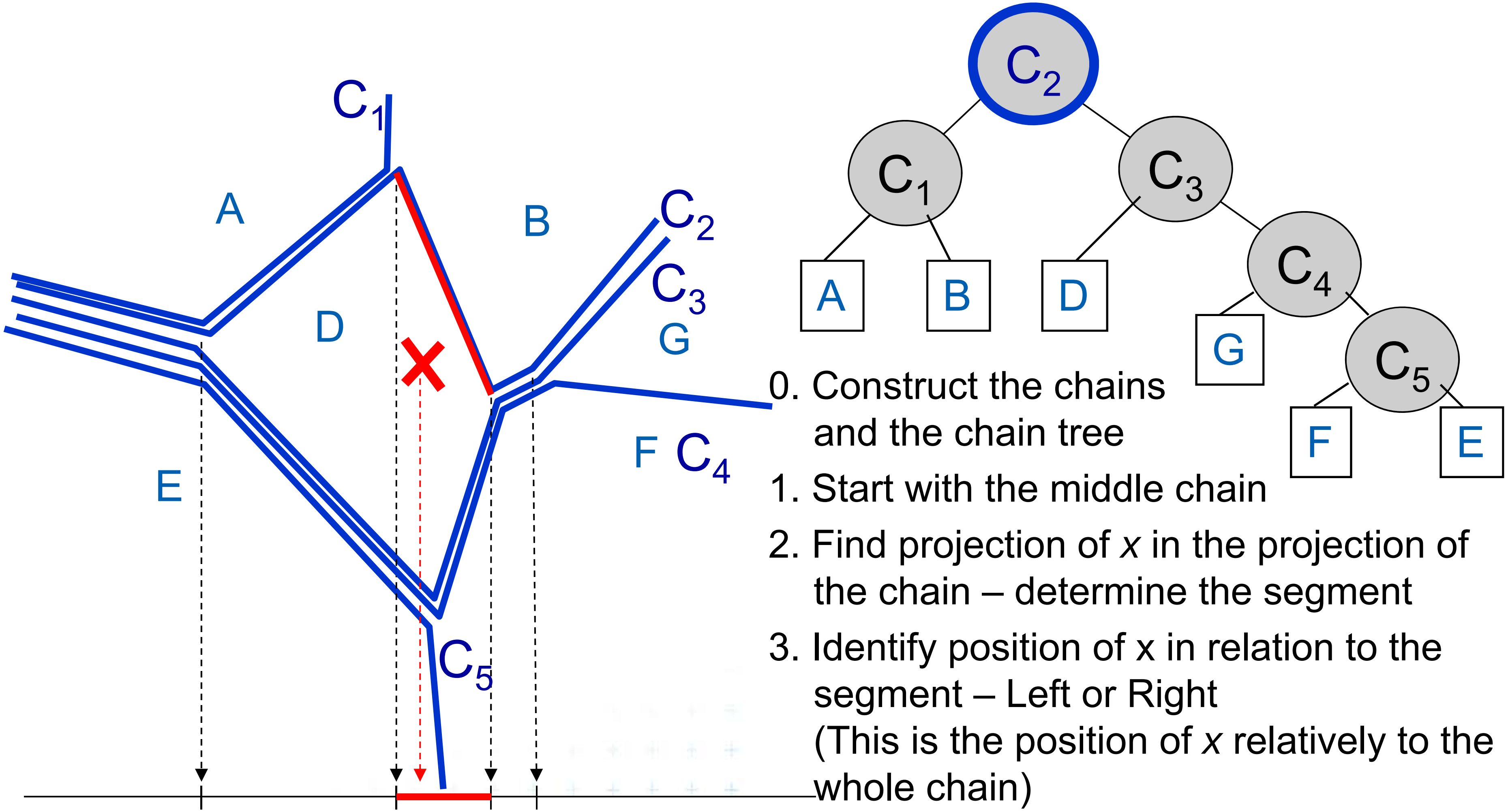
Monotone chain tree example





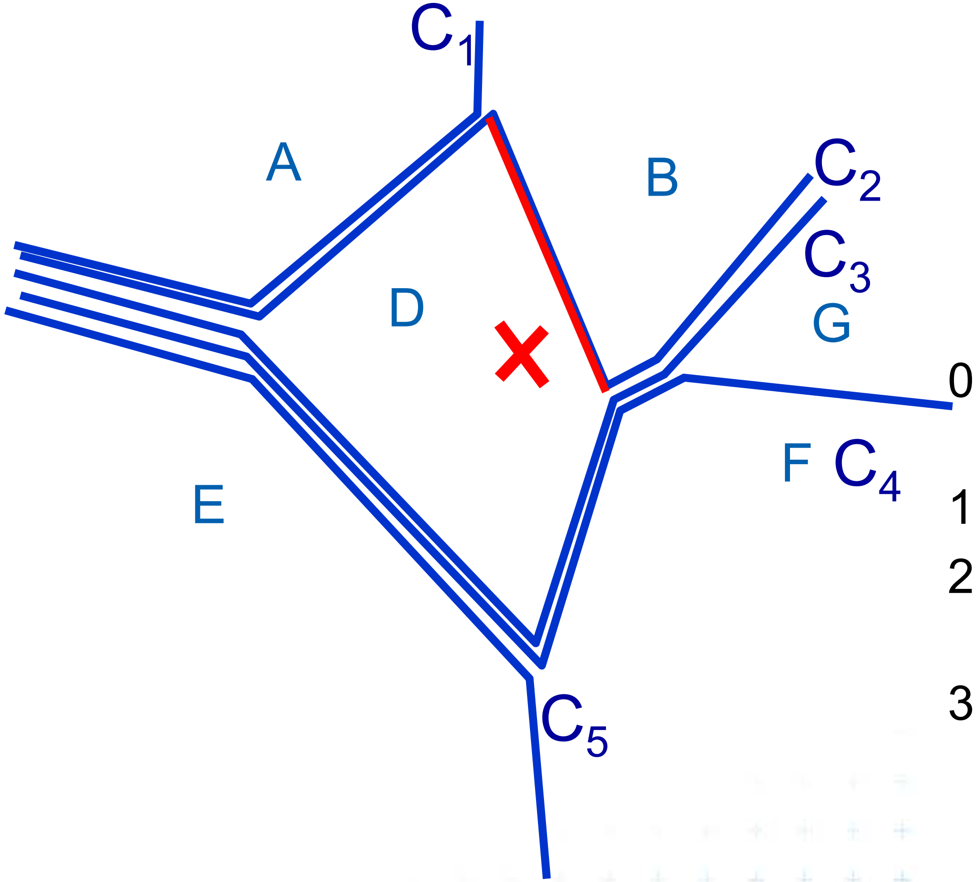
0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right
   (This is the position of *x* relatively to the whole chain)
4. Continue in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

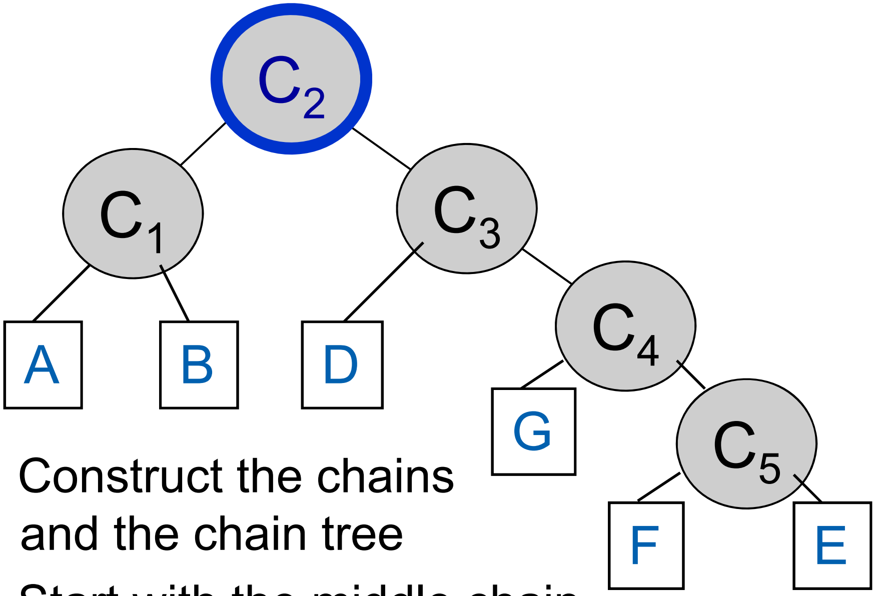
Monotone chain tree example



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

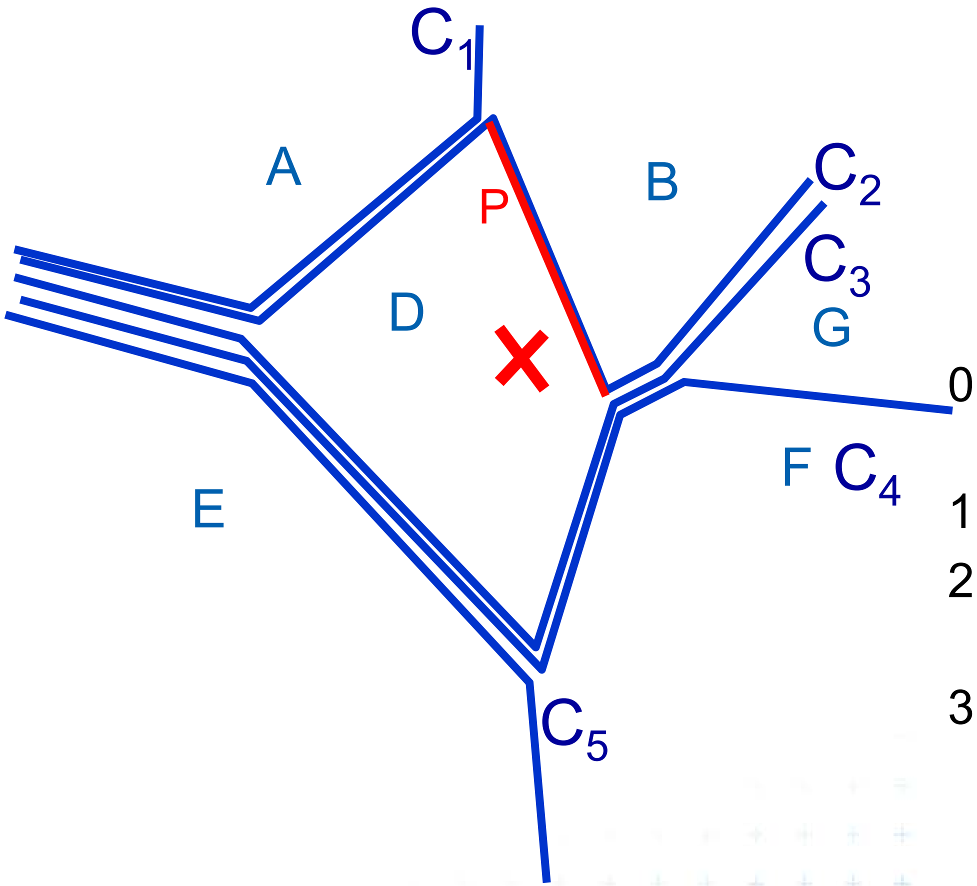
Monotone chain tree example

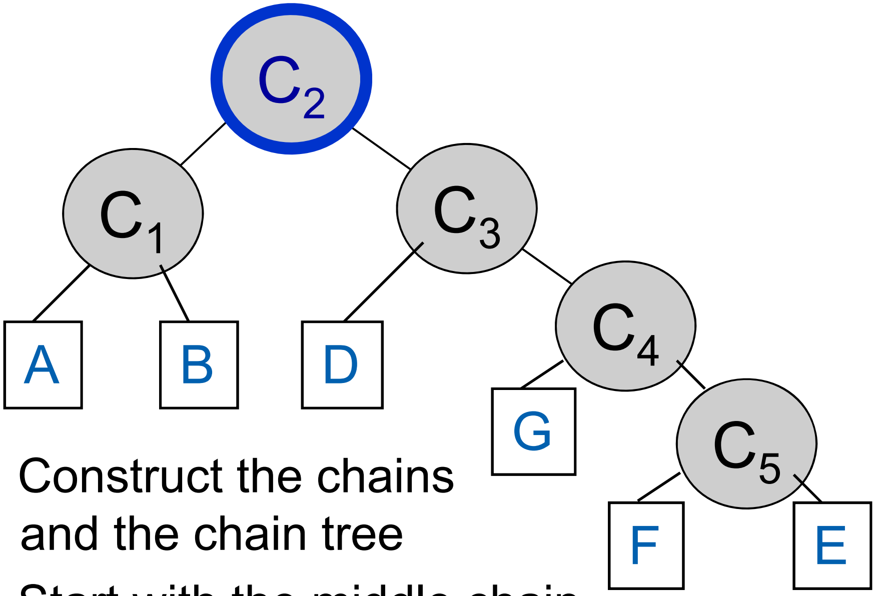




0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right (This is the position of *x* relatively to the whole chain)
4. Continue in L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

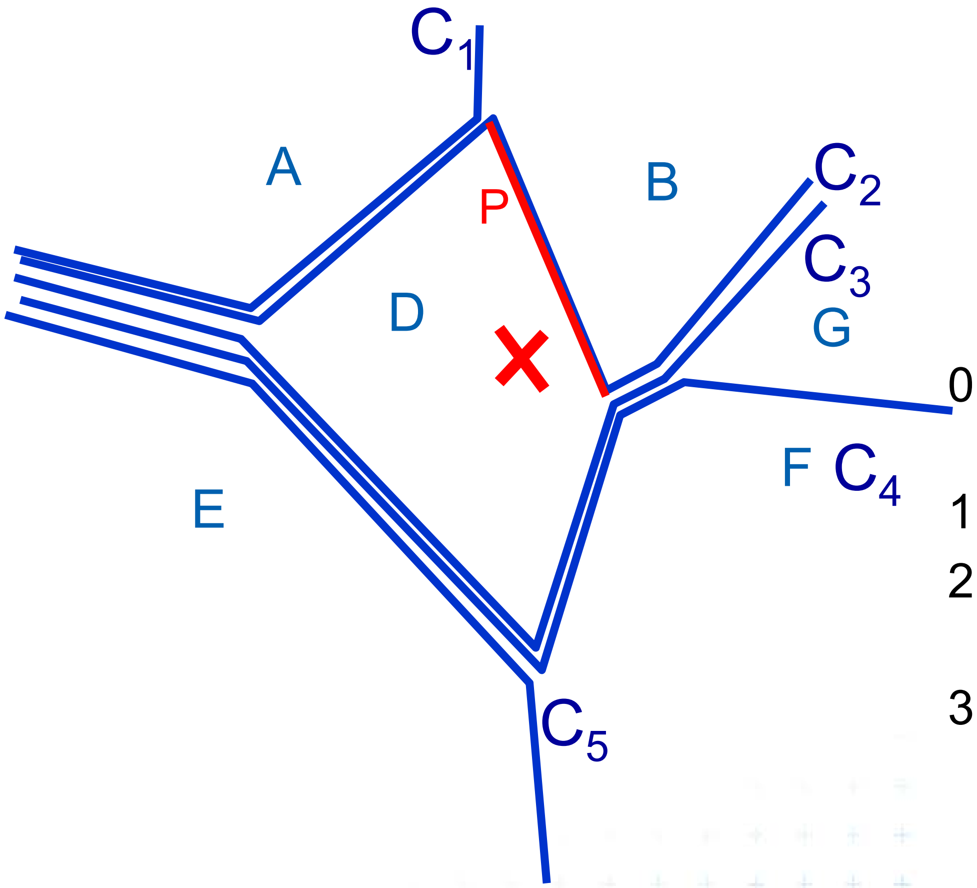
Monotone chain tree example

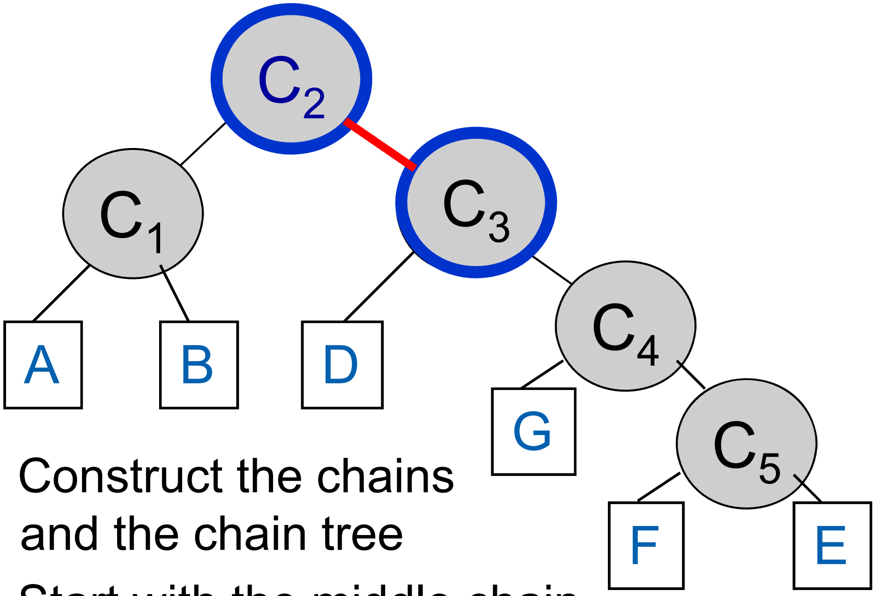




0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right
   (This is the position of *x* relatively to the whole chain)
4. Continue in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

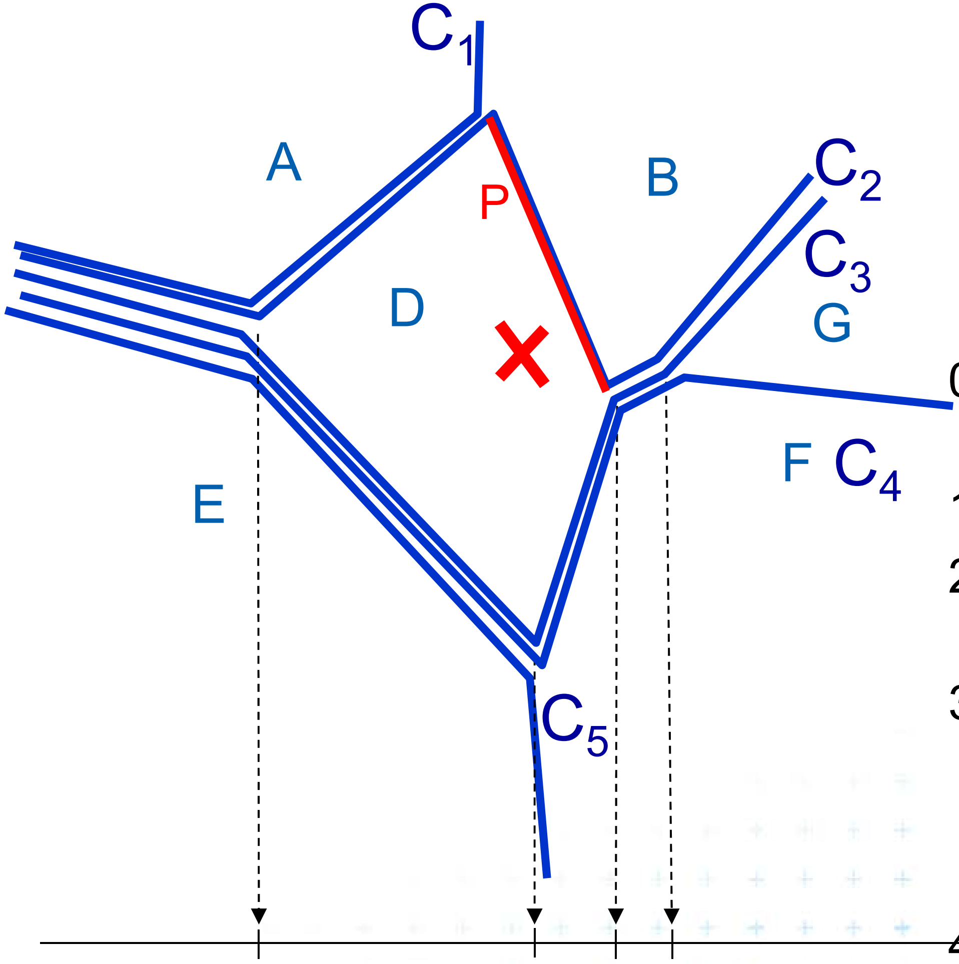
Monotone chain tree example

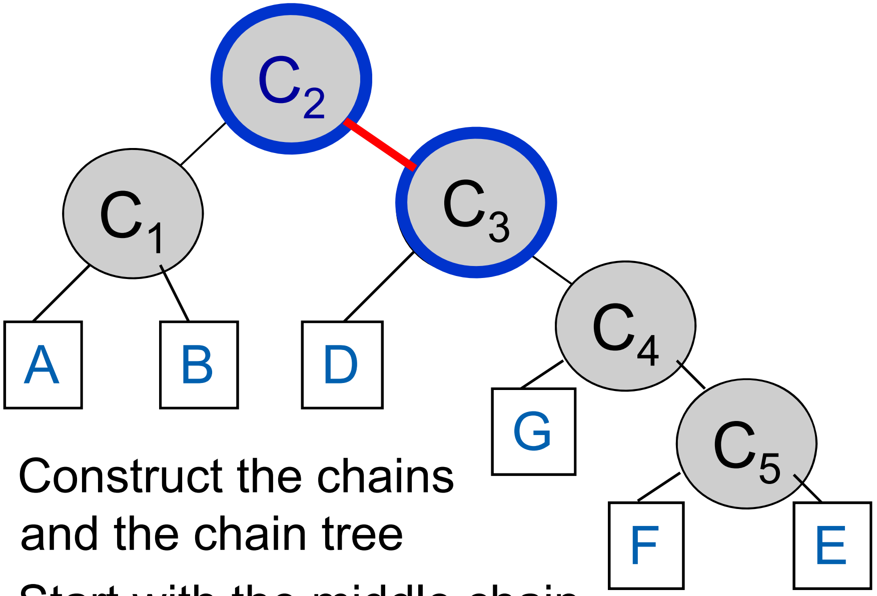




0. Construct the chains
   and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of
   the chain – determine the segment
3. Identify position of x in relation to the
   segment – Left or Right
   (This is the position of *x* relatively to the
   whole chain)
4.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

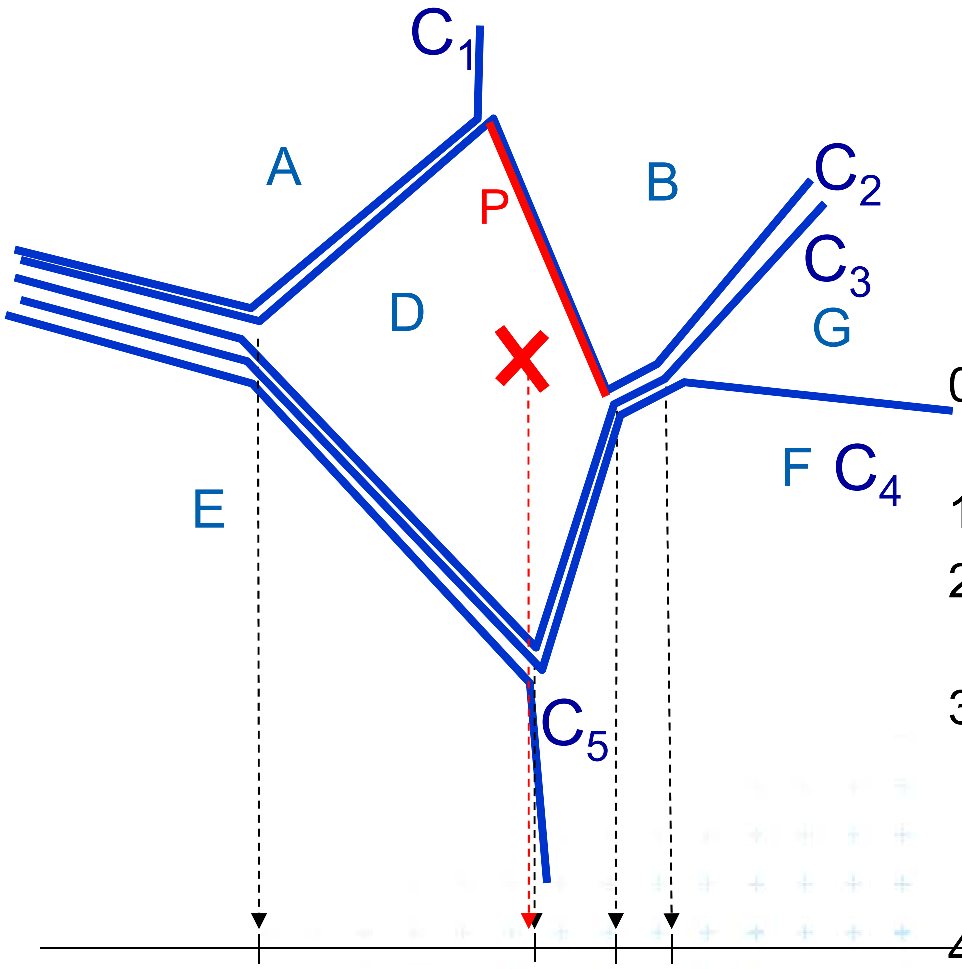
Monotone chain tree example

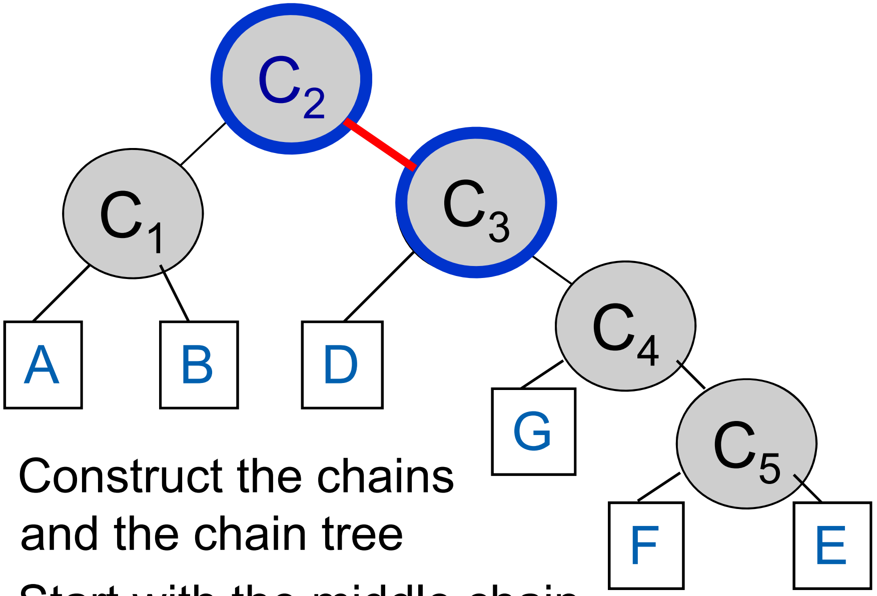




0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right
   (This is the position of *x* relatively to the whole chain)
4. Continue in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

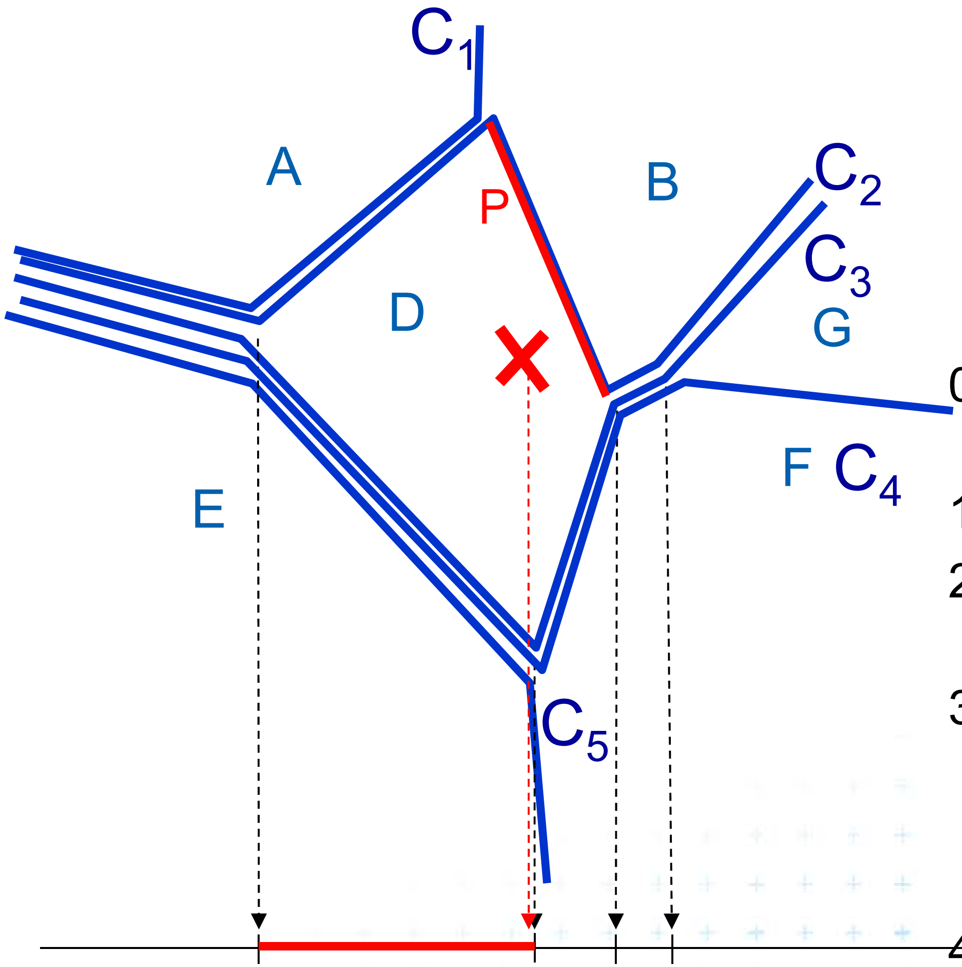
Monotone chain tree example

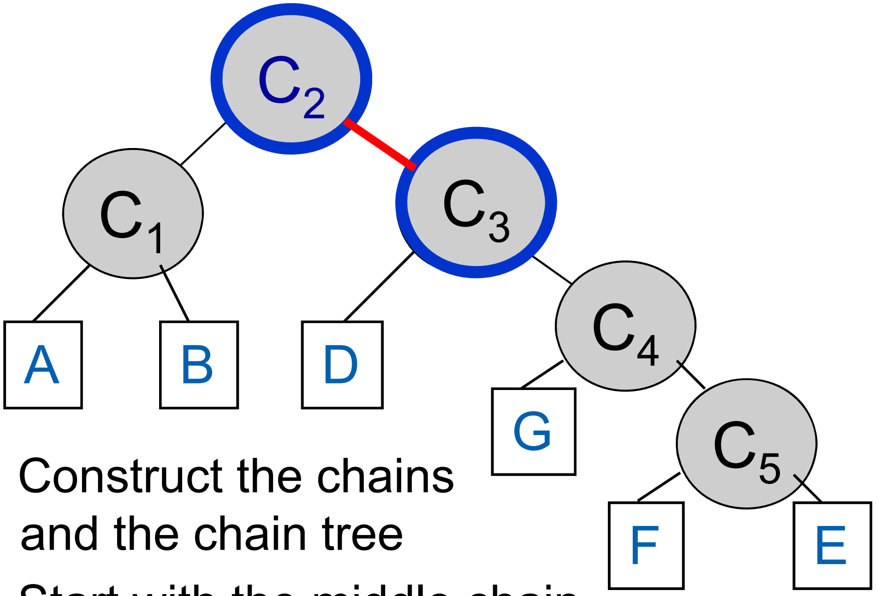




0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right
   (This is the position of *x* relatively to the whole chain)
4. Continue in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

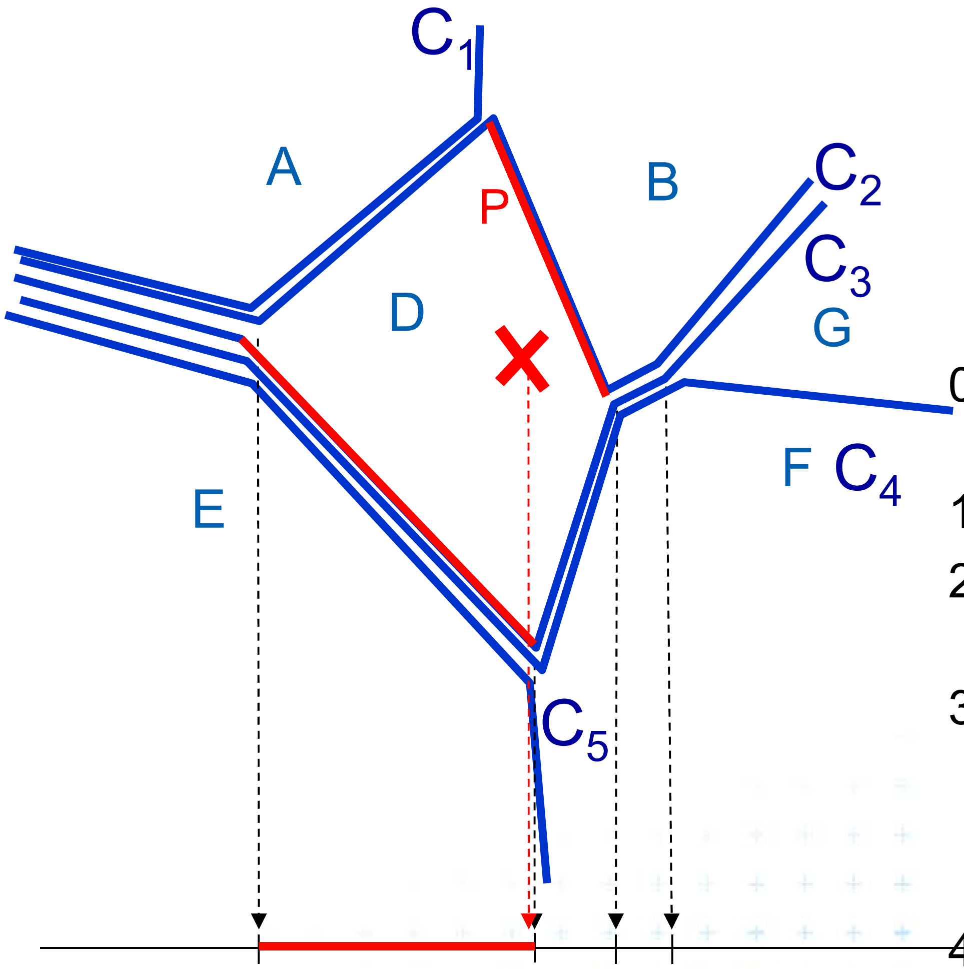
Monotone chain tree example

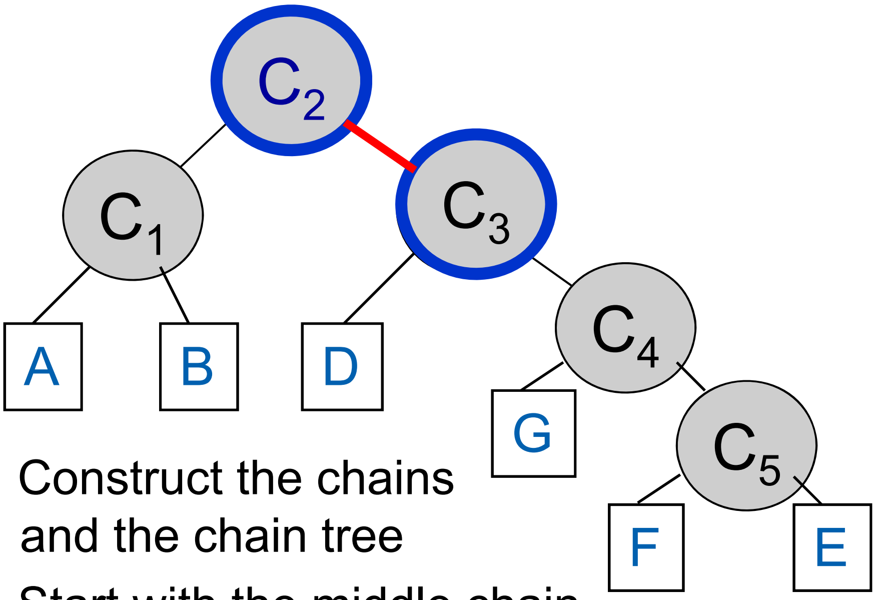




0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right (This is the position of *x* relatively to the whole chain)
4. Continue in L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

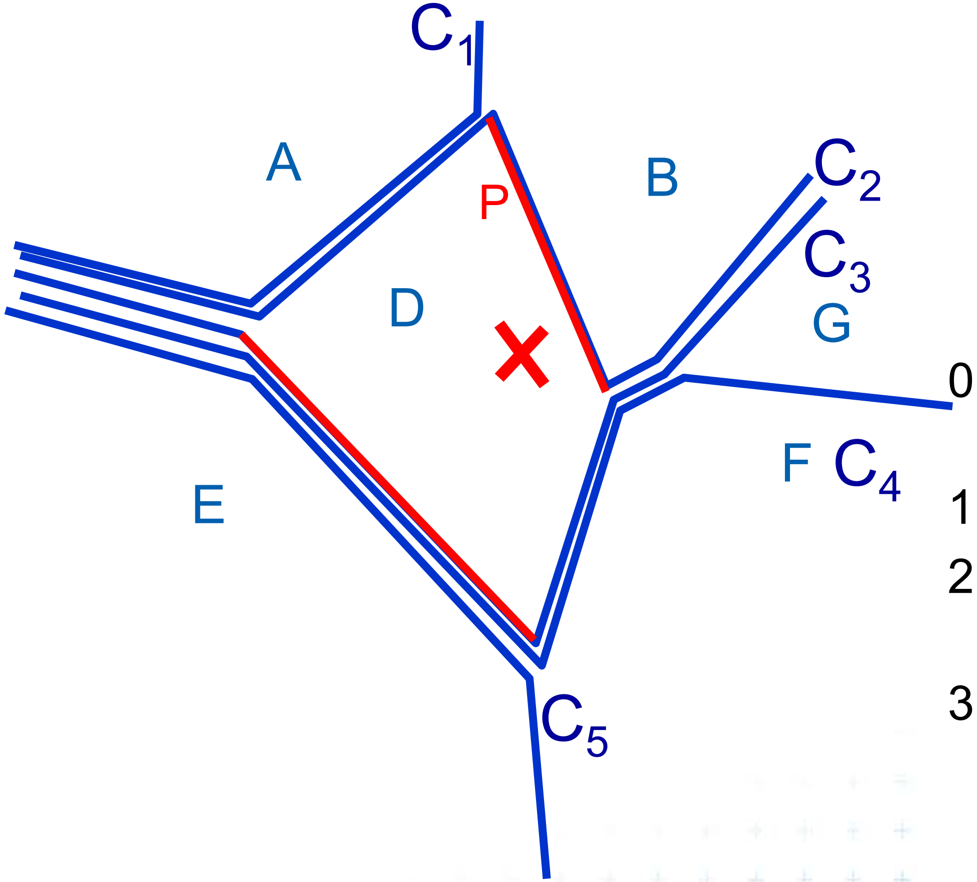
Monotone chain tree example

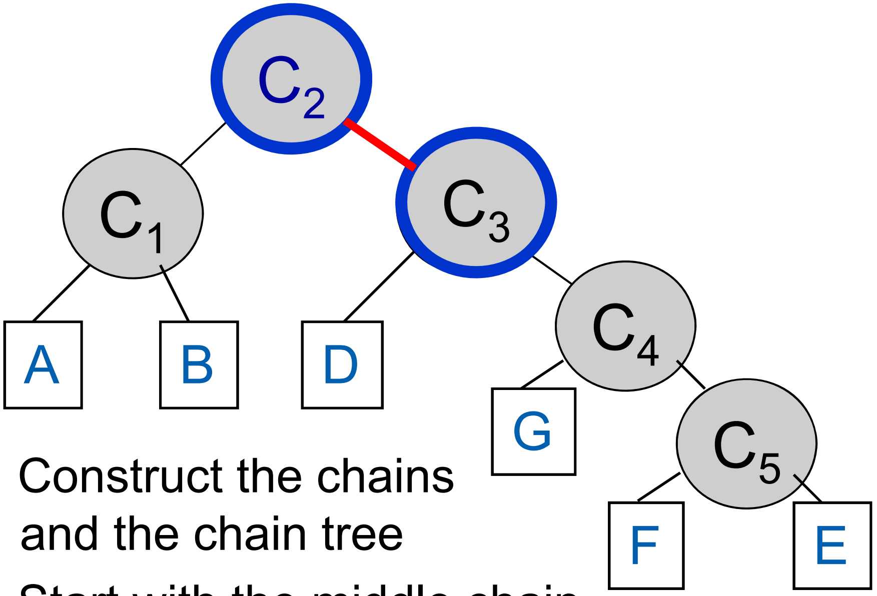




0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right (This is the position of *x* relatively to the whole chain)
4. Continue in L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

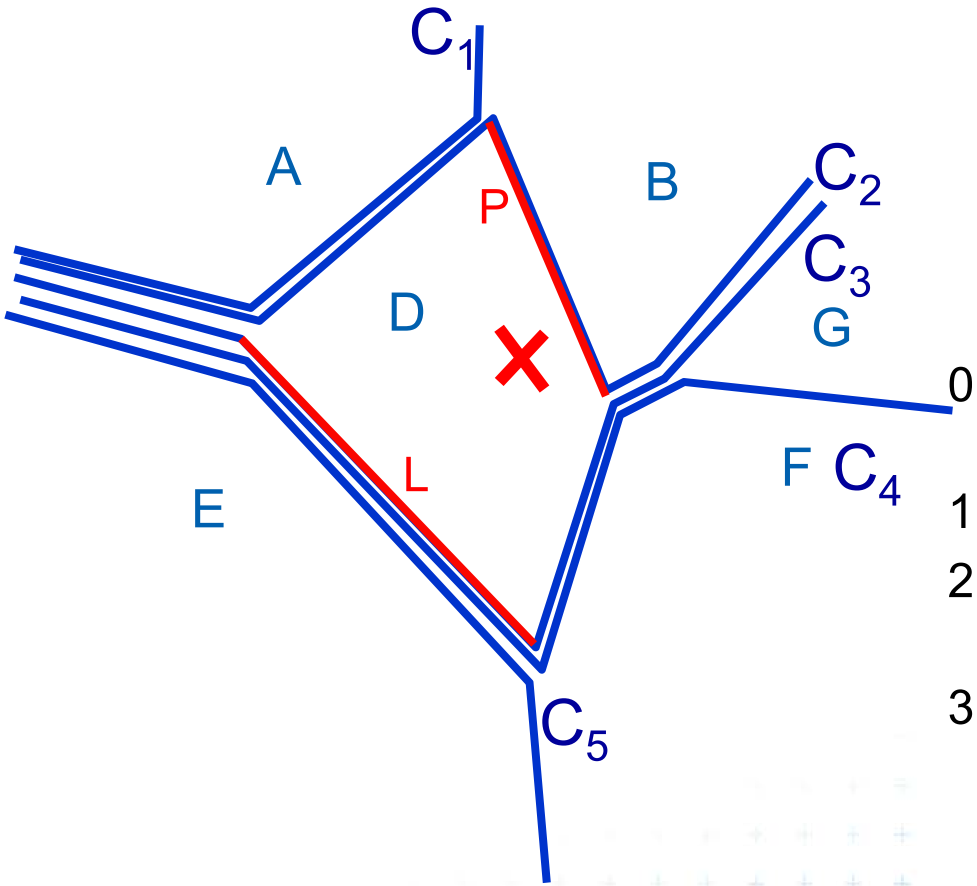
Monotone chain tree example

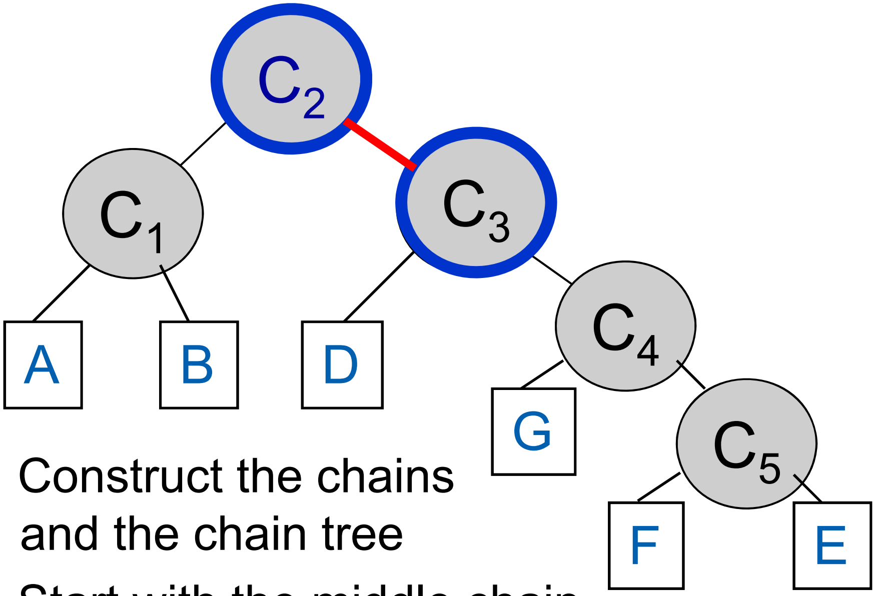




0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right
   (This is the position of *x* relatively to the whole chain)
4. Continue in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

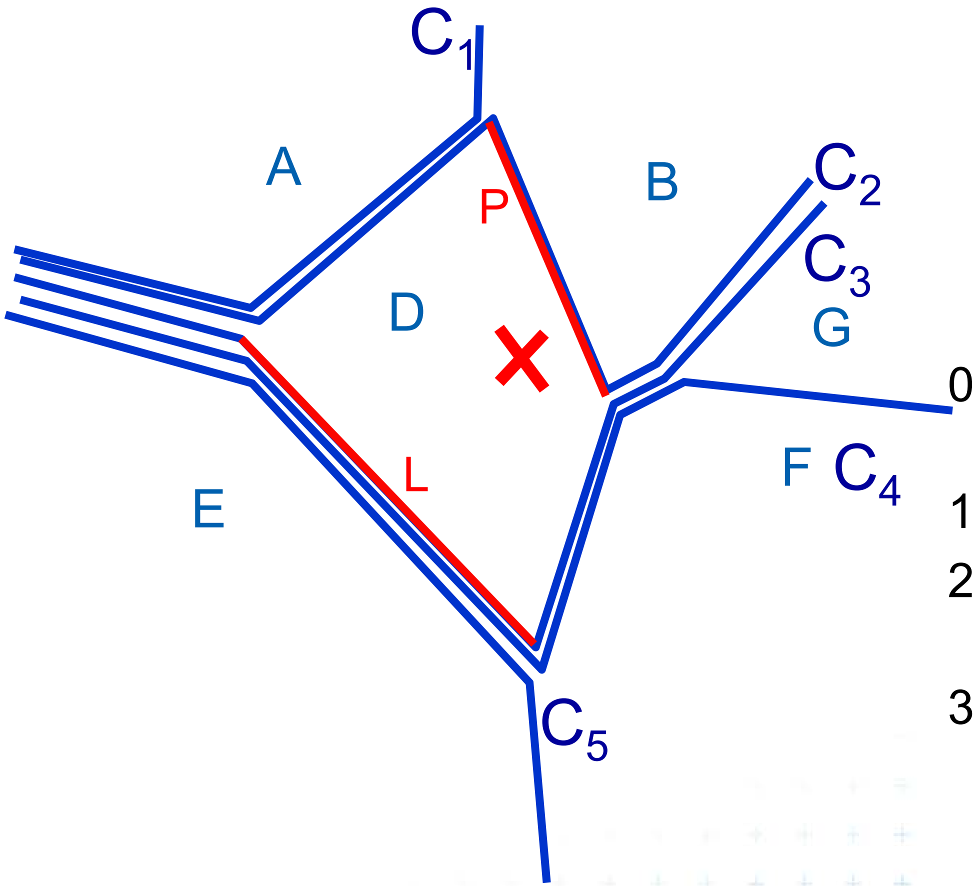
Monotone chain tree example

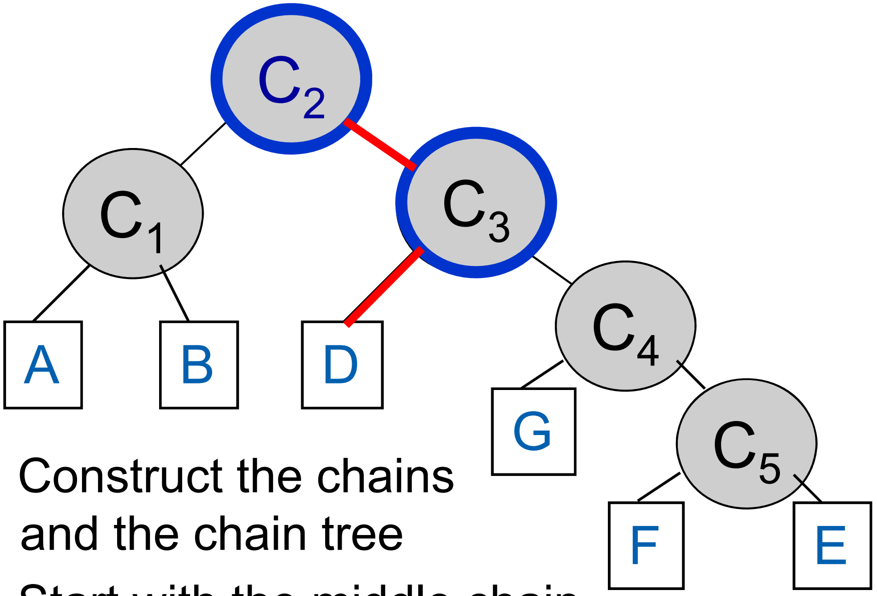




0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right
   (This is the position of *x* relatively to the whole chain)
4. Continue in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

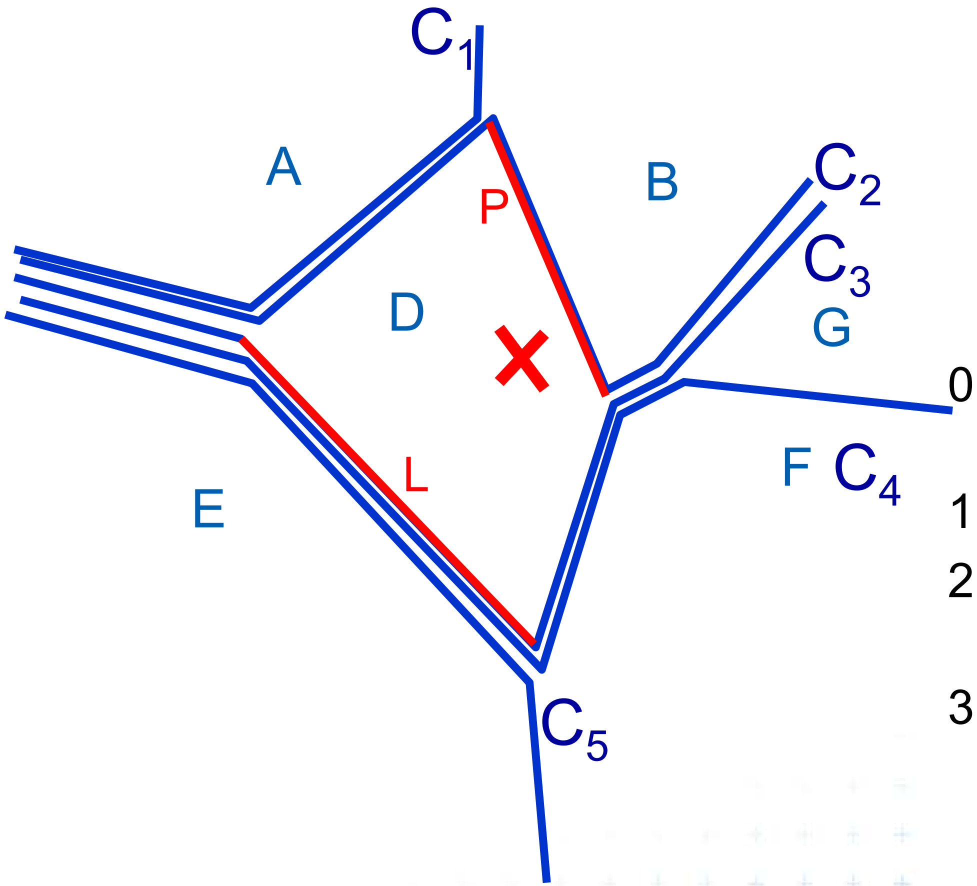
Monotone chain tree example

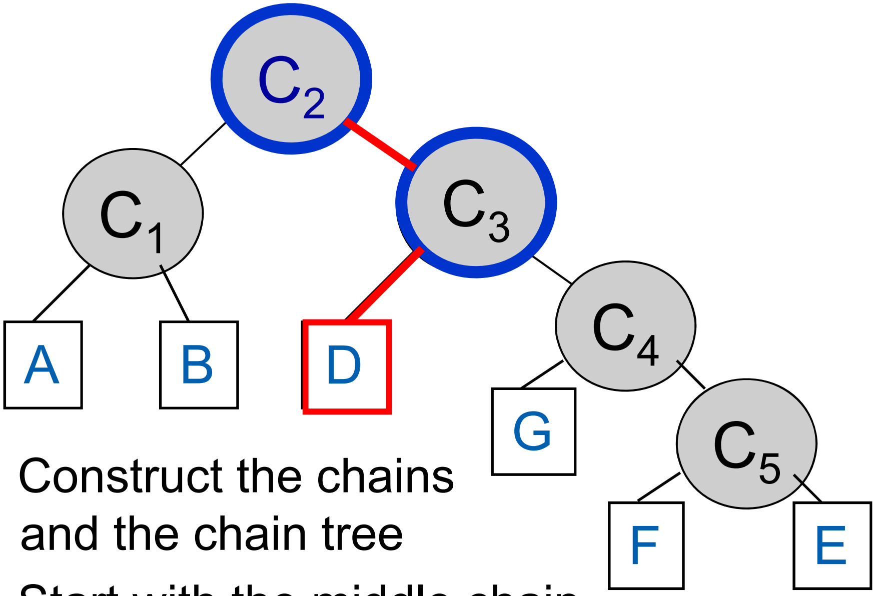




0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right (This is the position of *x* relatively to the whole chain)
4. Continue in L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

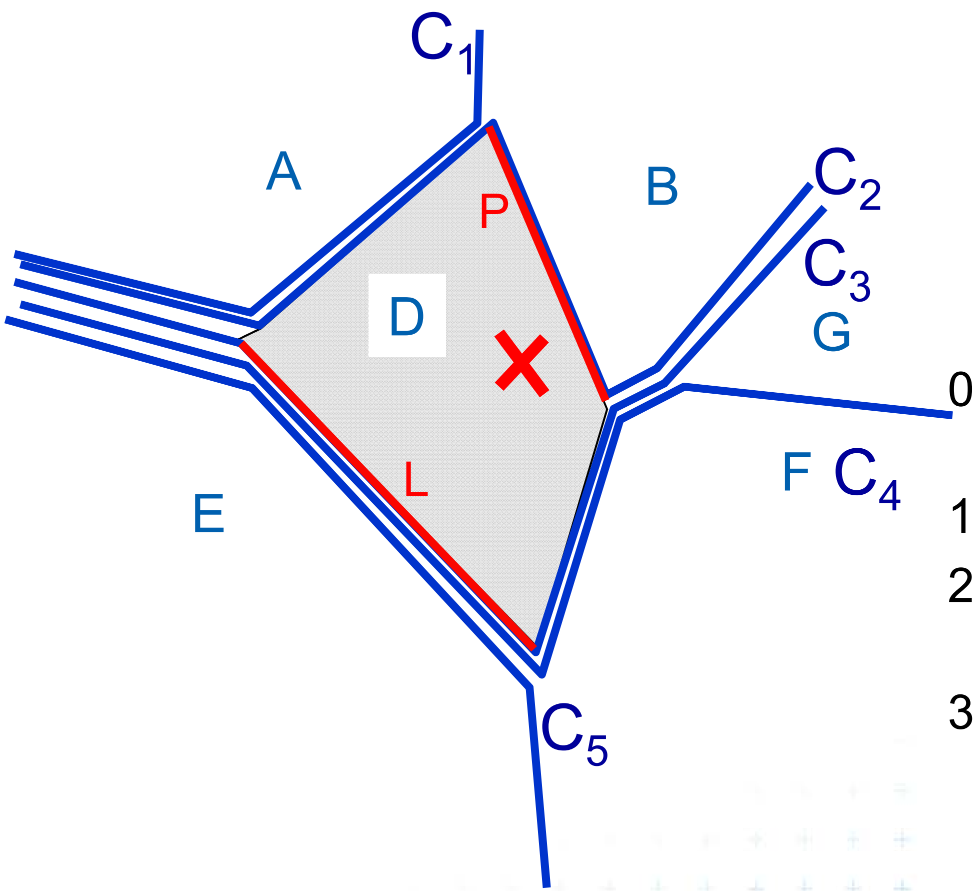
Monotone chain tree example

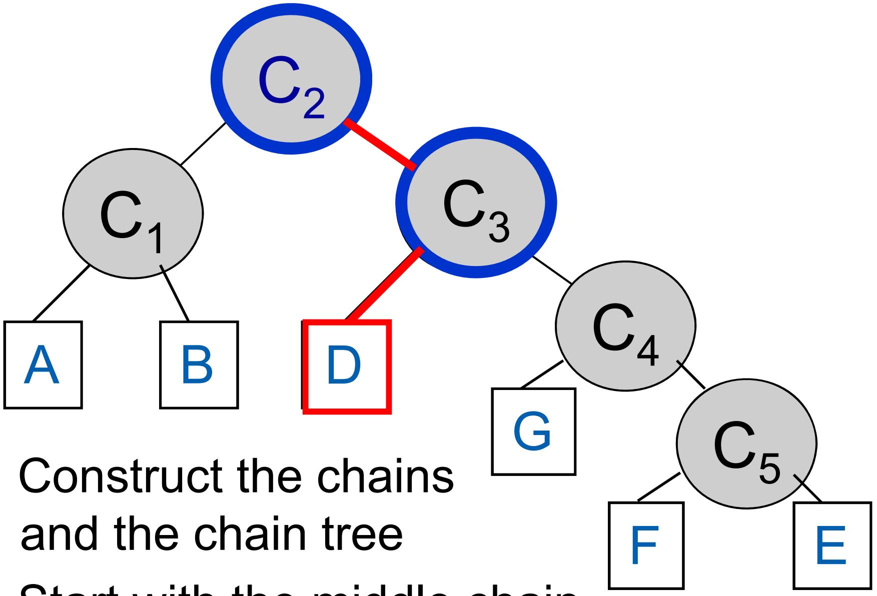




0. Construct the chains
   and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of
   the chain – determine the segment
3. Identify position of x in relation to the
   segment – Left or Right
   (This is the position of *x* relatively to the
   whole chain)
4.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

Monotone chain tree example

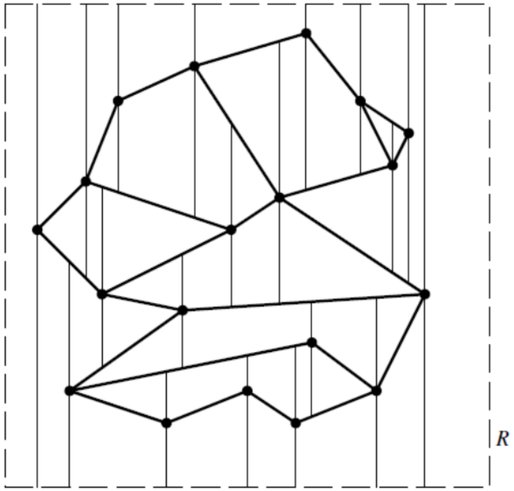




0. Construct the chains and the chain tree
1. Start with the middle chain
2. Find projection of *x* in the projection of the chain – determine the segment
3. Identify position of x in relation to the segment – Left or Right
   (This is the position of *x* relatively to the whole chain)
4. Continue in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

3. Trapezoidal map (TM) search

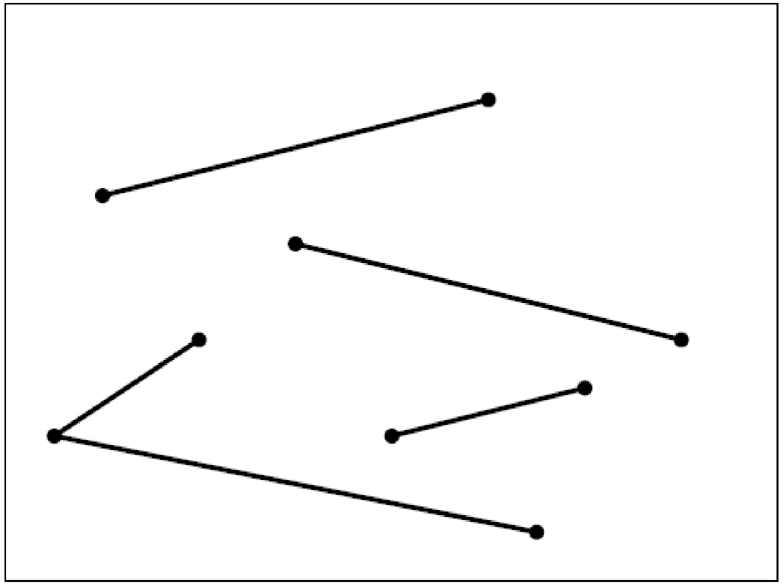
- The simplest and most practical known optimal algorithm
- Randomized algorithm with O(n) expected storage and O(log n) expected query time
- Expectation depends on the random order of segments during construction, not on the position of the segments
- TM is refinement of original subdivision
- Converts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



[Berg]

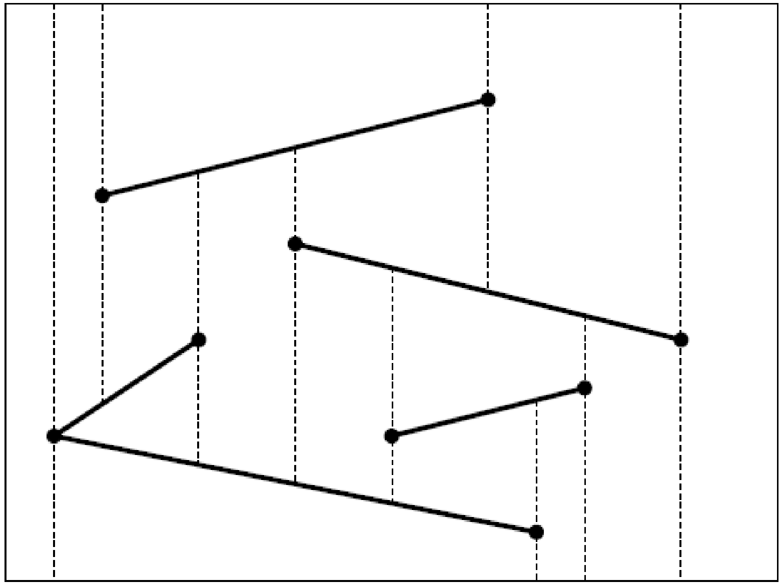
Trapezoidal map of line segments in general position

Input: individual segments *S*



Constru-
ction

Trapezoidal map *T*

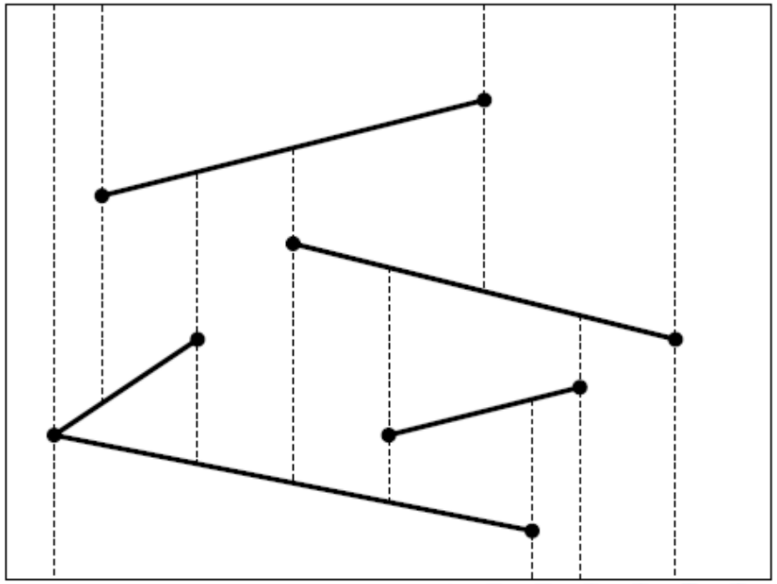


[Mount]

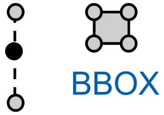
- They do not intersect, except in endpoints
- No vertical segments
- No 2 distinct endpoints with the same x-coordinate

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

Trapezoidal map of line segments in general position


[Mount]

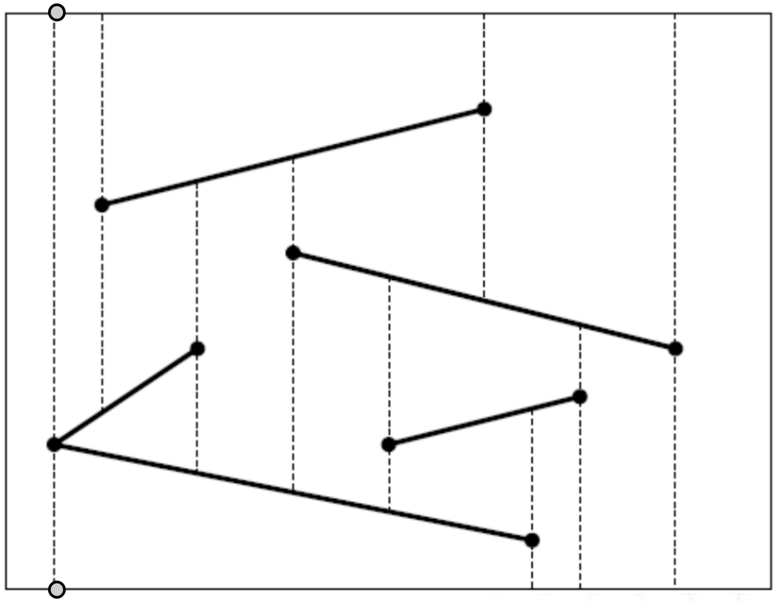
- Faces are trapezoids Δ with vertical sides
- Given n segments, TM has
  - at most 6n+4 vertices
  - at most 3n+1 trapezoids
- Proof:
  - each point 2 bullets -> 1+2 points
  - 2n endpoints * 3 + 4 = 6n+4 vertices



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

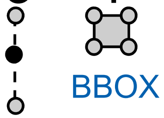
Trapezoidal map of line segments in general position

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



[Mount]

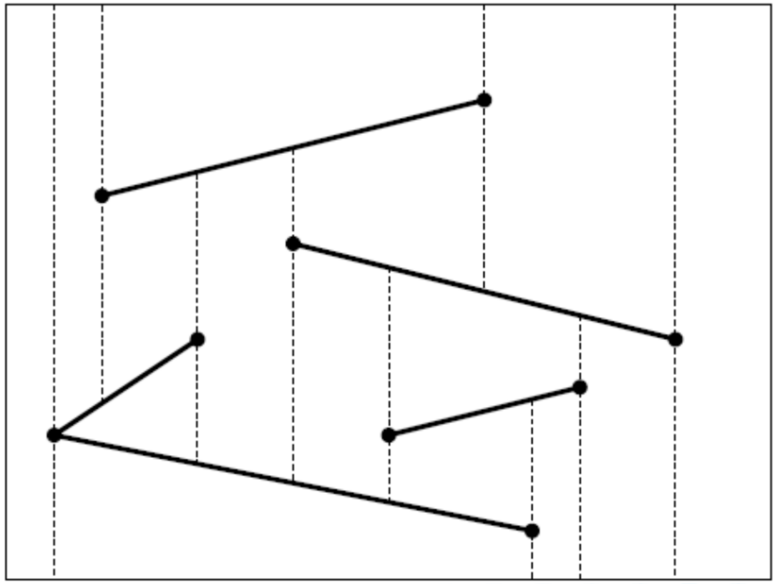
- Proof:
  - each point 2 bullets -> 1+2 points
  - 2n endpoints * 3 + 4 = 6n+4 vertices



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

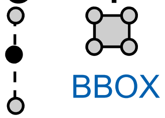
Trapezoidal map of line segments in general position

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



[Mount]

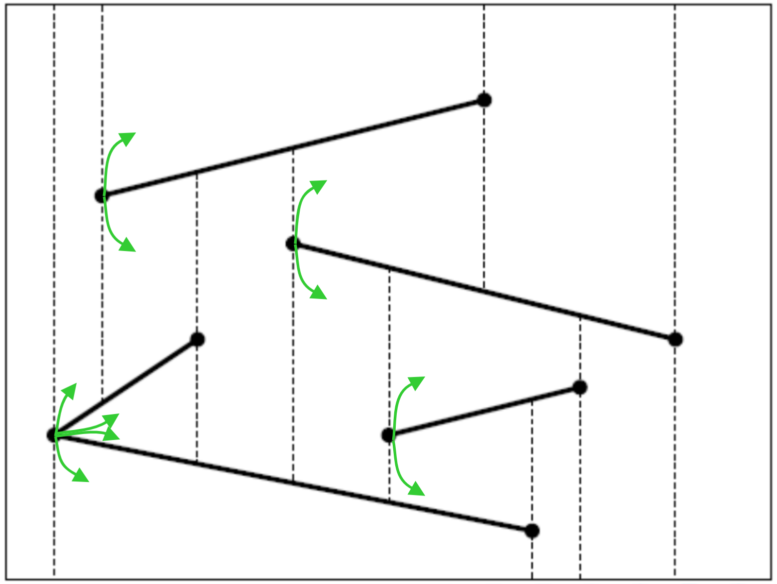
- Proof:
  - each point 2 bullets -> 1+2 points
  - 2n endpoints * 3 + 4 = 6n+4 vertices



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

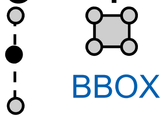
Trapezoidal map of line segments in general position

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



[Mount]

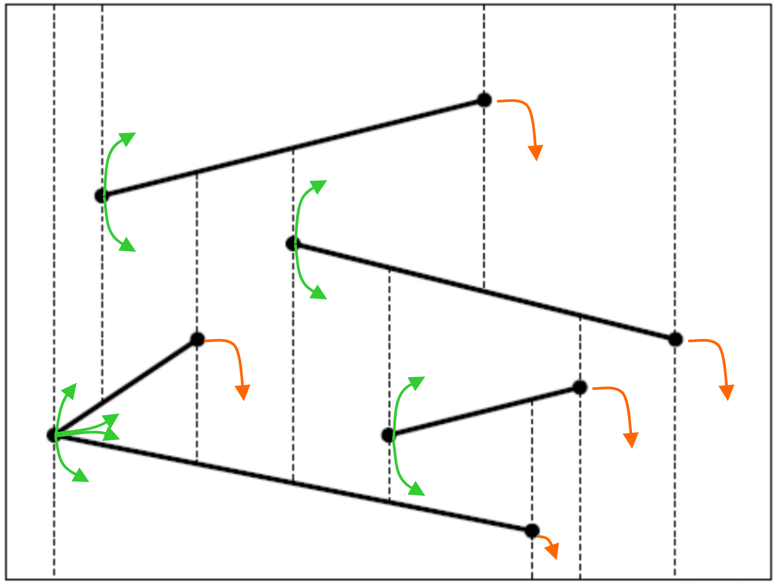
- Proof:
  - each point 2 bullets -> 1+2 points
  - 2n endpoints * 3 + 4 = 6n+4 vertices



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

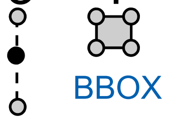
Trapezoidal map of line segments in general position

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



[Mount]

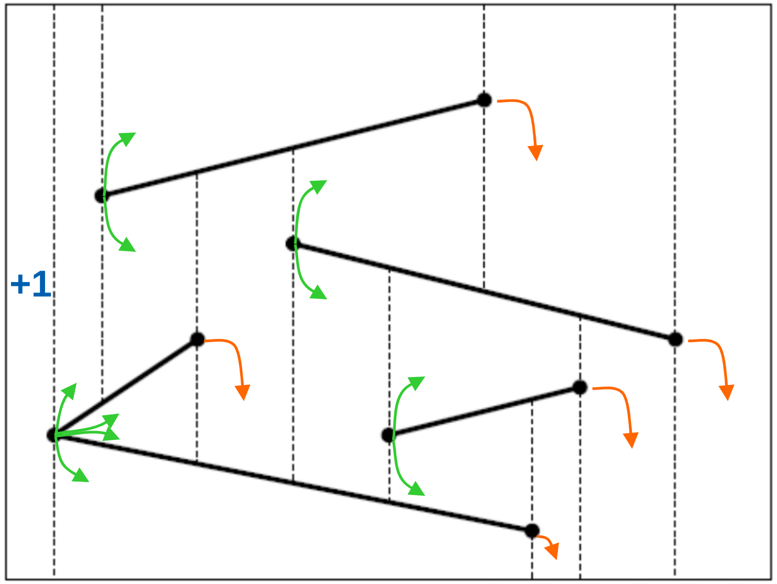
- Proof:
  - each point 2 bullets -> 1+2 points
  - 2n endpoints * 3 + 4 = 6n+4 vertices



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

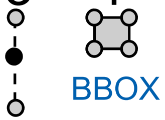
Trapezoidal map of line segments in general position

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



[Mount]

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

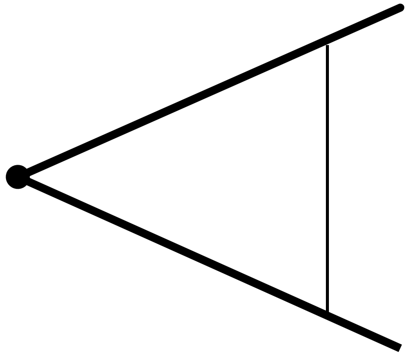


  - start point –> max 2 trapezoids
  - end point –> 1 trapezoid
  - 3 * (n segments) + 1 left ∆ => max 3n+1 ∆

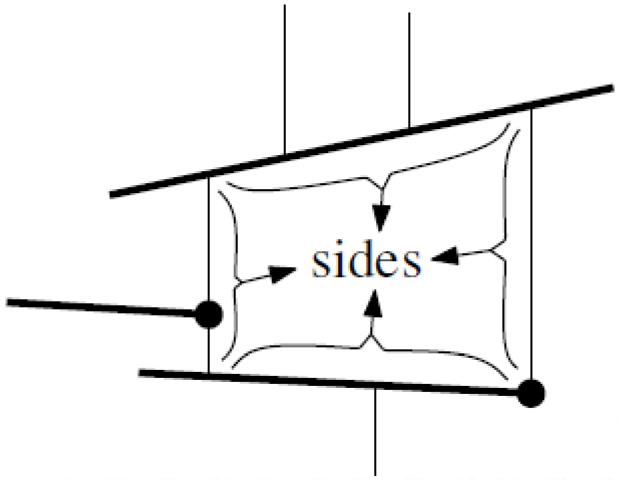
Trapezoidal map of line segments in general position

Each face has

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



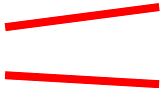
One vertical side



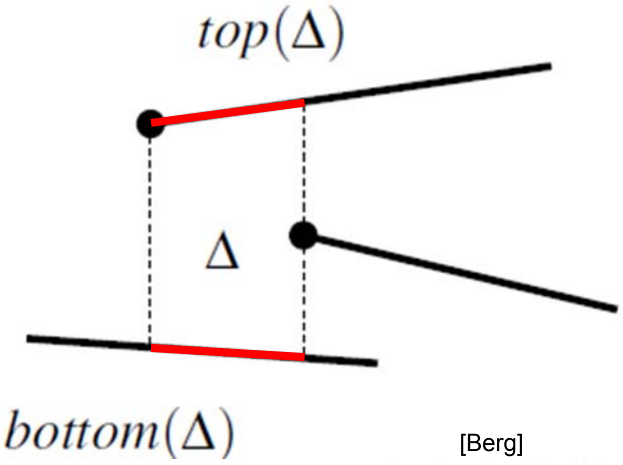
Two vertical sides

[Berg]

Two non-vertical sides

Non-vertical side 

- is contained in one of the segments of set *S*
- or in the horizontal edge of bounding rectangle *R*



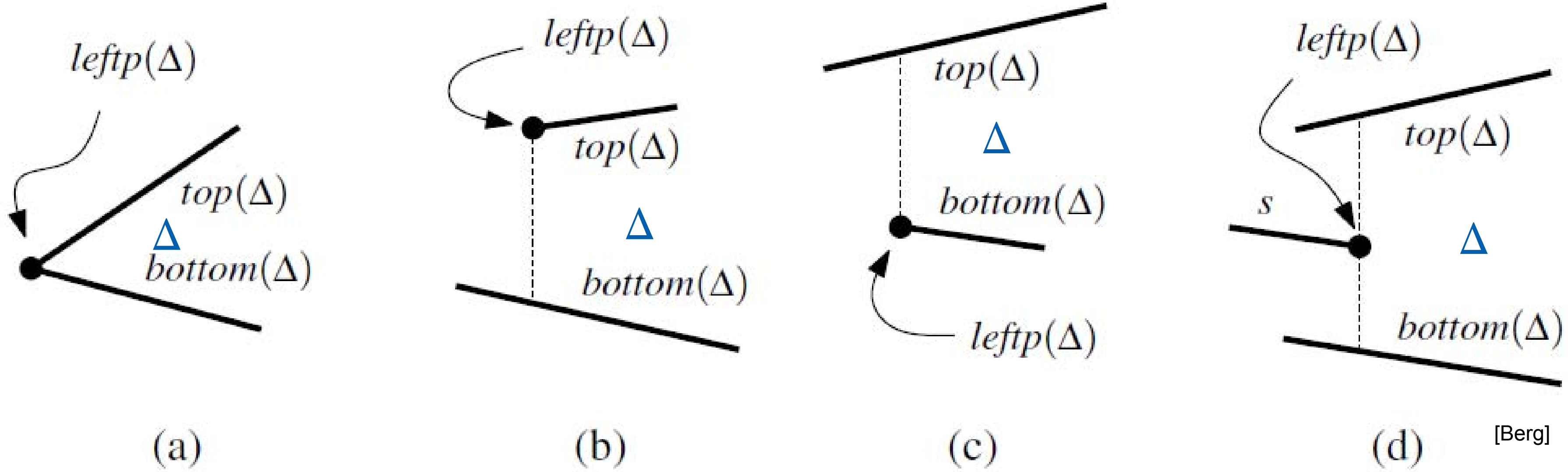
[Berg]

segments:

top(Δ) - bounds from above

bottom(Δ) - bounds from below

Vertical sides – left vertical side of Δ



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

(a) common left point *p* itself

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

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

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

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

Vertical sides - summary

Vertical edges are defined by segment endpoints

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

leftp(Δ) is

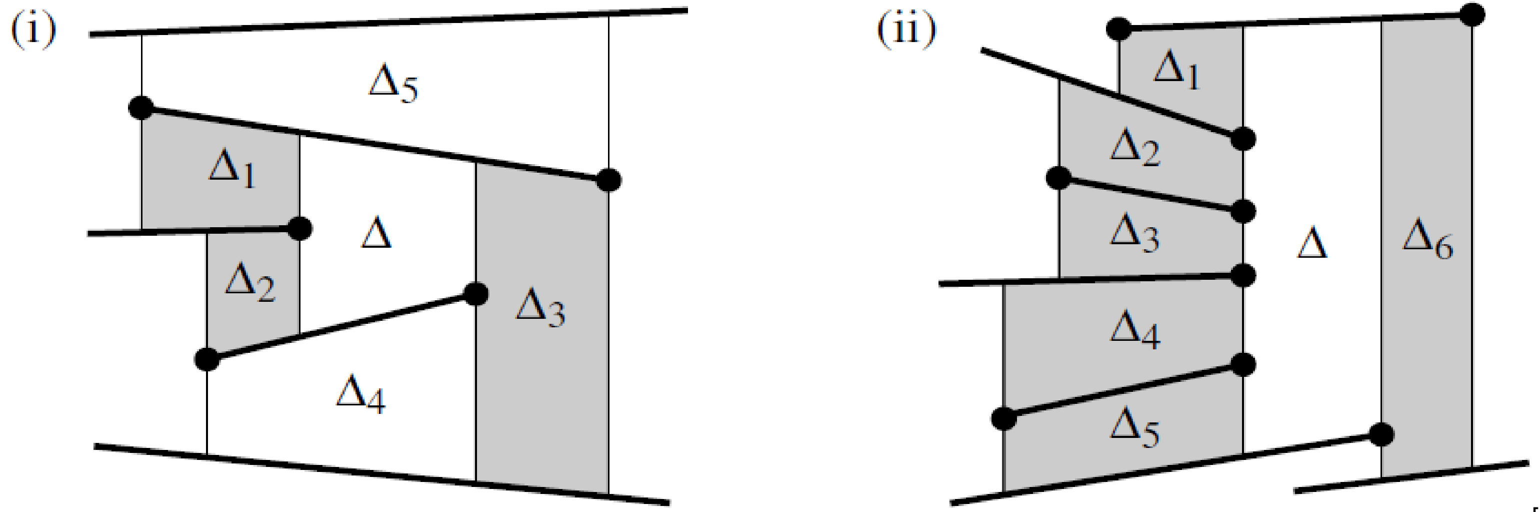
- the left endpoint of *top*() or *bottom*() or both (c, b, a)
- the right point of a third segment (d)
- the lower left corner of the bounding rectangle R (e)

Trapezoid Δ

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

Adjacency of trapezoids segments in general position

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

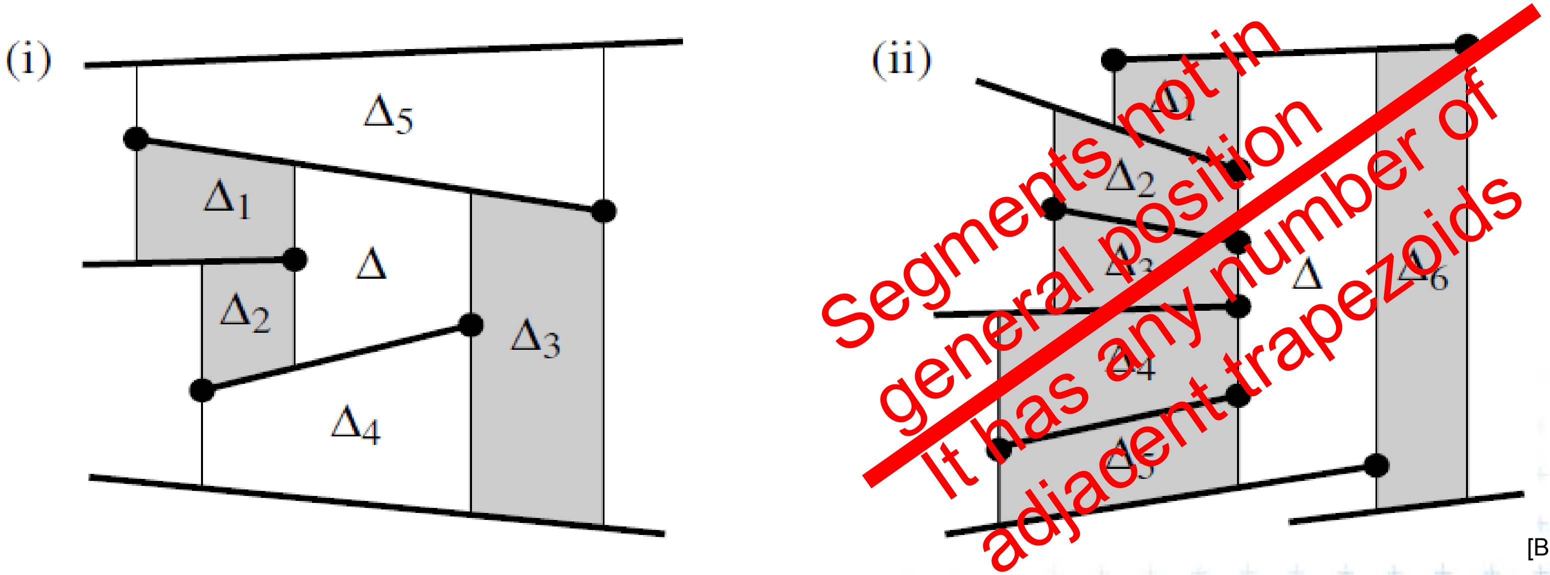


[Berg]

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

Adjacency of trapezoids segments in general position

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



Segments not in general position

It has any number of adjacent trapezoids

[Berg]

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

Representation of the trapezoidal map *T*

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

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

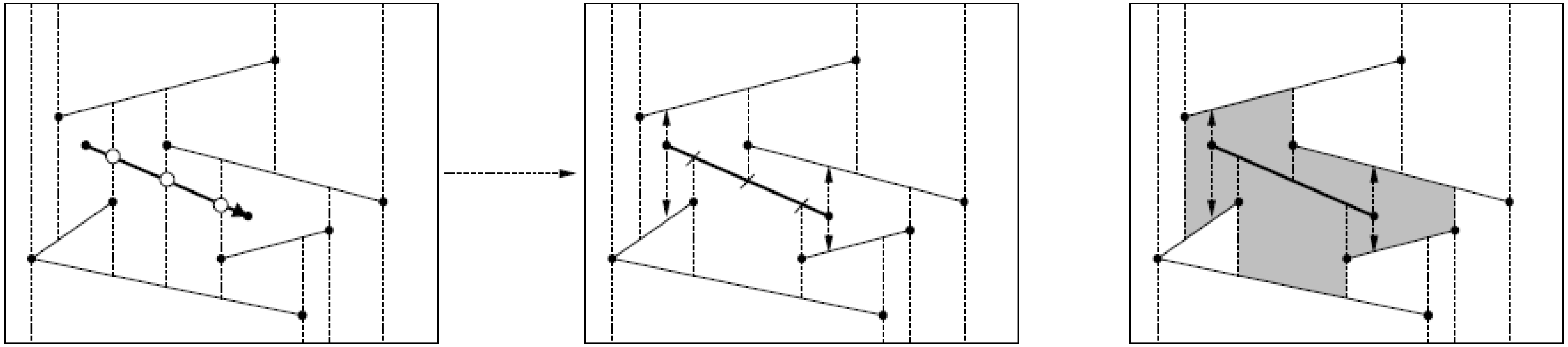
- Does not store the geometry explicitly!
- Geometry of trapezoids is computed in O(1)

Construction of trapezoidal map

- Randomized incremental algorithm

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

[Mount]



Locate left endpoint and determine intersections

Shoot new bullet paths and trim intersecting rays

Newly created trapezoids

Trapezoidal map point location

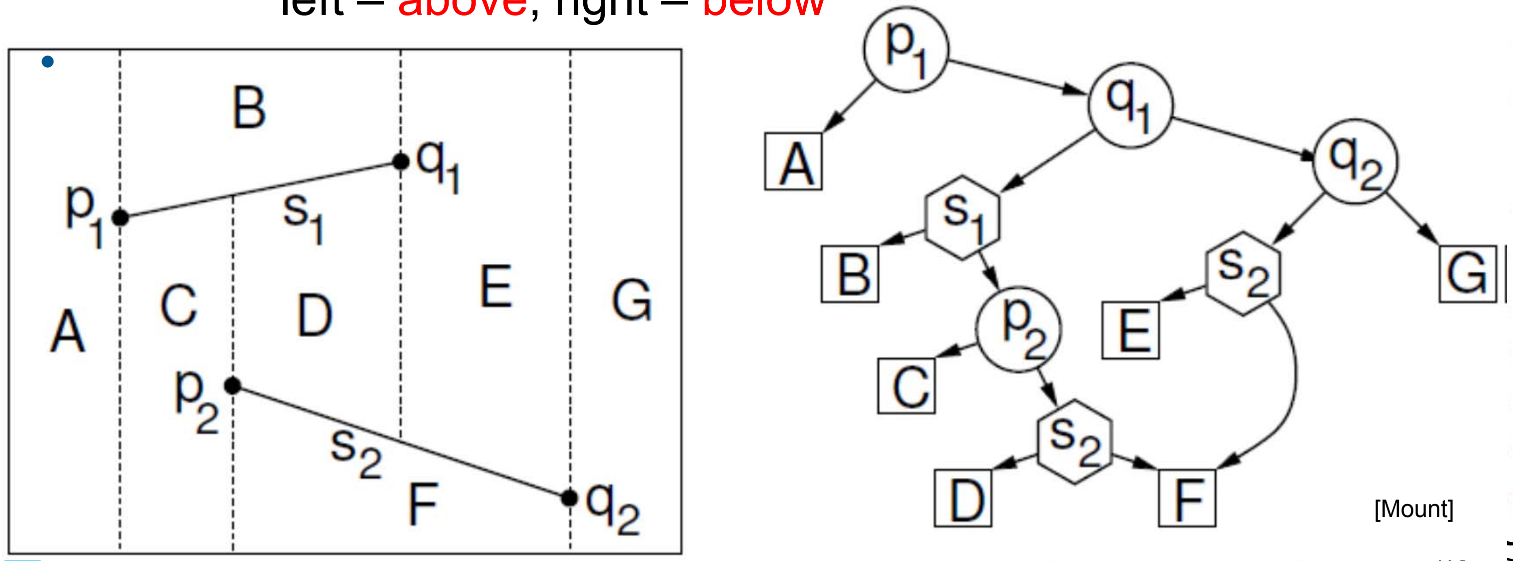
- While creating the trapezoidal map *T* construct the *Point location data structure D*
- Query this data structure

Point location data structure D

- Rooted directed acyclic graph (not a tree!!)
  - Leaves $\boxed{X}$ – trapezoids, each appears exactly once
  - Internal nodes – 2 outgoing edges, guide the search

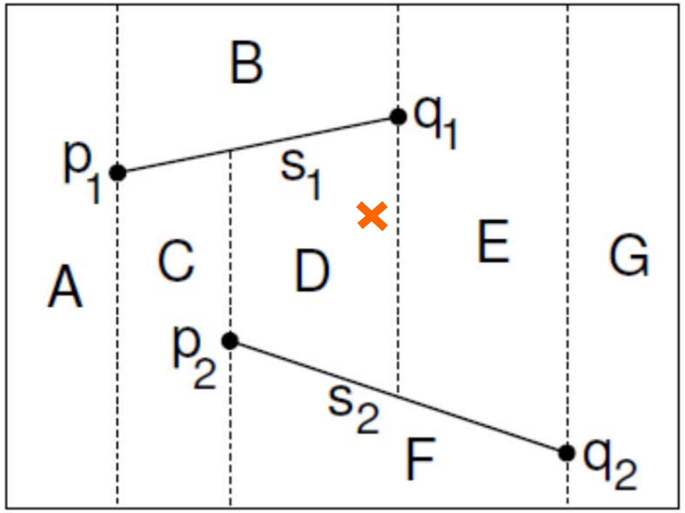
p_1 x-node – x-coord x_0 of segment start- or end-point
left child lies left of vertical line $x=x_0$
right child lies right of vertical line $x=x_0$
– used first to detect the vertical slab

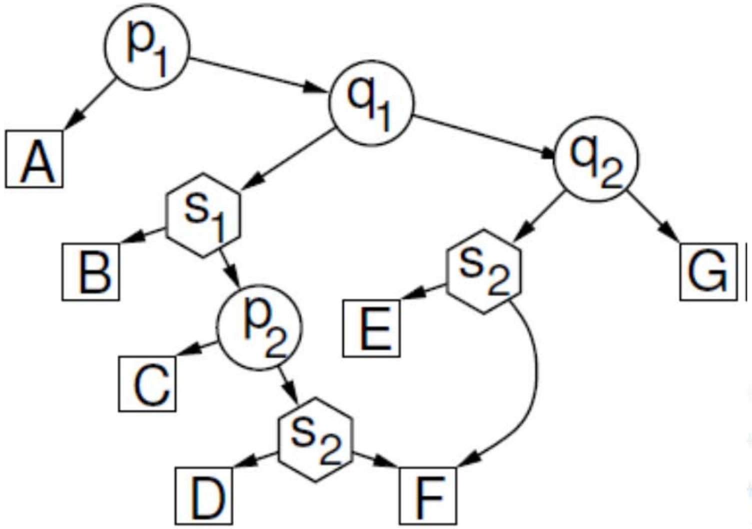
s_1 y-node – pointer to the line segment of the subdivision (not only its y!!!)
left – above, right – below



[Mount]

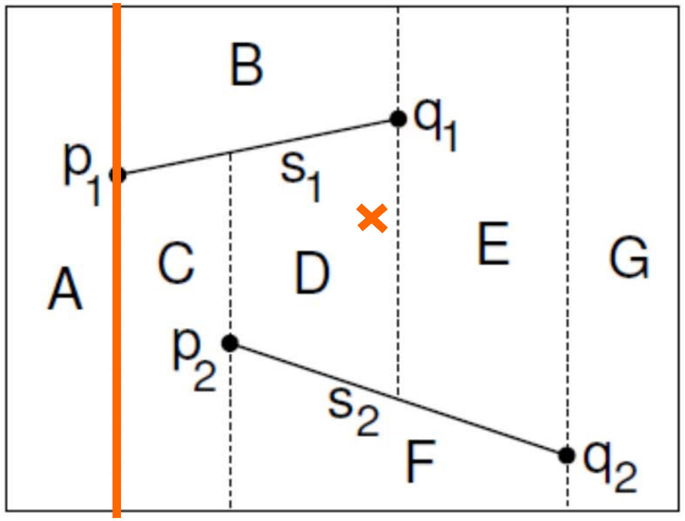
TM search example

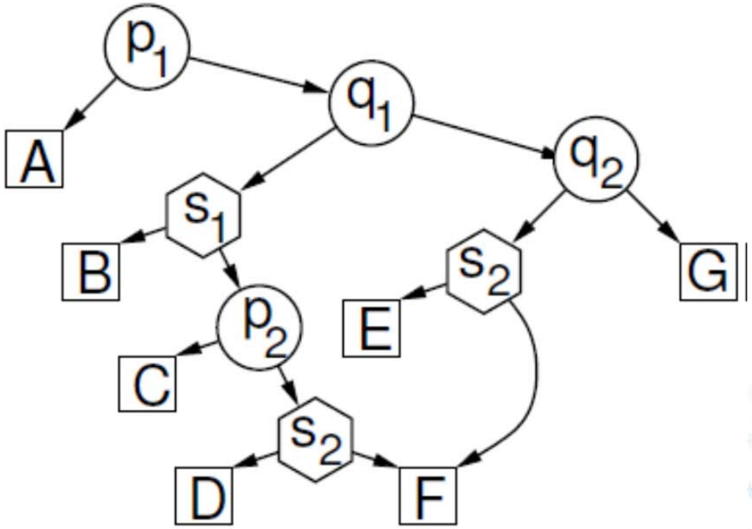




[Mount]

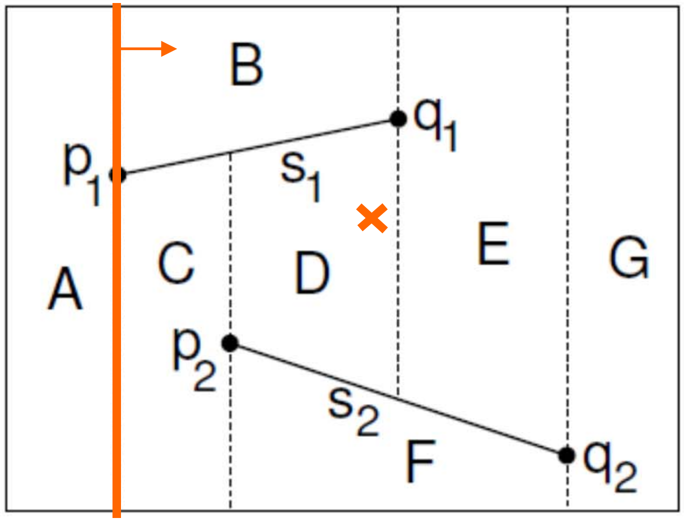
TM search example

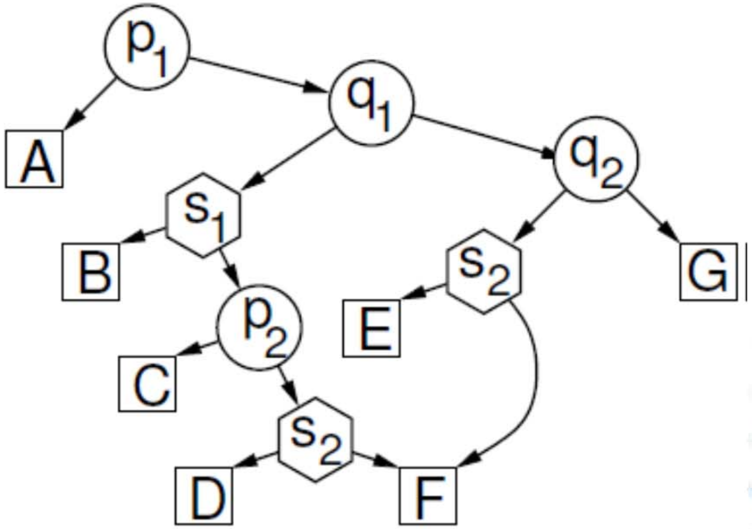




[Mount]

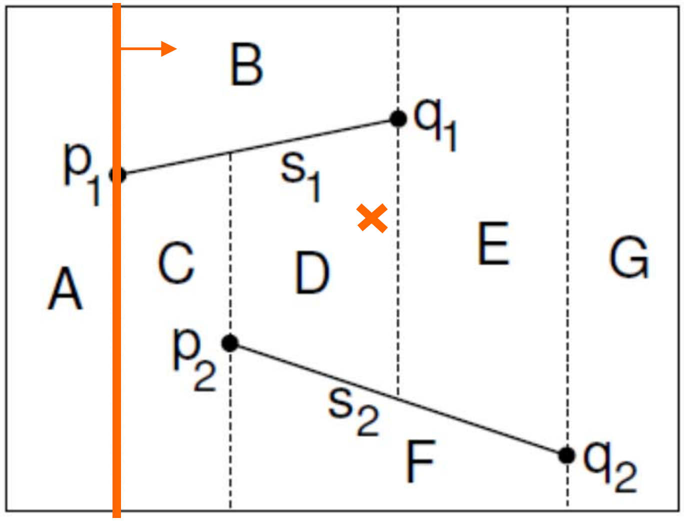
TM search example

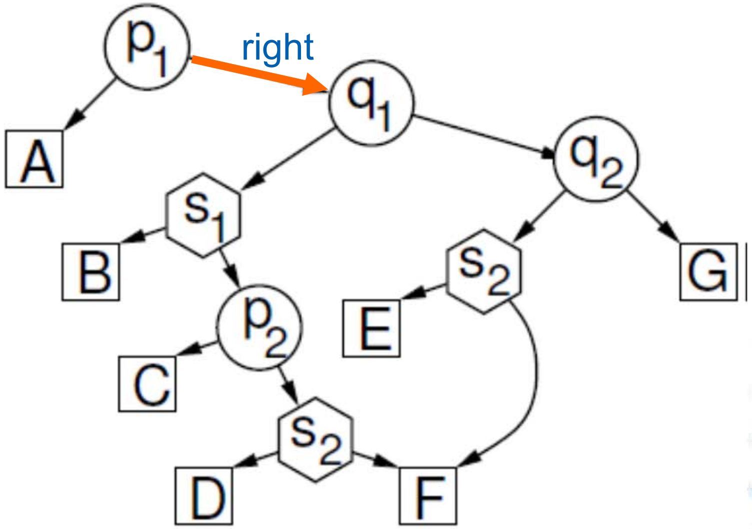




[Mount]

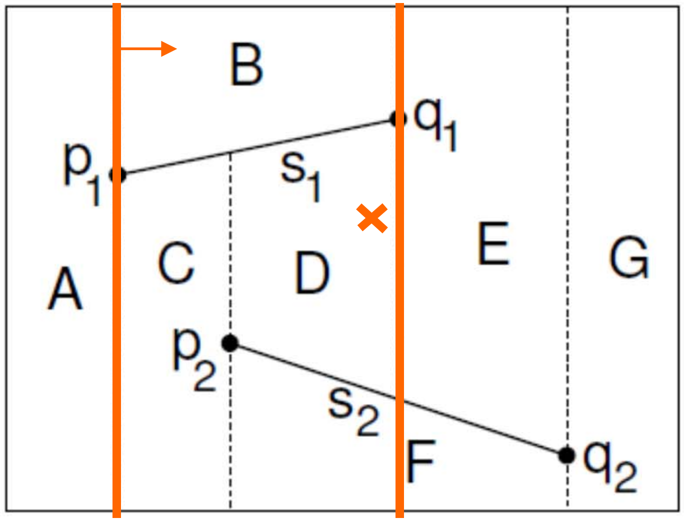
TM search example

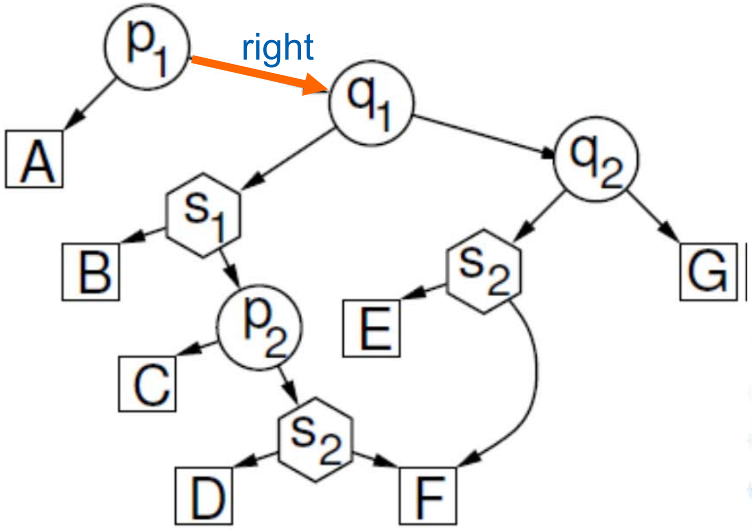




[Mount]

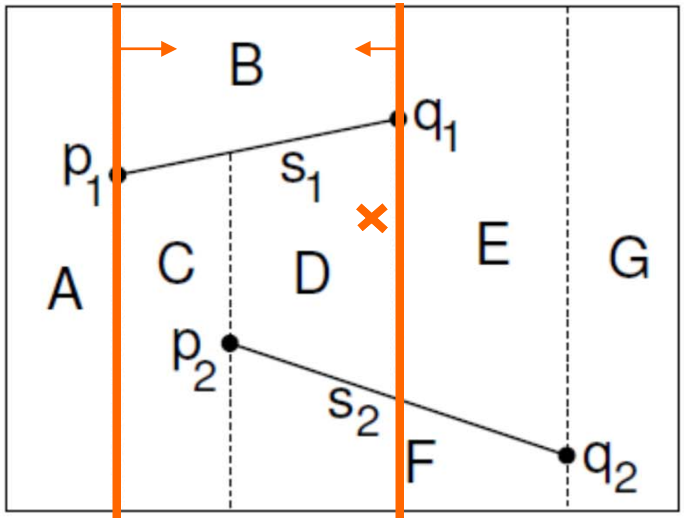
TM search example

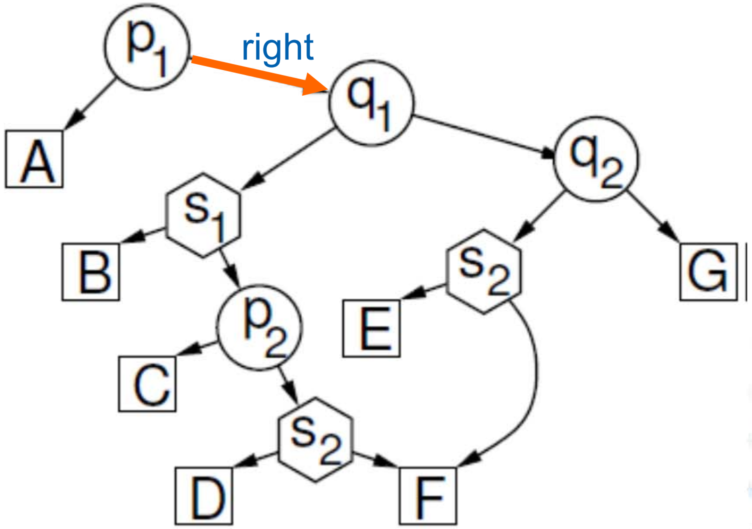




[Mount]

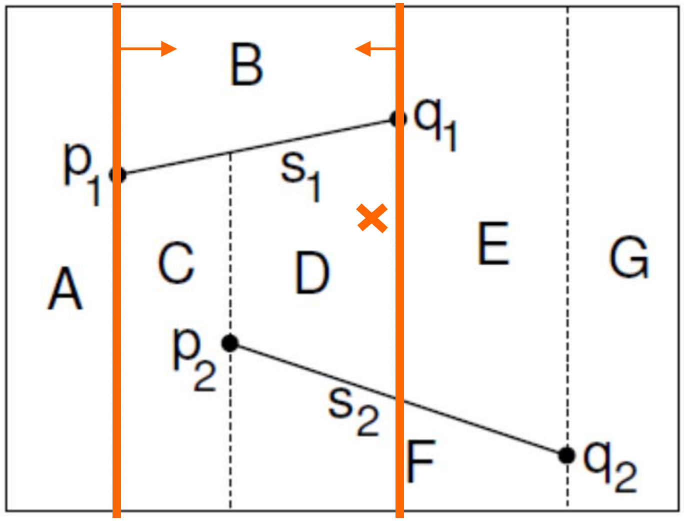
TM search example

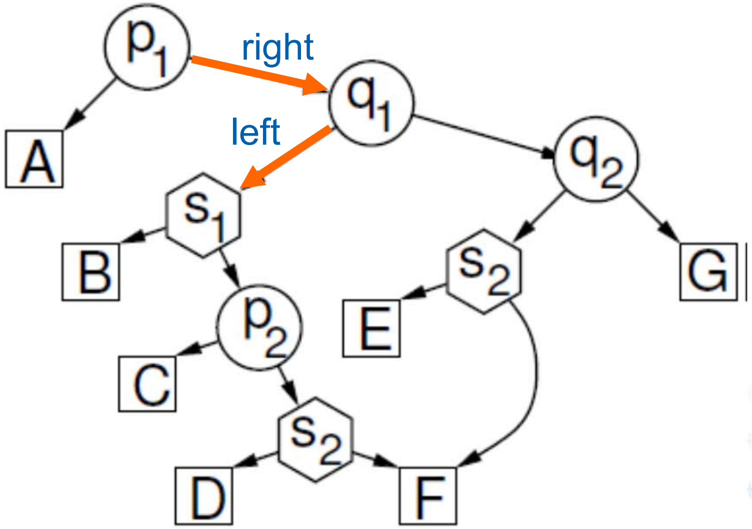




[Mount]

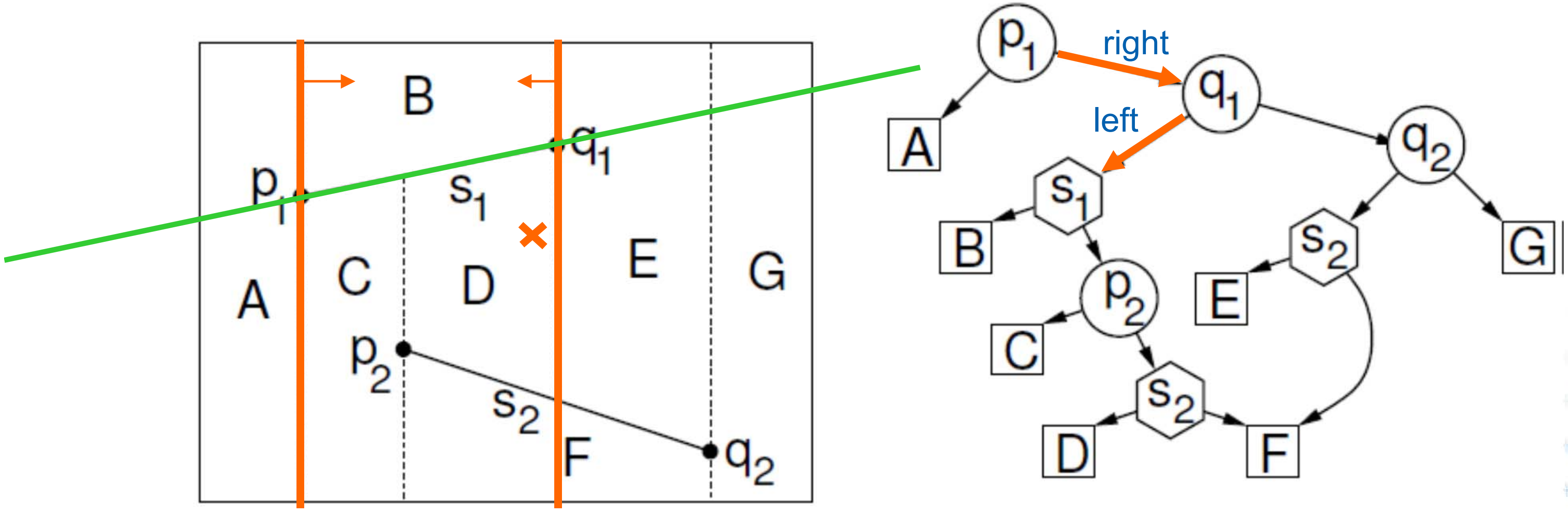
TM search example





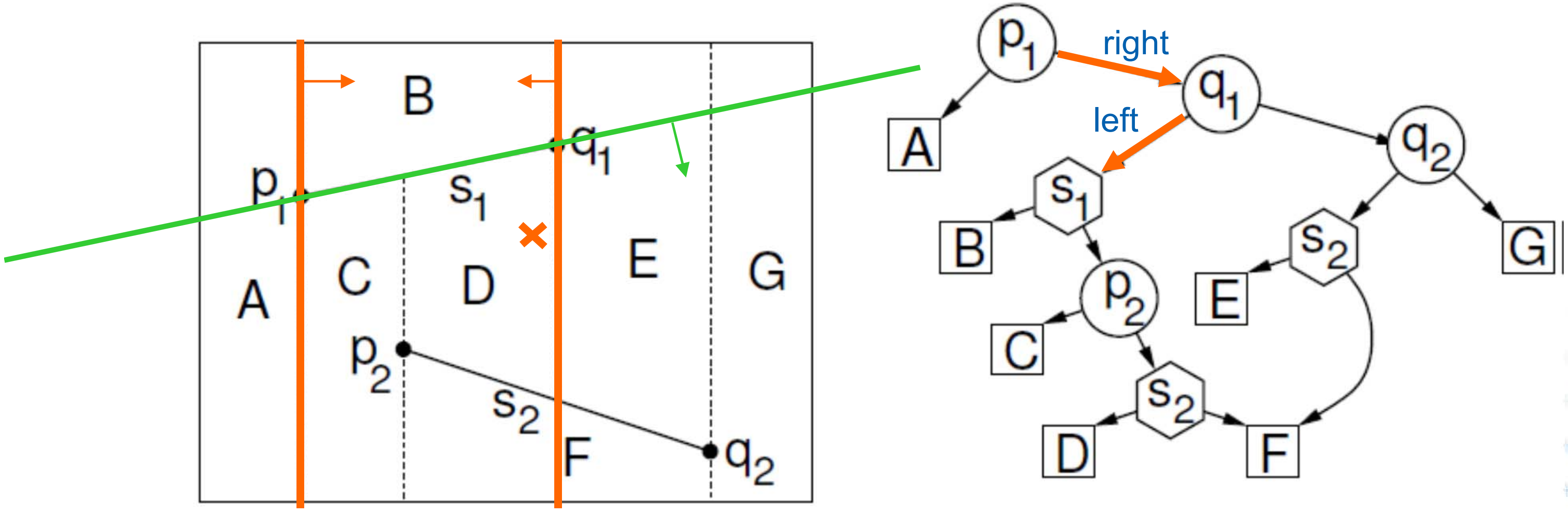
[Mount]

TM search example



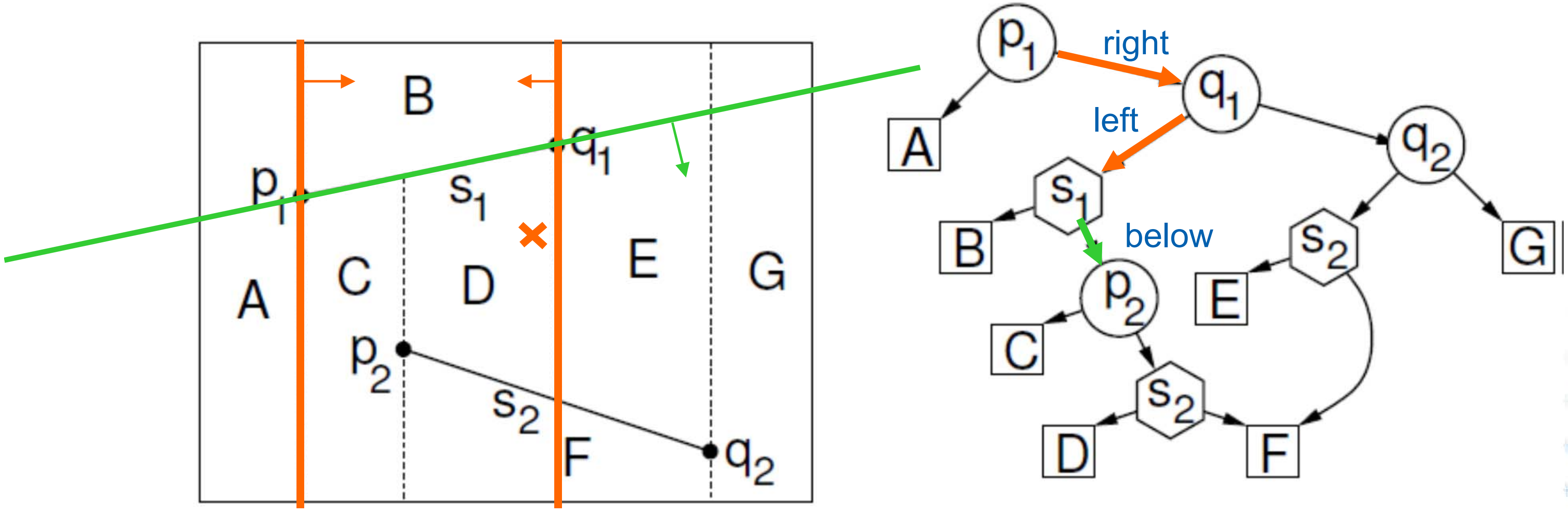
[Mount]

TM search example



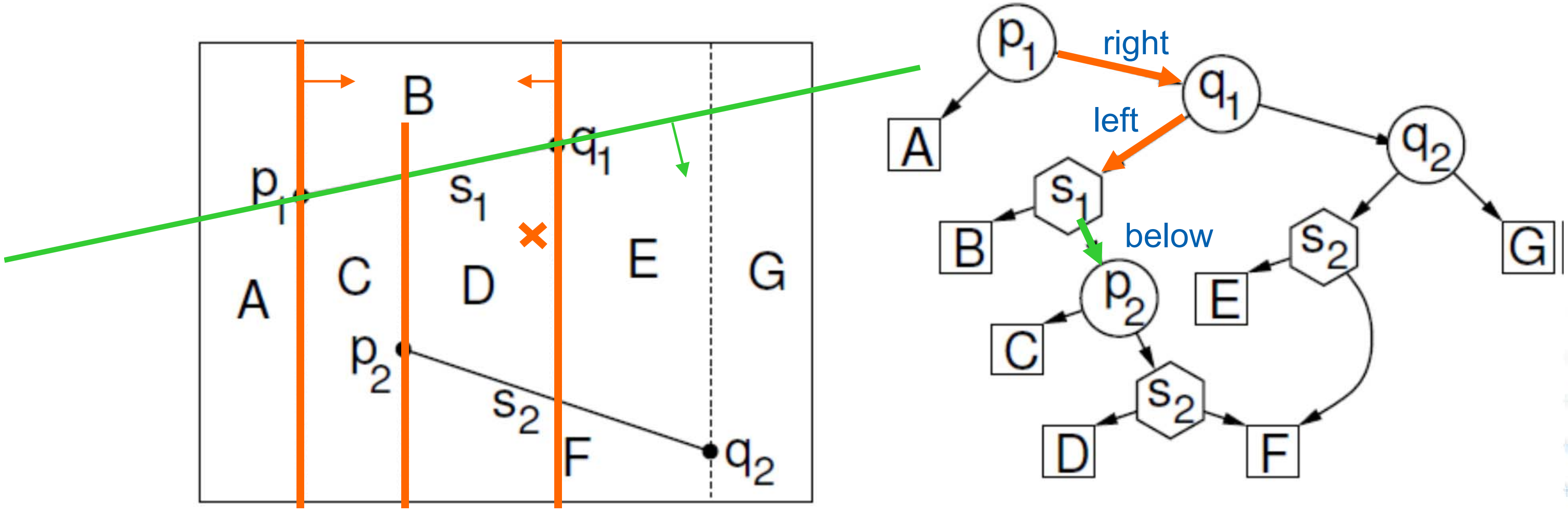
[Mount]

TM search example



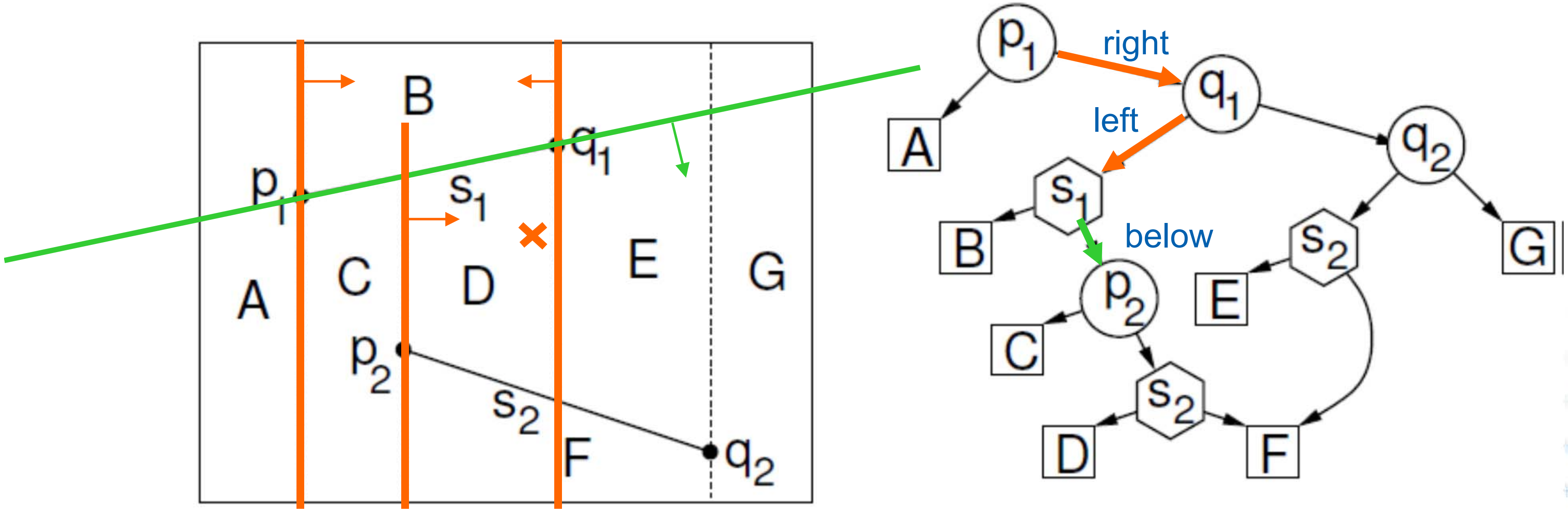
[Mount]

TM search example



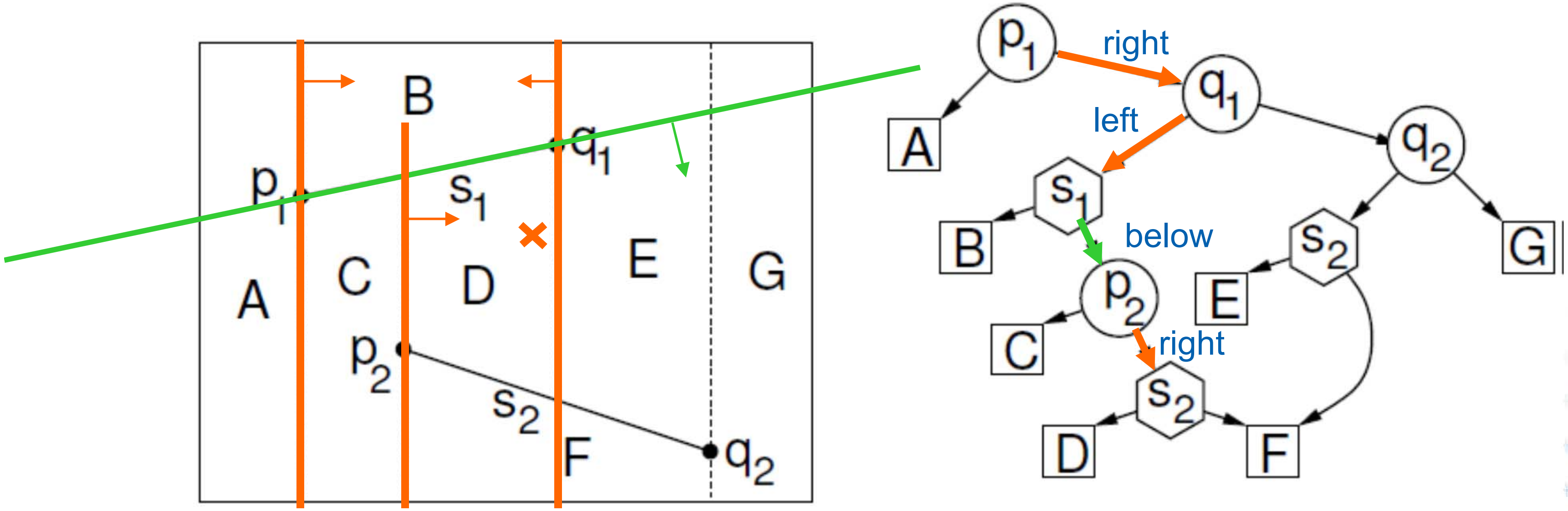
[Mount]

TM search example



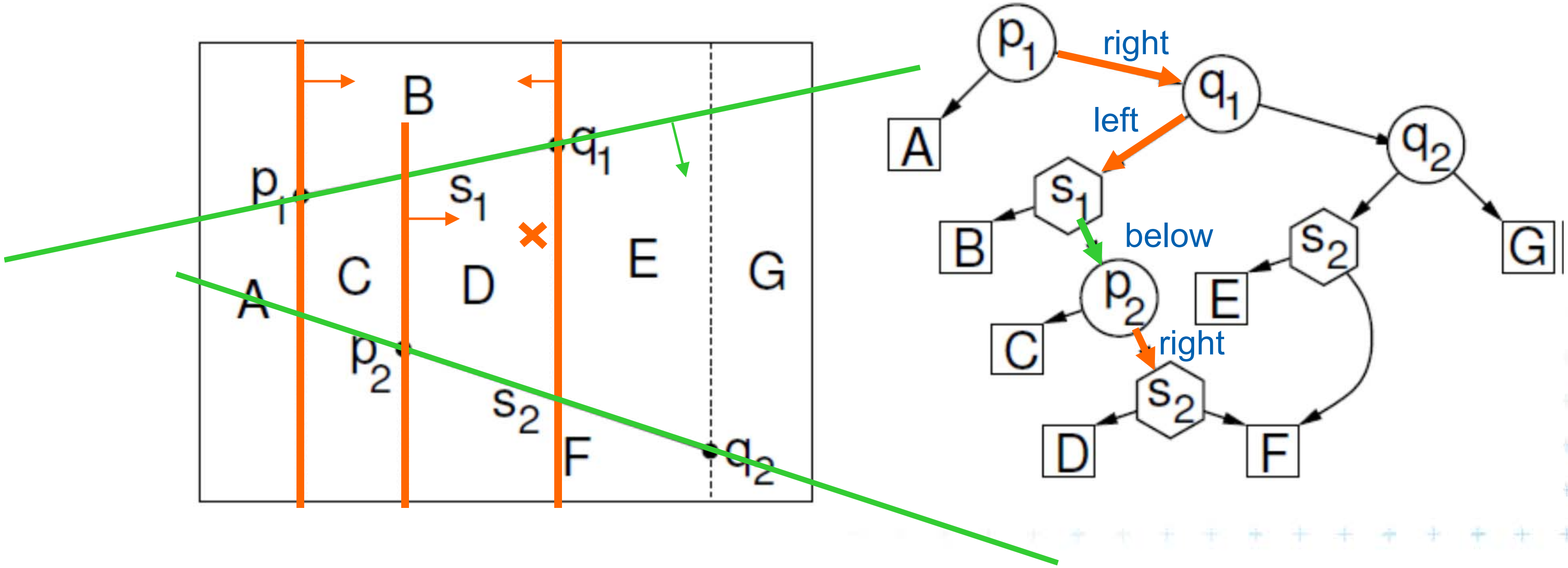
[Mount]

TM search example



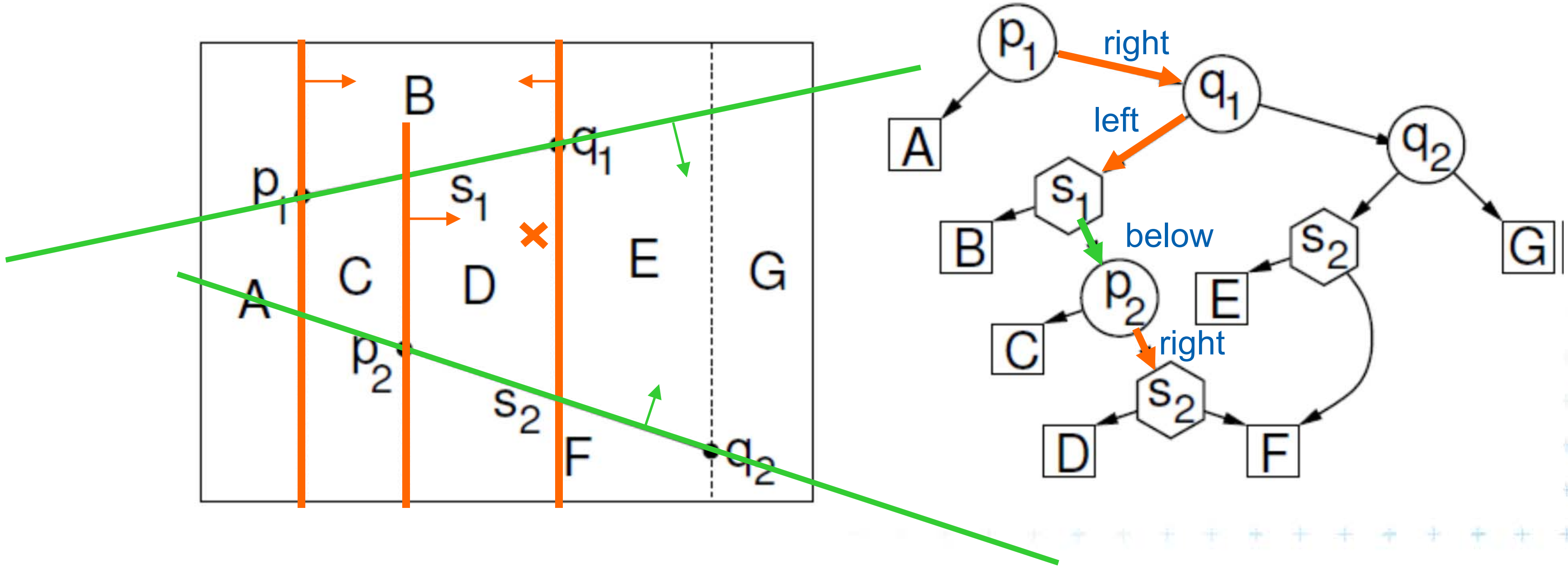
[Mount]

TM search example



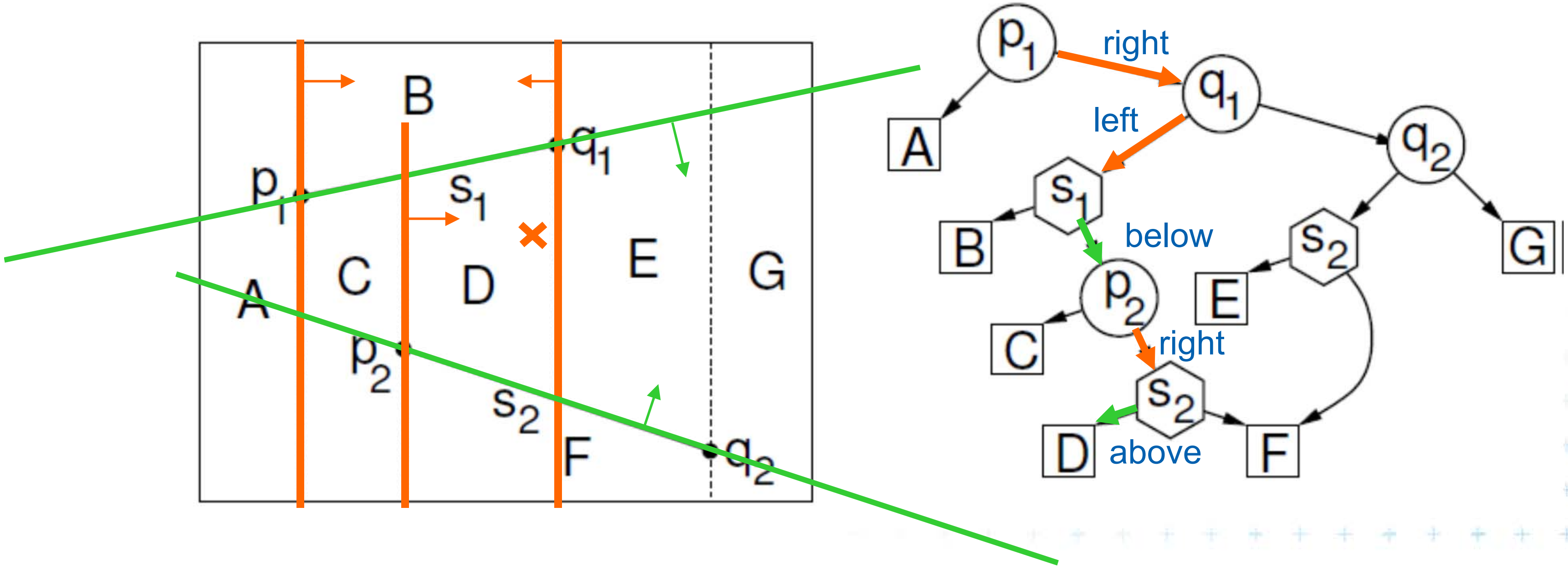
[Mount]

TM search example



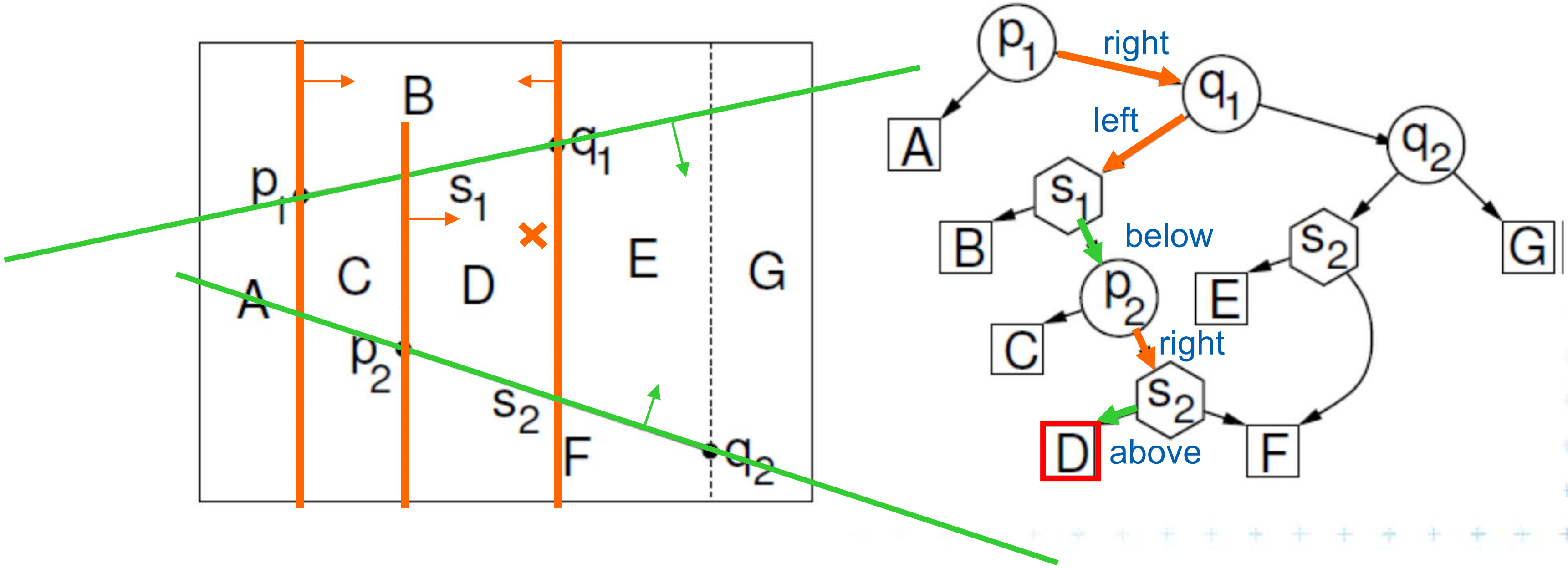
[Mount]

TM search example



[Mount]

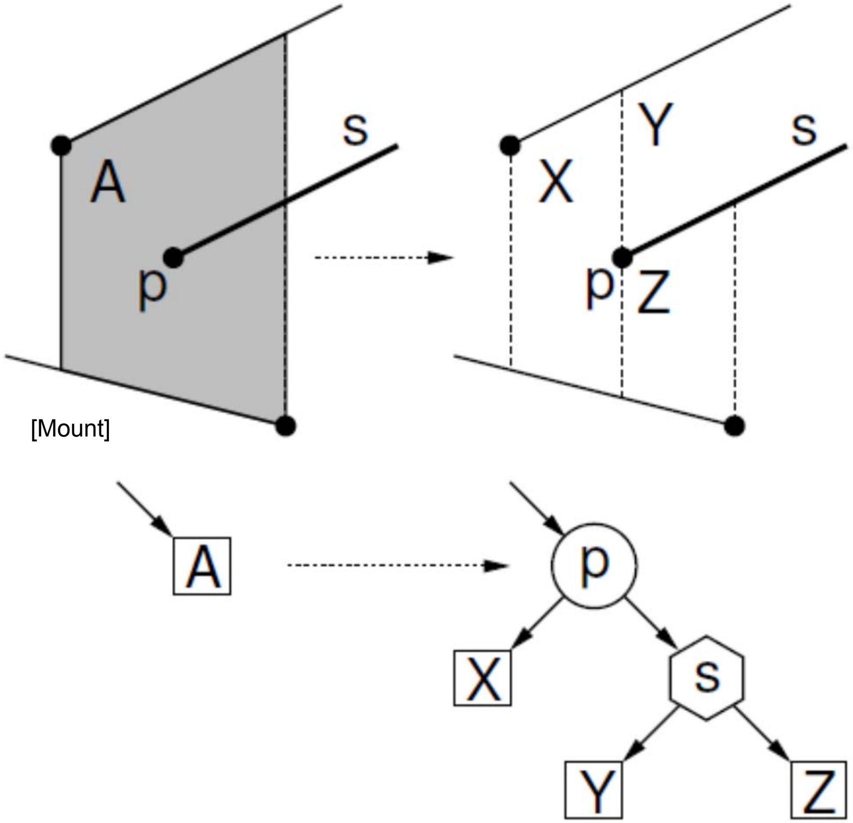
TM search example



[Mount]

Construction – addition of a segment

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



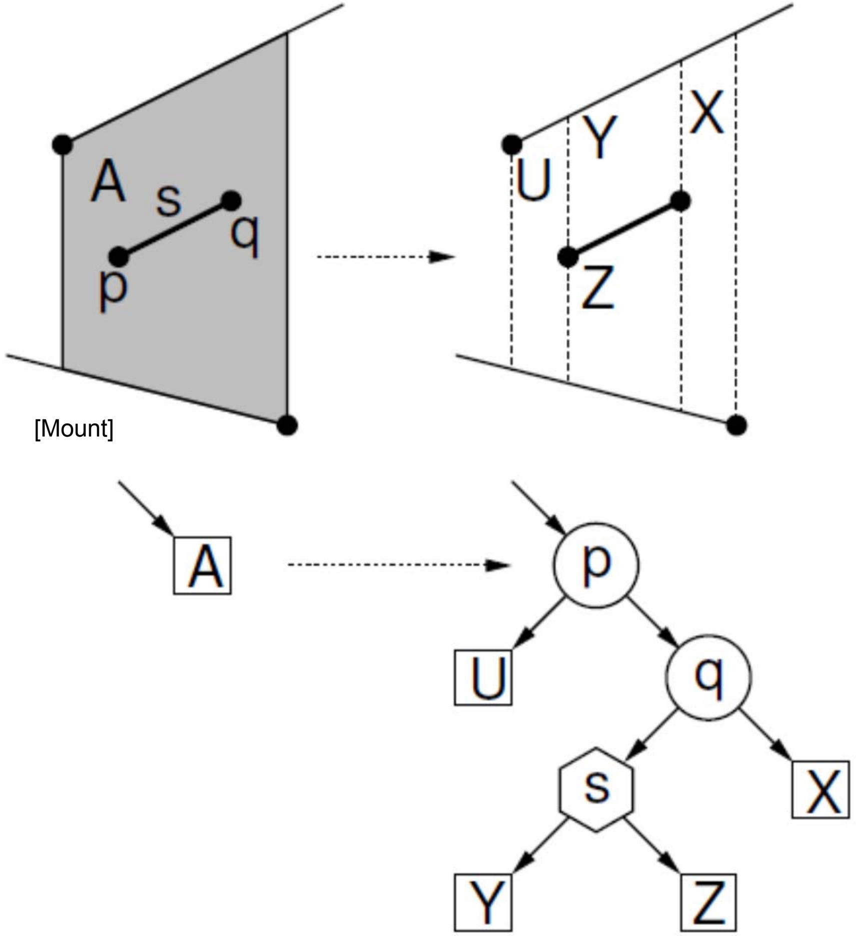
[Mount]

Trapezoid A replaced by

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

Construction – addition of a segment

b) Two segment endpoints – 4 new trapezoids



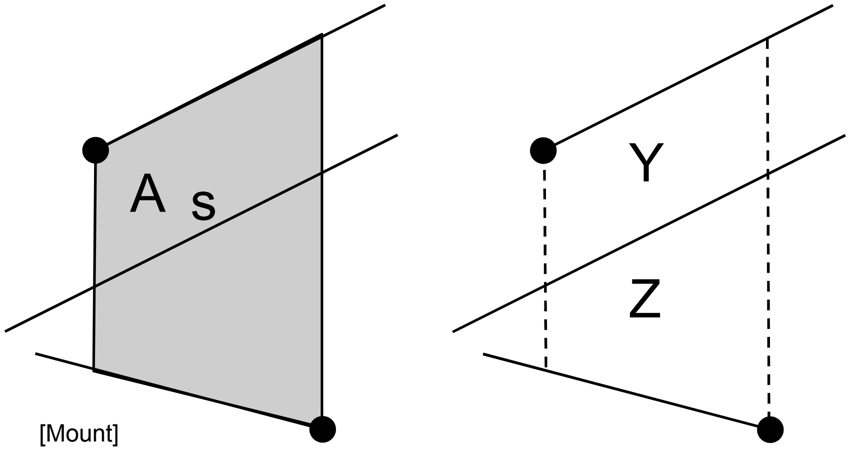
[Mount]

Trapezoid A replaced by

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

Construction – addition of a segment

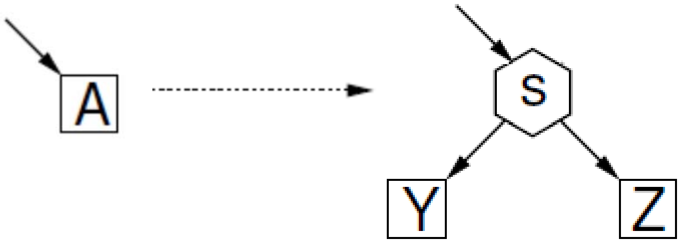
c) No segment endpoint – create 2 trapezoids



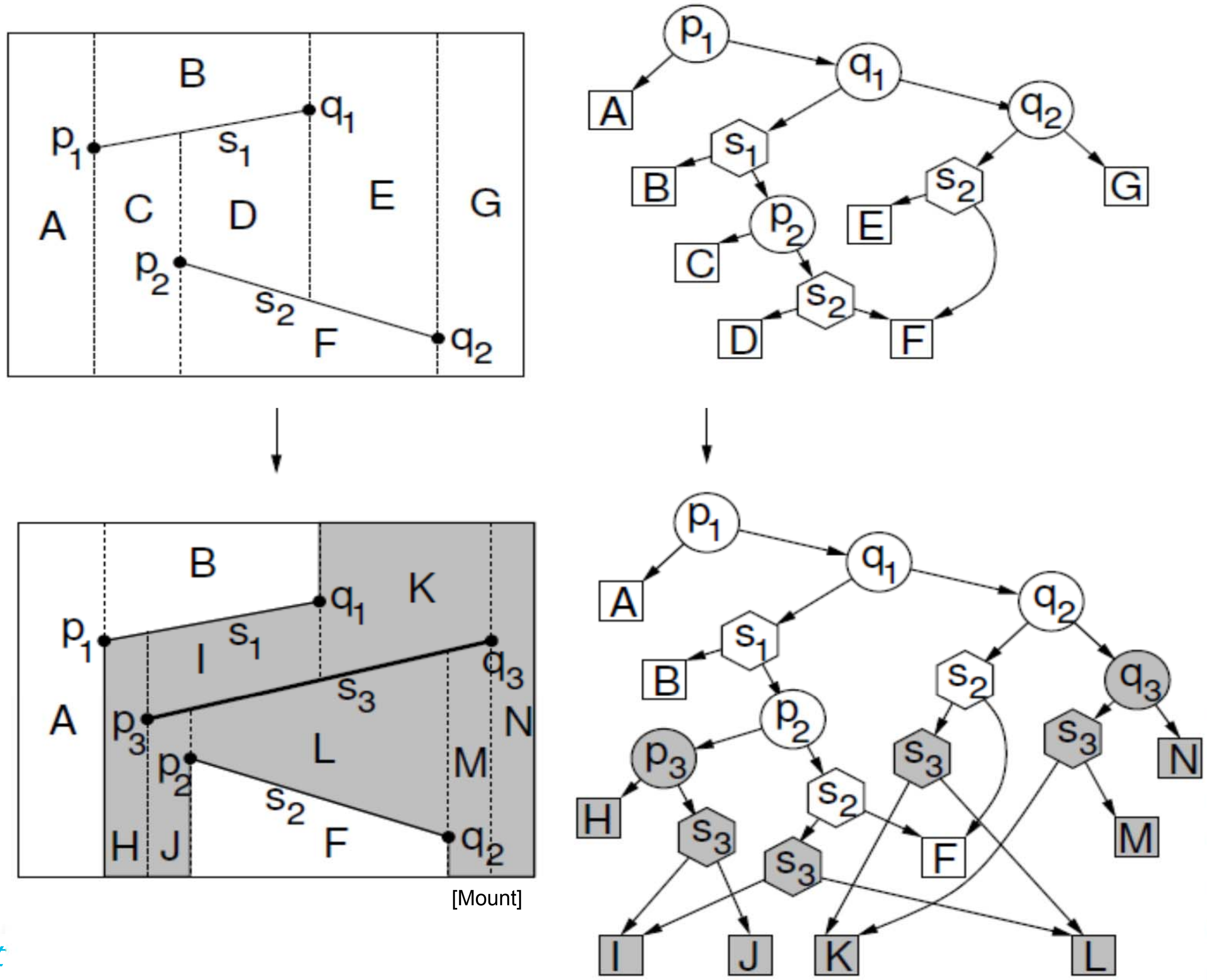
[Mount]

Trapezoid A replaced by

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

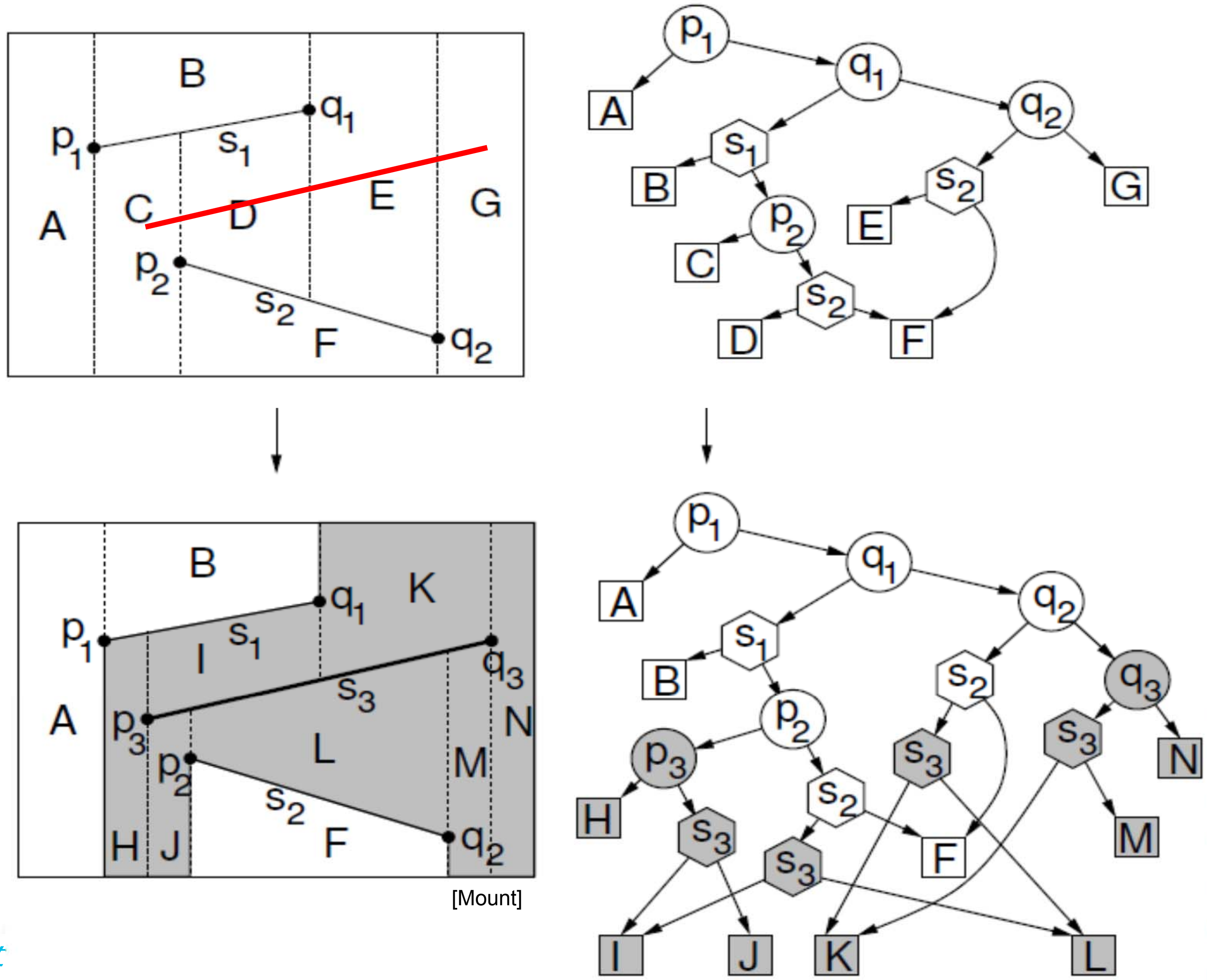


Segment insertion example



[Mount]

Segment insertion example



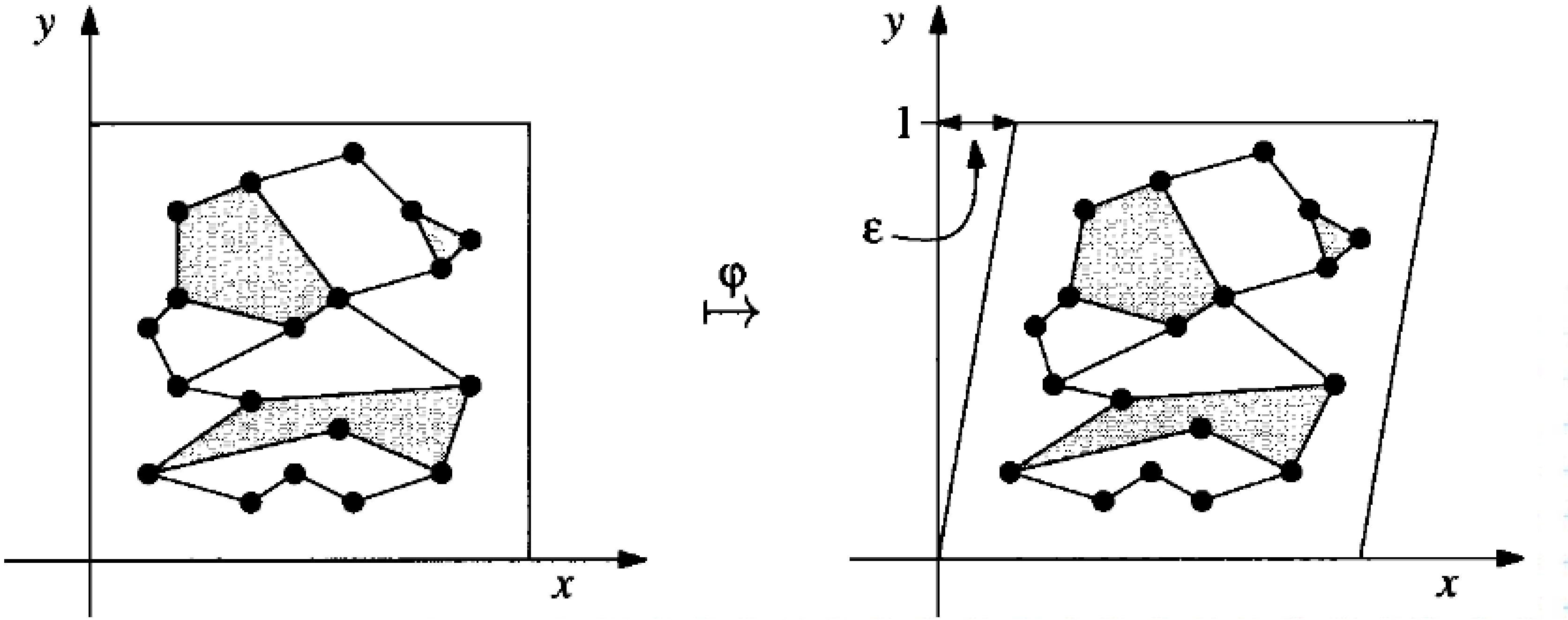
[Mount]

Analysis and proofs

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

Handling of degenerate cases - principle

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



[Berg]

Handling of degenerate cases - realization

- Trick
  - store original (x,y), not the sheared x',y'
  - we need to perform just 2 operations:

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

Point location summary

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

References

- [Berg] Mark de Berg, Otfried Cheong, Marc van Kreveld, Mark Overmars: Computational Geometry: *Algorithms and Applications*, Springer-Verlag, 3rd rev. ed. 2008. 386 pages, 370 fig. ISBN: 978-3-540-77973-5 http://www.cs.uu.nl/geobook/
- [Mount] Mount, D.: *Computational Geometry Lecture Notes for Fall 2016*, University of Maryland, Lectures 9, 10 http://www.cs.umd.edu/class/fall2016/cmsc754/Lects/cmsc754-fall16-lects.pdf

DCGI

DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

GEOMETRIC SEARCHING PART 2: RANGE SEARCH

PETR FELKEL

FEL CTU PRAGUE

Version from 19.10.2017

Range search

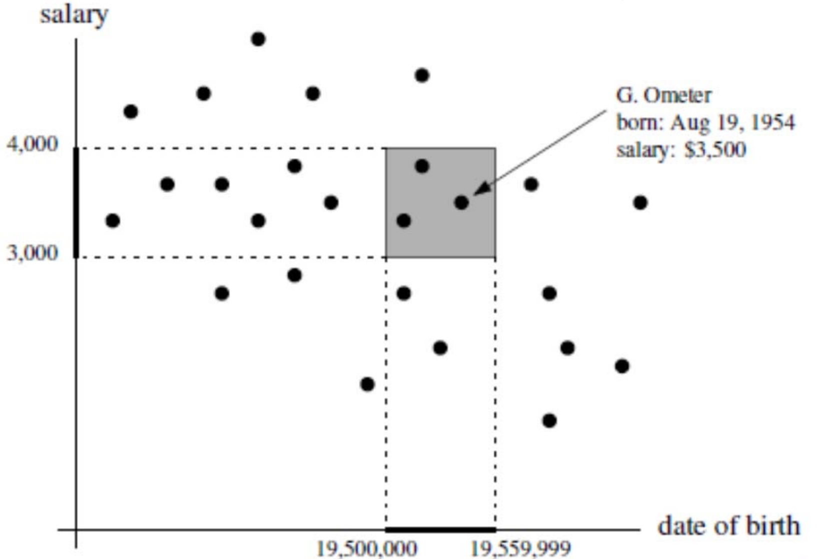
- Orthogonal range searching
- Canonical subsets
- 1D range tree
- 2D-nD Range tree
  - With fractional cascading (Layered tree)
- Kd-tree

Orthogonal range searching

- Given a set of points P, find the points in the region Q

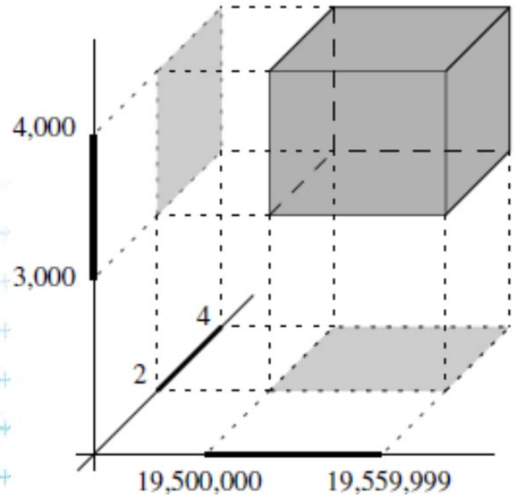
- Example: Databases (records->points)
  - Find the people with given range of salary, date of birth, kids, …

2D: axis parallel rectangle



in YYYYMMDD format

3D: axis parallel box



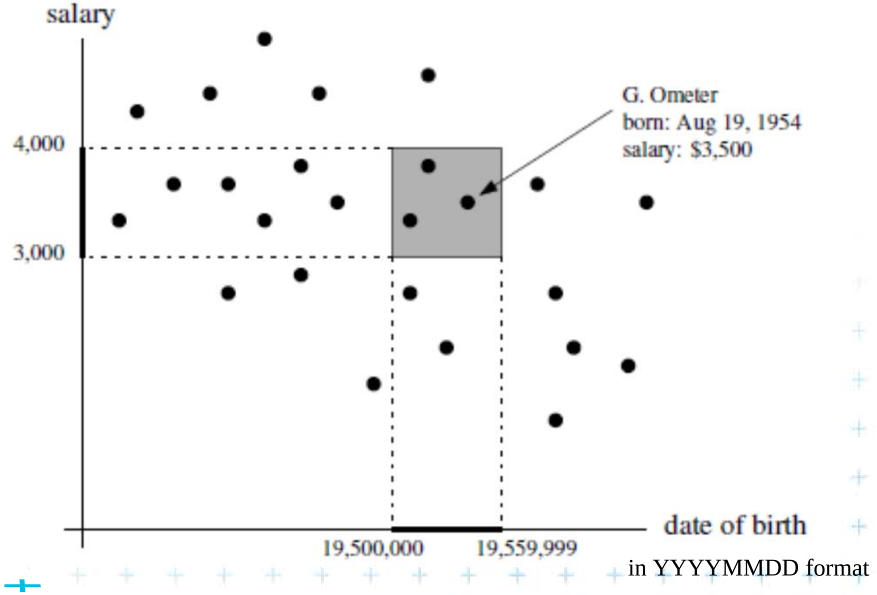
[Berg]

Orthogonal range searching

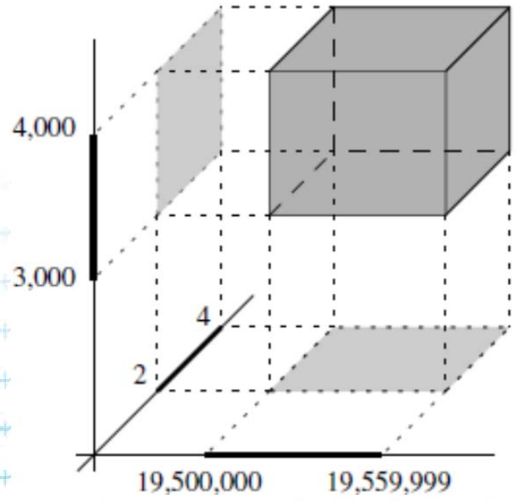
- Given a set of points P, find the points in the region Q
  - Search space: a set of points P (somehow represented)

- Example: Databases (records->points)
  - Find the people with given range of salary, date of birth, kids, …

2D: axis parallel rectangle



3D: axis parallel box



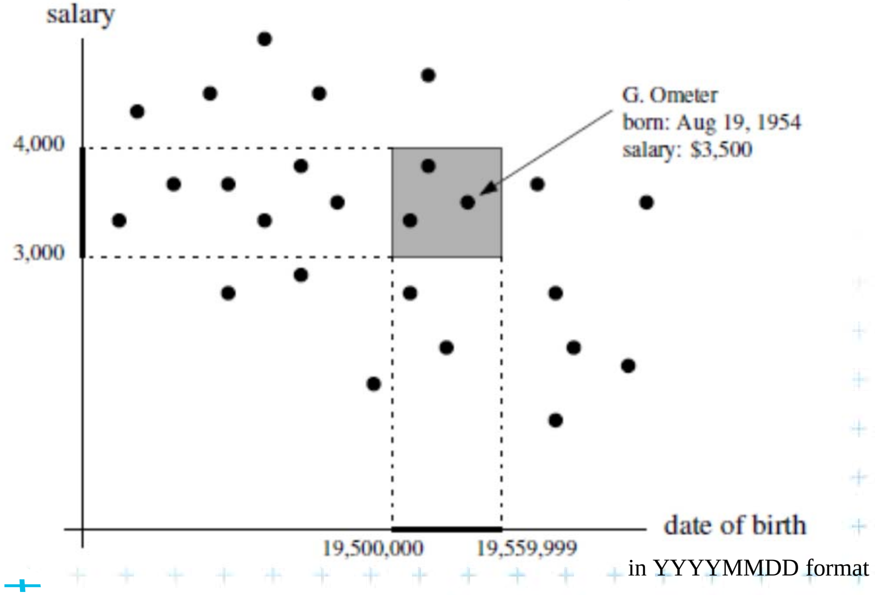
[Berg]

Orthogonal range searching

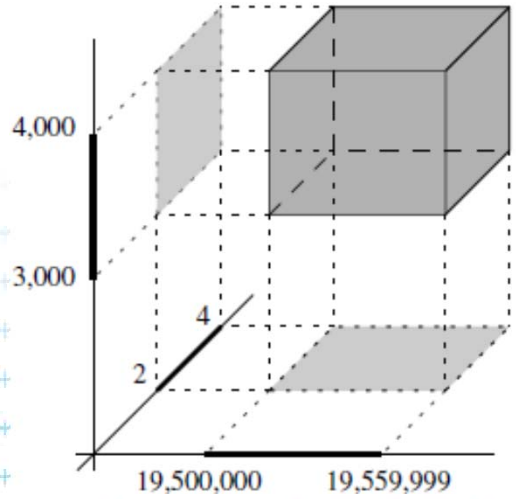
- Given a set of points P, find the points in the region Q
  - Search space: a set of points P (somehow represented)
  - Query: intervals Q (axis parallel rectangle)

- Example: Databases (records->points)
  - Find the people with given range of salary, date of birth, kids, …

2D: axis parallel rectangle



3D: axis parallel box

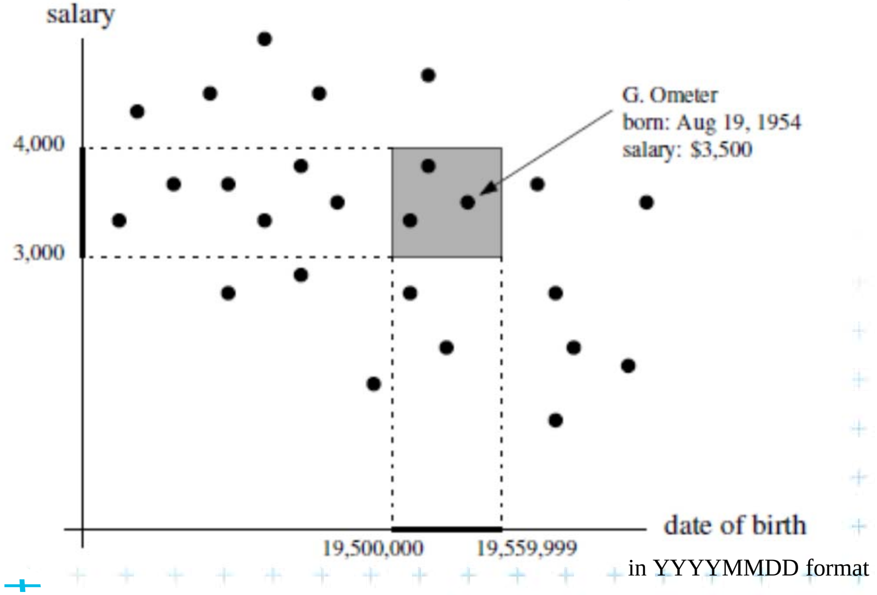


[Berg]

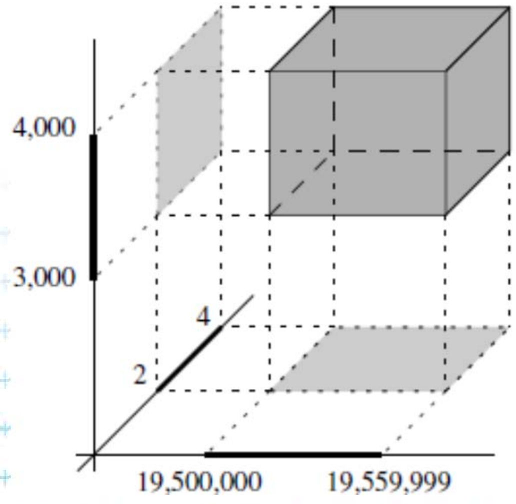
Orthogonal range searching

- Given a set of points P, find the points in the region Q
  - Search space: a set of points P (somehow represented)
  - Query: intervals Q (axis parallel rectangle)
  - Answer: points contained in Q
- Example: Databases (records->points)
  - Find the people with given range of salary, date of birth, kids, …

2D: axis parallel rectangle



3D: axis parallel box



[Berg]

Orthogonal range searching

- Query region = axis parallel rectangle
  - nDimensional search can be decomposed into set of 1D searches (separable)

Other range searching variants

- Search space S: set of
  - line segments,
  - rectangles, …
- Query region Q: any other searching region
  - disc,
  - polygon,
  - halfspace, …
- Answer: subset of S laying in Q

- We concentrate on points in orthogonal ranges

How to represent the search space?

Basic idea:

- Not all possible combination can be in the output (not the whole power set)
- => Represent only the “selectable” things (a well selected subset –> one of the canonical subsets)

How to represent the search space?

Basic idea:

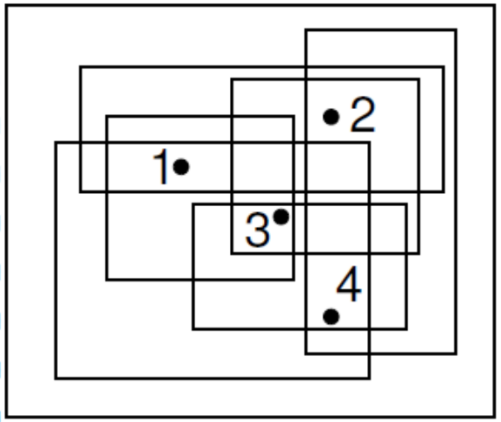
- Not all possible combination can be in the output (not the whole power set)
- => Represent only the “selectable” things
(a well selected subset –> one of the canonical subsets)

Example?

Subsets selectable by given range class

- The number of subsets that can be selected by simple ranges Q is limited
- It is usually much smaller than the power set of P
  - Power set of P where P = {1,2,3,4} (potenční množina) is {{ }, {1},{2},{3},{4}, {1,2},{1,3},{1,4}, {2,3},…,{2,3,4}, {1,2,3,4} } … $O(2^n)$ i.e. set of all possible subsets
  - Simple rectangular queries are limited
    - Defined by max 4 points along 4 sides => $O(n^4)$ of $O(2^n)$ power set
    - Moreover – not all sets can be formed by □ query Q

      e.g. sets {1,4} and {1,2,4} cannot be formed



[Mount]

Canonical subsets S_i

Search space $S = (P, Q)$ represented as a collection of canonical subsets $\{S_1, S_2, \dots, S_k\}$, each $S_i \quad S$,

- S_i may overlap each other (elements can be multiple times there)
- Any set can be represented as disjoint union disjunktní sjednocení of canonical subsets S_i each element knows from which subset it came
- Elements of disjoint union are ordered pairs (x, i) (every element x with index i of the subset S_i)

S_i may be selected in many ways

- from *n* singletons $\{pi\}$ … $O(n)$
- to power set of P … $O(2^n)$

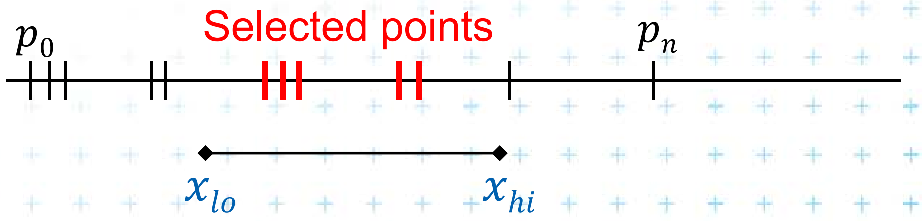
- Good DS balances between total number of canonical subsets and number of CS needed to answer the query

1D range queries (interval queries)

- Query: Search the interval $[x_{lo}, x_{hi}]$
- Search space: Points $P = \{p_1, p_2, \dots, p_n\}$ on the line

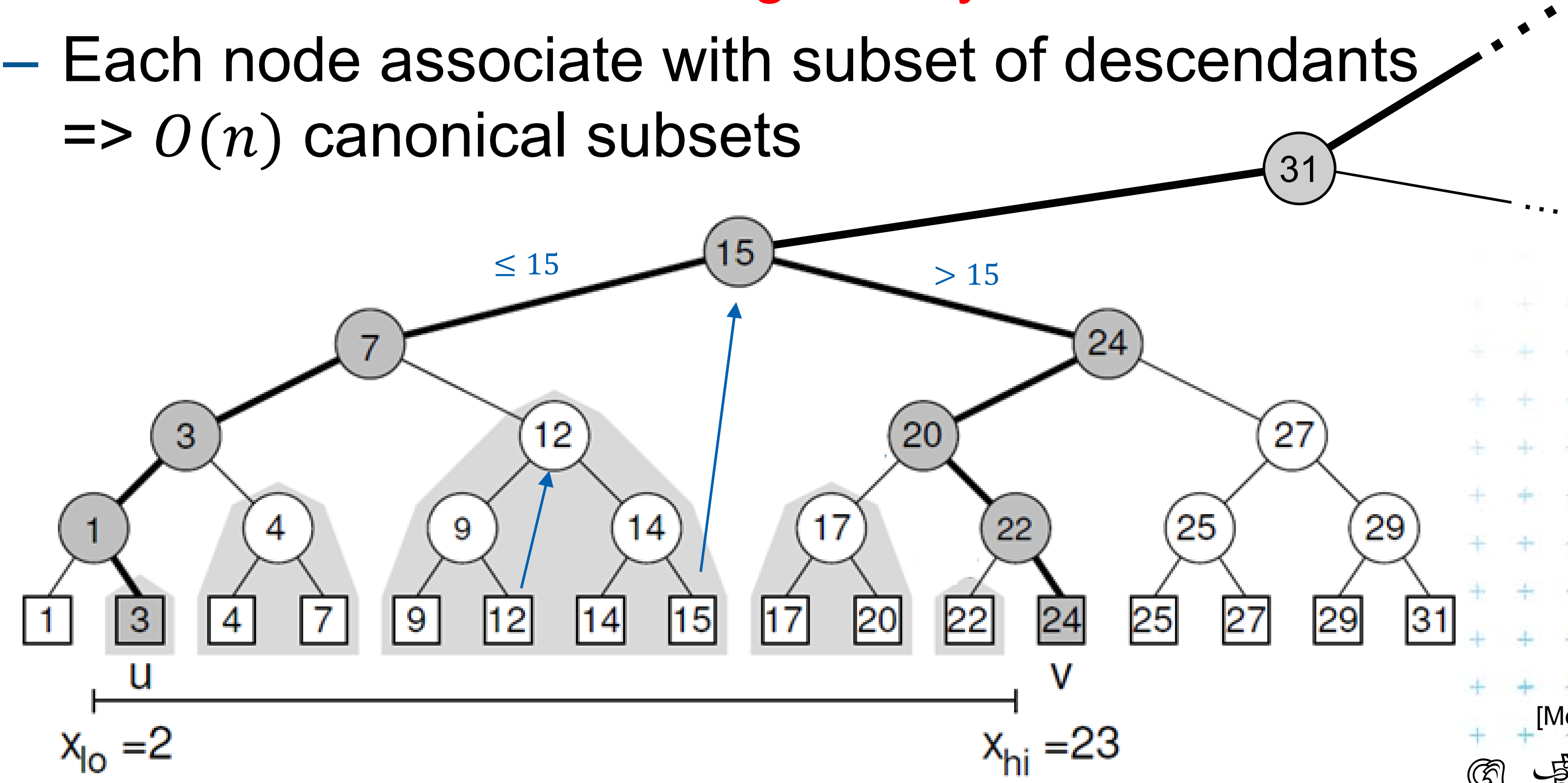
  a) Binary search in an array
  - Simple, but
  - not generalize to any higher dimensions

  b) Balanced binary search tree
  - 1D range tree
  - maintains canonical subsets
  - generalize to higher dimensions



1D range tree definition

- Balanced binary search tree (with repeated keys)
  - leaves – sorted points
  - inner node label – the largest key in its left child
  - Each node associate with subset of descendants => $O(n)$ canonical subsets



[Mount]

Canonical subsets and <2,23> search

- Canonical subsets for this subtree are #

{ {1}, {3}, …, {31}, 16

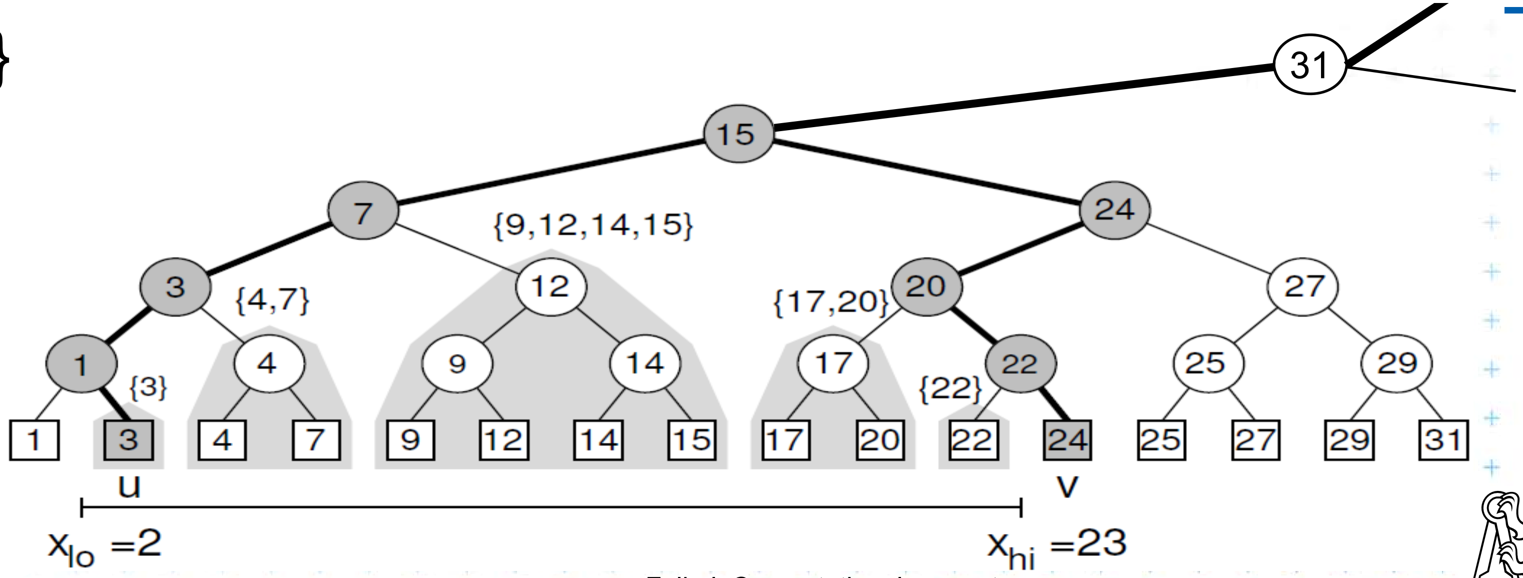
{1, 3}, {4, 7}, …, {29, 31} 8

{1, 3, 4, 7}, {9, 12, 14, 15}, …, {25, 27, 29, 31} 4

{1, 3, 4, 7, 9, 12, 14, 15}, {17, 20, 22, 24, 25, 27, 29, 31} 2

{1, 3, 4, 7, 9, 12, 14, 15, 17, 20, 22, 24, 25, 27, 29, 31} 1

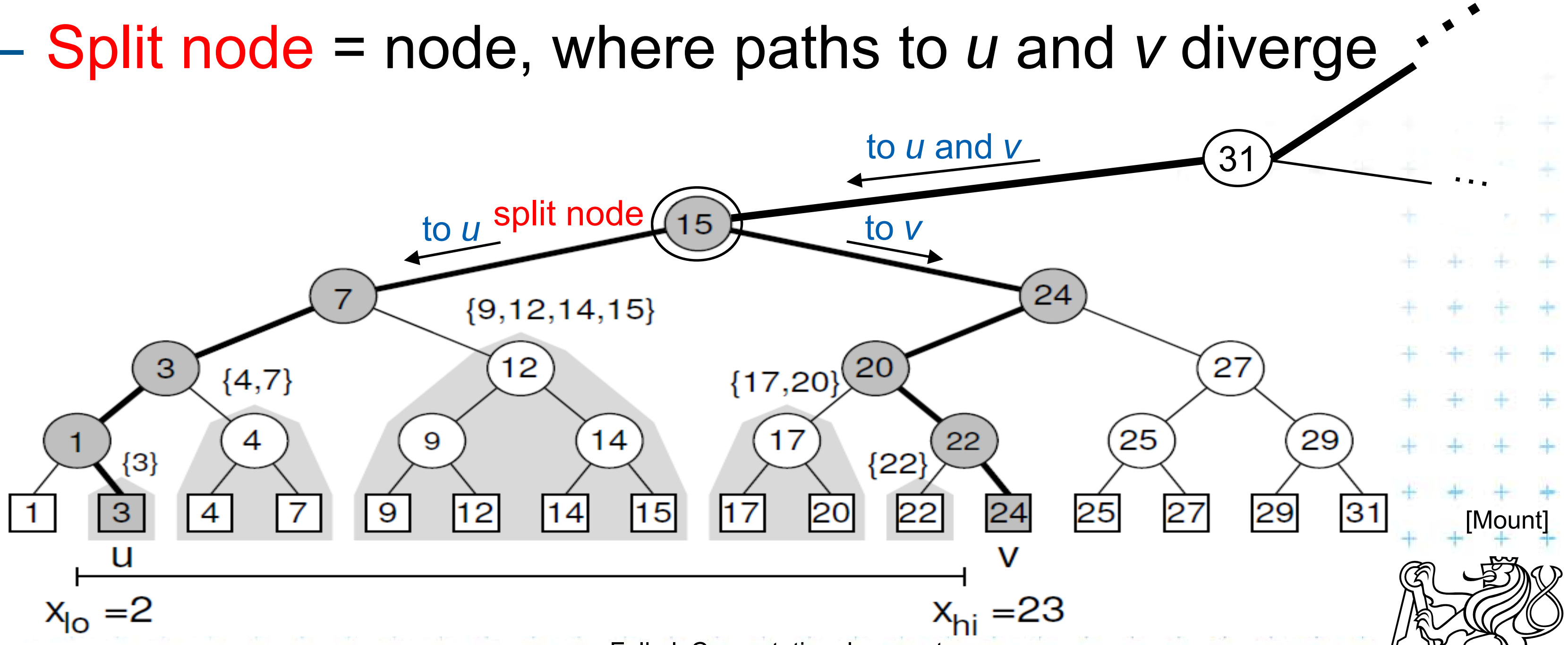
} O(n)



[Mount]

1D range tree search interval <2,23>

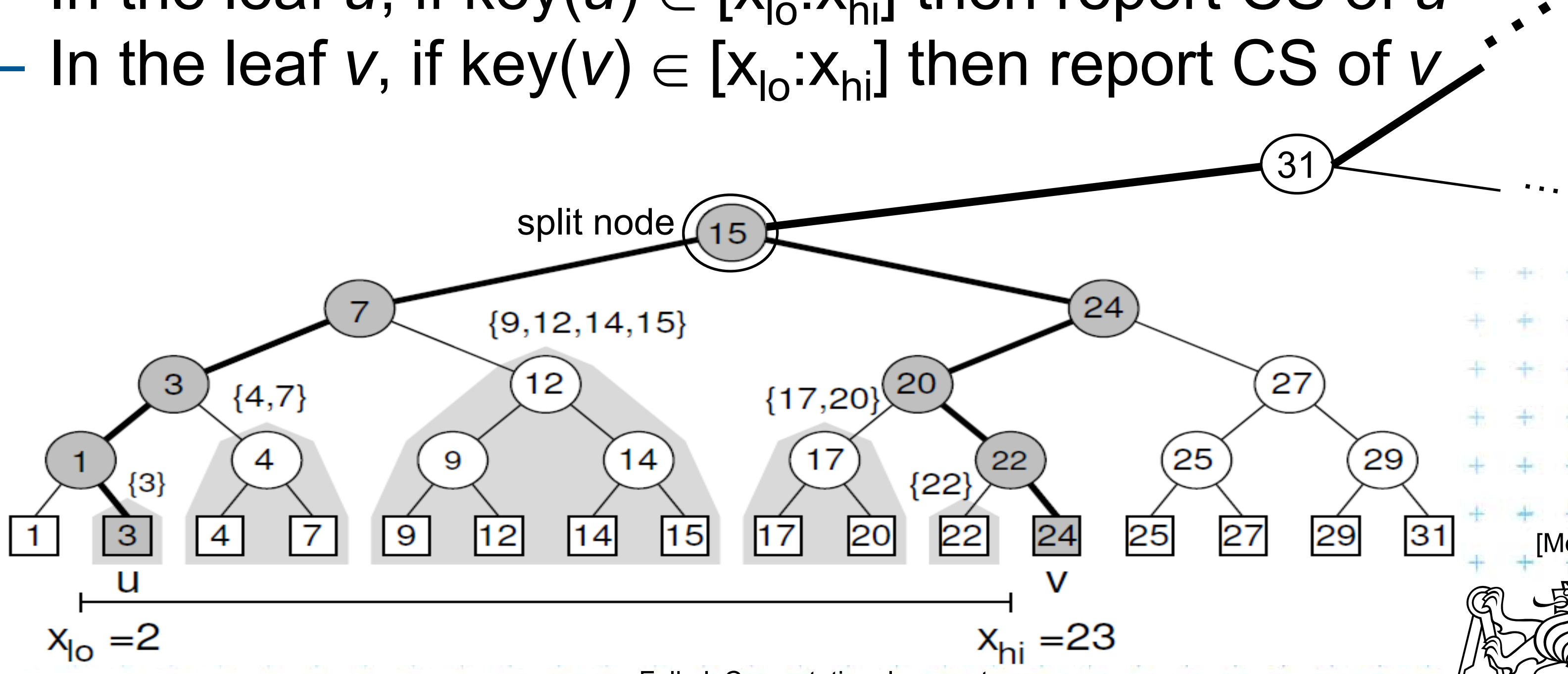
- Canonical subsets for any range found in O(log *n*)
  - Search x_{lo}: Find leftmost leaf *u* with key(*u*) ≥ x_{lo} 2 -> 3
  - Search x_{hi}: Find leftmost leaf *v* with key(*v*) ≥ x_{hi} 23 -> 24
  - Points between *u* and *v* lie within the range => report canon. subsets of maximal subtrees between *u* and *v*
  - Split node = node, where paths to *u* and *v* diverge



[Mount]

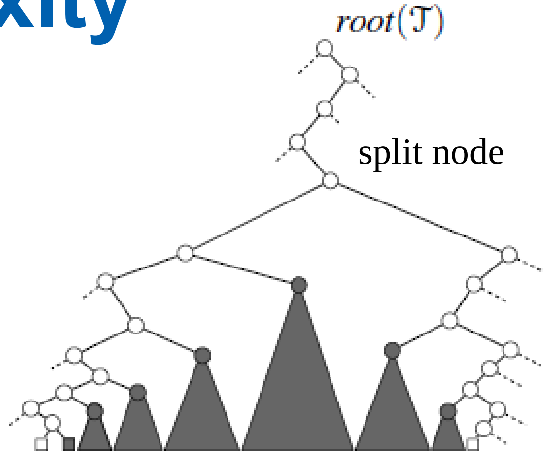
1D range tree search

- Reporting the subtrees (below the split node)
  - On the path to *u* whenever the *path goes left*, report the canonical subset (CS) associated to right child
  - On the path to *v* whenever the *path goes right*, report the canonical subset associated to left child
  - In the leaf *u*, if key(*u*) $\in [x_{lo}:x_{hi}]$ then report CS of *u*
  - In the leaf *v*, if key(*v*) $\in [x_{lo}:x_{hi}]$ then report CS of *v*



[Mount]

1D range tree search complexity



[Berg]

- Path lengths O(log n)

  => O(log n) canonical subsets (subtrees)

- Range counting queries
  - Return just the number of points in given range
  - Sum the total numbers of leaves stored in maximum subtree roots … O(log n) time
- Range reporting queries
  - Return all k points in given range
  - Traverse the canonical subtrees … O(log n + k) time
- O(n) storage, O(n log n) preprocessing (sort P)

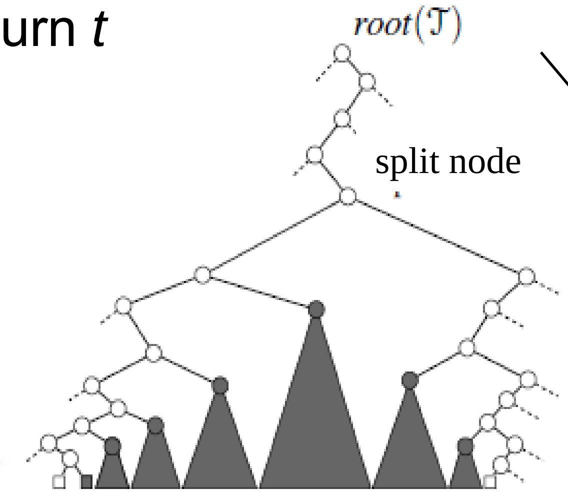
Find split node

FindSplitNode(*T*, [x:x’])

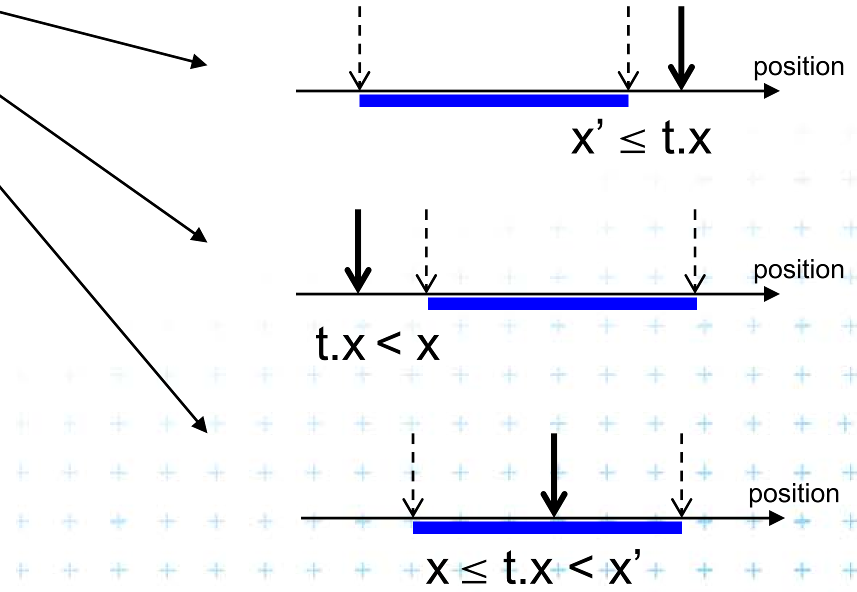
Input: Tree *T* and Query range [x:x’], $x \le x'$

Output: The node, where the paths to x and x’ split or the leaf, where both paths end

```
t = root(T)
while( t is not a leaf and ($x' \le t.x$ or $t.x < x$) ) // t out of the range [x:x’]
        if( $x' \le t.x$) t = t.left
        else          t = t.right
 return t
```



[Berg]

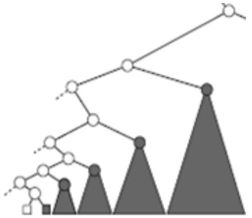


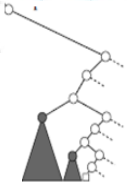
STOP

1D range search

(2D on slide 30)

```
1dRangeQuery( $t$, [x:x']])
Input:    1d range tree $t$ and Query range $[x:x']$
Output:   All points in $t$ lying in the range
tsplit = FindSplitNode( $t, x, x'$ )        // find interval point $t \in [x:x']$
if( $t_{split}$ is leaf )     // e.g. Searching [16:17] or [16:16.5] both stops in the leaf 17 in the previous example
    check if the point in $t_{split}$ must be reported    // $t_x \in [x:x']$
else // follow the path to $x$, reporting points in subtrees right of the path
    t = $t_{split}$.left
    while( t is not a leaf )
        if( $x \leq$ t.x)
            ReportSubtree( $t.right$ )    // any kind of tree traversal
            $t = t.left$
        else $t = t.right$
    check if the point in leaf $t$ must be reported
    // Symmetrically follow the path to x' reporting points left of the path
    t = $t_{split}$.right …
```



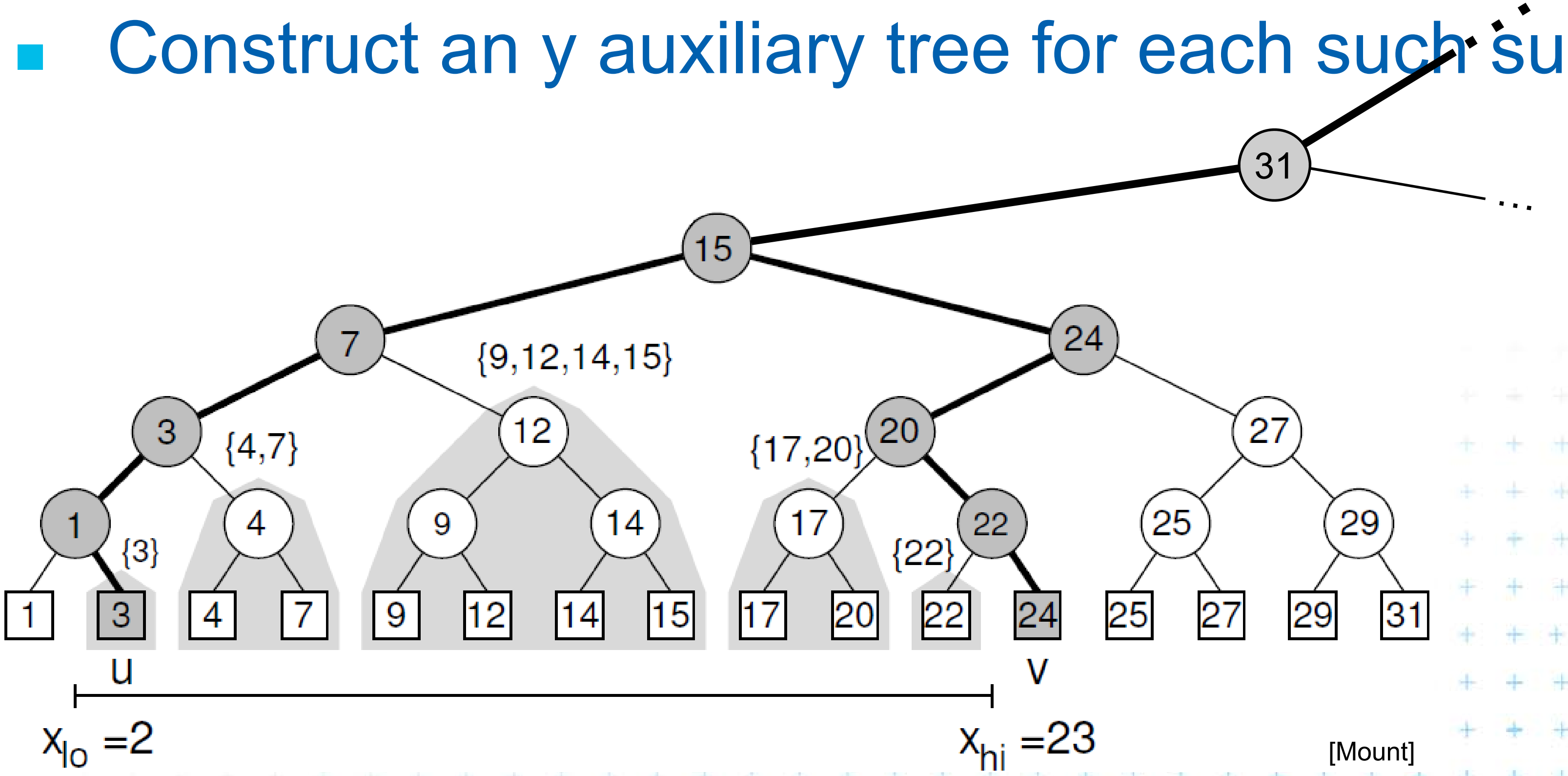


Multidimensional range searching

- Equal principle – find the largest subtrees contained within the range
- Separate one *n*-dimensional search into *n* 1-dimensional searches
- Different tree organization
  - Orthogonal (Multilevel) range search tree
    e.g. nd range tree
  - Kd tree

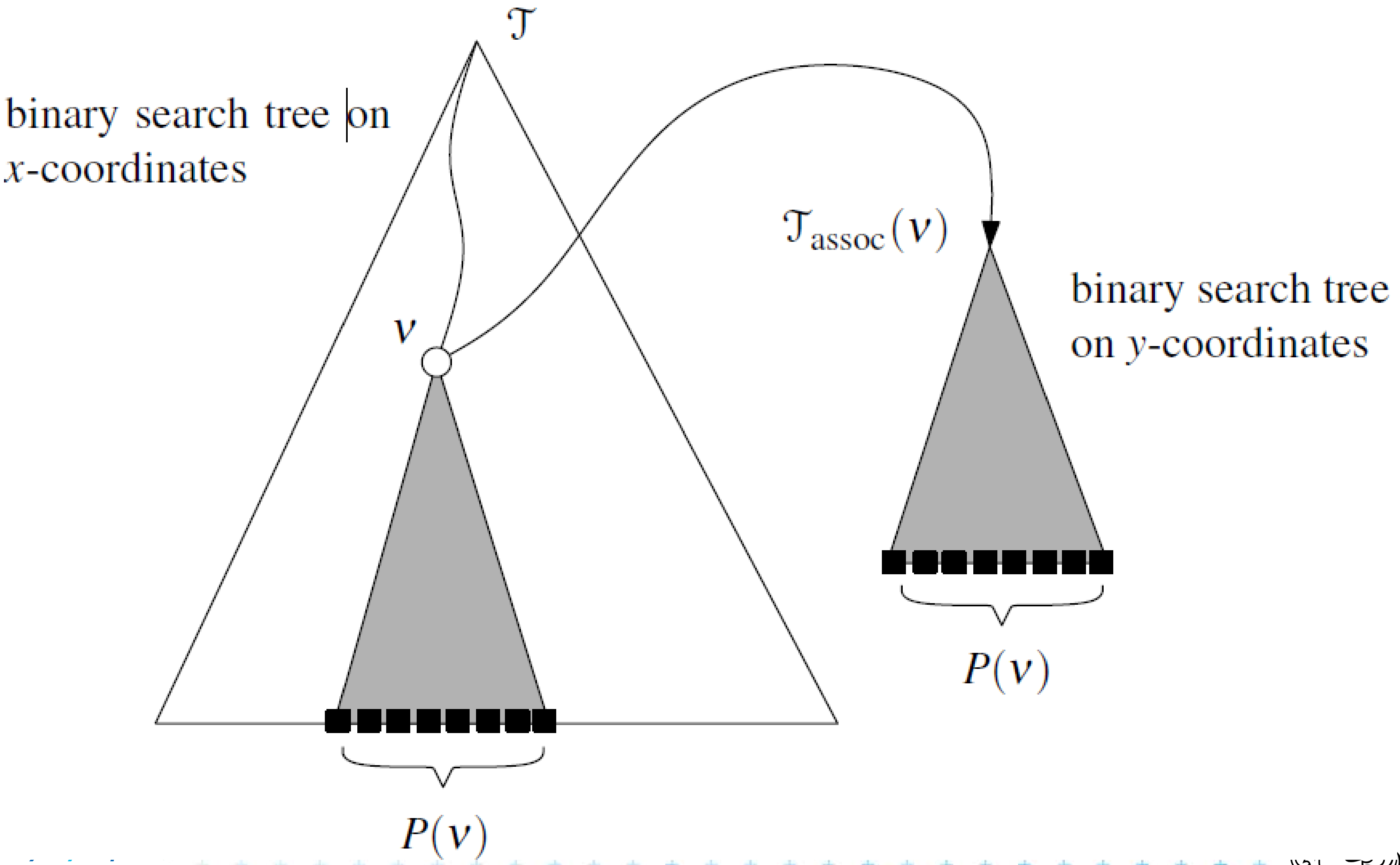
From 1D to 2D range tree

- Search points from [Q.$x_{lo,}$ Q.x_{hi}] [Q.$y_{lo,}$ Q.y_{hi}]
- 1d range tree: log n canonical subsets based on x
- Construct an y auxiliary tree for each such subset

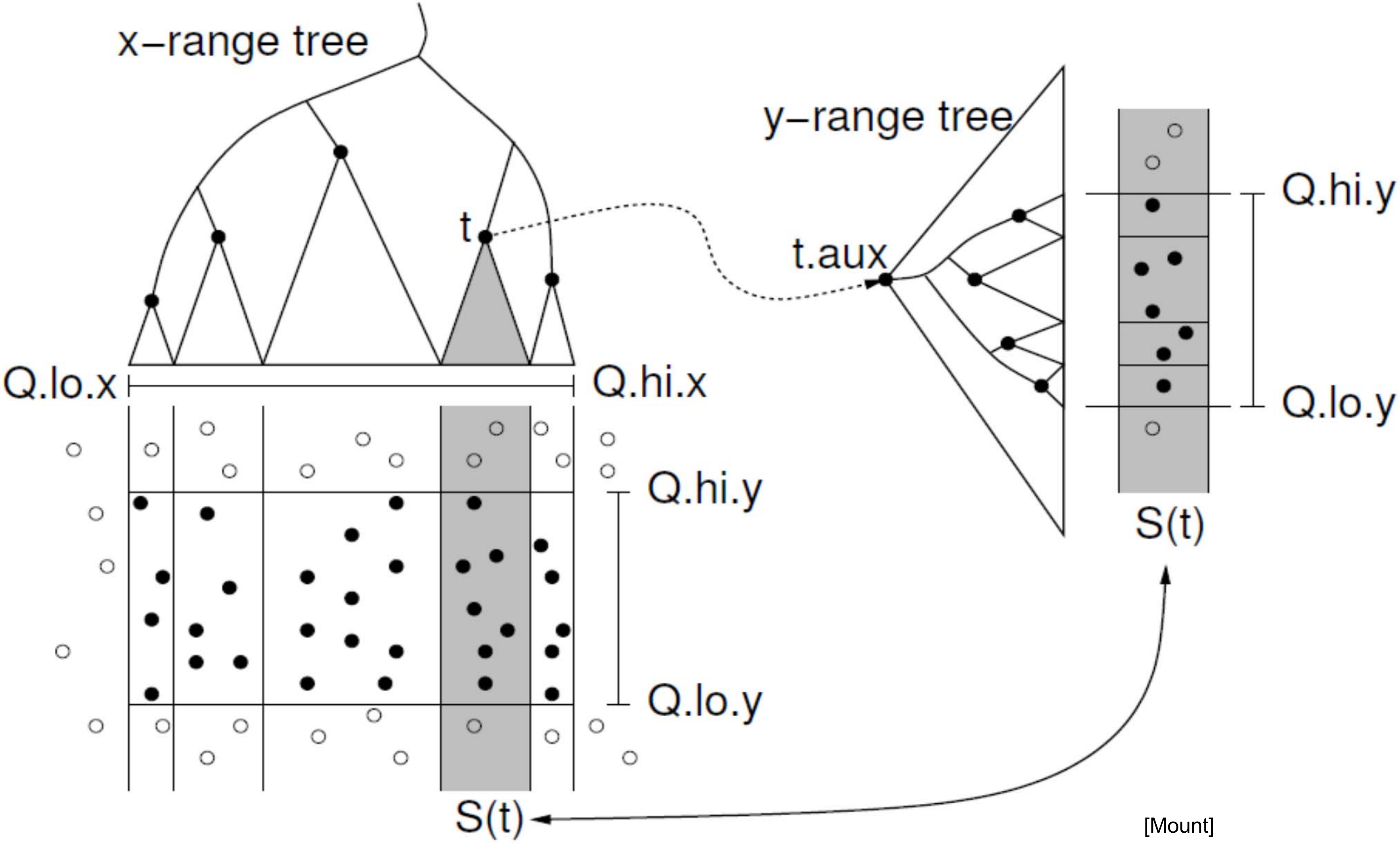


[Mount]

y-auxiliary tree for each canonical subset



2D range tree



[Mount]

2D range search

2dRangeQuery(*t*, [x:x’] × [y:y’])

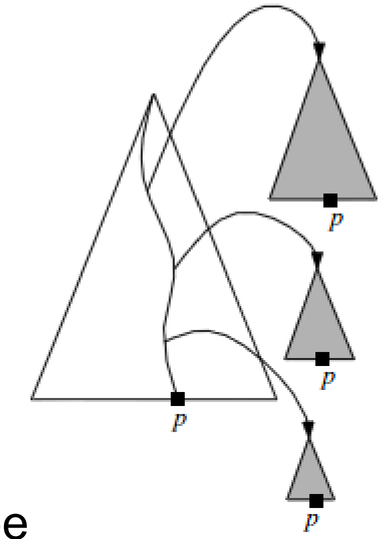
Input: 2d range tree *t* and Query range

Output: All points in *t* laying in the range

```
t_split = FindSplitNode( t, x, x’ )
if( t_split is leaf )
    check if the point in t_split must be reported      … $t.x \in [x:x']$, $t.y \in [y:y']$
else // follow the path to x, calling 1dRangeQuery on y
    t = t_split.left    // path to the left
    while( t is not a leaf )
        if( $x \le t.x$)
            1dRangeQuerry( t_assoc( t.right ), [y:y’] ) // check associated subtree
            t = t.left
        else t = t.right
  check if the point in leaf t must be reported      … $t.x \le x'$, $t.y \in [y:y']$
  Similarly for the path to x’     … // path to the right
```

2D range tree

- Search $O(\log^2 n + k)$ … $\log n$ in x, $\log n$ in y
- Space $O(n \log n)$
  - $O(n)$ the tree for x-coords
  - $O(n \log n)$ trees for y-coords
    - Point p is stored in all canonical subsets along the path from root to leaf with p,
    - once for x-tree level (only in one x-range)
    - each canonical subsets is stored in one auxiliary tree
    - $\log n$ levels of x-tree => $O(n \log n)$ space for y-trees



[Berg]

- Construction - $O(n \log n)$
  - Sort points (by x and by y). Bottom up construction

Canonical subsets

- Canonical subsets for this subtree are #

{ {1}, {3}, …, {31}, 16

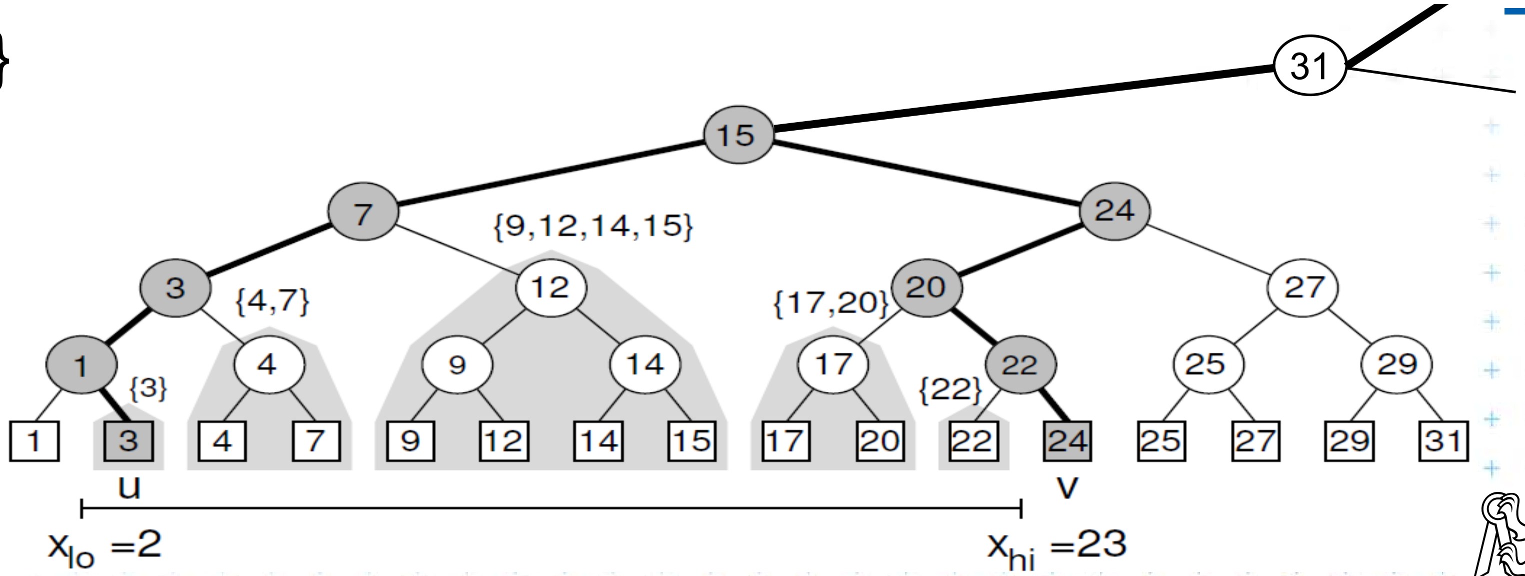
{1, 3}, {4, 7}, …, {29, 31} 8

{1, 3, 4, 7}, {9, 12, 14, 15}, …, {25, 27, 29, 31} 4

{1, 3, 4, 7, 9, 12, 14, 15}, {17, 20, 22, 24, 25, 27, 29, 31} 2

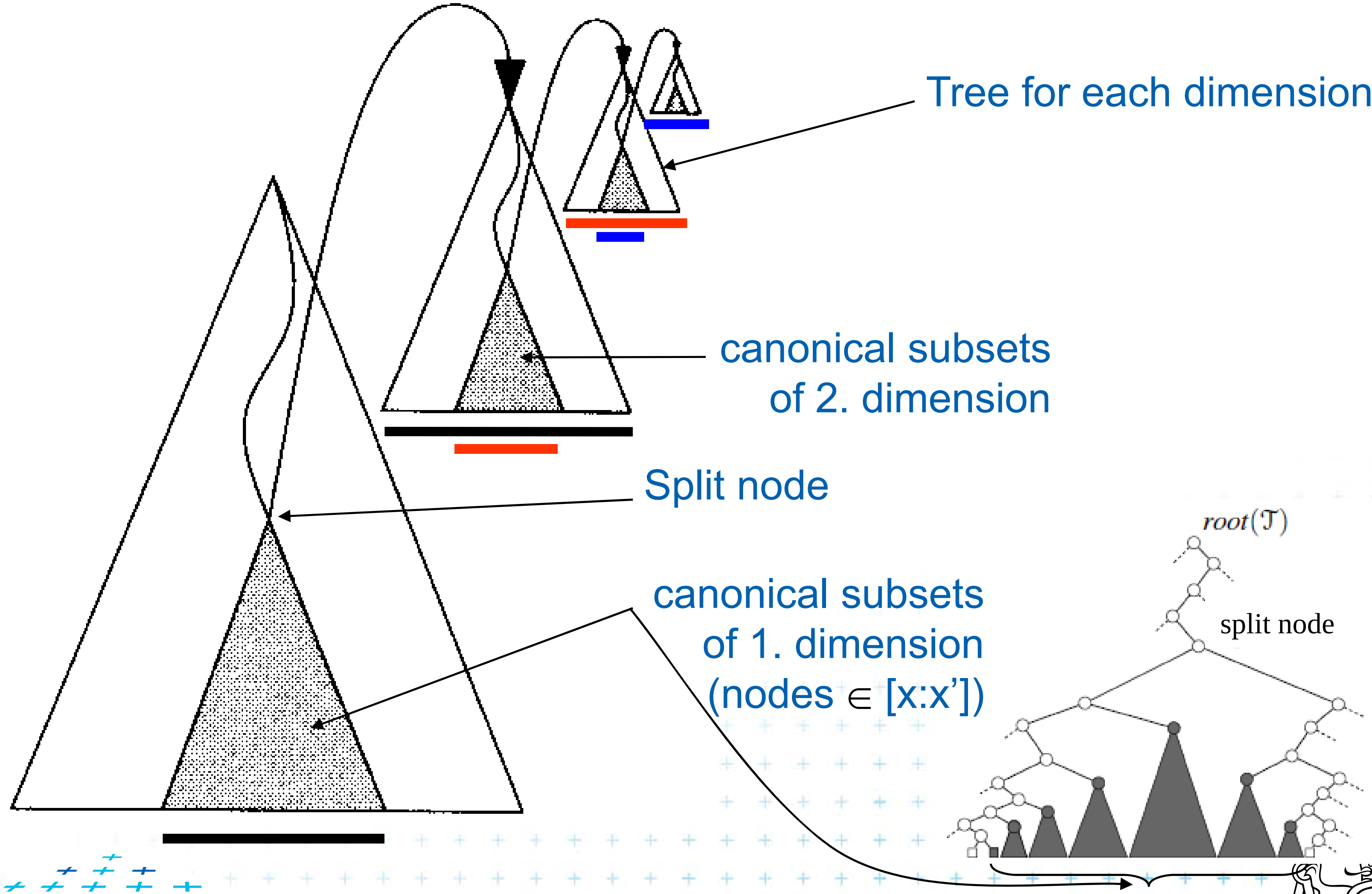
{1, 3, 4, 7, 9, 12, 14, 15, 17, 20, 22, 24, 25, 27, 29, 31} 1

} O(n)



[Mount]

nD range tree (multilevel search tree)



Tree for each dimension

canonical subsets of 2. dimension

Split node

canonical subsets of 1. dimension (nodes ∈ [x:x']])

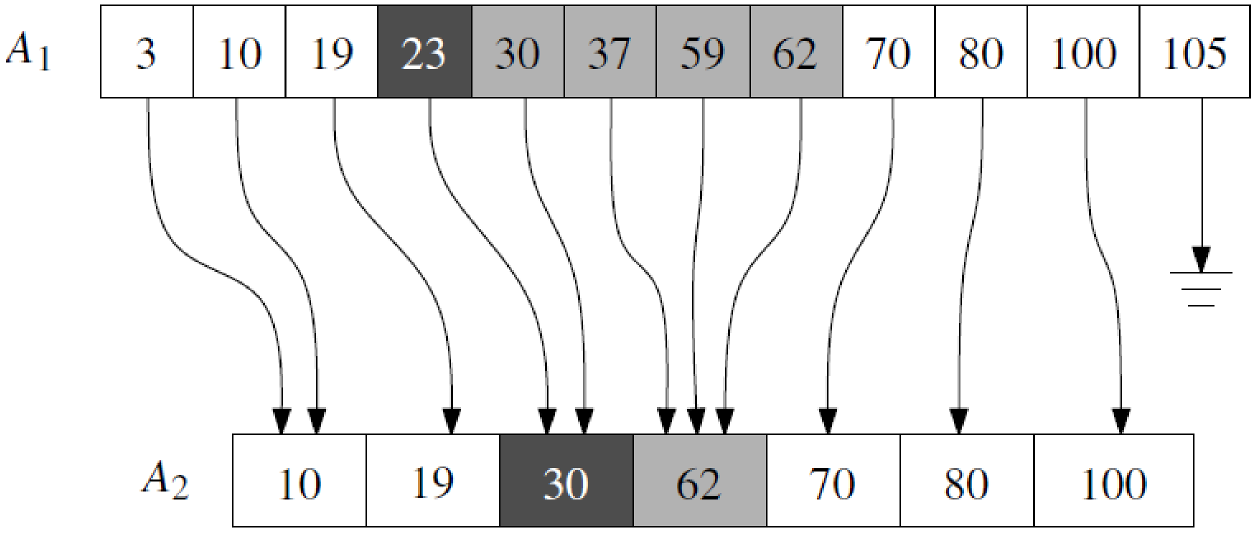
[Berg]

Fractional cascading - principle

- Two sets S_1, S_2 stored in sorted arrays A_1, A_2
- Report objects in both arrays whose keys in [y:y’]
- Naïve approach – search twice independently
  - $O(\log n_1 + k_1)$ – search in A_1 + report k_1 elements
  - $O(\log n_2 + k_2)$ – search in A_2 + report k_2 elements
- Fractional cascading – adds pointers from A_1 to A_2
  - $O(\log n_1 + k_1)$ – search in A_1 + report k_1 elements
  - $O(1 + k_2)$ – jump to A_2 + report k_2 elements
  - Saves the $O(\log n_2)$ – search

Fractional cascading – principle for arrays

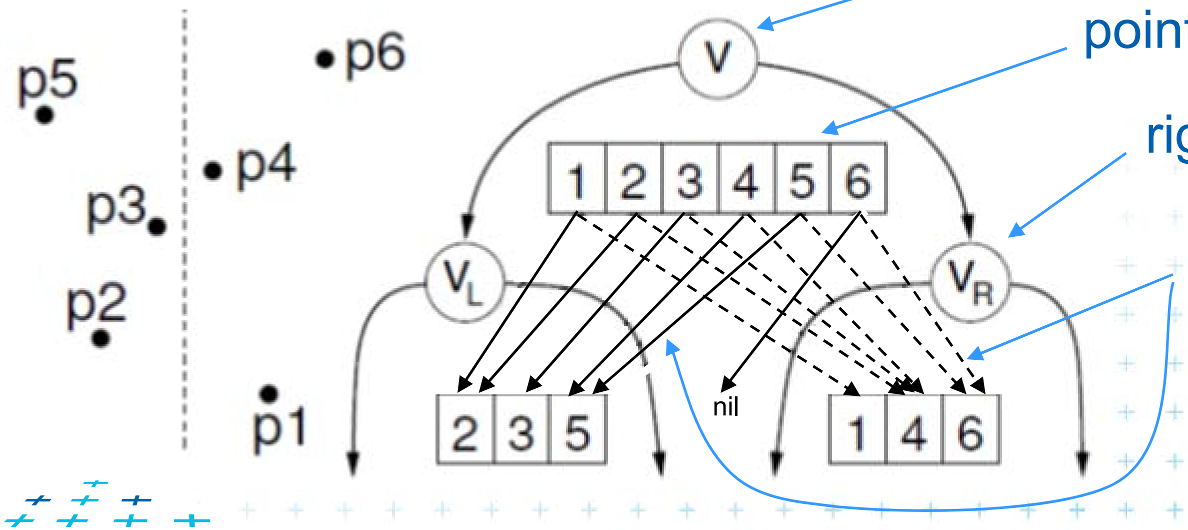
- Add pointers from A_1 to A_2
  - From element in A_1 with a key y_i point to the element in A_2 with the smallest key *larger or equal* to y_i
- Example query with the range [20 : 65]



[Berg]

Fractional cascading in the 2D range tree

- How to save one log n during last dim. search?
  - Store canonical subsets in arrays sorted by y
  - Pointers to subsets for both child nodes v_L and v_R
  - $\mathcal{O}(1)$ search in lower levels => in two dimensional search $\mathcal{O}(\log^2 n)$ time -> $\mathcal{O}(\log n)$



internal node in x-tree

points p1 to p6 sorted by - y

right son of v

Pointer to the smallest larger or equal y-value

[Mount]

Orthogonal range tree - summary

- Orthogonal range queries in plane
  - Counting queries $O(\log^2 n)$ time, or with fractional cascading $O(\log n)$ time
  - Reporting queries plus $O(k)$ time, for k reported points
  - Space $O(n \log n)$
  - Construction $O(n \log n)$
- Orthogonal range queries in d-dimensions, $d \geq 2$
  - Counting queries $O(log^d n)$ time, or with fractional cascading $O(\log^{d-1} n)$ time
  - Reporting queries plus $O(k)$ time, for k reported points
  - Space $O(n \log^{d-1} n)$
  - Construction $O(n \log^{d-1} n)$ time

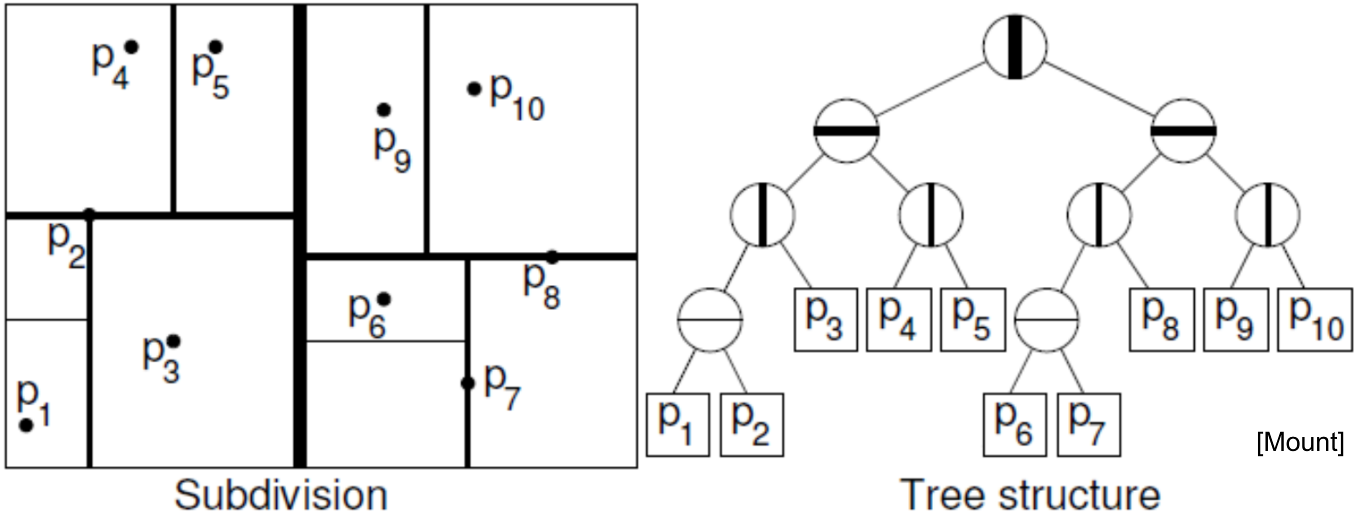
Kd-tree

- Easy to implement
- Good for different searching problems (counting queries, nearest neighbor,…)
- Designed by Jon Bentley as k-dimensional tree (2-dimensional kd-tree was a 2-d tree, …)
- Not the asymptotically best for orthogonal range search (=> range tree is better)
- Types of queries
  - Reporting – points in range
  - Counting – number of points in range

Kd-tree principle

- Subdivide space according to different dimension (x-coord, then y-coord, …)
- This subdivides space into rectangular cells => hierarchical decomposition of space
- In node *t* store: cutDim, cutVal, (size (for counting queries))

= Cutting line



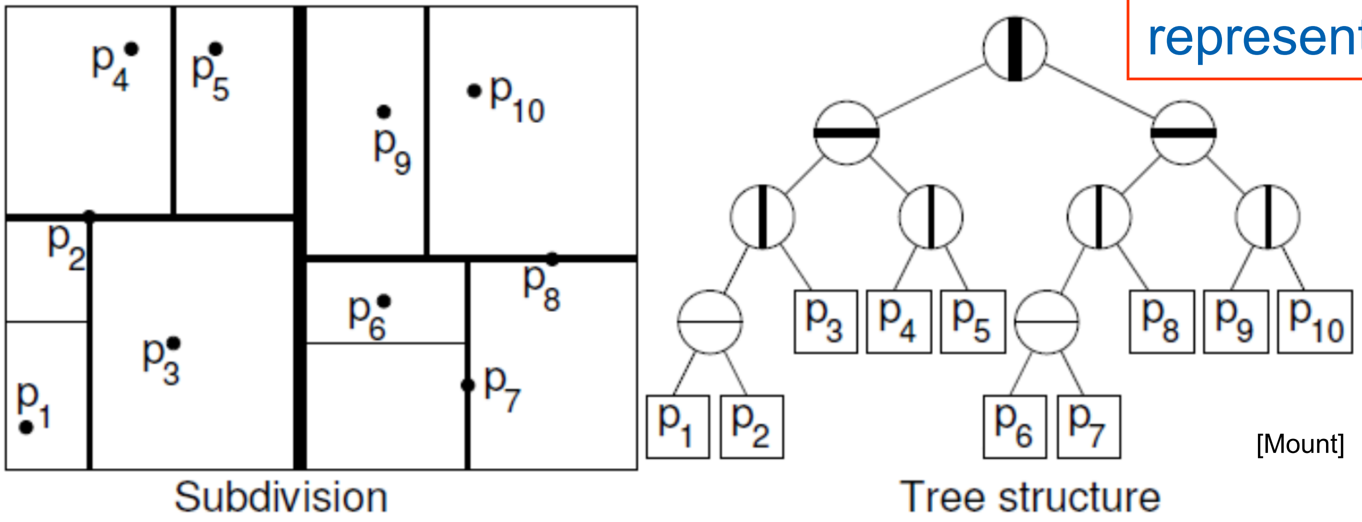
[Mount]

Kd-tree principle

- Subdivide space according to different dimension (x-coord, then y-coord, …)
- This subdivides space into rectangular cells => hierarchical decomposition of space
- In node *t* store: cutDim, cutVal, (size (for counting queries))

= Cutting line

Each tree node represents a region



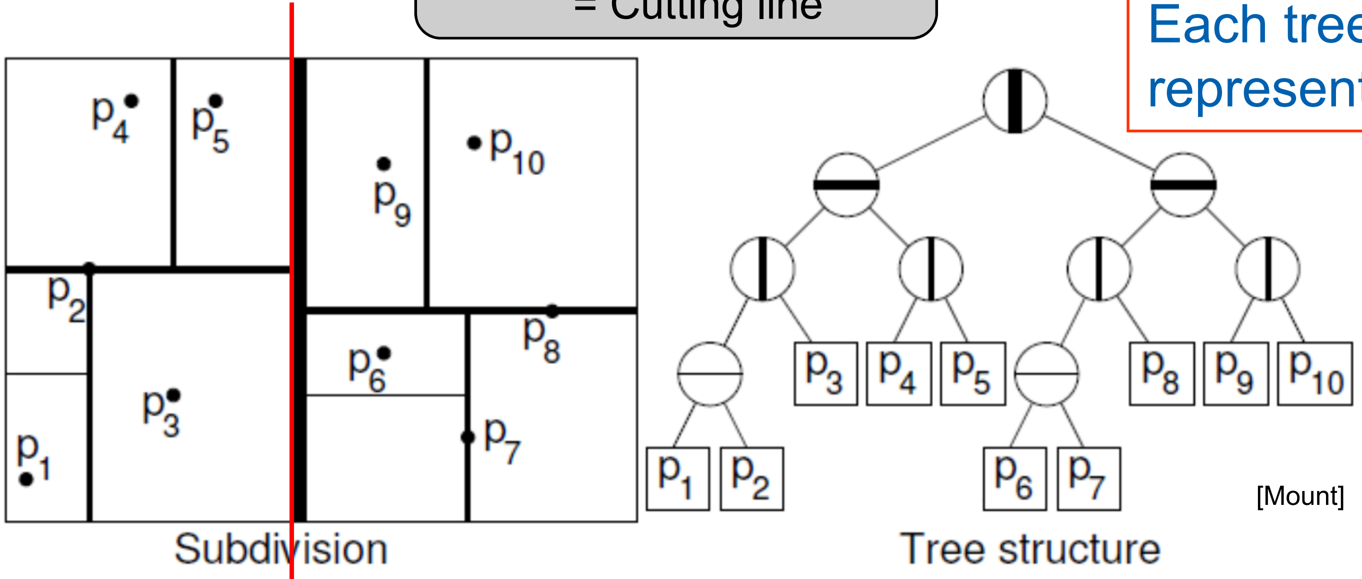
[Mount]

Kd-tree principle

- Subdivide space according to different dimension (x-coord, then y-coord, …)
- This subdivides space into rectangular cells => hierarchical decomposition of space
- In node *t* store: cutDim, cutVal, (size (for counting queries))

= Cutting line

Each tree node represents a region



Subdivision

Tree structure

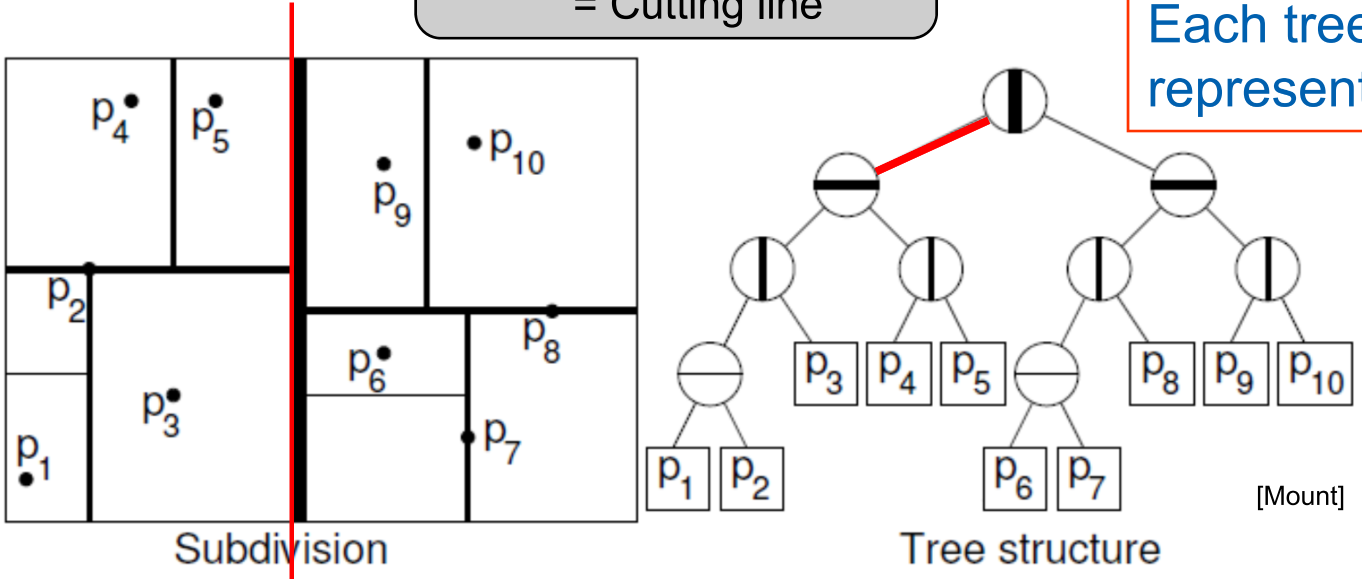
[Mount]

Kd-tree principle

- Subdivide space according to different dimension (x-coord, then y-coord, …)
- This subdivides space into rectangular cells => hierarchical decomposition of space
- In node *t* store: cutDim, cutVal, (size (for counting queries))

= Cutting line

Each tree node represents a region



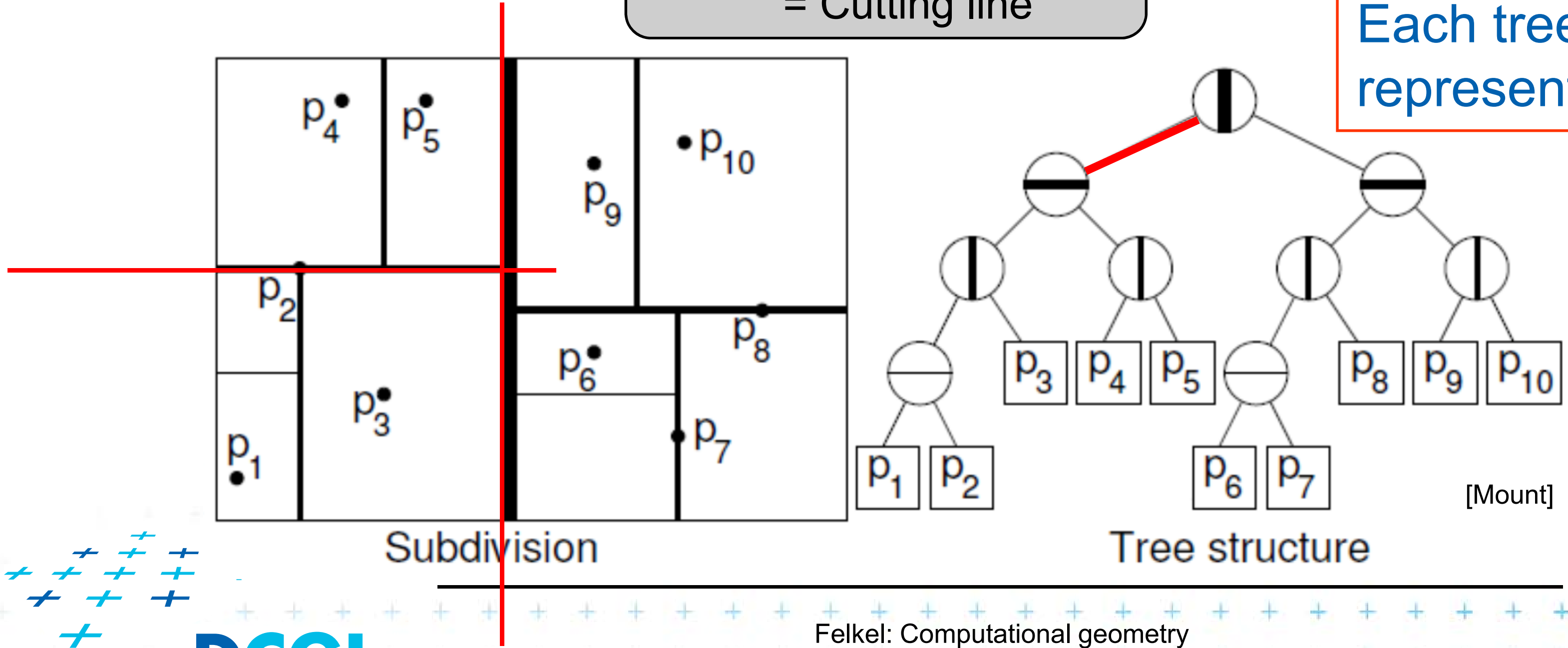
[Mount]

Kd-tree principle

- Subdivide space according to different dimension (x-coord, then y-coord, …)
- This subdivides space into rectangular cells => hierarchical decomposition of space
- In node *t* store: cutDim, cutVal, (size (for counting queries))

= Cutting line

Each tree node represents a region



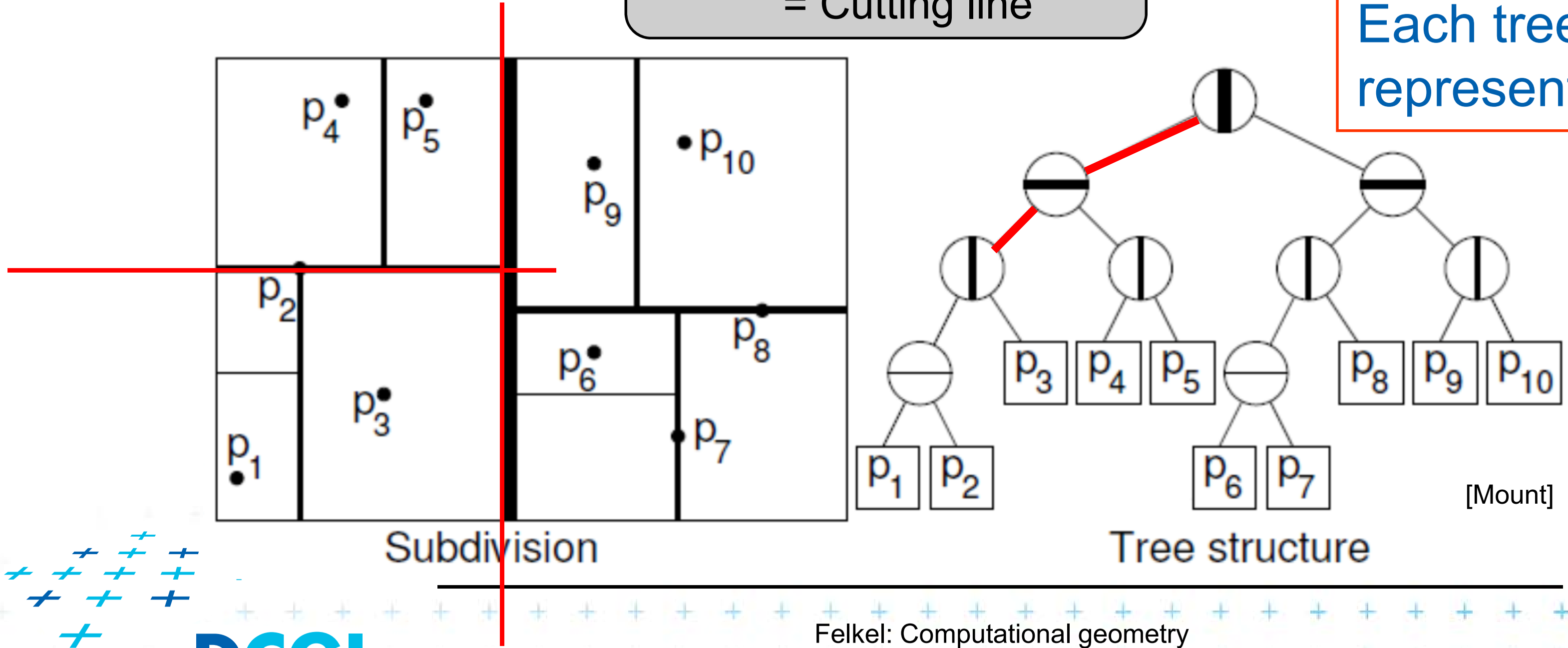
[Mount]

Kd-tree principle

- Subdivide space according to different dimension (x-coord, then y-coord, …)
- This subdivides space into rectangular cells => hierarchical decomposition of space
- In node *t* store: cutDim, cutVal, (size (for counting queries))

= Cutting line

Each tree node represents a region



Subdivision

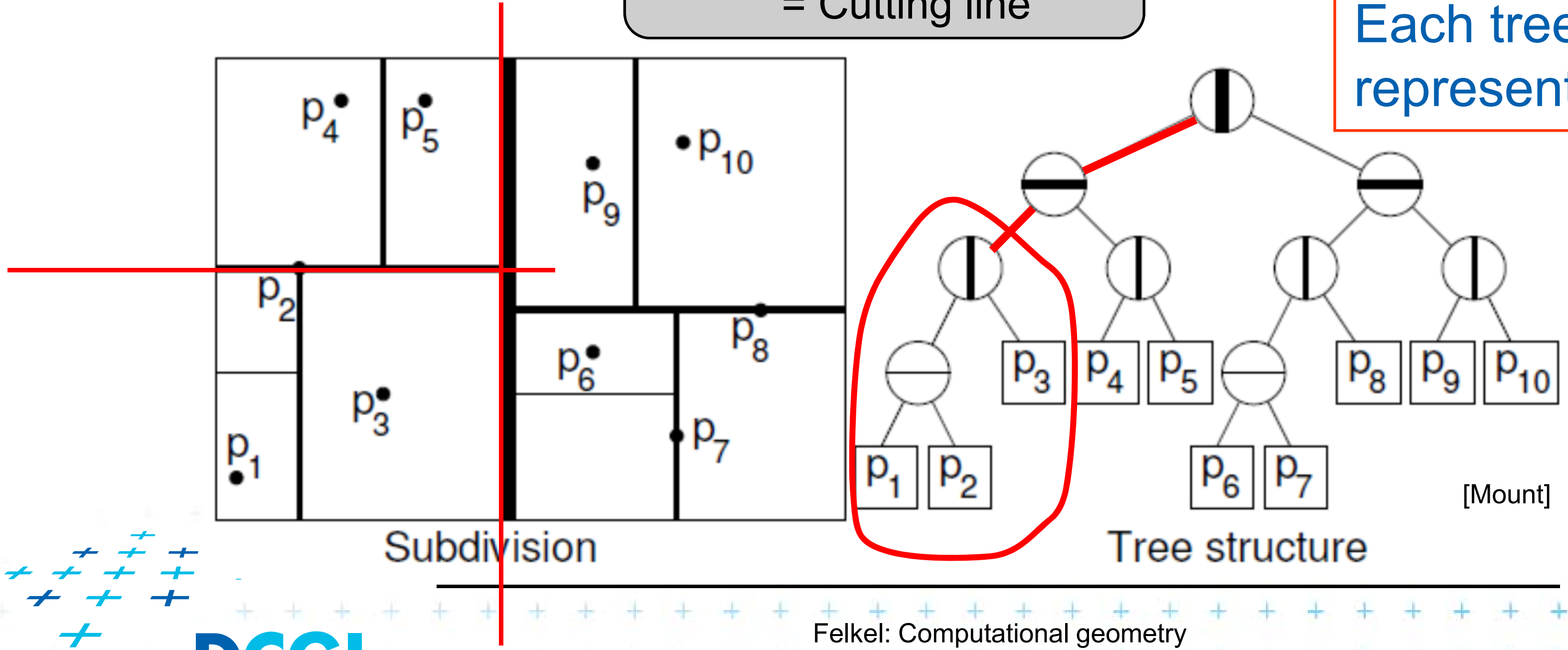
Tree structure

Kd-tree principle

- Subdivide space according to different dimension (x-coord, then y-coord, …)
- This subdivides space into rectangular cells
  => hierarchical decomposition of space
- In node *t* store: cutDim, cutVal, (size (for counting queries))

= Cutting line

Each tree node represents a region



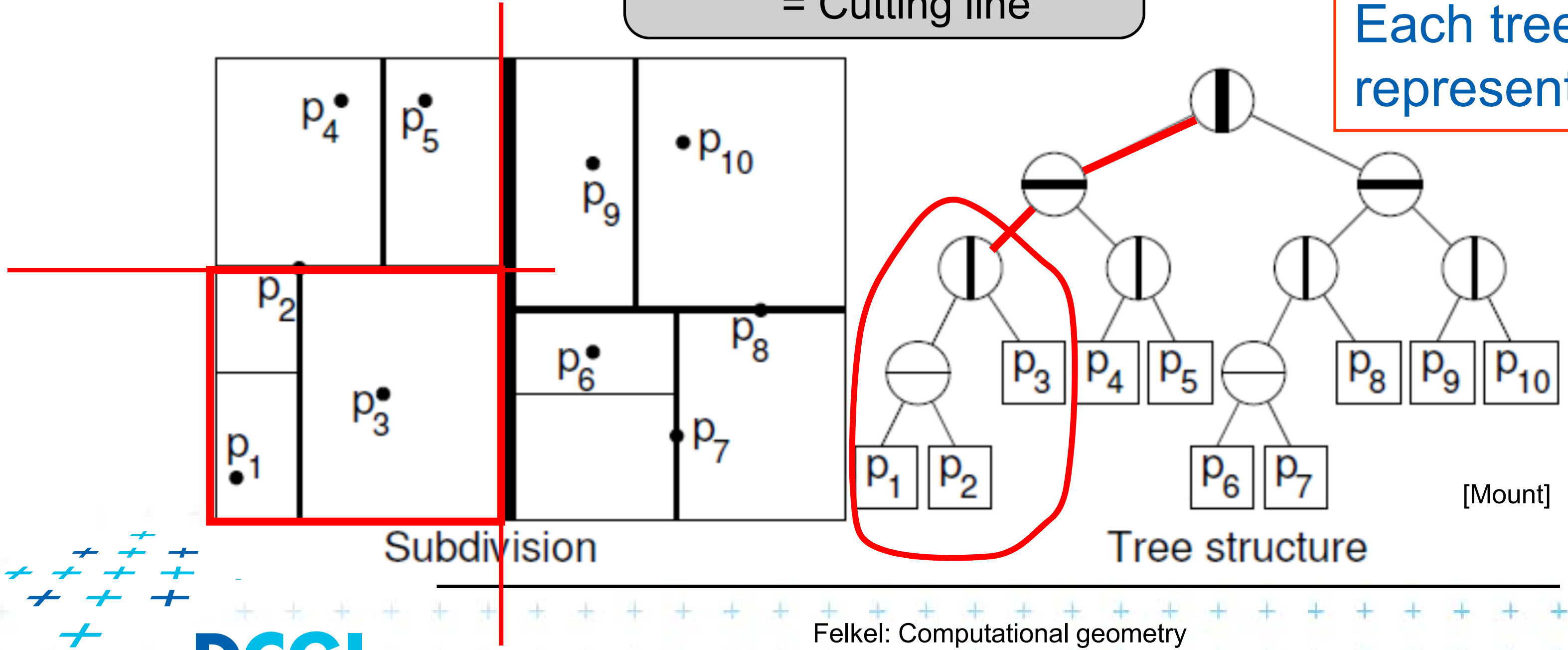
[Mount]

Kd-tree principle

- Subdivide space according to different dimension (x-coord, then y-coord, …)
- This subdivides space into rectangular cells => hierarchical decomposition of space
- In node *t* store: cutDim, cutVal, (size (for counting queries))

= Cutting line

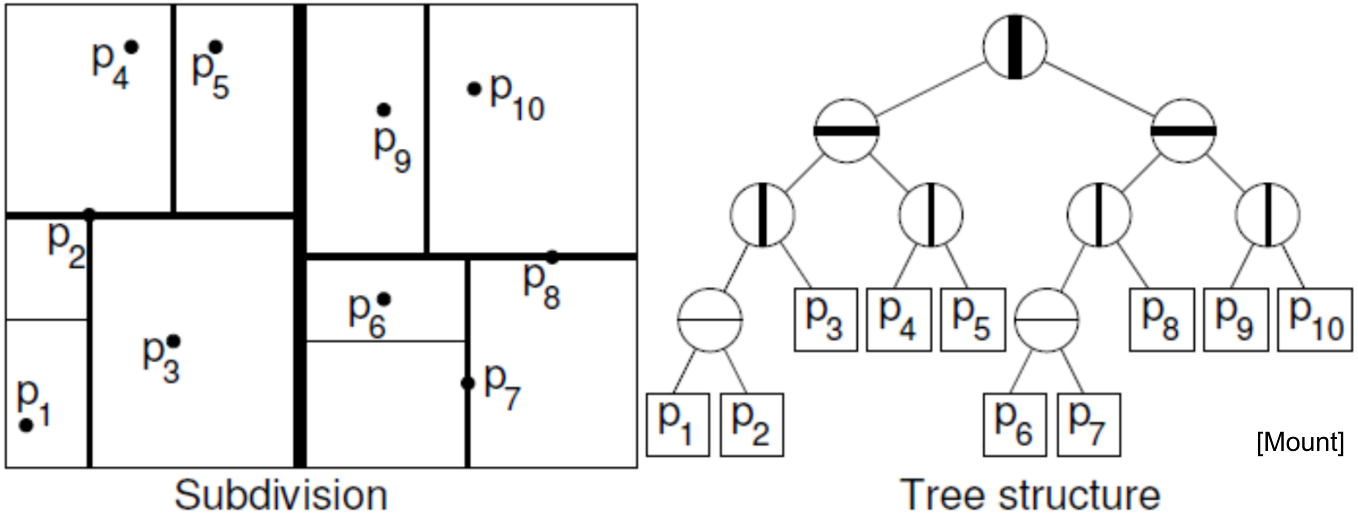
Each tree node represents a region



Kd-tree principle

- Subdivide space according to different dimension (x-coord, then y-coord, …)
- This subdivides space into rectangular cells => hierarchical decomposition of space
- In node *t* store: cutDim, cutVal, (size (for counting queries))

= Cutting line

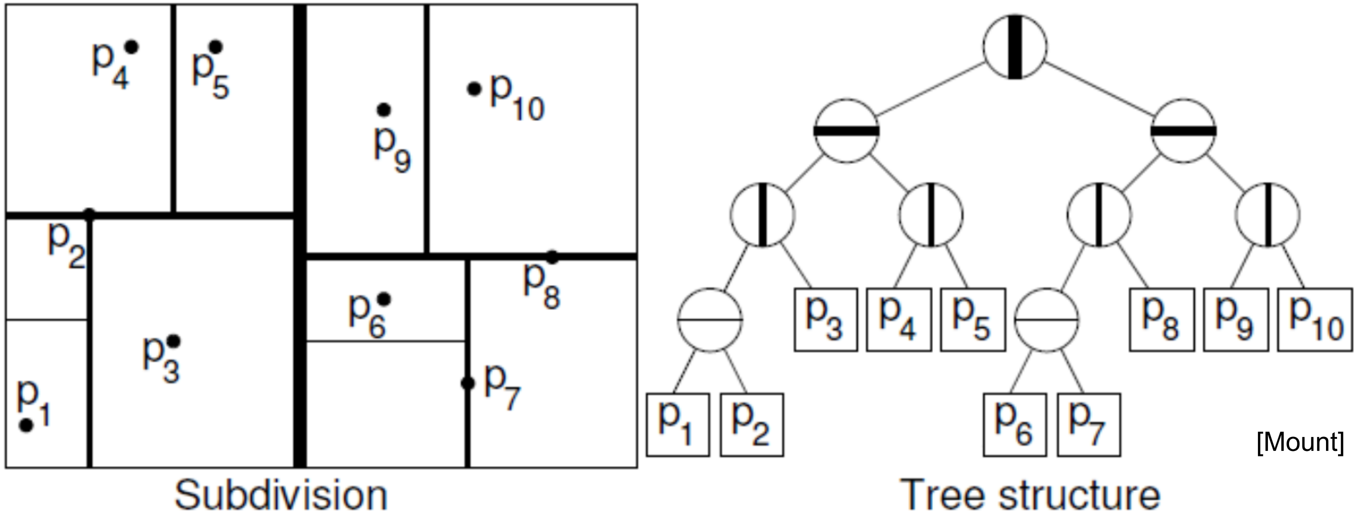


[Mount]

Kd-tree principle

- Subdivide space according to different dimension (x-coord, then y-coord, …)
- This subdivides space into rectangular cells => hierarchical decomposition of space
- In node *t* store: cutDim, cutVal, (size (for counting queries))

= Cutting line



[Mount]

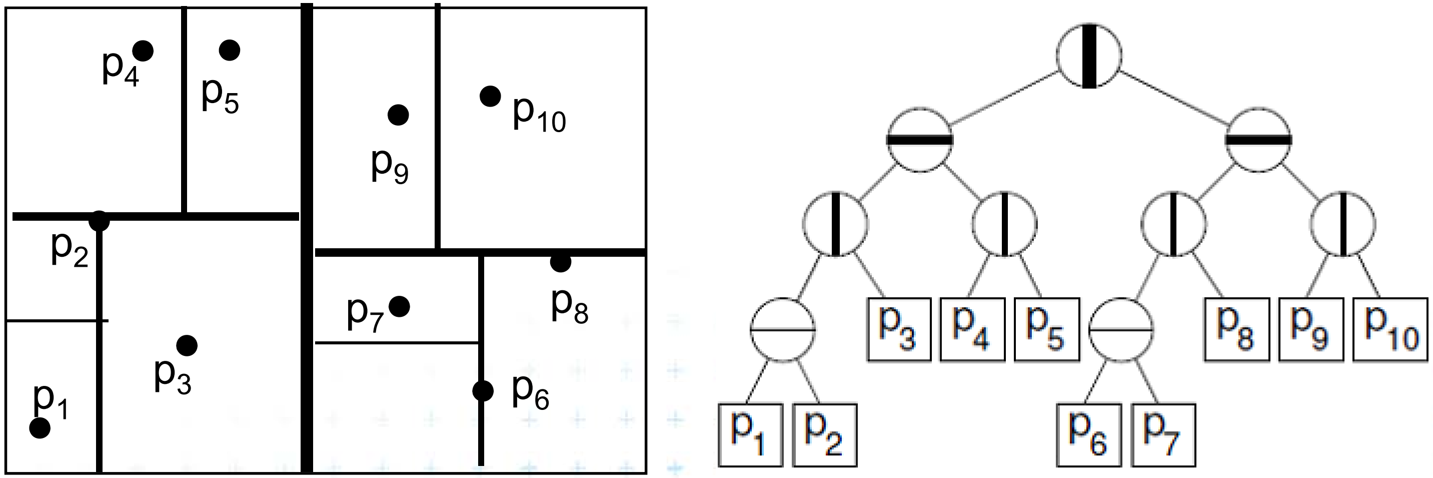
Where is a mistake in the figure?

Kd-tree principle

- Which dimension to cut? (cutDim)
  - Cycle through dimensions (round robin)
    - Save storage – cutDim is implicit ~ depth in the tree
    - May produce elongated cells (if uneven data distribution)
  - Greatest spread (the largest difference of coordinates)
    - Adaptive
    - Called “Optimal kd-tree”
- Where to cut? (cutVal)
  - Median, or midpoint between upper and lower median -> $O(n)$
  - Presort coords of points in each dimension $(x, y, \dots)$ for $O(1)$ median – resp. $O(d)$ for all d dimensions

Kd-tree principle

- What about points on the cell boundary?
  - Boundary belongs to the left child
  - Left: $p_{\text{cutDim}} \leq \text{cutVal}$
  - Right: $p_{\text{cutDim}} > \text{cutVal}$



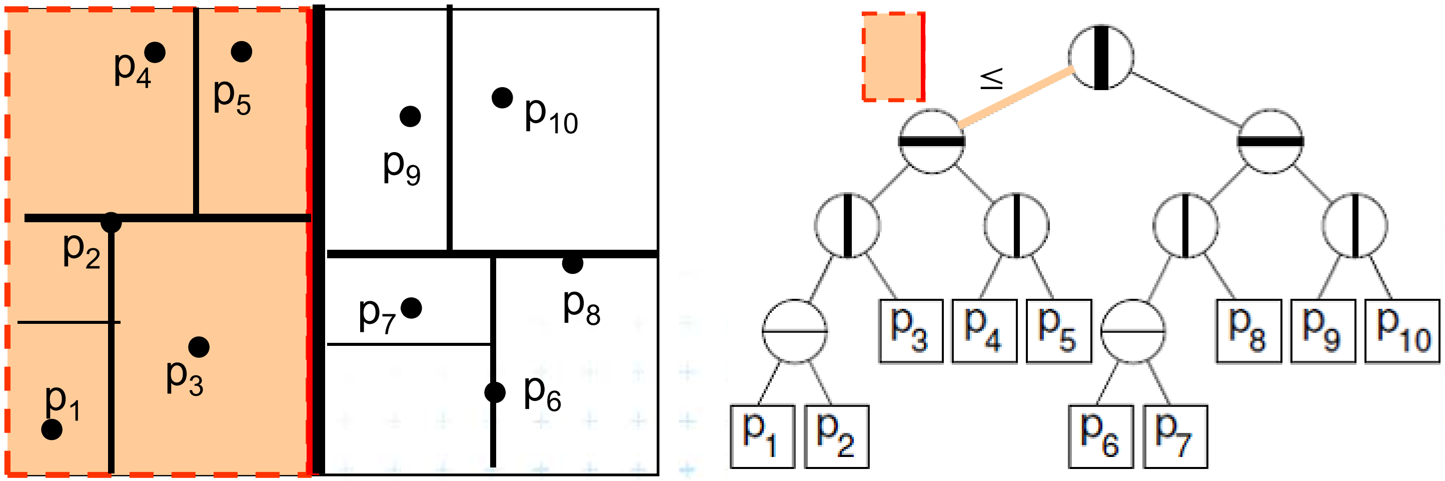
Subdivision

Tree structure

[Mount]

Kd-tree principle

- What about points on the cell boundary?
  - Boundary belongs to the left child
  - Left: $p_{cutDim} \le cutVal$
  - Right: $p_{cutDim} > cutVal$



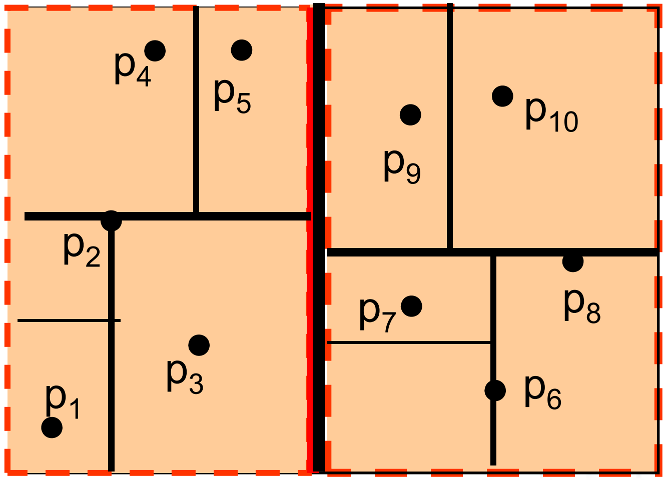
Subdivision

Tree structure

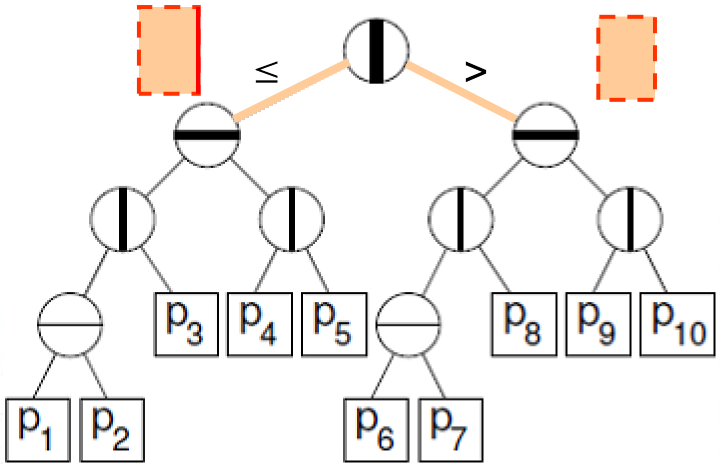
[Mount]

Kd-tree principle

- What about points on the cell boundary?
  - Boundary belongs to the left child
  - Left: $p_{\text{cutDim}} \leq \text{cutVal}$
  - Right: $p_{\text{cutDim}} > \text{cutVal}$



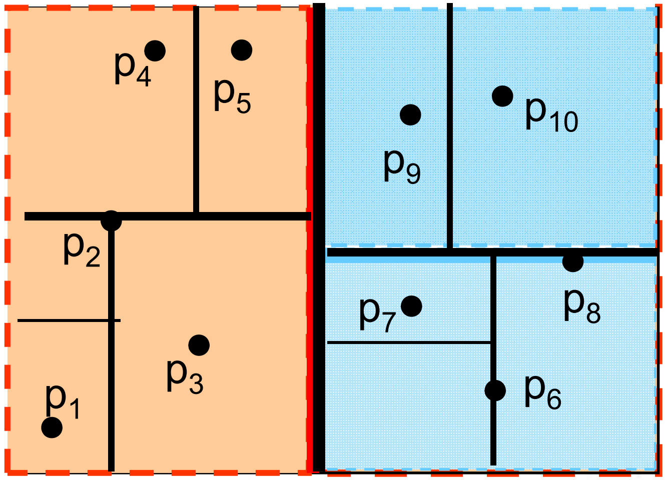
Subdivision



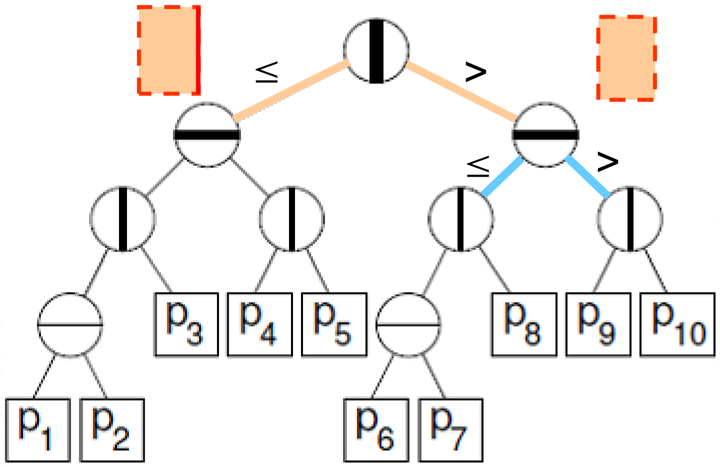
Tree structure [Mount]

Kd-tree principle

- What about points on the cell boundary?
  - Boundary belongs to the left child
  - Left: $p_{cutDim} \leq cutVal$
  - Right: $p_{cutDim} > cutVal$



Subdivision

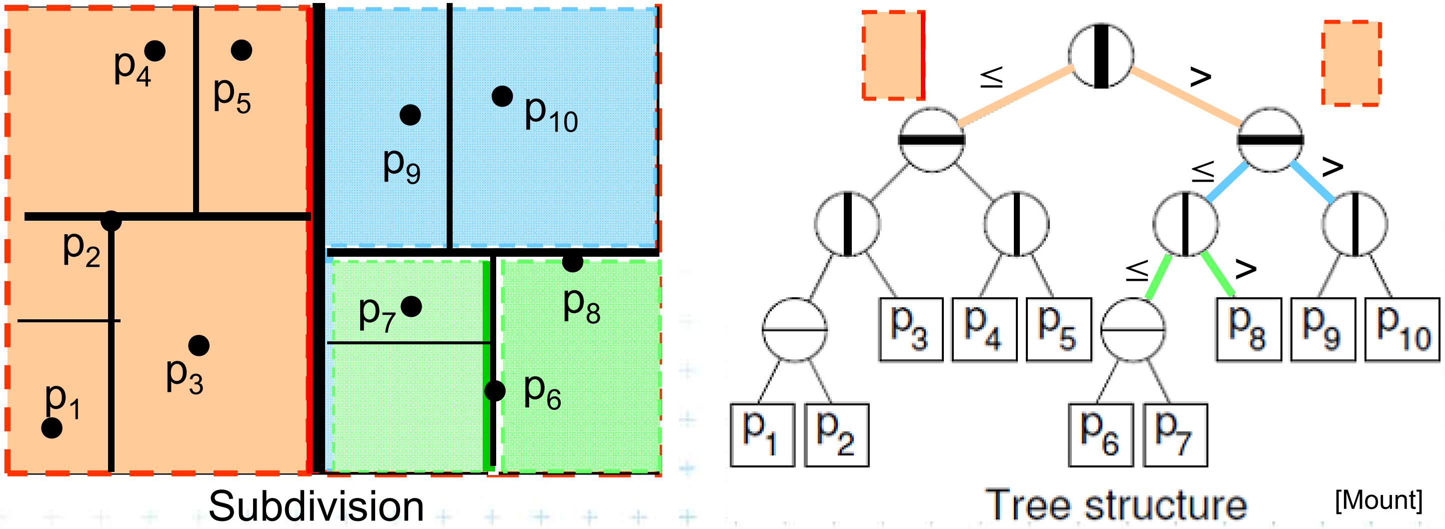


Tree structure

[Mount]

Kd-tree principle

- What about points on the cell boundary?
  - Boundary belongs to the left child
  - Left: $p_{cutDim} \leq \text{cutVal}$
  - Right: $p_{cutDim} > \text{cutVal}$



[Mount]

Kd-tree construction in 2-dimensions

BuildKdTree(*P, depth*)

Input: A set of points *P* and current *depth.*

Output: The root of a kD tree storing P.

```
If ($P$ contains only one point)   [or small set of (10 to 20) points]
    then return a leaf storing this point
    else if ($depth$ is even)
        then split $P$ with a vertical line $l$ through median $x$ into two subsets
                $P_1$ and $P_2$ (left and right from median)
        else split $P$ with a horiz. line $l$ through median y into two subsets
                $P_1$ and $P_2$ (below and above the median)
    $t_{\text{left}}$ = BuildKdTree($P_1$, $depth+1$)
    $t_{\text{right}}$ = BuildKdTree($P_2$, $depth+1$)
    create node $t$ storing $l$, $t_{\text{left}}$ and $t_{\text{right}}$ children     // l = cutDim, cutVal
    return $t$
```

If median found in O(1) and array split in O(n)

$T(n) = 2\,T(n/2) + n \Rightarrow O(n \log n)$ construction

Kd-tree construction in 2-dimensions

BuildKdTree(*P, depth*)

Input: A set of points *P* and current *depth.*

Output: The root of a kD tree storing P.

Split according to (*depth%max_dim)* dimension

```
If (P contains only one point)   [or small set of (10 to 20) points]
    then return a leaf storing this point
    else if (depth is even)
        then split P with a vertical line l through median x into two subsets
                $P_1$ and $P_2$ (left and right from median)
        else split P with a horiz. line l through median y into two subsets
                $P_1$ and $P_2$ (below and above the median)
        $t_{left}$ = BuildKdTree($P_1$, depth+1)
        $t_{right}$ = BuildKdTree($P_2$, depth+1)
        create node t storing l, $t_{left}$ and $t_{right}$ children      // l = cutDim, cutVal
        return t
```

If median found in O(1) and array split in O(n)

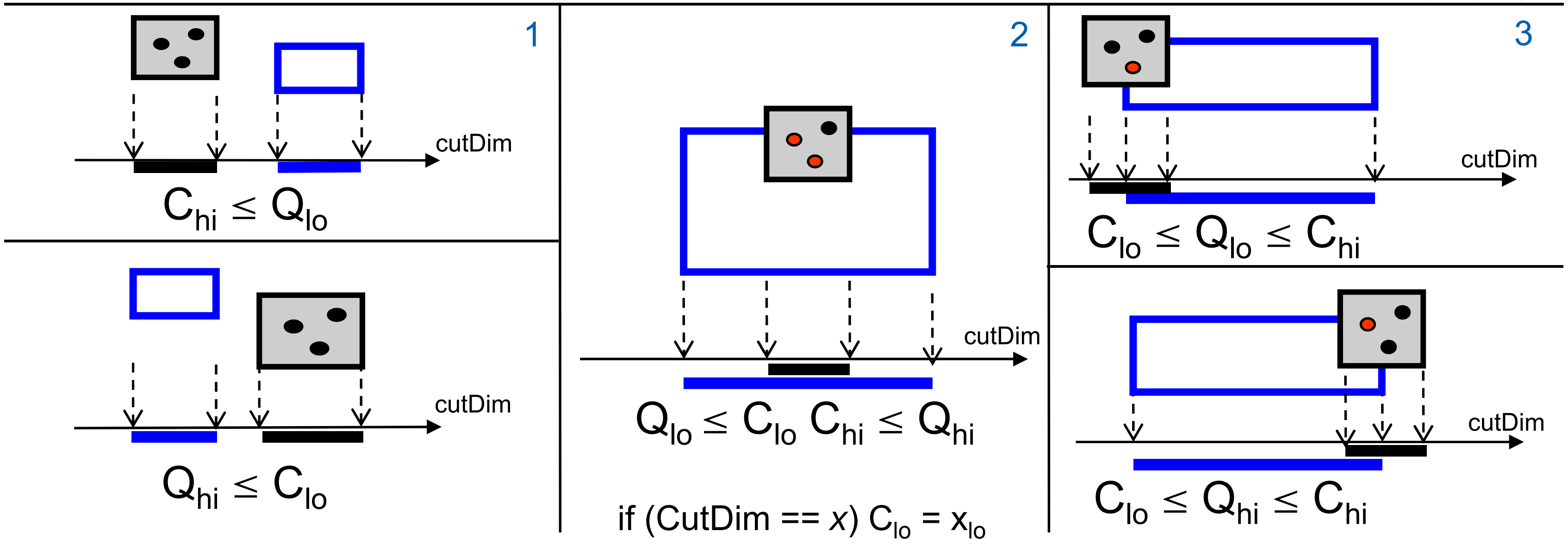
T(n) = 2 T(n/2) + n => O(*n* log *n*) construction

Kd-tree test variants

Test interval-interval

a) Compare rectang. array Q with rectangular cells C

- Rectangle C:[x_{lo}, x_{hi}, y_{lo}, y_{hi}] – computed on the fly
- Test of kD node cell C against query Q (in one cutDim)
  1. if cell is disjoint with Q … $C \cap Q = \emptyset$ … stop
  2. If cell C completely inside Q … $C \subseteq Q$ … stop and report cell points
  3. else cell C overlaps Q … recurse on both children
- Recursion stops on the largest subtree (in/out)



1
cutDim
$C_{hi} \le Q_{lo}$
cutDim
$Q_{hi} \le C_{lo}$
2
cutDim
$Q_{lo} \le C_{lo}$ $C_{hi} \le Q_{hi}$
if (CutDim == x) $C_{lo} = x_{lo}$
3
cutDim
$C_{lo} \le Q_{lo} \le C_{hi}$
cutDim
$C_{lo} \le Q_{hi} \le C_{hi}$

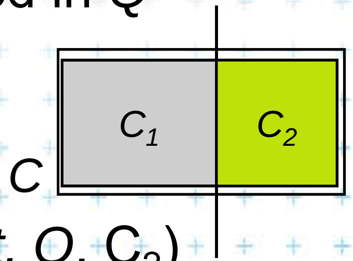
Kd-tree rangeCount (with rectangular cells)

int rangeCount(*t*, *Q*, *C*)

Input: The root *t* of kD tree, query range *Q and t's* cell C.

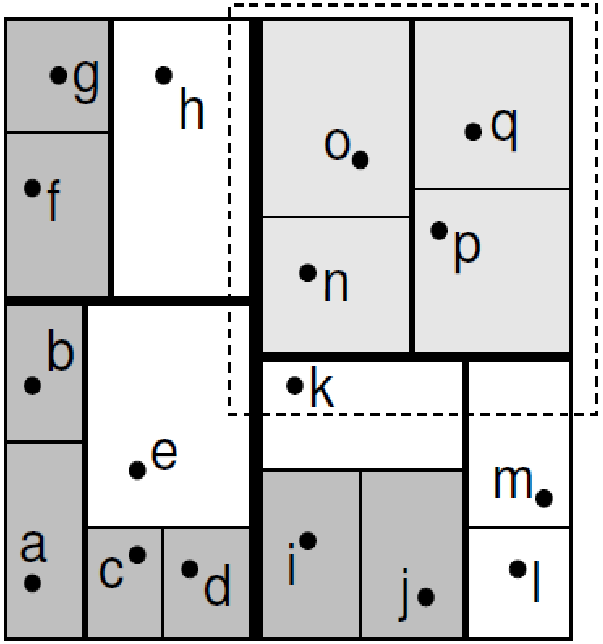
Output: Number of points at leaves below *t* that lie in the range.

```
if (t is a leaf)
    if (t.point lies in Q) return 1   k ❶   // or loop this test for all points in leaf
    else return 0                     l     // visited, not counted
else  // (t is not a leaf)
    if ($C \cap Q = \emptyset$)  return 0         ●      … disjoint
    else if ($C \subseteq Q$)  return t.size       ◯ ❹   … C is fully contained in Q
    else                                           ○
        split C along t's cutting value and dimension,
        creating two rectangles $C_1$ and $C_2$.
        return rangeCount(t.left, Q, $C_1$) + rangeCount(t.right, Q, $C_2$)
```



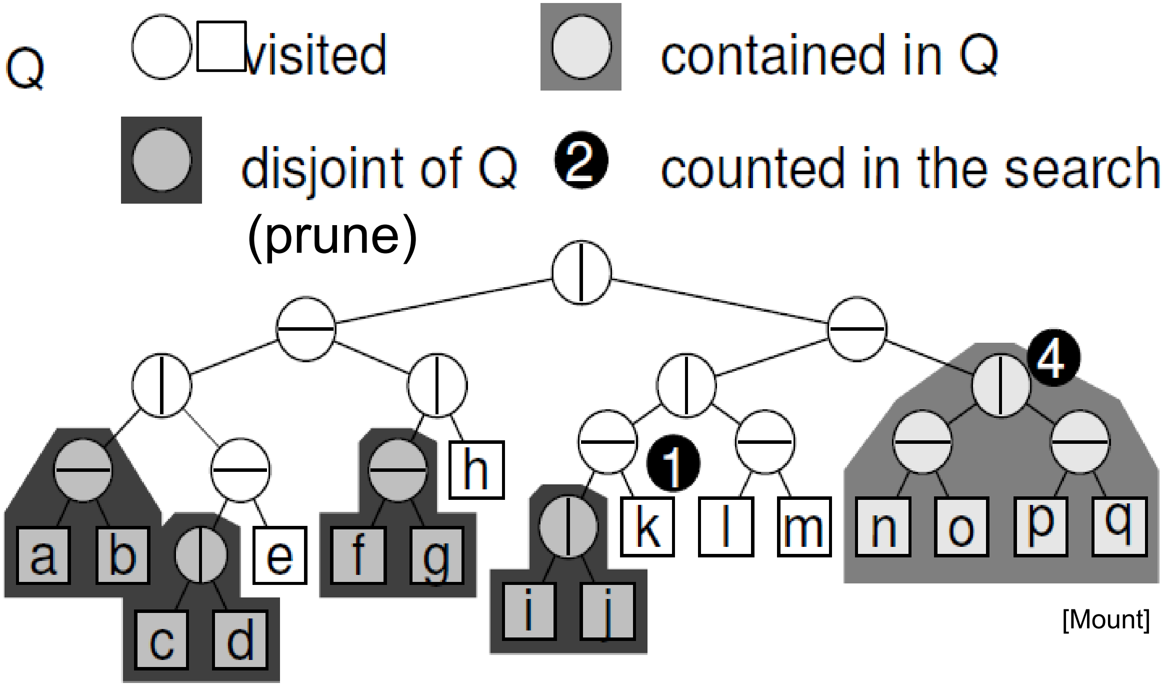
// (pictograms refer to the next slide)

Kd-tree rangeCount example



kd–tree subdivision

Tree node (rectangular region)



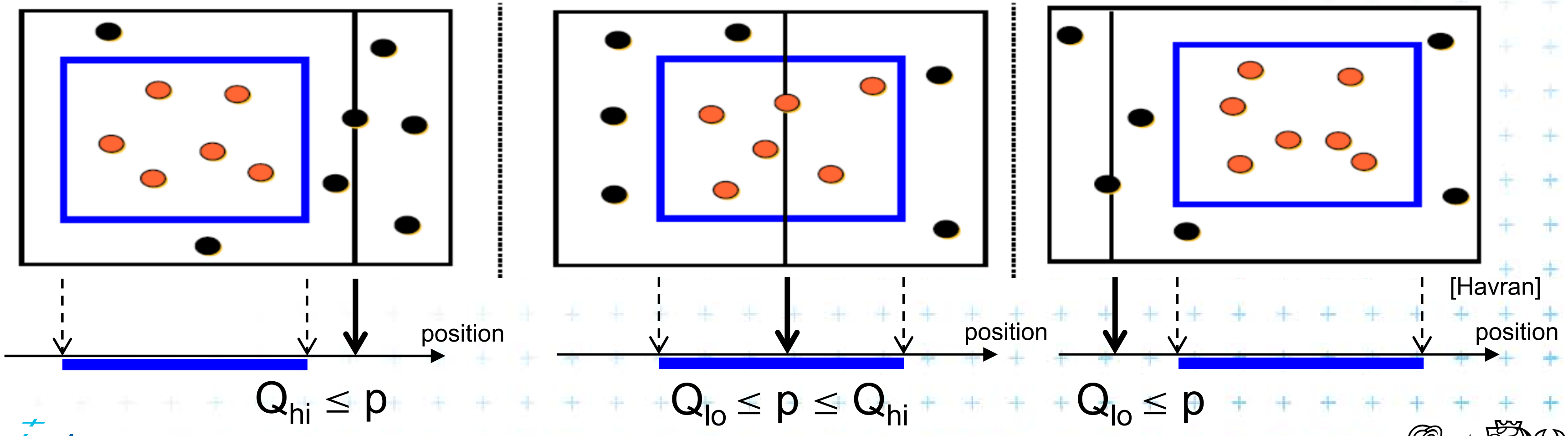
Nodes visited in range search

Kd-tree test variants

Test point-interval

b) Compare Q with cutting lines

- Line = Splitting value *p* in one of the dimensions
- Test of single position given by dimension against Q
  1. Line *p* is right from Q … recurse on left child only (prune right child)
  2. Line *p* intersects Q … recurse on both children
  3. Line *p* is left from Q … recurse on right child only (prune left ch.)
- Recursion stops in leaves - traverses the whole tree



[Havran]

Kd-tree rangeSearch (with cutting lines)

int rangeSearch(*t*, *Q*)

Input: The root *t* of (a subtree of a) kD tree and query range *Q*.

Output: Points at leaves below *t* that lie in the range.

```
if (t is a leaf)
    if (t.point lies in Q) report t.point   // or loop test for all points in leaf
    else return
else  (t is not a leaf)
    if ($Q_{hi} \le$ t.cutVal) rangeSearch(t.left, Q)   // go left only
    if ($Q_{lo} >$ t.cutVal) rangeSearch(t.right, Q) // go right only
    else
        rangeSearch(t.left, Q)                       // go to both
        rangeSearch(t.right, Q)
```

Kd-tree - summary

- Orthogonal range queries in the plane (in balanced 2d-tree)
  - Counting queries O(√n) time
  - Reporting queries O(√n + k) time, where k = No. of reported points
  - Space O(n)
  - Preprocessing: Construction O(n log n) time (Proof: if presorted points to arrays in dimensions. Median in O(1) and split in O(n) per level, log n levels of the tree)
- For d≥2:
  - Construction O(d n log n), space O(dn), Search O(d n^(1-1/d) + k)

Proof sqrt(n)

Každé sudé patro se testuje osa x.

- V patře 0 je jeden uzel a jde se do obou synů (v patře 1 se jde taky do obou)
- v patře 2 jsou 4 uzly, z nich jsou ale 2 bud úplně mimo, nebo úplně in => stab jen 2
- v 4. patře stab 4 z 8, …
- v i-tém patře stab 2^i uzlů

Výška stromu je log n

Proto tedy sčítám sudé členy z 0..log n z 2^i. Je to exponenciála, proto dominuje poslední člen

2^(log n /2) = 2^log (sqrt(n)) = sqrt(n)

Orthogonal range tree (RT)

- DS highly tuned for orthogonal range queries
- Query times in plane

2d tree	versus	2d range tree
O(√n + k) time of Kd	>	O(log *n*) time query
O(*n*) space of Kd	<	O(*n* log *n*) space

n = number of points

k = number of reported points

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, Chapter 5, http://www.cs.uu.nl/geobook/
- [Mount] David Mount, - CMSC 754: Computational Geometry, Lecture Notes for Spring 2007, University of Maryland , Lectures 17 and 18. http://www.cs.umd.edu/class/spring2007/cmsc754/lectures.shtml
- [Havran] Vlastimil Havran, Materiály k předmětu Datové struktury pro počítačovou grafiku, přednáška č. 6, Proximity search and its Applications 1, CTU FEL, 2007

DCGI

DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

CONVEX HULLS

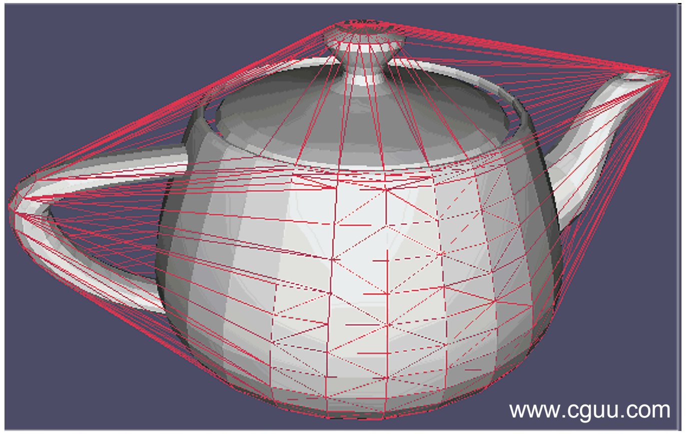
PETR FELKEL

FEL CTU PRAGUE

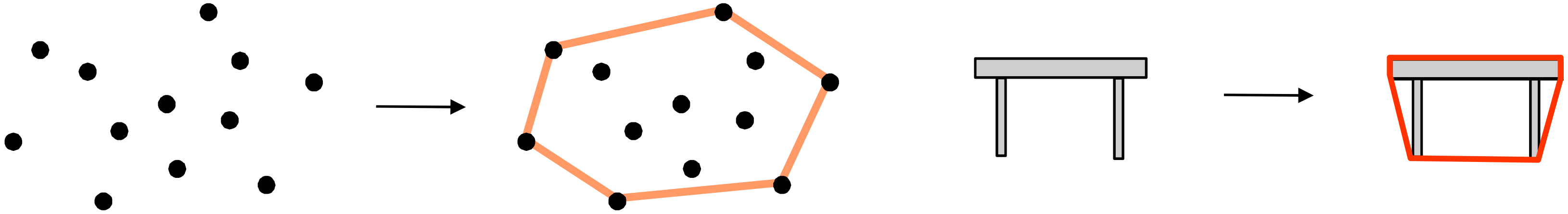
Version from 16.11.2017

Talk overview

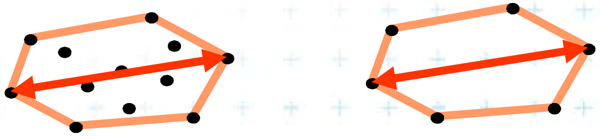
- Motivation and Definitions
- Graham’s scan – incremental algorithm
- Divide & Conquer
- Quick hull
- Jarvis’s March – selection by gift wrapping
- Chan’s algorithm – optimal algorithm

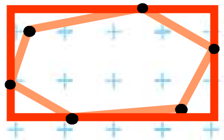

www.cguu.com

Convex hull (CH) – why to deal with it?



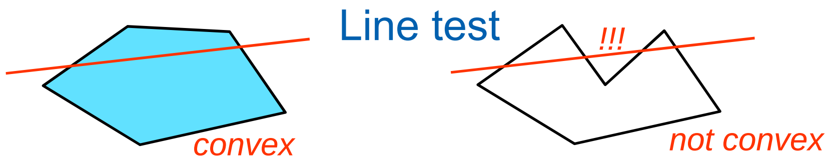
- *Shape approximation* of a point set or complex shapes (other common approximations include: minimal area enclosing rectangle, circle, and ellipse,…) – e.g., for collision detection
- *Initial stage* of many algorithms to filter out irrelevant points, e.g.:
  - diameter of a point set
  - minimum enclosing convex shapes (such as rectangle, circle, and ellipse) depend only on points on CH



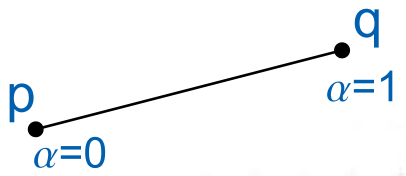


Convexity

- A set *S* is *convex*



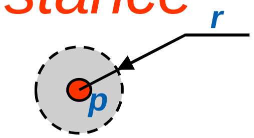
  - if for any points p,q $\in S$ the lines segment $\overline{pq} \subseteq S$, or
  - if any convex combination of *p* and *q* is in *S*
- Convex combination of points *p, q* is any point that can be expressed as $(1 - \alpha)\, p + \alpha q$, where $0 \le \alpha \le 1$



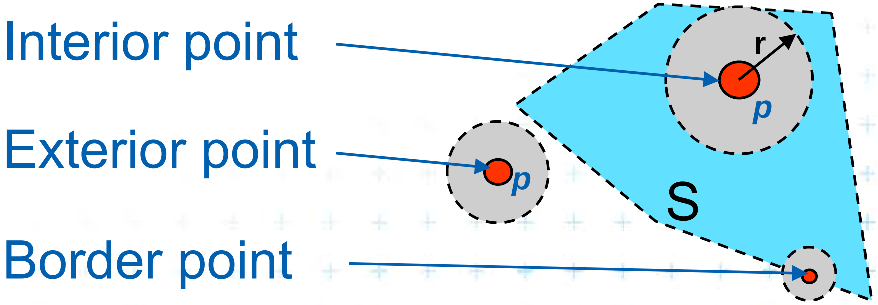
- *Convex hull* CH(S) of set *S* – is (similar definitions)
  - the smallest set that contains *S (convex)*
  - or: intersection of all convex sets that contain *S*
  - Or in 2D for points: the smallest convex polygon containing all given points

Definitions from topology in metric spaces

- *Metric space* – each two of points have defined a *distance*
- *r-neighborhood* of a point *p* and radius *r > 0* = set of points whose distance to *p* is strictly less than *r* (open ball of diameter *r* centered about *p*)

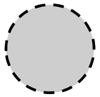
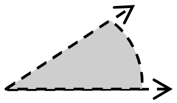
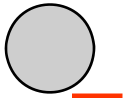
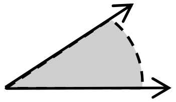


- Given set S, point *p* is
  - Interior point of S – if $\exists r, r > 0$, (r-neighborhood about *p*) ⊂ S
  - Exterior point – if it lies in interior of the complement of S
  - Border point – is neither interior neither exterior



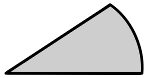
Definitions from topology in metric spaces

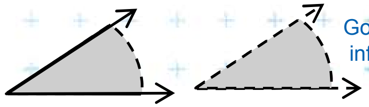
Are border points p ∈ *S*?

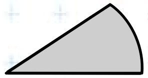
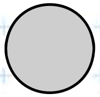
- Set S is Open (otevřená)  
  - – ∀p ∈ S ∃ (*r*-neighborhood about *p* of radius *r*) ⊆ S
  - – it contains only interior points, none of its border points
- Closed (uzavřená)  
  - – If it is equal to its closure $\overline{S}$ (uzávěr = smallest closed set containing S in topol. space)
    ∀(*r*-neighborhood about *p* of radius *r*) ∩ S ≠ ∅)
- Clopen (otevřená i uzavřená) – Ex. Empty set ∅, finite set of disjoint components
  - – if it is both closed and open
    (S= all positive rational numbers whose square is bigger than 2)

    space Q = rational numbers
    S = (√2, ∞) in Q, √2 ∉ Q, S = $\overline{S}$

Goes to infinity?

- Bounded (ohraničená) 

  Unbounded 

  - – if it can be enclosed in a ball of finite radius
- Compact (kompaktní)  
  - – if it is both closed and bounded

Clopen (otevřená i uzavřená)

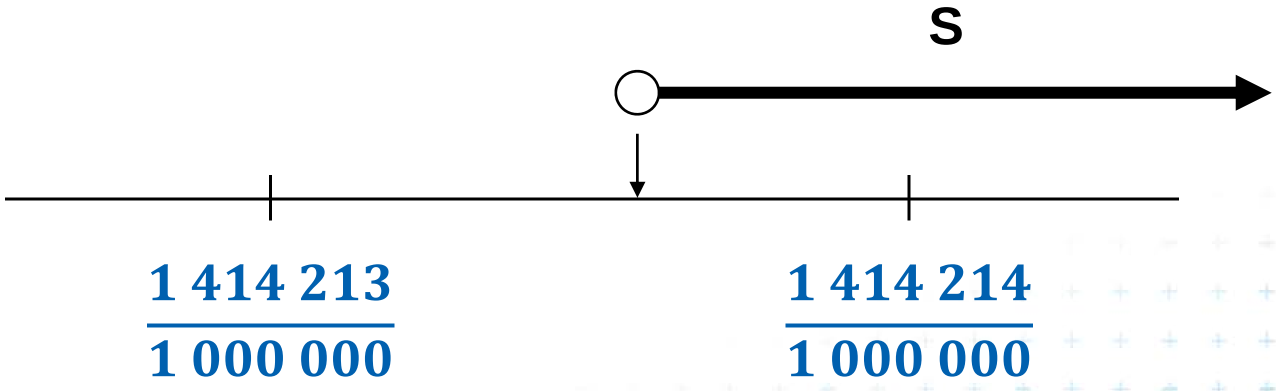
– Ex. Empty set ∅, finite set of disjoint components

if it is both closed and open

space Q = rational numbers

(S= all positive rational numbers whose square is bigger than 2) S = (√2, ∞) in Q, √2 ∉ Q, S = S

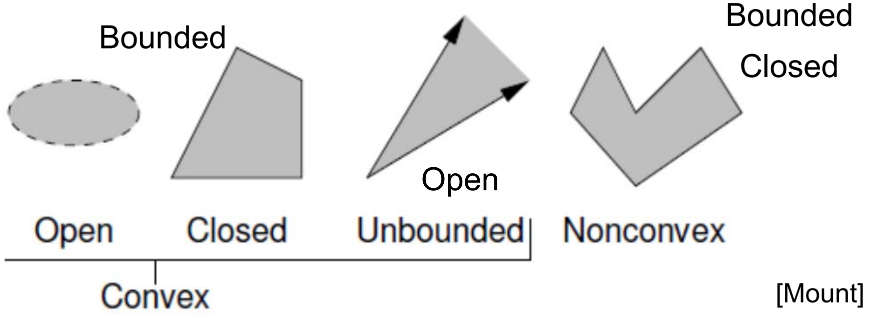
$$\sqrt{2} = 1.414213562$$



$\frac{1\,414\,213}{1\,000\,000}$ $\frac{1\,414\,214}{1\,000\,000}$

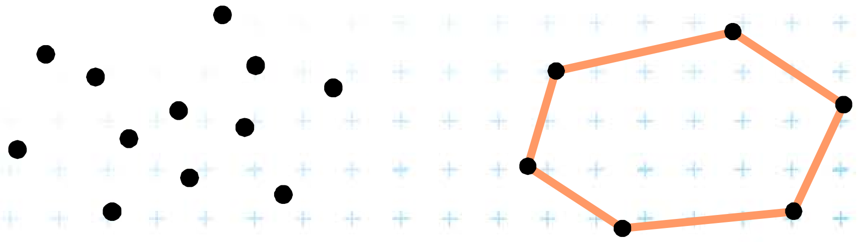
Definitions from topology in metric spaces

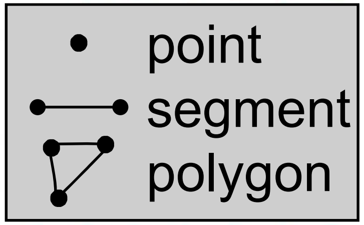
- *Convex set S may be bounded or unbounded*



- *Convex hull* CH(*S*) of a finite set *S* of points in the plane

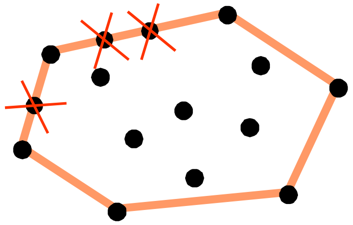
  = Bounded, closed, (= compact) convex polygon





Convex hull representation

- CCW enumeration of vertices
- Contains only the extreme points ("endpoints" of collinear points)



- Simplification for the whole semester:
  Assume the input points are in general position,
  - no two points have the same *x*-coordinates and
  - no three points are collinear

  -> We avoid problem with non-extreme points on *x*
  (solution may be simple – e.g. lexicographic ordering)

Online x offline algorithms

- Incremental algorithm
  - Proceeds one element at a time (step-by-step)
- Online algorithm (must be incremental)
  - is started on a partial (or empty) input and
  - continues its processing as additional input data becomes available (comes online, thus the name).
  - Ex.: insertion sort
- Offline algorithm (may be incremental)
  - requires the entire input data from the beginning
  - than it can start
  - Ex.: selection sort (any algorithm using sort)

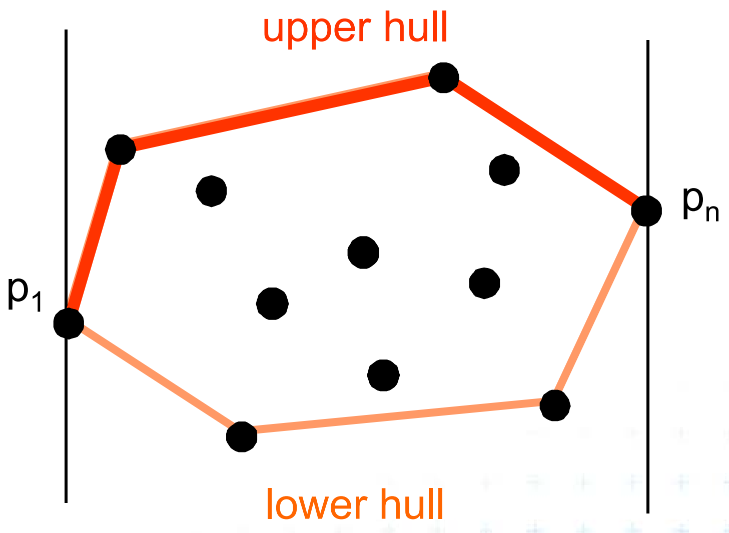
Graham’s scan

- Incremental O(*n* log *n*) algorithm
- Objects (points) are added one at a time
- Order of insertion is important
  1. Random insertion
     –> we need to test: *is-point-inside-the-hull(p)*
  2. Ordered insertion
     Find the point p with the smallest *y* coordinate first
     a) Sort points p_i according to *increasing angles* around the point p (angle of pp_i and x axis)
     b) Andrew’s modification: sort points p_i according to *x* and add them left to right (construct upper & lower hull)

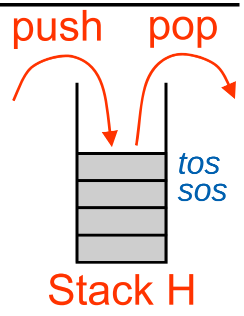
     Sorting *x-coordinates* is simpler to implement than sorting of angles

Graham's scan – b) modification by Andrew

- O(*n* log *n*) for unsorted points, O(*n*) for sorted pts.
- Upper hull, then lower hull. Merge.
- Minimum and maximum on *x* belong to CH



Graham’s scan – incremental algorithm

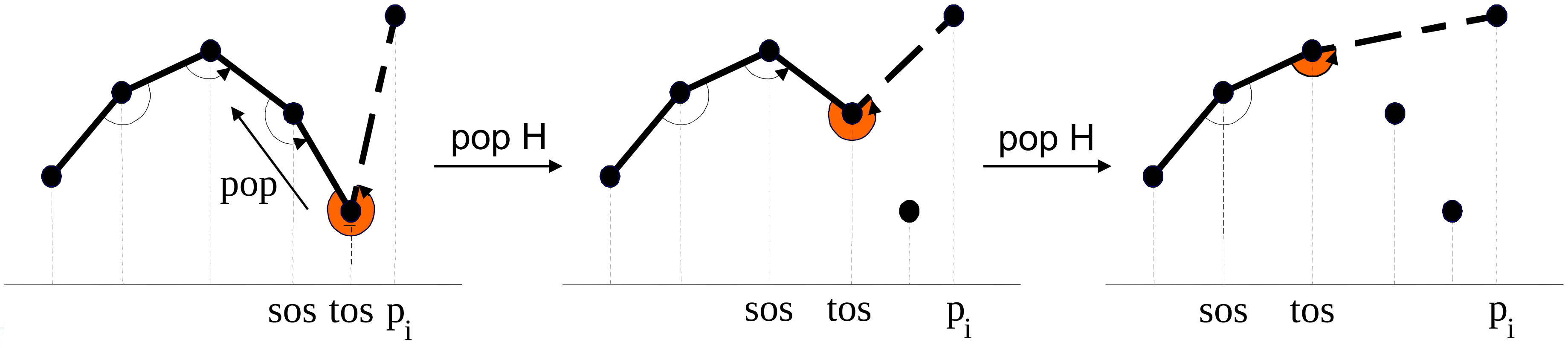


GrahamsScan(points p)

Input: points p

Output: CCW points on the convex hull

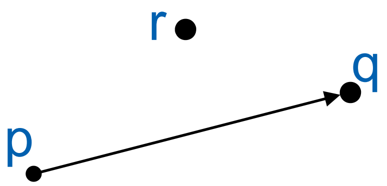
```
sort points according to increasing x-coord -> {$p_1$, $p_2$, …, $p_n$}
push( $p_1$, H), push( $p_2$, H )                                      upper hull
for $i$ = 3 to $n$ do
    while( size(H) $\geq$ 2 and orient( sos, tos, $p_i$ ) $\geq$ 0 )   // skip left turns
        pop H                                                          // (back-tracking)
    push( $p_i$, H )                                                   // store right turn
store H to the output (in reverse order)   // upper hull
Symmetrically the lower hull
```

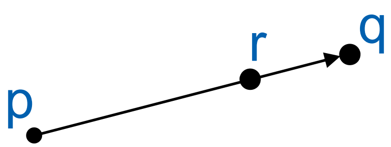


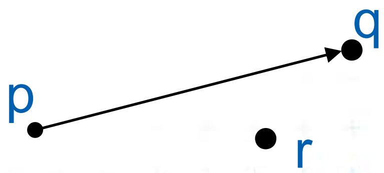
Position of point in relation to segment

$$\text{orient}(\,p, q, r\,) \begin{cases} > 0 & r \text{ is left from } pq\text{, CCW orient} \\ = 0 & \text{if } (\,p, q, r\,) \text{ are collinear} \\ < 0 & r \text{ is right from } pq\text{, CW orient} \end{cases}$$

Point *r* is: **left from *pq*** **on segment *pq*** **right from *pq***

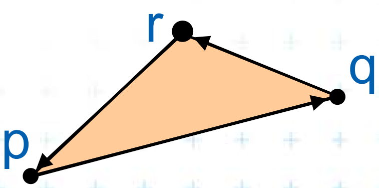


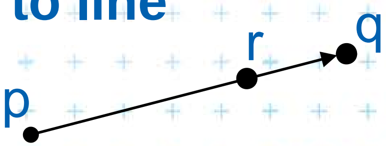


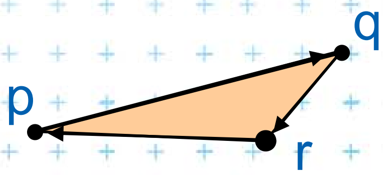


Convex polygon with edges pq and qr or

Triangle *pqr*: is CCW oriented **degenerated to line** **is CW oriented**



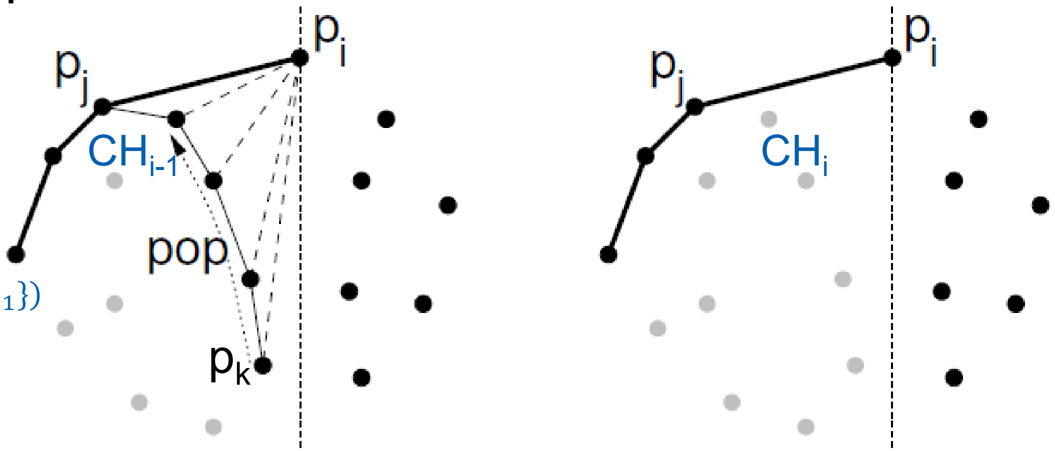




Is Graham’s scan correct?

Stack H at any stage contains upper hull of the points $\{p_1, \dots, p_j, p_i\}$, processed so far

- For induction basis $H = \{p_1, p_2\}$ … true
- p_i = last added point to CH, p_j = its predecessor on CH
- Each point p_k that lies between p_j and p_i lies below $p_j p_i$ and should not be part of UH after addition of p_i => is removed before push p_i. [orient$(p_j, p_k, p_i) > 0$, p_k is right from $p_j p_i \Rightarrow p_k$ is removed from UH]
- Stop, if 2 points in the stack or after construction of the upper hull



Points on stack H = CH $(\{p_1, p_2, \dots, p_{i-1}\})$

[Mount]

Complexity of Graham's scan

- Sorting according x – O(n log n)
- Each point pushed once – O(n)
- Some ($d_i \le n$) points deleted while processing p_i – O(n)
- The same for lower hull – O(n)

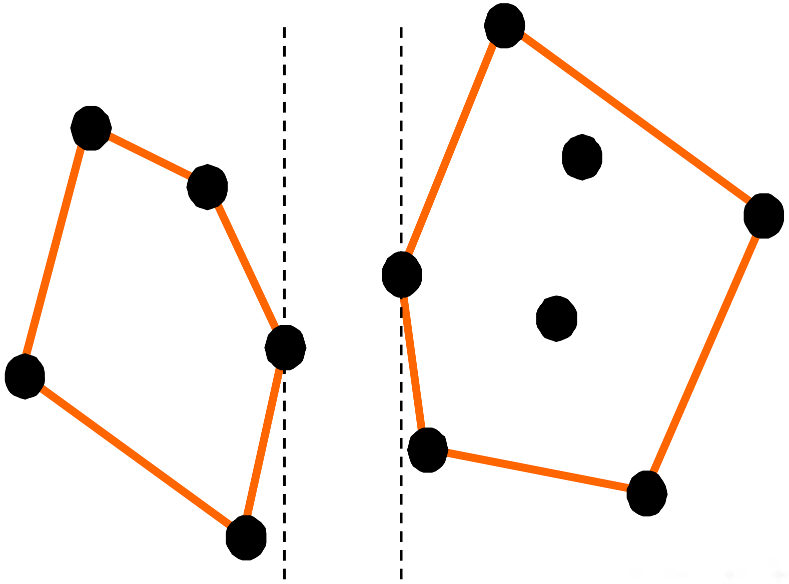
- Total O(n log n) for unsorted points
  O(n) for sorted points

Divide & Conquer

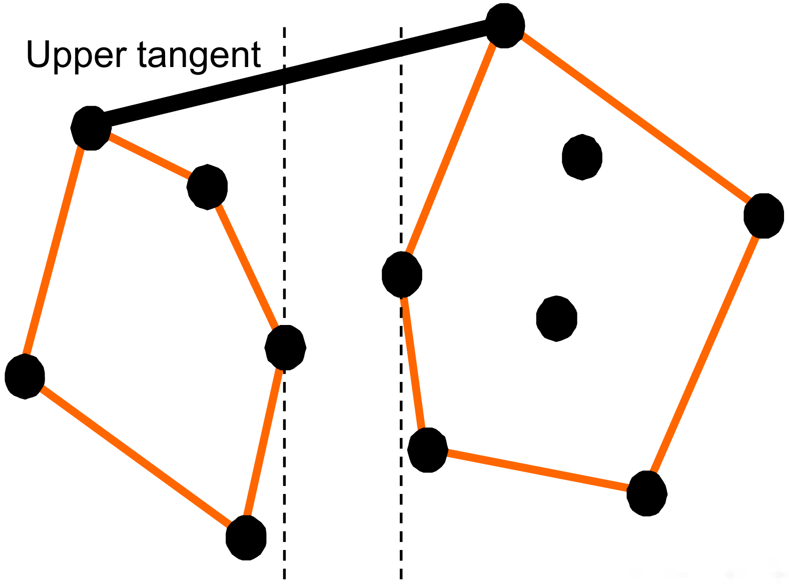
- $\Theta(n \log(n))$ algorithm
- Extension of mergesort
- Principle
  - Sort points according to *x*-coordinate,
  - recursively partition the points and solve CH.

Convex hull by D&C

```
ConvexHullD&C( points P )
Input:    points p
Output:   CCW points on the convex hull
Sort points P according to $x$
return hull( P )

hull( points P )
    if $|P| \le 3$ then
            compute CH by brute force,
            return
    Partition P into two sets L and R (with lower & higher coords $x$)
    Recursively compute $H_L$ = hull(L), $H_R$ = hull(R)
    H = Merge hulls($H_L$, $H_R$) by computing
        Upper_tangent( $H_L$, $H_R$) // find nearest points, $H_L$ CCW, $H_R$ CW
        Lower_tangent( $H_L$, $H_R$) // ($H_L$ CW, $H_R$ CCW)
        discard points between these two tangents
    return H
```



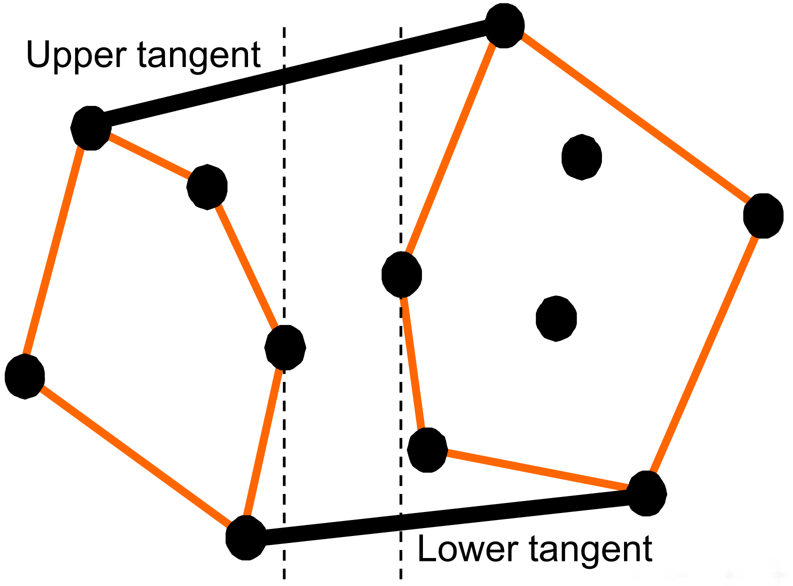
Convex hull by D&C



```
**ConvexHullD&C(** points **P )**
*Input:*    points p
*Output:*   CCW points on the convex hull
Sort points P according to *x*
return hull( P )

**hull( points P )**
    if |P| $\leq$ 3 then
            compute CH by brute force,
            return
    Partition P into two sets L and R (with lower & higher coords *x*)
    Recursively compute $H_L$ = hull(L), $H_R$ = hull(R)
    H = Merge hulls($H_L$, $H_R$) by computing
        Upper_tangent( $H_L$, $H_R$) // find nearest points, $H_L$ CCW, $H_R$ CW
        Lower_tangent( $H_L$, $H_R$) // ($H_L$ CW, $H_R$ CCW)
        discard points between these two tangents
    return H
```

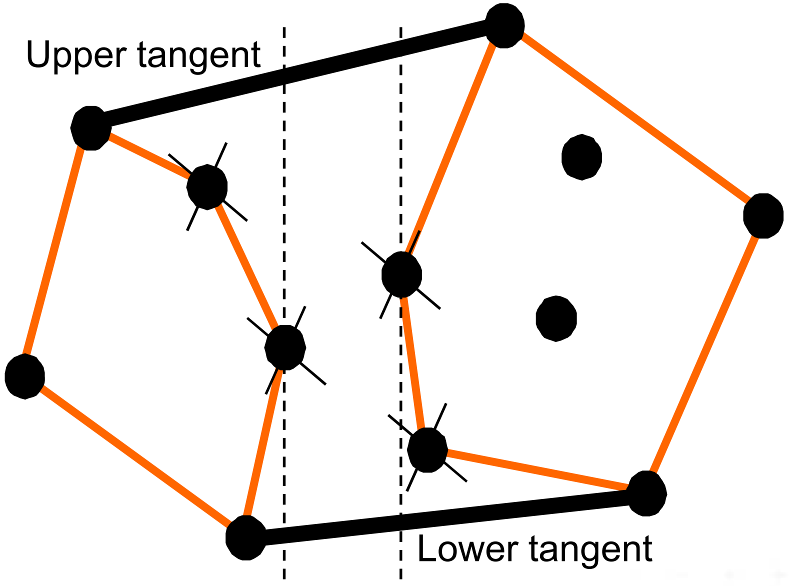
Convex hull by D&C

```
ConvexHullD&C( points P )
Input:    points p
Output:   CCW points on the convex hull
Sort points P according to $x$
return hull( P )

hull( points P )
    if $|P| \le 3$ then
            compute CH by brute force,
            return
    Partition P into two sets L and R (with lower & higher coords $x$)
    Recursively compute $H_L$ = hull(L), $H_R$ = hull(R)
    H = Merge hulls($H_L$, $H_R$) by computing
        Upper_tangent( $H_L$, $H_R$) // find nearest points, $H_L$ CCW, $H_R$ CW
        Lower_tangent( $H_L$, $H_R$) // ($H_L$ CW, $H_R$ CCW)
        discard points between these two tangents
    return H
```

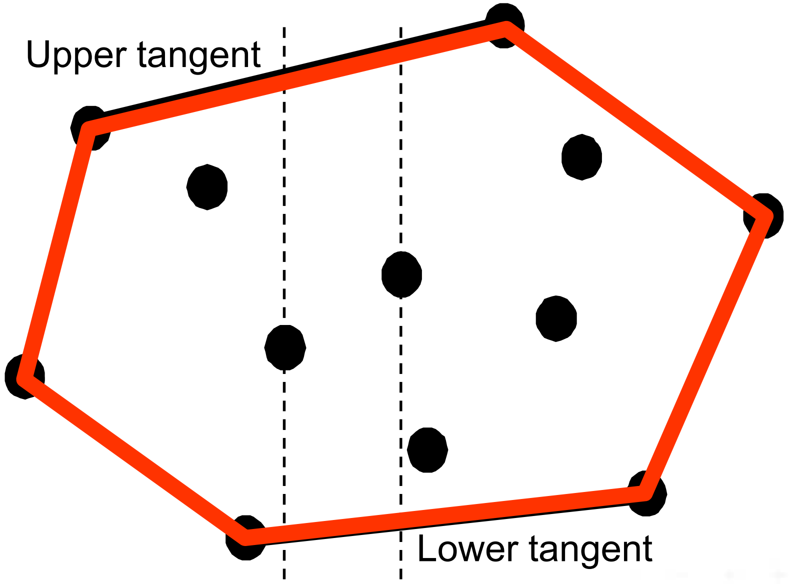


Convex hull by D&C

```
ConvexHullD&C( points P )
Input:    points p
Output:   CCW points on the convex hull
Sort points P according to $x$
return hull( P )

hull( points P )
    if $|P| \leq 3$ then
            compute CH by brute force,
            return
    Partition P into two sets L and R (with lower & higher coords $x$)
    Recursively compute $H_L$ = hull(L), $H_R$ = hull(R)
    H = Merge hulls($H_L$, $H_R$) by computing
        Upper_tangent( $H_L$, $H_R$) // find nearest points, $H_L$ CCW, $H_R$ CW
        Lower_tangent( $H_L$, $H_R$) // ($H_L$ CW, $H_R$ CCW)
        discard points between these two tangents
    return H
```

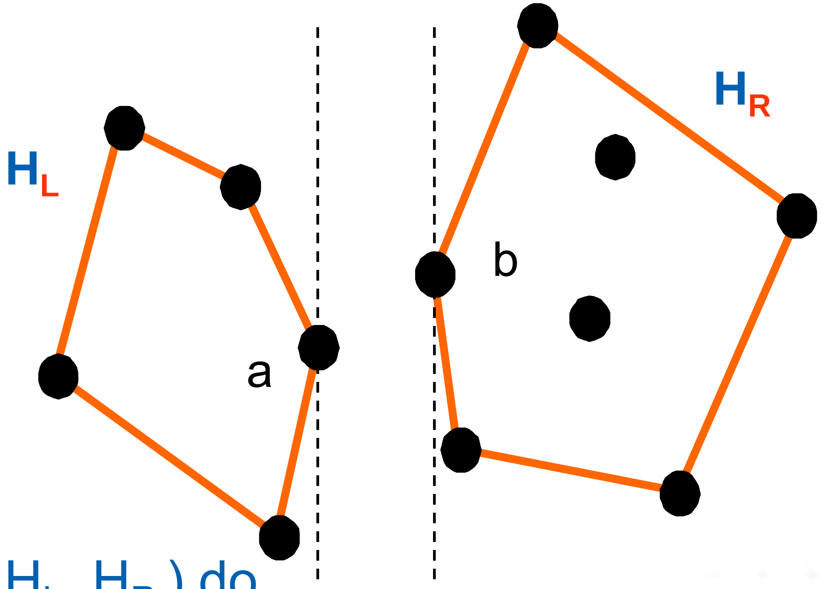


Convex hull by D&C



```
ConvexHullD&C( points P )
Input:    points p
Output:   CCW points on the convex hull
Sort points P according to x
return hull( P )

hull( points P )
    if |P| ≤ 3 then
            compute CH by brute force,
            return
    Partition P into two sets L and R (with lower & higher coords x)
    Recursively compute $H_L$ = hull(L), $H_R$ = hull(R)
    H = Merge hulls($H_L$, $H_R$) by computing
        Upper_tangent( $H_L$, $H_R$) // find nearest points, $H_L$ CCW, $H_R$ CW
        Lower_tangent( $H_L$, $H_R$) // ($H_L$ CW, $H_R$ CCW)
        discard points between these two tangents
    return H
```

Search for upper tangent (lower is symmetrical)



Upper_tangent(H_L, H_R)
Input: two non-overlapping CH's
Output: upper tangent *ab*

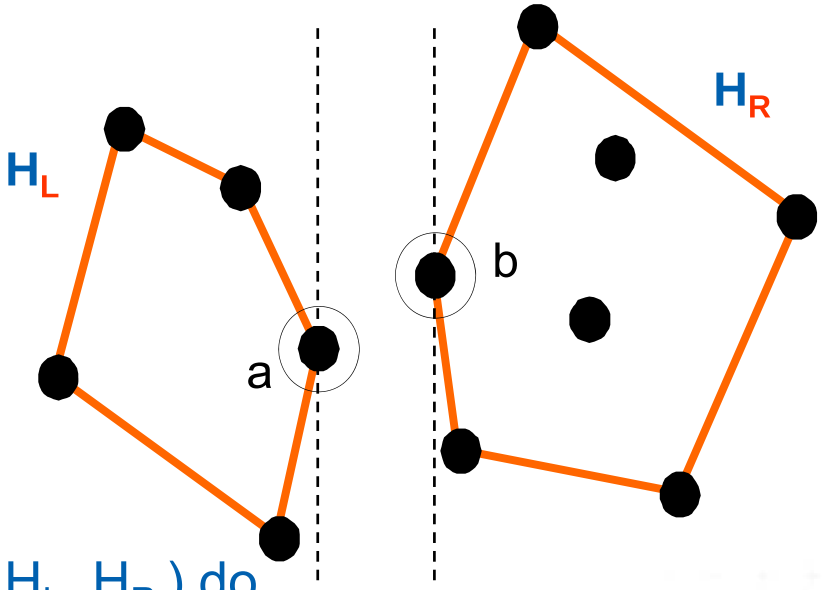
```
1. a = rightmost $H_L$
2. b = leftmost $H_R$
3. while( ab is not the upper tangent for $H_L$, $H_R$ ) do
4.     while( ab is not the upper tangent for $H_L$)  a = a.succ    // move CCW
5.     while( ab is not the upper tangent for $H_R$)  b = b.pred    // move  CW
6. Return ab
```

Where: (ab is not the upper tangent for H_L) => orient$(a, b, a.succ) \geq 0$
which means *a.succ* is left from line *ab*

$m = |H_L| + |H_R| \leq |L| + |R|$ => Upper Tangent: $O(m) = O(n)$

Search for upper tangent (lower is symmetrical)

Upper_tangent(H_L, H_R)
Input: two non-overlapping CH's
Output: upper tangent *ab*



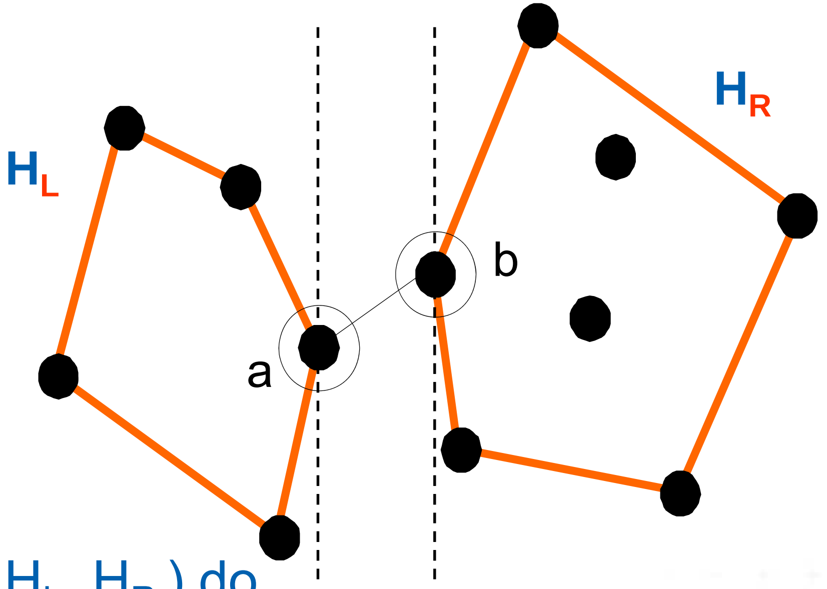
```
1.  a = rightmost $H_L$
2.  b = leftmost $H_R$
3.  while( ab is not the upper tangent for $H_L$, $H_R$ ) do
4.      while( ab is not the upper tangent for $H_L$)  a = a.succ   // move CCW
5.      while( ab is not the upper tangent for $H_R$)  b = b.pred   // move  CW
6.  Return ab
```

Where: (ab is not the upper tangent for H_L) => orient*(a, b, a.succ)* ≥ 0
which means *a.succ* is left from line *ab*

$$m = |H_L| + |H_R| \leq |L| + |R| \quad \Rightarrow \text{Upper Tangent: } O(m) = O(n)$$

Search for upper tangent (lower is symmetrical)

Upper_tangent(H_L, H_R)
Input: two non-overlapping CH's
Output: upper tangent *ab*



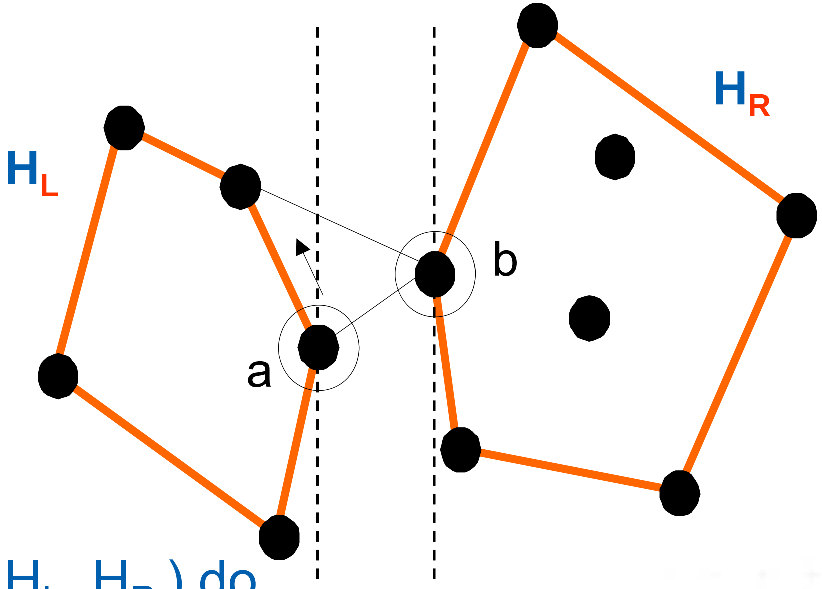
```
1.  a = rightmost $H_L$
2.  b = leftmost $H_R$

3.  while( ab is not the upper tangent for $H_L$, $H_R$ ) do
4.      while( ab is not the upper tangent for $H_L$)  a = a.succ    // move CCW
5.      while( ab is not the upper tangent for $H_R$)  b = b.pred    // move  CW
6.  Return ab
```

Where: (ab is not the upper tangent for H_L) => orient(*a, b, a.succ*) ≥ 0
which means *a.succ* is left from line *ab*

$m = |H_L| + |H_R| \leq |L| + |R|$ => Upper Tangent: $O(m) = O(n)$

Search for upper tangent (lower is symmetrical)

Upper_tangent(H_L, H_R)
Input: two non-overlapping CH's
Output: upper tangent *ab*



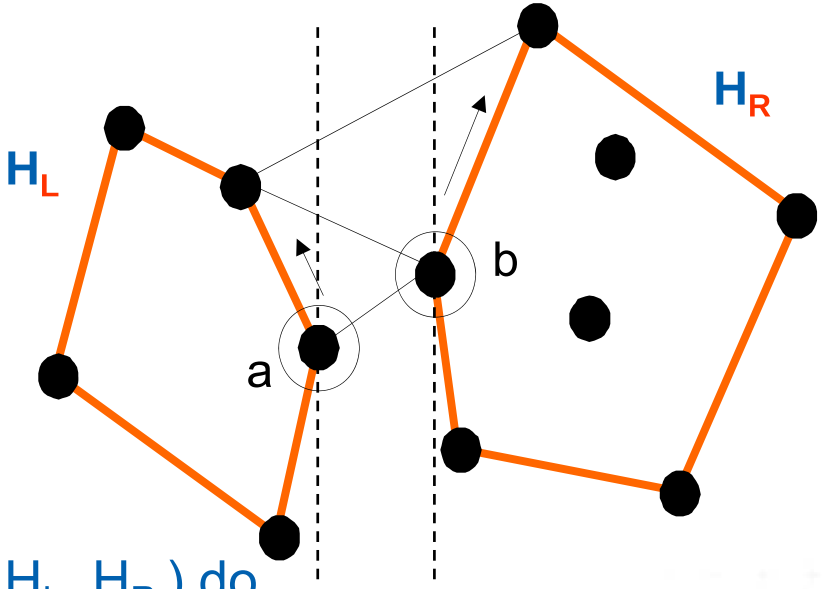
```
1.  a = rightmost $H_L$
2.  b = leftmost $H_R$
3.  while( ab is not the upper tangent for $H_L$, $H_R$ ) do
4.      while( ab is not the upper tangent for $H_L$)  $a = a.succ$    // move CCW
5.      while( ab is not the upper tangent for $H_R$)  $b = b.pred$    // move  CW
6.  Return ab
```

Where: (ab is not the upper tangent for H_L) => orient$(a, b, a.succ) \geq 0$
which means *a.succ* is left from line *ab*

$$m = |H_L| + |H_R| \leq |L| + |R| \quad \Rightarrow \text{Upper Tangent: } O(m) = O(n)$$

Search for upper tangent (lower is symmetrical)

Upper_tangent(H_L, H_R)
Input: two non-overlapping CH's
Output: upper tangent *ab*



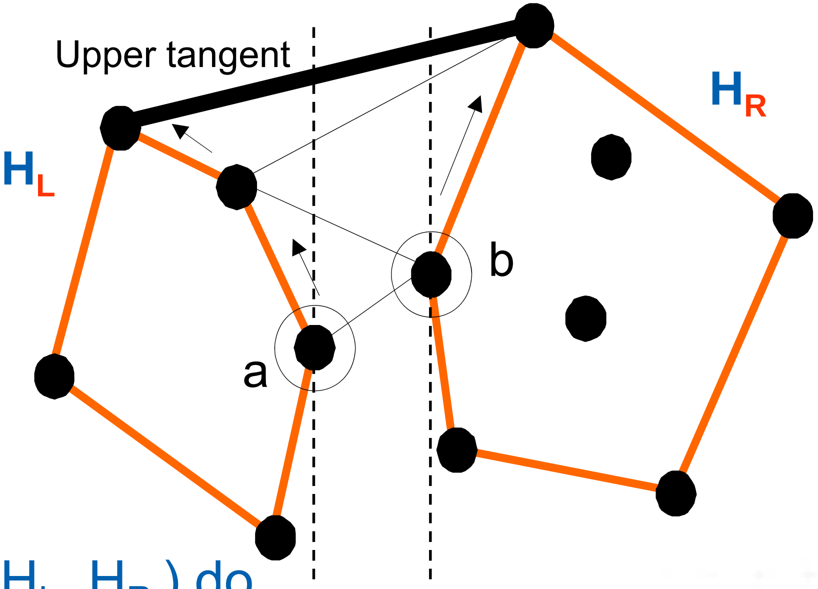
```
1.  a = rightmost $H_L$
2.  b = leftmost $H_R$

3.  while( ab is not the upper tangent for $H_L$, $H_R$ ) do
4.       while( ab is not the upper tangent for $H_L$)  a = a.succ    // move CCW
5.       while( ab is not the upper tangent for $H_R$)  b = b.pred    // move  CW
6.  Return ab
```

Where: (ab is not the upper tangent for H_L) => orient*(a, b, a.succ)* ≥ 0
which means *a.succ* is left from line *ab*

$m = |H_L| + |H_R| \leq |L| + |R|$ => Upper Tangent: $O(m) = O(n)$

Search for upper tangent (lower is symmetrical)

Upper_tangent(H_L, H_R)
Input: two non-overlapping CH's
Output: upper tangent *ab*



```
1.  a = rightmost $H_L$
2.  b = leftmost $H_R$

3.  while( ab is not the upper tangent for $H_L$, $H_R$ ) do
4.      while( ab is not the upper tangent for $H_L$)  $a = a.succ$    // move CCW
5.      while( ab is not the upper tangent for $H_R$)  $b = b.pred$    // move  CW
6.  Return $ab$
```

Where: (ab is not the upper tangent for H_L) => orient($a, b, a.succ$) ≥ 0
which means *a.succ* is left from line *ab*

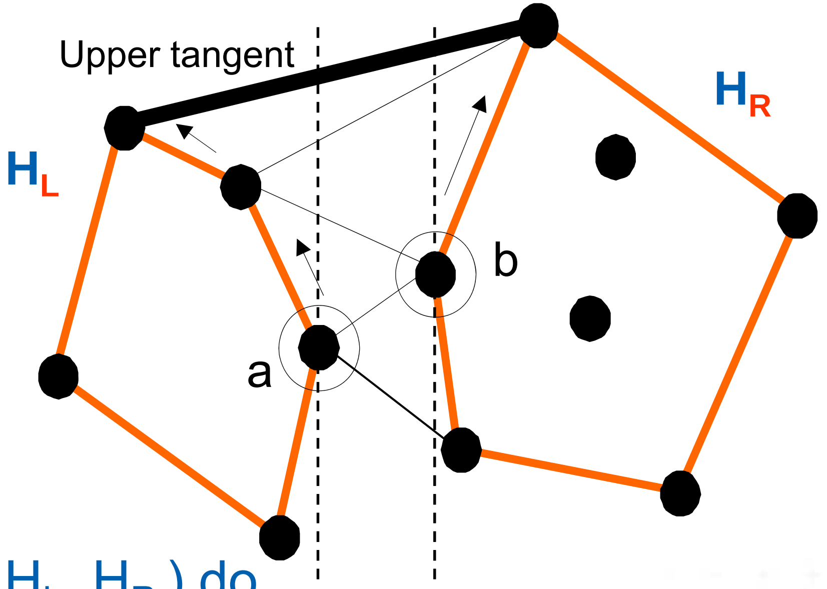
$m = |H_L| + |H_R| \leq |L| + |R|$ => Upper Tangent: $O(m) = O(n)$

Search for upper tangent (lower is symmetrical)

Upper_tangent(H_L, H_R)

Input: two non-overlapping CH's

Output: upper tangent *ab*



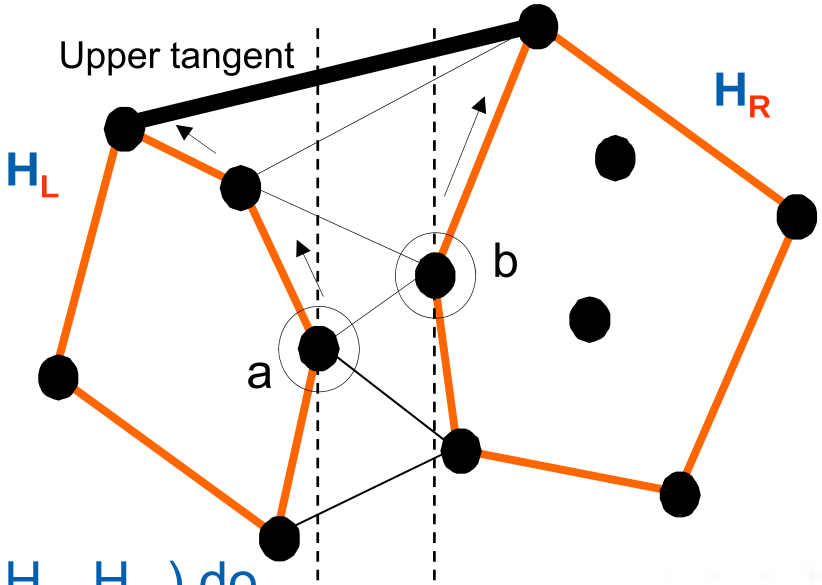
```
1.  a = rightmost $H_L$
2.  b = leftmost $H_R$

3.  while( ab is not the upper tangent for $H_L$, $H_R$ ) do
4.      while( ab is not the upper tangent for $H_L$)  a = a.succ    // move CCW
5.      while( ab is not the upper tangent for $H_R$)  b = b.pred    // move  CW
6.  Return ab
```

Where: (ab is not the upper tangent for H_L) => orient*(a, b, a.succ)* ≥ 0
which means *a.succ* is left from line *ab*

$$m = |H_L| + |H_R| \leq |L| + |R| \quad \Rightarrow \text{Upper Tangent: } O(m) = O(n)$$

Search for upper tangent (lower is symmetrical)

Upper_tangent(H_L, H_R)
Input: two non-overlapping CH's
Output: upper tangent *ab*

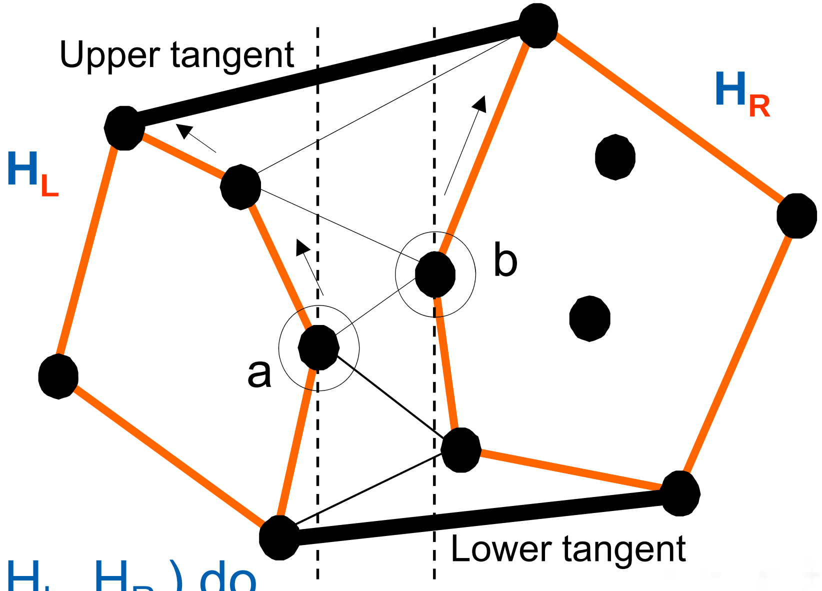


```
a = rightmost $H_L$
b = leftmost $H_R$

while( ab is not the upper tangent for $H_L$, $H_R$ ) do
    while( ab is not the upper tangent for $H_L$)  $a = a.succ$    // move CCW
    while( ab is not the upper tangent for $H_R$)  $b = b.pred$    // move  CW
Return $ab$
```

Where: (ab is not the upper tangent for H_L) => orient$(a, b, a.succ) \geq 0$
which means *a.succ* is left from line *ab*

$$m = |H_L| + |H_R| \leq |L| + |R| \quad \Rightarrow \text{Upper Tangent: } O(m) = O(n)$$

Search for upper tangent (lower is symmetrical)



Upper_tangent(H_L, H_R)
Input: two non-overlapping CH's
Output: upper tangent *ab*

```
1.  a = rightmost $H_L$
2.  b = leftmost $H_R$

3.  while( ab is not the upper tangent for $H_L$, $H_R$ ) do
4.      while( ab is not the upper tangent for $H_L$)  a = a.succ     // move CCW
5.      while( ab is not the upper tangent for $H_R$)  b = b.pred     // move  CW
6.  Return ab
```

Where: (ab is not the upper tangent for H_L) => orient*(a, b, a.succ)* ≥ 0
which means *a.succ* is left from line *ab*

$$m = |H_L| + |H_R| \leq |L| + |R| \quad \Rightarrow \text{Upper Tangent: } O(m) = O(n)$$

Convex hull by D&C complexity

- Initial sort O(*n* log(*n*))
- Function hull()

$$\left.\begin{array}{ll}\text{– Upper and lower tangent} & O(n)\\ \text{– Merge hulls} & O(1)\\ \text{– Discard points between tangents} & O(n)\end{array}\right\} O(n)$$

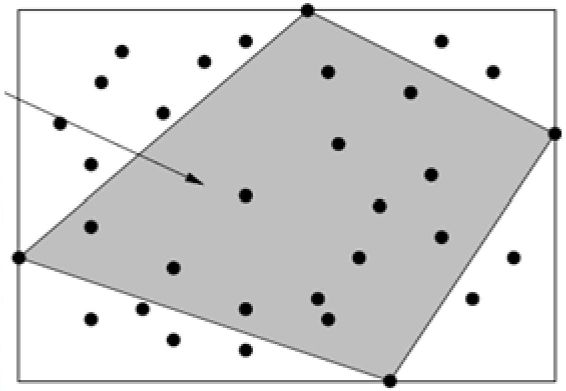
- Overall complexity
  - Recursion

$$T(n) = \begin{cases} 1 & \dots \text{ if } n \le 3 \\ 2T(n/2) + O(n) & \dots \text{ otherwise} \end{cases}$$

  - Overall complexity of CH by D&C: => O(*n* log(*n*))

Quick hull

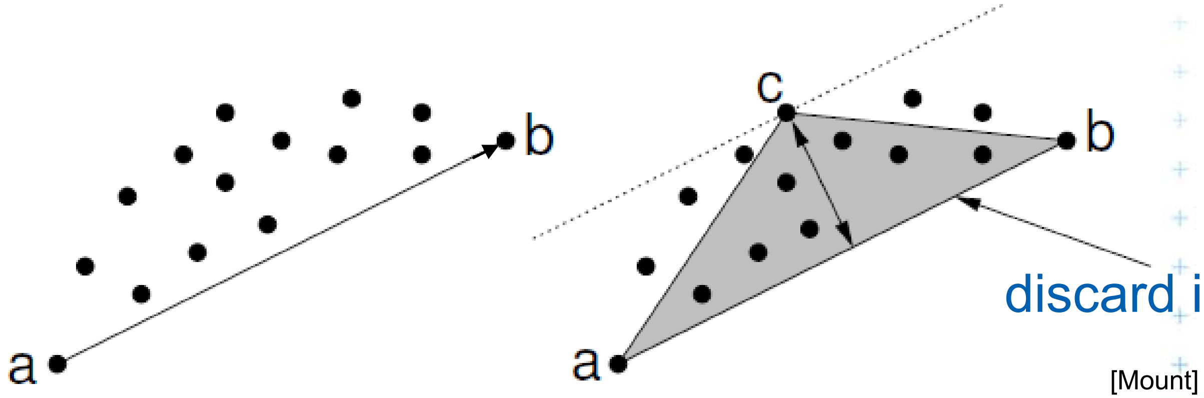
- A variant of Quick Sort
- O(*n* log *n*) expected time, max O(n^2)
- Principle
  - in praxis, most of the points lie in the interior of CH
  - E.g., for uniformly distributed points in unit square, we expect only O(log *n*) points on CH
- Find extreme points (parts of CH) quadrilateral, discard inner points
  - Add 4 edges to temp hull T
  - Process points outside 4 edges



[Mount]

Process each of four groups of points outside

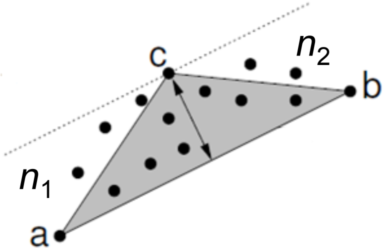
- For points outside *ab* (left from *ab* for clockwise CH)
  - – Find point *c* on the hull – max. perpend. distance to *ab*
  - – Discard points inside triangle *abc* (right from the edges)
  - – Split points into two subsets
    - outside *ac* (left from *ac*) and outside *cb* (left from *cb*)
  - – Process points outside *ac* and *cb* recursively
  - – Replace edge *ab* in *T* by edges *ac* and *cb*



discard inner points

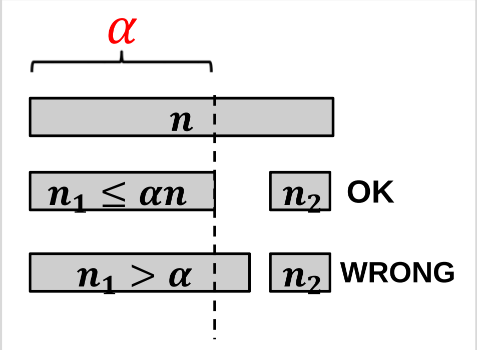
[Mount]

Quick hull complexity



- *n* points remain outside the hull
- *T*(*n*) = running time for such *n* points outside
  - O(*n*) - selection of splitting point *c*
  - O(*n*) - point classification to inside & (n_1+n_2) outside
  - $n_1+n_2 \leq n$
  - The running time is given by recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n_1) + T(n_2) & \text{where } n_1+n_2 \leq n \end{cases}$$



  - If evenly distributed that $\max(n_1, n_2) \leq \alpha n, 0 < \alpha < 1$ then solves as QuickSort to O(*cn* log *n*) where c=f(α) else O(n^2) for unbalanced splits

Jarvis's March – selection by gift wrapping

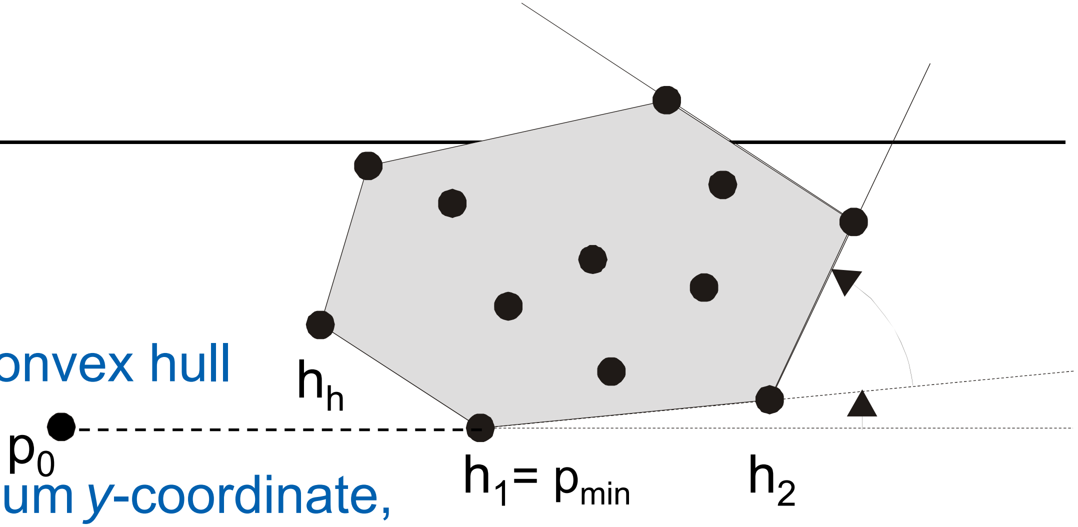
- Variant of O(n^2) selection sort
- Output sensitive algorithm
- O(*nh*) … *h* = *number of points on convex hull*

Jarvis’s March

JarvisCH(points P)

Input: points p

Output: CCW points on the convex hull

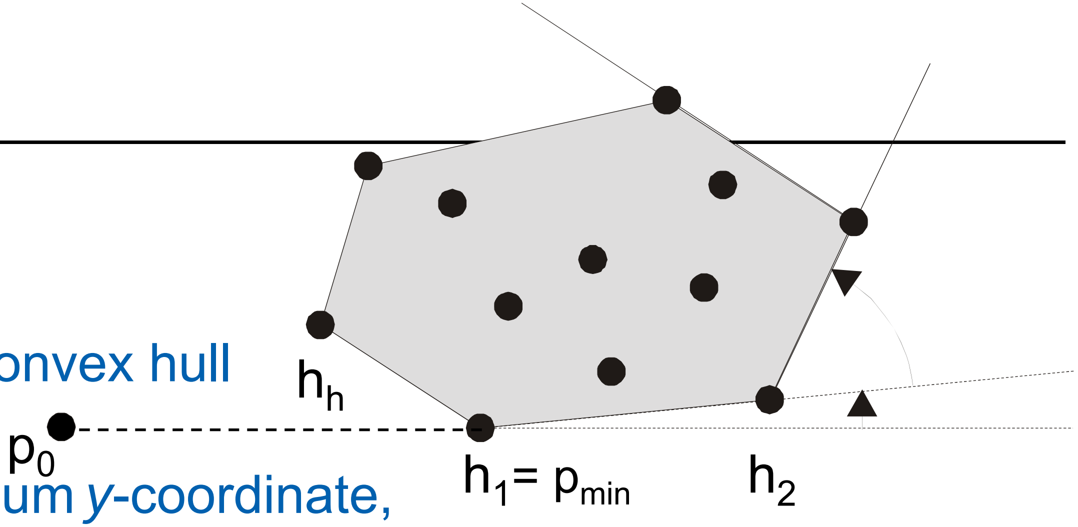


```
Take point $p_{min}$ with minimum $y$-coordinate,
// $p_{min}$ will be the first point in the hull – append it to the hull as $h_1$
Take a horizontal line, i.e., create temporary point $p_0 = (-\infty, h_1.y)$
j = 1
repeat
    Rotate the line around $h_j$ until it bounces to the nearest point q = $p_q$
    // compute the smallest angle by the “smallest orient($h_{j-1}$ , $h_j$, $q$)”
    j++
    append the bounced nearest point q to the hull as next $h_j$
until (q $\neq$ $p_{min}$)
```

Complexity: $O(n) + O(n) * h \Rightarrow O(h*n)$

good for low number of points on convex hull

Jarvis's March



JarvisCH(points P)
Input: points p
Output: CCW points on the convex hull

```
1. Take point $p_{min}$ with minimum $y$-coordinate,
   // $p_{min}$ will be the first point in the hull – append it to the hull as $h_1$
2. Take a horizontal line, i.e., create temporary point $p_0 = (-\infty, h_1.y)$
3. j = 1
4. repeat
5.     Rotate the line around $h_j$ until it bounces to the nearest point q = $p_q$
       // compute the smallest angle by the "smallest orient($h_{j-1}$ , $h_j$, $q$)"
6.     j++
       append the bounced nearest point q to the hull as next $h_j$
7. until ($q \neq p_{min}$)
```

Output sensitive algorithm

Complexity: $O(n) + O(n) * h \Rightarrow O(h*n)$

good for low number of points on convex hull

Output sensitive algorithm

- Worst case complexity analysis analyzes the worst case data
  - Presumes, that all (const fraction of) points lie **on** the CH
  - The points are ordered along CH

    => We need sorting => $\Omega(n \log n)$ of CH algorithm
- Such assumption is rare
  - usually only much less of points are on CH
- Output sensitive algorithms
  - Depend on: input size *n* and the size of the output *h*
  - Are more efficient for small output sizes
  - Reasonable time for CH is $O(n \log h)$, *h = Number of points on the CH*

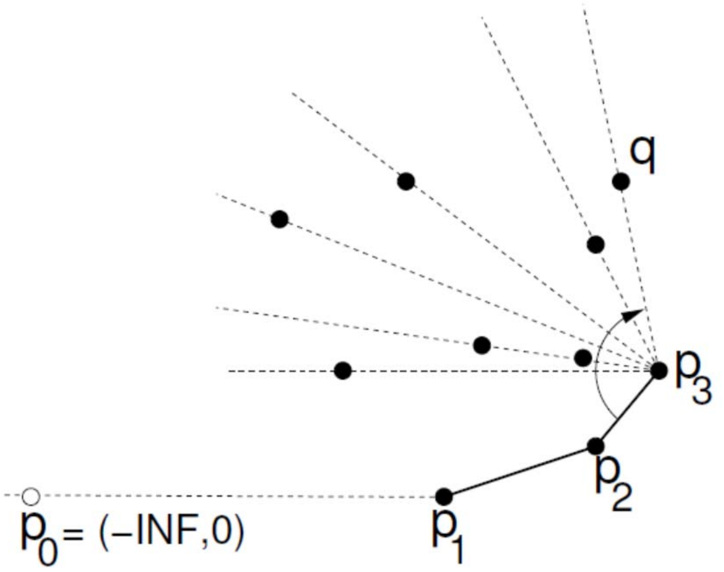
Chan's algorithm

- Cleverly combines Graham's scan and Jarvis's march algorithms
- Goal is O(n log h) running time
  - We cannot afford sorting of all points - $\Omega(n \log n)$

  => Idea: work on parts, limit the part sizes to polynomial h^c the complexity does not change => log h^c = log h
  - h is unknown – we get the estimation later
  - Use estimation m, better not too high => $h \leq m \leq h^2$
- 1. Partition points P into r-groups of size m, $r = n/m$
  - Each group take O(m log m) time - sort + Graham
  - r-groups take O(r m log m) = O(n log m) - Jarvis

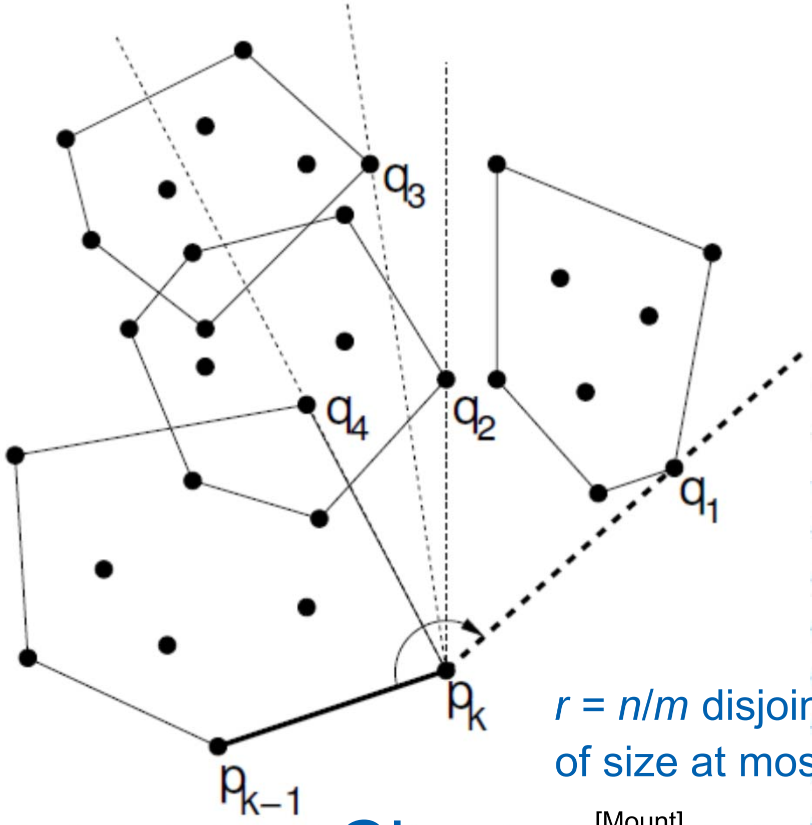
Merging of *m* parts in Chan’s algorithm

- 2. Merge *r*-group CHs as “fat points”
  - Tangents to convex *m*-gon can be found in O(log *m*) by binary search



Jarvis

[Mount]



r = *n*/*m* disjoint subsets of size at most *m*

Chan

[Mount]

Chan’s algorithm complexity

- *h* points on the final convex hull

  => at most *h* steps in the Jarvis march algorithm

  – each step computes *r*-tangents, O(log *m*) each

  – merging together O(*hr* log *m*)

r-groups of size *m*, $r = n/m$

- Complete algorithm O(*n* log *h*)

  – Graham’s scan on partitions O(*r* .*m* log *m*)=O(*n* log *m*)

  – Jarvis Merging: O(*hr* log *m*) = O(*h n*/*m* log *m*), …4a)

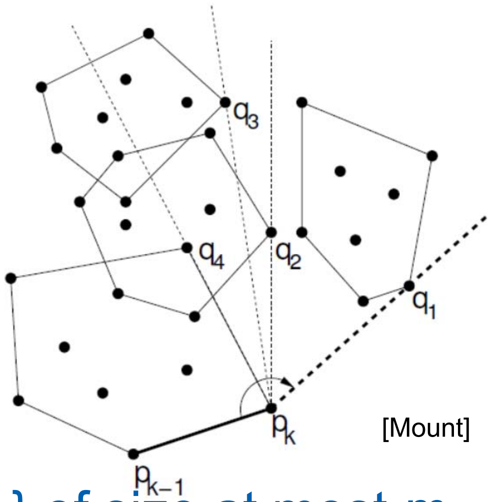
  $h \le m \le h^2$ = O(*n* log *m*)

  – Altogether $\underline{\underline{O(n \log m)}}$

  – How to guess *m*? *Wait!*

  1) use m as an estimation of h 2) if it fails, increase m

Chan's algorithm for known *m*



[Mount]

PartialHull(*P*, *m*)
Input: points P
Output: group of size *m*

```
1. Partition $P$ into $r = \lceil n/m \rceil$ disjoint subsets $\{p_1, p_2, \dots, p_r\}$ of size at most $m$
2. for $i=1$ to $r$ do
     a) Convex hull by GrahamsScan($P_i$), store vertices in ordered array
3. let $p_1$ = the bottom most point of P and $p_0 = (-\infty, p_1.y)$
4. for $k = 1$ to $m$ do      // compute merged hull points
     a) for $i = 1$ to $r$ do  // angle to all $r$ subsets => points $q_i$
            Compute the point $q_i \in P$ that maximizes the angle $\angle\, p_{k-1}, p_k, q_i$
     b) let $p_{k+1}$ be the point $q \in \{q_1, q_2, \dots, q_r\}$ that maximizes $\angle\, p_{k-1}, p_k, q$
        ($p_{k+1}$ is the new point in CH)
     c) if $p_{k+1} = p_1$ then return $\{p_1, p_2, \dots, p_k\}$
5. return "Fail, $m$ was too small"
```

O(log *m*)

Jarvis

Chan’s algorithm – estimation of *m*

```
ChansHull
Input:    points P
Output:   convex hull $p_1 \ldots p_k$

1.  for $t$ = 1, 2, … , $\lceil \lg \lg h \rceil$ do {
      a) let $m = \min(2^{2^\wedge t}, n)$
      b) $L$ = PartialHull( $P$, $m$)
      c) if $L \neq$ “Fail, $m$ was too small” then return $L$
    }
```

Sequence of choices of *m* are { 4, 16, 256,…, $2^{2^\wedge t}$,…, *n* } … squares

Example: for h = 23 points on convex hull of n = 57 points, the algorithm will try this sequence of choices of *m* { 4, 16, 57 }

1. 4 and 16 will fail
2. 256 will be replaced by *n=57*

Complexity of Chan’s Convex Hull?

- The worst case: Compute all iterations
- t^{th} iteration takes O(n log $2^{2^\wedge t}$) = O($n\ 2^t$)
- Algorithm stops when $2^{2^\wedge t} \geq h$ => $t = \lceil \lg \lg h \rceil$
- All $t = \lceil \lg \lg h \rceil$ iterations take:

Using the fact that $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$

$$\sum_{t=1}^{\lg\lg h} n2^t = n\sum_{t=1}^{\lg\lg h} 2^t \leq n2^{1+\lg\lg h} = 2n\lg h = O(n\log h)$$

2x more work in the worst case

Complexity of Chan's Convex Hull?

- The worst case: Compute all iterations
- tth iteration takes O(n log $2^{2\wedge t}$) = O(n 2^t)
- Algorithm stops when $2^{2\wedge t} \geq h$ => $t = \lceil \lg \lg h \rceil$
- All $t = \lceil \lg \lg h \rceil$ iterations take:

Using the fact that $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$

one iteration

$$\sum_{t=1}^{\lg \lg h} n2^t = n \sum_{t=1}^{\lg \lg h} 2^t \leq n2^{1+\lg \lg h} = 2n \lg h = O(n \log h)$$

2x more work in the worst case

Complexity of Chan’s Convex Hull?

- The worst case: Compute all iterations
- t^th iteration takes $O(n \log 2^{2^{\wedge}t}) = O(n\, 2^t)$
- Algorithm stops when $2^{2^{\wedge}t} \geq h \Rightarrow t = \lceil \lg \lg h \rceil$
- All $t = \lceil \lg \lg h \rceil$ iterations take:

Using the fact that $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$

one iteration

t iterations

$$\sum_{t=1}^{\lg \lg h} n2^t = n \sum_{t=1}^{\lg \lg h} 2^t \leq n2^{1+\lg \lg h} = 2n \lg h = O(n \log h)$$

2x more work in the worst case

Conclusion in 2D

Graham's scan:	O(n log n), O(n) for sorted pts
Divide & Conquer:	O(n log n)
Quick hull:	O(n log n), max O(n^2) ~ distrib.
Jarvis's march:	O(hn), max O(n^2) ~ pts on CH
Chan's alg.:	O(n log h) ~ pts on CH

asymptotically optimal

but

constants are too high to be useful

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, Chapter 5, http://www.cs.uu.nl/geobook/
- [Mount] David Mount, - CMSC 754: Computational Geometry, Lecture Notes for Spring 2007, University of Maryland, Lectures 3 and 4. http://www.cs.umd.edu/class/spring2007/cmsc754/lectures.shtml
- [Chan] Timothy M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions., *Discrete and Computational Geometry*, 16, 1996, 361-368.

  http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.389

DCGI

DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

CONVEX HULL IN 3 DIMENSIONS

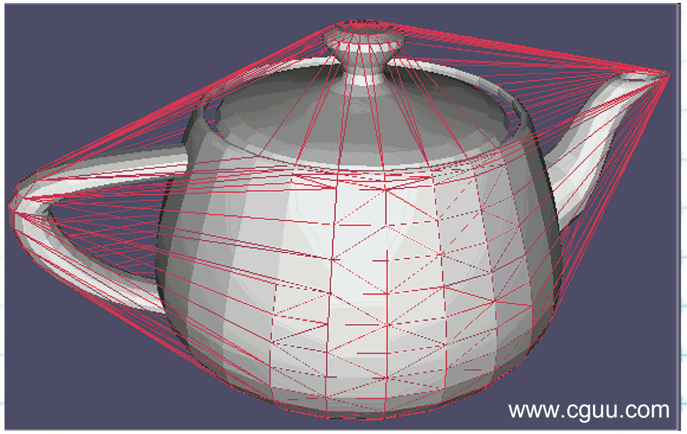
PETR FELKEL

FEL CTU PRAGUE

Version from 1.11.2018

Talk overview

- Upper bounds for convex hull in 2D and 3D
- Other criteria for CH algorithm classification
- Recapitulation of CH algorithms
- Terminology refresh
- Convex hull in 3D
  - Terminology
  - Algorithms
    - Gift wrapping
    - D&C Merge
    - Randomized Incremental


www.cguu.com

Upper bounds for Convex hull algorithms

- O(n) for sorted points and for simple polygon
- O($n \log n$) in E^2, E^3 with sorting
  – insensitive about output
- O($n\ h$), O($n \log h$), h is number of CH facets
  – output sensitive
  – O(n^2) *or* O($n \log n$) for $n \sim h$
- O(log n) for new point insertion in realtime algs.
  => O($n \log n$) for n-points
  O(log n) search where to insert

Other criteria for CH algorithm classification

- Optimality – depends on data order (or distribution)

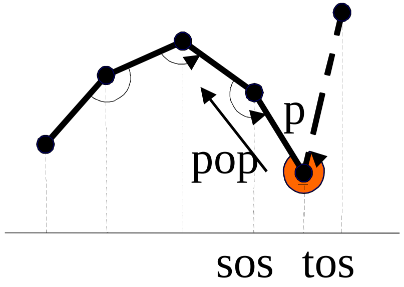
  In the worst case x In the expected case

- Output sensitivity – depends on the result ~ O(f(h))
- Extendable to higher dimensions?
- Off-line versus on-line
  - Off-line – all points available, preprocessing for search speedup
  - On-line – stream of points, new point p_i on demand, just one new point at a time, CH valid for $\{p_1, p_2, \ldots, p_i\}$
  - Real-time – points come as they "want"

    (come not faster than optimal constant O(log n) inter-arrival delay)
- Parallelizable x serial
- Dynamic – points can be deleted
- Deterministic x approximate (lecture 13)

Graham scan

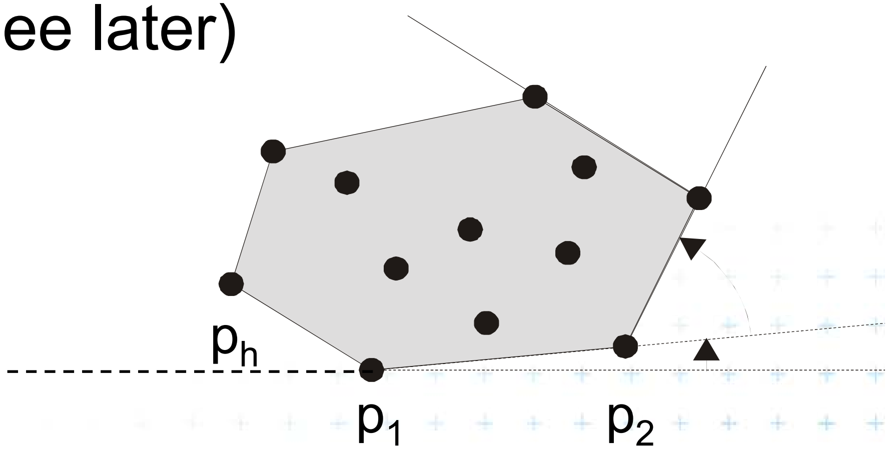
- O(*n* log *n*) time and O(*n*) space is
  - optimal in the worst case
  - not optimal in average case (not output sensitive)
  - only 2D
  - off-line
  - serial (not parallel)
  - not dynamic (no deleted points)



O(*n*) for polygon (discussed in seminar)

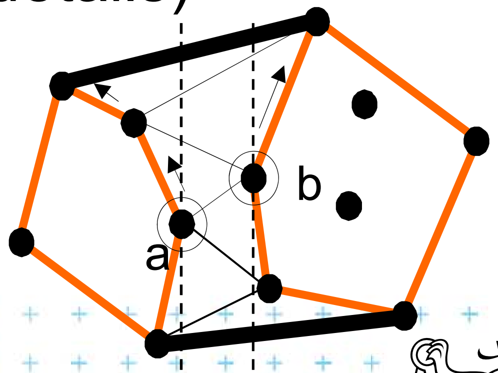
Jarvis March – Gift wrapping

- *O*(*hn*) time and O(*n*) space is
  - not optimal in worst case O(n^2)
  - may be optimal if *h* << *n* (output sensitive)
  - 3D or higher dimensions (see later)
  - off-line
  - serial (not parallel)
  - not dynamic



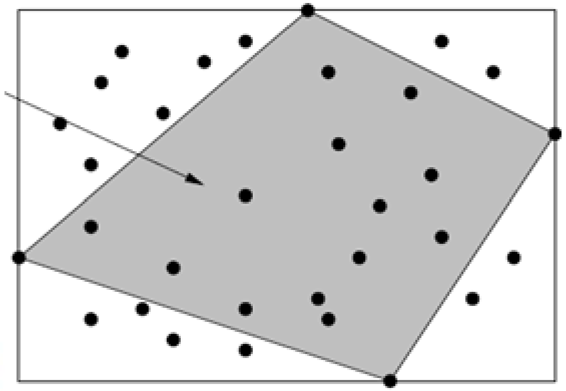
Divide & Conquer

- O(*n* log *n*) time and O(*n*) space is
  - optimal in worst case (in 2D or 3D)
  - not optimal in average case (not output sensitive)
  - 2D or 3D (circular ordering), in higher dims not optimal
  - off-line
  - Version with sorting (the presented one) – serial
  - Parallel for overlapping merged hulls
    (see Chapter 3.3.5 in Preparata for details)
  - not dynamic



Quick hull

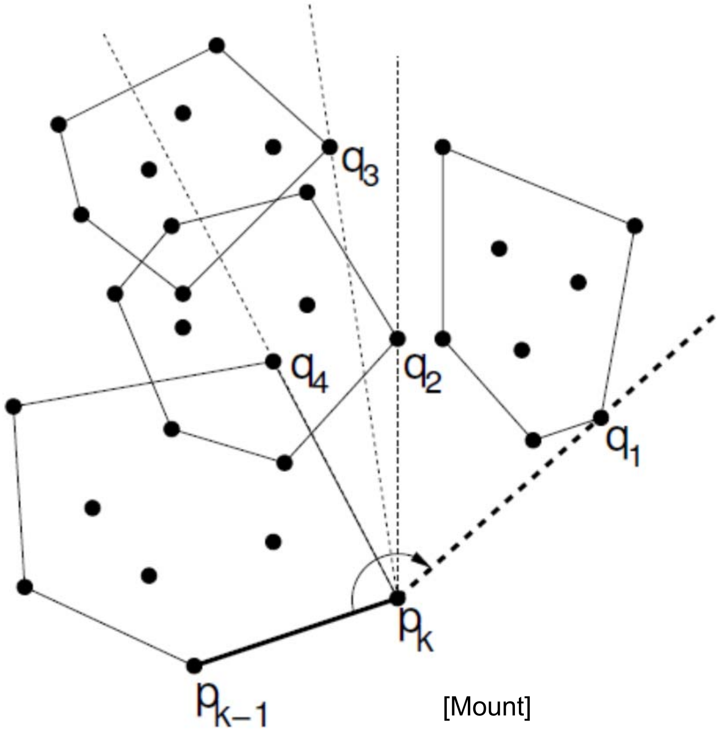
- O(n log n) expected time, O(n^2) the worst case and O(n) space *in 2D* is
  - not optimal in worst case O(n^2)
  - optimal if uniform distribution then $h << n$ (output sensitive)
  - 2D, or higher dimensions [see http://www.qhull.org/]
  - off-line
  - parallelizable
  - not dynamic



[Mount]

Chan

- *O(n* log *h*) time and O(*n*) space is
  - optimal for *h* points on convex hull (output sensitive)
  - 2D and 3D --- gift wrapping
  - off-line
  - Serial (not parallel)
  - not dynamic



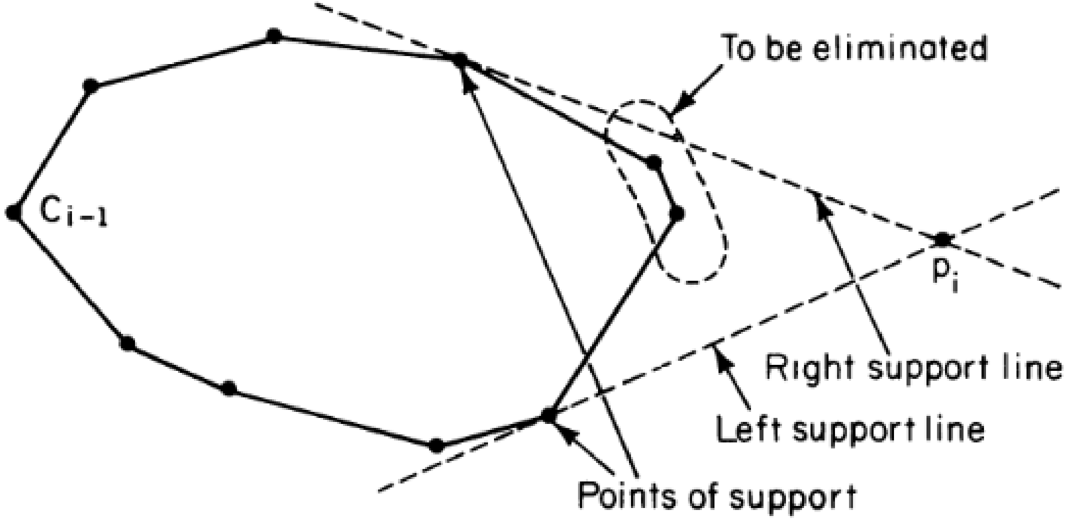
[Mount]

On-line algorithms

- Preparata’s on-line algorithm
- Overmars and van Leeuven

Preparata’s 2D on-line algorithm

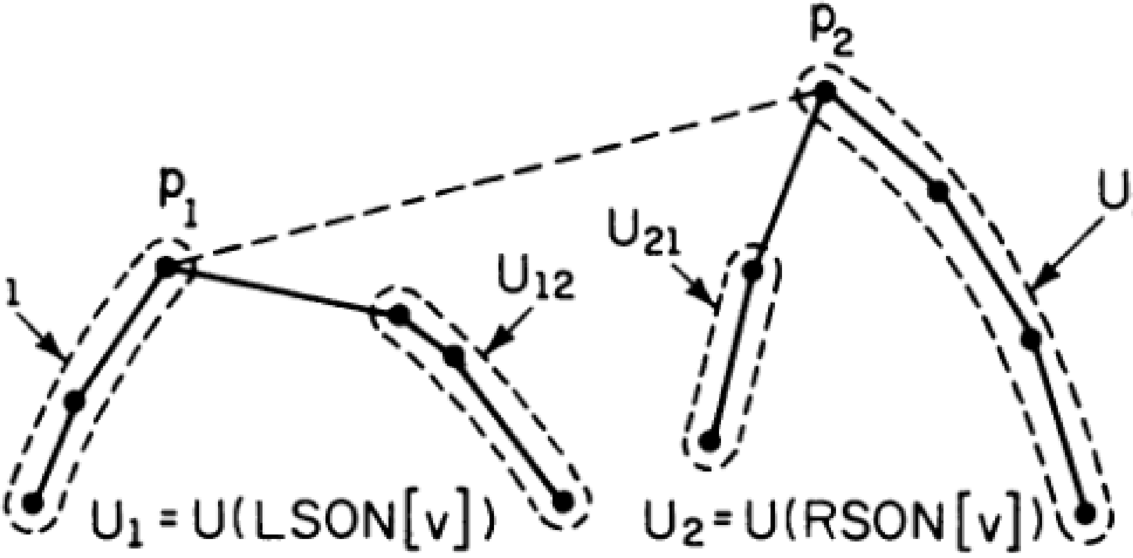
- New point *p* is tested
  - Inside –> ignored
  - Outside –> added to hull
    - Find left and right supporting lines (touch at supporting points)
    - Remove points between supporting points
    - Add *p* to CH between supporting lines



[Preparata]

Overmars and van Leeuven

- Allow dynamic 2D CH
  (on-line insert & delete)
- Manage special tree with all intermediate CHs
- Will be discussed on seminar [7]



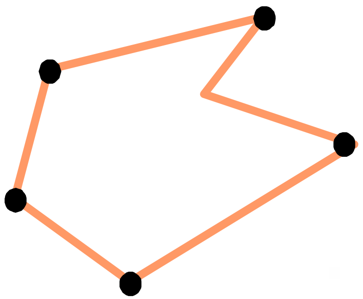
[Preparata]

Convex hull in 3D

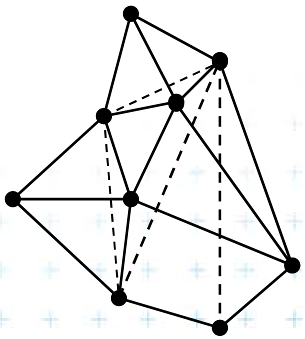
- Terminology
- Algorithms
  1. Gift wrapping
  2. D&C Merge
  3. Randomized Incremental
  4. Quick hull … minule

Terminology

- Polytope (d-polytope)
  = a geometric object with "flat" sides E^d
  (may be or may not be convex)
- Flat sides mean that
  the sides of a (*k*)-polytope
  consist of (*k-1)*-polytopes that
  may have (*k*-2)-polytopes in common.



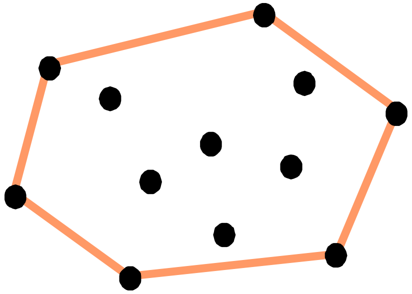
2-polytop
= polygon



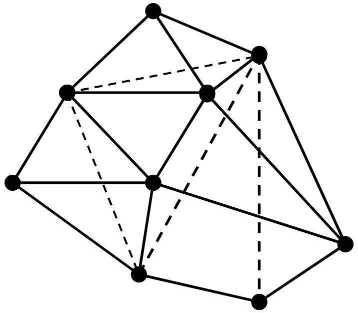
3-polytop
= polyhedron

Terminology

- Convex Polytope (convex d-polytope)
  = convex hull of finite set of points in E^d



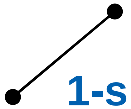
convex
2-polytop



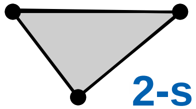
convex
3-polytop

- Simplex (k-simplex, d-simplex)
  = CH of *k* + 1 *affine independent points*

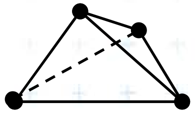
(vectors $u_k - u_0$ are linearly independent)



1-simplex



2-simplex



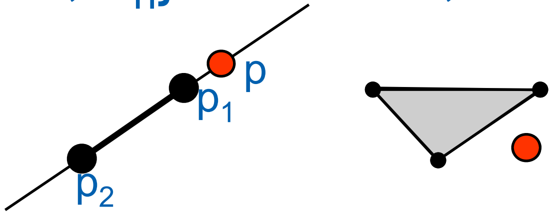
3-simplex

= "Special" Convex Polytope with all the points on the CH

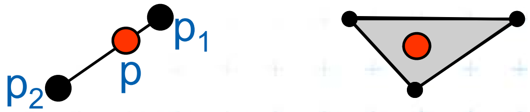
Terminology (2)

- Affine combination
  = linear combination of the points $\{p_1, p_2, \ldots, p_n\}$
  whose coefficients $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ sum to 1, and $\lambda_i \in R$

$$\sum_{i=1}^{n} \lambda_i p_i$$



- Affine independent points
  = no one point can be expressed as affine combination of the others



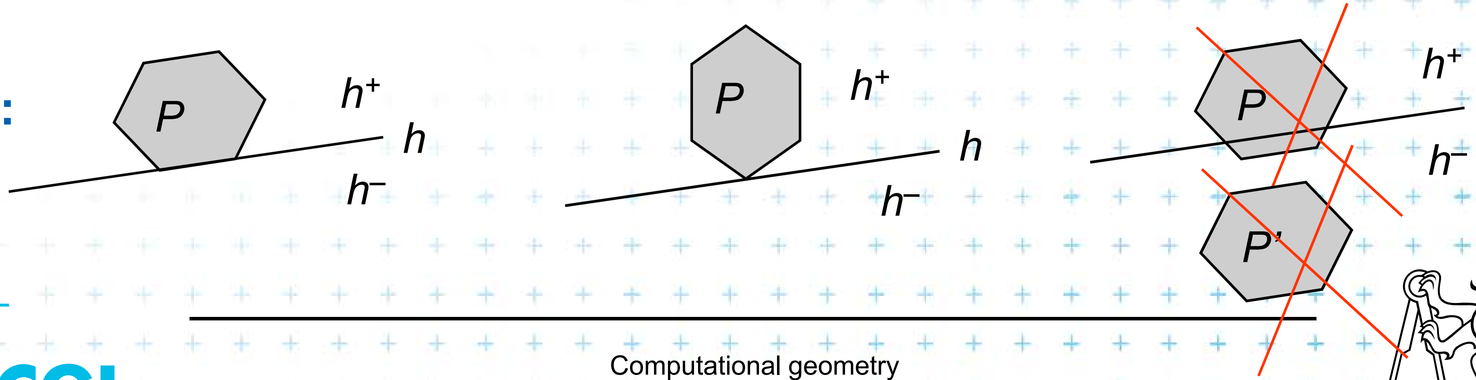
- Convex combination
  = linear combination of the points $\{p_1, p_2, \ldots, p_n\}$
  whose coefficients $\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ sum to 1, and $\lambda_i \in R^+_0$
  (i.e., $\forall i \in \{1,\ldots,n\}$, $\lambda_i \geq 0$)

$$\Rightarrow \lambda_i \in \langle 0, 1 \rangle$$

Terminology (3)

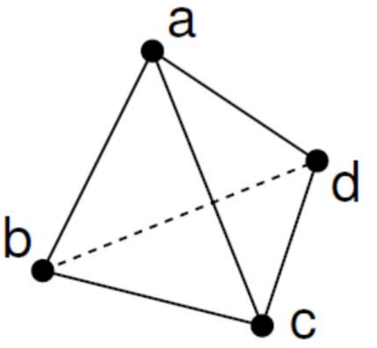
- Any (d-1)-dimensional hyperplane *h* divides the space into (open) halfspaces h^+ *and* h^-, *so that* $E^n = h^+ \cup h \cup h^-$
- *Def:* $\overline{h^+} = h^+ \cup h$, $\overline{h^-} = h^- \cup h$ (closed halfspaces)
- Hyperplane supports a convex polytope *P* (Supporting hyperplane – *opěrná nadrovina*)
  - if $h \cap P$ is not empty and
  - if *P* is entirely contained within either $\overline{h^+}$ or $\overline{h^-}$

In 2D:



Faces and facets

- Face of the convex polytope
  = Intersection of convex polytope *P* with a supporting hyperplane *h*
  - Faces are convex polytopes of dimension *d* ranging from 0 to d – 1
  - 0-face = vertex
  - 1-face = edge
  - (d – 1)-face = facet



Proper faces:
Vertices: a,b,c,d
Edges: ab, ac, ad, bc, bd, cd
Facets: abc, abd, acd, bcd

In 3D we often say *face*, but more precisely a *facet*

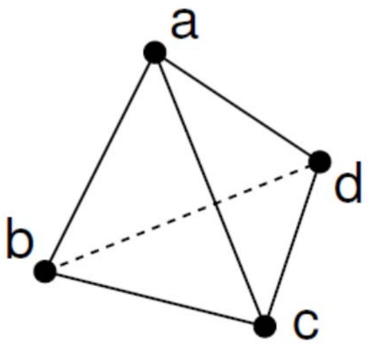
(In 3D a 2-face = facet)
(In 2D a 1-face = facet)

Proper faces

- Proper faces

  = Faces of dimension *d* ranging from 0 to *d* – 1
- Improper faces

  = proper faces + two additional faces:
  - {} = Empty set = face of dimension -1
  - Entire convex polytope = face of dimension *d*



Improper faces in 3D:

Empty set {}

Vertices: a,b,c,d

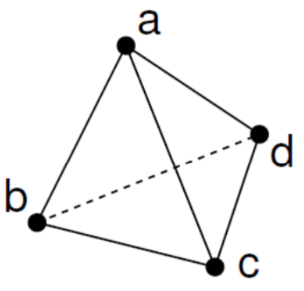
Edges: ab, ac, ad, bc, bd, cd

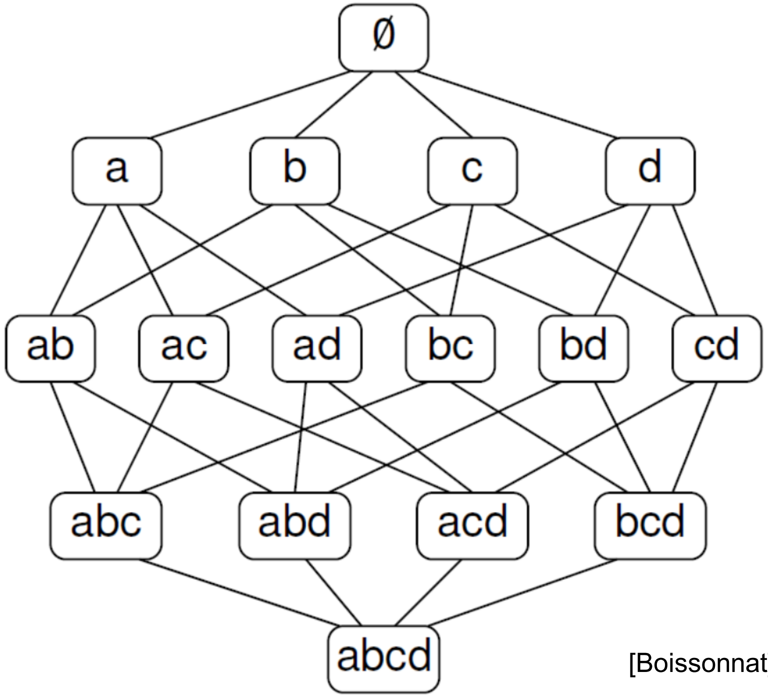
Facets: abc, abd, bcd

Entire polytope: abcd

Incident graph

- Stores topology of the polytope
- Ex: 3-simplex:





Dimension

-1

0

1

2

3

[Boissonnat]

- d-simplex is a very regular face structure:
  - 1-face for each pair of vertices
  - 2-face for each triple of vertices

Facts about polytopes

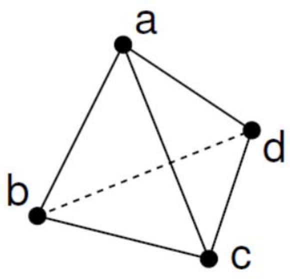
- Boundary o polytope is *union of its proper faces*
- Polytope has *finite number of faces (next slide)*. Each face is a polytope
- Convex polytope is *convex hull of its vertices (the def),* its bounded
- Convex polytope is the *intersection of finite number of closed halfspaces* $\overline{h^+}$ (conversely not: intersection of closed halfspaces may be unbounded => called *unbounded polytope*)

Number of faces on a d-simplex

- Number of *j*-dimensional faces on a *d*-simplex

$$\binom{d+1}{j+1} = \frac{(d+1)!}{(j+1)!(d-j)!}$$

- Ex.: Tetrahedron = 3-simplex:
  - facets (2-dim. faces) $\binom{3+1}{2+1} = \frac{4!}{3!1!} = 4$
  - edges (1-dim. faces) $\binom{3+1}{1+1} = \frac{4!}{2!2!} = 6$
  - vertices (0-dim faces) $\binom{3+1}{0+1} = \frac{4!}{1!3!} = 4$



Complexity of 3D convex hull is O(n)

- 3-polytope - has polygonal faces
- convex 3-polytope (CH of a point set in 3D)
- simplical 3-polytope
  - has triangular faces (=> more edges and vertices)
- simplical convex 3-polytope with all *n* points on CH
  - the worst case complexity
  - => maximum # of edges and vertices
  - has triangular facets, each generates 3 edges, shared by 2 triangles => 3F = 2E   2-manifold

F = 2V – 4   => F ≤ 2V – 4   F = O(n)

E = 3V – 6   => E ≤ 3V – 6   E = O(n)

Complexity of 3D convex hull is O(n)

- The worst case complexity → if all *n* points on CH

=> use simplical convex 3-polytop for complexity derivation

1. has all points on its surface – on the Convex Hull
2. has triangular facets, each generates 3 edges, shared by 2 triangles => $3F = 2E$

$F = 2E / 3$

2-manifold

- $V - E + \boxed{F} = 2$ … Euler formula for V = n points

$V - E + 2E/3 = 2$

$V - 2 = E / 3$

$E = 3V - 6, \quad V = n$

$E = O(n)$

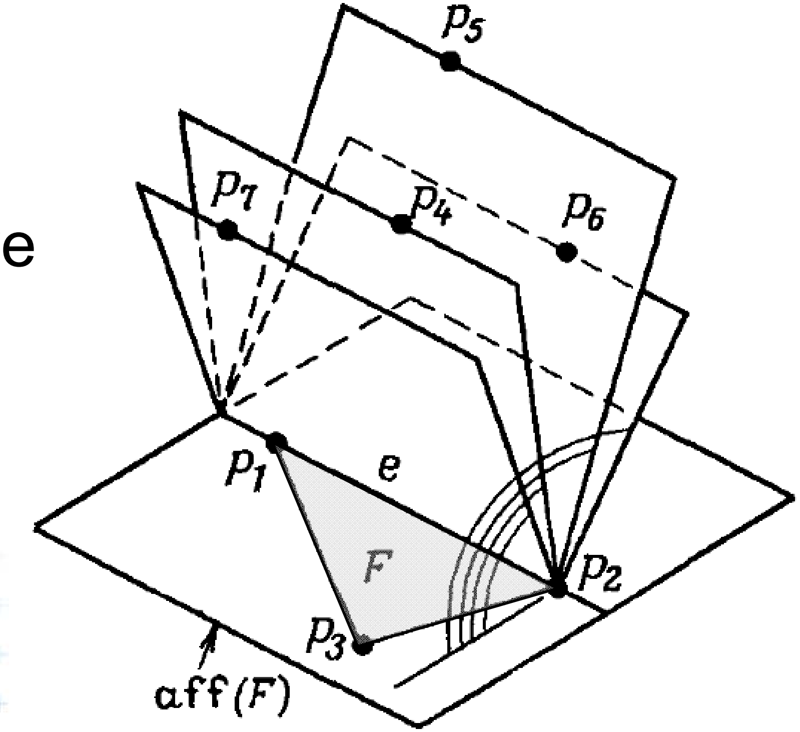
$F = 2\boxed{E} / 3$

$F = 2V - 4$

$F = O(n)$

1. Gift wrapping in higher dimensions

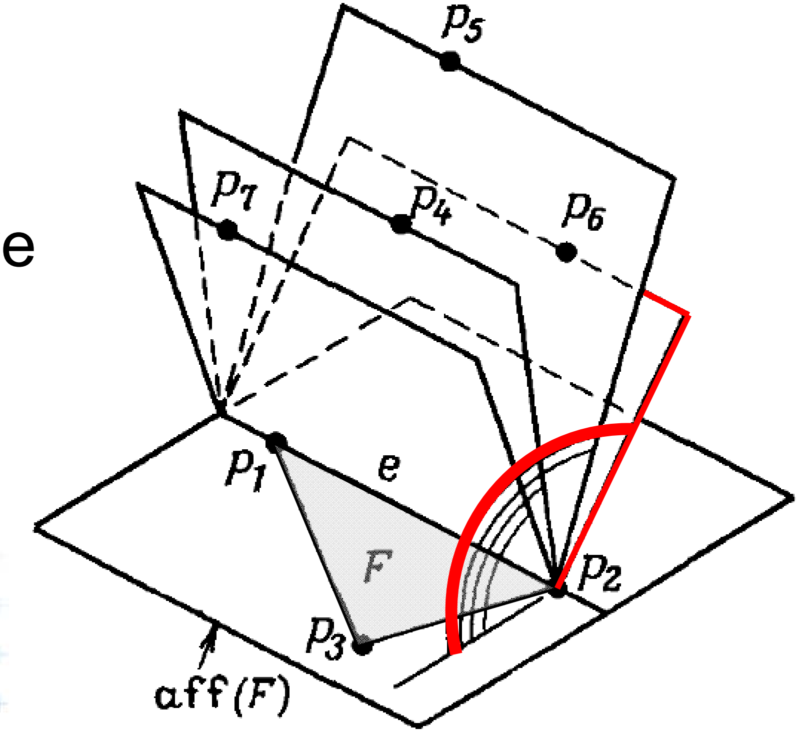
- First known algorithm for n-dimensions (1970)
- Direct extension of 2D alg.
- Complexity O(nF)
  - F is number of CH facets
  - Algorithm is output sensitive
  - Details on seminar, assignment [10]



[Preparata]

1. Gift wrapping in higher dimensions

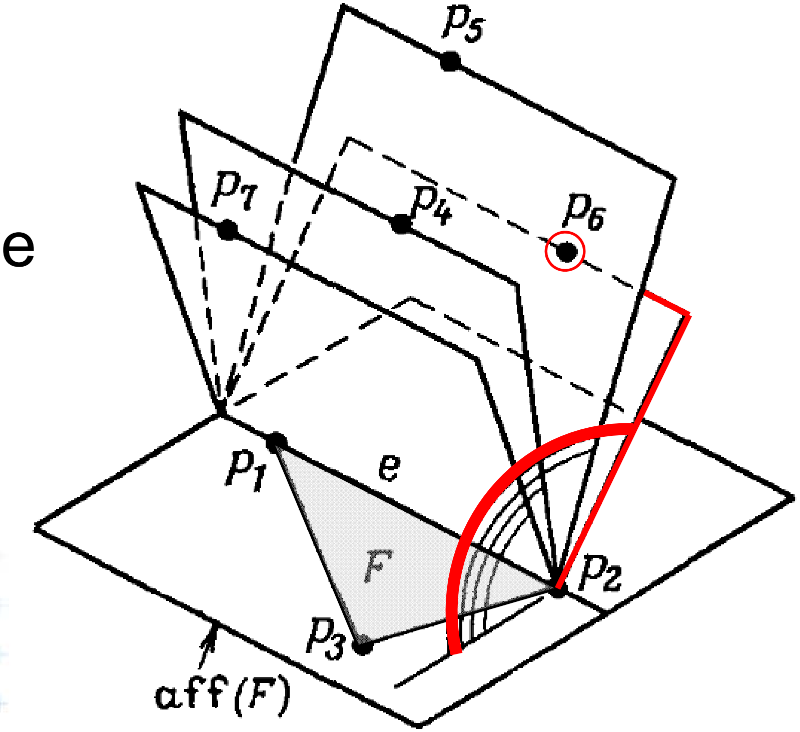
- First known algorithm for n-dimensions (1970)
- Direct extension of 2D alg.
- Complexity O(nF)
  - F is number of CH facets
  - Algorithm is output sensitive
  - Details on seminar, assignment [10]



[Preparata]

1. Gift wrapping in higher dimensions

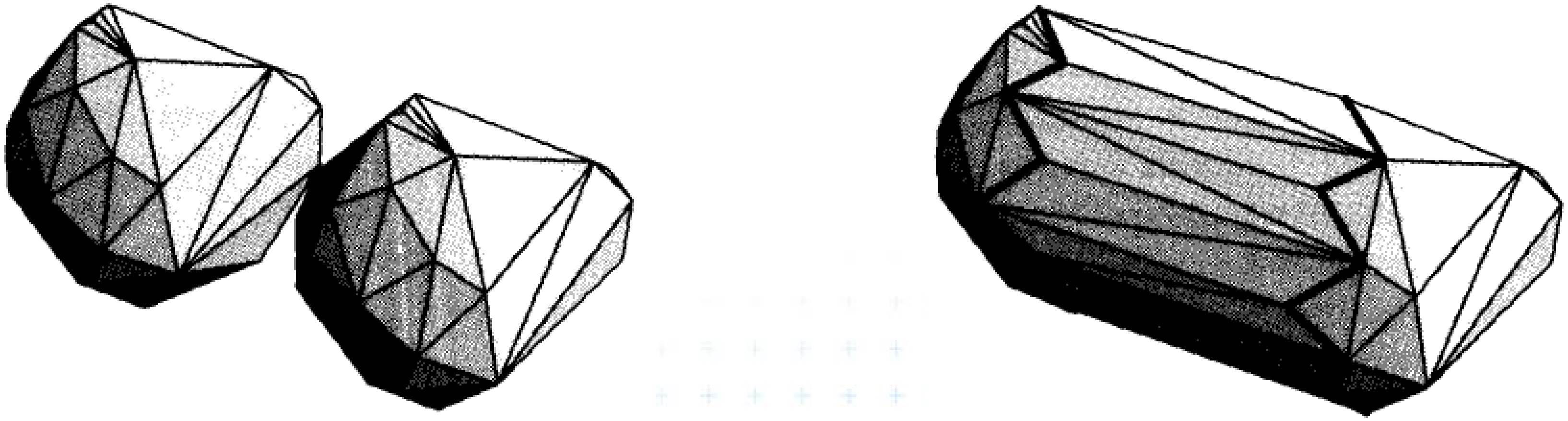
- First known algorithm for n-dimensions (1970)
- Direct extension of 2D alg.
- Complexity O(nF)
  - F is number of CH facets
  - Algorithm is output sensitive
  - Details on seminar, assignment [10]



[Preparata]

2. Divide & conquer 3D convex hull [Preparata, Hong77]

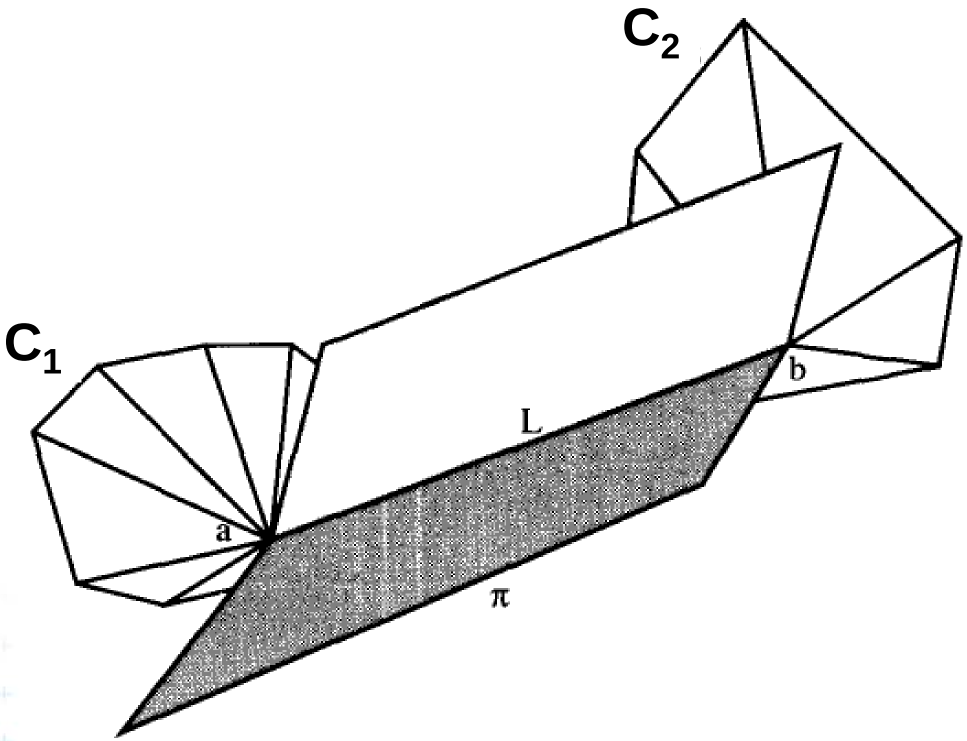
- Sort points in x-coord
- Recursively split, construct CH, merge
- Merge takes O(n) => O(*n* log *n*) total time



[Rourke]

Divide & conquer 3D convex hull [Preparata, Hong 77]

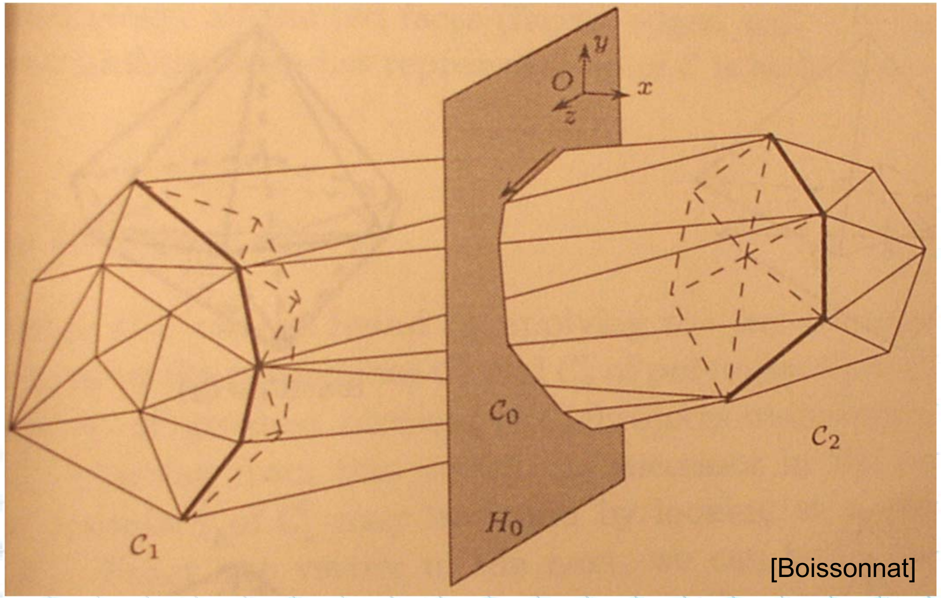
- Merge(C_1 with C_2) uses gift wrapping
  - Gift wrap plane around edge *e* – find new point *p* on C_1 or on C_2 (neighbor of *a* or *b*)
  - Search just the CW or CCW neighbors around *a, b*



[Rourke]

Divide & conquer 3D convex hull [Preparata, Hong 77]

- Performance O(*n* log *n*) rely on circular ordering
  - In 2D: Ordering of points around CH
  - In 3D: Ordering of vertices around 2-polytop C_0 (vertices on intersection of new CH edges with separating plane H_0) [ordering around horizon of C_1 and C_2 does not exist, both horizons may be non-convex and even not simple polygons]
  - In ≥ 4D: Such ordering does not exist



[Boissonnat]

Divide & conquer 3D convex hull [Preparata, Hong 77]

Merge(C_1 with C_2)

- Find the first CH edge *L* connecting C_1 with C_2
- *e* = *L*
- While not back at *L do*
  - store *e* to *C*
  - Gift wrap plane around edge *e* – find new point *P* on C_1 or on C_2 (neighbor of *a* or *b*)
  - *e* = new edge to just found end-point *P*
  - Store new triangle *eP* to *C*
- Discard hidden faces inside CH from *C*
- Report merged convex hull *C*

Divide & conquer 3D convex hull [Preparata, Hong 77]

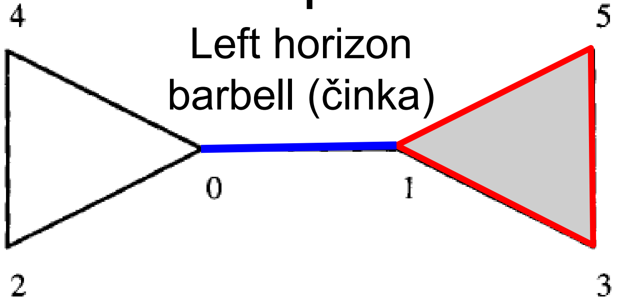
Merge(C_1 with C_2)

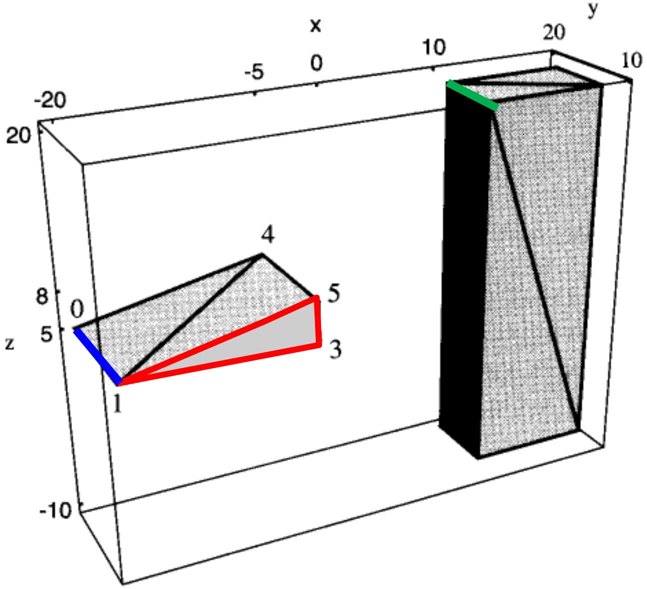
- Find the first CH edge *L* connecting C_1 with C_2
- *e* = *L*
- While not back at *L do* **CHYBA**
  - store *e* to *C*
  - Gift wrap plane around edge *e* – find new point *P* on C_1 or on C_2 (neighbor of *a* or *b*)
  - *e* = new edge to just found end-point *P*
  - Store new triangle *eP* to *C*
- Discard hidden faces inside CH from *C*
- Report merged convex hull *C*

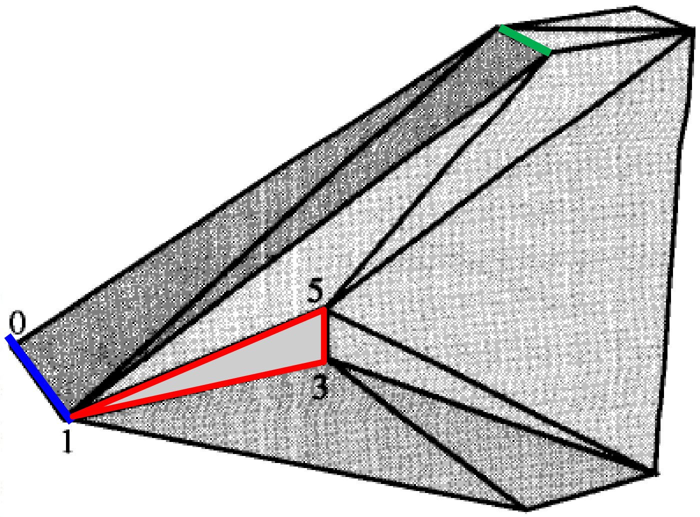
Divide & conquer 3D convex hull [Preparata, Hong 77]

- Problem of the wrapping phase [Edelsbrunner 88]
  - The edges on horizon do not form simple circle but a "barbell" 0,2,4,0,1,3,5,1

Do not stop here! ↑


Left horizon barbell (činka)





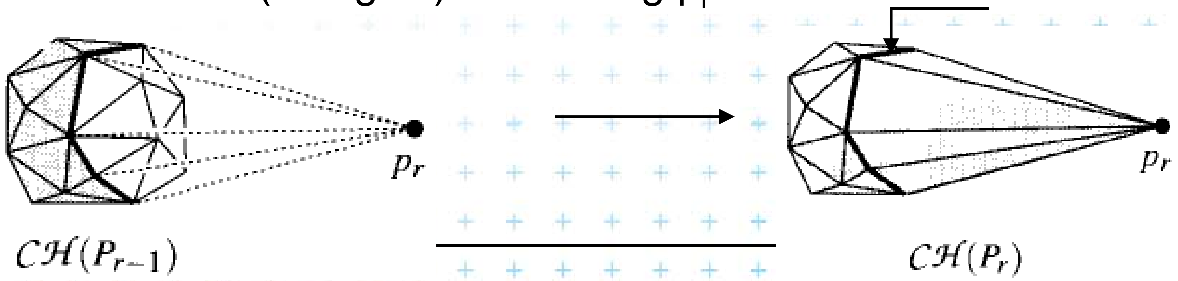
[Berg]

3. Randomized incremental alg. principle

1. Create tetrahedron (smallest CH in 3D)
   - Take 2 points p_1 and p_2
   - Search the 3rd point not lying on line p_1p_2
   - Search the 4th point not lying in plane $p_1p_2\,p_3$ …if not found, use 2D CH
2. Perform random permutation of remaining points $\{p_5, \ldots, p_n\}$
3. For p_r in $\{p_5, \ldots, p_n\}$ do add point p_r to CH(P_{r-1})

   Notation: for $r \geq 1$ let $P_r = \{p_1, \ldots, p_r\}$ is set of already processed pts

   - If p_r lies inside or on the boundary of CH(P_{r-1}) then do nothing
   - If p_r lies outside of CH(P_{r-1}) then
     - find and remove visible faces
     - create new faces (triangles) connecting p_r with lines of horizon



[Berg]

Conflict graph

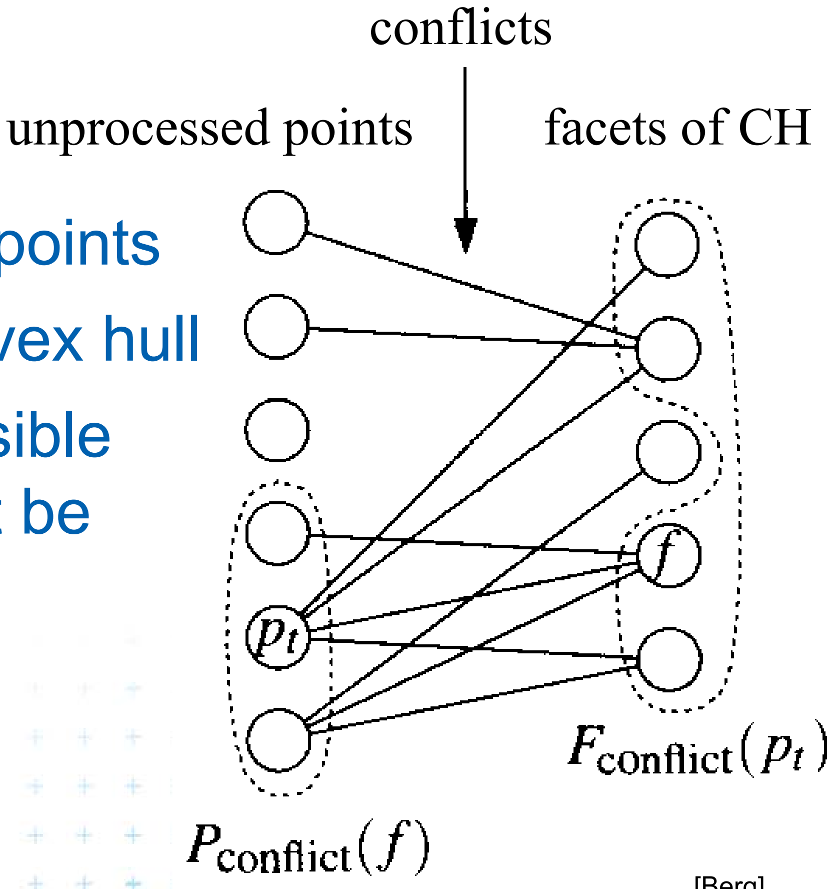
- Stores unprocessed points with facets of CH they see

- Bipartite graph

  points p_t, $t > r$ … unprocessed points

  facets of CH(P_r)… facets of convex hull

  conflict arcs … conflict, as visible facets cannot be in CH



[Berg]

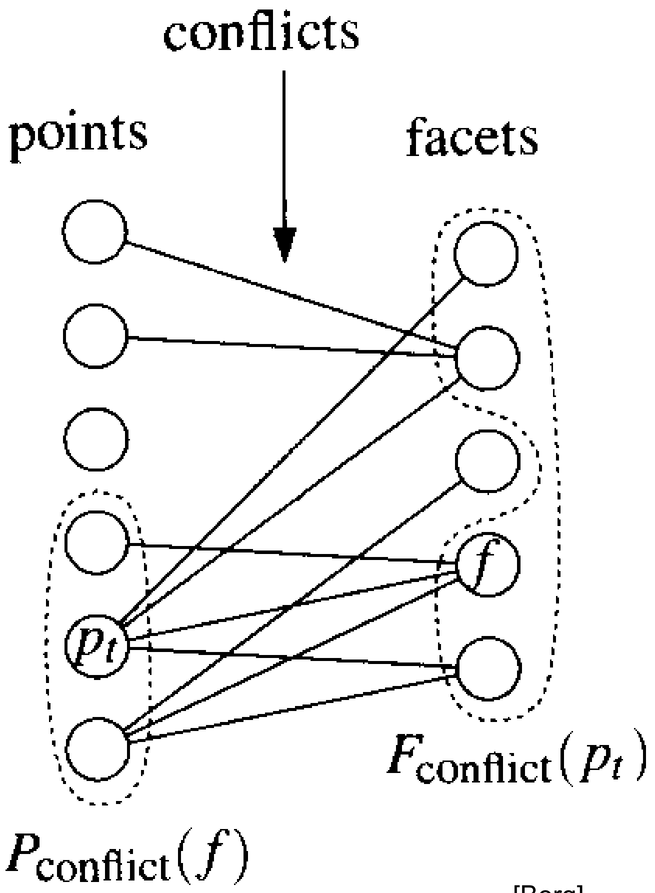
- Maintains sets:

  $P_{\text{conflict}}(f)$ … points, that see f

  $F_{\text{conflict}}(p_r)$… facets visible from p_r (visible region – deleted after insertion of p_r)

Conflict graph – init and final state

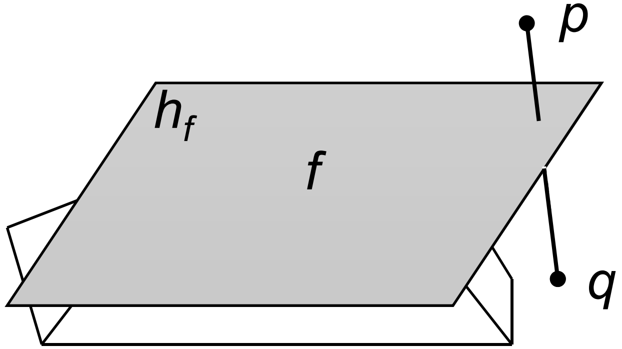
- Initialization
  - Points $\{p_5, \ldots, p_n\}$ (not in tetrahedron)
  - Facets of the tetrahedron (four)
  - Arcs – connect each tetrahedron facet with points visible from it
- Final state
  - Points – {} = empty set
  - Facets of the convex hull
  - Arcs - none



[Berg]

Visibility between point and face

- Face *f* is visible from a point *p* if that point lies in the open half-space on the other side of h_f than the polytope



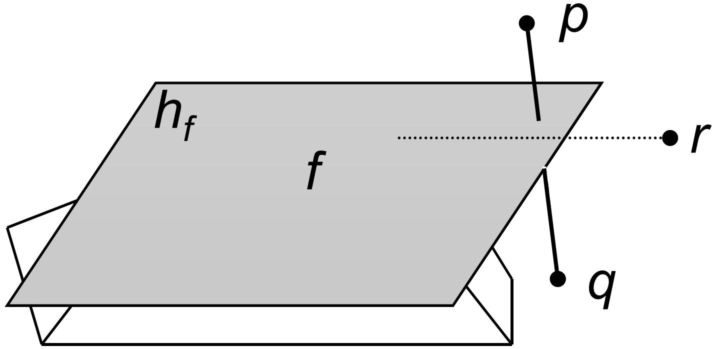
f is visible from *p* (*p* is *above* the plane)

f is not visible from *q*

$p \in P_{conflict}(f)$, p is among the points that see the face f
$f \in F_{conflict}(p)$ f is among the faces visible from point p

Visibility between point and face

- Face *f* is visible from a point *p* if that point lies in the open half-space on the other side of h_f than the polytope



f is visible from *p* (*p* is *above* the plane)

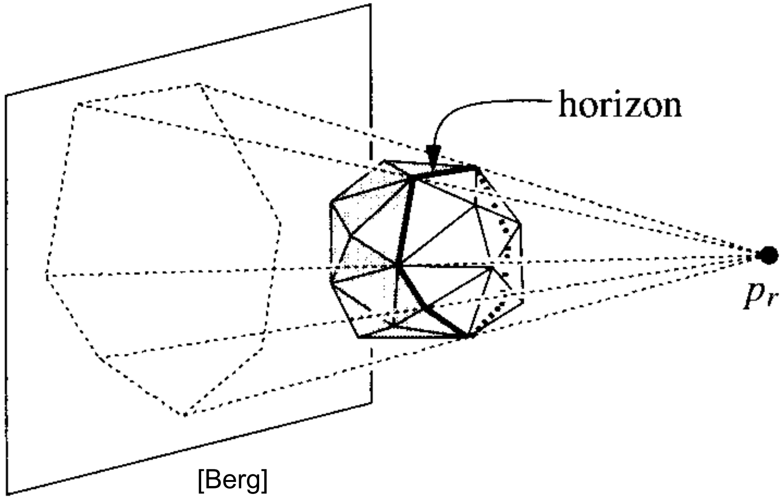
f is not visible from *r* lying *in the plane* of *f* (this case will be discussed next)

f is not visible from *q*

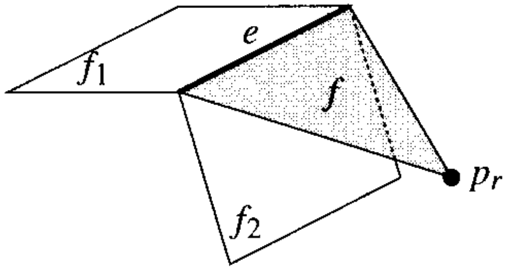
$p \in P_{conflict}(f)$, p is among the points that see the face f
$f \in F_{conflict}(p)$ f is among the faces visible from point p

New triangles to horizon

- Horizon = edges *e* incident to visible and invisible facets



[Berg]



[Berg]

- New triangle *f* connects edge *e* on horizon and point p_r and
  - creates new node for facet *f* updates the conflict graph
  - add arcs to points visible from *f* (subset from $P_{coflict}(f_1) \cup P_{coflict}(f_2)$)
- Coplanar triangles on the plane ep_r are merged with new triangle. Conflicts in G are copied from the deleted triangle (same plane)

Overview of new point insertion

Processing of point p_r outside

- Remove facets that p_r sees from the CH (do not delete them from the graph G)
- Find horizon edges (around the hole in CH)
- Create new facets from horizon edges to p_r
  - add them to CH
  - create face nodes f in G for them
- Compute what p_r sees – search only from $P(e) \ = P_{conflict}(f_1) \cup P_{conflict}(f_2)$)
- Delete node p_r and face $F_{conflict}(p_r)$ from G

Incremental Convex hull algorithm

```
IncrementalConvexHull($P$)
Input:   Set of $n$ points in general position in 3D space
Output:  The convex hull $C = CH(P)$ of $P$
Find four points that form an initial tetrahedron, $C = CH(\{p_1, p_2, p_3, p_4\})$
Compute random permutation $\{p_5, p_6, \dots, p_n\}$ of the remaining points
Initialize the conflict graph $G$ with all visible pairs $(p_t, f)$,
where $f$ is facet of $C$ and $p_t, t > 4$, are non-processed points
for $r$ = 5 to $n$ do                                   …inserting $p_r$, into $C$
    if($F_{conflict}(p_r)$ is not empty) then            …$p_r$ is outside, insert $p_r$, into $C$
        Delete all facets $F_{conflict}(p_r)$ from $C$ … only from hull C, not from $G$
        Walk around visible region boundary, create list $L$ of horizon edges
        for all $e \in L$ do
            connect $e$ to $p_r$ by a new triangular facet $f$
            if $f$ is coplanar with its neighbor facet $f'$ along $e$
                then merge $f$ and $f'$ in $C$, take conflict list from $f'$
                else … determine conflicts for new facet $f$
                    … [continue on the next slide]
```

Incremental Convex hull algorithm (cont…)

```
            else … not coplanar => determine conflicts for new facet f
                Insert $f$ into hull $C$
                Create node for $f$ in $G$          //… new face in conflict graph G
                Let $f_1$ and $f_2$ be the facets incident to $e$ in the old $CH(P_{r-1})$
                $P(e) \ = P_{conflict}(f_1) \cup P_{conflict}(f_2)$
                for all points $p \ \in P(e)$ do
                    if $f$ is visible from $p$, then add$(p, f)$ to $G$ … new edges in G
        Delete the node corresponding to $p_r$ and the nodes corresponding
        to facets in $F_{conflict}(p_r)$ from $G$, together with their incident arcs
return $C$
```

Complexity: Convex hull of a set of points in E^3 can be computed incrementally in $O(n \log n)$ randomized expected time (process $O(n)$ points, but number of facets and arcs depend on the order of inserting points – up to $O(n^2)$)

For proof see: [Berg, Section11.3]

Convex hull in higher dimensions

- Convex hull in *d* dimensions can have $\Omega(n^{\lfloor d/2 \rfloor})$
  Proved by [Klee, 1980]
- Therefore, 4D hull can have quadratic size
- No *O(n log n)* algorithm possible for d>3
- These approaches can extend to d>3
  - Gift wrapping
  - D&C
  - Randomized incremental
  - QuickHull

Conclusion

- Recapitulation of 2D algorithms
- >=3D algorithms
  - Gift wrapping
  - D&C
  - Randomized incremental
  - QuickHull

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, Chapter 11, http://www.cs.uu.nl/geobook/

[Boissonnat] J.-D. Boissonnat and M. Yvinec, *Algorithmic Geometry*, Cambridge University Press, UK, 1998. Chapter 9 – Convex hulls

[Preparata] Preperata, F.P., Shamos, M.I.: *Computational Geometry. An Introduction.* Berlin, Springer-Verlag,1985.

[Mount] David Mount, - *CMSC 754: Computational Geometry, Lecture Notes for Spring 2007*, University of Maryland, Lecture 3. http://www.cs.umd.edu/class/spring2007/cmsc754/lectures.shtml

[Chan] Timothy M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions., *Discrete and Computational Geometry*, 16, 1996, 361-368. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.389

DCGI

DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

VORONOI DIAGRAM

PETR FELKEL

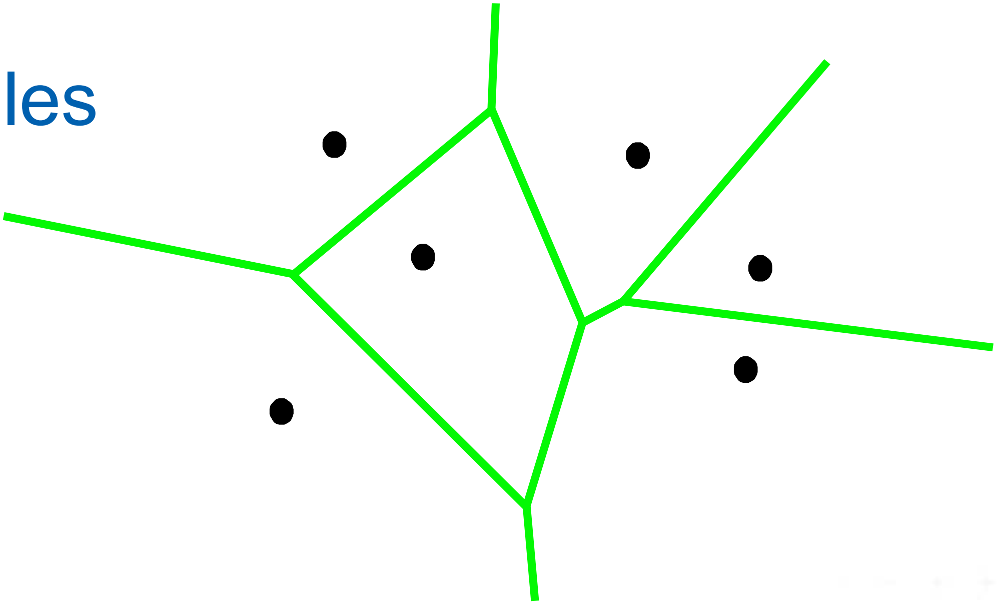
FEL CTU PRAGUE

felkel@fel.cvut.cz

Version from 8.11.2018

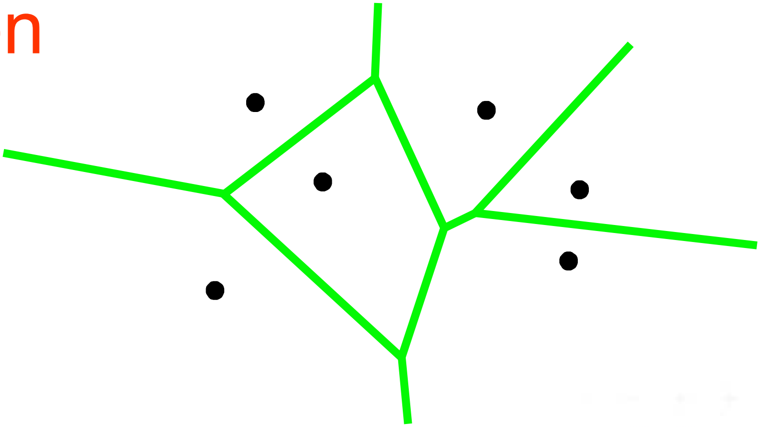
Talk overview

- Definition and examples
- Applications
- Algorithms in 2D
  - D&C O(n log n)
  - Sweep line O(n log n)

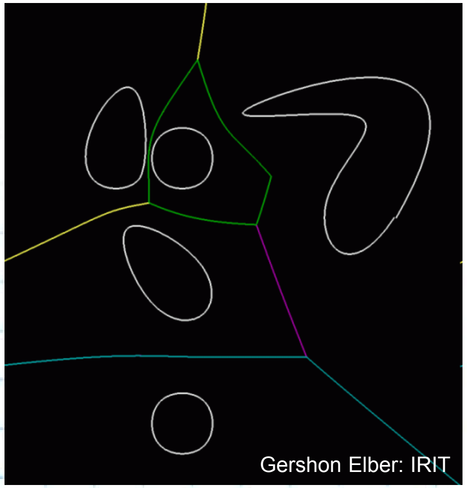


Voronoi diagram (VD)

- One of the most important structure in Comp. geom.
- Encodes proximity information
  What is close to what?



- Standard VD – this lecture
  - Set of points - nDim
  - Euclidean space & metric
- Generalizations
  - Set of line segments or curves
  - Different metrics
  - Higher order VD's (furthest point)



Gershon Elber: IRIT

Voronoi cell (for points in plane)

- Let $P = \{p_1, p_2, \ldots, p_n\}$ be a set of points (*sites*) in dDim space … 2D space (plane) here
- Voronoi cell $V(p_i)$ – is open!
  = set of points q closer to p_i than to any other site:

  $$V(p_i) = \{q, \|p_i q\| < \|p_j q\|, \forall j \neq i\},$$ where

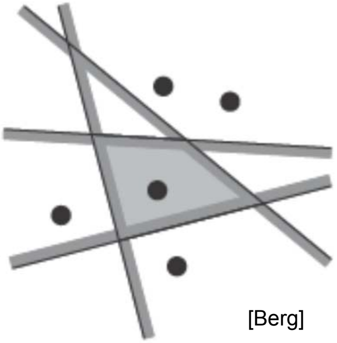
  $\|pq\|$ is the Euclidean distance between p and q

  = intersection of open halfplanes

  $$V(p_i) = \bigcap_{j \neq i} h(p_i, p_j)$$

  $h(p_i, p_j)$ = open halfplane

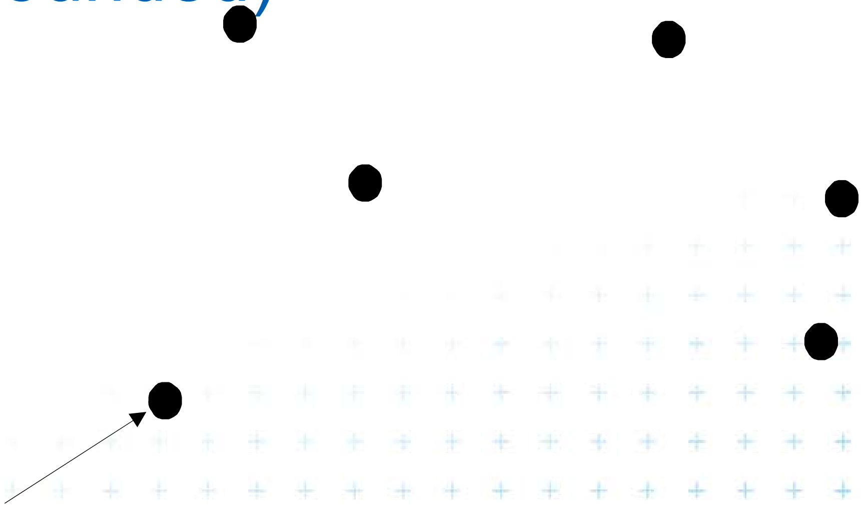
  = set of pts strictly closer to p_i than to p_j



[Berg]

Voronoi diagram (in plane)

- Voronoi diagram Vor(*P*) of points *P*
  = what is left of the plane after removing all the open Voronoi cells
  = collection of line segments (possibly unbounded)

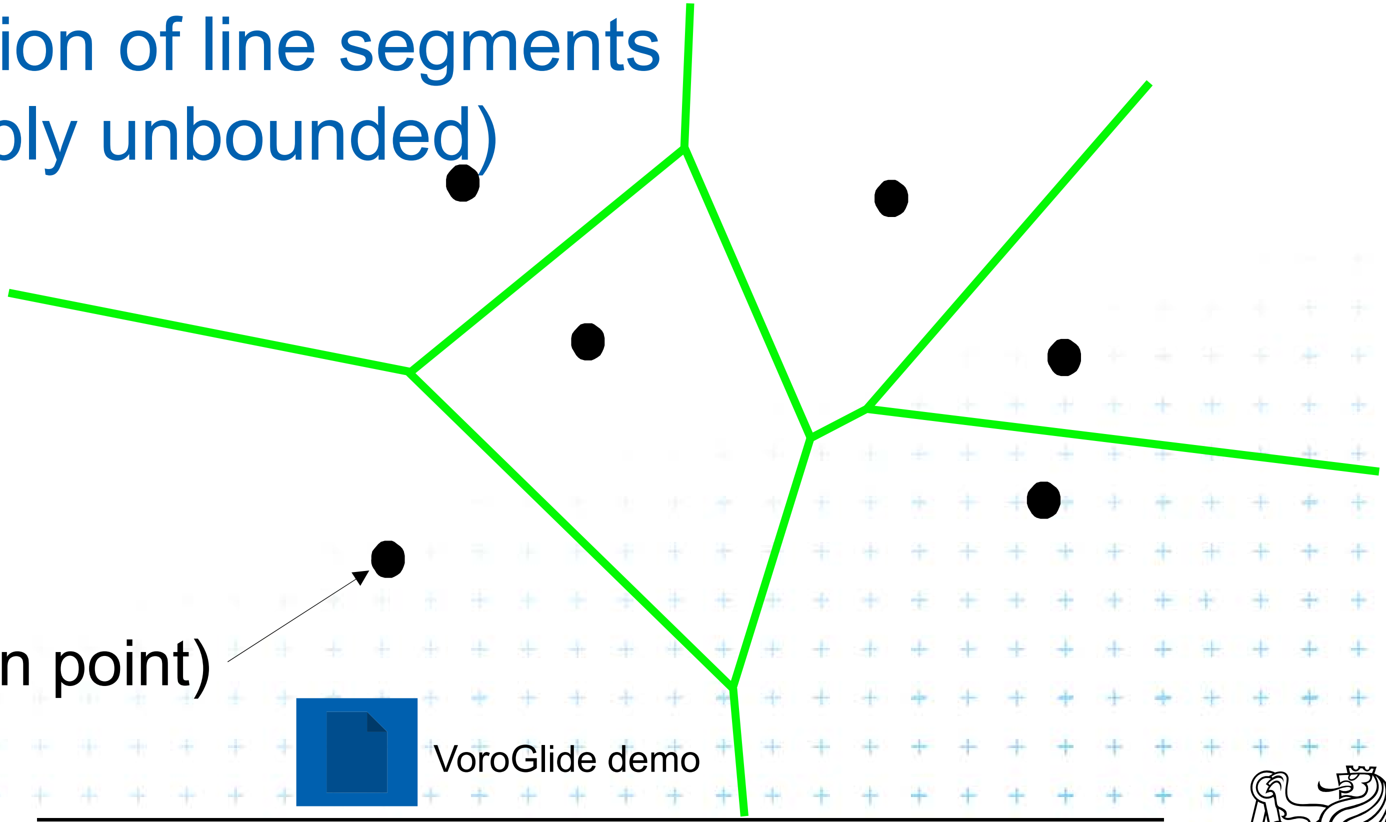


Site (given point)

VoroGlide demo

Voronoi diagram (in plane)

- Voronoi diagram Vor(P) of points P
  = what is left of the plane after removing all the open Voronoi cells
  = collection of line segments (possibly unbounded)

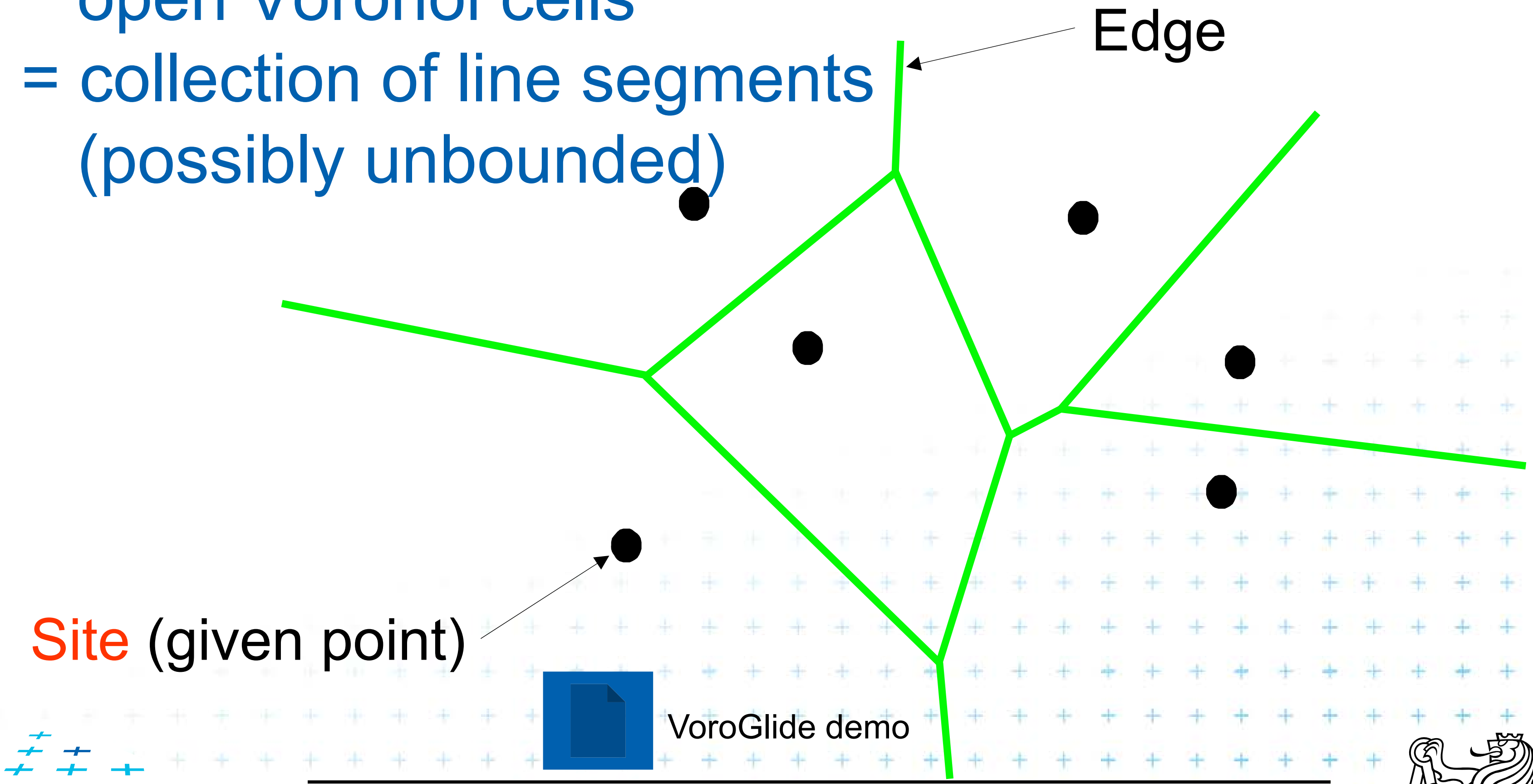


Site (given point)

VoroGlide demo

Voronoi diagram (in plane)

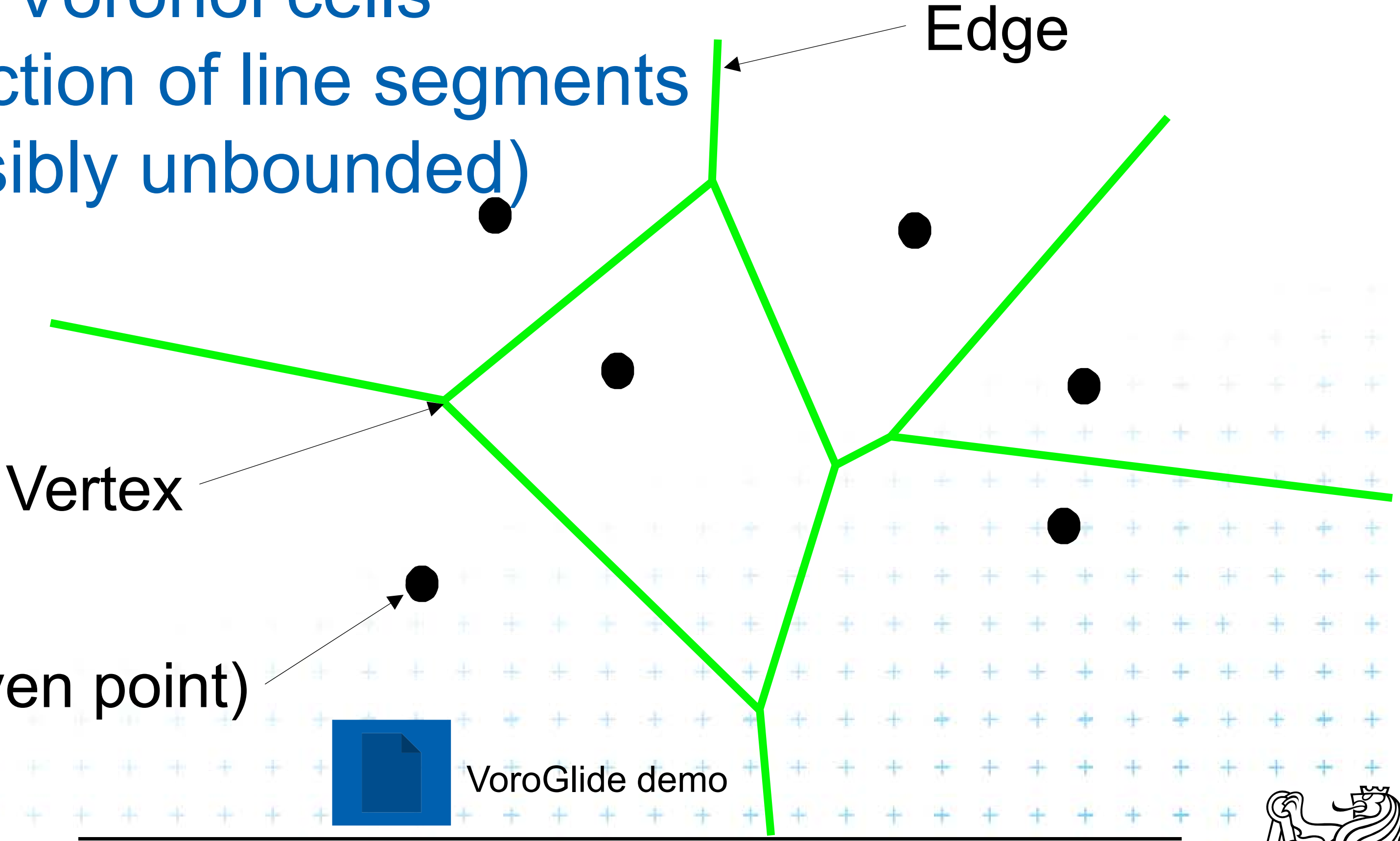
- Voronoi diagram Vor(*P*) of points *P*
  = what is left of the plane after removing all the open Voronoi cells
  = collection of line segments (possibly unbounded)



VoroGlide demo

Voronoi diagram (in plane)

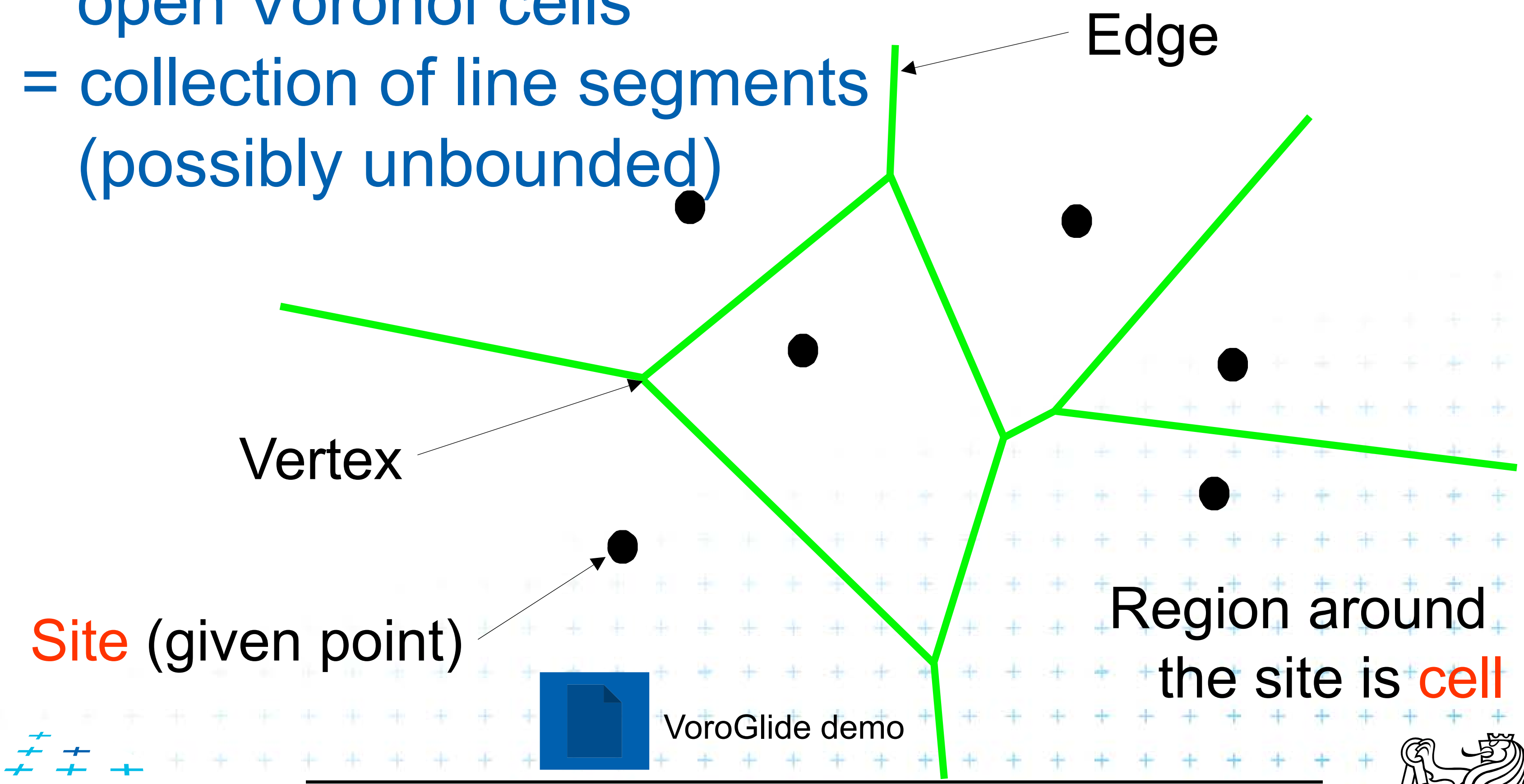
- Voronoi diagram Vor(*P*) of points *P*
  = what is left of the plane after removing all the open Voronoi cells
  = collection of line segments (possibly unbounded)



VoroGlide demo

Voronoi diagram (in plane)

- Voronoi diagram Vor(*P*) of points *P*
  = what is left of the plane after removing all the open Voronoi cells
  = collection of line segments (possibly unbounded)



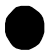
VoroGlide demo

Voronoi diagram examples

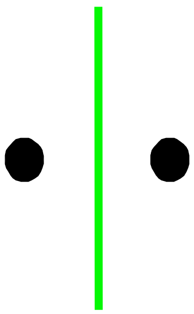
1 point

Voronoi diagram examples

1 point

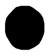


2 points

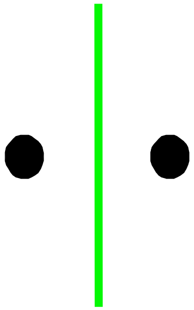


Voronoi diagram examples

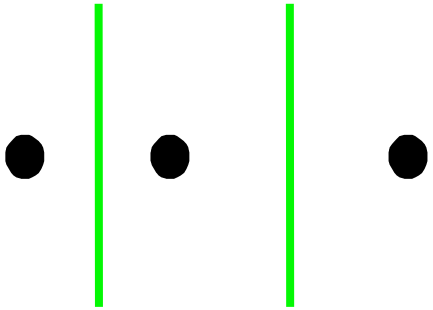
1 point



2 points

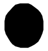


3 points

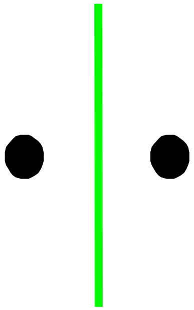


Voronoi diagram examples

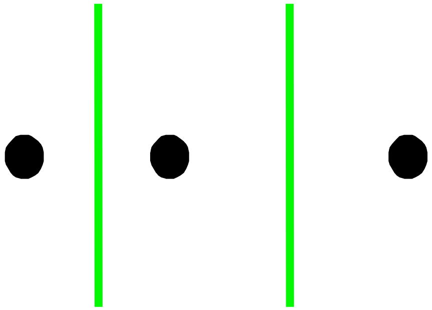
1 point

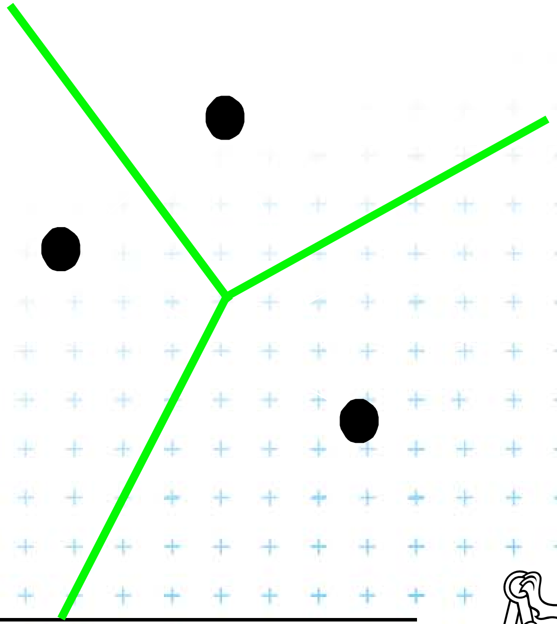


2 points



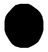
3 points



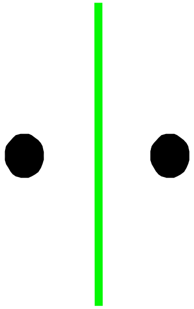


Voronoi diagram examples

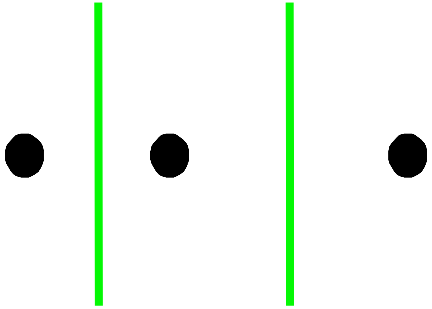
1 point

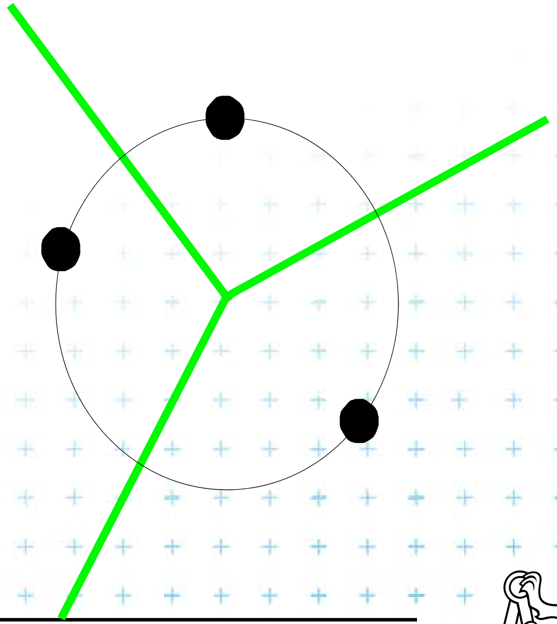


2 points



3 points



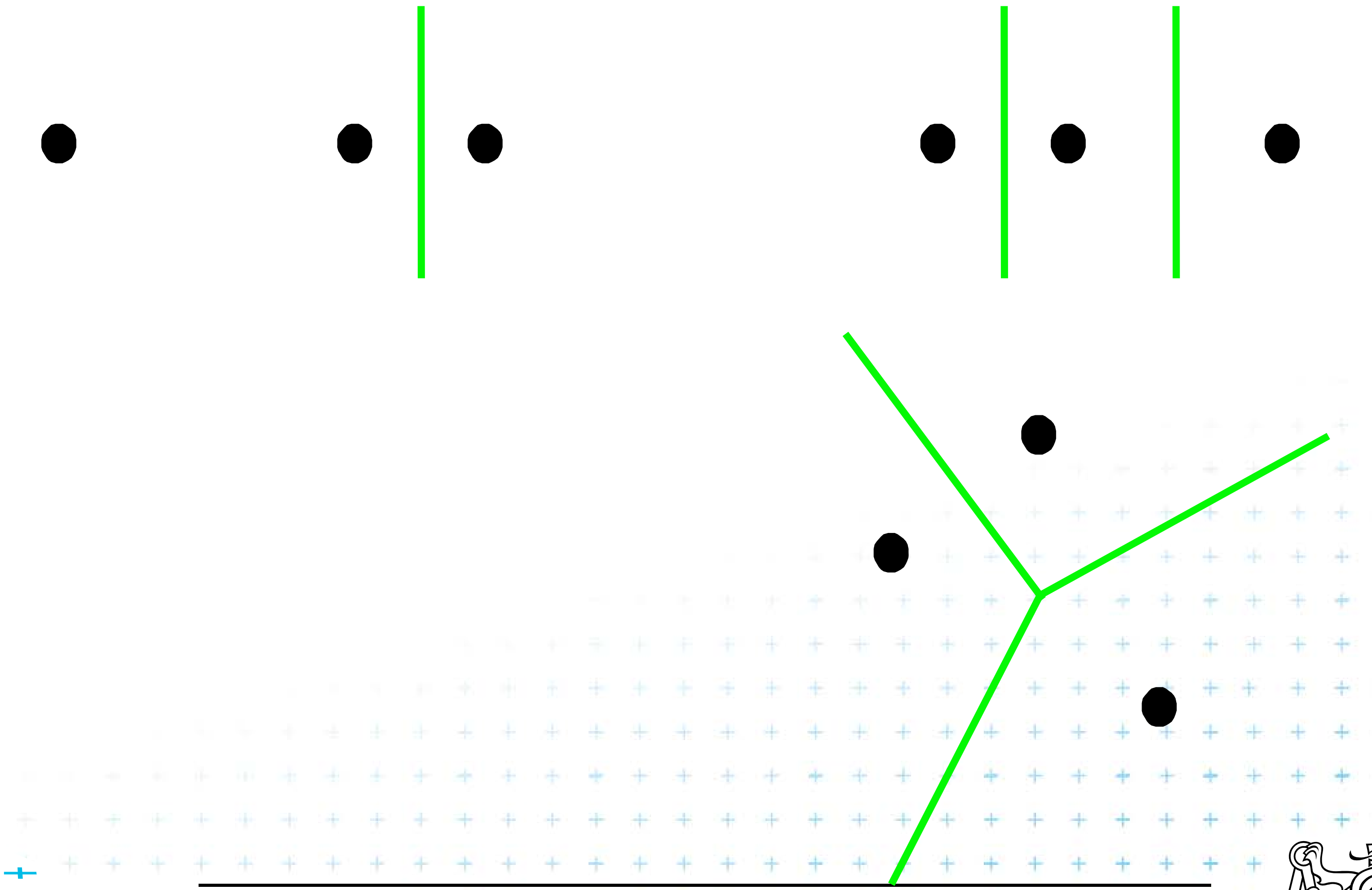


Voronoi diagram examples

1 point

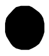
2 points

3 points

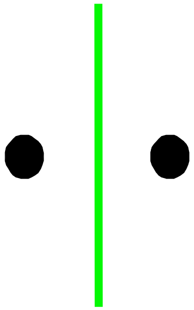


Voronoi diagram examples

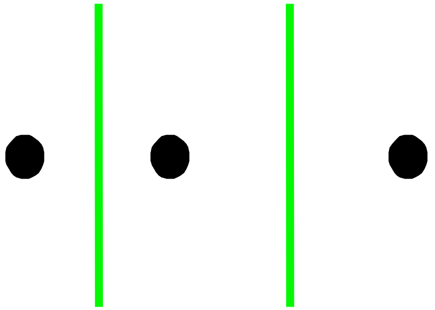
1 point



2 points



3 points

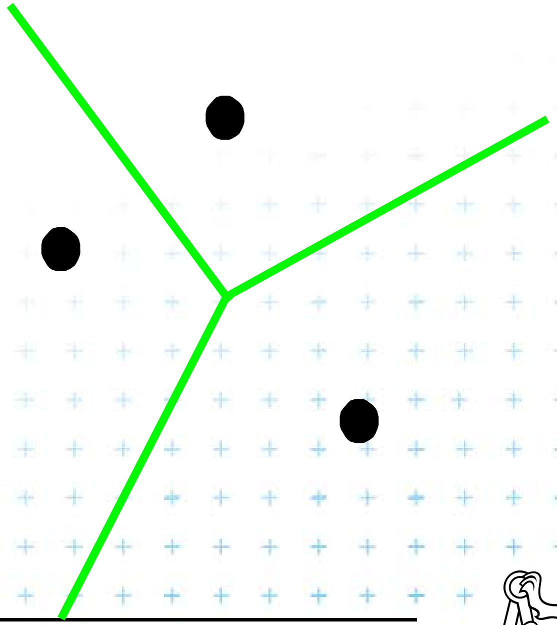


Cell

- The whole plain for 1 point
- Halfplane or strip for collinear points
- Convex (possibly unbounded) polygon

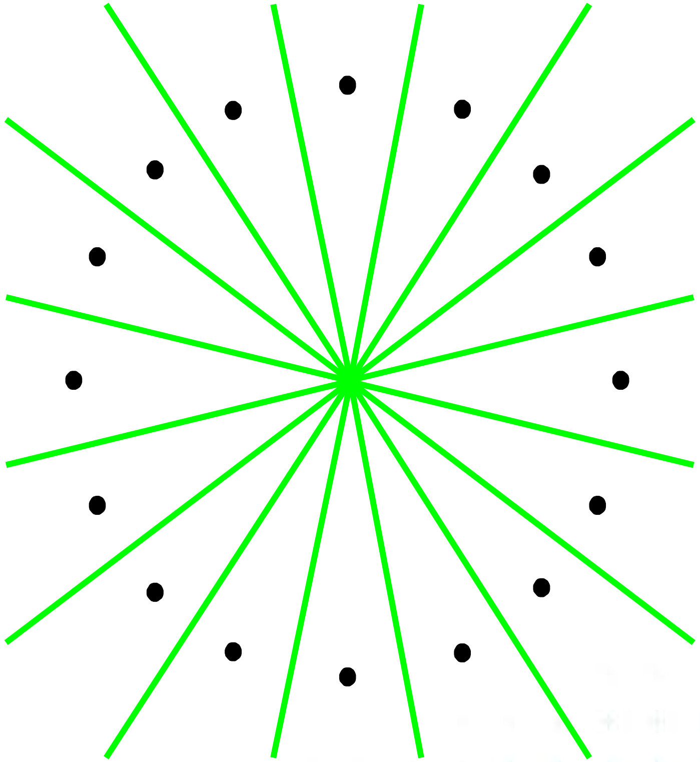
Edges of VD

- || lines for collinear points
- Halflines (for non-collinear CH points)
- Line segments (for bounded cells)



Voronoi diagram examples

16 points



[Håkan Jonsson]

Vertex with O(n) incident edges

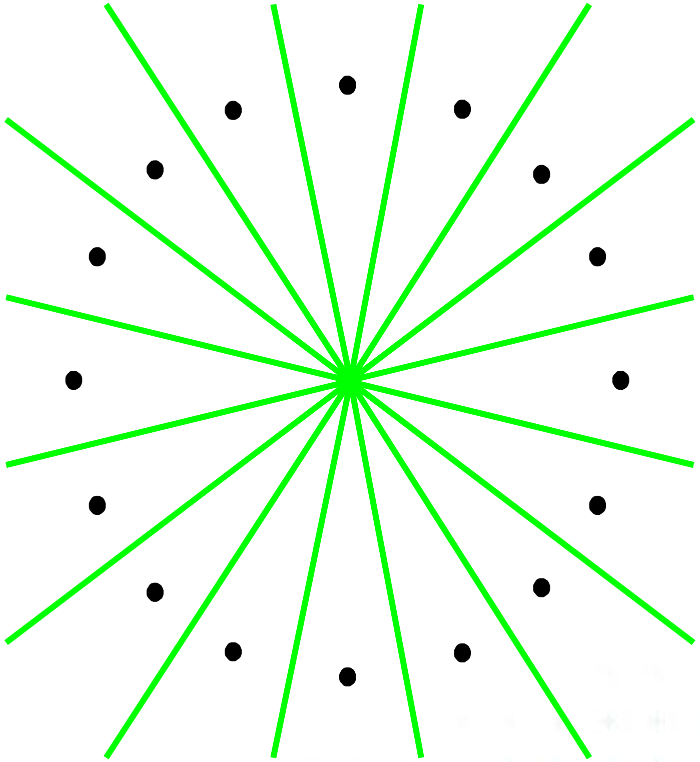
From total $|n_e| \leq 3n - 6$

16 <= 42

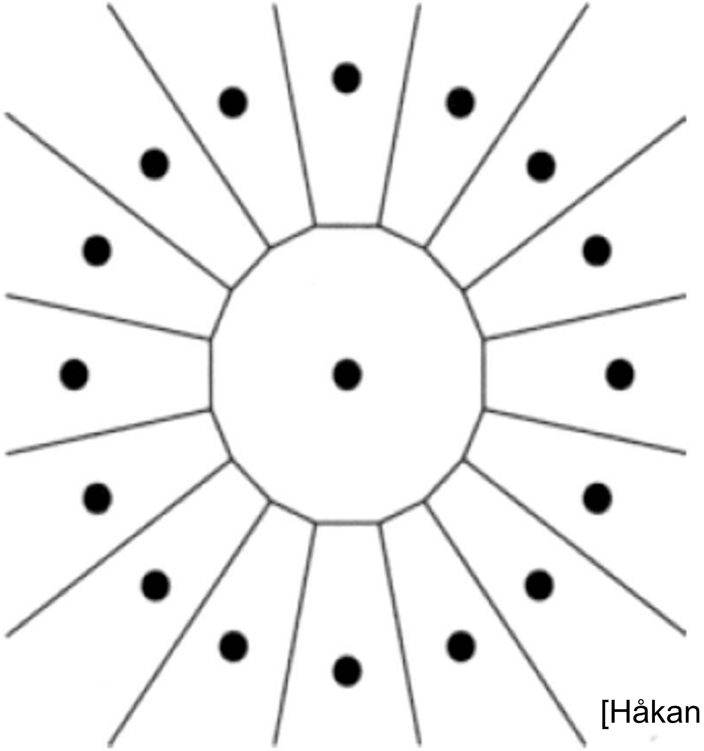
17 <= 29

Voronoi diagram examples

16 points



17 points



[Håkan Jonsson]

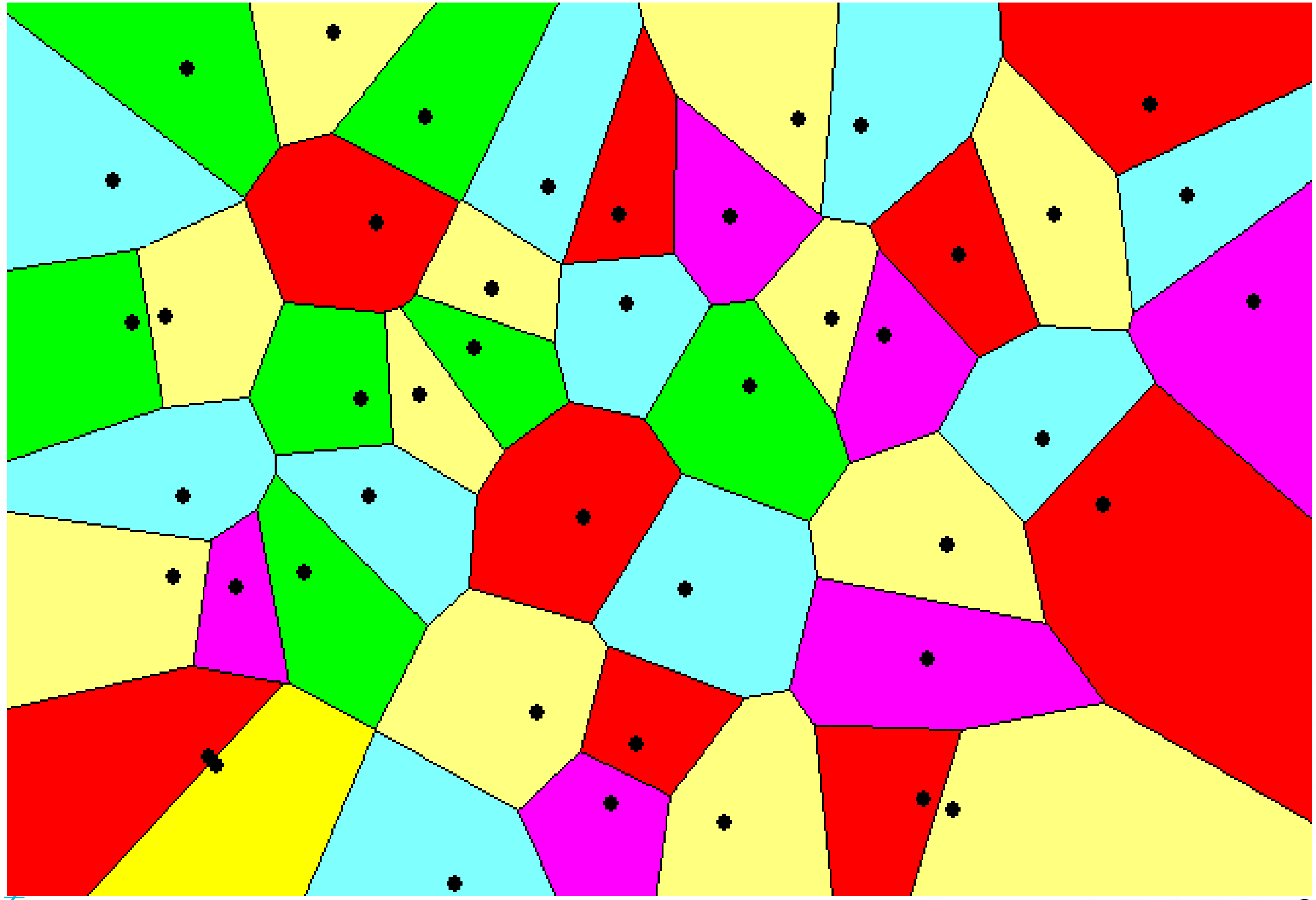
Vertex with O(n) incident edges
From total $|n_e| \leq 3n - 6$

16 <= 42

Cell with O(n) vertices
From total $|n_v| \leq 2n\text{-}5$

17 <= 29

Voronoi diagram examples

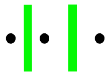


Voronoi diagram (in plane)

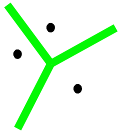
= planar graph

- Subdivides plane into *n* cells (*n* = num. of input sites |P|)
- Edge = locus of equidistant pairs of points (cells)
  = part of the bisector of these points
- Vertex = center of the circle defined by ≥ 3 points
  => vertices have degree ≥ 3
- Number of vertices $n_v \leq 2n - 5$ => O(*n*)
- Number of edges $n_e \leq 3n - 6$ => O(*n*)
  (only O(*n*) from O(n^2) intersections of bisectors)- -
- In higher dimensions complexity from O(*n*) up to O($n^{|d/2|}$)
- Unbounded cells belong to sites (points) on convex hull

Voronoi diagram O(n) complexity derivation



- For *n* collinear sites: $n_v = 0 \leq 2n - 5$, $n_e = (n-1) \leq 3n - 6$ both hold



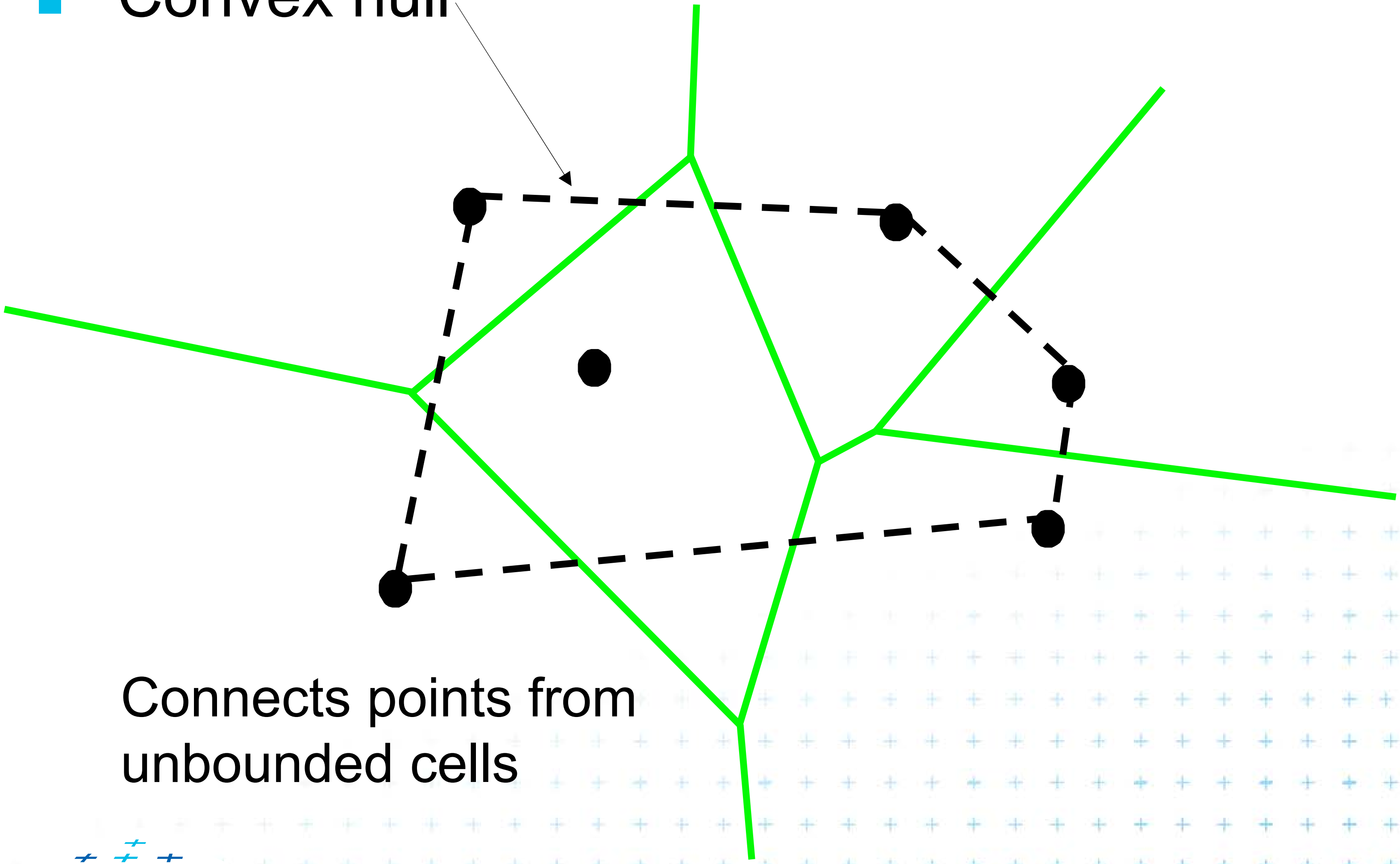
- For *n* non-collinear sites:
  - Add extra VD vertex *v* in infinity $m_v = n_n + 1$
  - Apply Euler's formula: $m_v - m_e + m_f = 2$
  - Obtain $(n_v + 1) - n_e + n = 2$ $\left\{ \begin{array}{l} n_e = n_v + n - 1 \\ n_v = n_e - n + 1 \end{array} \right.$
  - Every VD edge has 2 vertices Sum of vertex degrees = $2n_e$
  - Every VD vertex has degree ≥ 3 Sum of vertex degrees = $3m_v = 3(n_v + 1)$
  - Together $2n_e \geq 3(n_v + 1)$

$$
\begin{aligned}
2n_e &\geq 3(n_v + 1) \\
2(n_v + n - 1) &\geq 3(n_v + 1) \\
2n_v + 2n - 2 &\geq 3n_v + 3 \\
n_v &\leq 2n - 5
\end{aligned}
$$

$$
\begin{aligned}
2n_e &\geq 3(n_v + 1) \\
2n_e &\geq 3(n_e - n + 1 + 1) \\
2n_e &\geq 3n_e - 3n + 6 \\
n_e &\leq 3n - 6
\end{aligned}
$$

Voronoi diagram and convex hull

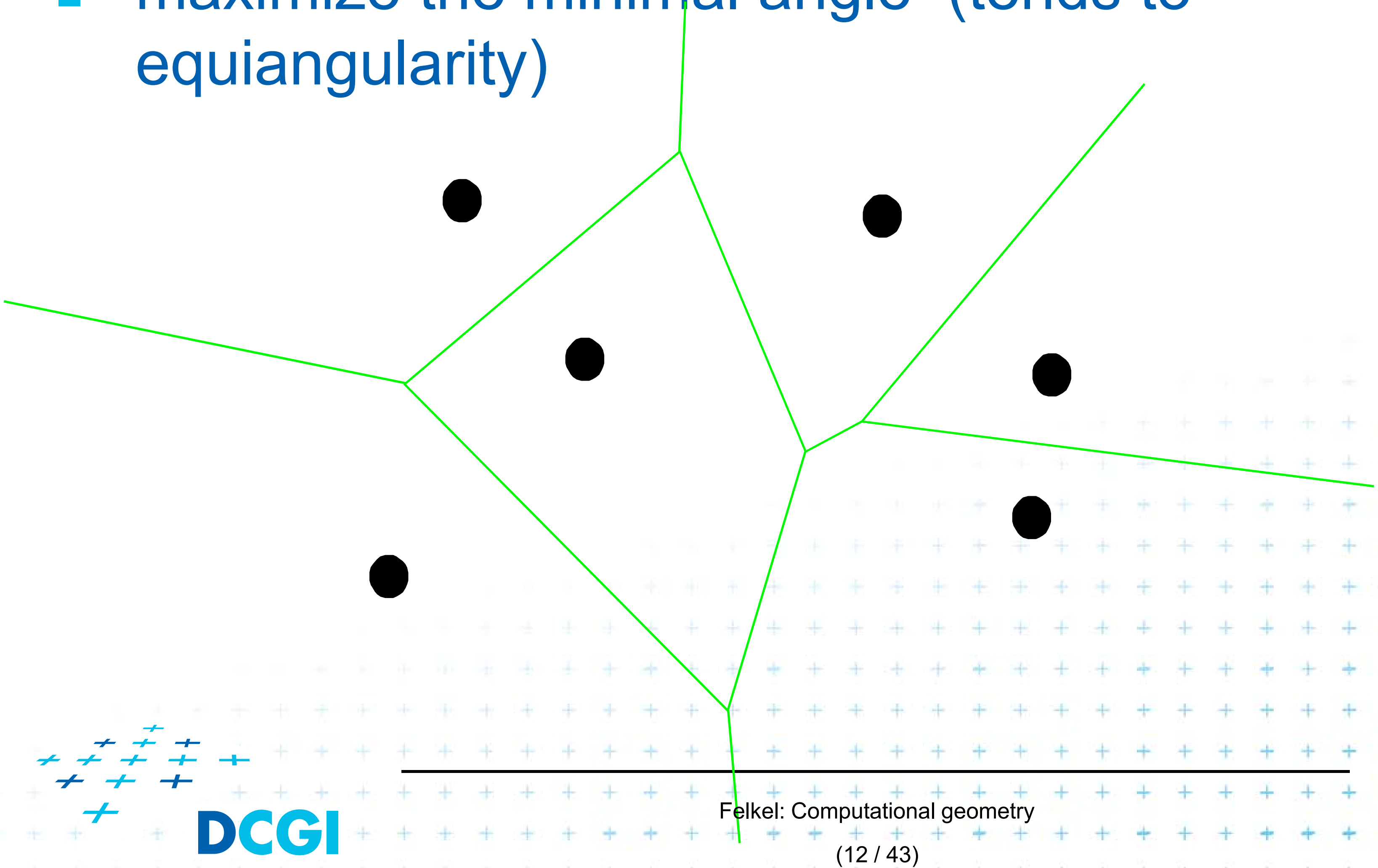
- Convex hull



Connects points from unbounded cells

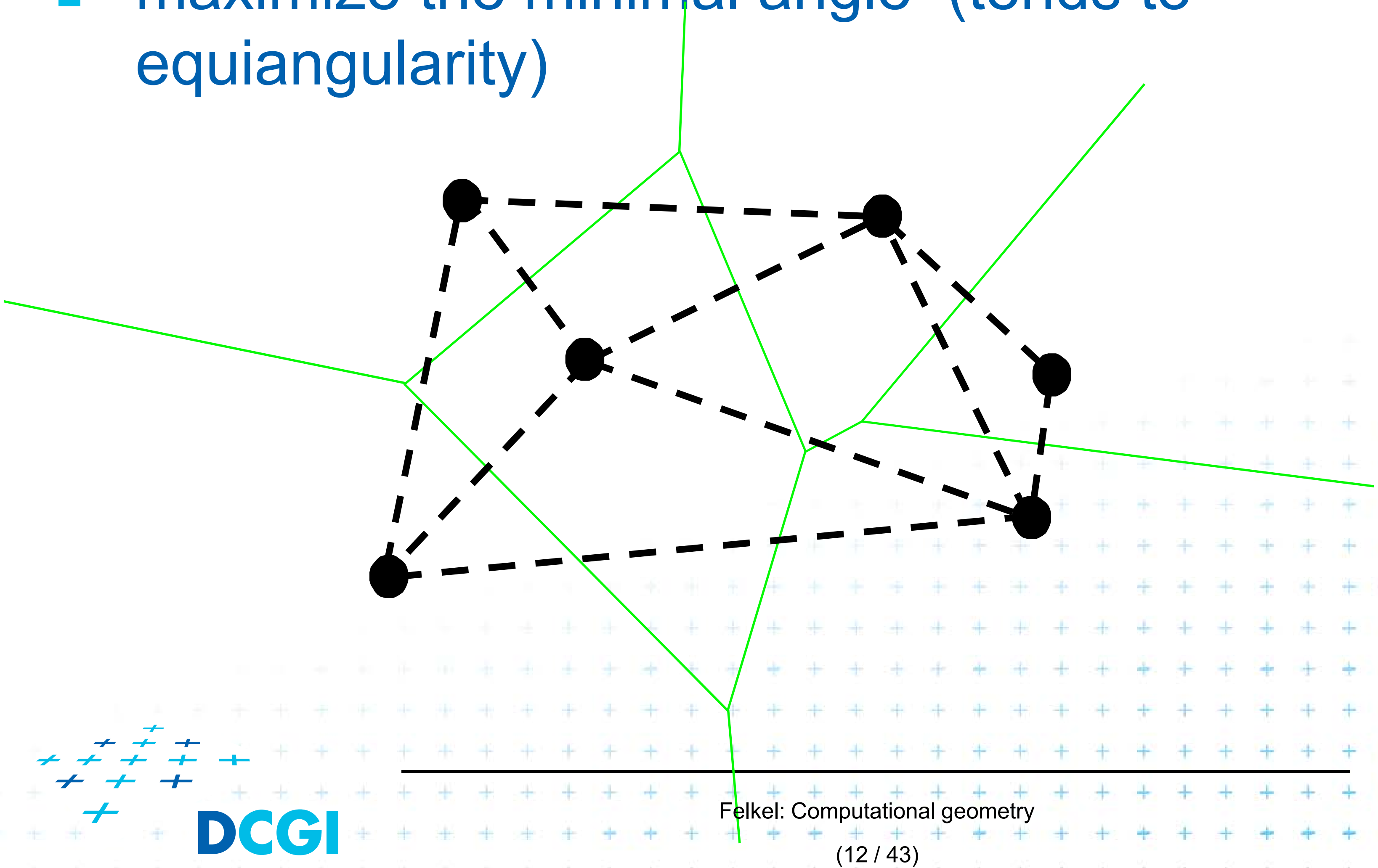
Delaunay triangulation

- point set triangulation   (straight line dual to VD)
- maximize the minimal angle (tends to equiangularity)



Delaunay triangulation

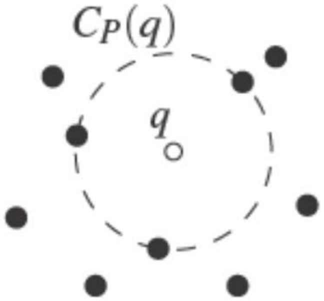
- point set triangulation (straight line dual to VD)
- maximize the minimal angle (tends to equiangularity)



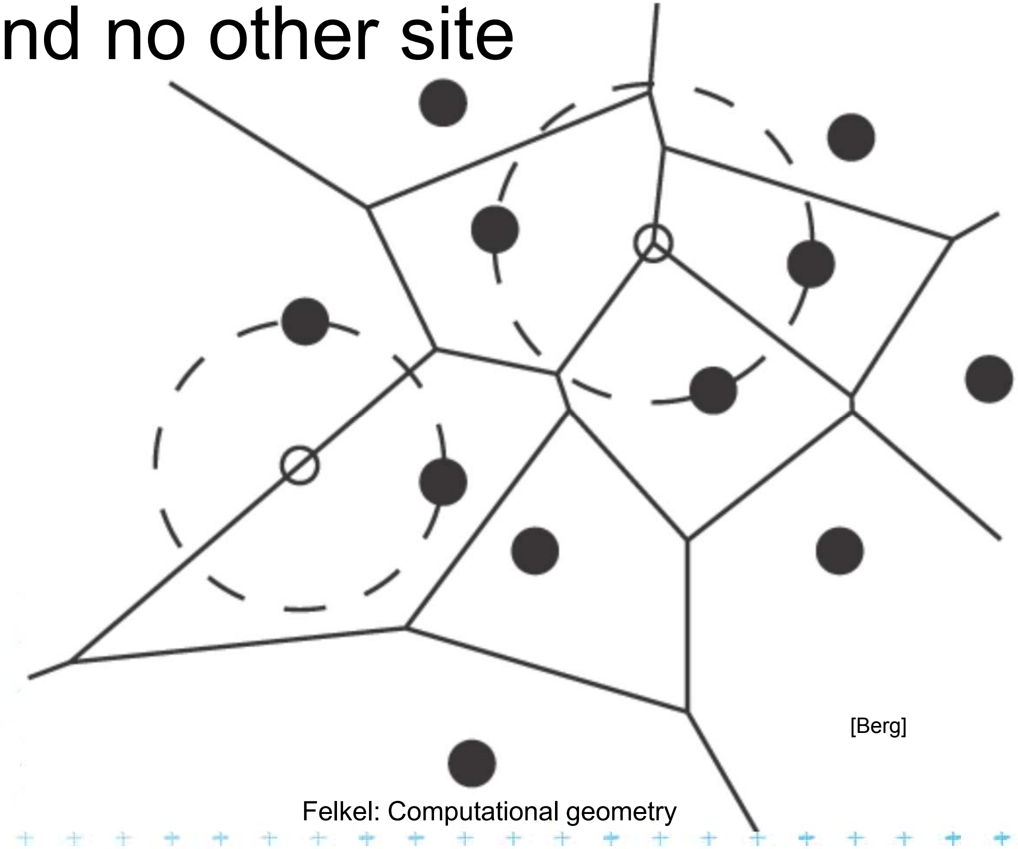
Edges, vertices and largest empty circles

Largest empty circle $C_P(q)$ with center in

1. In VD vertex q: has 3 or more sites on its boundary
2. On VD edge: contains exactly 2 sites on its boundary and no other site



[Berg]

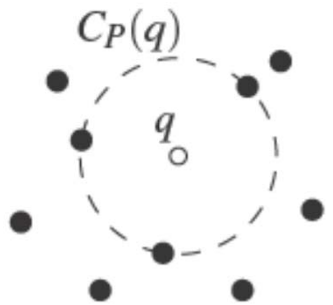


[Berg]

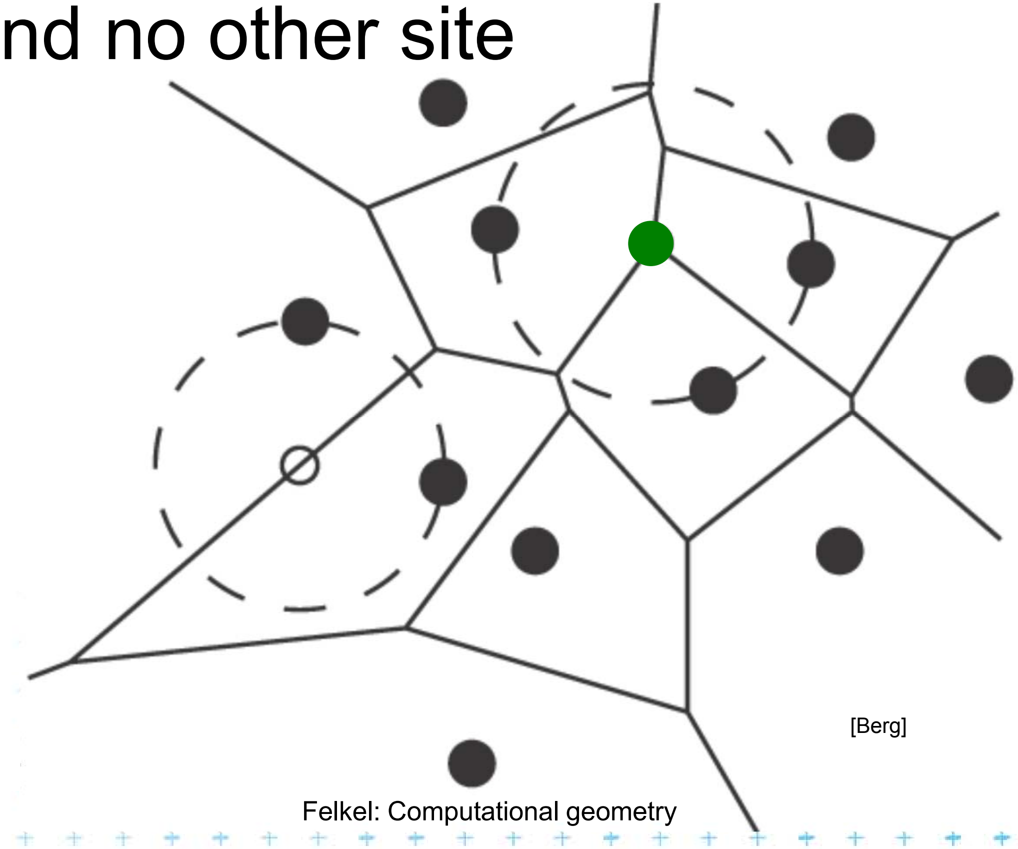
Edges, vertices and largest empty circles

Largest empty circle $C_P(q)$ with center in

1. In VD vertex q: has 3 or more sites on its boundary
2. On VD edge: contains exactly 2 sites on its boundary and no other site



[Berg]

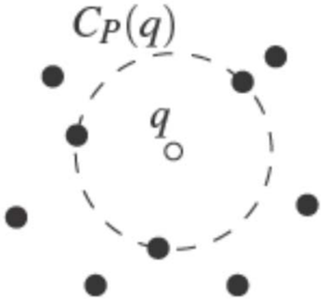


[Berg]

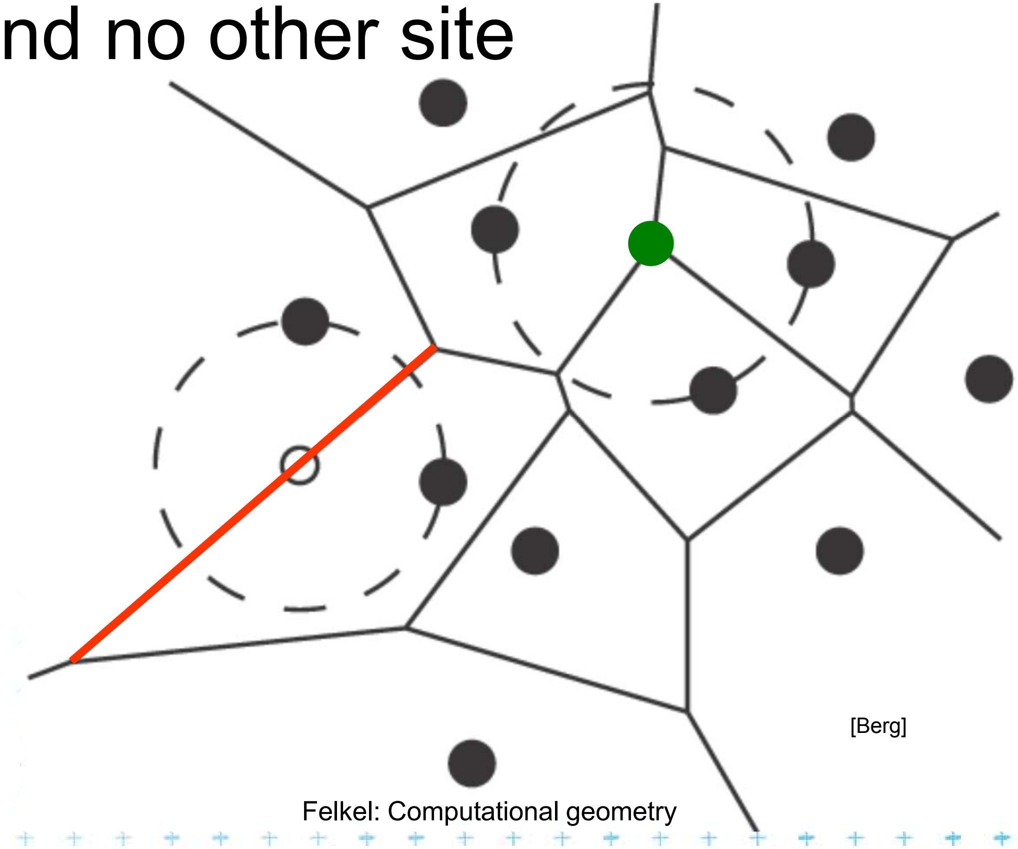
Edges, vertices and largest empty circles

Largest empty circle $C_P(q)$ with center in

1. In VD vertex q: has 3 or more sites on its boundary
2. On VD edge: contains exactly 2 sites on its boundary and no other site



[Berg]

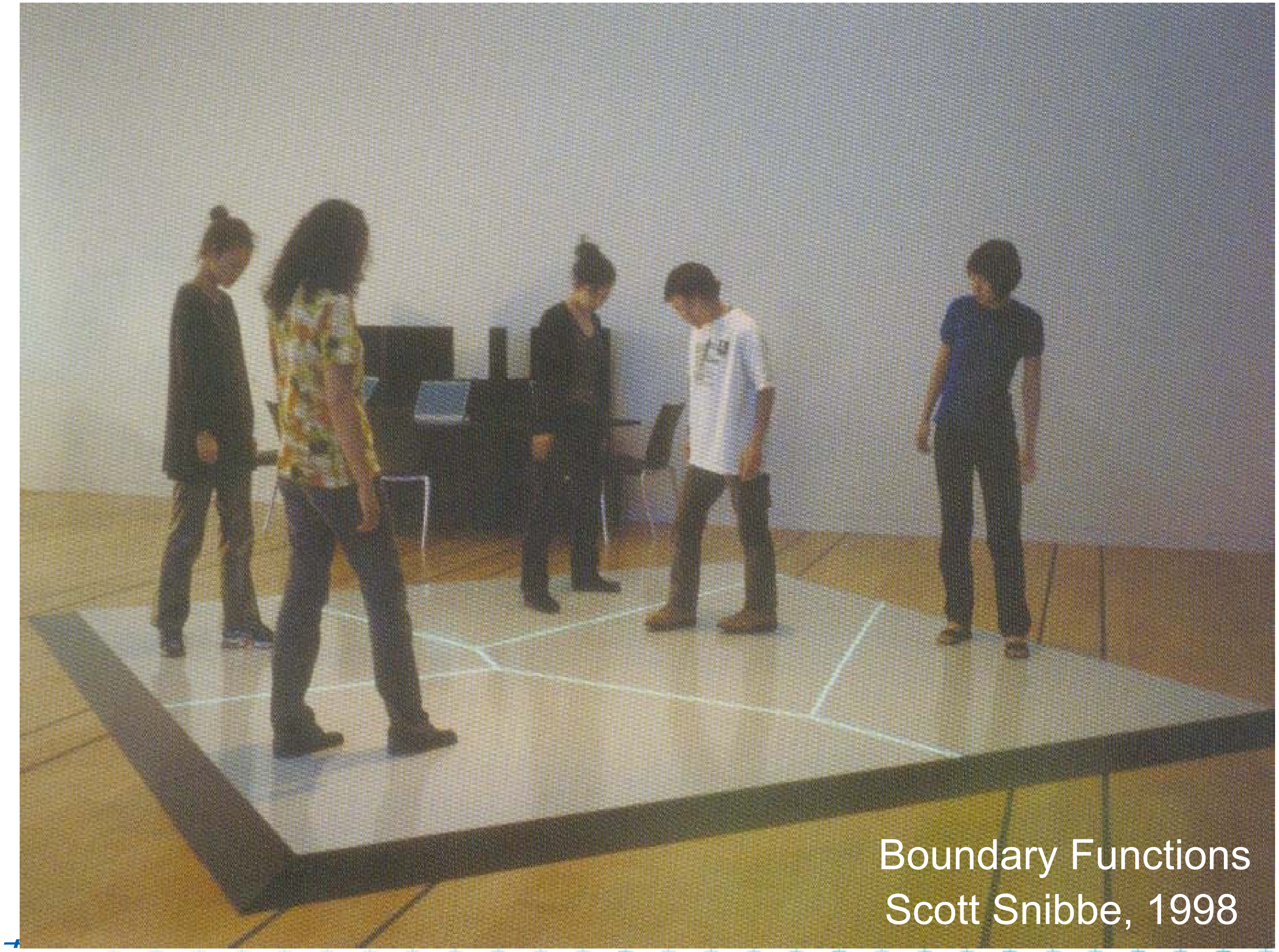


[Berg]

Some applications

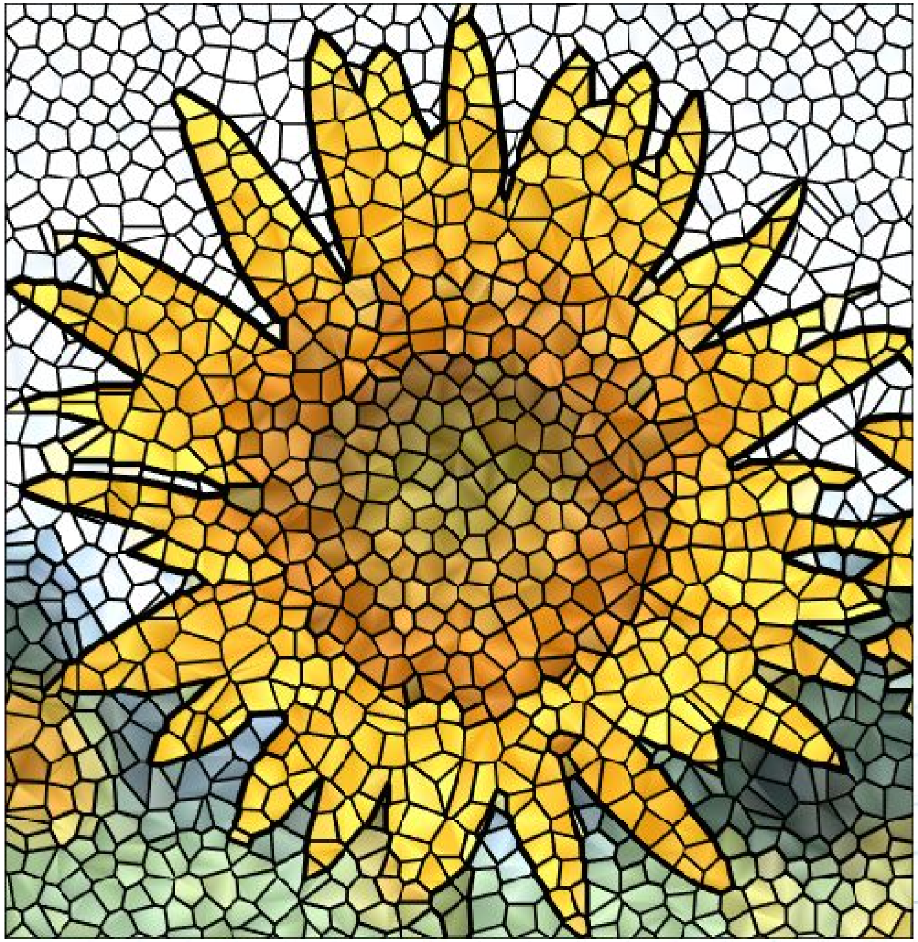
- Nearest neighbor queries in Vor(P) of points P
  - Point q ∈ P … search sites across the edges around the cell q
  - Point q ∉ P … point location queries – see Lecture 2 (the cell where point *q* falls)
- Facility location (shop or power plant)
  - Largest empty circle (better in Manhattan metric VD)
- Neighbors and Interpolation
  - Interpolate with the nearest neighbor, in 3D: surface reconstruction from points
- Art

Voronoi Art



Boundary Functions
Scott Snibbe, 1998

Voronoi Art



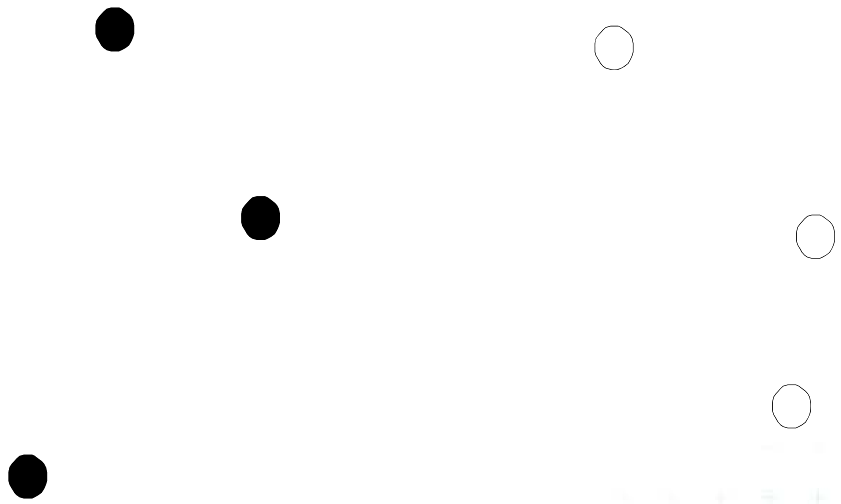
Courtesy [Gold]

Algorithms in 2D

D&C	O(n log n)
Fortune’s Sweep line	O(n log n)

Voronoi diagram (VD)

Divide and Conquer method

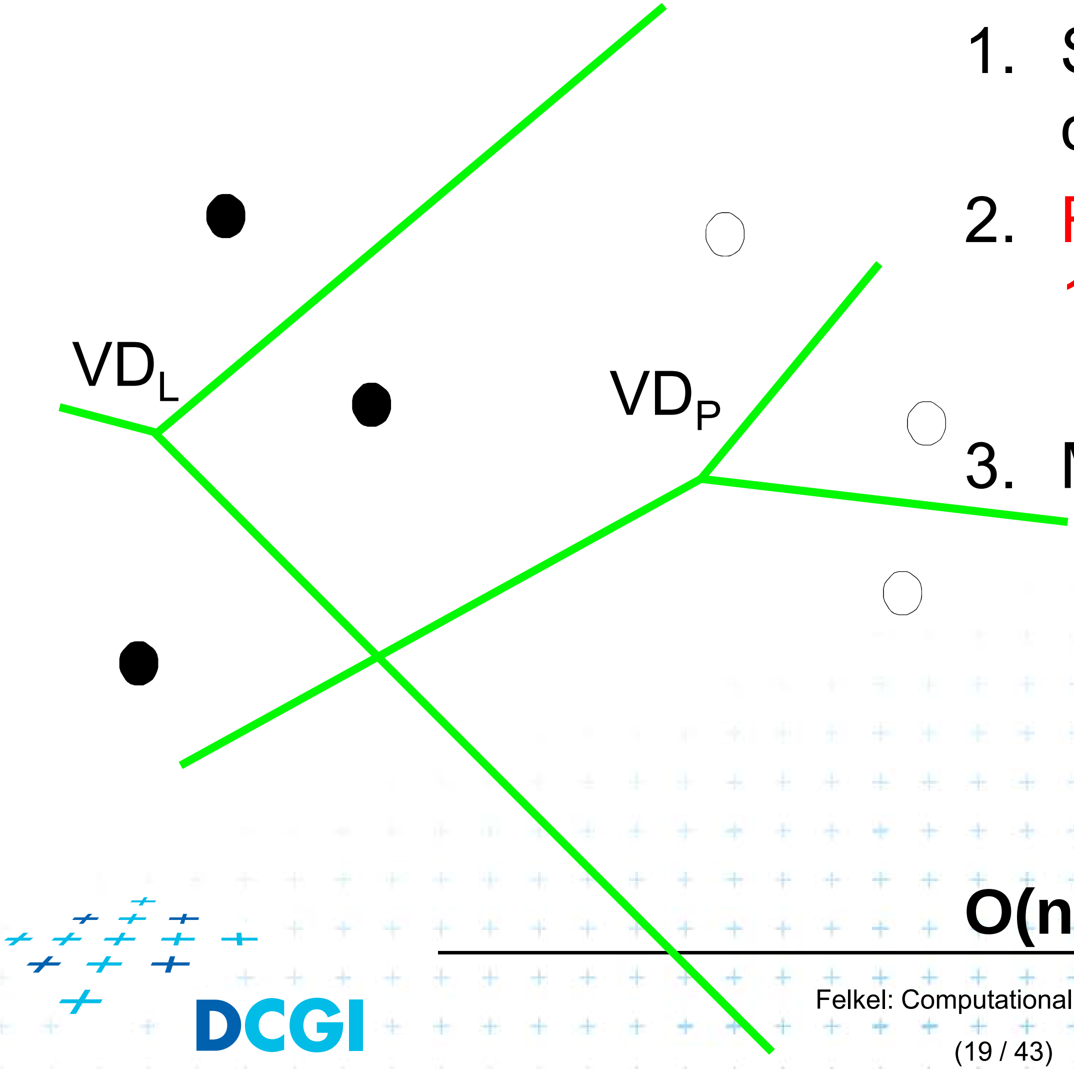


1. Split points based on x-coord into L and R
2. Recursion on L and R
   1-3 points => return
   >3 points => recursion
3. Merge VD_L and VD_R
   - monotone chain
   - trim intersected edges
   - Add new edges from the chain

O(n log n)

Voronoi diagram (VD)

Divide and Conquer method

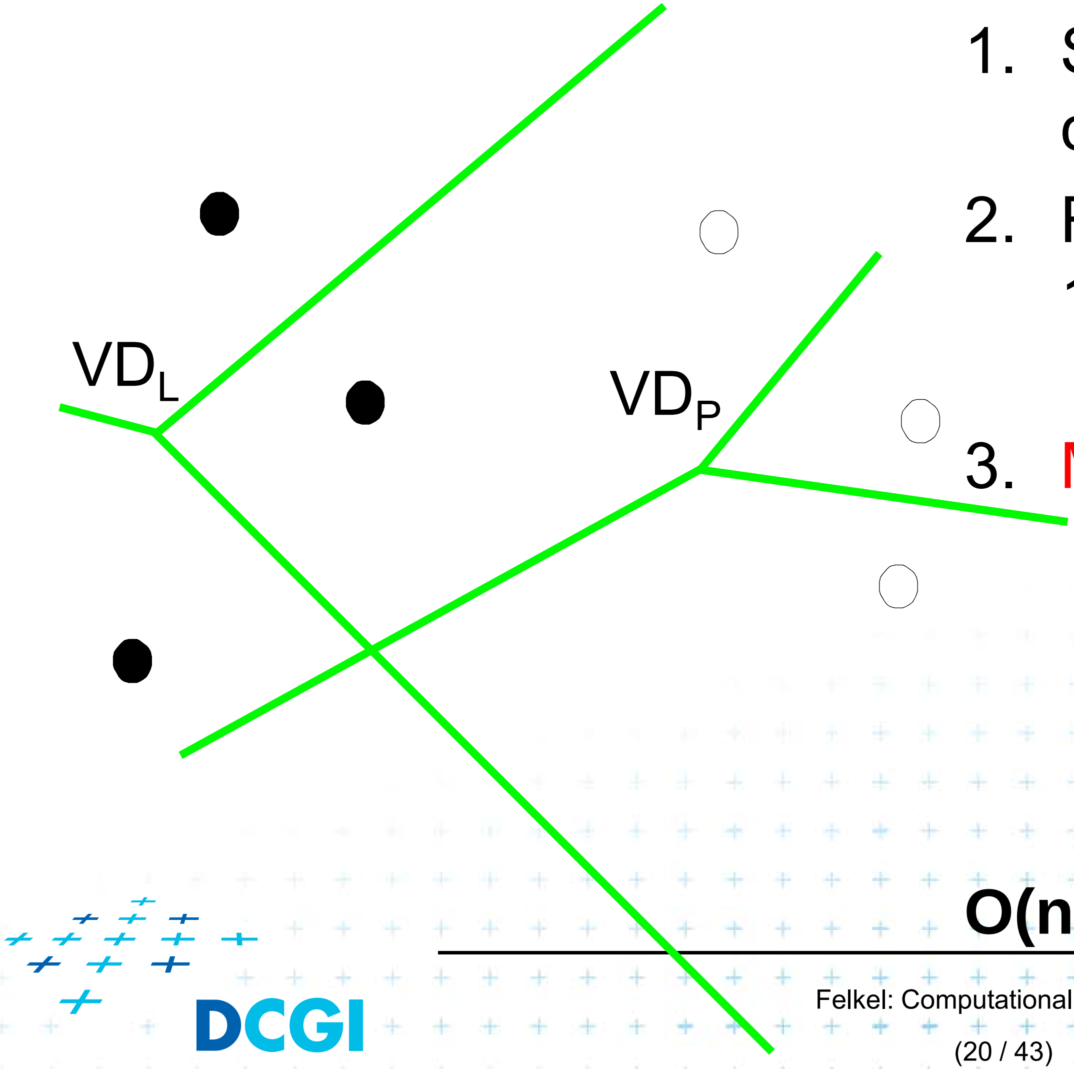


1. Split points based on x-coord into L and R
2. Recursion on L and R
   1-3 points => return
   >3 points => recursion
3. Merge VD_L and VD_R
   - monotone chain
   - trim intersected edges
   - Add new edges from the chain

O(n log n)

Voronoi diagram (VD)

Divide and Conquer method

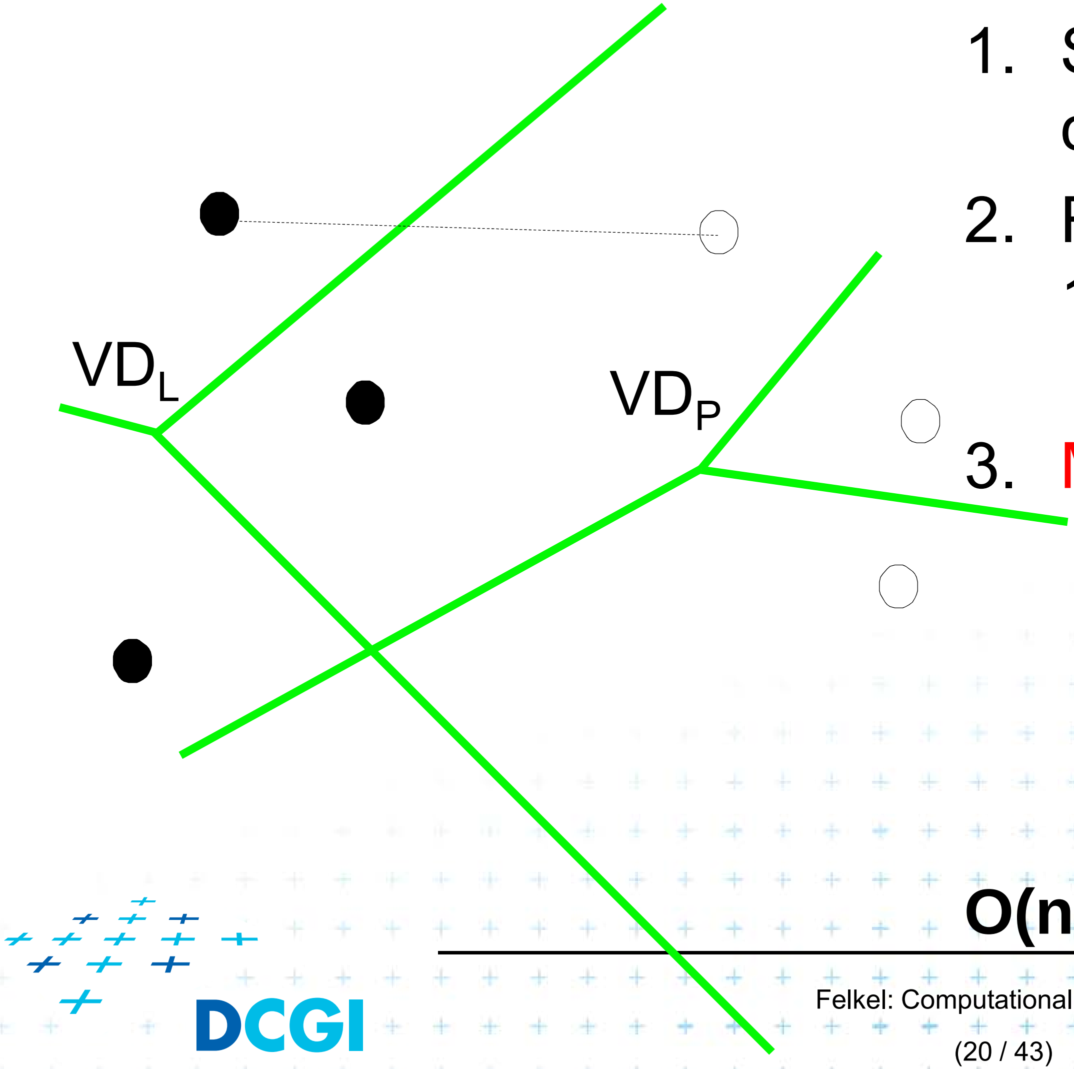


1. Split points based on x-coord into L and R
2. Recursion on L and R
   1-3 points => return
   >3 points => recursion
3. Merge VD_L and VD_R
   - monotone chain
   - trim intersected edges
   - Add new edges from the chain

O(n log n)

Voronoi diagram (VD)

Divide and Conquer method

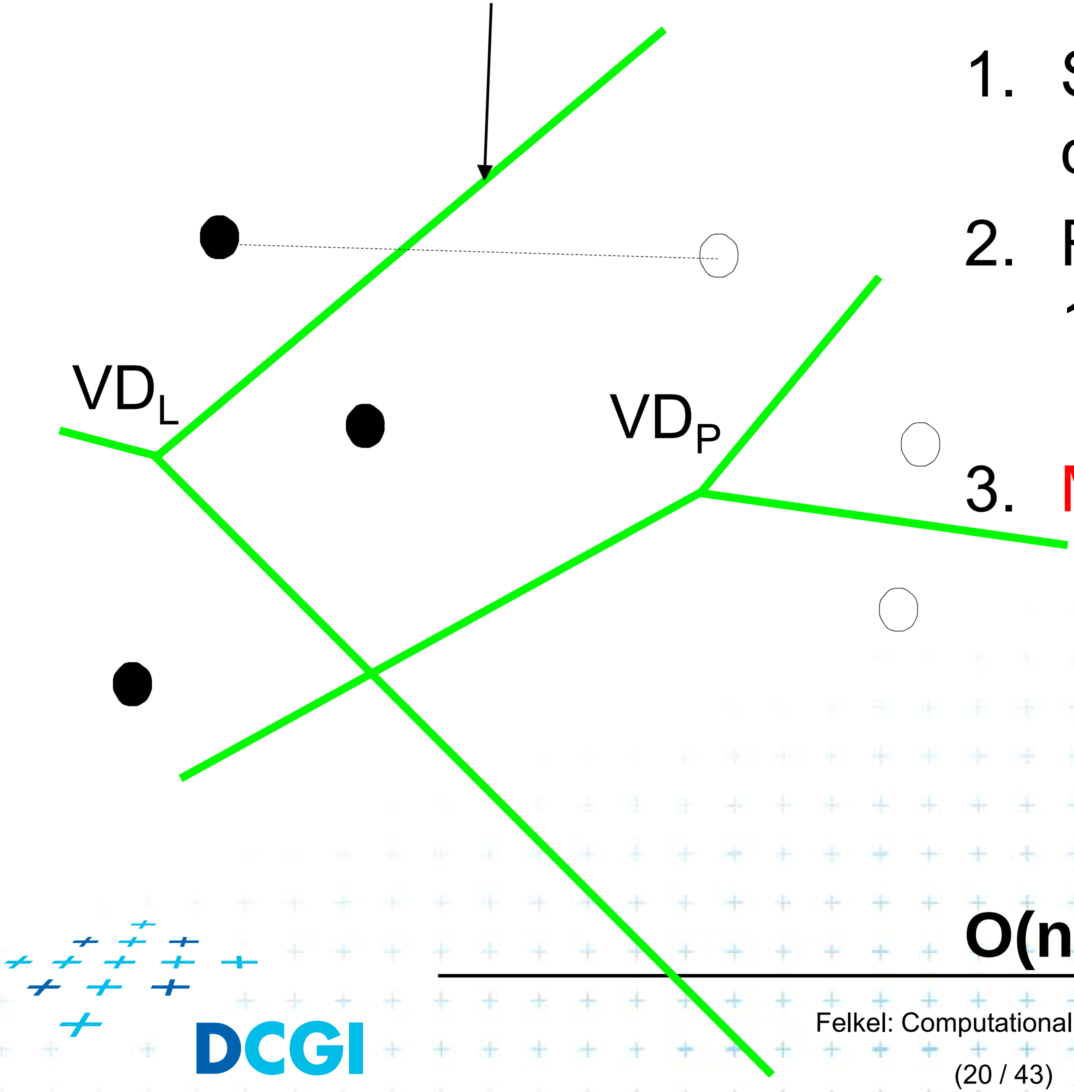


1. Split points based on x-coord into L and R
2. Recursion on L and R
   1-3 points => return
   >3 points => recursion
3. Merge VD_L and VD_R
   - monotone chain
   - trim intersected edges
   - Add new edges from the chain

O(n log n)

Voronoi diagram (VD)

Divide and Conquer method

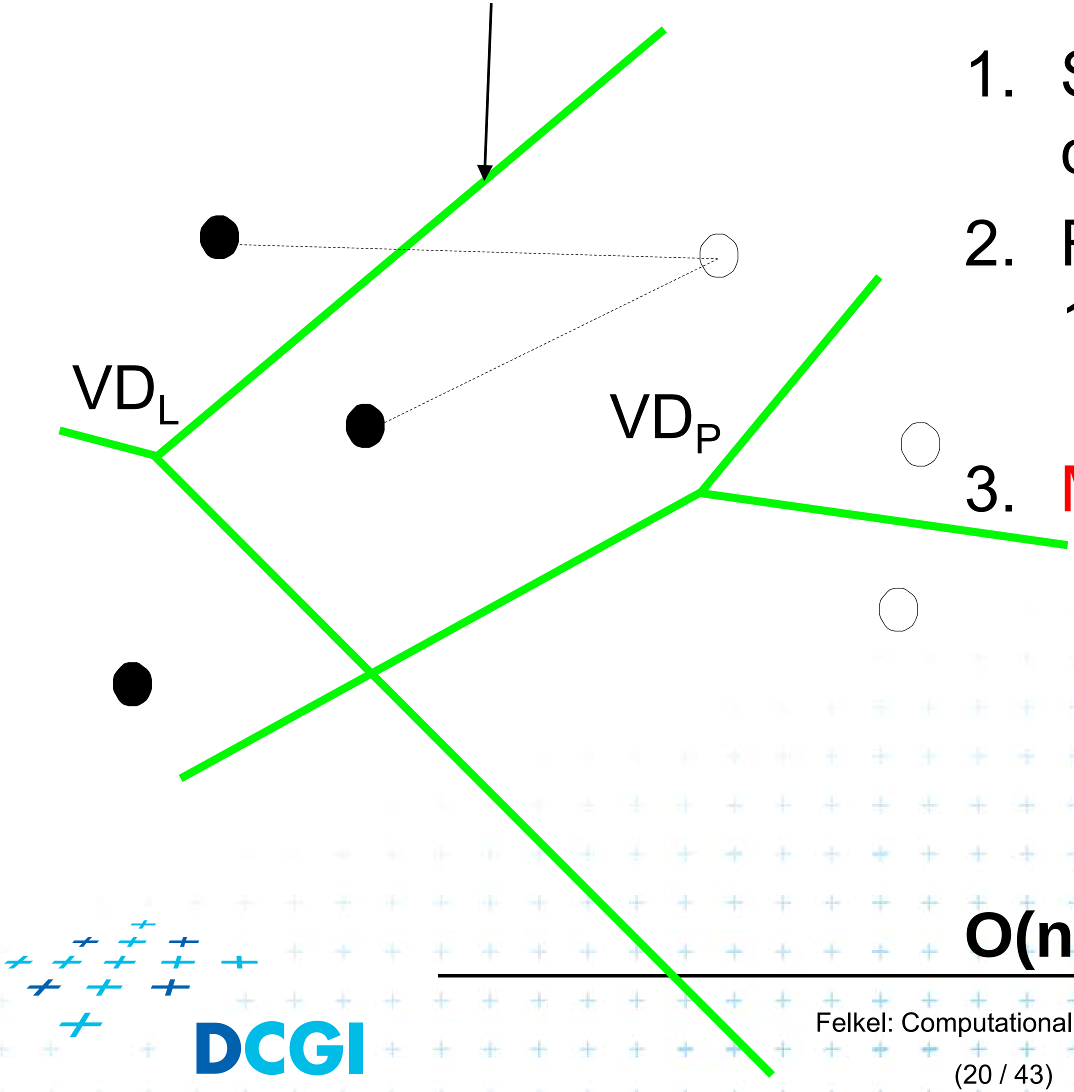


1. Split points based on x-coord into L and R
2. Recursion on L and R
   1-3 points => return
   >3 points => recursion
3. Merge VD$_L$ and VD$_R$
   - monotone chain
   - trim intersected edges
   - Add new edges from the chain

O(n log n)

Voronoi diagram (VD)

Divide and Conquer method

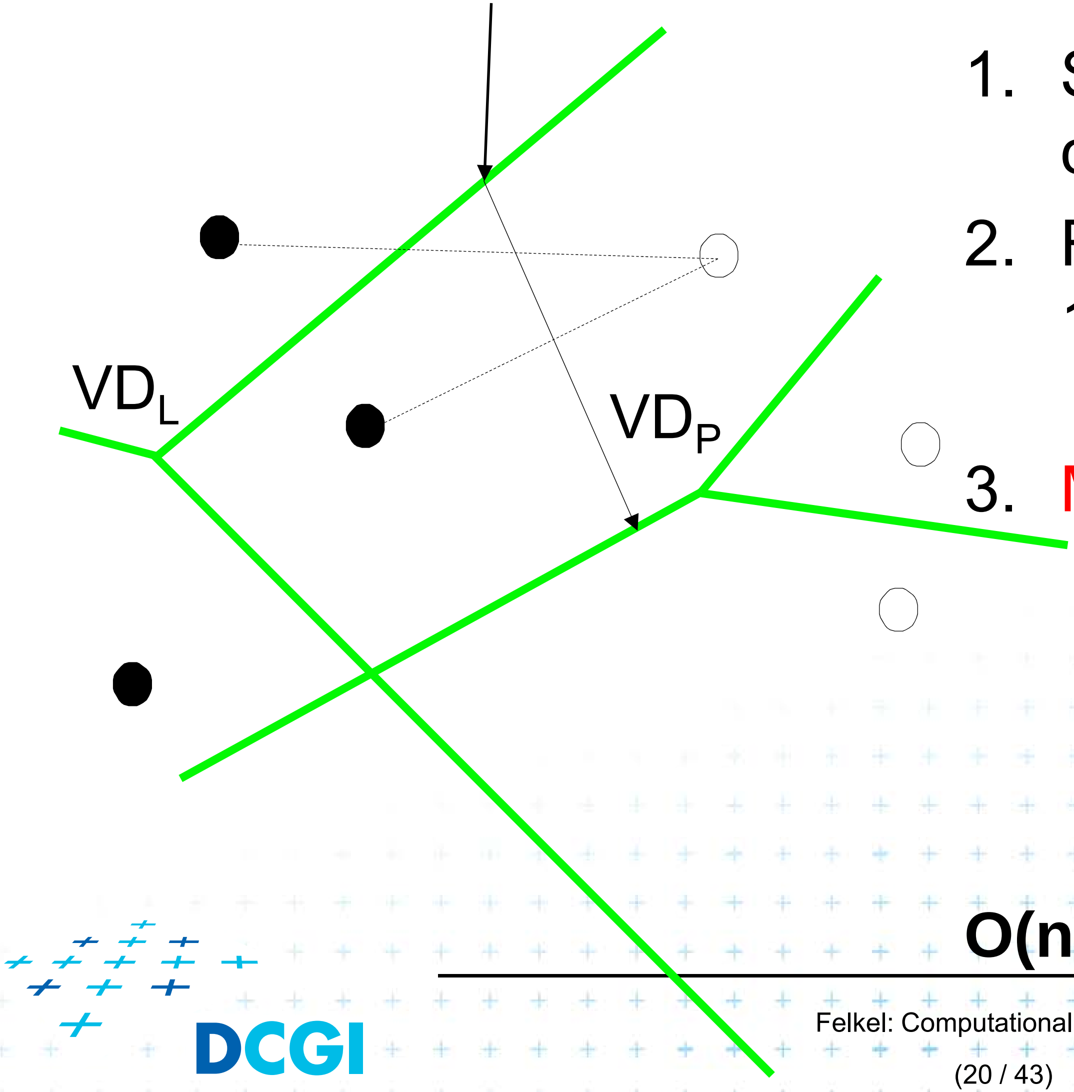


1. Split points based on x-coord into L and R
2. Recursion on L and R
   1-3 points => return
   >3 points => recursion
3. Merge VD_L and VD_R
   - monotone chain
   - trim intersected edges
   - Add new edges from the chain

O(n log n)

Voronoi diagram (VD)

Divide and Conquer method

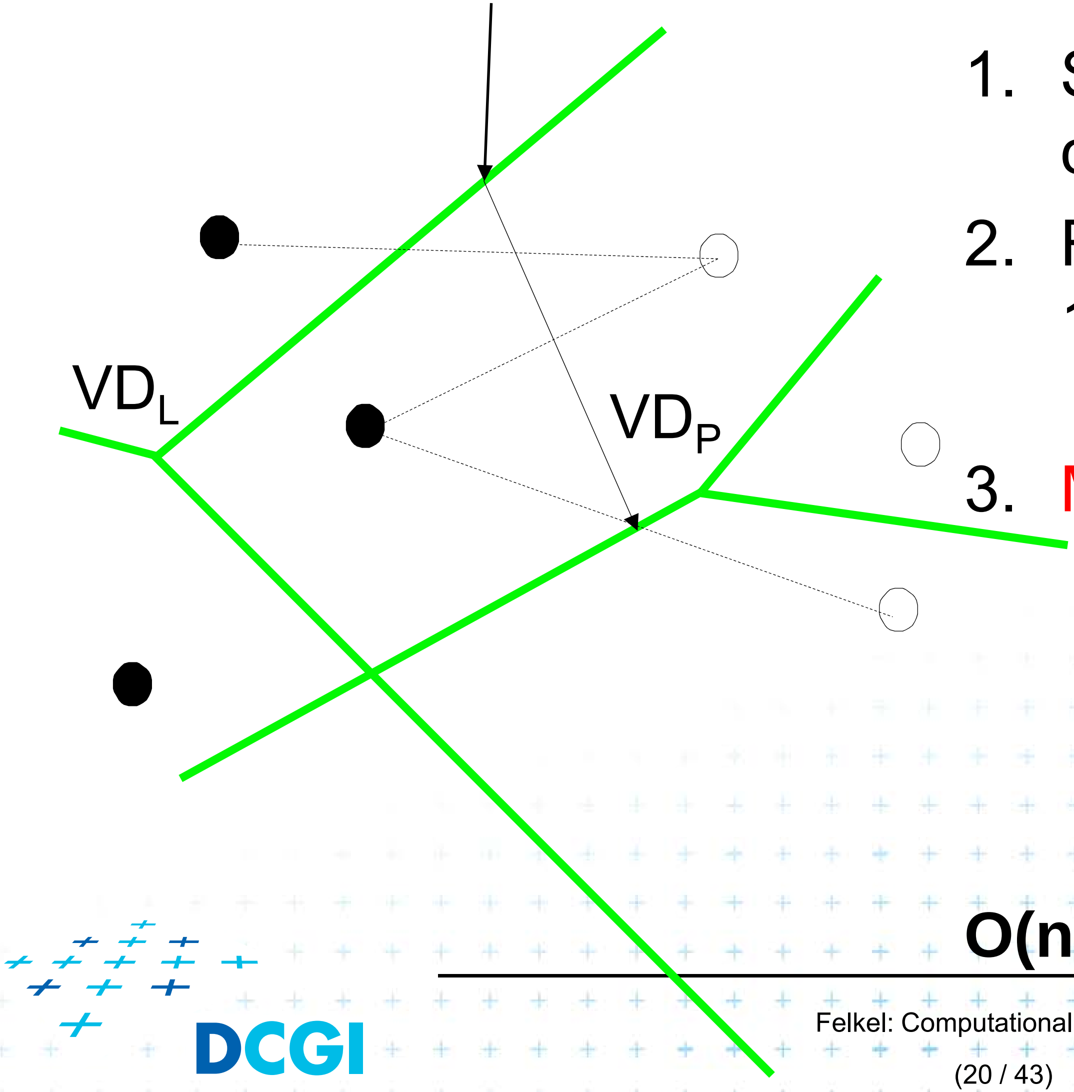


1. Split points based on x-coord into L and R
2. Recursion on L and R
   1-3 points => return
   >3 points => recursion
3. Merge VD_L and VD_R
   - monotone chain
   - trim intersected edges
   - Add new edges from the chain

O(n log n)

Voronoi diagram (VD)

Divide and Conquer method

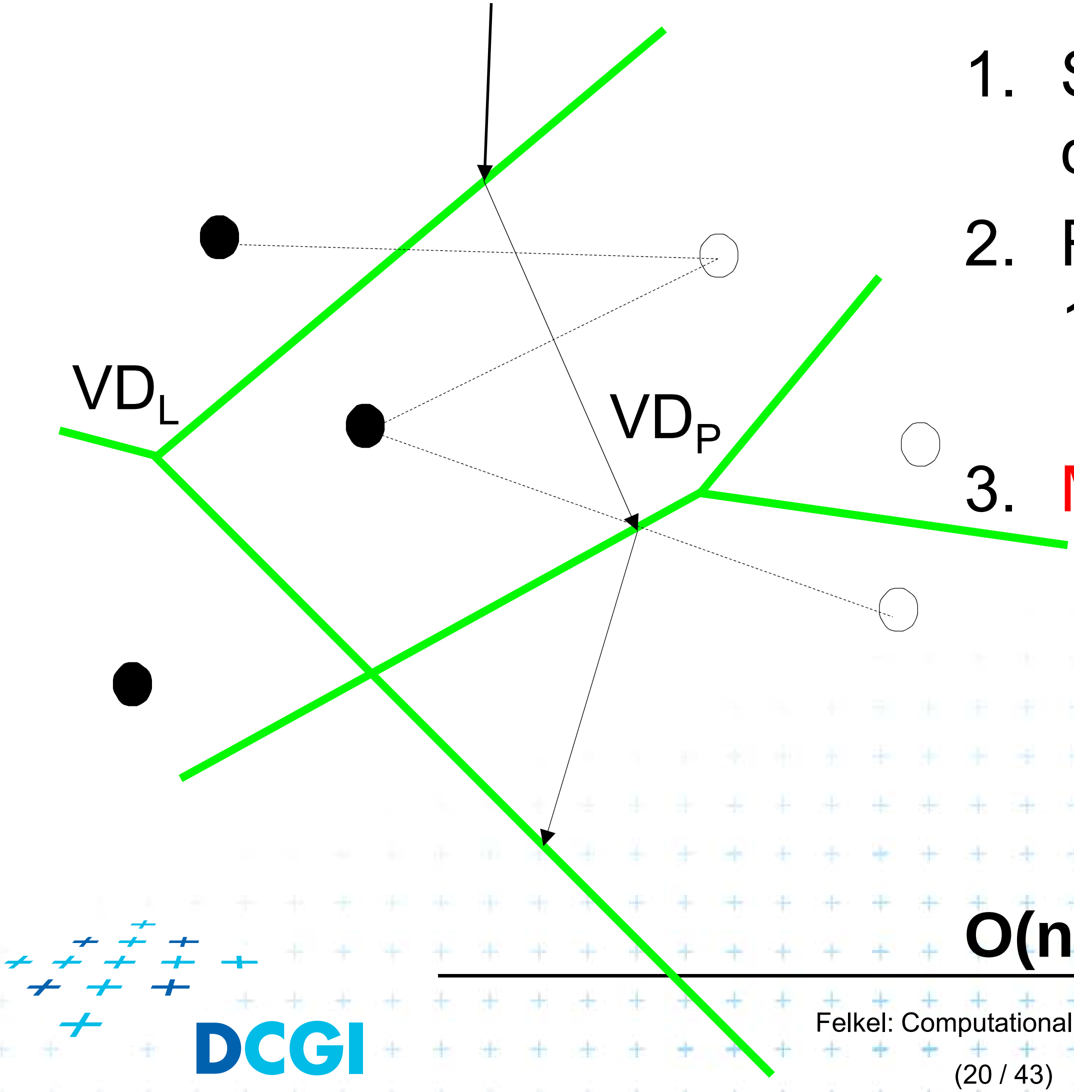


1. Split points based on x-coord into L and R
2. Recursion on L and R
   1-3 points => return
   >3 points => recursion
3. Merge VD_L and VD_R
   - monotone chain
   - trim intersected edges
   - Add new edges from the chain

O(n log n)

Voronoi diagram (VD)

Divide and Conquer method

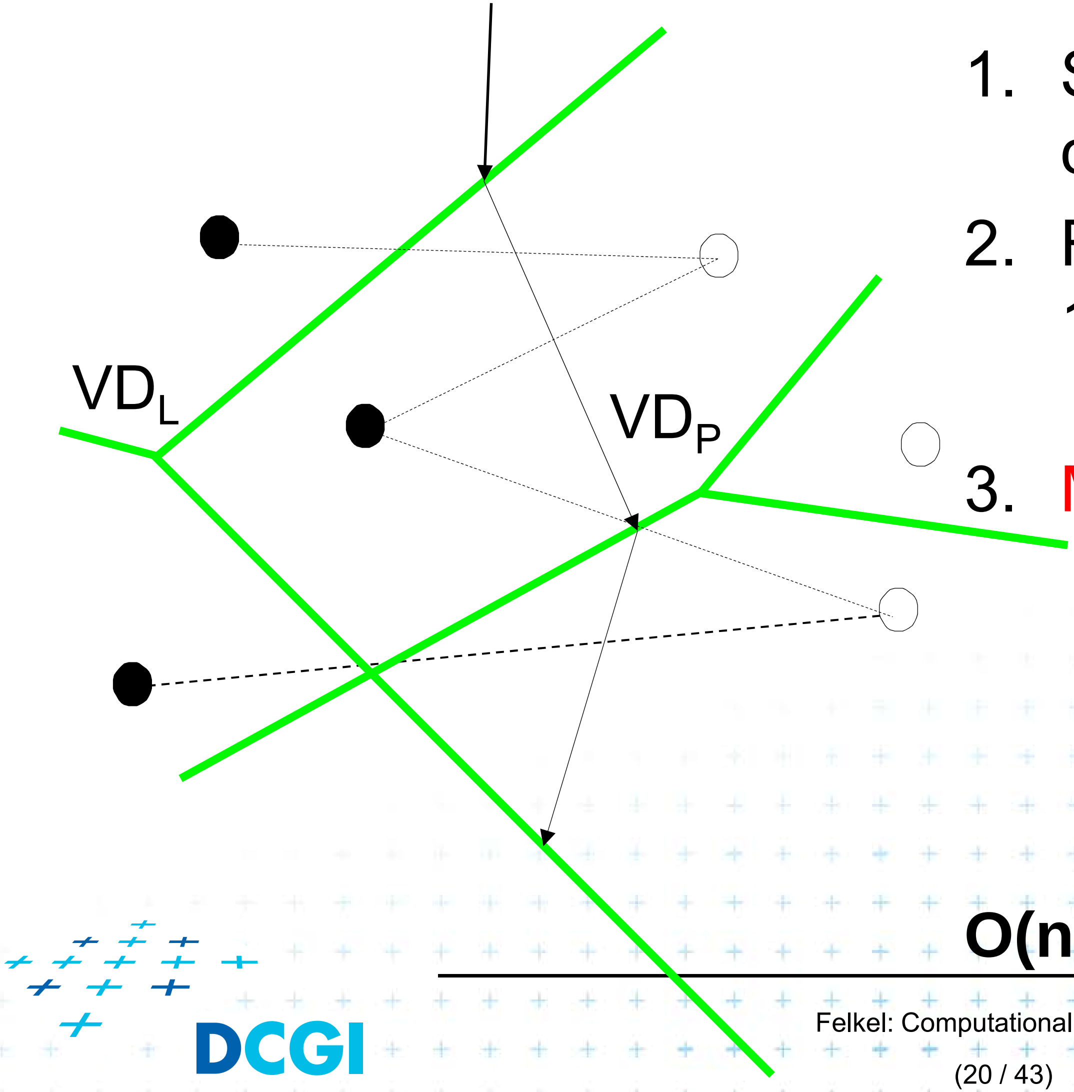


1. Split points based on x-coord into L and R
2. Recursion on L and R
   1-3 points => return
   >3 points => recursion
3. Merge VD_L and VD_R
   - monotone chain
   - trim intersected edges
   - Add new edges from the chain

O(n log n)

Voronoi diagram (VD)

Divide and Conquer method

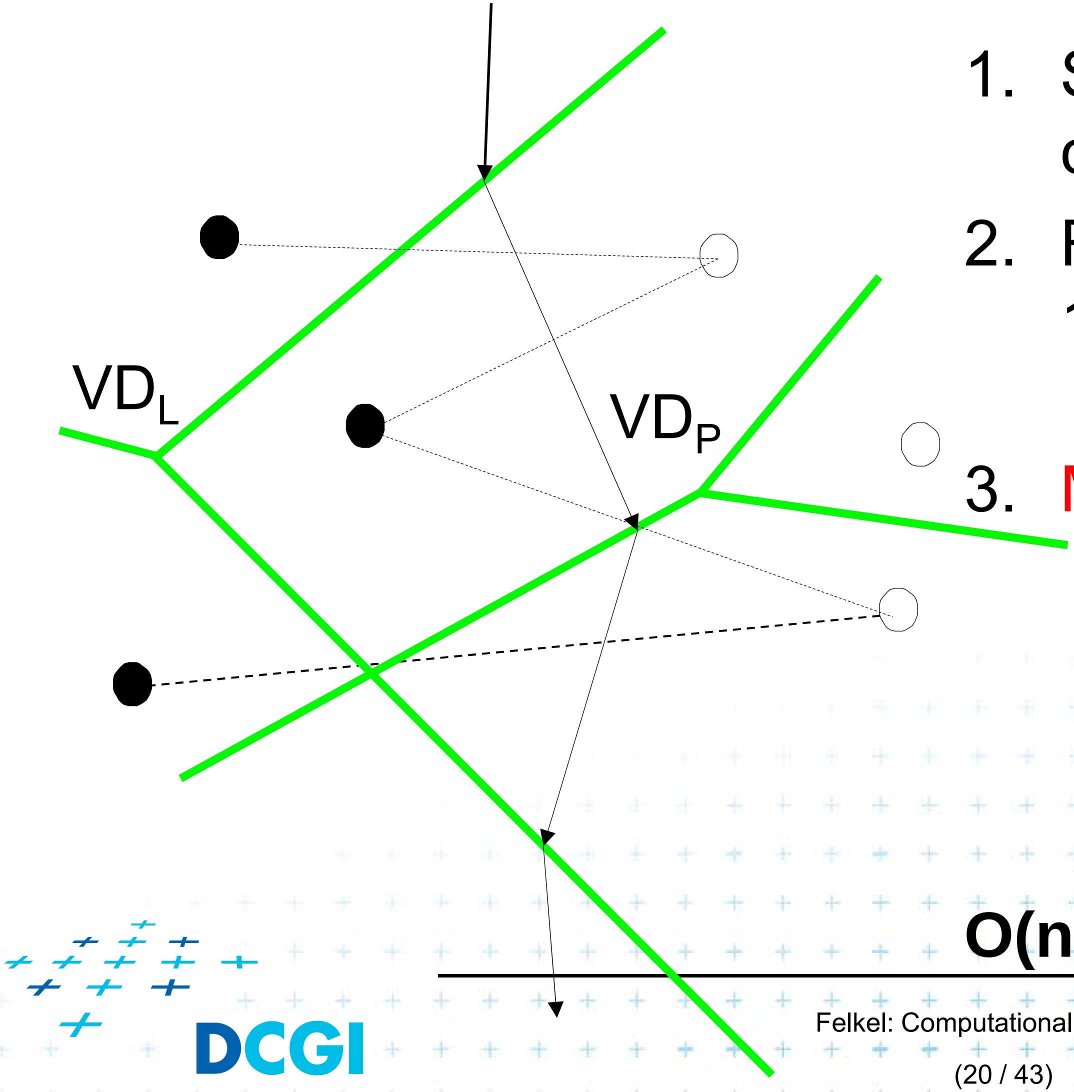


1. Split points based on x-coord into L and R
2. Recursion on L and R
   1-3 points => return
   >3 points => recursion
3. Merge VD_L and VD_R
   - monotone chain
   - trim intersected edges
   - Add new edges from the chain

O(n log n)

Voronoi diagram (VD)

Divide and Conquer method

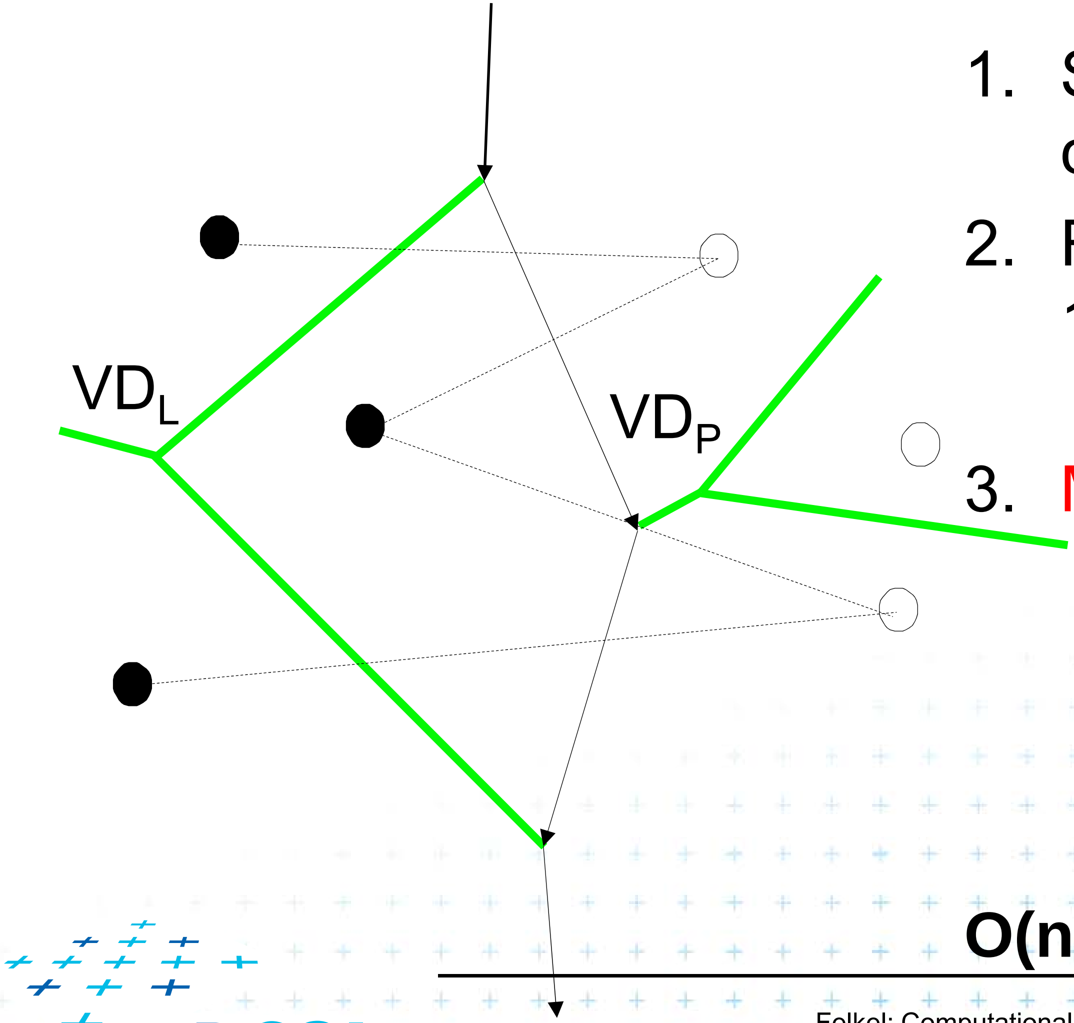


1. Split points based on x-coord into L and R
2. Recursion on L and R
   1-3 points => return
   >3 points => recursion
3. Merge VD_L and VD_R
   - monotone chain
   - trim intersected edges
   - Add new edges from the chain

O(n log n)

Voronoi diagram (VD)

Divide and Conquer method

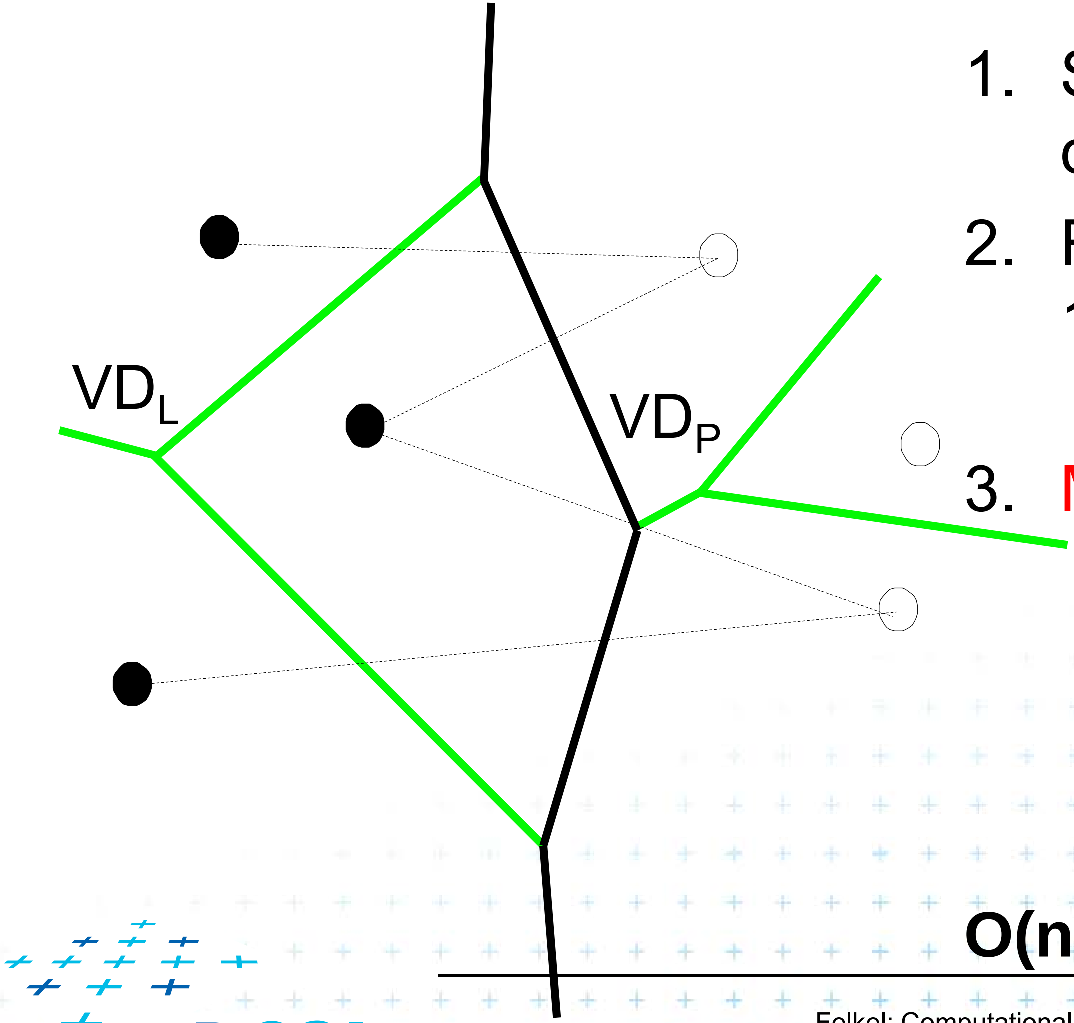


1. Split points based on x-coord into L and R
2. Recursion on L and R
   1-3 points => return
   >3 points => recursion
3. Merge VD_L and VD_R
   - monotone chain
   - trim intersected edges
   - Add new edges from the chain

O(n log n)

Voronoi diagram (VD)

Divide and Conquer method

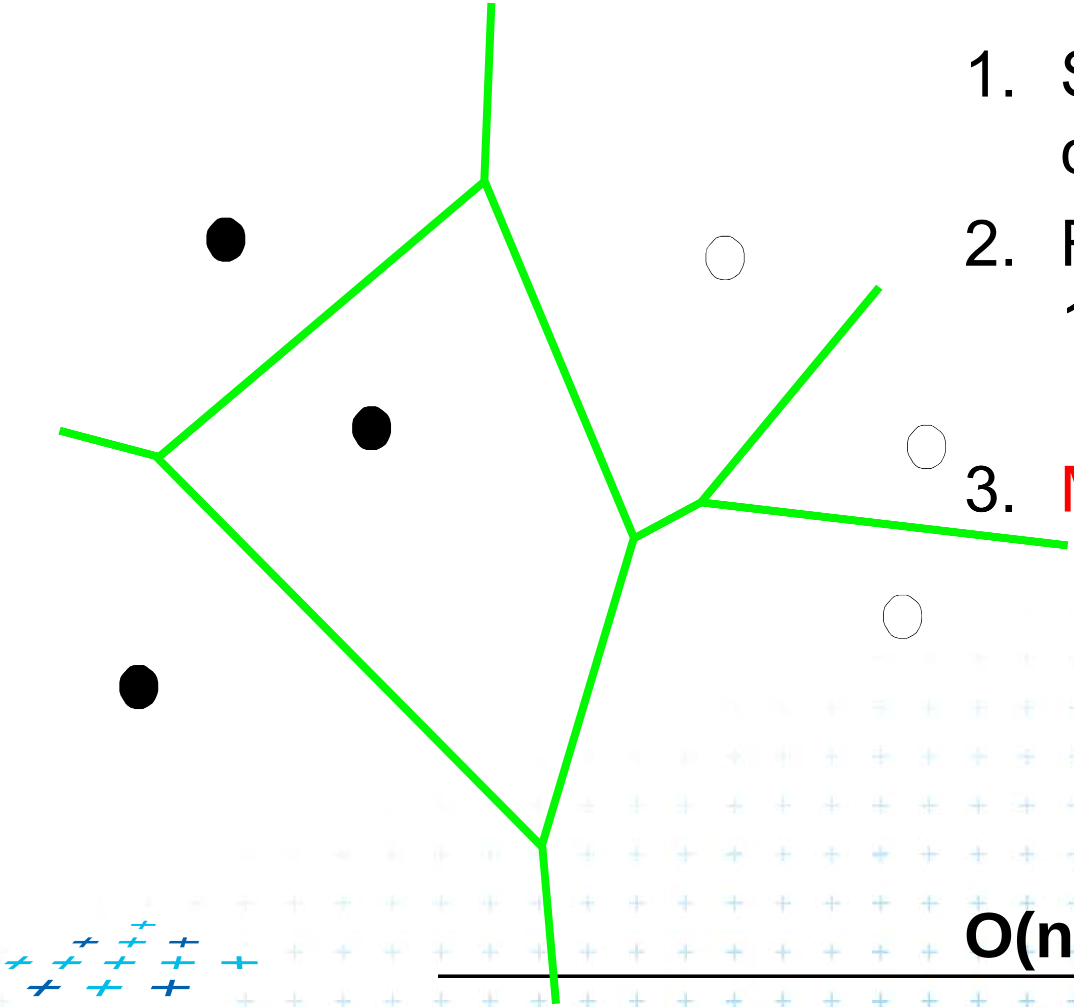


1. Split points based on x-coord into L and R
2. Recursion on L and R
   1-3 points => return
   >3 points => recursion
3. Merge VD_L and VD_R
   - monotone chain
   - trim intersected edges
   - Add new edges from the chain

O(n log n)

Voronoi diagram (VD)

Divide and Conquer method

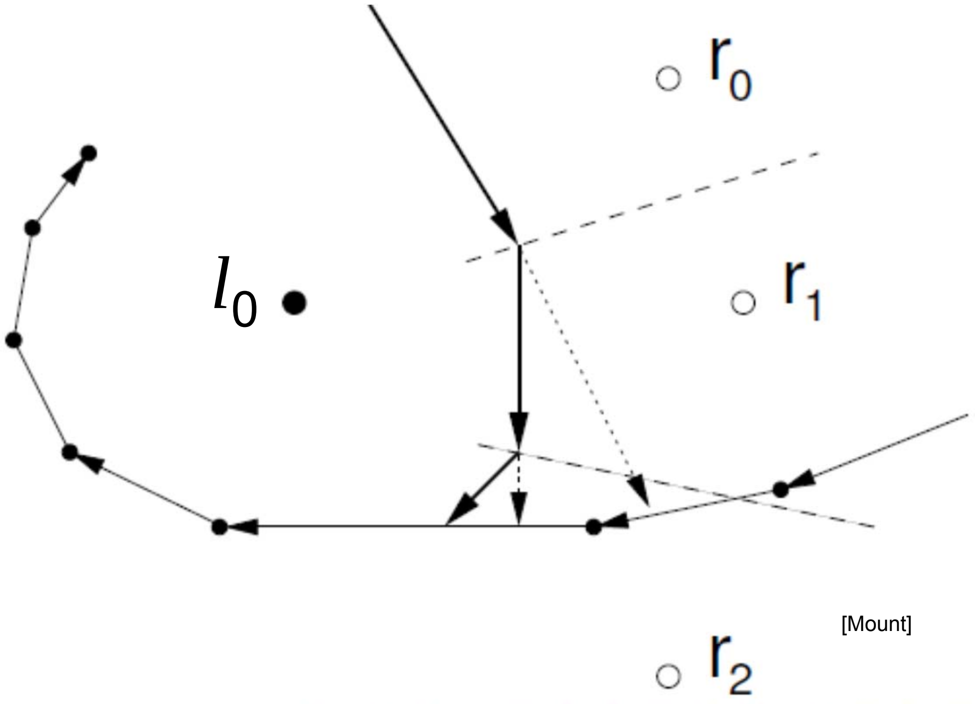


1. Split points based on x-coord into L and R
2. Recursion on L and R
   1-3 points => return
   >3 points => recursion
3. Merge VD_L and VD_R
   - monotone chain
   - trim intersected edges
   - Add new edges from the chain

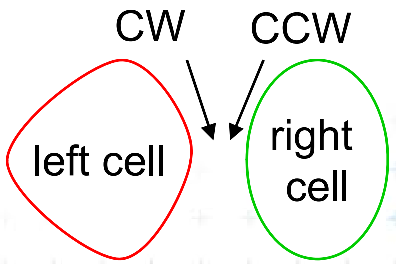
O(n log n)

Monotone chain search in O(n)

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested ~once)
- In the left cell l_i continue CW, in the right cell r_i go CCW
- Image shows CW search on cell l_0 and CCW on cells r_i:

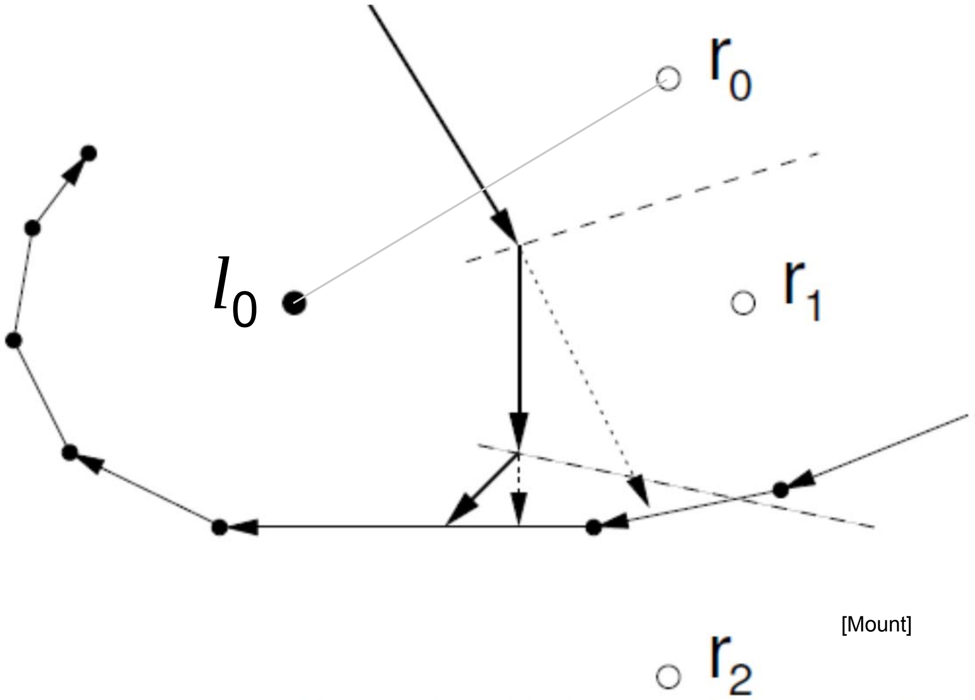


[Mount]

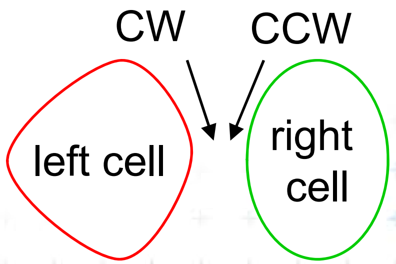


Monotone chain search in O(n)

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested ~once)
- In the left cell l_i continue CW, in the right cell r_i go CCW
- Image shows CW search on cell l_0 and CCW on cells r_i :

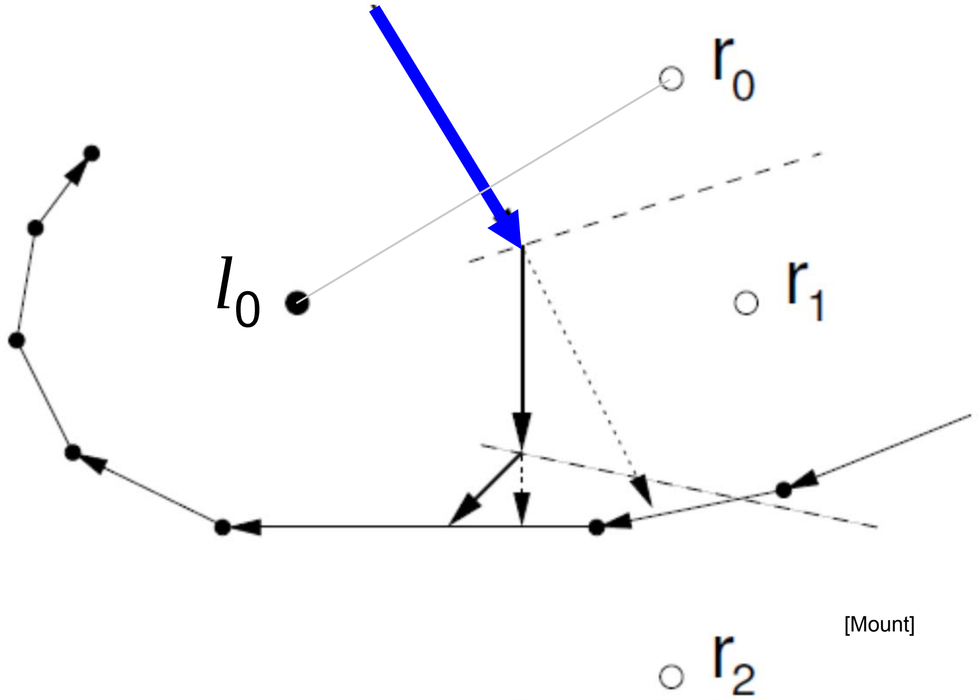


[Mount]

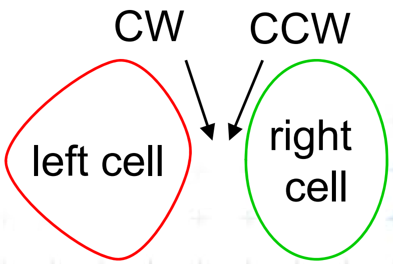


Monotone chain search in O(n)

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested ~once)
- In the left cell l_i continue CW, in the right cell r_i go CCW
- Image shows CW search on cell l_0 and CCW on cells r_i:

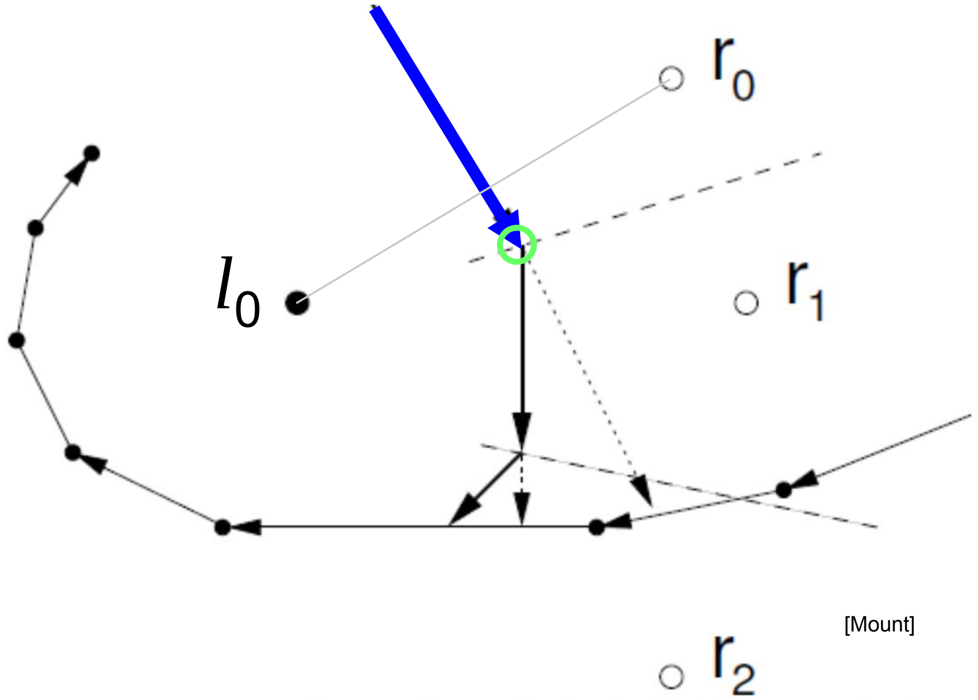


[Mount]

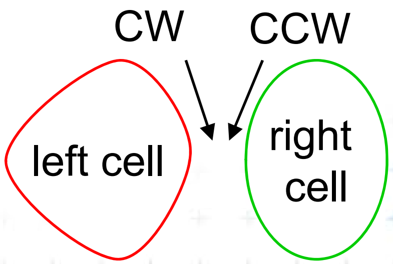


Monotone chain search in O(n)

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested ~once)
- In the left cell l_i continue CW, in the right cell r_i go CCW
- Image shows CW search on cell l_0 and CCW on cells r_i:

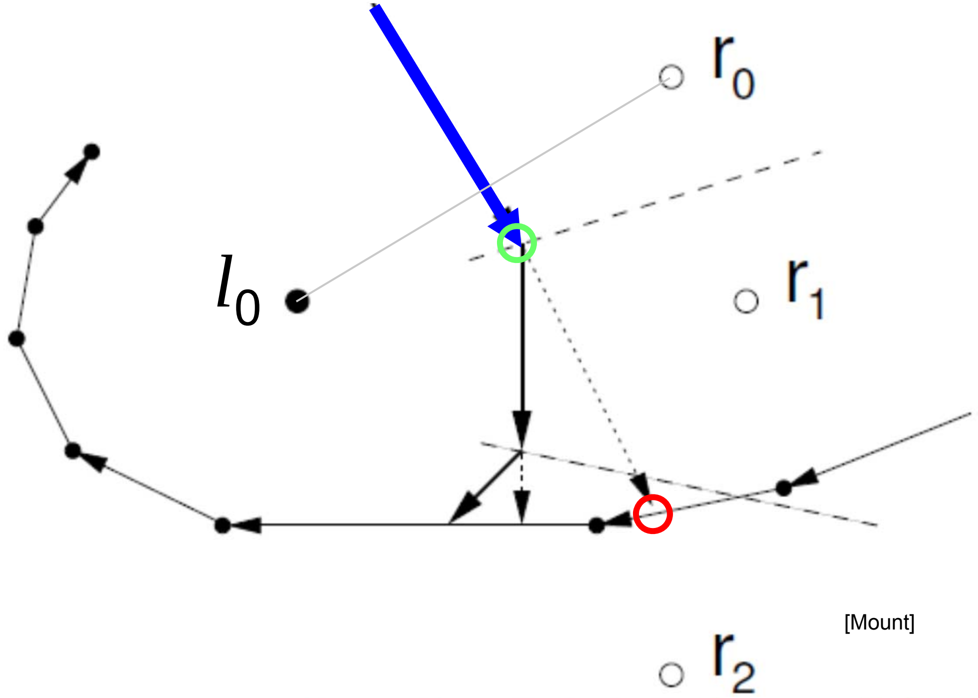


[Mount]

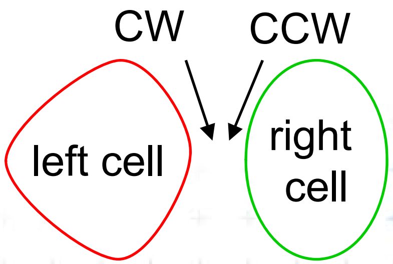


Monotone chain search in O(n)

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested ~once)
- In the left cell l_i continue CW, in the right cell r_i go CCW
- Image shows CW search on cell l_0 and CCW on cells r_i:

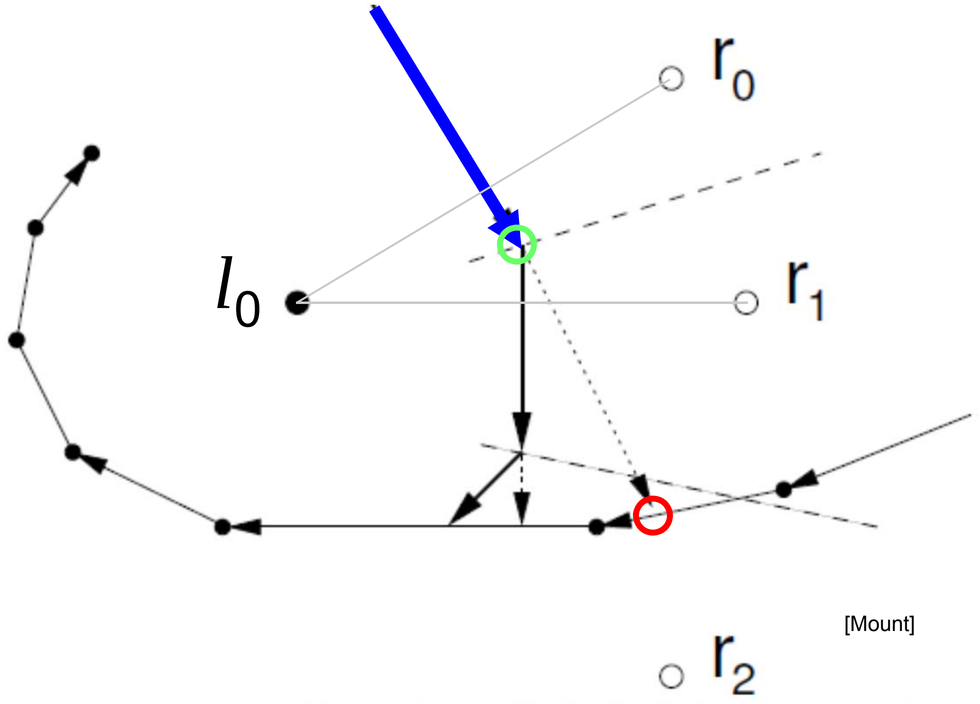


[Mount]

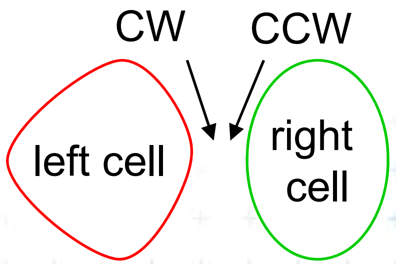


Monotone chain search in O(n)

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested ~once)
- In the left cell l_i continue CW, in the right cell r_i go CCW
- Image shows CW search on cell l_0 and CCW on cells r_i:

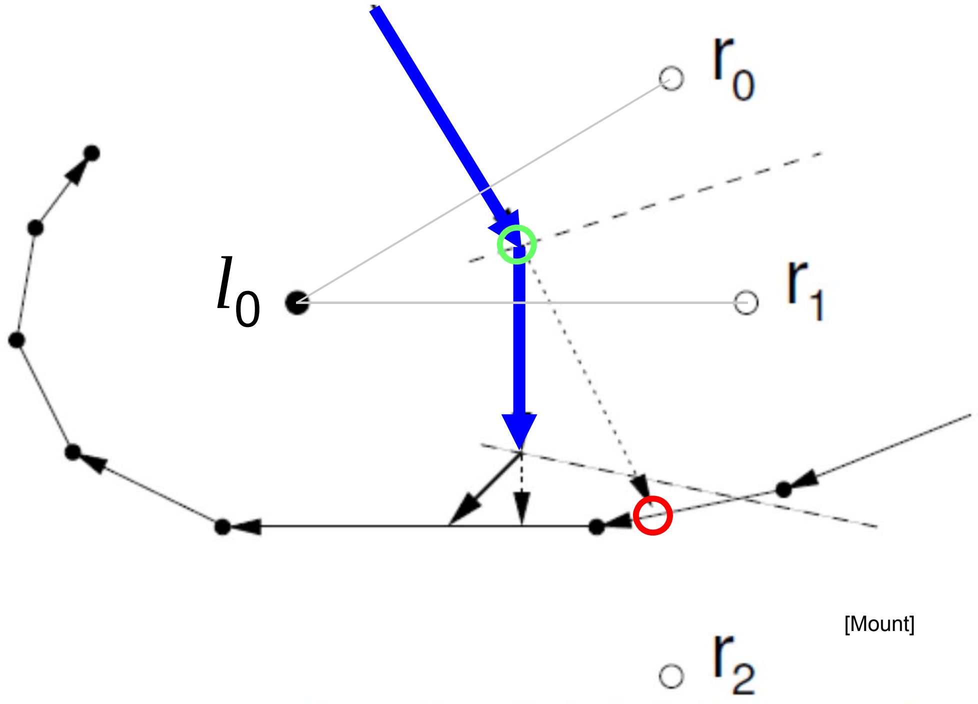


[Mount]

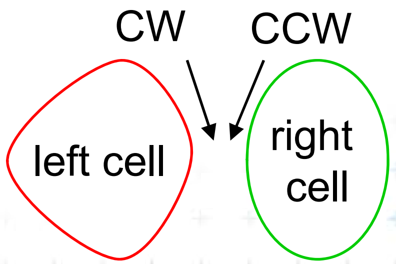


Monotone chain search in O(n)

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested ~once)
- In the left cell l_i continue CW, in the right cell r_i go CCW
- Image shows CW search on cell l_0 and CCW on cells r_i:

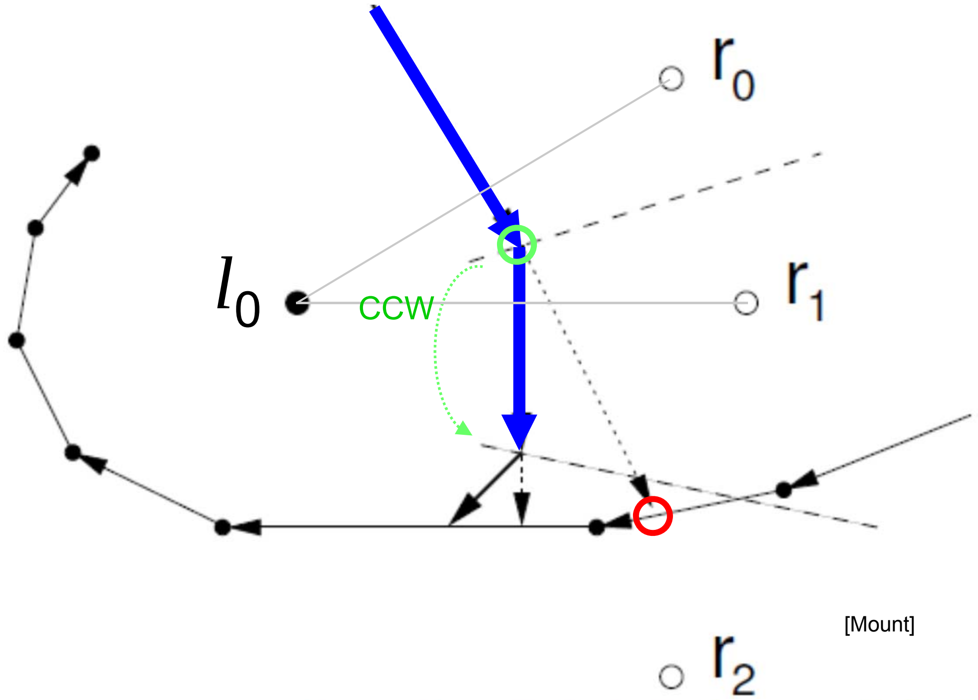


[Mount]

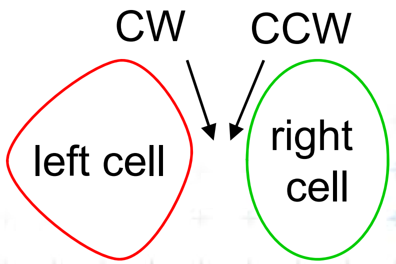


Monotone chain search in O(n)

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested ~once)
- In the left cell l_i continue CW, in the right cell r_i go CCW
- Image shows CW search on cell l_0 and CCW on cells r_i :

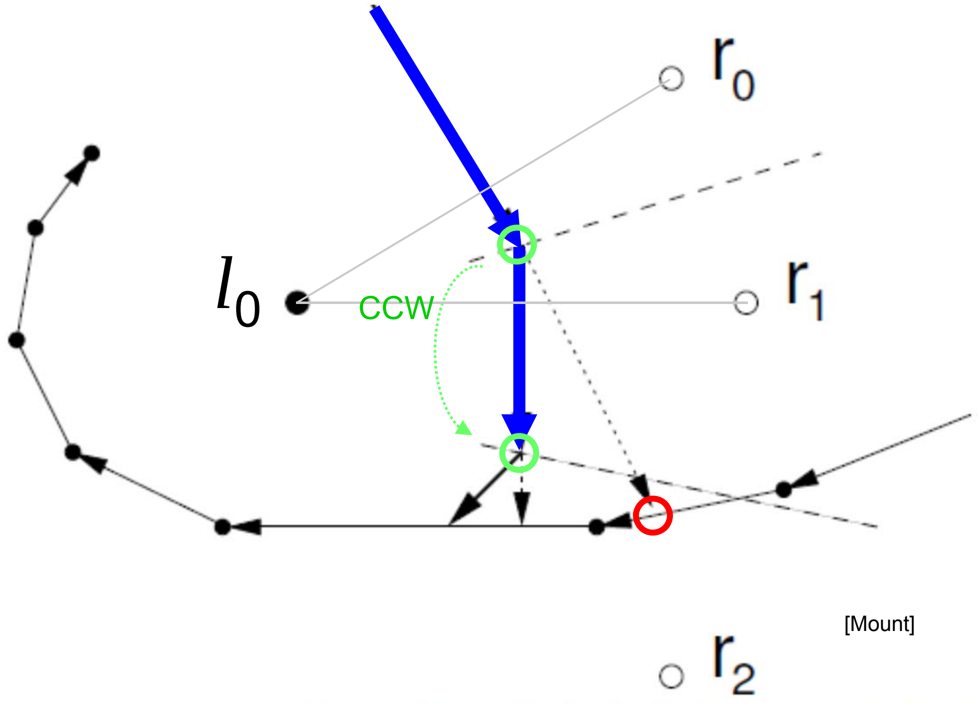


[Mount]

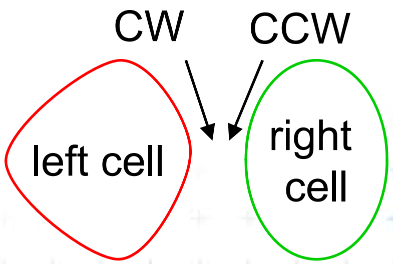


Monotone chain search in O(n)

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested ~once)
- In the left cell l_i continue CW, in the right cell r_i go CCW
- Image shows CW search on cell l_0 and CCW on cells r_i:

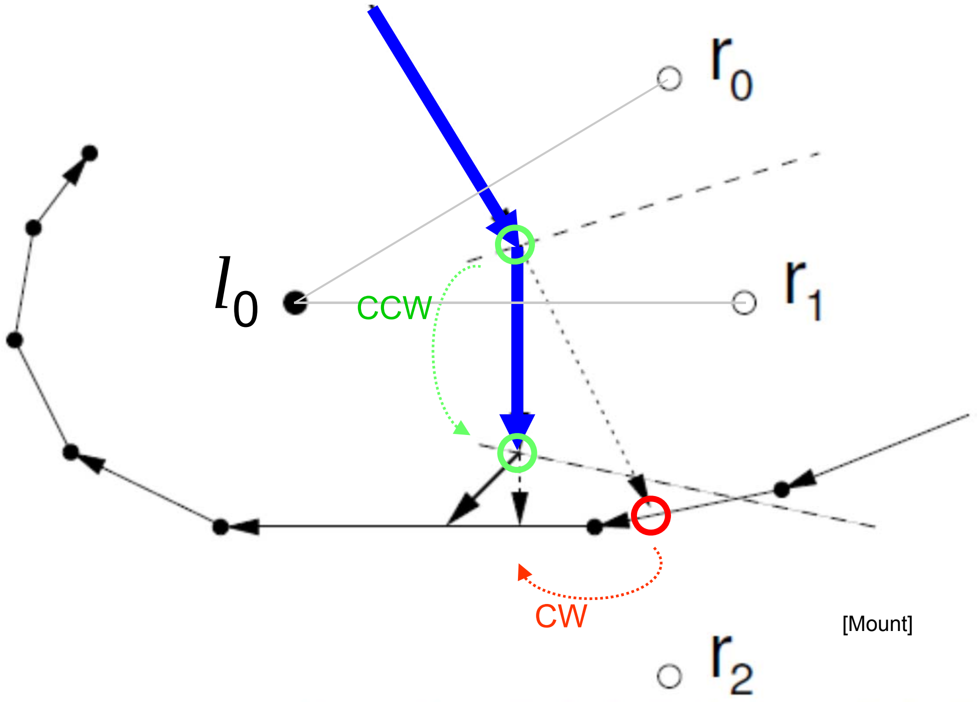


[Mount]

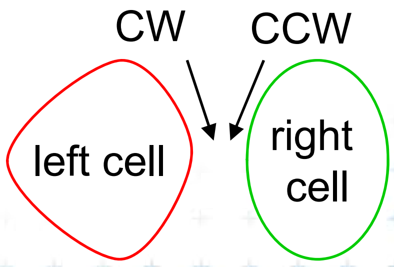


Monotone chain search in O(n)

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested ~once)
- In the left cell l_i continue CW, in the right cell r_i go CCW
- Image shows CW search on cell l_0 and CCW on cells r_i:

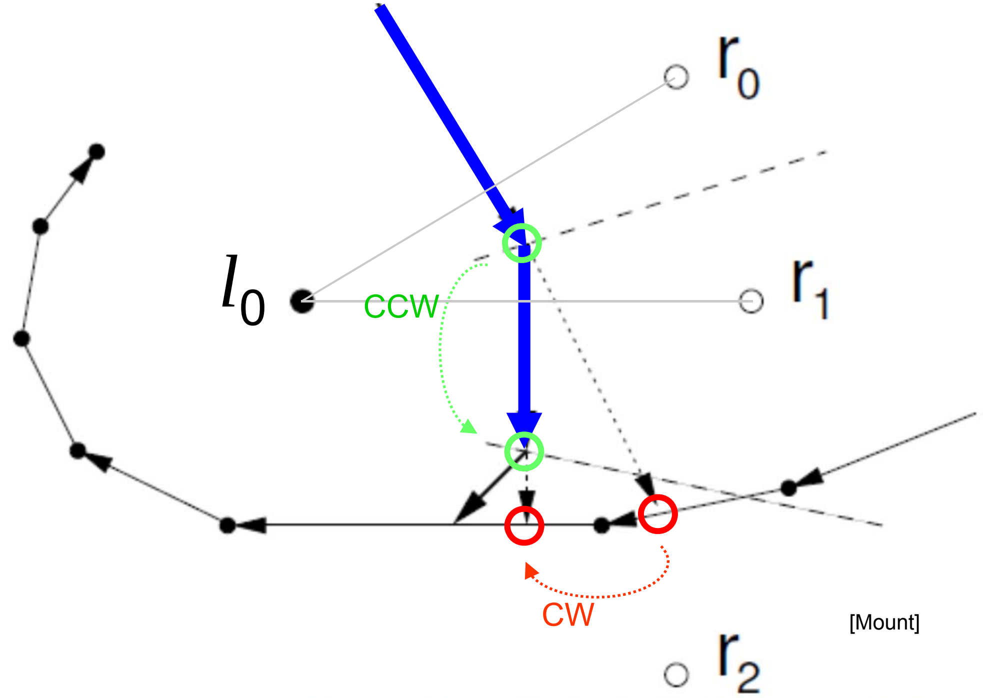


[Mount]

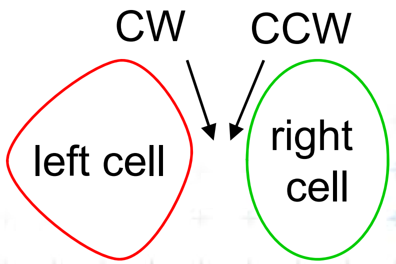


Monotone chain search in O(n)

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested ~once)
- In the left cell l_i continue CW, in the right cell r_i go CCW
- Image shows CW search on cell l_0 and CCW on cells r_i:

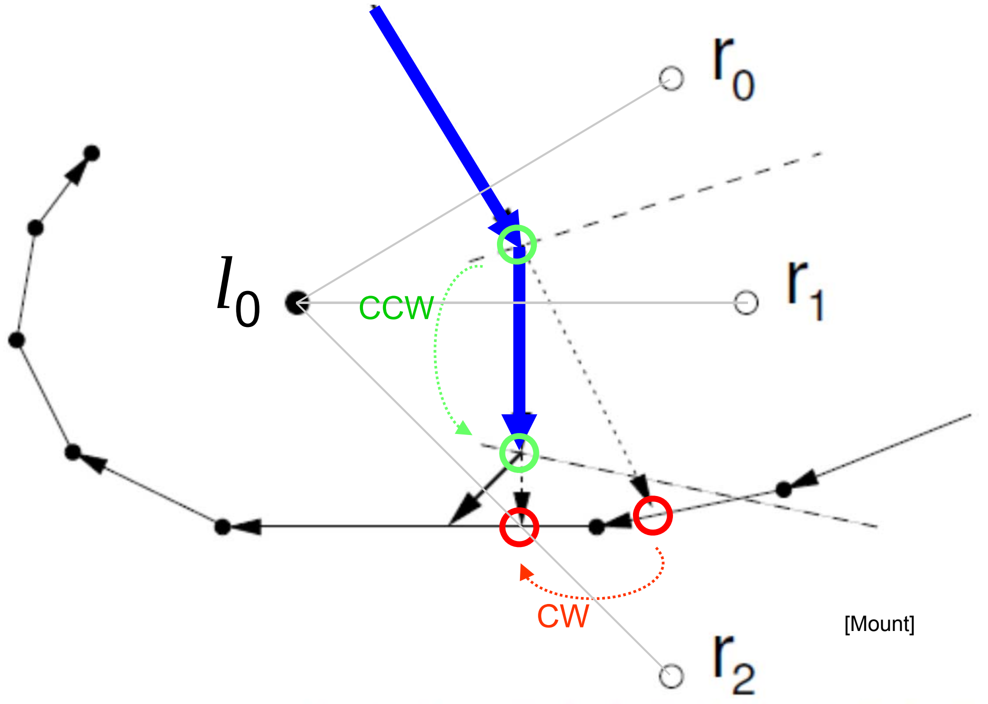


[Mount]

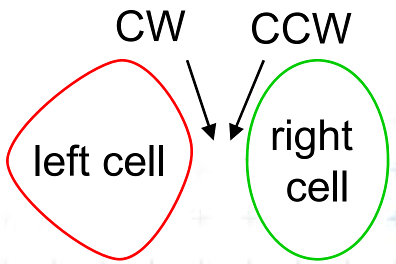


Monotone chain search in O(n)

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested ~once)
- In the left cell l_i continue CW, in the right cell r_i go CCW
- Image shows CW search on cell l_0 and CCW on cells r_i :

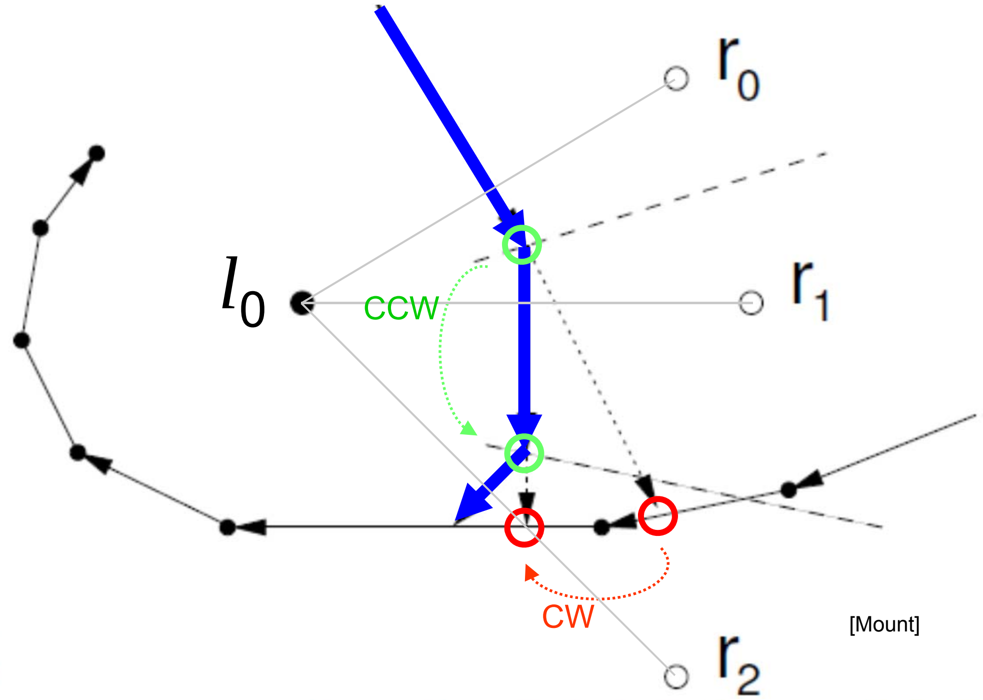


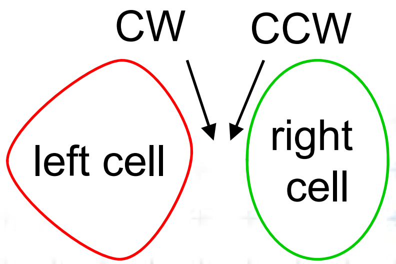
[Mount]



Monotone chain search in O(n)

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested ~once)
- In the left cell l_i continue CW, in the right cell r_i go CCW
- Image shows CW search on cell l_0 and CCW on cells r_i:

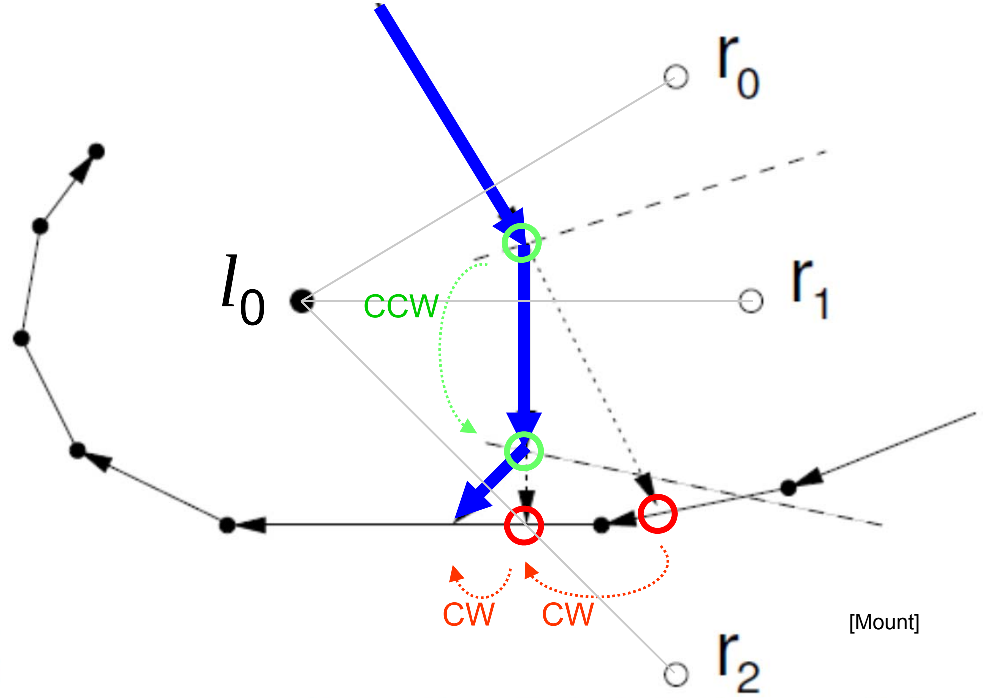




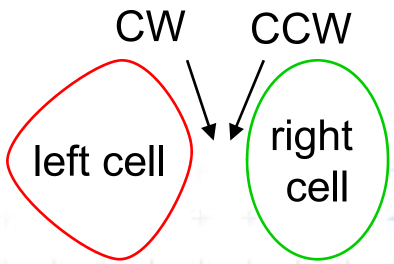
[Mount]

Monotone chain search in O(n)

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested ~once)
- In the left cell l_i continue CW, in the right cell r_i go CCW
- Image shows CW search on cell l_0 and CCW on cells r_i:

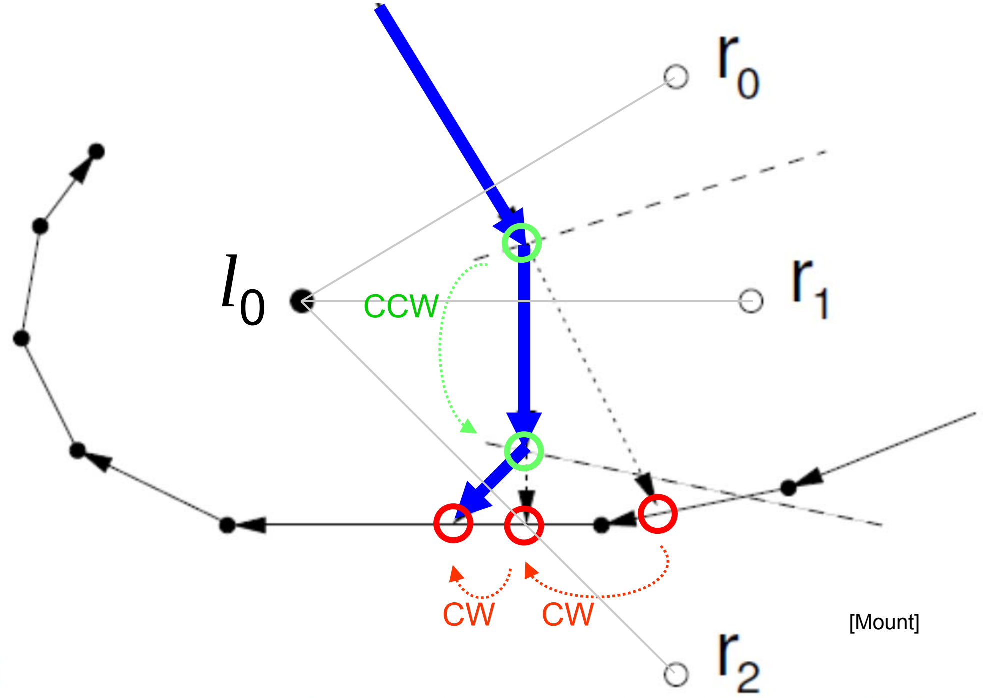


[Mount]

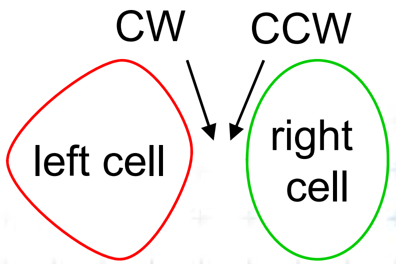


Monotone chain search in O(n)

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested ~once)
- In the left cell l_i continue CW, in the right cell r_i go CCW
- Image shows CW search on cell l_0 and CCW on cells r_i:

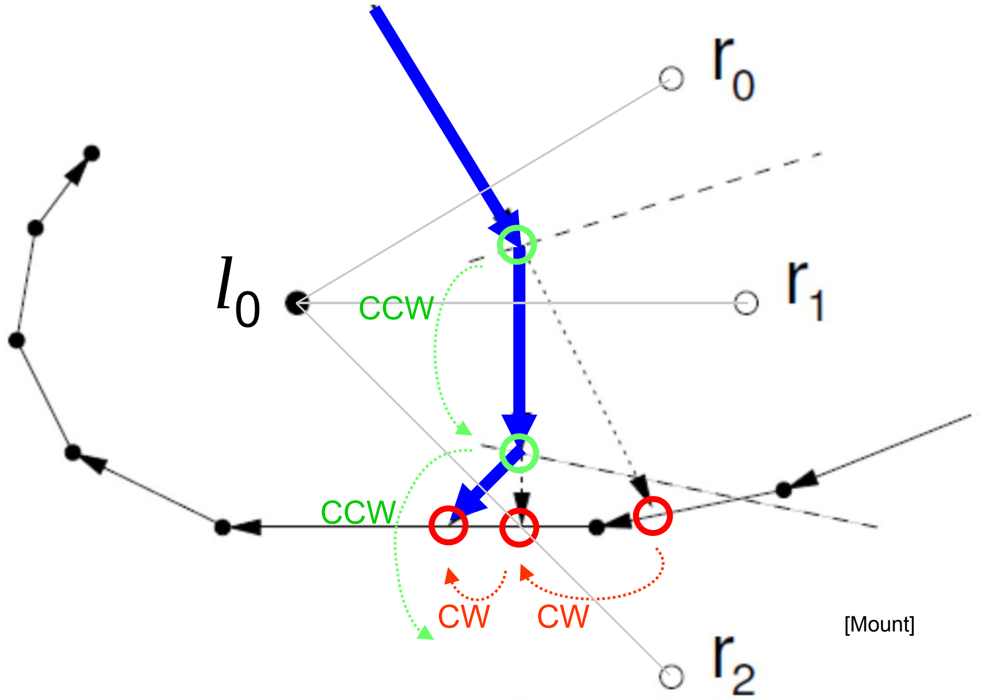


[Mount]

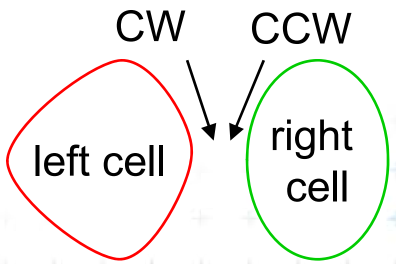


Monotone chain search in O(n)

- Avoid repeated rescanning of cell edges
- Start in the last tested edge of the cell (each edge tested ~once)
- In the left cell l_i continue CW, in the right cell r_i go CCW
- Image shows CW search on cell l_0 and CCW on cells r_i:



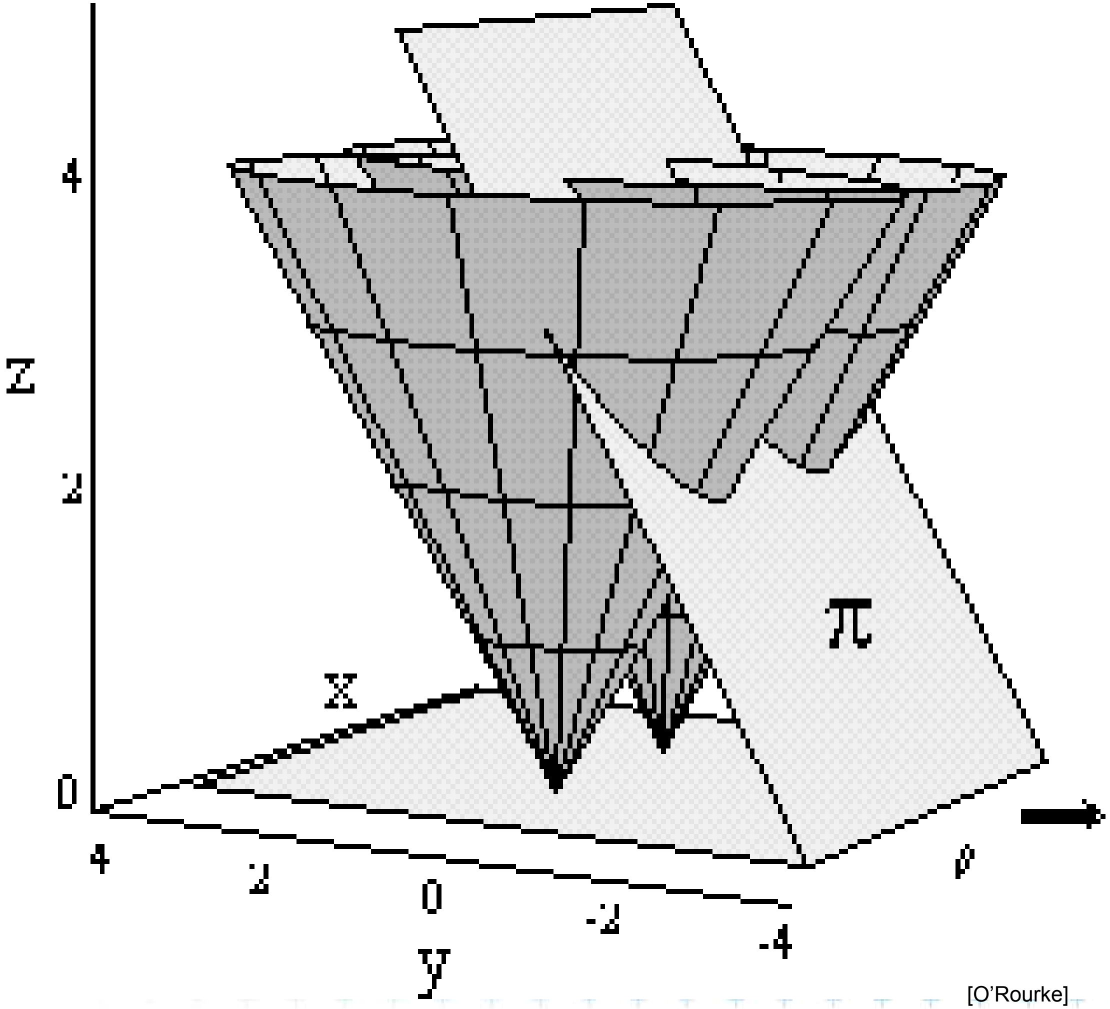
[Mount]



Divide and Conquer method complexity

- Initial sort $O(n \log n)$
- $O(\log n)$ recursion levels
  - O(n) each merge (chain search, trim, add edges to VD)
- Altogether $O(n \log n)$

Fortune’s sweep line algorithm – idea in 3D



[O’Rourke]

Cones in sites

Scanning plane π

Both slanted 45º

Projection of the intersection to xy:

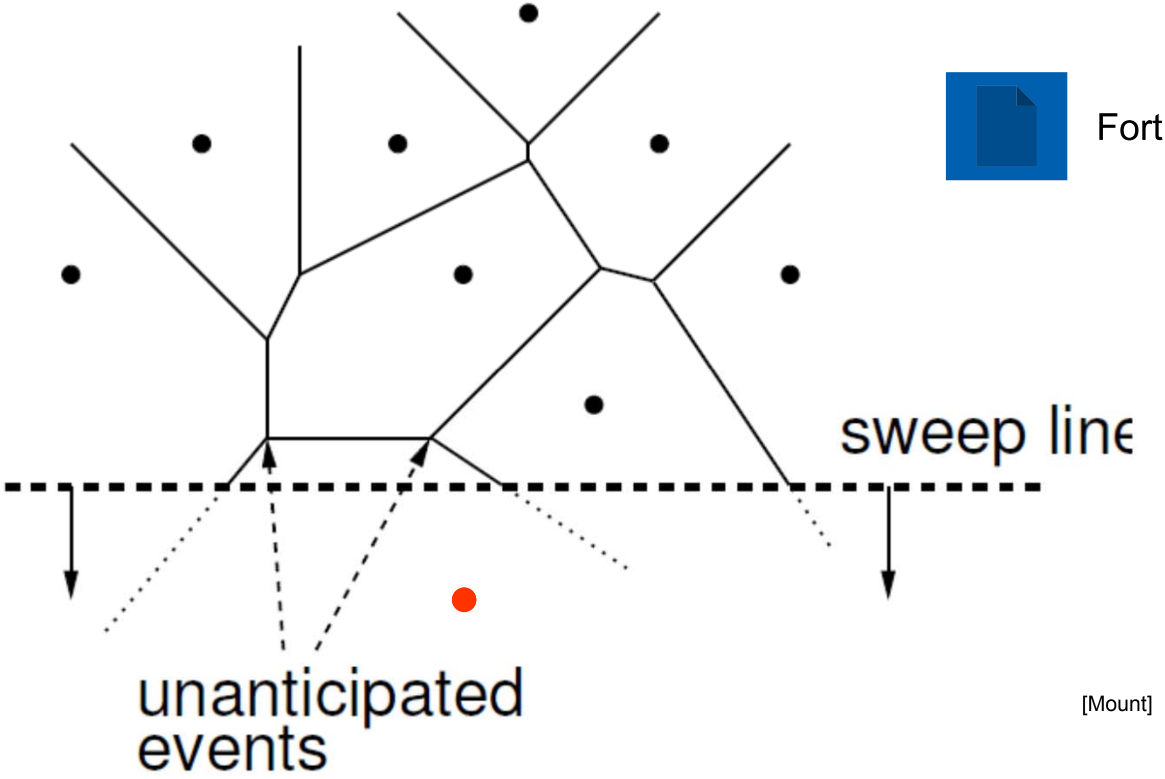
- Cone x plane => parabolic arcs
- Cone x cone => edges of VD

Fortune’s sweep line algorithm

DONE

TODO

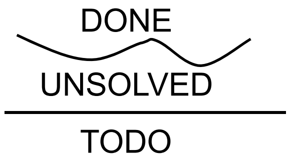
- Differs from “typical” sweep line algorithm
- Unprocessed sites ahead from sweep line may generate Voronoi vertex behind the sweep line



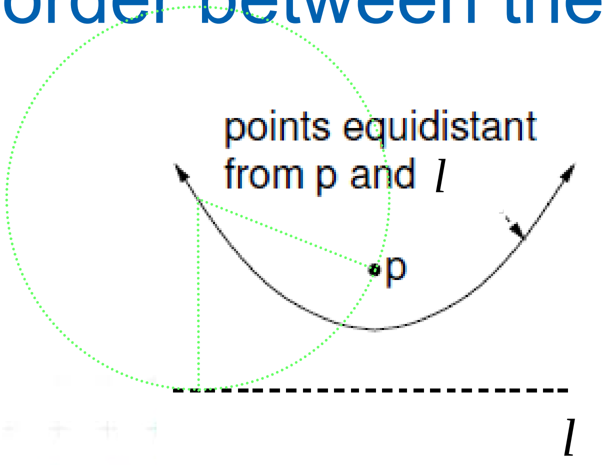
Fortune’s applet

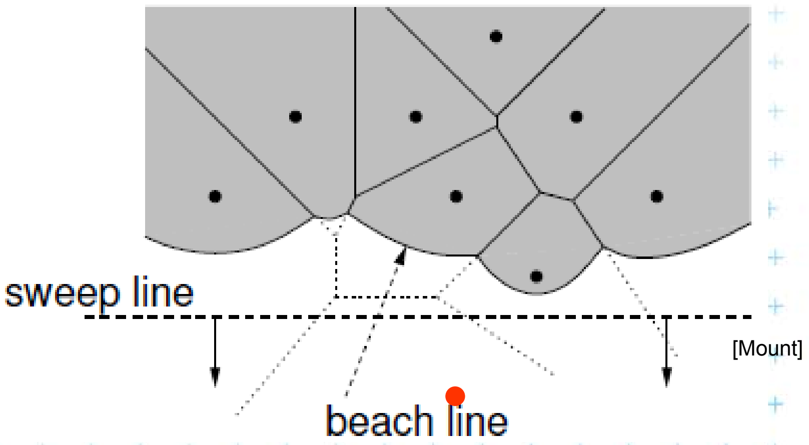
[Mount]

Fortune’s sweep line algorithm idea



- Subdivide the halfplane above the sweep line *l* into 2 regions
  1. Points closer to some site above than to sweep line *l* (solved part)
  2. Points closer to sweep line *l* than any point above (unsolved part – can be changed by sites below *l*)
- Border between these 2 regions is a beach line





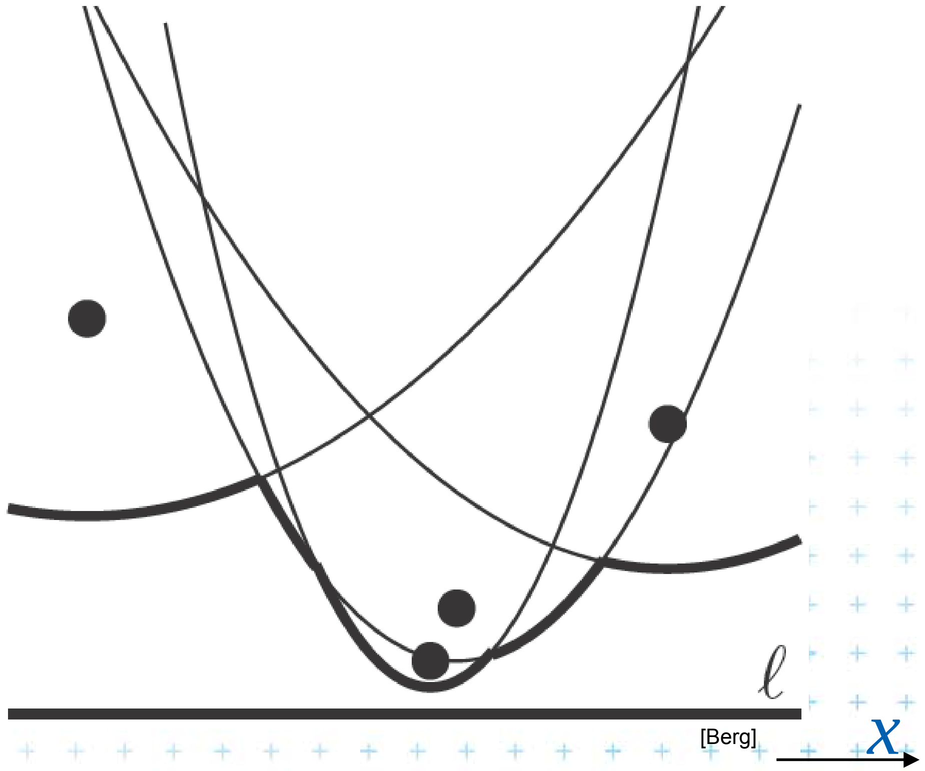
[Mount]

Sweep line and beach line

- Straight sweep line l
  - Separates processed and unprocessed sites (points)
- Beach line (Looks like waves rolling up on a beach)
  - Separates *solved* and *unsolved* regions above sweep line (separates sites above l that can be changed from sites that cannot be changed by sites below l)
  - *x*-monotonic curve made of parabolic arcs
  - Follows the sweep line
  - Prevents us from missing unanticipated events until the sweep line encounters the corresponding site

Beach line

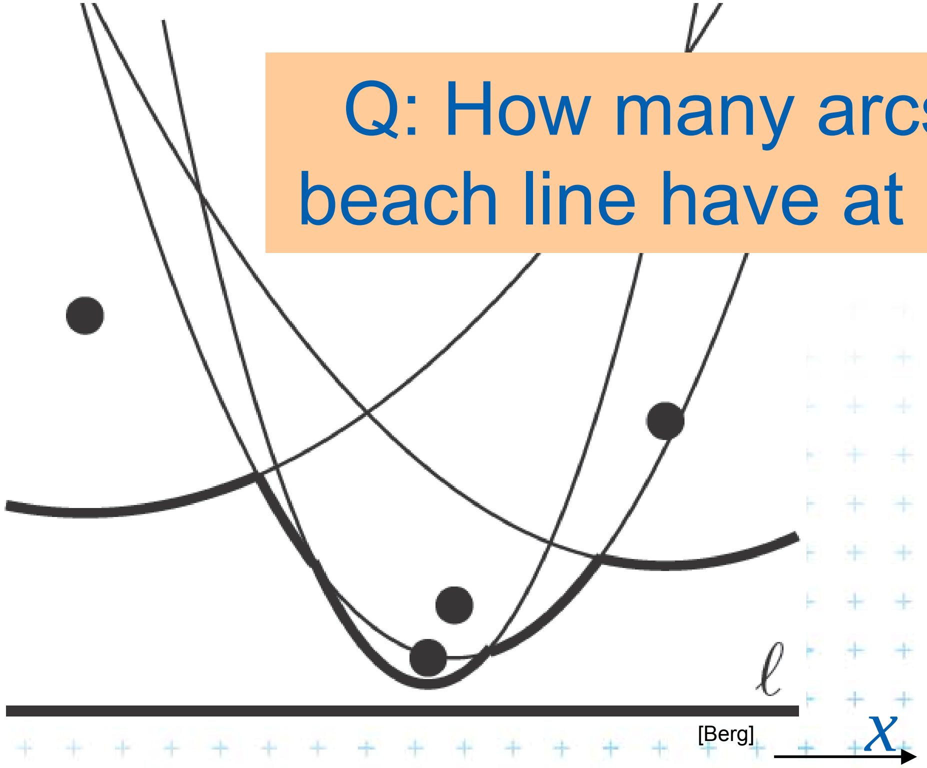
- Every site p_i above *l* defines a complete parabola
- Beach line is the function, that passes through the lowest points of all the parabolas (lower envelope)



[Berg]

Beach line

- Every site p_i above l defines a complete parabola
- Beach line is the function, that passes through the lowest points of all the parabolas (lower envelope)

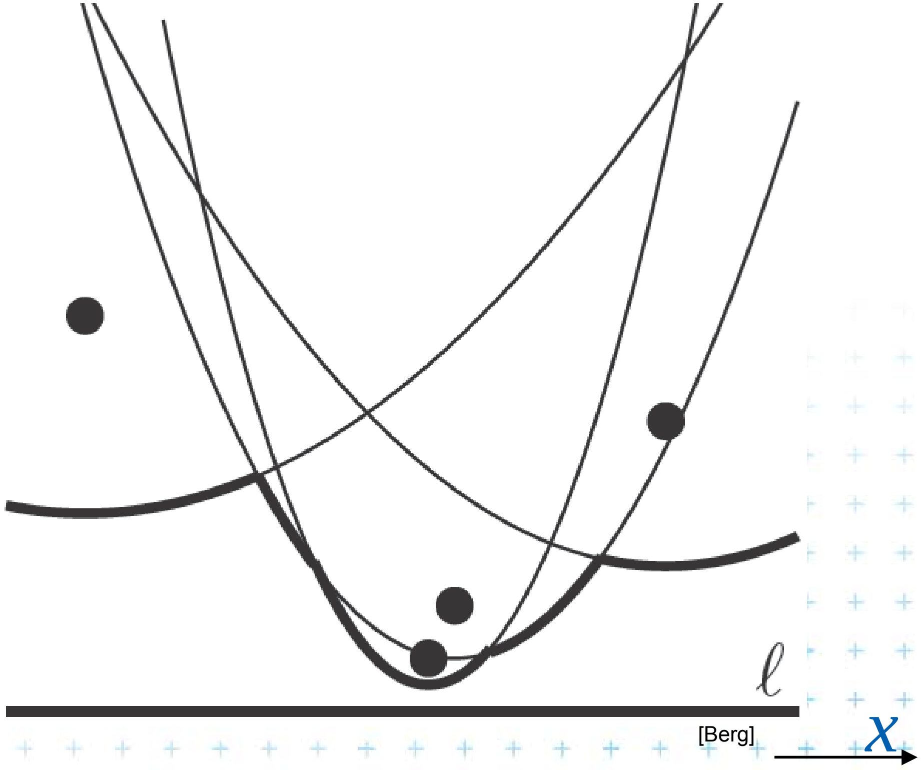


Q: How many arcs may the beach line have at maximum?

[Berg]

Beach line

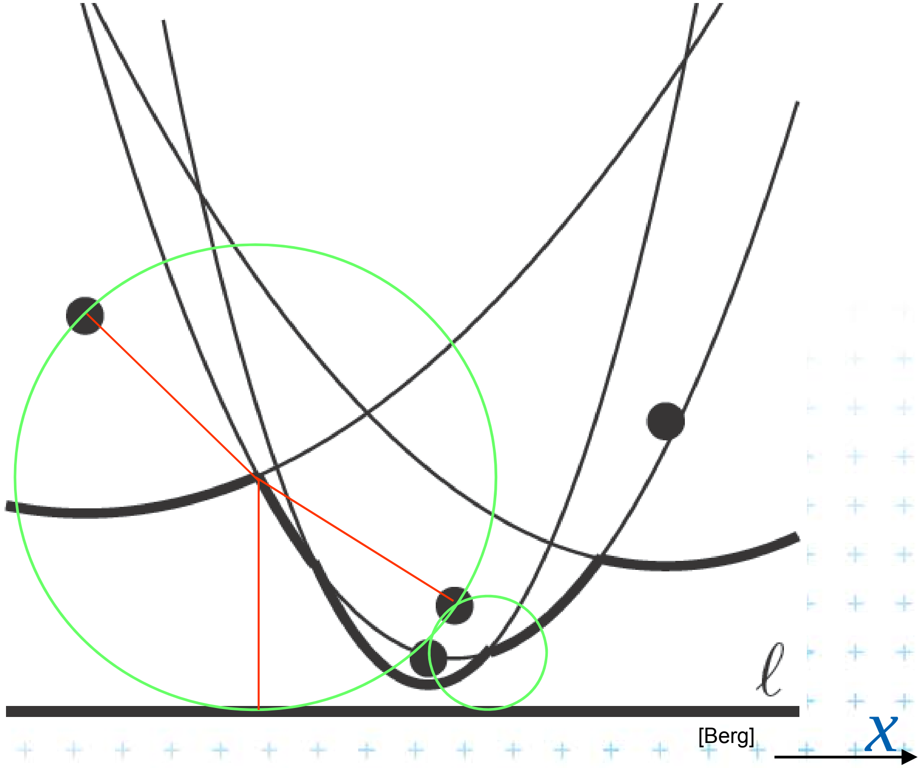
- Every site p_i above *l* defines a complete parabola
- Beach line is the function, that passes through the lowest points of all the parabolas (lower envelope)



Break point (*bod zlomu*)

= Intersection of two arcs on the beach line

- Equidistant to 2 sites and sweep line *l*
- Lies on Voronoi edge of the final diagram



Notes

Beach line is x-monotone

= every vertical line intersects it in exactly ONE point

Along the beach line

- Parabolic arcs are ordered
- Breakpoints are ordered

Breakpoints

- trace the Voronoi edges
- compute their position on the fly from neighboring arcs

Events

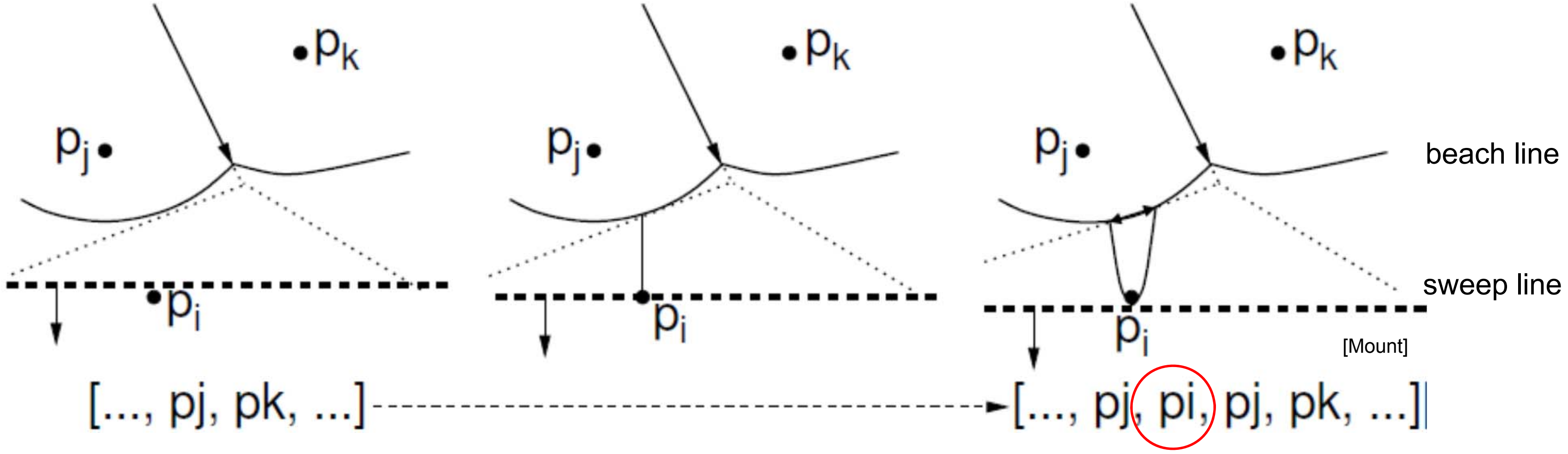
What event types exist?

Events

There are two types of events:

- Site events (SE)
  - When the sweep line passes over a new site p_i,
    - *new arc* is added to the beach line
    - *new edge fragment* added to the VD.
  - All SEs known from the beginning (sites sorted by *y*)
- Voronoi vertex event ([Berg] calls a circle event)
  - When the parabolic arc shrinks to zero and disappears, *new Voronoi vertex* is created.
  - Created dynamically by the algorithm for triples or more neighbors on the beach line (triples changed by both types of events)

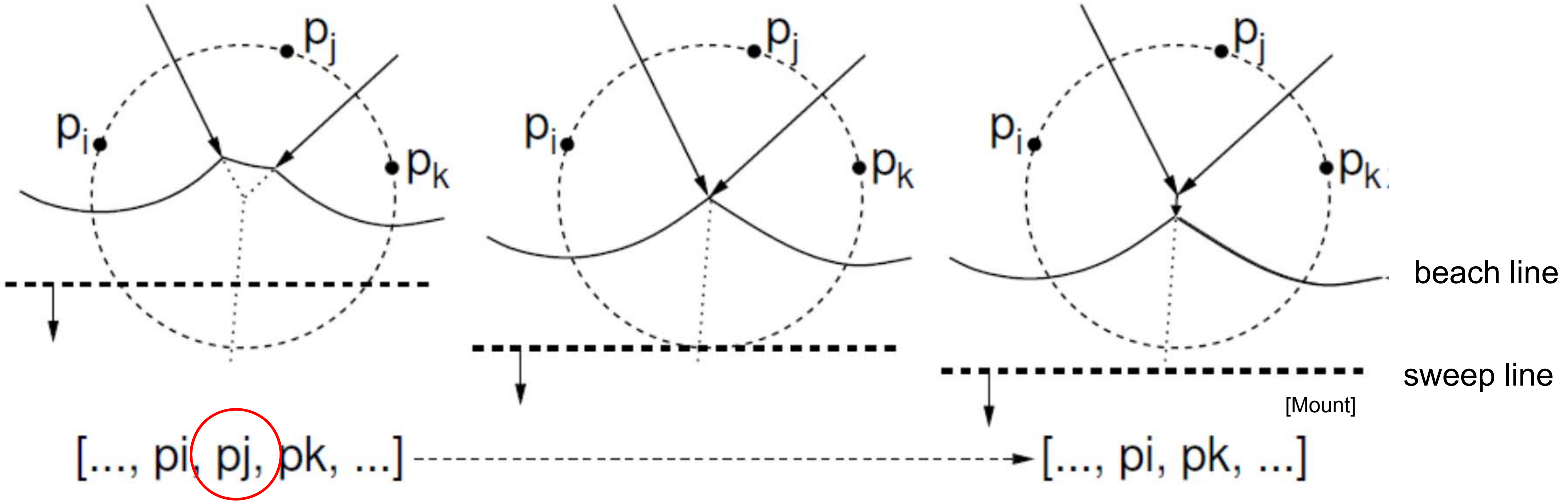
Site event



Generated when the sweep line passes over a site p_i

- New parabolic arc created,
  it starts as a vertical ray from p_i to the beach line
- As the sweep line sweeps on, the arc grows wider
- The entry $\langle\ldots, p_j, \ldots\rangle$ on the sweep line status is replaced by the triple $\langle\ldots, p_j, p_i, p_j, \ldots\rangle$
- Dangling future VD edge created on the bisector (p_i, p_j)

Voronoi vertex event (circle event)



[..., pi, pj, pk, ...] ⟶ [..., pi, pk, ...]

Generated when l passes the lowest point of a circle

- Sites p_i , p_j , p_k appear consecutively on the beach line
- Circumcircle lies partially below the sweep line (Voronoi vertex has not yet been generated)
- This circumcircle contains no point below the sweep line (no future point will block the creation of the vertex)
- Vertex & bisector (p_i, p_k) created, (p_i, p_j) & (p_j, p_k) finished
- One parabolic arc removed from the beach line

Data structures

1. (Partial) Voronoi diagram
2. Beach line data structure T
3. Event queue Q

Data structures

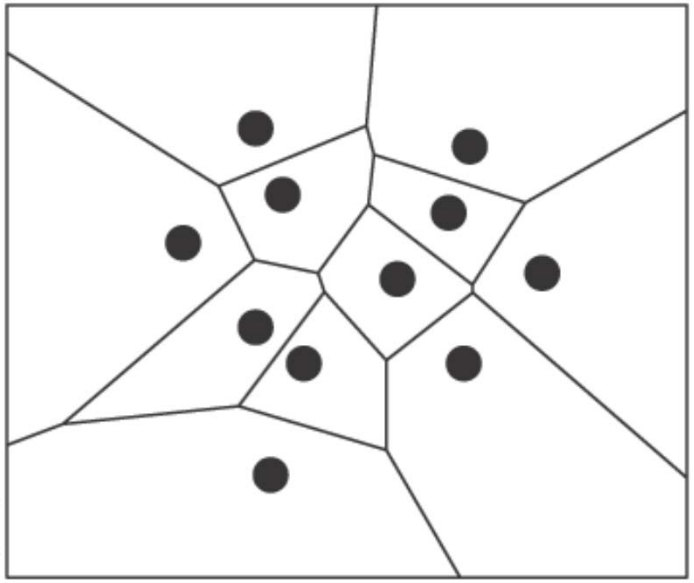
1. (Partial) Voronoi diagram
2. Beach line data structure T
3. Event queue Q

1. VD edges arise during: site event circle event?
2. VD vertices arise during: site event circle event?
3. Site events known from the beginning: yes no?
4. Circle events known from the beginning: yes no?

1. (Partial) Voronoi diagram data structure

Any PSLG data structure, e.g. DCEL (planar stright line graph)

- Stores the VD during the construction
- Contain unbounded edges
  - dangling edges during the construction (managed by the beach line DS) and
  - edges of unbounded cells at the end
    => create a bounding box



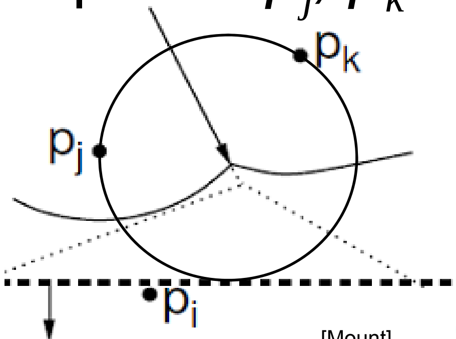
[Berg]

2. Beach line tree data structure T – status

- Used to locate the arc directly above a new site
- E.g. Binary tree *T*

p_i – possibly multiple times

  - Leaves - ordered arcs along the beach line (x-monotone)
    - *T* stores only the sites p_i in leaves, *T* does not store the parabolas
  - Inner tree nodes - breakpoints as ordered pairs $<p_j, p_k>$
    - p_j, p_k are neighboring sites
    - Breakpoint position computed on the fly from p_j, p_k and y-coord of the sweep line

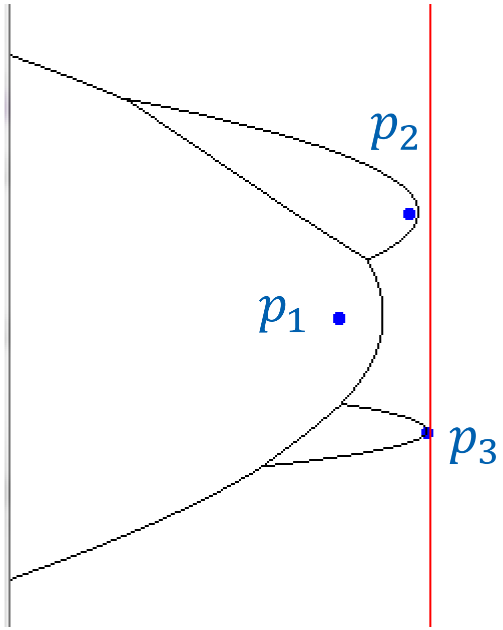


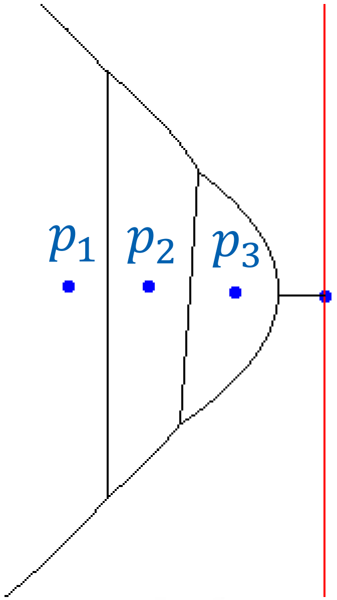
[Mount]

  - Pointers to other two DS
    - In leaves – pointer to event queue, point to node when arc disappears via Voronoi vertex event – if it exists
    - In inner nodes - pointer to (dangling) half-edge in DCEL of VD, that is being traced out by the break point

Max 2n -1 arcs on the beach line

New site splits just one arc





p_1 +1

$p_1p_2p_1$ +2

$p_1p_3p_1p_2p_1$ +2

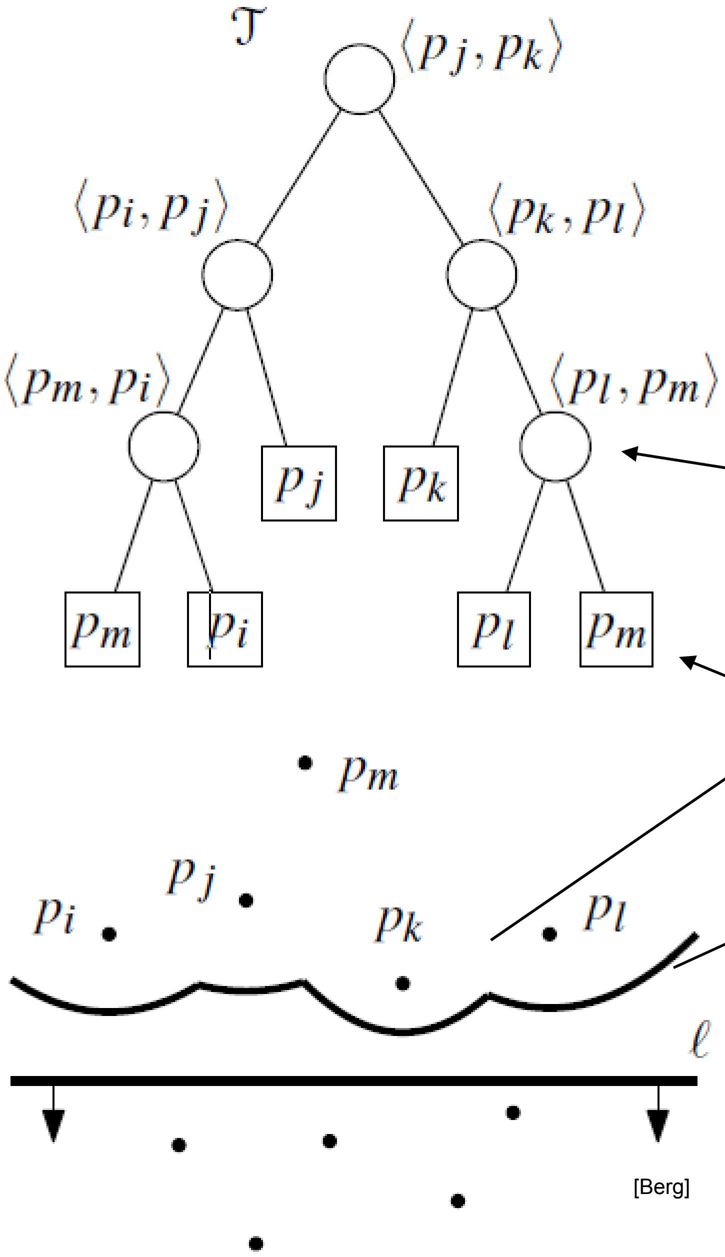
p_1 +1

$p_1p_2p_1$ +2

$p_1p_2p_3p_2p_1$ +2 → Leaves in T

2. Beach line tree T

x-coord computed on the fly for a given position of the beach line l


$\mathcal{T}$
$\langle p_j, p_k \rangle$
$\langle p_i, p_j \rangle$
$\langle p_k, p_l \rangle$
$\langle p_m, p_i \rangle$
$\langle p_l, p_m \rangle$
p_j
p_k
p_m
p_i
p_l
p_m
p_m
p_j
p_i
p_k
p_l
ℓ

[Berg]

Break points
= inner nodes in T

Arcs = Leaves in T

3. Event queue Q

- Priority queue, ordered by y-coordinate
- For site event
  - stores the site itself
  - known from the beginning
- For Voronoi vertex event (circle event)
  - stores the lowest point of the circle
  - stores also pointer to the leaf in tree T (represents the parabolic arc that will disappear)
  - created by both events, when triples of points become neighbors (possible max three triples for a site)
  - p_i, p_j, p_k, p_l, p_m insert of p_k can create up to 3 triples and delete up to 2 triples (p_i, p_j, p_l) and (p_j, p_l, p_m)

Fortune’s algorithm

FortuneVoronoi(*P*)

Input: A set of point sites $P = \{p_1, p_2, \ldots, p_n\}$ in the plane

Output: Voronoi diagram Vor(*P*) inside a bounding box in a DCEL struct.

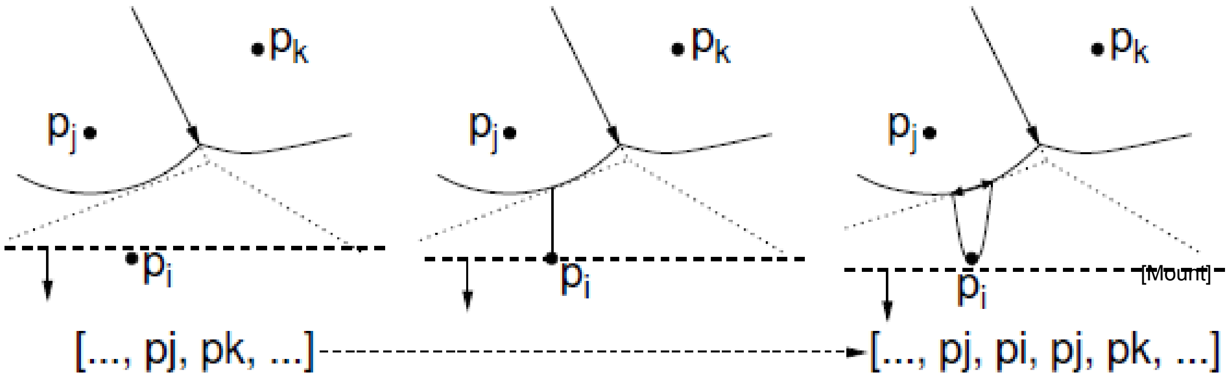
```
Init event queue Q with all site events
while( Q not empty) do
    consider the event with largest $y$-coordinate in Q (next in the queue)
    if( event is a site event at site $p_i$ )
        then HandleSiteEvent($p_i$)
        else  HandleVoroVertexEvent($p_i$), where $p_i$ is the lowest point
                  of the circle causing the event
    remove the event from Q
Create a bbox and attach half-infinite edges in $T$ to it in DCEL.
Traverse the halfedges in DCEL and
    add cell records and pointers to and from them
```

Handle site event

HandleSiteEvent(*p_i*)

Input: event site p_i

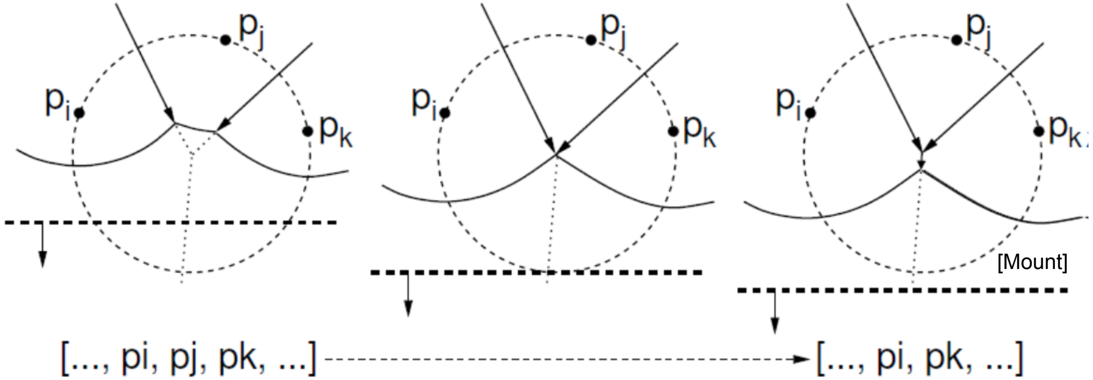
Output: updated DCEL



1. Search in T for arc α vertically above p_i. Let p_j be the corresponding site
2. Apply insert-and-split operation, inserting a new entry of p_i to the beach line T (new arc), thus replacing $\langle \ldots, p_j, \ldots \rangle$ with $\langle \ldots, p_j, p_i, p_j, \ldots \rangle$
3. Create a new (dangling) edge in the Voronoi diagram, which lies on the bisector between p_i and p_j
4. Neighbors on the beach line changed -> check the neighboring triples of arcs and *insert or delete Voronoi vertex events* (insert only if the circle intersects the sweep line and it is not present yet).
   Note: Newly created triple p_j , p_i , p_j cannot generate a circle event because it only involves two distinct sites.

Handle Voronoi vertex (circle) event

HandleVoroVertexEvent(p_j)
Input: event site p_j
Output: updated DCEL



Let p_i , p_j , p_k be the sites that generated this event (from left to right).

1. Delete the entry p_j from the beach line (thus eliminating its arc α), i.e.: Replace a triple $\langle \ldots, p_i, p_j, p_k, \ldots \rangle$ with $\langle \ldots, p_i, p_k, \ldots \rangle$ in *T.*
2. Create a new vertex in the Voronoi diagram (at circumcenter of $\langle p_i, p_j, p_k \rangle$) and join the two Voronoi edges for the bisectors $\langle p_i, p_j \rangle$ and $\langle p_j, p_k \rangle$ to this vertex (dangling edges – created in step 3 above).
3. Create a new (dangling) edge for the bisector between $\langle p_j, p_k \rangle$
4. Delete any Voronoi vertex events (max. three) from Q that arose from triples involving the arc α of p_j and generate (two) new events corresponding to consecutive triples involving p_i, and p_k.

Beach line modification

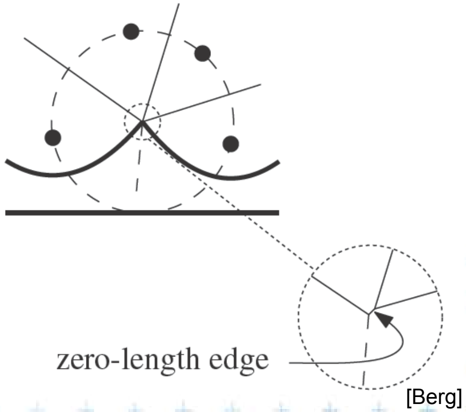
Q: Beach line contains: abcdef

After deleting of d, which triples vanish and which triples are added to the beach line?

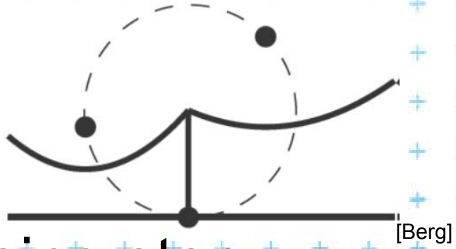
Handling degeneracies

Algorithm handles degeneracies correctly

- 2 or more events with the same y
  - if x coords are different, process them in any order
  - if x coords are the same (cocircular sites)
    process them in any order,
    it creates duplicated vertices with
    zero-length edges,
    remove them in post processing step



- degeneracies while handling an event
  - Site below a beach line breakpoint
  - Creates circle event on the same position
    remove zero-length edges in post processing step



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, Chapter 7, http://www.cs.uu.nl/geobook/

[Mount] David Mount, - *CMSC 754: Computational Geometry, Lecture Notes for Spring 2007*, University of Maryland, Lectures 12 and 29. http://www.cs.umd.edu/class/spring2007/cmsc754/lectures.shtml

[Preparata] Preperata, F.P., Shamos, M.I.: *Computational Geometry. An Introduction.* Berlin, Springer-Verlag,1985. Chapter 5

[VoroGlide] VoroGlide applet: http://www.pi6.fernuni-hagen.de/GeomLab/VoroGlide/

[Fortune] Fortune's algorithm applet: http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/Voronoi/Fortune/fortune.htm

[Muhama] http://www.personal.kent.edu/~rmuhamma/Compgeometry/compgeom.html

http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/Voronoi/DivConqVor/divConqVor.htm

DCGI

DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

VORONOI DIAGRAM PART II

PETR FELKEL

FEL CTU PRAGUE

Version from 16.11.2017

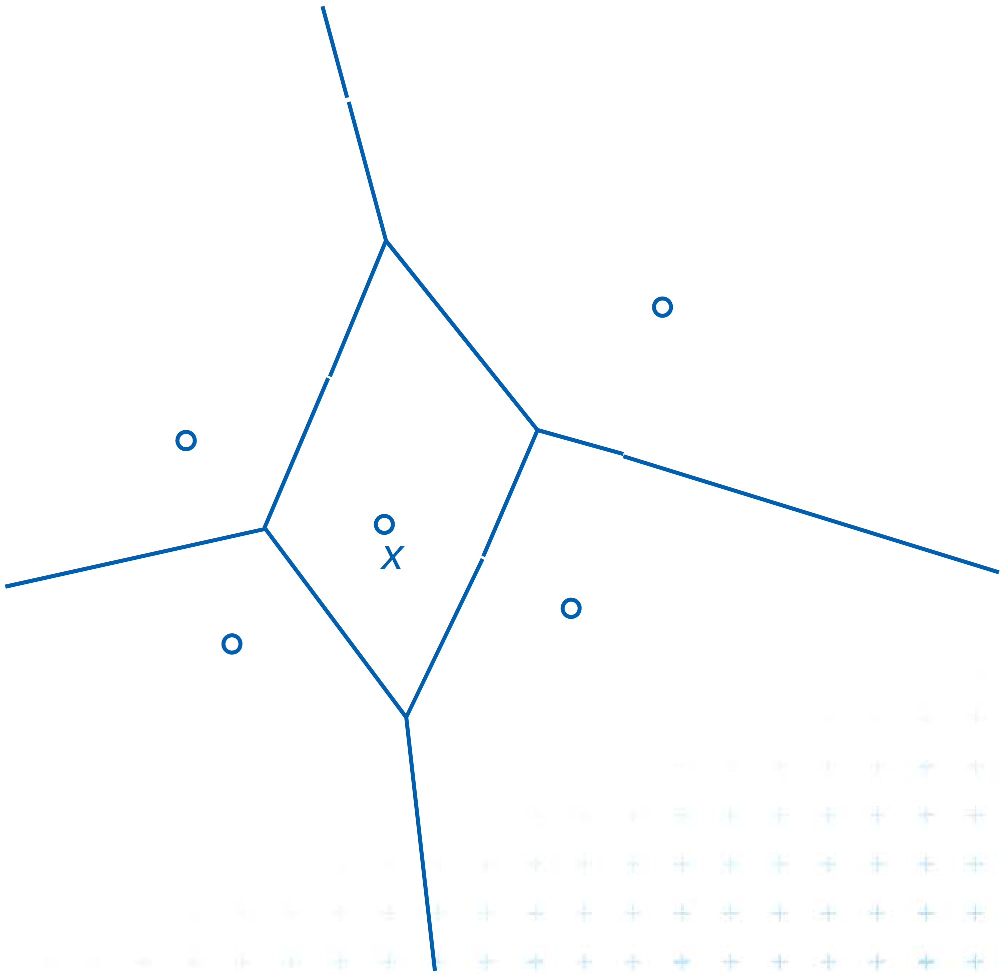
Talk overview

- Incremental construction
- Voronoi diagram of line segments
- VD of order k
- Farthest-point VD

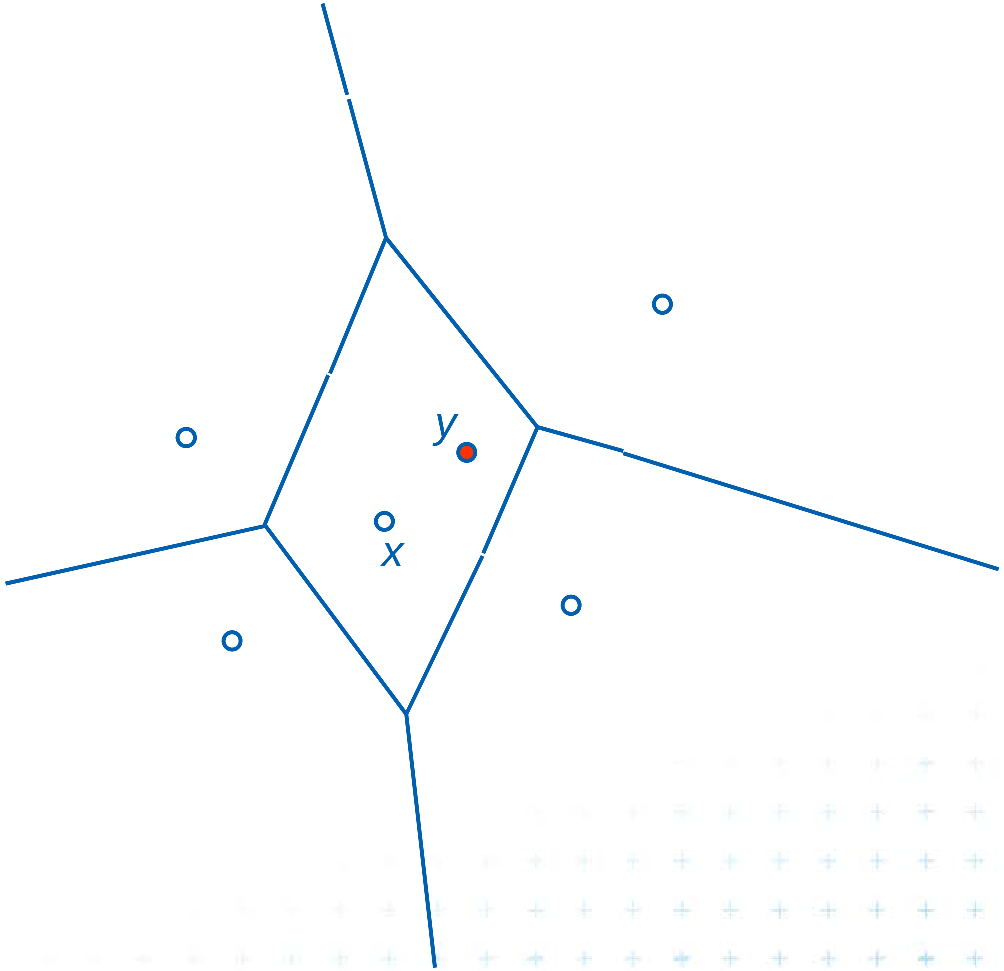
Summary of the VD terms

- Site = input point, line segment, …
- Cell = area around the site, in VD_1 the nearest to site
- Edge, arc = part of Voronoi diagram (border between cells)
- Vertex = intersection of VD edges

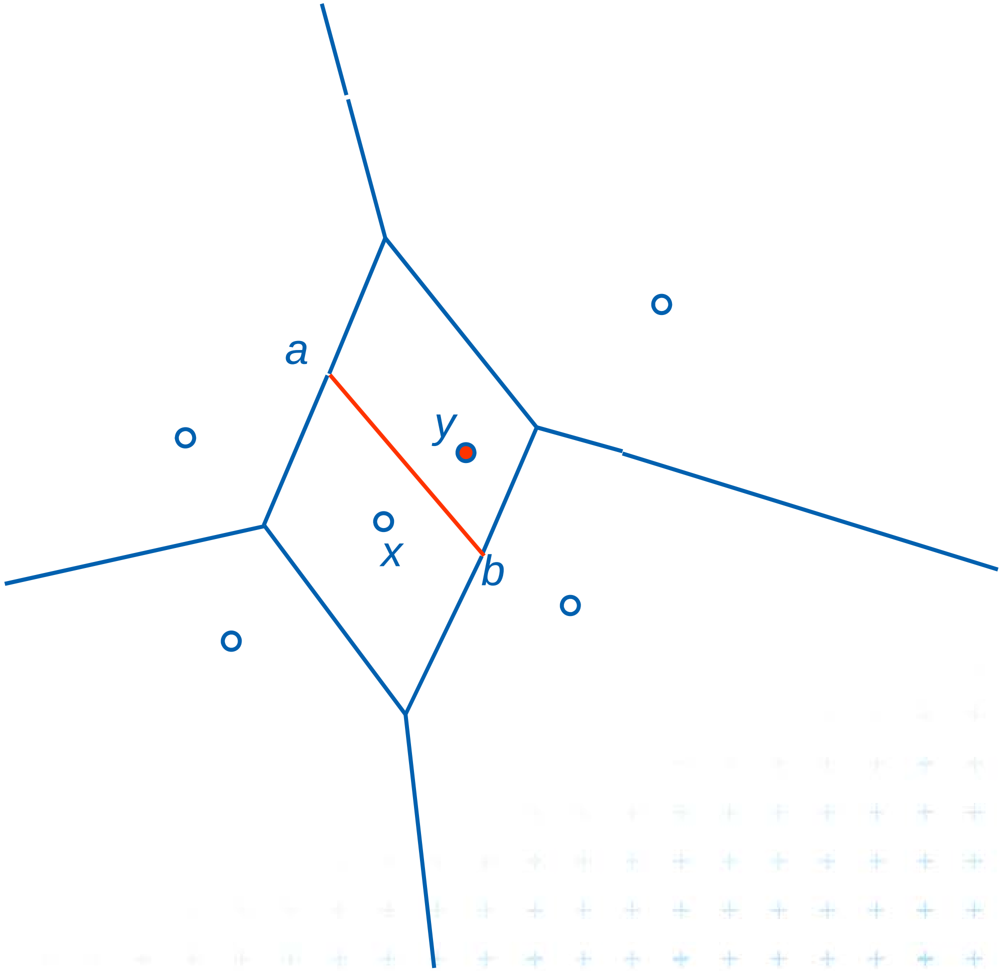
Incremental construction – bounded cell



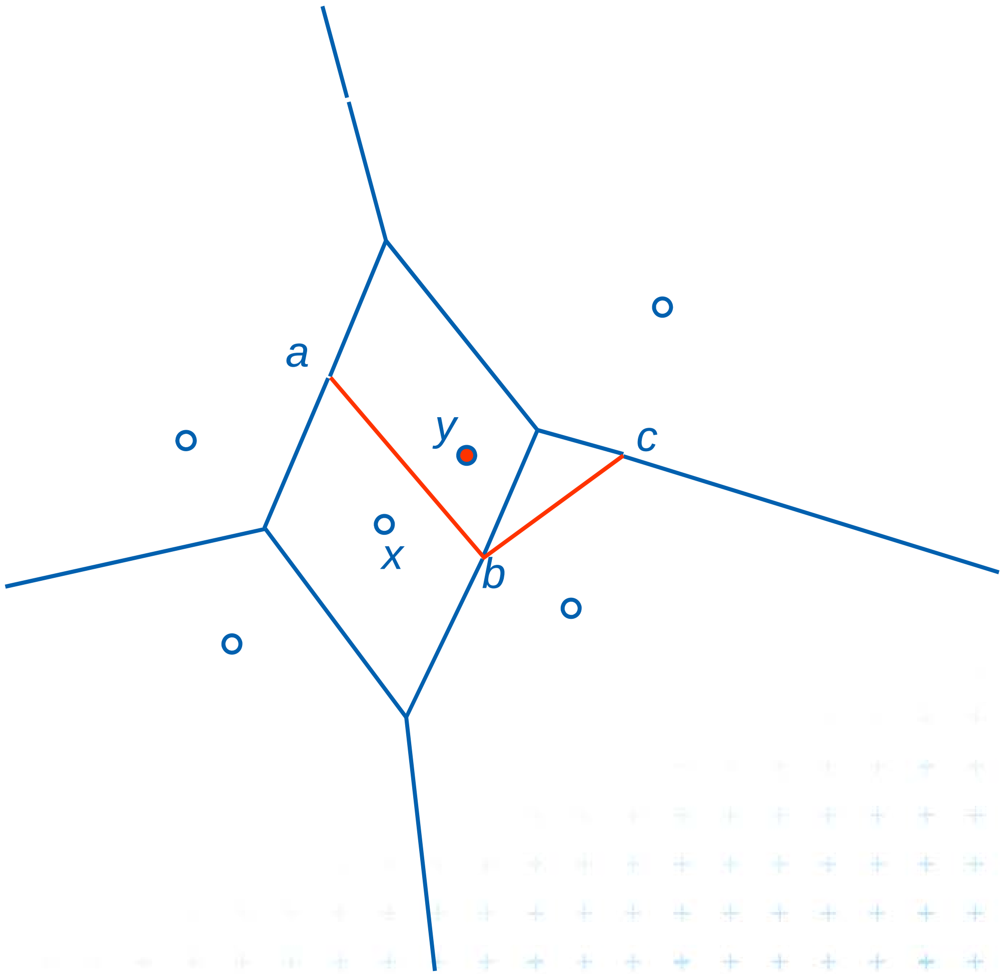
Incremental construction – bounded cell



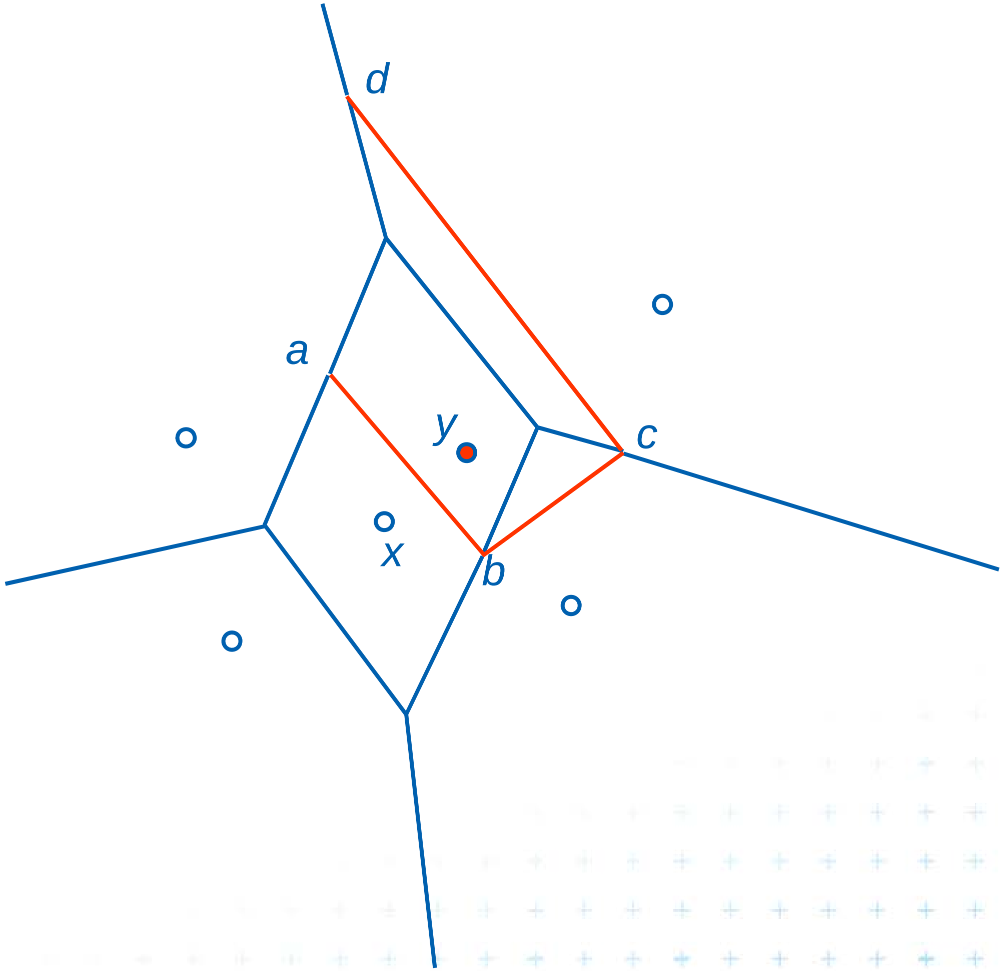
Incremental construction – bounded cell



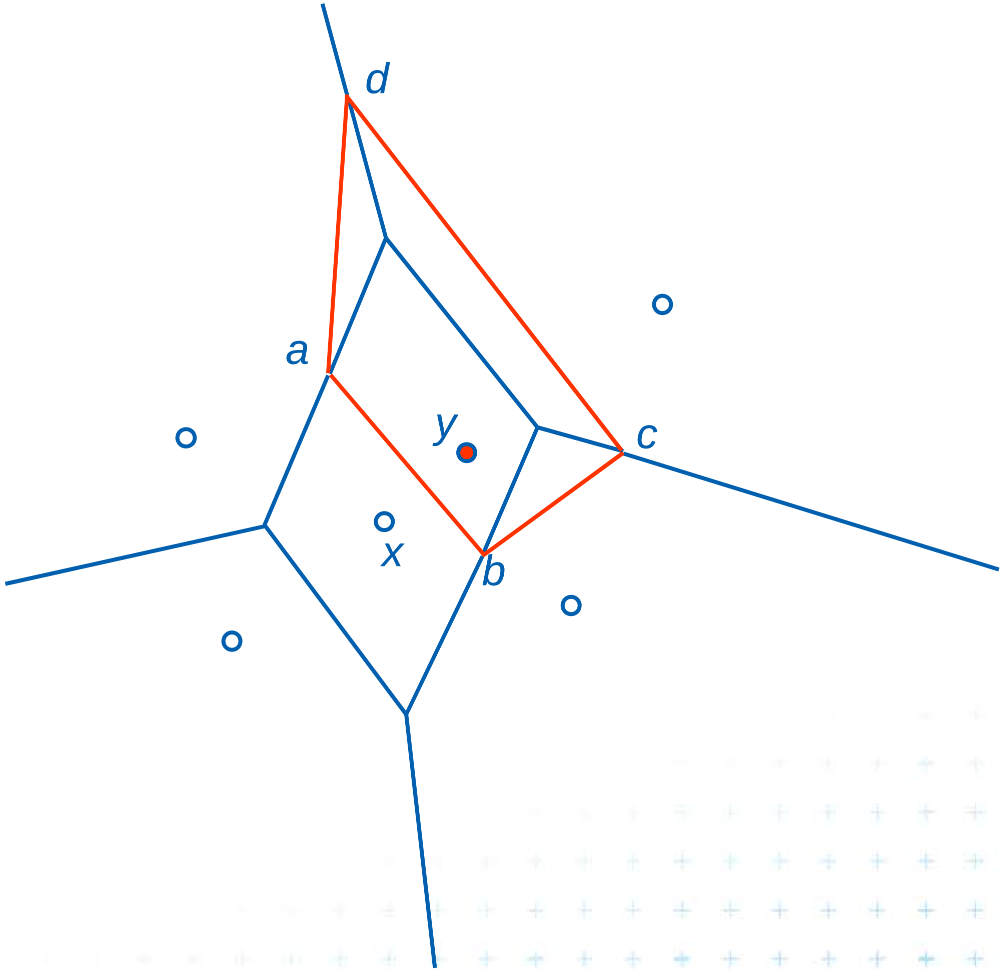
Incremental construction – bounded cell



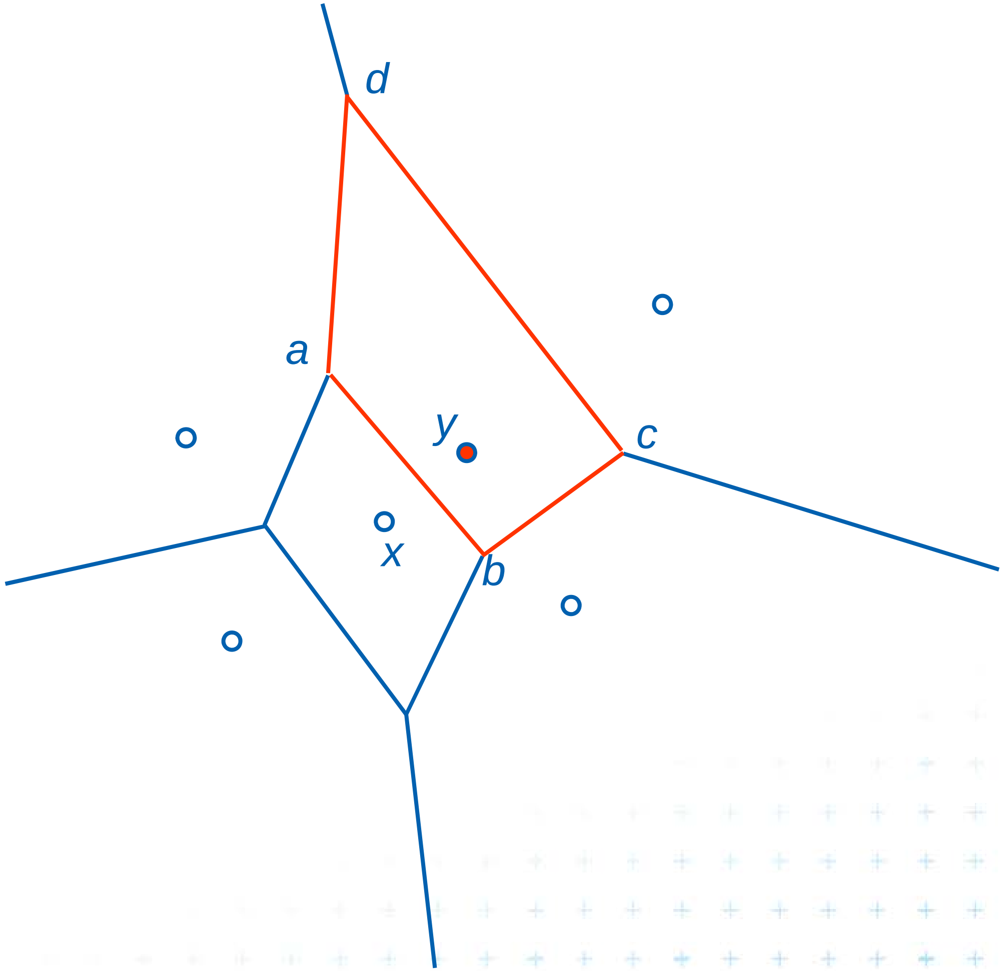
Incremental construction – bounded cell



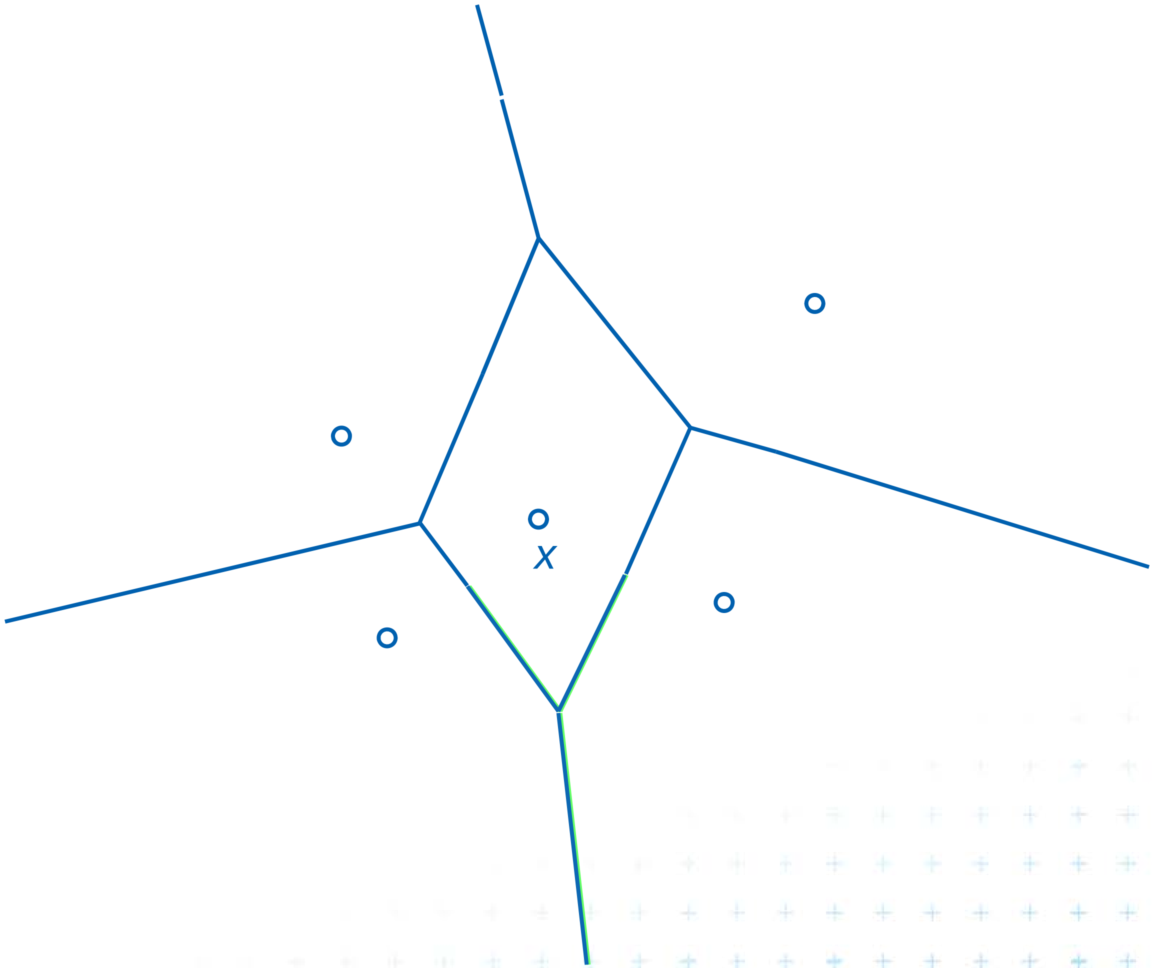
Incremental construction – bounded cell



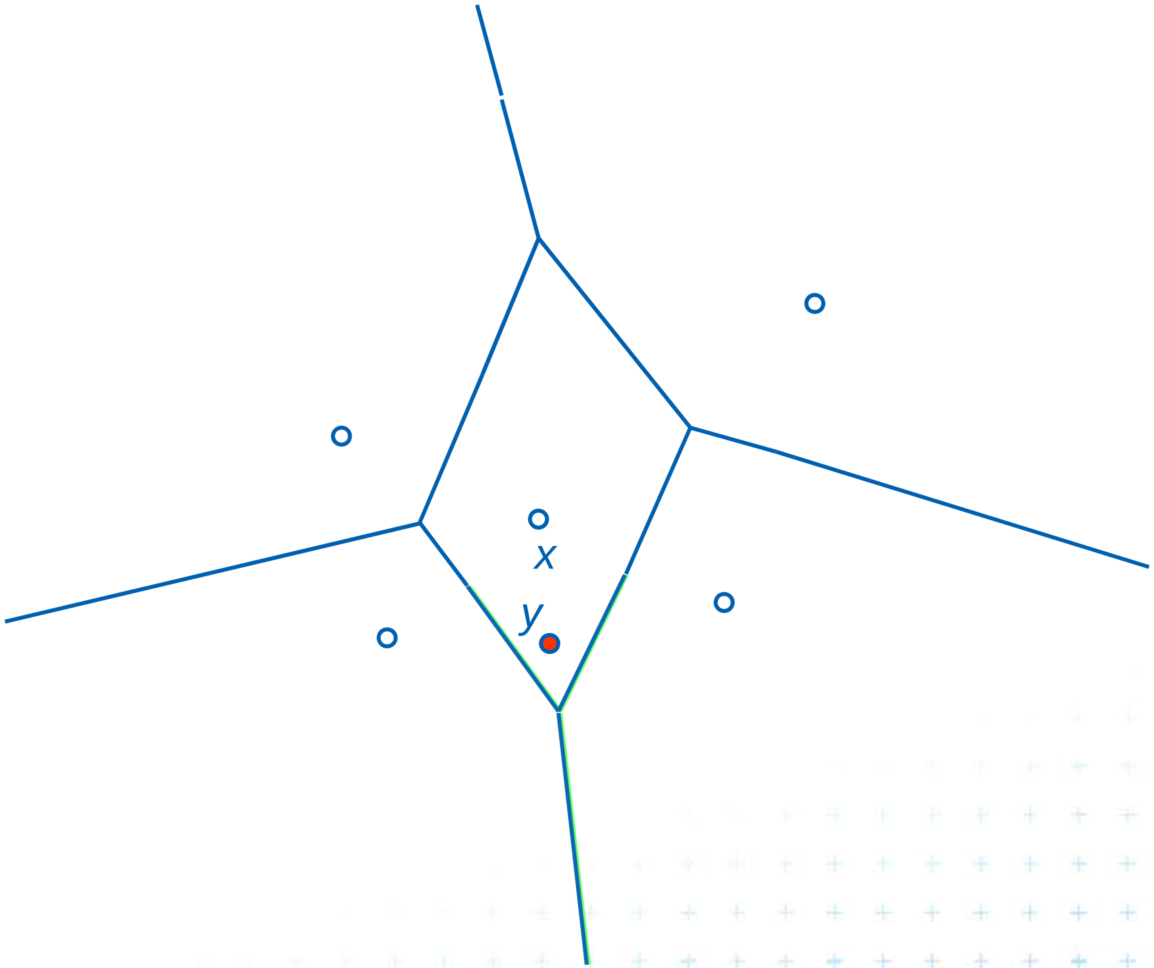
Incremental construction – bounded cell



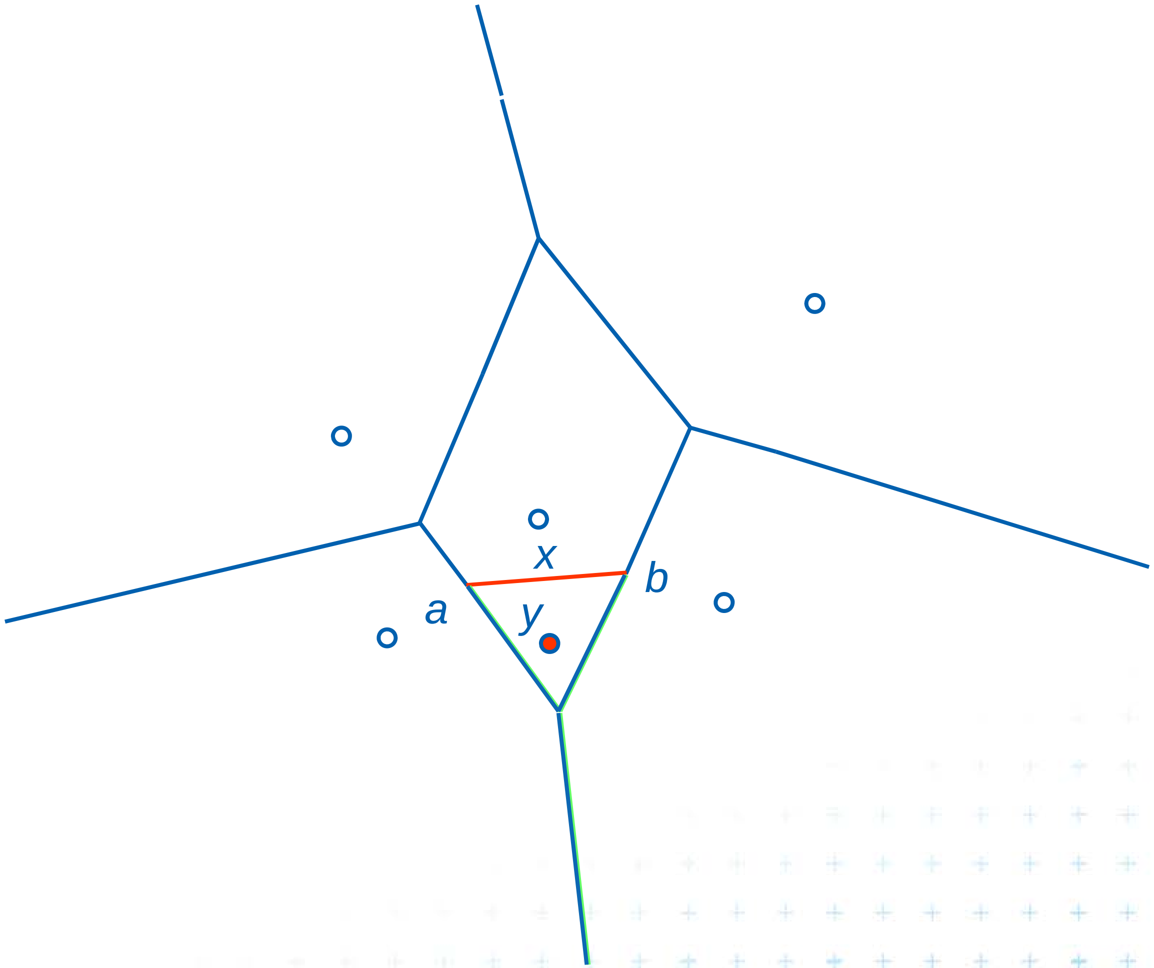
Incremental construction – unbounded cell



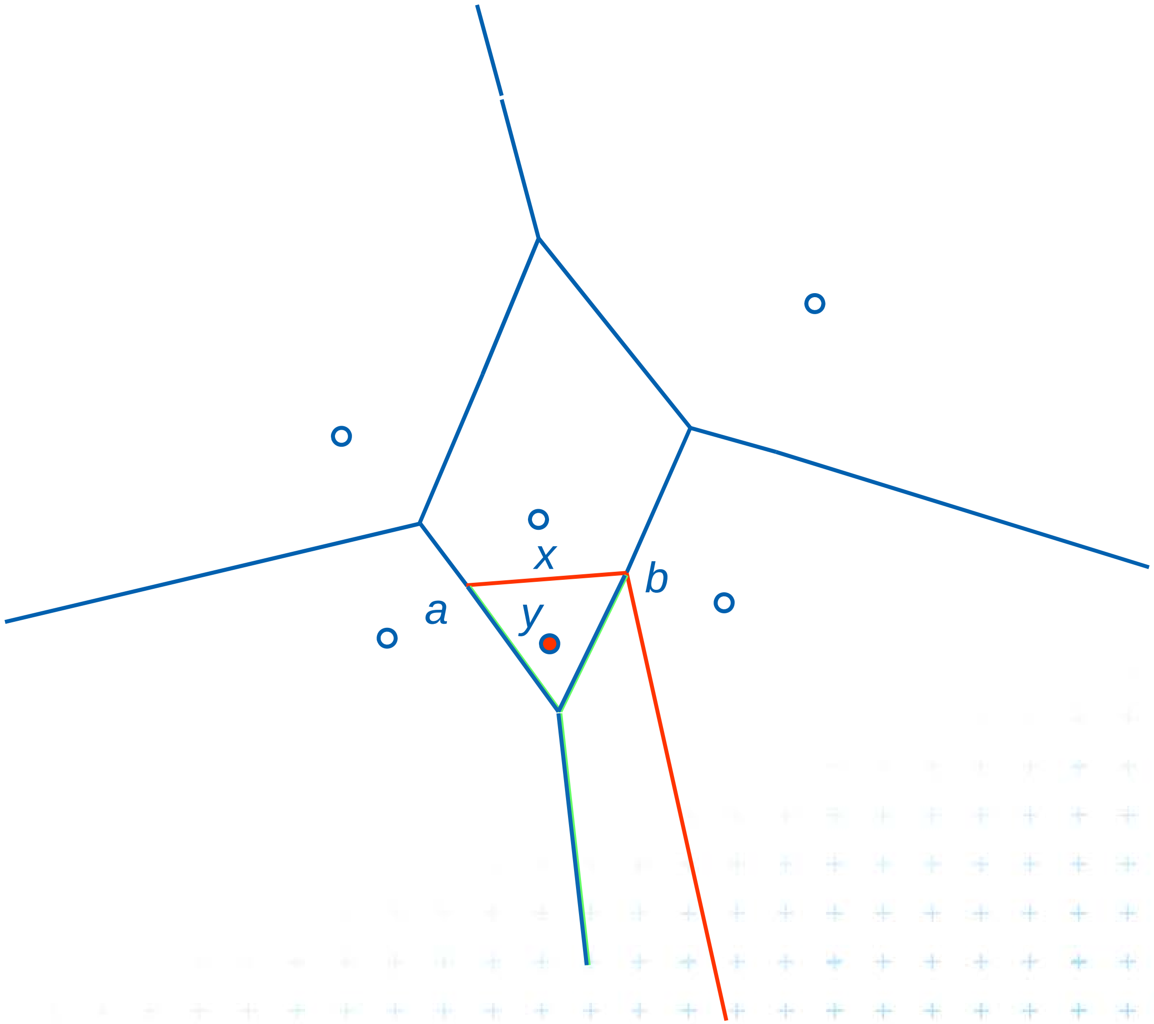
Incremental construction – unbounded cell



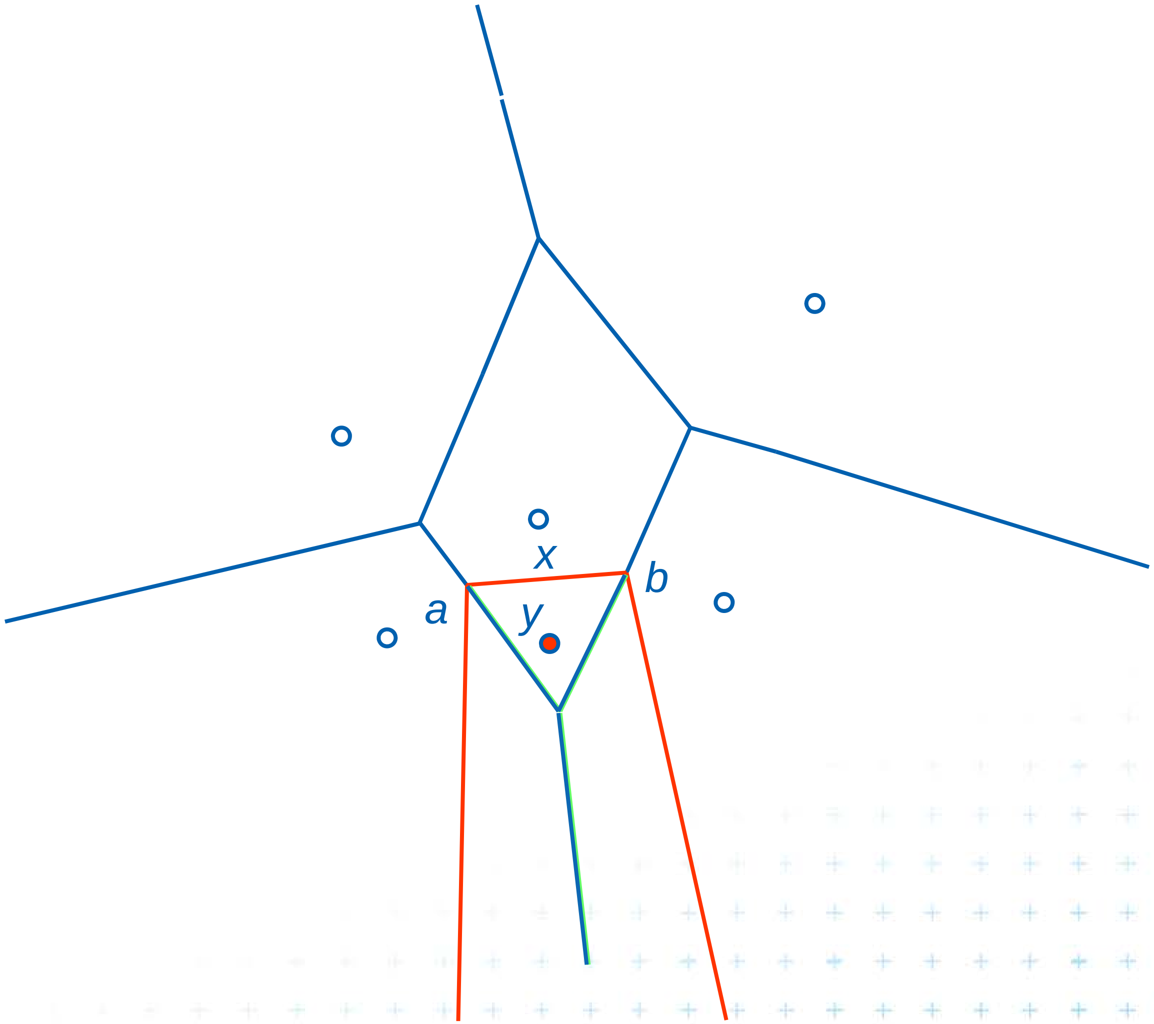
Incremental construction – unbounded cell



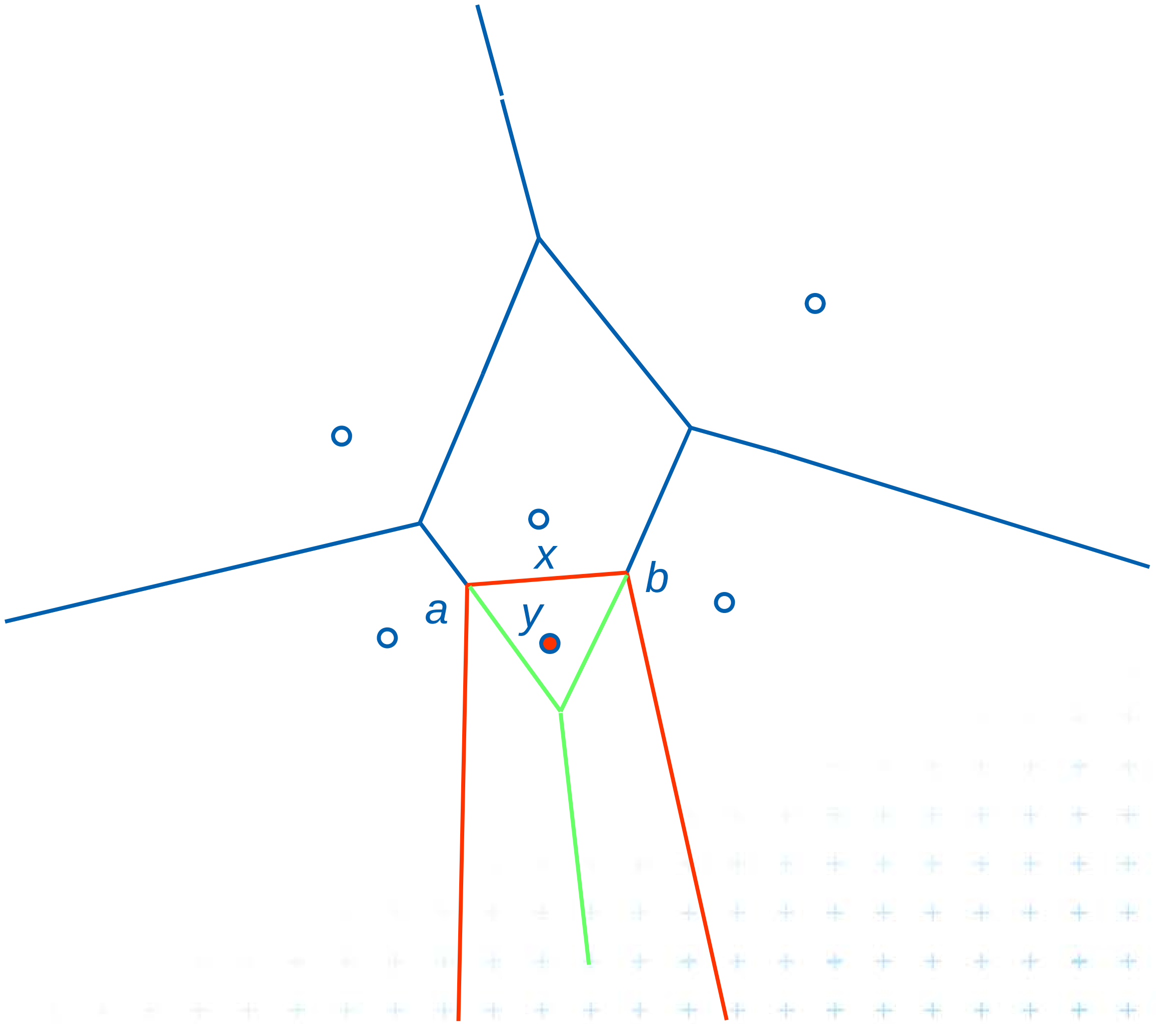
Incremental construction – unbounded cell



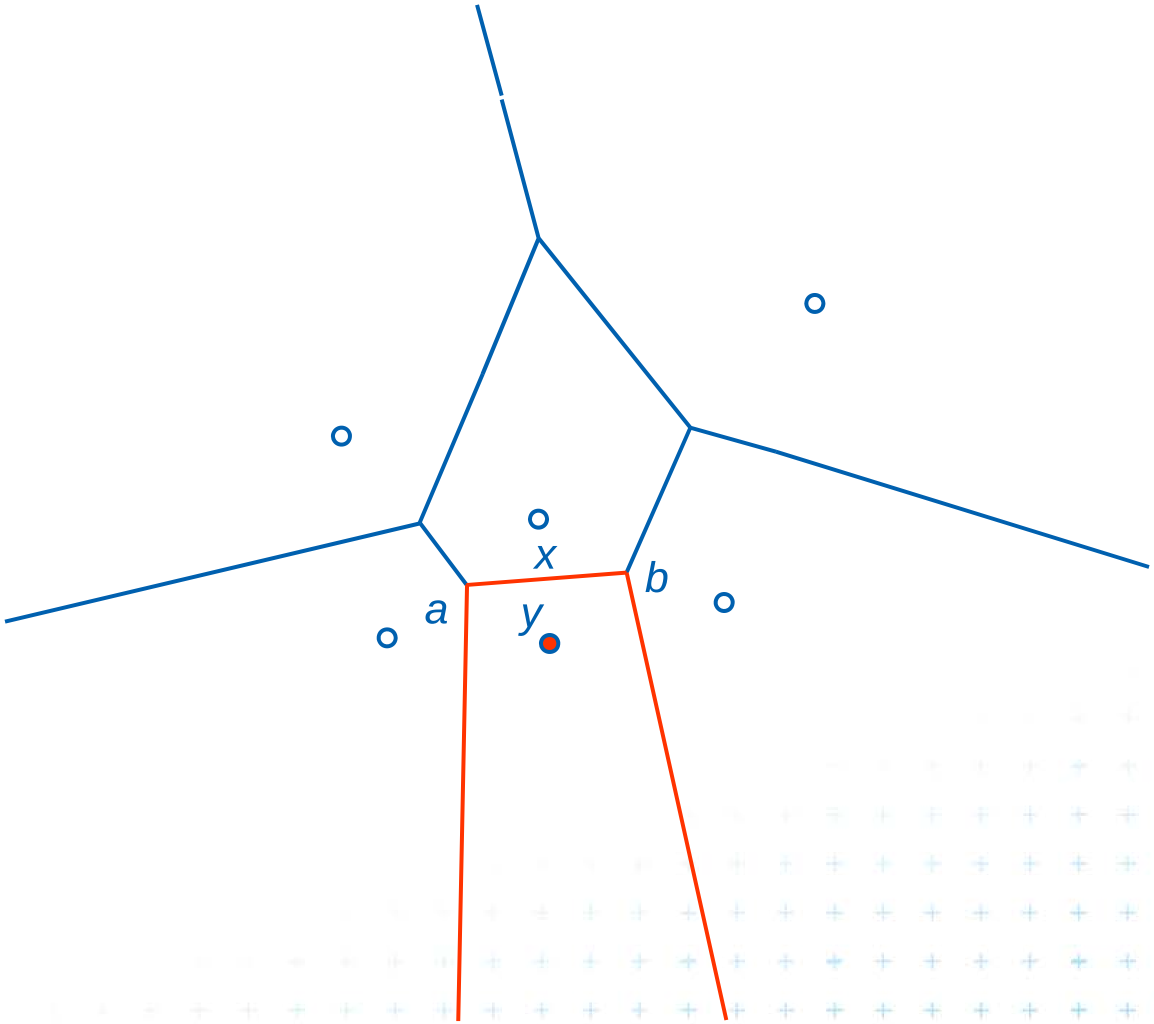
Incremental construction – unbounded cell



Incremental construction – unbounded cell



Incremental construction – unbounded cell



Incremental construction algorithm

InsertPoint(S, Vor(S), y) … **y = a new site**

Input: Point set S, its Voronoi diagram, and inserted point $y \notin S$

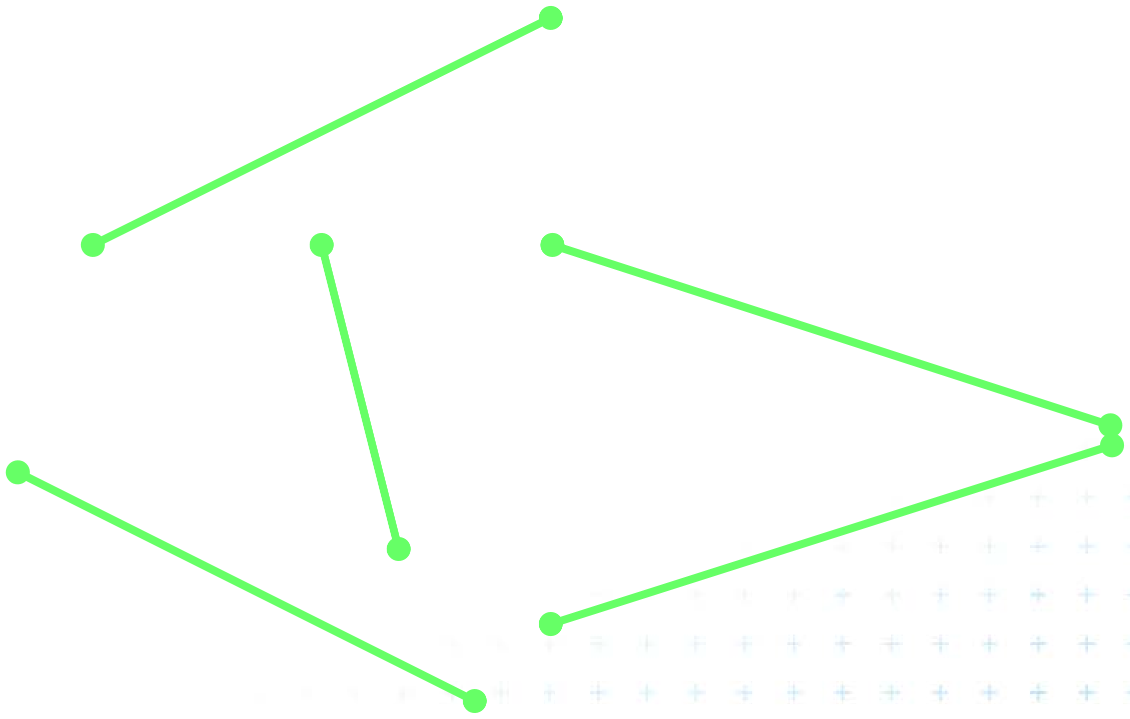
Output: VD after insertion of y

```
1.  Find the site $x$ in which cell point $y$ falls,                       …O(log $n$)
2.  Detect the intersections $\{a,b\}$ of bisector $L(x,y)$ with cell $x$ boundary
    => create the first edge $e = ab$ on the border of site $x$            …O($n$)
3.  Set start intersection point $p = b$, set new intersection $c$ = undef
4.  site $z$ = neighbor site across the border with intersection $b$     …O(1)
5.  while( exists($p$) and $c \neq a$ )  // trace the bisectors from $b$ in one direction
      a.  Detect intersection $c$ of  $L(y,z)$ with border of cell $z$    ⎫
      b.  Report Voronoi edge $pc$                                         ⎬ …O($n^2$)
      c.  $p = c$, $z$=neighbor site across border with intersec. $c$      ⎭
5.  if( c ≠ a ) then // trace the bisectors from $a$ in other direction
      a.  p = a
      b.  Similarly as in steps 3,4,5 with a
```

O(n^2) worst-case, O(n) expected time for some distributions

Voronoi diagram of line segments

Input: $S = \{s_1, \ldots, s_n\}$ = set of n disjoint line segments (sites)



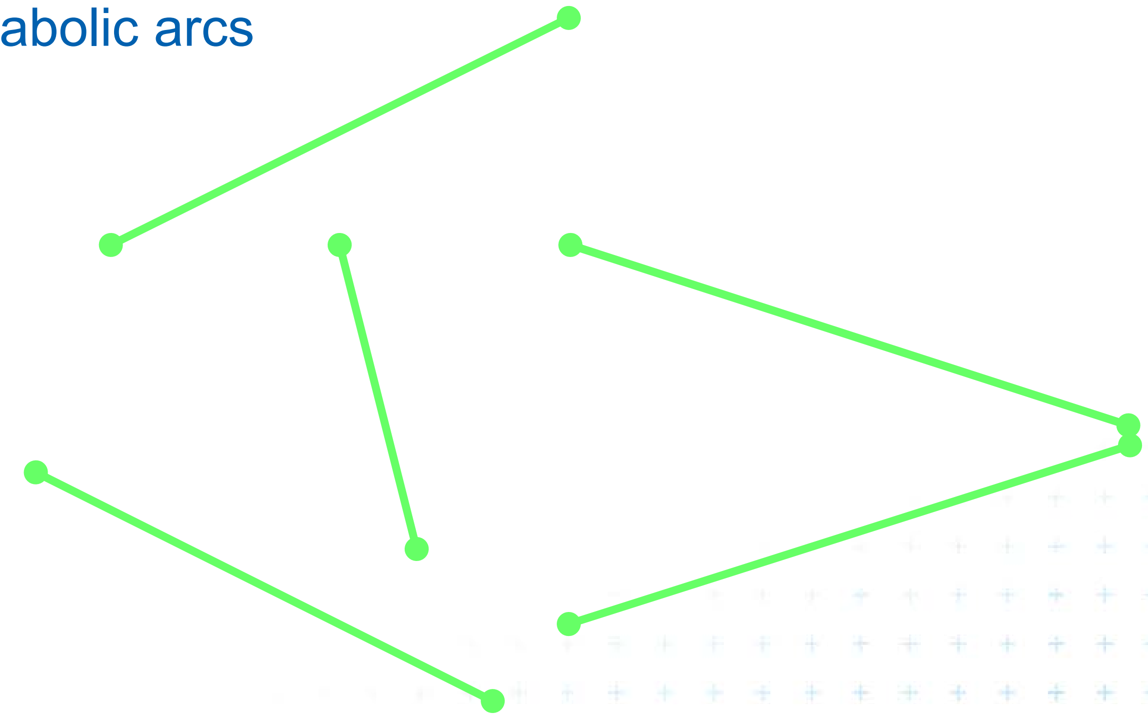
[Berg]

Voronoi diagram of line segments

Input: $S = \{s_1, \ldots, s_n\}$ = set of n disjoint line segments (sites)

VD: line segments

parabolic arcs



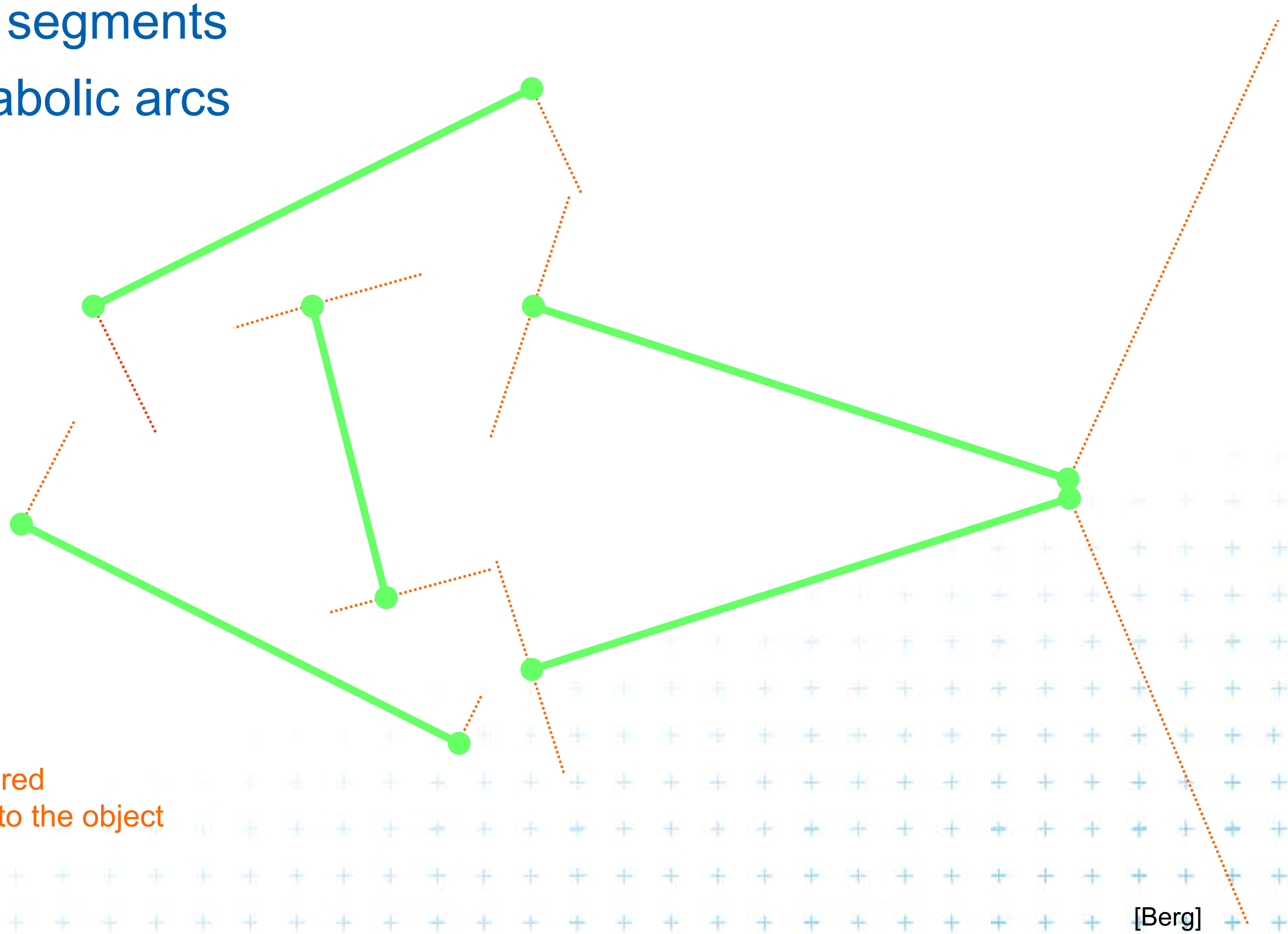
[Berg]

Voronoi diagram of line segments

Input: $S = \{s_1, \ldots, s_n\}$ = set of n disjoint line segments (sites)

VD: line segments

parabolic arcs



Distance measured perpendicularly to the object (line segment)

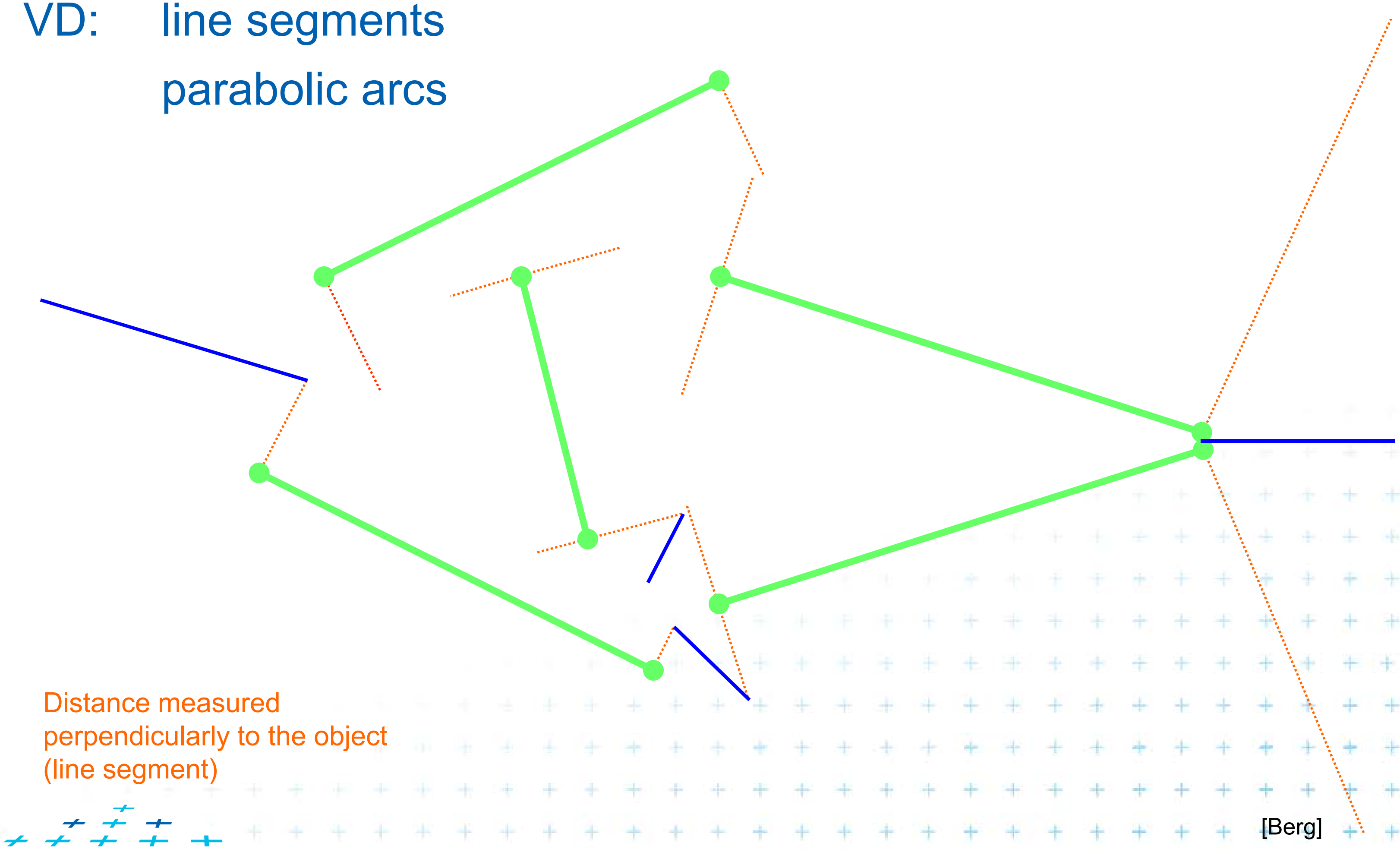
[Berg]

Voronoi diagram of line segments

Input: $S = \{s_1, \ldots, s_n\}$ = set of n disjoint line segments (sites)

VD: line segments

parabolic arcs



Distance measured perpendicularly to the object (line segment)

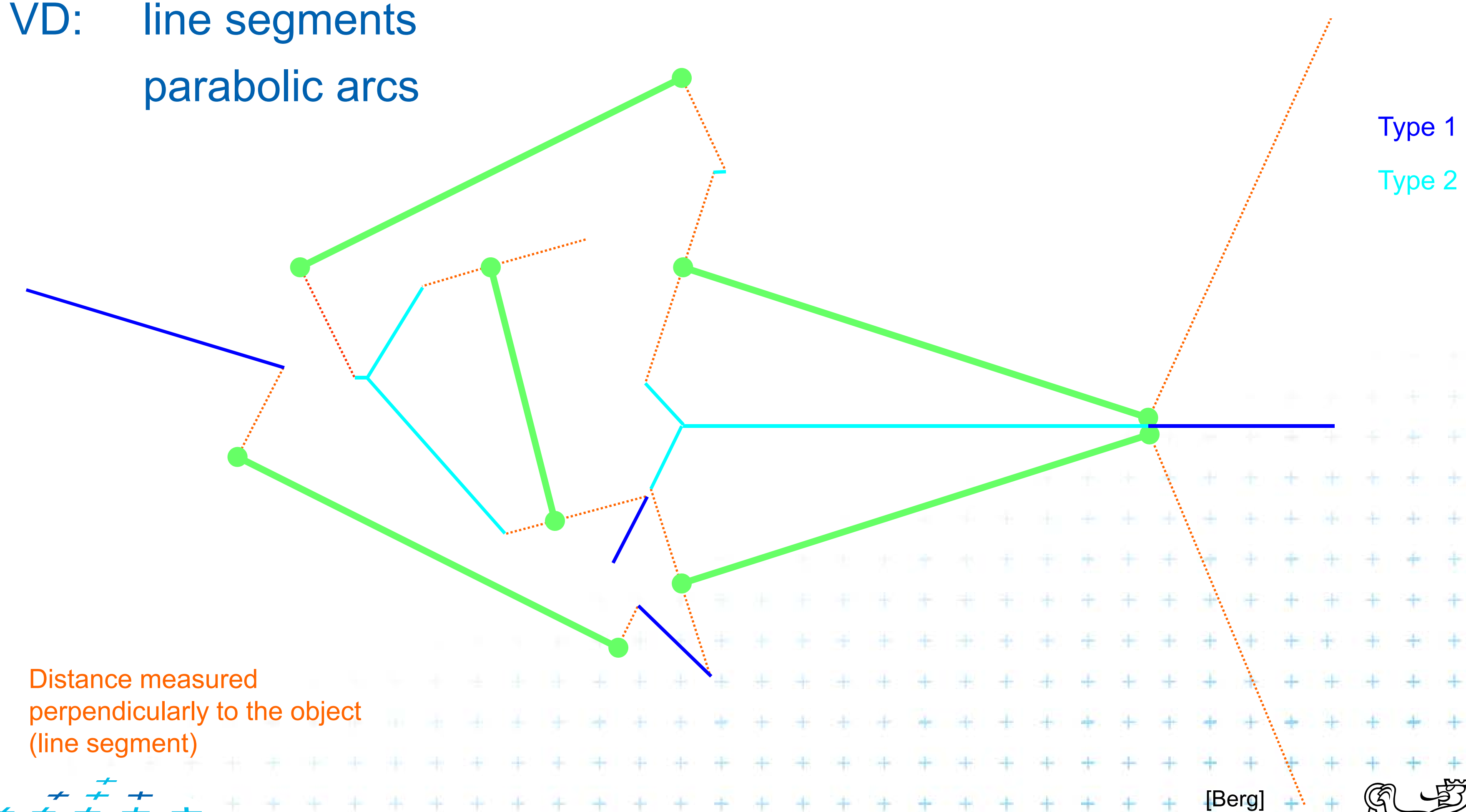
[Berg]

Voronoi diagram of line segments

Input: $S = \{s_1, \ldots, s_n\}$ = set of n disjoint line segments (sites)

VD: line segments

parabolic arcs



Distance measured perpendicularly to the object (line segment)

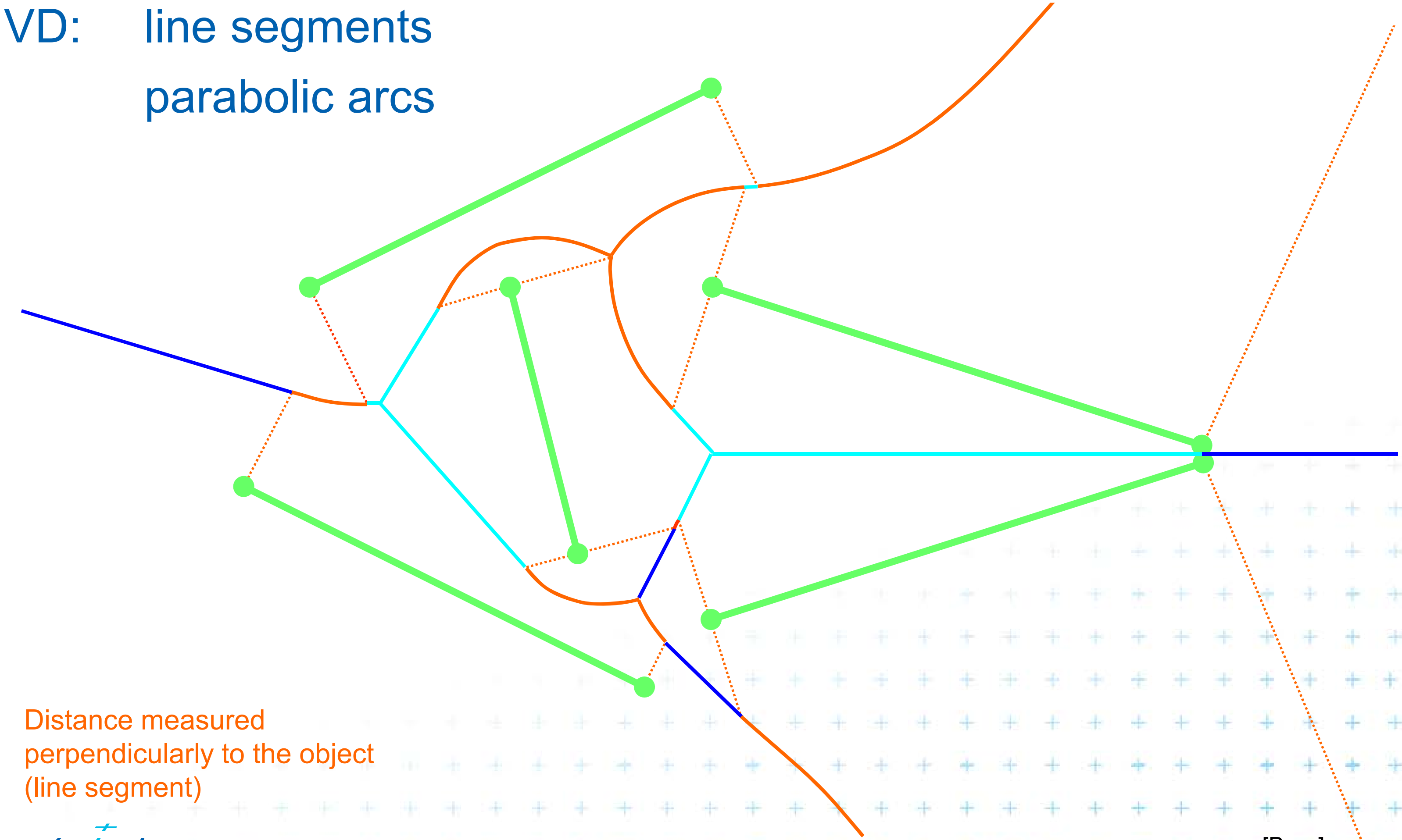
[Berg]

Voronoi diagram of line segments

Input: $S = \{s_1, \ldots, s_n\}$ = set of n disjoint line segments (sites)

VD: line segments

parabolic arcs



Distance measured perpendicularly to the object (line segment)

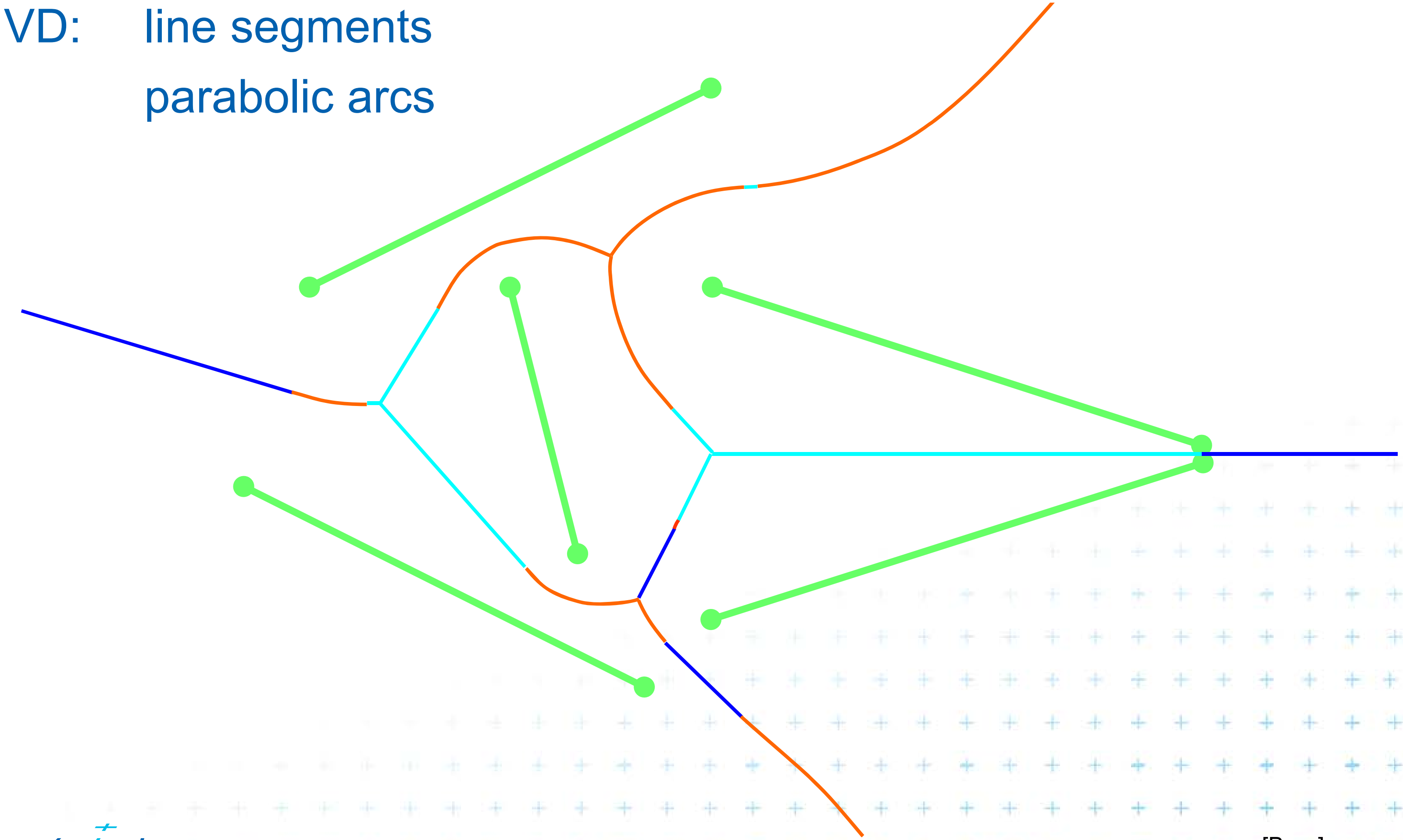
[Berg]

Voronoi diagram of line segments

Input: $S = \{s_1, \ldots, s_n\}$ = set of n disjoint line segments (sites)

VD: line segments

parabolic arcs



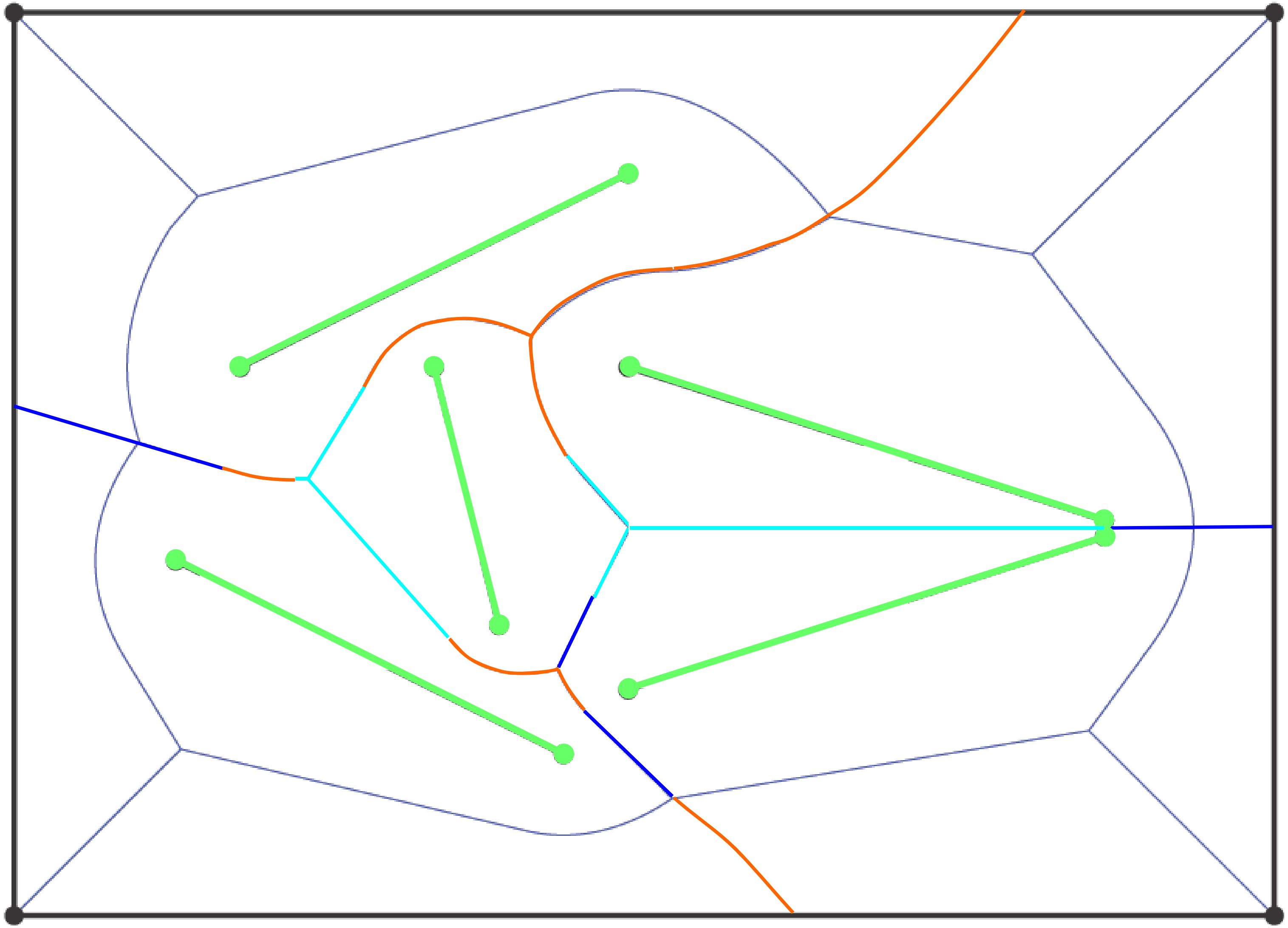
Type 1

Type 2

Type 3

[Berg]

VD of line segments with bounding box



BBOX

=>

standard

DCEL

[Berg]

Bisector of 2 line-segments in detail

- Consists of line segments and parabolic arcs

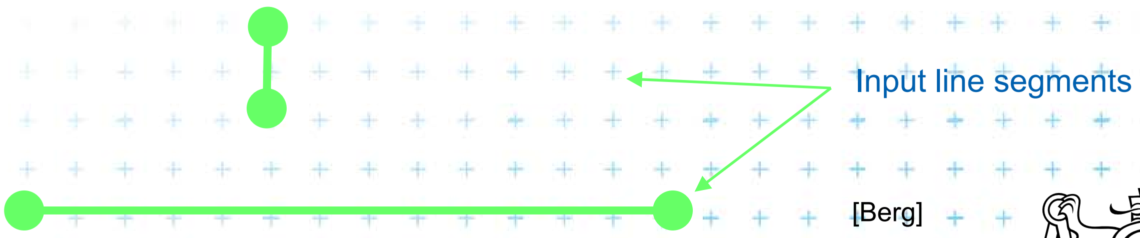
  Distance from point-to-object is measured to the closest point on the object (perpendicularly to the object silhouette)

  - Line segment – bisector of end-points(1) or of interiors(2)
  - Parabolic arc – of point and interior(3) of a line segment

Type 1

Type 2

Type 3



[Berg]

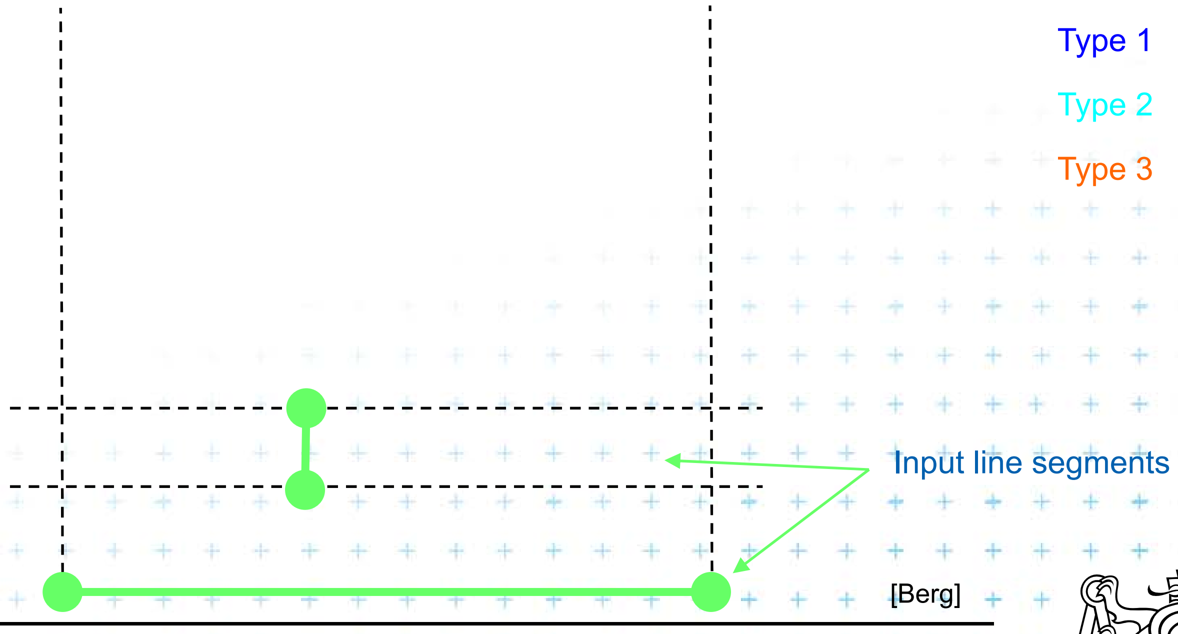
Bisector of 2 line-segments in detail

- Consists of line segments and parabolic arcs

  Distance from point-to-object is measured to the closest point on the object (perpendicularly to the object silhouette)

  – Line segment – bisector of end-points(1) or of interiors(2)

  – Parabolic arc – of point and interior(3) of a line segment



[Berg]

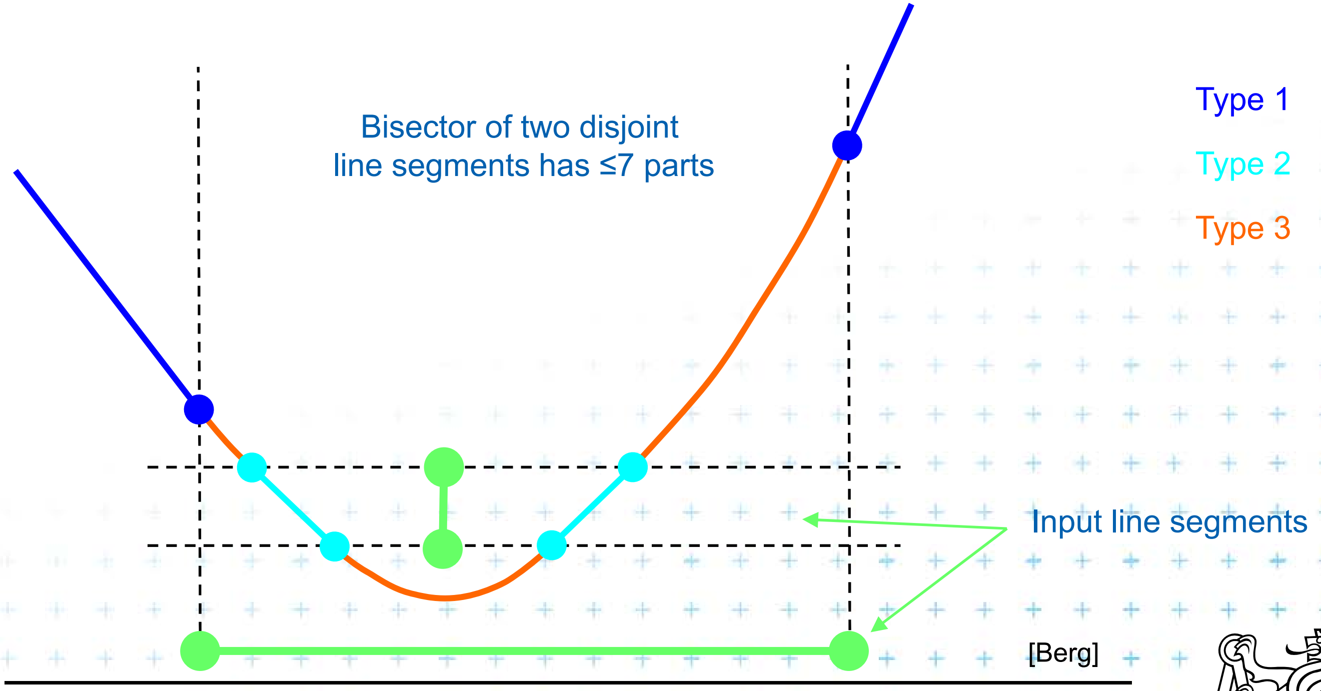
Bisector of 2 line-segments in detail

- Consists of line segments and parabolic arcs

  Distance from point-to-object is measured to the closest point on the object (perpendicularly to the object silhouette)

  – Line segment – bisector of end-points(1) or of interiors(2)

  – Parabolic arc – of point and interior(3) of a line segment



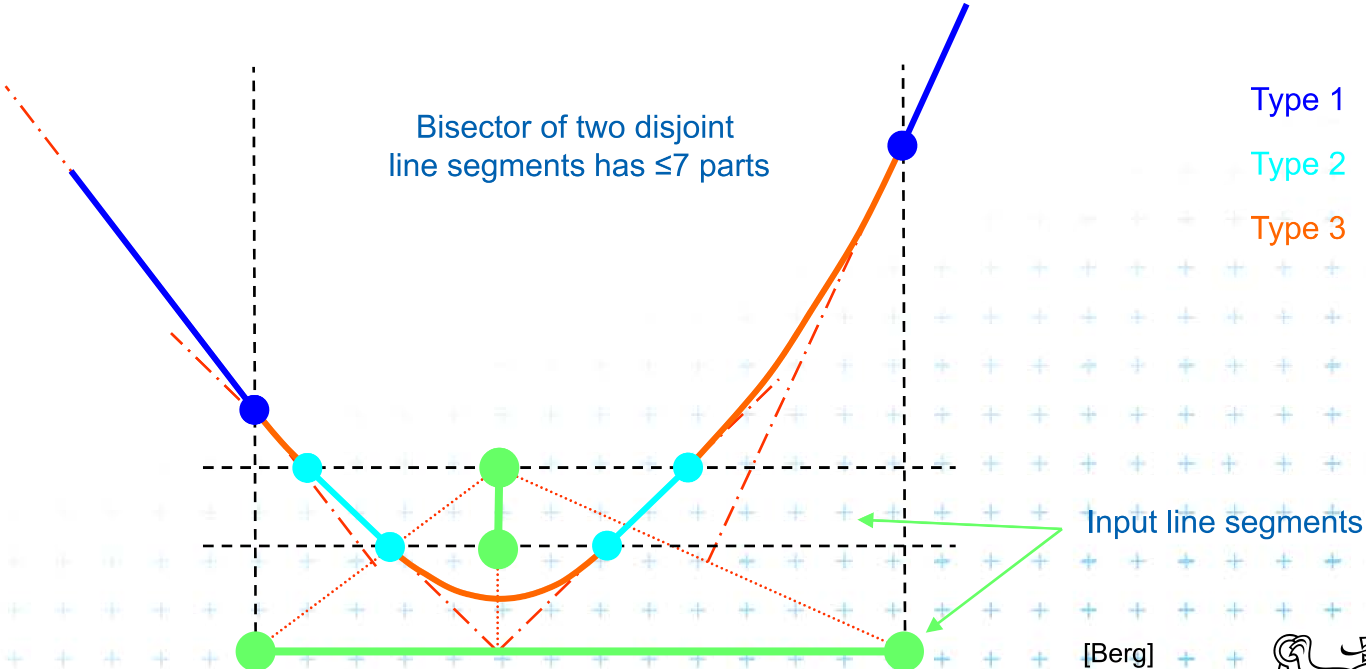
Bisector of 2 line-segments in detail

- Consists of line segments and parabolic arcs

  Distance from point-to-object is measured to the closest point on the object (perpendicularly to the object silhouette)

  – Line segment – bisector of end-points(1) or of interiors(2)

  – Parabolic arc – of point and interior(3) of a line segment



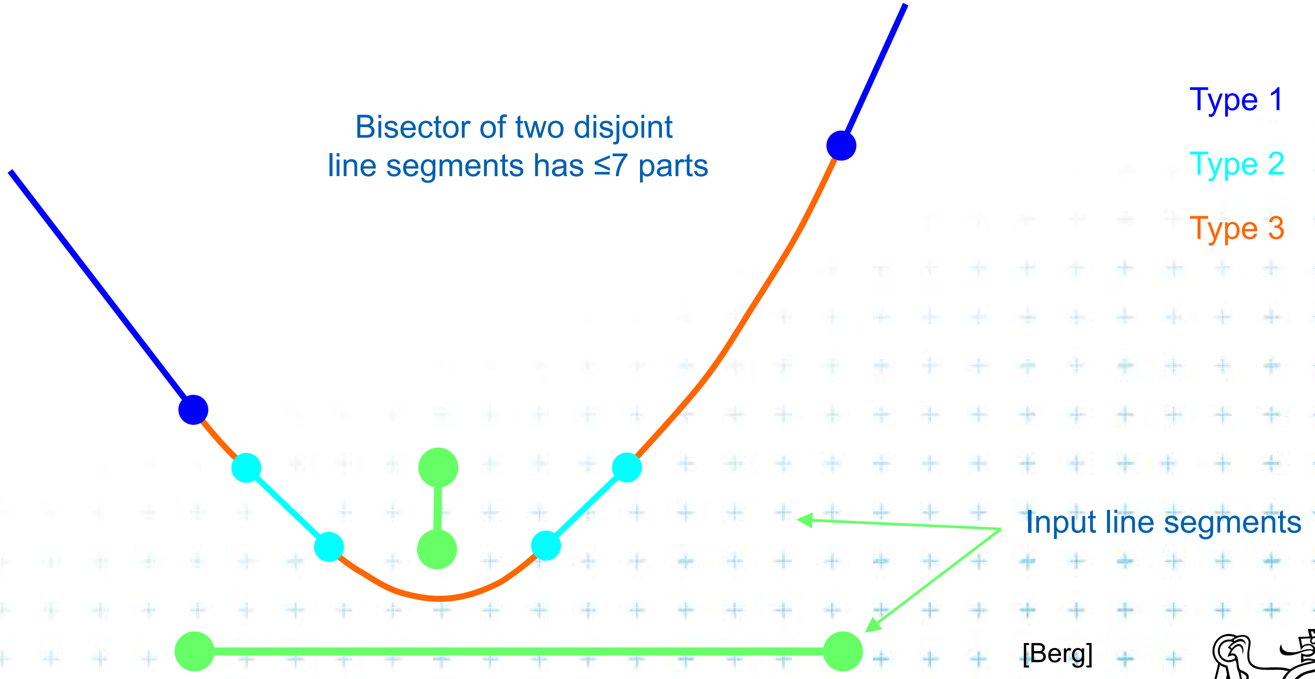
[Berg]

Bisector of 2 line-segments in detail

- Consists of line segments and parabolic arcs

  Distance from point-to-object is measured to the closest point on the object (perpendicularly to the object silhouette)

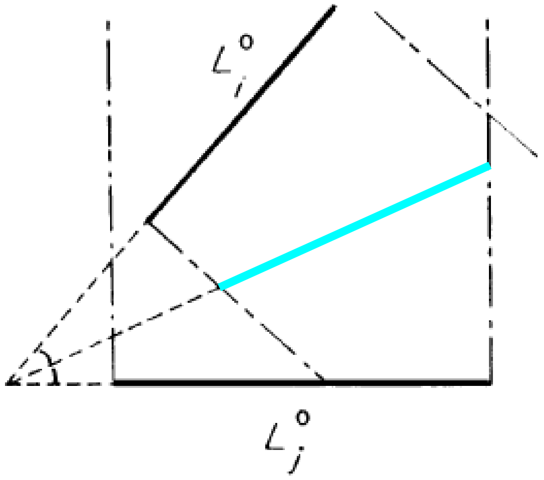
  - Line segment – bisector of end-points(1) or of interiors(2)
  - Parabolic arc – of point and interior(3) of a line segment



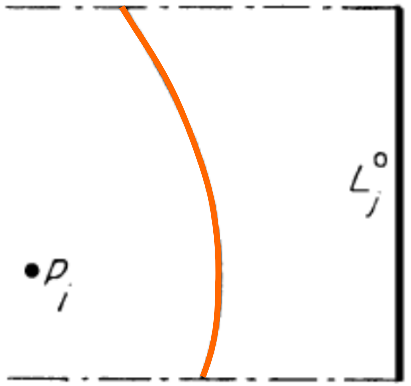
[Berg]

Bisector in greater details

Type 2



Type 3



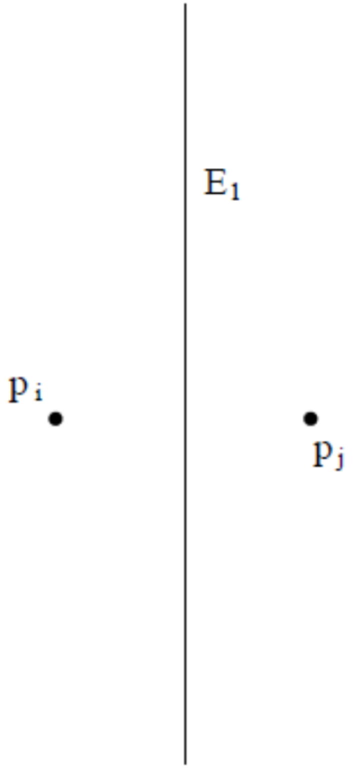
[Reiberg]

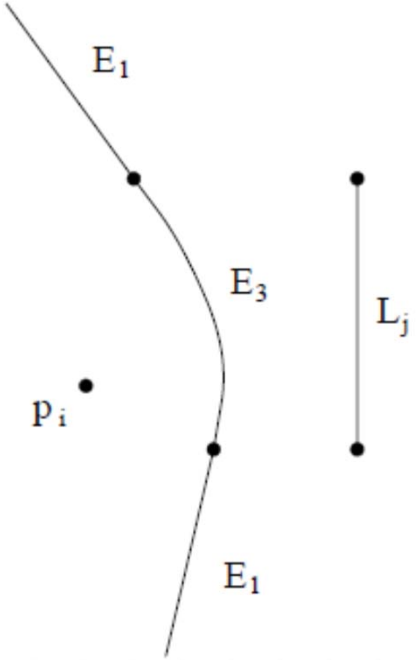
Bisector of two line segment interiors (in intersection of perpendicular slabs only)

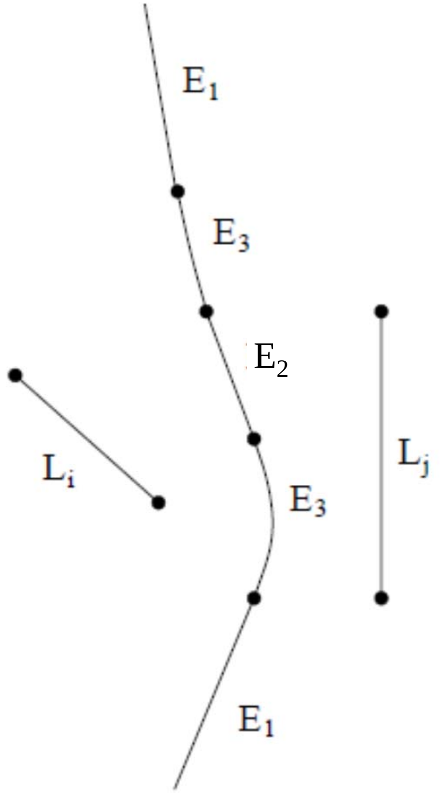
Bisector of (end-)point and line segment interior

VD of points and line segments examples

2 points Point & segment 2 line segments



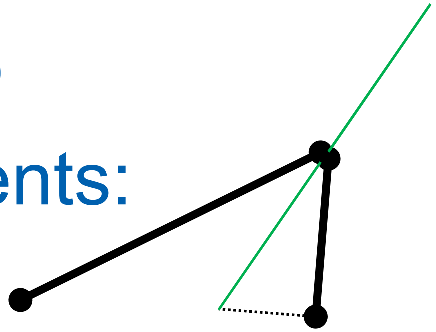


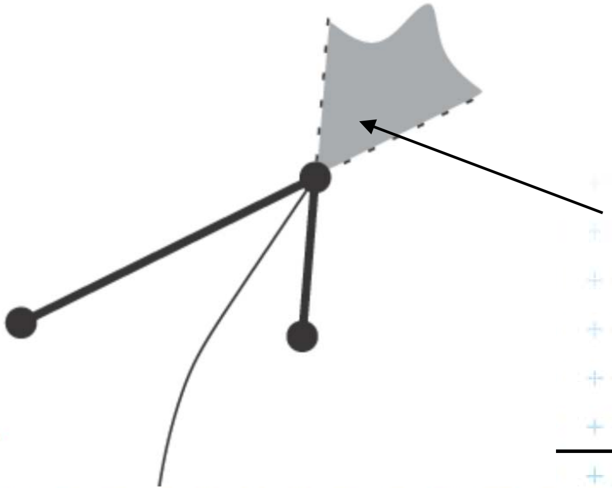


[Reiberg]

Voronoi diagram of line segments

- More complex bisectors of line segments
  - – VD contains line segments and parabolic arcs
- Still combinatorial complexity of O(n)
- Assumptions on the input line segments:
  - – non-crossing
  - – strictly disjoint end-points (slightly shorten the segm.)

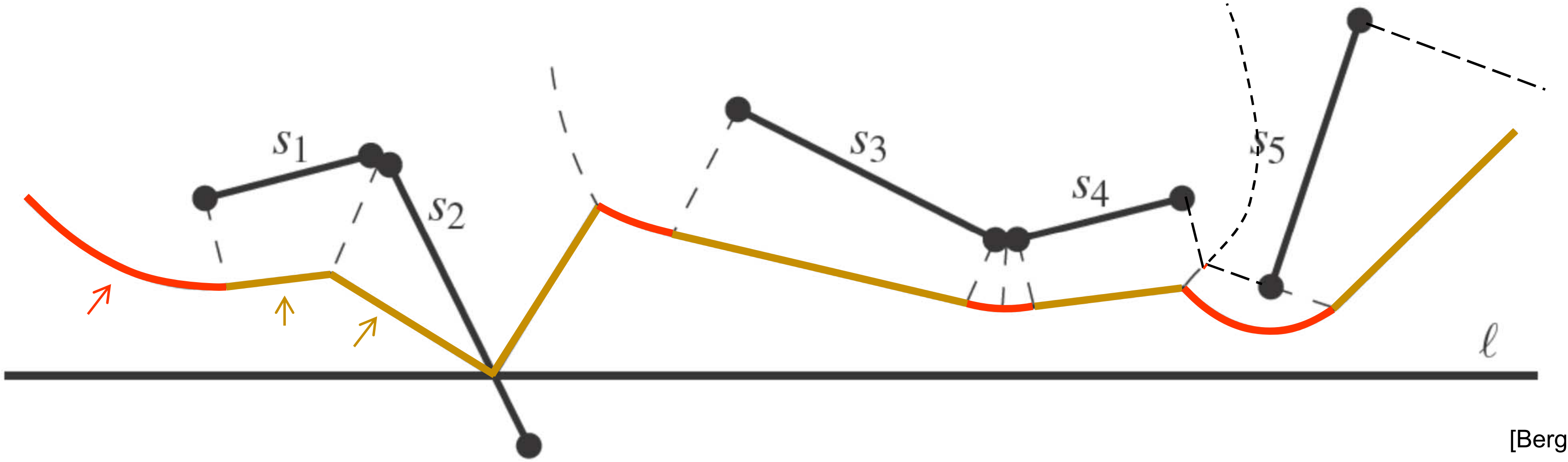




if(we allow touching segments)

Shared endpoints cause complication: The whole region is equally close to two line segments

Shape of Beach line for line segments



[Berg]

= Points with distance to the closest site above sweep line *l* equal to the distance to *l*

- Beach line contains
  - *parabolic arcs* when closest to a site end-point
  - *straight line segments* when closest to a site interior (or just the part of the site interior above *l* if the site *s* intersects *l*)

(This is the shape of the beach line)

Beach line breakpoints types

Breakpoint *p* is equidistant from *l* and equidistant and closest to:

points

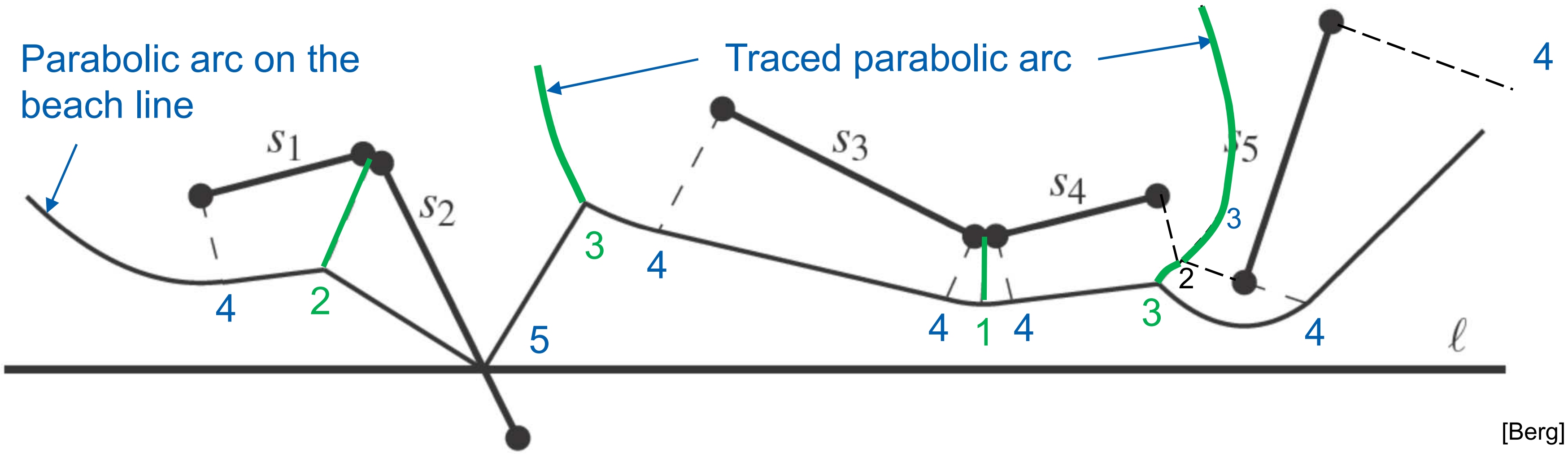
1. two site end-points => *p* traces a VD line segment

segments

2. two site interiors => *p* traces a VD line segment
3. end-point and interior => *p* traces a VD parabolic arc
4. one site end-point => *p* traces a line segment (border of the slab perpendicular to the site)
5. site interior intersects the scan line *l* => *p* = intersection, traces the input line segment

Cases 4 and 5 involve only one site and therefore do not form a Voronoi diagram edge (are used by alg.only)

Breakpoints types and what they trace



[Berg]

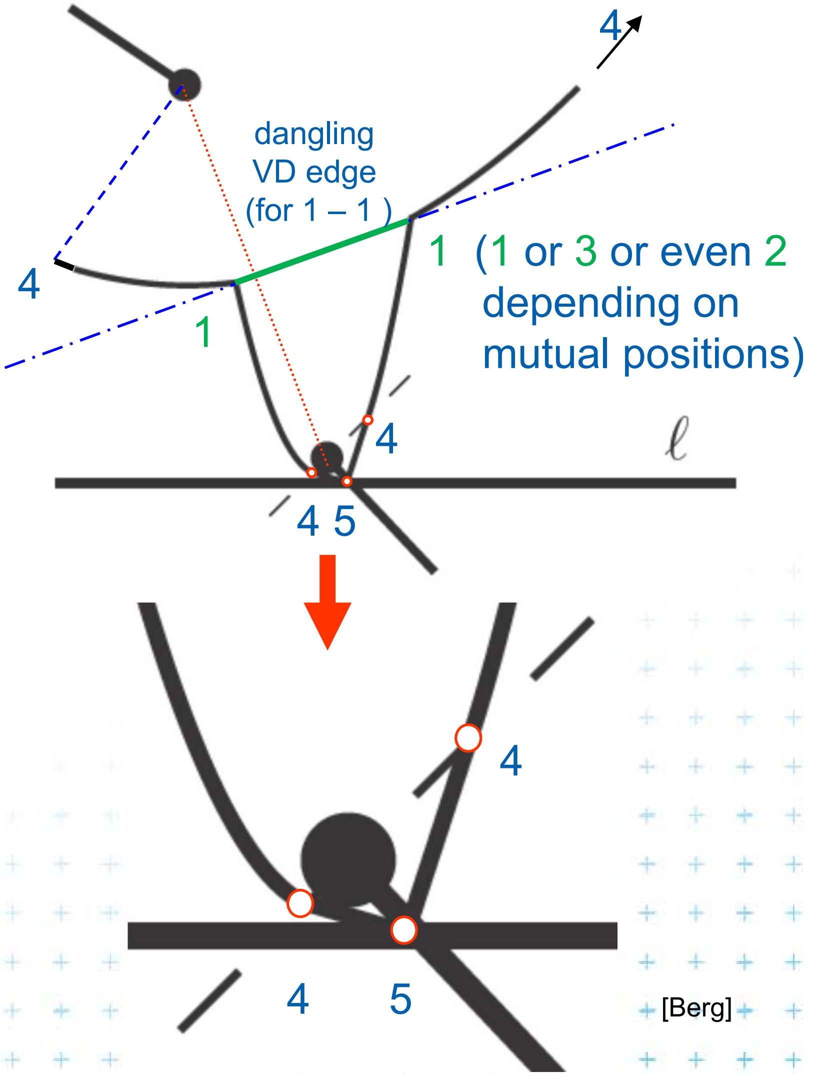
- 1,2 trace a Voronoi line segment (part of VD edge) DRAW
- 3 traces a Voronoi parabolic arc (part of VD edge) DRAW
- 4,5 trace a line segment (used only by the algorithm) MOVE
  - 4 limits the slab perpendicular to the line segment
  - 5 traces the intersection of input segment with a sweep line

(This is the shape of the traced VD arcs)

Site event – sweep line reaches an endpoint

I. At upper endpoint of ●⟍●

- Arc above is split into two
- four new arcs are created (2 segments + 2 parabolas)
- Breakpoints for two segments are of type 4-5-4
- Breakpoints for parabolas depend on the surrounding sites
  - Type 1 for two end-points
  - Type 3 for endpoint and interior
  - etc…



dangling VD edge (for 1 – 1)

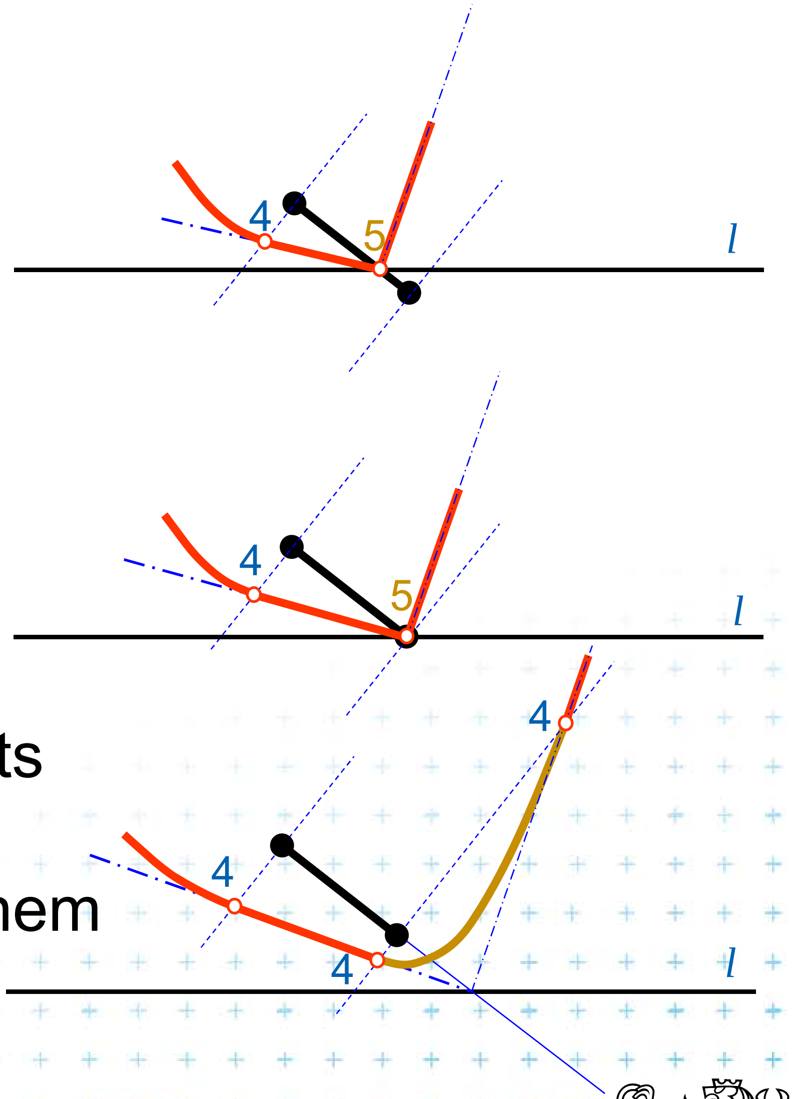
(1 or 3 or even 2 depending on mutual positions)

[Berg]

Site event – sweep line reaches an endpoint

II. At lower endpoint of

- Intersection with interior (breakpoint of type 5)
- is replaced by two breakpoints (of type 4) with parabolic arc between them

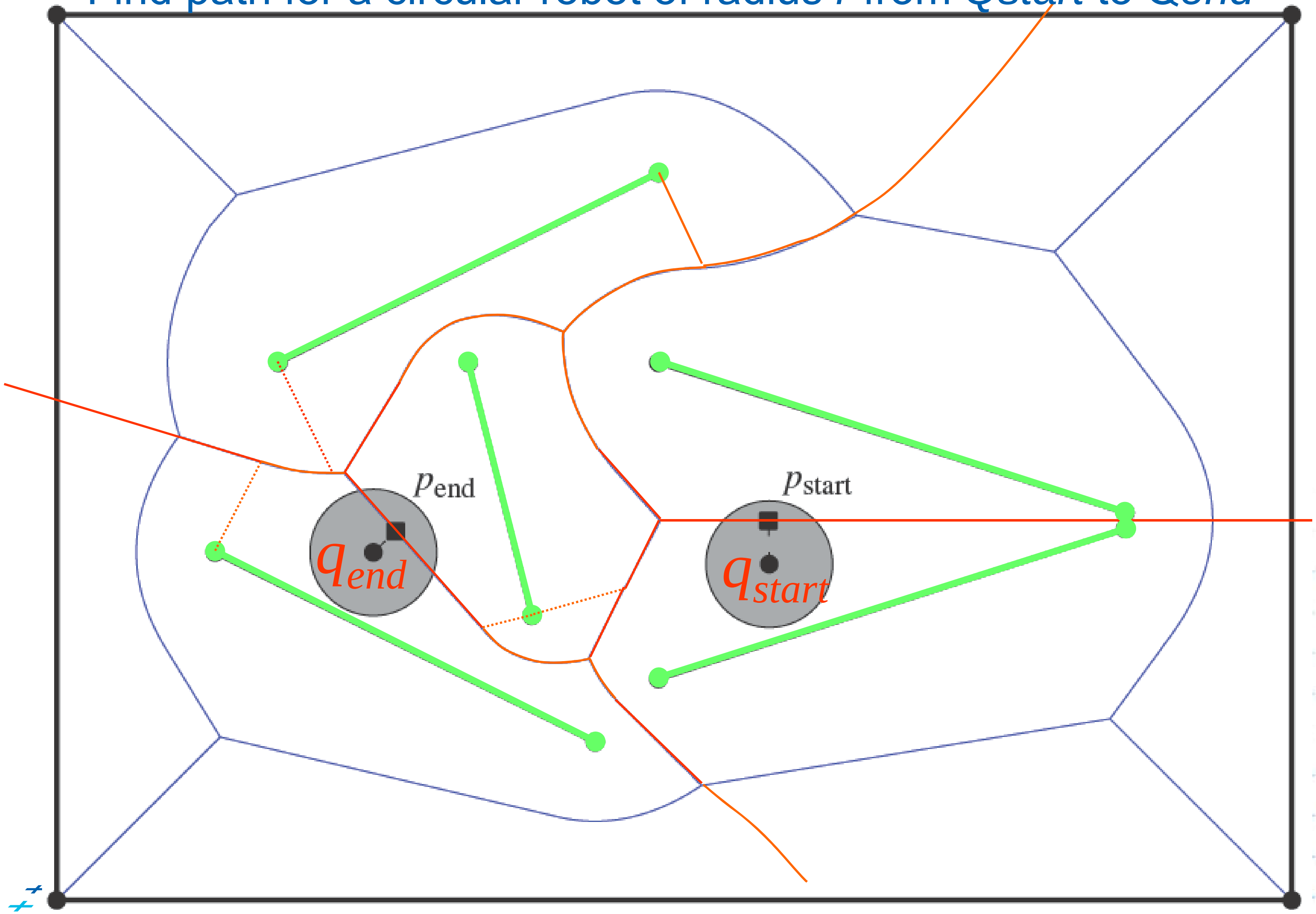


Circle event – lower point of circle of 3 sites

- Two breakpoints meet (on the beach-line)
- Solution depends on their type
  - Any of first three types (1,2,or 3) meet
    - 3 sites involved – Voronoi vertex created
  - Type 4 with something else
    - two sites involved – breakpoint changes its type
    - Voronoi vertex not created (Voronoi edge may change its shape)
  - Type 5 with something else
    - never happens for disjoint segments (meet with type 4 happens before)

Motion planning example - retraction Rušení hran

Find path for a circular robot of radius *r* from *Qstart* to *Qend*

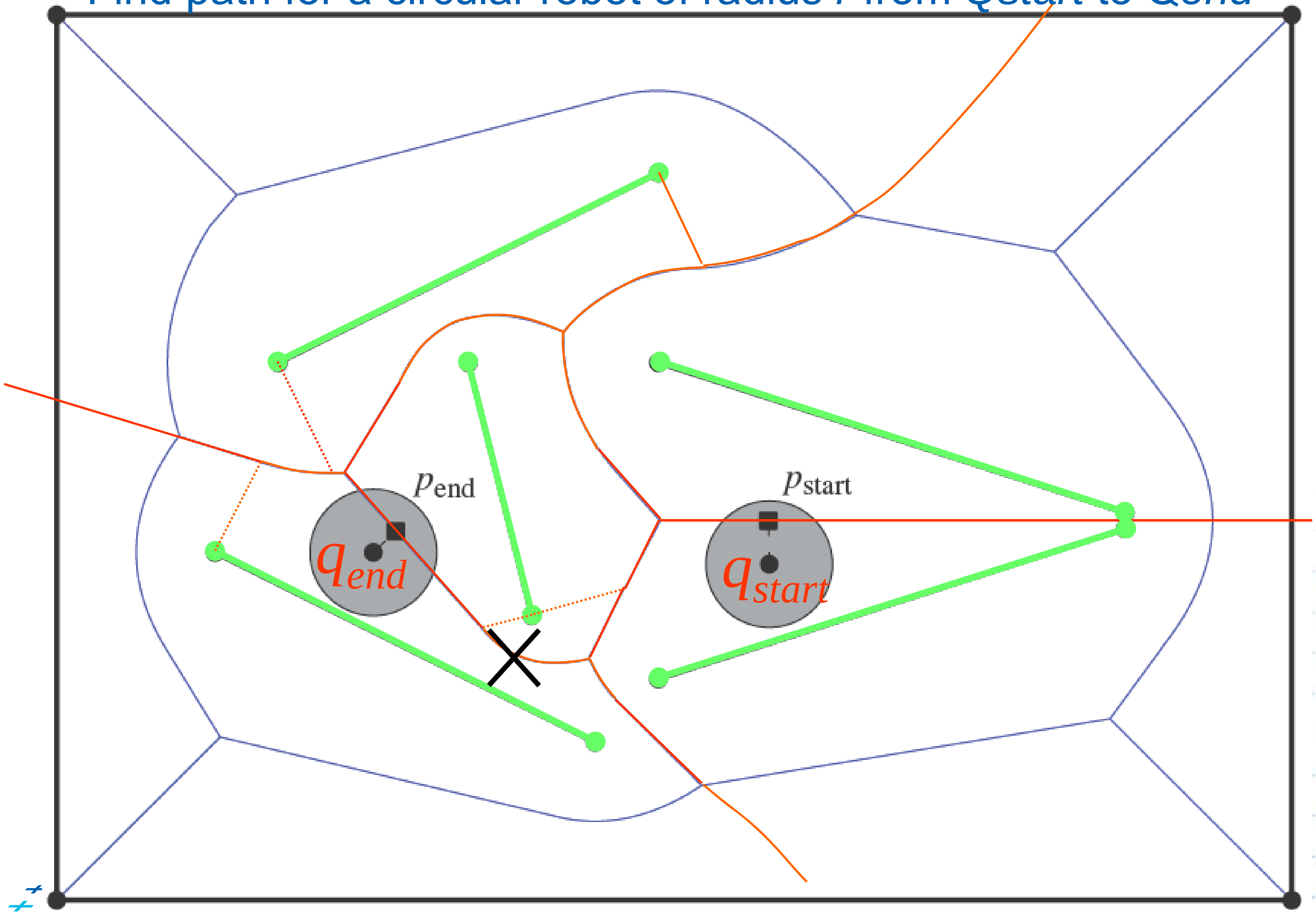


[Berg]

Motion planning example - retraction

Rušení hran

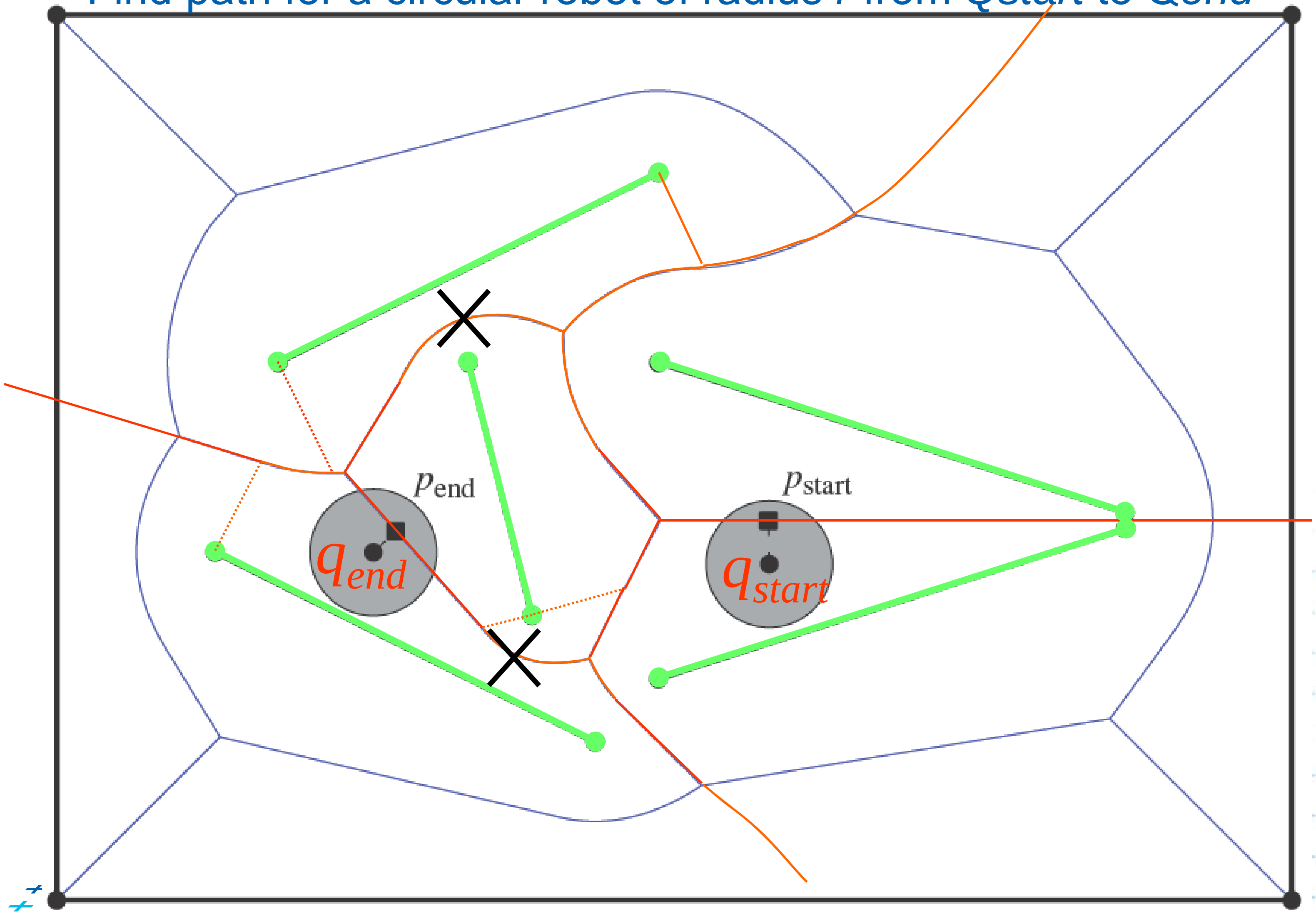
Find path for a circular robot of radius *r* from *Qstart* to *Qend*



[Berg]

Motion planning example - retraction Rušení hran

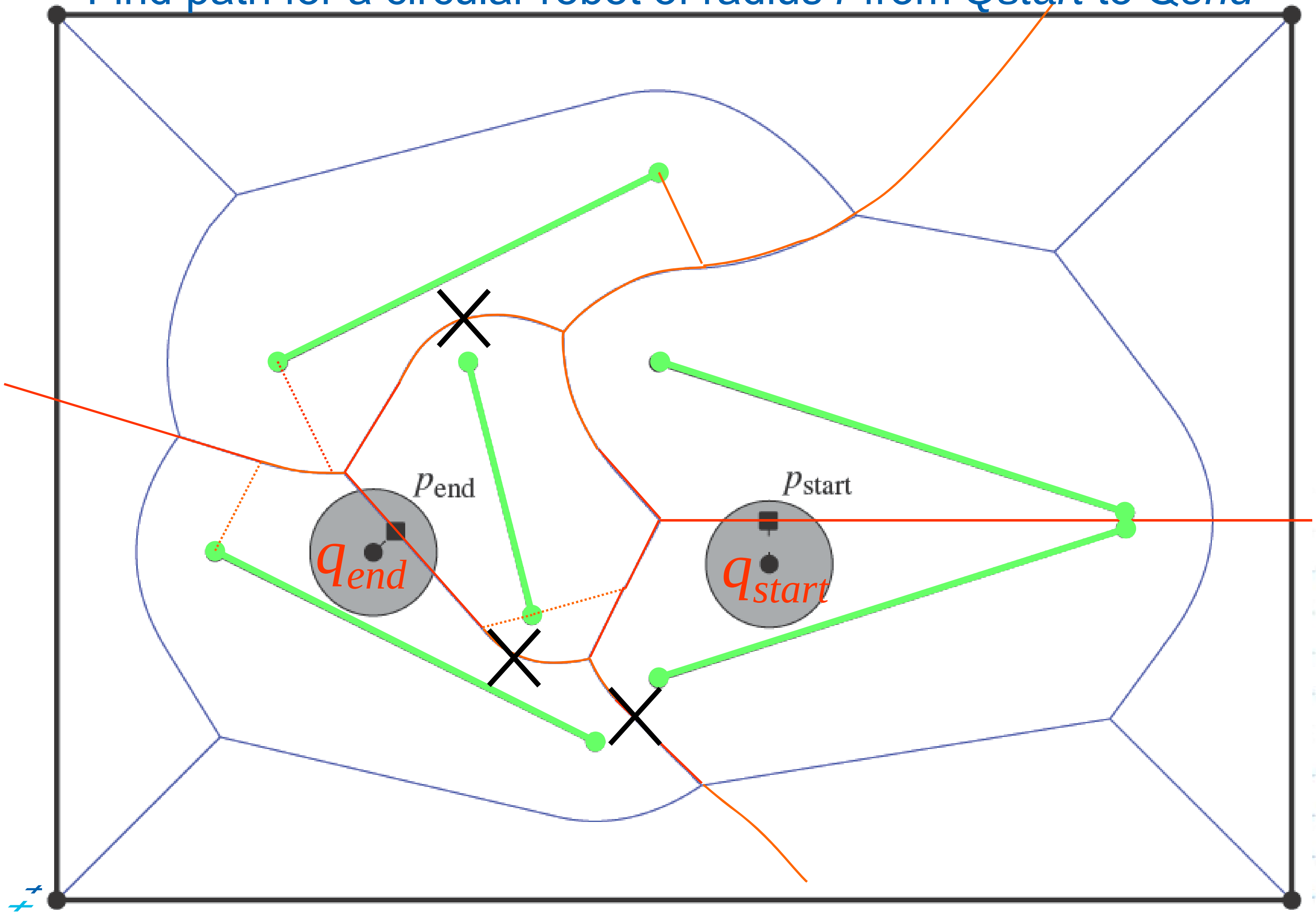
Find path for a circular robot of radius *r* from *Qstart* to *Qend*



[Berg]

Motion planning example - retraction Rušení hran

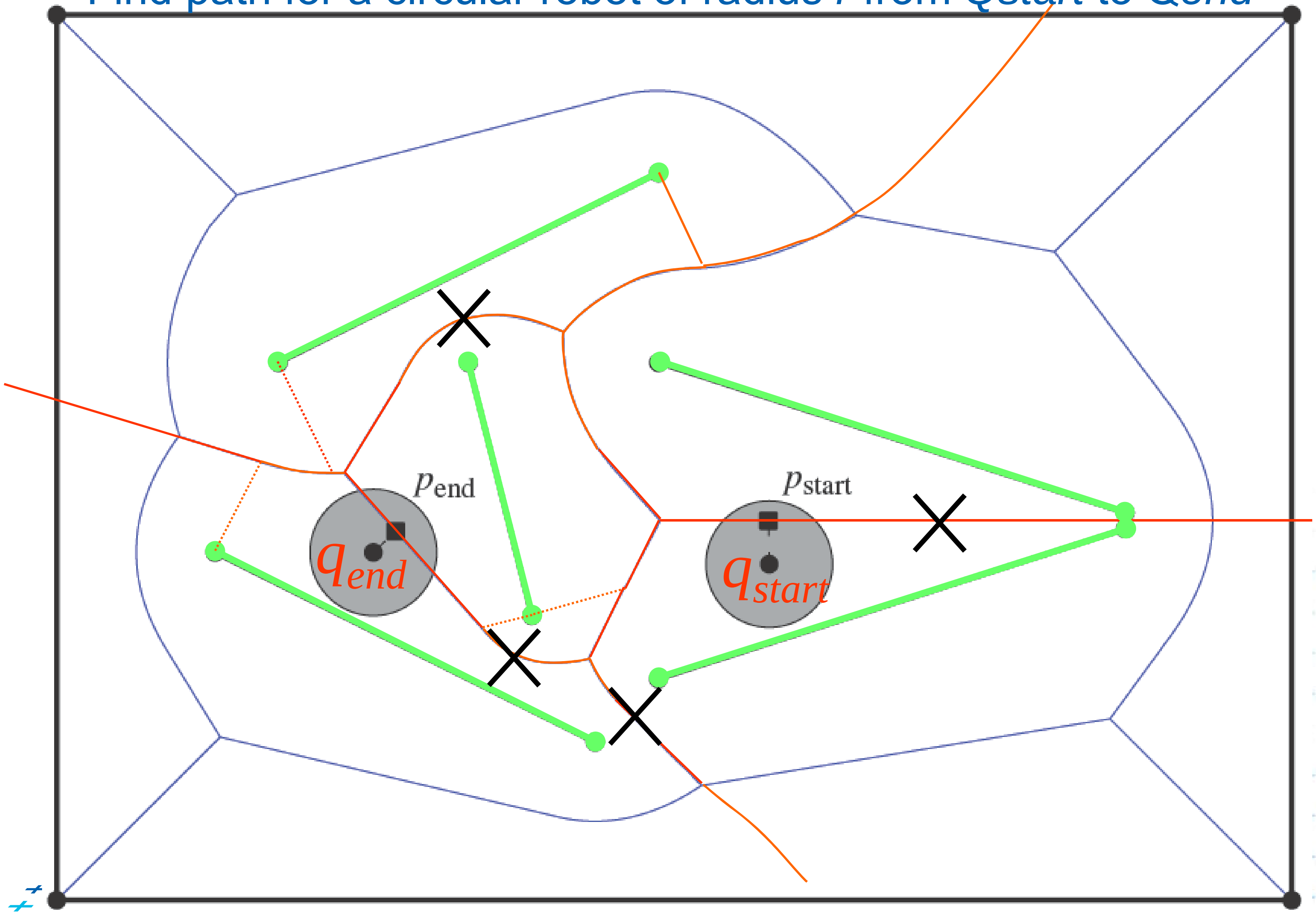
Find path for a circular robot of radius *r* from *Qstart* to *Qend*



[Berg]

Motion planning example - retraction Rušení hran

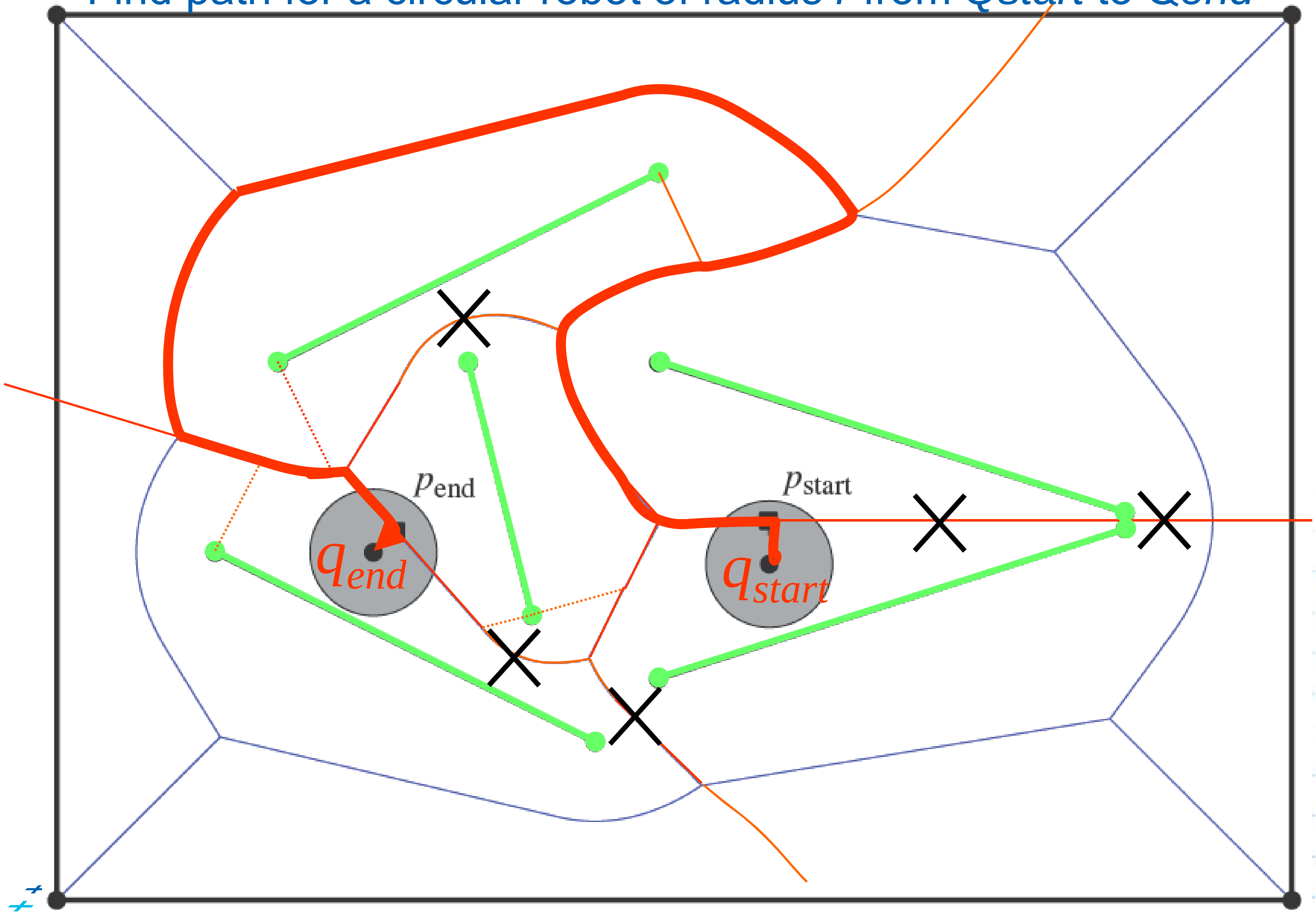
Find path for a circular robot of radius *r* from *Qstart* to *Qend*



[Berg]

Motion planning example - retraction Rušení hran

Find path for a circular robot of radius *r* from *Qstart* to *Qend*



[Berg]

Motion planning example - retraction Rušení hran

Find path for a circular robot of radius *r* from Q_{start} to Q_{end}

- Create Voronoi diagram of line segments,
  take it as a graph
- Project Q_{start} to P_{start} *on VD and* Q_{end} to P_{end}
- Remove segments with distance to sites smaller than radius *r* of a robot
- Depth first search if path from P_{start} to P_{end} *exists*
- Report path $Q_{start}\,P_{start}$*…path…* P_{end} to Q_{end}

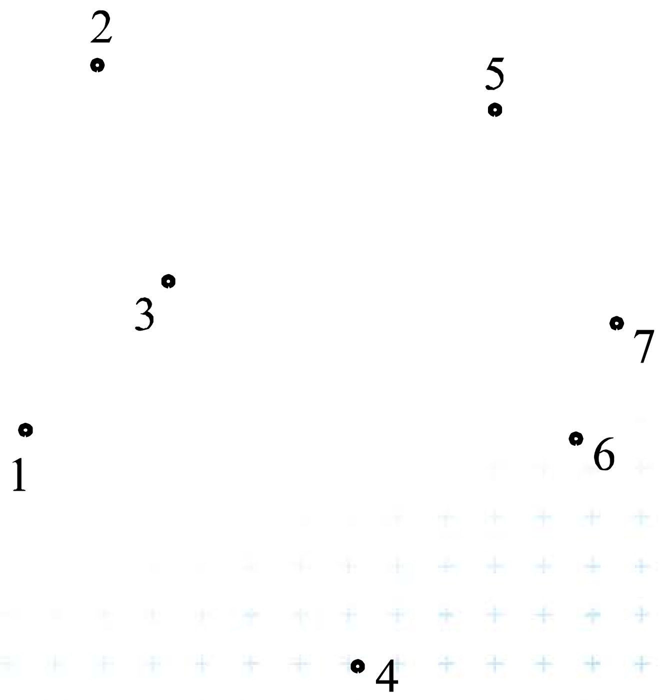
- $O(n \log n)$ time using $O(n)$ storage

Order-2 Voronoi diagram

[Nandy]

Order-2 Voronoi diagram

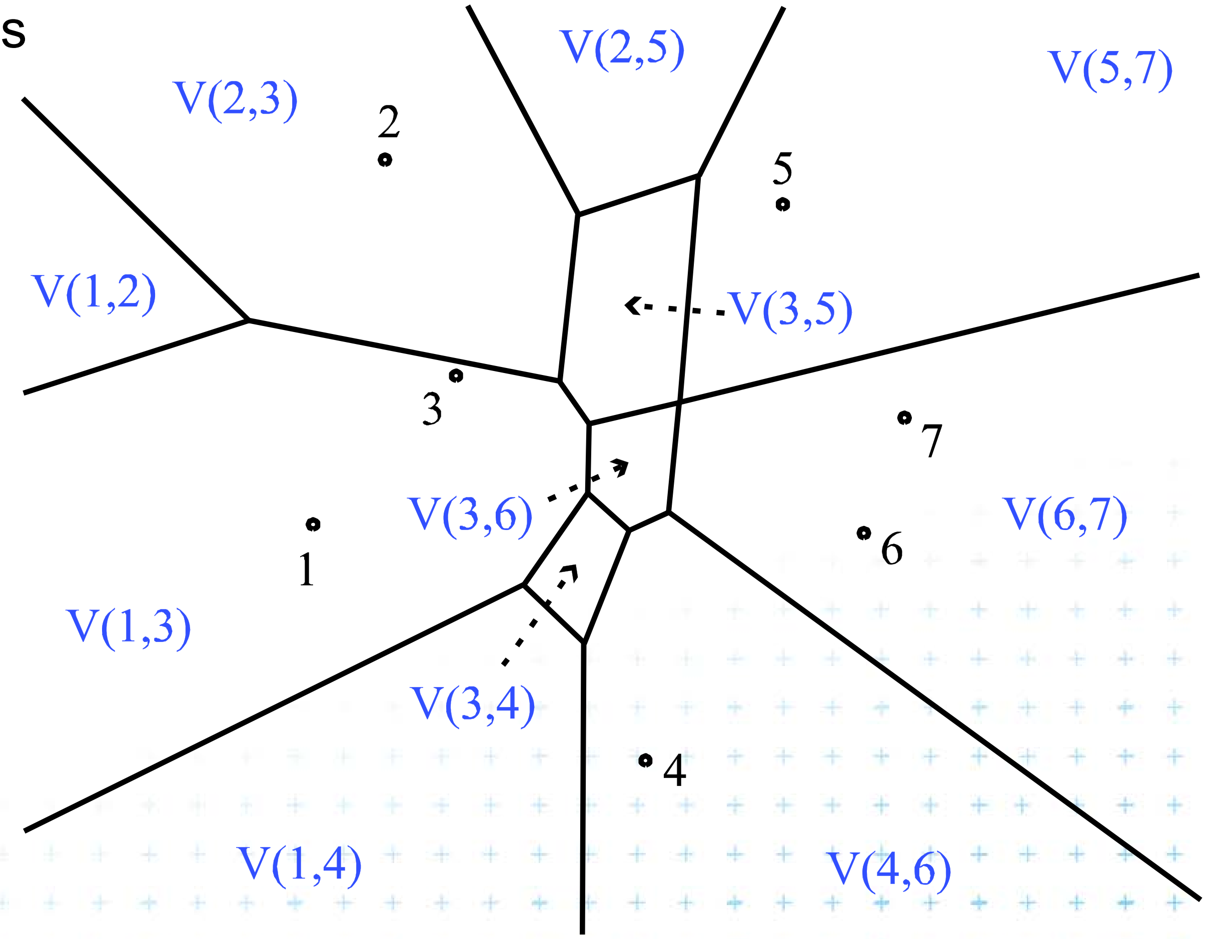
$V(p_i,p_j)$: the set of points of the plane closer to each of p_i and p_j than to any other site



[Nandy]

Order-2 Voronoi diagram

$V(p_i,p_j)$: the set of points of the plane closer to each of p_i and p_j than to any other site



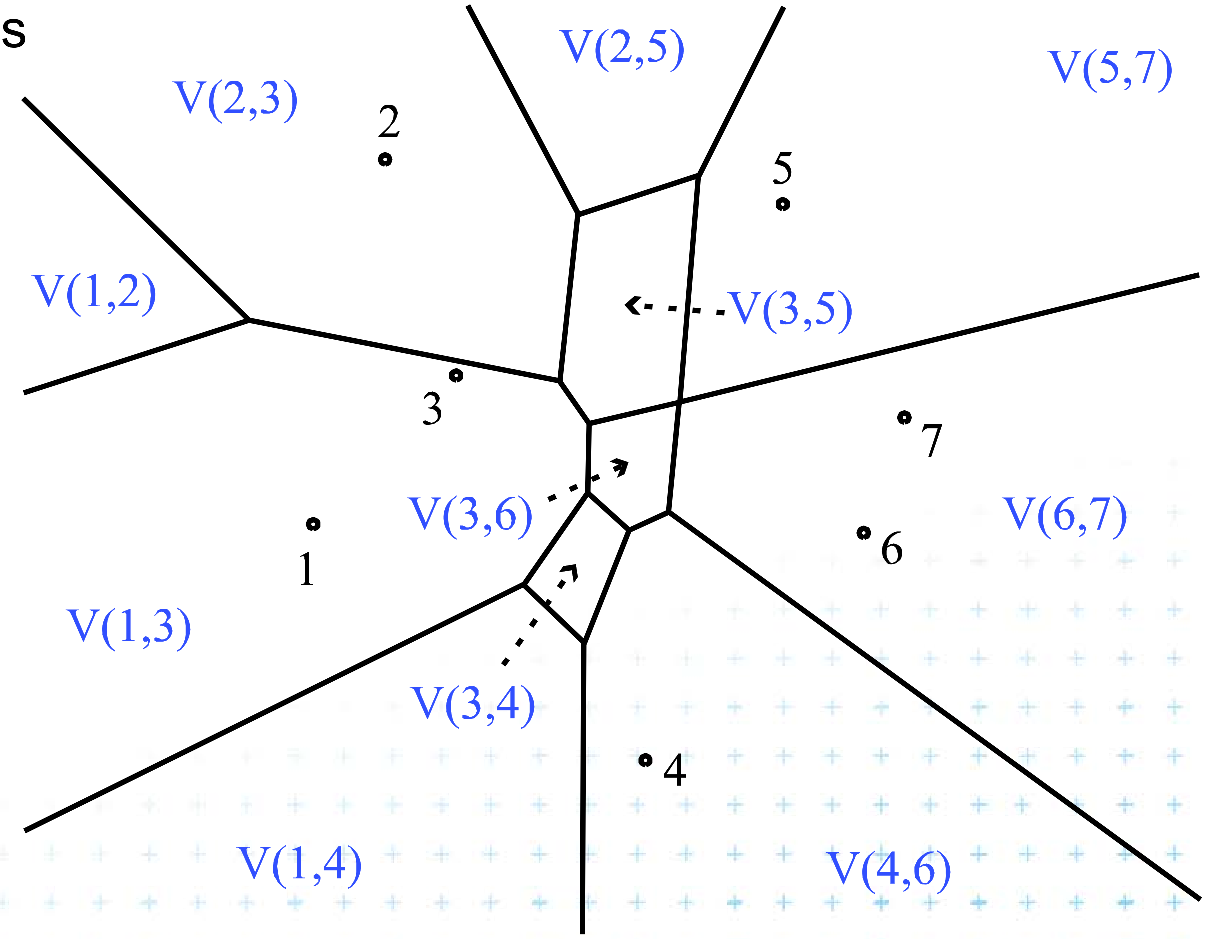
[Nandy]

Order-2 Voronoi diagram

$V(p_i,p_j)$: the set of points of the plane closer to each of p_i and p_j than to any other site

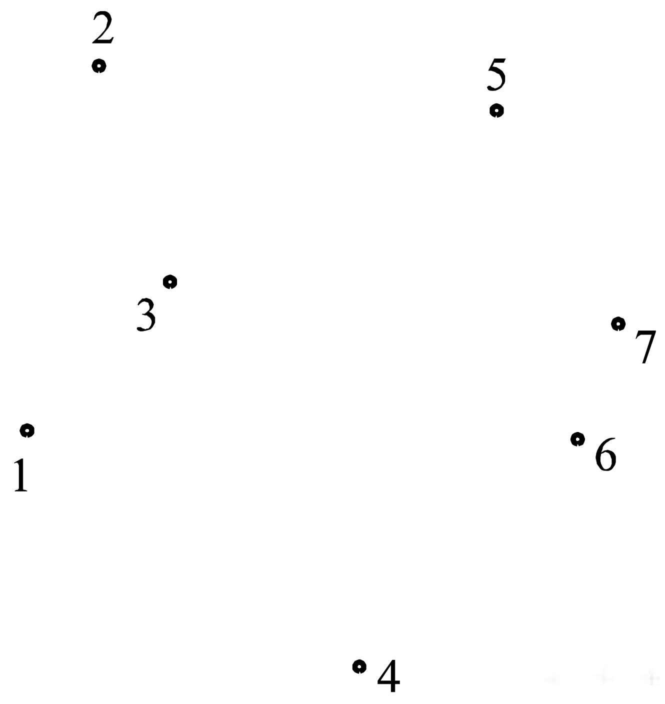
Property

The order-2 Voronoi regions are convex



[Nandy]

Construction of V(3,5) = V(5,3)

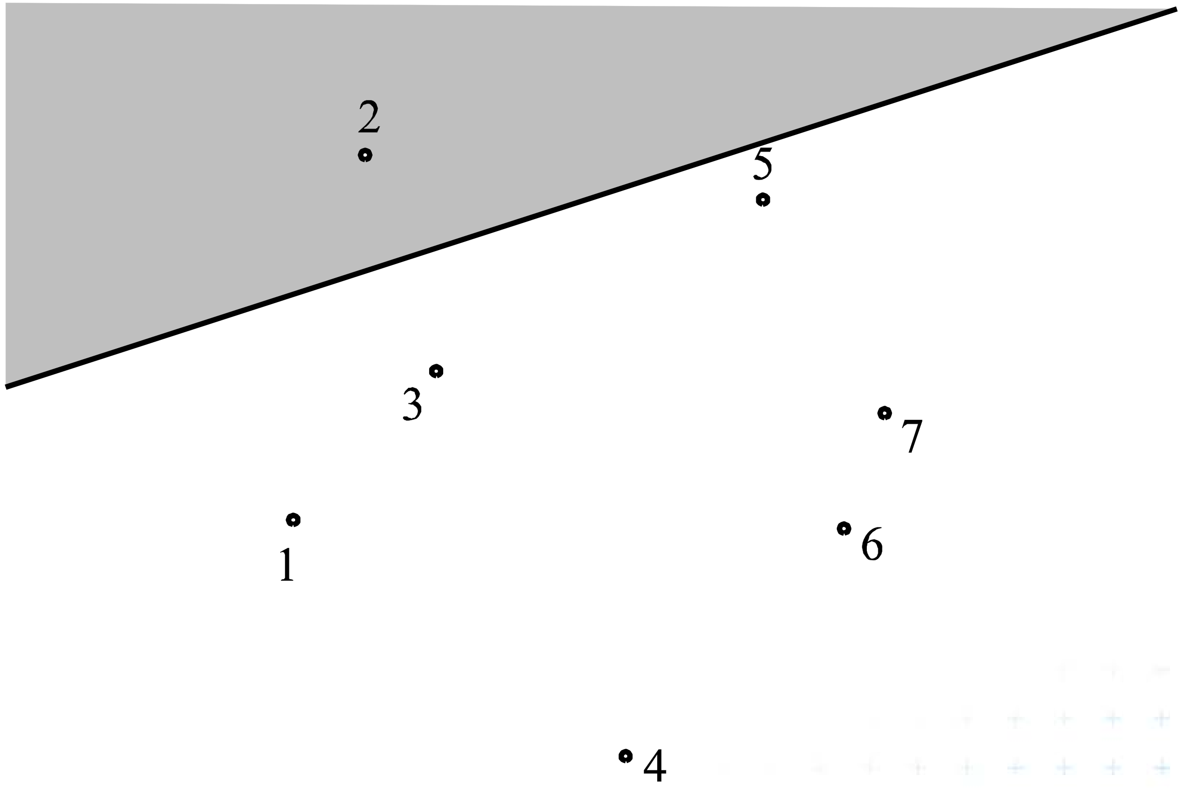


[Nandy]

Intersection of all halfplanes except *h*(3,5) and *h*(5,3)

$$\underbrace{\bigcap_{x\neq 5} h(3,x)}\cap\underbrace{\bigcap_{x\neq 3} h(5,x)}$$

Construction of V(3,5) = V(5,3)

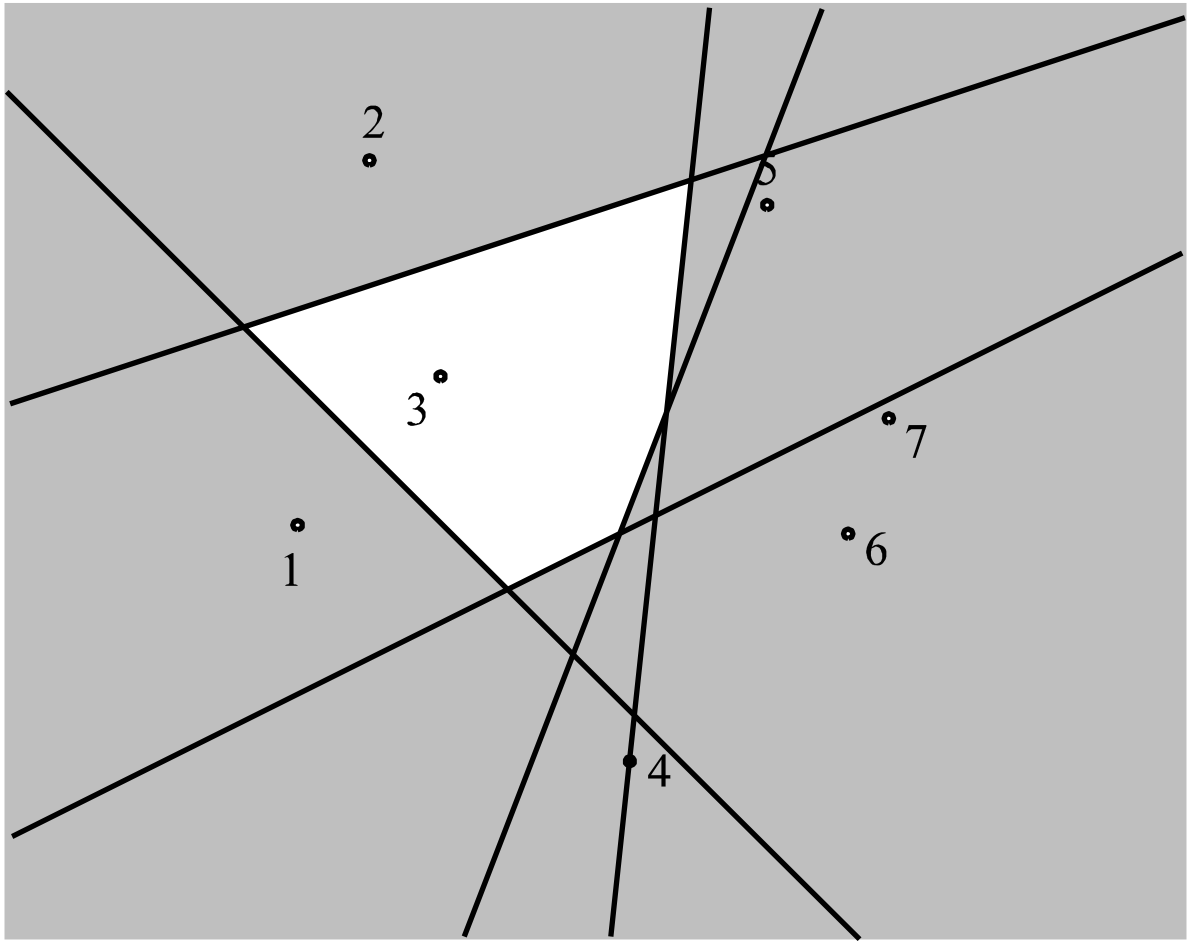


[Nandy]

Intersection of all halfplanes except *h*(3,5) and *h*(5,3)

$$\underbrace{\bigcap_{x\neq 5} h(3,x)}\cap\underbrace{\bigcap_{x\neq 3} h(5,x)}$$

Construction of V(3,5) = V(5,3)

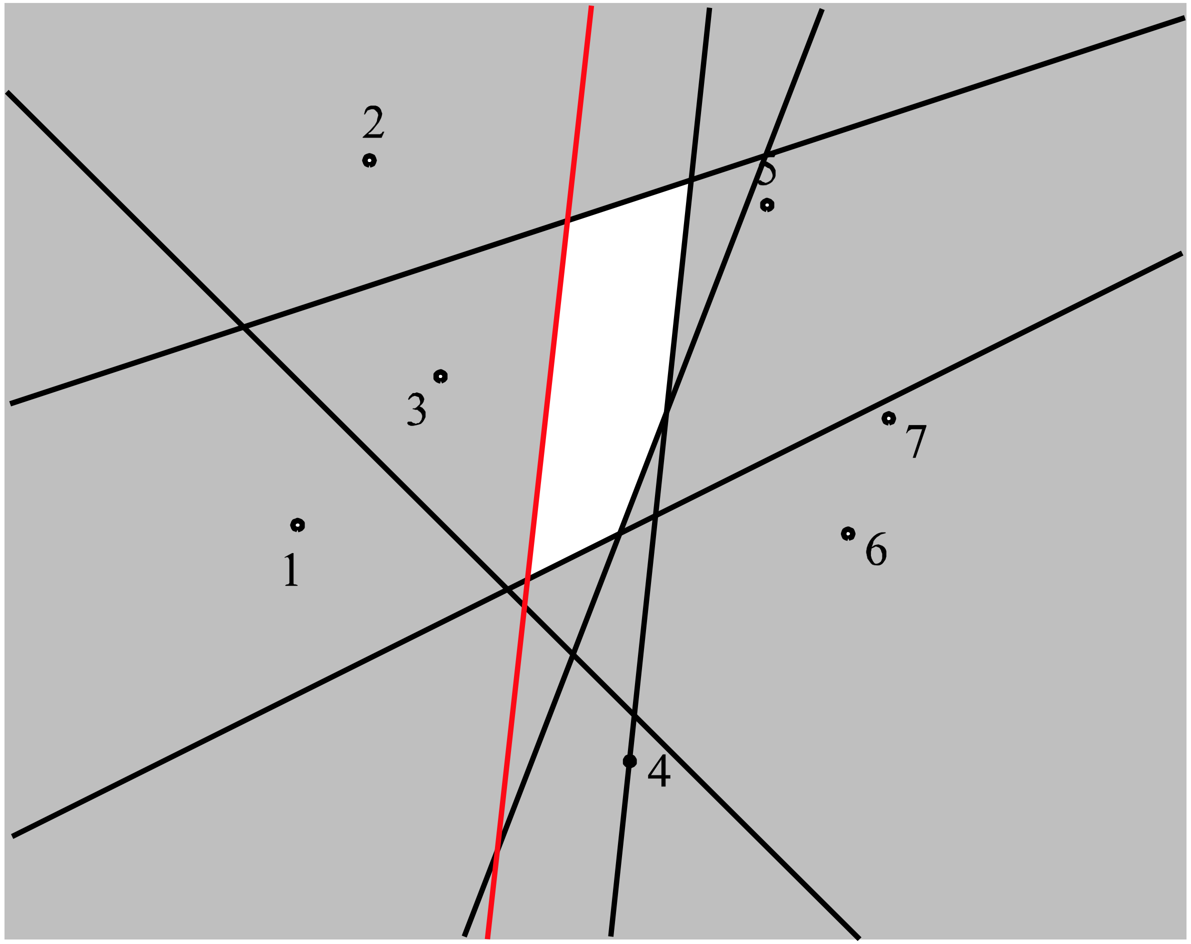


[Nandy]

Intersection of all halfplanes except *h*(3,5) and *h*(5,3)

$$\bigcap_{x \neq 5} h(3,x) \cap \bigcap_{x \neq 3} h(5,x)$$

Construction of V(3,5) = V(5,3)

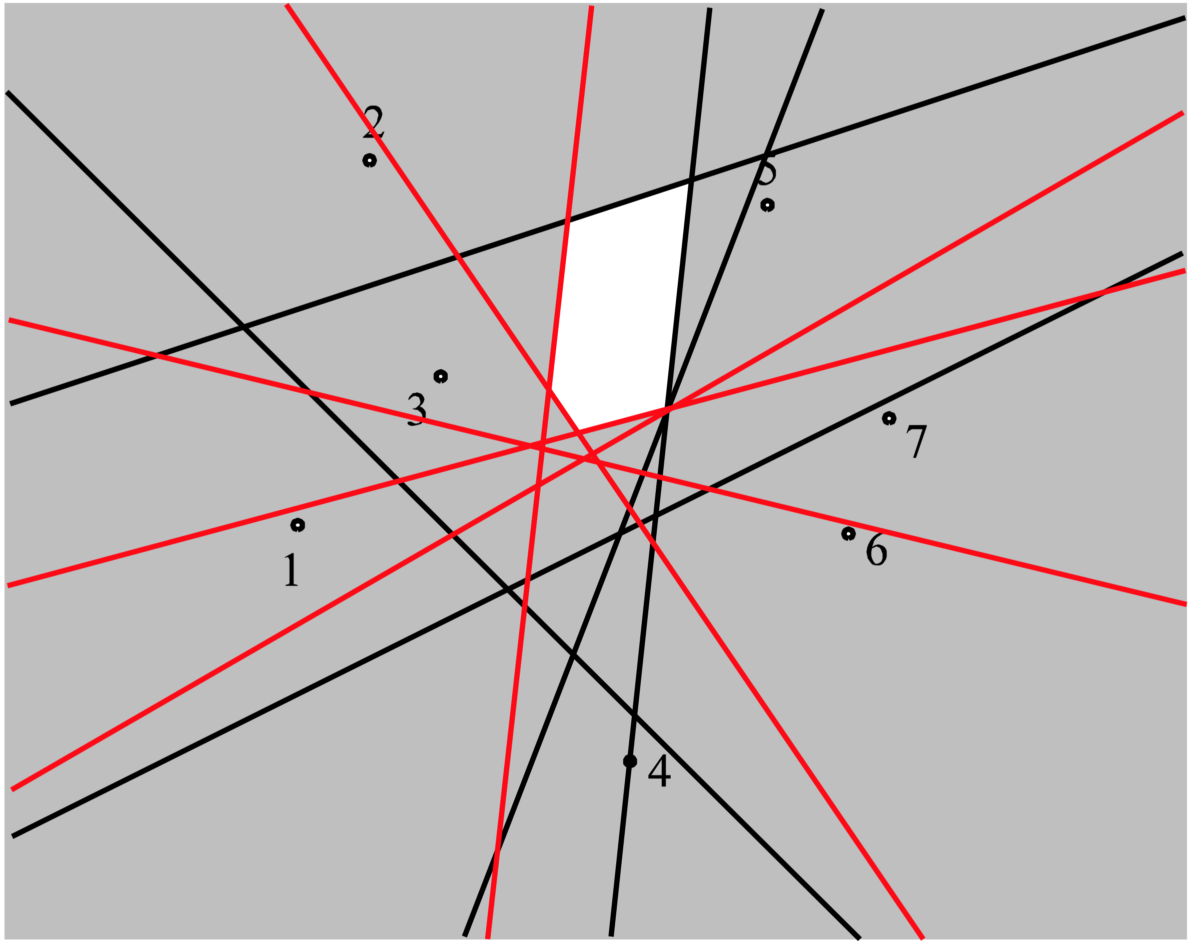


[Nandy]

Intersection of all halfplanes
except *h*(3,5) and *h*(5,3)

$$\underbrace{\bigcap_{x \neq 5} h(3,x)}\cap \underbrace{\bigcap_{x \neq 3} h(5,x)}$$

Construction of V(3,5) = V(5,3)

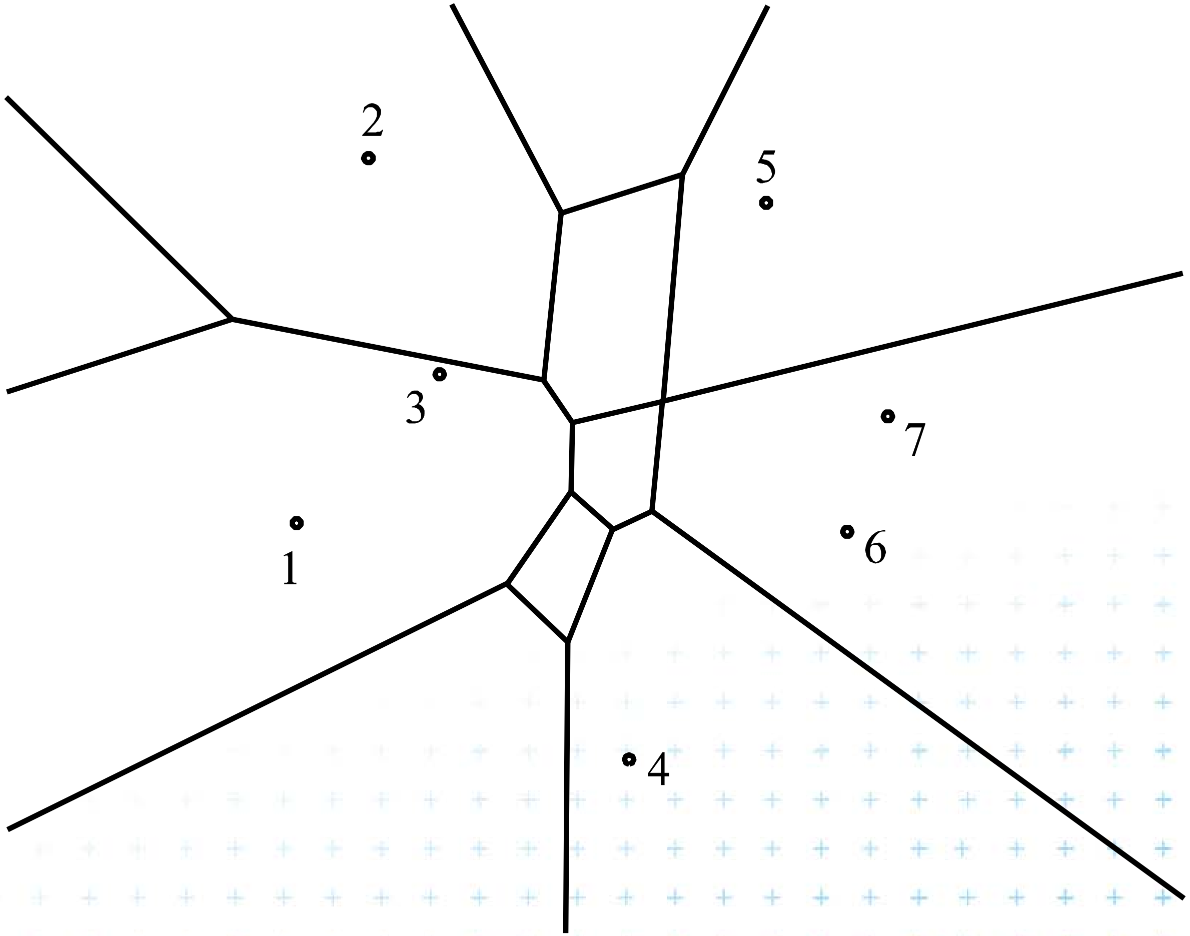


[Nandy]

Intersection of all halfplanes except *h*(3,5) and *h*(5,3)

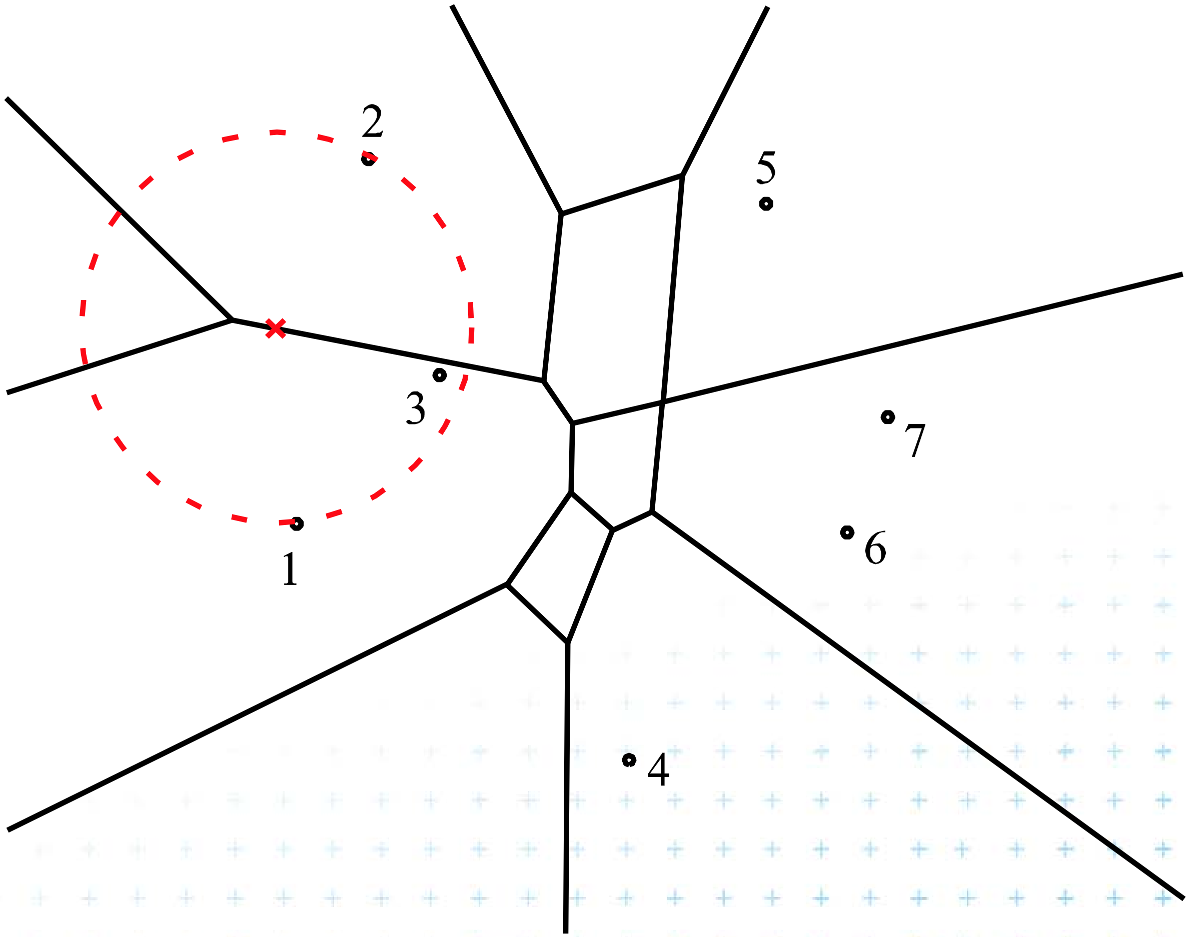
$$\underbrace{\bigcap_{x\neq 5} h(3,x)} \cap \underbrace{\bigcap_{x\neq 3} h(5,x)}$$

Order-2 Voronoi edges



[Nandy]

Order-2 Voronoi edges

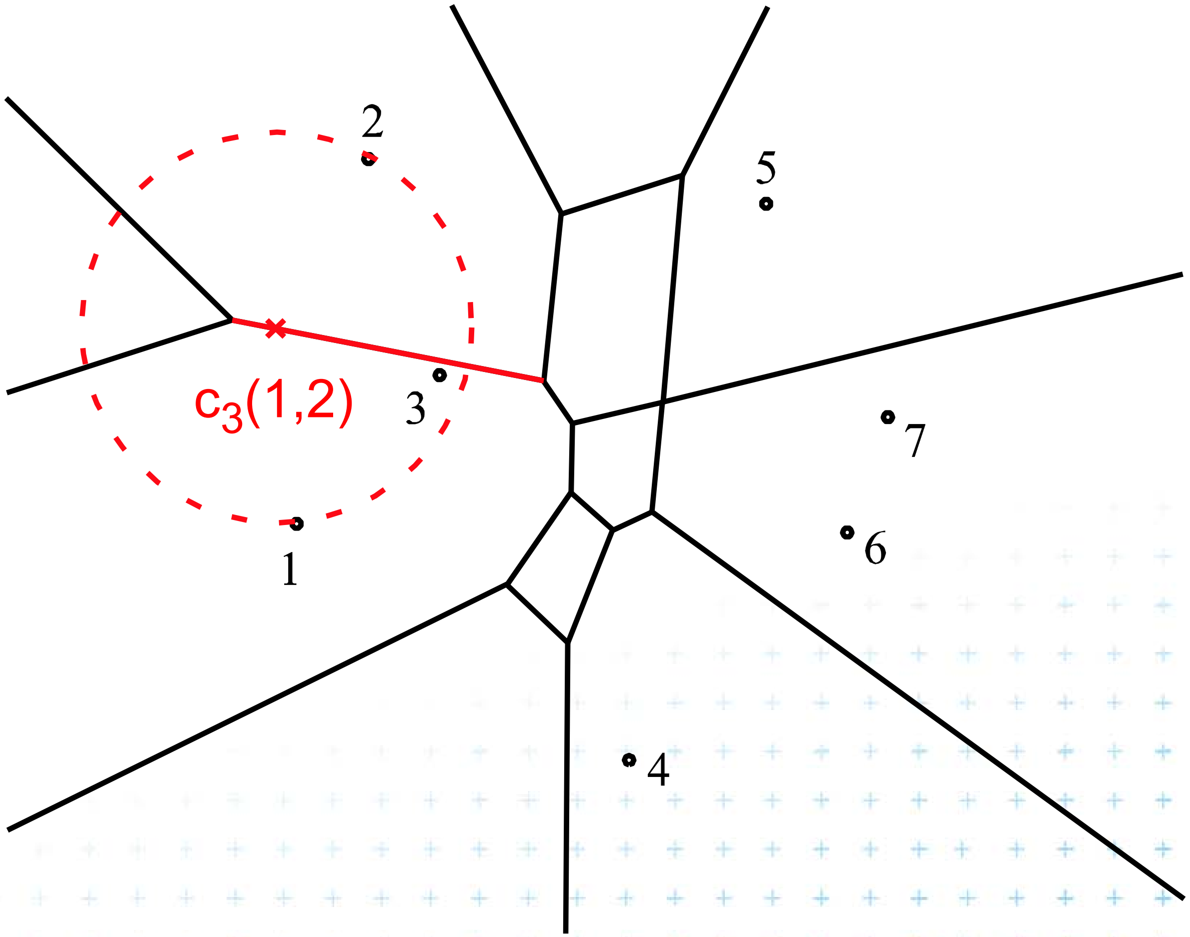


[Nandy]

Order-2 Voronoi edges

edge : set of centers of circles passing through 2 sites *s* and *t* and containing one site *p*

=> $c_p(s,t)$

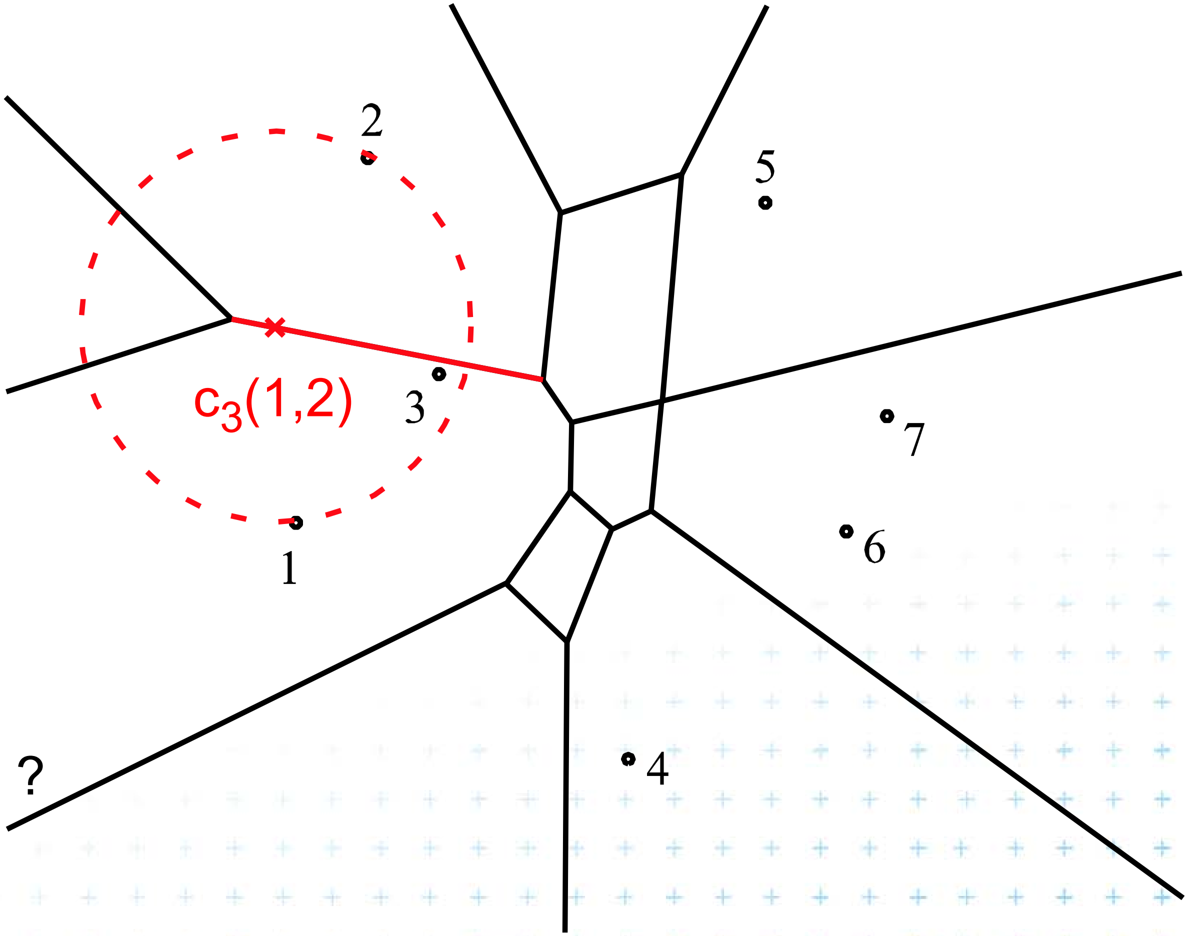


[Nandy]

Order-2 Voronoi edges

edge : set of centers of circles passing through 2 sites *s* and *t* and containing one site *p*

=> $c_p(s,t)$



Question

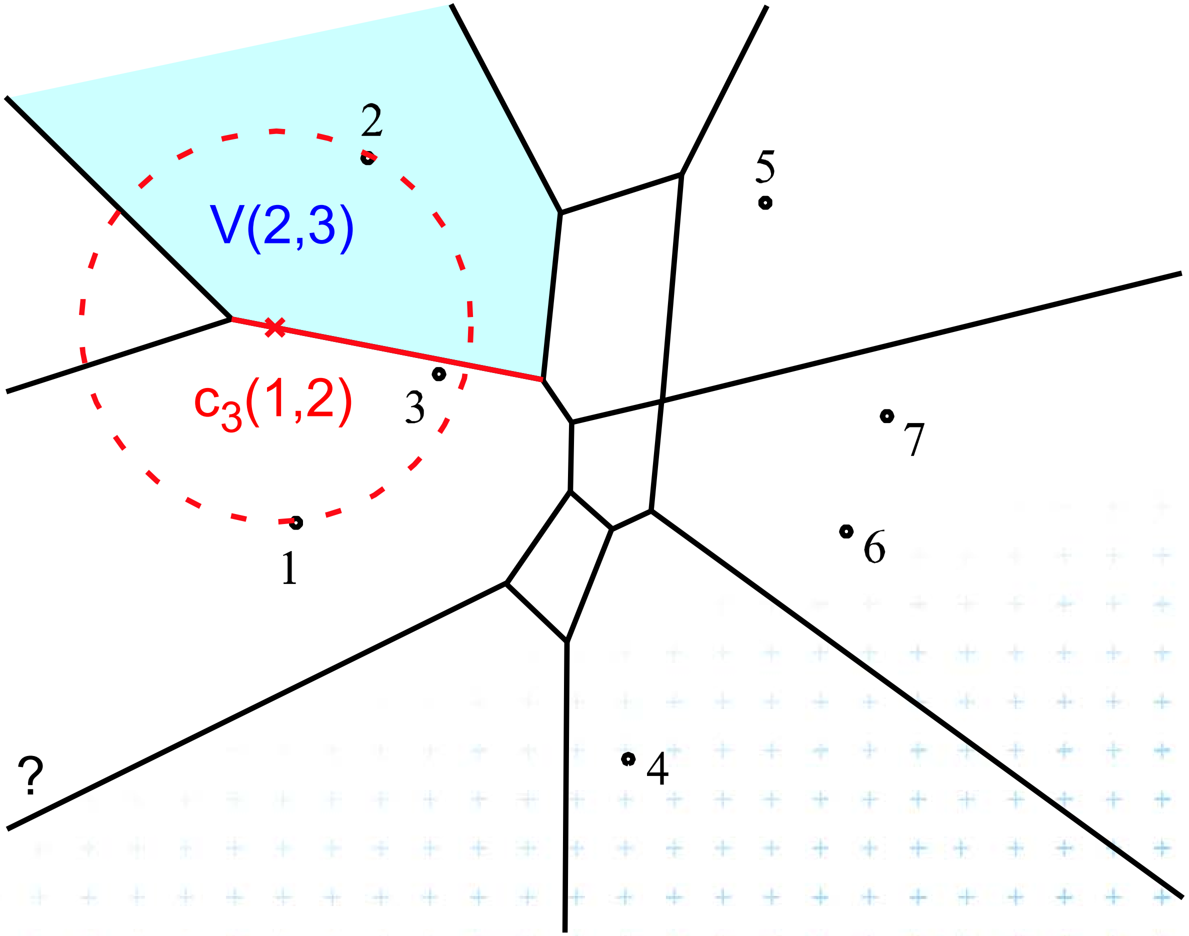
Which are the regions on both sides of $c_p(s,t)$?

[Nandy]

Order-2 Voronoi edges

edge : set of centers of circles passing through 2 sites *s* and *t* and containing one site *p*

=> $c_p(s,t)$



Question

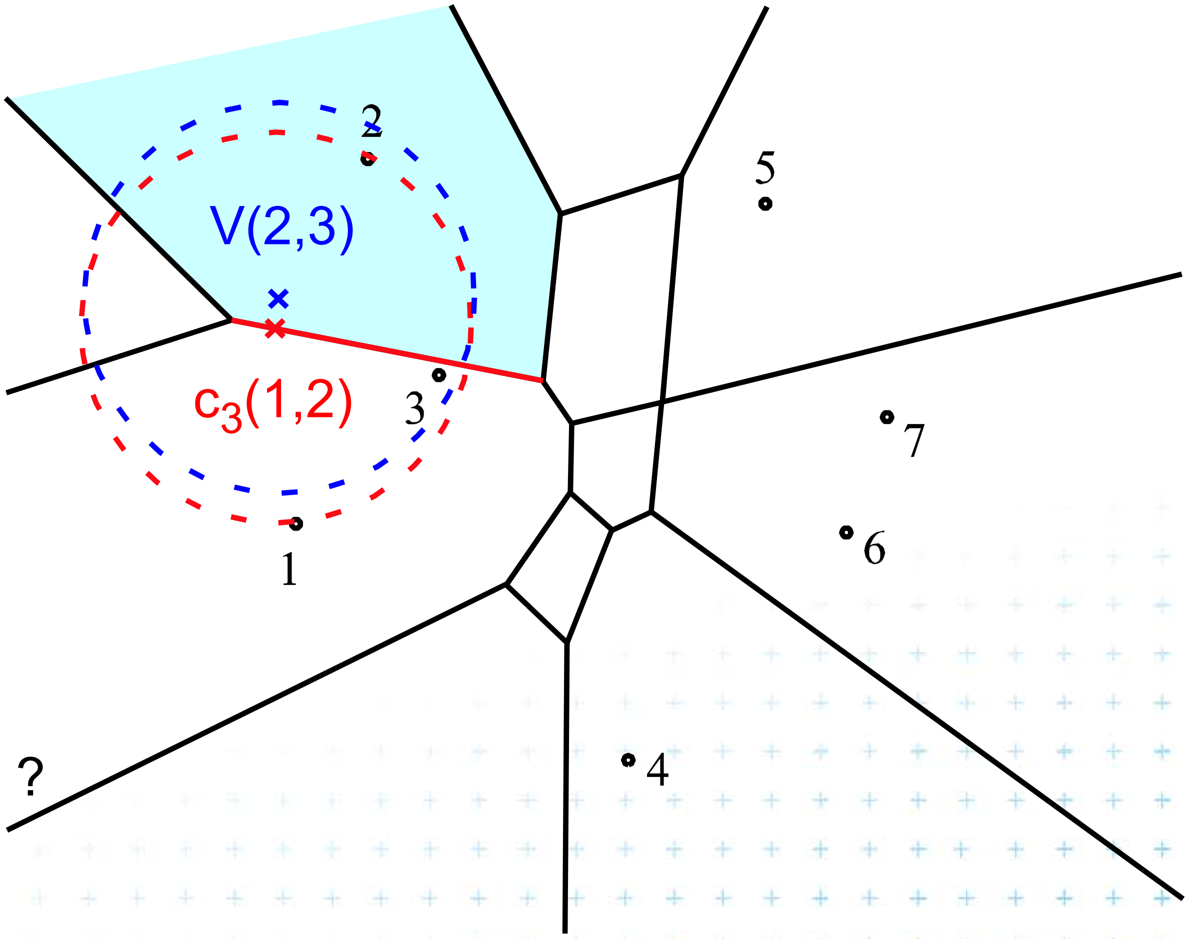
Which are the regions on both sides of $c_p(s,t)$?

[Nandy]

Order-2 Voronoi edges

edge : set of centers of circles passing through 2 sites *s* and *t* and containing one site *p*

=> $c_p(s,t)$



Question

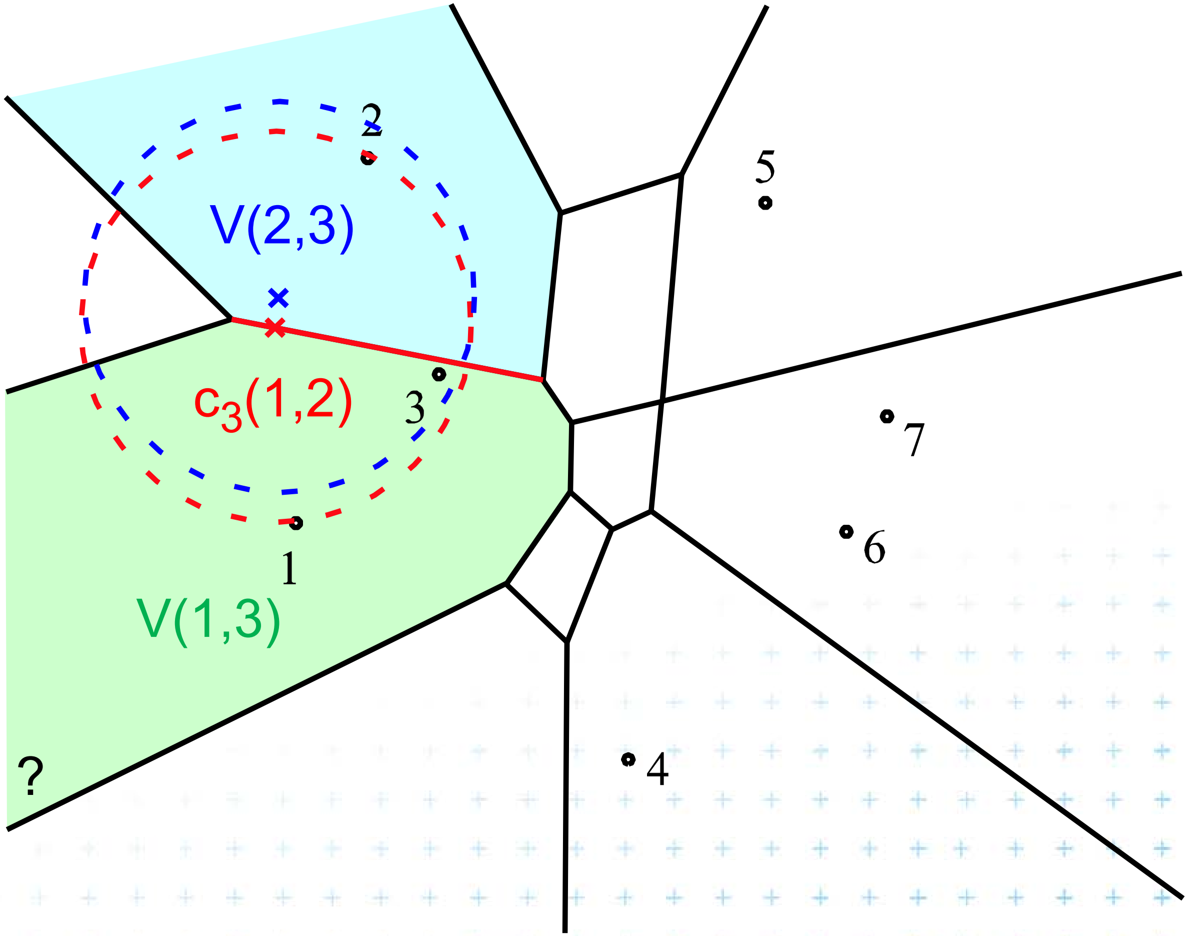
Which are the regions on both sides of $c_p(s,t)$?

[Nandy]

Order-2 Voronoi edges

edge : set of centers of circles passing through 2 sites *s* and *t* and containing one site *p*

=> $c_p(s,t)$



Question

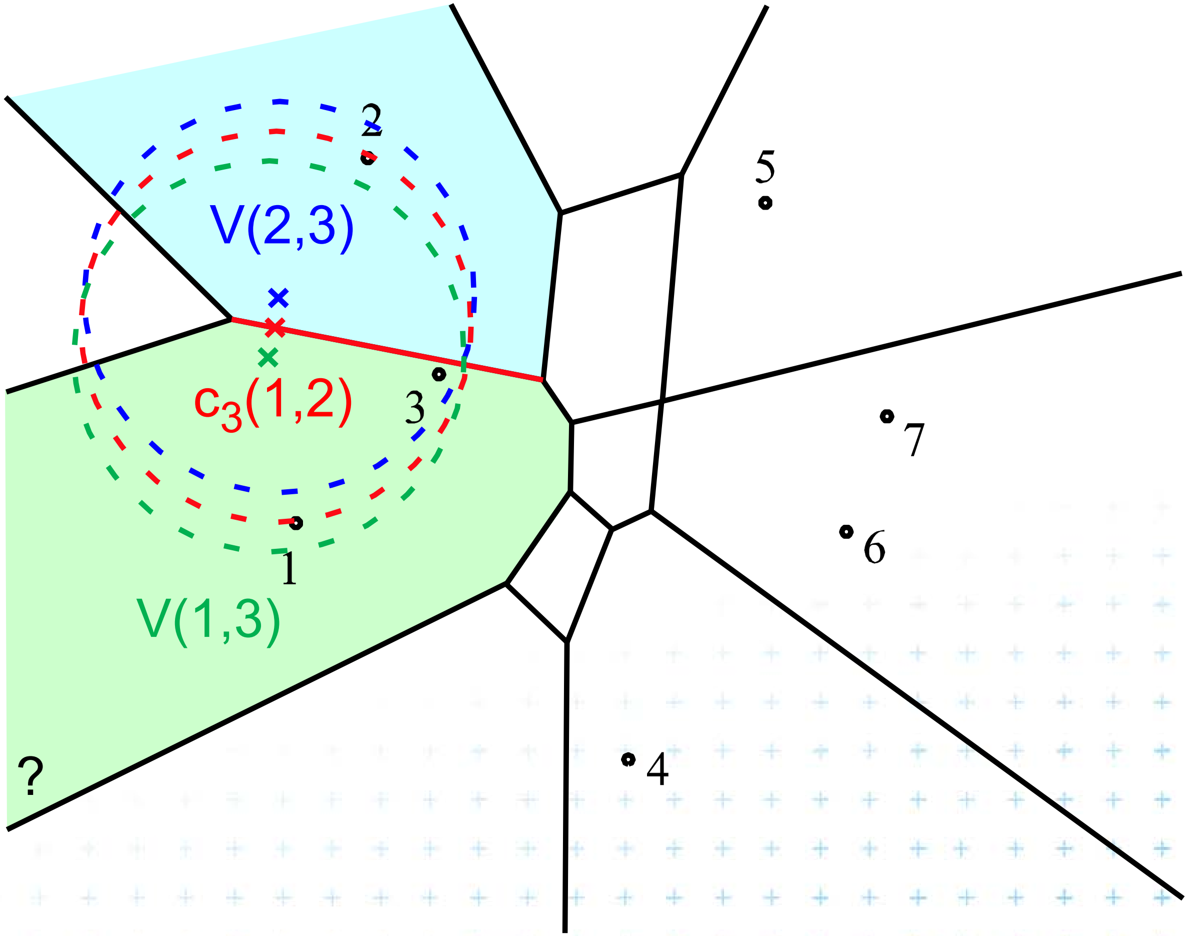
Which are the regions on both sides of $c_p(s,t)$?

[Nandy]

Order-2 Voronoi edges

edge : set of centers of circles passing through 2 sites *s* and *t* and containing one site *p*

=> $c_p(s,t)$



Question

Which are the regions on both sides of $c_p(s,t)$?

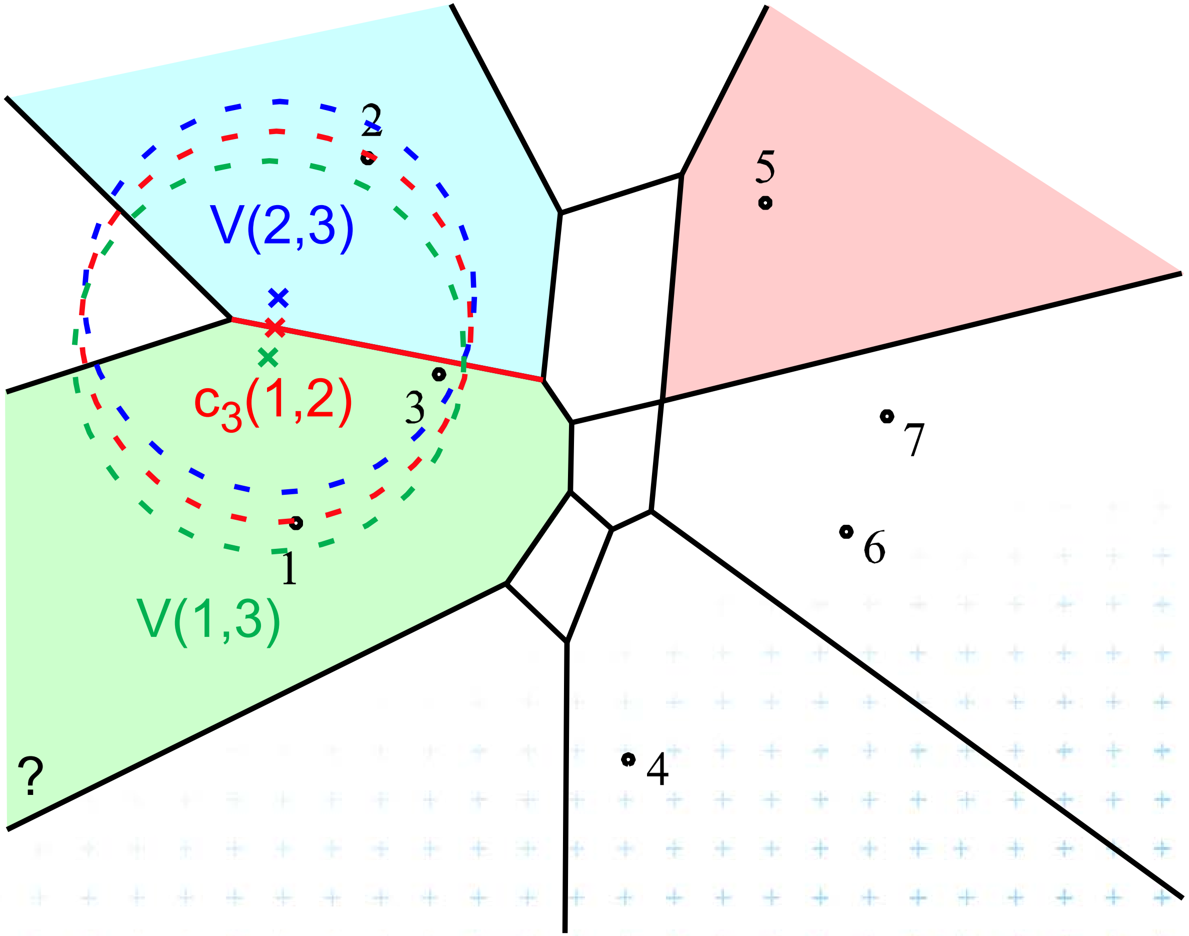
=> V(p,s) and V(p,t)

[Nandy]

Order-2 Voronoi edges

edge : set of centers of circles passing through 2 sites *s* and *t* and containing one site *p*

=> $c_p(s,t)$



Question

Which are the regions on both sides of $c_p(s,t)$?

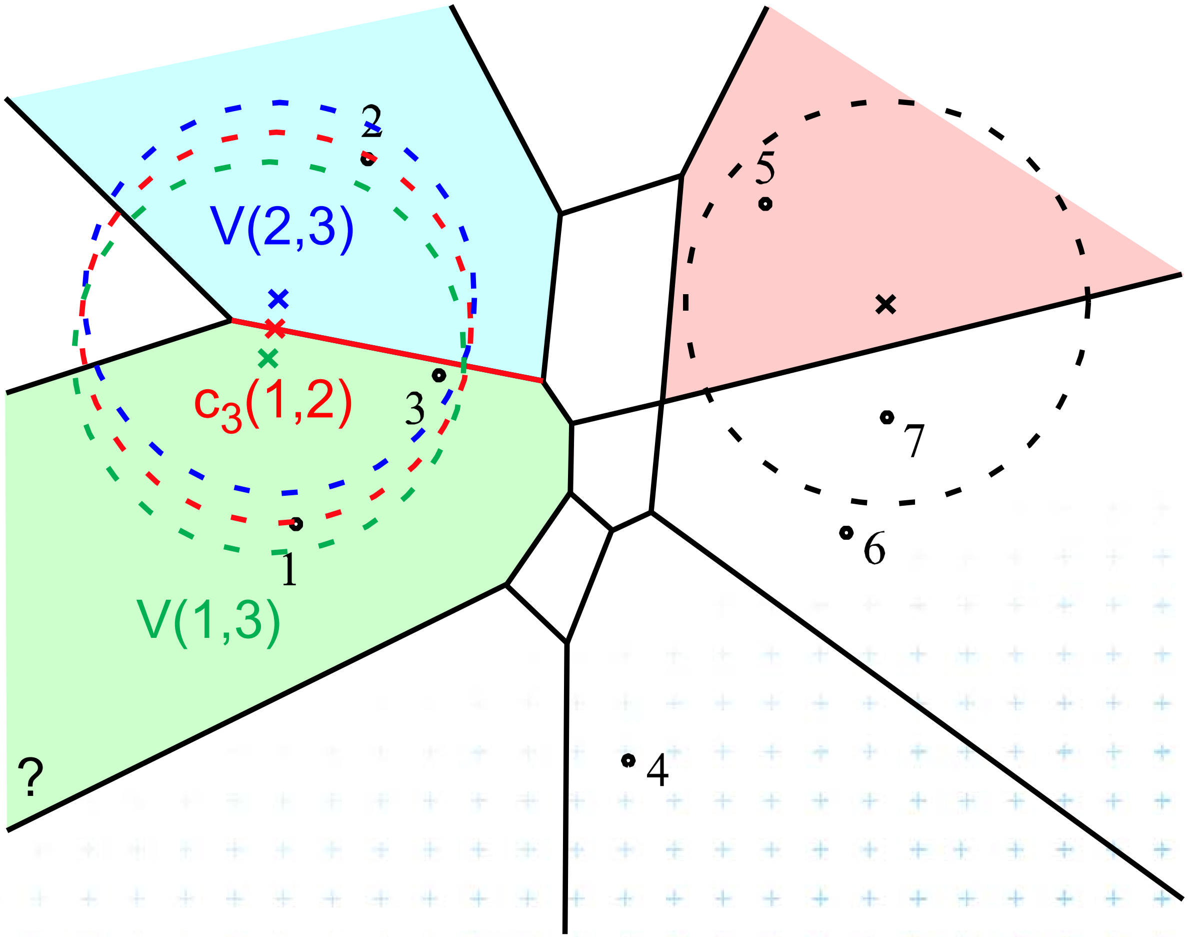
=> V(p,s) and V(p,t)

[Nandy]

Order-2 Voronoi edges

edge : set of centers of circles passing through 2 sites *s* and *t* and containing one site *p*

=> $c_p(s,t)$



Question

Which are the regions on both sides of $c_p(s,t)$?

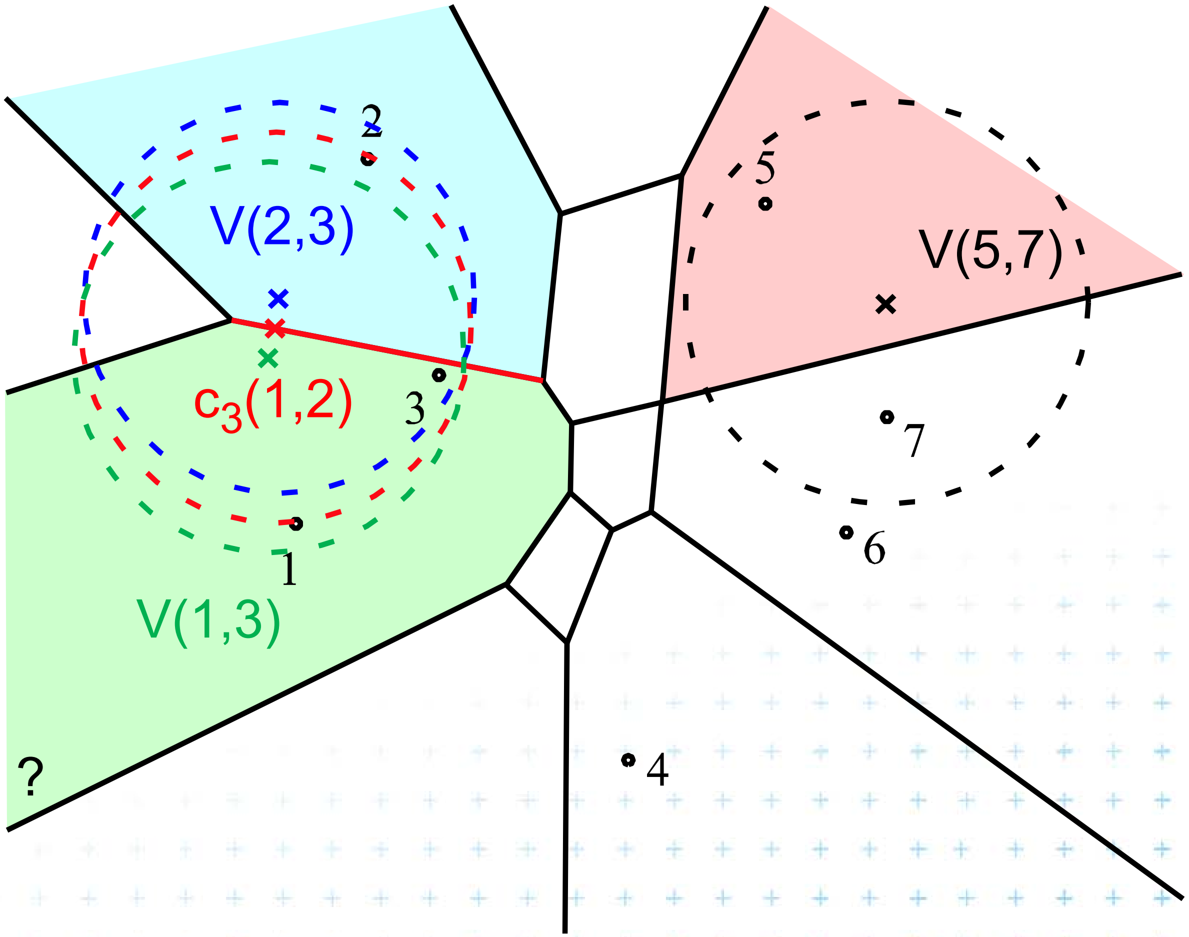
=> V(p,s) and V(p,t)

[Nandy]

Order-2 Voronoi edges

edge : set of centers of circles passing through 2 sites *s* and *t* and containing one site *p*

=> $c_p(s,t)$



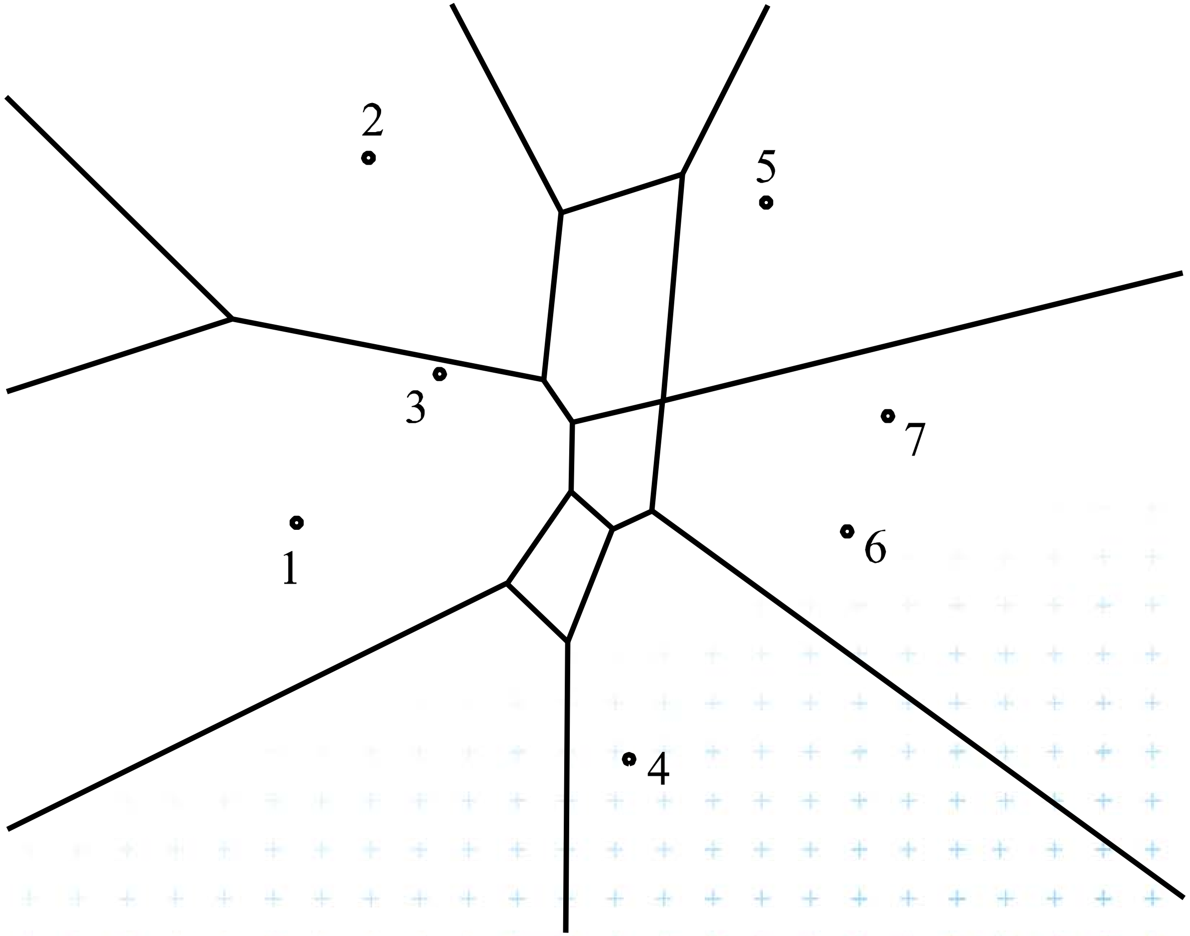
Question

Which are the regions on both sides of $c_p(s,t)$?

=> V(p,s) and V(p,t)

[Nandy]

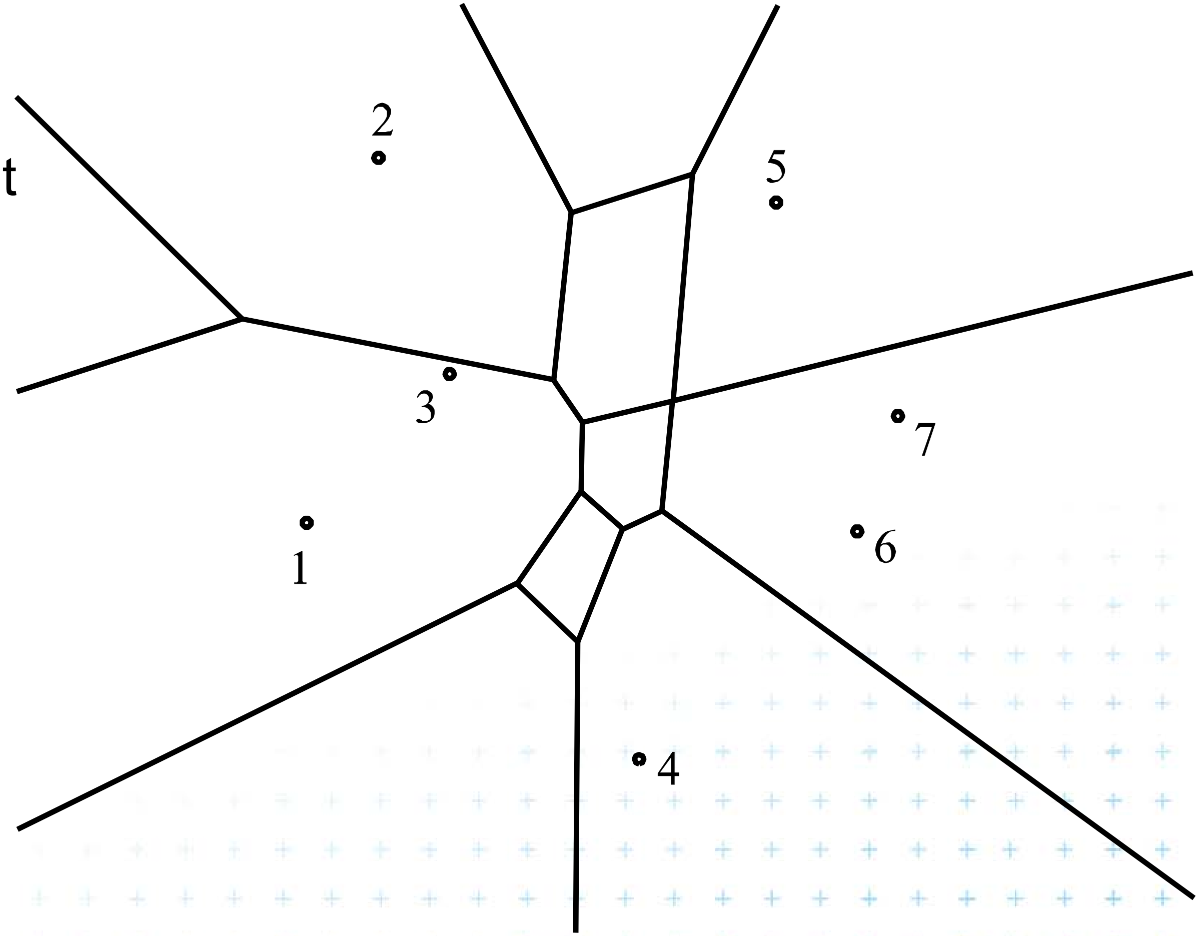
Order-2 Voronoi vertices



[Nandy]

Order-2 Voronoi vertices

vertex : center of a circle passing through at least 3 sites and containing either site p or nothing

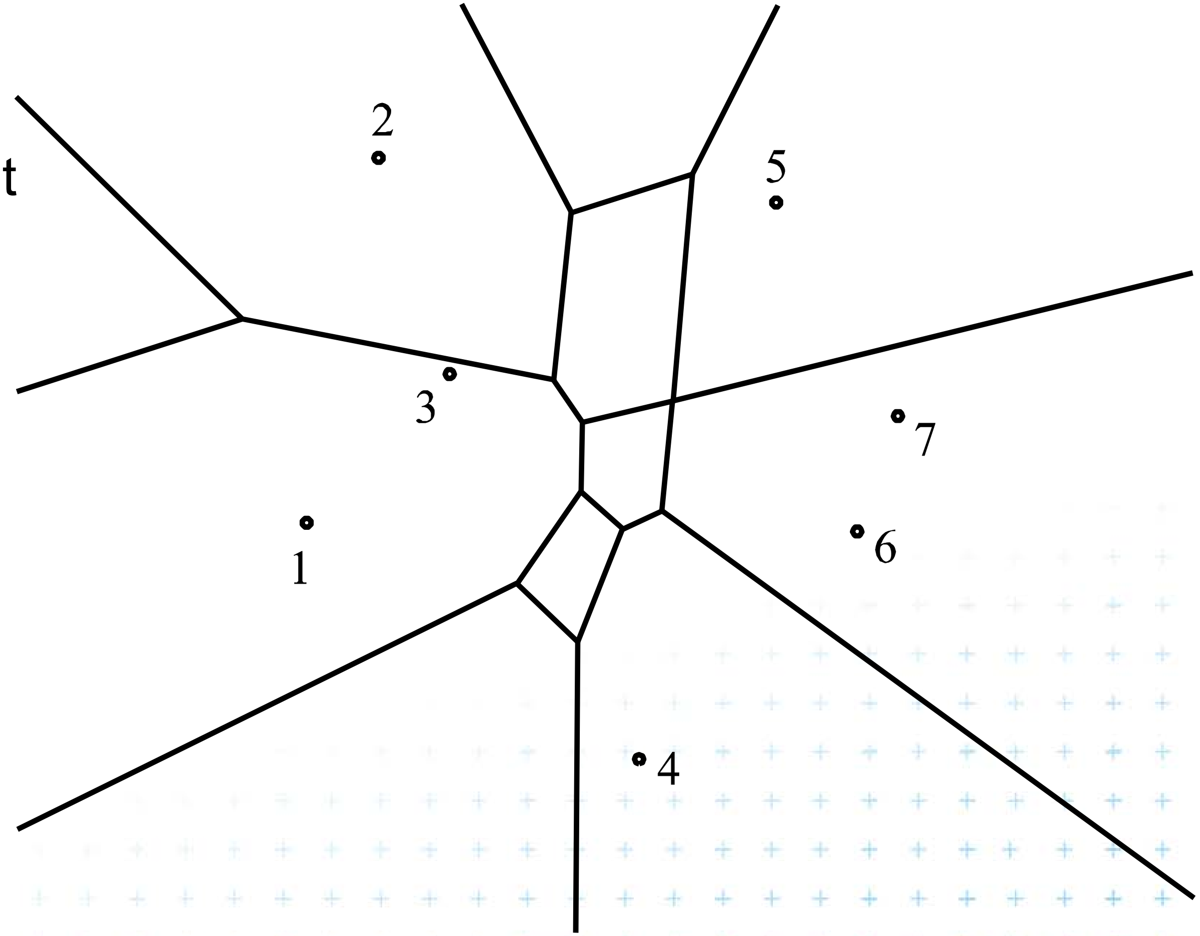


[Nandy]

Order-2 Voronoi vertices

vertex : center of a circle passing through at least 3 sites and containing either site p or nothing

⇒ $u_p(Q)$
$u_5(2,3,7)$,

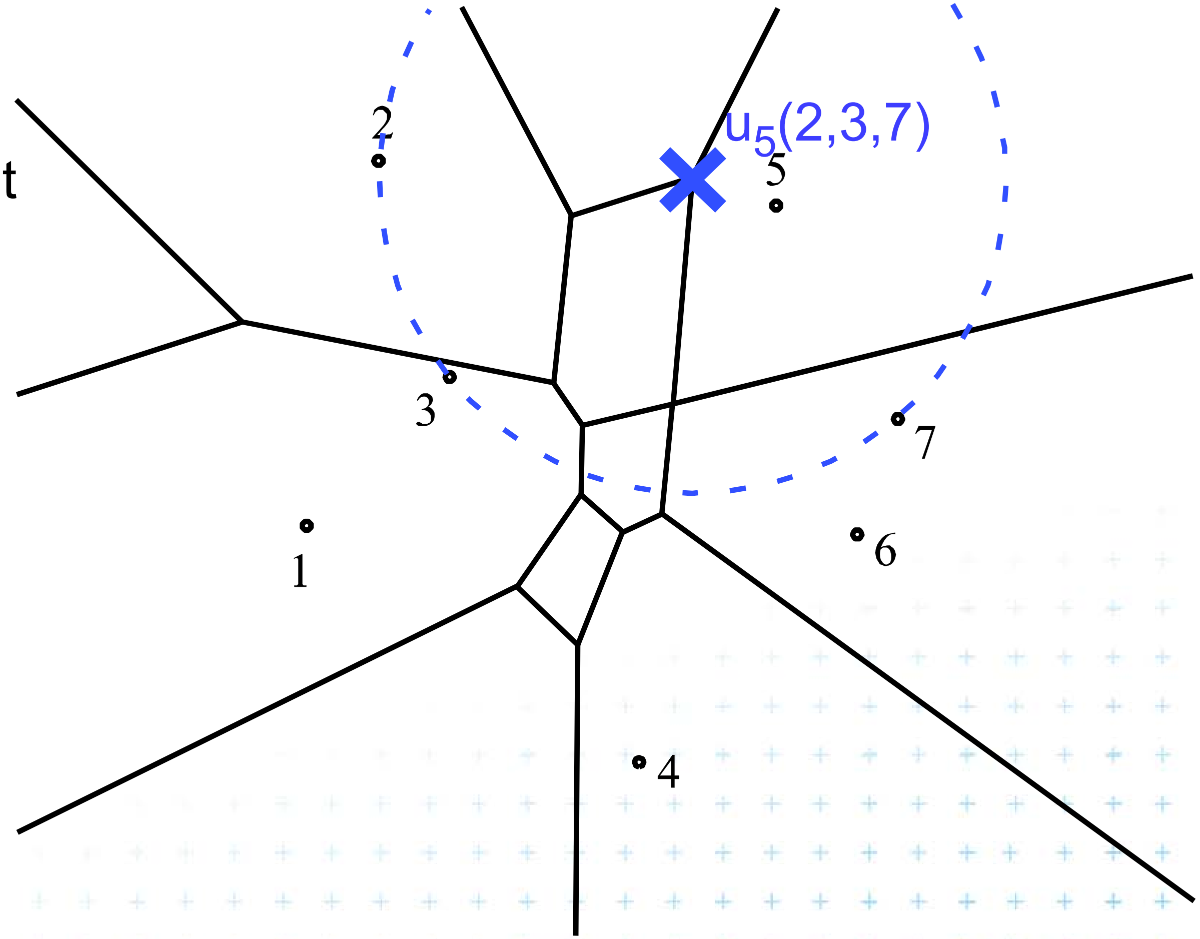


[Nandy]

Order-2 Voronoi vertices

vertex : center of a circle passing through at least 3 sites and containing either site p or nothing

⇒ $u_p(Q)$
$u_5(2,3,7)$,



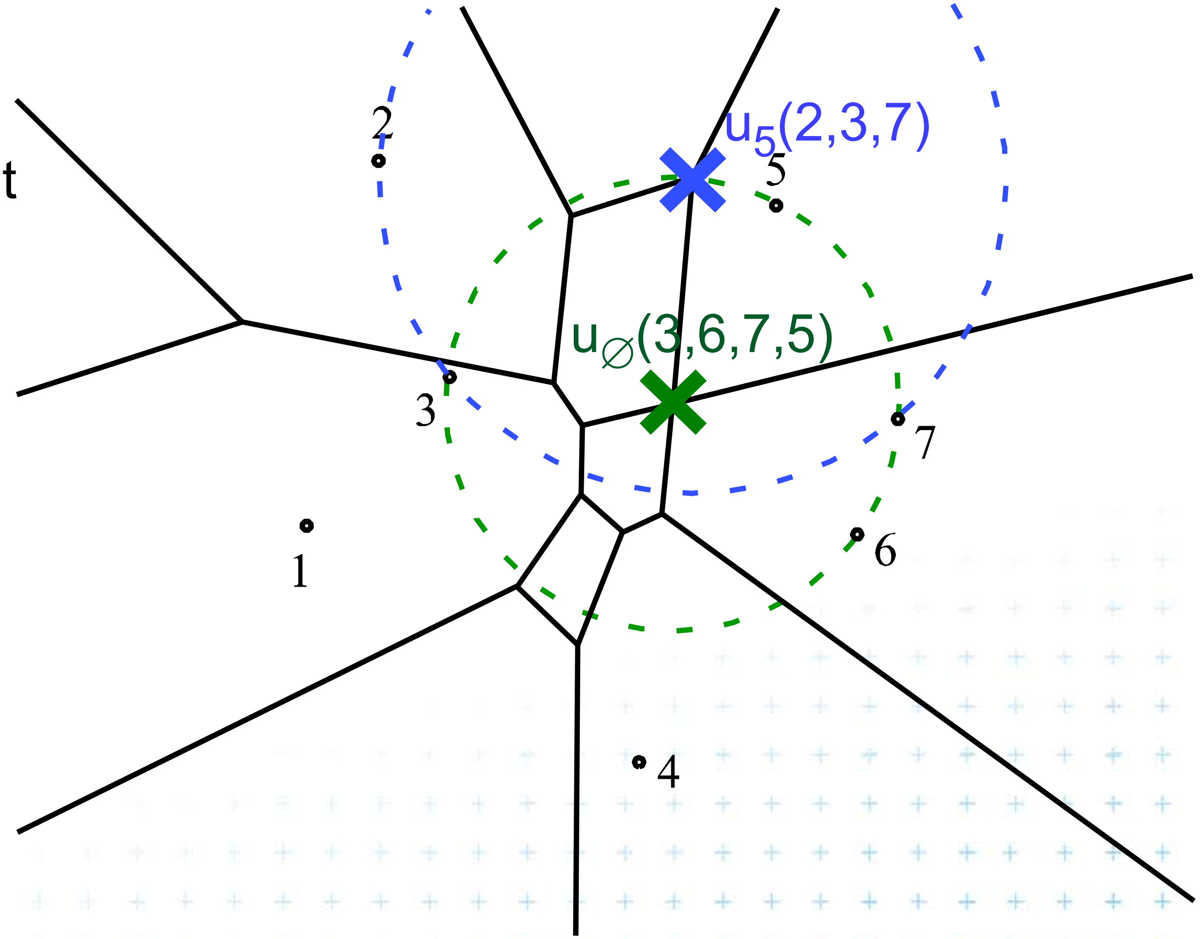
[Nandy]

Order-2 Voronoi vertices

vertex : center of a circle passing through at least 3 sites and containing either site p or nothing

$\Rightarrow u_p(Q)$ or $u_\varnothing(Q \cup p)$

$u_5(2,3,7)$, $u_\varnothing(3,6,7)$



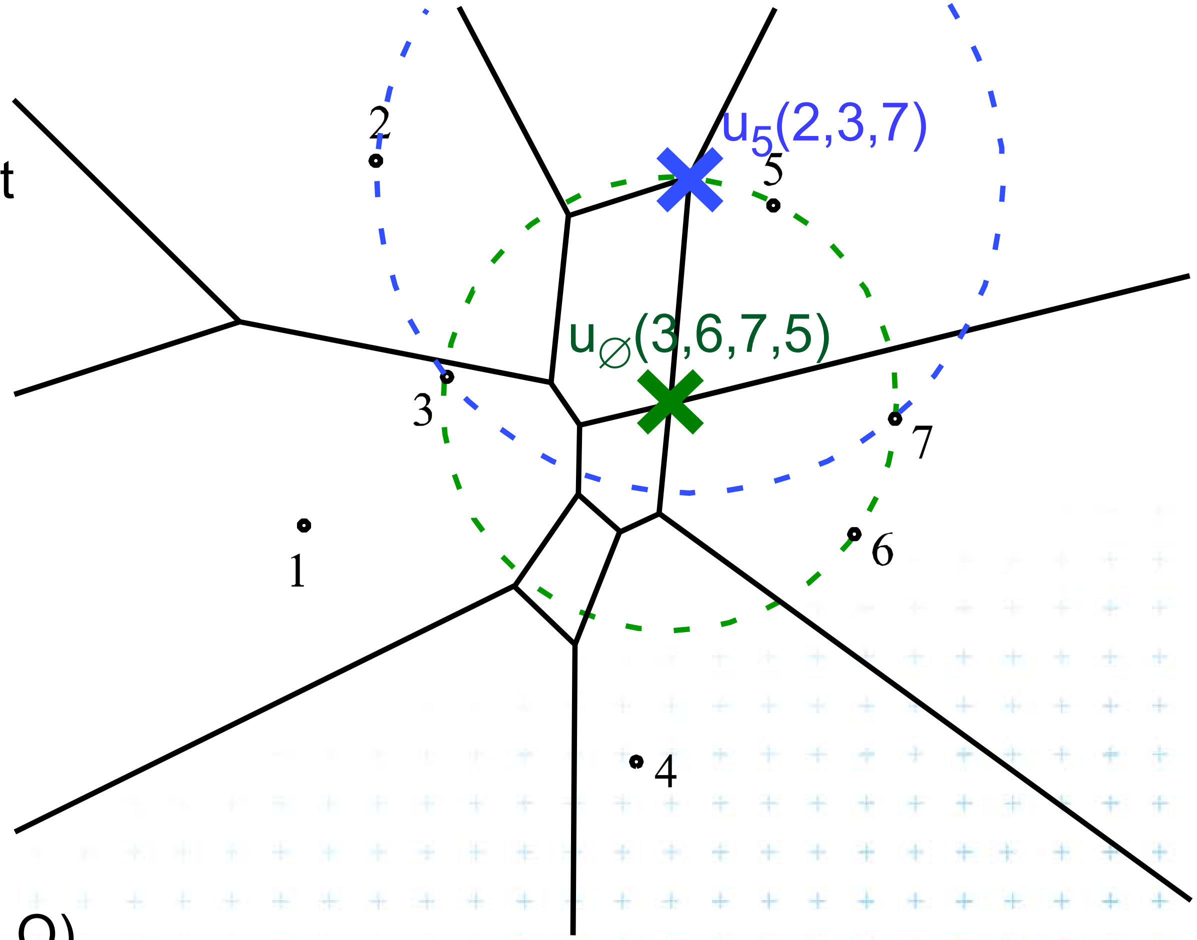
[Nandy]

Order-2 Voronoi vertices

vertex : center of a circle passing through at least 3 sites and containing either site p or nothing

⇒ $u_p(Q)$ or $u_{\varnothing}(Q \cup p)$

$u_5(2,3,7)$, $u_{\varnothing}(3,6,7)$



(circle circumscribed to Q)

[Nandy]

Order-2 Voronoi vertex $u_p(Q)$

vertex : center of a circle passing through at least 3 sites and containing either site p or nothing

Case $u_p(Q)$

$u_5(2,3,7)$

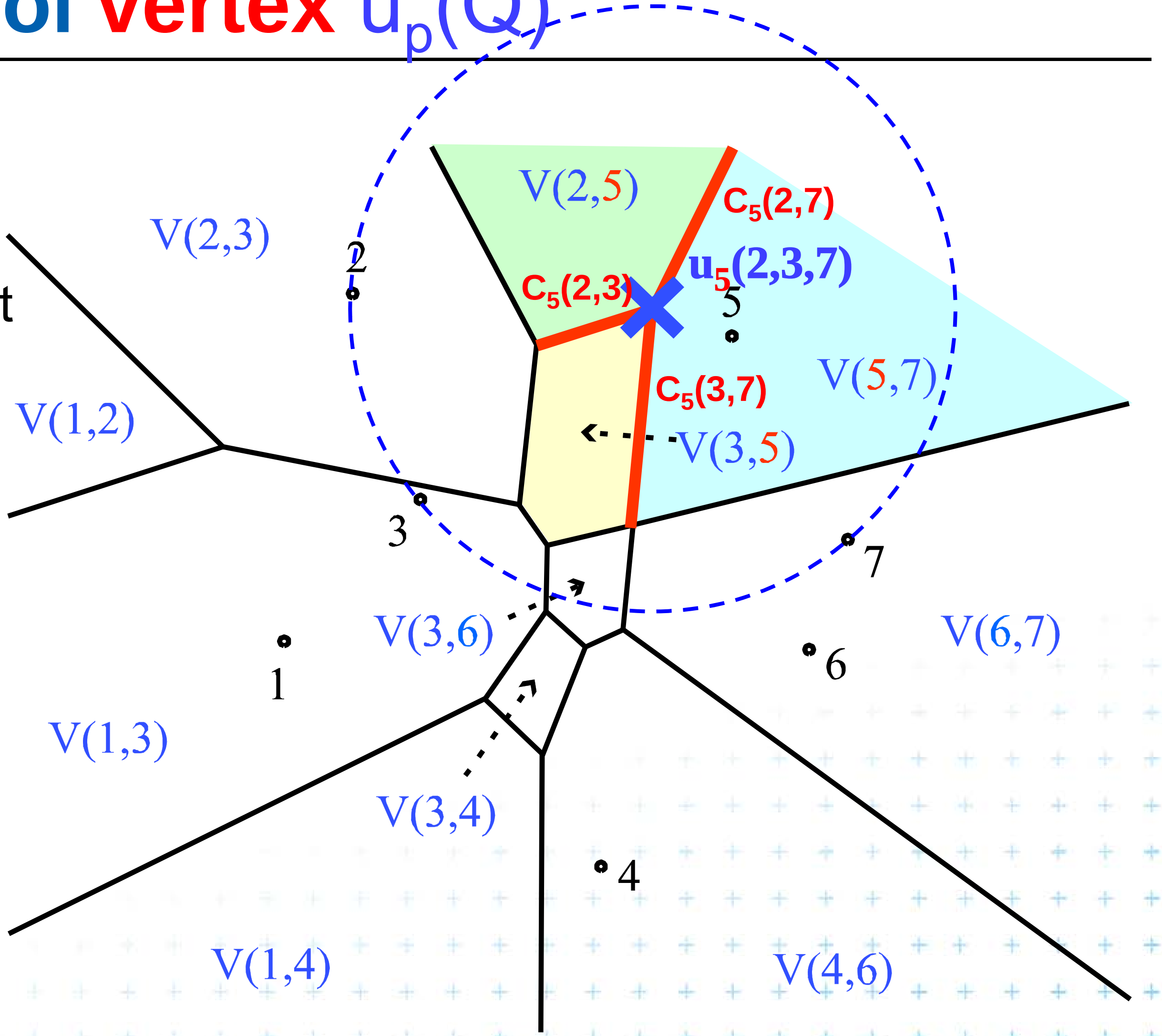
5 is inside for all incident edges:

$C_5(2,3)$

$C_5(2,7)$

$C_5(3,7)$

=> is inside for circle with center in vertex

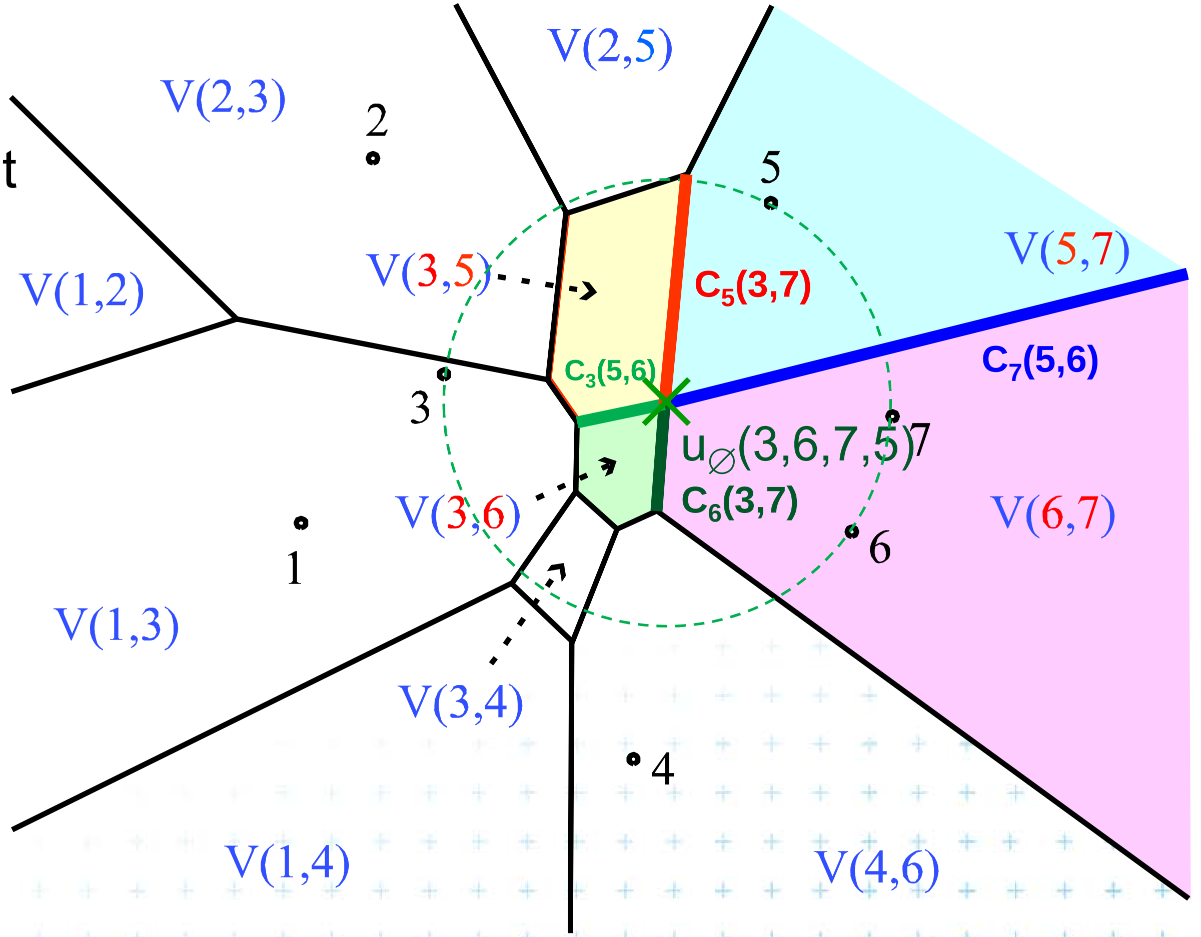


[Nandy]

Order-2 Voronoi vertex $u_{\emptyset}(Q \cup p)$

vertex : center of a circle passing through at least 3 sites and containing either site p or nothing

Case $u_{\emptyset}(Q \cup p)$
$u_{\emptyset}(3,6,7,5)$



[Nandy]

Order-k Voronoi Diagram

Theorem věta

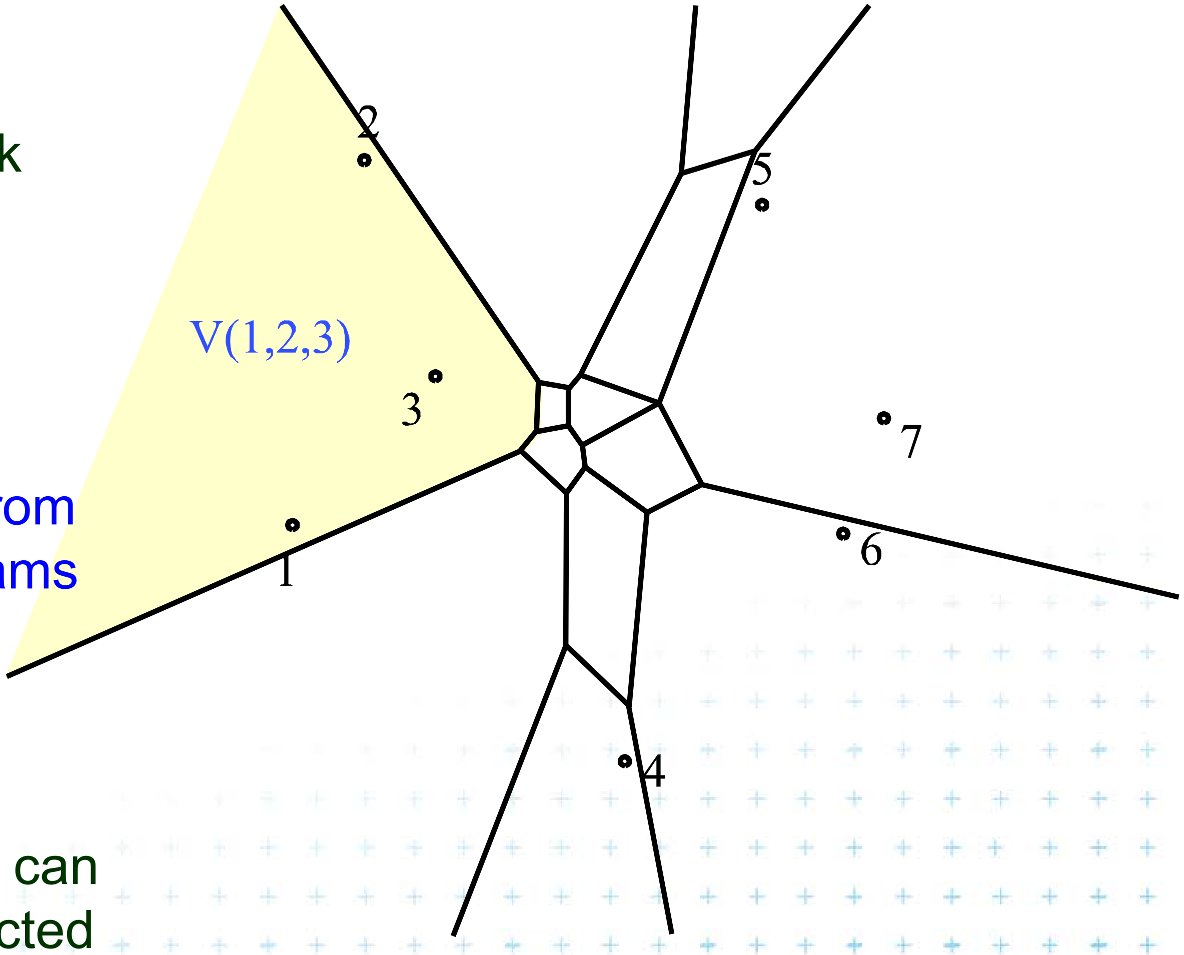
The size of the order-k diagrams is O(k(n-k))

Theorem věta

The order-k diagrams can be constructed from the order-(k-1) diagrams in O(k(n-k)) time

Corollary důsledek

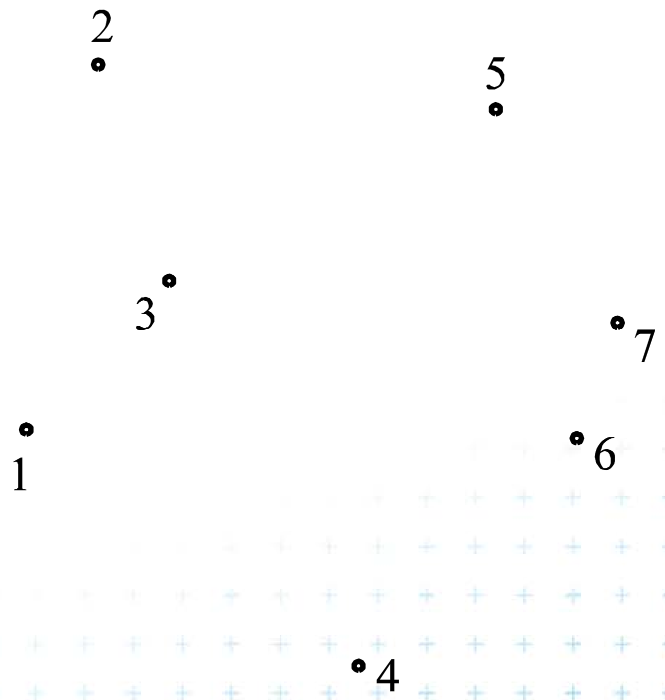
The order-k diagrams can be iteratively constructed in O(n log n + k^2(n-k)) time



[Nandy]

Order n-1 = Farthest-point Voronoi diagram

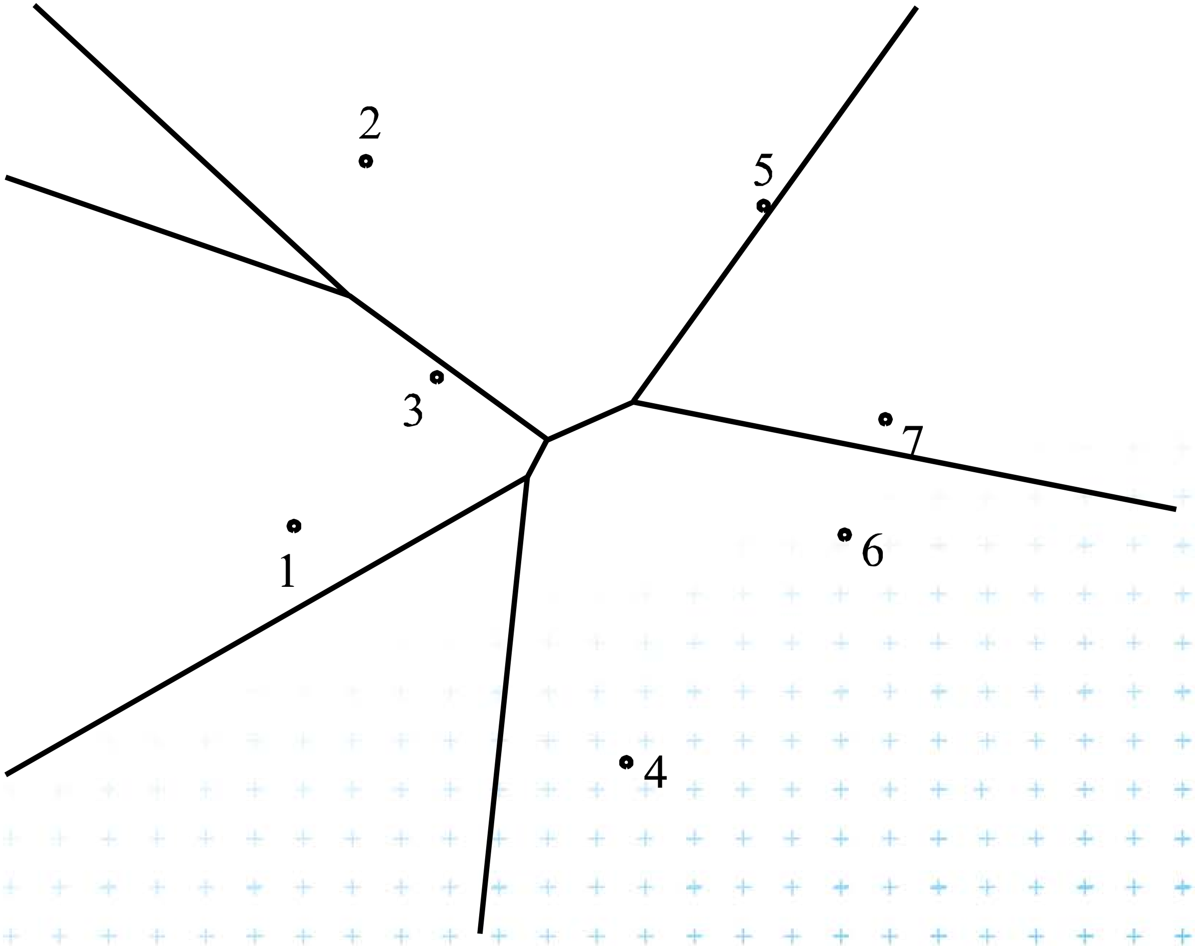
cell $V_{-1}(7) = V_{n-1}(\{1,2,3,4,5,6\})$
= set of points in the plane farther from p_i=7 than from any other site



[Nandy]

Order n-1 = Farthest-point Voronoi diagram

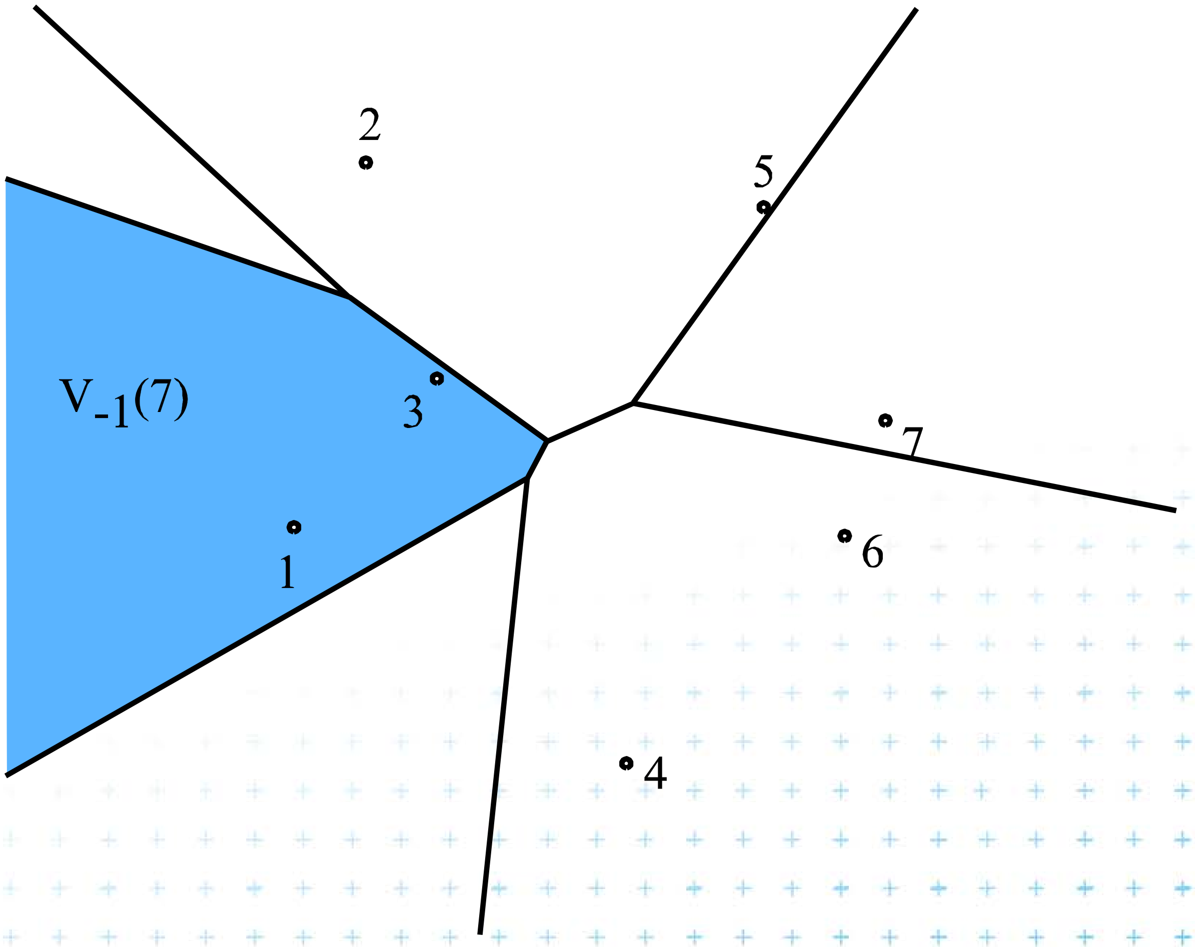
cell $V_{-1}(7) = V_{n-1}(\{1,2,3,4,5,6\})$
= set of points in the plane farther from p_i=7 than from any other site



[Nandy]

Order n-1 = Farthest-point Voronoi diagram

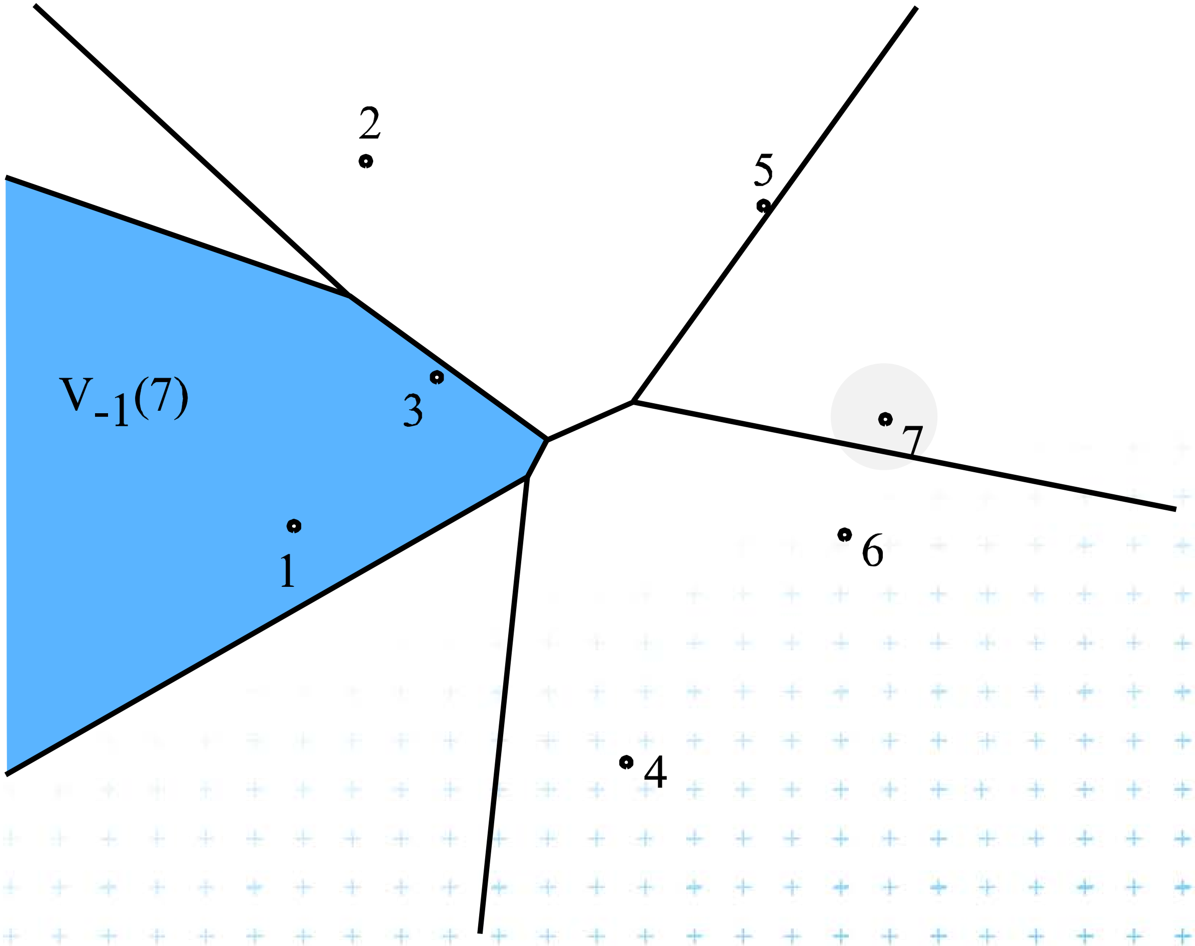
cell $V_{-1}(7) = V_{n-1}(\{1,2,3,4,5,6\})$
= set of points in the plane farther from p_i=7 than from any other site



[Nandy]

Order n-1 = Farthest-point Voronoi diagram

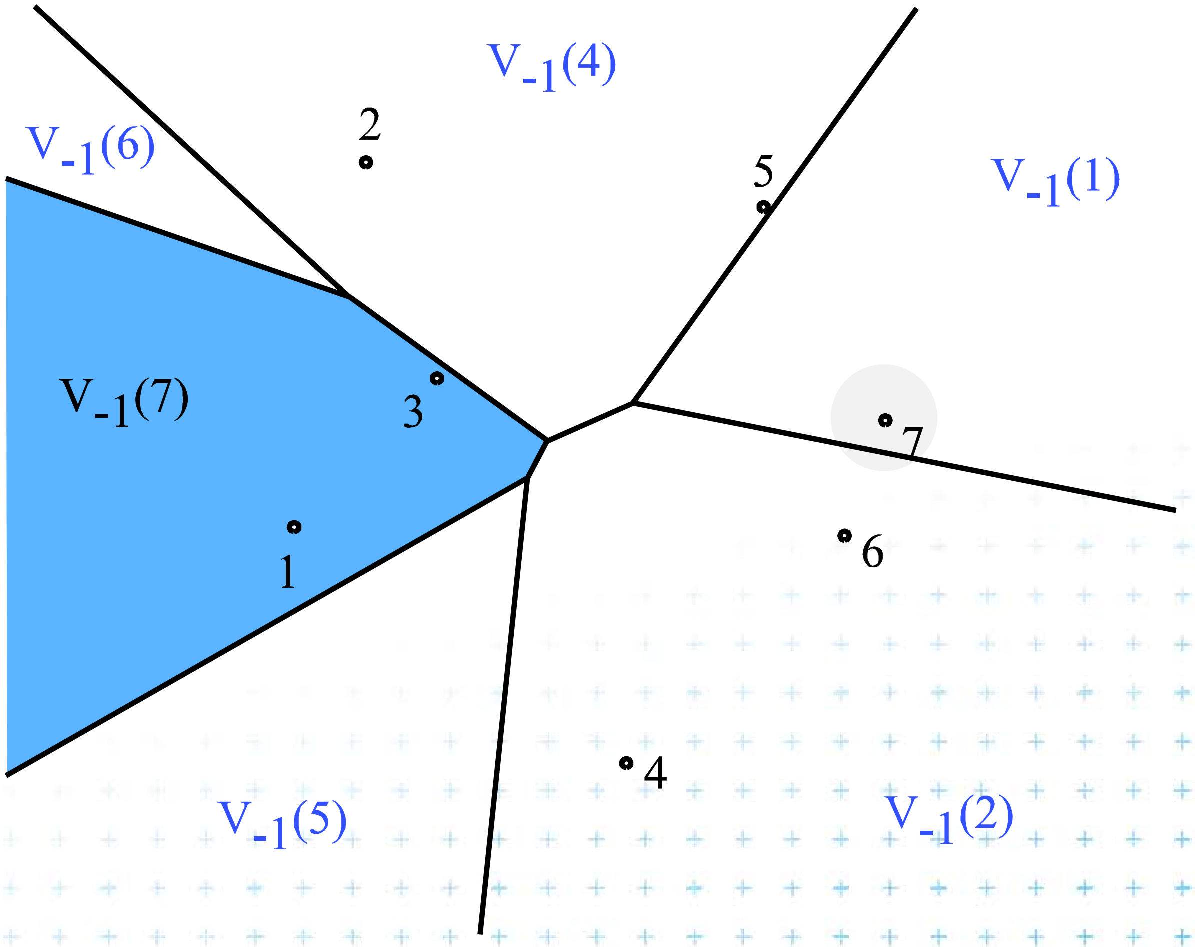
cell $V_{-1}(7) = V_{n-1}(\{1,2,3,4,5,6\})$ = set of points in the plane farther from p_i=7 than from any other site



[Nandy]

Order n-1 = Farthest-point Voronoi diagram

cell $V_{-1}(7) = V_{n-1}(\{1,2,3,4,5,6\})$
= set of points in the plane farther from p_i=7 than from any other site

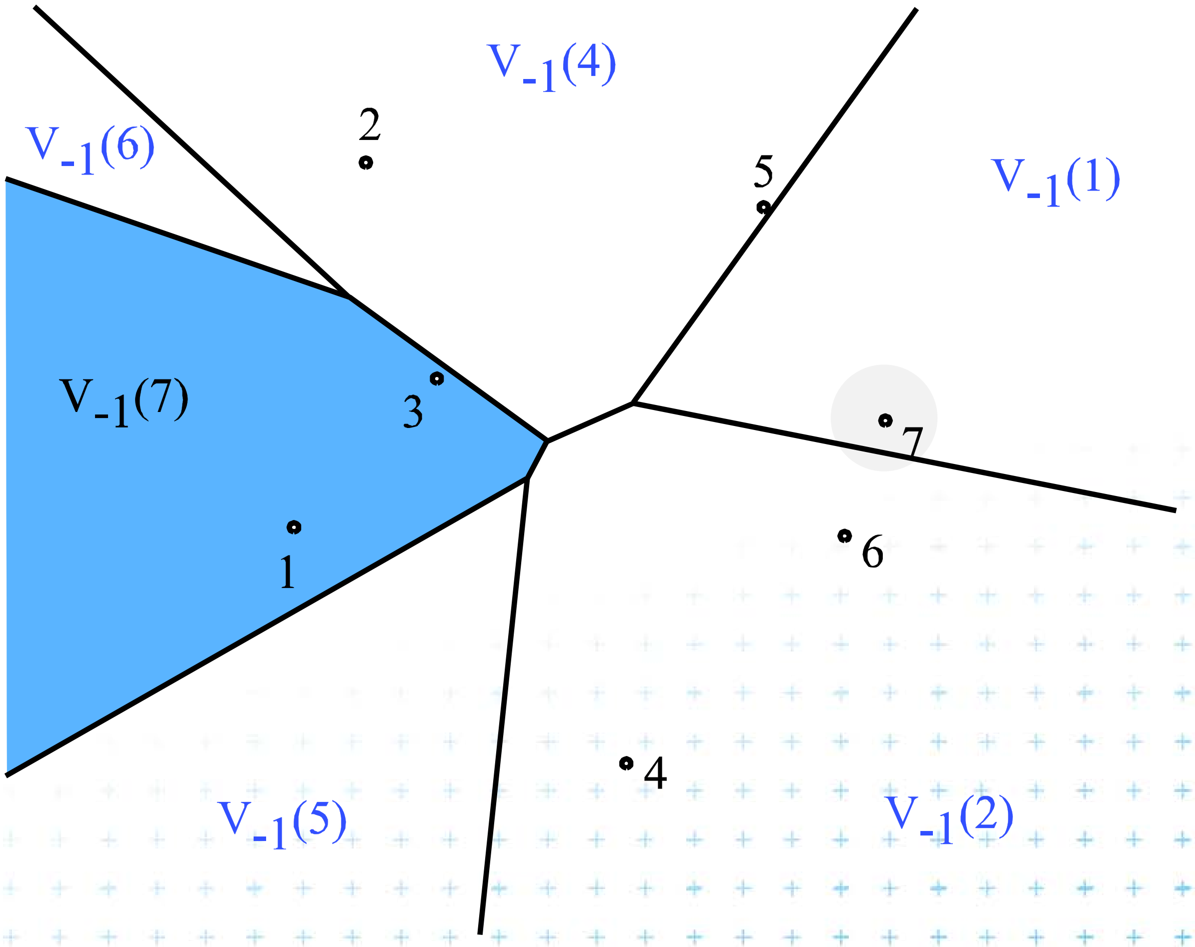


[Nandy]

Order n-1 = Farthest-point Voronoi diagram

cell $V_{-1}(7) = V_{n-1}(\{1,2,3,4,5,6\})$
= set of points in the plane farther from p_i=7 than from any other site

$Vor_{-1}(P) = Vor_{n-1}(P)$
= partition of the plane formed by the farthest point Voronoi regions, their edges, and vertices



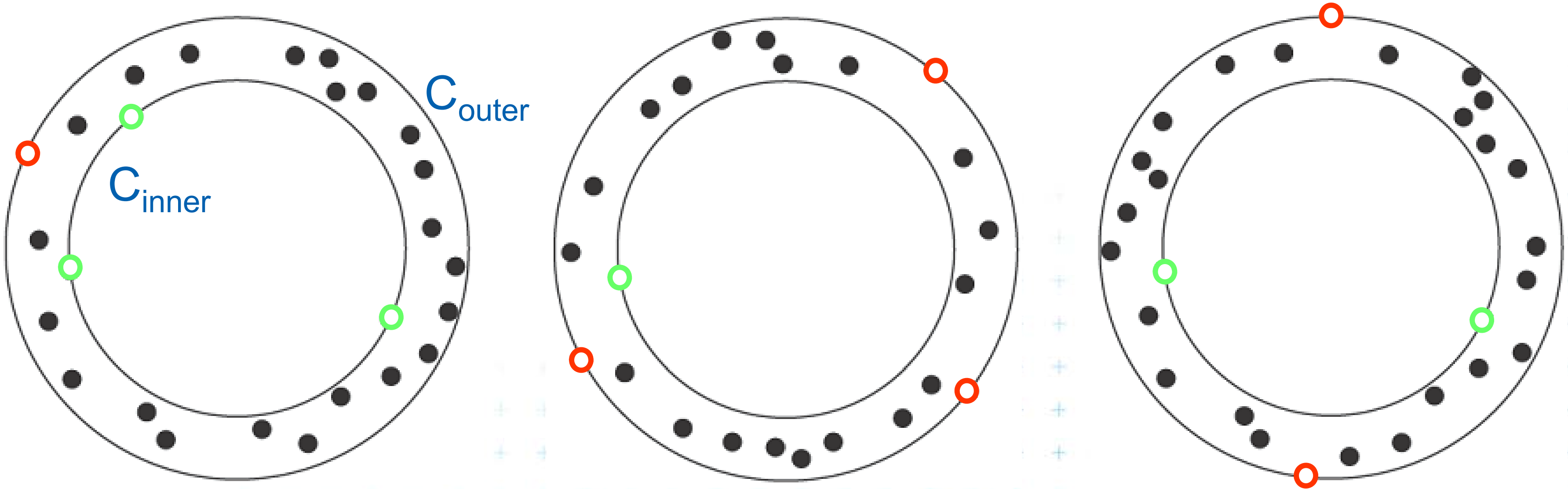
[Nandy]

Farthest-point Voronoi diagrams example

Roundness of manufactured objects

- Input: set of measured points in 2D
- Output: width of the smallest-width annulus mezikruží s nejmenší šířkou (region between two concentric circles C_{inner} and C_{outer})

Three cases to test – one will win:



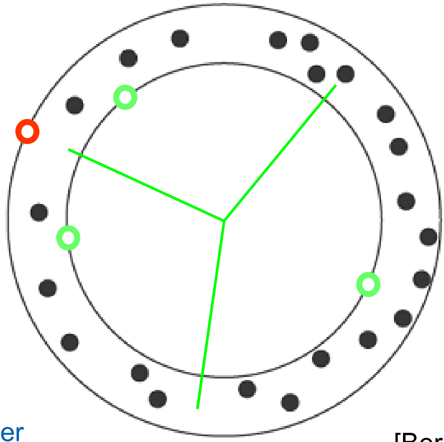
a) 3 in – 1 out b) 1 point in – 3 out c) 2 in – 2 out

[Berg]

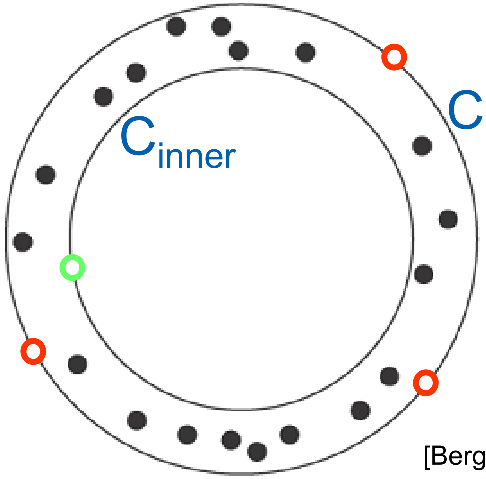
Smallest width annulus – cases with 3 pts

a) C_{inner} contains at least 3 points

- Center is the *vertex of normal Voronoi diagram* (1st order VD)
- The remaining point on C_{outer} in O(n) for each vertex
  - ⇒ not the largest (inscribed) empty circle - as discussed on seminar as we must test all VD vertices in combination with point on C outer
  - ⇒ $O(n^2)$


[Berg]

3 in – 1 out



[Berg]

1 point in – 3 out

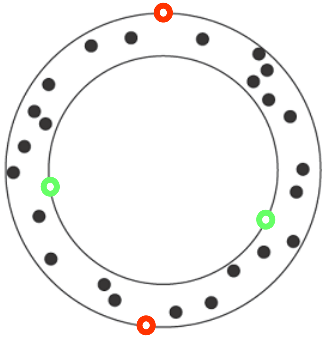
b) C_{outer} contains at least 3 points

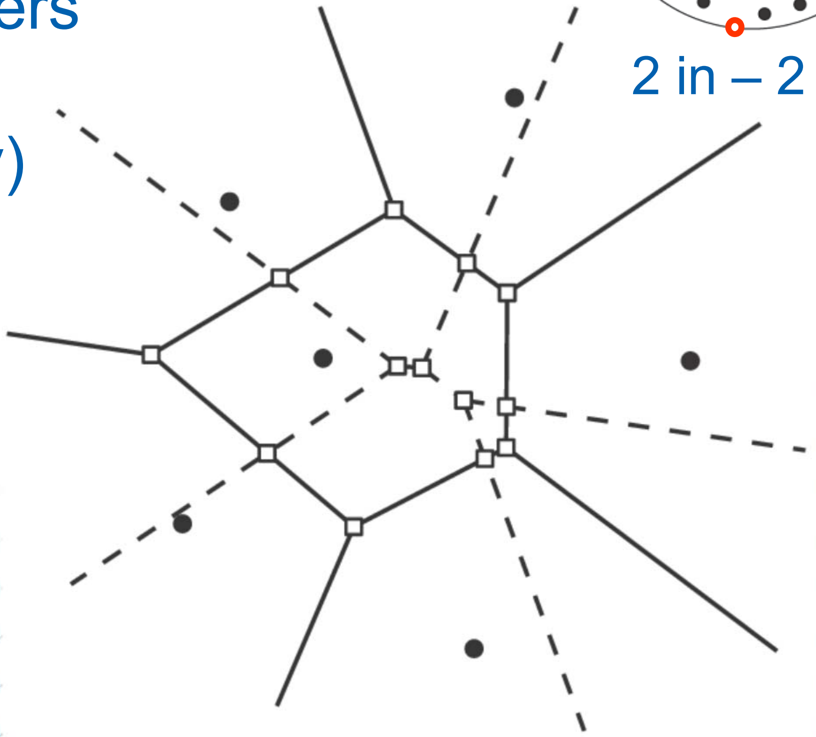
- Center is the *vertex of the farthest Voronoi diagram*
- The remaining point on C_{inner} in O(n)
  - ⇒ not the smallest enclosing circle - as discussed on seminar as we must test all vertices **in combination** with point on C inner
  - ⇒ $O(n^2)$

Smallest width annulus – case with 2+2 pts

c) C_{inner} and C_{outer} contain 2 points each

- Generate vertices of overlay of Voronoi (___) and farthest-point Voronoi (- - -) diagrams => $O(n^2)$ candidates for centers
  (we need only vertices,
  not the complete overlay)
- annulus computed in O(1) from center and 4 points (same for all 3 cases)
- $O(n^2)$

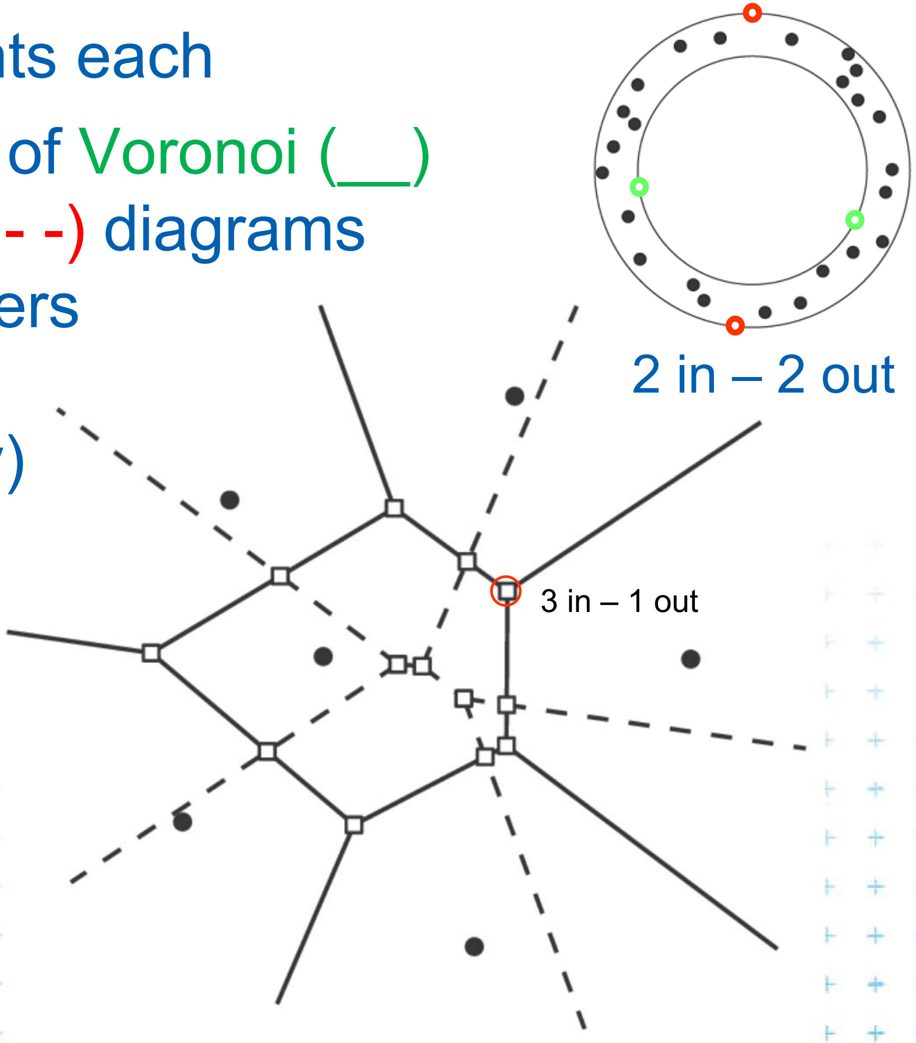

2 in – 2 out


[Berg]

Smallest width annulus – case with 2+2 pts

c) C_{inner} and C_{outer} contain 2 points each

- Generate vertices of overlay of Voronoi (__) and farthest-point Voronoi (- - -) diagrams => O(n^2) candidates for centers
  (we need only vertices,
  not the complete overlay)
- annulus computed in O(1) from center and 4 points (same for all 3 cases)
- O(n^2)

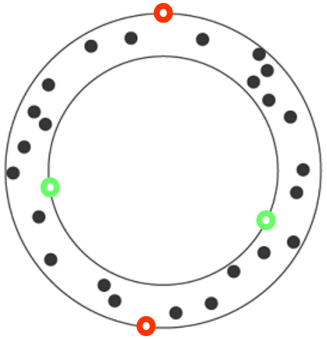


[Berg]

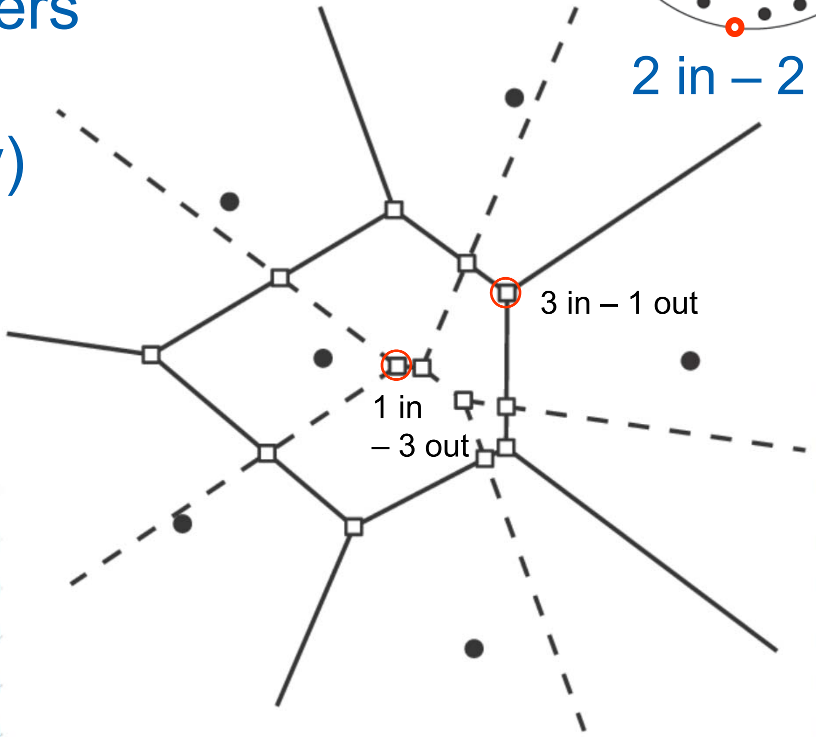
Smallest width annulus – case with 2+2 pts

c) C_{inner} and C_{outer} contain 2 points each

- Generate vertices of overlay of Voronoi (__) and farthest-point Voronoi (- - -) diagrams => $O(n^2)$ candidates for centers
  (we need only vertices, not the complete overlay)
- annulus computed in O(1) from center and 4 points (same for all 3 cases)
- $O(n^2)$



2 in – 2 out

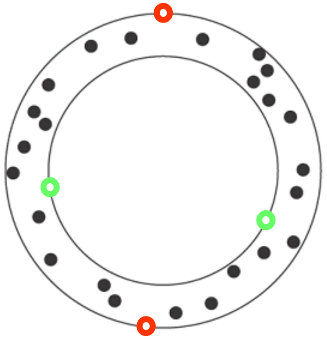


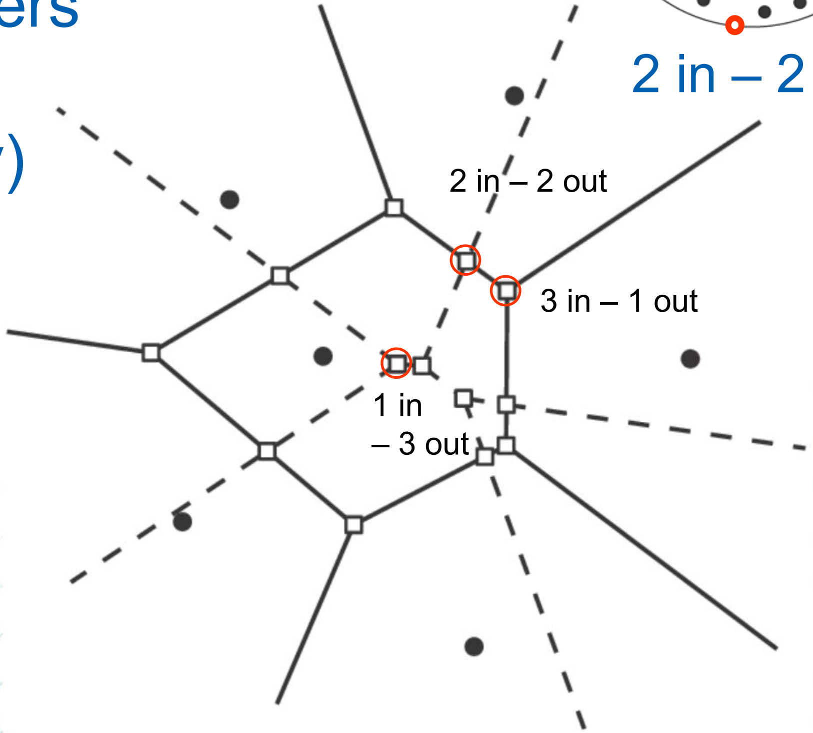
[Berg]

Smallest width annulus – case with 2+2 pts

c) C_{inner} and C_{outer} contain 2 points each

- Generate vertices of overlay of Voronoi (__) and farthest-point Voronoi (- - -) diagrams => $O(n^2)$ candidates for centers
  (we need only vertices,
  not the complete overlay)
- annulus computed in O(1) from center and 4 points (same for all 3 cases)
- $O(n^2)$





[Berg]

Smallest width annulus

Smallest-Width-Annulus

Input: Set P of n points in the plane

Output: Smallest width annulus center and radii r and R (roundness)

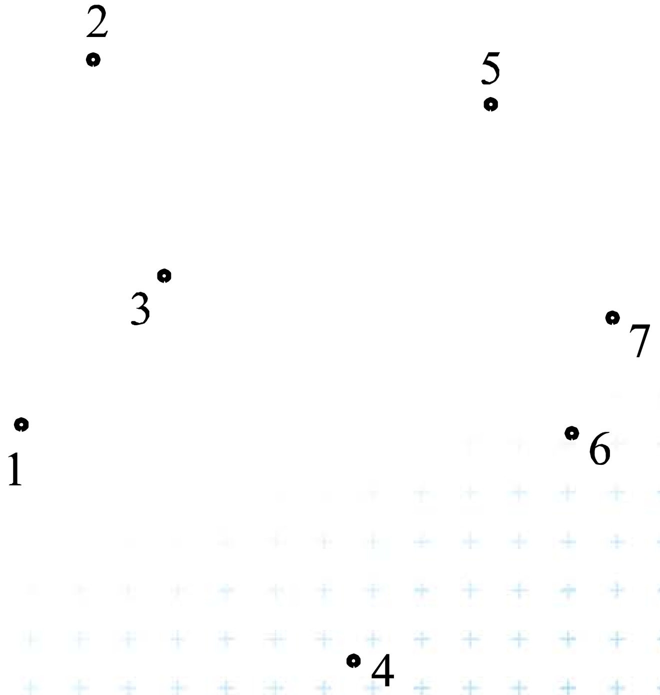
1. Compute Voronoi diagram Vor(P) and farthest-point Voronoi diagram $\mathrm{Vor}_{-1}(P)$ of P
2. For each vertex of Vor(P) (r) determine the *farthest point* (R) from P => $O(n)$ sets of four points defining candidate annuli – case a)
3. For each vertex of $\mathrm{Vor}_{-1}(P)$ (R) determine the *closest point* (r) from P => $O(n)$ sets of four points defining candidate annuli – case b)
4. For every pair of edges Vor(P) and $\mathrm{Vor}_{-1}(P)$ test if they intersect => another set of four points defining candidate annulus – c)
5. For all candidates of all three types chose the smallest-width annulus

1. $O(n \log n)$
2. $O(n^2)$
3. $O(n^2)$
4. $O(n^2)$
5. $O(n^2)$

$O(n^2)$ time using $O(n)$ storage

Farthest-point Voronoi diagram

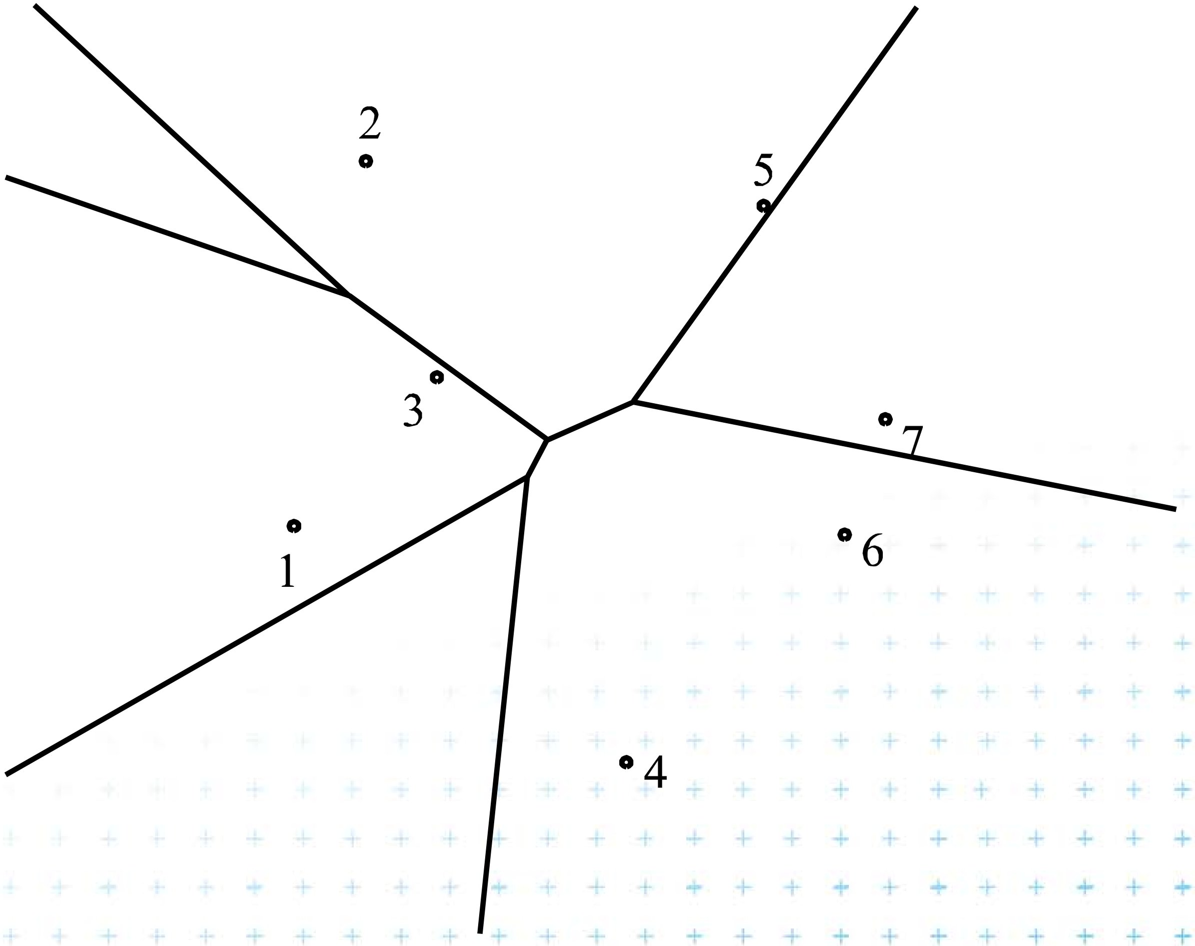
$V_{-1}(p_i)$ cell
= set of points in the plane farther from p_i than from any other site



[Nandy]

Farthest-point Voronoi diagram

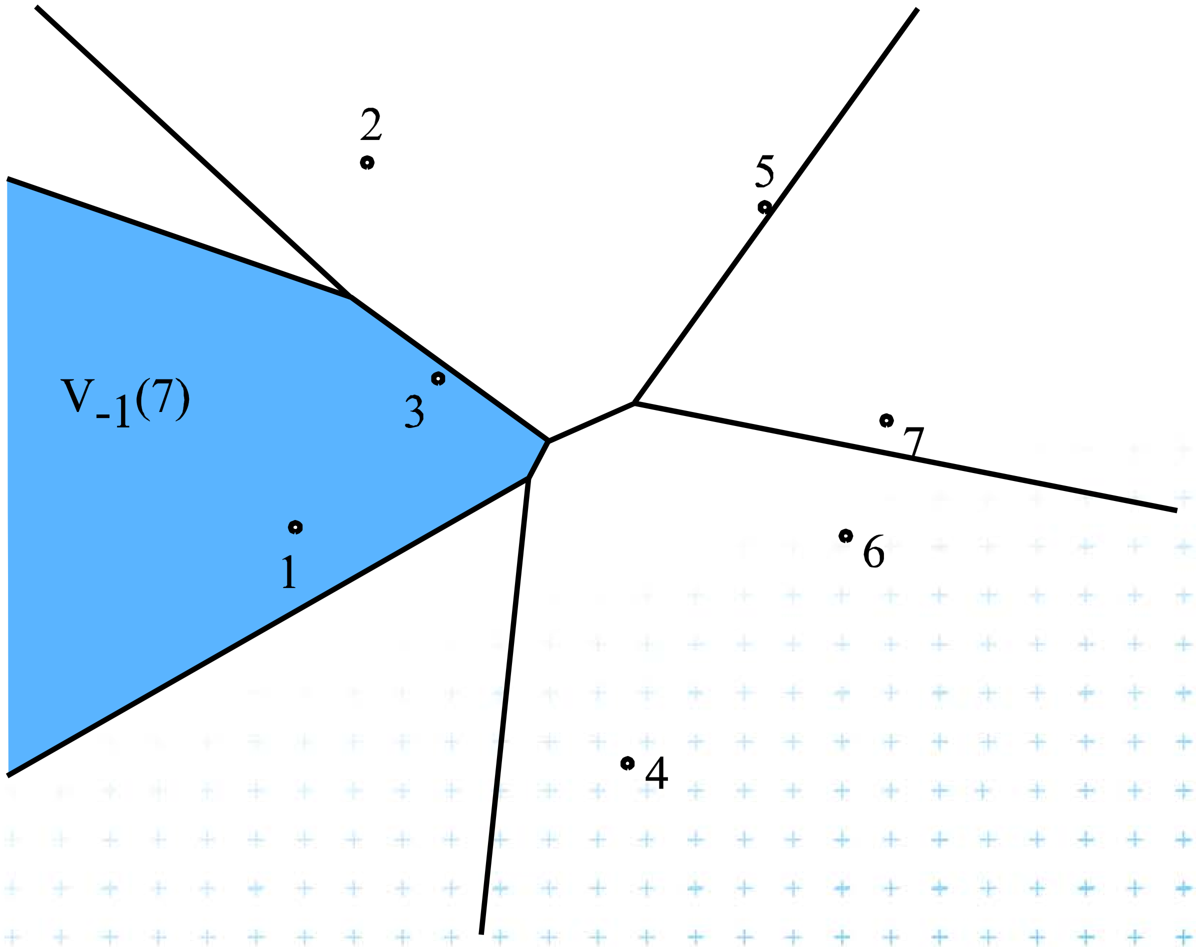
$V_{-1}(p_i)$ cell
= set of points in the plane farther from p_i than from any other site



[Nandy]

Farthest-point Voronoi diagram

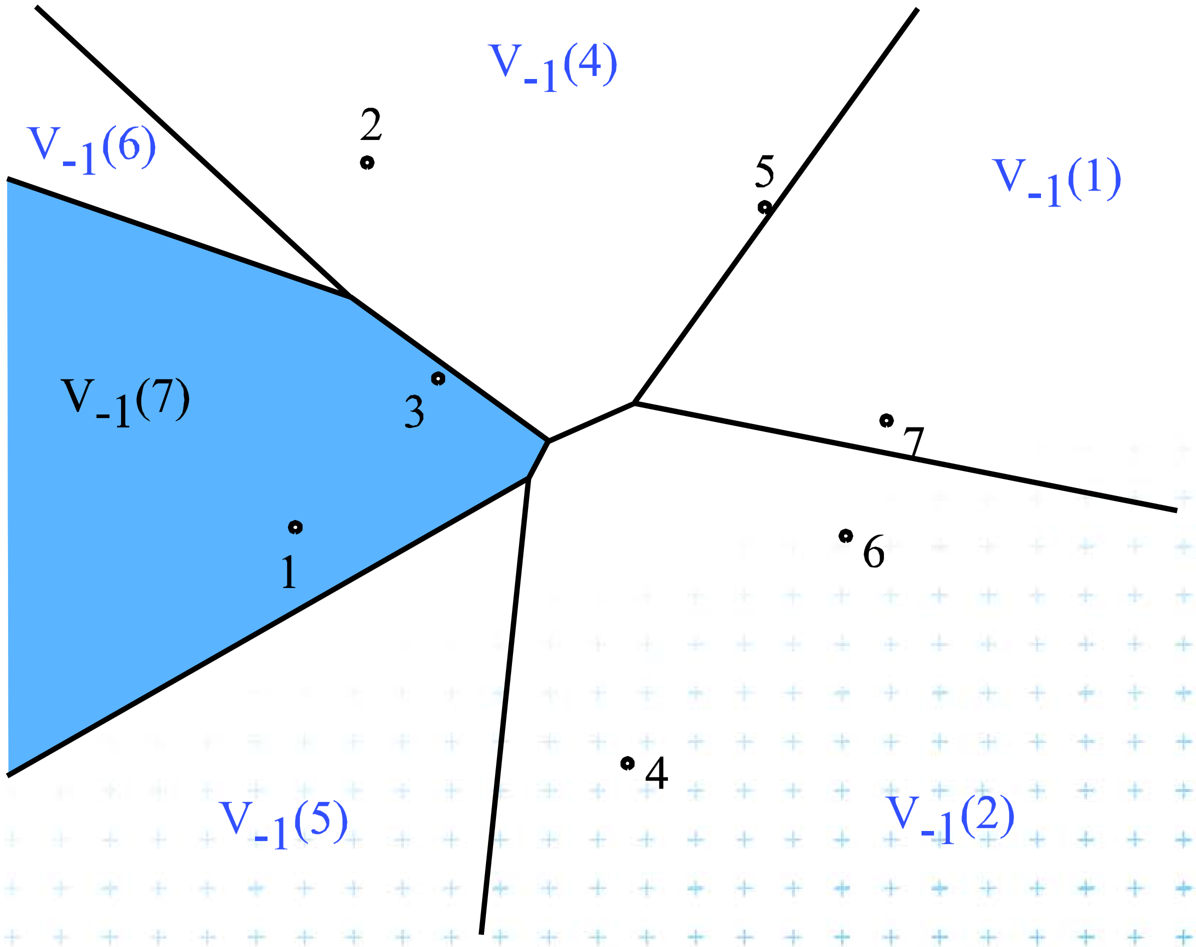
$V_{-1}(p_i)$ cell
= set of points in the plane farther from p_i than from any other site



[Nandy]

Farthest-point Voronoi diagram

$V_{-1}(p_i)$ cell
= set of points in the plane farther from p_i than from any other site

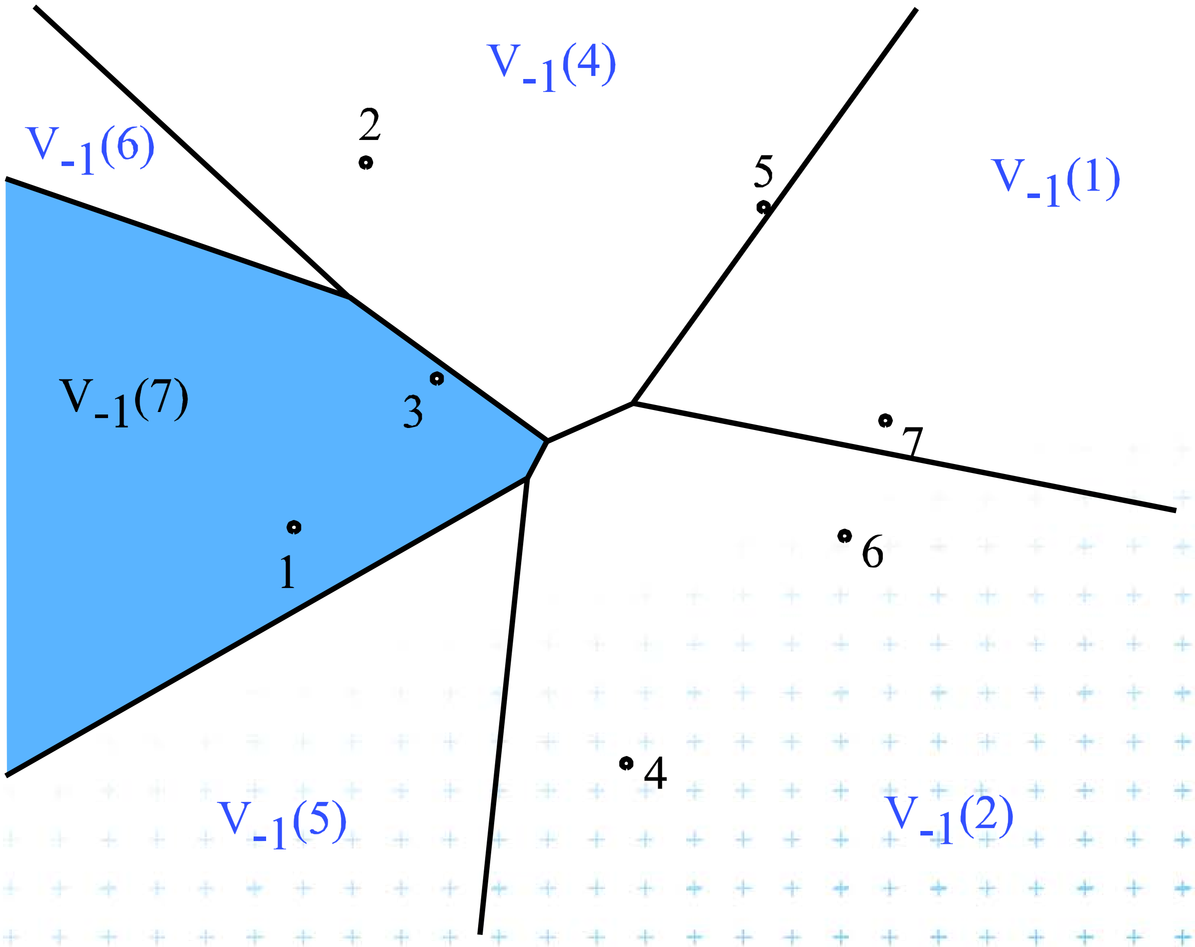


[Nandy]

Farthest-point Voronoi diagram

$V_{-1}(p_i)$ cell
= set of points in the plane farther from p_i than from any other site

$Vor_{-1}(P)$ diagram
= partition of the plane formed by the farthest point Voronoi regions, their edges, and vertices



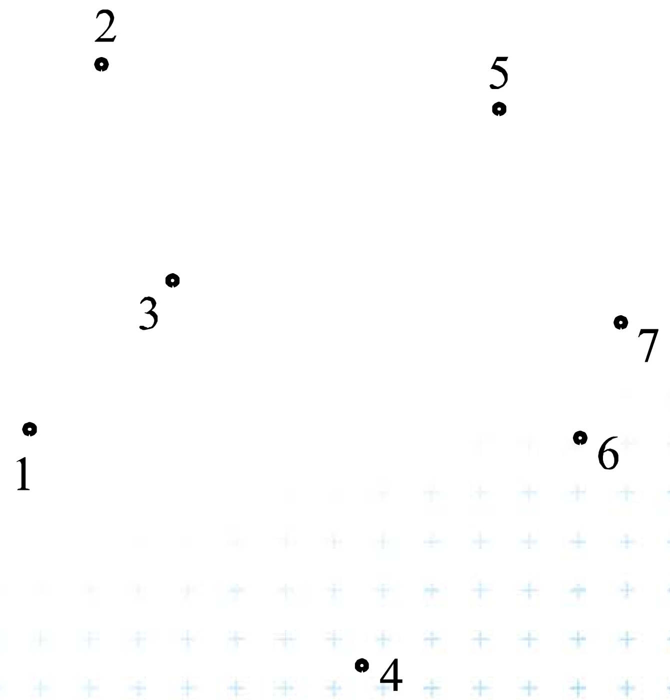
[Nandy]

Farthest-point Voronoi region (cell)

Computed as intersection of halfplanes, but we take "other sides" of bisectors

Construction of $V_{-1}(7)$

$$V_{-1} = \cap_{x=1}^{n} h(y, x),\ y \neq x$$



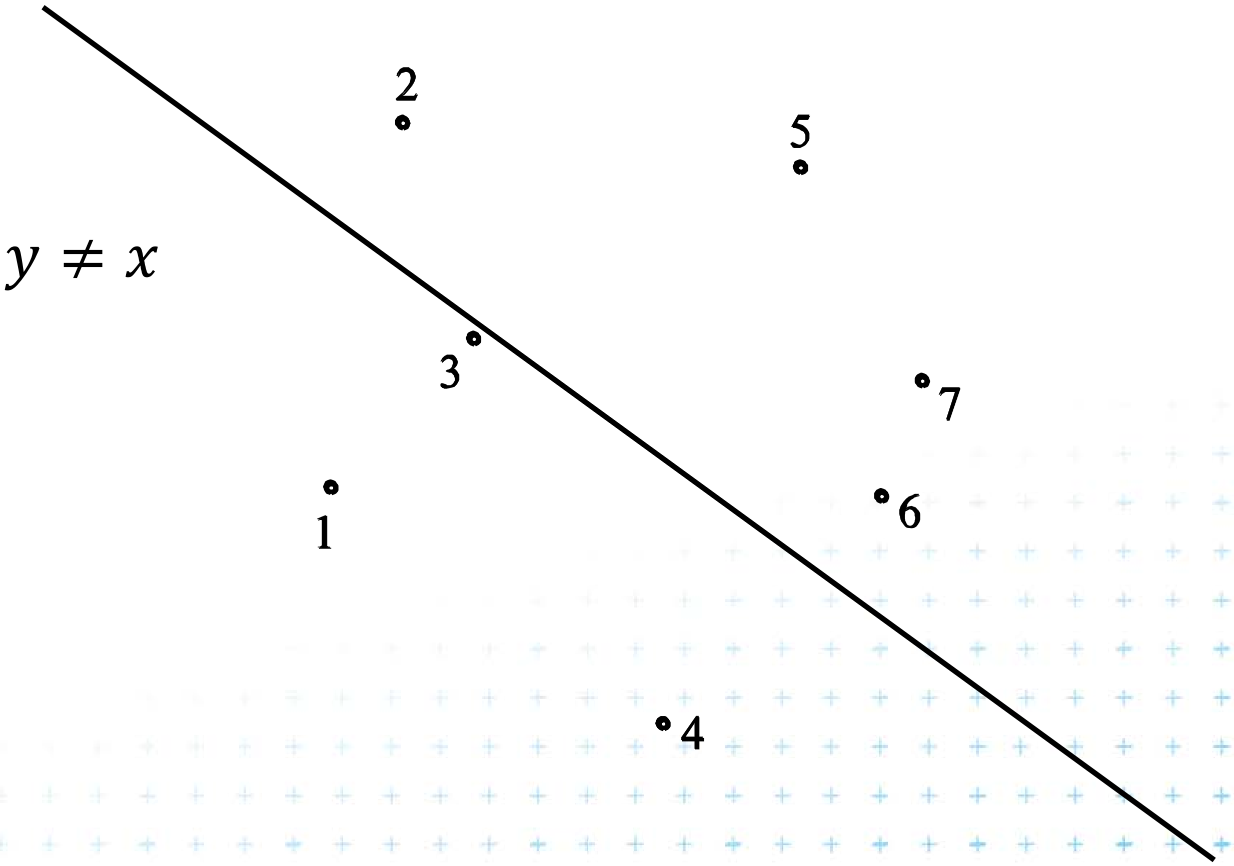
[Nandy]

Farthest-point Voronoi region (cell)

Computed as intersection of halfplanes, but we take "other sides" of bisectors

Construction of $V_{-1}(7)$

$$V_{-1} = \cap_{x=1}^{n} h(y,x),\ y \neq x$$



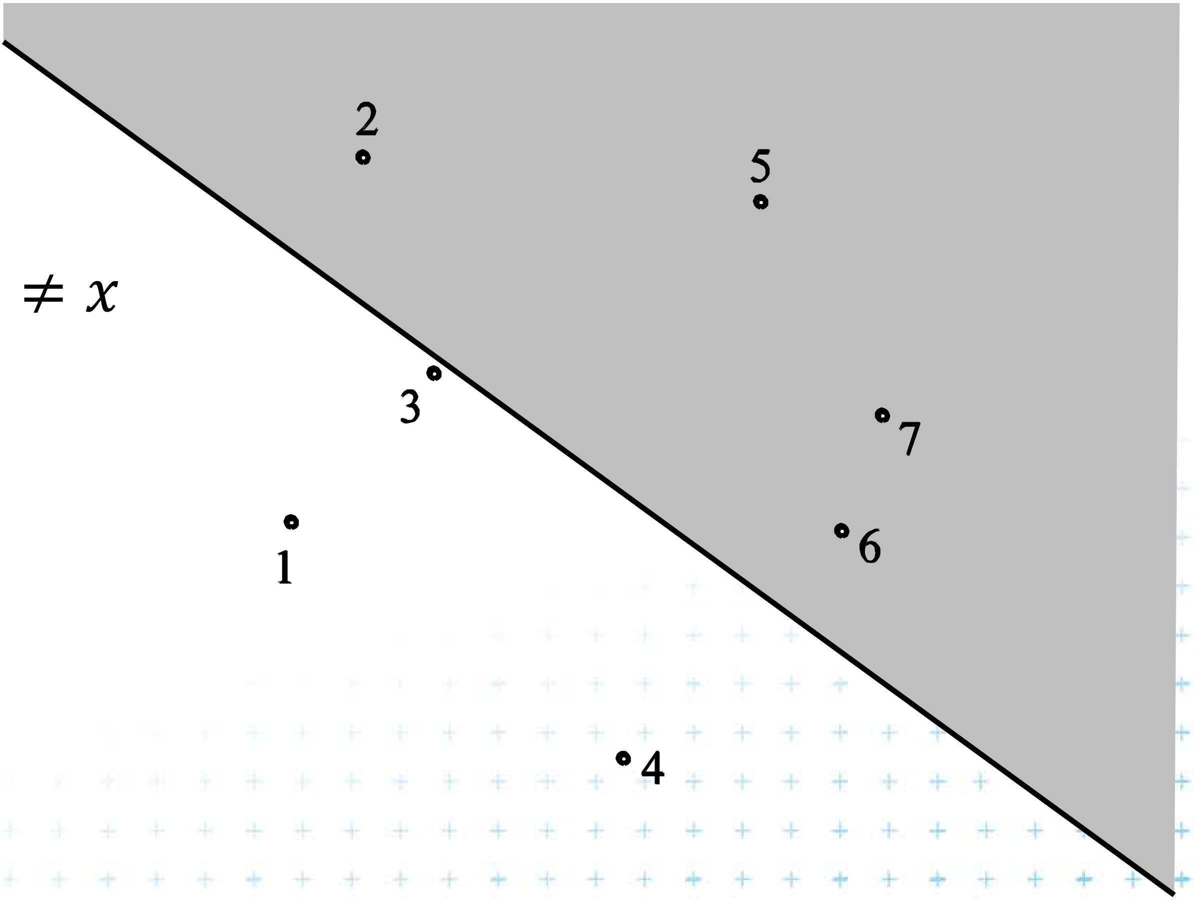
[Nandy]

Farthest-point Voronoi region (cell)

Computed as intersection of halfplanes, but we take “other sides” of bisectors

Construction of $V_{-1}(7)$

$$V_{-1} = \cap_{x=1}^{n} h(y,x),\ y \neq x$$



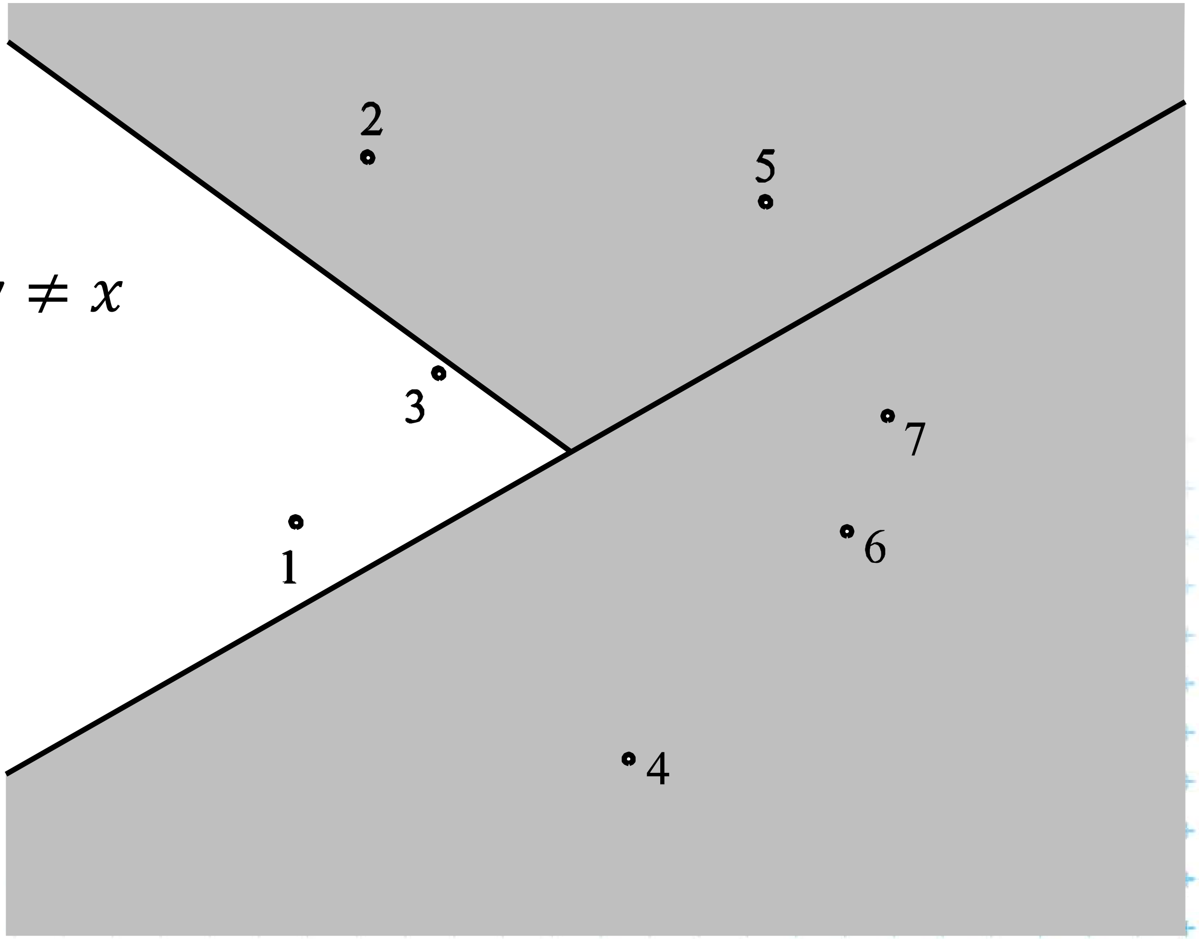
[Nandy]

Farthest-point Voronoi region (cell)

Computed as intersection of halfplanes, but we take “other sides” of bisectors

Construction of V$_{-1}$(7)

$$V_{-1} = \cap_{x=1}^{n} h(y,x),\ y \neq x$$



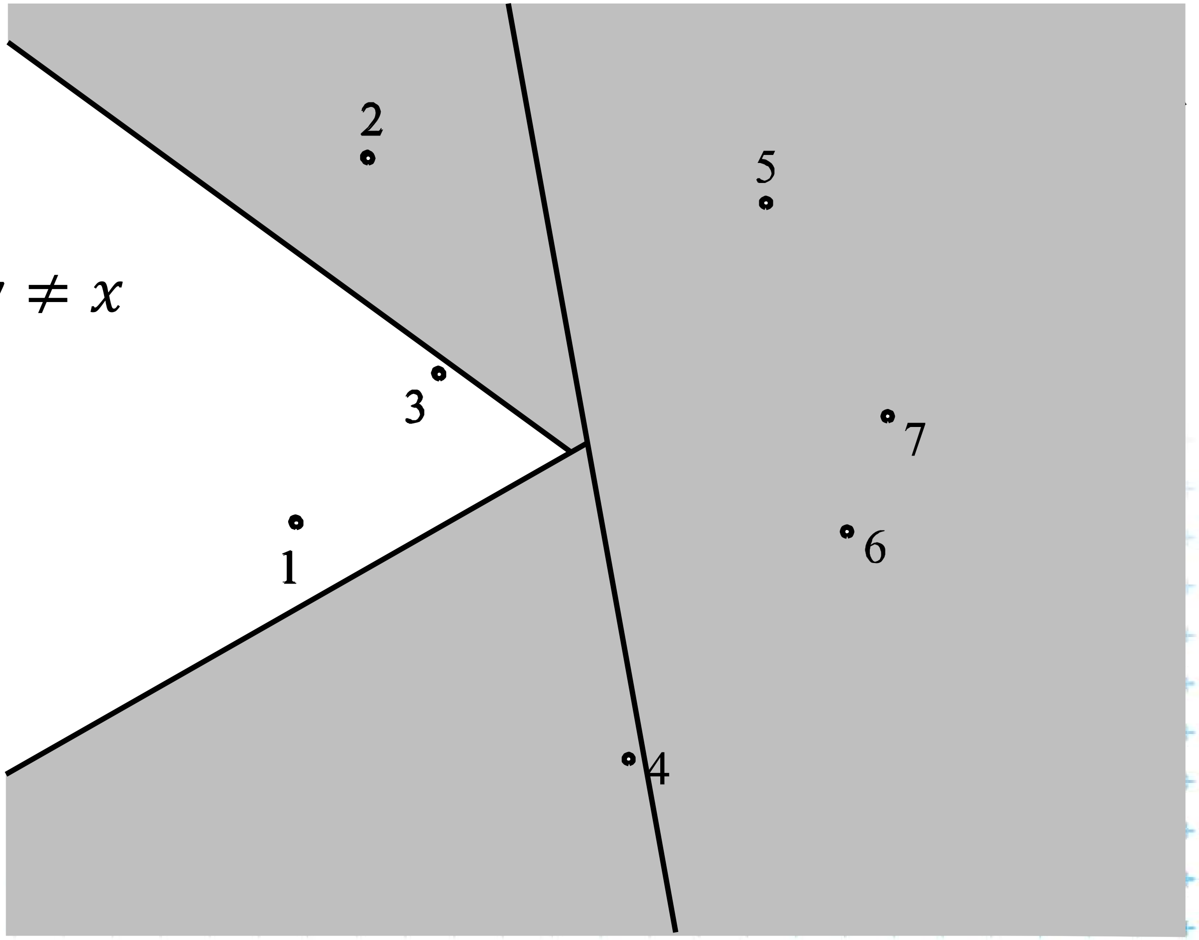
[Nandy]

Farthest-point Voronoi region (cell)

Computed as intersection of halfplanes, but we take “other sides” of bisectors

Construction of V$_{-1}$(7)

$$V_{-1} = \bigcap_{x=1}^{n} h(y,x),\ y \neq x$$



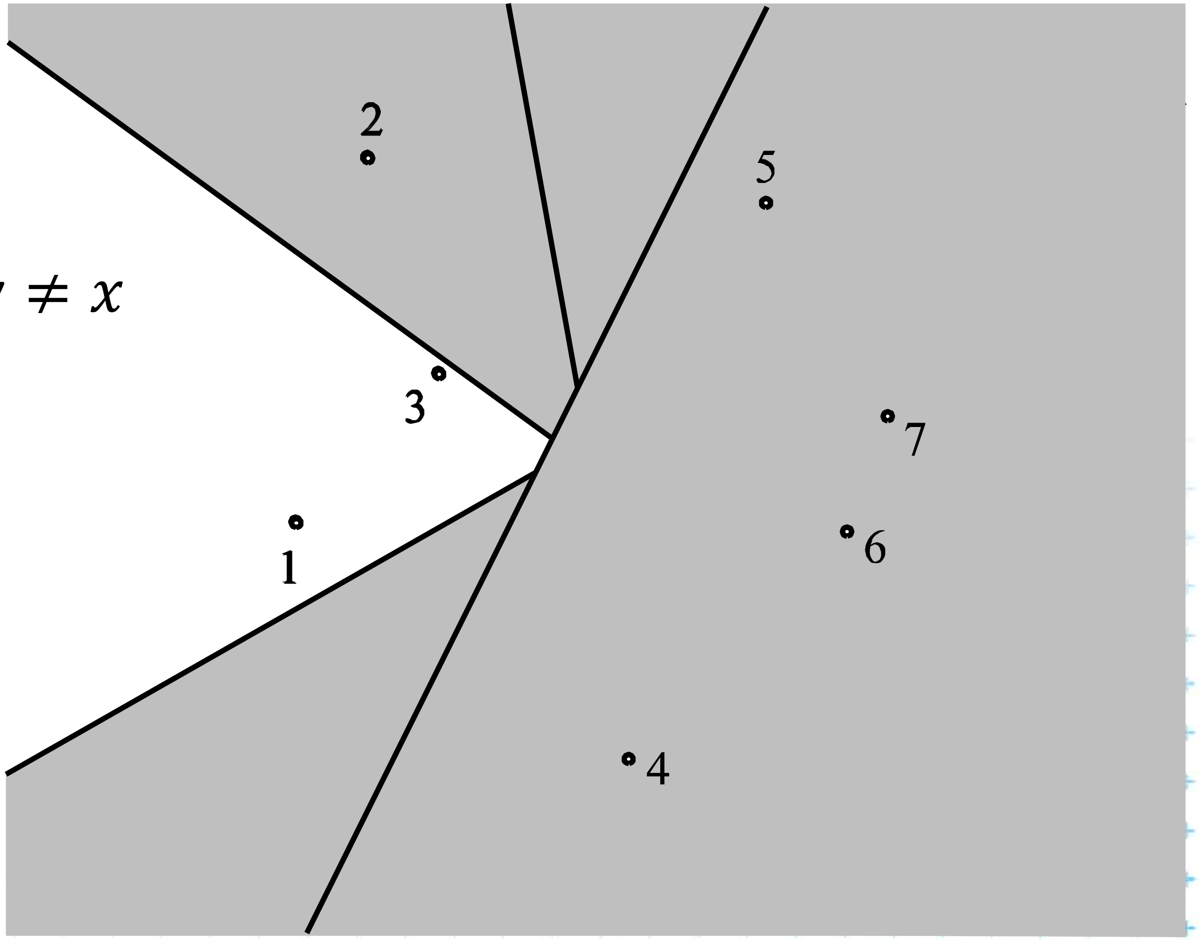
[Nandy]

Farthest-point Voronoi region (cell)

Computed as intersection of halfplanes, but we take “other sides” of bisectors

Construction of $V_{-1}(7)$

$$V_{-1} = \bigcap_{x=1}^{n} h(y,x),\ y \neq x$$



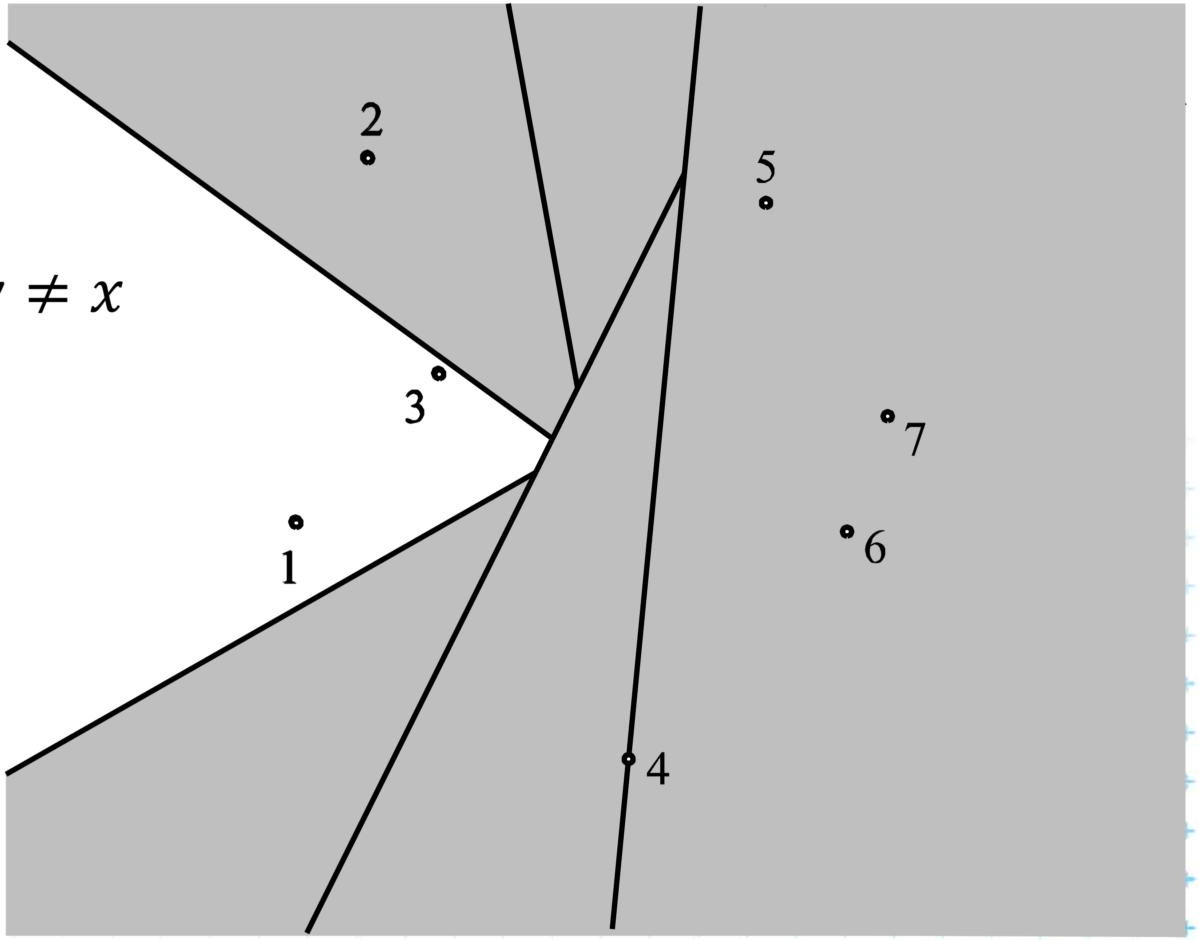
[Nandy]

Farthest-point Voronoi region (cell)

Computed as intersection of halfplanes, but we take "other sides" of bisectors

Construction of $V_{-1}(7)$

$$V_{-1} = \cap_{x=1}^{n} h(y,x),\ y \neq x$$



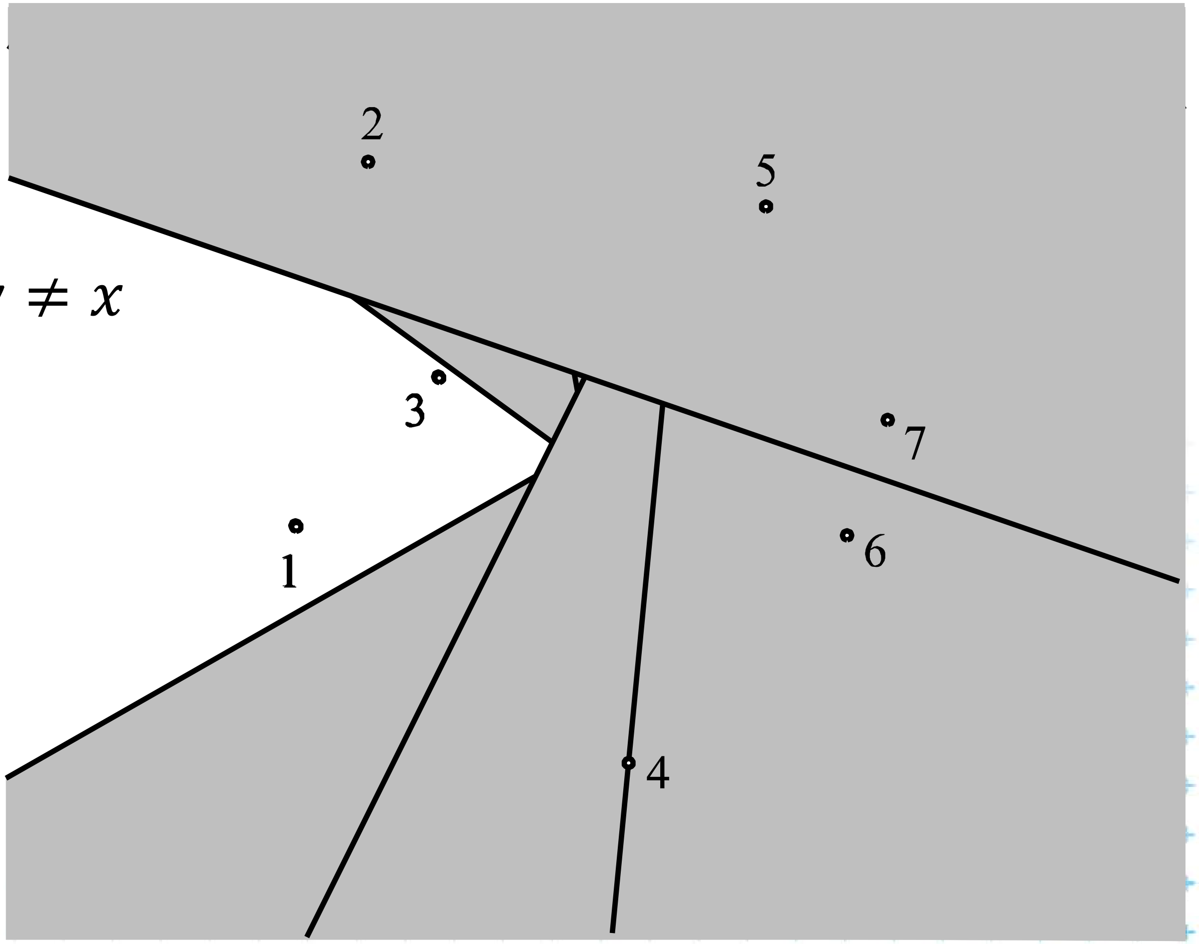
[Nandy]

Farthest-point Voronoi region (cell)

Computed as intersection of halfplanes, but we take "other sides" of bisectors

Construction of $V_{-1}(7)$

$$V_{-1} = \bigcap_{x=1}^{n} h(y,x),\ y \neq x$$



[Nandy]

Farthest-point Voronoi region (cell)

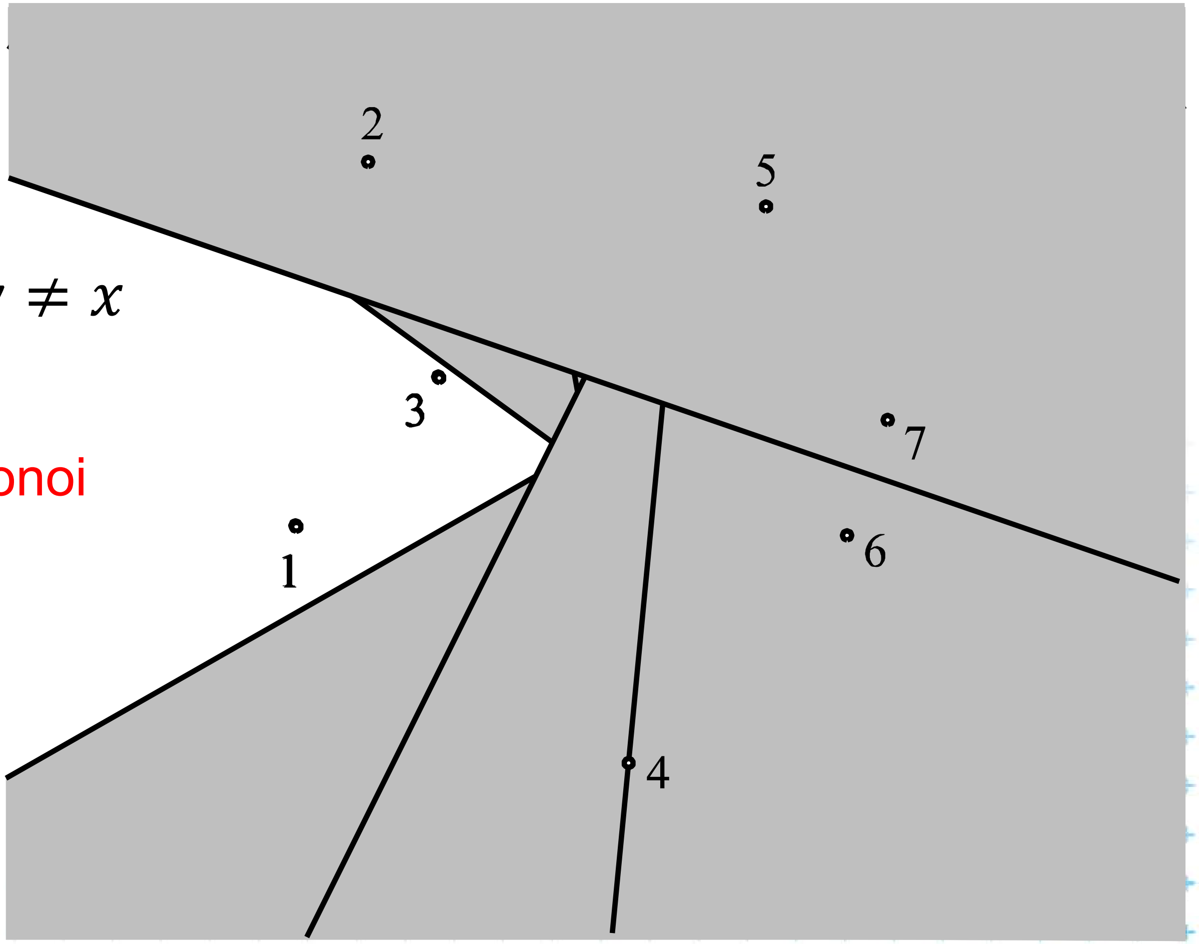
Computed as intersection of halfplanes, but we take "other sides" of bisectors

Construction of $V_{-1}(7)$

$$V_{-1} = \cap_{x=1}^{n} h(y,x),\ y \neq x$$

Property

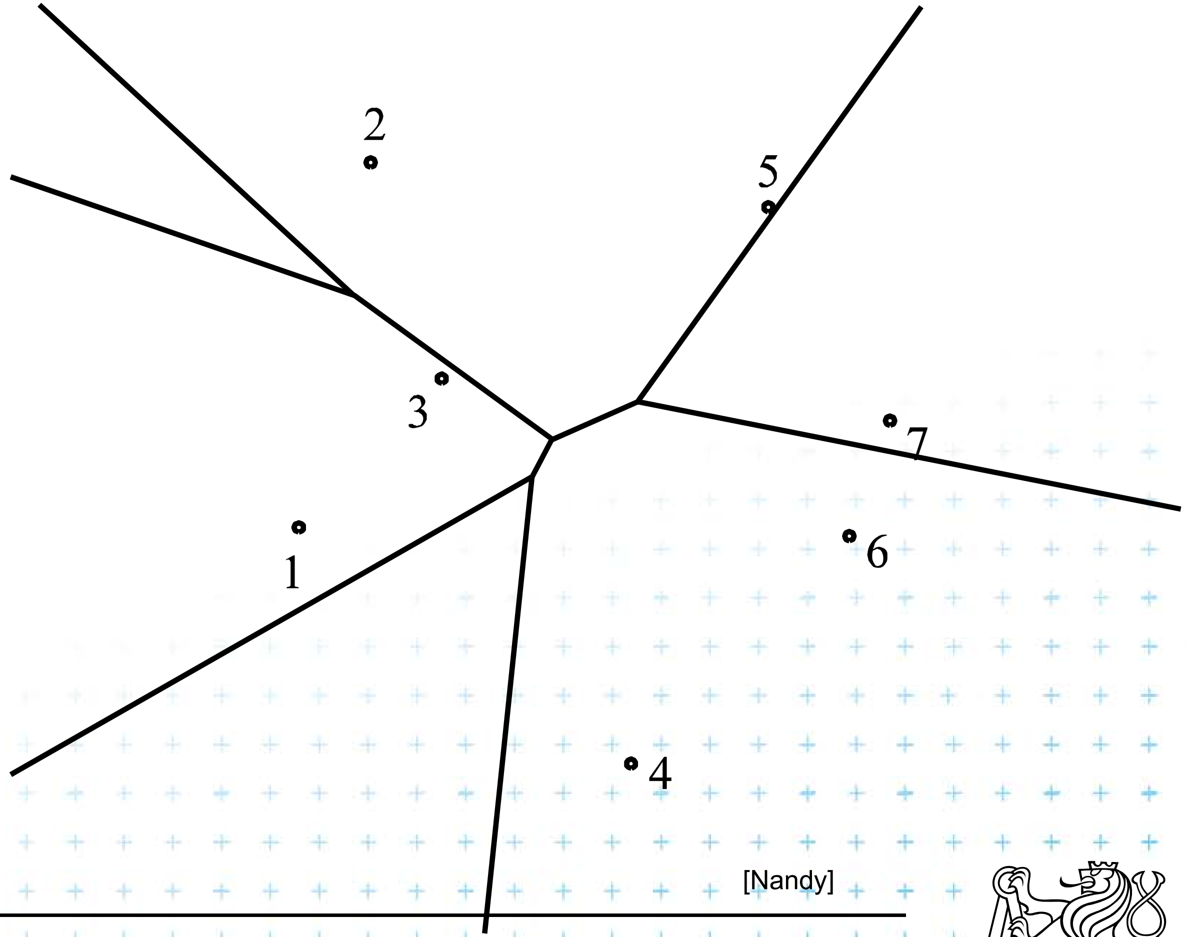
The farthest point Voronoi regions are convex and unbounded



[Nandy]

Farthest-point Voronoi region

Properties:

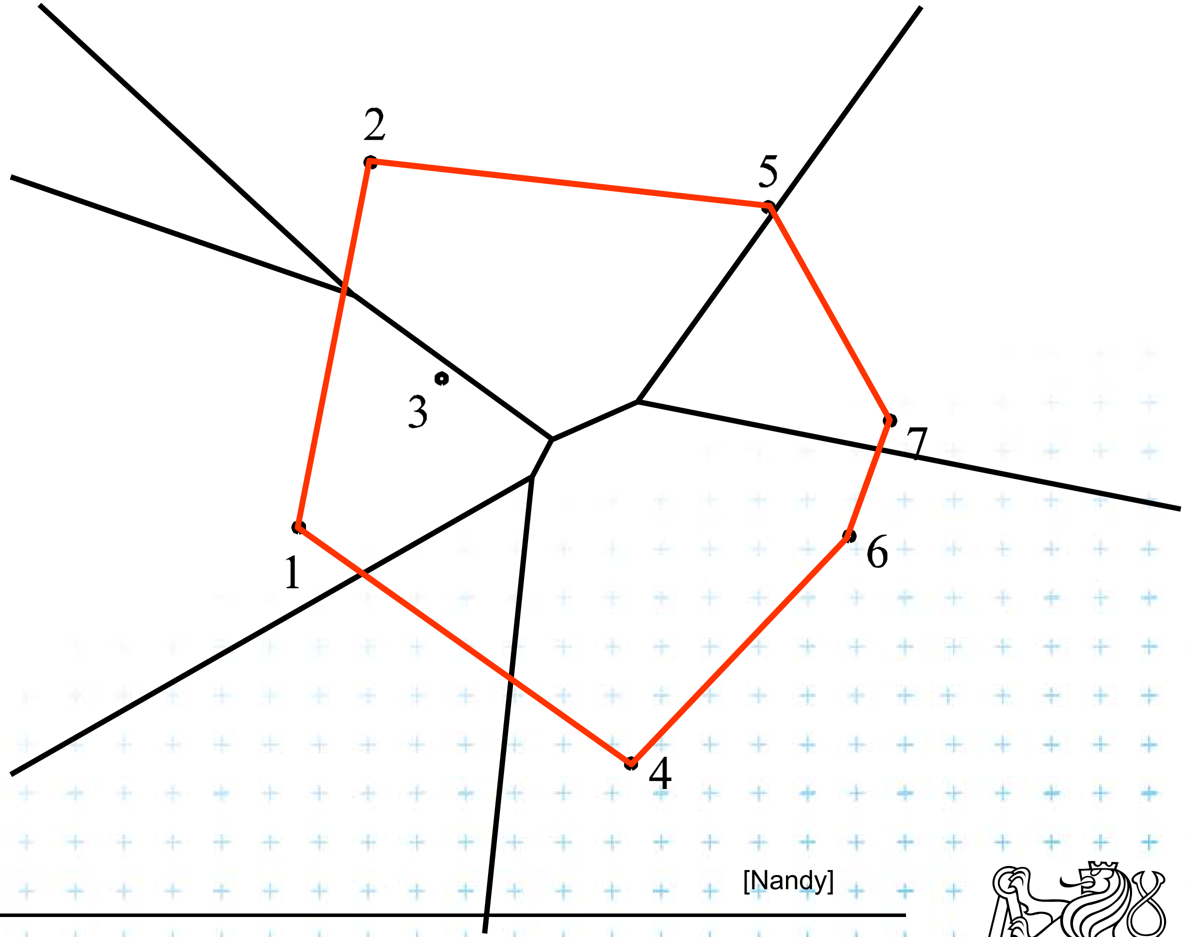


[Nandy]

Farthest-point Voronoi region

Properties:

- Only vertices of the convex hull have their cells in farthest Voronoi diagram

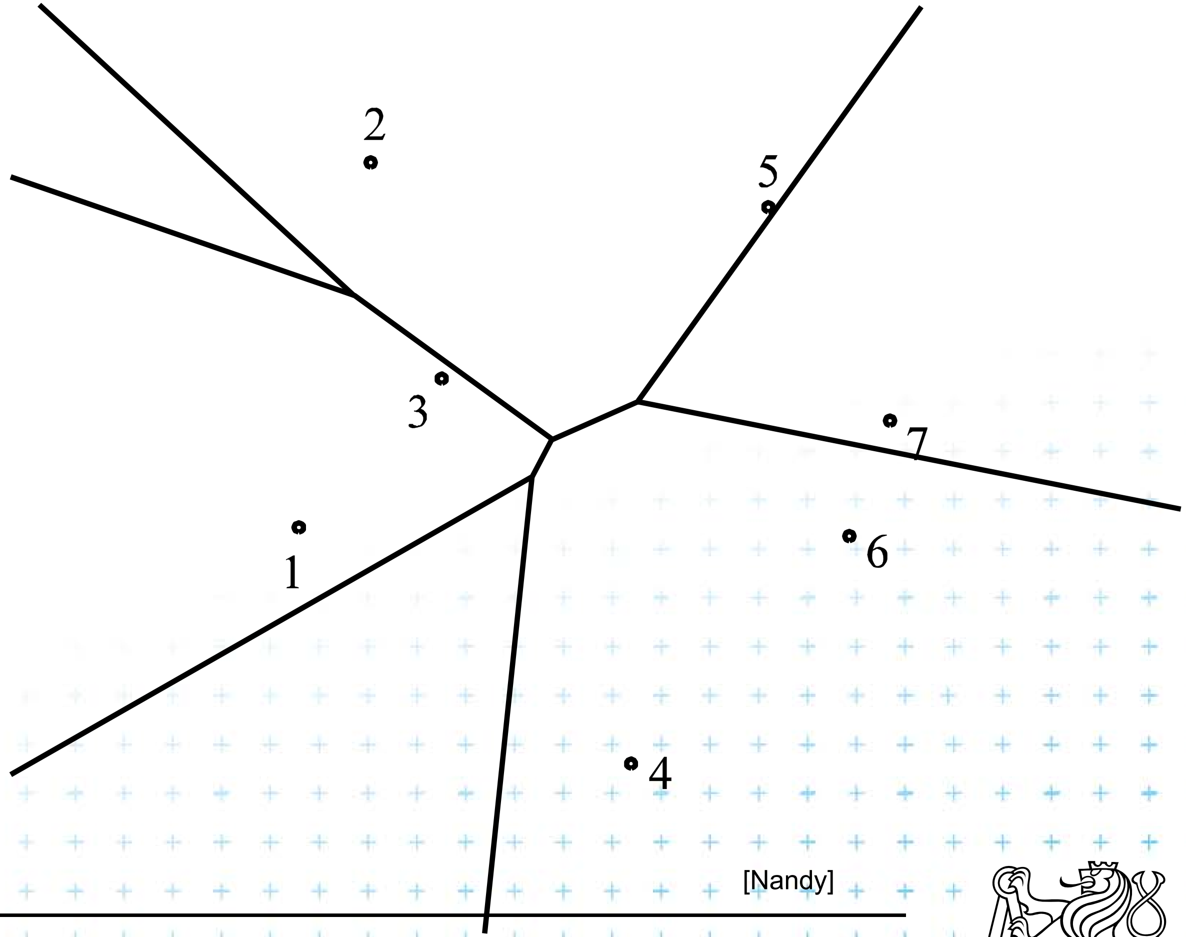


[Nandy]

Farthest-point Voronoi region

Properties:

- Only vertices of the convex hull have their cells in farthest Voronoi diagram

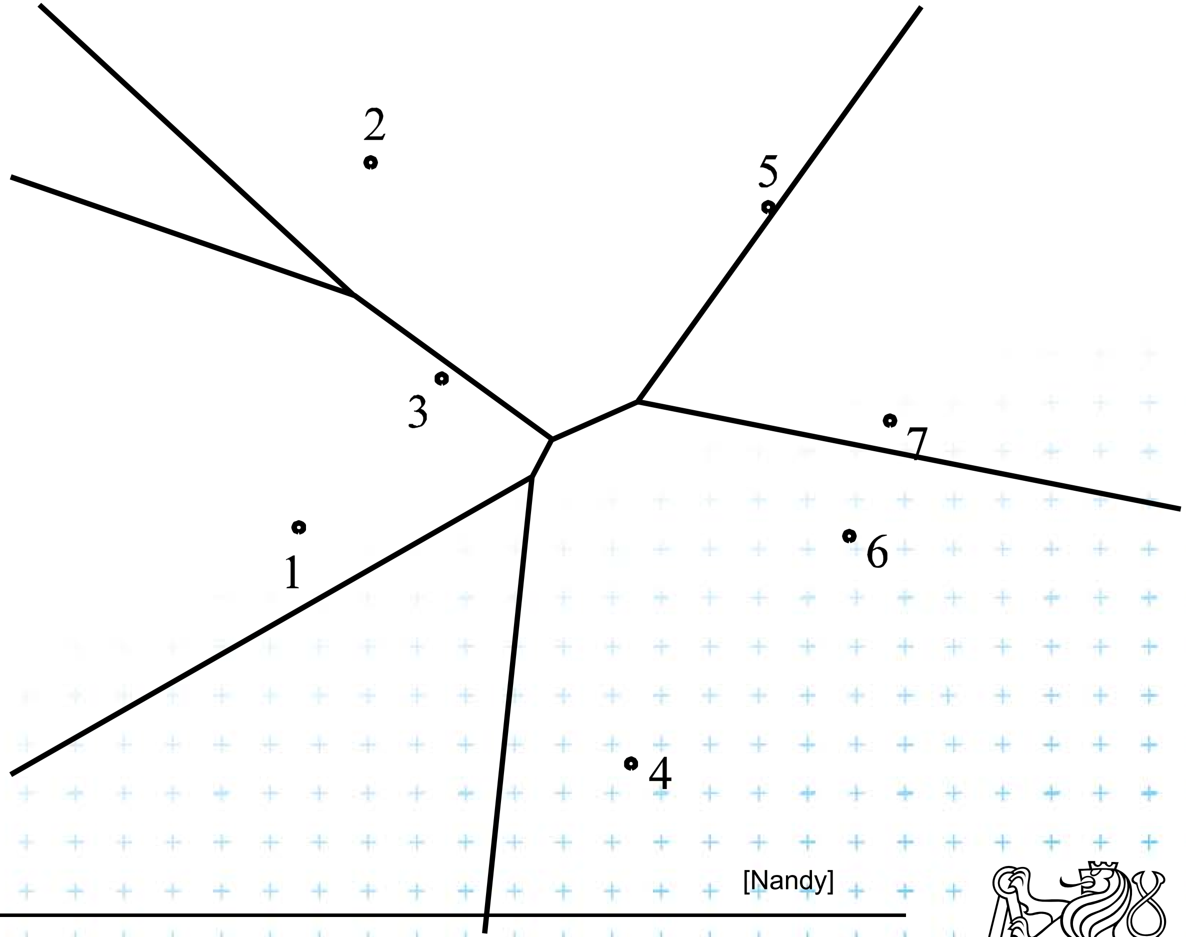


[Nandy]

Farthest-point Voronoi region

Properties:

- Only vertices of the convex hull have their cells in farthest Voronoi diagram
- The farthest point Voronoi regions are unbounded

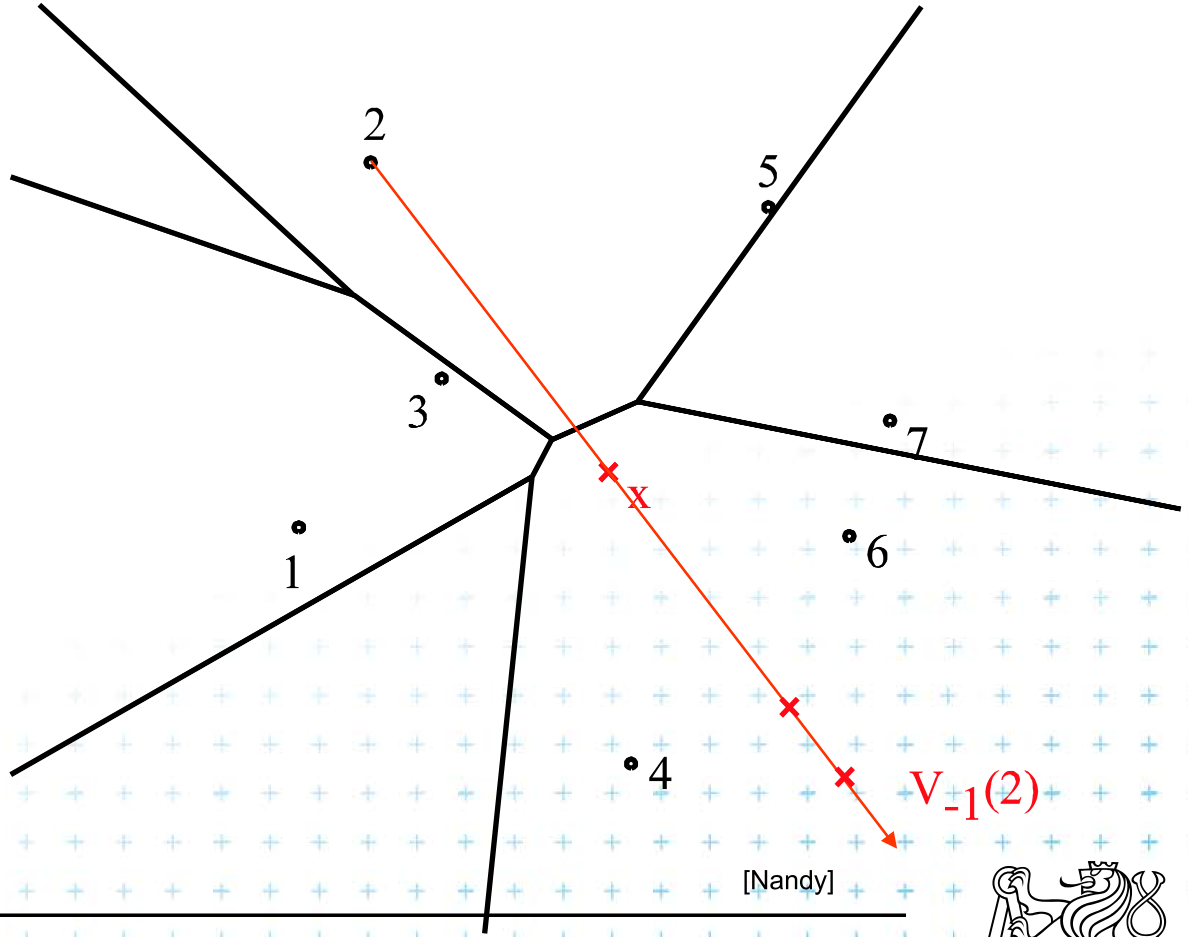


[Nandy]

Farthest-point Voronoi region

Properties:

- Only vertices of the convex hull have their cells in farthest Voronoi diagram
- The farthest point Voronoi regions are unbounded

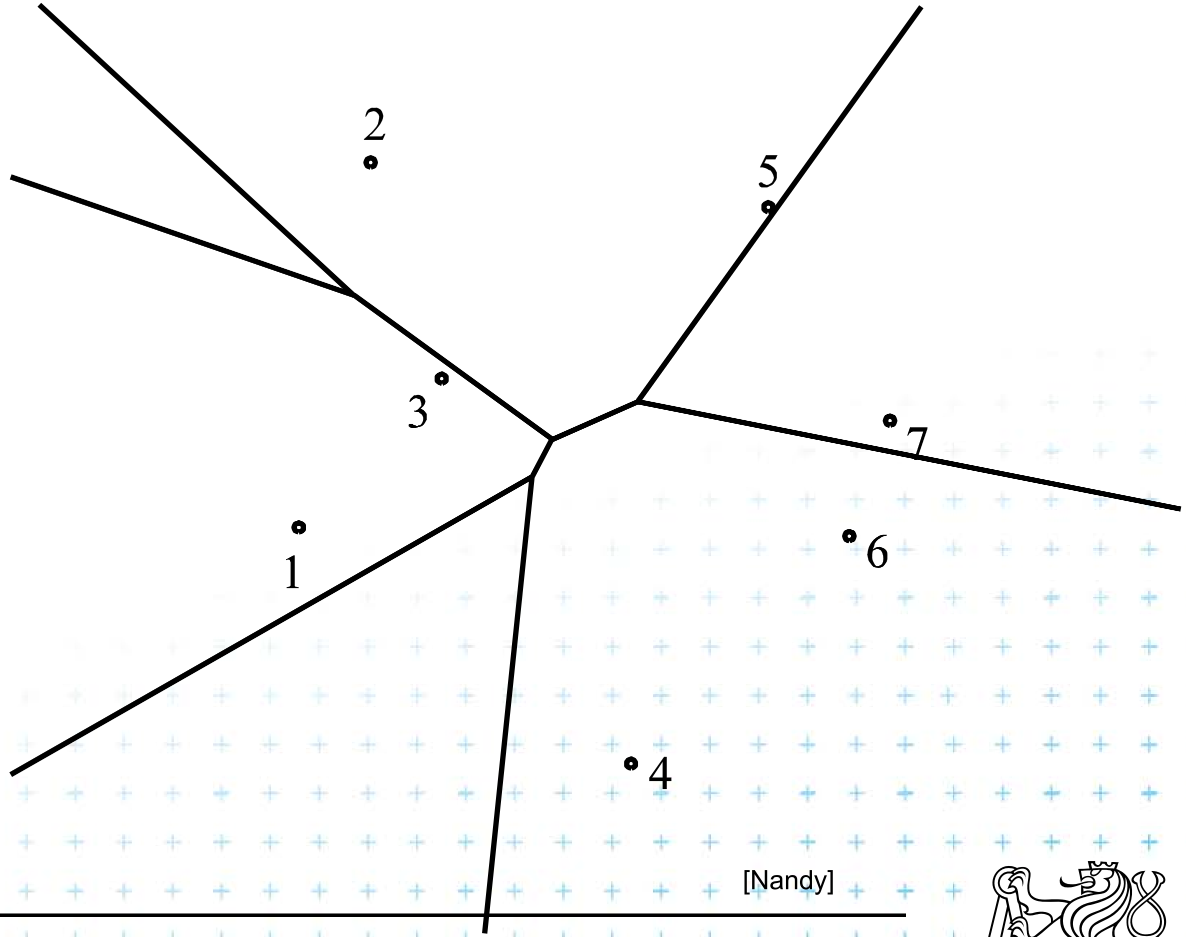


[Nandy]

Farthest-point Voronoi region

Properties:

- Only vertices of the convex hull have their cells in farthest Voronoi diagram
- The farthest point Voronoi regions are unbounded

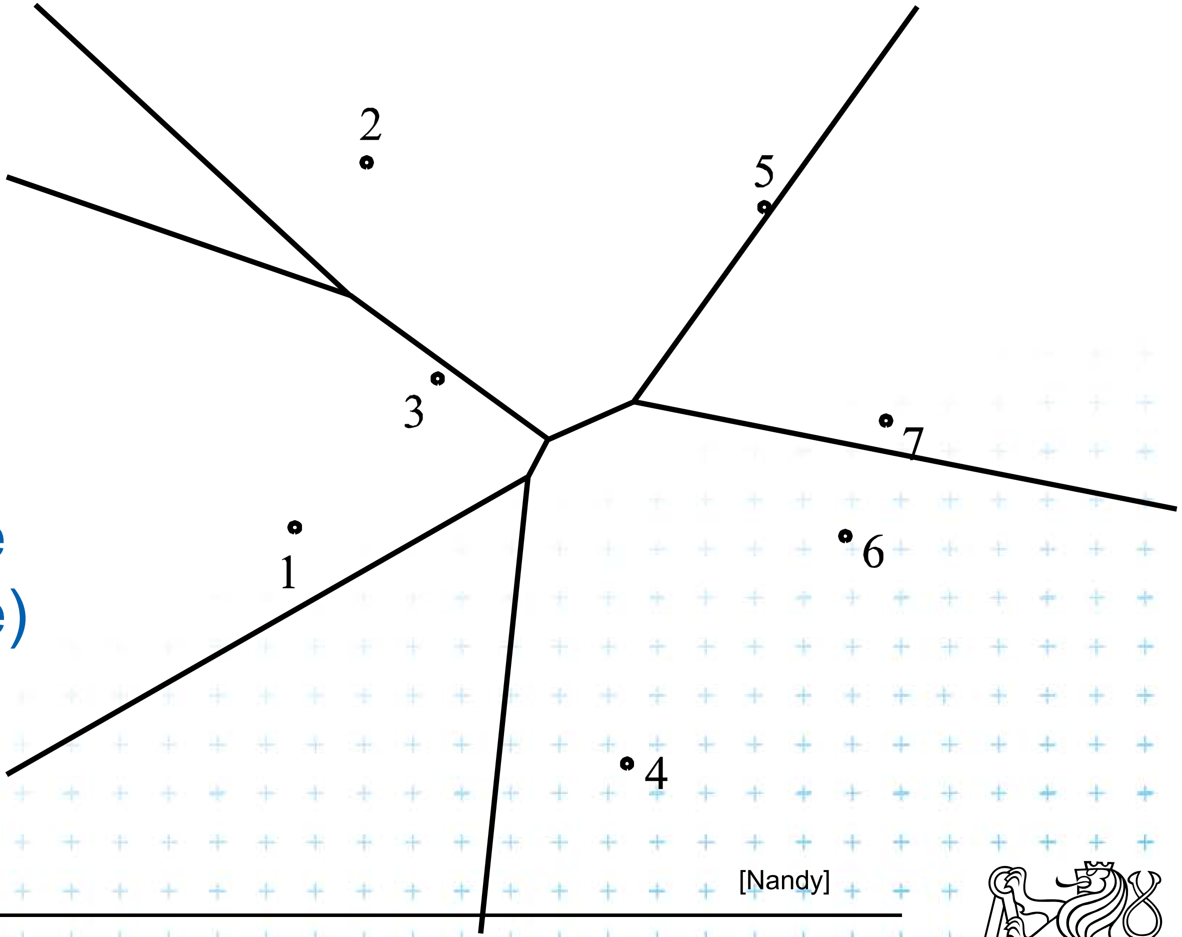


[Nandy]

Farthest-point Voronoi region

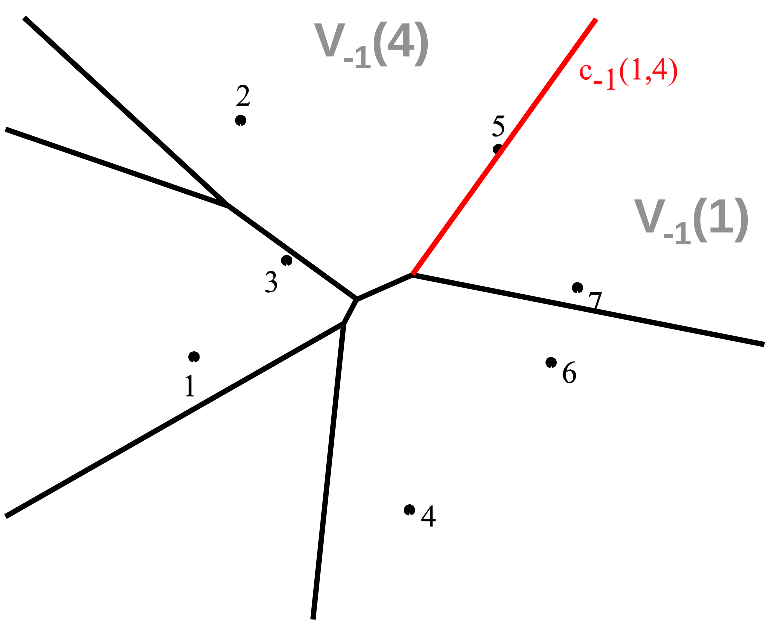
Properties:

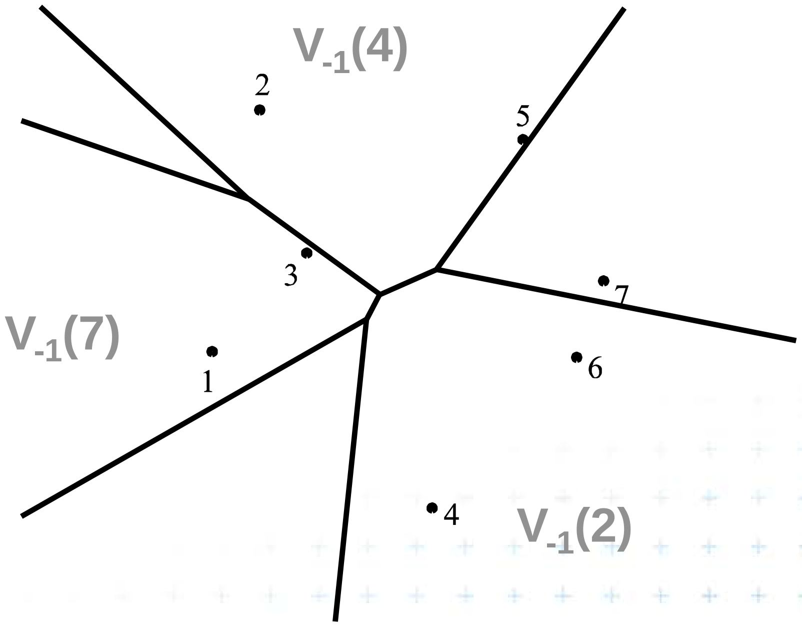
- Only vertices of the convex hull have their cells in farthest Voronoi diagram
- The farthest point Voronoi regions are unbounded
- The farthest point Voronoi edges and vertices form a tree (in the graph sense)



[Nandy]

Farthest point Voronoi edges and vertices

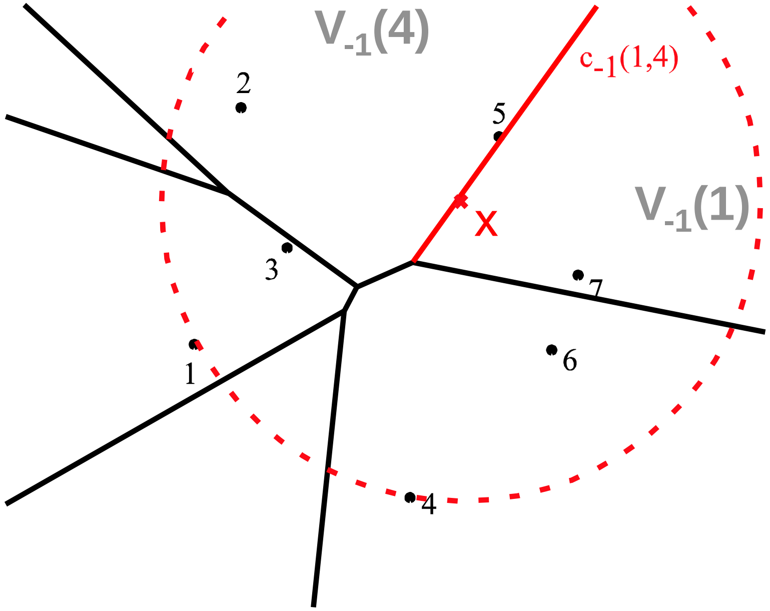


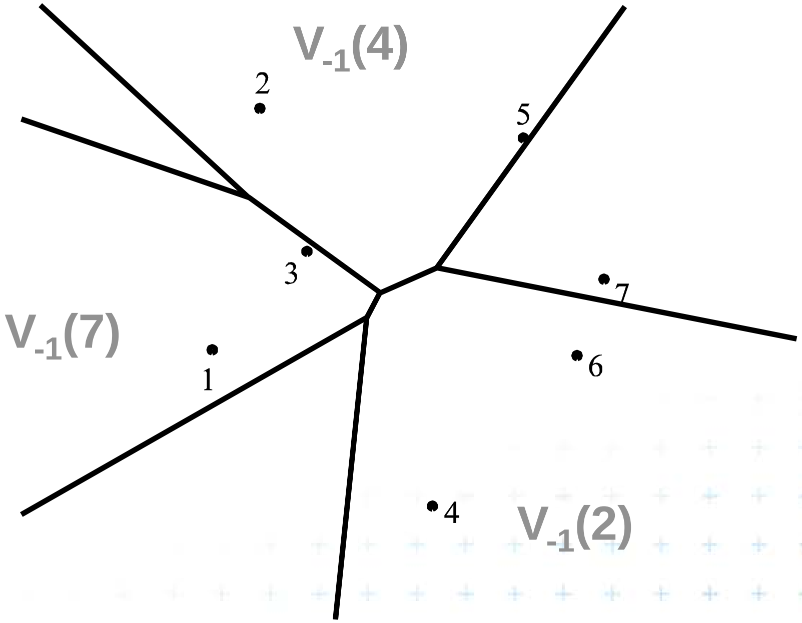


edge : set of points equidistant
from 2 sites and closer to
all the other sites

[Nandy]

Farthest point Voronoi edges and vertices

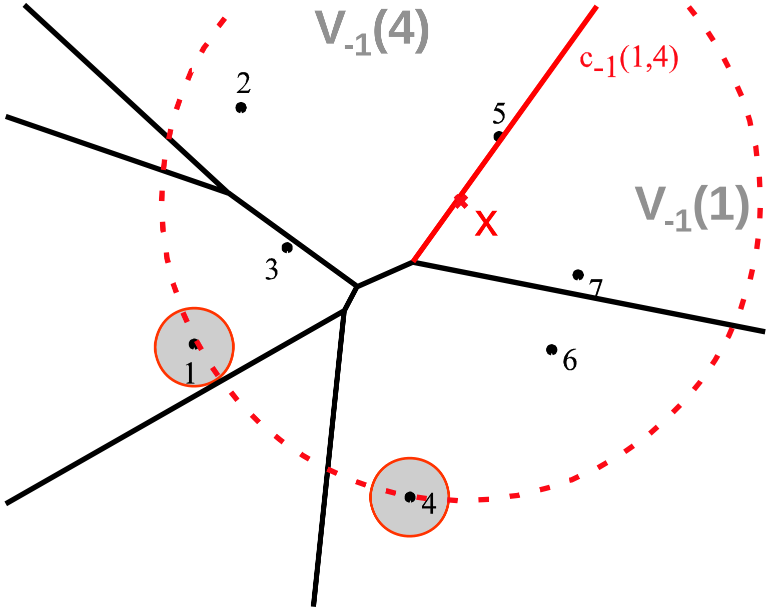


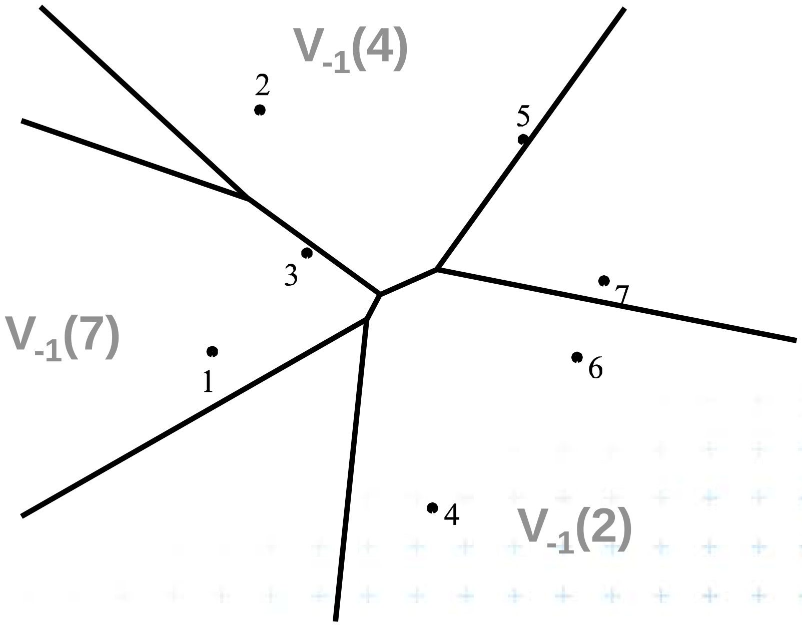


edge : set of points equidistant from 2 sites and closer to all the other sites

[Nandy]

Farthest point Voronoi edges and vertices

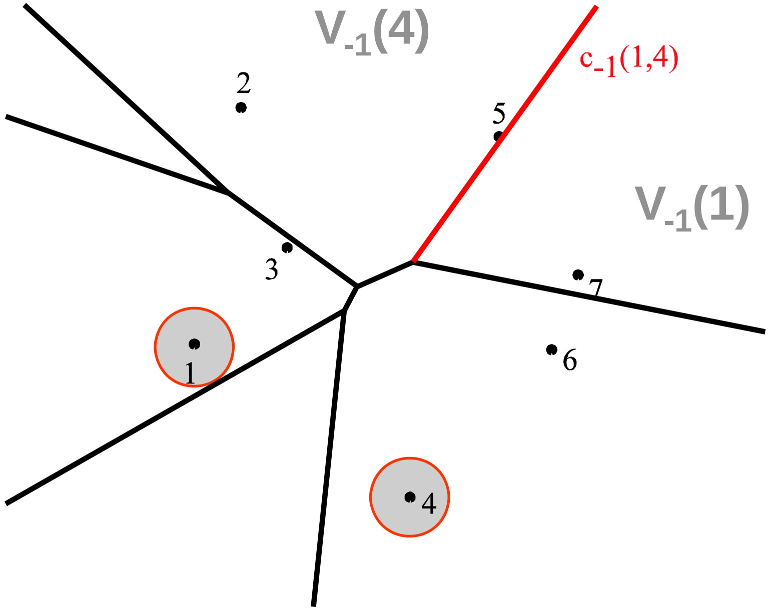


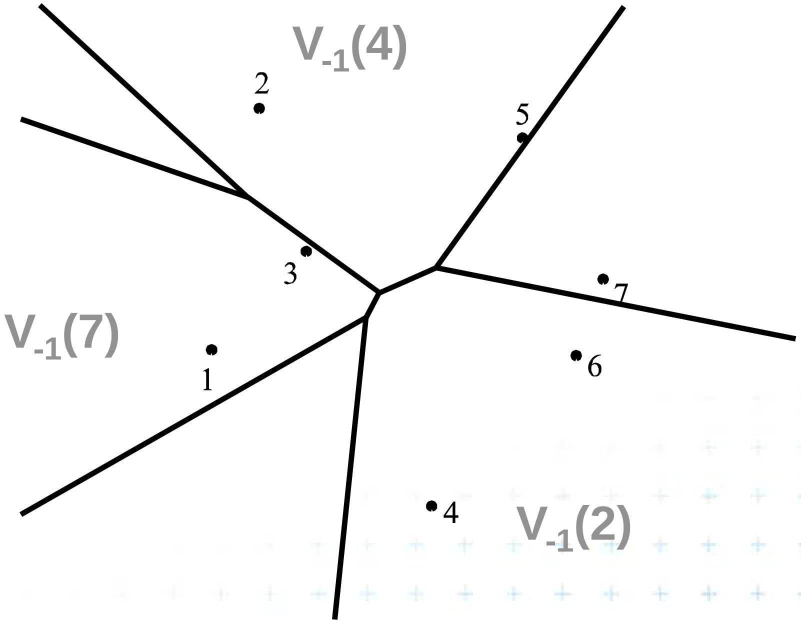


edge : set of points equidistant
from 2 sites and closer to
all the other sites

[Nandy]

Farthest point Voronoi edges and vertices

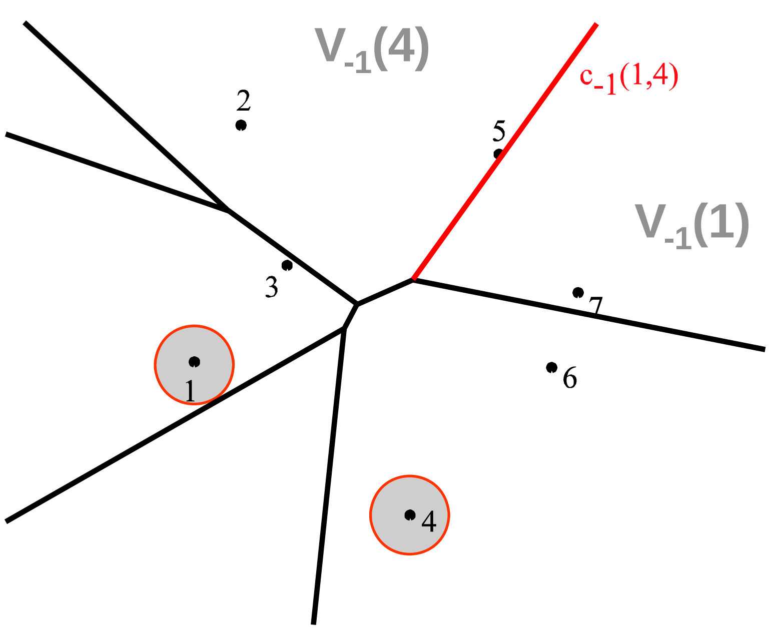


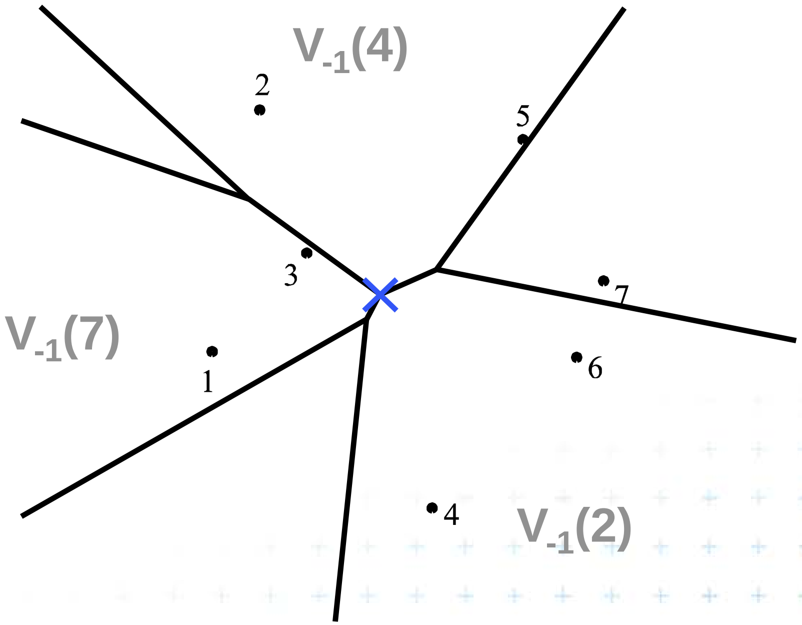


edge : set of points equidistant from 2 sites and closer to all the other sites

[Nandy]

Farthest point Voronoi edges and vertices

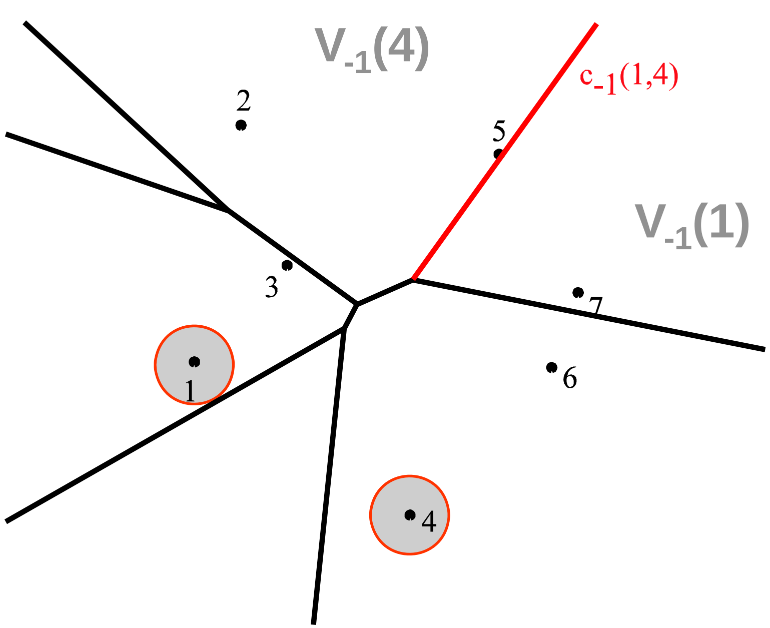


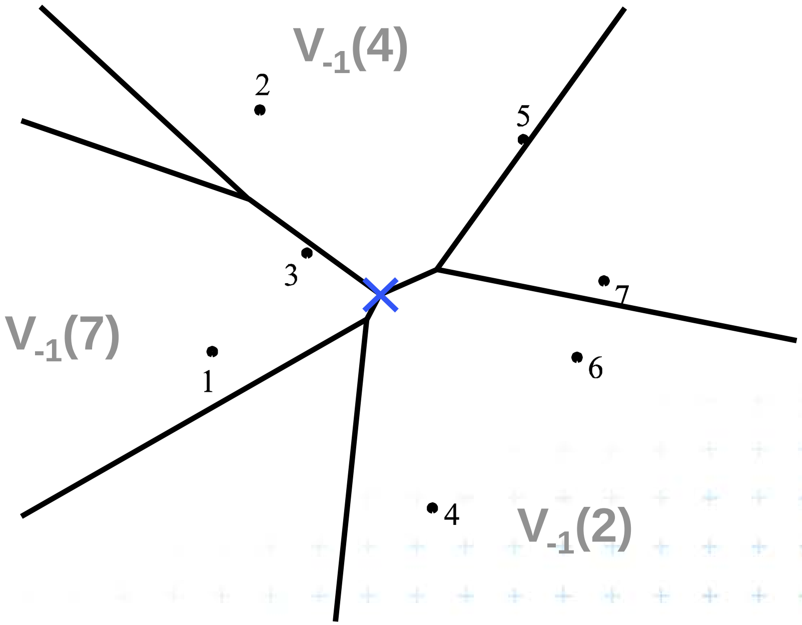


edge : set of points equidistant from 2 sites and closer to all the other sites

[Nandy]

Farthest point Voronoi edges and vertices



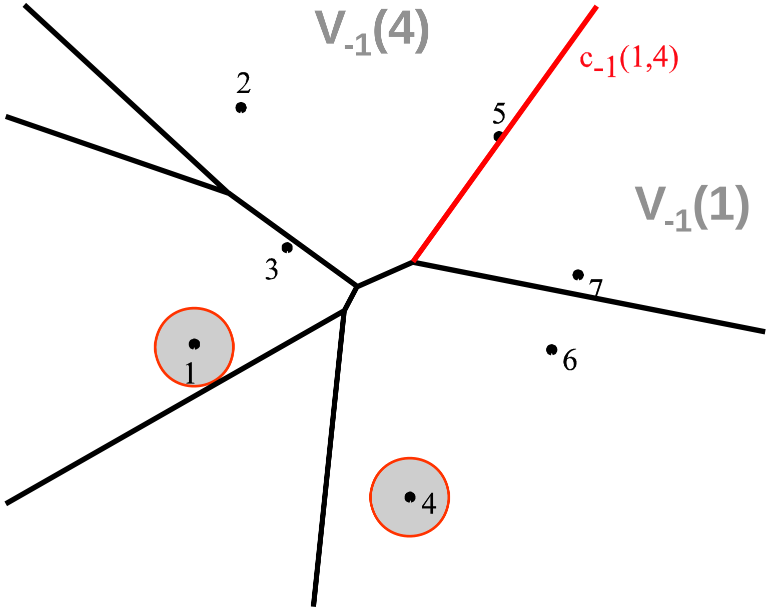


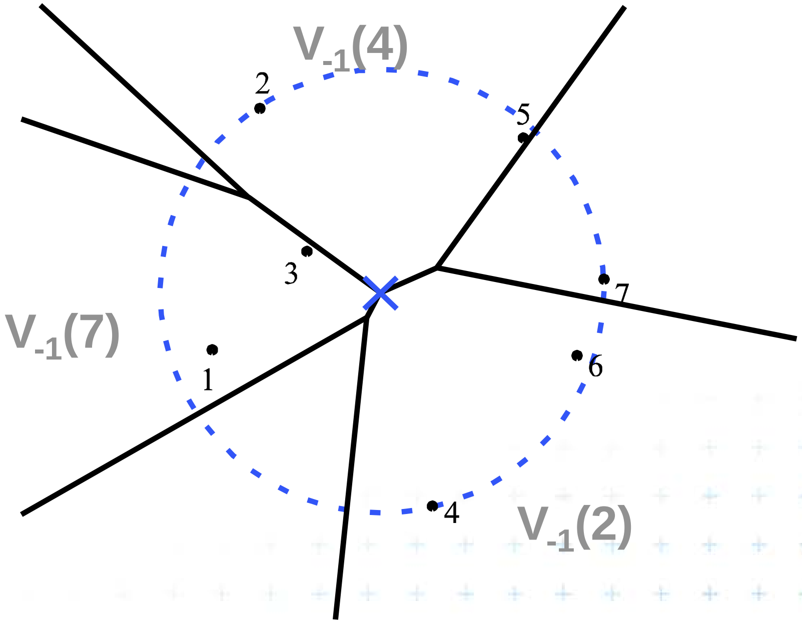
edge : set of points equidistant from 2 sites and closer to all the other sites

vertex : point equidistant from at least 3 sites and closer to all the other sites

[Nandy]

Farthest point Voronoi edges and vertices



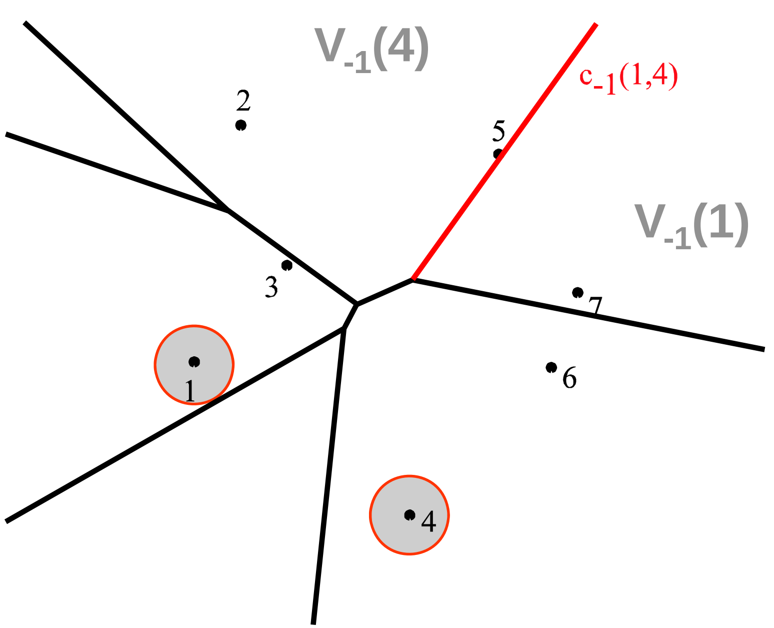


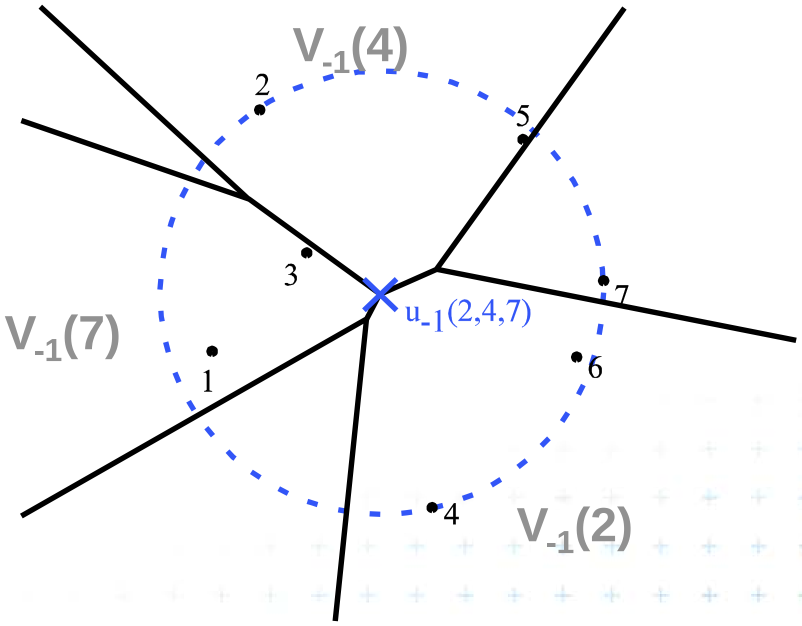
edge : set of points equidistant from 2 sites and closer to all the other sites

vertex : point equidistant from at least 3 sites and closer to all the other sites

[Nandy]

Farthest point Voronoi edges and vertices





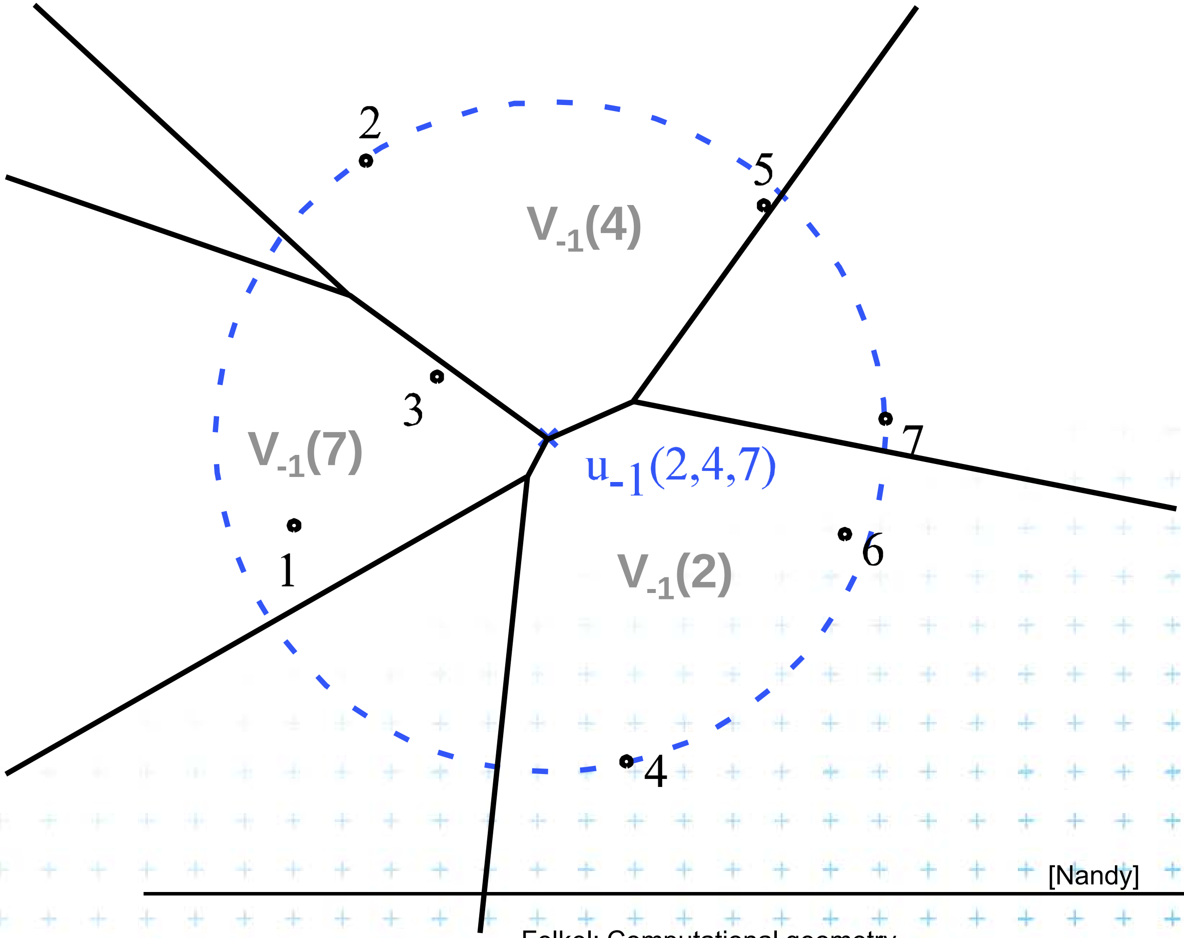
edge : set of points equidistant from 2 sites and closer to all the other sites

vertex : point equidistant from at least 3 sites and closer to all the other sites

[Nandy]

Application of $\mathrm{Vor}_{-1}(P)$: Smallest enclosing circle

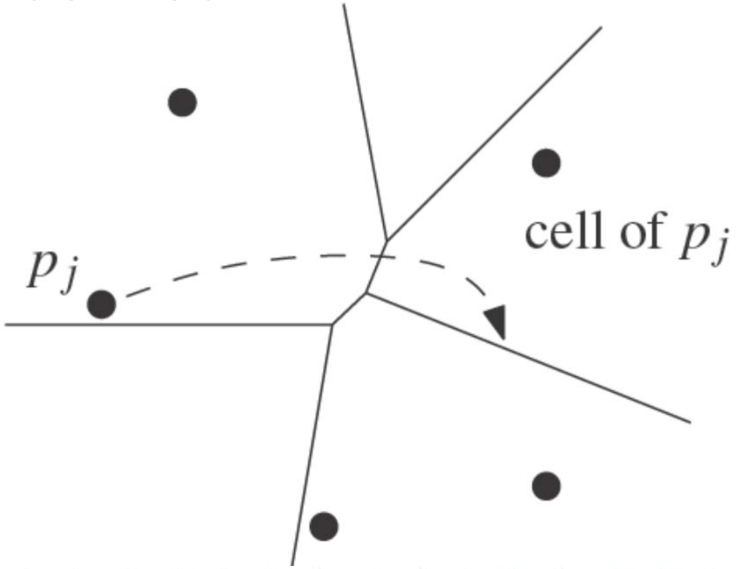
- Construct $\mathrm{Vor}_{-1}(P)$ and find minimal circle with center in $\mathrm{Vor}_{-1}(P)$ vertices or on edges



[Nandy]

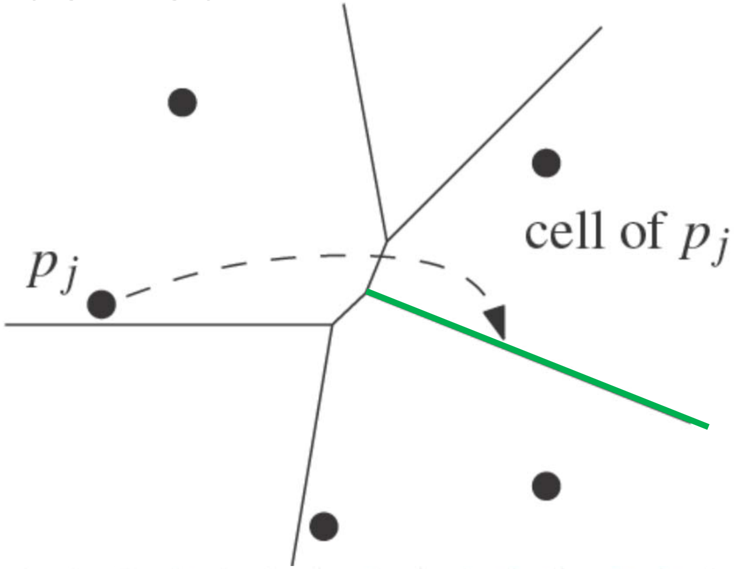
Modified DCEL for farthest-point Voronoi d

- Half-infinite edges -> we adapt DCEL
- Half-edges with origin in infinity
  - Special vertex-like record for origin in infinity
  - Store direction instead of coordinates
  - Next(e) or Prev(e) pointers undefined
- For each inserted site p_j
  - store a pointer to the most CCW half-infinite half-edge of its cell in DCEL



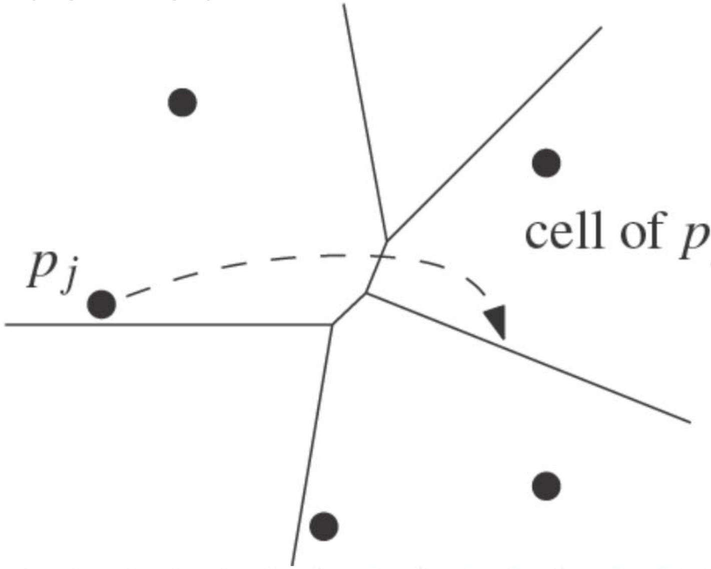
Modified DCEL for farthest-point Voronoi d

- Half-infinite edges -> we adapt DCEL
- Half-edges with origin in infinity
  - Special vertex-like record for origin in infinity
  - Store direction instead of coordinates
  - Next(e) or Prev(e) pointers undefined
- For each inserted site p_j
  - store a pointer to the most CCW half-infinite half-edge of its cell in DCEL



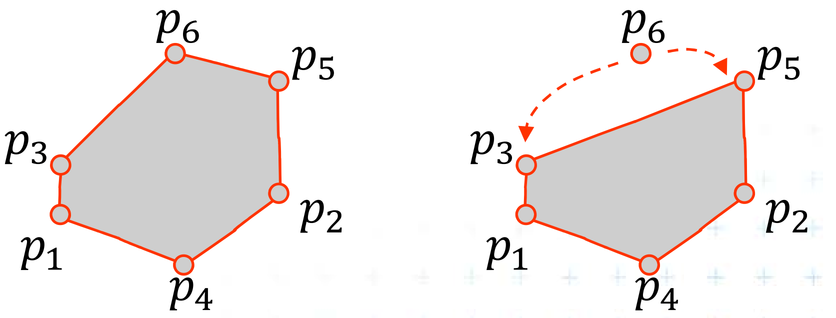
Modified DCEL for farthest-point Voronoi d

- Half-infinite edges -> we adapt DCEL
- Half-edges with origin in infinity
  - Special vertex-like record for origin in infinity
  - Store direction instead of coordinates
  - Next(e) or Prev(e) pointers undefined
- For each inserted site p_j
  - store a pointer to the most CCW half-infinite half-edge of its cell in DCEL



Idea of the algorithm

1. Create the convex hull
   and number the CH points randomly
2. Remove the points starting in the last of this random order and store *cw*(p_i) and *ccw*(p_i) points at the time of removal.
3. Include the points back and compute V_{-1}



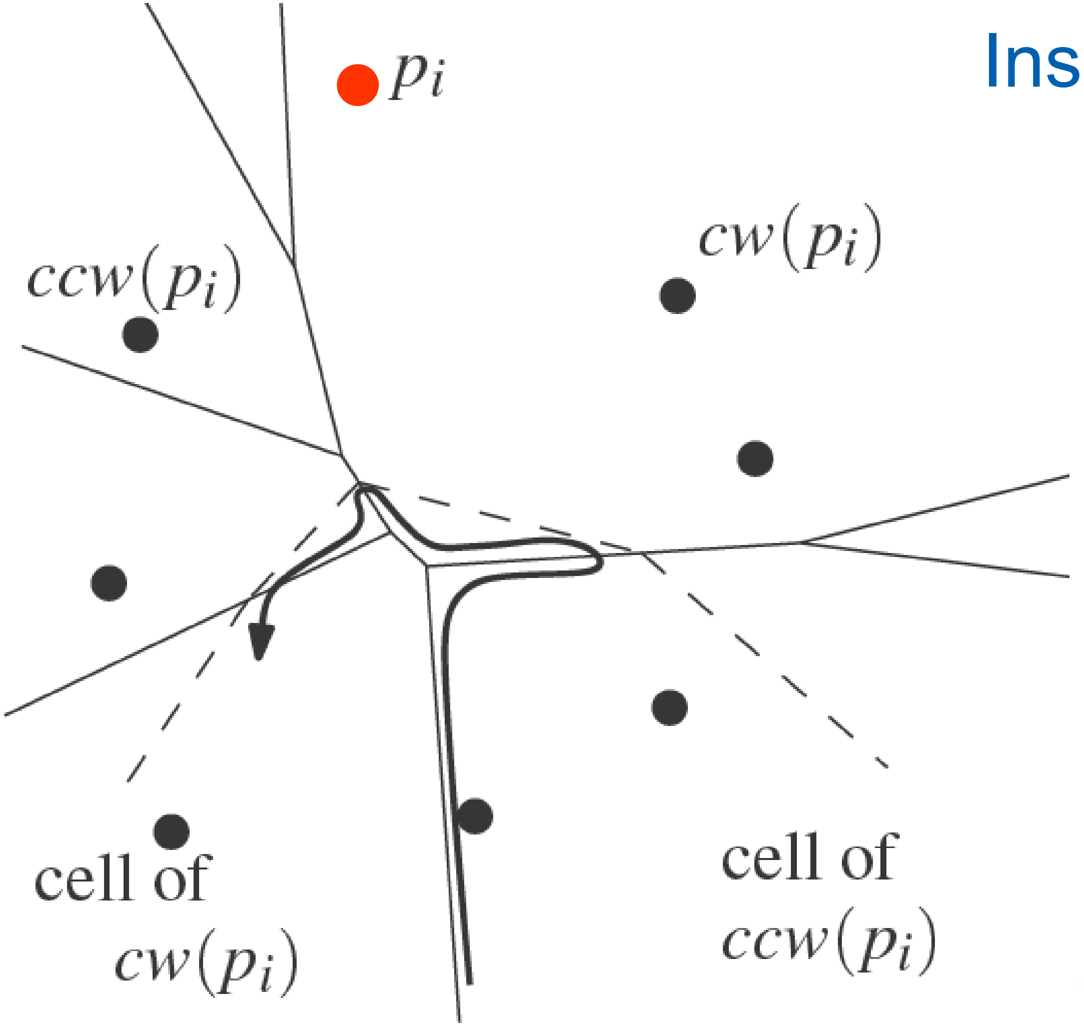
p_i	$ccw(p_i)$	$cw(p_i)$
p_6	p_3	p_5
p_5	p_3	p_2
…		

Farthest-point Voronoi d. construction

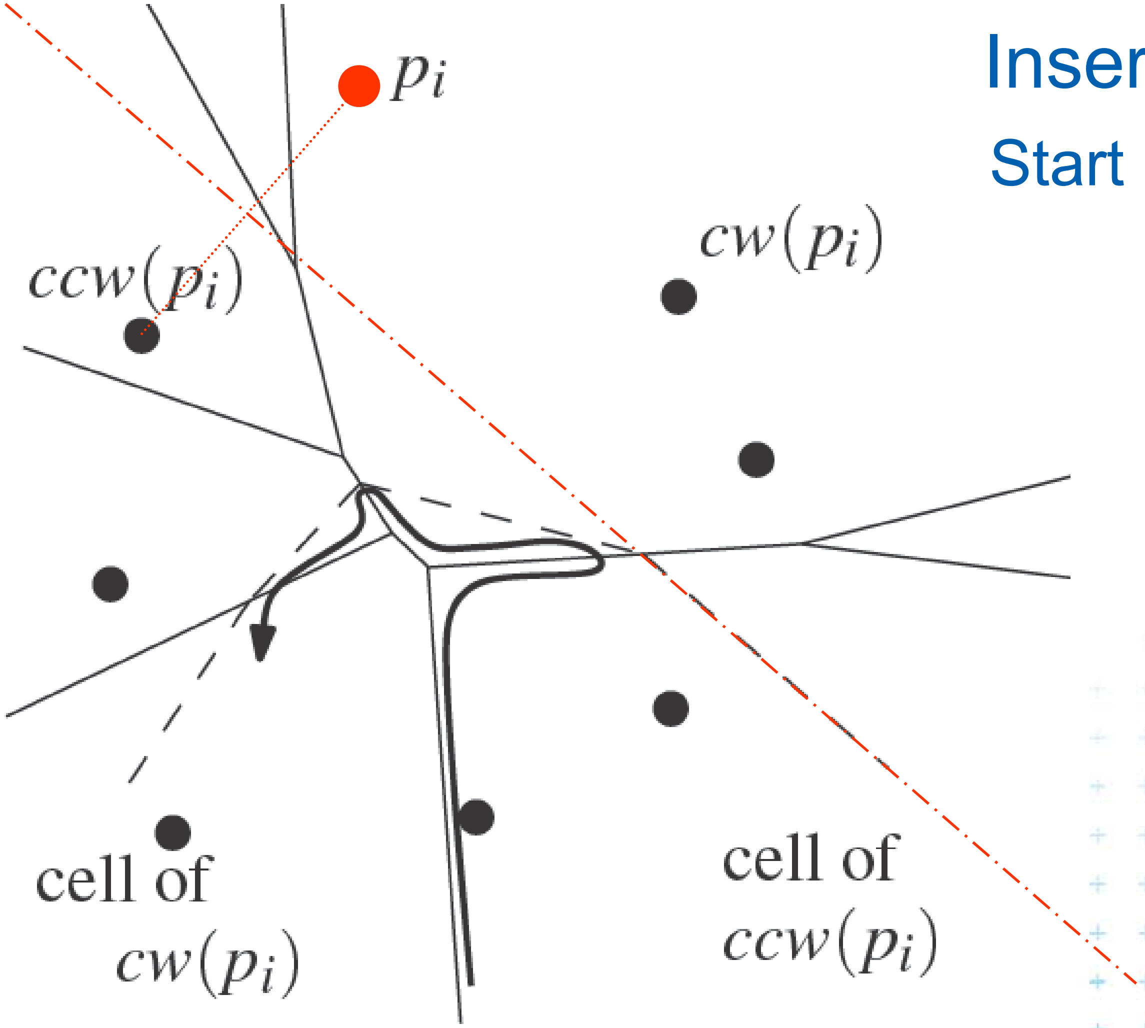
```
Farthest-pointVoronoi                    O(nlog n) time in O(n) storage
Input:   Set of points $P$ in plane
Output:  Farthest-point VD $Vor_{-1}(P)$
1.  Compute convex hull of $P$
2.  Put points in CH($P$) of $P$ in random order $p_1,\dots,p_h$
3.  Remove $p_h, \dots ,p_4$ from the cyclic order (around the CH).
    When removing $p_i$, store the neighbors: $cw(p_i)$ and $ccw(p_i)$ at the time of
    removal. (This is done to know the neighbors needed in step 6.)
4.  Compute $Vor_{-1}( \{ p_1, p_2, p_3 \} )$ as init
5.  for  i = 4  to $h$ do
6.      Add site $p_i$ to $Vor_{-1}(\{ p_1, p_2,\dots, p_{i-1} \})$ between site $cw(p_i)$ and $ccw(p_i)$
7.          - start at most CCW edge of the cell $ccw(p_i)$
8.          - continue CW to find intersection with bisector( $ccw(p_i), p_i$ )
9.          - trace borders of Voronoi cell $p_i$ in CCW order, add edges
10.         - remove invalid edges inside of Voronoi cell $p_i$
```

Farthest-point Voronoi d. construction

Insertion of site p_i



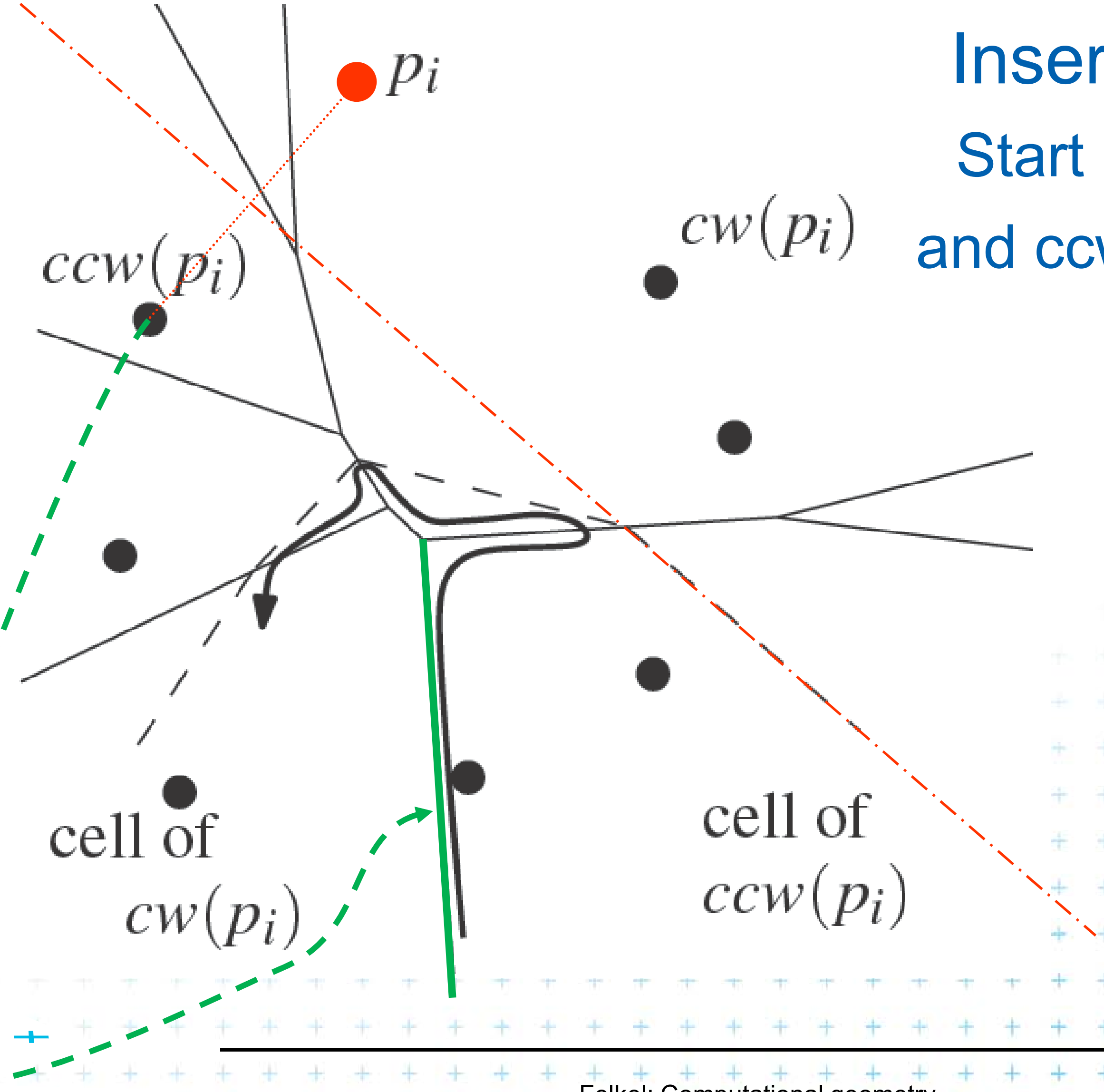
Farthest-point Voronoi d. construction



Insertion of site p_i

Start with site ccw(p_i)

Farthest-point Voronoi d. construction

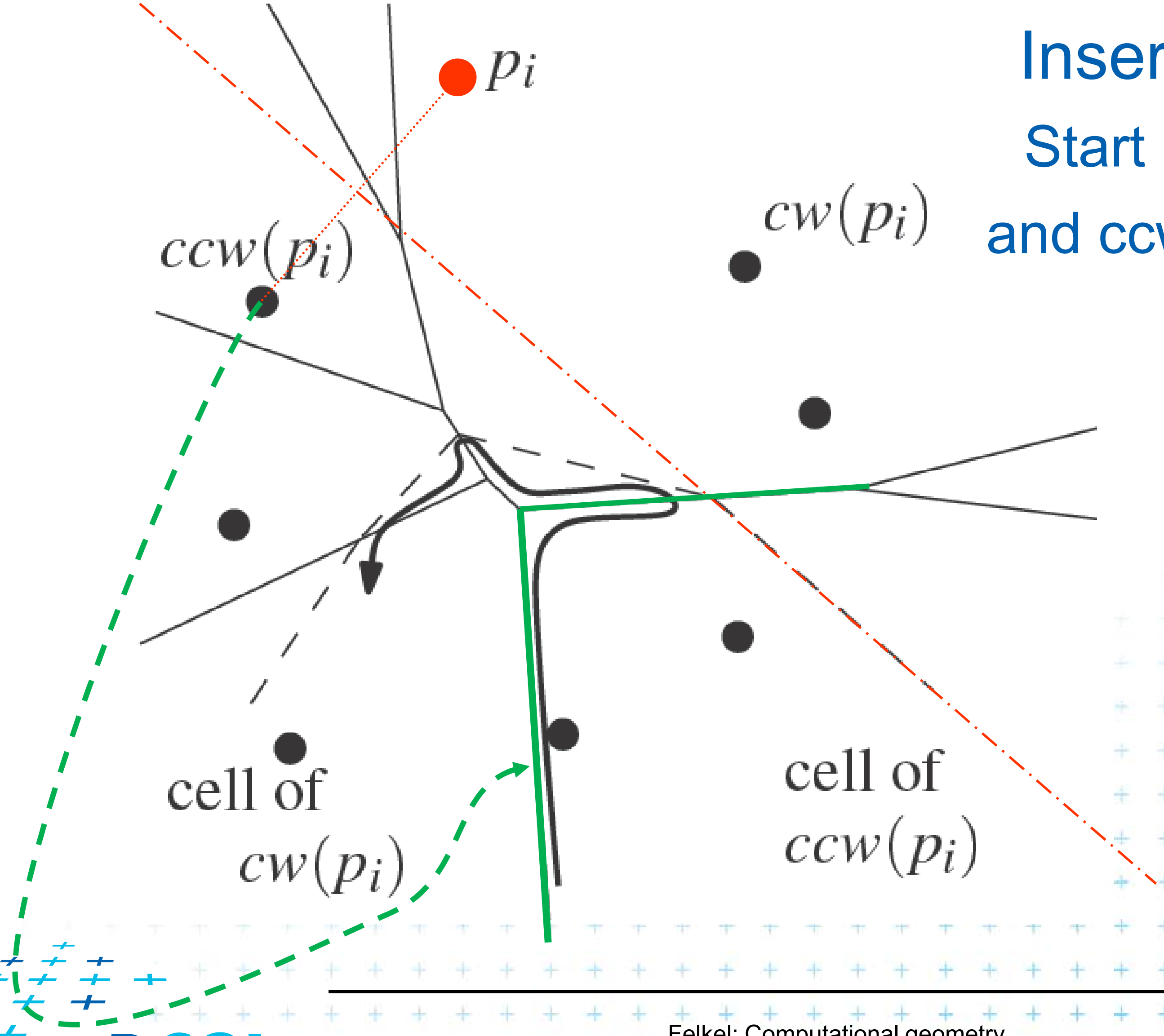


Insertion of site p_i

Start with site ccw(p_i)

and ccw edge of its cell

Farthest-point Voronoi d. construction

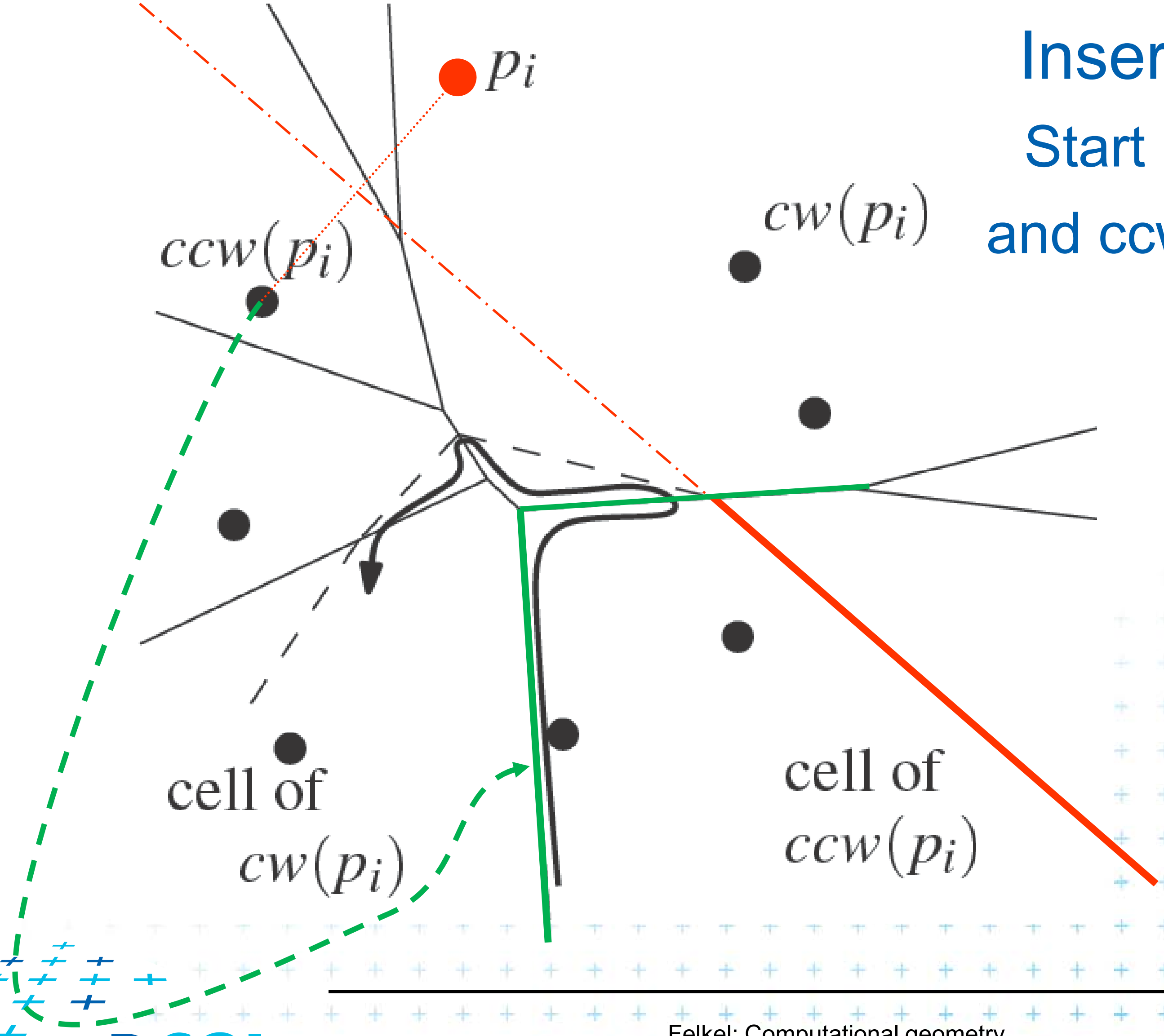


Insertion of site p_i

Start with site ccw(p_i)

and ccw edge of its cell

Farthest-point Voronoi d. construction

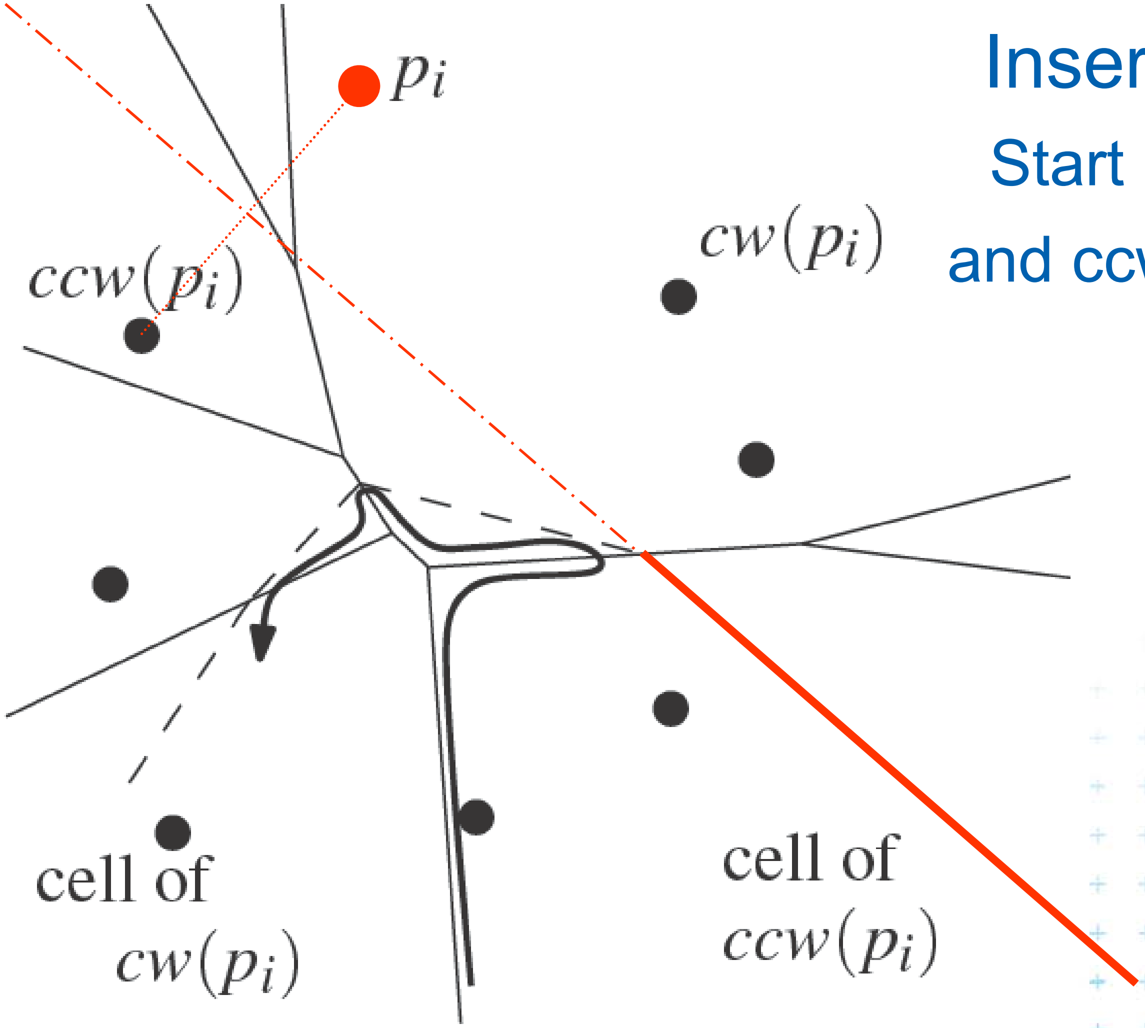


Insertion of site p_i

Start with site ccw(p_i)

and ccw edge of its cell

Farthest-point Voronoi d. construction

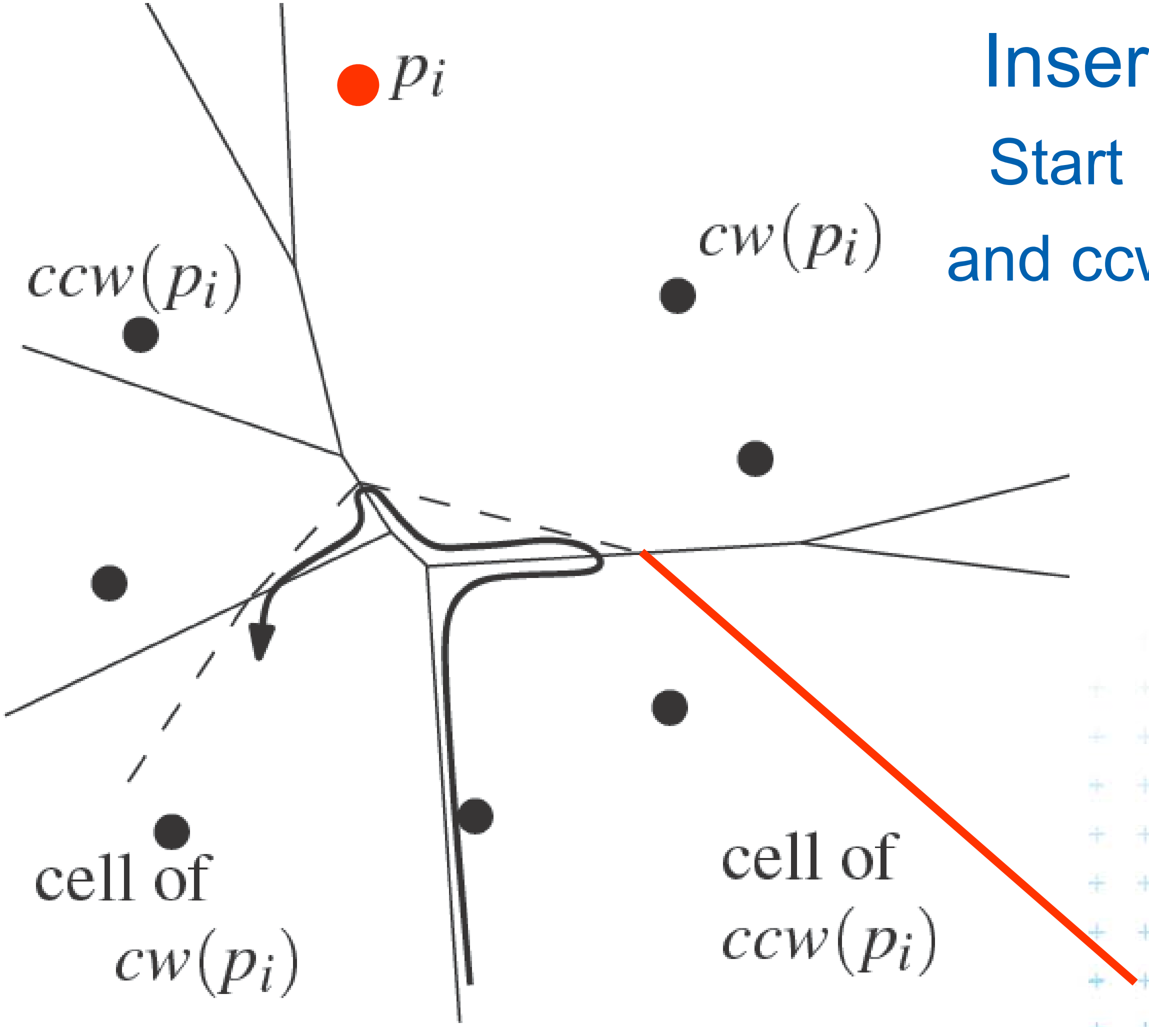


Insertion of site p_i

Start with site ccw(p_i)

and ccw edge of its cell

Farthest-point Voronoi d. construction

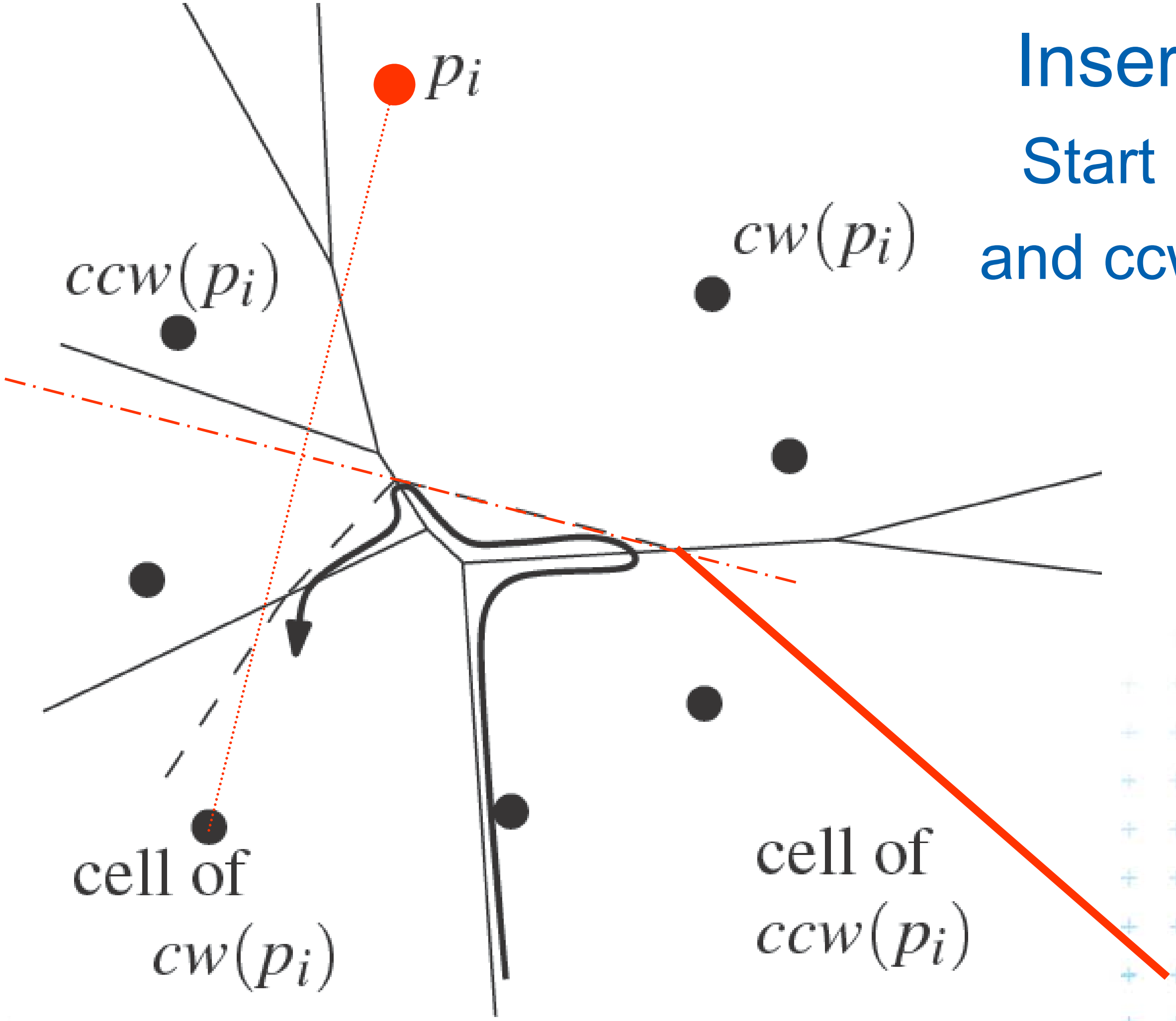


Insertion of site p_i

Start with site ccw(p_i)

and ccw edge of its cell

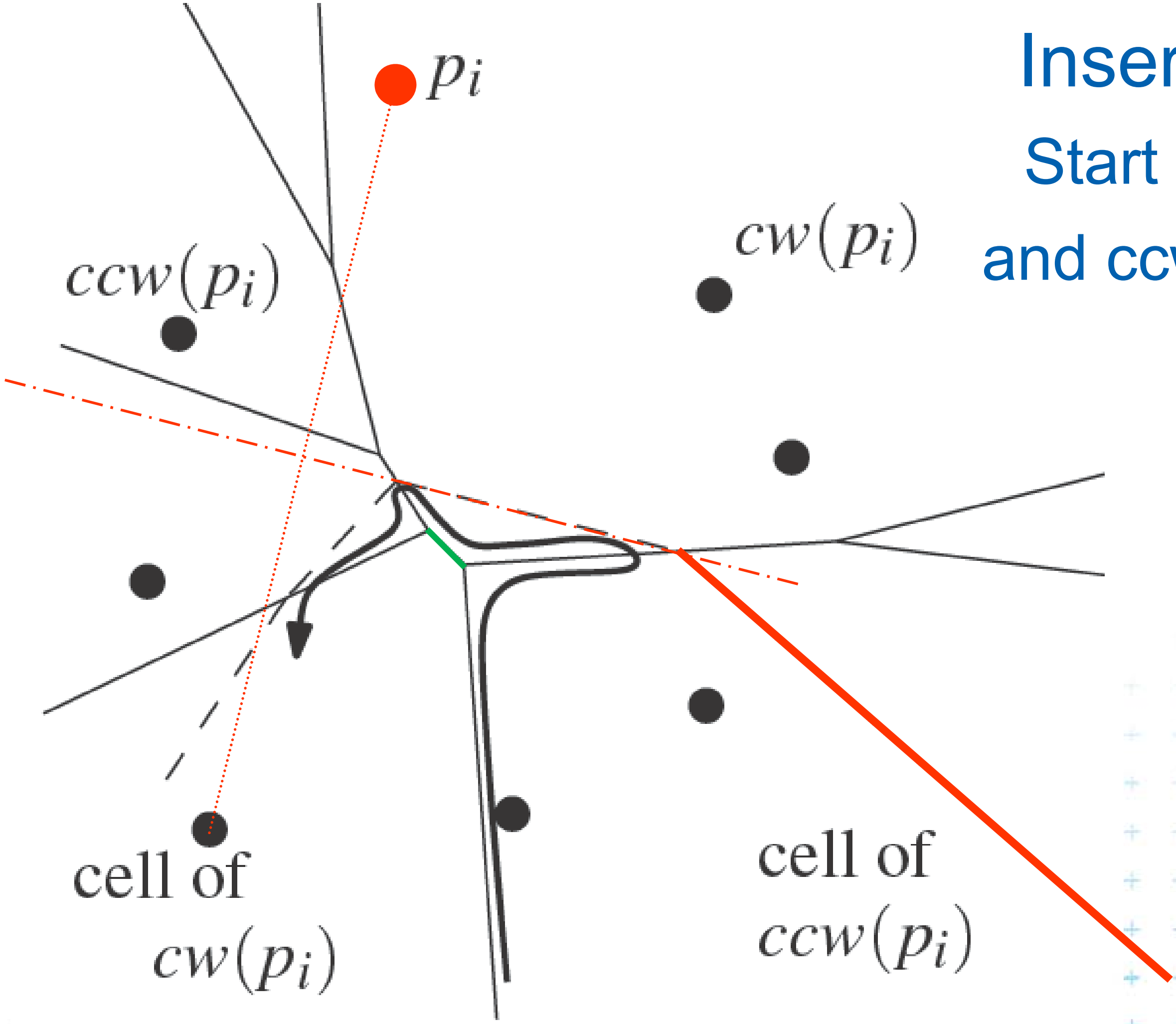
Farthest-point Voronoi d. construction



Insertion of site p_i

Start with site ccw(p_i) and ccw edge of its cell

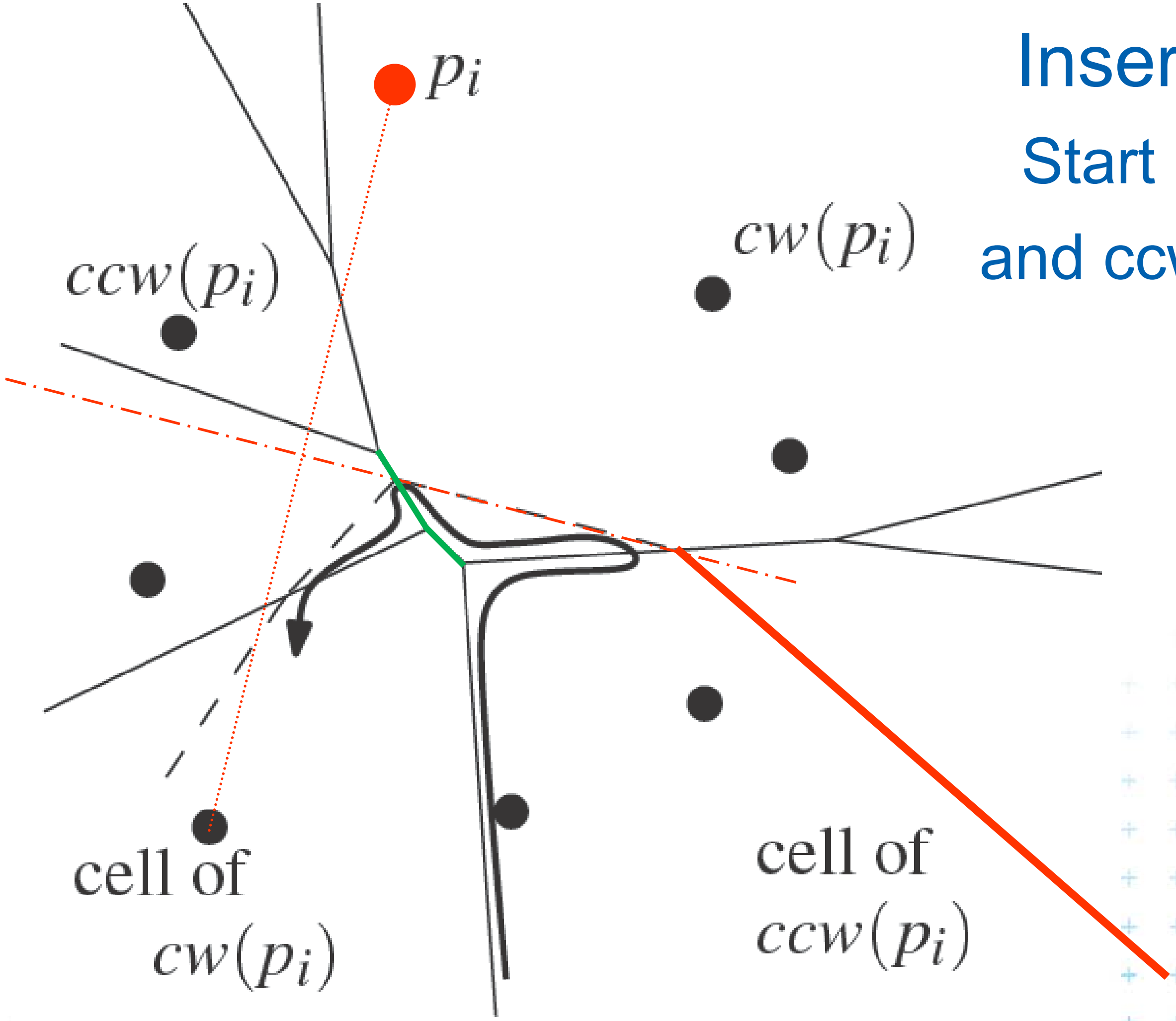
Farthest-point Voronoi d. construction



Insertion of site p_i

Start with site ccw(p_i) and ccw edge of its cell

Farthest-point Voronoi d. construction

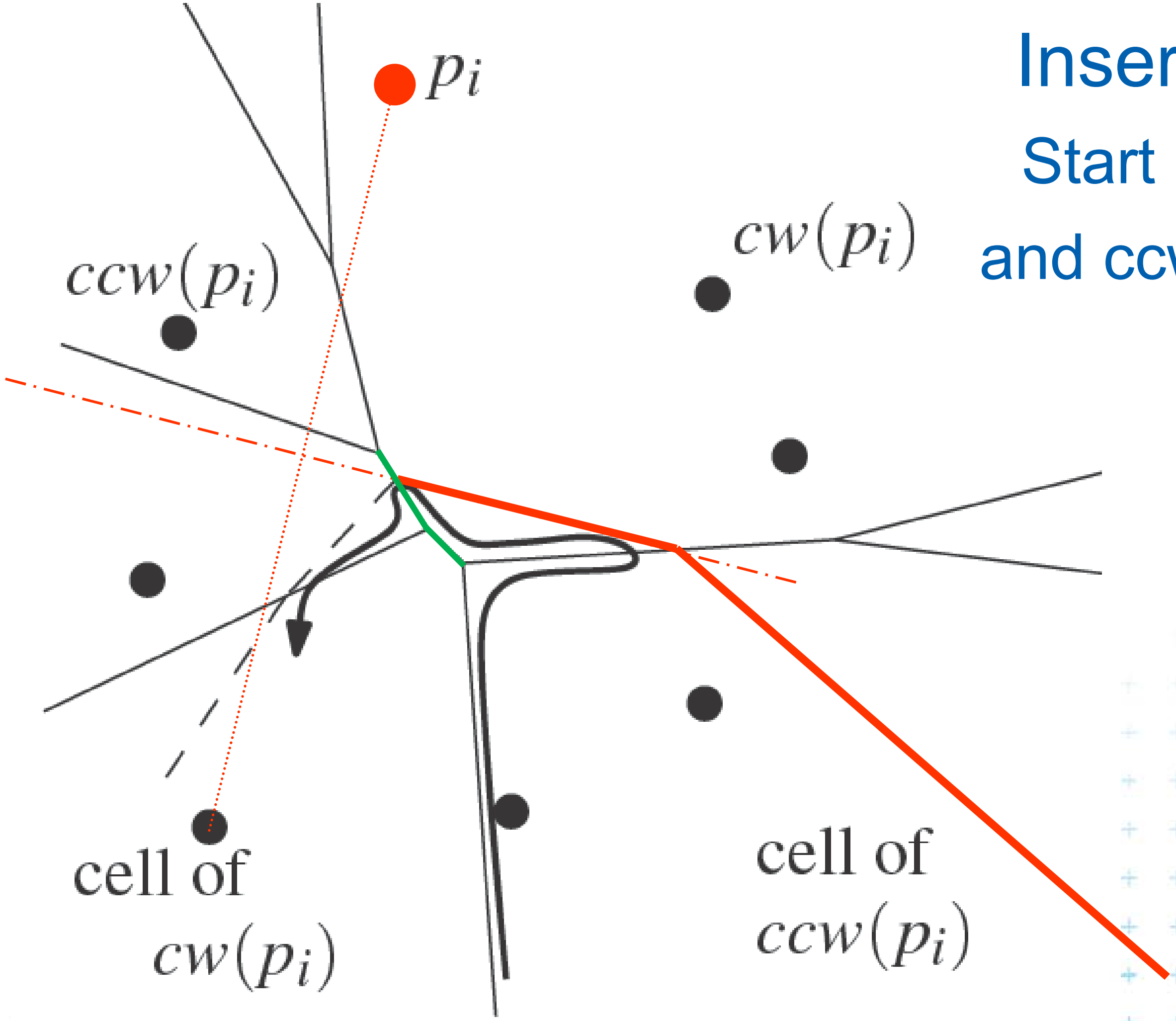


Insertion of site p_i

Start with site ccw(p_i)

and ccw edge of its cell

Farthest-point Voronoi d. construction

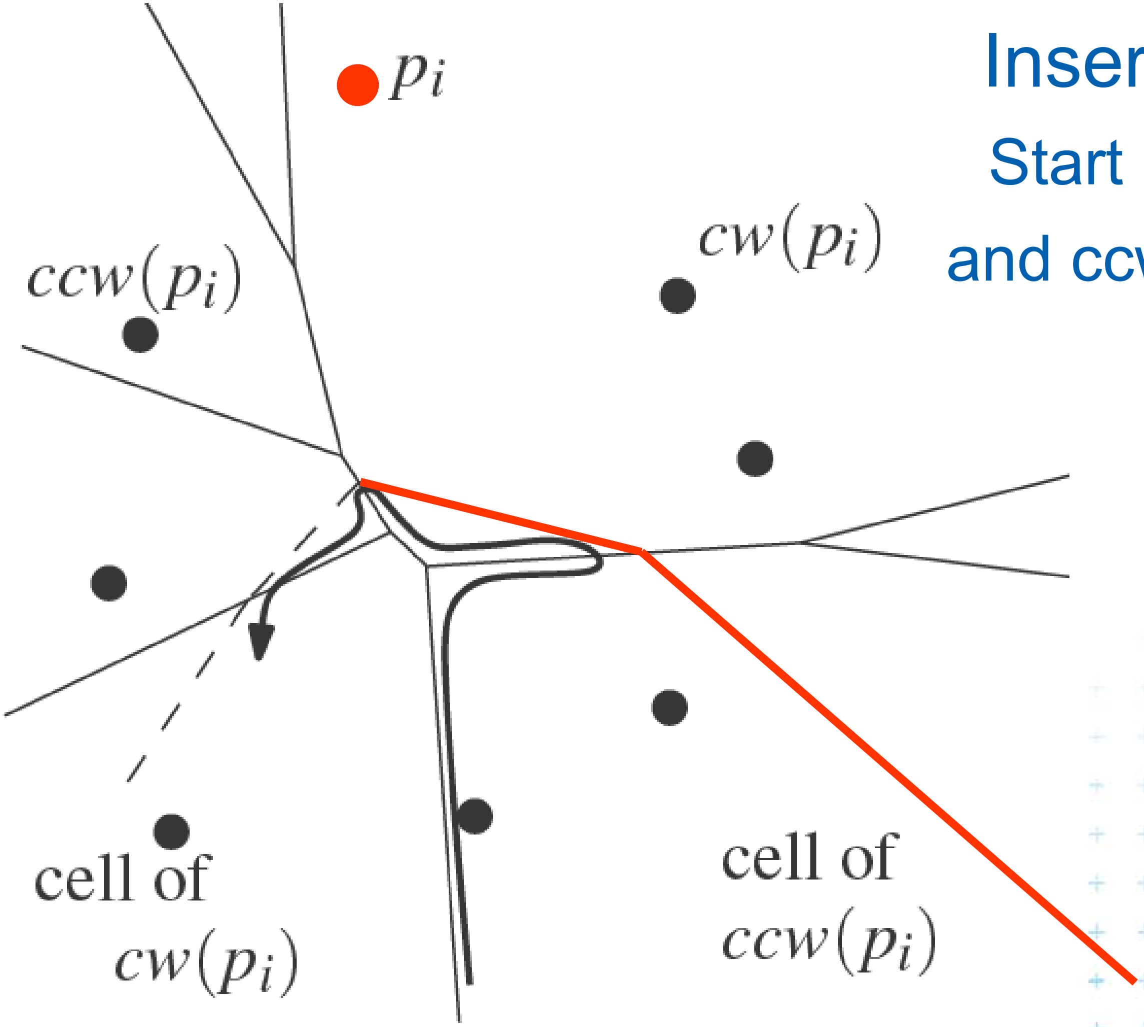


Insertion of site p_i

Start with site ccw(p_i)

and ccw edge of its cell

Farthest-point Voronoi d. construction

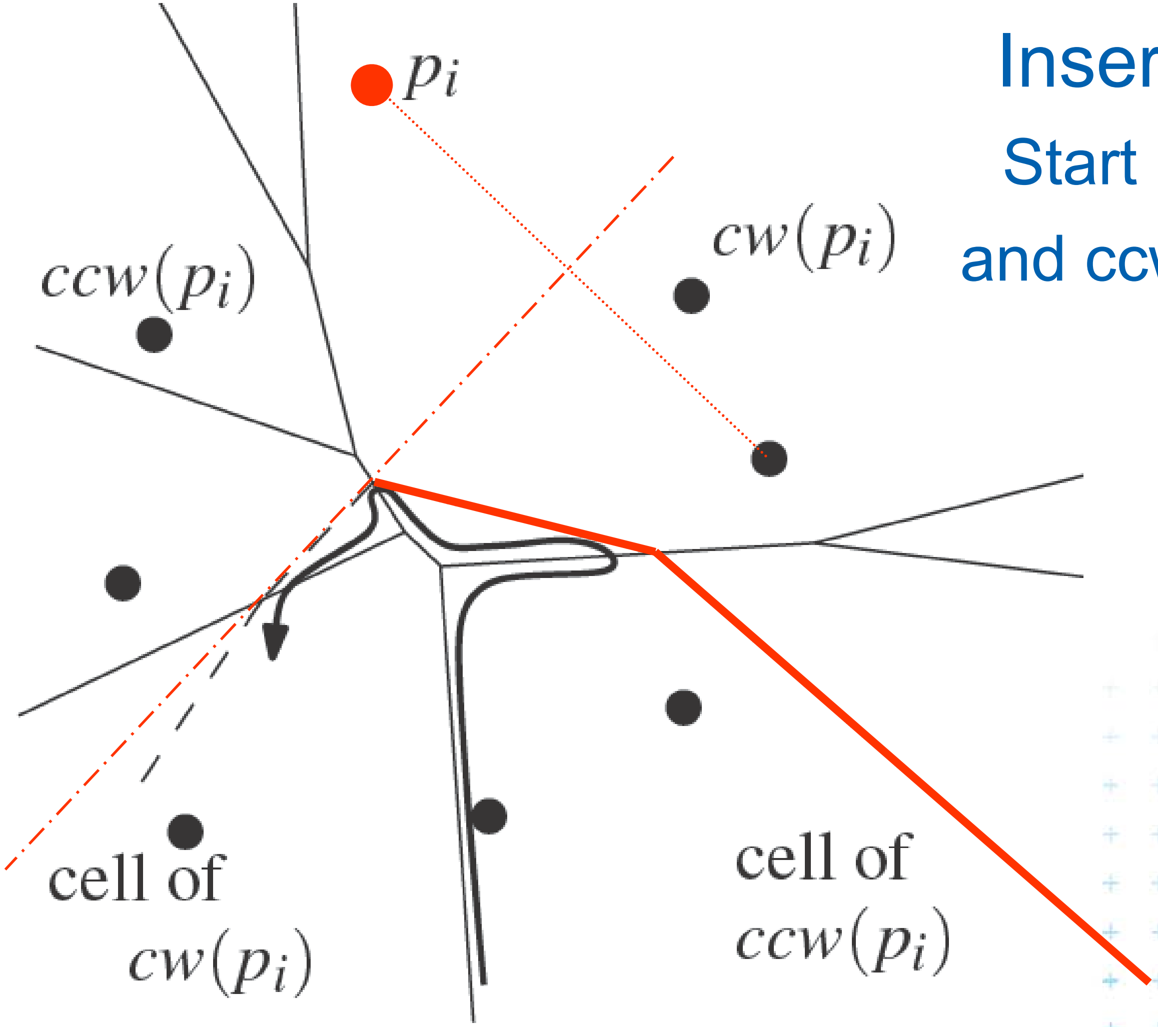


Insertion of site p_i

Start with site ccw(p_i)

and ccw edge of its cell

Farthest-point Voronoi d. construction

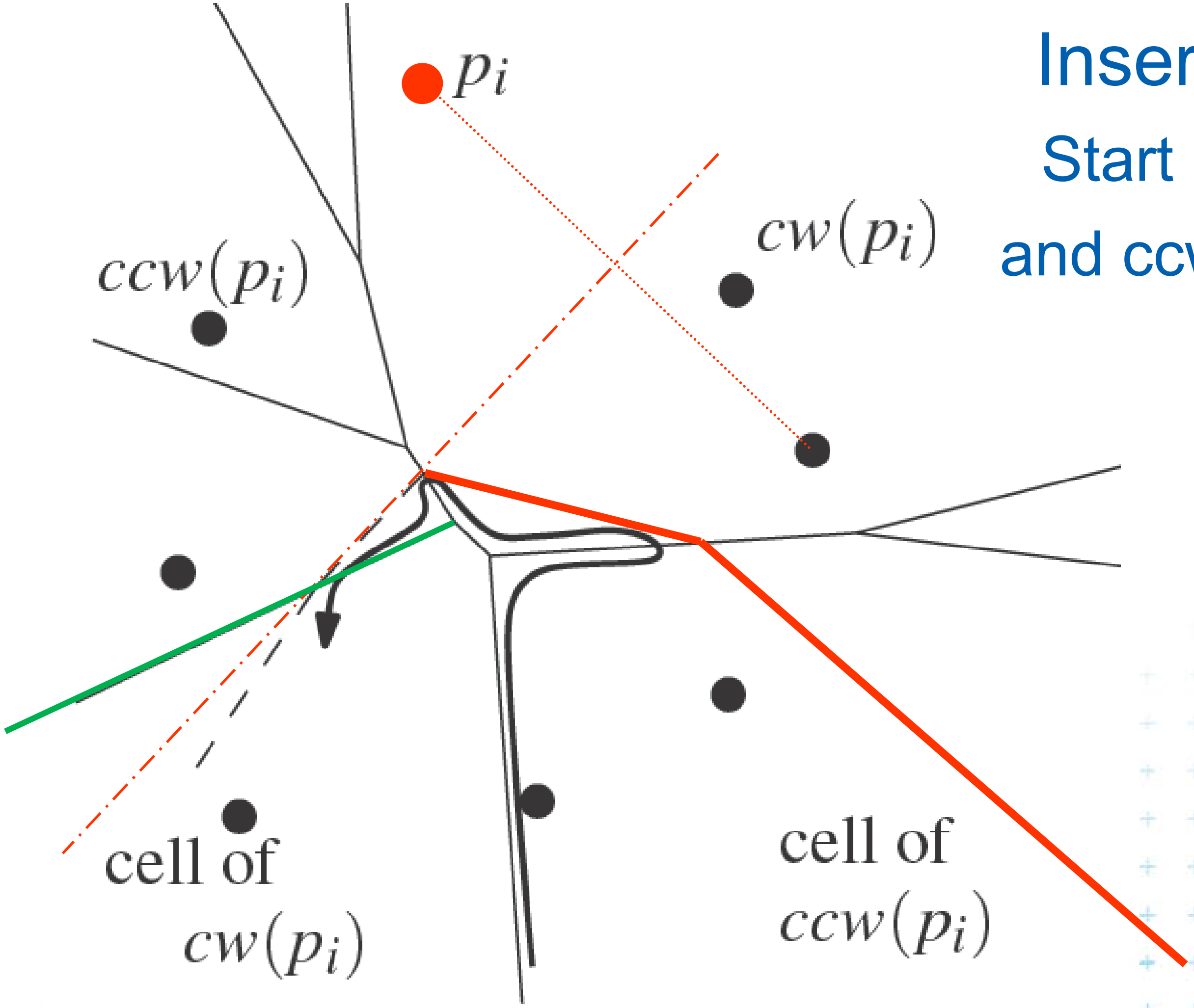


Insertion of site p_i

Start with site ccw(p_i)

and ccw edge of its cell

Farthest-point Voronoi d. construction

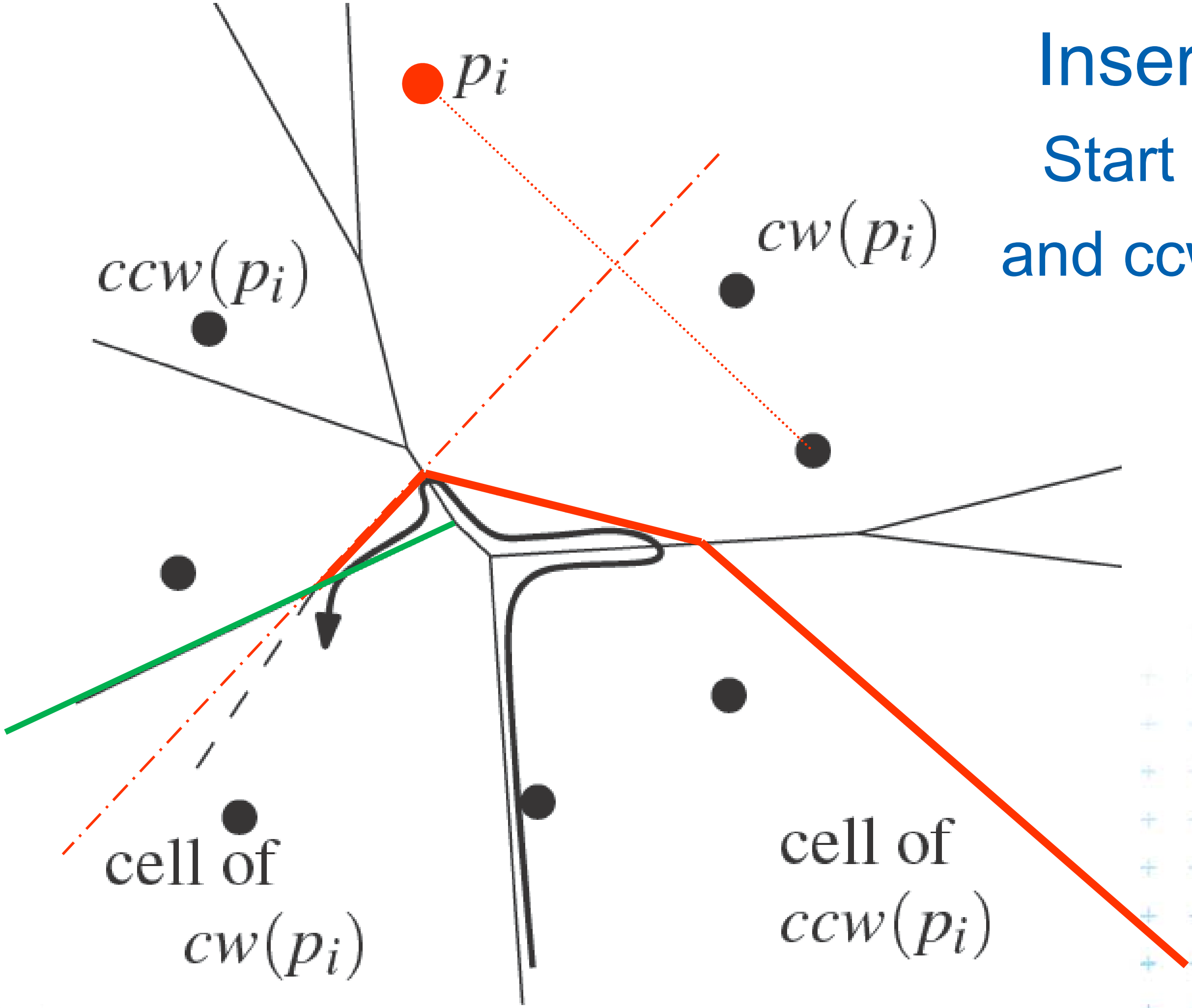


Insertion of site p_i

Start with site ccw(p_i)

and ccw edge of its cell

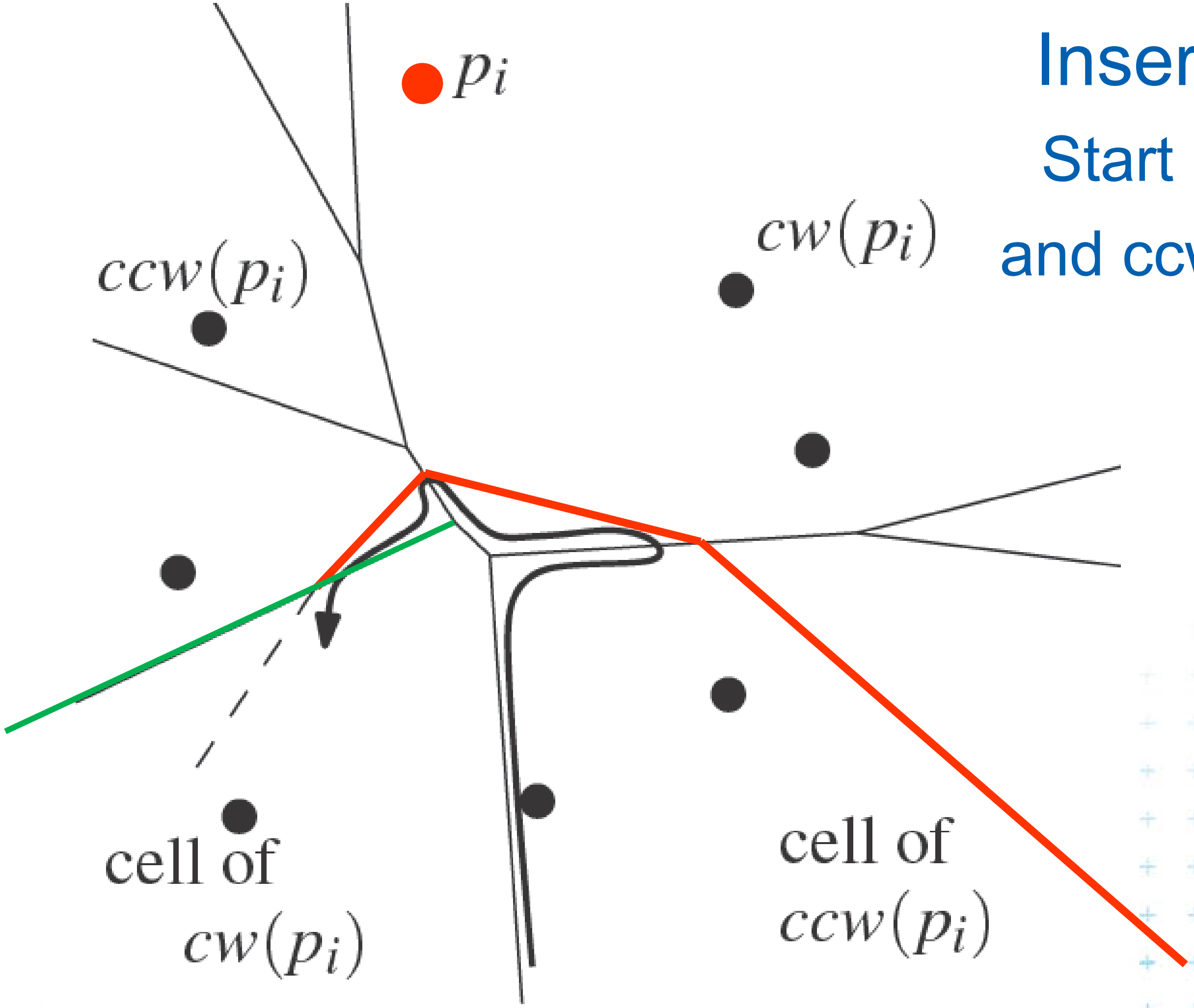
Farthest-point Voronoi d. construction



Insertion of site p_i

Start with site ccw(p_i) and ccw edge of its cell

Farthest-point Voronoi d. construction

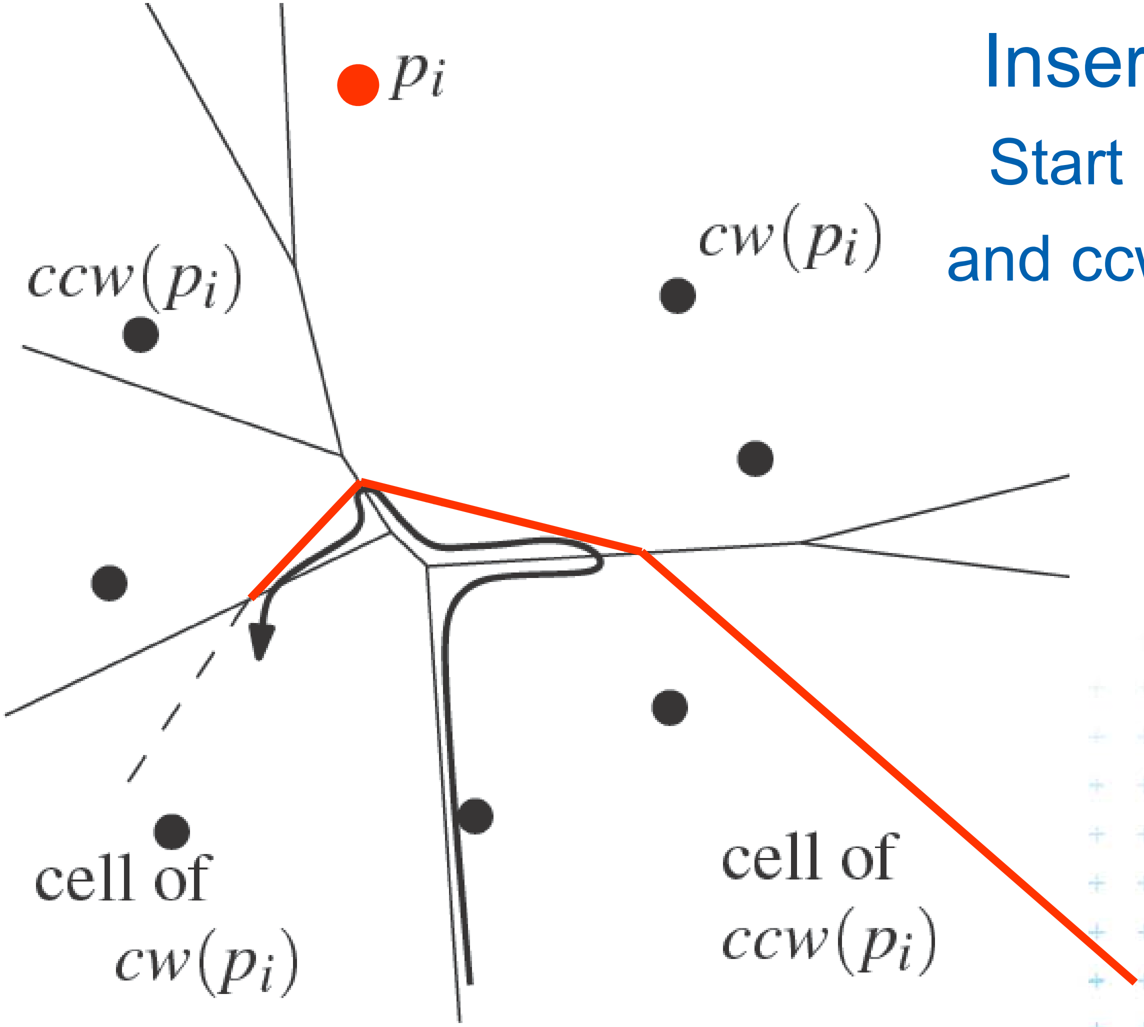


Insertion of site p_i

Start with site ccw(p_i)

and ccw edge of its cell

Farthest-point Voronoi d. construction

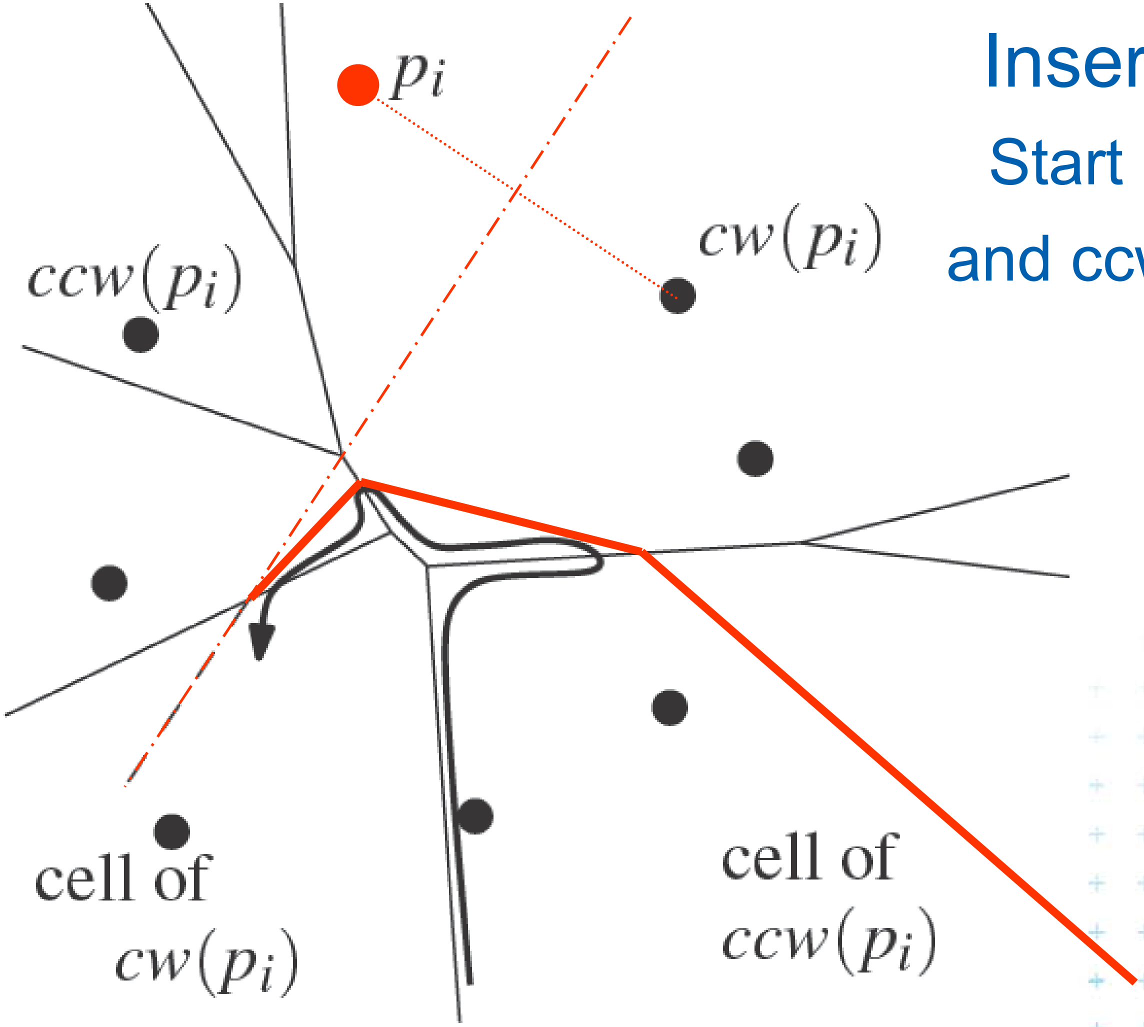


Insertion of site p_i

Start with site ccw(p_i)

and ccw edge of its cell

Farthest-point Voronoi d. construction

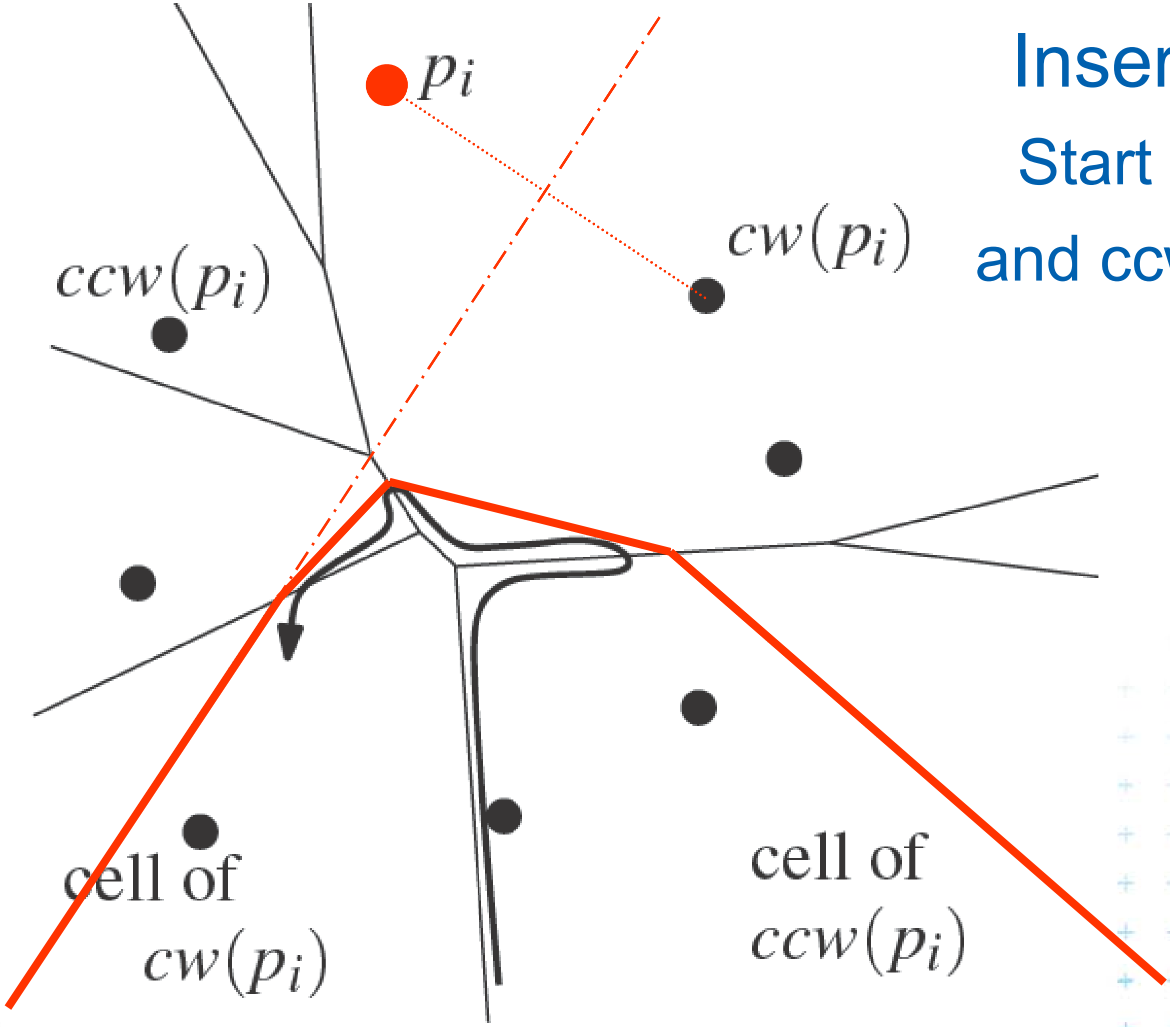


Insertion of site p_i

Start with site ccw(p_i)

and ccw edge of its cell

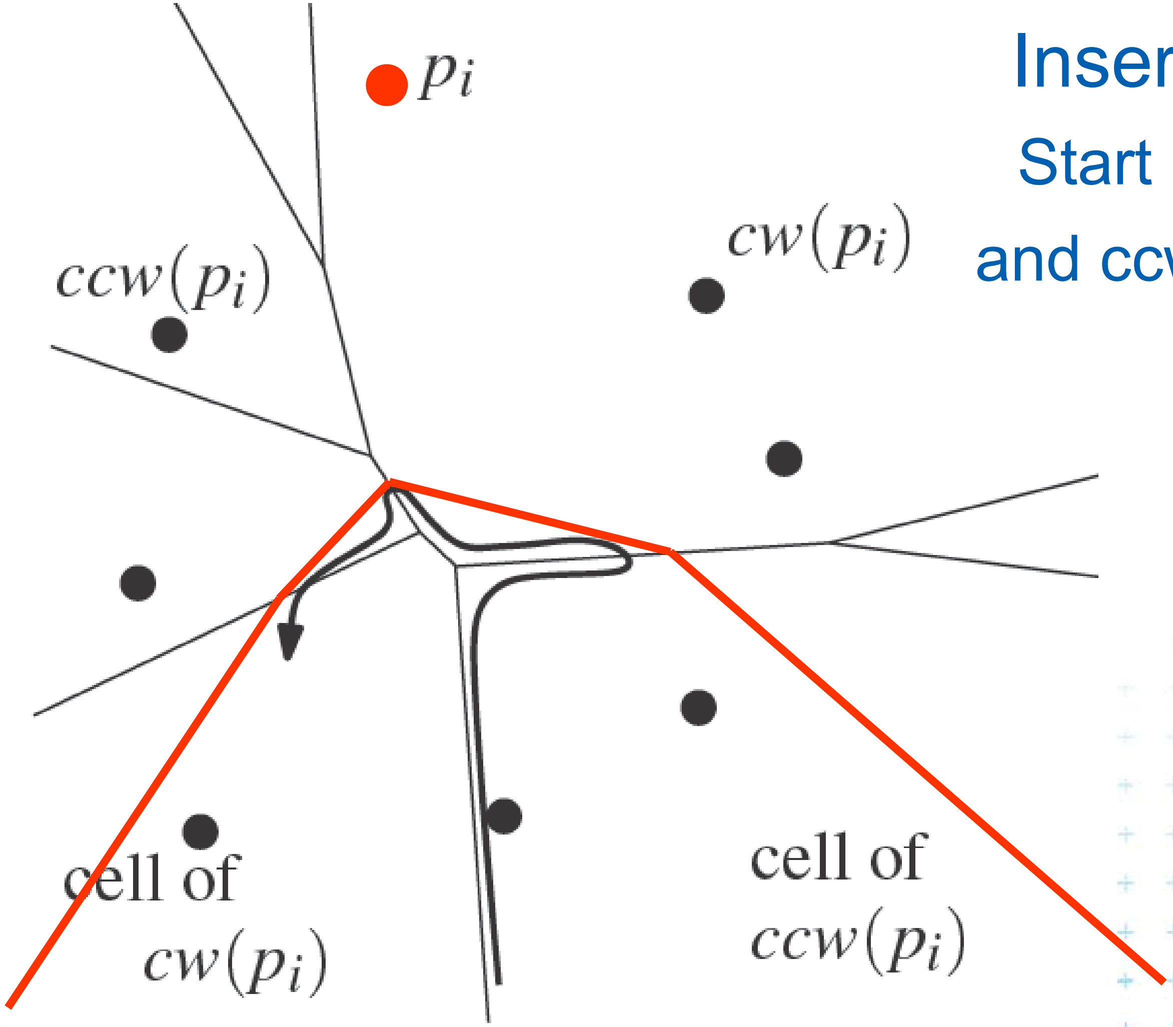
Farthest-point Voronoi d. construction



Insertion of site p_i

Start with site ccw(p_i) and ccw edge of its cell

Farthest-point Voronoi d. construction

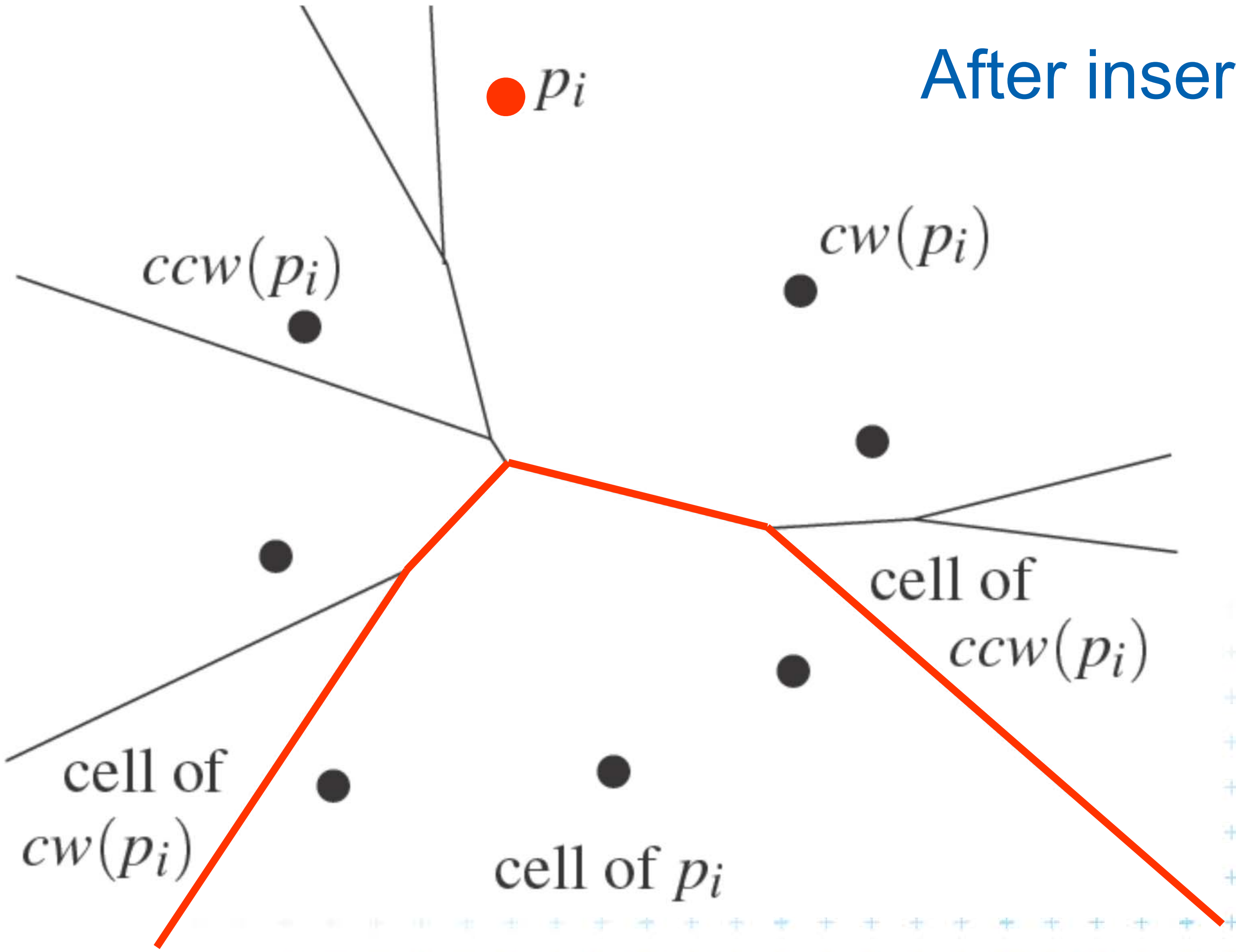


Insertion of site p_i

Start with site ccw(p_i) and ccw edge of its cell

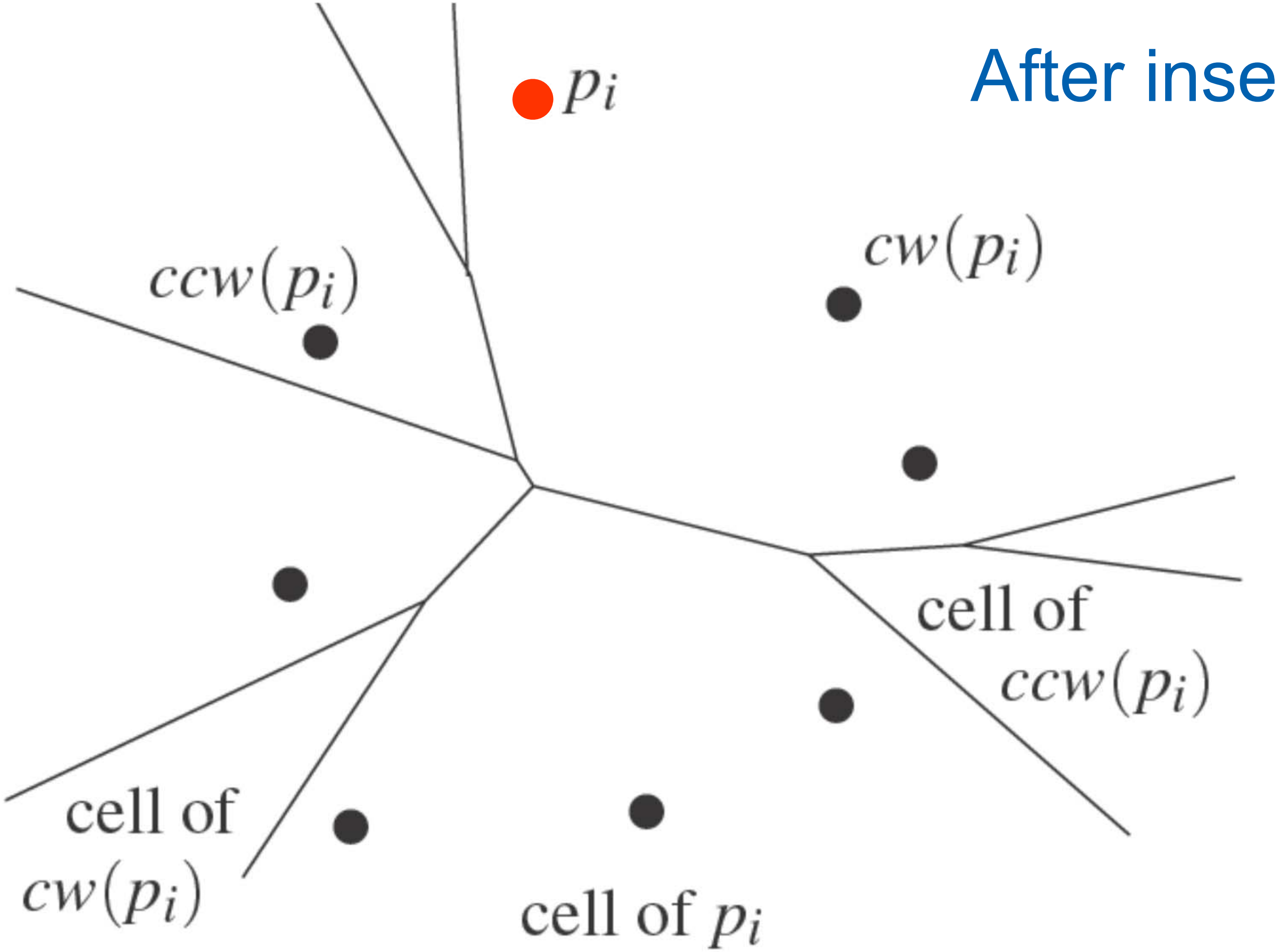
Farthest-point Voronoi d. construction

After insertion of site p_i



Farthest-point Voronoi d. construction

After insertion of site p_i



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, Chapter 7, http://www.cs.uu.nl/geobook/

[Preparata] Preperata, F.P., Shamos, M.I.: *Computational Geometry. An Introduction.* Berlin, Springer-Verlag,1985. Chapters 5 and 6

[Reiberg] Reiberg, J: Implementierung Geometrischer Algoritmen. Berechnung von Voronoi Diagrammen fuer Liniensegmente. http://www.reiberg.net/project/voronoi/avortrag.ps.gz

[Nandy] Subhas C. Nandy: Voronoi Diagram – presentation. Advanced Computing and Microelectronics Unit. Indian Statistical Institute. Kolkata 700108 http://www.tcs.tifr.res.in/~igga/lectureslides/vor-July-08-2009.ppt

[CGAL] http://www.cgal.org/Manual/3.1/doc_html/cgal_manual/Segment_Voronoi_diagram_2/Chapter_main.html

[applets] http://www.personal.kent.edu/~rmuhamma/Compgeometry/MyCG/Voronoi/Fortune/fortune.htm a http://www.liefke.com/hartmut/cis677/

DCGI

DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

TRIANGULATIONS

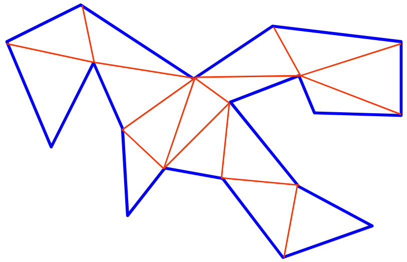
PETR FELKEL

FEL CTU PRAGUE

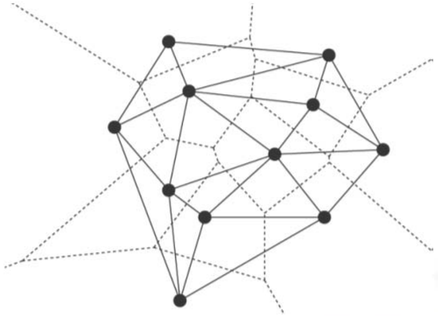
Version from 30.11.2017

Talk overview

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

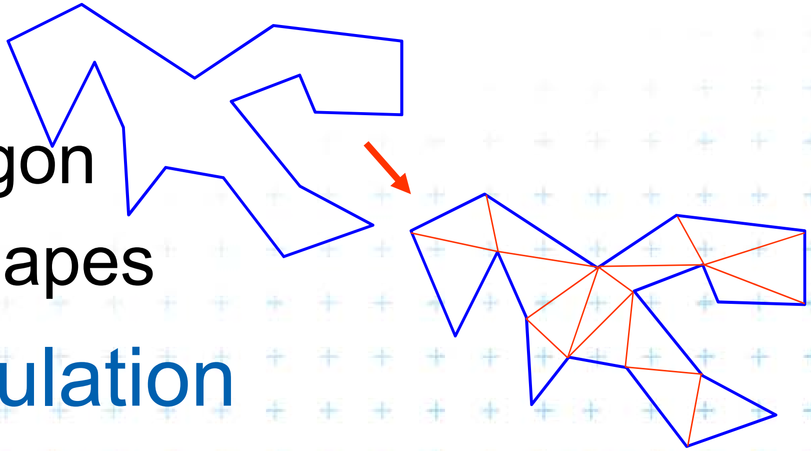


- Delaunay triangulation (DT) of points
  - Input: set of 2D points
  - Properties
  - Incremental Algorithm
  - Relation of DT in 2D and lower envelope (CH) in 3D
    and
    relation of VD in 2D to upper envelope in 3D



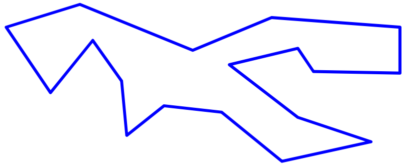
Polygon triangulation problem

- 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



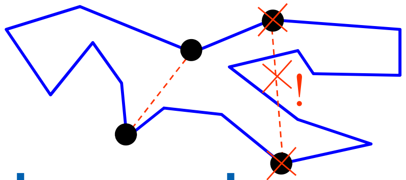
- Complexity of polygon triangulation
  - O(*n*) alg. exists [Chazelle91], but it is too complicated
  - practical algorithms run in O(*n* log *n*)

Terminology



Simple polygon

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



Visible points

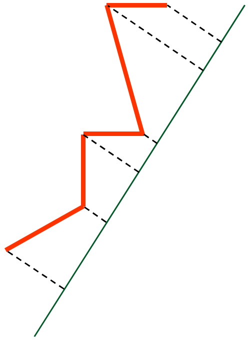
= two points on the boundary are visible if the interior of the line segment joining them lies entirely in the interior of the polygon

Diagonal

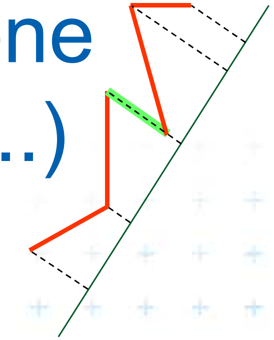
= line segment joining any pair of visible vertices

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*



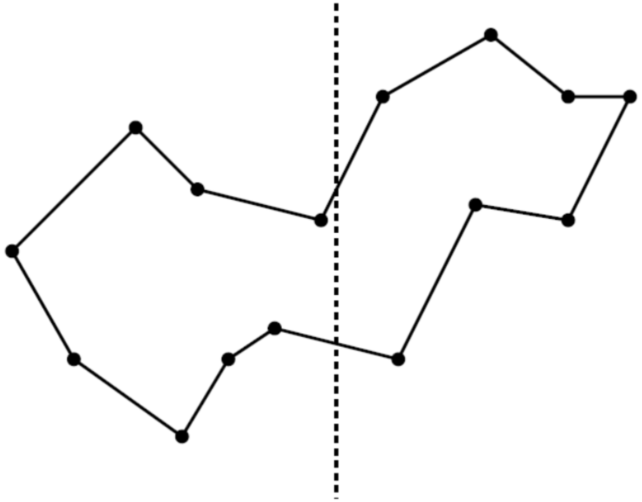
- A chain C is monotone with respect to line L, if any line orthogonal to L intersects C in at most one *connected component* (point, line segment,...)



- 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



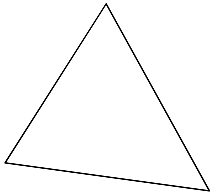
x–monotone polygon

[Mount]

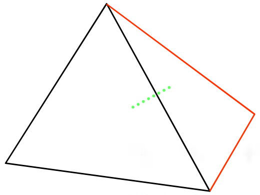
Terminology

- 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

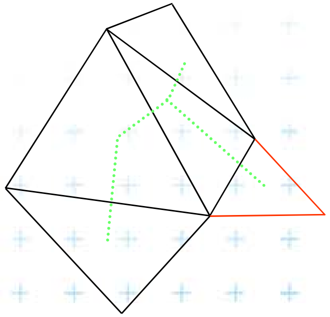
Proof by induction



n = 3 => 0 diagonal



n = 4 => 1 diagonal
n – 3



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)

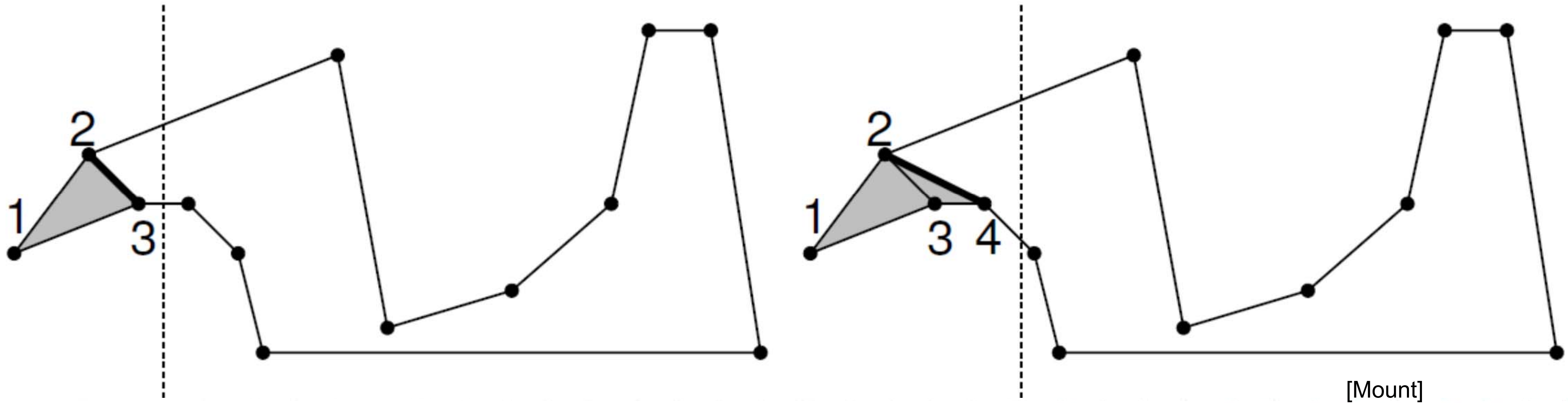
Simple polygon triangulation

- 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

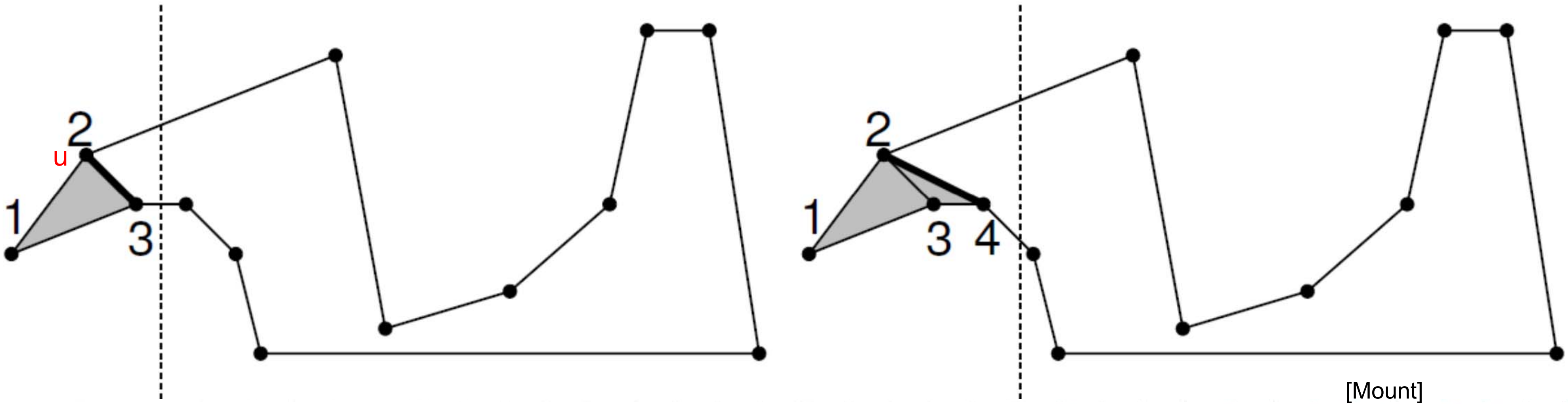
- Sweep left to right - in O(n) time
- Triangulate everything you can by adding diagonals between visible points
- Remove triangulated region from further consideration – mark as DONE



[Mount]

2. Triangulation of the monotone polygon

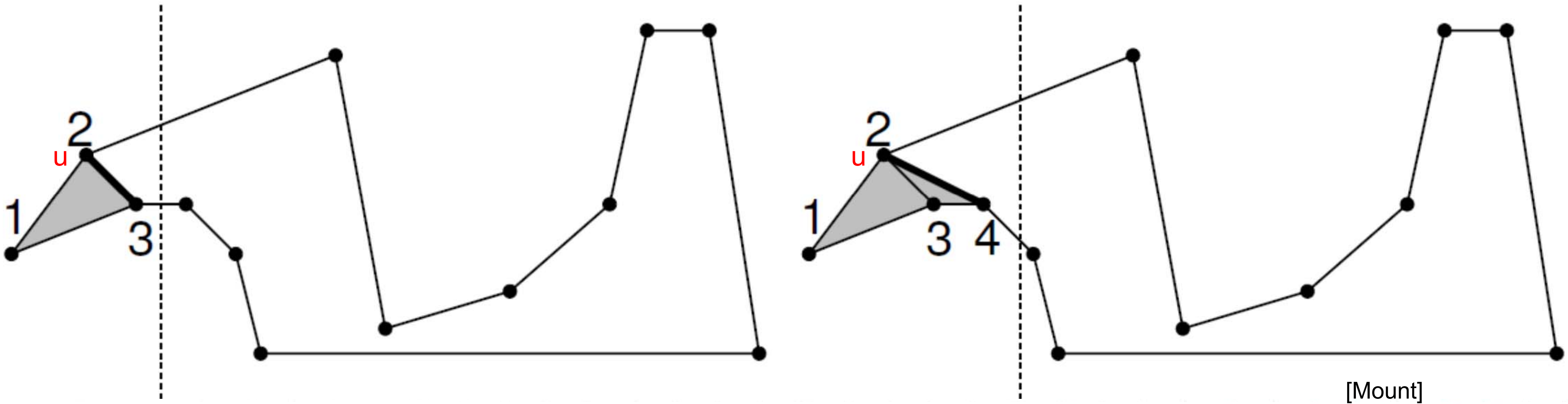
- Sweep left to right - in O(n) time
- Triangulate everything you can by adding diagonals between visible points
- Remove triangulated region from further consideration – mark as DONE



[Mount]

2. Triangulation of the monotone polygon

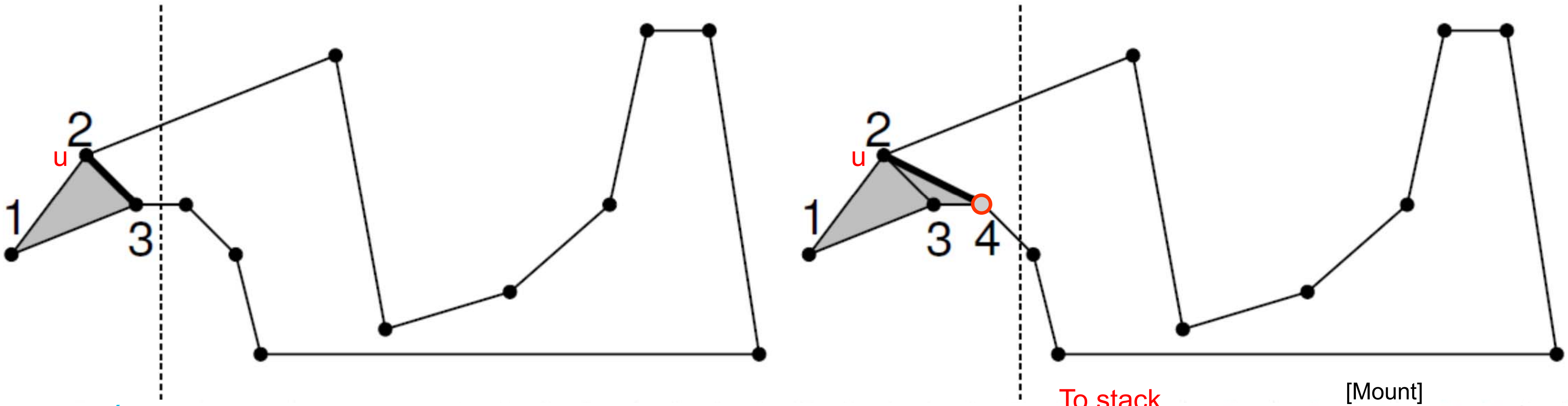
- Sweep left to right - in O(n) time
- Triangulate everything you can by adding diagonals between visible points
- Remove triangulated region from further consideration – mark as DONE



[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

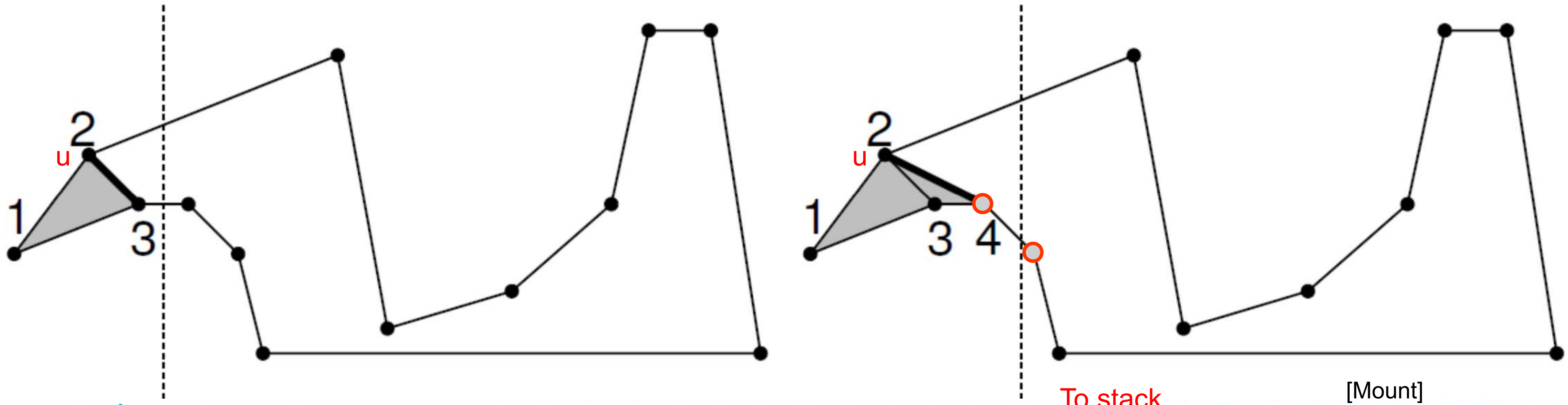


To stack

[Mount]

2. Triangulation of the monotone polygon

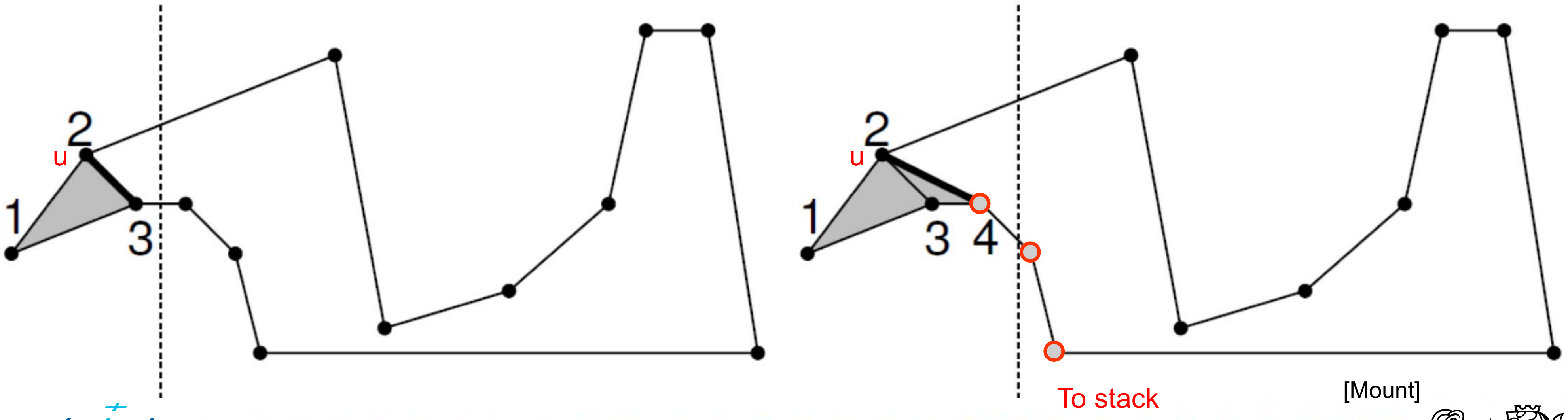
- Sweep left to right - in O(n) time
- Triangulate everything you can by adding diagonals between visible points
- Remove triangulated region from further consideration – mark as DONE



[Mount]

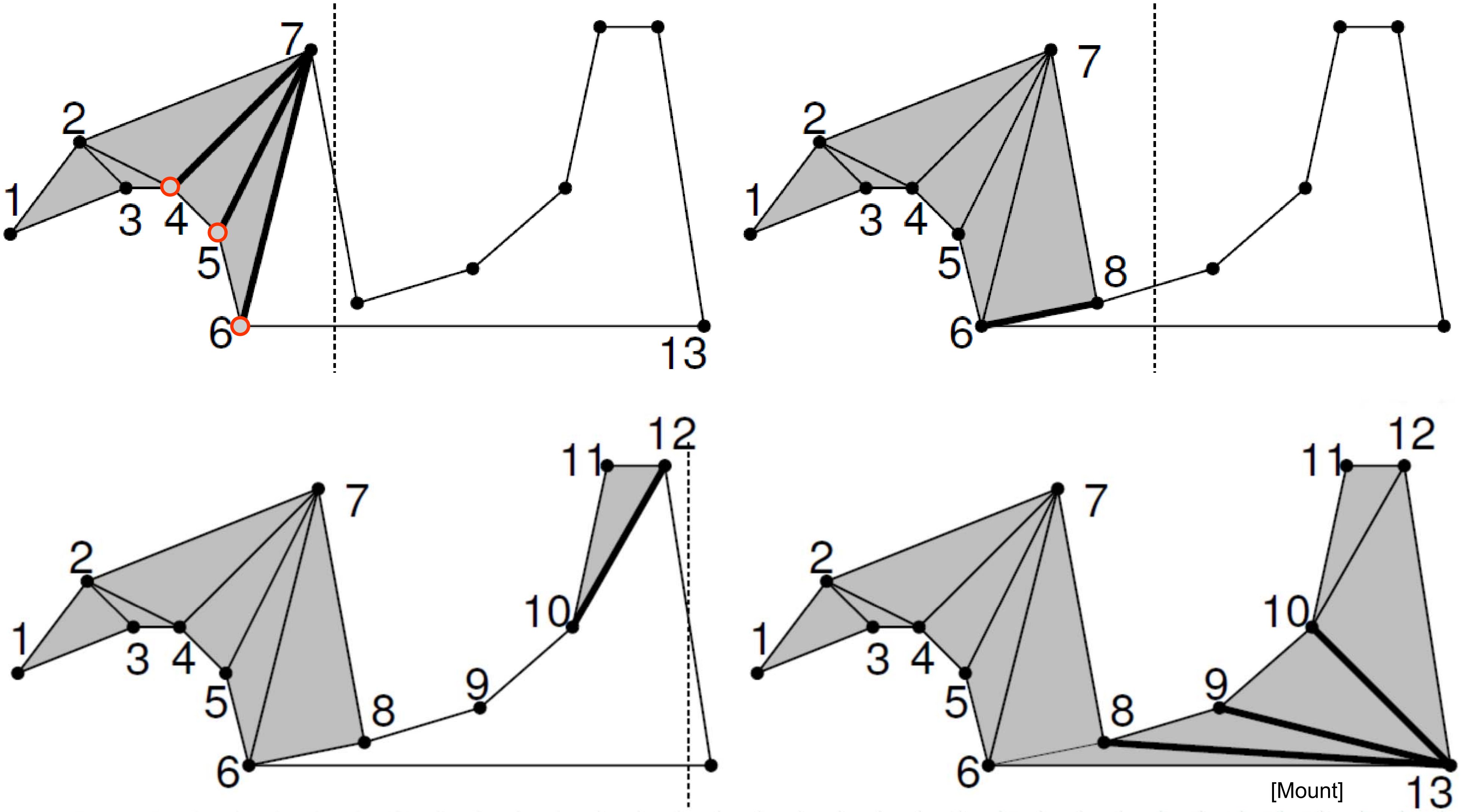
2. Triangulation of the monotone polygon

- 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



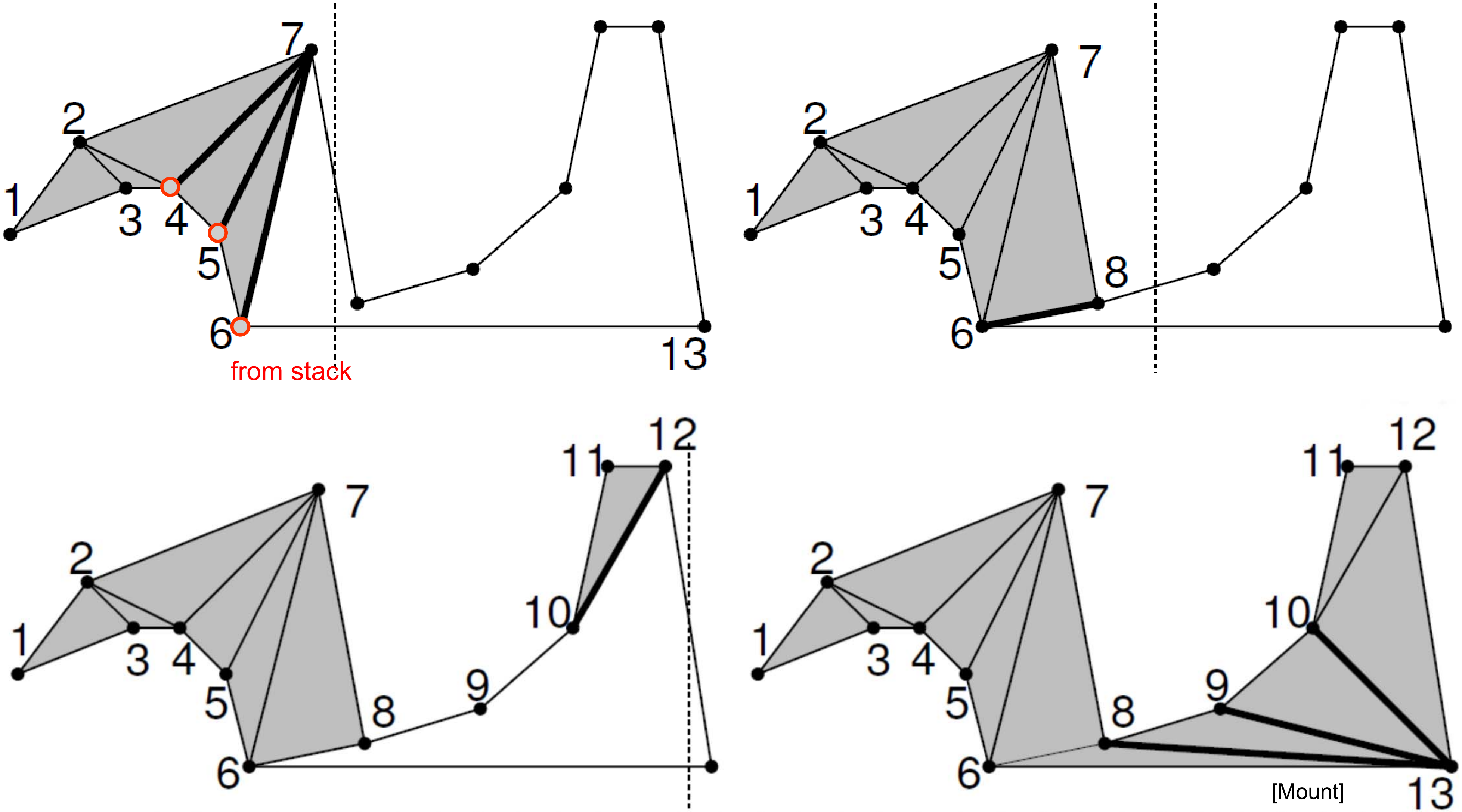
[Mount]

Triangulation of the monotone polygon



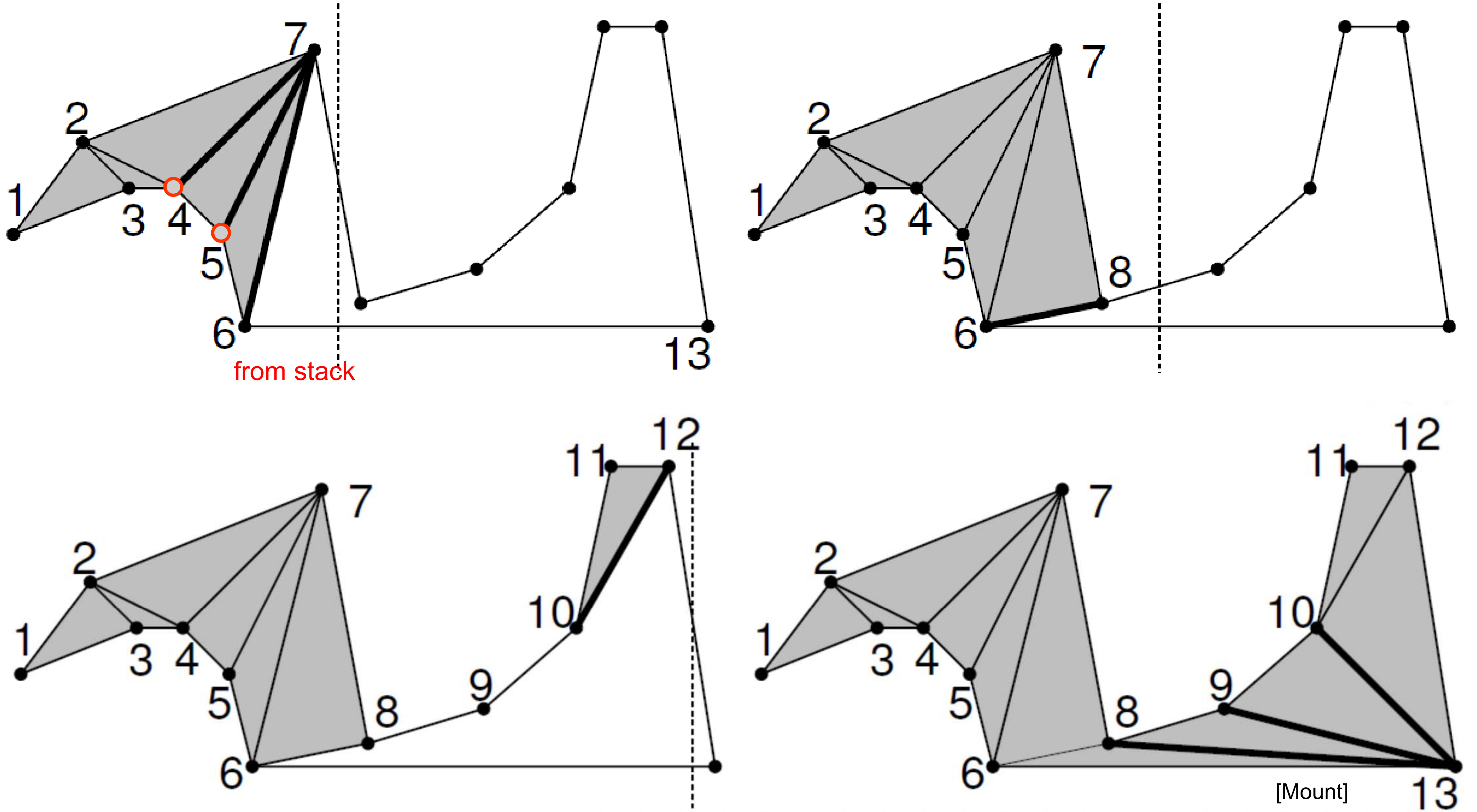
[Mount]

Triangulation of the monotone polygon



[Mount]

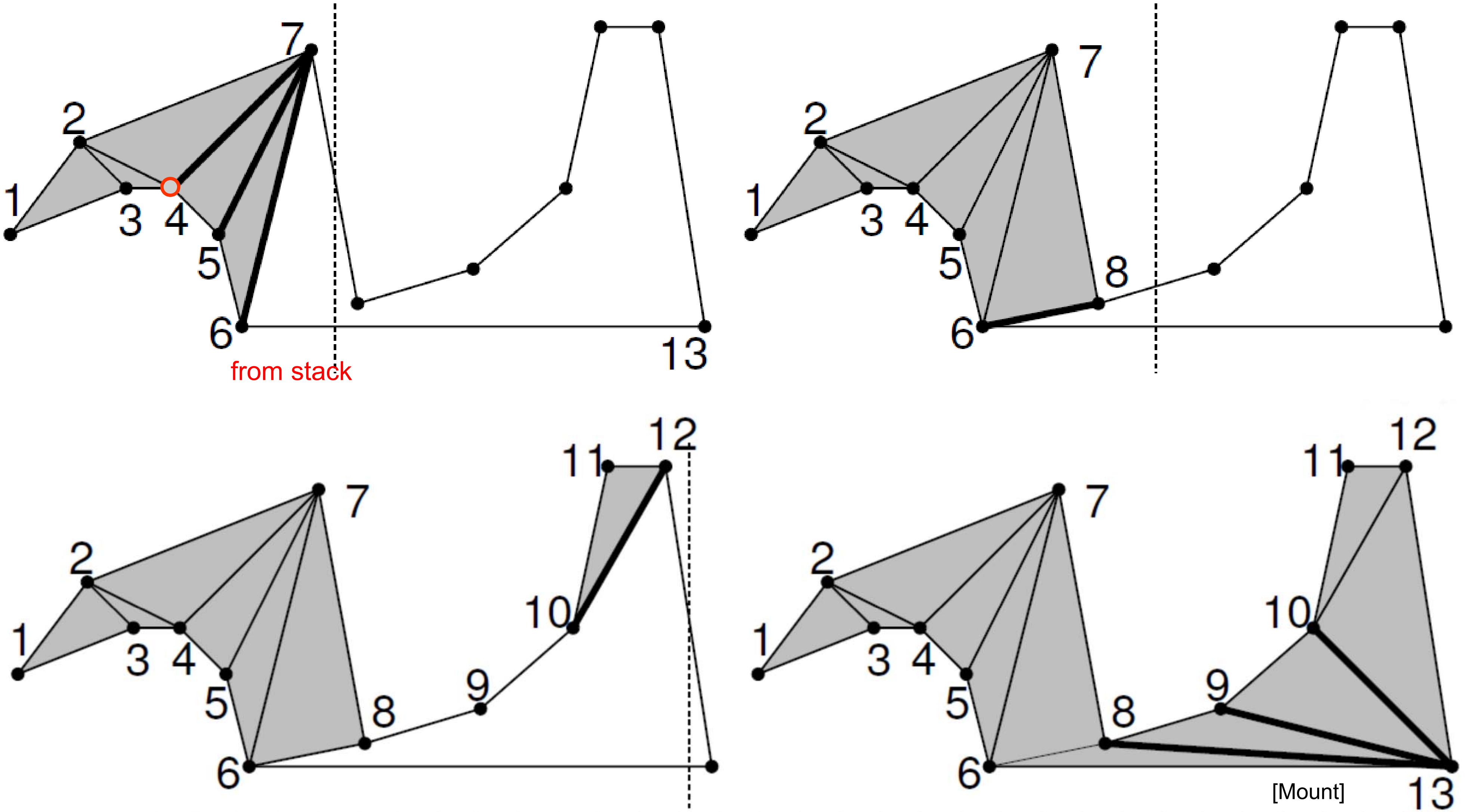
Triangulation of the monotone polygon



from stack

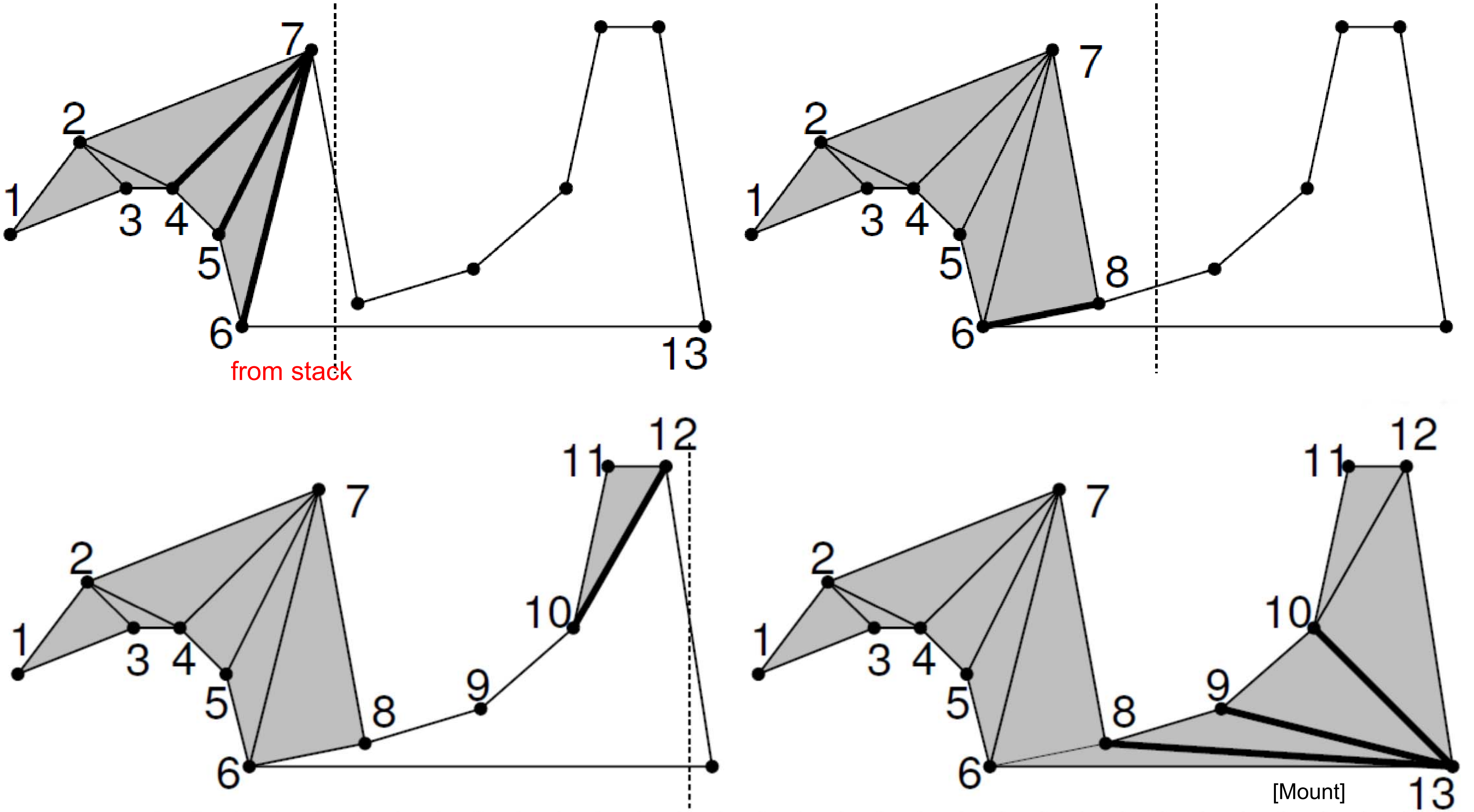
[Mount]

Triangulation of the monotone polygon



[Mount]

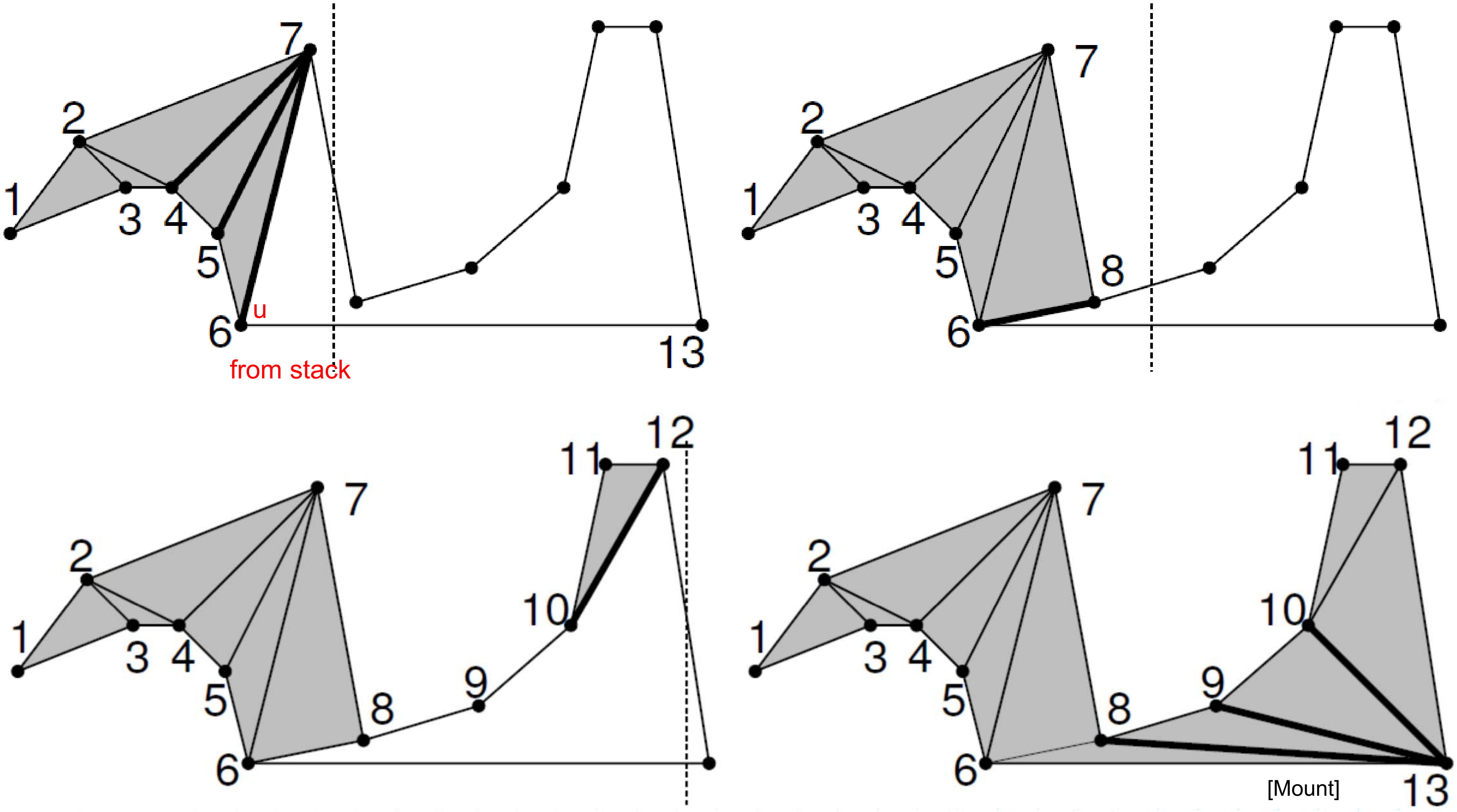
Triangulation of the monotone polygon



from stack

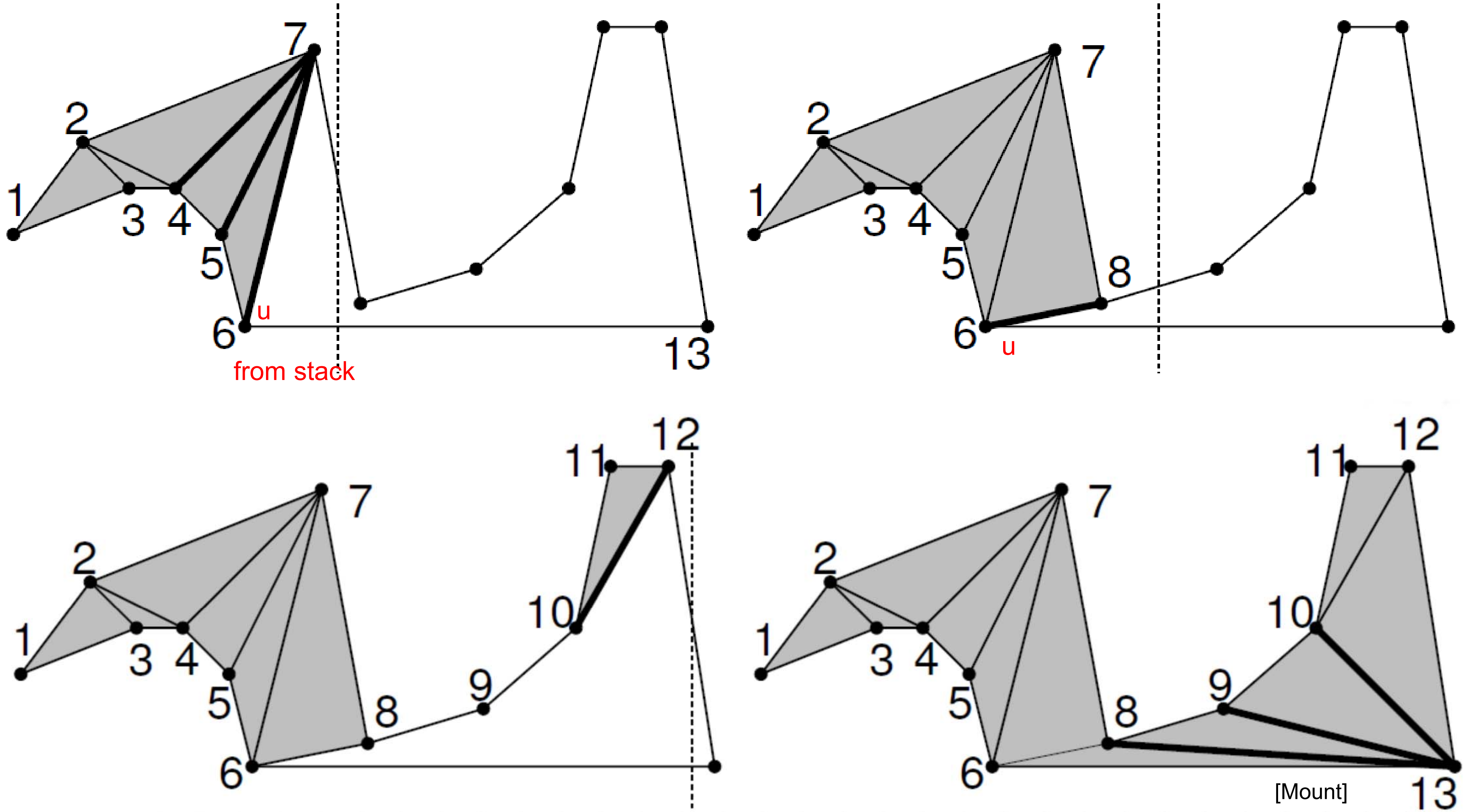
[Mount]

Triangulation of the monotone polygon



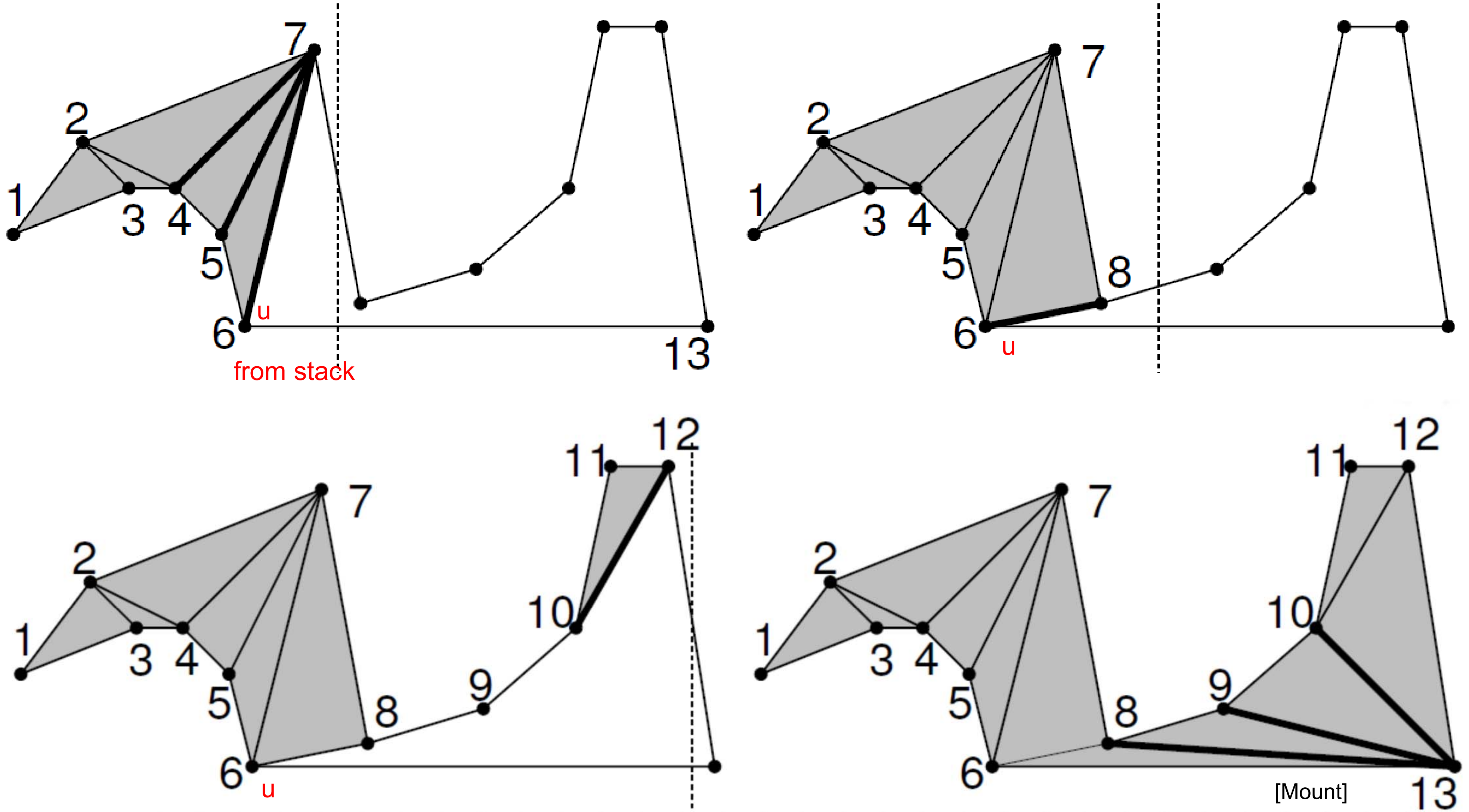
[Mount]

Triangulation of the monotone polygon

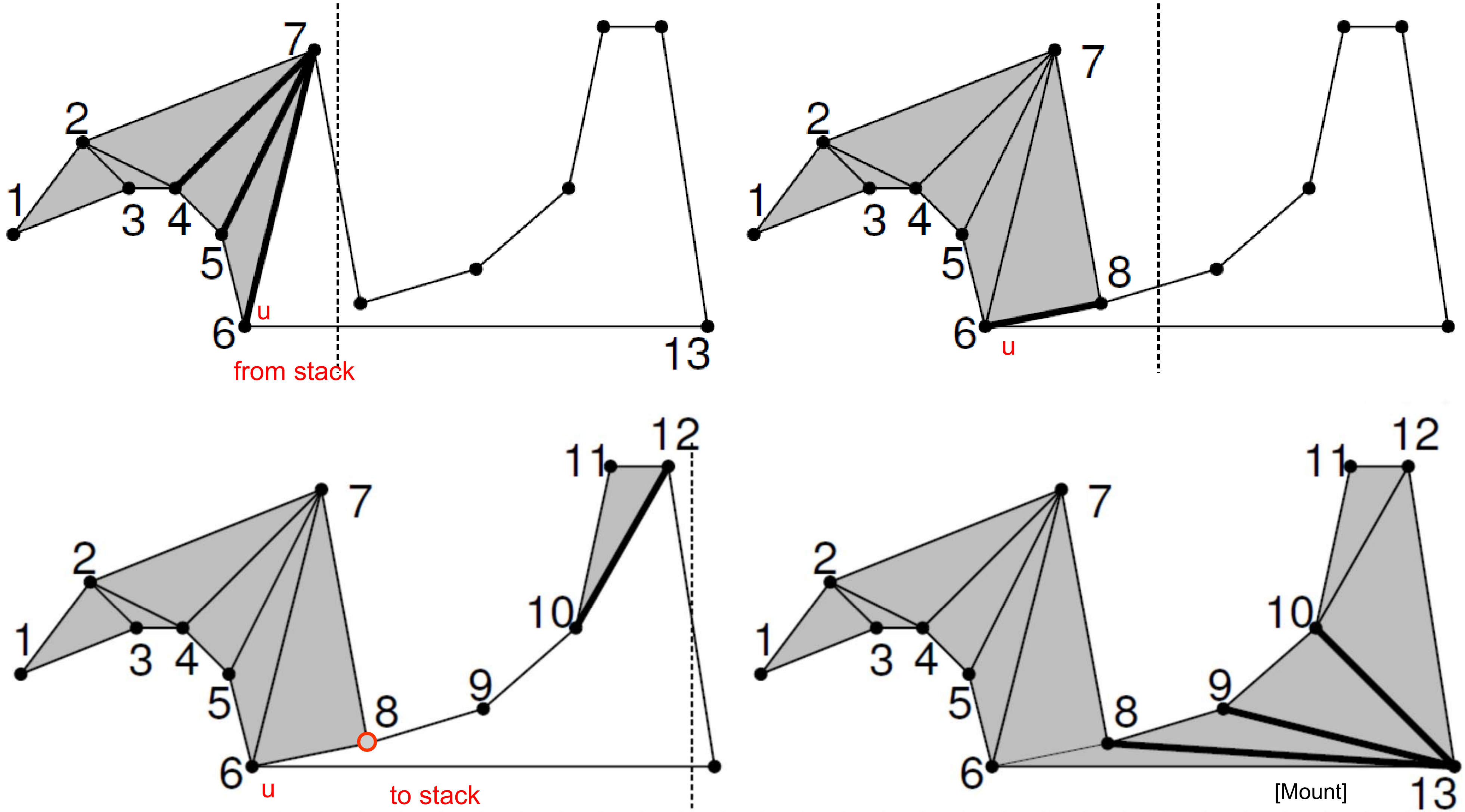


[Mount]

Triangulation of the monotone polygon

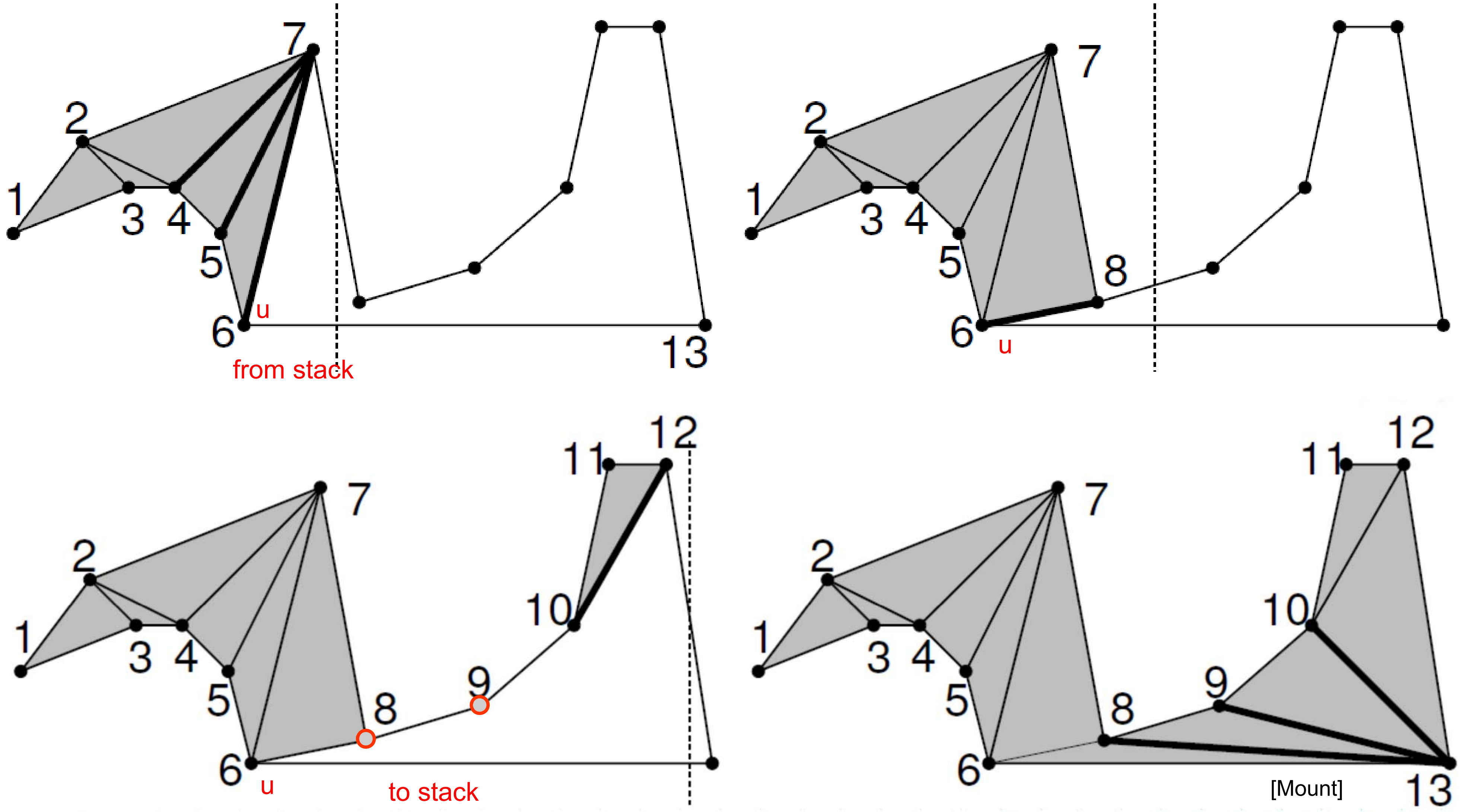


Triangulation of the monotone polygon



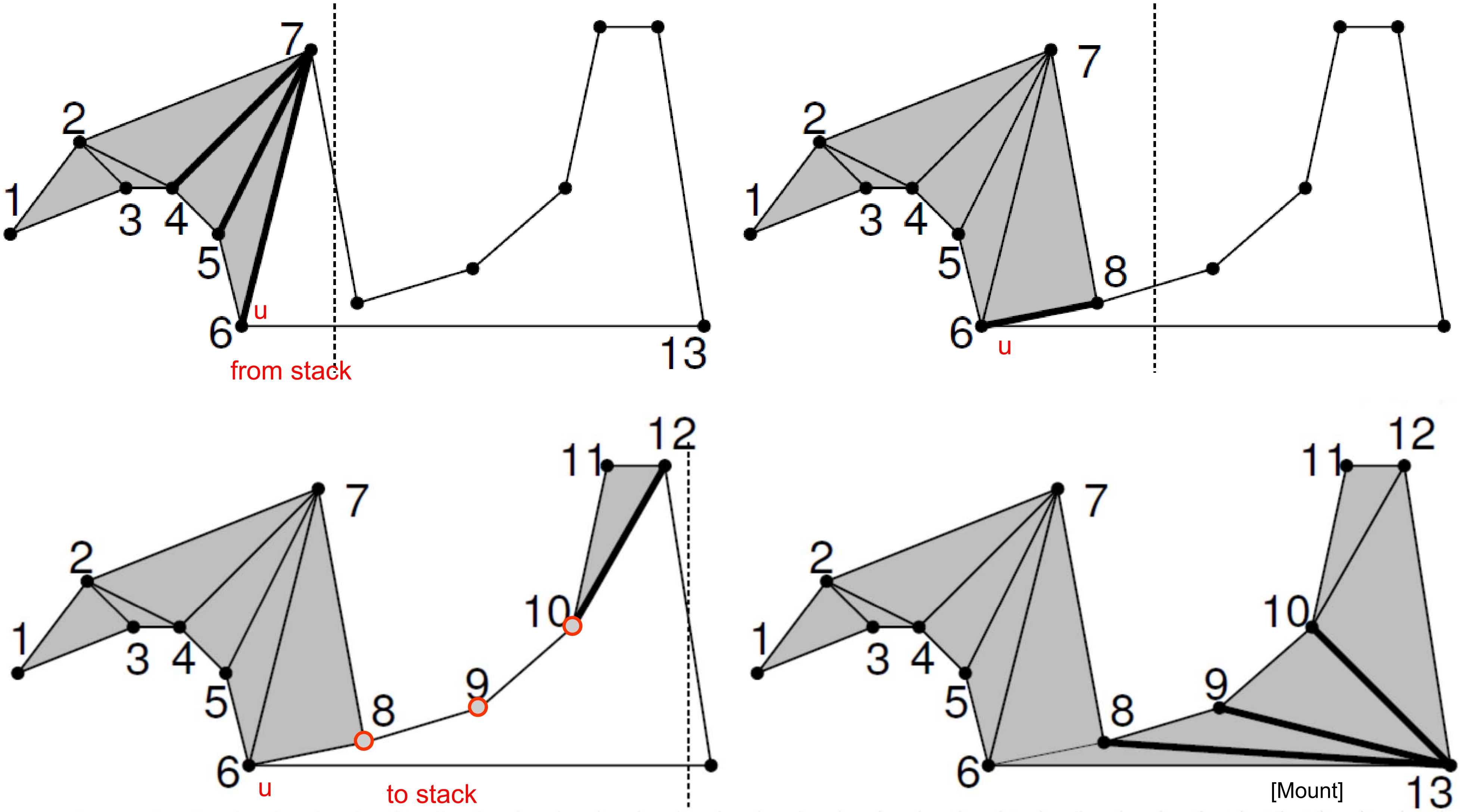
[Mount]

Triangulation of the monotone polygon



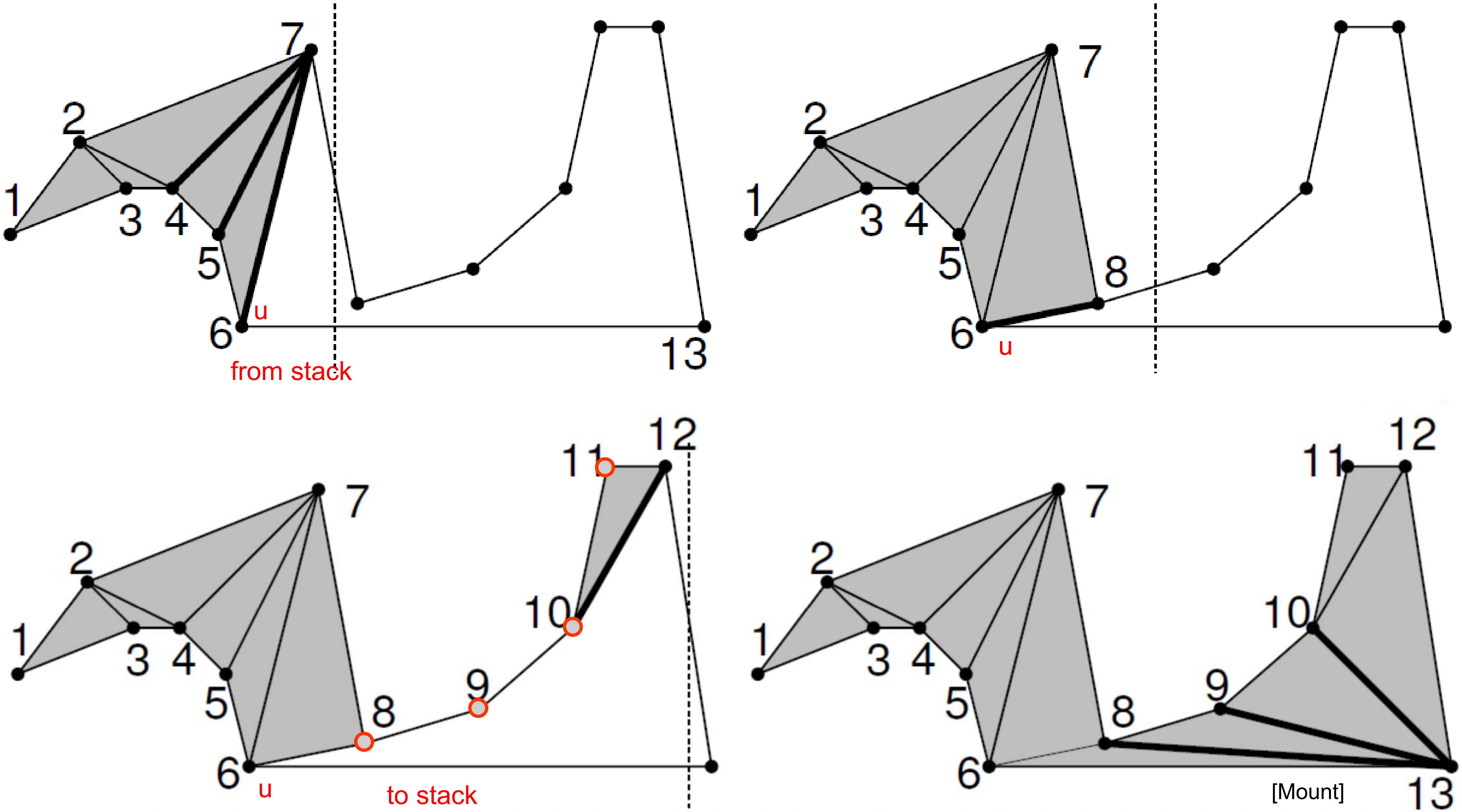
[Mount]

Triangulation of the monotone polygon



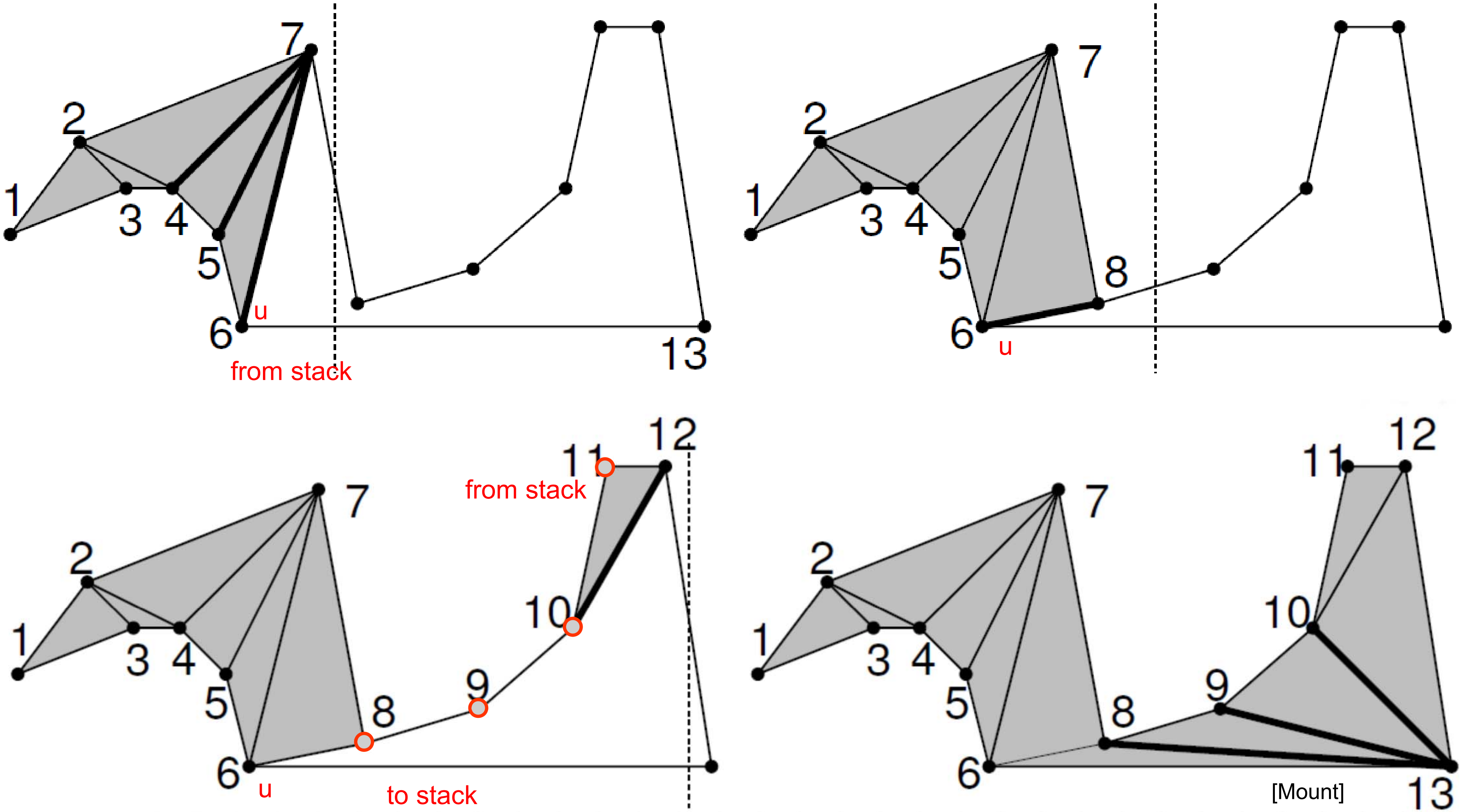
[Mount]

Triangulation of the monotone polygon



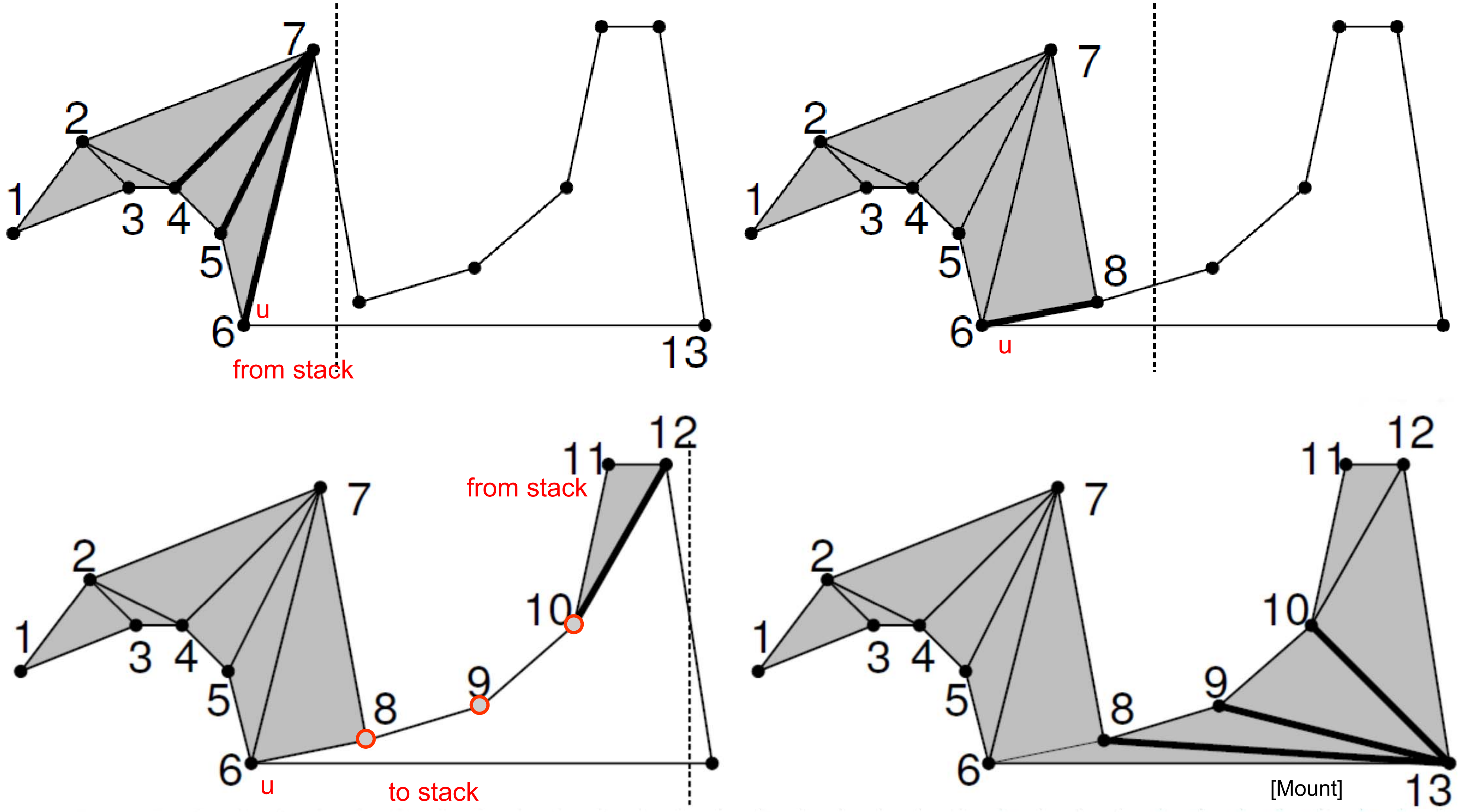
[Mount]

Triangulation of the monotone polygon

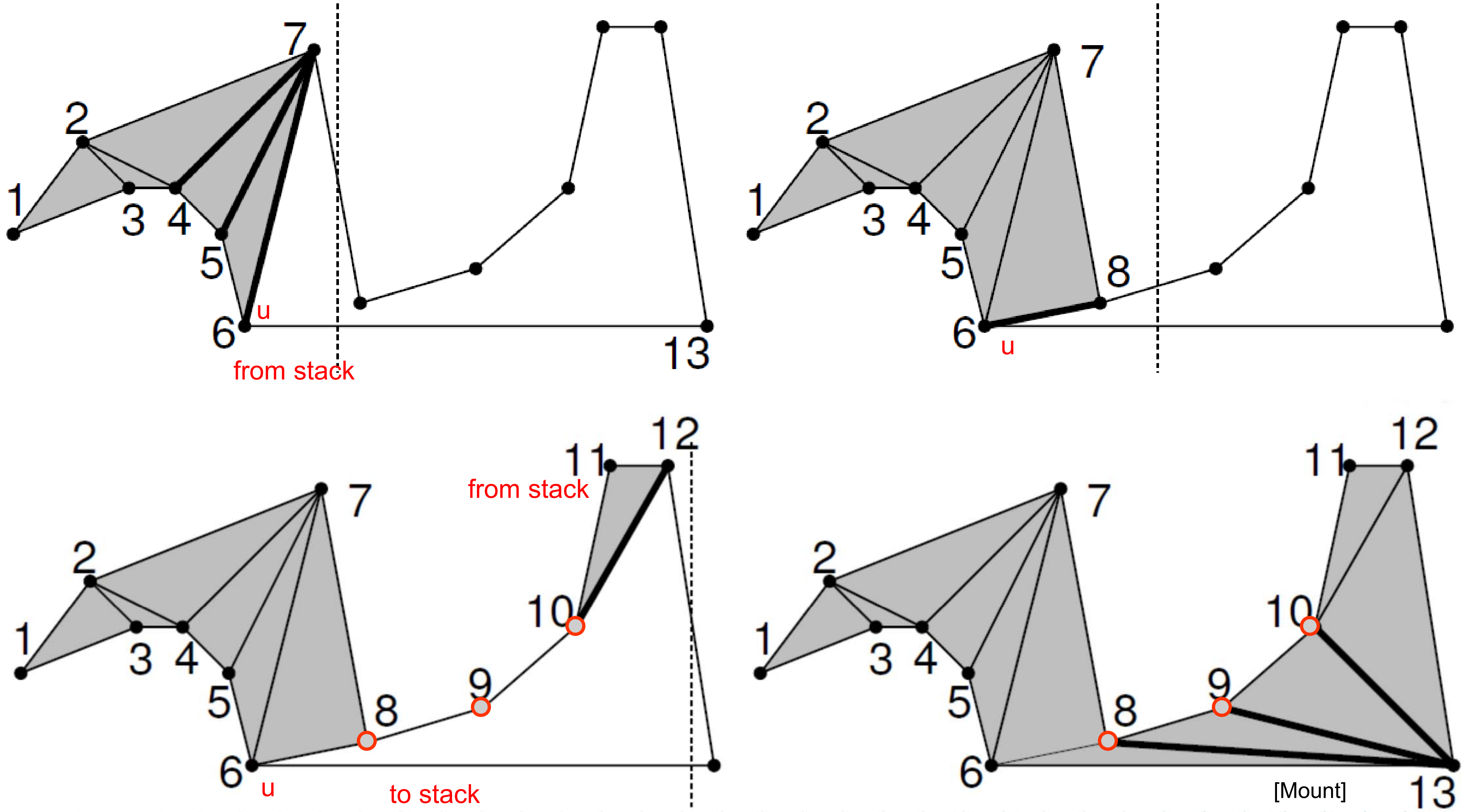


[Mount]

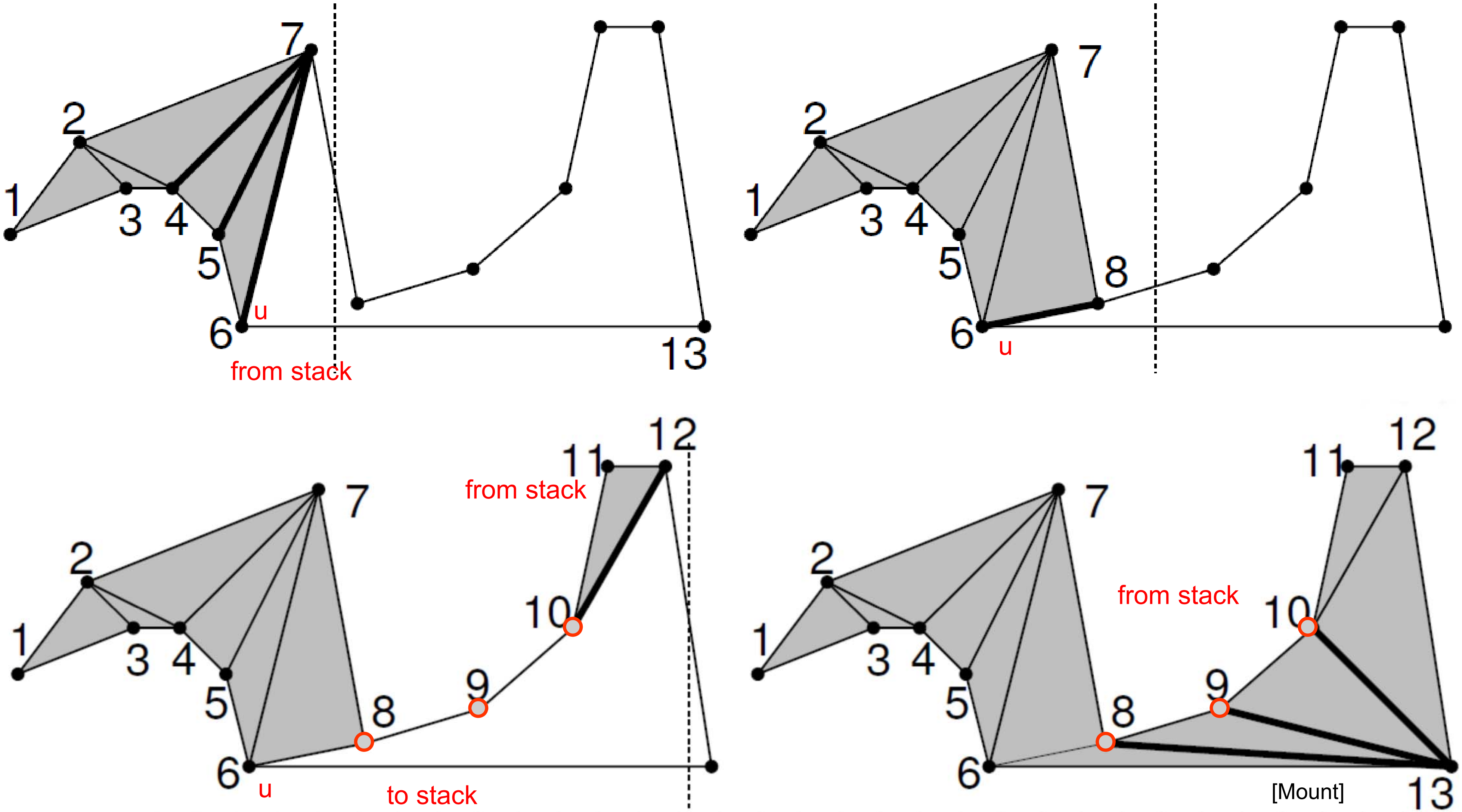
Triangulation of the monotone polygon



Triangulation of the monotone polygon

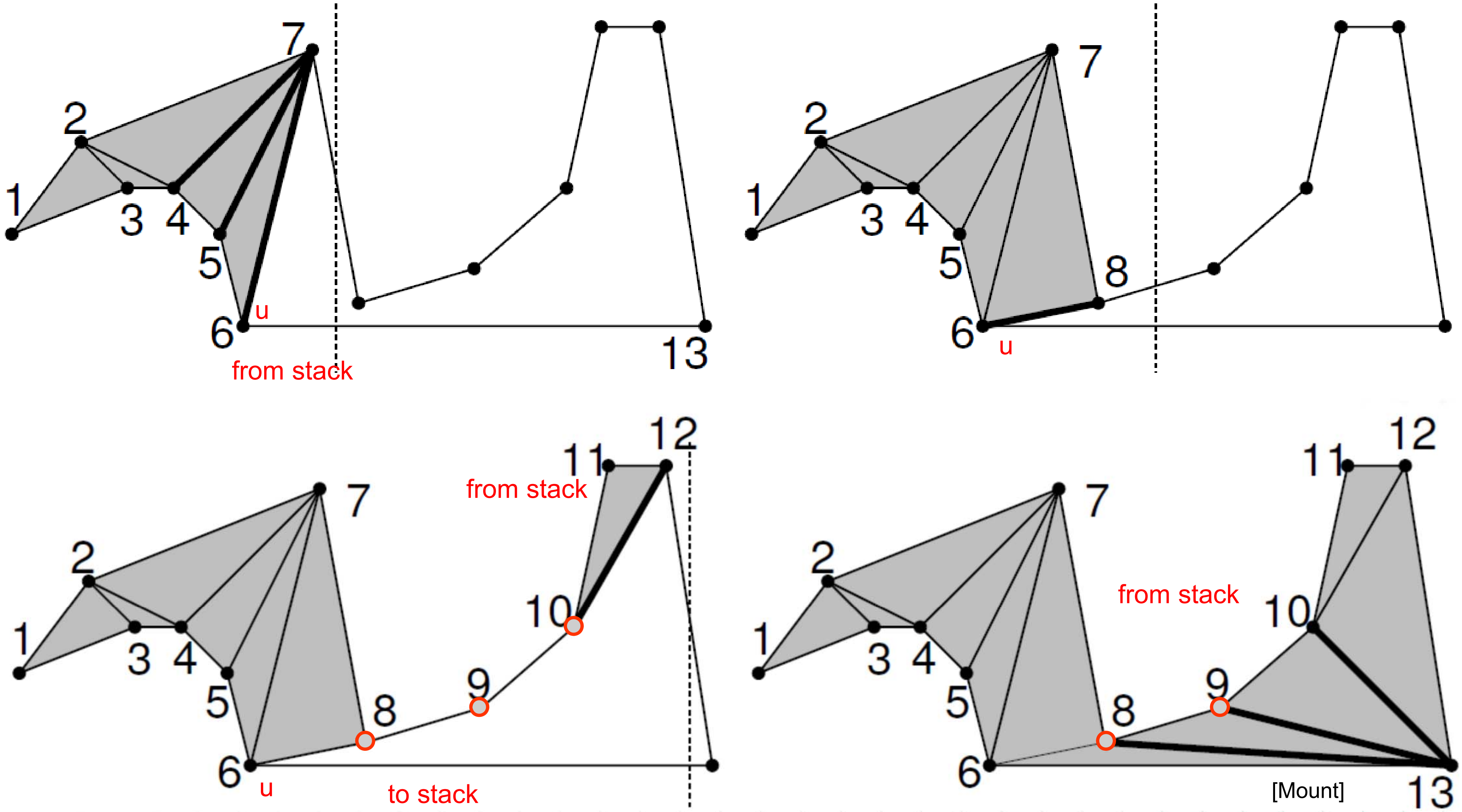


Triangulation of the monotone polygon



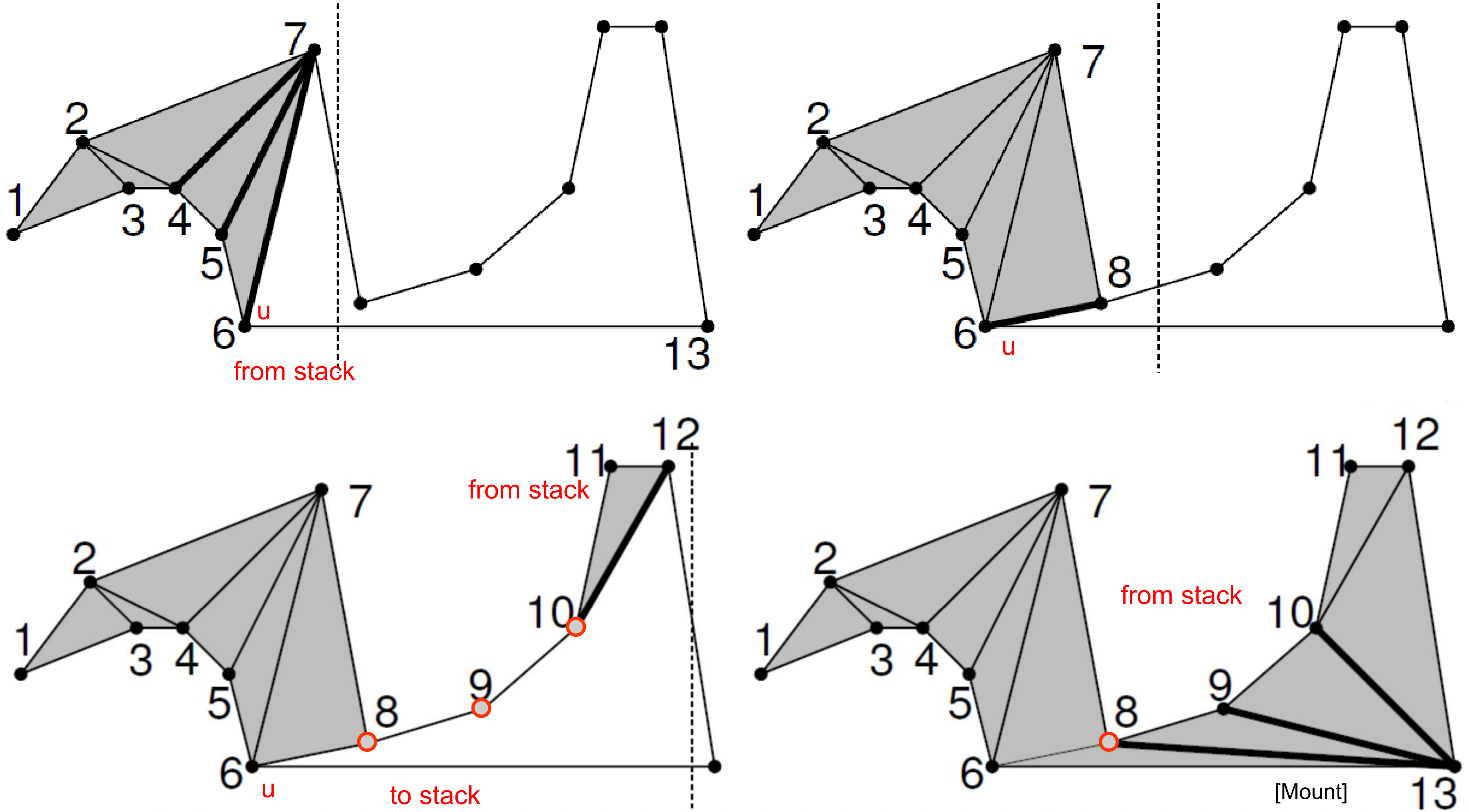
[Mount]

Triangulation of the monotone polygon



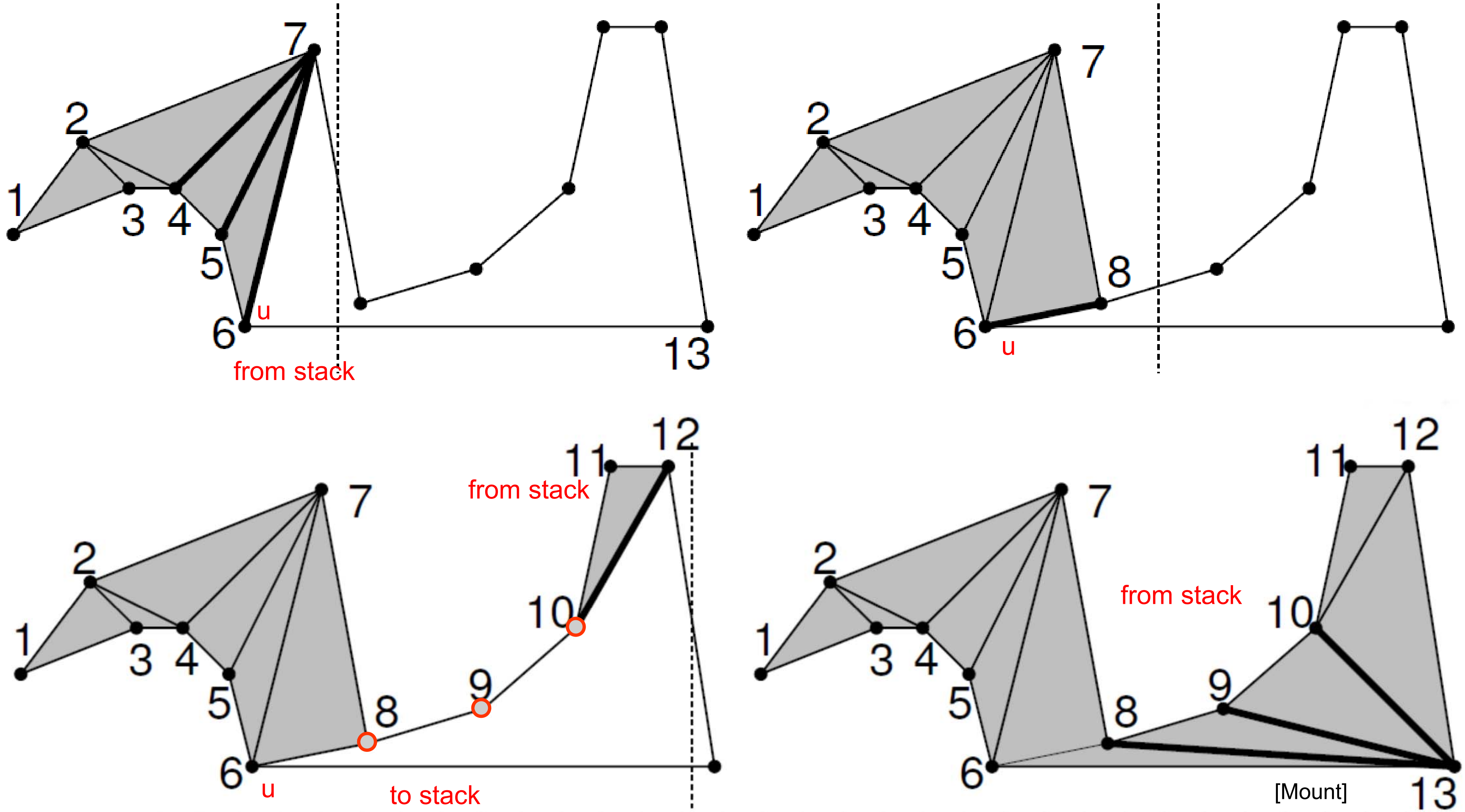
[Mount]

Triangulation of the monotone polygon



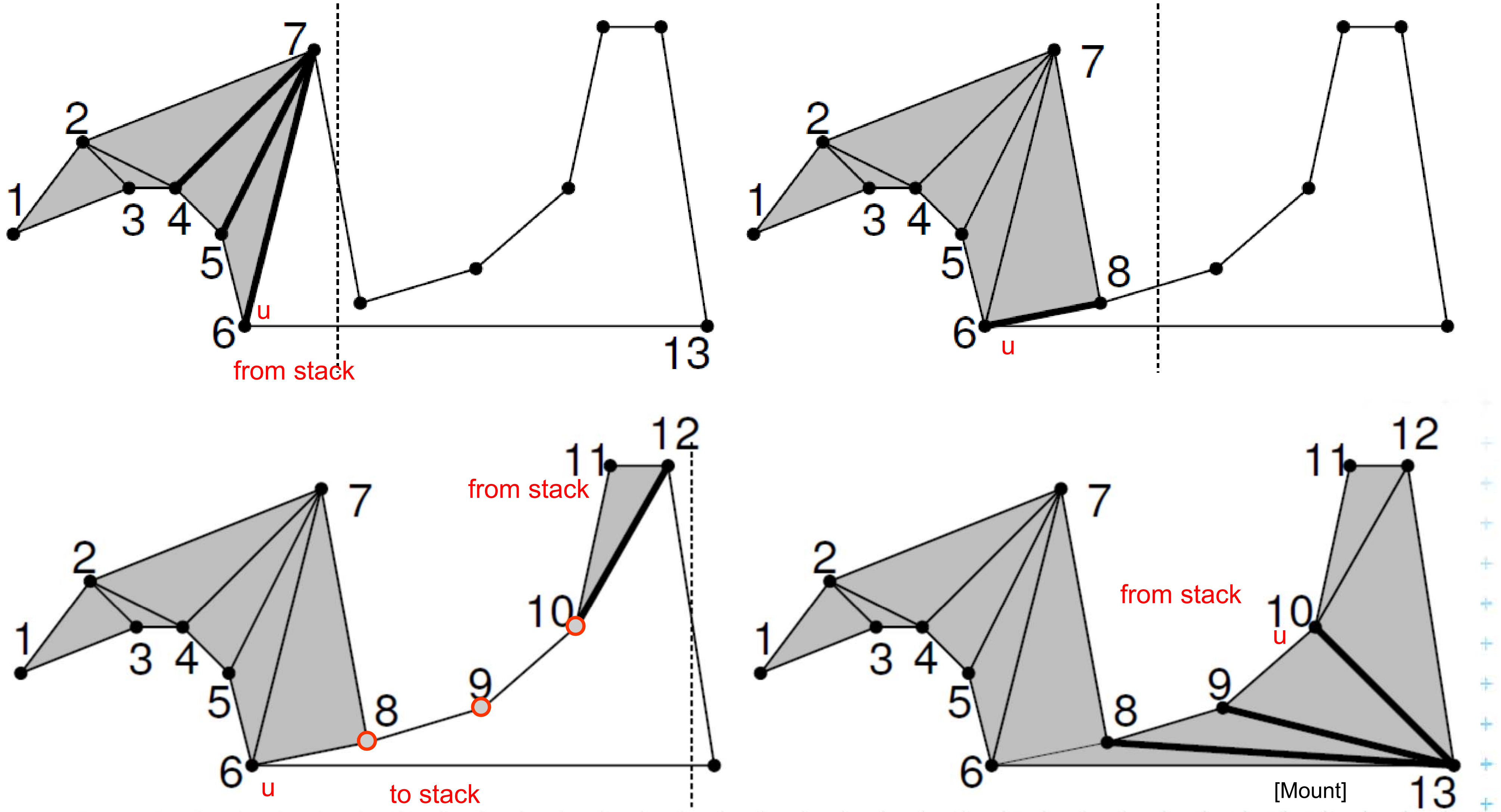
[Mount]

Triangulation of the monotone polygon



[Mount]

Triangulation of the monotone polygon

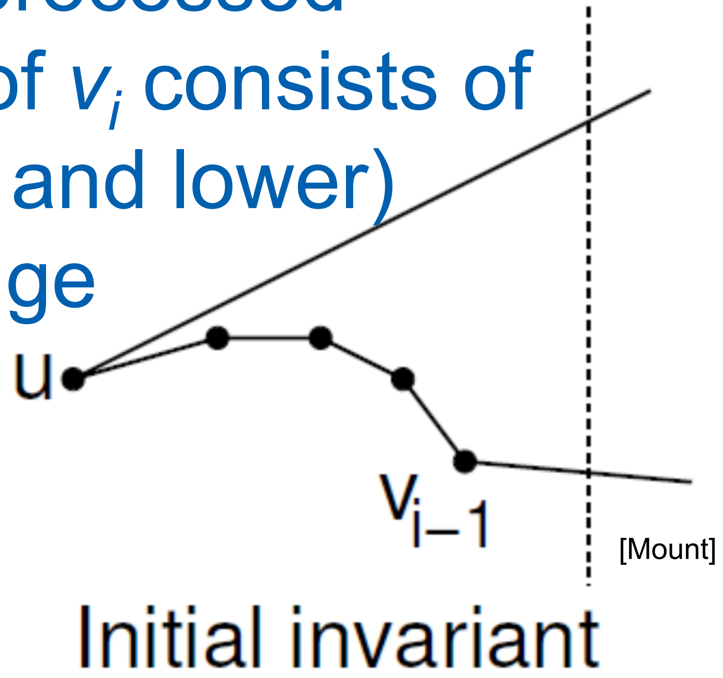


[Mount]

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



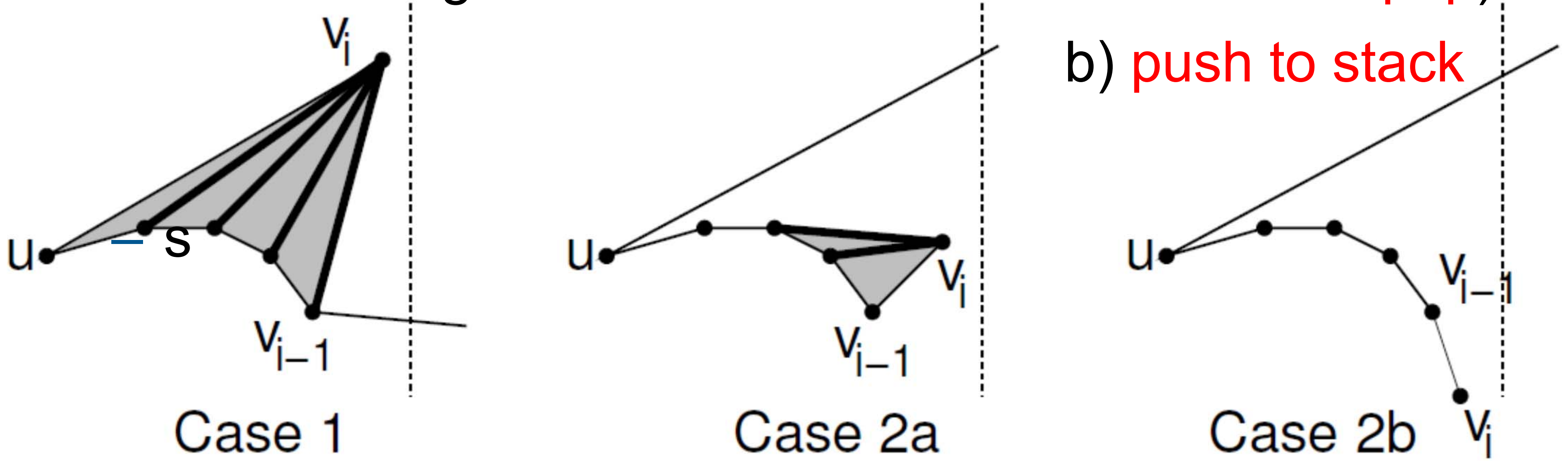
Triangulation cases for v_i (vertex being just processed)

- Case 1: v_i lies on the opposite chain
  - Add diagonals from next(u) to v_{i-1} (empty the stack-pop)
  - 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



Case 1 Case 2a Case 2b

[Mount]

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)

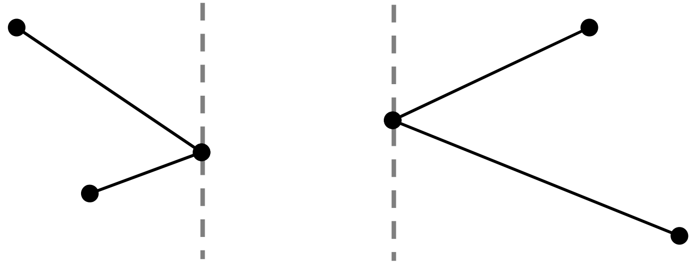
Simple polygon triangulation

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

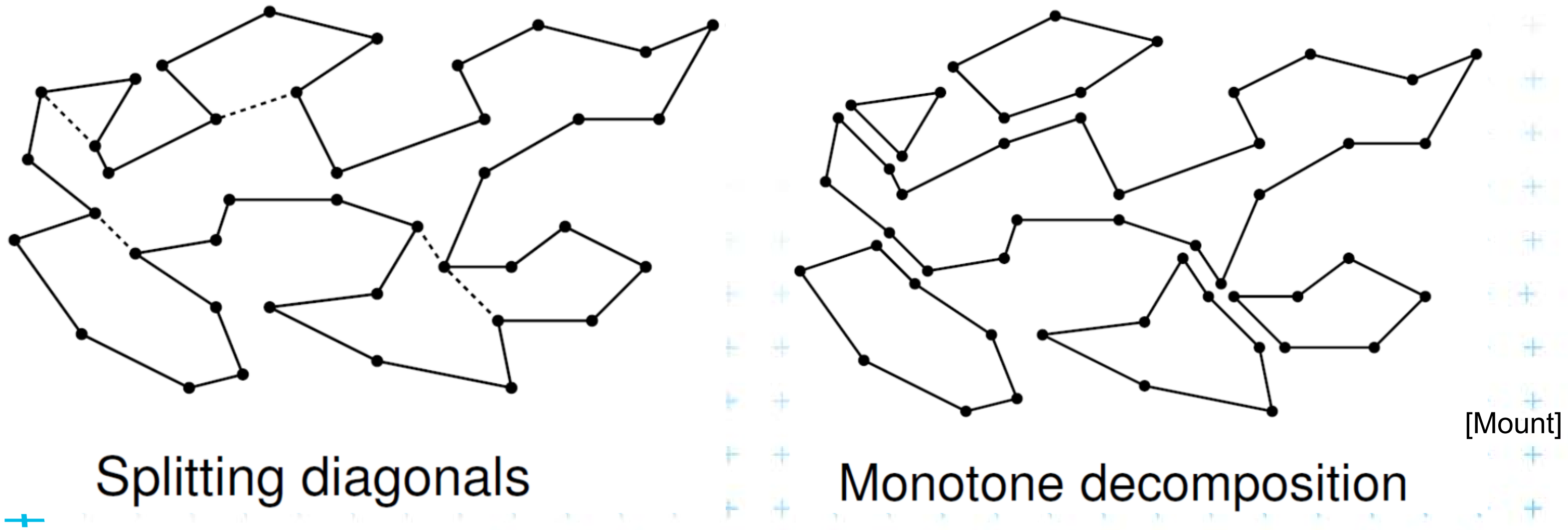
(we will discuss the steps in the reversed order)

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)



[Mount]

Splitting diagonals

Monotone decomposition

Data structures for subdivision

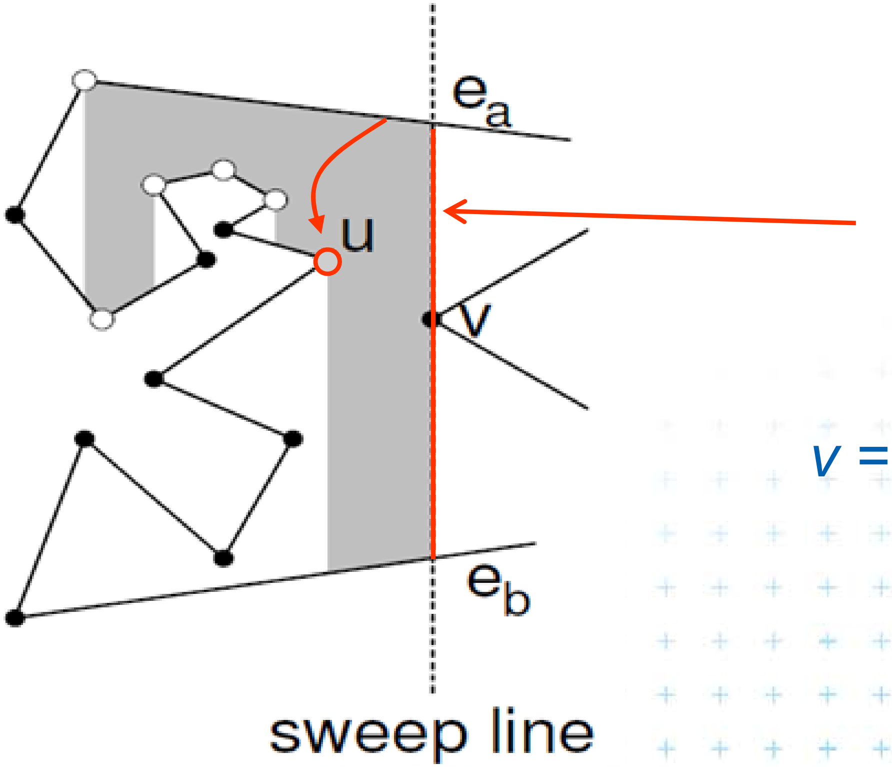
- Events
  - Endpoints of edges, known from the beginning
  - Can be stored in sorted list – no priority queue
- Sweep status
  - List of edges intersecting sweep line (top to bottom)
  - Stored in O(log n) time dictionary (like balanced tree)
- Event processing
  - Six event types based on local structure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

Helper – definition

helper(e_a)

= the rightmost vertically visible processed vertex *u on or* 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



all these vertices

see ○ u = helper(e_a)

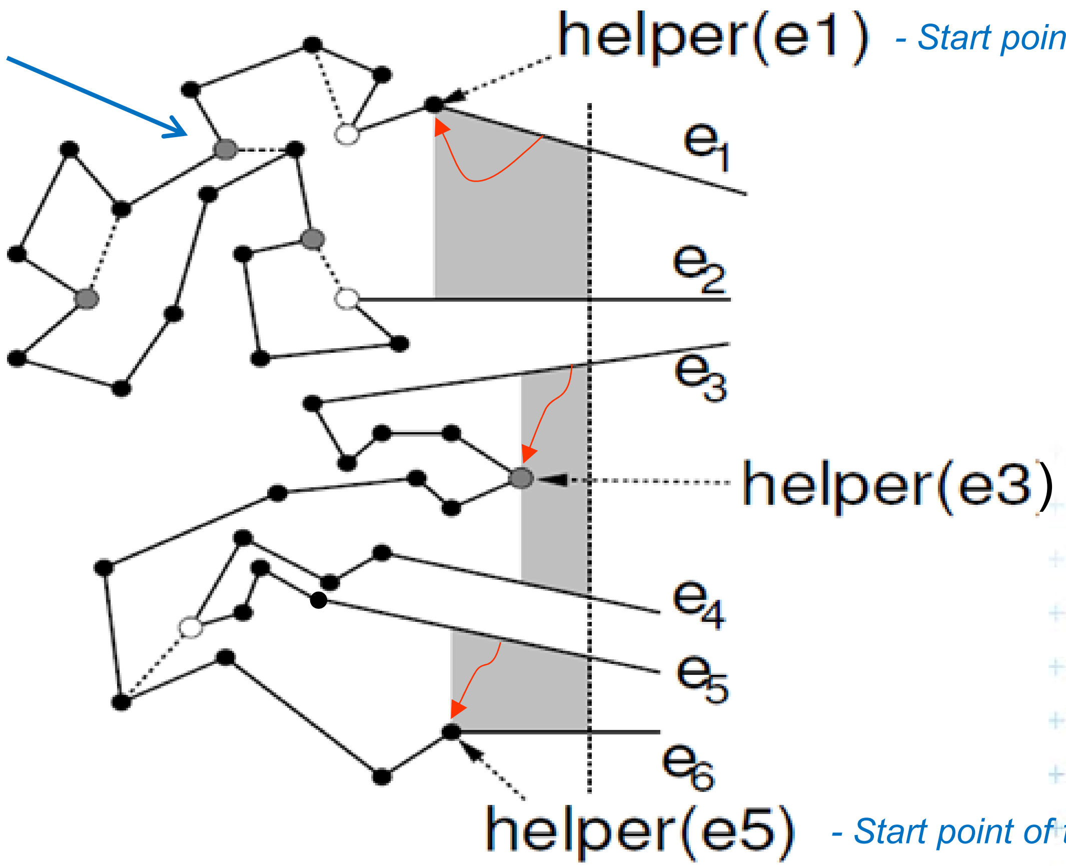
v = current vertex
(sweep line stop)

Helper

helper(e_a)

is defined only for edges intersected by the sweep line

Previous helper h(e)



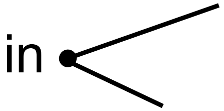
- Start point of the edge itself

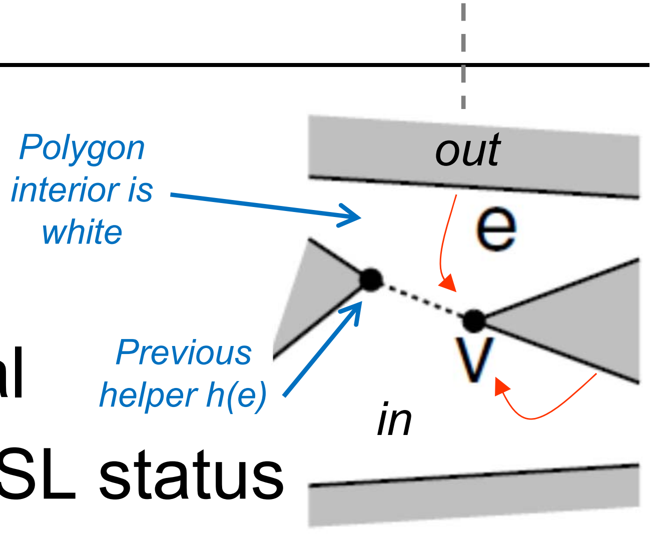
rightmost vertically visible processed vertex

- Start point of the edge below

Six event types of vertex *v*

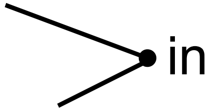
1. Split vertex

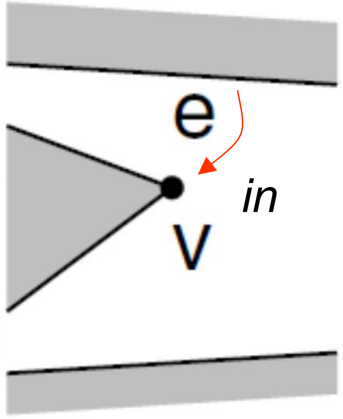




- Find edge *e* above *v*, connect *v* with helper(e) by diagonal
- Add 2 new edges incident to *v* into SL status
- Set new helper(e) = helper(lower edge of these two) = *v*

2. Merge vertex





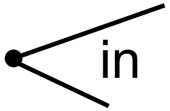
[Mount]

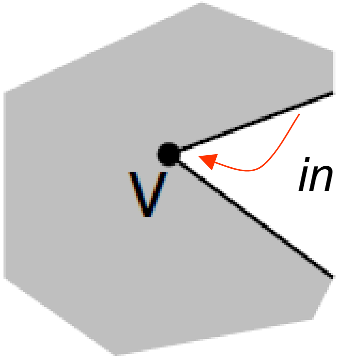
- Find two edges incident with *v* in SL status
- Delete both from SL status
- Let *e* is edge immediately above *v*
- Make helper(e) = *v*

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

Six event types of vertex *v*

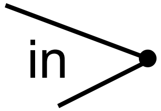
3. Start vertex

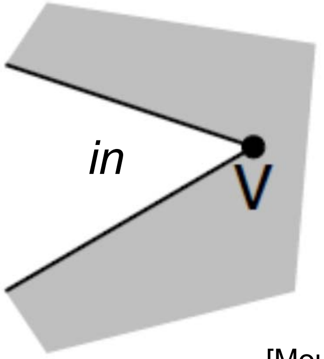




- Both incident edges lie right from *v*
- But interior angle <180°
- Insert both edges to SL status
- Set helper(upper edge) = *v*

4. End vertex



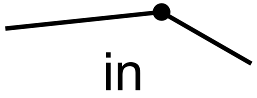


[Mount]

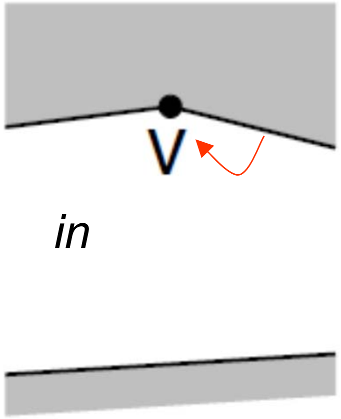
- Both incident edges lie left from *v*
- But interior angle <180°
- Delete both edges from SL status
- No helper set – we are out of the polygon

Six event types of vertex *v*

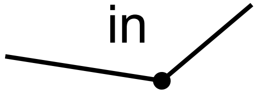
5. Upper chain-vertex



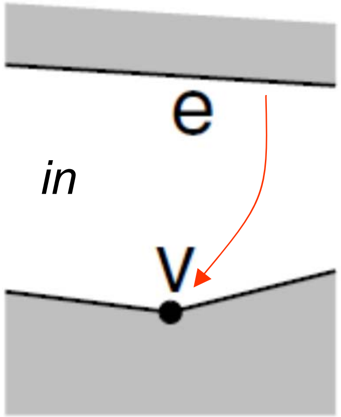
- one side is to the left, one side to the right, interior is below
- replace the left edge with the right edge in SL status
- Make *v* helper of the new (upper) edge



6. Lower chain-vertex



- 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*



[Mount]

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

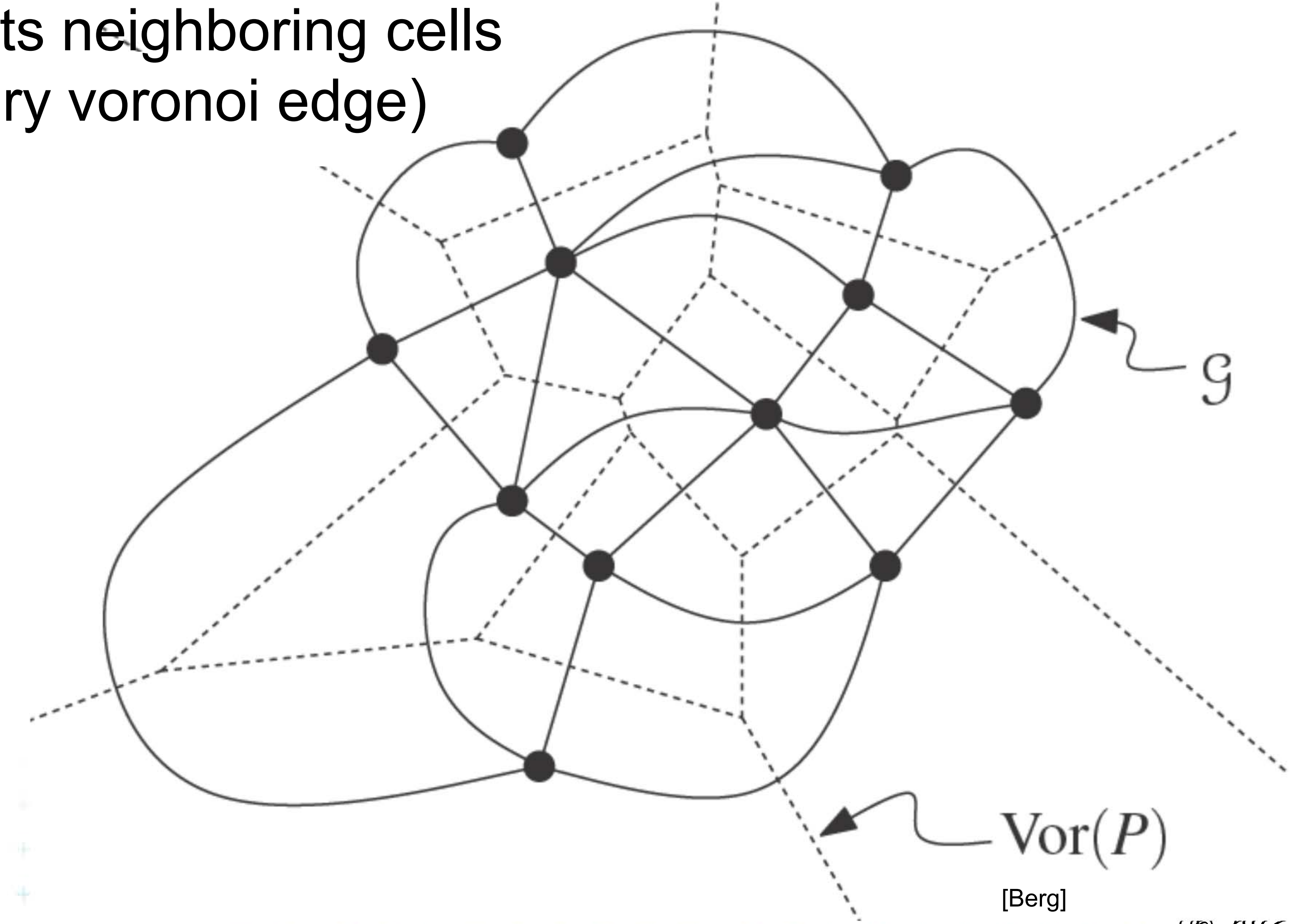
- Complete simple polygon triangulation
  - O(*n* log *n*) time for partitioning into monotone polygons
  - O(*n*) time for triangulation
  - O(*n*) storage

Delaunay triangulation

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)



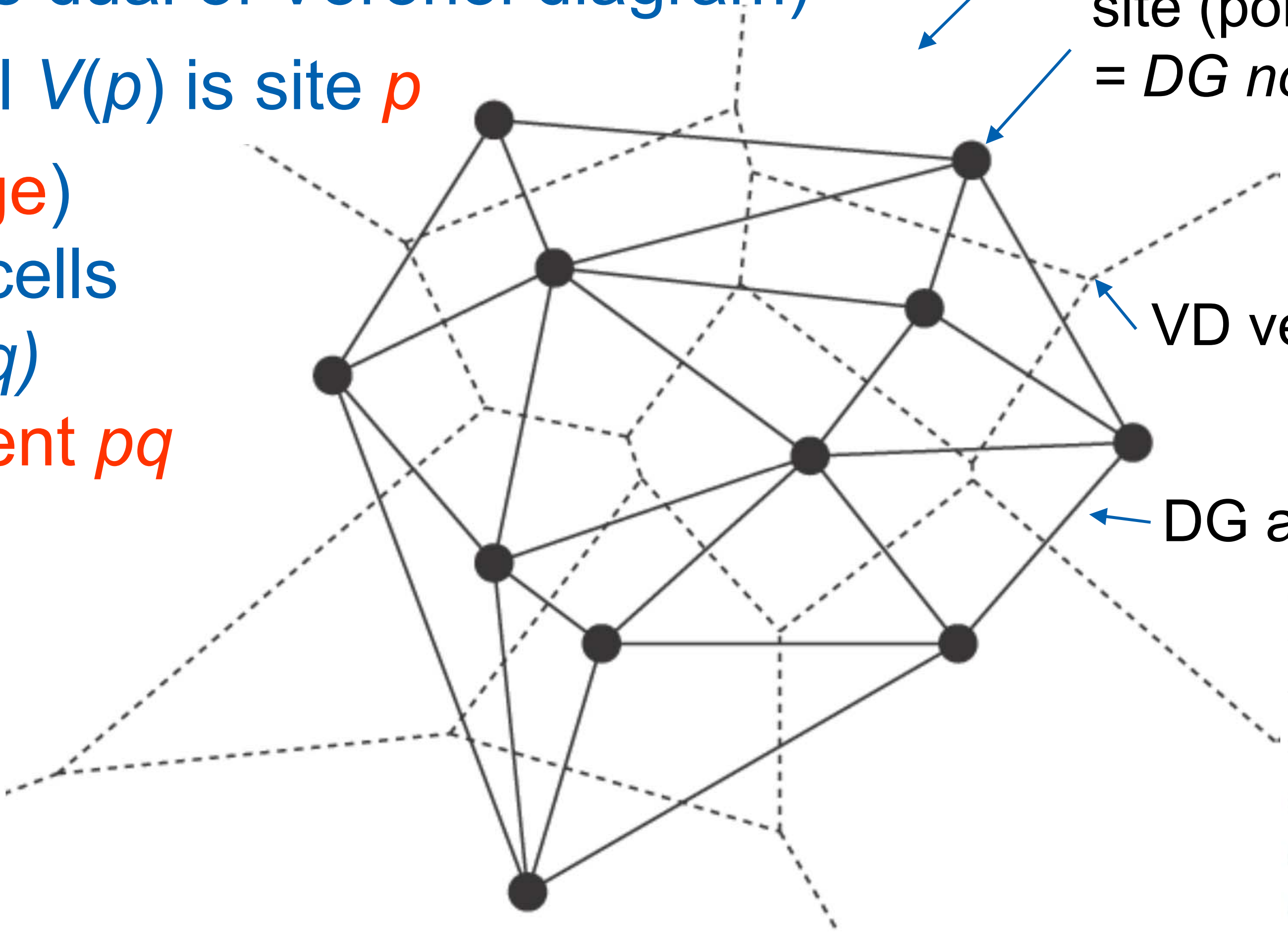
[Berg]

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*

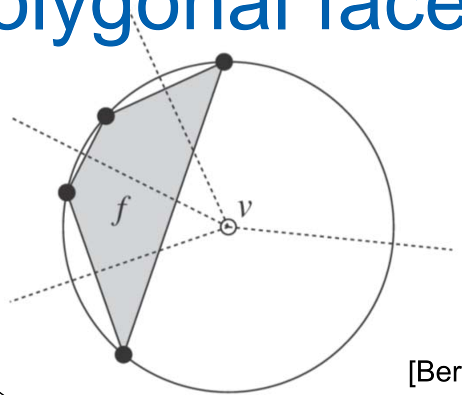
VD vertex

DG arc

[Berg]

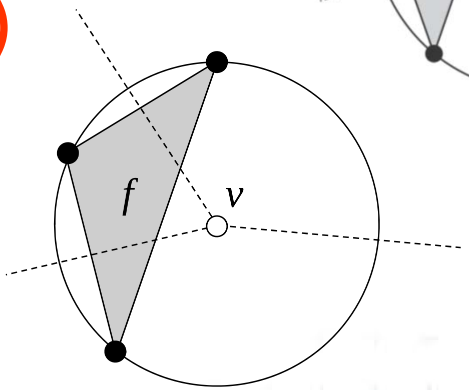
Delaunay graph and Delaunay triangulation

- 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



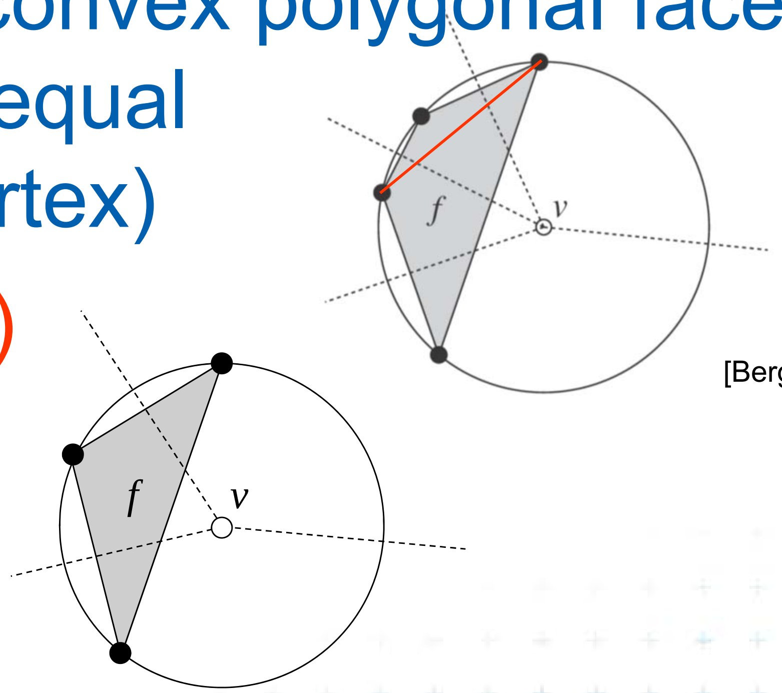
  - No four sites on a circle
  - Faces are triangles (Voronoi vertices have degree = 3)
  - DT is unique (DG not! Can be triangulated differently)

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)
- 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)



[Berg]

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)

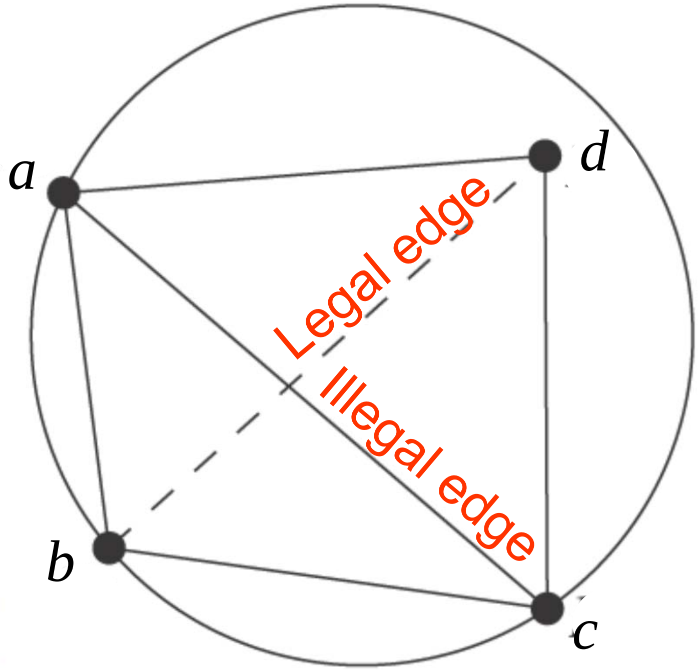
Closest pair property

- 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

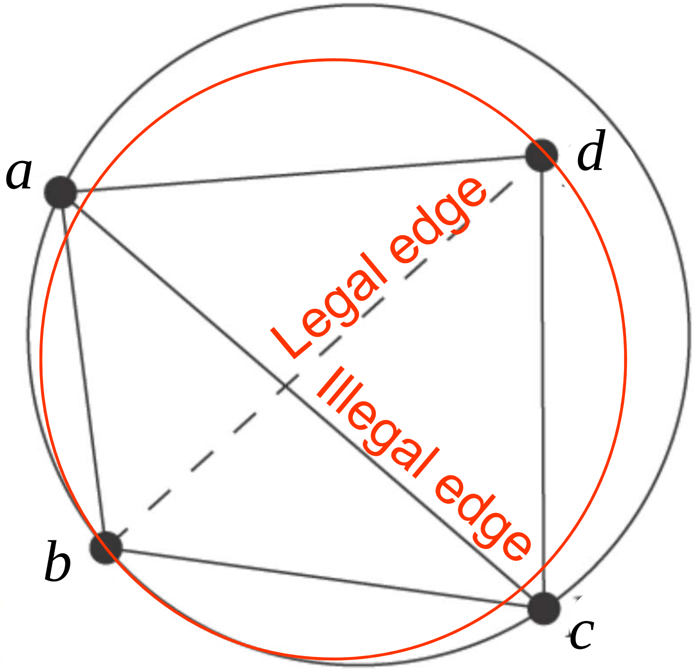


[Berg]

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



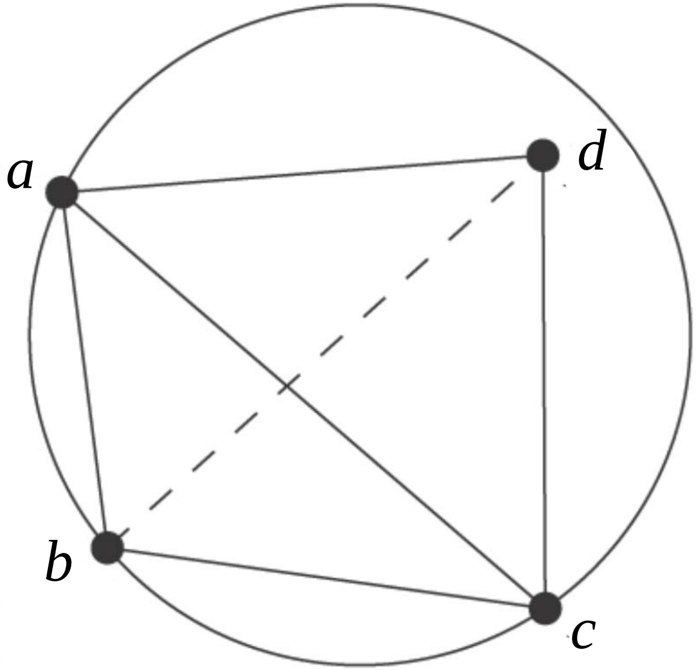
[Berg]

Edge flip operation

Edge flip

= a local operation, that increases the angle vector

- Given two adjacent triangles Δabc and Δcda such that their union forms a convex quadrilateral, the edge flip operation replaces the diagonal *ac* with *bd*.



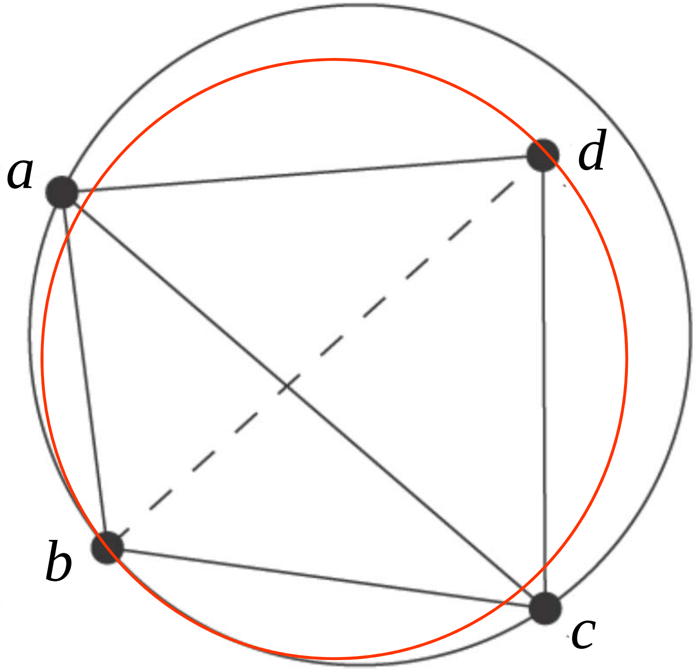
[Berg]

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*.



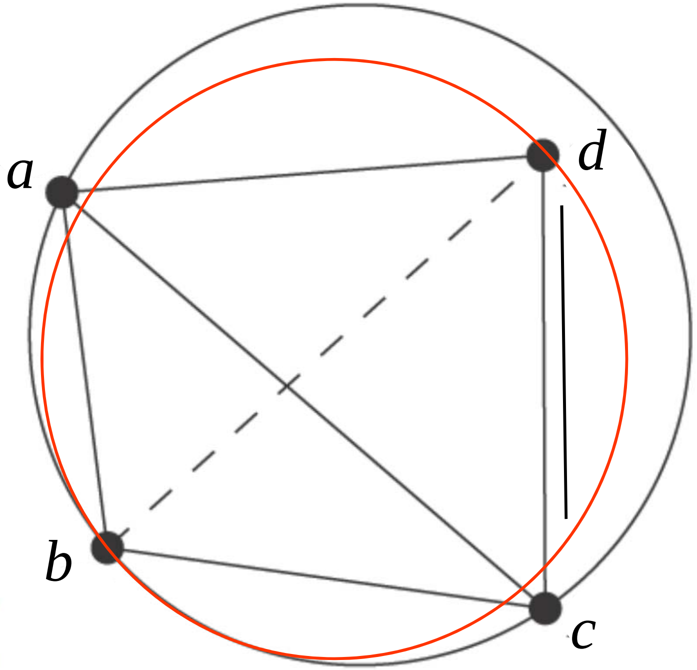
[Berg]

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*.



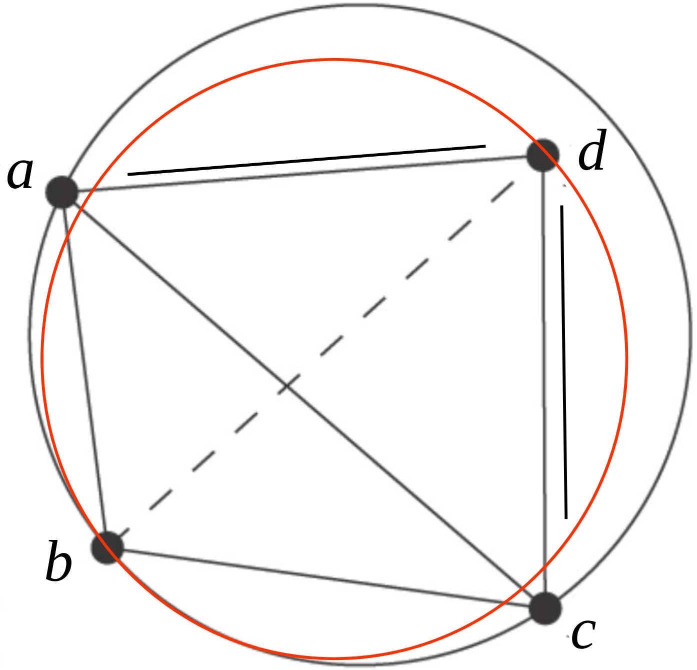
[Berg]

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

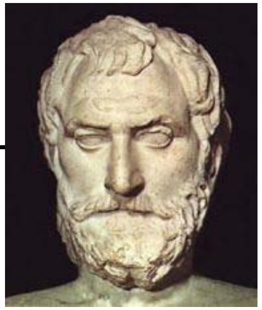
- Let *T* be a triangulation with *m* triangles (and 3*m angles*)
- Angle-vector

  = 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

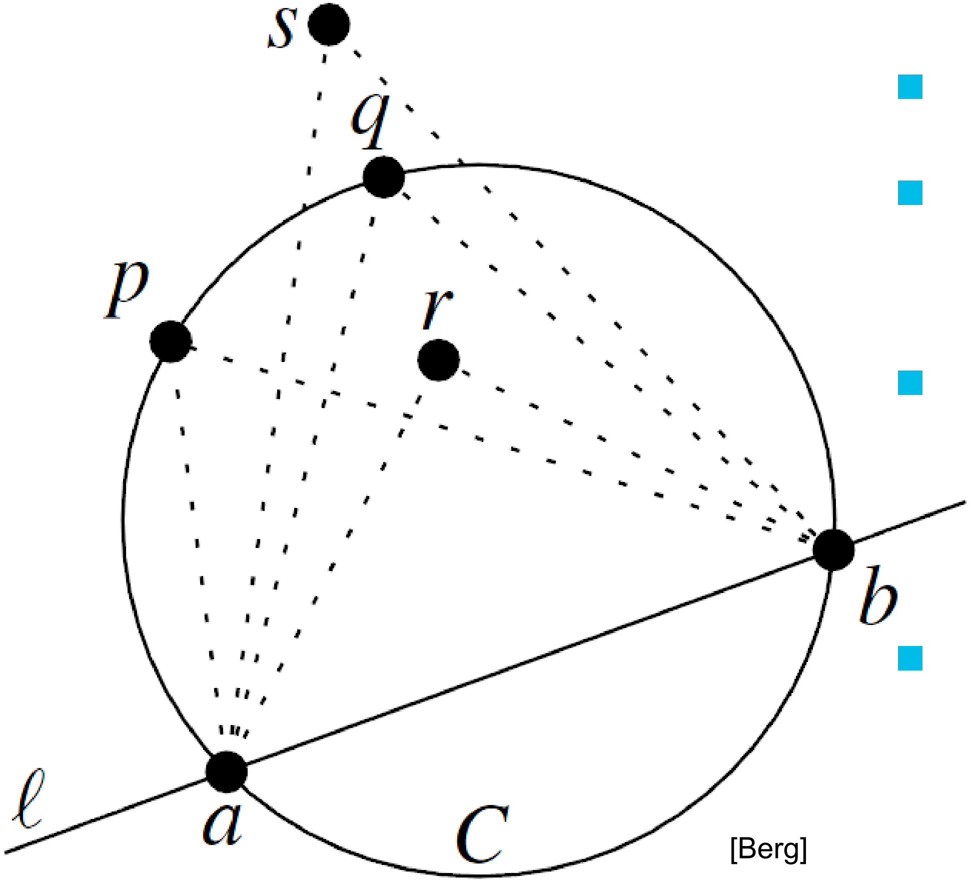
Delaunay triangulation

- 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.
  - does not necessarily minimize the length of the edges.

Thales's theorem (624-546 BC)



Respective Central Angle Theorem



[Berg]

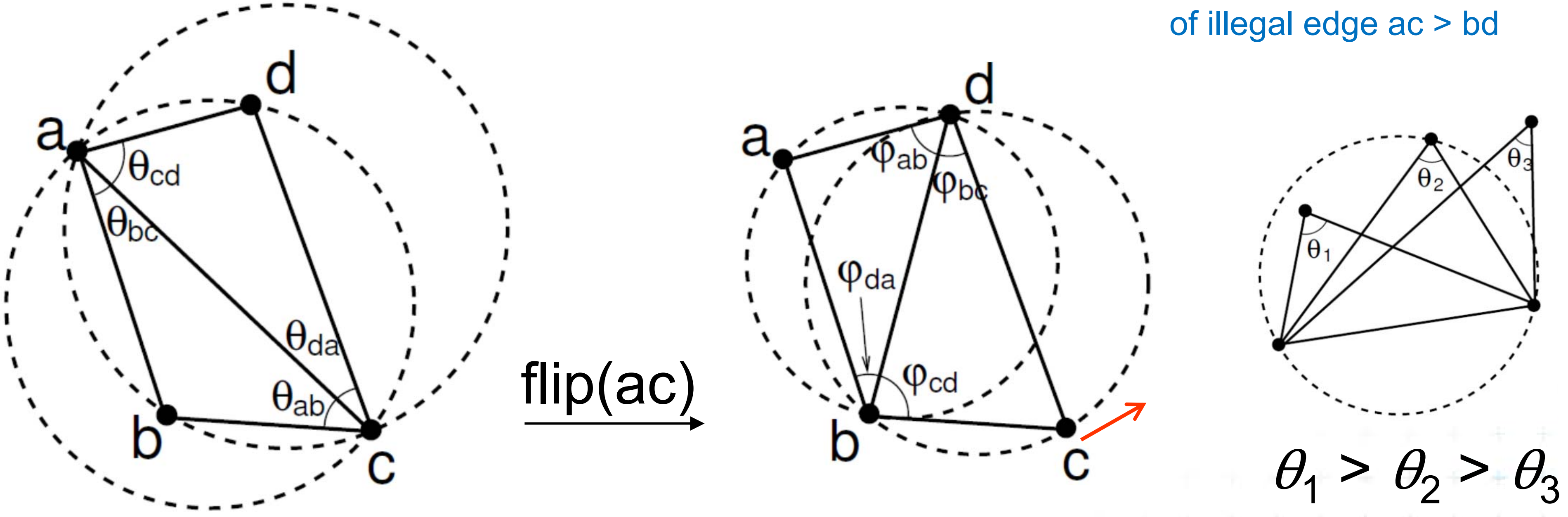
- Let C = circle,
- l =line intersecting C in points a, b
- p, q, r, s = points on the same side of l
  p,q on C , r is *in,* s is out
- Then for the angles holds:
  $\sphericalangle arb > \sphericalangle apb = \sphericalangle aqb > \sphericalangle asb$

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]

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

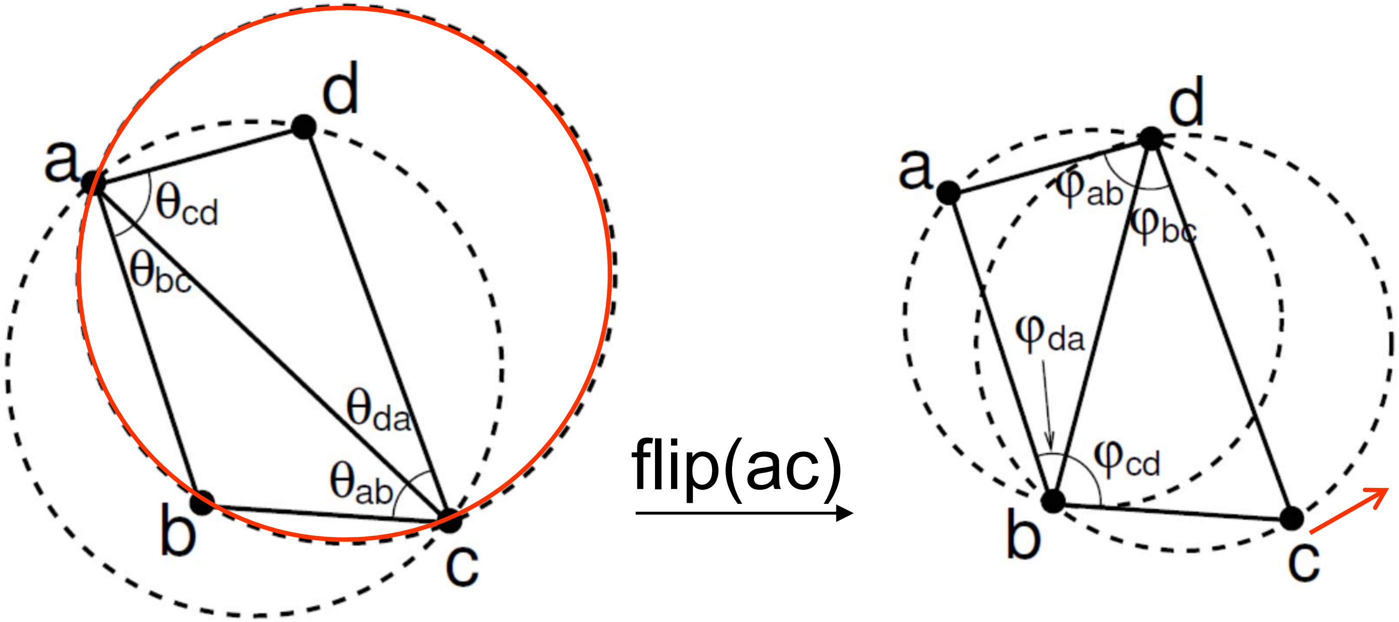
=> After limited number of edge flips

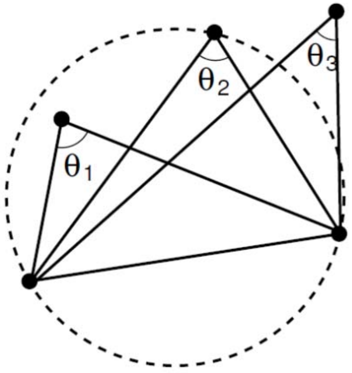
- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

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]

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

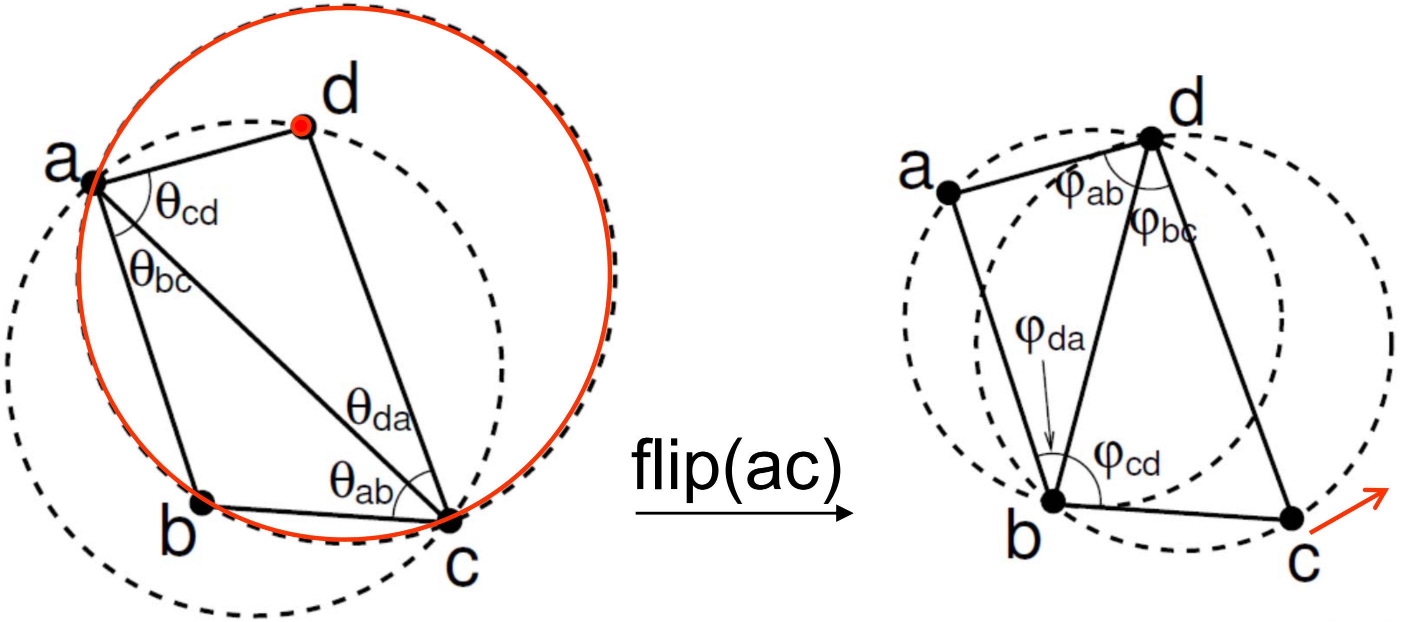
=> After limited number of edge flips

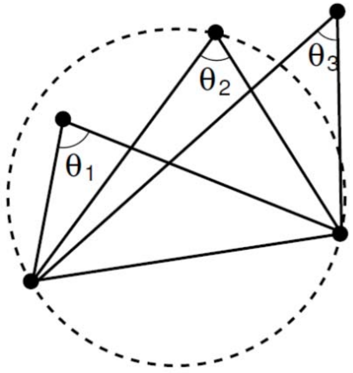
- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

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]

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

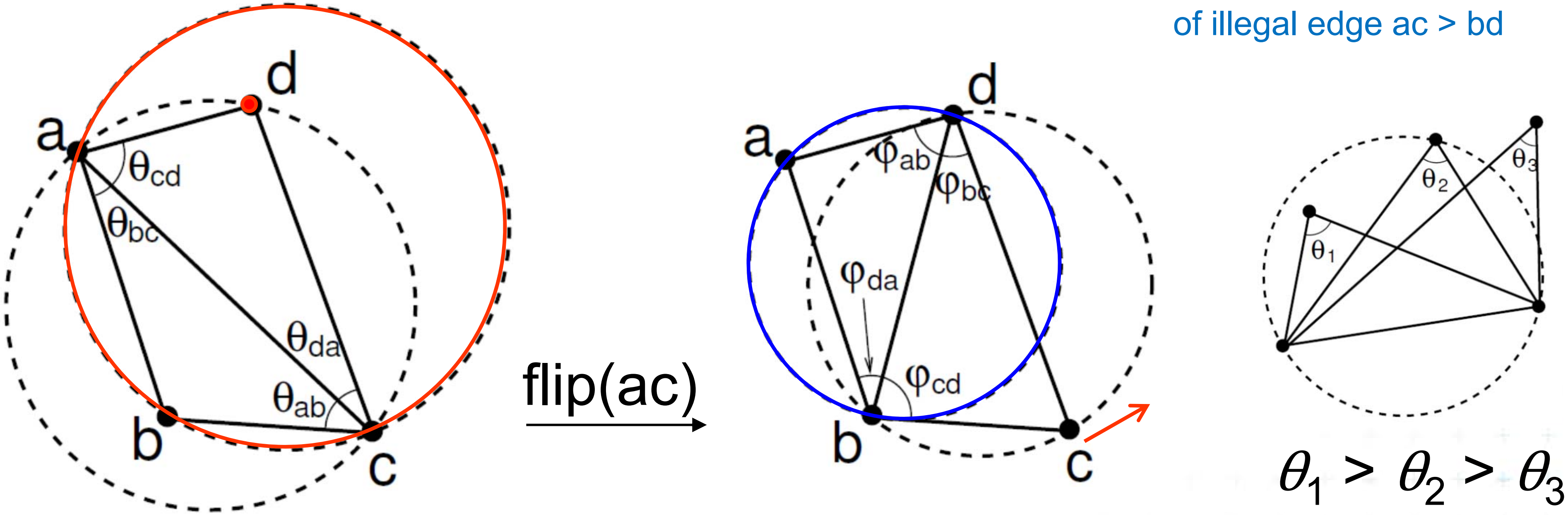
=> After limited number of edge flips

- 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]

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

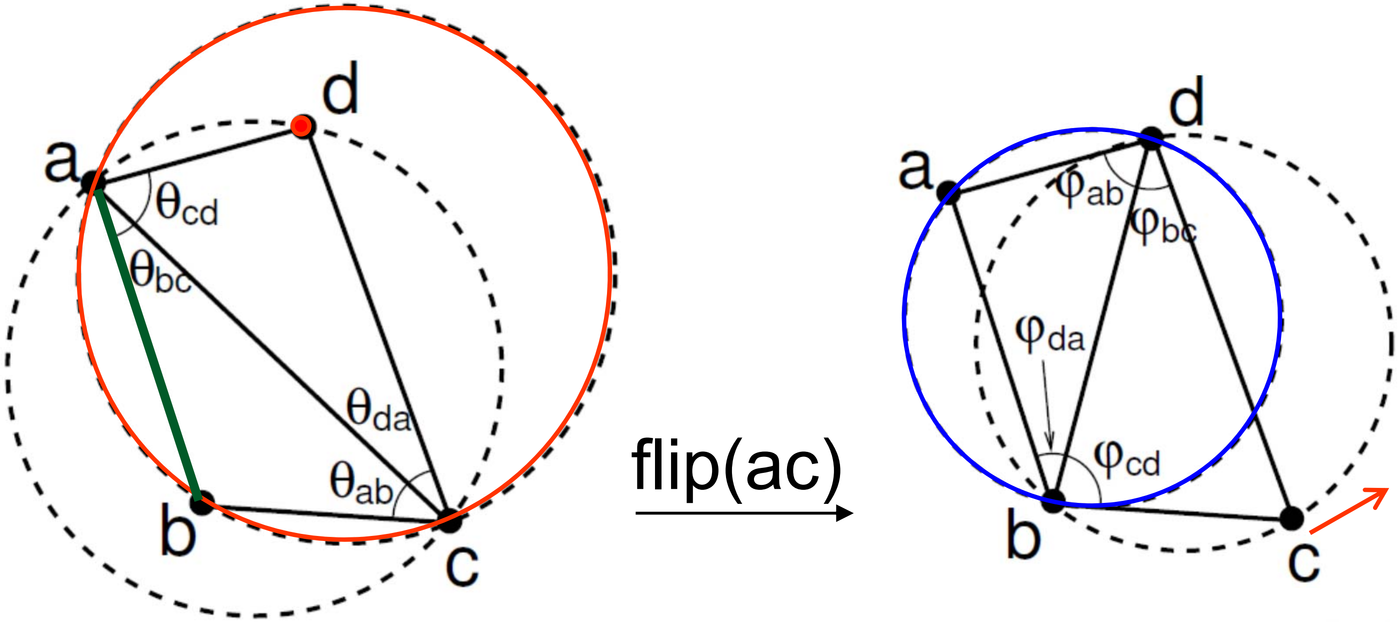
=> After limited number of edge flips

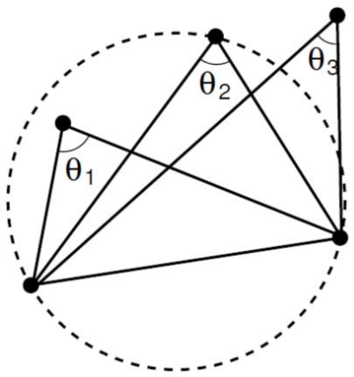
- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T

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]

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

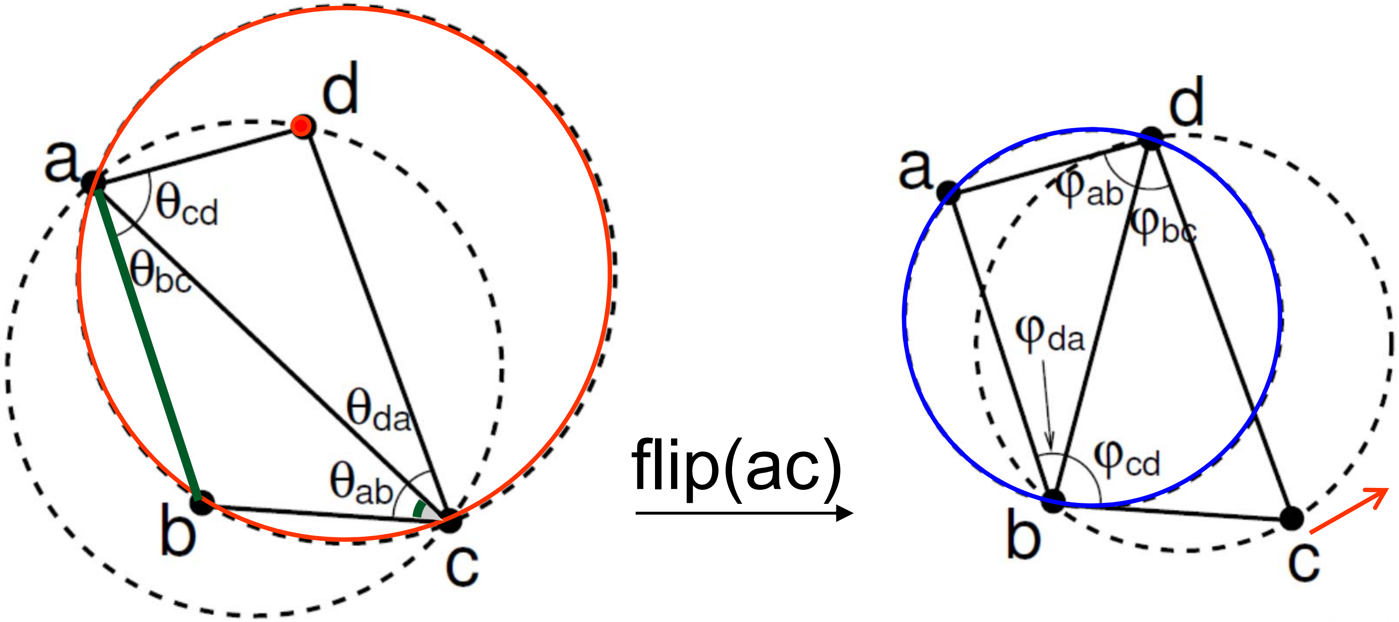
=> After limited number of edge flips

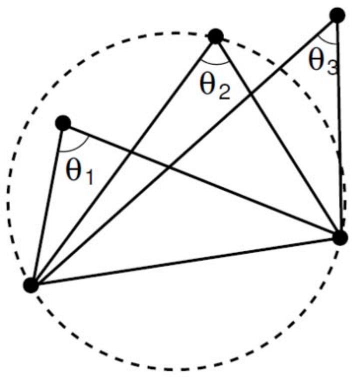
- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

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]

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

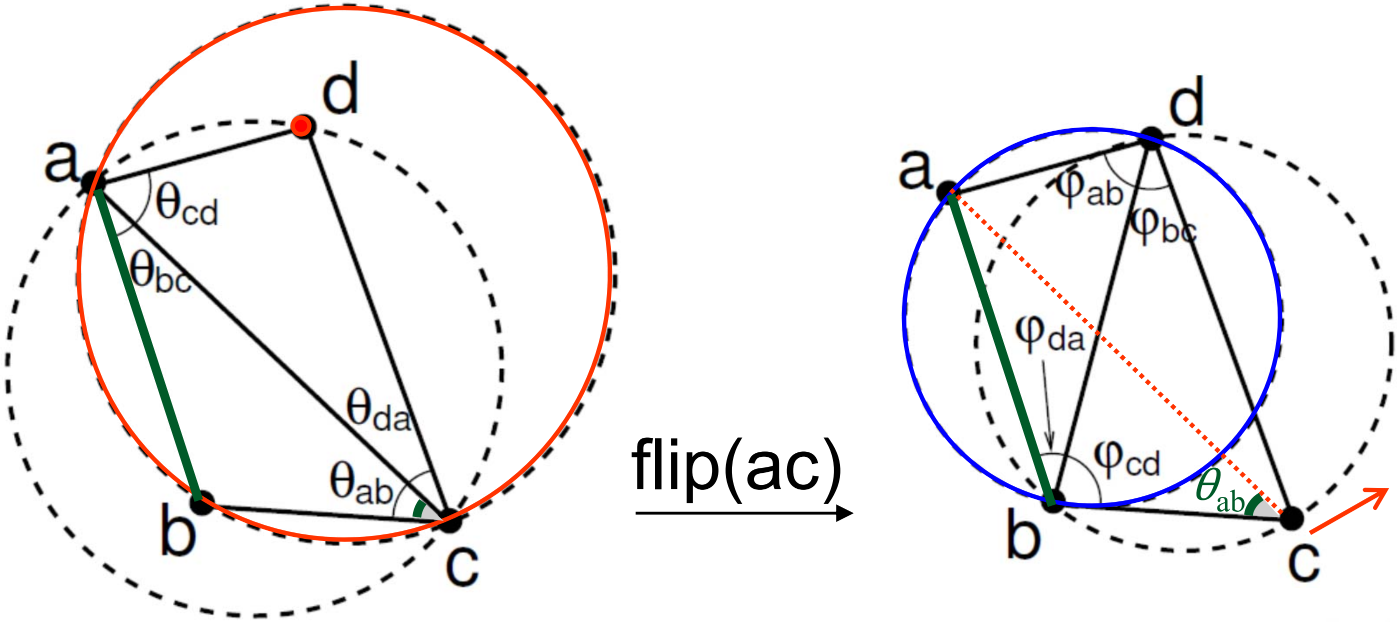
=> After limited number of edge flips

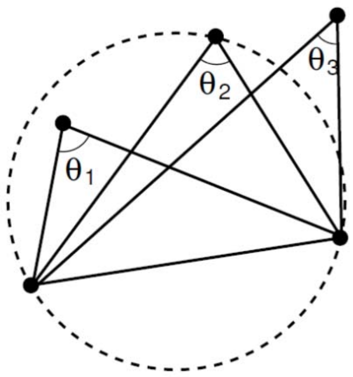
- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T

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]

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

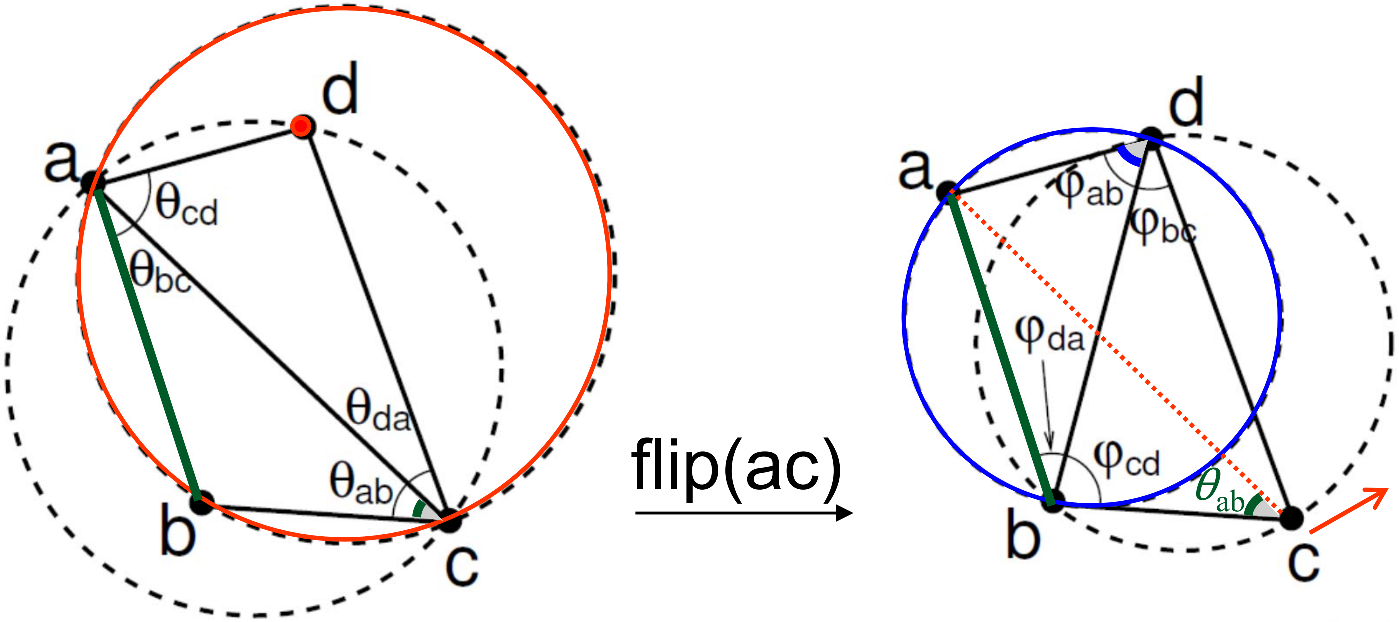
=> After limited number of edge flips

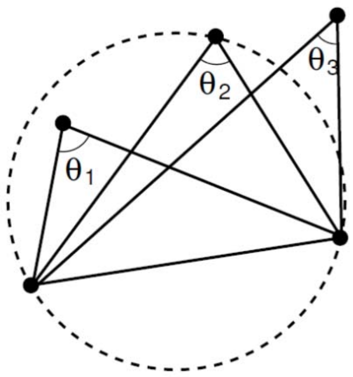
- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T

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]

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

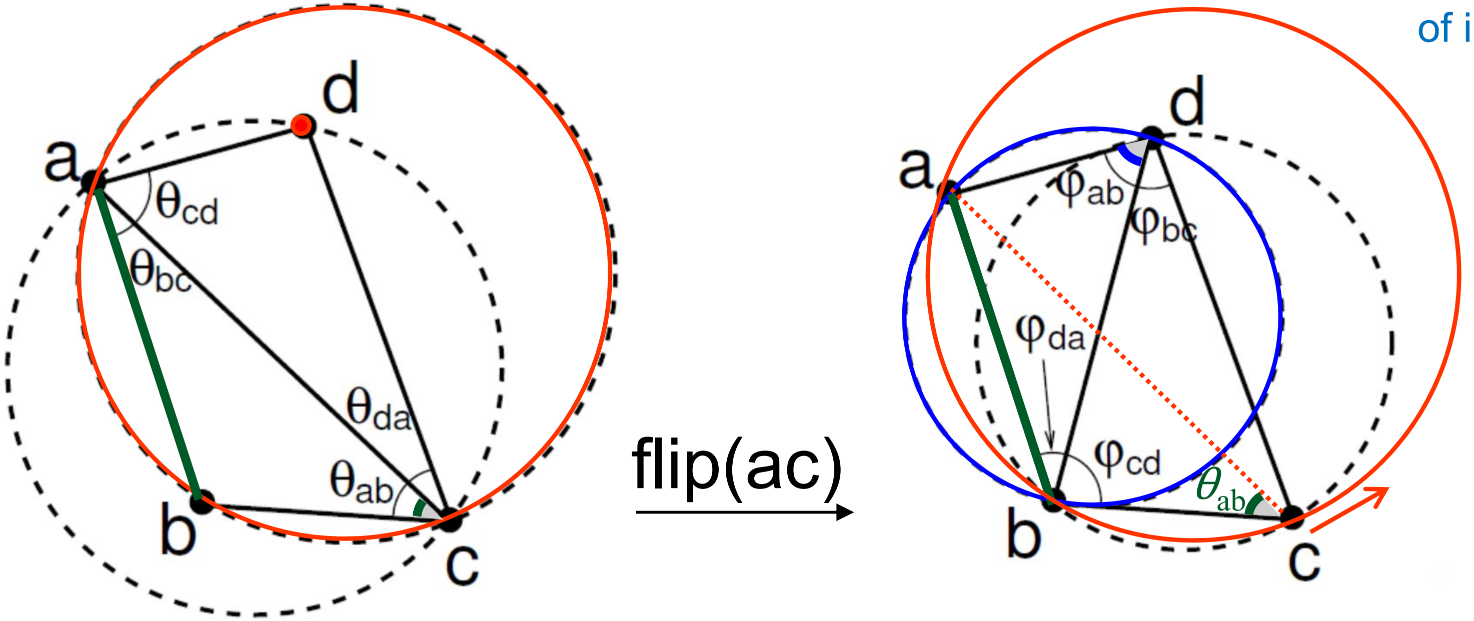
=> After limited number of edge flips

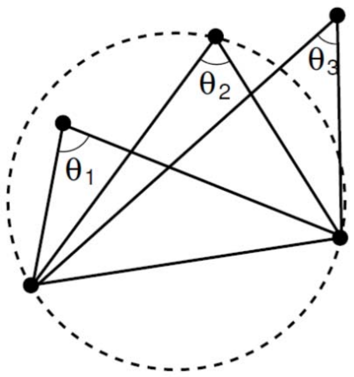
- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

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]

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

=> After limited number of edge flips

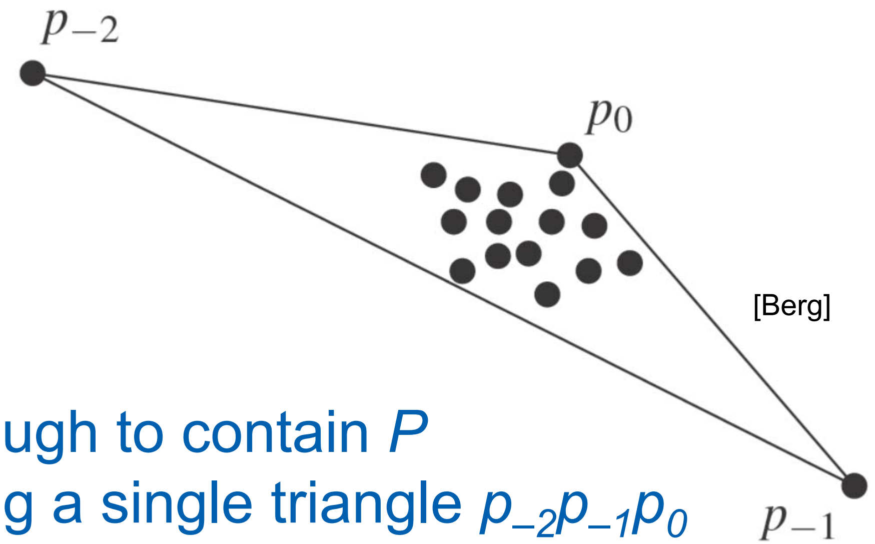
- Terminate with lexicographically maximum triangulation
- It satisfies the empty circle condition => Delauney T.

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

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

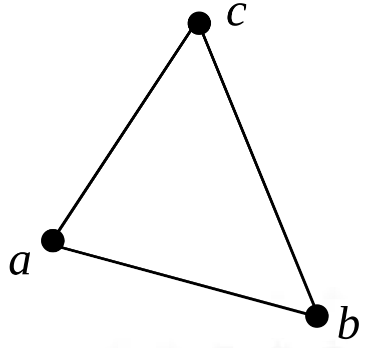


[Berg]

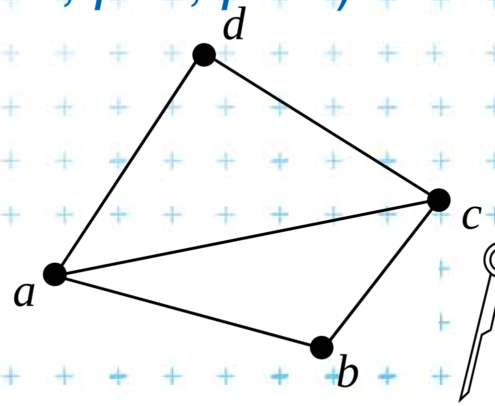
```
1. Let $p_{-2}, p_{-1}, p_0$ form a triangle large enough to contain $P$
2. Initialize $T$ as the triangulation consisting a single triangle $p_{-2}p_{-1}p_0$
3. Compute random permutation $p_1, p_2, \ldots, p_n$ of $P \setminus \{p_0\}$
4. for $r = 1$ to $n$ do
5.     $T$ = Insert( $p_r$ , $T$ )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



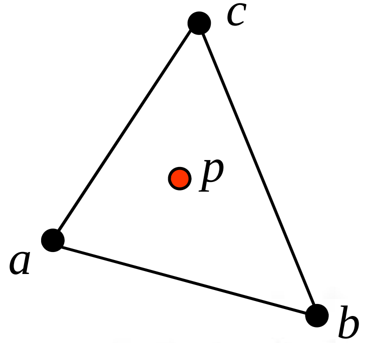
[Berg]



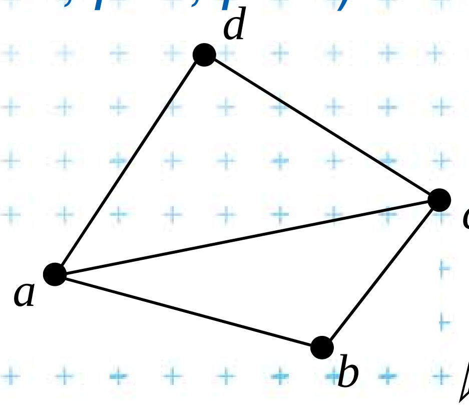
[Berg]

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 $abc \in T$ 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
```



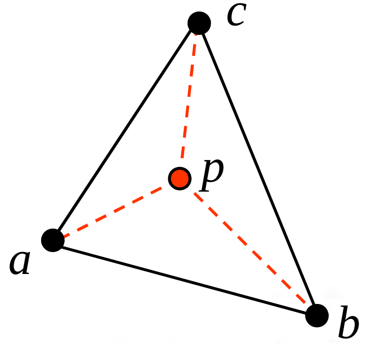
[Berg]



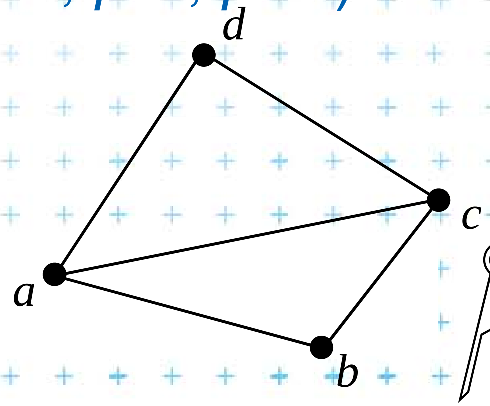
[Berg]

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
```



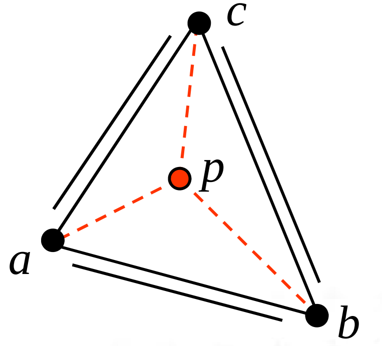
[Berg]



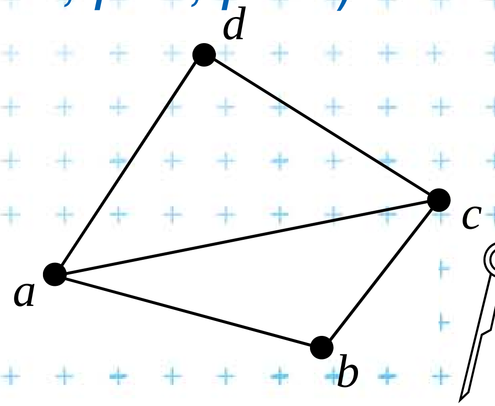
[Berg]

Incremental algorithm – 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
```



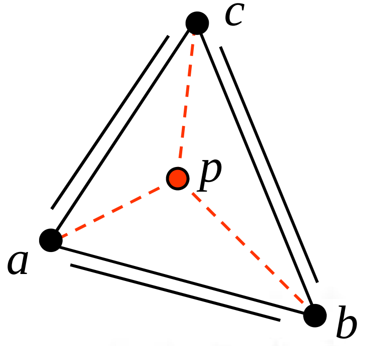
[Berg]



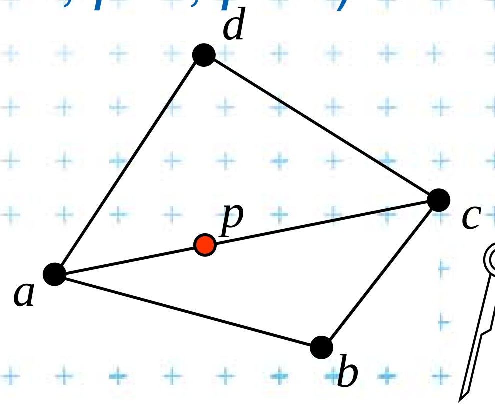
[Berg]

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
```



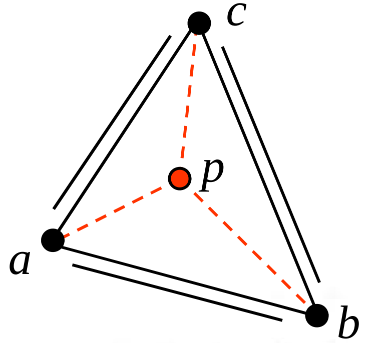
[Berg]



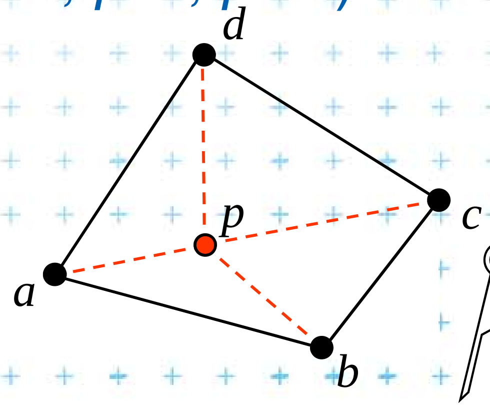
[Berg]

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 $T$
```



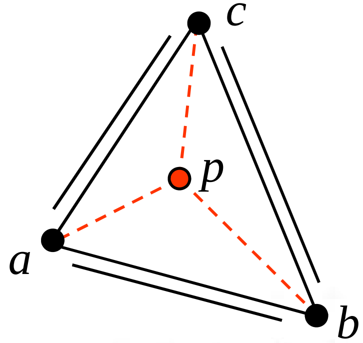
[Berg]



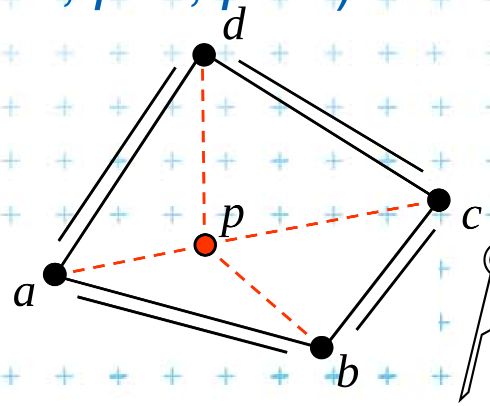
[Berg]

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 ∈ 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]

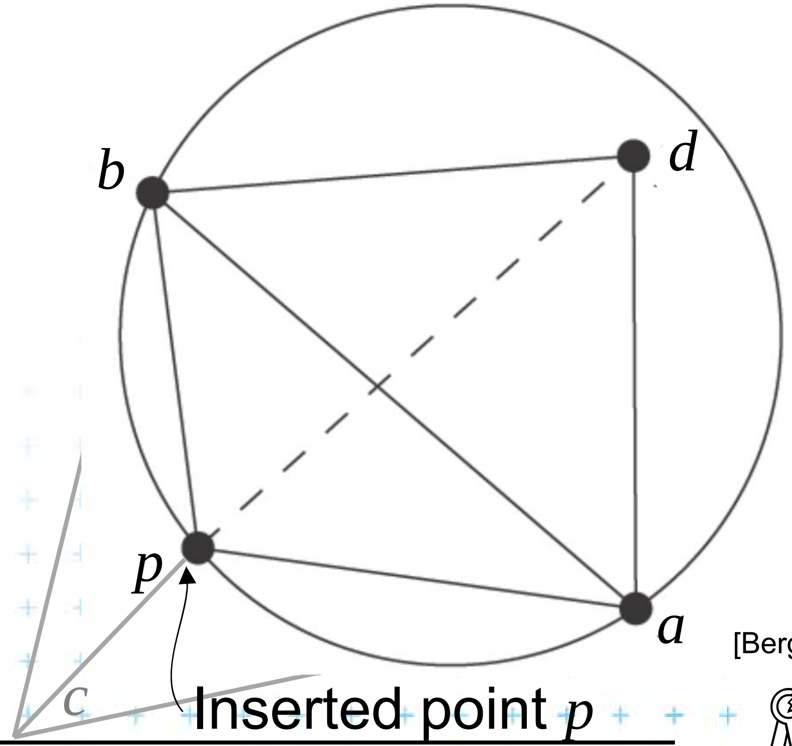
Incremental algorithm – edge legalization

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

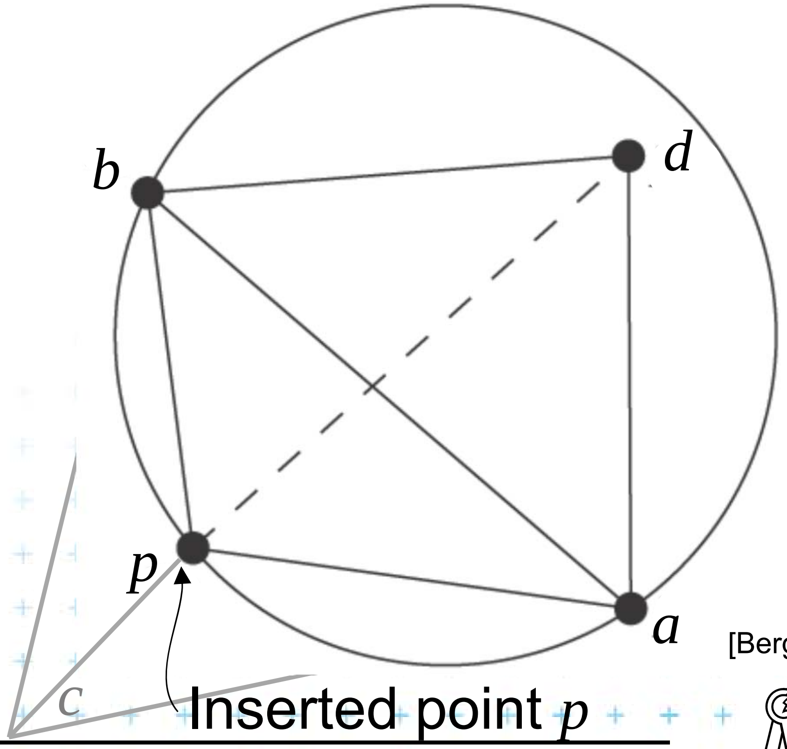
LegalizeEdge(*p, ab, T*)

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

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

```
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*)



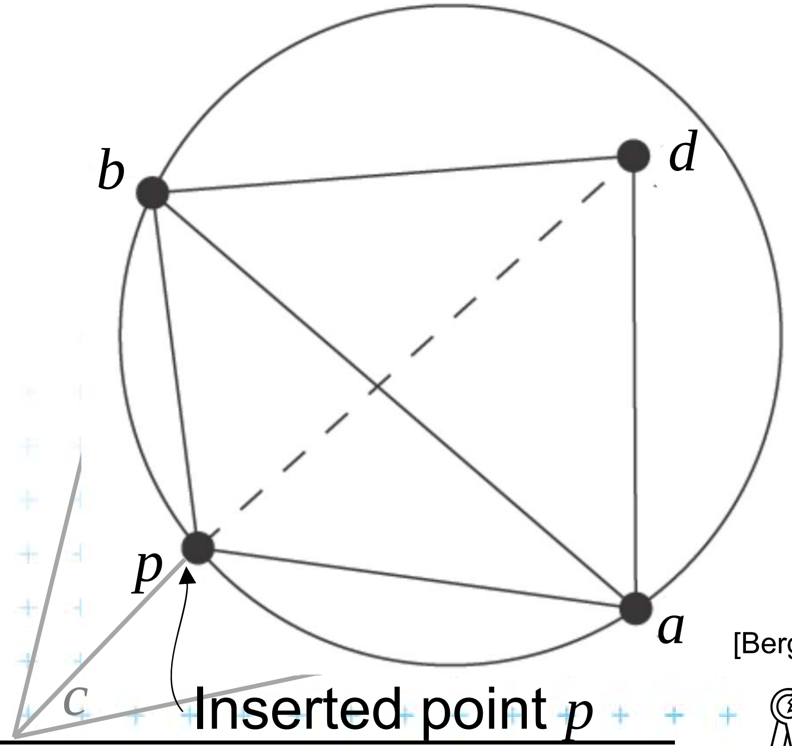
[Berg]

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)



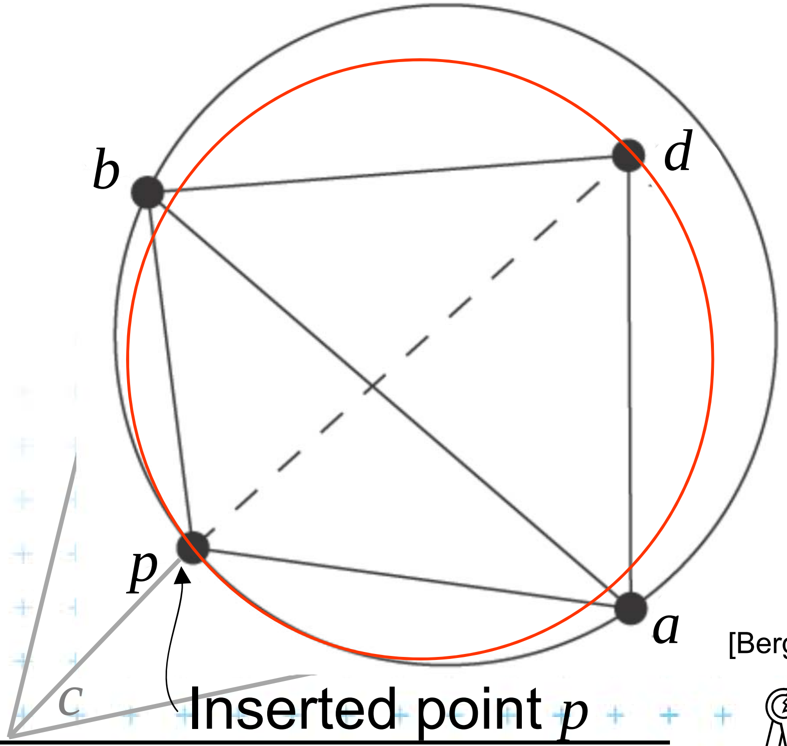
[Berg]

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 ∪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*

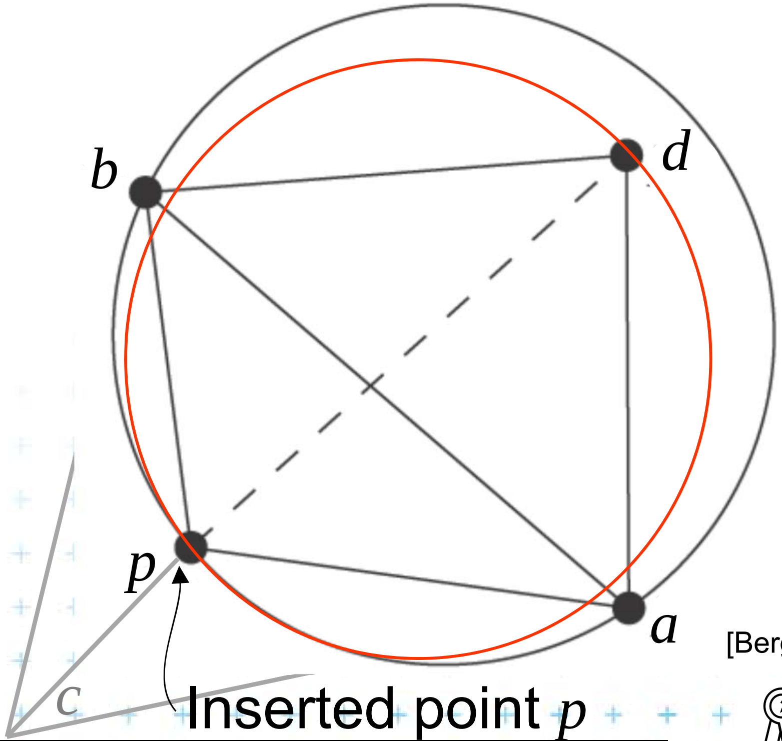
Output: Delaunay triangulation of *p* ∪*T*

```
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*



[Berg]

Incremental algorithm – edge legalization

LegalizeEdge(*p, ab, T*)

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

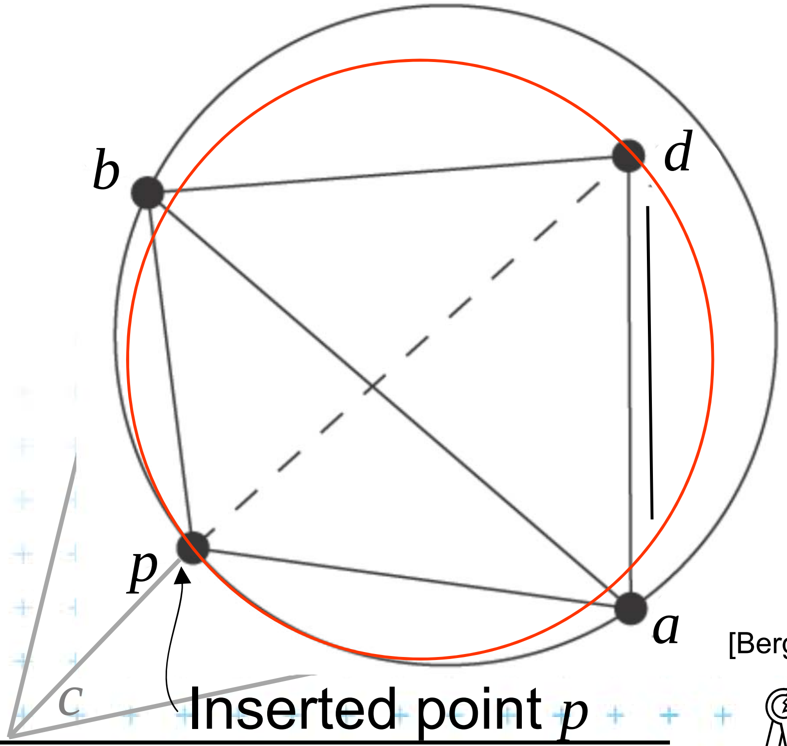
Output: Delaunay triangulation of *p* ∪*T*

```
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*



[Berg]

Incremental algorithm – edge legalization

LegalizeEdge(*p, ab, T*)

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

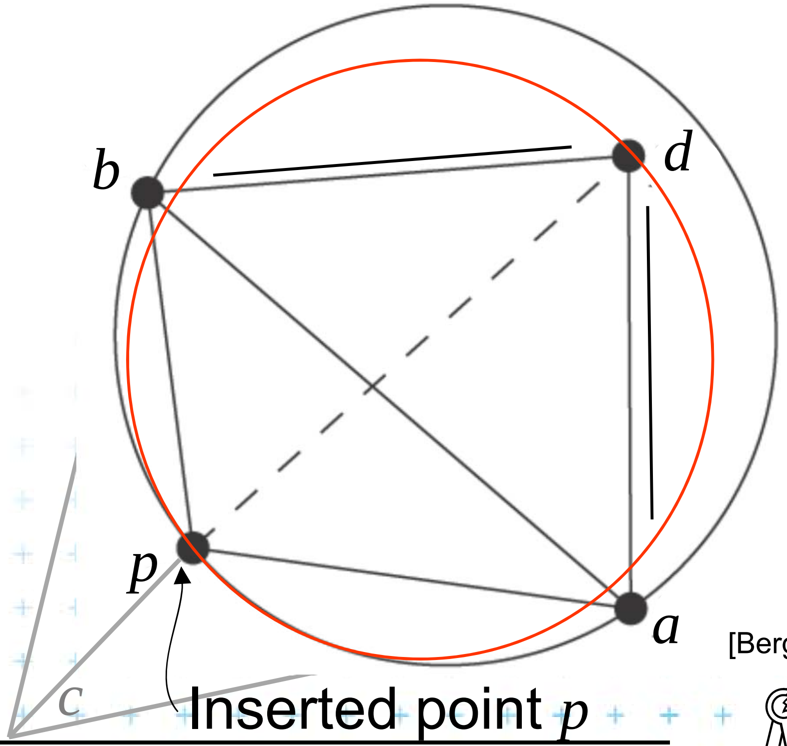
Output: Delaunay triangulation of *p* ∪*T*

```
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*



[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*

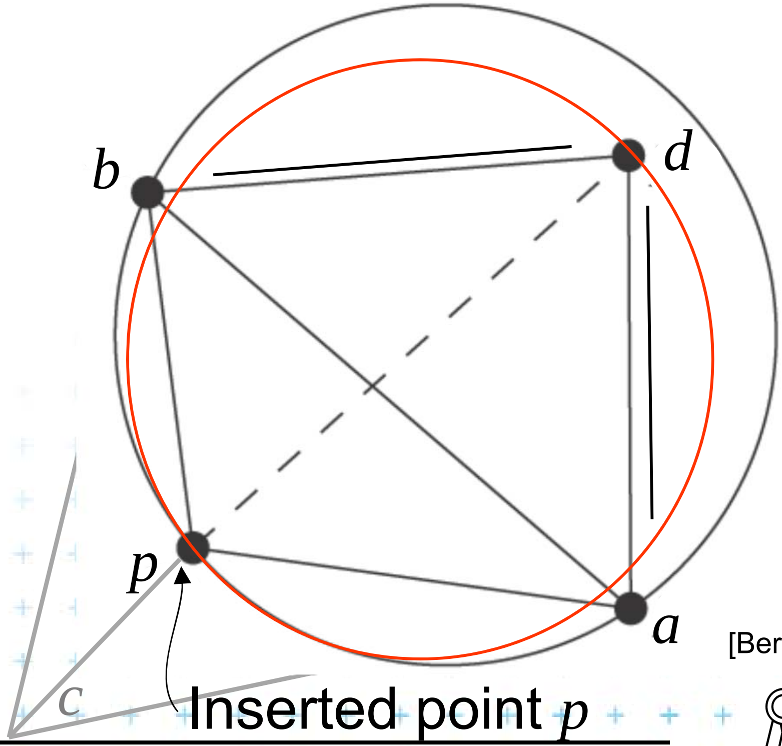
```
if( ab is edge on the exterior face ) return
let d be the vertex to the right of edge ab
if( inCircle( p, a, b, d ) )   // d is in the circle around pab => d is illegal
    Flip edge ab for pd
    LegalizeEdge( p, ad, T )
    LegalizeEdge( p, db, T )
```

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))

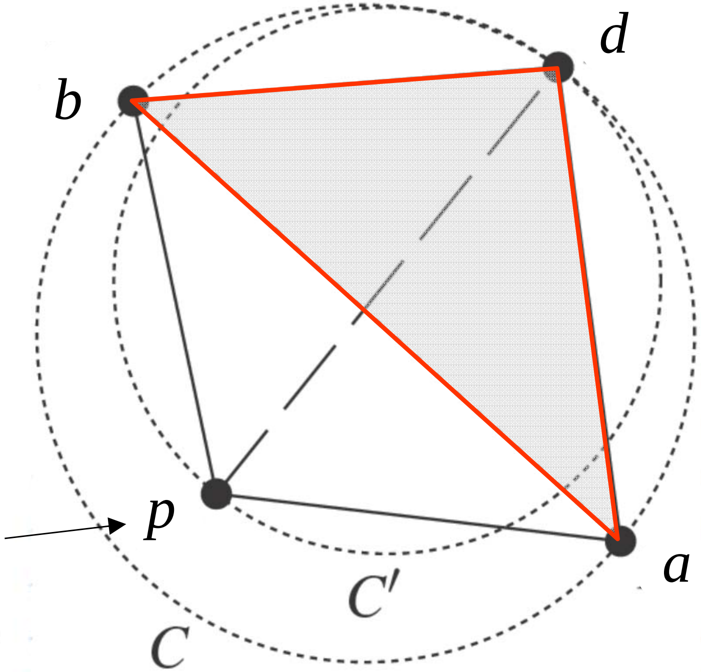


[Berg]

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



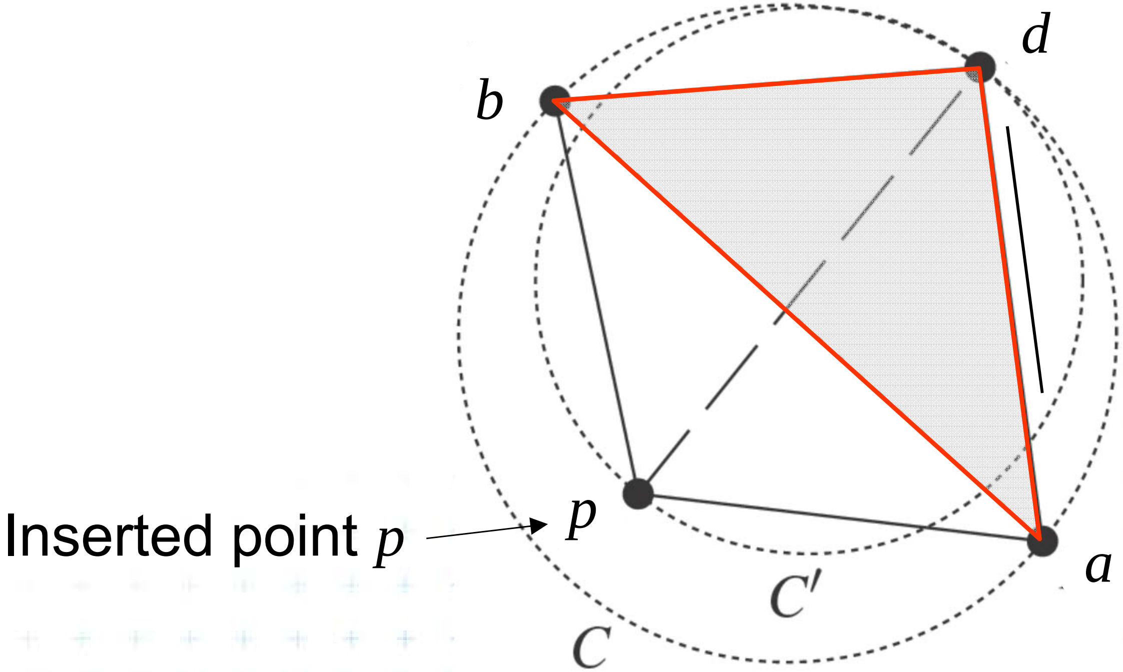
[Berg]

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



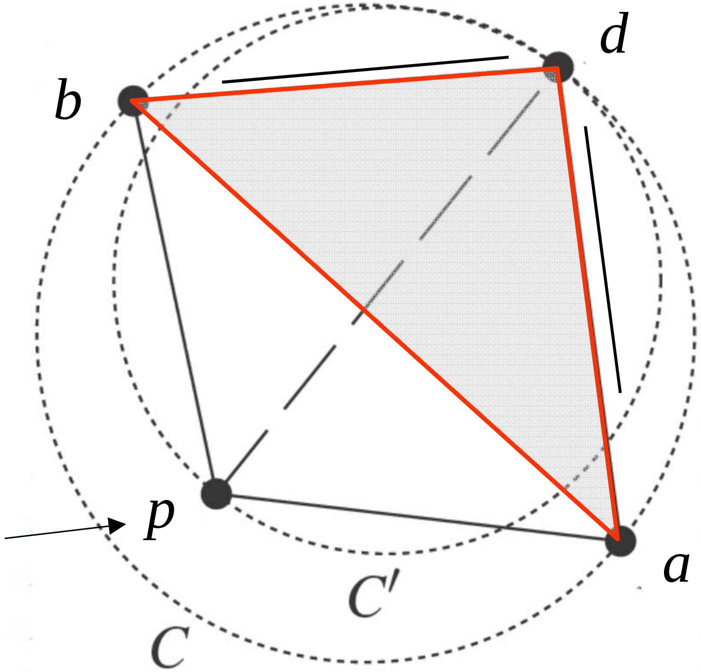
[Berg]

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 ($a, b \notin C$)

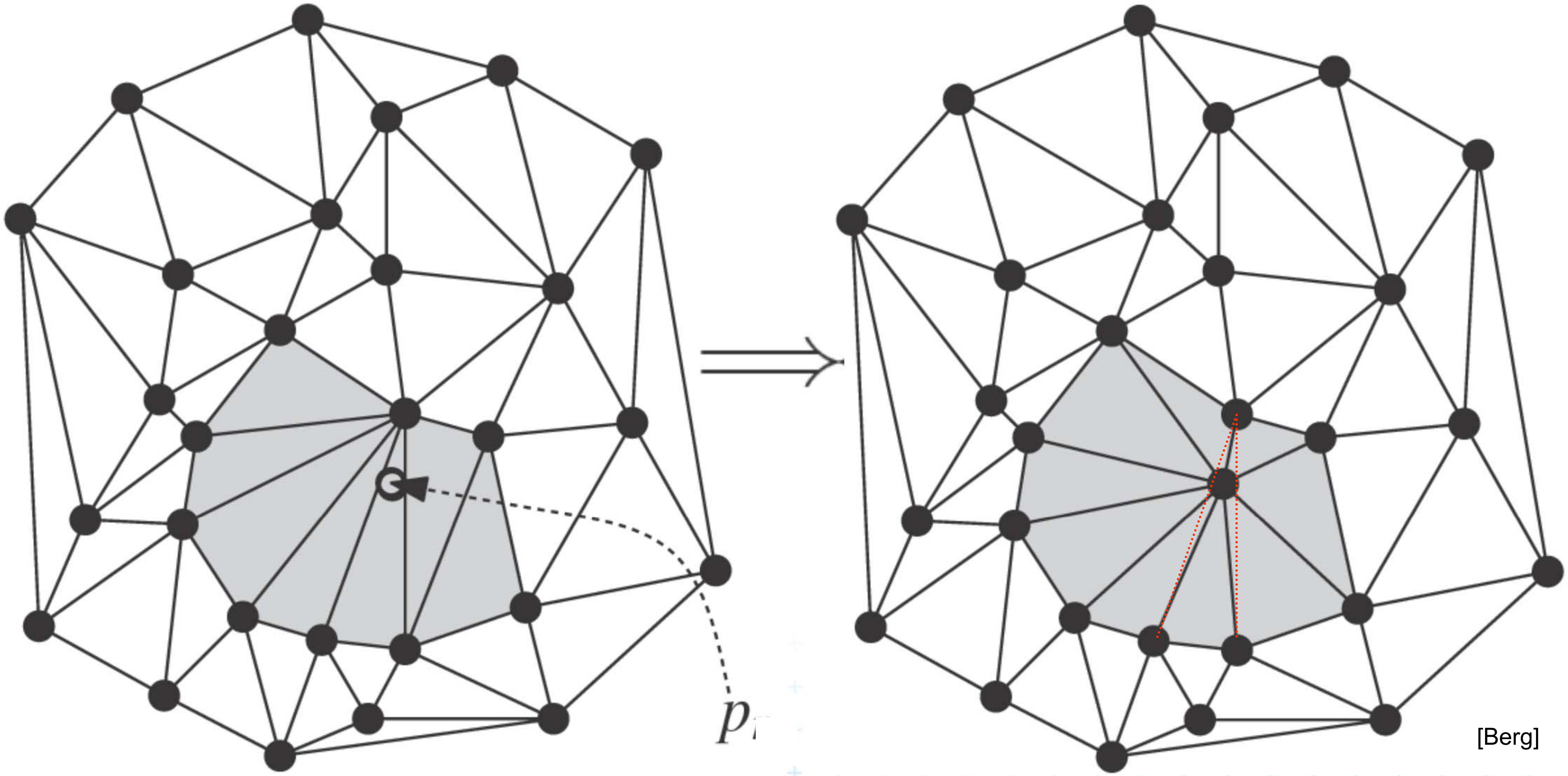
  => new edge *pd* is a Delaunay edge



Inserted point p

[Berg]

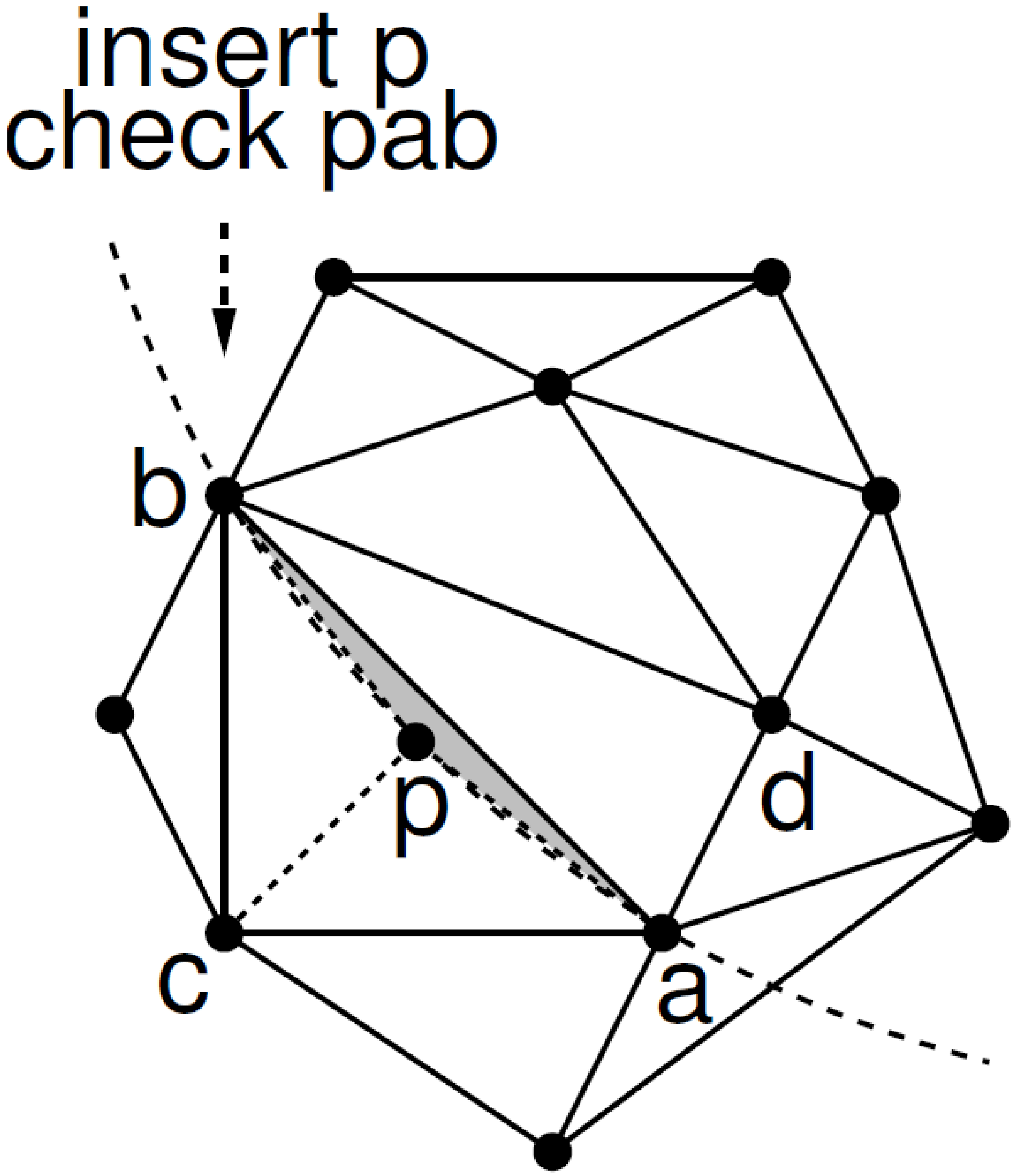
DT- point insert and mesh legalization



[Berg]

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

Delaunay triangulation – other point insert



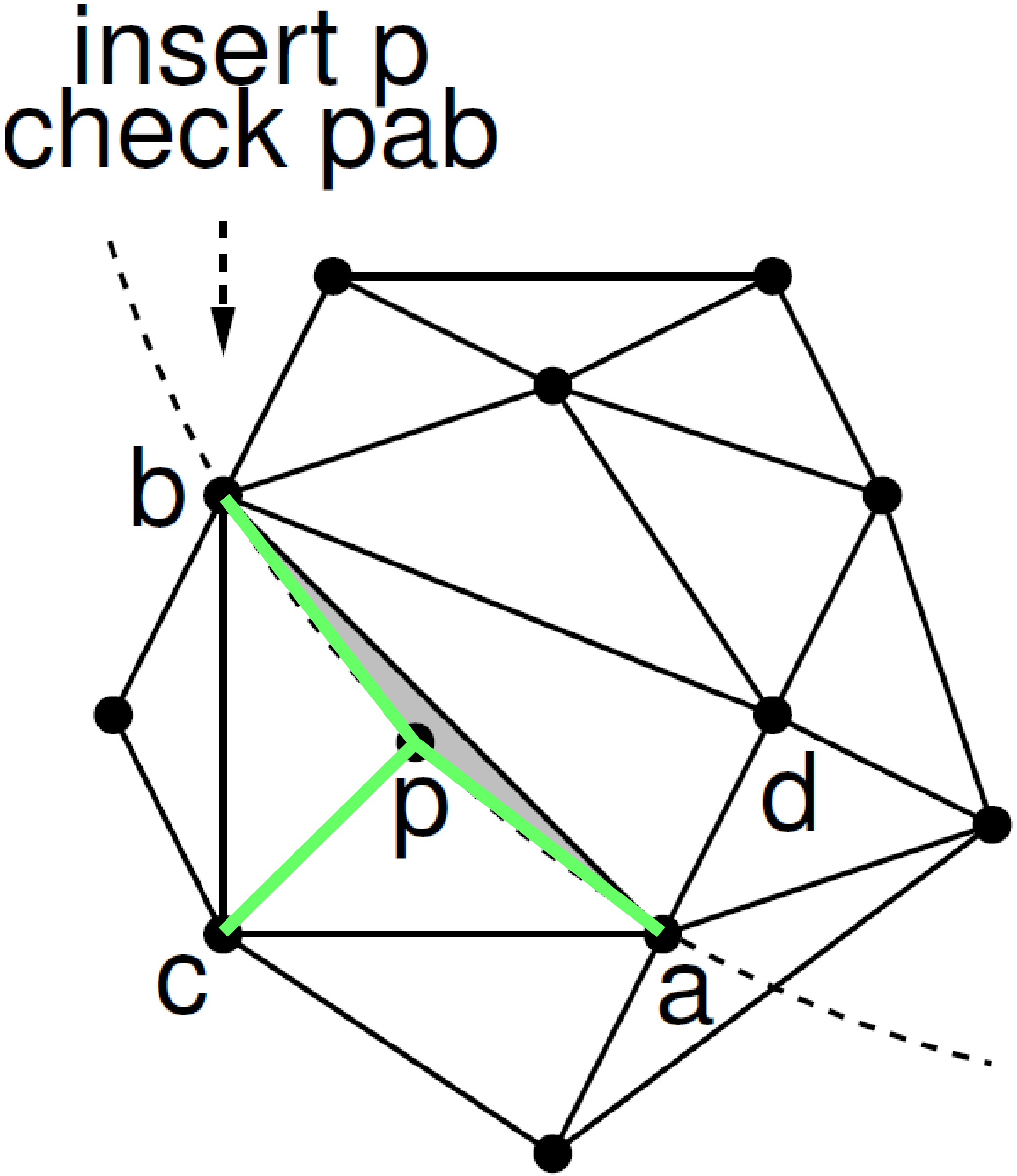
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



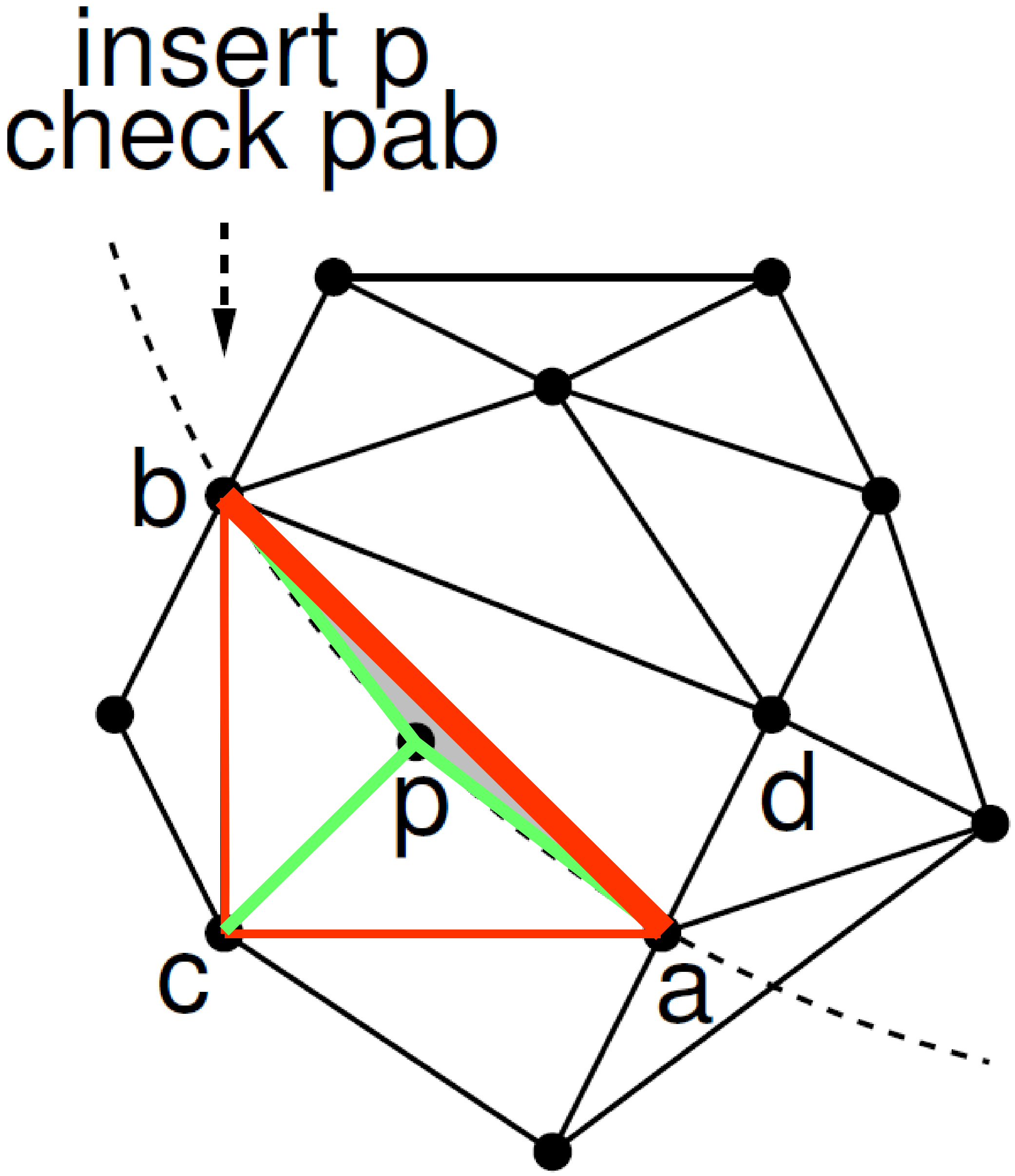
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



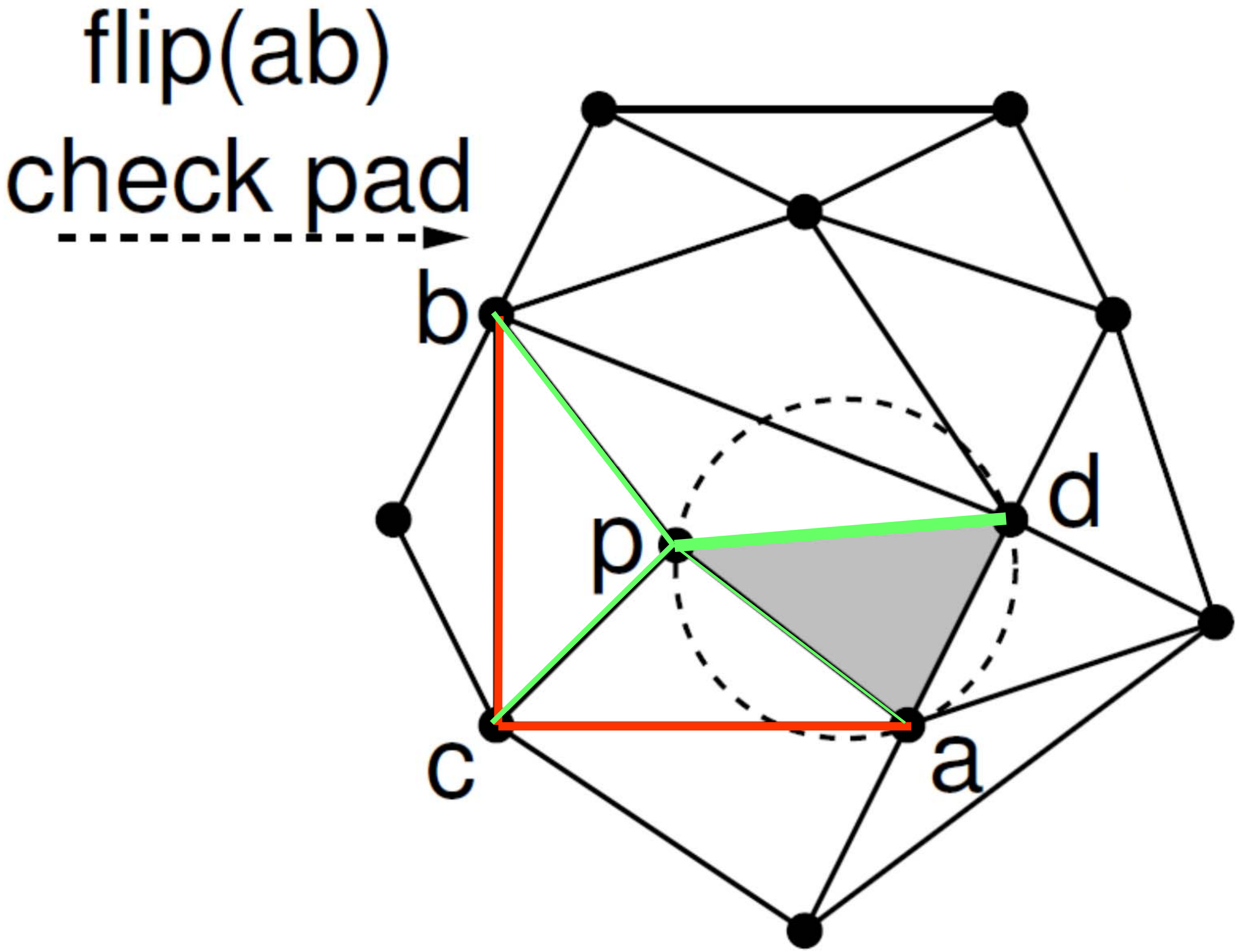
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



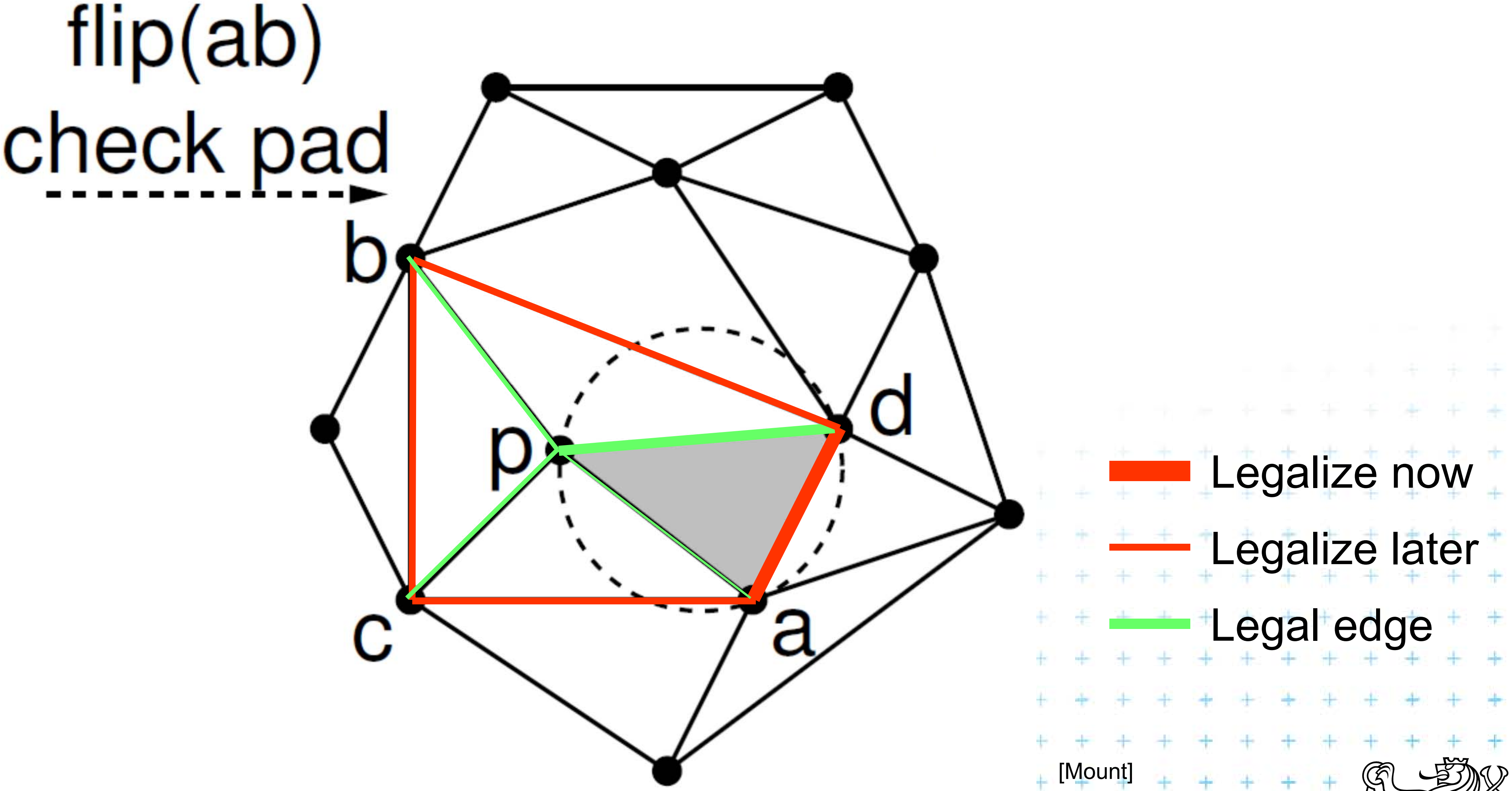
🟥 Legalize now

▬ Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



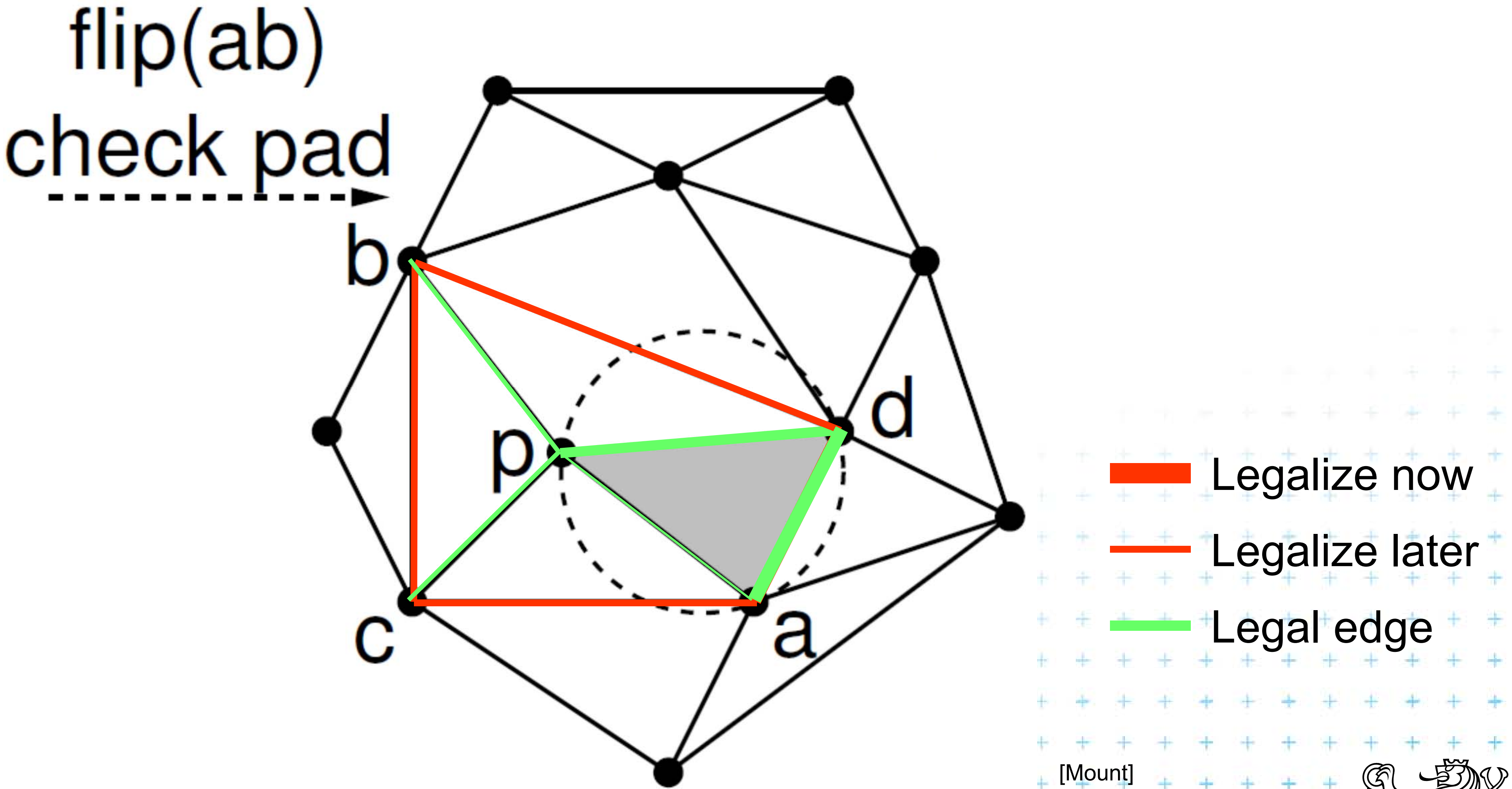
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

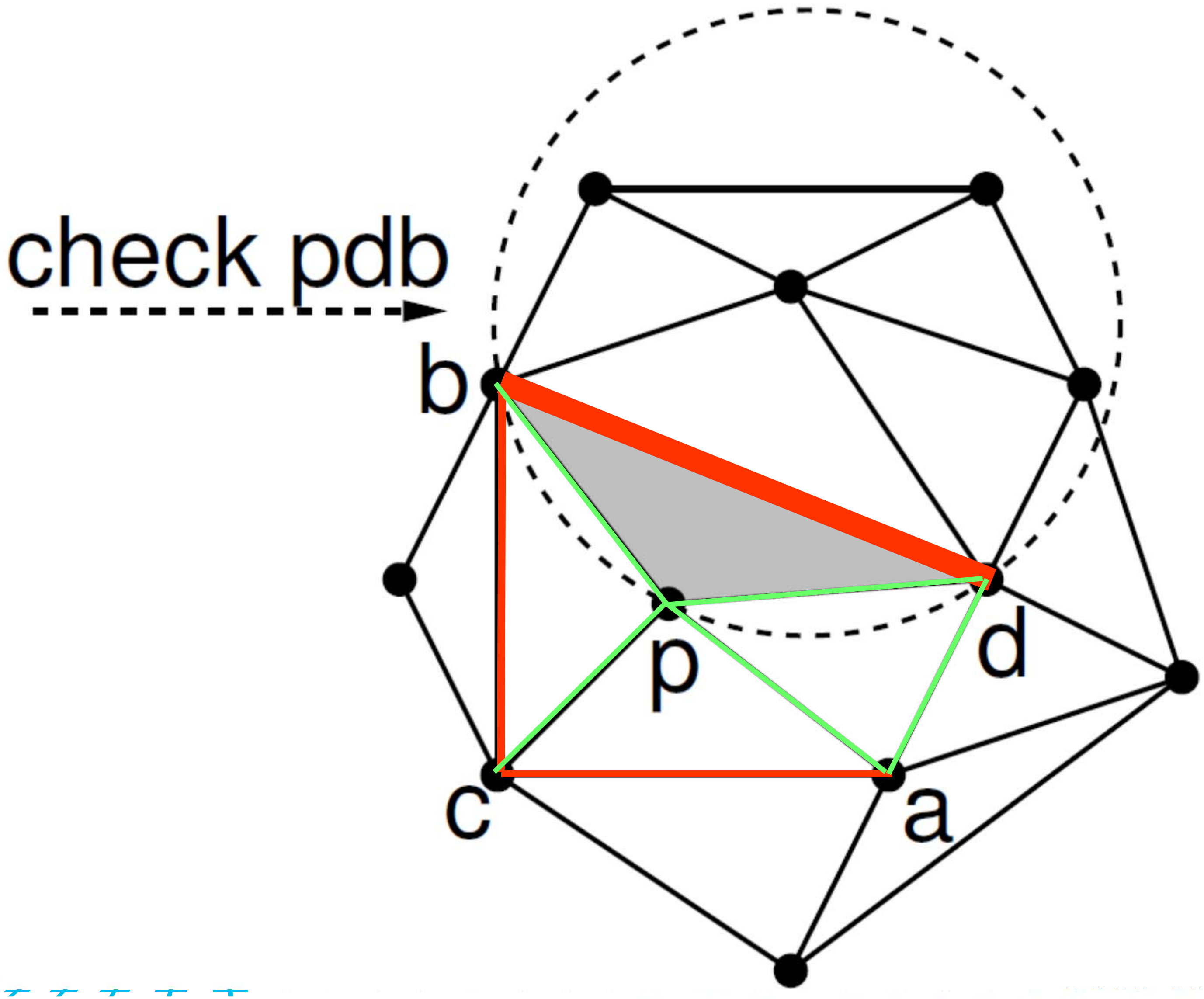
[Mount]

Delaunay triangulation – other point insert



[Mount]

Delaunay triangulation – other point insert



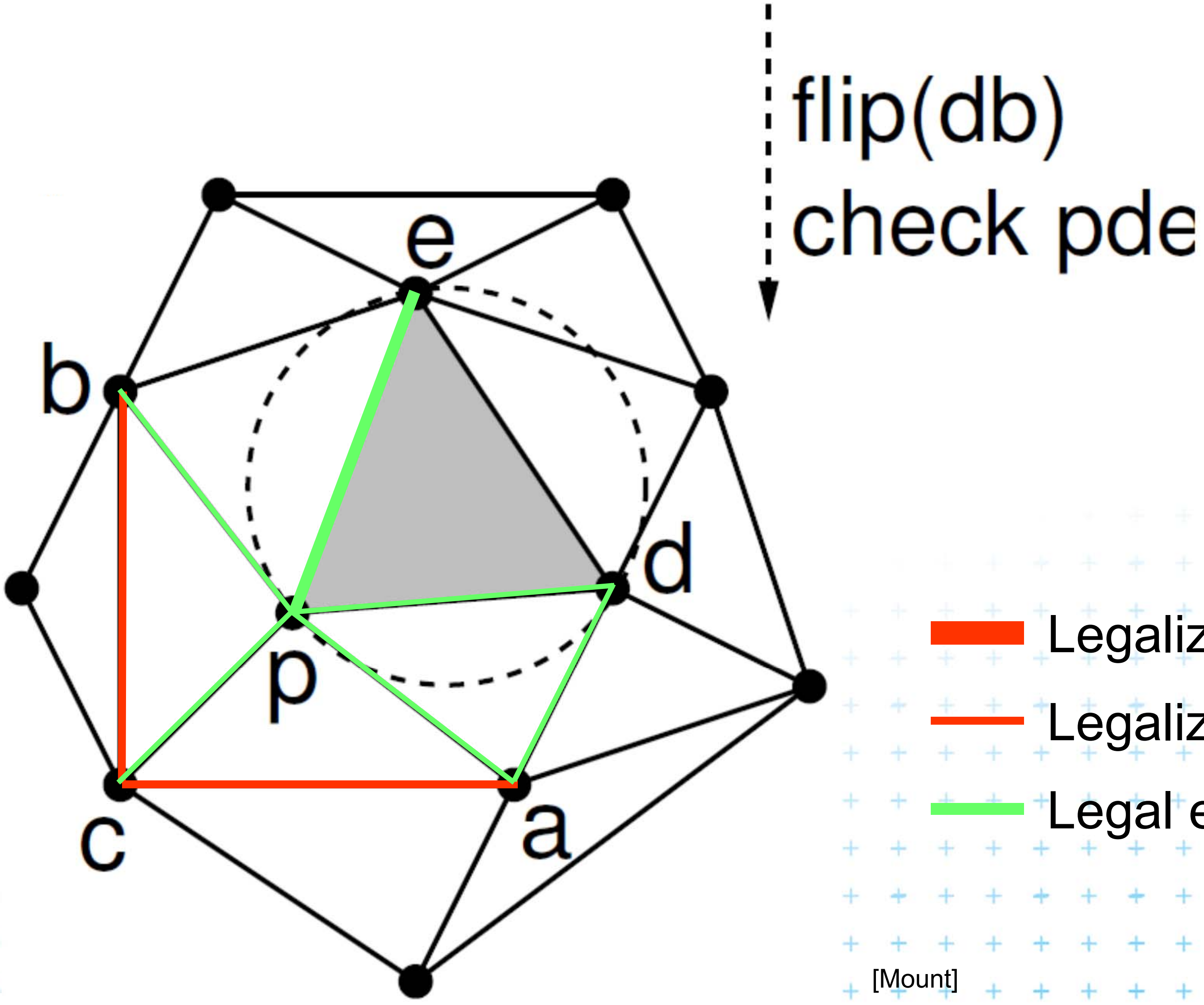
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



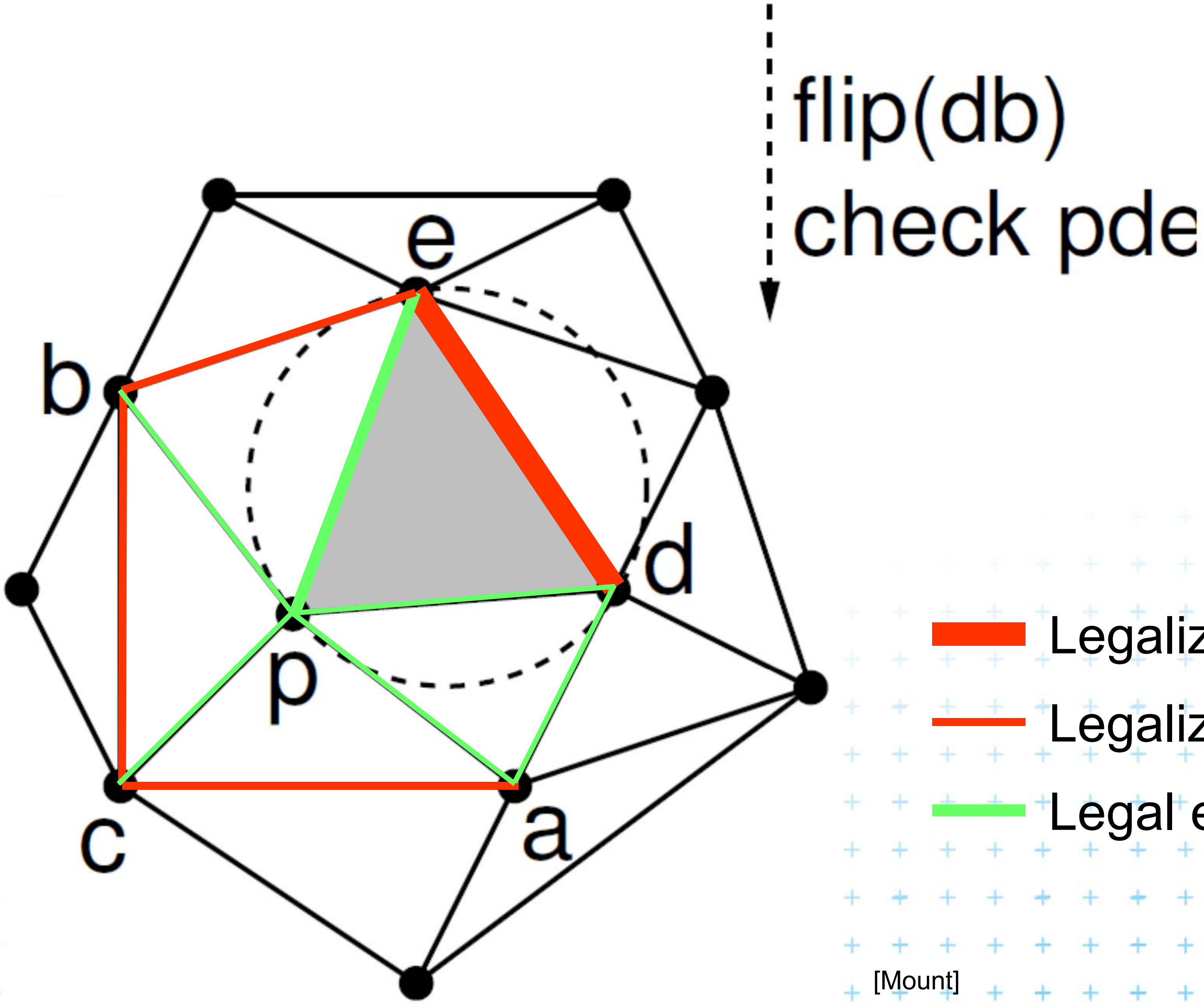
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



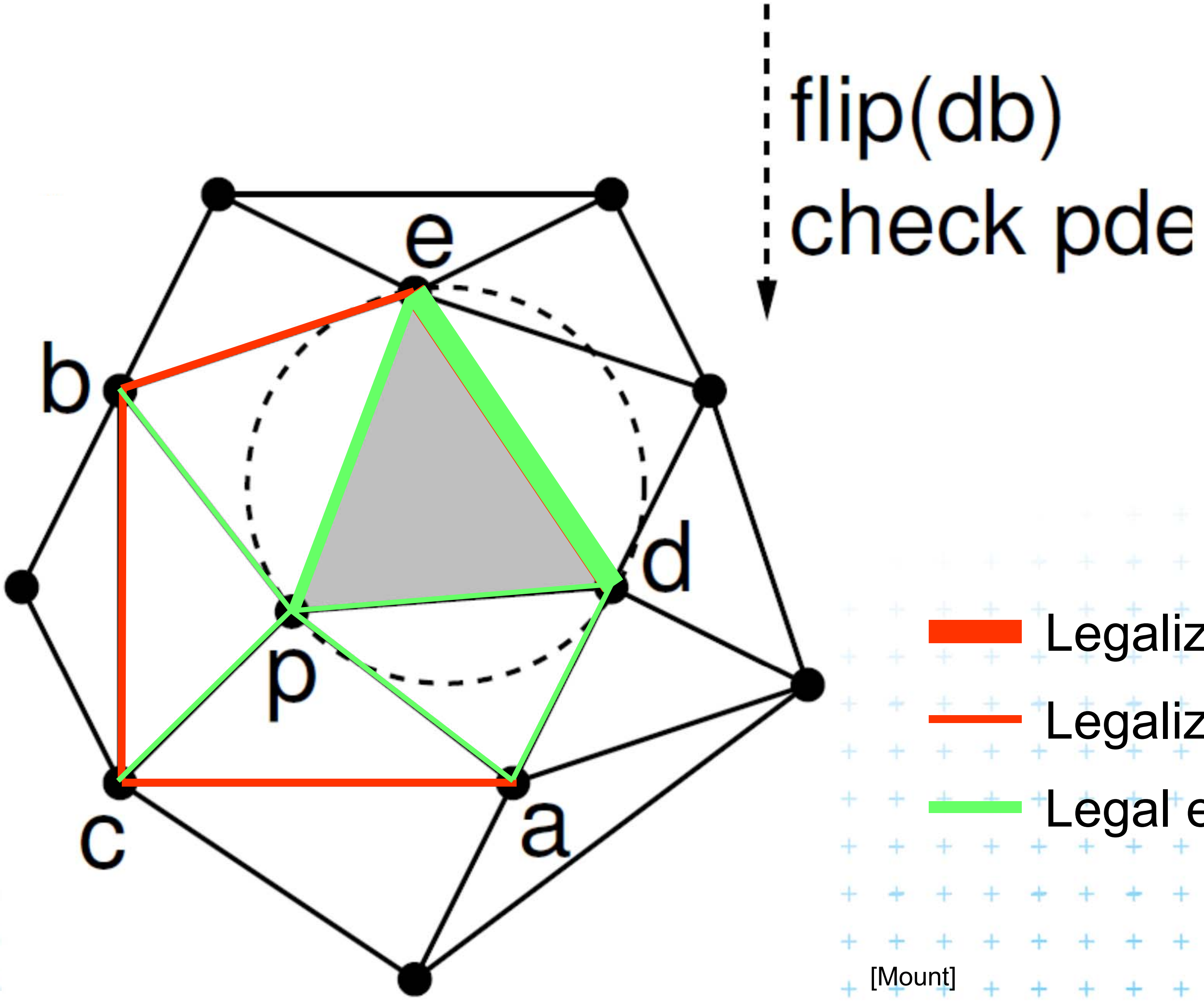
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



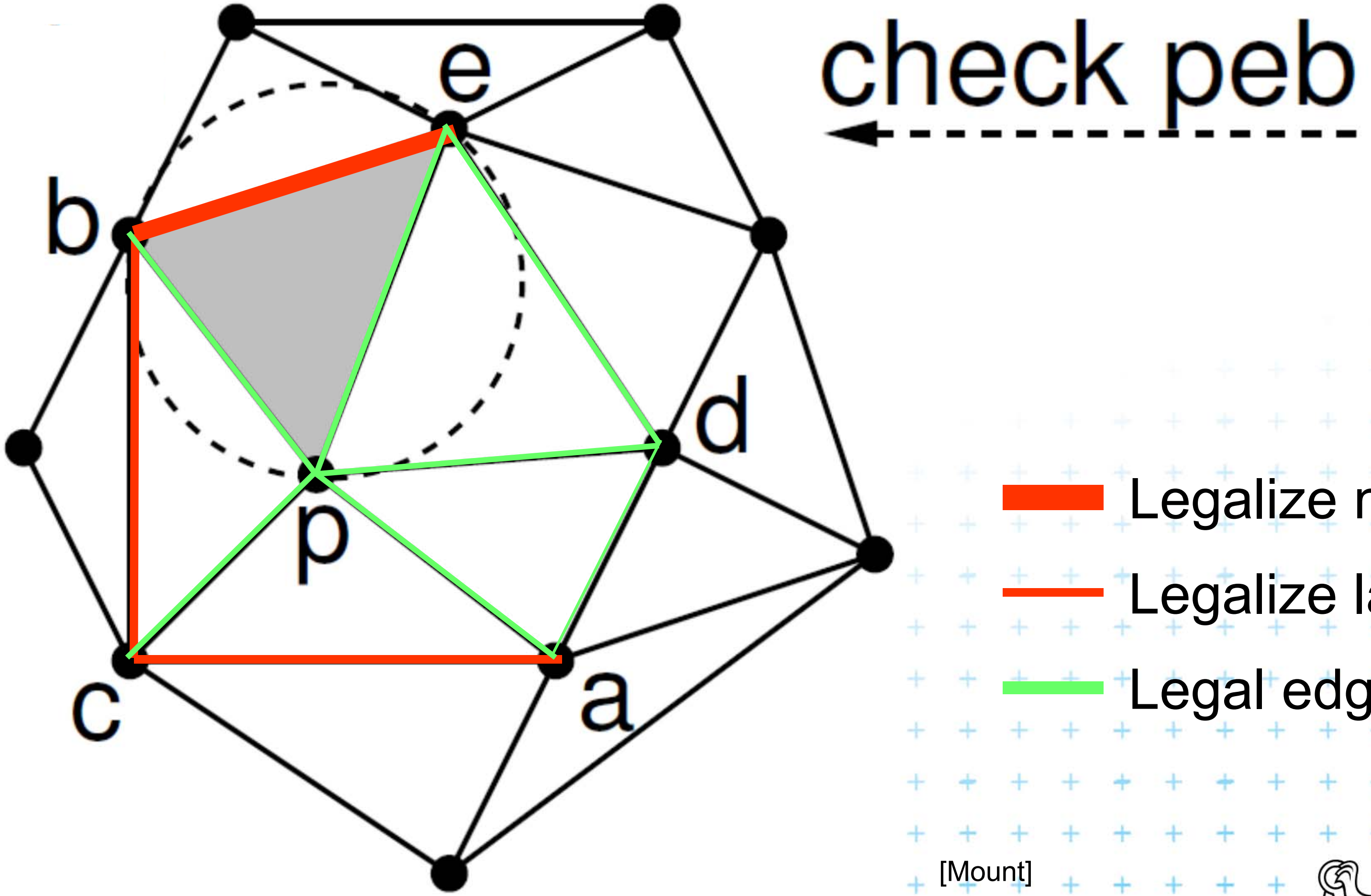
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



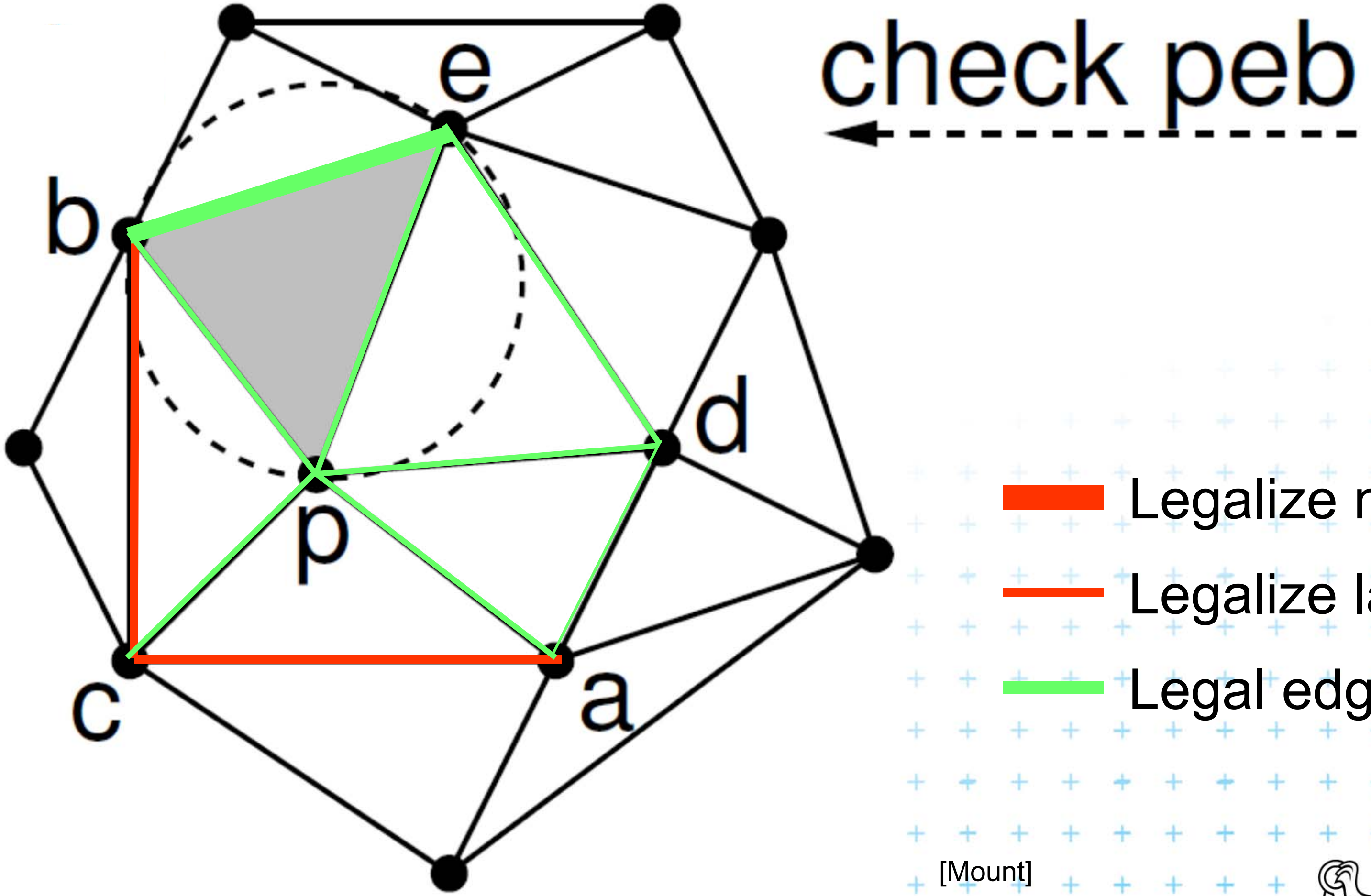
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



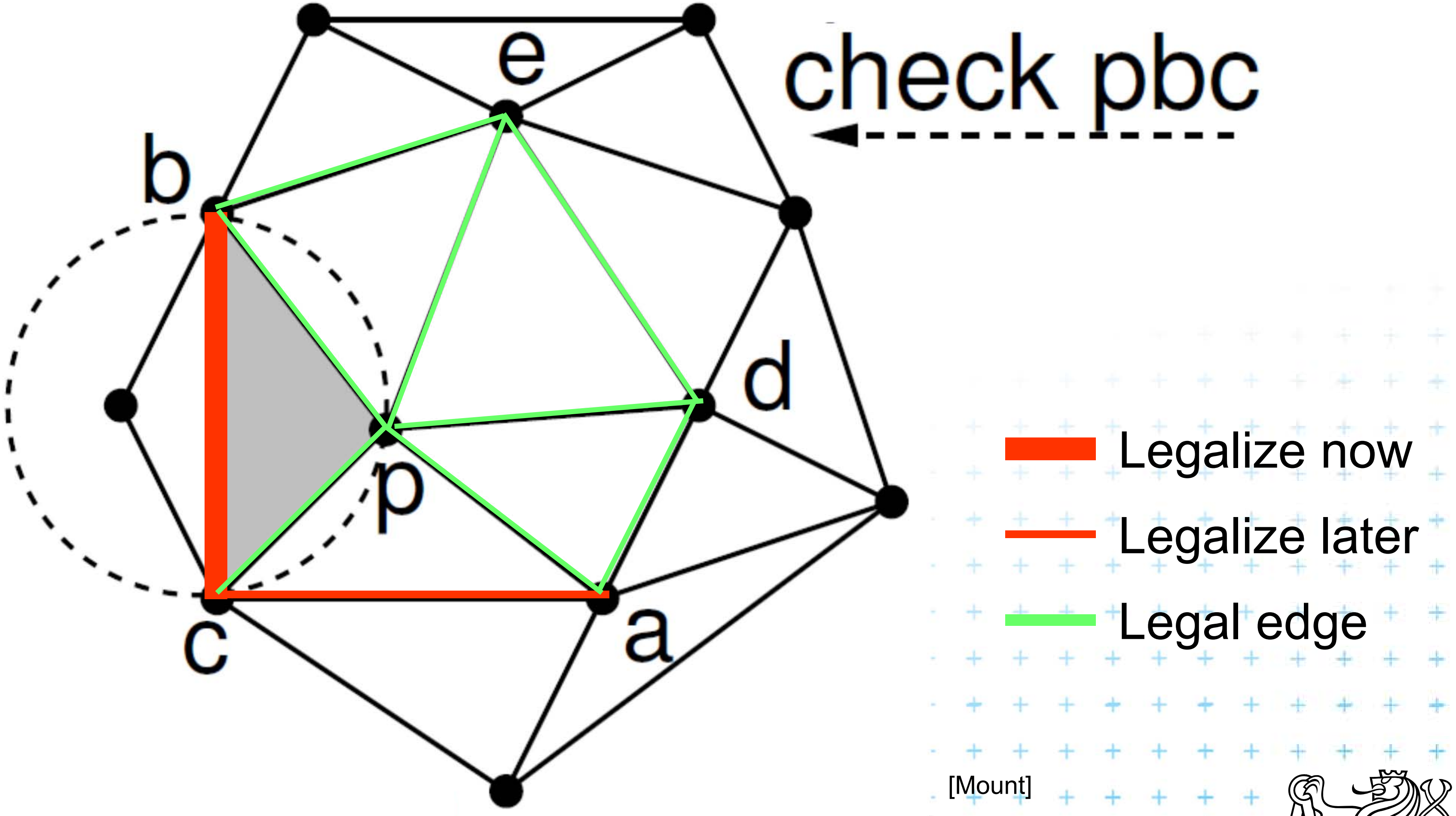
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

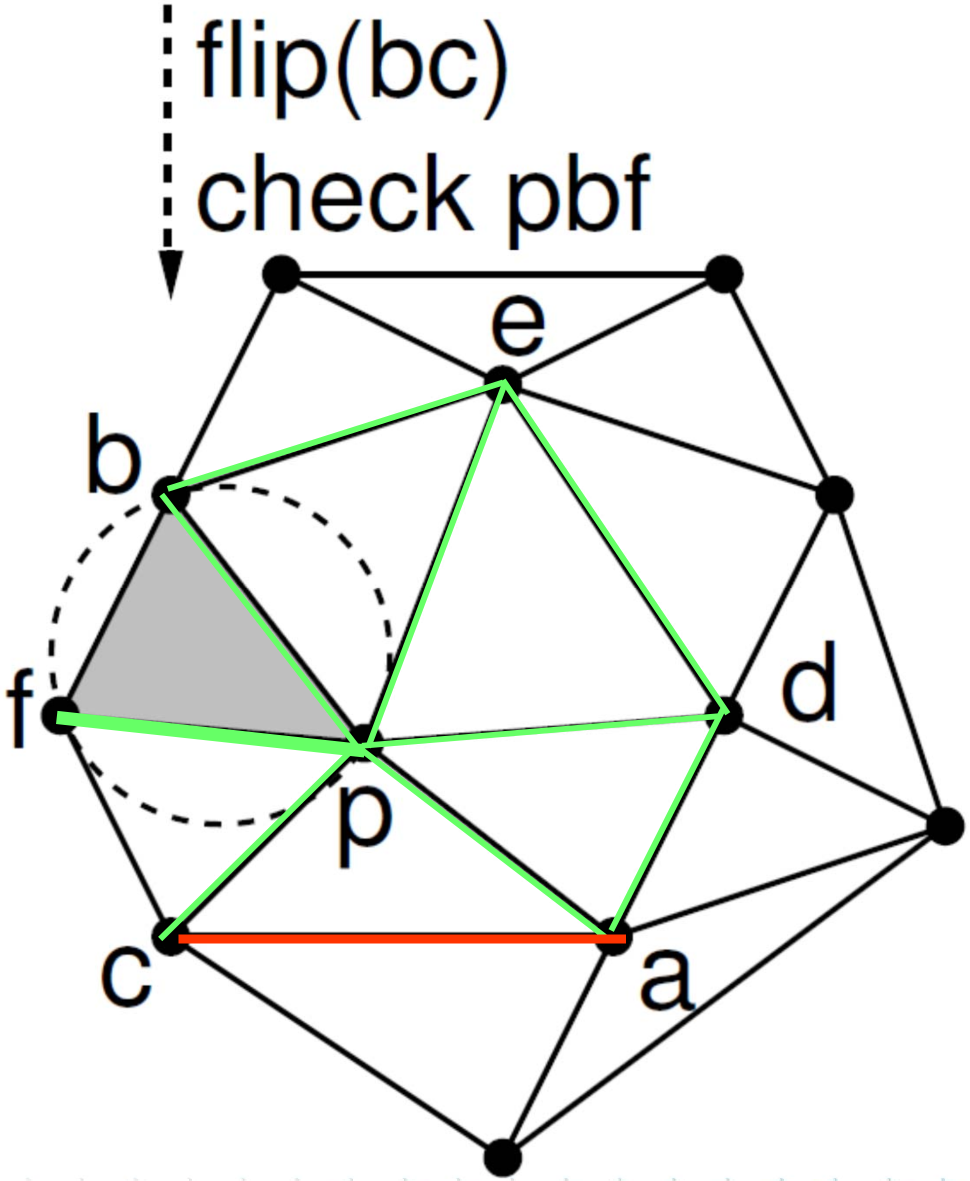
[Mount]

Delaunay triangulation – other point insert



[Mount]

Delaunay triangulation – other point insert



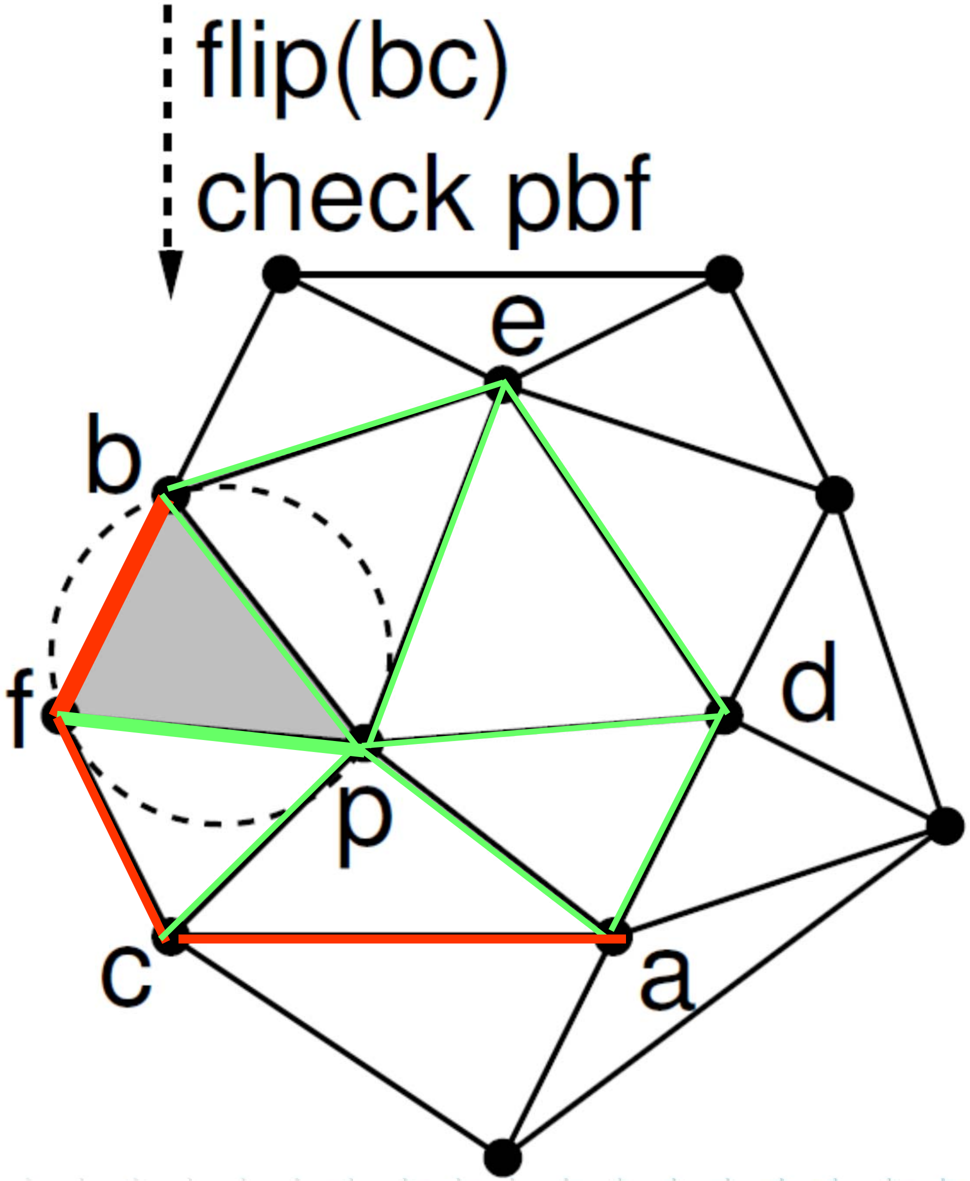
🟥 Legalize now

— Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



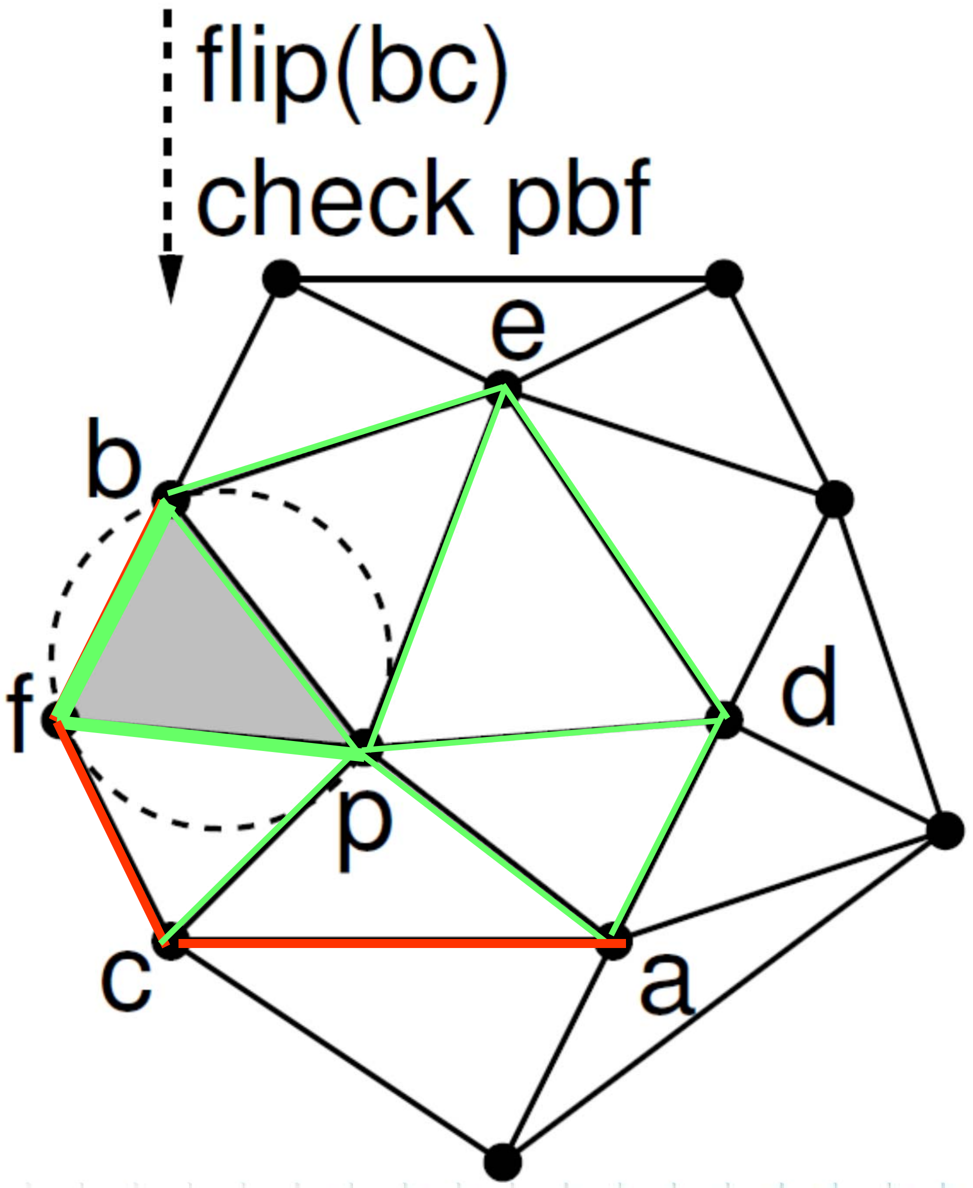
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



flip(bc)

check pbf

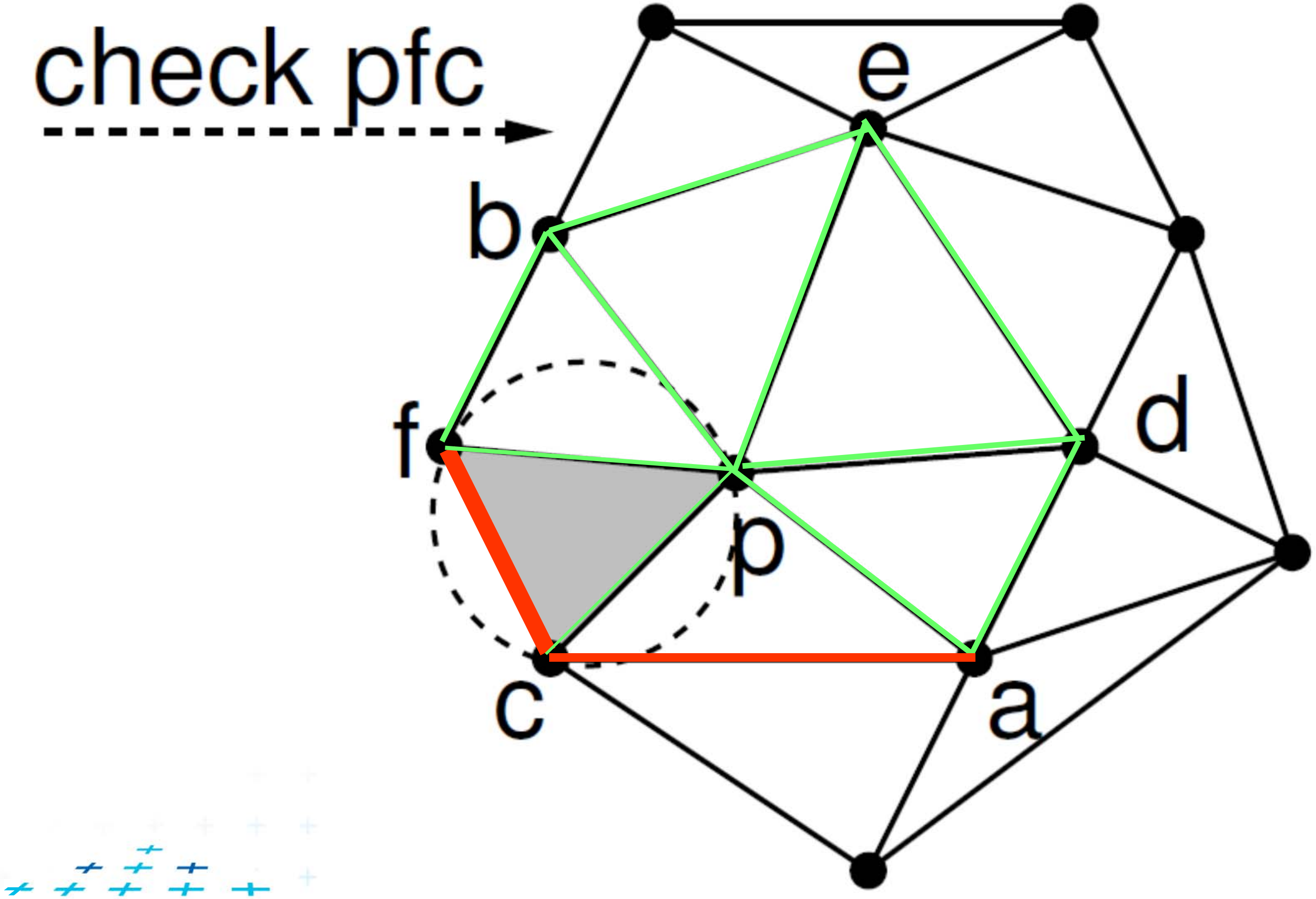
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



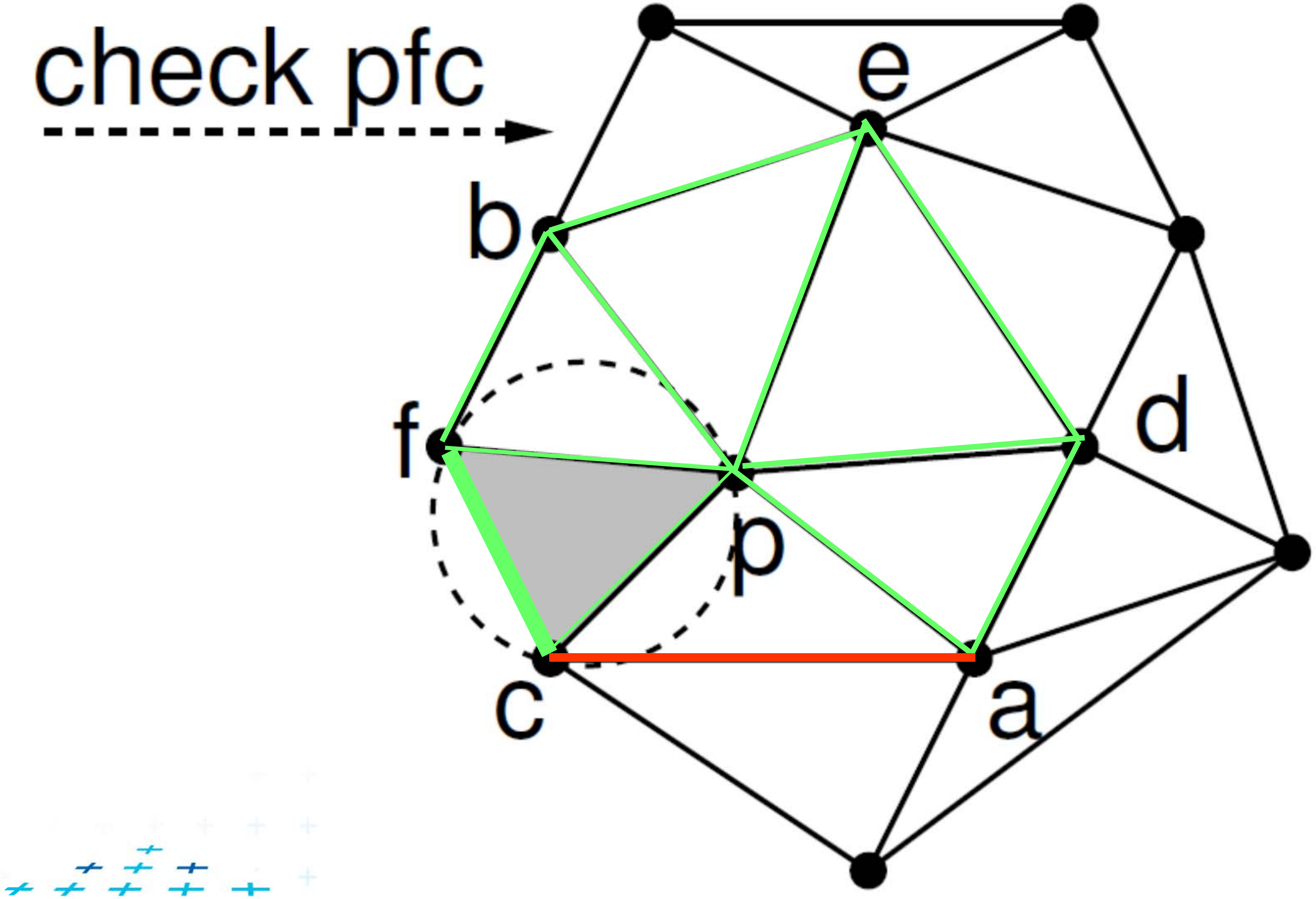
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



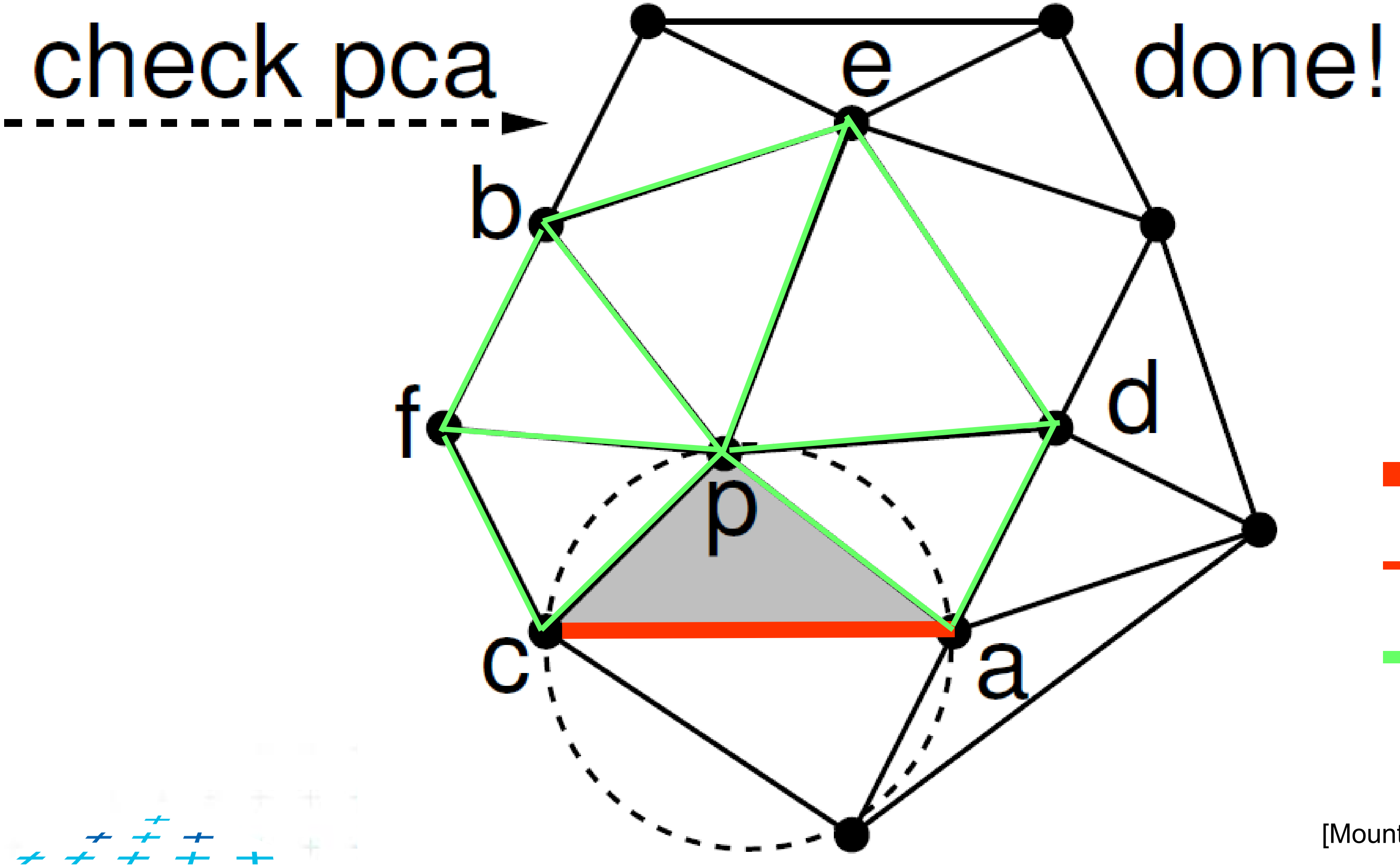
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



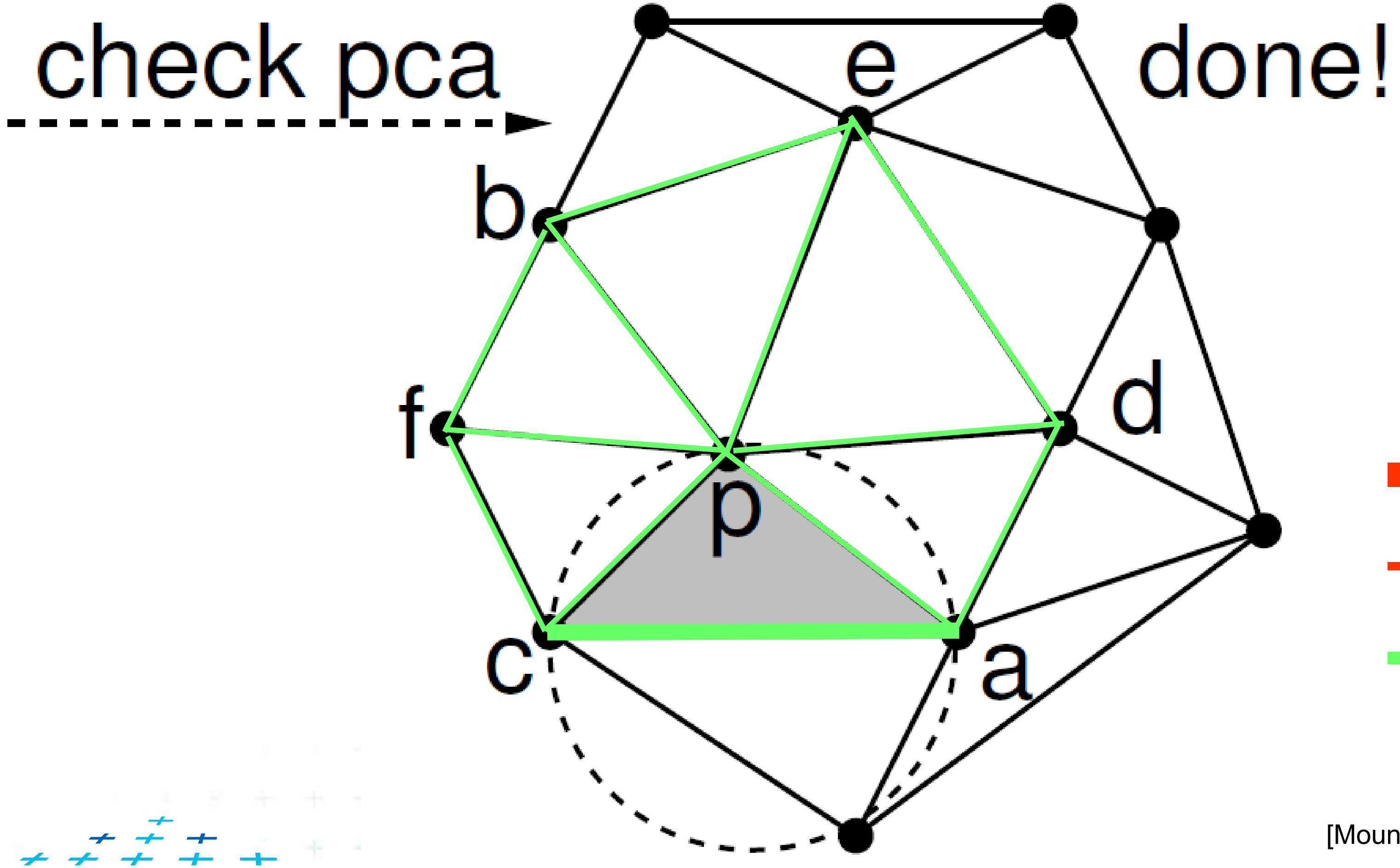
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



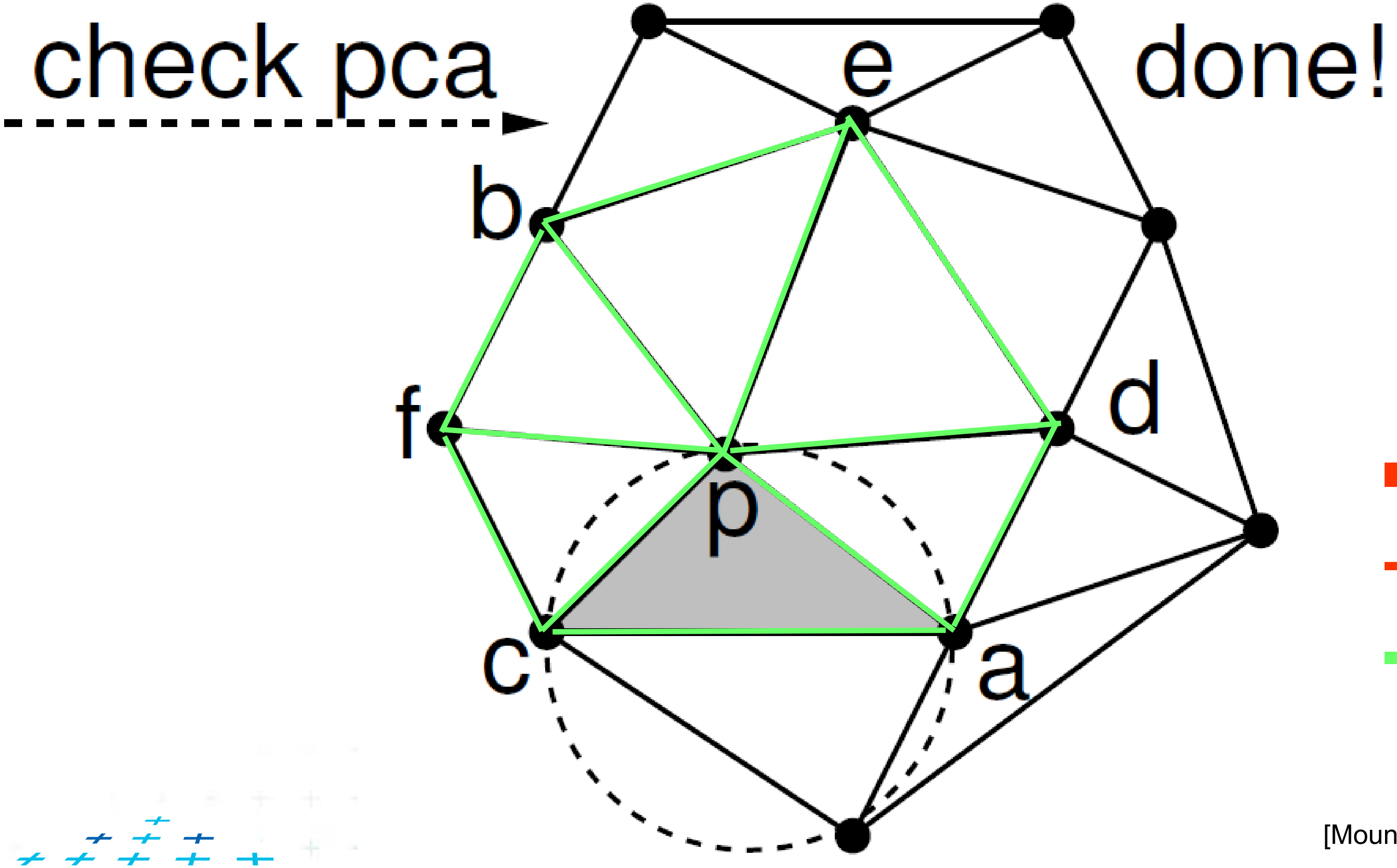
🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Delaunay triangulation – other point insert



🟥 Legalize now

🟥 Legalize later

🟩 Legal edge

[Mount]

Correctness of the algorithm

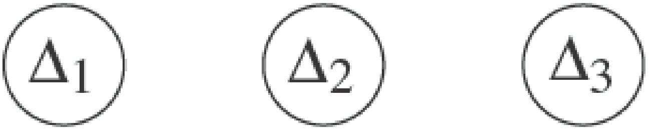
- Every new edge (created due to insertion of *p)*
  - is incident to *p*
  - must be legal
    => no need to test them
- Edge can only become illegal if one of its incident triangle changes
  - Algorithm tests any edge that may become illegal
    => the algorithm is correct
- Every edge flip makes the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

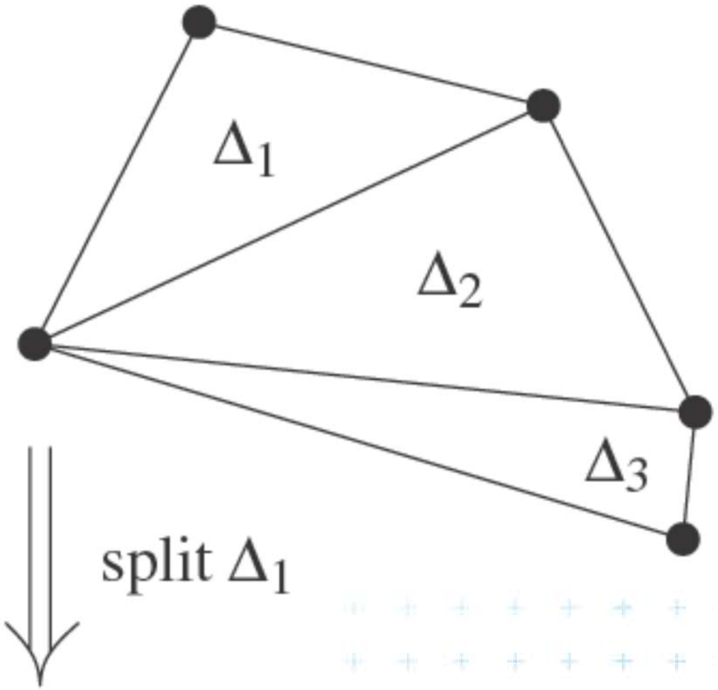
Point location data structure

- For finding a triangle *abc* ∈ *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

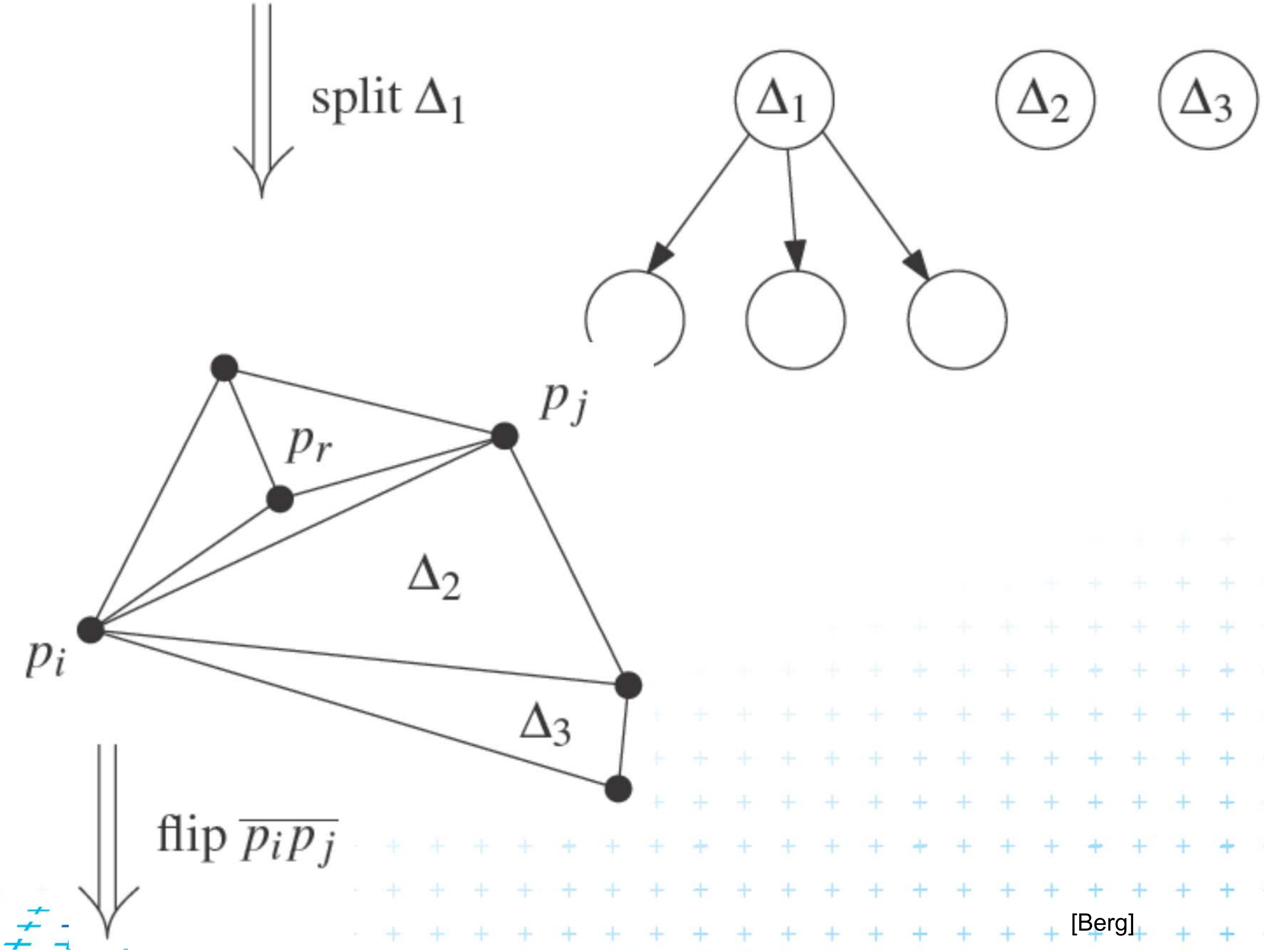
Simplified
- it should also contain the root node





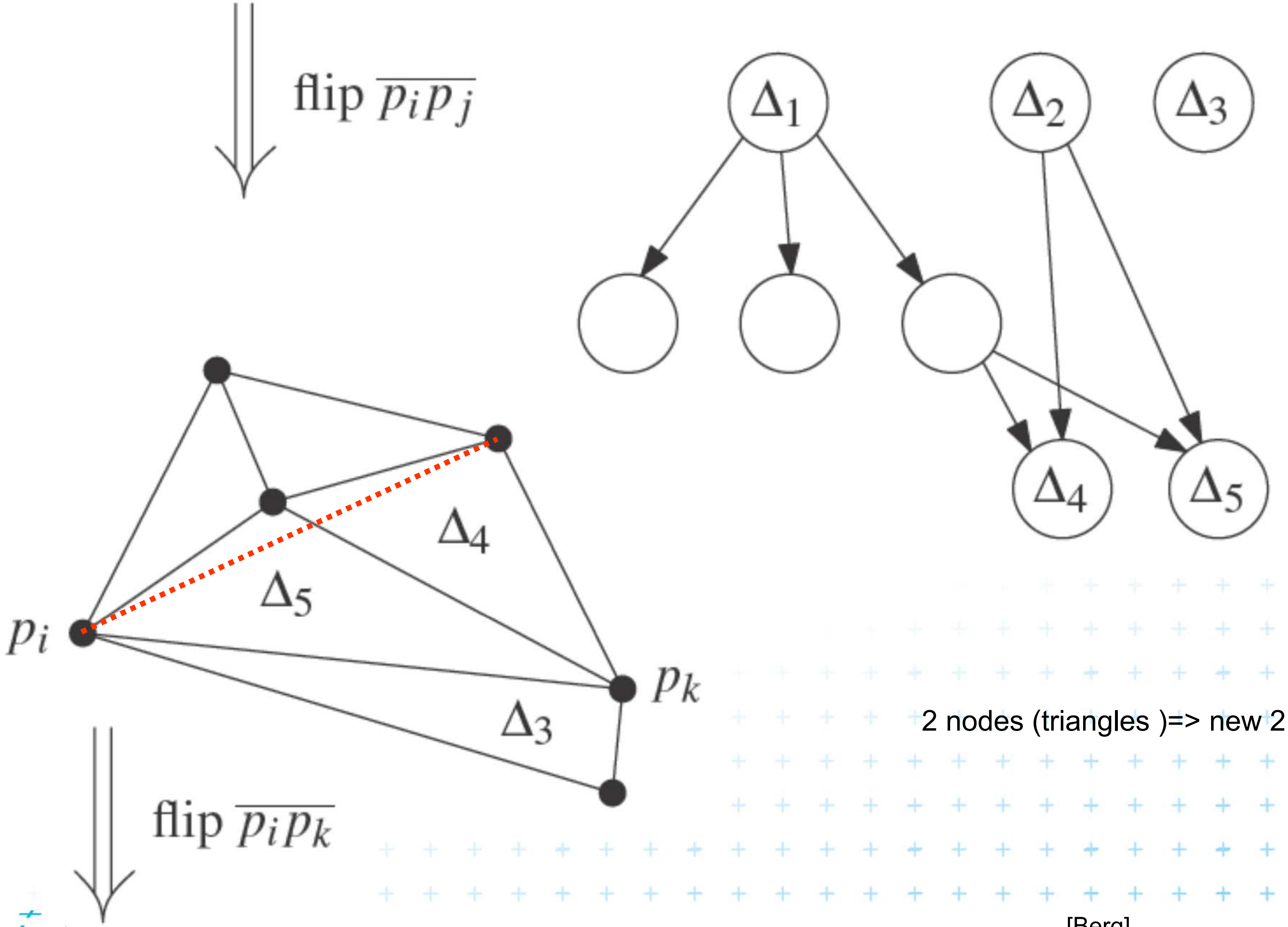
[Berg]

Point location data structure



[Berg]

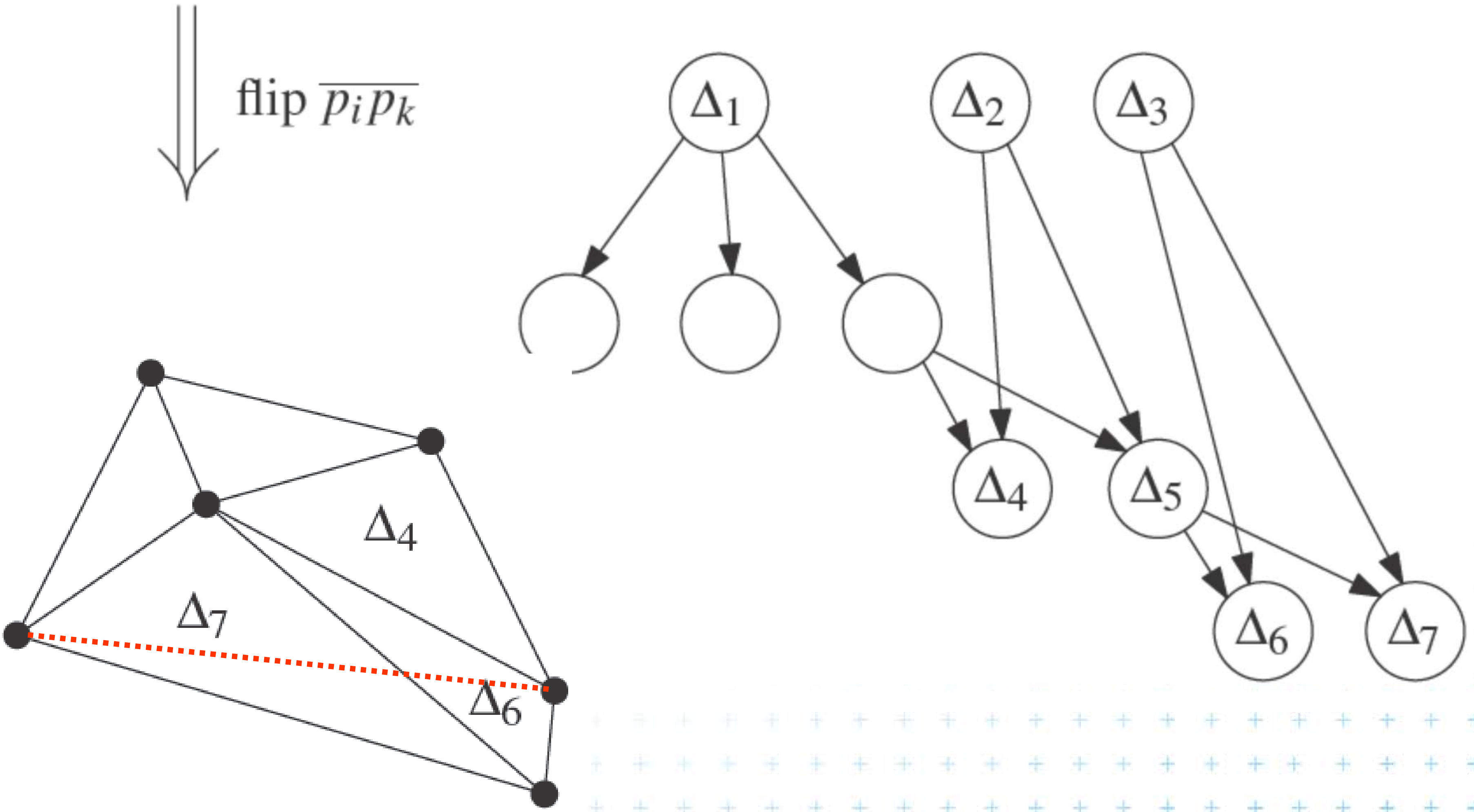
Point location data structure



2 nodes (triangles)=> new 2 nodes

[Berg]

Point location data structure

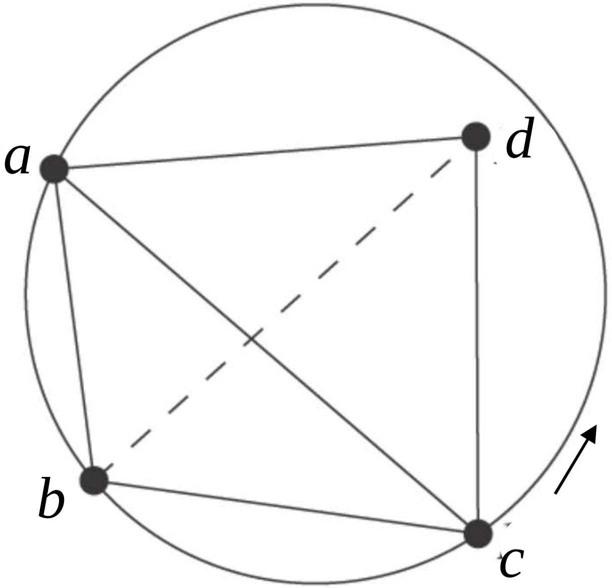


[Berg]

InCircle test

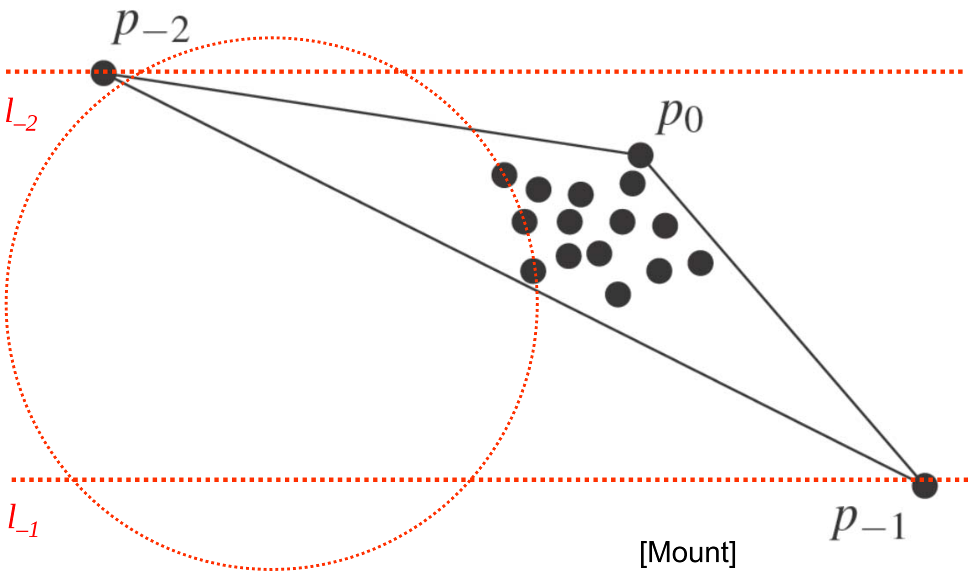
- *a,b,c* are counterclockwise in the plane
- Test, if *d* lies to the left of the oriented circle through *a,b,c*

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



[Mount]

Creation of the initial triangle



[Mount]

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*

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

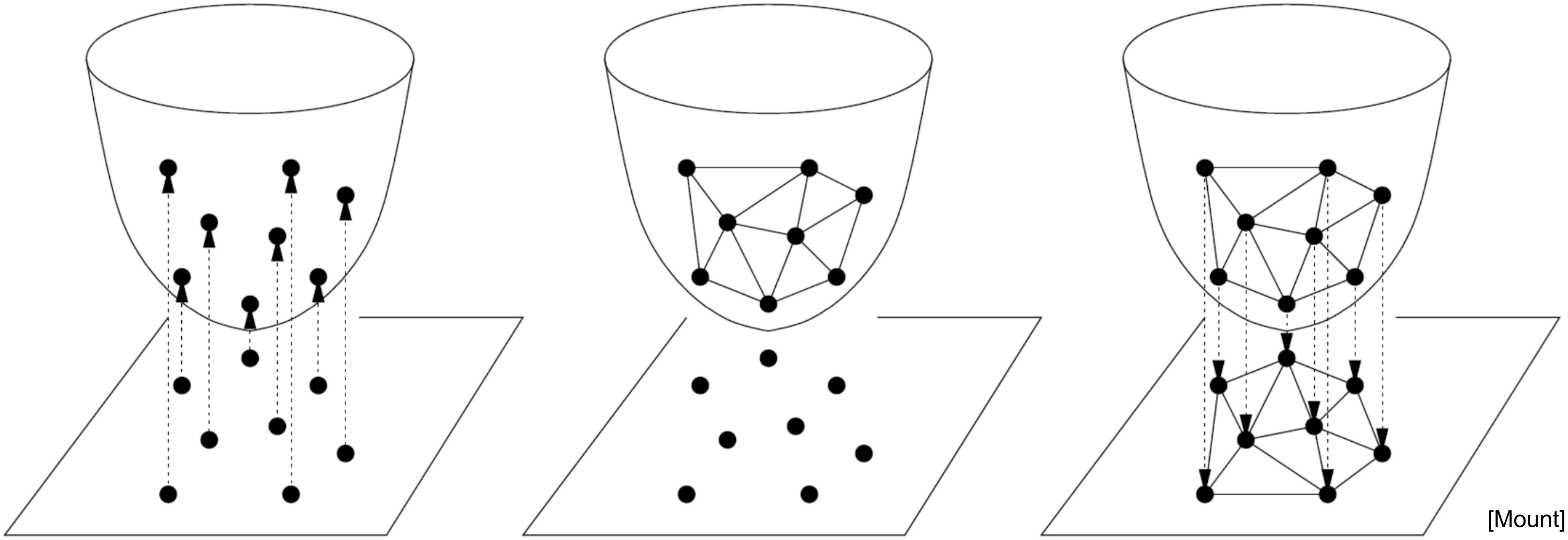
Complexity of incremental DT algorithm

- 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]

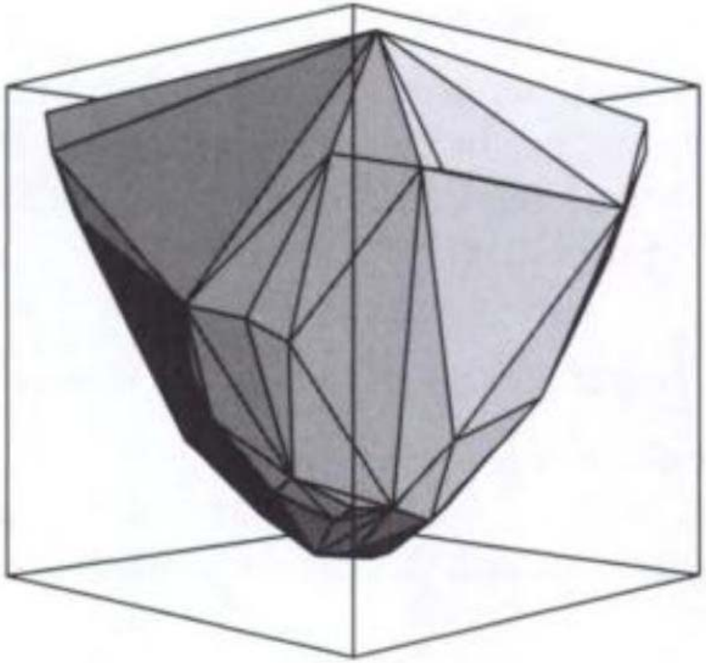
Project onto paraboloid. Compute convex hull. Project hull faces back to plane.

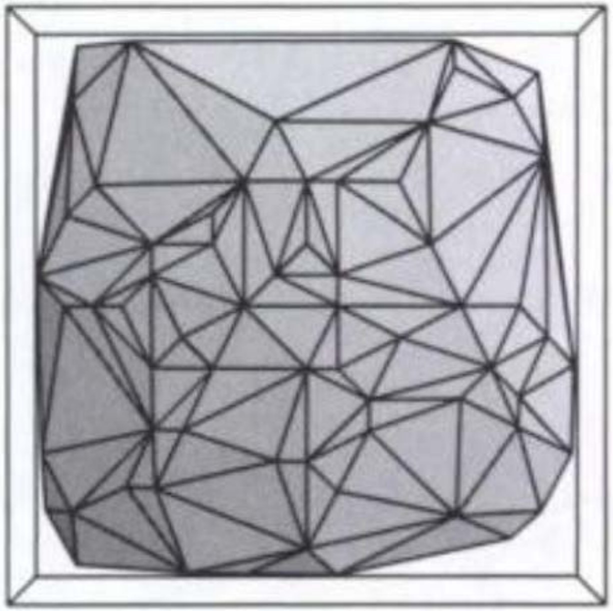
Vertical projection of points to paraboloid

- 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)



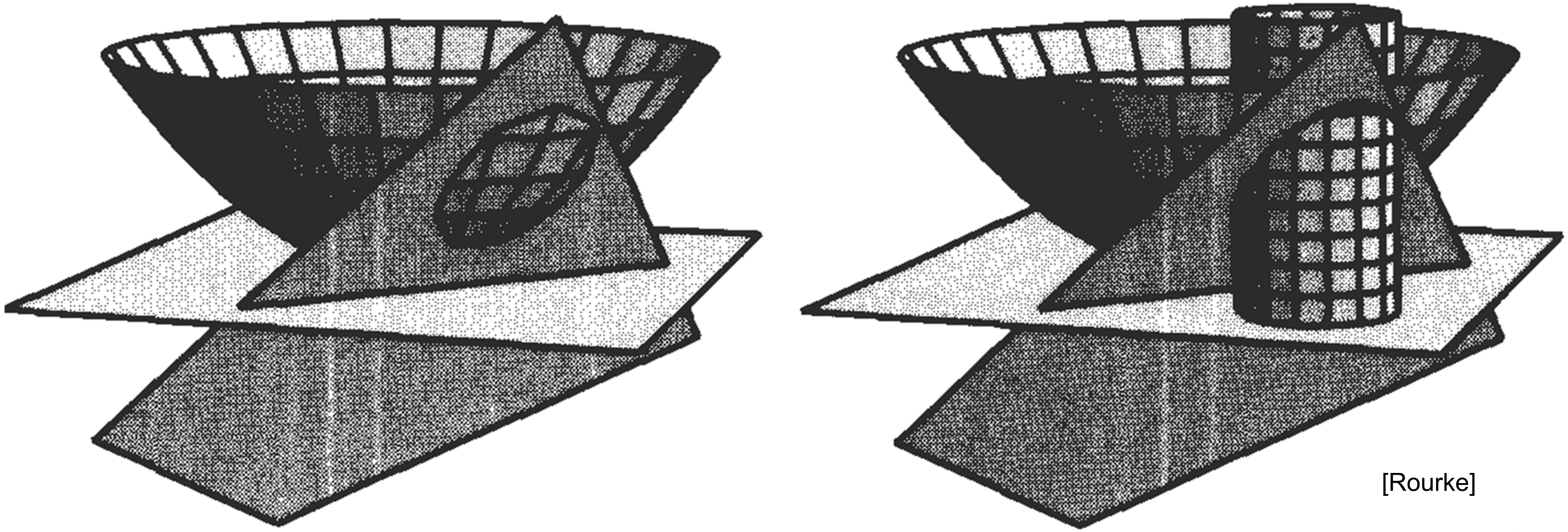


[Rourke]

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')

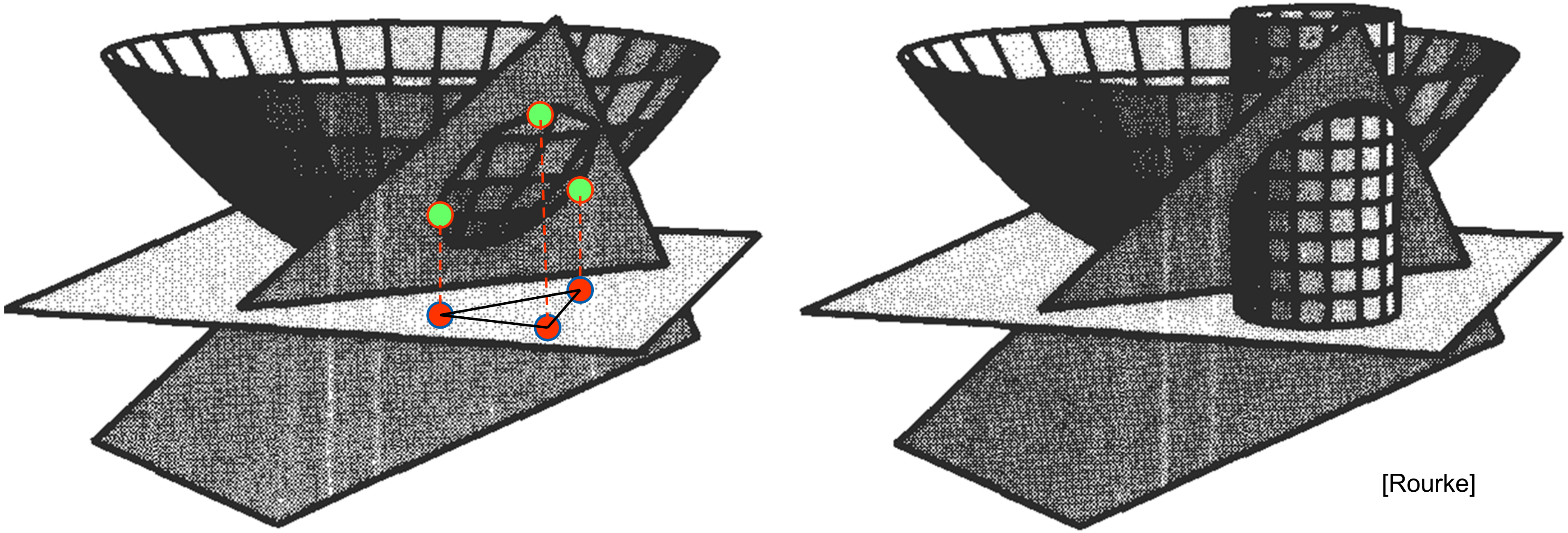
Relation between CH and DT



[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'*.

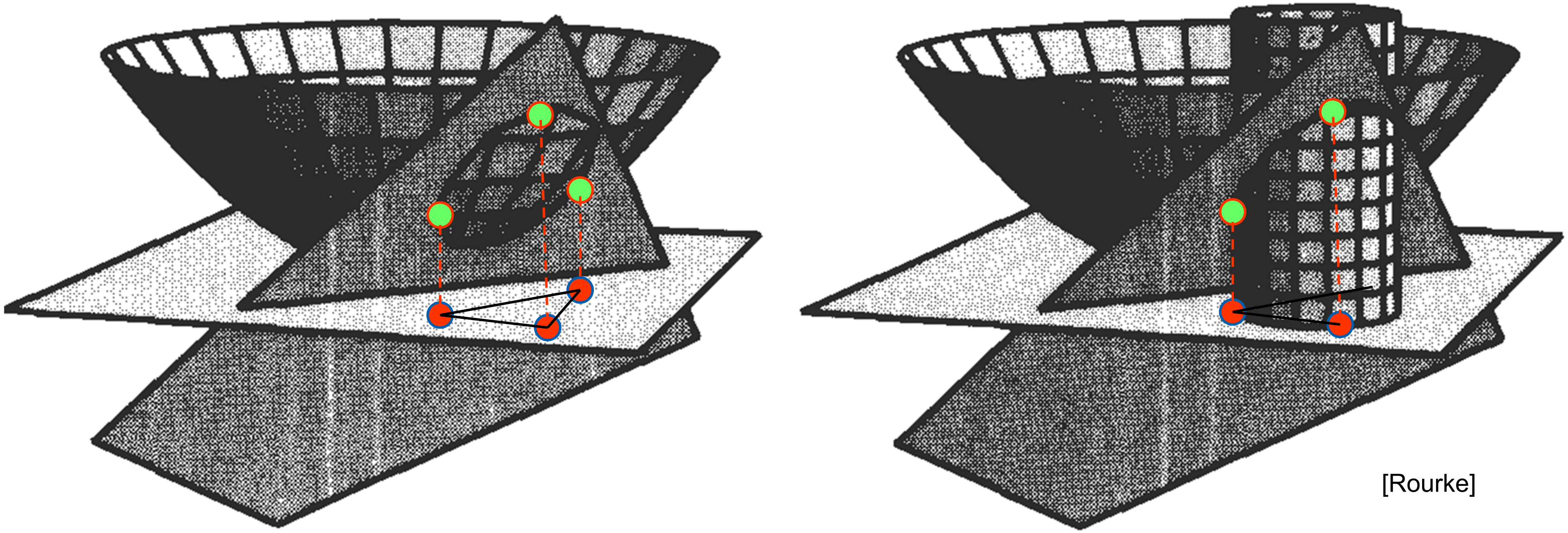
Relation between CH and DT



[Rourke]

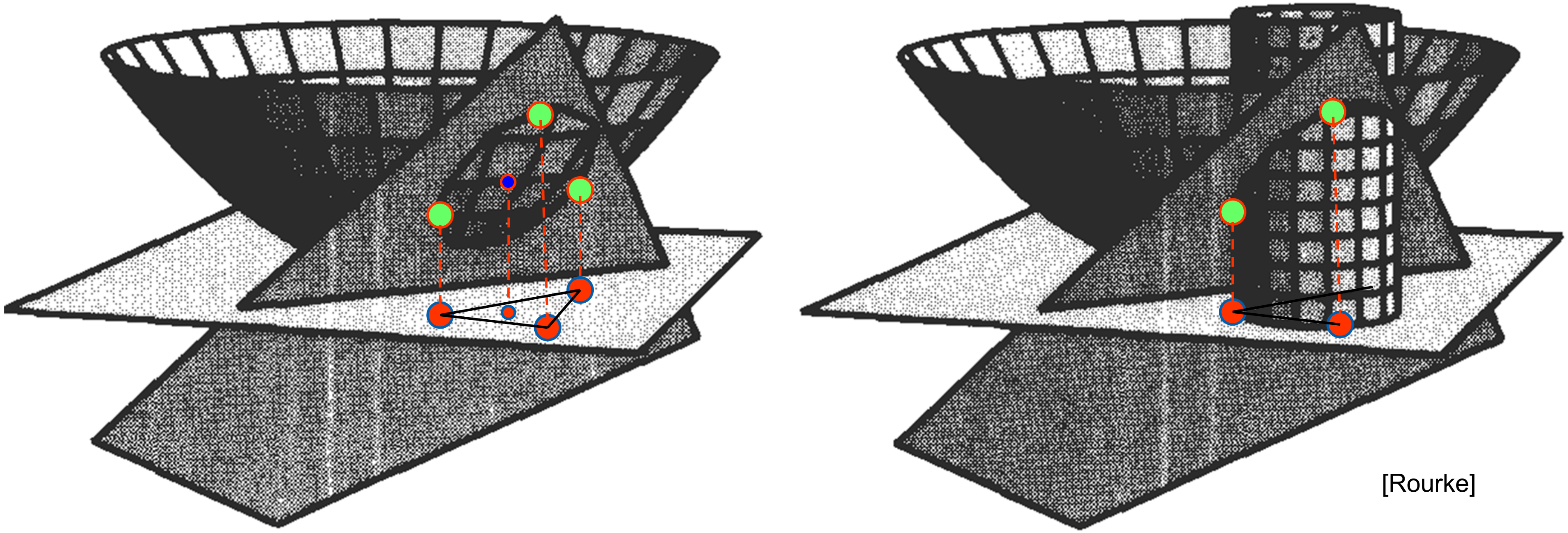
- 4 distinct points *p,q,r,s* in the plane, and let *p'*, *q'*, *r'*, *s'* be their respective projections onto the paraboloid, $z = x^2 + y^2$.
- The point *s* lies within the circumcircle of *pqr* **iff** s' lies on the lower side of the plane passing through *p'*, *q'*, *r'*.

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

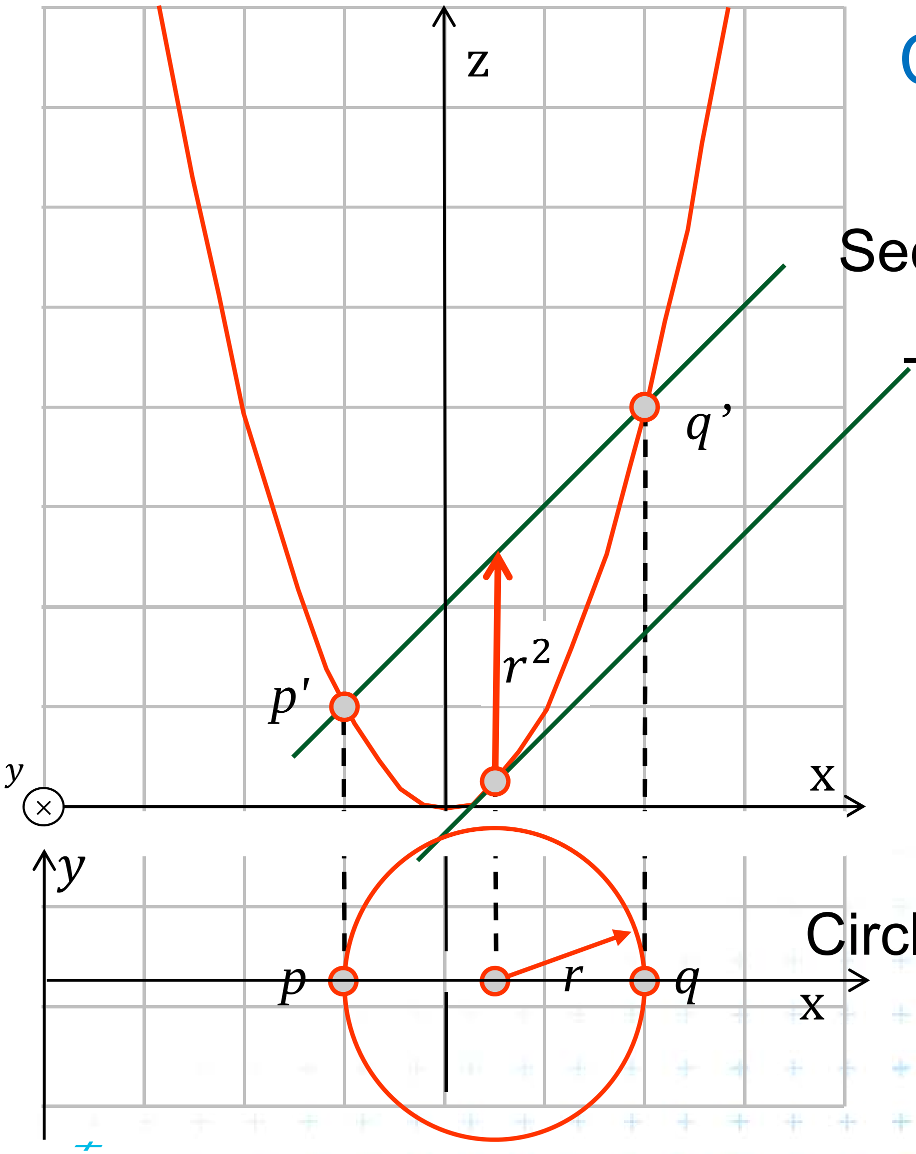


[Rourke]

- 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: $z = 2ax + 2by + \gamma$

$$a^2 + b^2 = 2a.a + 2b.b + \gamma$$

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

- 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) $z = x^2 + y^2$

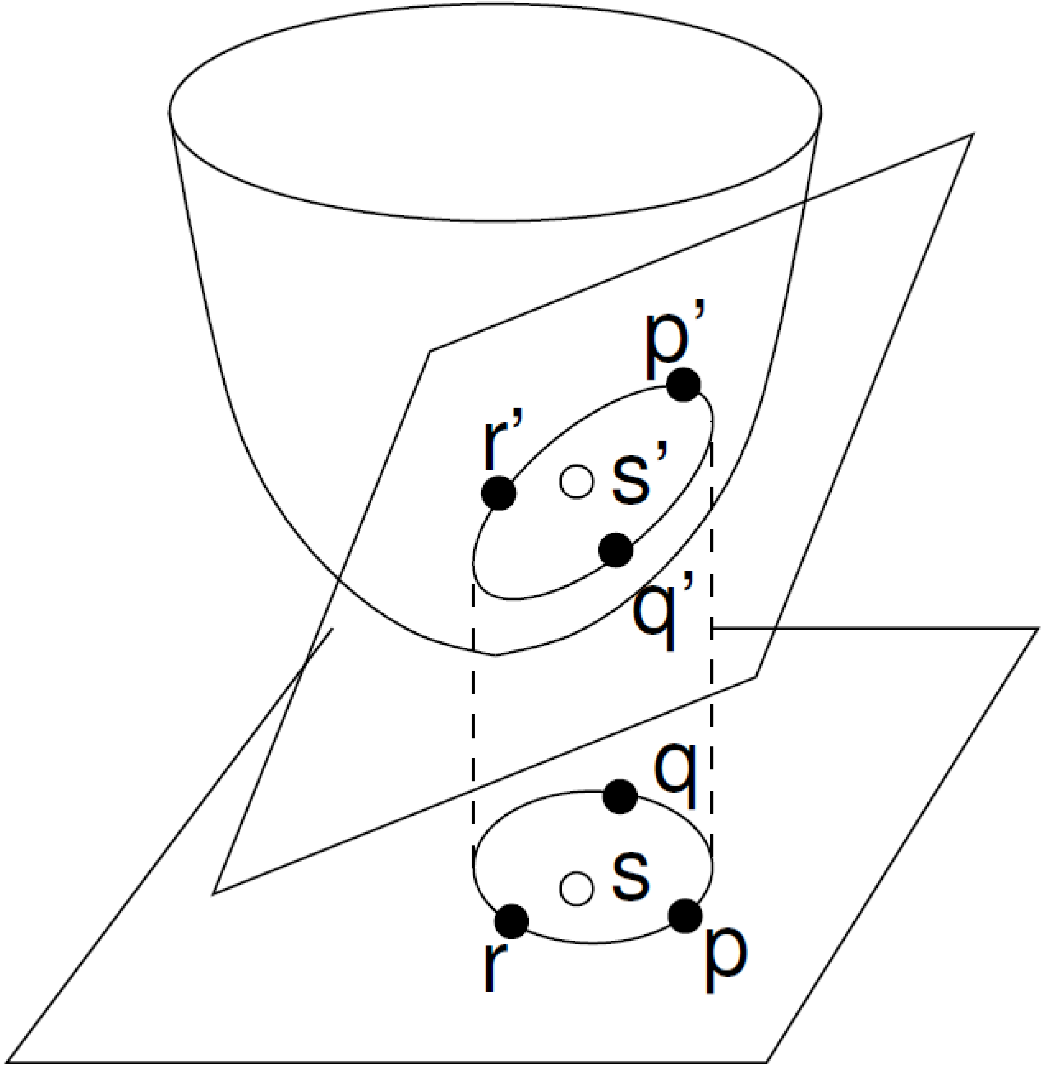
  $x^2 + y^2 = 2ax + 2by - (a^2 + b^2) + r^2$

- 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

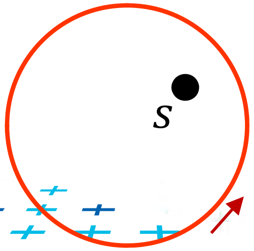


[Mount]

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)

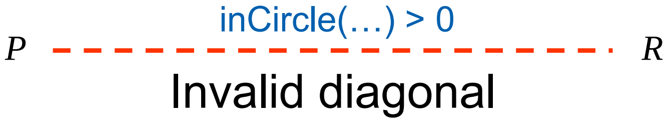
$$\mathrm{in}(p,q,r,s) = \det\begin{pmatrix} p_x & p_y & p_x^2+p_y^2 & 1 \\ q_x & q_y & q_x^2+q_y^2 & 1 \\ r_x & r_y & r_x^2+r_y^2 & 1 \\ s_x & s_y & s_x^2+s_y^2 & 1 \end{pmatrix} > 0.$$



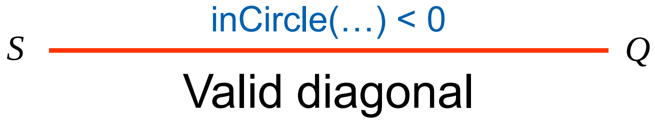
[Mount]

Delaunay triangulation and inCircle test

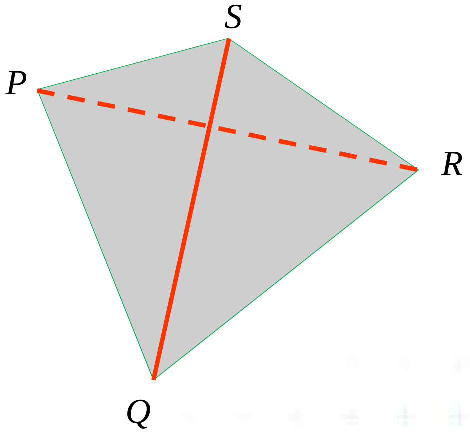
- DT splits each quadrangle by one of its two diagonals
- For a valid diagonal, the fourth point is not inCircle
  - => the fourth point is right from the oriented circumcircle (outside)
  - => inCircle(….) < 0 for CCW orientation
- inCircle(P,Q,R,S) = inCircle(P,R,S,Q) = – inCircle(P,Q,S,R) = – inCircle(S,Q,R,P)



Invalid diagonal



Valid diagonal



Delaunay triangulation and inCircle test

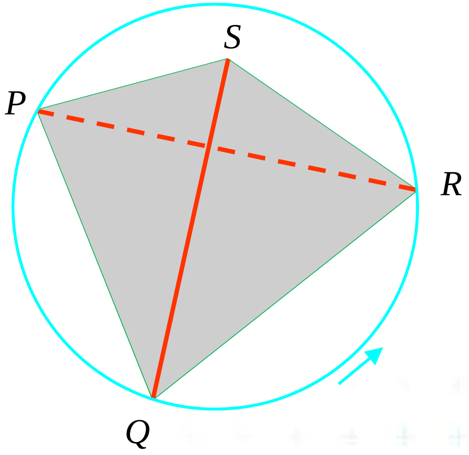
- DT splits each quadrangle by one of its two diagonals
- For a valid diagonal, the fourth point is not inCircle
  - => the fourth point is right from the oriented circumcircle (outside)
  - => inCircle(….) < 0 for CCW orientation
- inCircle(P,Q,R,S) = inCircle(P,R,S,Q) = – inCircle(P,Q,S,R) = – inCircle(S,Q,R,P)

P - - - - inCircle(…) > 0 - - - - *R*

Invalid diagonal

S ——— inCircle(…) < 0 ——— *Q*

Valid diagonal

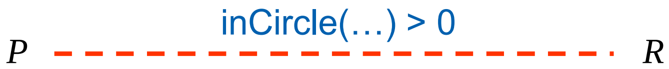


Delaunay triangulation and inCircle test

- 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)

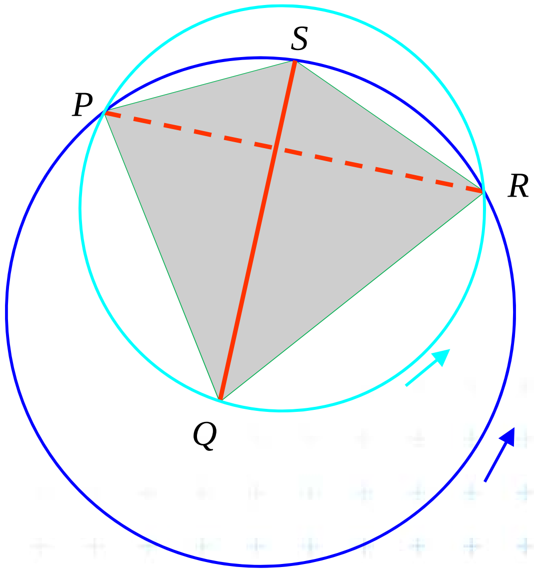
  => inCircle(….) < 0 for CCW orientation
- inCircle(P,Q,R,S) = inCircle(P,R,S,Q) = – inCircle(P,Q,S,R) = – inCircle(S,Q,R,P)



Invalid diagonal

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)

inCircle(…) > 0

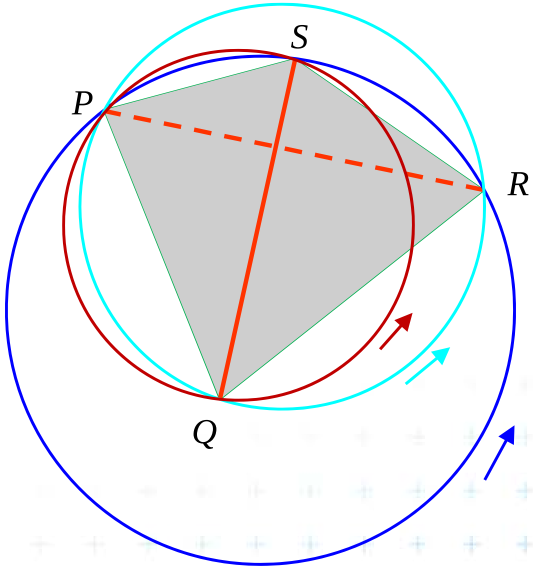
P - - - - - - - - - - *R*

Invalid diagonal

inCircle(…) < 0

S ———————— *Q*

Valid diagonal



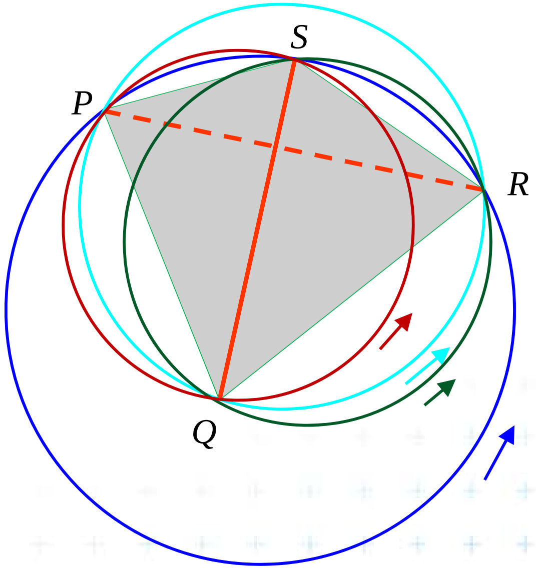
Delaunay triangulation and inCircle test

- 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)

inCircle(…) > 0
P
R
inCircle(…) < 0
S
Q

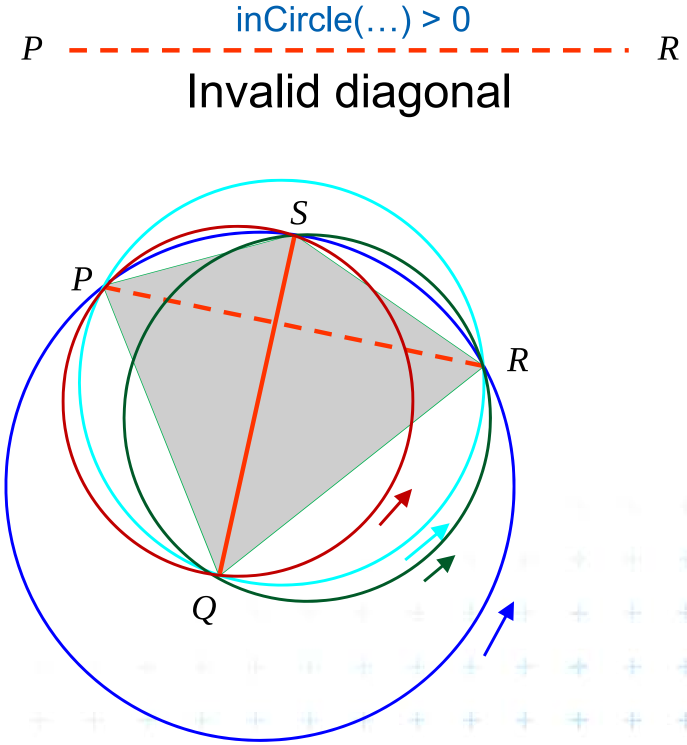
Invalid diagonal

Valid diagonal

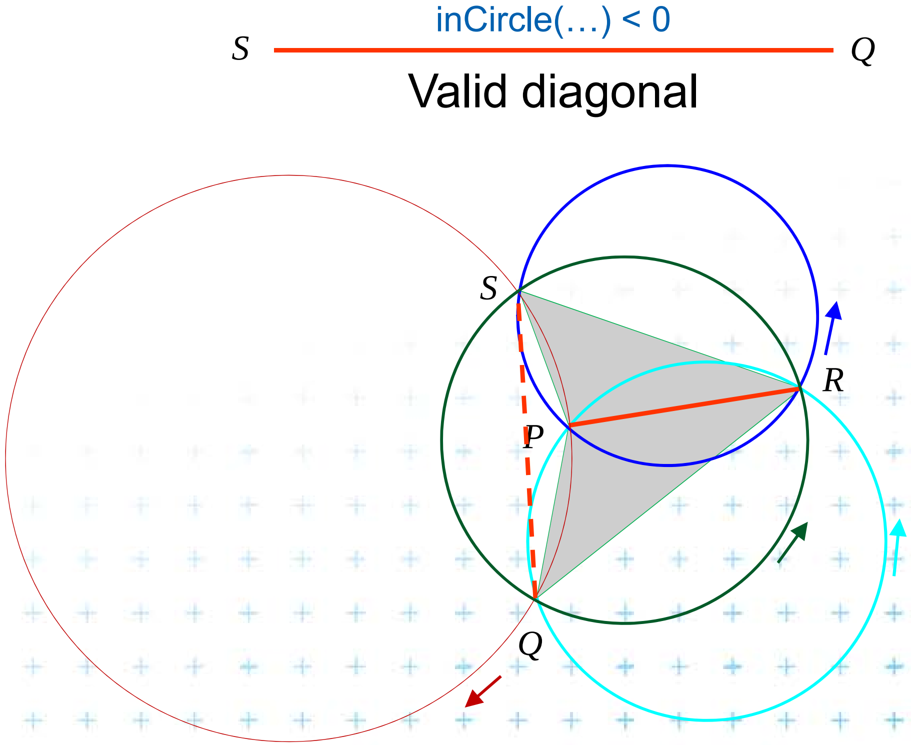


Delaunay triangulation and inCircle test

- 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

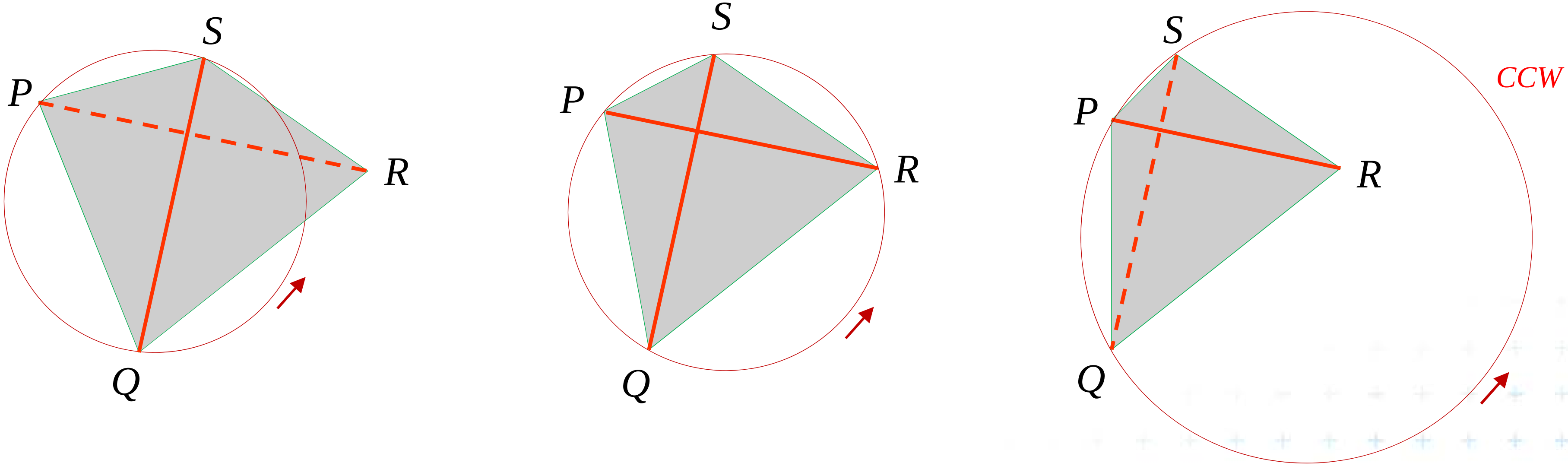


Valid diagonal

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)

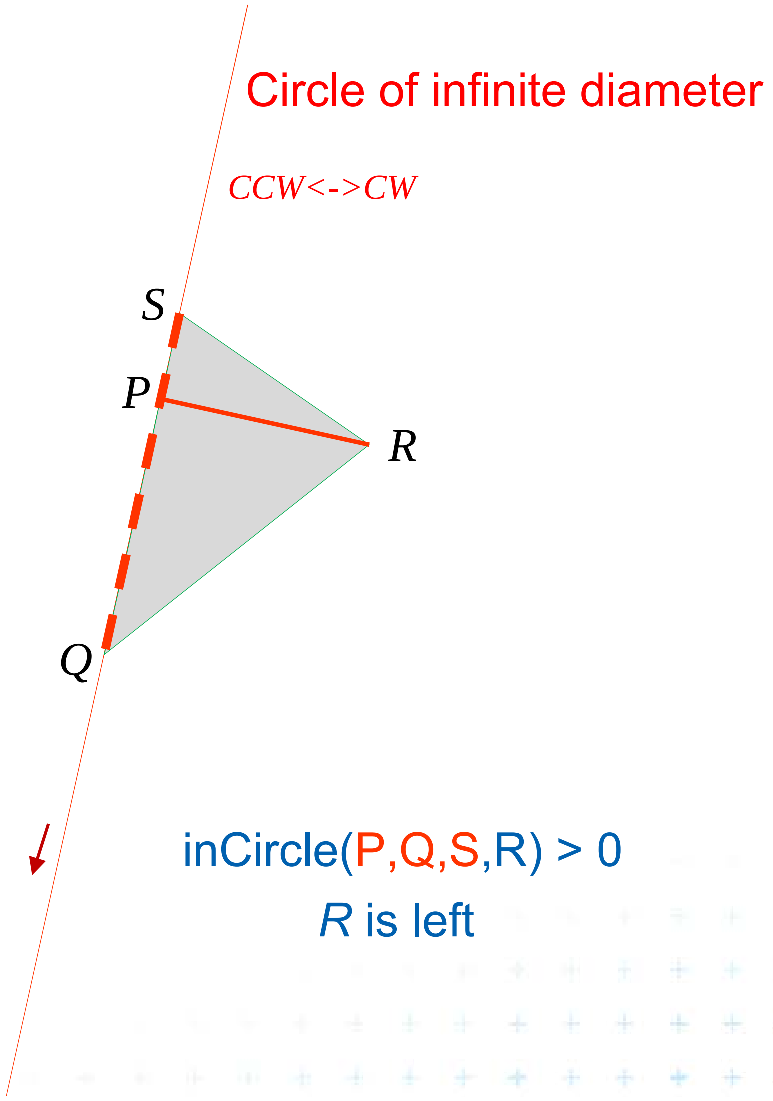
inCircle(P,Q,S,R) = 0
R is on the circle

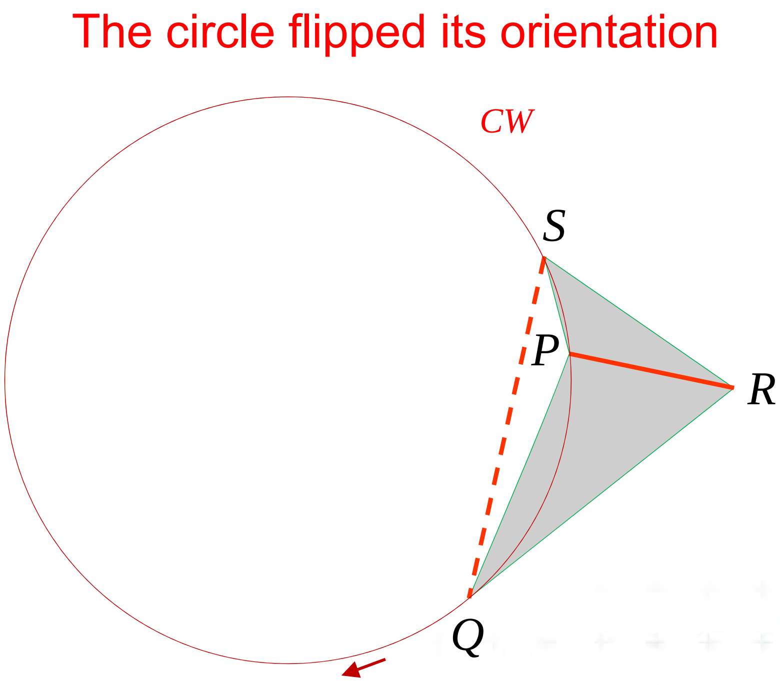
inCircle(P,Q,S,R) > 0
R is left (in)

Invalid diagonal

Valid diagonal

inCircle test detail





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

R is left

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

R is left

Invalid diagonal

Valid diagonal

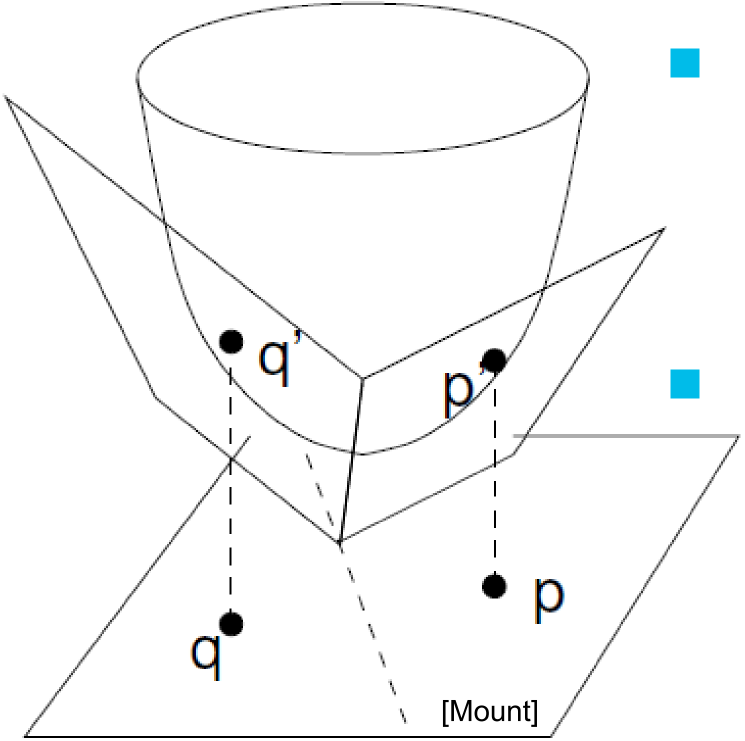
An the Voronoi diagram?

- 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

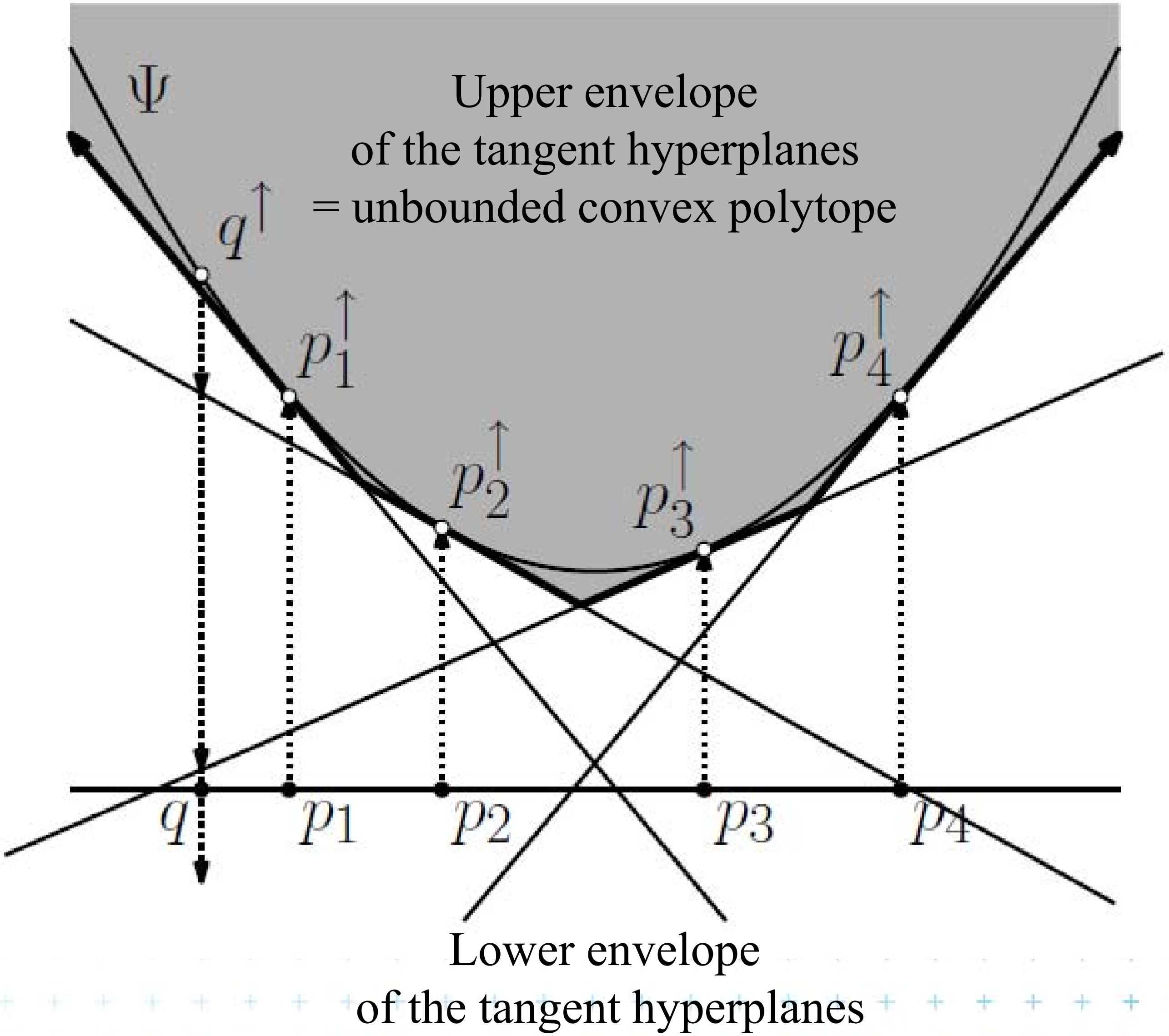
$$H^+(p) = \{(x, y, z) \mid z \geq 2ax + 2by - (a^2 + b^2)\}$$



[Mount]

- 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 $H^+(p_i)$

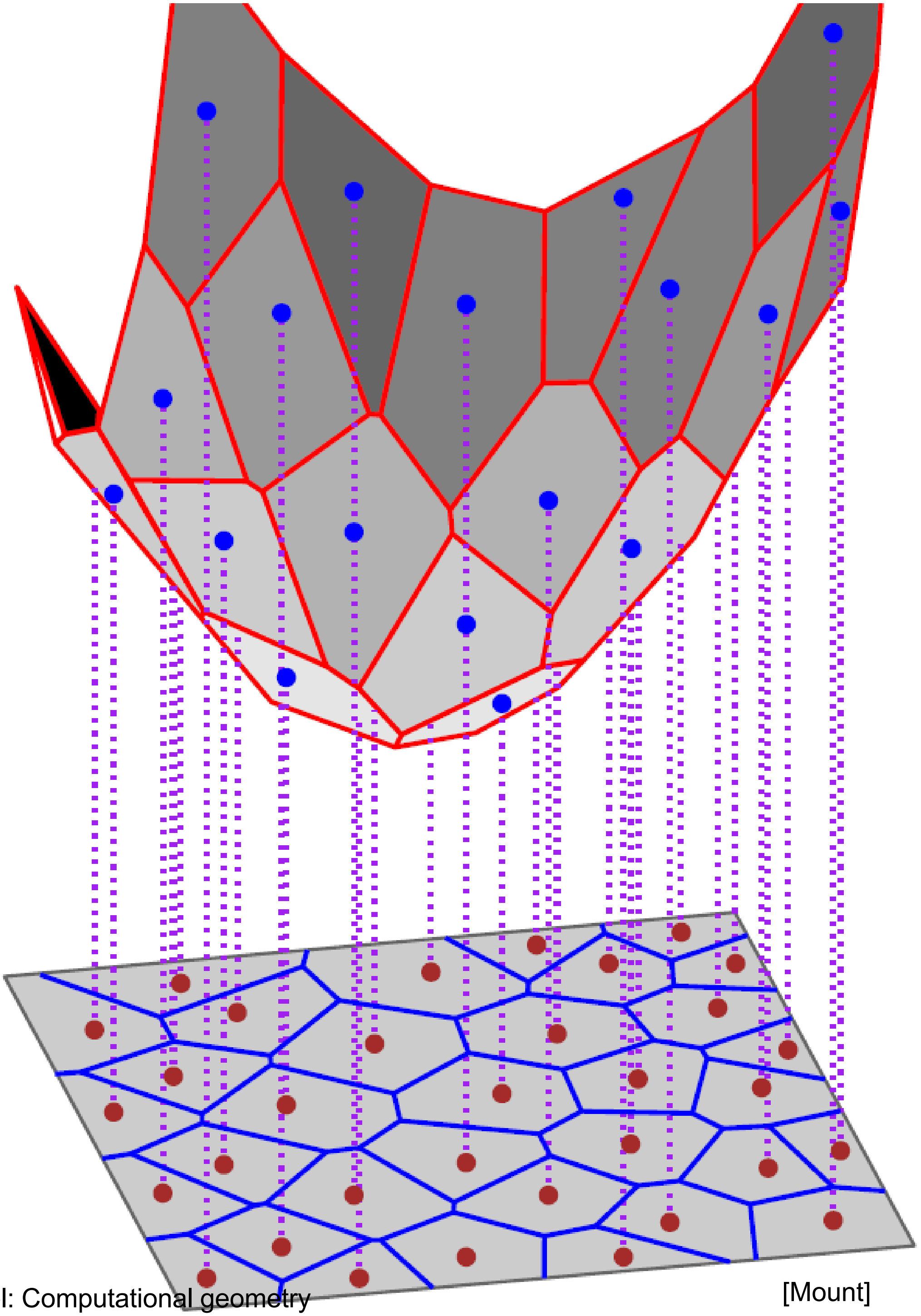
Upper envelope of planes



[Mount]

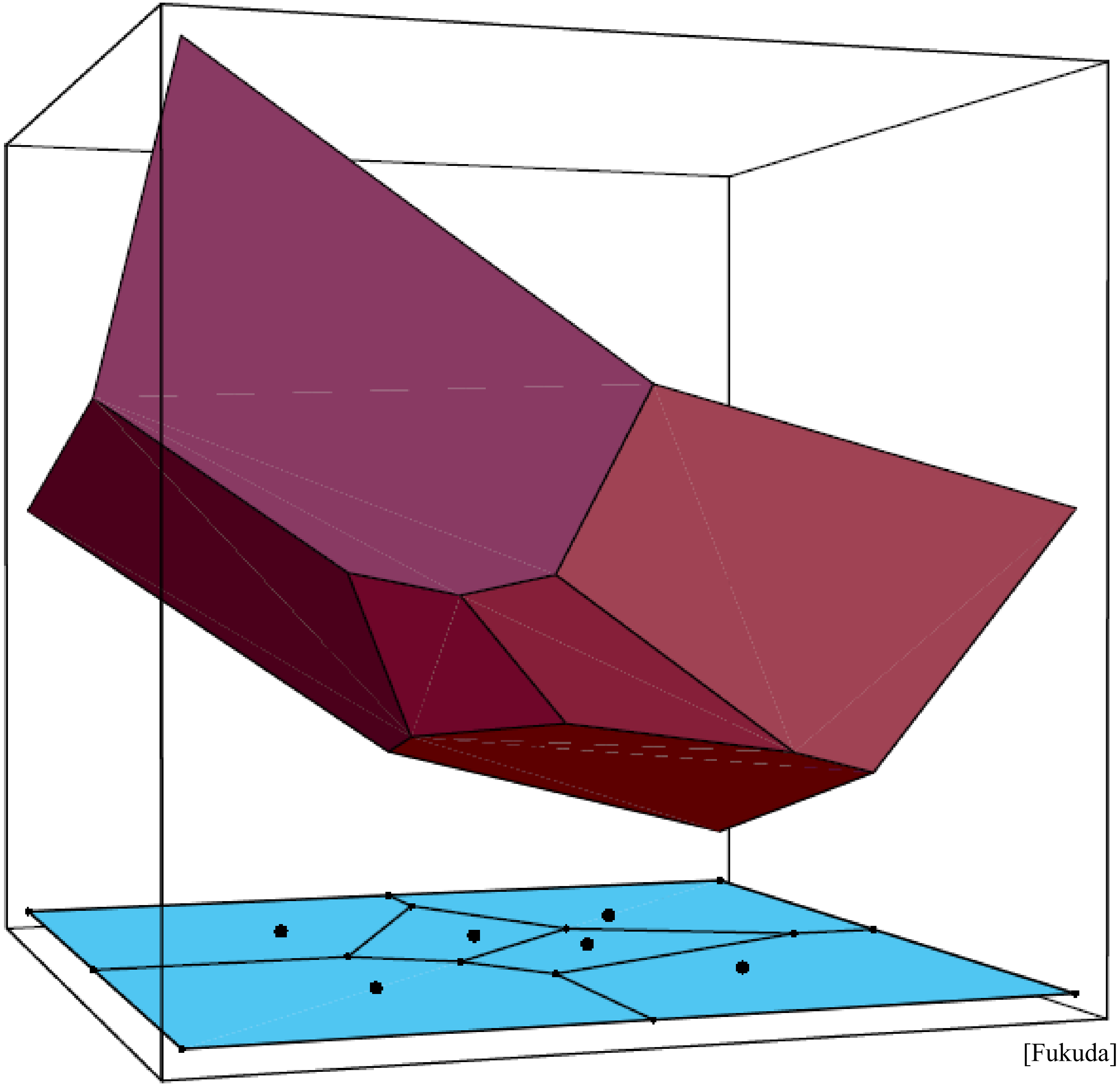
Projection to 2D

- 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*)

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

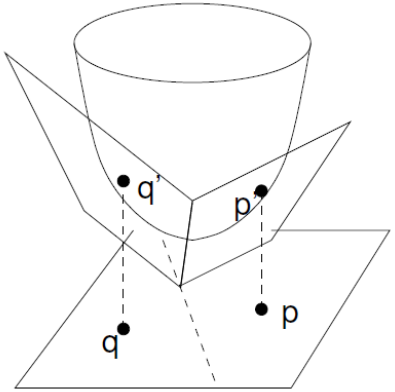
- This line passes through midpoint between *p* and *q*

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

- 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

DCGI

DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

INTERSECTIONS OF LINE SEGMENTS AND POLYGONS

PETR FELKEL

FEL CTU PRAGUE

Version from 17.1.2019

Talk overview

- Intersections of line segments (Bentley-Ottmann)
  - Motivation
  - Sweep line algorithm recapitulation
  - Sweep line intersections of line segments
- Intersection of polygons or planar subdivisions
  - See assignment [21] or [Berg, Section 2.3]
- Intersection of axis parallel rectangles
  - See assignment [26]

Geometric intersections – what are they for?

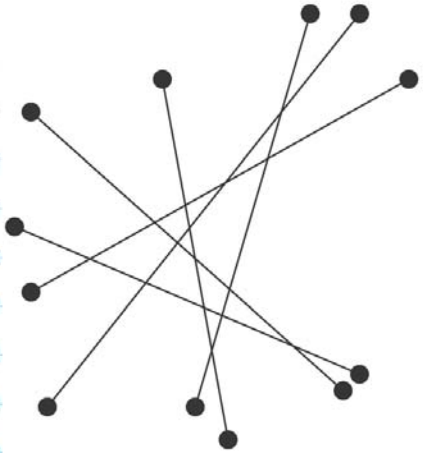
One of the most basic problems in computational geometry

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

Line segment intersection

- Intersection of complex shapes is often reduced to simpler and simpler intersection problems
- Line segment intersection is the most basic intersection algorithm
- Problem statement:

  Given *n* line segments in the plane, report all points where a pair of line segments intersect.
- Problem complexity
  - Worst case – $I = O(n^2)$ intersections
  - Practical case – only some intersections
  - Use an output sensitive algorithm
    - $O(n \log n + I)$ optimal randomized algorithm
    - $O(n \log n + I \log n)$ sweep line algorithm - %



[Berg]

Plane sweep line algorithm recapitulation

Postupový plán

- Horizontal line (sweep line, *scan line*) ℓ moves top-down (or vertical line: left to right) over the set of objects
- The move is not continuous, but ℓ jumps from one event point to another
  - Event points are in priority queue or sorted list (~y)
  - The (left) top-most event point is removed first
  - New event points may be created (usually as interaction of neighbors on the sweep line) and inserted into the queue

Status

- Scan-line status
  - Stores information about the objects intersected by ℓ
  - It is updated while stopping on event point

Line segment intersection - Sweep line alg.

- Avoid testing of pairs of segments far apart
- Compute intersections of neighbors on the sweep line only
- O(n log n + I log n) time in O(n) memory
  - 2n steps for end points,
  - I steps for intersections,
  - log n search the status tree
- Ignore “nasty cases” (most of them will be solved later on)
  - No segment is parallel to the sweep line
  - Segments intersect in one point and do not overlap
  - No three segments meet in a common point

Line segment intersections

Status = ordered sequence of segments intersecting the sweep line ℓ

Stav

Events (waiting in the priority queue)

Postupový plán

= points, where the algorithm actually does something

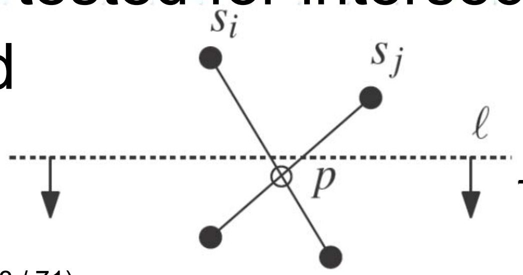
- Segment *end-points*
  - known at algorithm start
- Segment *intersections* between neighboring segments along SL
  - discovered as the sweep executes

Detecting intersections

- Intersection events must be detected and inserted to the event queue before they occur
- Given two segments *a, b* intersecting in point *p*, there must be a placement of sweep line ℓ prior to *p*, such that segments *a, b* are adjacent along ℓ (only adjacent will be tested for intersection)
  - segments *a, b* are not adjacent when the alg. starts
  - segments *a, b* are adjacent just before *p*

  => there must be an event point when *a,b* become adjacent and therefore are tested for intersection

  => All intersections are found

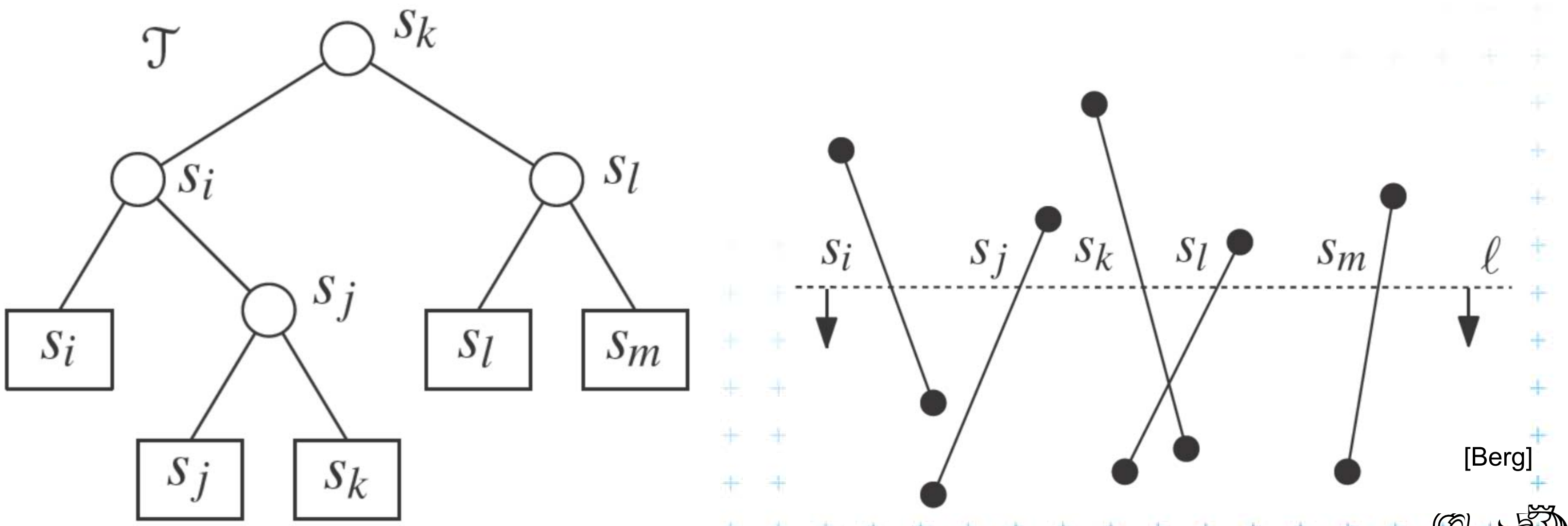


[Berg]

Data structures

Sweep line ℓ status = order of segments along ℓ

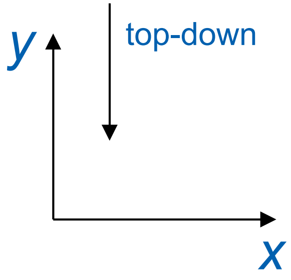
- Balanced binary search tree of segments
- Coords of intersections with ℓ vary as ℓ moves
  => store pointers to line segments in tree nodes
  - Position of ℓ is plugged in the *y=mx+b* to get the x-key



[Berg]

Data structures

Event queue (*postupový plán, časový plán*)



- Define: Order □ (top-down, lexicographic)

  p □ q **iff** $p_y > q_y$ **or** $p_y = q_y$ and $p_x < q_x$

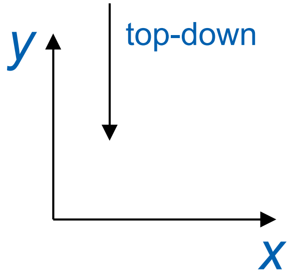
  top-down, left-right approach

  (points on ℓ treated left to right)

- Operations
  - Insertion of computed intersection points
  - Fetching the next event
    (highest *y* below ℓ or the leftmost right of e)
  - Test, if the segment is already present in the queue
    (Locate and delete intersection event in the queue)

Data structures

Event queue (*postupový plán, časový plán*)



- Define: Order □ (top-down, lexicographic)

  p □ q **iff** $p_y > q_y$ **or** $p_y = q_y$ and $p_x < q_x$

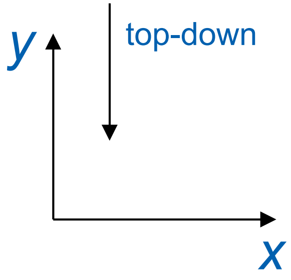
  top-down, left-right approach

  (points on ℓ treated left to right)

- Operations
  - Insertion of computed intersection points
  - Fetching the next event
    (highest *y* below ℓ or the leftmost right of e)

    must have
  - Test, if the segment is already present in the queue
    (Locate and delete intersection event in the queue)

Data structures



Event queue (*postupový plán, časový plán*)

- Define: Order □ (top-down, lexicographic)

  p □ q **iff** $p_y > q_y$ **or** $p_y = q_y$ and $p_x < q_x$

  top-down, left-right approach

  (points on ℓ treated left to right)

- Operations
  - Insertion of computed intersection points
  - Fetching the next event (highest *y* below ℓ or the leftmost right of e)

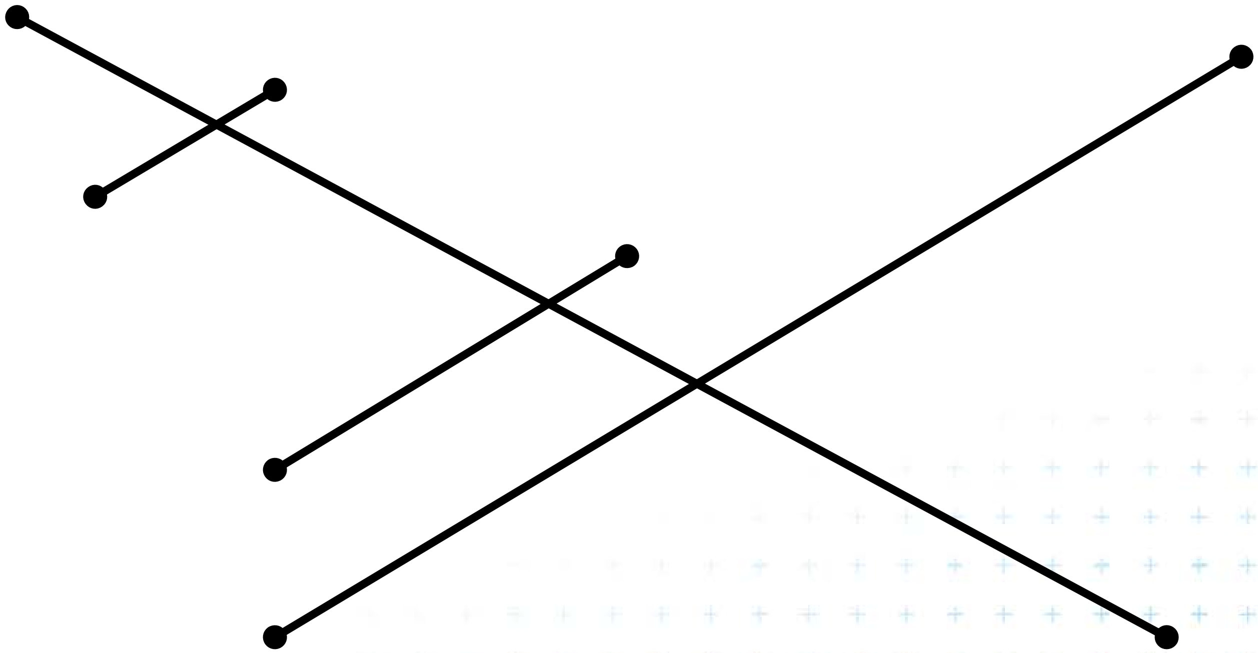
  } must have

  - Test, if the segment is already present in the queue (Locate and delete intersection event in the queue)

  } may have

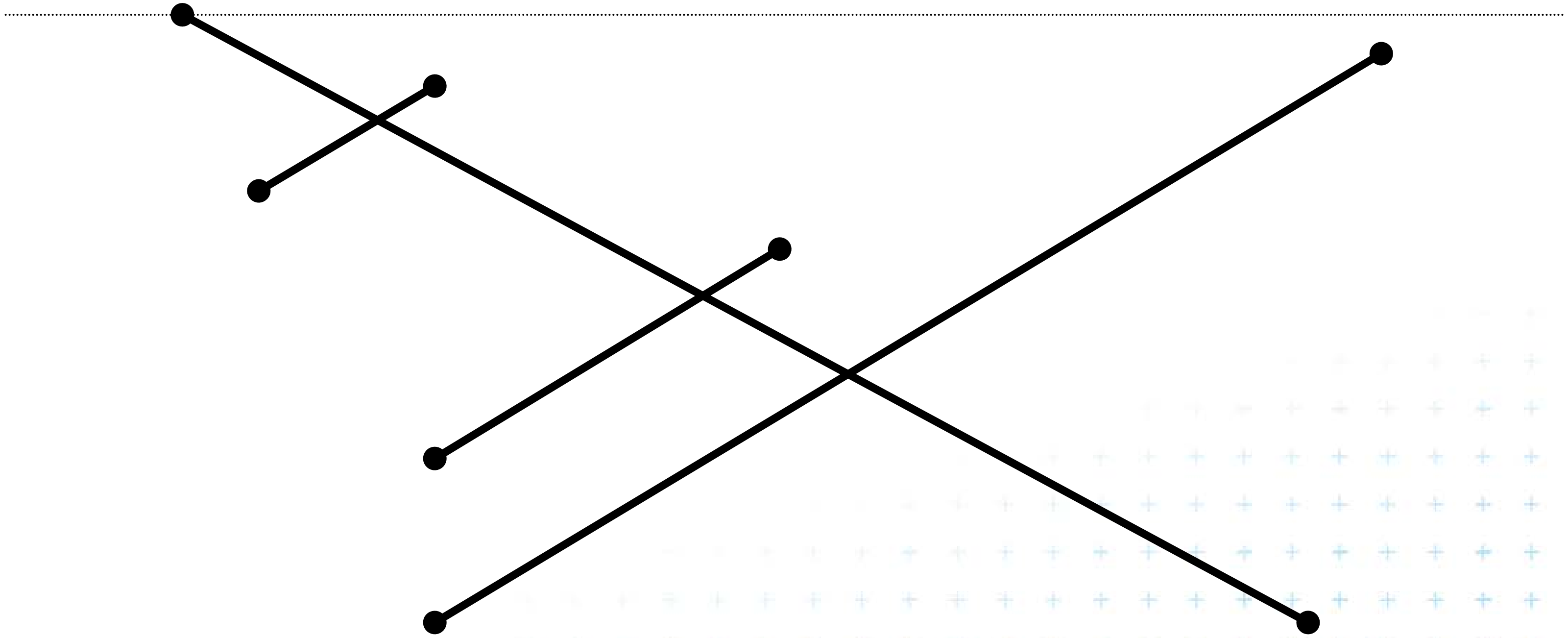
Problem with duplicities of intersections

Intersection may be detected many times



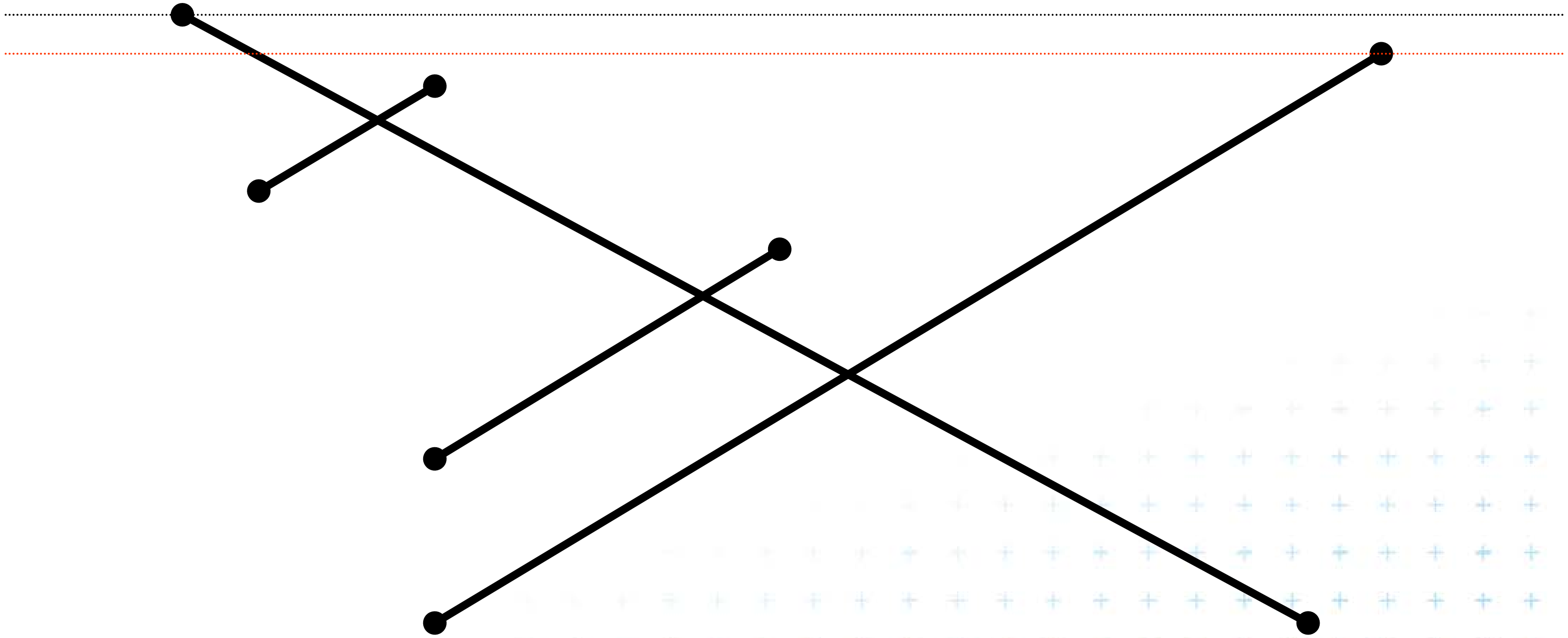
Problem with duplicities of intersections

Intersection may be detected many times



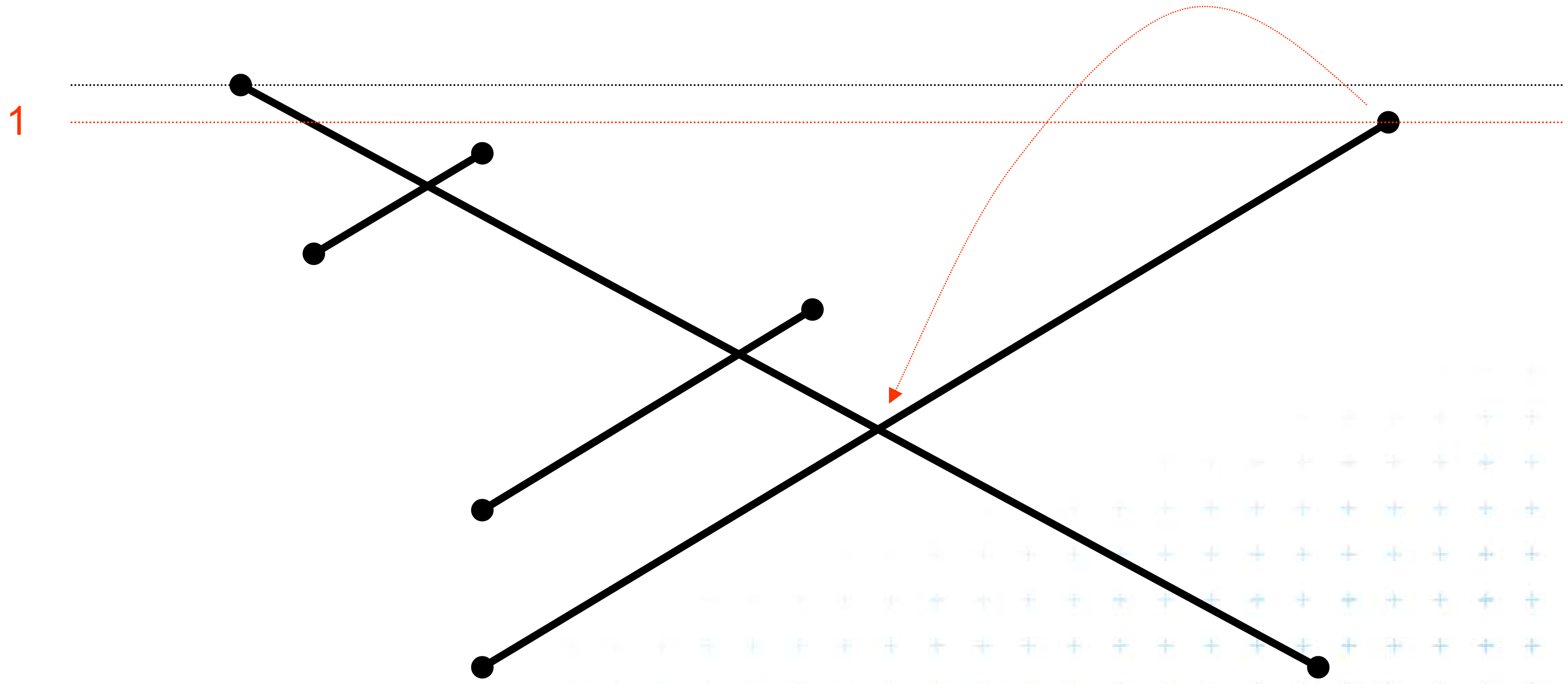
Problem with duplicities of intersections

Intersection may be detected many times



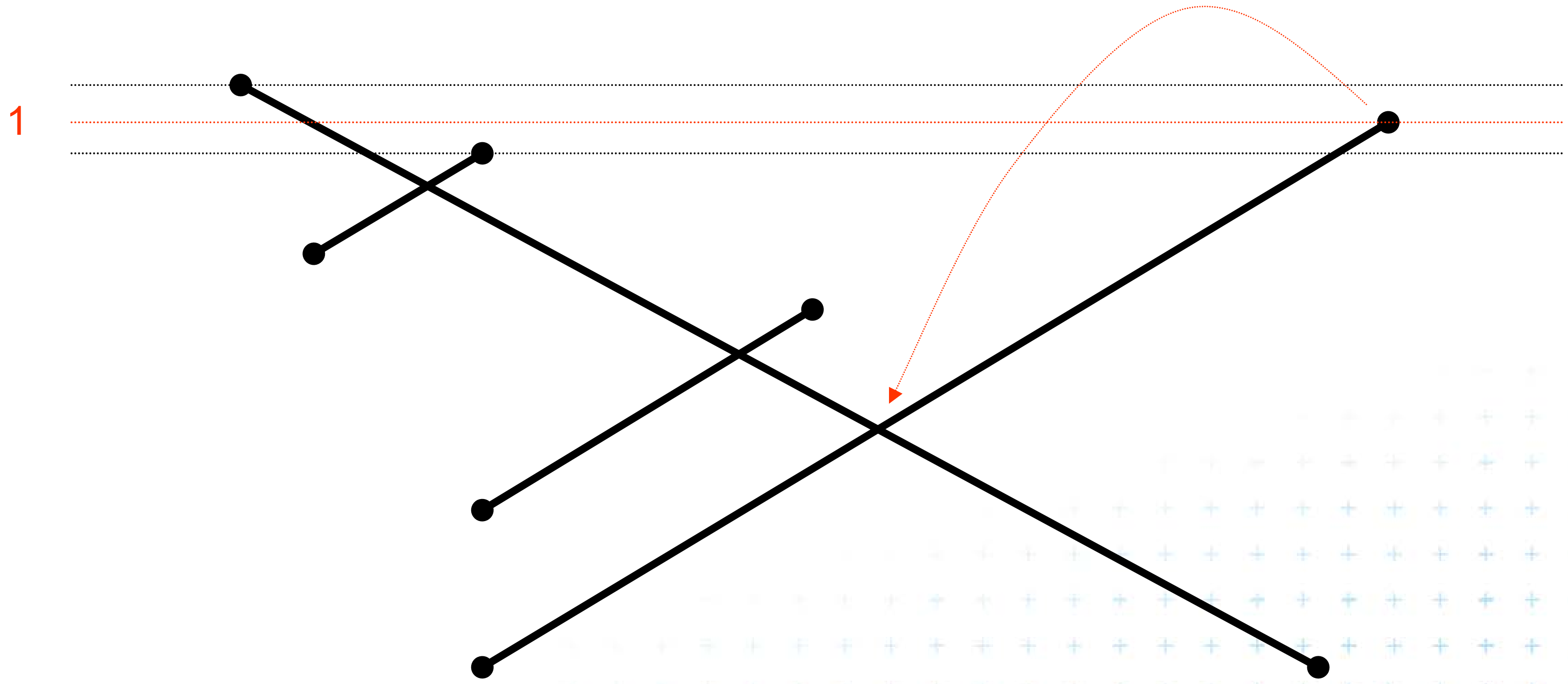
Problem with duplicities of intersections

Intersection may be detected many times



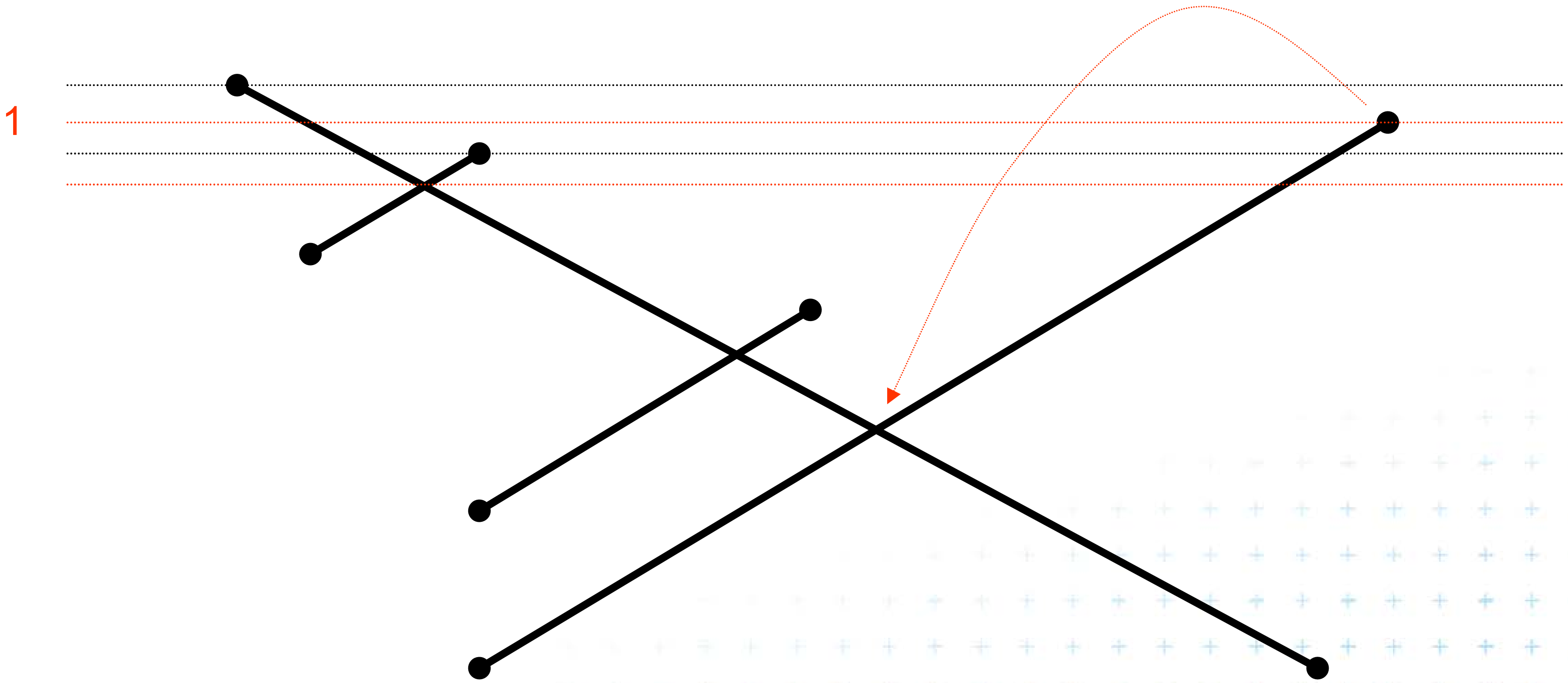
Problem with duplicities of intersections

Intersection may be detected many times



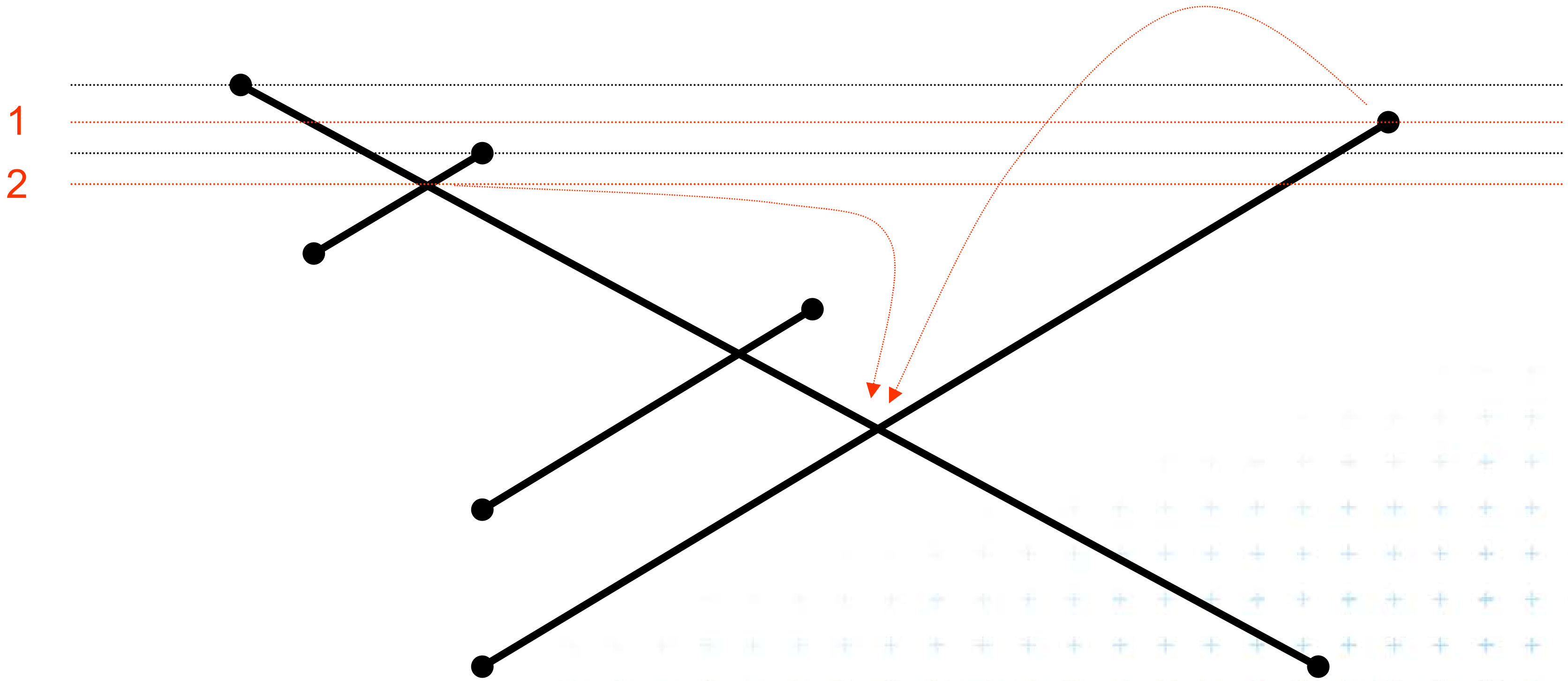
Problem with duplicities of intersections

Intersection may be detected many times



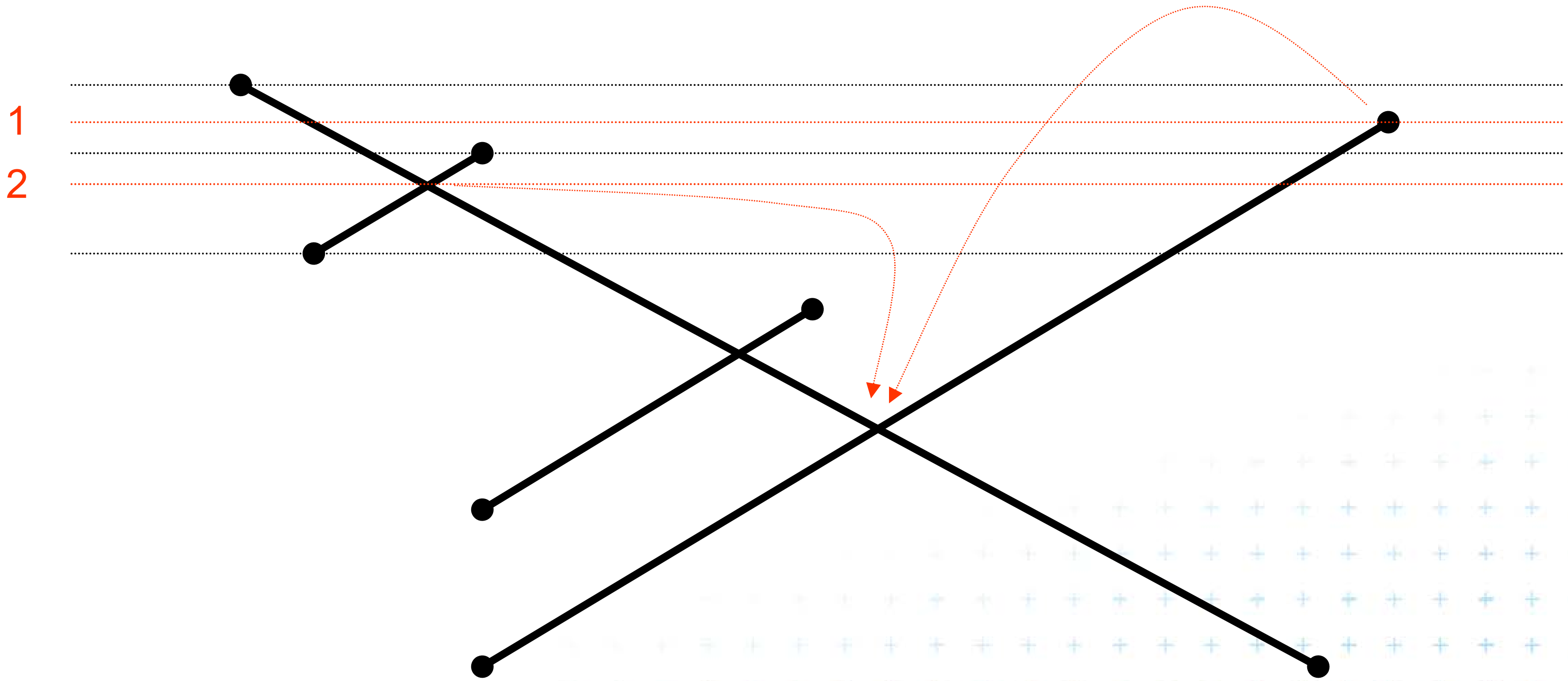
Problem with duplicities of intersections

Intersection may be detected many times



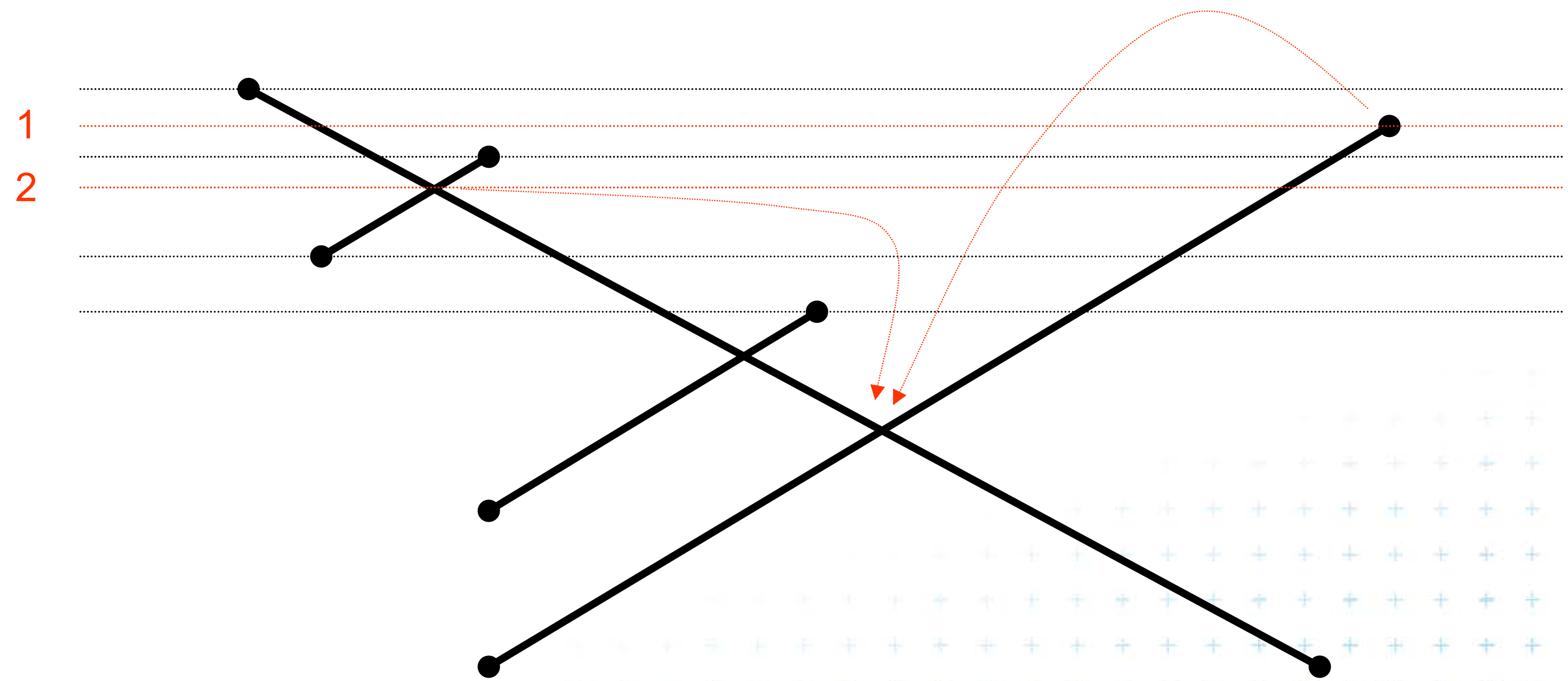
Problem with duplicities of intersections

Intersection may be detected many times



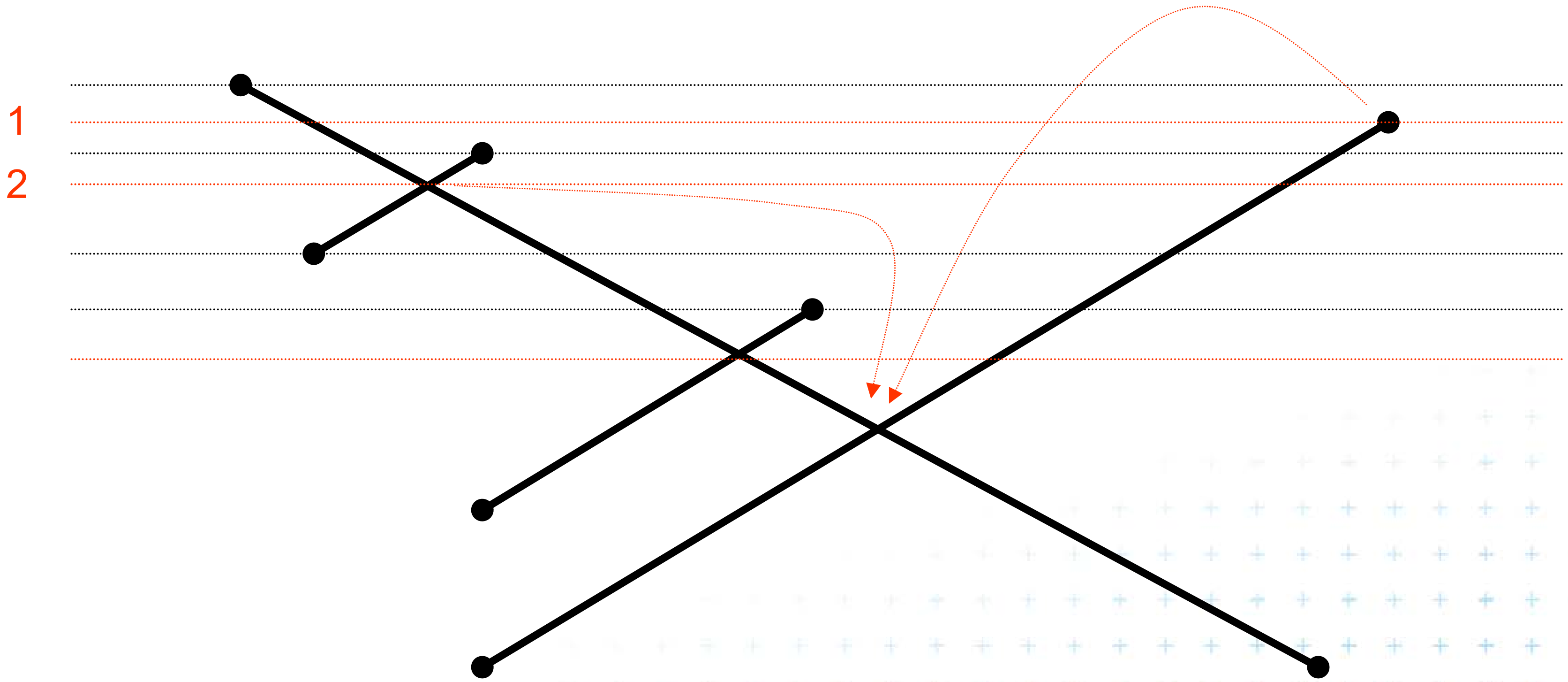
Problem with duplicities of intersections

Intersection may be detected many times



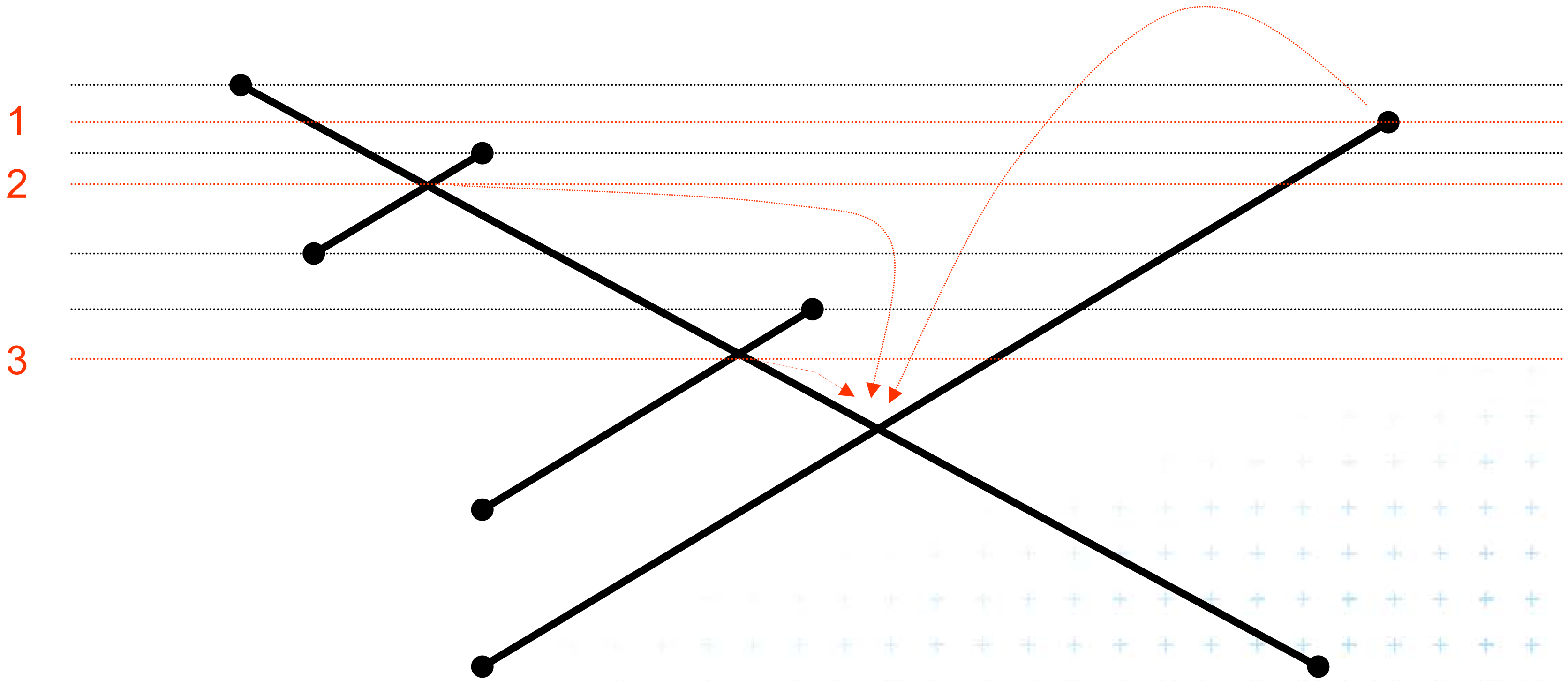
Problem with duplicities of intersections

Intersection may be detected many times



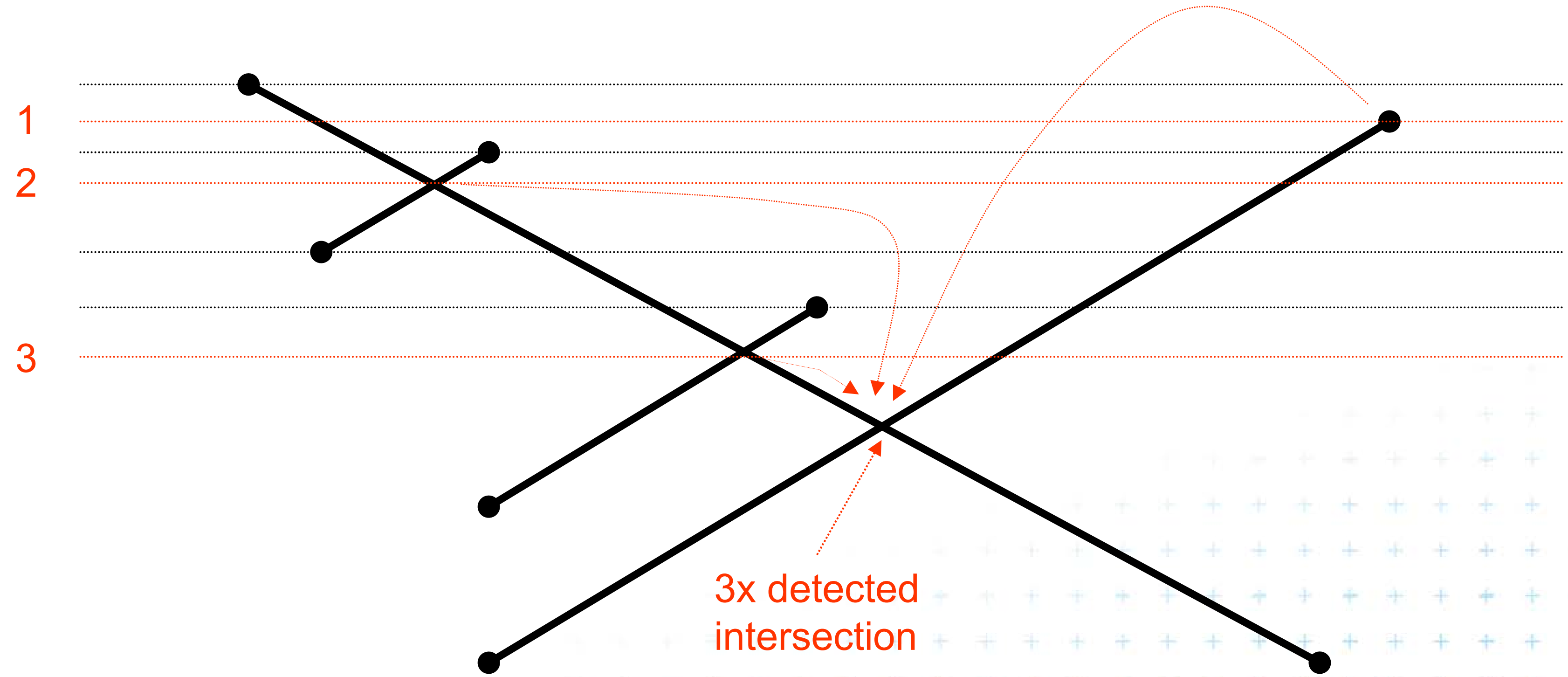
Problem with duplicities of intersections

Intersection may be detected many times



Problem with duplicities of intersections

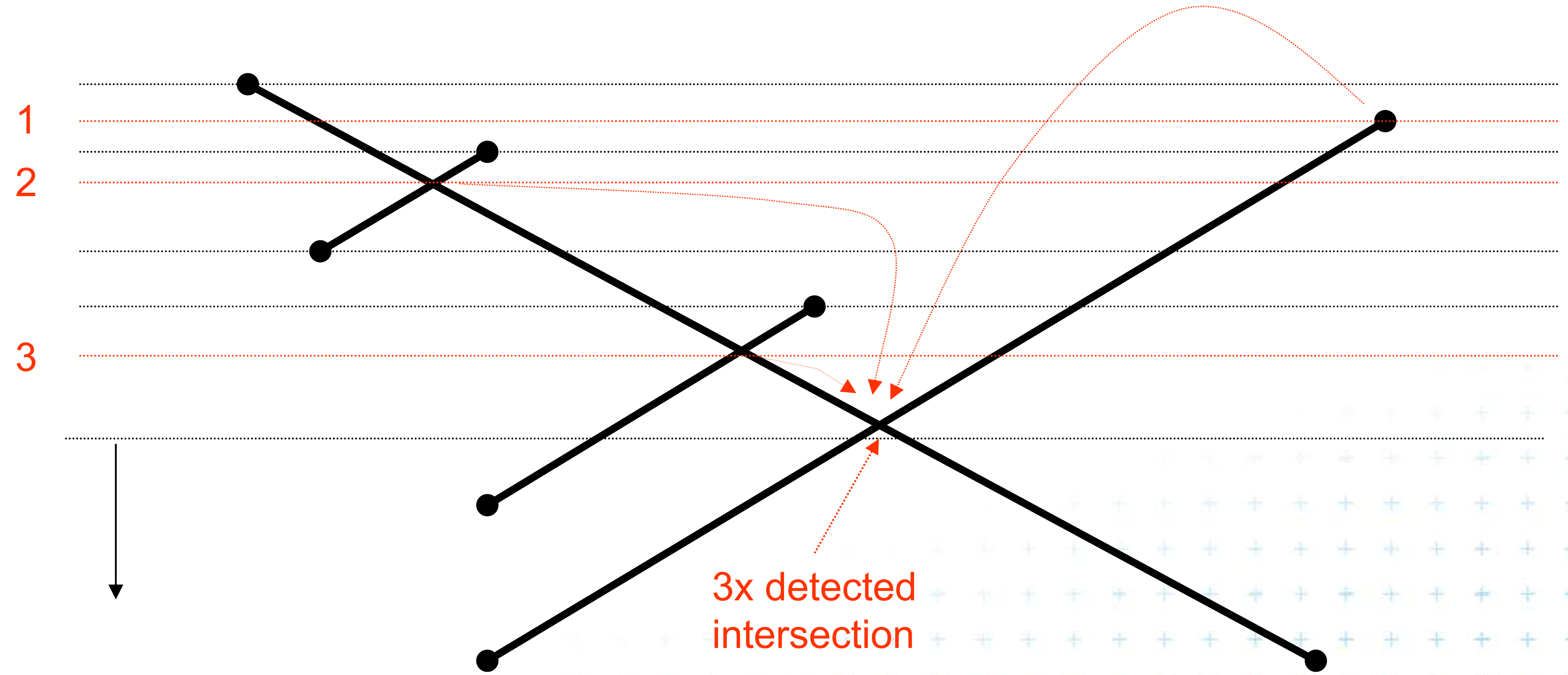
Intersection may be detected many times



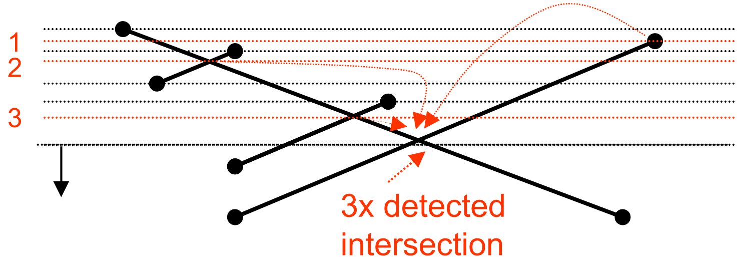
3x detected intersection

Problem with duplicities of intersections

Intersection may be detected many times



Data structures



Event queue data structure

a) Heap

- Problem: can not check duplicated intersection events (reinvented & stired more than once)
- Intersections processed twice or even more times
- Memory complexity up to O(n^2)

b) Ordered dictionary (balanced binary tree)

- Can check duplicated events (adds just constant factor)
- Nothing inserted twice
- If non-neighbor intersections are deleted i.e., if only intersections of neighbors along ℓ are stored then memory complexity just O(n)

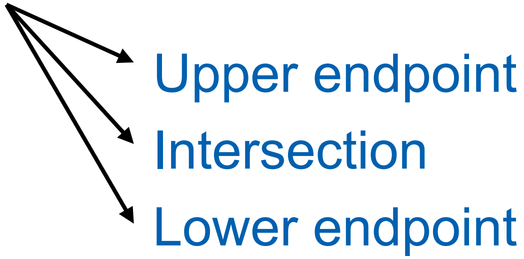
Line segment intersection algorithm

FindIntersections(*S*)

Input: A set *S* of line segments in the plane

Output: The set of intersection points + pointers to segments in each

```
init an empty event queue $Q$ and insert the segment endpoints
init an empty status structure $T$
while Q in not empty
    remove next event $p$ from $Q$
    handleEventPoint($p$)
```



Upper endpoint

Intersection

Lower endpoint

Note: Upper-end-point events store info about the segment

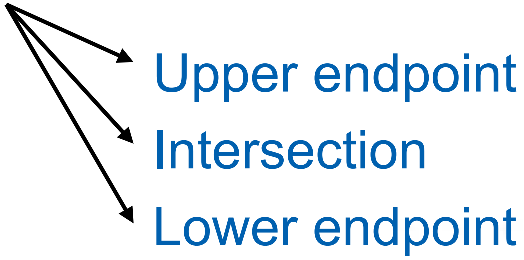
Line segment intersection algorithm

FindIntersections(*S*)

Input: A set *S* of line segments in the plane

Output: The set of intersection points + pointers to segments in each

```
init an empty event queue $Q$ and insert the segment endpoints
init an empty status structure $T$
while Q in not empty
    remove next event $p$ from $Q$
    handleEventPoint($p$)
```



Upper endpoint

Intersection

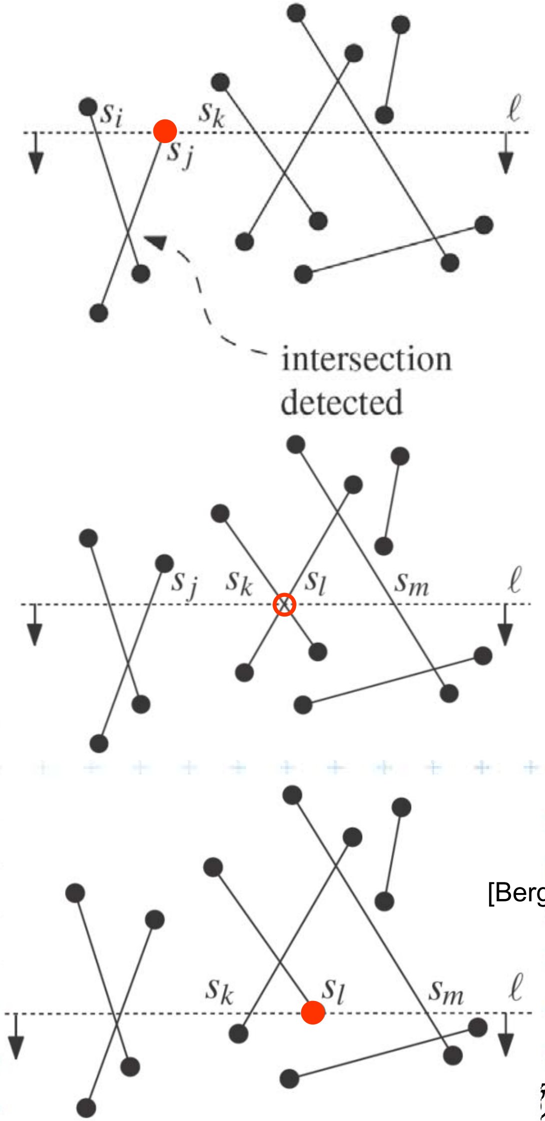
Lower endpoint

Improved algorithm:
Handles all in *p*
in a single step

Note: Upper-end-point events store info about the segment

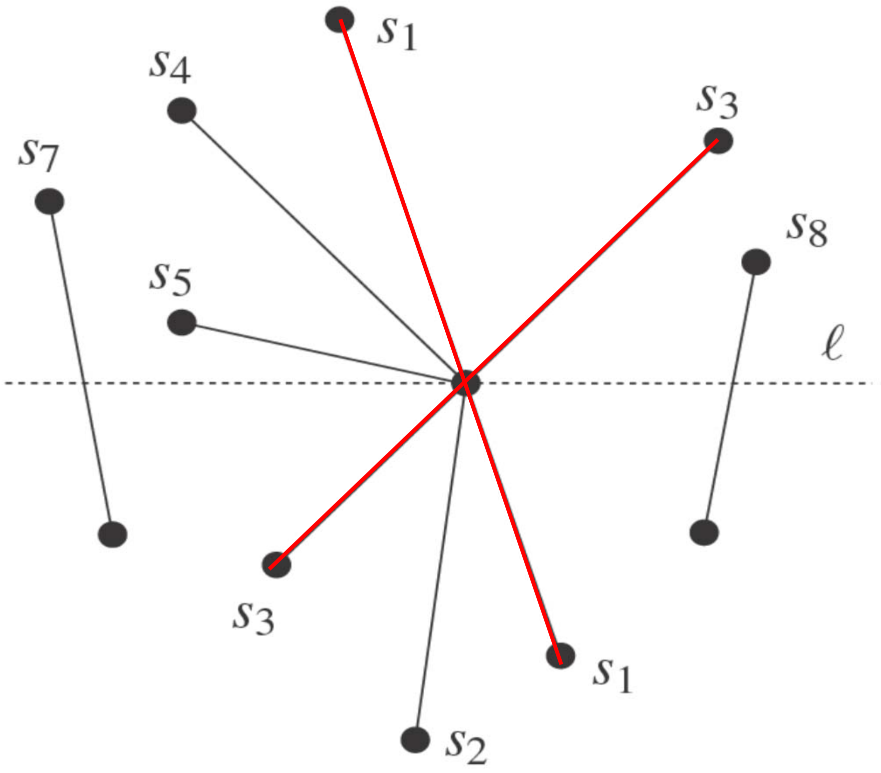
handleEventPoint() principle

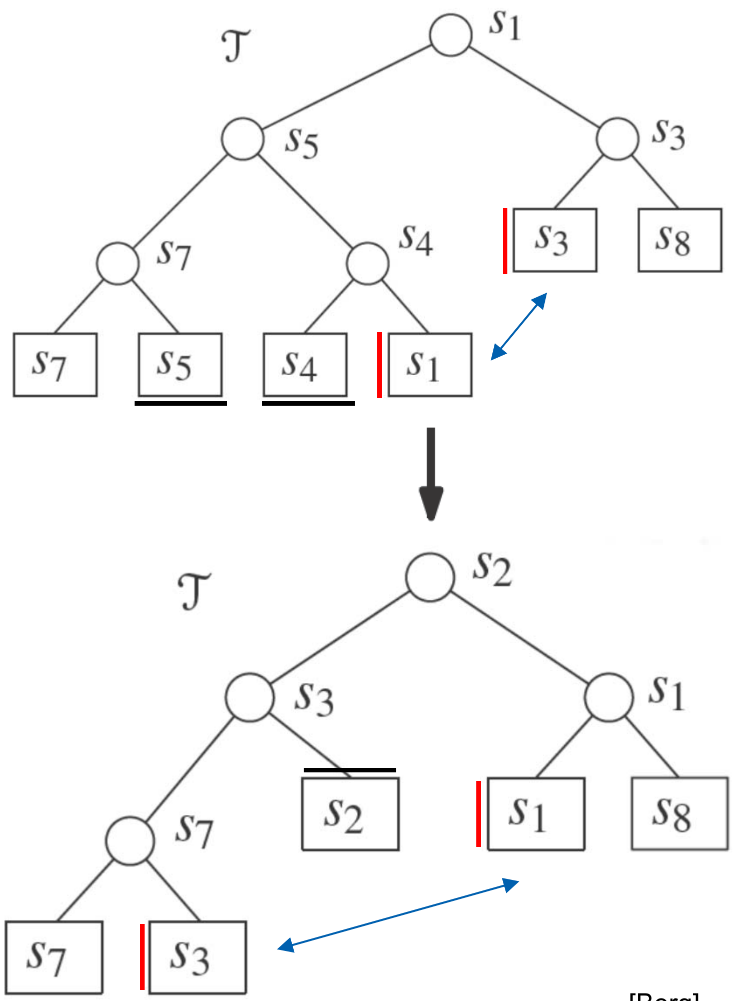
- Upper endpoint *U*(*p*)
  - insert *p* (on s_j) to status *T*
  - add intersections with left and right neighbors to *Q*
- Intersection *C*(*p*)
  - switch order of segments in *T*
  - add intersections with nearest left and nearest right neighbor to *Q*
- Lower endpoint *L*(*p*)
  - remove *p* (on s_l) from *T*
  - add intersections of left and right neighbors to *Q*



[Berg]

More than two segments incident





[Berg]

$U(p) = \{s_2\}$ start here

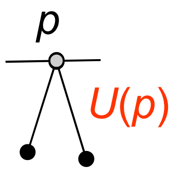
$C(p) = \{s_1, s_3\}$ cross on ℓ

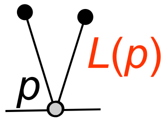
$L(p) = \{s_4, s_5\}$ end here

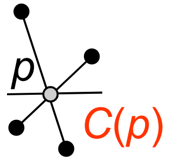
Handle Events [Berg, page 25]

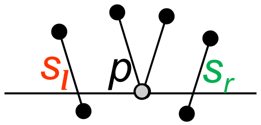
```
handleEventPoint(p)
1.  Let $U(p)$ = set of segments whose Upper endpoint is $p$.
    These segmets are stored with the event point $p$ (will be added to $T$)
2.  Search $T$ for all segments $S(p)$ that contain $p$ (are adjacent in $T$):
        Let $L(p) \subset S(p)$ = segments whose Lower endpoint is $p$
        Let $C(p) \subset S(p)$ = segments that Contain $p$ in interior
3.  if( $L(p) \cup U(p) \cup C(p)$ contains more than one segment )
4.      report $p$ as intersection ∘ together with $L(p)$, $U(p)$, $C(p)$
5.  Delete the segments in $L(p) \cup C(p)$ from $T$
6.  Insert  the segments in $U(p) \cup C(p)$ into $T$
    (order as below $\ell$, horizontal segment as the last)
7.  if( $U(p) \cup C(p) = \emptyset$ ) then findNewEvent($s_l$, $s_r$, $p$) // left & right neighbors
8.  else  s'  = leftmost segment of $U(p) \cup C(p)$;   findNewEvent($s_l$ , s', $p$)
          s'' = rightmost segment of $U(p) \cup C(p)$; findNewEvent(s'', $s_r$ , $p$)
```

Reverse order of $C(p)$ in T







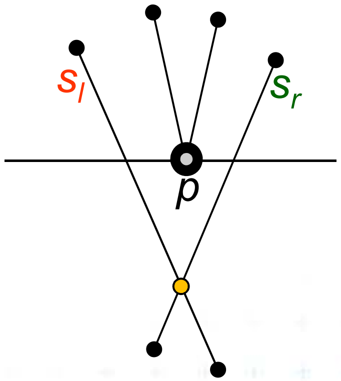


Detection of new intersections

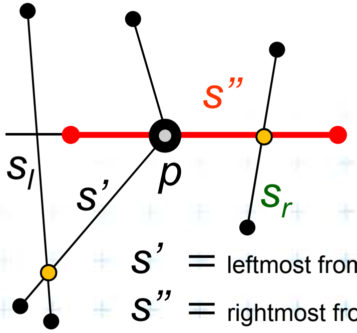
```
findNewEvent($s_l$, $s_r$, $p$)    // with handling of horizontal segments
Input:    two segments (left & right from $p$ in $T$) and a current event point $p$
Output:   updated event queue $Q$ with new intersection
1.  if [ ( $s_l$ and $s_r$ intersect below the sweep line $\ell$ ) // line 7. above
                                                          Non-overlapping
        or ($s_r$ intersect $s''$ on $\ell$ and to the right of $p$ ) ]  // horizontal segm.
        and( the intersection ● is not present in $Q$ )
2.  then
        insert intersection ● as a new event into $Q$
```

○ Intersection - line 4

● Intersection - line 7,8

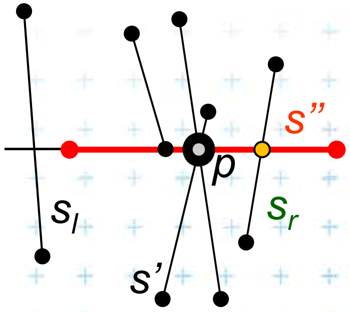


s_l and s_r intersect below



s' = leftmost from $\mathrm{U(p)} \cup \mathrm{C(p)}$

s'' = rightmost from $\mathrm{U(p)} \cup \mathrm{C(p)}$



s_r and s'' intersect on ℓ,

s'' is horizontal and to the right of p

Line segment intersections

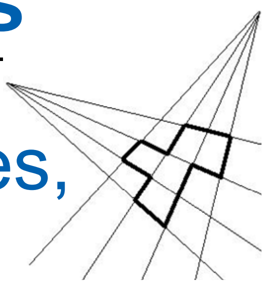
- Memory O(I) = O(n^2) with duplicities in Q
  or O(n) with duplicities in Q deleted
- Operational complexity
  - $n + I$ stops
  - log n each

  => O($I + n$) log n total
- The algorithm is by Bentley-Ottmann

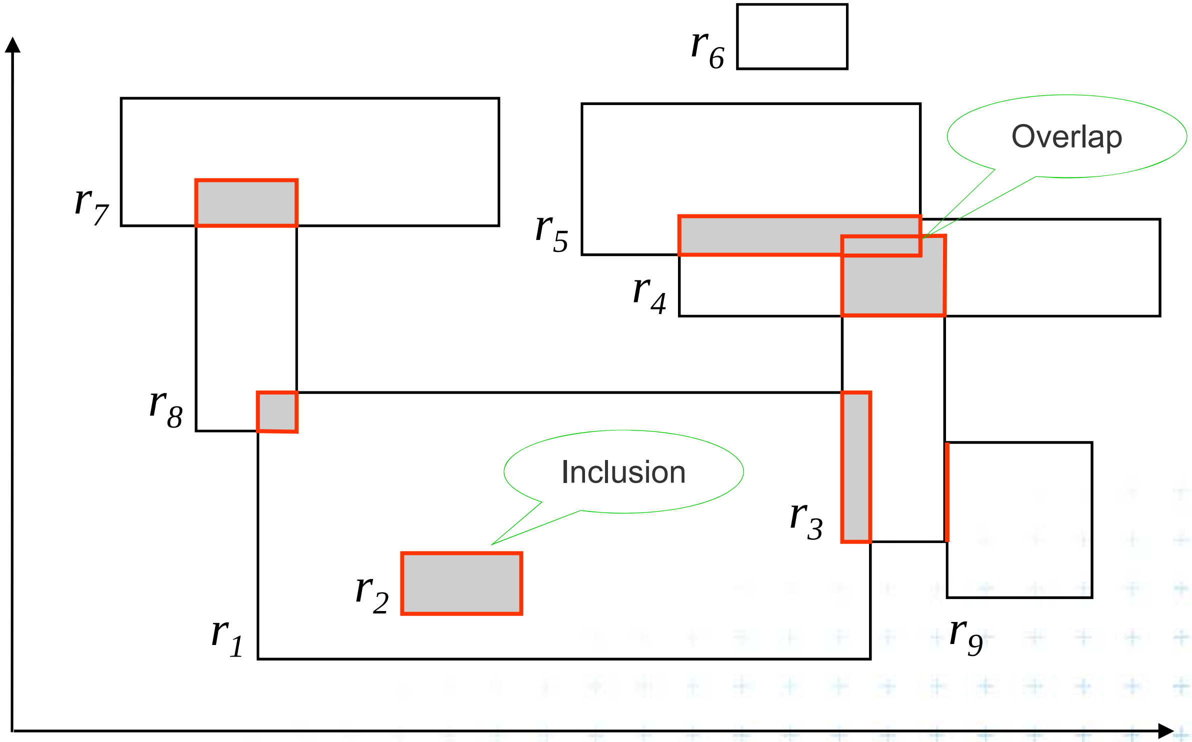
  Bentley, J. L.; Ottmann, T. A. (1979), "Algorithms for reporting and counting geometric intersections", *IEEE Transactions on Computers* **C-28** (9): 643-647, doi:10.1109/TC.1979.1675432 .

  See also http://wapedia.mobi/en/Bentley%E2%80%93Ottmann_algorithm

Intersection of axis parallel rectangles



- Given the collection of *n isothetic* rectangles, report all intersecting parts



Alternate sides belong to two pencils of lines
(trsy přímek)
(often used with points in infinity = axis parallel)
2D => 2 pencils

[?]

Answer: (r_1, r_2) (r_1, r_3) (r_1, r_8) (r_3, r_4) (r_3, r_5) (r_3, r_9) (r_4, r_5) (r_7, r_8)

Brute force intersection

Brute force algorithm

Input: set *S* of axis parallel rectangles
Output: pairs of intersected rectangles

```
1. For every pair $(r_i, r_j)$ of rectangles $\in S$, $i \neq j$
2.     if $(r_i \cap r_j \neq \emptyset)$ then
3.         report $(r_i, r_j)$
```

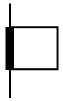
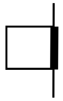
Analysis

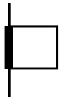
Preprocessing: None.

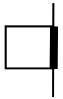
Query: $O(N^2)$ $\binom{N}{2} = \frac{N(N-1)}{2} \in O(N^2)$.

Storage: $O(N)$

Plane sweep intersection algorithm

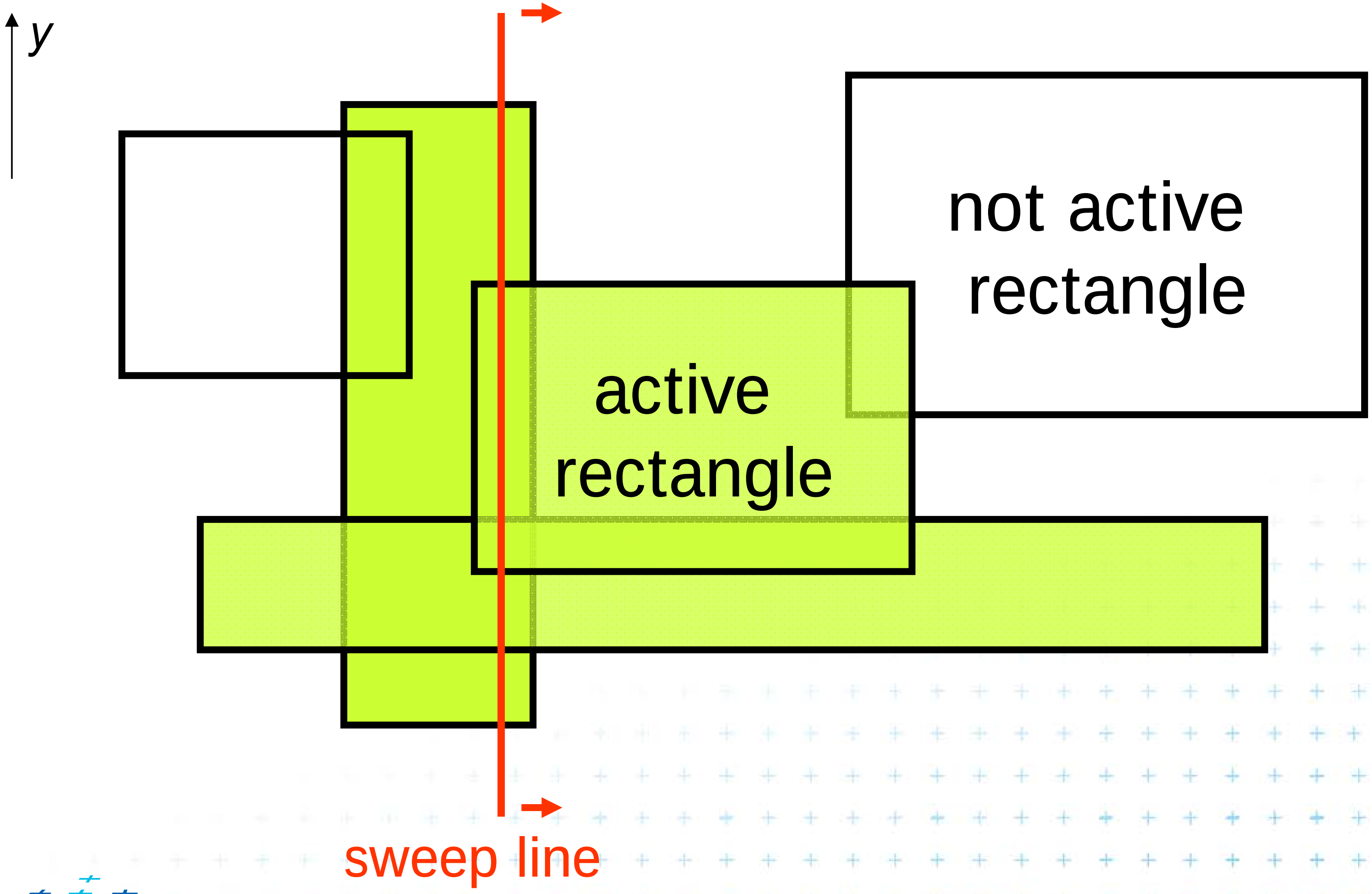
- Vertical sweep line moves from left to right
- Stops at every x-coordinate of a rectangle (either at its left side or at its right side).  
- active rectangles – a set

  = rectangles currently intersecting the sweep line
  - left side event of a rectangle  – start

    => the rectangle is added to the active set.
  - right side  – end

    => the rectangle is deleted from the active set.
- The active set used to detect rectangle intersection

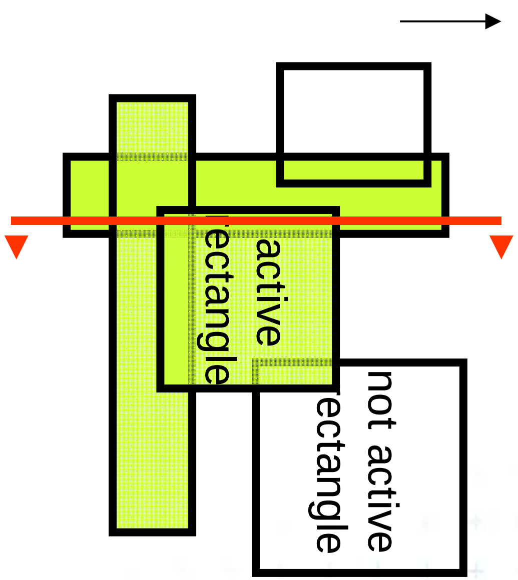
Example rectangles and sweep line



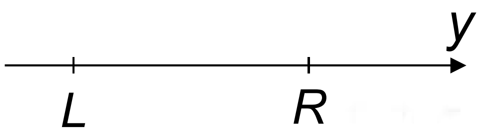
[Drtina]

Interval tree as sweep line status structure

- Vertical sweep-line => only *y*-coordinates along it
- The status tree is drawn horizontal - turn 90° right as if the sweep line (*y*-axis) is horizontal



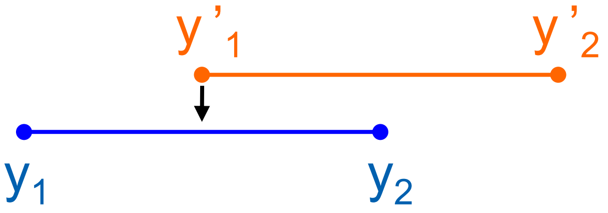
sweep line [Drtina]



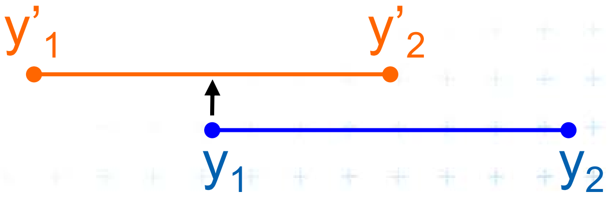
Intersection test – between pair of intervals

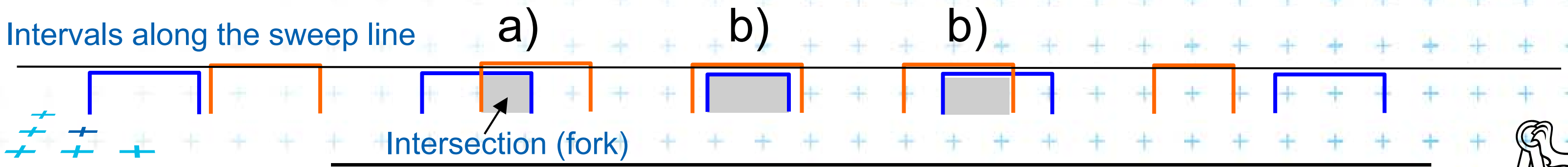
- Given two intervals $R = [y_1, y_2]$ and $R' = [y'_1, y'_2]$ the condition $R \cap R'$ is equivalent to one of these mutually exclusive conditions:

a) $y_1 \leq y'_1 \leq y_2$



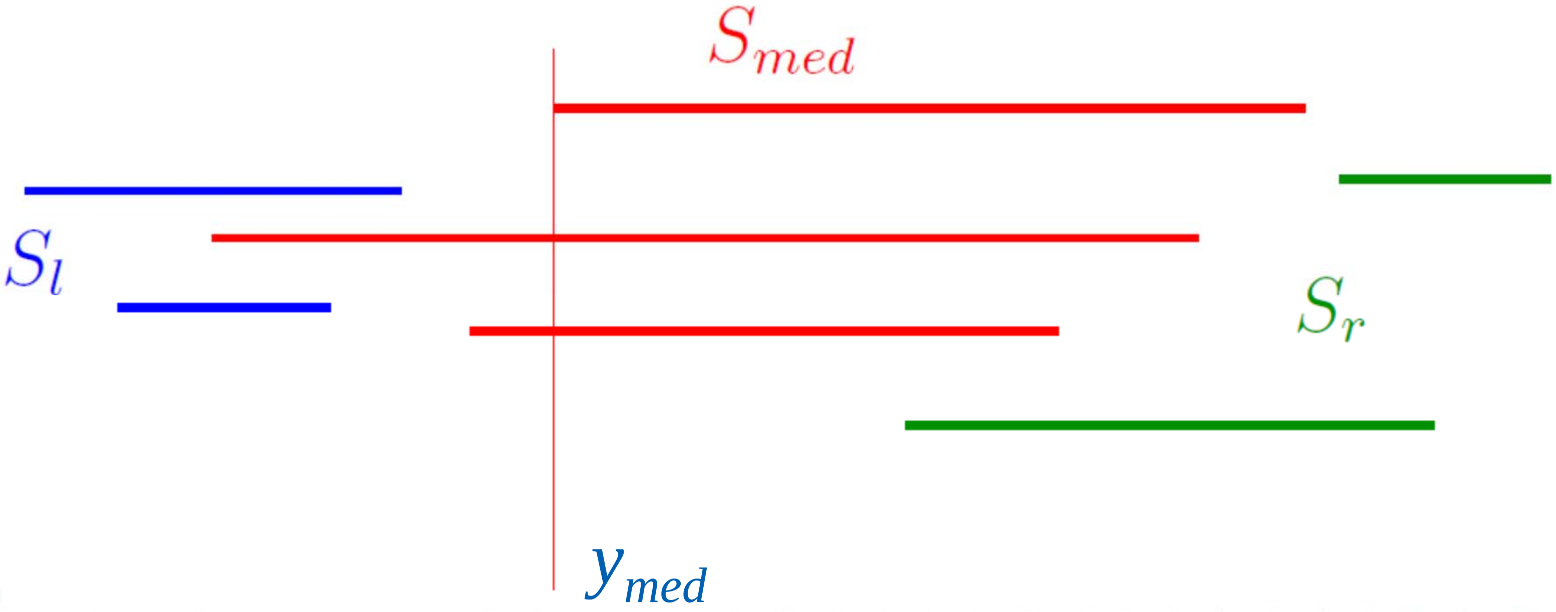
b) $y'_1 \leq y_1 \leq y'_2$





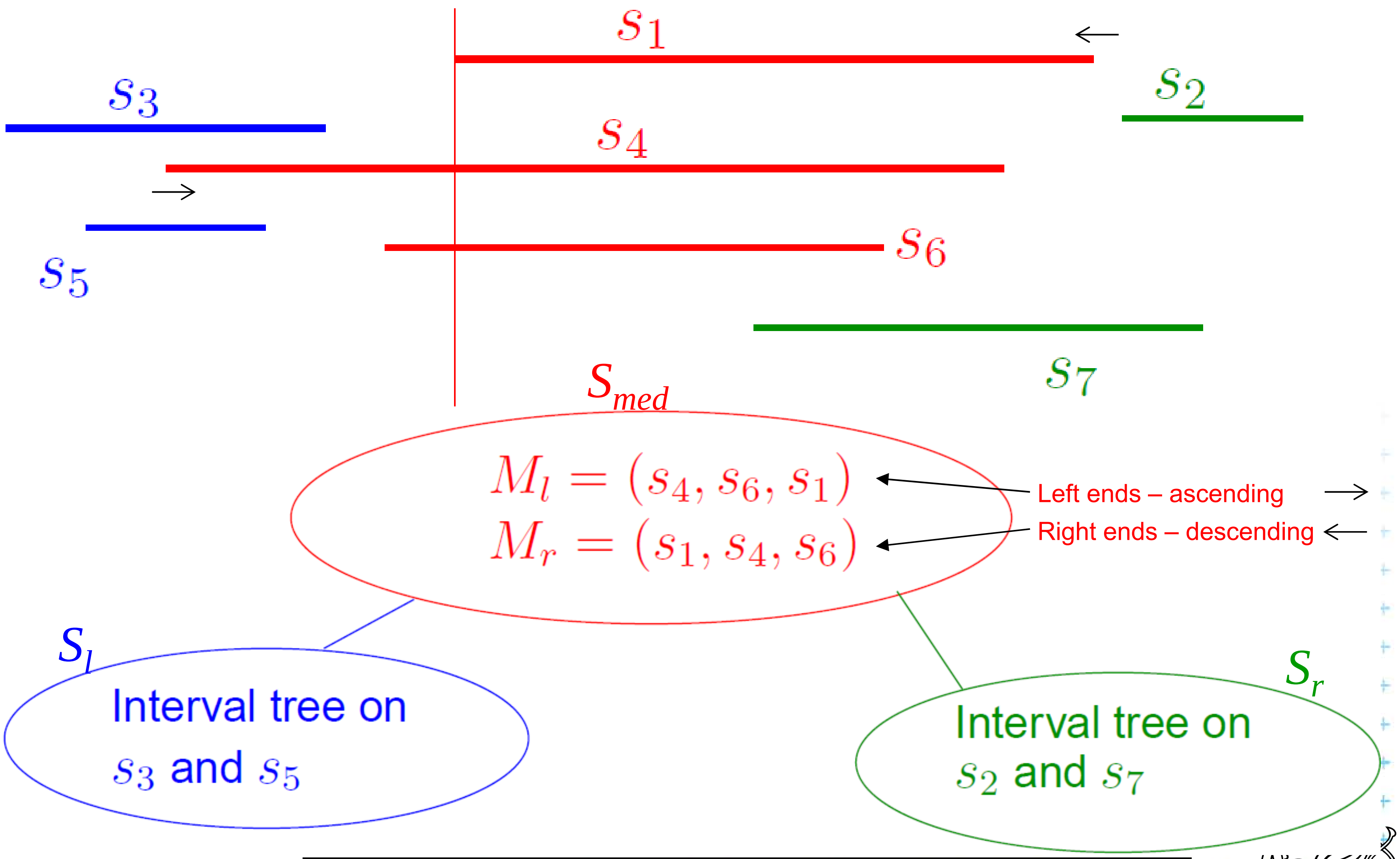
Static interval tree – stores all end points

- Let *v* = y_{med} be the median of end-points of segments
- S_l : segments of S that are completely to the left of y_{med}
- S_{med} : segments of S that contain $\mathrm{y}_{\mathrm{med}}$
- S_r : segments of S that are completely to the right of $\mathrm{y}_{\mathrm{med}}$



[Vigneron]

Static interval tree – Example



$M_l = (s_4, s_6, s_1)$

$M_r = (s_1, s_4, s_6)$

Left ends – ascending →

Right ends – descending ←

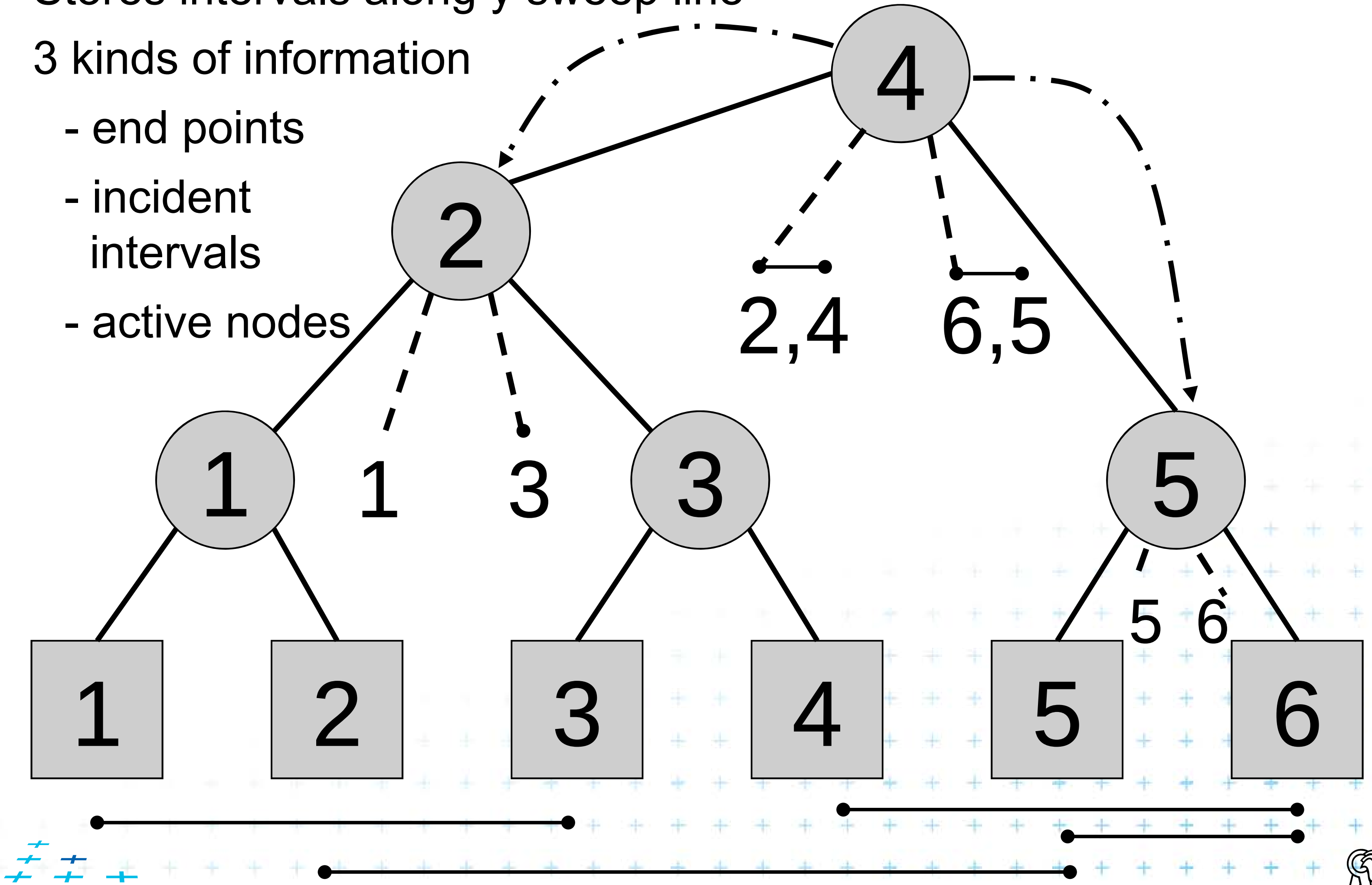
Interval tree on s_3 and s_5

Interval tree on s_2 and s_7

[Vigneron]

Static interval tree [Edelsbrunner80]

- Stores intervals along y sweep line
- 3 kinds of information
  - end points
  - incident intervals
  - active nodes

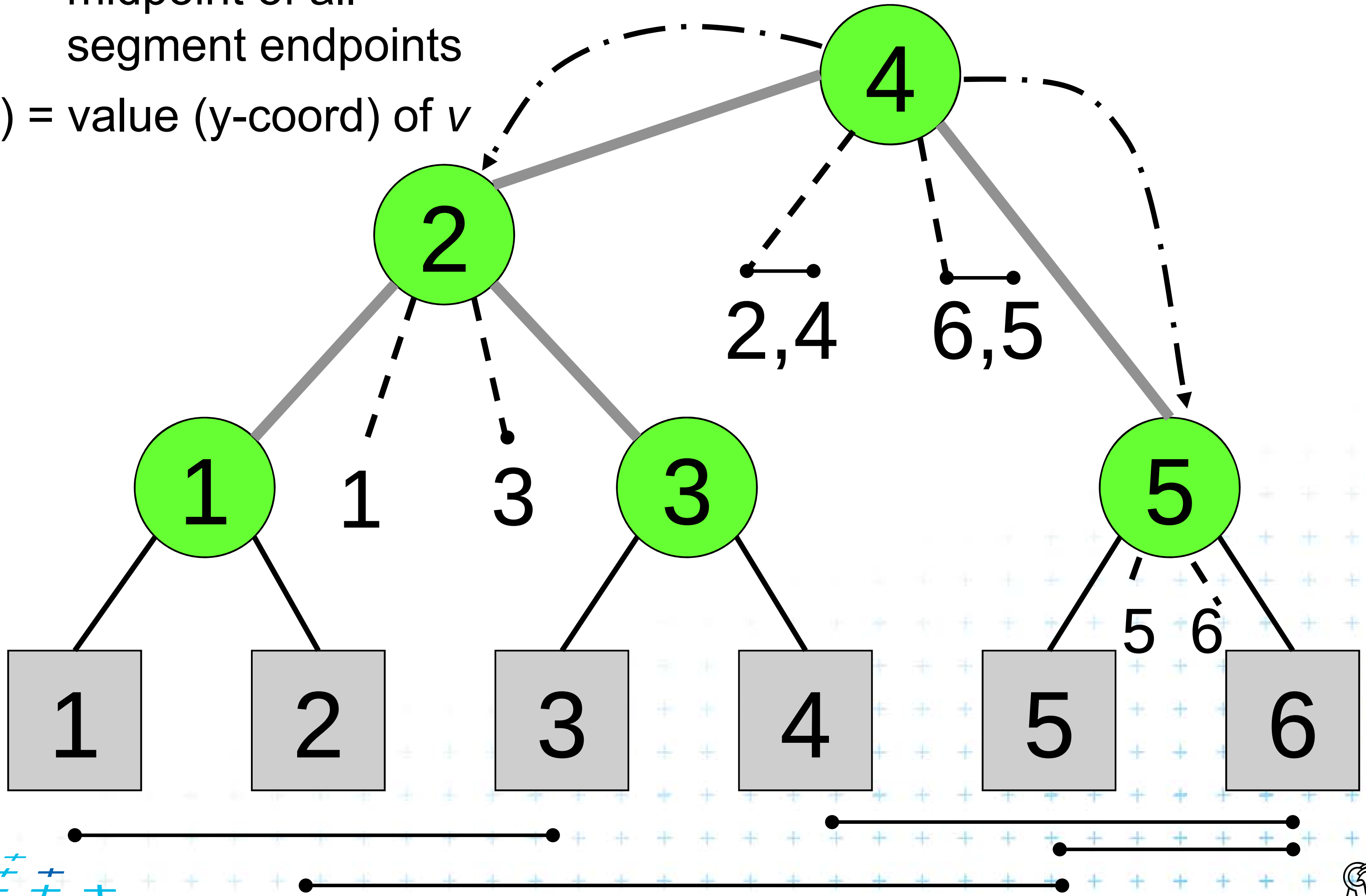


[Kukral]

Primary structure – static tree for endpoints

v = midpoint of all segment endpoints

H(v) = value (y-coord) of *v*

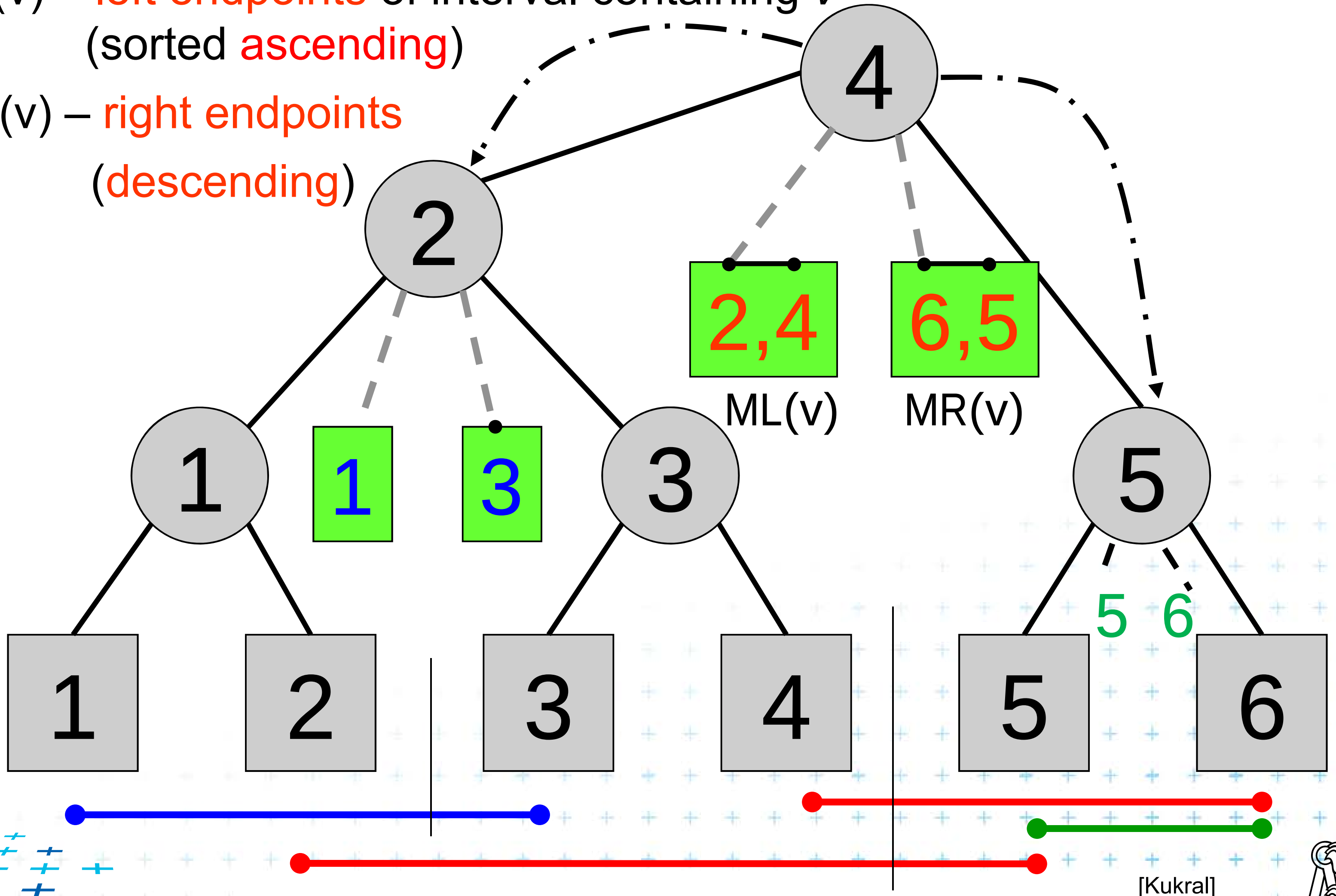


[Kukral]

Secondary lists of incident interval end-pts.

ML(v) – left endpoints of interval containing *v*
(sorted ascending)

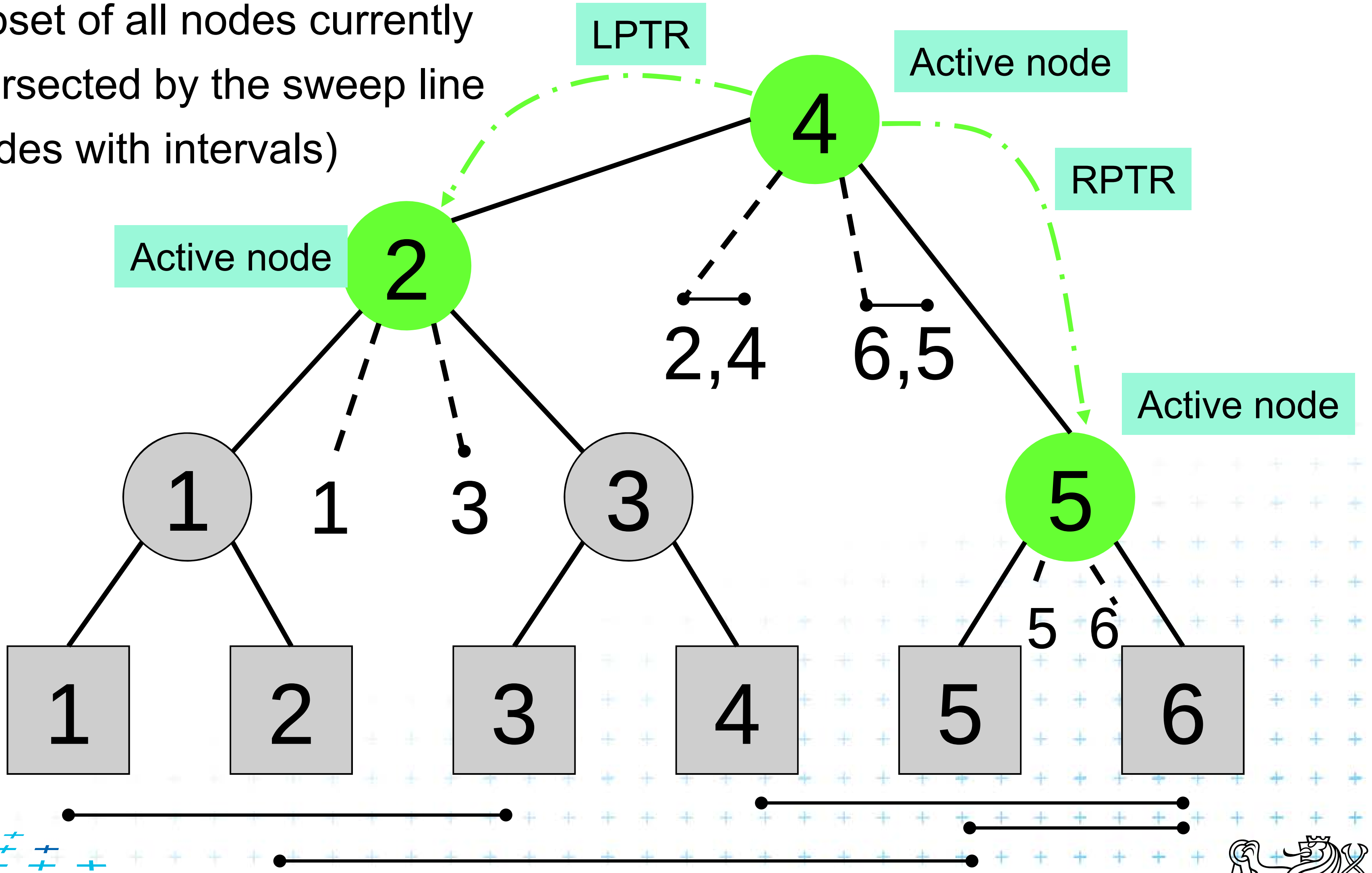
MR(v) – right endpoints
(descending)



[Kukral]

Active nodes – intersected by the sweep line

Subset of all nodes currently intersected by the sweep line (nodes with intervals)



[Kukral]

Query = sweep and report intersections

RectangleIntersections(*S*)
Input: Set *S* of rectangles
Output: Intersected rectangle pairs

```
Preprocess( S )            // create the interval tree $T$ (for y-coords)
                           // and event queue $Q$         (for x-coords)
while ( $Q \neq \emptyset$ ) do
    Get next entry ($x_i, y_{il}, y_{ir}, t$) from $Q$      // $t \in \{ left \mid right \}$
    if ( $t$ = left )    // left edge
            a) QueryInterval ( $y_{il}, y_{ir}$, root($T$))   // report intersections
            b) InsertInterval ( $y_{il}, y_{ir}$, root($T$))  // insert new interval
    else                 // right edge
            c) DeleteInterval ( $y_{il}, y_{ir}$, root($T$) )
```

Preprocessing

Preprocess(S)

Input: Set *S* of rectangles

Output: Primary structure of the interval tree *T* and the event queue *Q*

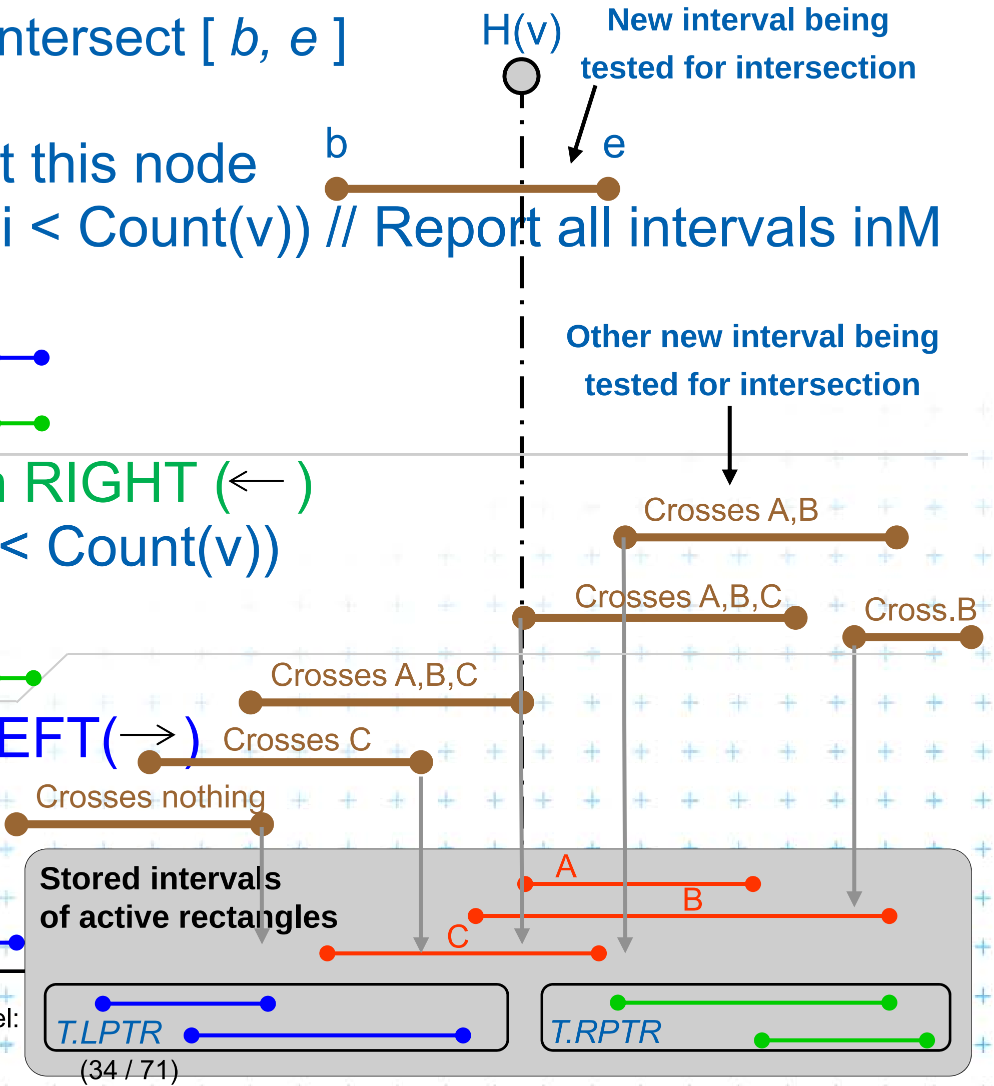
```
$T$ = PrimaryTree($S$)        // Construct the static primary structure
                              // of the interval tree -> sweep line STATUS $T$

// Init event queue Q with vertical rectangle edges in ascending order ~x
// Put the left edges with the same $x$ ahead of right ones
for $i$ = 1 to $n$
    insert( ( $x_{il}$ , $y_{il}$ , $y_{ir}$ , left ), Q)        // left edges of i-th rectangle
    insert( ( $x_{ir}$, $y_{il}$ , $y_{ir}$ , right ), Q)        // right edges
```

Interval tree – primary structure construction

```
PrimaryTree($S$)        // only the y-tree structure, without intervals
Input:   Set $S$ of rectangles
Output:  Primary structure of an interval tree $T$
    $S_y$ = Sort endpoints of all segments in $S$ according to $y$-coordinate
    $T$ = BST( $S_y$ )
    return $T$

BST( $S_y$ )
    if( $|S_y| = 0$ ) return null
    $yMed$ = median of $S_y$        // the smaller item for even $S_y$.size
    $L$ = endpoints $p_y \leq yMed$
    R = endpoints $p_y > yMed$
    $t$ = new IntervalTreeNode( $yMed$ )
    $t.left$  = BST($L$)
    $t.right$ = BST($R$)
    return $t$
```

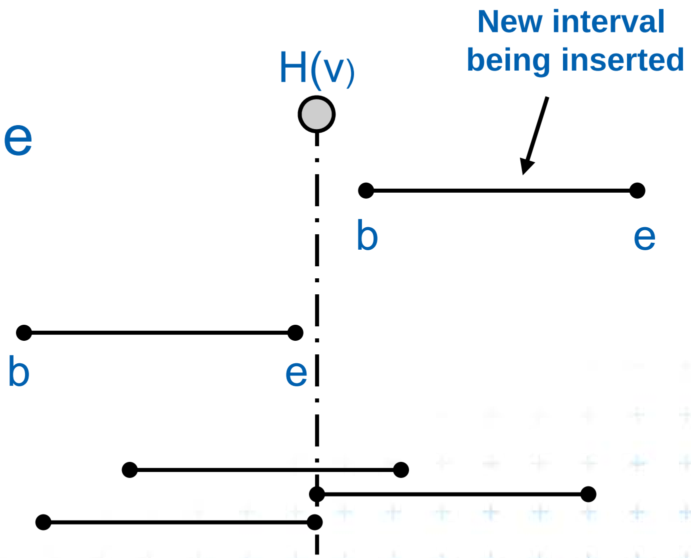
Interval tree – search the intersections

```
QueryInterval ( b, e, T )
Input:    Interval of the edge and current tree T
Output:   Report the rectangles that intersect [ b, e ]
if( T =  null ) return
i=0; if( b < H(v) < e )  // forks at this node
    while ~~( MR(v).[i] >= b ) &&~~ (i < Count(v)) // Report all intervals inM
        ReportIntersection; i++
    QueryInterval( b,e,T.LPTR ) ●—●
    QueryInterval( b,e,T.RPTR ) ●—●
else if (H(v) ≤ b < e)  // search RIGHT (← )
    while (MR(v).[i] >= b) && (i < Count(v))
        ReportIntersection; i++
    QueryInterval( b,e,T.RPTR ) ●—●
else  // b < e ≤ H(v) //search LEFT(→ )
    while (ML(v).[i] <= e)
        ReportIntersection; i++
    QueryInterval( b,e,T.LPTR ) ●—●
```


H(v)
New interval being tested for intersection
b
e
Other new interval being tested for intersection
Crosses A,B
Crosses A,B,C
Cross.B
Crosses A,B,C
Crosses C
Crosses nothing
Stored intervals of active rectangles
A
B
C
T.LPTR
T.RPTR

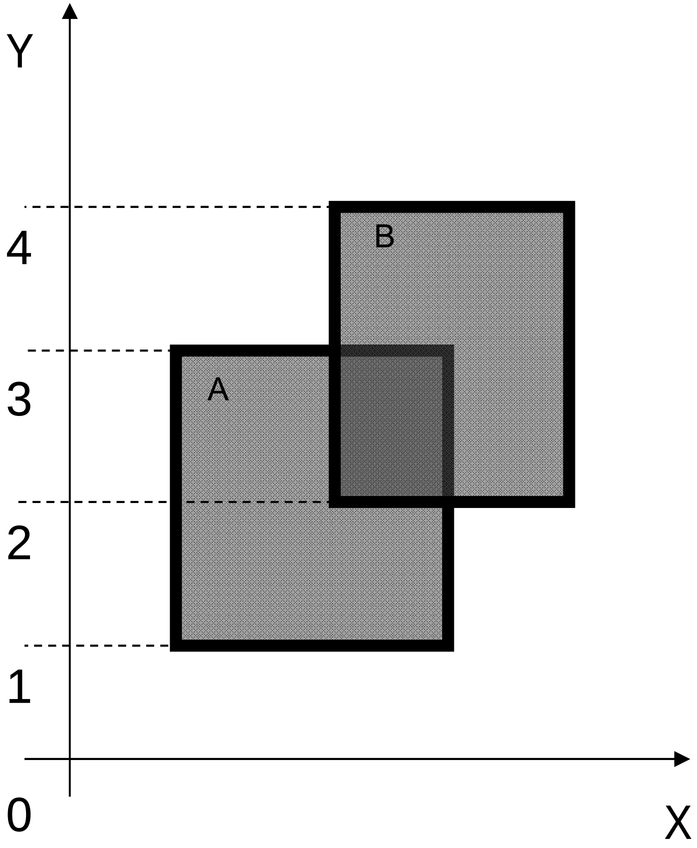
Interval tree - interval insertion

```
InsertInterval ( b, e, T )
Input:    Interval [b,e] and interval tree T
Output:   T after insertion of the interval
v = root(T )
while( v != null )          // find the fork node
    if (H(v) < b < e)
        v = v.right         // continue right
    else if (b < e < H(v))
        v = v.left          // continue left
    else  // b ≤ H(v) ≤ e  // insert interval
        set v node to active
        connect LPTR resp. RPTR to its parent
        insert [b,e] into list ML(v) – sorted in ascending order of b's
        insert [b,e] into list MR(v) – sorted in descending order of e's
        break
endwhile
return T
```

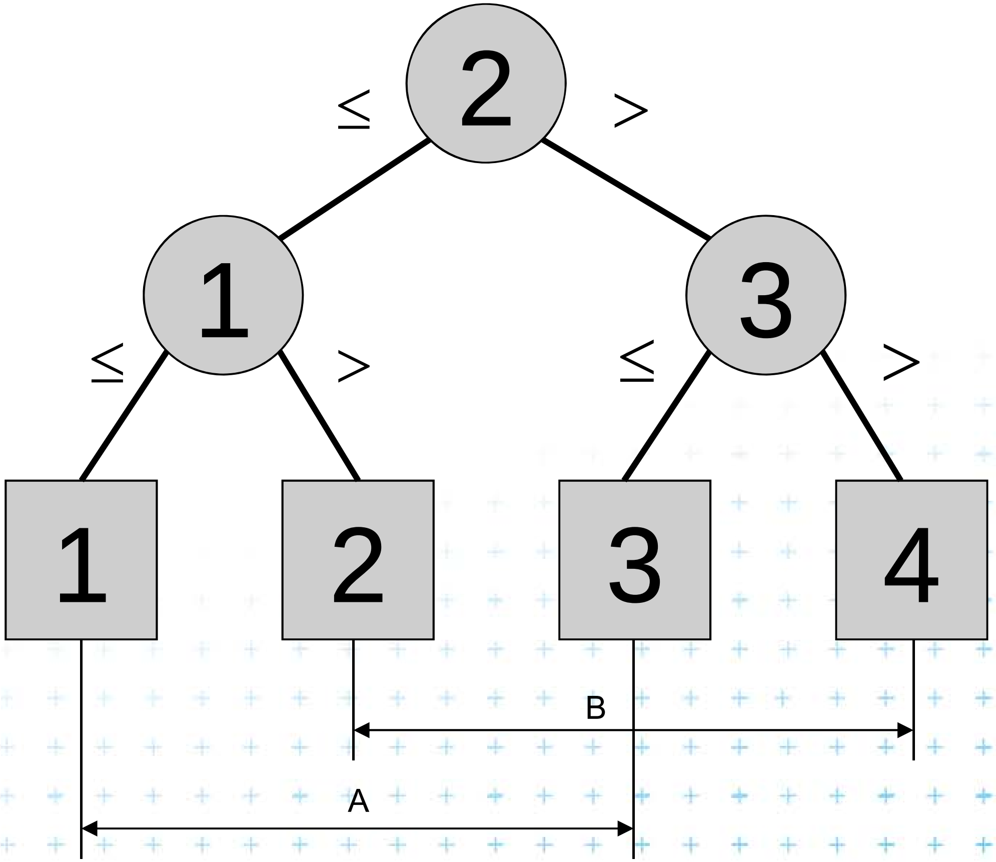


Example 1

Example 1 – static tree on endpoints



H(v) – value of node *v*



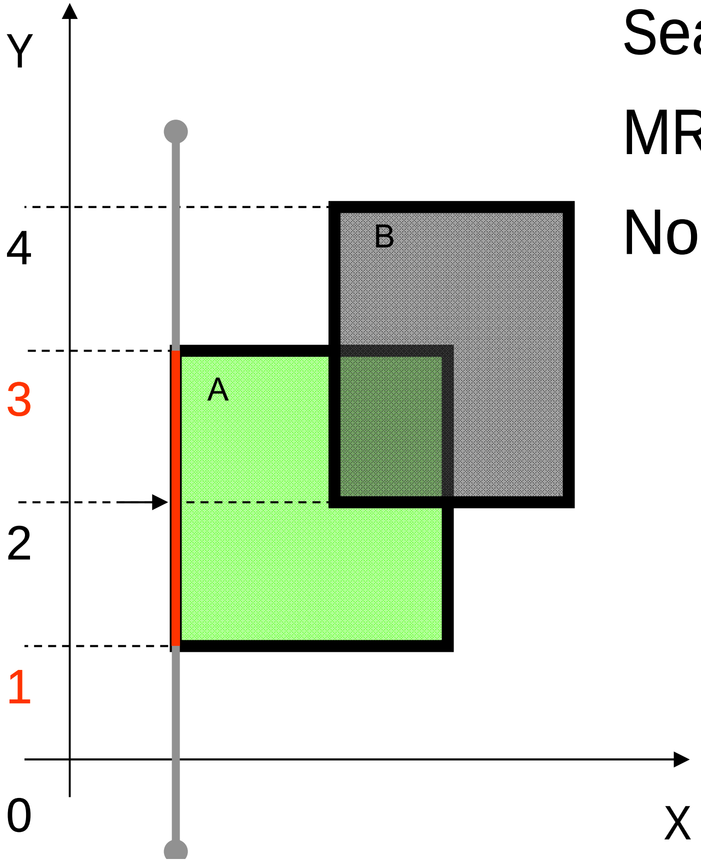
[Drtina]

Interval insertion [1,3] a) Query Interval

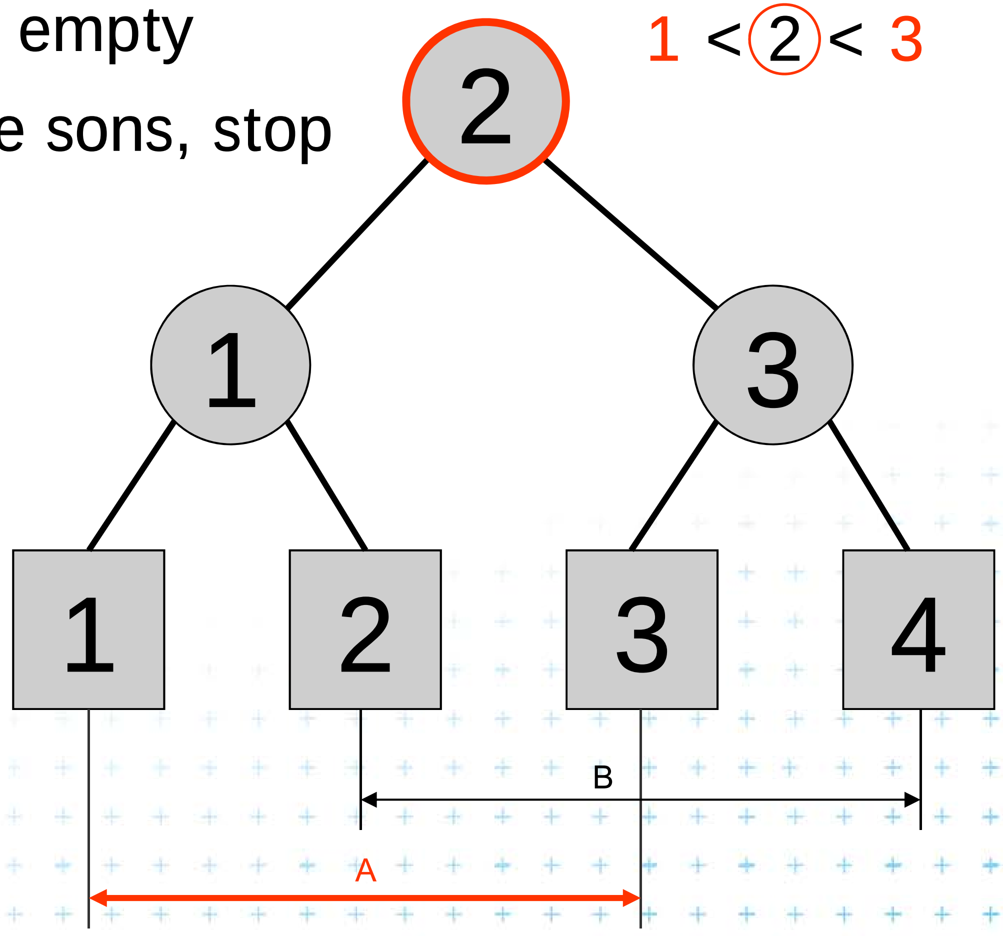
Search MR(v) or ML(v): ← b < H(v) < e

MR(v) is empty

No active sons, stop



$1 < 2 < 3$



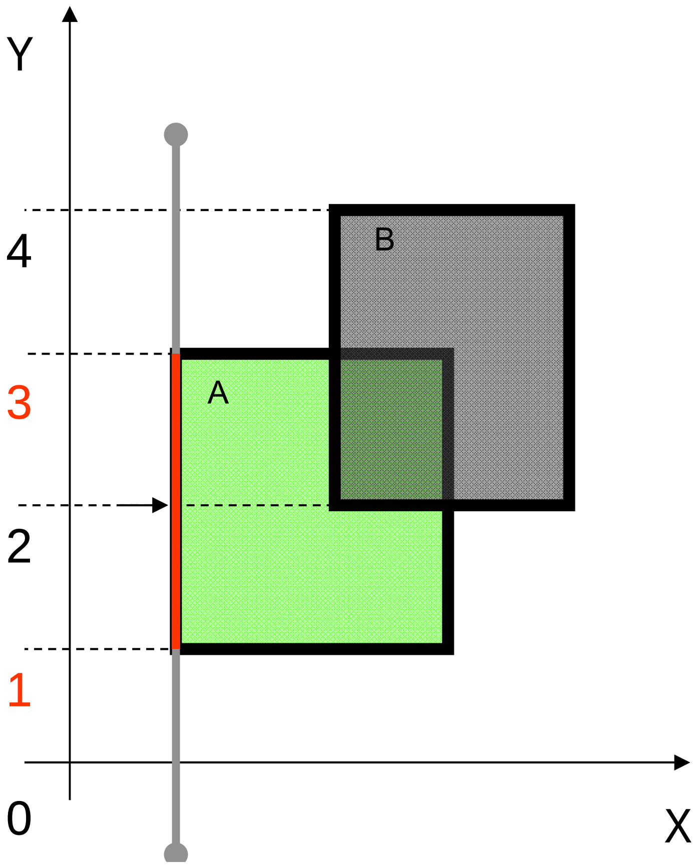
🟩 Active rectangle

○ Current node

● Active node

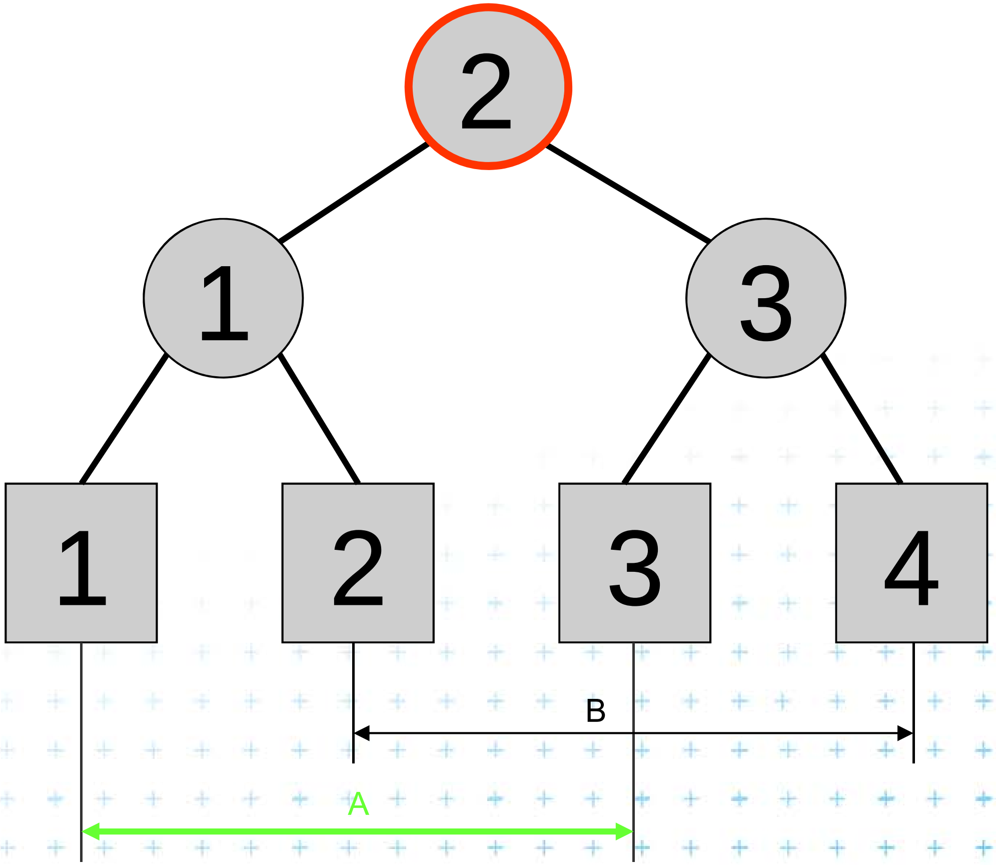
[Drtina]

Interval insertion [1,3] b) Insert Interval



$b \le H(v) \le e$

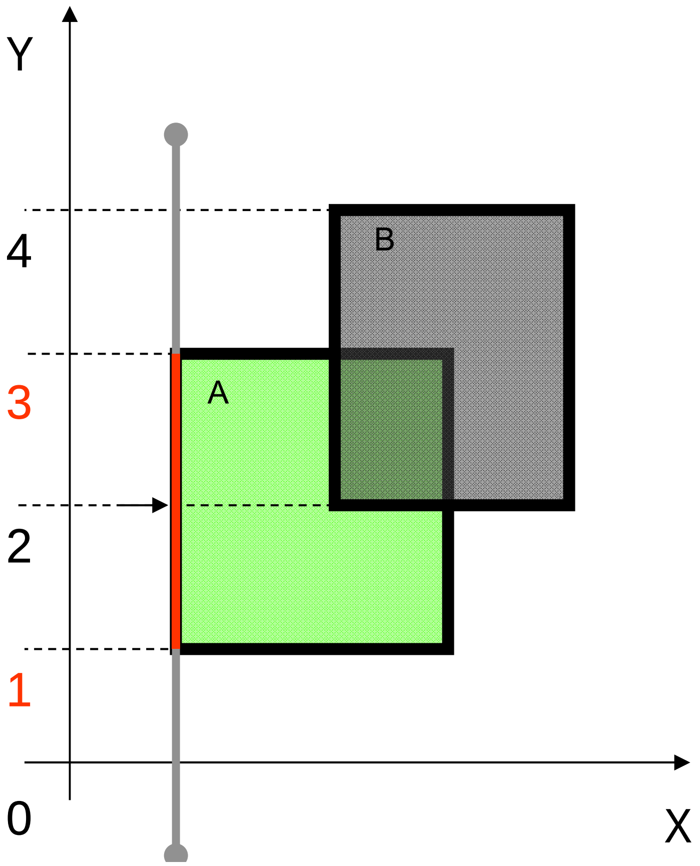
$? \ 1 \le 2 \le 3 \ ?$



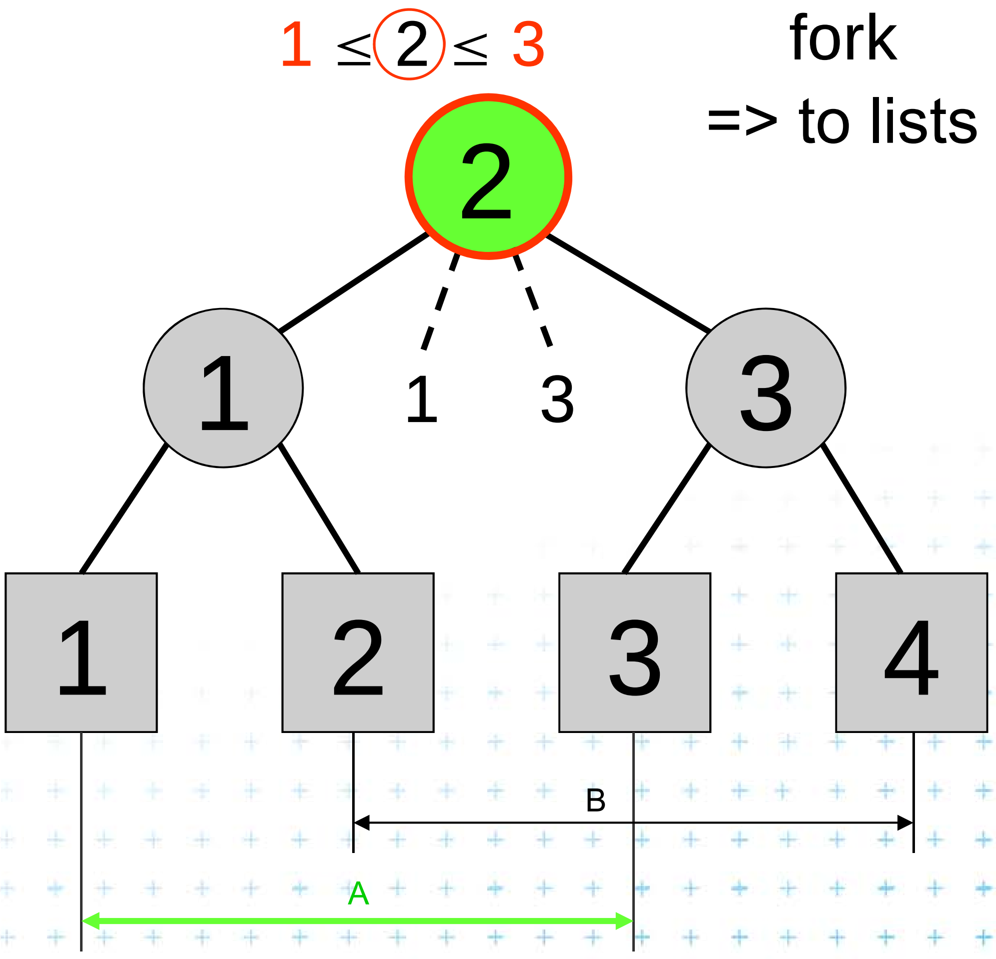
[Drtina]

- 🟩 Active rectangle
- ○ Current node
- ● Active node

Interval insertion [1,3] b) Insert Interval



$b \le H(v) \le e$



fork

=> to lists

🟩 Active rectangle

○ Current node

● Active node

[Drtina]

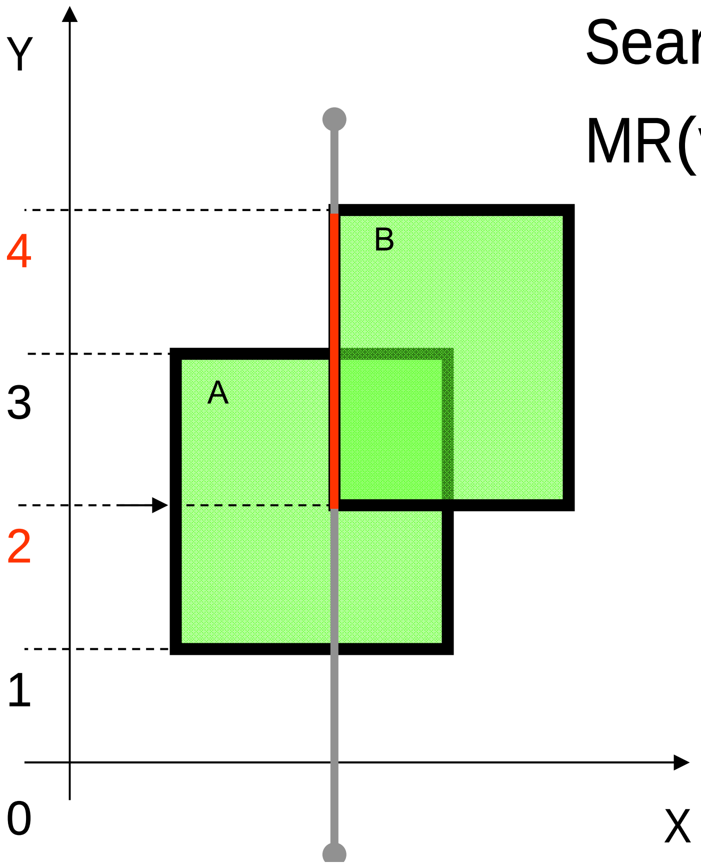
Interval insertion [2,4]   a) Query Interval

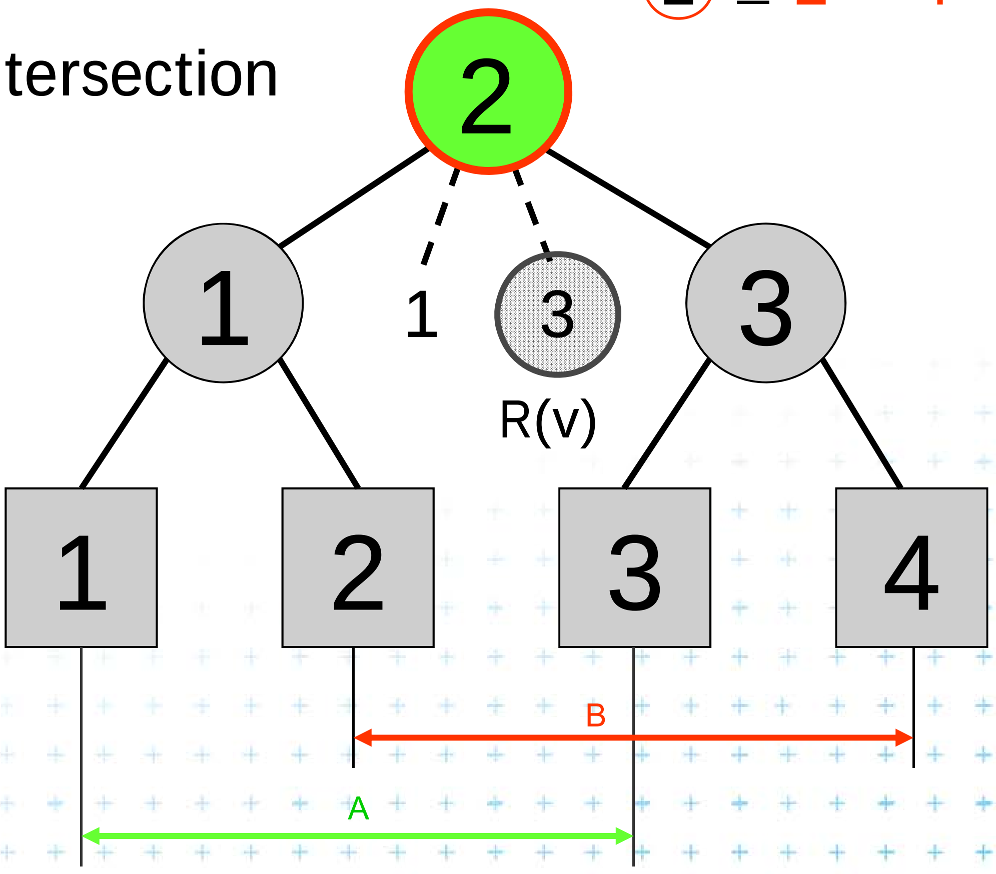
Search MR(v) only: ← $H(v) \le b < e$

$MR(v)[1] = 3 \ge 2$?

=> intersection

$\textcircled{2} \le 2 < 4$





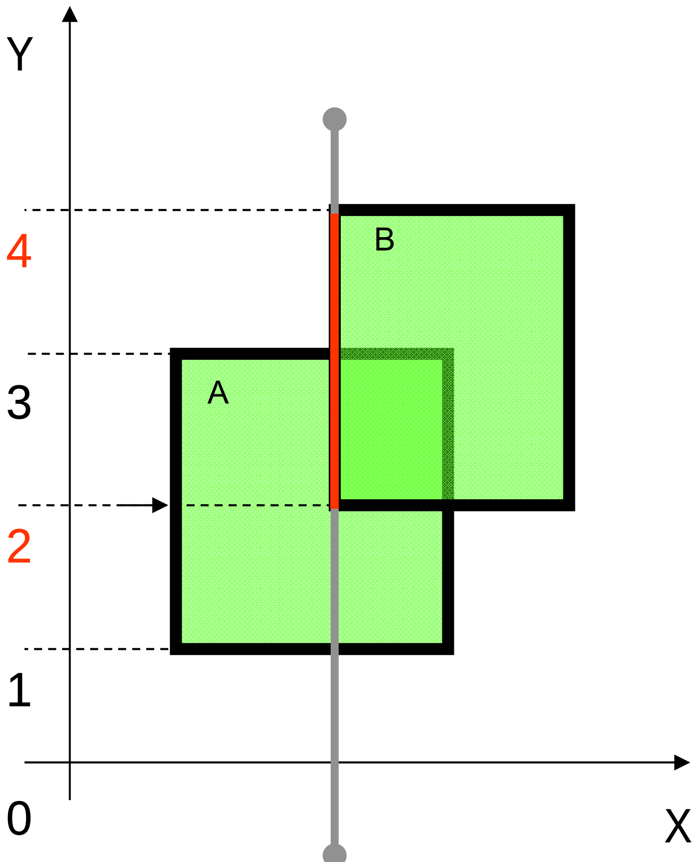
🟩 Active rectangle

○ Current node

● Active node

[Drtina]

Interval insertion [2,4]   b) Insert Interval

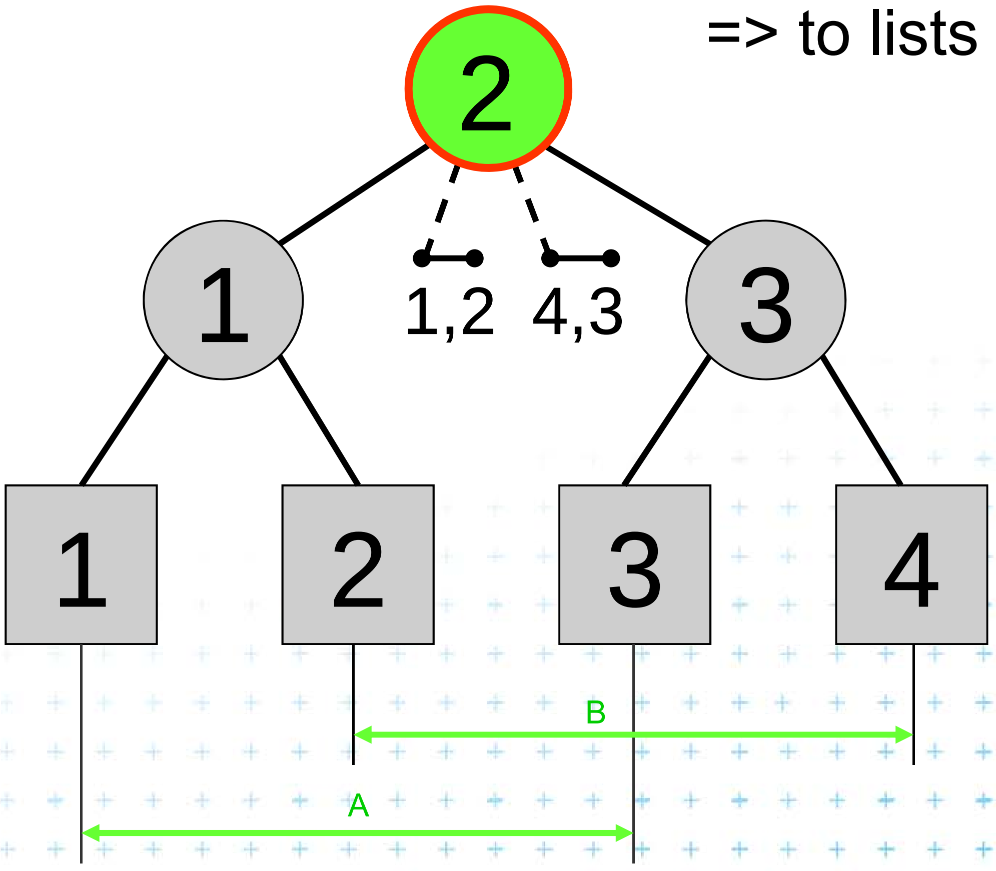


$b \le H(v) \le e$

$2 \le \textcircled{2} \le 4$

fork

=> to lists



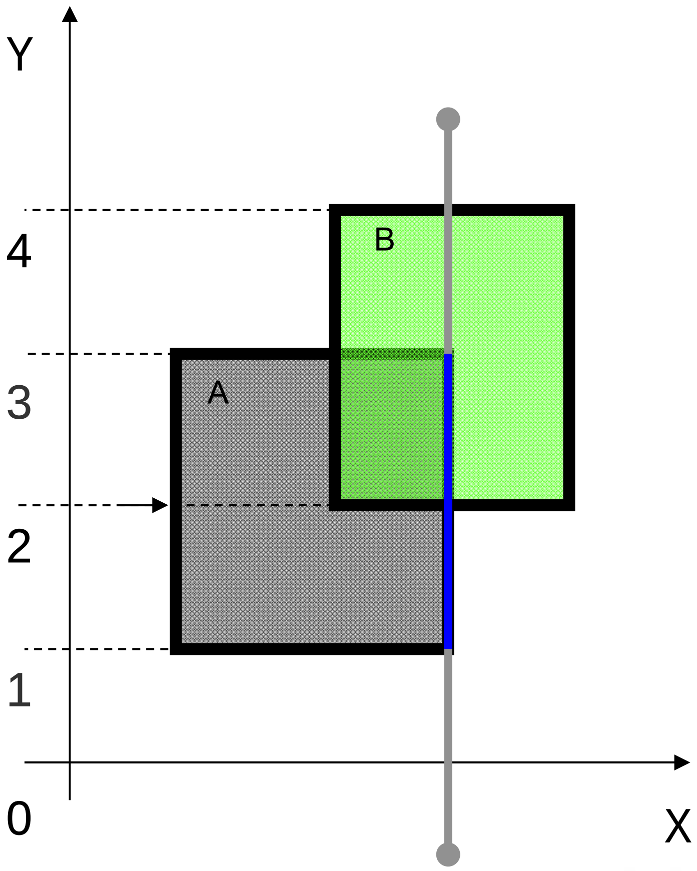
🟩 Active rectangle

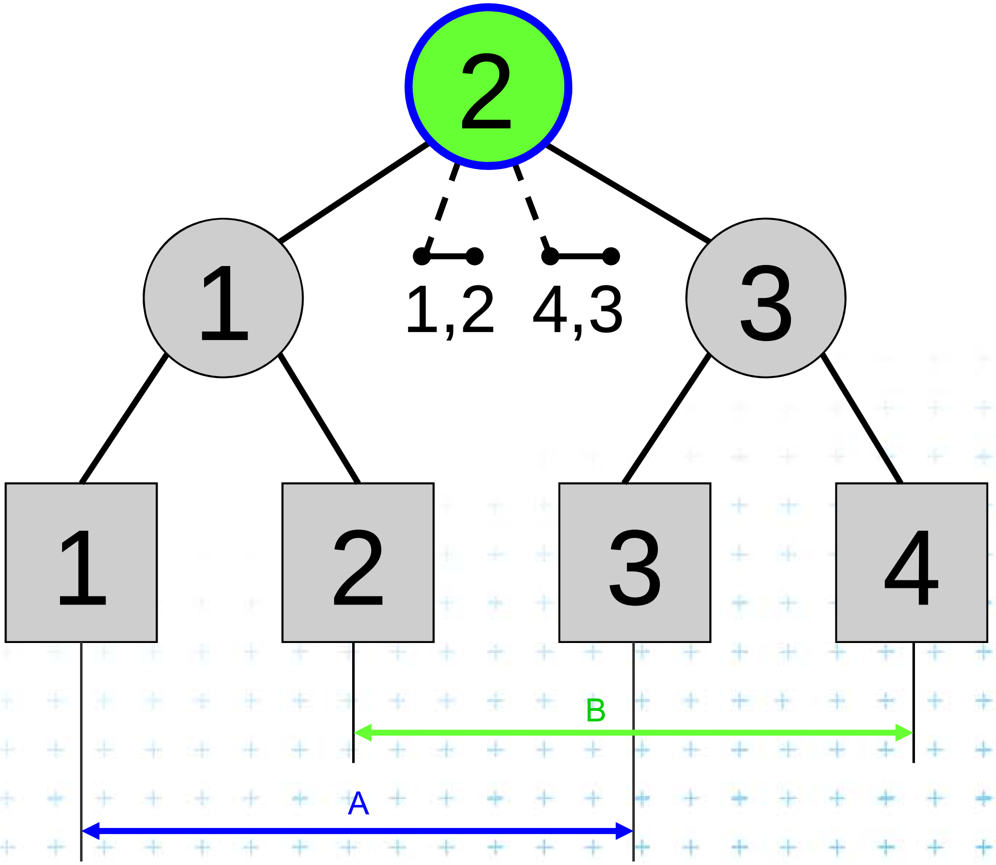
○ Current node

● Active node

[Drtina]

Interval delete [1,3]





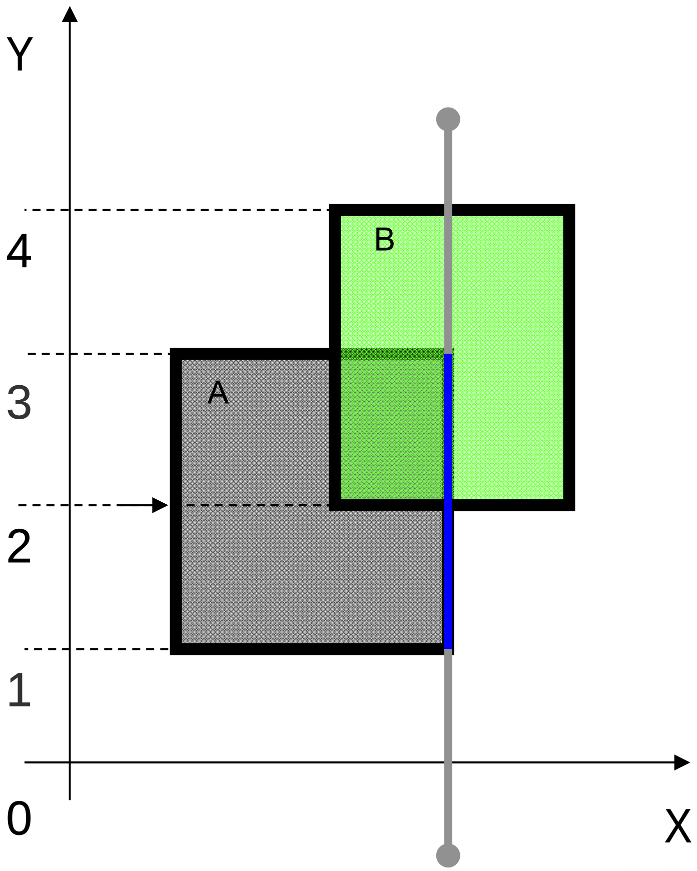
🟩 Active rectangle

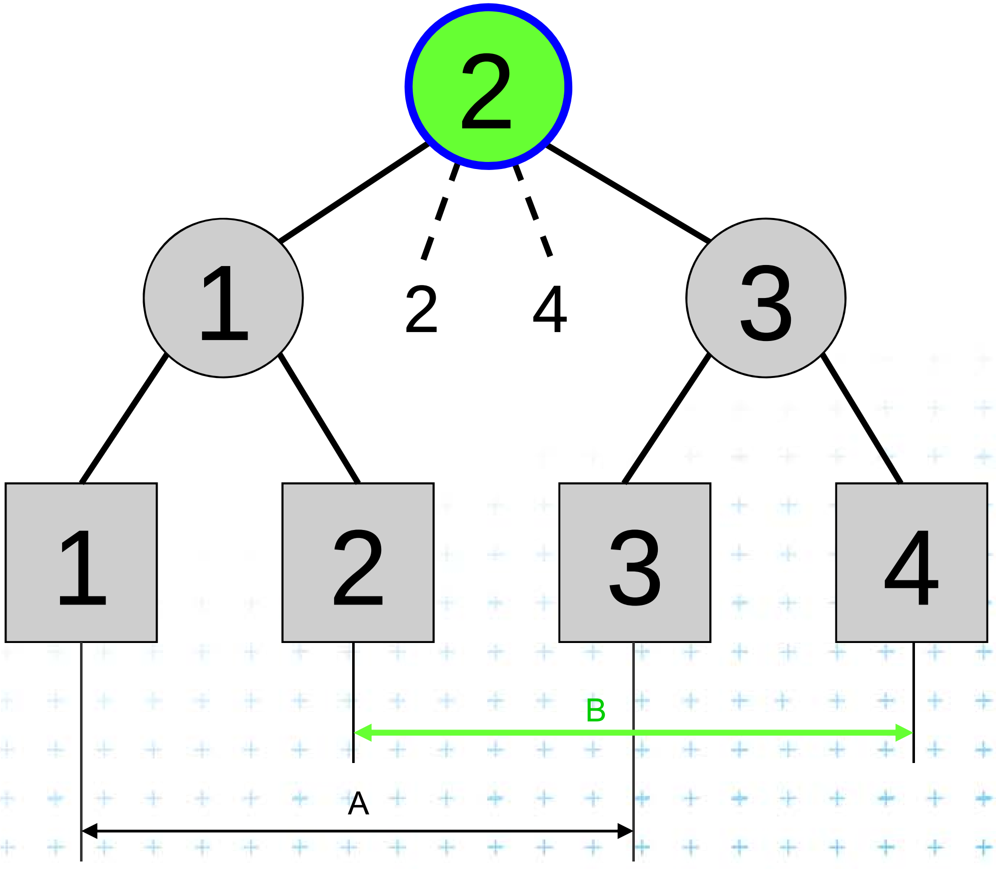
○ Current node

● Active node

[Drtina]

Interval delete [1,3]





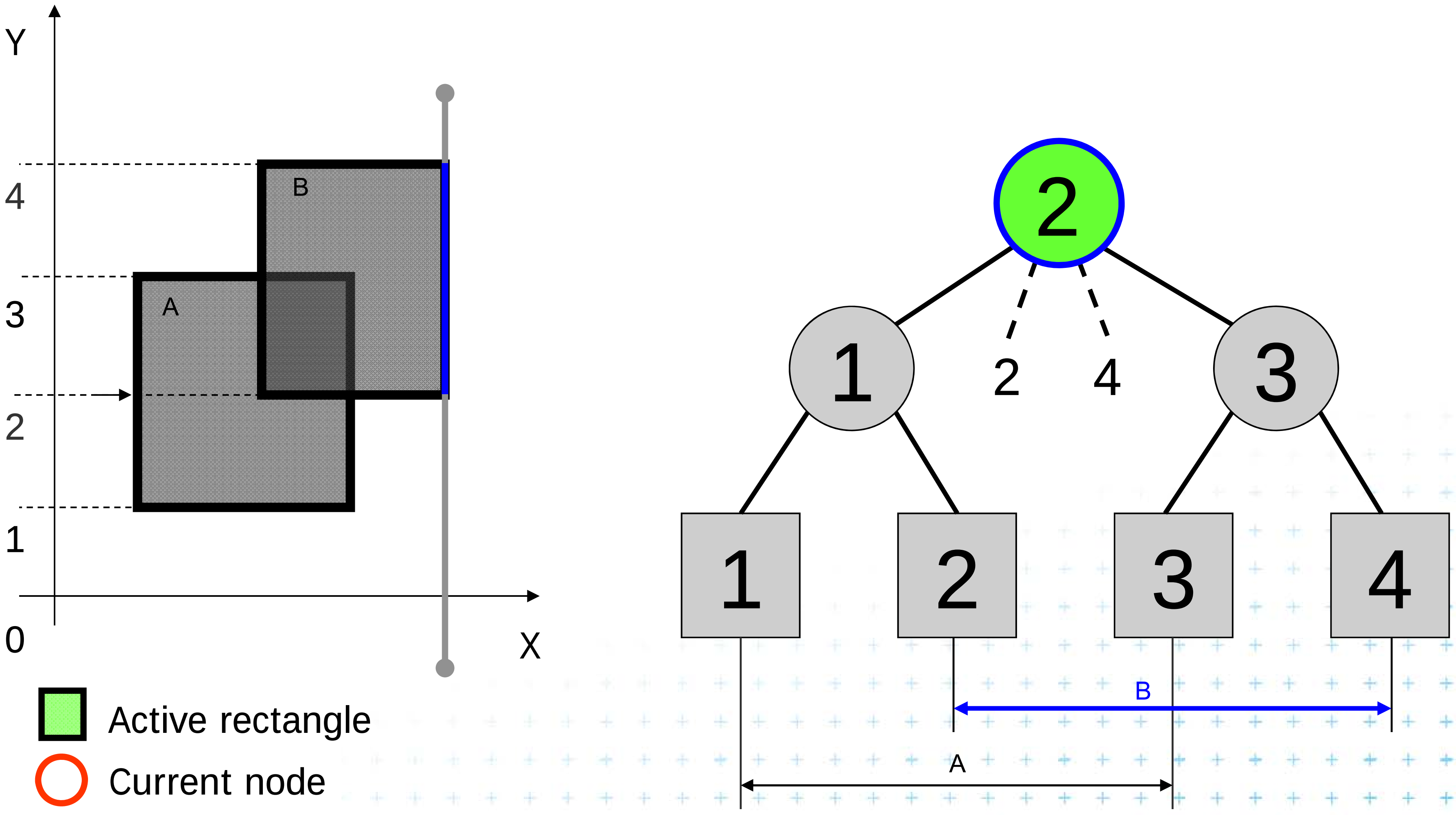
🟩 Active rectangle

○ Current node

● Active node

[Drtina]

Interval delete [2,4]



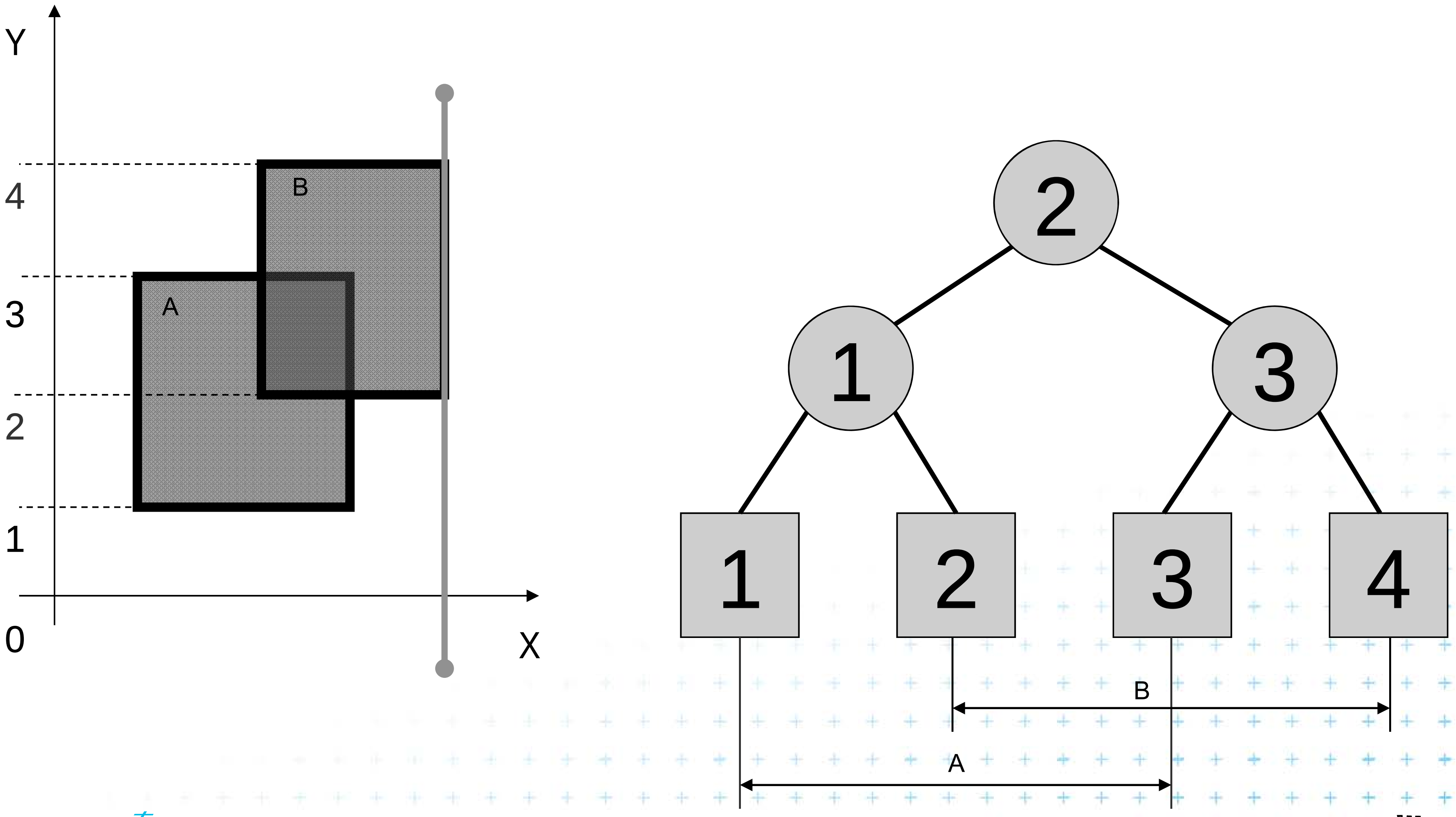
🟩 Active rectangle

○ Current node

● Active node

[Drtina]

Interval delete [2,4]



[Drtina]

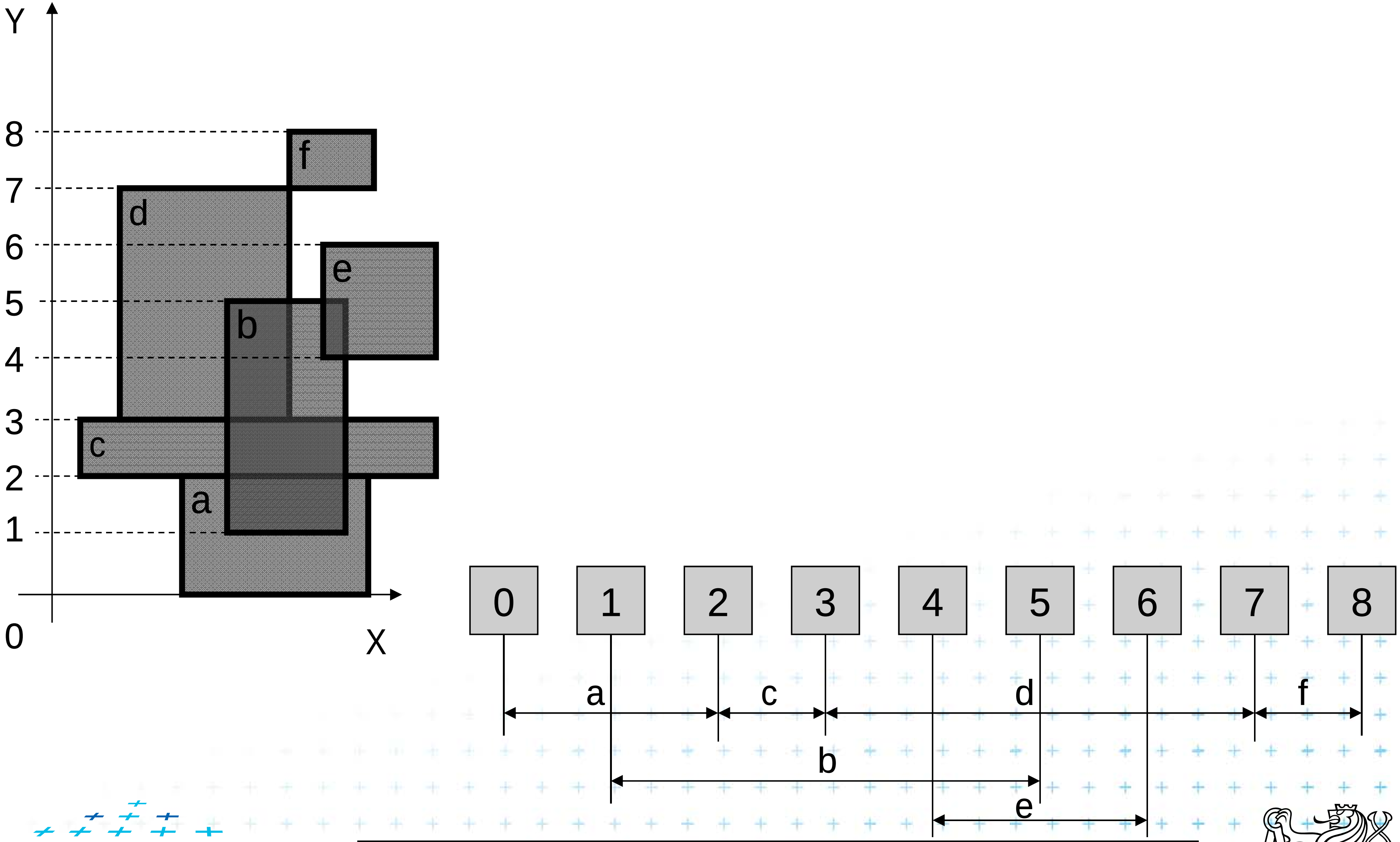
Example 2

// this is a copy of the slide before
// just to remember the algorithm

RectangleIntersections(*S*)
Input: Set *S* of rectangles
Output: Intersected rectangle pairs

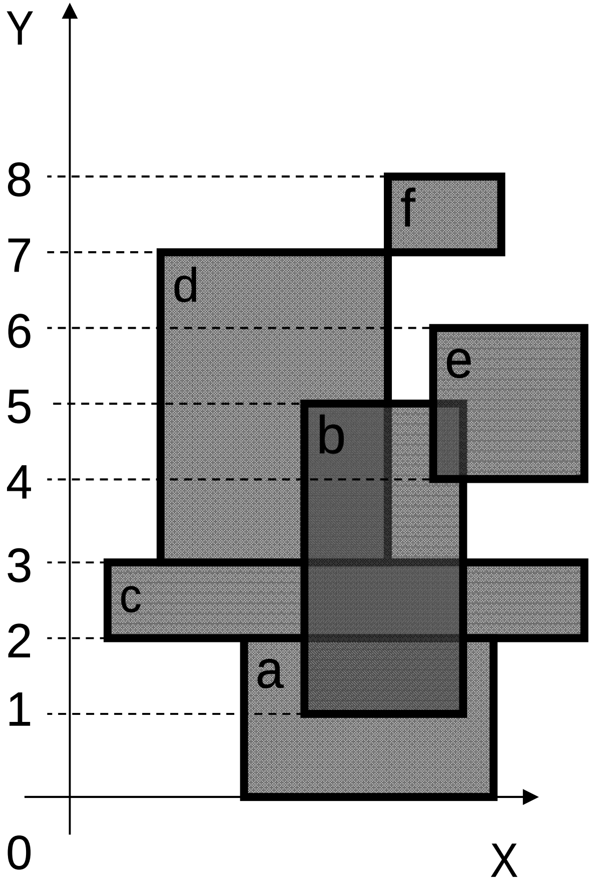
```
Preprocess( S )            // create the interval tree T and event queue Q

while ( $Q \neq \emptyset$ ) do
    Get next entry ($x_{il}, y_{il}, y_{ir}, t$) from Q     // $t \in$ { left | right }
    if ( $t$ = left )    // left edge
            a) QueryInterval ( $y_{il}, y_{ir}$, root($T$))   // report intersections
            b) InsertInterval ( $y_{il}, y_{ir}$, root($T$))   // insert new interval
    else              // right edge
            c) DeleteInterval ( $y_{il}, y_{ir}$, root($T$) )
```

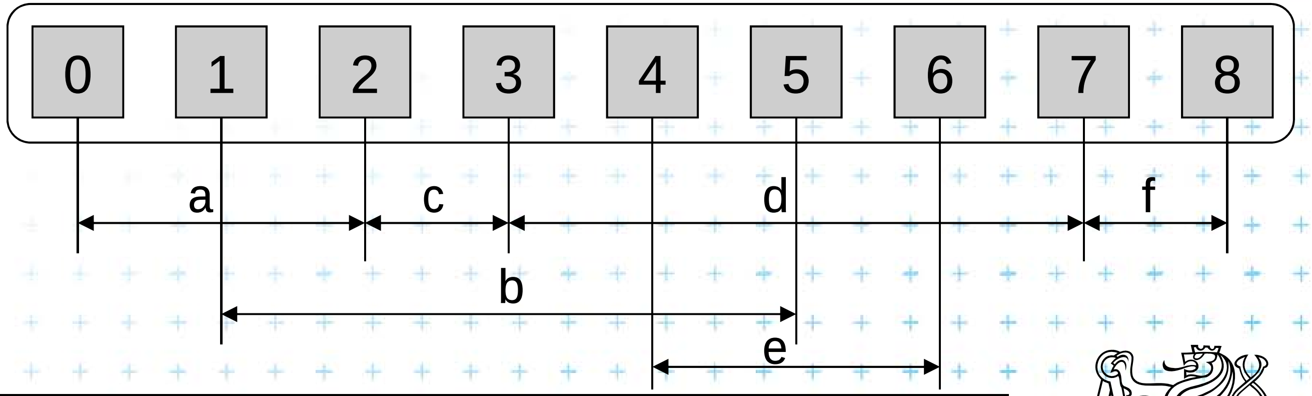
Example 2 – tree from PrimaryTree(*S*)



[Drtina]

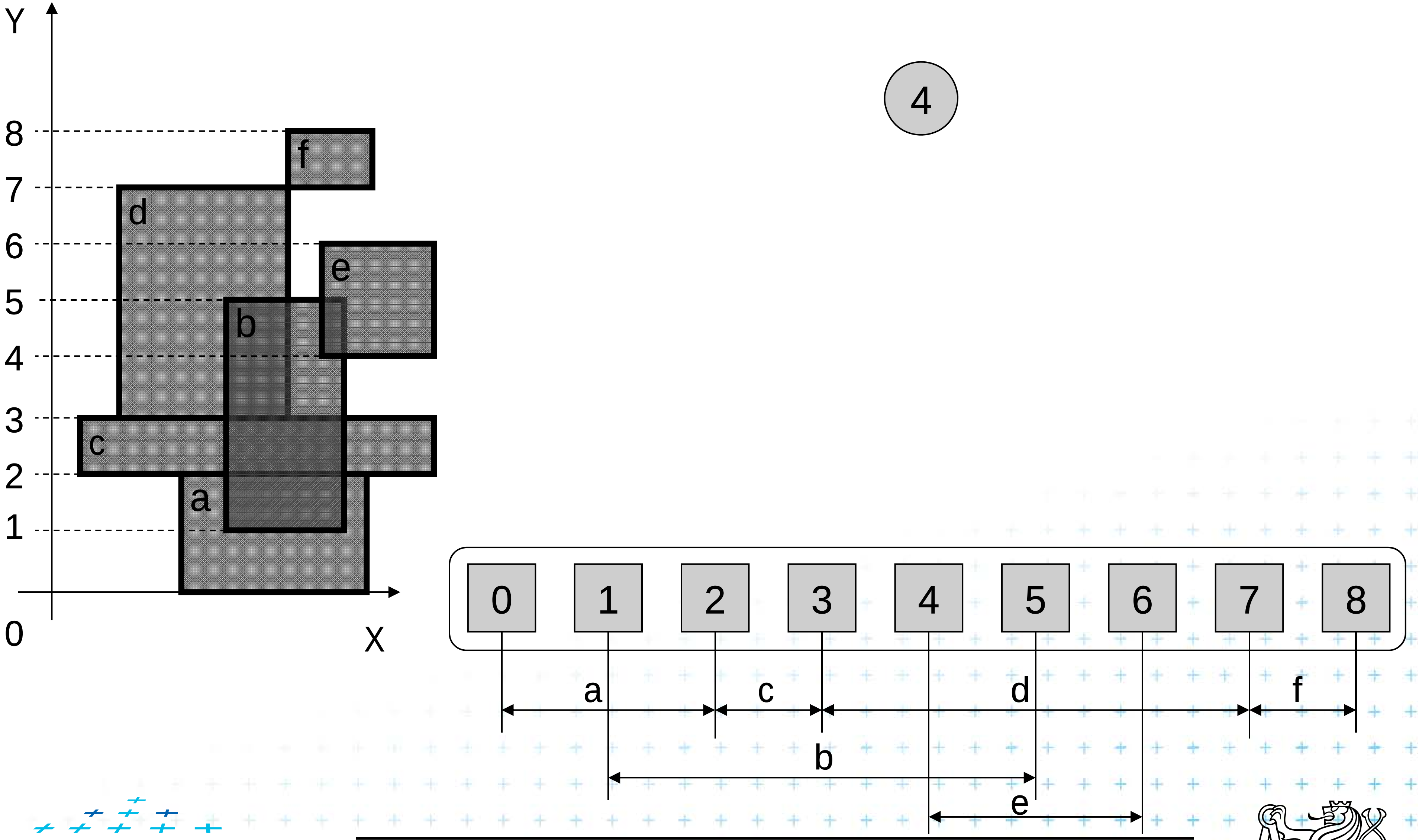
Example 2 – tree from PrimaryTree(*S*)





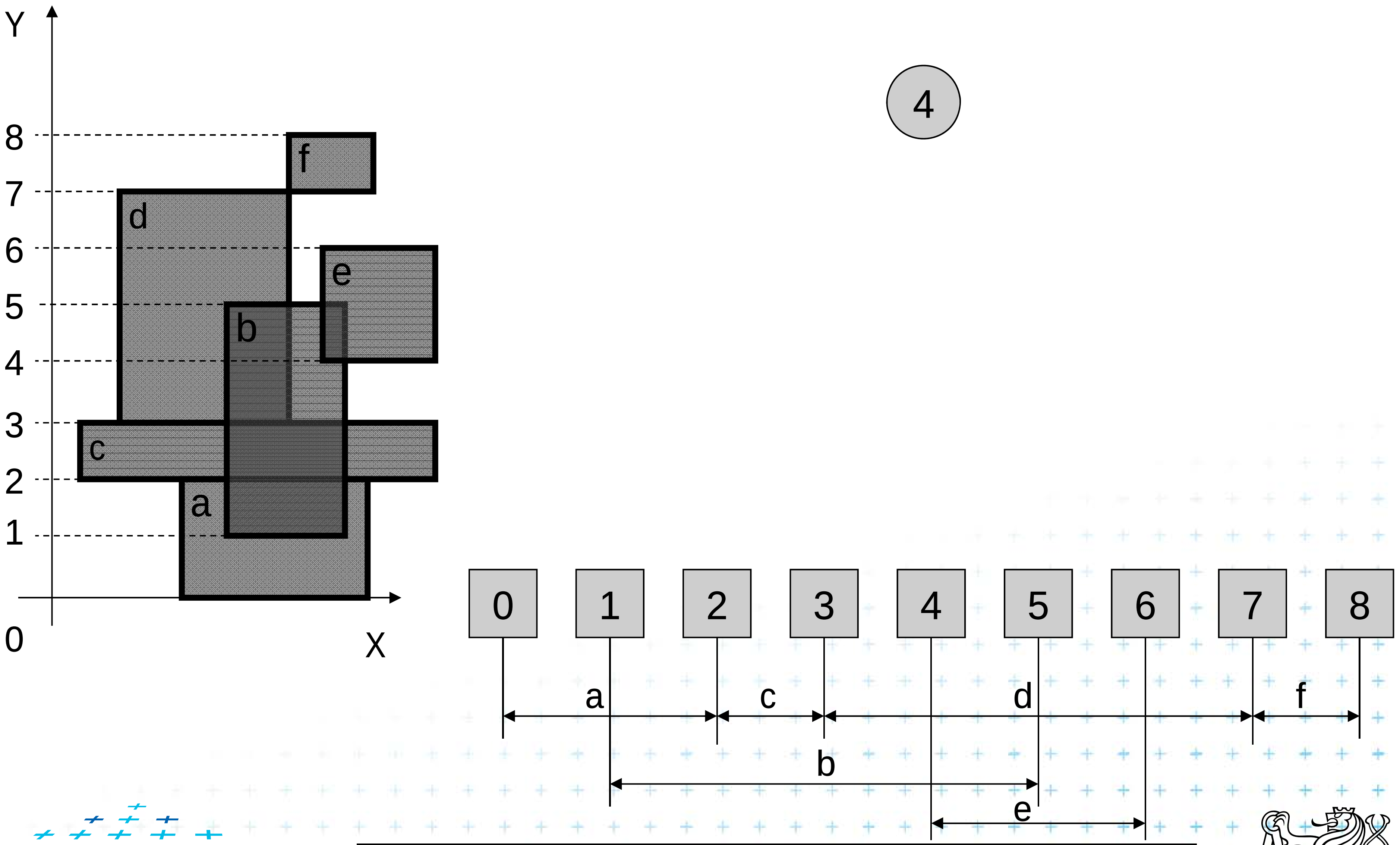
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



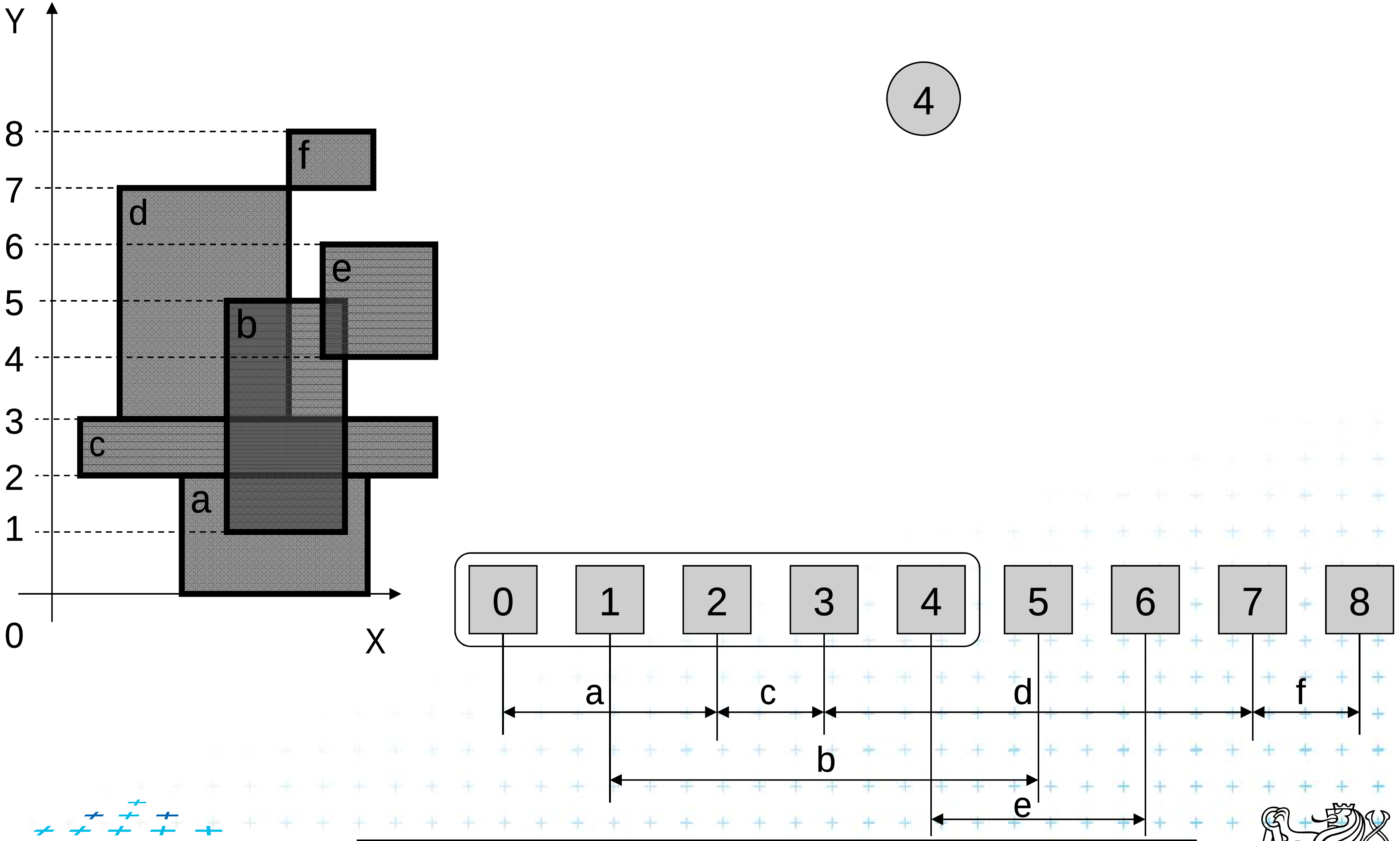
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



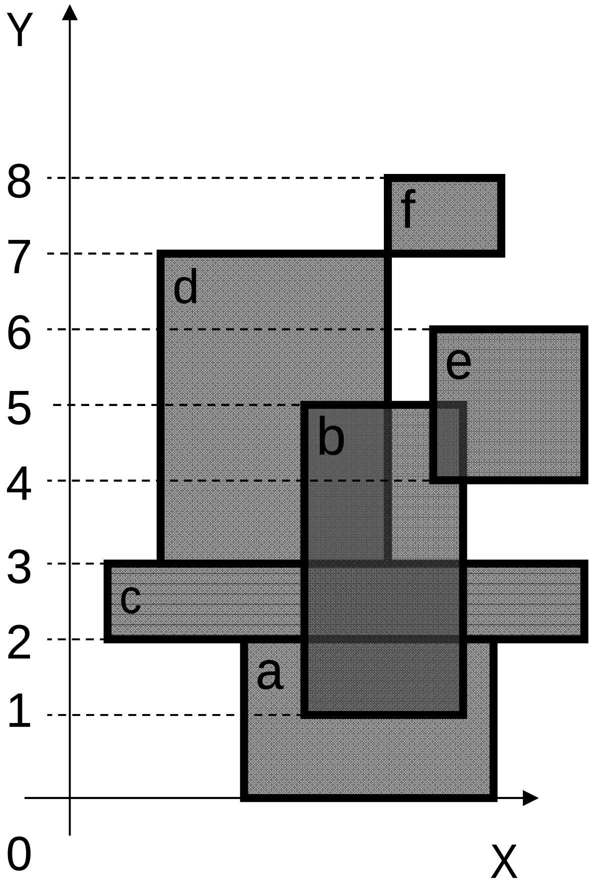
[Drtina]

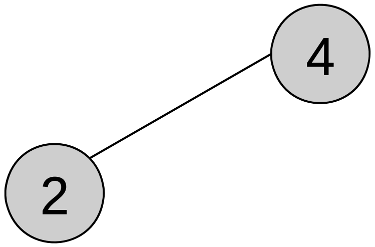
Example 2 – tree from PrimaryTree(*S*)

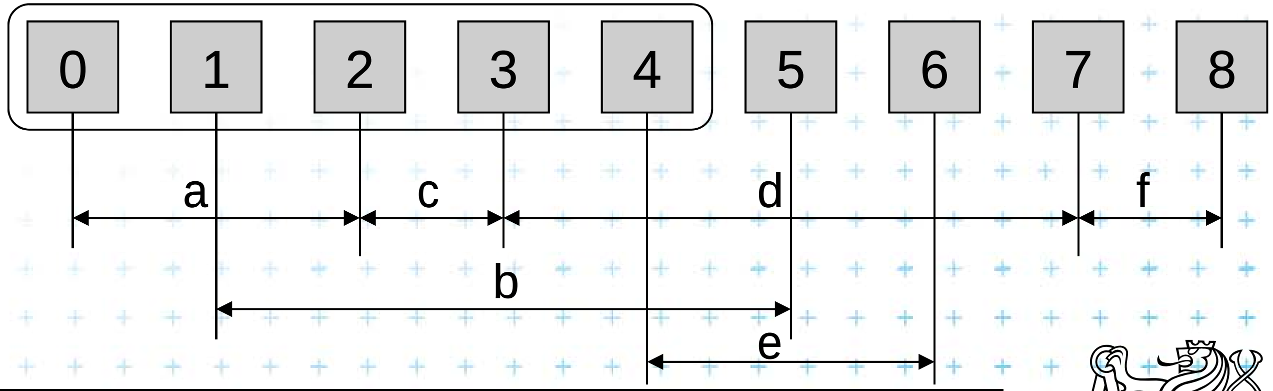


[Drtina]

Example 2 – tree from PrimaryTree(*S*)

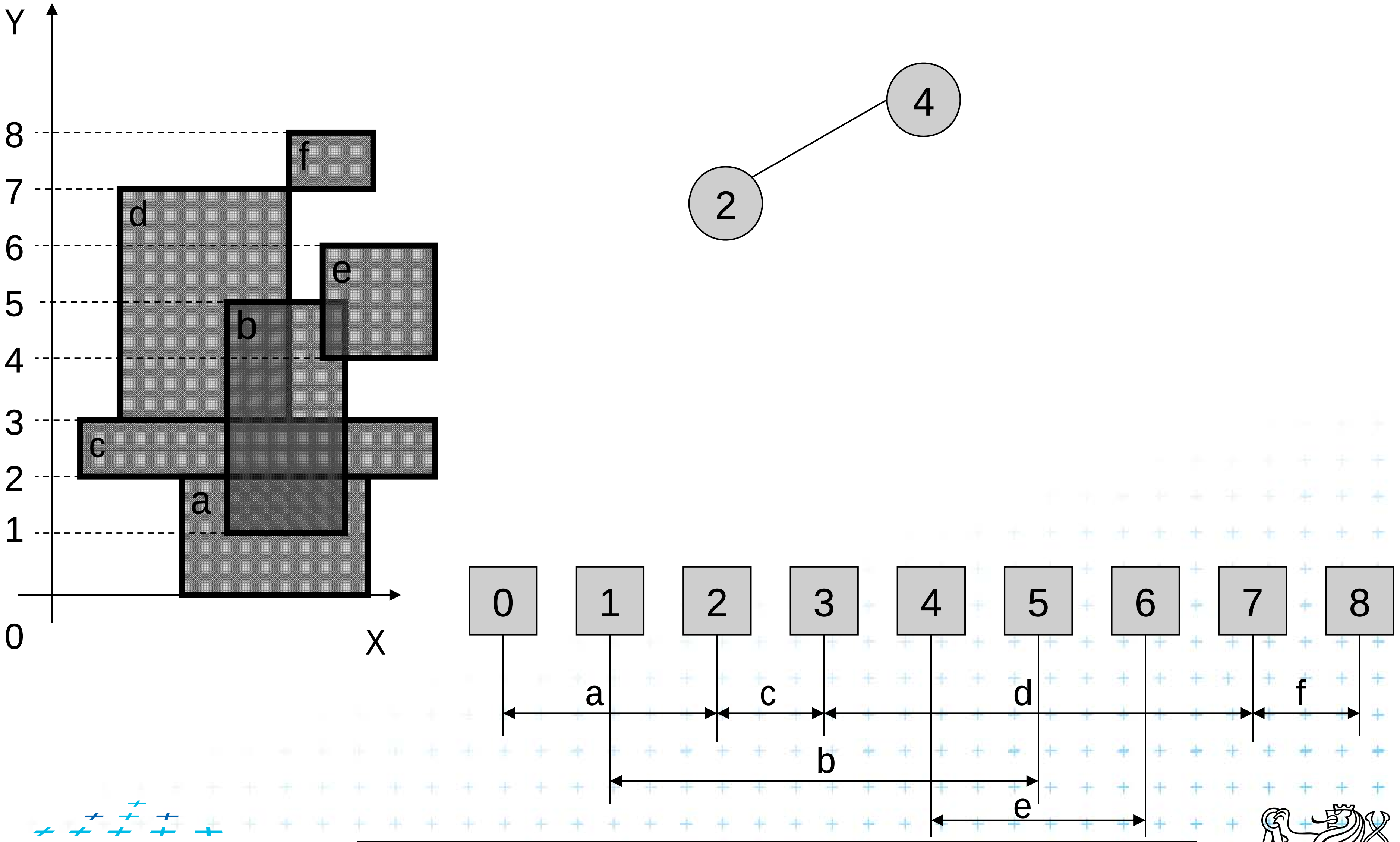






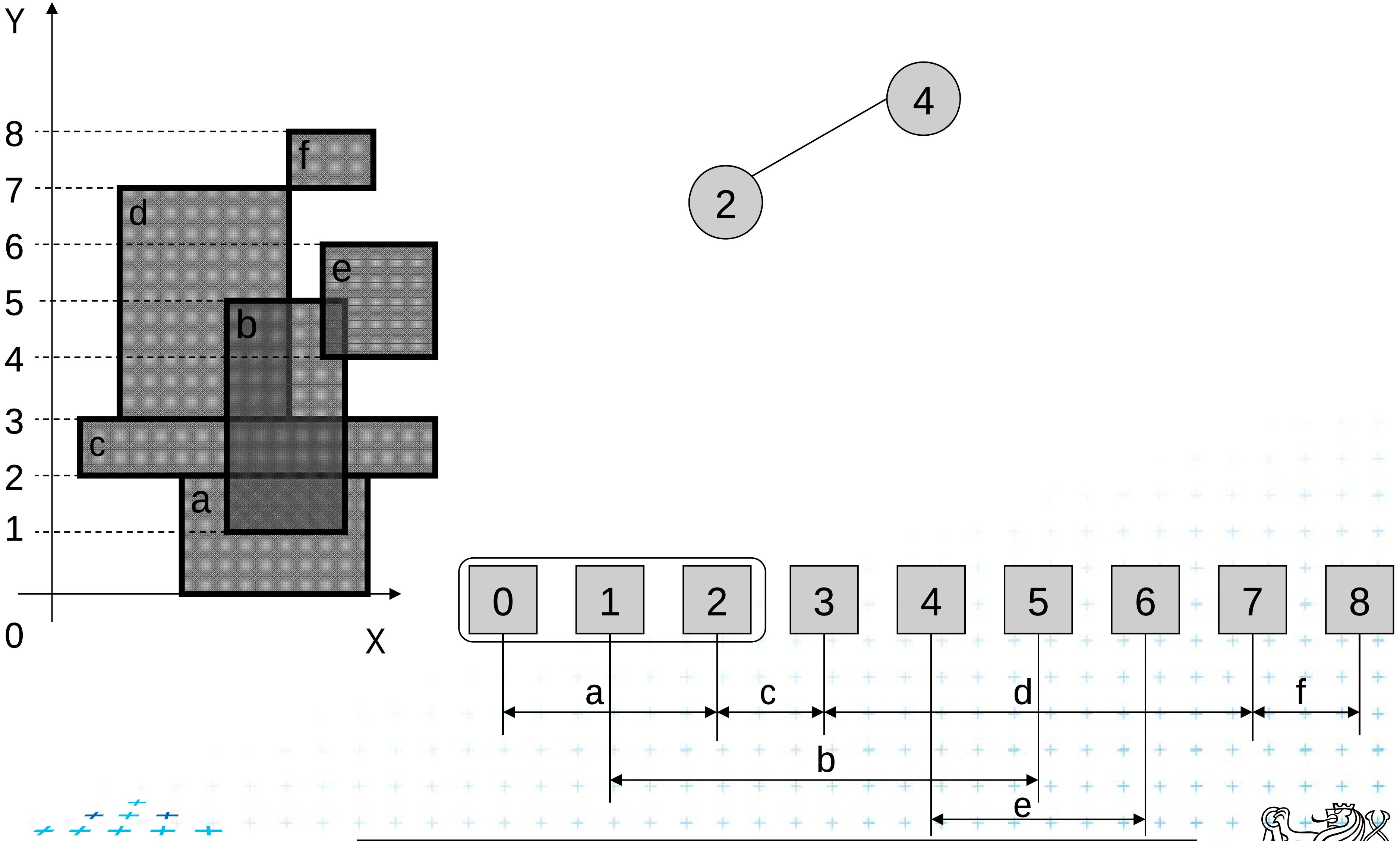
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



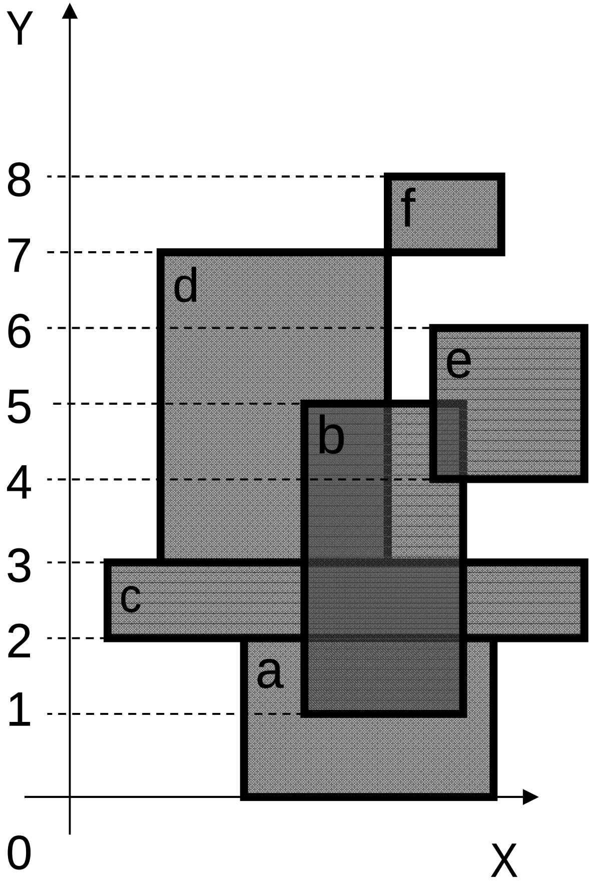
[Drtina]

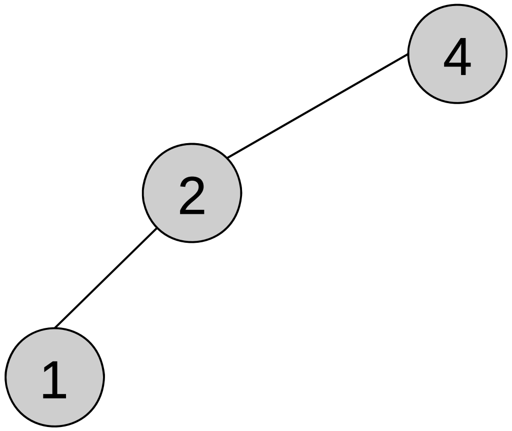
Example 2 – tree from PrimaryTree(*S*)

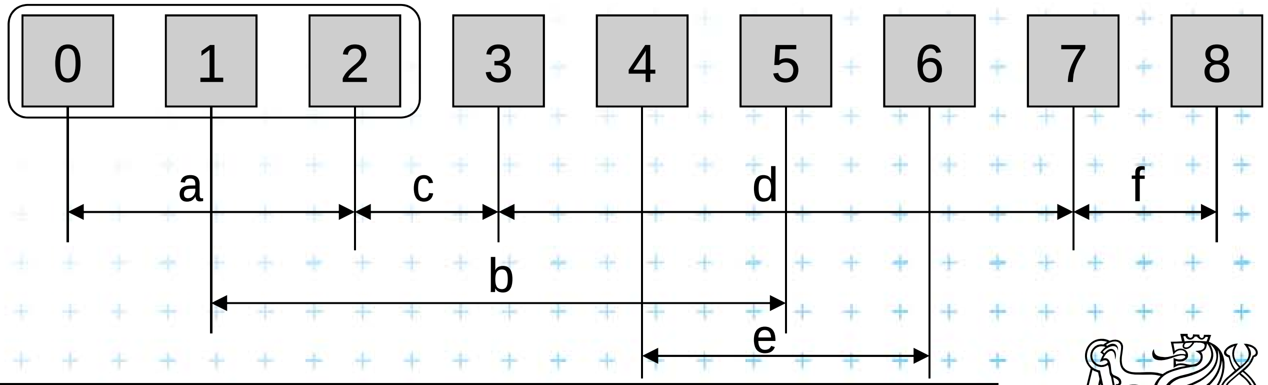


[Drtina]

Example 2 – tree from PrimaryTree(*S*)

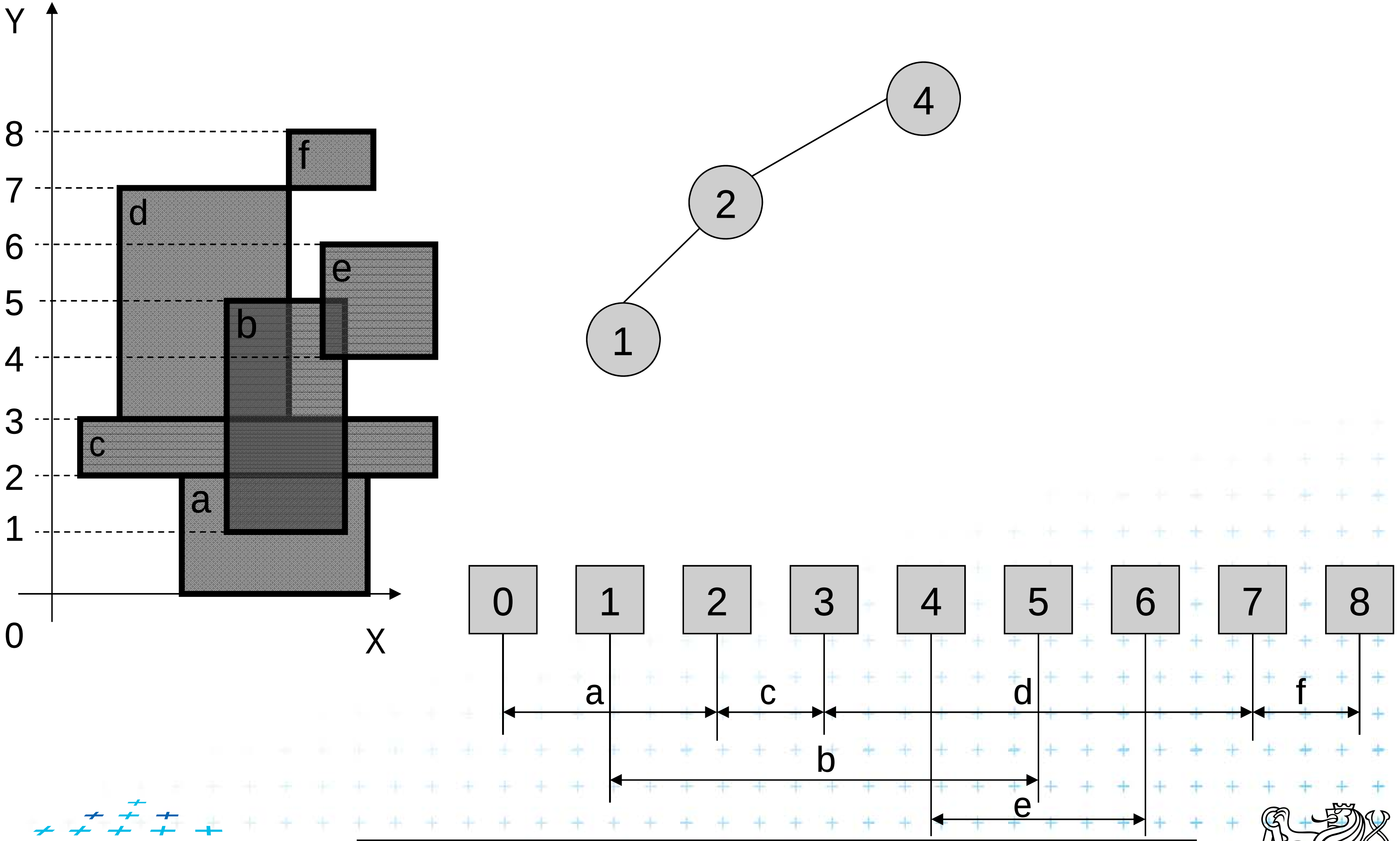






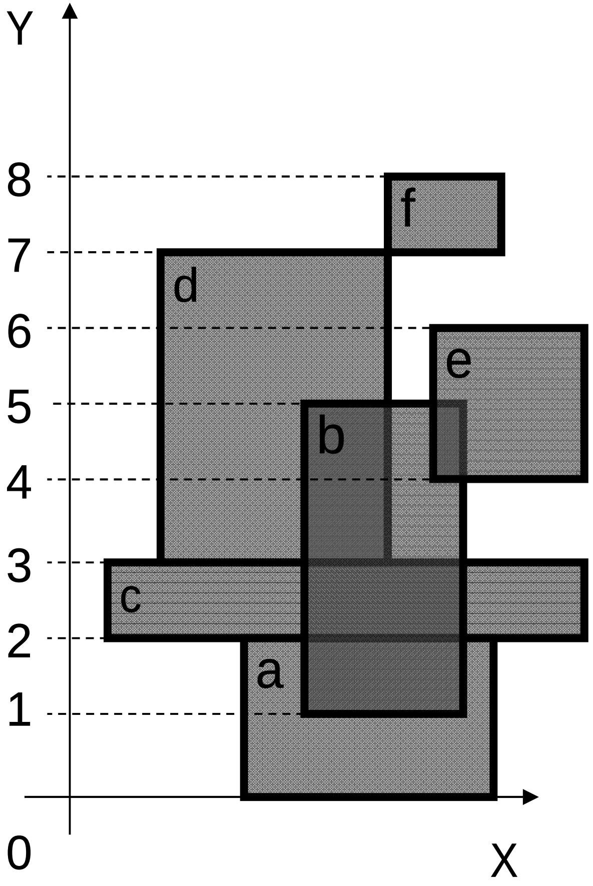
[Drtina]

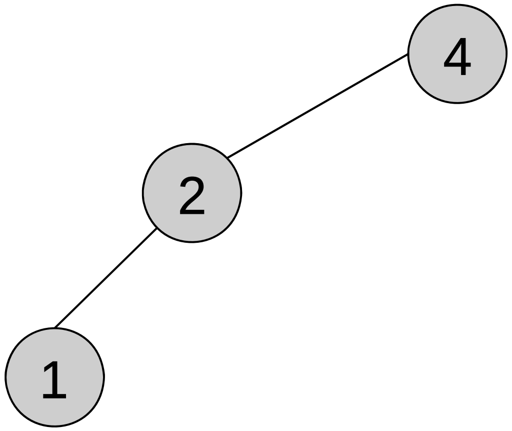
Example 2 – tree from PrimaryTree(*S*)

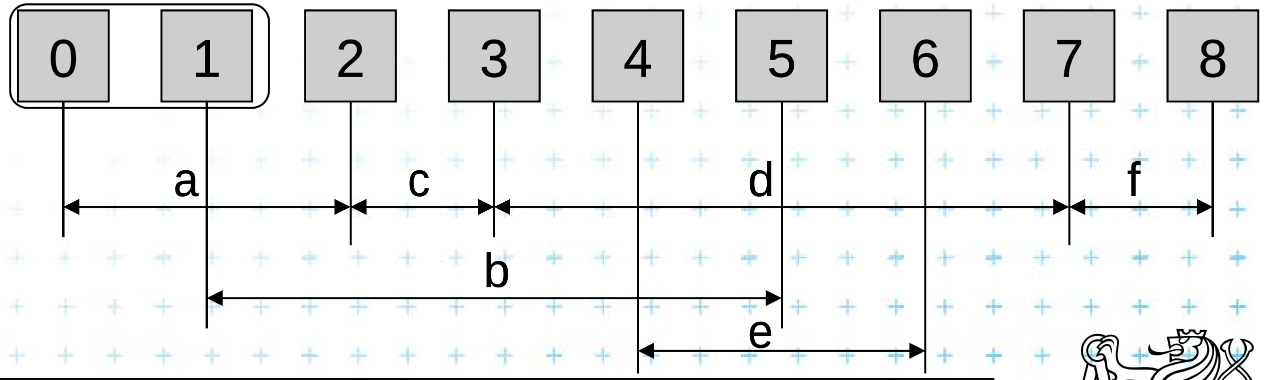


[Drtina]

Example 2 – tree from PrimaryTree(*S*)

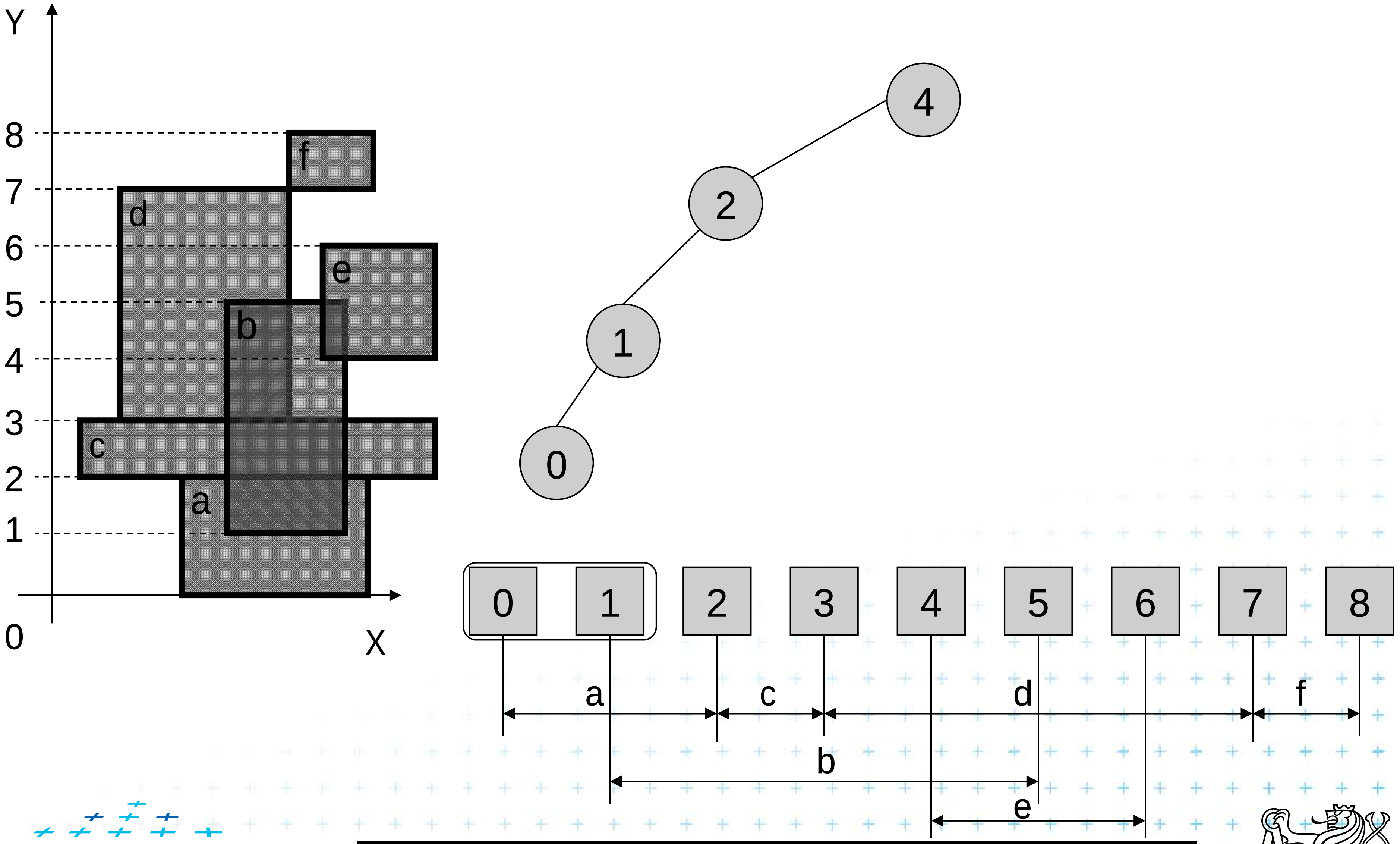






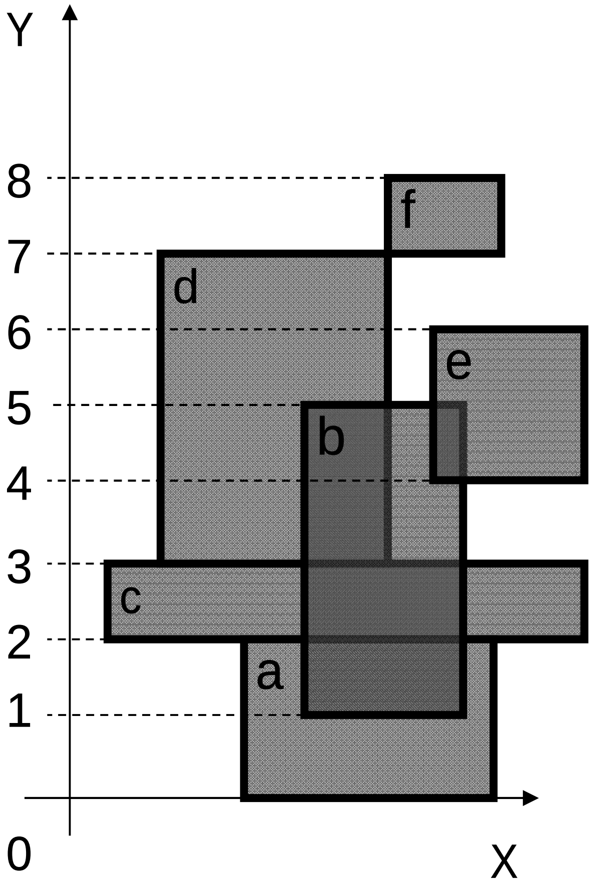
[Drtina]

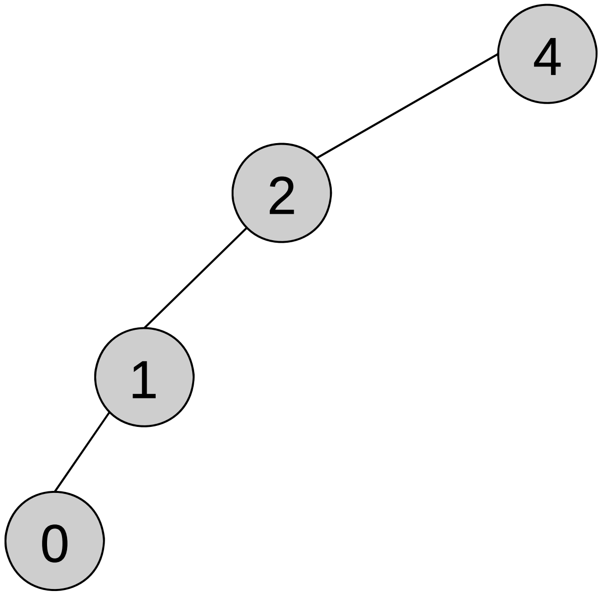
Example 2 – tree from PrimaryTree(*S*)

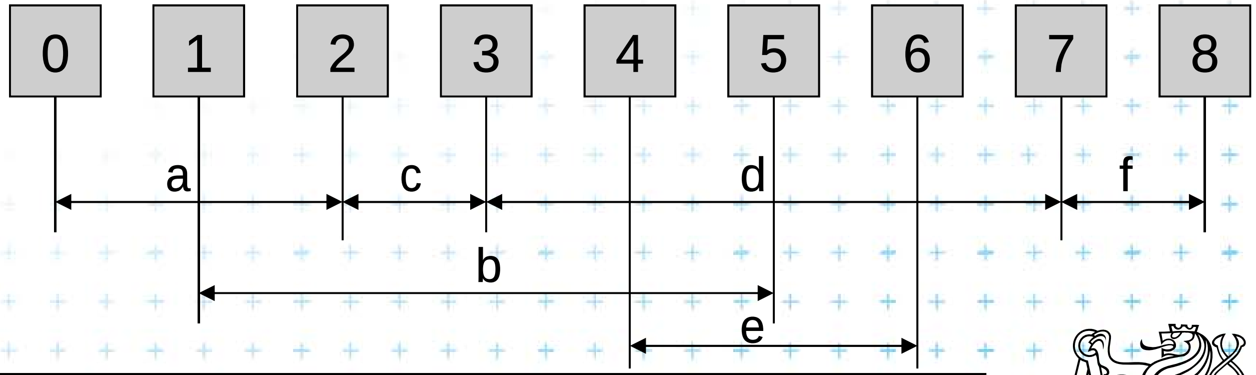


[Drtina]

Example 2 – tree from PrimaryTree(*S*)

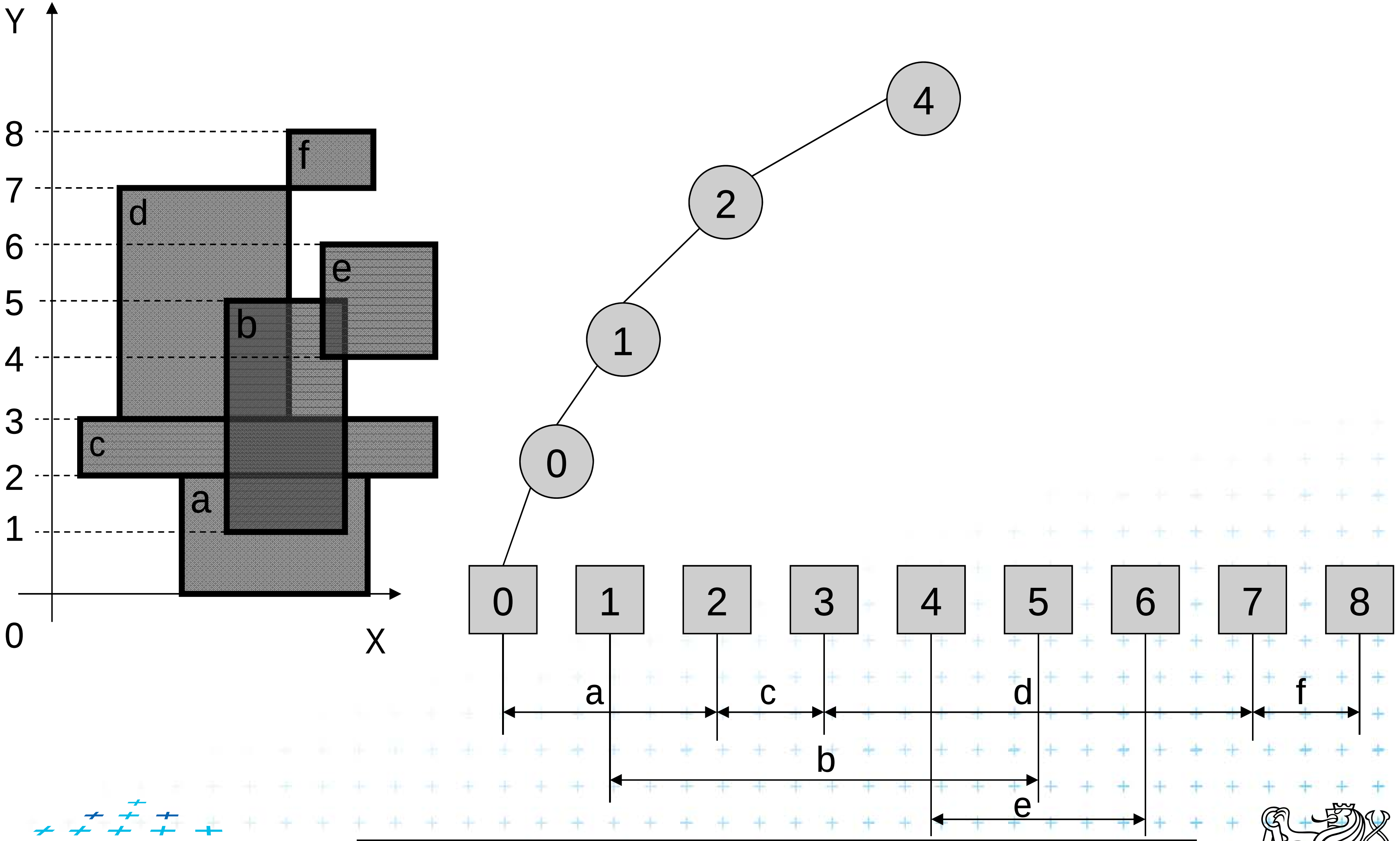






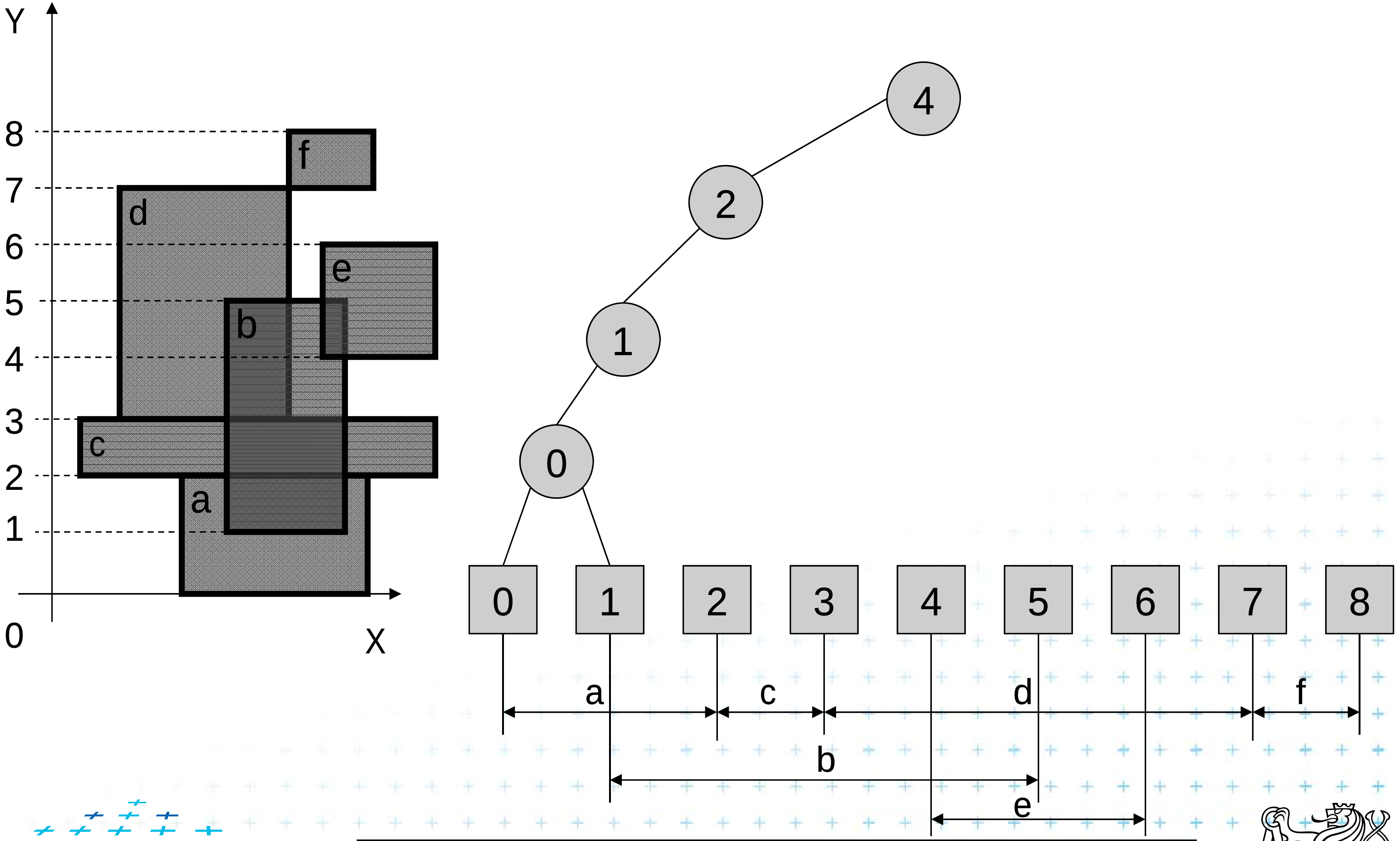
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



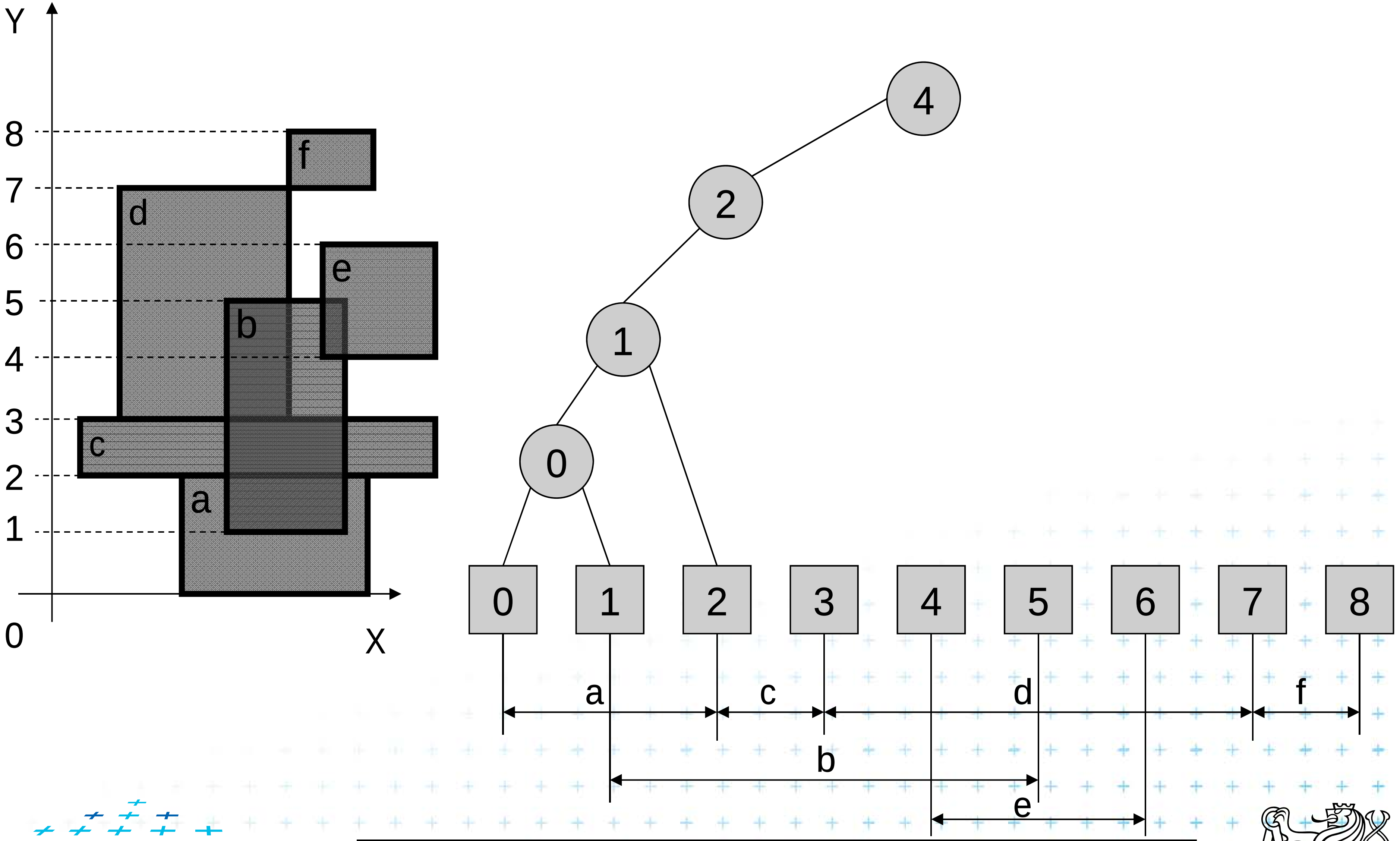
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



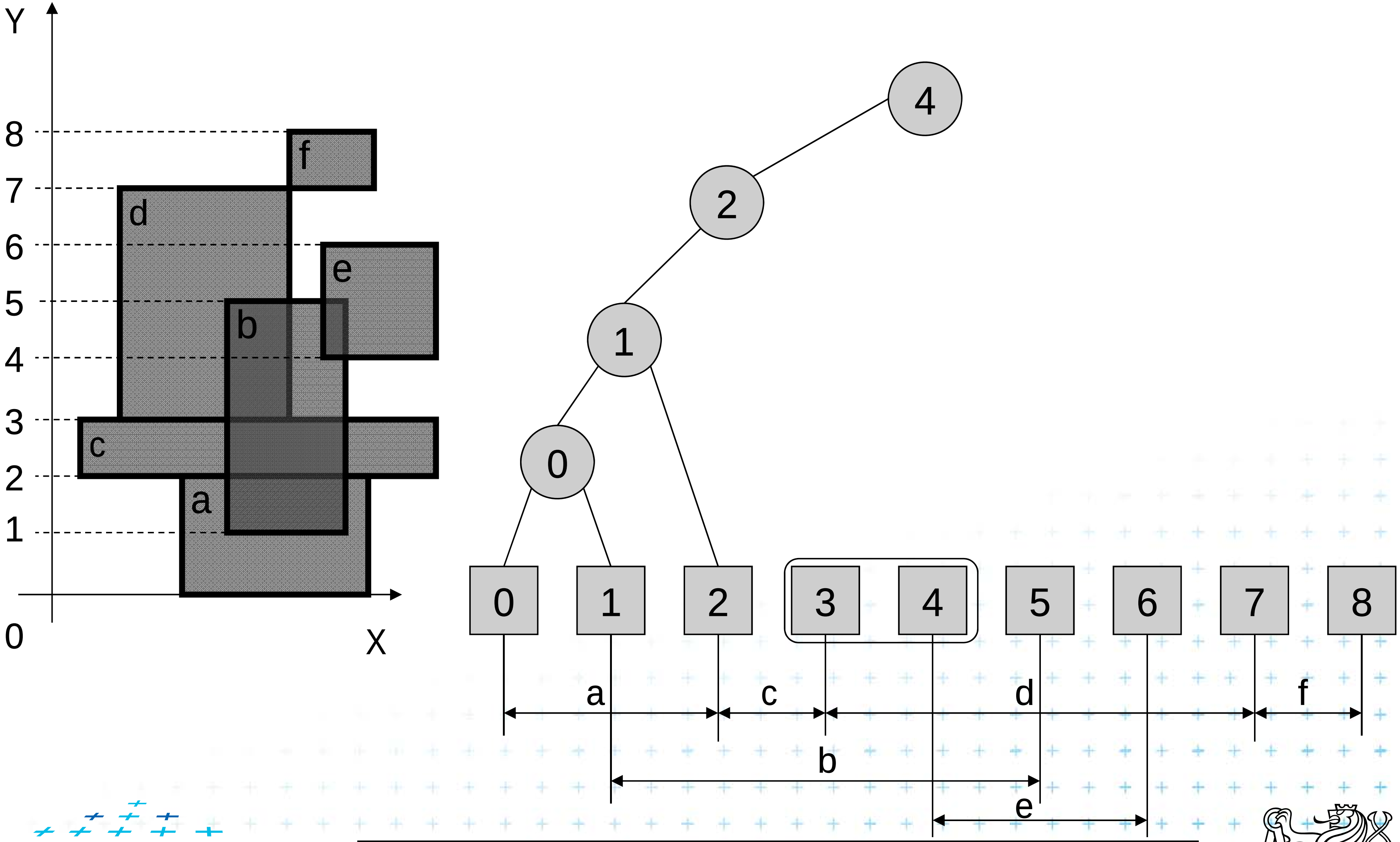
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



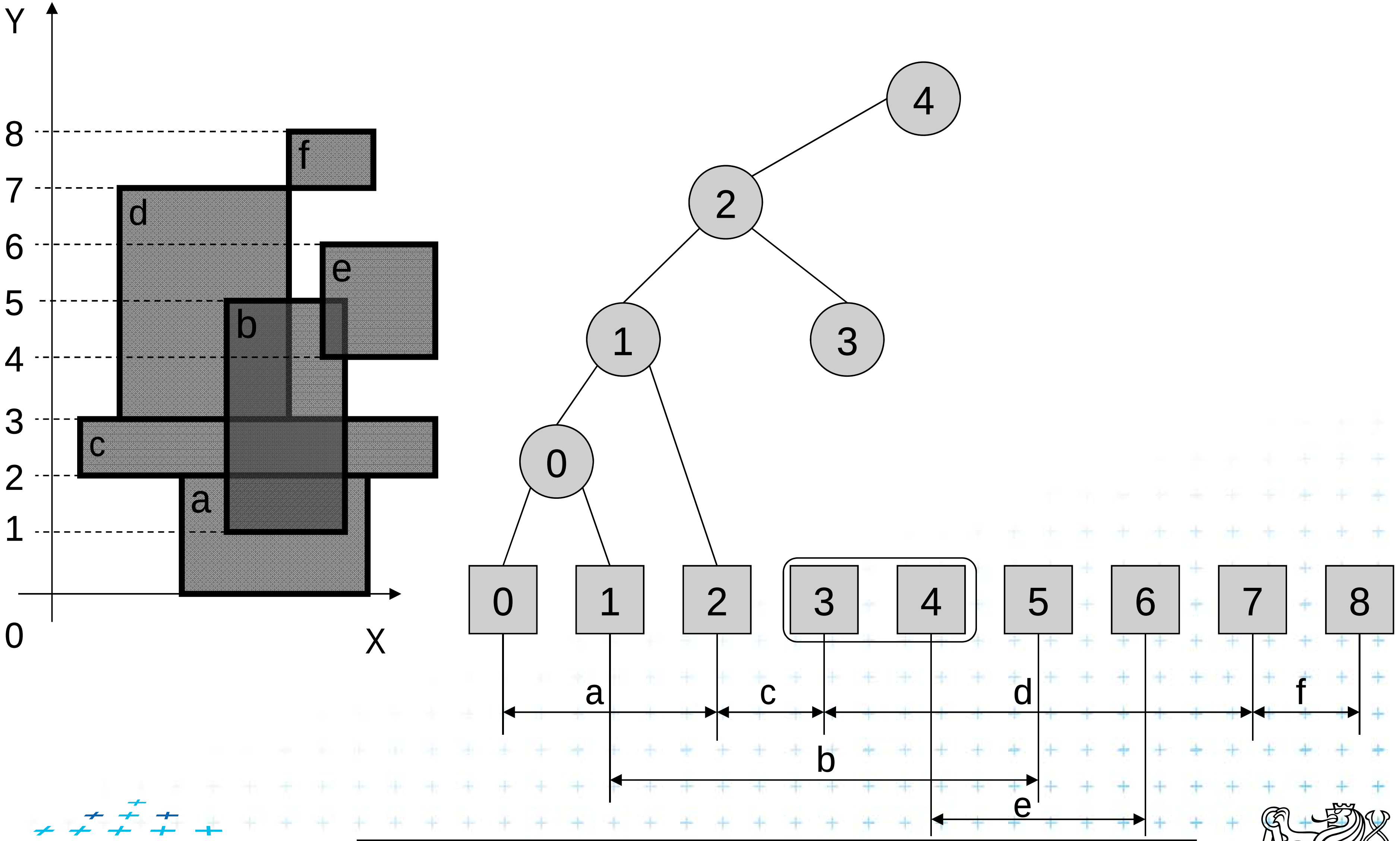
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



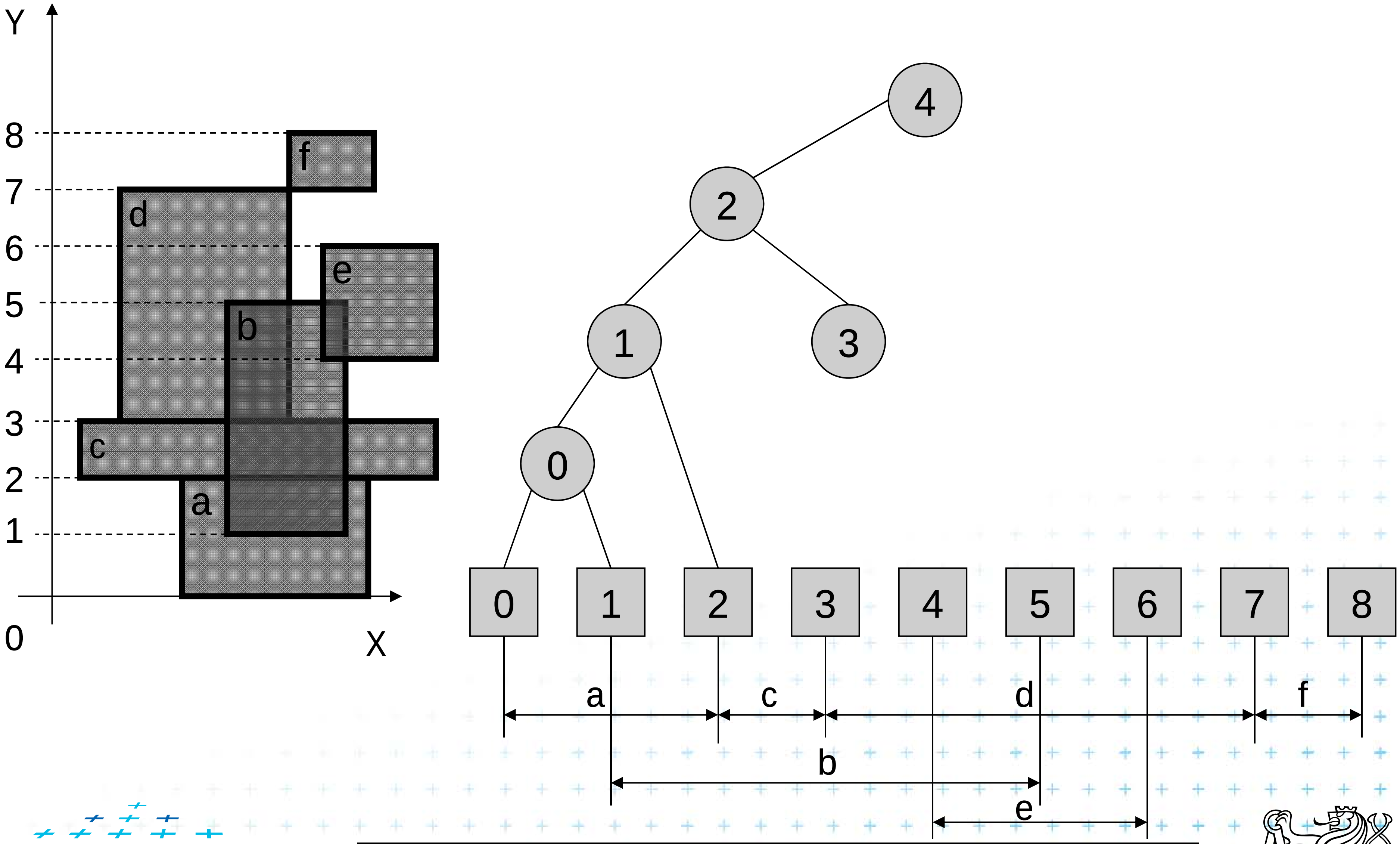
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



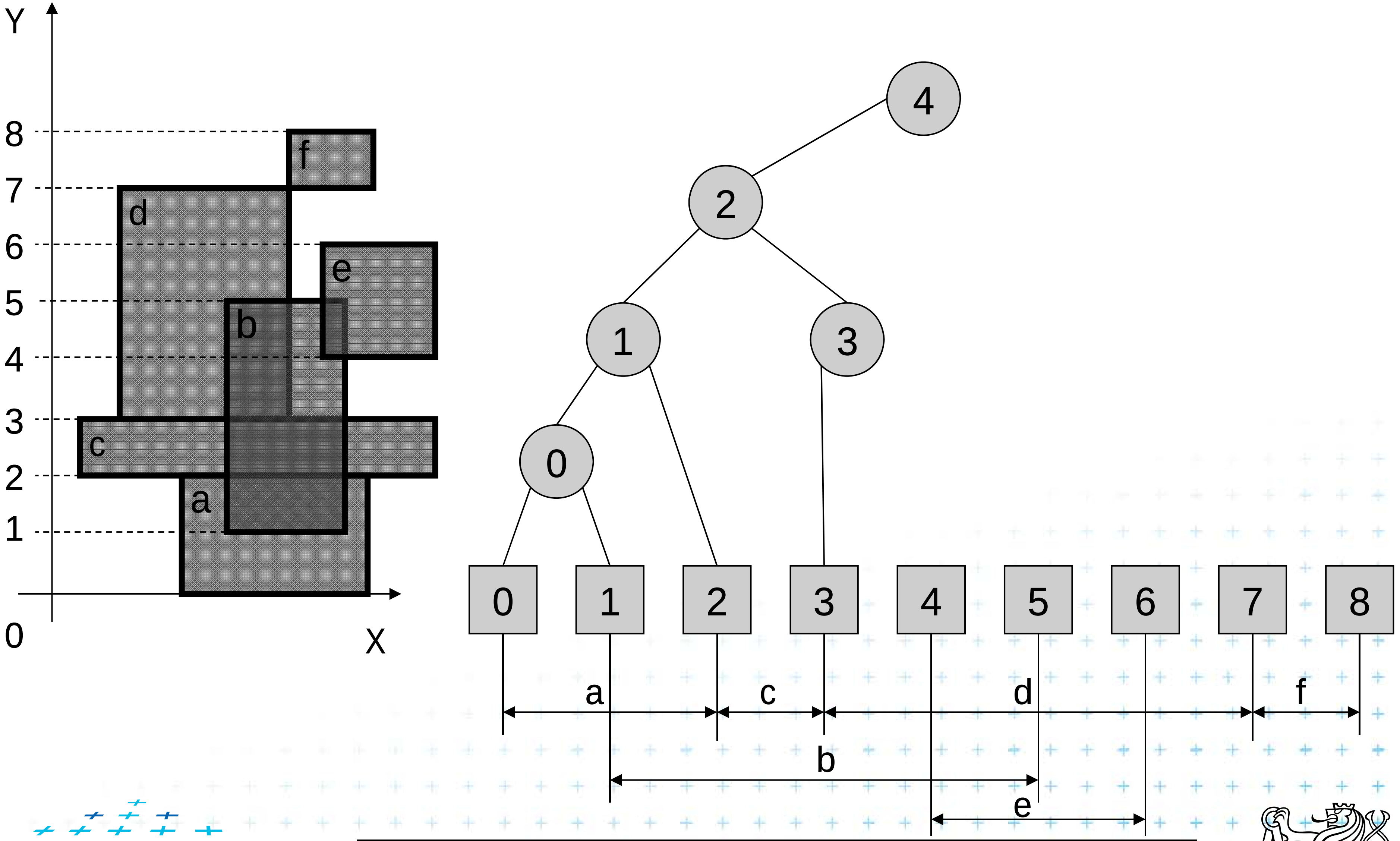
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



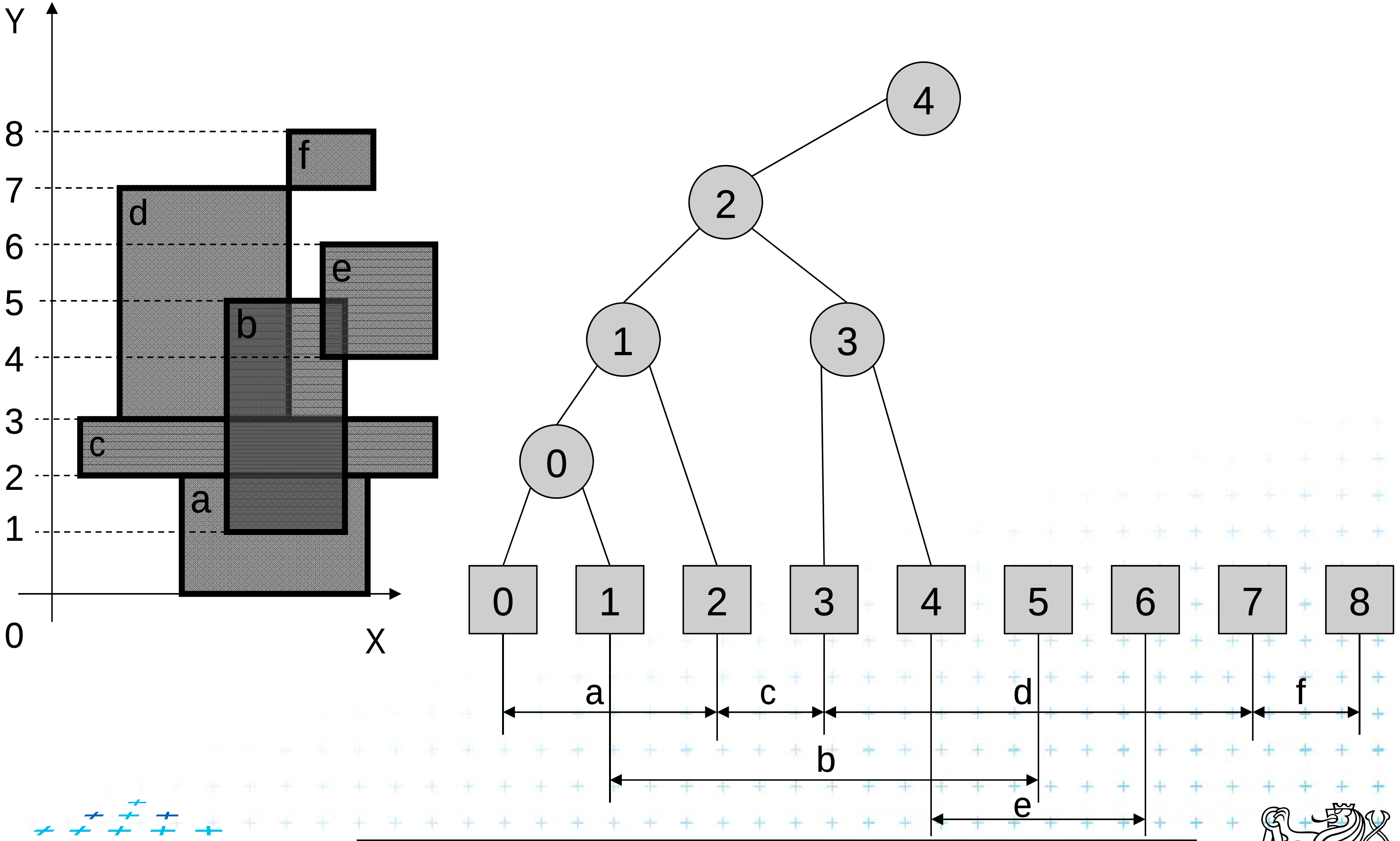
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



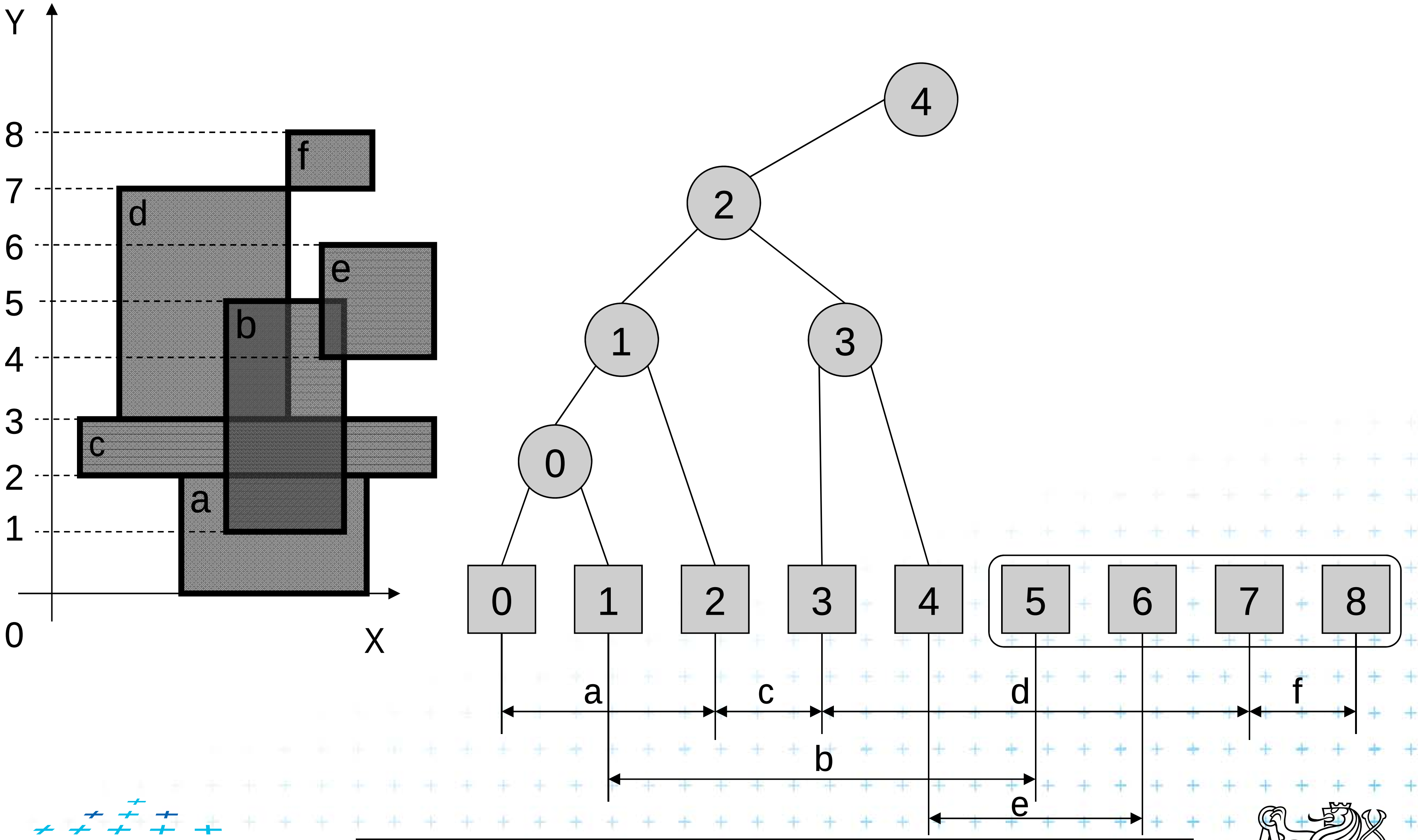
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



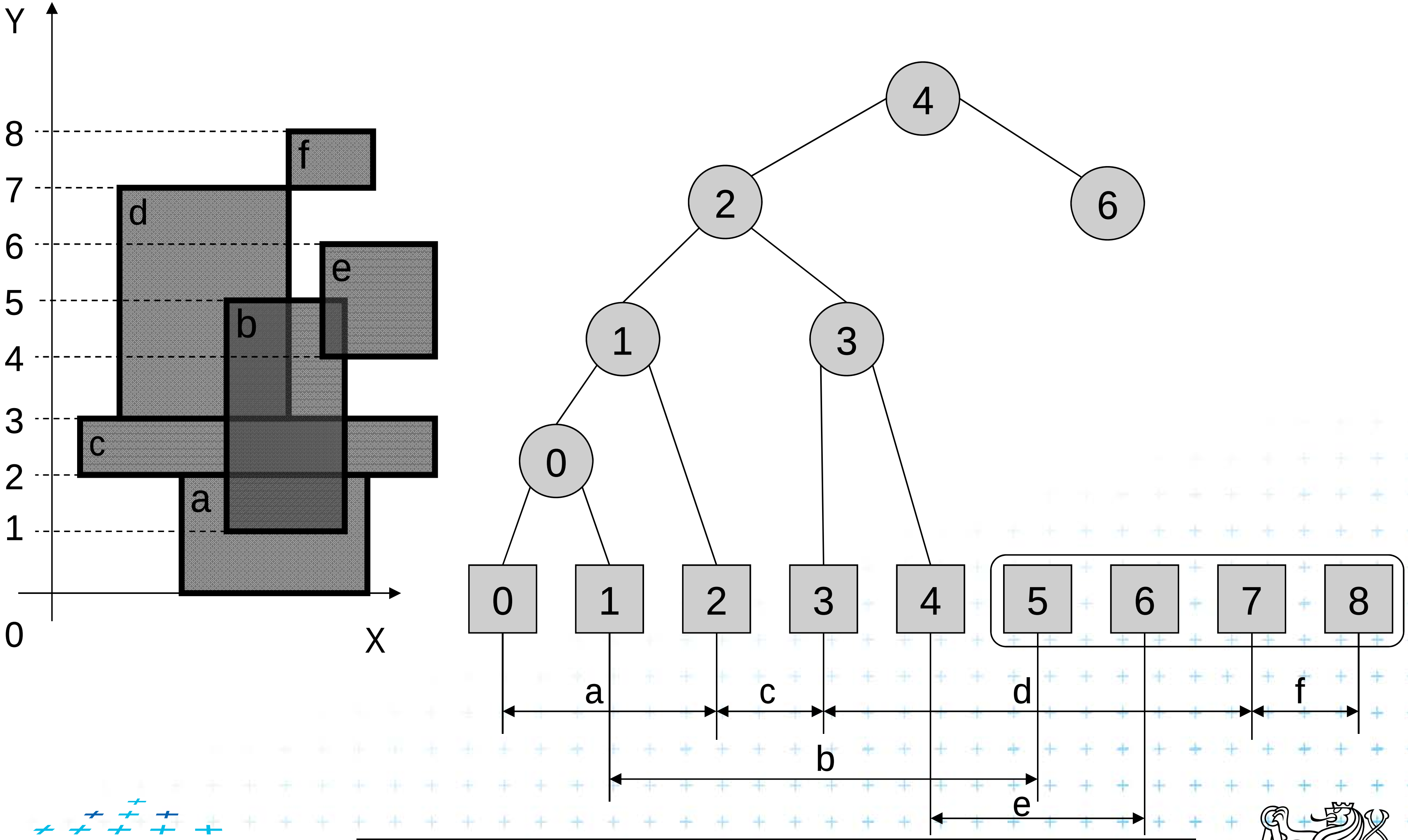
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



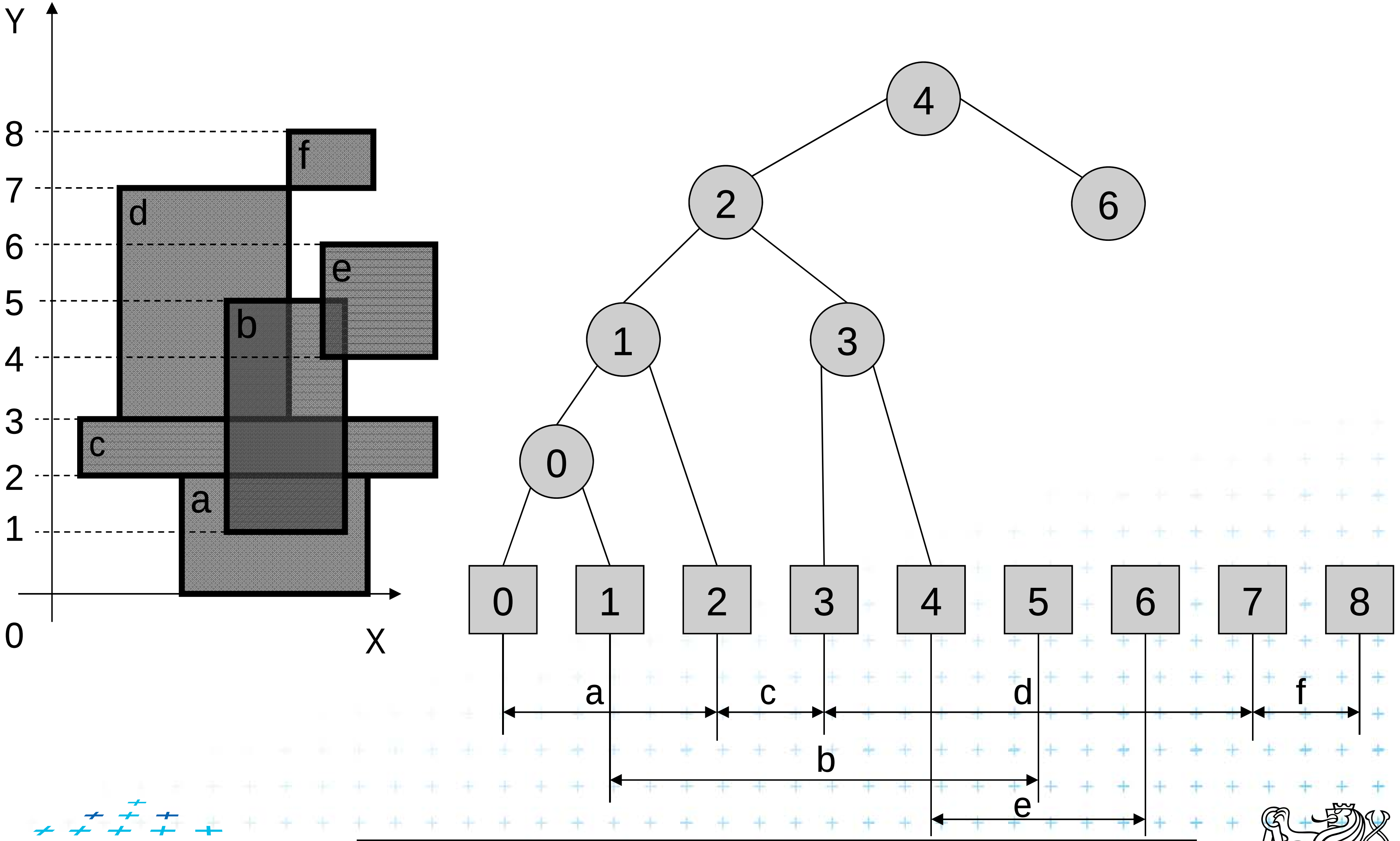
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



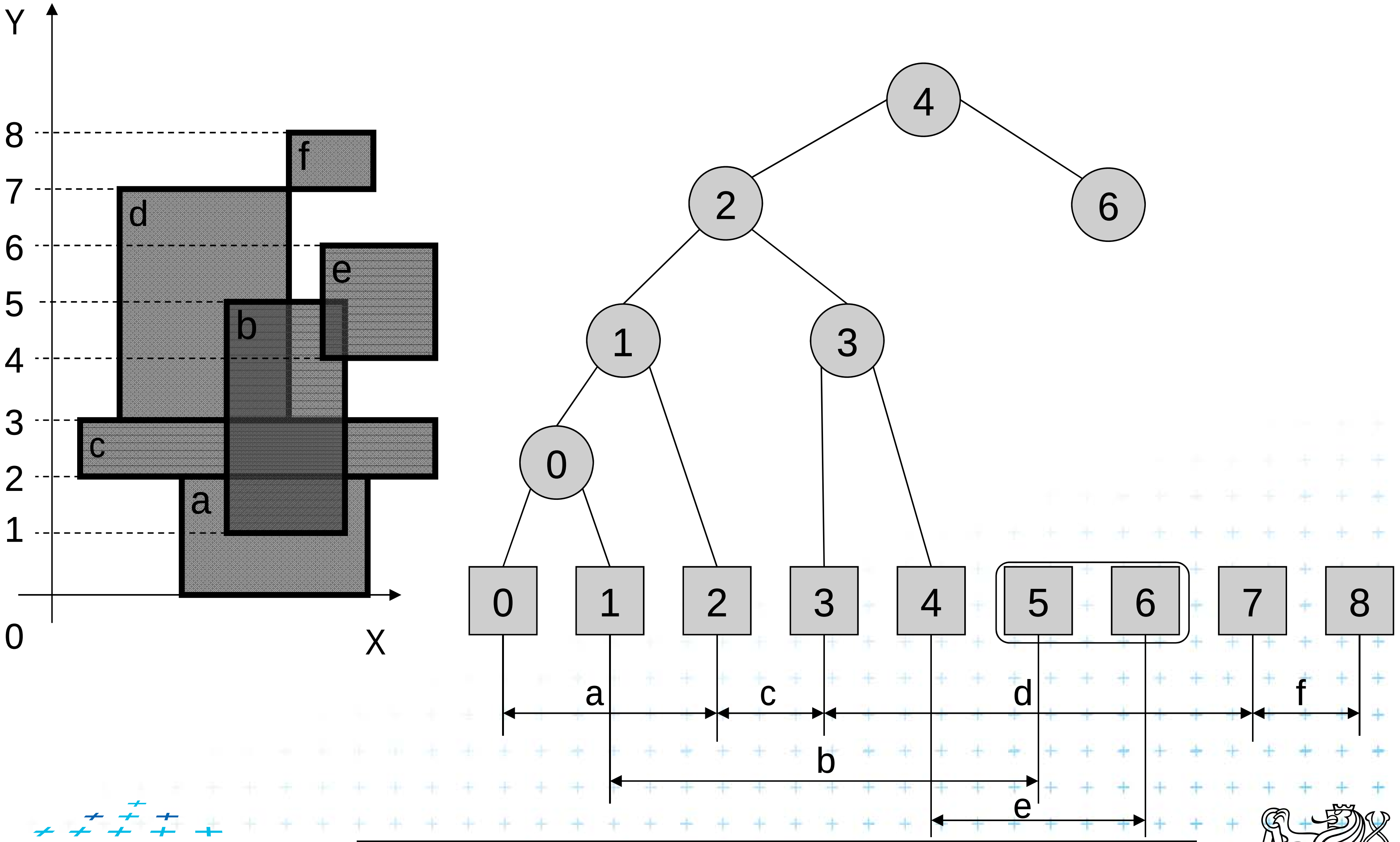
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



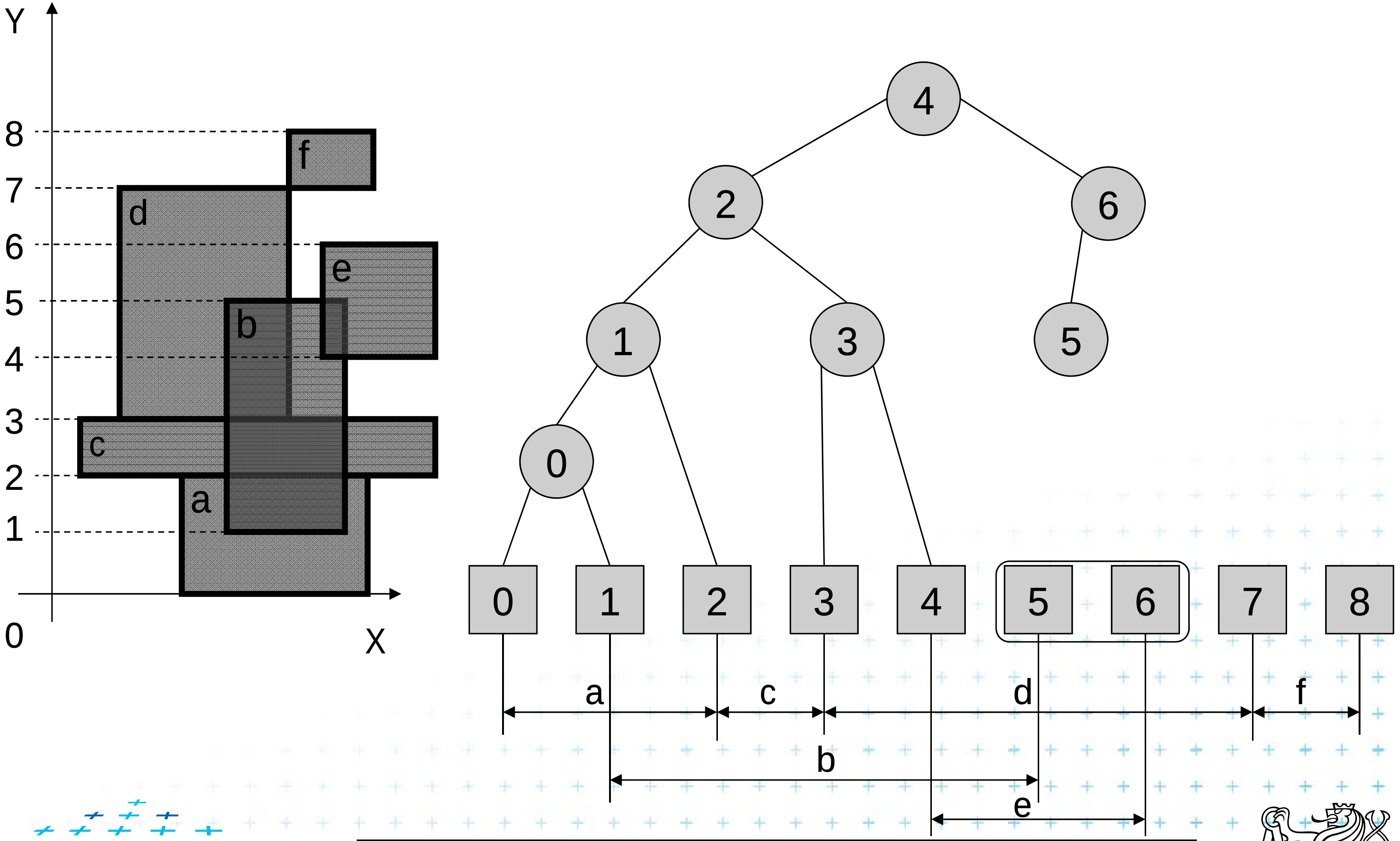
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



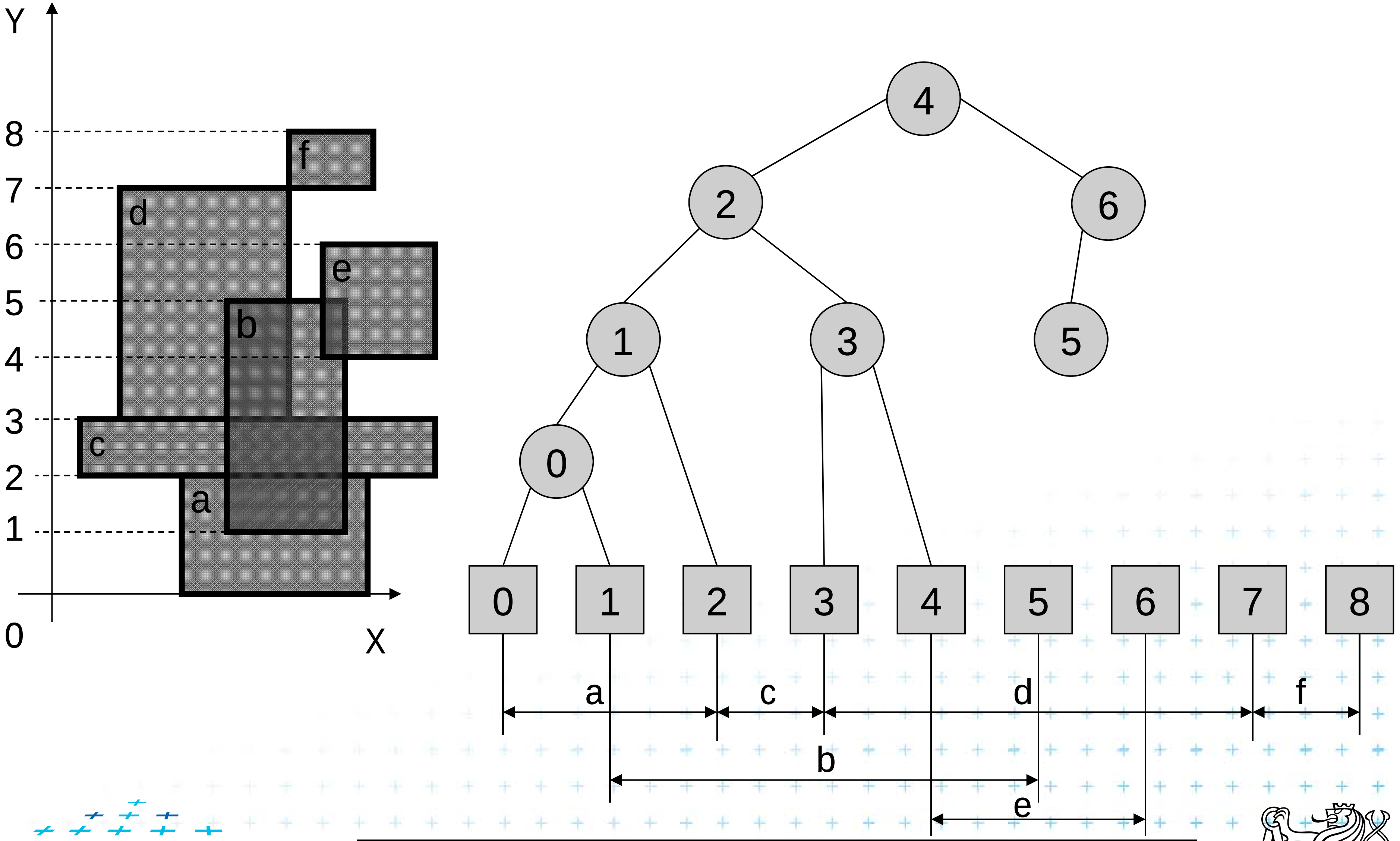
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



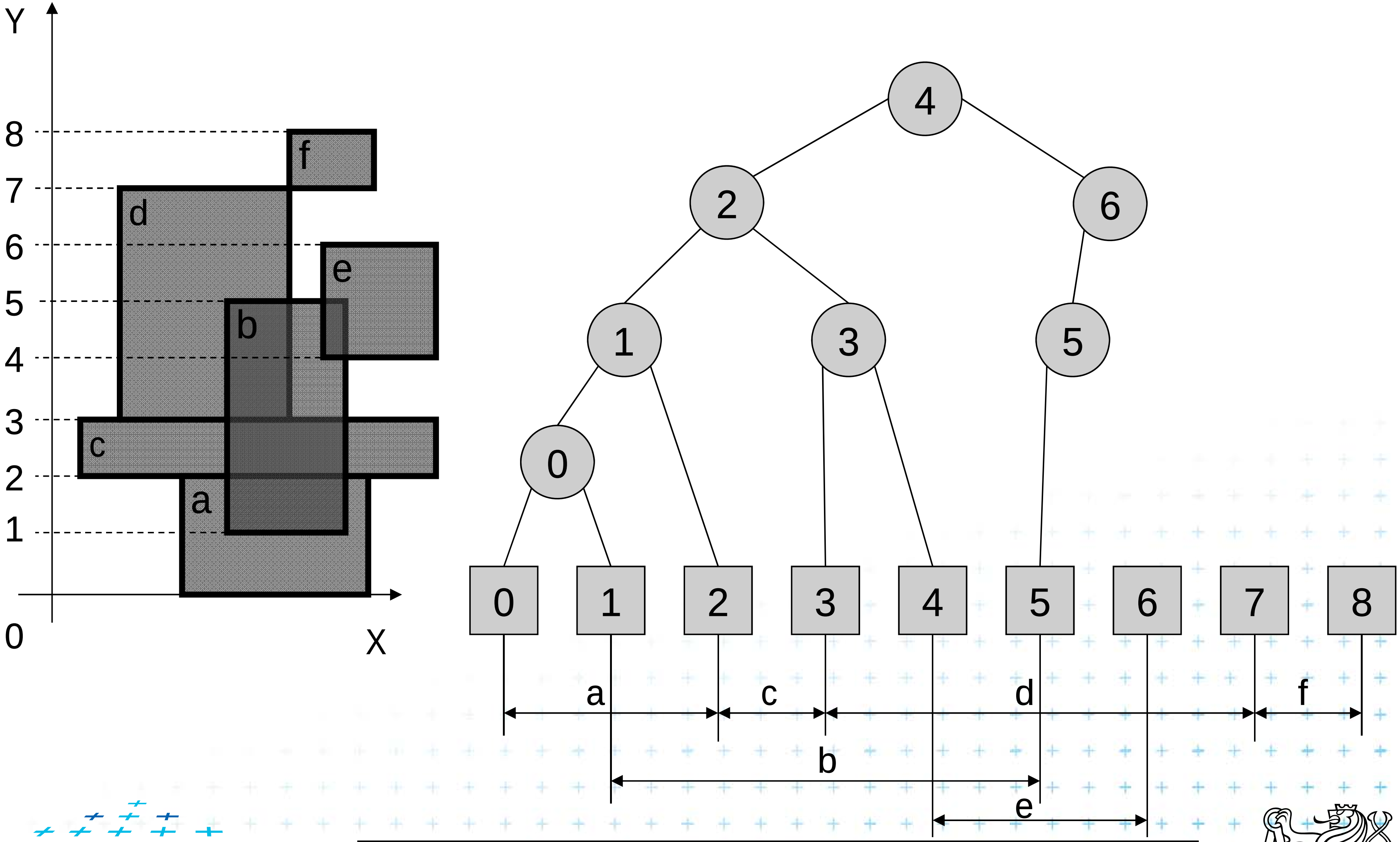
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



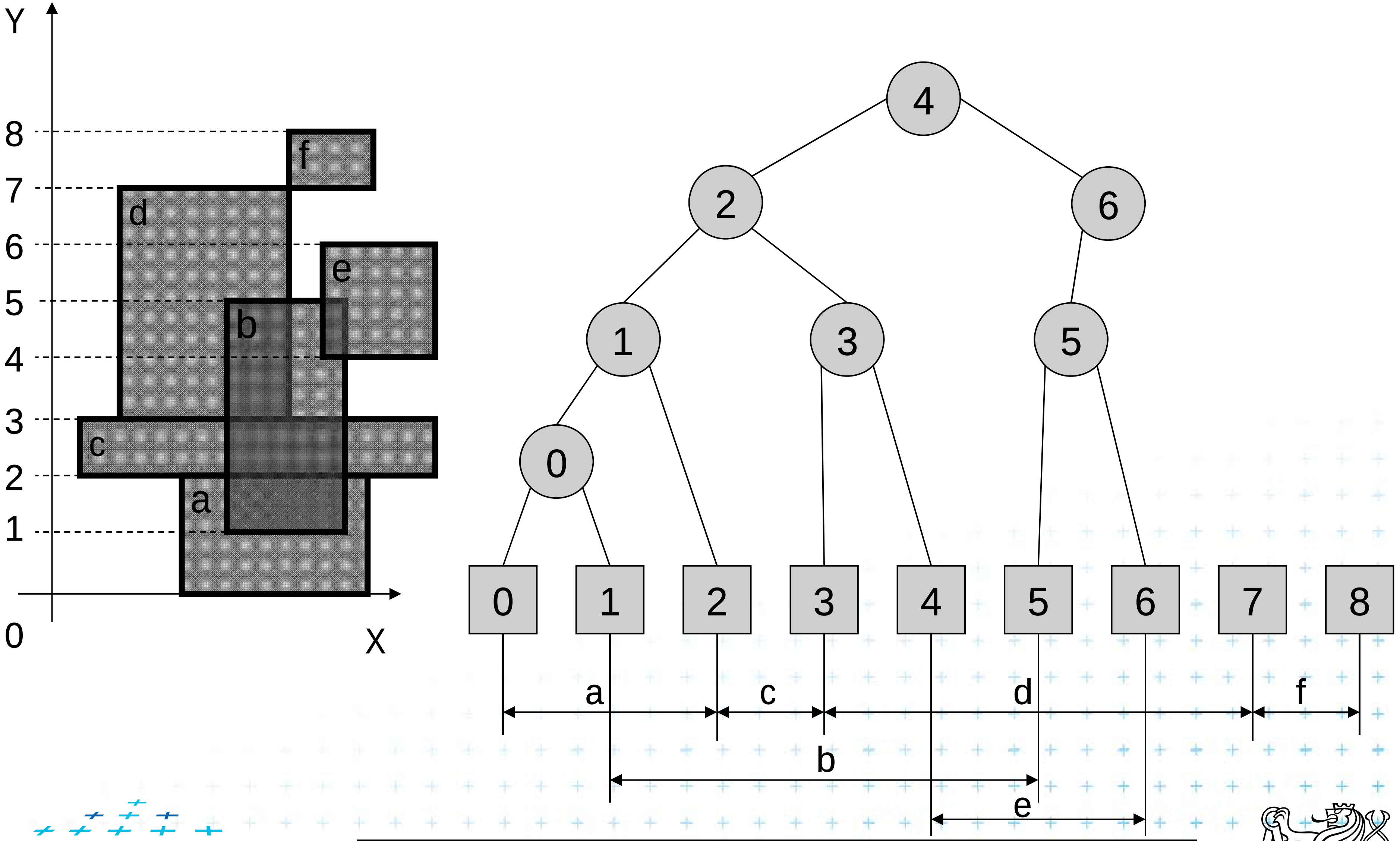
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



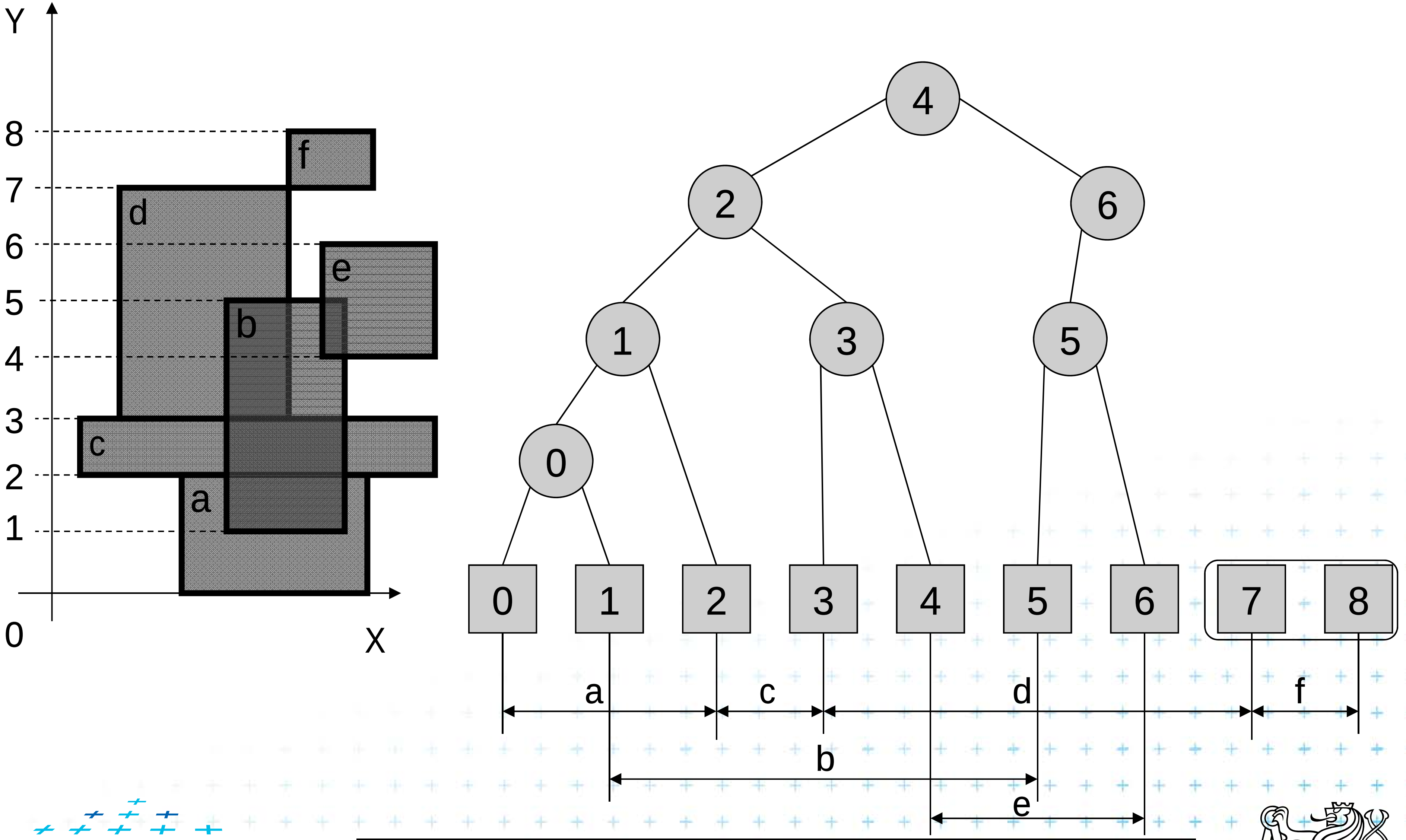
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



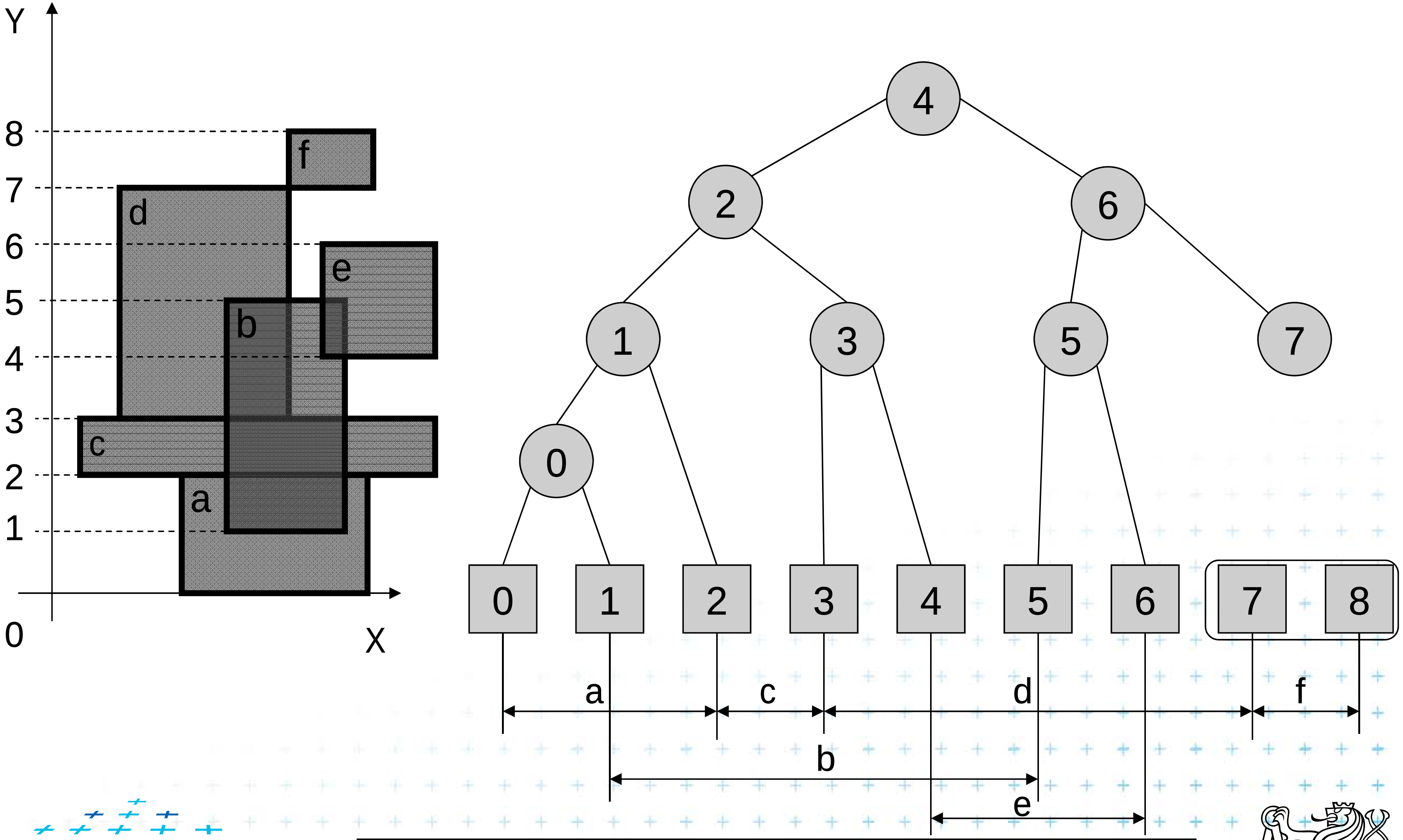
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



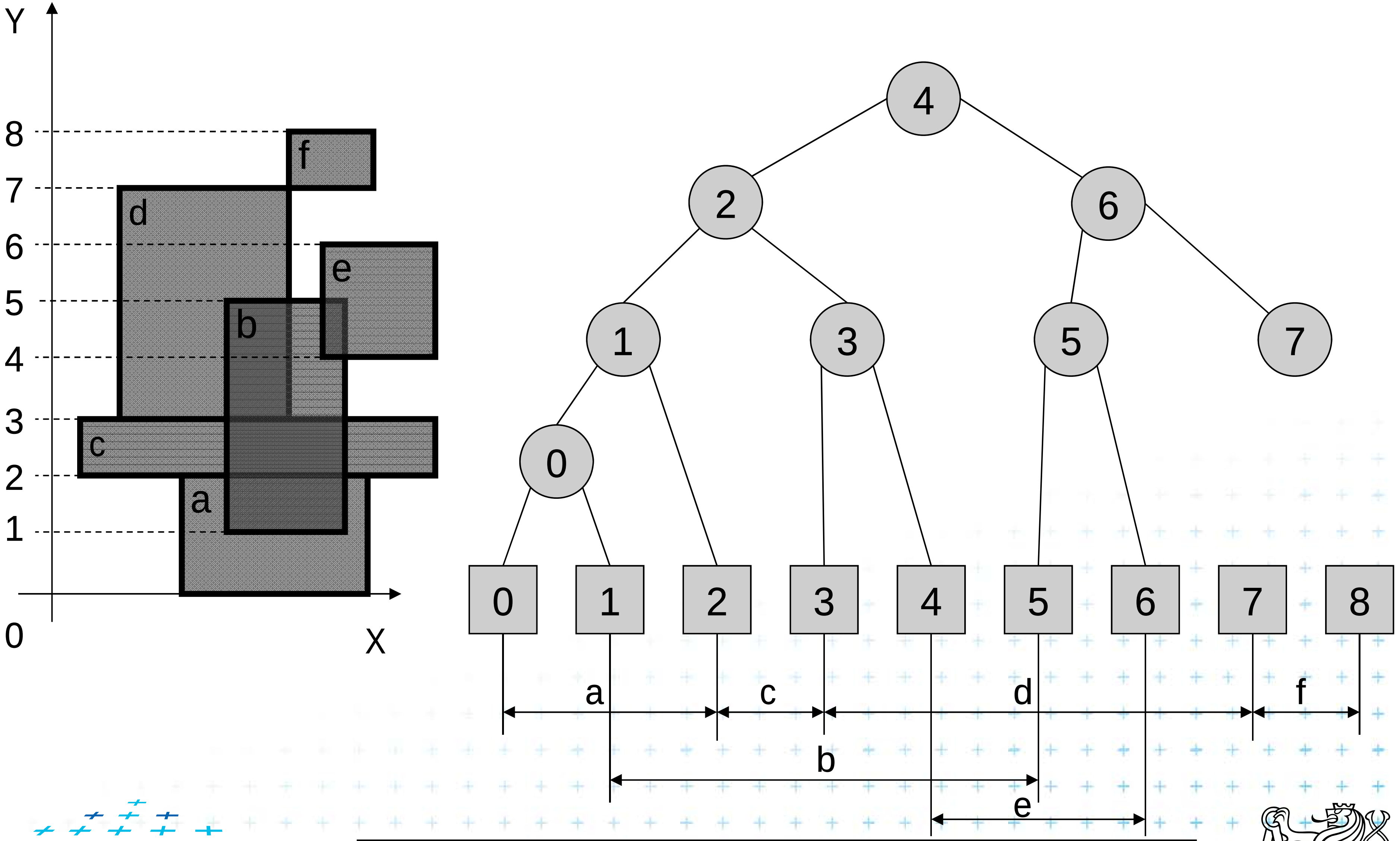
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



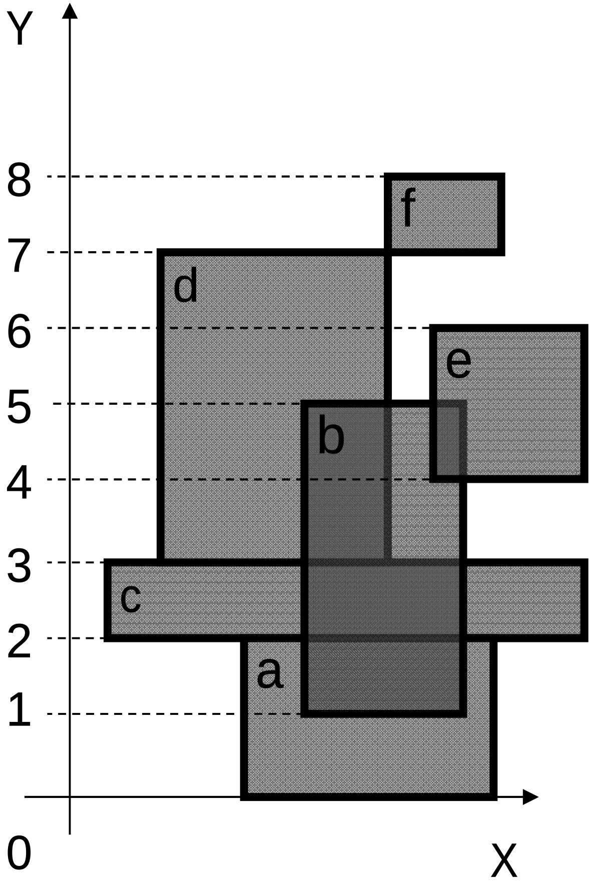
[Drtina]

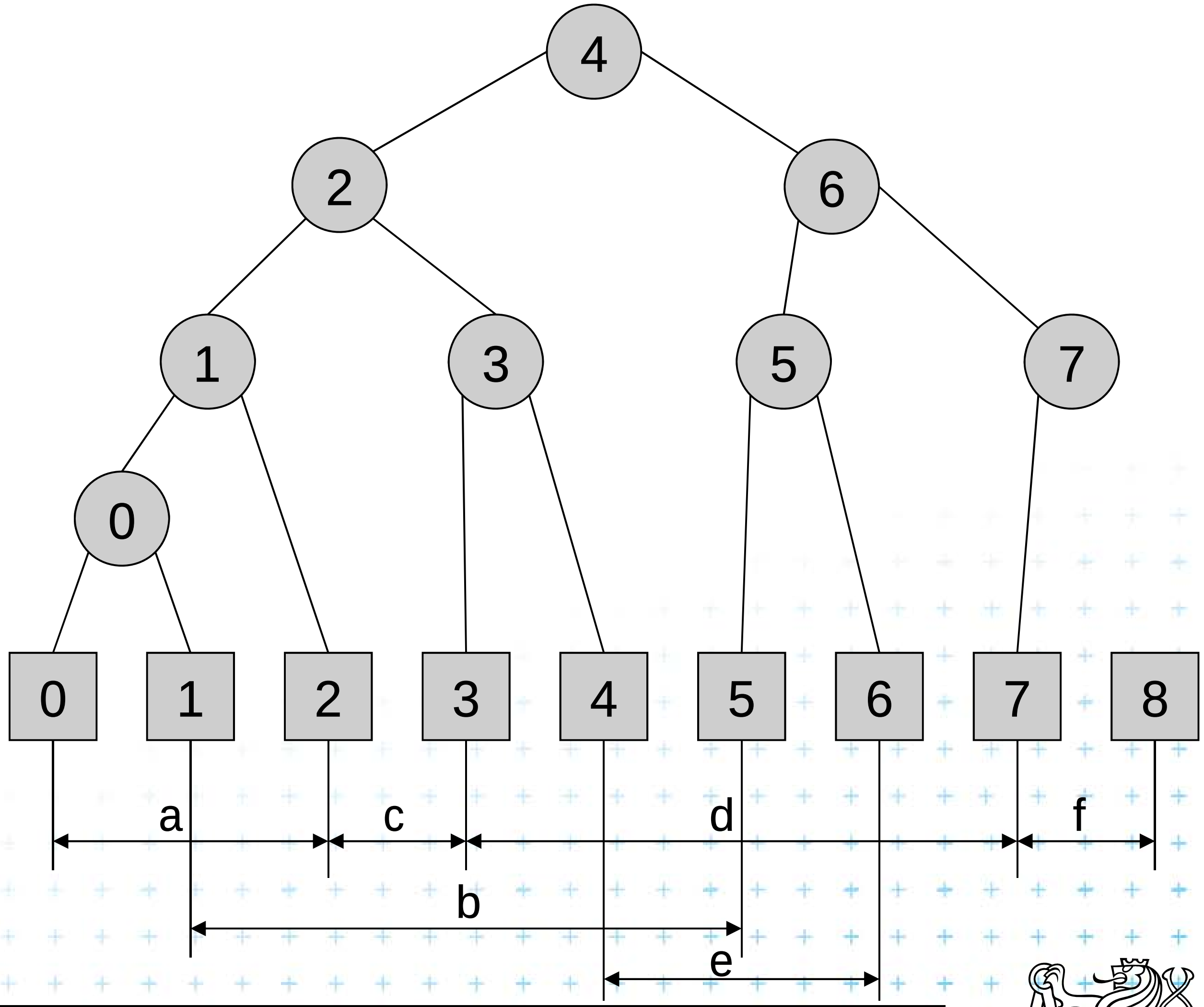
Example 2 – tree from PrimaryTree(*S*)



[Drtina]

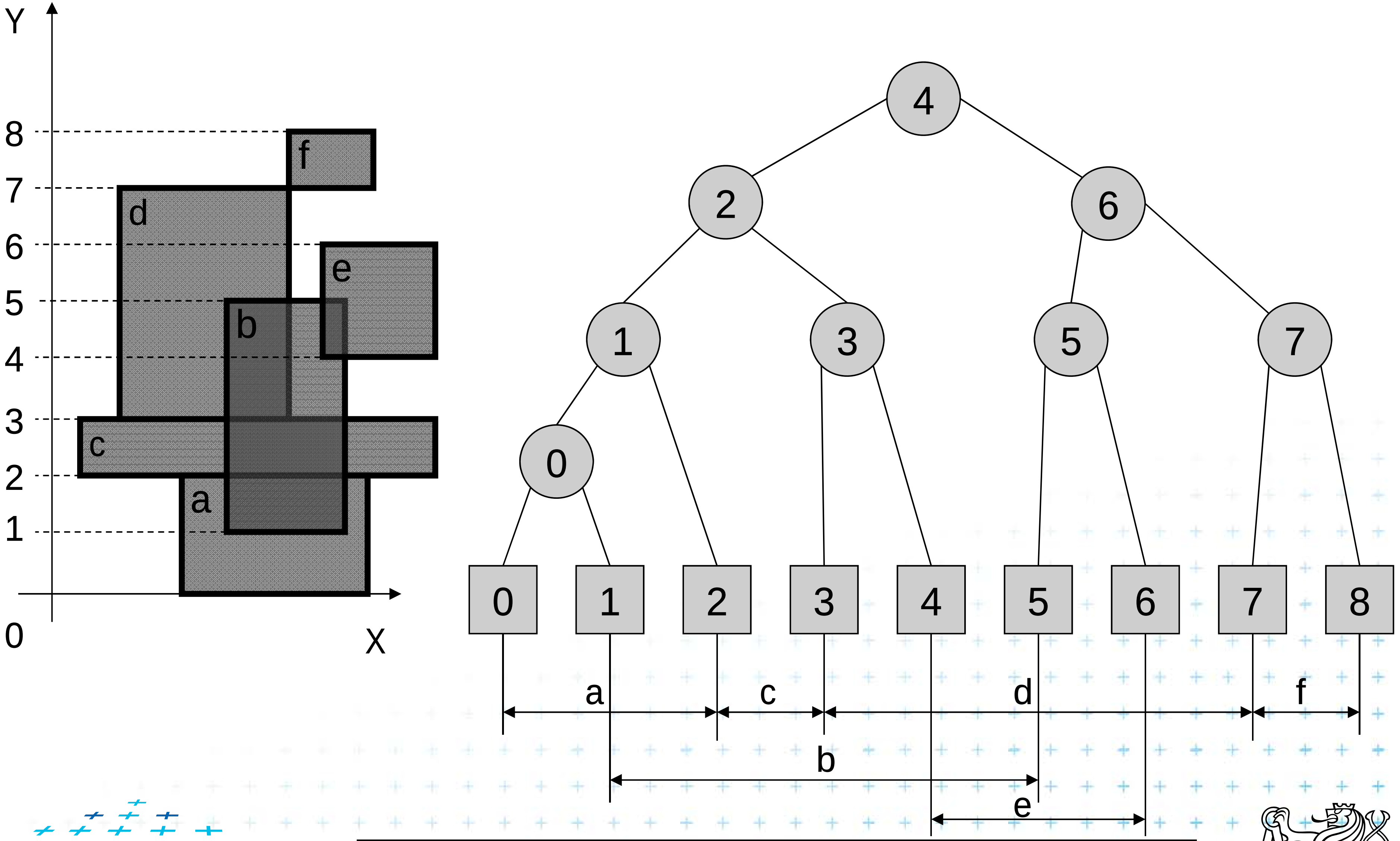
Example 2 – tree from PrimaryTree(*S*)





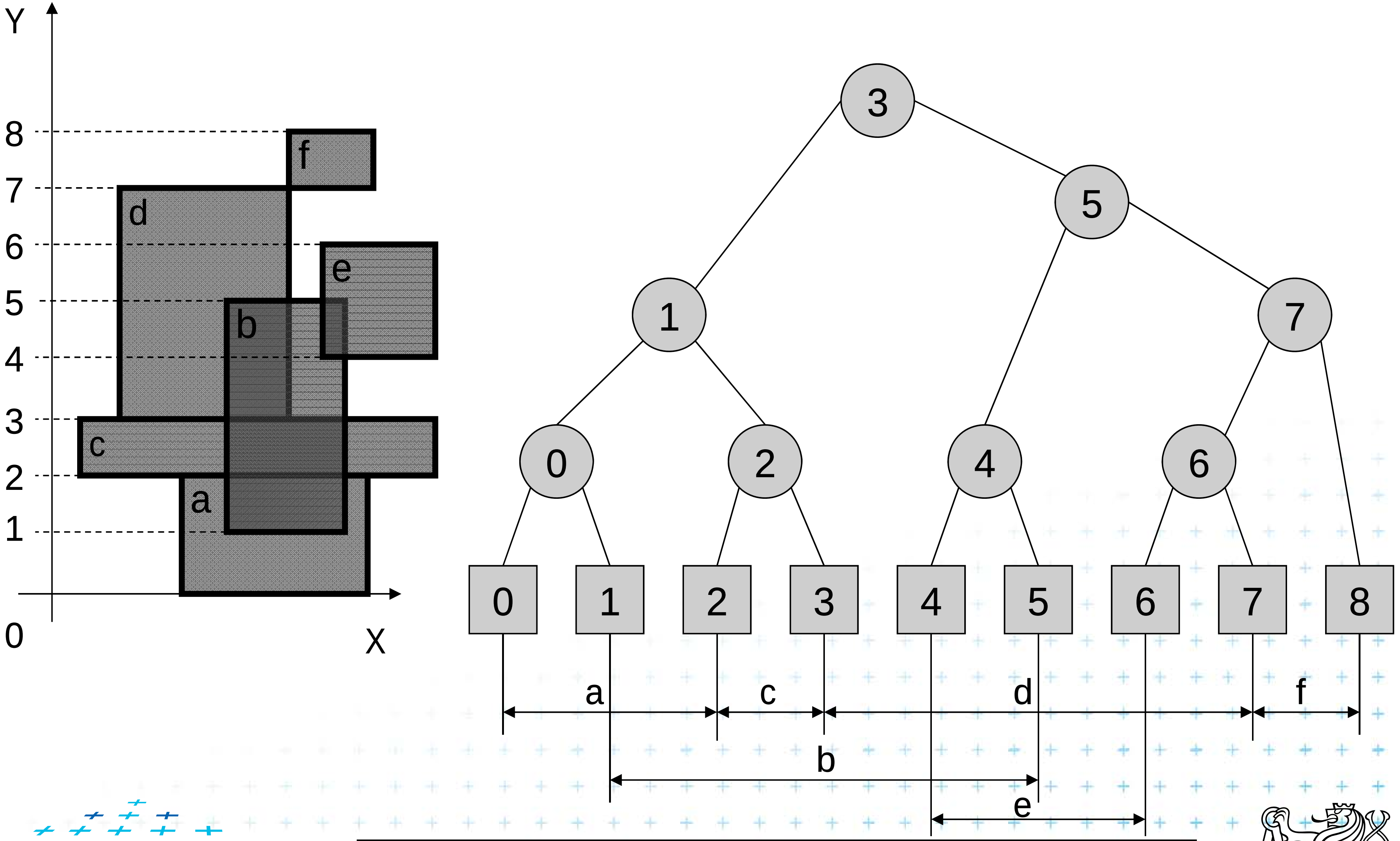
[Drtina]

Example 2 – tree from PrimaryTree(*S*)



[Drtina]

Example 2 – slightly unbalanced tree

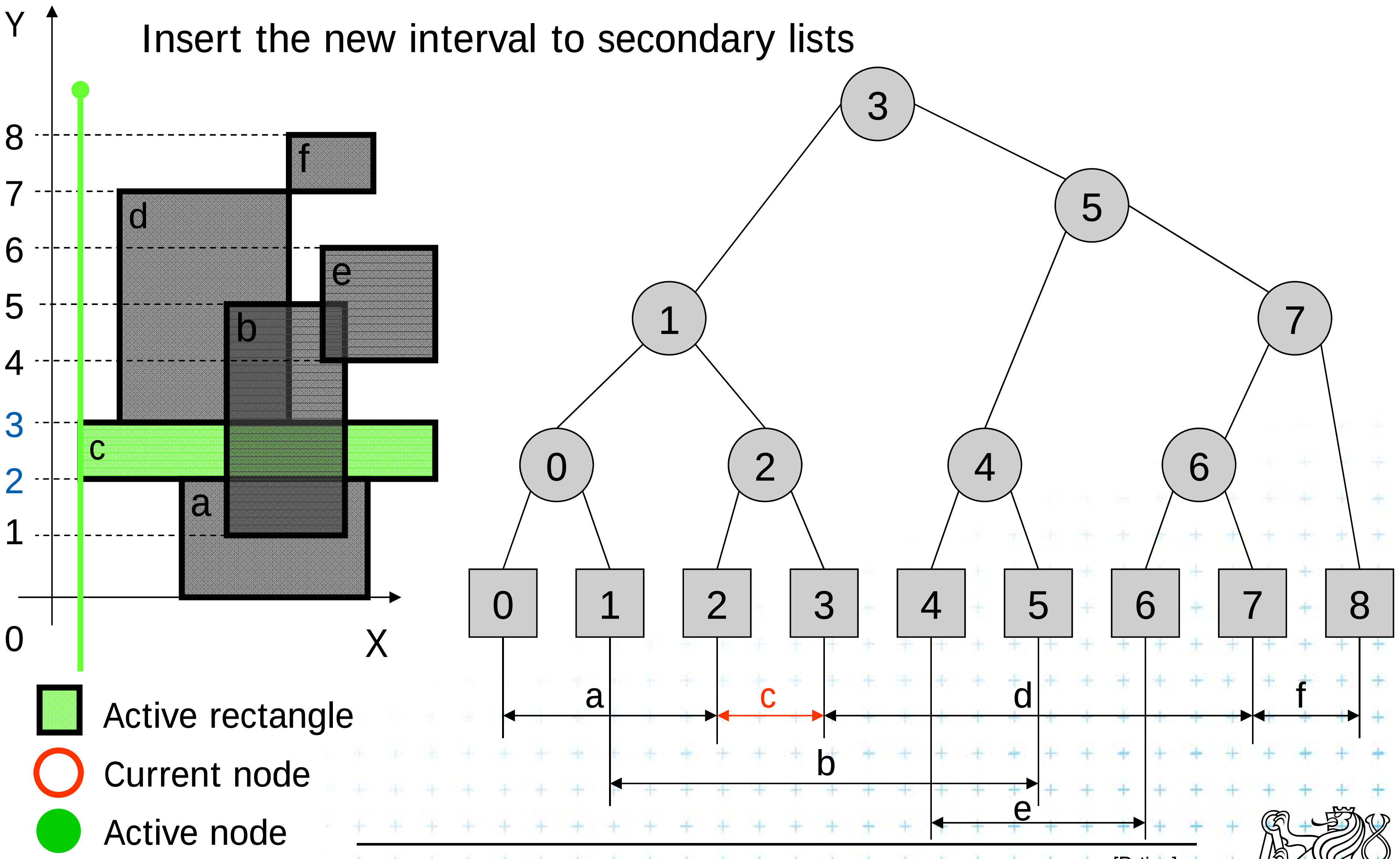


[Drtina]

Insert [2,3] – empty => b) Insert Interval

b ≤ H(v) ≤ e

Insert the new interval to secondary lists

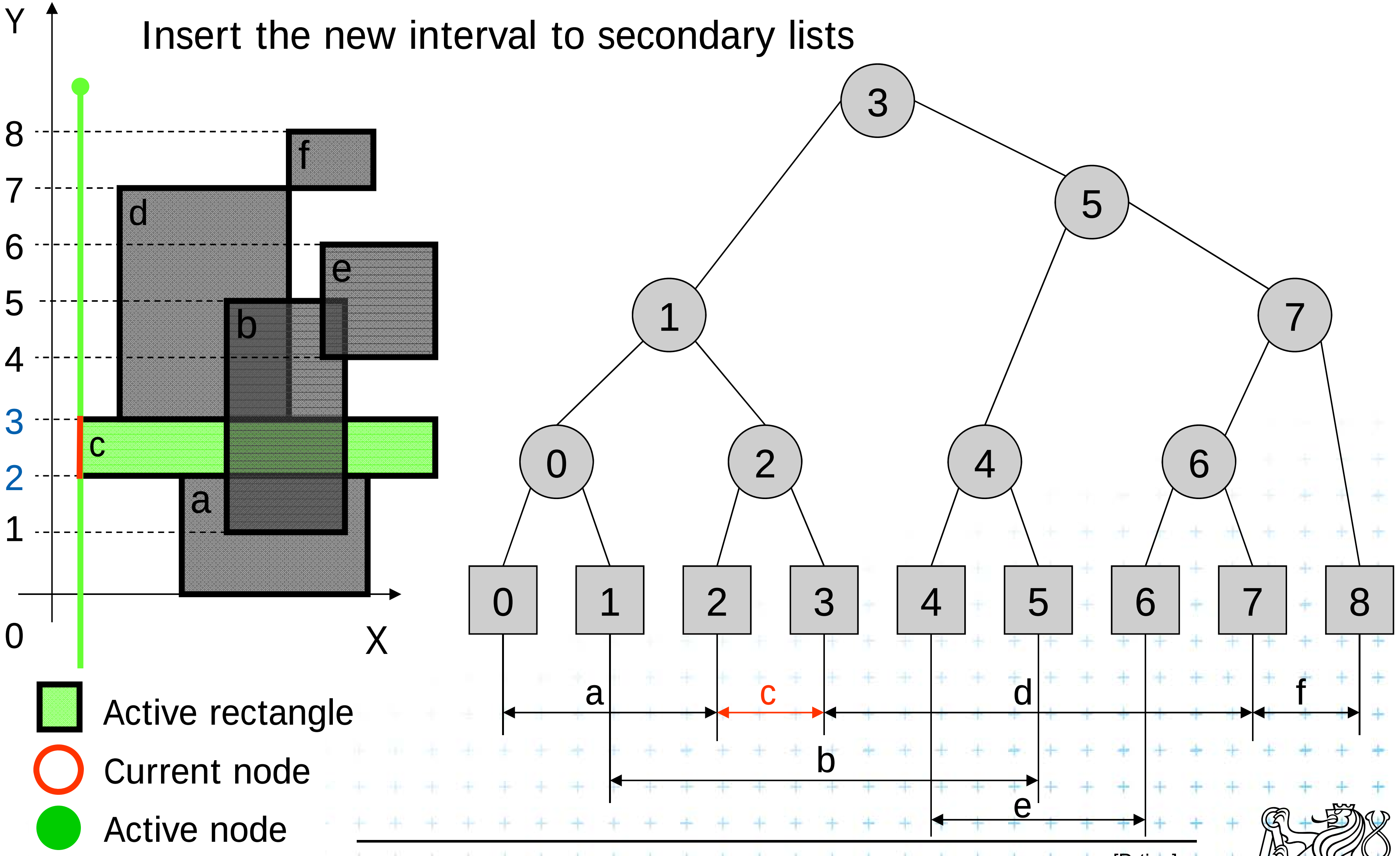


[Drtina]

Insert [2,3] – empty => b) Insert Interval

$b \leq H(v) \leq e$

Insert the new interval to secondary lists

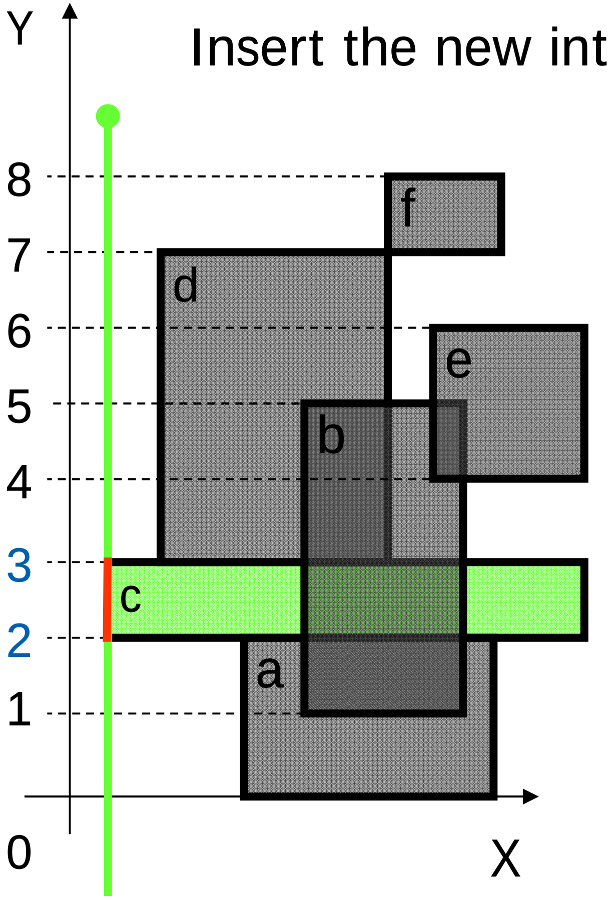


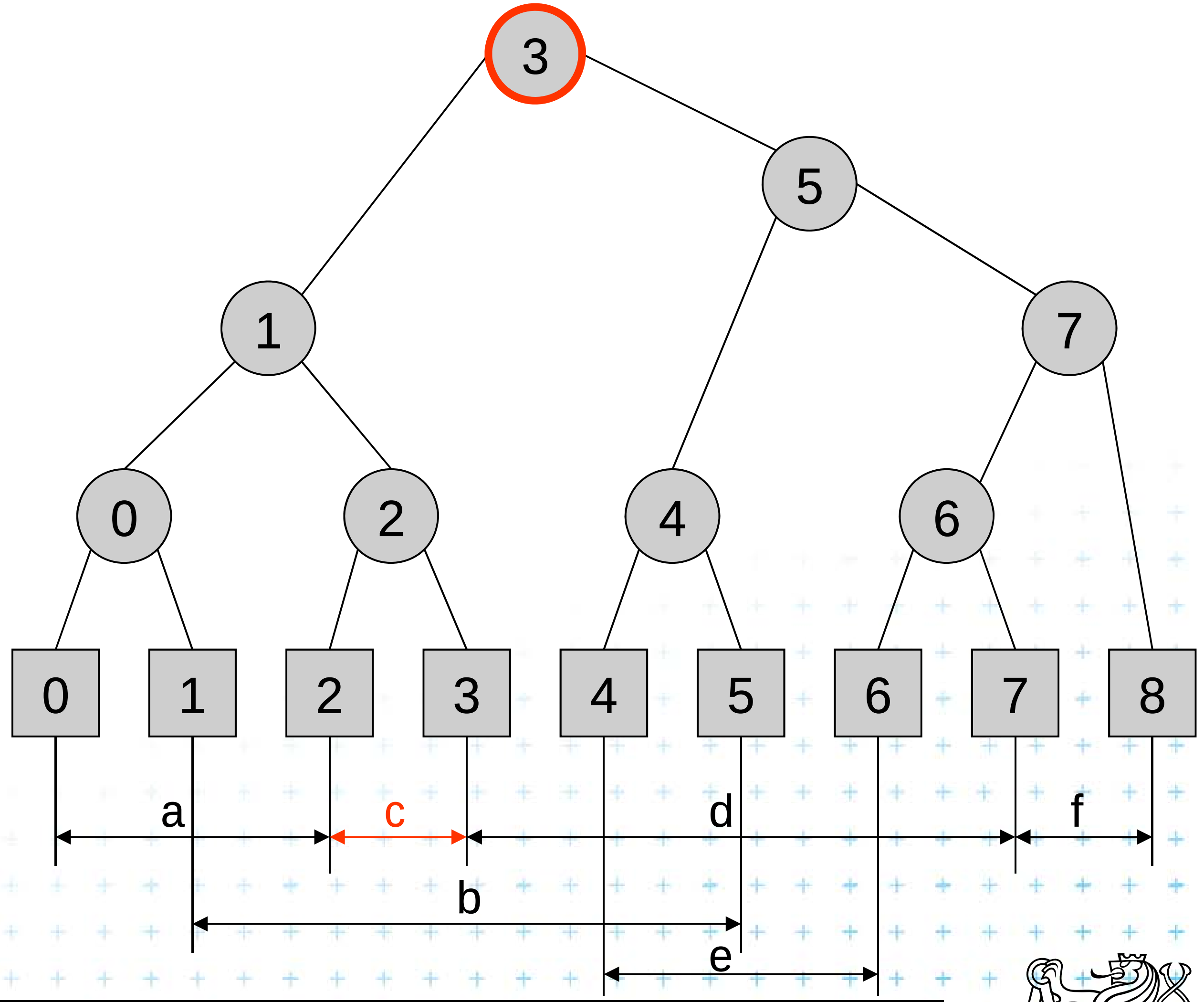
[Drtina]

Insert [2,3] – empty => b) Insert Interval

b ≤ H(v) ≤ e

Insert the new interval to secondary lists





🟩 Active rectangle

○ Current node

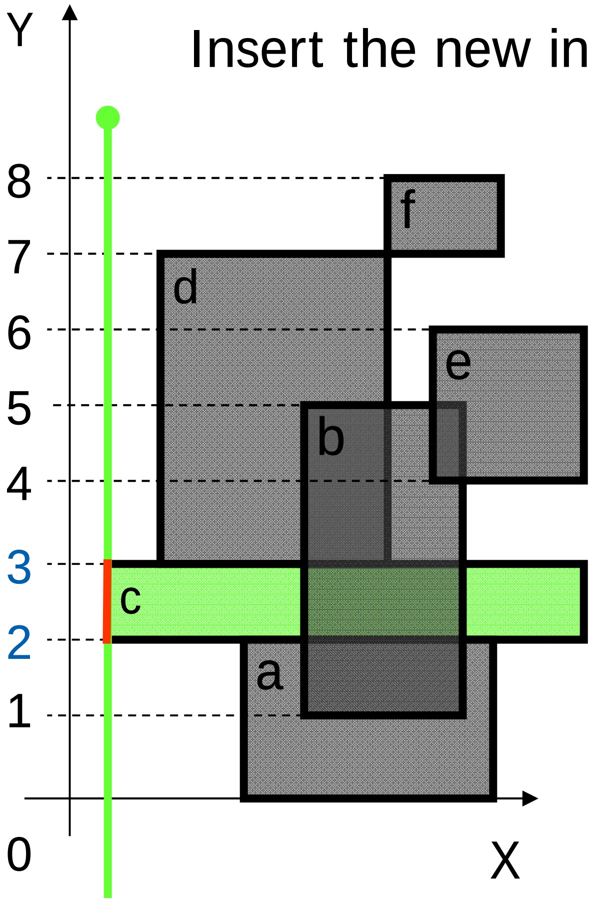
● Active node

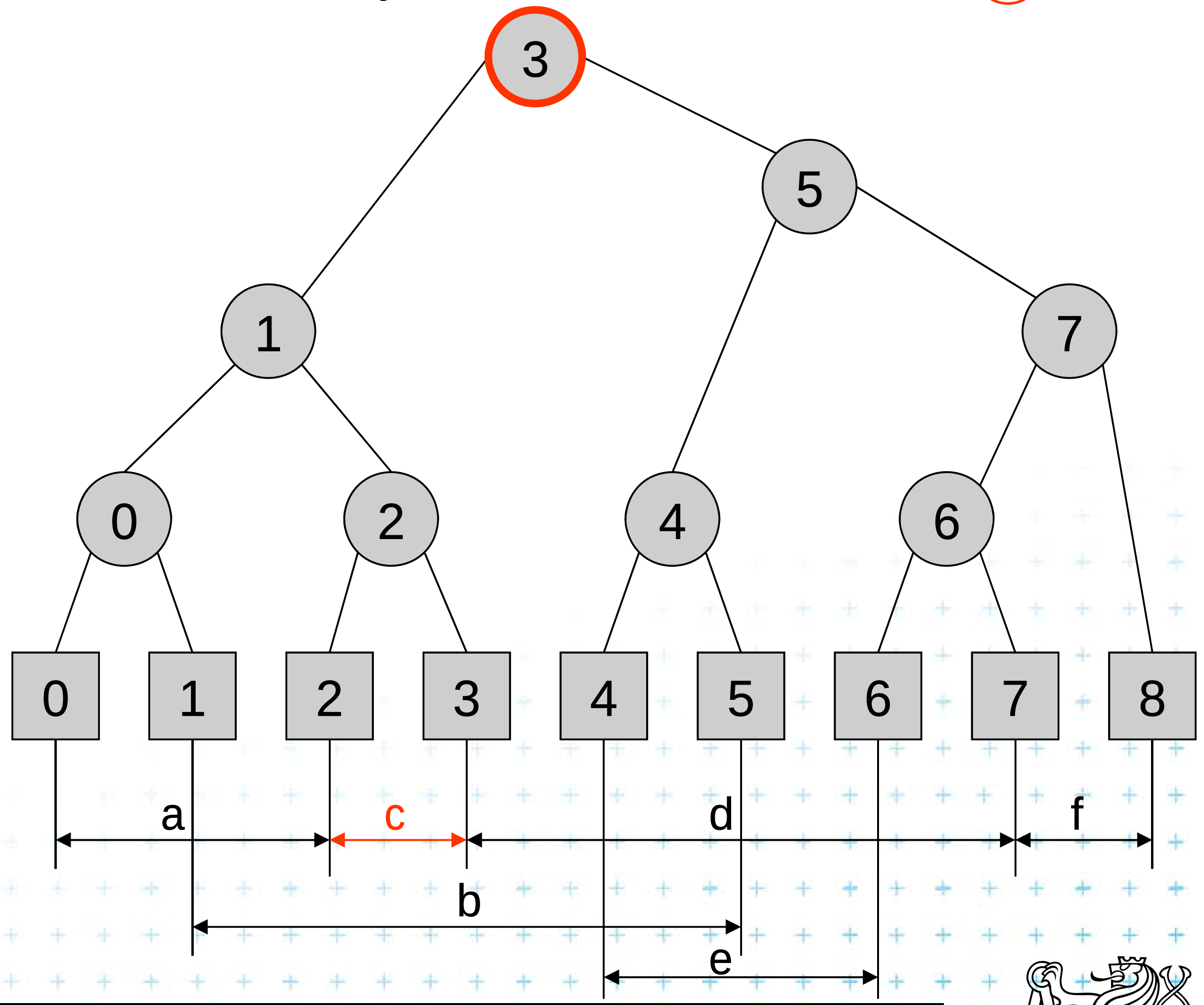
[Drtina]

Insert [2,3] – empty => b) Insert Interval $b \le H(v) \le e$

Insert the new interval to secondary lists

? 2 ≤ 3 ≤ 3 ?





🟩 Active rectangle

○ Current node

● Active node

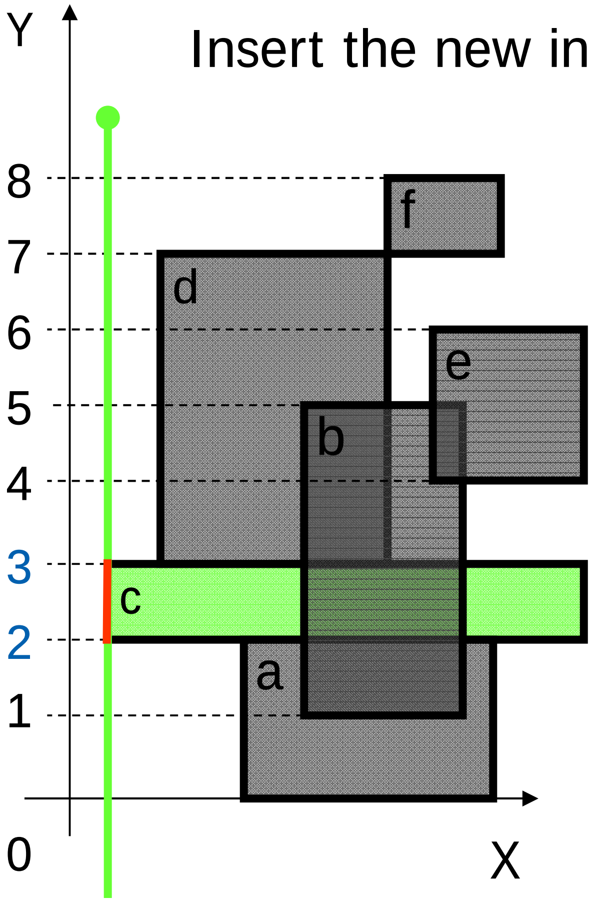
[Drtina]

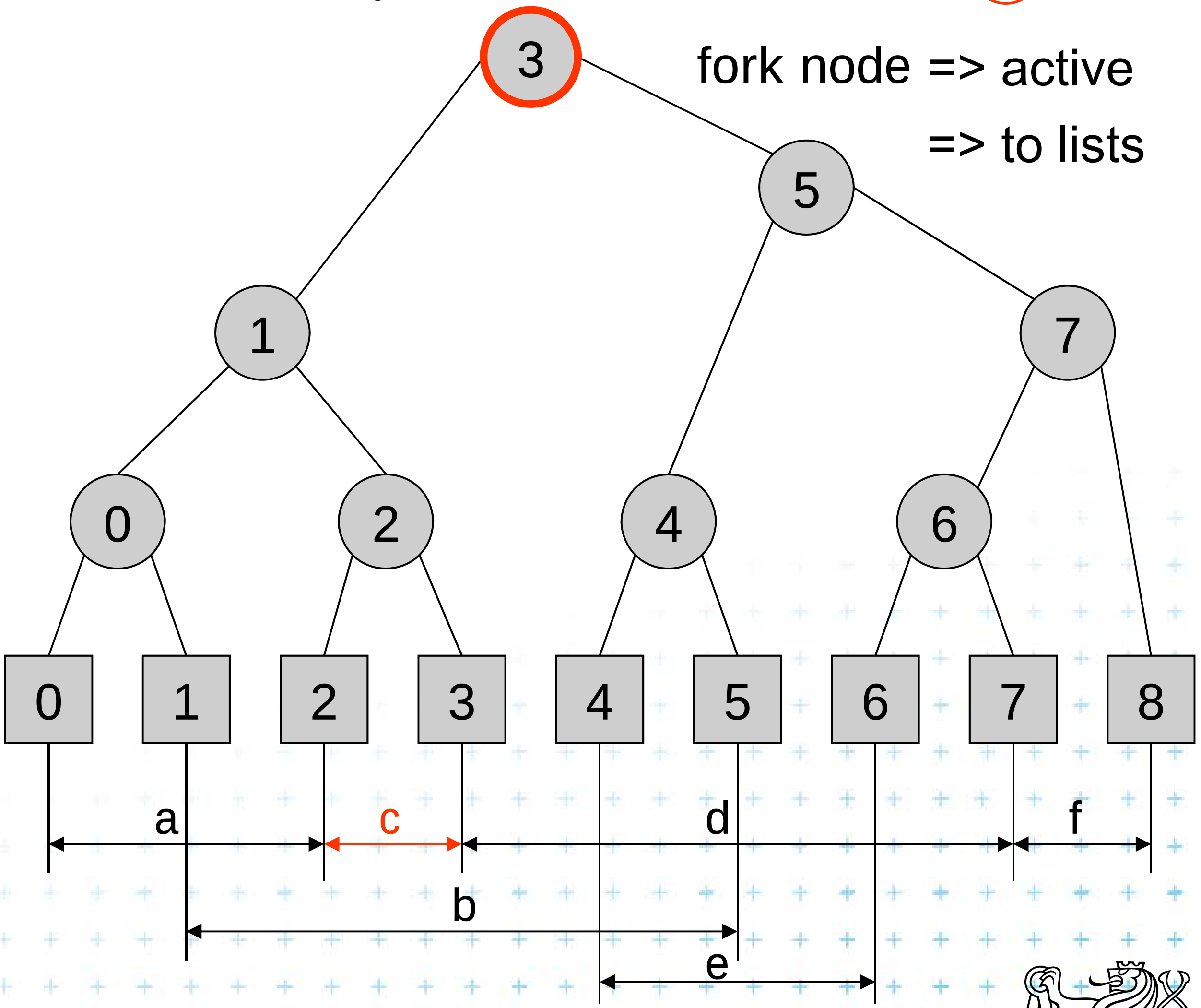
Insert [2,3] – empty => b) Insert Interval

b ≤ H(v) ≤ e

Insert the new interval to secondary lists

? 2 ≤ 3 ≤ 3 ?





fork node => active

=> to lists

🟩 Active rectangle

○ Current node

● Active node

[Drtina]

Insert [2,3] – empty => b) Insert Interval

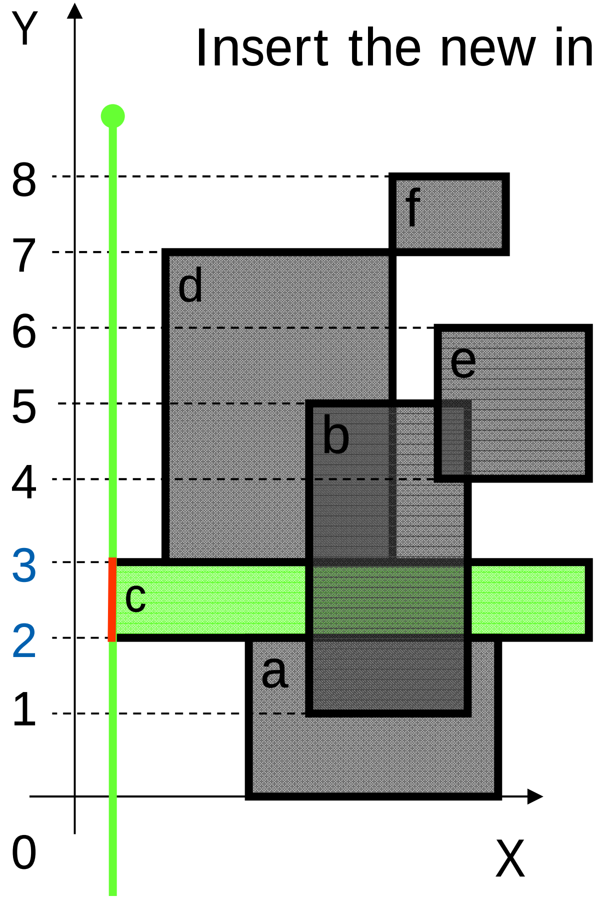
b ≤ H(v) ≤ e

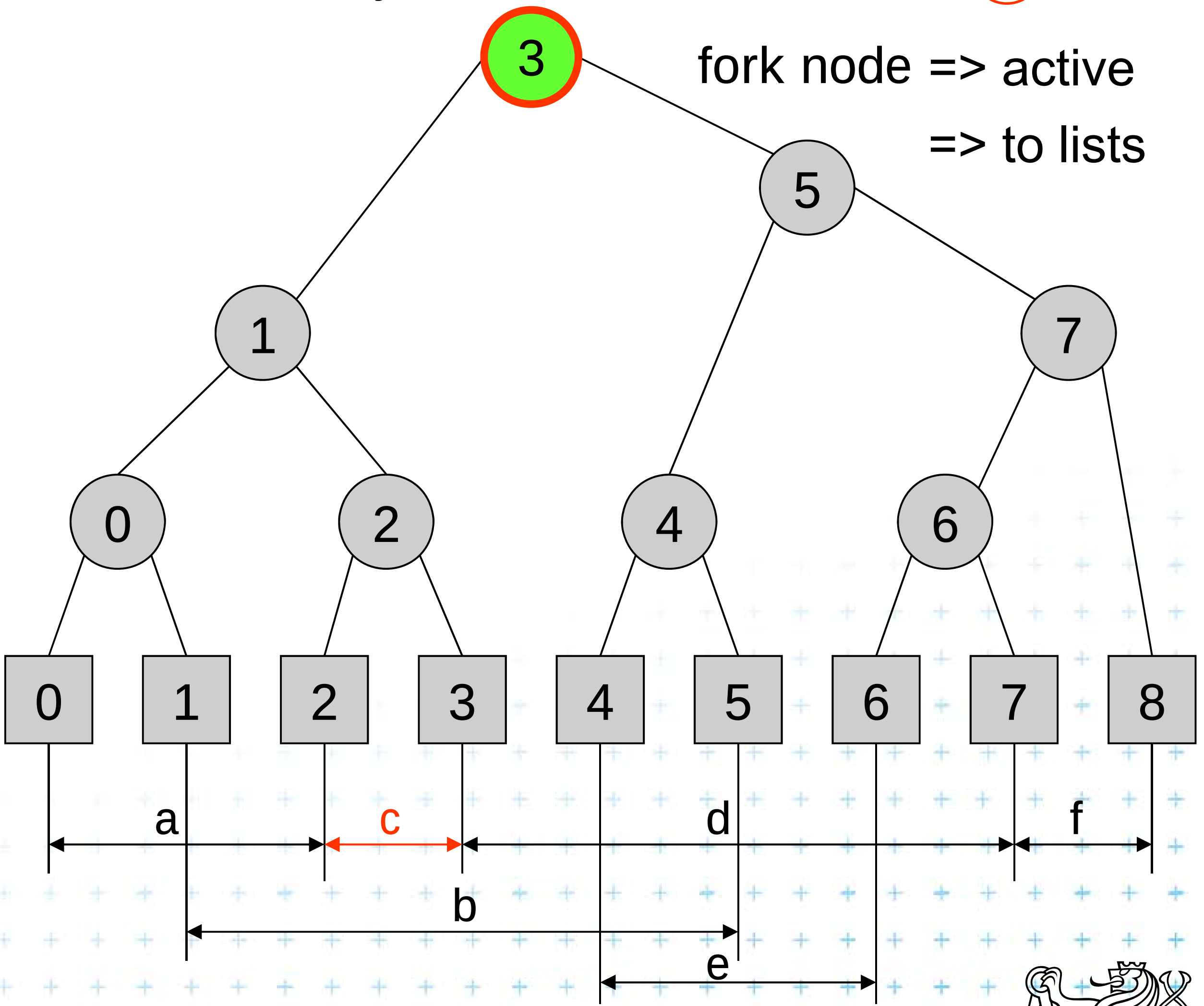
Insert the new interval to secondary lists

? 2 ≤ 3 ≤ 3 ?

fork node => active

=> to lists





🟩 Active rectangle

○ Current node

● Active node

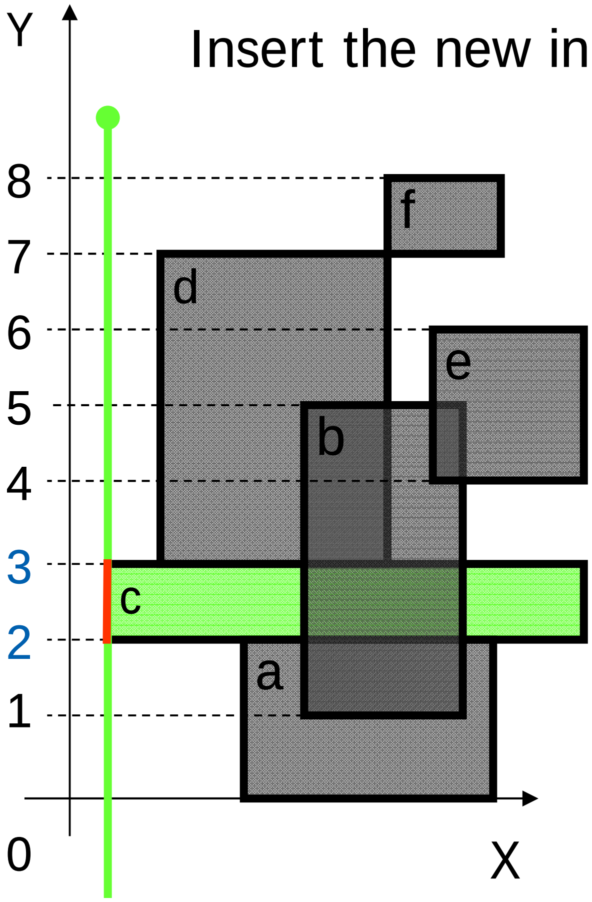
[Drtina]

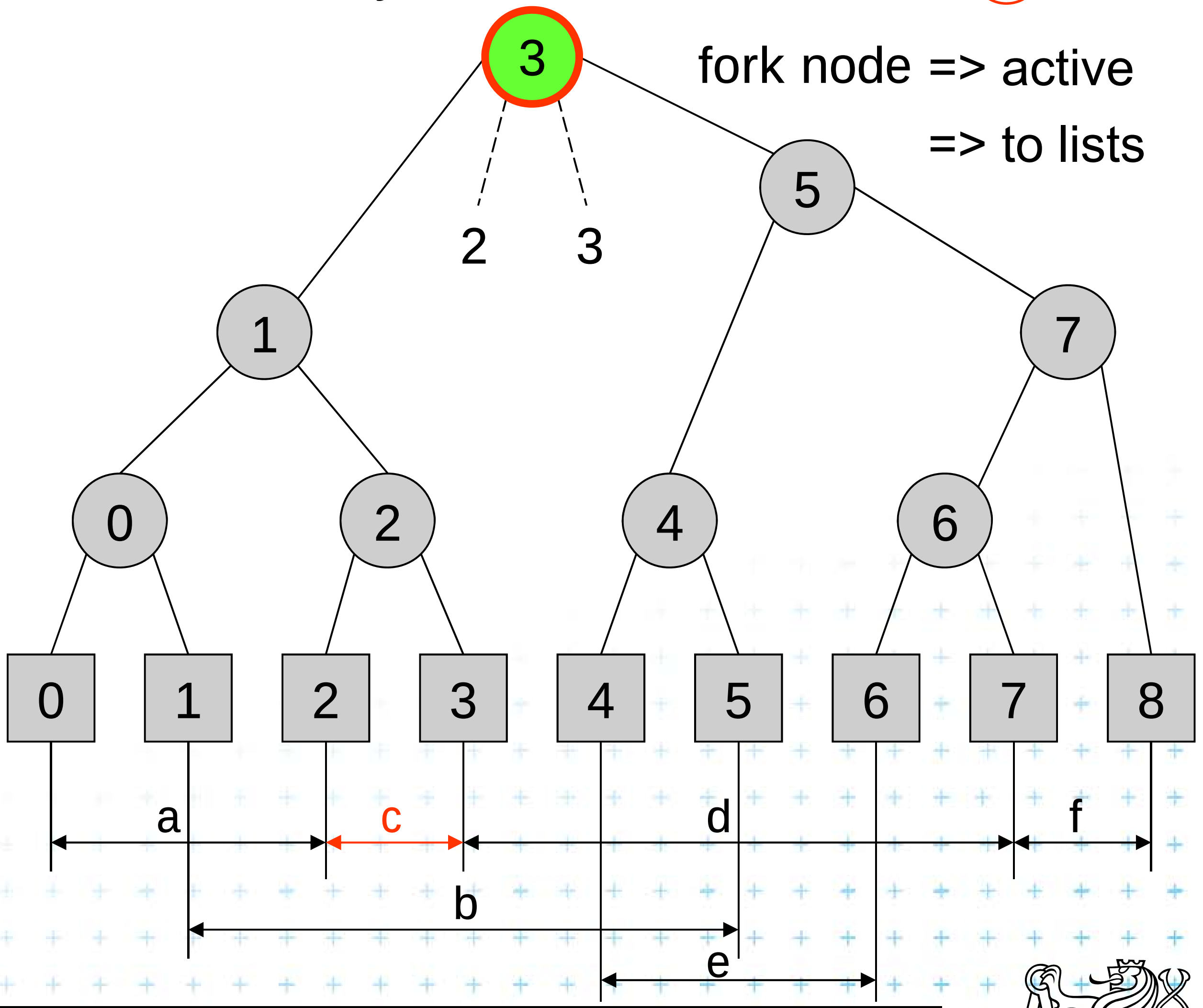
Insert [2,3] – empty => b) Insert Interval

$b \le H(v) \le e$

Insert the new interval to secondary lists

? 2 ≤ 3 ≤ 3 ?





fork node => active

=> to lists

🟩 Active rectangle

○ Current node

● Active node

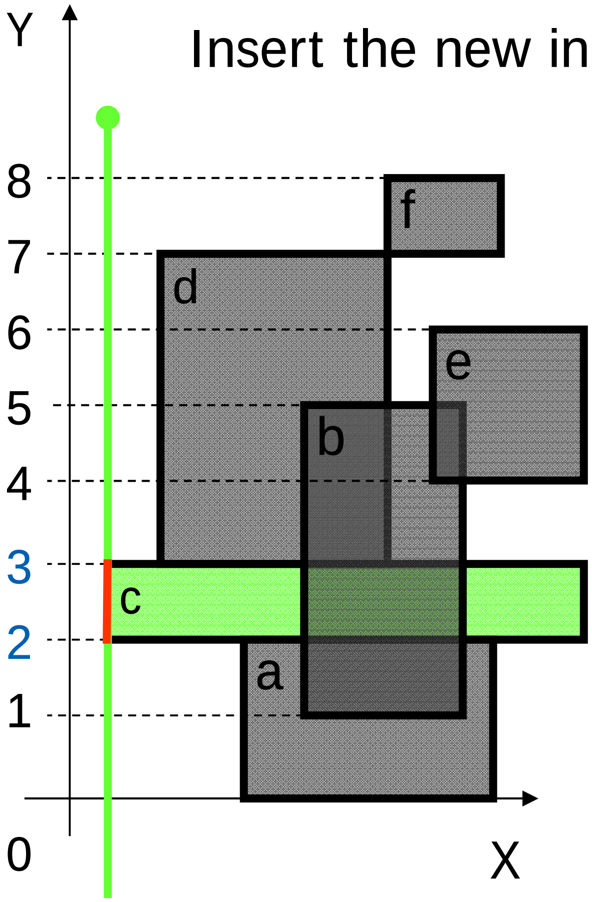
[Drtina]

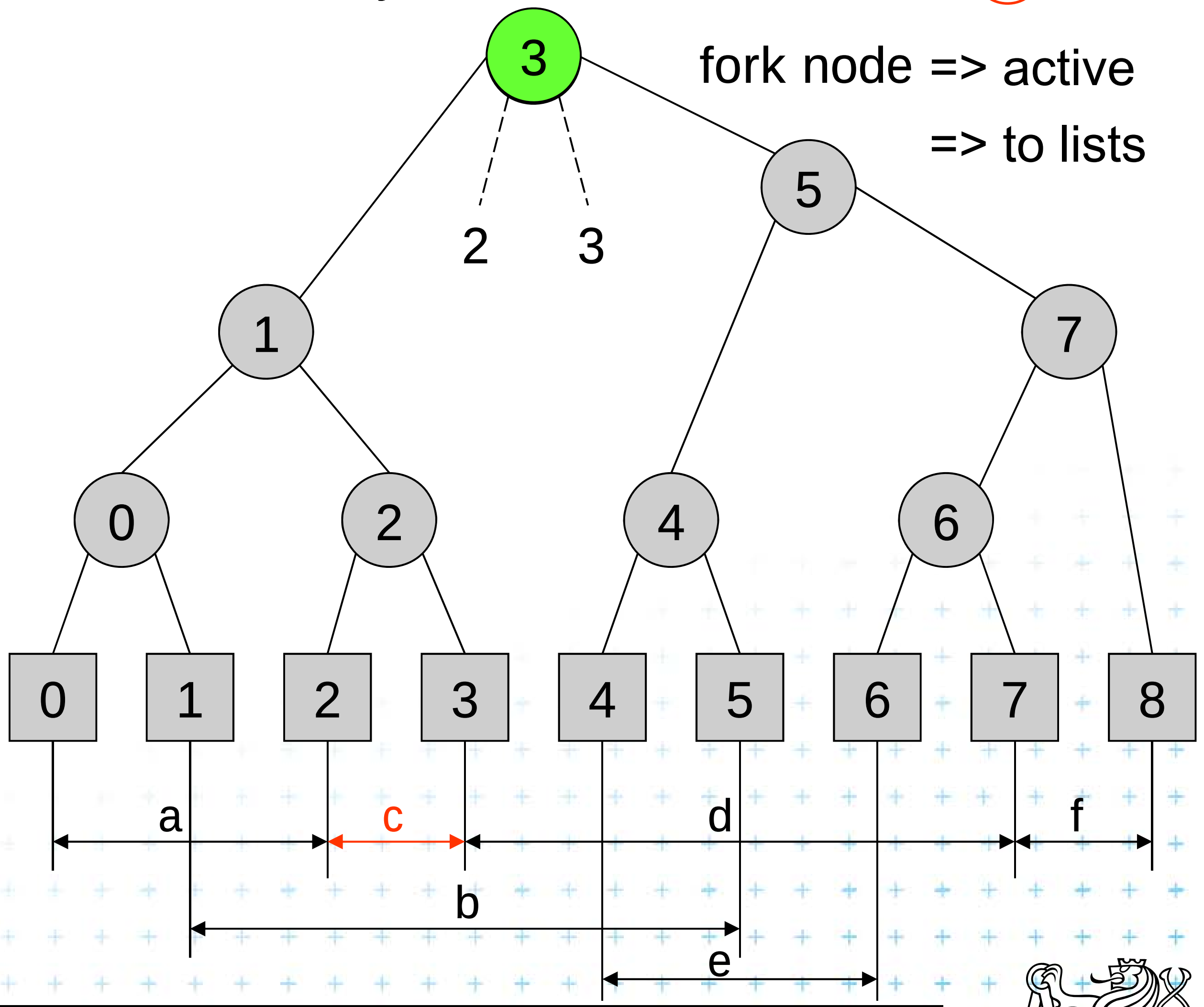
Insert [2,3] – empty => b) Insert Interval

b ≤ H(v) ≤ e

Insert the new interval to secondary lists

? 2 ≤ 3 ≤ 3 ?





fork node => active

=> to lists

🟩 Active rectangle

○ Current node

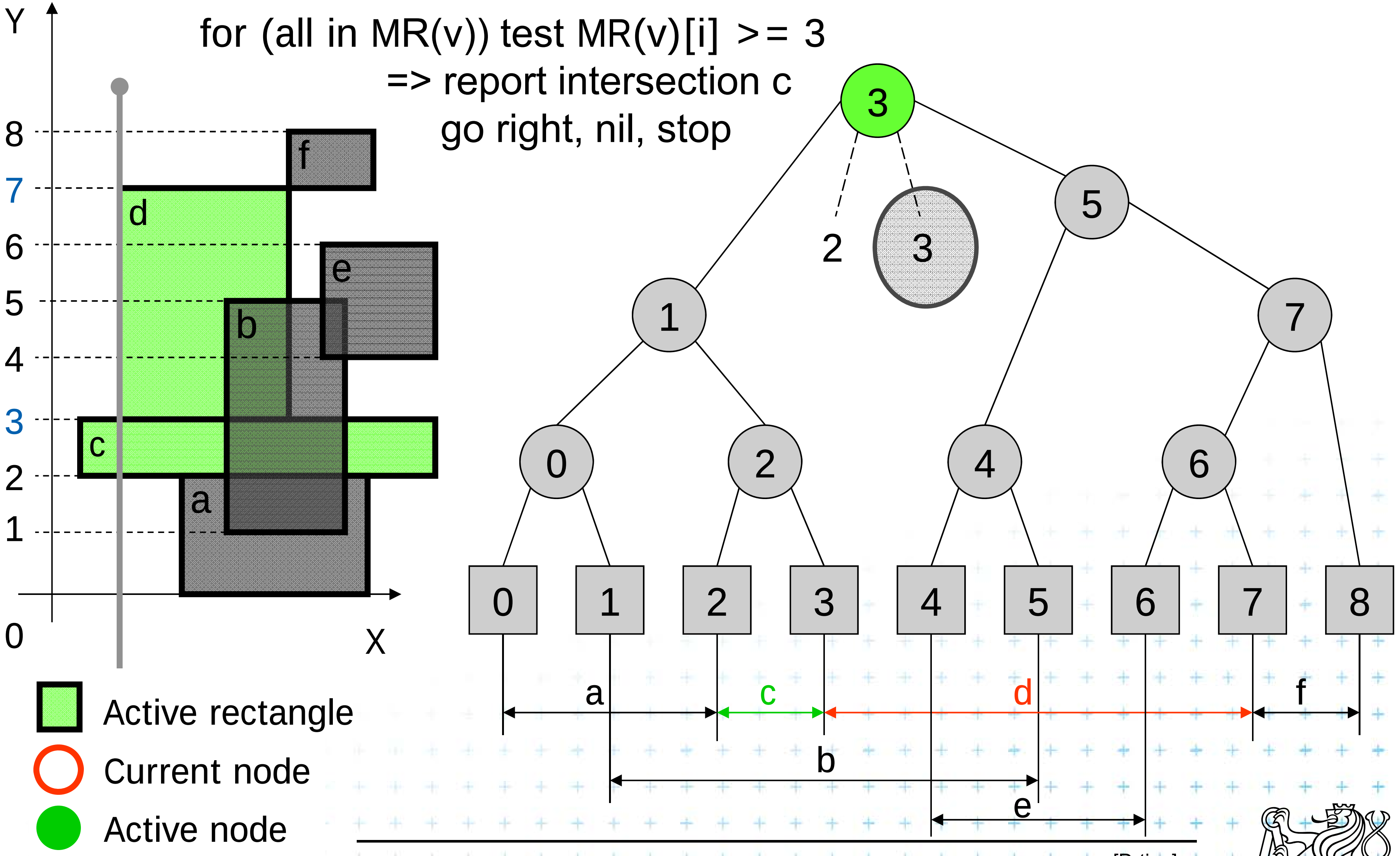
● Active node

[Drtina]

Insert [3,7] a) Query Interval

H(v) ≤ b < e

for (all in MR(v)) test MR(v)[i] >= 3
=> report intersection c
go right, nil, stop



🟩 Active rectangle

○ Current node

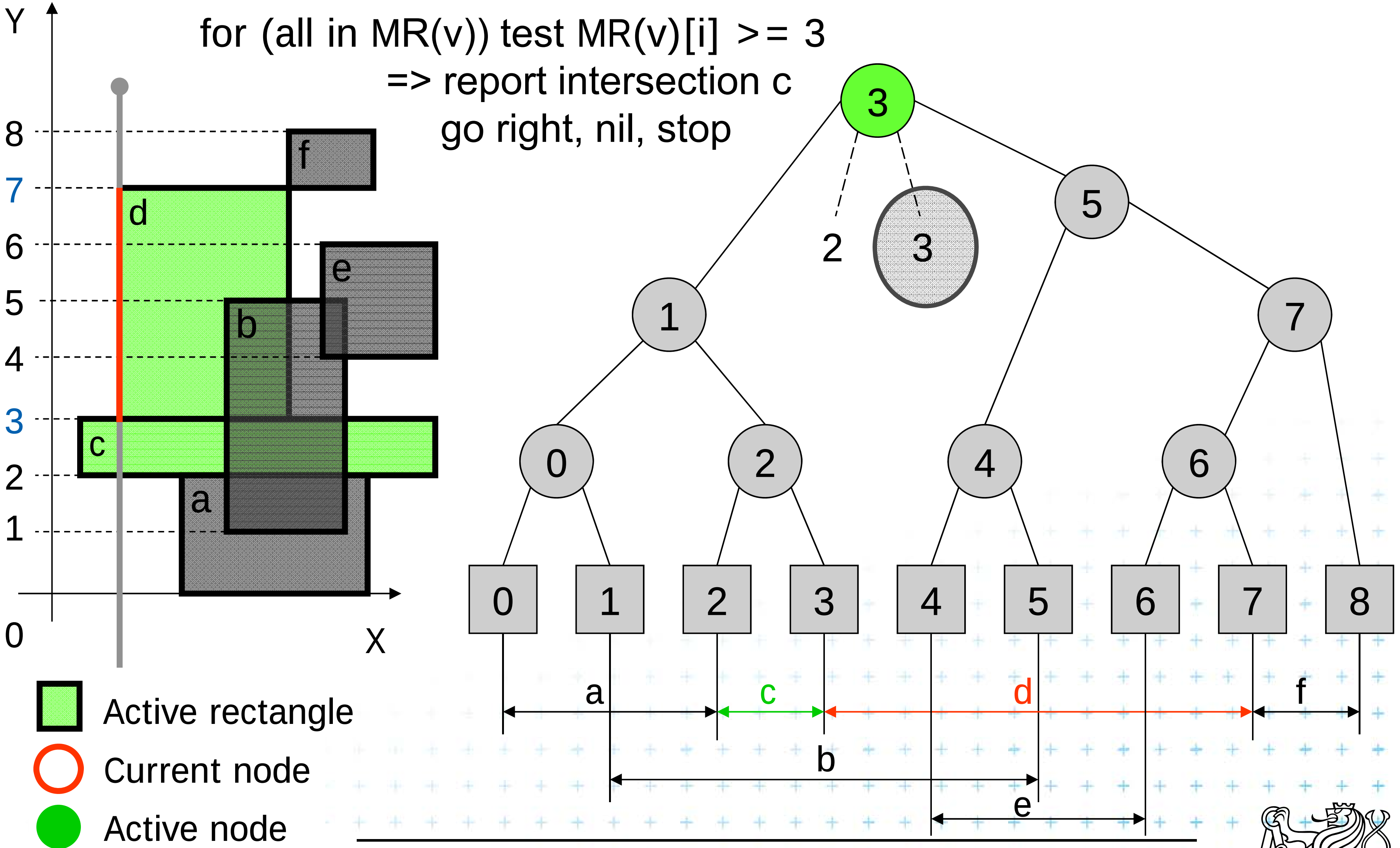
● Active node

[Drtina]

Insert [3,7] a) Query Interval

H(v) ≤ b < e

for (all in MR(v)) test MR(v)[i] >= 3
=> report intersection c
go right, nil, stop



🟩 Active rectangle

○ Current node

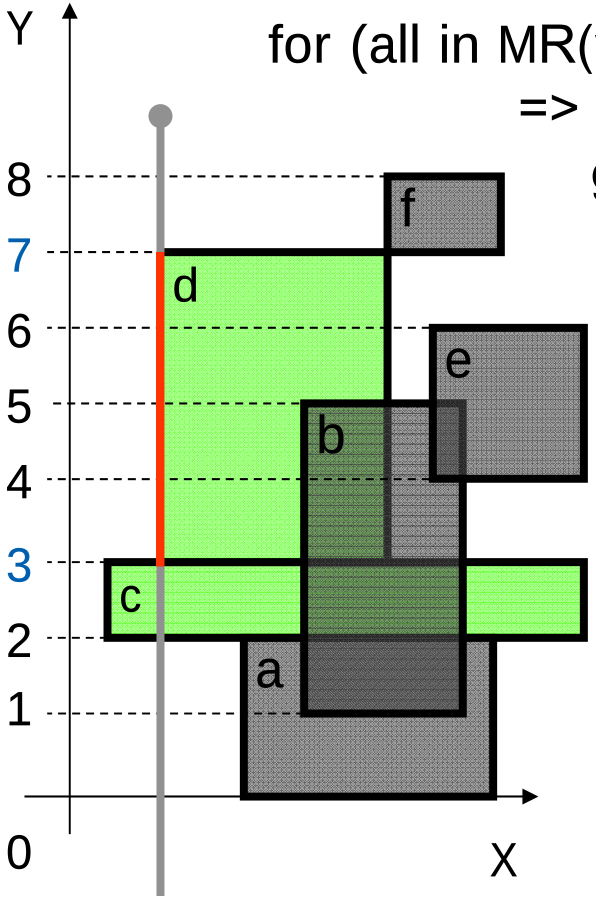
● Active node

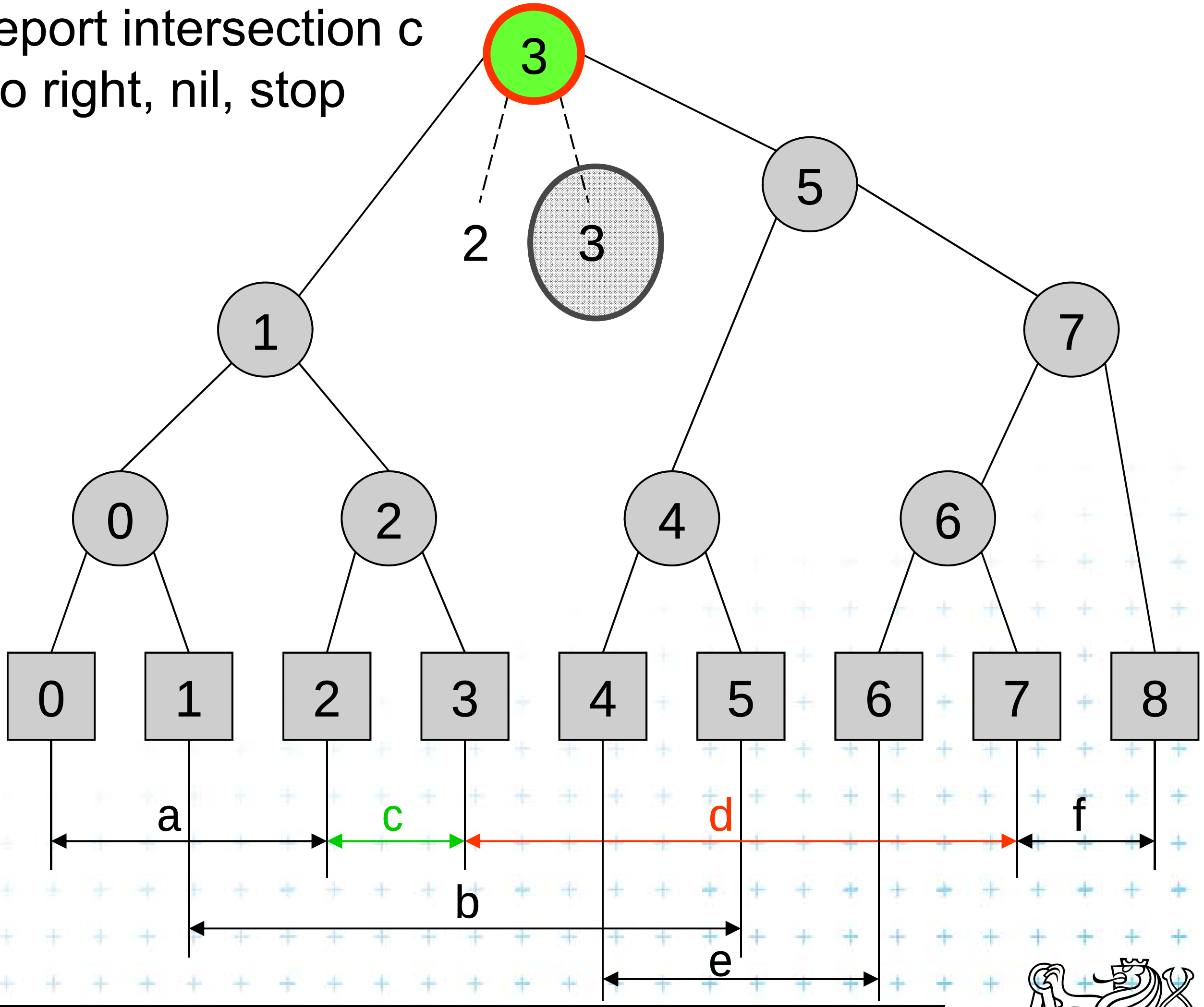
[Drtina]

Insert [3,7] a) Query Interval

H(v) ≤ b < e

for (all in MR(v)) test MR(v)[i] >= 3
=> report intersection c
go right, nil, stop





🟩 Active rectangle

○ Current node

● Active node

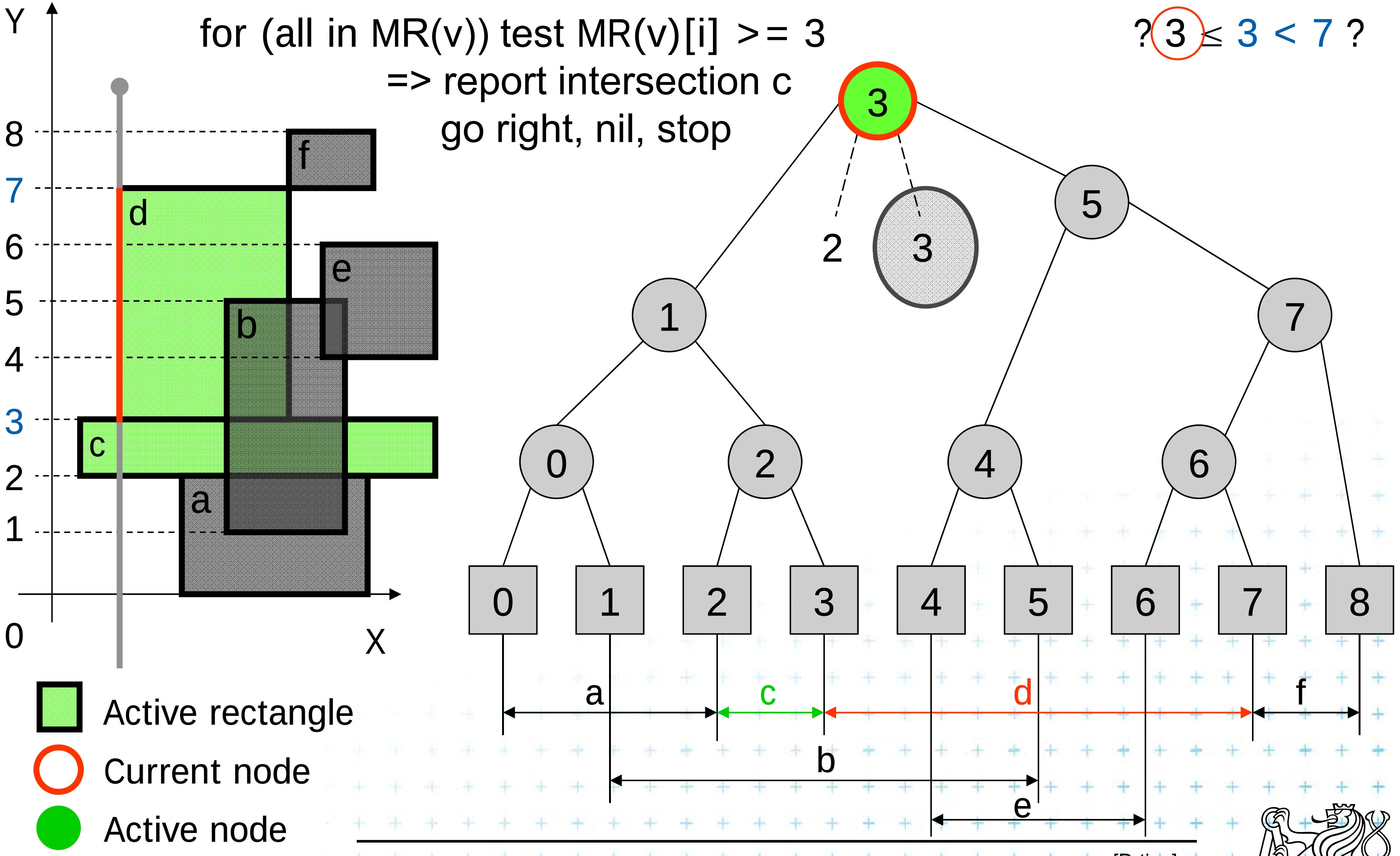
[Drtina]

Insert [3,7] a) Query Interval

H(v) ≤ b < e

for (all in MR(v)) test MR(v)[i] >= 3
=> report intersection c
go right, nil, stop

? 3 ≤ 3 < 7 ?



🟩 Active rectangle

○ Current node

● Active node

[Drtina]

Insert [3,7] b) Insert Interval

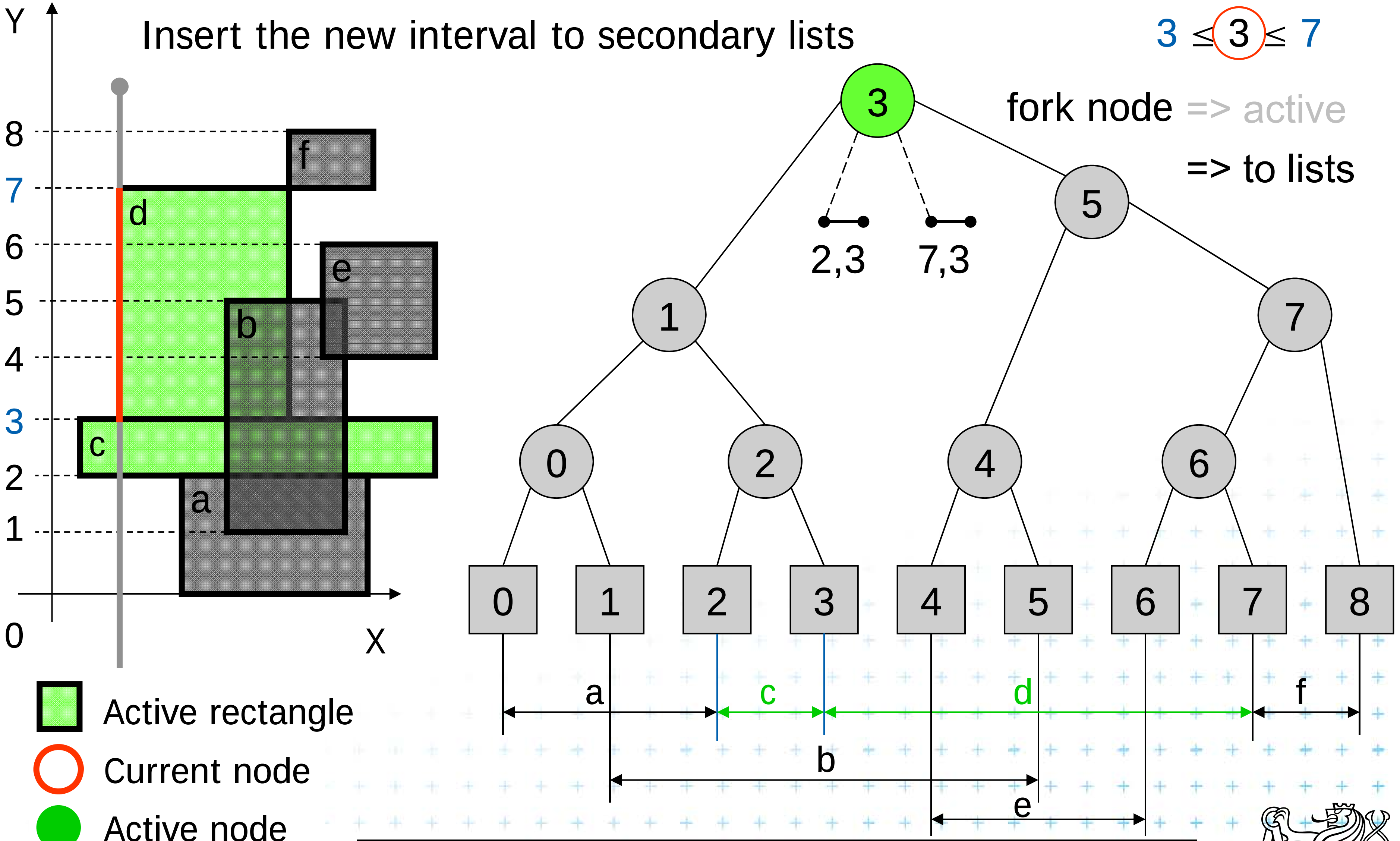
b ≤ H(v) ≤ e

Insert the new interval to secondary lists

3 ≤ 3 ≤ 7

fork node => active

=> to lists



🟩 Active rectangle

○ Current node

● Active node

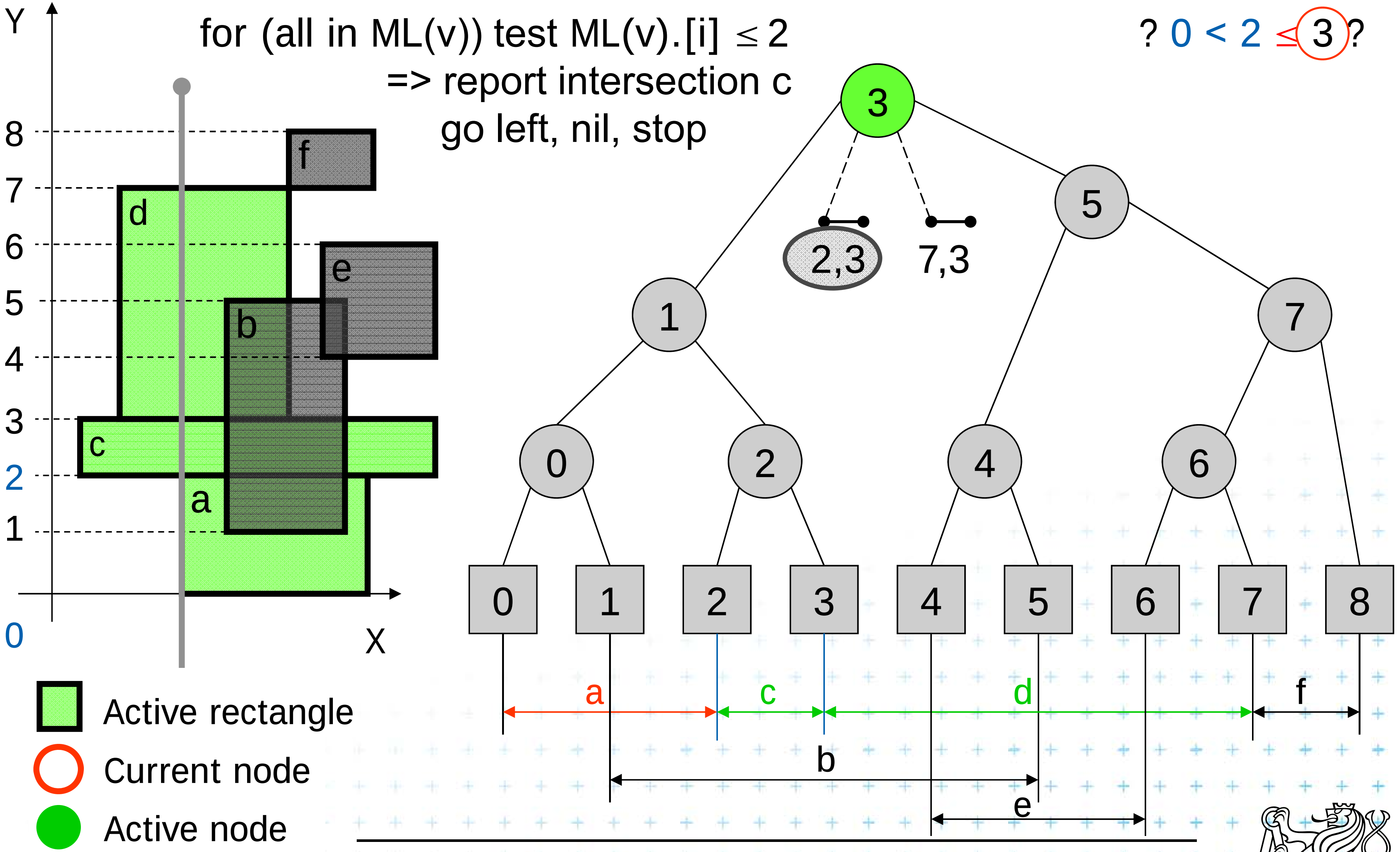
[Drtina]

Insert [0,2] a) Query Interval

b < e ≤ H(v)

for (all in ML(v)) test ML(v).[i] ≤ 2
=> report intersection c
go left, nil, stop

? 0 < 2 ≤ 3 ?



Active rectangle

Current node

Active node

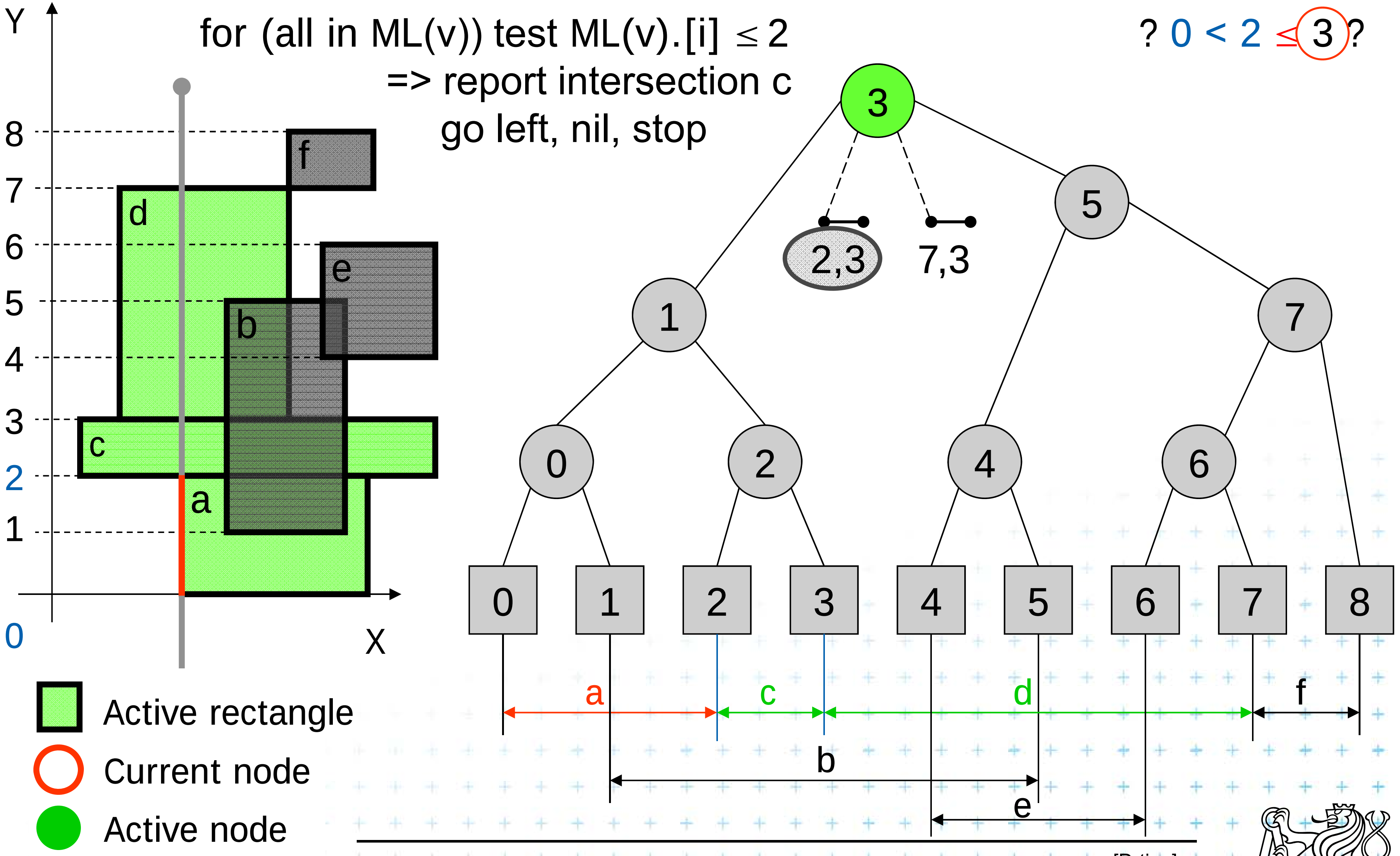
[Drtina]

Insert [0,2] a) Query Interval

b < e ≤ H(v)

for (all in ML(v)) test ML(v).[i] ≤ 2
=> report intersection c
go left, nil, stop

? 0 < 2 ≤ 3 ?



🟩 Active rectangle

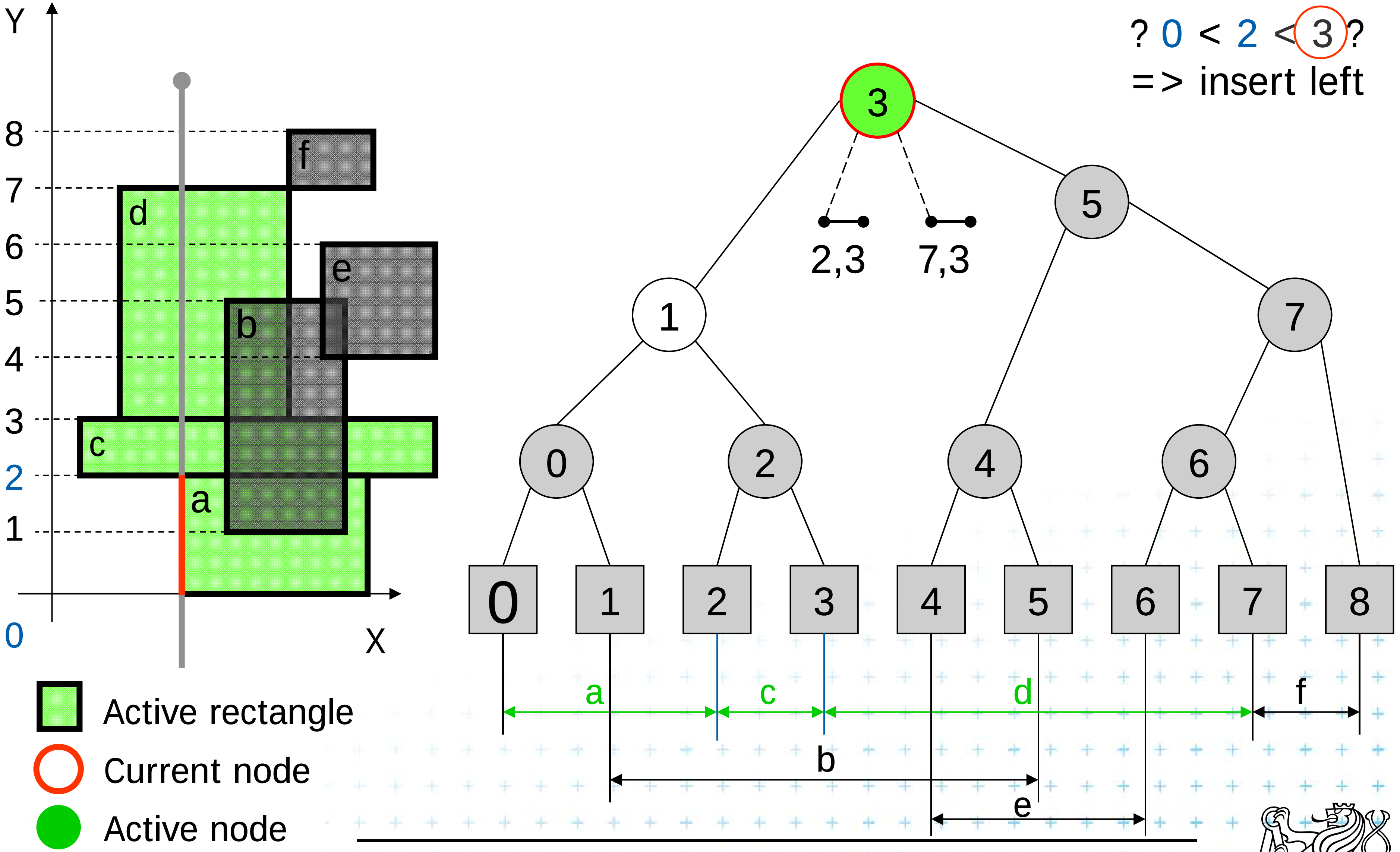
○ Current node

● Active node

[Drtina]

Insert [0,2] b) Insert Interval 1/2

b < e <H(v)



[Drtina]

Insert [0,2] b) Insert Interval 2/2

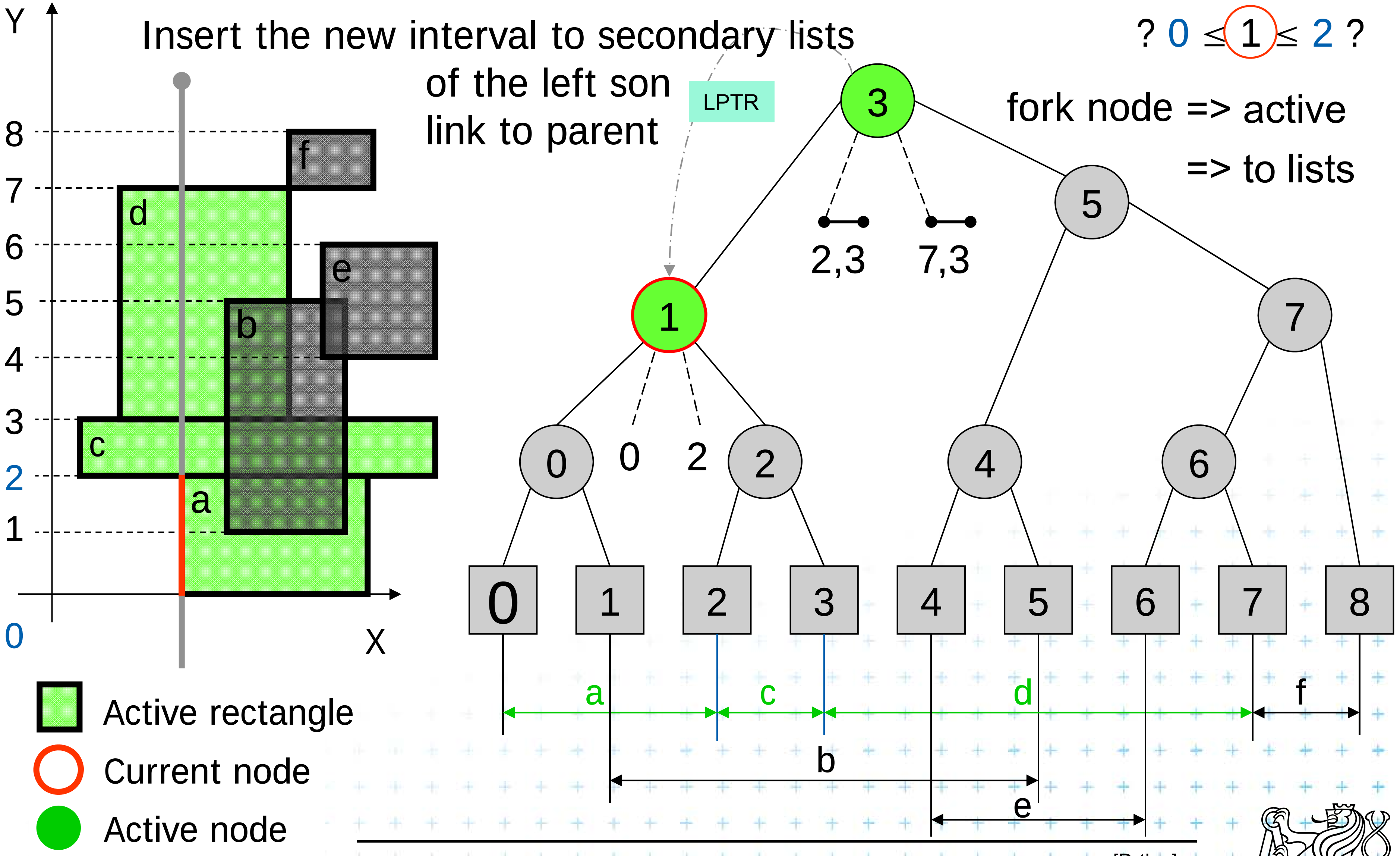
b ≤ H(v) ≤ e

Insert the new interval to secondary lists of the left son link to parent

? 0 ≤ 1 ≤ 2 ?

fork node => active

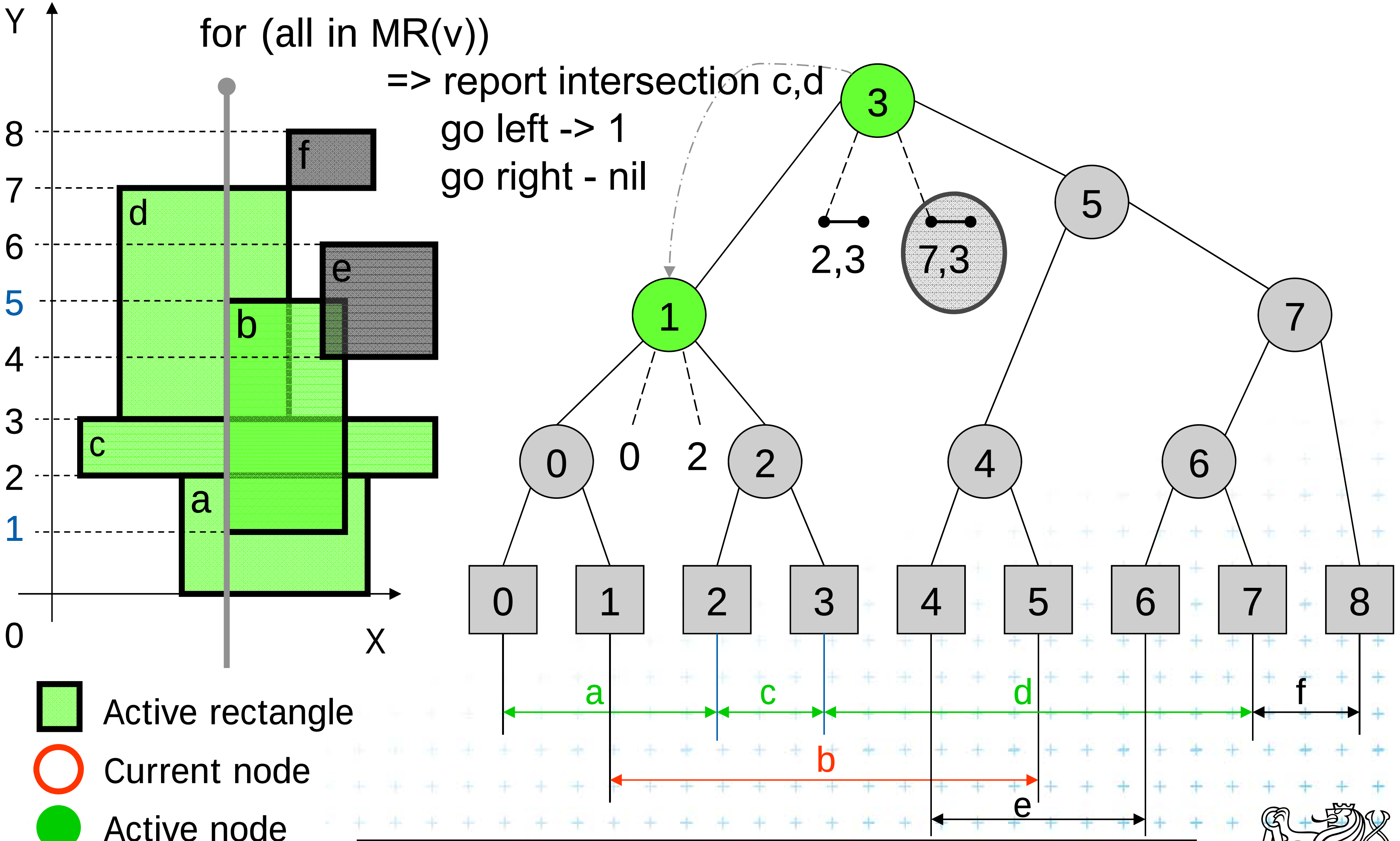
=> to lists



[Drtina]

Insert [1,5] a) Query Interval 1/2

b < H(v) < e



for (all in MR(v))

=> report intersection c,d

go left -> 1

go right - nil

🟩 Active rectangle

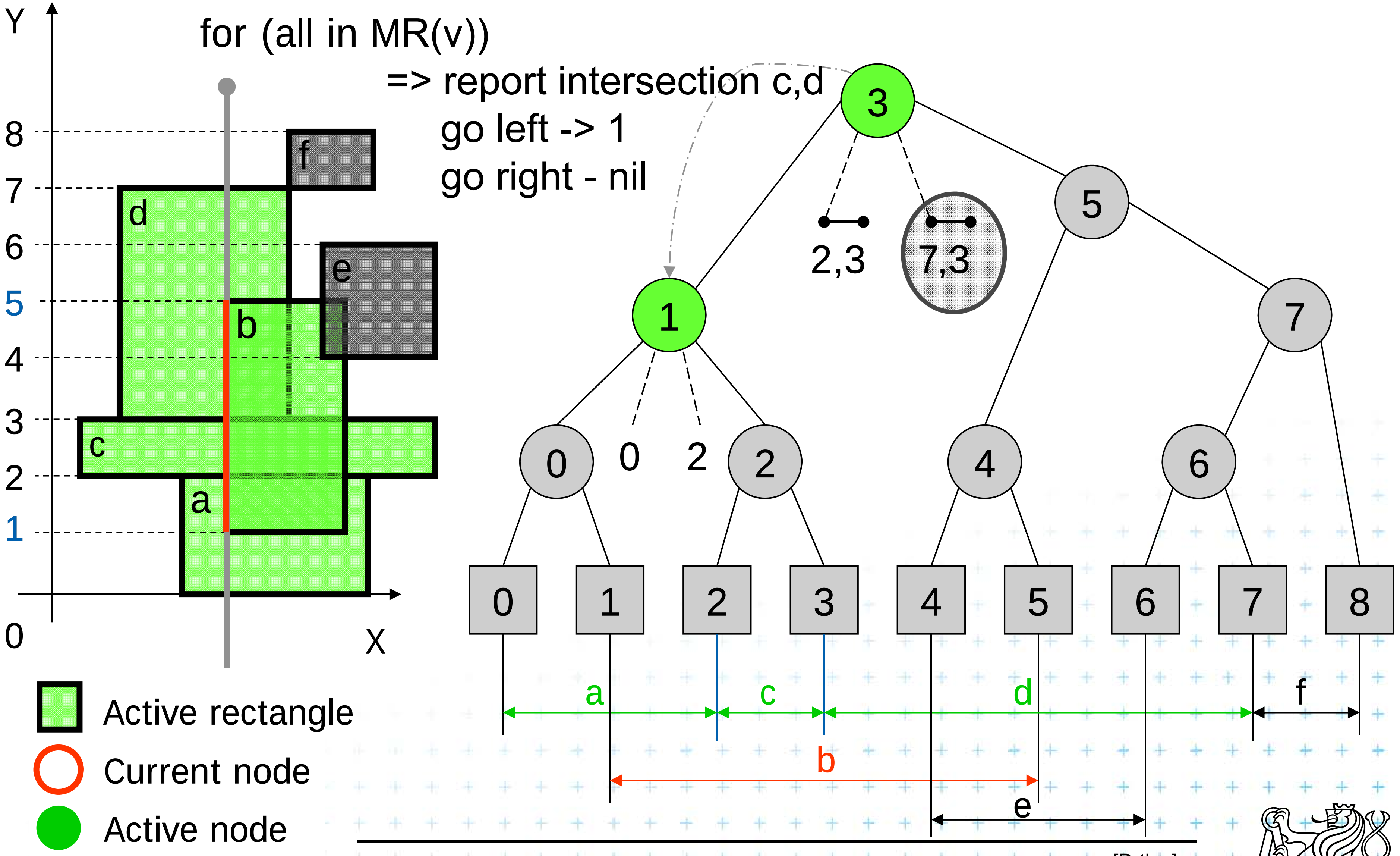
○ Current node

● Active node

[Drtina]

Insert [1,5] a) Query Interval 1/2

b < H(v) < e



🟩 Active rectangle

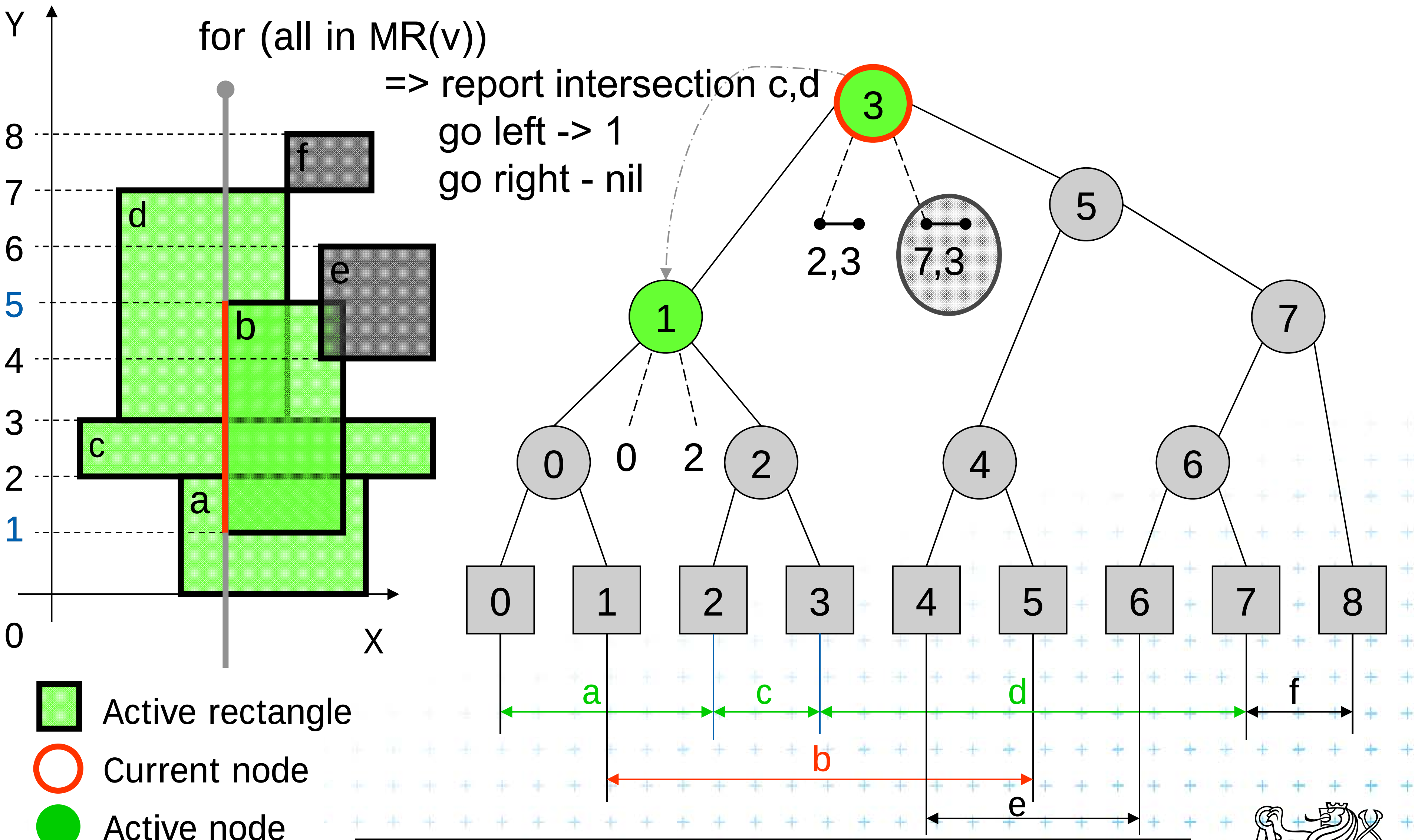
○ Current node

● Active node

[Drtina]

Insert [1,5] a) Query Interval 1/2

b < H(v) < e



for (all in MR(v))

=> report intersection c,d

go left -> 1

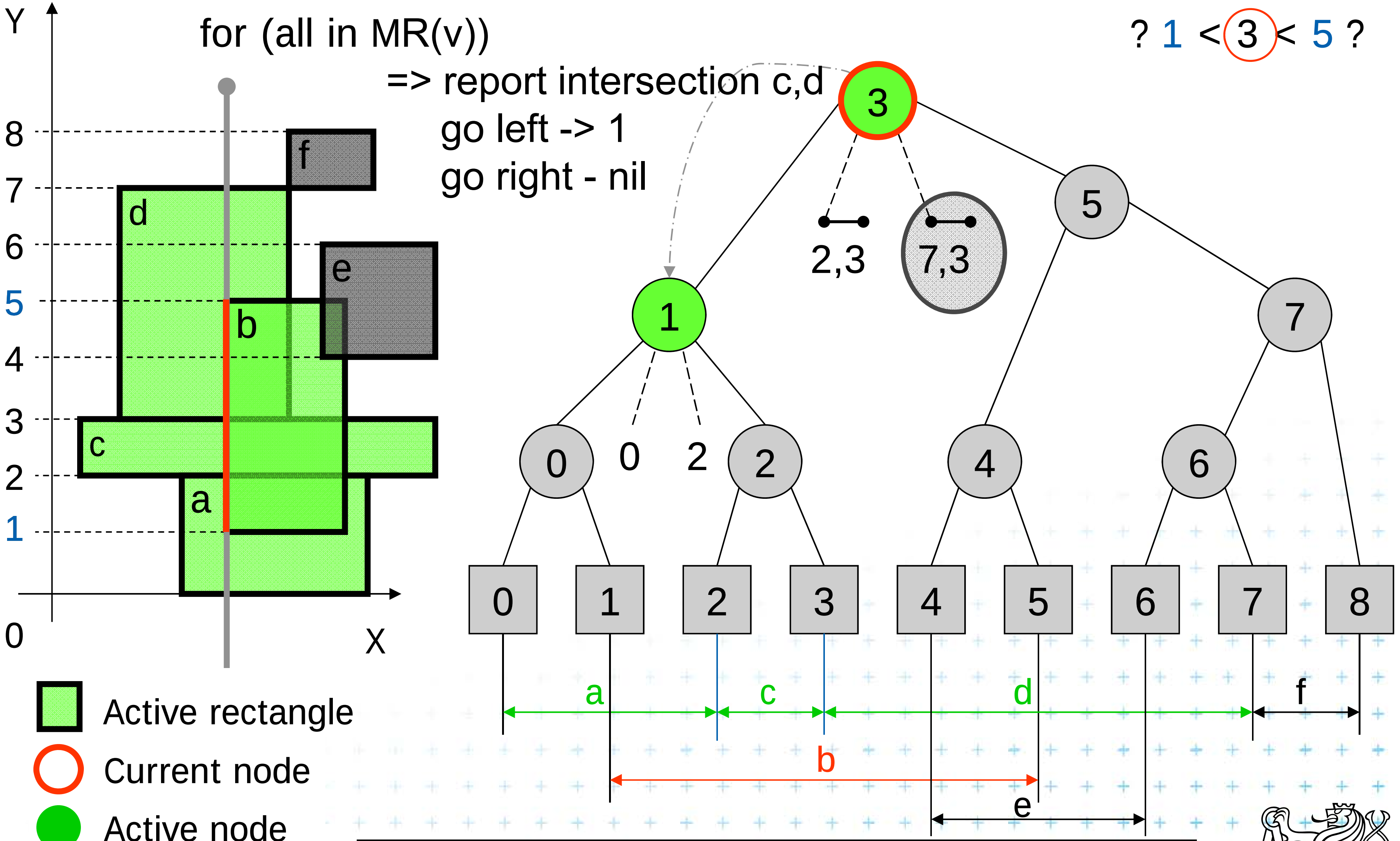
go right - nil

- 🟩 Active rectangle
- ○ Current node
- ● Active node

[Drtina]

Insert [1,5] a) Query Interval 1/2

b < H(v) < e



for (all in MR(v))
=> report intersection c,d
go left -> 1
go right - nil

? 1 < 3 < 5 ?

🟩 Active rectangle

○ Current node

● Active node

[Drtina]

Insert [1,5] a) Query Interval 1/2

b < H(v) < e

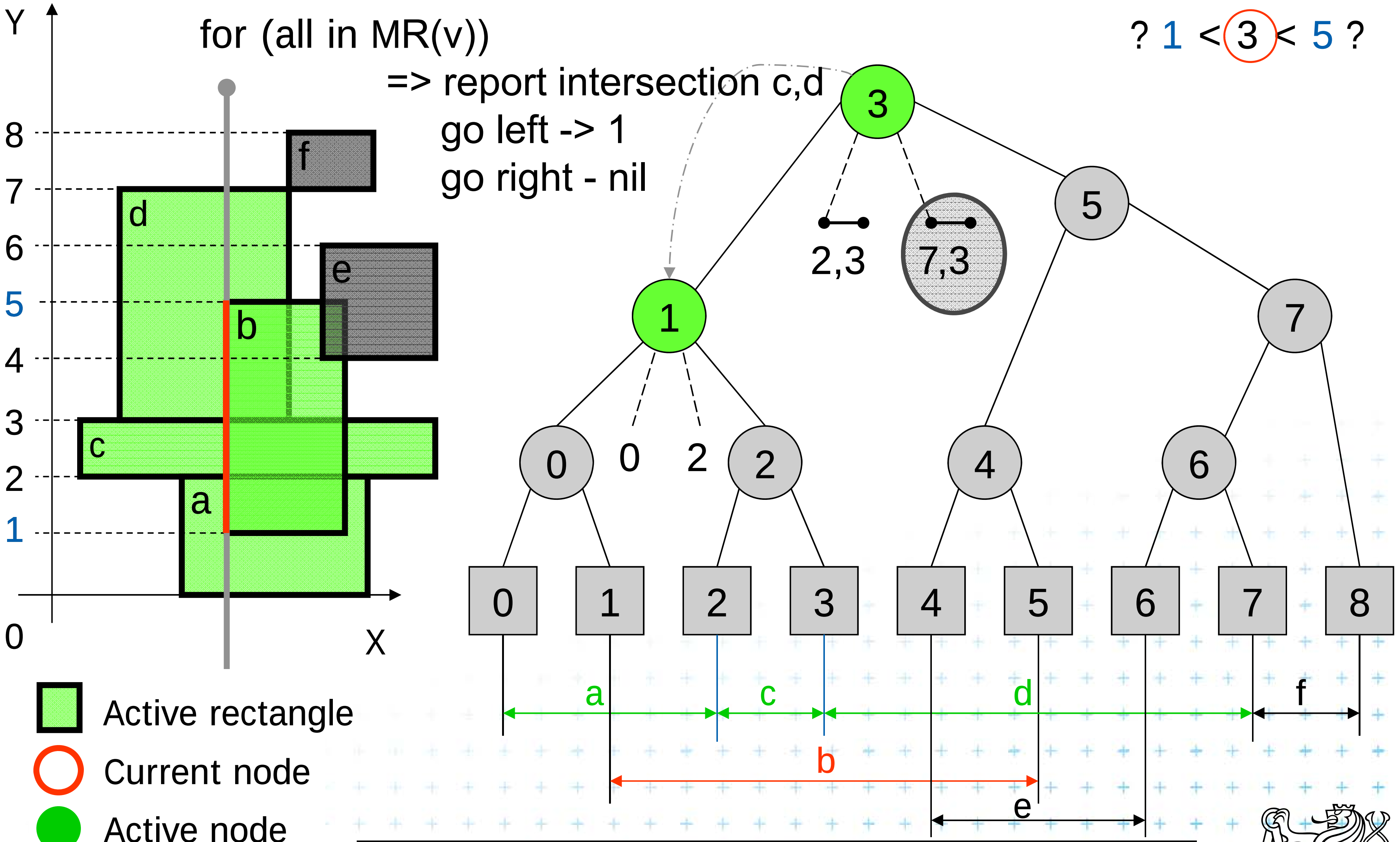
for (all in MR(v))

=> report intersection c,d

go left -> 1

go right - nil

? 1 < 3 < 5 ?



🟩 Active rectangle

○ Current node

● Active node

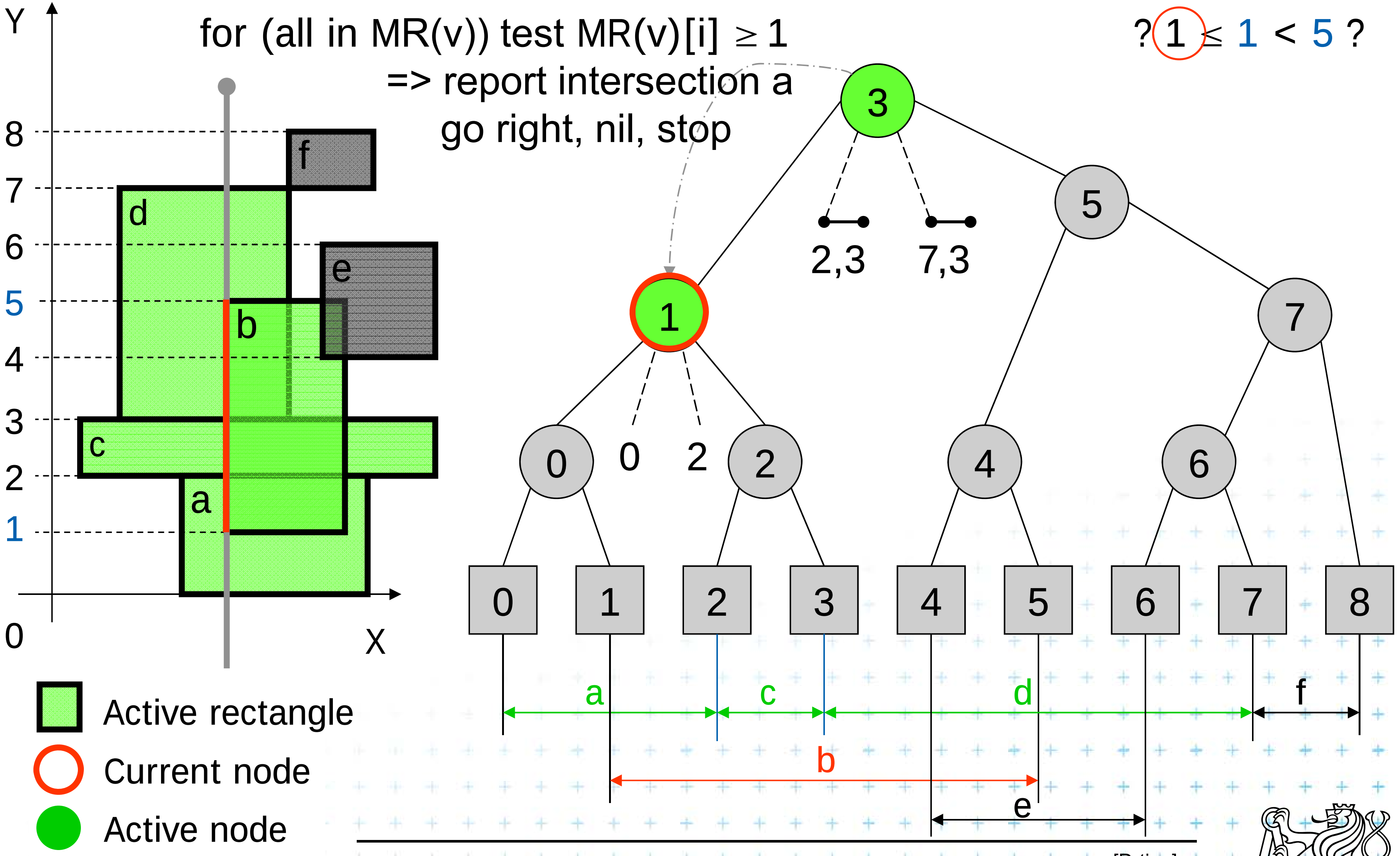
[Drtina]

Insert [1,5] a) Query Interval 2/2

H(v) ≤ b < e

? 1 ≤ 1 < 5 ?

for (all in MR(v)) test MR(v)[i] ≥ 1
=> report intersection a
go right, nil, stop



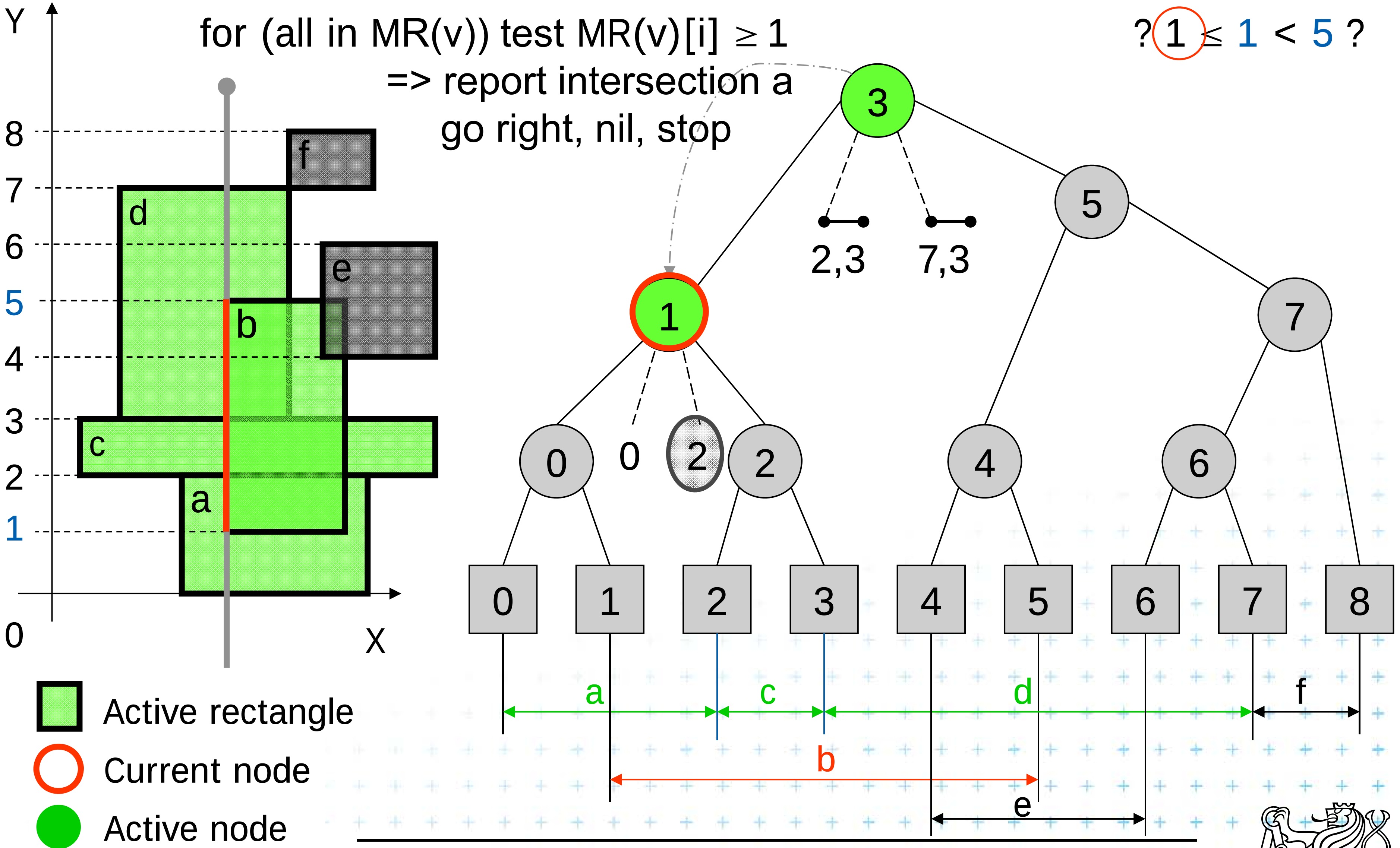
[Drtina]

Insert [1,5] a) Query Interval 2/2

H(v) ≤ b < e

for (all in MR(v)) test MR(v)[i] ≥ 1
=> report intersection a
go right, nil, stop

? 1 ≤ 1 < 5 ?



■ Active rectangle

○ Current node

● Active node

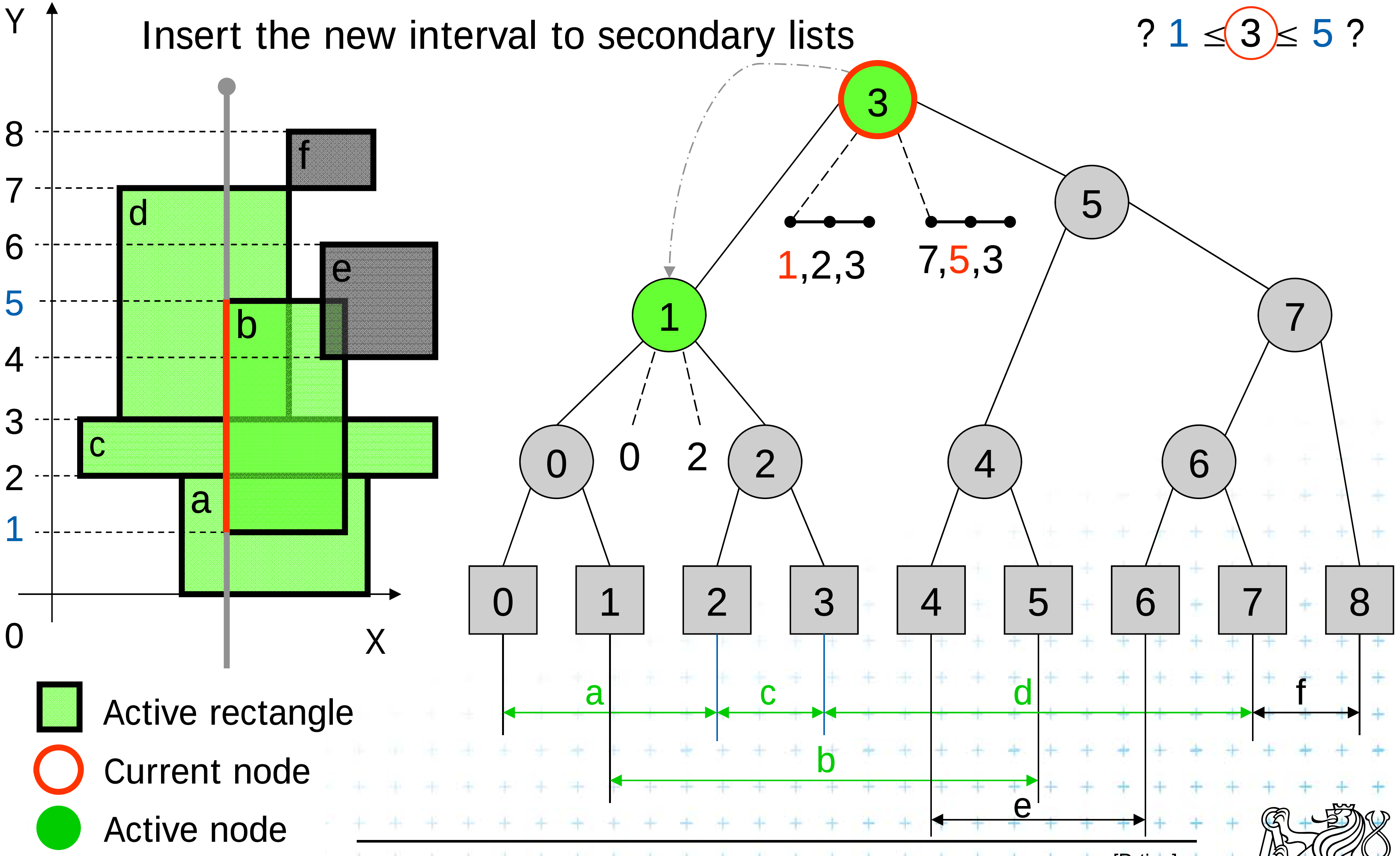
[Drtina]

Insert [1,5] b) Insert Interval

b ≤ H(v) ≤ e

Insert the new interval to secondary lists

? 1 ≤ 3 ≤ 5 ?



🟩 Active rectangle

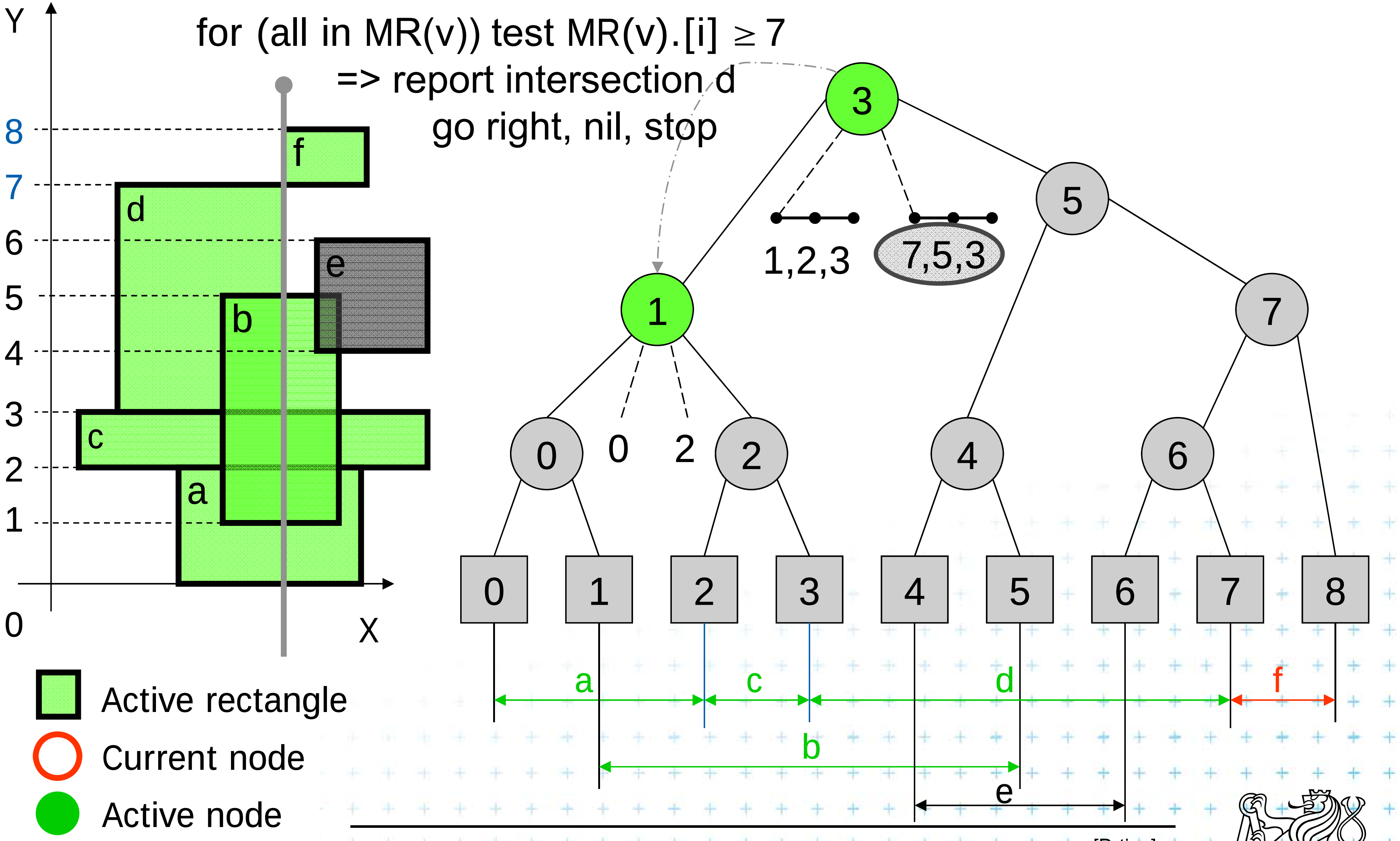
○ Current node

● Active node

[Drtina]

Insert [7,8] a) Query Interval

H(v) ≤ b < e

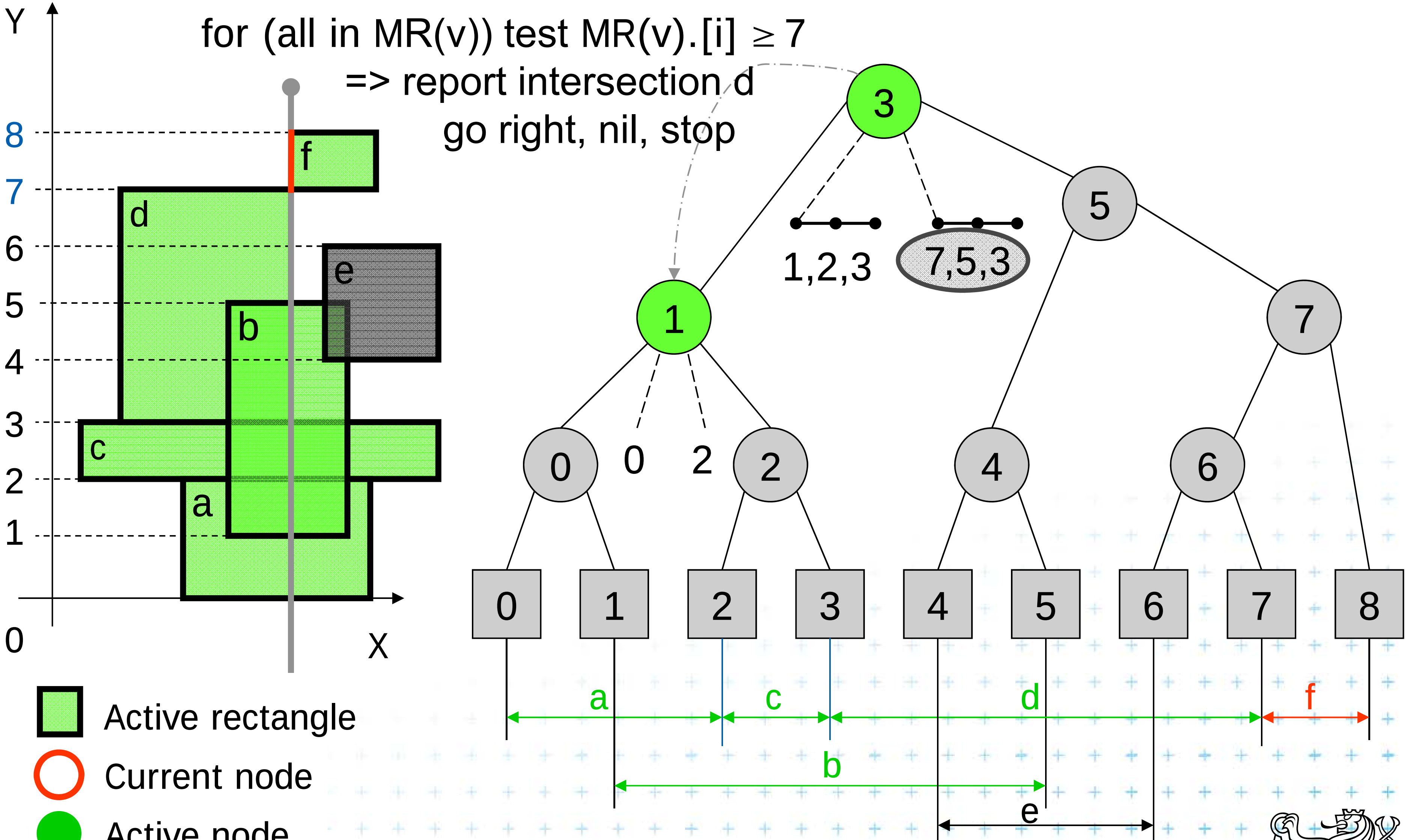


- 🟩 Active rectangle
- ○ Current node
- ● Active node

[Drtina]

Insert [7,8] a) Query Interval

H(v) ≤ b < e



for (all in MR(v)) test MR(v).[i] ≥ 7
=> report intersection d
go right, nil, stop

🟩 Active rectangle

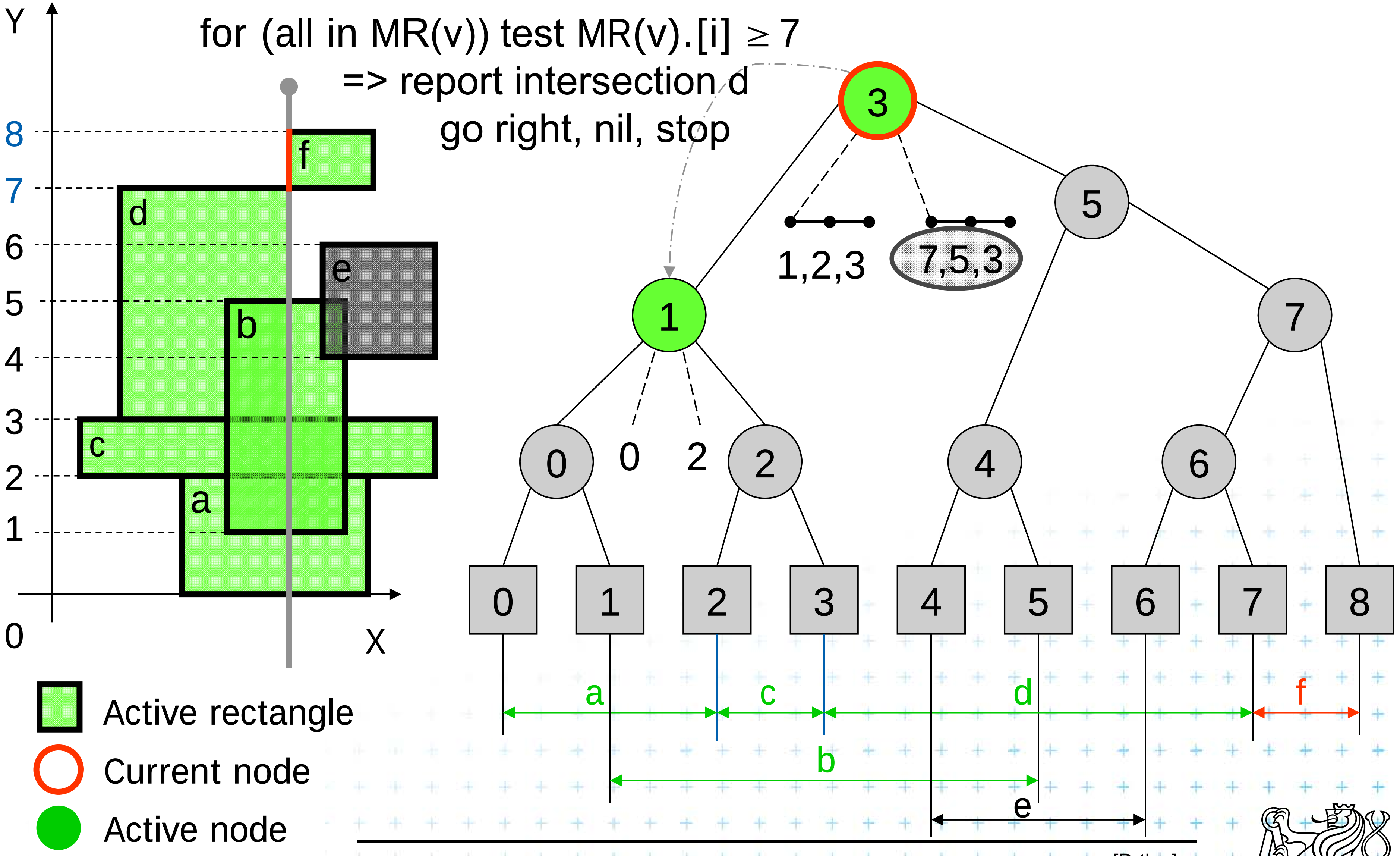
○ Current node

● Active node

[Drtina]

Insert [7,8] a) Query Interval

H(v) ≤ b < e



for (all in MR(v)) test MR(v).[i] ≥ 7
=> report intersection d
go right, nil, stop

🟩 Active rectangle

○ Current node

● Active node

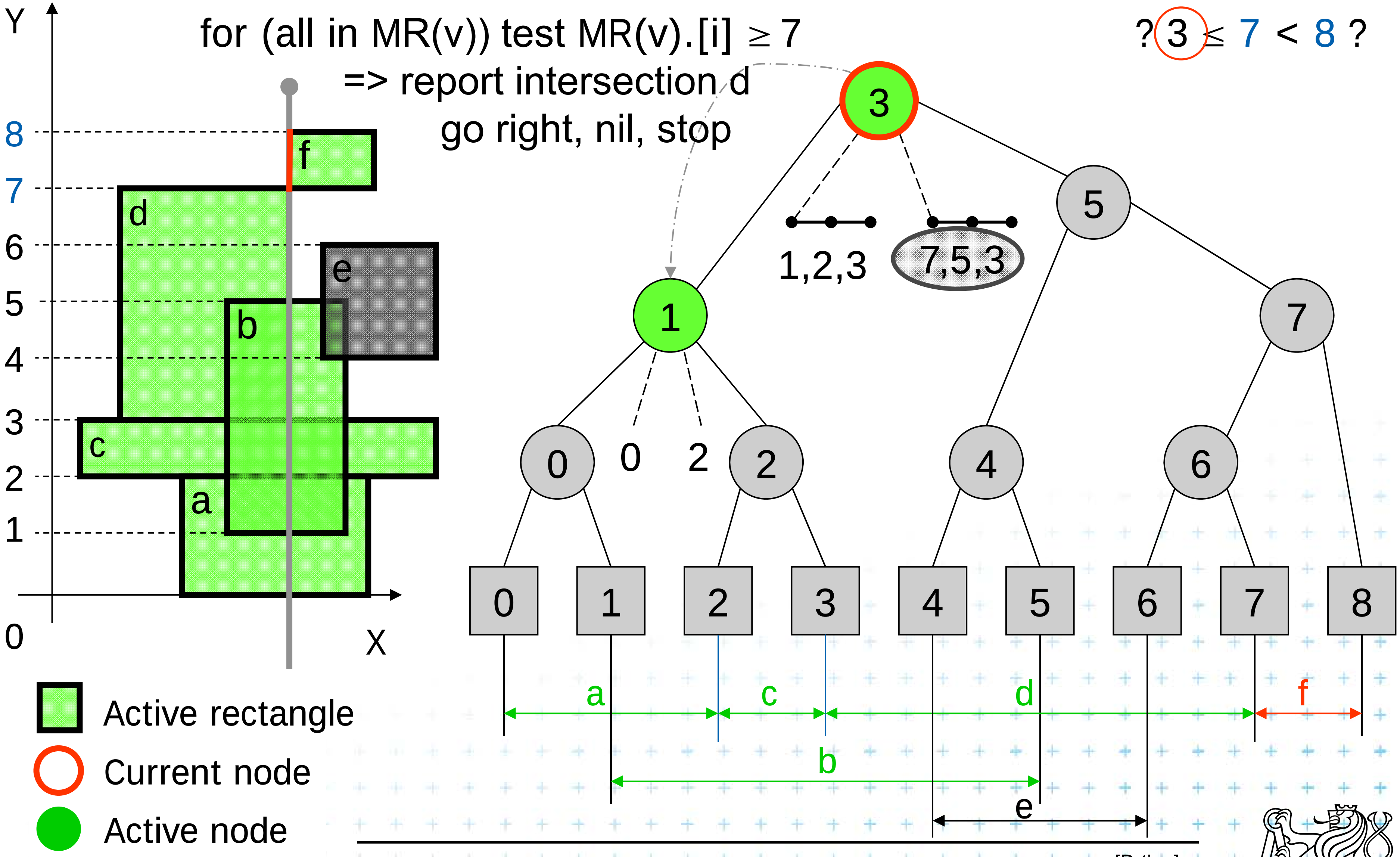
[Drtina]

Insert [7,8] a) Query Interval

H(v) ≤ b < e

for (all in MR(v)) test MR(v).[i] ≥ 7
=> report intersection d
go right, nil, stop

? 3 ≤ 7 < 8 ?



🟩 Active rectangle

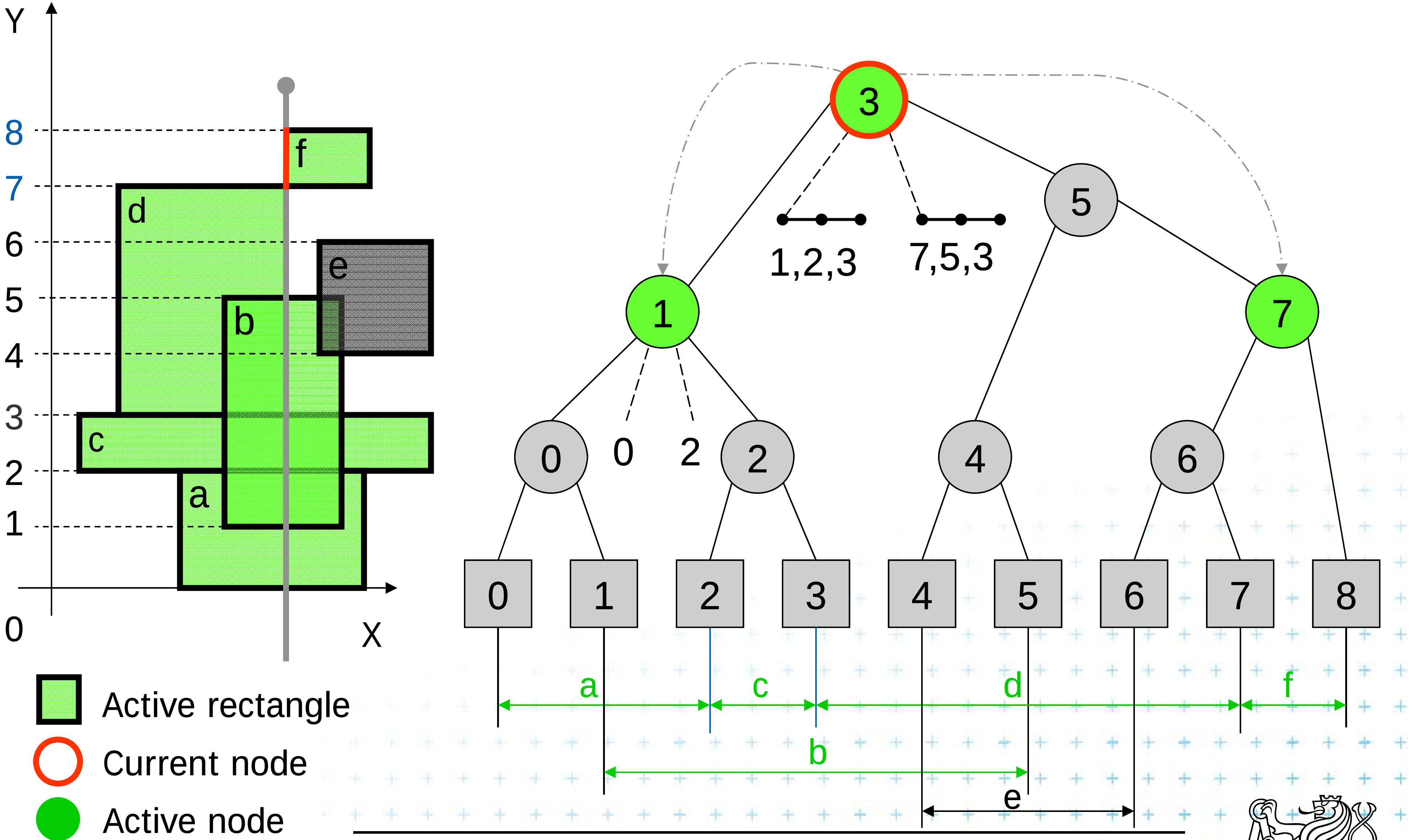
○ Current node

● Active node

[Drtina]

Insert [7,8] b) Insert Interval

$b \leq H(v) \leq e$



🟩 Active rectangle

○ Current node

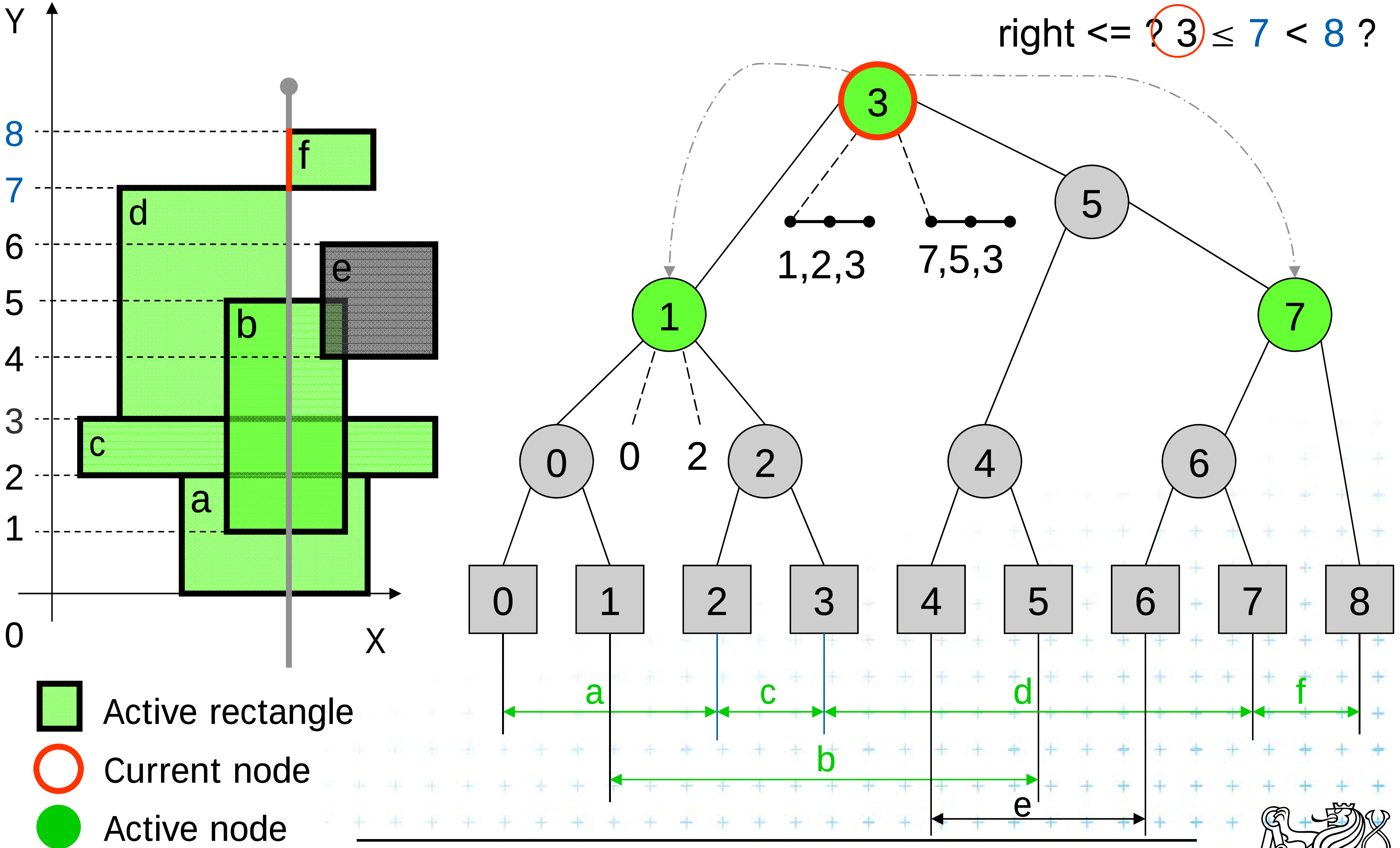
● Active node

[Drtina]

Insert [7,8] b) Insert Interval

$b \leq H(v) \leq e$

right <= ? 3 ≤ 7 < 8 ?



🟩 Active rectangle

○ Current node

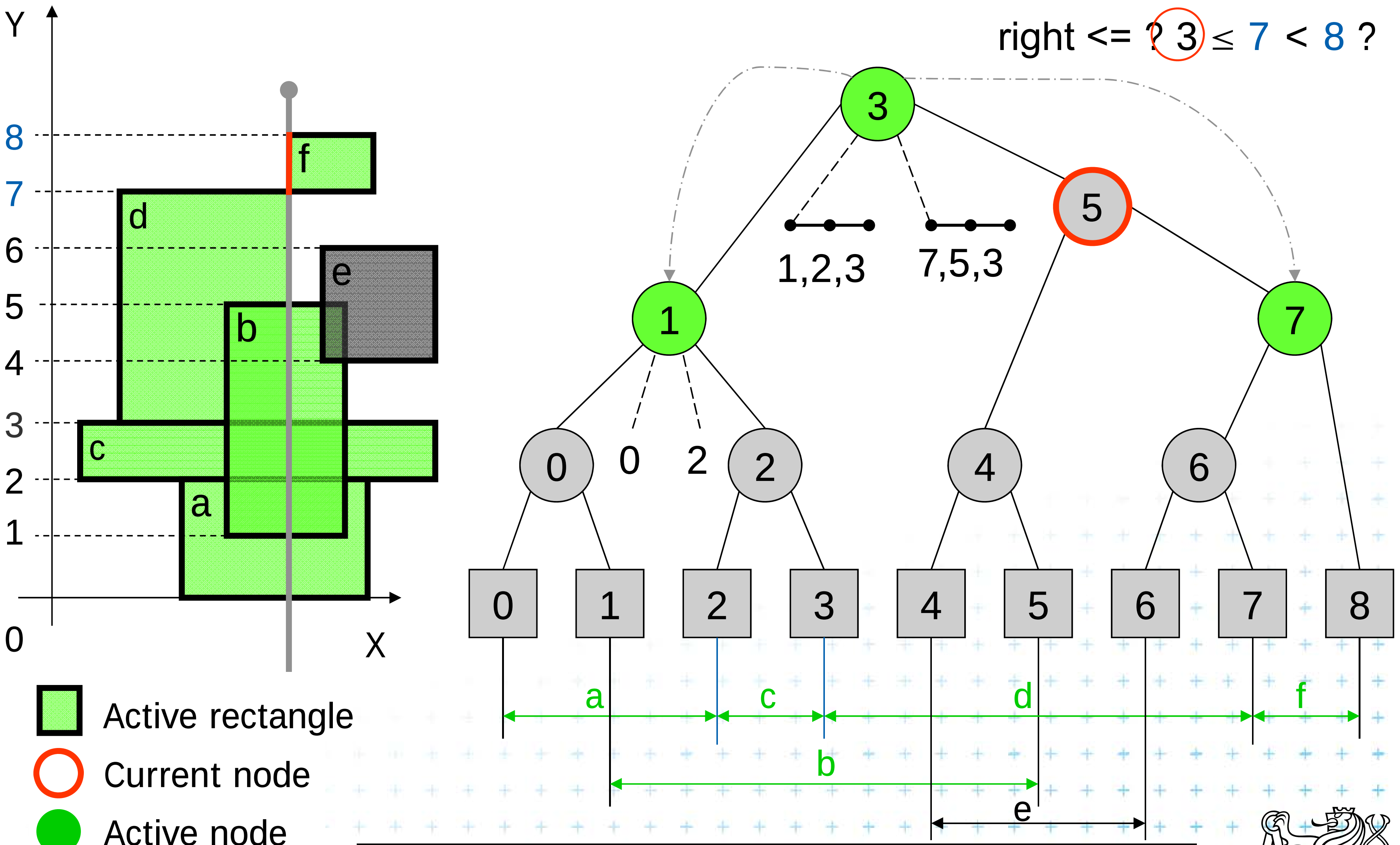
● Active node

[Drtina]

Insert [7,8] b) Insert Interval

b ≤ H(v) ≤ e

right <= ? 3 ≤ 7 < 8 ?



🟩 Active rectangle

○ Current node

● Active node

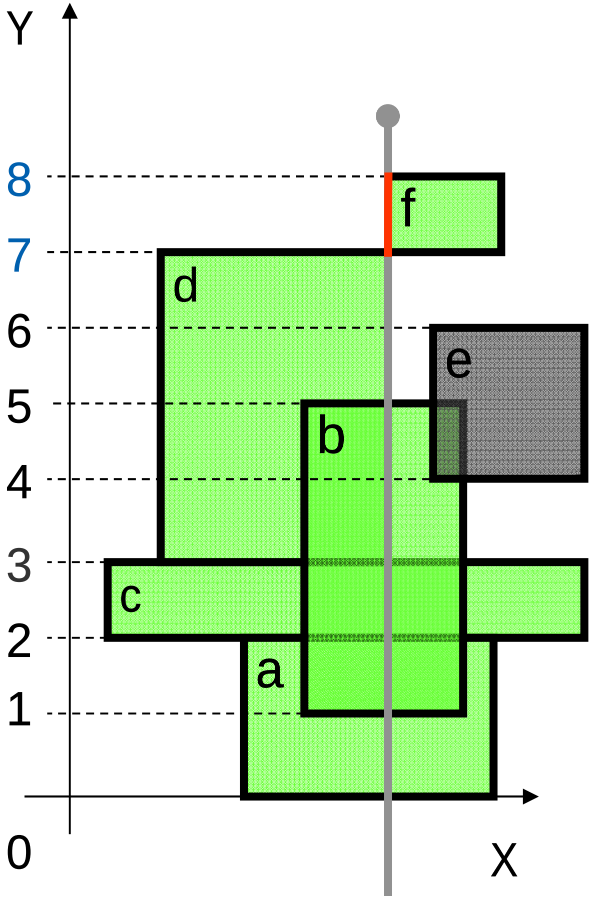
[Drtina]

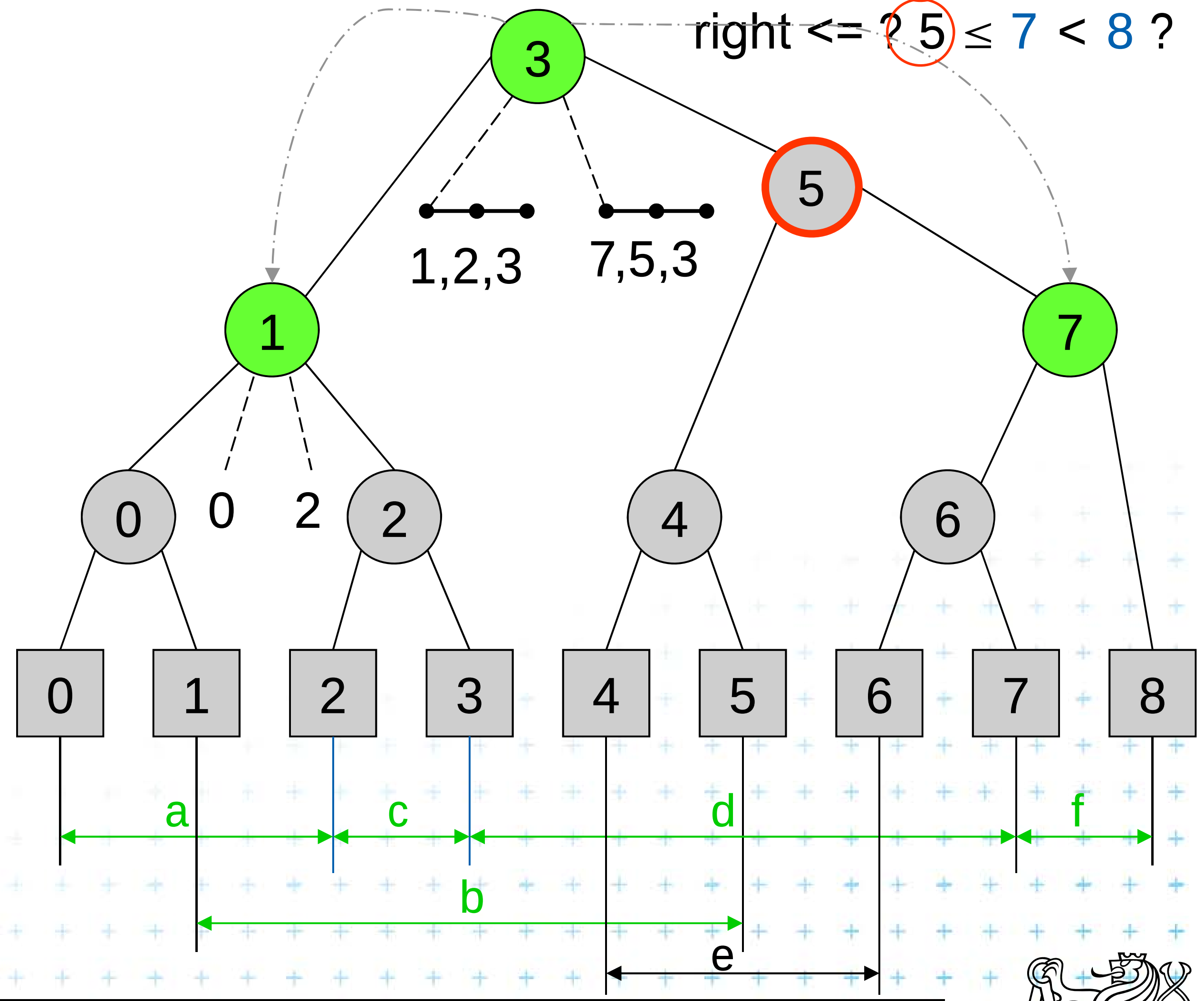
Insert [7,8] b) Insert Interval

b ≤ H(v) ≤ e

right <= ? 3 ≤ 7 < 8 ?

right <= ? 5 ≤ 7 < 8 ?





🟩 Active rectangle

○ Current node

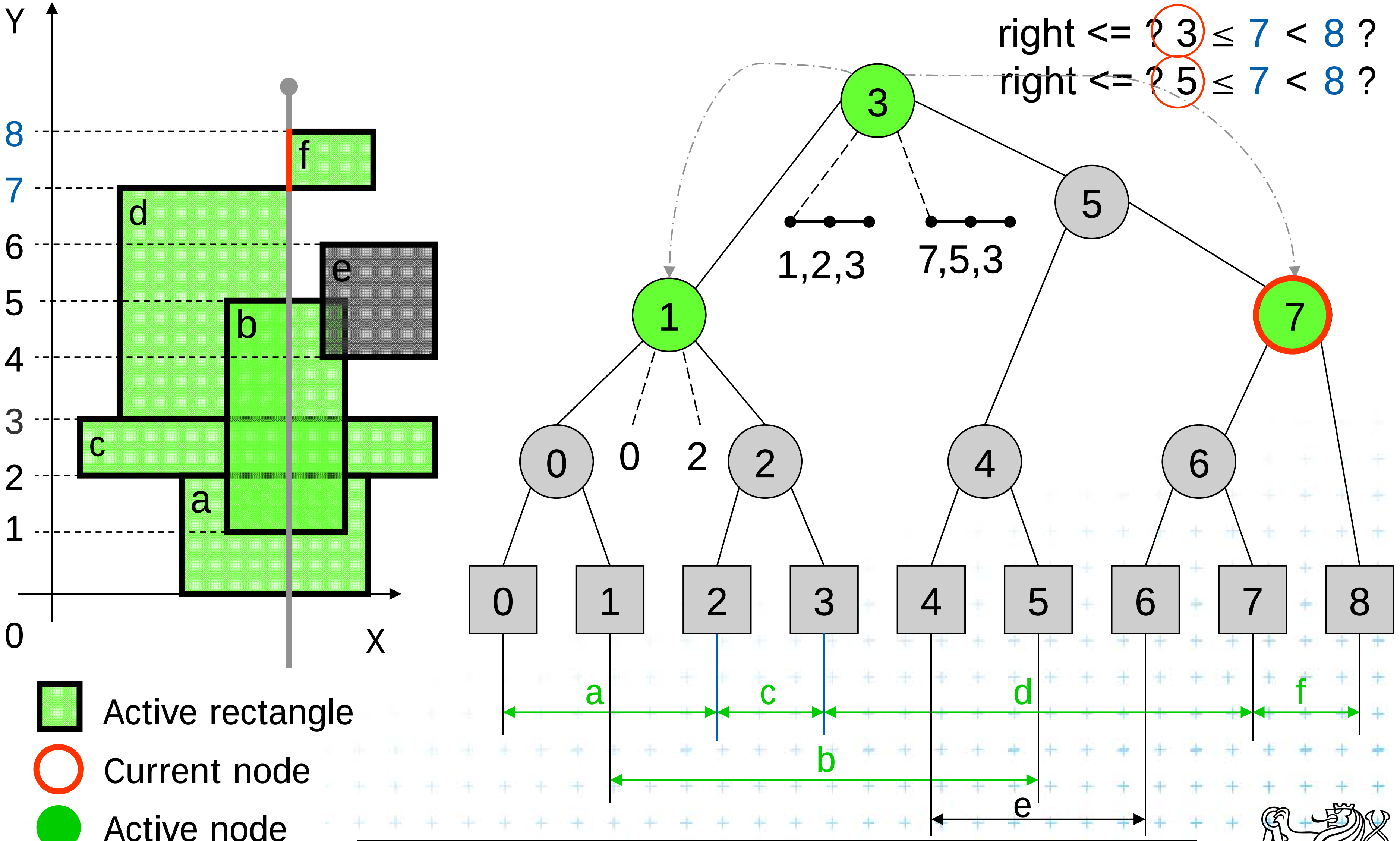
● Active node

[Drtina]

Insert [7,8] b) Insert Interval

$b \leq H(v) \leq e$

right <= ? 3 ≤ 7 < 8 ?
right <= ? 5 ≤ 7 < 8 ?



🟩 Active rectangle

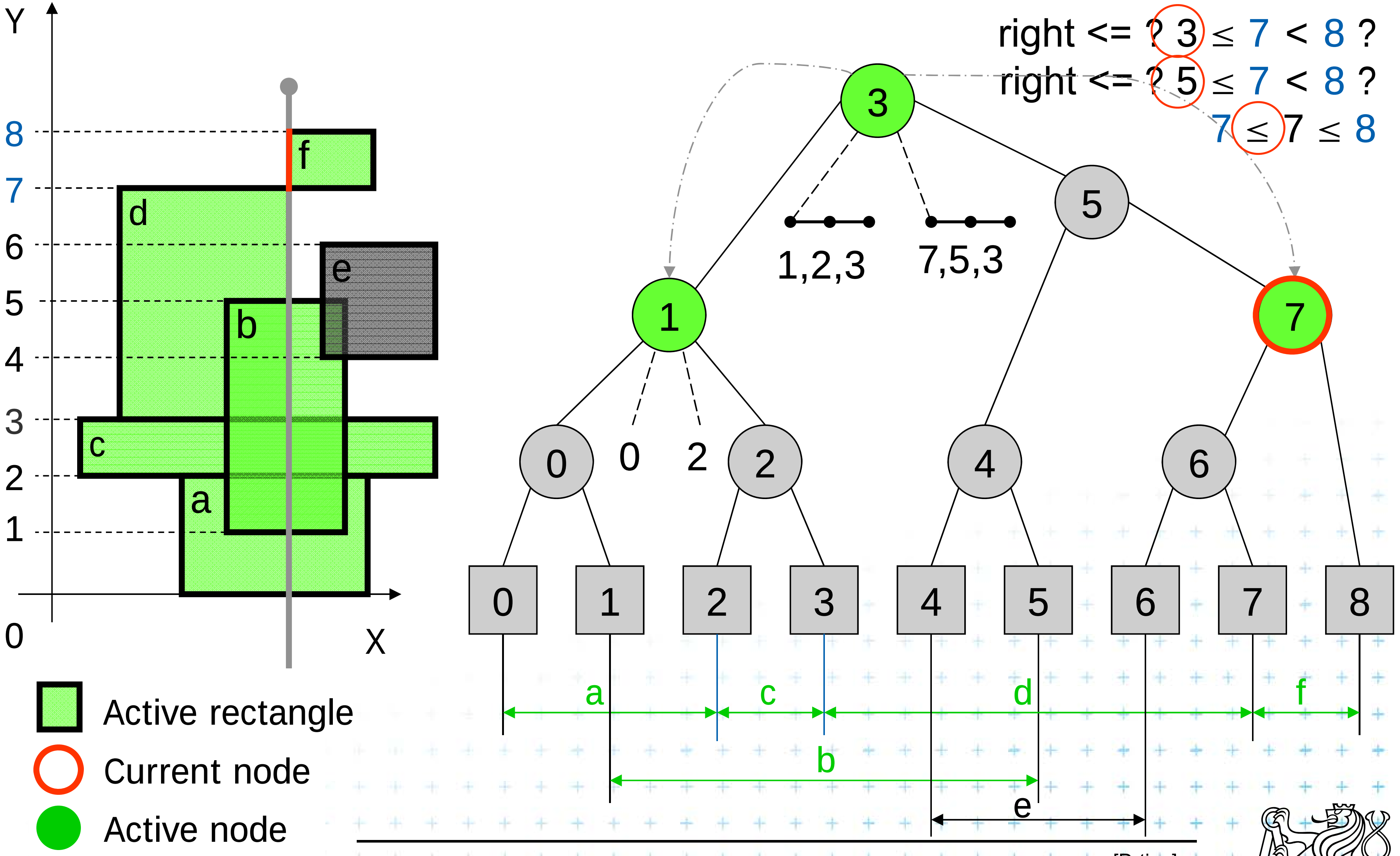
○ Current node

● Active node

[Drtina]

Insert [7,8] b) Insert Interval

b ≤ H(v) ≤ e



🟩 Active rectangle

○ Current node

● Active node

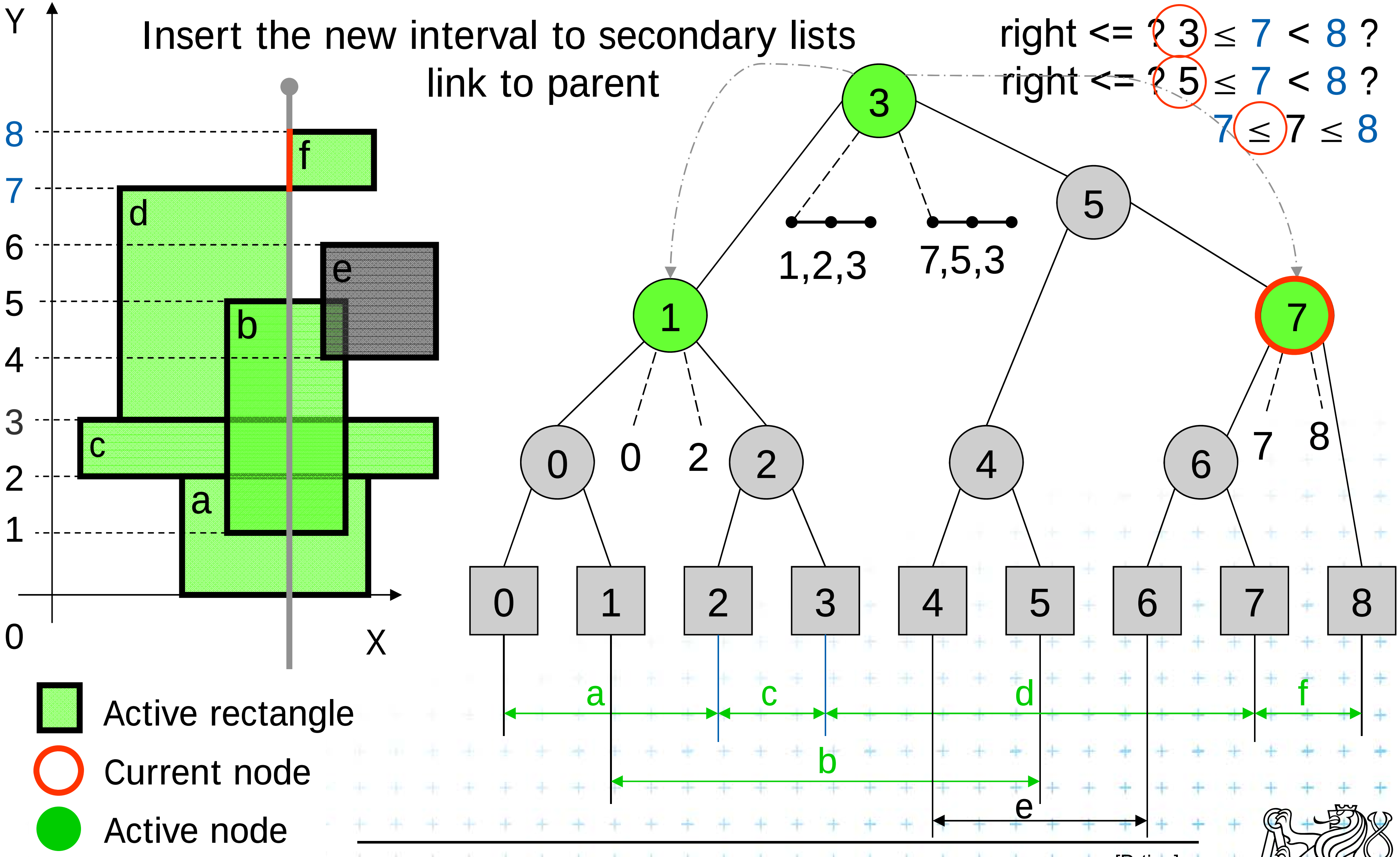
[Drtina]

Insert [7,8] b) Insert Interval

b ≤ H(v) ≤ e

Insert the new interval to secondary lists

link to parent



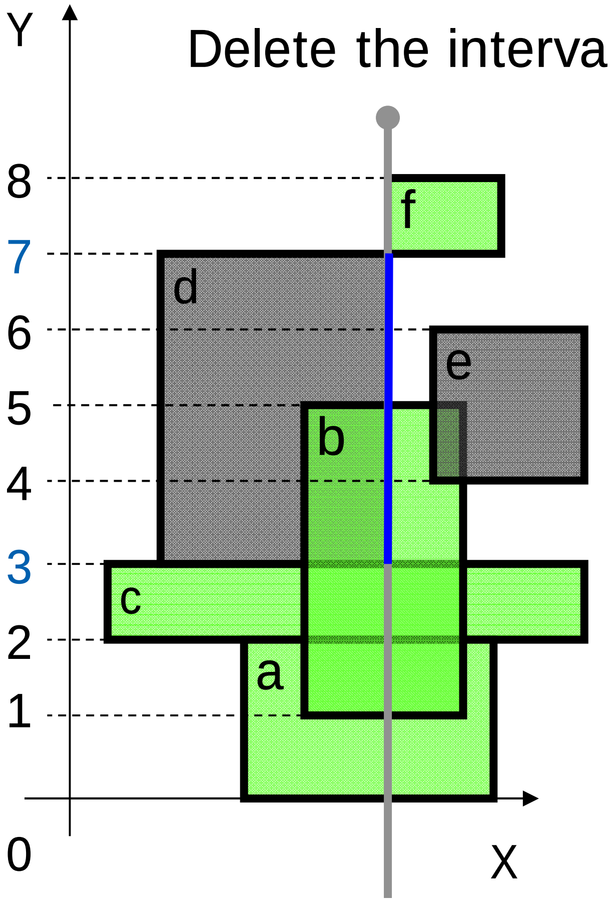
[Drtina]

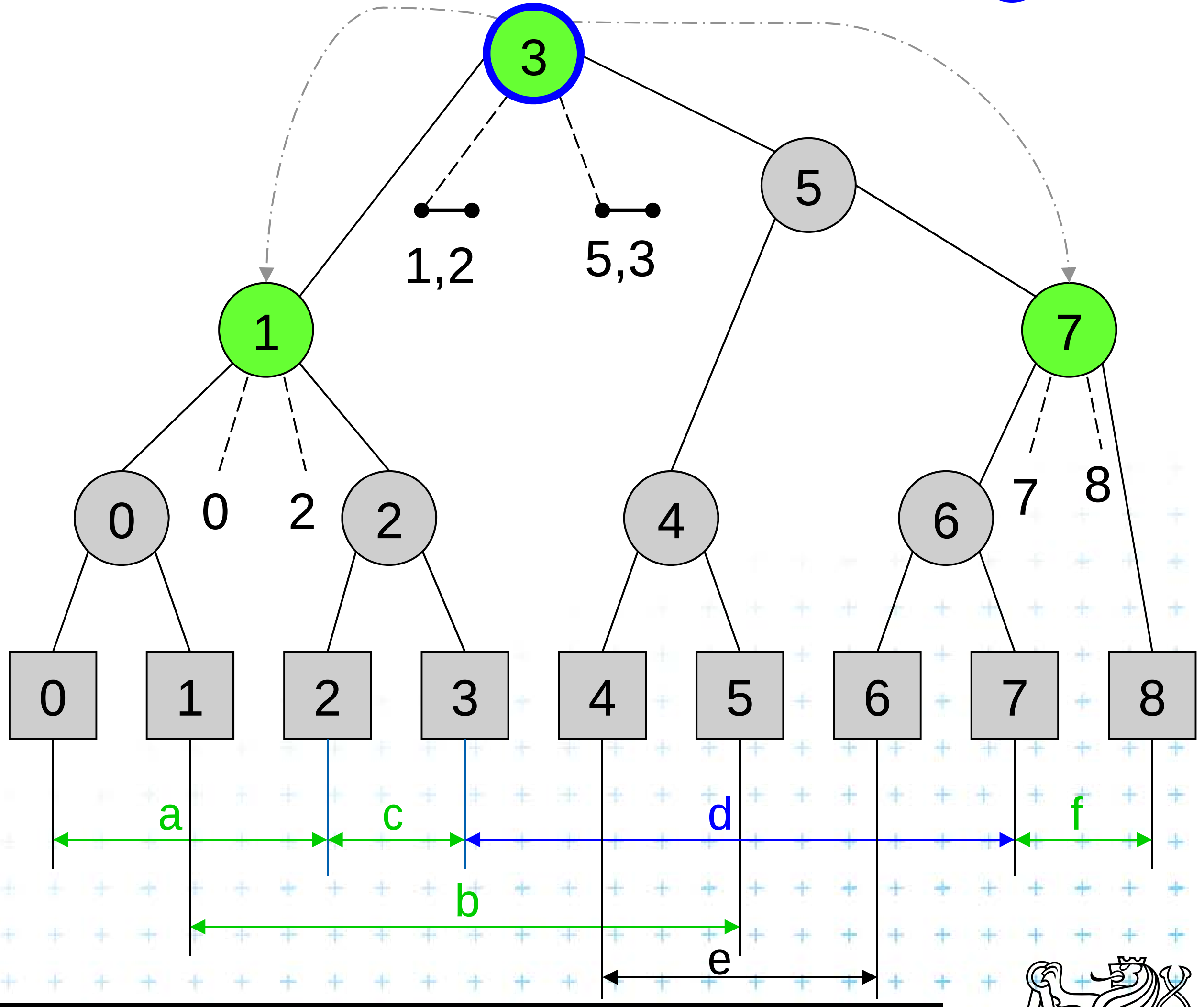
Delete [3,7] Delete Interval

b ≤ H(v) ≤ e

Delete the interval [3,7] from secondary lists

? 3 ≤ 7 ≤ 8 ?





■ Active rectangle

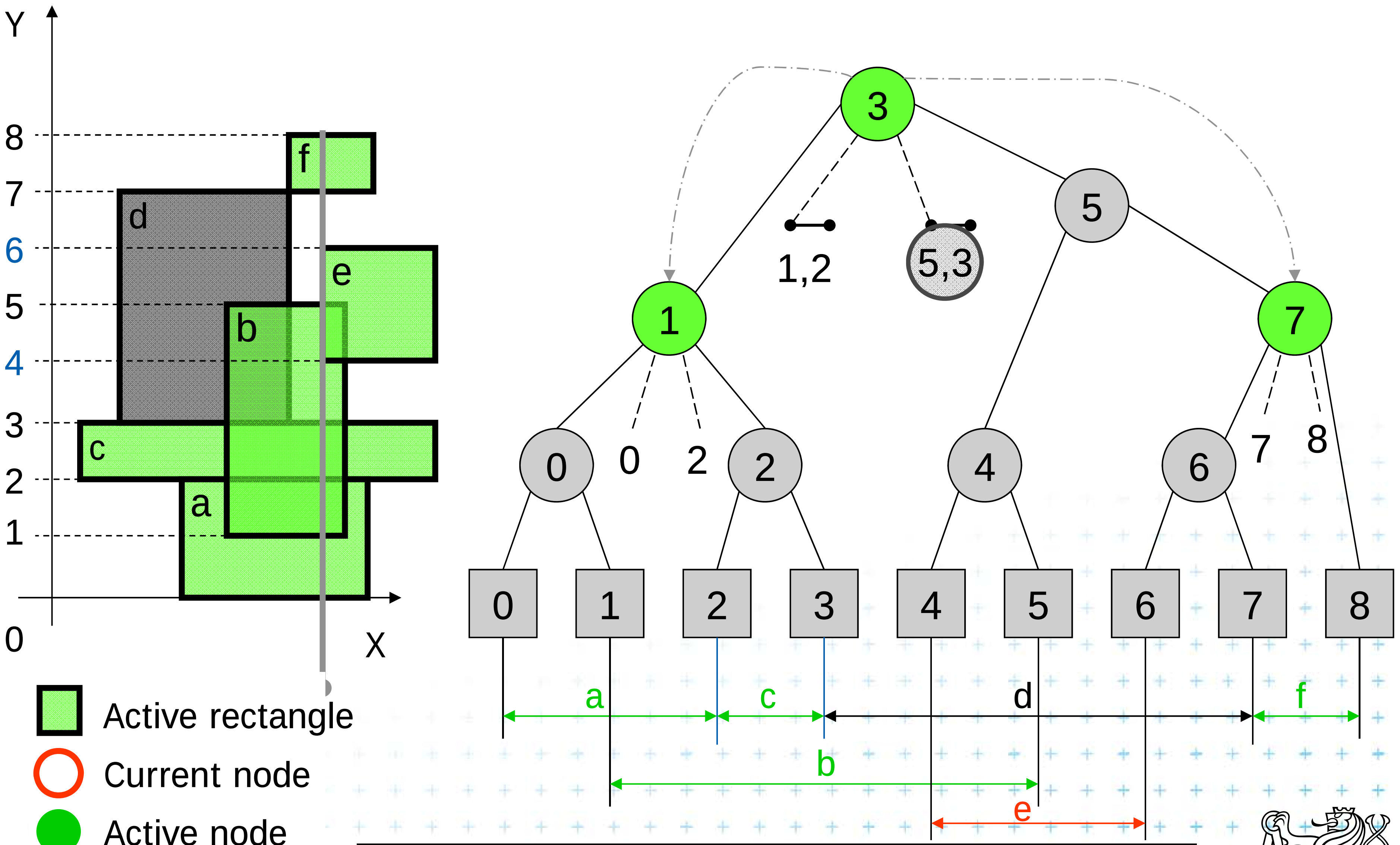
○ Current node

● Active node

[Drtina]

Insert [4,6] a) Query Interval

H(v) ≤ b < e



🟩 Active rectangle

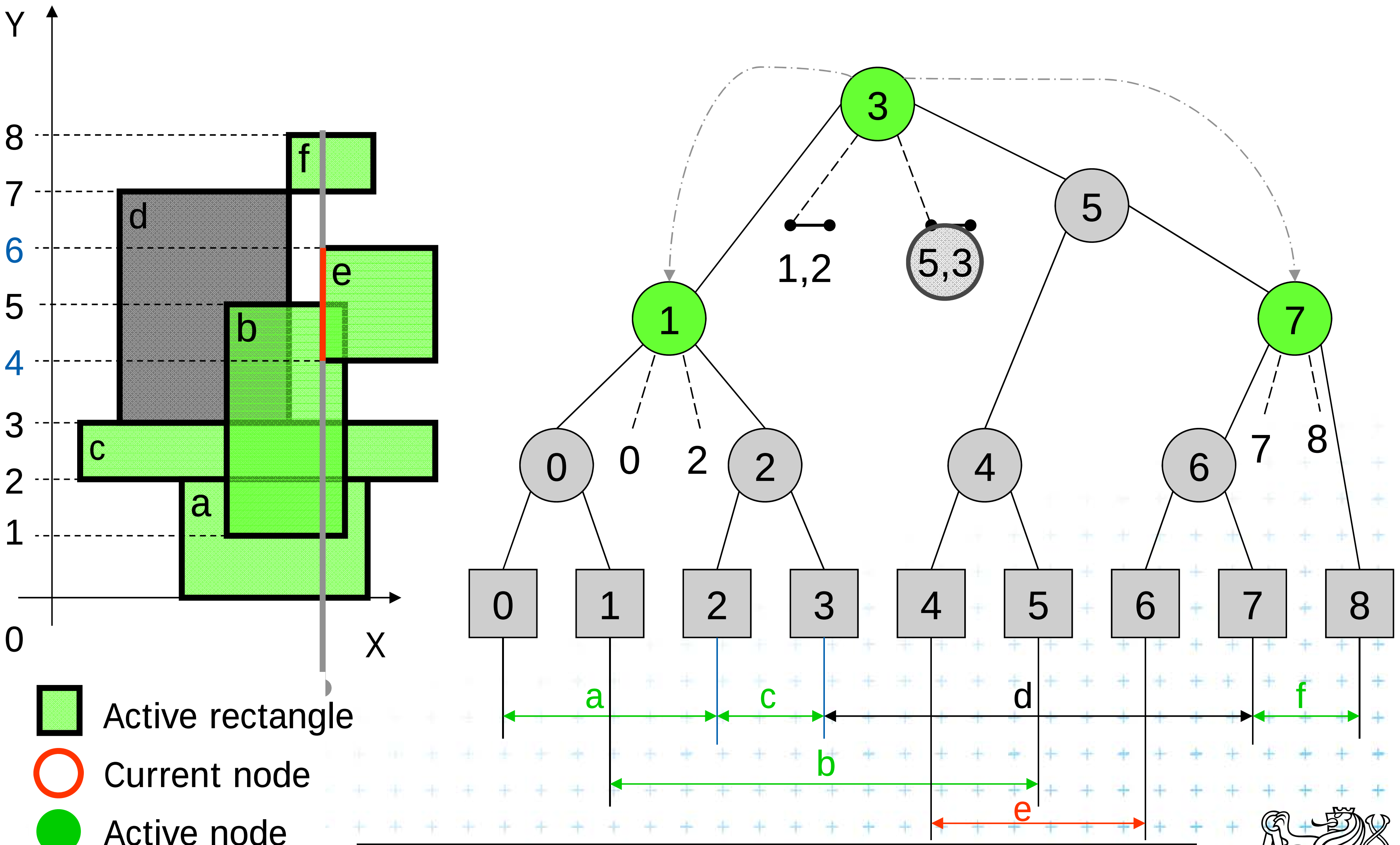
○ Current node

● Active node

[Drtina]

Insert [4,6] a) Query Interval

$H(v) \leq b < e$



🟩 Active rectangle

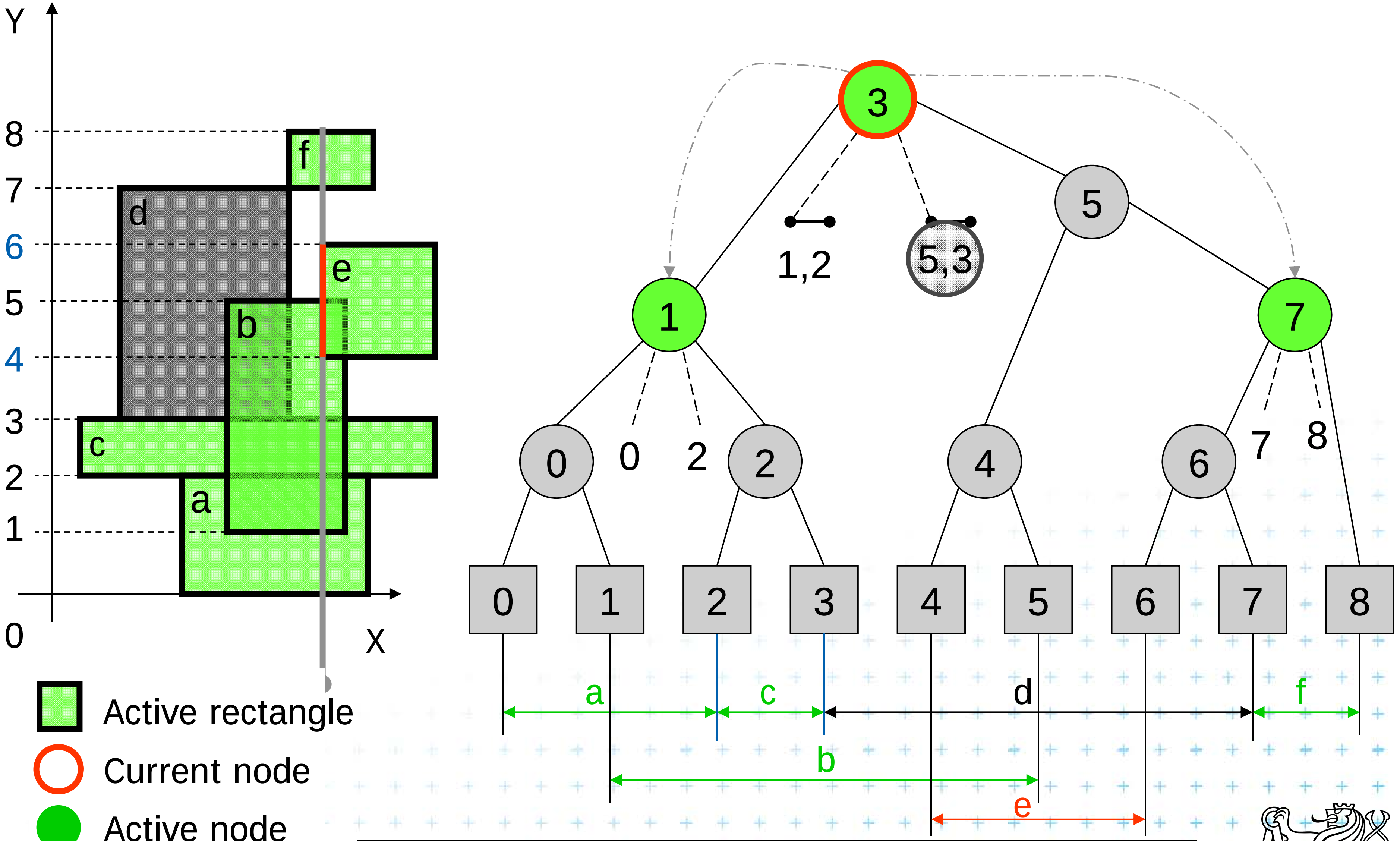
○ Current node

● Active node

[Drtina]

Insert [4,6] a) Query Interval

H(v) ≤ b < e



🟩 Active rectangle

○ Current node

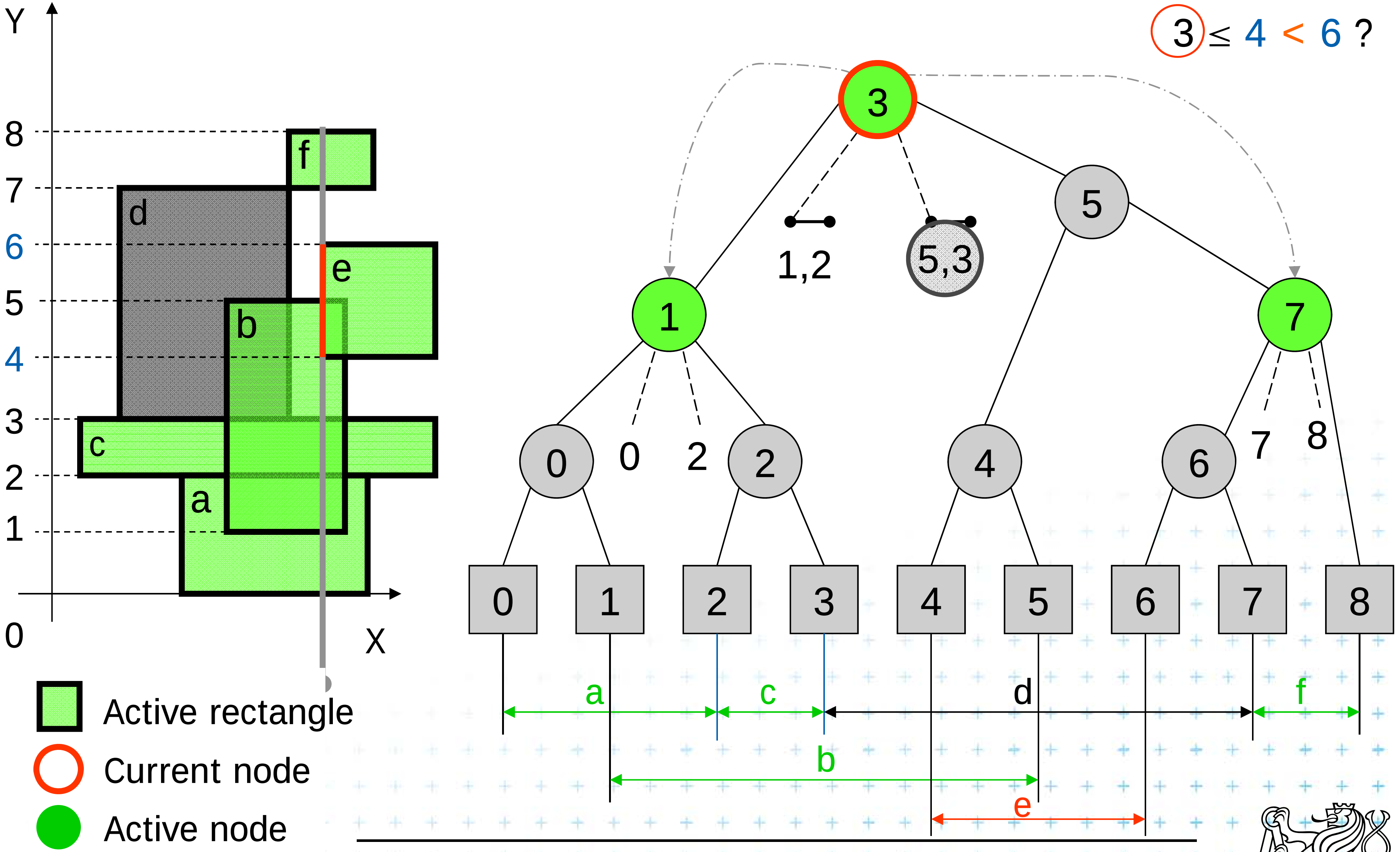
● Active node

[Drtina]

Insert [4,6] a) Query Interval

H(v) ≤ b < e

3 ≤ 4 < 6 ?

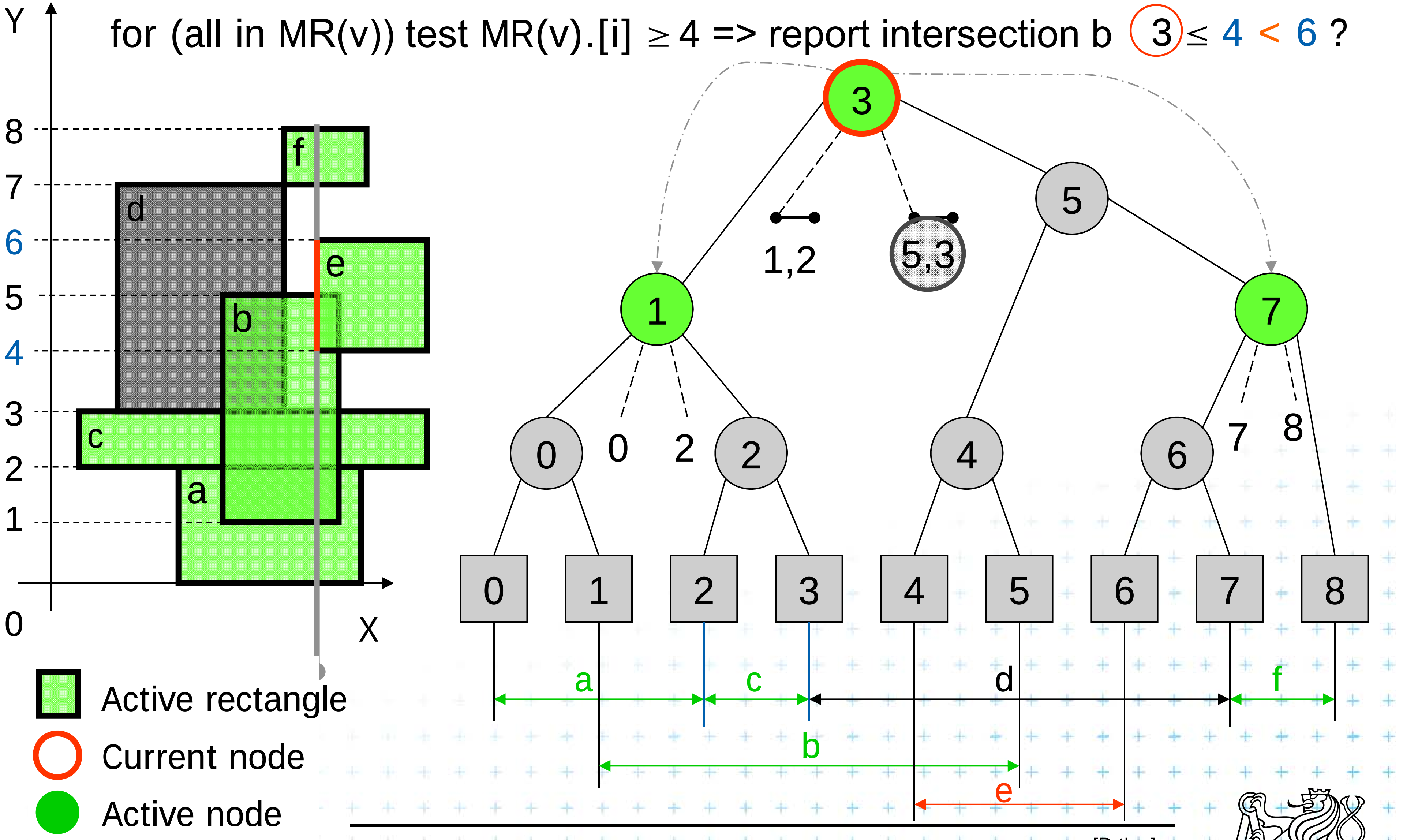


[Drtina]

Insert [4,6] a) Query Interval

H(v) ≤ b < e

for (all in MR(v)) test MR(v).[i] ≥ 4 => report intersection b 3 ≤ 4 < 6 ?



🟩 Active rectangle

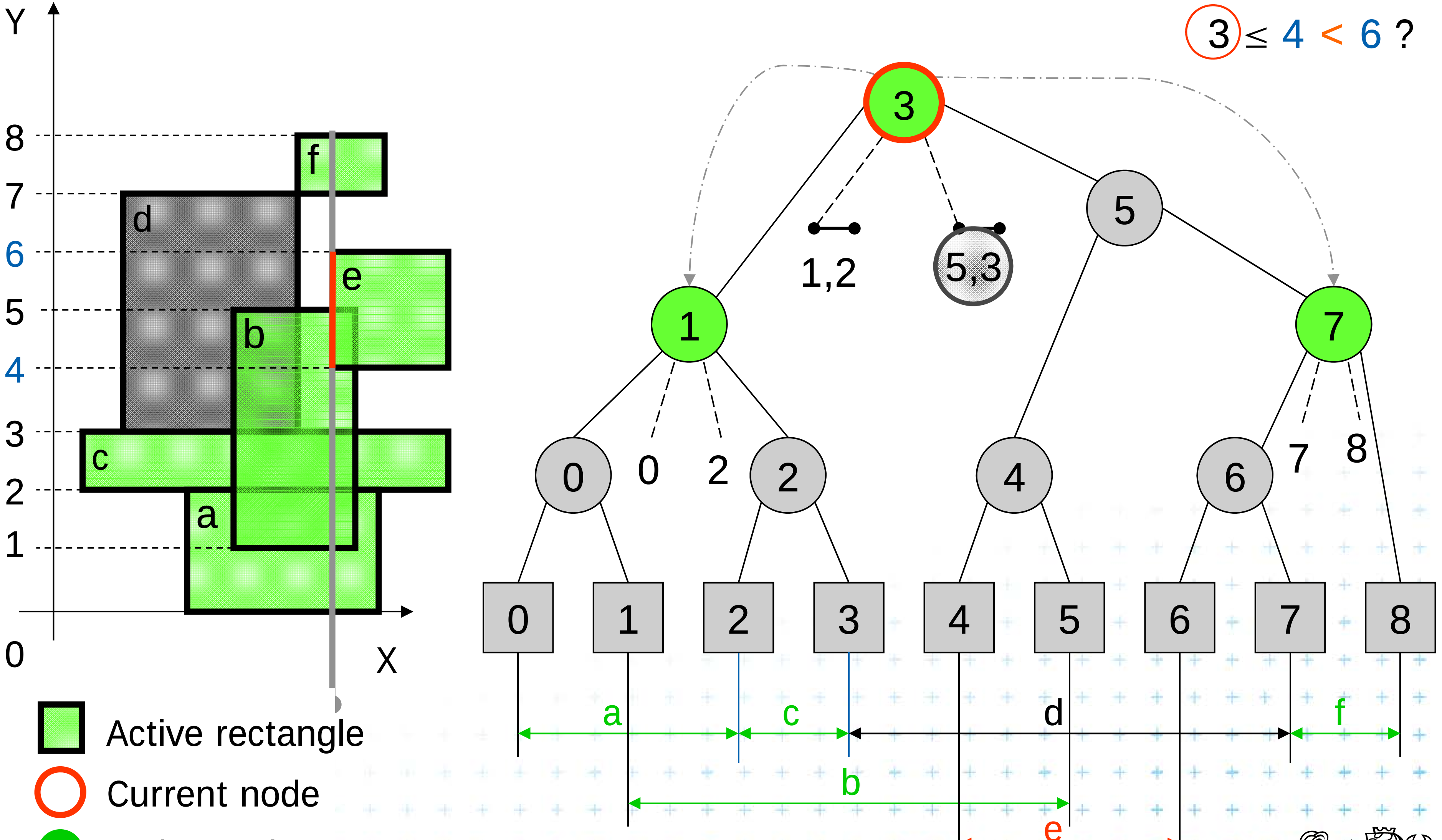
○ Current node

● Active node

[Drtina]

Insert [4,6] a) Query Interval

H(v) ≤ b < e

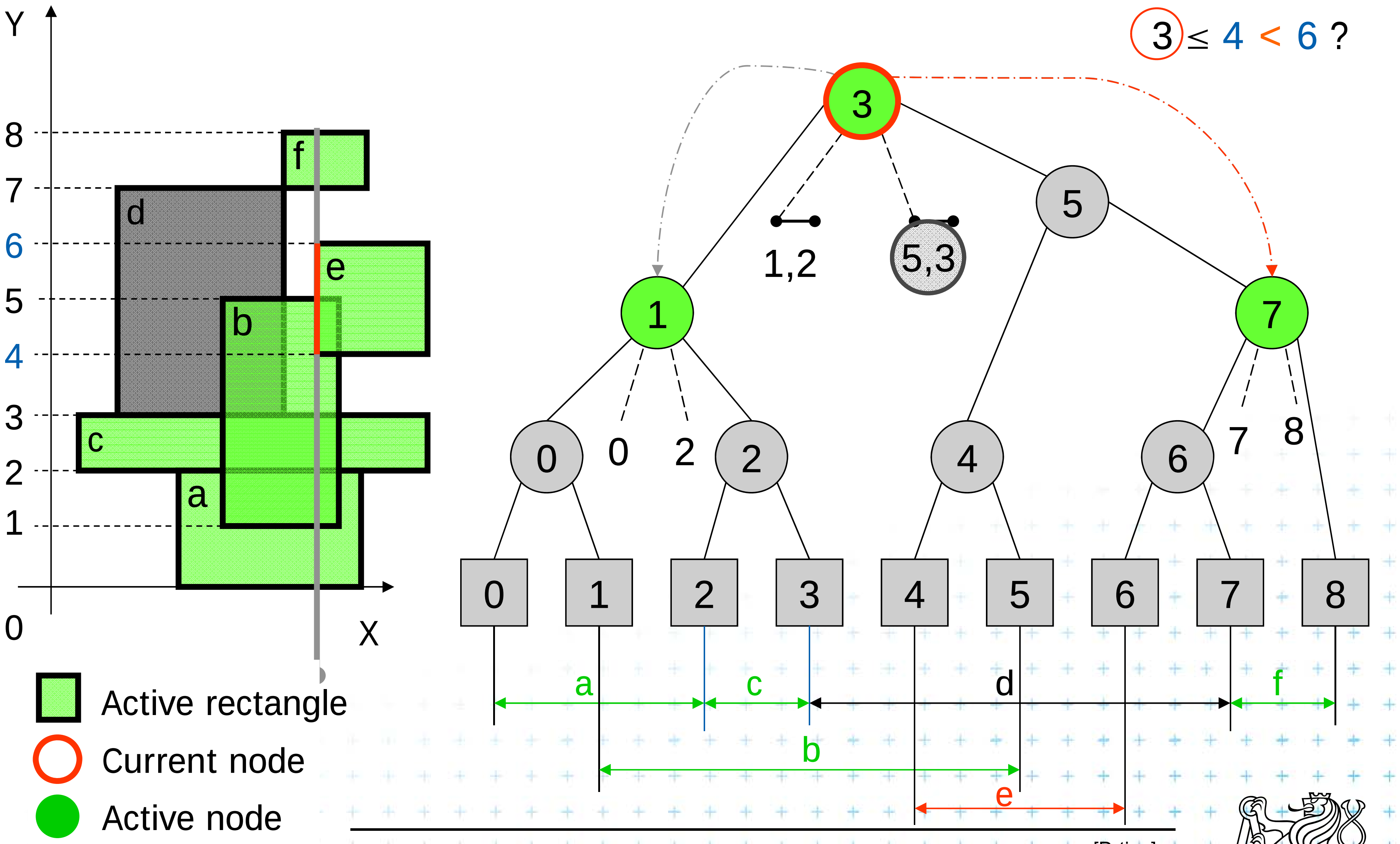


- 🟩 Active rectangle
- ○ Current node
- ● Active node

[Drtina]

Insert [4,6] a) Query Interval

H(v) ≤ b < e



🟩 Active rectangle

○ Current node

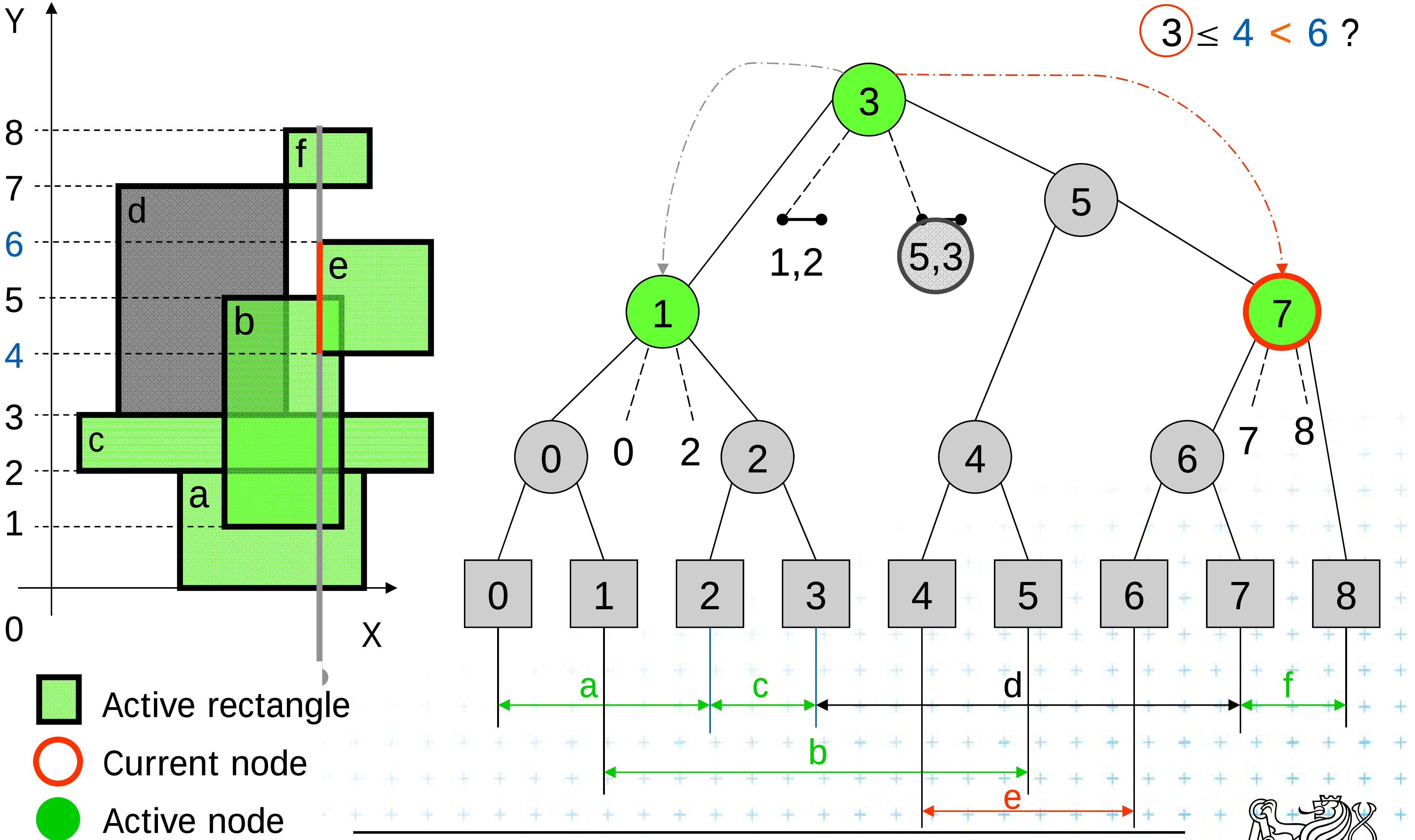
● Active node

[Drtina]

Insert [4,6] a) Query Interval

$H(v) \leq b < e$

$3 \leq 4 < 6$?



🟩 Active rectangle

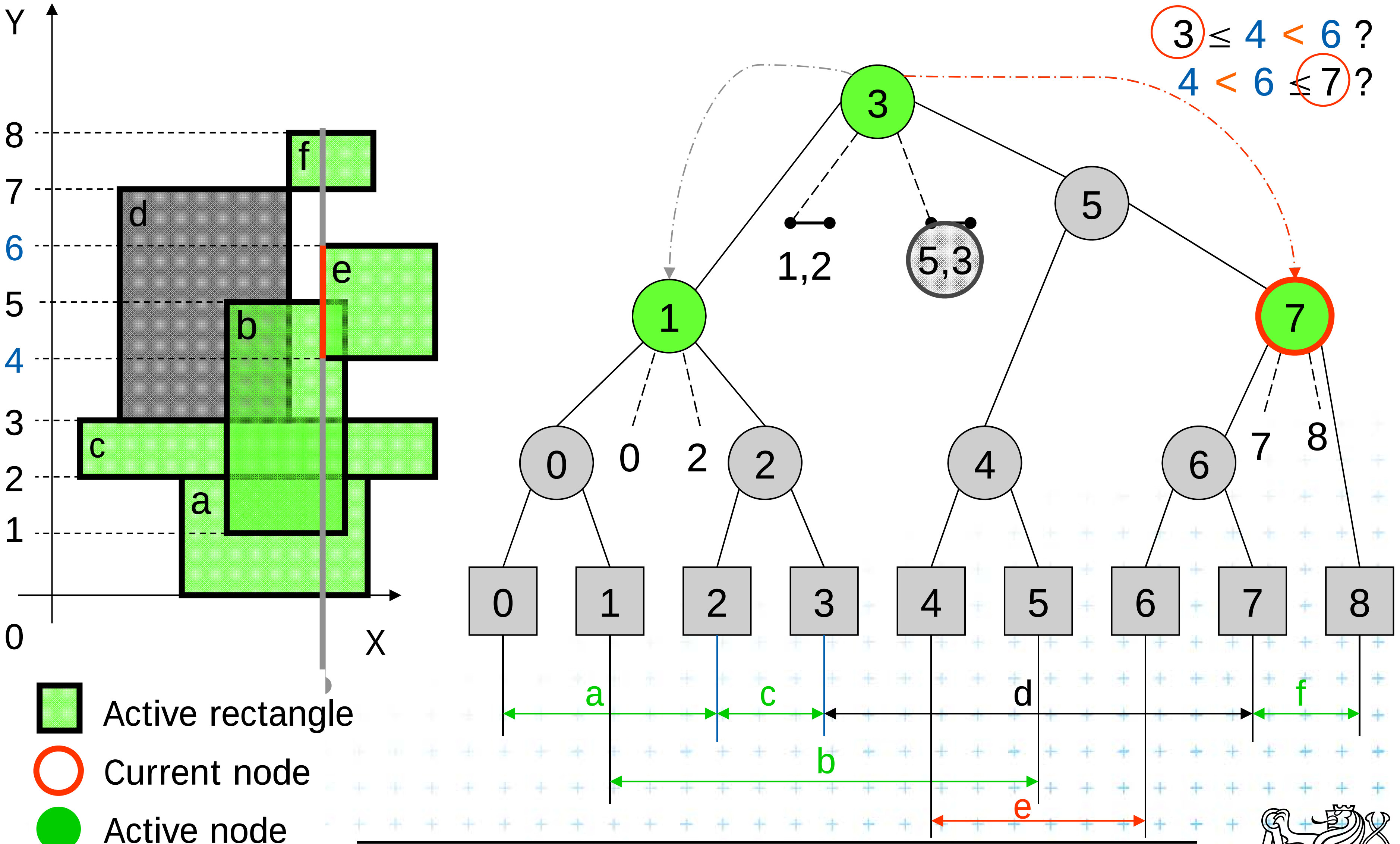
○ Current node

● Active node

[Drtina]

Insert [4,6] a) Query Interval

H(v) ≤ b < e



🟩 Active rectangle

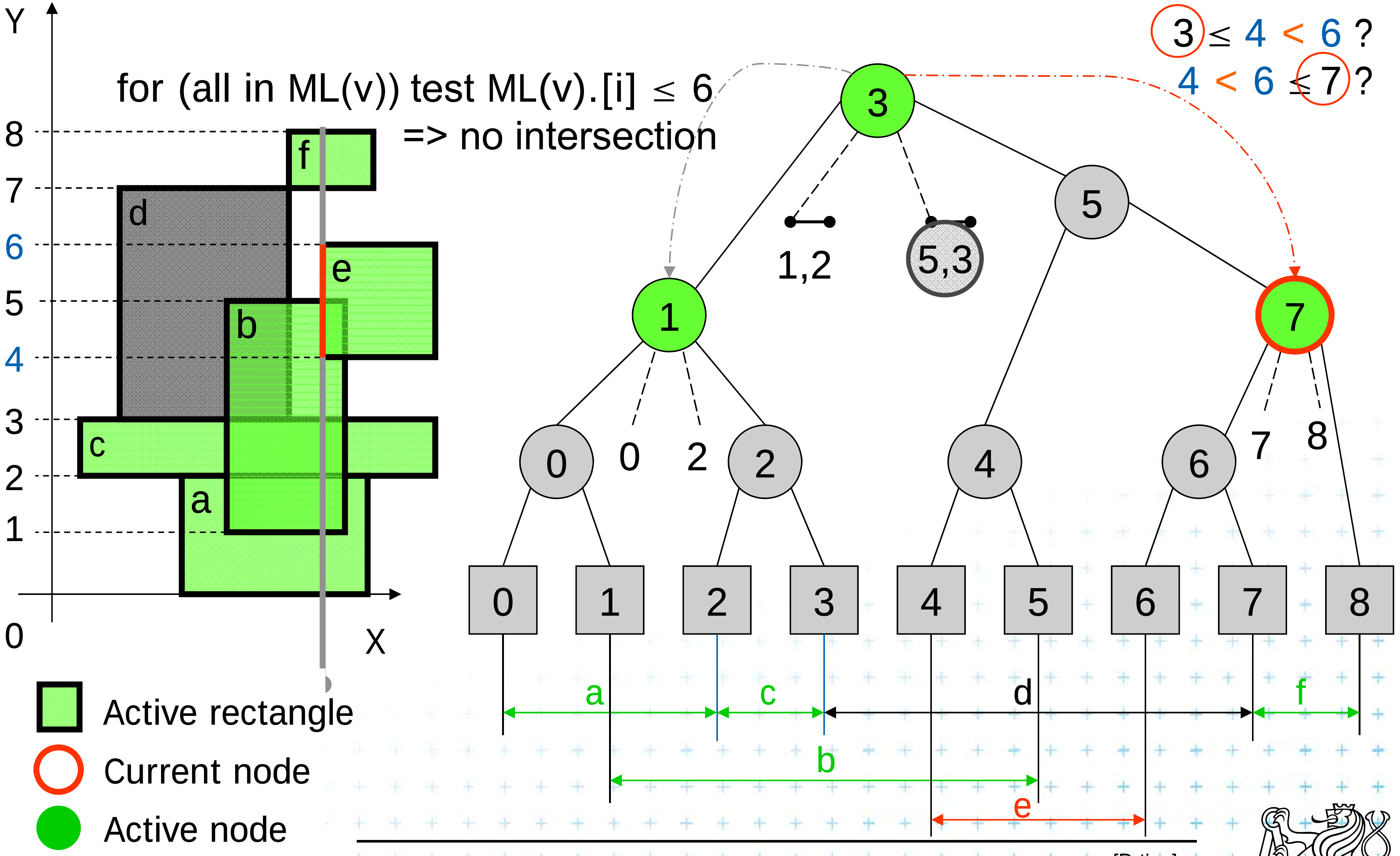
○ Current node

● Active node

[Drtina]

Insert [4,6] a) Query Interval

$H(v) \leq b < e$



🟩 Active rectangle

○ Current node

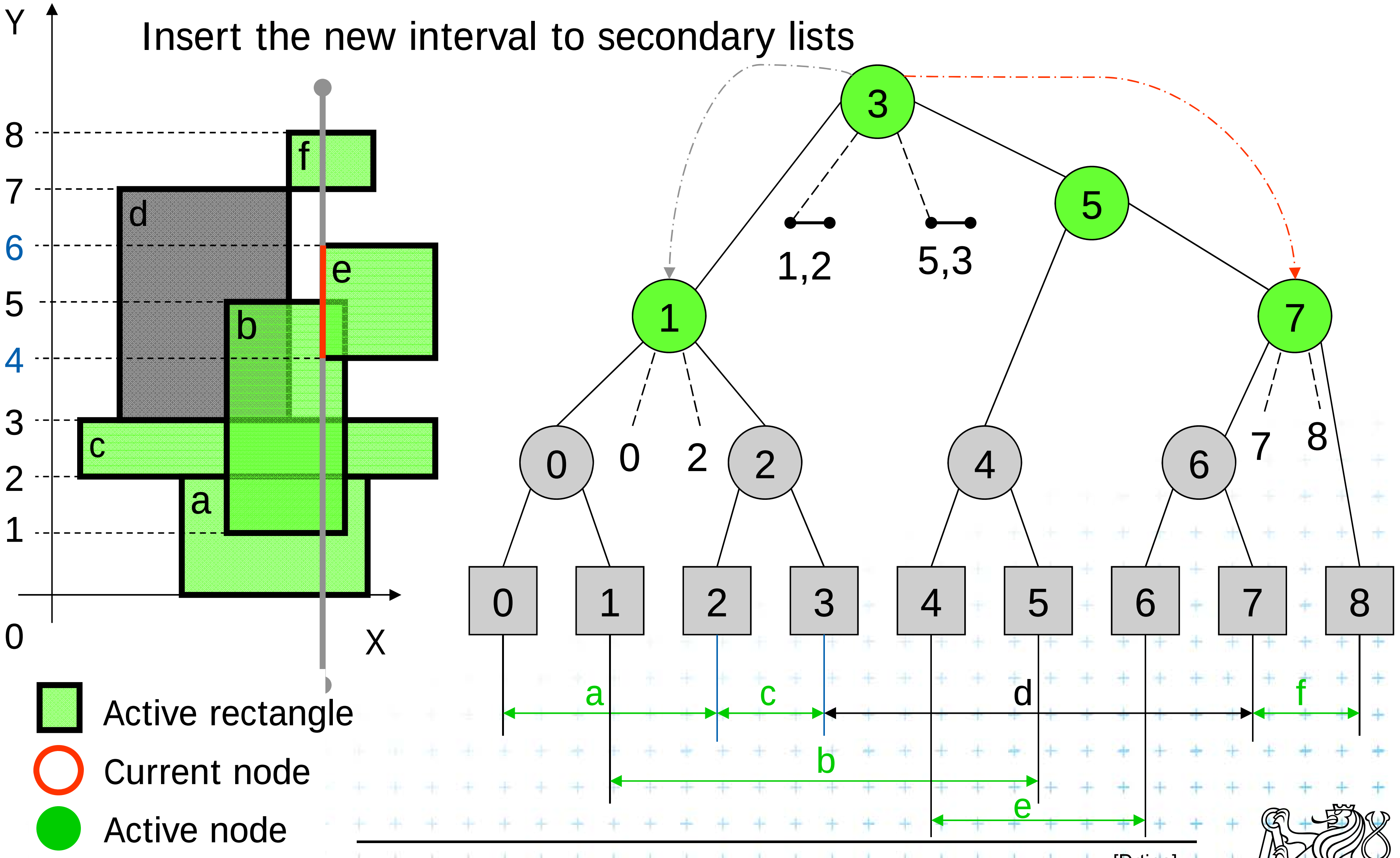
● Active node

[Drtina]

Insert [4,6] b) Insert Interval

H(v) ≤ b < e

Insert the new interval to secondary lists

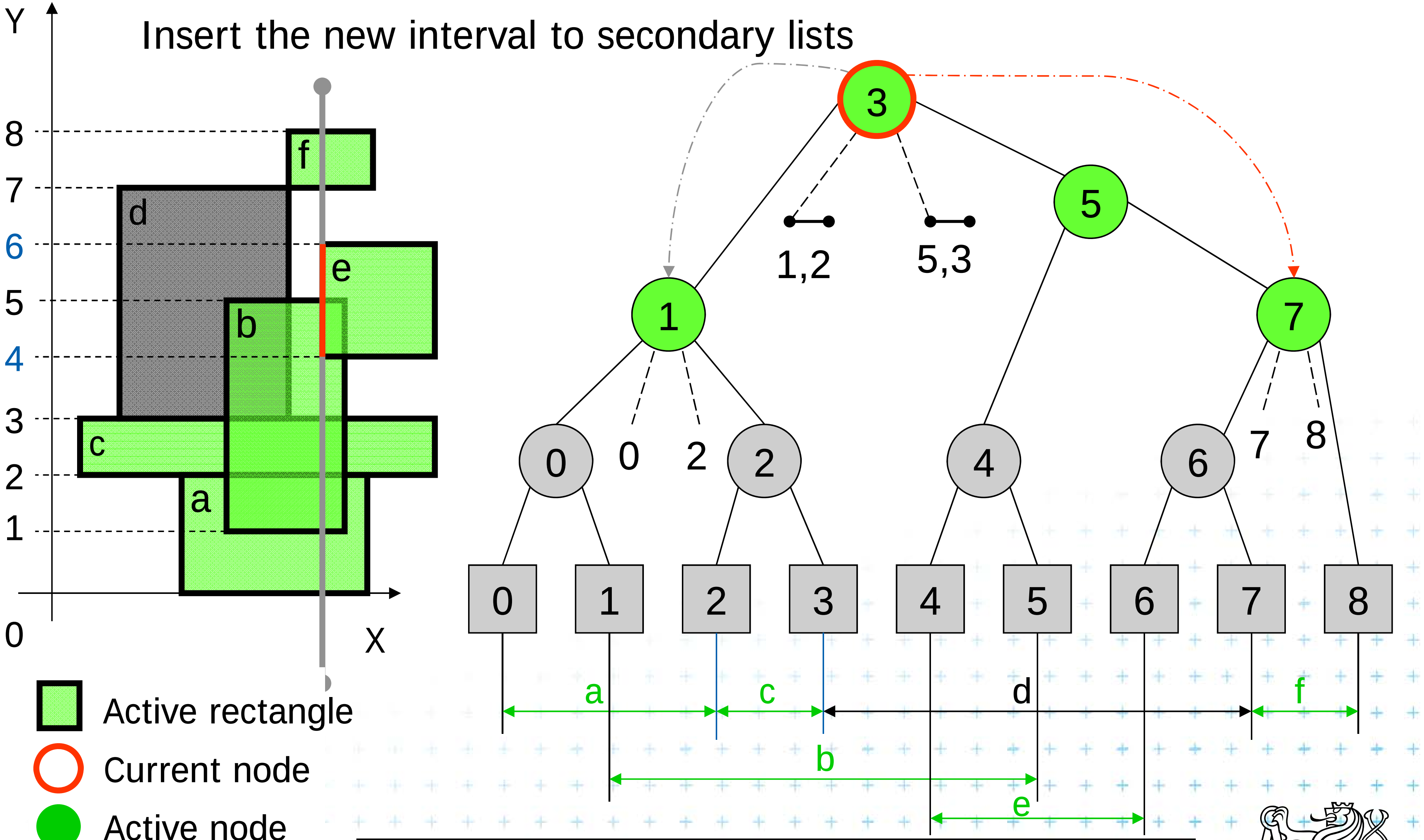


[Drtina]

Insert [4,6] b) Insert Interval

H(v) ≤ b < e

Insert the new interval to secondary lists



🟩 Active rectangle

○ Current node

● Active node

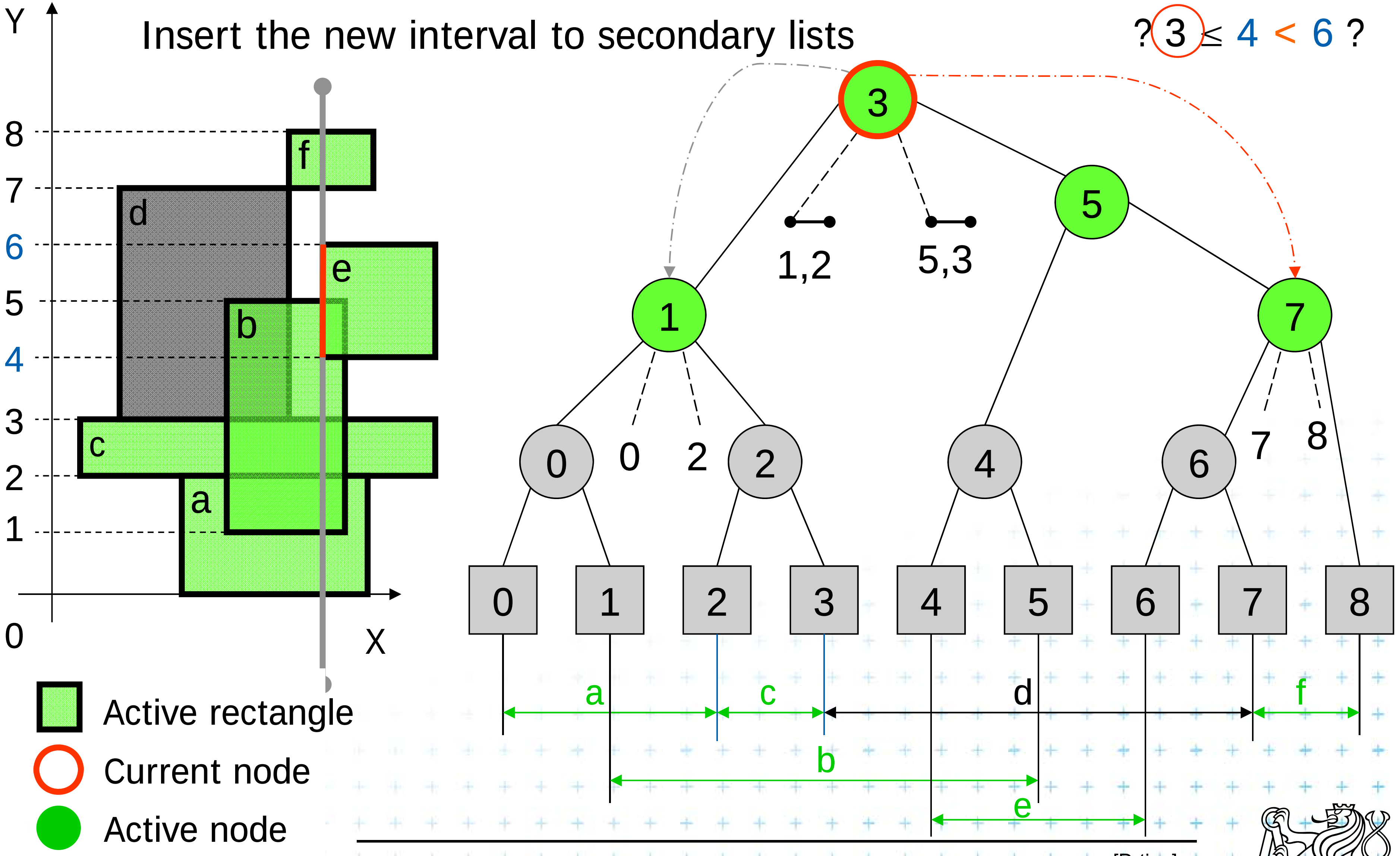
[Drtina]

Insert [4,6] b) Insert Interval

H(v) ≤ b < e

Insert the new interval to secondary lists

? 3 ≤ 4 < 6 ?



🟩 Active rectangle

○ Current node

● Active node

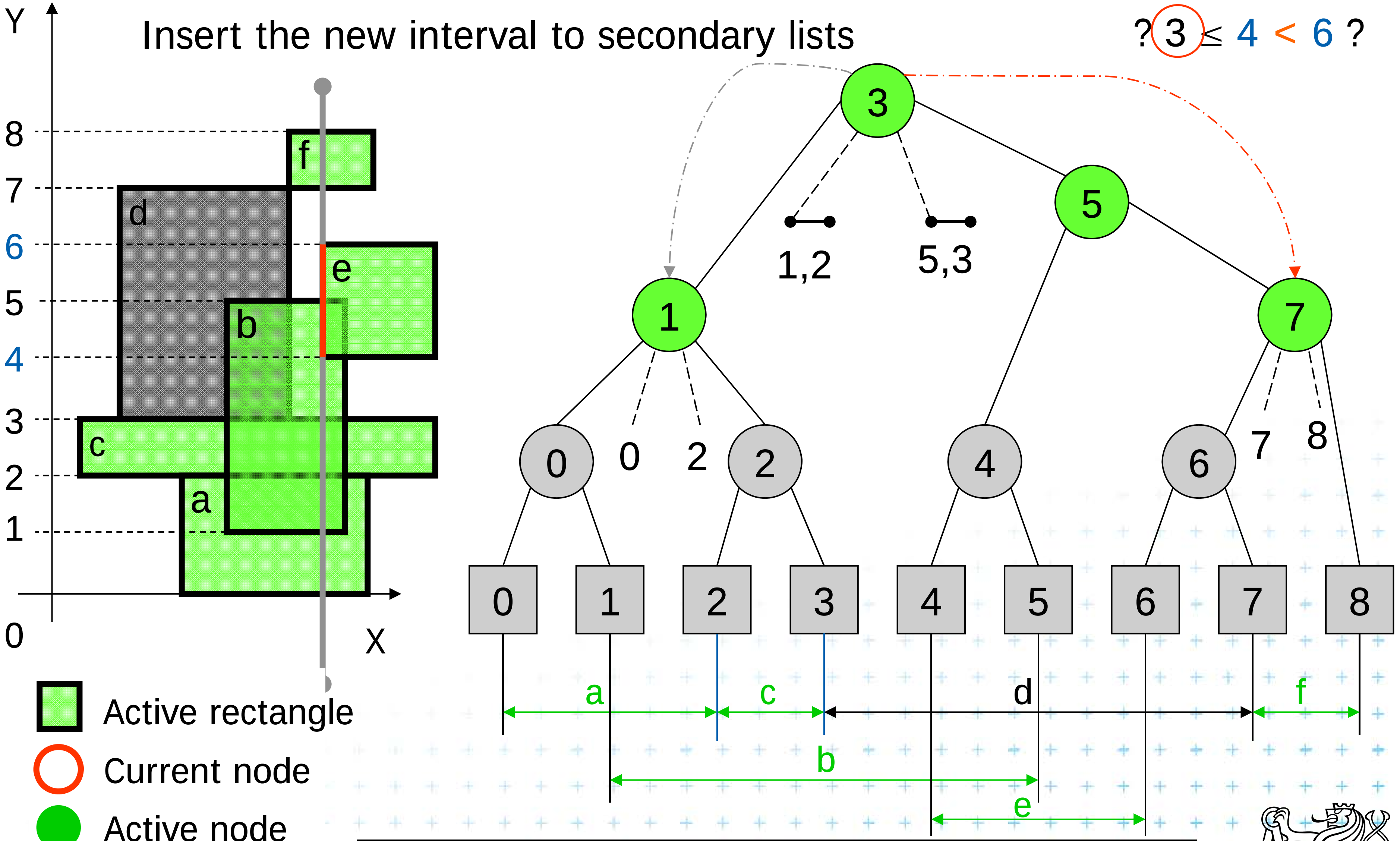
[Drtina]

Insert [4,6] b) Insert Interval

H(v) ≤ b < e

Insert the new interval to secondary lists

? 3 ≤ 4 < 6 ?



🟩 Active rectangle

○ Current node

● Active node

[Drtina]

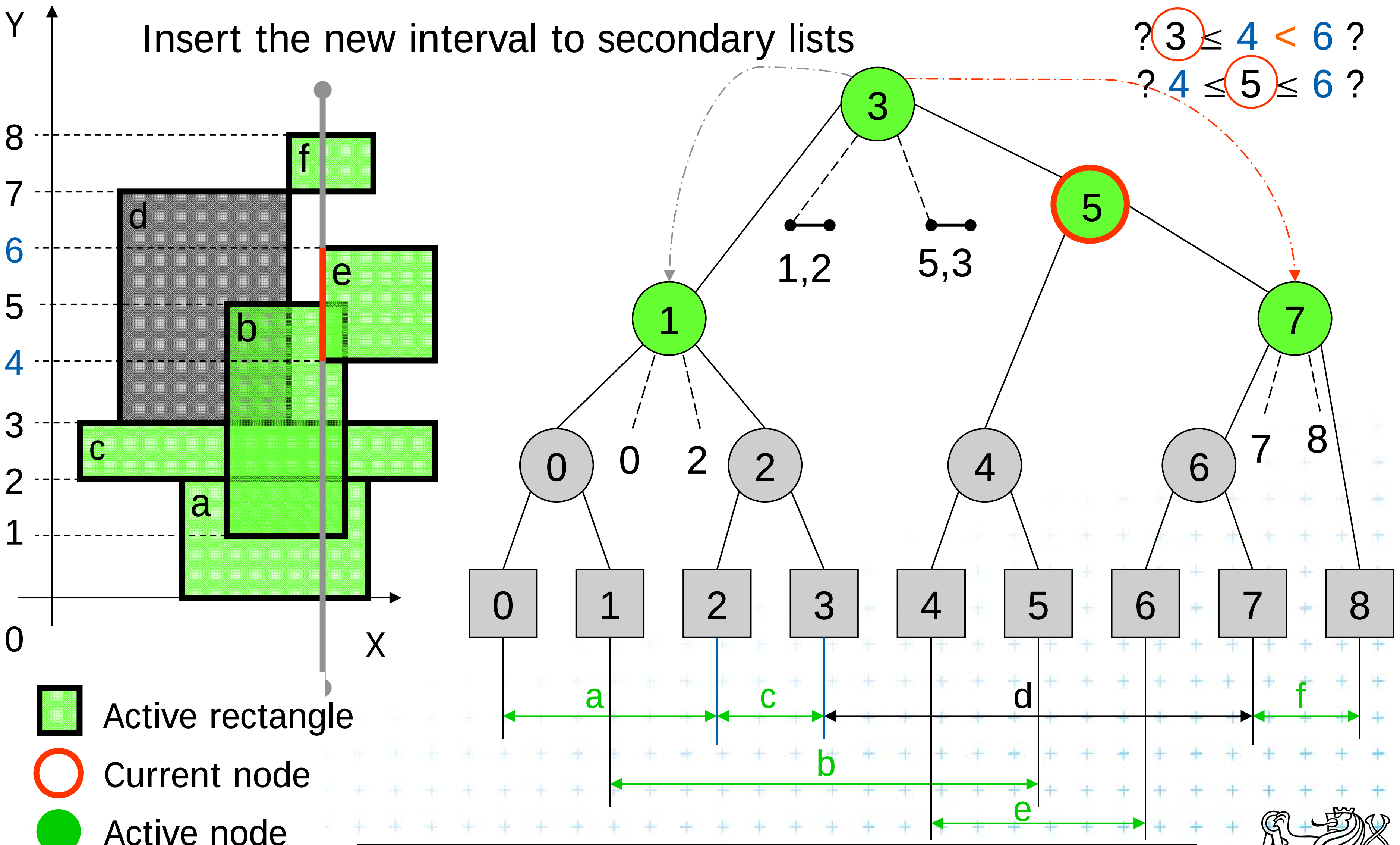
Insert [4,6] b) Insert Interval

H(v) ≤ b < e

Insert the new interval to secondary lists

? 3 ≤ 4 < 6 ?

? 4 ≤ 5 ≤ 6 ?



🟩 Active rectangle

○ Current node

● Active node

[Drtina]

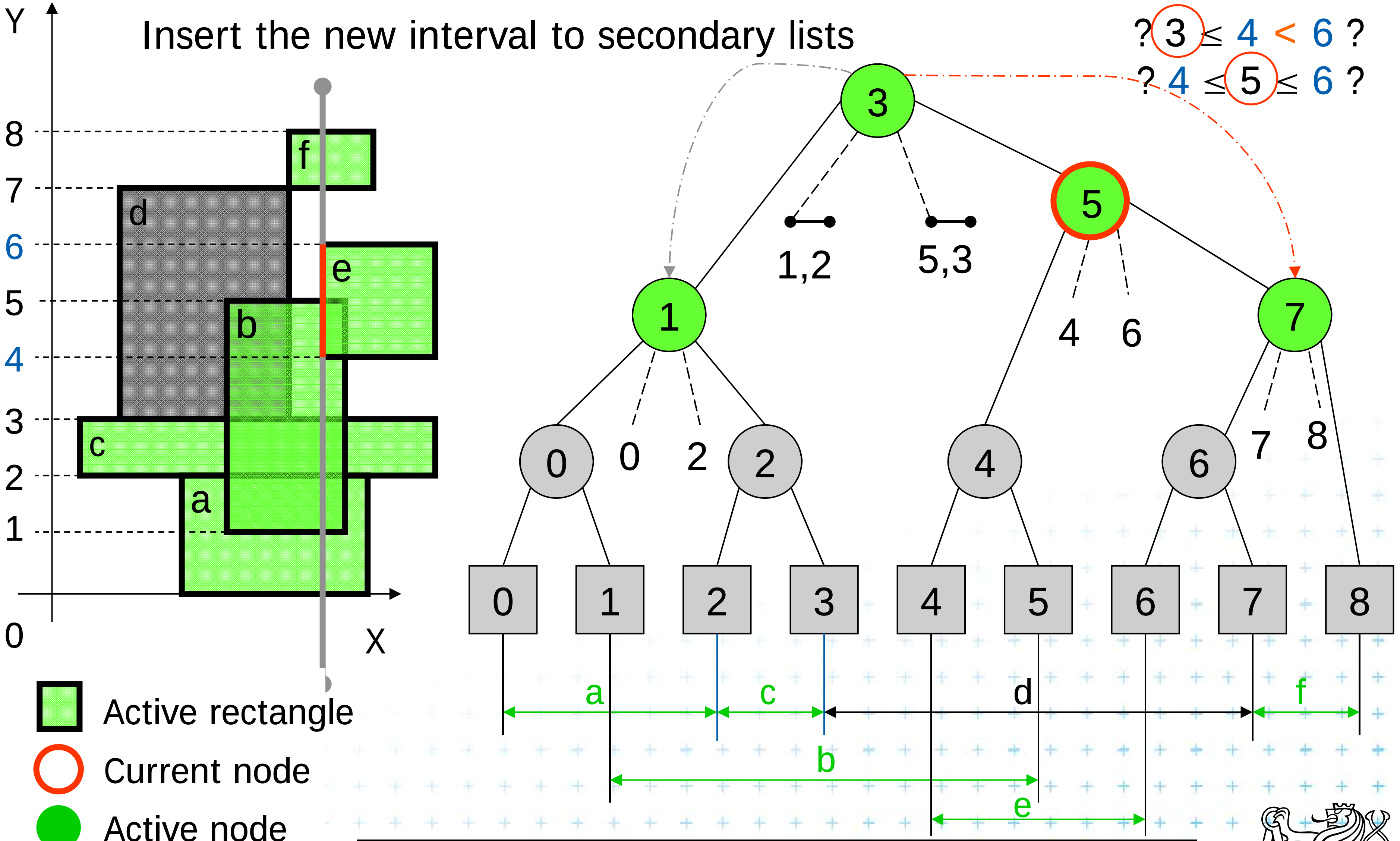
Insert [4,6] b) Insert Interval

$H(v) \le b < e$

Insert the new interval to secondary lists

? $3 \le 4 < 6$?

? $4 \le 5 \le 6$?



🟩 Active rectangle

○ Current node

● Active node

[Drtina]

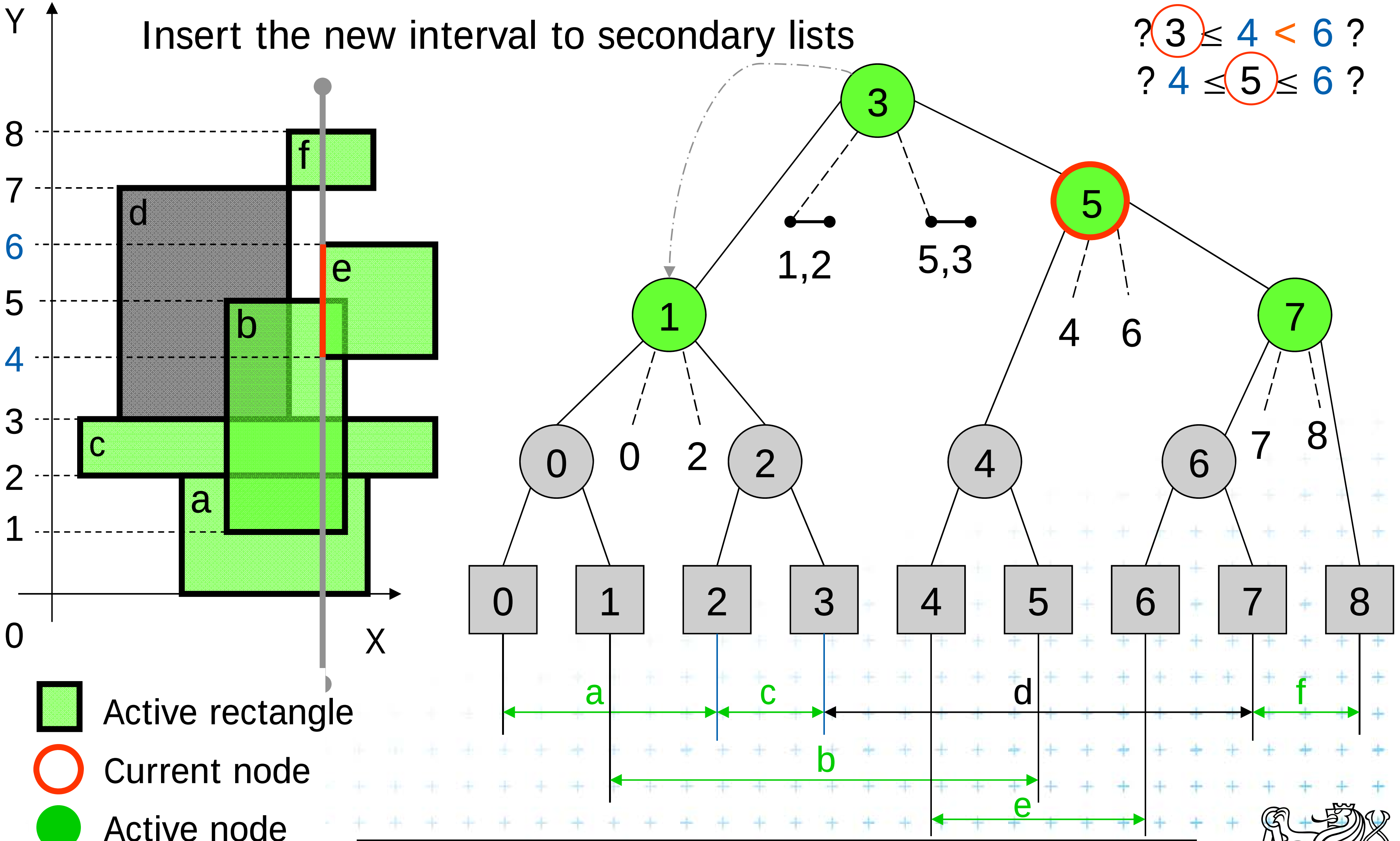
Insert [4,6] b) Insert Interval

H(v) ≤ b < e

Insert the new interval to secondary lists

? 3 ≤ 4 < 6 ?

? 4 ≤ 5 ≤ 6 ?



🟩 Active rectangle

○ Current node

● Active node

[Drtina]

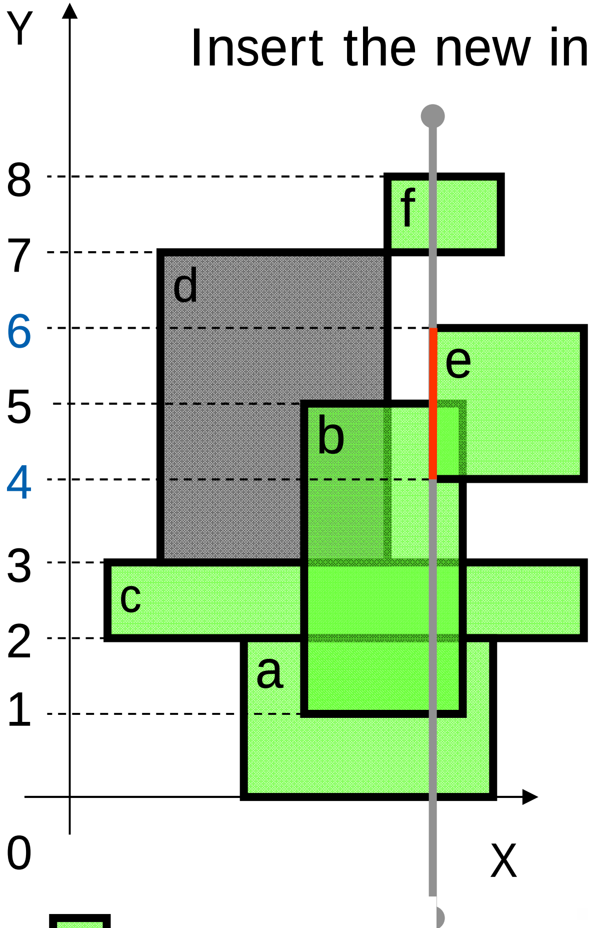
Insert [4,6] b) Insert Interval

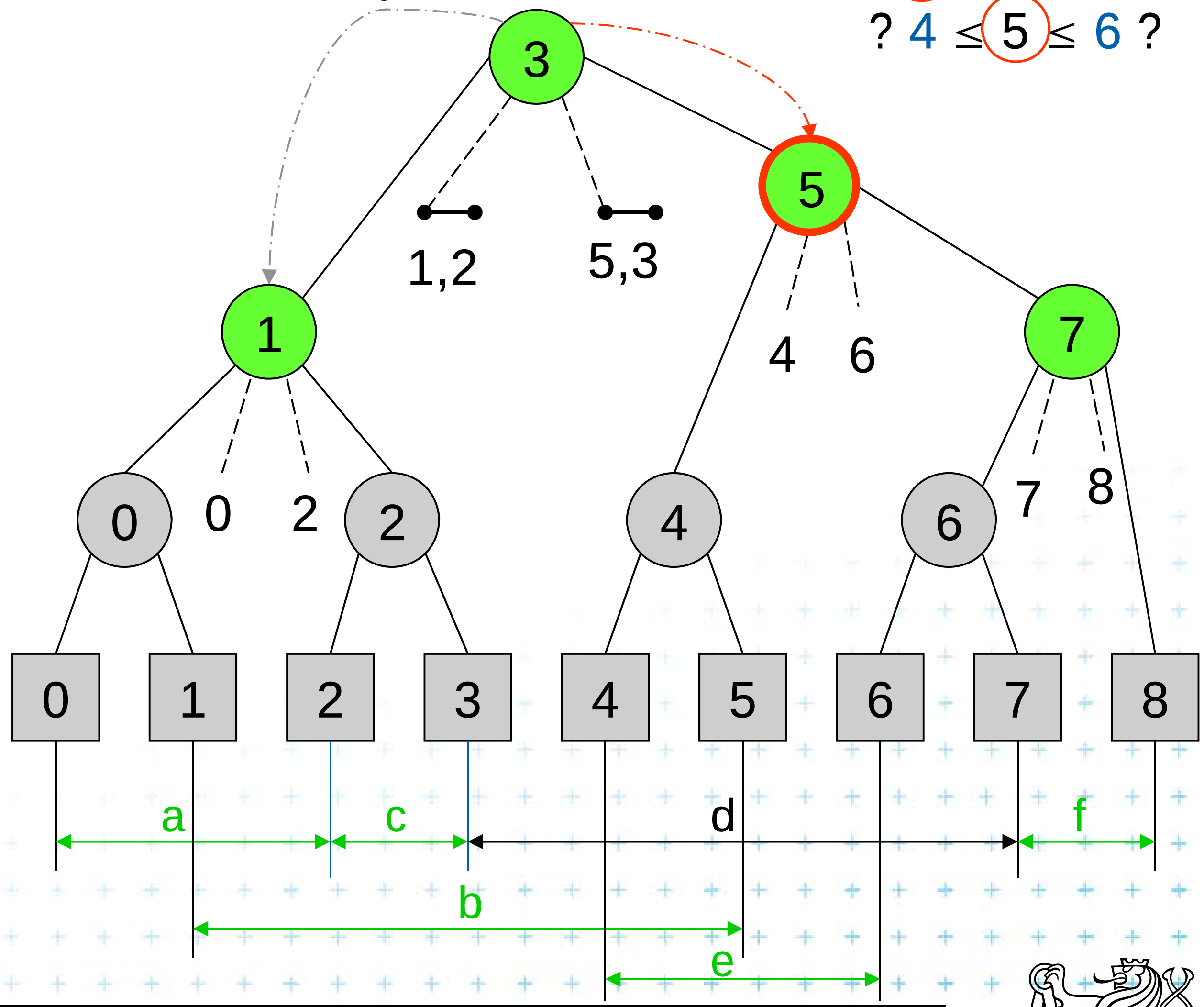
H(v) ≤ b < e

Insert the new interval to secondary lists

? 3 ≤ 4 < 6 ?

? 4 ≤ 5 ≤ 6 ?





- 🟩 Active rectangle
- ○ Current node
- ● Active node

[Drtina]

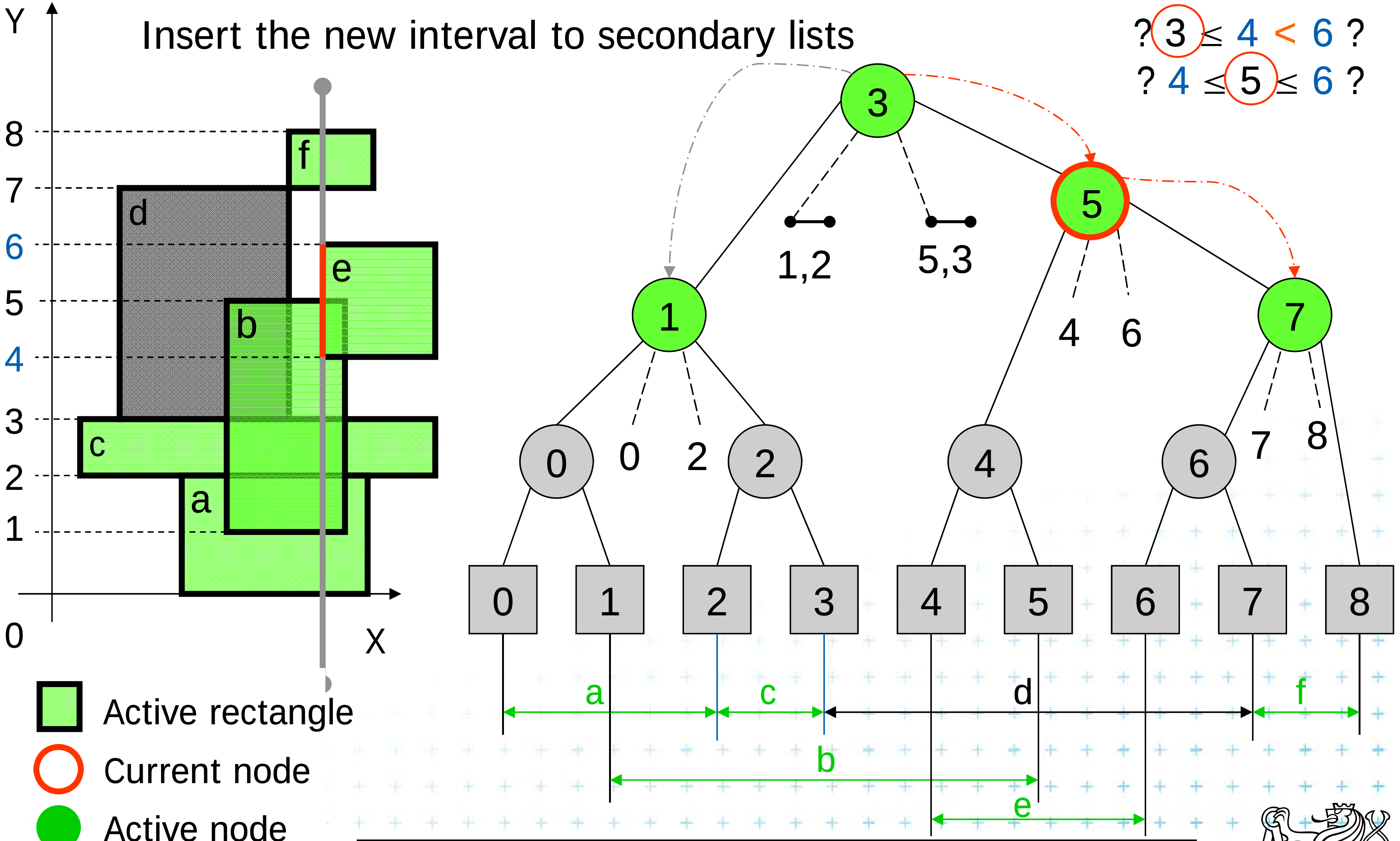
Insert [4,6] b) Insert Interval

H(v) ≤ b < e

Insert the new interval to secondary lists

? 3 ≤ 4 < 6 ?

? 4 ≤ 5 ≤ 6 ?



🟩 Active rectangle

○ Current node

● Active node

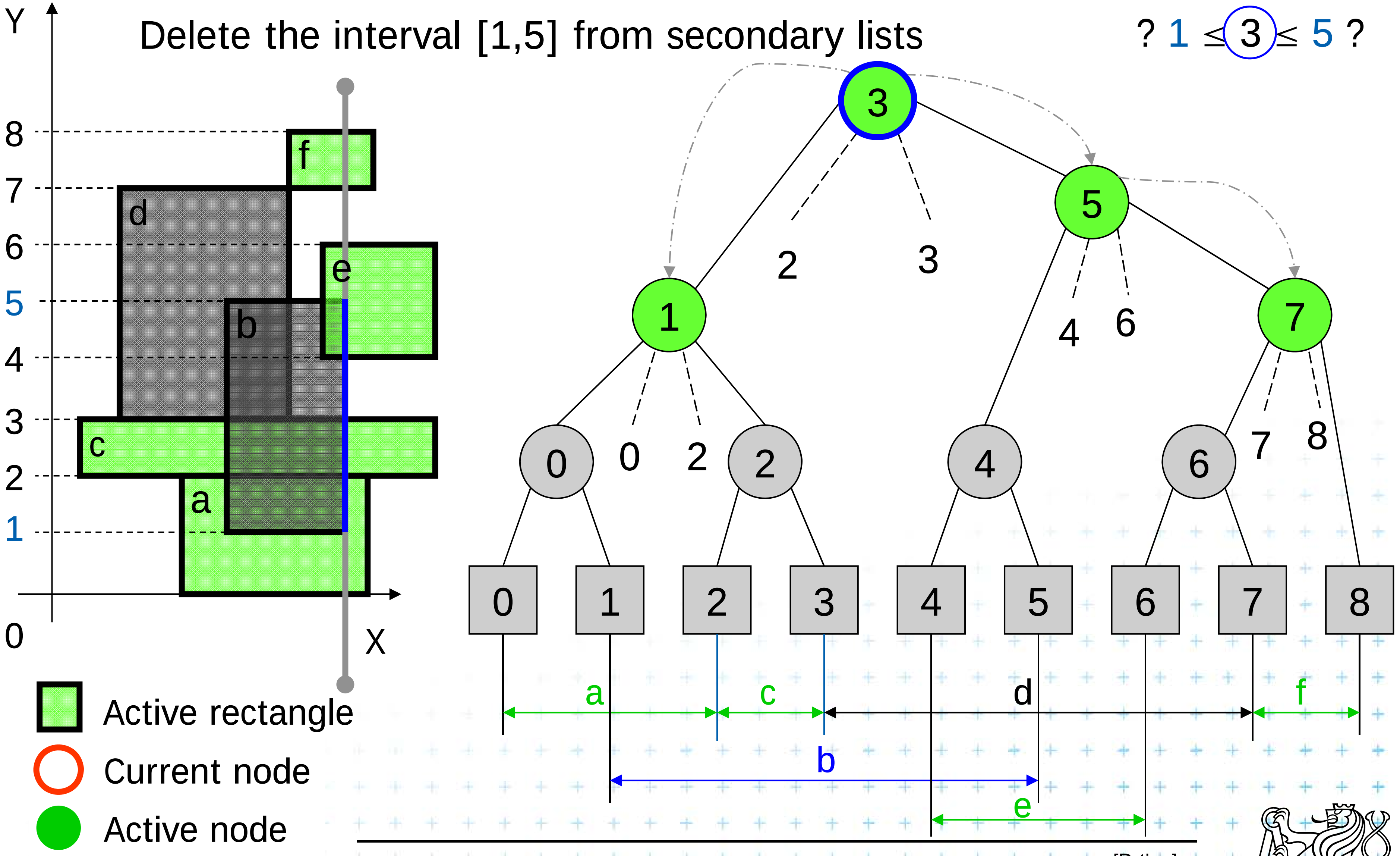
[Drtina]

Delete [1,5] Delete Interval

b ≤ H(v) ≤ e

Delete the interval [1,5] from secondary lists

? 1 ≤ 3 ≤ 5 ?



🟩 Active rectangle

○ Current node

● Active node

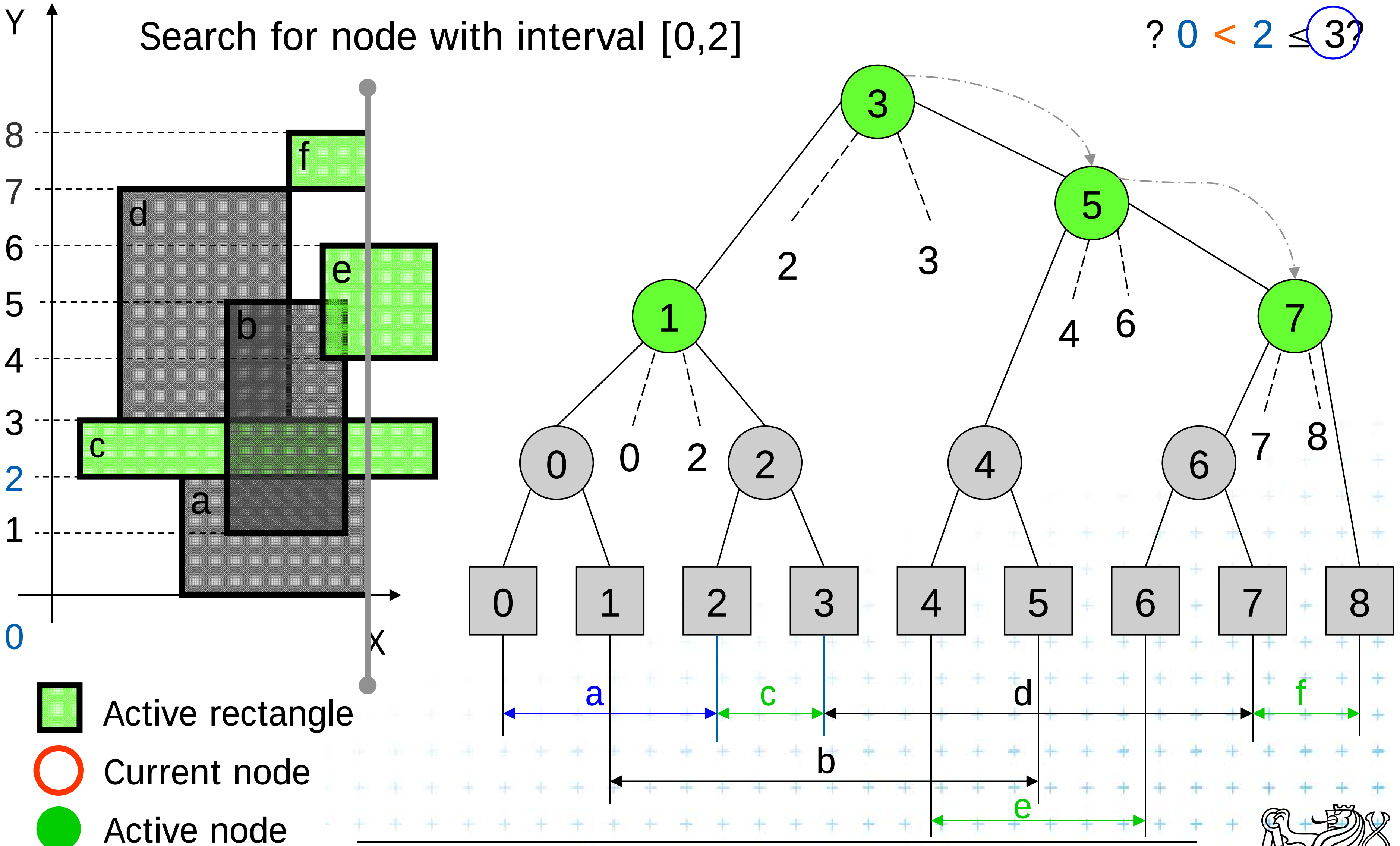
[Drtina]

Delete [0,2] Delete Interval 1/2

b < e ≤ H(v)

Search for node with interval [0,2]

? 0 < 2 ≤ 3?



🟩 Active rectangle

○ Current node

● Active node

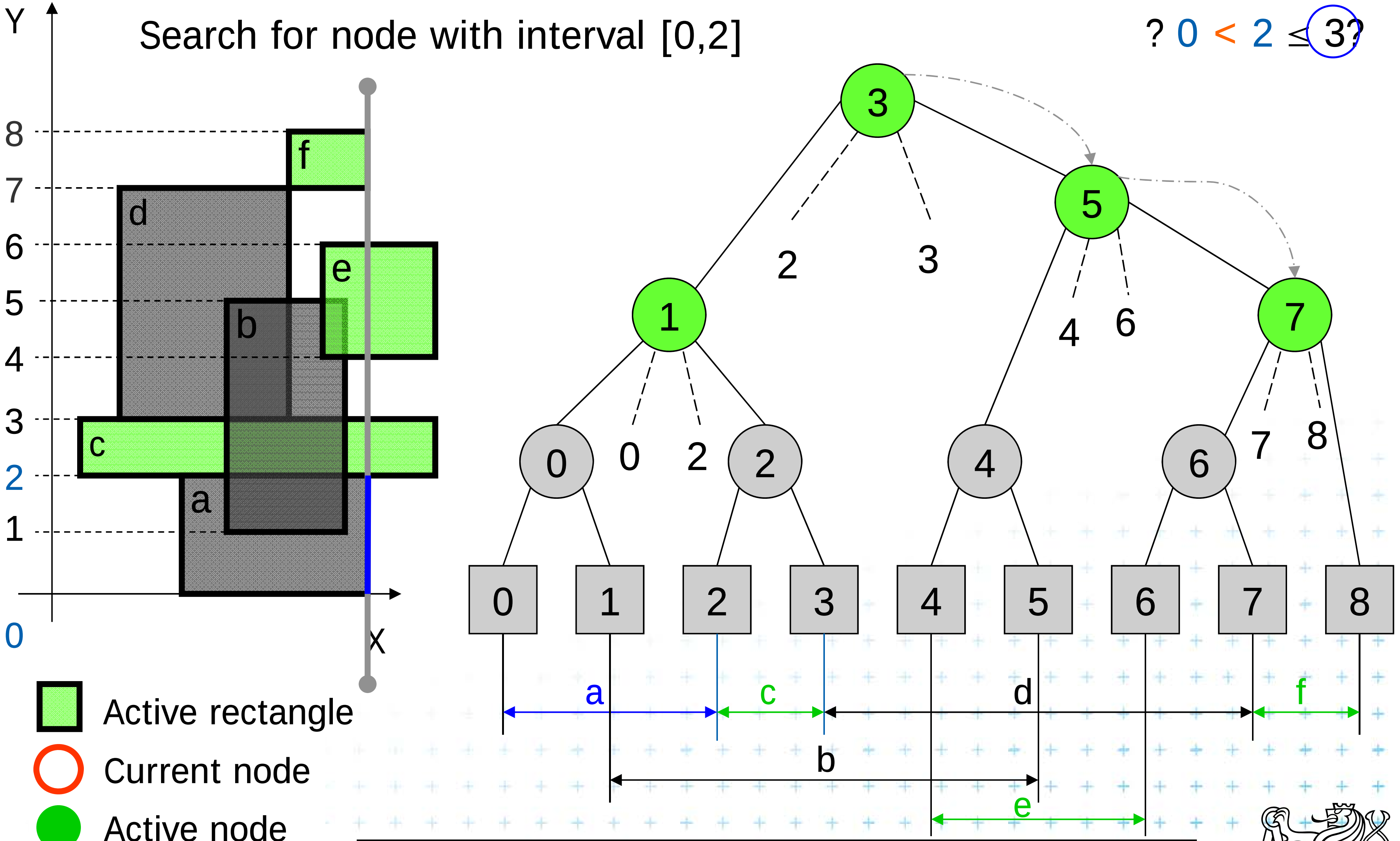
[Drtina]

Delete [0,2] Delete Interval 1/2

b < e ≤ H(v)

Search for node with interval [0,2]

? 0 < 2 ≤ 3?



🟩 Active rectangle

○ Current node

● Active node

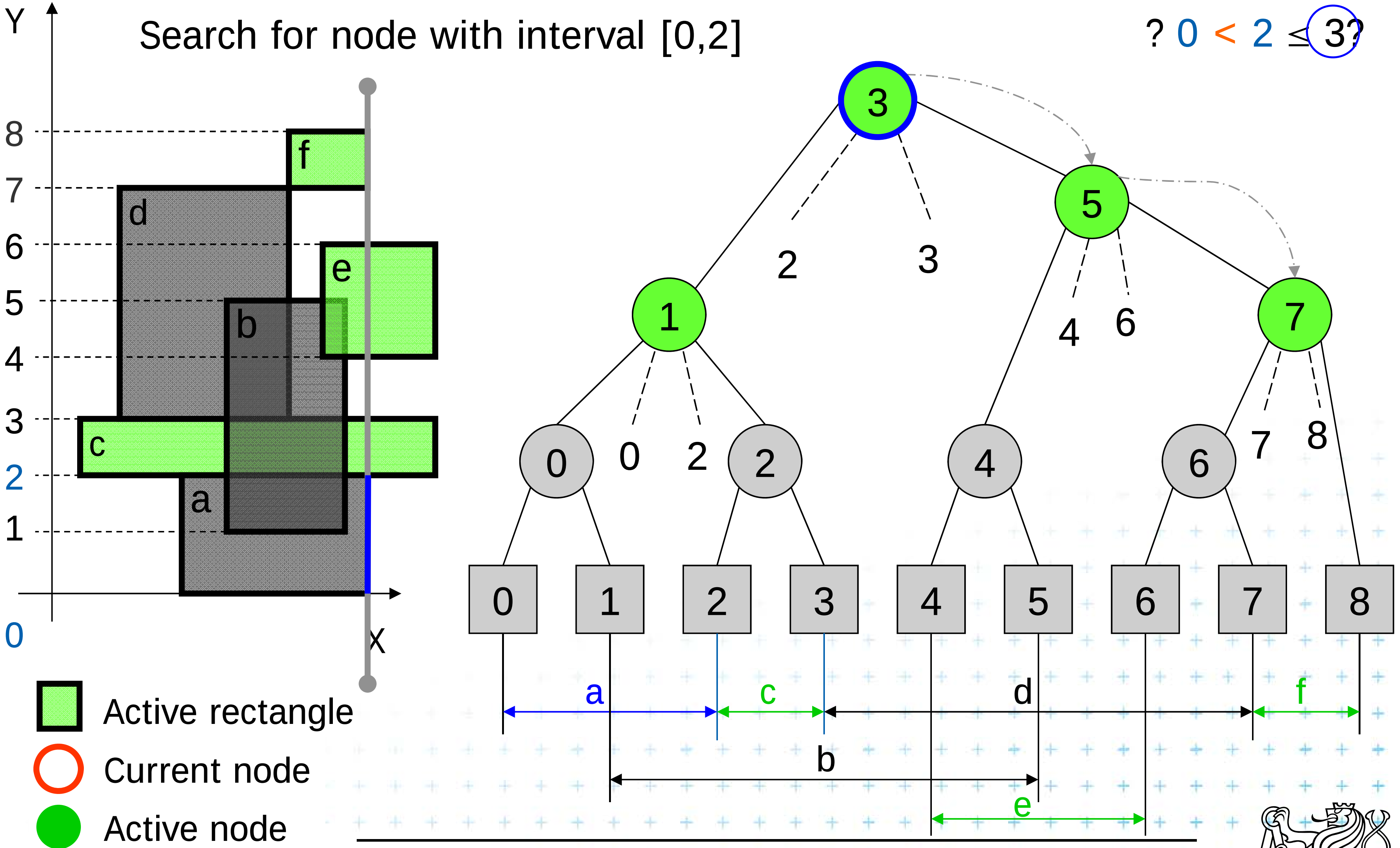
[Drtina]

Delete [0,2] Delete Interval 1/2

b < e ≤ H(v)

Search for node with interval [0,2]

? 0 < 2 ≤ 3?



- 🟩 Active rectangle
- ○ Current node
- ● Active node

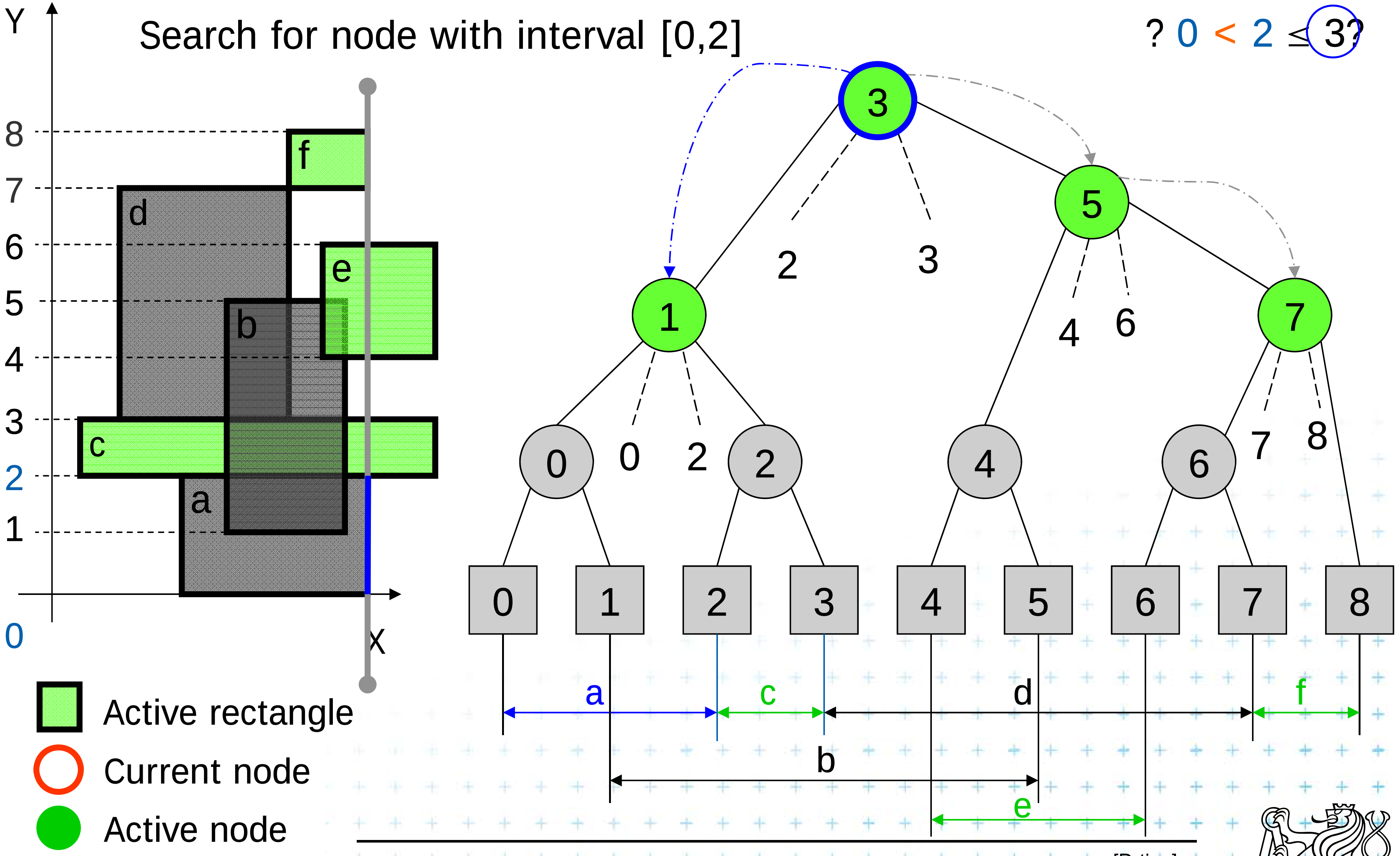
[Drtina]

Delete [0,2] Delete Interval 1/2

b < e ≤ H(v)

Search for node with interval [0,2]

? 0 < 2 ≤ 3?



🟩 Active rectangle

○ Current node

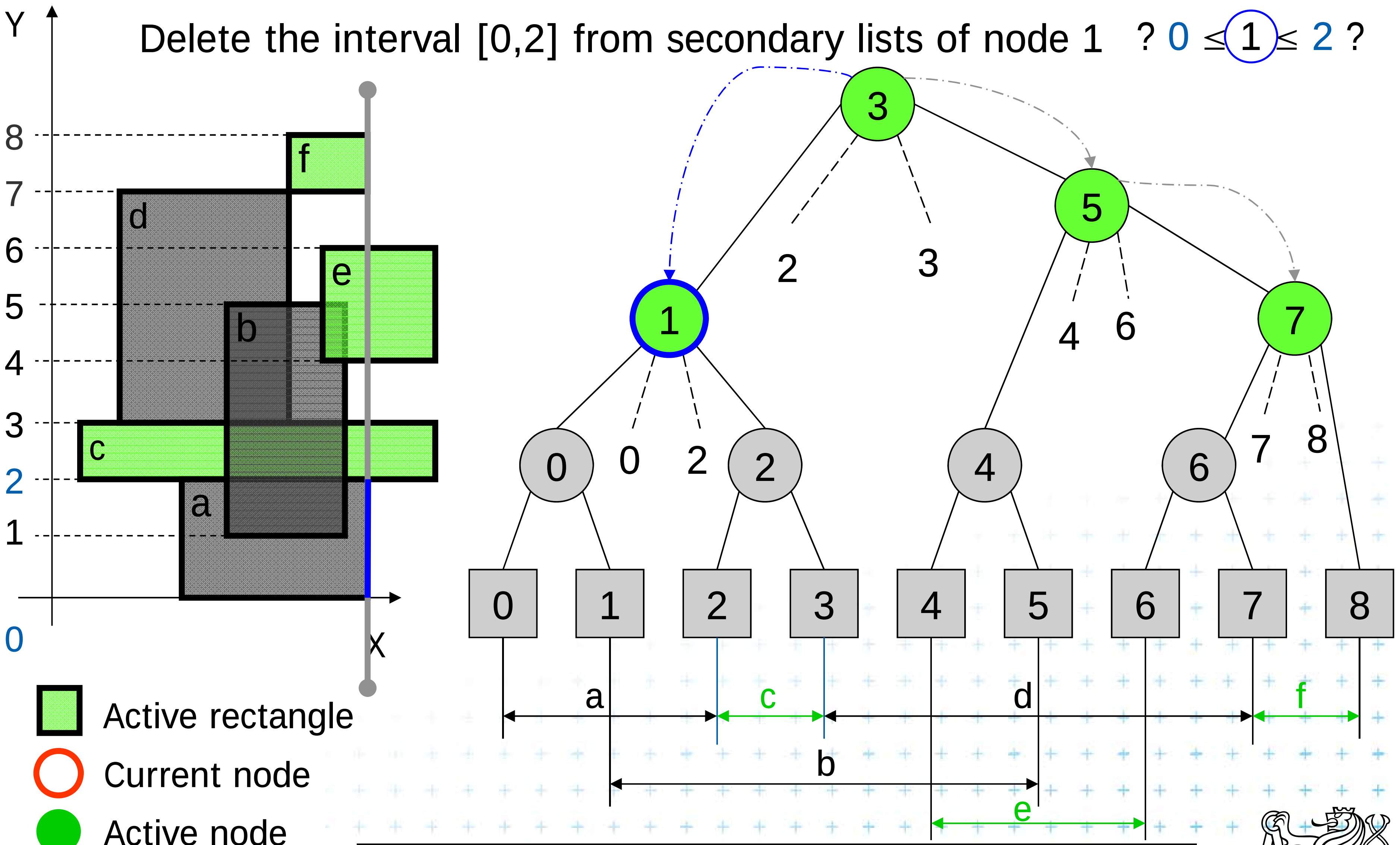
● Active node

[Drtina]

Delete [0,2] Delete Interval 2/2

b ≤ H(v) ≤ e

Delete the interval [0,2] from secondary lists of node 1 ? 0 ≤ 1 ≤ 2 ?



🟩 Active rectangle

○ Current node

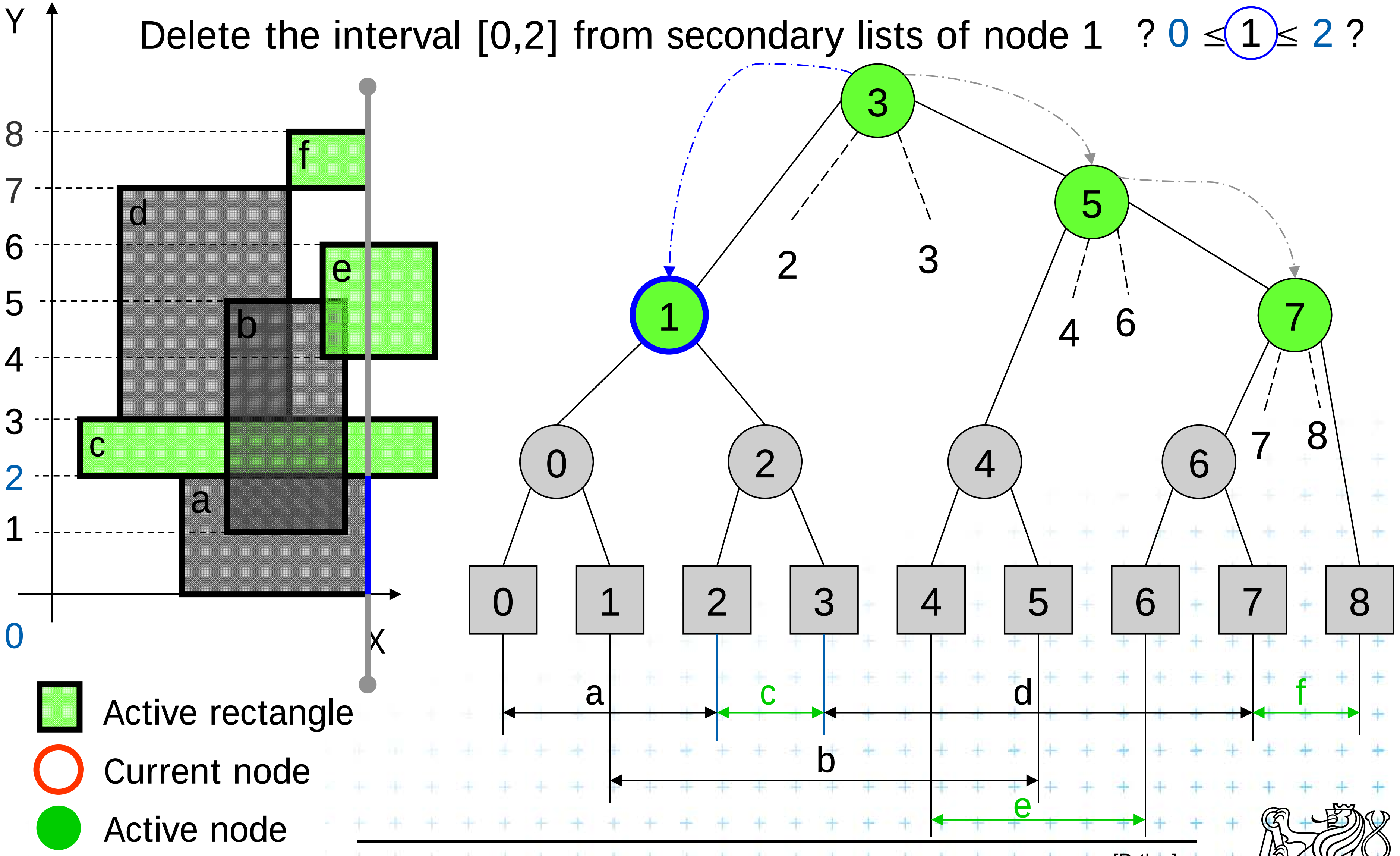
● Active node

[Drtina]

Delete [0,2] Delete Interval 2/2

b ≤ H(v) ≤ e

Delete the interval [0,2] from secondary lists of node 1 ? 0 ≤ 1 ≤ 2 ?



🟩 Active rectangle

○ Current node

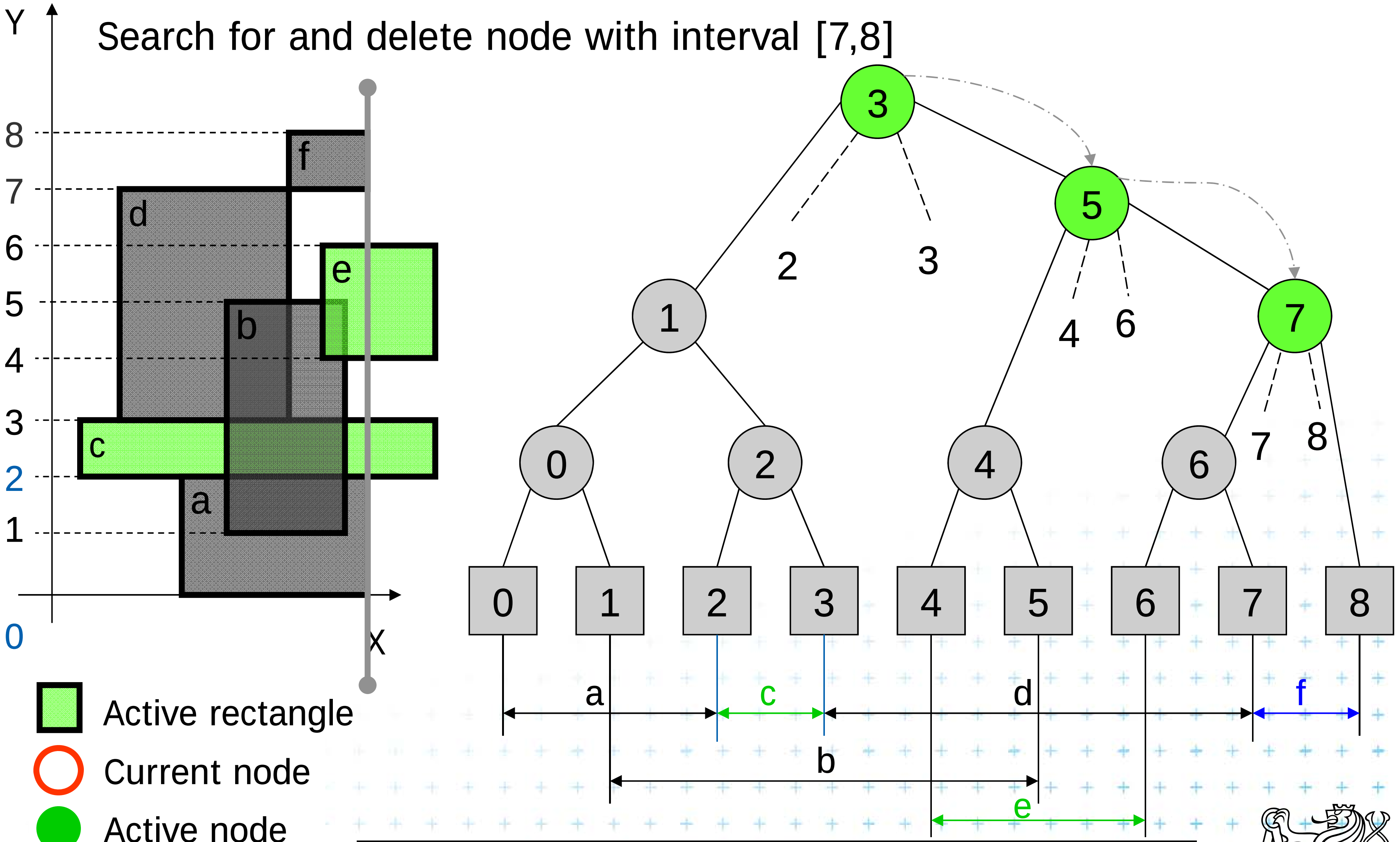
● Active node

[Drtina]

Delete [7,8] Delete Interval

b ≤ H(v) ≤ e

Search for and delete node with interval [7,8]



🟩 Active rectangle

○ Current node

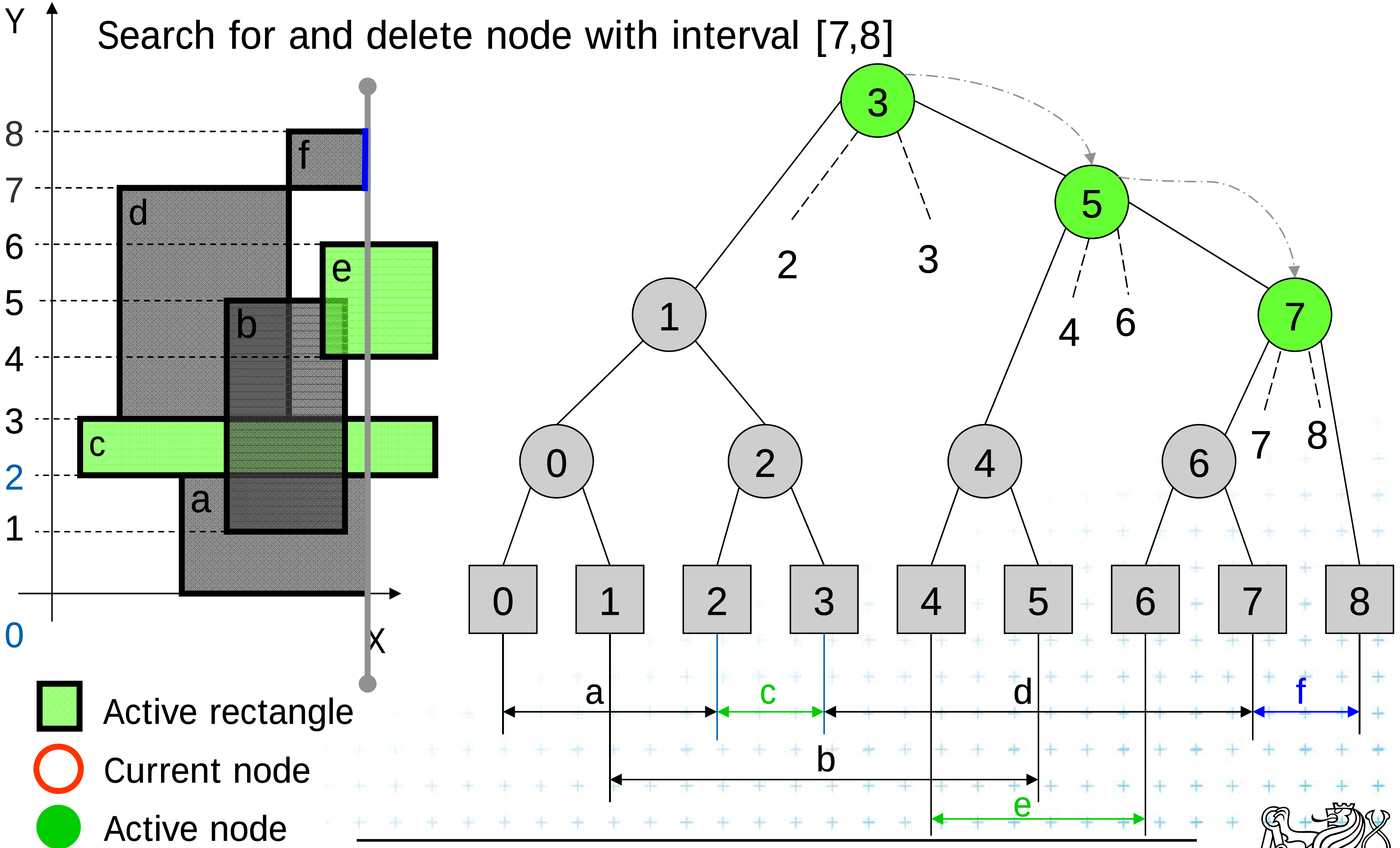
● Active node

[Drtina]

Delete [7,8] Delete Interval

b ≤ H(v) ≤ e

Search for and delete node with interval [7,8]



🟩 Active rectangle

○ Current node

● Active node

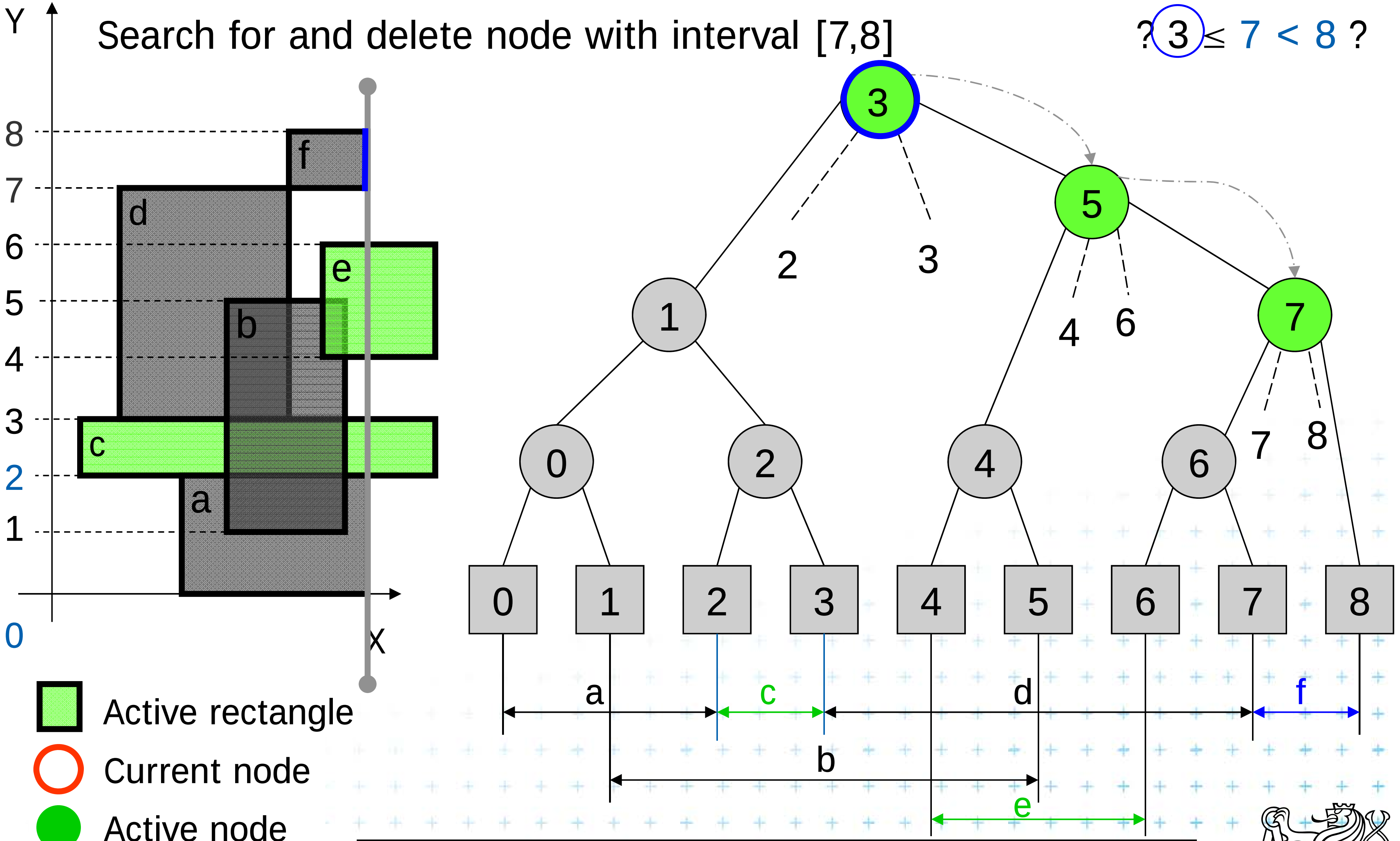
[Drtina]

Delete [7,8] Delete Interval

b ≤ H(v) ≤ e

Search for and delete node with interval [7,8]

? 3 ≤ 7 < 8 ?



🟩 Active rectangle

○ Current node

● Active node

[Drtina]

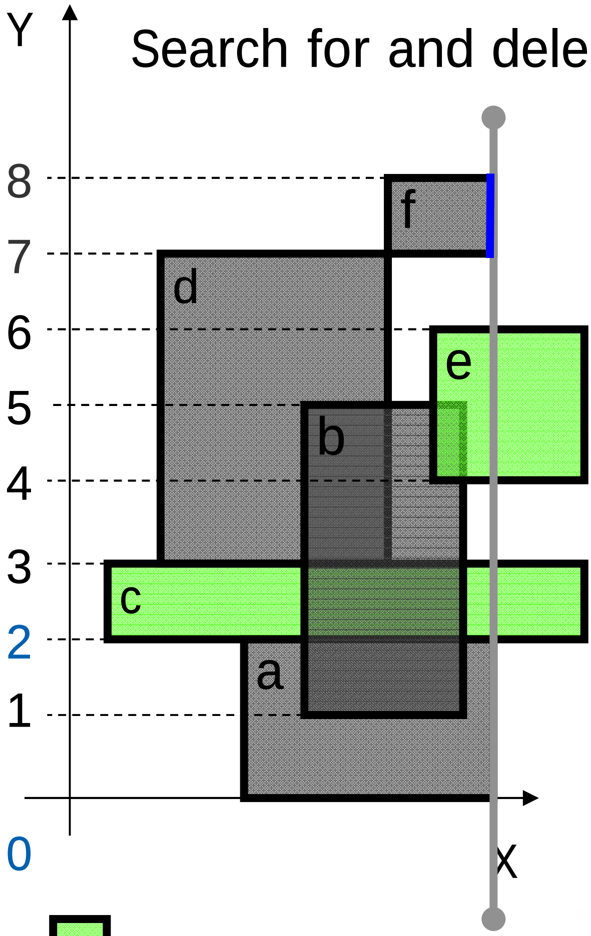
Delete [7,8] Delete Interval

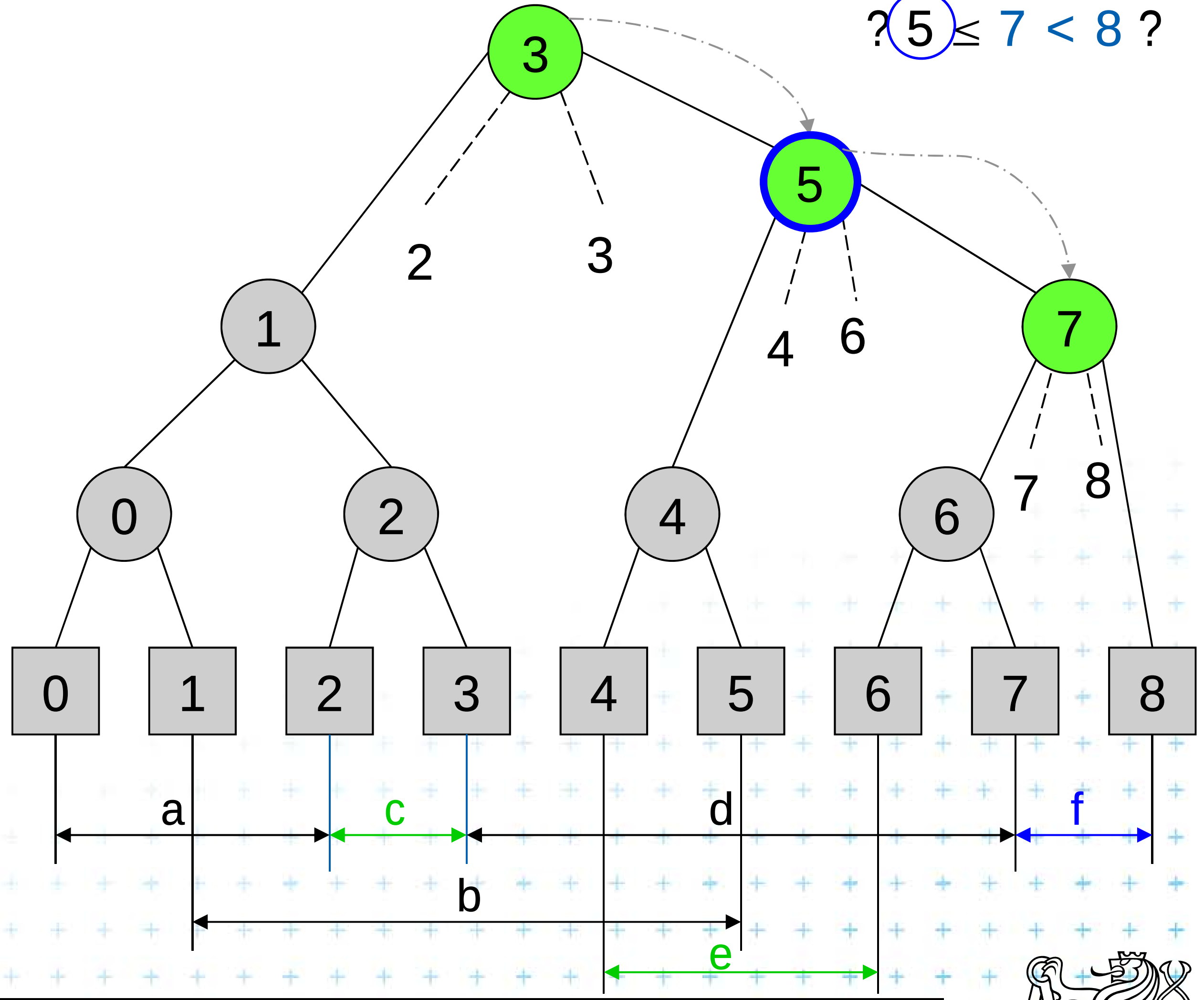
b ≤ H(v) ≤ e

Search for and delete node with interval [7,8]

? 3 ≤ 7 < 8 ?

? 5 ≤ 7 < 8 ?





🟩 Active rectangle

○ Current node

● Active node

[Drtina]

Delete [7,8] Delete Interval

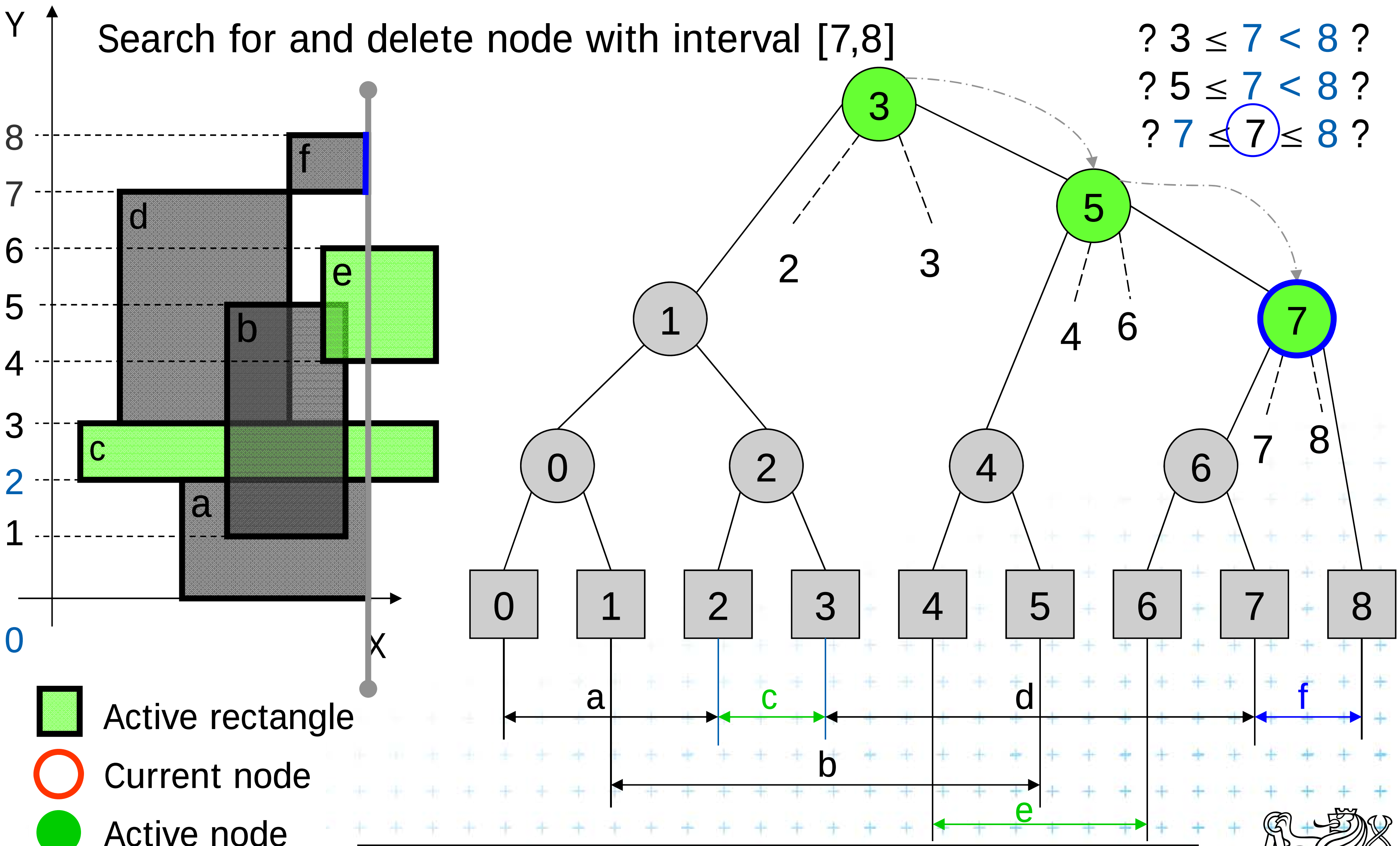
b ≤ H(v) ≤ e

Search for and delete node with interval [7,8]

? 3 ≤ 7 < 8 ?

? 5 ≤ 7 < 8 ?

? 7 ≤ 7 ≤ 8 ?



🟩 Active rectangle

○ Current node

● Active node

[Drtina]

Delete [7,8] Delete Interval

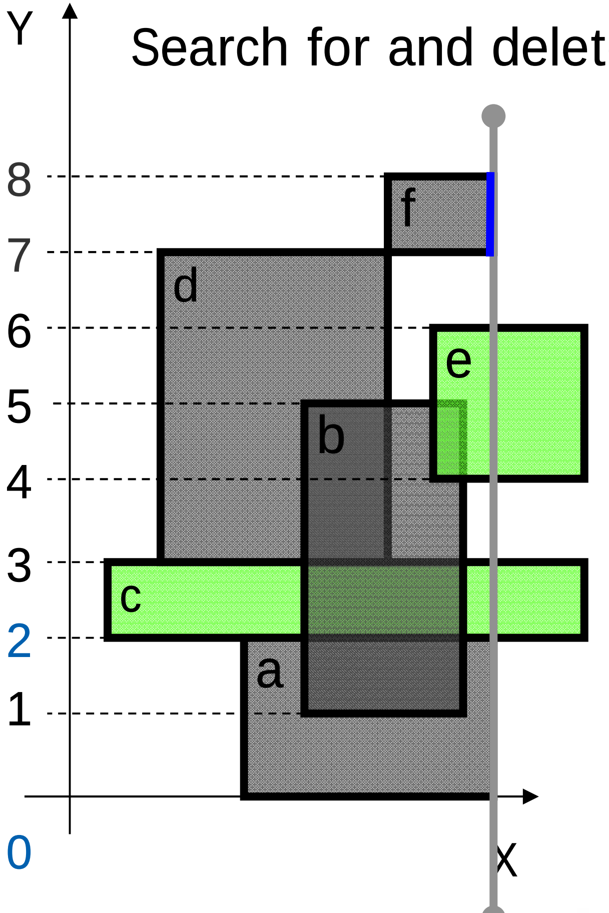
b ≤ H(v) ≤ e

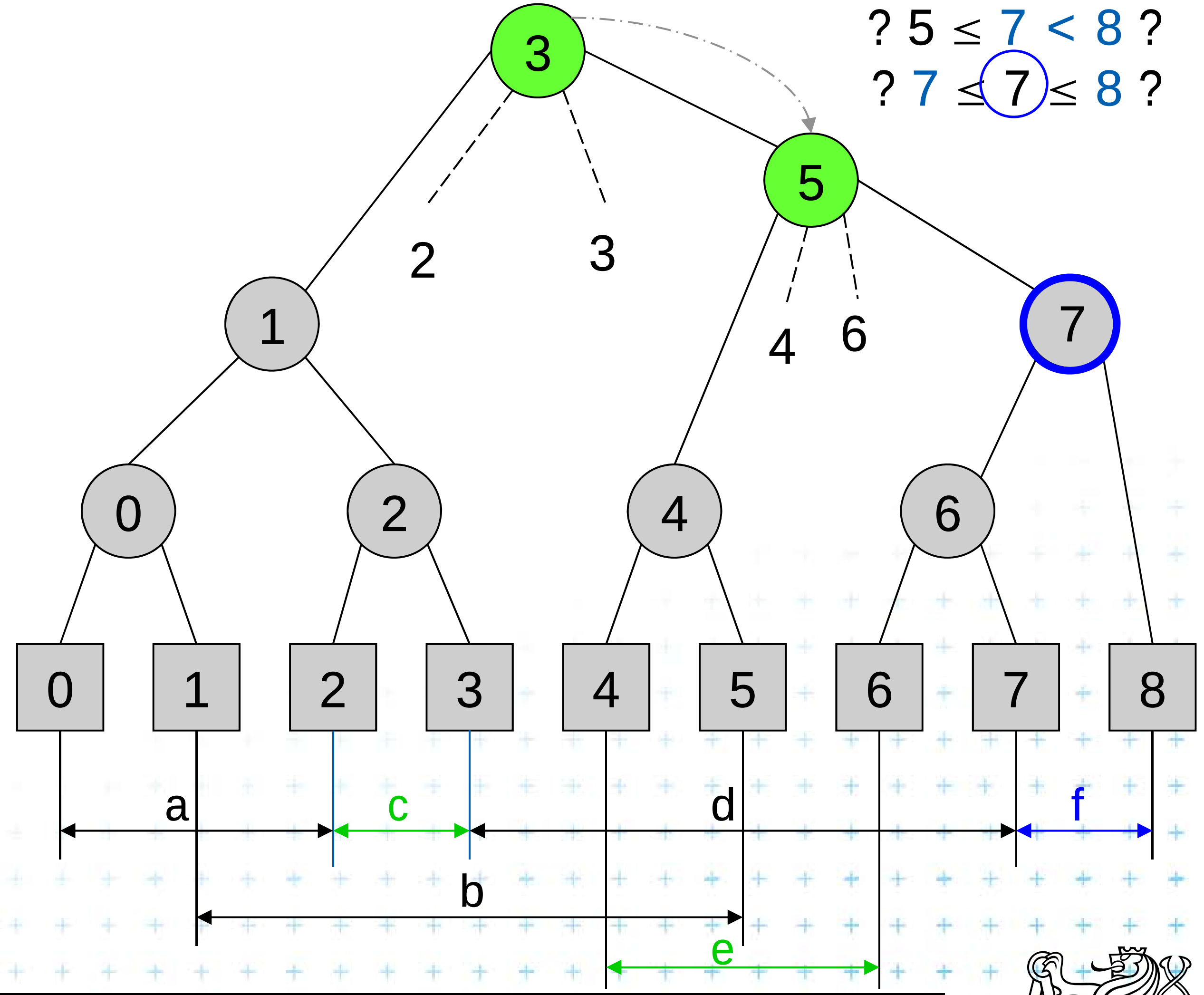
Search for and delete node with interval [7,8]

? 3 ≤ 7 < 8 ?

? 5 ≤ 7 < 8 ?

? 7 ≤ 7 ≤ 8 ?





🟩 Active rectangle

○ Current node

● Active node

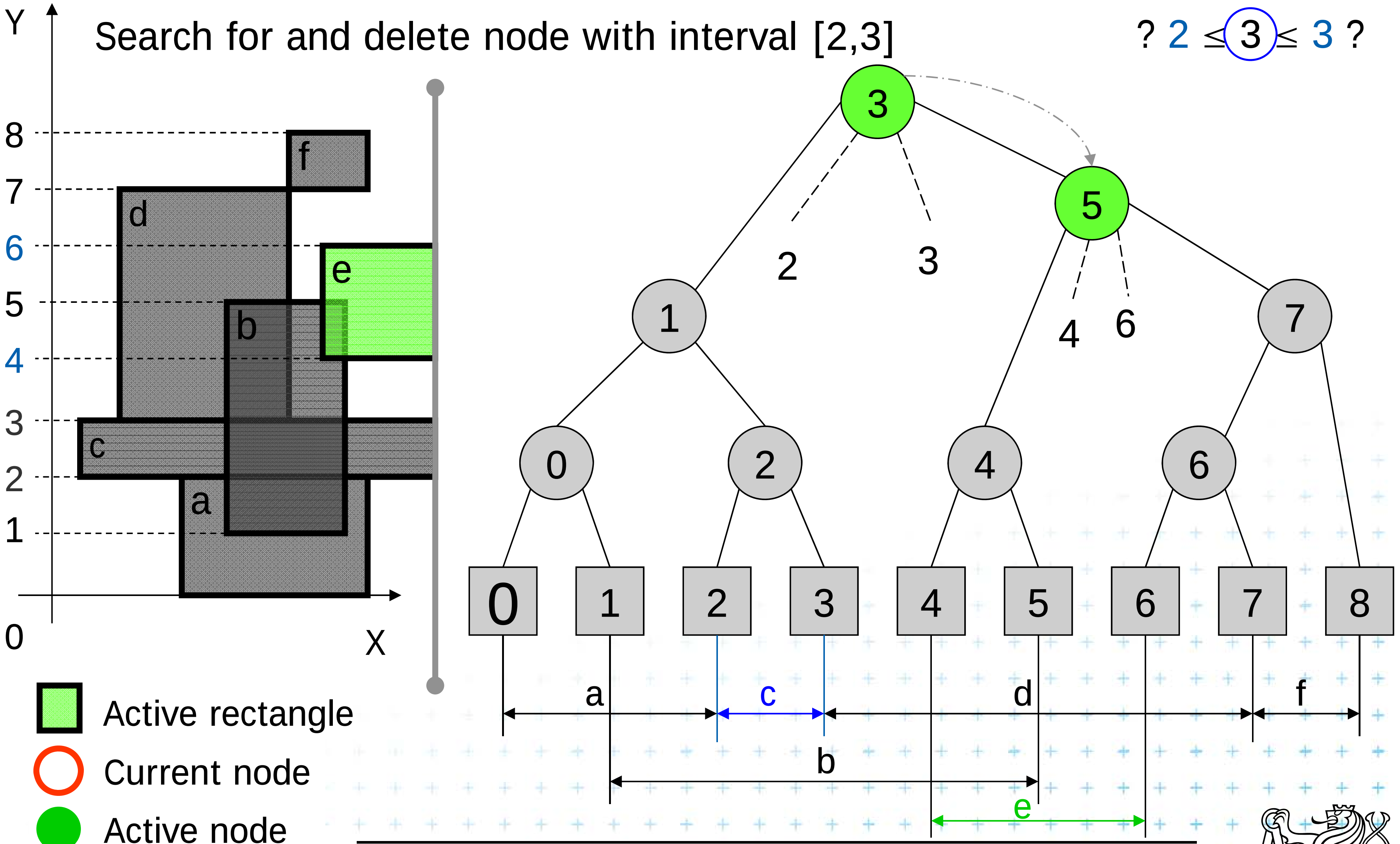
[Drtina]

Delete [2,3] Delete Interval

b ≤ H(v) ≤ e

Search for and delete node with interval [2,3]

? 2 ≤ 3 ≤ 3 ?



🟩 Active rectangle

○ Current node

● Active node

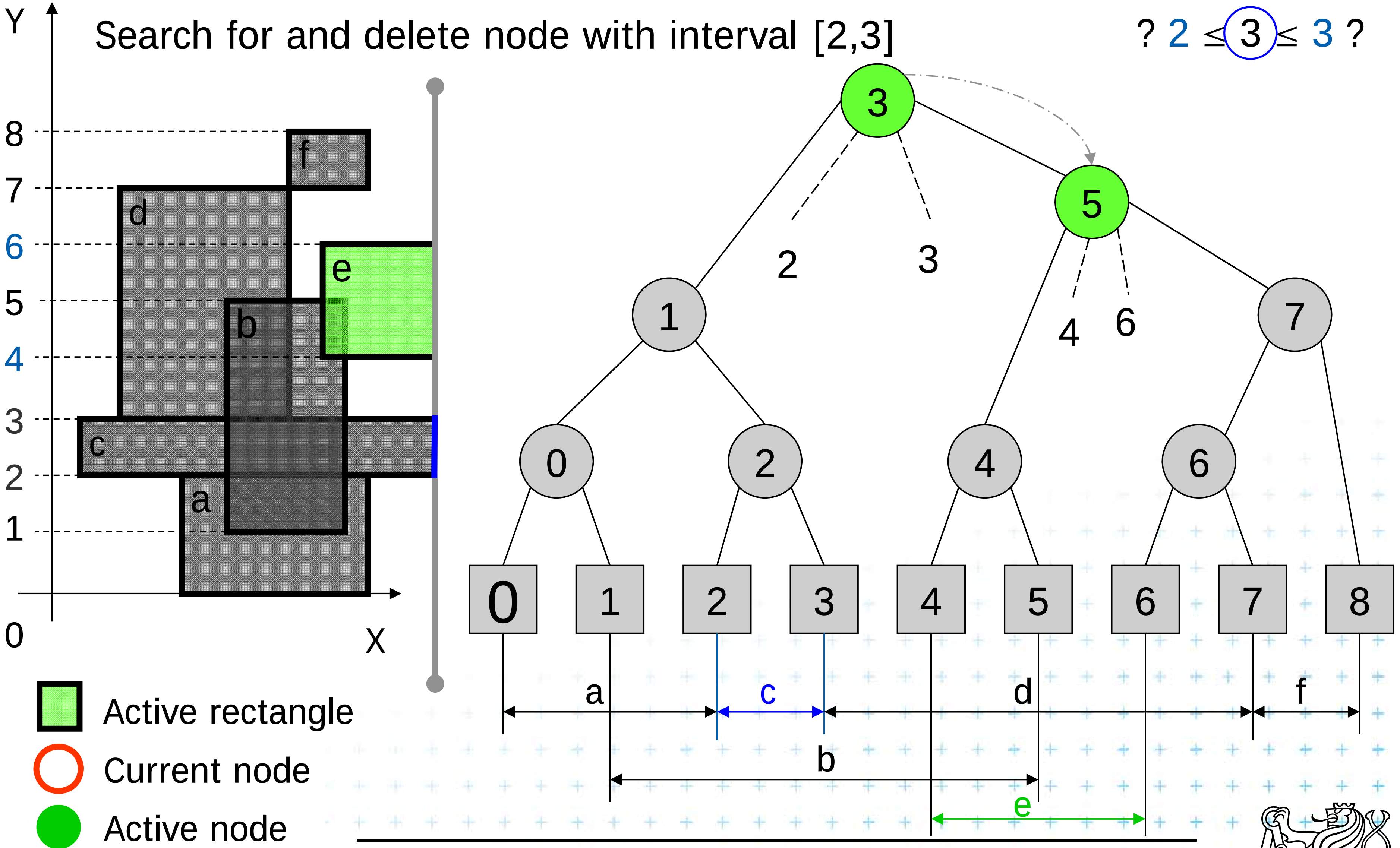
[Drtina]

Delete [2,3] Delete Interval

b ≤ H(v) ≤ e

Search for and delete node with interval [2,3]

? 2 ≤ 3 ≤ 3 ?



■ Active rectangle

○ Current node

● Active node

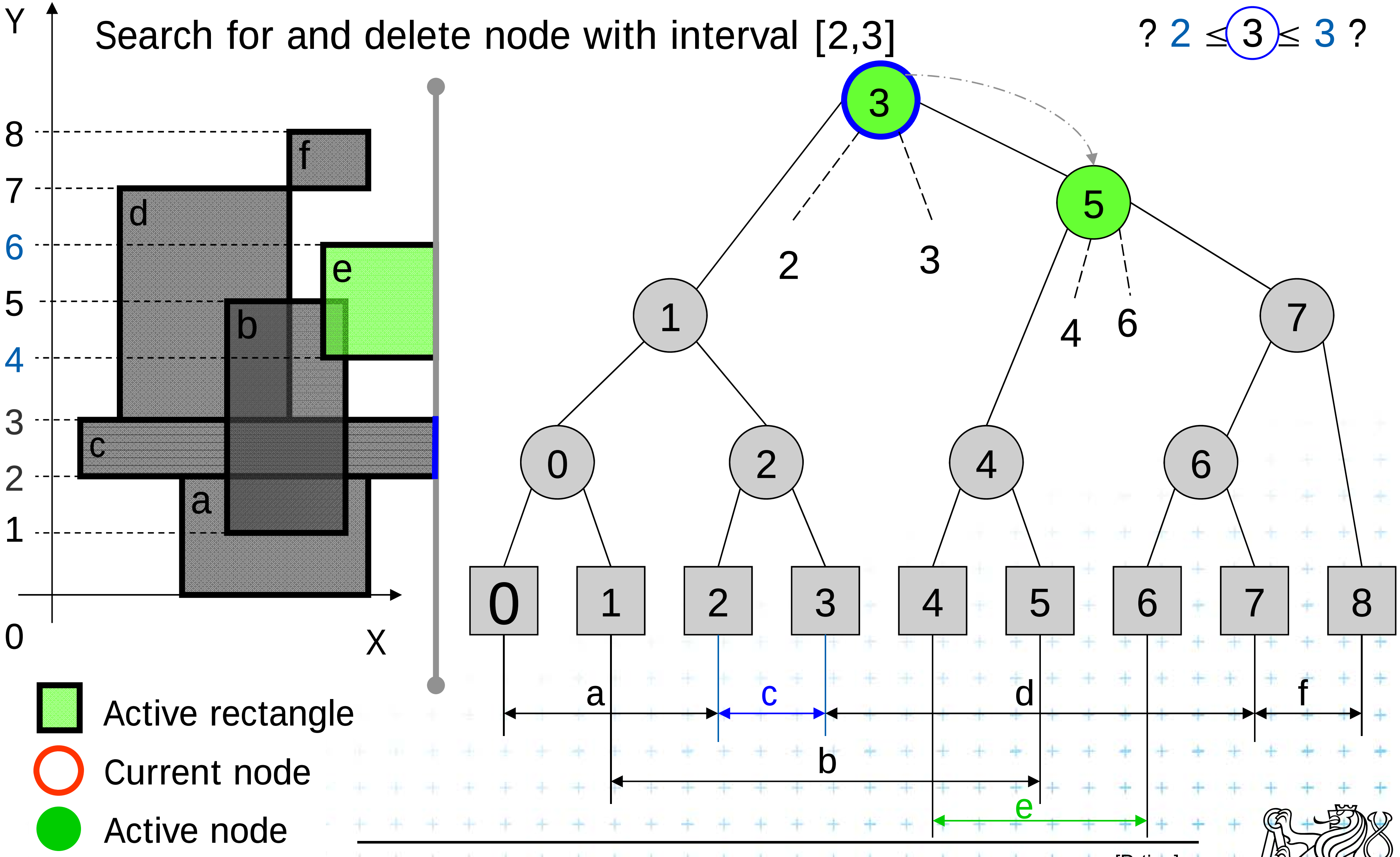
[Drtina]

Delete [2,3] Delete Interval

b ≤ H(v) ≤ e

Search for and delete node with interval [2,3]

? 2 ≤ 3 ≤ 3 ?



🟩 Active rectangle

○ Current node

● Active node

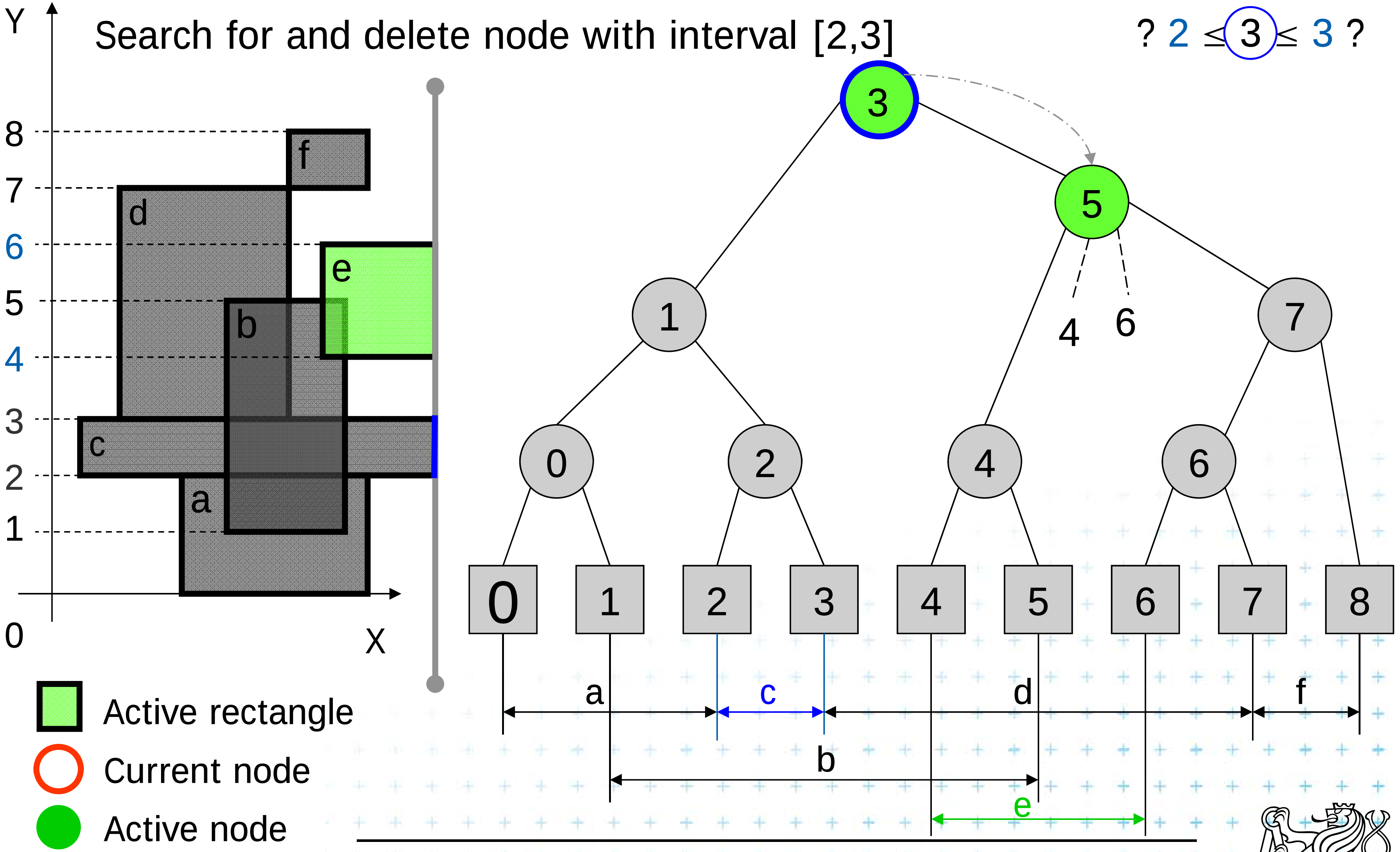
[Drtina]

Delete [2,3] Delete Interval

b ≤ H(v) ≤ e

Search for and delete node with interval [2,3]

? 2 ≤ 3 ≤ 3 ?



🟩 Active rectangle

○ Current node

● Active node

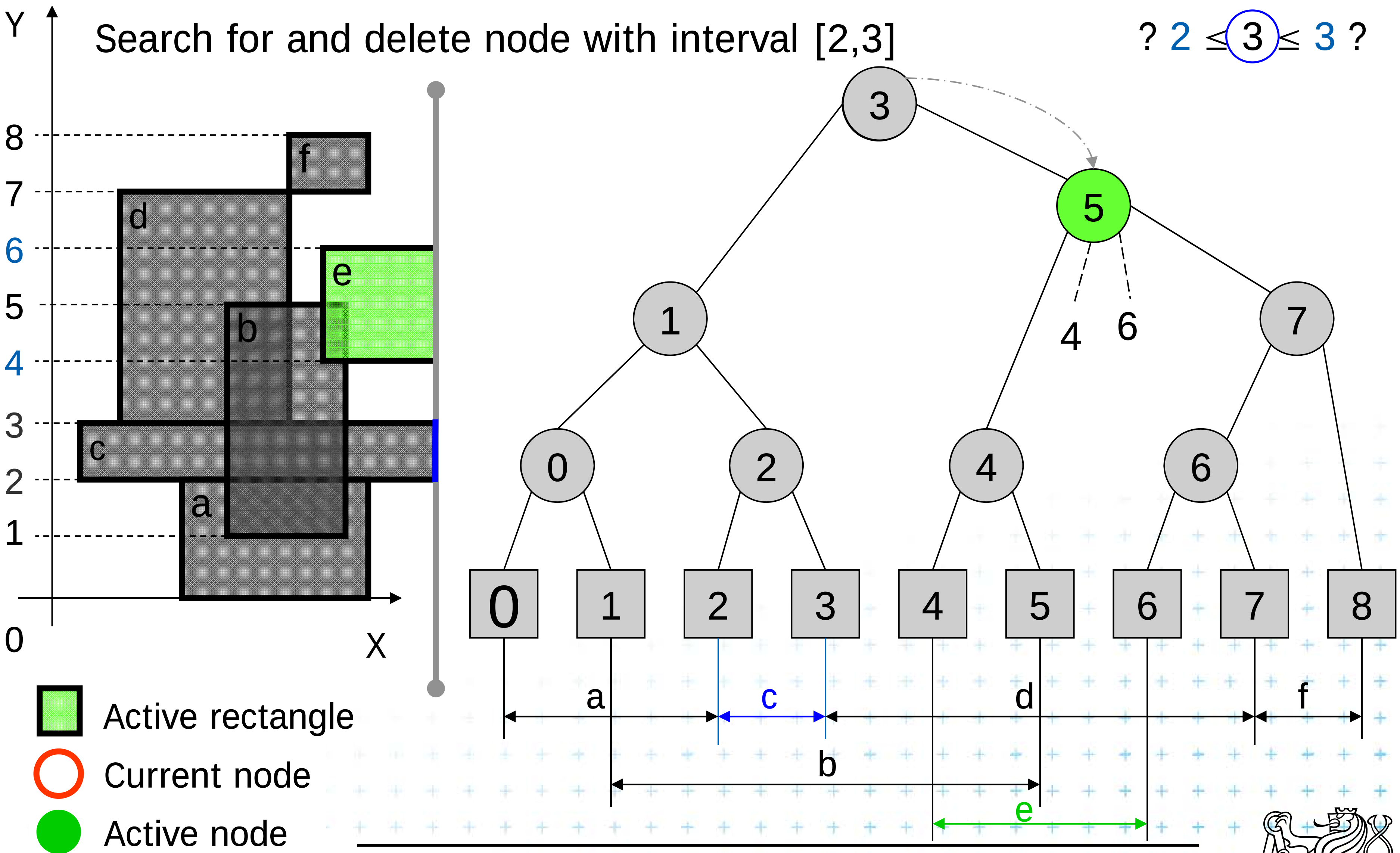
[Drtina]

Delete [2,3] Delete Interval

b ≤ H(v) ≤ e

Search for and delete node with interval [2,3]

? 2 ≤ 3 ≤ 3 ?



🟩 Active rectangle

○ Current node

● Active node

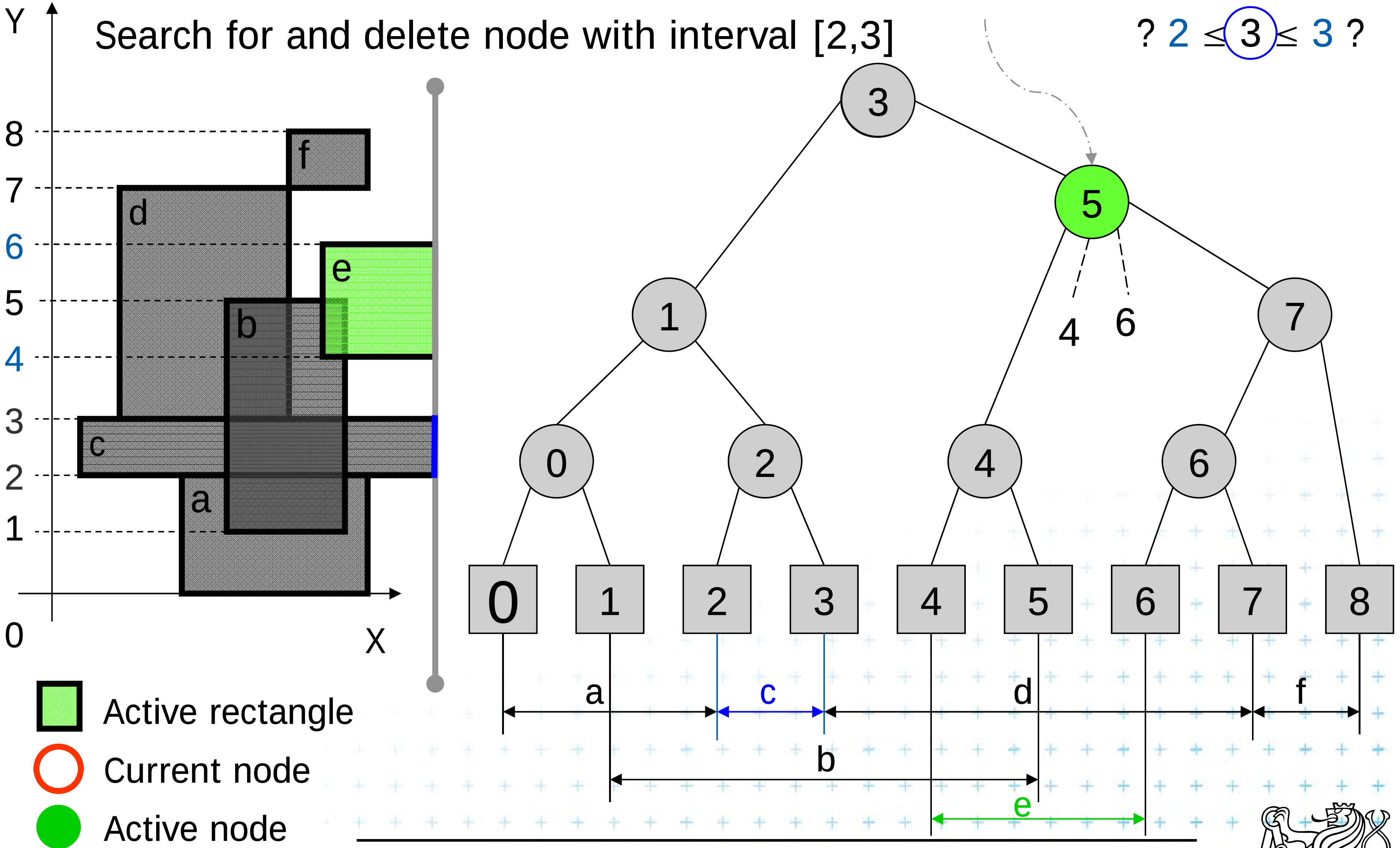
[Drtina]

Delete [2,3] Delete Interval

b ≤ H(v) ≤ e

Search for and delete node with interval [2,3]

? 2 ≤ 3 ≤ 3 ?



🟩 Active rectangle

○ Current node

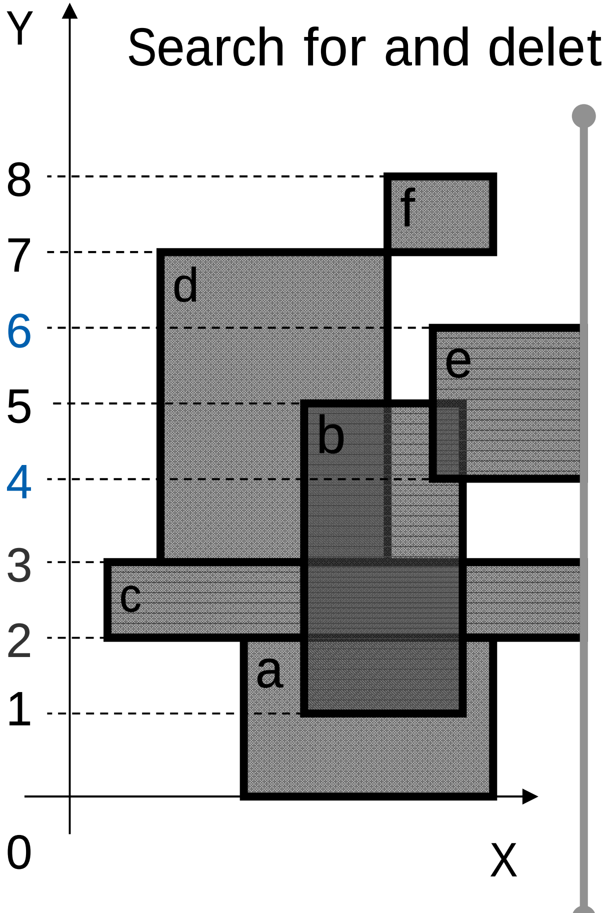
● Active node

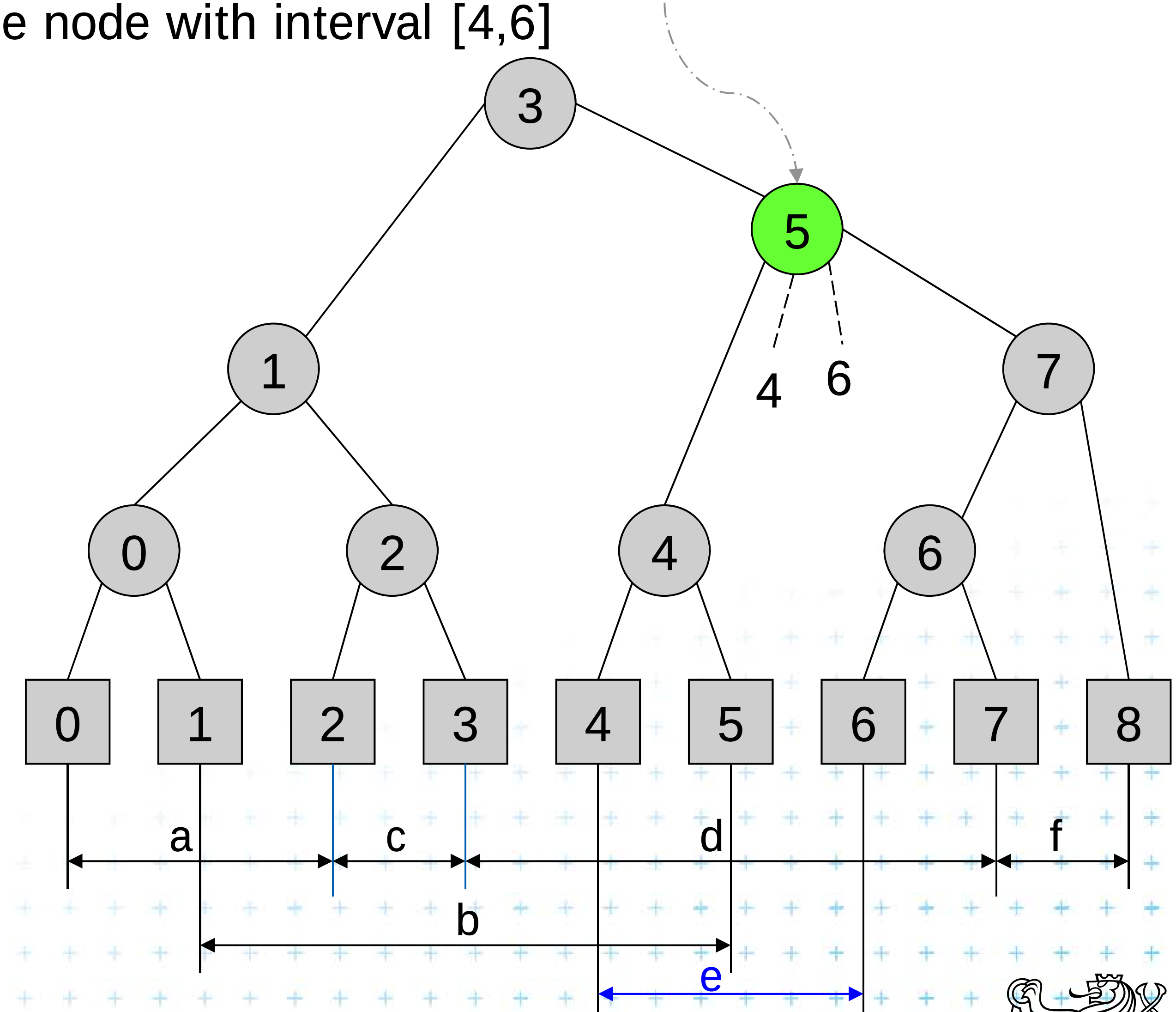
[Drtina]

Delete [4,6] Delete Interval

$b \leq H(v) \leq e$

Search for and delete node with interval [4,6]





🟩 Active rectangle

○ Current node

● Active node

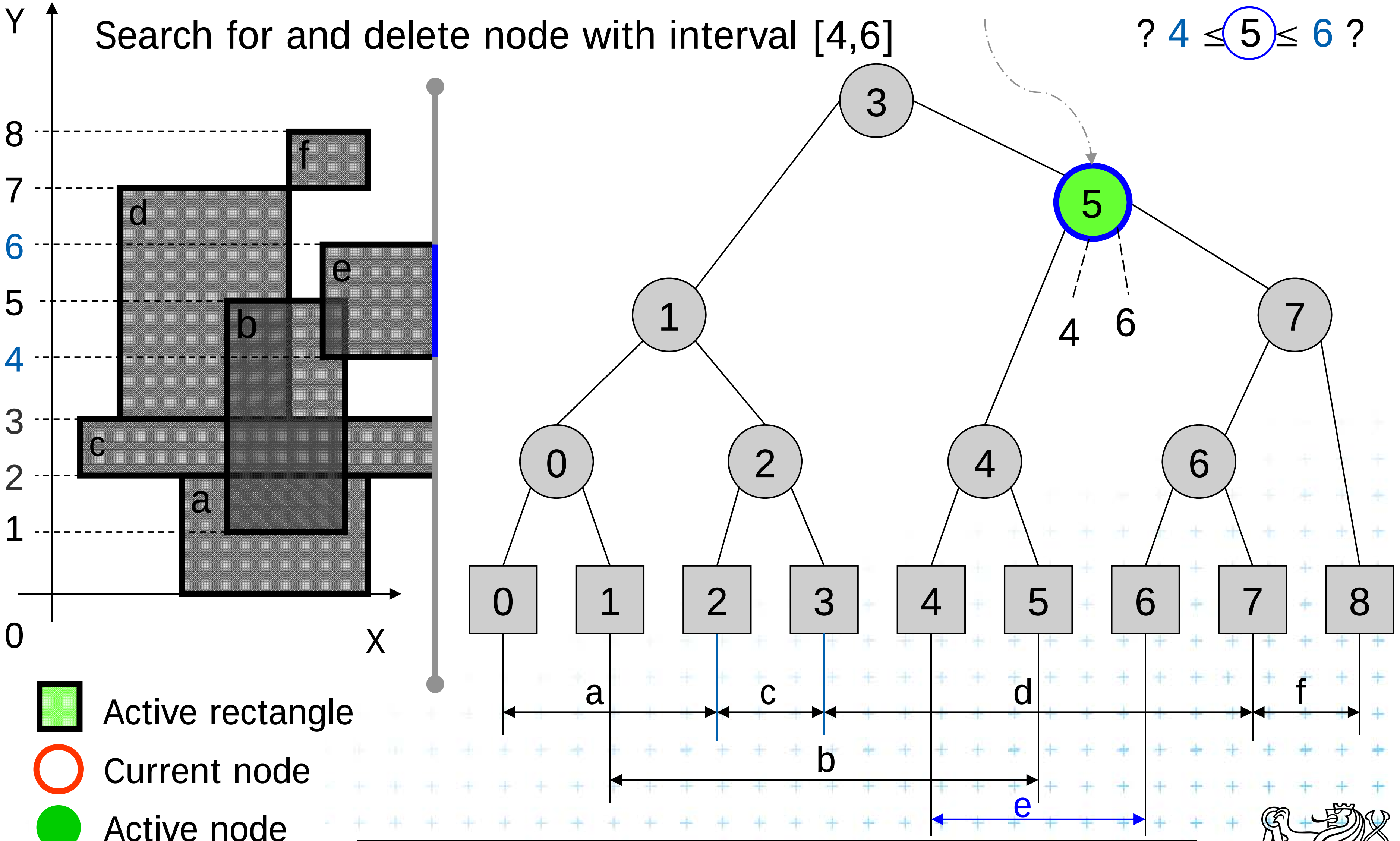
[Drtina]

Delete [4,6] Delete Interval

$b \le H(v) \le e$

Search for and delete node with interval [4,6]

? 4 ≤ 5 ≤ 6 ?



🟩 Active rectangle

○ Current node

● Active node

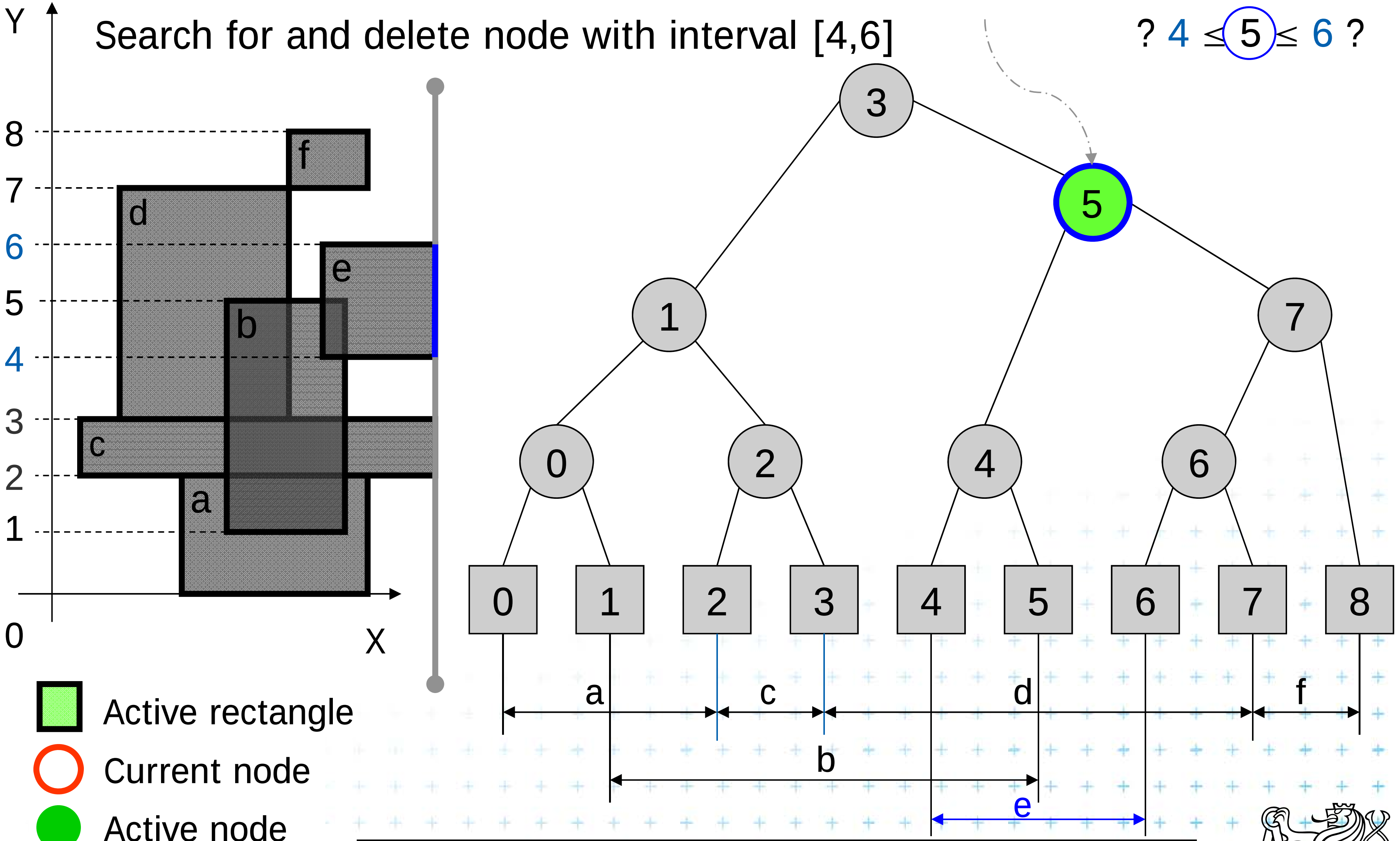
[Drtina]

Delete [4,6] Delete Interval

b ≤ H(v) ≤ e

Search for and delete node with interval [4,6]

? 4 ≤ 5 ≤ 6 ?



- 🟩 Active rectangle
- ○ Current node
- ● Active node

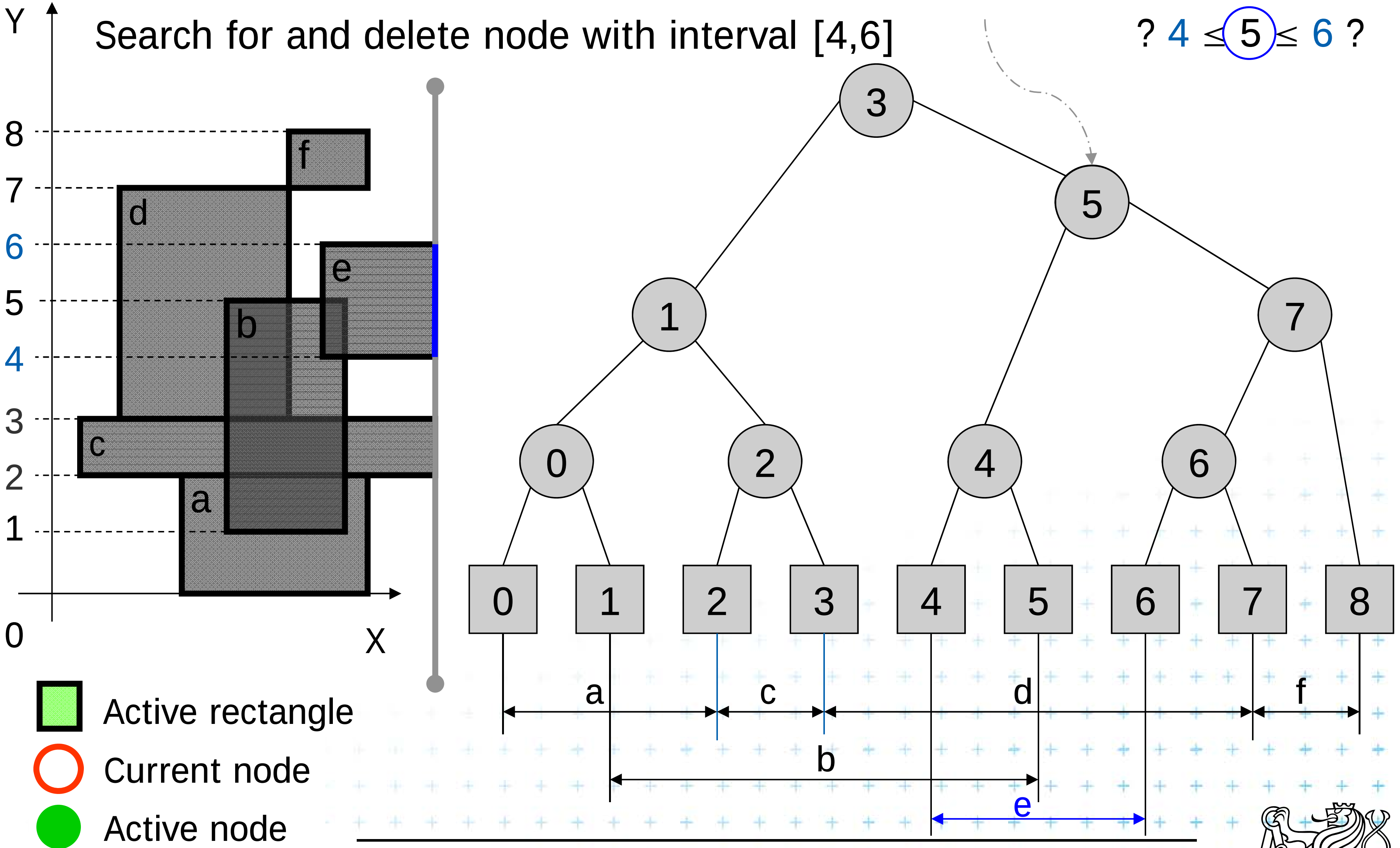
[Drtina]

Delete [4,6] Delete Interval

b ≤ H(v) ≤ e

Search for and delete node with interval [4,6]

? 4 ≤ 5 ≤ 6 ?



🟩 Active rectangle

○ Current node

● Active node

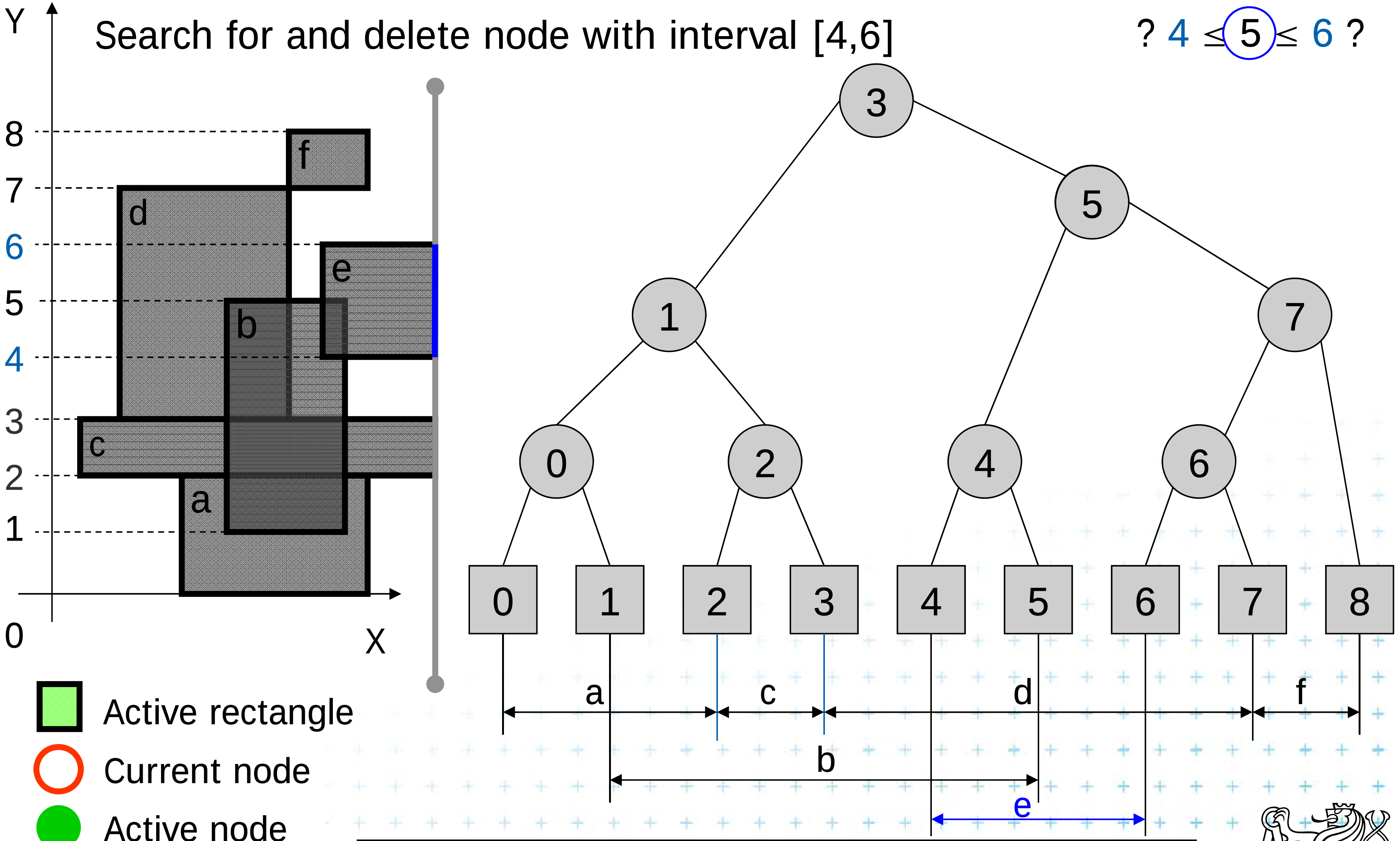
[Drtina]

Delete [4,6] Delete Interval

b ≤ H(v) ≤ e

Search for and delete node with interval [4,6]

? 4 ≤ 5 ≤ 6 ?



🟩 Active rectangle

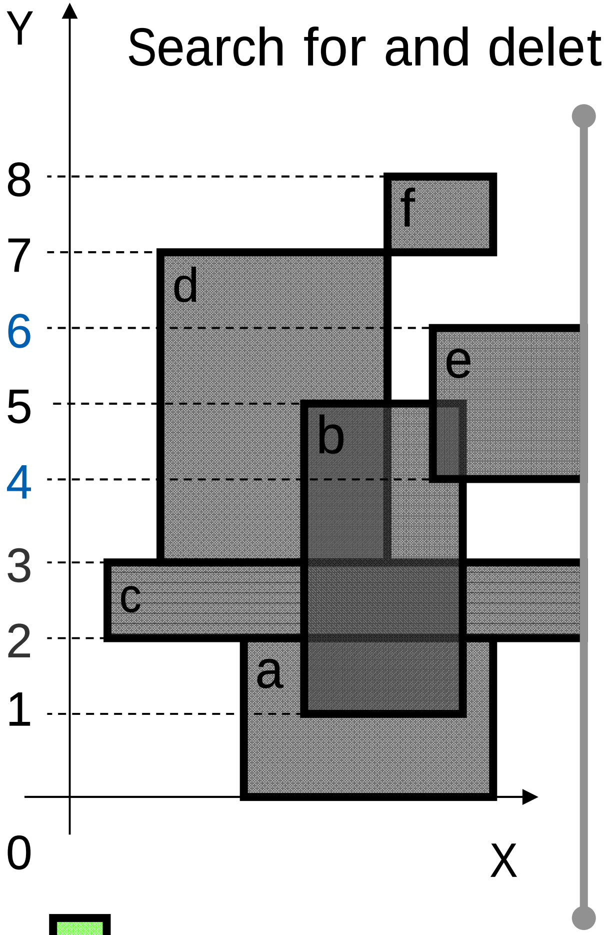
○ Current node

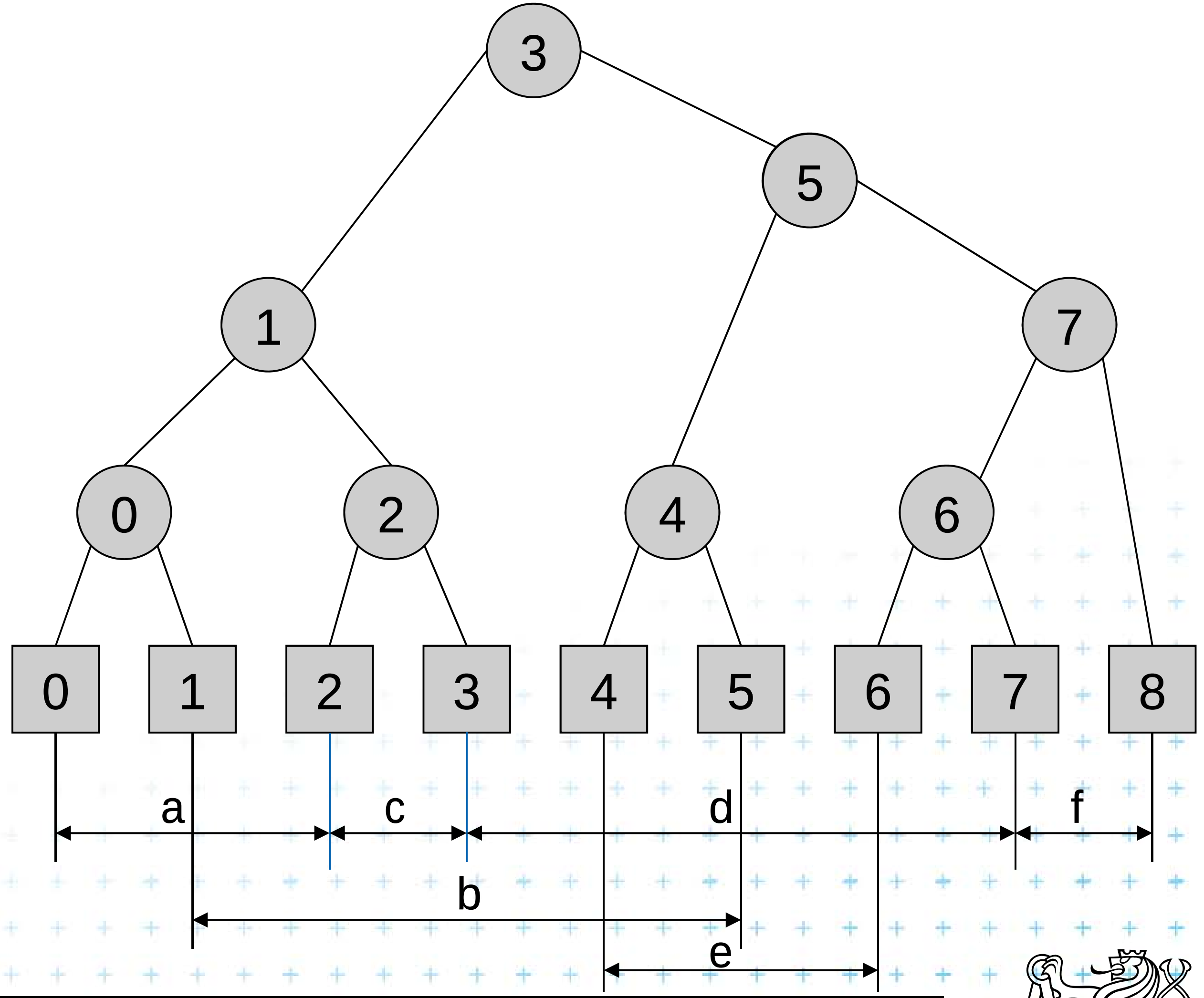
● Active node

[Drtina]

Empty tree

Search for and delete node with interval [4,6]





🟩 Active rectangle

○ Current node

● Active node

[Drtina]

Complexities of rectangle intersections

- n rectangles, s intersected pairs found
- O(n log n) preprocessing time to separately sort
  - x-coordinates of the rectangles for the plane sweep
  - the y-coordinates for initializing the interval tree.
- The plane sweep itself takes O(n log n + s) time, so the overall time is O(n log n + s)
- O(n) space
- This time is optimal for a decision-tree algorithm (i.e., one that only makes comparisons between rectangle coordinates).

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] Mount, D.: *Computational Geometry Lecture Notes for Fall 2016*, University of Maryland, Lecture 5. http://www.cs.umd.edu/class/fall2016/cmsc754/Lects/cmsc754-fall16-lects.pdf

[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

[Drtina] Tomáš Drtina: Intersection of rectangles. Semestral Assignment. Computational Geometry course, FEL CTU Prague, 2006

[Kukral] Petr Kukrál: Intersection of rectangles. Semestral Assignment. Computational Geometry course, FEL CTU Prague, 2006

[Vigneron] Segment trees and interval trees, presentation, INRA, France, http://w3.jouy.inra.fr/unites/miaj/public/vigneron/cs4235/slides.html

DCGI

DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

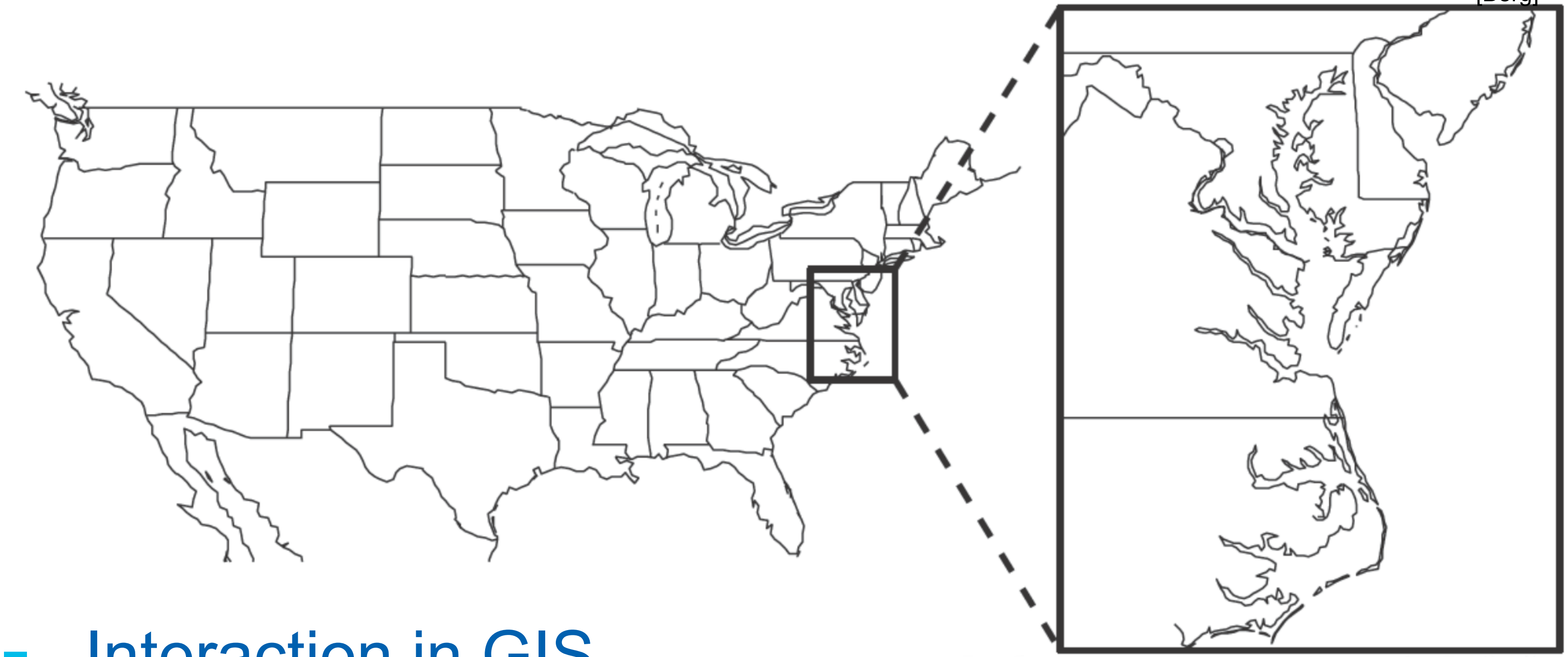
WINDOWING

PETR FELKEL

FEL CTU PRAGUE

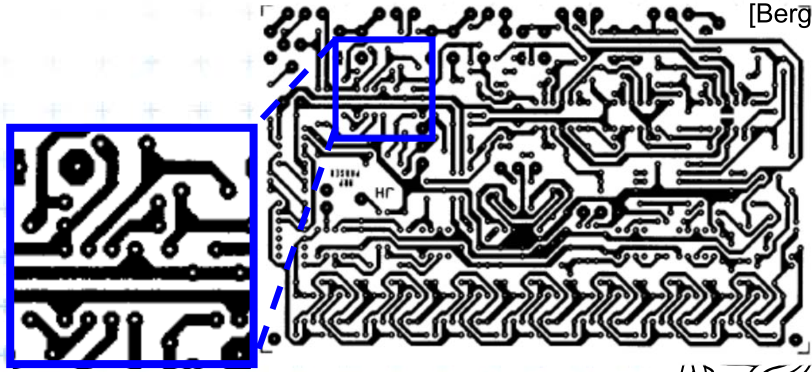
Version from 15.12.2016

Windowing queries - examples



[Berg]

- Interaction in GIS
  - Select subset by outlining
  - Zoom in and re-center
- Circuit board inspection,…



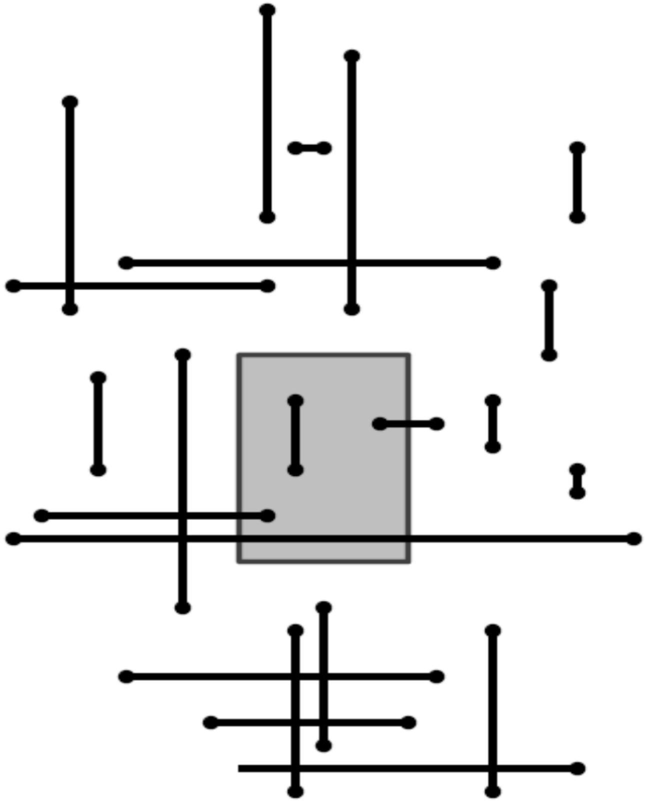
[Berg]

[Vakken]

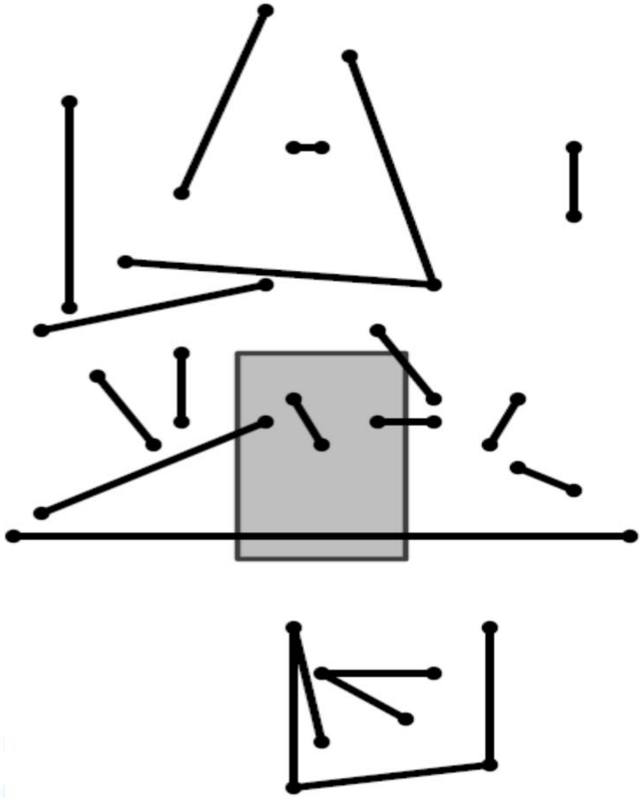
Windowing versus range queries

- Range queries (see range trees in Lecture 03)
  - Points
  - Often in higher dimensions
- Windowing queries
  - Line segments, curves, …
  - Usually in low dimension (2D, 3D)
- The goal for both:
  Preprocess the data into a data structure
  - so that the objects intersected by the query rectangle can be reported efficiently

Windowing queries on line segments



1. Axis parallel line segments



2. Arbitrary line segments (non-crossing)

[Vakken]

Talk overview

1. Windowing of axis parallel line segments in 2D
   - 3 variants of *interval tree – IT in x-direction*
   - Differ in storage of segment end points M_L and M_R

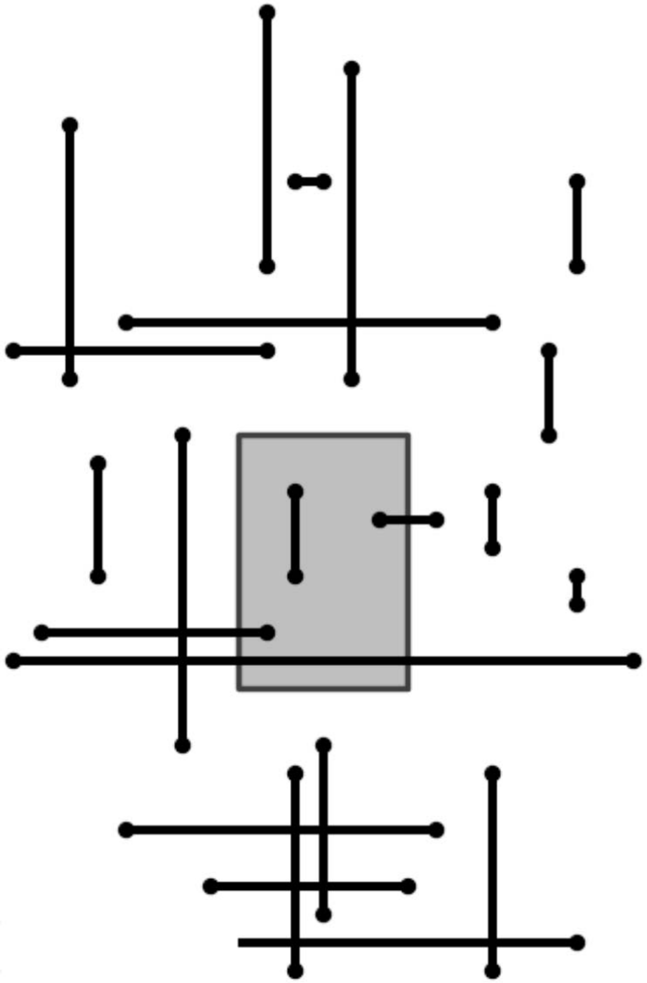
   i. Line stabbing (standard *IT* with *sorted lists*) lecture 9 - intersections

   ii. Line segment stabbing (*IT* with *range trees*)

   iii. Line segment stabbing (*IT* with *priority search trees*)

2. Windowing of line segments in general position
   - *segment tree*

1. Windowing of axis parallel line segments

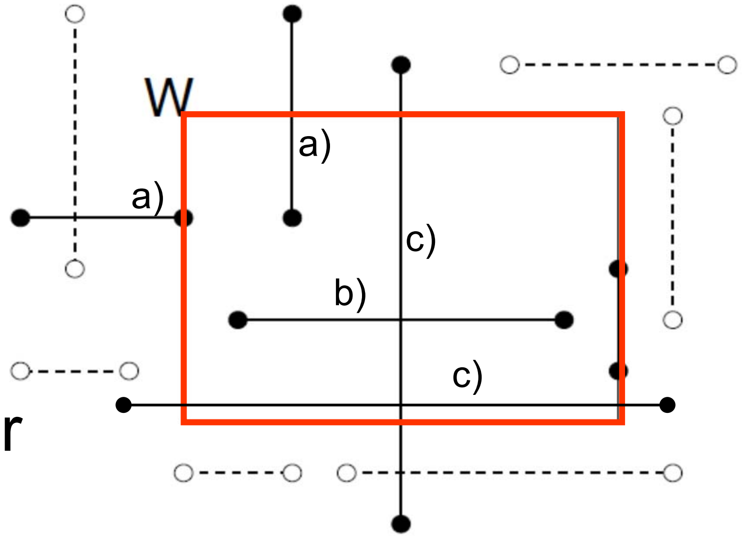


[Vakken]

1. Windowing of axis parallel line segments

Window query

- Given
  - a set of orthogonal line segments S (preprocessed),
  - and orthogonal query rectangle $W = [\, x : x' \,] \times [\, y : y' \,]$
- Count or report all the line segments of S that intersect W
- Such segments have
  a) 1 endpoint in
  b) 2 end points in – Included
  c) no end point in – Cross over

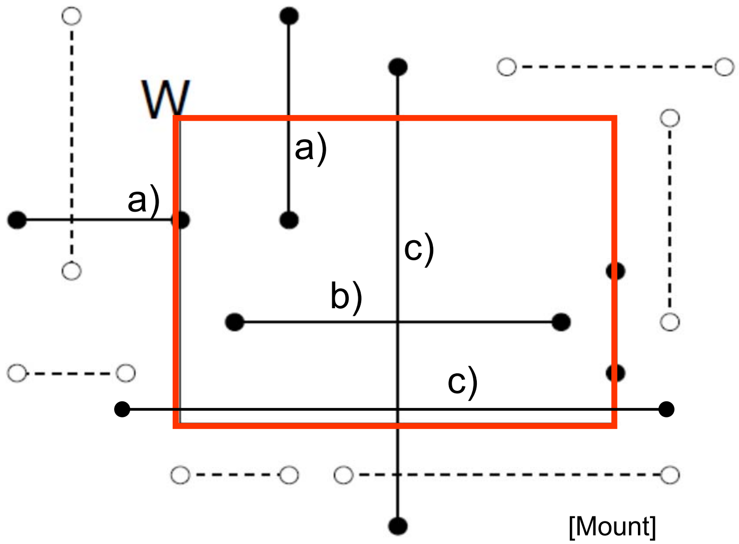


[Mount]

Line segments with 1 or 2 points inside

a) 1 point inside

- Use a range tree (Lesson 3)
- O(n log n) storage
- O($\log^2 n + k$) query time or
- O($\log n + k$) with fractional cascading



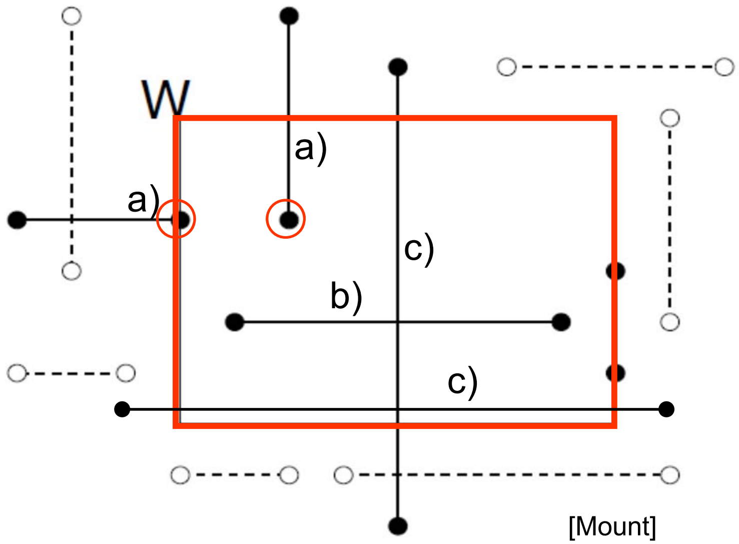
b) 2 points inside – as a) 1 point inside

- Avoid reporting twice

1. Mark segment when reported (clear after the query)
2. When end point found, check the other end-point. Report only the leftmost or bottom endpoint

Line segments with 1 or 2 points inside

a) 1 point inside
- Use a range tree (Lesson 3)
- $O(n \log n)$ storage
- $O(\log^2 n + k)$ query time or
- $O(\log n + k)$ with fractional cascading



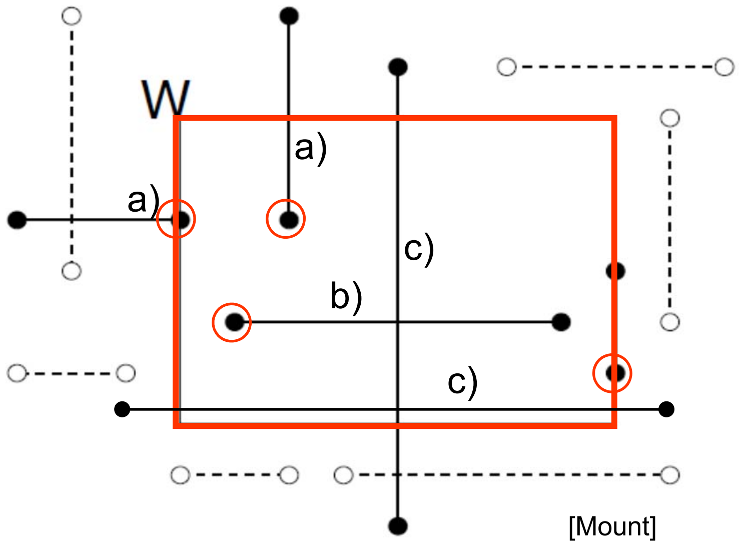
b) 2 points inside – as a) 1 point inside
- Avoid reporting twice

1. Mark segment when reported (clear after the query)
2. When end point found, check the other end-point. Report only the leftmost or bottom endpoint

Line segments with 1 or 2 points inside

a) 1 point inside

- Use a range tree (Lesson 3)
- O(n log n) storage
- O(log^2 n + k) query time or
- O(log n + k) with fractional cascading



b) 2 points inside – as a) 1 point inside

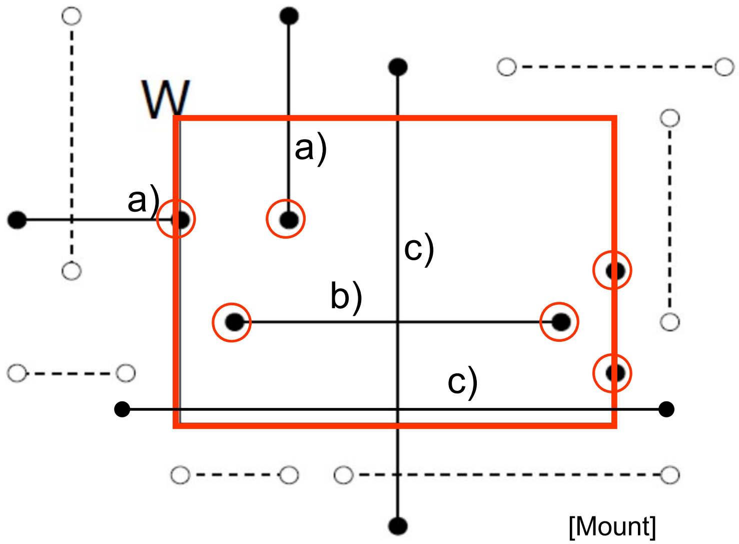
- Avoid reporting twice

1. Mark segment when reported (clear after the query)
2. When end point found, check the other end-point. Report only the leftmost or bottom endpoint

Line segments with 1 or 2 points inside

a) 1 point inside

- Use a range tree (Lesson 3)
- O(n log n) storage
- O($\log^2 n + k$) query time or
- O($\log n + k$) with fractional cascading



[Mount]

b) 2 points inside – as a) 1 point inside

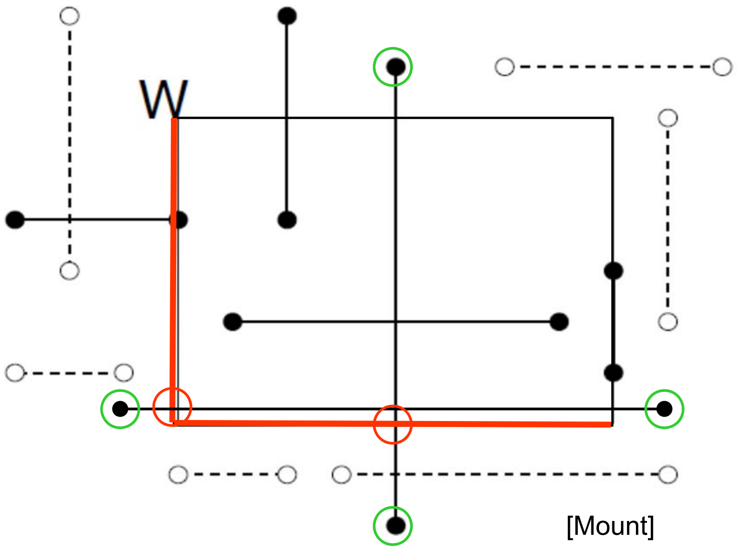
- Avoid reporting twice

1. Mark segment when reported (clear after the query)
2. When end point found, check the other end-point. Report only the leftmost or bottom endpoint

Line segments that cross over the window

c) No points inside

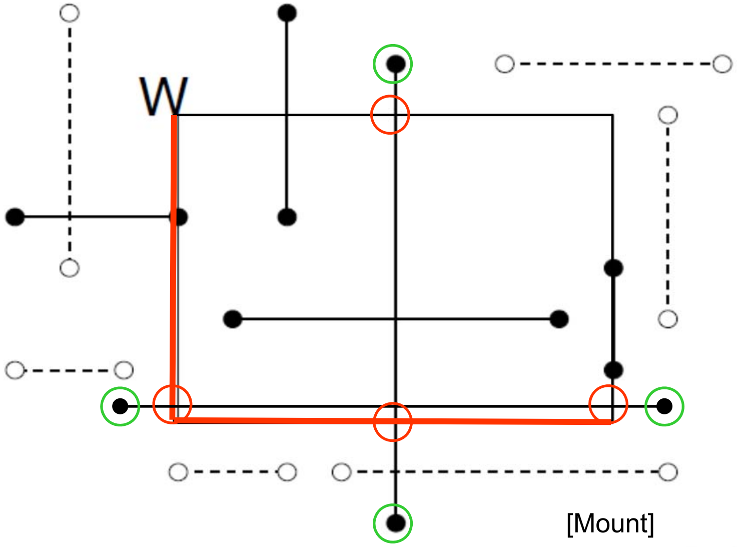
- Such segments not detected using end-point range tree
- Cross the boundary twice
  or
  contain one boundary edge
- It is enough to
  detect segments intersected by the left and bottom boundary edges (not having end point inside)
- For left boundary: Report the segments intersecting vertical query *line segment* (1/ii.)
- Let's discuss vertical query *line* first (1/i.)
- Bottom boundary is rotated 90°



Line segments that cross over the window

c) No points inside

- Such segments not detected using end-point range tree
- Cross the boundary twice
  or
  contain one boundary edge
- It is enough to
  detect segments intersected by the left and bottom boundary edges (not having end point inside)
- For left boundary: Report the segments intersecting vertical query *line segment* (1/ii.)
- Let's discuss vertical query *line* first (1/i.)
- Bottom boundary is rotated 90°



[Mount]

Talk overview

1. Windowing of axis parallel line segments in 2D (variants of *interval tree - IT*)
   1. Line stabbing (standard *IT* with *sorted lists*)
   2. Line segment stabbing (*IT* with *range trees*)
   3. Line segment stabbing (*IT* with *priority search trees*)
2. Windowing of line segments in general position
   - *segment tree*

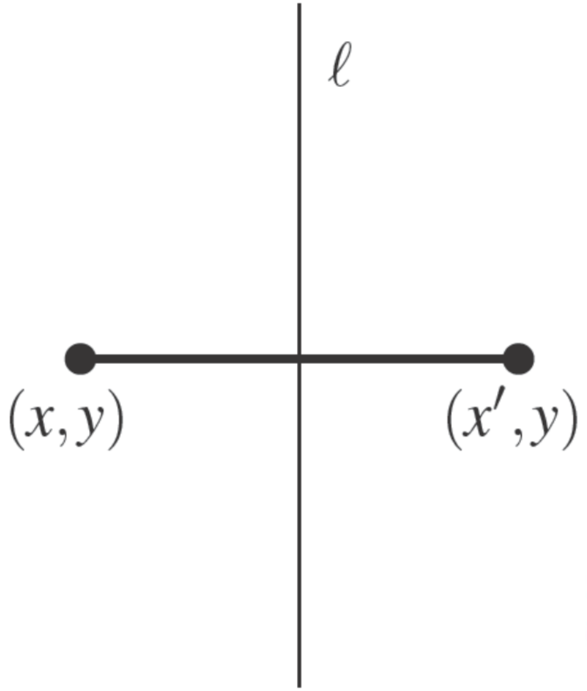
i. Segment intersected by vertical line – 1D

- Query line ℓ := (x=q_x)

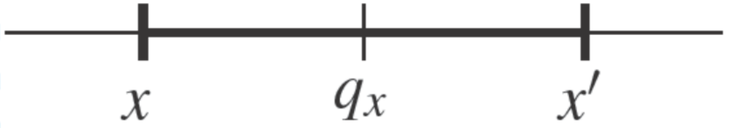
  Report the segments stabbed by a vertical line

  = 1 dimensional problem

  (ignore y coordinate)



⇒ Report the interval containing query point q_x

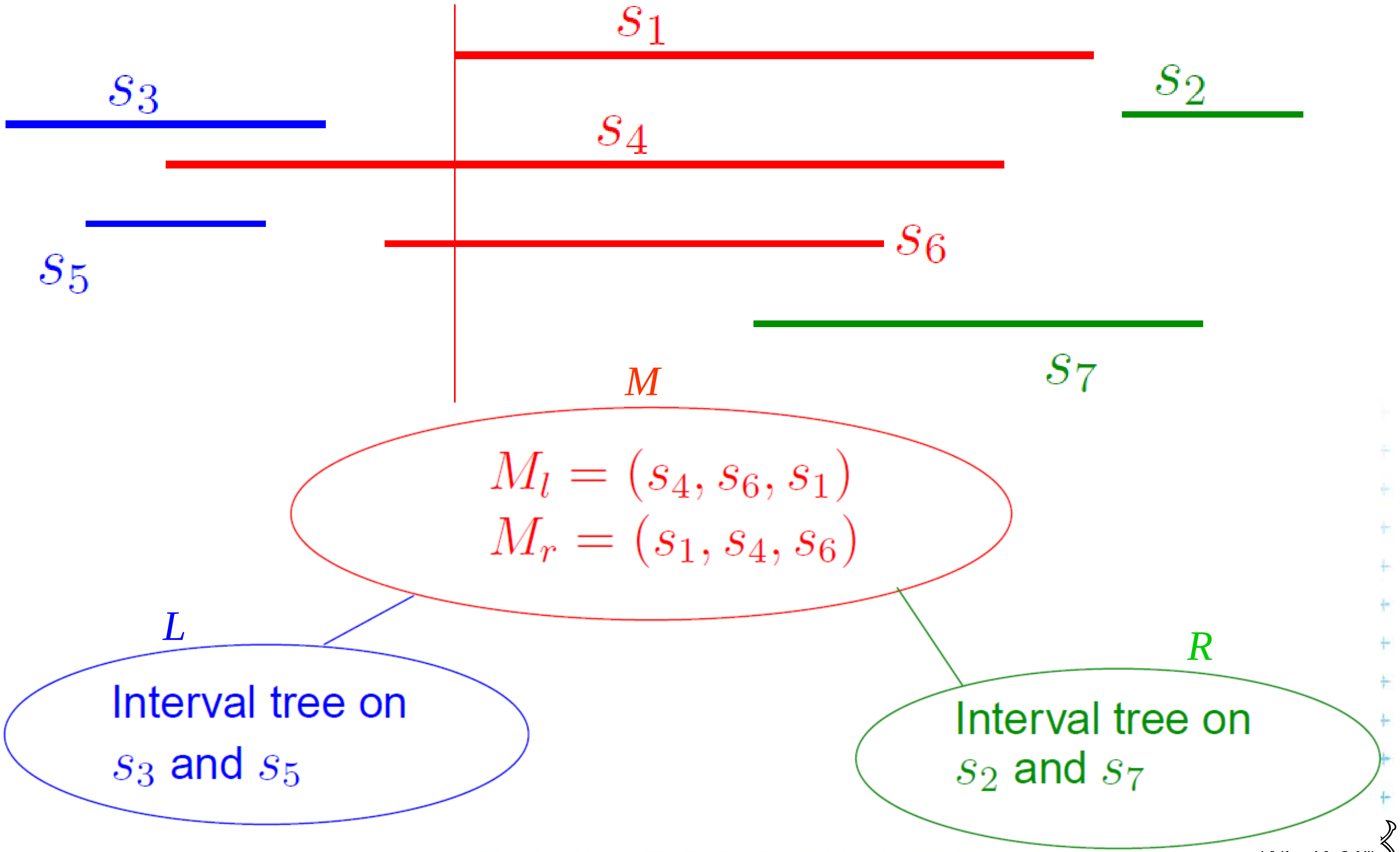


DS: Interval tree with sorted lists

[Mount]

Interval tree principle

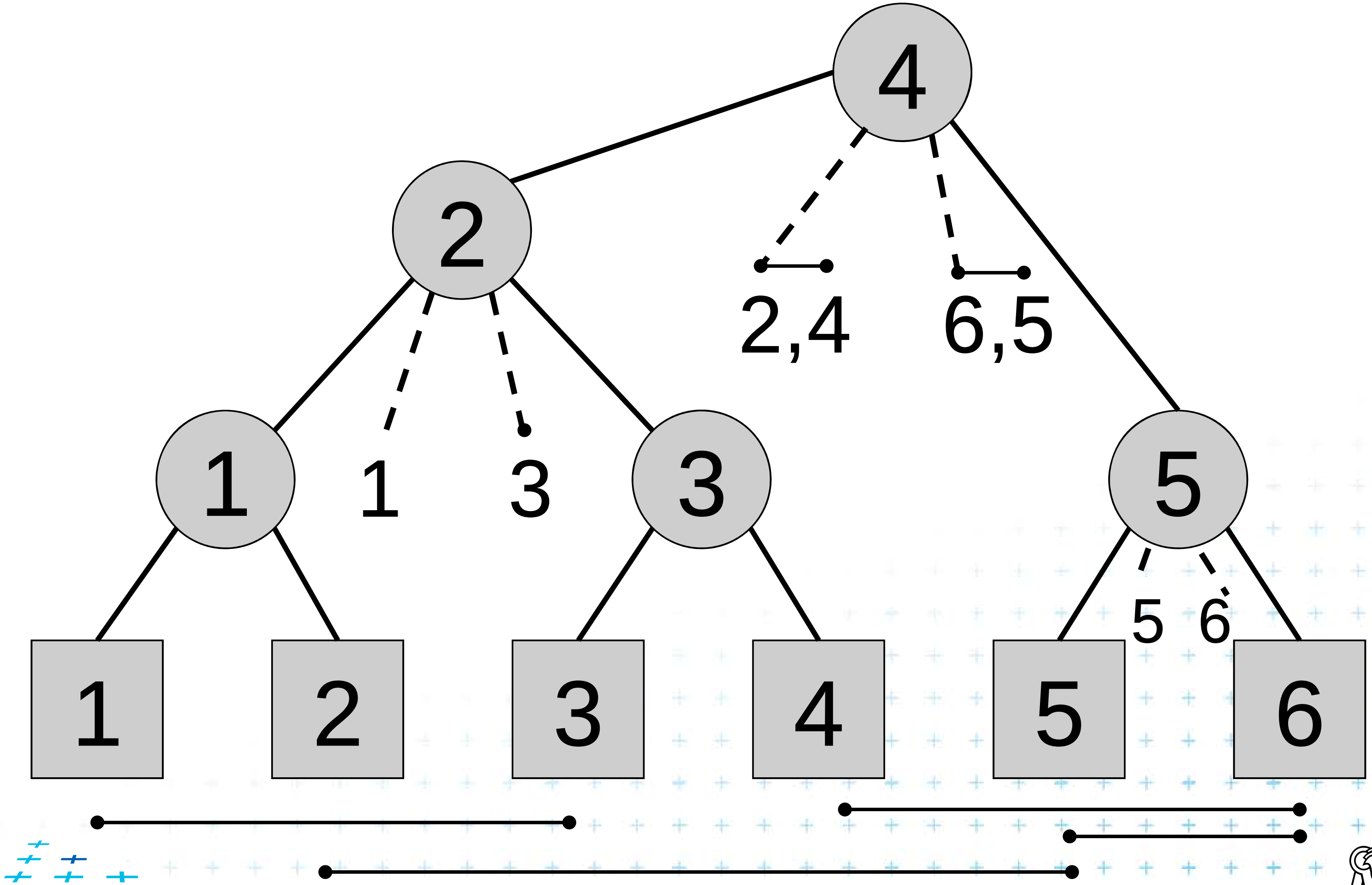
(see lecture 9 - intersections)



[Vigneron]

Static interval tree [Edelsbrunner80]

Tree over sorted segment end-points

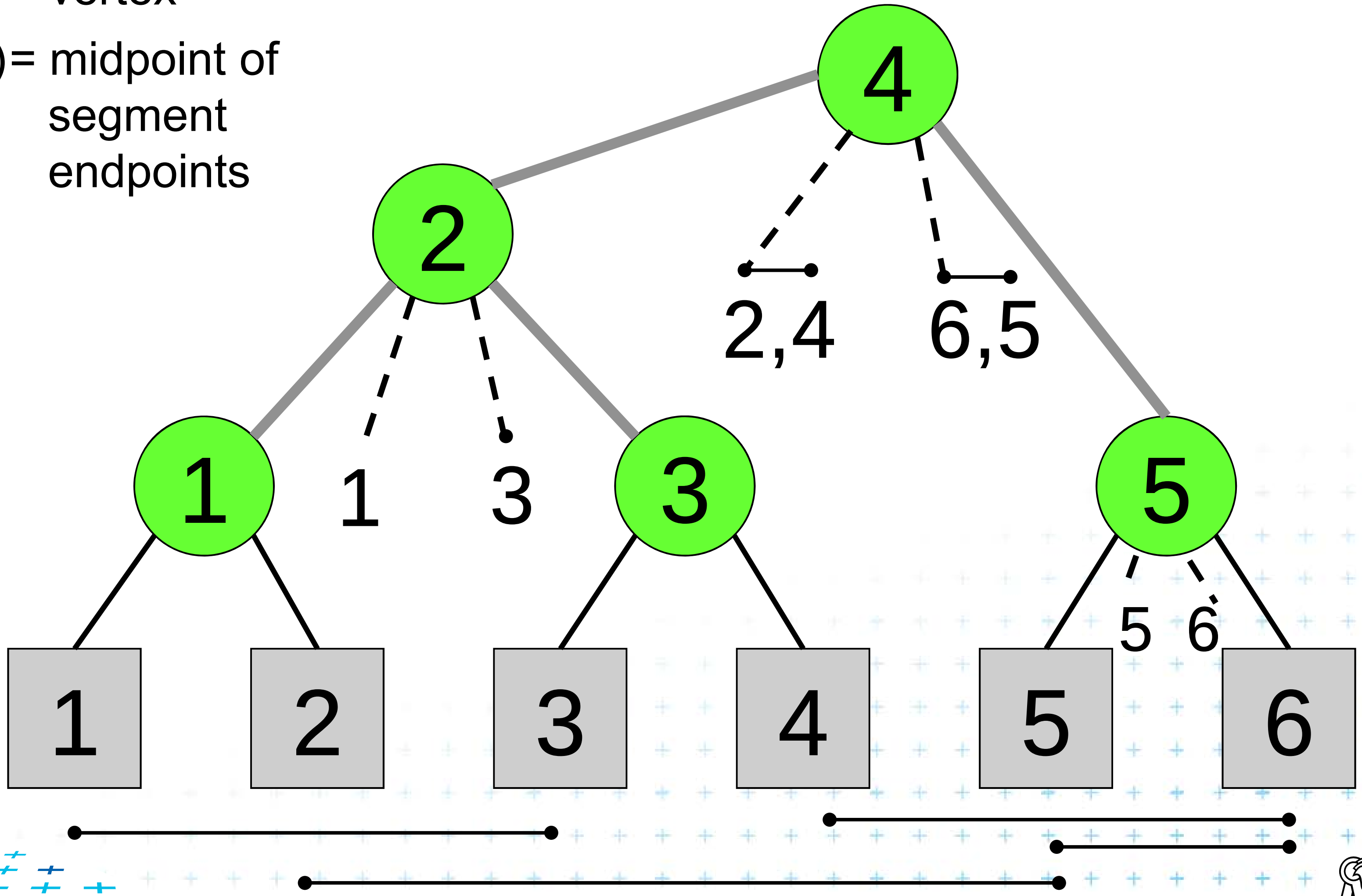


[Kukral]

Primary structure – static tree for endpoints

v = vertex

d(v)= midpoint of segment endpoints



[Kukral]

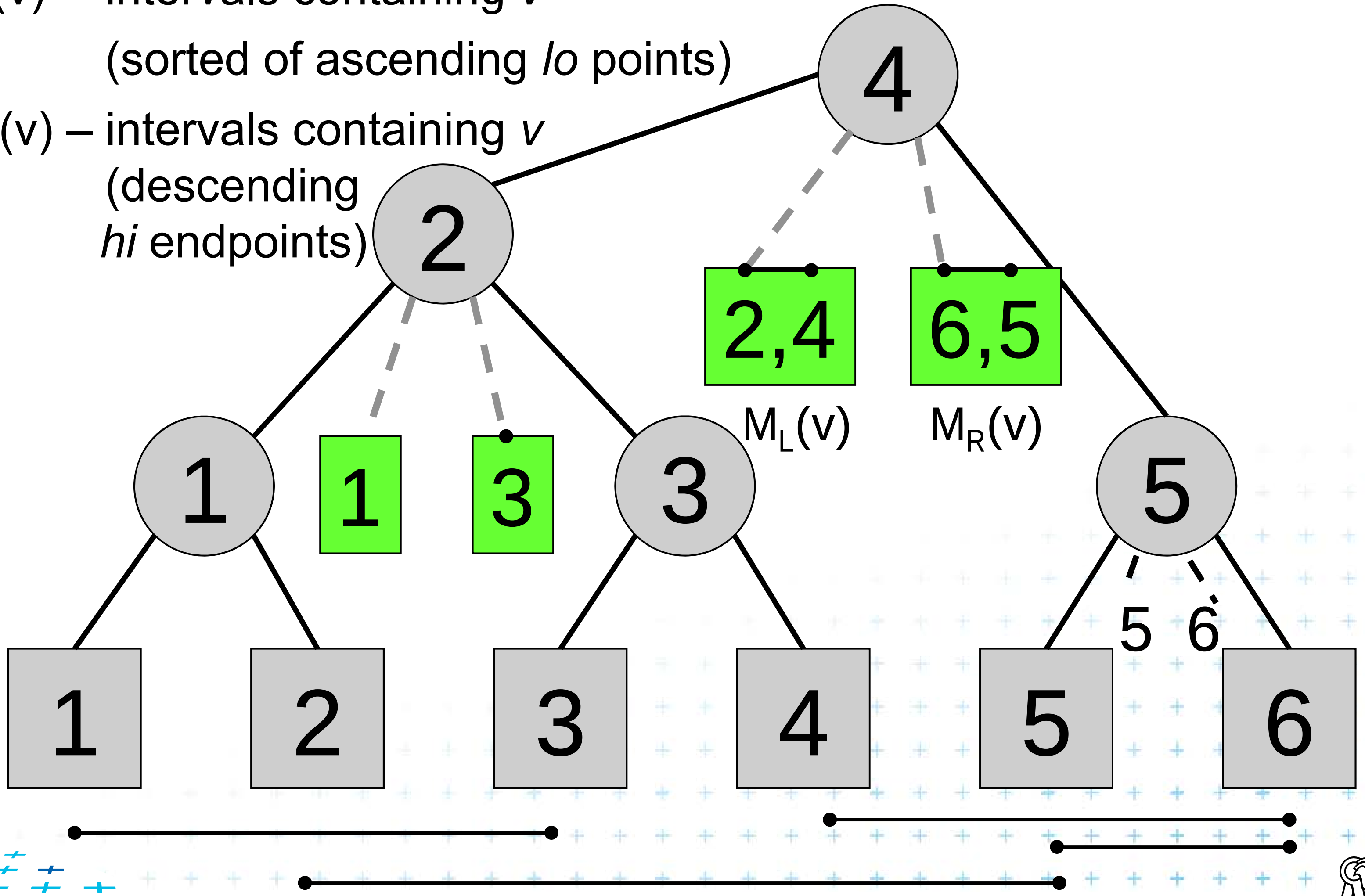
Secondary lists – sorted segments in M

ML(v) – intervals containing *v*

(sorted of ascending *lo* points)

MR(v) – intervals containing *v*

(descending *hi* endpoints)



[Kukral]

Interval tree construction

Merged procedures from in lecture 09

- PrimaryTree(*S*) on slide 33
- InsertInterval (*b, e, T*) on slide 35

ConstructIntervalTree(*S*) **// Intervals all active – no active lists**

Input: Set S of intervals on the real line – on *x-axis*

Output: The root of an interval tree for *S*

```
if (|S| == 0) return null                                   // no more intervals
else
    xMed = median endpoint of intervals in S                // median endpoint
    L = { [xlo, xhi] in S | xhi < xMed }                    // left of median
    R = { [xlo, xhi] in S | xlo > xMed }                    // right of median
    M = { [xlo, xhi] in S | xlo <= xMed <= xhi }            // contains median
    ML = sort M in increasing order of xlo                  // sort M
    MR = sort M in decreasing order of xhi
    t = new IntTreeNode(xMed, ML, MR)                       // this node
    t.left  = ConstructIntervalTree(L)                      // left subtree
    t.right = ConstructIntervalTree(R)                      // right subtree
    return t
```

[Mount]

Line stabbing query for an interval tree

Less effective variant of QueryInterval (b, e, T)
on slide 34 in lecture 09
with merged parts: fork and search right

Stab(t, xq)

Input: IntTreeNode t, Scalar xq

Output: prints the intersected intervals

```
if (t == null) return                           // no leaf: fell out of the tree
if (xq < t.xMed)                                // left of median?
    for (i = 0; i < t.ML.length; i++)           // traverse ML
            if (t.ML[i].lo ≤ xq) print(t.ML[i])  // ..report if in range
            else break                          // ..else done
    stab(t.left, xq)                            // recurse on left
else  // (xq ≥ t.xMed)                          // right of or equal to median
    for (i = 0; i < t.MR.length; i++) {         // traverse MR
            if (t.MR[i].hi ≥ xq) print(t.MR[i])  // ..report if in range
            else break                          // ..else done
    stab(t.right, xq)                           // recurse on right
```

Note: Small inefficiency for xq == t.xMed – recurse on right

[Mount]

Complexity of line stabbing via interval tree

- Construction - $O(n \log n)$ time
  - Each step divides at maximum into two halves or less (minus elements of M) => tree of height $h = O(\log n)$
  - If presorted endpoints in three lists L,R, and M then median in O(1) and copy to new L,R,M in O(n)]
- Vertical line stabbing query - $O(k + \log n)$ time
  - One node processed in $O(1 + k')$, k'reported intervals
  - v visited nodes in $O(v + k)$, k total reported intervals
  - $v = h =$ tree height = $O(\log n)$ $k = \Sigma k'$
- Storage - $O(n)$
  - Tree has $O(n)$ nodes, each segment stored twice (two endpoints)

Talk overview

1. Windowing of axis parallel line segments in 2D (variants of *interval tree – IT*)
   1. Line stabbing (standard *IT* with *sorted lists*)
   2. Line segment stabbing (*IT* with *range trees*)
   3. Line segment stabbing (*IT* with *priority search trees*)
2. Windowing of line segments in general position
   - *segment tree*

DCGI

Line segment stabbing (*IT* with *range trees*)

Enhance 1D interval trees to 2D

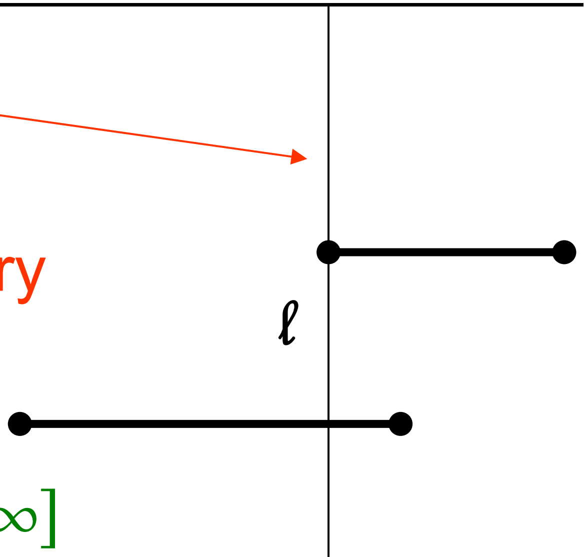
- Change 1D test $q_x \in \langle x, x' \rangle$
  done by interval tree with sorted lists M_L and M_R
  into 2D test $q_x \in (-\infty \ : q_x \ ]$
- and change lines $q_x \times [-\infty : \infty]$ (no y-test)
  to segments $q_x \times [q_y : q'_y]$ (additional y-test)

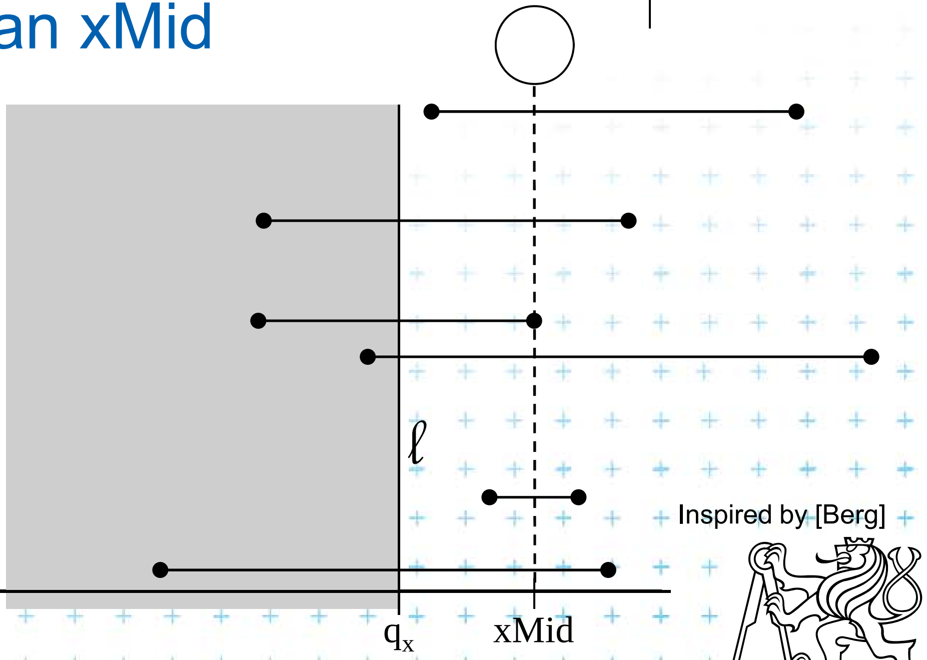
i. Segment intersected by vertical line - 2D

- Query line $\ell := q_x \times [-\infty : \infty]$
- Horizontal segment of *M* stabs the query line ℓ **iff** its left endpoint lies in halph-space

$$q := (-\infty \;: q_x\,] \times [-\infty : \infty]$$

- In IT node with stored median xMid report all segments from M
  - M_L: whose left point lies in $(-\infty \;: q_x\,]$ if ℓ lies left from xMid
  - M_R: whose right point lies in $[q_x \,: +\infty)$ if ℓ lies right from xMid



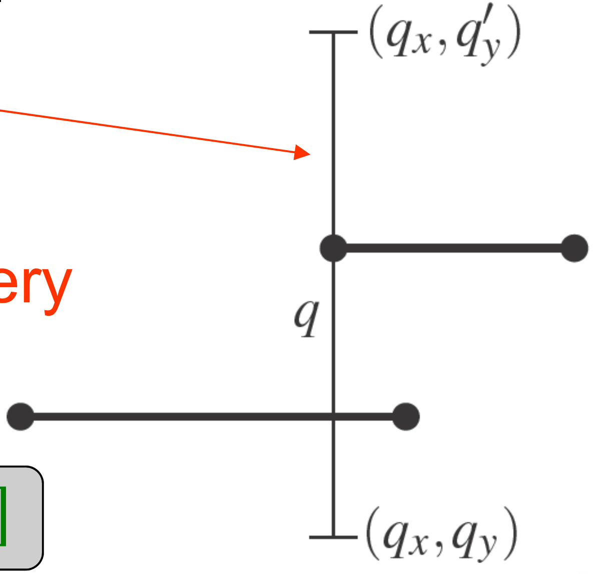


Inspired by [Berg]

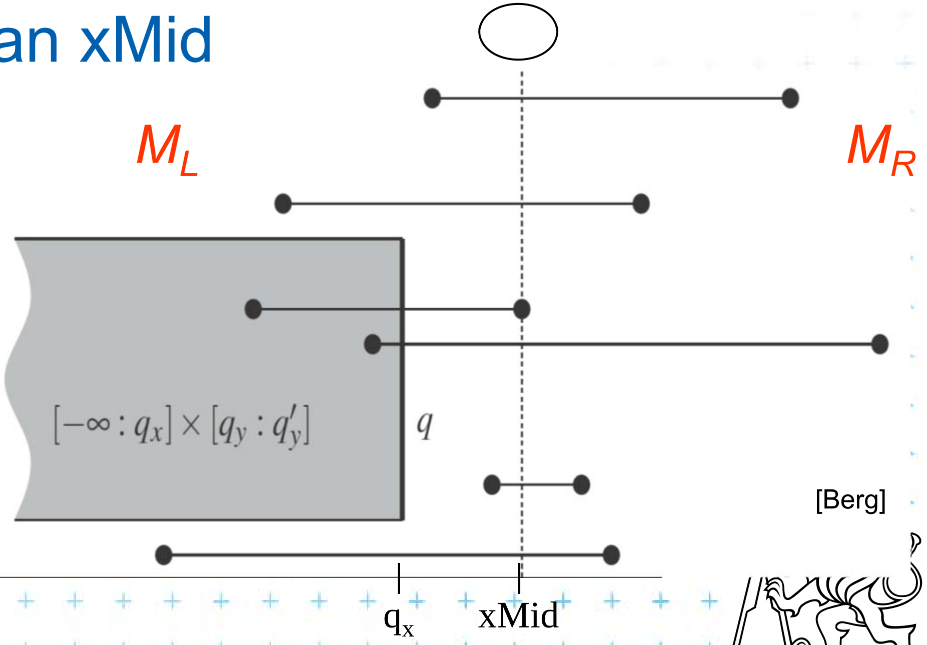
ii. Segment intersected by vertical line segment

- Query segment $q := q_x \times [q_y : q'_y]$
- Horizontal segment of M_L stabs the query segment q **iff** its left endpoint lies in semi-infinite rectangular region

$$q := (-\infty : q_x] \times \boxed{[q_y : q'_y]}$$



- In IT node with stored median xMid report all segments
  - M_L: whose left points lie in $(-\infty : q_x] \times [q_y : q'_y]$ where q_x lies left from xMid
  - M_R: whose right point lies in $[q_x : +\infty) \times [q_y : q'_y]$ where q_x lies right from xMid

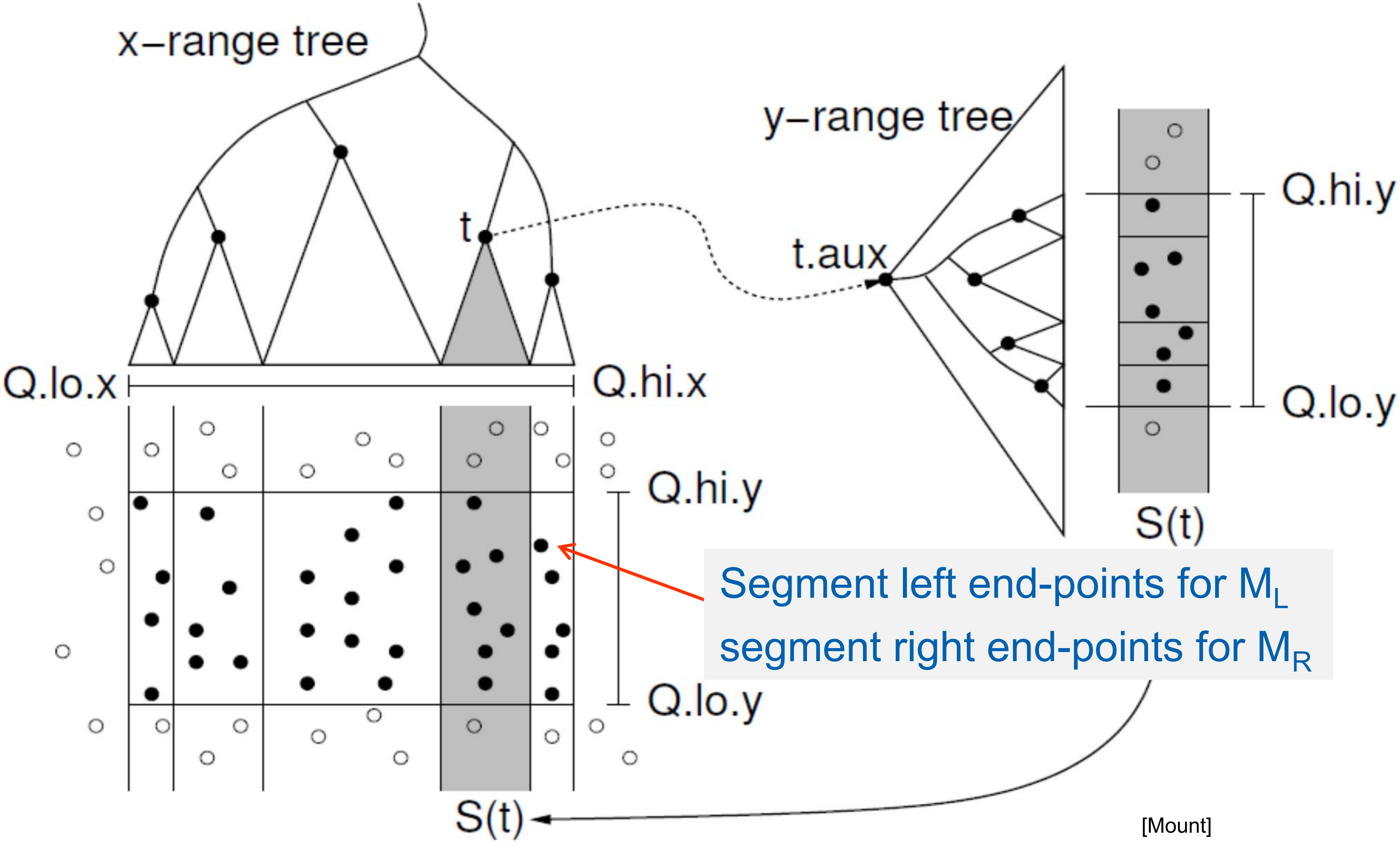


[Berg]

Data structure for endpoints

- Storage of M_L and M_R
  - 1D Sorted lists not enough for line segments
  - Use two 2D range trees
- Instead $O(n)$ sequential search in M_L and M_R perform $O(\log n)$ search in range tree with fractional cascading

2D range tree (without fractional cascading-more in Lecture 3)



Segment left end-points for M_L

segment right end-points for M_R

[Mount]

Complexity of line segment stabbing

- Construction - O(n log n) time
  - Each step divides at maximum into two halves L,R or less (minus elements of M) => tree height O(log n)
  - If the range trees are efficiently build in O(n) after points sorted
- Vertical line segment stab. q. - O(k + log^2 n) time
  2D range tree search with Fractional Cascading
  - One node processed in O(log n + k'), k'=reported inter.
  - v-visited nodes in O(v log n + k), k=total reported inter.
  - v = interval tree height = O(log n)
  - O(k + log^2 n) time - range tree with fractional cascading
  - O(k + log^3 n) time - range tree without fractional casc.
- Storage - O(n log n)
  - Dominated by the range trees

DCGI

Talk overview

1. Windowing of axis parallel line segments in 2D (variants of *interval tree - IT*)
   1. Line stabbing (standard *IT* with *sorted lists*)
   2. Line segment stabbing (*IT* with *range trees*)
   3. Line segment stabbing (*IT* with *priority search trees*)
2. Windowing of line segments in general position
   - *segment tree*

iii. Priority search trees [McCreight85]

- Priority search trees – in case c) on slide 9
  - Exploit the fact that query rectangle in range queries is unbounded (in x direction)
  - Can be used as secondary data structures for both left and right endpoints (ML and MR) of segments in nodes of interval tree – one for ML, one for MR
  - Improve the storage to O(*n*) for horizontal segment intersection with window edge (Range tree has O(*n* log *n*))
- For cases a) and b) - O(*n* log *n*) remains
  - we need range trees for windowing segment endpoints

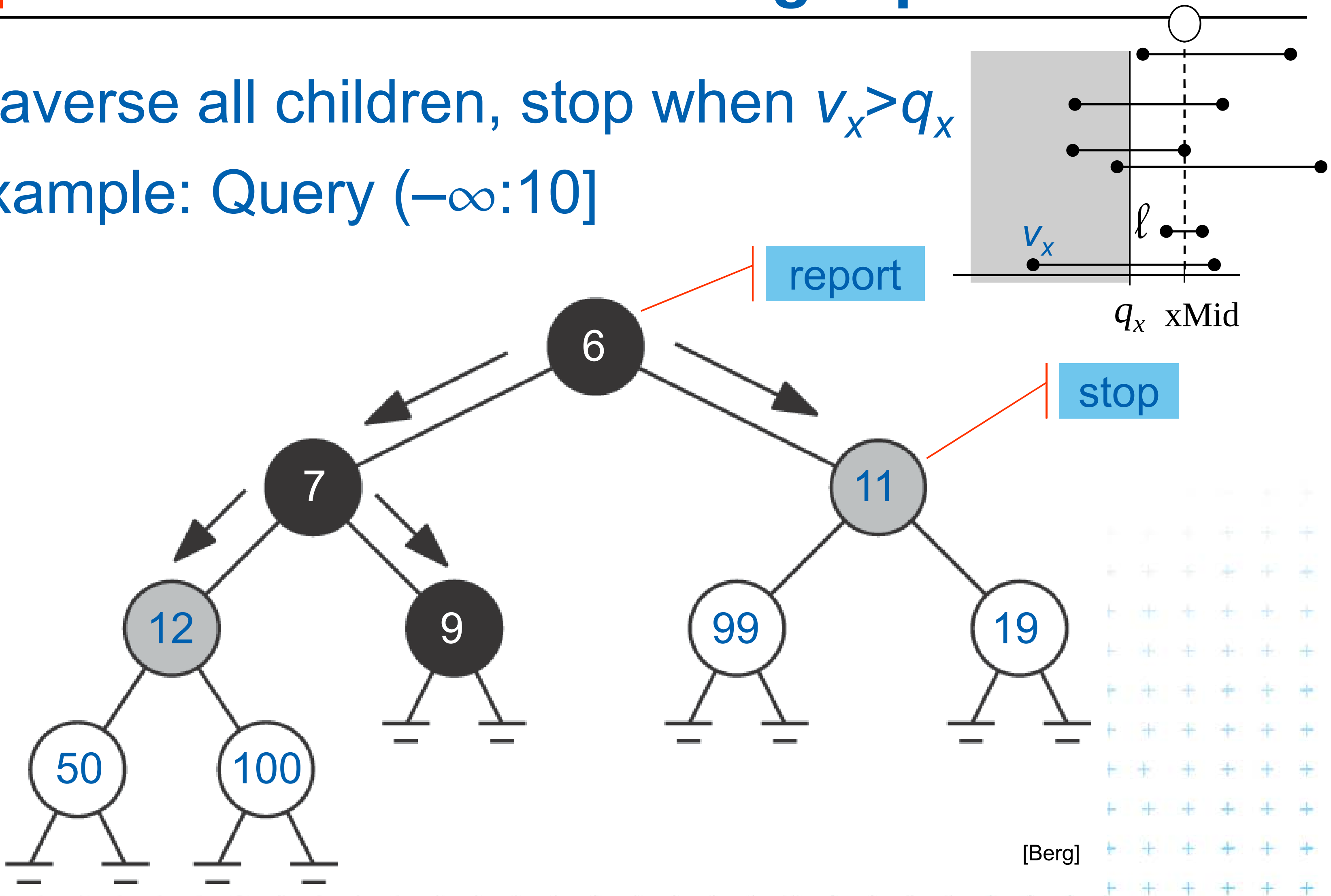
Rectangular range queries variants

- Let $P = \{ p_1, p_2, \ldots, p_n \}$ is set of points in plane
- Goal: rectangular range queries of the form $(-\infty : q_x] \times [q_y ; q'_y]$
- In 1D: search for nodes v with $v_x \in (-\infty : q_x]$
  - range tree O(log n + k) time
  - ordered list O(1 + k) time (start in the leftmost, stop on v with $v_x > q_x$)
  - use heap O(1 + k) time ! (traverse all children, stop when $v_x > q_x$)
- In 2D – use heap for points with x $\in (-\infty : q_x]$

  + integrate information about y-coordinate

Heap for 1D unbounded range queries

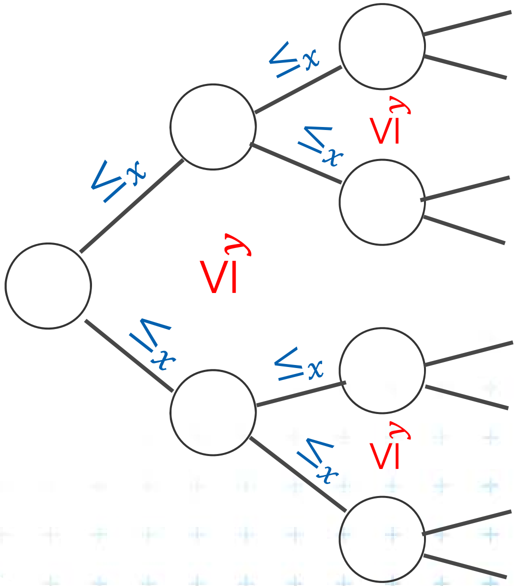
- Traverse all children, stop when $v_x > q_x$
- Example: Query (–∞:10]



[Berg]

Principle of priority search tree

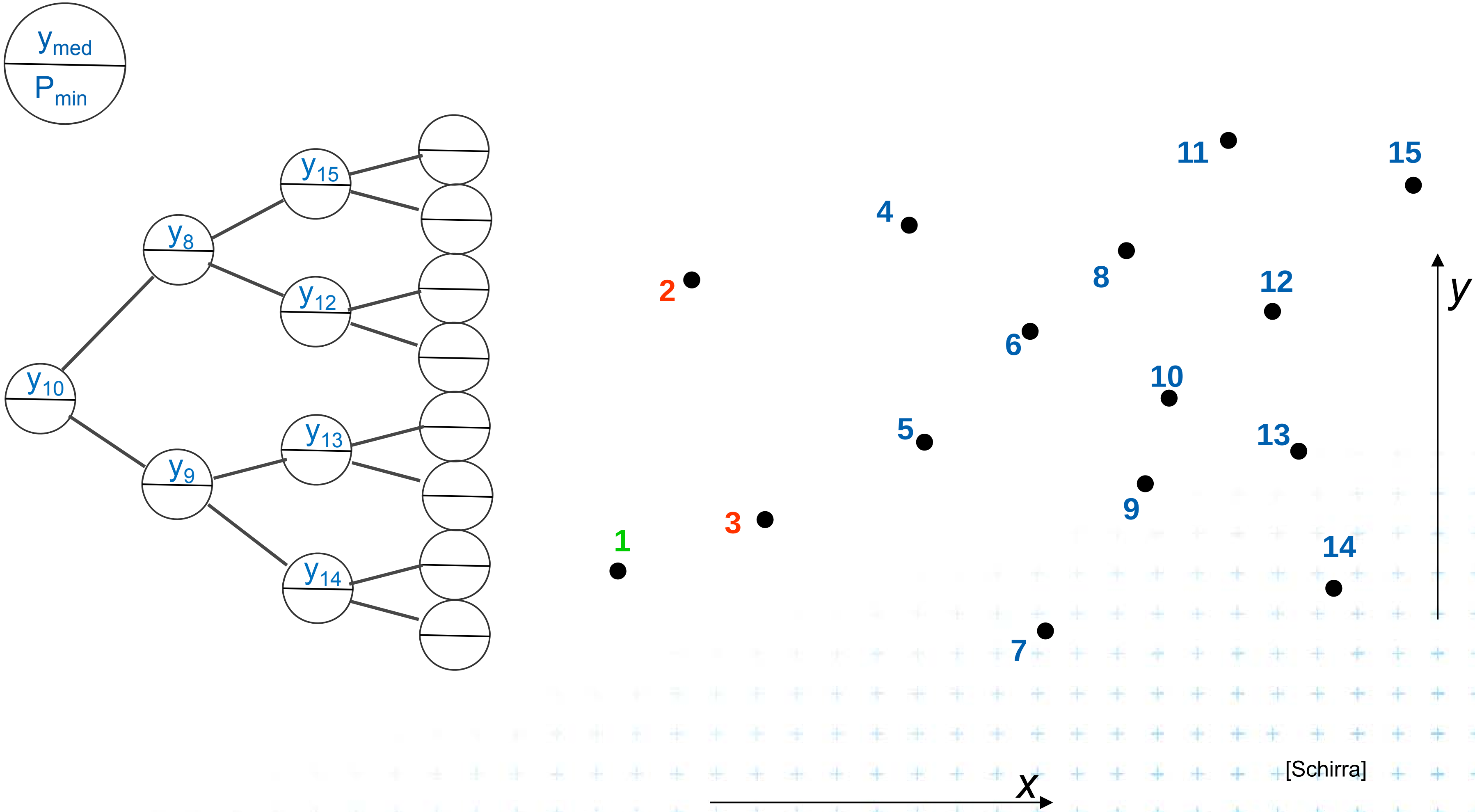
- Heap
  - relation between parent and its child nodes
  - no relation between the child nodes themselves
- Priority search tree
  - relate the child nodes according to y



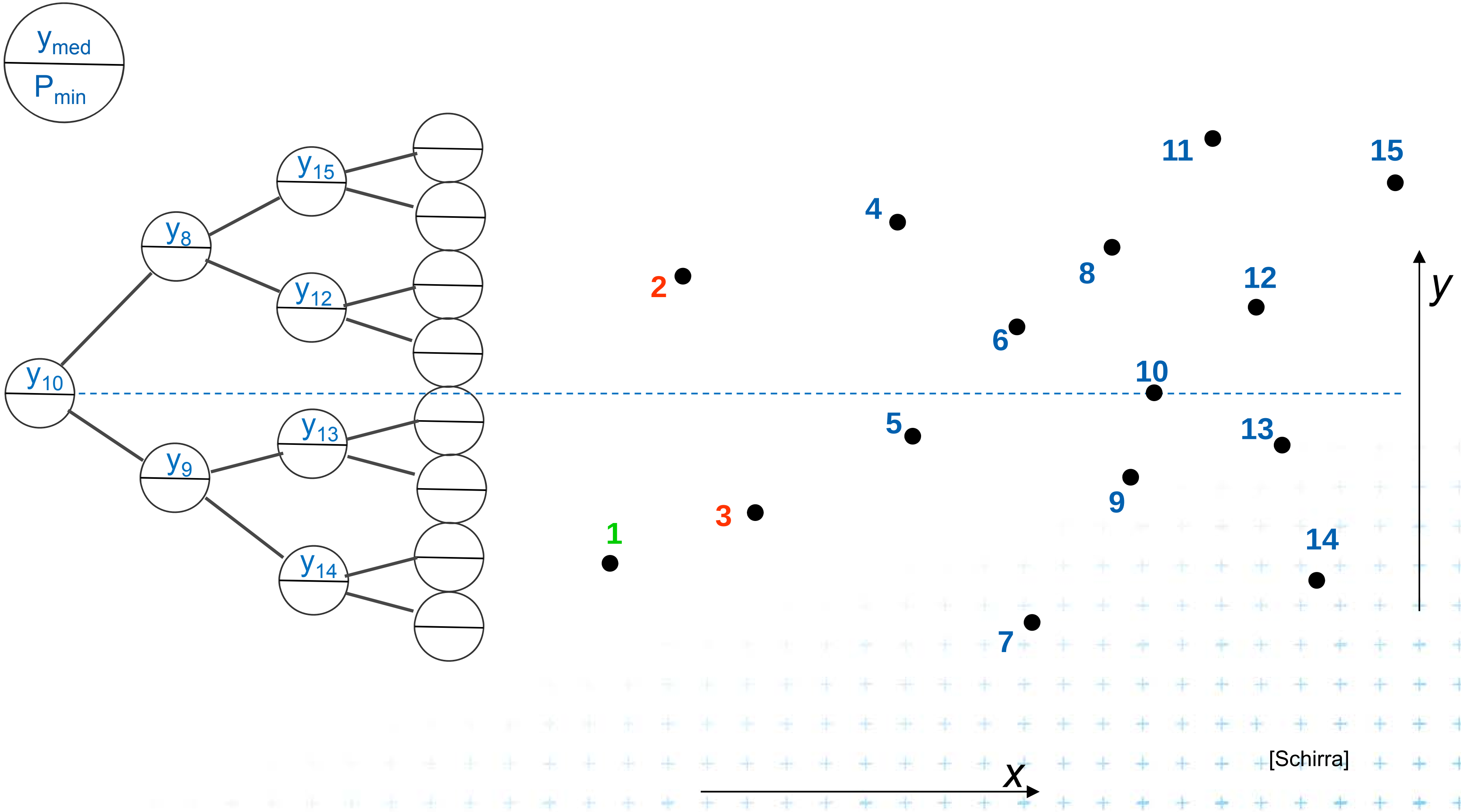
Priority search tree (PST)

- Heap in 2D can incorporate info about both *x,y*
  - BST on *y*-coordinate (horizontal slabs) ~ range tree
  - Heap on *x*-coordinate (minimum *x* from slab along *x*)
- If *P* is empty, PST is empty leaf
- else
  - p_{min} = point with smallest x-coordinate in *P* --- a heap root
  - y_{med} = *y*-coord. median of points $P \setminus \{p_{min}\}$ --- BST root
  - $P_{below} := \{ p \in P \setminus \{p_{min}\} : p_y \leq y_{med}\}$
  - $P_{above} := \{ p \in P \setminus \{p_{min}\} : p_y > y_{med}\}$
- Point p_{min} and scalar y_{med} are stored in the PST root
- The left subtree is PST of P_{below}
- The right subtree is PST of P_{above}

Priority search tree construction example

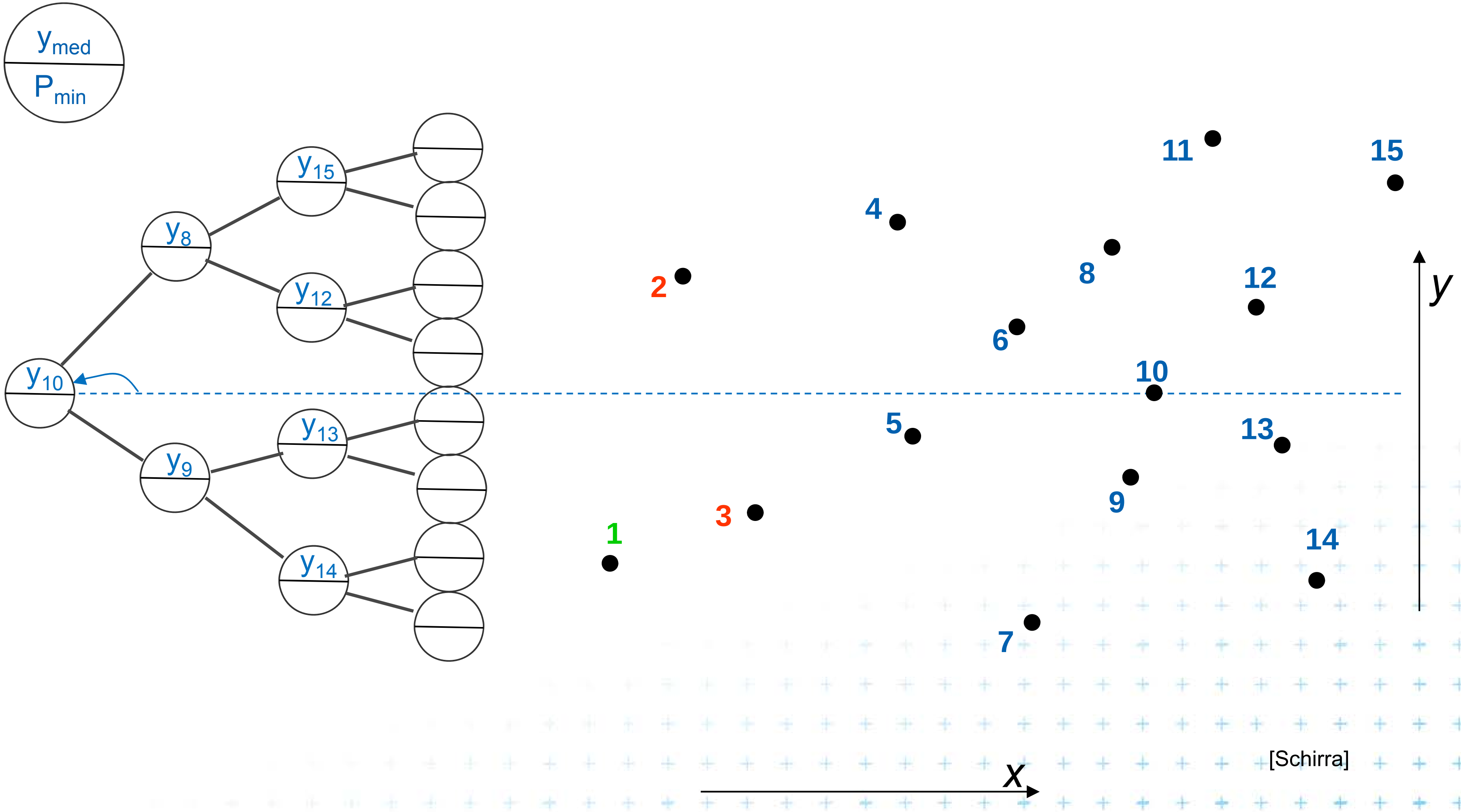


Priority search tree construction example



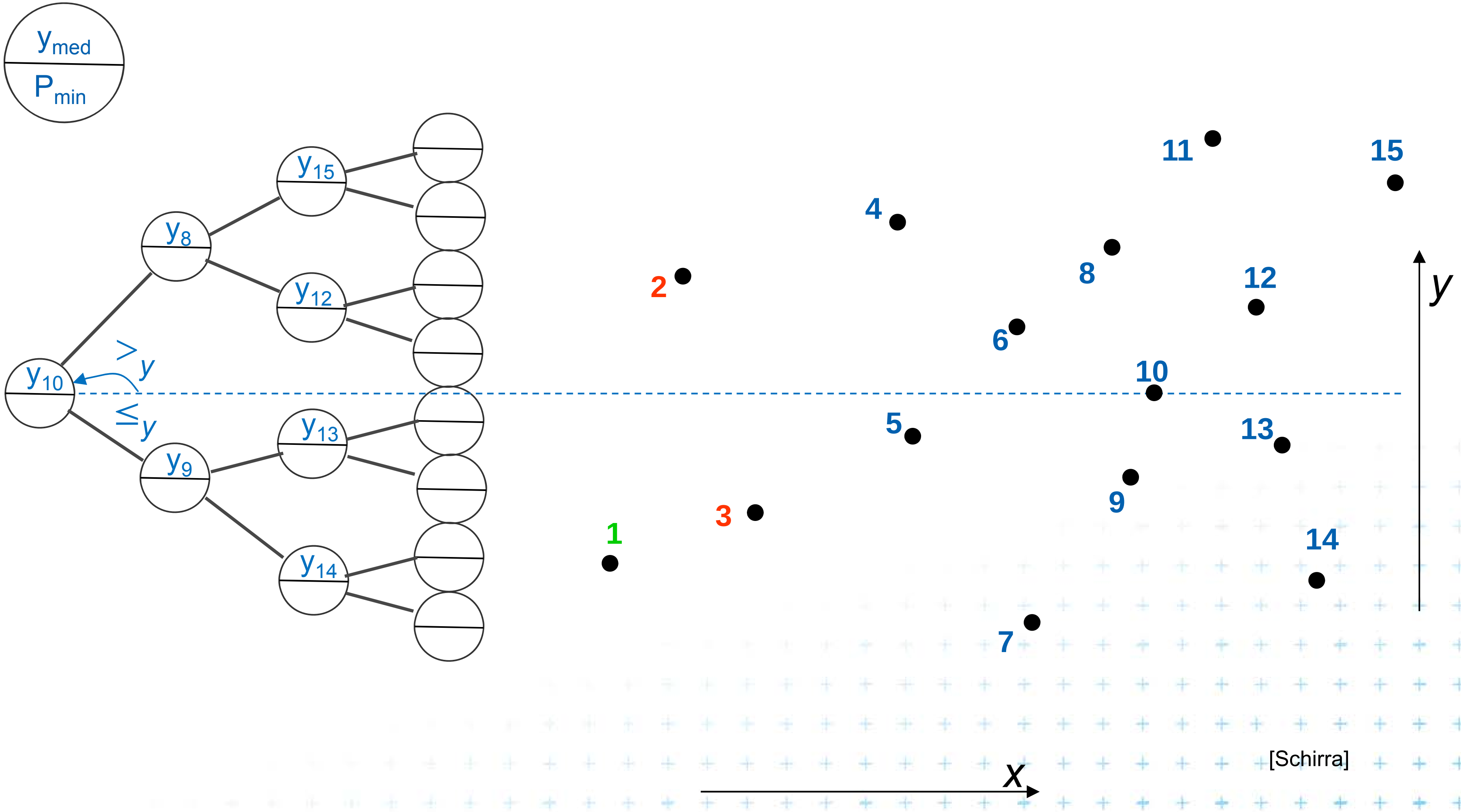
[Schirra]

Priority search tree construction example



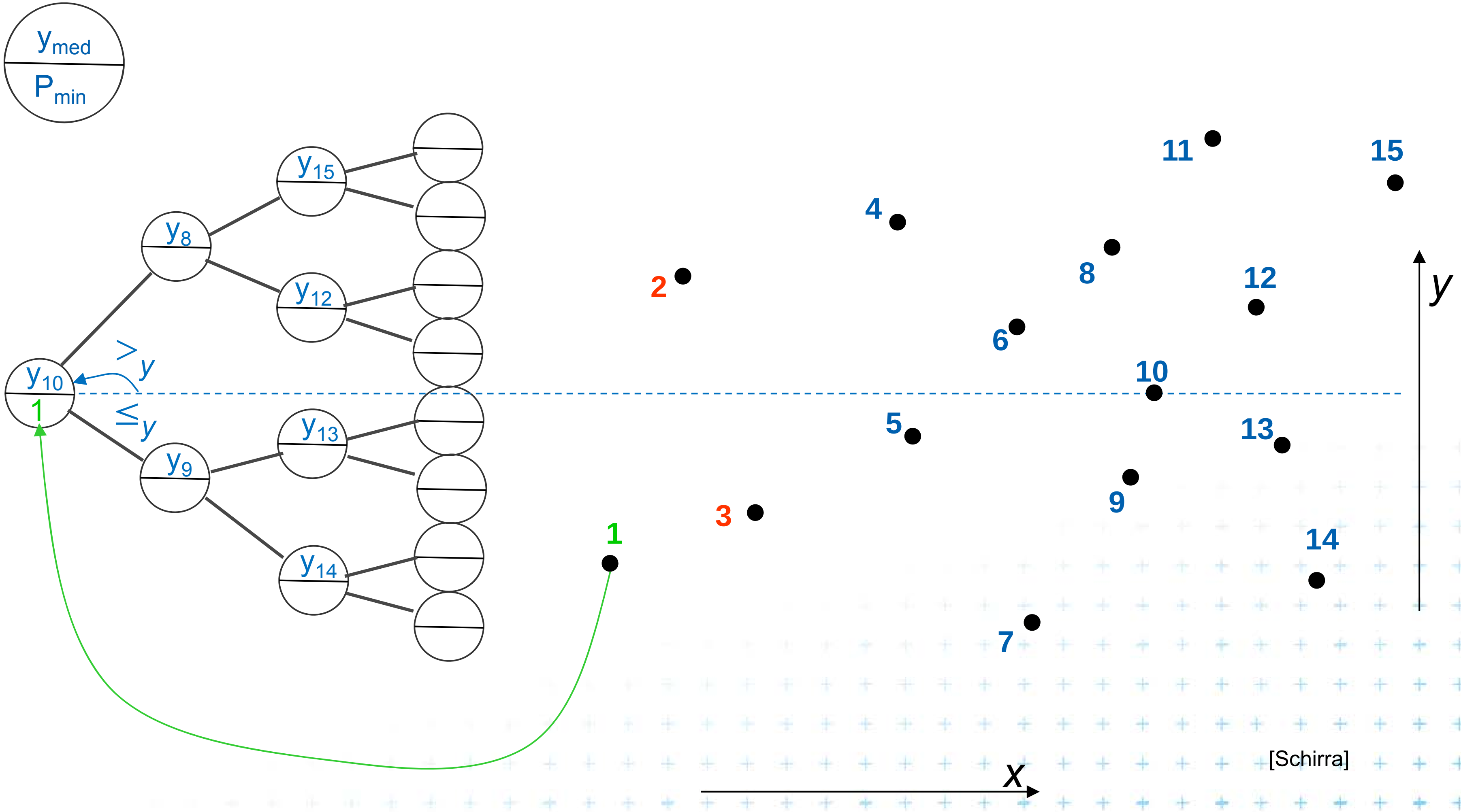
[Schirra]

Priority search tree construction example



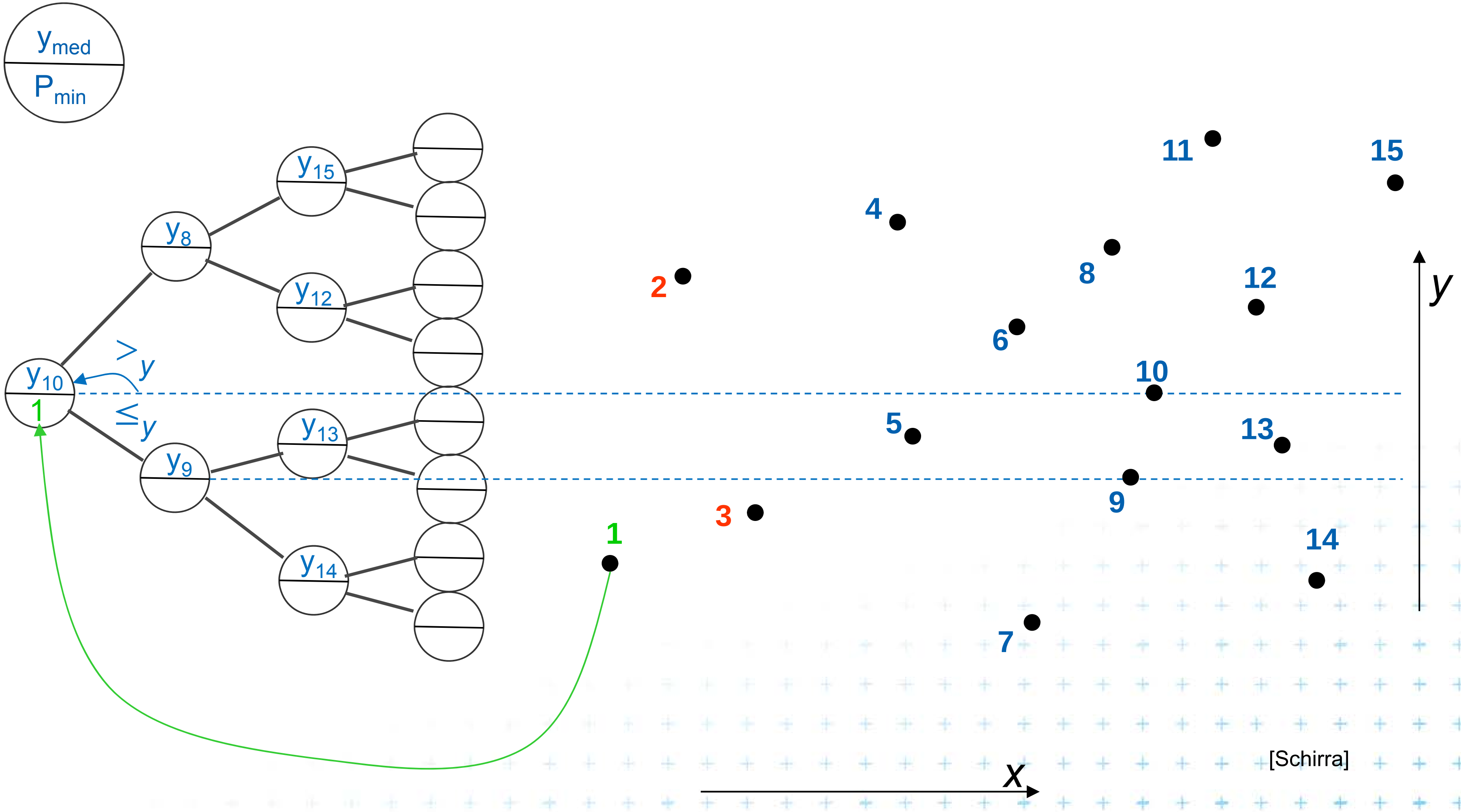
[Schirra]

Priority search tree construction example



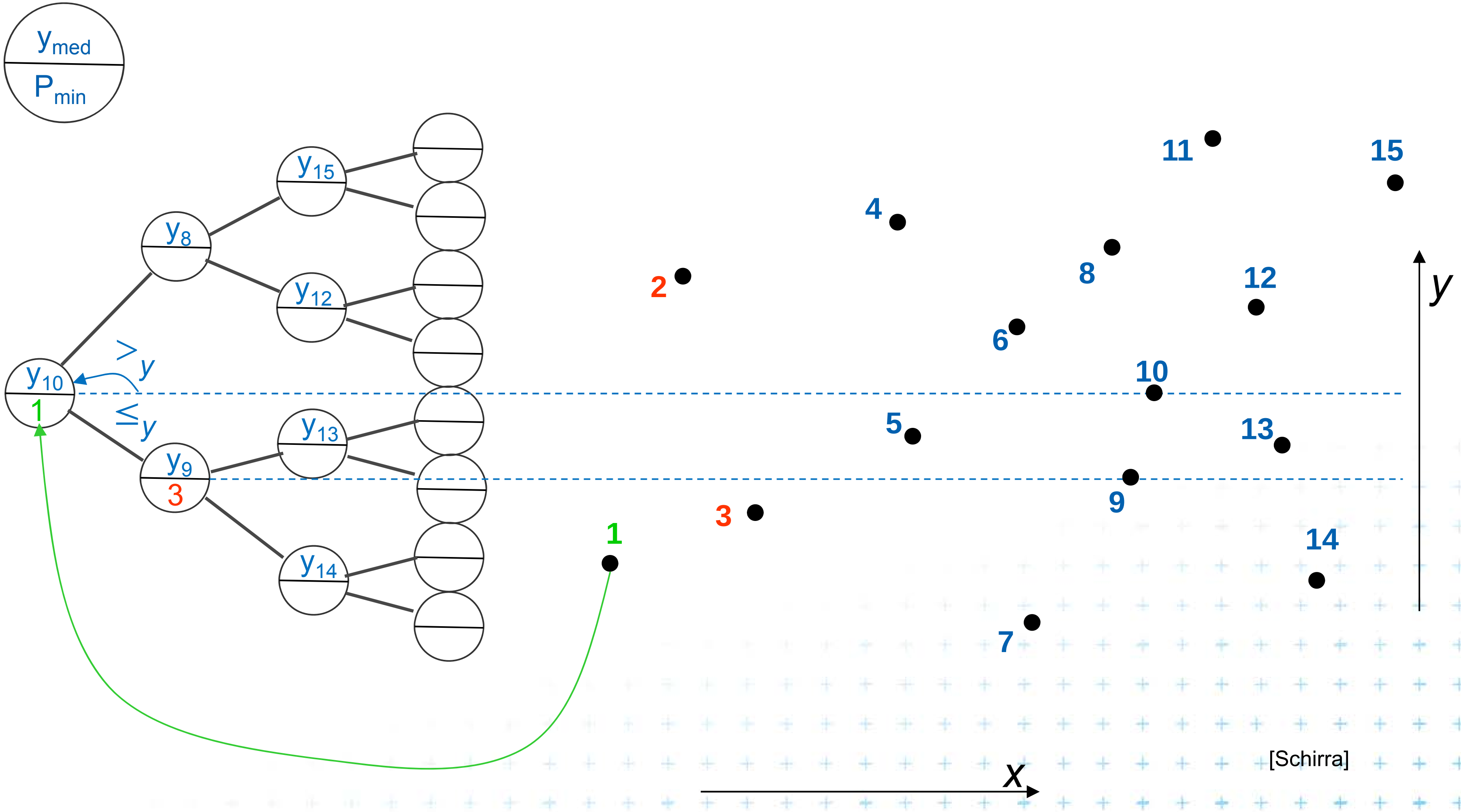
[Schirra]

Priority search tree construction example



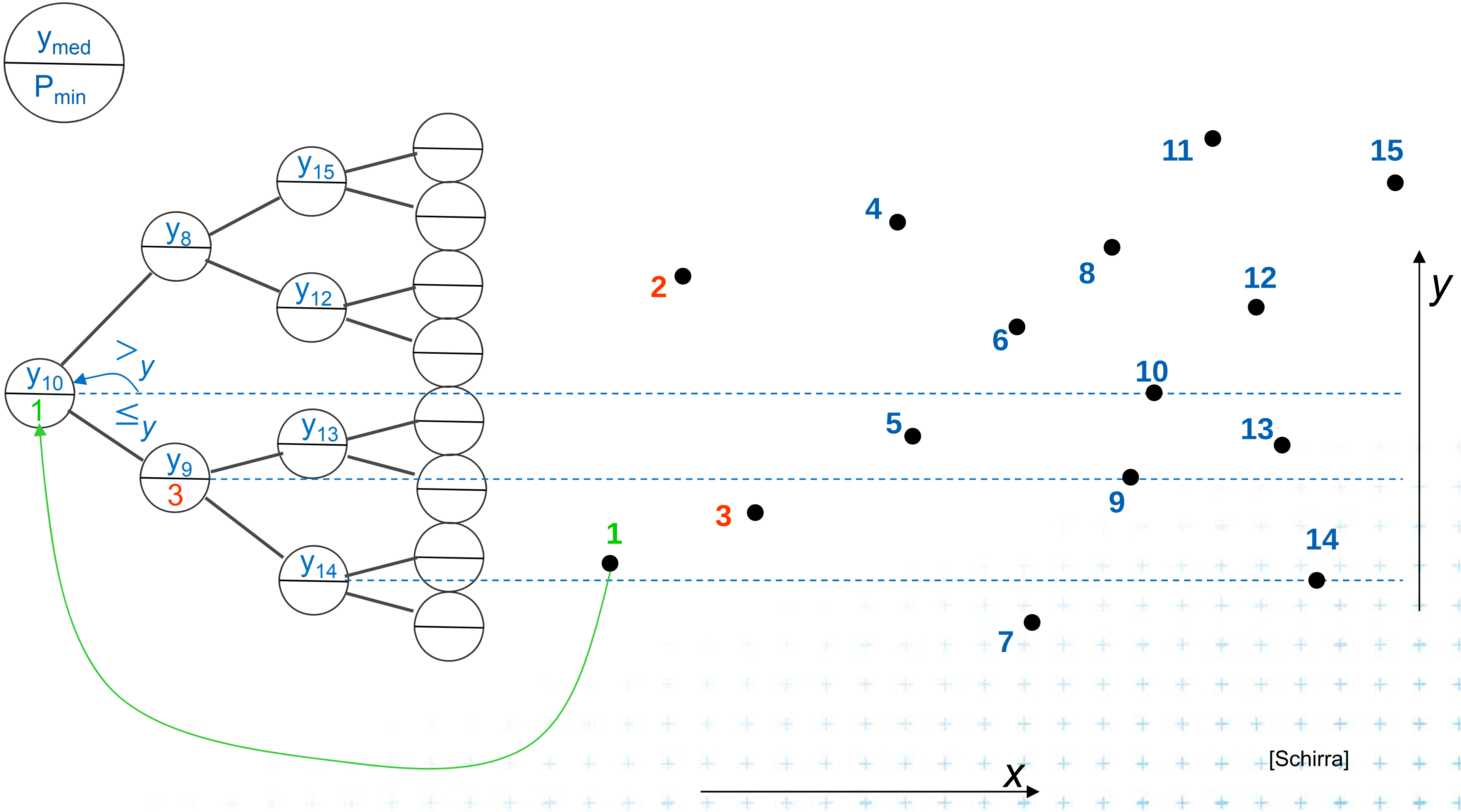
[Schirra]

Priority search tree construction example



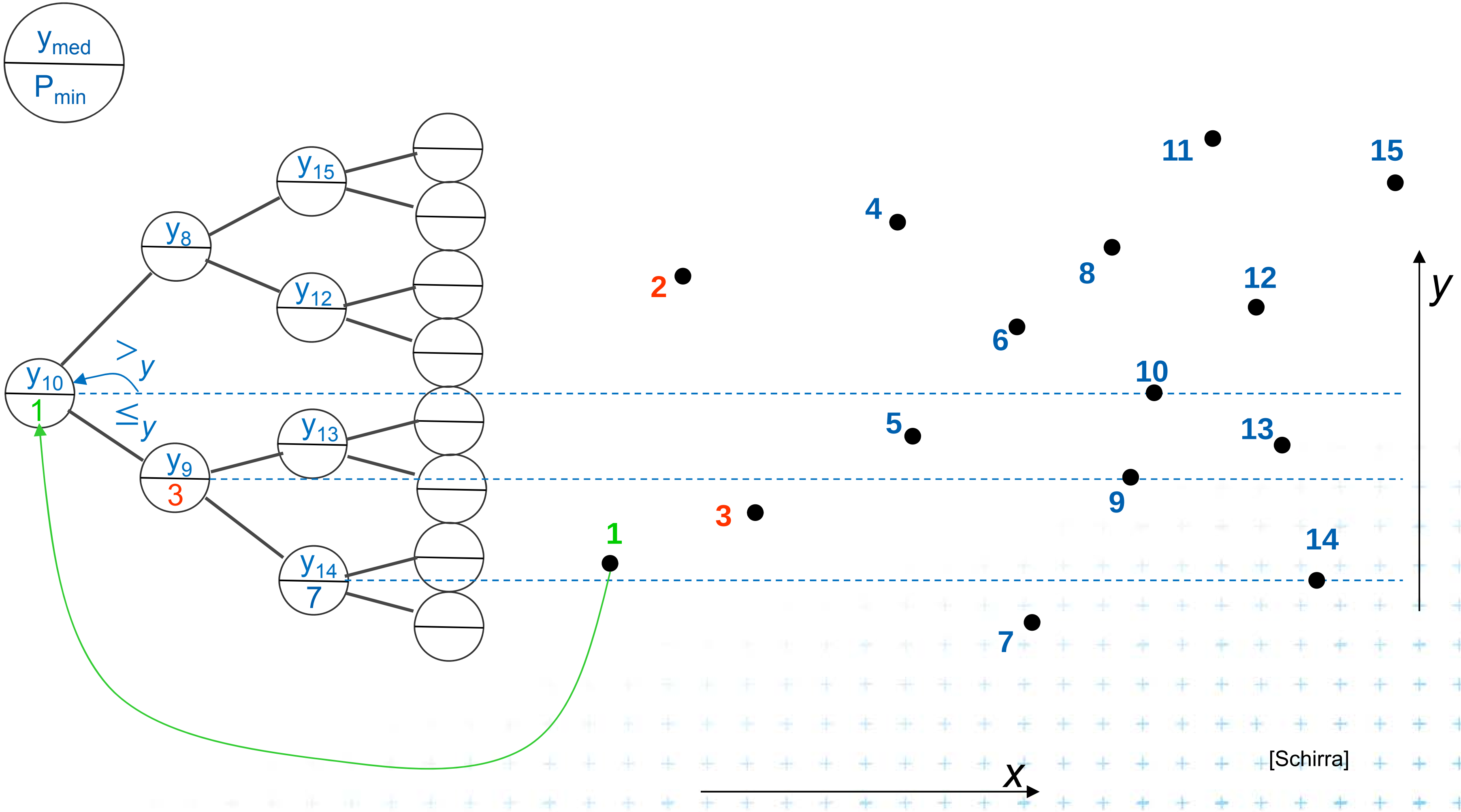
[Schirra]

Priority search tree construction example



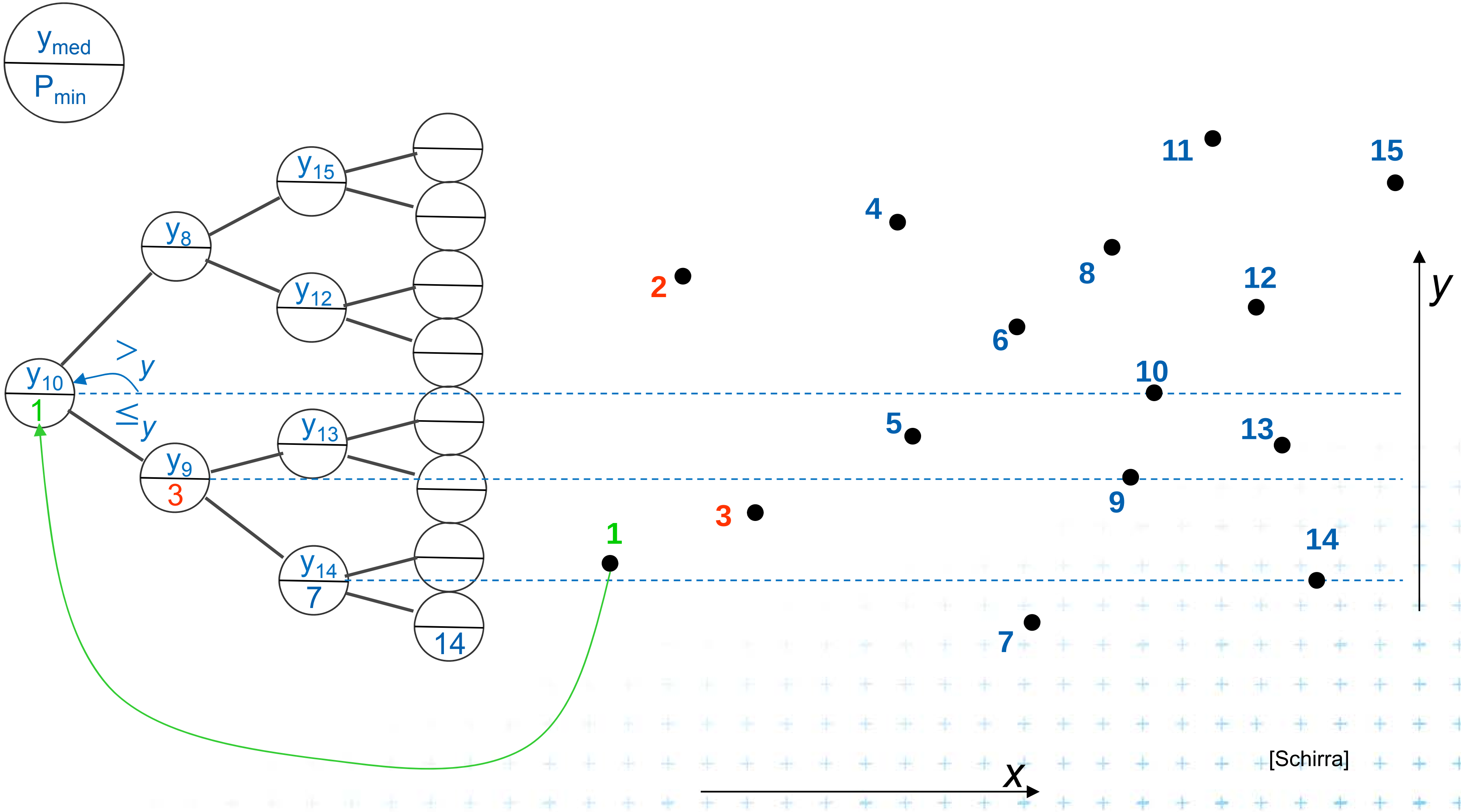
[Schirra]

Priority search tree construction example



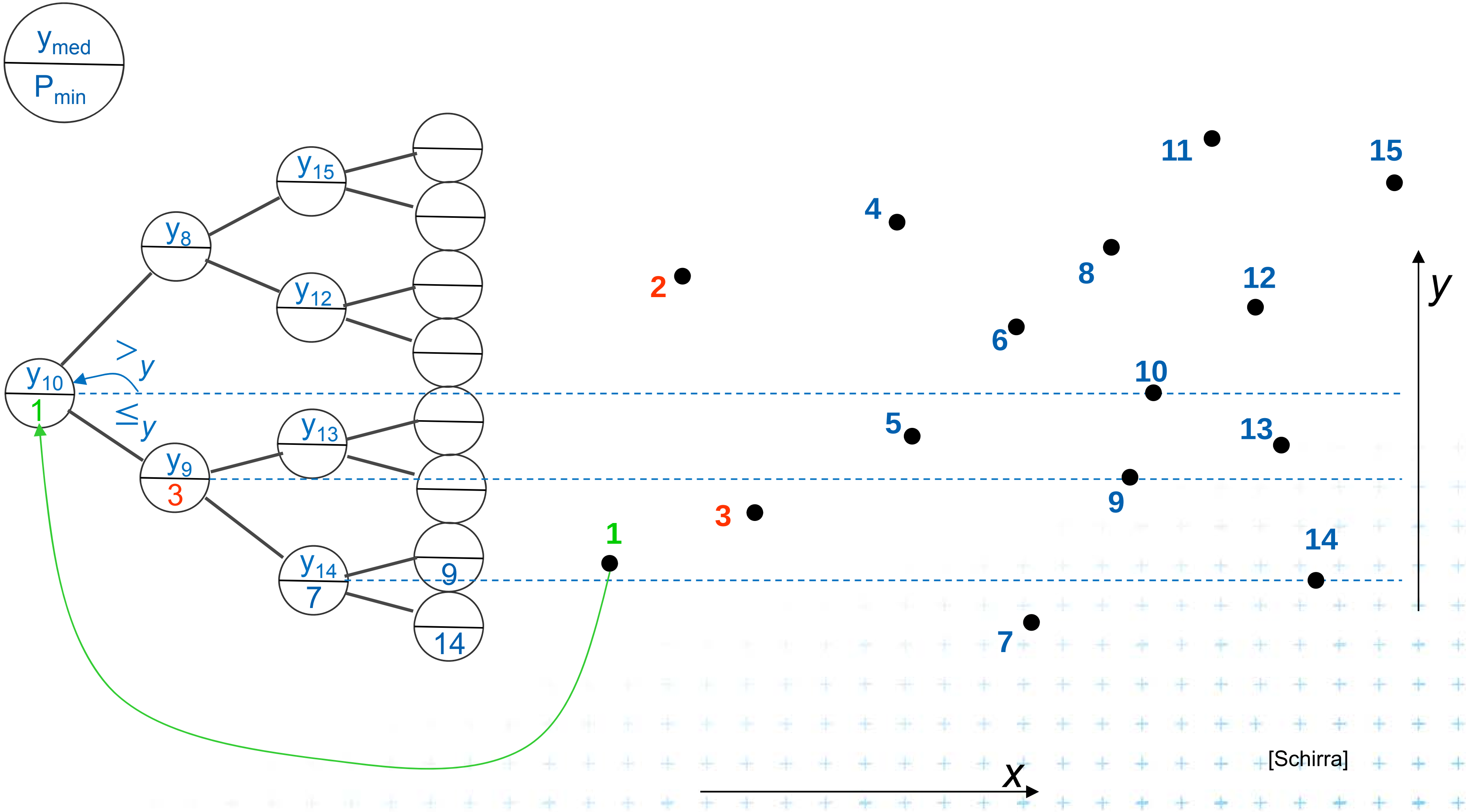
[Schirra]

Priority search tree construction example

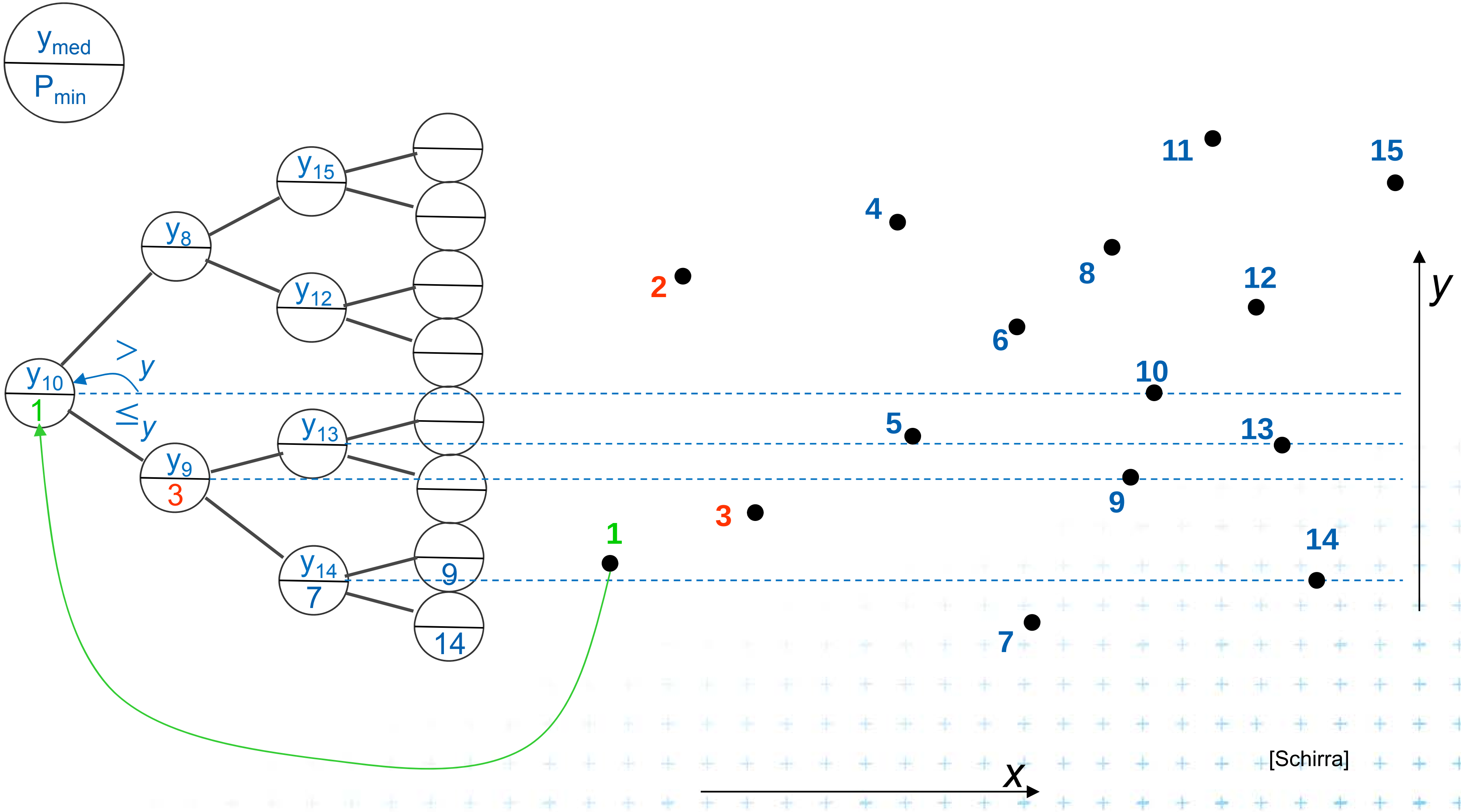


[Schirra]

Priority search tree construction example

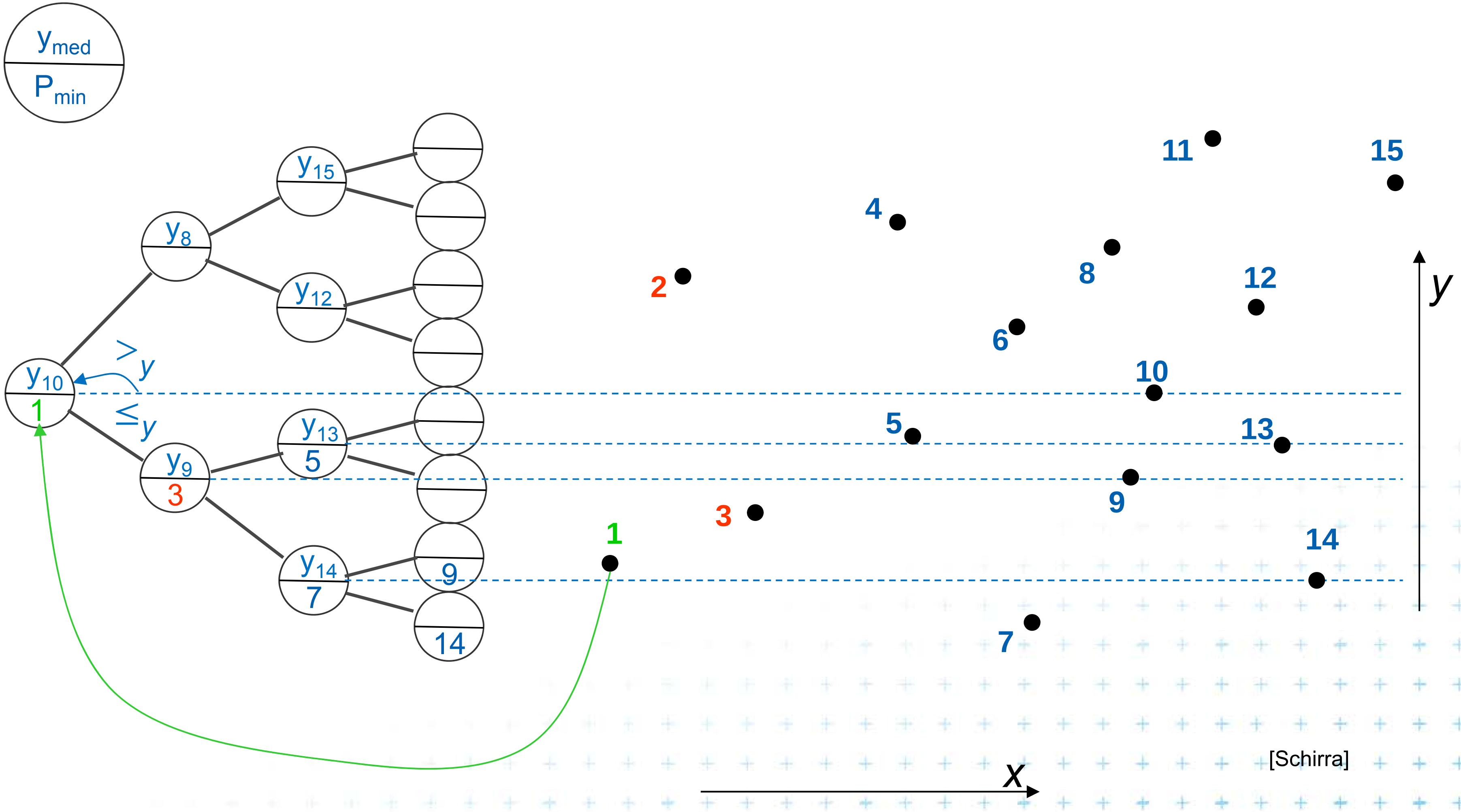


Priority search tree construction example

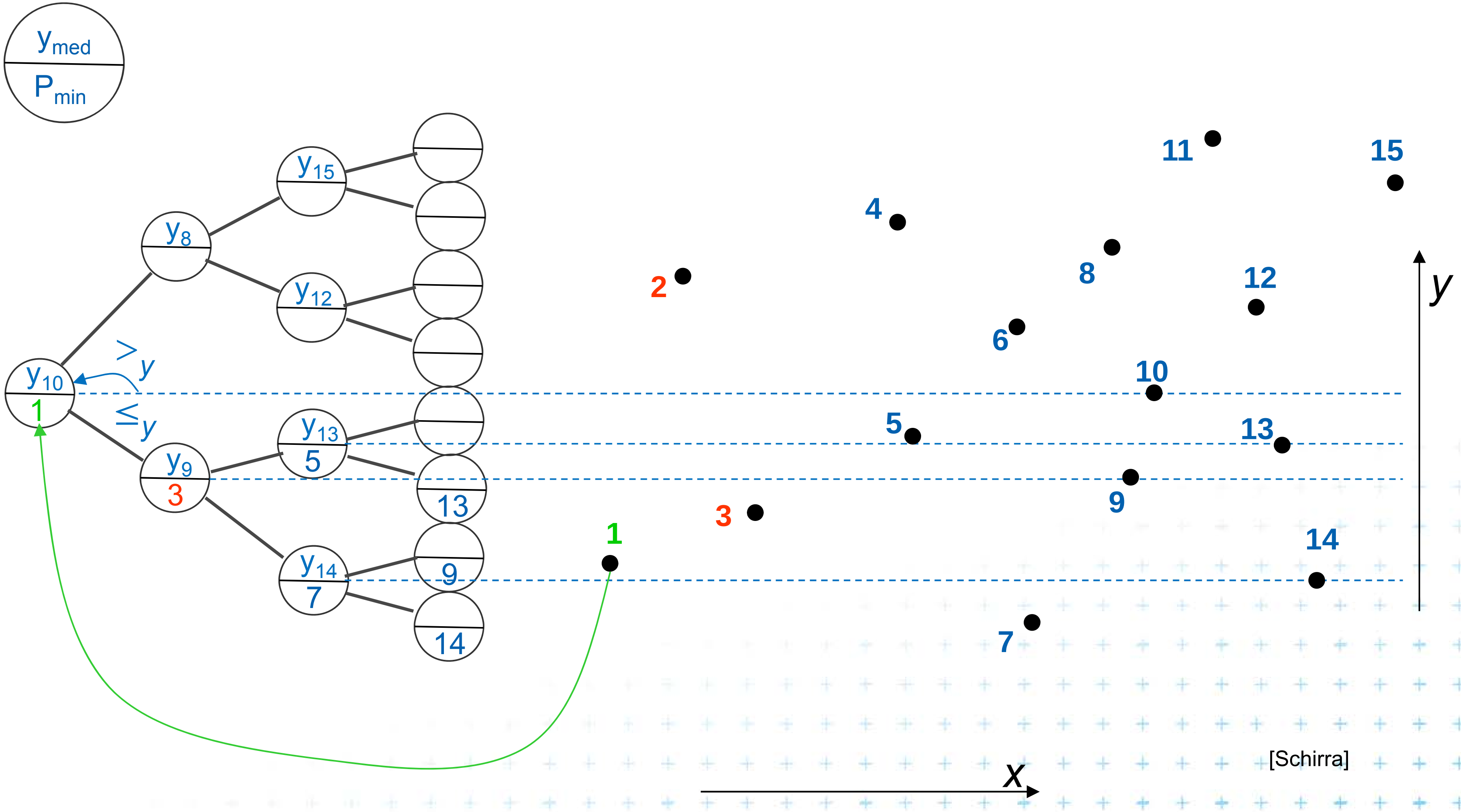


[Schirra]

Priority search tree construction example

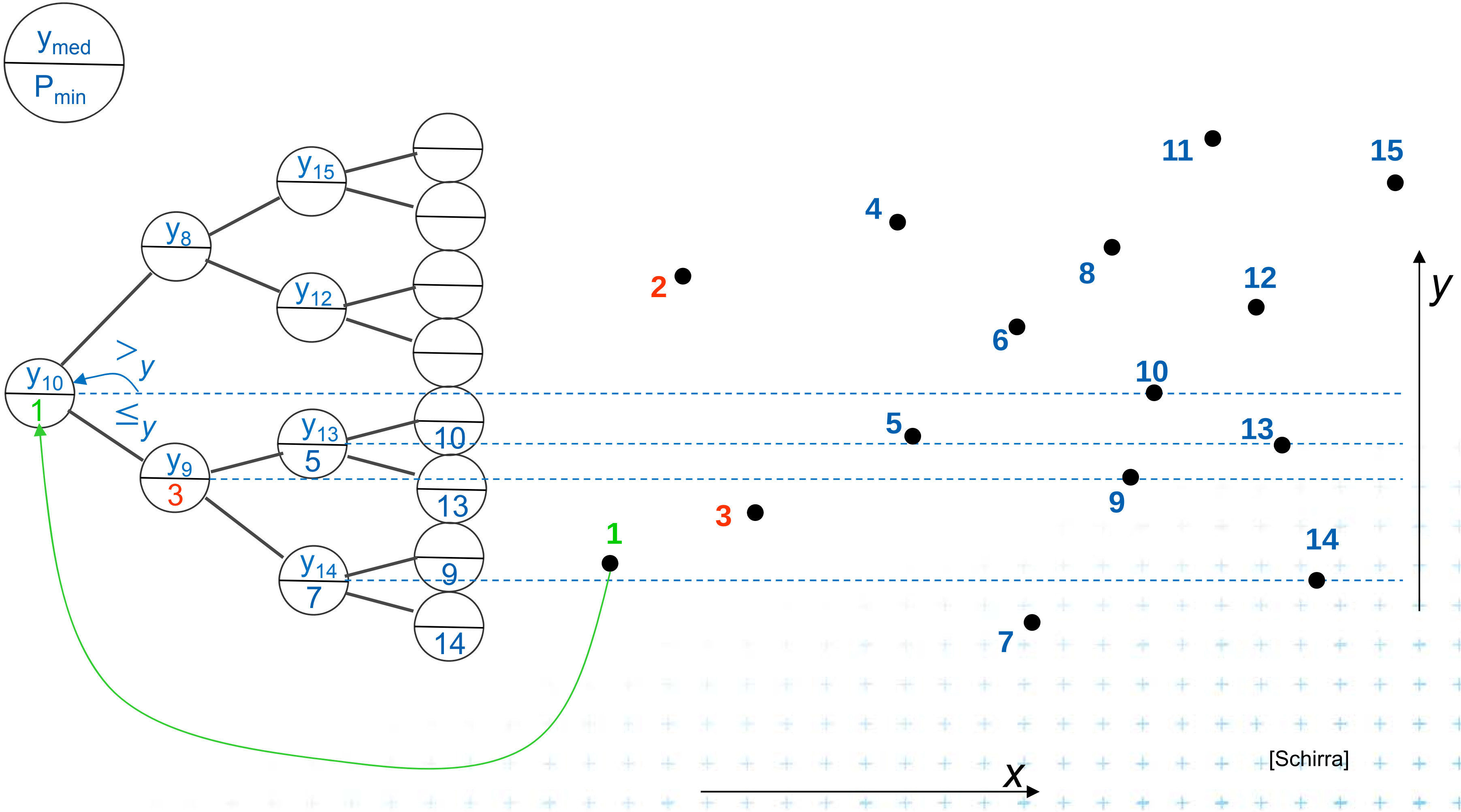


Priority search tree construction example



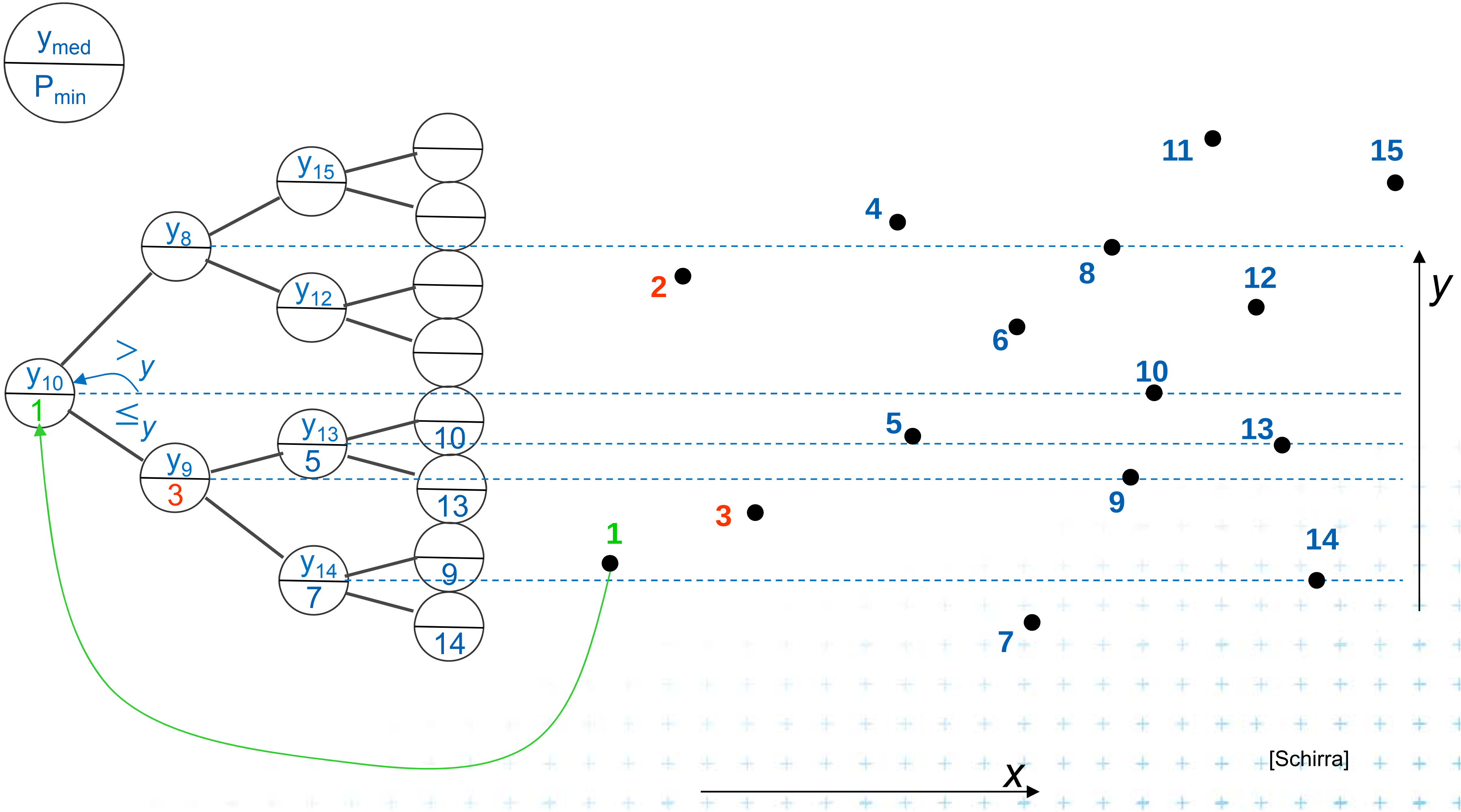
[Schirra]

Priority search tree construction example



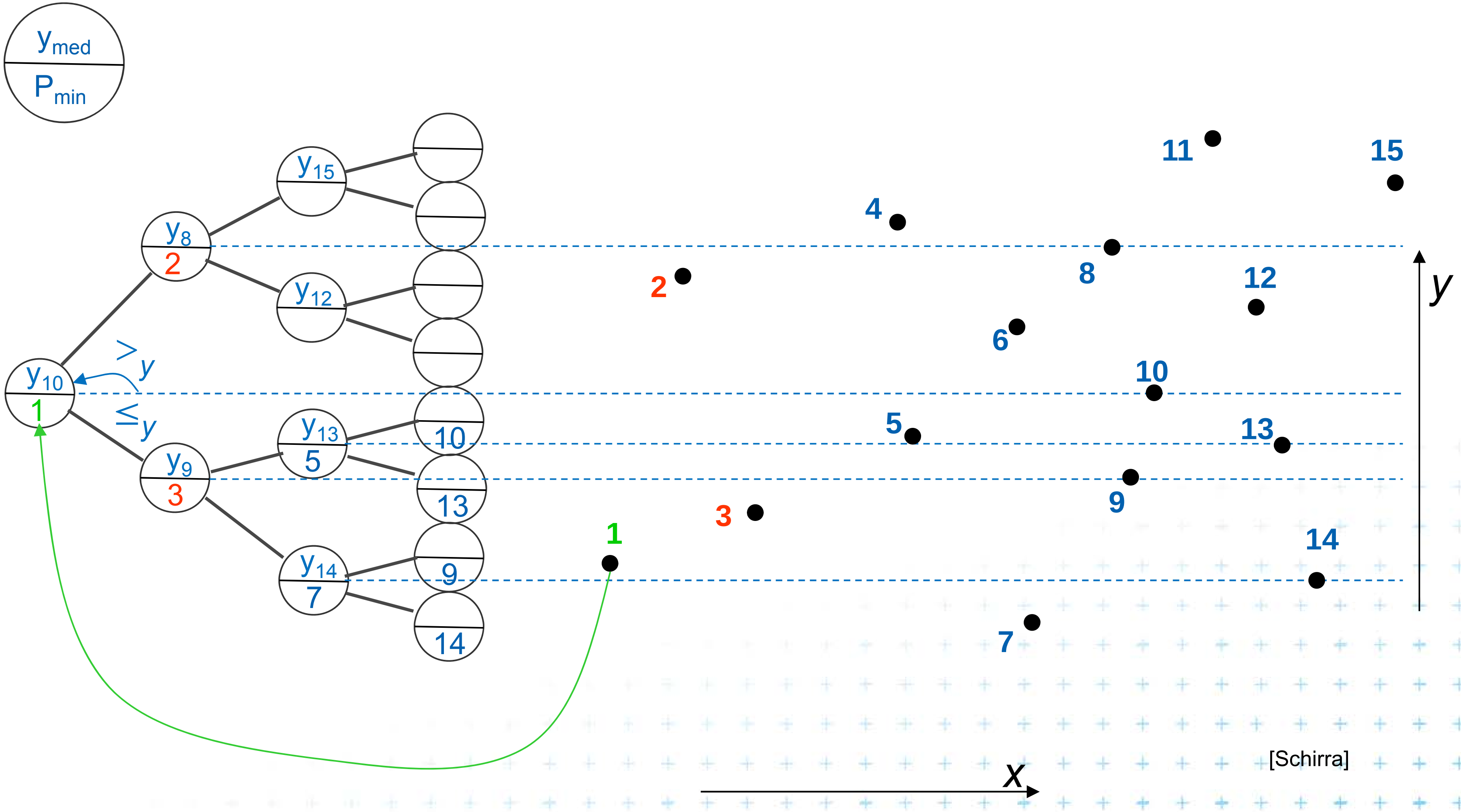
[Schirra]

Priority search tree construction example



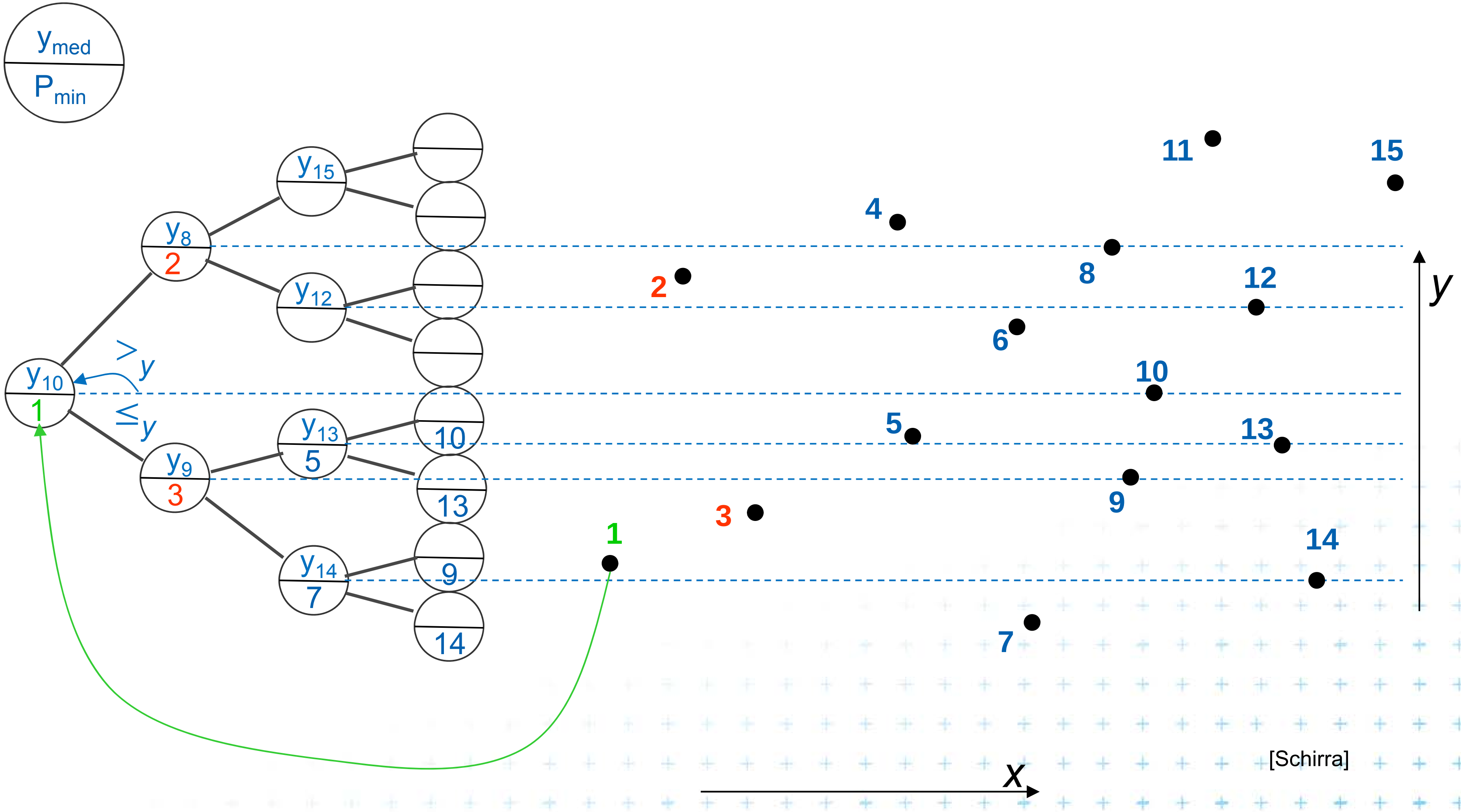
[Schirra]

Priority search tree construction example



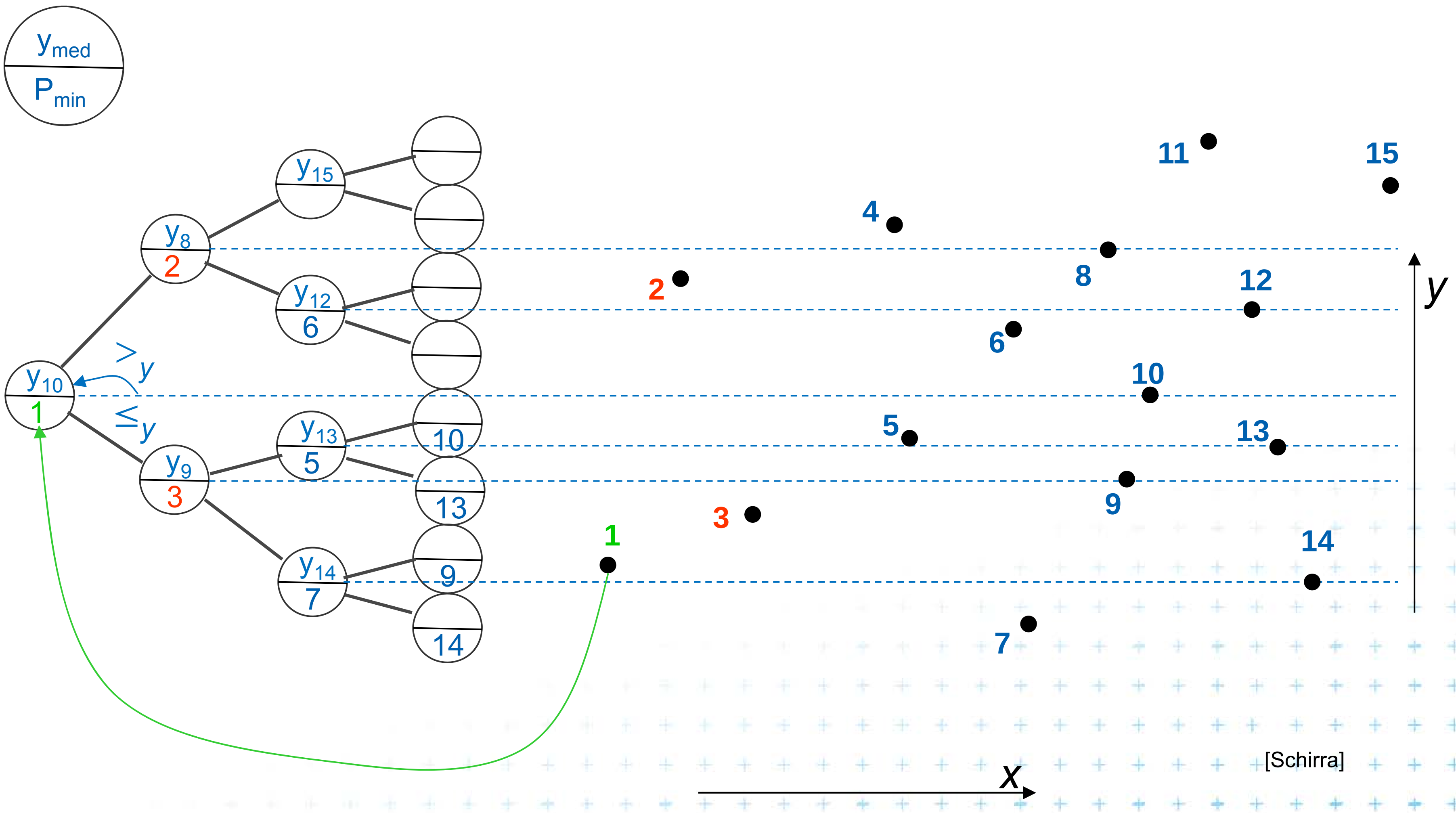
[Schirra]

Priority search tree construction example



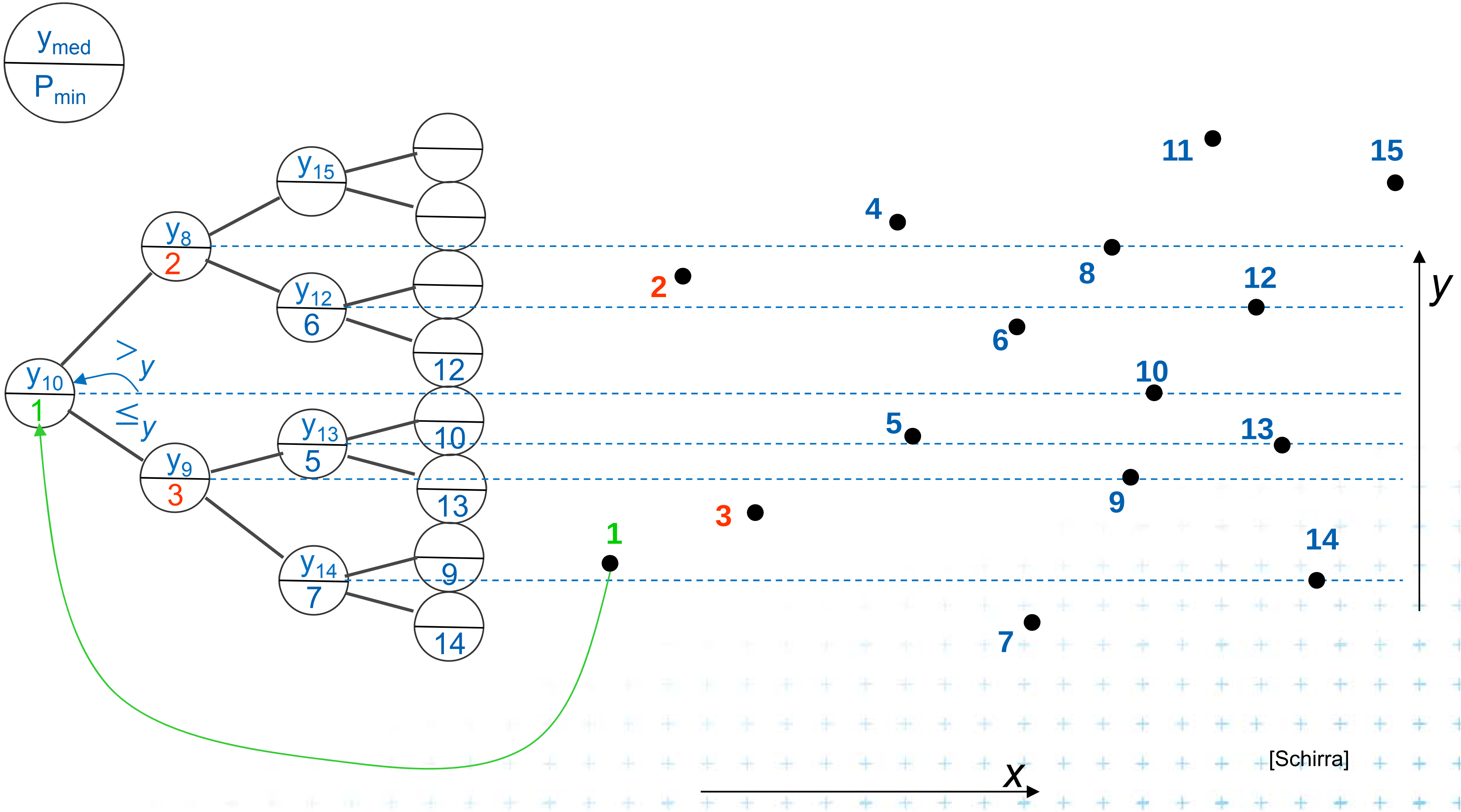
[Schirra]

Priority search tree construction example



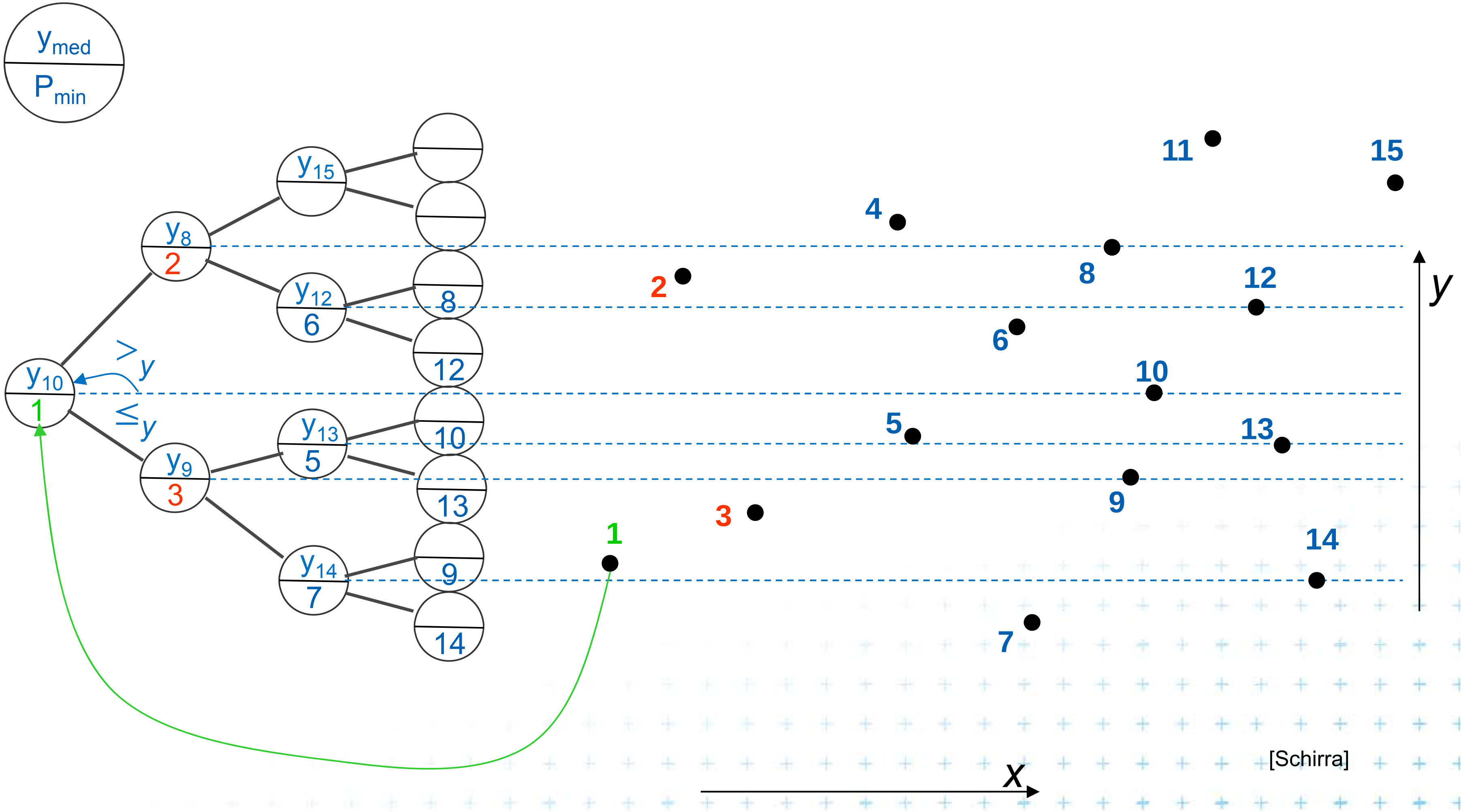
[Schirra]

Priority search tree construction example



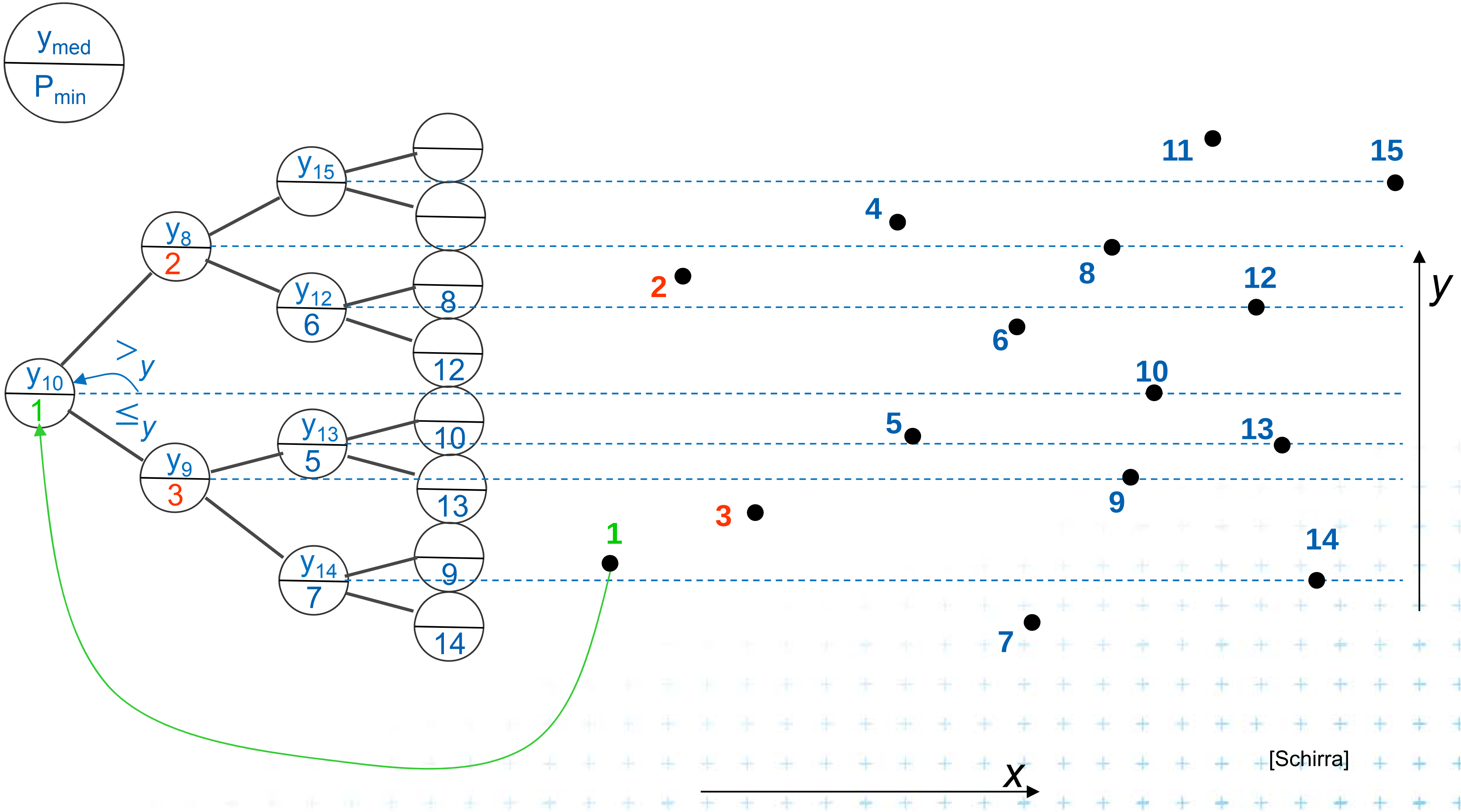
[Schirra]

Priority search tree construction example



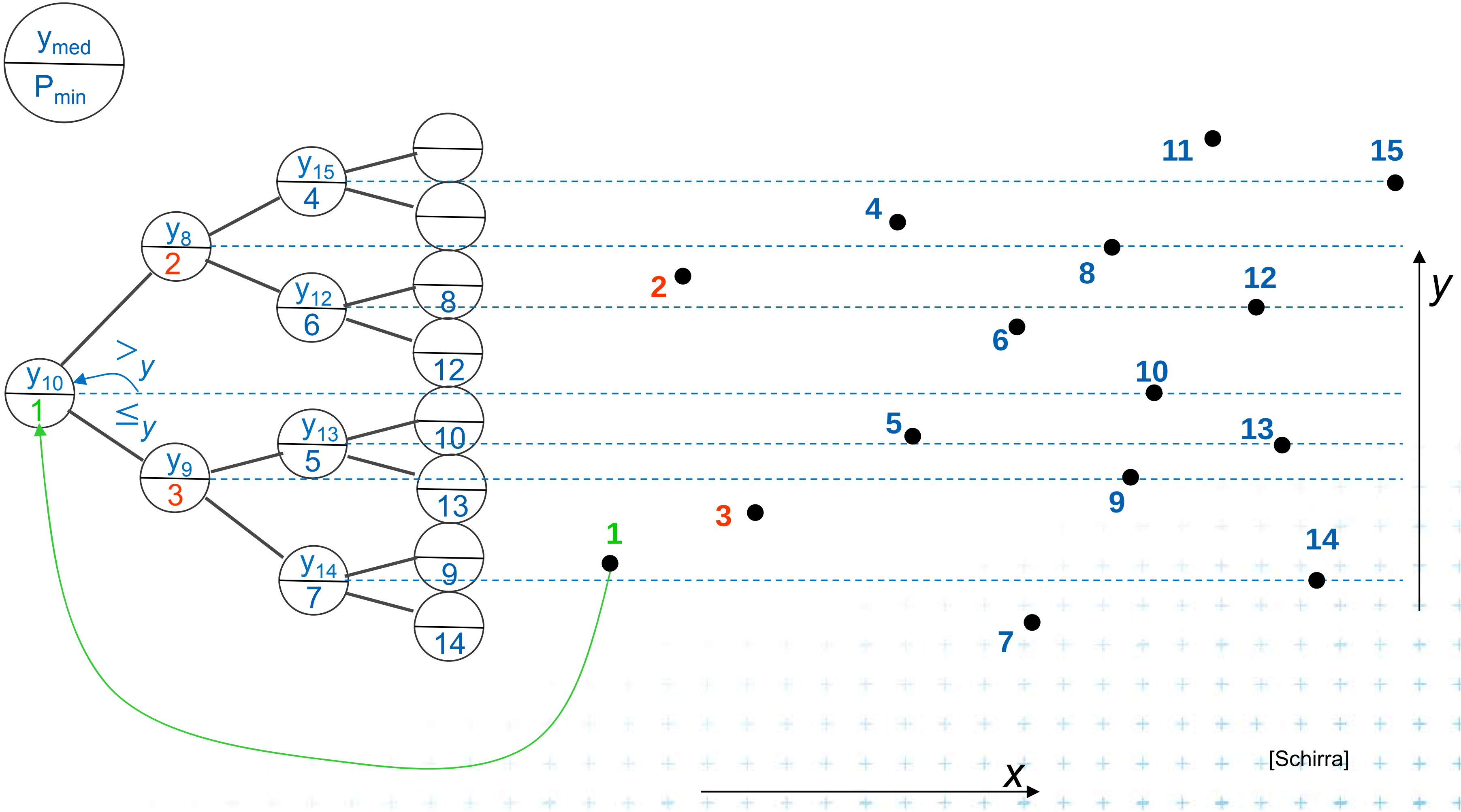
[Schirra]

Priority search tree construction example



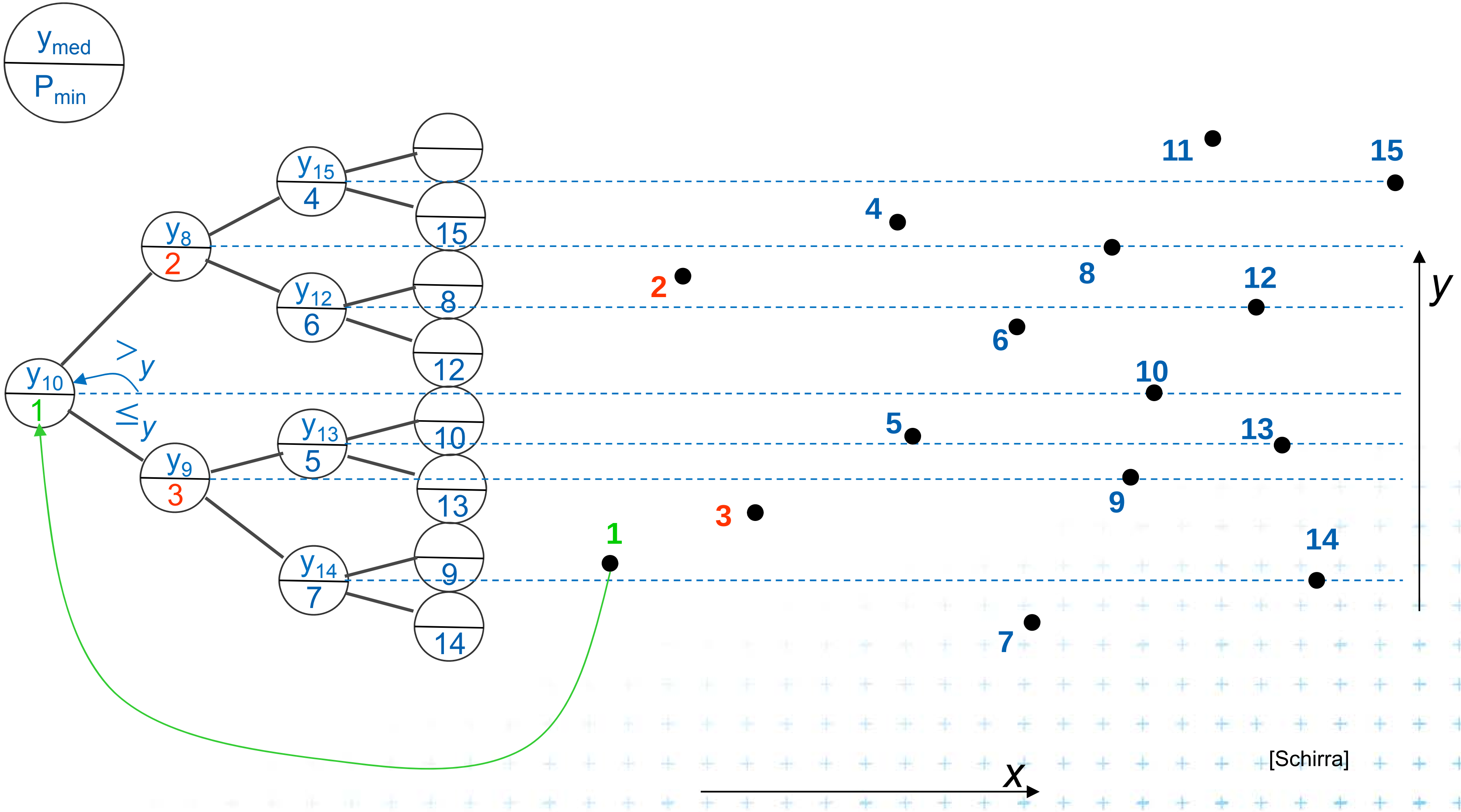
[Schirra]

Priority search tree construction example



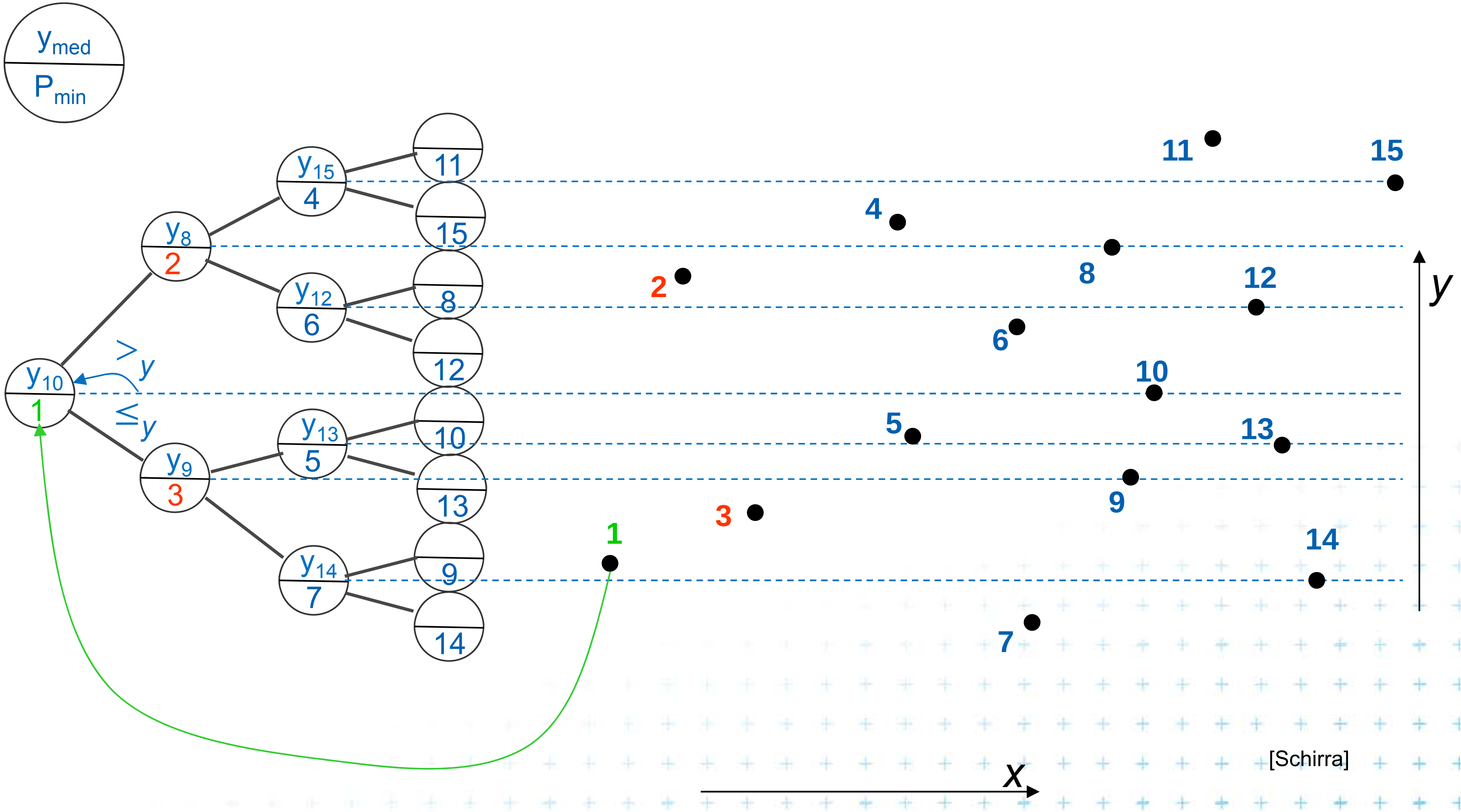
[Schirra]

Priority search tree construction example



[Schirra]

Priority search tree construction example



[Schirra]

Priority search tree construction

```
PrioritySearchTree( P )
Input:    set P of points in plane
Output:   priority search tree  T
if $P=\emptyset$ then PST is an empty leaf
else
    $p_{min}$  = point with smallest x-coordinate in $P$        // heap on x root
    $y_{med}$  = $y$-coord. median of points $P \setminus \{p_{min}\}$      // BST on y root
    Split points $P \setminus \{p_{min}\}$ into two subsets – according to $y_{med}$
            $P_{below} := \{ p \in P \setminus \{p_{min}\} : p_y \leq y_{med} \}$
            $P_{above} := \{ p \in P \setminus \{p_{min}\} : p_y > y_{med} \}$
    $T$ = newTreeNode()                                       Notation in alg:
    $T.p = p_{min}$      // point [ $x, y$ ]                          … p(v)
    $T.y = y_{mid}$      // skalar                                   … y(v)
    $T.left$  = PrioritySearchTree( $P_{below}$ )                     … lc(v)
    $T.rigft$ = PrioritySearchTree( $P_{above}$ )                     … rc(v)
```

13. O(n log n) , but O(n) if presorted on y-coordinate and bottom up

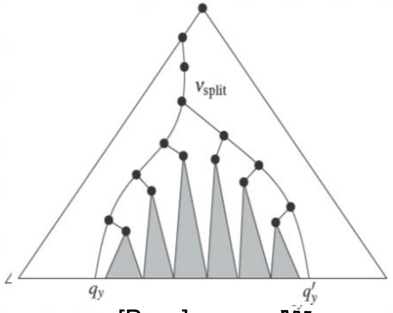
Query Priority Search Tree

QueryPrioritySearchTree(T, $(-\infty : q_x] \times [q_y ; q'_y]$)

Input: A priority search tree and a range, unbounded to the left

Output: All points lying in the range

```
Search with $q_y$ and $q'_y$ in $T$      // BST on $y$-coordinate – select $y$ range
Let $v_{split}$ be the node where the two search paths split (split node)

for each node $v$ on the search path of $q_y$ or $q'_y$   // points along the paths
    if $p(v) \in (-\infty : q_x] \times [q_y ; q'_y]$ then report $p(v)$  // starting in tree root

for each node $v$ on the path of $q_y$ in the left subtree of $v_{split}$ // inner trees
    if the search path goes left at $v$
        ReportInSubtree( $rc(v)$, $q_x$ )   // report right subtree
for each node $v$ on the path of $q'_y$ in right subtree of $v_{split}$
    if the search path goes right at $v$
        ReportInSubtree( $lc(v)$, $q_x$ )   // rep. left subtree
```

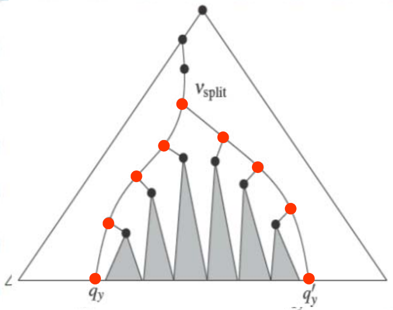


[Berg]

Query Priority Search Tree

QueryPrioritySearchTree(T, $(-\infty : q_x] \times [q_y\,;\, q'_y\,]$)
Input: A priority search tree and a range, unbounded to the left
Output: All points lying in the range

```
Search with $q_y$ and $q'_y$ in $T$      // BST on $y$-coordinate – select $y$ range
Let $v_{split}$ be the node where the two search paths split (split node)

for each node $v$ on the search path of $q_y$ or $q'_y$   // points along the paths
    if $p(v) \in (-\infty : q_x] \times [q_y\,;\, q'_y\,]$ then report $p(v)$  // starting in tree root

for each node $v$ on the path of $q_y$ in the left subtree of $v_{split}$ // inner trees
    if the search path goes left at $v$
        ReportInSubtree( $rc(v)$, $q_x$ )   // report right subtree
for each node $v$ on the path of $q'_y$ in right subtree of $v_{split}$
    if the search path goes right at $v$
        ReportInSubtree( $lc(v)$, $q_x$ )   // rep. left subtree
```



[Berg]

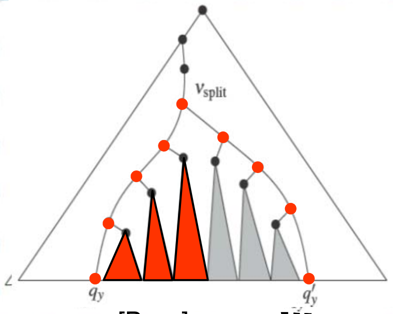
Query Priority Search Tree

QueryPrioritySearchTree(T, $(-\infty : q_x] \times [q_y ; q'_y]$)

Input: A priority search tree and a range, unbounded to the left

Output: All points lying in the range

```
Search with $q_y$ and $q'_y$ in $T$      // BST on $y$-coordinate – select $y$ range
Let $v_{split}$ be the node where the two search paths split (split node)

for each node $v$ on the search path of $q_y$ or $q'_y$   // points along the paths
    if $p(v) \in (-\infty : q_x] \times [q_y ; q'_y ]$ then report $p(v)$  // starting in tree root

for each node $v$ on the path of $q_y$ in the left subtree of $v_{split}$ // inner trees
    if the search path goes left at $v$
        ReportInSubtree( $rc(v)$, $q_x$ )   // report right subtree
for each node $v$ on the path of $q'_y$ in right subtree of $v_{split}$
    if the search path goes right at $v$
        ReportInSubtree( $lc(v)$, $q_x$ )   // rep. left subtree
```



[Berg]

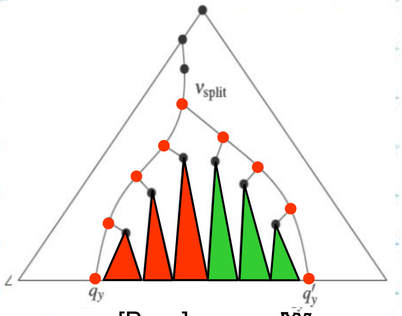
Query Priority Search Tree

QueryPrioritySearchTree(T, $(-\infty : q_x] \times [q_y\, ;\, q'_y\,]$)

Input: A priority search tree and a range, unbounded to the left

Output: All points lying in the range

```
Search with $q_y$ and $q'_y$ in $T$      // BST on $y$-coordinate – select $y$ range
Let $v_{split}$ be the node where the two search paths split (split node)

for each node $v$ on the search path of $q_y$ or $q'_y$   // points along the paths
    if $p(v) \in (-\infty : q_x] \times [q_y\, ;\, q'_y\,]$ then report $p(v)$  // starting in tree root

for each node $v$ on the path of $q_y$ in the left subtree of $v_{split}$ // inner trees
    if the search path goes left at $v$
        ReportInSubtree( $rc(v)$, $q_x$ )   // report right subtree
for each node $v$ on the path of $q'_y$ in right subtree of $v_{split}$
    if the search path goes right at $v$
        ReportInSubtree( $lc(v)$, $q_x$ )   // rep. left subtree
```



[Berg]

Reporting of subtrees between the paths

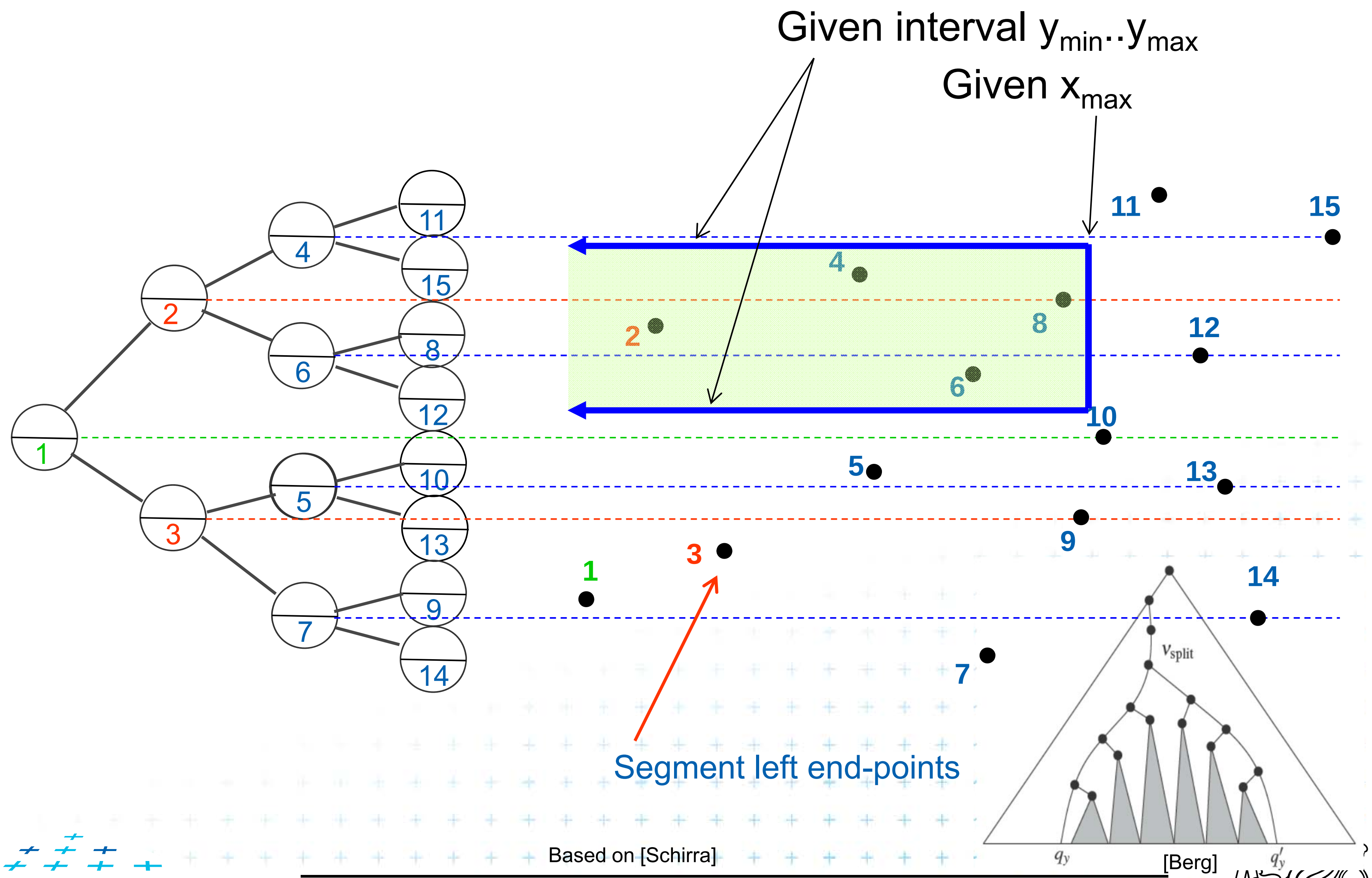
ReportInSubtree(v, q_x)

Input: The root v of a subtree of a priority search tree and a value q_x.

Output: All points in the subtree with x-coordinate at most q_x.

```
1. if $v$ is not a leaf and $x( p(v) ) \le q_x$                // $x \in (-\infty : q_x]$ -- heap condition
2.     Report $p(v)$.
3.     ReportInSubtree( $lc(v)$, $q_x$ )
4.     ReportInSubtree( $rc(v)$, $q_x$ )
```

Priority search tree query



Given interval $y_{min}..y_{max}$

Given x_{max}

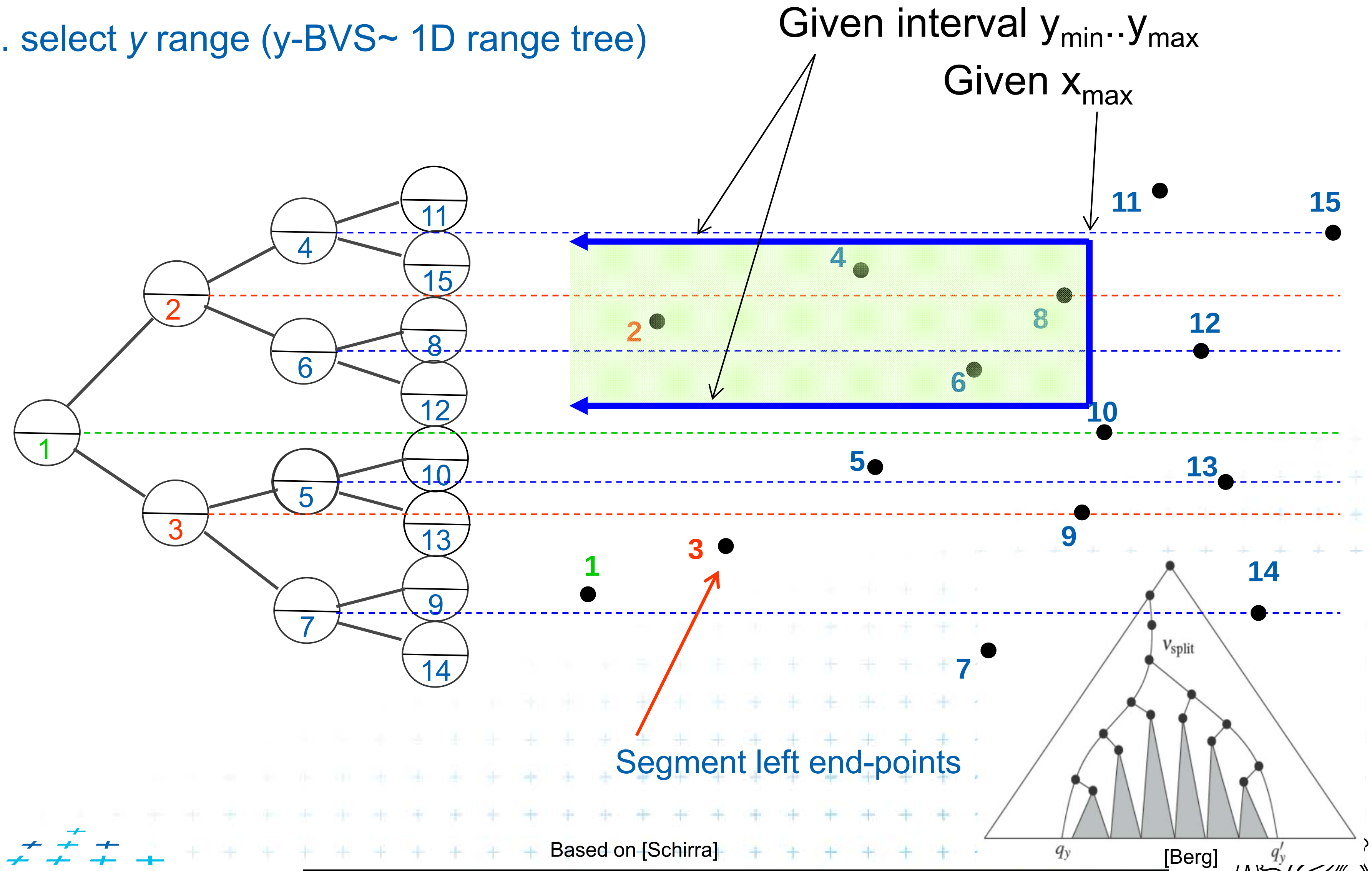
Segment left end-points

Based on [Schirra]

[Berg]

Priority search tree query

1. select *y* range (y-BVS~ 1D range tree)



Based on [Schirra]

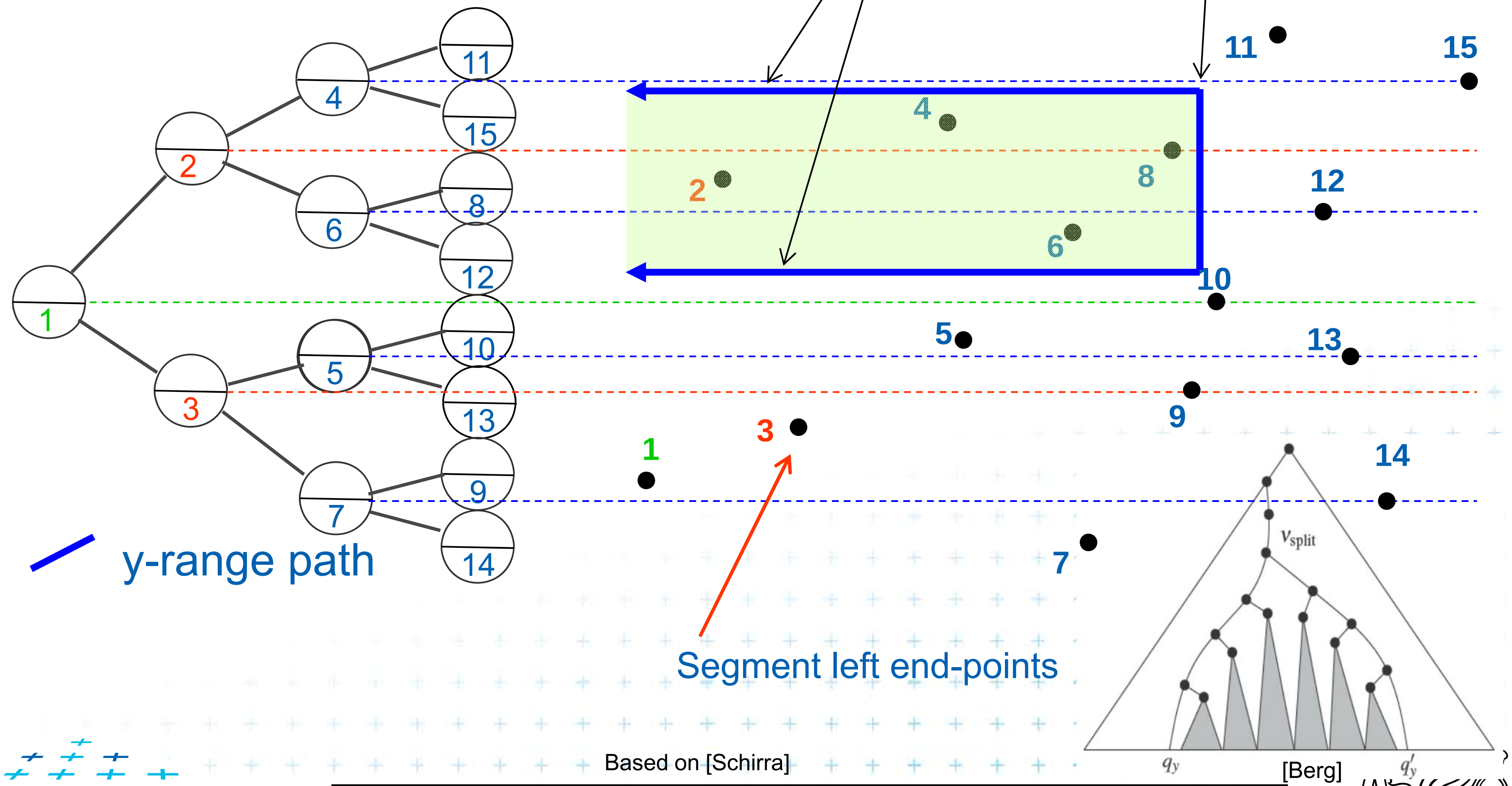
[Berg]

Priority search tree query

1. select *y* range (y-BVS~ 1D range tree)

Given interval $y_{min}..y_{max}$

Given x_{max}



y-range path

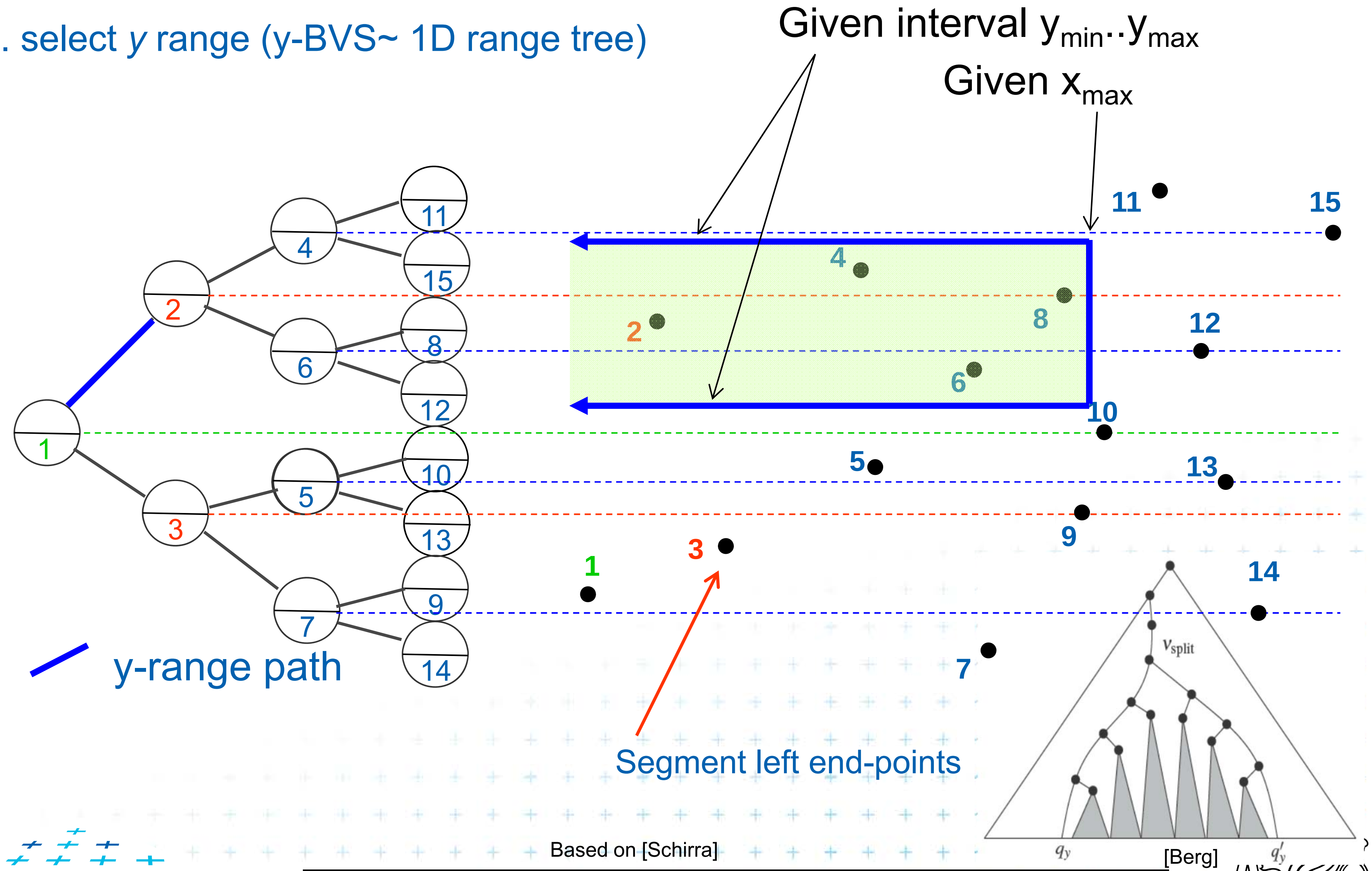
Segment left end-points

Based on [Schirra]

[Berg]

Priority search tree query

1. select *y* range (y-BVS~ 1D range tree)



Given interval $y_{min}..y_{max}$

Given x_{max}

y-range path

Segment left end-points

Based on [Schirra]

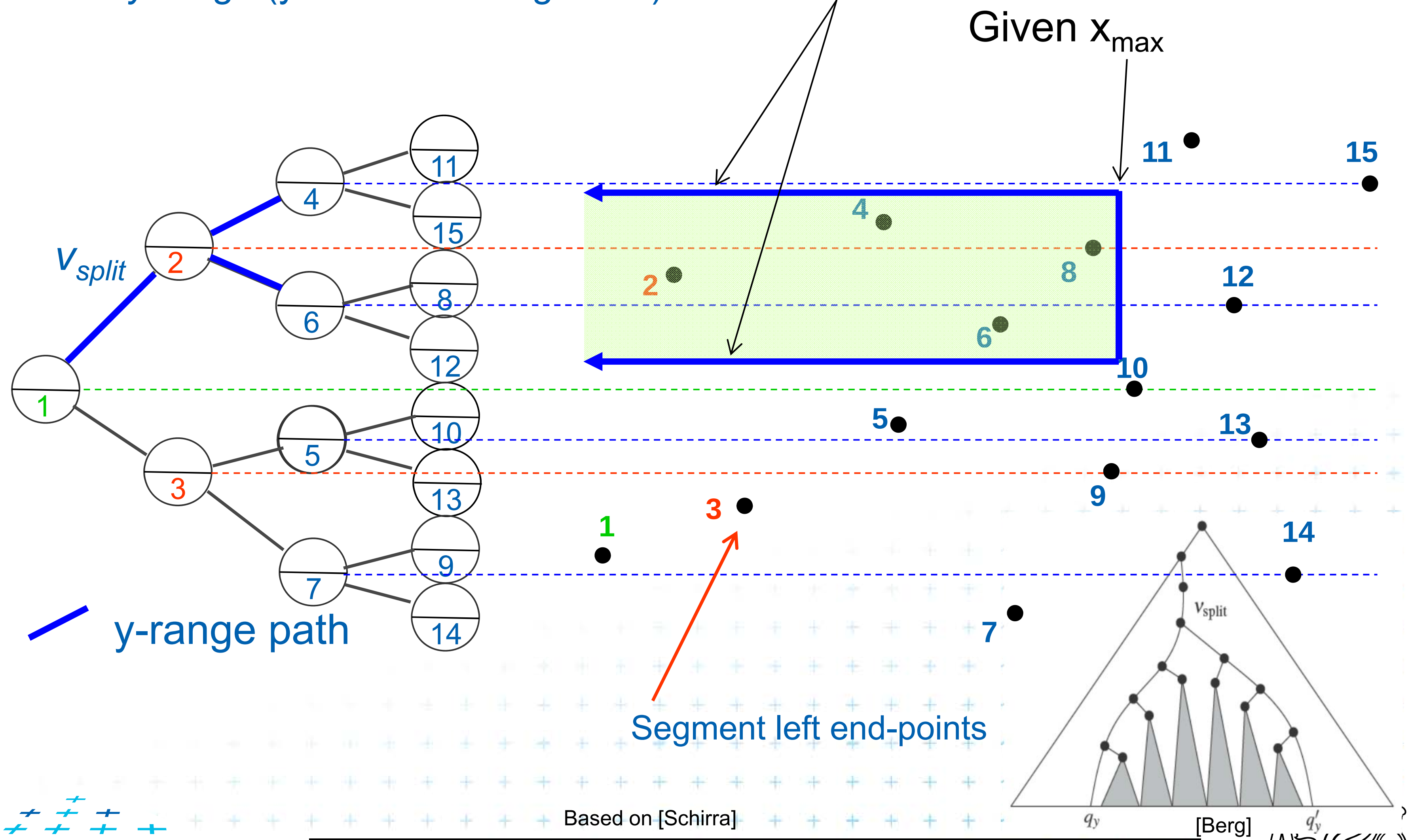
[Berg]

Priority search tree query

1. select *y* range (y-BVS~ 1D range tree)

Given interval $y_{min}..y_{max}$

Given x_{max}



y-range path

Segment left end-points

Based on [Schirra]

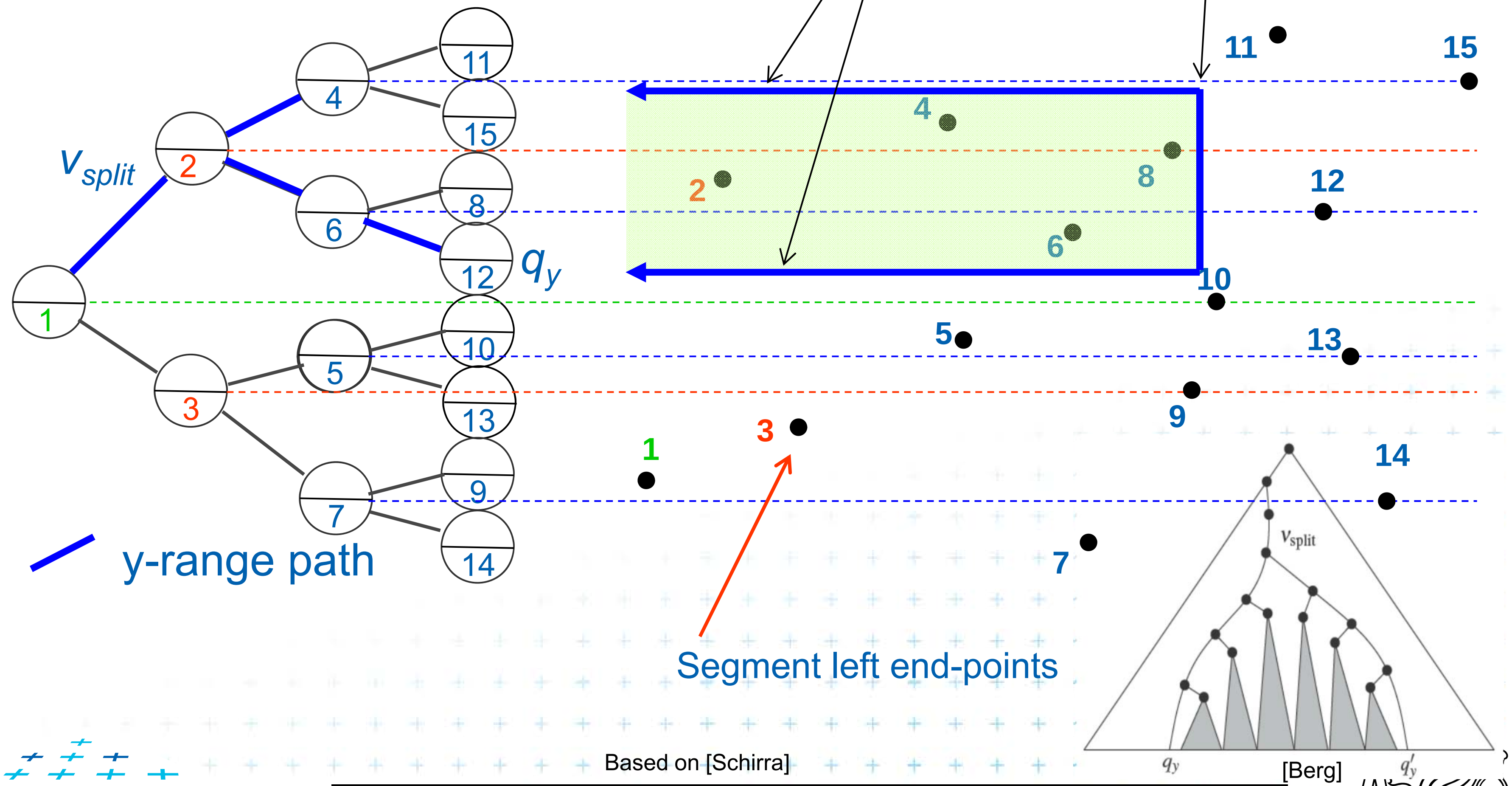
[Berg]

Priority search tree query

1. select *y* range (y-BVS~ 1D range tree)

Given interval $y_{min}..y_{max}$

Given x_{max}



y-range path

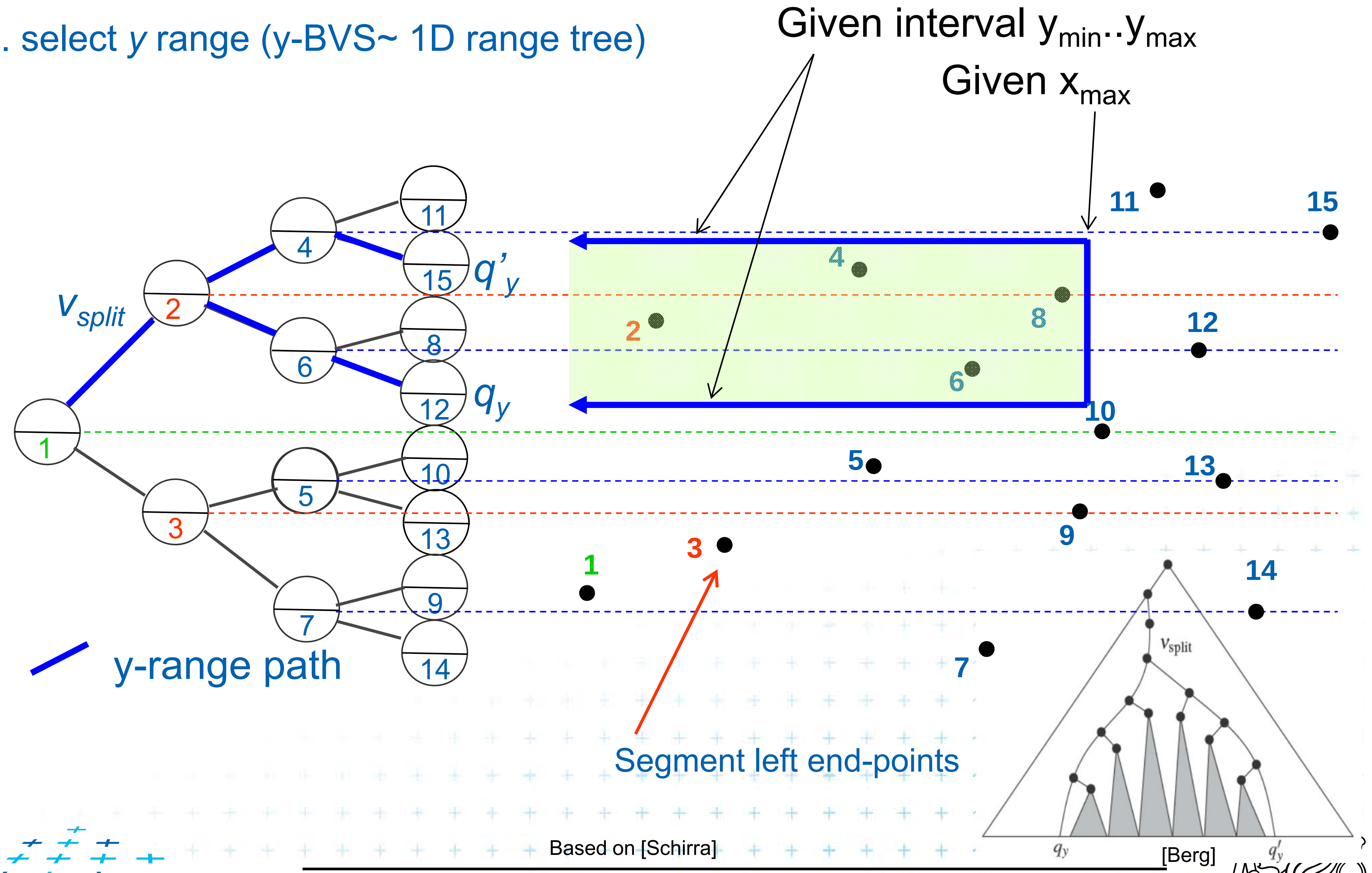
Segment left end-points

Based on [Schirra]

[Berg]

Priority search tree query

1. select *y* range (y-BVS~ 1D range tree)



Given interval $y_{min}..y_{max}$

Given x_{max}

y-range path

Segment left end-points

Based on [Schirra]

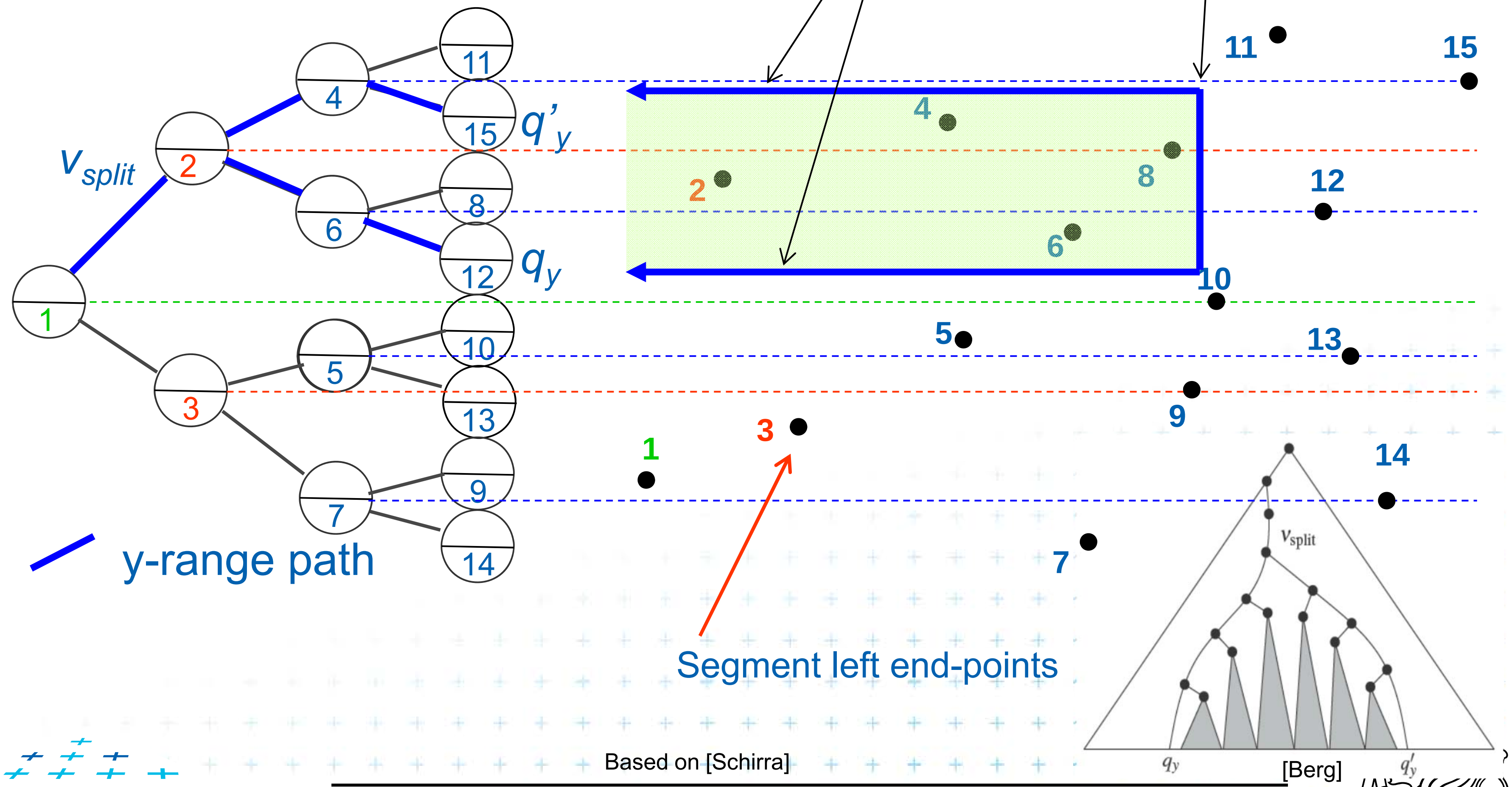
[Berg]

Priority search tree query

1. select *y* range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)

Given interval $y_{min}..y_{max}$

Given x_{max}



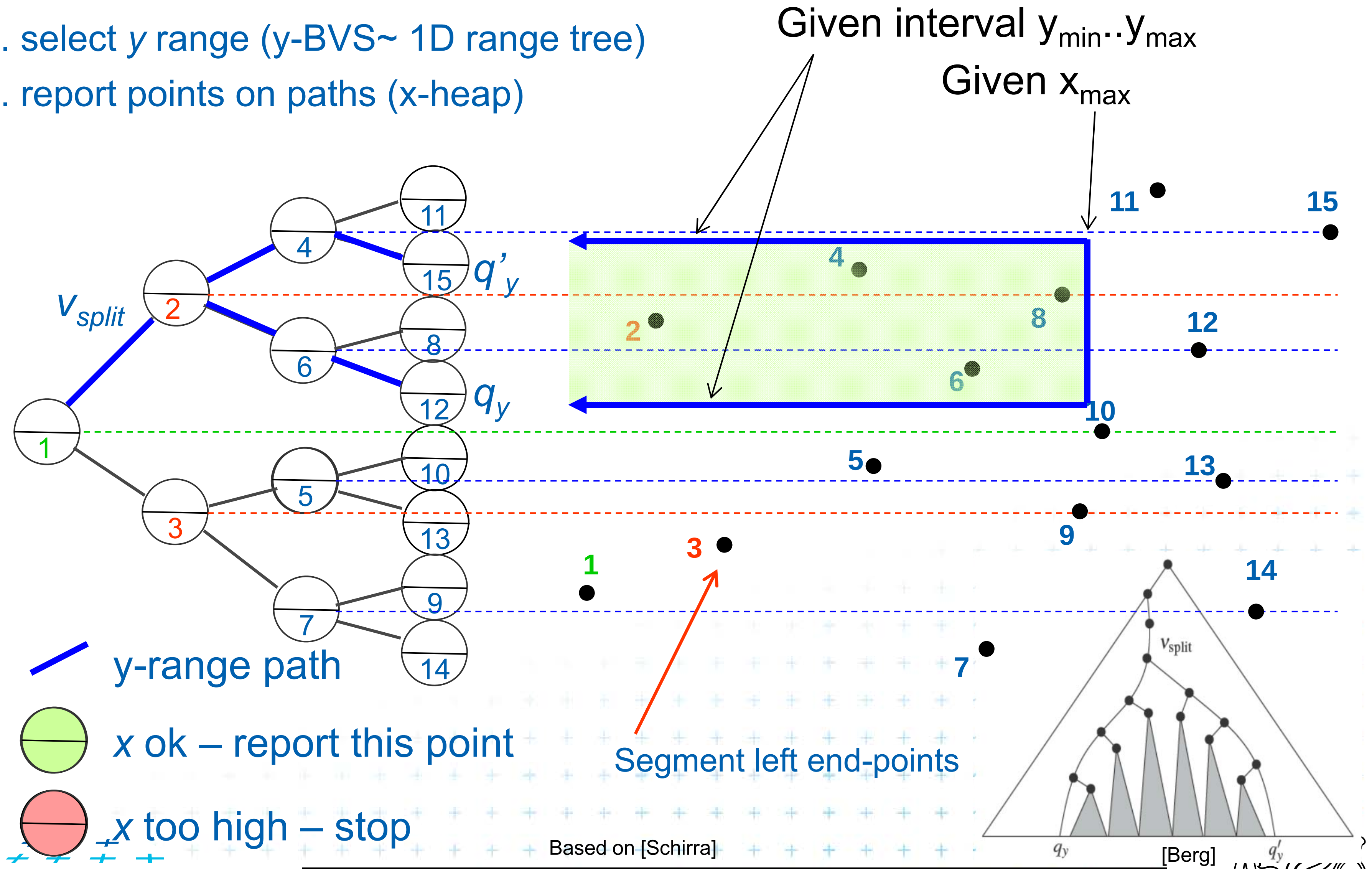
Segment left end-points

Based on [Schirra]

[Berg]

Priority search tree query

1. select *y* range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)



Given interval $y_{min}..y_{max}$

Given x_{max}

y-range path

🟩 *x* ok – report this point

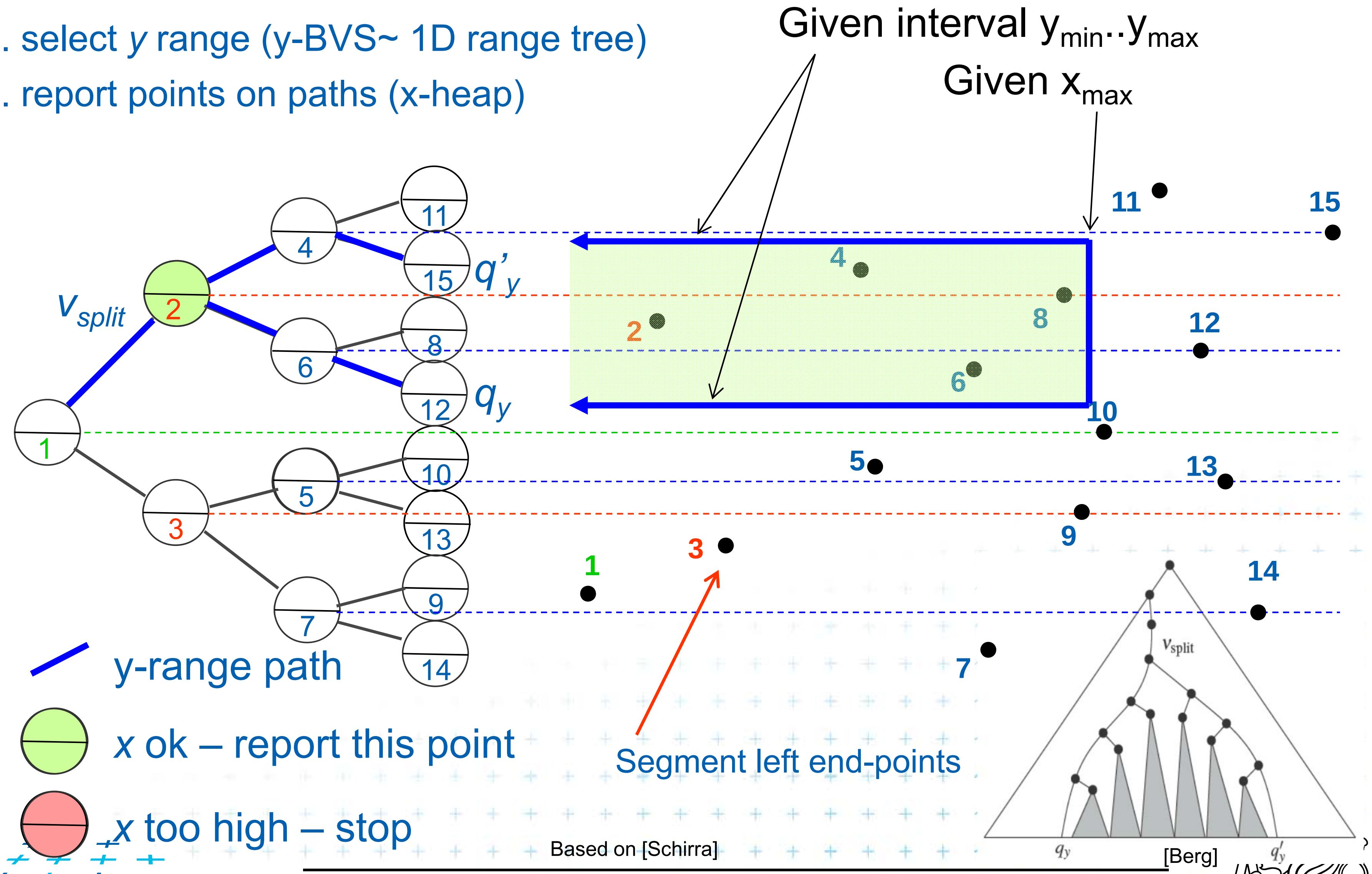
🟥 *x* too high – stop

Based on [Schirra]

[Berg]

Priority search tree query

1. select *y* range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)



Given interval $y_{min}..y_{max}$

Given x_{max}

Segment left end-points

y-range path

🟩 *x* ok – report this point

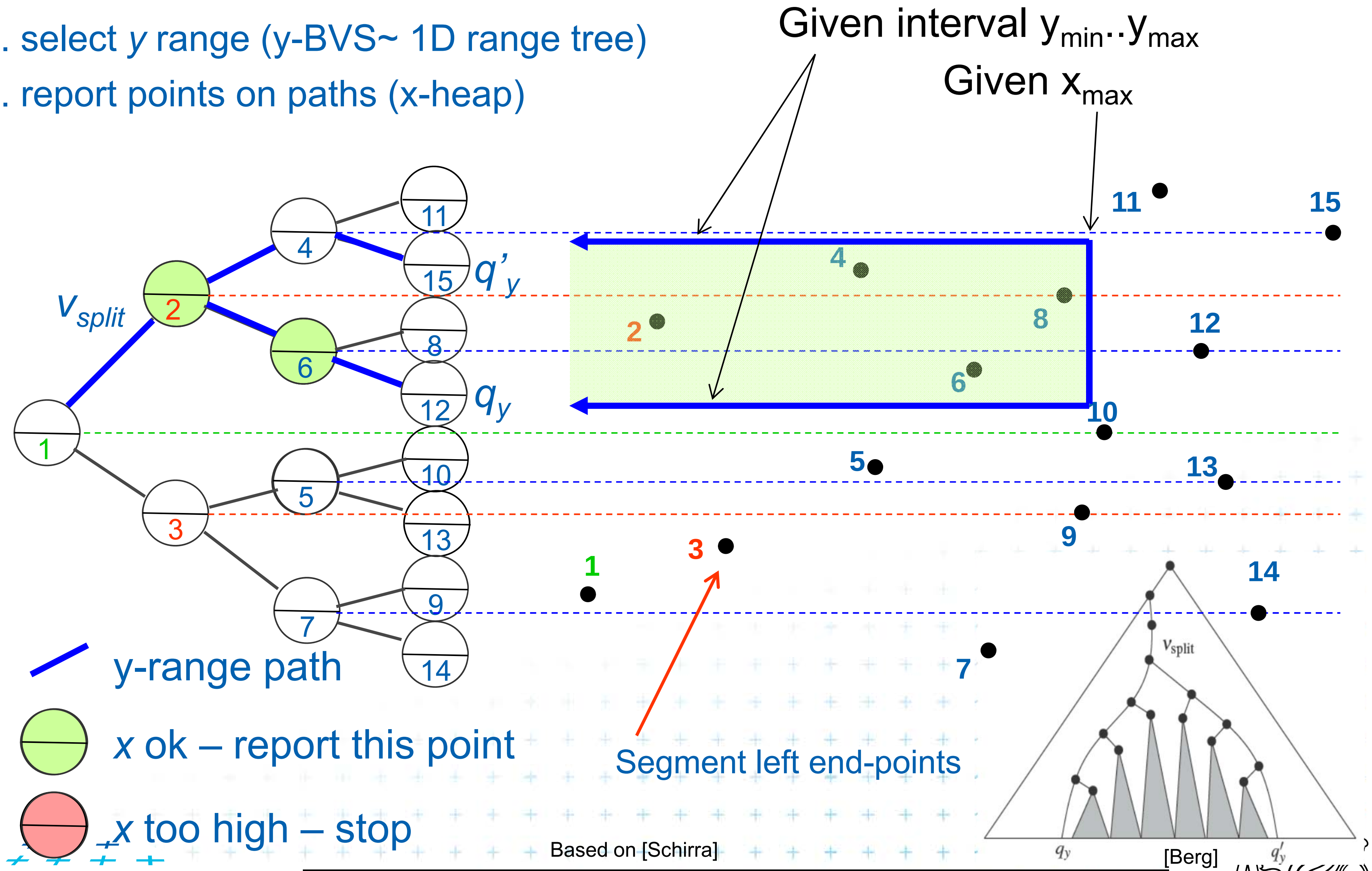
🟥 *x* too high – stop

Based on [Schirra]

[Berg]

Priority search tree query

1. select *y* range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)



y-range path

🟩 *x* ok – report this point

🟥 *x* too high – stop

Based on [Schirra]

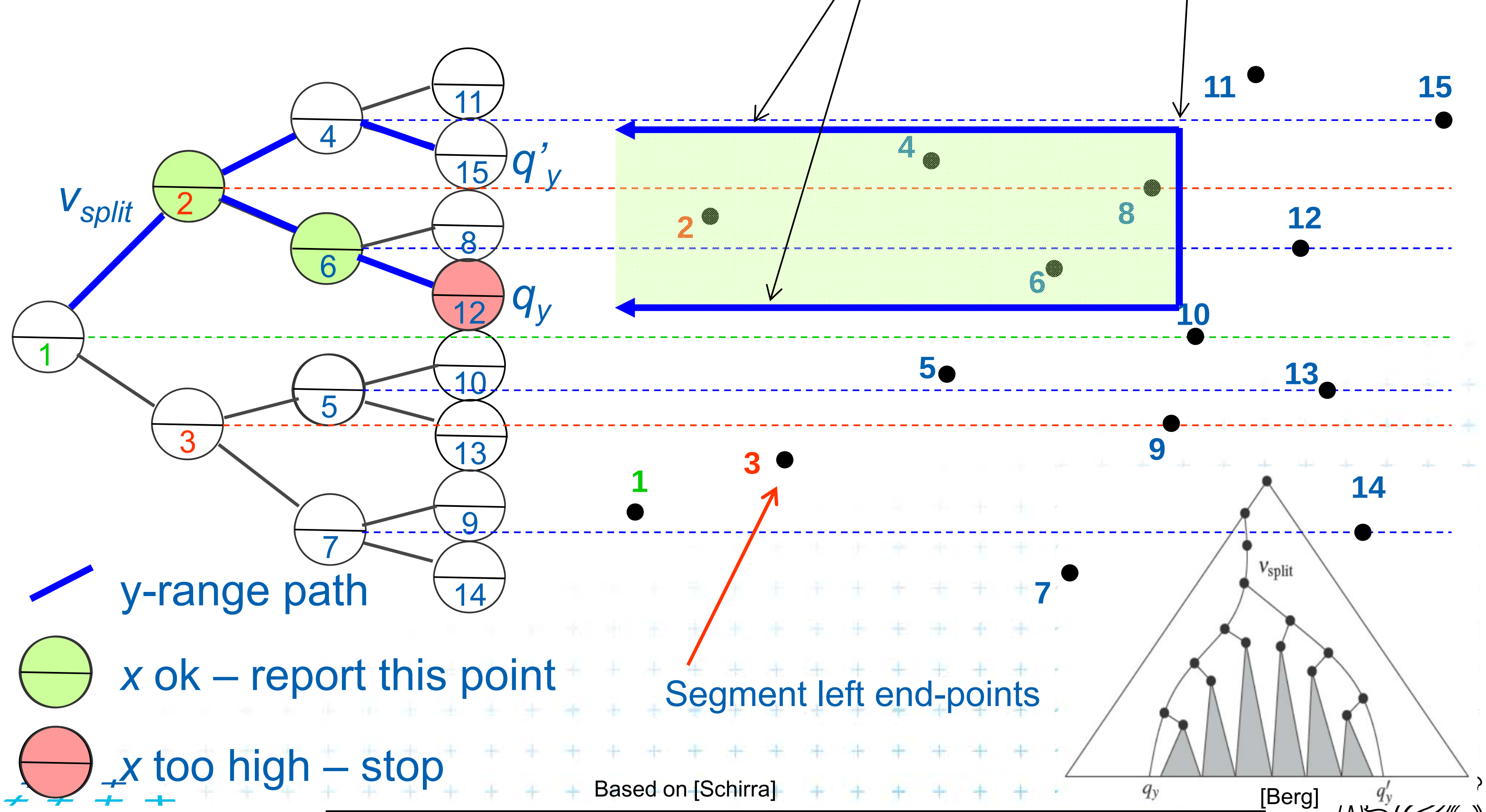
[Berg]

Priority search tree query

1. select *y* range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)

Given interval $y_{min}..y_{max}$

Given x_{max}



Segment left end-points

y-range path

🟩 *x* ok – report this point

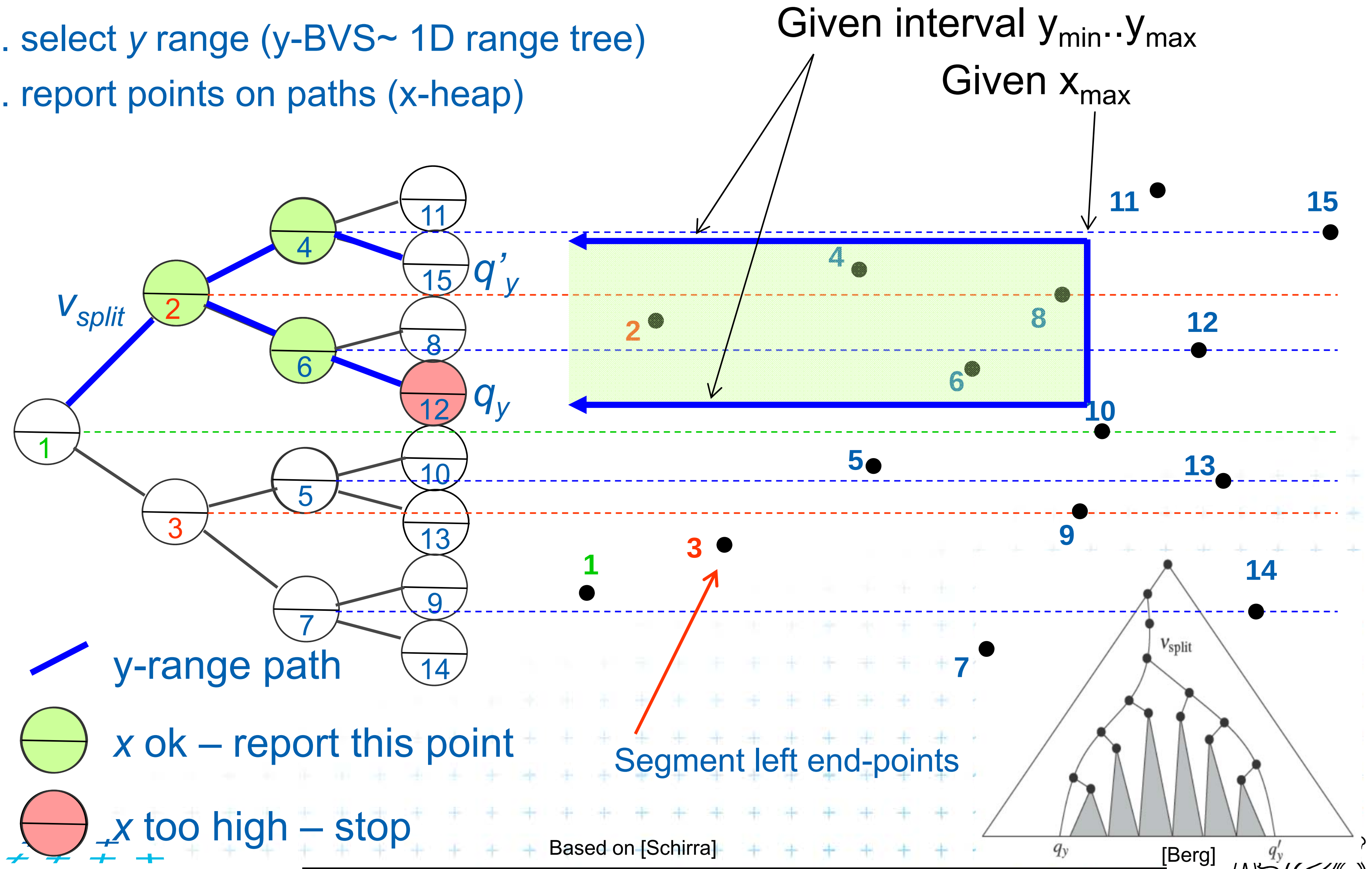
🟥 *x* too high – stop

Based on [Schirra]

[Berg]

Priority search tree query

1. select *y* range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)



Given interval $y_{min}..y_{max}$

Given x_{max}

Segment left end-points

y-range path

x ok – report this point

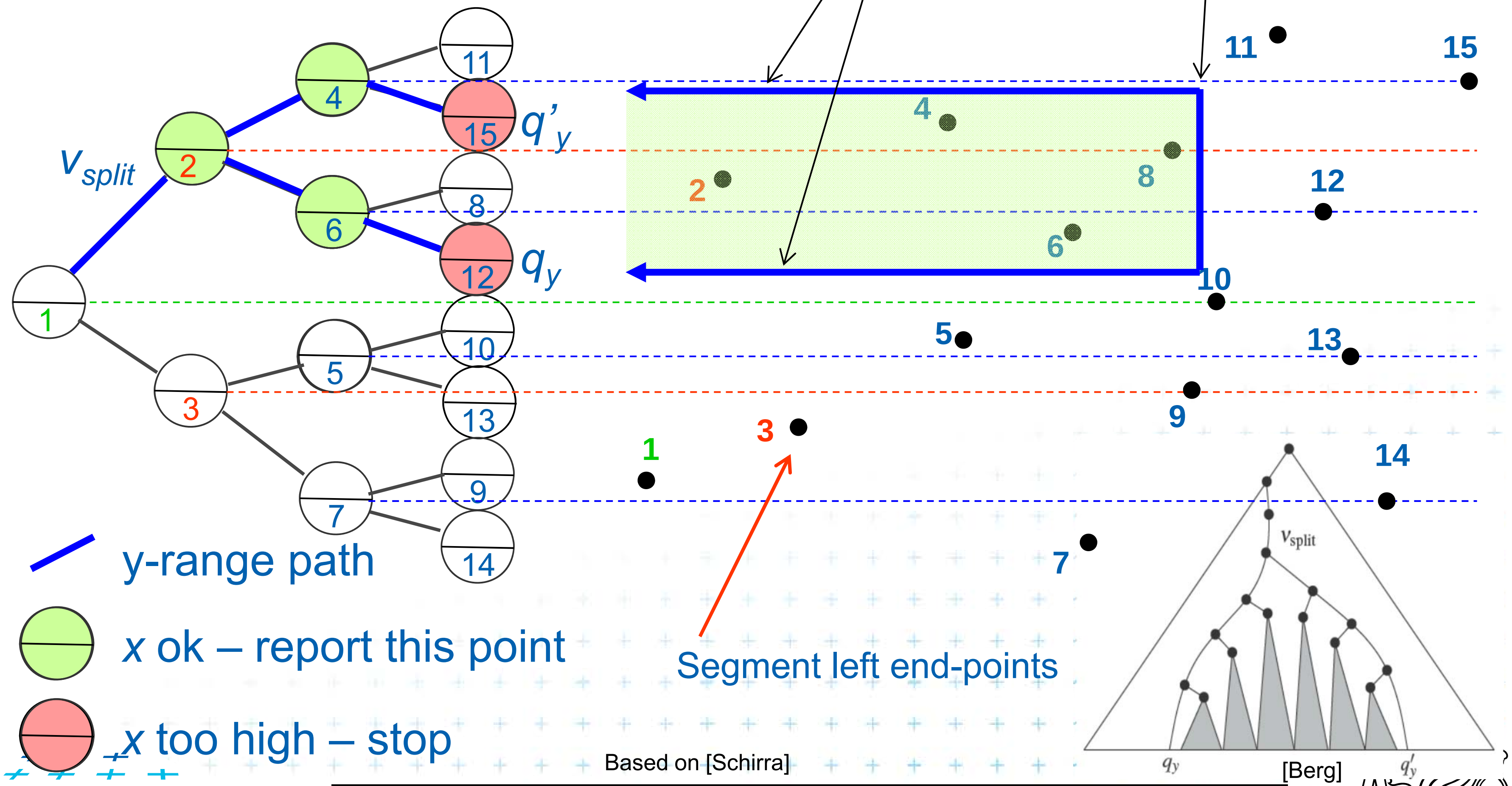
x too high – stop

Based on [Schirra]

[Berg]

Priority search tree query

1. select *y* range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)



Given interval $y_{min}..y_{max}$

Given x_{max}

Segment left end-points

y-range path

x ok – report this point

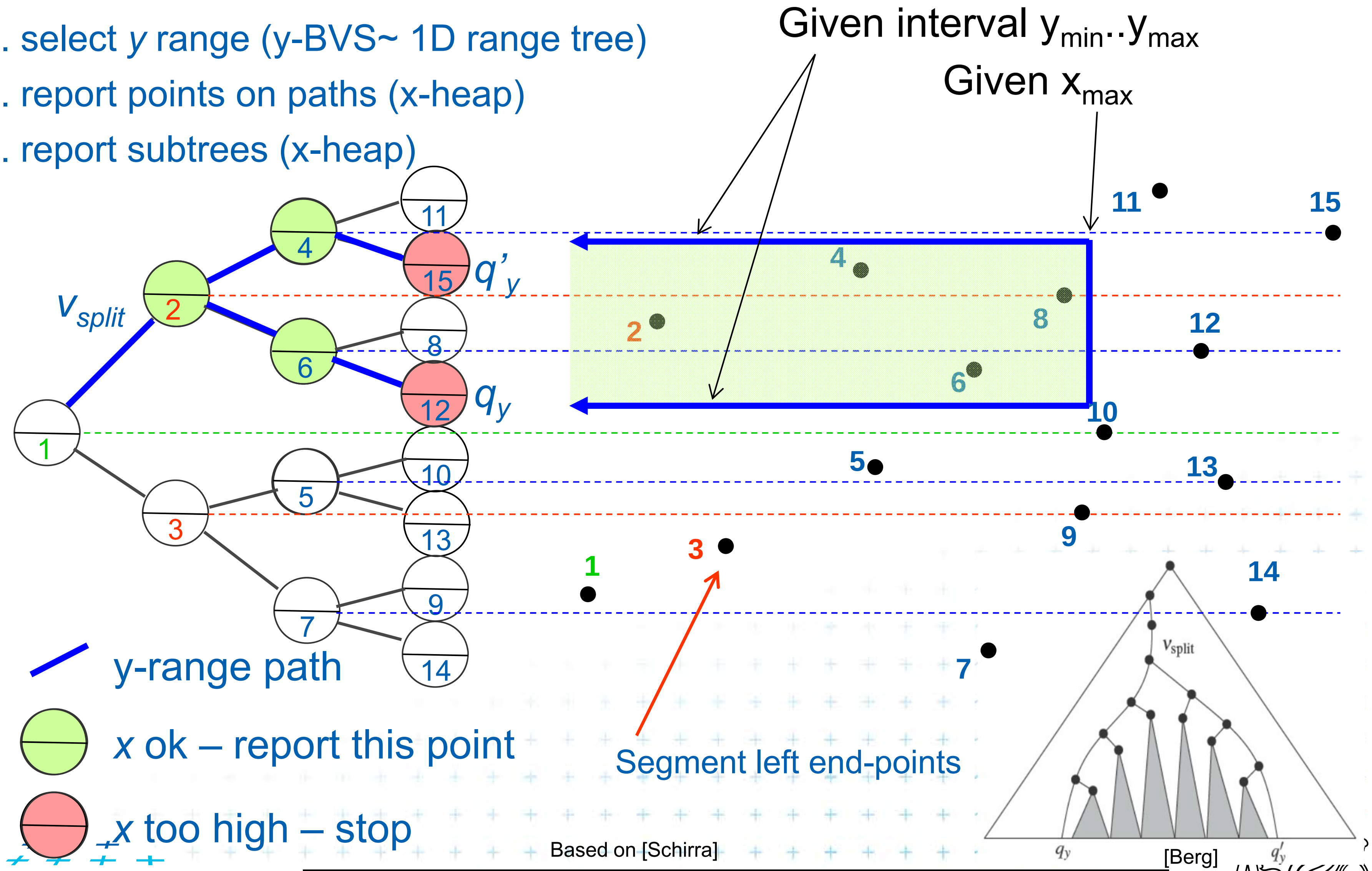
x too high – stop

Based on [Schirra]

[Berg]

Priority search tree query

1. select *y* range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)
3. report subtrees (x-heap)



y-range path

🟩 *x* ok – report this point

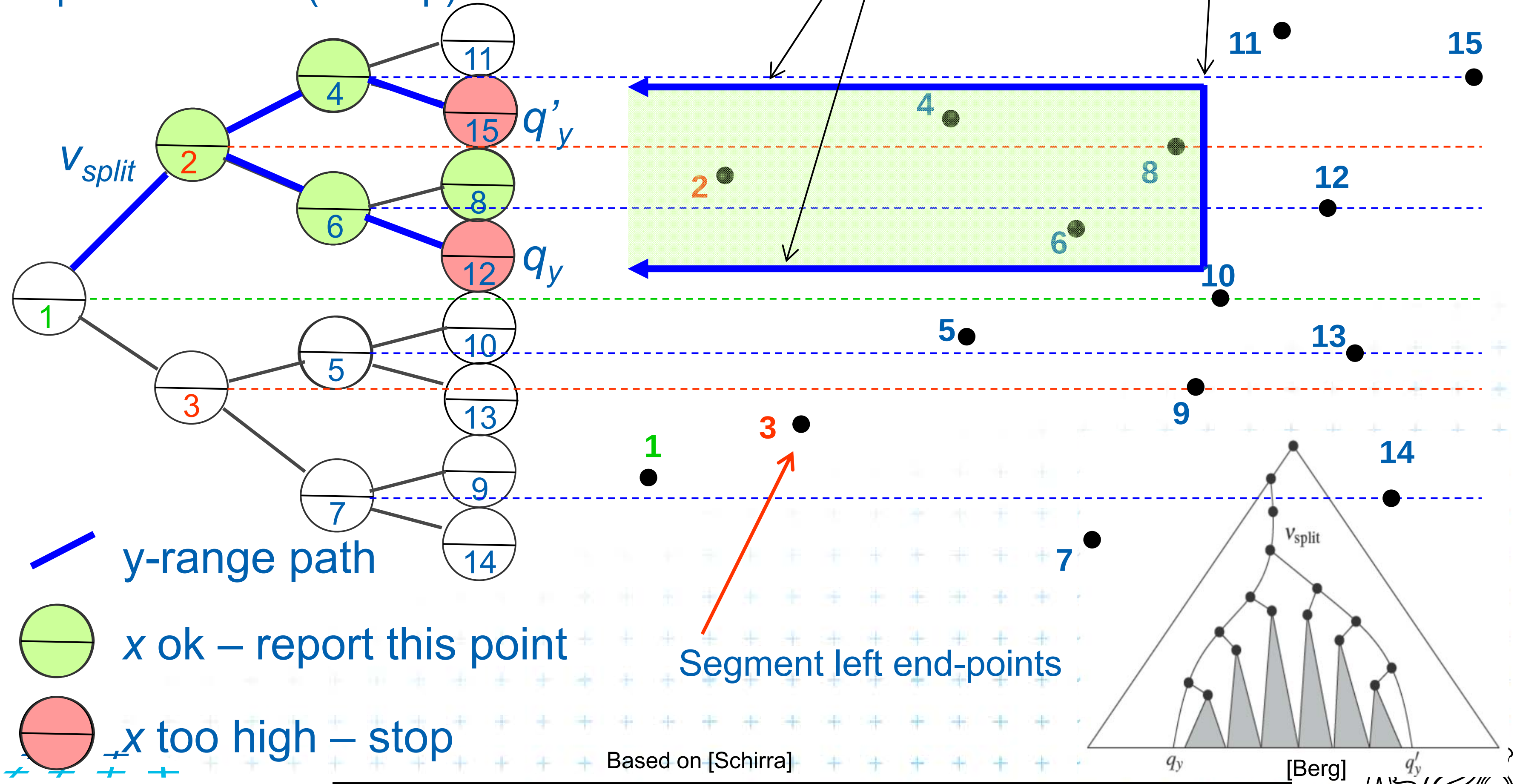
🟥 *x* too high – stop

Based on [Schirra]

[Berg]

Priority search tree query

1. select *y* range (y-BVS~ 1D range tree)
2. report points on paths (x-heap)
3. report subtrees (x-heap)



Given interval $y_{min}..y_{max}$

Given x_{max}

Segment left end-points

y-range path

🟩 *x* ok – report this point

🟥 *x* too high – stop

Based on [Schirra]

[Berg]

Priority search tree complexity

For set of *n* points in the plane

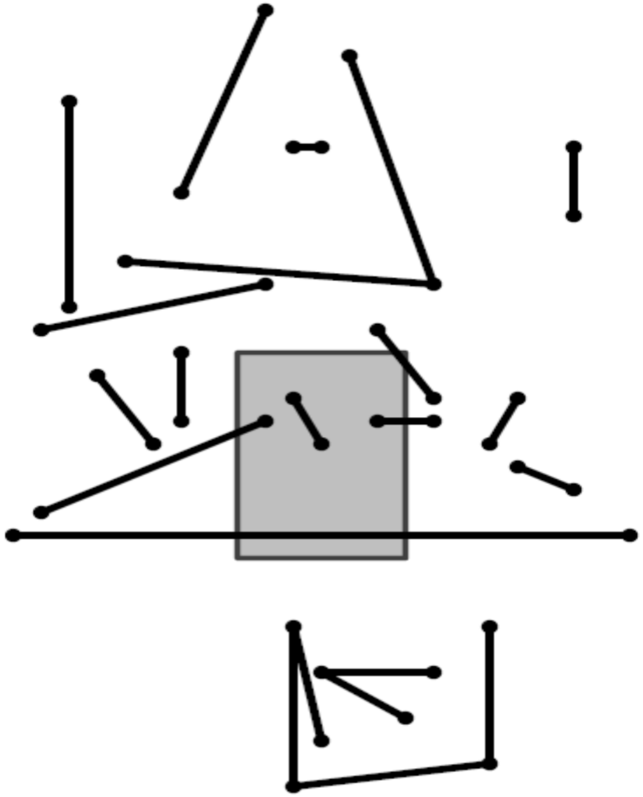
- Build O(*n* log *n*)
- Storage O(*n*)
- Query O(*k* + log *n*)
  - points in query range $(-\infty : q_x] \times [q_y ; q'_y])$
  - *k* is number of reported points

- Use Priority search tree as associated data structure for interval trees for storage of M (one for M_L, one for M_R)

Talk overview

1. Windowing of axis parallel line segments in 2D (variants of *interval tree - IT*)
   1. Line stabbing (standard *IT* with *sorted lists*)
   2. Line segment stabbing (*IT* with *range trees*)
   3. Line segment stabbing (*IT* with *priority search trees*)
2. Windowing of line segments in general position
   - *segment tree*

2. Windowing of line segments in general position



[Vakken]

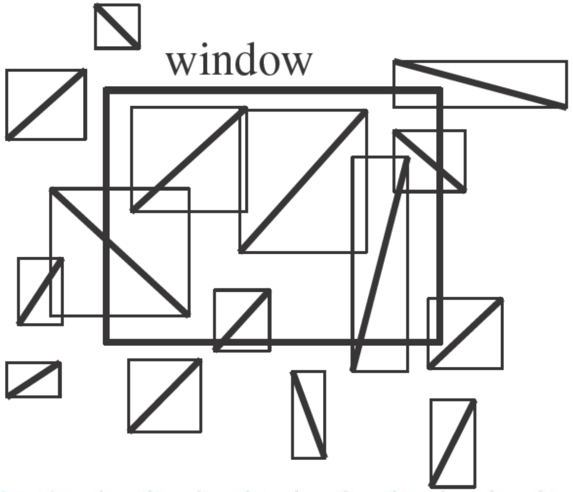
Windowing of arbitrary oriented line segments

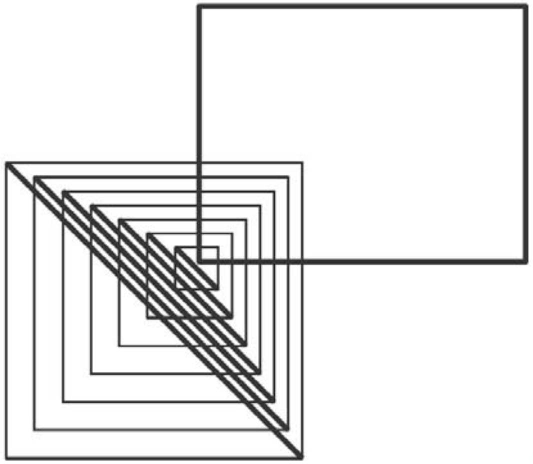
- Two cases of intersection

  a,b) Endpoint inside the query window => range tree

  c) Segment intersects side of query window => ???

- Intersection with BBOX (segment bounding box)?
  - Intersection with 4n sides
  - But segments may not intersect the window –> query y



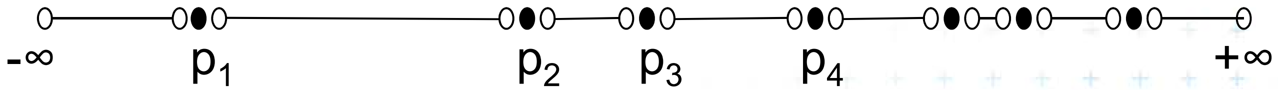


Talk overview

1. Windowing of axis parallel line segments in 2D (variants of *interval tree - IT*)
   1. Line stabbing (*IT* with *sorted lists*)
   2. Line segment stabbing (*IT* with *range trees*)
   3. Line segment stabbing (*IT* with *priority search trees*)
2. Windowing of line segments in general position
   - *segment tree*

Segment tree [Bentley, 1977]

- Exploits locus approach
  - Partition parameter space into regions of same answer
  - Localization of such region = knowing the answer
- For given set *S* of *n* intervals (segments) on real line
  - Finds *m* elementary intervals (induced by interval end-points)
  - Partitions 1D parameter space into these elementary intervals



$$(-\infty : p_1), [p_1 : p_1], (p_1 : p_2), [p_2 : p_2], \ldots,$$
$$(p_{m-1} : p_m), [p_m : p_m], (p_m : +\infty)$$

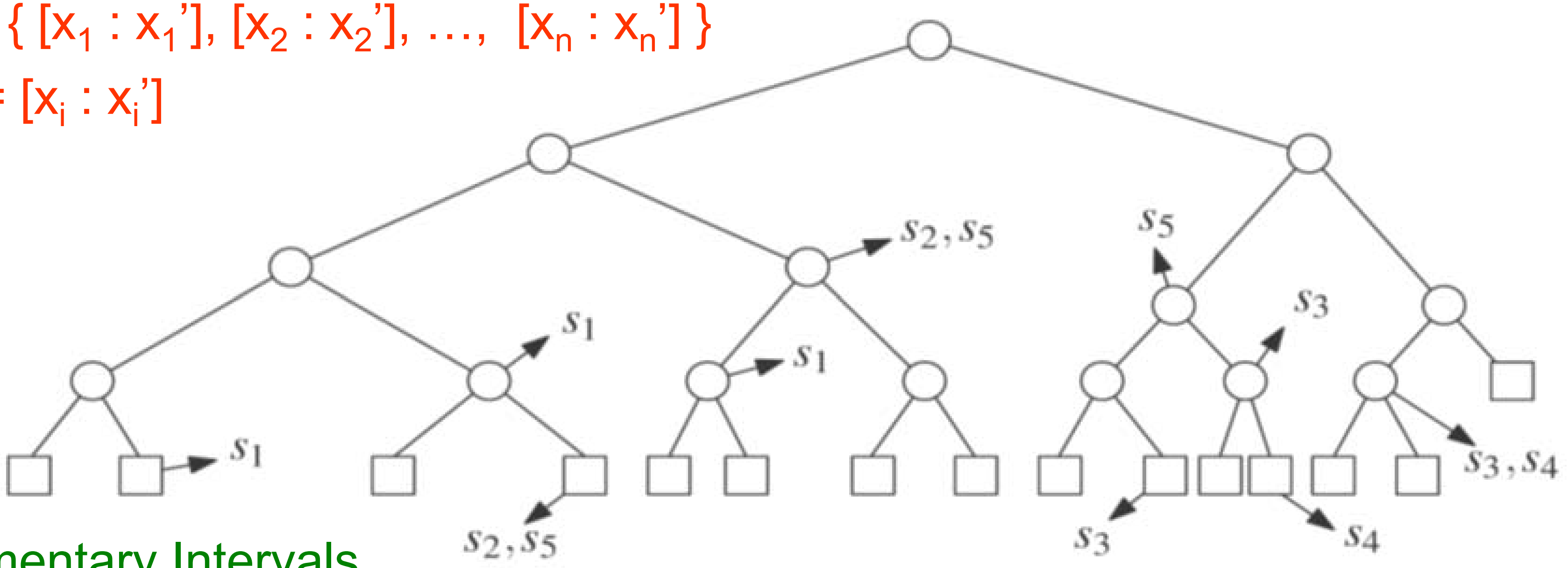
  - Stores intervals s_i with the elementary intervals
  - Reports the intervals s_i containing query point q_x.

Segment tree example

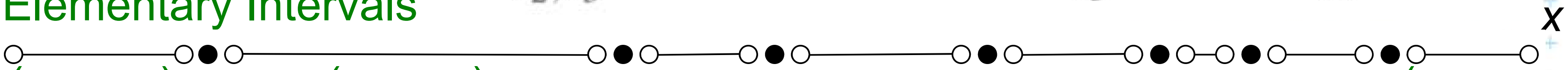
Intervals

S = { [x_1 : x_1'], [x_2 : x_2'], …, [x_n : x_n'] }

s_i = [x_i : x_i']



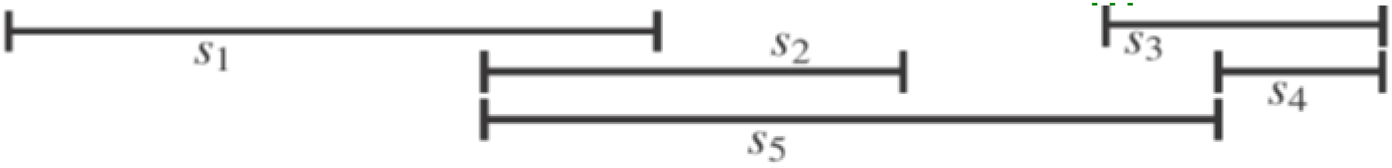
Elementary Intervals



(-∞ : p_1) (p$_1$: p_2) … (p_m : +∞)

[p_1: p_1] [p_2: p_2] [p_2: p_3]

Intervals



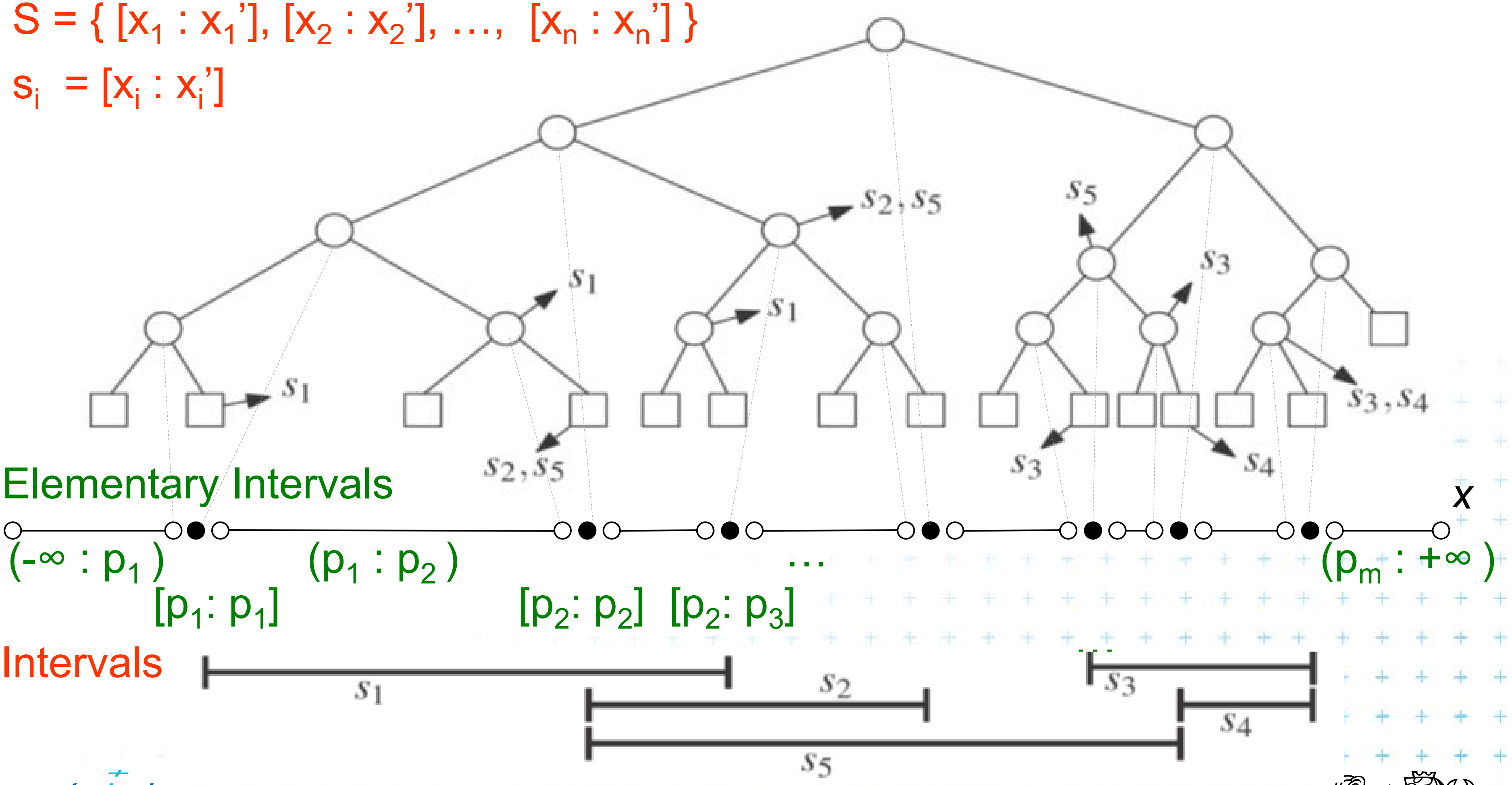
[Berg]

Segment tree example

Intervals

$S = \{ [x_1 : x_1'], [x_2 : x_2'], \ldots, \; [x_n : x_n'] \}$

$s_i \; = [x_i : x_i']$



Elementary Intervals

$(-\infty : p_1)$ $[p_1 : p_1]$ $(p_1 : p_2)$ $[p_2 : p_2]$ $[p_2 : p_3]$ … $(p_m : +\infty)$

Intervals

[Berg]

Segment tree definition

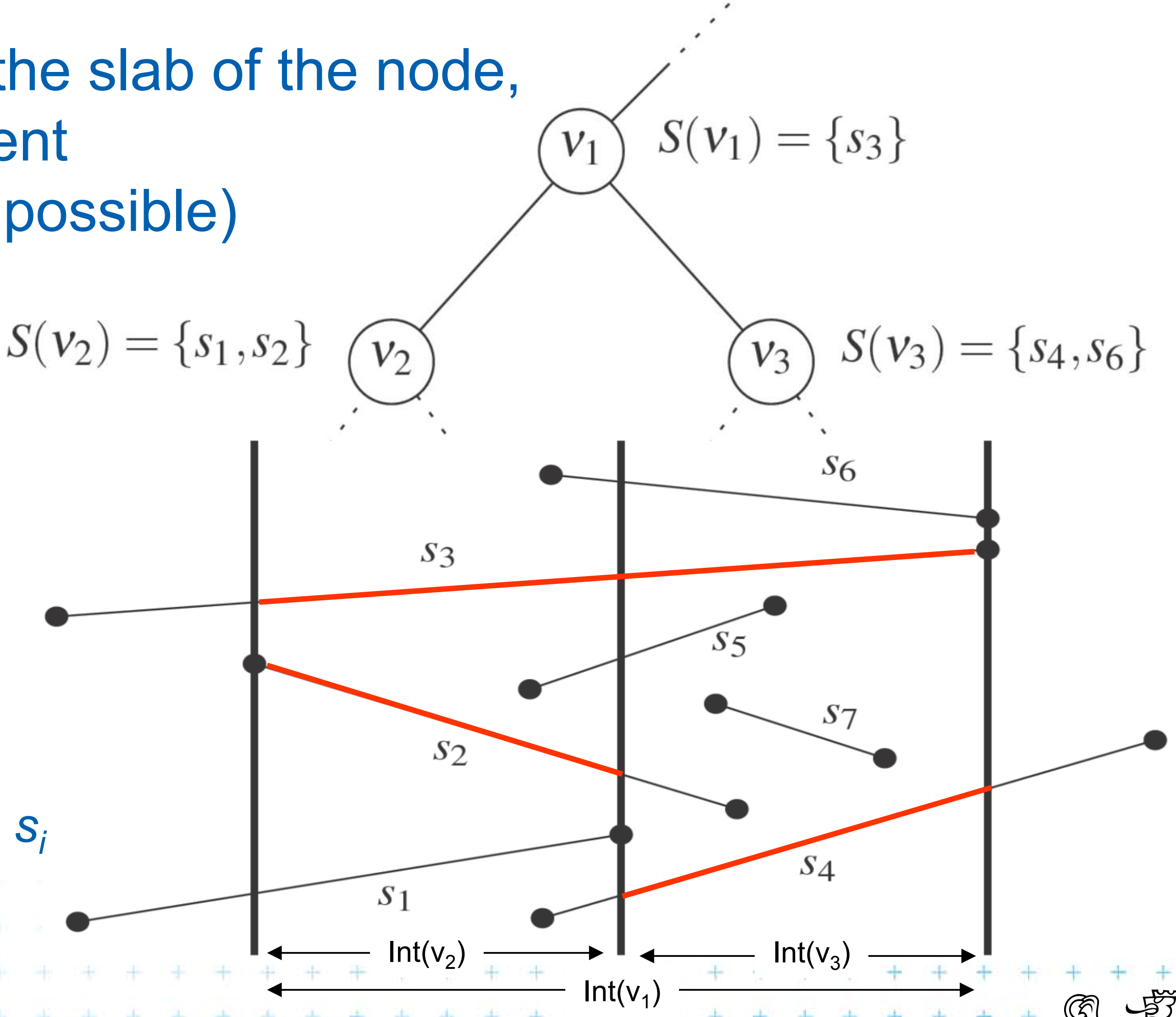
Segment tree

- Skeleton is a balanced binary tree T
- Leaves ~ elementary intervals Int(v)
- Internal nodes v
  ~ union of elementary intervals of its children
  - Store: 1. interval Int(v) = union of elementary intervals of its children
    2. canonical set $S(v)$ of intervals $[x : x'] \in S$ segments s_i
  - Holds $\text{Int}(v) \subseteq [x : x']$ and $\text{Int}(\text{parent}(v)) \not\subseteq [x : x']$
    (node interval is not larger than the segment)
  - Intervals $[x : x']$ are stored as high as possible, such that $\text{Int}(v)$ is completely contained in the segment

DCGI

Segments span the slab

Segments span the slab of the node,
but not of its parent
(stored as up as possible)



$\text{Int}(v_j) \subseteq s_i$

and

$\text{Int}(\text{parent}(v_j)] \not\subseteq s_i$

[Berg]

Query segment tree – stabbing query

QuerySegmentTree(v, q_x)
Input: The root of a (subtree of a) segment tree and a query point q_x
Output: All intervals in the tree containing q_x.

```
Report all the intervals $s_i$ in $S(v)$.        // current node
if $v$ is not a leaf
    if $q_x \in$ Int( $lc(v)$ )                   // go left
        QuerySegmentTree( $lc(v), q_x$ )
    else                                          // or go right
        QuerySegmentTree( $rc(v), q_x$ )
```

Query time O(log n + k), where k is the number of reported intervals

O(1 + k_v) for one node

Height O(log n)

Segment tree construction

```
ConstructSegmentTree( $S$ )
Input:    Set of intervals $S$ - segments
Output:   segment tree

1.  Sort endpoints of segments in $S$ -> get elemetary intervals …O($n$ log $n$)
2.  Construct a binary search tree $T$ on elementary intervals    …O($n$)
     (bottom up) and determine the interval Int($v$) it represents
3.  Compute the canonical subsets for the nodes (lists of their segments):
4.      v = root( $T$ )
5.      for all segments $s_i = [x : x'] \in S$
6.          InsertSegmentTree( $v$, $[x : x']$ )
```

Segment tree construction – interval insertion

InsertSegmentTree(v, $[x : x']$)

Input: The root of (a subtree of) a segment tree and an interval.

Output: The interval will be stored in the subtree.

```
if Int($v$) $\subseteq [\, x : x' \,]$                    // Int($v$) contains $s_i = [\, x : x' \,]$
    store $[\, x : x' \,]$ at $v$
else if Int( $lc(v)$ ) $\cap [\, x : x' \,] \neq \emptyset$
        InsertSegmentTree( $lc(v)$, $[x : x' \,]$ )
    if Int( $rc(v)$ ) $\cap [\, x : x' \,] \neq \emptyset$
        InsertSegmentTree( $rc(v)$, $[x : x' \,]$ )
```

One interval is stored at most twice in one level =>

Single interval insert $O(\log n)$, insert n intervals $O(\cancel{2}n \log n)$

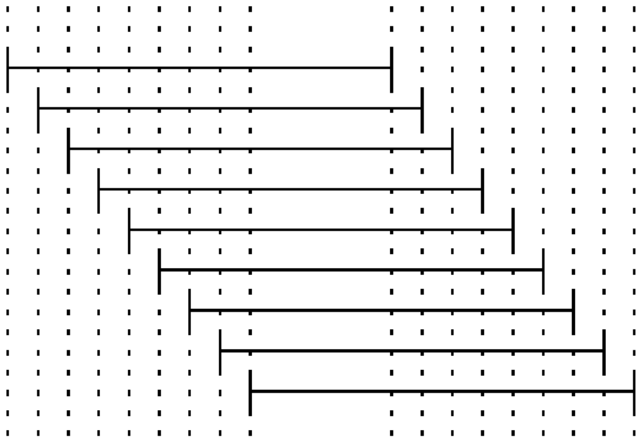
Construction total $O(n \log n)$

Storage $O(n \log n)$

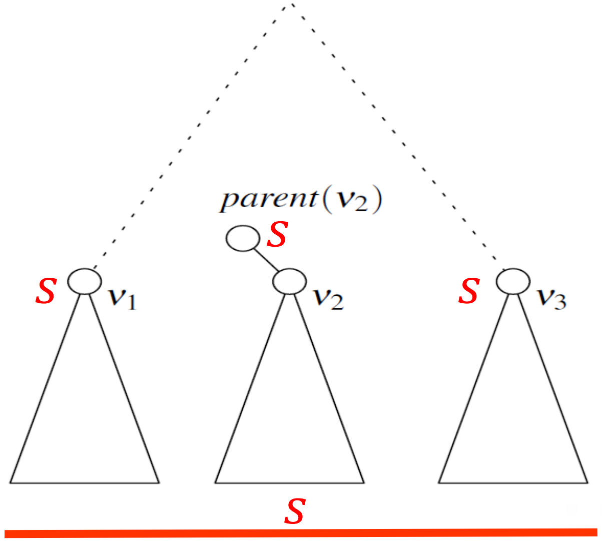
Tree height $O(\log n)$, name stored max 2x in one level

Storage total $O(n \log n)$ – see next slide

Space complexity - notes



[Berg]



[Berg]

Worst case – $O(n^2)$ segments in leaf

But

Store segments as high, as possible

Segment max 2 times in one level ⇐

$\max 4n + 1$ elementary intervals (leaves)
⇒ $O(n)$ space for the tree

⇒ $O(n \log n)$ space for interval names

s covered by v_1 and v_3

⇒ v_2 covered, $Int(v_2) \in s$

As v_2 lies between v_1 and v_3

⇒ $Int(parent(v_2)) \in s$ ⇒ segment s will not be stored in v_2

Segment tree complexity

A segment tree for set *S* of *n* intervals in the plane,

- Build O(*n* log *n*)
- Storage O(*n* log *n*)
- Query O(*k* + log *n*)
  - Report all intervals that contain a query point
  - *k* is number of reported intervals

Segment tree versus Interval tree

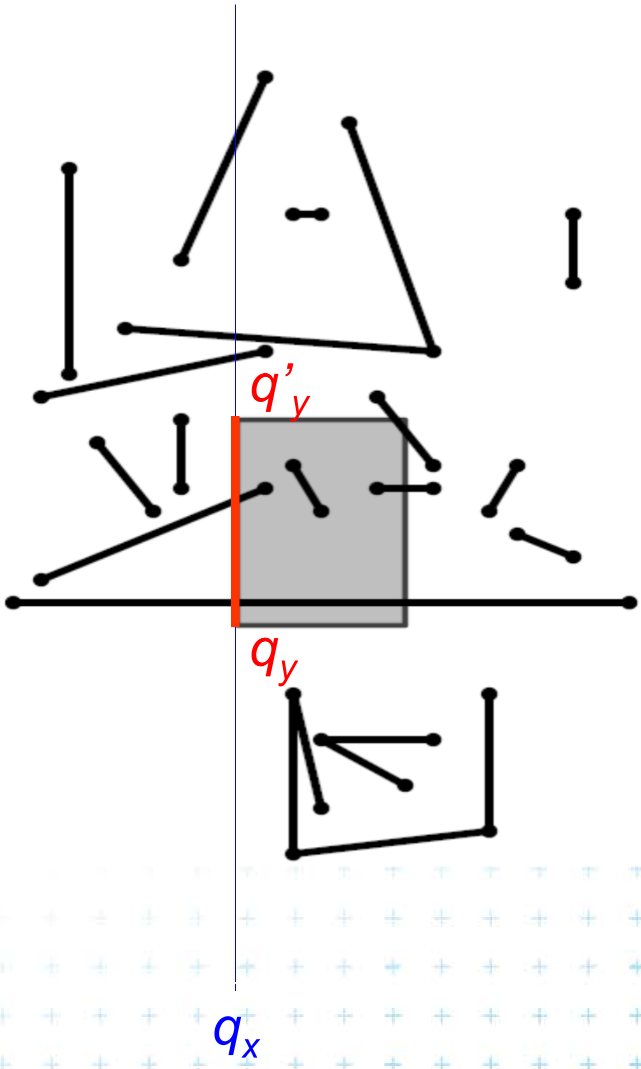
- Segment tree
  - O(*n* log *n*) storage x O(*n*) of Interval tree
  - But returns exactly the intersected segments s_i, interval tree must search the lists ML and/or MR
- Good for
  1. extensions (allows different structuring of intervals)
  2. stabbing counting queries
     - store number of intersected intervals in nodes
     - O(n) storage and O(log *n*) query time = optimal
  3. higher dimensions – multilevel segment trees

     (Interval and priority search trees do not exist in ^dims)

Talk overview

1. Windowing of axis parallel line segments in 2D (variants of *interval tree - IT*)
   1. Line stabbing (standard *IT* with *sorted lists*)
   2. Line segment stabbing (*IT* with *range trees*)
   3. Line segment stabbing (*IT* with *priority search trees*)
2. Windowing of line segments in general position
   - *segment tree*
   - the algorithm

2. Windowing of line segments in general position

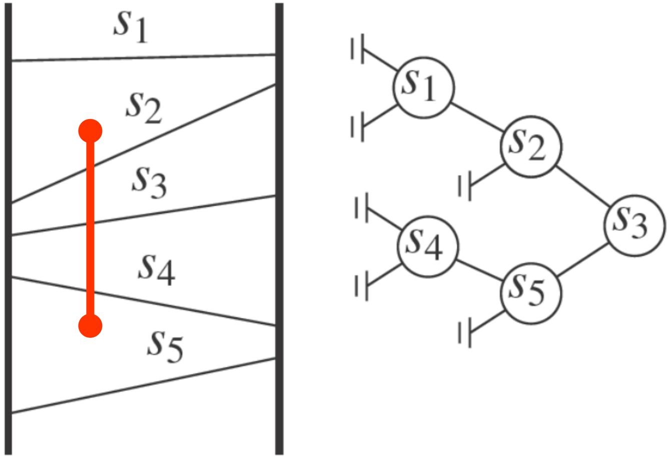


[Vakken]

Windowing of arbitrary oriented line segments

- Let S be a set of arbitrarily oriented line segments in the plane.
- Report the segments intersecting a vertical query segment $q := q_x \times [q_y : q'_y]$
- Segment tree T on x intervals of segments in S
  - node v of T corresponds to vertical slab Int(v) × (-∞ : ∞)
  - segments span the slab of the node, but not of its parent
  - segments do not intersect

  => segments in the slab (node) can be vertically ordered – BST



[Berg]

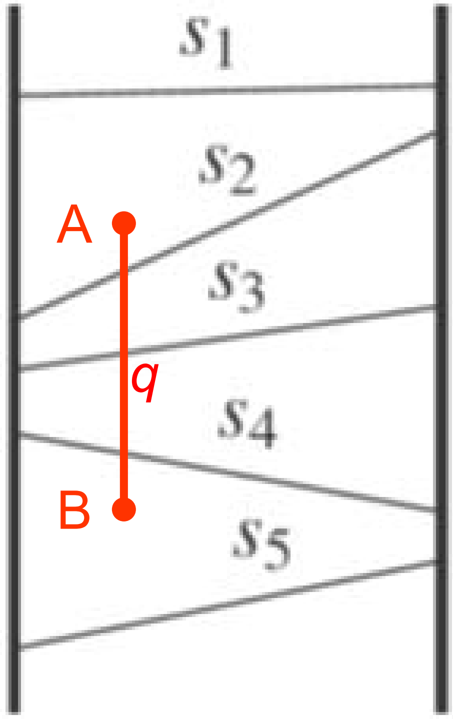
Segments between vertical segment endpoints

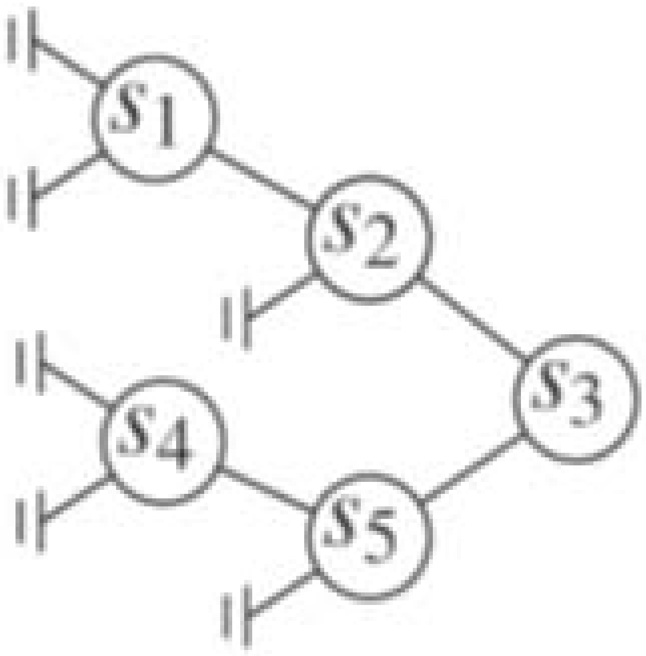
- Segments (in the slab) do not mutually intersect
  => segments can be vertically ordered and stored in BST

  - Each node *v* of the x segment tree has an associated y BST
  - BST *T*(*v*) of node *v* stores the canonical subset *S*(*v*) according to the vertical order
  - Intersected segments can be found by searching *T*(*v*) in O(k_v + log *n*), k_v is the number of intersected segments

Segments between vertical segment endpoints

- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*

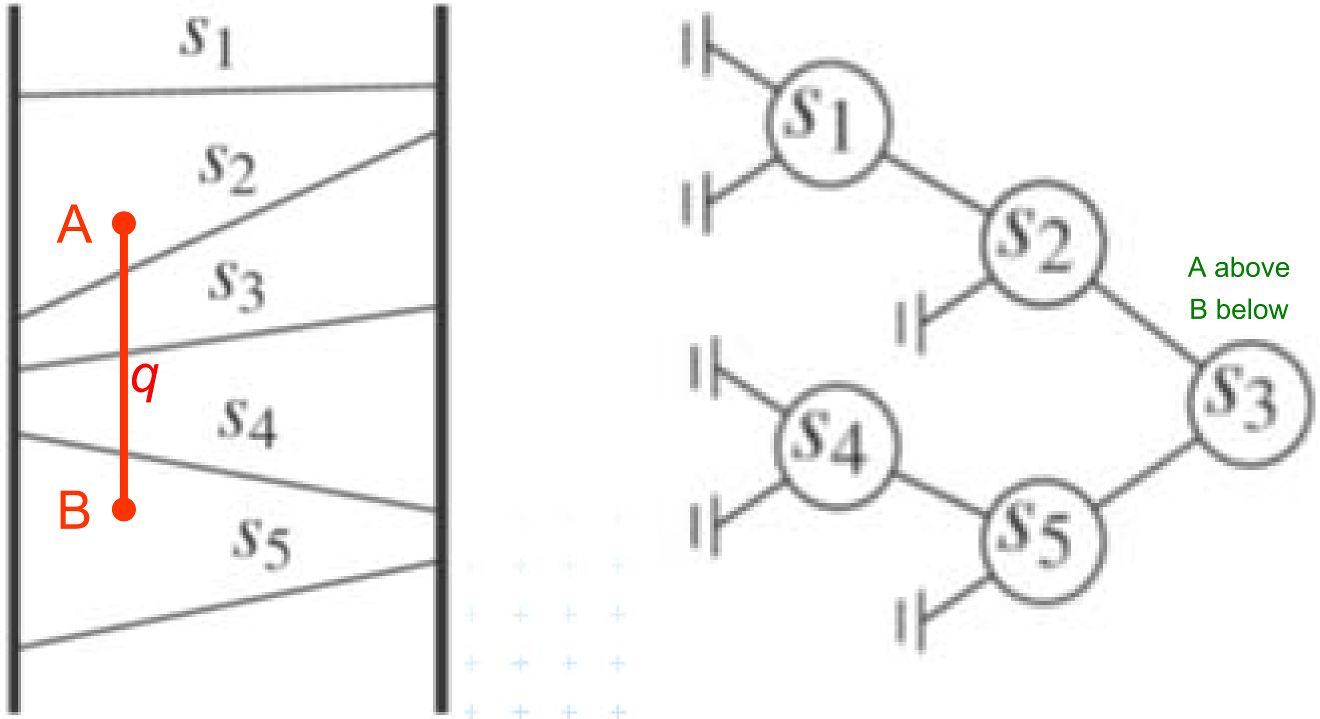




[Berg]

Segments between vertical segment endpoints

- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*



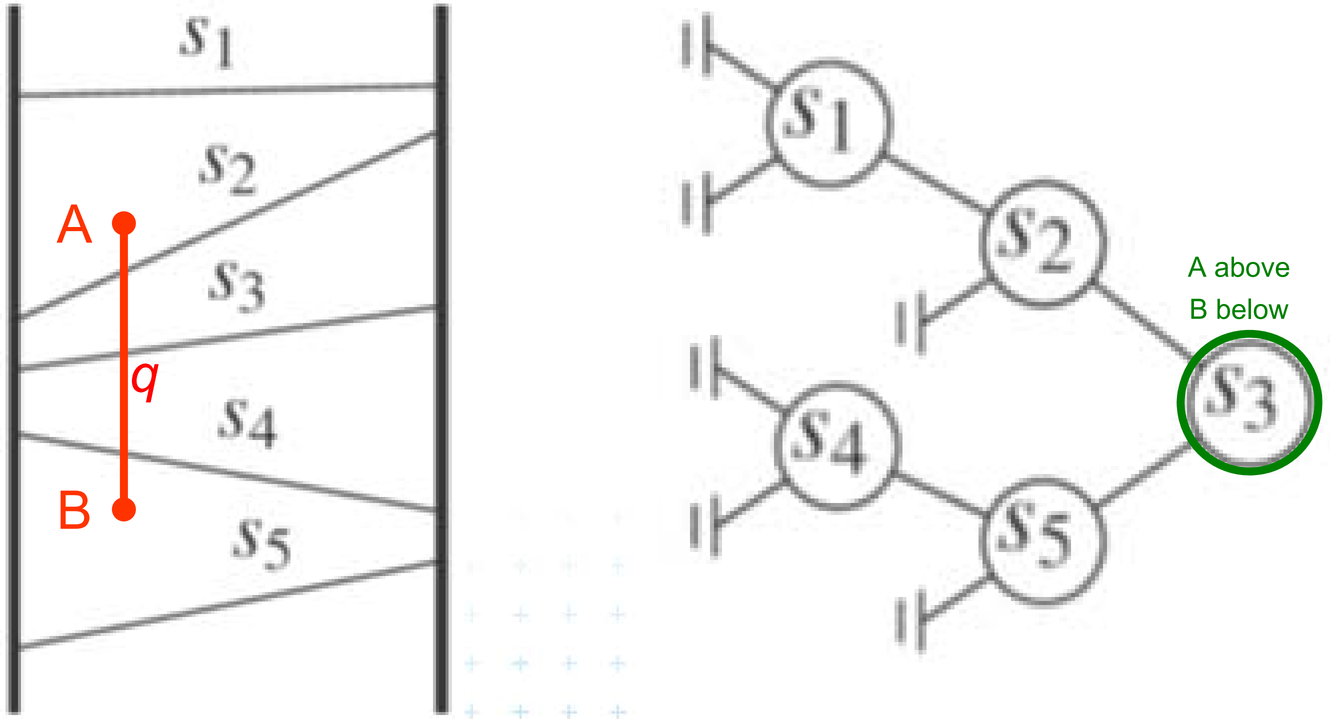
A above

B below

[Berg]

Segments between vertical segment endpoints

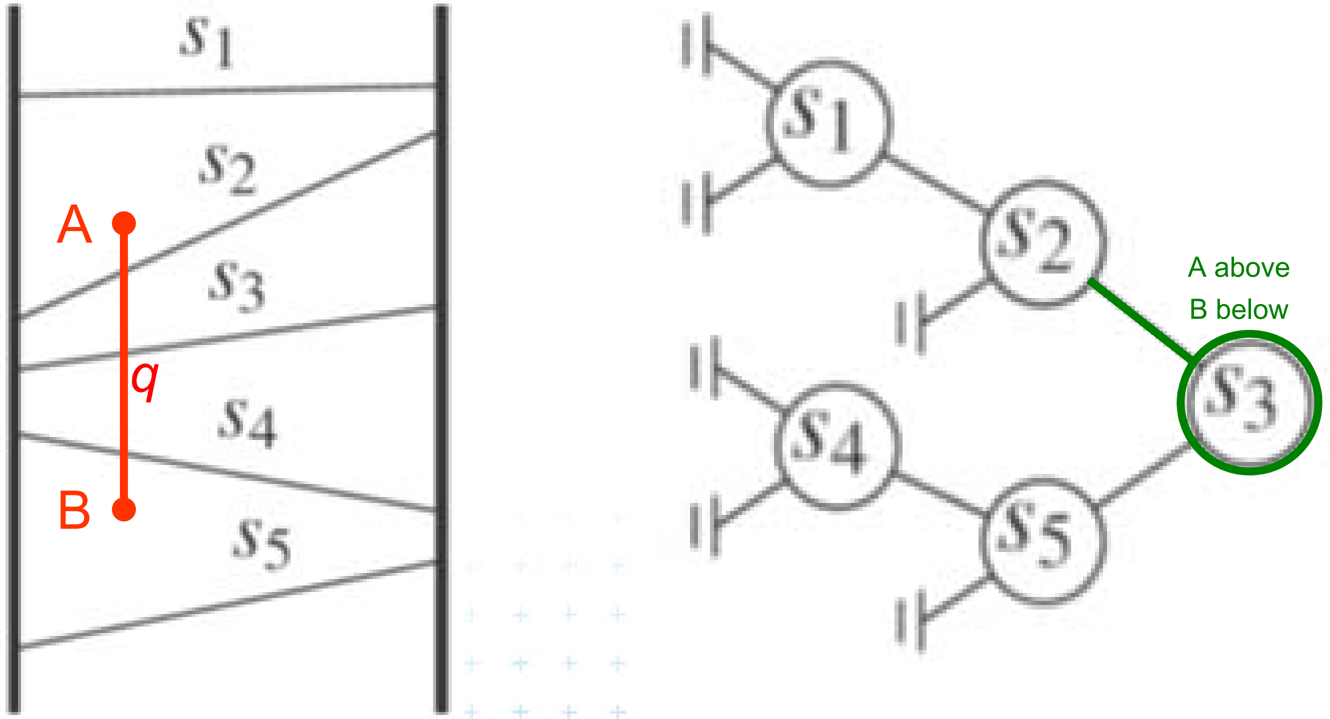
- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*



[Berg]

Segments between vertical segment endpoints

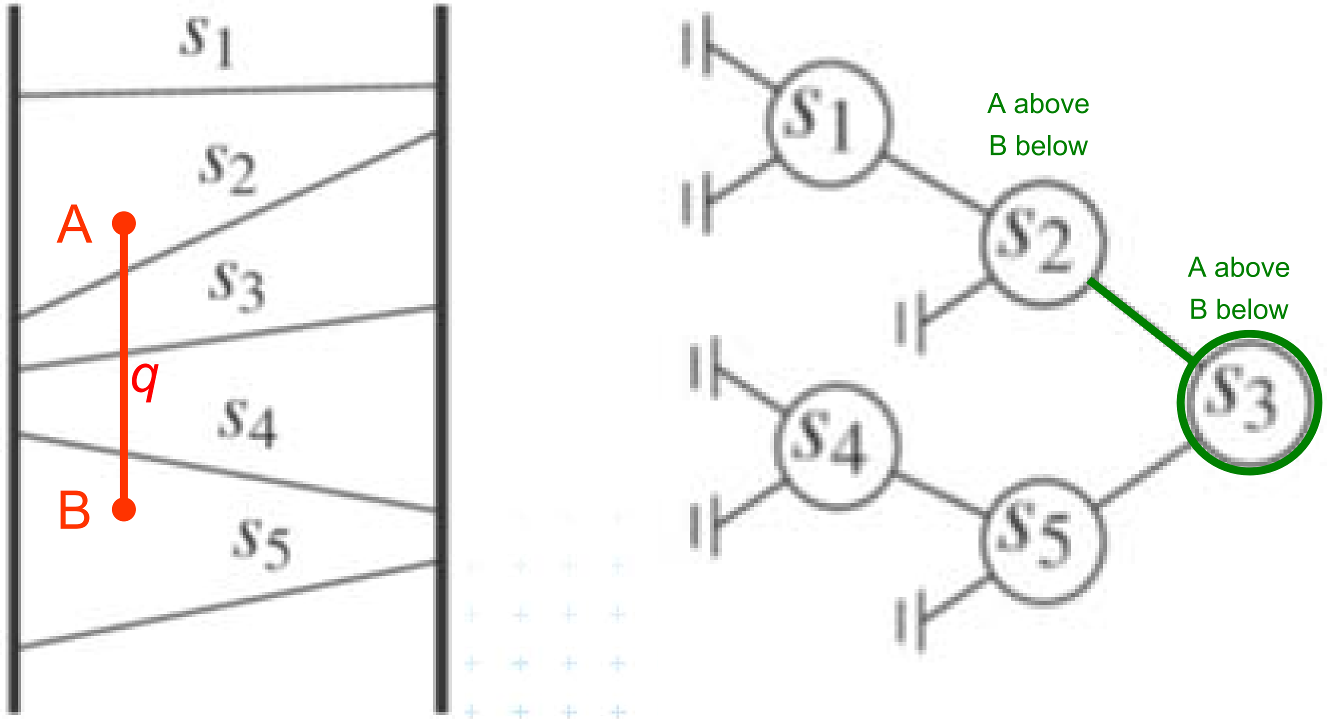
- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*



[Berg]

Segments between vertical segment endpoints

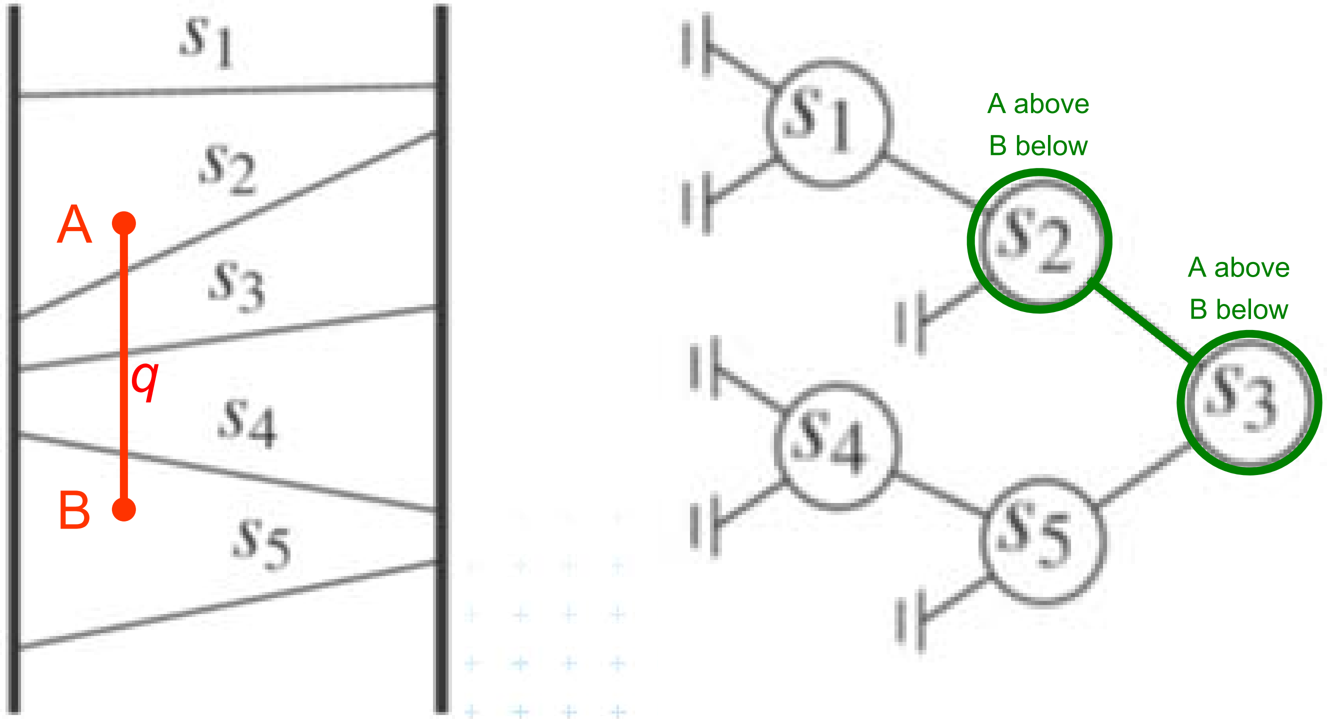
- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*



[Berg]

Segments between vertical segment endpoints

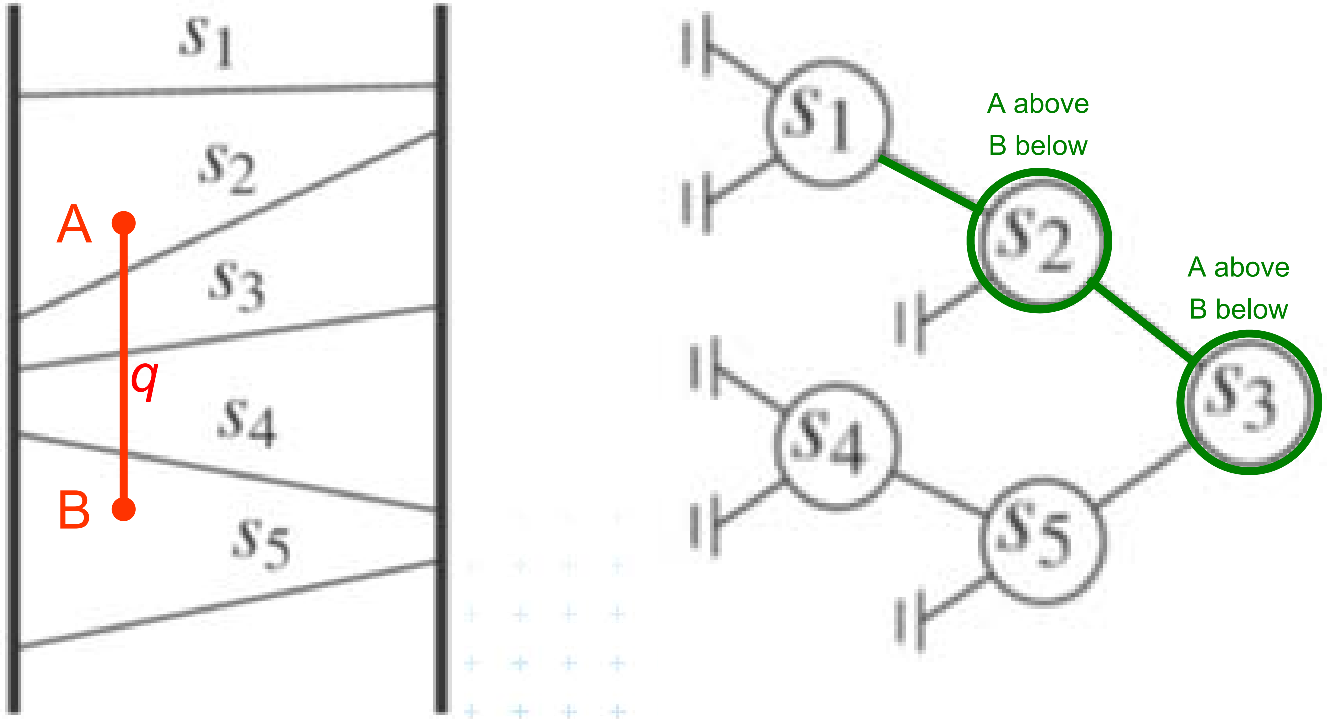
- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*



[Berg]

Segments between vertical segment endpoints

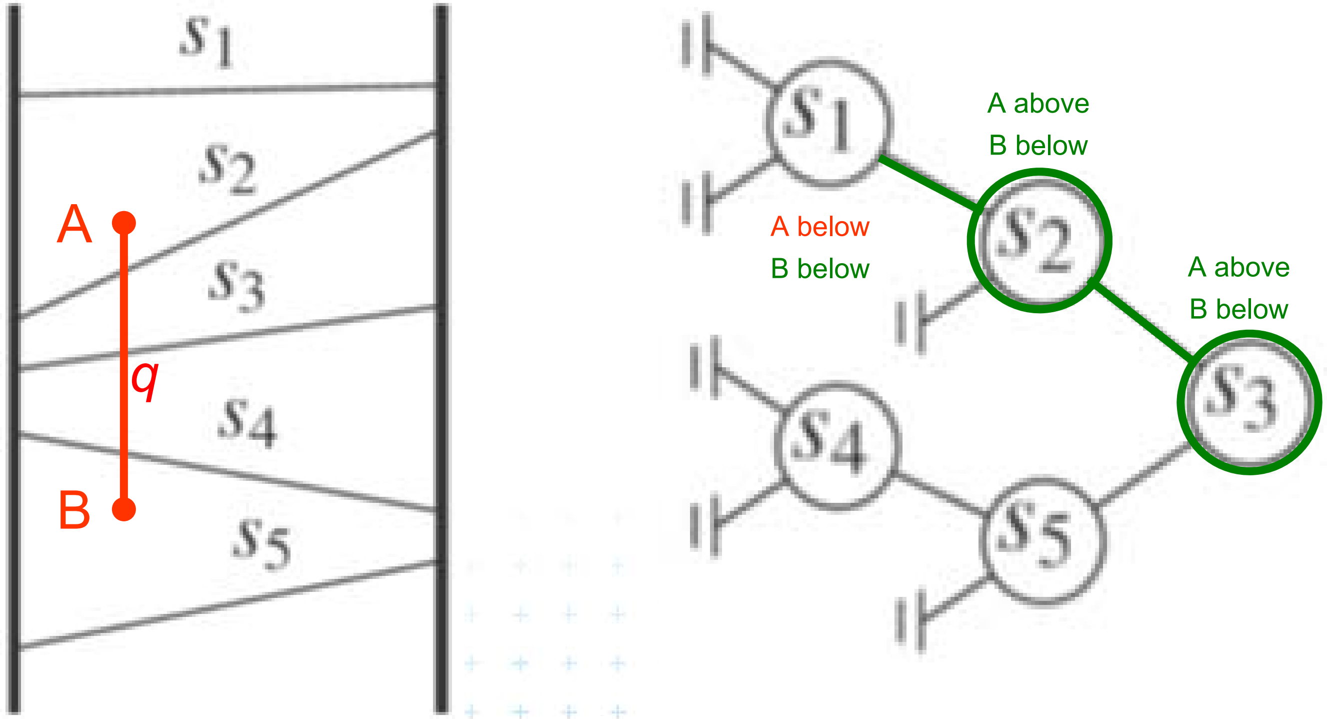
- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*



[Berg]

Segments between vertical segment endpoints

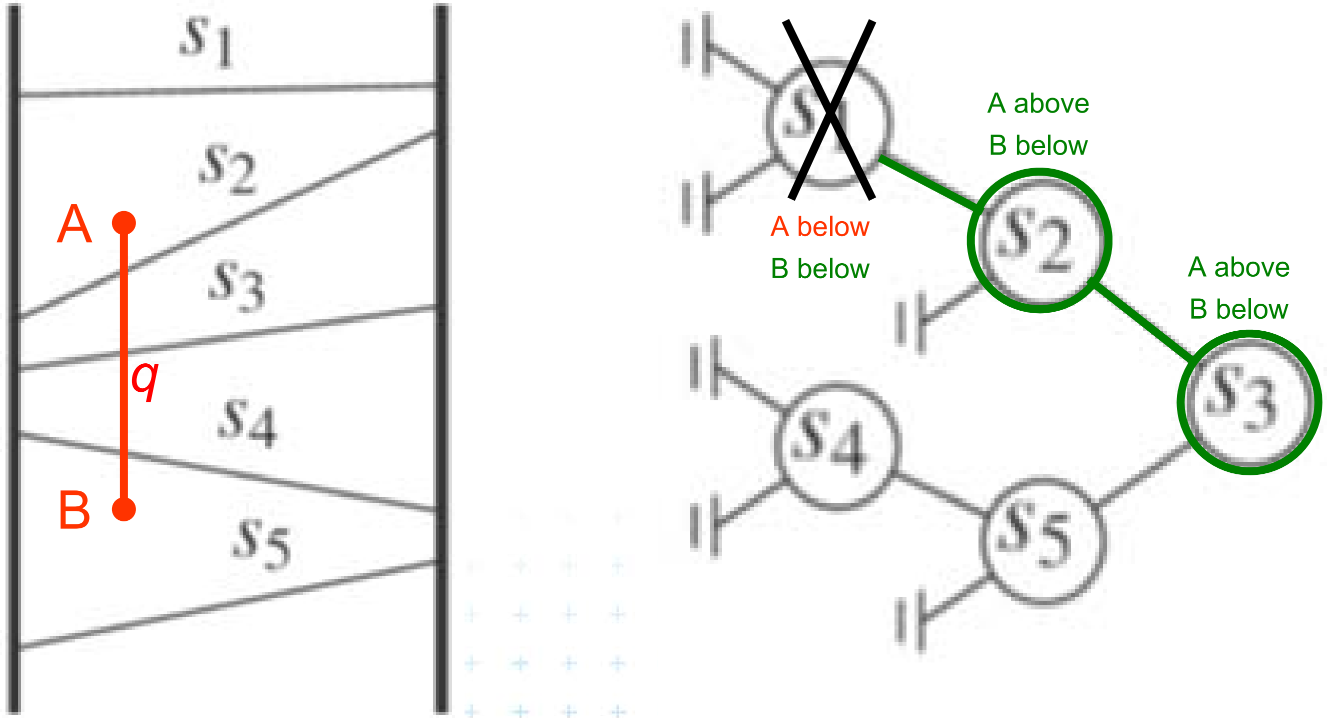
- Segment s is intersected by vert.query segment q iff
  - The lower endpoint (B) of q is below s and
  - The upper endpoint (A) of q is above s



[Berg]

Segments between vertical segment endpoints

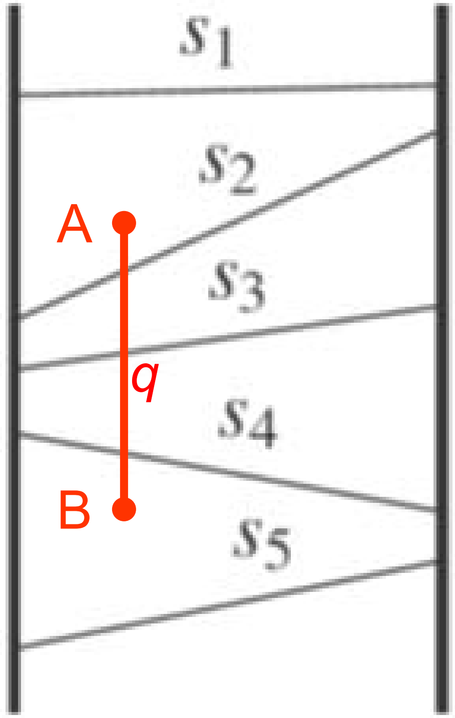
- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*

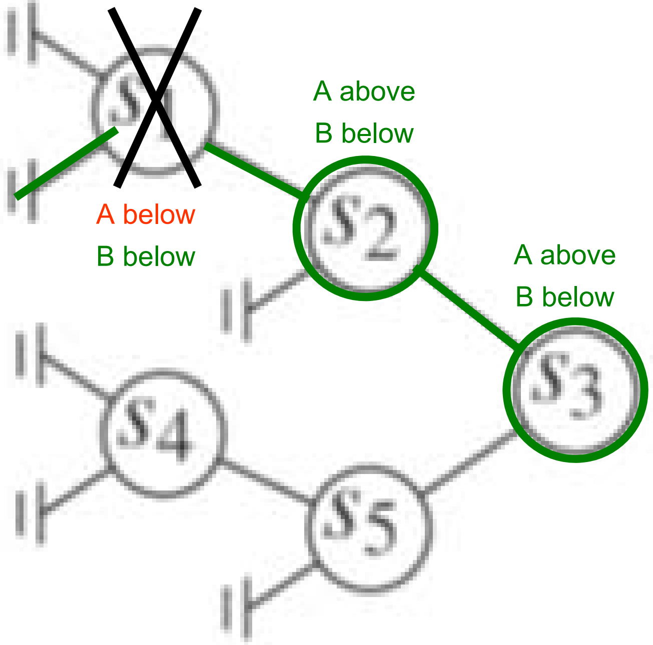


[Berg]

Segments between vertical segment endpoints

- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*

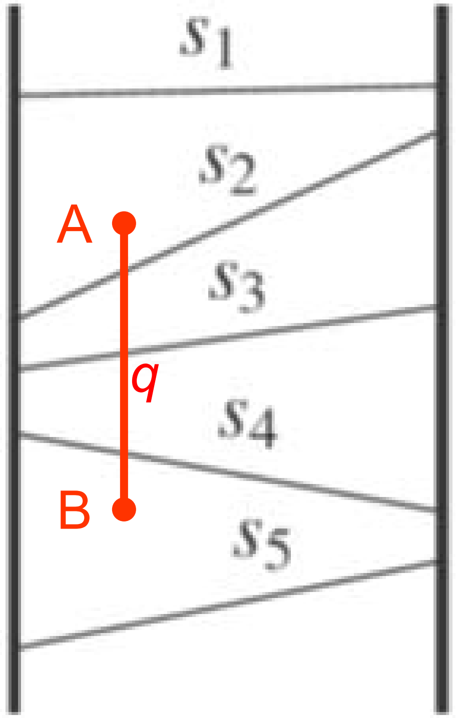


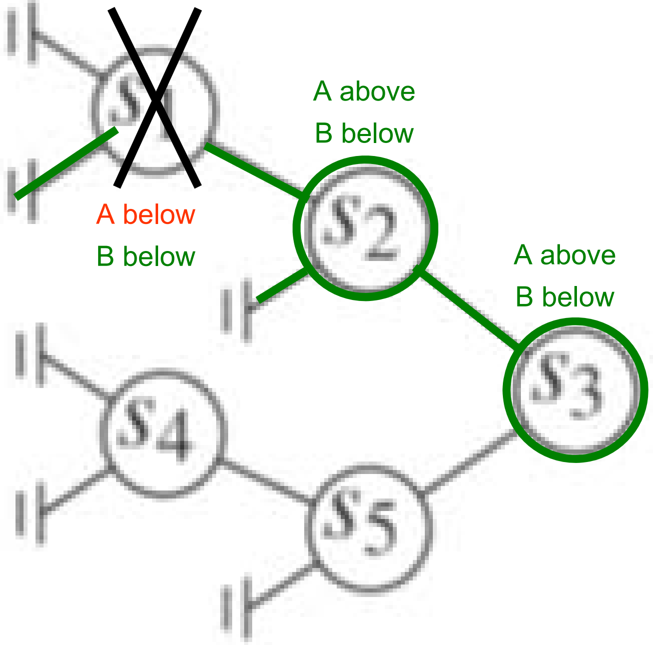


[Berg]

Segments between vertical segment endpoints

- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*

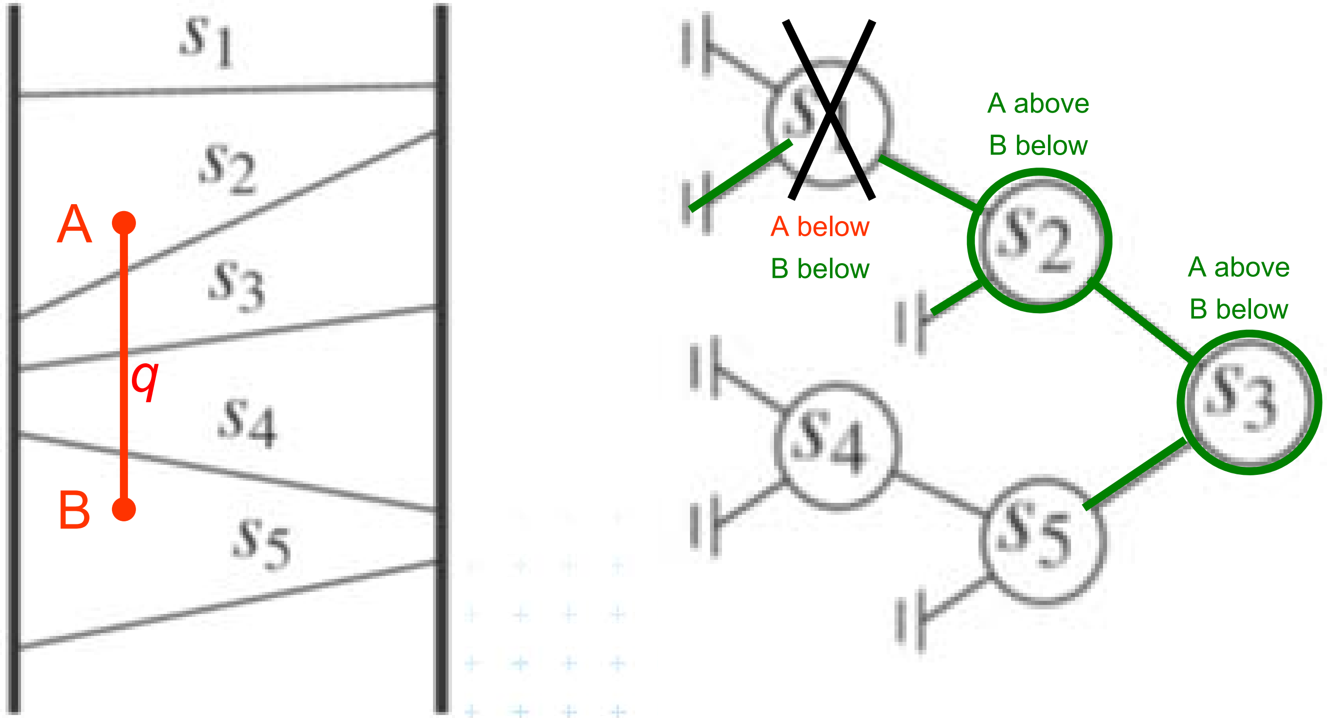




[Berg]

Segments between vertical segment endpoints

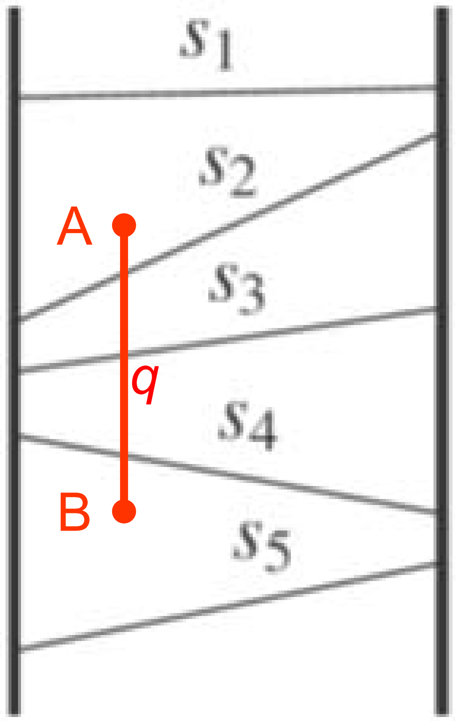
- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*

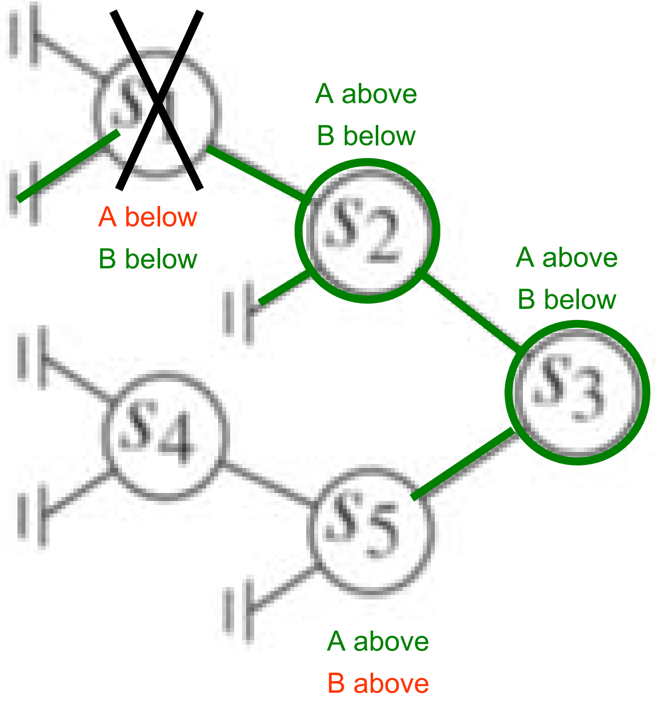


[Berg]

Segments between vertical segment endpoints

- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*

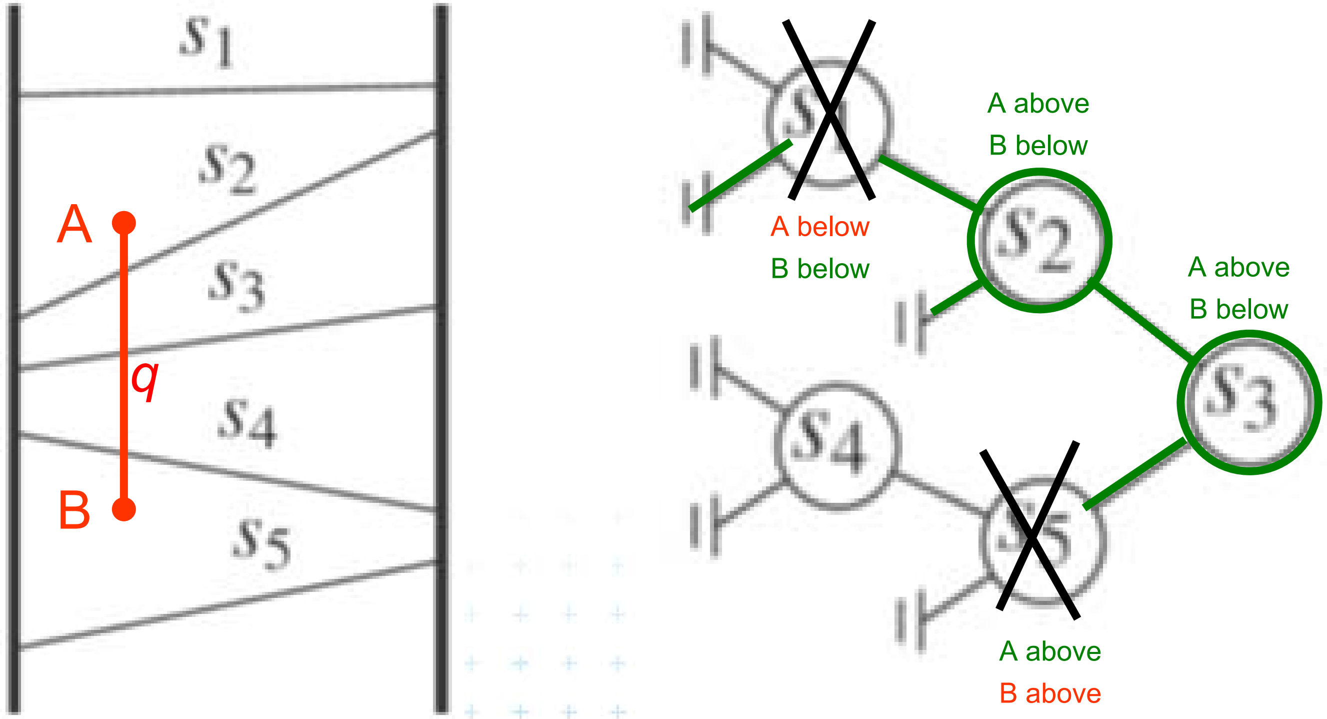




[Berg]

Segments between vertical segment endpoints

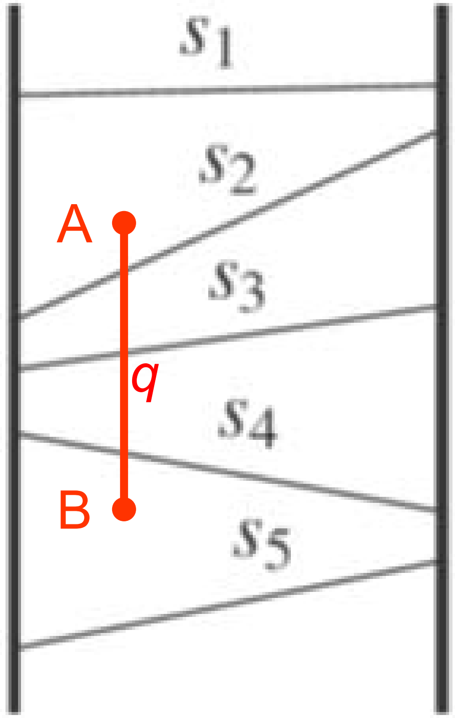
- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*

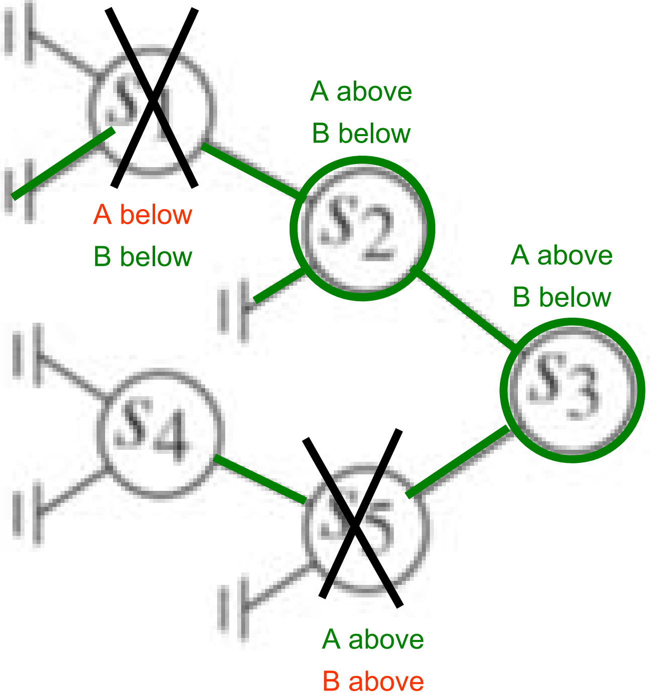


[Berg]

Segments between vertical segment endpoints

- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*

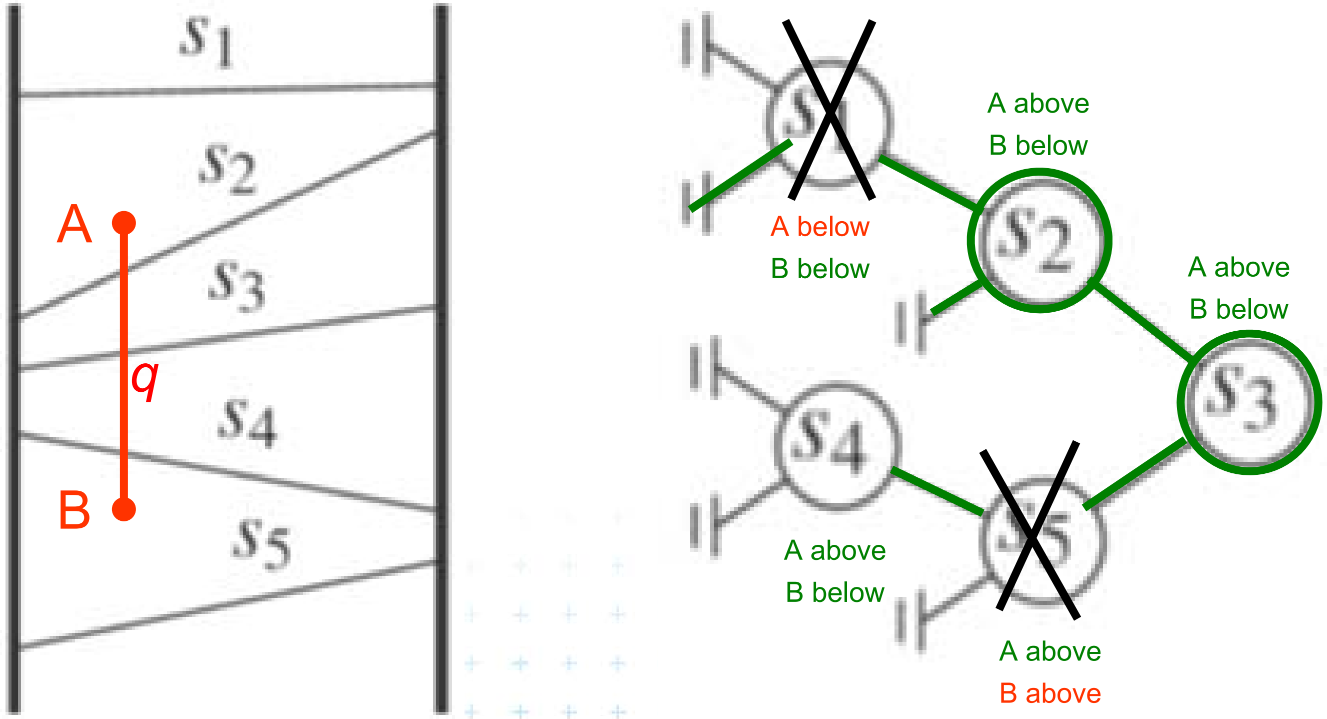




[Berg]

Segments between vertical segment endpoints

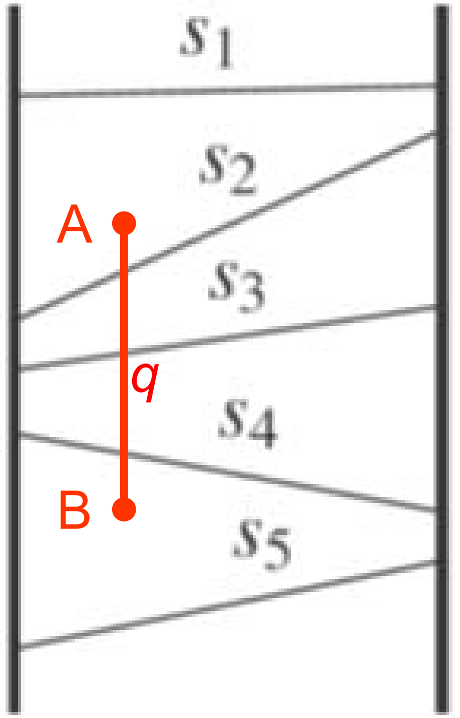
- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*

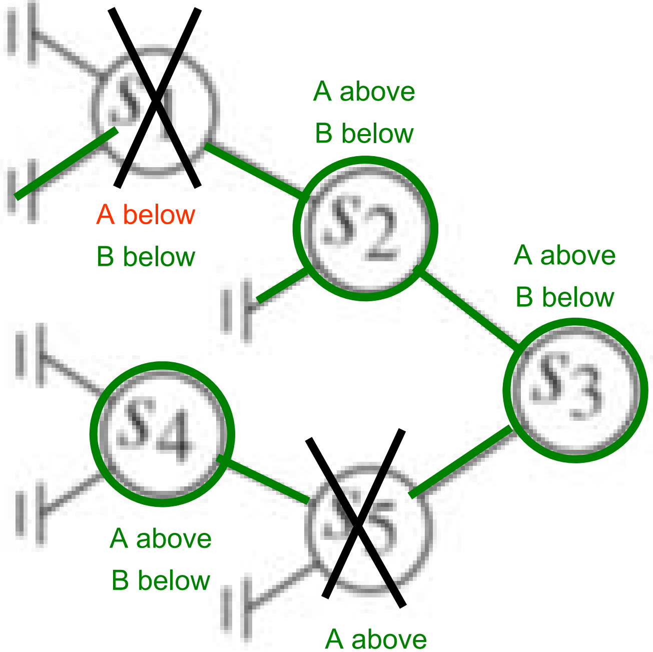


[Berg]

Segments between vertical segment endpoints

- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*





A above
B below

A below
B below

A above
B below

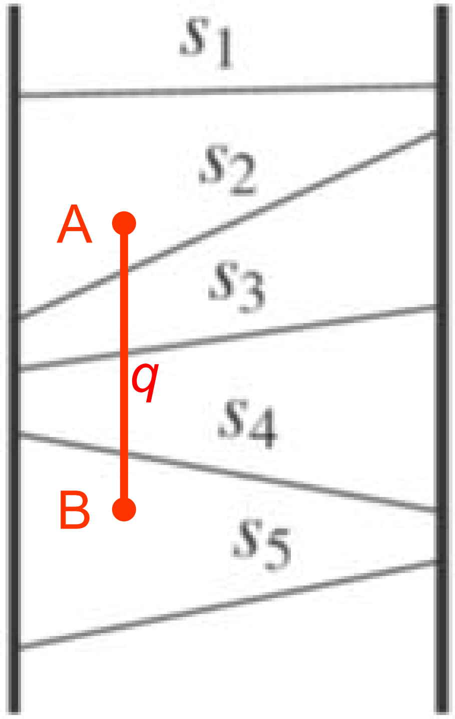
A above
B below

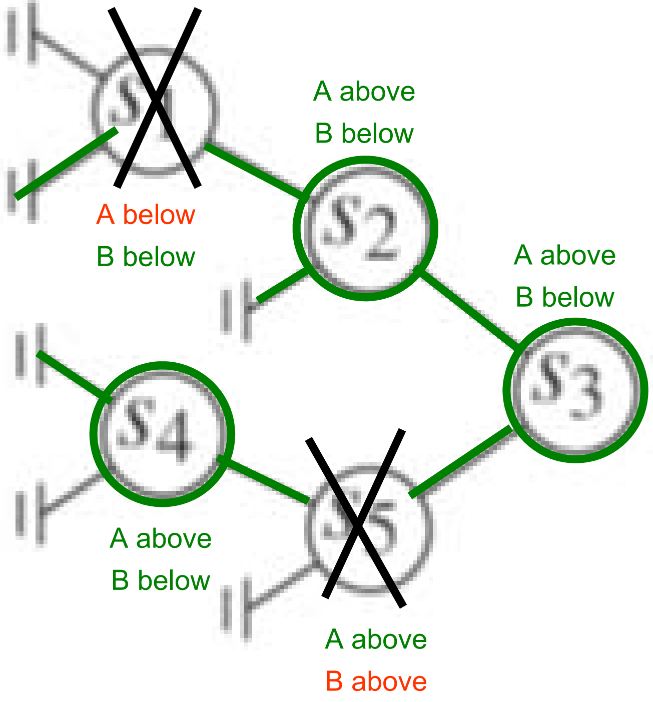
A above
B above

[Berg]

Segments between vertical segment endpoints

- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*

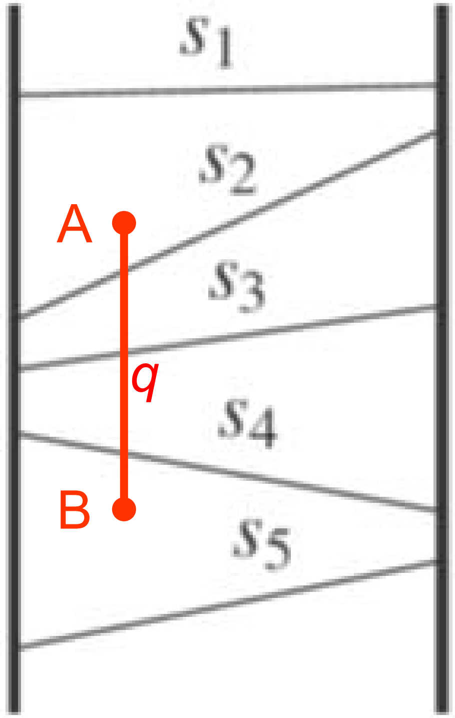


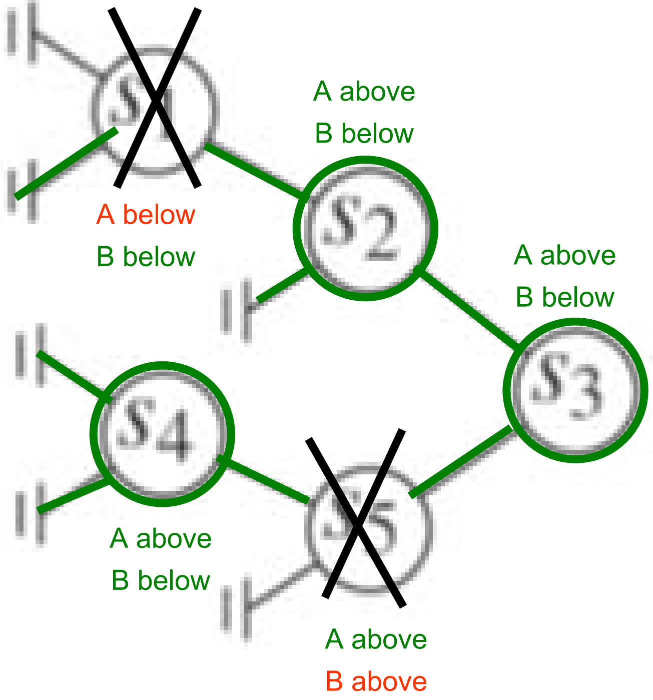


[Berg]

Segments between vertical segment endpoints

- Segment *s* is intersected by vert.query segment *q* iff
  - The lower endpoint (B) of *q* is below *s* and
  - The upper endpoint (A) of *q* is above *s*





[Berg]

Windowing of arbitrary oriented line segments complexity

Structure associated to node (BST) uses storage linear in the size of $S(v)$

- Build — $O(n \log n)$
- Storage — $O(n \log n)$
- Query — $O(k + \log^2 n)$
  - Report all segments that contain a query point
  - k is number of reported segments

Windowing of line segments in 2D – conclusions

Construction: all variants O(n logn)

1. Axis parallel	Search	Memory
i. Line (*sorted lists*)	O(k + log n)	O(n)
ii. Segment (*range trees*)	O(k + log^2 n)	O(n log n)
iii. Segment (*priority s. tr.*)	O(k + log n)	O(n)
2. In general position		
– *segment tree*	O(k + log^2 n)	O(n log n)

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

[Vigneron] Segment trees and interval trees, presentation, INRA, France, http://w3.jouy.inra.fr/unites/miaj/public/vigneron/cs4235/slides.html

[Schirra] Stefan Schirra. Geometrische Datenstrukturen. Sommersemester 2009 http://wwwisg.cs.uni-magdeburg.de/ag/lehre/SS2009/GDS/slides/S10.pdf

DCGI

DEPARTMENT OF COMPUTER GRAPHICS AND INTERACTION

ARRANGEMENTS (uspořádání)

PETR FELKEL

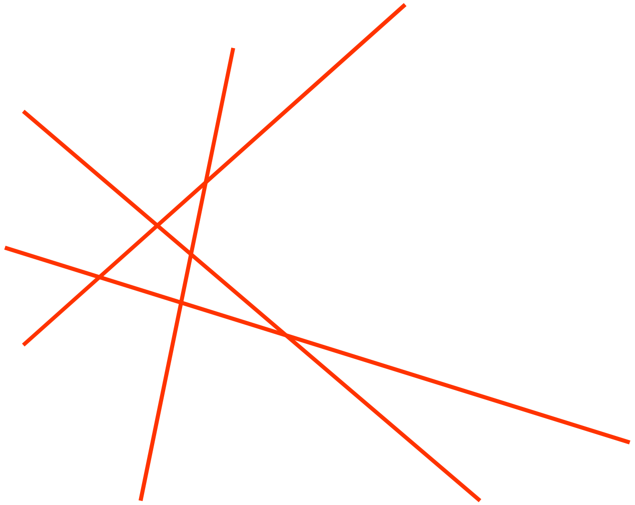
FEL CTU PRAGUE

Version from 25.1.2019

Talk overview

- Arrangements of lines
  - Incremental construction
  - Topological plane sweep

- Duality – next lesson



Arrangements

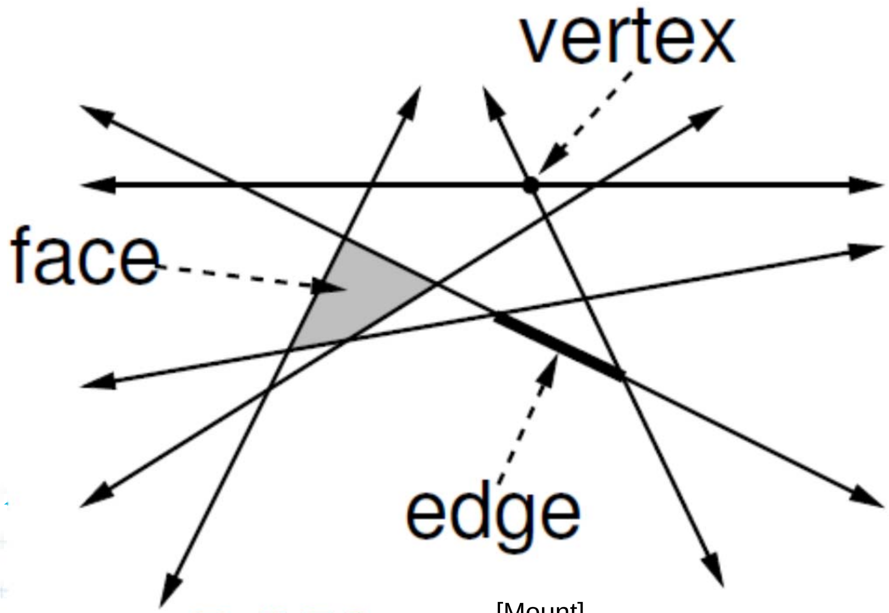
- The next most important structure in CG after CH, VD, and DT
- Possible in any dimension
  arrangement of (d-1)-dimensional hyperplanes
- We concentrate on arrangement of lines in plane
- Typical application: problems of point sets in dual plane (collinear points, point on circles, …)

Some more applications (see CGAL)

- Finding the minimum-area triangle defined by a set of points,
- computation of the sorted angular sequences of points,
- finding the ham-sandwich cut,
- planning the motion of a polygon translating among polygons in the plane,
- computing the offset polygon,
- constructing the farthest-point Voronoi diagram,
- coordinating the motion of two discs moving among obstacles in the plane,
- performing Boolean operations on curved polygons.

Line arrangement

- A finite set L of lines subdivides the plane into a cell complex, called arrangement $A(L)$
- In plane, arrangement defines a planar graph
  - Vertices – intersections of (2 or more) lines
  - Edges – intersection free segments (or rays or lines)
  - Faces – convex regions containing no line (possibly unbounded)



[Mount]

Line arrangement

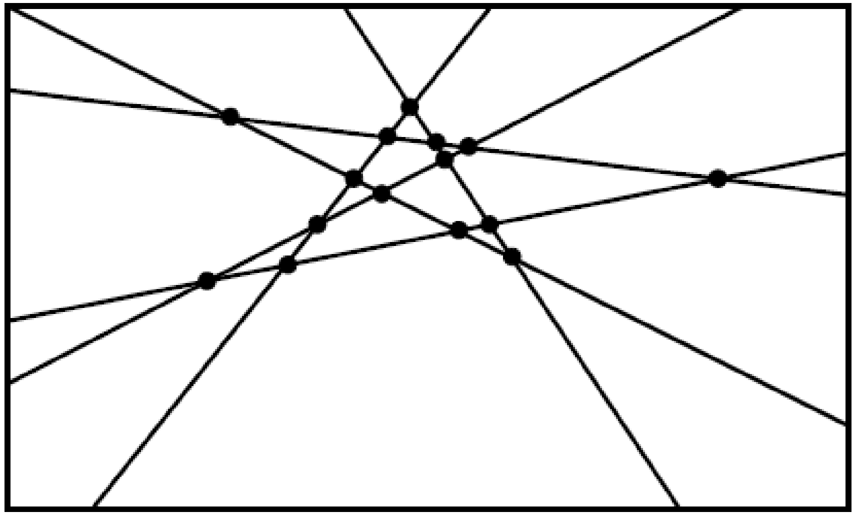
- Simple arrangement assumption

  = no three lines intersect in a single point

  - Can be solved by careful implementation or symbolic perturbation

Line arrangement

- Formal problem: graph must have bounded edges
  - – Topological fix: add vertex in infinity
  - – Geometrical fix: BBOX, often enough as abstract with corners $\{-\infty,-\infty\},\{\infty,\infty\}$



bounding box

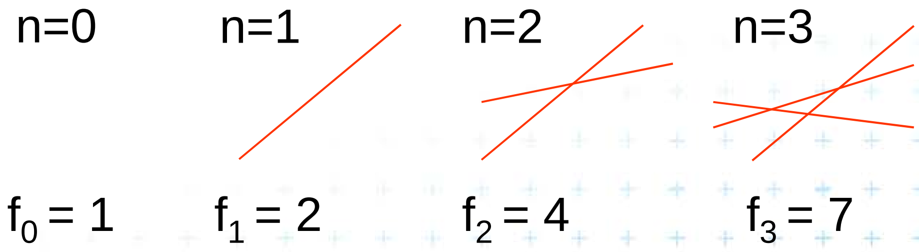
[Mount]

Combinatorial complexity of line arrangement

- $O(n^2)$
- Given *n* lines in general position, max numbers are
  - Vertices $\binom{n}{2} = \frac{n(n-1)}{2}$ - each line intersect n – 1 others
  - Edges n^2 - *n*–1 intersections create *n* edges on each of *n* lines
  - Faces $\frac{n(n+1)}{2} + 1 = \binom{n}{2} + n + 1$

$$f_0 = 1$$ (celá rovina)

$$f_n = f_{n-1} + n$$



$$f_n = f_0 + \sum_{i=1}^{n} i = \frac{n(n+1)}{2} + 1$$

Construction of line arrangement

$$n^2 \log n^2 = 2n^2 \log n = O(n^2 \log n)$$

(0. Plane sweep method)

- O(n^2 log n) time and O(n) storage plus O(n^2) storage for the arrangement (n^2 vertices, edges, faces. $\log n^2$ - heap & BVS access)

A. Incremental method

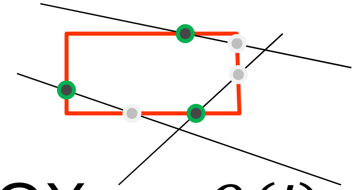
- O(n^2) time and O(n^2) storage
- Optimal method

B. Topological plane sweep

- O(n^2) time and O(n) storage only
- Does not store the result arrangement
- Useful for applications that may throw out the arrangement after processing

A. Incremental construction of arrangement

- $O(n^2)$ time, $O(n^2)$ space
  ~size of arrangement => it is an optimal algorithm
- Not randomized – depends on *n* only, not on order
- Add line l_i one by one $(i = 1 \ldots n)$
  – Find the leftmost intersection with the BBOX among $2(i-1)+4$ edges already on the BBOX …$O(i)$
  – Trace the line through the arrangement $A(L_{i-1})$ and split the intersected faces …$O(i)$ – why? See later
  – Update the subdivision (cell split) …$O(1)$



- Altogether $O(ni) = O(n^2)$

A. Incremental construction of arrangement

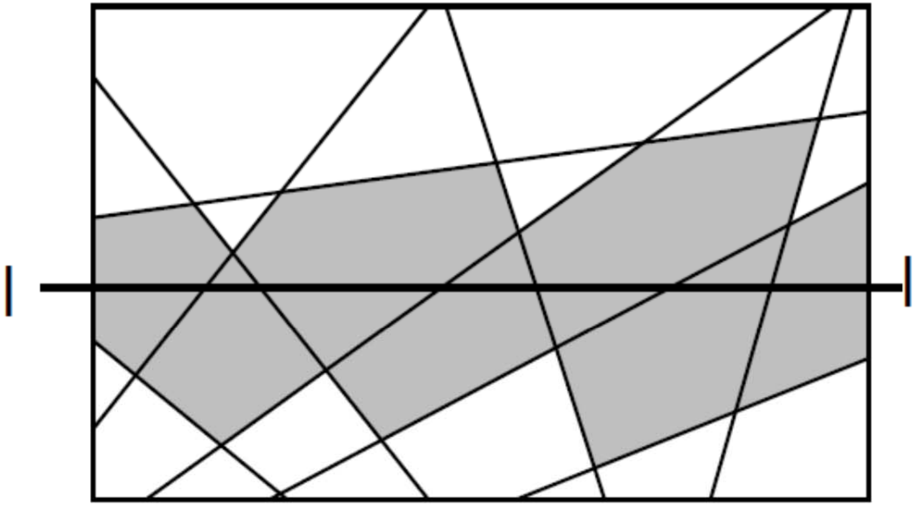
Arrangement(*L*)
Input: Set of lines *L* in general position (no 3 intersect in 1 common point)
Output: Line arrangement *A*(*L*) (resp. part of the arrangement stored in BBOX *B*(*L*) containing all the vertices of *A*(*L*))

```
1. Compute the BBOX $B(L)$ containing all the vertices of $A(L)$        …O($n^2$)
2. Construct DCEL for the subdivision induced by BBOX  $B(L)$
   …O(1)
3. for $i$ = 1 to $n$ do       // insert line $l_i$
4.     find edge $e$, where line $l_i$ intersects the BBOX of 2($i$-1)+4 edges …O($i$)
5.     $f$ = bounded face incident to the edge $e$
6.     while $f$ is in $B(L)$     (bounded face f  = f is in the BBOX)     … O(i)
7.             split $f$ and set $f$  to be the next intersected face
                           across the intersected edge
8.             update the DCEL (split the cell)                        …O(1)
```

See later…

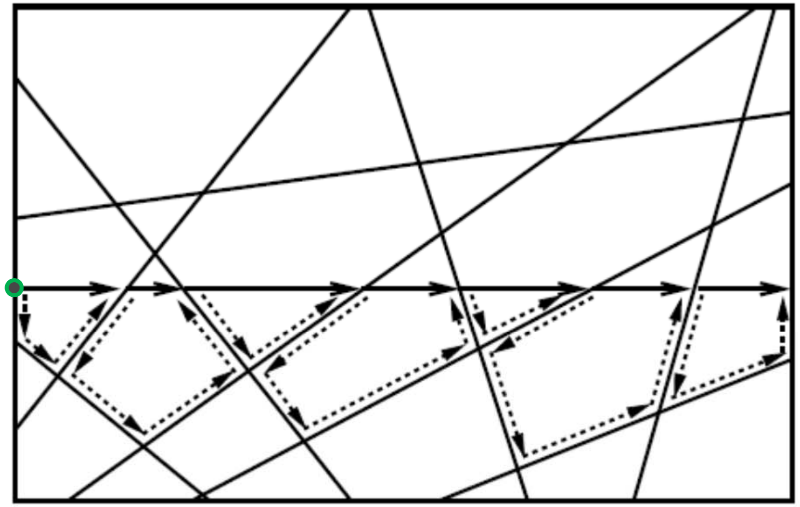
Tracing the line through the arrangement

- Walk around edges of current face (face walking)
- Determine if the line l_i intersects current edge *e*
- When intersection found, jump to the face on the other side of edge *e*



The zone of l

n=8 lines, 7 faces in the zone, 16 edges tested of max 48

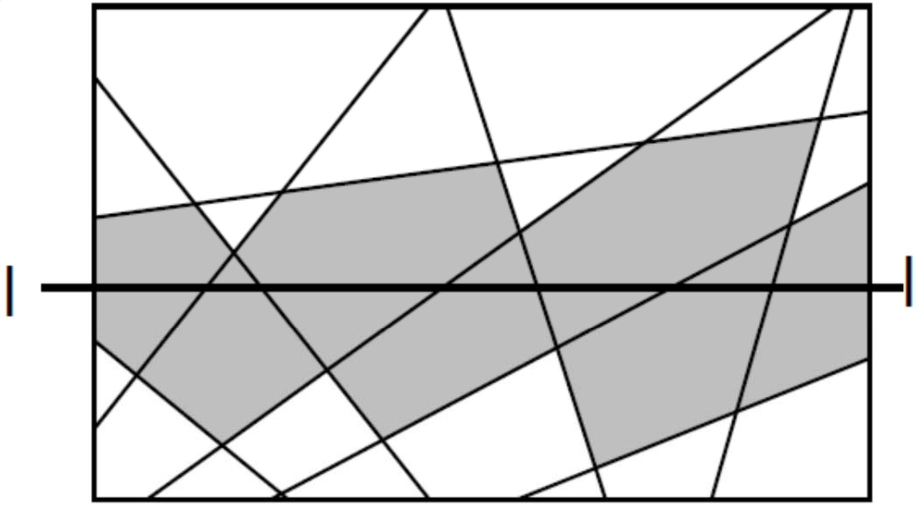


Walking the lower part of the zone

[Berg]

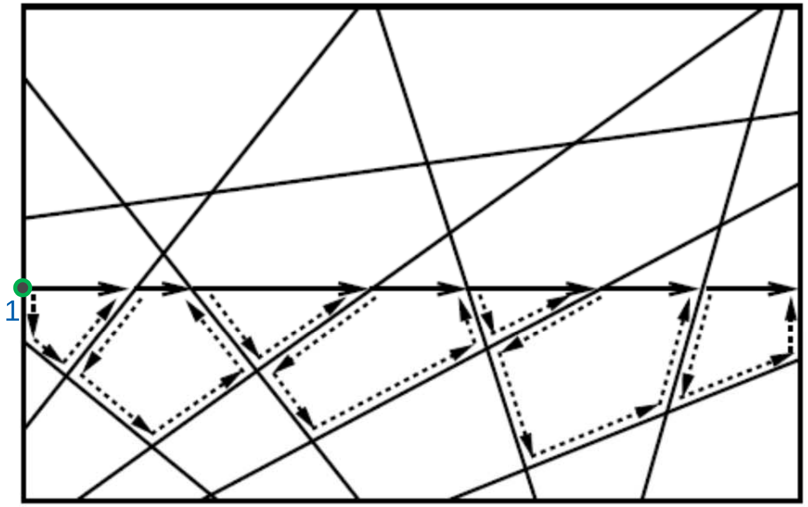
Tracing the line through the arrangement

- Walk around edges of current face (face walking)
- Determine if the line l_i intersects current edge *e*
- When intersection found, jump to the face on the other side of edge *e*



The zone of l

n=8 lines, 7 faces in the zone, 16 edges tested of max 48

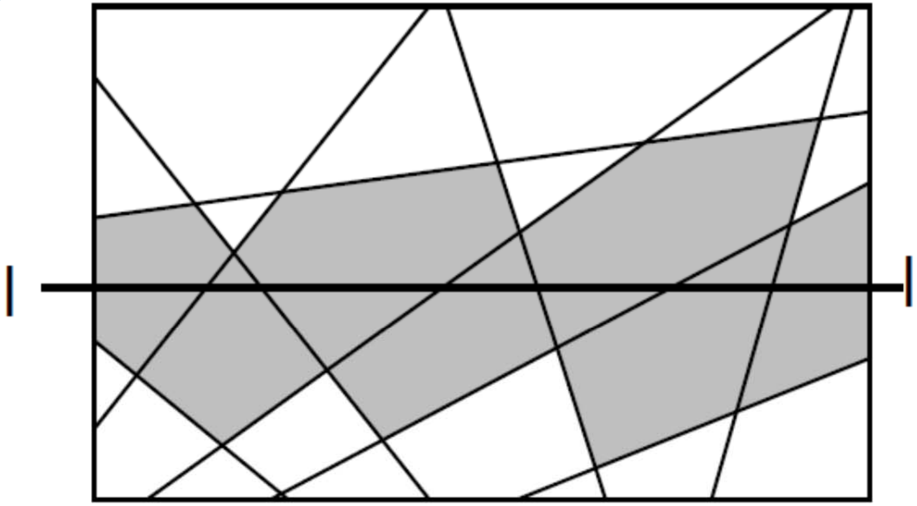


Walking the lower part of the zone

[Berg]

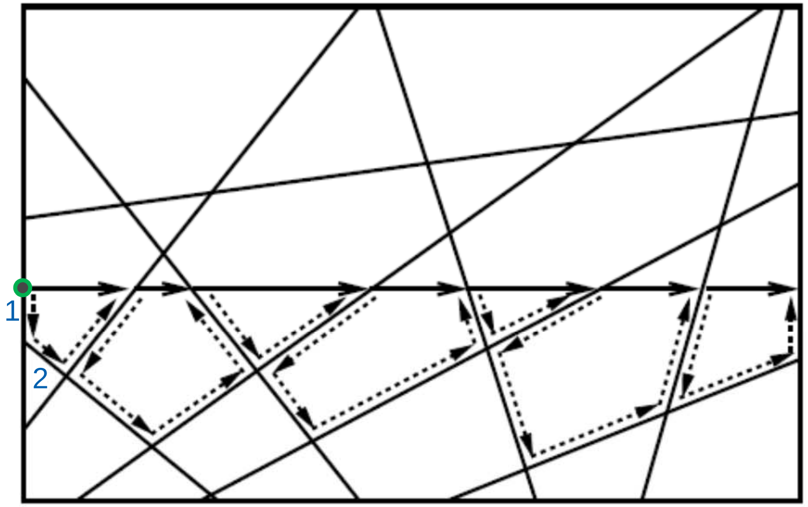
Tracing the line through the arrangement

- Walk around edges of current face (face walking)
- Determine if the line l_i intersects current edge e
- When intersection found, jump to the face on the other side of edge e



The zone of l

n=8 lines, 7 faces in the zone, 16 edges tested of max 48



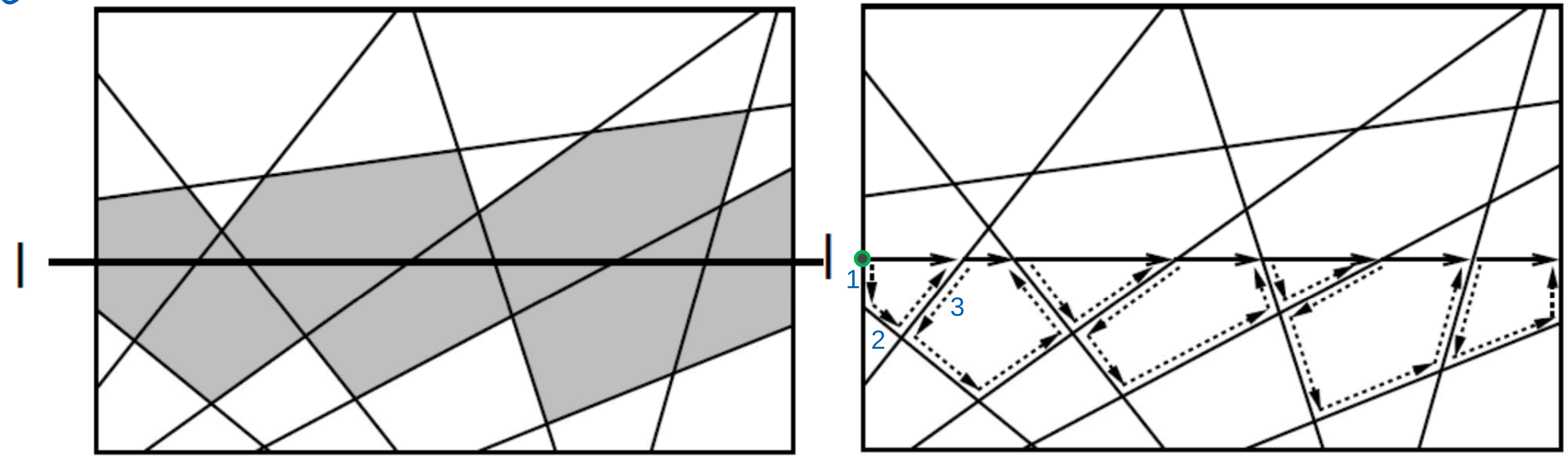
Walking the lower part of the zone

[Berg]

Tracing the line through the arrangement

- Walk around edges of current face (face walking)
- Determine if the line l_i intersects current edge e
- When intersection found, jump to the face on the other side of edge e

n=8 lines, 7 faces in the zone, 16 edges tested of max 48



The zone of l

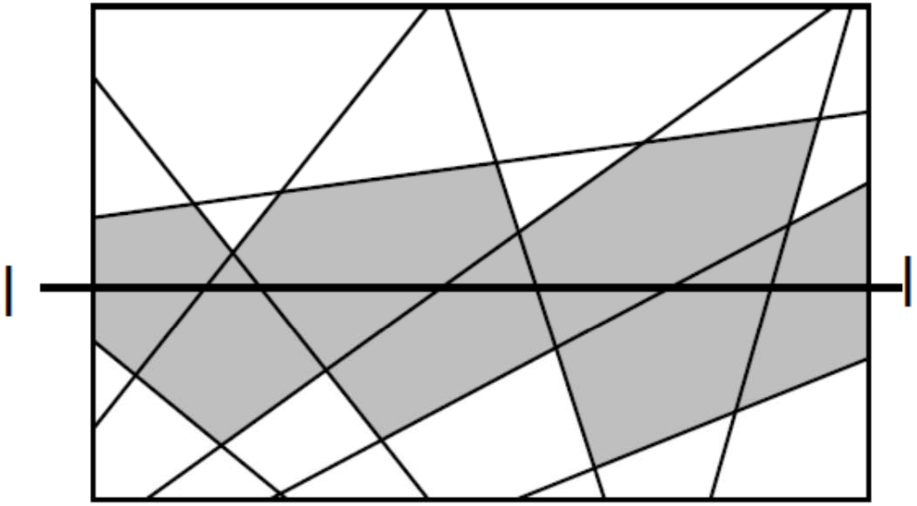
Walking the lower part of the zone

[Berg]

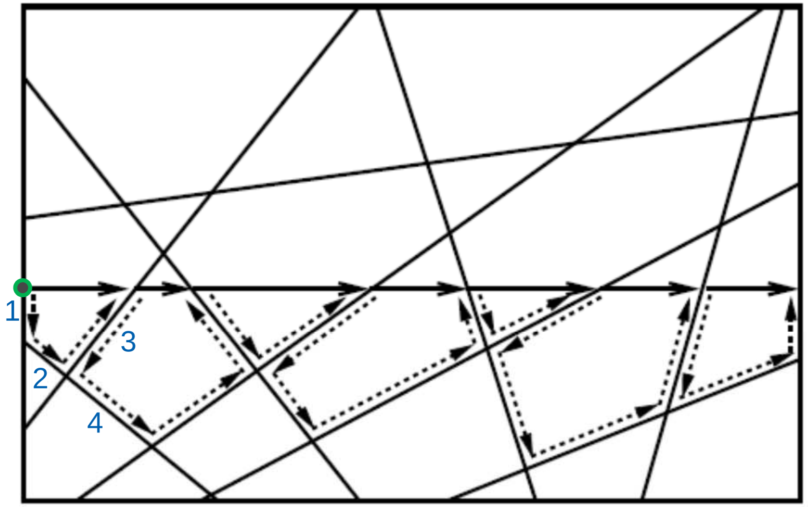
Tracing the line through the arrangement

- Walk around edges of current face (face walking)
- Determine if the line l_i intersects current edge e
- When intersection found, jump to the face on the other side of edge e

n=8 lines, 7 faces in the zone, 16 edges tested of max 48



The zone of l

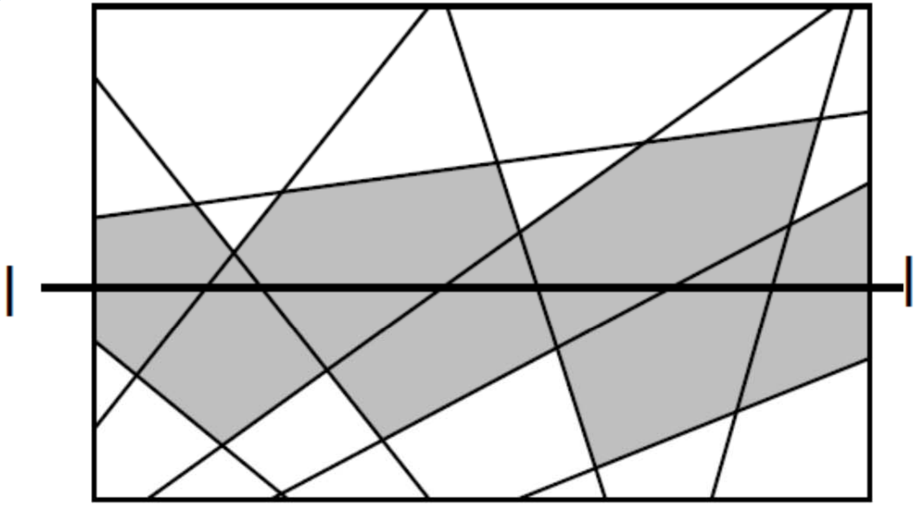


Walking the lower part of the zone

[Berg]

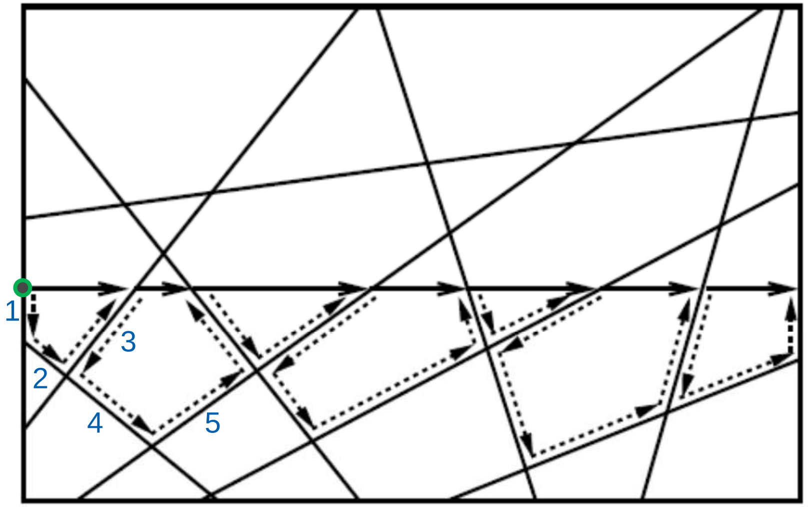
Tracing the line through the arrangement

- Walk around edges of current face (face walking)
- Determine if the line l_i intersects current edge e
- When intersection found, jump to the face on the other side of edge e



The zone of l

n=8 lines, 7 faces in the zone, 16 edges tested of max 48

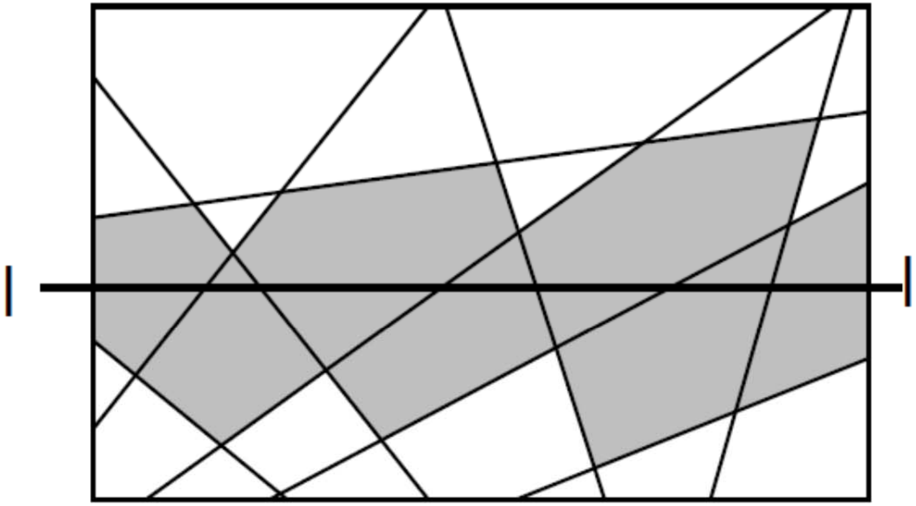


Walking the lower part of the zone

[Berg]

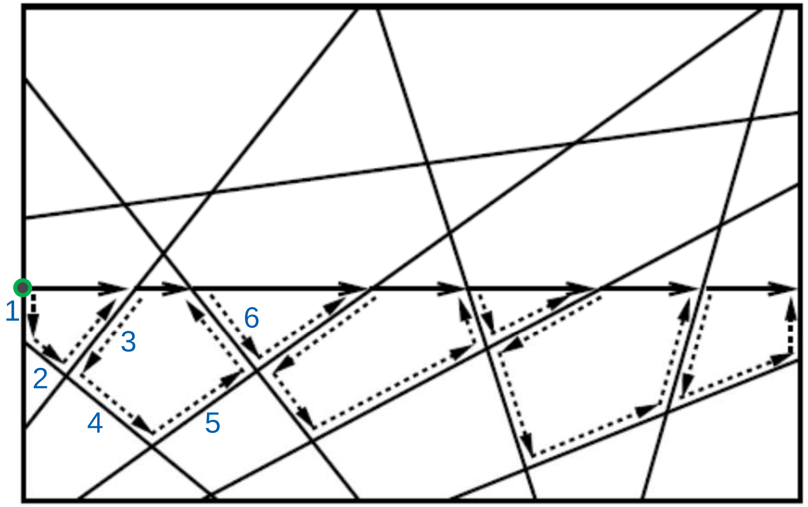
Tracing the line through the arrangement

- Walk around edges of current face (face walking)
- Determine if the line l_i intersects current edge *e*
- When intersection found, jump to the face on the other side of edge *e*



The zone of l

n=8 lines, 7 faces in the zone, 16 edges tested of max 48



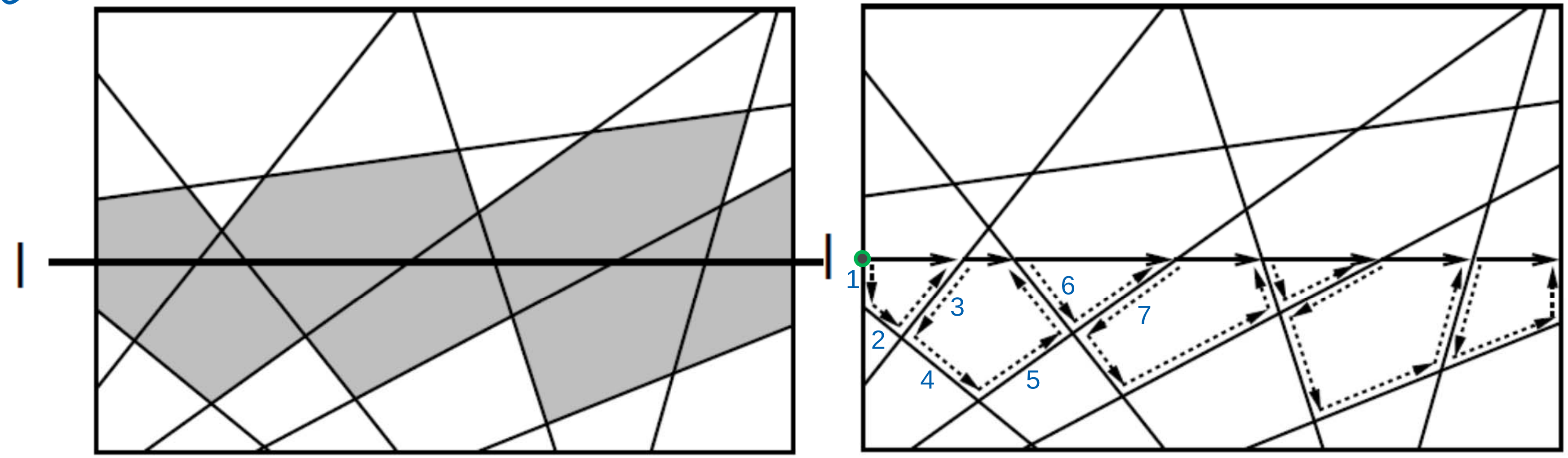
Walking the lower part of the zone

[Berg]

Tracing the line through the arrangement

- Walk around edges of current face (face walking)
- Determine if the line l_i intersects current edge *e*
- When intersection found, jump to the face on the other side of edge *e*

n=8 lines, 7 faces in the zone, 16 edges tested of max 48



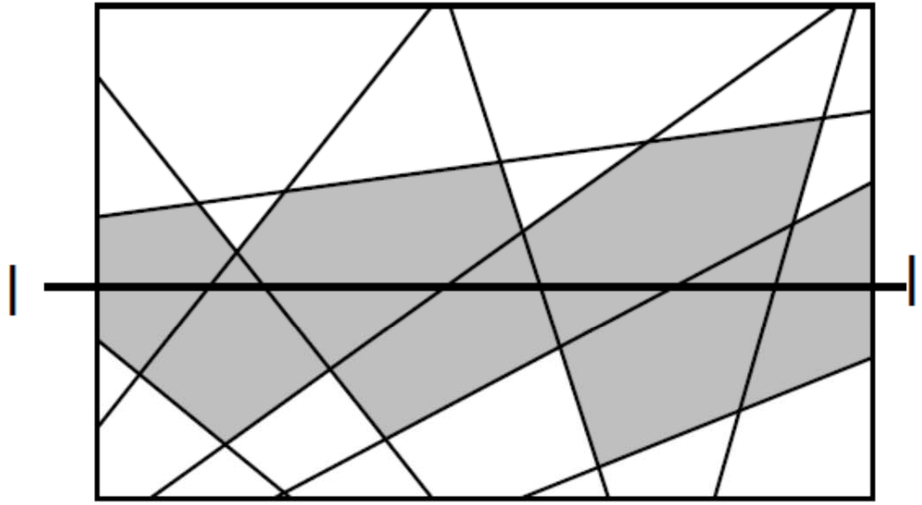
The zone of l

Walking the lower part of the zone

[Berg]

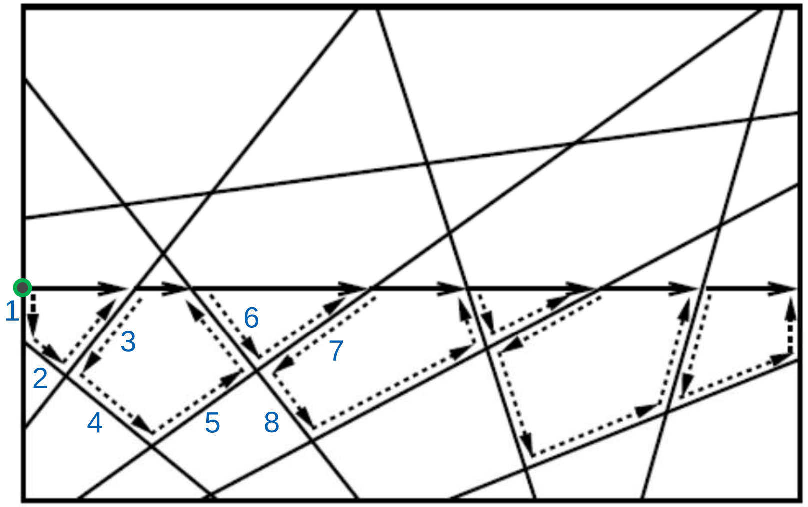
Tracing the line through the arrangement

- Walk around edges of current face (face walking)
- Determine if the line l_i intersects current edge *e*
- When intersection found, jump to the face on the other side of edge *e*



The zone of l

n=8 lines, 7 faces in the zone, 16 edges tested of max 48

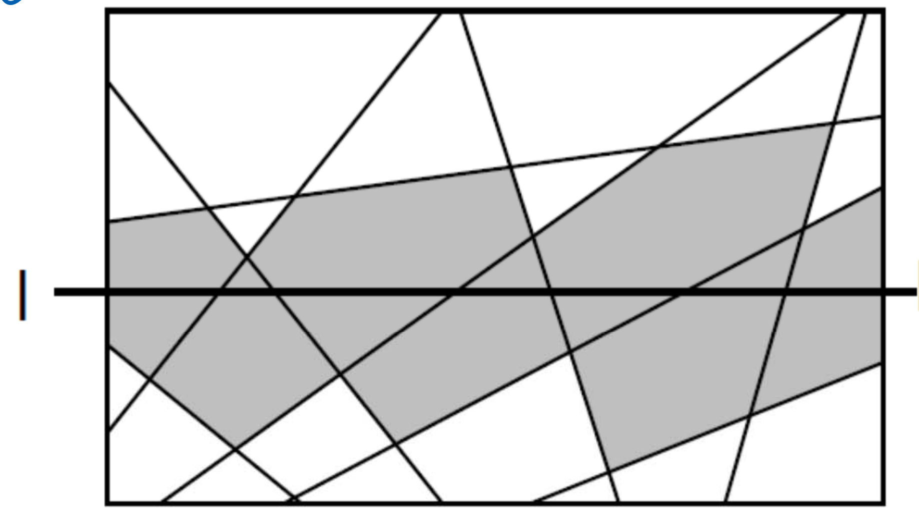


Walking the lower part of the zone

[Berg]

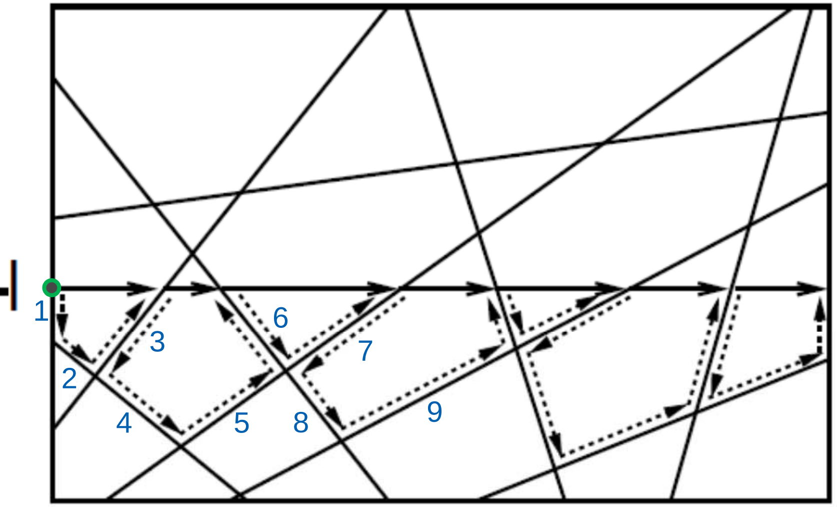
Tracing the line through the arrangement

- Walk around edges of current face (face walking)
- Determine if the line l_i intersects current edge e
- When intersection found, jump to the face on the other side of edge e



The zone of l

n=8 lines, 7 faces in the zone, 16 edges tested of max 48



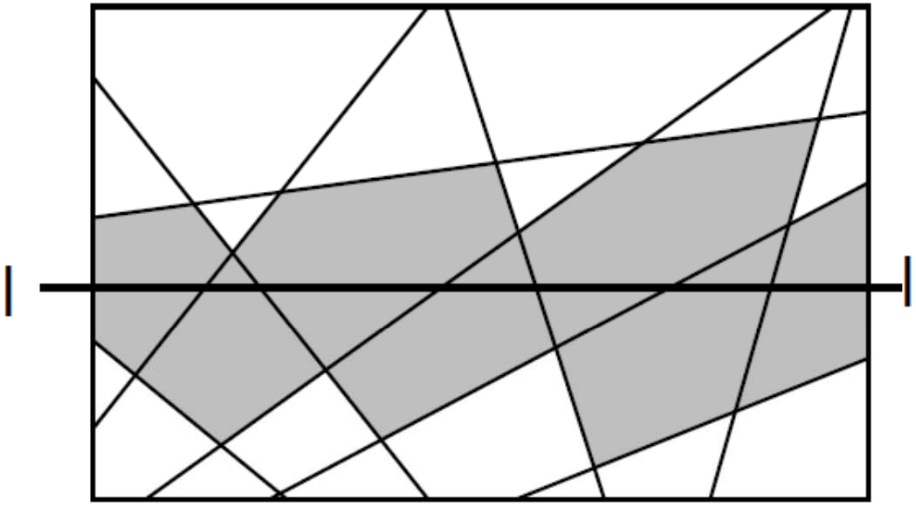
Walking the lower part of the zone

[Berg]

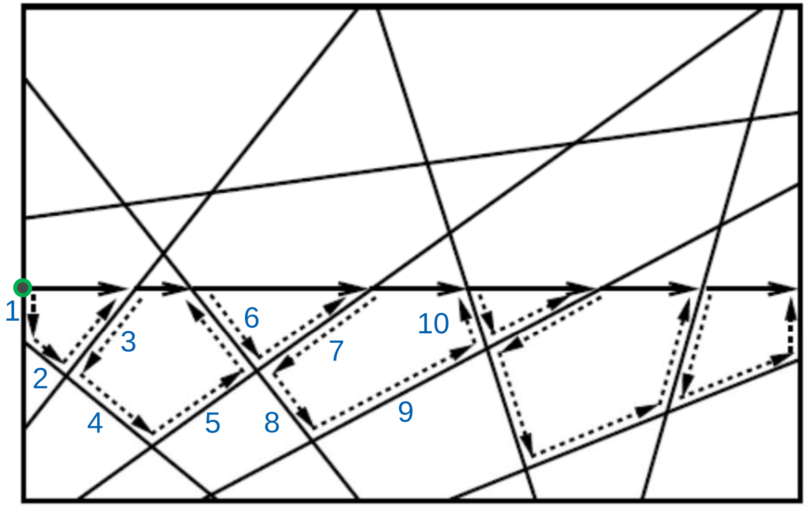
Tracing the line through the arrangement

- Walk around edges of current face (face walking)
- Determine if the line l_i intersects current edge *e*
- When intersection found, jump to the face on the other side of edge *e*

n=8 lines, 7 faces in the zone, 16 edges tested of max 48



The zone of l

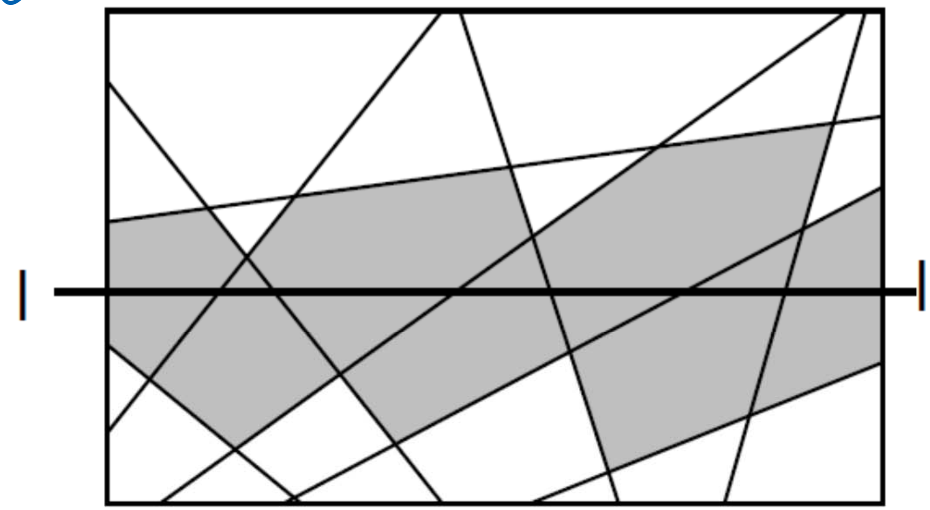


Walking the lower part of the zone

[Berg]

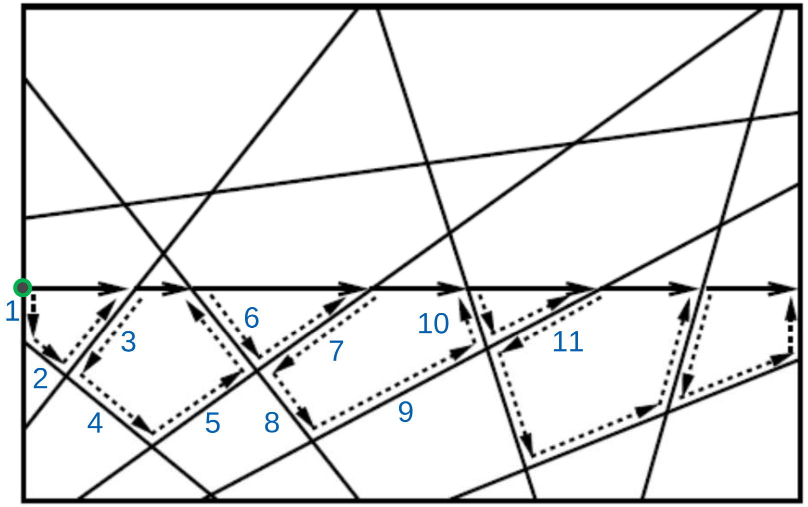
Tracing the line through the arrangement

- Walk around edges of current face (face walking)
- Determine if the line l_i intersects current edge *e*
- When intersection found, jump to the face on the other side of edge *e*



The zone of l

n=8 lines, 7 faces in the zone, 16 edges tested of max 48

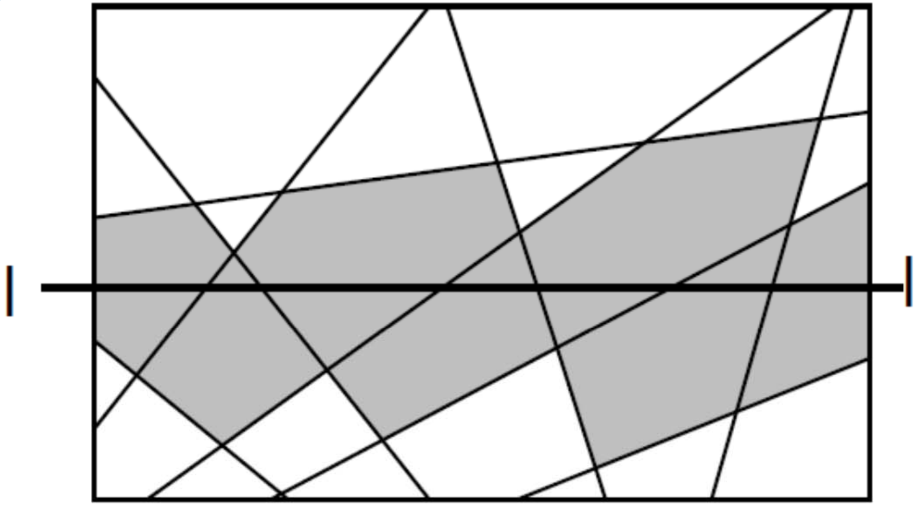


Walking the lower part of the zone

[Berg]

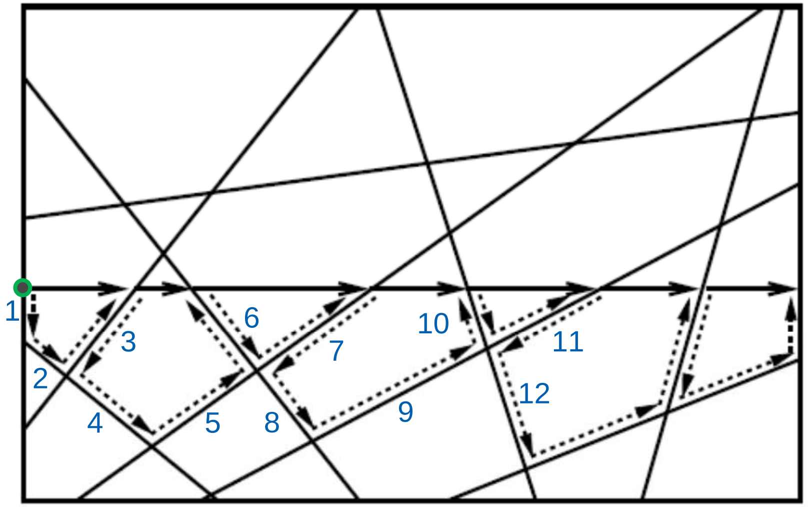
Tracing the line through the arrangement

- Walk around edges of current face (face walking)
- Determine if the line l_i intersects current edge e
- When intersection found, jump to the face on the other side of edge e



The zone of l

n=8 lines, 7 faces in the zone, 16 edges tested of max 48

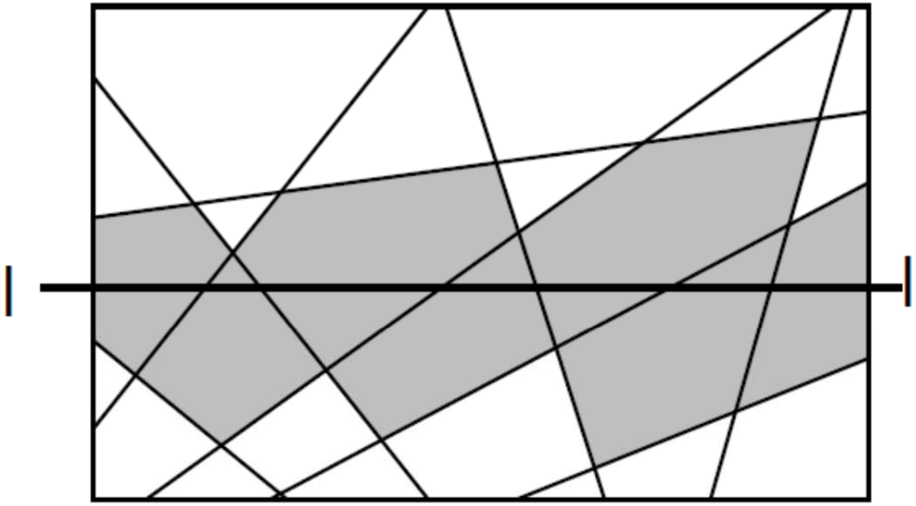


Walking the lower part of the zone

[Berg]

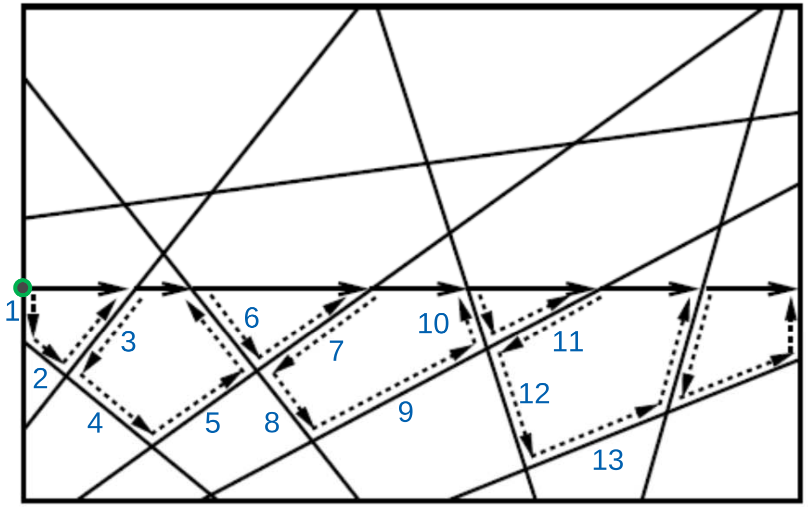
Tracing the line through the arrangement

- Walk around edges of current face (face walking)
- Determine if the line l_i intersects current edge *e*
- When intersection found, jump to the face on the other side of edge *e*



The zone of l

n=8 lines, 7 faces in the zone, 16 edges tested of max 48



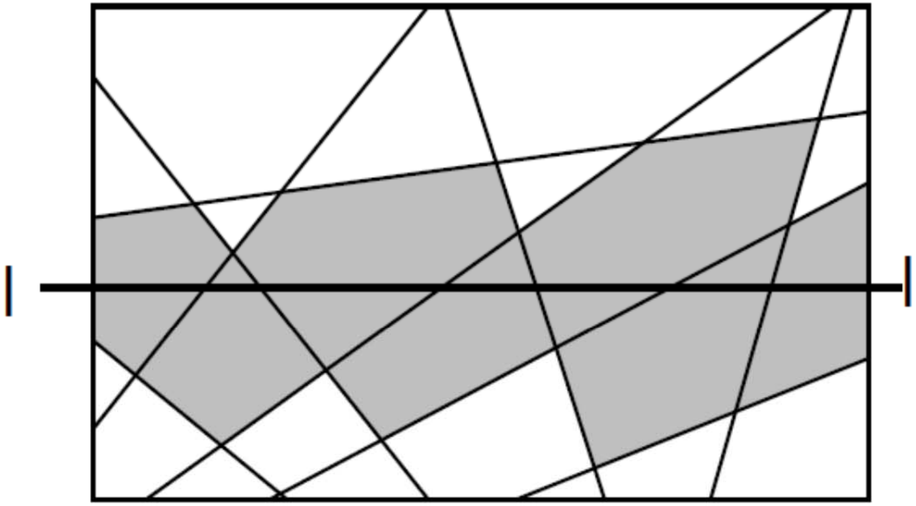
Walking the lower part of the zone

[Berg]

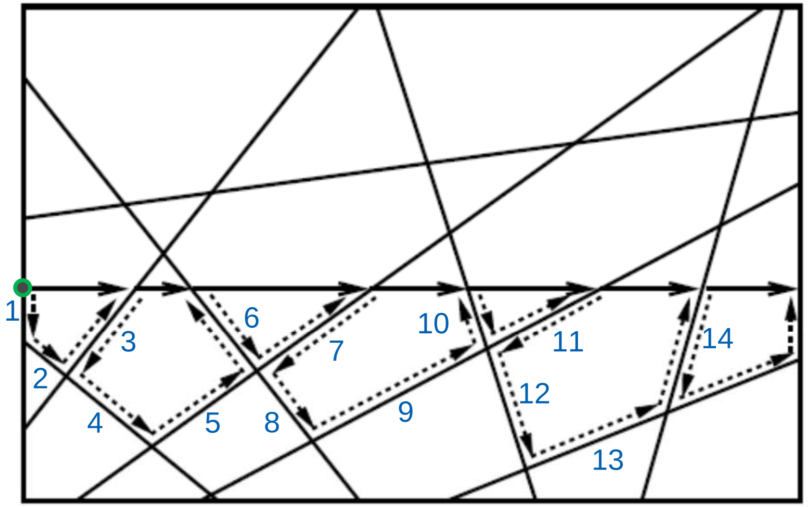
Tracing the line through the arrangement

- Walk around edges of current face (face walking)
- Determine if the line l_i intersects current edge *e*
- When intersection found, jump to the face on the other side of edge *e*

n=8 lines, 7 faces in the zone, 16 edges tested of max 48



The zone of l

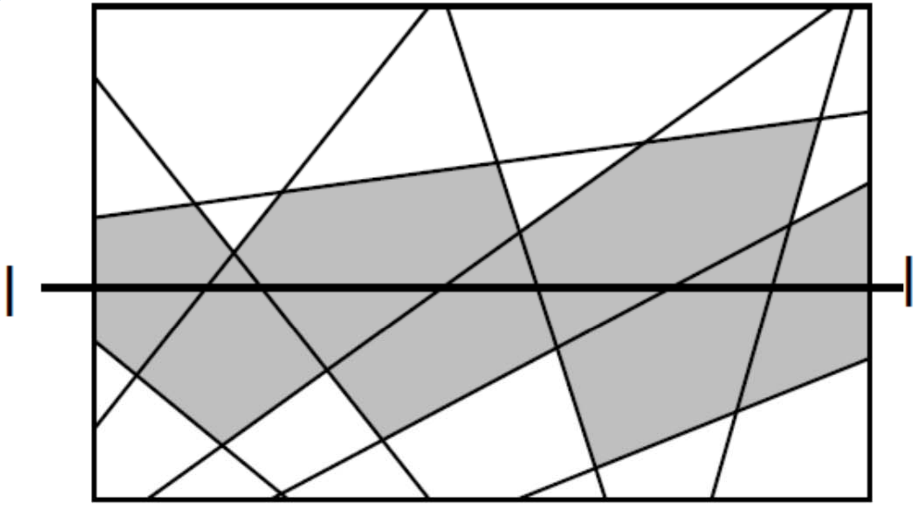


Walking the lower part of the zone

[Berg]

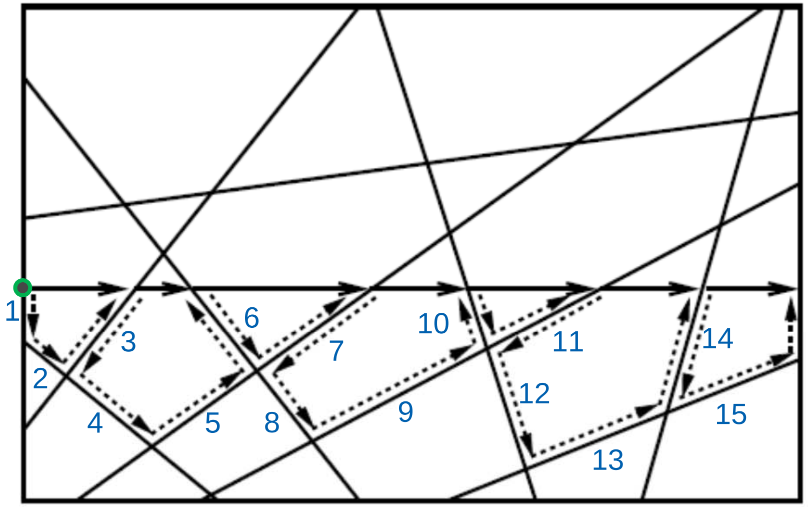
Tracing the line through the arrangement

- Walk around edges of current face (face walking)
- Determine if the line l_i intersects current edge *e*
- When intersection found, jump to the face on the other side of edge *e*



The zone of l

n=8 lines, 7 faces in the zone, 16 edges tested of max 48

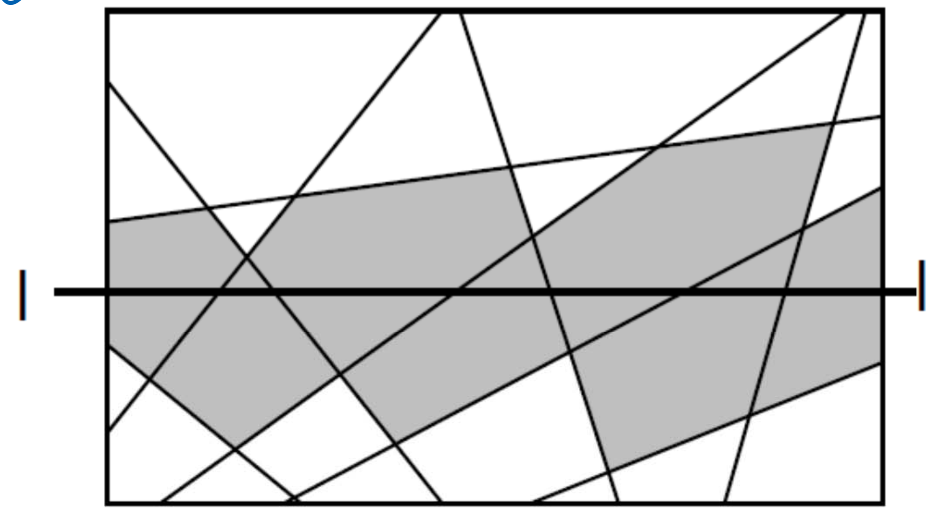


Walking the lower part of the zone

[Berg]

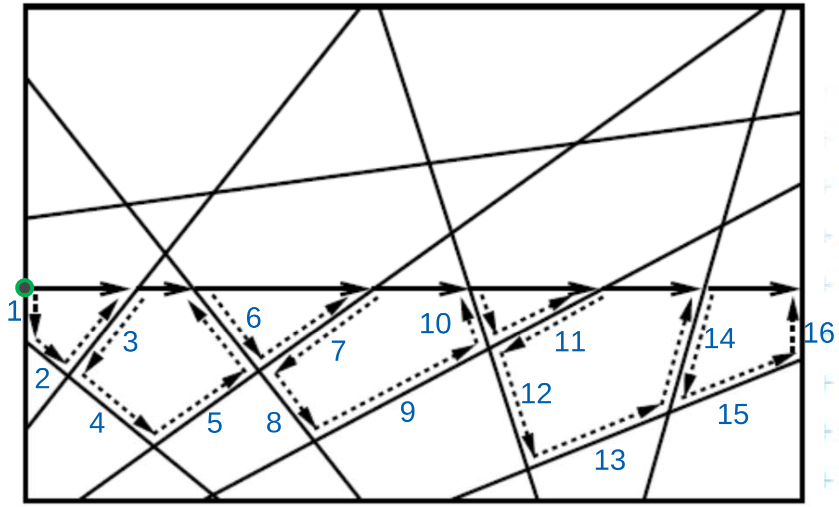
Tracing the line through the arrangement

- Walk around edges of current face (face walking)
- Determine if the line l_i intersects current edge *e*
- When intersection found, jump to the face on the other side of edge *e*



The zone of l

n=8 lines, 7 faces in the zone, 16 edges tested of max 48



Walking the lower part of the zone

[Berg]

Tracing the line through the arrangement

- Number of traversed edges determines the insertion complexity
- Naïve estimation would be $O(i^2)$ traversed edges (i faces, i lines per face, i^2 edges)
- According to the Zone theorem, it is $O(i)$ edges only!

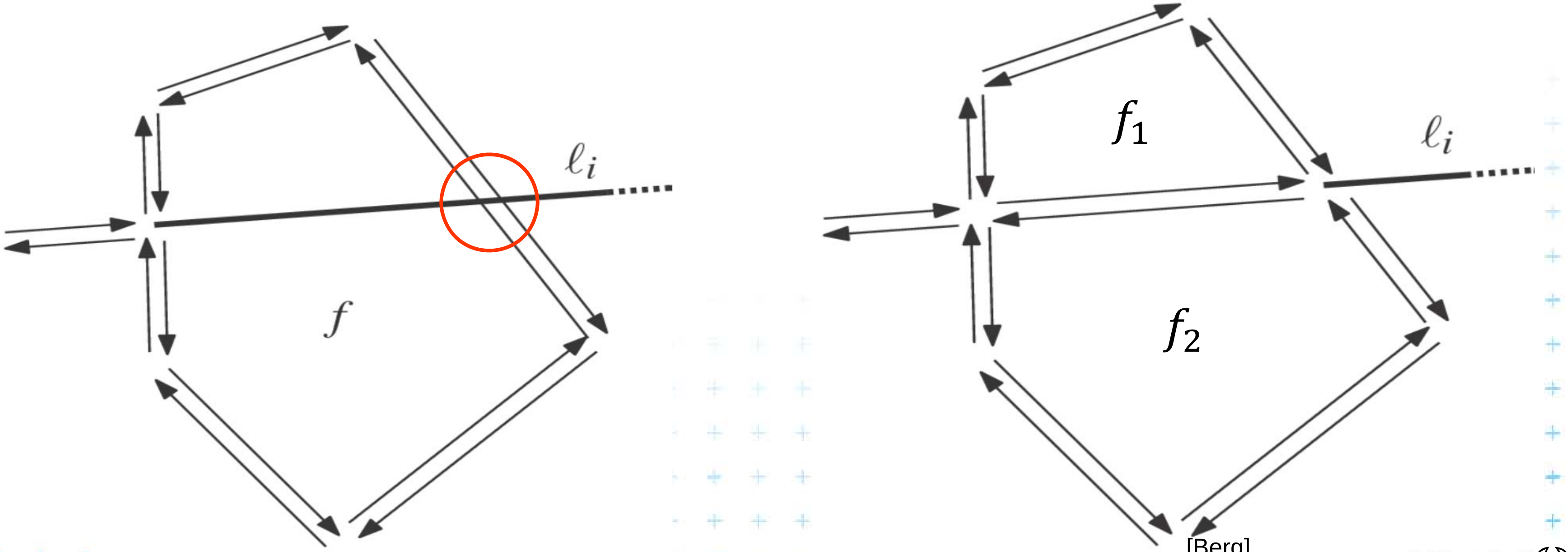
Zone theorem

= given an arrangement $A(L)$ of n lines in the plane and given any line l in the plane, the total number of edges in all the cells of the zone $Z_A(L)$ is at most $6n$.

For proof see [Mount, page 69]

Cell split in *O*(1)

- 1 new vertex
- 2 new face records, 1 face record (f) destroyed
- 3x2 new half-edges, 2 half-edges destroyed
- update pointers … O(1)



[Berg]

Complexity of incremental algorithm

- n insertions
- $O(i) = O(n)$ time for one line insertion
  instead of O(*i*^{2})
  (Zone theorem)

=> Complexity: $\underbrace{O(n^2)}_{\text{bbox}} + \underbrace{n.O(i)}_{\text{edges walked}} = O(n^2)$

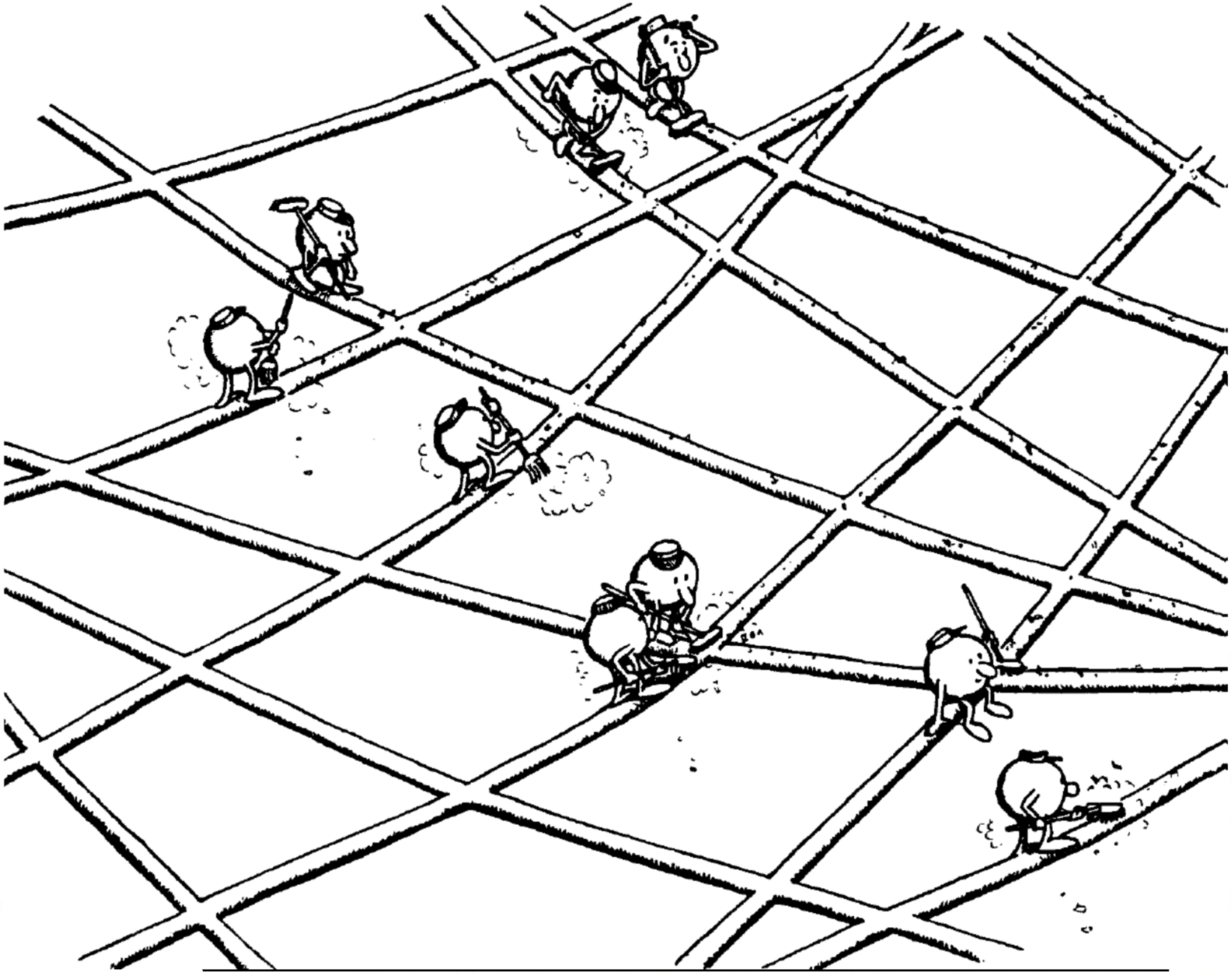
B. Topological plane sweep algorithm

- Complete arrangement needs $O(n^2)$ storage
- Often we need just to process each arrangement element just once – and we can throw it out then
- Classical Sweep line algorithm (for arrangement of lines)
  - – needs $O(n)$ storage
  - – needs $\log n$ for heap manipulation in $O(n^2)$ event points

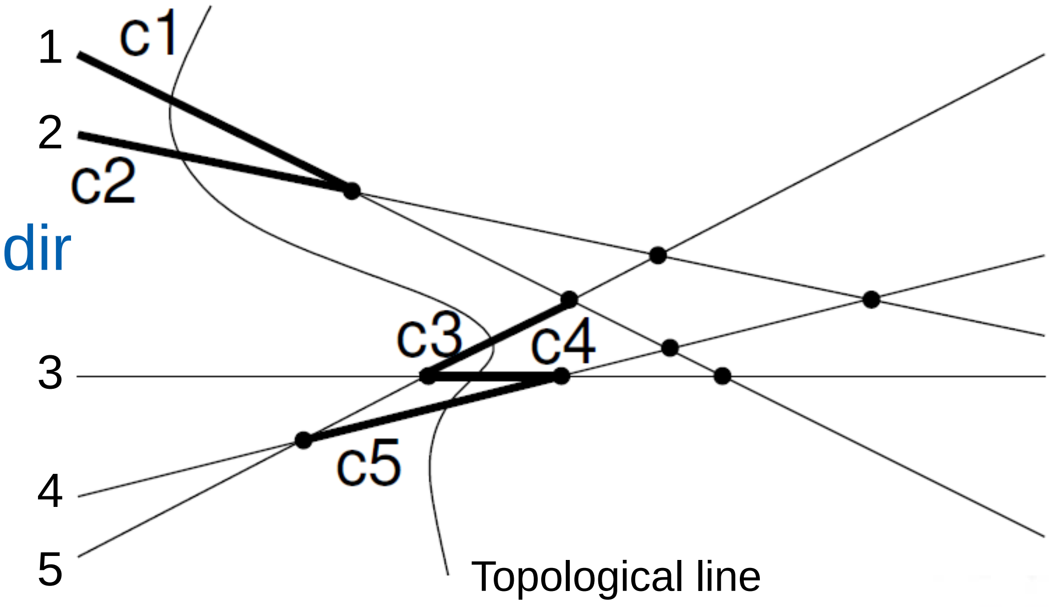
  => $O(n^2 \log n)$ algorithm
- Topological sweep line - TSL
  - – no $O(\log n)$ factor in time complexity
  - – array of *n* neighbors and a stack of ready vertices $O(1)$

  => $O(n^2)$ algorithm

Illustration from Edelsbrunner & Guibas



Topological line and cut



Topological line (curve)

(an intuitive notion)

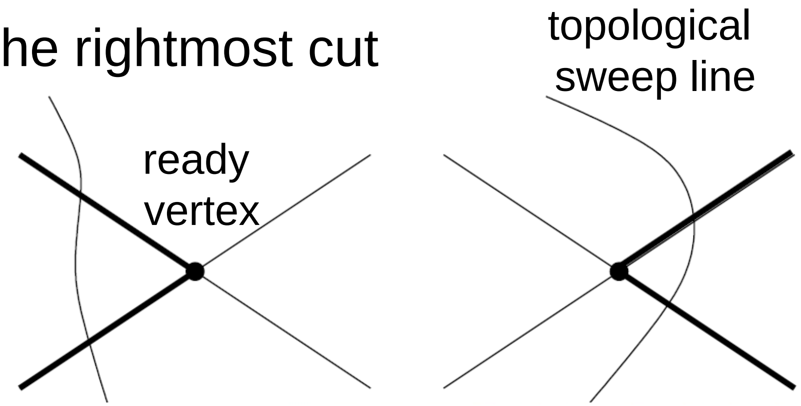
- Monotonic curve in y-dir
- intersects each line exactly once (as a sweep line)

Cut in an arrangement A

- is an ordered sequence of edges c_1, c_2,…,c_n in A (one taken from each line), such that for 1 ≤ *i* ≤ n-1, c_i and c_{i+1} are incident to the same face of A and c_i is above and c_{i+1} below the face
- Edges in the cut are not necessarily connected (as c_2 and c_3)

Topological plane sweep algorithm

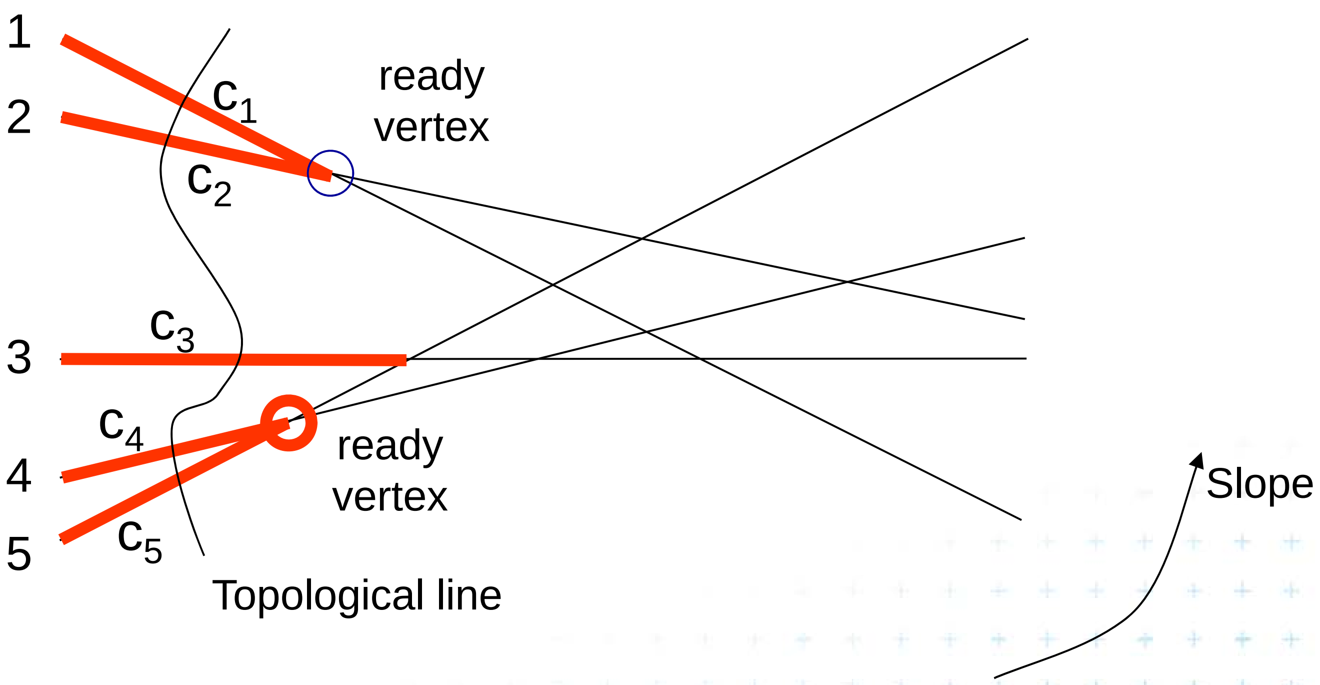
- Starts at the leftmost cut
  - – Consist of left-unbounded edges of *A* (ending at $-\infty$)
  - – Computed in O(*n* log *n*) time – order of slopes
- The sweep line is
  - – pushed from the leftmost cut to the rightmost cut
  - – Advances in elementary steps



- Elementary step

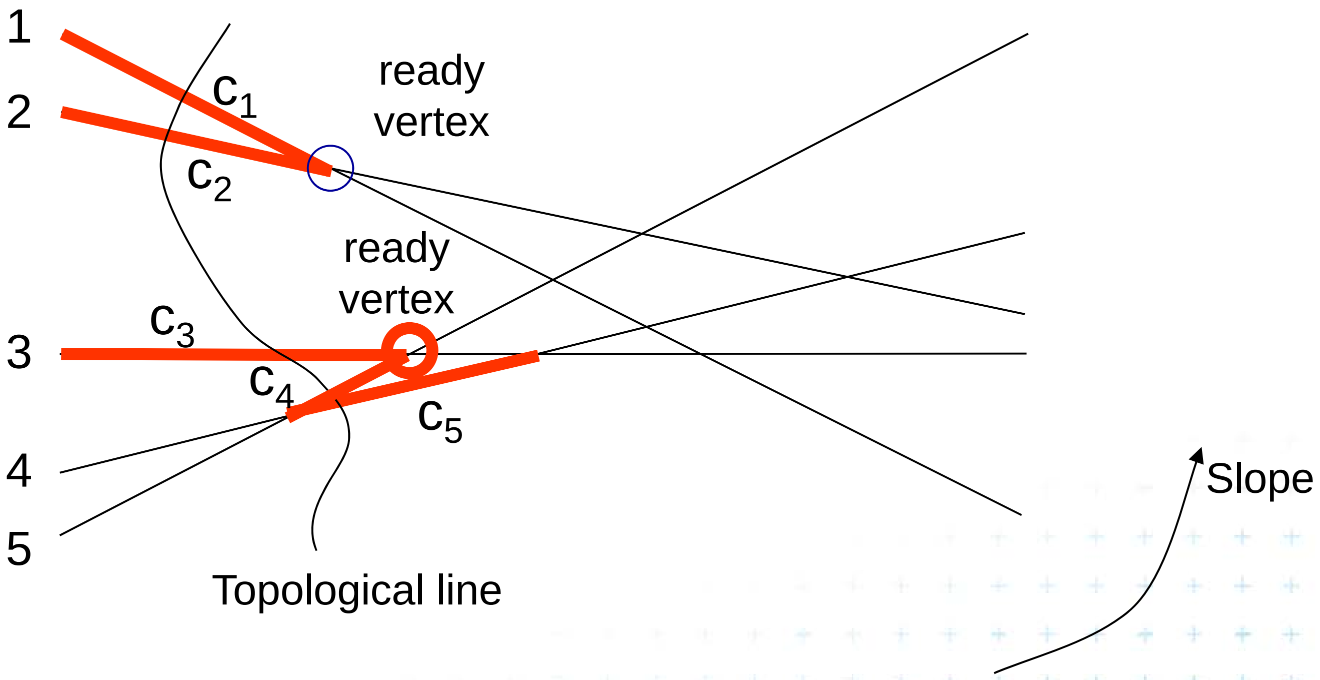
  = Processing of any *ready vertex* (intersection of consecutive edges at their right-point)
  - – Swaps the order of lines along the sweep line
  - – Is always possible (e.g., the point with smallest *x*)
  - – Searching of smallest x would need O(log *n*) time …

Step 0 – the leftmost cut

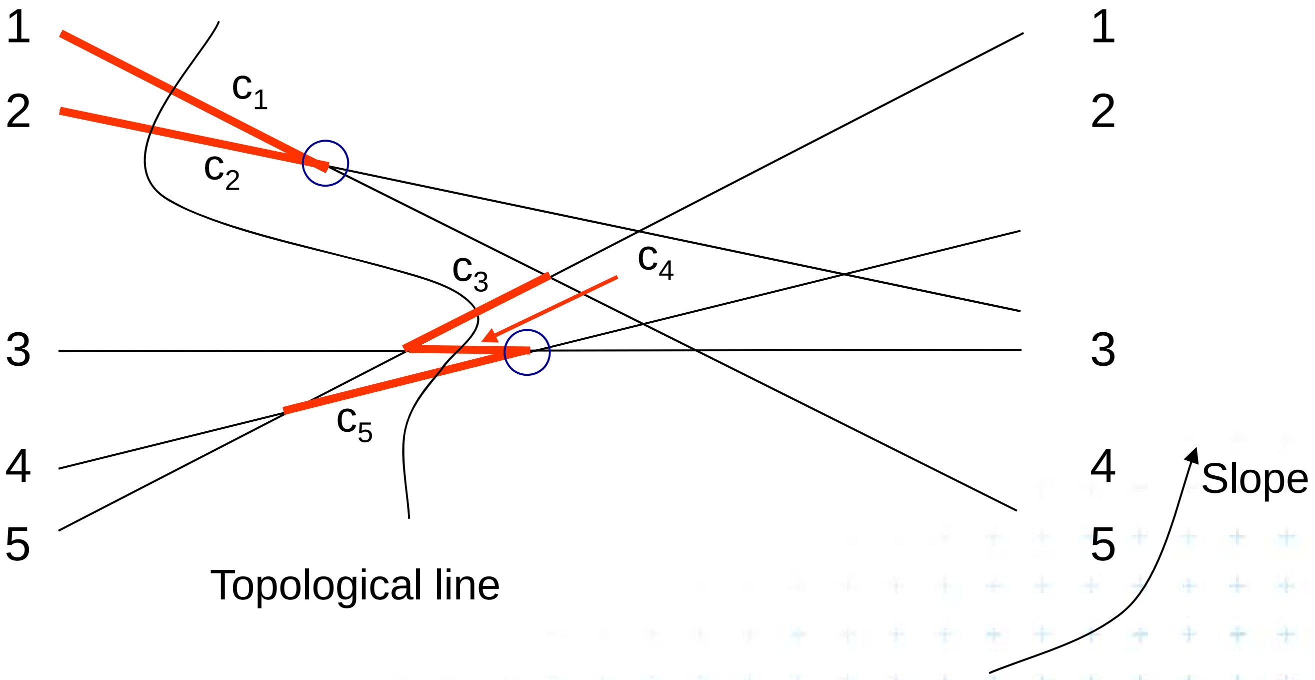


c_i = ordered sequence of edges along the topological sweep line

Step 1 – after processing of c4 x c5

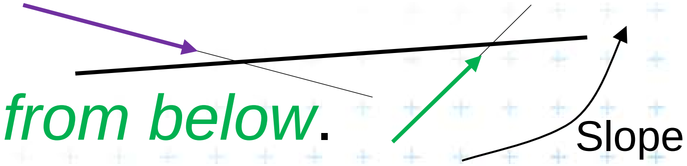


Step 2 – after processing of c3 x c4



How to determine the next right point?

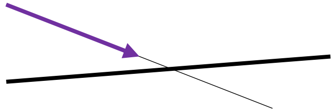
- Elementary step (intersection at edges right-point)
  - Is always possible (e.g., the point with smallest *x*)
  - But searching the smallest x would need O(log *n*) time
  - We need O(1) time
- Right endpoint of the edge in the cut results from
  - UHT a line of *smaller slope* intersecting it *from above* (traced from L to R) or
  - LHT line of *larger slope* intersecting it *from below*.



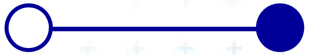
- Use Upper and Lower Horizon Trees (UHT, LHT)
  - Common segments of UHT and LHT belong to the cut
  - Intersect the trees, find pairs of consecutive edges
  - use the right points as legal steps (push to stack)

Upper and lower horizon tree

- Upper horizon tree (UHT)



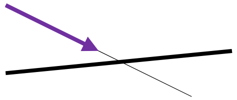
  - Insert lines in order of decreasing slope (cw)
  - When two edges meet, keep the edge with higher slope and trim the inserted edge (with lower slope)
  - To get one tree and not the forest of trees (if not connected) add a vertical line in +∞ (slope +90°)
  - Left endpoints of the edges in the cut do not belong to the tree

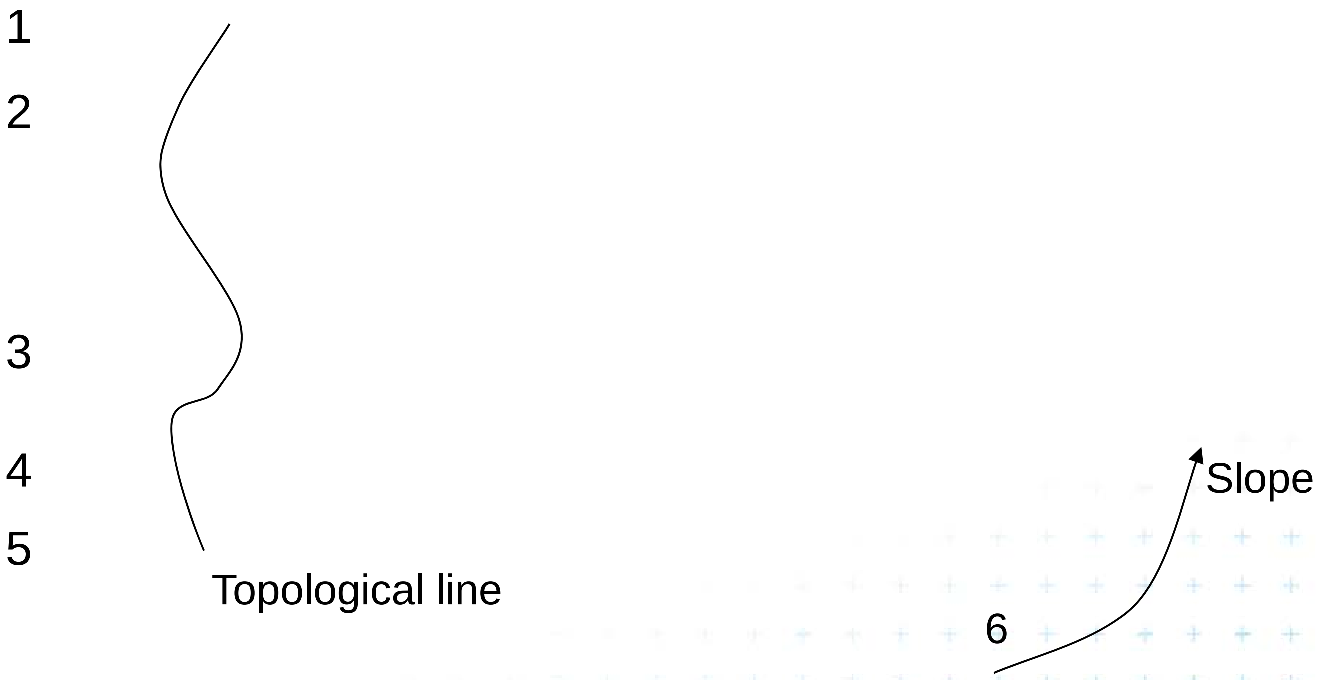


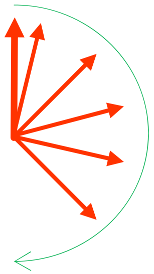
- Lower horizon tree (LHT) construction is symmetrical
- UHT and LHT serve for right endpts determination

Upper horizon tree (UHT) – initial tree

Insert lines in order of decreasing slope (“cw”)



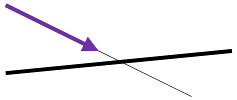


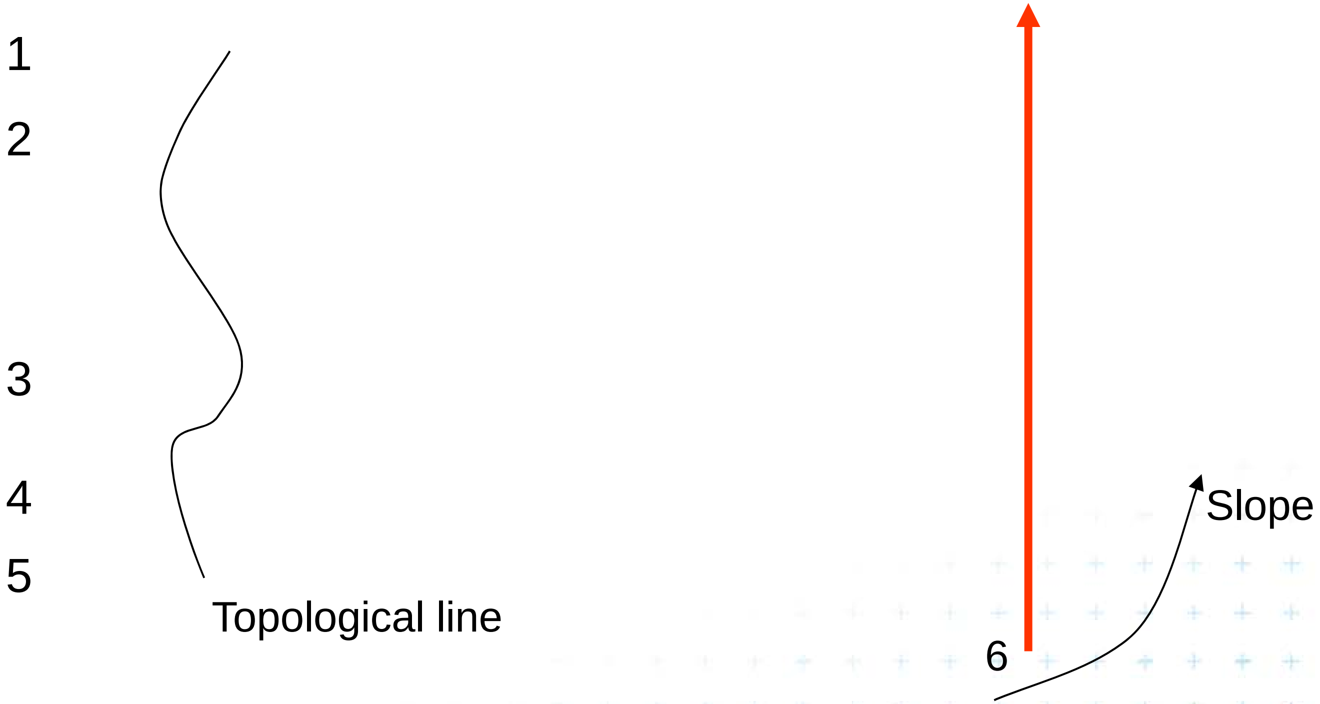


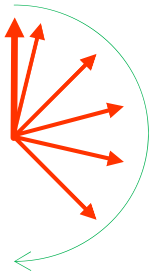
Insertion order: 6, 5, 4, 3, 2, 1

Upper horizon tree (UHT) – initial tree

Insert lines in order of decreasing slope (“cw”)



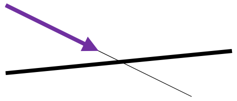


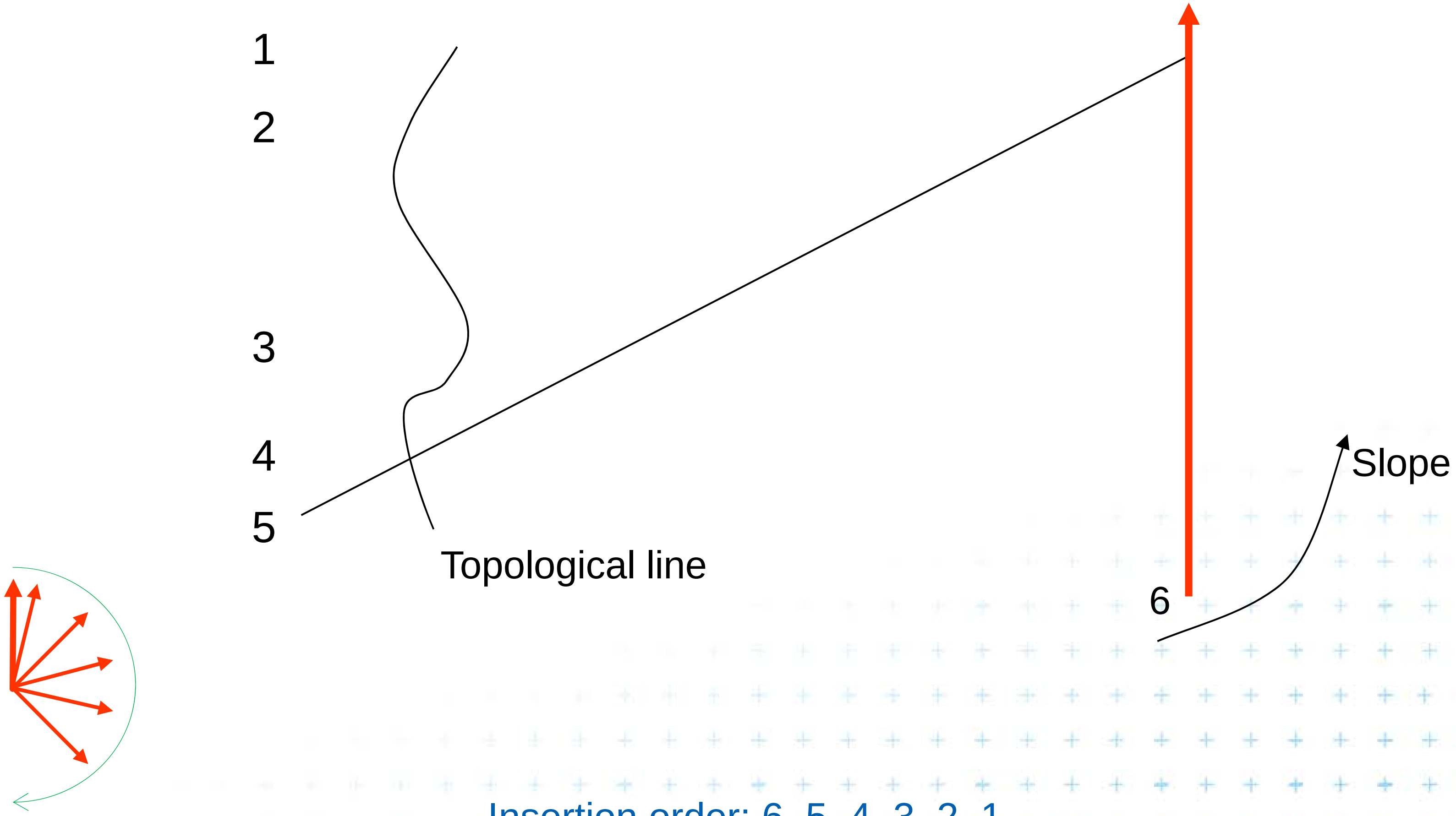


Insertion order: 6, 5, 4, 3, 2, 1

Upper horizon tree (UHT) – initial tree

Insert lines in order of decreasing slope (“cw”)

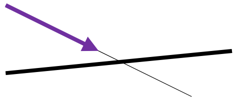


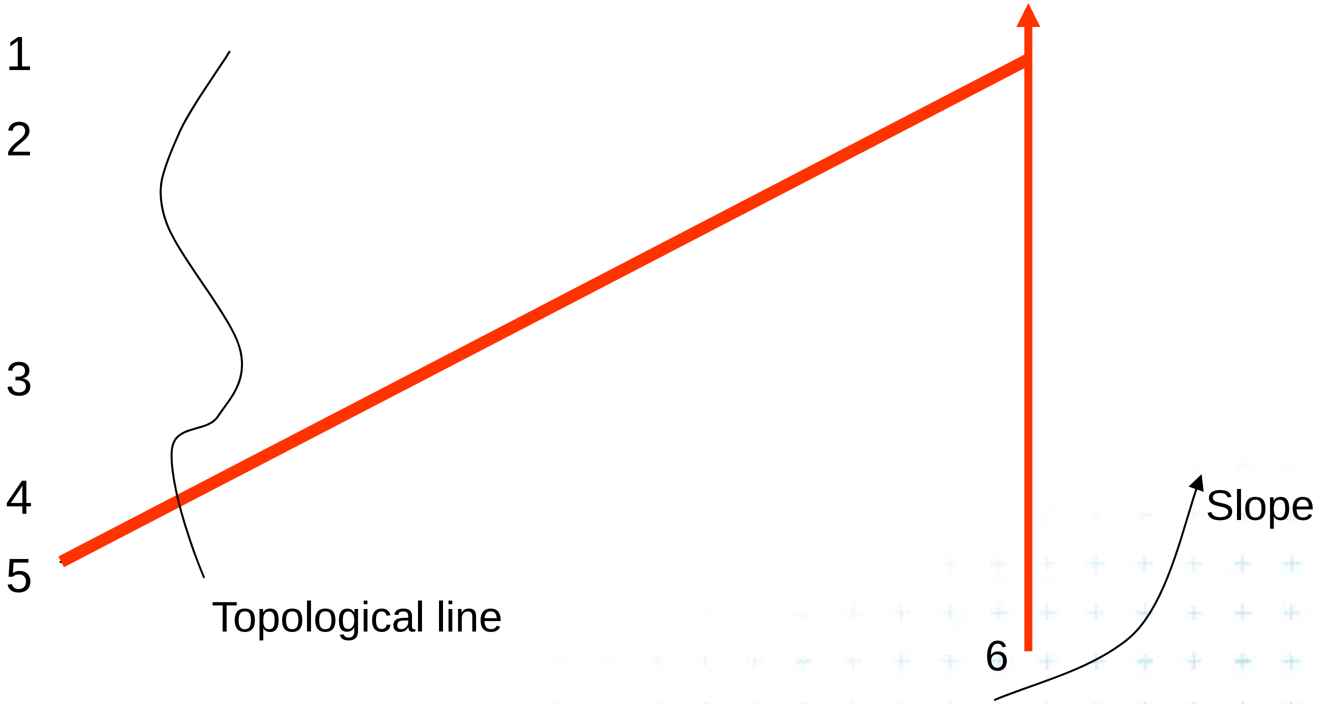


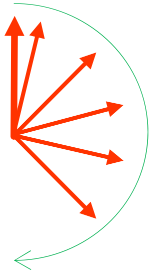
Insertion order: 6, 5, 4, 3, 2, 1

Upper horizon tree (UHT) – initial tree

Insert lines in order of decreasing slope (“cw”)



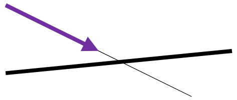


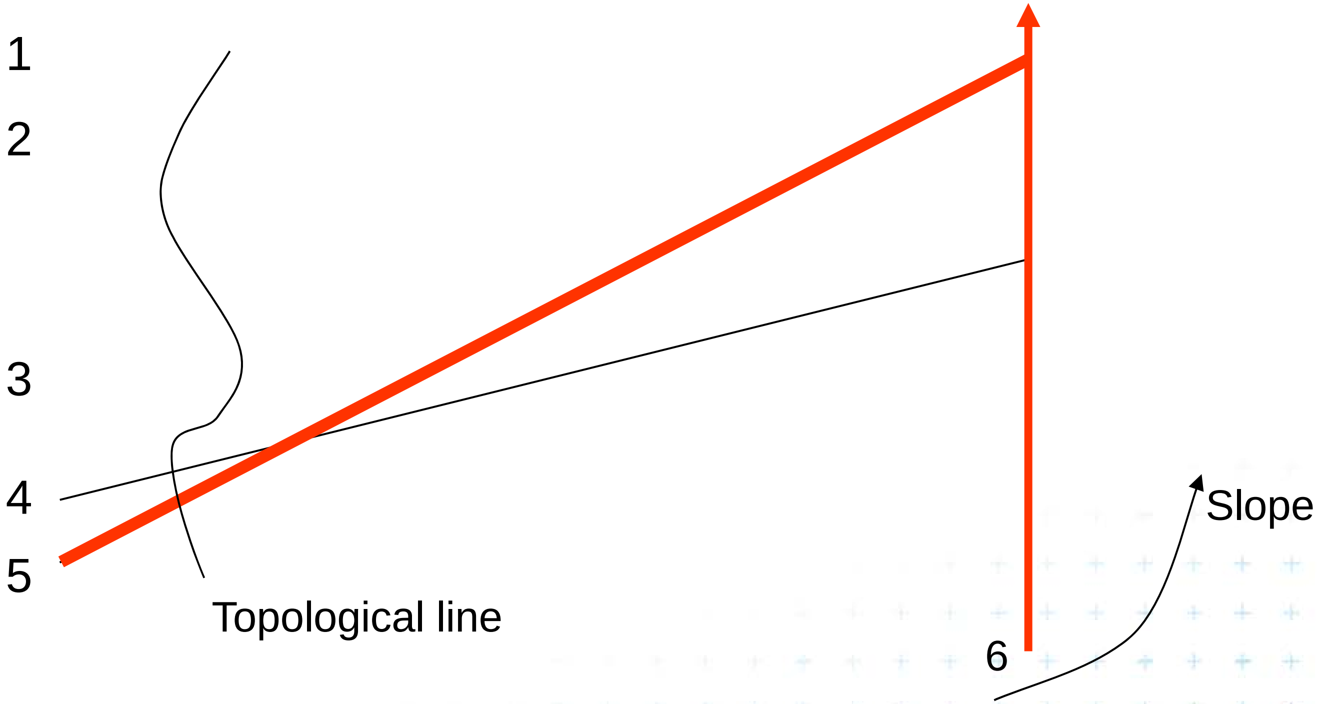


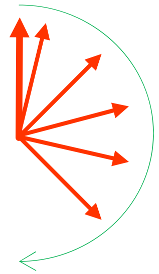
Insertion order: 6, 5, 4, 3, 2, 1

Upper horizon tree (UHT) – initial tree

Insert lines in order of decreasing slope (“cw”)



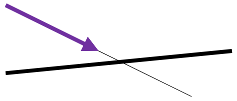


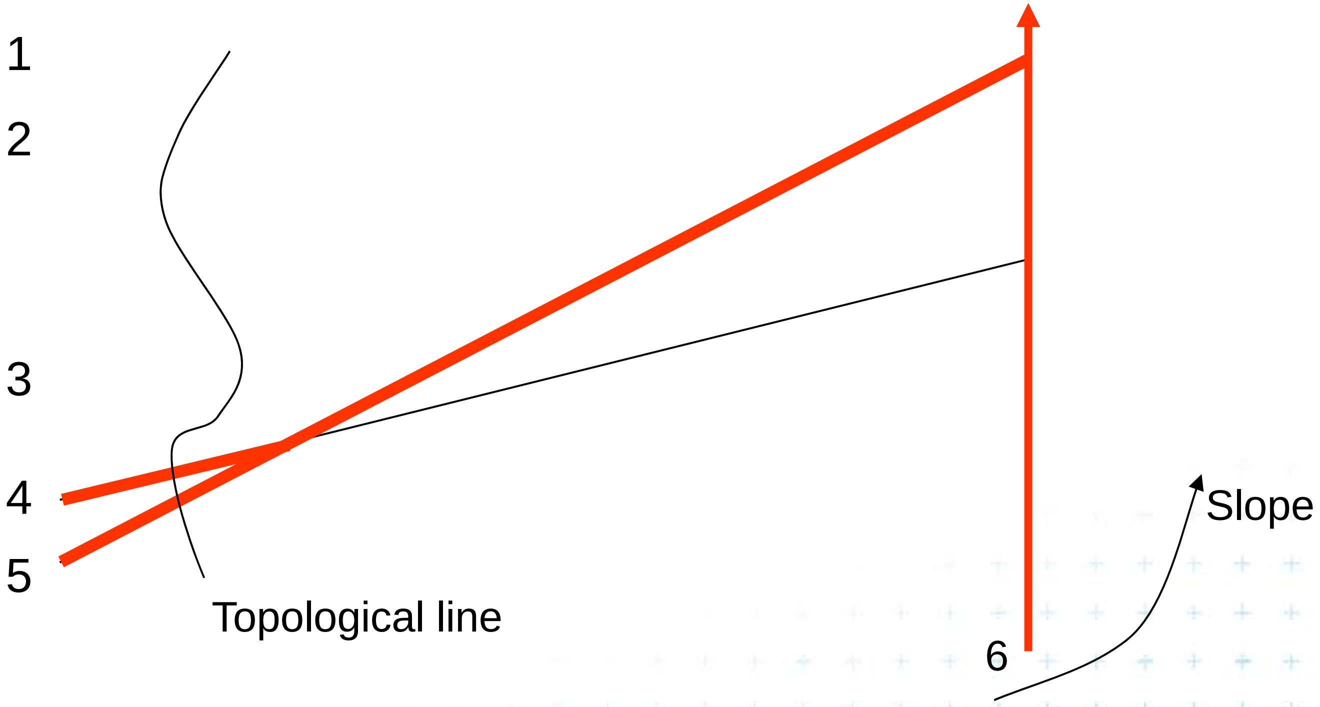


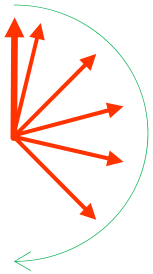
Insertion order: 6, 5, 4, 3, 2, 1

Upper horizon tree (UHT) – initial tree

Insert lines in order of decreasing slope (“cw”)



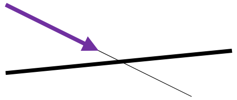


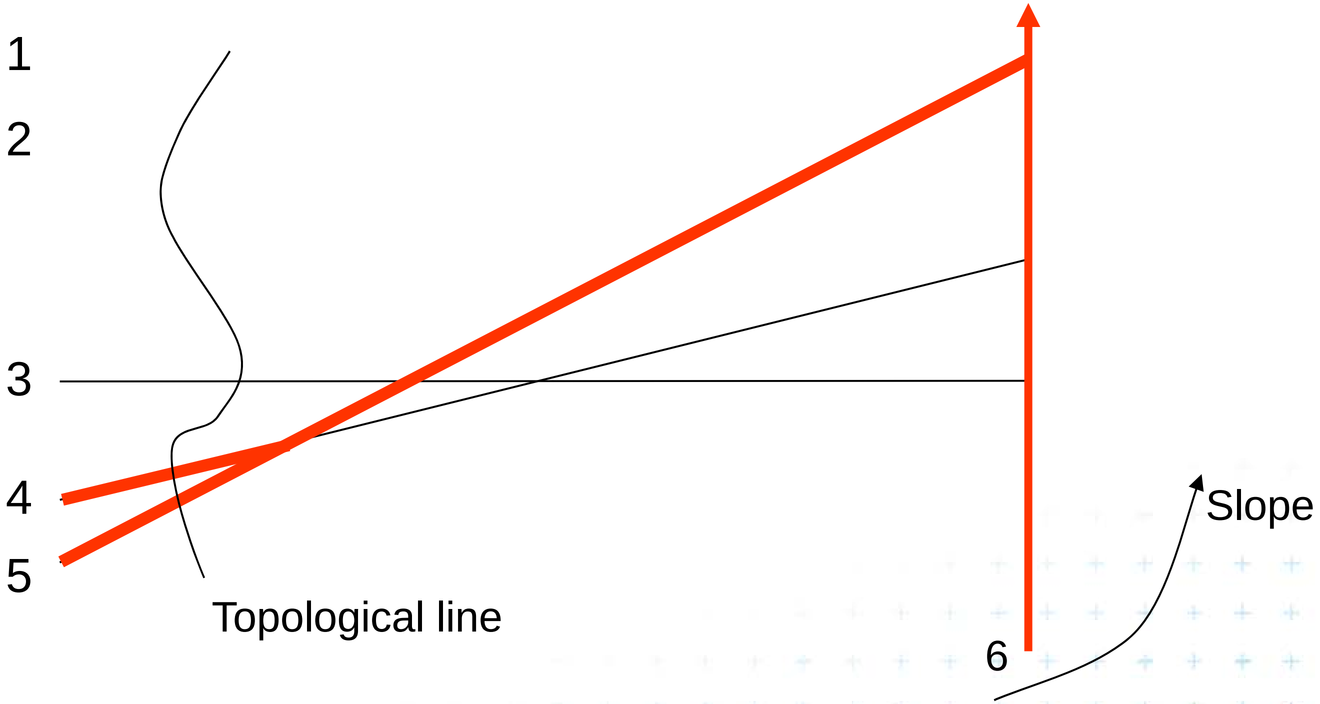


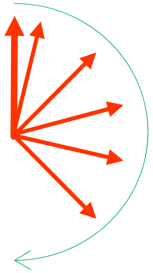
Insertion order: 6, 5, 4, 3, 2, 1

Upper horizon tree (UHT) – initial tree

Insert lines in order of decreasing slope (“cw”)



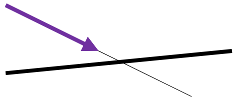


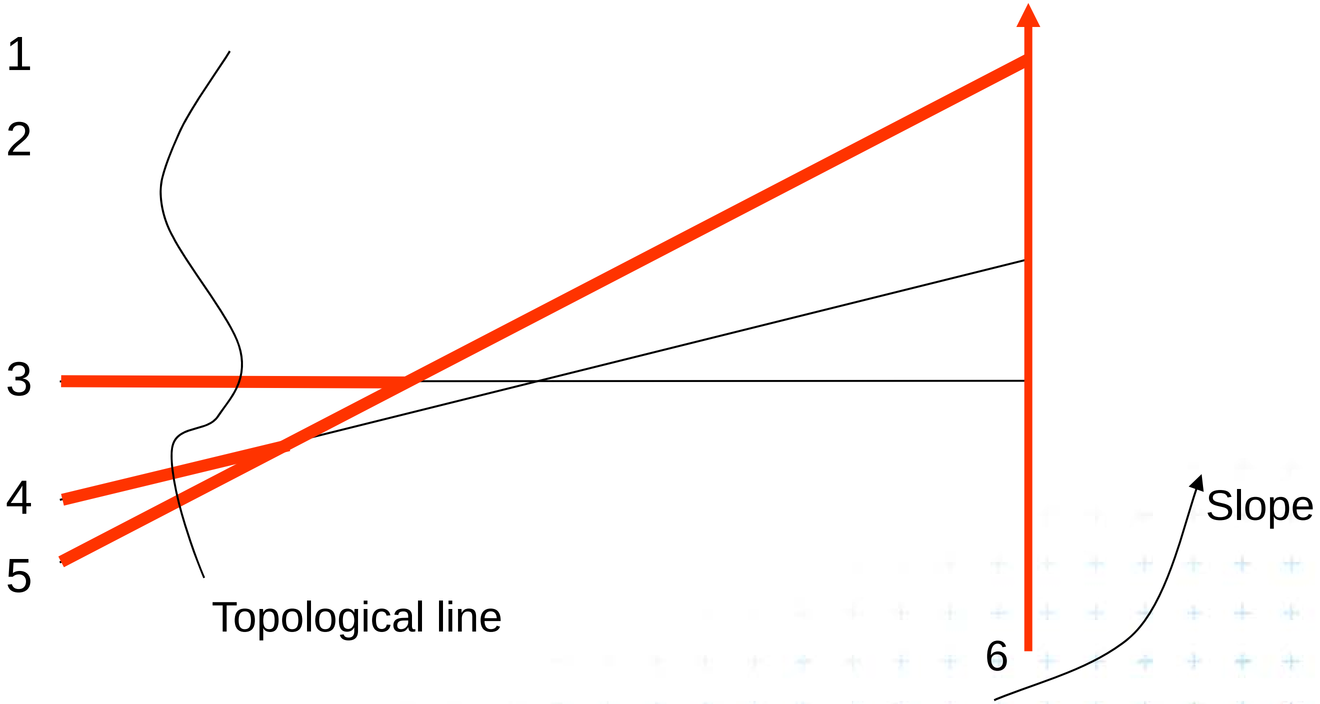


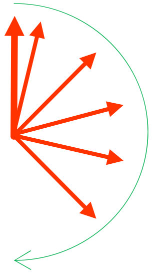
Insertion order: 6, 5, 4, 3, 2, 1

Upper horizon tree (UHT) – initial tree

Insert lines in order of decreasing slope (“cw”)



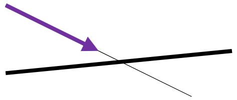


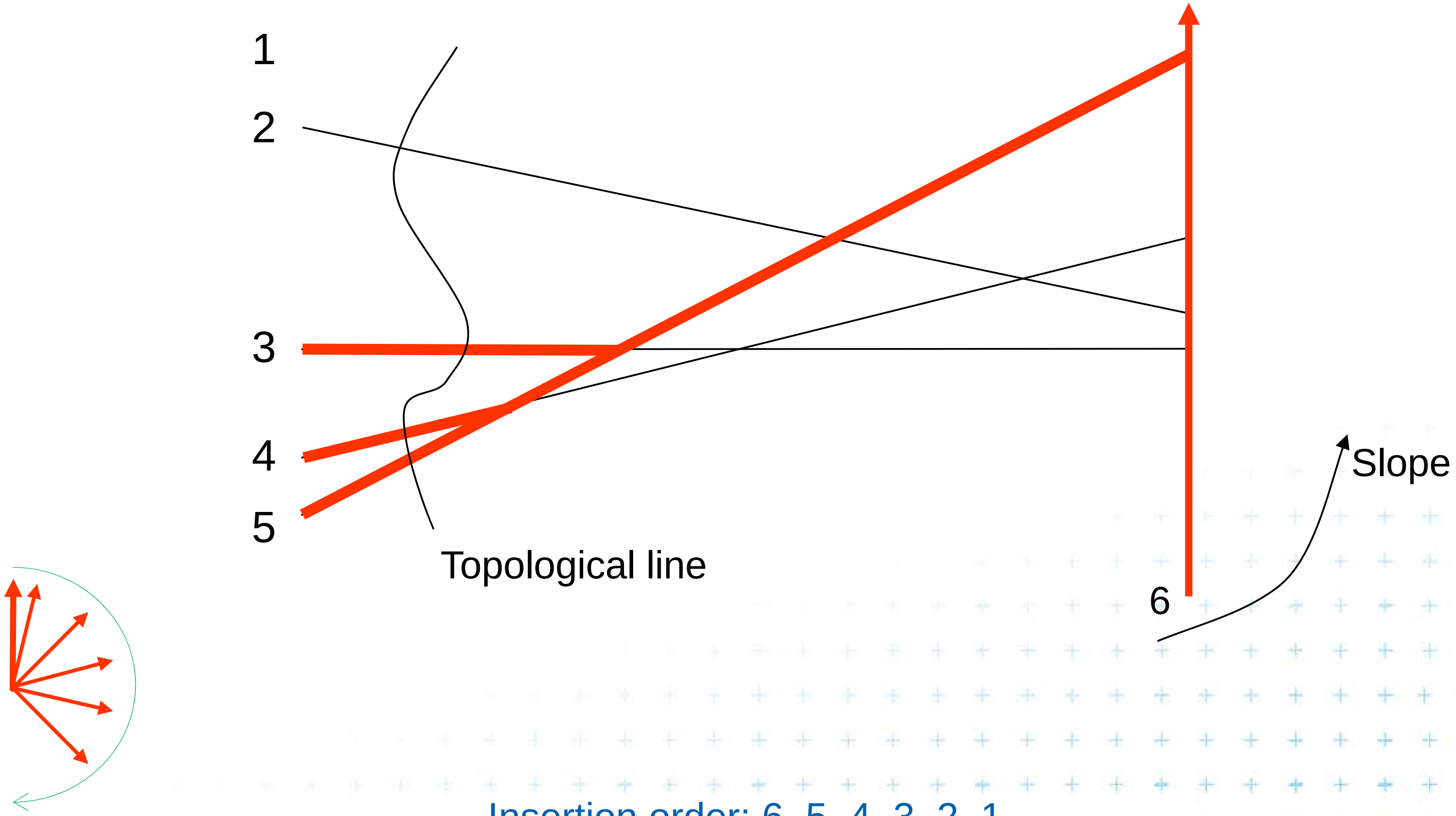


Insertion order: 6, 5, 4, 3, 2, 1

Upper horizon tree (UHT) – initial tree

Insert lines in order of decreasing slope ("cw")

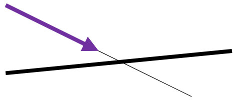


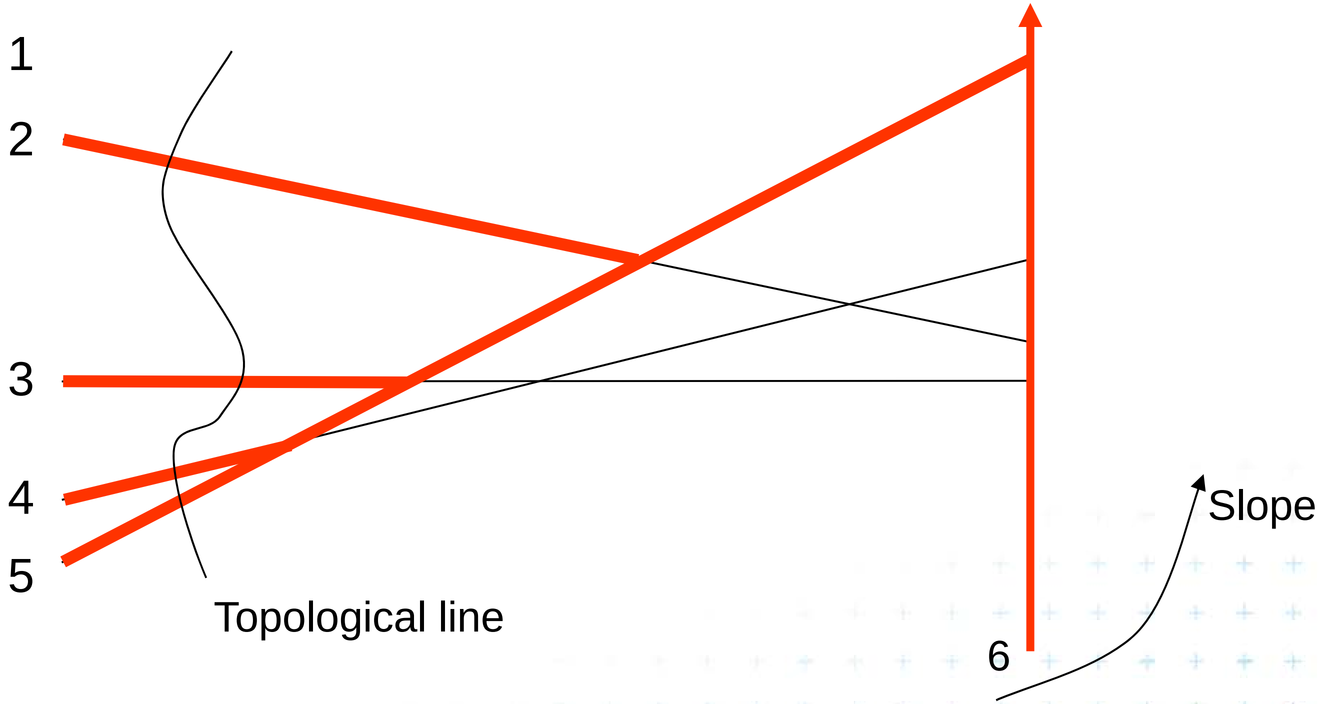


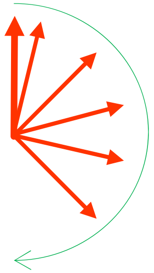
Insertion order: 6, 5, 4, 3, 2, 1

Upper horizon tree (UHT) – initial tree

Insert lines in order of decreasing slope (“cw”)



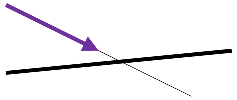


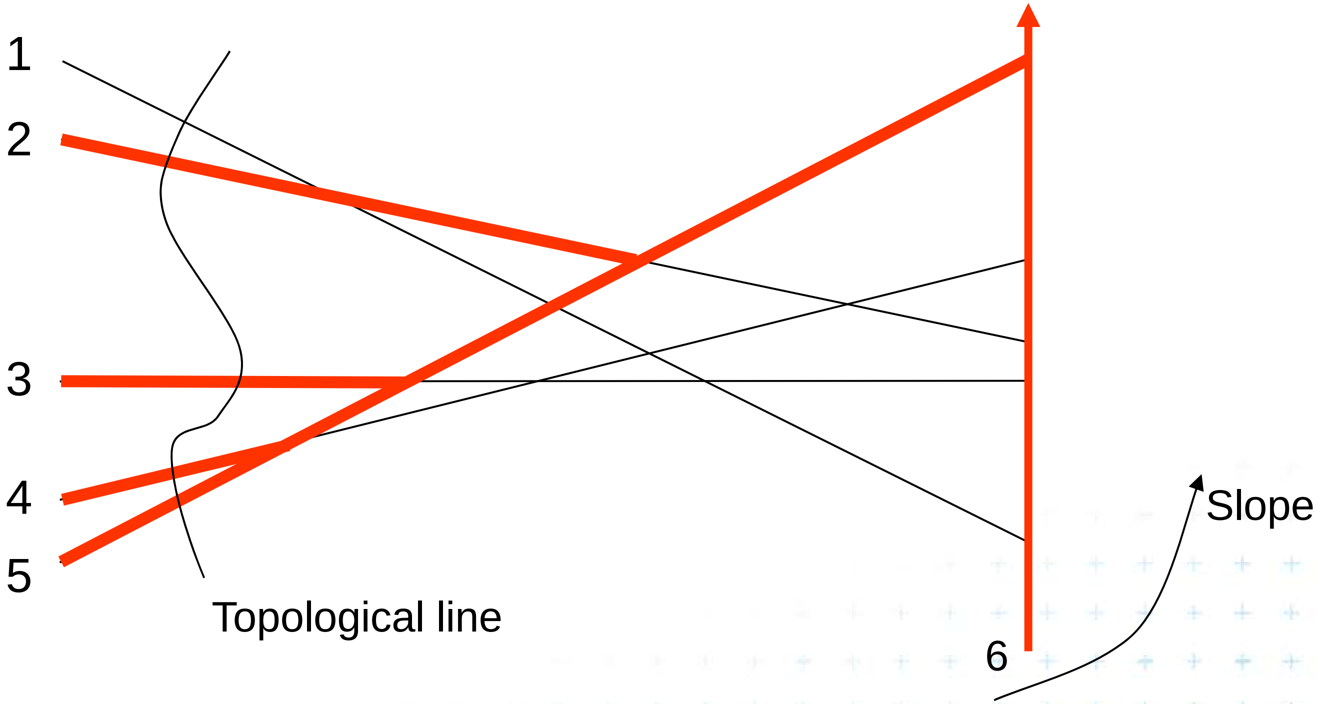


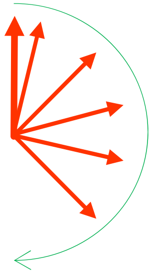
Insertion order: 6, 5, 4, 3, 2, 1

Upper horizon tree (UHT) – initial tree

Insert lines in order of decreasing slope ("cw")



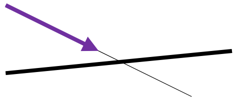


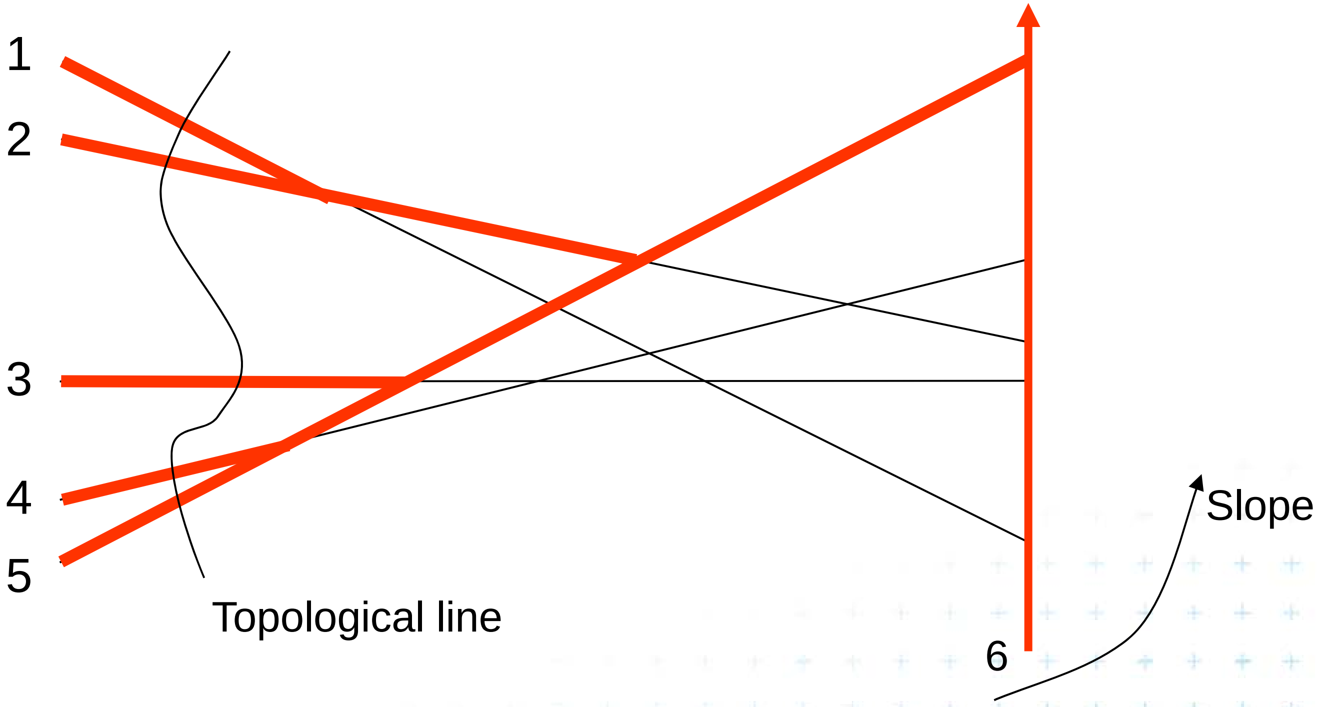


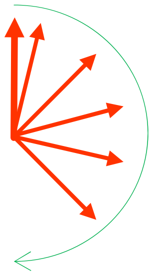
Insertion order: 6, 5, 4, 3, 2, 1

Upper horizon tree (UHT) – initial tree

Insert lines in order of decreasing slope ("cw")



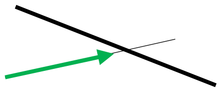


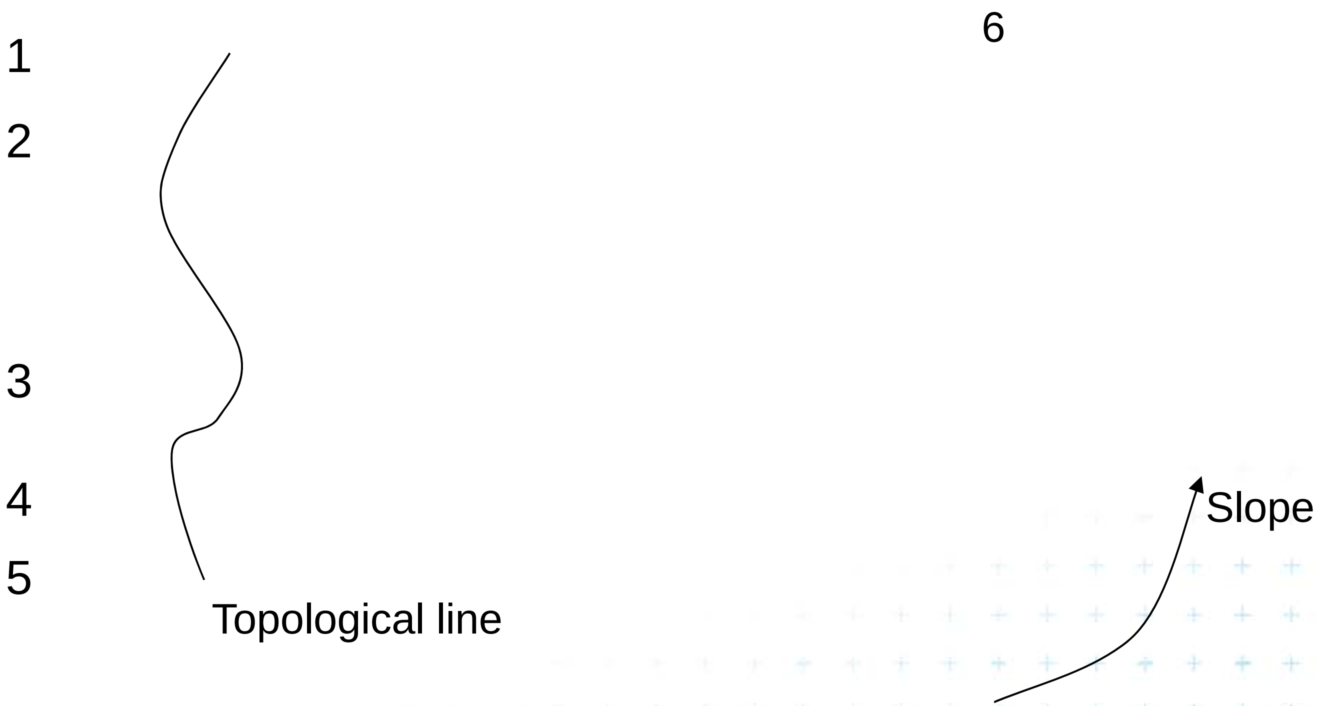


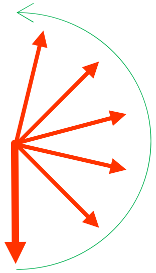
Insertion order: 6, 5, 4, 3, 2, 1

Lower horizon tree (LHT) – initial tree

Insert lines in order of increasing slope (“ccw”)



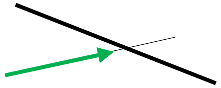


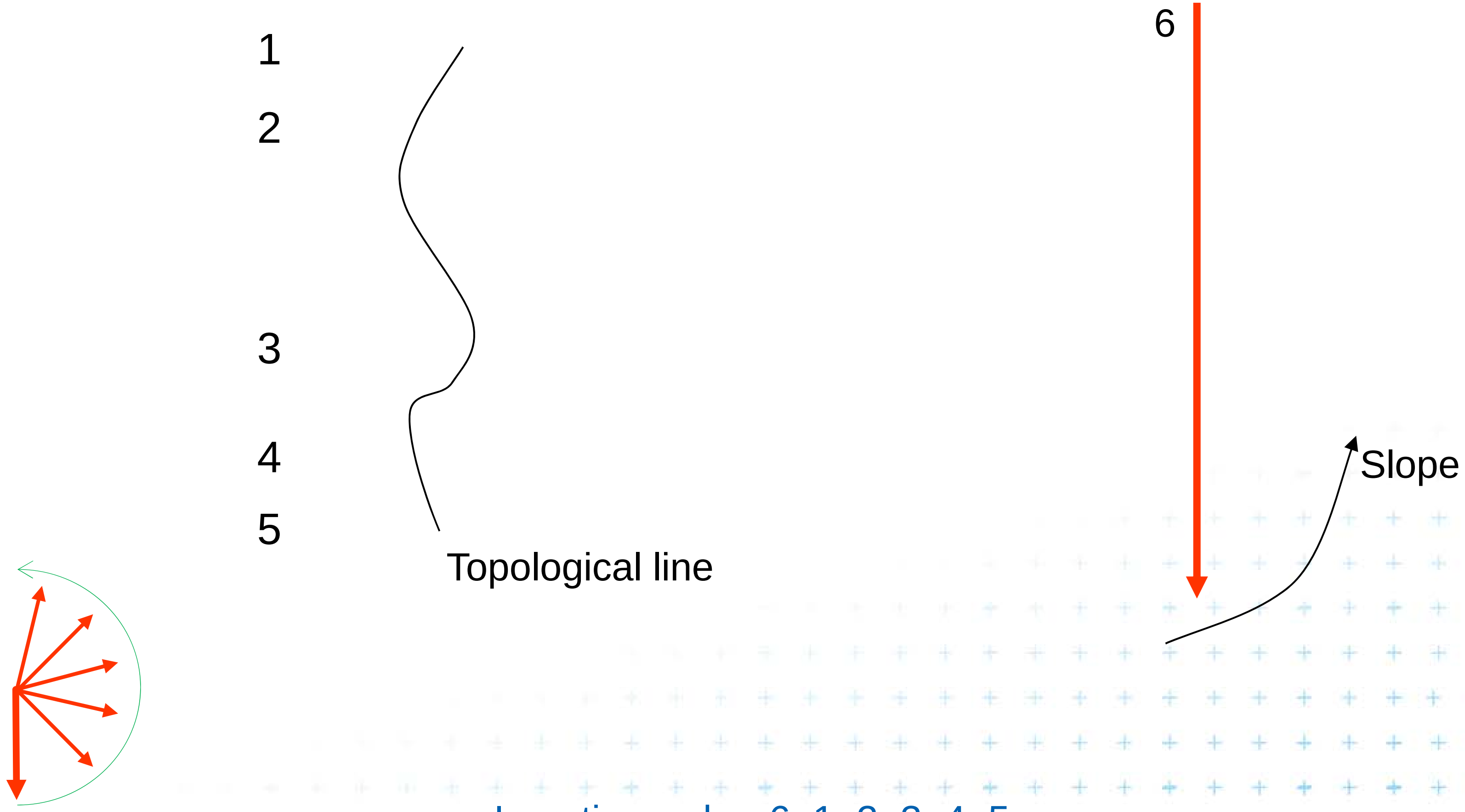


Insertion order: 6, 1, 2, 3, 4, 5

Lower horizon tree (LHT) – initial tree

Insert lines in order of increasing slope (“ccw”)

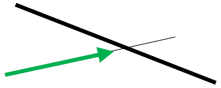


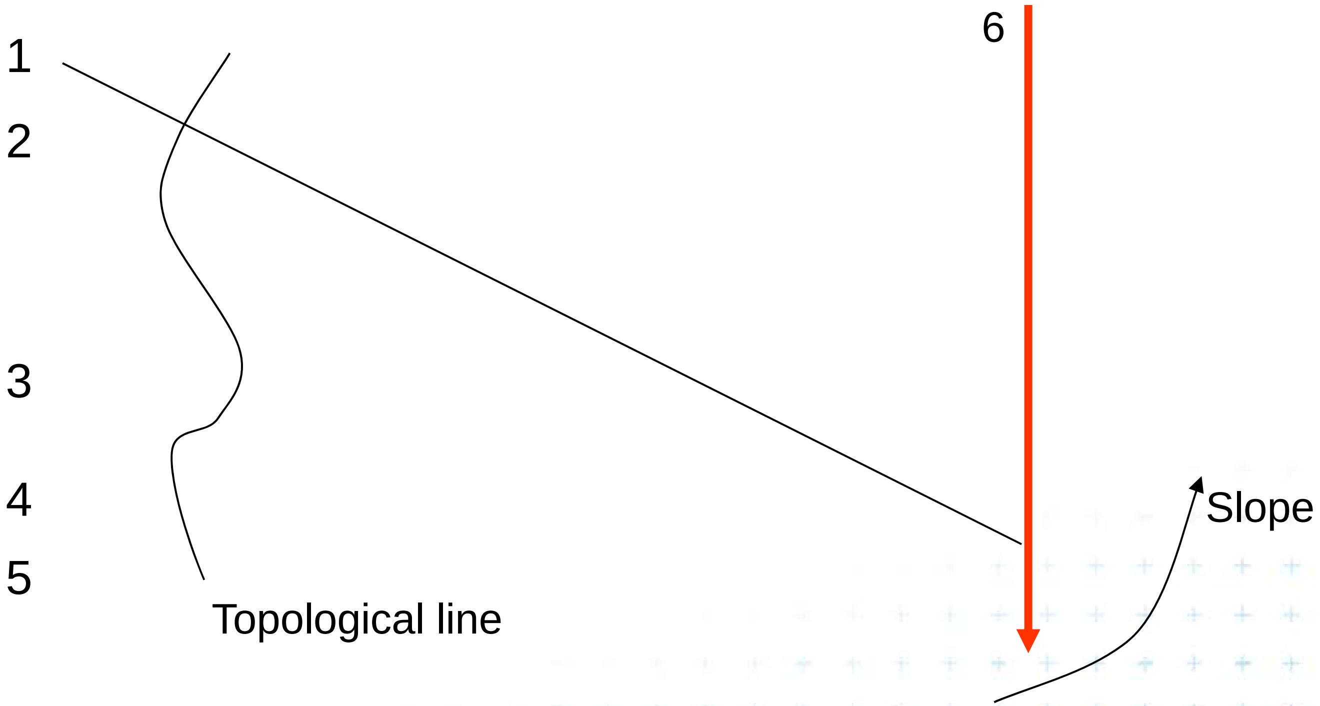


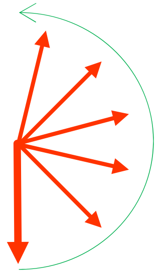
Insertion order: 6, 1, 2, 3, 4, 5

Lower horizon tree (LHT) – initial tree

Insert lines in order of increasing slope (“ccw”)



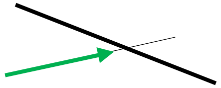


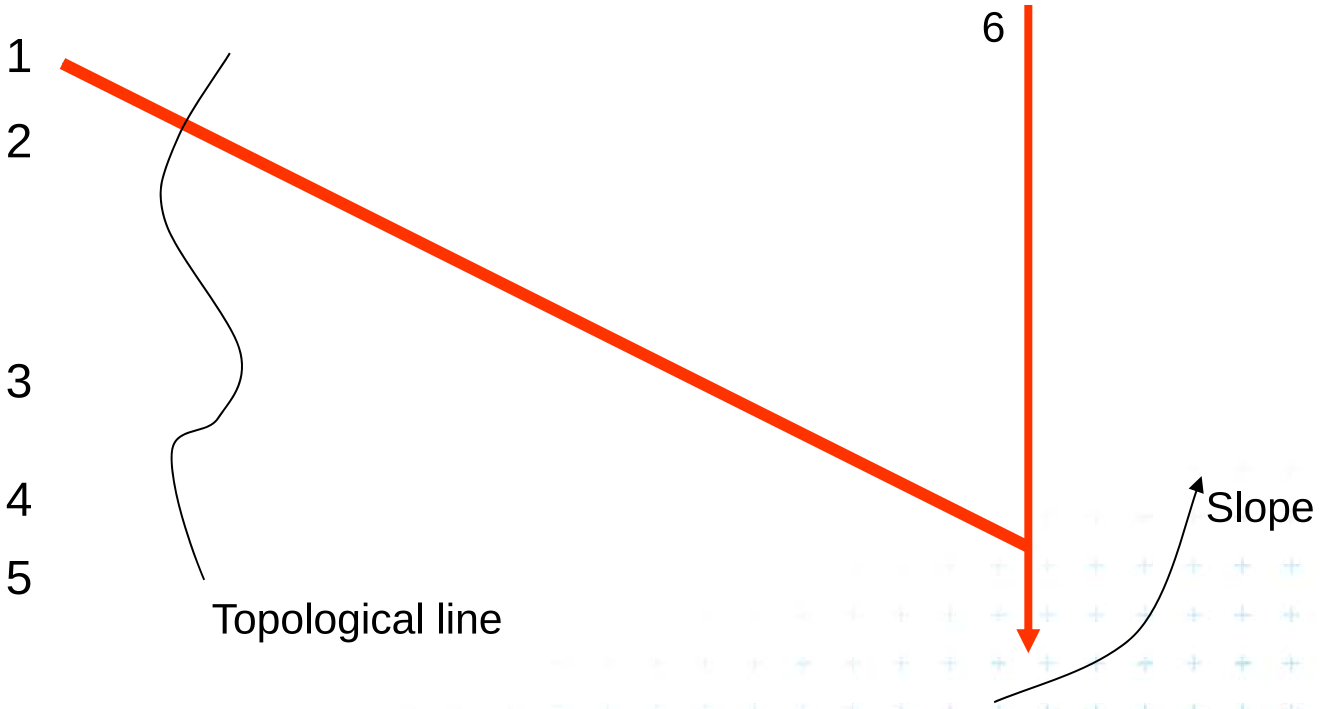


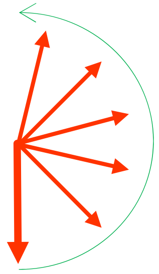
Insertion order: 6, 1, 2, 3, 4, 5

Lower horizon tree (LHT) – initial tree

Insert lines in order of increasing slope (“ccw”)



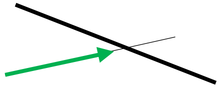


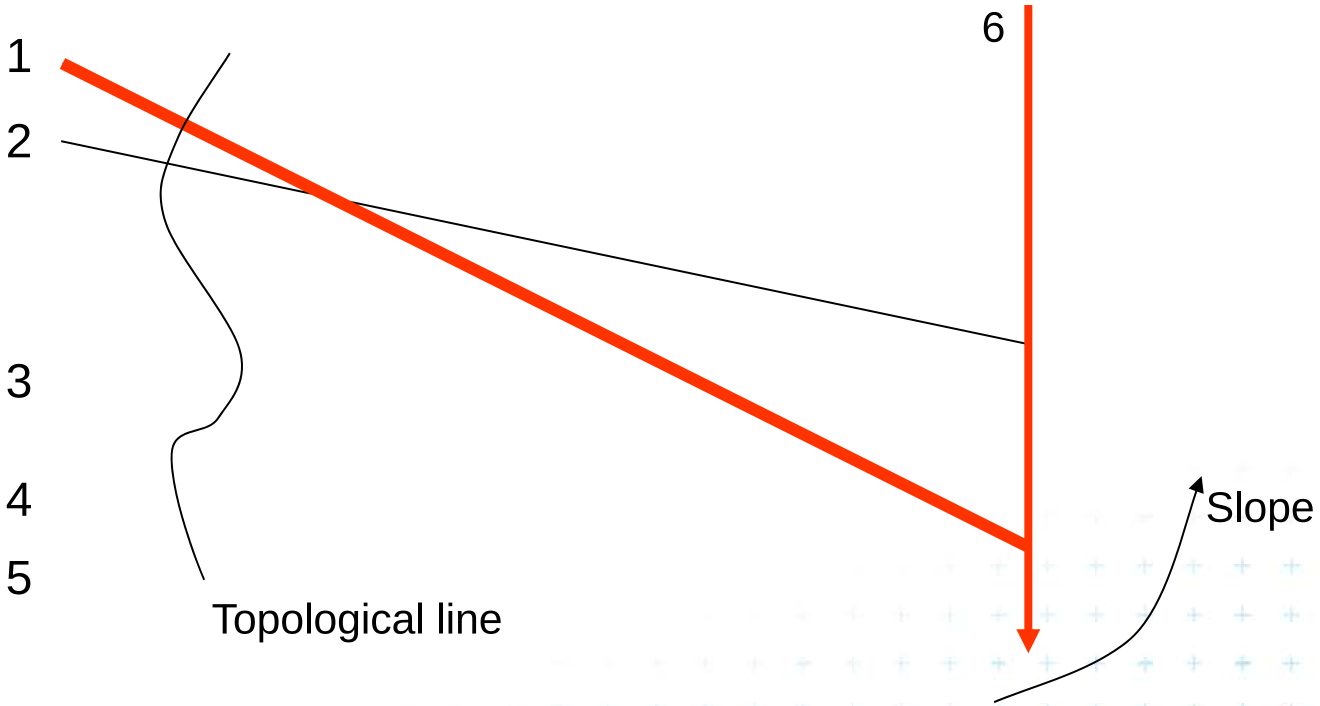


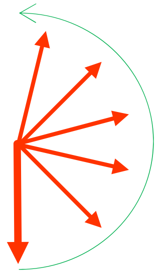
Insertion order: 6, 1, 2, 3, 4, 5

Lower horizon tree (LHT) – initial tree

Insert lines in order of increasing slope (“ccw”)



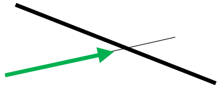


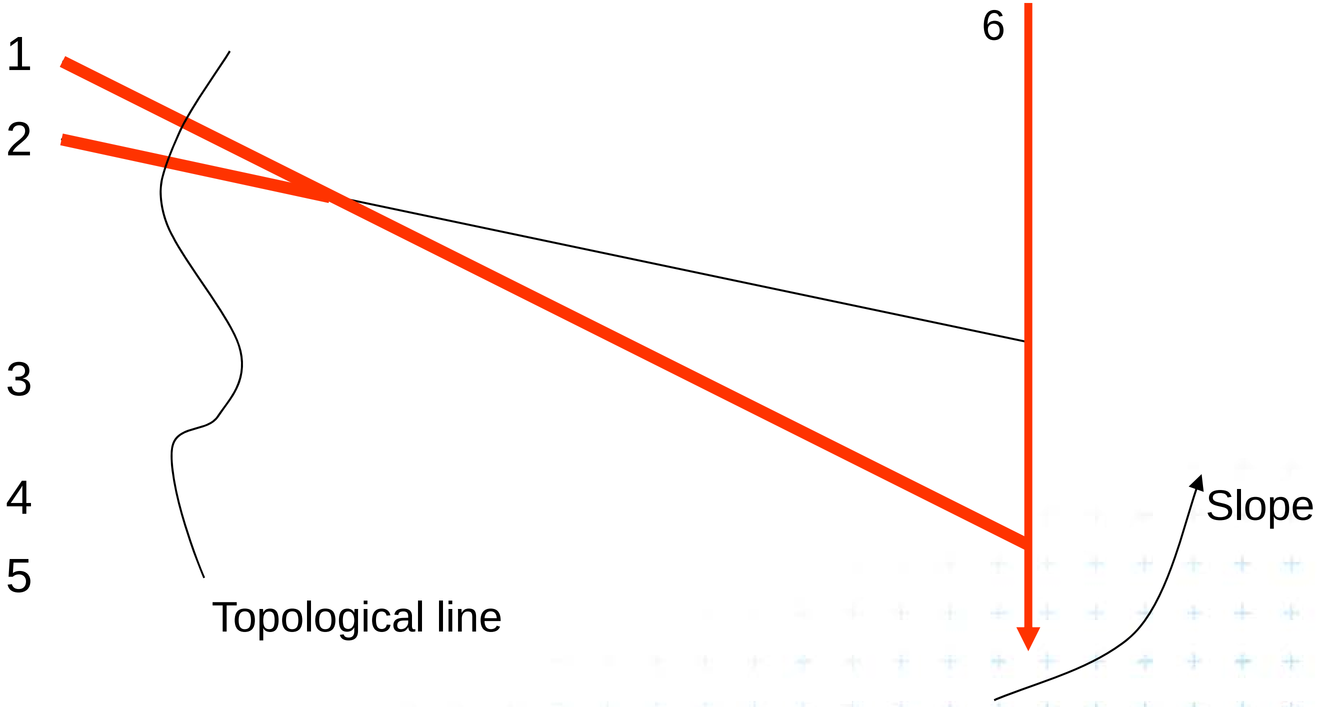


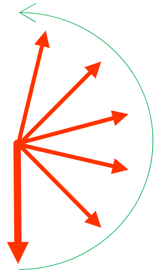
Insertion order: 6, 1, 2, 3, 4, 5

Lower horizon tree (LHT) – initial tree

Insert lines in order of increasing slope (“ccw”)



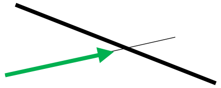


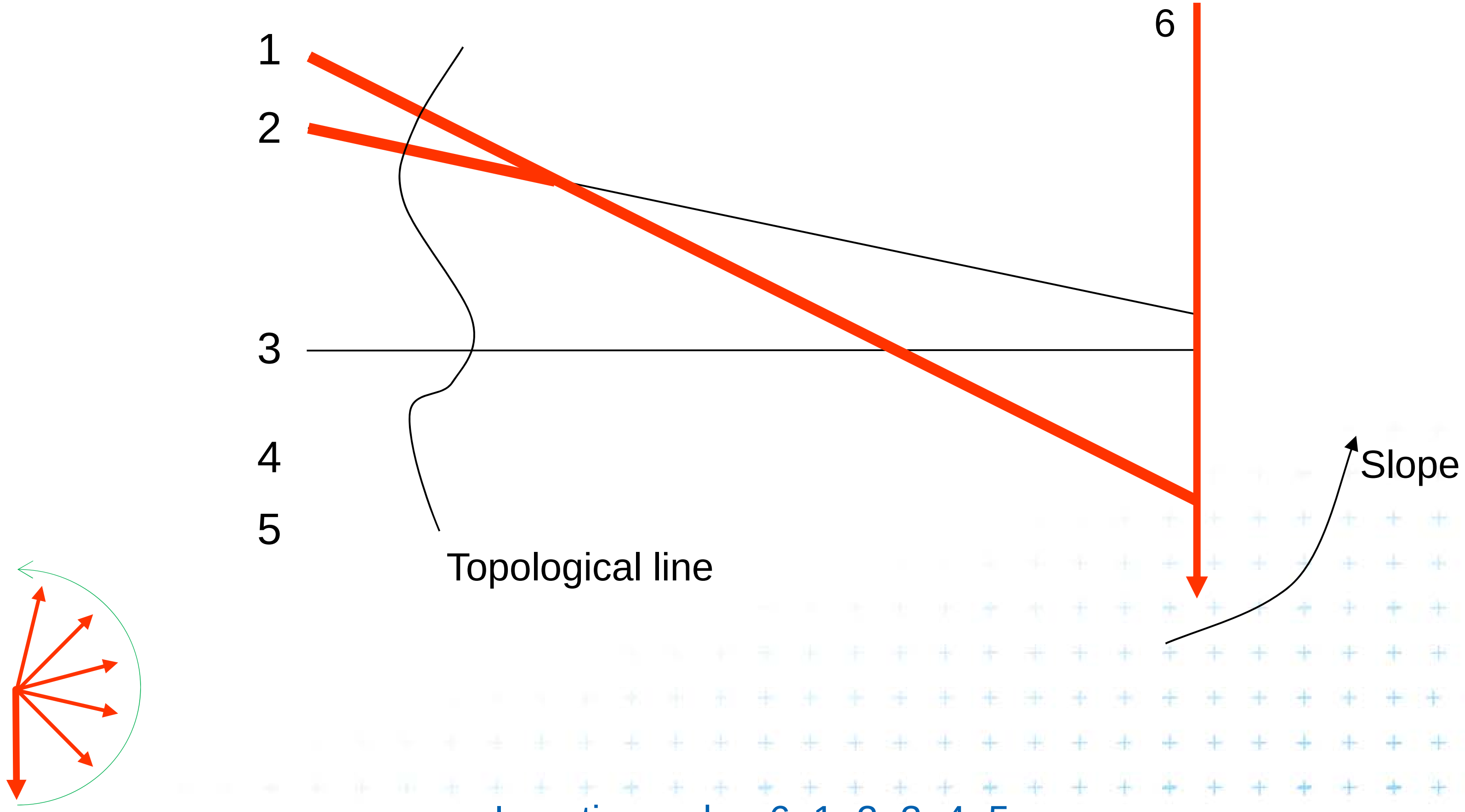


Insertion order: 6, 1, 2, 3, 4, 5

Lower horizon tree (LHT) – initial tree

Insert lines in order of increasing slope (“ccw”)

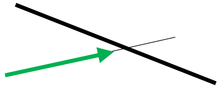


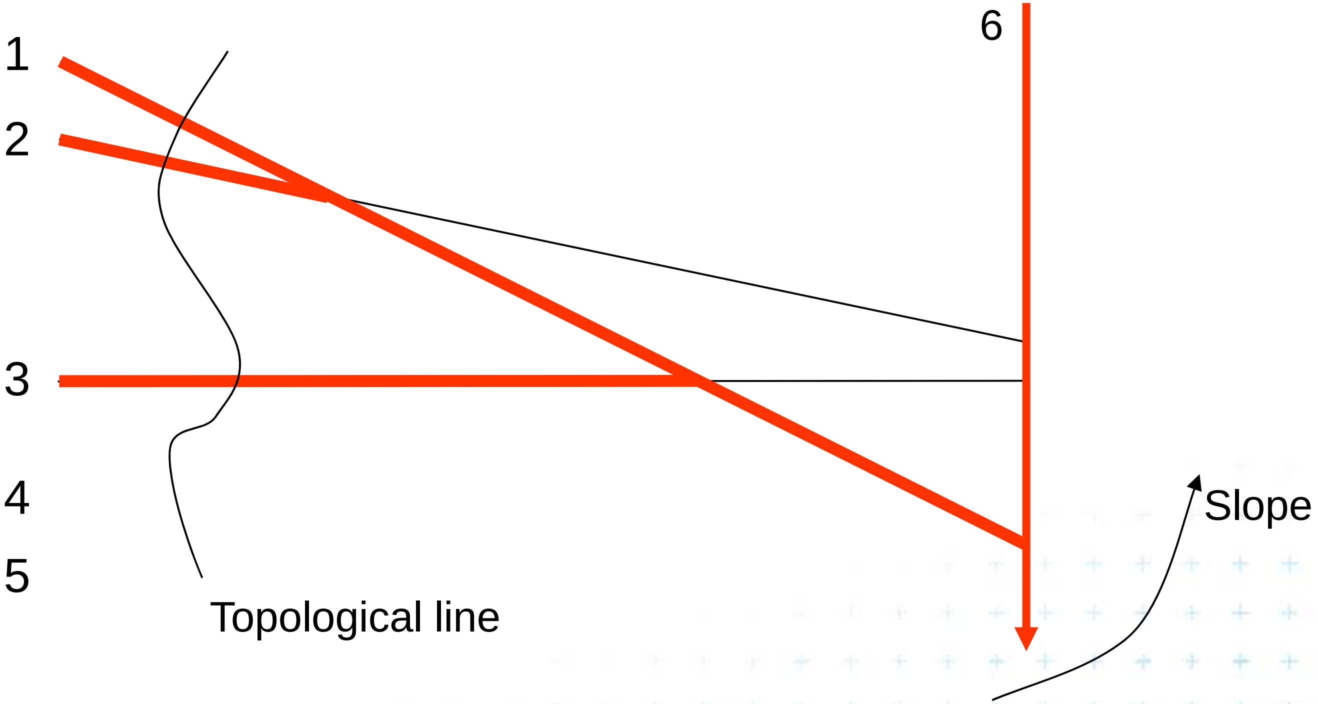


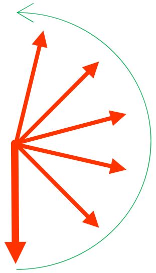
Insertion order: 6, 1, 2, 3, 4, 5

Lower horizon tree (LHT) – initial tree

Insert lines in order of increasing slope (“ccw”)



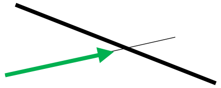


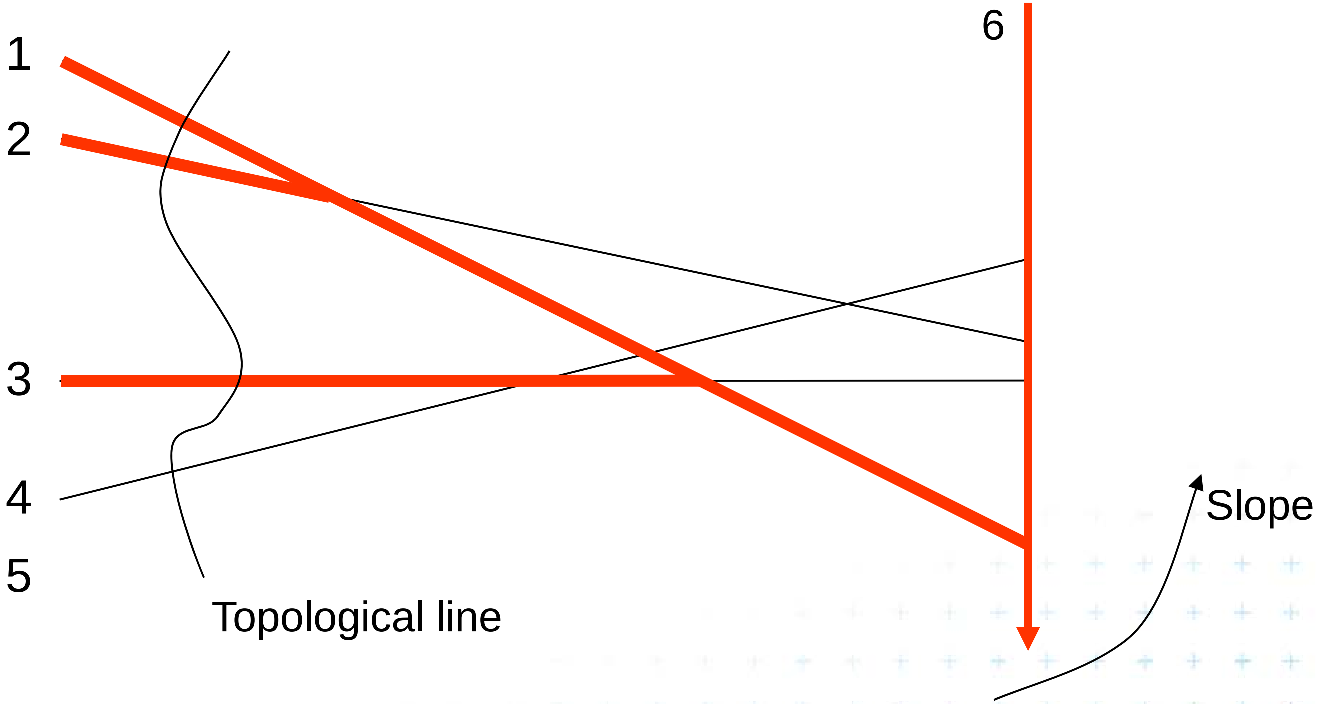


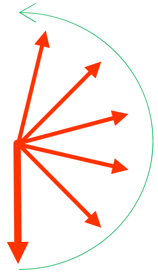
Insertion order: 6, 1, 2, 3, 4, 5

Lower horizon tree (LHT) – initial tree

Insert lines in order of increasing slope (“ccw”)



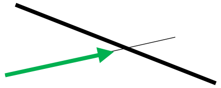


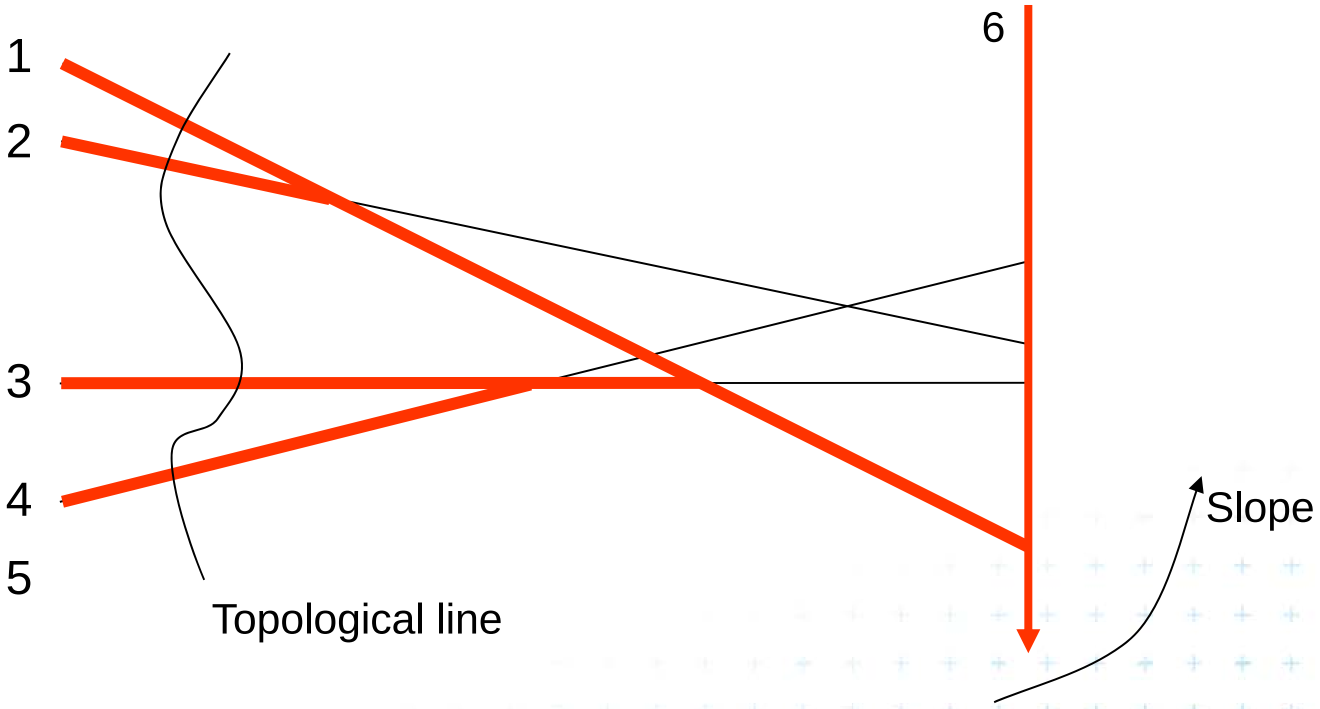


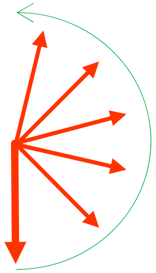
Insertion order: 6, 1, 2, 3, 4, 5

Lower horizon tree (LHT) – initial tree

Insert lines in order of increasing slope (“ccw”)



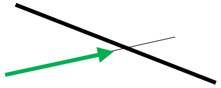


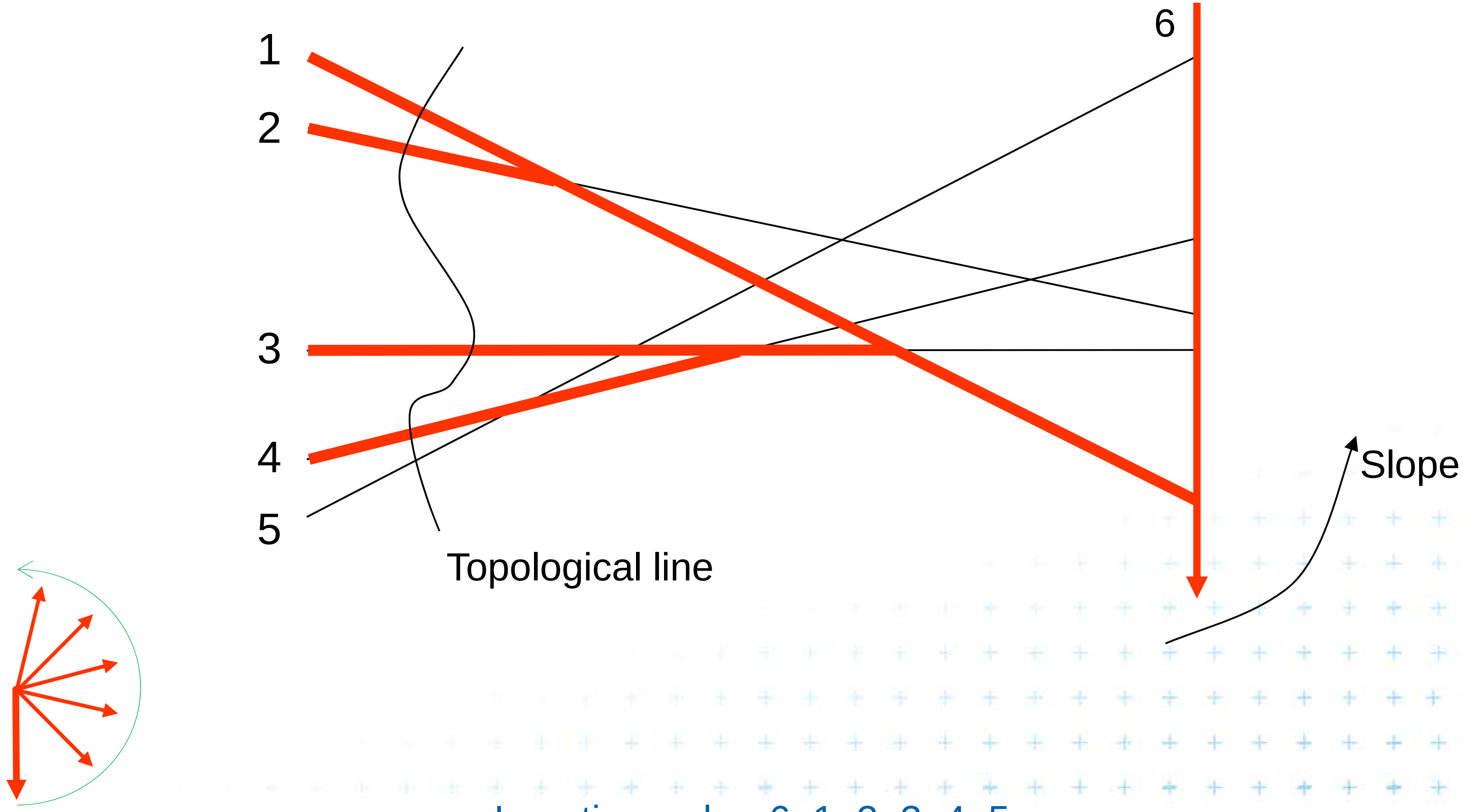


Insertion order: 6, 1, 2, 3, 4, 5

Lower horizon tree (LHT) – initial tree

Insert lines in order of increasing slope ("ccw")

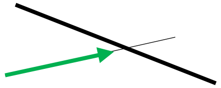


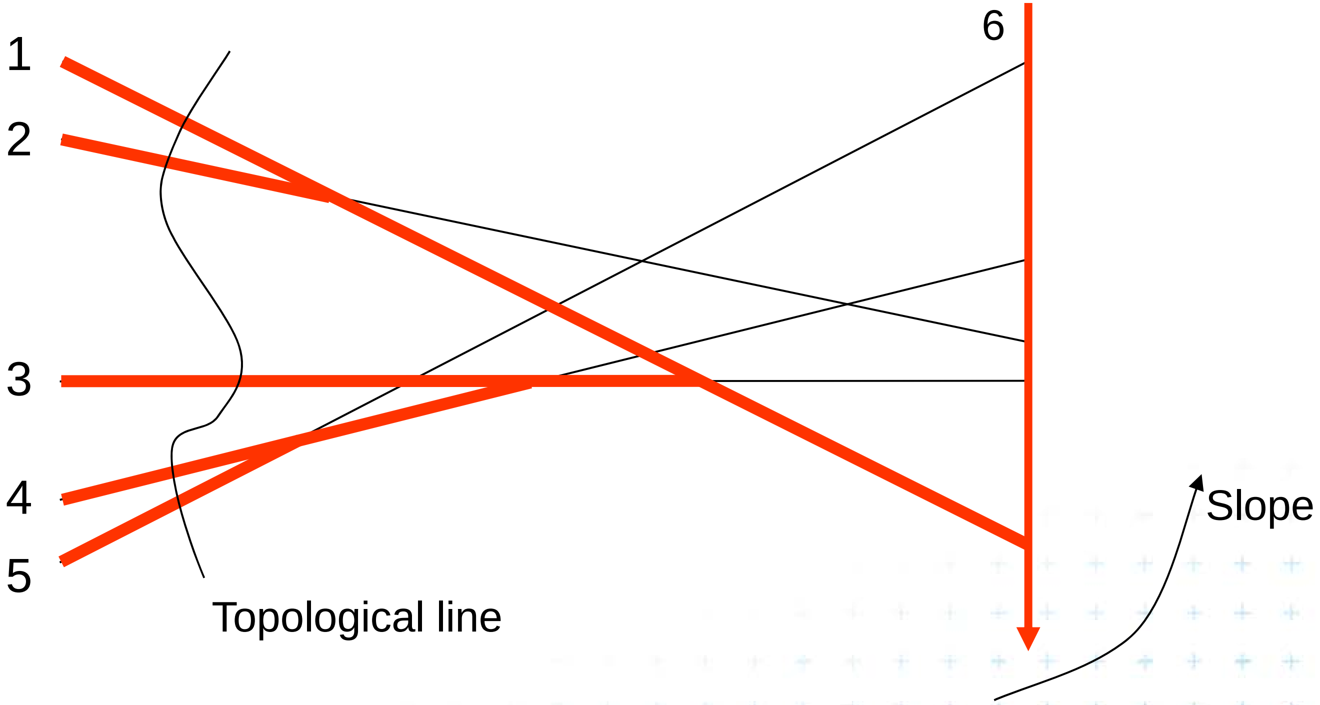


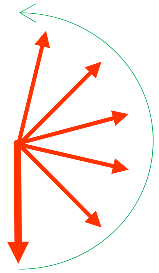
Insertion order: 6, 1, 2, 3, 4, 5

Lower horizon tree (LHT) – initial tree

Insert lines in order of increasing slope (“ccw”)

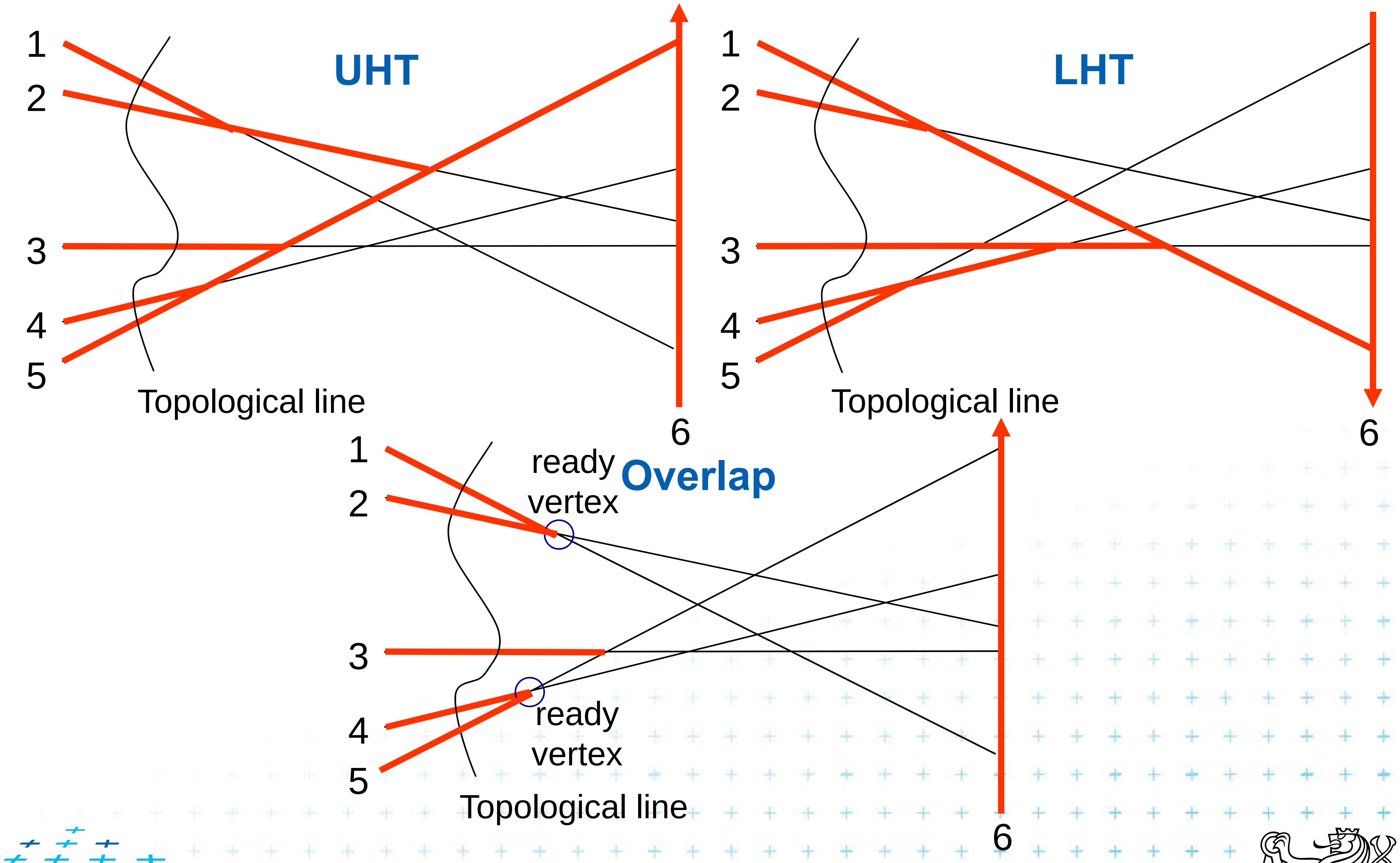






Insertion order: 6, 1, 2, 3, 4, 5

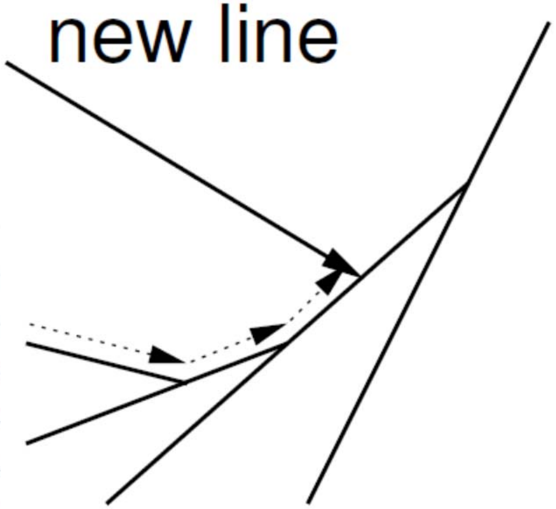
Overlap UHT and LHT – detect ready vertices



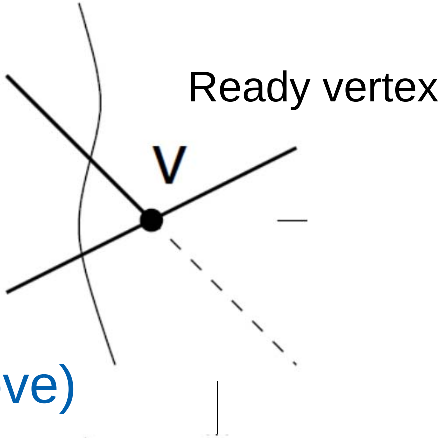
Upper horizon tree (UHT) – init. construction

- Insert lines in order of decreasing slope (cw)
- Each new line starts above all the current lines
- The uppermost face = convex polygonal chain
- Walk left to right along the chain
  to determine the intersection
- Never walk twice over a segment
  - Such segment is no longer part of the upper chain
  - O(n) segments in UHT

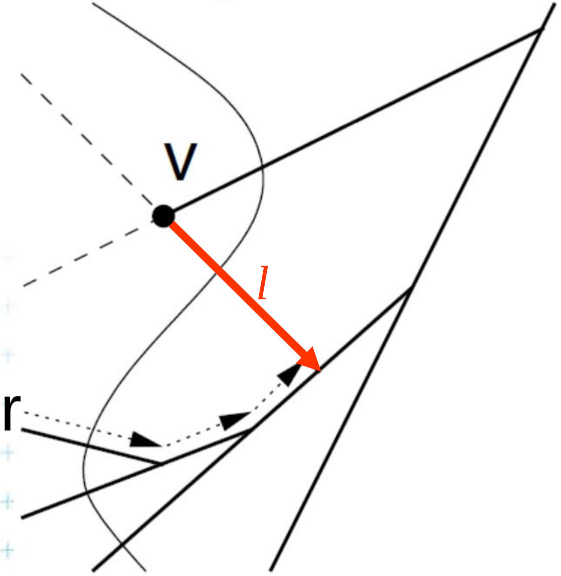
  => O(n) initial construction
  (after n log n sorting of the lines ~slope)



Upper horizon tree (UHT) – update



- After the elementary step
- Two edges swap position along the sweep line
- Lower edge *l* (lower slope, comes from above)
  - Reenter to UHT
  - Terminate at nearest edge of UHT
  - Start in edge below in the current cut
  - Traverse the face in CCW order
  - Intersection must exist, as *l* has lower slope than the other edge from *v* and both belong to the same face



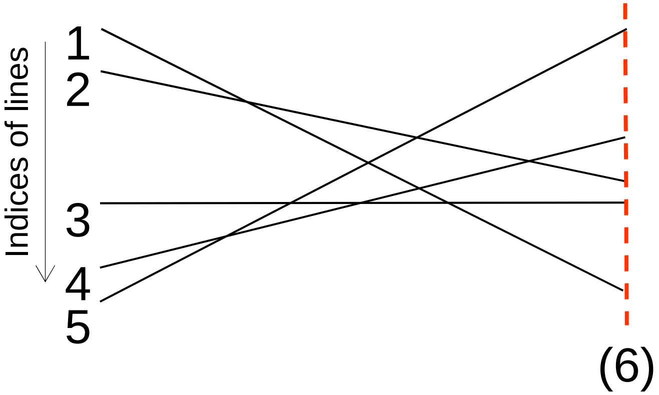
Data structures for topological sweep alg.

Topological sweep line algorithm uses 5 arrays:

1) Line equation coefficients – *E* [1:*n*]
2) Upper horizon tree – UHT [1:*n*]
3) Lower horizon tree – LHT [1:*n*]
4) Order of lines cut by the sweep line – C [1:*n*]
5) Edges along the sweep line – N [1:*n*]
6) Stack for ready vertices (events) – S

(*n* number of lines)

1) Line equation coefficients *E* [1:*n*]



Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

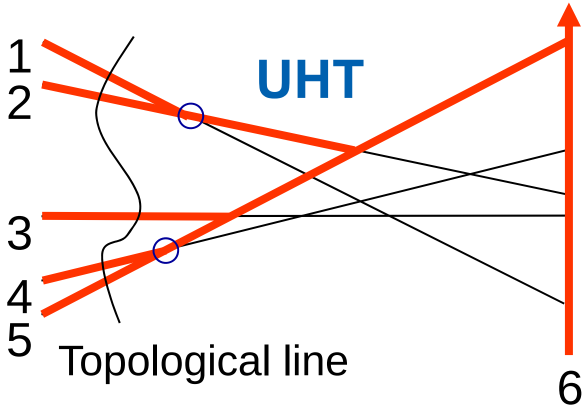
- Array of line equation coefs. *E*
  - Contains coefficients a_i and b_i of line equations $y = a_i x + b_i$
  - *E* is indexed by the line index
  - Lines are ordered according to their slope (angle from -90° to 90°)

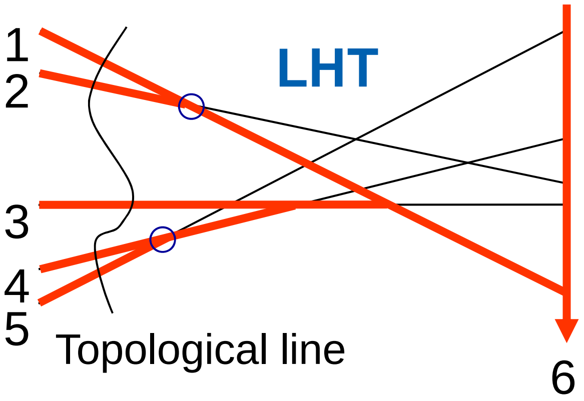
Slope

2) and 3) – Horizon trees UHT and LHT

Their intersection is used for searching of legal steps (right points)

- the shorter edge wins





UHT array

Delimiting lines indices

1	−∞	2
2	−∞	5
3	−∞	5
4	−∞	5
5	−∞	6
6	−∞	+∞

LHT array

Delimiting lines indices

1	−∞	6
2	−∞	1
3	−∞	1
4	−∞	3
5	−∞	4
6	+∞	−∞

- Store pairs of line indices in E that delimit segment l_i to the left and to the right
- Segments are half open ○——●
- Unlimited line has "indices"

  (−∞, +∞] (+∞, −∞]
- One additional vertical line
  - prevents the tree from splitting into forest of trees
  - is inserted first and never trimmed
  - is (−∞, +∞] for UHT
  - is (+∞, −∞] for LHT

4) Order of lines cut by sweep line – C [1:*n*]

- The topological sweep line cuts each line once
- Order of the cuts (along the topological sweep line) is stored in array C as a sequence of line indices
- Array C "points" to the array E of line equations
- For the initial leftmost cut, the order is the same as in E
- Index *ci* addresses *i-th* line from top along the sweep line

CUT Lines C

Indexes of supporting lines

c1	1
c2	2
c3	3
c4	4
c5	5

5) Edges along the sweep line – N [1:*n*]

- Edges intersected by the topological sweep line are stored here (edges along the sweep line)
- Instead of endpoints themselves, we store the indices of lines whose intersections delimit the edge
- Order of these edges is the same as in C (both use the index *ci*)
- Index *ci* stores the index of *i-th* edge from top along the sweep line

CUT edges N

Pairs of line indices delimiting the edge

c1	$-\infty$	**2**
c2	$-\infty$	**1**
c3	$-\infty$	**5**
c4	$-\infty$	**5**
c5	$-\infty$	**4**

The first edge along the sweep line:

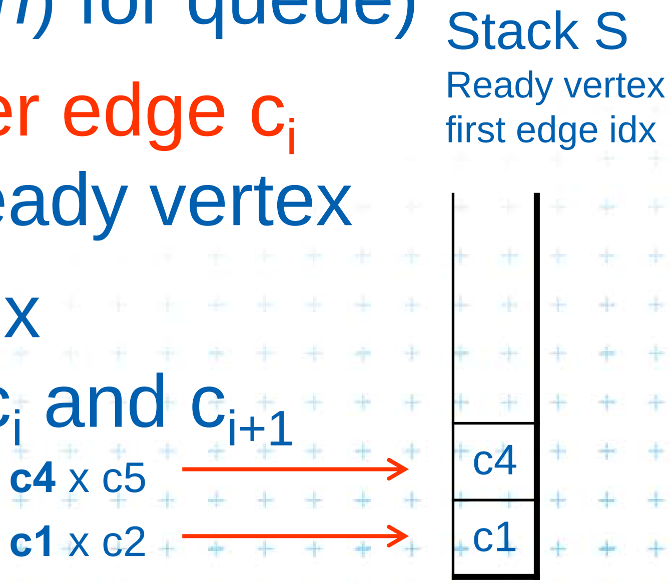
- lies on line C[c1]

- Comes from infinity

- is delimited by edge E[2]

6) Stack S

- The exact order of events is not important
  (event = intersection in ready vertex)
- Alg. can process any of the “ready vertex”
- Event queue is therefore replaced by a stack
  (faster: O(1) instead of O(log *n*) for queue)
- The stack stores just the upper edge c_i
  from the pair intersecting in ready vertex
- Intersection in the ready vertex
  is computed between stored c_i and c_{i+1}



Stack S

Ready vertex
first edge idx

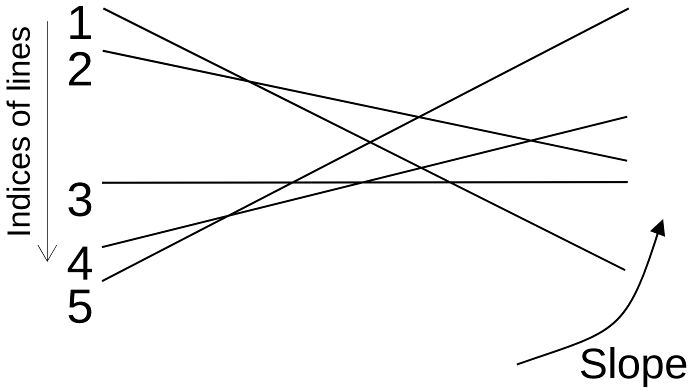
c4 x c5

c1 x c2

c4

c1

Topological sweep line demo



Input

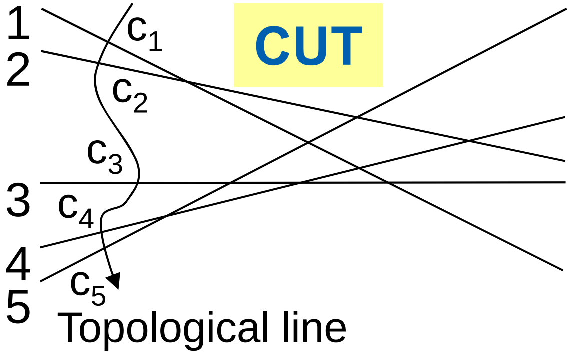
- set of lines *L* in the plane
- ordered in increasing slope (∠ -90° to 90°), simple, not vertical
- line parameters in array *E*

Array of line equations E

$y = a_i x + b$

Indices of lines		
1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

1) Initial leftmost cut - C



- Store the indices of lines in E into the Cut lines array C in increasing slope order

Array of line equations E

$y = a_i x + b$

Indices of lines

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

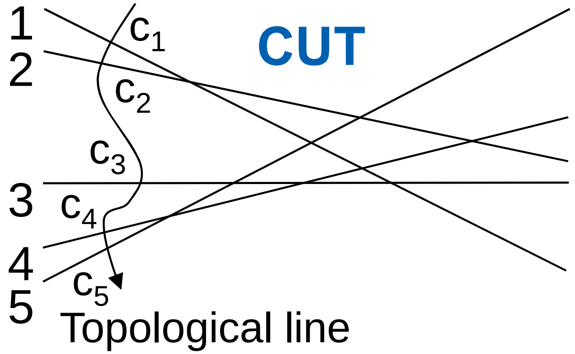
CUT Lines C

Indexes of supporting lines

Line indices along the cut

c1	1
c2	2
c3	3
c4	4
c5	5

1) Initial leftmost cut - N



- Prepare array *N* for endpoints of the cut edges (resp. for line indices delimiting these edges)
- Init it by line “ends” $-\infty, +\infty$

Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

indices of lines

CUT edges N

Pairs of line indices delimiting the edge

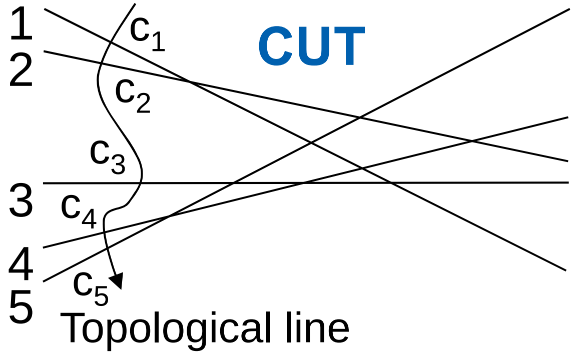
c1	$-\infty$	∞
c2	$-\infty$	∞
c3	$-\infty$	∞
c4	$-\infty$	∞
c5	$-\infty$	∞

CUT Lines C

Indexes of supporting lines

c1	1
c2	2
c3	3
c4	4
c5	5

1) Initial leftmost cut - N



- Prepare array *N* for endpoints of the cut edges (resp. for line indices delimiting these edges)
- Init it by line “ends” $-\infty, +\infty$

Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

indices of lines

CUT edges N

Pairs of line indices delimiting the edge

c1	$-\infty$	∞
c2	$-\infty$	∞
c3	$-\infty$	∞
c4	$-\infty$	∞
c5	$-\infty$	∞

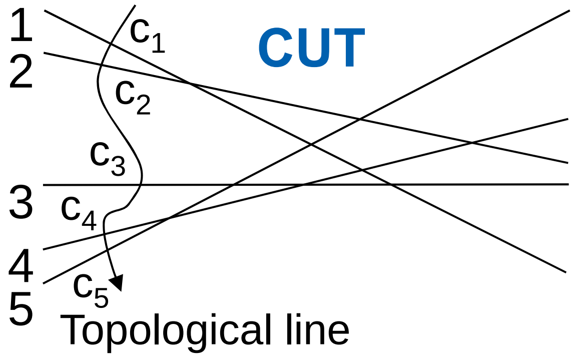
Index of delimiter edge in $-\infty$

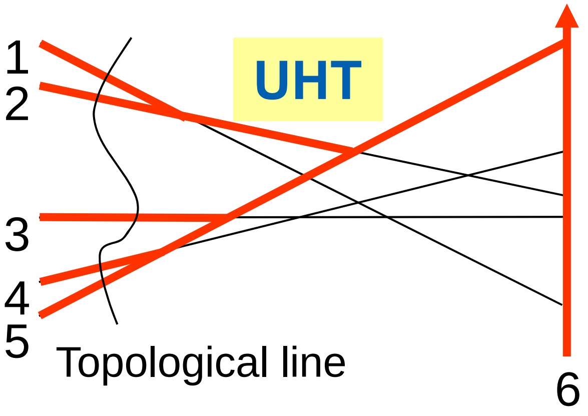
CUT Lines C

Indexes of supporting lines

c1	1
c2	2
c3	3
c4	4
c5	5

2a) Compute Upper Horizon Tree - UHT





Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	$-\infty$	2
2	$-\infty$	5
3	$-\infty$	5
4	$-\infty$	5
5	$-\infty$	6
6	$-\infty$	$+\infty$

Order of insertion into UHT

Additional “help edge“

Unlimited, bottom-up

Inserted first, never changed

CUT edges N

Pairs of line indices delimiting the edge

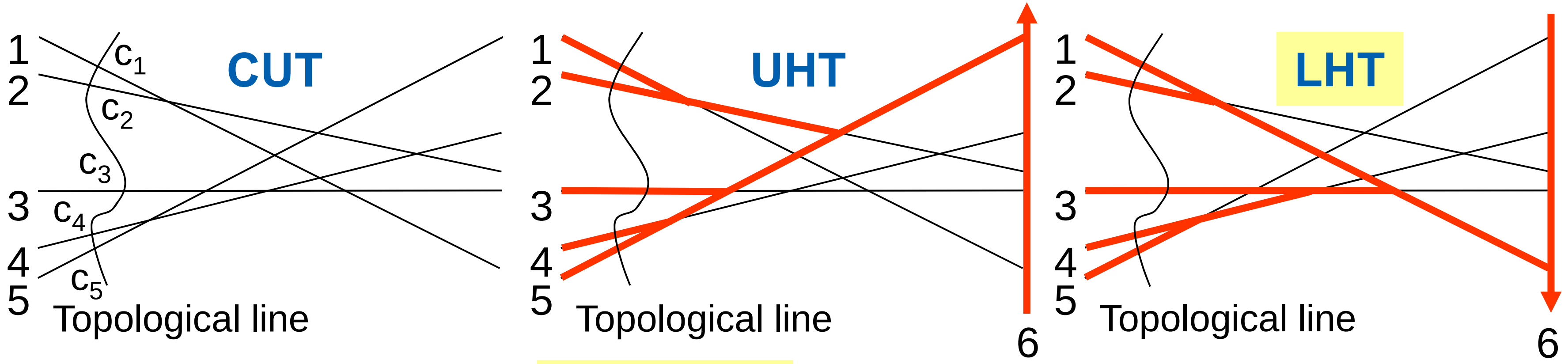
c1	$-\infty$	∞
c2	$-\infty$	∞
c3	$-\infty$	∞
c4	$-\infty$	∞
c5	$-\infty$	∞

CUT Lines C

Indexes of supporting lines

c1	1
c2	2
c3	3
c4	4
c5	5

2b) Compute Lower Horizon Tree - LHT



Array of line equations E
$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array
Delimiting lines indices

1	$-\infty$	2
2	$-\infty$	5
3	$-\infty$	5
4	$-\infty$	5
5	$-\infty$	6
6	$-\infty$	$+\infty$

LHT array
Delimiting lines indices

1	$-\infty$	6
2	$-\infty$	1
3	$-\infty$	1
4	$-\infty$	3
5	$-\infty$	4
6	$+\infty$	$-\infty$

Order of insertion into LHT

Inserted first, never changed top to bottom

CUT edges N
Pairs of line indices delimiting the edge

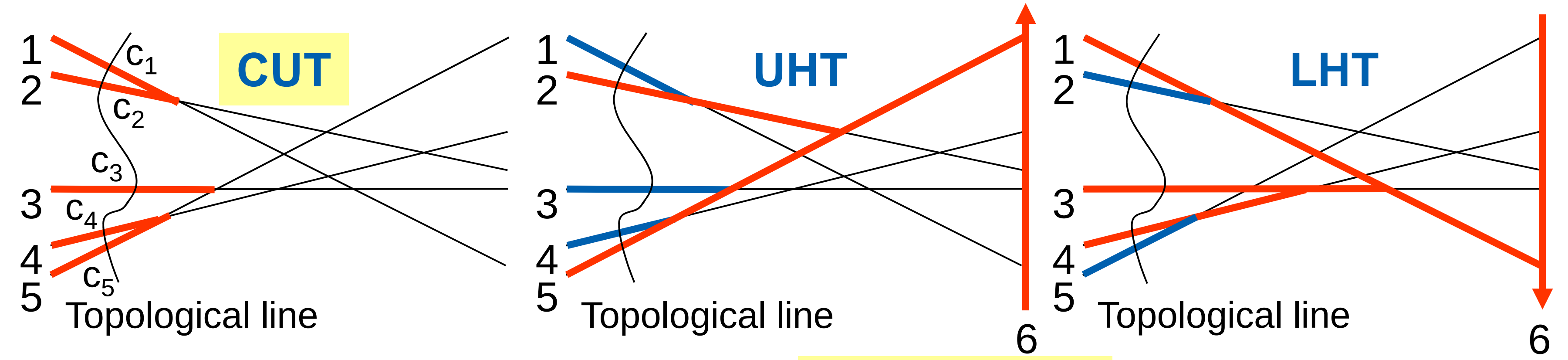
c1	$-\infty$	∞
c2	$-\infty$	∞
c3	$-\infty$	∞
c4	$-\infty$	∞
c5	$-\infty$	∞

CUT Lines C
Indexes of supporting lines

c1	1
c2	2
c3	3
c4	4
c5	5

Stack S
Ready vertex first edge idx

3a) Determine right delimiters of edges - N



Array of line equations E
$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array
Delimiting lines indices

1	$-\infty$	2
2	$-\infty$	5
3	$-\infty$	5
4	$-\infty$	5
5	$-\infty$	6
6	$-\infty$	$+\infty$

LHT array
Delimiting lines indices

1	$-\infty$	6
2	$-\infty$	1
3	$-\infty$	1
4	$-\infty$	3
5	$-\infty$	4
6	$+\infty$	$-\infty$

CUT edges N
Pairs of line indices delimiting the edge

c1	$-\infty$	**2**
c2	$-\infty$	**1**
c3	$-\infty$	**5**
c4	$-\infty$	**5**
c5	$-\infty$	**4**

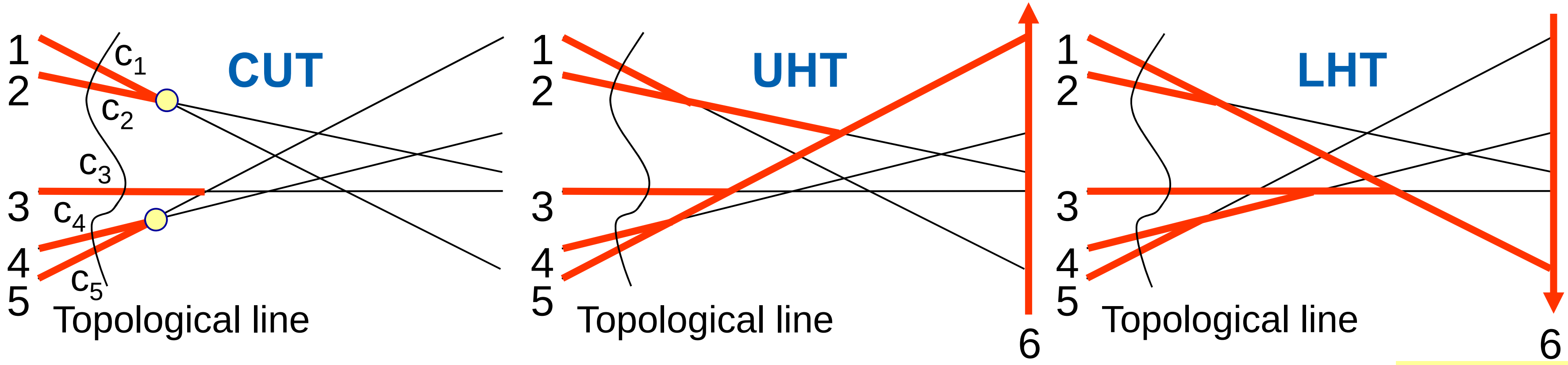
CUT Lines C
Indexes of supporting lines

c1	1
c2	2
c3	3
c4	4
c5	5

Stack S
Ready vertex first edge idx

Intersect the trees – take the shorter edge

3b) Ready vertices = inters. of neighbors – S



Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	$-\infty$	2
2	$-\infty$	5
3	$-\infty$	5
4	$-\infty$	5
5	$-\infty$	6
6	$-\infty$	$+\infty$

LHT array

Delimiting lines indices

1	$-\infty$	6
2	$-\infty$	1
3	$-\infty$	1
4	$-\infty$	3
5	$-\infty$	4
6	$+\infty$	$-\infty$

CUT edges N

Pairs of line indices delimiting the edge

c1	$-\infty$	**2**
c2	$-\infty$	**1**
c3	$-\infty$	**5**
c4	$-\infty$	**5**
c5	$-\infty$	**4**

CUT Lines C

Indexes of supporting lines

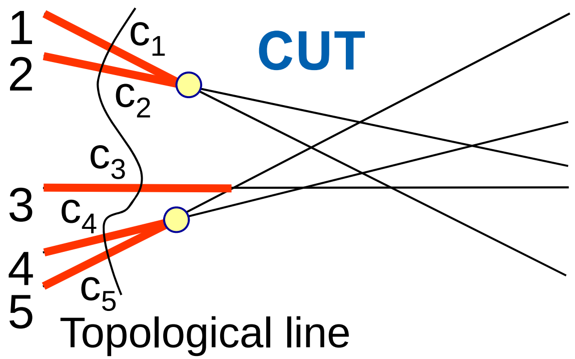
c1	1
c2	2
c3	3
c4	4
c5	5

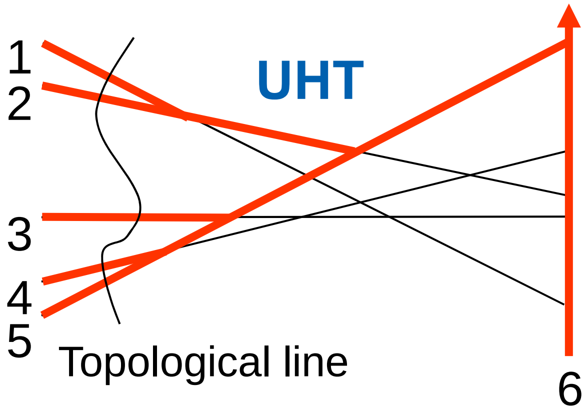
Stack S

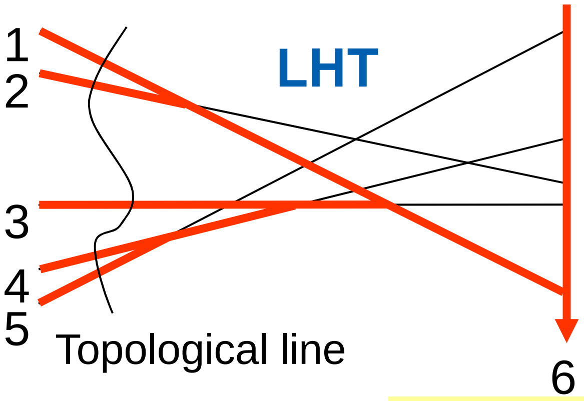
Ready vertex first edge idx

Compute intersections of neighbors – push into stack

3b) Ready vertices = inters. of neighbors – S







Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	$-\infty$	2
2	$-\infty$	5
3	$-\infty$	5
4	$-\infty$	5
5	$-\infty$	6
6	$-\infty$	$+\infty$

LHT array

Delimiting lines indices

1	$-\infty$	6
2	$-\infty$	1
3	$-\infty$	1
4	$-\infty$	3
5	$-\infty$	4
6	$+\infty$	$-\infty$

CUT edges N

Pairs of line indices delimiting the edge

c1	$-\infty$	**2**
c2	$-\infty$	**1**
c3	$-\infty$	**5**
c4	$-\infty$	**5**
c5	$-\infty$	**4**

CUT Lines C

Indexes of supporting lines

c1	1
c2	2
c3	3
c4	4
c5	5

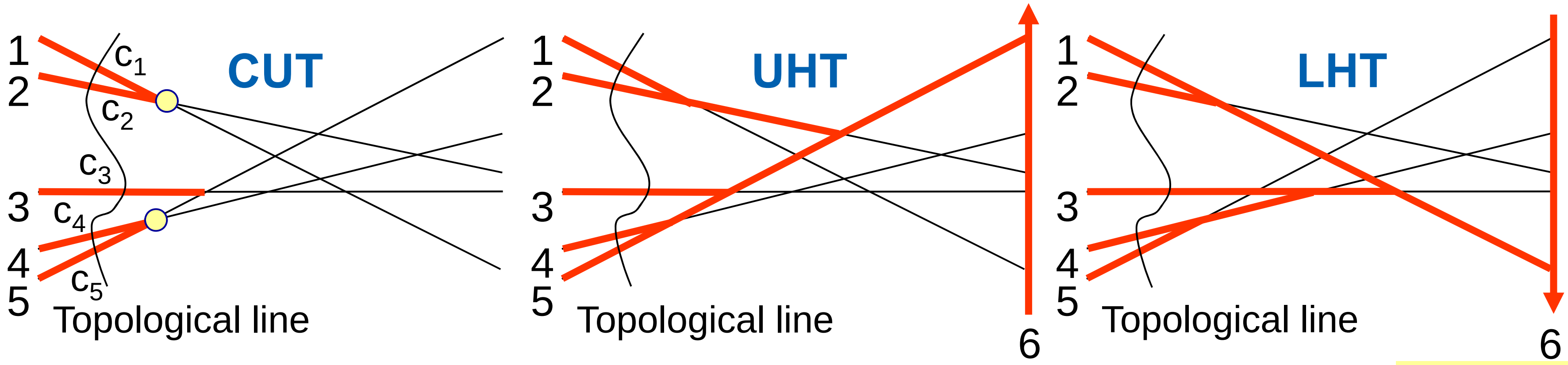
Stack S

Ready vertex first edge idx

c1

Compute intersections of neighbors – push into stack

3b) Ready vertices = inters. of neighbors – S



Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	$-\infty$	2
2	$-\infty$	5
3	$-\infty$	5
4	$-\infty$	5
5	$-\infty$	6
6	$-\infty$	$+\infty$

LHT array

Delimiting lines indices

1	$-\infty$	6
2	$-\infty$	1
3	$-\infty$	1
4	$-\infty$	3
5	$-\infty$	4
6	$+\infty$	$-\infty$

CUT edges N

Pairs of line indices delimiting the edge

c1	$-\infty$	**2**
c2	$-\infty$	**1**
c3	$-\infty$	**5**
c4	$-\infty$	**5**
c5	$-\infty$	**4**

CUT Lines C

Indexes of supporting lines

c1	1
c2	2
c3	3
c4	4
c5	5

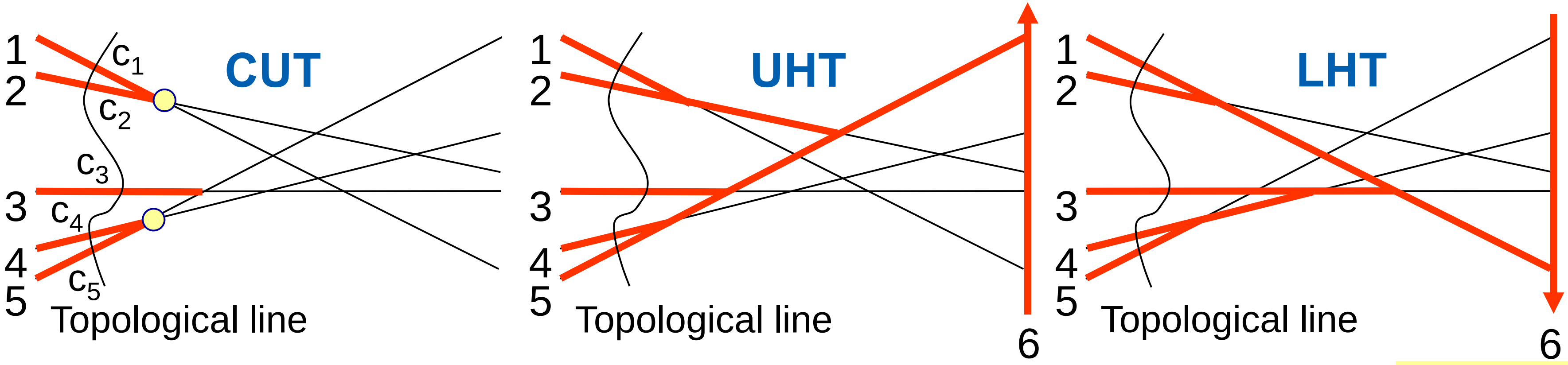
Stack S

Ready vertex first edge idx

c1

Compute intersections of neighbors – push into stack

3b) Ready vertices = inters. of neighbors – S



Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	$-\infty$	2
2	$-\infty$	5
3	$-\infty$	5
4	$-\infty$	5
5	$-\infty$	6
6	$-\infty$	$+\infty$

LHT array

Delimiting lines indices

1	$-\infty$	6
2	$-\infty$	1
3	$-\infty$	1
4	$-\infty$	3
5	$-\infty$	4
6	$+\infty$	$-\infty$

CUT edges N

Pairs of line indices delimiting the edge

c1	$-\infty$	**2**
c2	$-\infty$	**1**
c3	$-\infty$	**5**
c4	$-\infty$	**5**
c5	$-\infty$	**4**

CUT Lines C

Indexes of supporting lines

c1	1
c2	2
c3	3
c4	4
c5	5

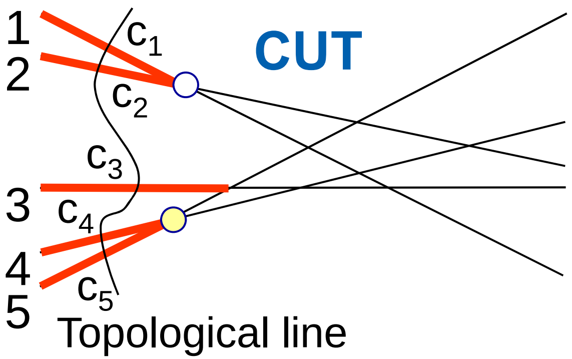
Stack S

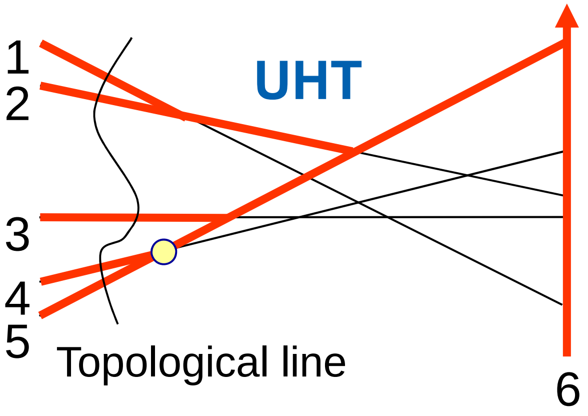
Ready vertex first edge idx

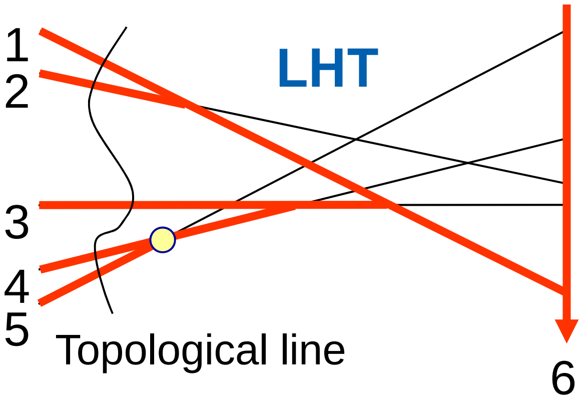
c4
c1

Compute intersections of neighbors – push into stack

4a) Pop ready vertex from S – process c4







Array of line equations E
$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array
Delimiting lines indices

1	$-\infty$	2
2	$-\infty$	5
3	$-\infty$	5
4	$-\infty$	5
5	$-\infty$	6
6	$-\infty$	$+\infty$

LHT array
Delimiting lines indices

1	$-\infty$	6
2	$-\infty$	1
3	$-\infty$	1
4	$-\infty$	3
5	$-\infty$	4
6	$+\infty$	$-\infty$

CUT edges N
Pairs of line indices delimiting the edge

c1	$-\infty$	2
c2	$-\infty$	1
c3	$-\infty$	5
c4	$-\infty$	5
c5	$-\infty$	4

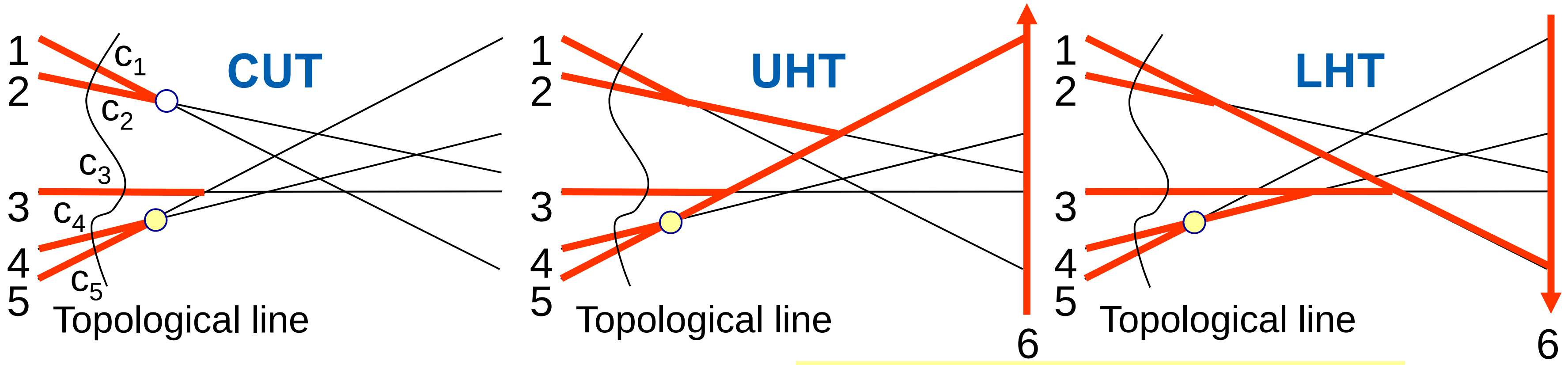
CUT Lines C
Indexes of supporting lines

c1	1
c2	2
c3	3
c4	4
c5	5

Stack S
Ready vertex first edge idx

c4
c1

4b) Swap lines c4 and c5 – swap 4 and 5



Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	$-\infty$	2
2	$-\infty$	5
3	$-\infty$	5
4	$-\infty$	5
5	$-\infty$	6
6	$-\infty$	$+\infty$

LHT array

Delimiting lines indices

1	$-\infty$	6
2	$-\infty$	1
3	$-\infty$	1
4	$-\infty$	3
5	$-\infty$	4
6	$+\infty$	$-\infty$

CUT edges N

Pairs of line indices delimiting the edge

c1	$-\infty$	2
c2	$-\infty$	1
c3	$-\infty$	5
c4	$-\infty$	4
c5	$-\infty$	5

CUT Lines C

Indexes of supporting lines

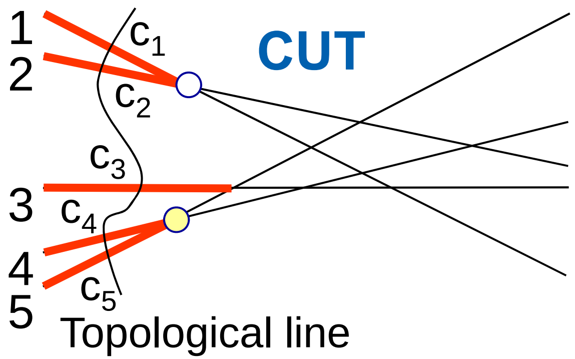
c1	1
c2	2
c3	3
c4	**5**
c5	**4**

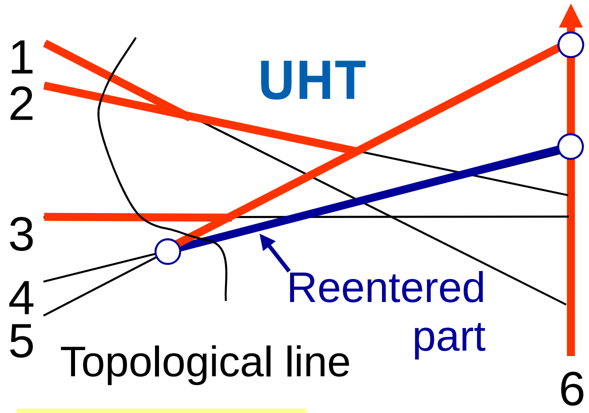
Stack S

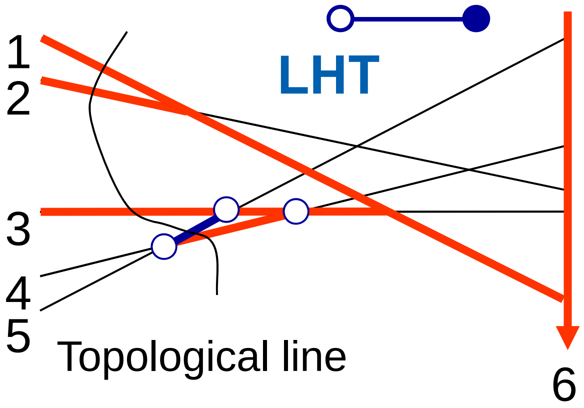
Ready vertex first edge idx

c1

4c) Update the horizon trees – UHT and LHT







Array of line equations E
$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array
Delimiting lines indices

1	–∞	2
2	–∞	5
3	–∞	5
4	**5**	**6**
5	**4**	**6**
6	–∞	+∞

LHT array
Delimiting lines indices

1	–∞	6
2	–∞	1
3	–∞	1
4	**5**	3
5	**4**	**3**
6	+∞	–∞

CUT edges N
Pairs of line indices delimiting the edge

c1	–∞	2
c2	–∞	1
c3	–∞	5
c4	–∞	4
c5	–∞	5

CUT Lines C
Indexes of supporting lines

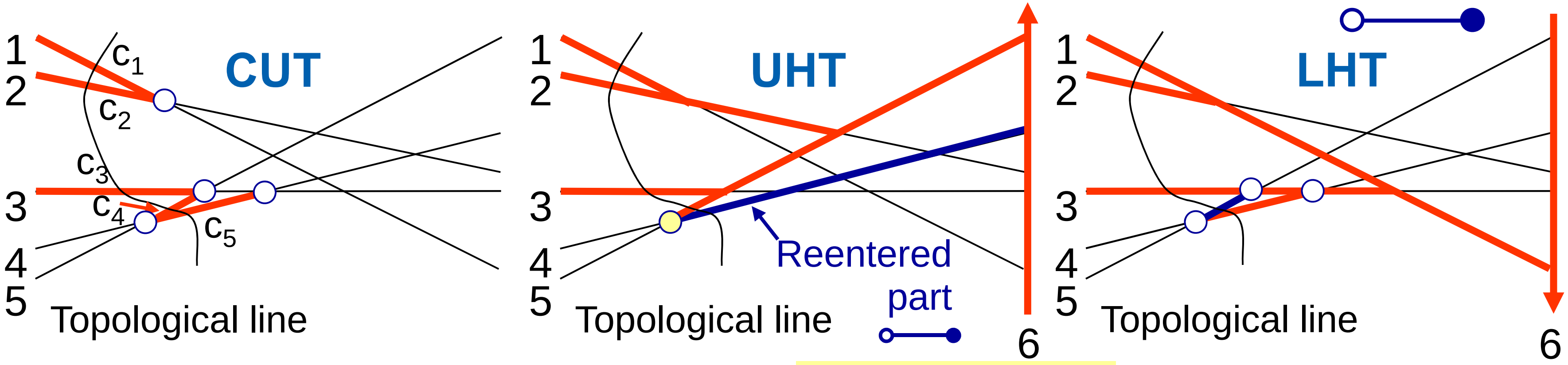
c1	1
c2	2
c3	3
c4	**5**
c5	**4**

Stack S
Ready vertex upper edge idx

c1

Note: ○—● Edges are half open to prevent the tree after reinsertion

4d) Determine new cut edges endpoints – N



Array of line equations E
$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array
Delimiting lines indices

1	–∞	2
2	–∞	5
3	–∞	5
4	**5**	**6**
5	**4**	**6**
6	–∞	+∞

LHT array
Delimiting lines indices

1	–∞	6
2	–∞	1
3	–∞	1
4	**5**	3
5	**4**	3
6	+∞	–∞

CUT edges N
Pairs of line indices delimiting the edge

c1	–∞	2
c2	–∞	1
c3	–∞	5
c4	**4**	**3**
c5	**5**	**3**

CUT Lines C
Indexes of supporting lines

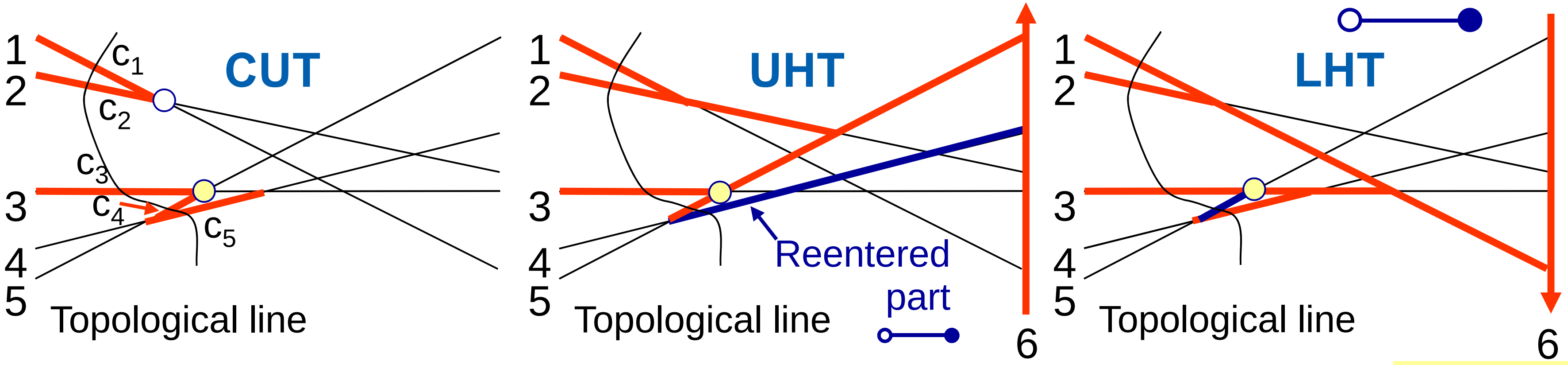
c1	1
c2	2
c3	3
c4	**5**
c5	**4**

Stack S
Ready vertex upper edge idx

c1

Intersect the trees – take the shorter edge

4e) Intersect with neighbors – push into S



Array of line equations E $y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array Delimiting lines indices

1	$-\infty$	2
2	$-\infty$	5
3	$-\infty$	5
4	**5**	**6**
5	**4**	**6**
6	$-\infty$	$+\infty$

LHT array Delimiting lines indices

1	$-\infty$	6
2	$-\infty$	1
3	$-\infty$	1
4	**5**	3
5	**4**	**3**
6	$+\infty$	$-\infty$

CUT edges N Pairs of line indices delimiting the edge

c1	$-\infty$	2
c2	$-\infty$	1
c3	$-\infty$	5
c4	**4**	**3**
c5	**5**	**3**

CUT Lines C Indexes of supporting lines

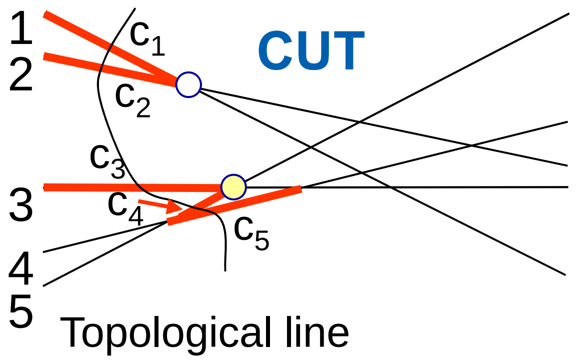
c1	1
c2	2
c3	3
c4	**5**
c5	**4**

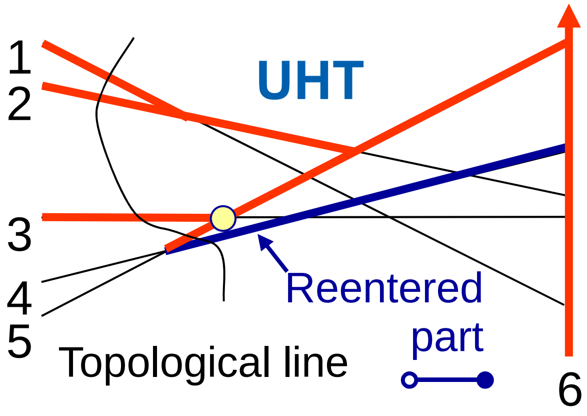
Stack S Ready vertex upper edge idx

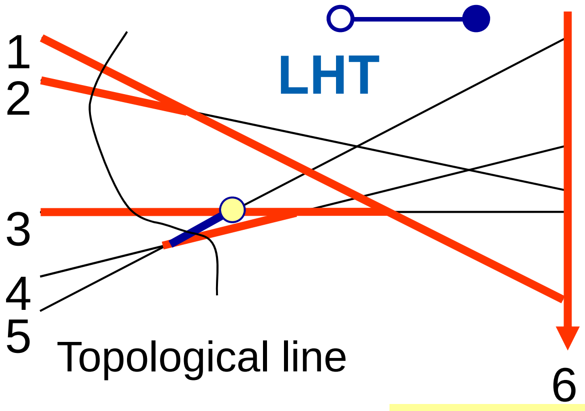
c1

Intersections of neighbors - into stack

4e) Intersect with neighbors – push into S







Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	−∞	2
2	−∞	5
3	−∞	5
4	**5**	**6**
5	**4**	**6**
6	−∞	+∞

LHT array

Delimiting lines indices

1	−∞	6
2	−∞	1
3	−∞	1
4	**5**	3
5	**4**	**3**
6	+∞	−∞

CUT edges N

Pairs of line indices delimiting the edge

c1	−∞	2
c2	−∞	1
c3	−∞	5
c4	**4**	**3**
c5	**5**	**3**

CUT Lines C

Indexes of supporting lines

c1	1
c2	2
c3	3
c4	**5**
c5	**4**

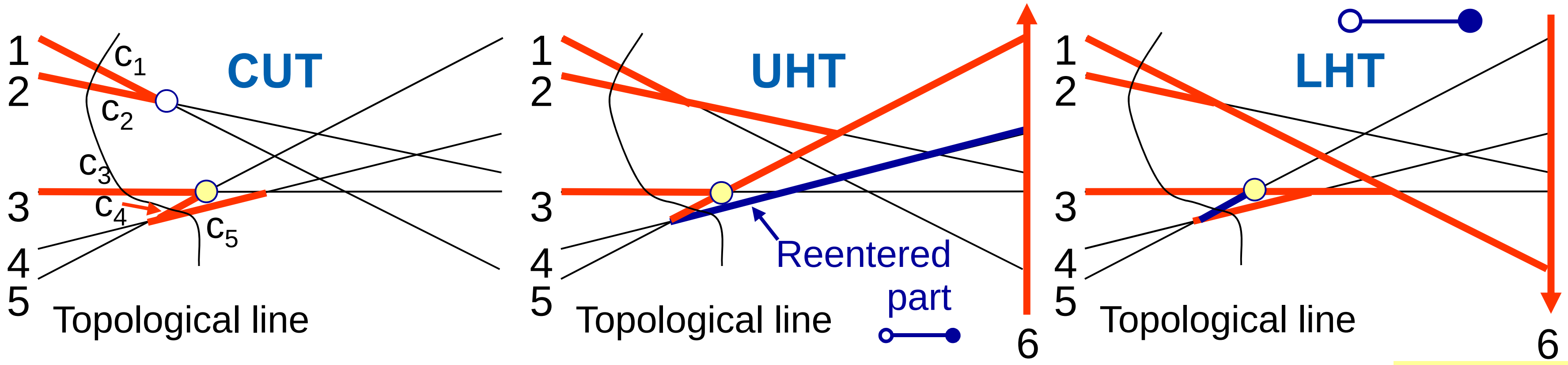
Stack S

Ready vertex upper edge idx

c1

Intersections of neighbors - into stack

4e) Intersect with neighbors – push into S



Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	−∞	2
2	−∞	5
3	−∞	5
4	**5**	**6**
5	**4**	**6**
6	−∞	+∞

LHT array

Delimiting lines indices

1	−∞	6
2	−∞	1
3	−∞	1
4	**5**	3
5	**4**	**3**
6	+∞	−∞

CUT edges N

Pairs of line indices delimiting the edge

c1	−∞	2
c2	−∞	1
c3	−∞	5
c4	**4**	**3**
c5	**5**	**3**

CUT Lines C

Indexes of supporting lines

c1	1
c2	2
c3	3
c4	**5**
c5	**4**

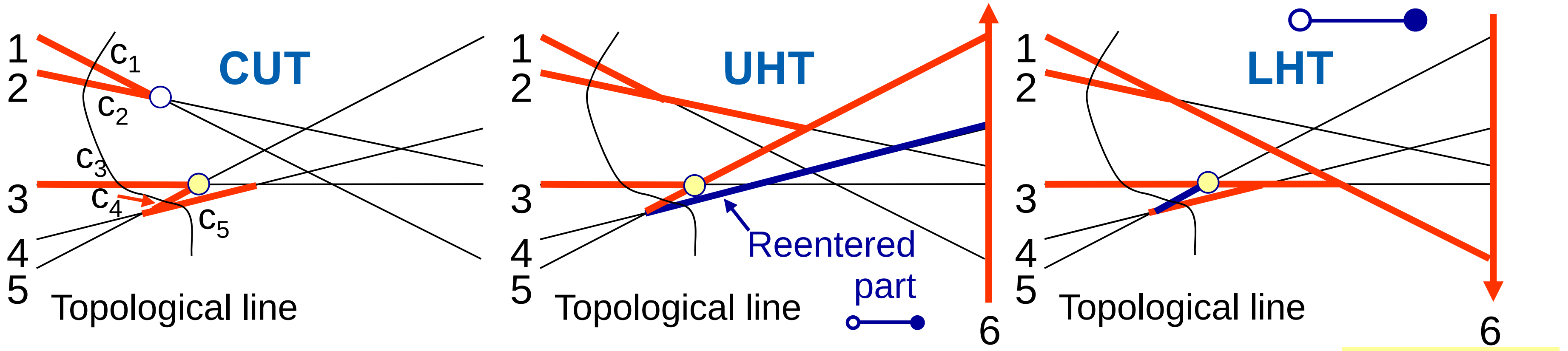
Stack S

Ready vertex upper edge id

c1

Intersections of neighbors - into stack

4e) Intersect with neighbors – push into S



Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	–∞	2
2	–∞	5
3	–∞	5
4	**5**	**6**
5	**4**	**6**
6	–∞	+∞

LHT array

Delimiting lines indices

1	–∞	6
2	–∞	1
3	–∞	1
4	**5**	3
5	**4**	**3**
6	+∞	–∞

CUT edges N

Pairs of line indices delimiting the edge

c1	–∞	2
c2	–∞	1
c3	–∞	5
c4	**4**	**3**
c5	**5**	**3**

CUT Lines C

Indexes of supporting lines

c1	1
c2	2
c3	3
c4	**5**
c5	**4**

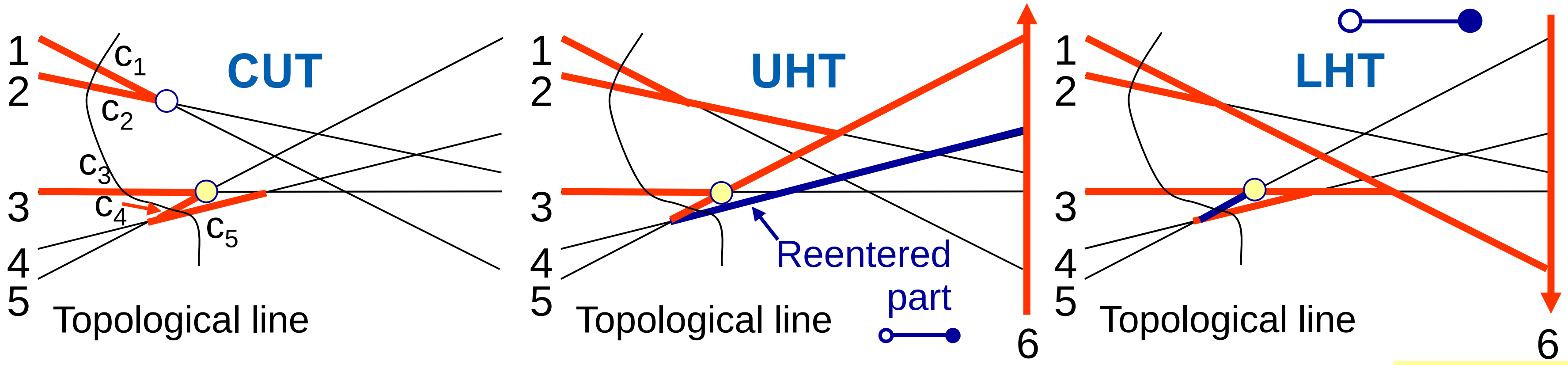
Stack S

Ready vertex upper edge idx

c1

Intersections of neighbors - into stack

4e) Intersect with neighbors – push into S



Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	–∞	2
2	–∞	5
3	–∞	5
4	**5**	**6**
5	**4**	**6**
6	–∞	+∞

LHT array

Delimiting lines indices

1	–∞	6
2	–∞	1
3	–∞	1
4	**5**	3
5	**4**	**3**
6	+∞	–∞

CUT edges N

Pairs of line indices delimiting the edge

c1	–∞	2
c2	–∞	1
c3	–∞	5
c4	**4**	**3**
c5	**5**	**3**

CUT Lines C

Indexes of supporting lines

c1	1
c2	2
c3	3
c4	**5**
c5	**4**

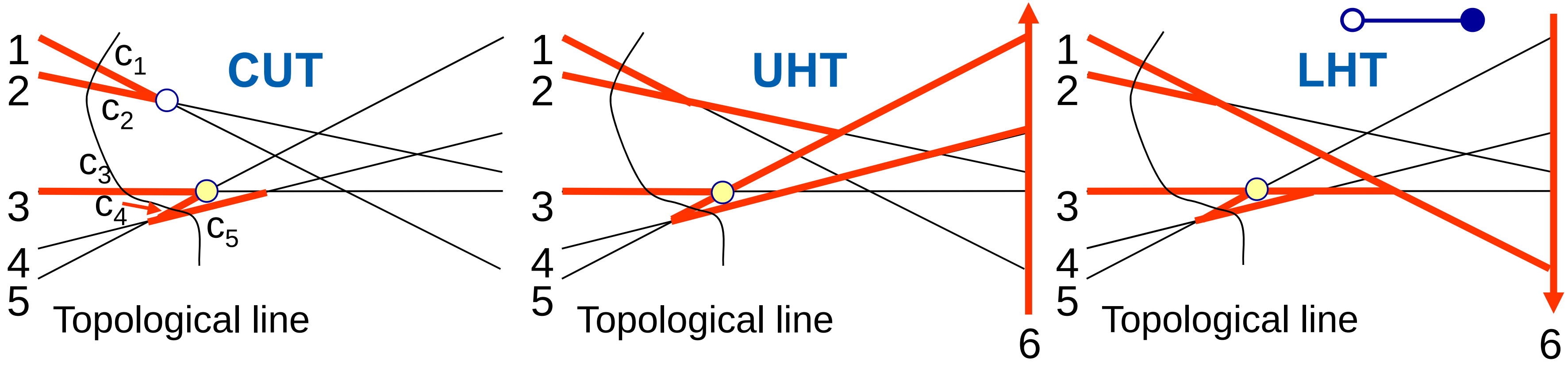
Stack S

Ready vertex upper edge idx

c3
c1

Intersections of neighbors - into stack

4a) Pop ready vertex from S – process c3



Array of line equations E

$y = a_ix + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	$-\infty$	2
2	$-\infty$	5
3	$-\infty$	5
4	5	6
5	4	6
6	$-\infty$	$+\infty$

LHT array

Delimiting lines indices

1	$-\infty$	6
2	$-\infty$	1
3	$-\infty$	1
4	5	3
5	4	3
6	$+\infty$	$-\infty$

CUT edges N

Pairs of line indices delimiting the edge

c1	$-\infty$	2
c2	$-\infty$	1
c3	$-\infty$	5
c4	4	3
c5	5	3

CUT Lines C

Indexes of supporting lines

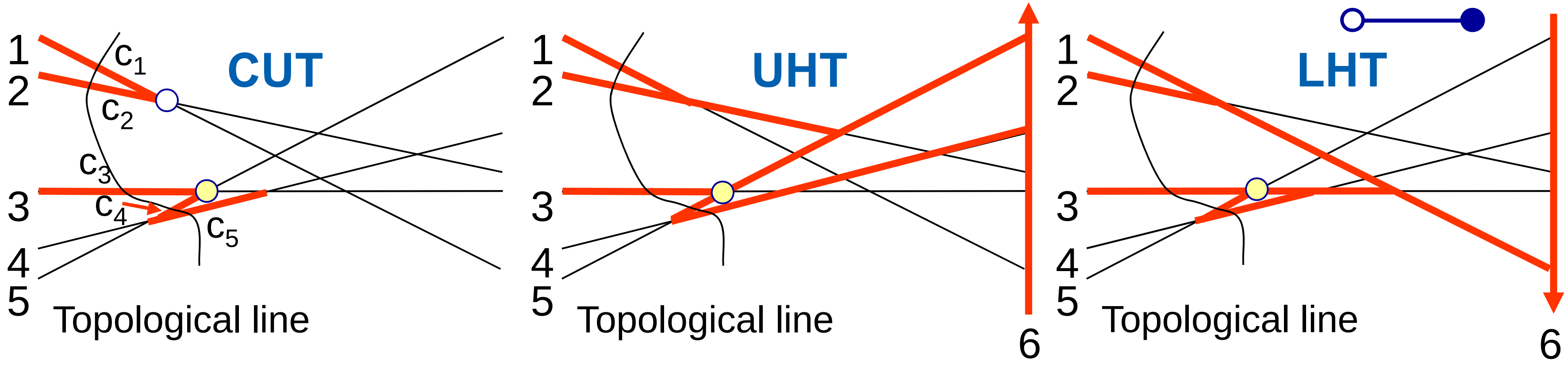
c1	1
c2	2
c3	**3**
c4	**5**
c5	4

Stack S

Ready vertex first edge idx

c3
c1

4a) Pop ready vertex from S – process c3



Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	–∞	2
2	–∞	5
3	–∞	5
4	5	6
5	4	6
6	–∞	+∞

LHT array

Delimiting lines indices

1	–∞	6
2	–∞	1
3	–∞	1
4	5	3
5	4	3
6	+∞	–∞

CUT edges N

Pairs of line indices delimiting the edge

c1	–∞	2
c2	–∞	1
c3	–∞	5
c4	4	3
c5	5	3

CUT Lines C

Indexes of supporting lines

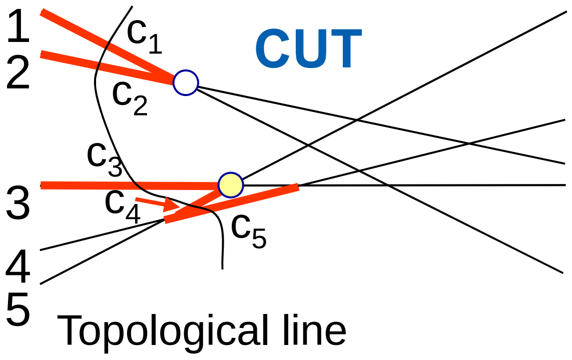
c1	1
c2	2
c3	**3**
c4	**5**
c5	4

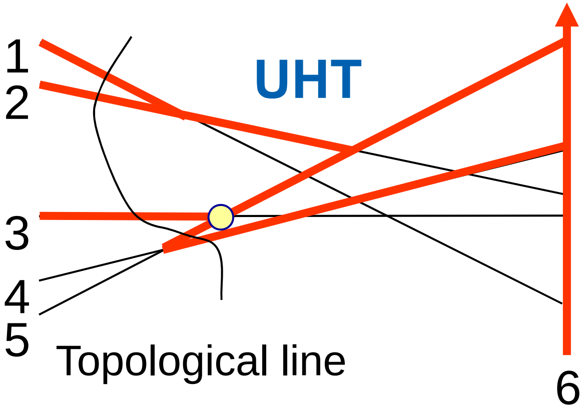
Stack S

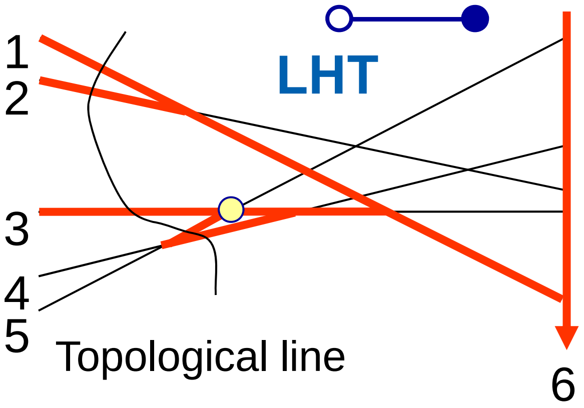
Ready vertex first edge idx

c3
c1

4a) Pop ready vertex from S – process c3







Array of line equations E

$y = a_ix + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	$-\infty$	2
2	$-\infty$	5
3	$-\infty$	5
4	5	6
5	4	6
6	$-\infty$	$+\infty$

LHT array

Delimiting lines indices

1	$-\infty$	6
2	$-\infty$	1
3	$-\infty$	1
4	5	3
5	4	3
6	$+\infty$	$-\infty$

CUT edges N

Pairs of line indices delimiting the edge

c1	$-\infty$	2
c2	$-\infty$	1
c3	$-\infty$	5
c4	4	3
c5	5	3

CUT Lines C

Indexes of supporting lines

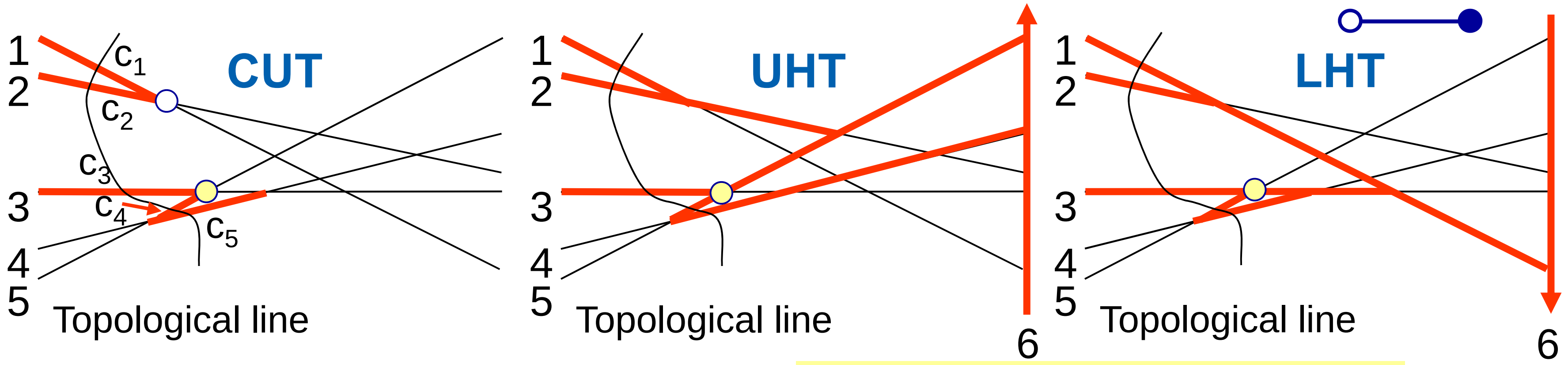
c1	1
c2	2
c3	**3**
c4	**5**
c5	4

Stack S

Ready vertex first edge idx

c1

4b) Swap lines c4 and c5 – swap 4 and 5



Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	$-\infty$	2
2	$-\infty$	5
3	$-\infty$	5
4	5	6
5	4	6
6	$-\infty$	$+\infty$

LHT array

Delimiting lines indices

1	$-\infty$	6
2	$-\infty$	1
3	$-\infty$	1
4	5	3
5	4	3
6	$+\infty$	$-\infty$

CUT edges N

Pairs of line indices delimiting the edge

c1	$-\infty$	2
c2	$-\infty$	1
c3	4	3
c4	$-\infty$	5
c5	5	3

Swapped invalidated

CUT Lines C

Indexes of supporting lines

c1	1
c2	2
c3	**5**
c4	**3**
c5	4

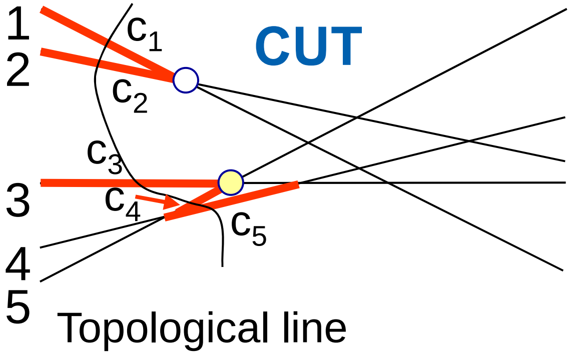
Swapped

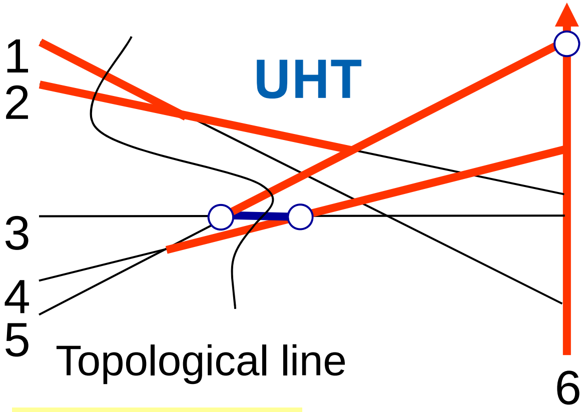
Stack S

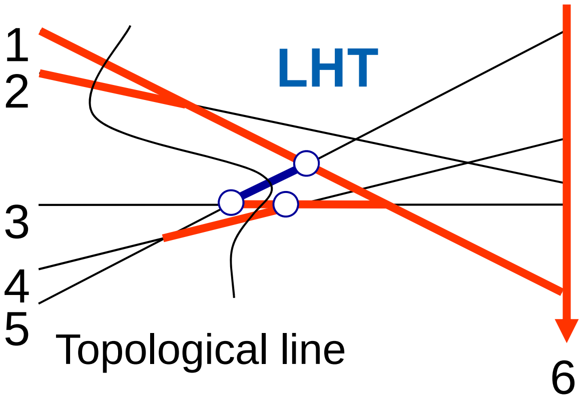
Ready vertex first edge idx

c1

4c) Update the horizon trees – UHT and LHT







Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	–∞	2
2	–∞	5
3	**5**	**4**
4	5	6
5	**3**	6
6	–∞	+∞

LHT array

Delimiting lines indices

1	–∞	6
2	–∞	1
3	**5**	1
4	5	3
5	**3**	**1**
6	+∞	–∞

CUT edges N

Pairs of line indices delimiting the edge

c1	–∞	2
c2	–∞	1
c3	4	3
c4	–∞	5
c5	5	3

CUT Lines C

Indexes of supporting lines

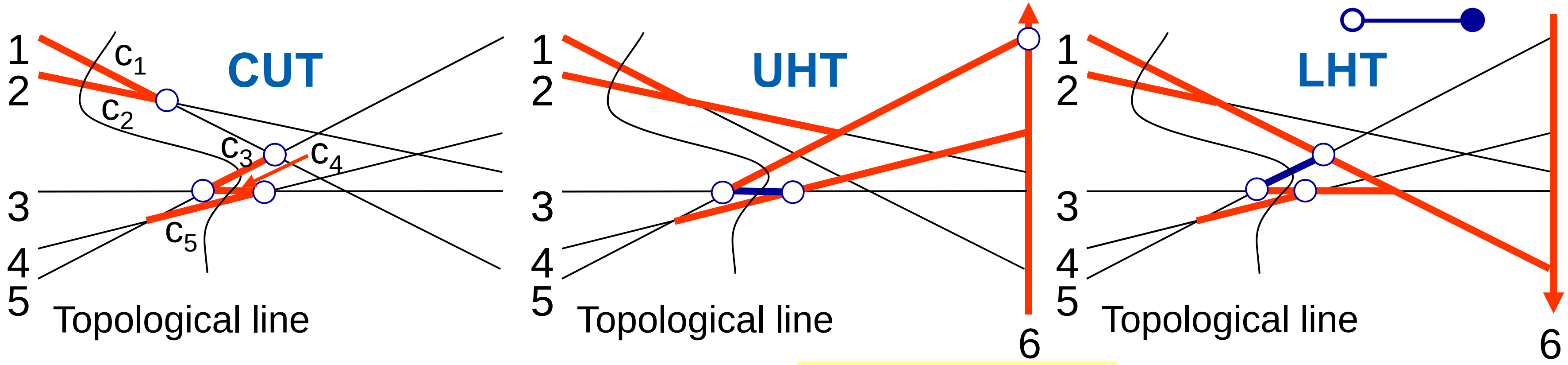
c1	1
c2	2
c3	**5**
c4	**3**
c5	4

Stack S

Ready vertex first edge idx

c1

4d) Determine new cut edges endpoints



Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	−∞	2
2	−∞	5
3	**5**	**4**
4	5	6
5	**3**	6
6	−∞	+∞

LHT array

Delimiting lines indices

1	−∞	6
2	−∞	1
3	**5**	1
4	5	3
5	**3**	**1**
6	+∞	−∞

CUT edges N

Pairs of line indices delimiting the edge

c1	−∞	2
c2	−∞	1
c3	3	1
c4	5	4
c5	5	3

CUT Lines C

Indexes of supporting lines

c1	1
c2	2
c3	**5**
c4	**3**
c5	4

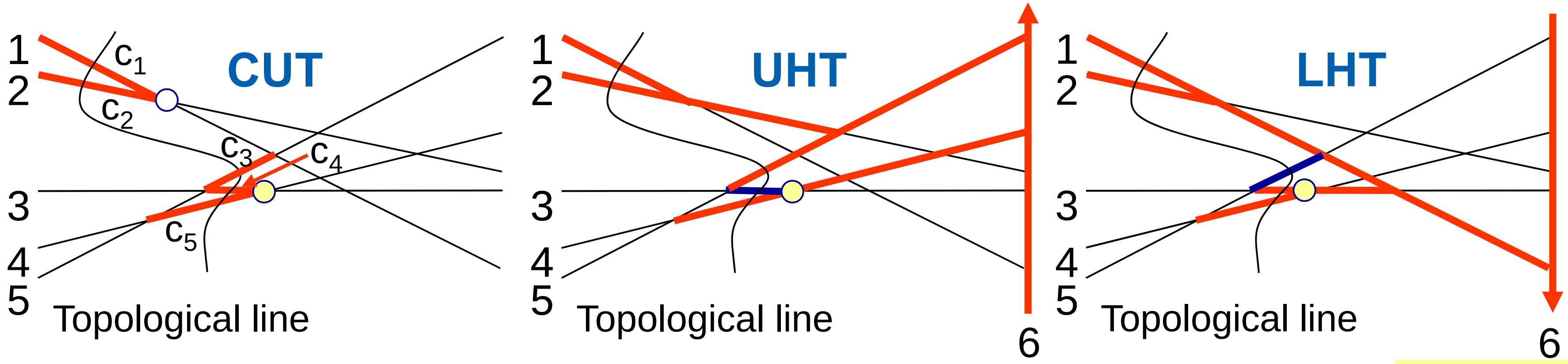
Stack S

Ready vertex first edge idx

c1

Intersect the trees – take the shorter edge

4e) Intersect with neighbors – push into S



Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	–∞	2
2	–∞	5
3	**5**	**4**
4	5	6
5	**3**	6
6	–∞	+∞

LHT array

Delimiting lines indices

1	–∞	6
2	–∞	1
3	**5**	1
4	5	3
5	**3**	**1**
6	+∞	–∞

CUT edges N

Pairs of line indices delimiting the edge

c1	–∞	2
c2	–∞	1
c3	3	1
c4	5	4
c5	5	3

CUT Lines C

Indexes of supporting lines

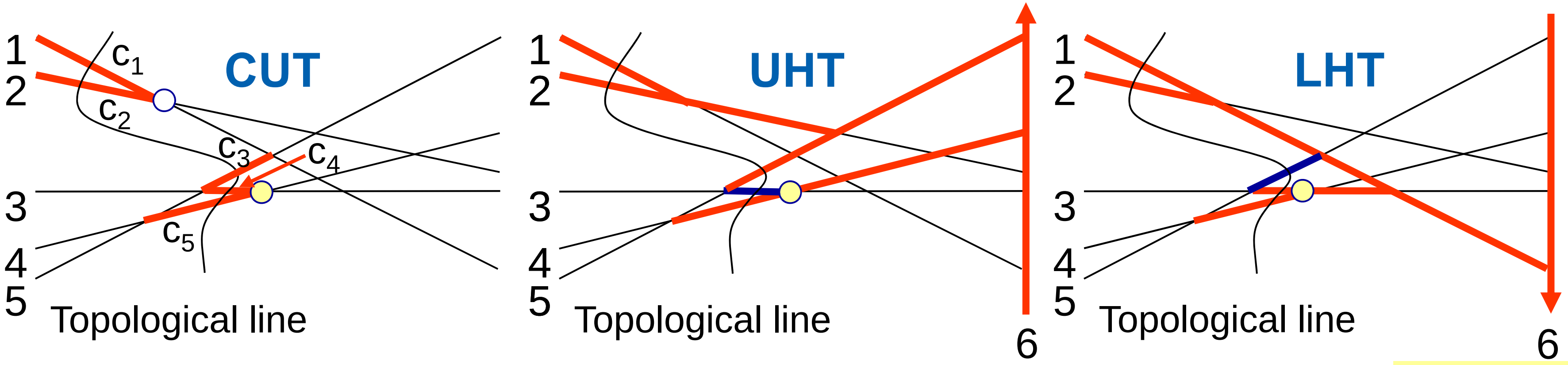
c1	1
c2	2
c3	**5**
c4	**3**
c5	4

Stack S

Ready vertex first edge idx

c1

4e) Intersect with neighbors – push into S



Array of line equations E

$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array

Delimiting lines indices

1	$-\infty$	2
2	$-\infty$	5
3	**5**	**4**
4	5	6
5	**3**	6
6	$-\infty$	$+\infty$

LHT array

Delimiting lines indices

1	$-\infty$	6
2	$-\infty$	1
3	**5**	1
4	5	3
5	**3**	**1**
6	$+\infty$	$-\infty$

CUT edges N

Pairs of line indices delimiting the edge

c1	$-\infty$	2
c2	$-\infty$	1
c3	3	1
c4	5	4
c5	5	3

CUT Lines C

Indexes of supporting lines

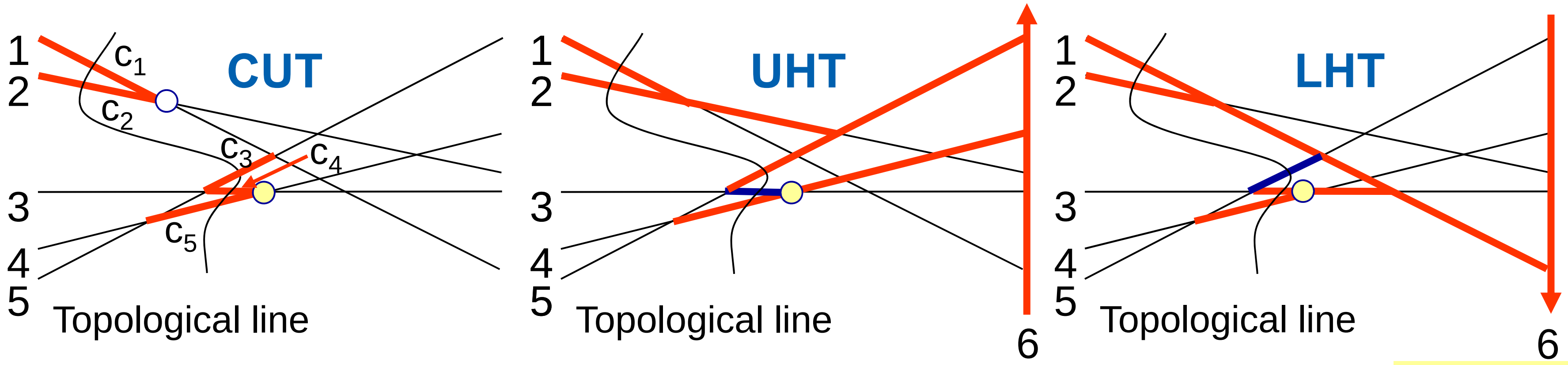
c1	1
c2	2
c3	**5**
c4	**3**
c5	4

Stack S

Ready vertex first edge idx

c1

4e) Intersect with neighbors – push into S



Array of line equations E
$y = a_i x + b$

1	a_1	b_1
2	a_2	b_2
3	a_3	b_3
4	a_4	b_4
5	a_5	b_5

UHT array
Delimiting lines indices

1	–∞	2
2	–∞	5
3	**5**	**4**
4	5	6
5	**3**	6
6	–∞	+∞

LHT array
Delimiting lines indices

1	–∞	6
2	–∞	1
3	**5**	1
4	5	3
5	**3**	**1**
6	+∞	–∞

CUT edges N
Pairs of line indices delimiting the edge

c1	–∞	2
c2	–∞	1
c3	3	1
c4	5	4
c5	5	3

CUT Lines C
Indexes of supporting lines

c1	1
c2	2
c3	**5**
c4	**3**
c5	4

Stack S
Ready vertex first edge idx

c4
c1

Topological sweep algorithm

Slope

TopoSweep(*L*)

Input: Set of **lines *L* sorted by slope (-90° to 90°)**, simple, not vertical

Output: All parts of an **Arrangement *A*(*L*)** detected and then destroyed

1. Let *C* be the initial (leftmost) cut – lines in increasing order of slope
2. Create the initial UHT and LHT incrementally:
   a) UHT by inserting lines in decreasing order of slope
   b) LHT by inserting lines in increasing order of slope
3. By consulting UHT and LHT
   a) Determine the right endpoints N of all edges of the initial cut C
   b) Store neighboring lines with common endpoint into stack *S* (initial set of *ready vertices*)
4. Repeat until stack not empty
   a) Pop next ready vertex from stack *S* (its upper edge c_i)
   b) Swap these lines within the cut *C* (c_i <-> c_{i+1})
   c) Update the horizon trees UHT and LHT (reenter edge parts)
   d) Consulting UHT and LHT determine new cut edges endpoints N
   e) If new neighboring edges share an endpoint -> push them on *S*

4d) Determining cut edges from UHT and LHT

- for lines *i* = 1 to n
  - Compare UHT and LHT edges on line *i*
  - Set the cut lying on edge *i* to the shorter edge of these
- Order of the cuts along the sweep line
  - Order changes only at the intersection *v* (neighbors)
  - Order of remaining cuts not incident with intersection *v* does not change
- After changes of the order, test the new neighbors for intersections
  - Store intersections right from sweep line into the stack

Complexity

- $O(n^2)$ intersections
  => $O(n^2)$ events (elementary steps)
- $O(1)$ amortized time for one step – 4c)

  => $O(n^2)$ time for the algorithm

Amortized time

= even though a single elementary step can take more than $O(1)$ time, the total time needed to perform $O(n^2)$ elementary steps is $O(n^2)$, hence the average time for each step is $O(1)$.

DCGI

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 8., http://www.cs.uu.nl/geobook/

[Mount] Mount, D.: *Computational Geometry Lecture Notes for Fall 2016*, University of Maryland, Lectures 14, 15, and 27. http://www.cs.umd.edu/class/fall2016/cmsc754/Lects/cmsc754-fall16-lects.pdf

[Edelsbrunner] Edelsbrunner and Guibas. Topologically sweeping an arrangement. TR 9, 1986, Digital www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-9.pdf

[Rafalin] E. Rafalin, D. Souvaine, I. Streinu, "Topological Sweep in Degenerate cases", in Proceedings of the 4th international workshop on Algorithm Engineering and Experiments, ALENEX 02, in LNCS 2409, Springer-Verlag, Berlin, Germany, pages 155-156. http://www.cs.tufts.edu/research/geometry/other/sweep/paper.pdf

[Agarwal] Pankaj K. Agarwal and Mica Sharir. Arrangements and Their Applications, 1998, http://www.math.tau.ac.il/~michas/arrsurv.pdf

DCGI

KATEDRA POČÍTAČOVÉ GRAFIKY A INTERAKCE

DUALITY AND APPLICATIONS OF ARRANGEMENTS

PETR FELKEL

FEL CTU PRAGUE

Version from 5.2.2017

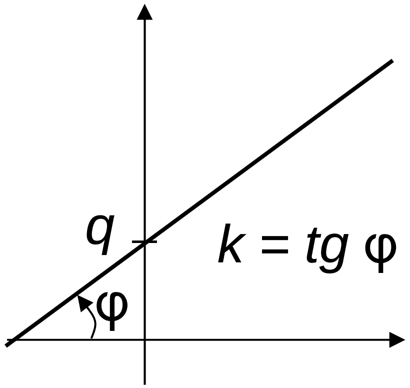
Talk overview

- Duality
  1. Points and lines
  2. Line segments
  3. Polar duality (different points and lines)
  4. Convex hull using duality
- Applications of duality and arrangements

1. Duality of lines and points in the plane

- Points and lines - both have 2 parameters:
  - Points – coords *x* and *y*
  - Lines – slope *k* and y-intercept *q*

    $y = kx + q$



- We can simply map points and lines 1:1
- Many mappings exist – it depends on the context

Why to use duality?

Some reasons why to use duality:

- Transforming a problem to dual plane may give a new view on the problem
- Looking from a different angle may give the insight needed to solve it
- Solution in dual space may be even simpler

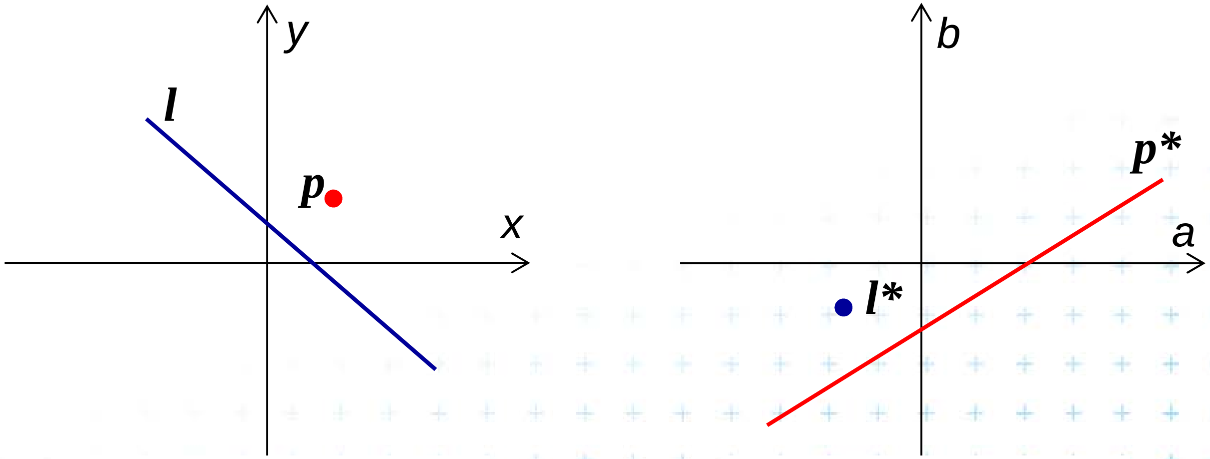
Definition of duality transformation D

Let D be the duality transform:

variables

- Point $p = [\ p_x,\ p_y\ ]$ is transformed to line $\boldsymbol{D_p} = p^* := (\ b = p_x a - p_y\ )$
- Line $\boldsymbol{l} : (\ y = ax - b\ )$ is transformed to point $\boldsymbol{D_l} = \boldsymbol{l}^* := [\ a,\ b\ ]$

constants



Primal plane (xy)

Dual plane (ab)

Example and more about duality *D*

See the [applet]

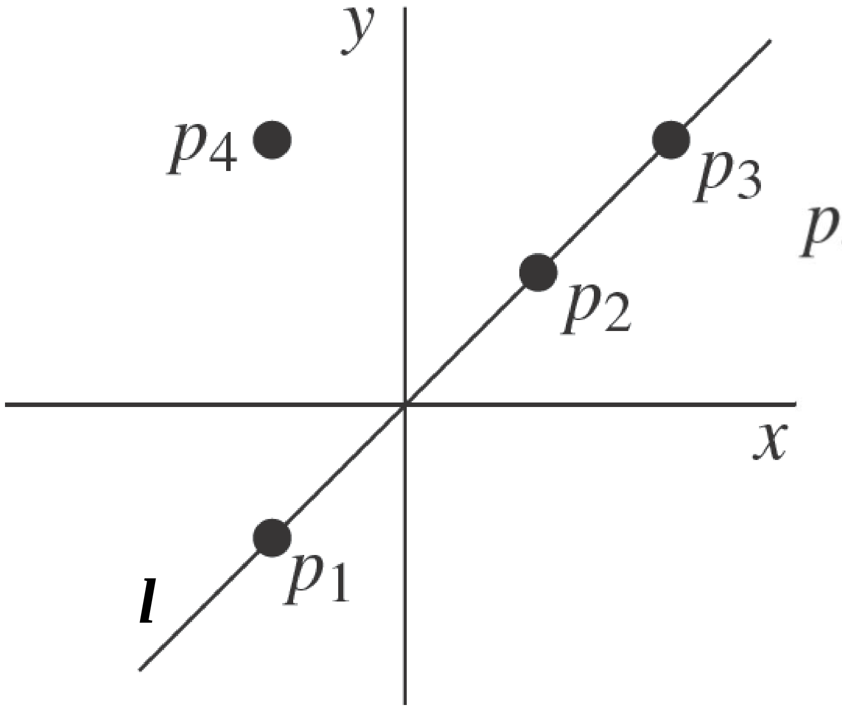
- Example:
  line $y = 5x - 3$
  can be represented as point $y^*=[5, 3]$

- Duality *D*
  - is its own inverse $DD_p = p$, $DD_l = l$
  - cannot represent vertical lines
    =>Take vertical lines as special cases, use lexicographic order, or rotate the problem space slightly.

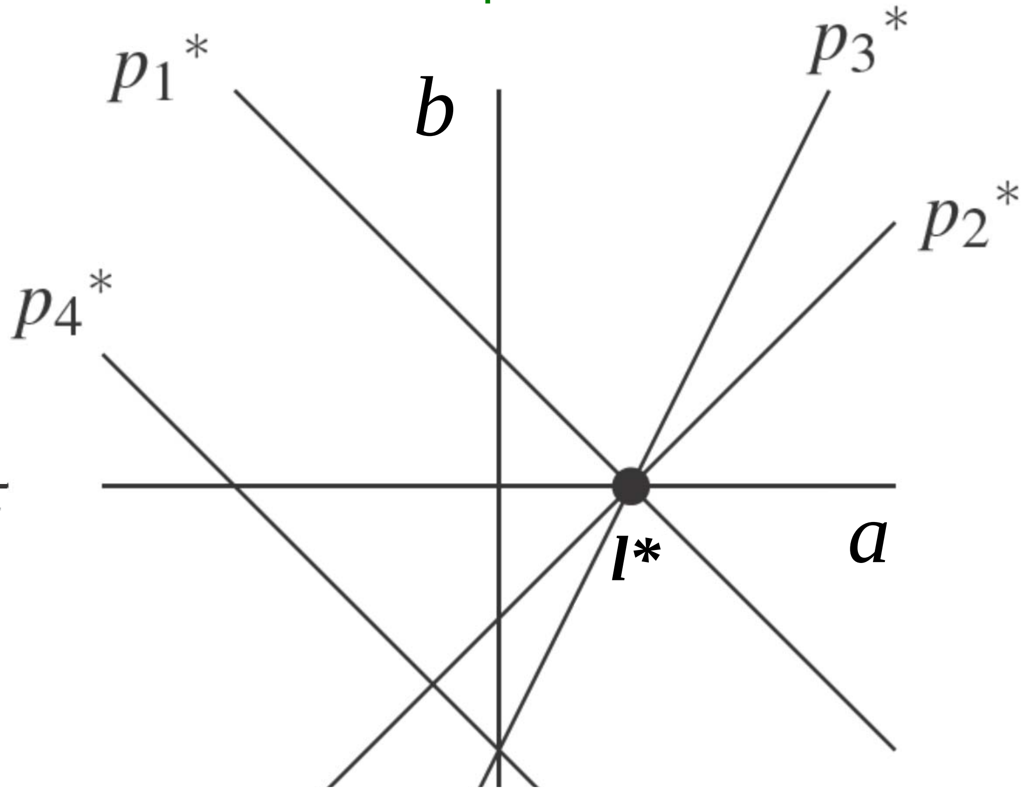
  - Primal plane – plane with coordinates x, y
  - Dual plane* – plane with coordinates a, b

Duality of lines and points in the plane

Primal plane

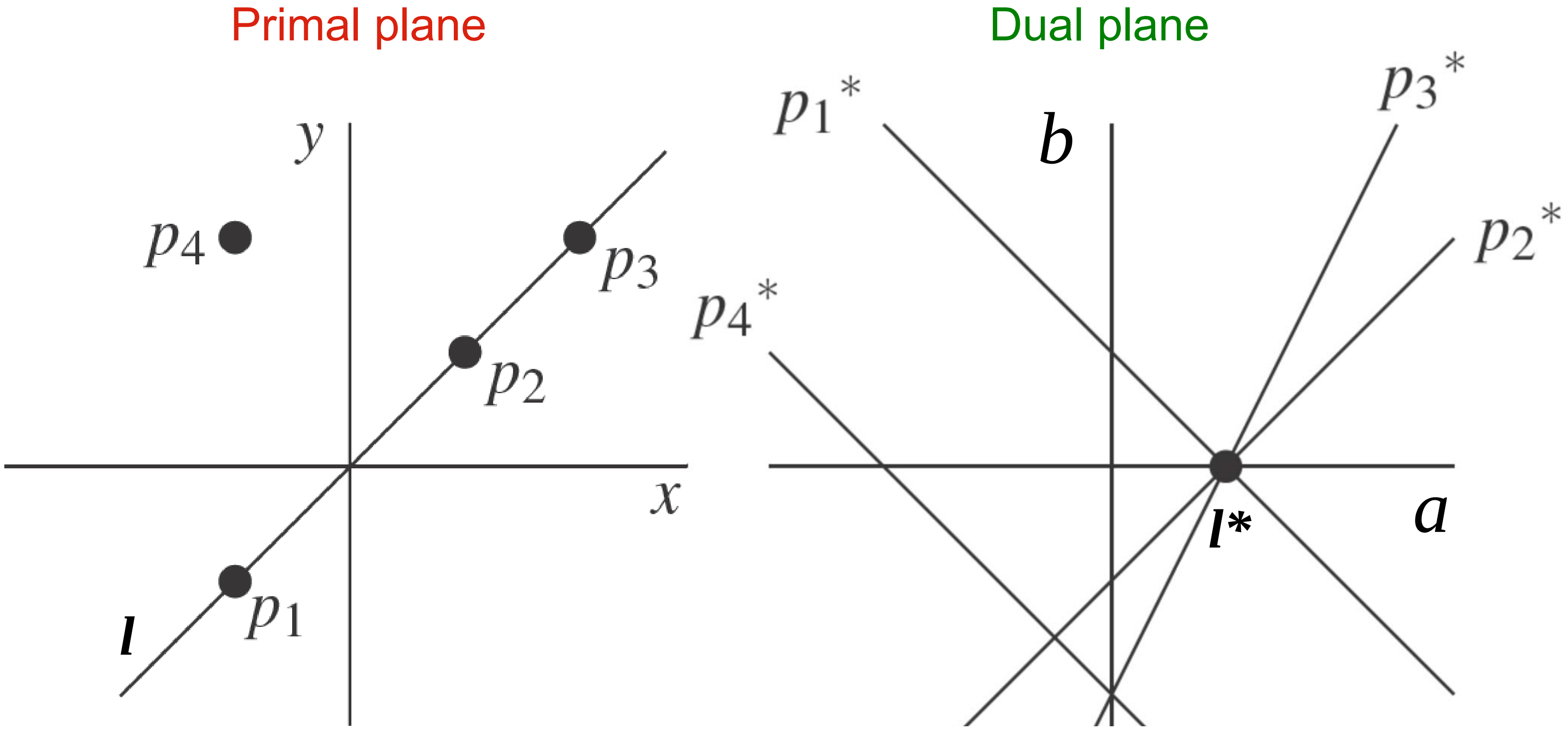


Dual plane



[Berg]

Duality of lines and points in the plane

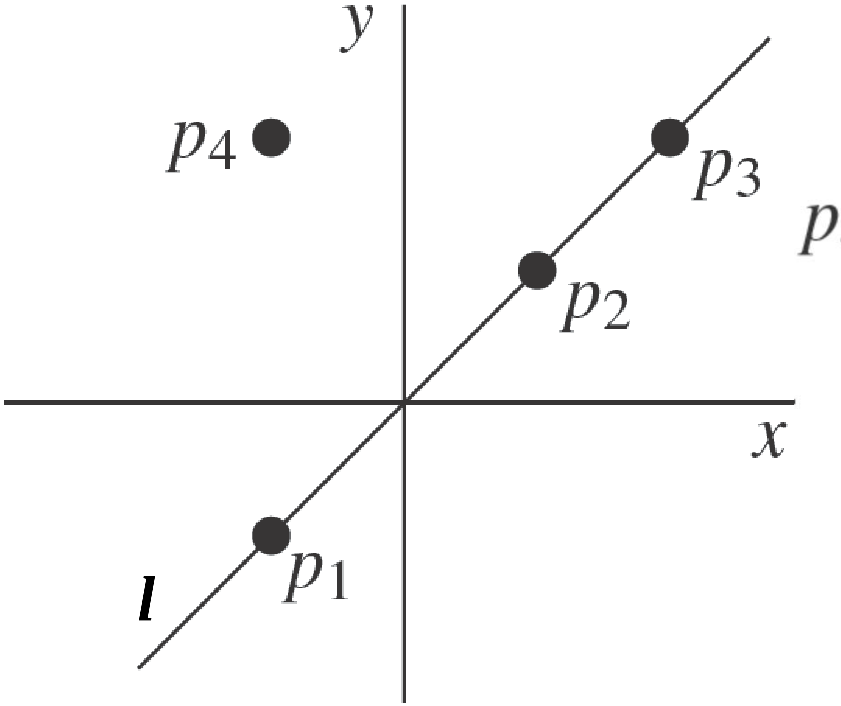


point p = [p_x, p_y]

[Berg]

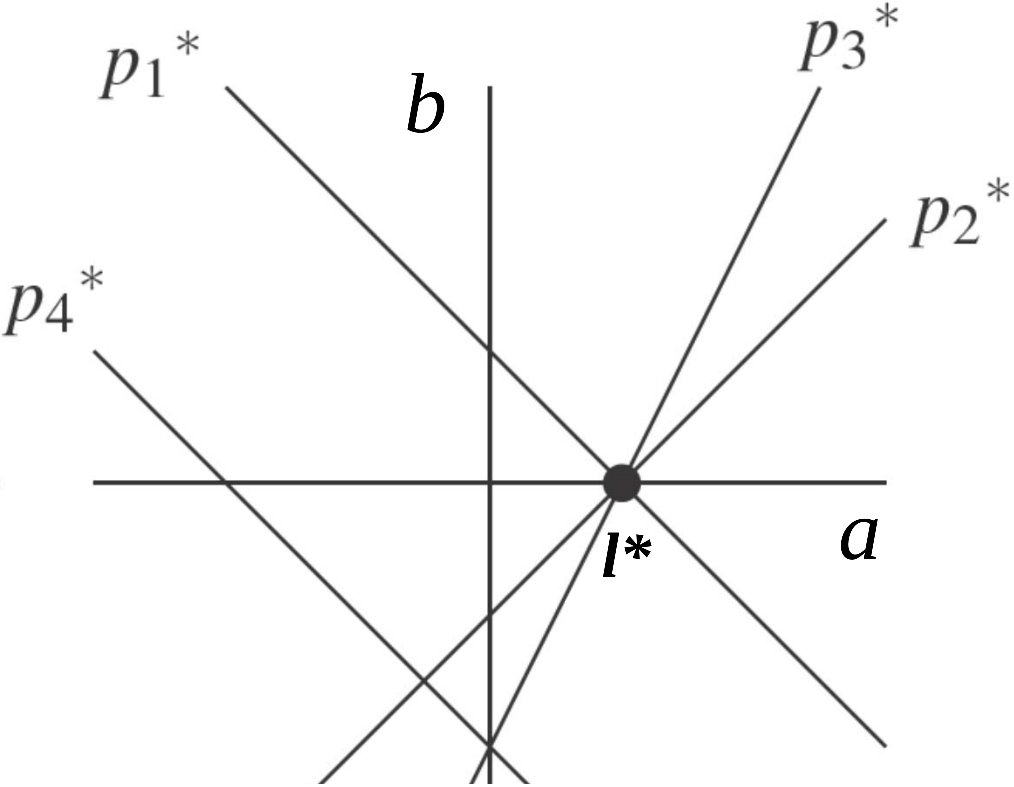
Duality of lines and points in the plane

Primal plane



point p = [p_x, p_y]

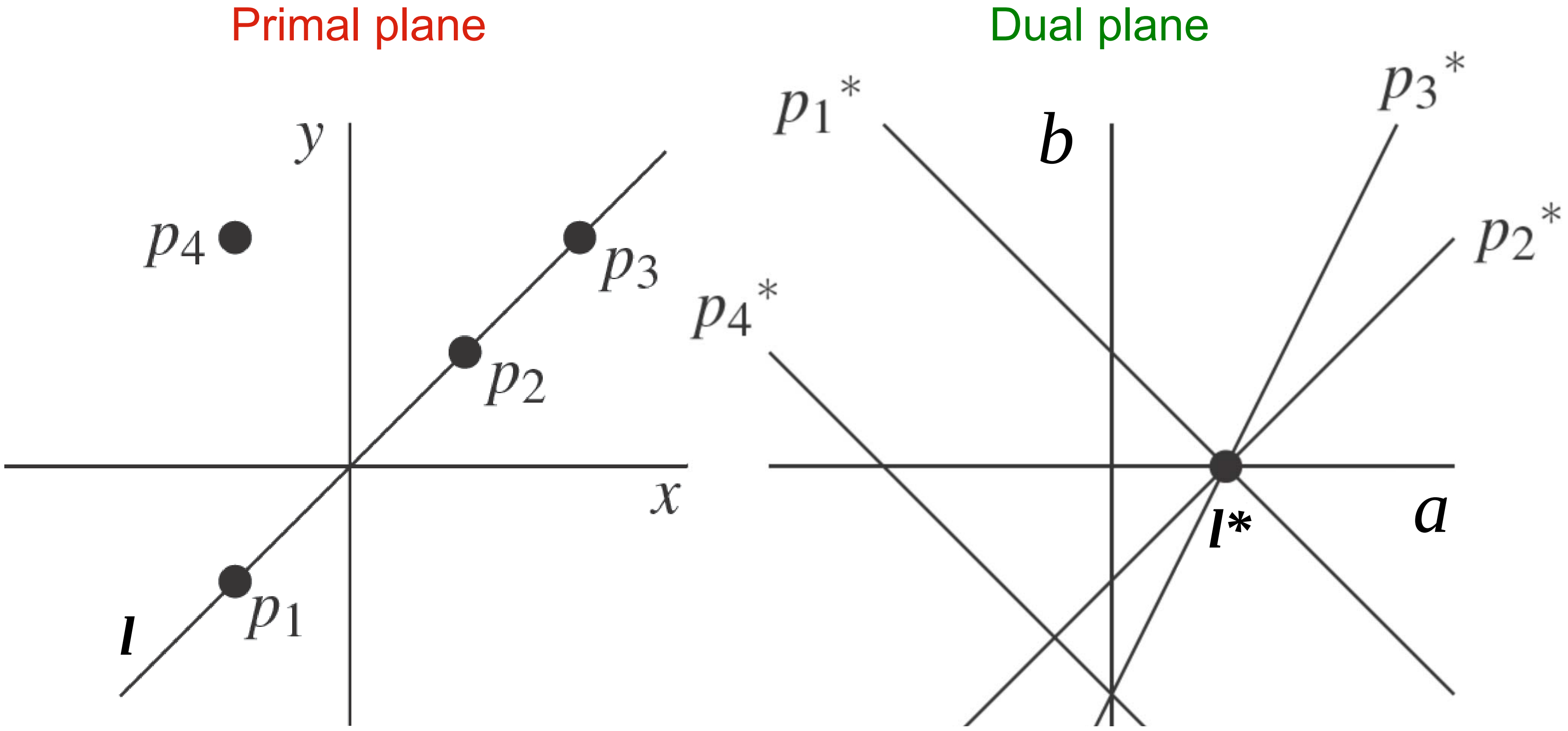
Dual plane



line p* := (b = $p_x a - p_y$)

[Berg]

Duality of lines and points in the plane



point p = [p_x, p_y]

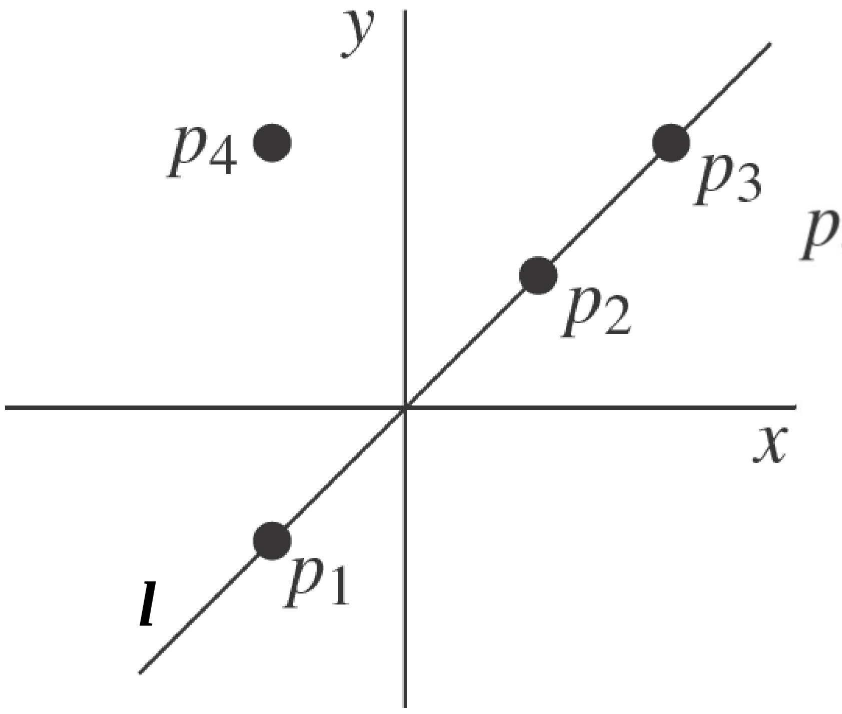
line ***l*** := (y = ax + b)

line p* := (b = $p_x a - p_y$)

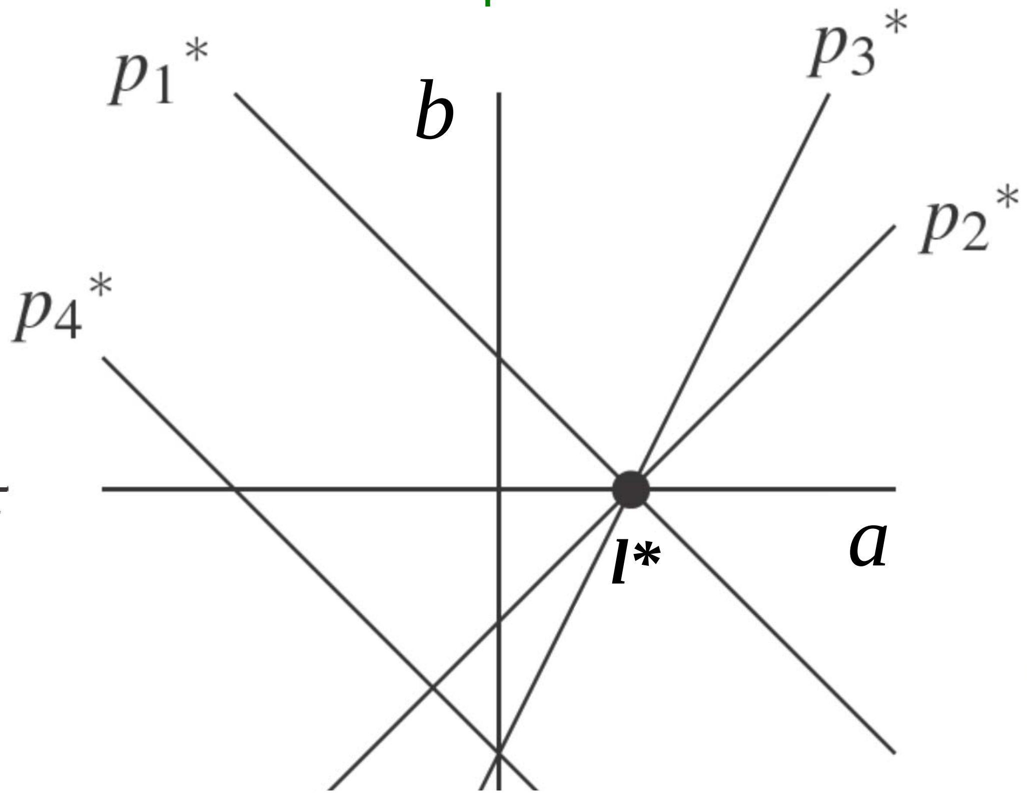
[Berg]

Duality of lines and points in the plane

Primal plane



Dual plane



point p = [p_x, p_y]

line *l* := (y = ax + b)

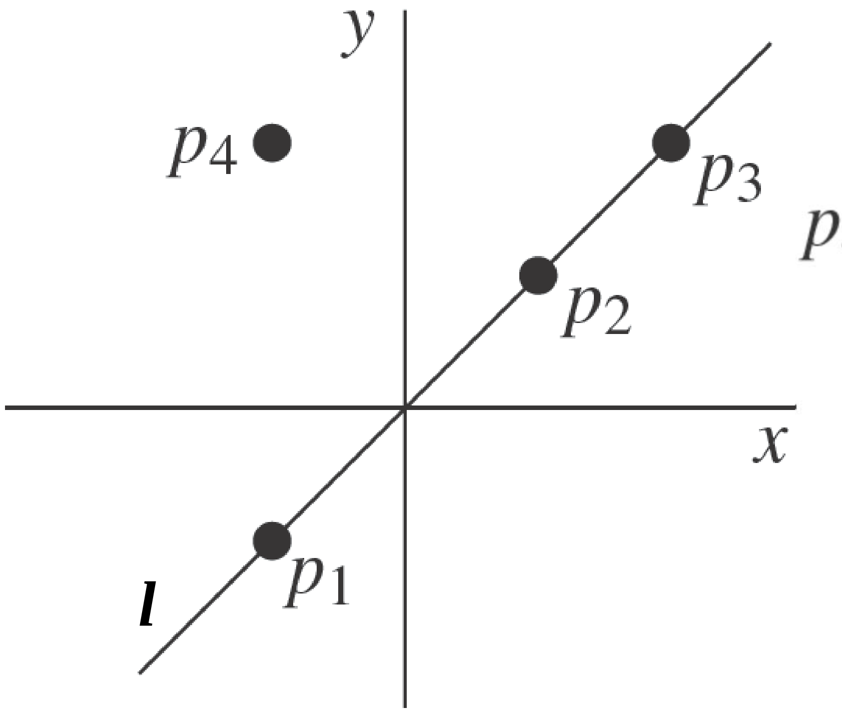
line p* := (b = $p_x a - p_y$)

Point *l** = [a, – b]

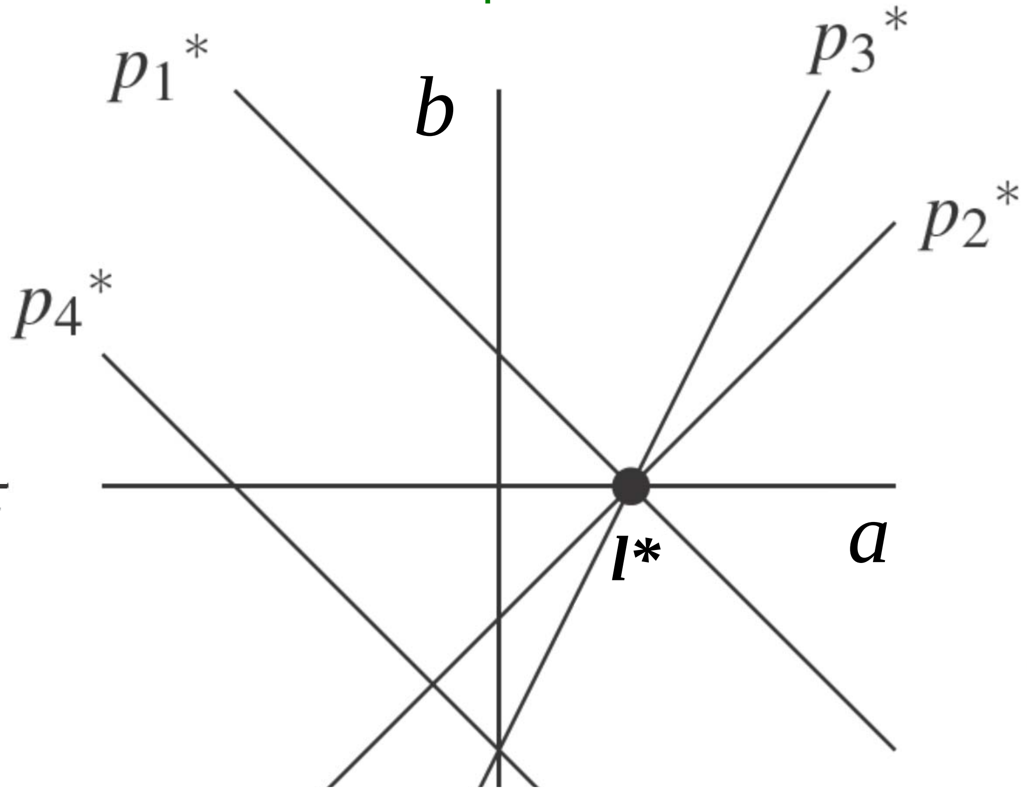
[Berg]

Duality of lines and points in the plane

Primal plane



Dual plane



point p = [p_x, p_y]

~~line l := (y = ax + b)~~

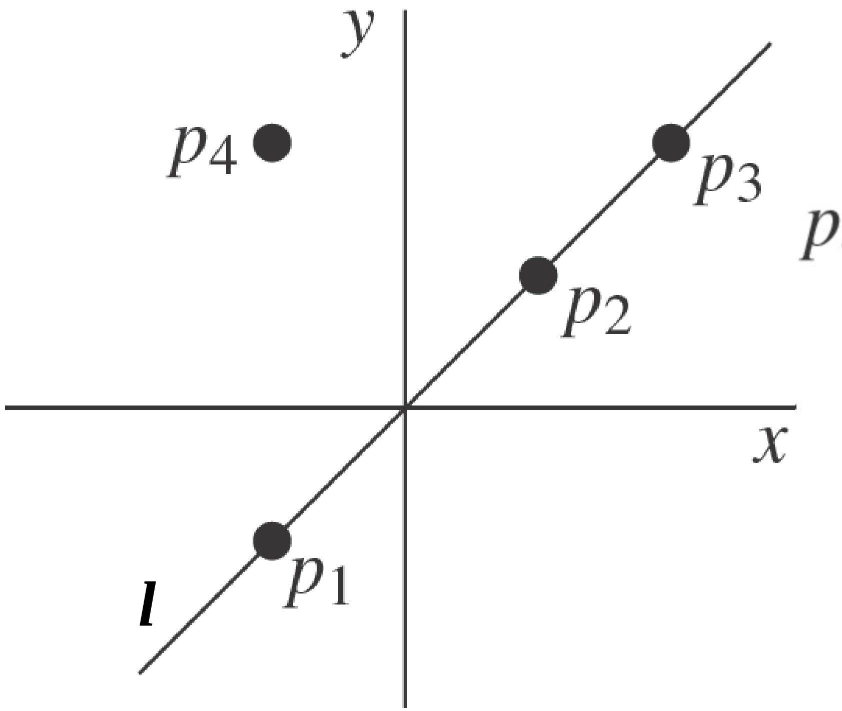
line p* := (b = $p_x a - p_y$)

~~Point l^* = [a, – b]~~

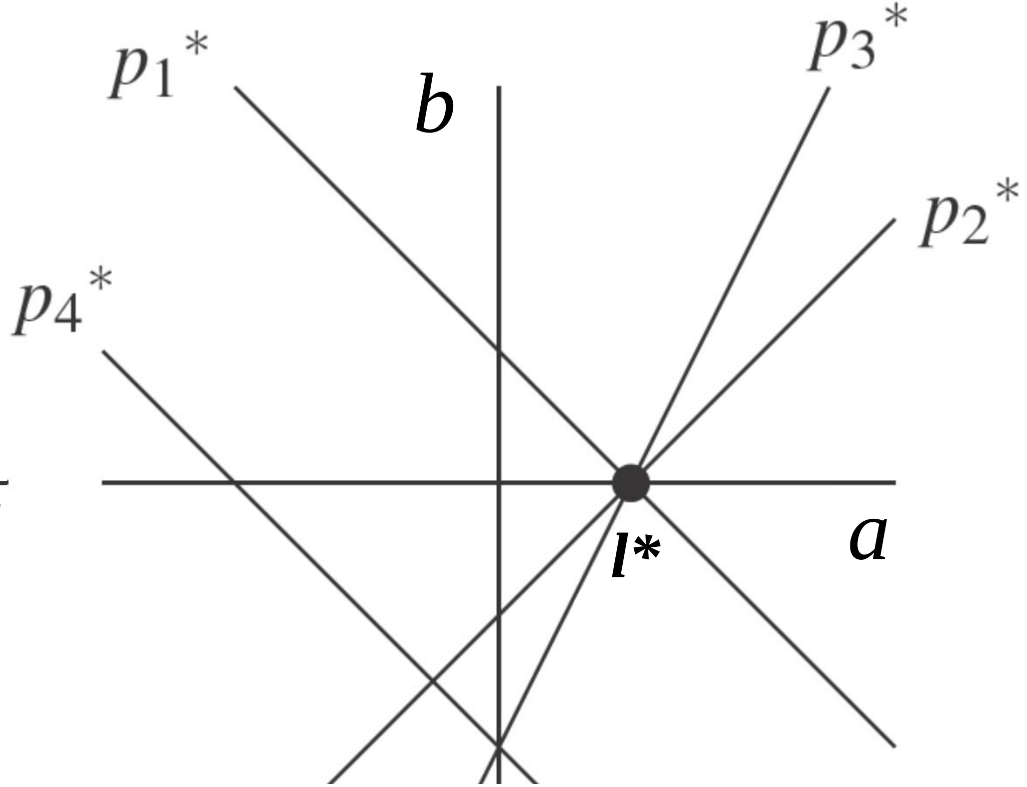
[Berg]

Duality of lines and points in the plane

Primal plane



Dual plane



point p = [p_x, p_y]

~~line ***l*** := (y = ax + b)~~

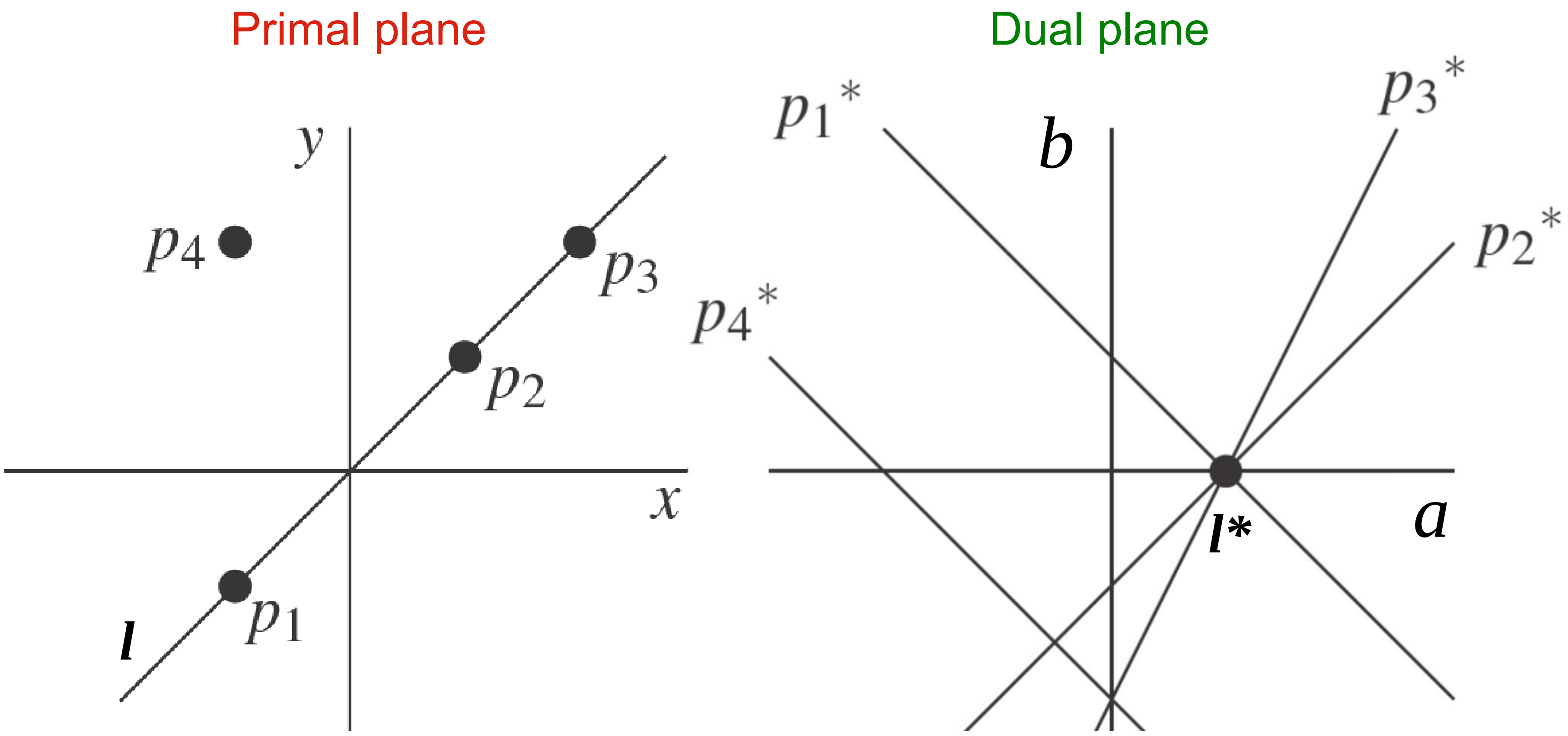
line ***l*** := (y = ax – b)

line p* := (b = $p_x a – p_y$)

~~Point ***l**** = [a, – b]~~

[Berg]

Duality of lines and points in the plane


Primal plane

Dual plane

point p = [p_x, p_y]

~~line ***l*** := (y = ax + b)~~

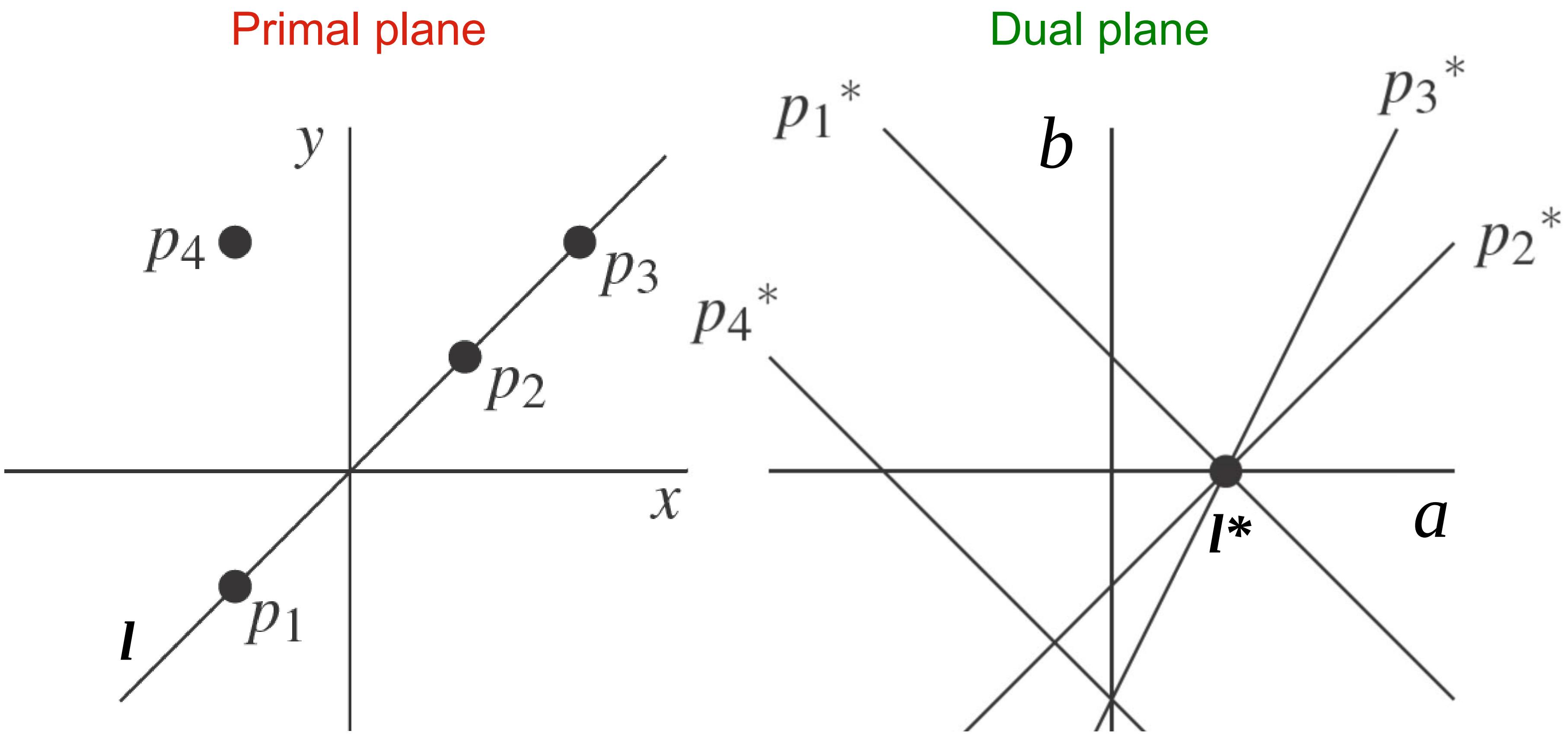
line ***l*** := (y = ax – b)

line p* := (b = $p_x a - p_y$)

~~Point ***l**** = [a, – b]~~

[Berg]

Duality of lines and points in the plane



Primal plane

Dual plane

point p = [p_x, p_y]

~~line l := (y = ax + b)~~

line l := (y = ax – b)

line p* := (b = $p_x a - p_y$)

~~Point l^* = [a, – b]~~

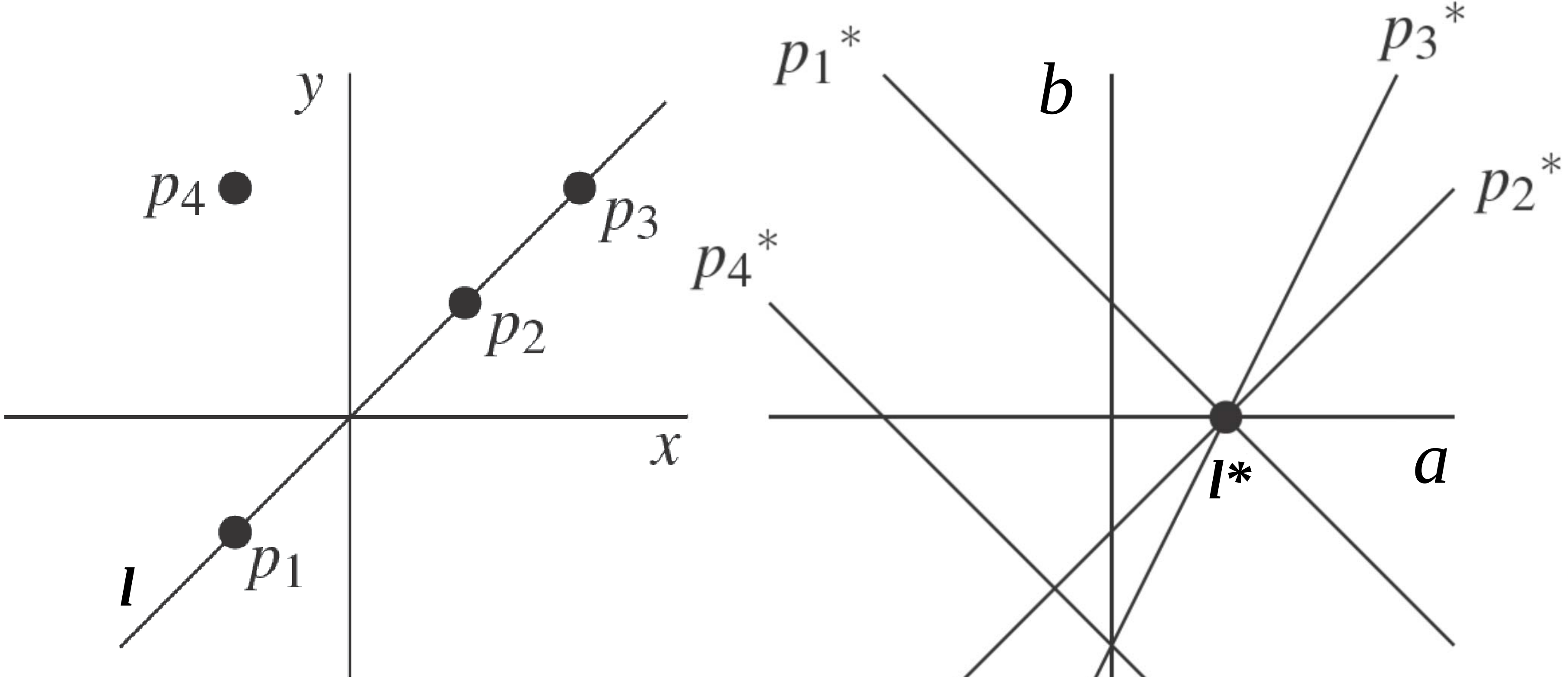
Point l^* = [a, b]

[Berg]

Duality of lines and points in the plane

Primal plane

Dual plane



point p = [p_x, p_y] line p* := (b = $p_x a - p_y$)

~~line $\boldsymbol{l}$:= (y = ax + b)~~ ~~Point $\boldsymbol{l}^*$ = [a, – b]~~

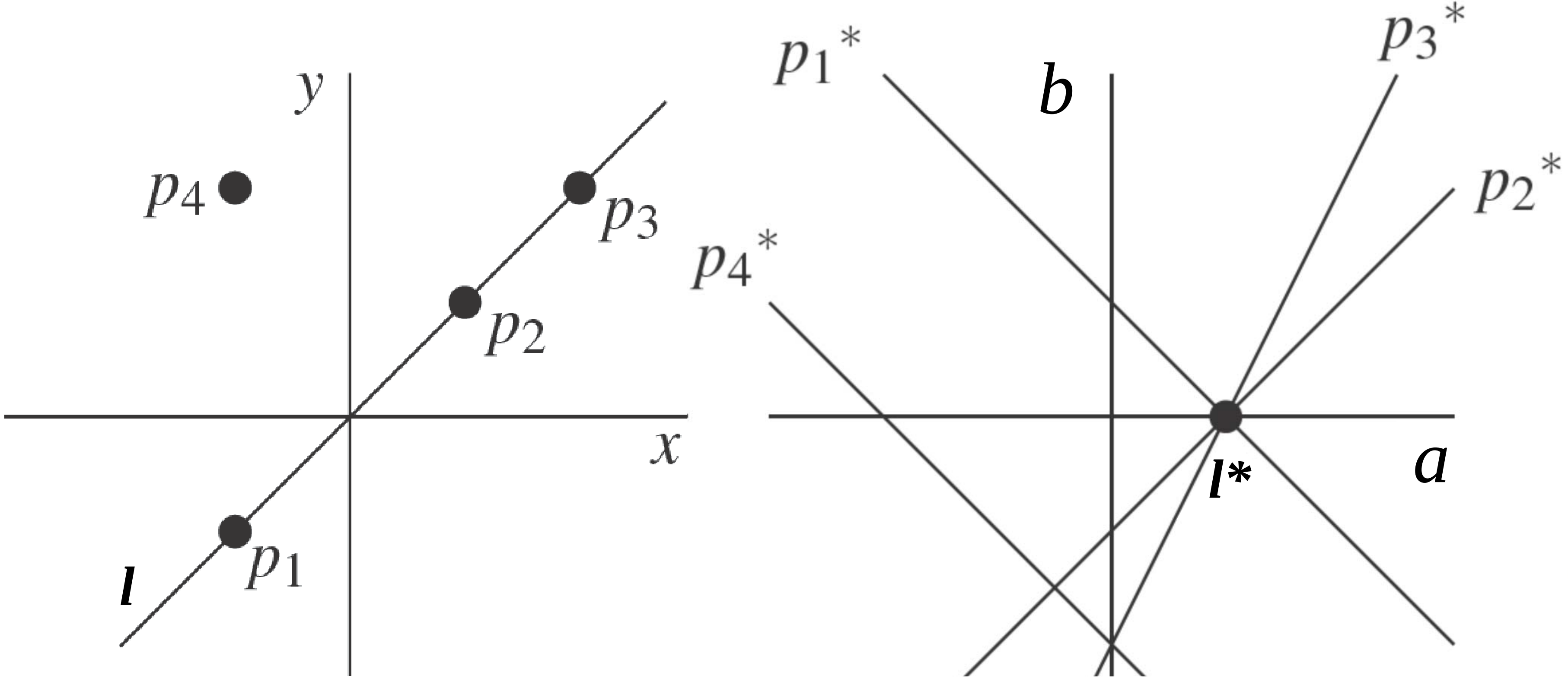
line $\boldsymbol{l}$:= (y = ax – b) Point $\boldsymbol{l}^*$ = [a, b]

[Berg]

Duality of lines and points in the plane

Primal plane

Dual plane



point p = [p_x, p_y]

~~line $\boldsymbol{l}$:= (y = ax + b)~~

line $\boldsymbol{l}$:= (y = ax – b)

line p* := (b = $p_x a$ – p_y)

~~Point $\boldsymbol{l}$* = [a, – b]~~

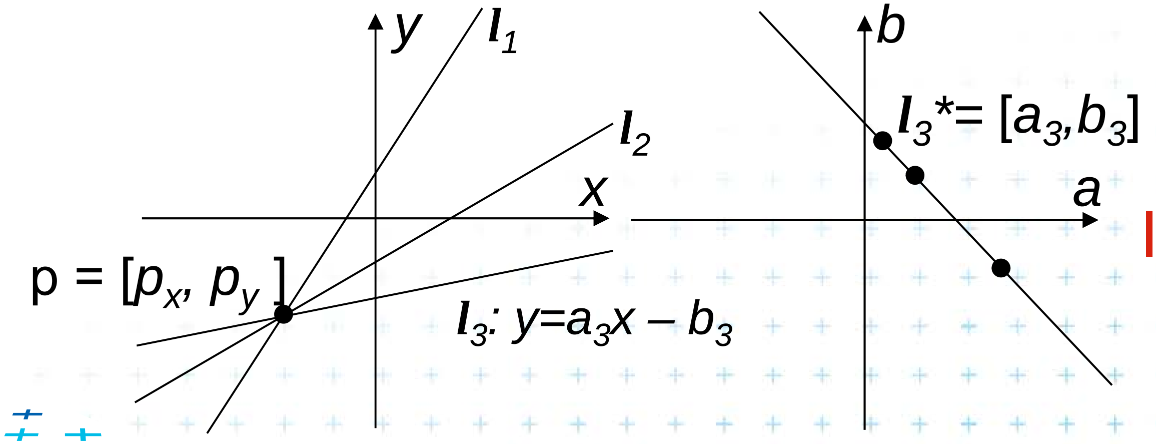
Point $\boldsymbol{l}$* = [a, b]

[Berg]

Same form => It is convenient to negate *b* in the line equation

Why is *b* negated in the line equation?

- In primal plane, consider
  - point $p = [\, p_x, p_y\,]$ and
  - set of non-vertical lines $\boldsymbol{l}_i : y = a_i x - b_i$
    passing through p satisfy the equation $p_y = a_i p_x - b_i$
    (each line with different constants a_i, b_i)
- In dual plane, these lines transform to collinear points $\{ l_i^* = [a_i, b_i] : b_i = p_x a_i - p_y \}$



Same form =>

It is convenient to negate *b* in the line equation

If *b not* negated in the line equation…

Lines $\boldsymbol{l}_i$ have equartion $\boldsymbol{l}_i$:$y = a_i x - b_i$ *OR* $y = a_i x + b_i$

Passing through point p = [p_x, p_y] :

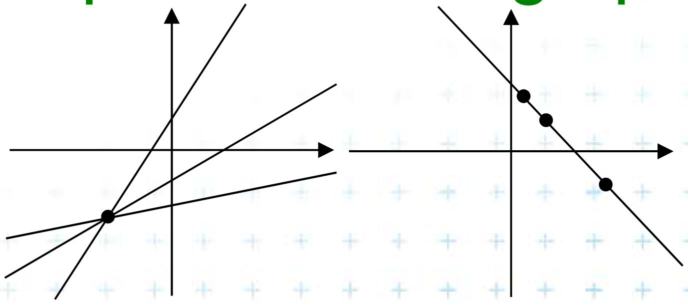
- With minus
  - equation $\boldsymbol{l}_i$: $p_y = a_i p_x - b_i$
    dual points $\{\boldsymbol{l}_i{}^* = [a_i,b_i] : b_i = p_x a_i - p_y\}$ … same form
- With plus
  - equation $\boldsymbol{l}_i$: $p_y = a_i p_x + b_i$
    dual $\{\boldsymbol{l}_i{}^* = [a_i,b_i] : b_i = - p_x a_i + p_y\}$ … different form

Properties of points and lines duality

Incidence is preserved

- Point *p* is incident to the line *l* in primal plane
  iff
  point *l** is incident to the line *p** in the dual plane.

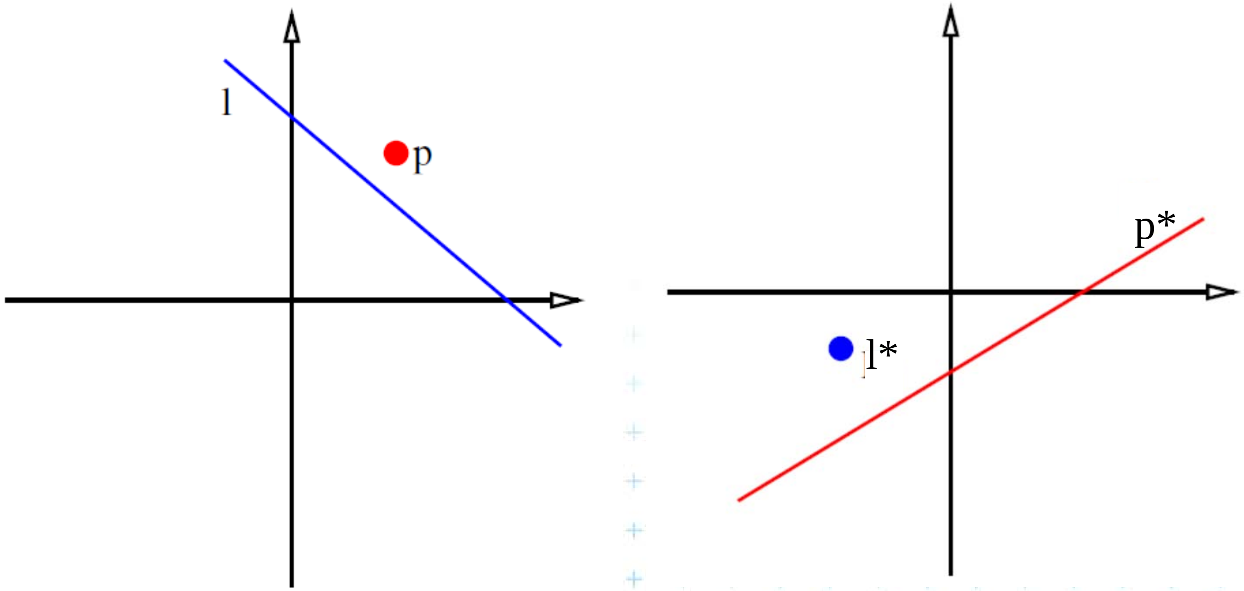
- Lines l_1, l_2 intersects at point *p*
  iff
  line *p** passes through points l_1*, l_2*.



Properties of points and lines duality

But order is reversed

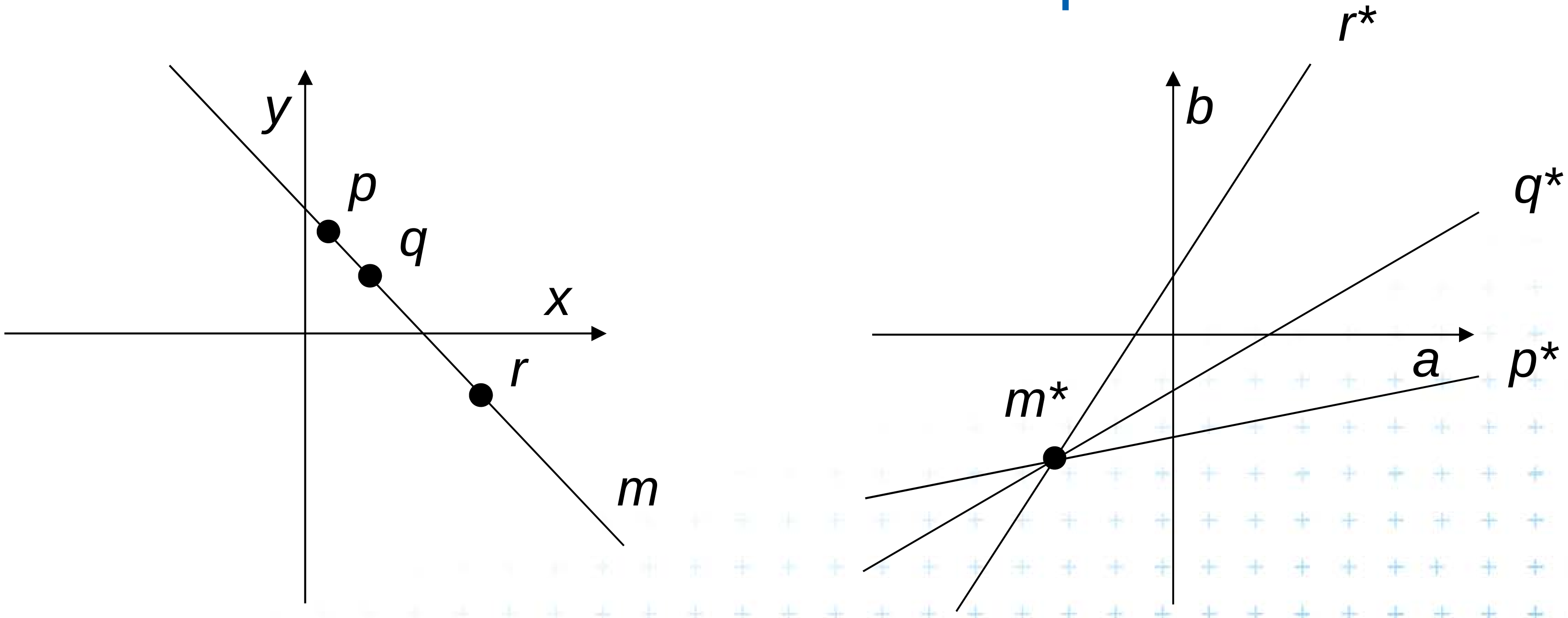
- Point *p* lies above (below) line *l* in the primal plane **iff**
  line *p** passes below (above) point *l** in the dual plane Or said order is preserved: … **iff** Point *l** lies above (below) line *p**



Properties of points and lines duality

Collinearity

- Points are collinear in the primal plane **iff** their dual lines intersect in a common point

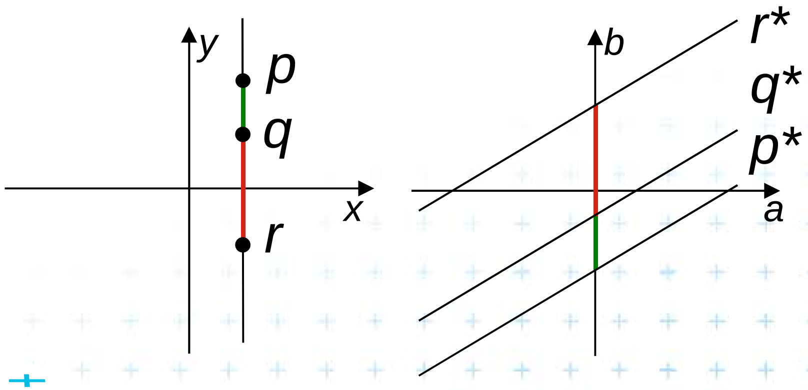


- This does not hold for points on vertical line

Handling of vertical lines

- Dual transform is undefined for vertical lines
  - Points with the same *x* coordinate dualize to lines with the same slope (parallel lines) and therefore
  - These dual lines do not intersect (as should for collinear points)
  - Vertical line through these points does not dualize to an intersection point
  - For detection of vertically collinear points use other method - O(*n*) vertical lines -> O(*n*^{2}) brute force 3|| lines s.

-> O(*n*) after O(*n* log *n*) sorting by *x*

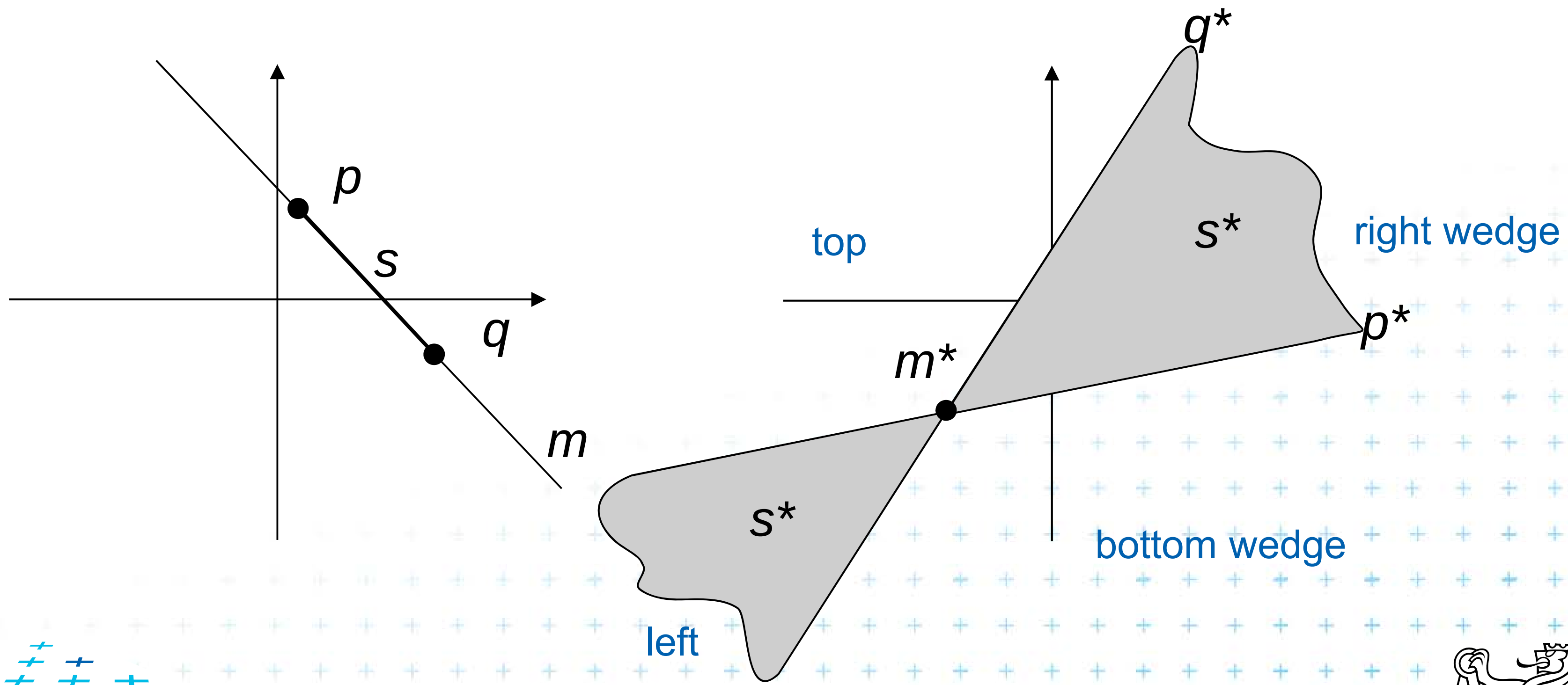


Vertical distances of such duals are "preserved". For $p_x = q_x$

vertDist(q^*_b, p^*_b) = $p_y - q_y$

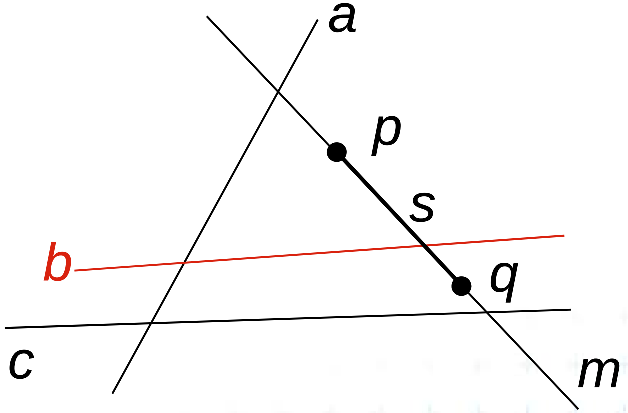
2. Duality of line segments

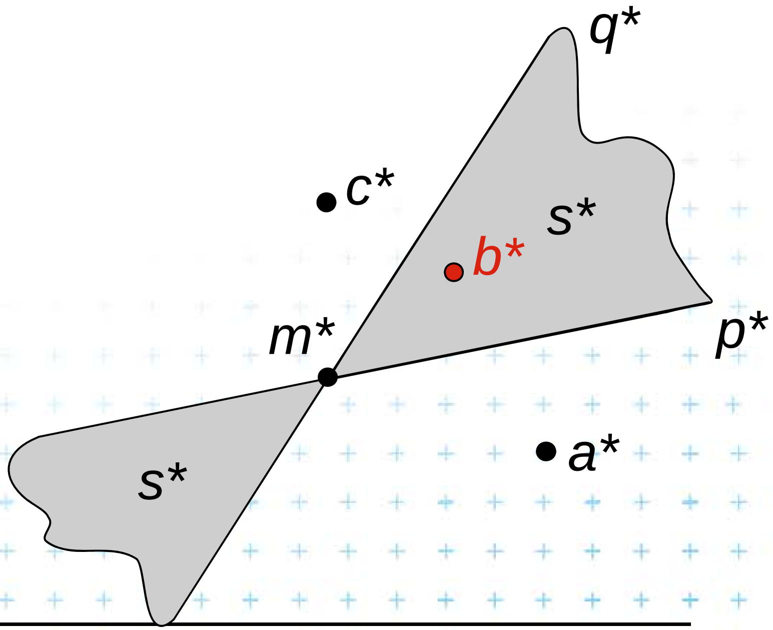
- Line segment *s*
  - = set of collinear points $\xrightarrow{\text{dual}}$ set of lines passing one point
  - – union of these lines is a (left-right) double wedge *s** Dvojitý klín



Intersection of line and line segment

- Line *b* intersects line segment *s*
  - if point *b** lays in the double wedge *s**,
    i.e., between the duals *p*,q** of segment endpoints *p,q*
  - point *p* lies above line *b* and *q* lies below line *b*
  - point b* lies above line *p** and *b** lies below line *q**



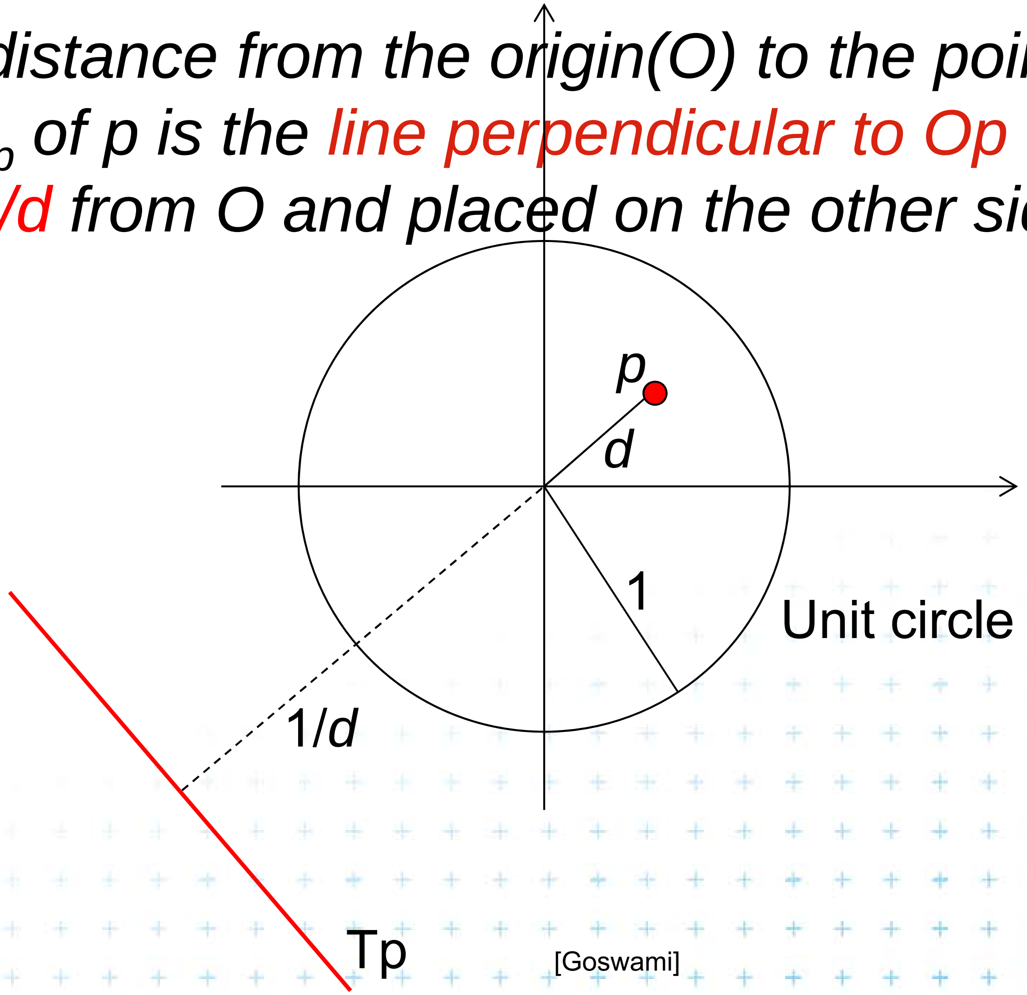


3. Polar duality (Polarity)

- Another example of point-line duality
- In 2D: Point $p = (p_x, p_y)$ in the primal plane corresponds to a line T_p with equation $ax + by = 1$ in the dual plane and vice versa $p_x x + p_y y = 1$
- In dD: Point p is taken as a radius-vector (starts in origin O). The dot product $(p\ .\ \mathbf{x}) = 1$ defines a polar hyperplane $p^* = T_p = \{\ \mathbf{x} \in R^d : (p\ .\ \mathbf{x}) = 1\ \}$
- Used in theory of polytopes

Polar duality (Polarity)

- *Geometrically in 2D, this means that*
  - *if d is the distance from the origin(O) to the point p, the dual T_p of p is the line perpendicular to Op at distance 1/d from O and placed on the other side of O.*



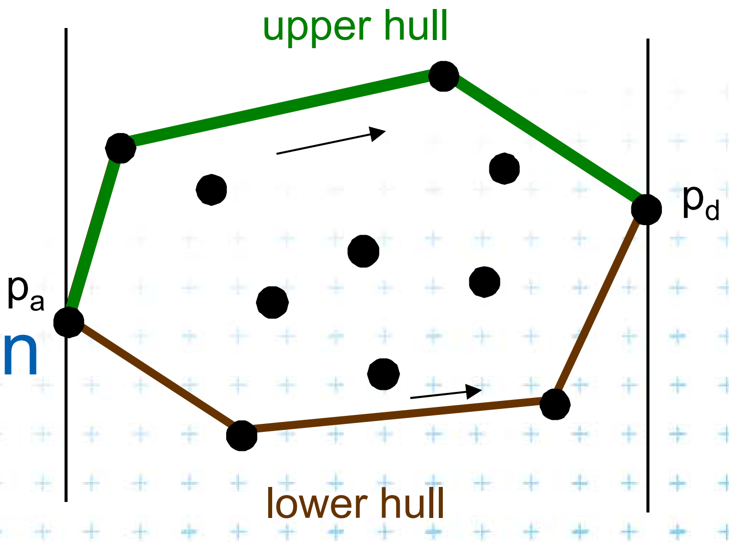
[Goswami]

4. Convex hull using duality – definitions

- An optimal algorithm
- Let *P* be the given set of *n* points in the plane.
- Let $p_a \in P$ be the point with smallest x-coordinate
- Let $p_d \in P$ be the point with largest x-coordinate
  Both p_a and $p_d \in CH(P)$

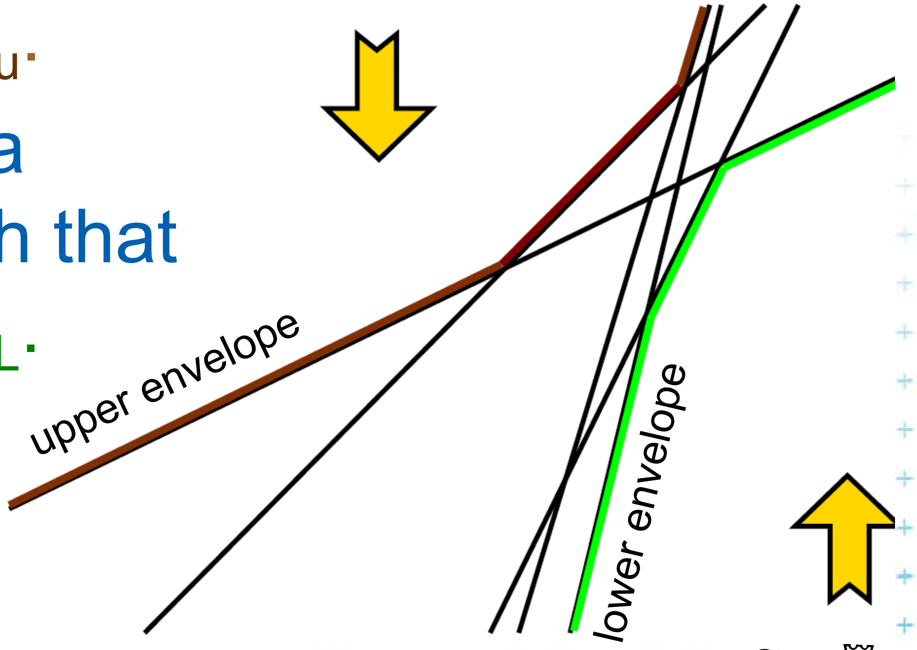
Upper hull = CW polygonal chain
$p_a, \ldots, p_d$ along the hull

Lower hull = CCW polygonal chain
$p_a, \ldots, p_d$ along the hull



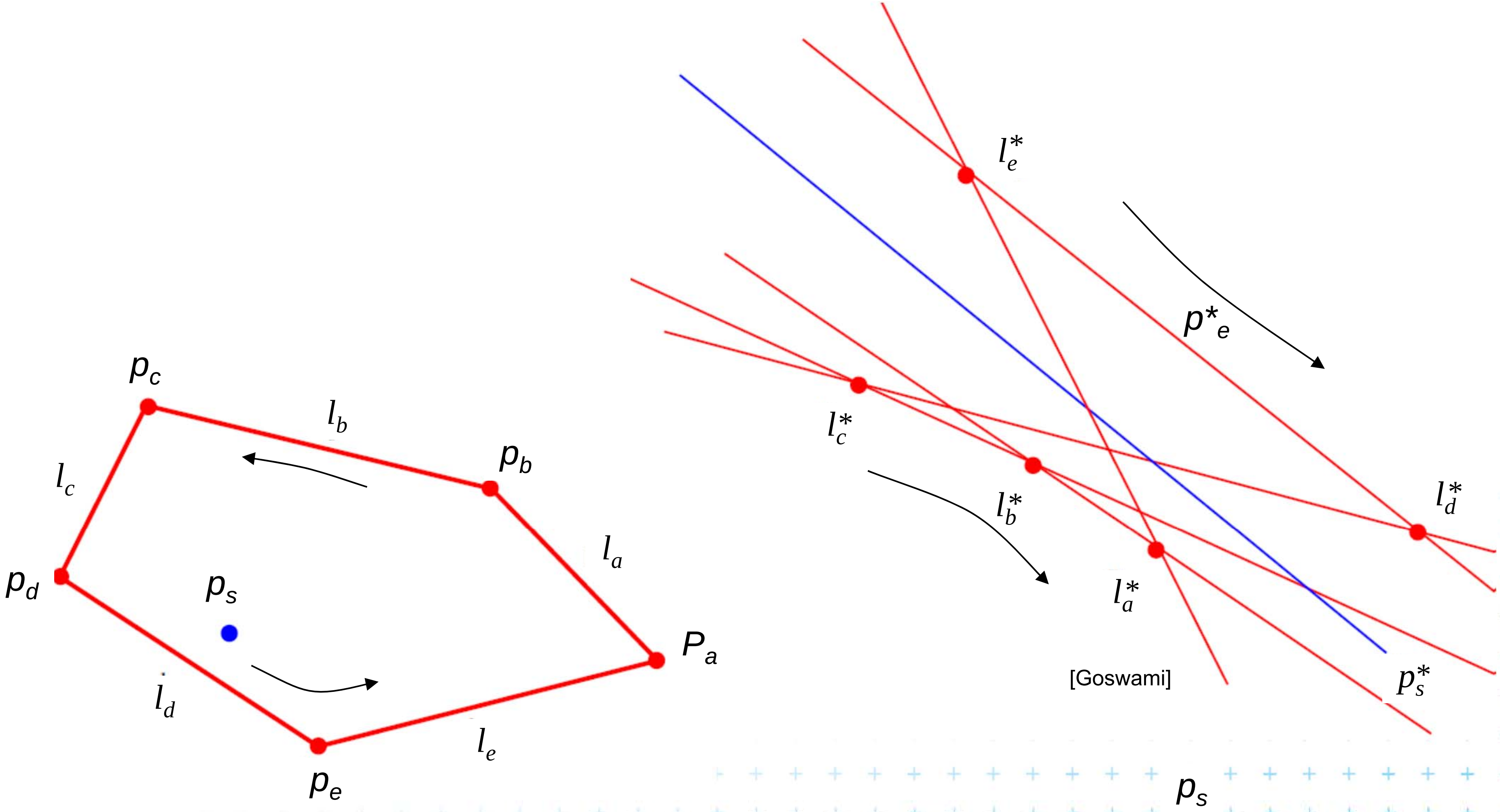
Definitions

- Let L be a set of lines in the plane
- The upper envelope is a polygonal chain E_u such that no line $l \in L$ is above E_u.
- The lower envelope is a polygonal chain E_L such that no line $\boldsymbol{l} \in \boldsymbol{L}$ is below E_L.



[Goswami]

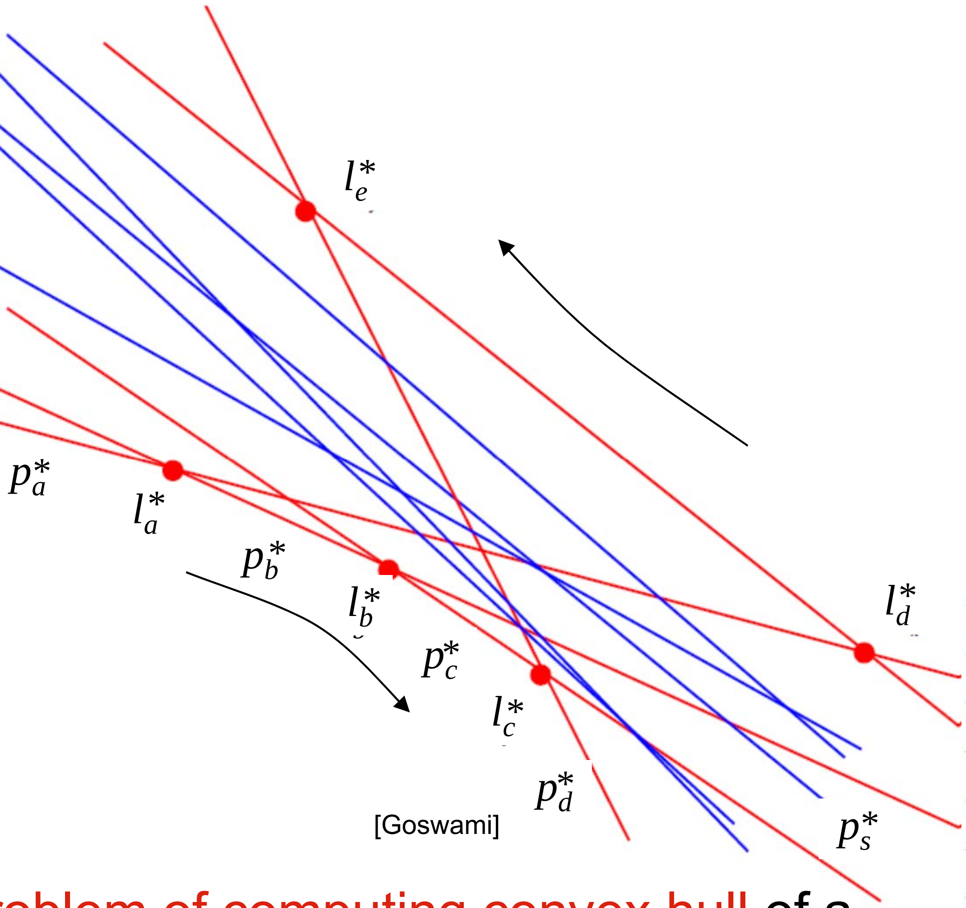
Connection between Hull and Envelope



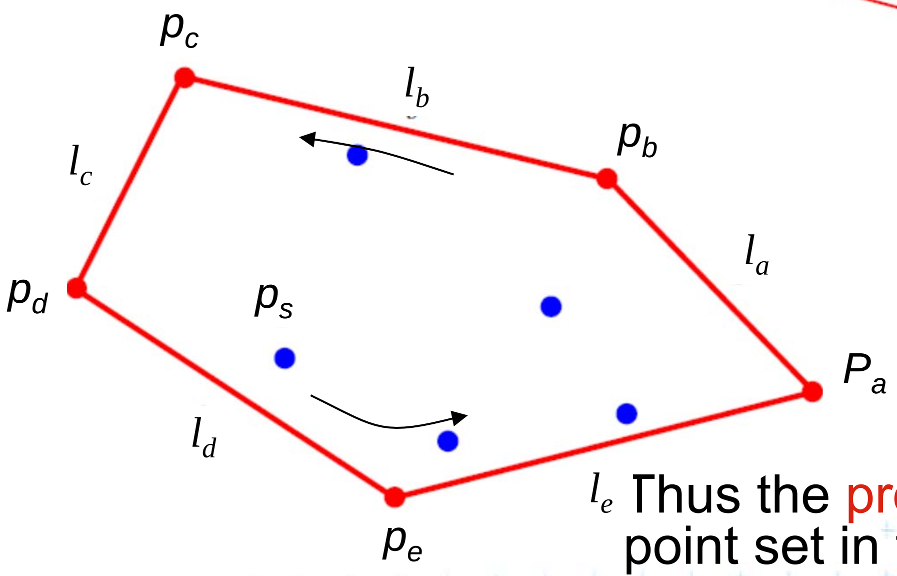
[Goswami]

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.



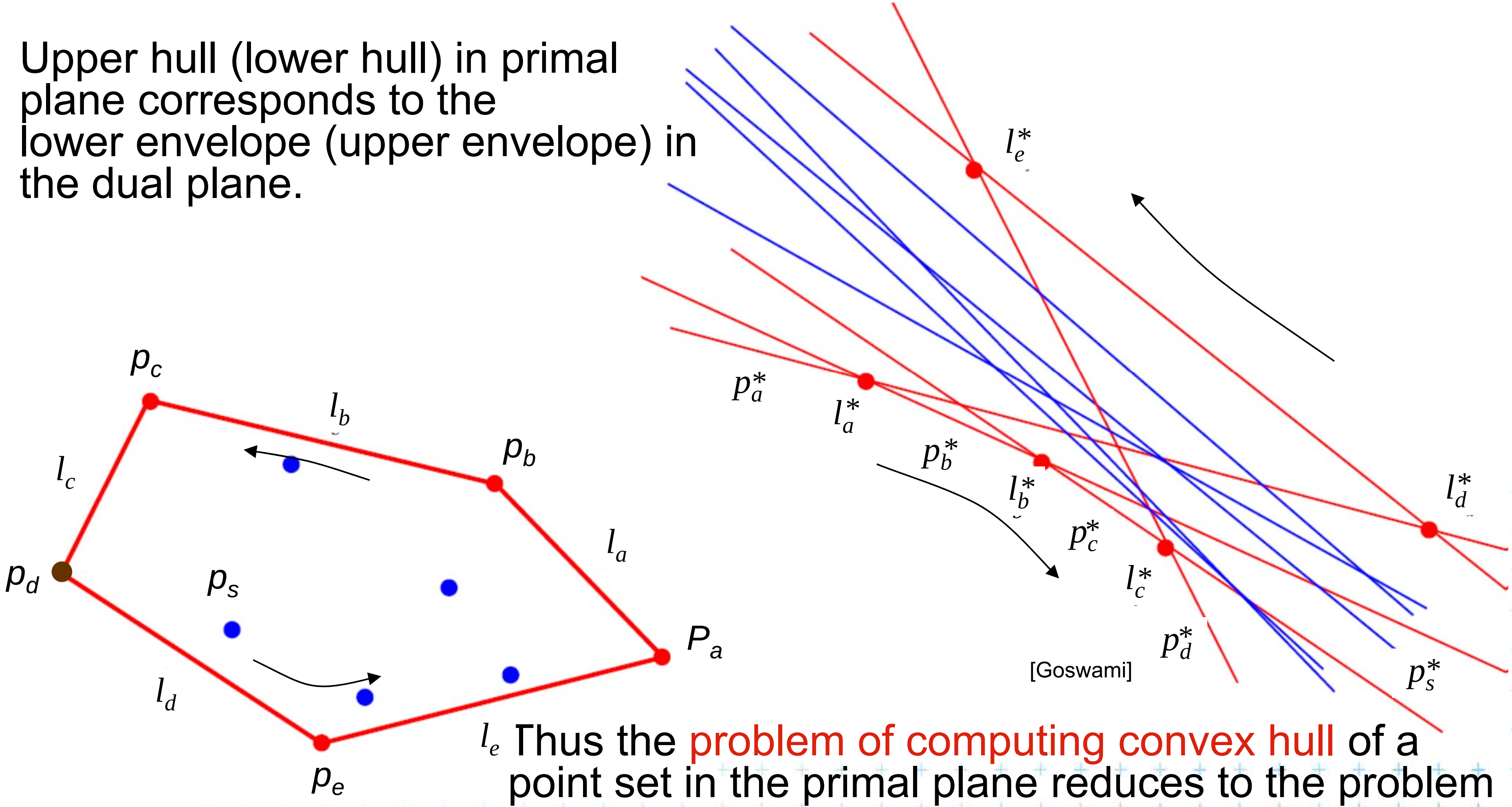
[Goswami]



Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.

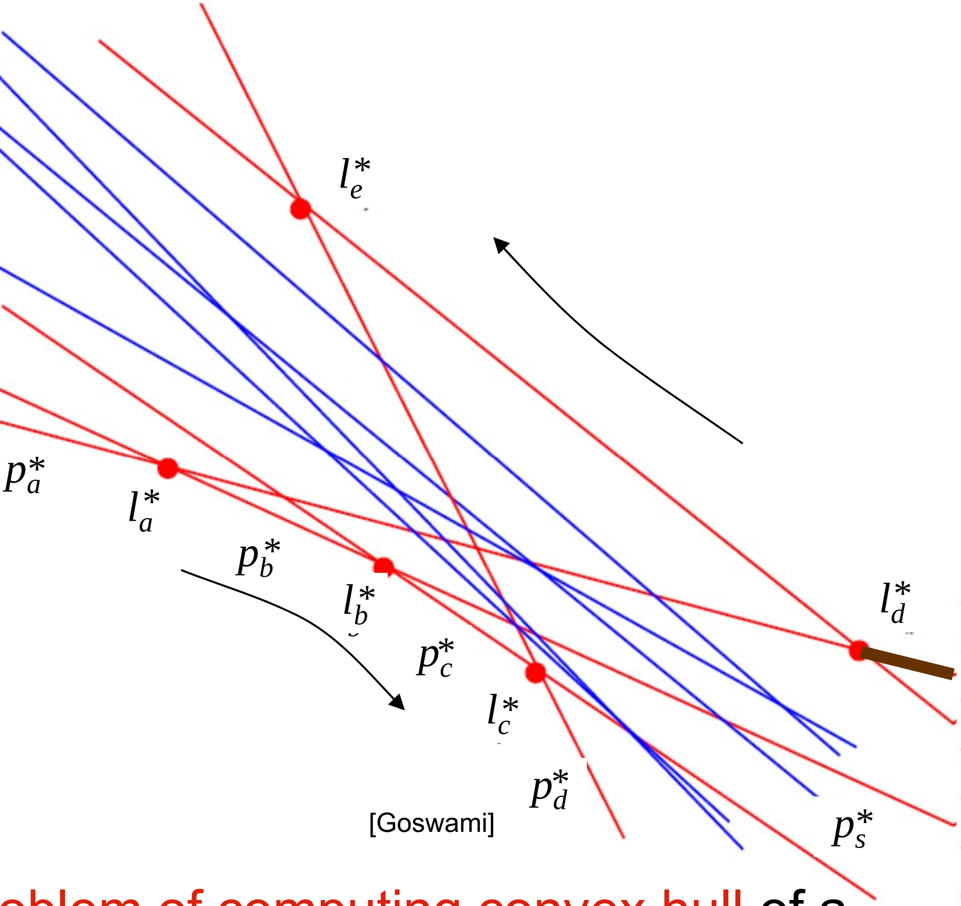


[Goswami]

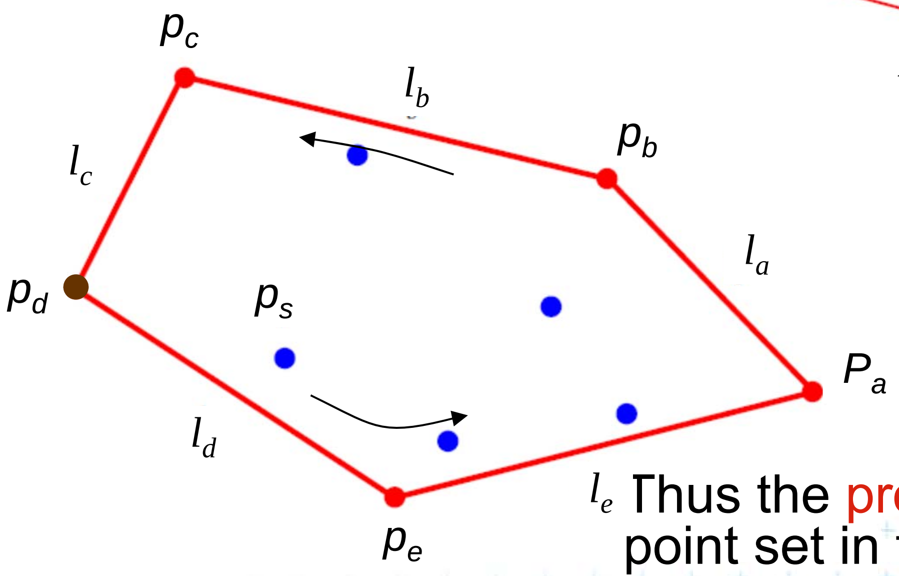
Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.



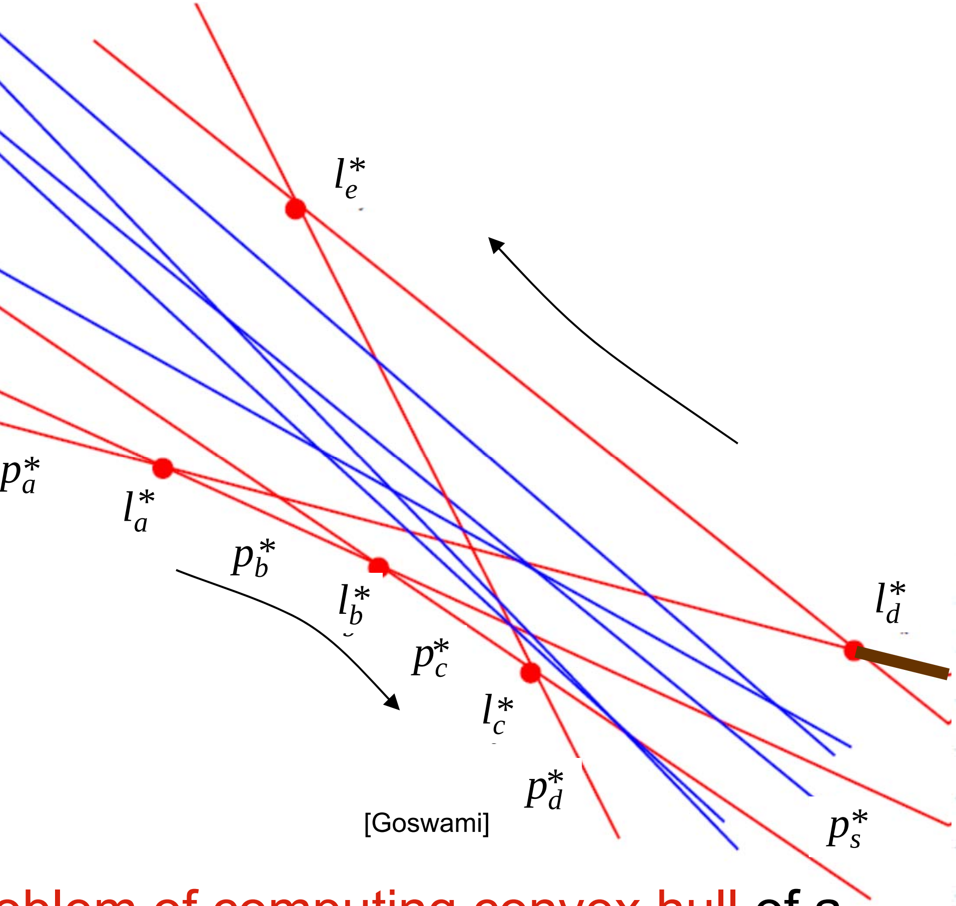
[Goswami]



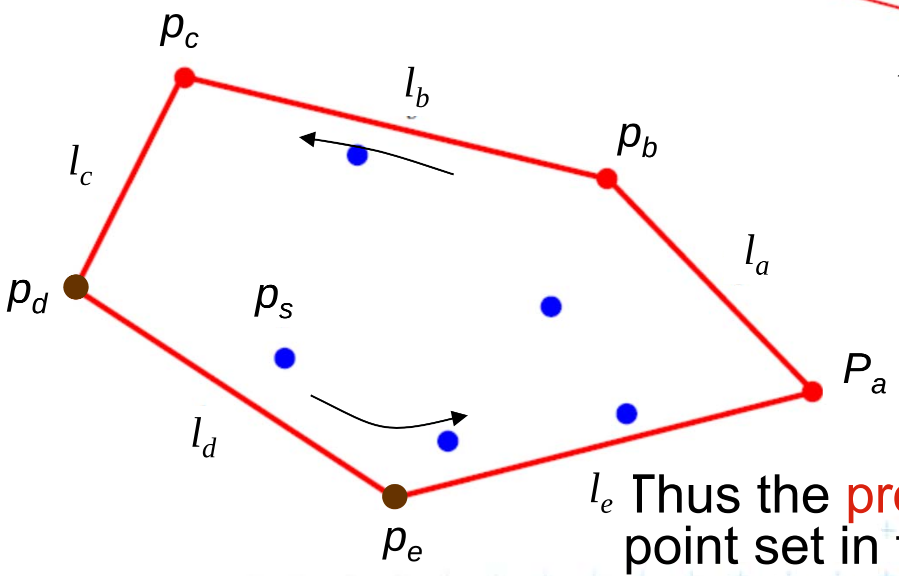
Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.



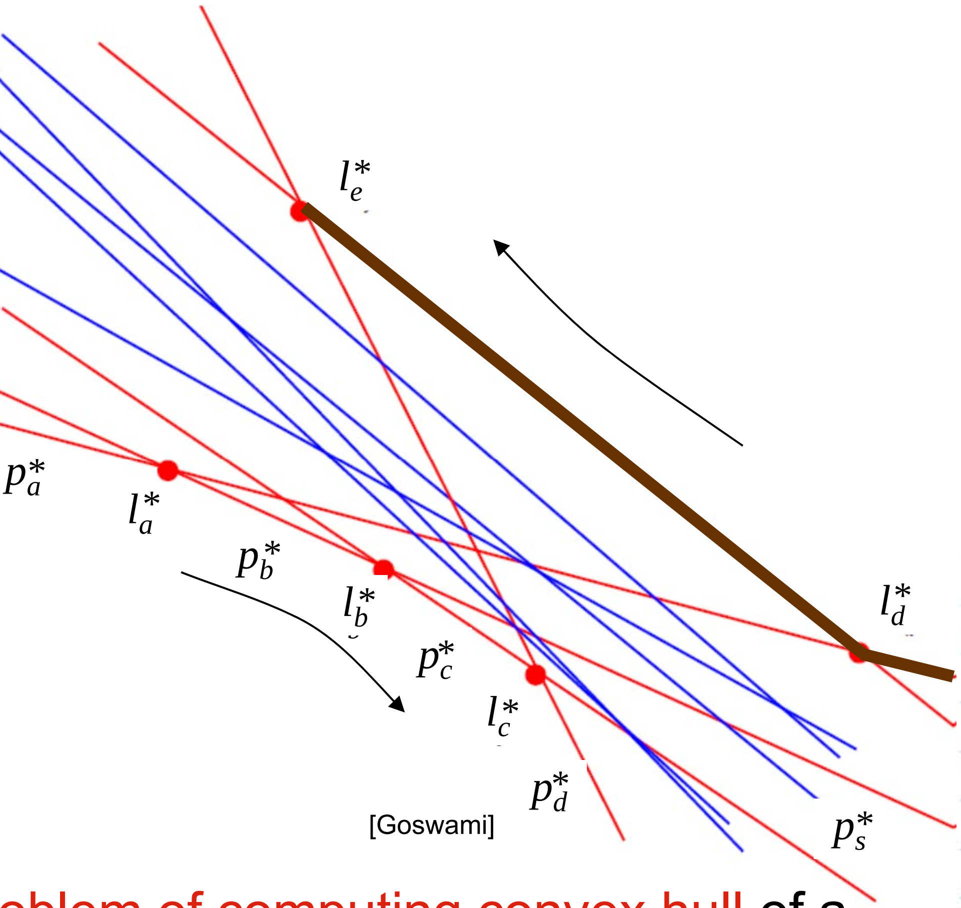
[Goswami]



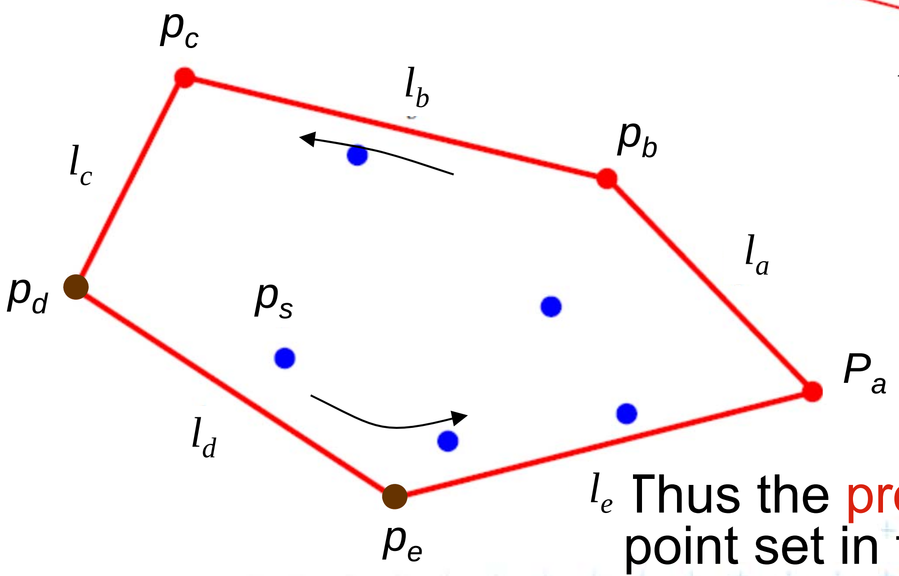
Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.



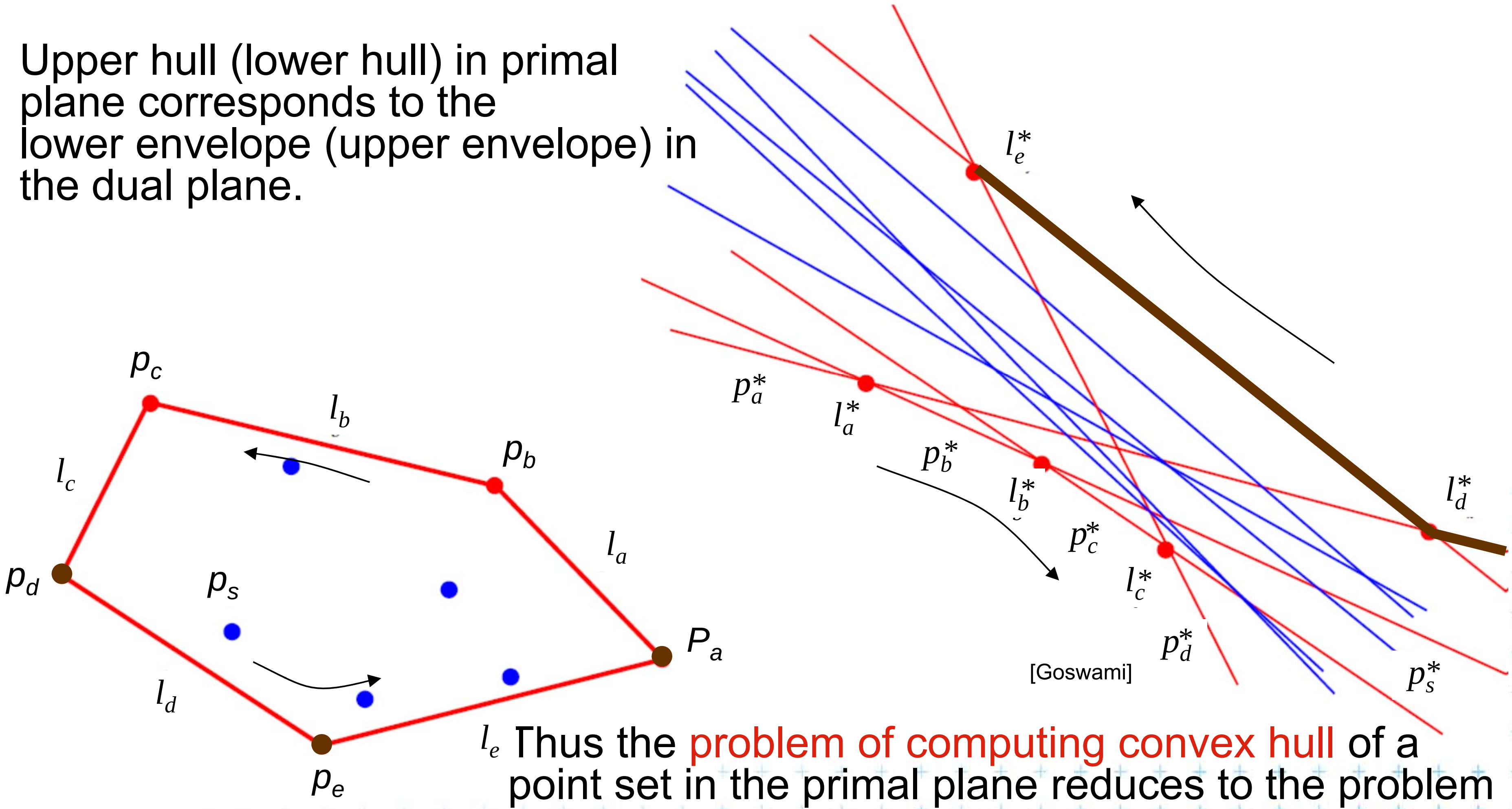
[Goswami]



Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.

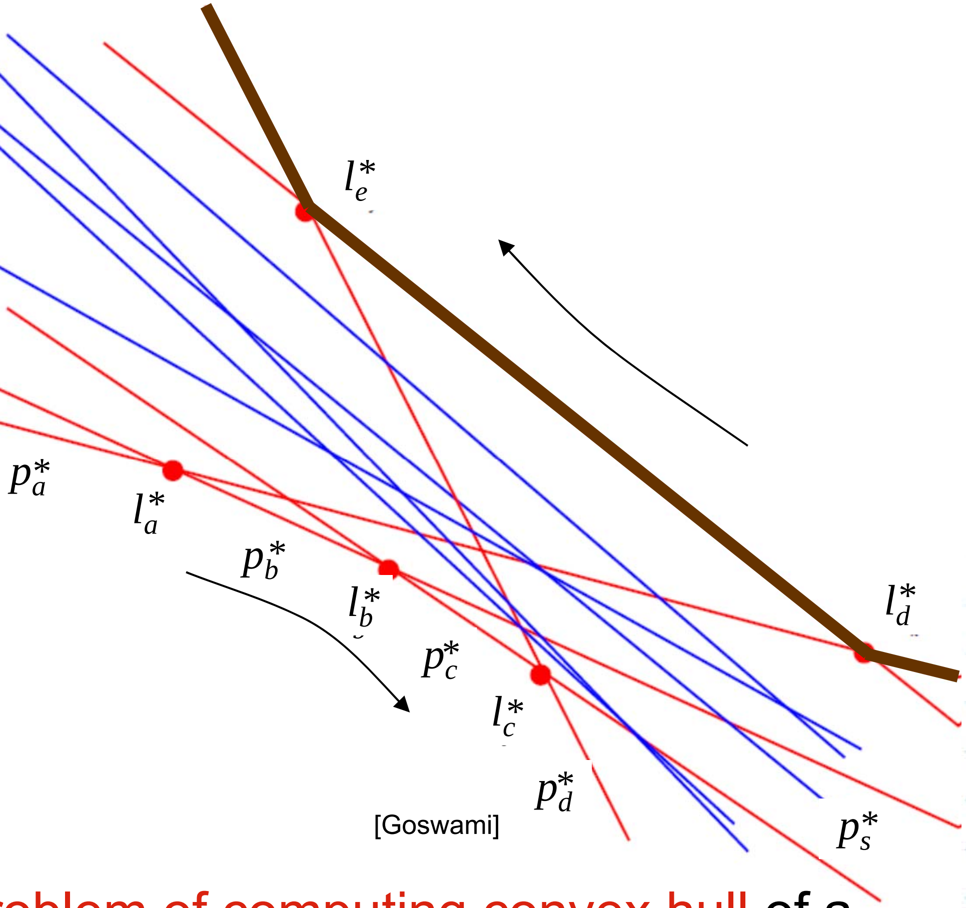


[Goswami]

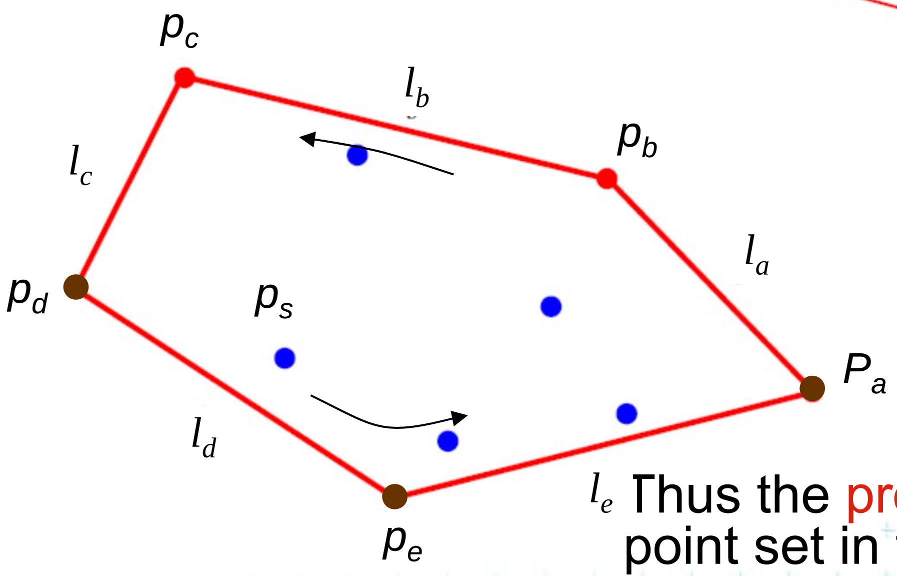
Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.



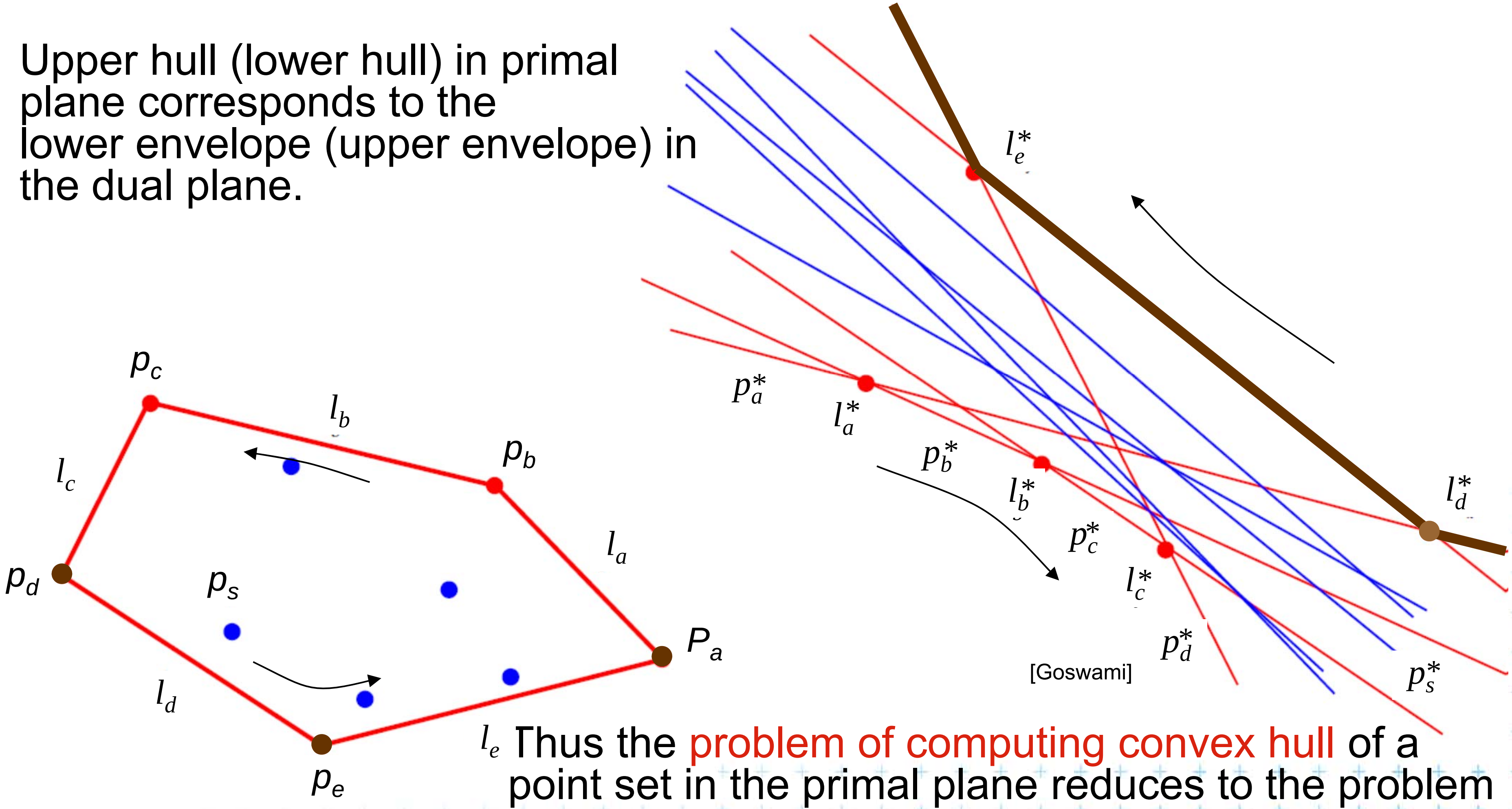
[Goswami]



l_e Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.

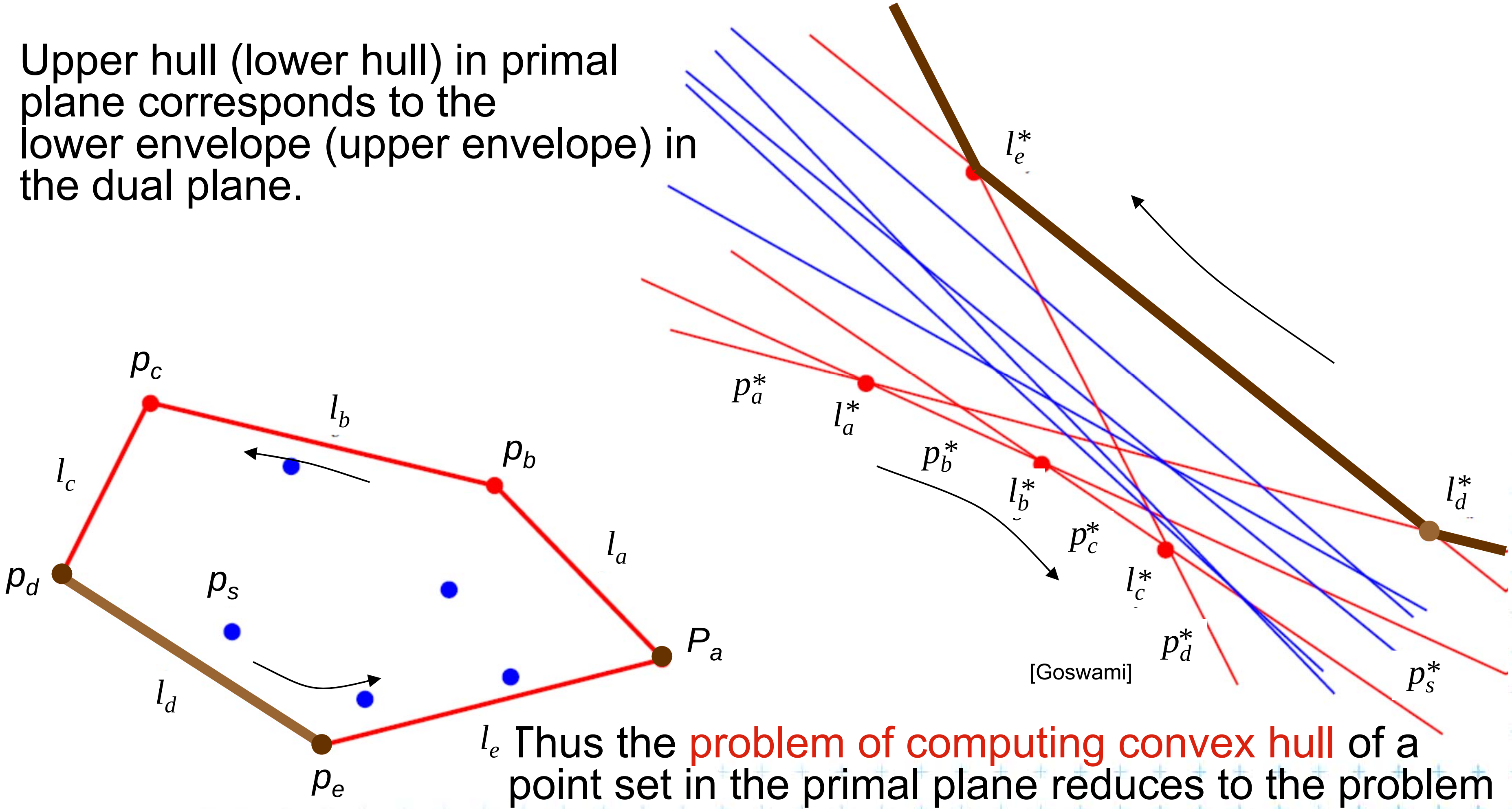


[Goswami]

Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.

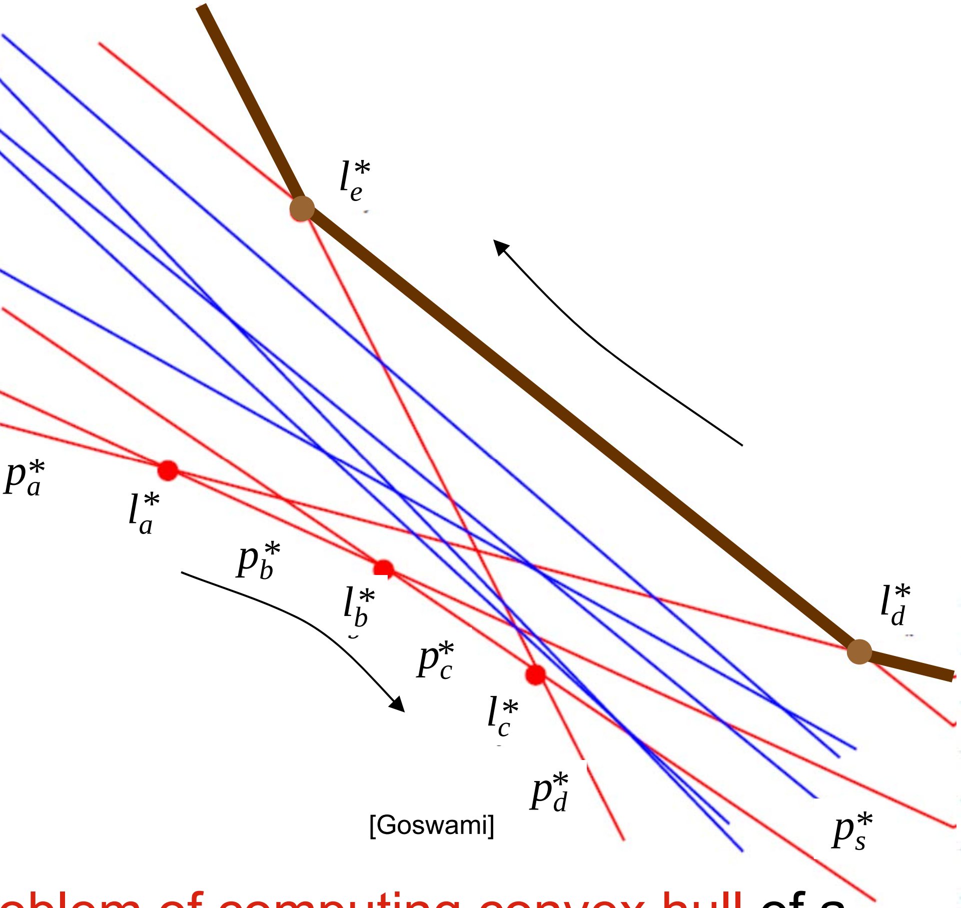


[Goswami]

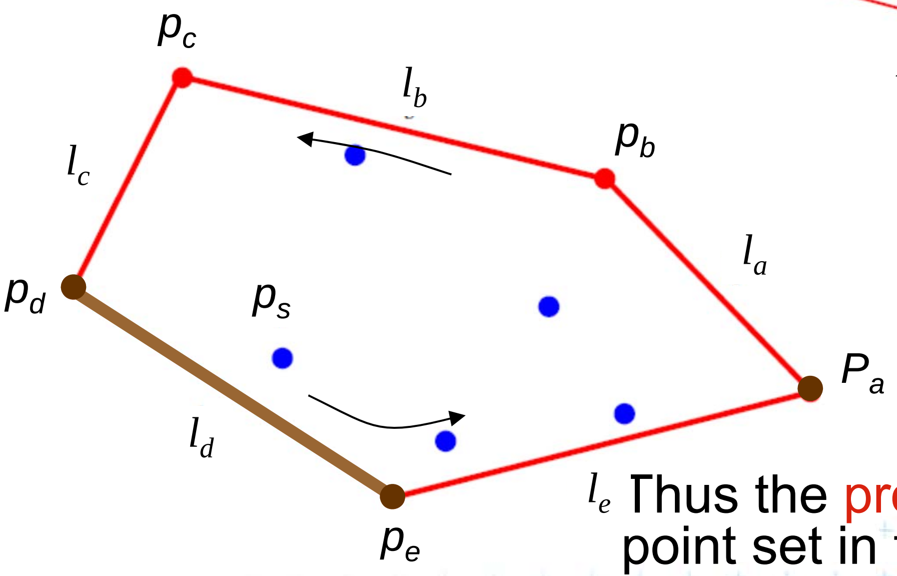
Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.



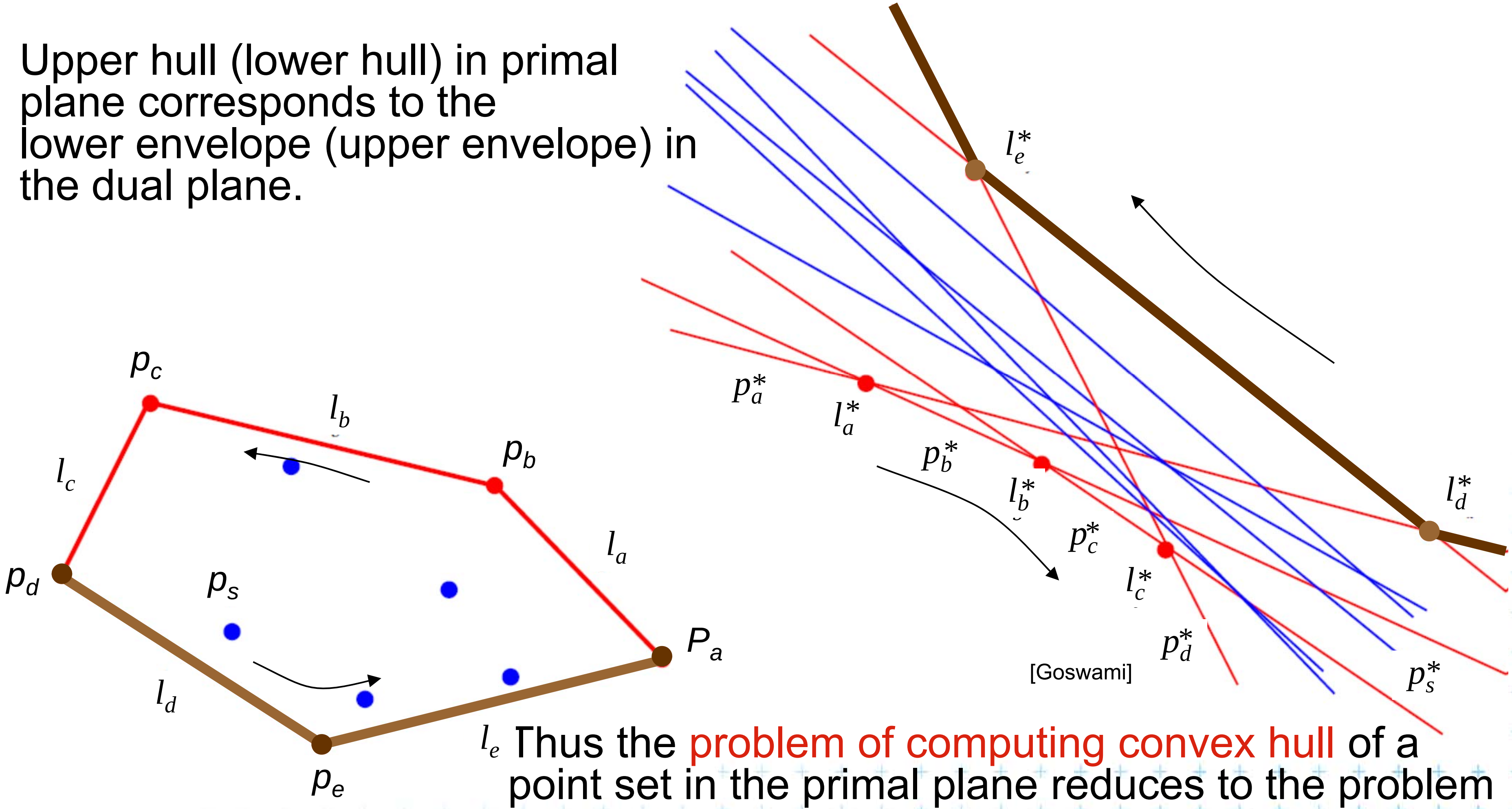
[Goswami]



Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.

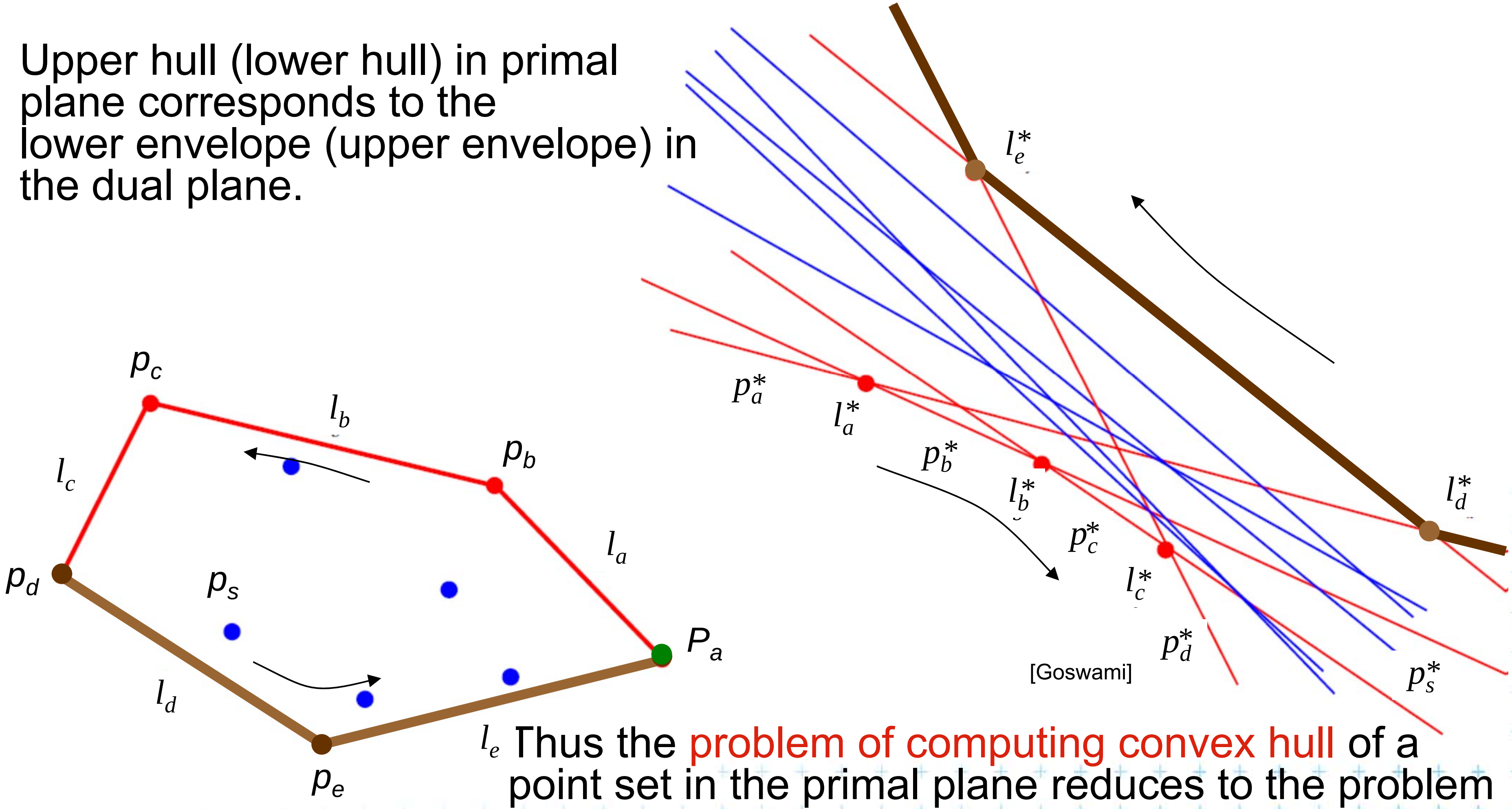


[Goswami]

Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.

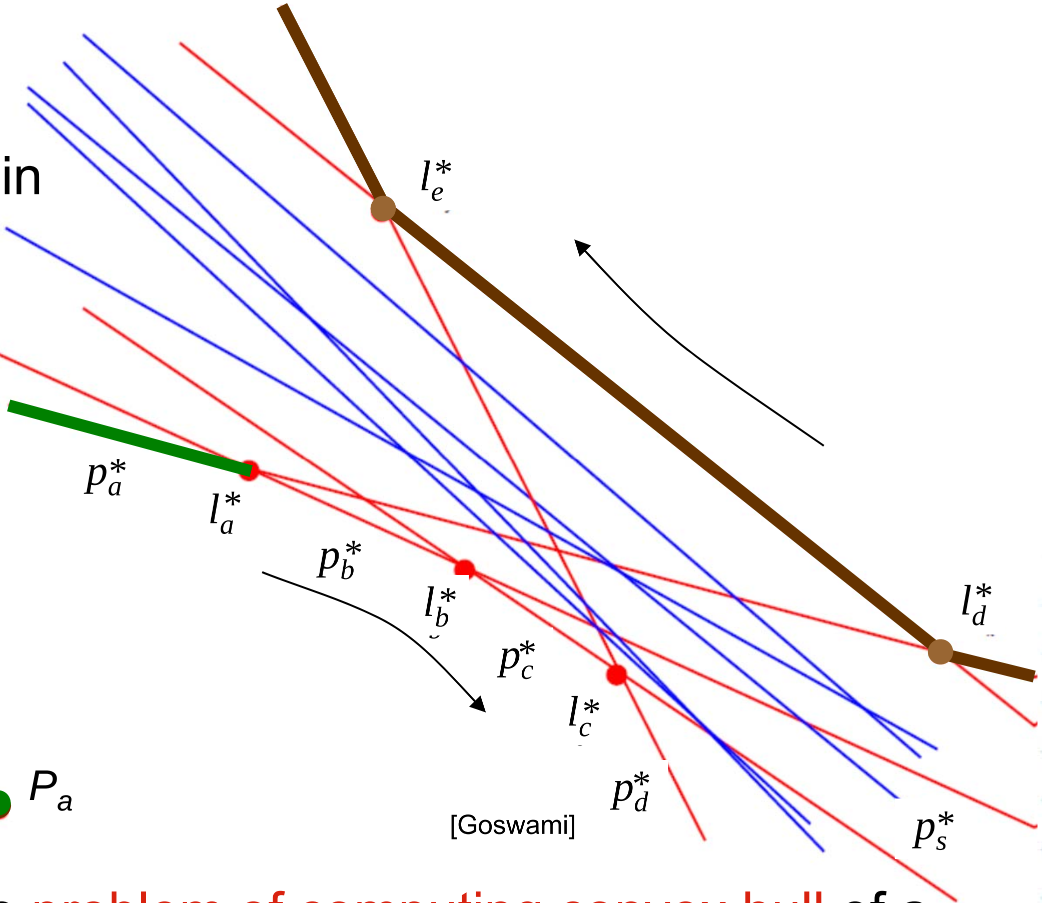


[Goswami]

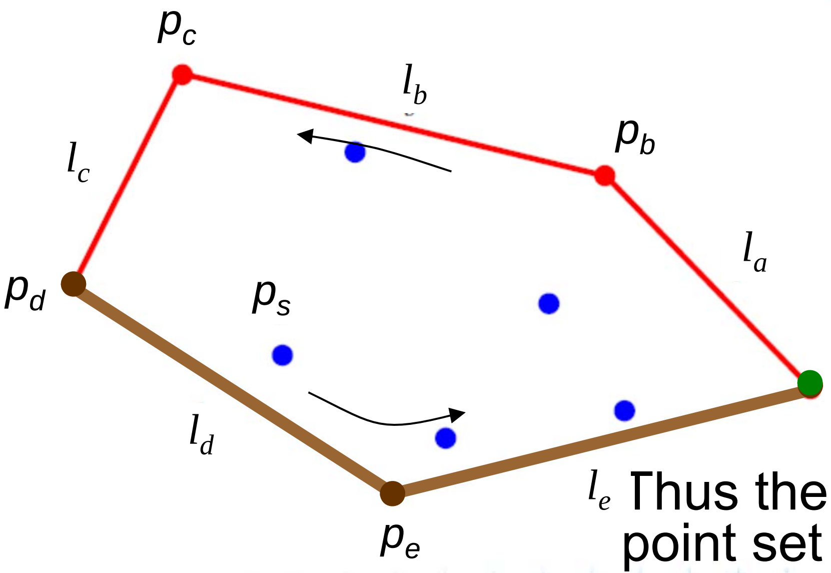
Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.



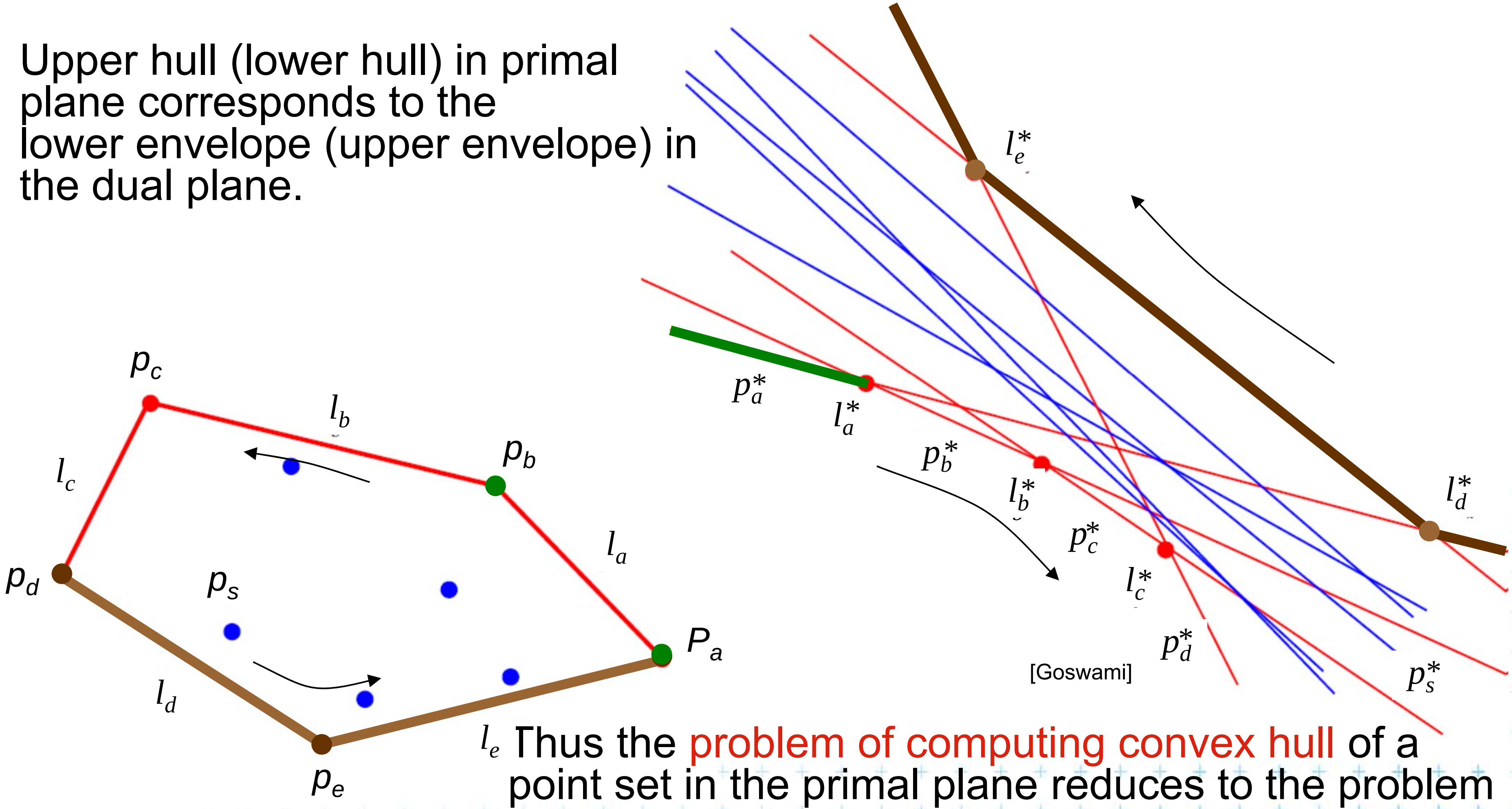
[Goswami]



Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.

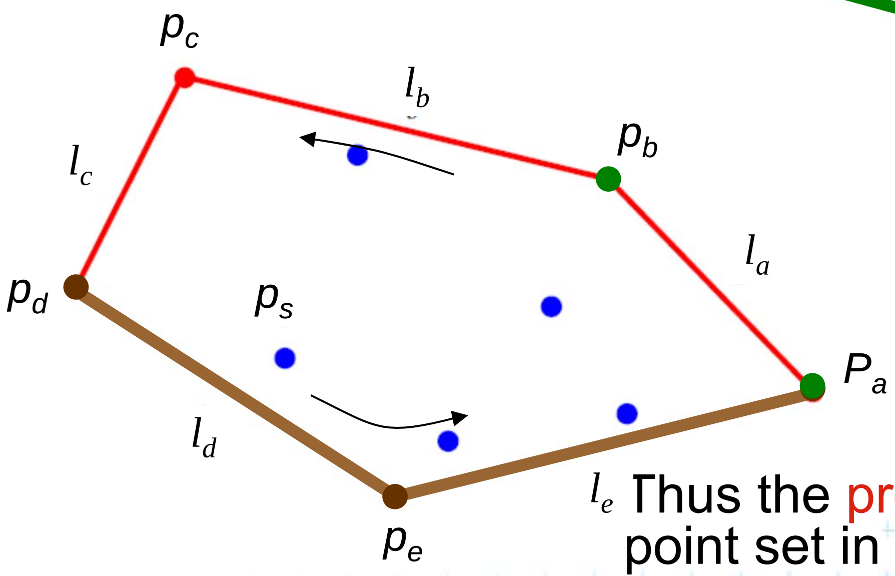


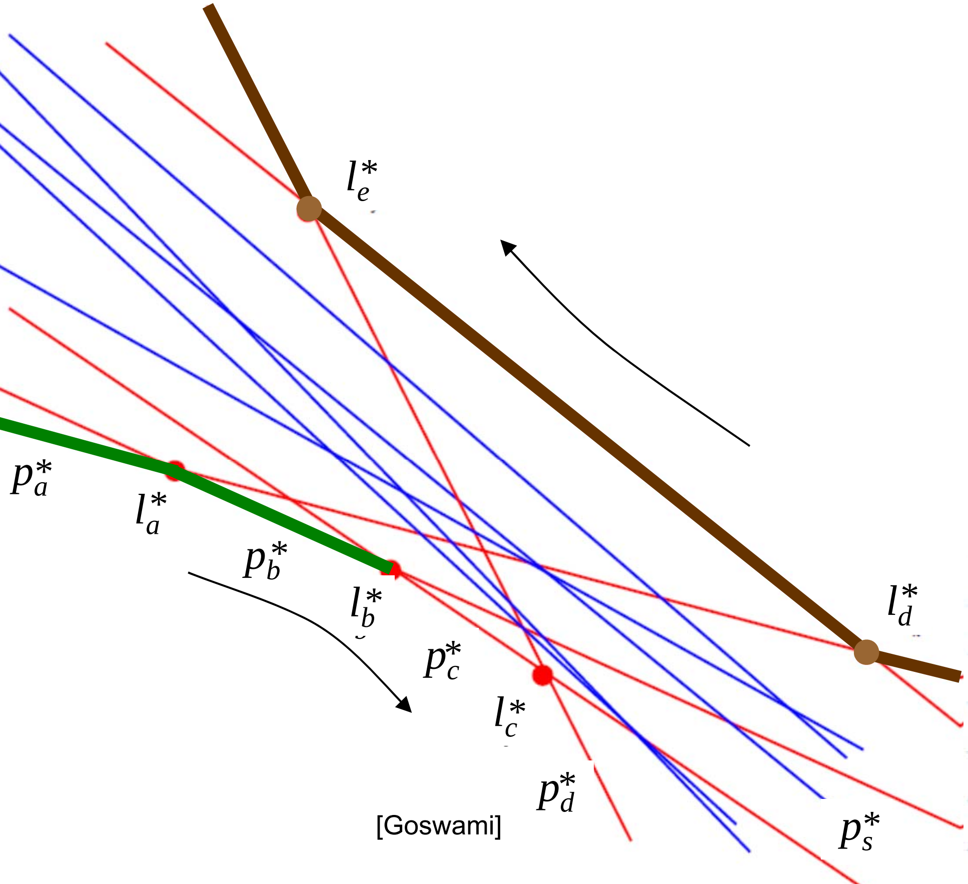
[Goswami]

Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.



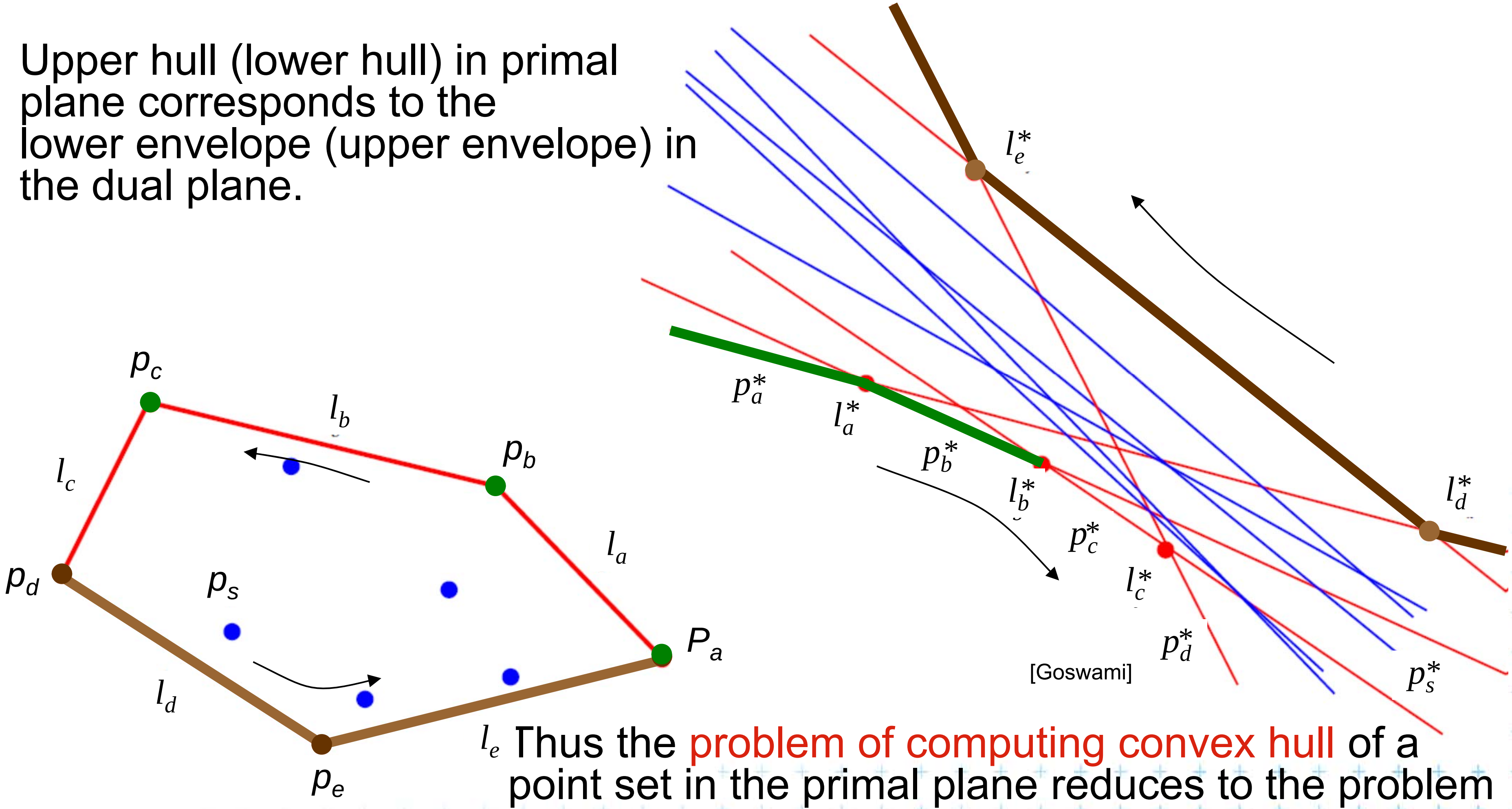


[Goswami]

Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.

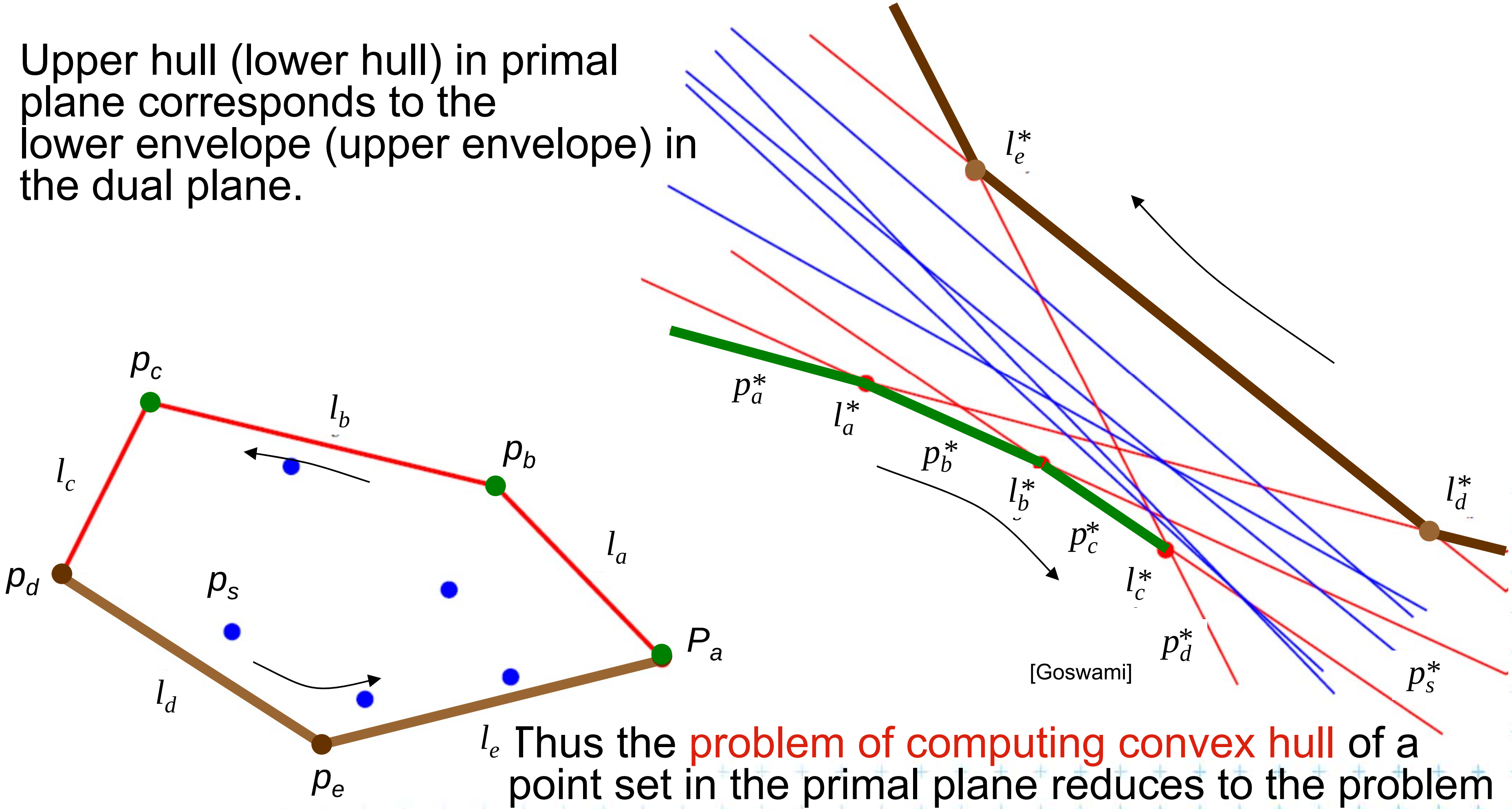


[Goswami]

Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.

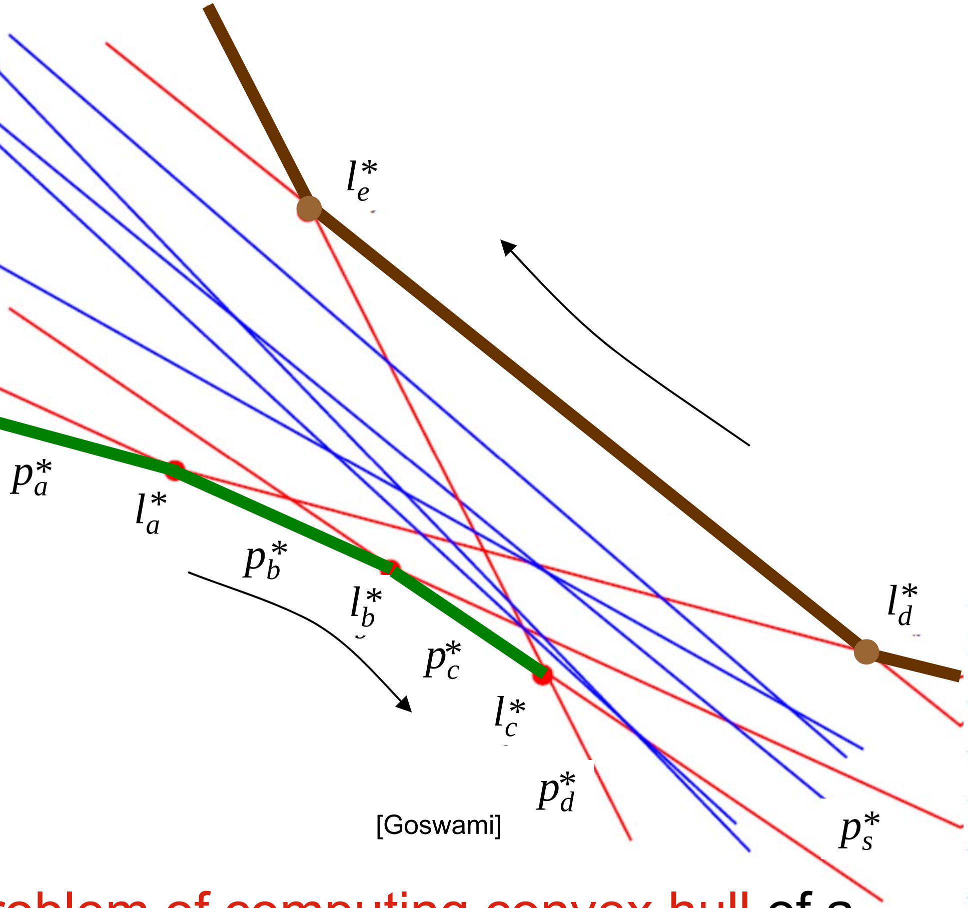


[Goswami]

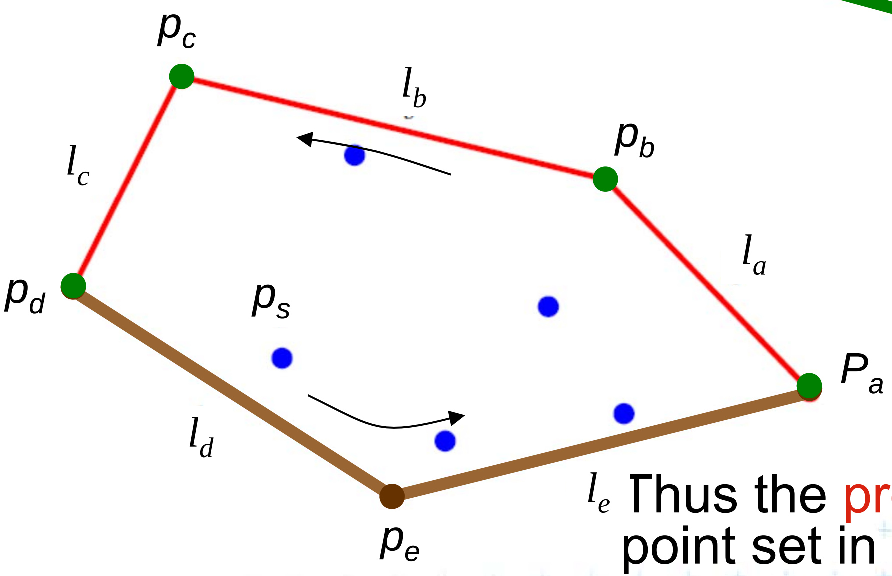
Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.



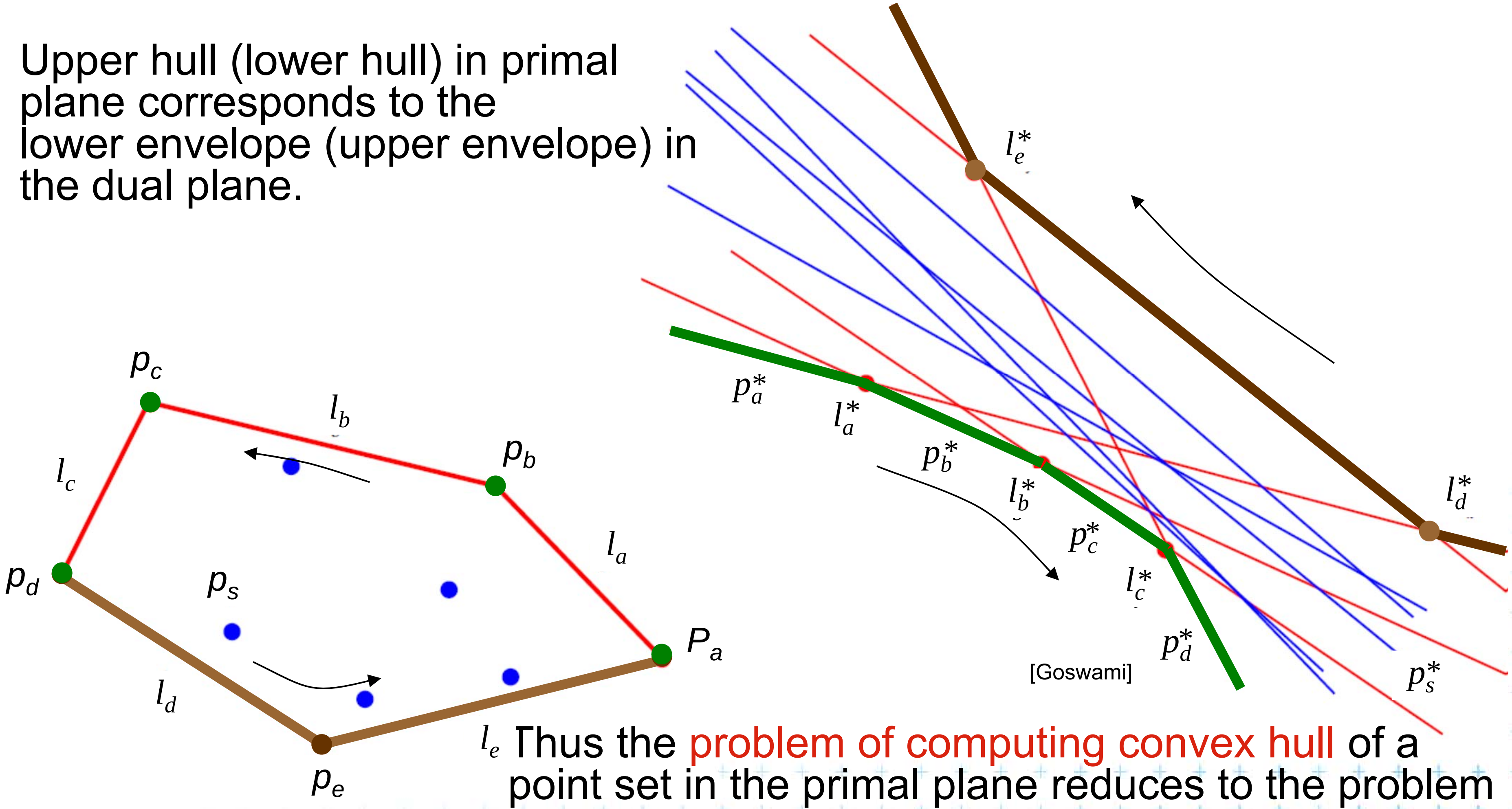
[Goswami]



Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.

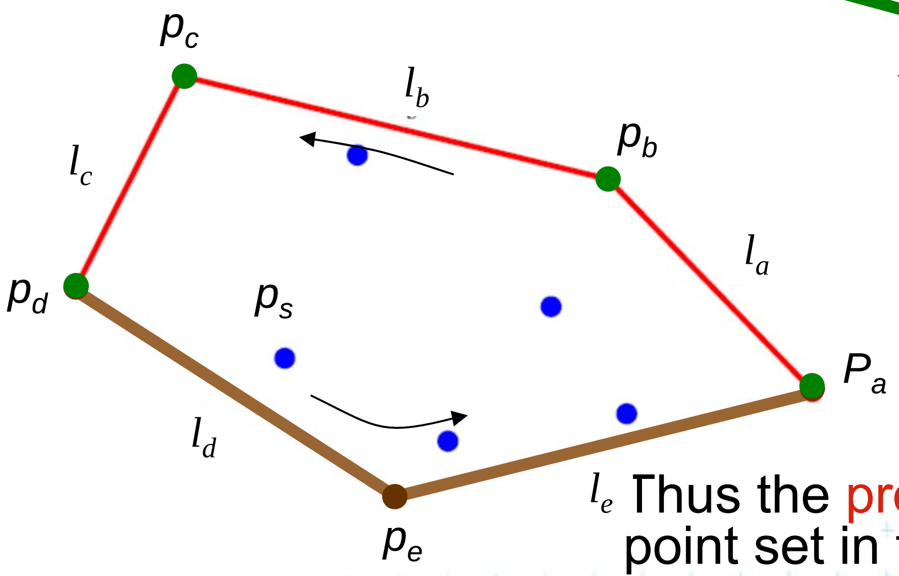


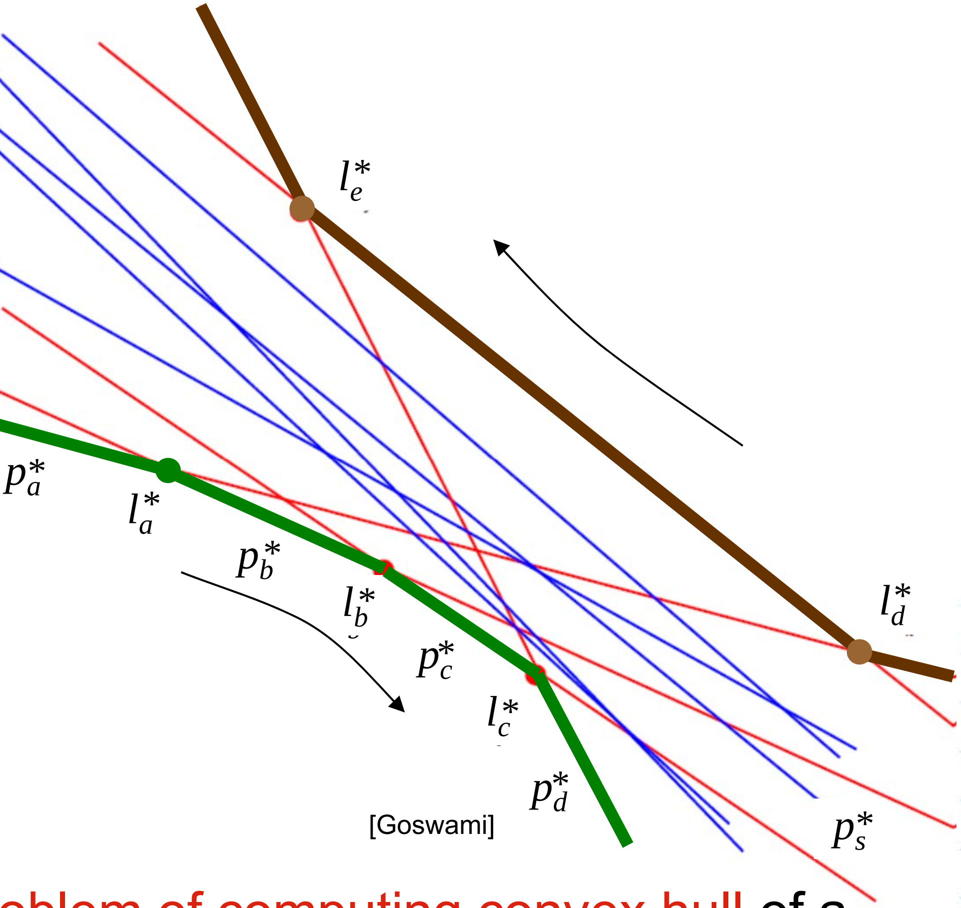
[Goswami]

Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.



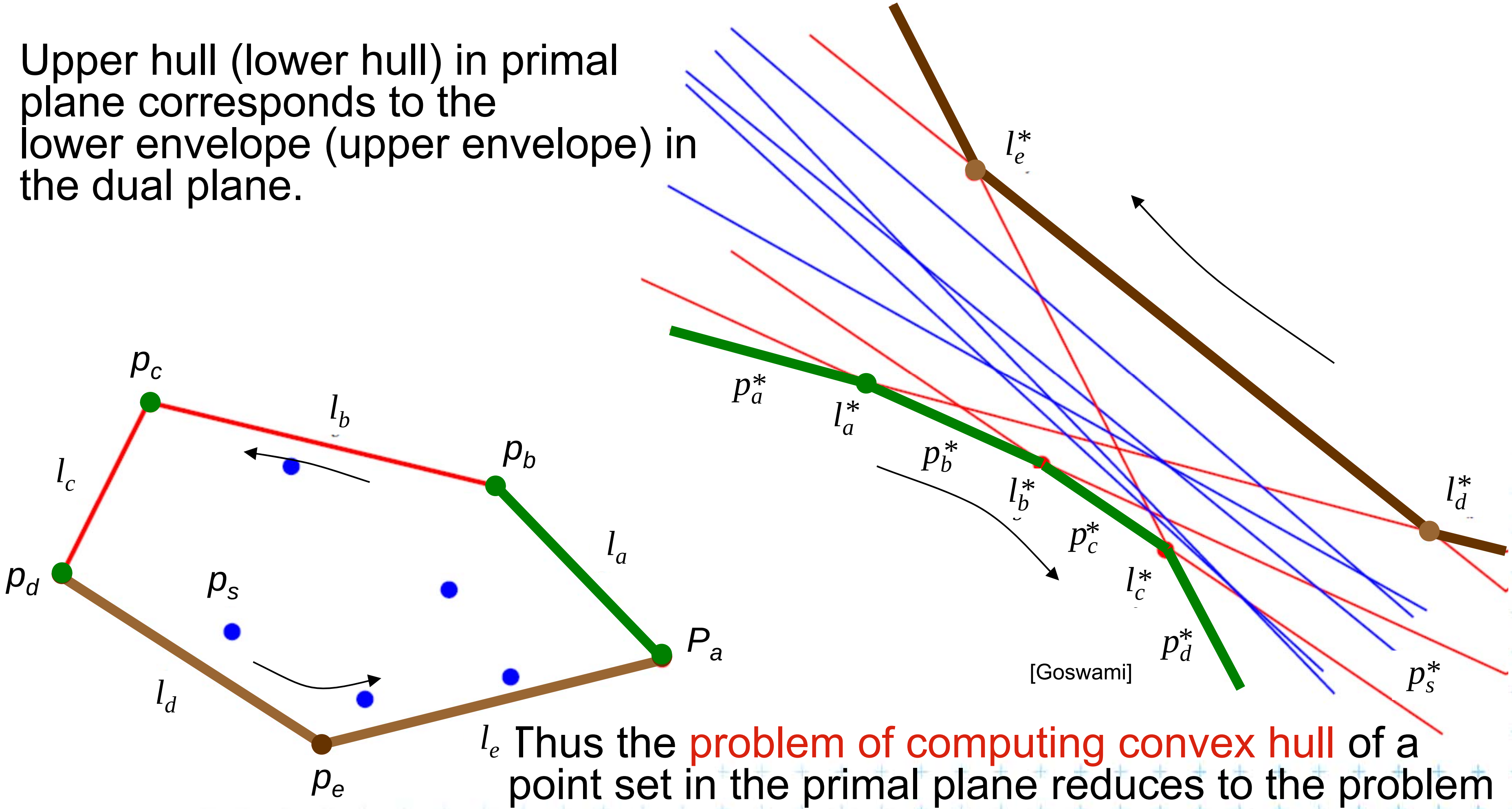


[Goswami]

Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.

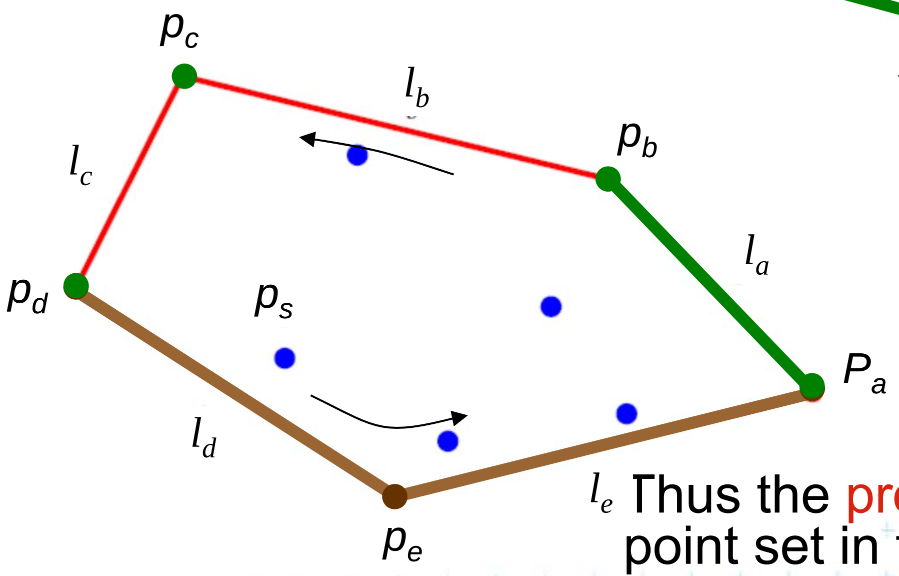


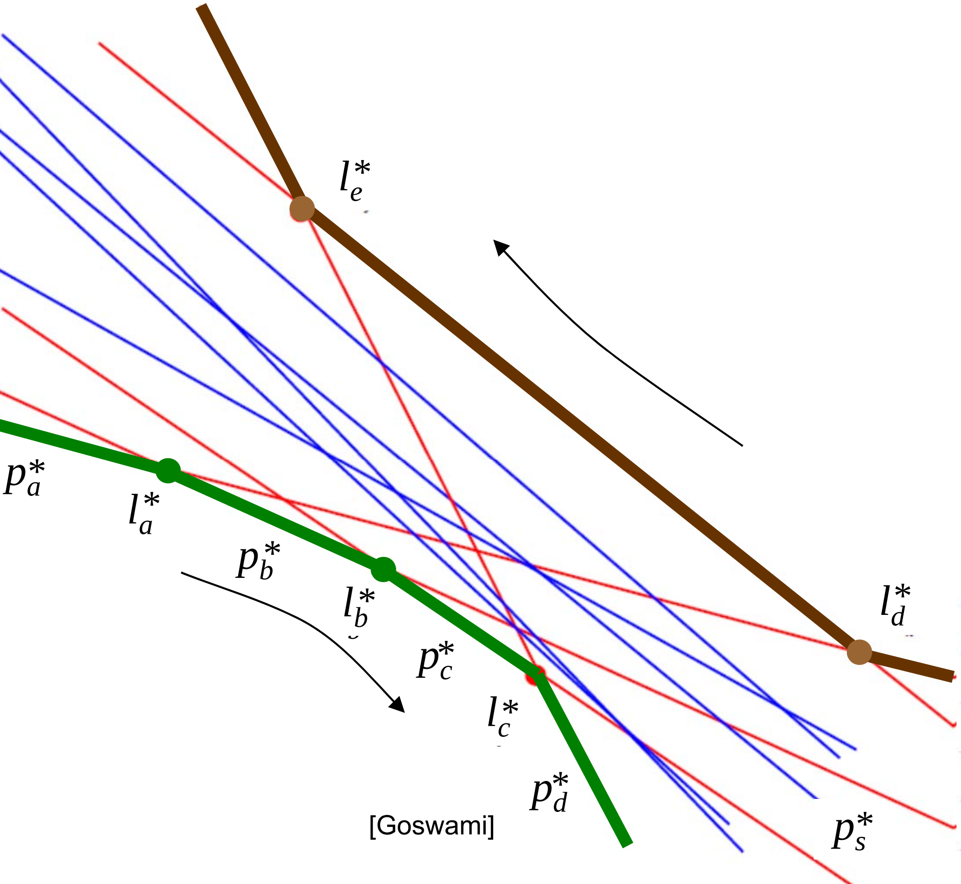
[Goswami]

Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.



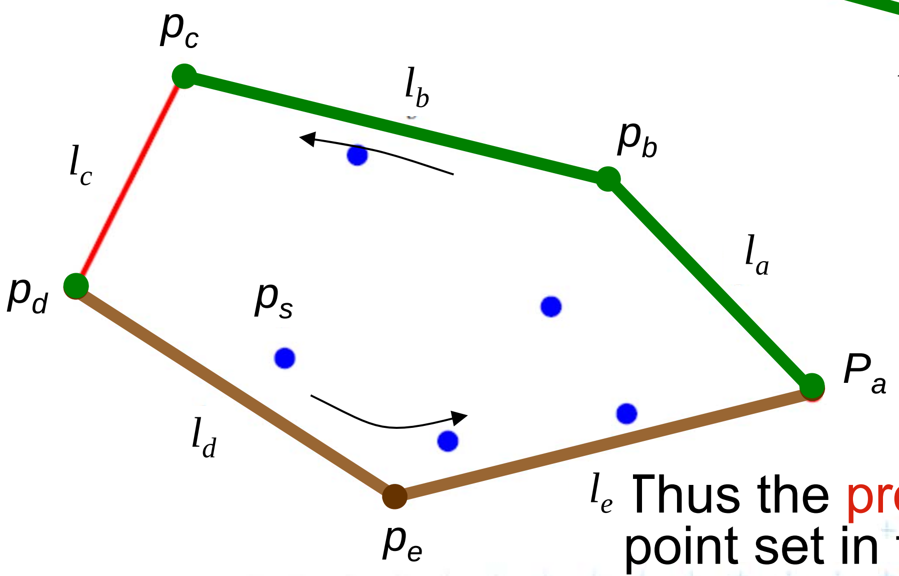


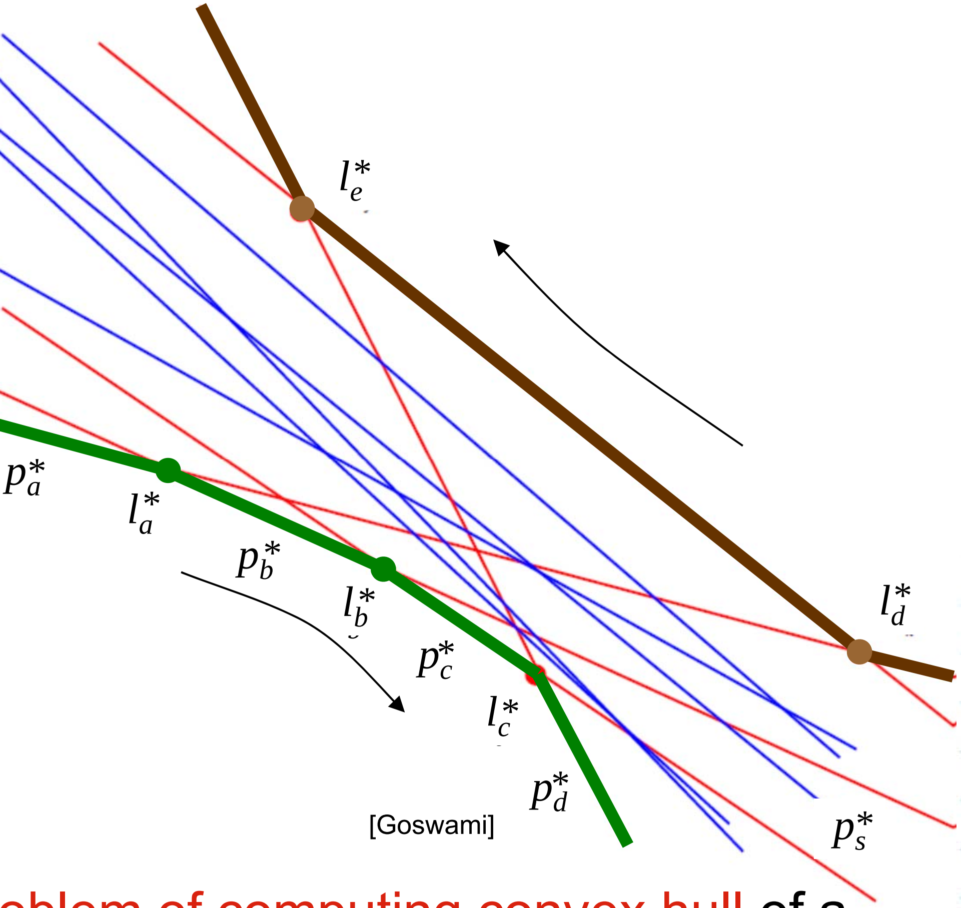
[Goswami]

Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.



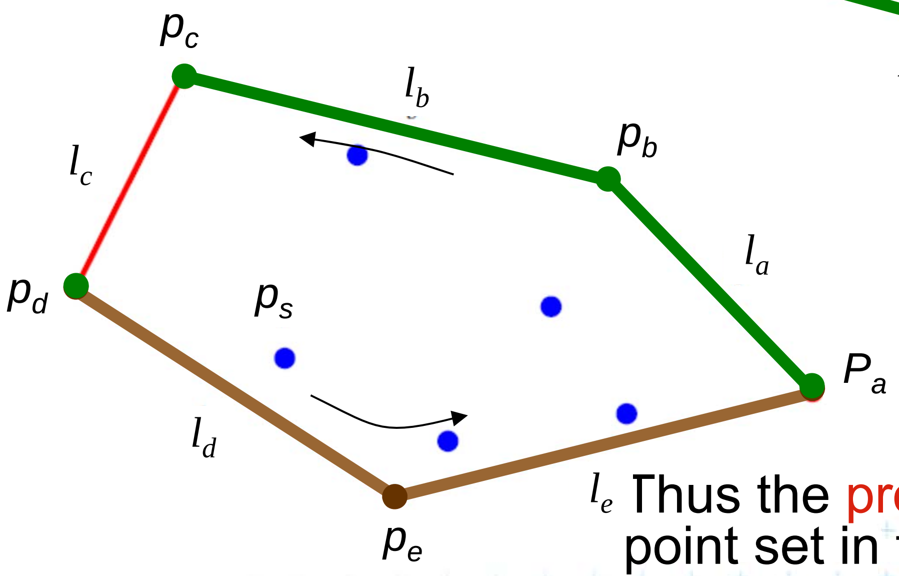


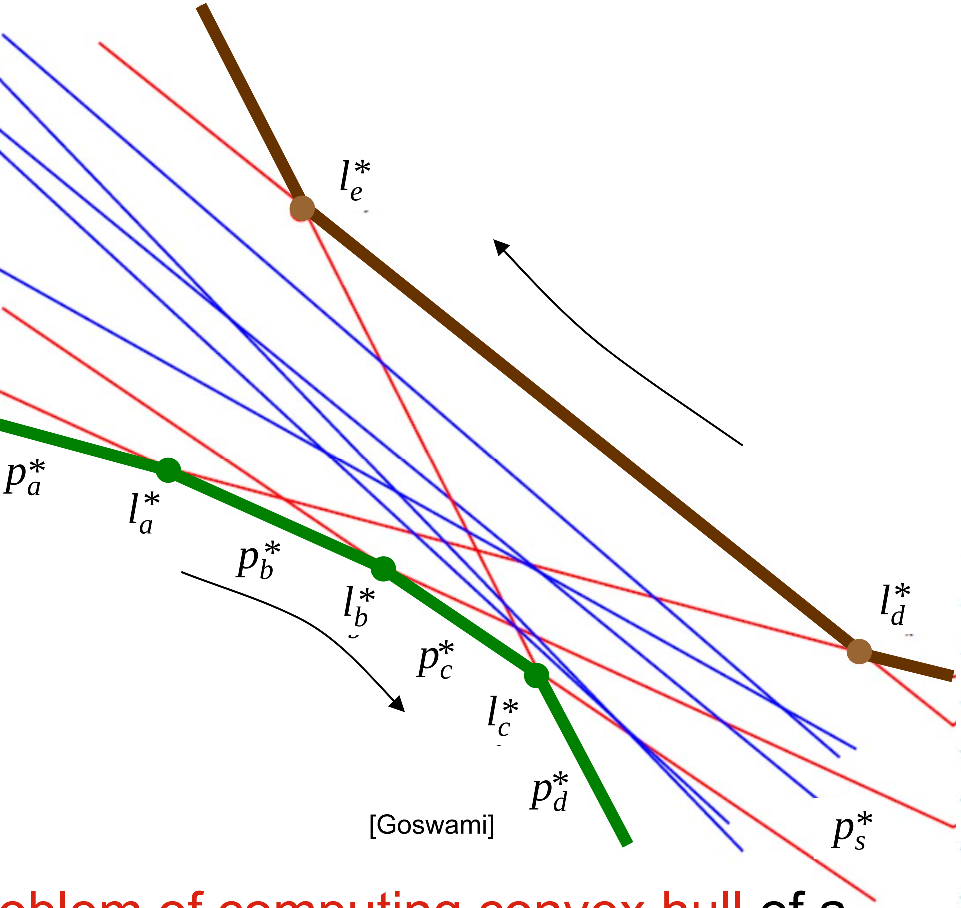
[Goswami]

Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.



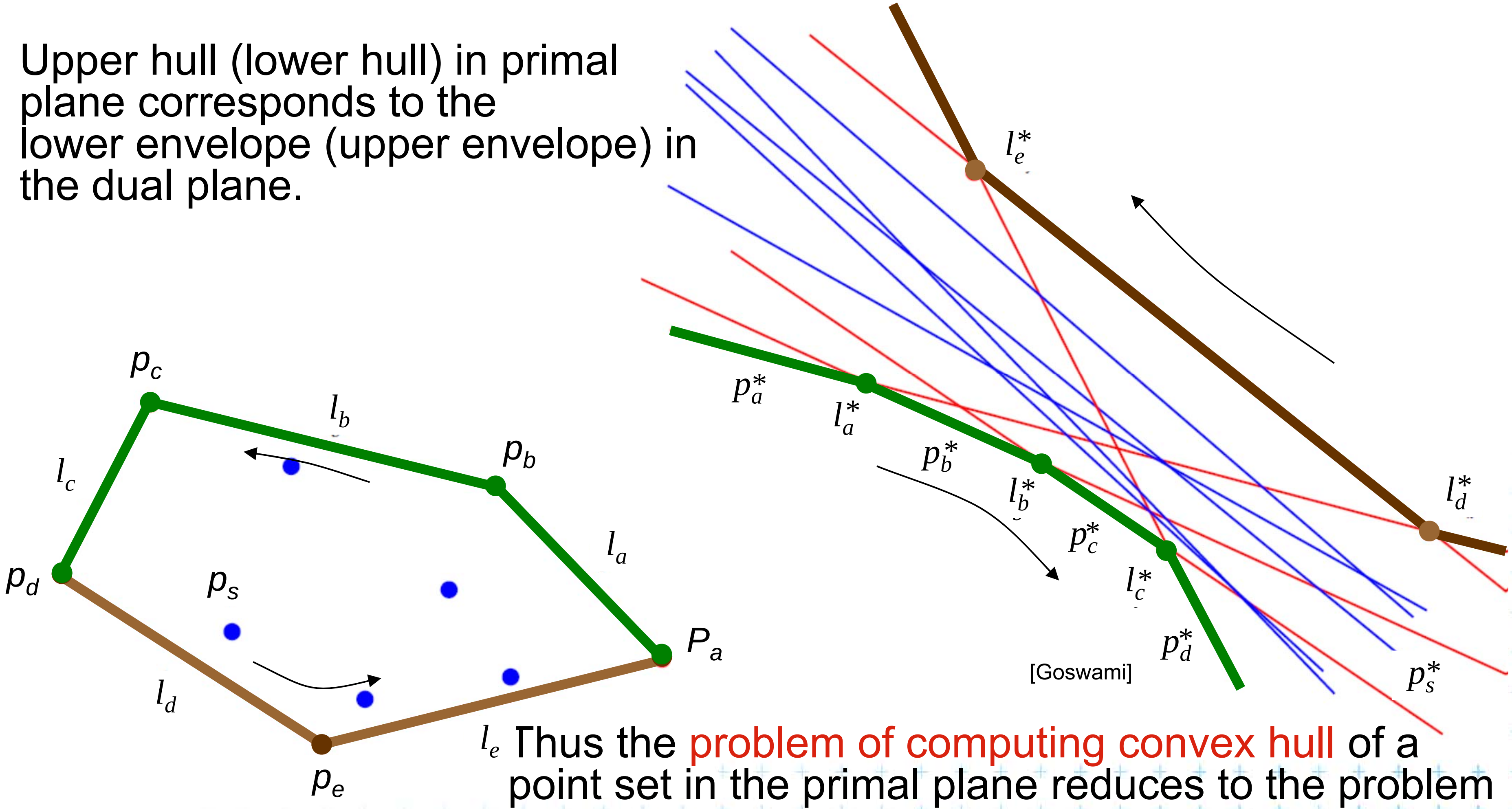


[Goswami]

Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Connection between Hull and Envelope

Upper hull (lower hull) in primal plane corresponds to the lower envelope (upper envelope) in the dual plane.

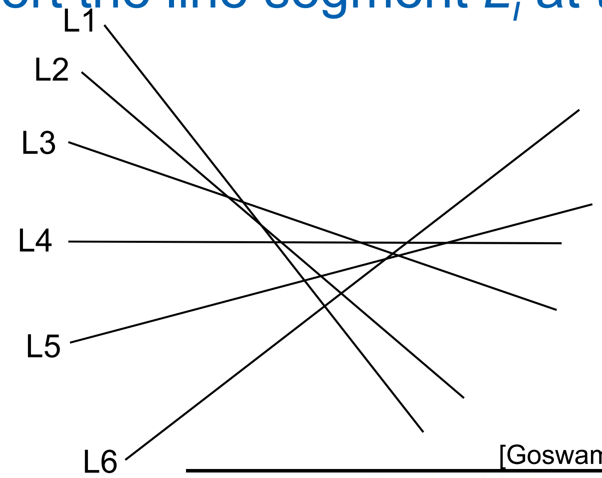


[Goswami]

Thus the problem of computing convex hull of a point set in the primal plane reduces to the problem of computing upper and lower envelopes of the line set in the dual plane.

Upper envelope algorithm

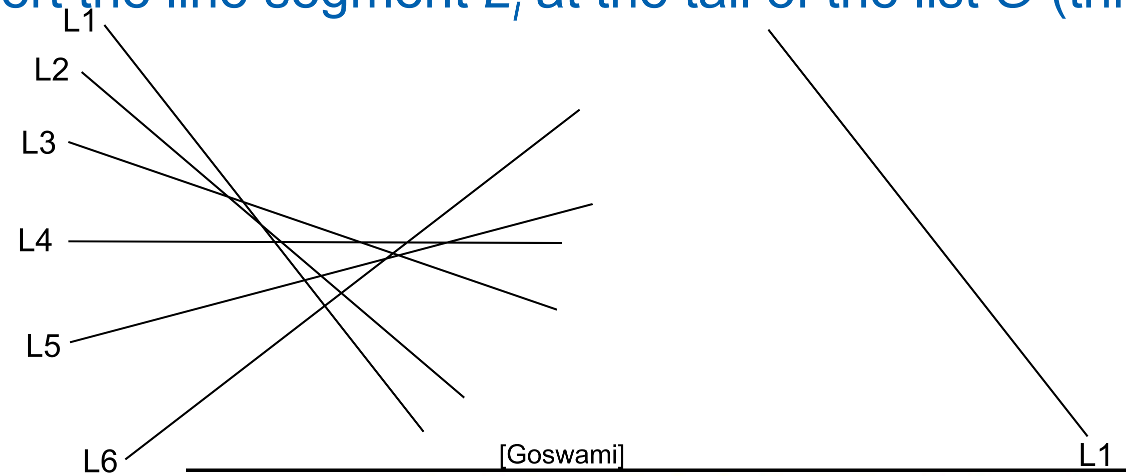
```
UpperEnvelope(L)
        Set of lines L sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                          // the only complete line in O
for i = 2 to n
    L = last entry in O  // O contains half-lines, or line segments,
                         //  except of complete line L1
    while( the line segment L does not intersect line $L_i$)
        remove L from O and replace L with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```



[Goswami]

Upper envelope algorithm

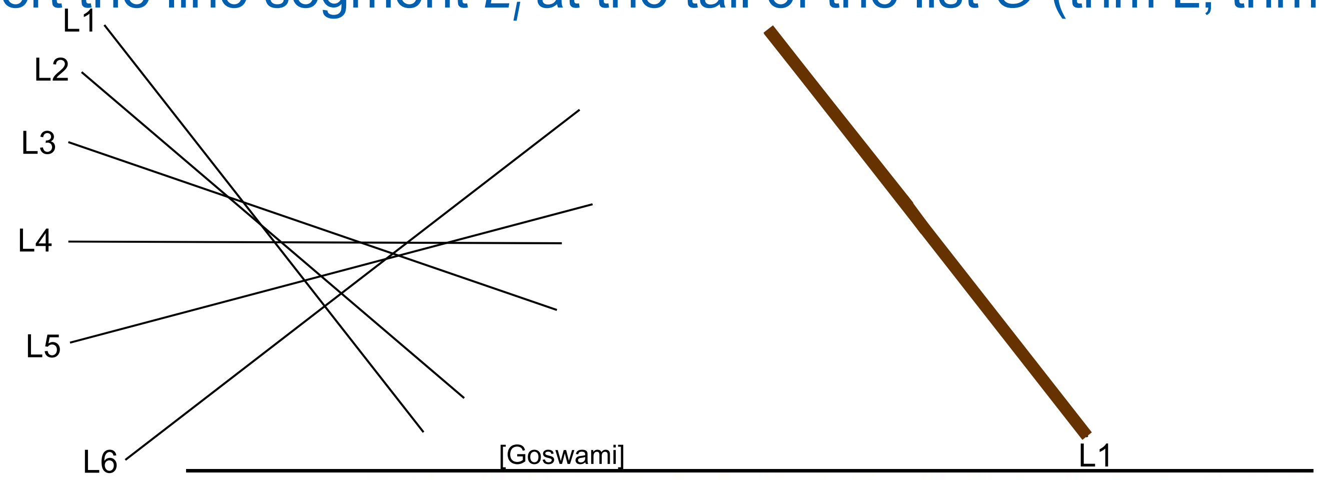
```
UpperEnvelope(L)
        Set of lines L sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                          // the only complete line in O
for i = 2 to n
    L = last entry in O  // O contains half-lines, or line segments,
                         // except of complete line L1
    while( the line segment L does not intersect line $L_i$)
        remove L from O and replace L with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```



[Goswami]

Upper envelope algorithm

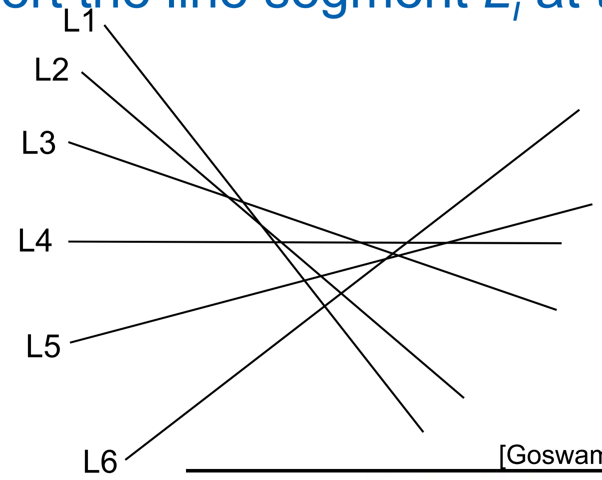
```
UpperEnvelope(L)
        Set of lines L sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                          // the only complete line in O
for i = 2 to n
    L = last entry in O   // O contains half-lines, or line segments,
                          //  except of complete line L1
    while( the line segment L does not intersect line $L_i$)
        remove L from O and replace L with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```

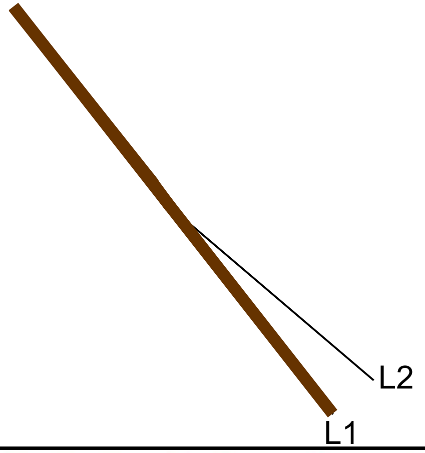


[Goswami]

Upper envelope algorithm

```
UpperEnvelope(L)
        Set of lines L sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                              // the only complete line in O
for i = 2 to n
    L = last entry in O  // O contains half-lines, or line segments,
                         //  except of complete line L1
    while( the line segment L does not intersect line $L_i$)
        remove L from O and replace L with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```



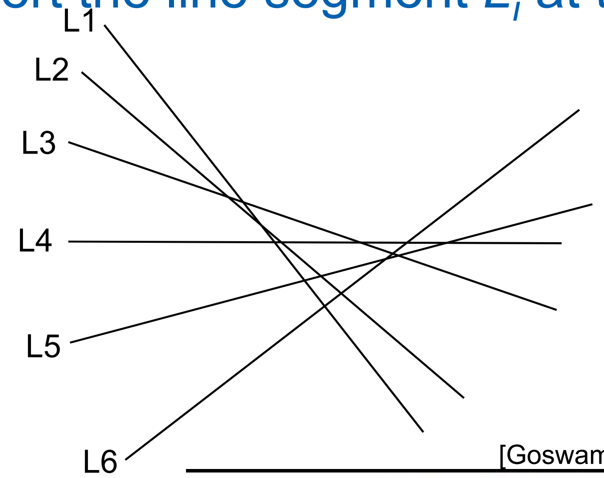


[Goswami]

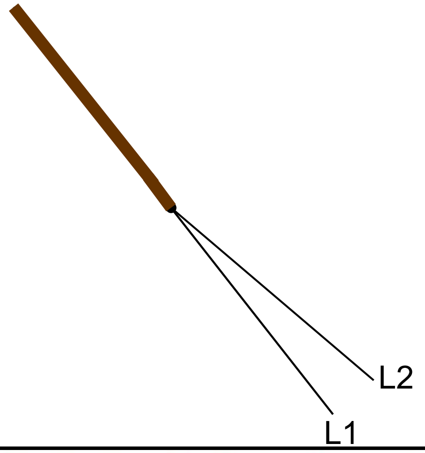
Upper envelope algorithm

UpperEnvelope(*L*)

```
        Set of lines $L$ sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                          // the only complete line in O
for $i$ = 2 to $n$
    L = last entry in $O$  // O contains half-lines, or line segments,
                           //  except of complete line L1
    while( the line segment $L$ does not intersect line $L_i$)
        remove $L$ from $O$ and replace $L$ with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list $O$ (trim L, trim $L_i$)
```

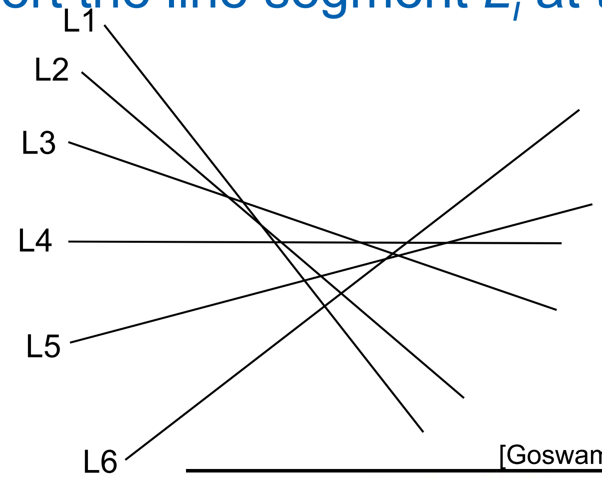


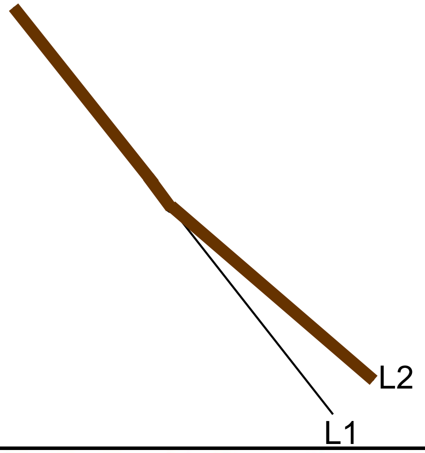
[Goswami]



Upper envelope algorithm

```
UpperEnvelope(L)
        Set of lines L sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                          // the only complete line in O
for i = 2 to n
    L = last entry in O  // O contains half-lines, or line segments,
                         //  except of complete line L1
    while( the line segment L does not intersect line $L_i$)
        remove L from O and replace L with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```





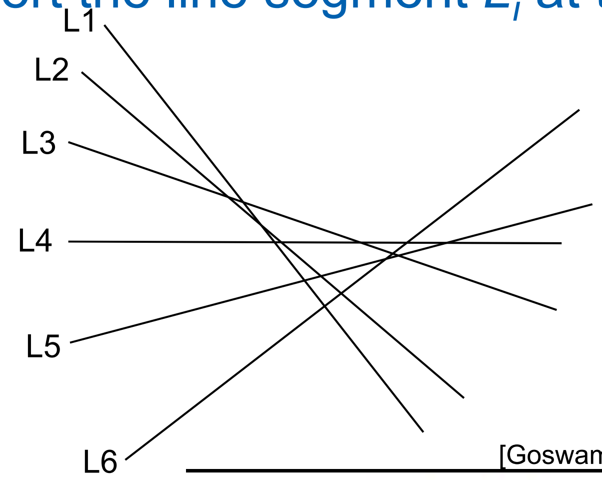
[Goswami]

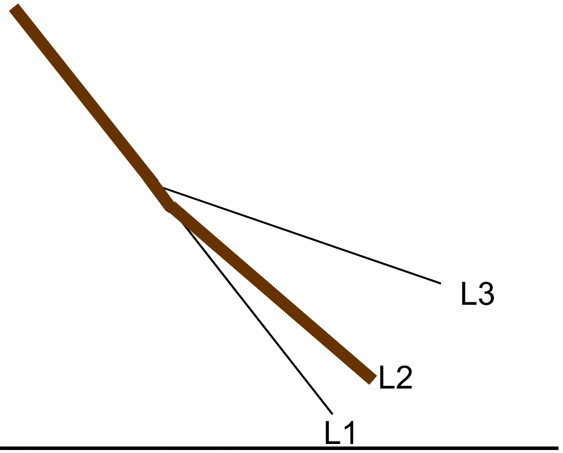
Upper envelope algorithm

UpperEnvelope(*L*)

Set of lines *L* sorted by increasing order of slopes (-90° to 90°)
Polygonal chain O representing the upper hull

```
O = L1                          // the only complete line in O
for i = 2 to n
    L = last entry in O   // O contains half-lines, or line segments,
                          //  except of complete line L1
    while( the line segment L does not intersect line $L_i$)
        remove L from O and replace L with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```

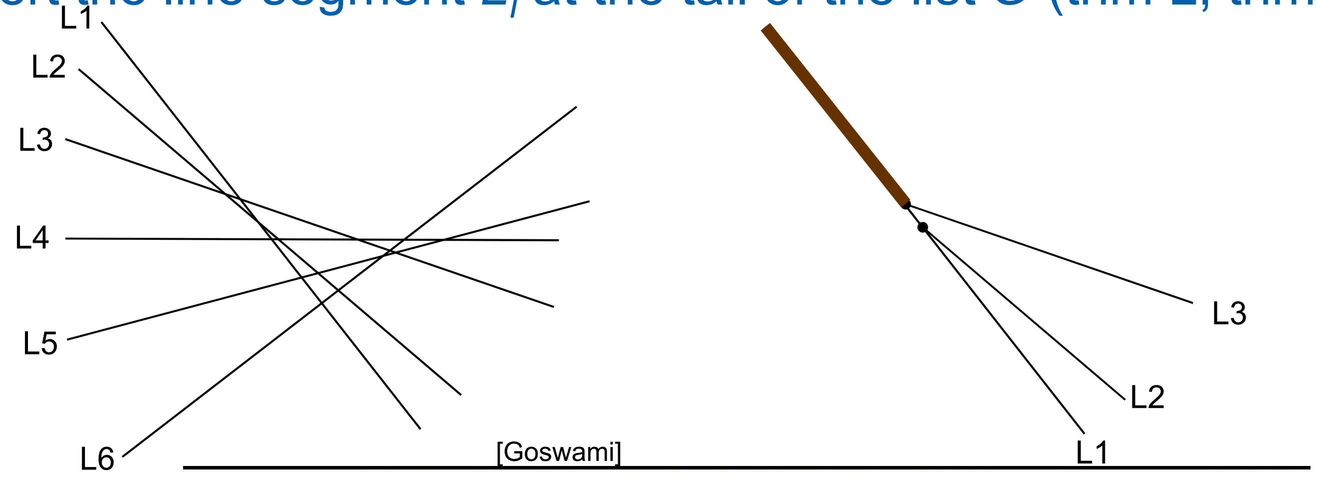




[Goswami]

Upper envelope algorithm

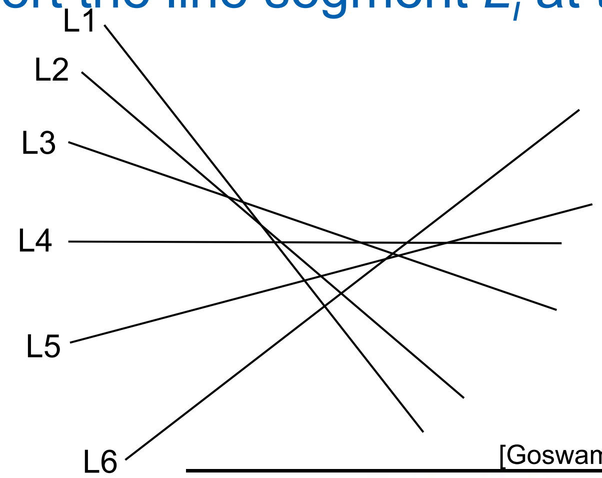
```
UpperEnvelope(L)
        Set of lines L sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                        // the only complete line in O
for i = 2 to n
    L = last entry in O  // O contains half-lines, or line segments,
                         //  except of complete line L1
    while( the line segment L does not intersect line $L_i$)
        remove L from O and replace L with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```

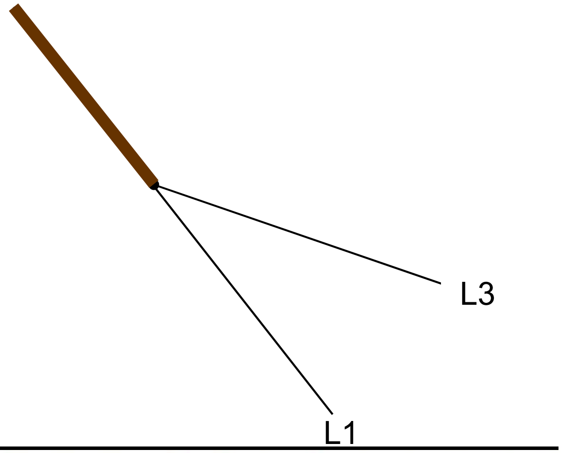


[Goswami]

Upper envelope algorithm

```
UpperEnvelope(L)
        Set of lines L sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                        // the only complete line in O
for i = 2 to n
    L = last entry in O  // O contains half-lines, or line segments,
                         //  except of complete line L1
    while( the line segment L does not intersect line $L_i$)
        remove L from O and replace L with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```

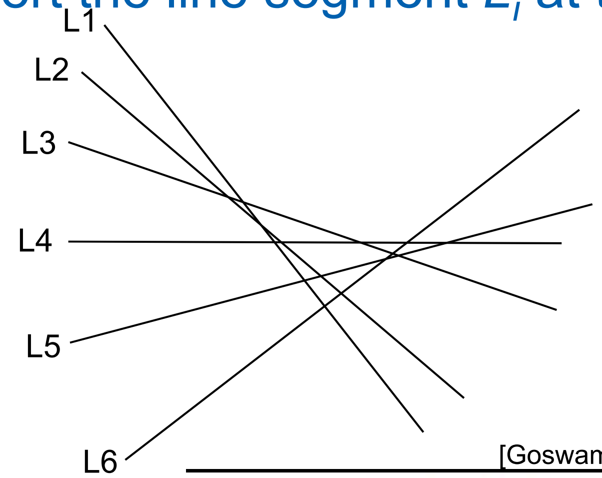




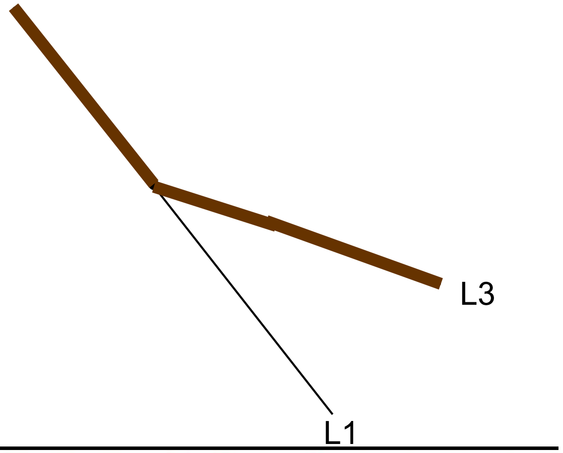
[Goswami]

Upper envelope algorithm

```
UpperEnvelope($L$)
        Set of lines $L$ sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                              // the only complete line in O
for $i$ = 2 to $n$
    L = last entry in O  // O contains half-lines, or line segments,
                         //  except of complete line L1
    while( the line segment $L$ does not intersect line $L_i$)
        remove $L$ from $O$ and replace $L$ with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list $O$ (trim L, trim $L_i$)
```

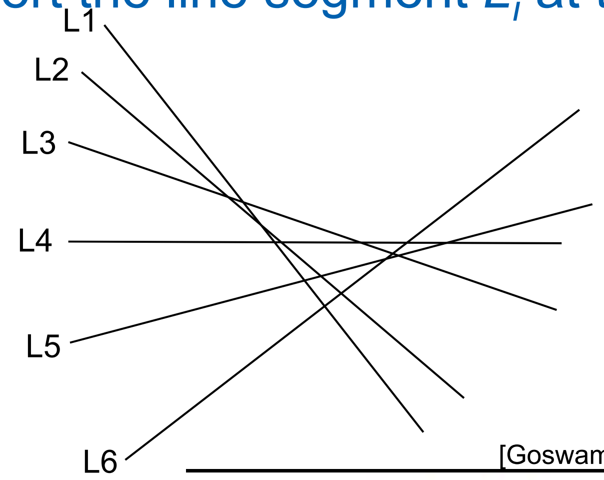


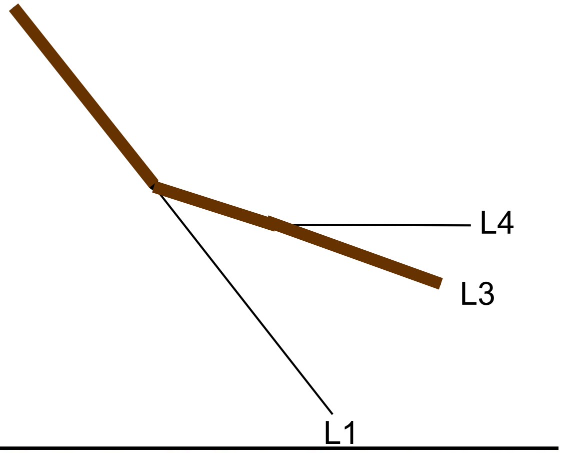
[Goswami]



Upper envelope algorithm

```
UpperEnvelope(L)
        Set of lines L sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                       // the only complete line in O
for i = 2 to n
    L = last entry in O  // O contains half-lines, or line segments,
                         //  except of complete line L1
    while( the line segment L does not intersect line $L_i$)
        remove L from O and replace L with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```

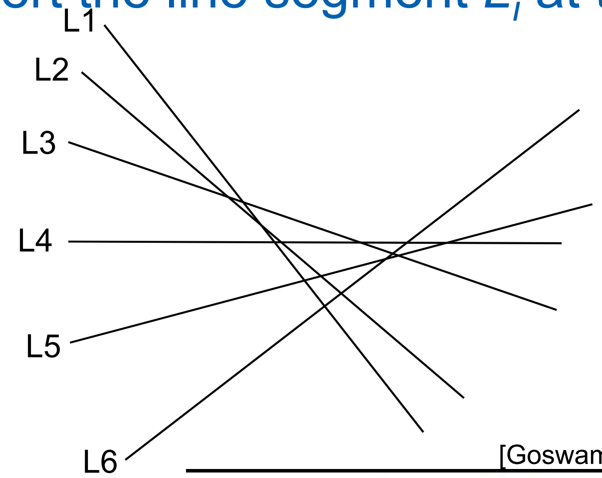


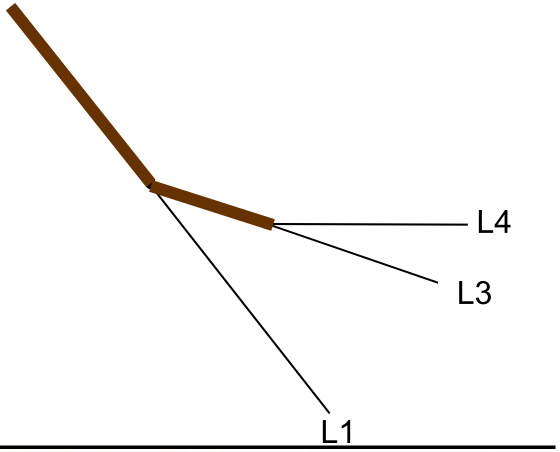


[Goswami]

Upper envelope algorithm

```
UpperEnvelope(L)
        Set of lines L sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                          // the only complete line in O
for i = 2 to n
    L = last entry in O   // O contains half-lines, or line segments,
                          //  except of complete line L1
    while( the line segment L does not intersect line $L_i$)
        remove L from O and replace L with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```

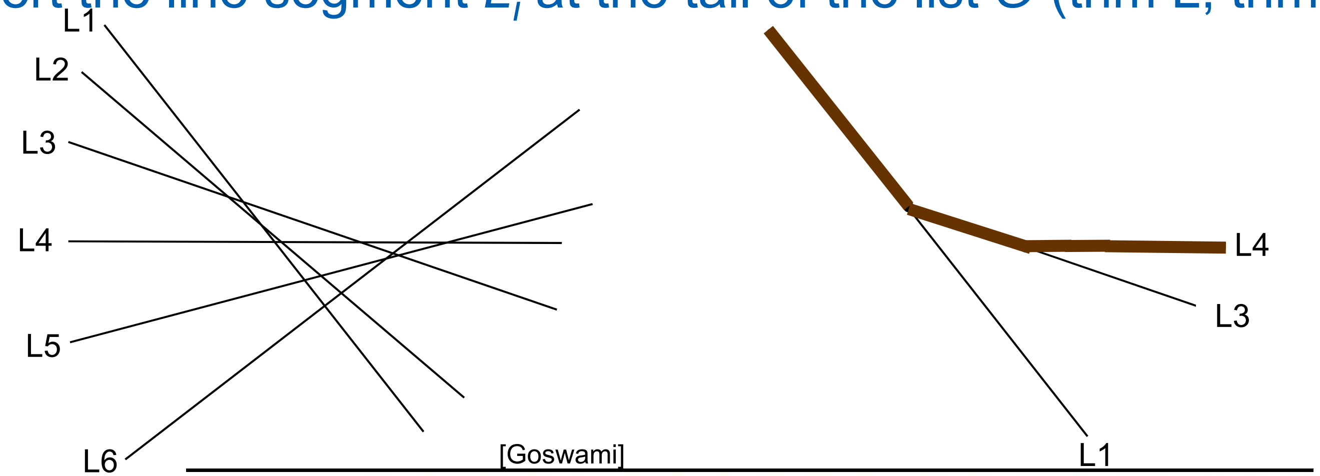




[Goswami]

Upper envelope algorithm

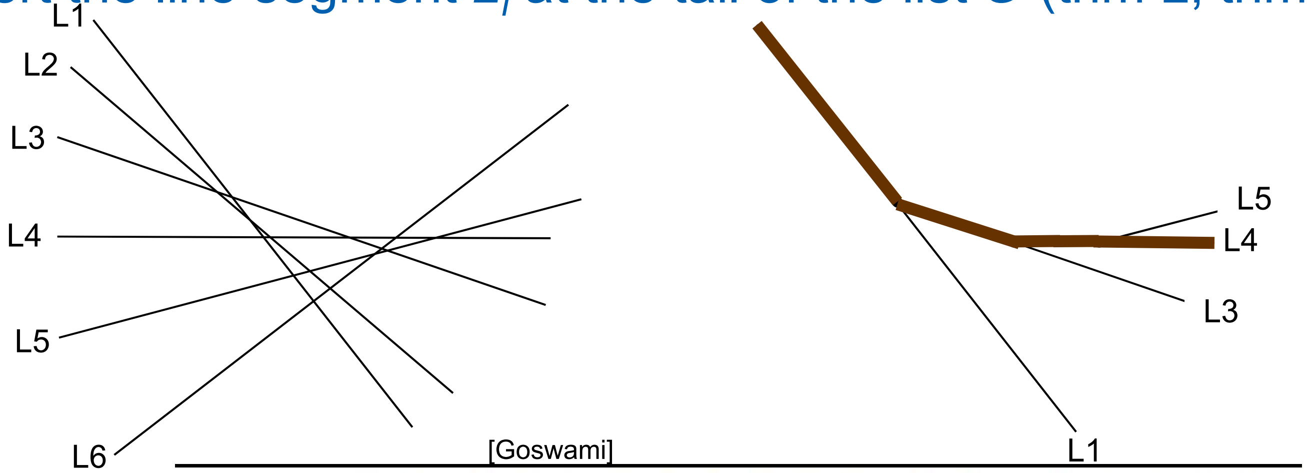
```
UpperEnvelope(L)
        Set of lines L sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                        // the only complete line in O
for i = 2 to n
    L = last entry in O  // O contains half-lines, or line segments,
                         //  except of complete line L1
    while( the line segment L does not intersect line $L_i$)
        remove L from O and replace L with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```



[Goswami]

Upper envelope algorithm

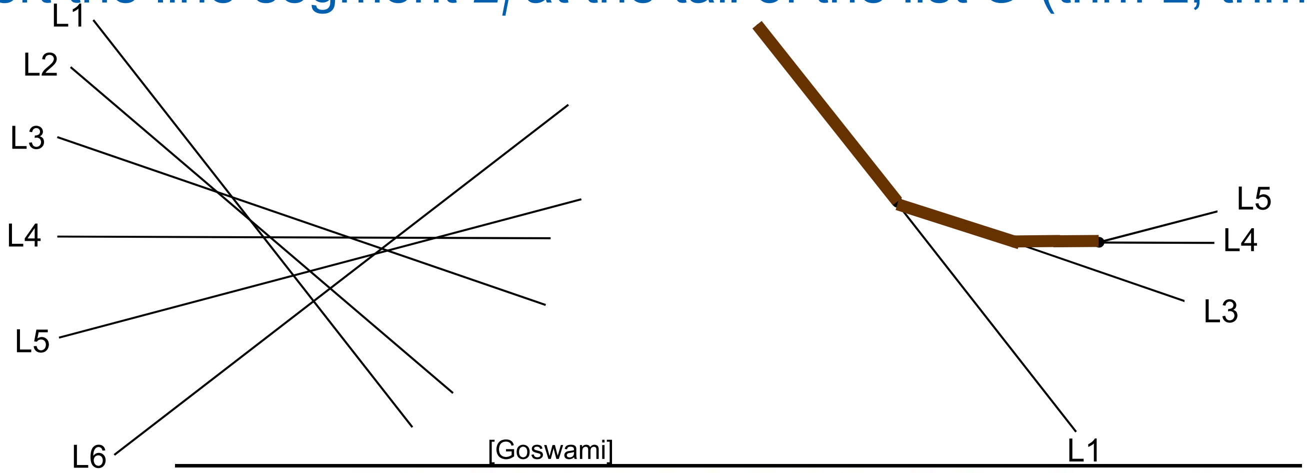
```
UpperEnvelope(L)
        Set of lines L sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                         // the only complete line in O
for i = 2 to n
    L = last entry in O  // O contains half-lines, or line segments,
                         //  except of complete line L1
    while( the line segment L does not intersect line $L_i$)
        remove L from O and replace L with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```



[Goswami]

Upper envelope algorithm

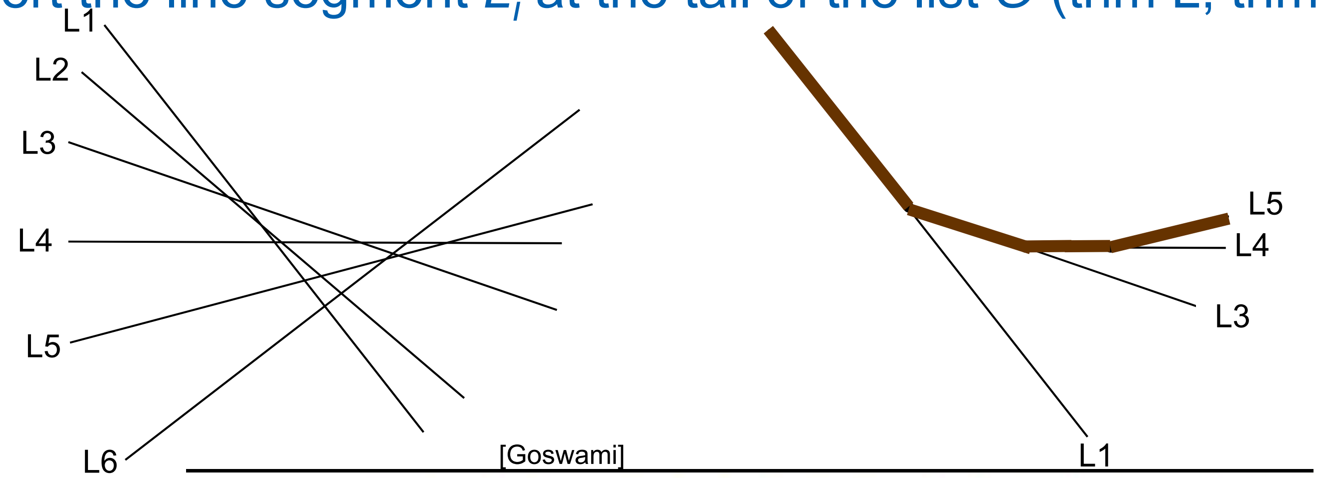
```
UpperEnvelope(L)
        Set of lines L sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                          // the only complete line in O
for i = 2 to n
    L = last entry in O         // O contains half-lines, or line segments,
                                //  except of complete line L1
    while( the line segment L does not intersect line $L_i$)
        remove L from O and replace L with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```



[Goswami]

Upper envelope algorithm

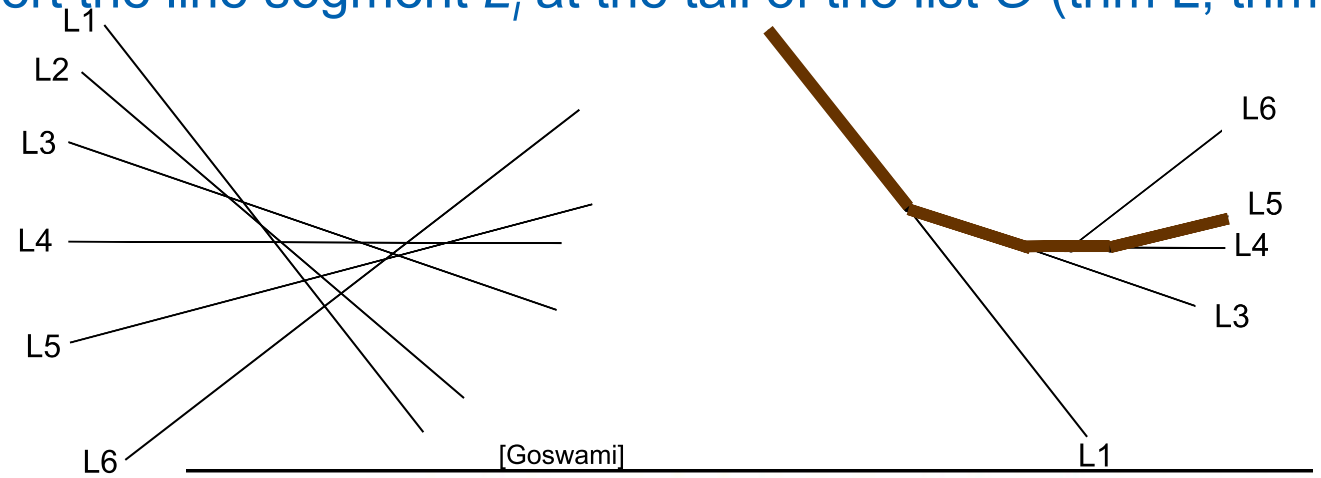
```
UpperEnvelope(L)
        Set of lines L sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                      // the only complete line in O
for i = 2 to n
    L = last entry in O     // O contains half-lines, or line segments,
                            //  except of complete line L1
    while( the line segment L does not intersect line $L_i$)
        remove L from O and replace L with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```



[Goswami]

Upper envelope algorithm

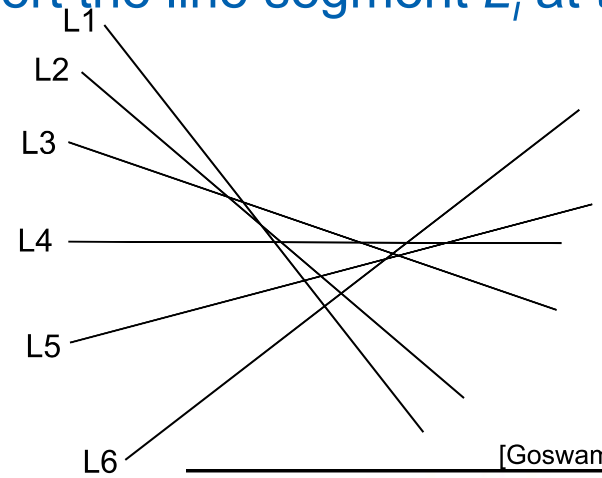
```
UpperEnvelope($L$)
        Set of lines $L$ sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                            // the only complete line in O
for $i$ = 2 to $n$
    L = last entry in O  // O contains half-lines, or line segments,
                         //  except of complete line L1
    while( the line segment $L$ does not intersect line $L_i$)
        remove $L$ from O and replace $L$ with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```

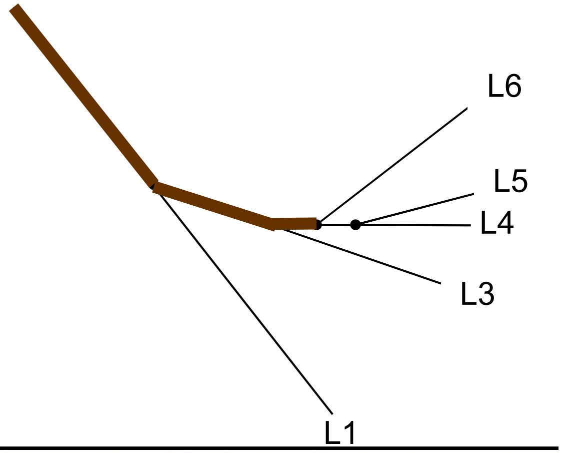


[Goswami]

Upper envelope algorithm

```
UpperEnvelope(L)
            Set of lines L sorted by increasing order of slopes (-90° to 90°)
            Polygonal chain O representing the upper hull
O = L1                        // the only complete line in O
for i = 2 to n
    L = last entry in O   // O contains half-lines, or line segments,
                          //  except of complete line L1
    while( the line segment L does not intersect line $L_i$)
        remove L from O and replace L with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```

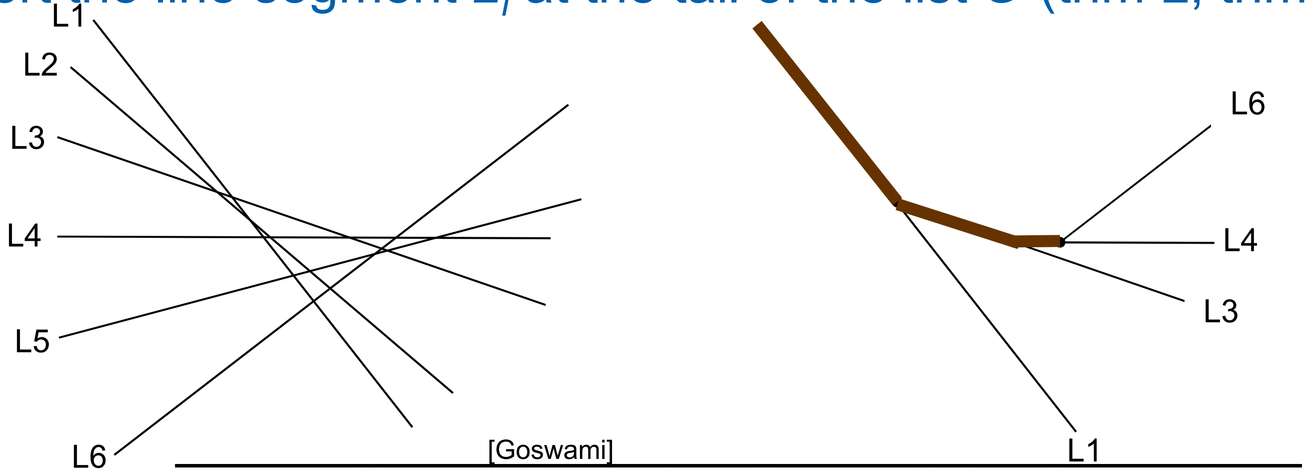




[Goswami]

Upper envelope algorithm

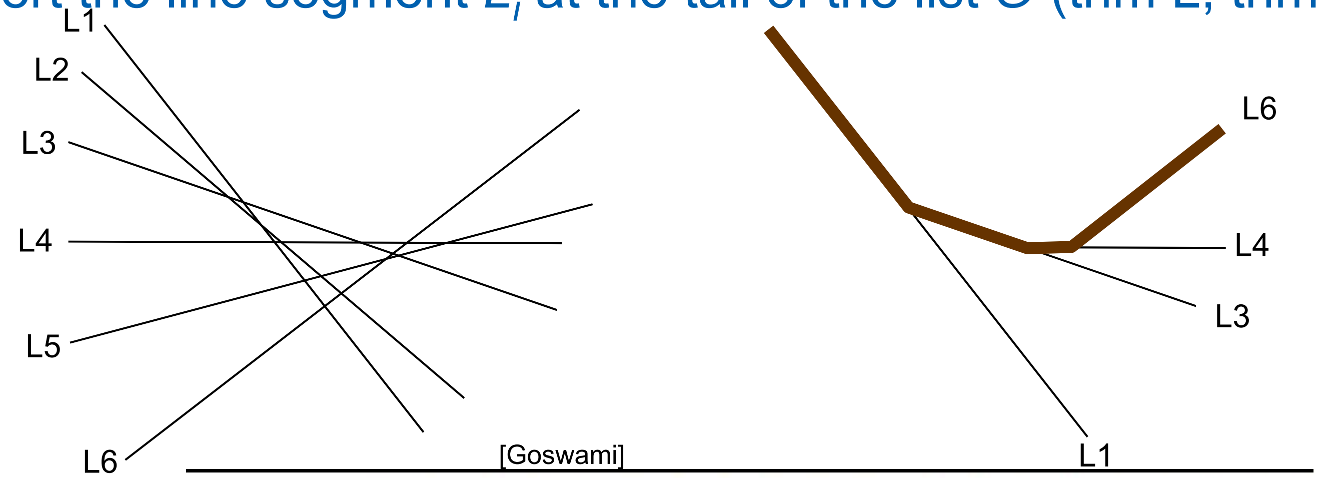
```
UpperEnvelope($L$)
        Set of lines $L$ sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                        // the only complete line in O
for $i$ = 2 to $n$
    L = last entry in O  // O contains half-lines, or line segments,
                         //  except of complete line L1
    while( the line segment $L$ does not intersect line $L_i$)
        remove $L$ from $O$ and replace $L$ with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list $O$ (trim L, trim $L_i$)
```



[Goswami]

Upper envelope algorithm

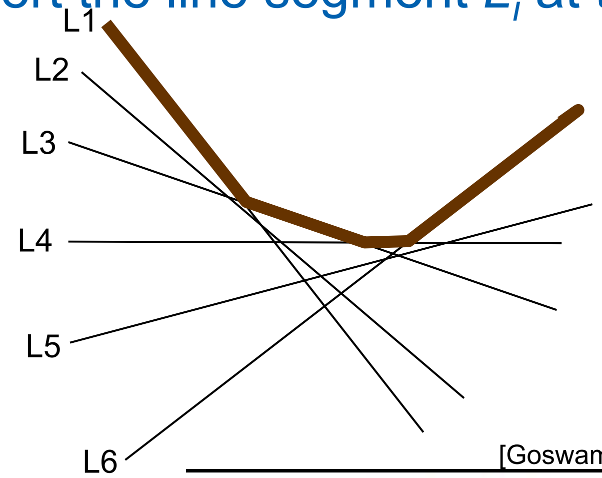
```
UpperEnvelope(L)
        Set of lines L sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                              // the only complete line in O
for i = 2 to n
    L = last entry in O   // O contains half-lines, or line segments,
                          //  except of complete line L1
    while( the line segment L does not intersect line $L_i$)
        remove L from O and replace L with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```



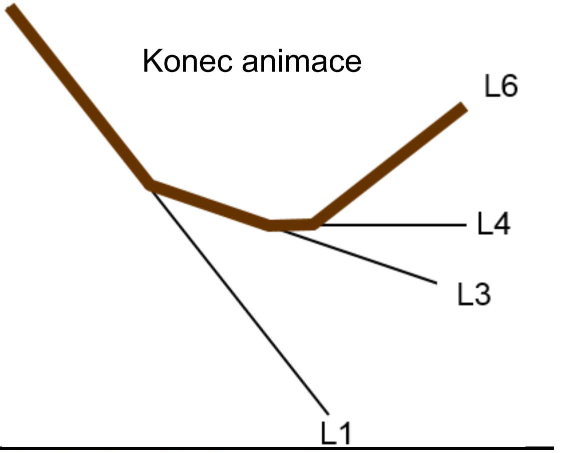
[Goswami]

Upper envelope algorithm

```
UpperEnvelope(L)
        Set of lines L sorted by increasing order of slopes (-90° to 90°)
        Polygonal chain O representing the upper hull
O = L1                          // the only complete line in O
for i = 2 to n
    L = last entry in O  // O contains half-lines, or line segments,
                         //  except of complete line L1
    while( the line segment L does not intersect line $L_i$)
        remove L from O and replace L with its predecessor    // L2, L5
    insert the line segment $L_i$ at the tail of the list O (trim L, trim $L_i$)
```



[Goswami]


Konec animace

Convex hull via upper and lower envelope

- Upper envelope complexity
  - After sorting n lines by their slopes in O(n log$n$) time, the upper envelope can be obtained in O(n) time
  - Proof: It may check more than one line segment when inserting a new line, but those ones checked are all removed except the last one.
    (O(n) insertions, max O(n) removals
    => O(n) all steps. Average step O(1) amortized time)
- Convex hull complexity
  - Given a set P of n points in the plane, CH(P) can be computed in O(n log n) time using O(n) space.

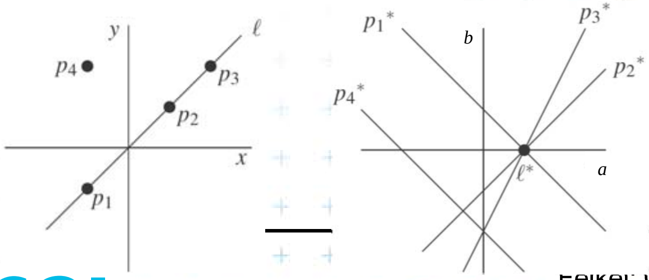
Applications of line arrangement

Examples of applications – solved in O(n^2) and O(n^2) space by constructing a line arrangement or O(*n*) space through topological plain sweep.

a) General position test:

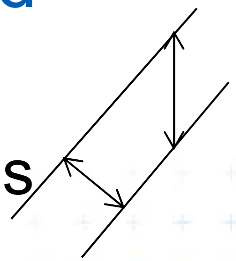
Given a set of *n* points in the plane, determine whether any three are collinear.

- Construct an arrangement in dual plane
- Report intersections of more than 2 lines



b) Minimum k-corridor

- Given a set of *n* points, and an integer *k* ∈ [1 *: n*], determine the narrowest pair of parallel lines that enclose at least *k* points of the set.
- The distance between the lines can be defined
  - either as the vertical distance between the lines
  - or as the perpendicular distance between the lines



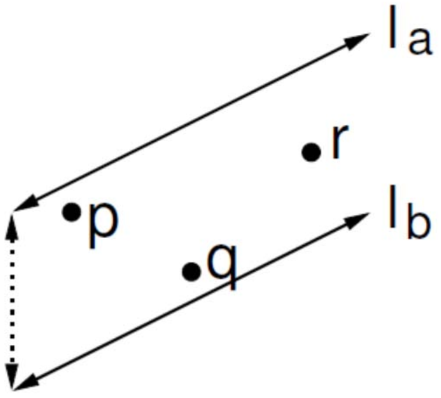
- Simplifications
  - Assume *k* = 3 and no 3 points are collinear

    => narrowest corridor - contains exactly 3 points
    - has width > 0

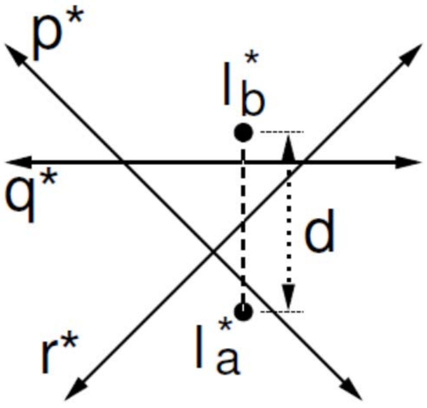
  vertical

  - No 2 points have the same *x* coordinate (avoid l duals)

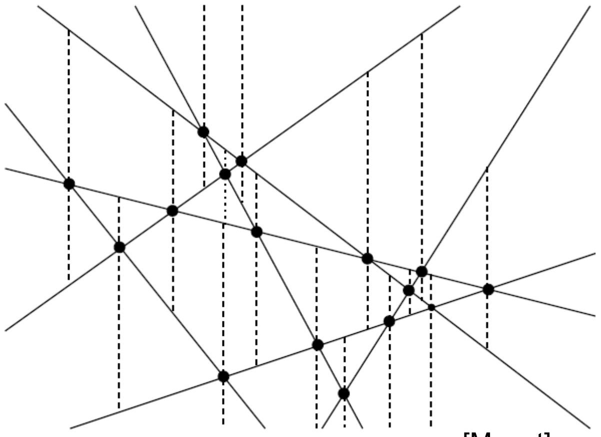
b) Minimum k-corridor



Primal Plane



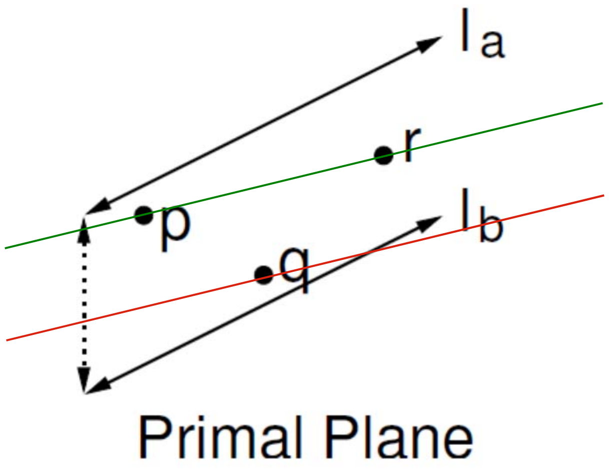
Dual Plane



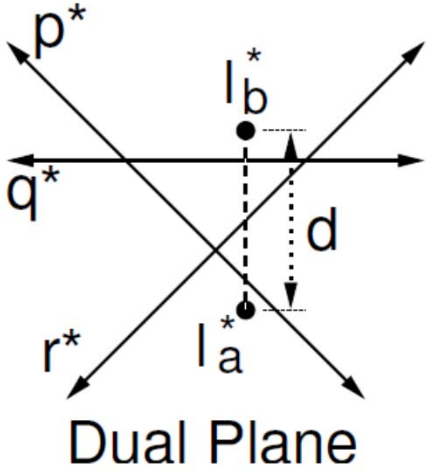
[Mount]

- Vertical distance of l_a, l_b = (-) distance of l_a^*, l_b^*
- Nearest lines – one passes 2 vertices, e.g., *p* & *r*
- In dual plane are represented as intersection $p^* \times r^*$
- Find nearest 3-stabber similarly as trapezoidal map
- $O(n^2)$ time and $O(n)$ space – topological line sweep

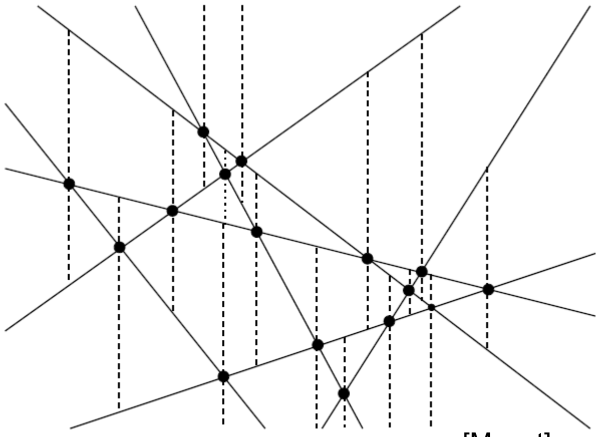
b) Minimum k-corridor



Primal Plane



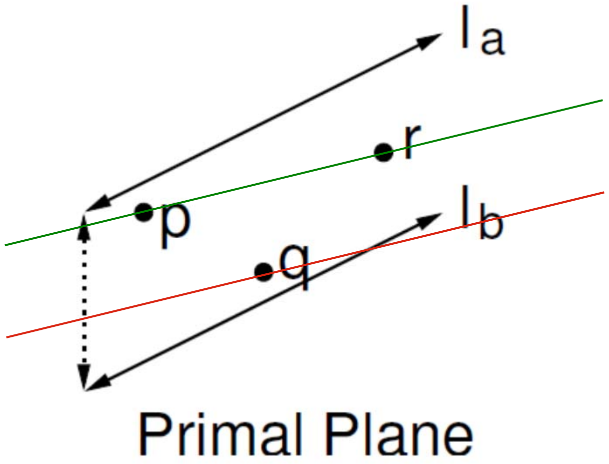
Dual Plane



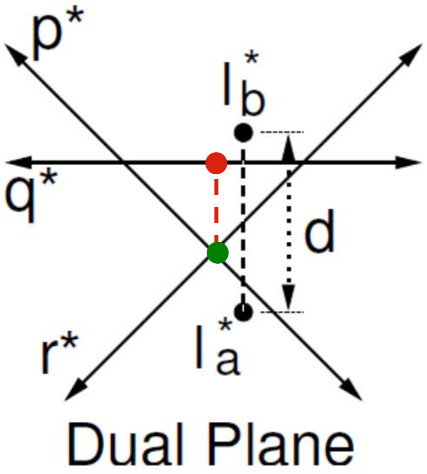
[Mount]

- $\color{red}{\text{Vertical distance}}$ of l_a,l_b = (-) distance of l_a^*,l_b^*
- Nearest lines – one passes 2 vertices, e.g., *p* & *r*
- In dual plane are represented as intersection $p^* \times r^*$
- Find nearest 3-stabber similarly as trapezoidal map
- O(n^2) time and O(n) space – topological line sweep

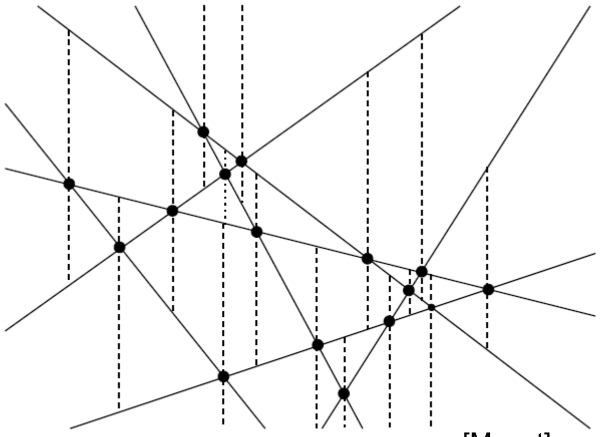
b) Minimum k-corridor



Primal Plane



Dual Plane

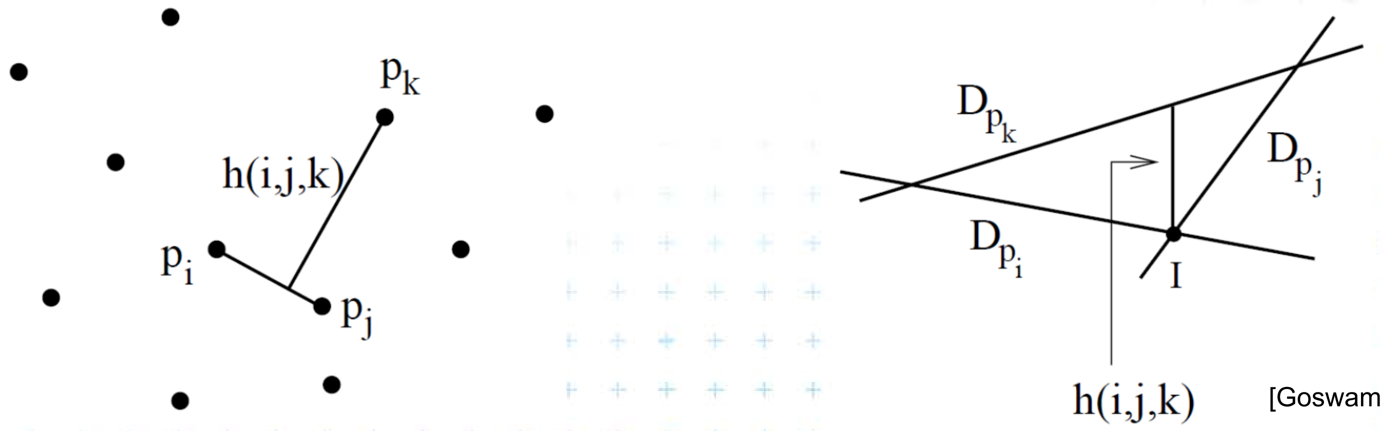


[Mount]

- Vertical distance of l_a, l_b = (-) distance of l_a^*, l_b^*
- Nearest lines – one passes 2 vertices, e.g., *p* & *r*
- In dual plane are represented as intersection $p^* \times r^*$
- Find nearest 3-stabber similarly as trapezoidal map
- $O(n^2)$ time and $O(n)$ space – topological line sweep

c) Minimum area triangle [Goswami]

- Given a set of *n* points in the plane, determine the minimum area triangle whose vertices are selected from these points
- Construct “trapezoids” as in the nearest corridor
- Minimize perpendicular distances (converted from vertical) multiplied by the distance from p_i to p_j



[Goswami]

d) Sorting all angular sequences – naïve

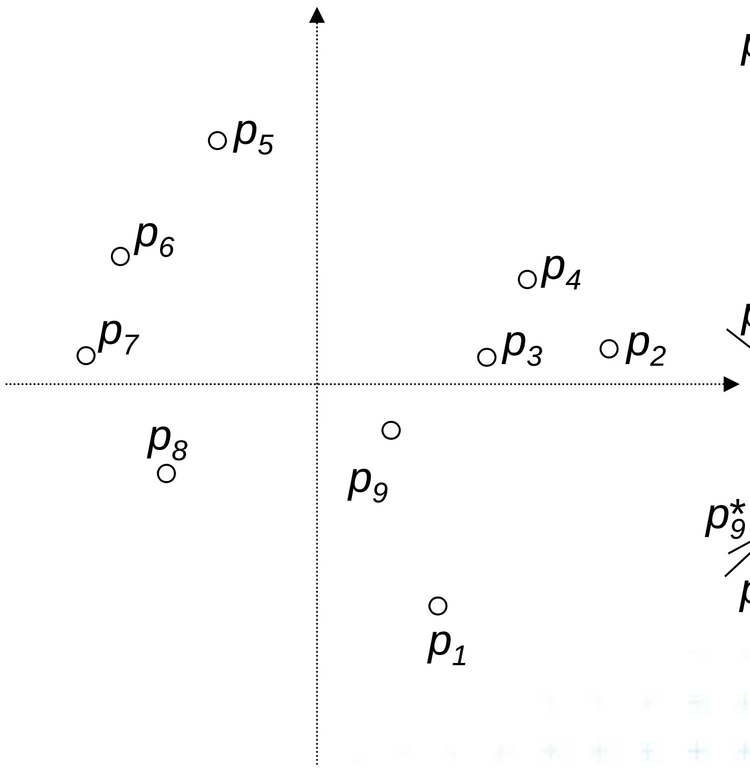
- Natural application of duality and arrangements
- Important for visibility graph computation
- Set of *n* points in the plane
- For each point perform an CCW angular sweep
- Naïve: for each point compute angles to remaining *n* – 1 points and sort them
- => $O(n \log n)$ time per point
- $O(n^2 \log n)$ time overall
- Arrangements can get rid of O(log *n*) factor

d) Sorting all angular sequences – optimal

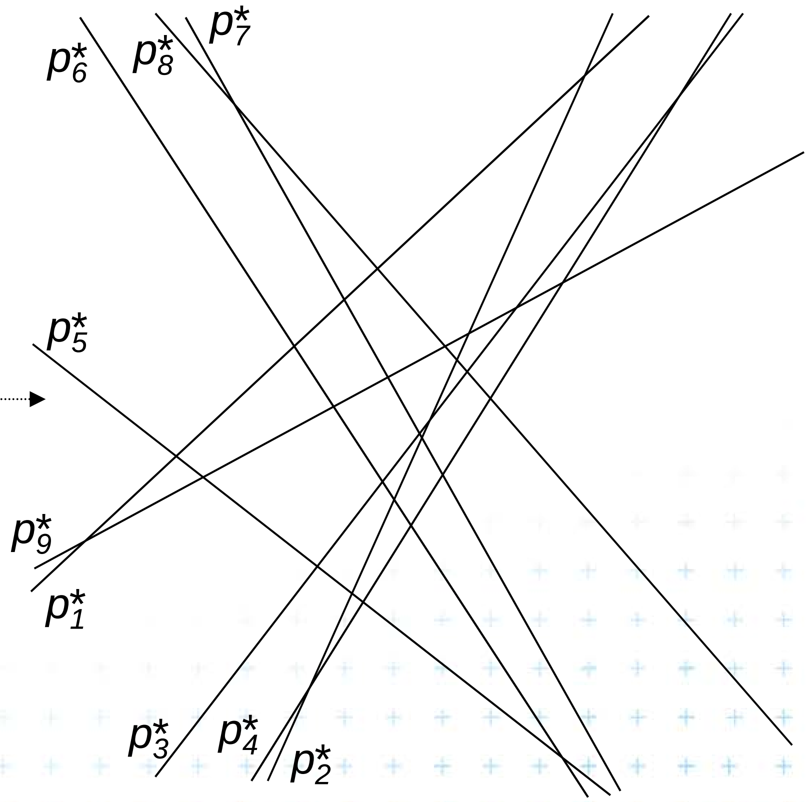
- For point p_i
  - Dual of point p_i is line p_i*
  - Line p_i* intersects other dual lines in order of slope (angles from -90° to 90°) (180°)
  - We need order of angles around p_i (angles from -90° to 270°) (360°)

  - Split points in primal plane by vertical line through p_i
  - First, report intersections of points right of p_i
  - Second, report the intersections of points left of p_i
  - Once the arrangement is constructed:
    $O(n)$ time for point, $O(n^2)$ time for all n points

d) Angular sequence around p_9

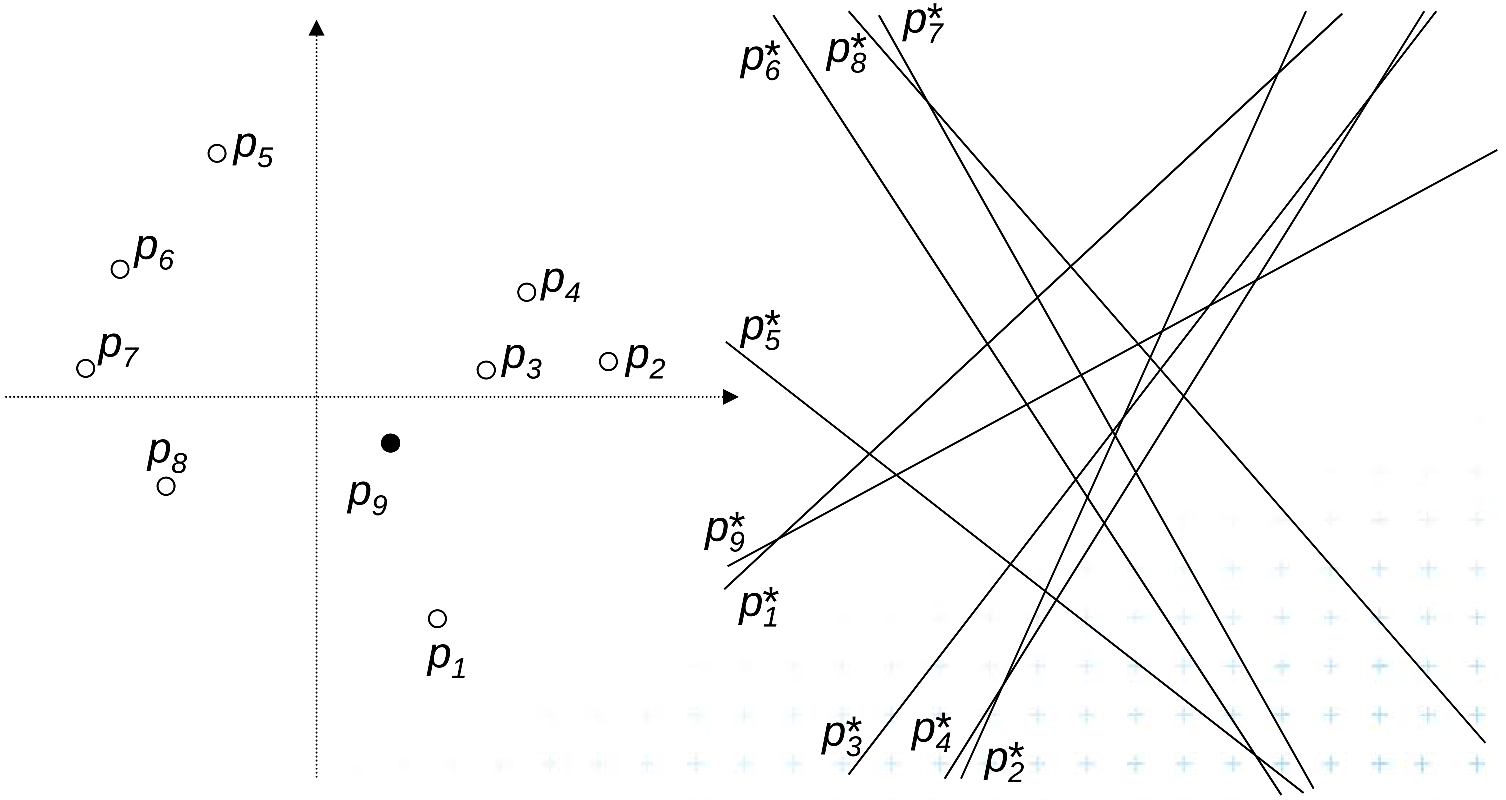


In primal plane



In dual plane

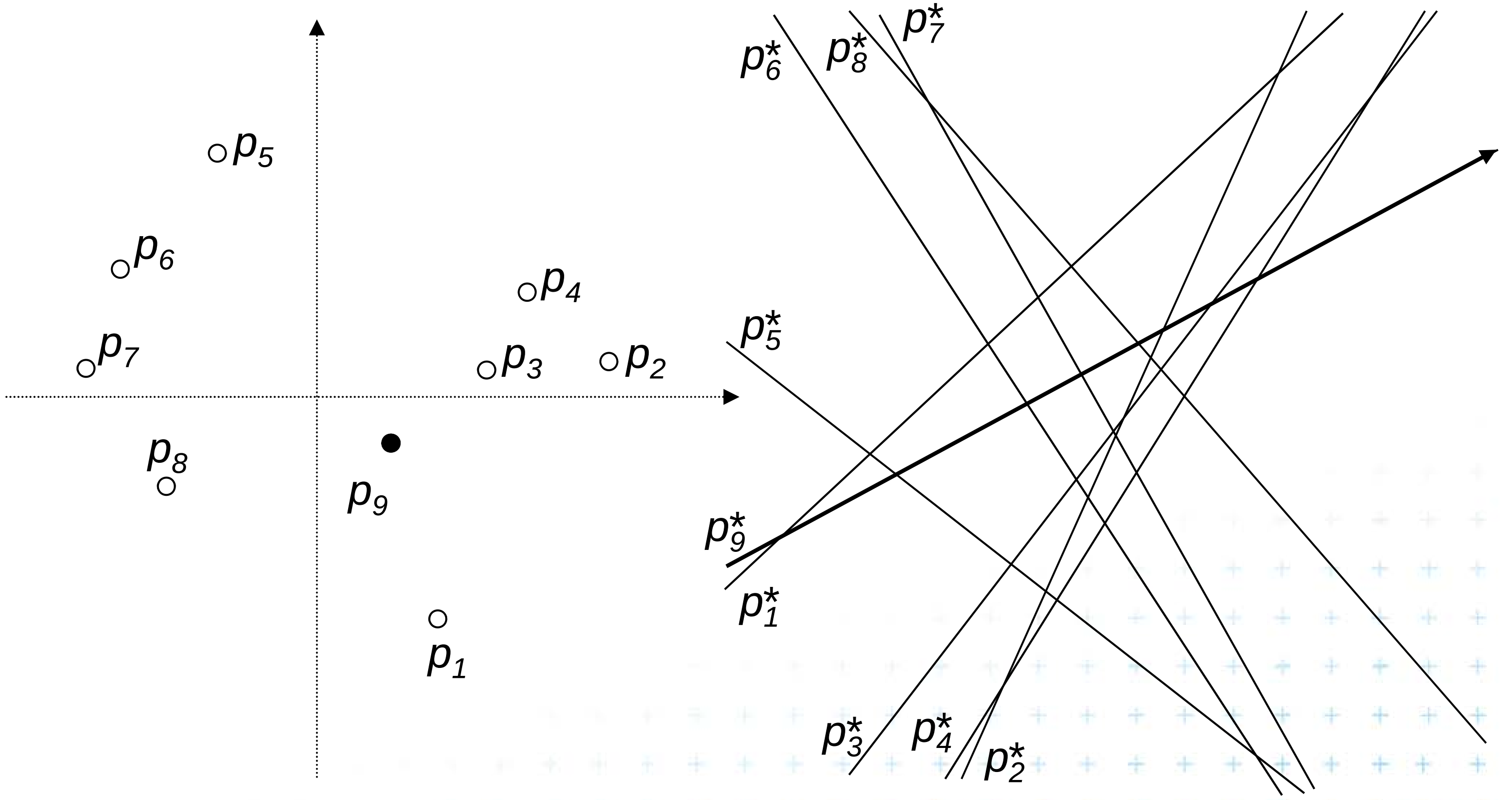
d) Angular sequence around p_9



In primal plane

In dual plane

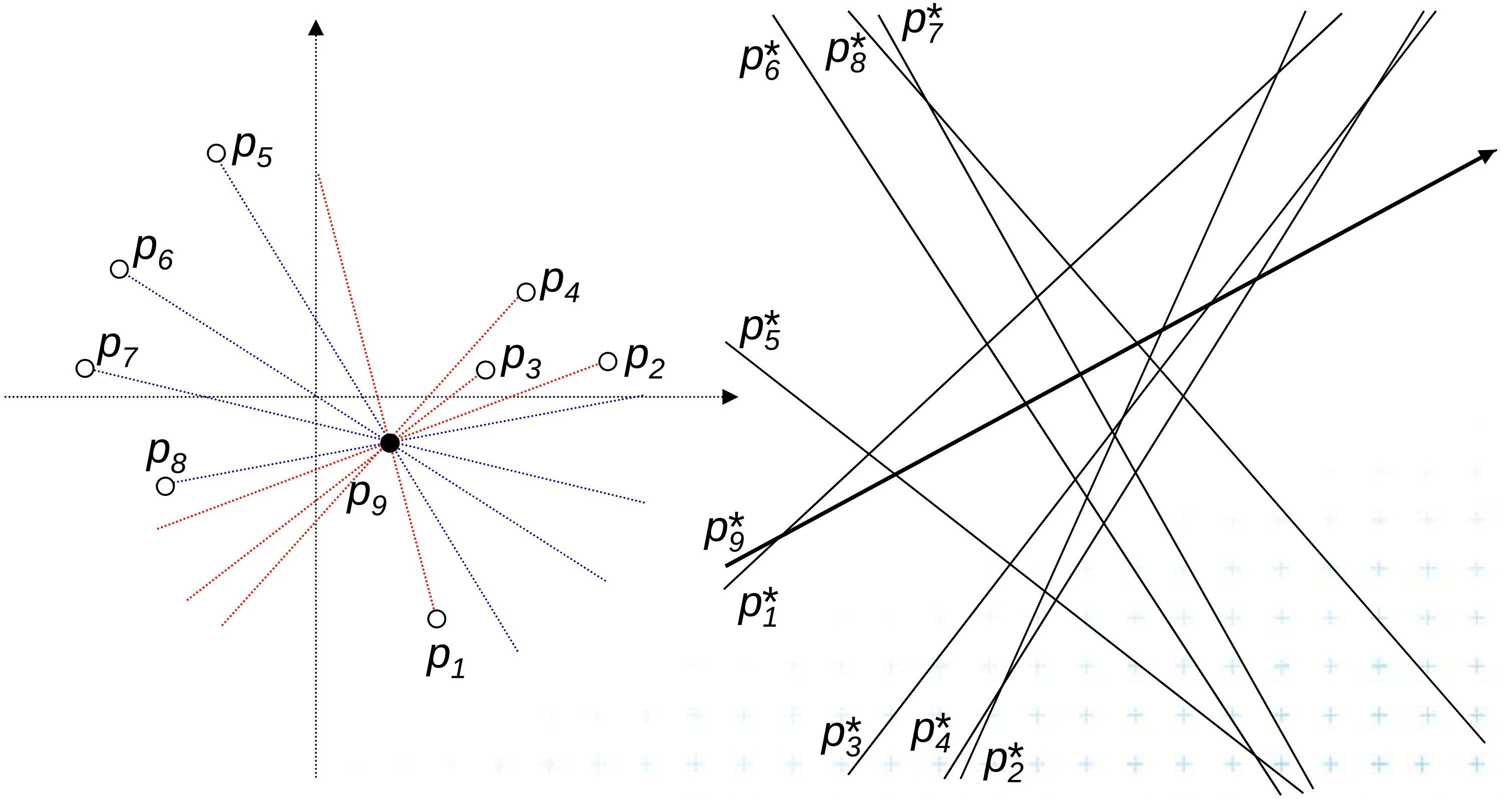
d) Angular sequence around p_9



In primal plane

In dual plane

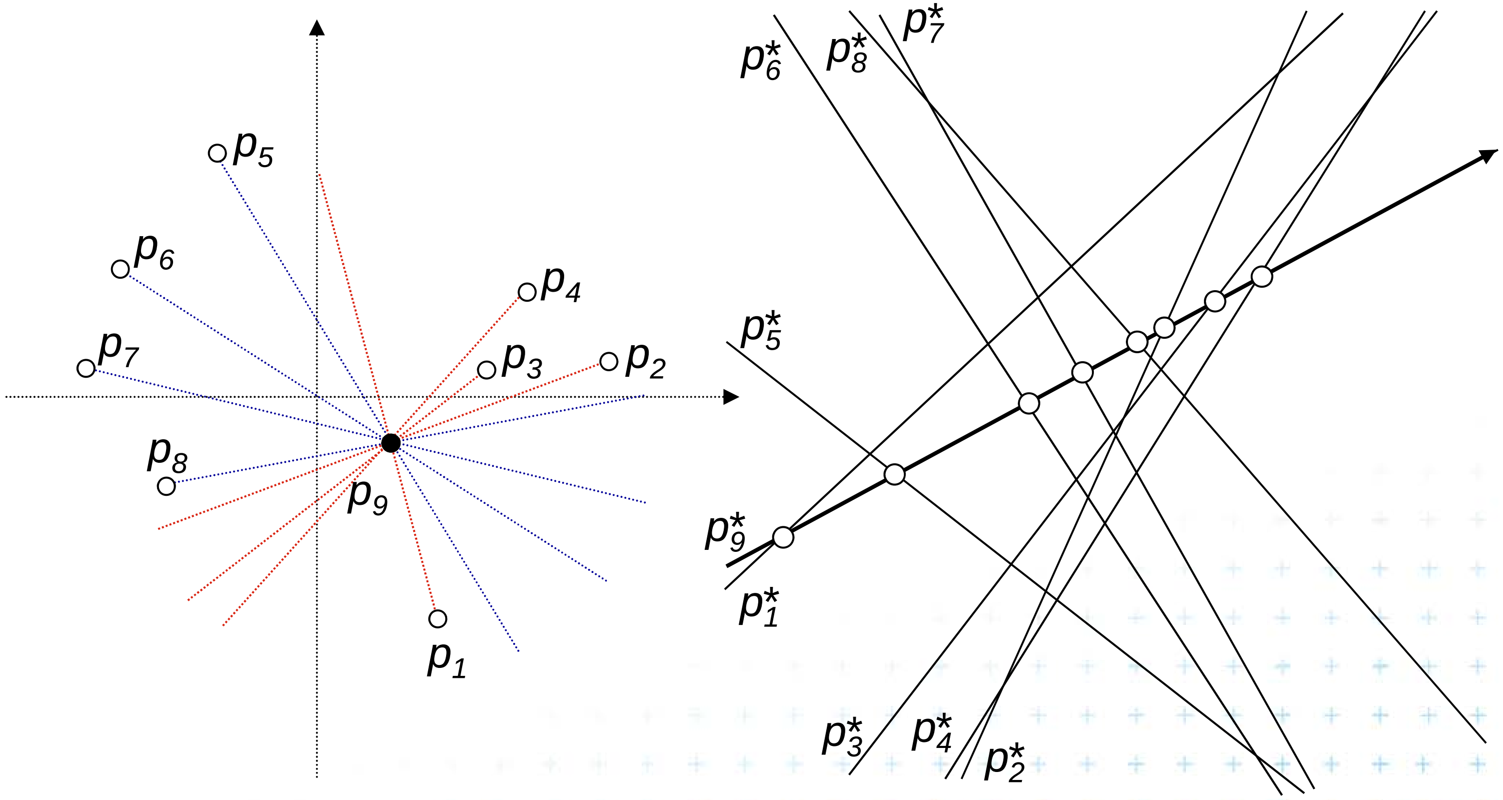
d) Angular sequence around p_9



In primal plane

In dual plane

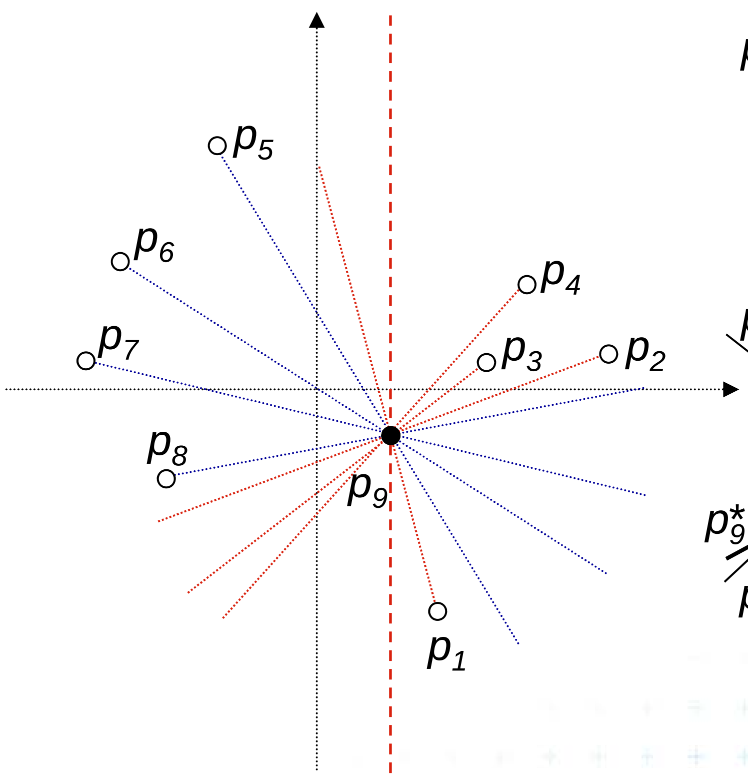
d) Angular sequence around p_9

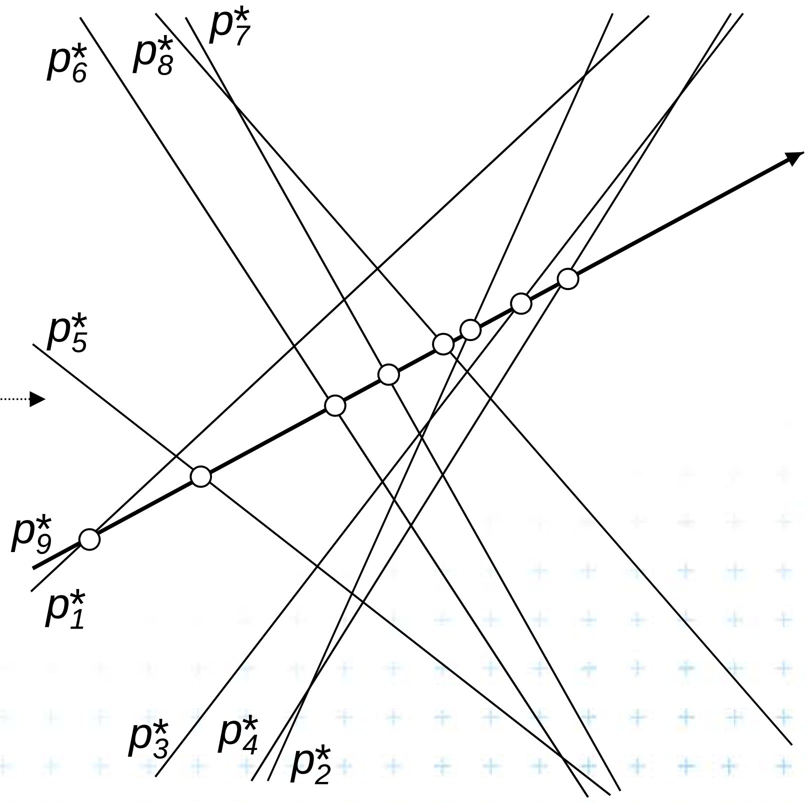


In primal plane

In dual plane

d) Angular sequence around p_9

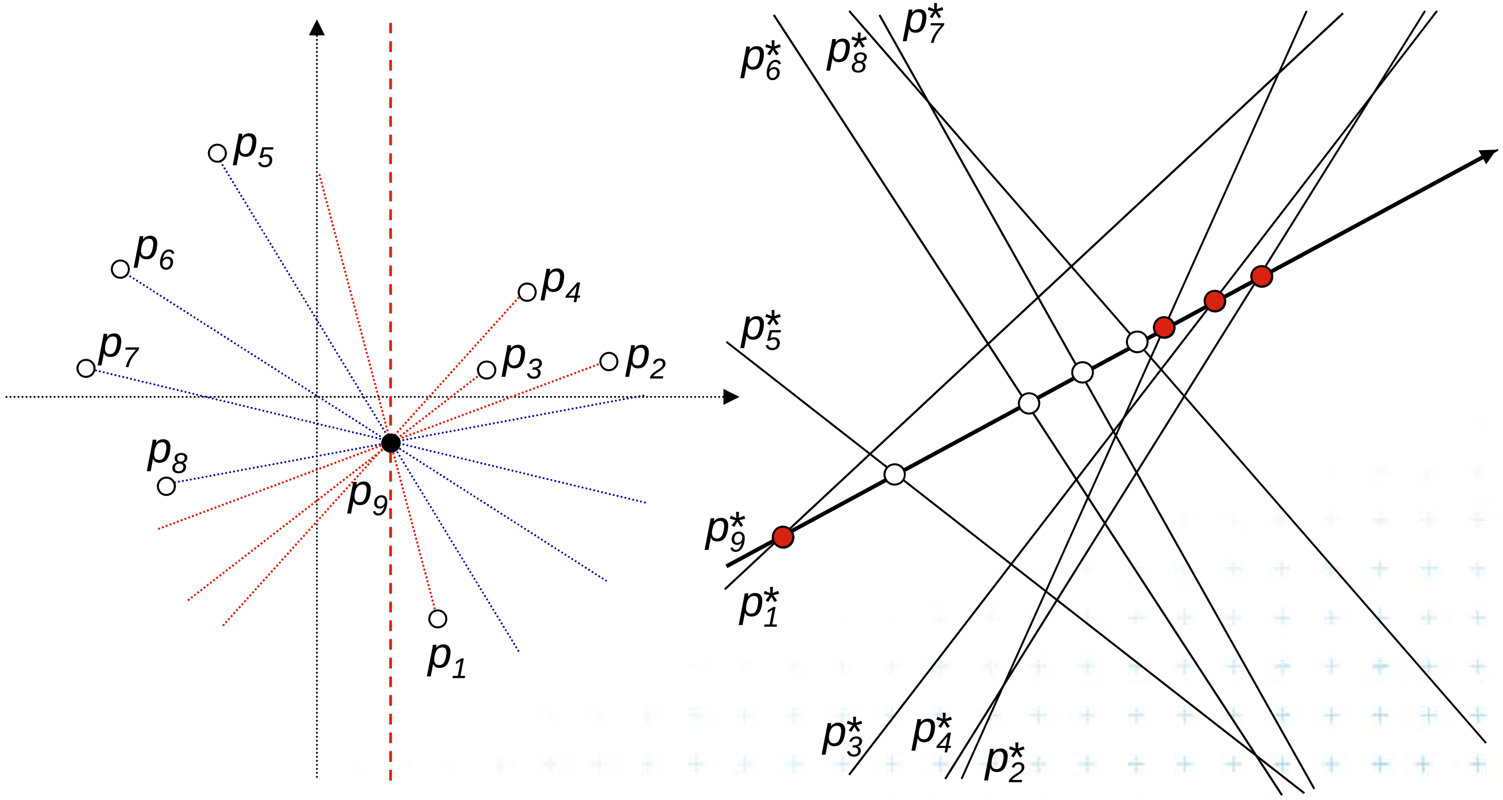




In primal plane

In dual plane

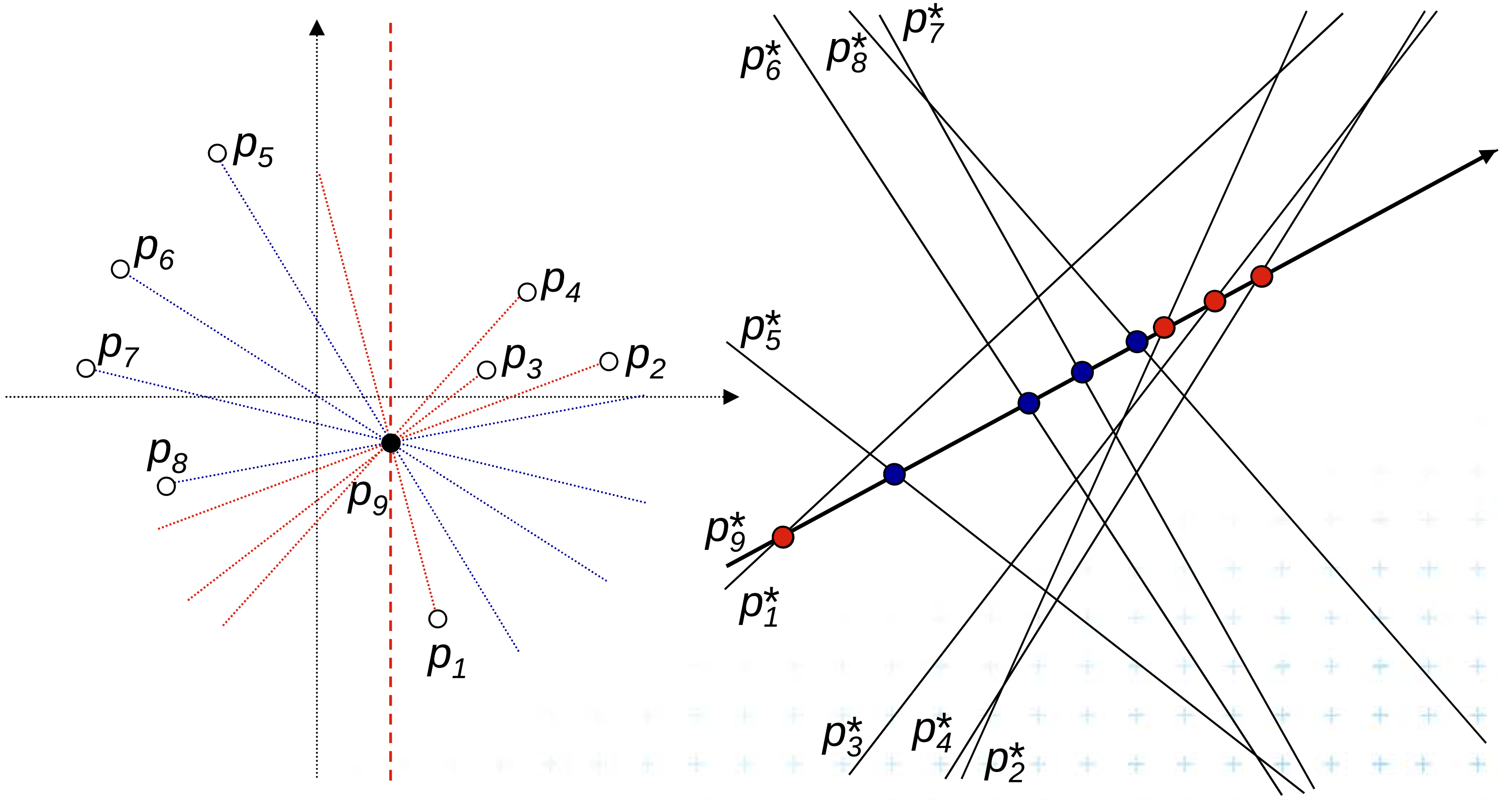
d) Angular sequence around p_9



In primal plane

In dual plane

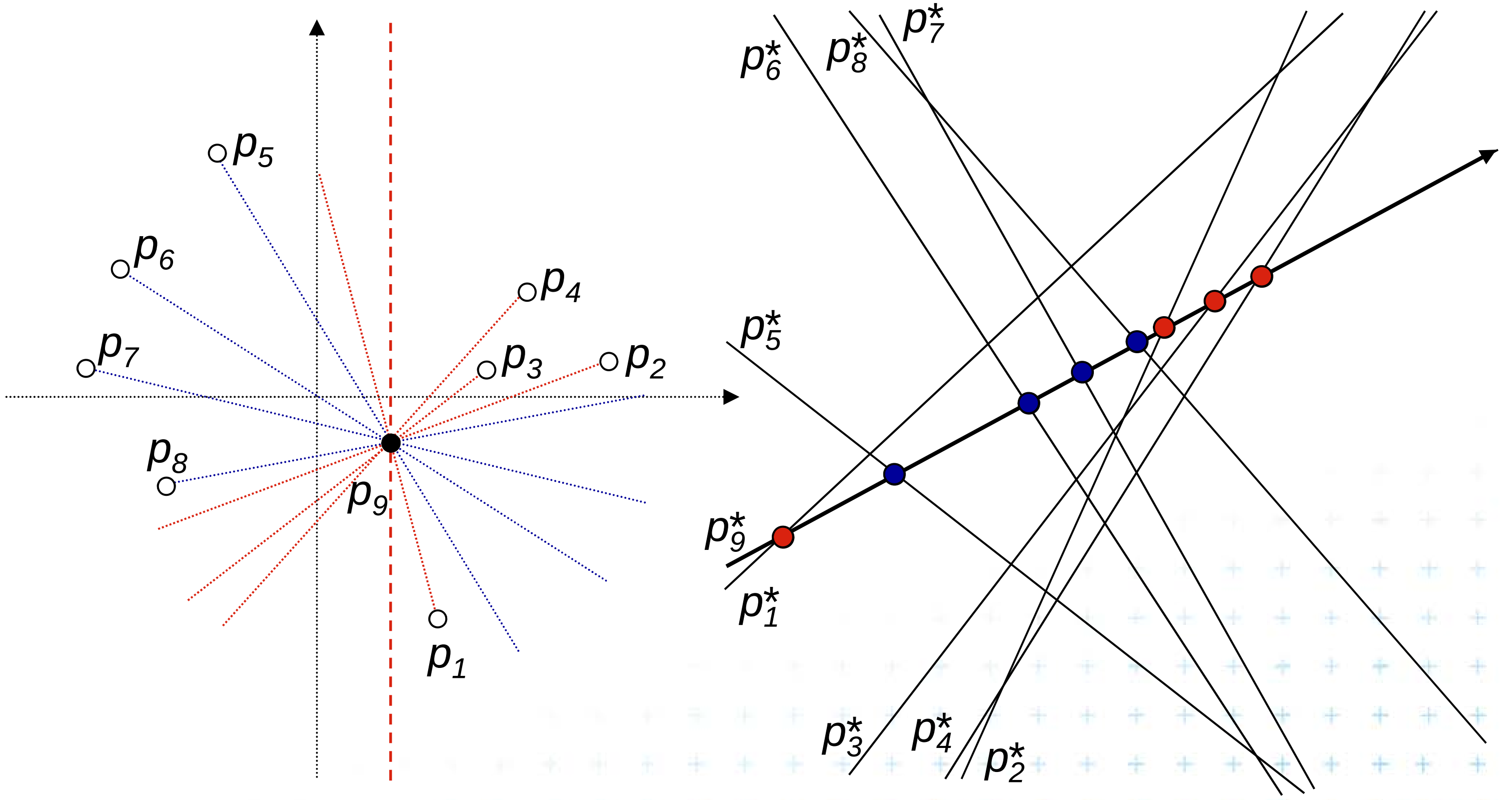
d) Angular sequence around p_9



In primal plane

In dual plane

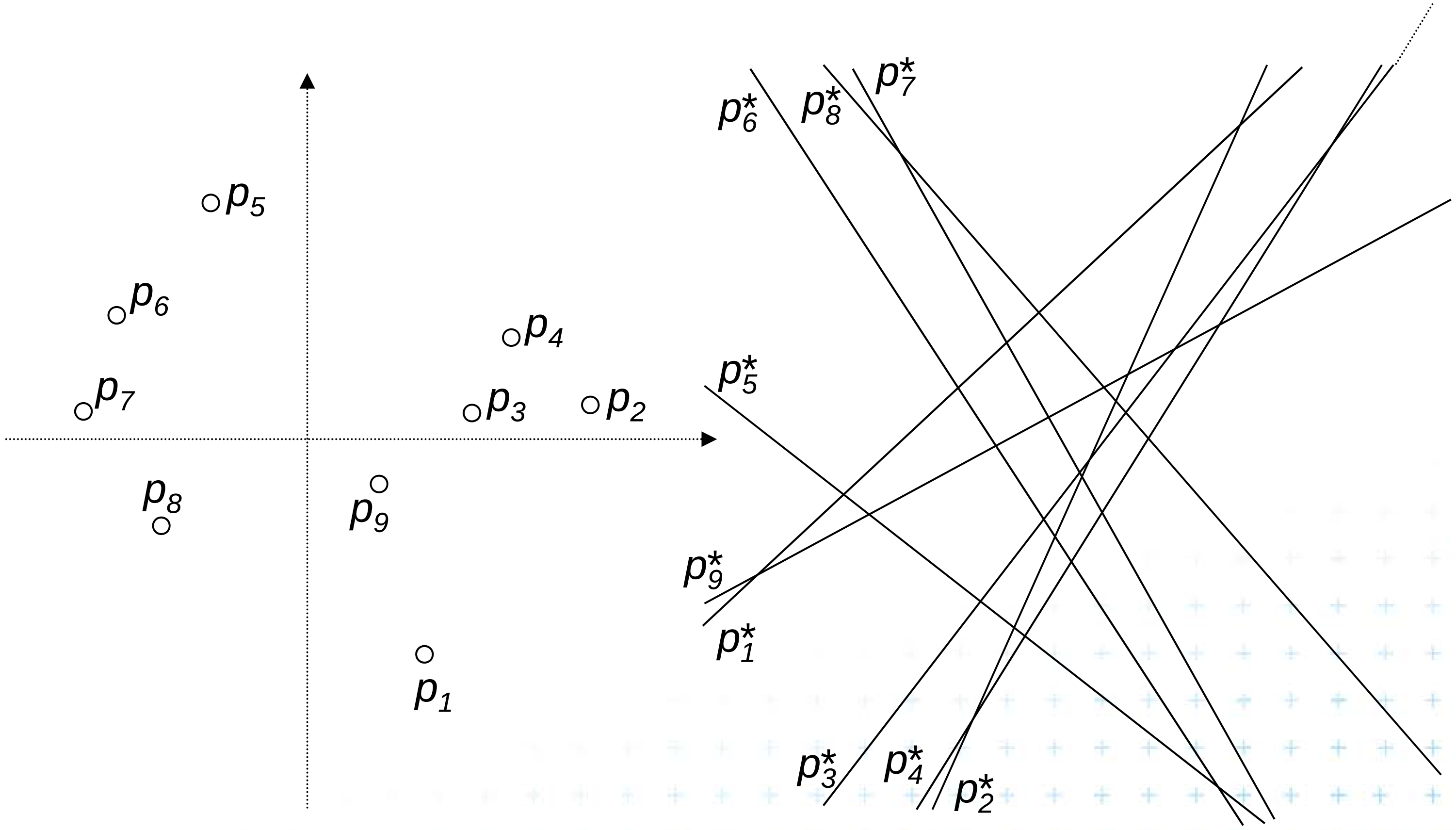
d) Angular sequence around p_9



In primal plane In dual plane

Point order around p_9 : p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8

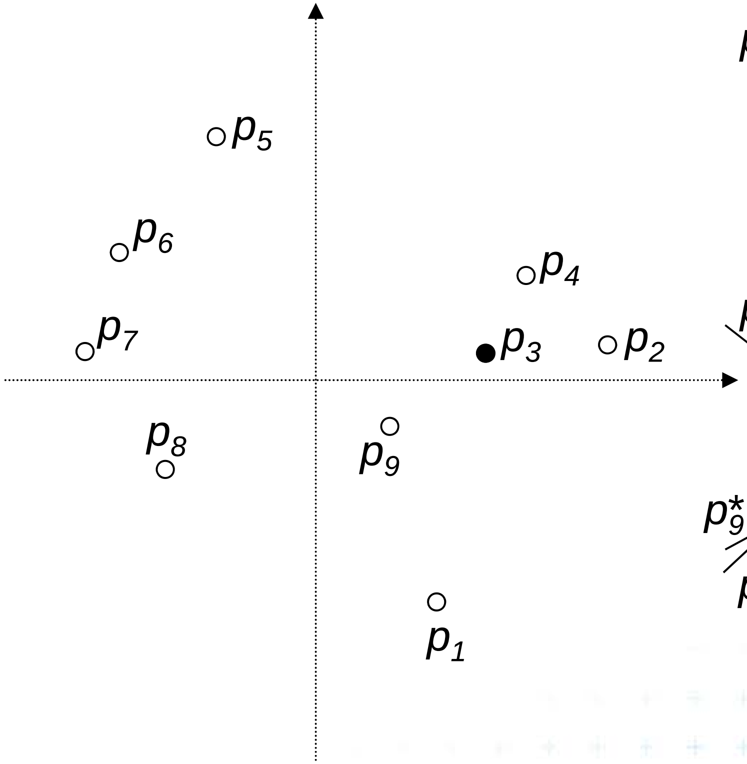
d) Angular sequences around p_3

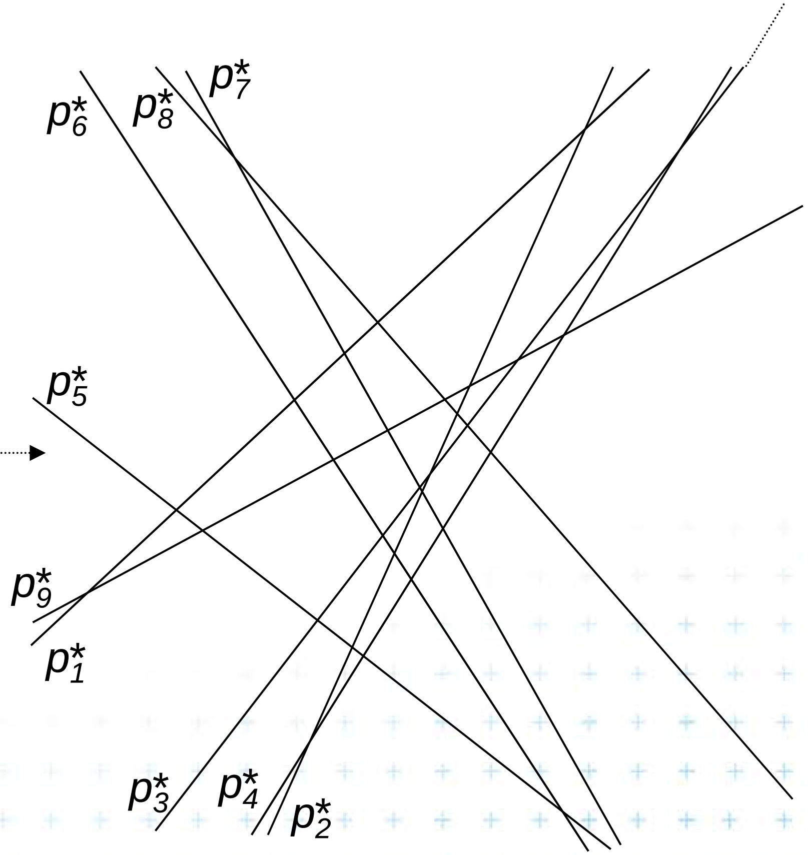


In primal plane

In dual plane

d) Angular sequences around p_3

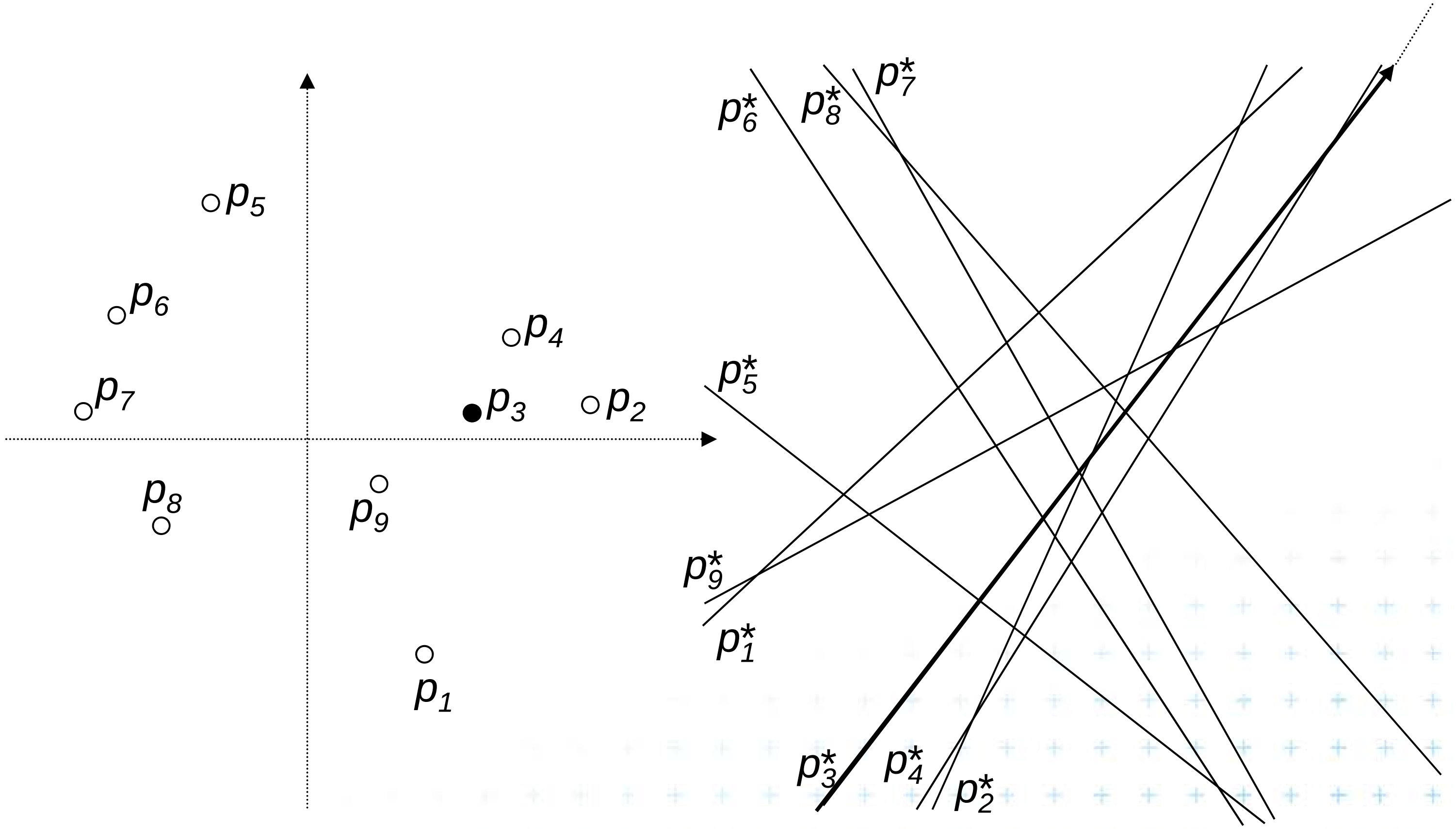




In primal plane

In dual plane

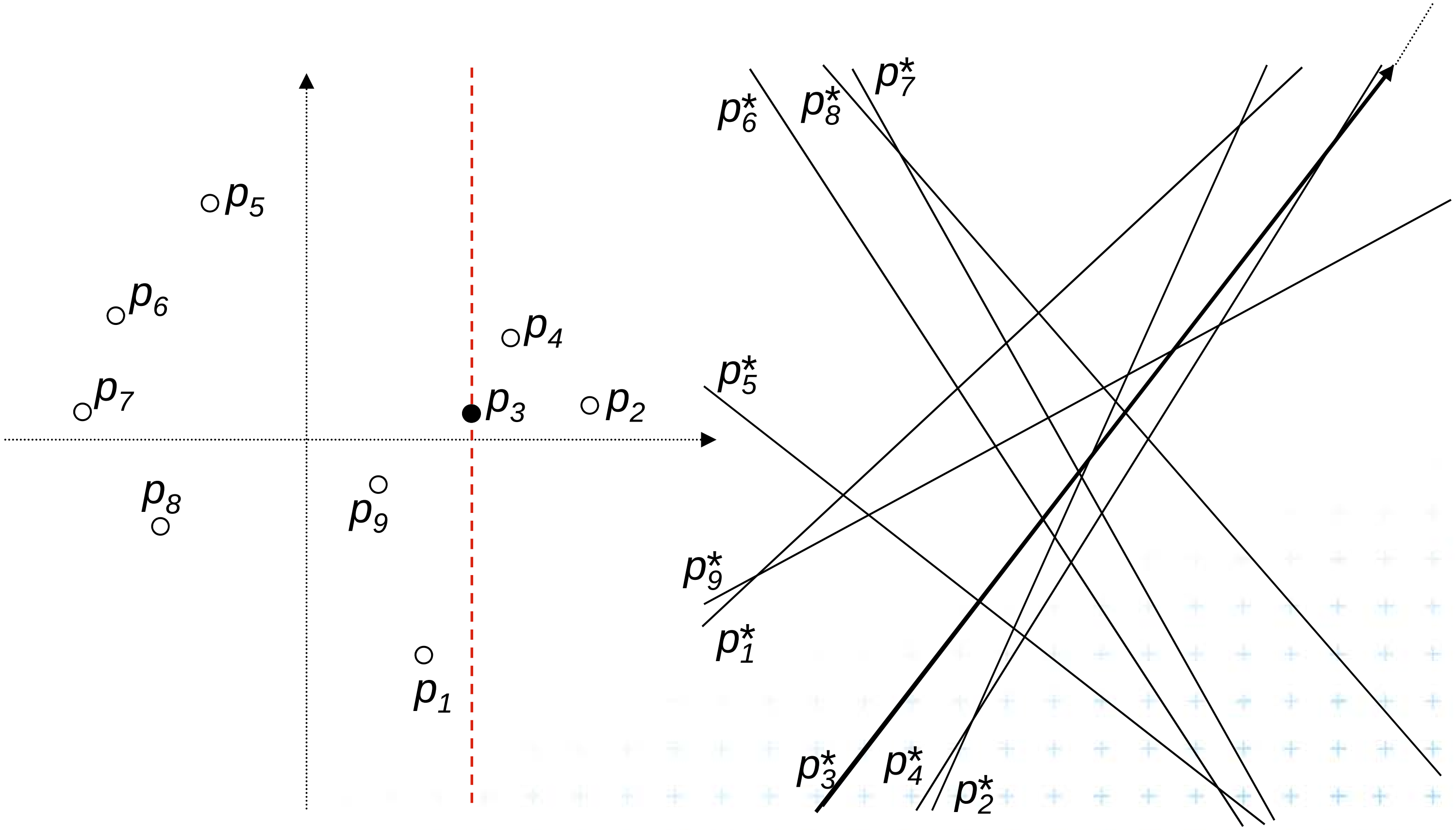
d) Angular sequences around p_3



In primal plane

In dual plane

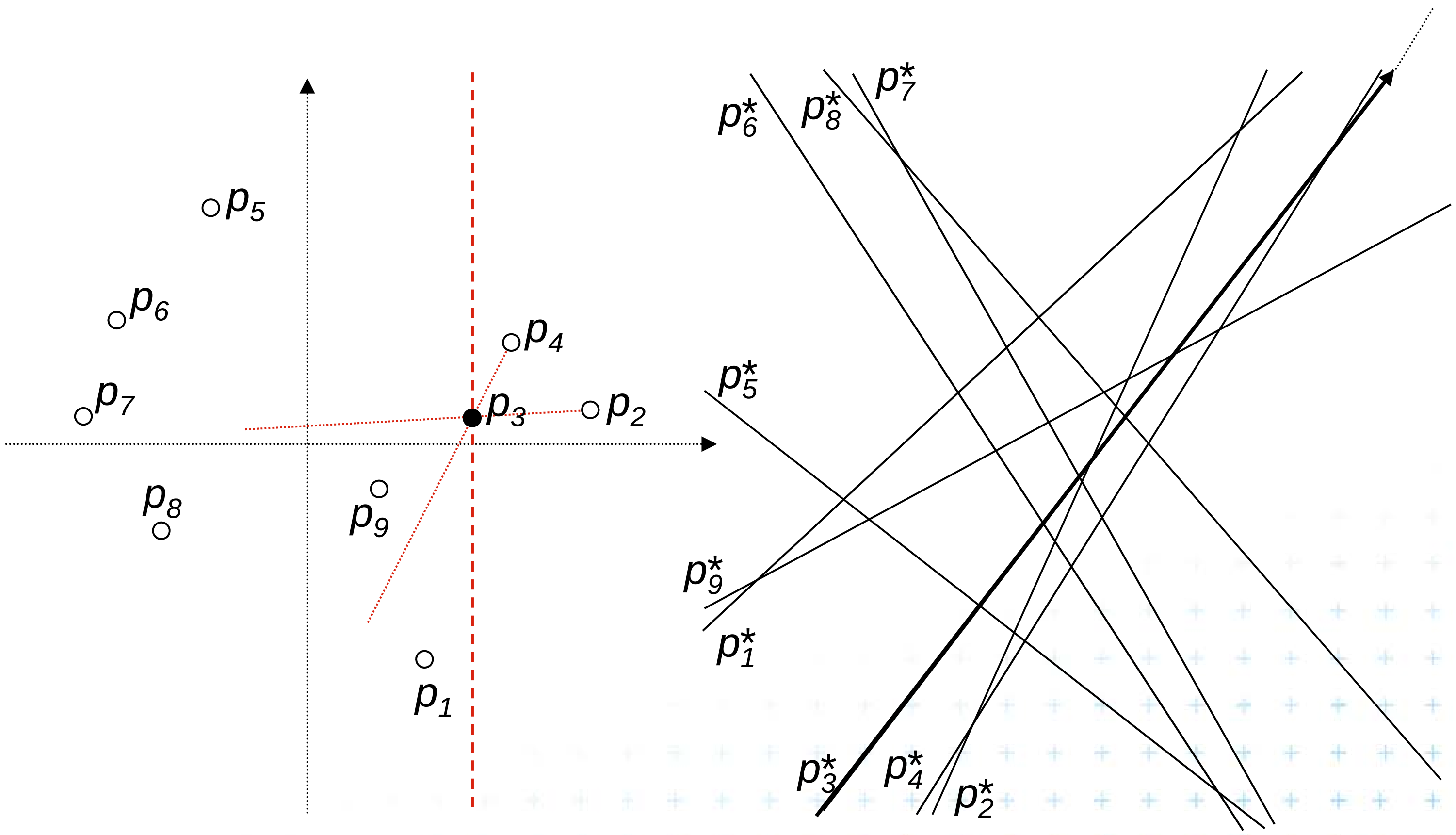
d) Angular sequences around p_3



In primal plane

In dual plane

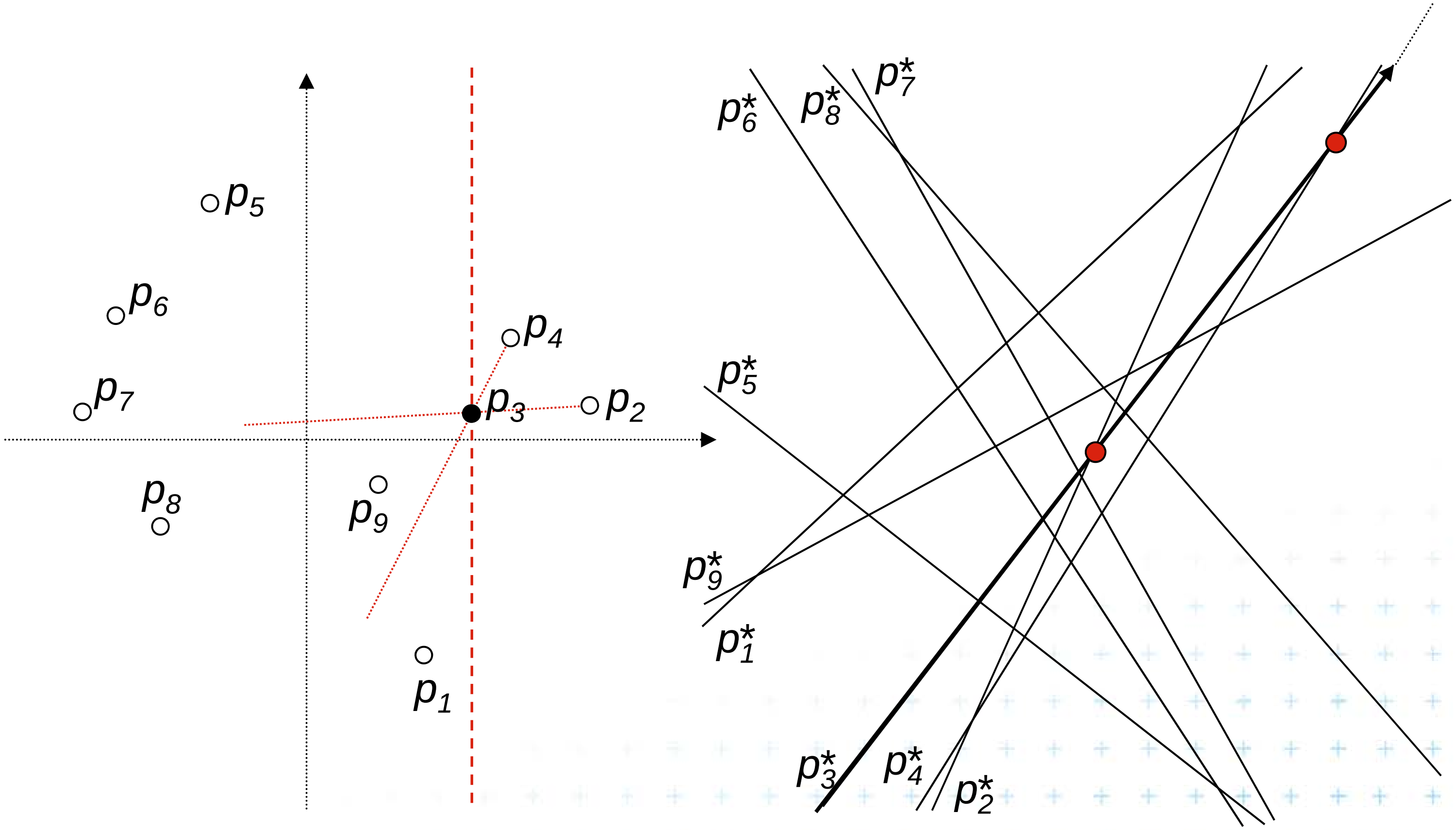
d) Angular sequences around p_3



In primal plane

In dual plane

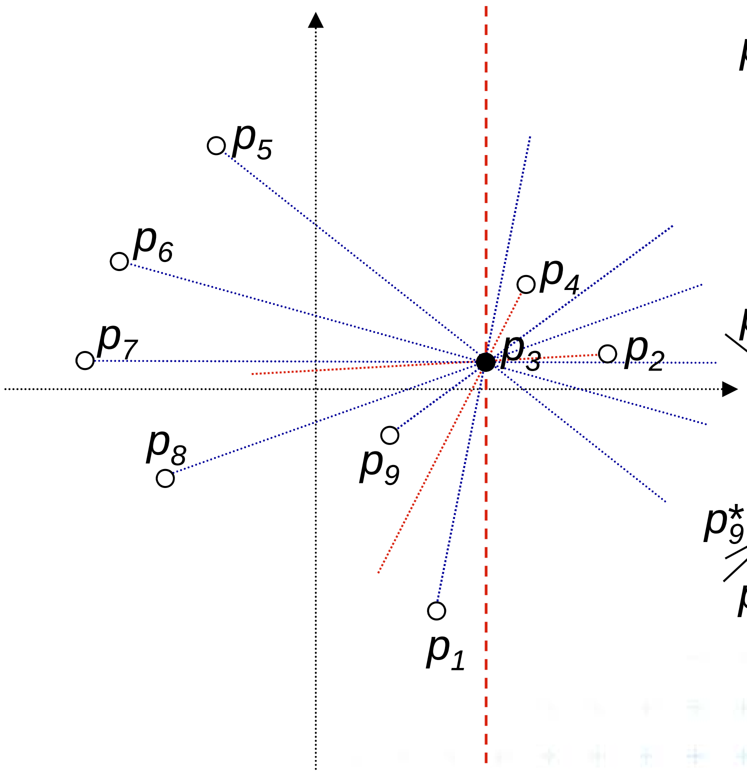
d) Angular sequences around p_3

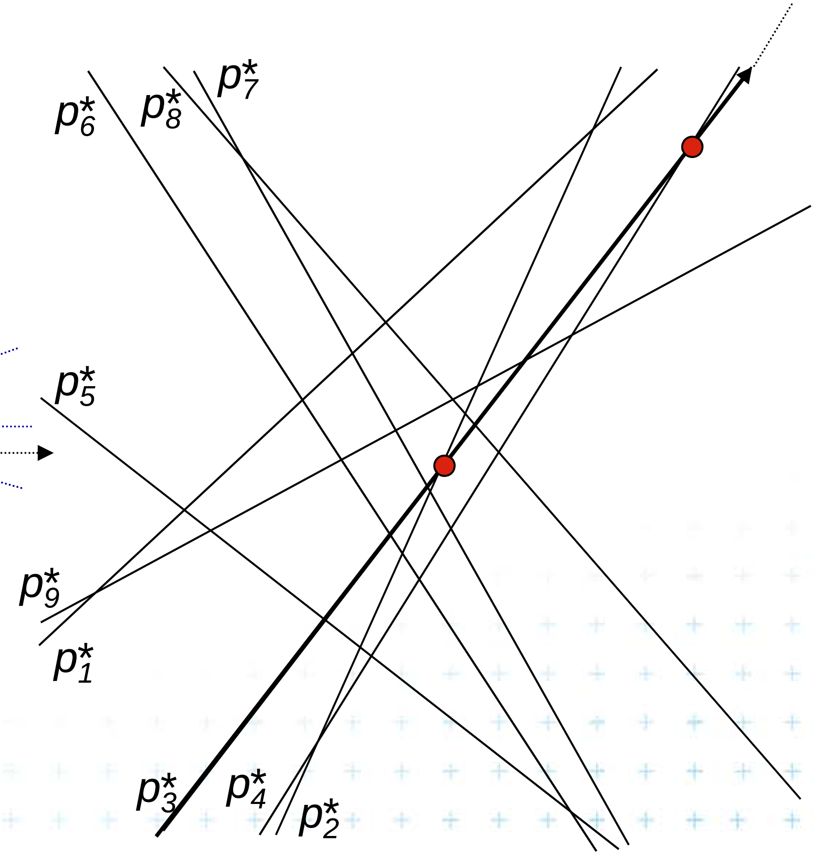


In primal plane

In dual plane

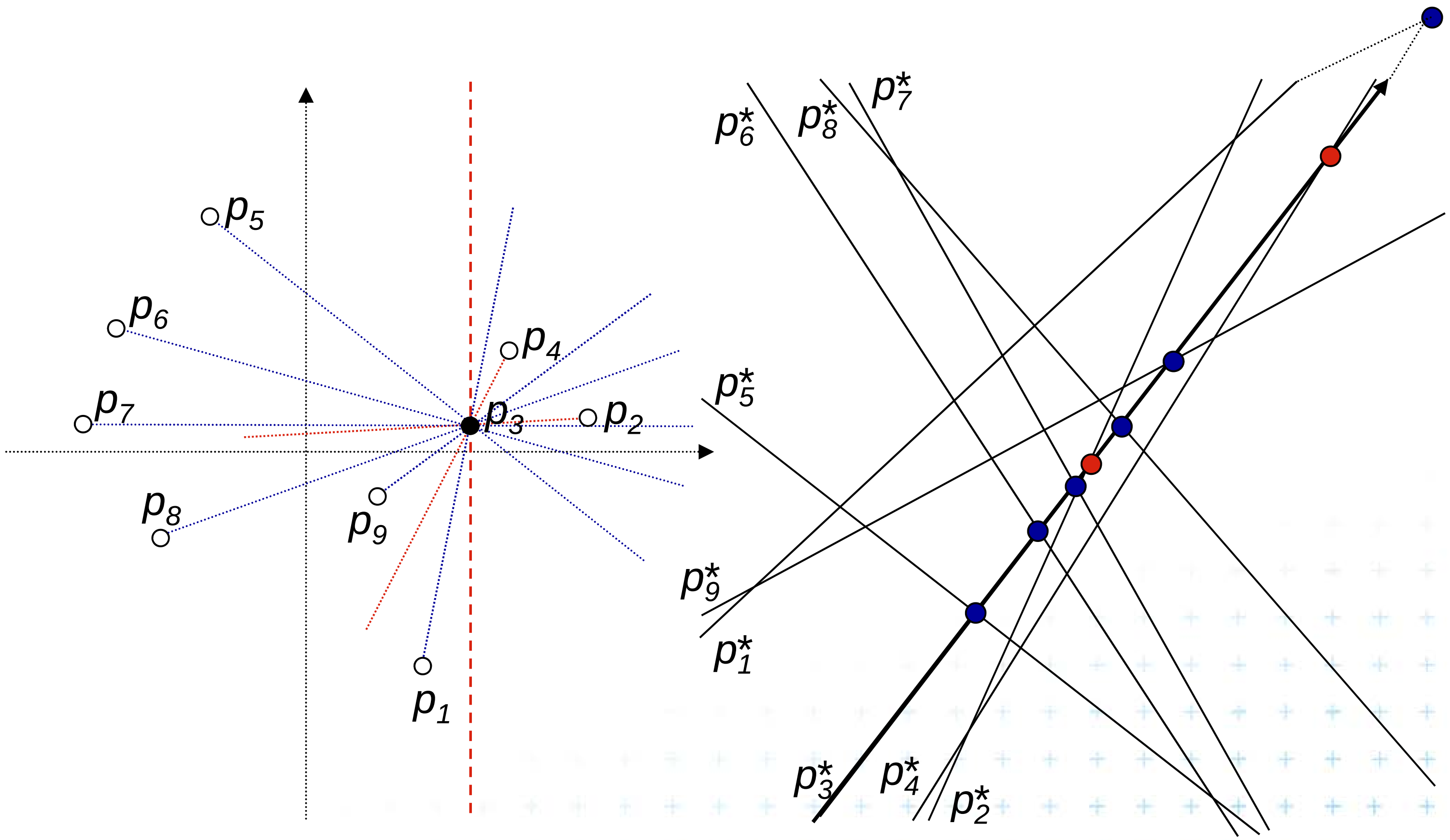
d) Angular sequences around p_3





In primal plane In dual plane

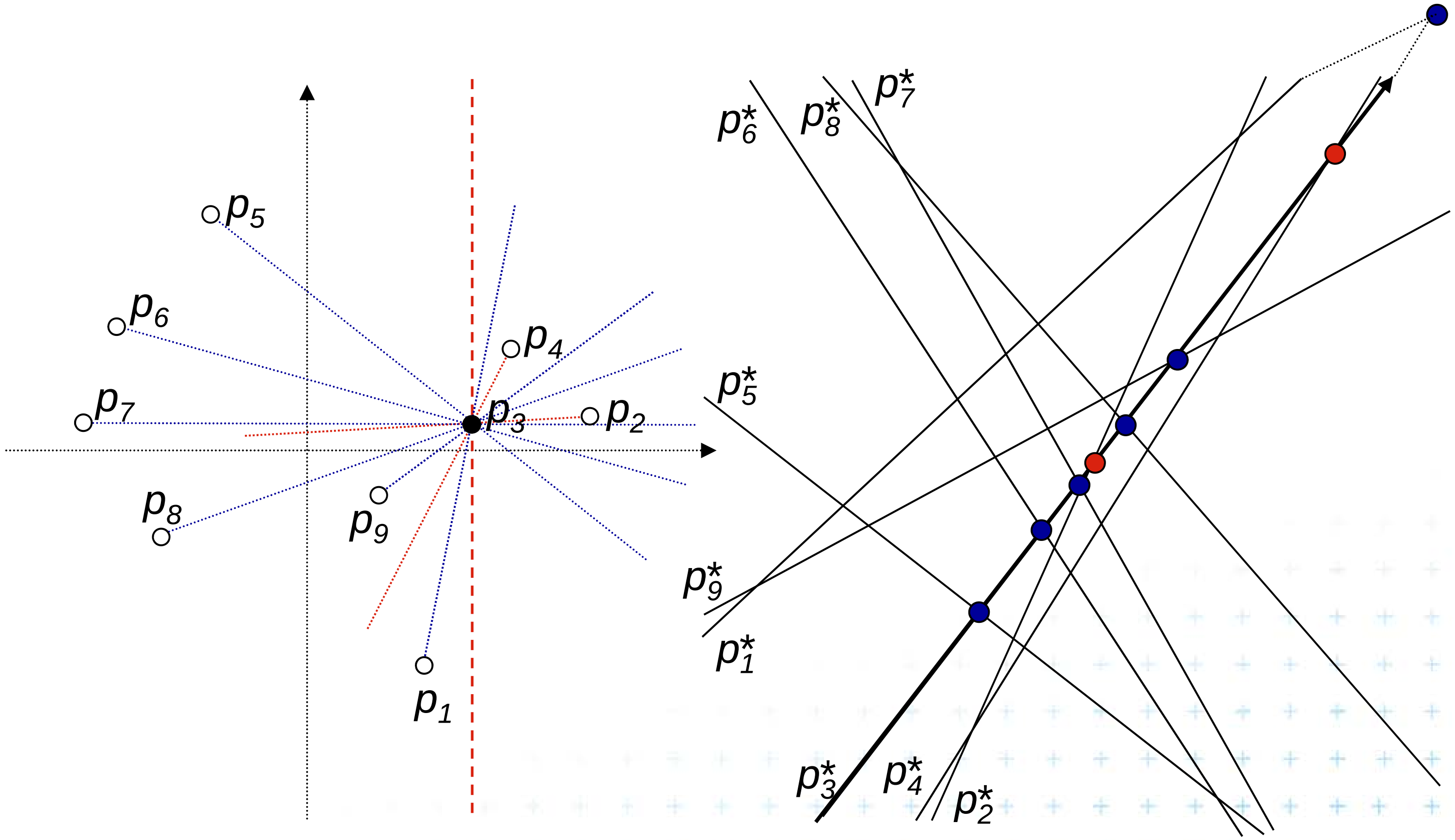
d) Angular sequences around p_3



In primal plane

In dual plane

d) Angular sequences around p_3

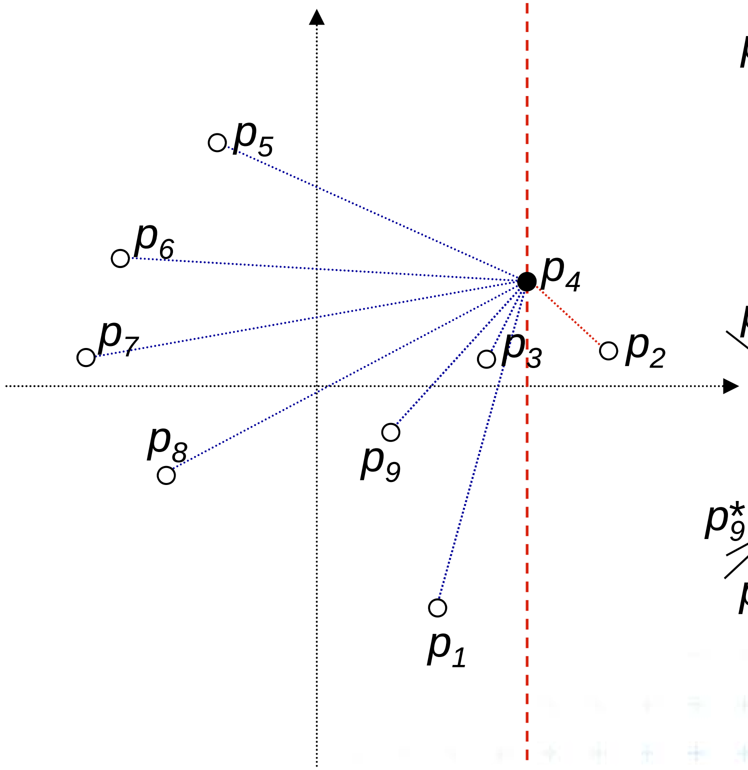


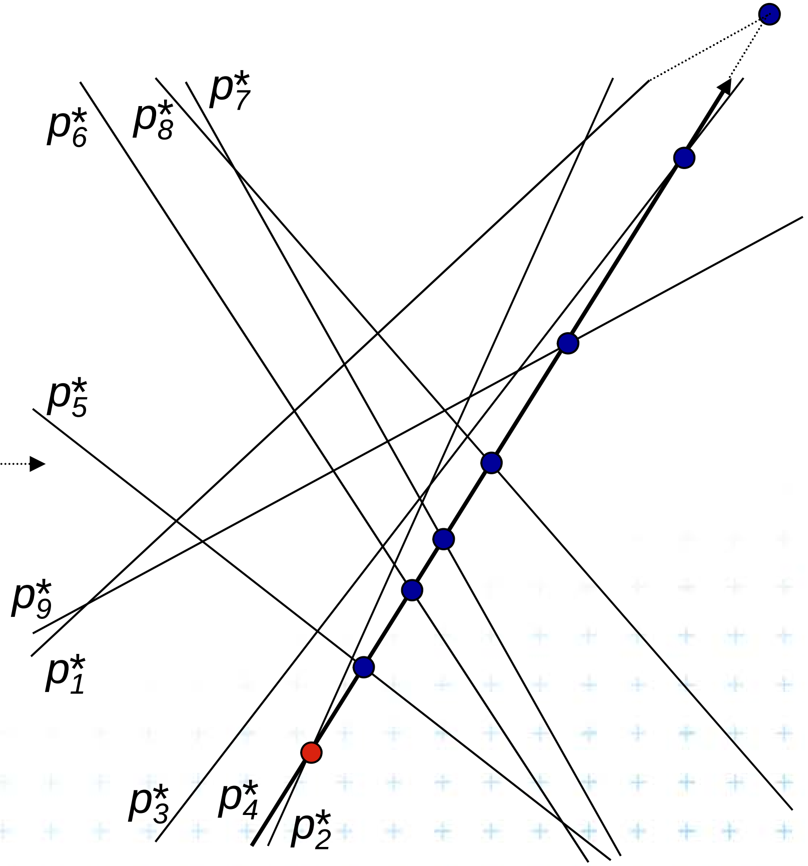
In primal plane

In dual plane

Point order around p_3 : p_2, p_4, p_5, p_6, p_7, p_8, p_3, p_1

d) Angular sequences around p_4

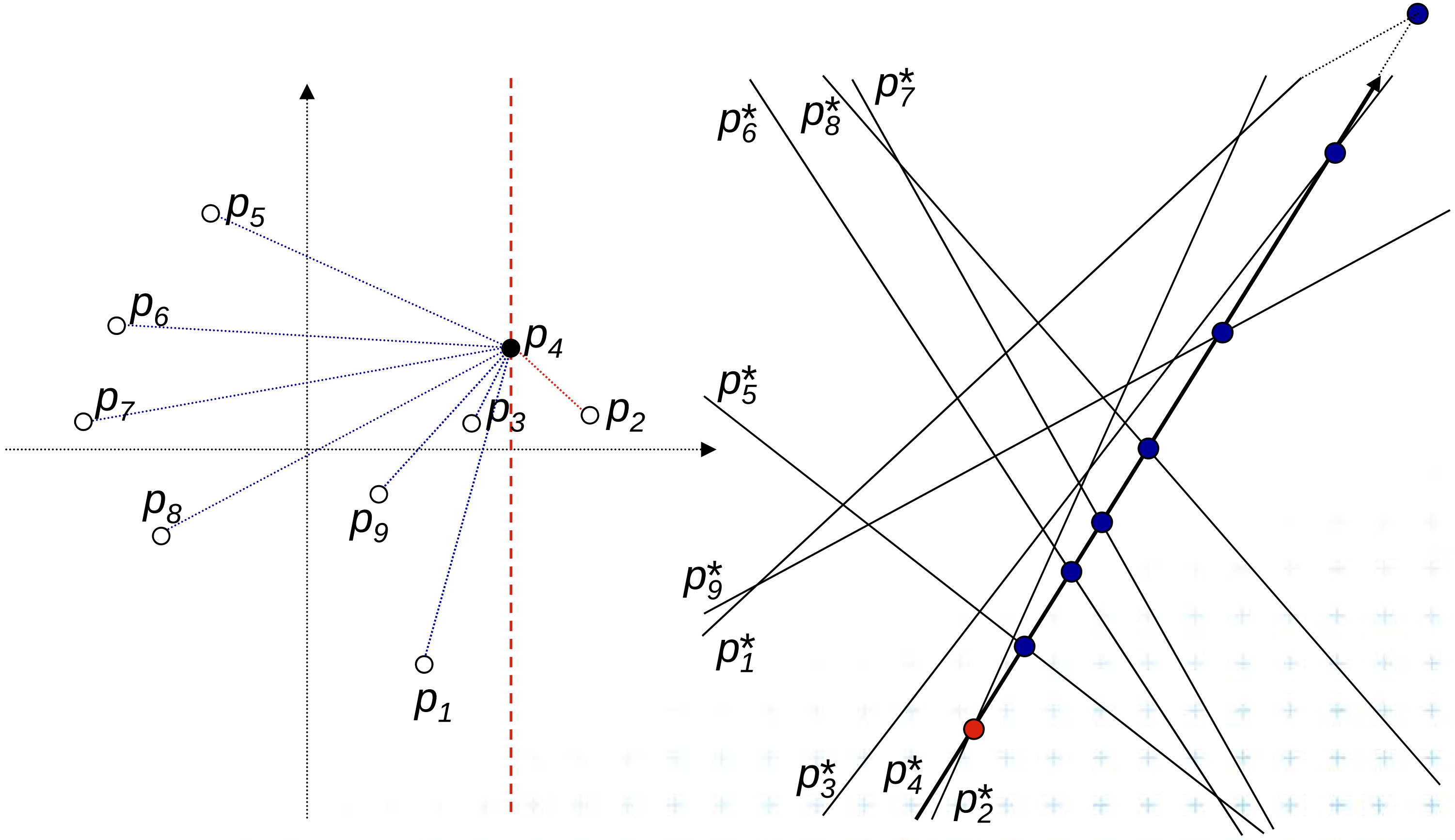




In primal plane

In dual plane

d) Angular sequences around p_4



In primal plane

In dual plane

Point order around p_4 : p_2, p_5, p_6, p_7, p_8, p_9, p_3, p_1

e) More applications of line arrangement

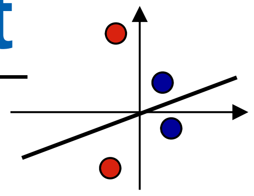
Visibility graph

Given a set of *n* non-intersecting line segments, compute the *visibility graph*, whose vertices are the endpoints of the segments, and whose edges are pairs of visible endpoints (use angular sequences).

Maximum stabbing line

Given a set of *n* line segments in the plane, compute the line that stabs (intersects) the maximum number of these line segments.

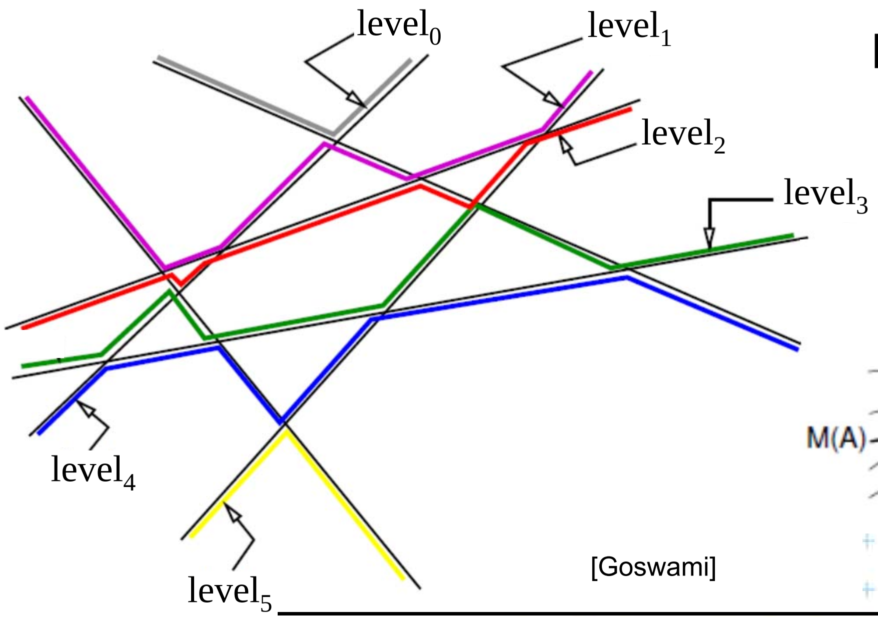
More applications of line arrangement



Ham-Sandwich cut

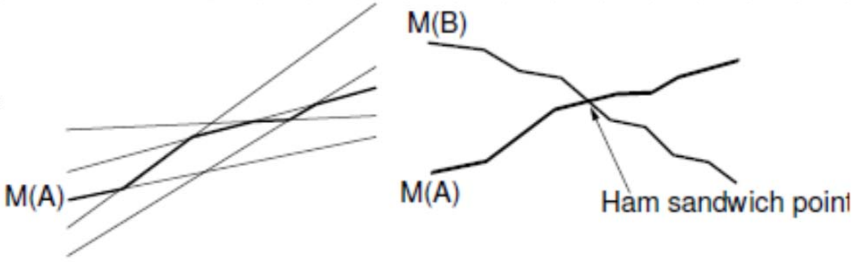
Given two sets of points, *n* red and *m* blue points compute a single line that simultaneously bisects both sets

Principle – intersect middle levels of arrangements



[Goswami]

Point at *k-th* level L_k has
at most k lines above and
at most n – k – 1 lines below



Dual arrangement of A. Overlay of A and B's median levels

[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 8., http://www.cs.uu.nl/geobook/

[Mount] David Mount, - CMSC 754: Computational Geometry, Lecture Notes for Spring 2007, University of Maryland, Lectures 8,15,16,31, and 32. http://www.cs.umd.edu/class/spring2007/cmsc754/lectures.shtml

[applet] Allen K. L. Miu: Duality Demo http://nms.lcs.mit.edu/~aklmiu/6.838/dual/

[Goswami] Partha P. Goswami: Duality Transformation and its Application to Computational Geometry, University of Calcutta, India http://www.tcs.tifr.res.in/~igga/lectureslides/partha-lec-iisc-jul09.pdf

DCGI

KATEDRA POČÍTAČOVÉ GRAFIKY A INTERAKCE

MODERN ALGORITHMS (not only in computational geometry)

PETR FELKEL

FEL CTU PRAGUE

Version from 2.1.2019

Modern algorithms

1. Computational geometry today
2. Space efficient algorithms
   (In-place / in situ algorithms)
3. Data stream algorithms
4. Randomized algorithms

Computational geometry today

- Popular: beauty as discipline, wide applicability
- Started in 2D with linear objects (points, lines,…), now 3D and nD, hyperplanes, curved objects,…
- Shift from purely mathematical approach and asymptotical optimality ignoring singular cases
- to practical algorithms, simpler data structures and robustness => algorithms and data structures provable efficient in realistic situations (application dependent)

Space efficient algorithms

Space efficient algorithms

- output is in the same location as the input and
- need only a small amount of additionally memory
  - *in-place* – O(1) extra storage
    sometimes including O(*log n)* bits for indice
  - *in situ* – O(*log n*) extra storage

Space efficient algorithms - practical advantages

- Allow for processing larger data sets
  - Algorithms with separate input and output need space for 2n points to store – O(*n*) extra space
  - Space efficient algs. – n points + O(1) or O(log *n*) space
- Greater locality of reference
  - Practical for modern HW with memory hierarchies (e.g., main RAM – ram on chip – registers, caches, disk latency, network latency)
- Less prone to failure
  - no allocation of large amounts of memory, which can fail at run time
  - good for mission critical applications
- Less memory => faster program

Ex: String reverse

```
function reverse(a[0..n])
    allocate b[0..n]
    for i from 0 to n
      b[n-i] = a[i]
    return b
```

×

```
function reverseInPlace(a[0..n])
    for i from 0 to floor(n/2)
      swap (a[n-i], a[i])
```

In-place sorting

- In array – continuous block in memory
  - n^{th} element in $O(1)$ time
  - Select sort, insert sort … in-place, $O(1)$ additional memory, $O(n^2)$ time
  - Heapsort – in-place, $O(1)$ add. memory, $O(n \log n)$ time
  - Quicksort – in-situ, $O(\log n)$ add. memory for recursion
  - Mergesort – not in-place, not in-situ, $O(n)$ add. memory
- In list – linked lists in dynamical memory
  - n^{th} element in $O(n)$ time
  - Mergesort –in-situ, $O(\log n)$ add. memory, $O(n \log n)$ time

Graham in-place algorithm

Graham-InPlaceScan(S, n, d)

Input: S – index to array of length n with points in plane, $d = \pm 1$ direction

Output: Convex Hull in clockwise order

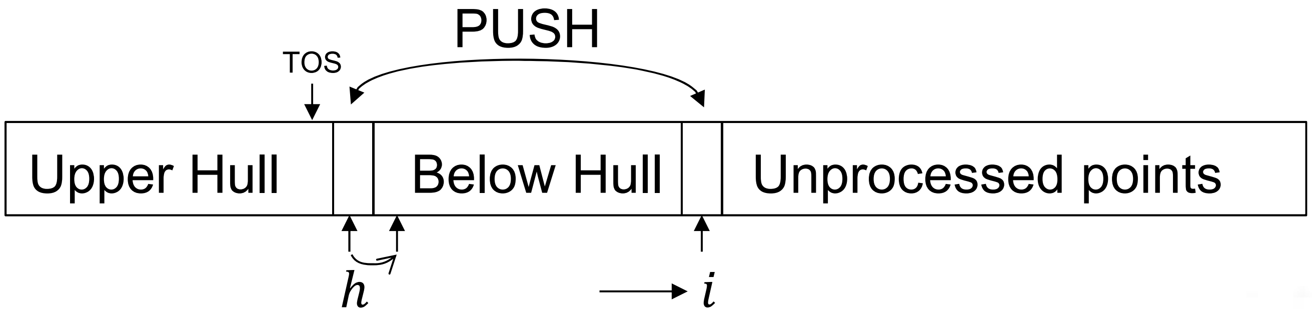
```
                                          // $d$ controls the sort direction:
InPlace-Sort($S, n, d$)                   //   $d =$   1 sort ascending    for upper hull
$h \leftarrow 1$    // empty stack        //   $d = -1$ sort descending  for lower hull
for $i \leftarrow 1 \ldots n - 1$ do
                                               TOS-1        TOS        NEW
    while $h \geq 2$ and not right turn( $S[h-2]$, $S[h-1]$, $S[i]$ ) do
        $h \leftarrow h - 1$             // pop top element from the stack
    swap $S[i] \leftrightarrow S[h]$     // push the new point to the stack
    $h \leftarrow h + 1$                 // increment stack length
return $h$                           // end of convex hull (the first point above the stack)
```

The array: S = offset of the sub-array (index of its first point)

h = index of the first point above the stack (offset to S)

i = index of the current point

Graham in-place algorithm



Graham in-place algorithm

Graham-InPlaceHull(S, n)

Input: S – an array of length n with points in plane

Output: Convex Hull in clockwise order (CW)

sort direction

O(n log n)

```
$h \leftarrow$ Graham-InPlaceScan($S, n$, 1)    // 1= ascending – CW upper hull
for $i \leftarrow 0\,..\,h-2$ do
        swap $S[i] \leftrightarrow S[i+1]$    // bubble $a$ to the right O(h)
$h' \leftarrow$ Graham-InPlaceScan($S+h-2, n-h+2, -1$)  // lower hull
return $h+h'-2$
```

Principle:

Stack at the beginning of the array S on indices $[0\,..\,h-1]$

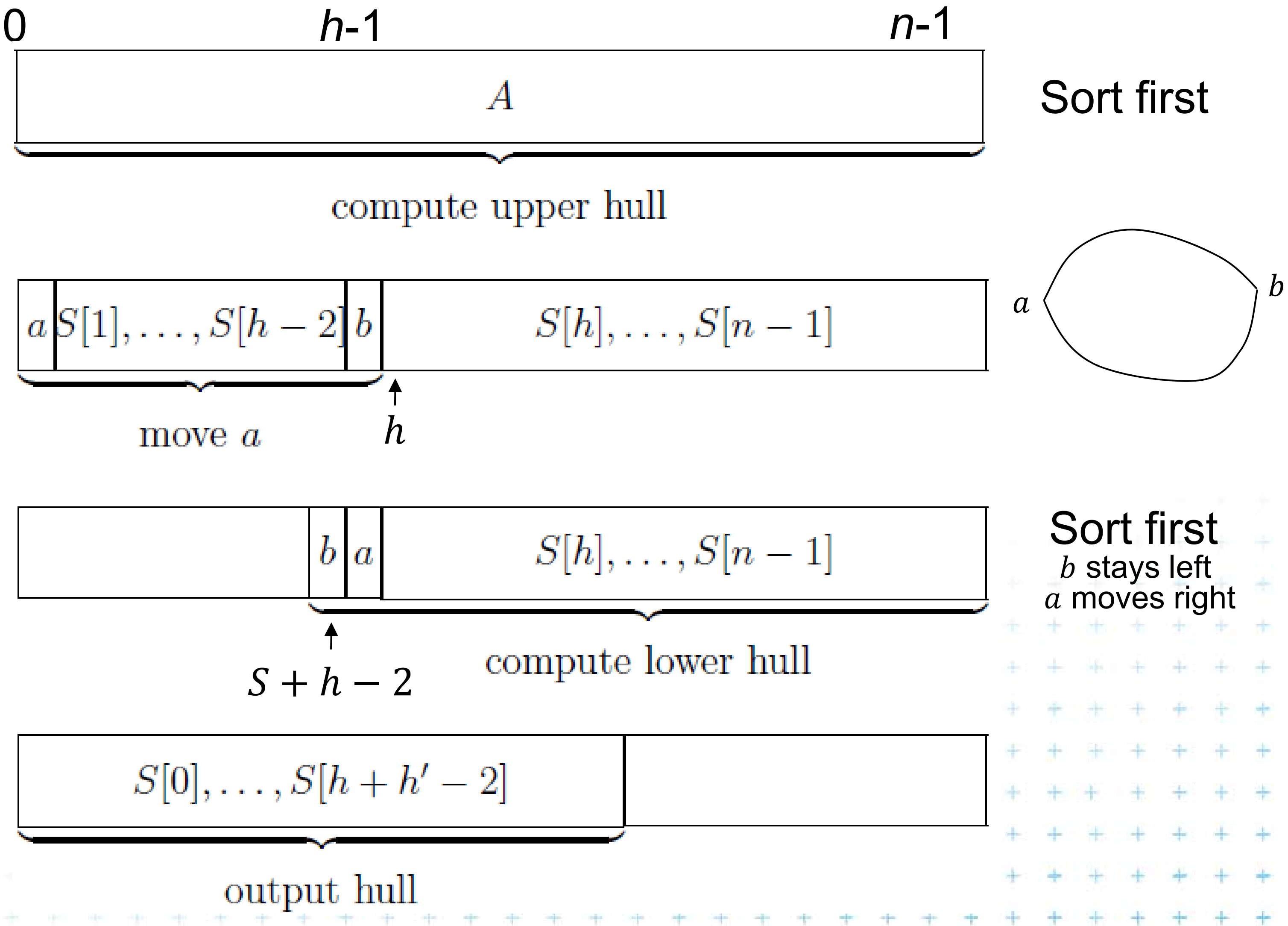
Exchange by swap operation

We need the in-place sort

Graham in-place algorithm

a left CH point

b right CH point



0 h-1 n-1

A

Sort first

compute upper hull

a | $S[1],\ldots,S[h-2]$ | b | $S[h],\ldots,S[n-1]$

$\underbrace{a\ S[1],\ldots,S[h-2]\ b}_{\text{move } a}$

h

a b

b | a | $S[h],\ldots,S[n-1]$

$S+h-2$

compute lower hull

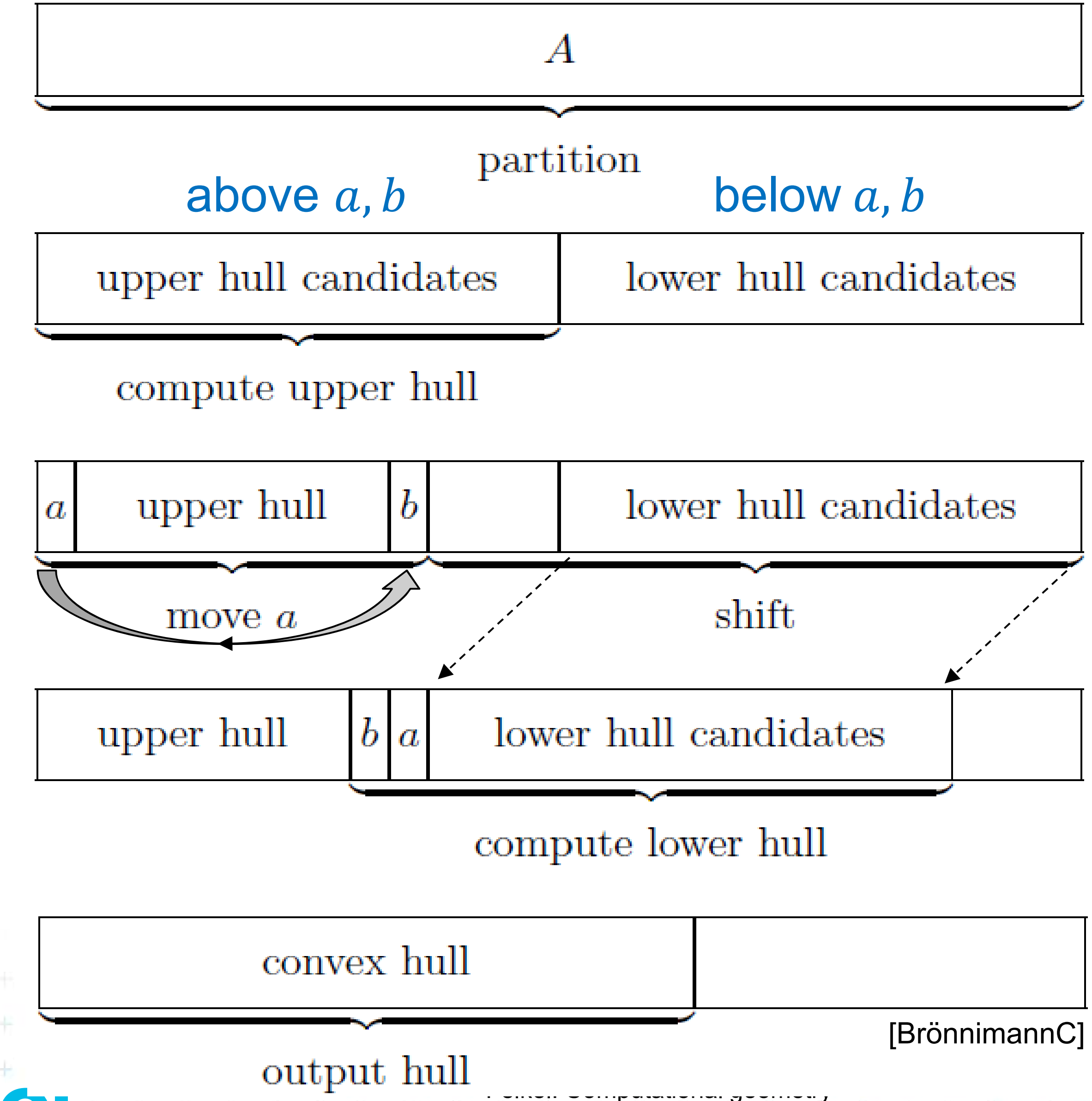
Sort first

b stays left

a moves right

$\underbrace{S[0],\ldots,S[h+h'-2]}_{\text{output hull}}$

Optimized Graham in-place algorithm



A

partition

above a, b

below a, b

upper hull candidates | lower hull candidates

compute upper hull

a | upper hull | b | | lower hull candidates

move a

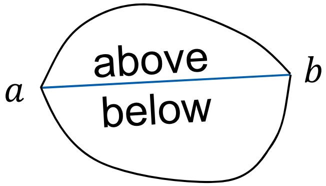
shift

upper hull | b | a | lower hull candidates |

compute lower hull

convex hull |

output hull



Sort first

b stays left

a moves right

[BrönnimannC]

Data stream algorithms

Data stream algorithms [Indyk]

- Data stream = a massive sequence of data
  - Too large to store (on disk, memory, cache,…)
- Examples
  - Network traffic
  - Database transactions
  - Sensor networks
  - Satellite data feeds
  - …
- Approaches
  - Ignore it (CERN ignores 9/10 of the data)
  - Develop algorithms for dealing with such data

Motivation example [Muthukrishnan]

- Paul presents numbers $x = \{1 \ldots n\}$ in random order, one number missing
- Carole must determine the missing number but has only $O(\log n)$ bits of memory

Any idea?

- Compute the sum of the numbers and subtracts the incoming numbers one by one.

$$missing\ number = \frac{n(n+1)}{2} - \sum_{i<n} x[i]$$

- The missing number “remains”

Motivation example [Muthukrishnan]

- And two missing numbers i, j ?

- Store sum of numbers s and sum of squares s'

$$i + j = \frac{n(n+1)}{2} - s$$

$$i^2 + j^2 = \frac{n(n+1)(2n+1)}{6} - s'$$

(this principle is applicable for k-missing numbers)

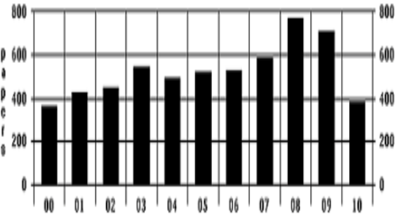
Basic data stream model [Indyk]

- Single pass over the data: $a_1, a_2, \dots ,\ a_n$
  - Typically n is known
- Bounded storage (typically n^α or $\log^c n$ or only c)
  - Units of storage: bits, words, or elements (such as points, nodes/edges, …)
  - Impossible to store the complete data
- Fast processing time per element
  - Randomness is OK (in fact, almost necessary)
  - Often sub-linear time for the whole data
  - Often approximation of the result

Data stream models classification

- Input stream $a_1, a_2, \ldots, a_n$
  - arrives sequentially, item by item
  - describes an underlying signal A, a 1D function A: $[1..N] \;-> \; R$
- Models differ on how the input a_i's describe the signal A for increasing i (in increasing order of generality):
  a) Time series model - a_i equals to signal $A[i]$
  b) Cash register model- a_i are increments to $A[j]$, $I_i \; > \; 0$
  c) Turnstile model - a_i are updates to $A[j]$, $U_i \in R$

a) Time series model (*časová řada*)



- Stream elements a_i are equal to $A[i]$ (a_i's are samples of the signal)
- a_i's appear in increasing order of i $(i \sim \text{time})$

- Applications
  - Observation of the traffic on IP address each 5 minutes
  - NASDAQ volume of trades per minute

b) Cash register model (*pokladna*)

+ only

- a_i are increments to signal $A[j]'s$
- Stream elements $a_i = (j, I_i)$, $I_i \geq 0$ to mean

I_i= Increment

$$A_i[j] = A_{i-1}[j] + I_i$$

where $(i{\sim}\text{time}, \text{j}{\sim}\text{bucket})$

  - $A_i[j]$ is the state of the signal after seeing i-th item
  - multiple a_i can increment given $A[j]$ over time
- A most popular data stream model
  - IP addresses accessing web server (histogram)
  - Source IP addresses sending packets over a link
  - access many times, send many packets,…

c) Turnstile model (*turniket*)

- a_i are updates to signal $A[j]'s$
- Stream elements $a_i = (j, U_i)$, $U_i \in R$ to mean

+ –

U_i= Update

$$A_i[j] = A_{i-1}[j] + U_i$$

where $(i\sim\text{time, j}\sim\text{bucket, turnstile})$

  - A_i is the state of the signal after seeing i-th item
  - U_i may be positive or negative
  - multiple a_i can update given $A[j]$ over time
- A most general data stream model
  - Passengers in NY subway arriving and departing
  - Useful for completely dynamic tasks
  - Hard to get reasonable solution in this model

c) Turnstile model variants (for completeness)

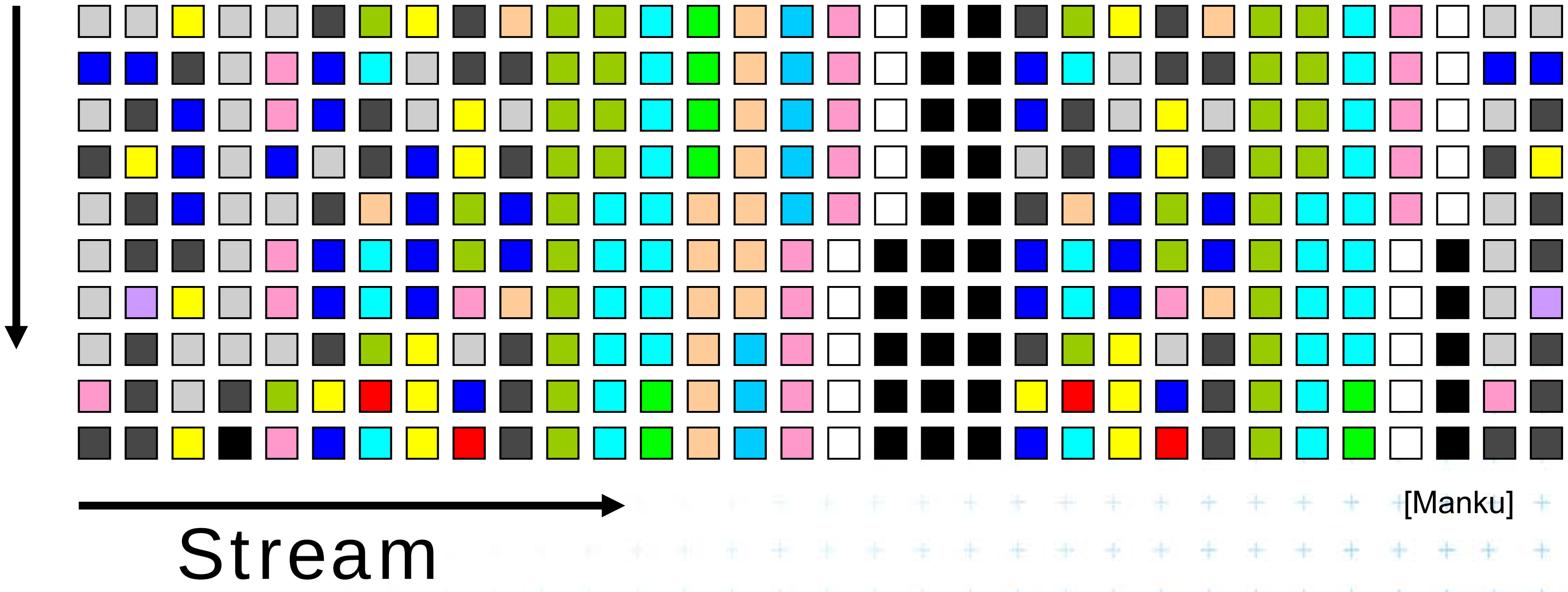
- strict turnstile model – $A_i[j] \geq 0$ for all i
  - People can only exit via the turnstile they entered in
  - Databases – delete only a record you inserted
  - Storage – you can take items only if they are there
- non-strict turnstile model – $A_i[j] < 0$ for some i
  - Difference between two cash register streams
  - ($A_i[j] < 0$ … negative amount of items for some i)

Examples: Iceberg queries

[Manku]

- Identify all elements whose current frequency f exceeds support threshold s = 0.1%

$$f \geq sN$$



[Manku]

Ex: Iceberg queries – a) ordinary solution

The ordinary solution in two passes (not data stream)

1. Pass – identify frequencies (count the hashes)
   - a set of counters is maintained. Each incoming item is hashed onto a counter, which is incremented.
   - These counters are then compressed into a bitmap, with a 1 denoting a large counter value.
2. Pass – count exact values for large counters only
   - exact frequencies counters for only those elements which hash to a value whose corresponding bitmap value is 1

- Hard to modify for data stream – unknown frequencies after only 1st pass

Ex: Iceberg queries – data stream definition

- Input: threshold $s \in (0,1)$, error $\varepsilon \in (0,1)$, length N
- Output: list of items and frequencies $\epsilon \ll s$
- Guarantees:
  - No item omitted (reported all items with frequency > sN)
  - No item added (no item with frequency < $(s - \epsilon)N$)
  - Estimated frequencies are not less than ϵN of the true frequencies
- Ex: $s = 0.1\%, \epsilon = 0.01\% \ \rightarrow \epsilon \text{ about } \frac{1}{10} \text{ to } \frac{1}{20} \text{ of } s$
  - All elements with freq. > 0.1% will output
  - None of element with freq. < 0.09% will output
  - Some elements between 0.09% and 0.1% will output

Ex: Iceberg queries – b) sticky sampling

- Probabilistic algorithm, given threshold s, error ϵ and probability of failure δ
  - Data structure S of entries (e, f), // S =subset of counters
    e element, f estimated frequency,
    r sampling rate, sampling probability $\frac{1}{r}$
- $S \leftarrow \emptyset, r \leftarrow 1$
- If $e \in S$ then $(e, f{+}{+})$ //count, if the counter exists
  else insert (e, f) into S with probability $\frac{1}{r}$
- S sweeps along the stream as a magnet, attracting all elements which already have an entry in S

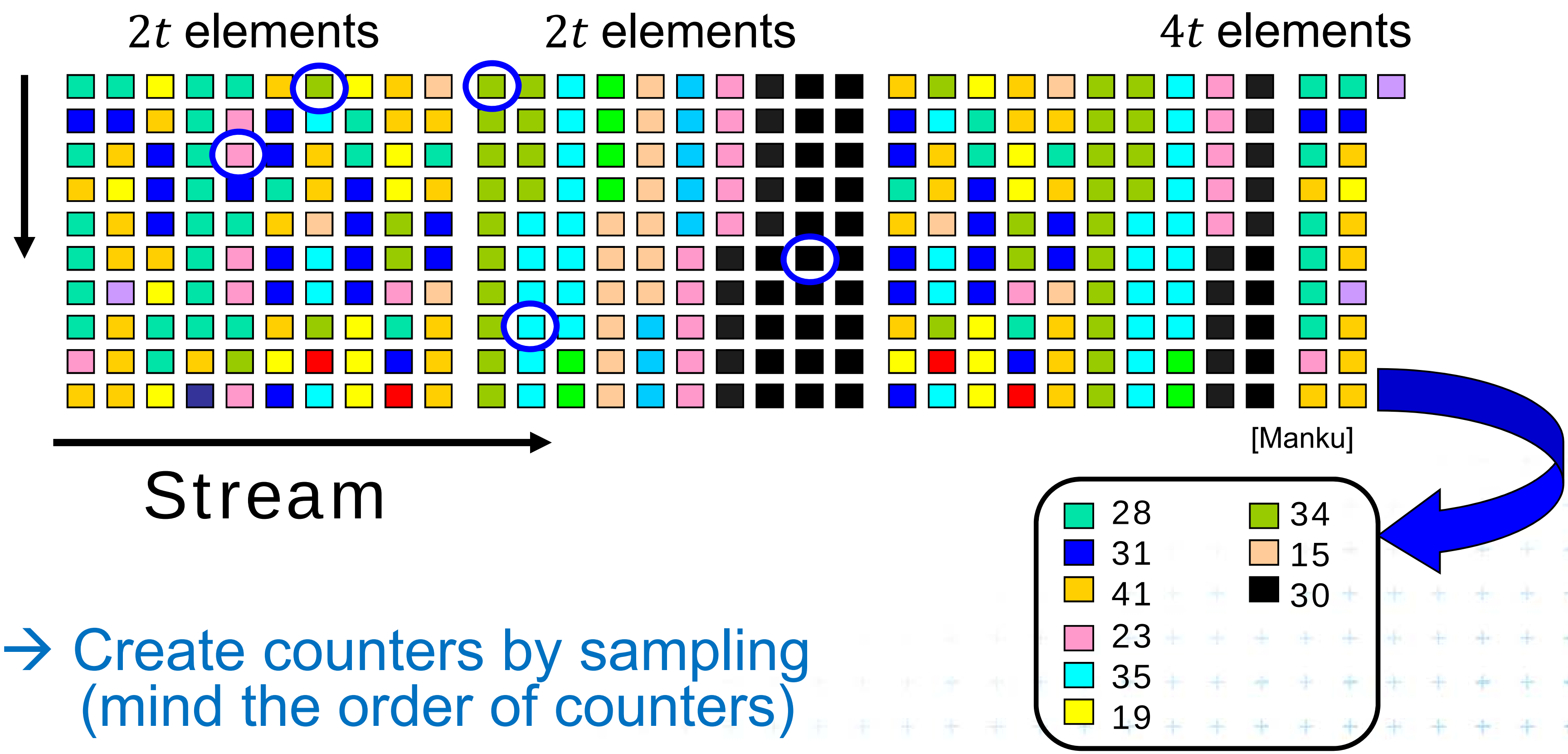
Ex: Iceberg queries – b) sticky sampling

- r changes over the stream, $t = \frac{1}{\epsilon}\log\left(\frac{1}{s\delta}\right), |S| < 2t$
  - $2t$ elements $r = 1$
  - next $2t$ elements $r = 2$
  - next $4t$ elements $r = \ 4$ …
- whenever r changes, we update S
  - For each entry (e, f) in S // random decrement of counters
    - toss a coin until successful (head) // with probability $1/2$
    - if not successful (tail), decrement f
    - if f becomes 0, remove entry (e, f) from S
- Output: list of items with threshold s i.e. all entries in S where $f \geq (s - \epsilon)N$

Ex: Iceberg queries – b) sticky sampling

- Space complexity is independent on N
- For
  - support threshold $s = 0.1\%$,
  - error $\epsilon = 0.01\%$,
  - and probability of failure $\delta = 1\%$
- Sticky sampling computes results
  - with $(1 - \delta) = 99\%$ probability
  - using at most 2t = 80 000 entries
  - $t = \frac{1}{\epsilon}\log\left(\frac{1}{s\delta}\right) = 40\,000, |S| < 2t$

Ex: Iceberg queries – b) sticky sampling

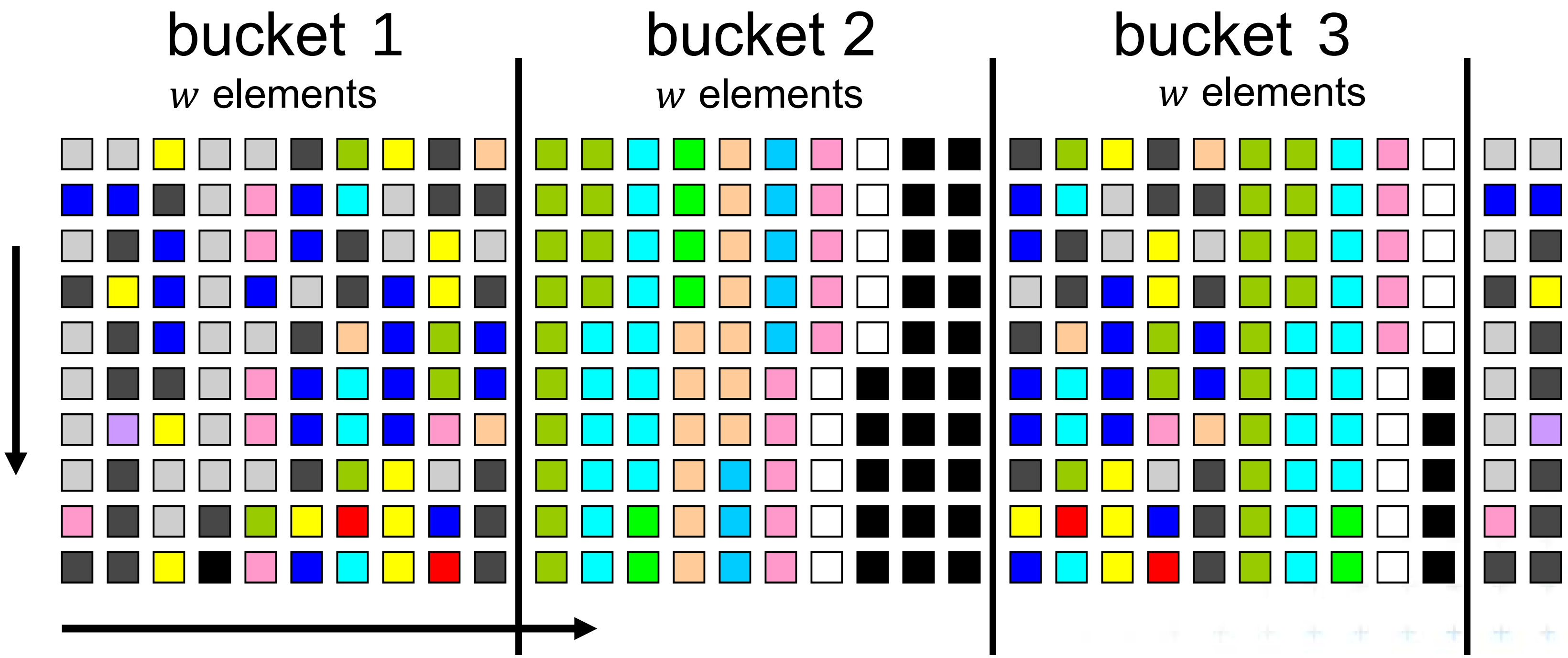


→ Create counters by sampling (mind the order of counters)

Ex: Iceberg queries – c) lossy counting

- Deterministic algorithm (user specifies error ε and threshold s)
- Stream conceptually divided into buckets
  - With bucket size $w = \lceil 1/\varepsilon \rceil$ items each
  - Numbered from 1, current bucket id is $b_{current}$
- Data structure D of entries (e, f, Δ),
  - e element,
  - f estimated frequency,
  - Δ maximum possible error of f, $\Delta = b_{current} - 1$ (max number of occurrences in the previous buckets)
- At most $\frac{1}{\epsilon}\log(\varepsilon N)$ entries

Ex: Iceberg queries – c) lossy counting



[Manku]

- Divide the stream into buckets
- Keep exact counters for items in the buckets
- Prune entries at bucket boundaries (remove entries for which $f + \Delta \leq b_{current}$)

Ex: Iceberg queries – c) lossy counting alg.

- $D \leftarrow \emptyset$
- New element e
  - If $e \in D$ then increment its f
  - If $e \notin D$ then
    - Create a new entry $(e, 1, b_{current} - 1)$
    - If on the bucket border, i.e., $N \; mod \; w = 0$ then delete entries with $f + \Delta \leq b_{current}$
    - i.e., with zero or one occurrence in each of the previous buckets
  - New $\Delta = b_{current} - 1$ is maximum number of times e could have occurred in the first $b_{current} - 1$ buckets
- Output: list of items with threshold s i.e. all entries in S where $f \geq (s - \epsilon)N$

Comparison of sticky and lossy sampling

- Sticky sampling performs worse
  - Tendency to remember every unique element
  - The worst case is for sequence without duplicates
- Lossy counting
  - Is good in pruning low frequency elements quickly
  - Worst case for pathological sequence which never occurs in reality

Number of mutually different entries 1/2

- Input: stream $a_1, a_2, \ldots, a_n$, with repeated entries
- Output: Estimate of number c of different entries
- Appl: # of different transactions in one day

a) Precise deterministic algorithm:

- Array $b[1..U]$, $U =$ max number of different entries
- Init by $b[i] \ = \ 0$ for all i, counter $c = 0$
- for each a_i

  if $b[a_i] \ = \ 0$ then $\mathrm{inc}(c)$, $b[i] = 1$
- Return c as number of different entries in $b[]$
- $O(1)$ update and query times, $O(U)$ memory

Number of mutually different entries 2/2

b) Approximate algorithm

- Array $b[1 \ldots \log U]$, $U =$ max number of different entries
- Init by $b[i] = 0$ for all i
- Hash function h: $\{1..U\} \rightarrow \{0..\log U\}$
- For each a_i

  Set $b[h(a_i)]$ = 1
- Extract probable number of different entries from $b[]$

Sublinear time example $O(\mathrm{alg}) < O(n)$

- Given mutually different numbers $a_1, a_2, \dots, a_n$
- Determine any number from upper half of values
- Alg: select k numbers equally randomly
  - Compute their maximum
  - Return this estimation as solution
- Probability of wrong answer = probability of all selected numbers are from the lower half = $\left(\frac{1}{2}\right)^k$
- For error ϵ take $\log\frac{1}{\epsilon}$ samples
- Not useful for MIN, MAX selection

Randomized algorithms

Randomized algorithms

Motivation

- Array of elements, half of char ”a”, half of char ”b”
- Find ”a”
- Deterministic alg: $n/2$ steps of sequential search (when all ”b” are first)
- Randomized:
  - Try random indices
  - Probability of finding ”a” soon is high regardless of the order of characters in the array
    (Las Vegas algorithm – keep trying up to $n/2$ steps)

Randomized algorithms

- May be simpler even if the same worst time
- Deterministic algorithm
  - is not known (prime numbers)
  - does not exist
- Randomization
  - can improve the average running time (with the same worst case time), while
  - the worst time depends on our luck – not on the data distribution
    (It is “hard” to prepare killing datasets)

Randomized algorithms

a) Incremental algorithms (insert something in random order)
   - Linear programming (random plane insertion)
   - Convex hulls
   - Intersections, space subdivisions

b) Divide and conquer (split in random place)
   - Random sampling
   - Nearest neighbors, trapezoidal subdivisions

Another classification

- Monte Carlo
  - We always get an answer, often not correct
  - Fast solution with risk of an error
  - It is not possible to determine, if the answer is correct
    → run multiple times and compare the results
  - Output can be understand as a random variable
  - Example: prime number test
    - Task: Find $\mathrm{a} \in \left\langle 2, \frac{n}{2} \right\rangle$ such as *n* is divisible by a
    - Algorithm: Sample 10 numbers from the given interval, answer
- Las Vegas

Las Vegas algorithms

Las Vegas

- We always get a correct answer
- The run time is random (typically ≤ deterministic time)
- Sometimes fails –> perform restart
- Example: Randomized quicksort
  - No median necessary
  - Simpler algorithm
  - Independent on data distribution
  - Return a correct result
  - The result will be ready in $\theta(n \log n)$ time with a high probability
  - Bad luck – we select the smallest element -> Selection sort

Randomized quicksort (Las Vegas alg.)

RQS(S) = Randomized Quicksort
Input: sequence of data elements $a_1, a_2, \dots, a_n \in S$
Output: sorted set S

1. Step 1: choose $i \in \langle 1, n \rangle$ in random
2. Step 2: Let A is a multiset $\{a_1, a_2, \dots, a_n\}$
   - if $n = 1$ then output(S)
   - else – create three subsets of $S_<, \; S_=, \; S_>$

$$S_< = \{b \in A: b < a_i\}$$
$$S_= = \{b \in A: b = a_i\}$$
$$S_> = \{b \in A: b > a_i\}$$

3. Step 3: *RQS($S_<$)* and *RQS($S_>$)*

4. Return: *RQS($S_<$)*, $S_=$, *RQS($S_>$)*

Conclusion on randomized algs.

- Randomized algorithms are often experimental
- We would not get perfect results, but nicely good
- We use randomized algorithm if we do not know how to proceed

References

[Kolingerová] Nové směry v algoritmizaci a výpočetní geometrii (1 a 2), přednáška z předmětu Aplikovaná výpočetní geometrie, MFF UK, 2008

[Brönnimann] Hervé Brönnimann. Towards Space-Efficient Geometric Algorithms, Polytechnic university, Brooklyn, NY,USA, ICCSA04, Italy, 2004

[BrönnimannC]Hervé Brönnimann, et al. 2002. In-Place Planar Convex Hull Algorithms. In *Proceedings of the 5th Latin American Symposium on Theoretical Informatics* (LATIN '02), Sergio Rajsbaum (Ed.). Springer-Verlag, London, UK, UK, 494-507. http://dl.acm.org/citation.cfm?id=690520

[Indyk] Piotr Indyk. 6.895: Sketching, Streaming and Sub-linear Space Algorithms, MIT course

[Muthukrishnan] Data streams: Algorithms and applications, (“adorisms” in Google)

[Mulmuley] Ketan Mulmuley. Computational Geometry. An Introduction Through Randomized Algorithms. Prentice Hall, NJ,1994

[Manku] G.S. Manku, R. Motwani. Approximate Frequency Counts over Data Streams, Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002. http://www.vldb.org/conf/2002/S10P03.pdf