Vertical Decomposition for Point Location

Algorithmic Problem: Point location

Algorithmic Technique: Randomized Incremental Construction

Data Structure: Vertical Decomposition

Where is Peru?

Where is Peru?
correct!

Where is Thailand?

Where is Thailand?
correct!

Where is Gabon?

Where is Gabon?
no, this is Ghana

Where is Gabon?

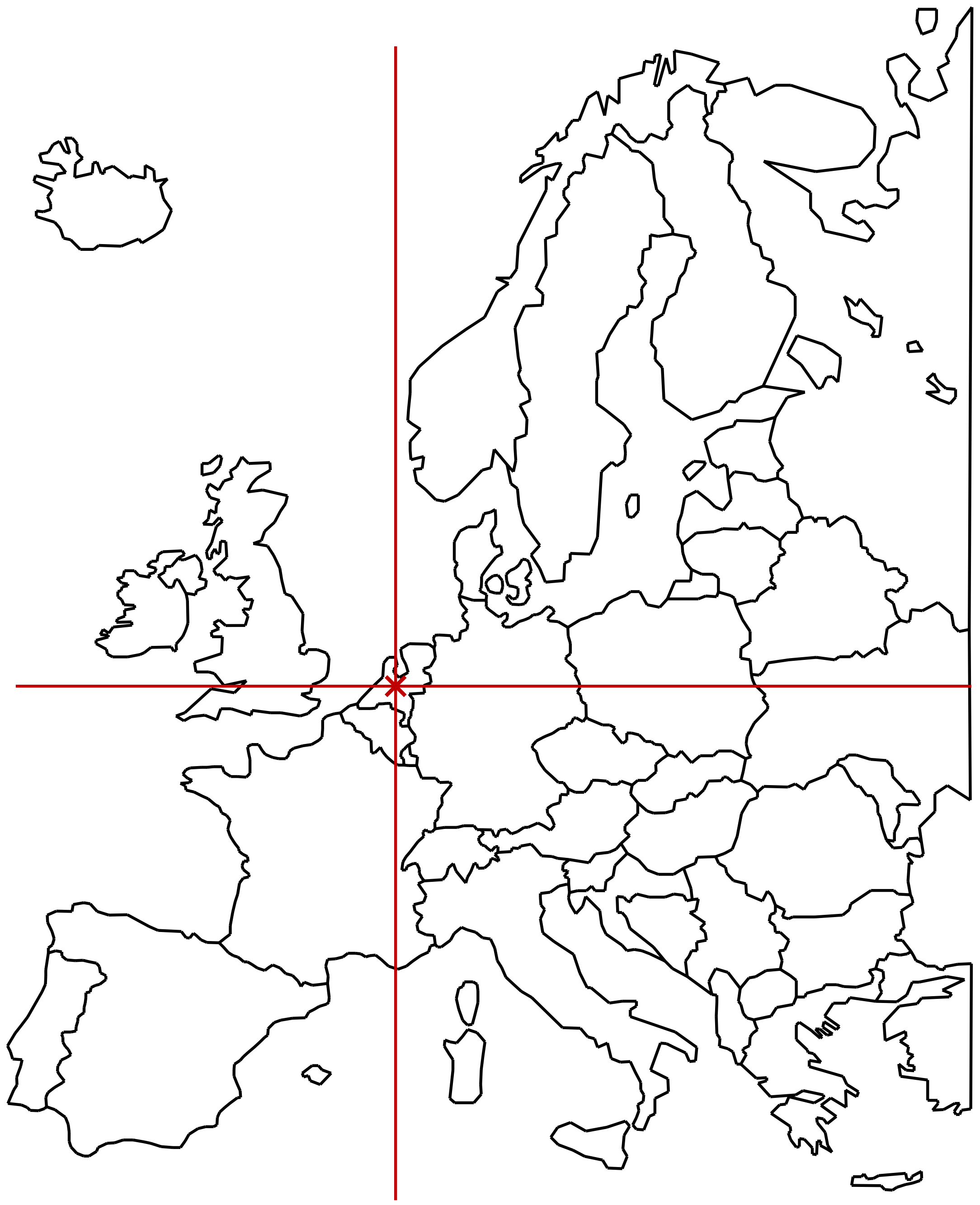
no, this is Uganda

Where is Gabon?
correct!

How can we efficiently compute which country was clicked on?

Motivation

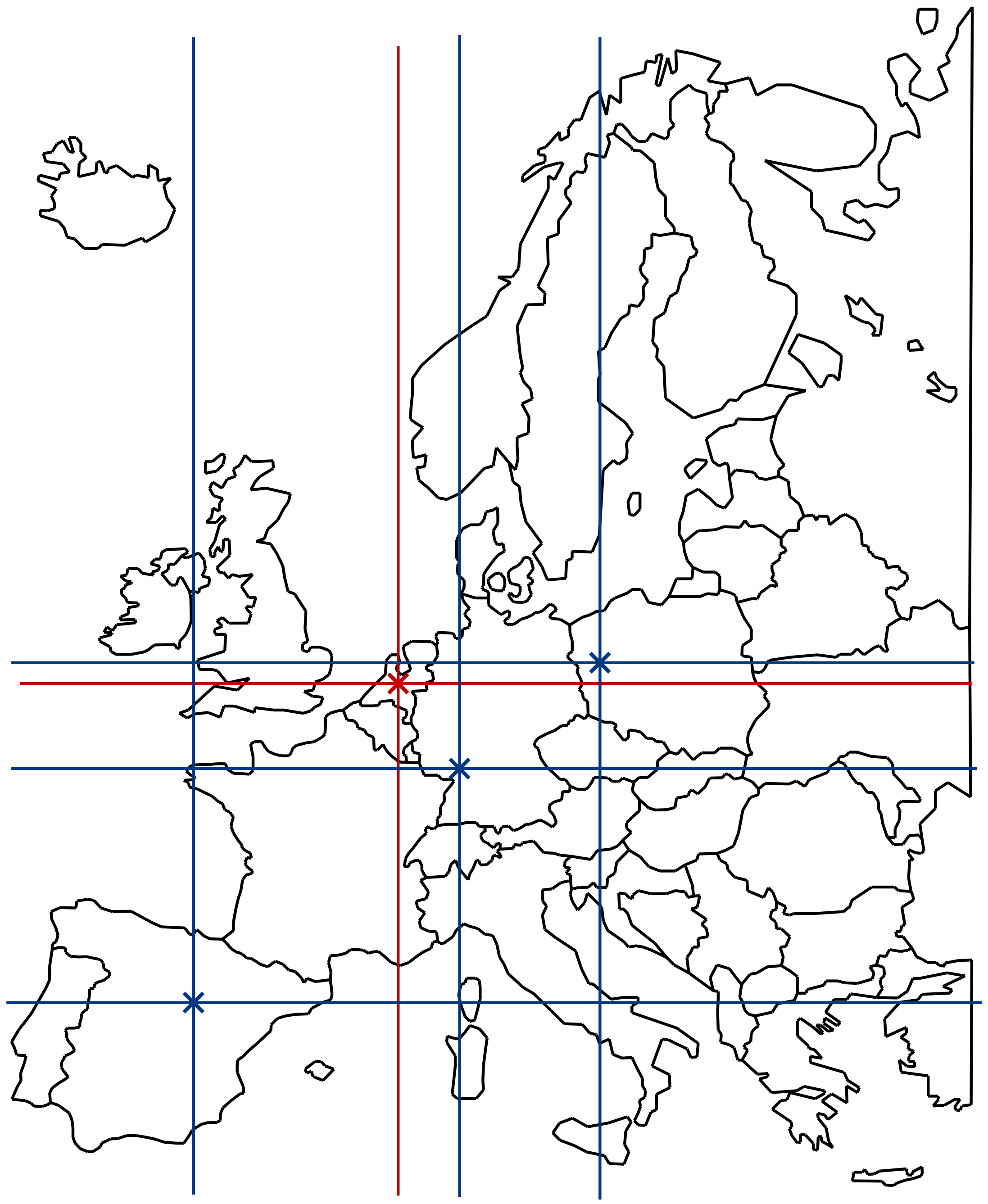
Motivation



Motivation

Given a position $p = (p_x, p_y)$ on a map determine which country p lies in.

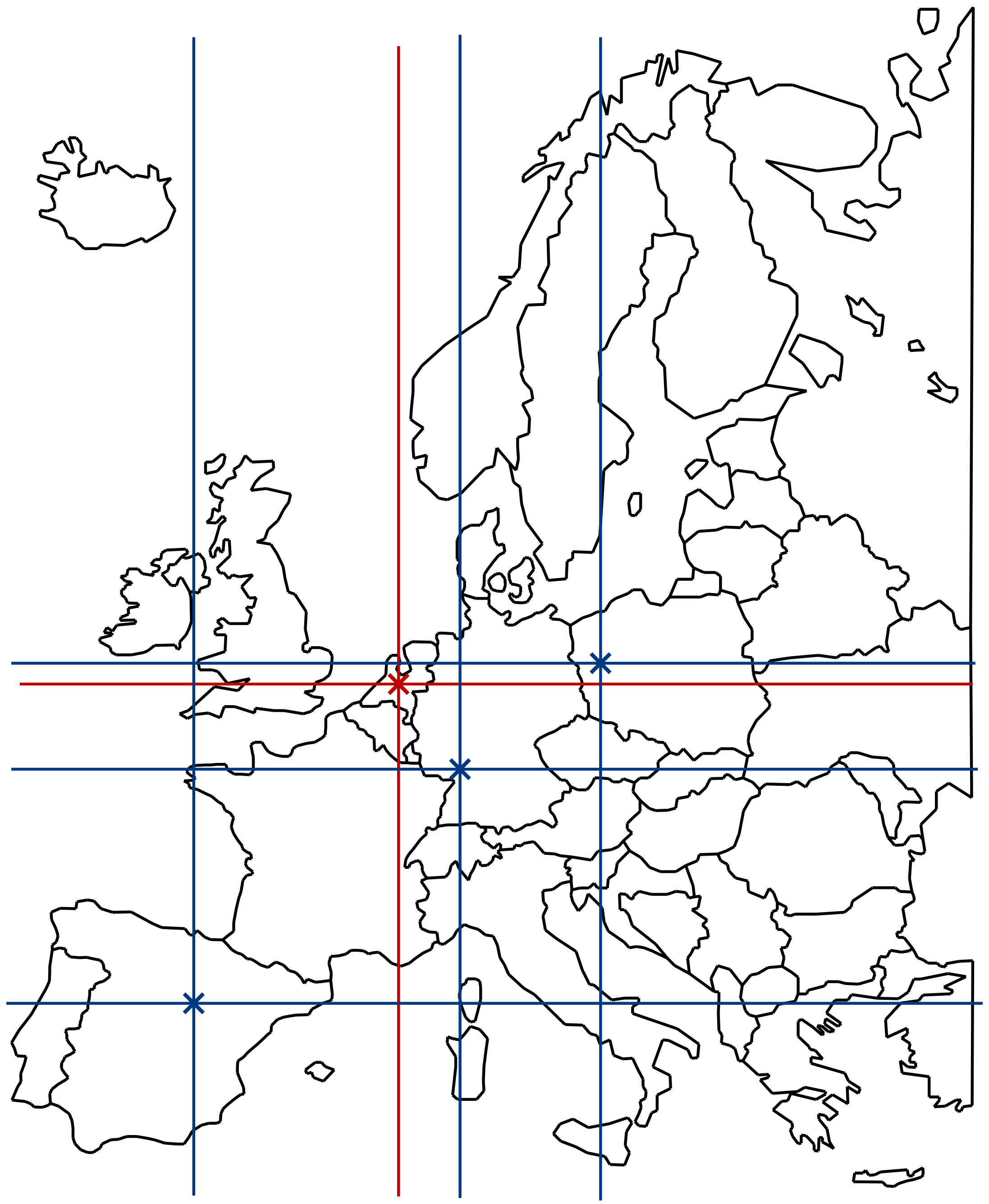
Motivation



Given a position $p = (p_x, p_y)$ on a map determine which country p lies in.

$\rightarrow$ Point location problem

Motivation



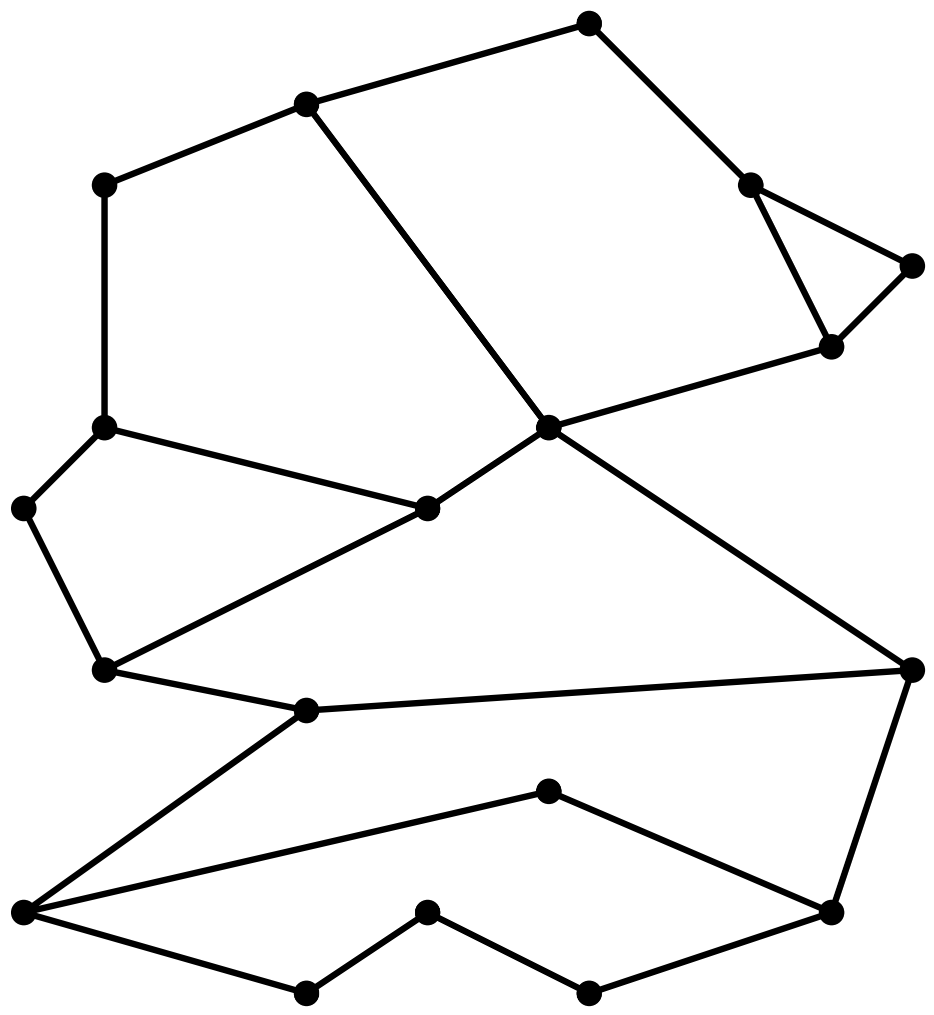
Given a position $p = (p_x, p_y)$ on a map determine which country p lies in.

$\rightarrow$ Point location problem

additional motivation:

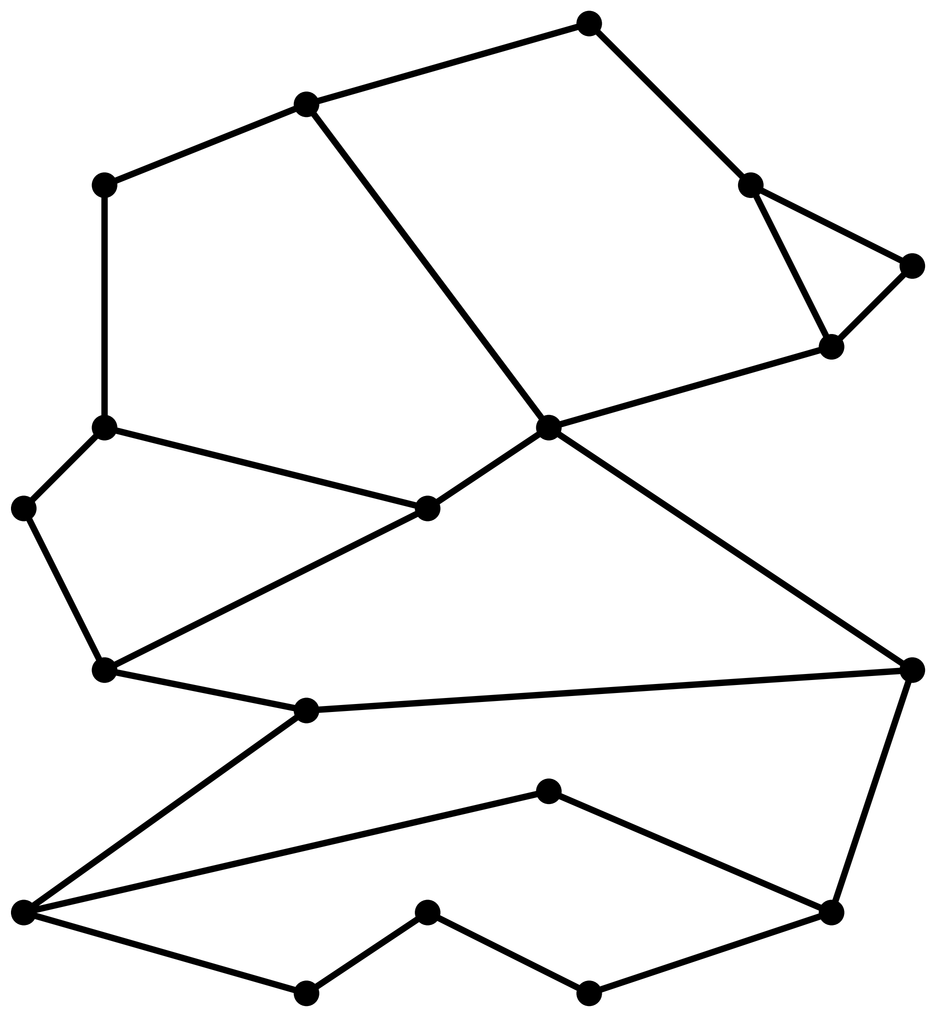
- subroutine for other geometric problems, e.g., motion planning (Ch. 13)
- randomized incremental construction

Point location



Point location problem: Preprocess a planar subdivision such that for any query point q, the face of the subdivision containing q can be given efficiently

Point location

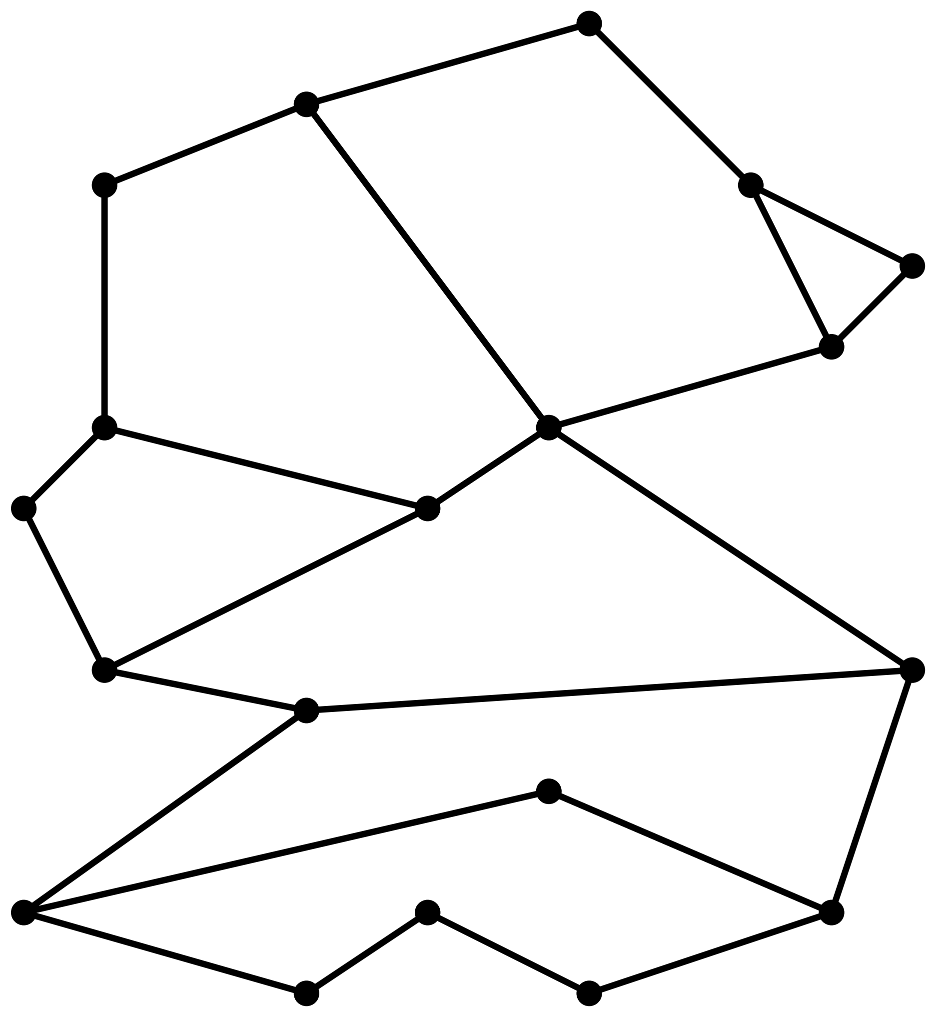


Point location problem: Preprocess a planar subdivision such that for any query point q, the face of the subdivision containing q can be given efficiently

Planar subdivision: Partition of the plane by a set of non-crossing line segments into vertices, edges, and faces

non-crossing: disjoint, or at most a shared endpoint

Point location

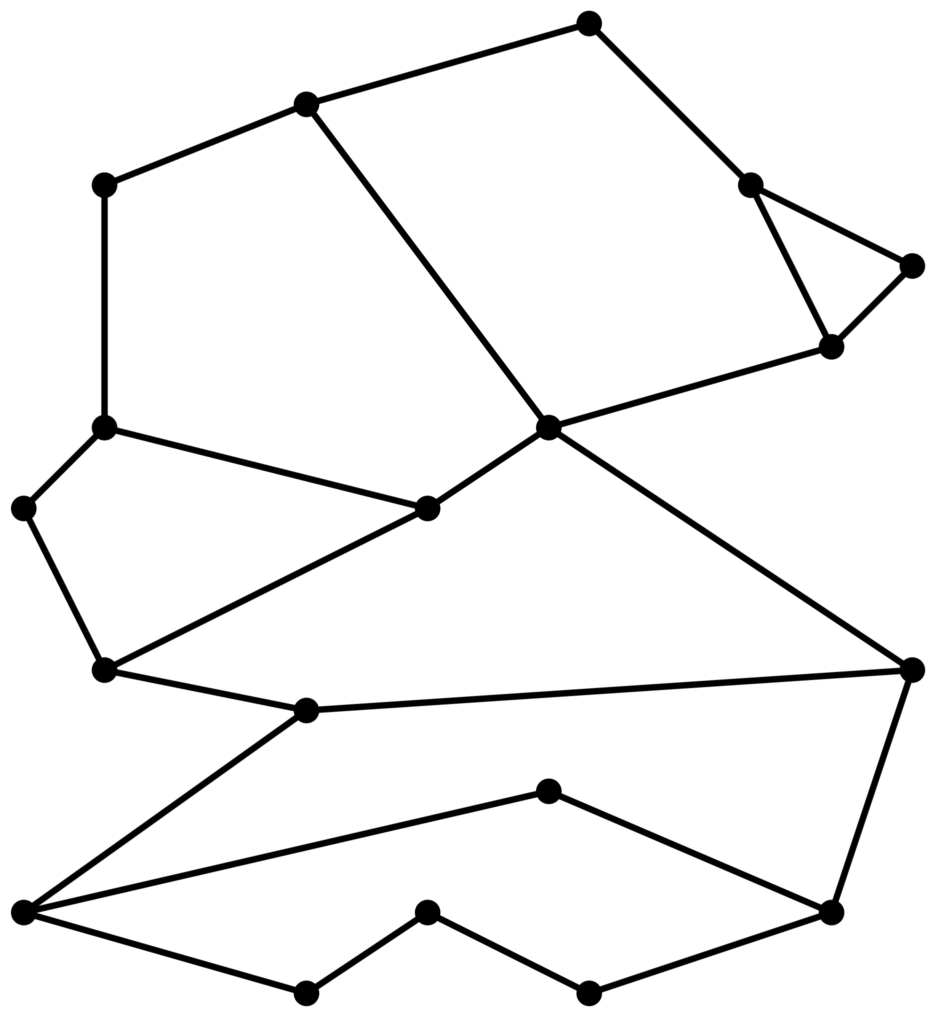


Point location problem: Preprocess a planar subdivision such that for any query point q, the face of the subdivision containing q can be given efficiently

Data structuring question, so interest in

- query time,
- storage requirements, and
- preprocessing time

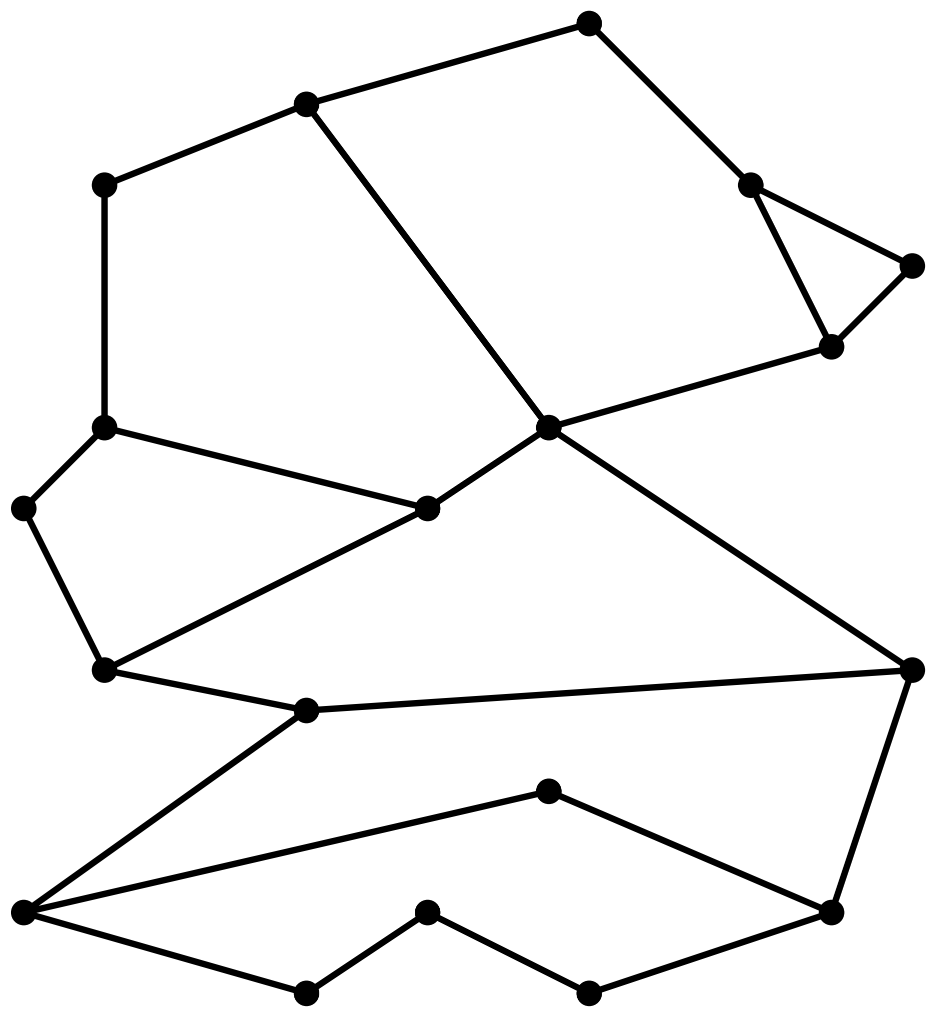
Point location



Point location problem: Preprocess a planar subdivision such that for any query point q, the face of the subdivision containing q can be given efficiently

Ideas?

Point location



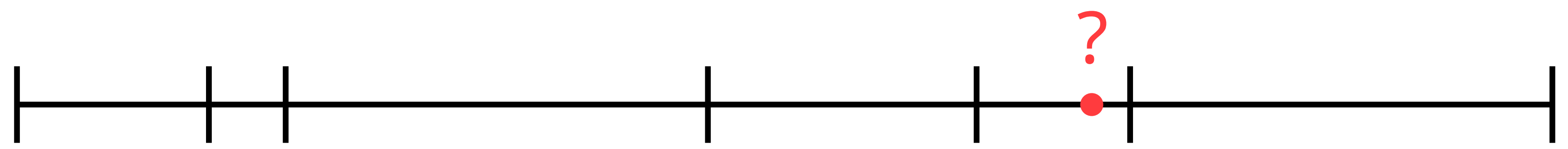
Point location problem: Preprocess a planar subdivision such that for any query point q, the face of the subdivision containing q can be given efficiently

Ideas?

- What is the corresponding 1D problem?
- How to solve 1D problem?

Quiz

How efficiently can we solve the 'point location' problem in 1D?



A: in $O(\log n)$ time using a data structure of size $\Theta(n)$

B: in $O(\log n)$ time using a data structure of size $\Theta(n^2)$

C: in the worst-case point location will take $\Theta(n)$ time, independent of the data structure

Quiz

How efficiently can we solve the ’point location’ problem in 1D?

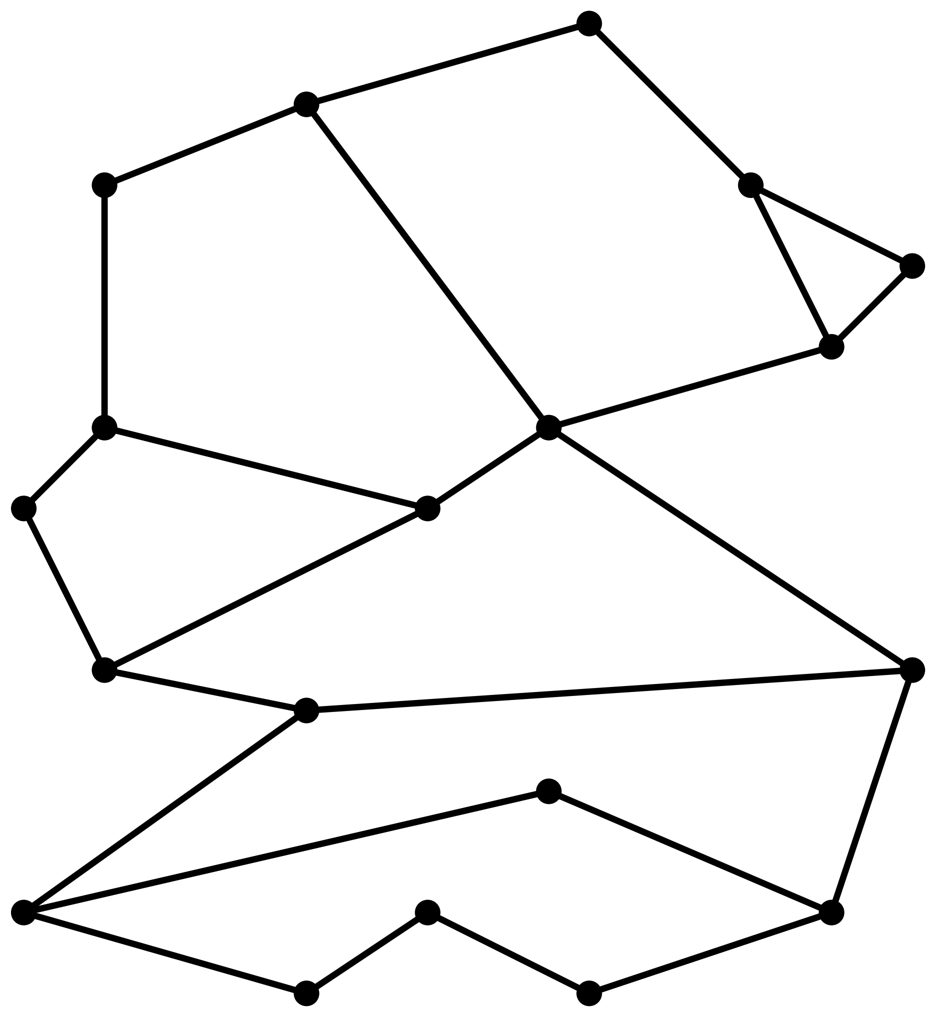
A: in $O(\log n)$ time using a data structure of size $\Theta(n)$

using a balanced binary search tree

B: in $O(\log n)$ time using a data structure of size $\Theta(n^2)$

C: in the worst-case point location will take $\Theta(n)$ time, independent of the data structure

Point location

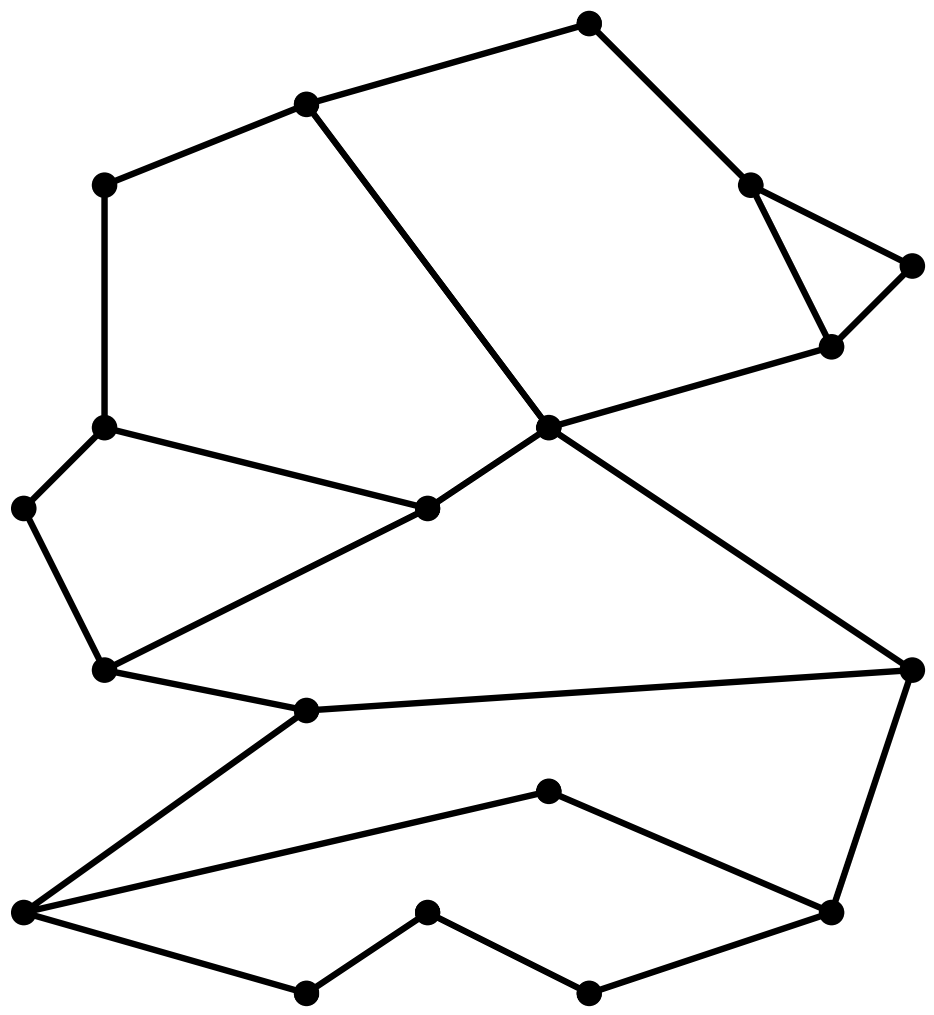


Point location problem: Preprocess a planar subdivision such that for any query point q, the face of the subdivision containing q can be given efficiently

Ideas?

- What is the corresponding 1D problem?
- How to solve 1D problem?
- Can we reduce the problem to the 1D, e.g., first x, then y?
- query time, . . .?

Point location



Point location problem: Preprocess a planar subdivision such that for any query point q, the face of the subdivision containing q can be given efficiently

Ideas?

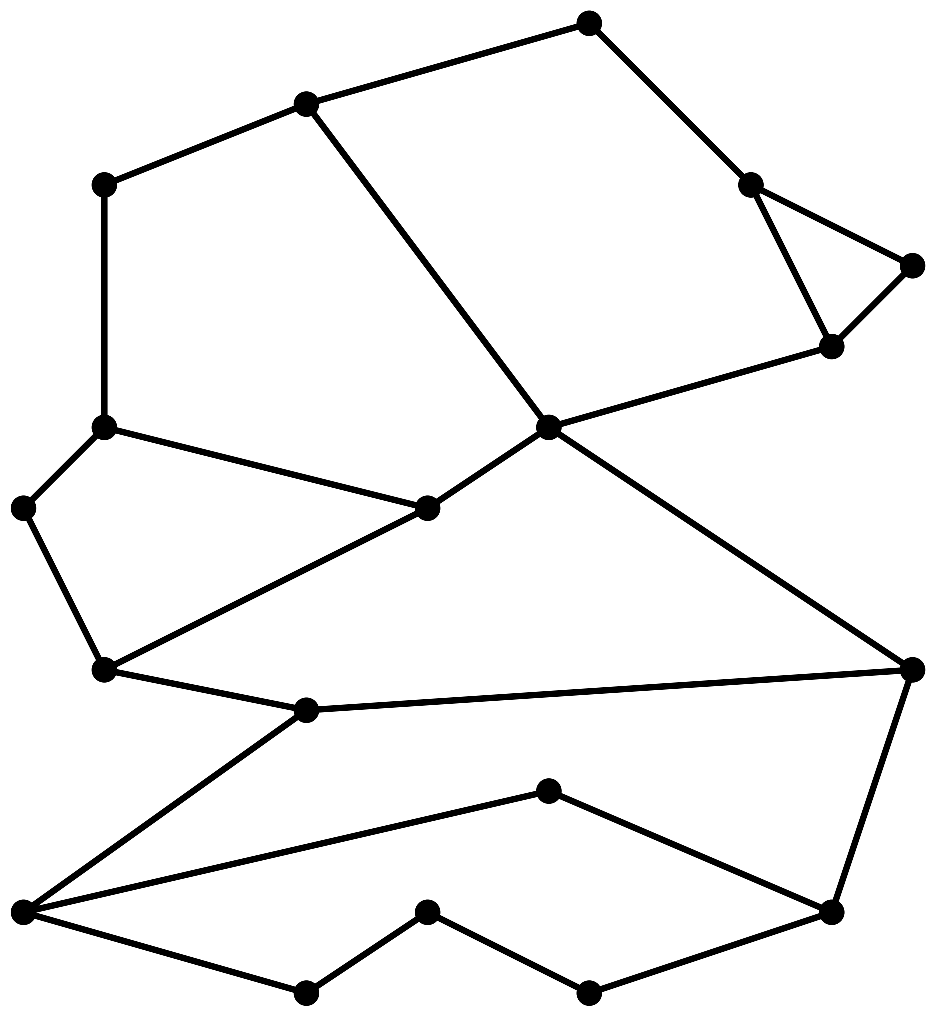
- What is the corresponding 1D problem?
- How to solve 1D problem?
- Can we reduce the problem to the 1D, e.g., first x, then y?
- query time, . . .?

next: slab decomposition for point location

Vertical Decomposition for Point Location

Slab decomposition

Point location

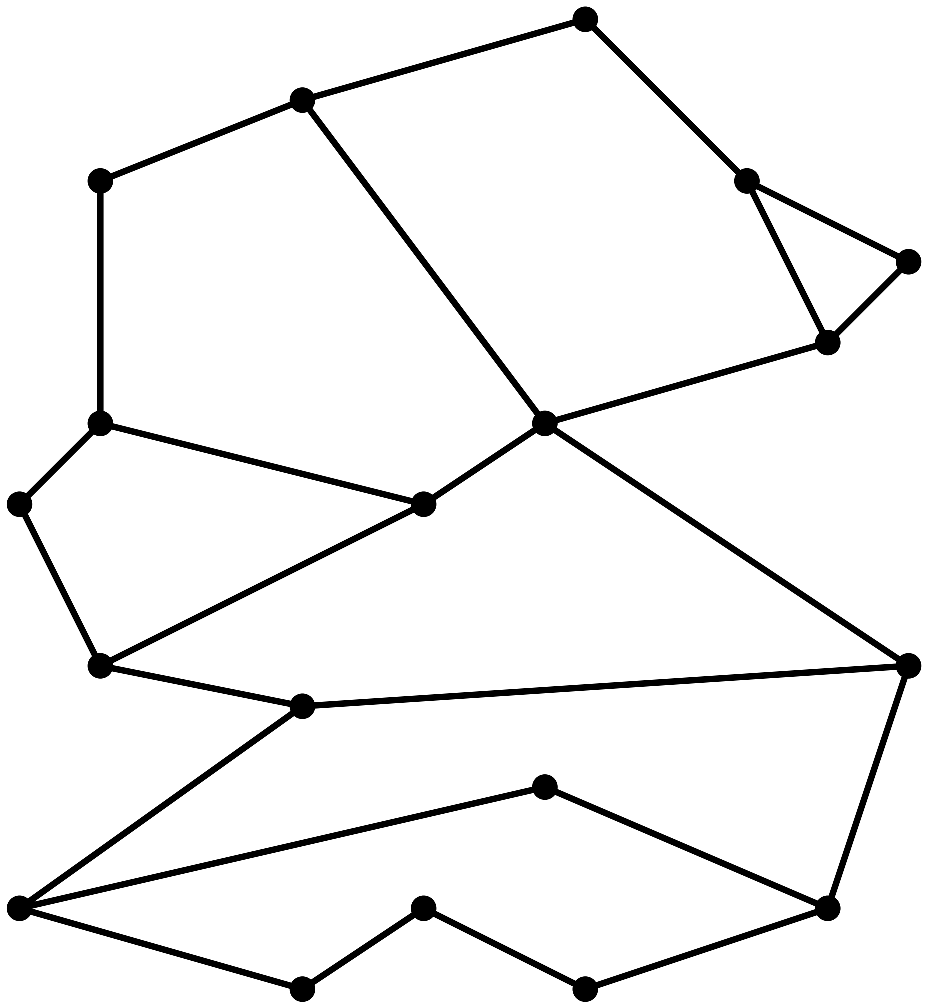


Point location problem: Preprocess a planar subdivision such that for any query point q, the face of the subdivision containing q can be given efficiently

Idea:

- reduce problem to 1D
- solve 1D problems using balanced binary search trees

Point location



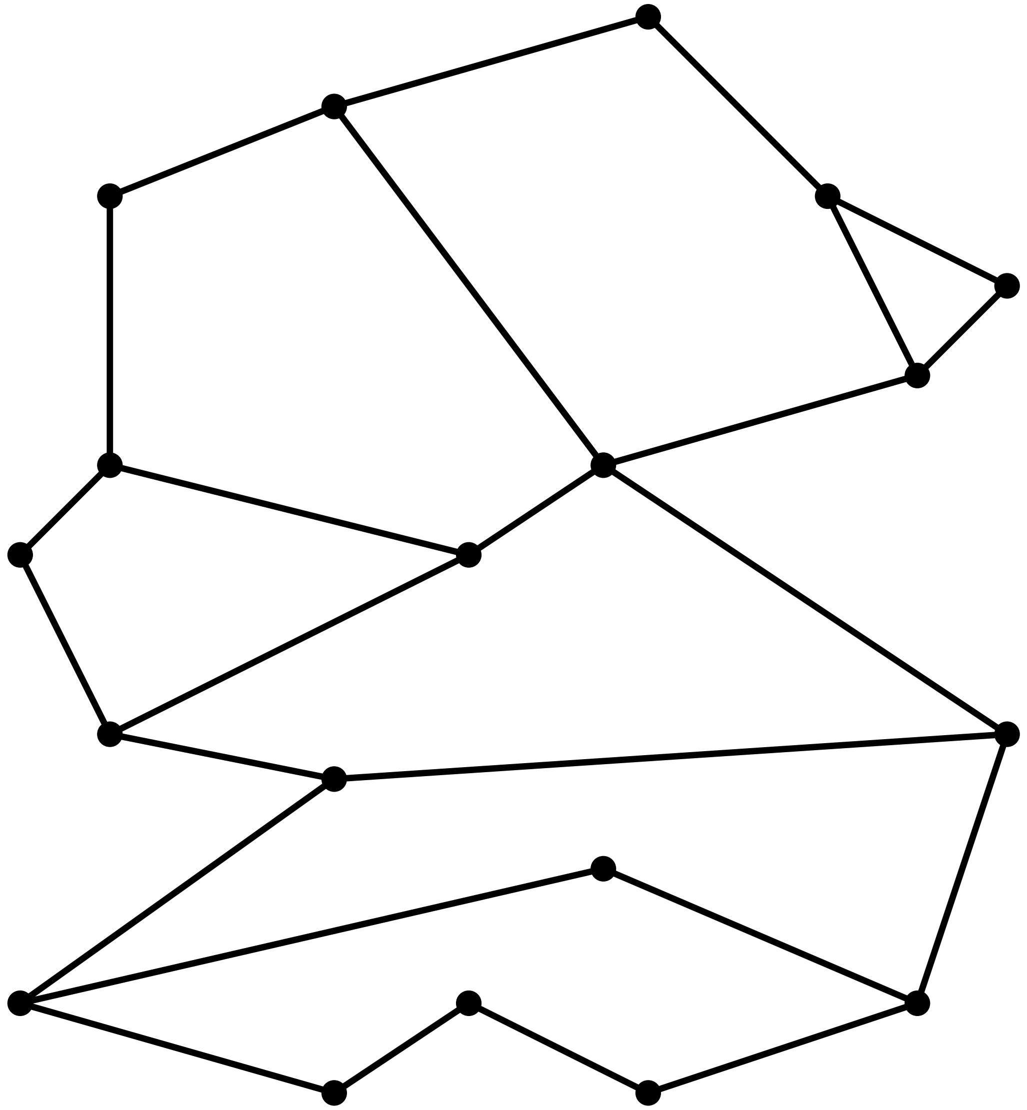
Point location problem: Preprocess a planar subdivision such that for any query point q, the face of the subdivision containing q can be given efficiently

Idea:

- reduce problem to 1D
- solve 1D problems using balanced binary search trees

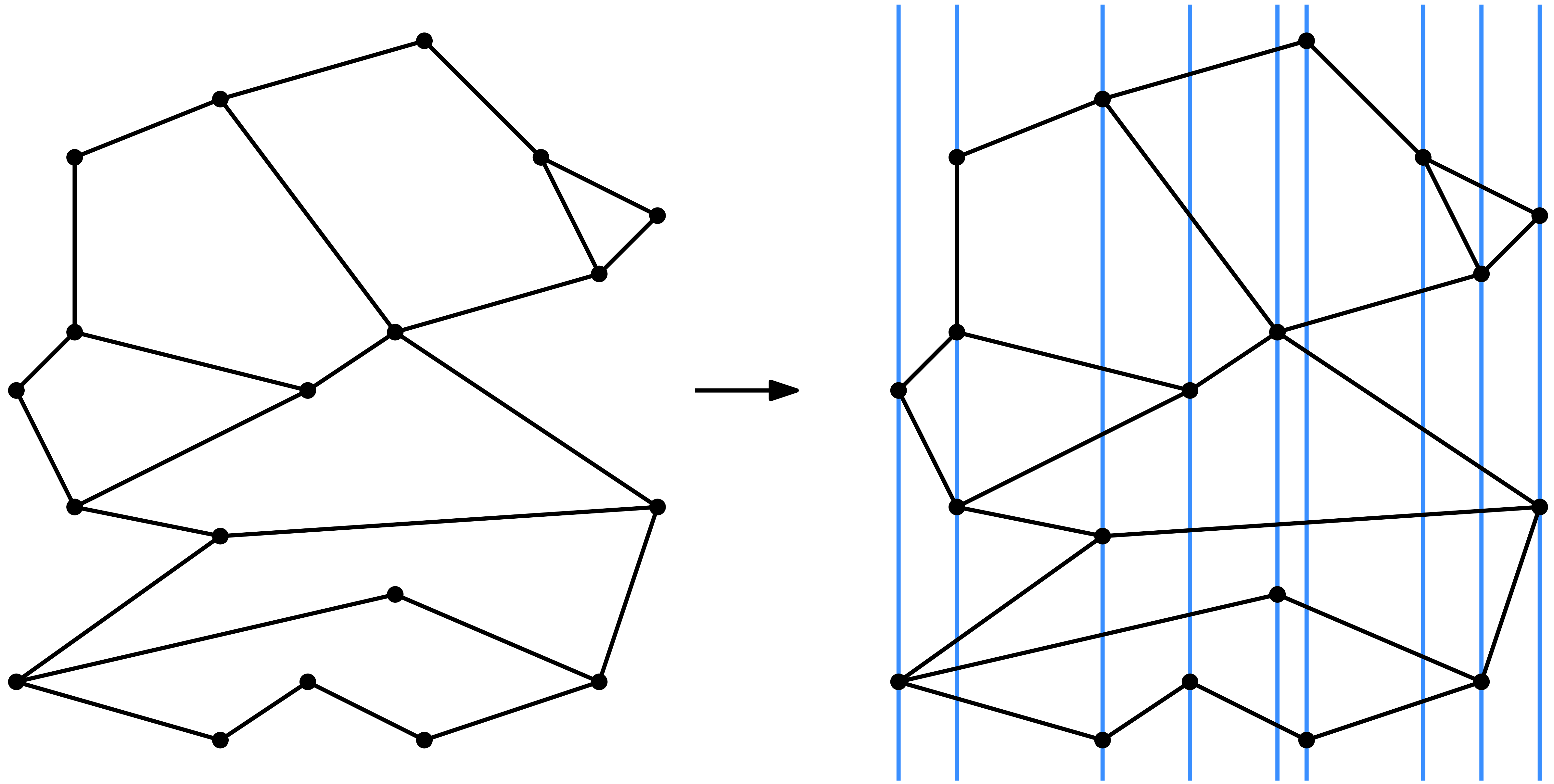
- **not** the data structure we will finally use
- 'first step' towards vertical decomposition

Idea



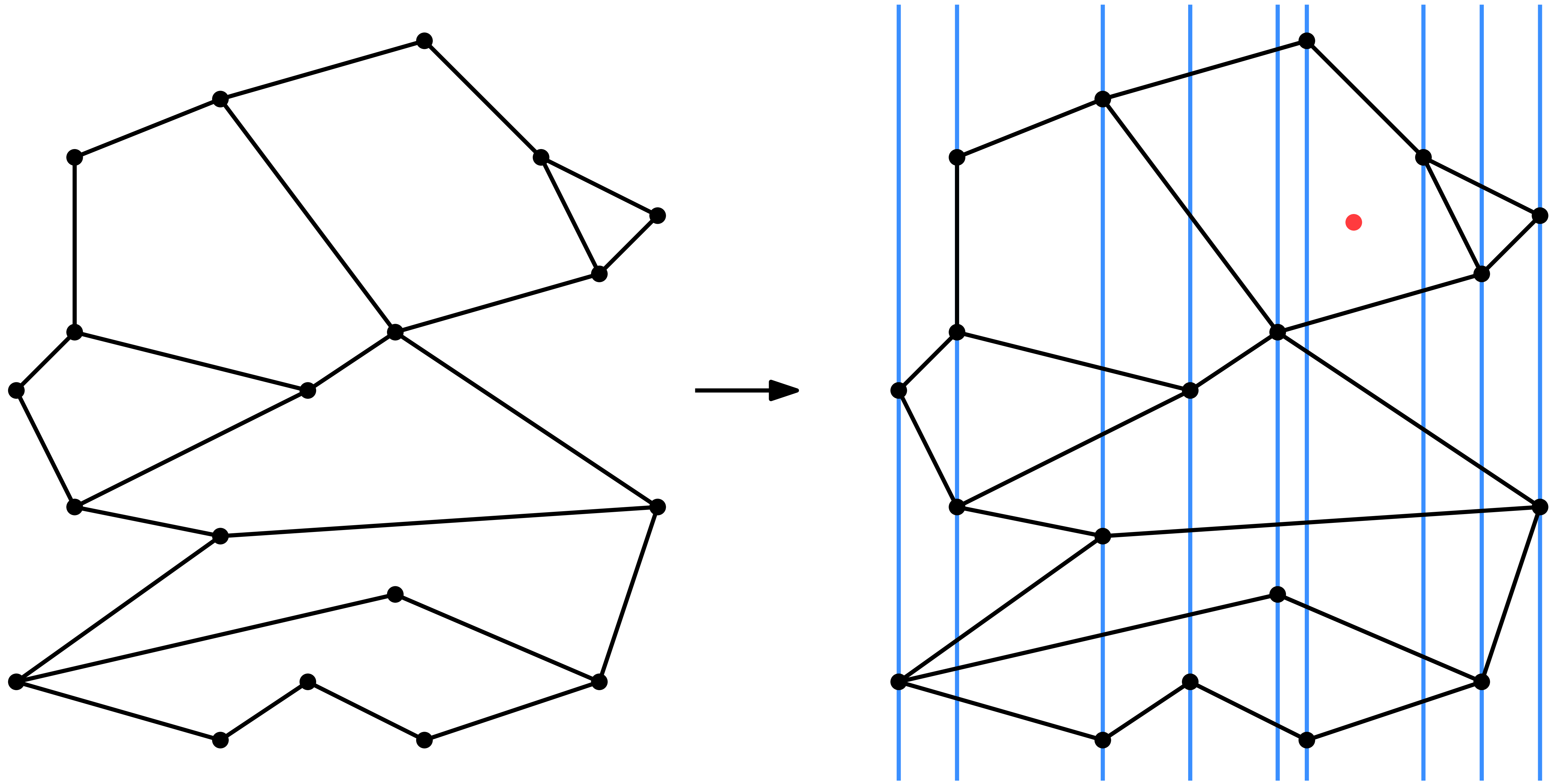
Idea: Draw vertical lines through all vertices, then do something for every vertical strip that appears

Idea



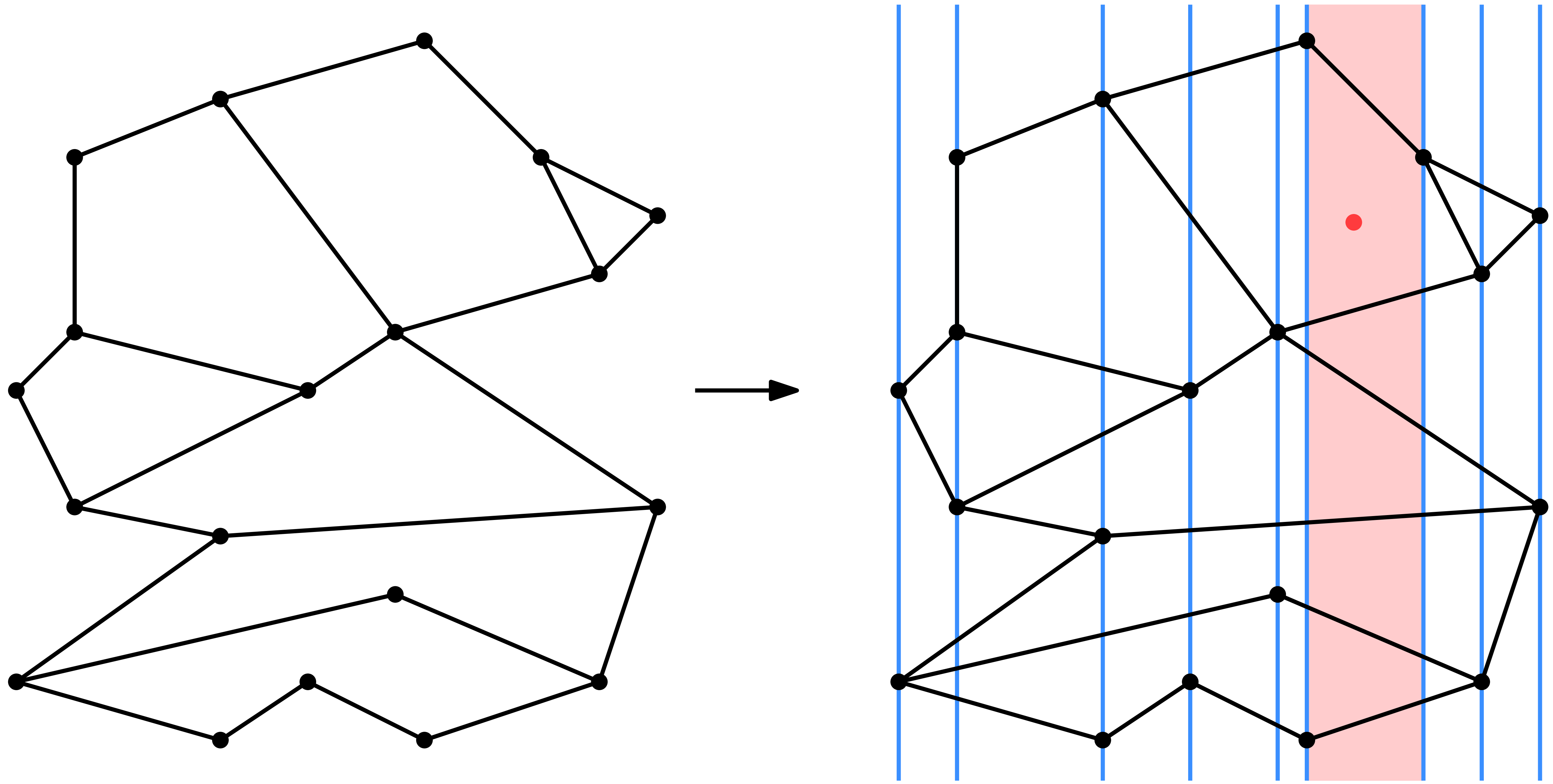
Idea: Draw vertical lines through all vertices, then do something for every vertical strip that appears

Idea



Idea: Draw vertical lines through all vertices, then do something for every vertical strip that appears

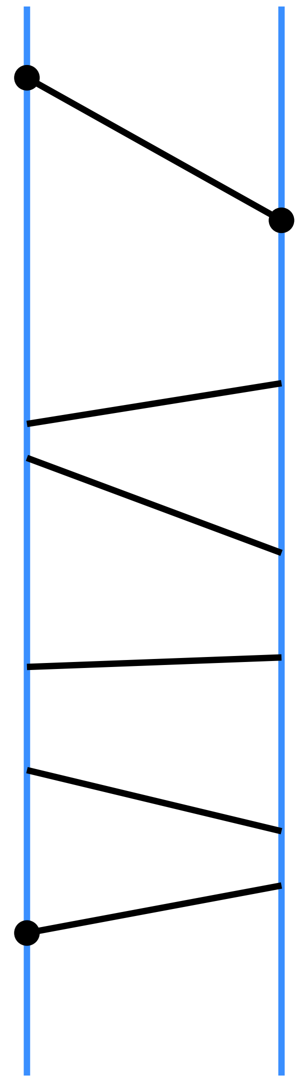
Idea



Idea: Draw vertical lines through all vertices, then do something for every vertical strip that appears

In one strip

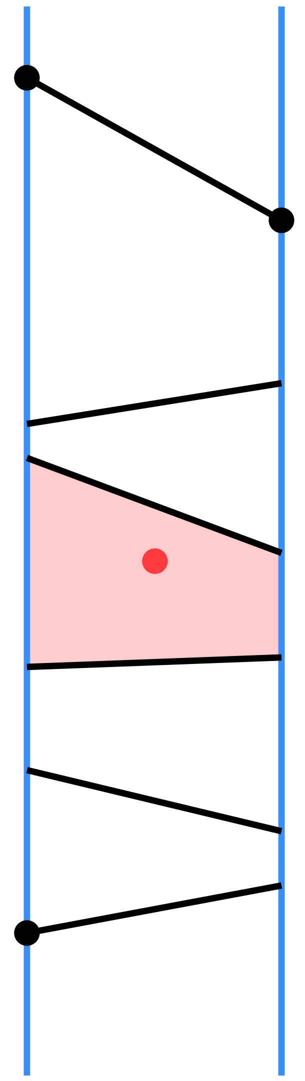
Inside a single strip, there is a well-defined bottom-to-top order on the line segments



In one strip

Inside a single strip, there is a well-defined bottom-to-top order on the line segments

Use this for a balanced binary search tree that is valid if the query point is in this strip

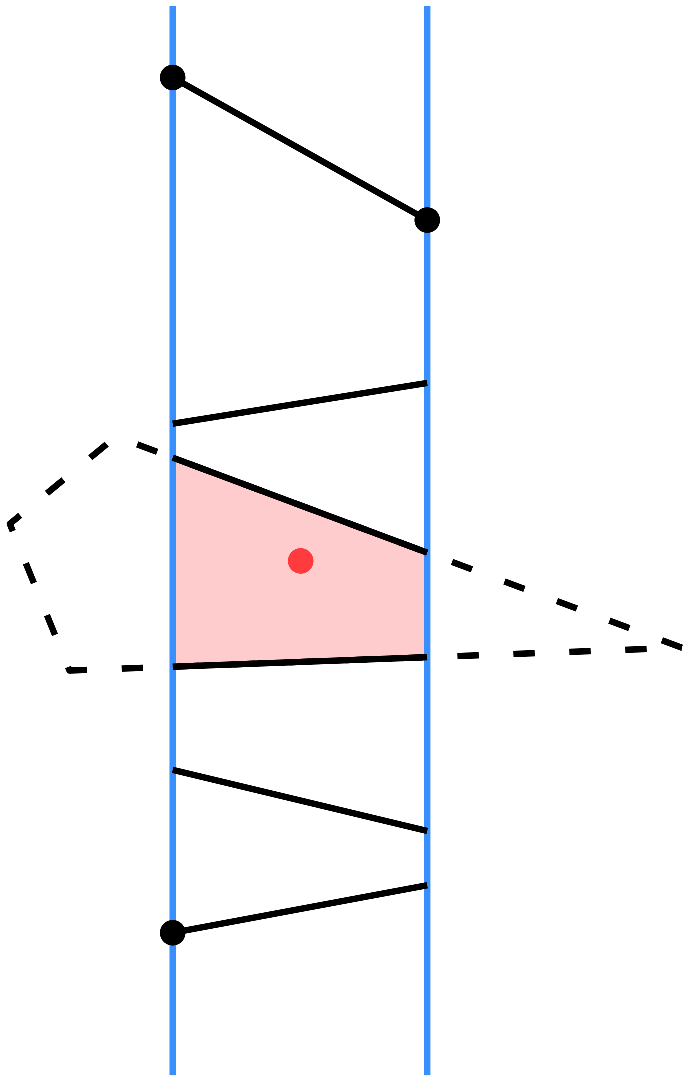


In one strip

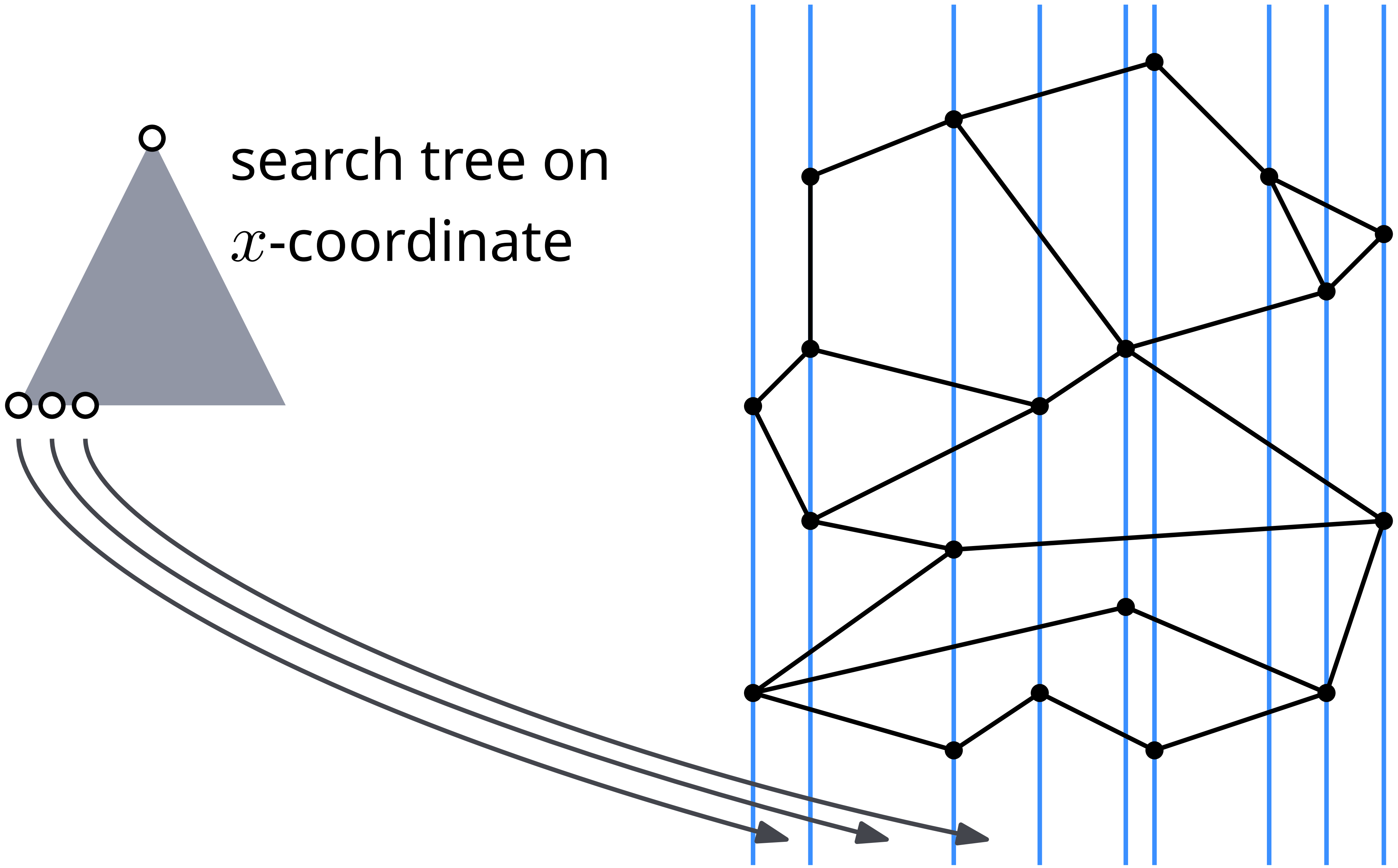
Inside a single strip, there is a well-defined bottom-to-top order on the line segments

Use this for a balanced binary search tree that is valid if the query point is in this strip

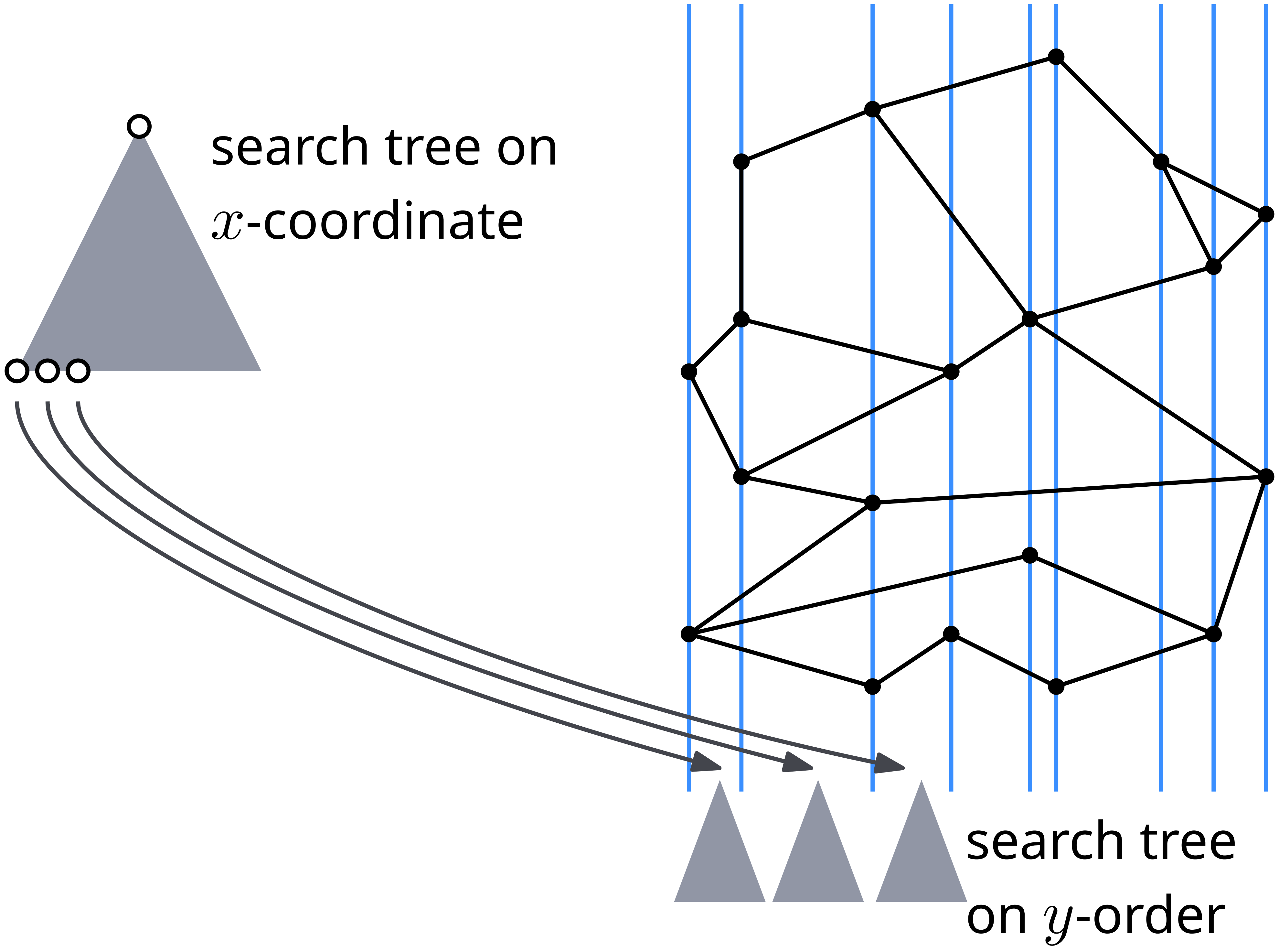
If we know between which edges the point is, we found the face of the subdivision containing the point



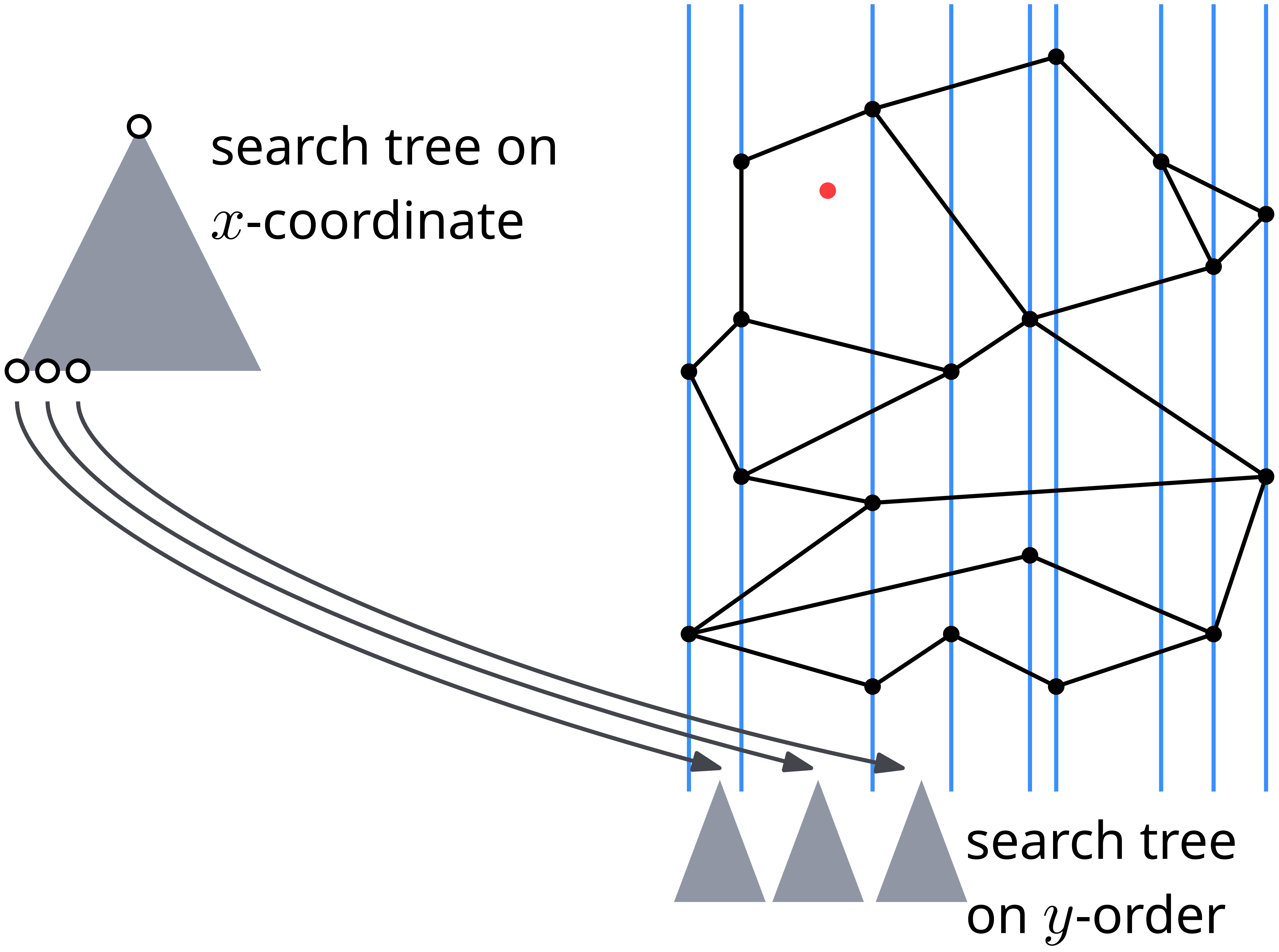
Slab Decomposition



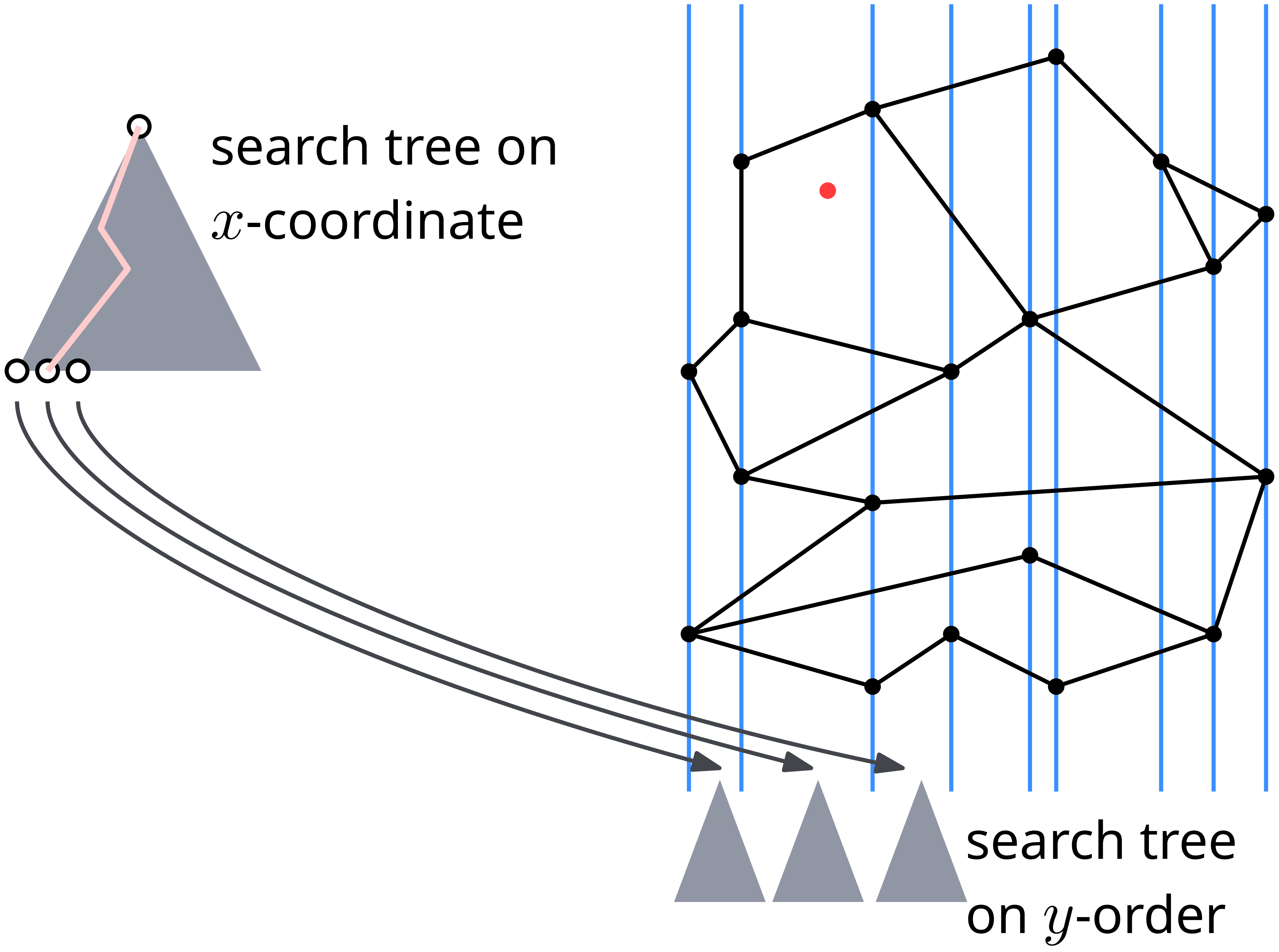
Slab Decomposition



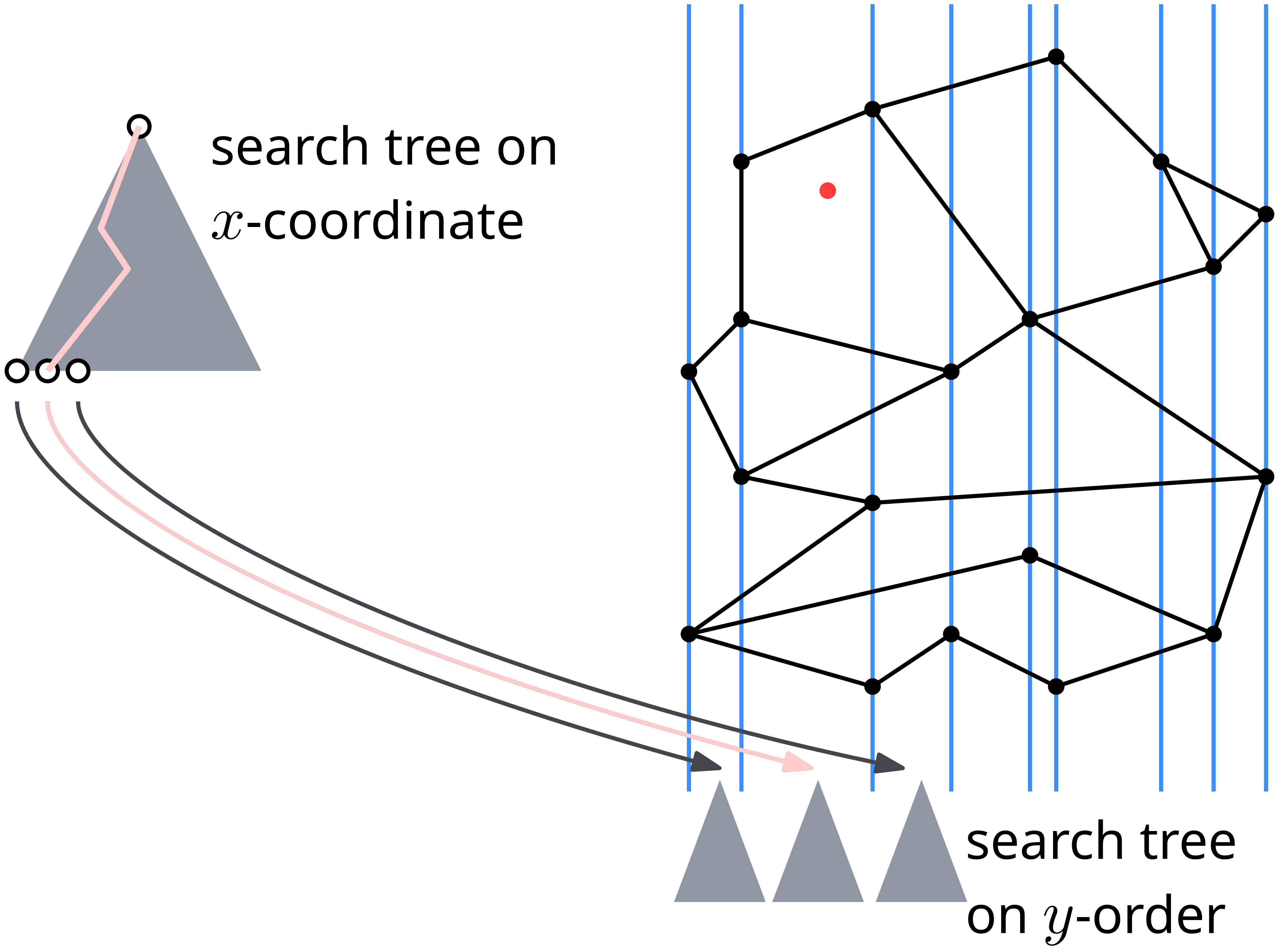
Slab Decomposition



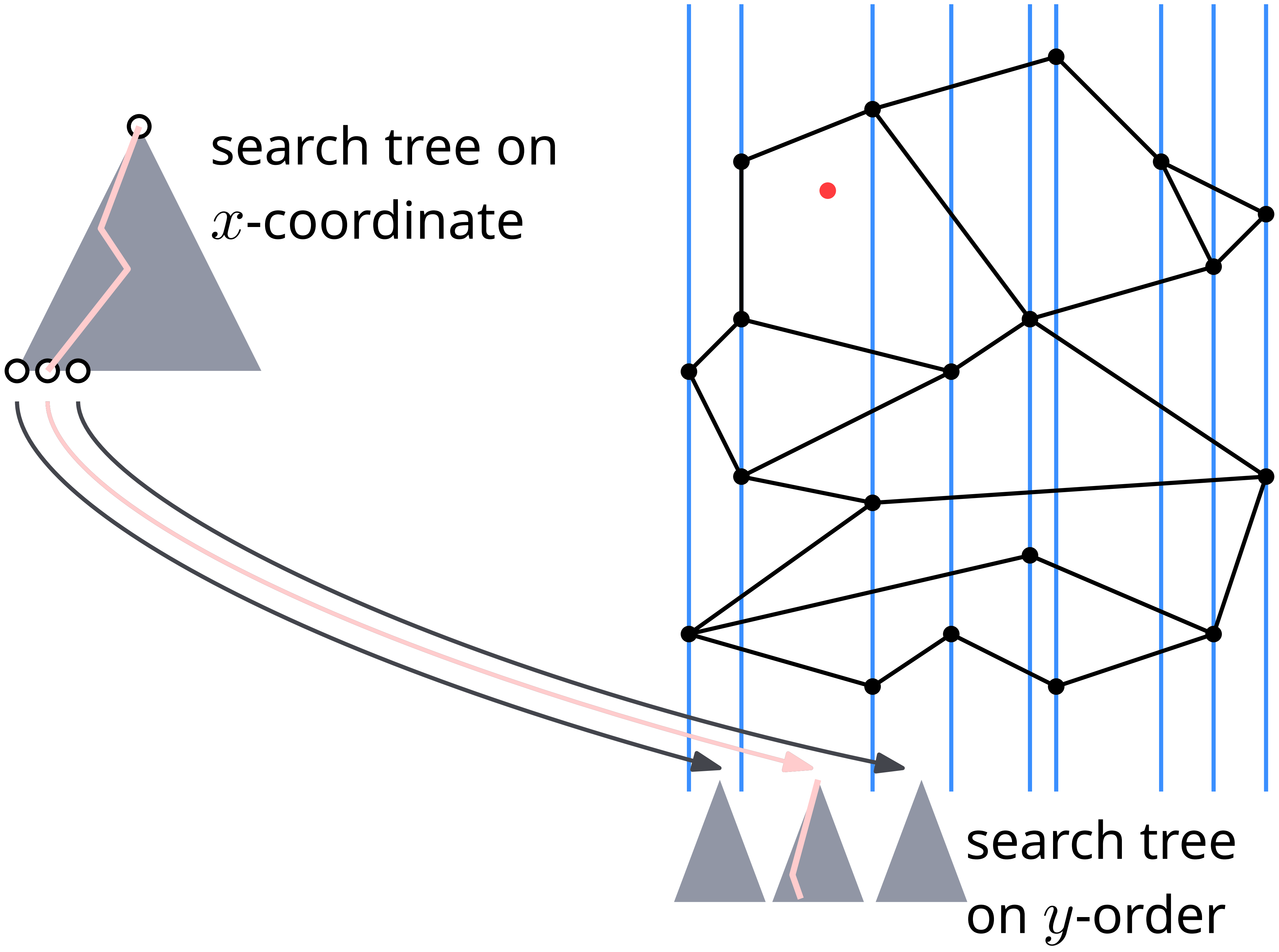
Slab Decomposition



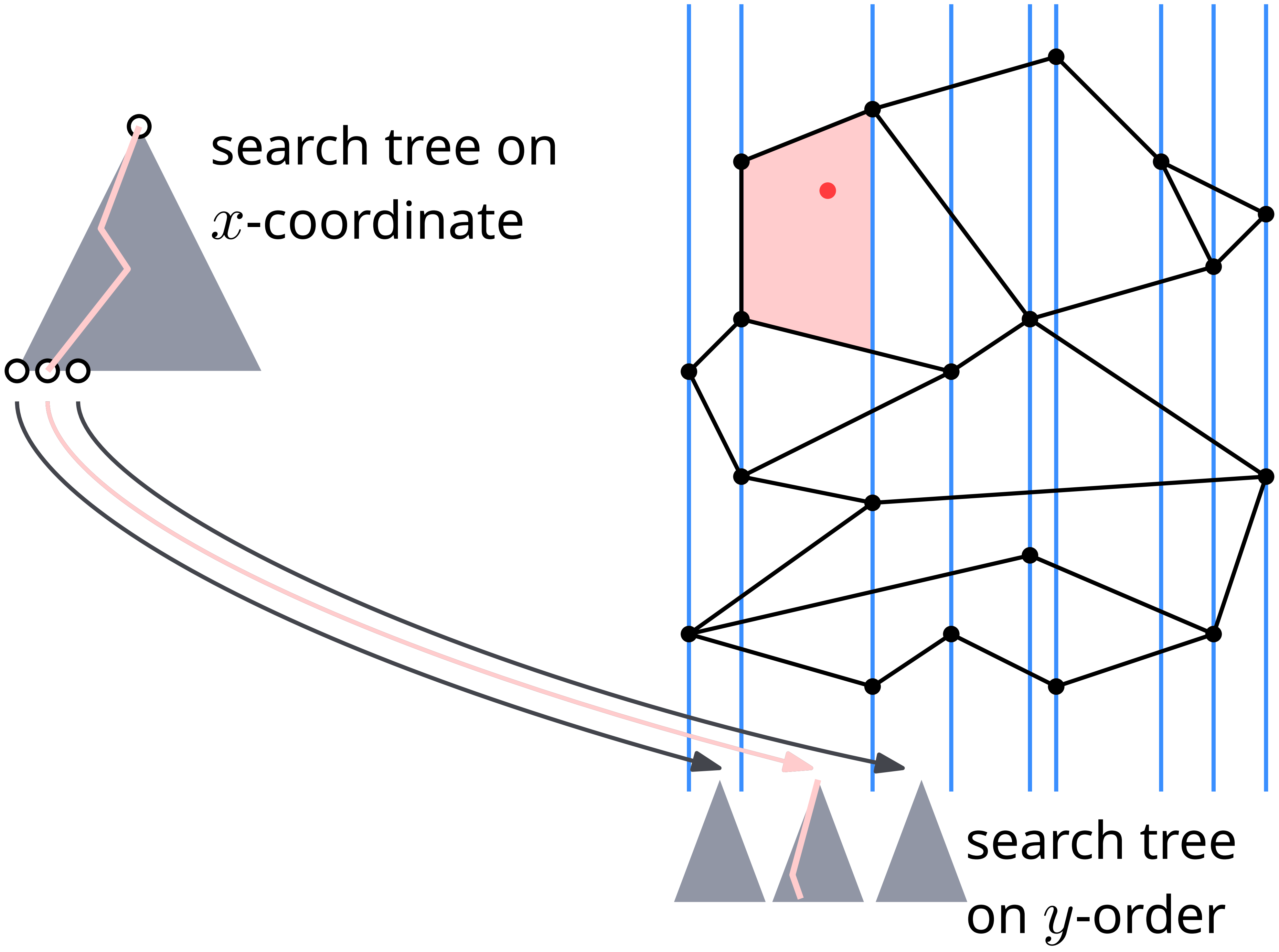
Slab Decomposition



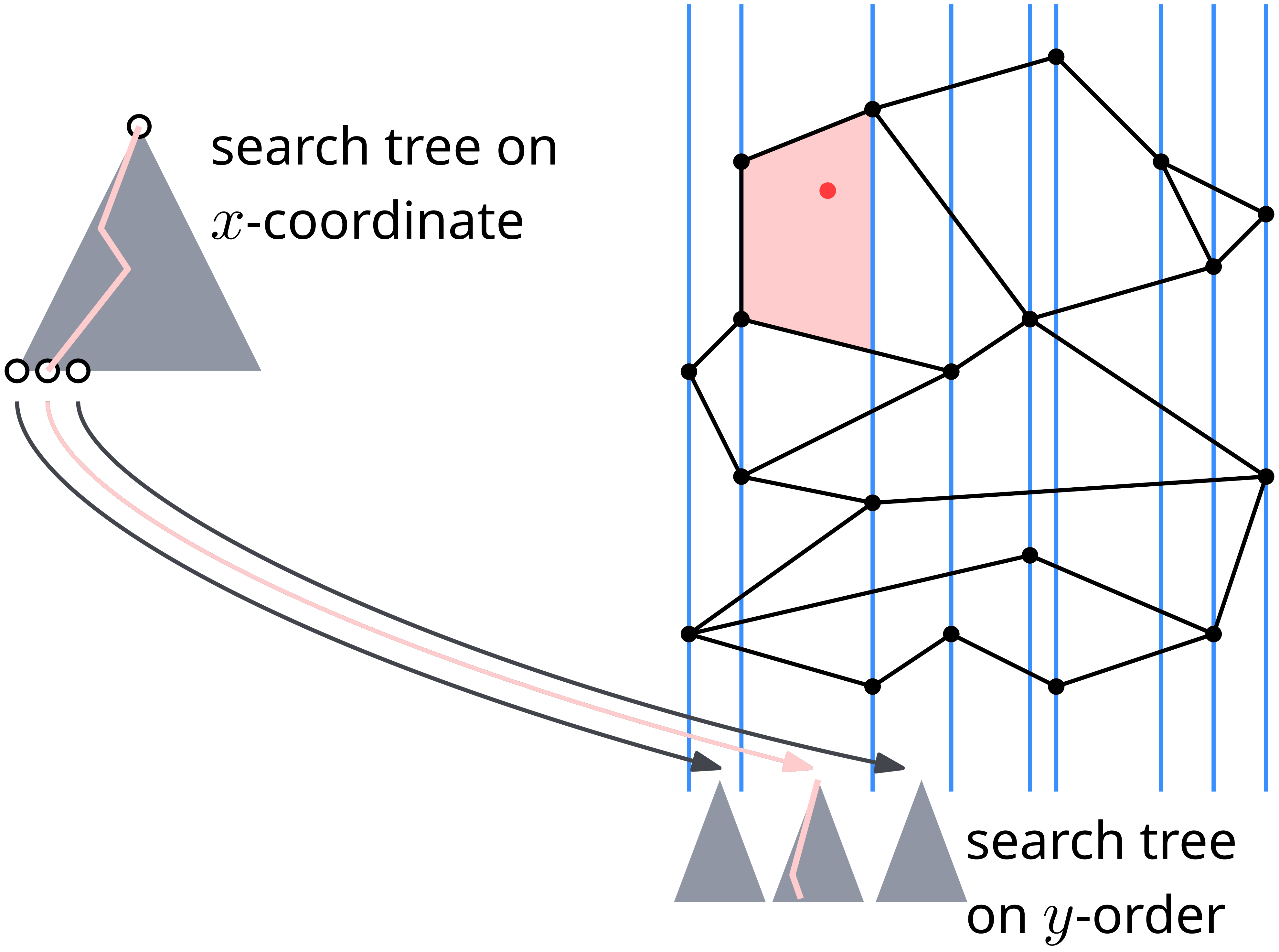
Slab Decomposition



Slab Decomposition



Slab Decomposition



Data Structure question:

- query time?
- space requirements?

Quiz: Slab Decomposition

What is the running time of a query if the subdivision has n edges?
How much space to we need to store the slab decomposition?

A: queries $O(\log n)$, space $\Theta(n)$

B: queries $O(\log n)$, space $\Theta(n^2)$

C: queries $O(\log^2 n)$, space $\Theta(n)$

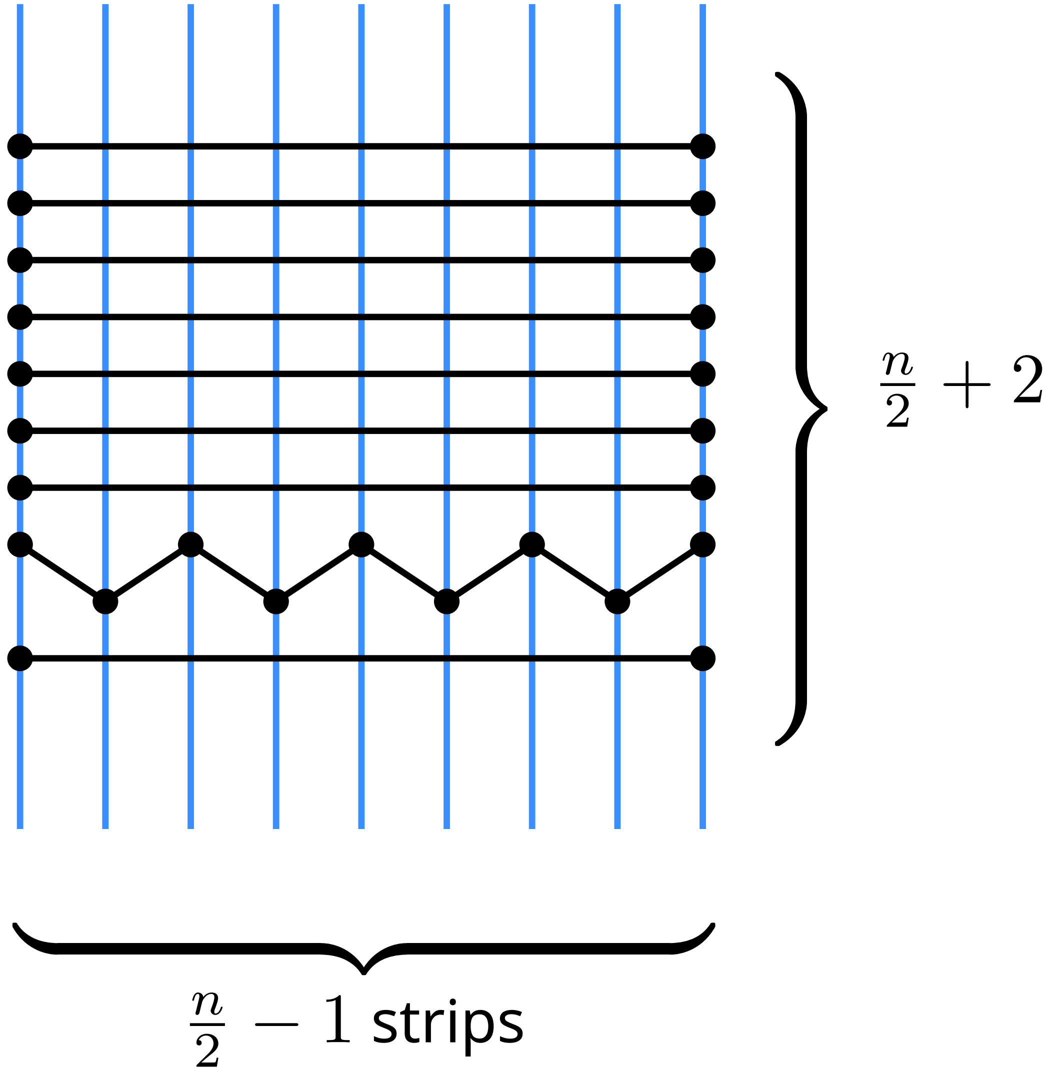
Quiz: Slab Decomposition

What is the running time of a query if the subdivision has n edges?
How much space to we need to store the slab decomposition?

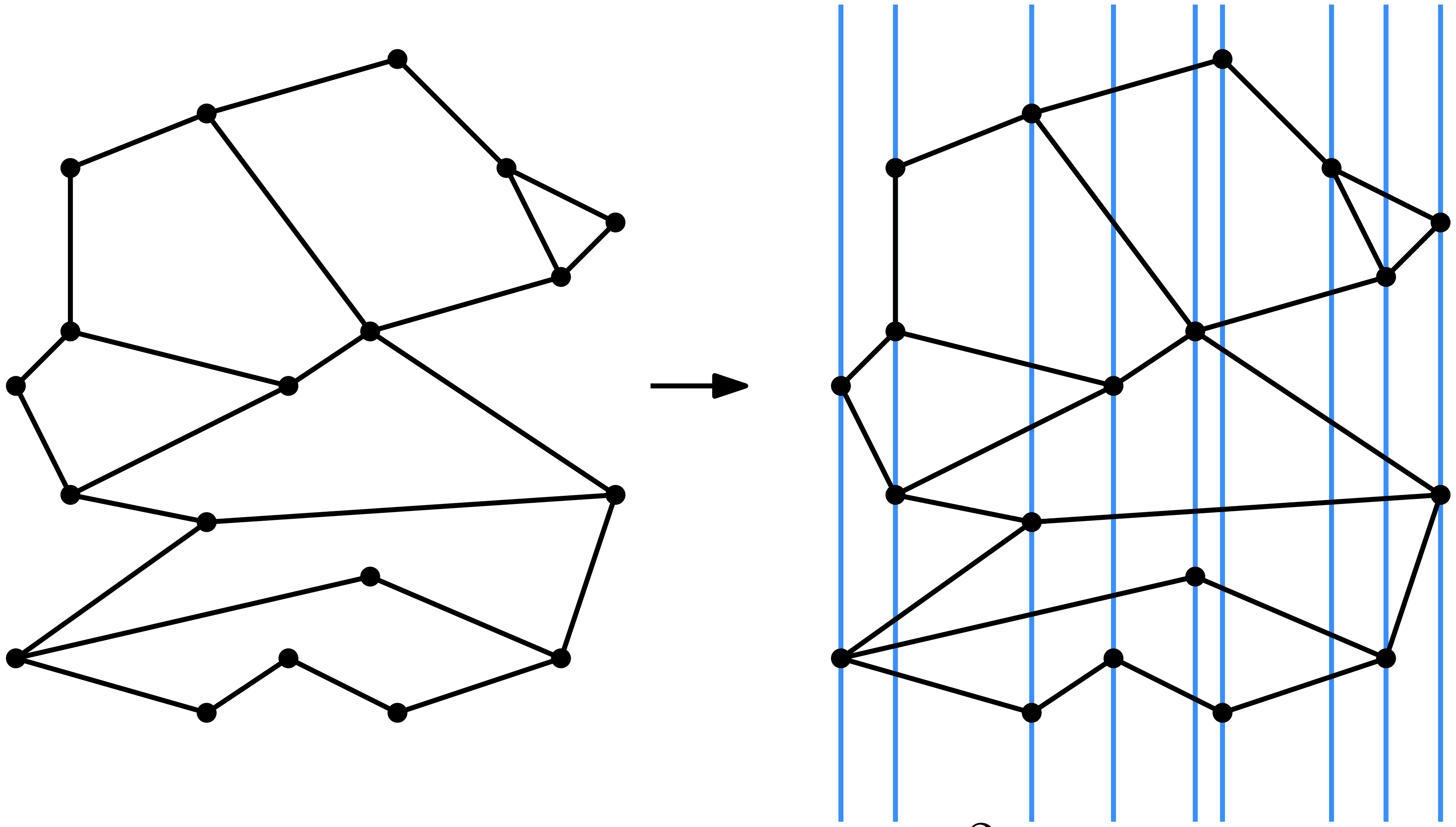
A: queries $O(\log n)$, space $\Theta(n)$

B: queries $O(\log n)$, space $\Theta(n^2)$

C: queries $O(\log^2 n)$, space $\Theta(n)$

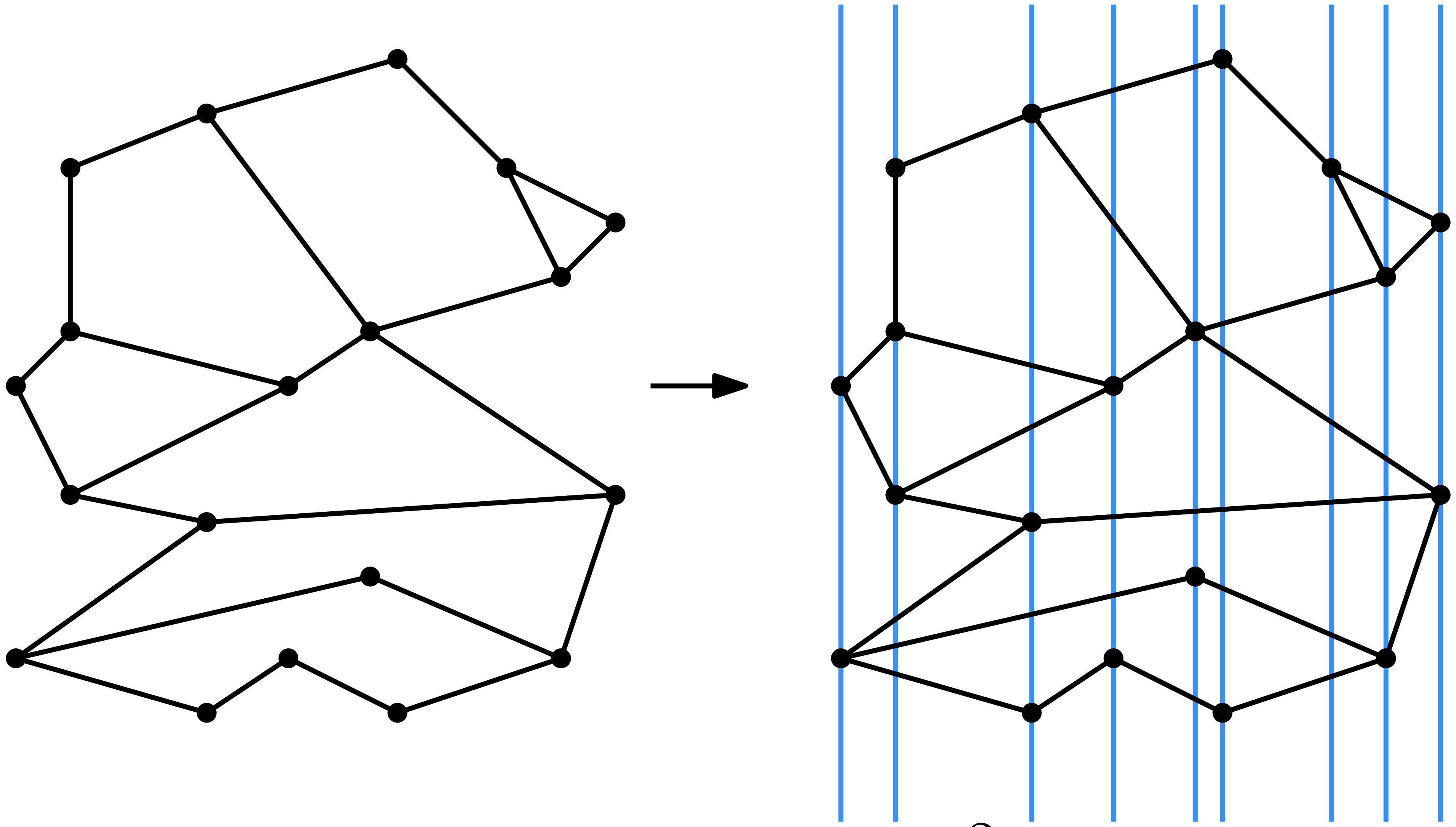


Avoid $O(n^2)$ storage?



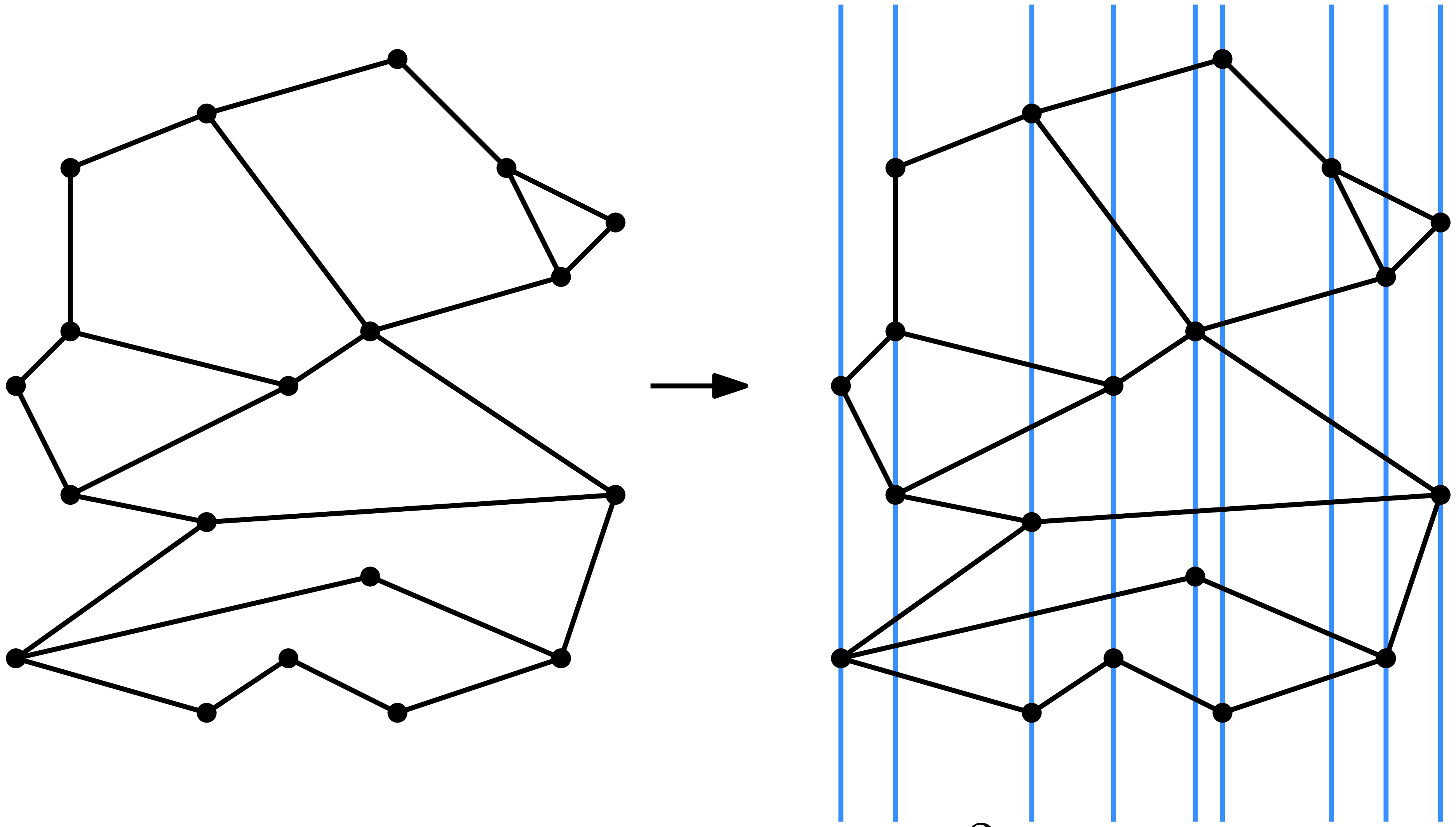
- query time good, storage of $O(n^2)$ too much

Avoid $O(n^2)$ storage?



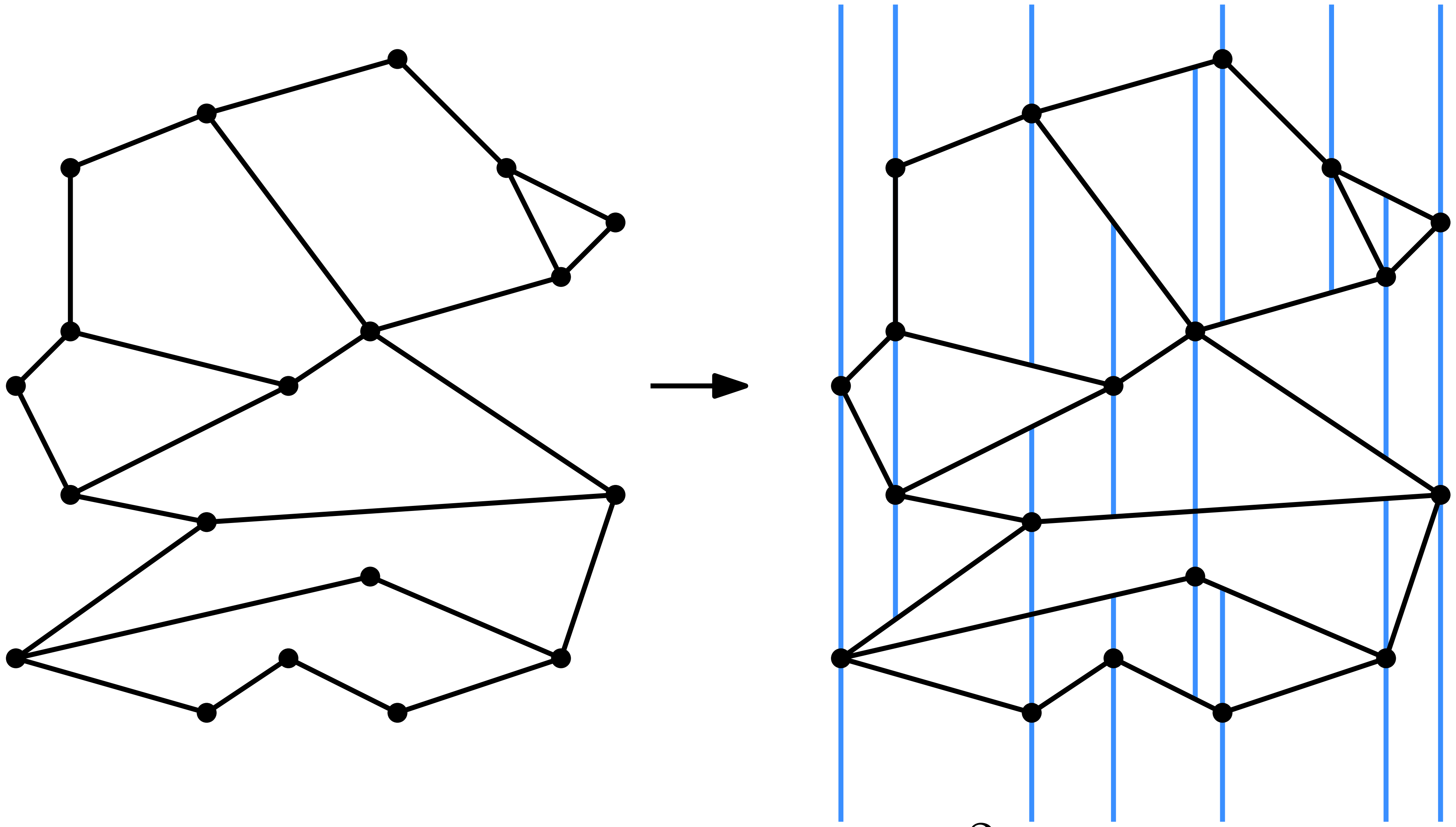
- query time good, storage of $O(n^2)$ too much
- $O(n^2)$, because every vertical line intersects $O(n)$ cells

Avoid $O(n^2)$ storage?



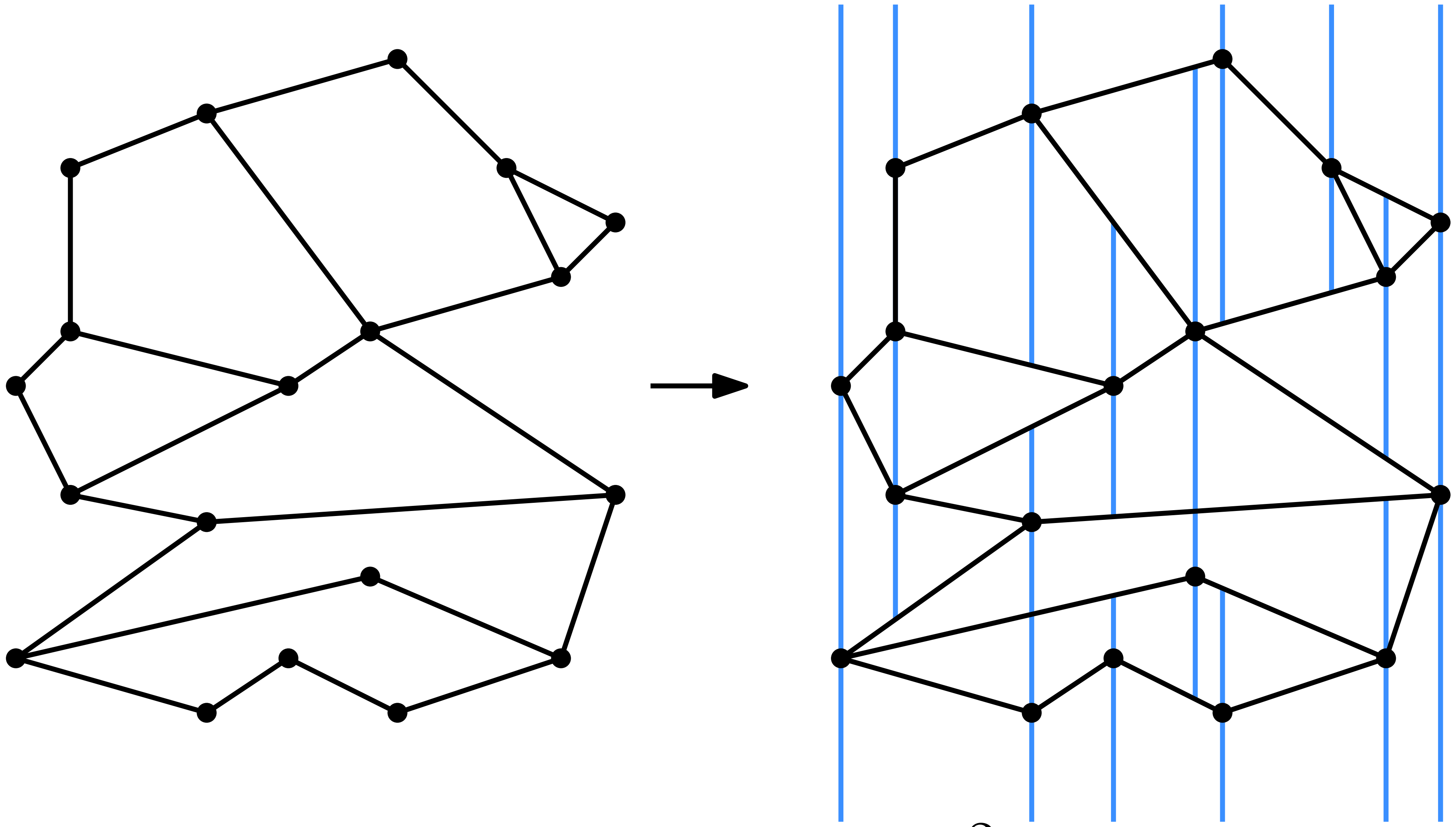
- query time good, storage of $O(n^2)$ too much
- $O(n^2)$, because every vertical line intersects $O(n)$ cells
- $O(n)$ storage? Fewer intersections?

Avoid $O(n^2)$ storage?



- query time good, storage of $O(n^2)$ too much
- $O(n^2)$, because every vertical line intersects $O(n)$ cells
- $O(n)$ storage? Fewer intersections?
- interrupt vertical line, when it would intersect an edge

Avoid $O(n^2)$ storage?



next:

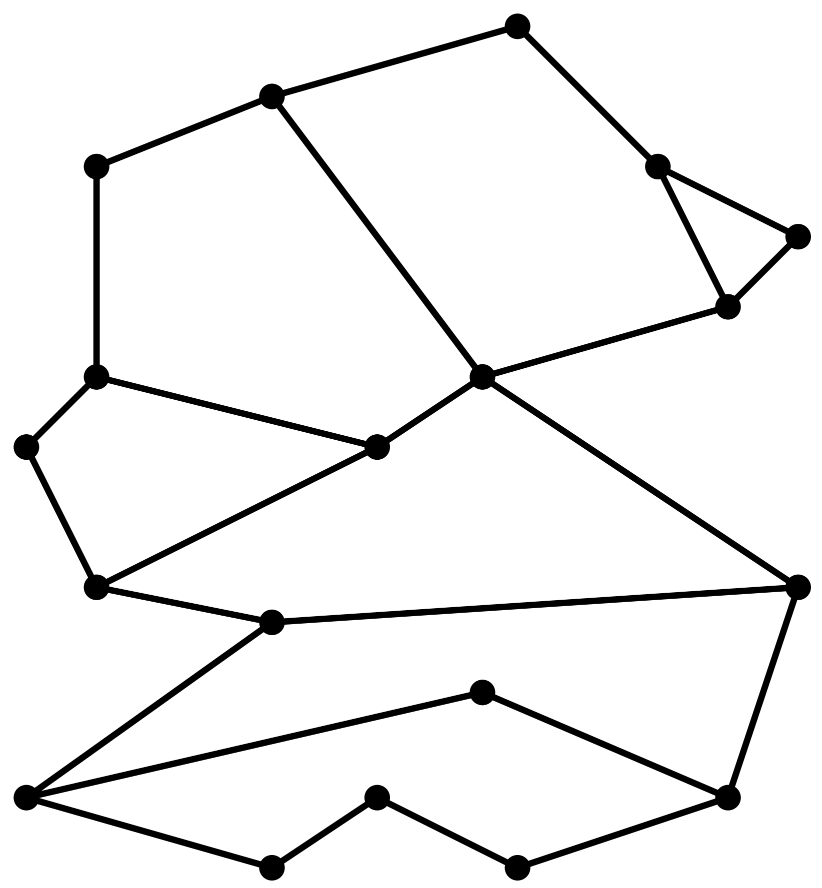
Vertical Decomposition

- query time good, storage of $O(n^2)$ too much
- $O(n^2)$, because every vertical line intersects $O(n)$ cells
- $O(n)$ storage? Fewer intersections?
- interrupt vertical line, when it would intersect an edge

Vertical Decomposition for Point Location

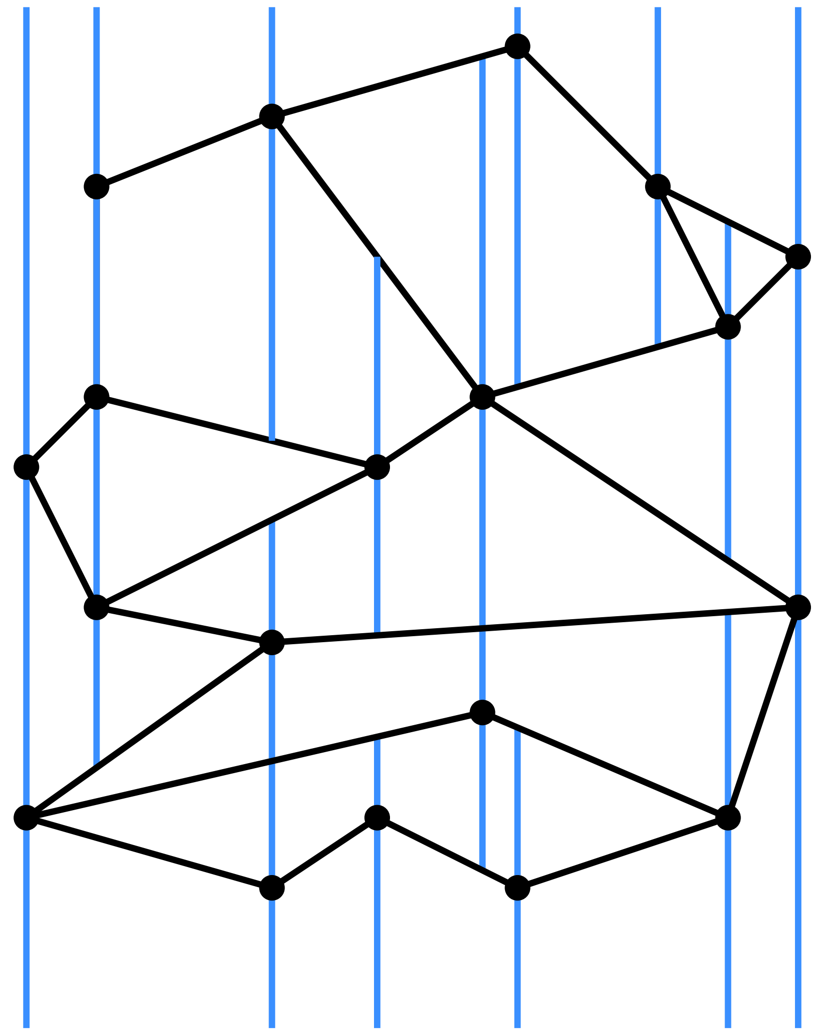
Vertical decomposition: concept

Vertical decomposition



Vertical decomposition

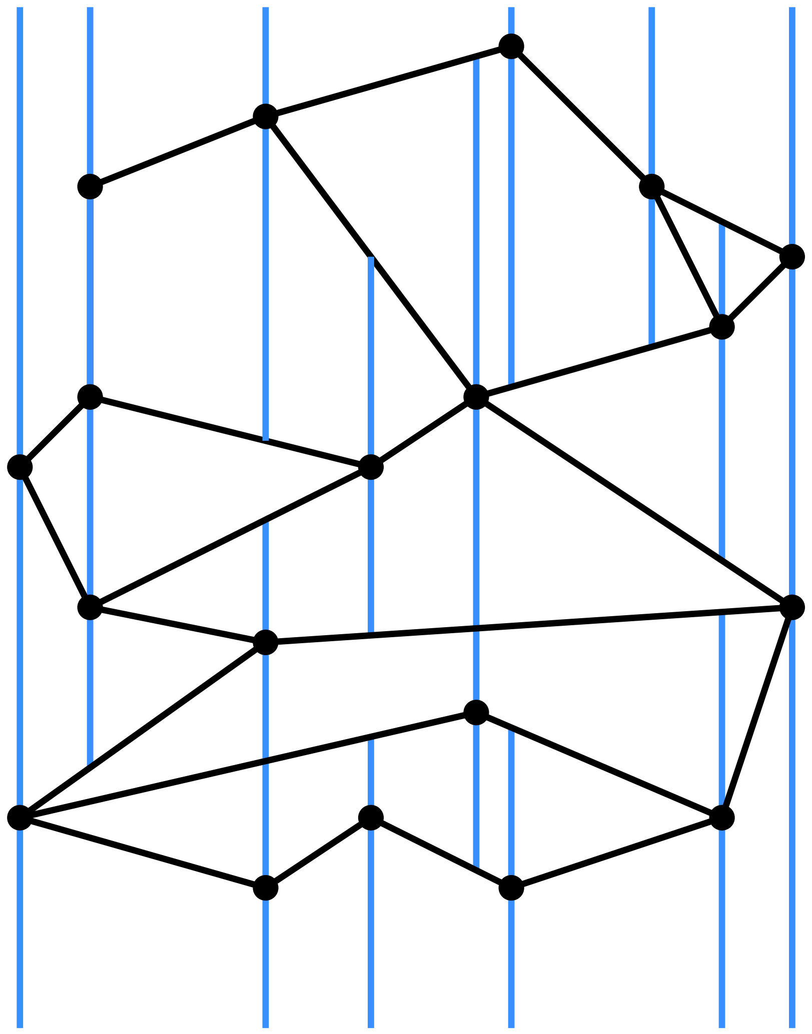
Suppose we draw vertical extensions from every vertex up and down, *but only until the next line segment*



Vertical decomposition

Suppose we draw vertical extensions from every vertex up and down, *but only until the next line segment*

This is called the vertical decomposition or trapezoidal decomposition

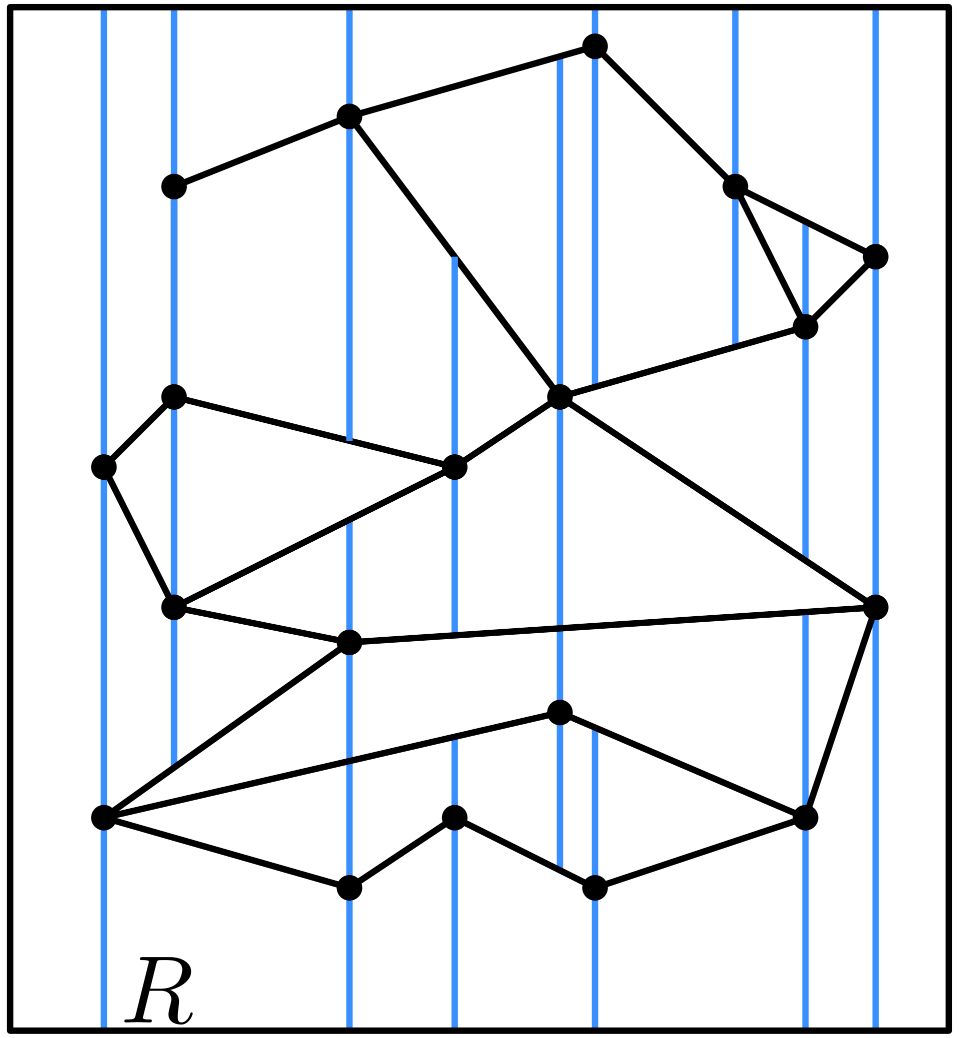


Vertical decomposition

Suppose we draw vertical extensions from every vertex up and down, *but only until the next line segment*

This is called the vertical decomposition or trapezoidal decomposition

- Assume we have a bounding box R that encloses all line segments that define the subdivision

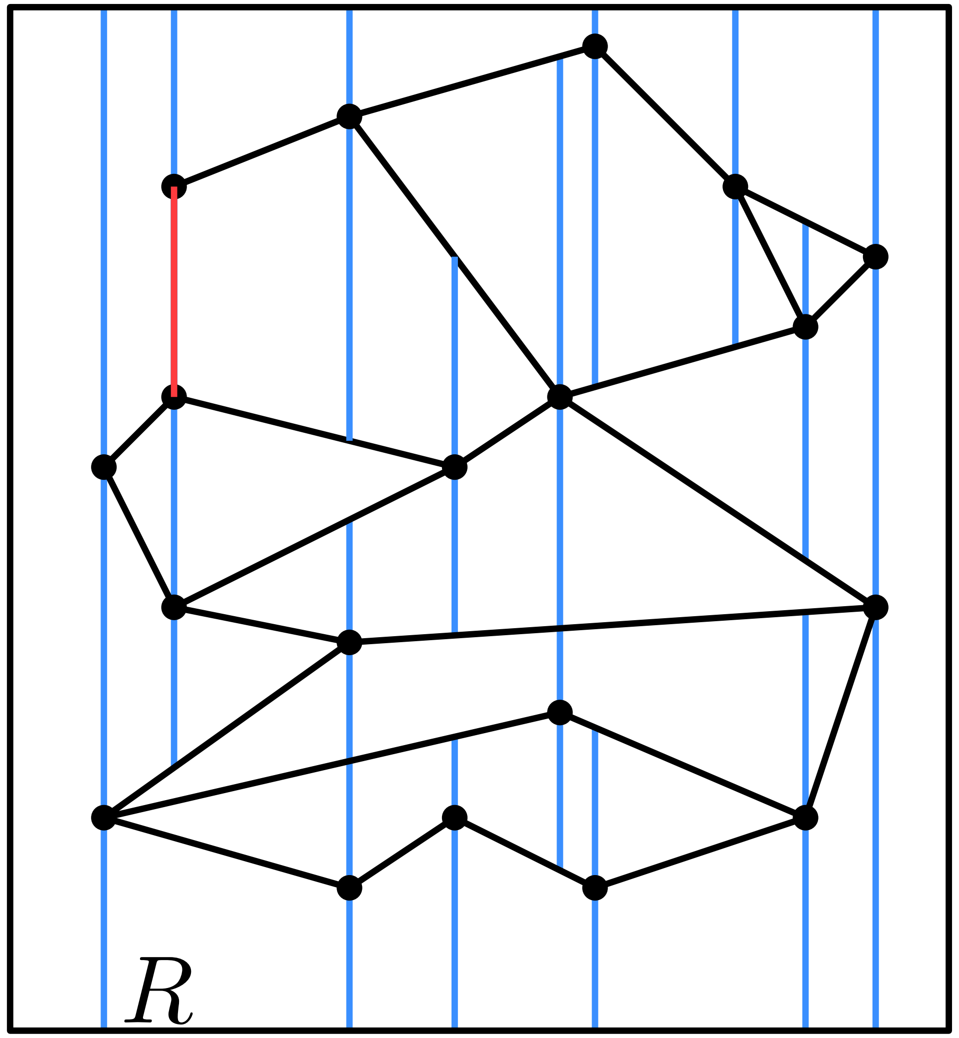


Vertical decomposition

Suppose we draw vertical extensions from every vertex up and down, *but only until the next line segment*

This is called the vertical decomposition or trapezoidal decomposition

- Assume we have a bounding box R that encloses all line segments that define the subdivision
- Assume the input line segments are not vertical

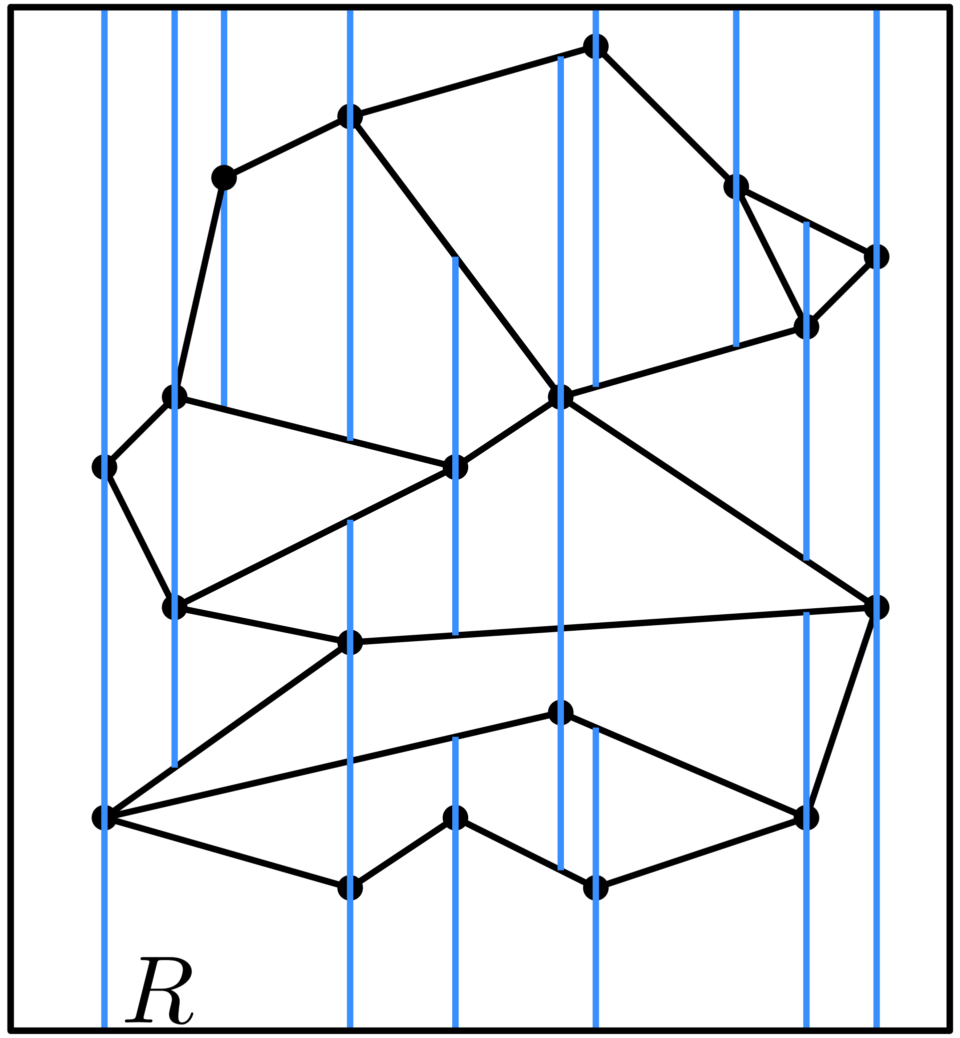


Vertical decomposition

Suppose we draw vertical extensions from every vertex up and down, *but only until the next line segment*

This is called the vertical decomposition or trapezoidal decomposition

- Assume we have a bounding box R that encloses all line segments that define the subdivision
- Assume the input line segments are not vertical

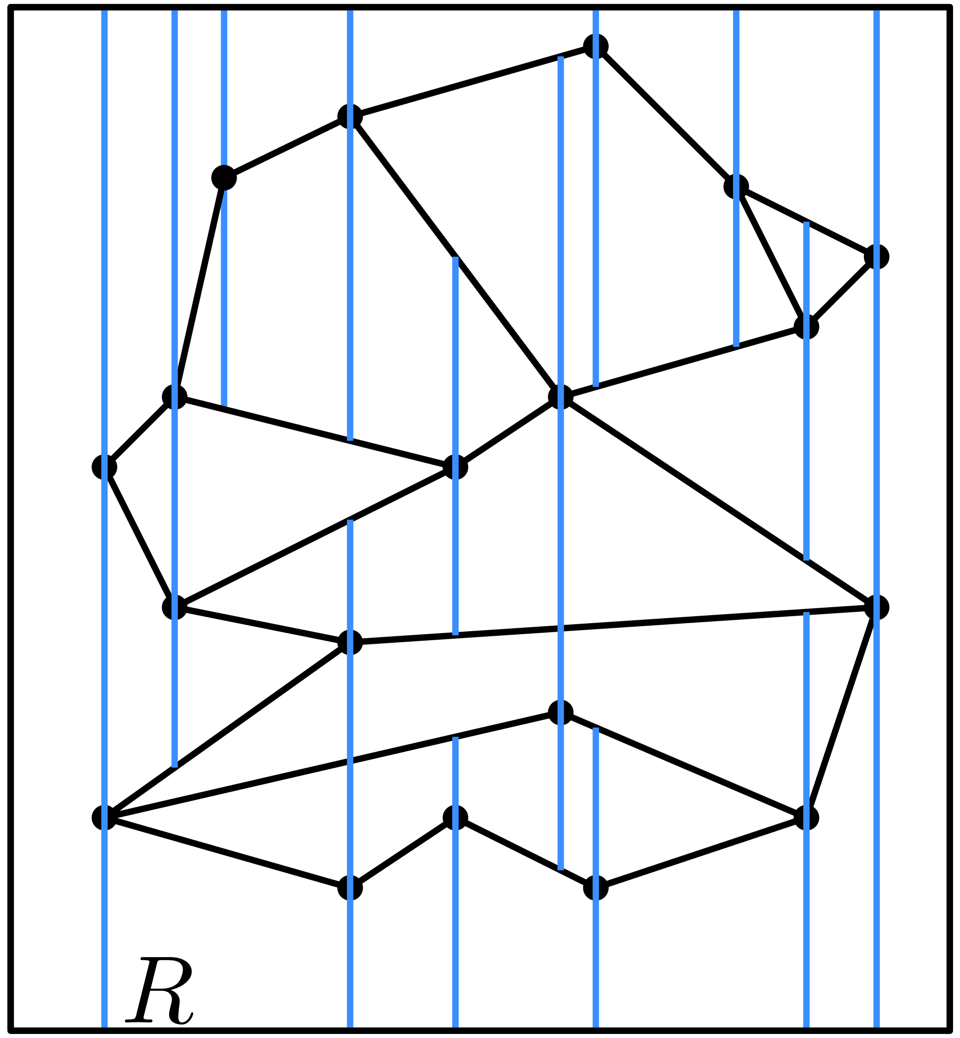


Vertical decomposition

Suppose we draw vertical extensions from every vertex up and down, *but only until the next line segment*

This is called the vertical decomposition or trapezoidal decomposition

- Assume we have a bounding box R that encloses all line segments that define the subdivision
- Assume the input line segments are not vertical
- Assume every vertex has a distinct x-coordinate

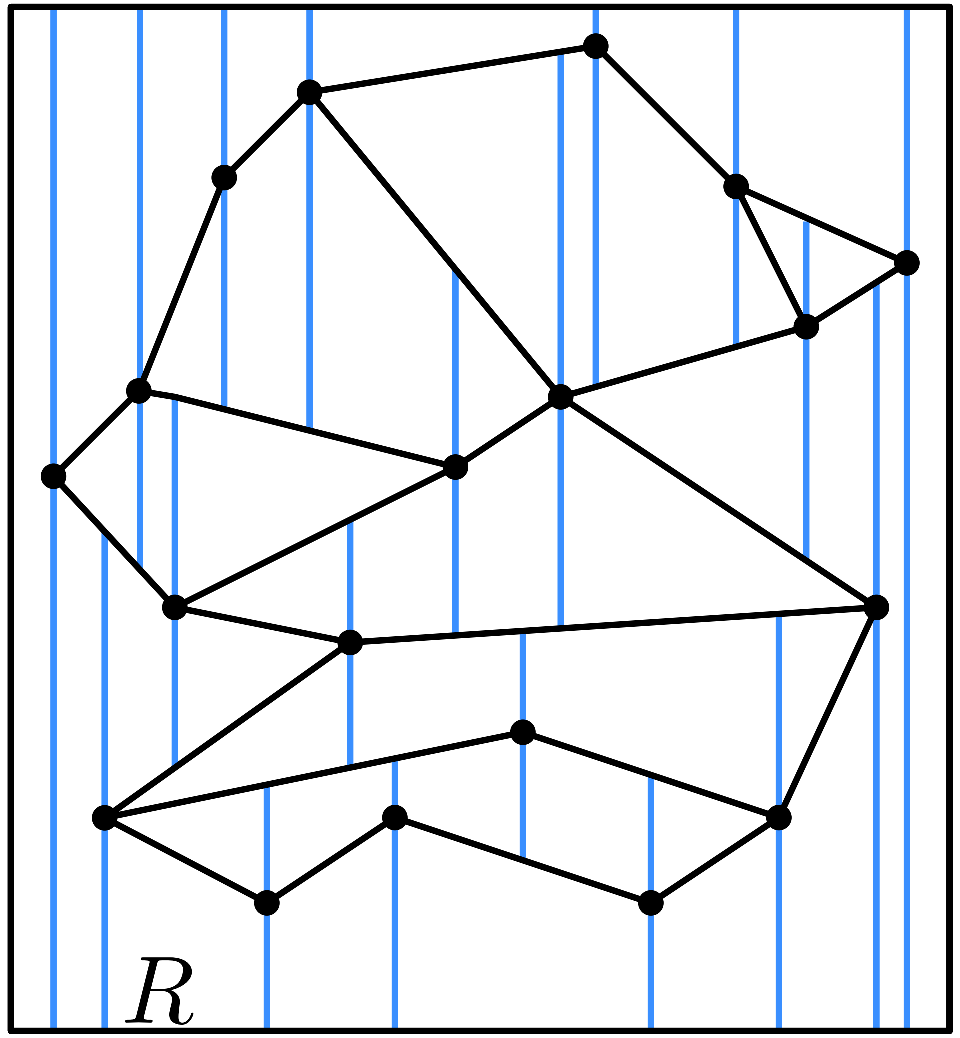


Vertical decomposition

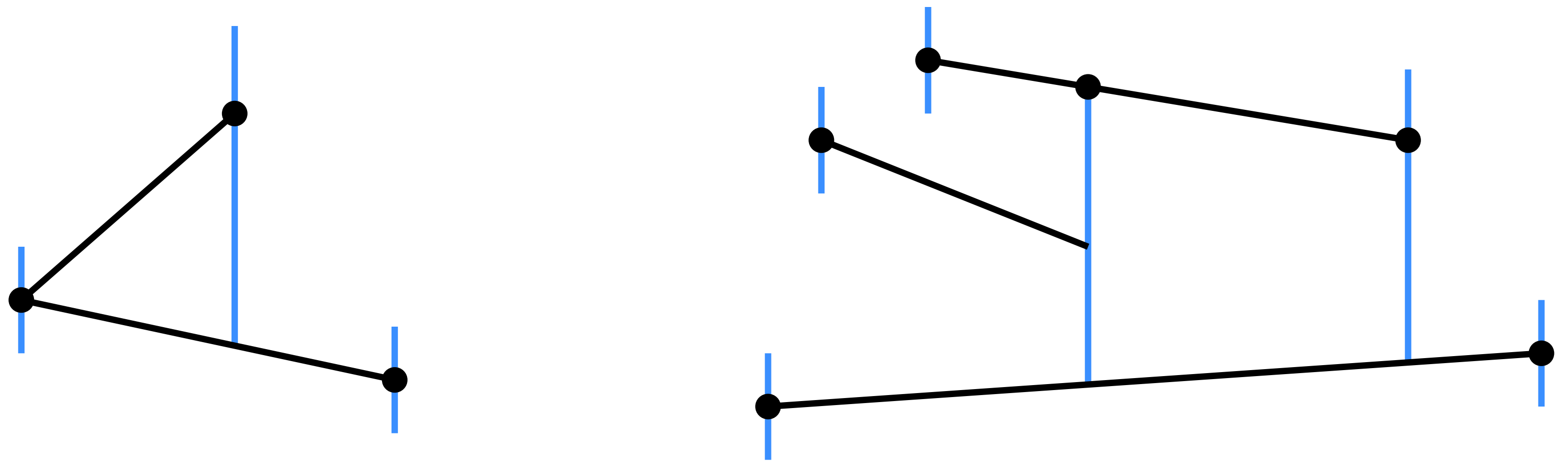
Suppose we draw vertical extensions from every vertex up and down, *but only until the next line segment*

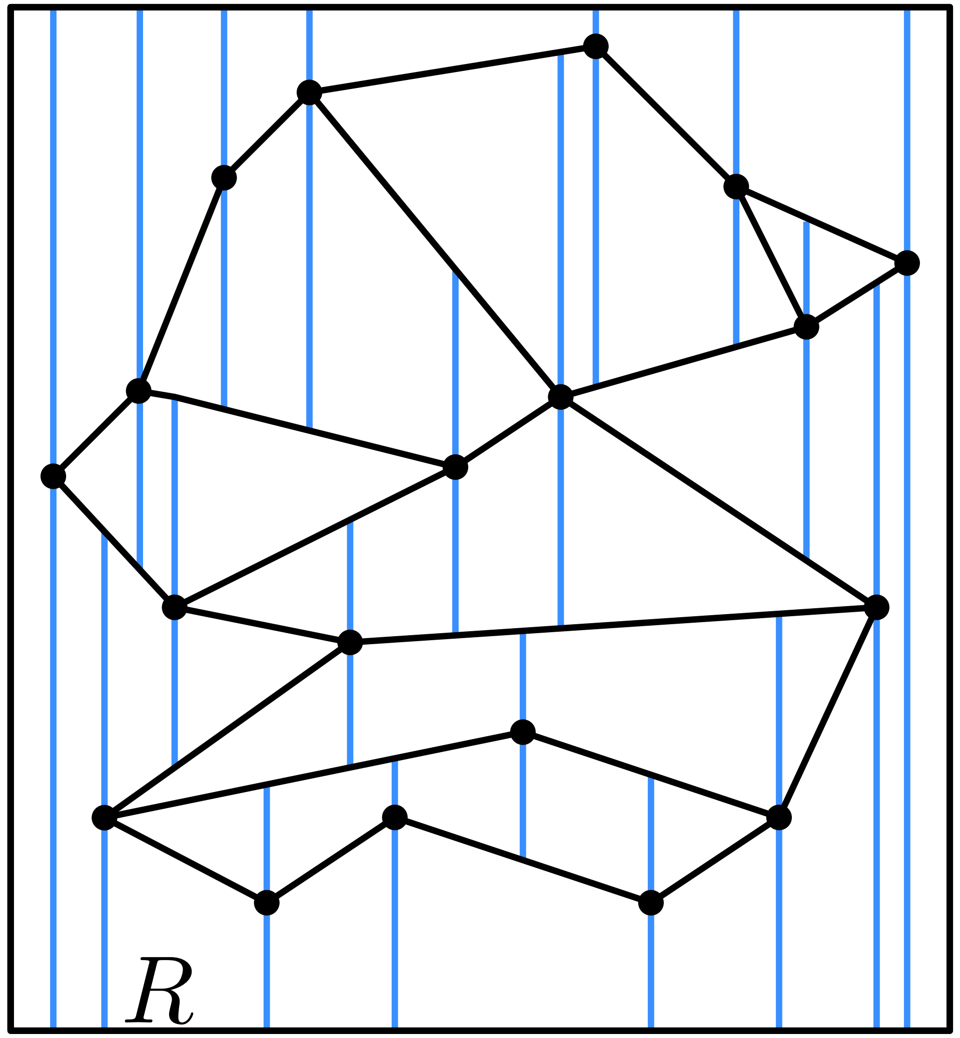
This is called the vertical decomposition or trapezoidal decomposition

- Assume we have a bounding box R that encloses all line segments that define the subdivision
- Assume the input line segments are not vertical
- Assume every vertex has a distinct x-coordinate



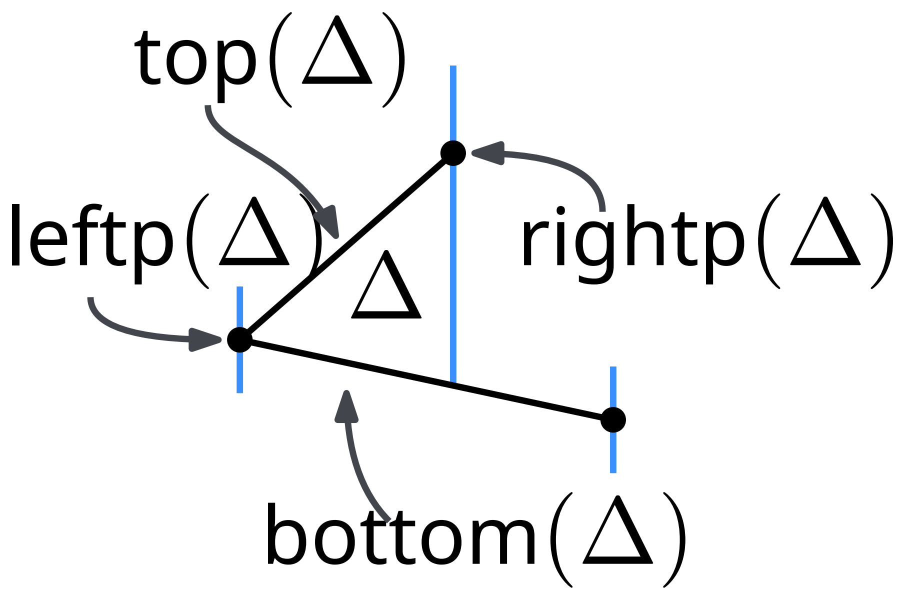
Vertical decomposition faces

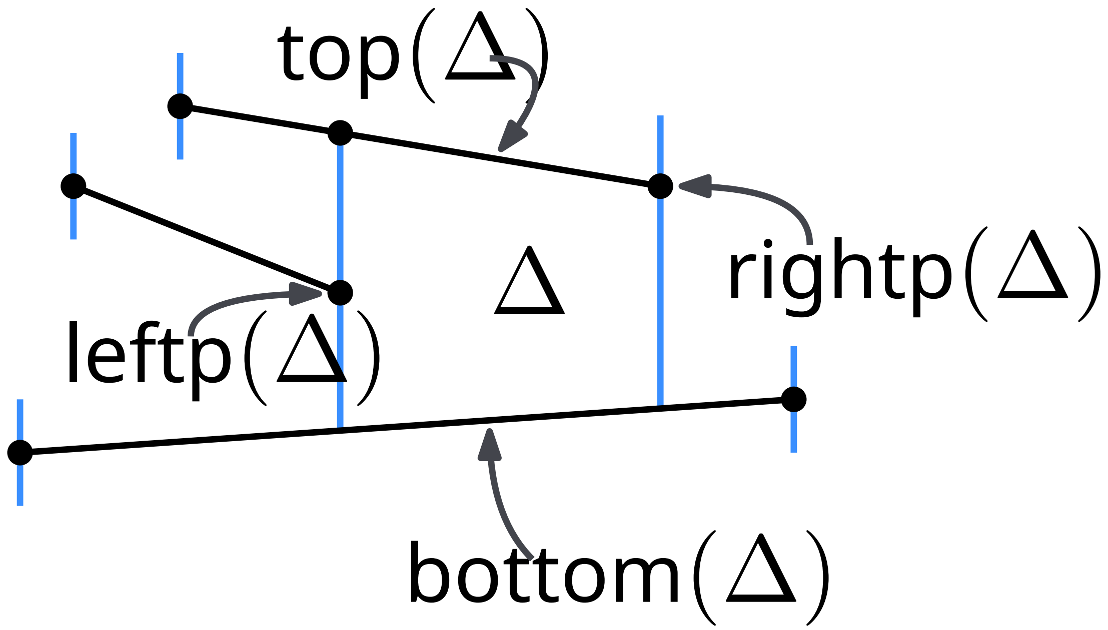


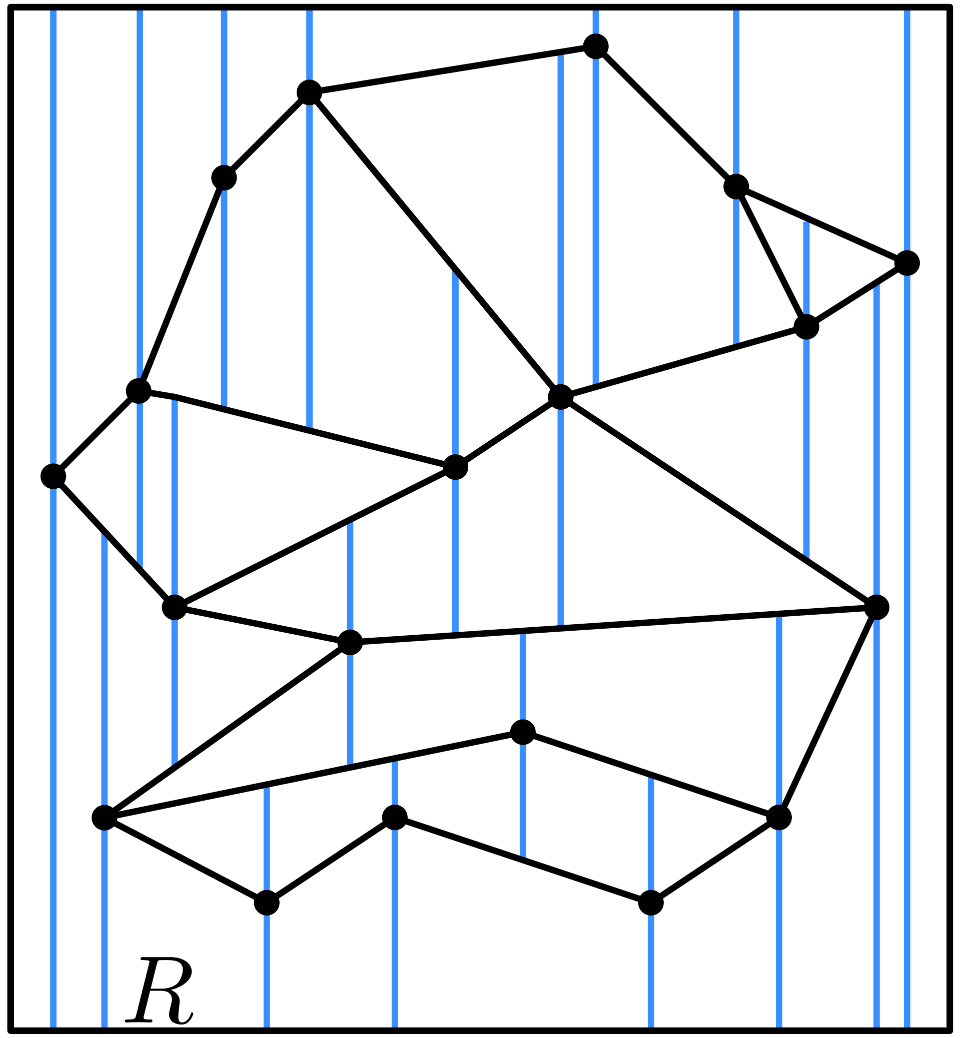


The vertical decomposition has triangular and trapezoidal faces

Vertical decomposition faces

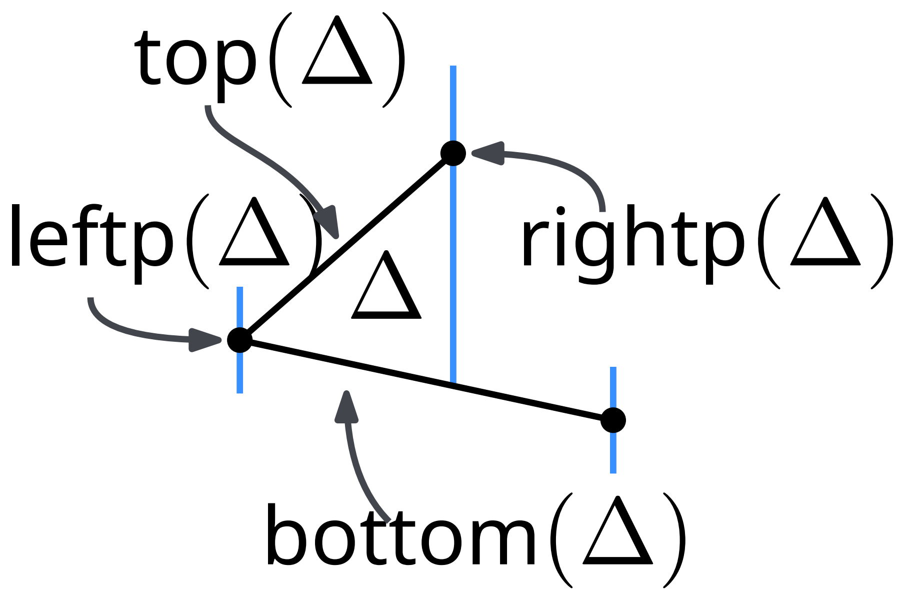


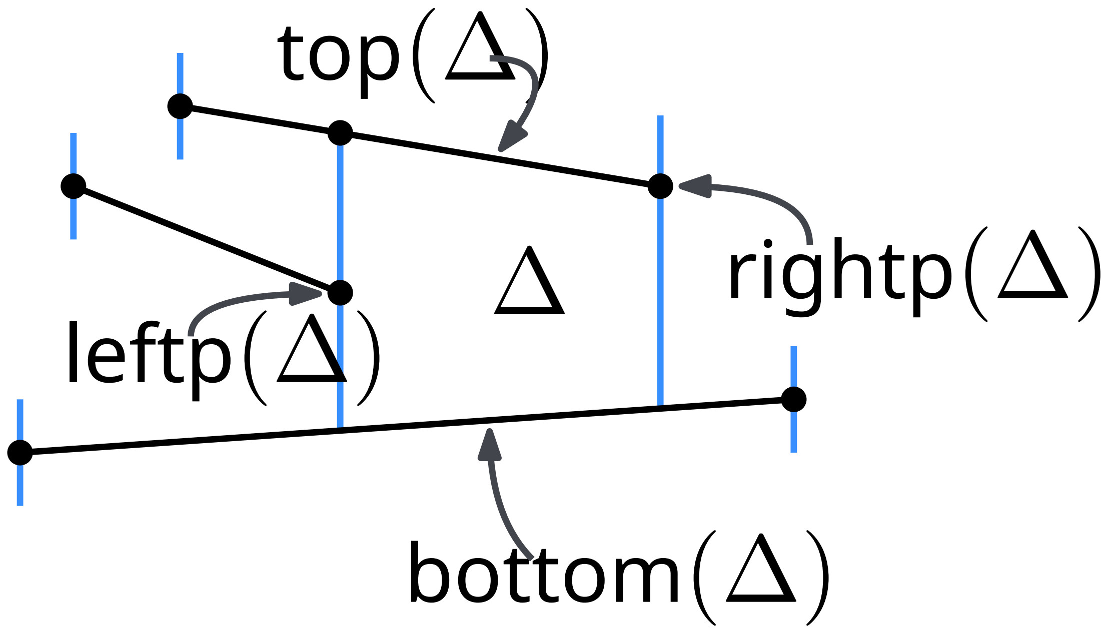


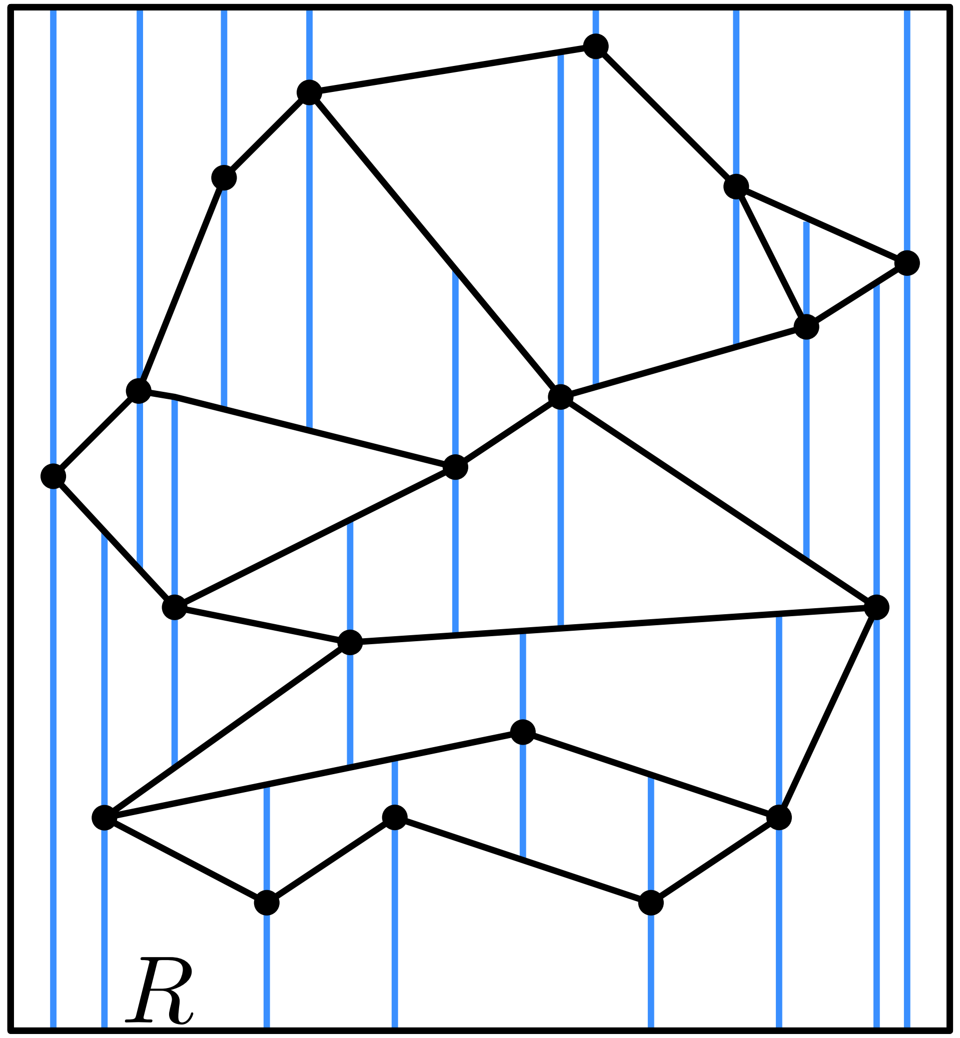


Every face has a vertical left and/or right side that is a vertical extension, and is bounded from above and below by some line segment of the input

Vertical decomposition faces





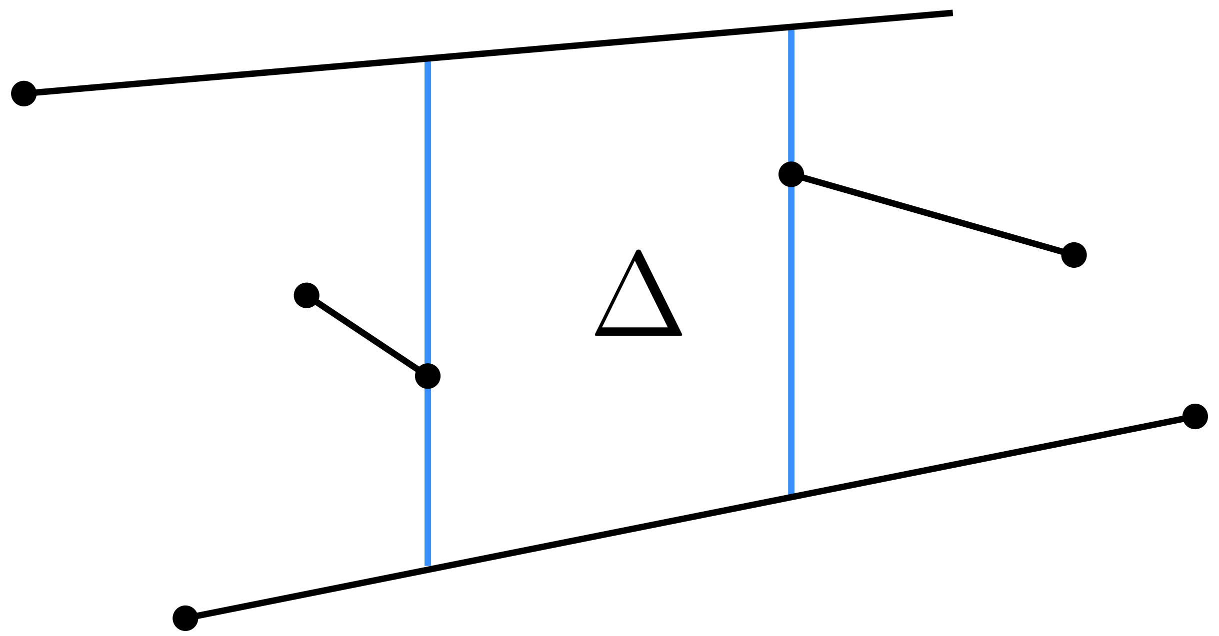


Every face has a vertical left and/or right side that is a vertical extension, and is bounded from above and below by some line segment of the input

The left and right sides are defined by some endpoint of a line segment

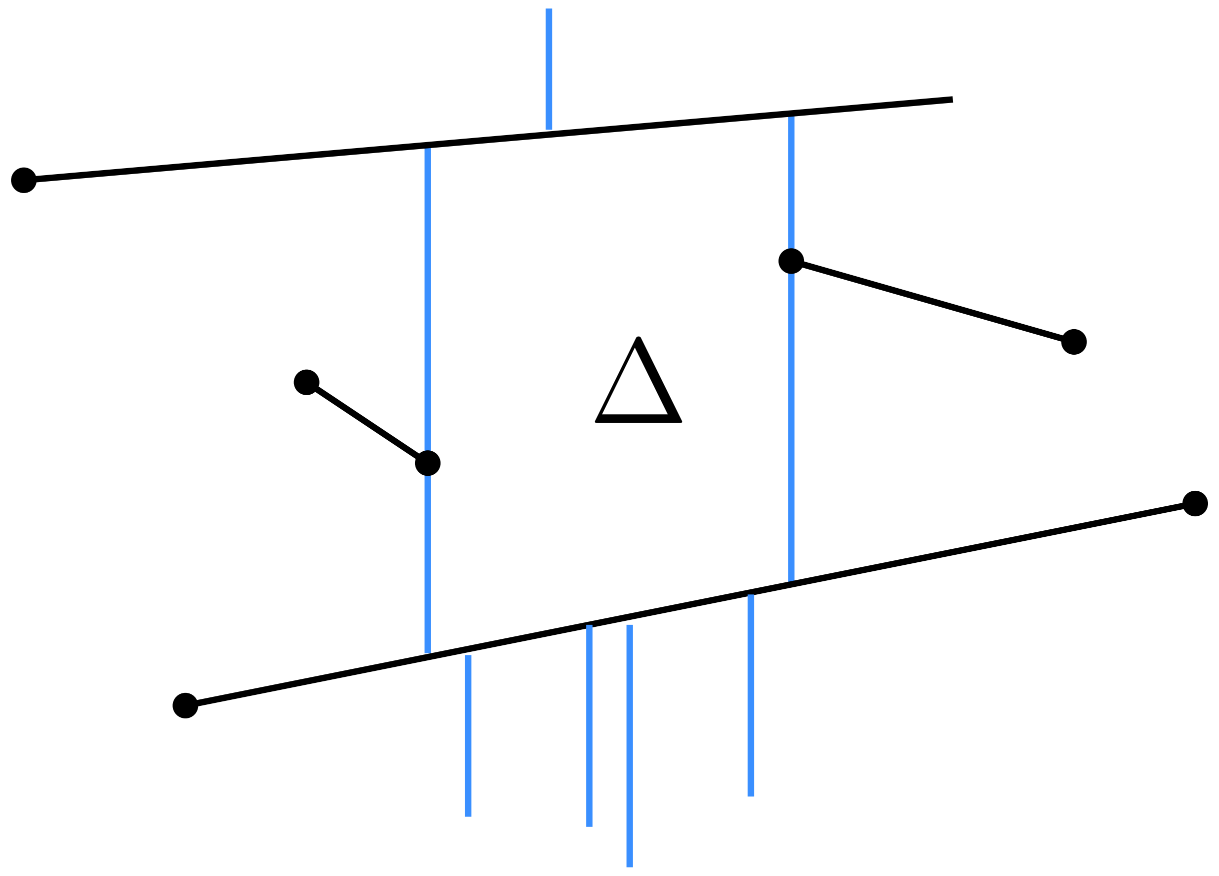
Vertical decomposition faces

Every face is defined by no more than four line segments



Vertical decomposition faces

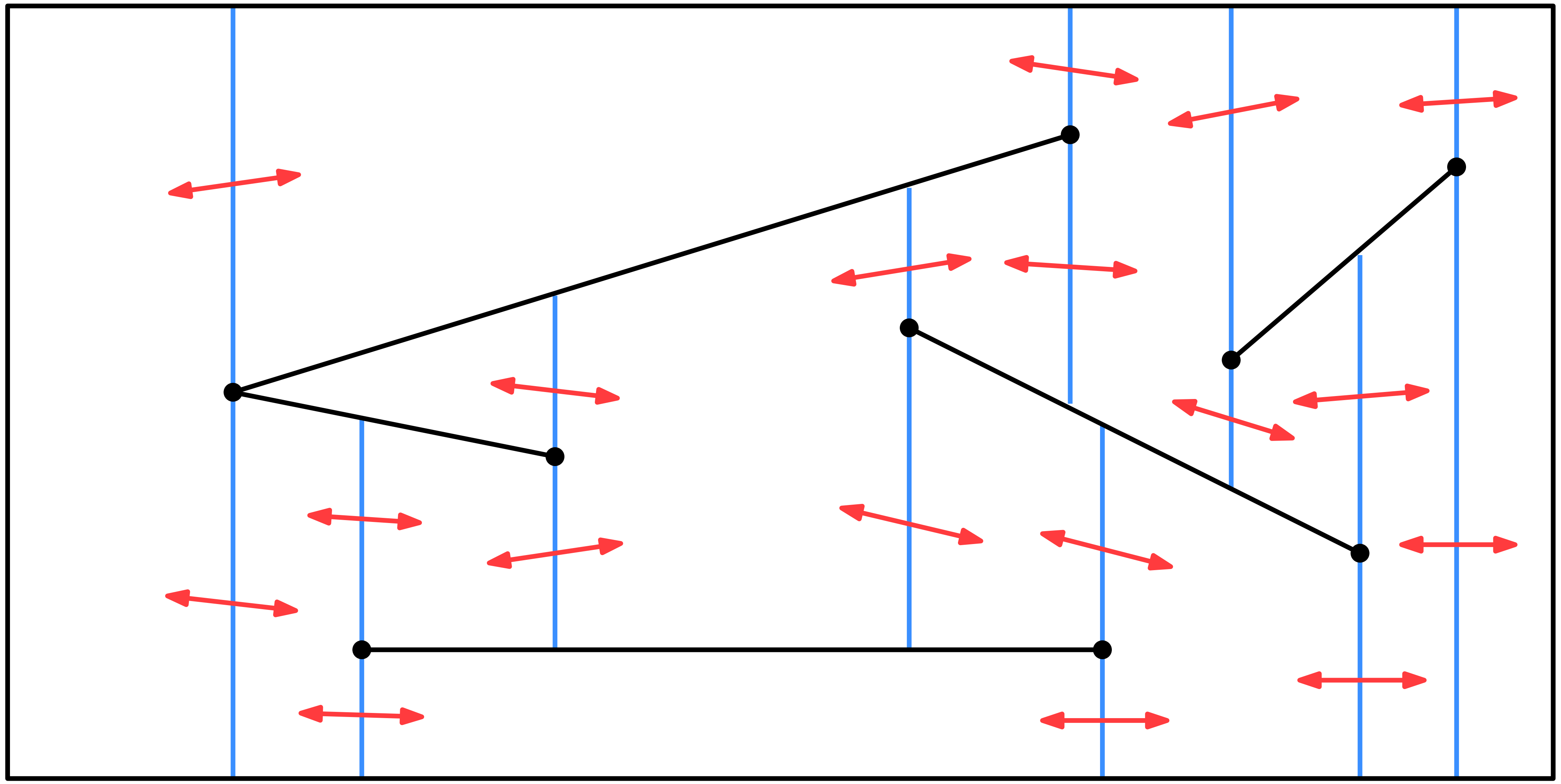
Every face is defined by no more than four line segments



For any face, we ignore vertical extensions that end on top(Δ) and bottom(Δ)

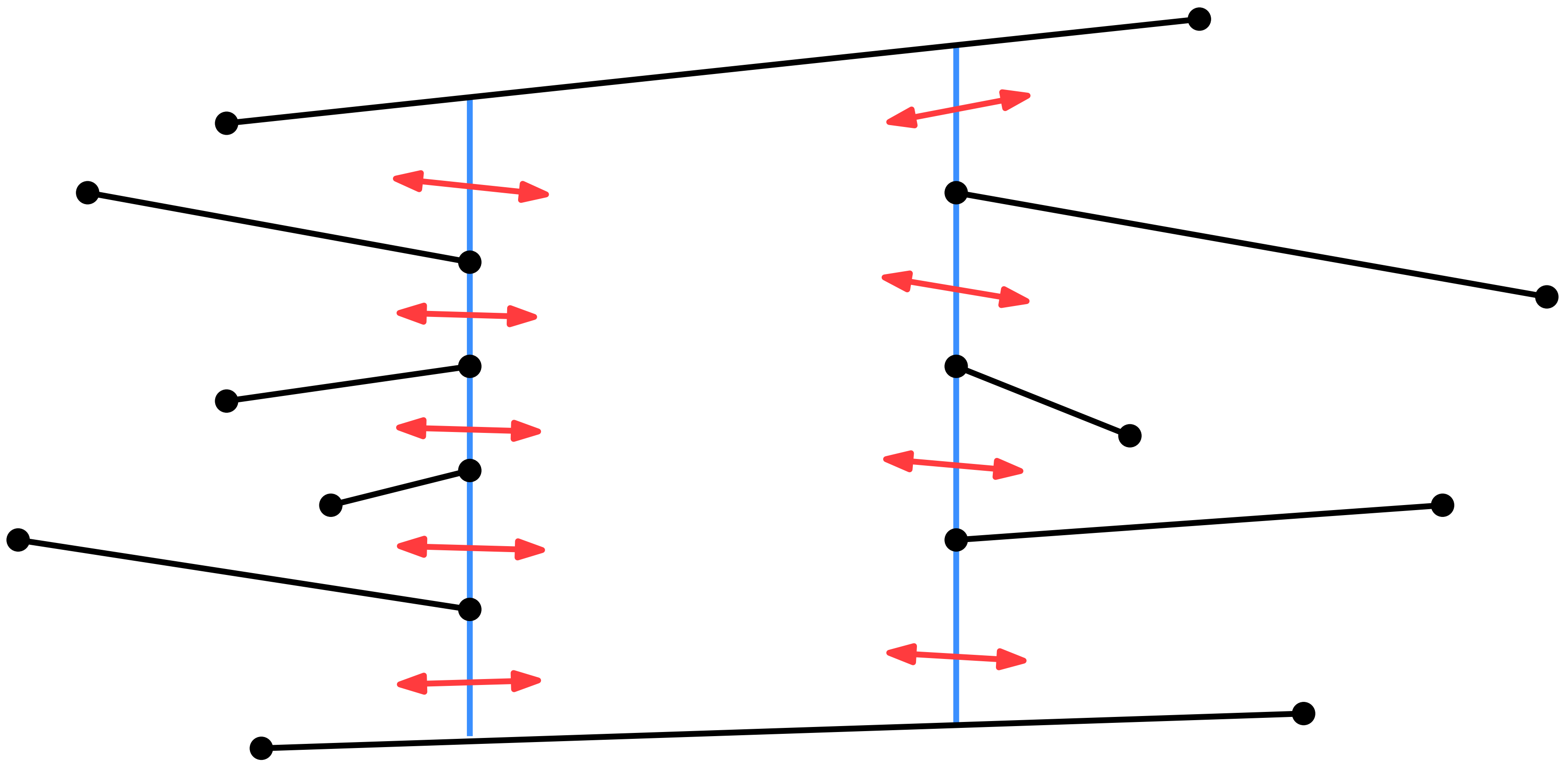
Neighbors of faces

Two trapezoids (including triangles) are neighbors if they share a vertical side

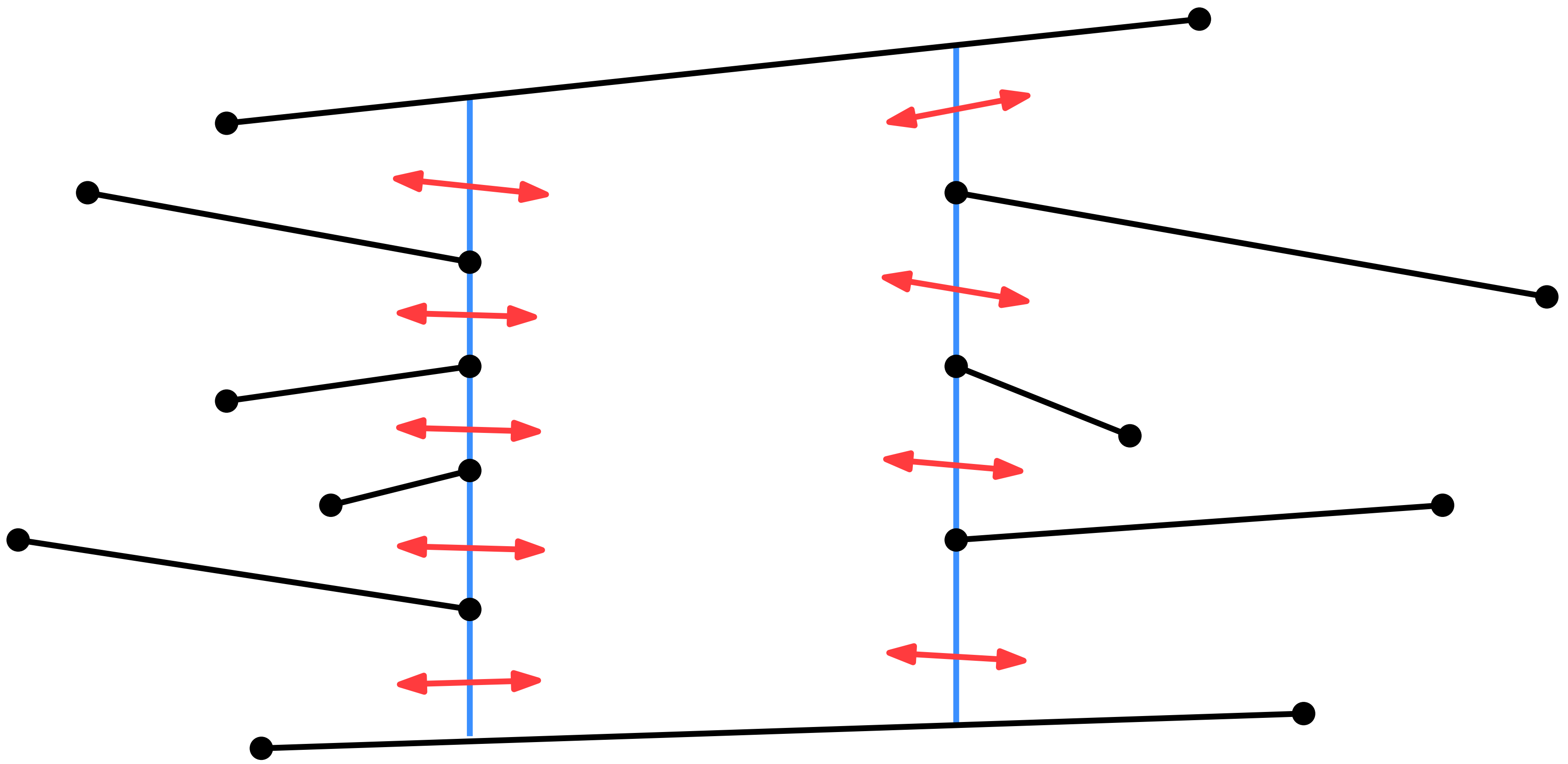


Each trapezoid has 1, 2, 3, or 4 neighbors

Neighbors of faces

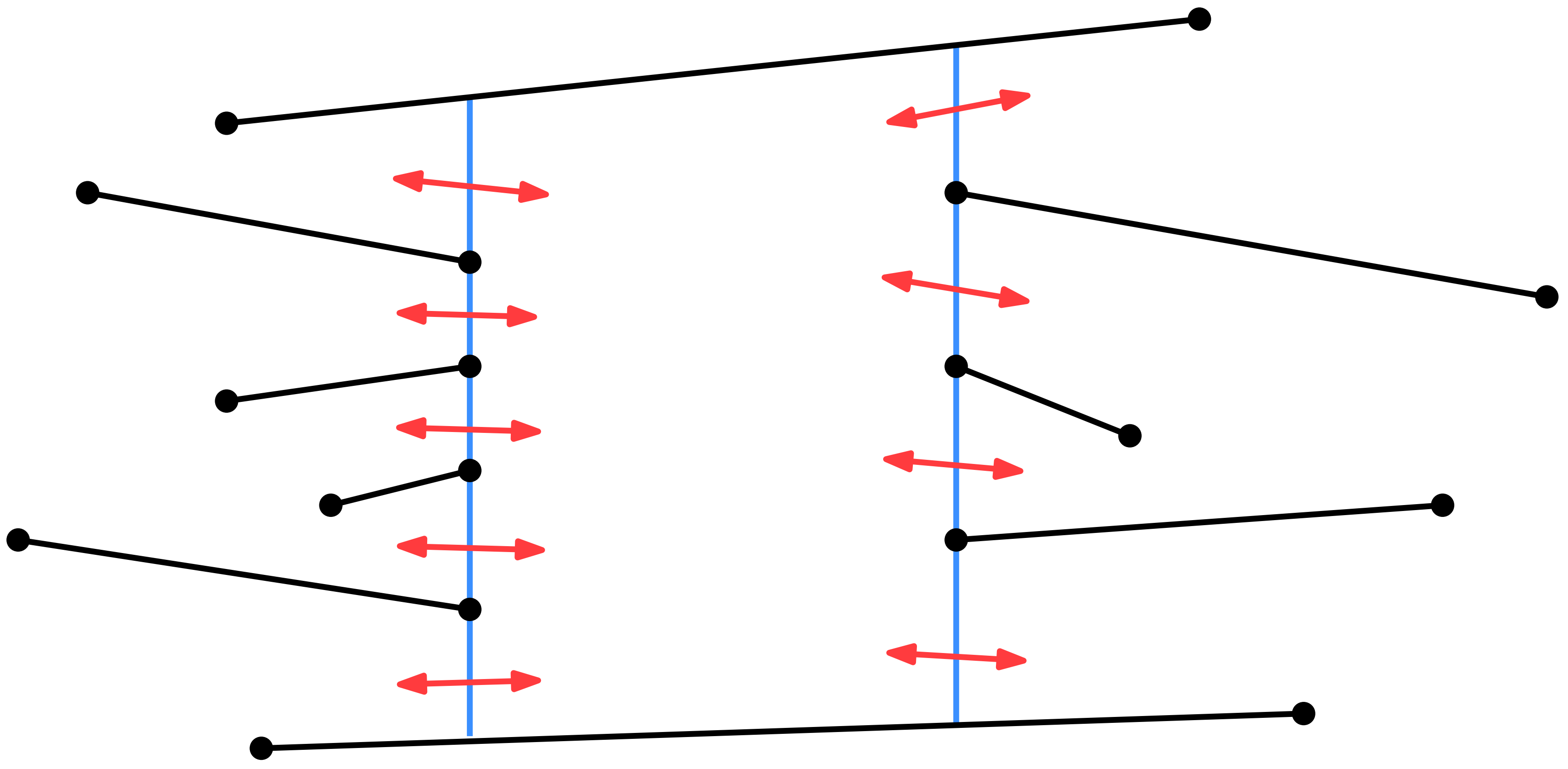


Neighbors of faces



A trapezoid could have many neighbors if vertices had the same x-coordinate

Neighbors of faces



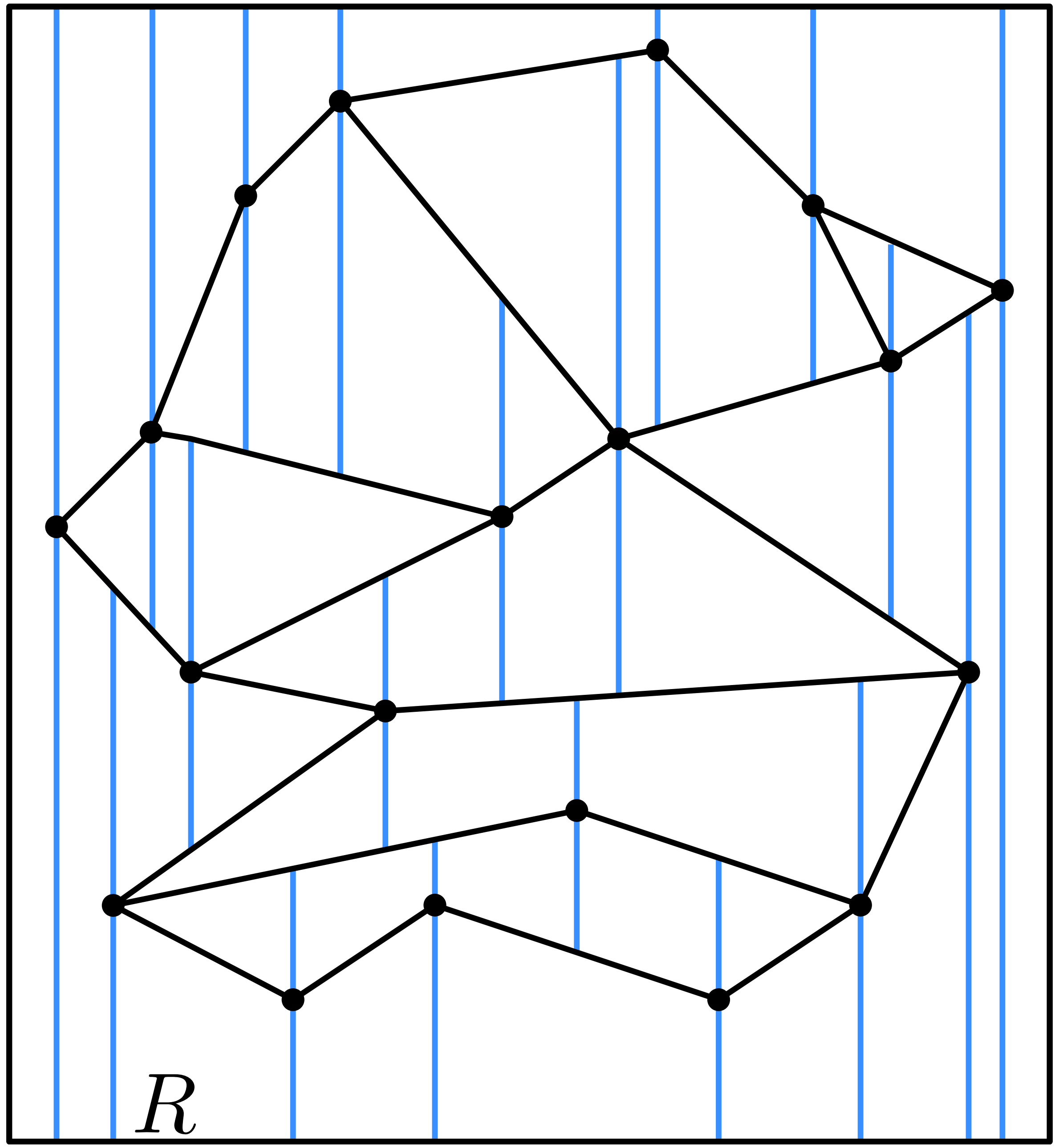
A trapezoid could have many neighbors if vertices had the same x-coordinate

next: data structure and complexity

Vertical Decomposition for Point Location

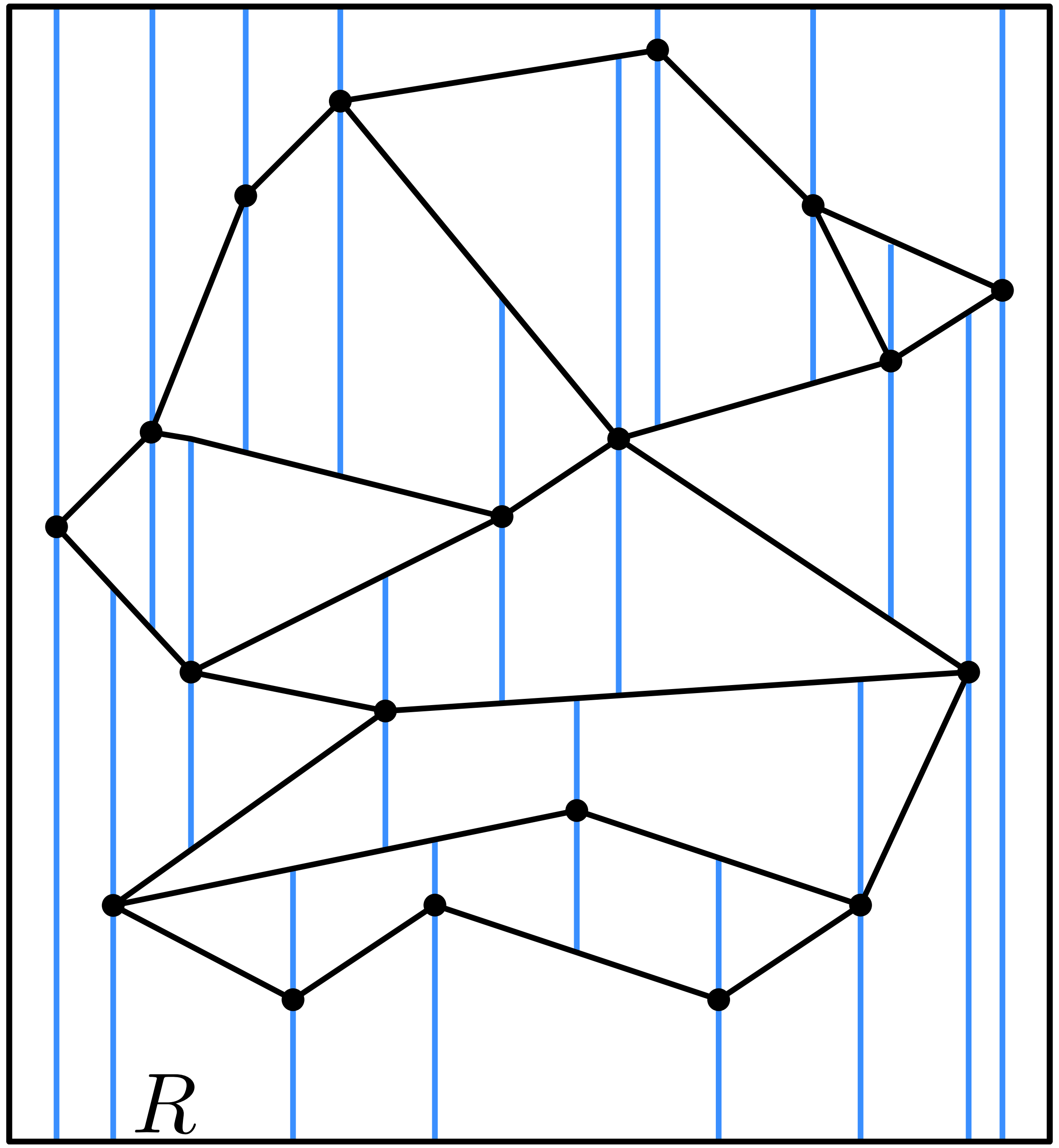
Vertical decomposition: data structure and complexity

Representation



Representation

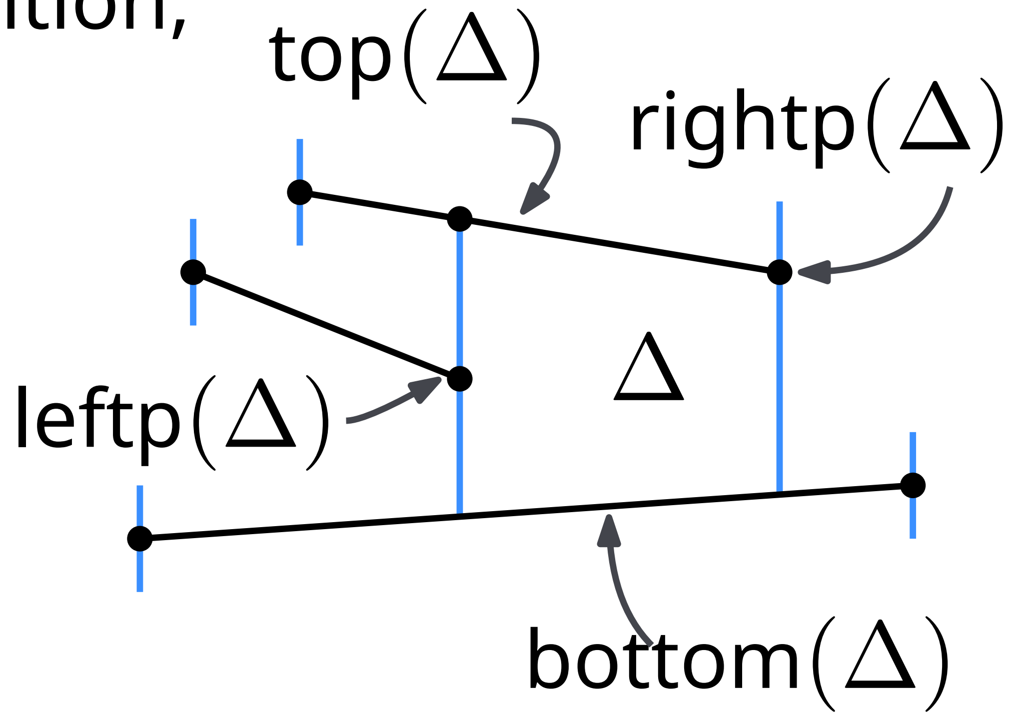
We could use a DCEL to represent a vertical decomposition, but we use a more direct & convenient structure



Representation

We could use a DCEL to represent a vertical decomposition, but we use a more direct & convenient structure

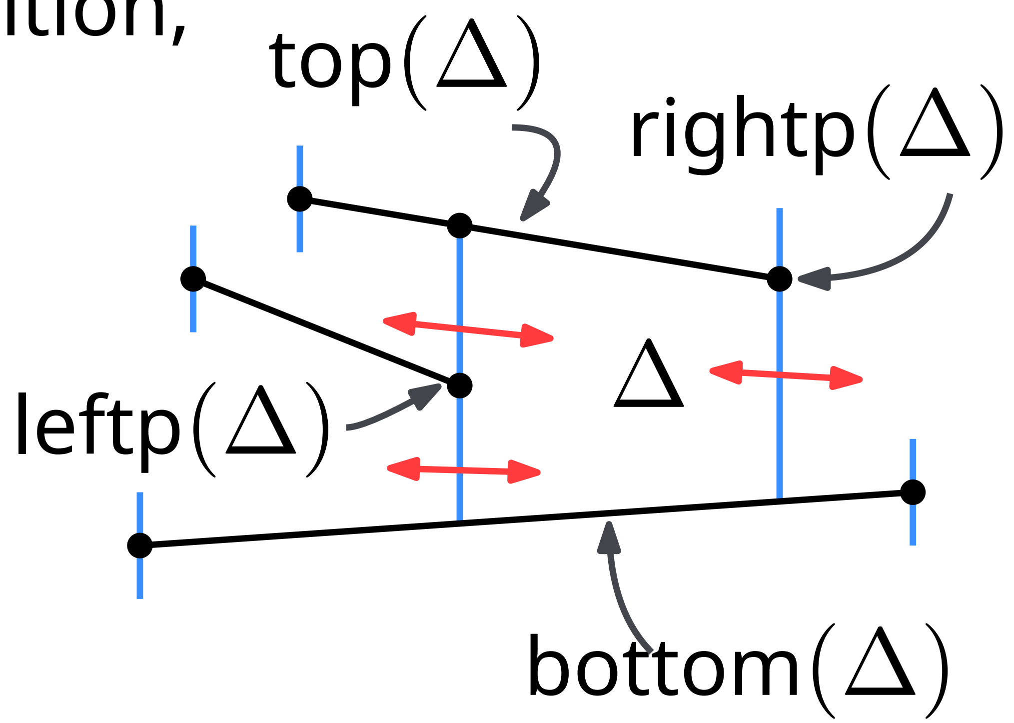
- Every face Δ is an object; it has fields for top(Δ), bottom(Δ), leftp(Δ), and rightp(Δ) (two line segments and two vertices)



Representation

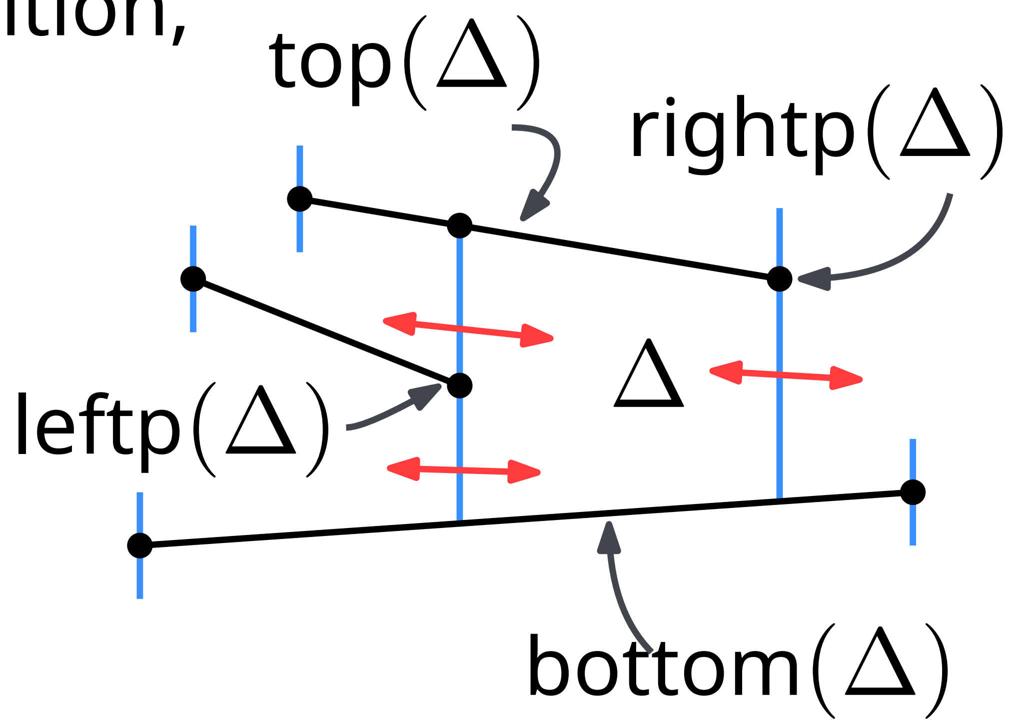
We could use a DCEL to represent a vertical decomposition, but we use a more direct & convenient structure

- Every face Δ is an object; it has fields for $\text{top}(\Delta)$, $\text{bottom}(\Delta)$, $\text{leftp}(\Delta)$, and $\text{rightp}(\Delta)$ (two line segments and two vertices)
- Every face has fields to access its up to four neighbors



Representation

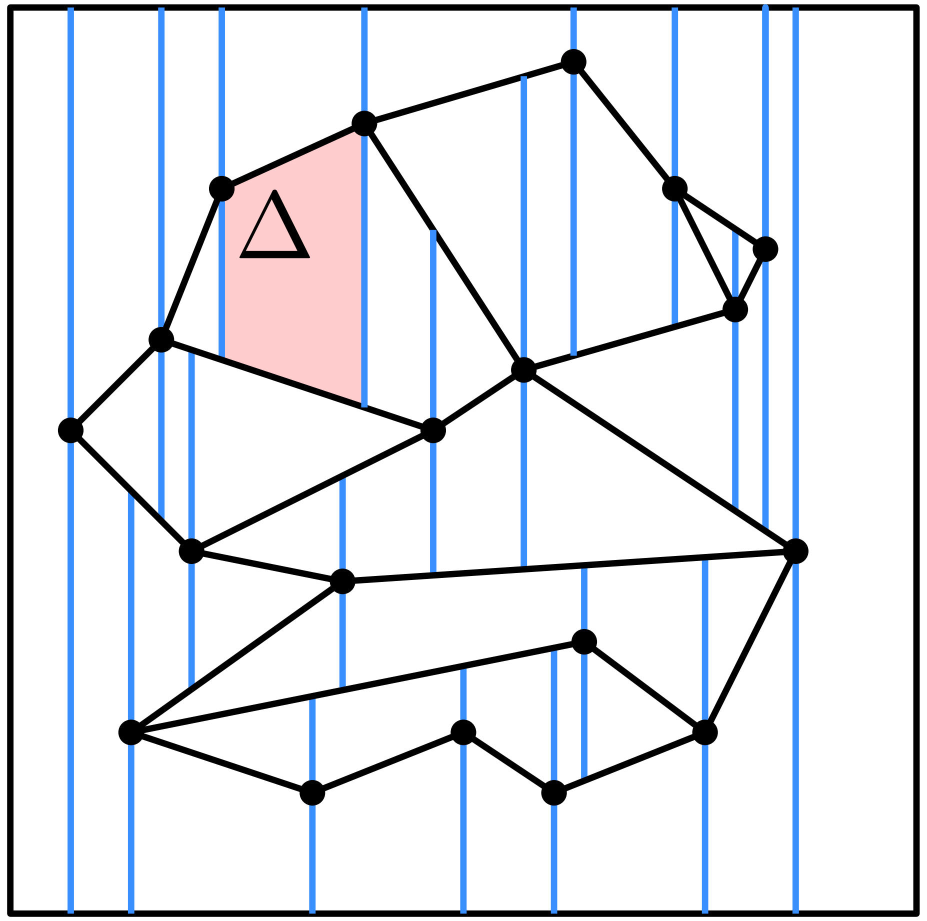
We could use a DCEL to represent a vertical decomposition, but we use a more direct & convenient structure



- Every face Δ is an object; it has fields for top(Δ), bottom(Δ), leftp(Δ), and rightp(Δ) (two line segments and two vertices)
- Every face has fields to access its up to four neighbors
- Every line segment is an object and has fields for its endpoints (vertices) and the name of the face in the original subdivision directly above it
- Each vertex stores its coordinates

Quiz

Given a trapezoid Δ, how do we find the face of the DCEL containing it?



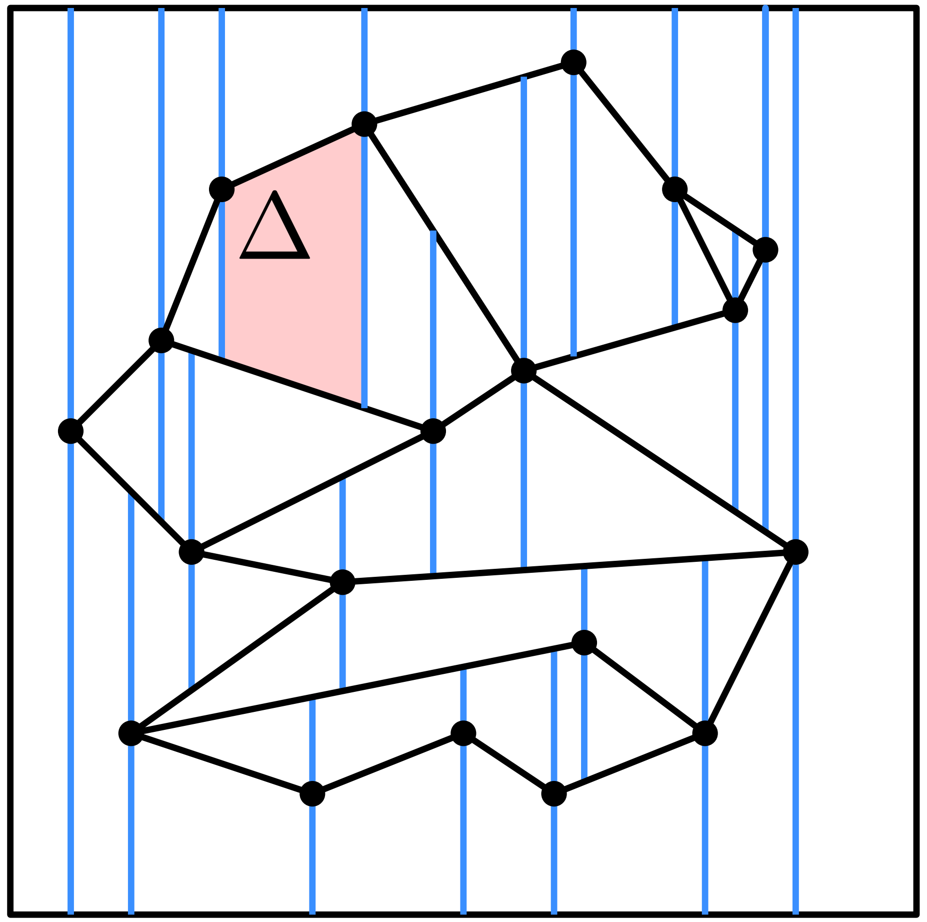
A: We store the face as attribute of Δ

B: We store this information in a hash table

C: We access bottom(Δ), which stores the face

Quiz

Given a trapezoid Δ, how do we find the face of the DCEL containing it?



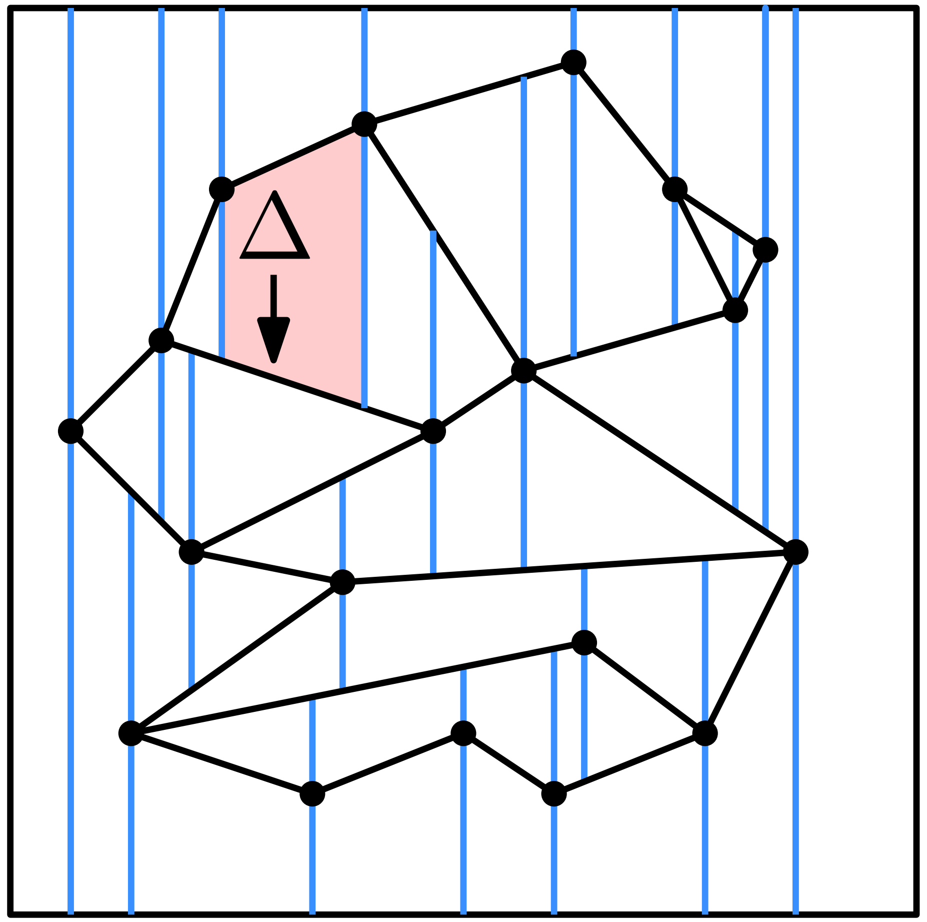
A: We store the face as attribute of Δ

B: We store this information in a hash table

C: We access bottom(Δ), which stores the face

Quiz

Given a trapezoid Δ, how do we find the face of the DCEL containing it?



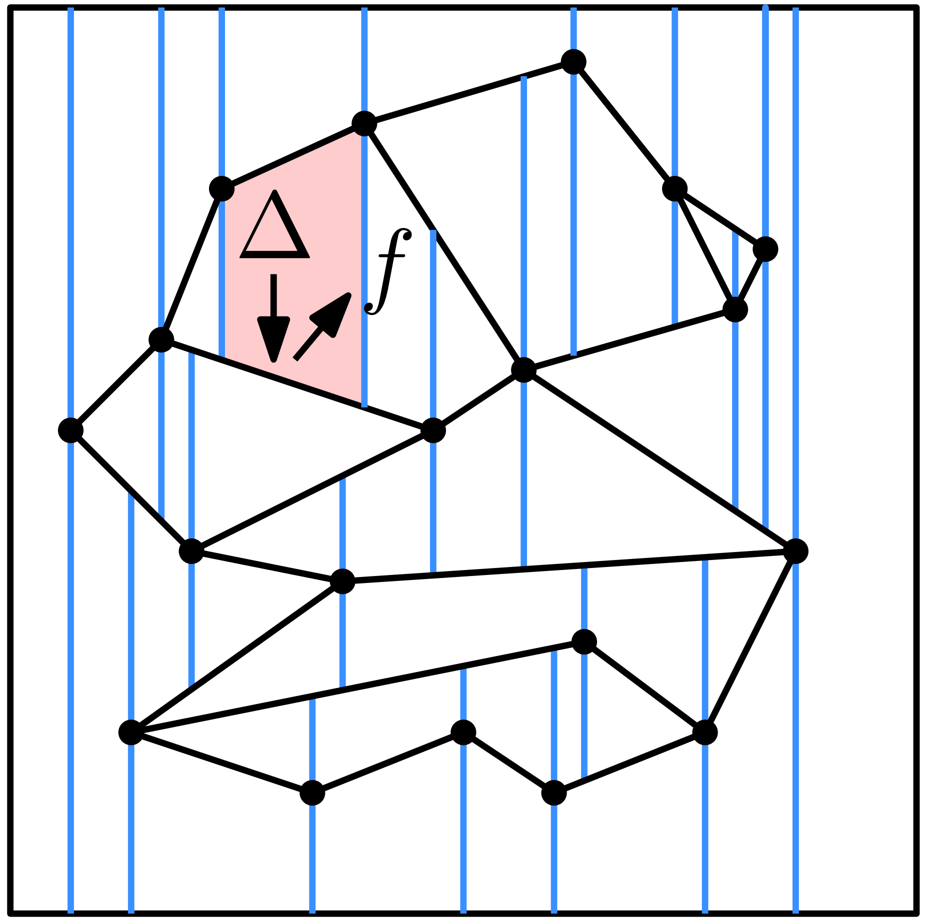
A: We store the face as attribute of Δ

B: We store this information in a hash table

C: We access bottom(Δ), which stores the face

Quiz

Given a trapezoid Δ, how do we find the face of the DCEL containing it?



A: We store the face as attribute of Δ

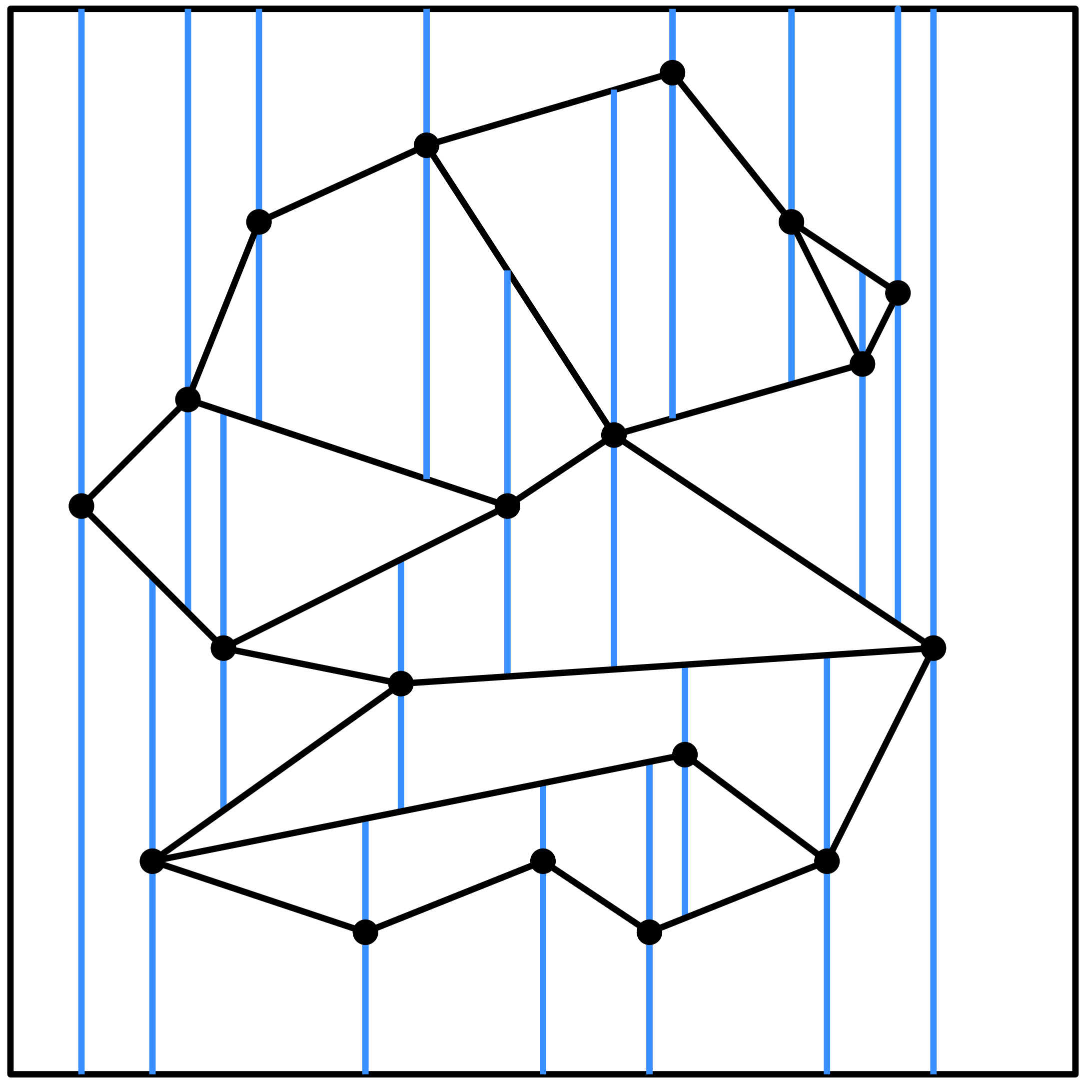
B: We store this information in a hash table

C: We access bottom(Δ), which stores the face

Complexity

A vertical decomposition of n non-crossing line segments inside a bounding box R, seen as a proper planar subdivision, has

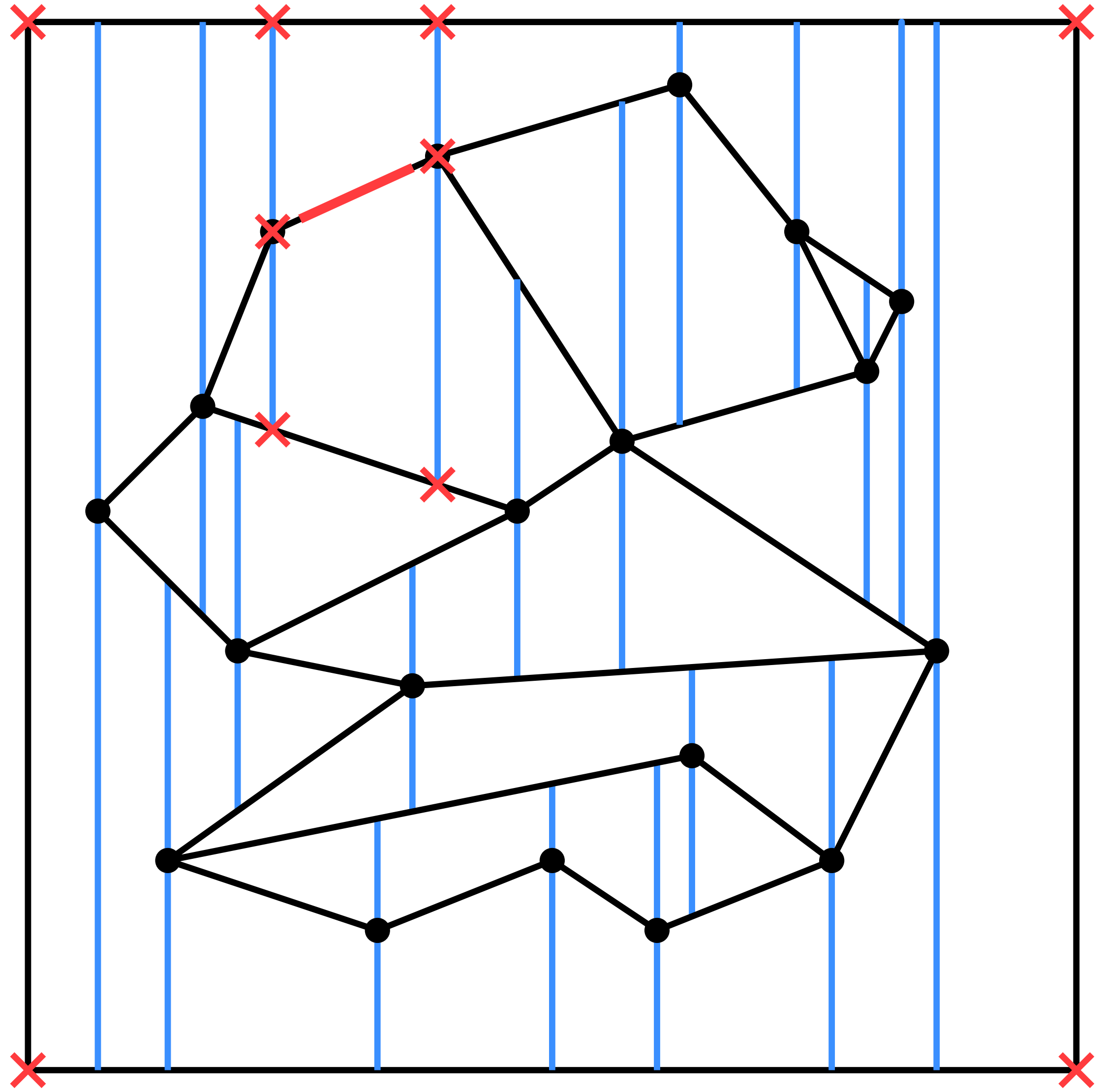
- at most vertices and
- at most trapezoids



Complexity

A vertical decomposition of n non-crossing line segments inside a bounding box R, seen as a proper planar subdivision, has

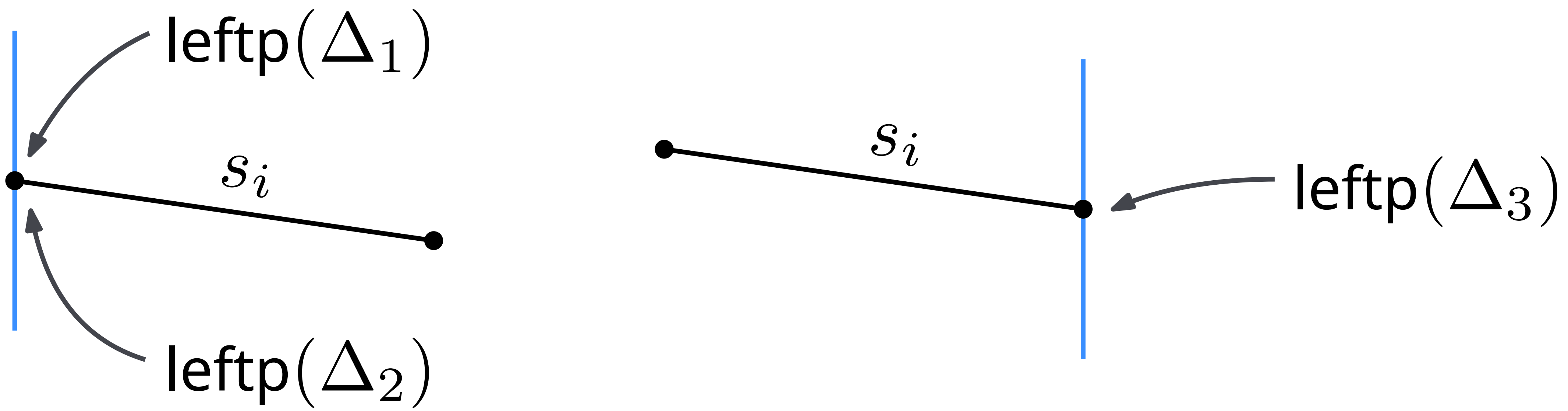
- at most $6n + 4$ vertices and
- at most trapezoids



Complexity

A vertical decomposition of n non-crossing line segments inside a bounding box R, seen as a proper planar subdivision, has

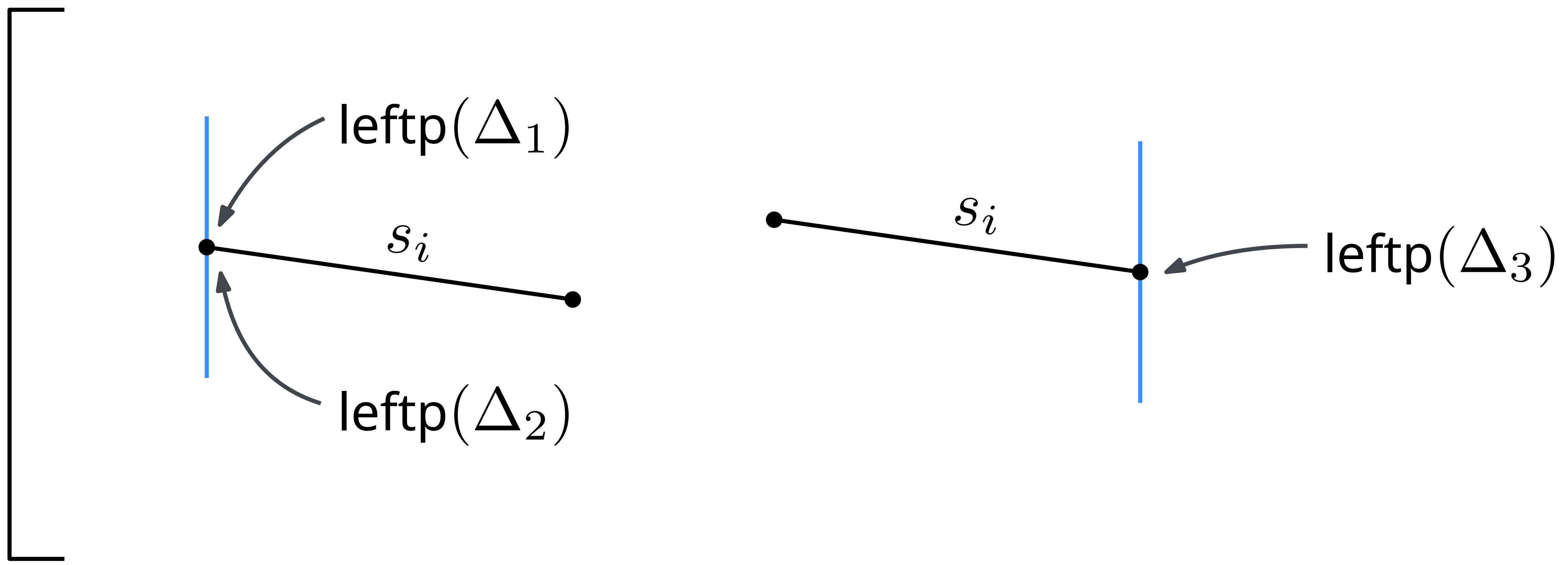
- at most $6n + 4$ vertices and
- at most trapezoids



Complexity

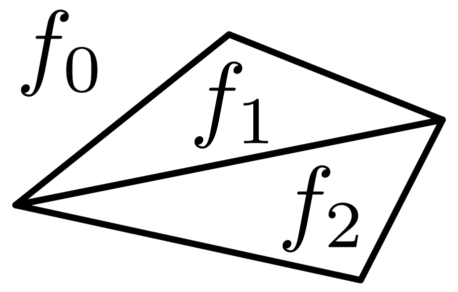
A vertical decomposition of n non-crossing line segments inside a bounding box R, seen as a proper planar subdivision, has

- at most $6n + 4$ vertices and
- at most $3n + 1$ trapezoids



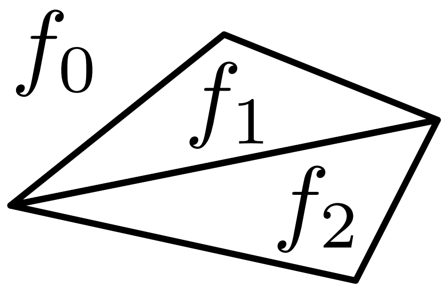
Point location preprocessing

The input to point location is a planar subdivision, for example in DCEL format

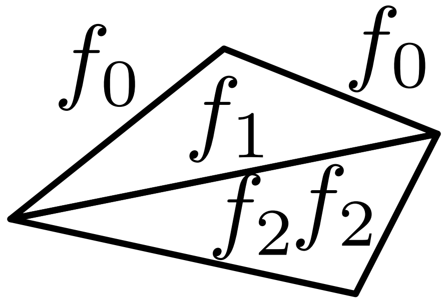


Point location preprocessing

The input to point location is a planar subdivision, for example in DCEL format

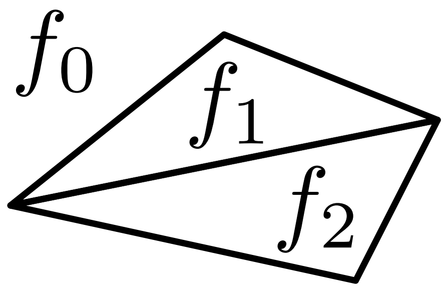


First, store with each edge the name of the face above it

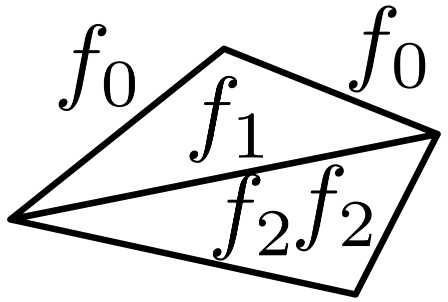


Point location preprocessing

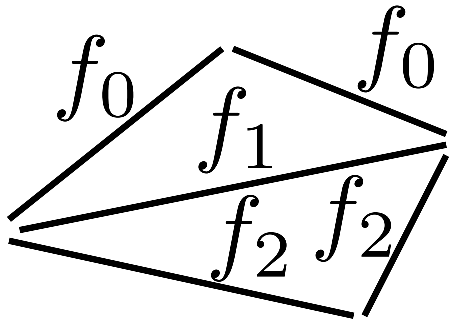
The input to point location is a planar subdivision, for example in DCEL format



First, store with each edge the name of the face above it

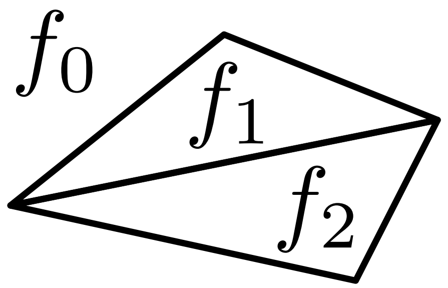


Then, extract the edges to define a set S of non-crossing line segments

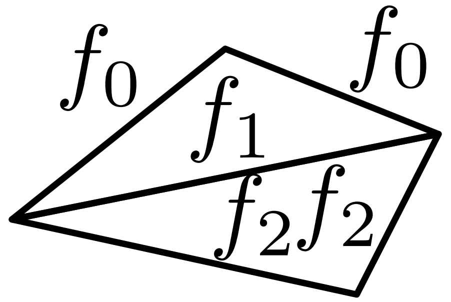


Point location preprocessing

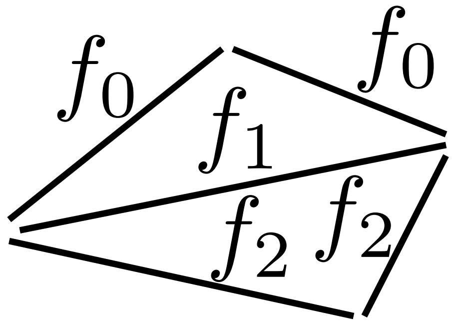
The input to point location is a planar subdivision, for example in DCEL format



First, store with each edge the name of the face above it

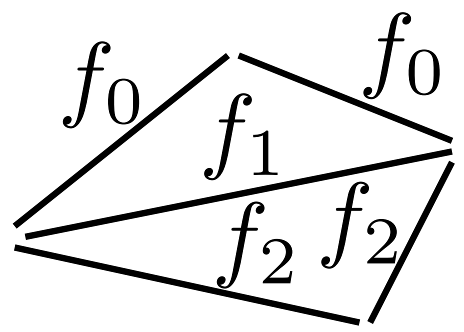


Then, extract the edges to define a set S of non-crossing line segments



. . . ignore the DCEL otherwise

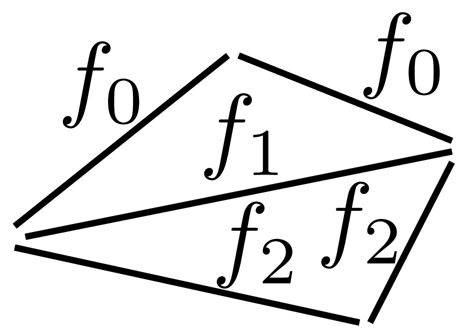
Point location preprocessing



. . . ignore the DCEL otherwise

Point location preprocessing

next: finally, building the point location data structure



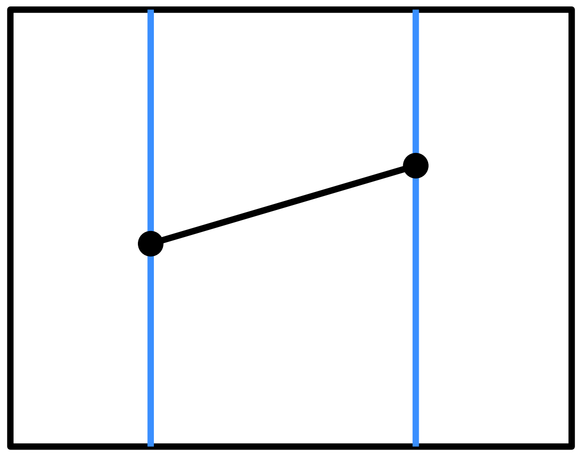
. . . ignore the DCEL otherwise

Vertical Decomposition for Point Location

Randomized Incremental Construction

Point location solution

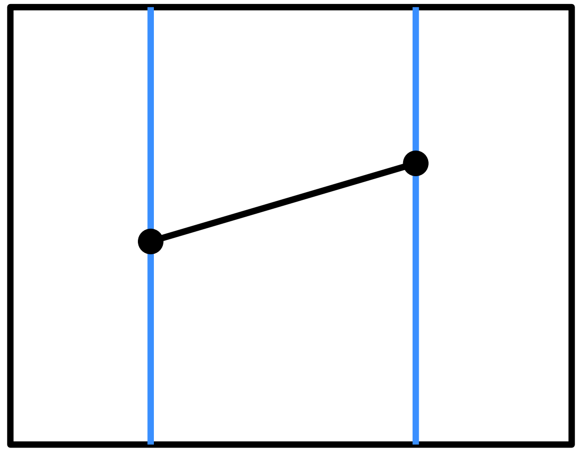
We will use randomized incremental construction to build, for a set S of non-crossing line segments,



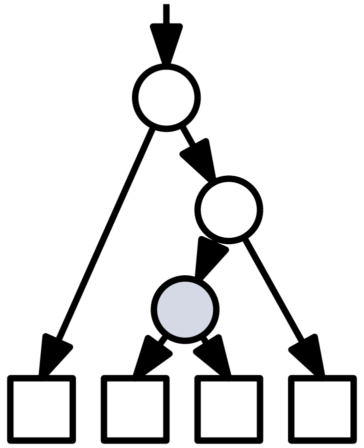
a vertical decomposition T of S and R

Point location solution

We will use randomized incremental construction to build, for a set S of non-crossing line segments,



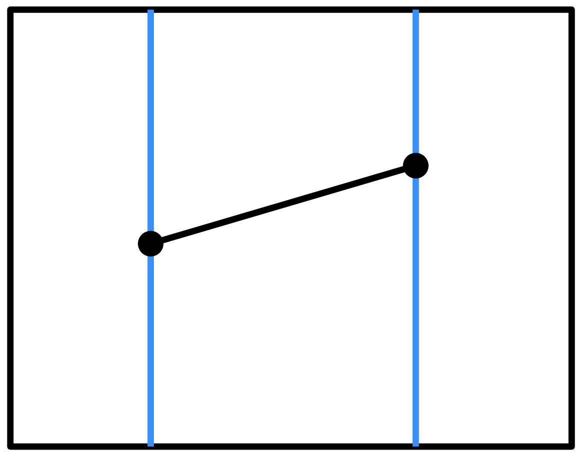
a vertical decomposition T of S and R



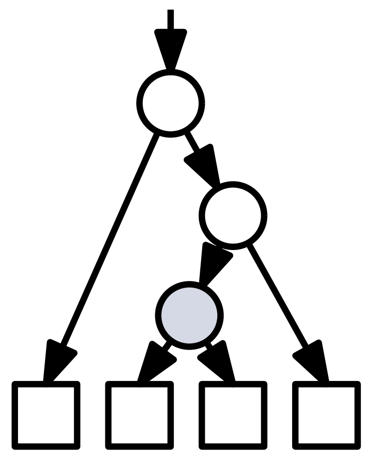
a search structure D whose leaves correspond to the trapezoids of T

Point location solution

We will use randomized incremental construction to build, for a set S of non-crossing line segments,



a vertical decomposition T of S and R

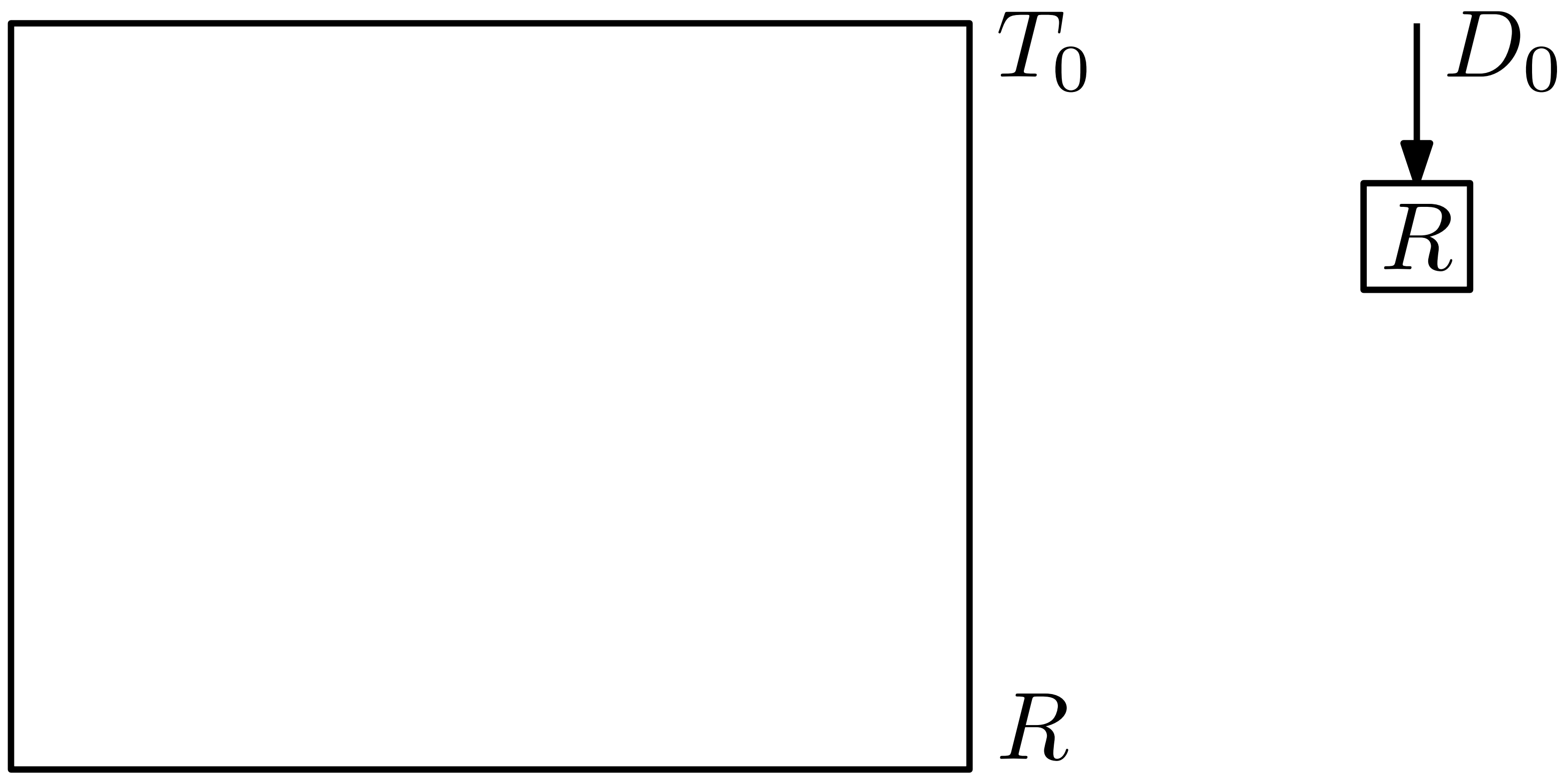


a search structure D whose leaves correspond to the trapezoids of T

The simple idea:

- Start with R, then
- add the line segments in random order and maintain T and D

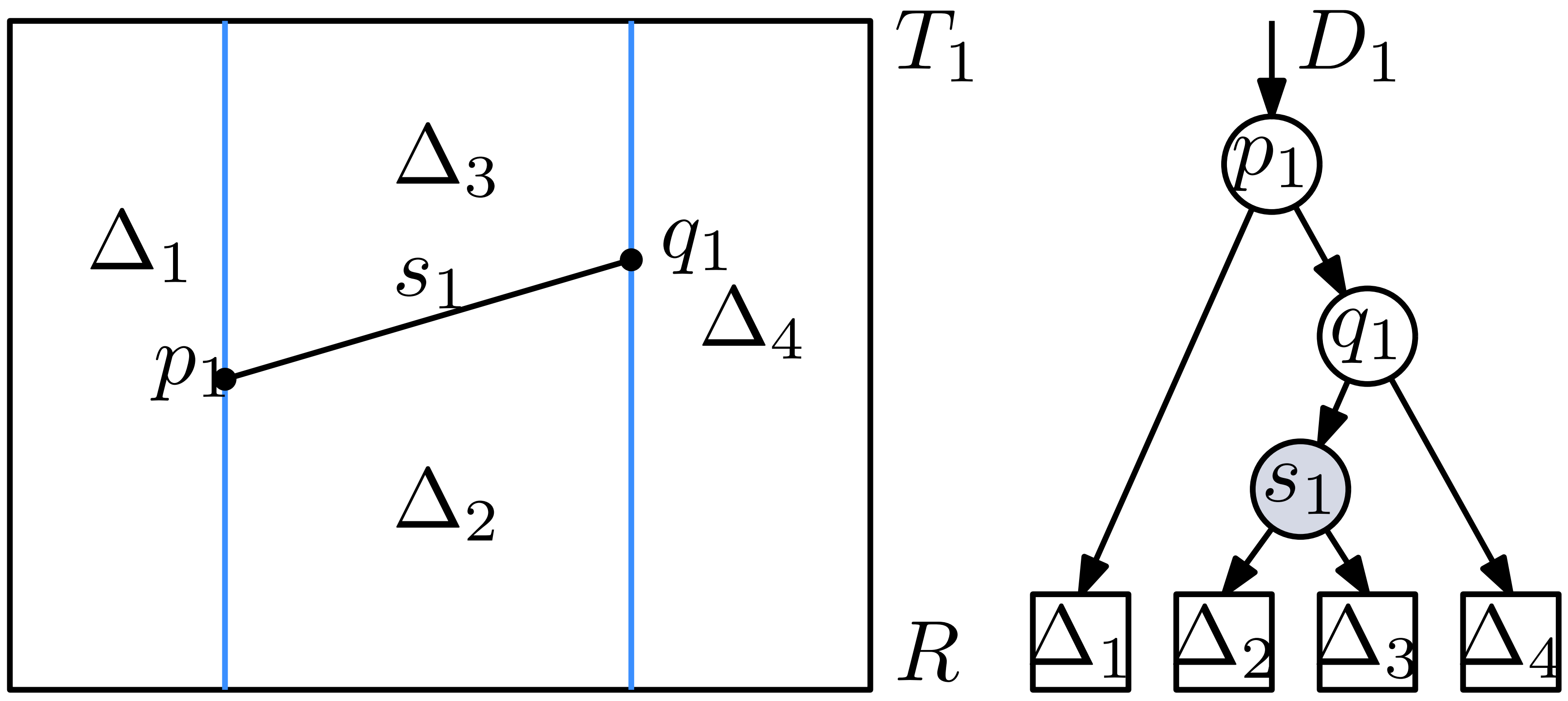
Point location solution



Let $s_1, \ldots, s_n$ be the n line segments in random order

Let T_i be the vertical decomposition of R and $s_1, \ldots, s_i$, and let D_i be the search structure obtained by inserting $s_1, \ldots, s_i$ in this order

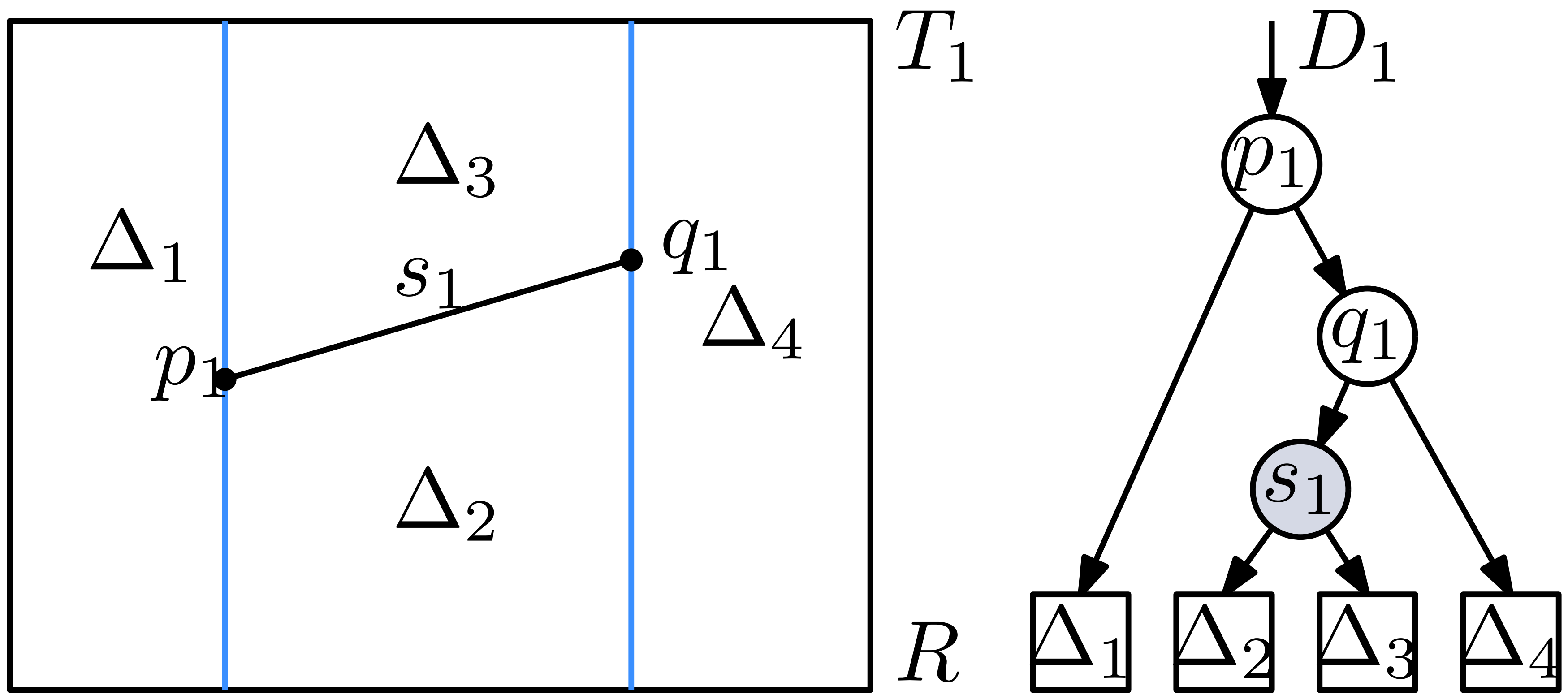
Point location solution



Let $s_1, \ldots, s_n$ be the n line segments in random order

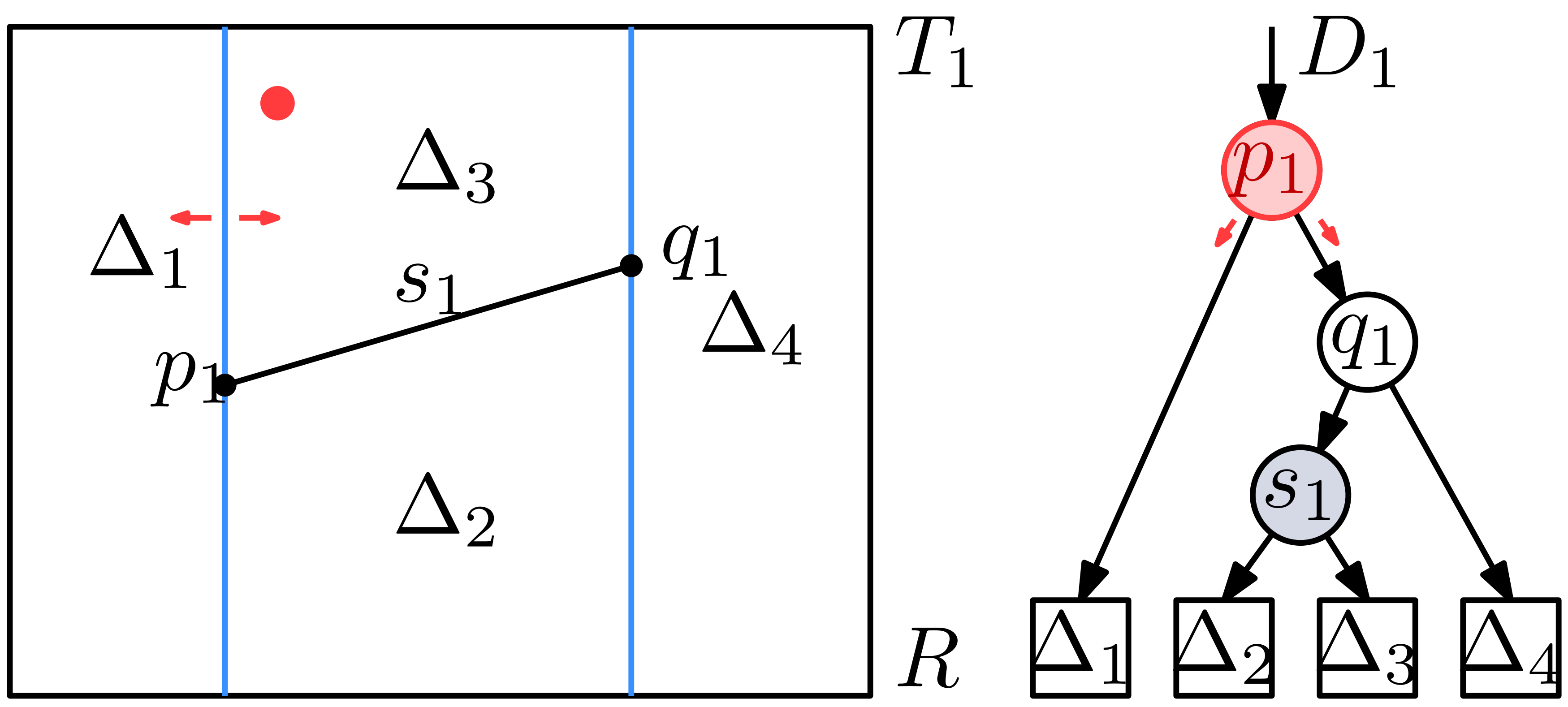
Let T_i be the vertical decomposition of R and $s_1, \ldots, s_i$, and let D_i be the search structure obtained by inserting $s_1, \ldots, s_i$ in this order

Point location solution



The search structure D has x-nodes, which store an endpoint, and y-nodes, which store a line segment s_j

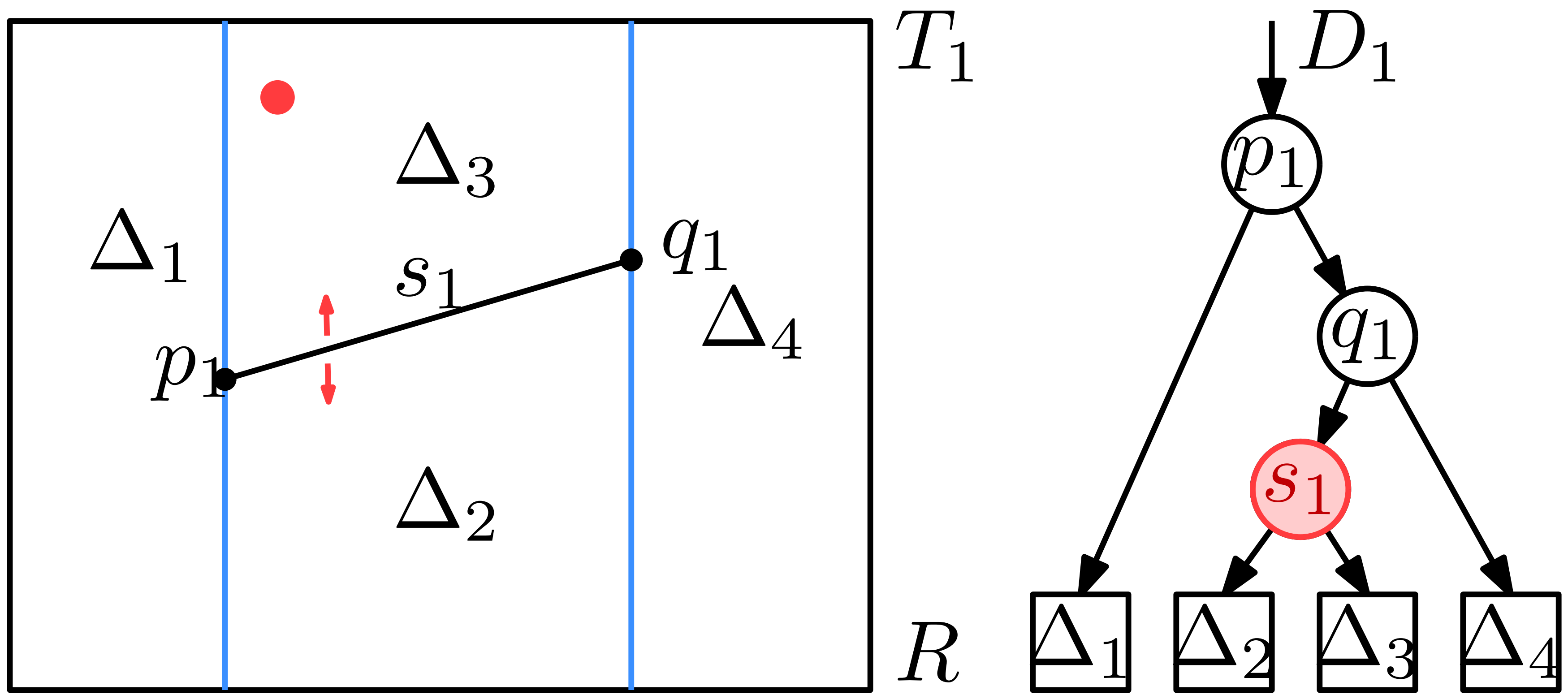
Point location solution



The search structure D has x-nodes, which store an endpoint, and y-nodes, which store a line segment s_j

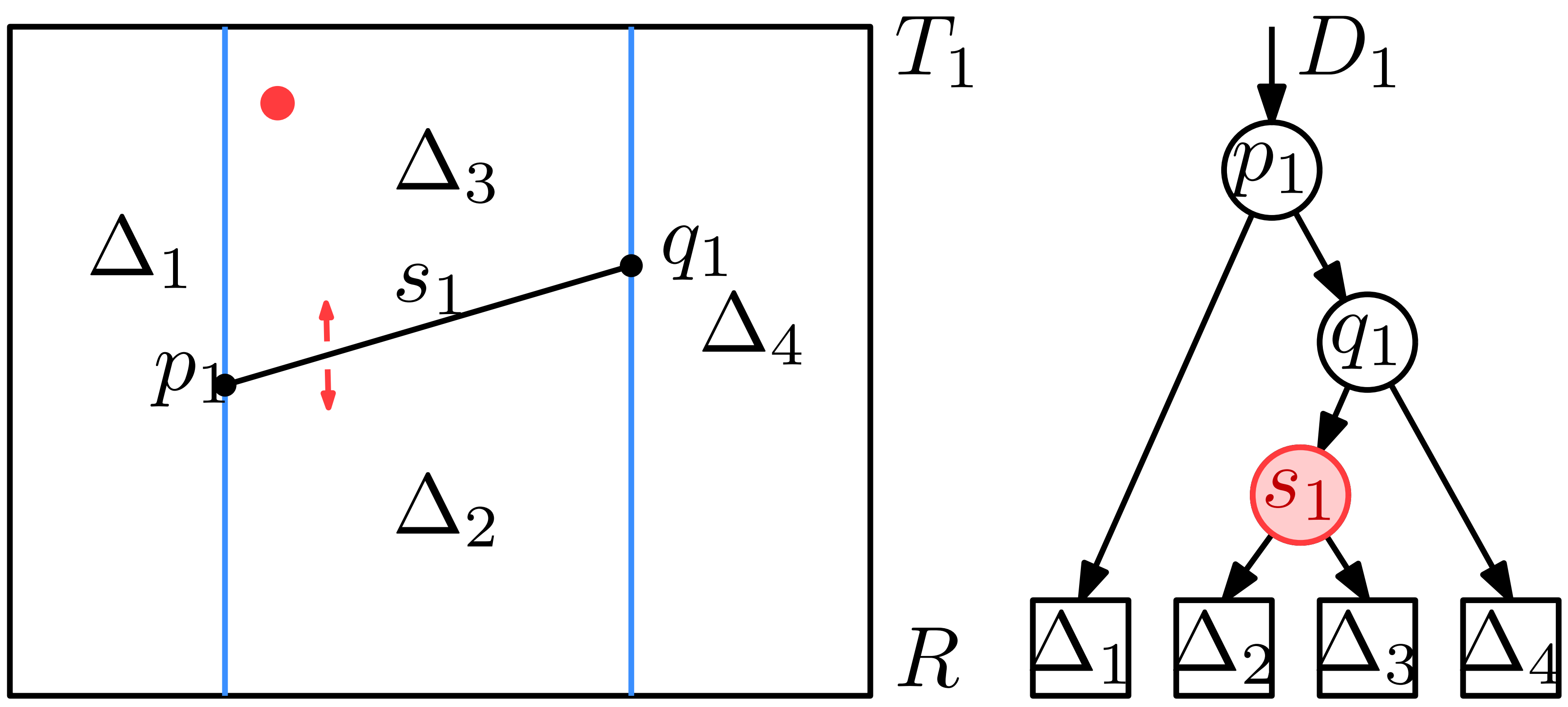
For any query point t, we only test at an x-node: Is t left or right of the vertical line through the stored point?

Point location solution



For any query point t, we only test at an y-node: Is t below or above the stored line segment?

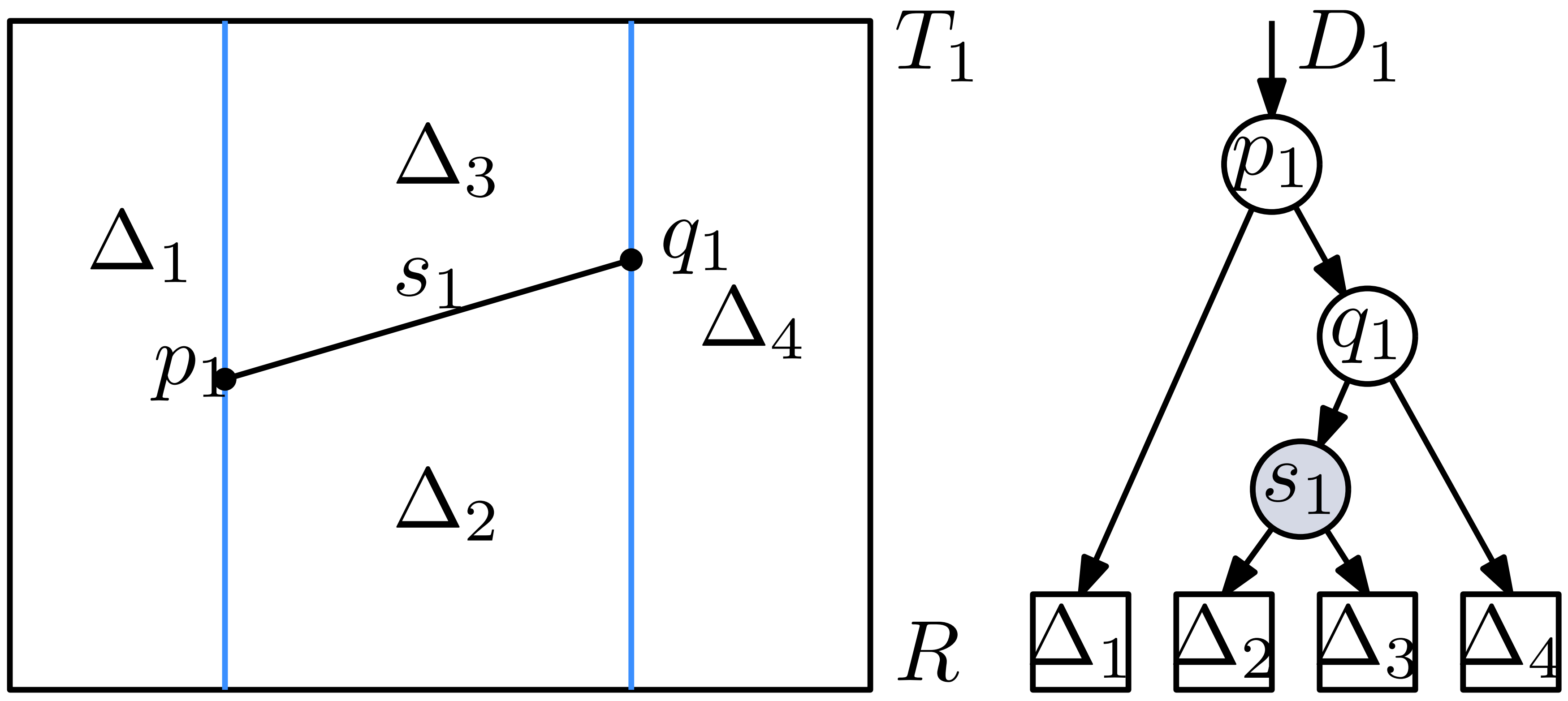
Point location solution



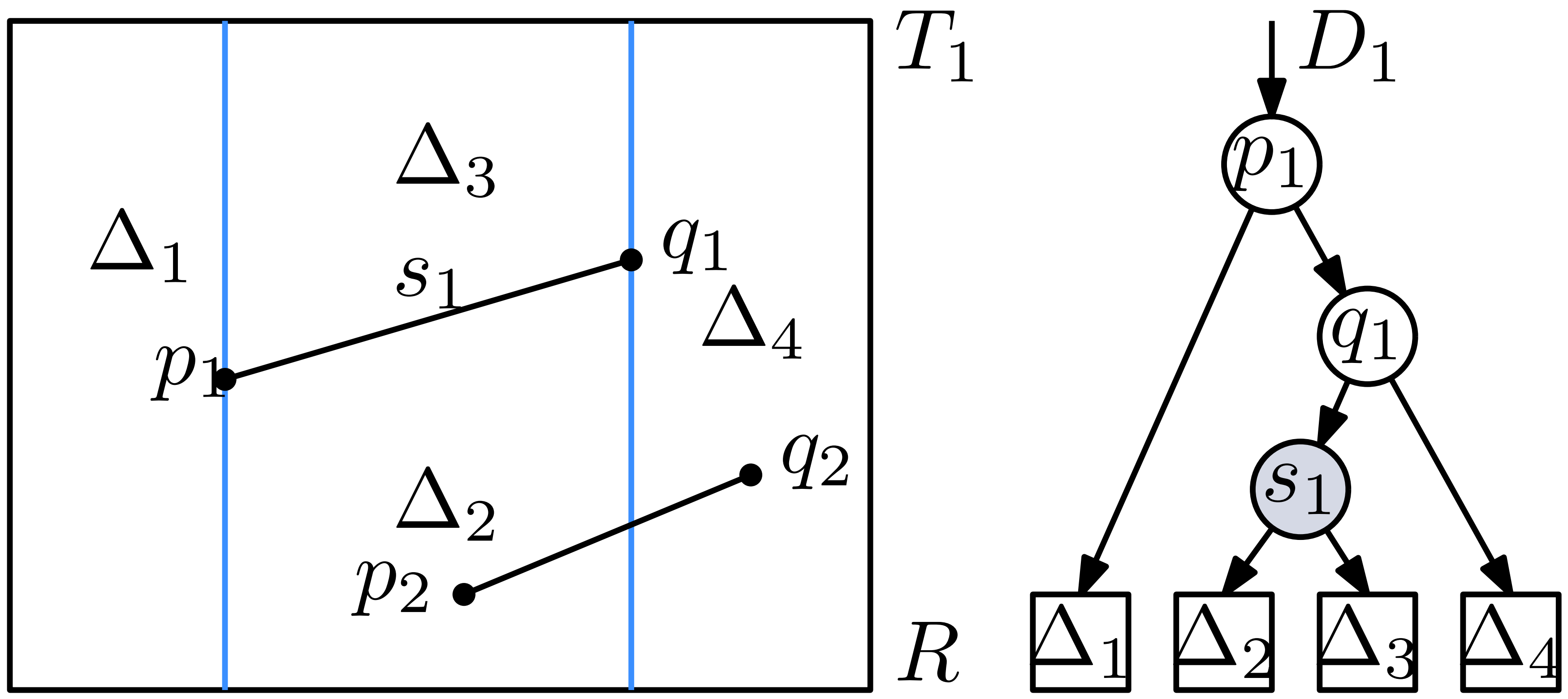
For any query point t, we only test at an y-node: Is t below or above the stored line segment?

We will guarantee that the question at a y-node is only asked if the query point t is between the vertical lines through p_j and q_j, if line segment $s_j = \overline{p_j q_j}$ is stored

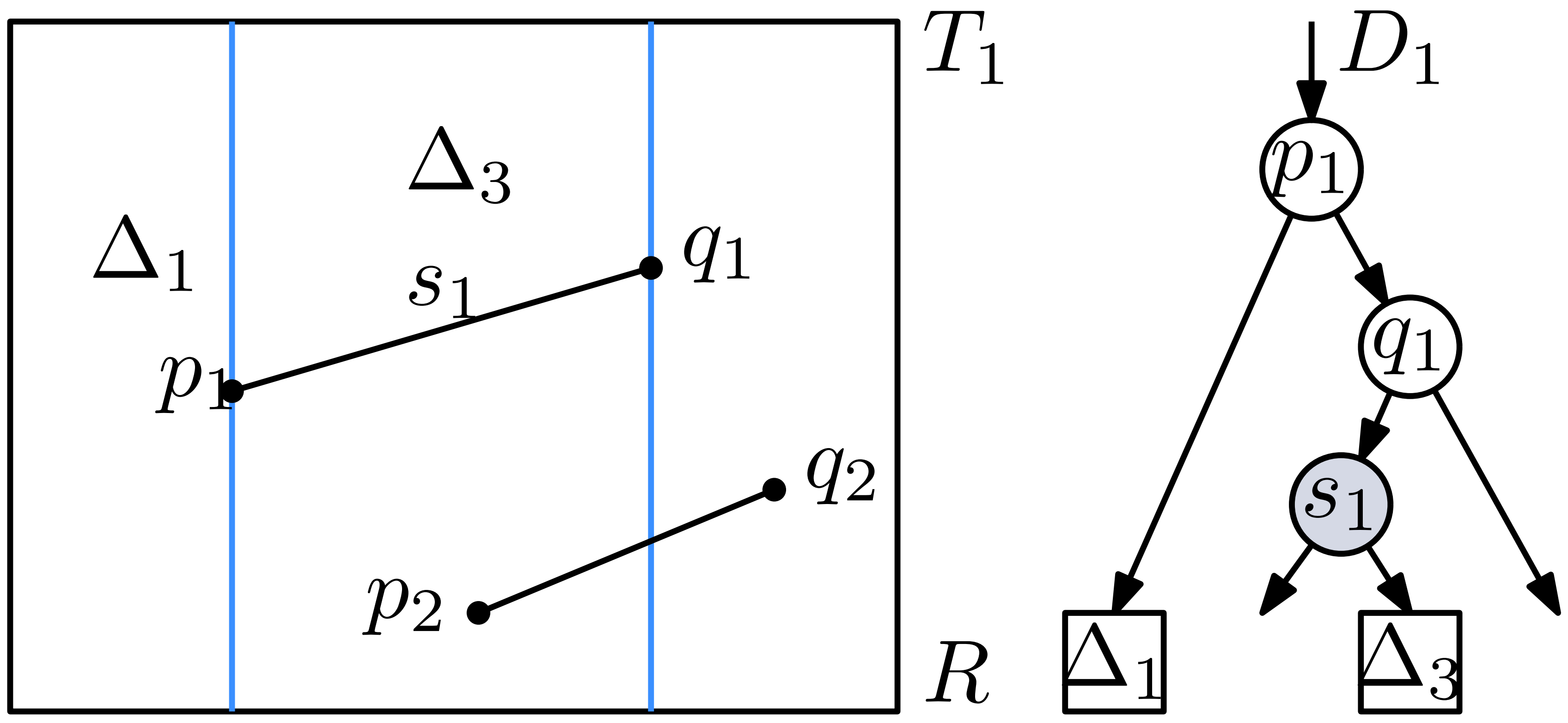
Point location solution



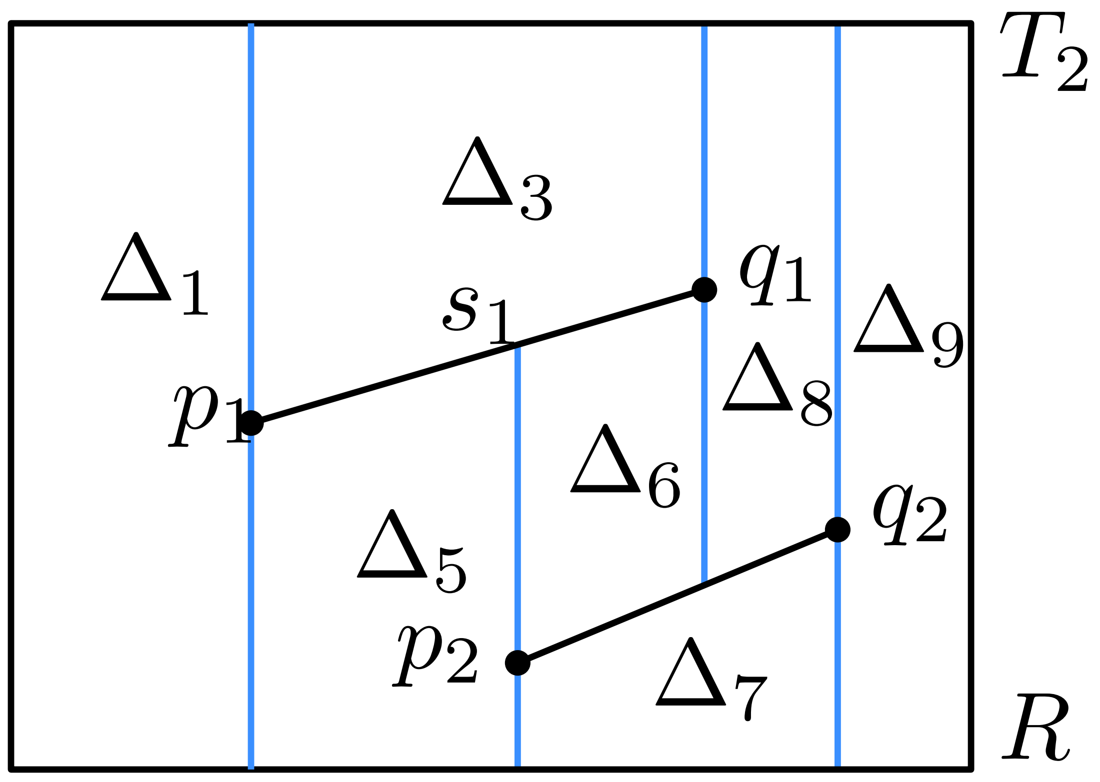
Point location solution

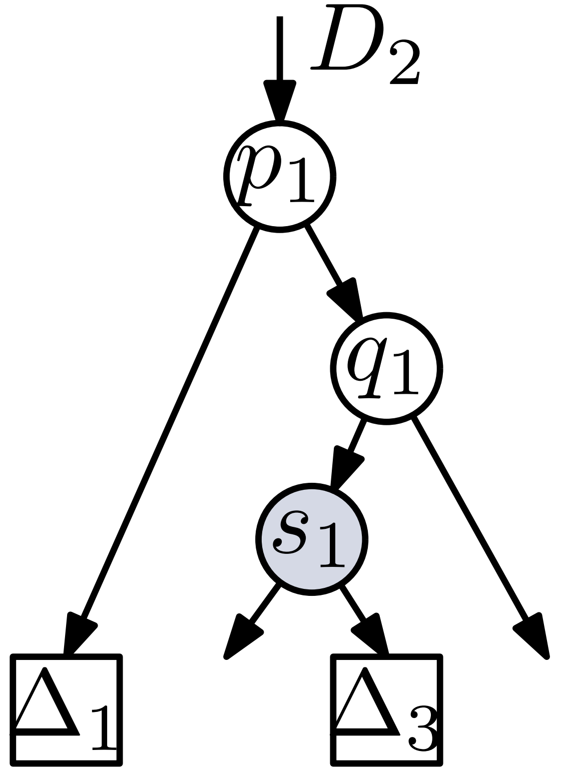


Point location solution

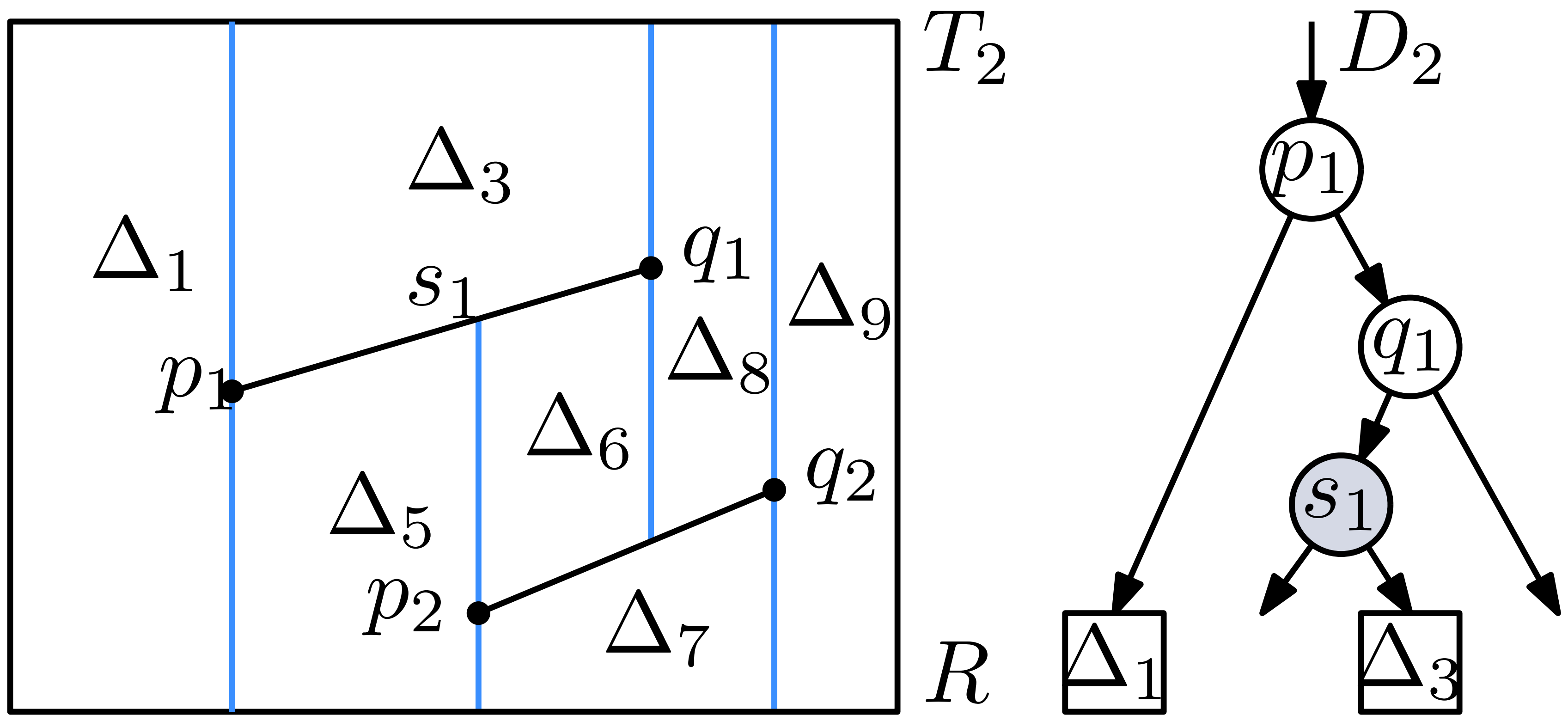


Point location solution





Point location solution



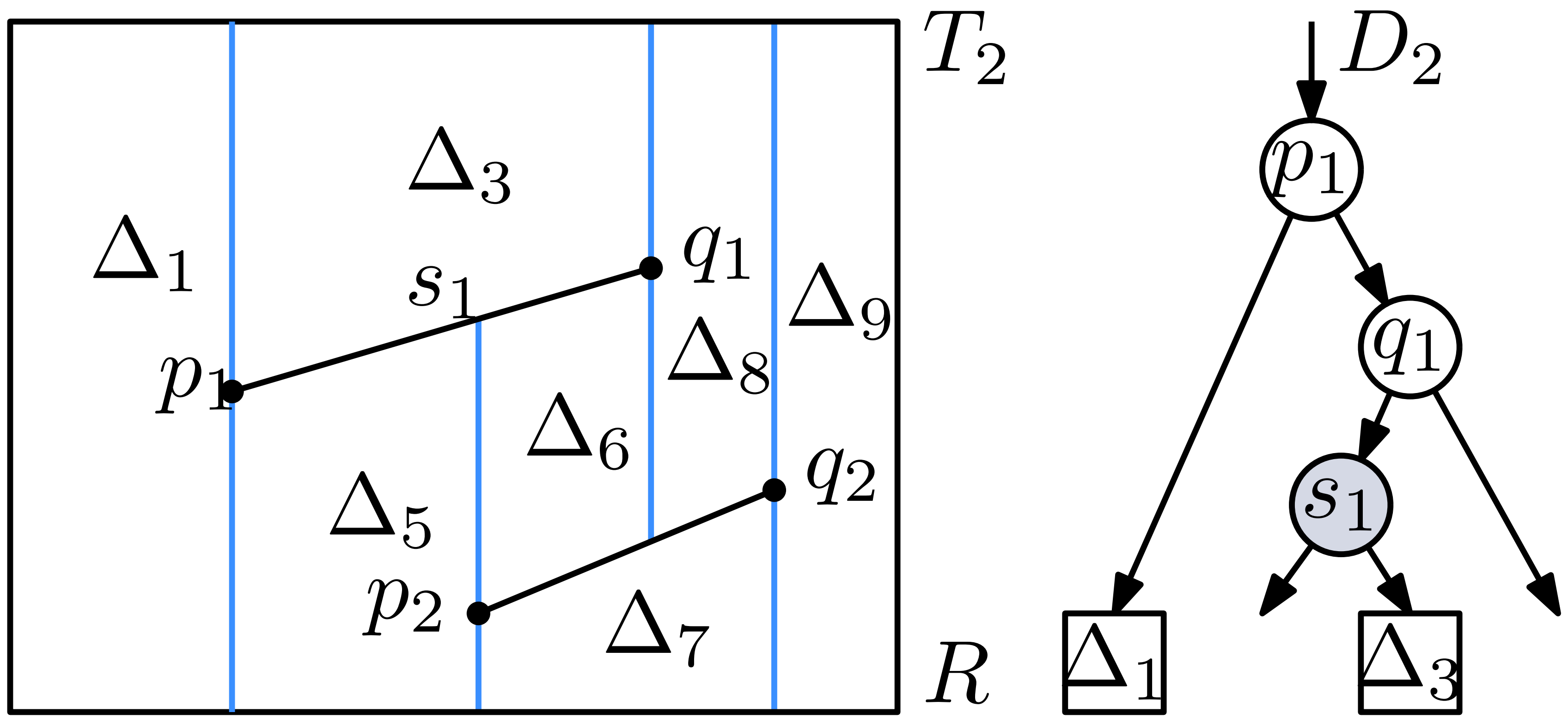
Which node will be the left child of the s_1 node?

A: p_2

B: s_2

C: Δ_6

Point location solution



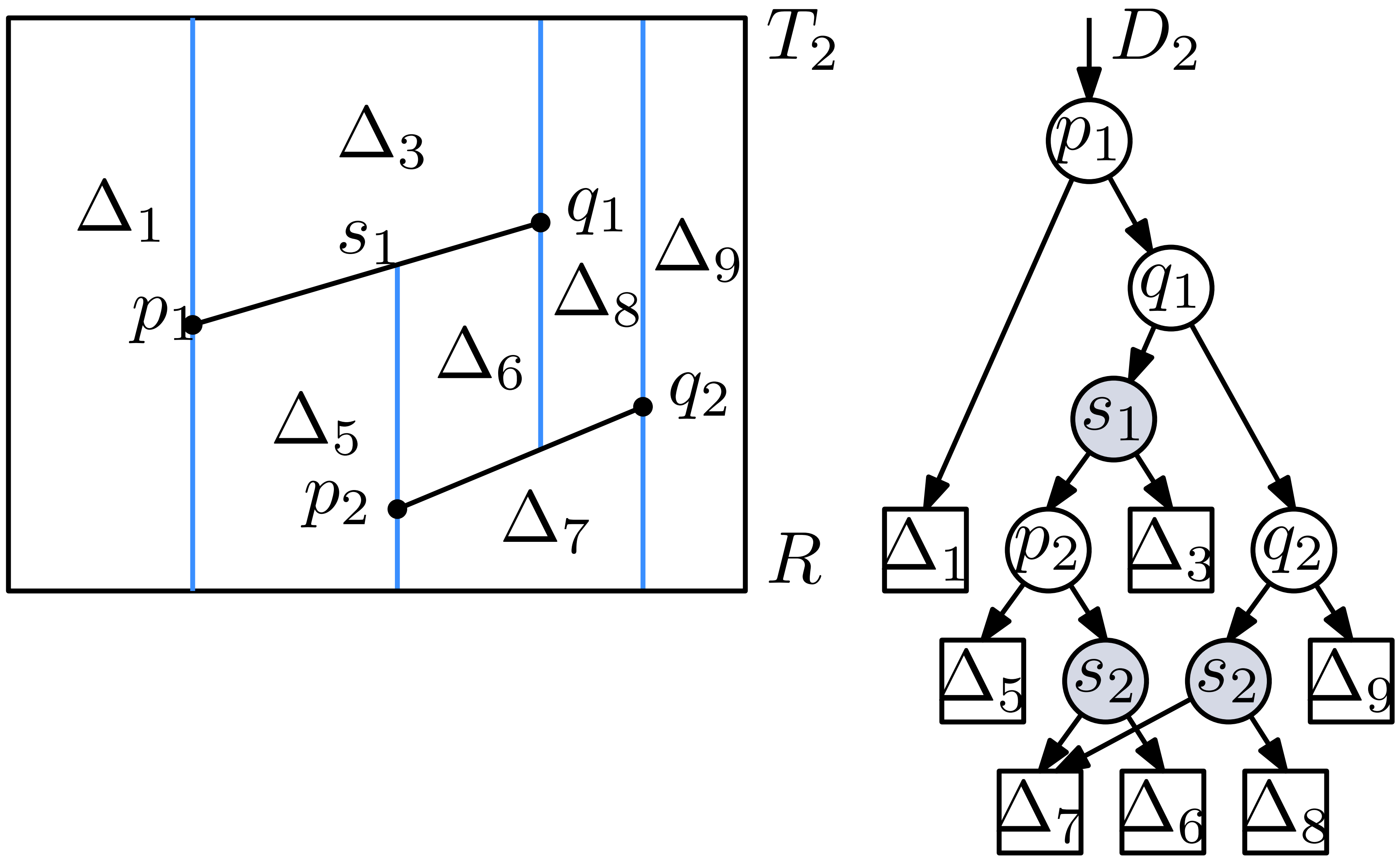
Which node will be the left child of the s_1 node?

A: p_2

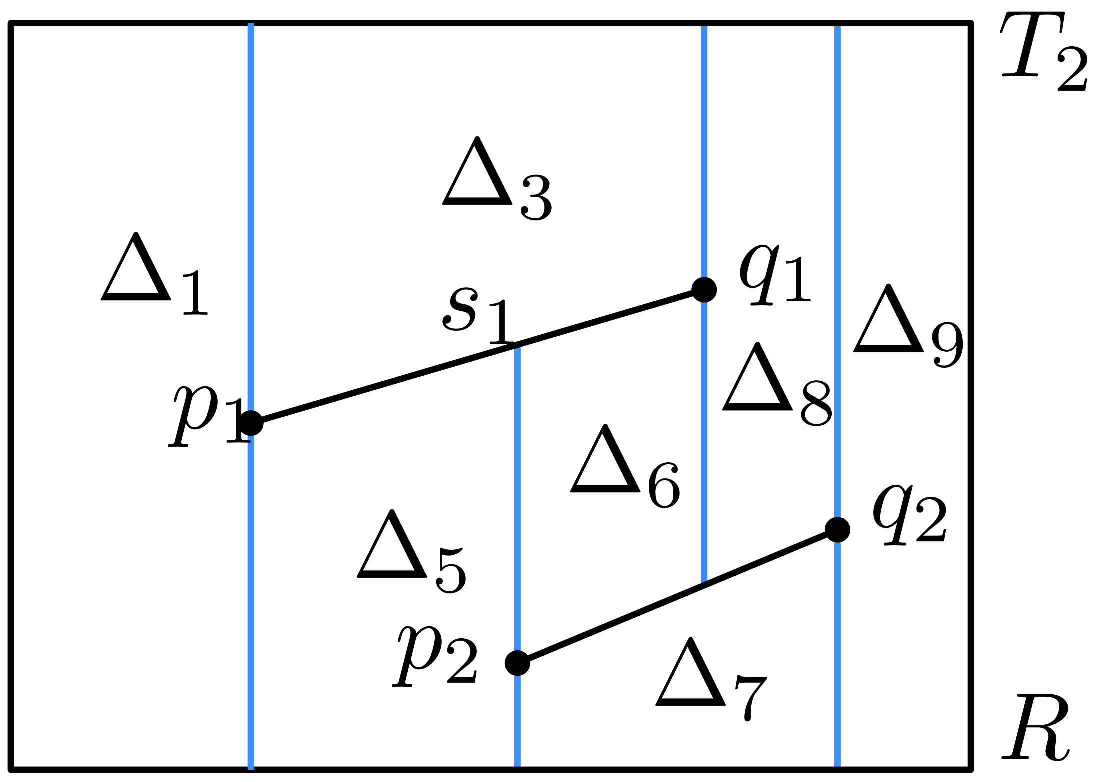
B: s_2

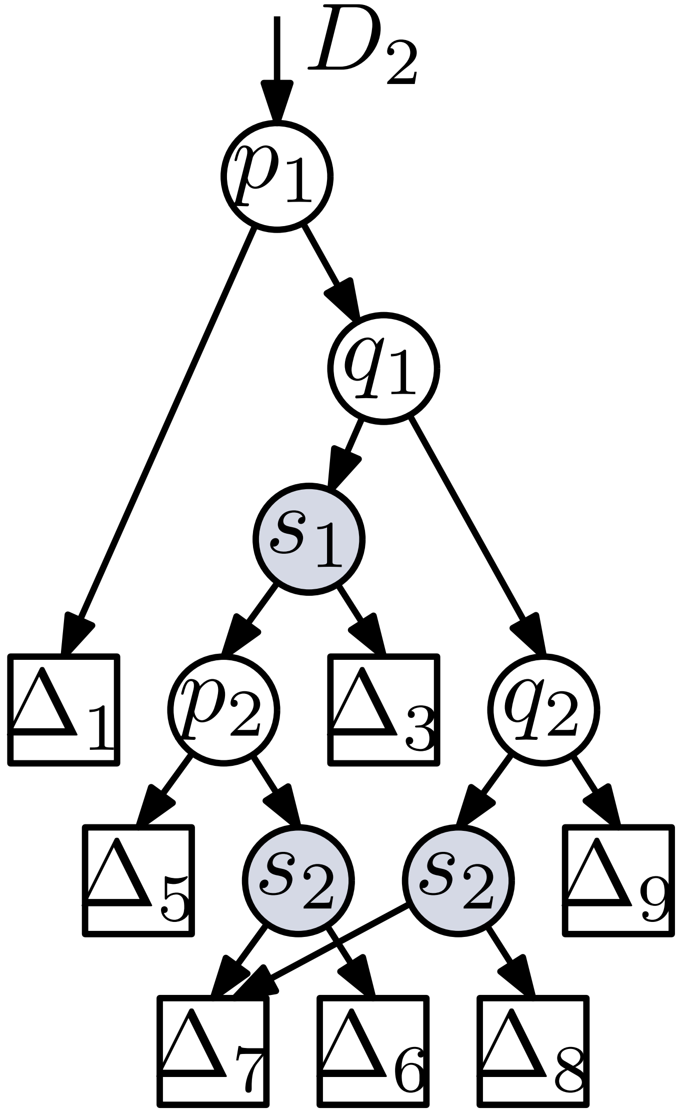
C: Δ_6

Point location solution



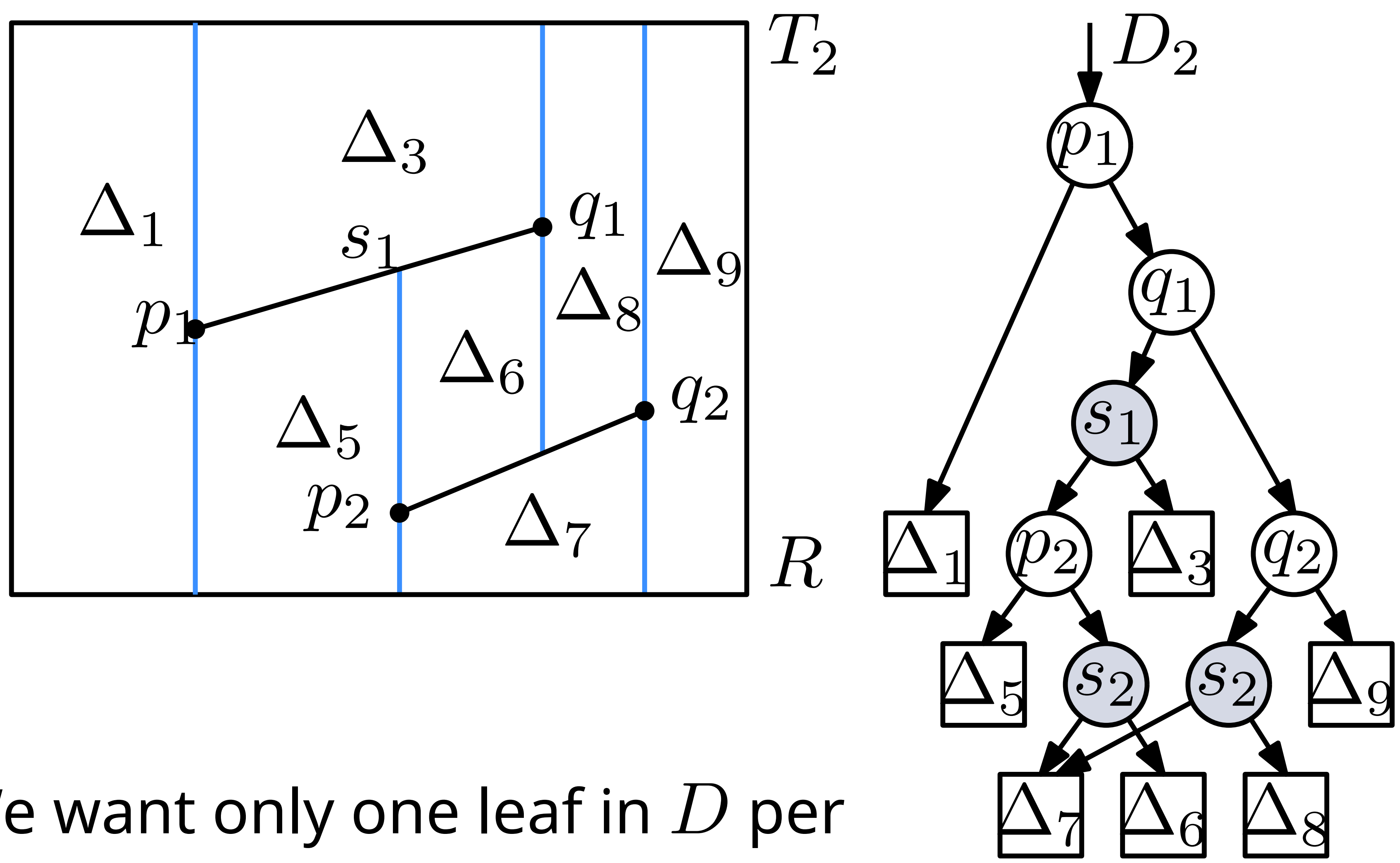
Point location solution





We want only one leaf in D per trapezoid; therefore, we get a search graph instead of a search tree

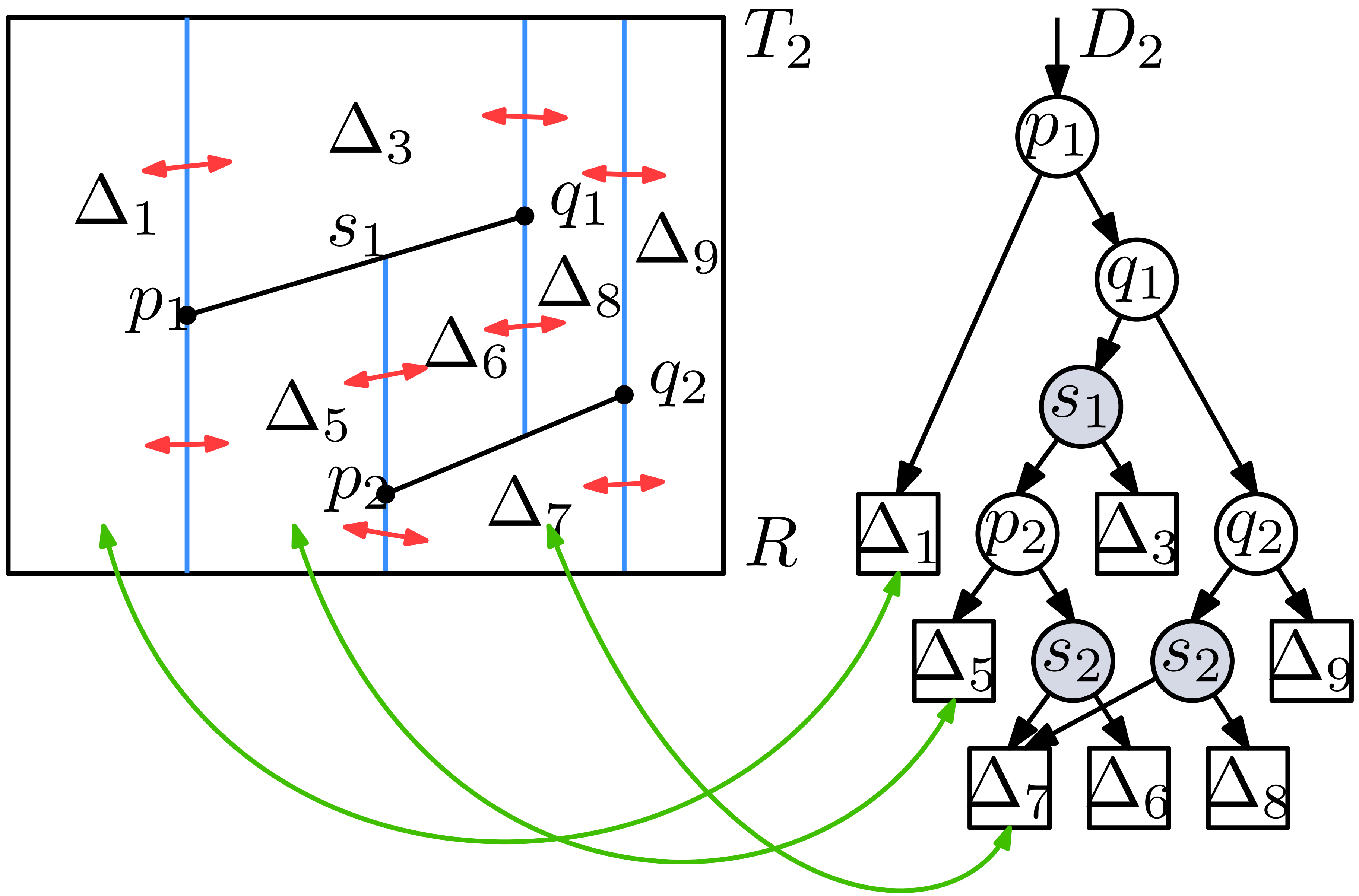
Point location solution



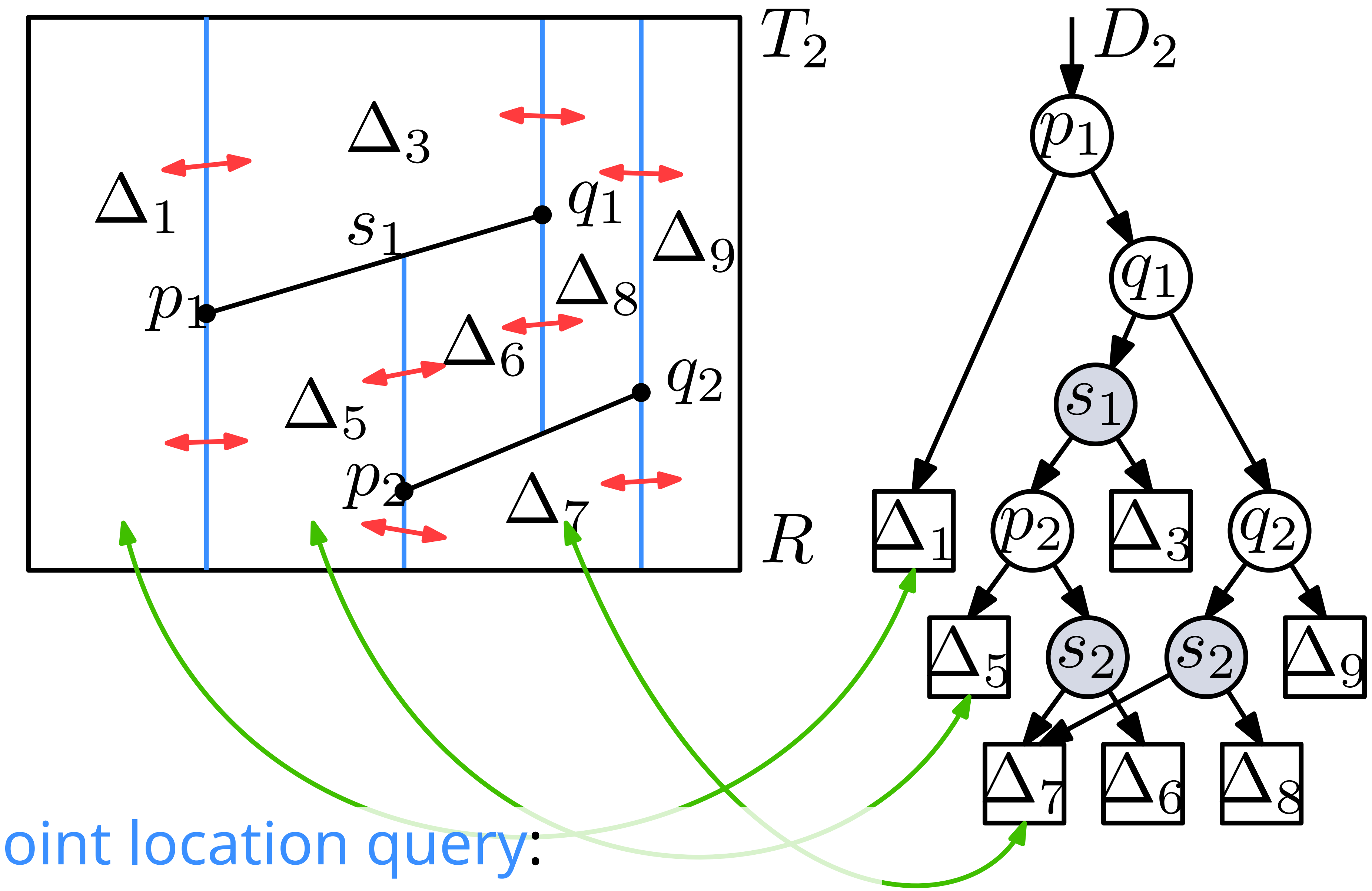
We want only one leaf in D per trapezoid; therefore, we get a search graph instead of a search tree

It is a directed acyclic graph, or DAG, hence the name D

Point location solution



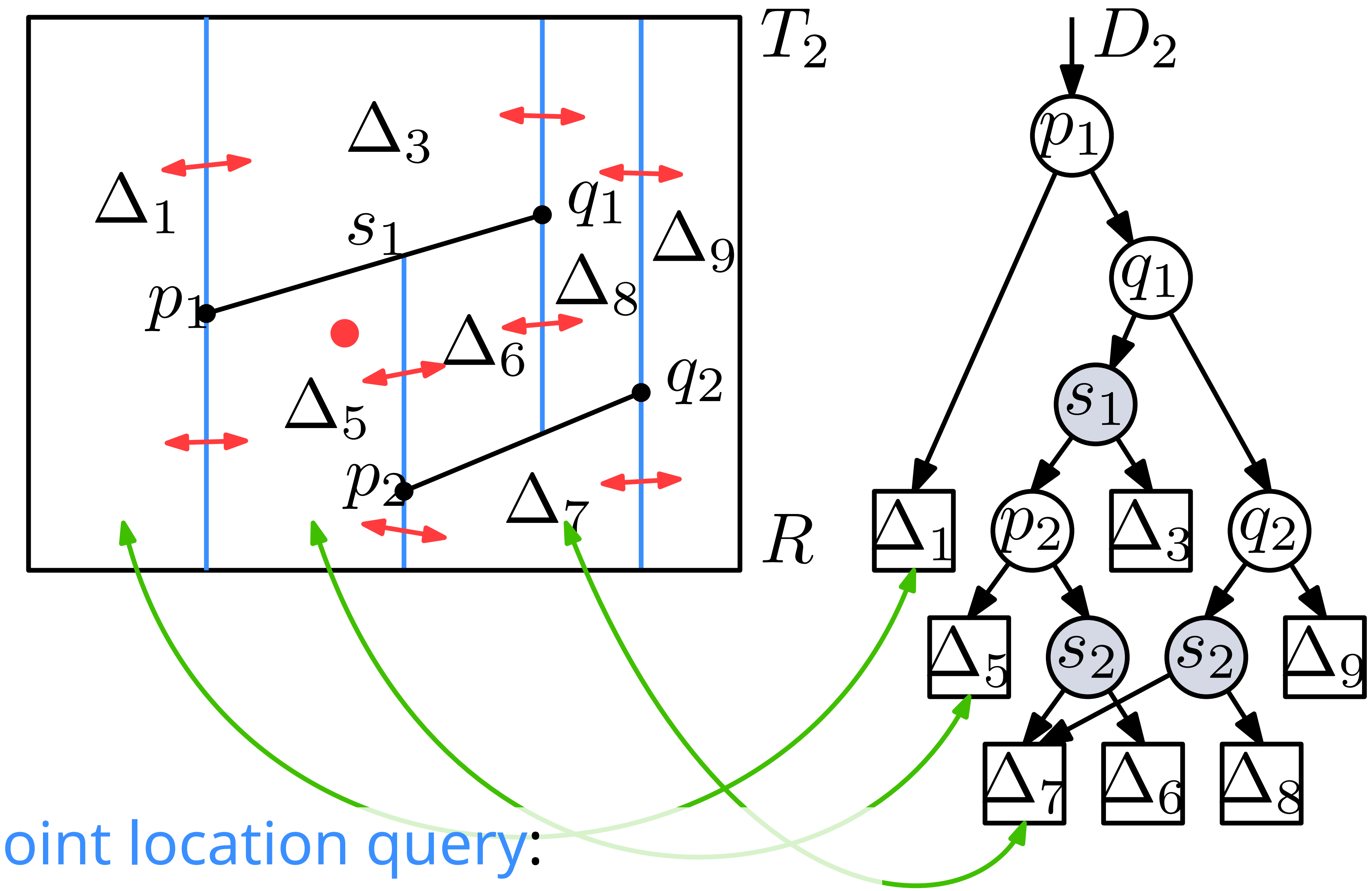
Point location solution



point location query:

- follow path in search structure D to leaf
- follow pointer to trapezoid T
- access bottom$(..)$ of T, and report name of face

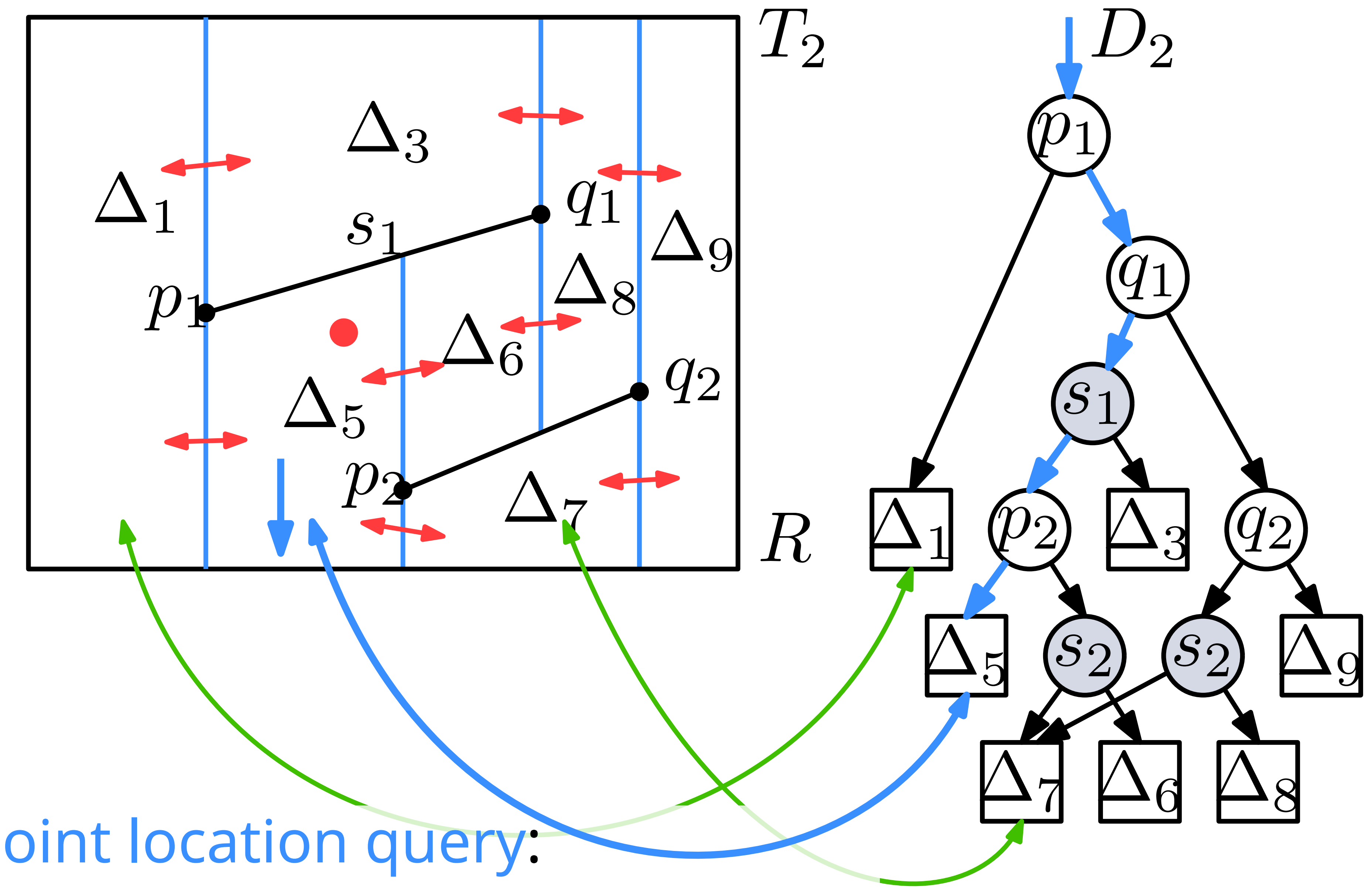
Point location solution



point location query:

- follow path in search structure D to leaf
- follow pointer to trapezoid T
- access bottom$(..)$ of T, and report name of face

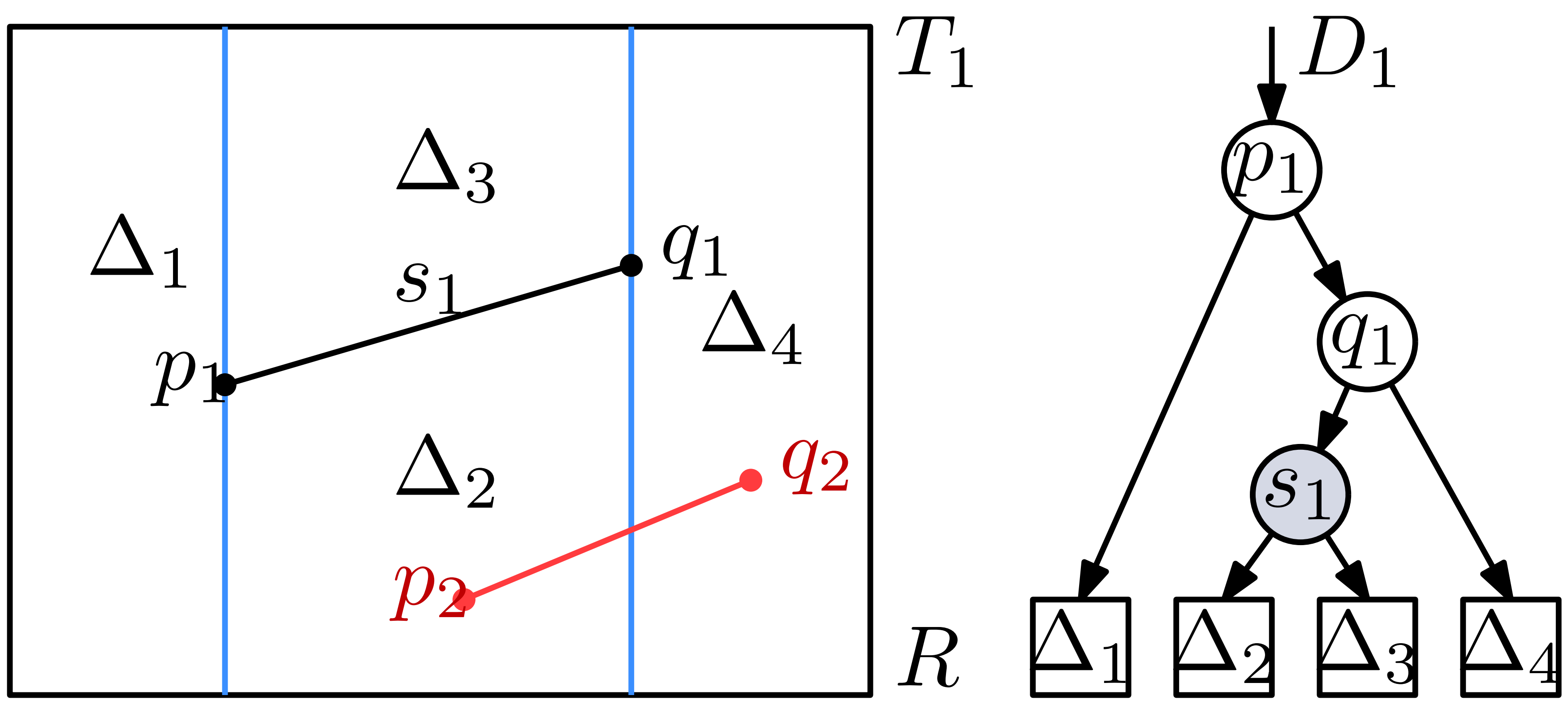
Point location solution



point location query:

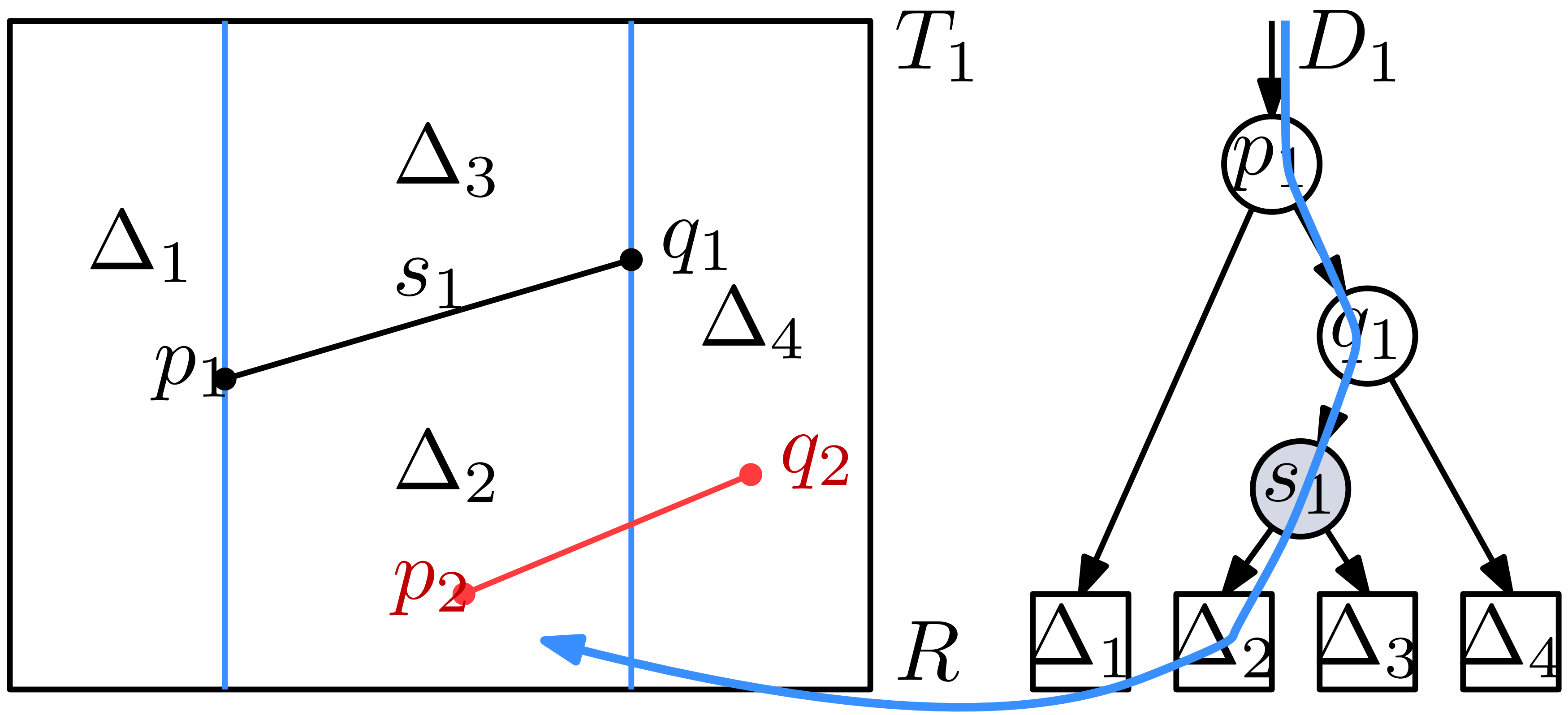
- follow path in search structure D to leaf
- follow pointer to trapezoid T
- access bottom$(..)$ of T, and report name of face

Incremental step



Suppose we have D_{i-1} and T_{i-1}, how do we add s_i?

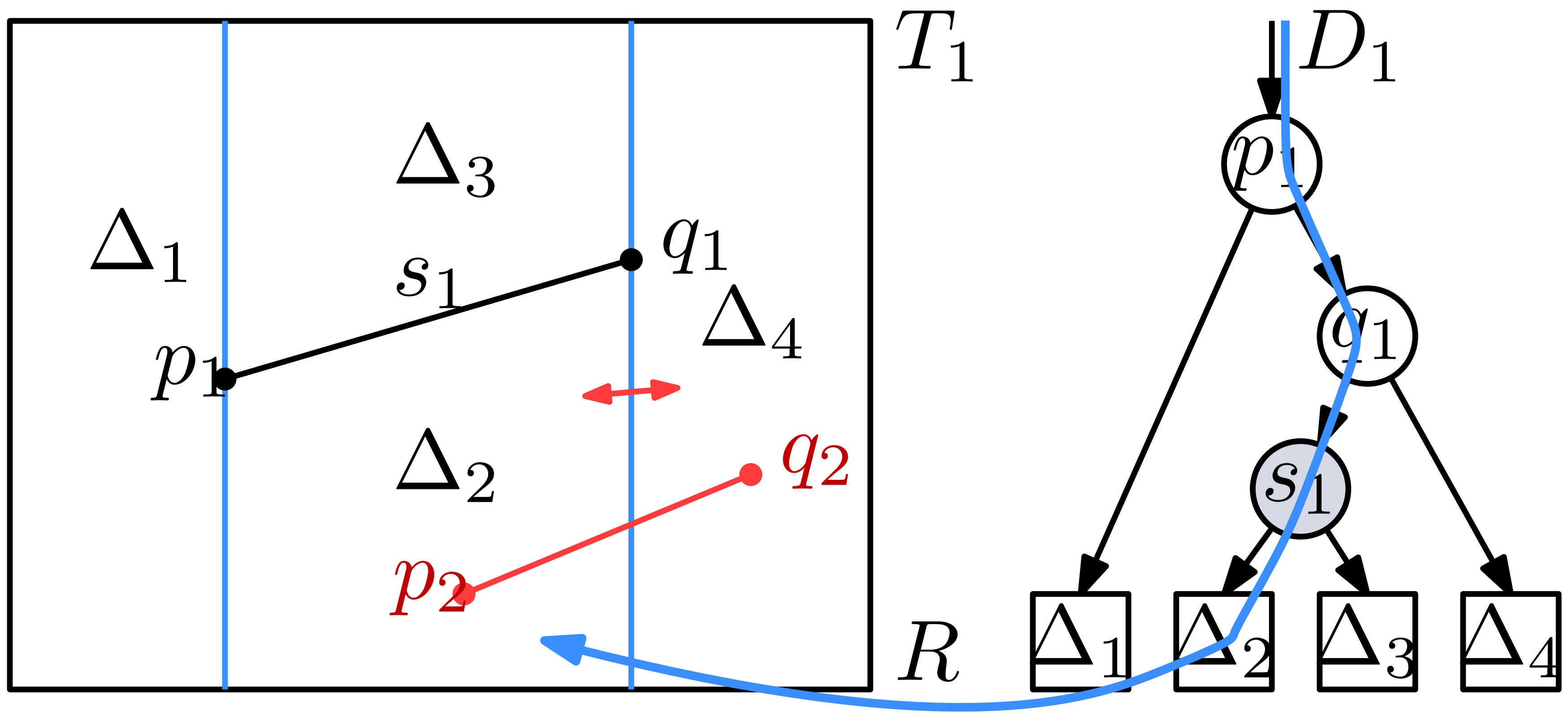
Incremental step



Suppose we have D_{i-1} and T_{i-1}, how do we add s_i?

Because D_{i-1} is a valid point location structure for $s_1, \ldots, s_{i-1}$, we can use it to find the trapezoid of T_{i-1} that contains p_i, the left endpoint of s_i.

Incremental step

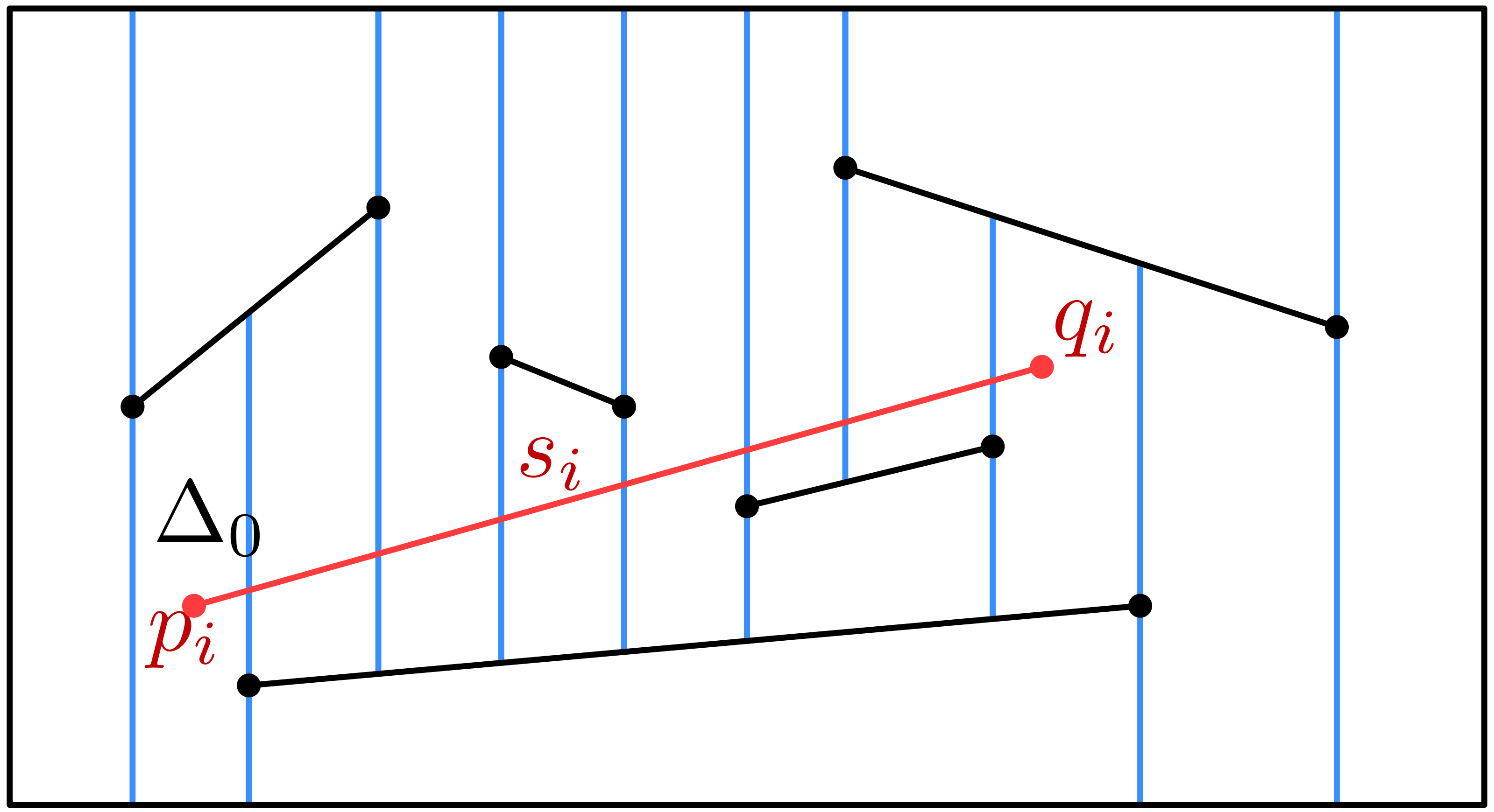


Suppose we have D_{i-1} and T_{i-1}, how do we add s_i?

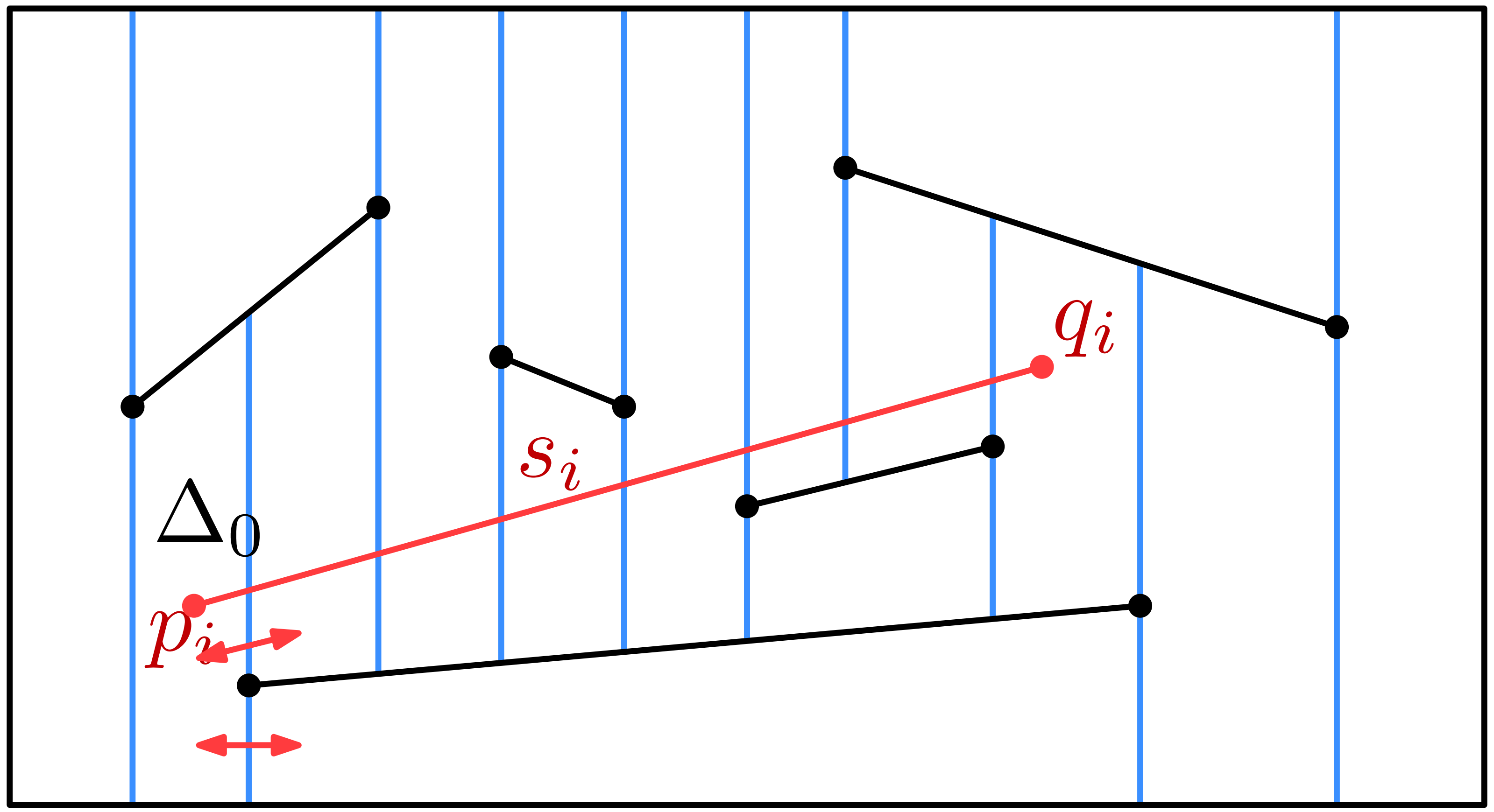
Because D_{i-1} is a valid point location structure for $s_1, \ldots, s_{i-1}$, we can use it to find the trapezoid of T_{i-1} that contains p_i, the left endpoint of s_i.

Then we use T_{i-1} to find all other trapezoids that intersect s_i.

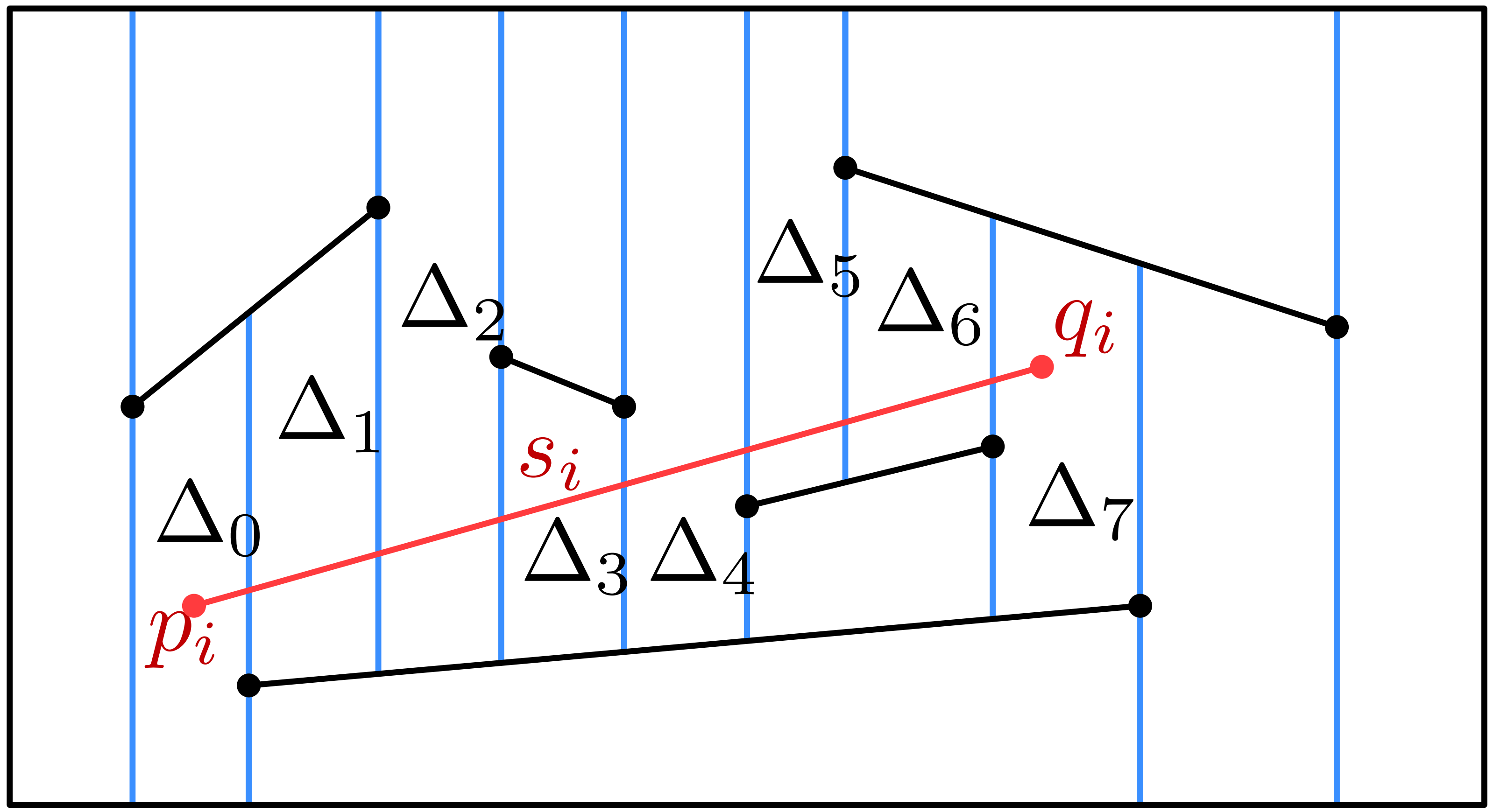
Find intersected trapezoids



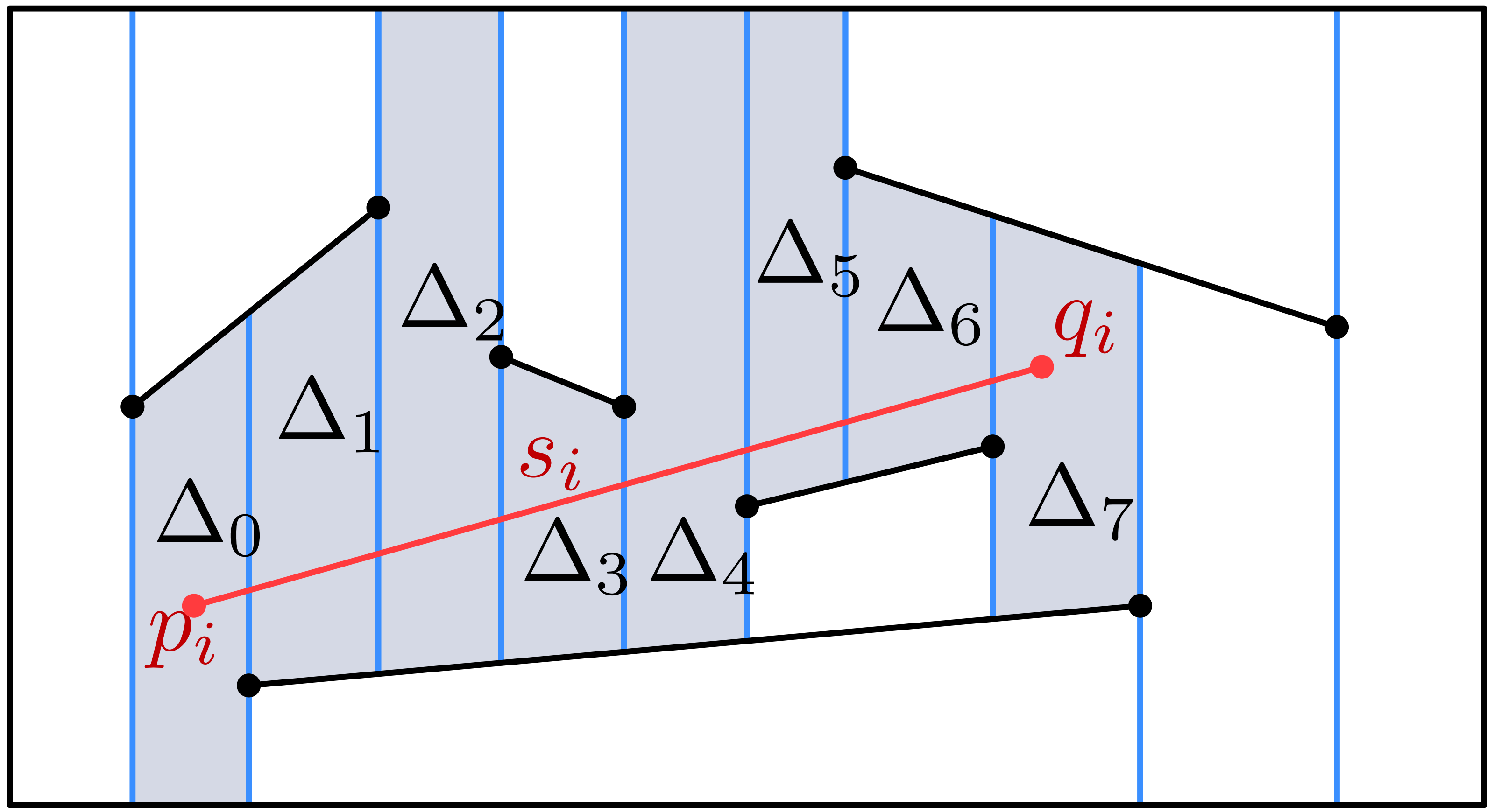
Find intersected trapezoids



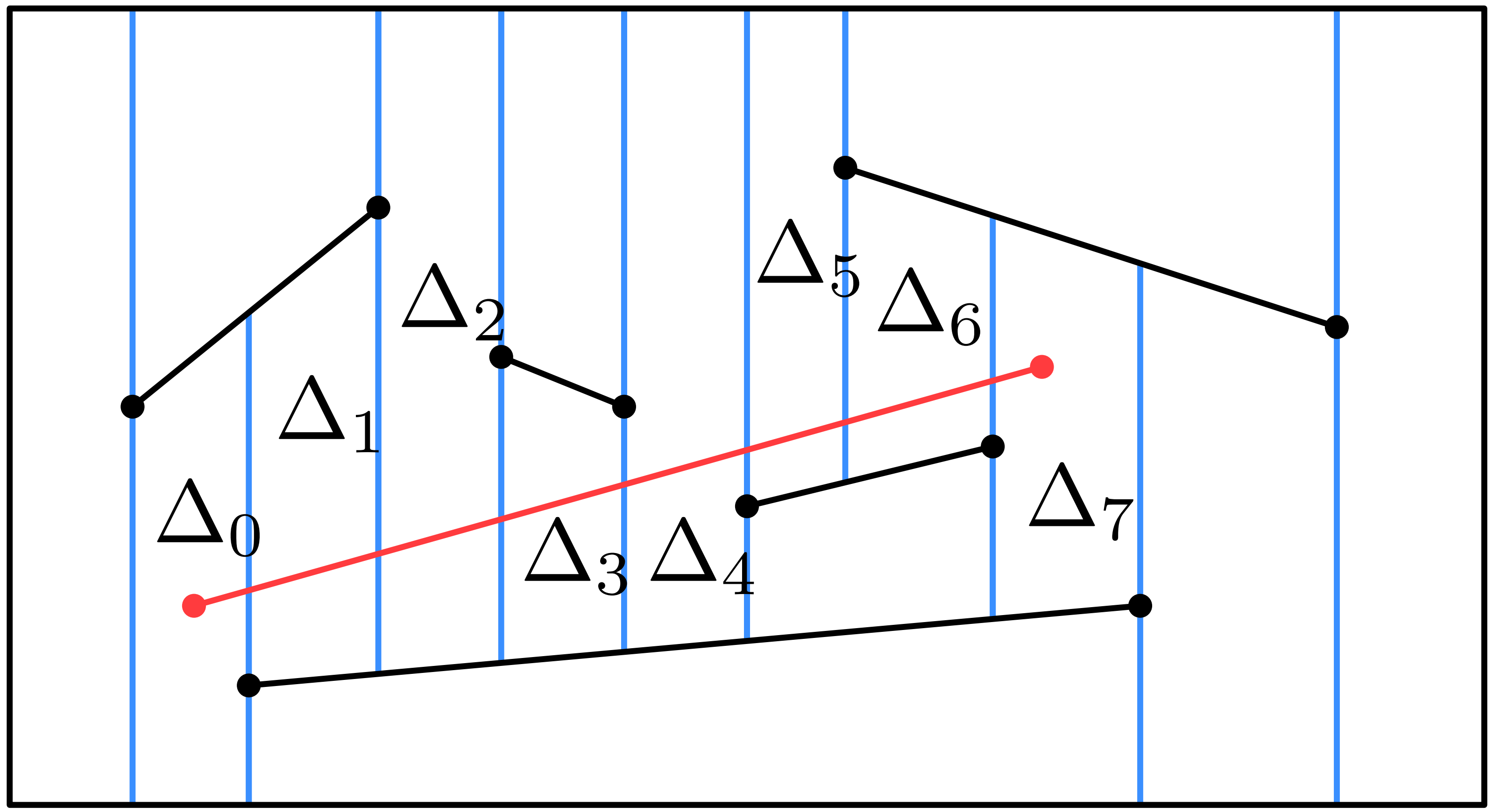
Find intersected trapezoids



Find intersected trapezoids

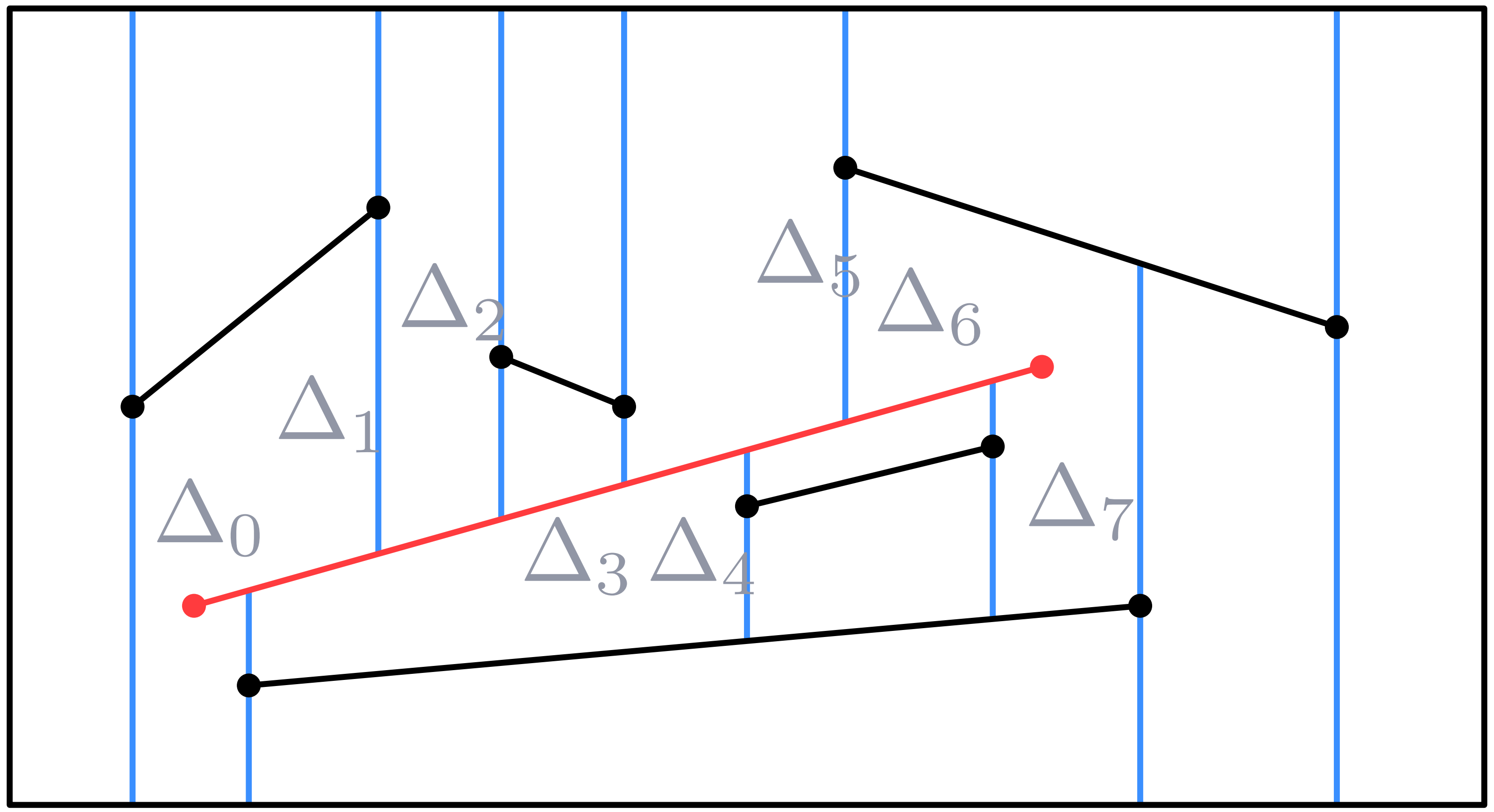


Find intersected trapezoids



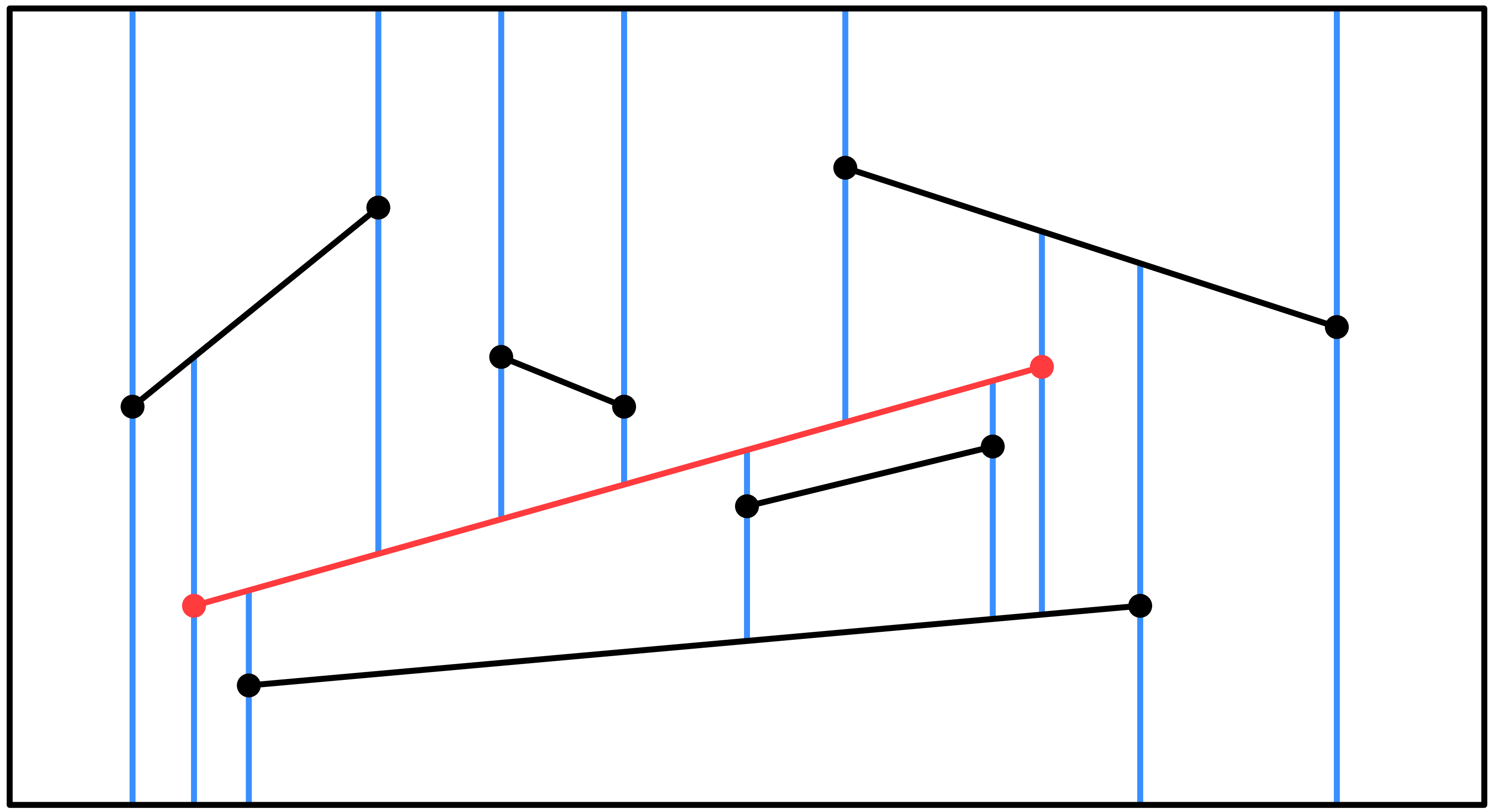
After locating the trapezoid that contains p_i, we can determine all k trapezoids that intersect s_i in $O(k)$ time by traversing T_{i-1}

Find intersected trapezoids



After locating the trapezoid that contains p_i, we can determine all k trapezoids that intersect s_i in $O(k)$ time by traversing T_{i-1}

Find intersected trapezoids



next:
updating the
search structure

After locating the trapezoid that contains p_i, we can determine all k trapezoids that intersect s_i in $O(k)$ time by traversing T_{i-1}

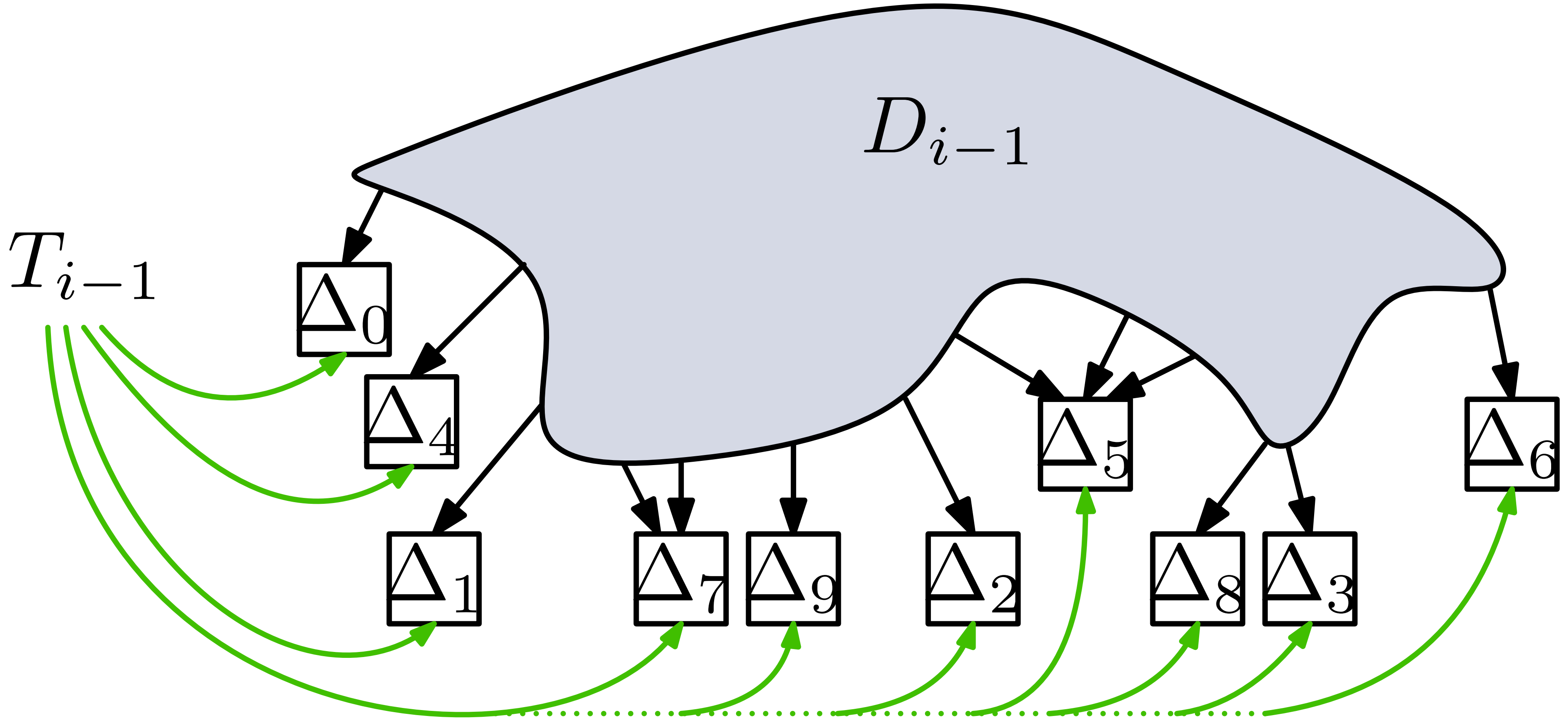
We can update the vertical decomposition in $O(k)$ time as well

Vertical Decomposition for Point Location

Randomized Incremental Construction: Updating the Search Structure

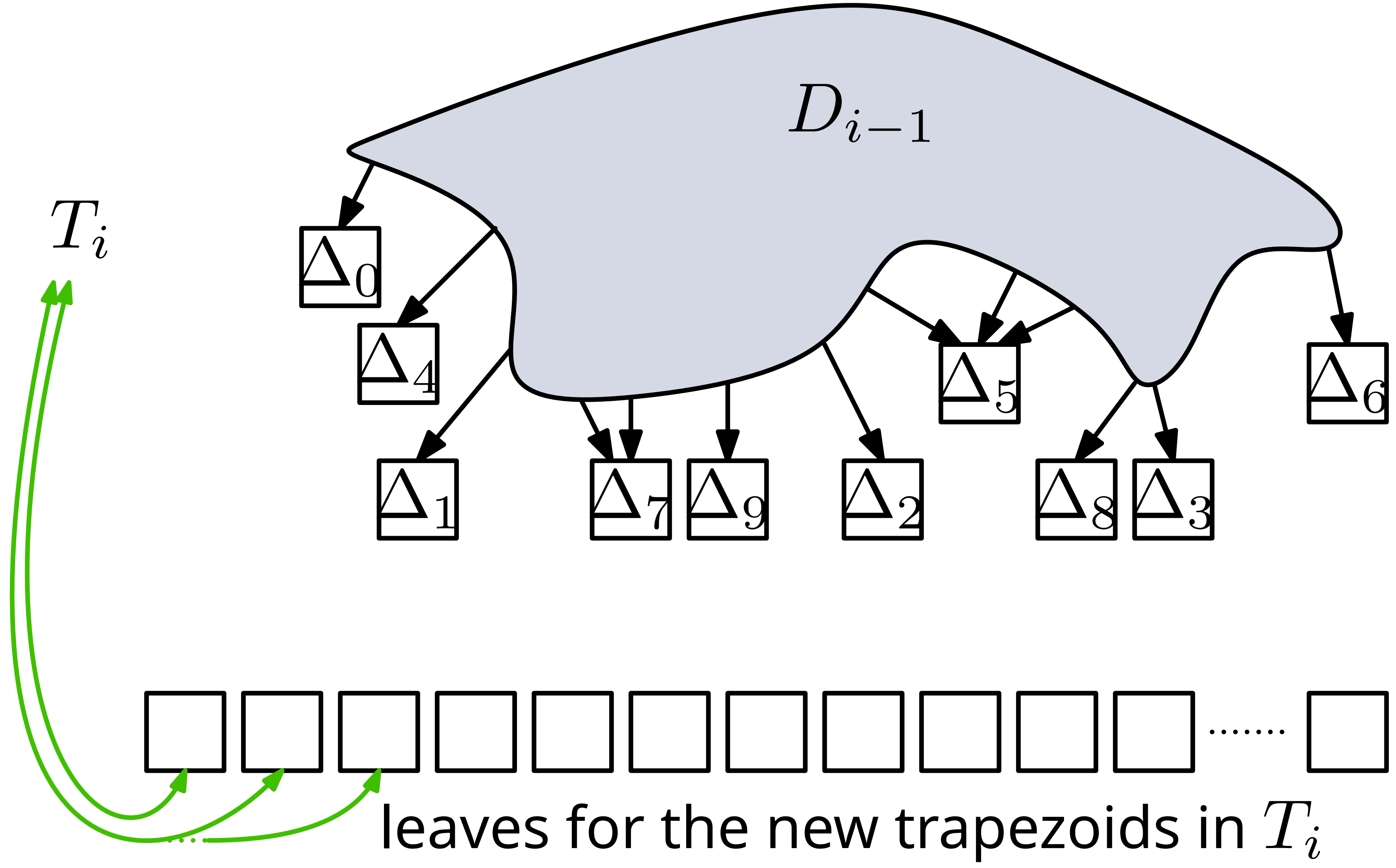
Updating the search structure

The search structure has k leaves that are no longer valid as leaves; we find these using the pointers from T_{i-1} to D_{i-1} and they become internal nodes



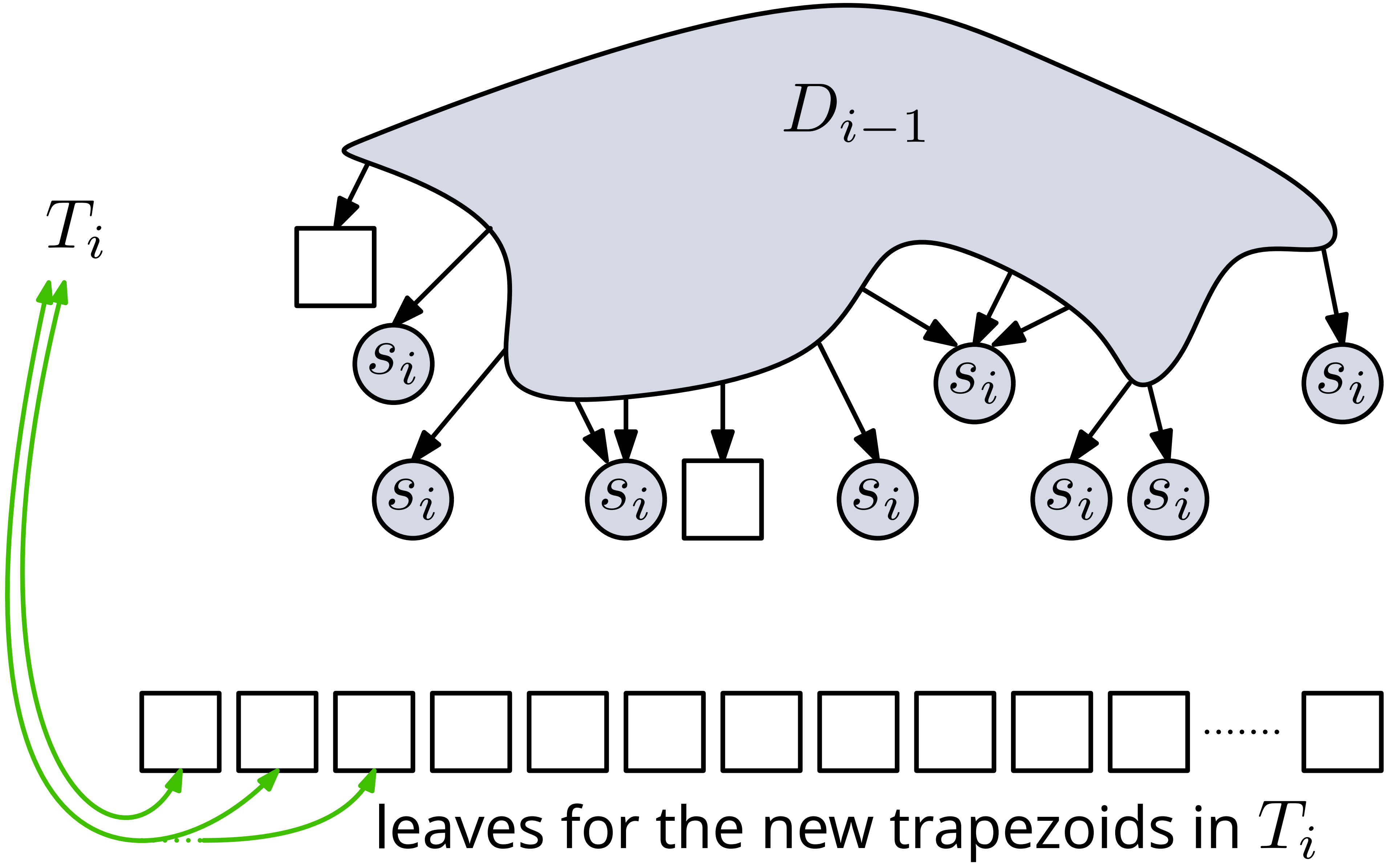
Updating the search structure

From the update of the vertical decomposition T_{i-1} into T_i we know what new leaves we must make for D_i



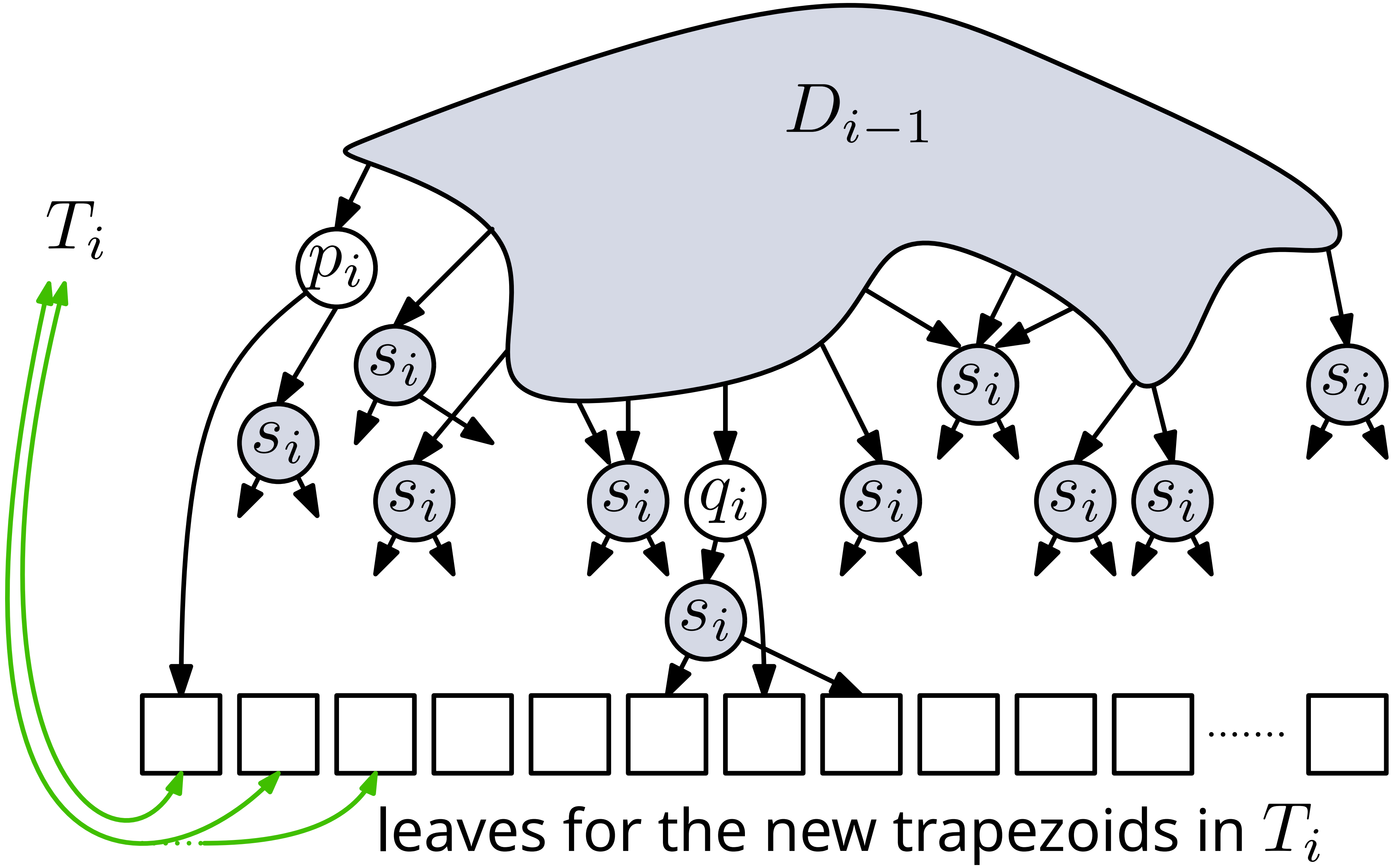
Updating the search structure

All new nodes besides the leaves are x-nodes with p_i and q_i and y-nodes with s_i



Updating the search structure

All new nodes besides the leaves are x-nodes with p_i and q_i and y-nodes with s_i



leaves for the new trapezoids in T_i

Observations

For a single update step, adding s_i and updating T_{i-1} and D_{i-1}, we observe:

- If s_i intersects k_i trapezoids of T_{i-1}, then we will create $O(k_i)$ new trapezoids in T_i
- We find the k_i trapezoids in time linear in the search path of p_i in D_{i-1}, plus $O(k_i)$ time
- We update by replacing k_i leaves by $O(k_i)$ new internal nodes and $O(k_i)$ new leaves
- The maximum depth increase is three nodes

Quiz

In what case does the length of a path increase by three nodes?

A: always

B: only if the new segment lies completely in one trapezoid

C: only in degenerate cases

Quiz

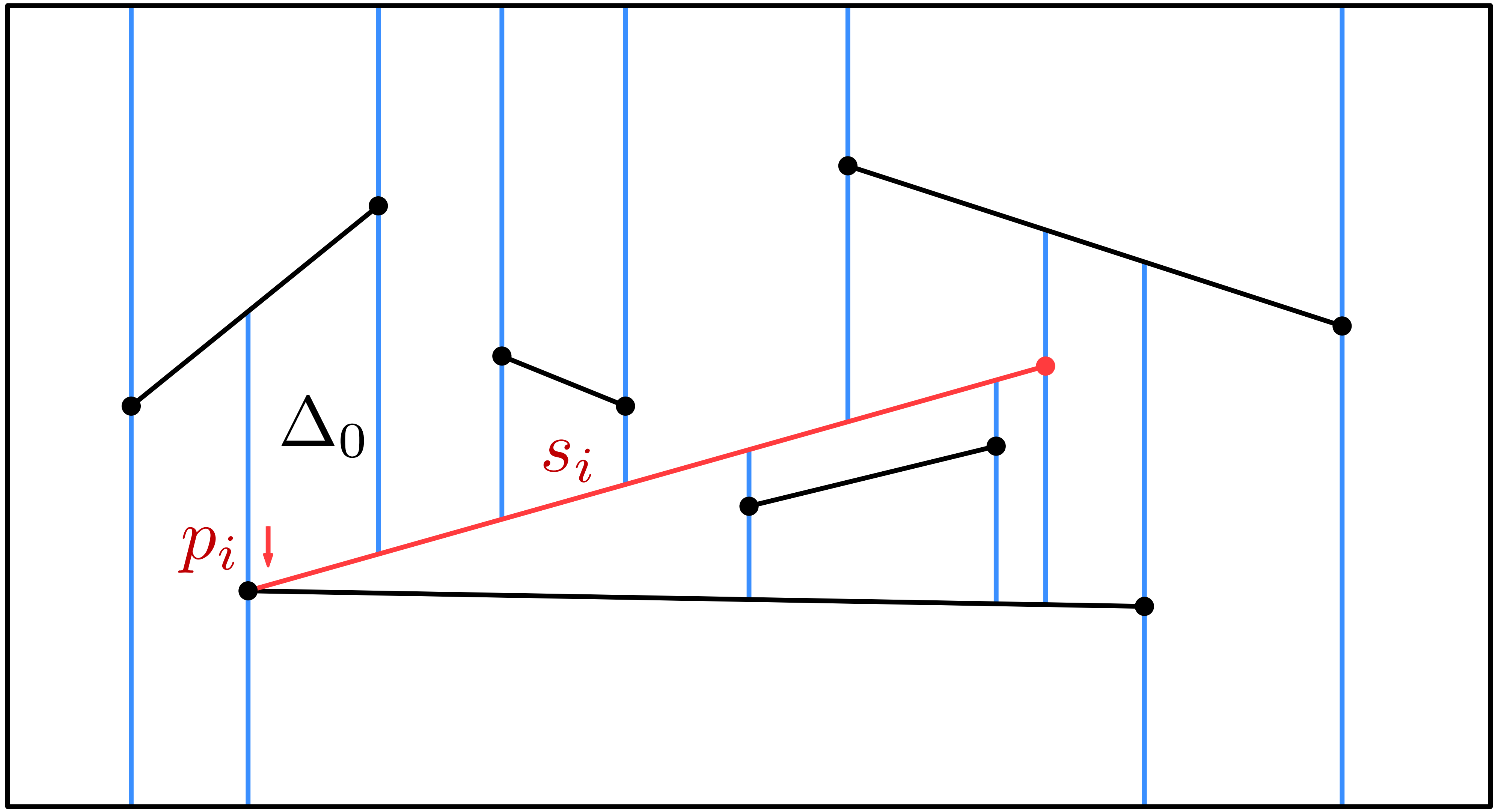
In what case does the length of a path increase by three nodes?

A: always

B: only if the new segment lies completely in one trapezoid

C: only in degenerate cases

A common but special update



If p_i was already an existing vertex, we search in D_{i-1} with a point a fraction to the right of p_i on s_i

Why Randomized?

Randomized incremental construction, where does it matter?

Why Randomized?

Randomized incremental construction, where does it matter?

- The vertical decomposition T_i is independent of the insertion order among $s_1, \ldots, s_i$

Why Randomized?

Randomized incremental construction, where does it matter?

- The vertical decomposition T_i is independent of the insertion order among $s_1, \ldots, s_i$
- The search structure D_i can be different for many orders of $s_1, \ldots, s_i$

Why Randomized?

Randomized incremental construction, where does it matter?

- The vertical decomposition T_i is independent of the insertion order among $s_1, \ldots, s_i$
- The search structure D_i can be different for many orders of $s_1, \ldots, s_i$
  - The depth of search paths in D_i depends on the order
  - The number of nodes in D_i depends on the order

Why Randomized?

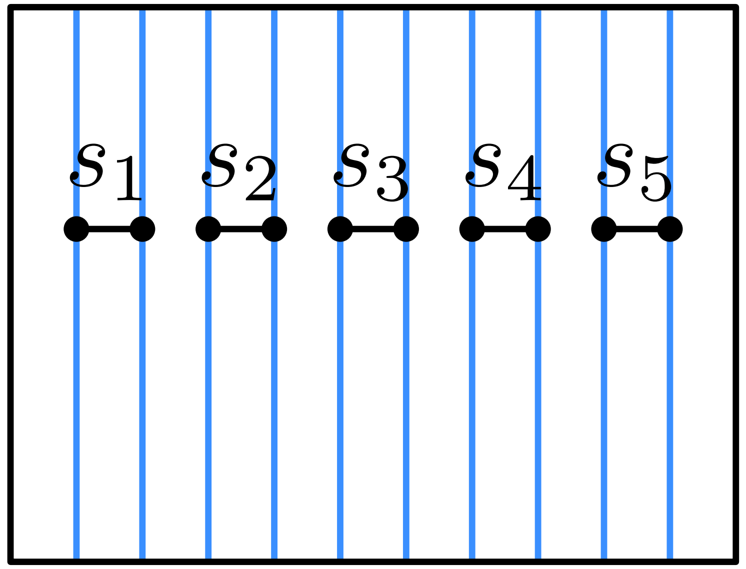
Randomized incremental construction, where does it matter?

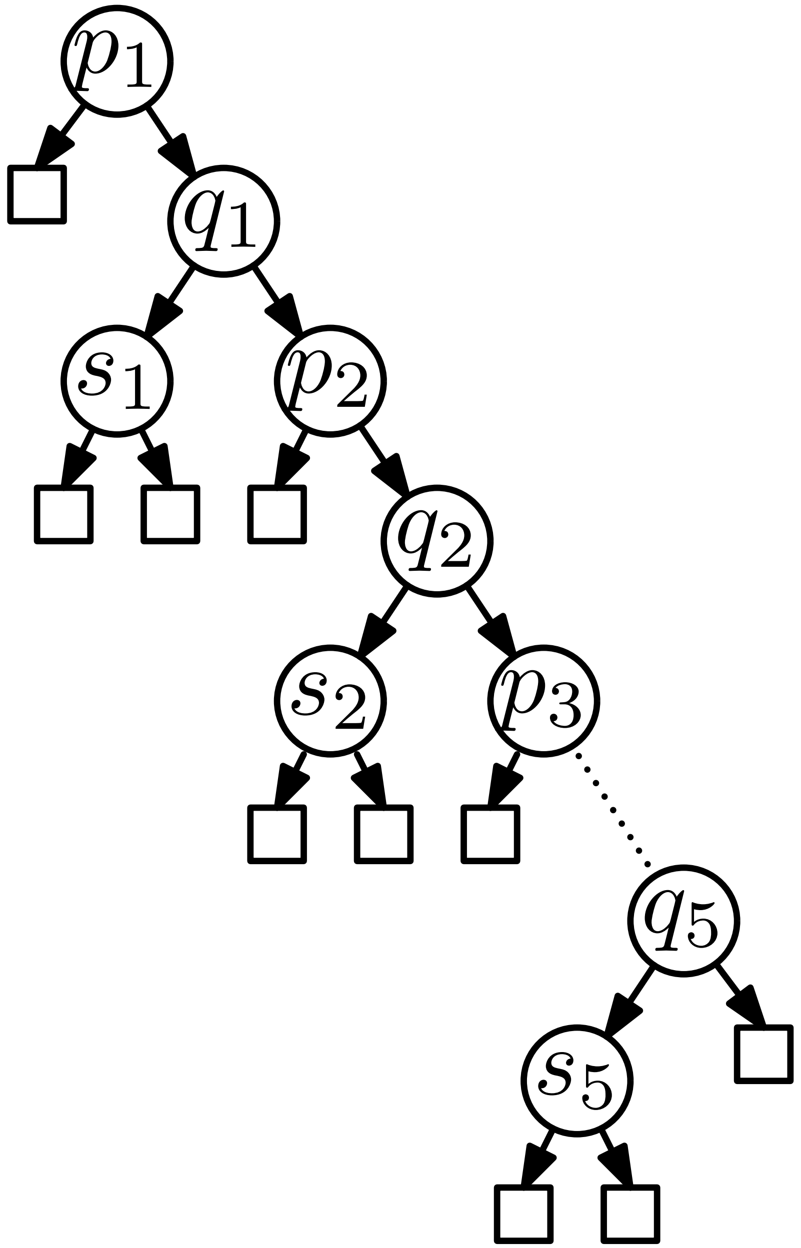
- The vertical decomposition T_i is independent of the insertion order among $s_1, \ldots, s_i$
- The search structure D_i can be different for many orders of $s_1, \ldots, s_i$
  - The depth of search paths in D_i depends on the order
  - The number of nodes in D_i depends on the order

Examples?

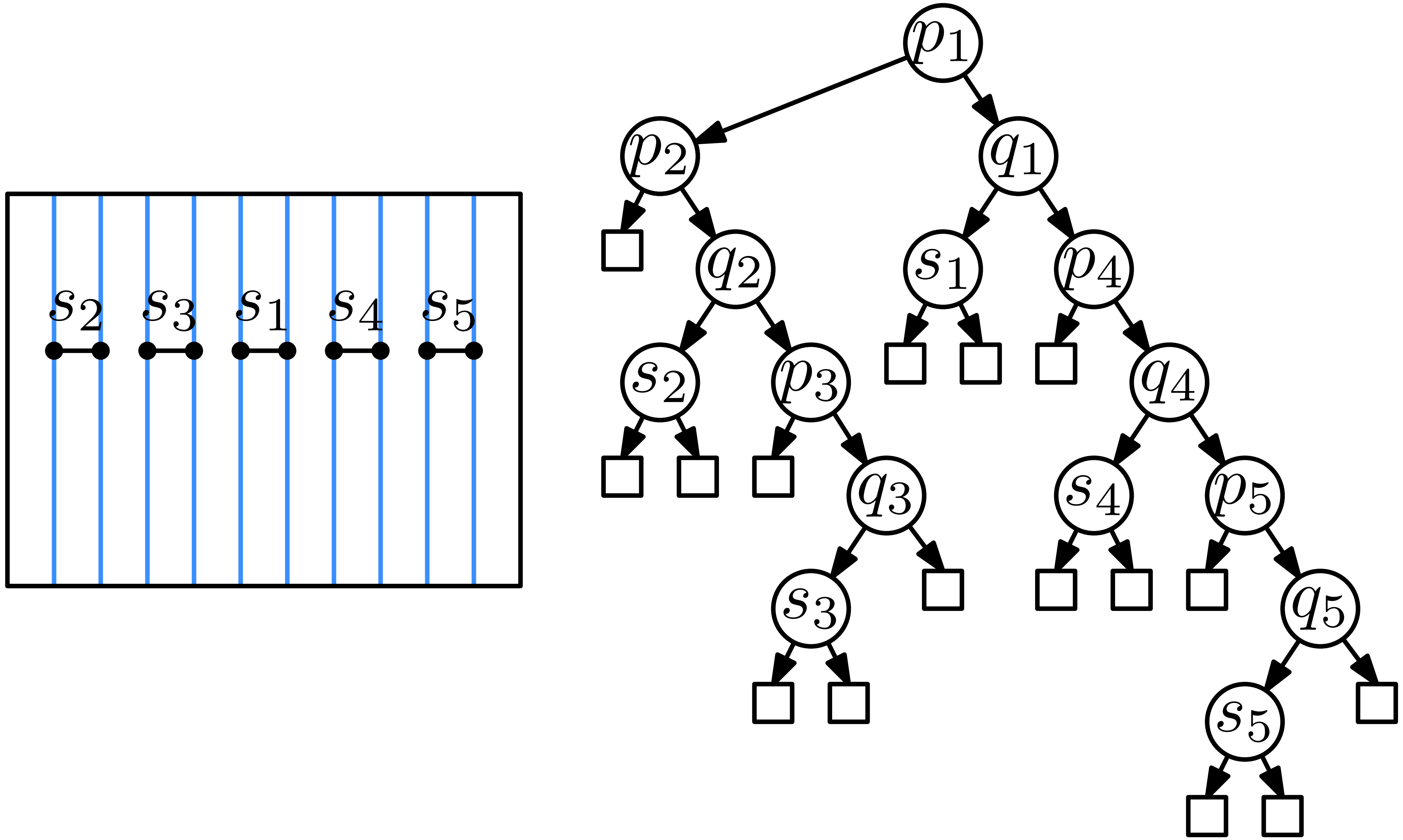
- set of line segments + order with long search path?
- set of line segments + order with many nodes?

Depth of Search Paths

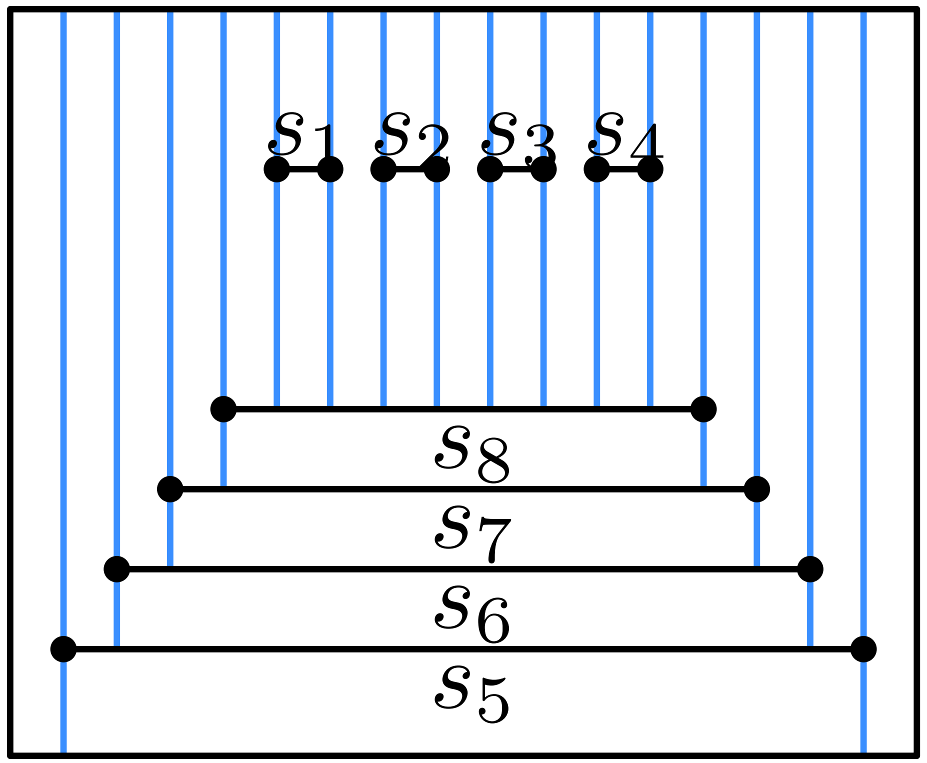




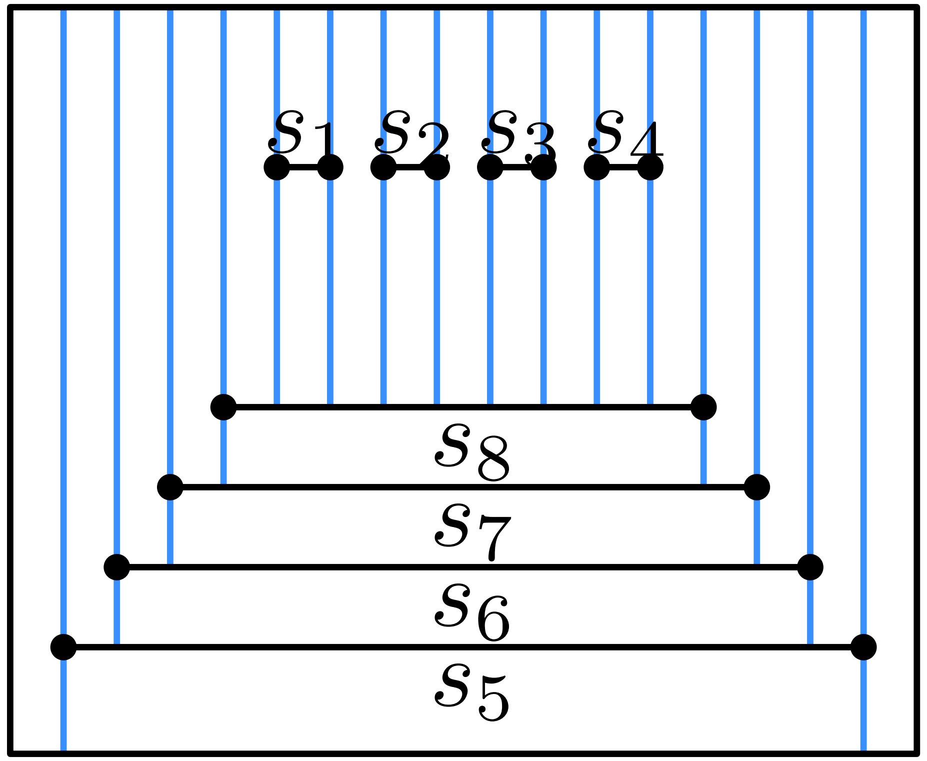
Depth of Search Paths

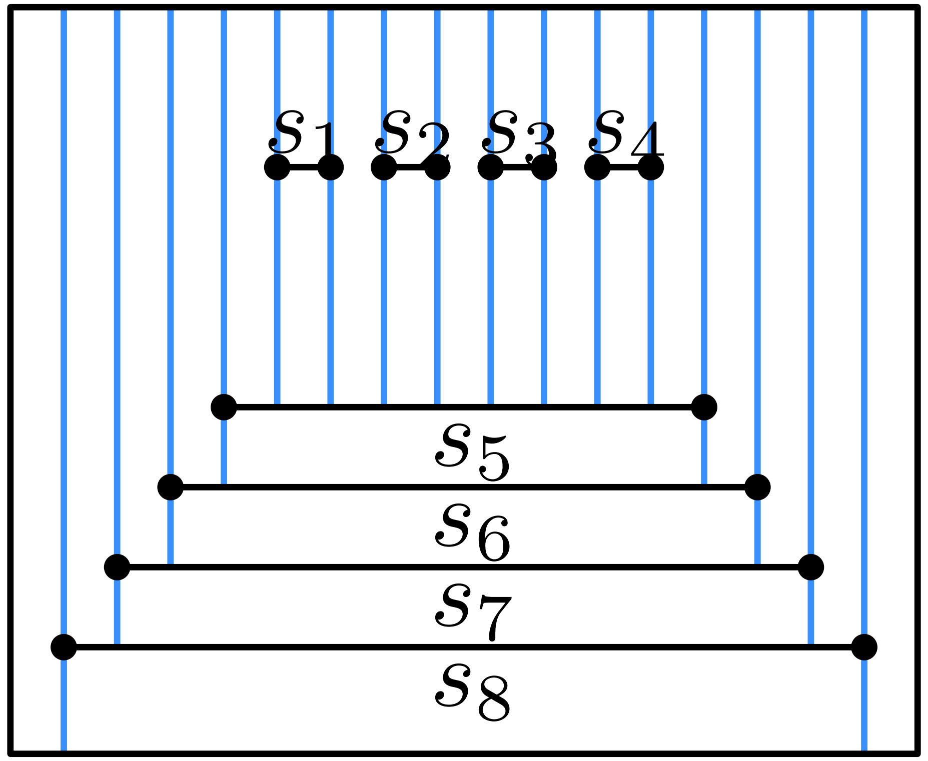


Number of Nodes



Number of Nodes





next: expected number of nodes and expected depth of search paths

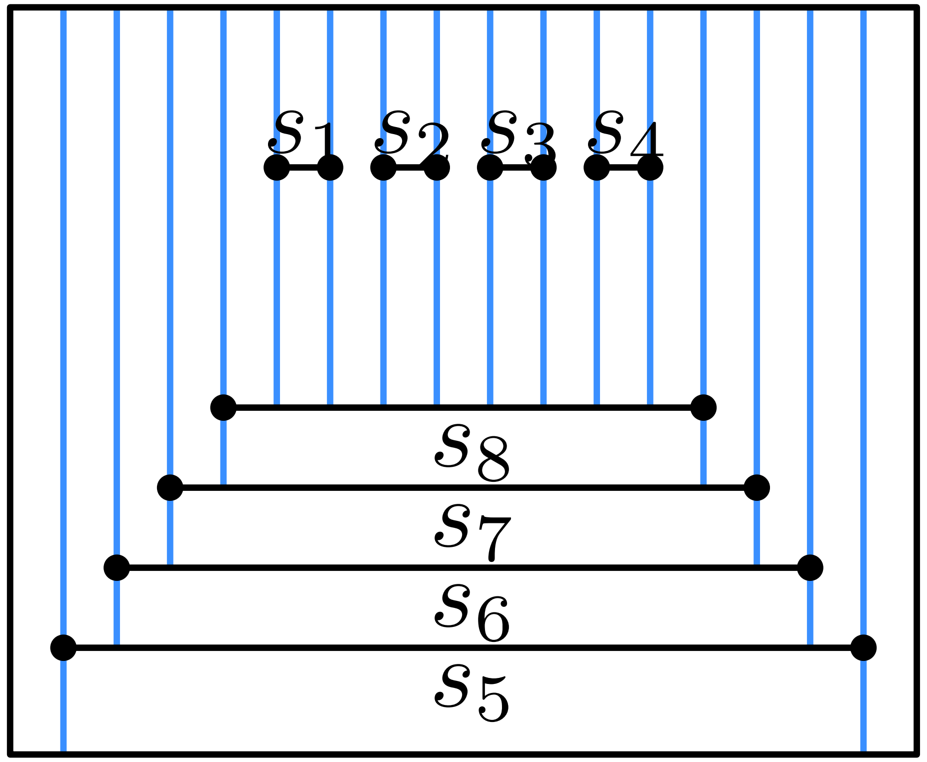
Vertical Decomposition for Point Location

Randomized Incremental Construction: Analysis

Storage of the structure

The vertical decomposition structure T always uses linear storage

The search structure D can use anything between linear and quadratic storage

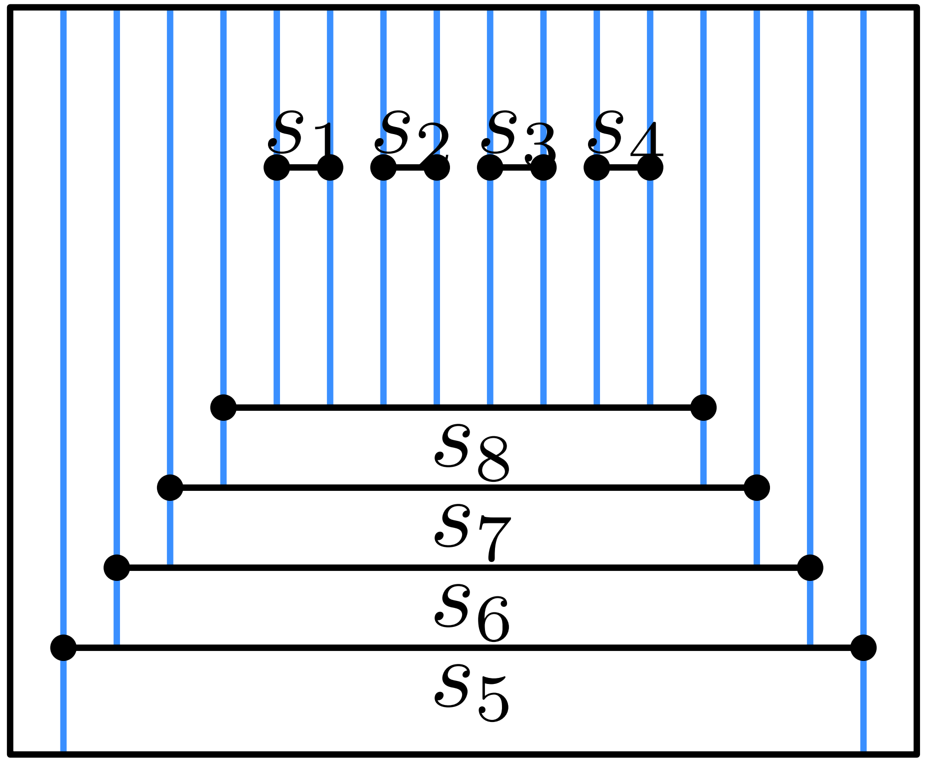


Storage of the structure

The vertical decomposition structure T always uses linear storage

The search structure D can use anything between linear and quadratic storage

We analyze the expected number of new nodes
when adding s_i, using backwards analysis



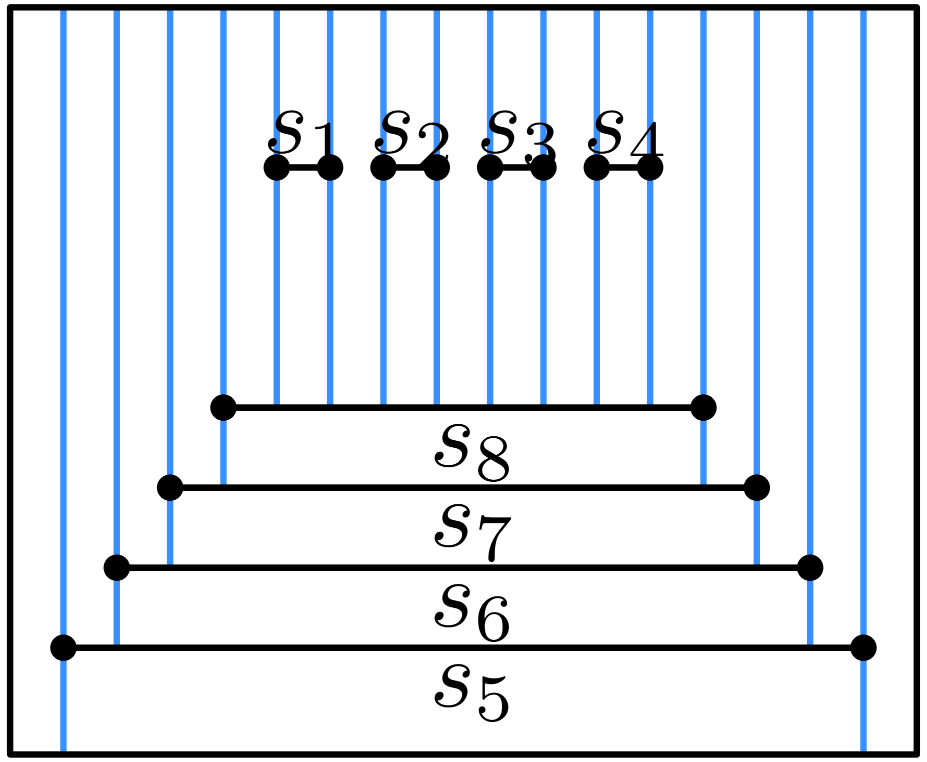
Storage of the structure

The vertical decomposition structure T always uses linear storage

The search structure D can use anything between linear and quadratic storage

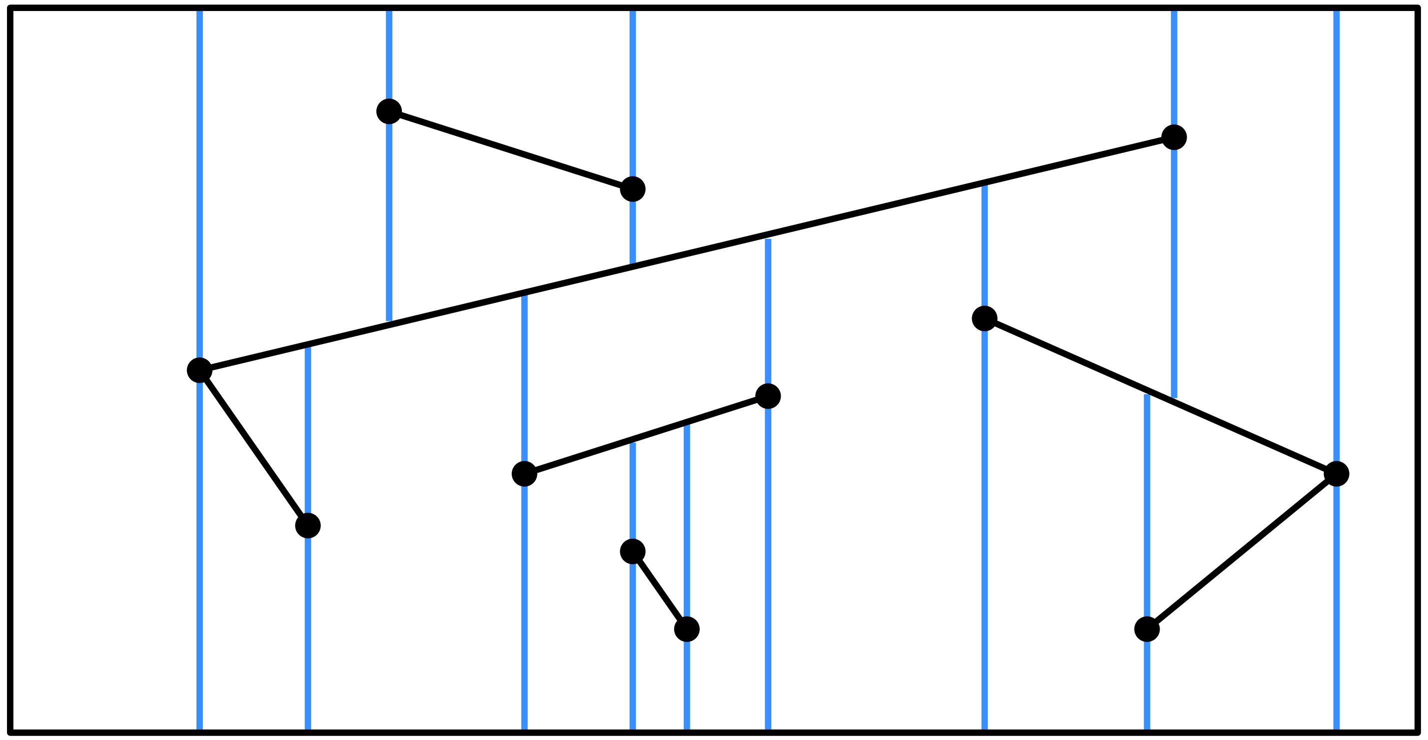
We analyze the expected number of new nodes when adding s_i, using backwards analysis

The number of created trapezoids is linear in the number of deleted trapezoids (leaves of D_{i-1}), or intersected trapezoids by s_i in T_{i-1}; this is linear in k_i



Storage of the structure

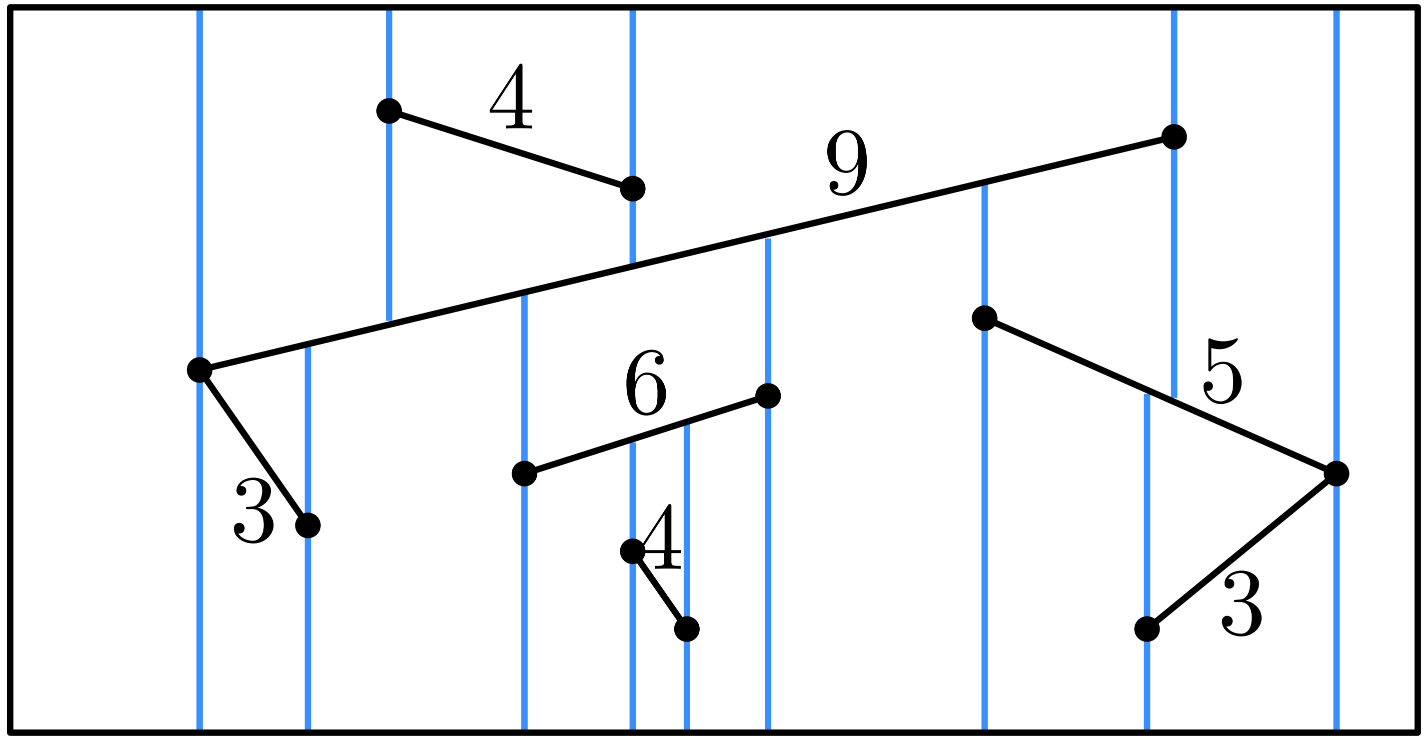
Backwards analysis in this case: Suppose we added s_i and have computed T_i and D_i. All line segments (existing in T_i) had the same probability of having been the last one added



Storage of the structure

Backwards analysis in this case: Suppose we added s_i and have computed T_i and D_i. All line segments (existing in T_i) had the same probability of having been the last one added

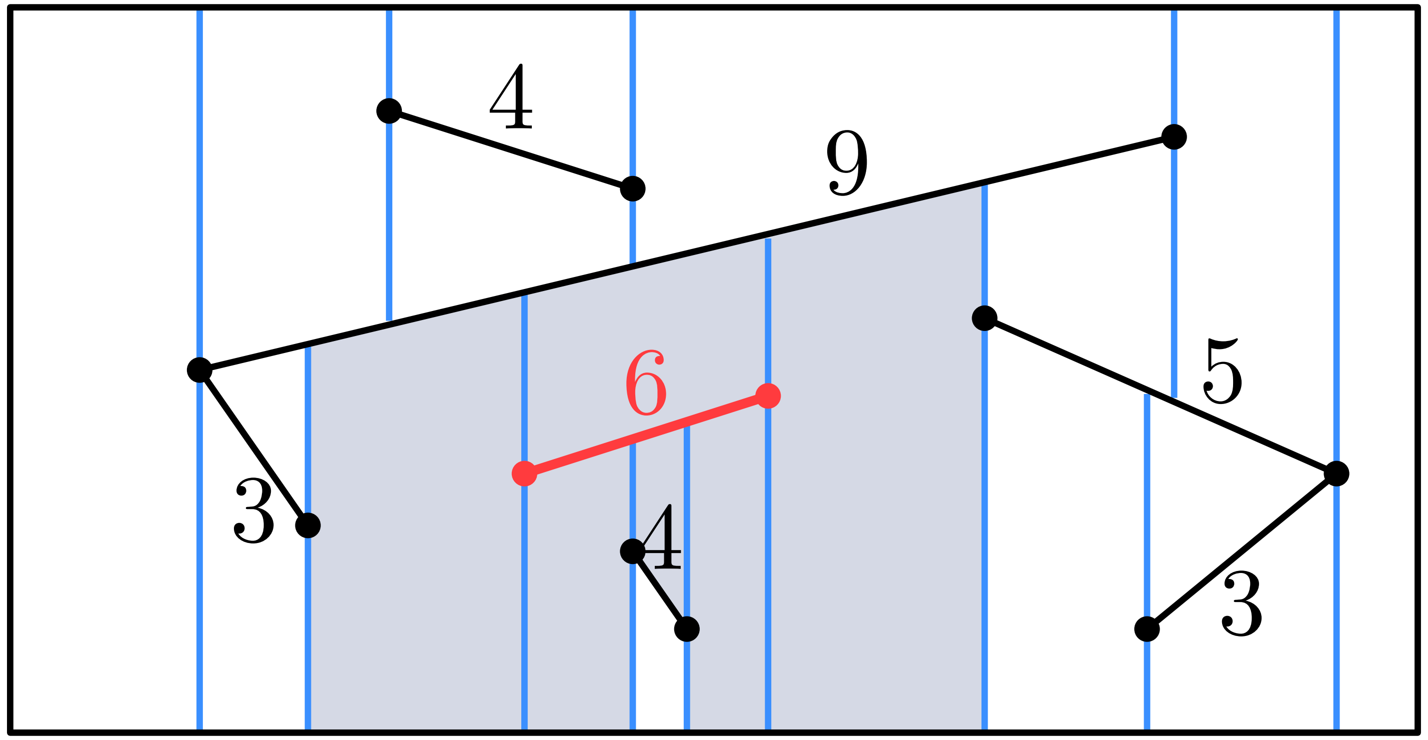
For each of the i line segments, we can see how many trapezoids would have been created if it were the last one added



Storage of the structure

Backwards analysis in this case: Suppose we added s_i and have computed T_i and D_i. All line segments (existing in T_i) had the same probability of having been the last one added

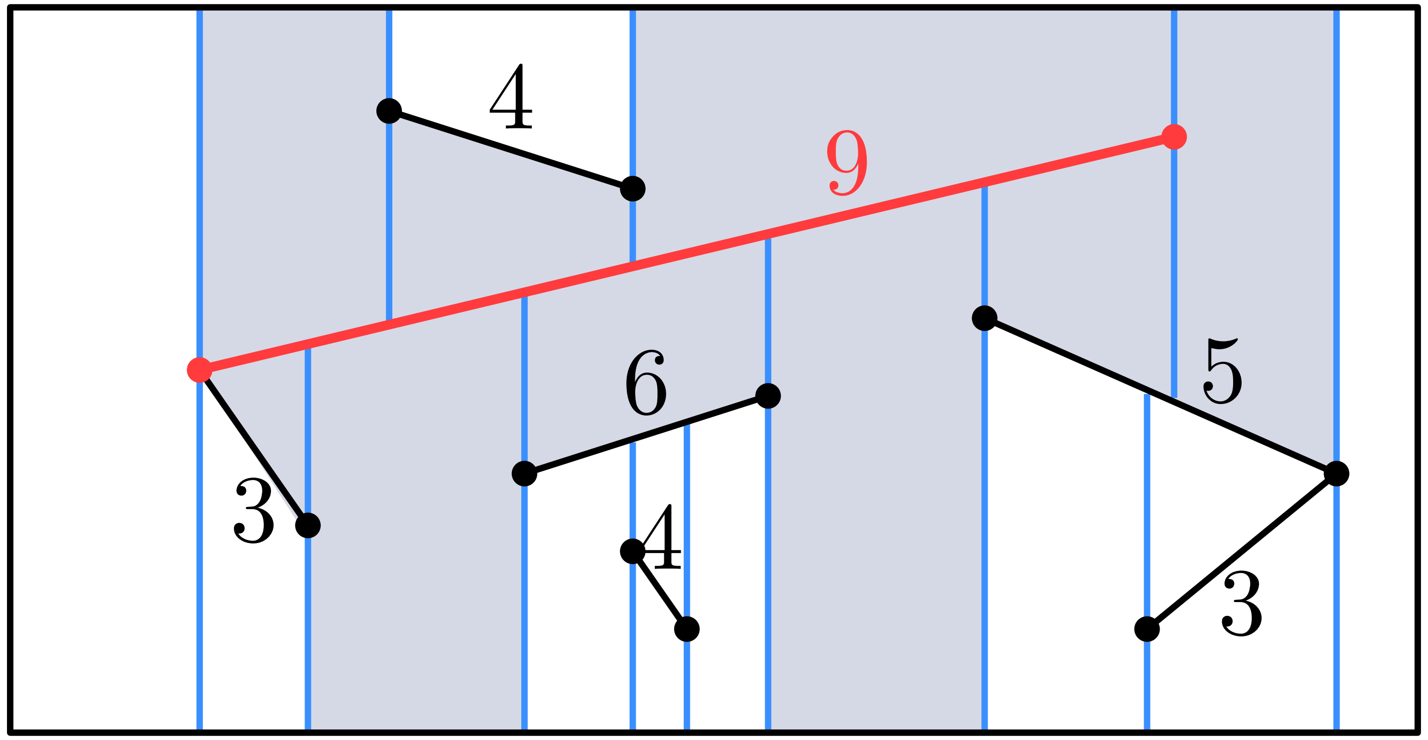
For each of the i line segments, we can see how many trapezoids would have been created if it were the last one added



Storage of the structure

Backwards analysis in this case: Suppose we added s_i and have computed T_i and D_i. All line segments (existing in T_i) had the same probability of having been the last one added

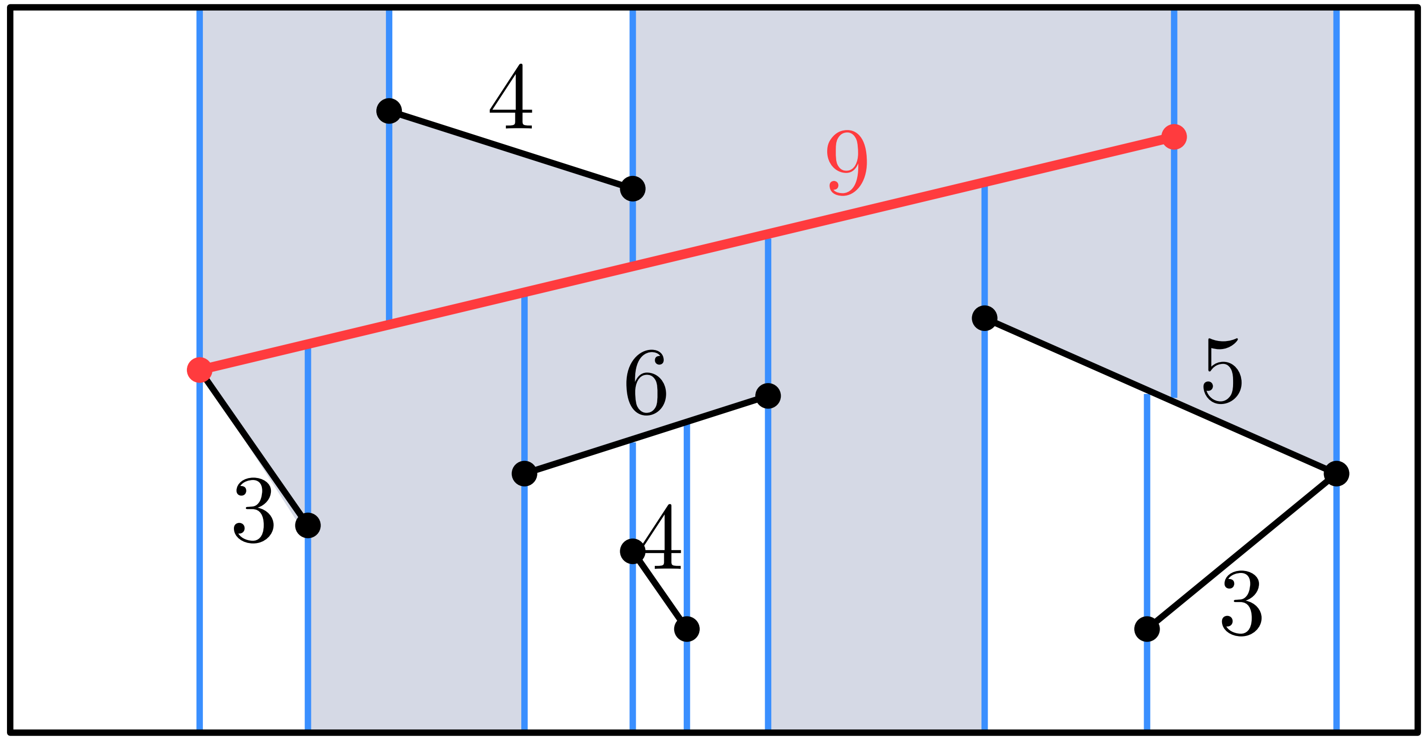
For each of the i line segments, we can see how many trapezoids would have been created if it were the last one added



Storage of the structure

Backwards analysis in this case: Suppose we added s_i and have computed T_i and D_i. All line segments (existing in T_i) had the same probability of having been the last one added

For each of the i line segments, we can see how many trapezoids would have been created if it were the last one added

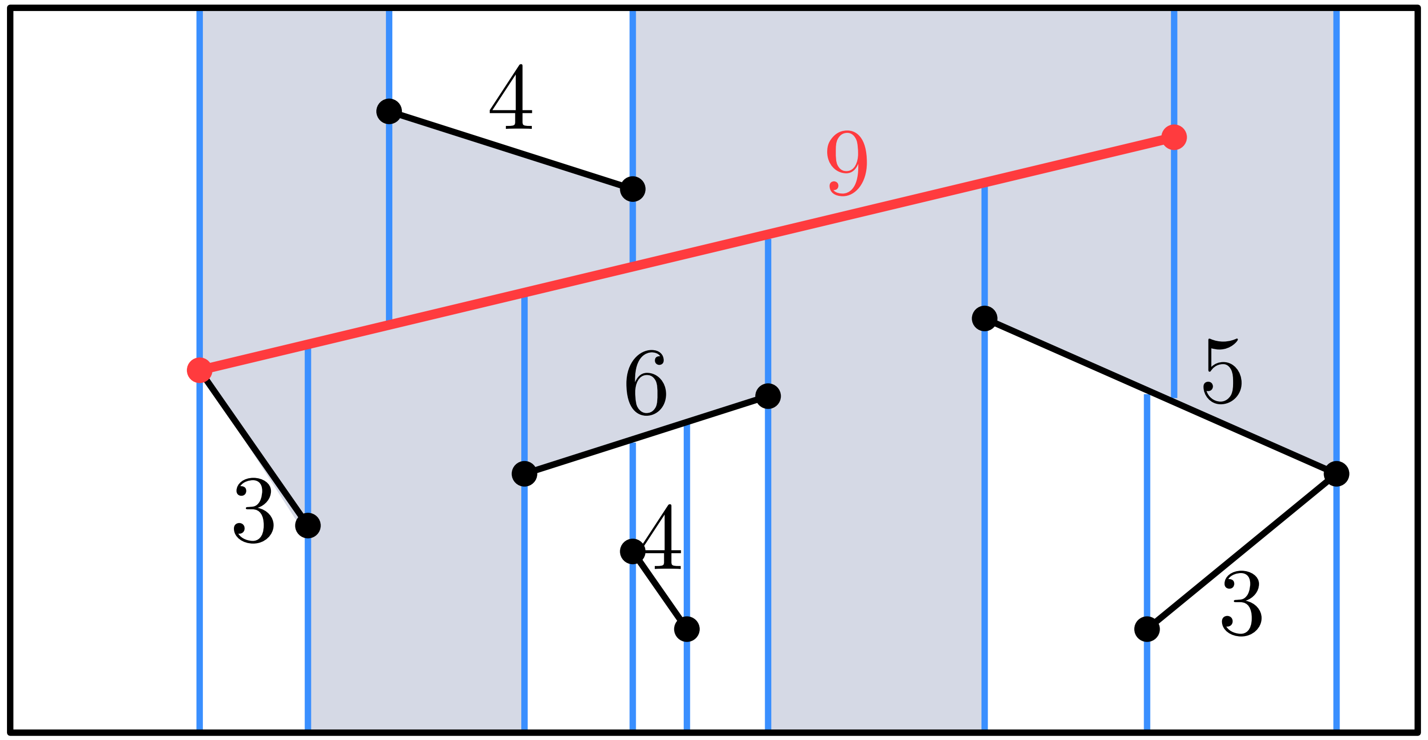


The average is the expected number of created trapezoids

Storage of the structure

Backwards analysis in this case: Suppose we added s_i and have computed T_i and D_i. All line segments (existing in T_i) had the same probability of having been the last one added

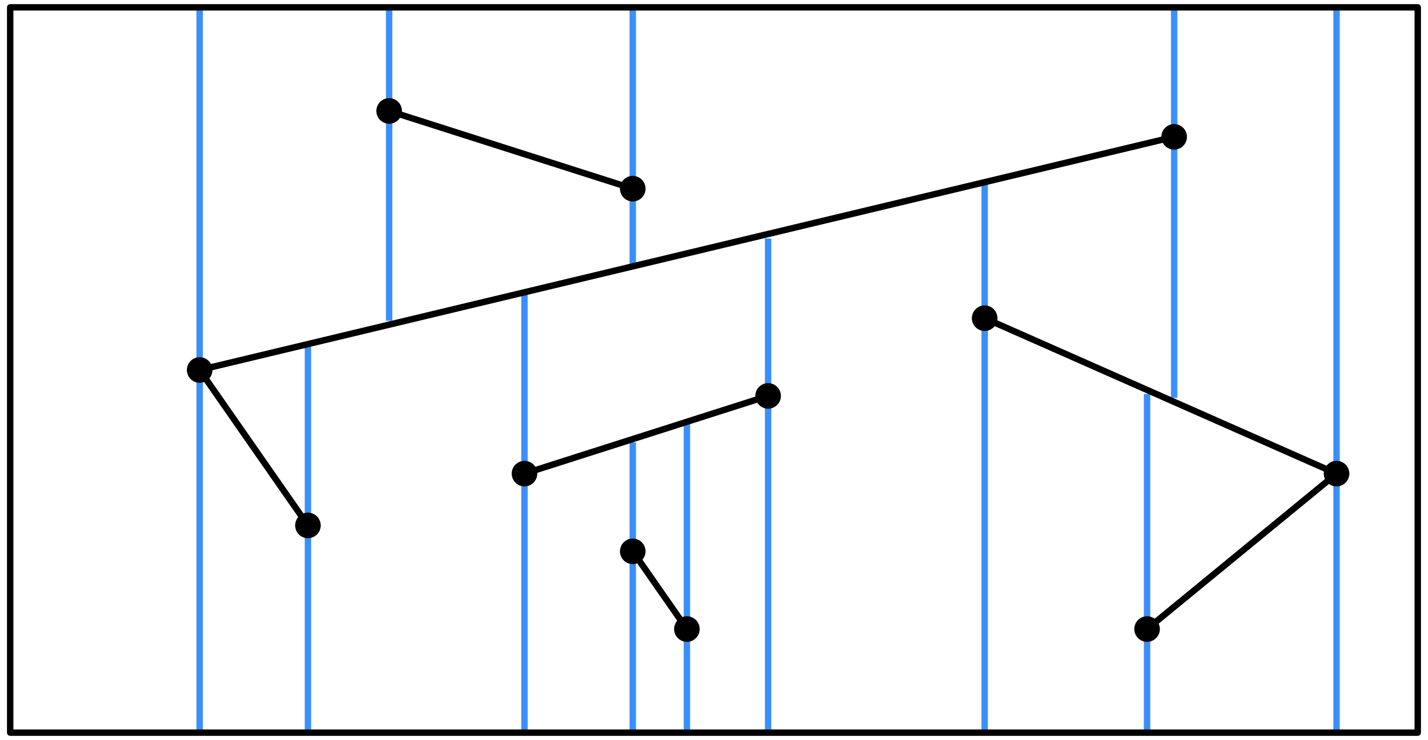
We will analyze



$$K_i = \sum_{j=1}^{i} [\text{no. of trapezoids created if } s_j \text{ were last}]$$

Storage of the structure

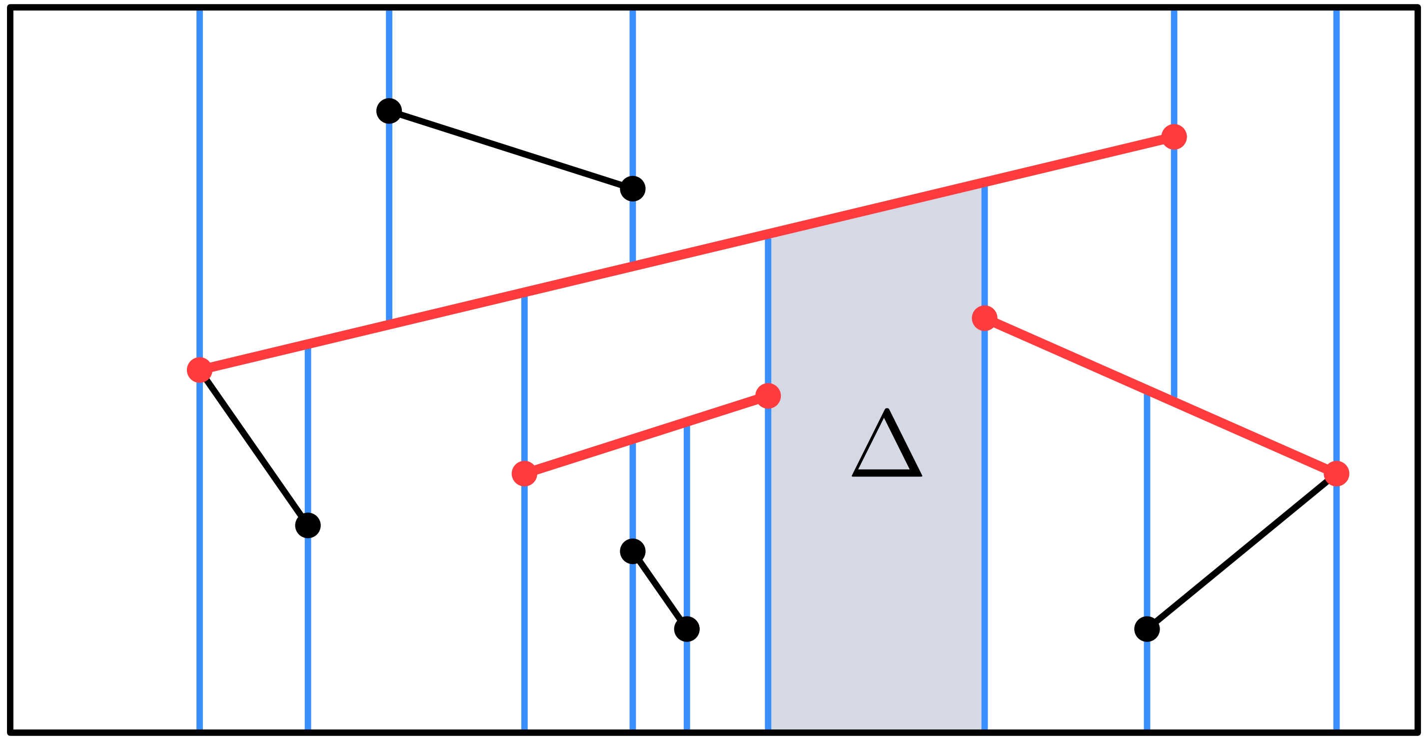
Consider K_i from the "trapezoid perspective": For any trapezoid Δ, there are at most four line segments whose insertion would have created it (top(Δ), bottom(Δ), leftp(Δ), and rightp(Δ))



$$K_i = \sum_{j=1}^{i} \left[\text{no. of trapezoids created if } s_j \text{ were last}\right]$$

Storage of the structure

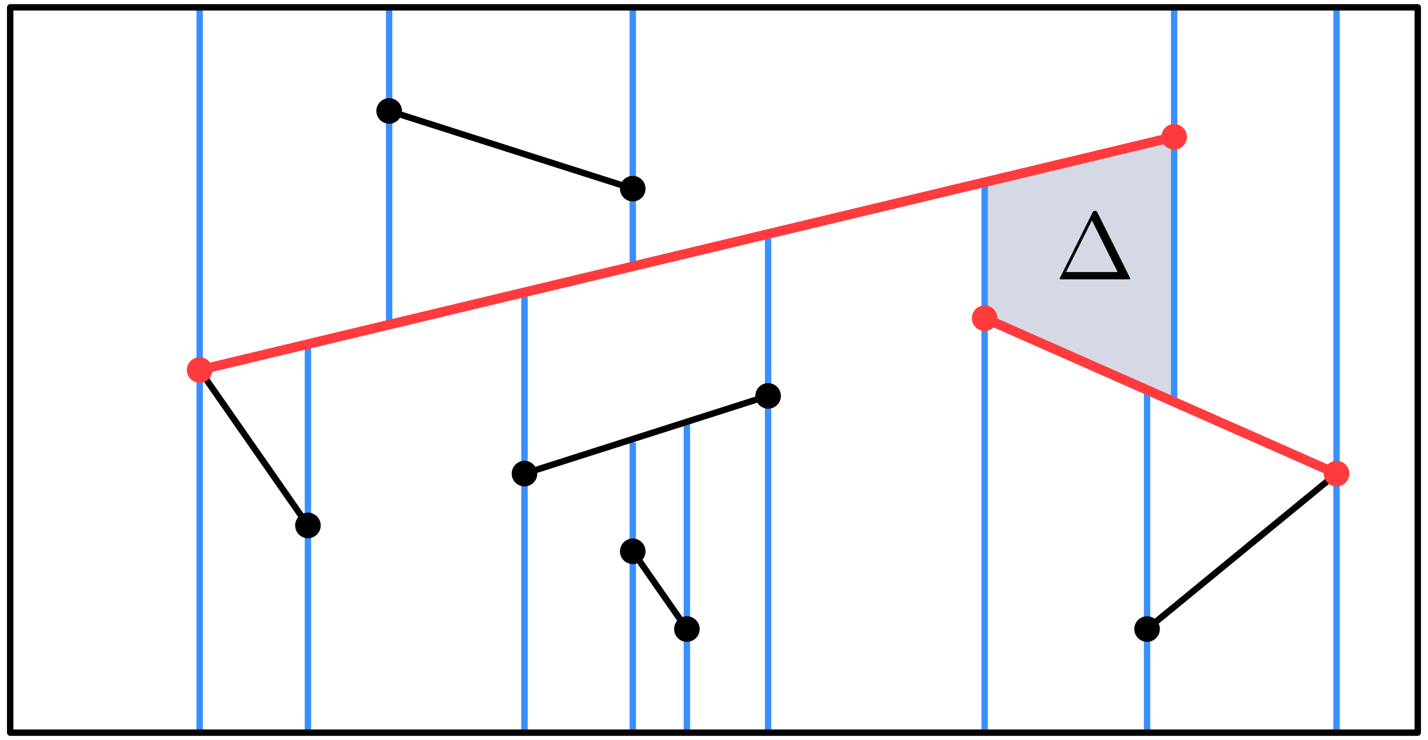
Consider K_i from the "trapezoid perspective": For any trapezoid Δ, there are at most four line segments whose insertion would have created it (top(Δ), bottom(Δ), leftp(Δ), and rightp(Δ))



$$K_i = \sum_{j=1}^{i} [\text{no. of trapezoids created if } s_j \text{ were last}]$$

Storage of the structure

Consider K_i from the "trapezoid perspective": For any trapezoid Δ, there are at most four line segments whose insertion would have created it (top(Δ), bottom(Δ), leftp(Δ), and rightp(Δ))



$$K_i = \sum_{j=1}^{i} [\text{no. of trapezoids created if } s_j \text{ were last}]$$

Storage of the structure

Consider K_i from the "trapezoid perspective": For any trapezoid Δ, there are at most four line segments whose insertion would have created it (top(Δ), bottom(Δ), leftp(Δ), and rightp(Δ))

Hence,

$$K_i = \sum_{j=1}^{i} \left[\text{no. of trapezoids created if } s_j \text{ were last}\right]$$

Storage of the structure

Consider K_i from the "trapezoid perspective": For any trapezoid Δ, there are at most four line segments whose insertion would have created it (top(Δ), bottom(Δ), leftp(Δ), and rightp(Δ))

Hence,

$$\begin{aligned} K_i &= \sum_{j=1}^{i} [\text{no. of trapezoids created if } s_j \text{ were last}] \\ &= \sum_{\Delta \in T_i} [\text{no. of segments that would create } \Delta] \\ &\leq \sum 4 = 12i + 4 \end{aligned}$$

since there are at most $3i + 1$ trapezoids in a vertical decomposition of i line segments in R

Storage of the structure

$$K_i \leq (12i + 4)$$

Storage of the structure

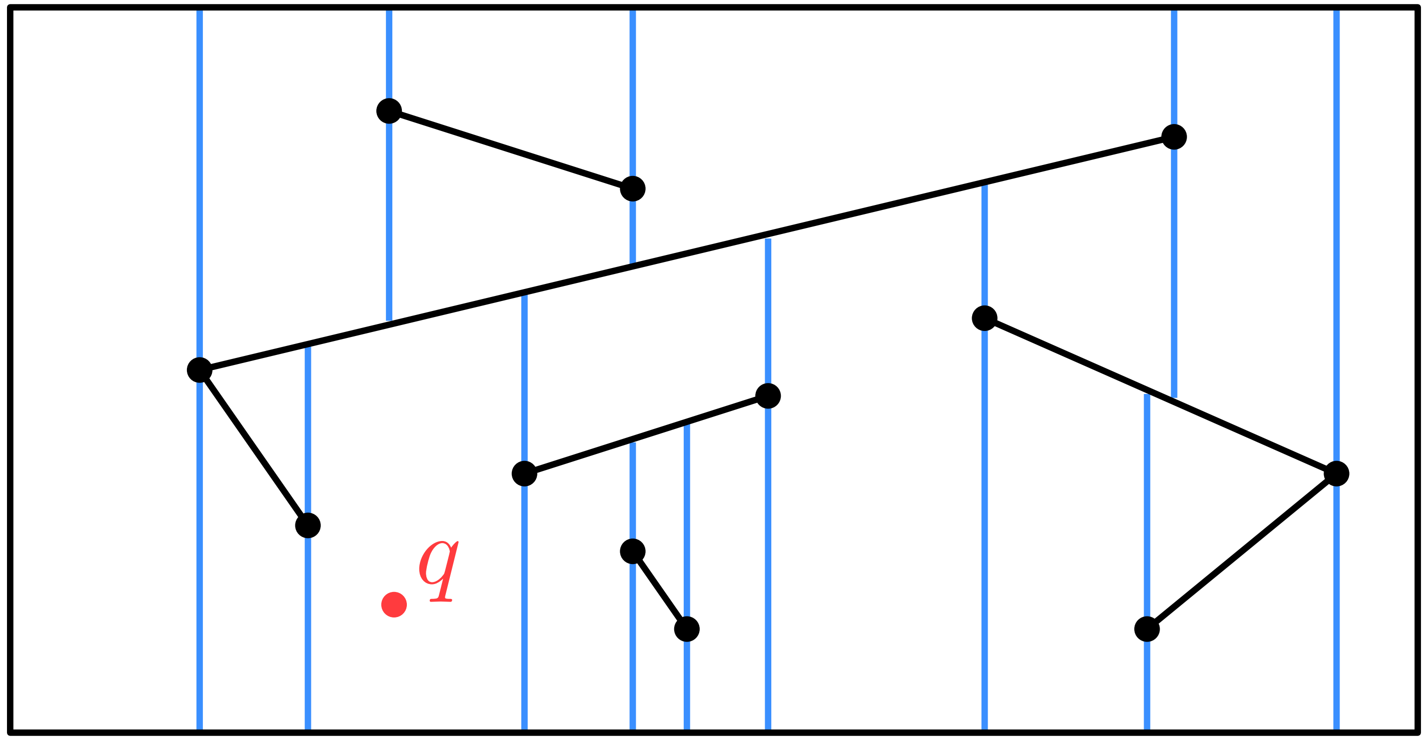
$K_i \leq (12i + 4)$

Since K_i is defined as a sum over i line segments, the average number of trapezoids in T_i created by s_i is at most $(12i + 4)/i \leq 13$

Since the expected number of new nodes is at most 13 in every step, the expected size of the structure after adding n line segments is $O(n)$

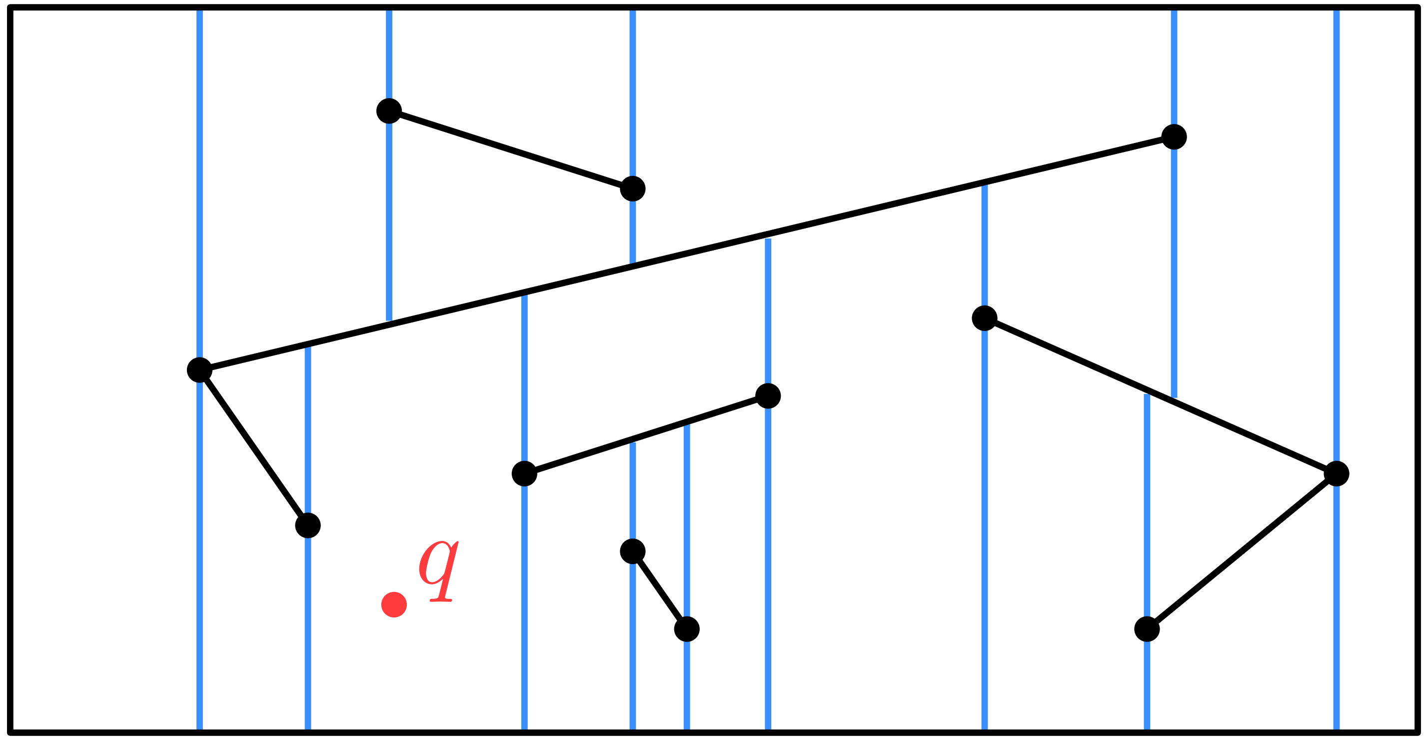
Query time of the structure

Fix any point q in the plane as a query point, we will analyze the probability that inserting s_i makes the search path to q longer



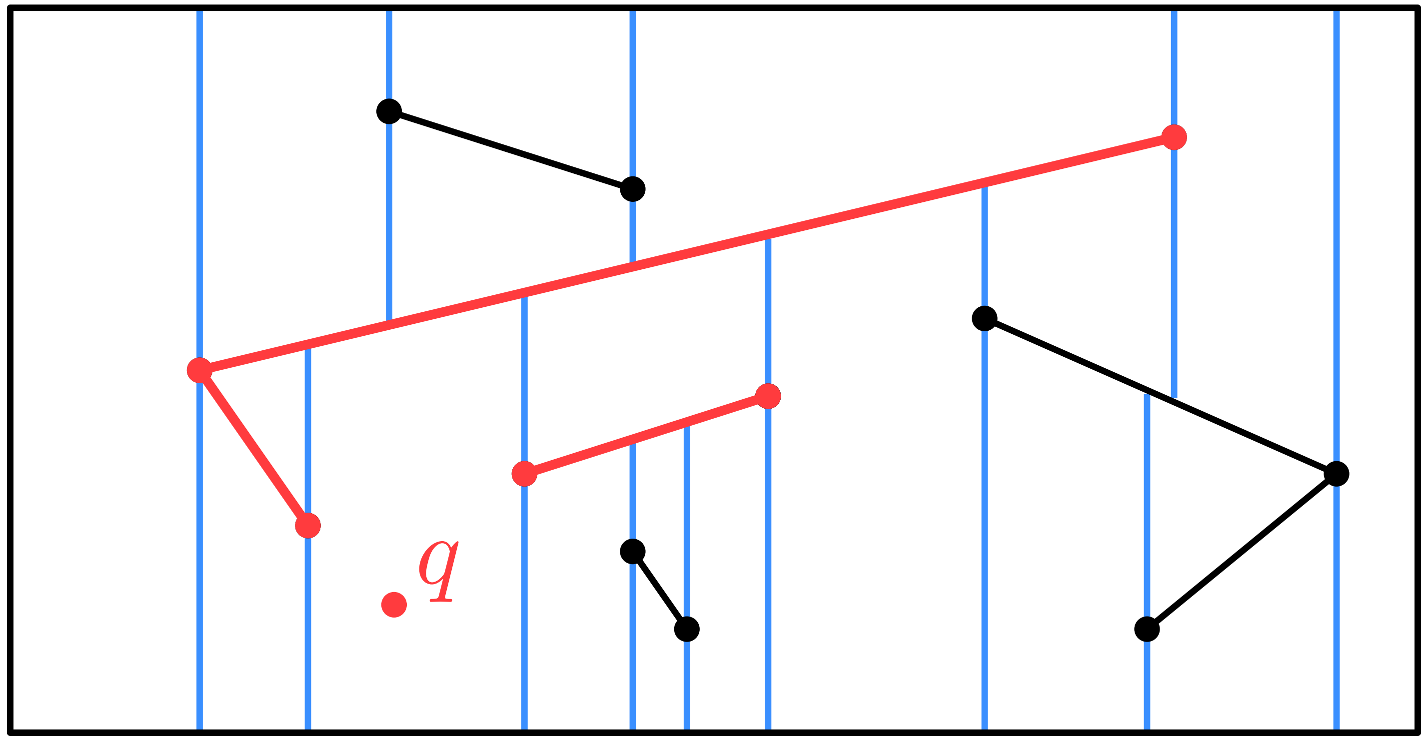
Query time of the structure

Backwards analysis: Take the situation after s_i has been added, and ask the question: How many of the i line segments made the search path to q longer?



Query time of the structure

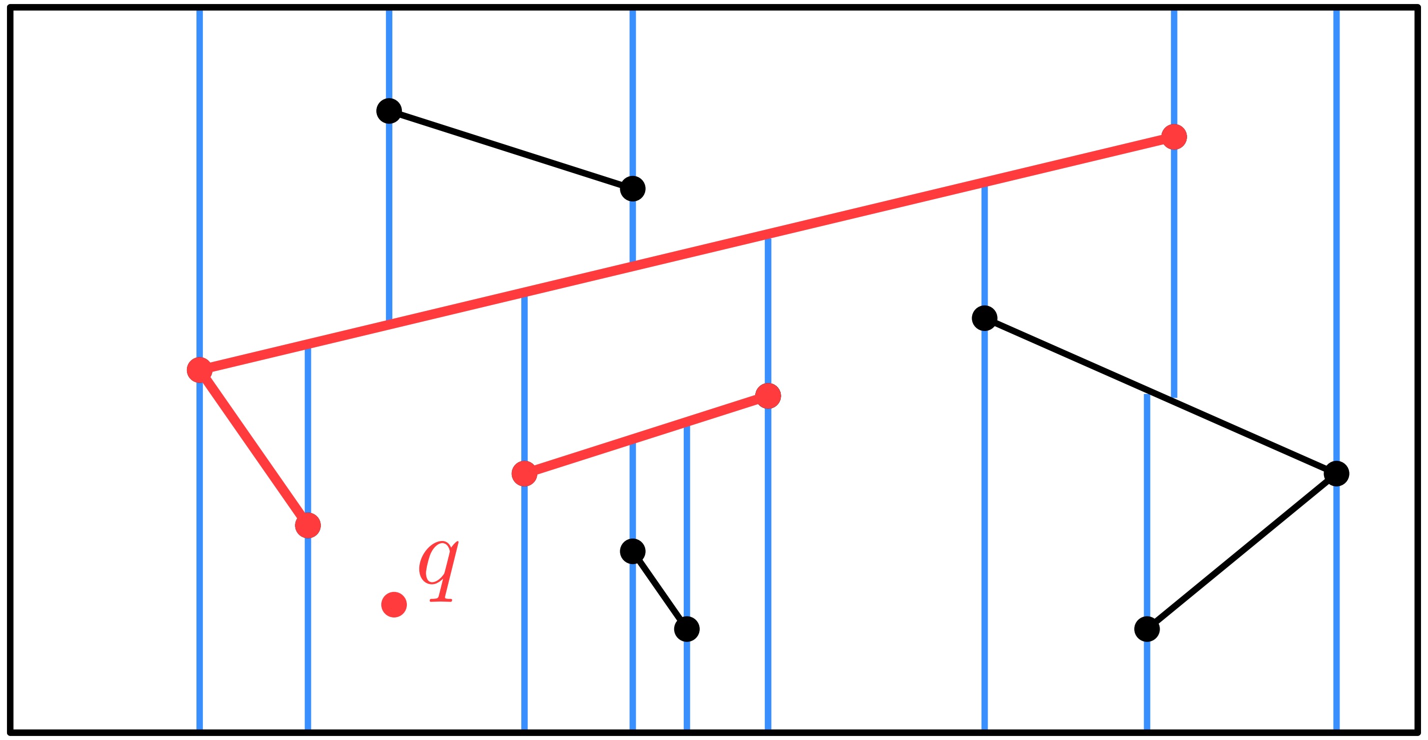
Backwards analysis: Take the situation after s_i has been added, and ask the question: How many of the i line segments made the search path to q longer?



Query time of the structure

Backwards analysis: Take the situation after s_i has been added, and ask the question: How many of the i line segments made the search path to q longer?

The search path to q only became longer if q is in a trapezoid that was just created by the latest insertion!

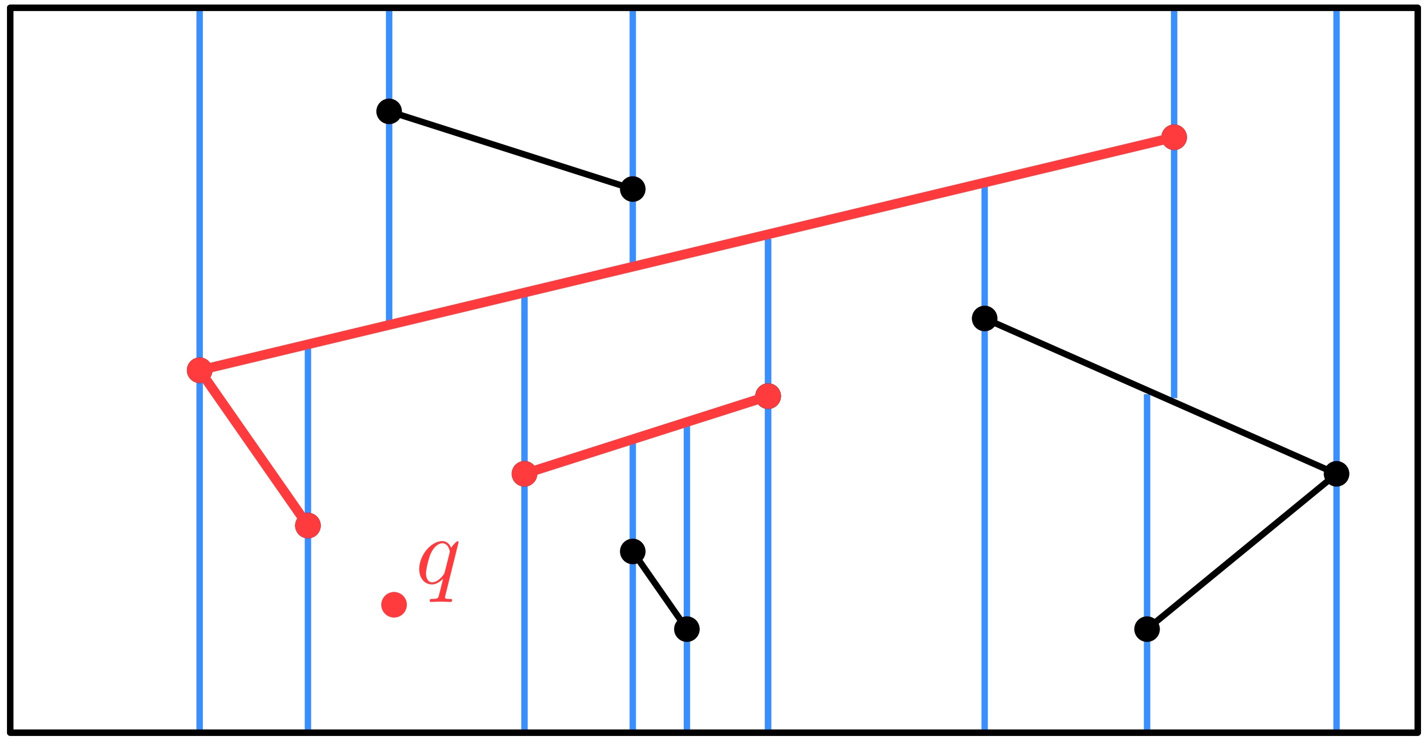


At most four line segments define the trapezoid that contains q, so the probability is $4/i$

Query time of the structure

We analyze

$$\sum_{i=1}^{n} Pr[\text{search path became longer due to } i\text{-th addition}]$$

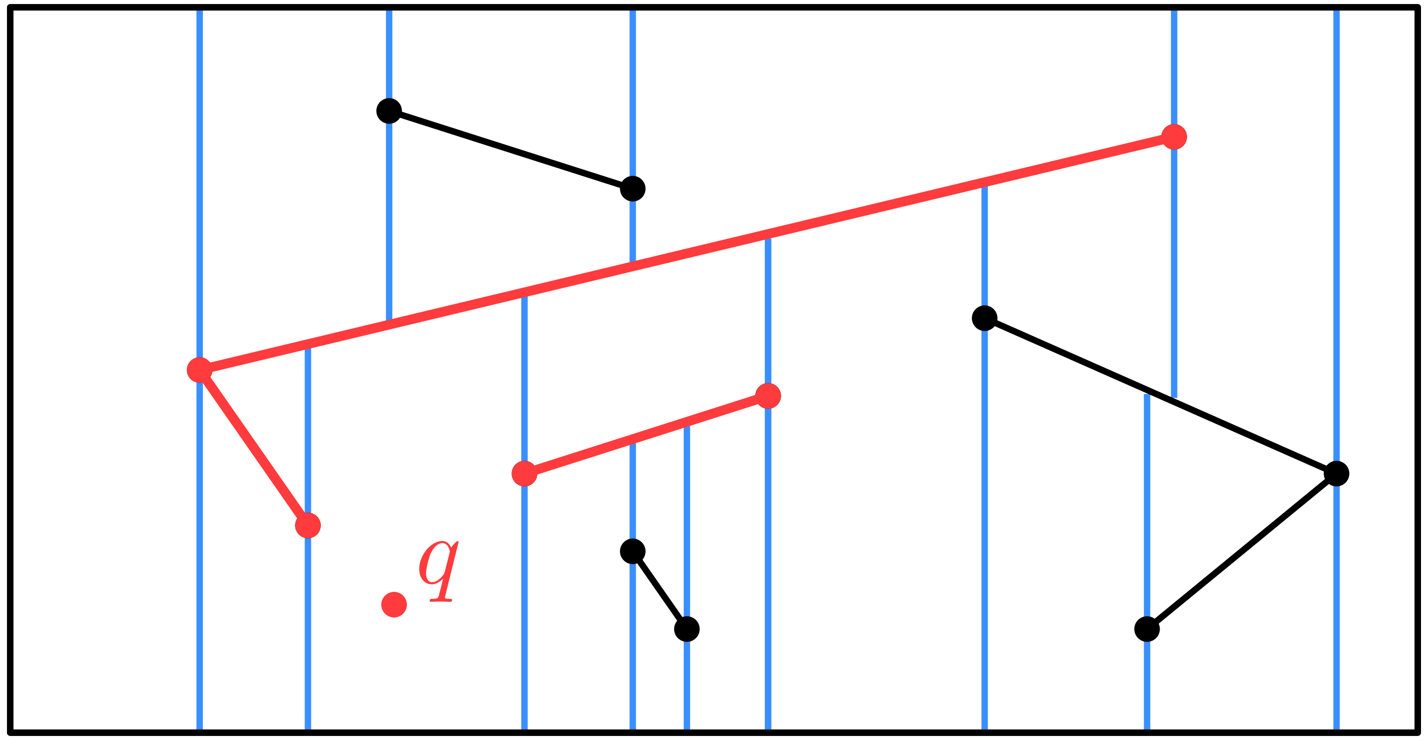


Query time of the structure

We analyze

$$\sum_{i=1}^{n} Pr[\text{search path became longer due to } i\text{-th addition}]$$

$$\leq \sum_{i=1}^{n} 4/i = 4 \cdot \sum_{i=1}^{n} 1/i \leq 4(1 + \log_e n)$$

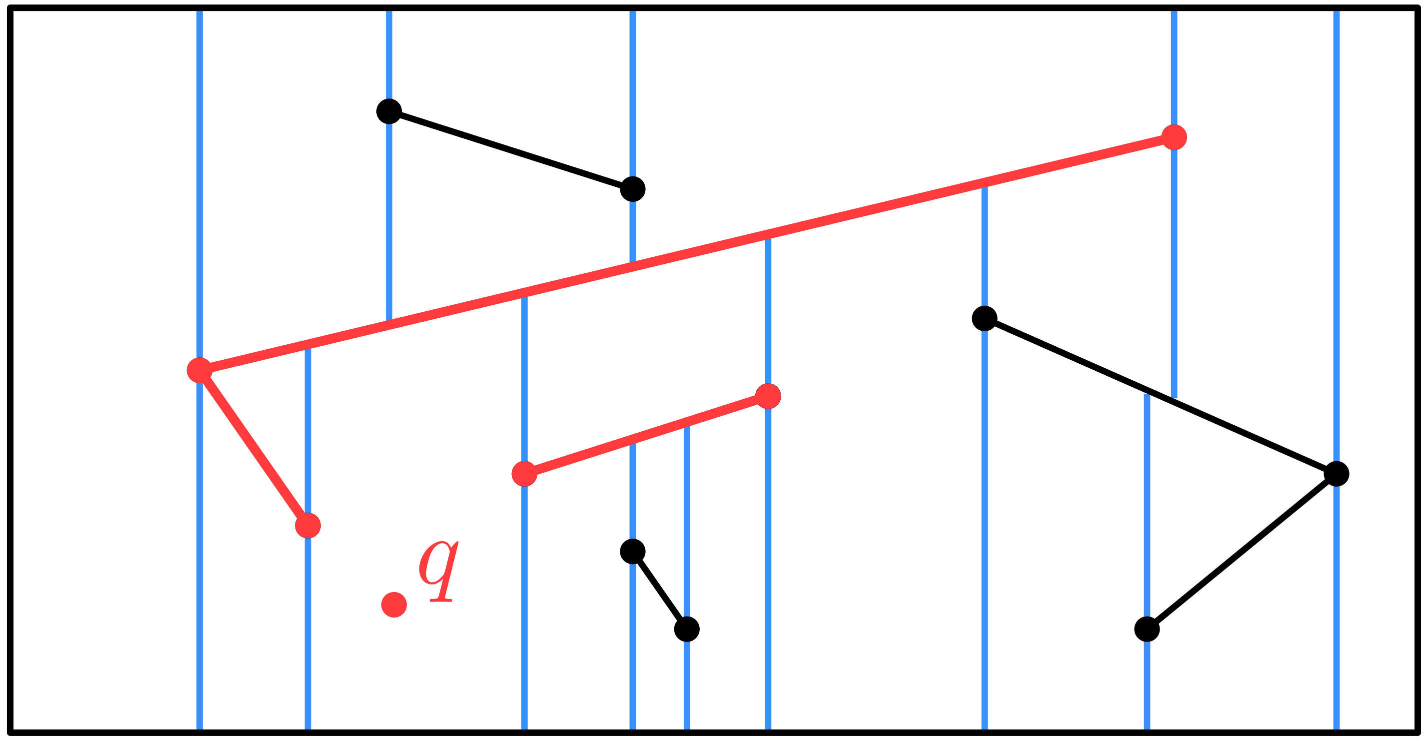


Query time of the structure

We analyze

$$\sum_{i=1}^{n} Pr[\text{search path became longer due to } i\text{-th addition}]$$

$$\leq \sum_{i=1}^{n} 4/i = 4 \cdot \sum_{i=1}^{n} 1/i \leq 4(1 + \log_e n)$$



So the expected query time is $O(\log n)$

Result, so far

Theorem: Given a planar subdivision defined by a set of n non-crossing line segments (where any two distinct segment endpoints have different x-coordinates) in the plane, we can preprocess it for planar point location queries in $O(n \log n)$ expected time, the structure uses $O(n)$ expected storage, and the expected query time is $O(\log n)$.

Vertical Decomposition for Point Location

Wrapping up

Degenerate cases

Assumption, so far

- unique x-coordinates
- left/right queries are answered correctly

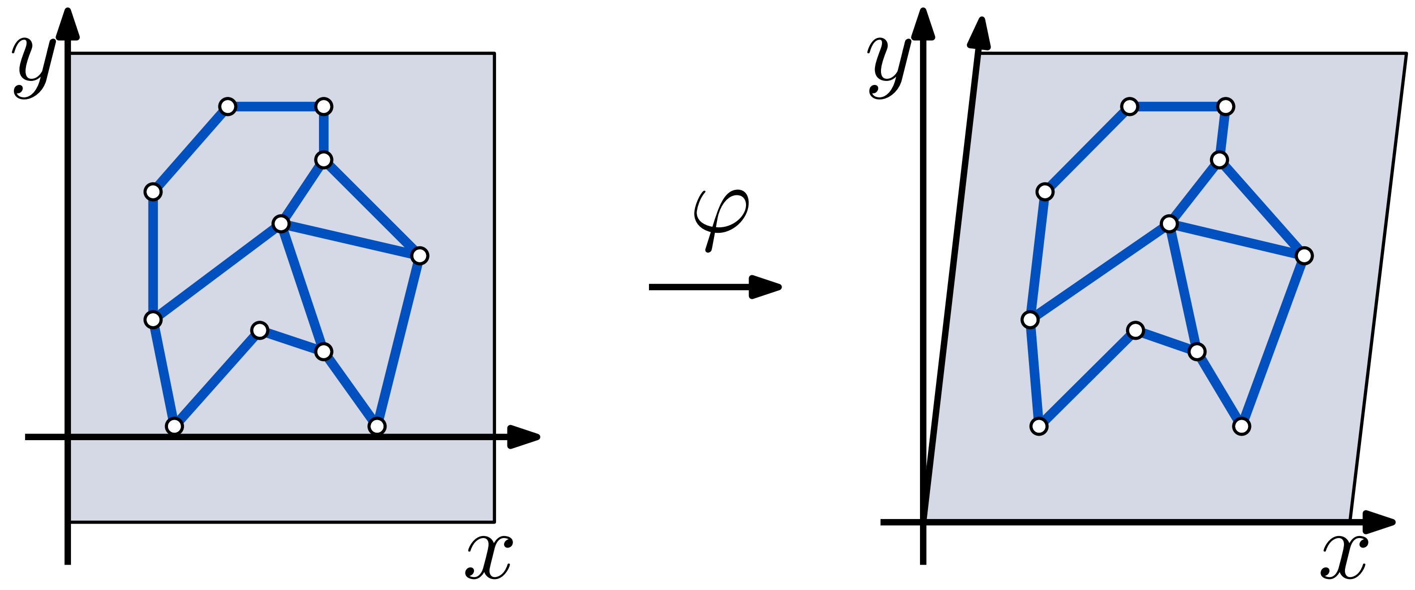
Degenerate cases

Assumption, so far

- unique x-coordinates
- left/right queries are answered correctly

Solution: symbolic shearing transform
$\varphi\colon (x,y) \mapsto (x+\varepsilon y, y)$

pick $\varepsilon > 0$ small, such that x-order $<$ of points does not change.



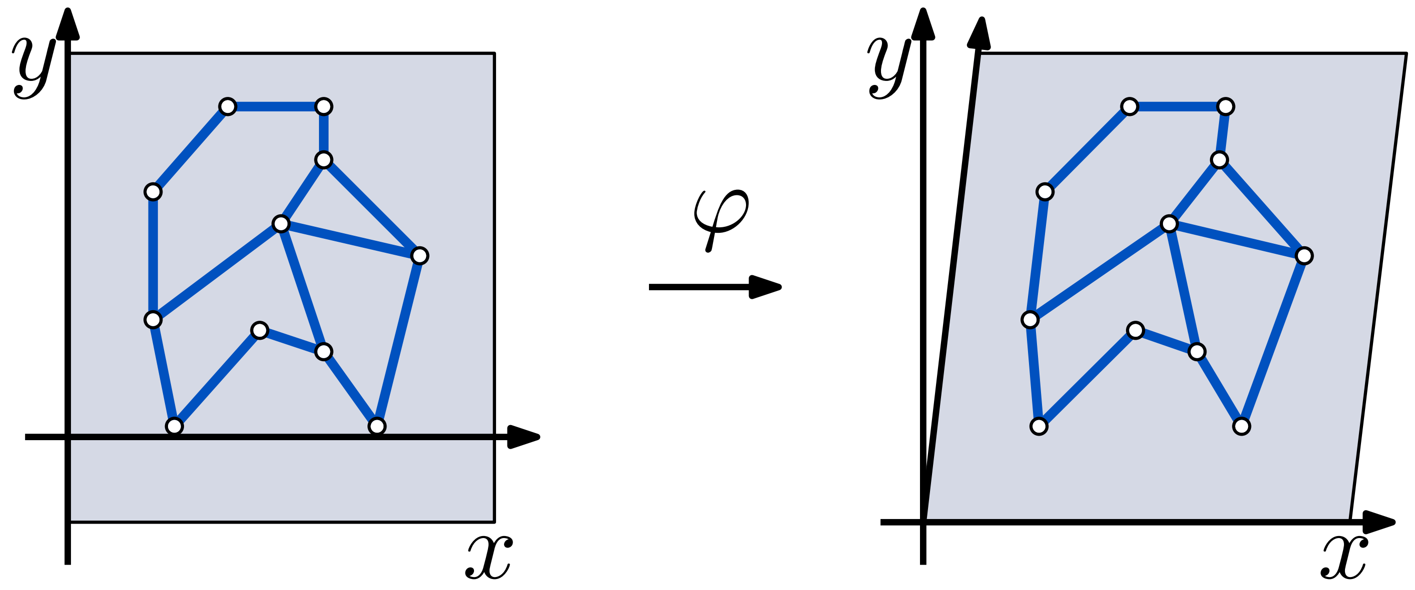
Degenerate cases

Assumption, so far

- unique x-coordinates
- left/right queries are answered correctly

Solution: symbolic shearing transform
$\varphi\colon (x, y) \mapsto (x + \varepsilon y, y)$

pick $\varepsilon > 0$ small, such that x-order $<$ of points does not change.



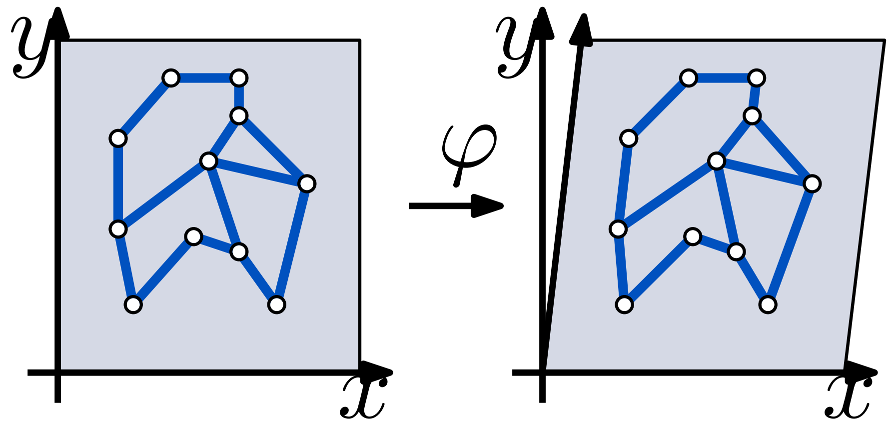
Symbolic: Don’t actually apply φ, but execute algorithm as if it was

Degenerate cases

Effect of shear transform φ:

Effect 1: lexicographical order

Effect 2: point-line incidences are maintained since φ is affine

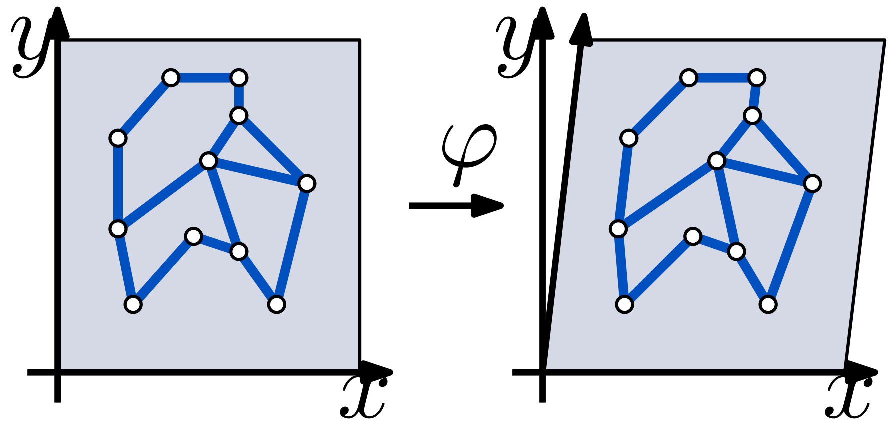


Degenerate cases

Effect of shear transform φ:

Effect 1: lexicographical order

Effect 2: point-line incidences are maintained since φ is affine



Algorithm uses two elementary operations/tests:

Test 1: Is q left or right of vertical line through p?

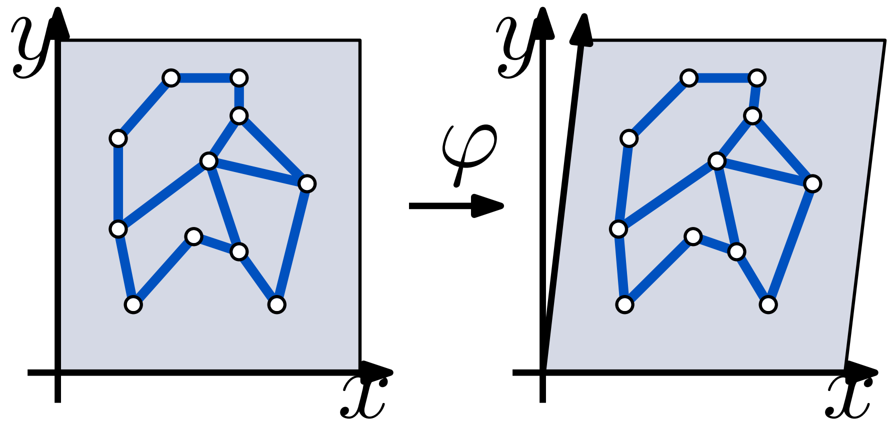
Test 2: Is q above or below segment s?

Degenerate cases

Effect of shear transform φ:

Effect 1: lexicographical order

Effect 2: point-line incidences are maintained since φ is affine



Algorithm uses two elementary operations/tests:

Test 1: Is q left or right of vertical line through p?

Test 2: Is q above or below segment s?

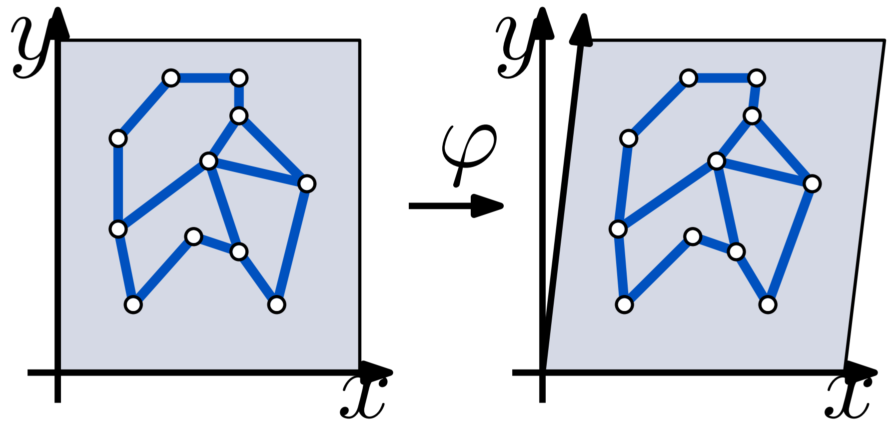
Effect 1 $\Rightarrow$ Use lexicographical order on S to get Test 1 correct for sheared set

Degenerate cases

Effect of shear transform φ:

Effect 1: lexicographical order

Effect 2: point-line incidences are maintained since φ is affine



Algorithm uses two elementary operations/tests:

Test 1: Is q left or right of vertical line through p?

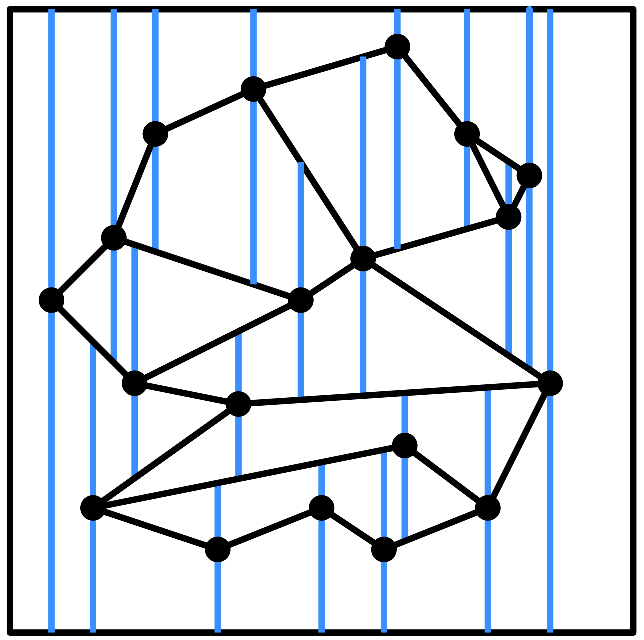
Test 2: Is q above or below segment s?

Effect 1 $\Rightarrow$ Use lexicographical order on S to get Test 1 correct for sheared set

Effect 2 $\Rightarrow$ Test 2 is equivalent on original and sheared segments

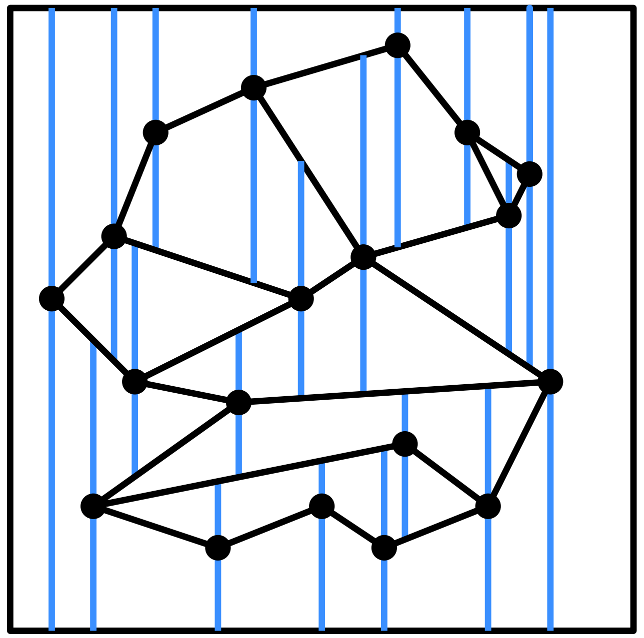
Result

Theorem: Given a planar subdivision defined by a set of n non-crossing line segments in the plane, we can preprocess it for planar point location queries in $O(n \log n)$ expected time, the structure uses $O(n)$ expected storage, and the expected query time is $O(\log n)$.



Result

Theorem: Given a planar subdivision defined by a set of n non-crossing line segments in the plane, we can preprocess it for planar point location queries in $O(n \log n)$ expected time, the structure uses $O(n)$ expected storage, and the expected query time is $O(\log n)$.



What have we seen?

Algorithmic Problem: Point location

Data Structure: Vertical Decomposition with Search Graph

Technique: Randomized Incremental Construction

Analysis: Backwards Analysis