POINT LOCATION IN PLANAR SUBDIVISIONS

Vera Sacristán

Rodrigo Silveira

Research Group on
Discrete, Combinatorial and Computational Geometry
Universitat Politècnica de Catalunya

Introduction

POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

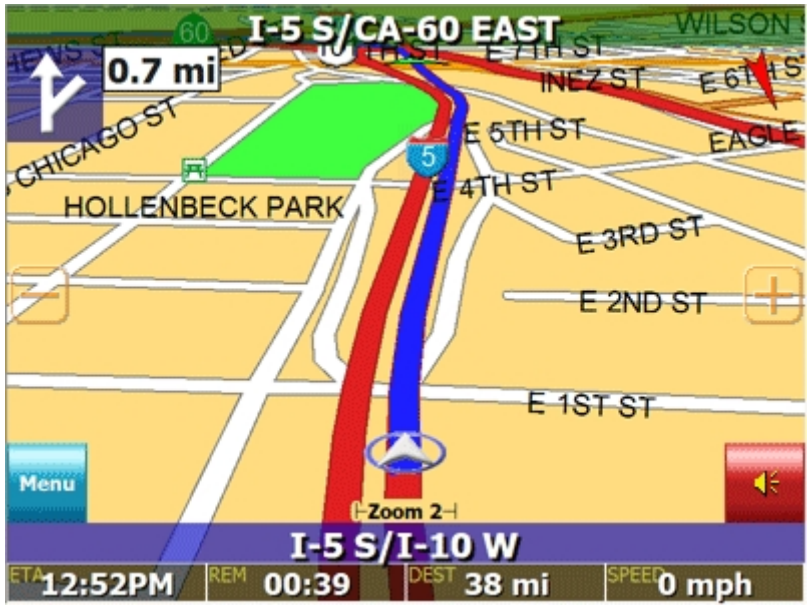
Applications

POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

Applications



POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

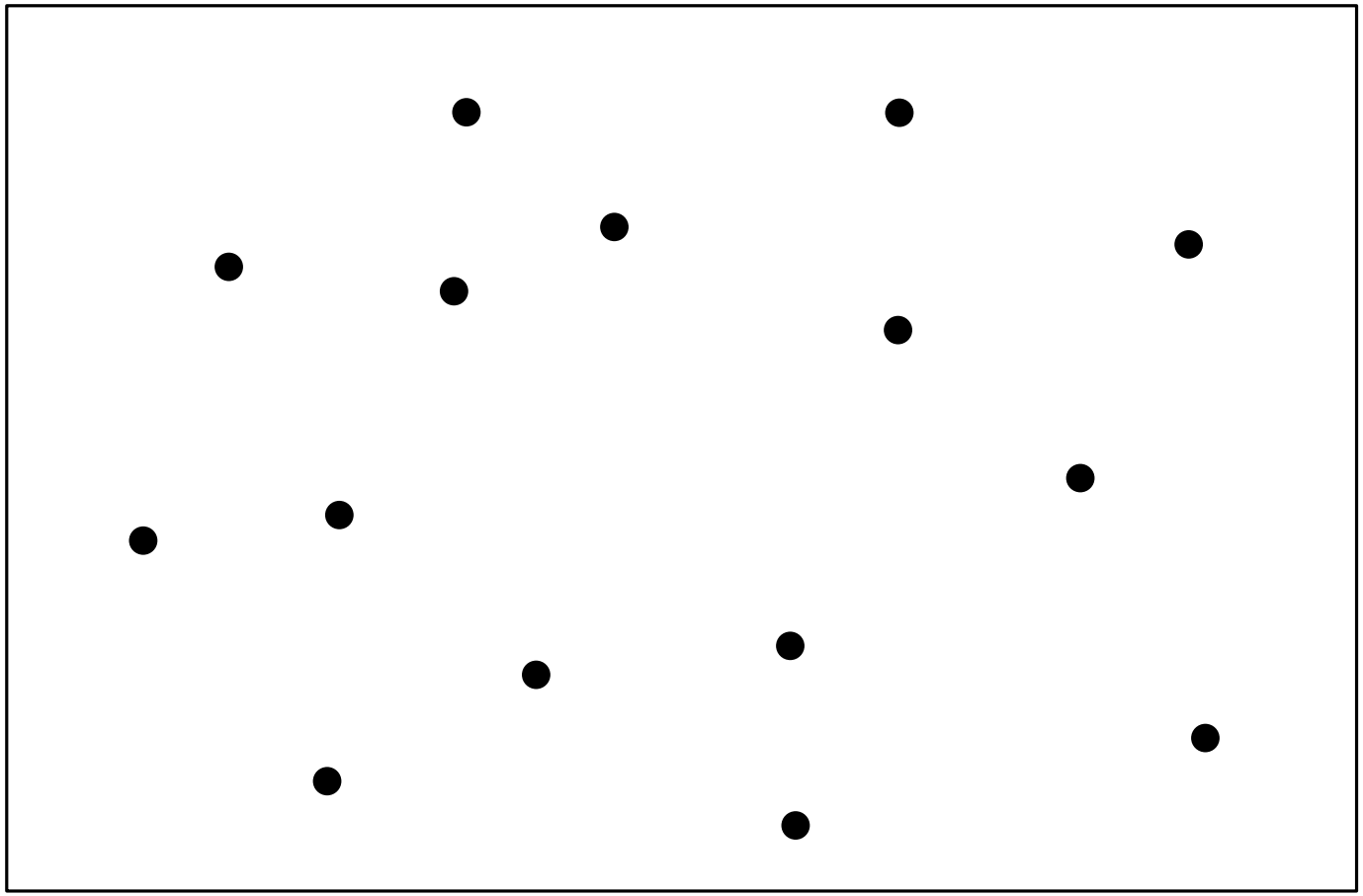
Applications

POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

Applications

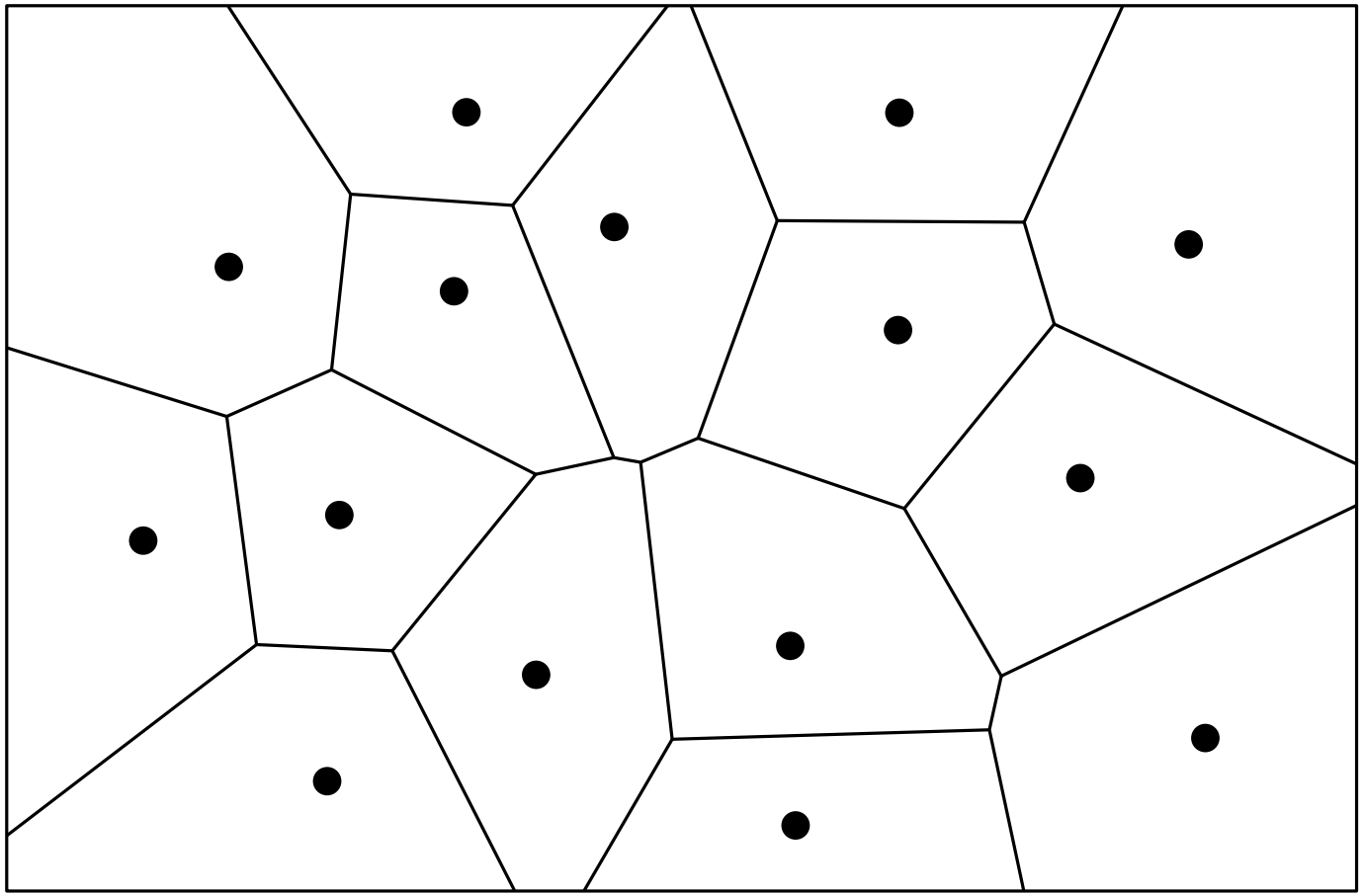


POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

Applications

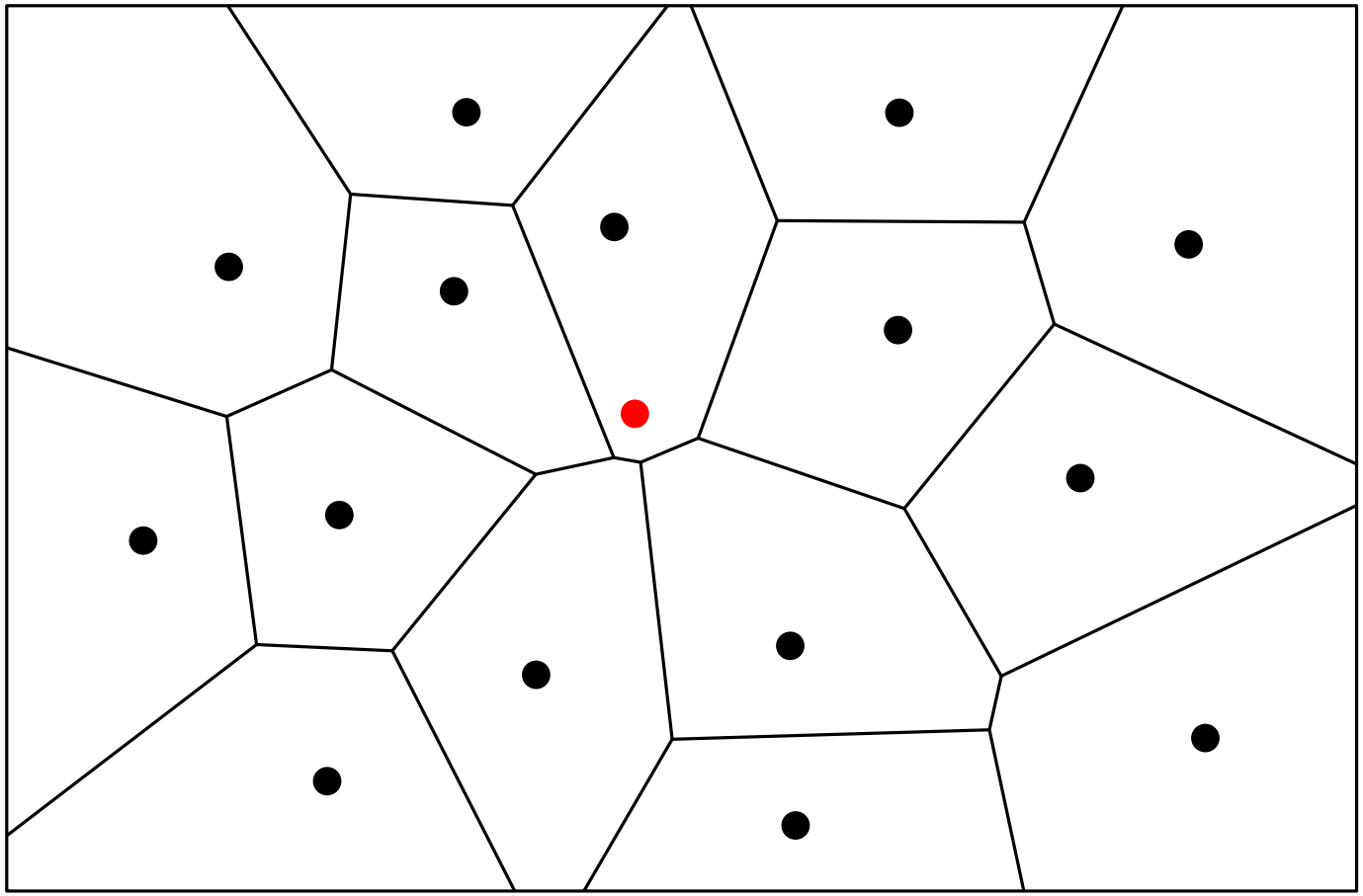


POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

Applications

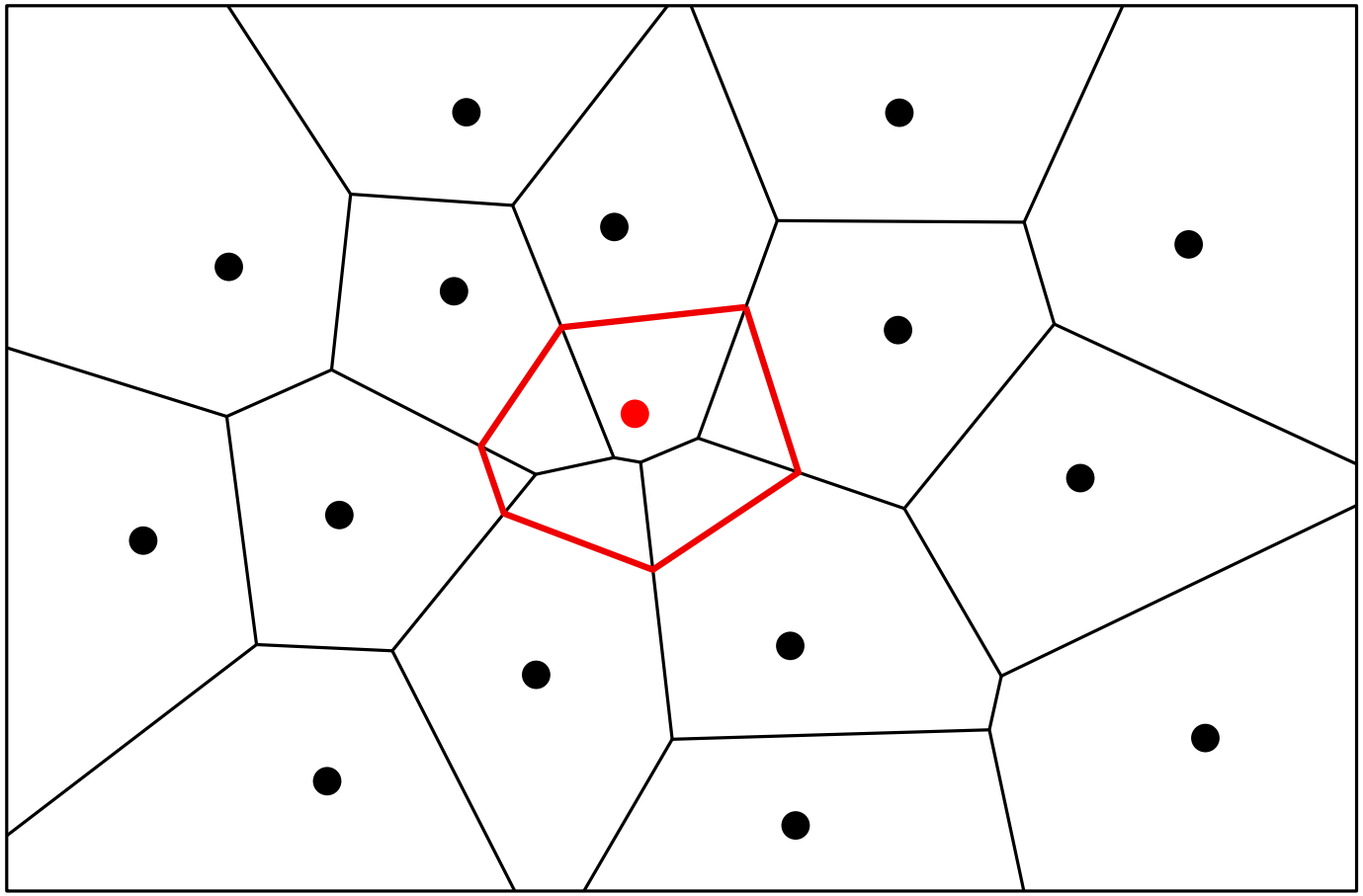


POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

Applications

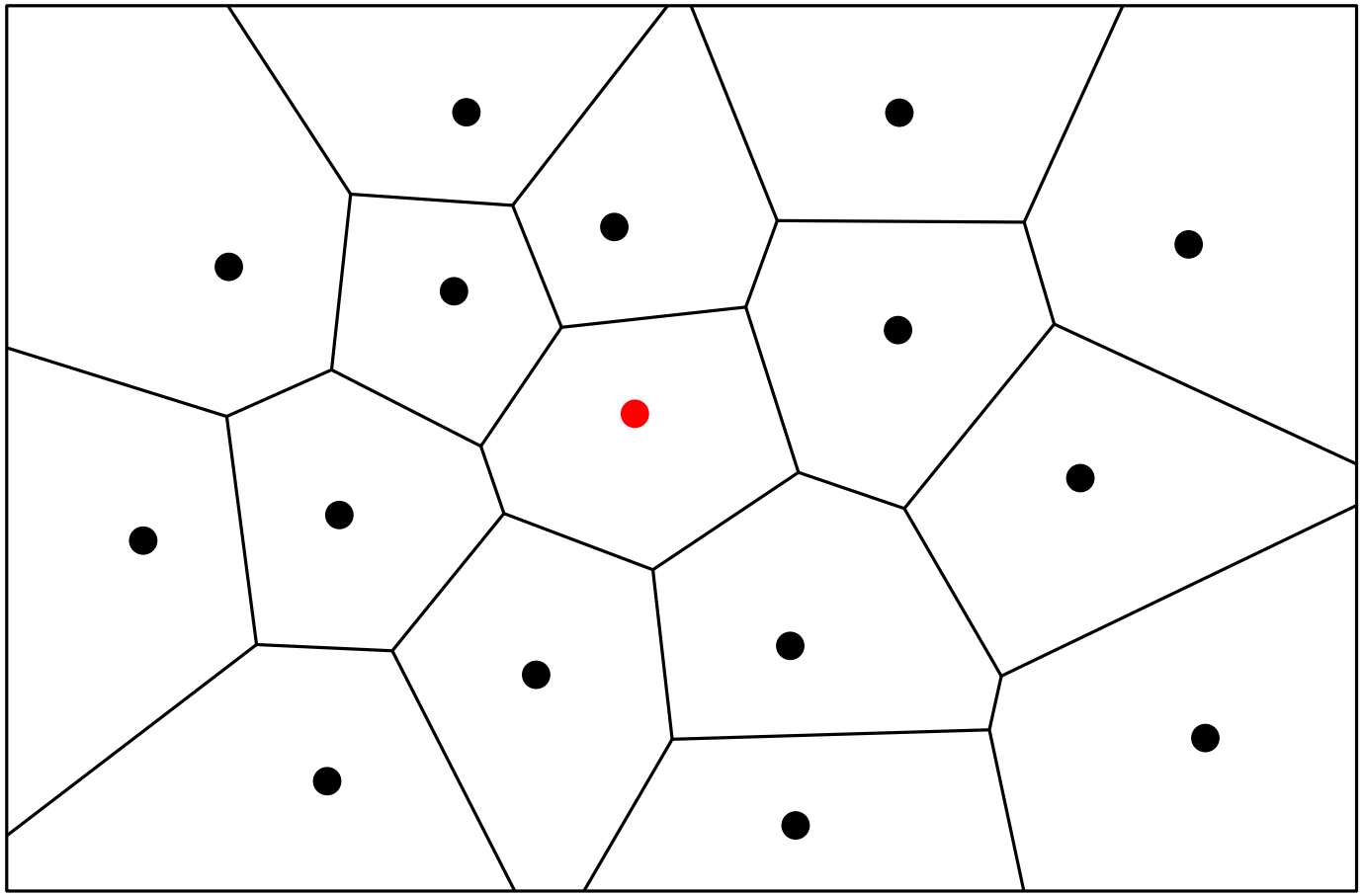


POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

Applications

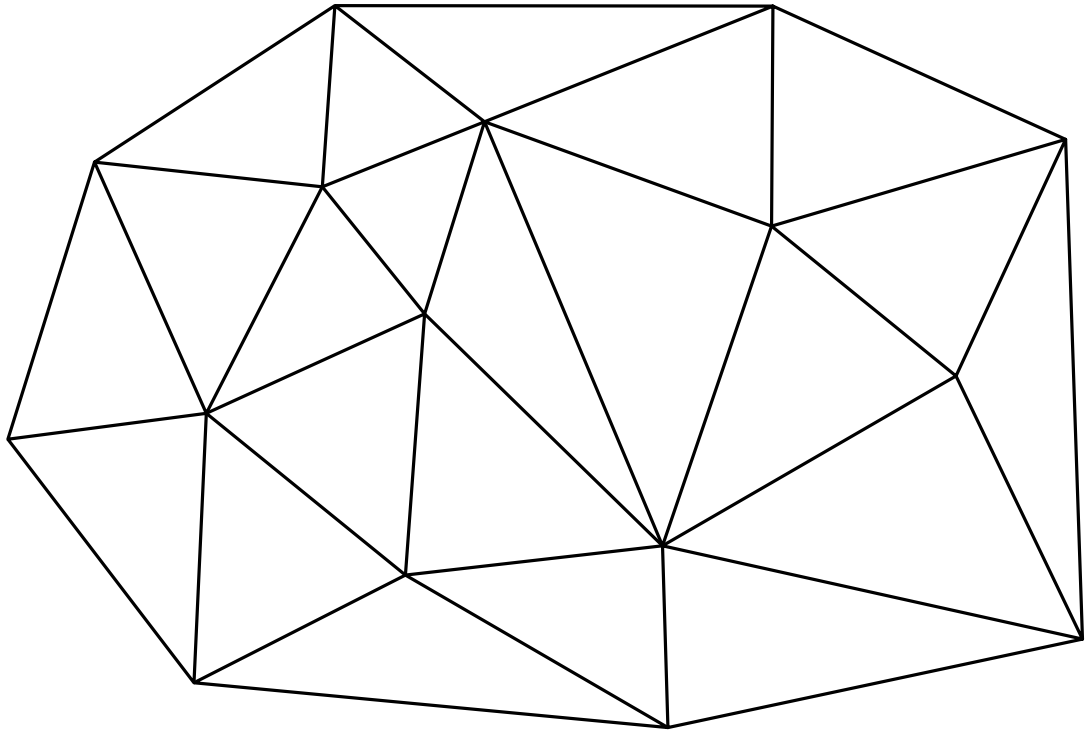


POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

Applications

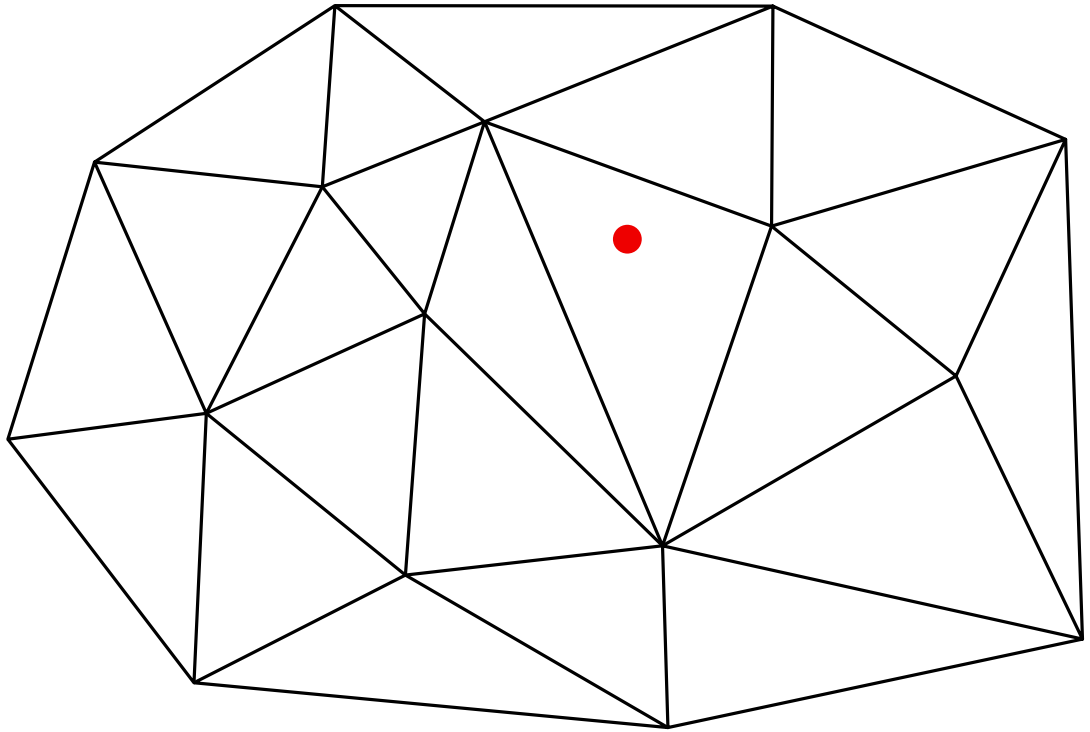


POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

Applications

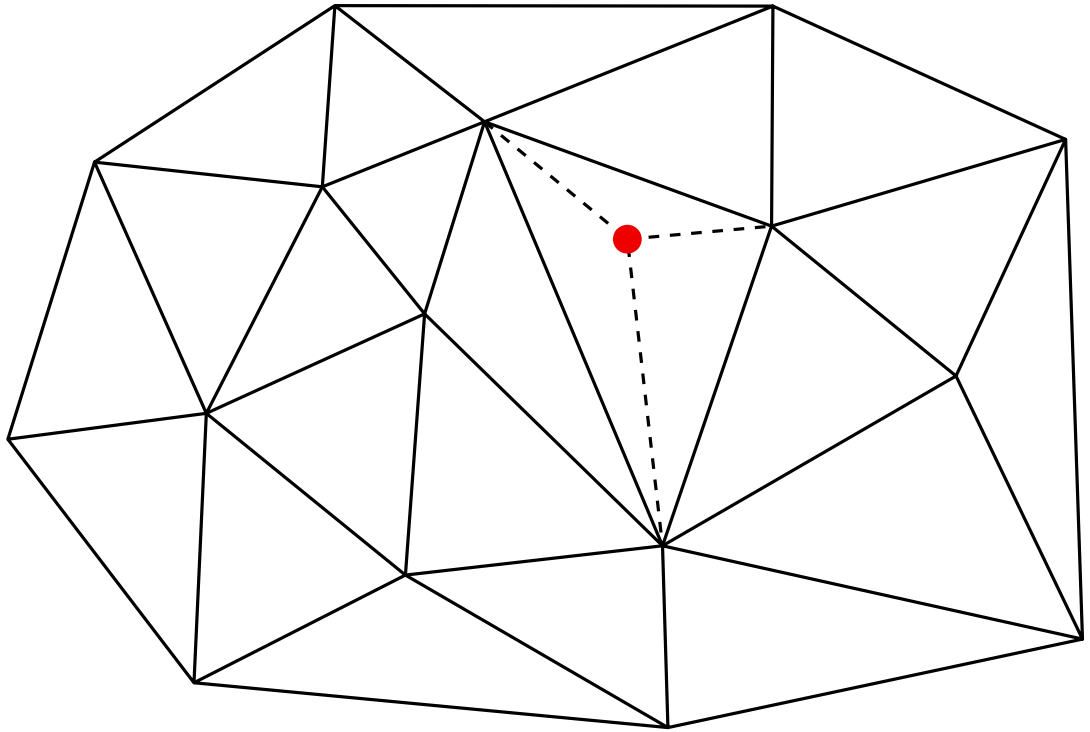


POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

Applications



POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

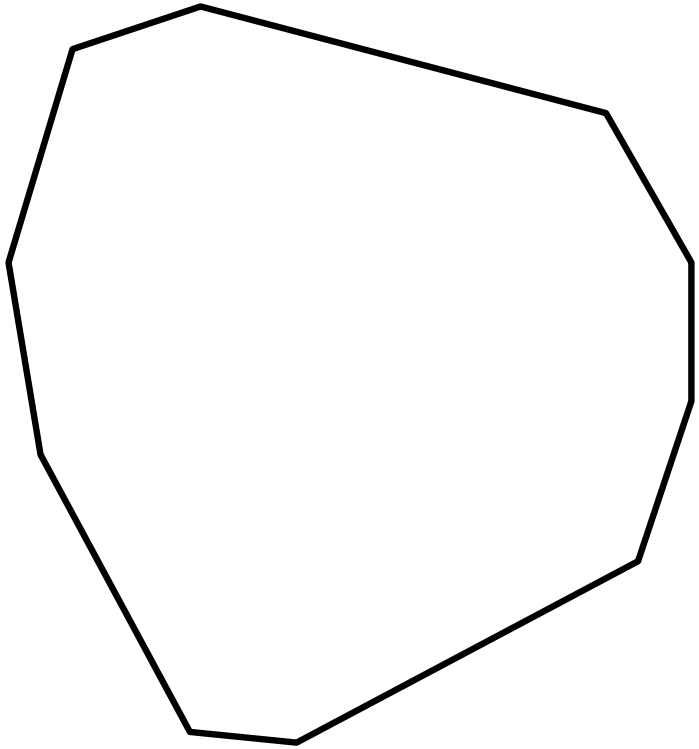
Particular cases

POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

Particular cases

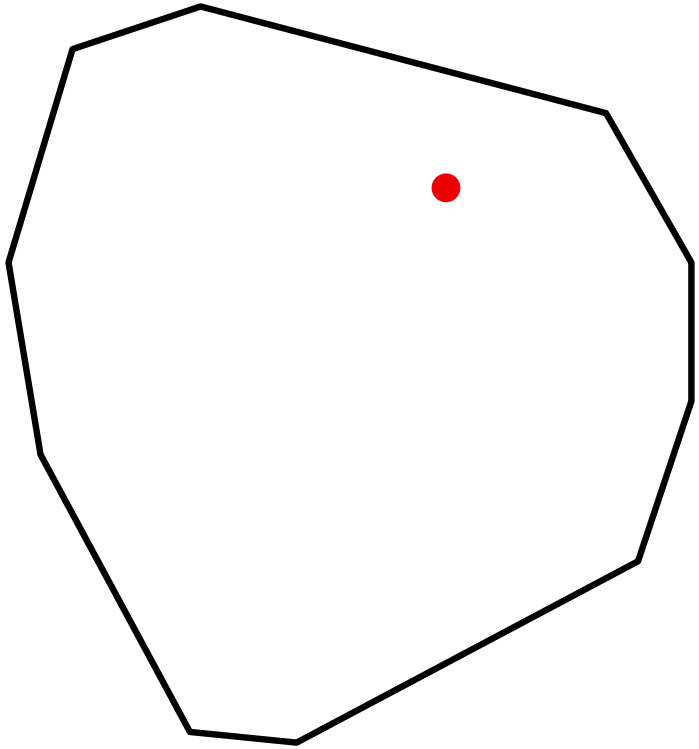


POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

Particular cases

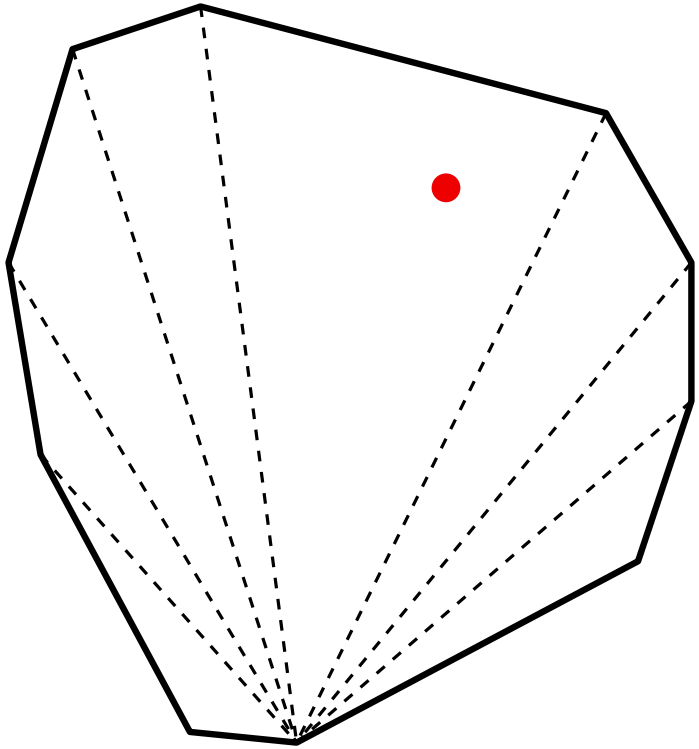


POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

Particular cases

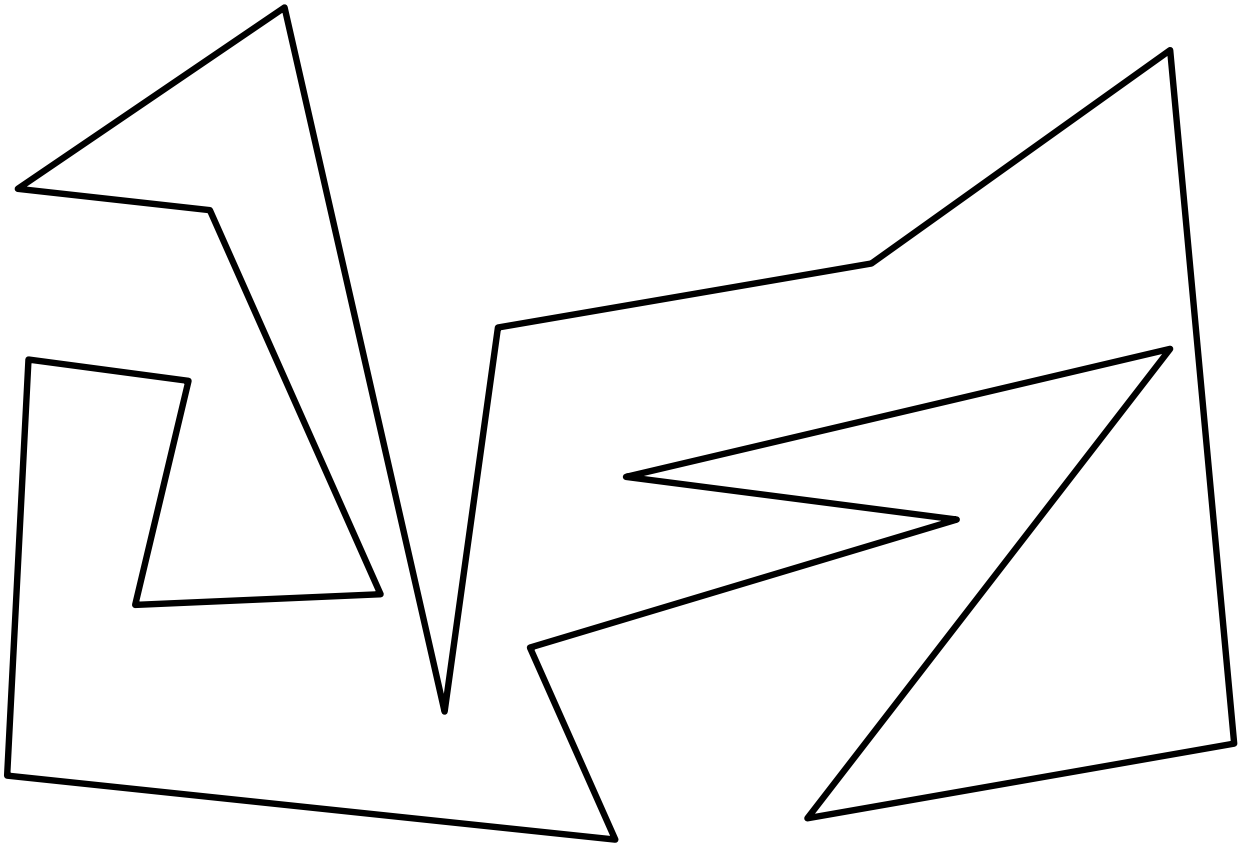


POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

Particular cases

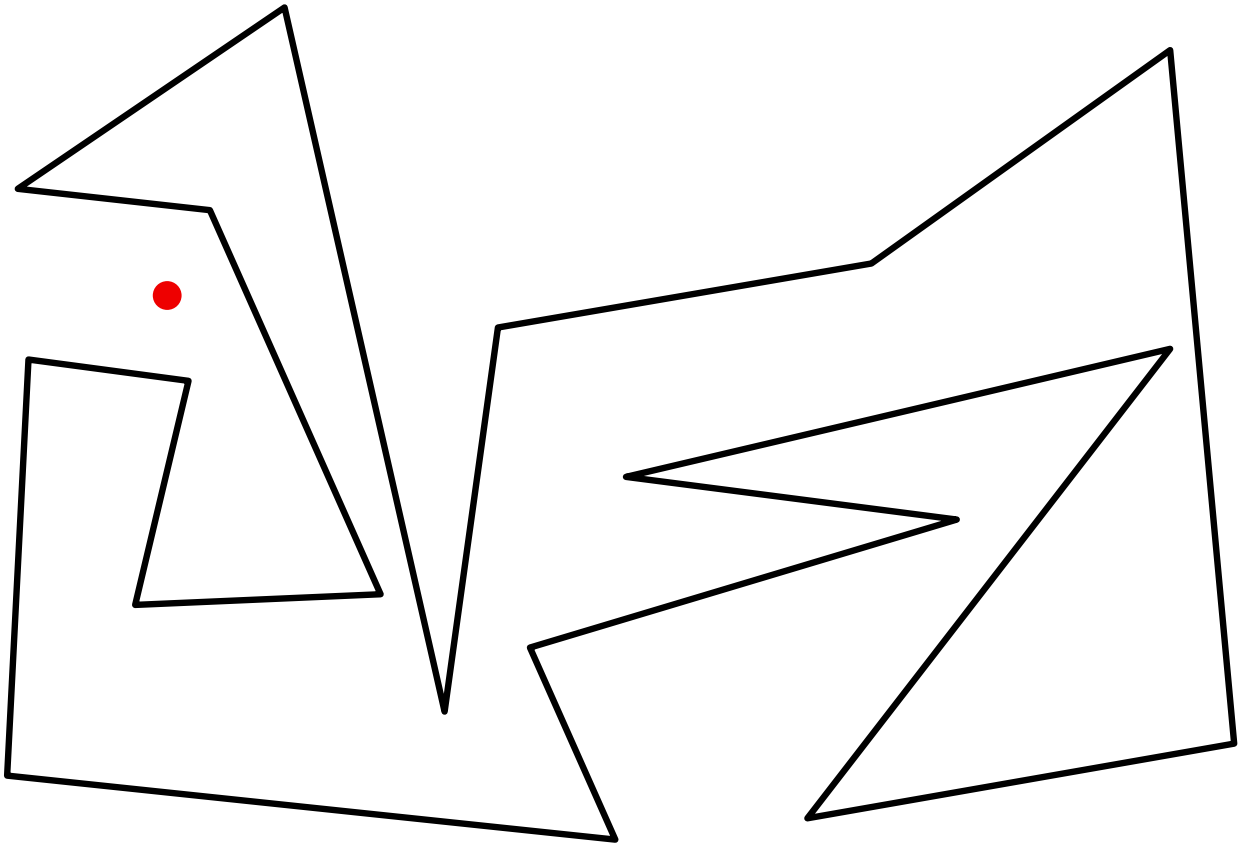


POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

Particular cases

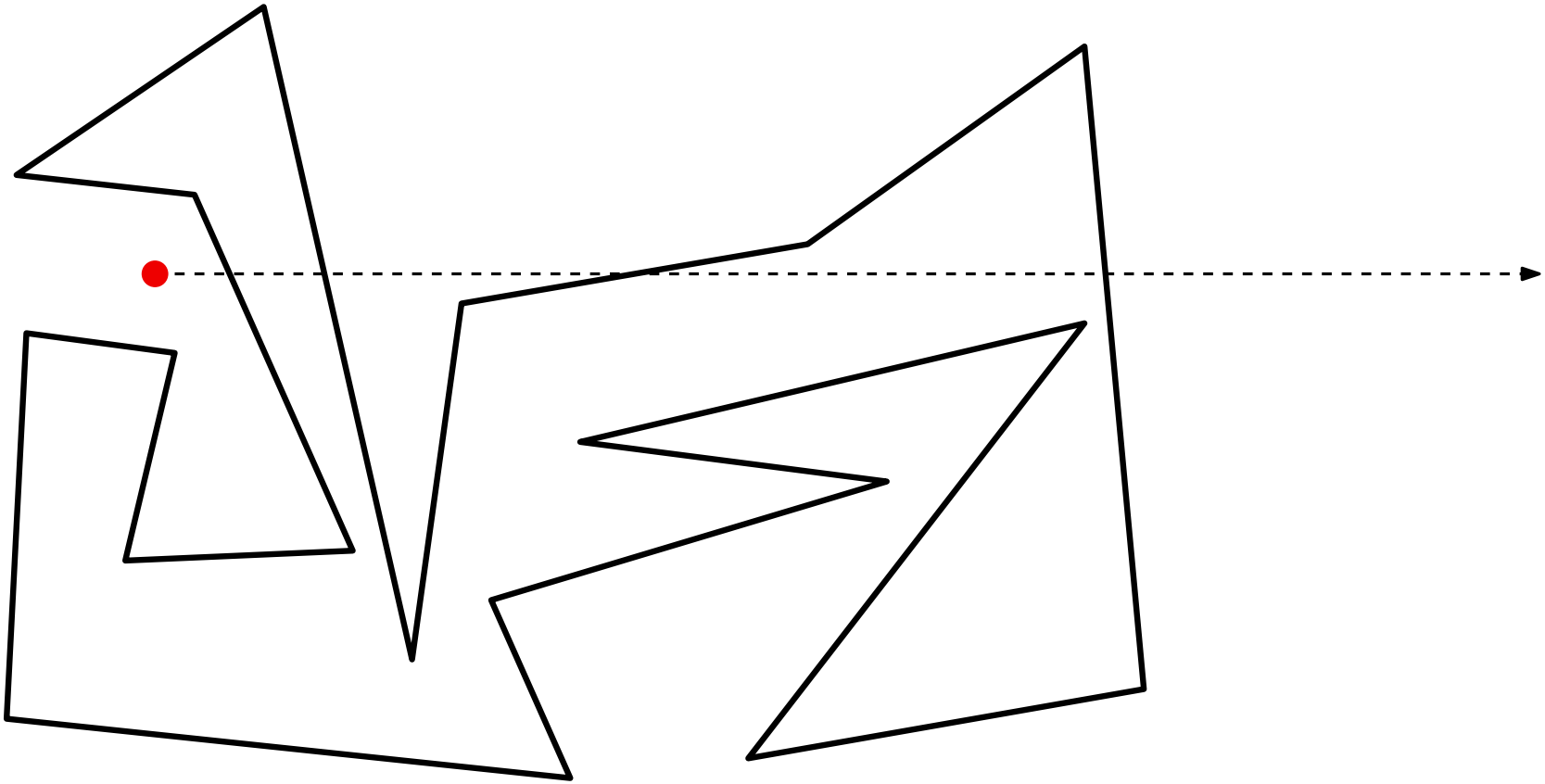


POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

Particular cases

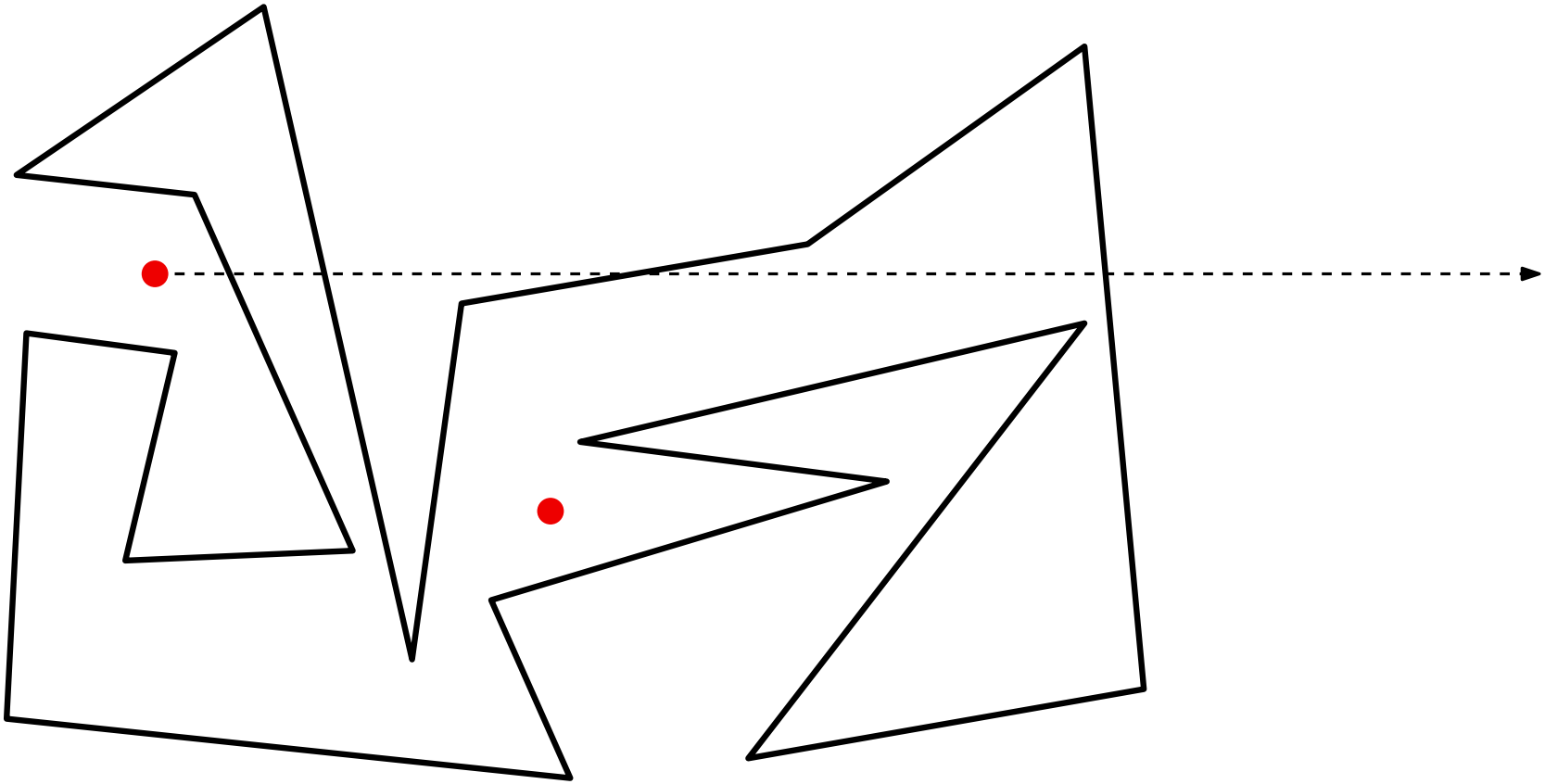


POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

Particular cases

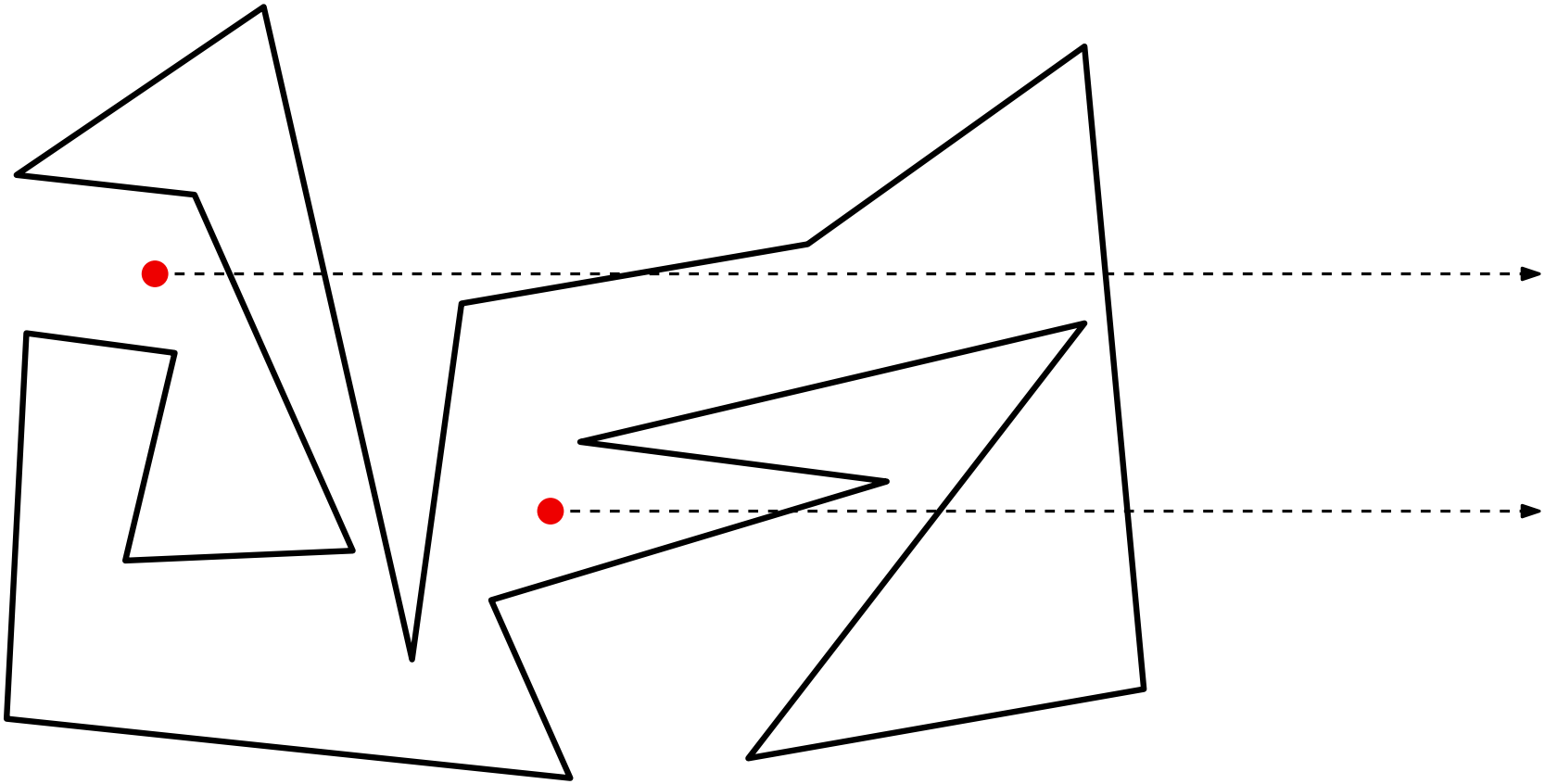


POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

Particular cases



POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

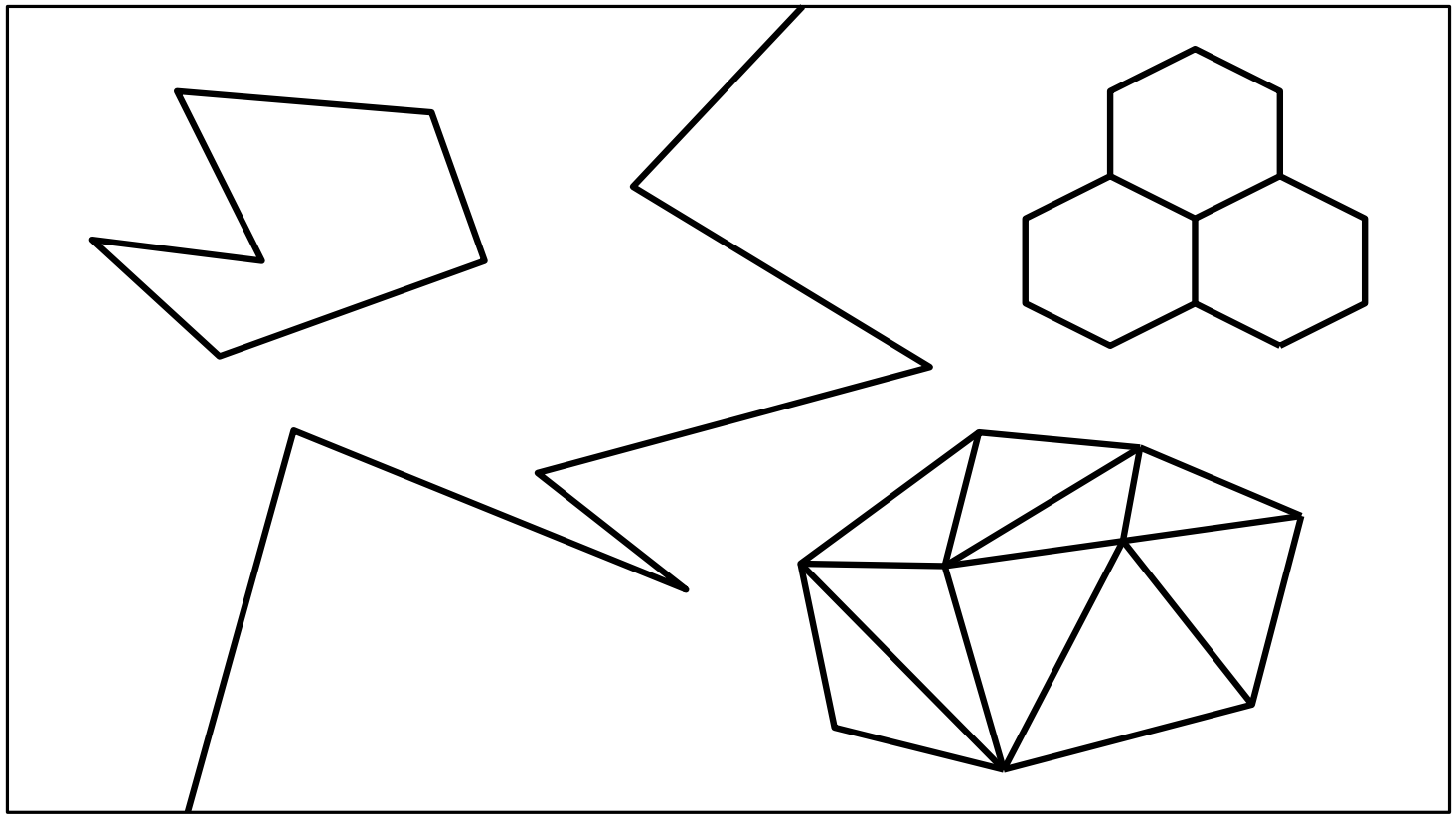
General case

POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

General case

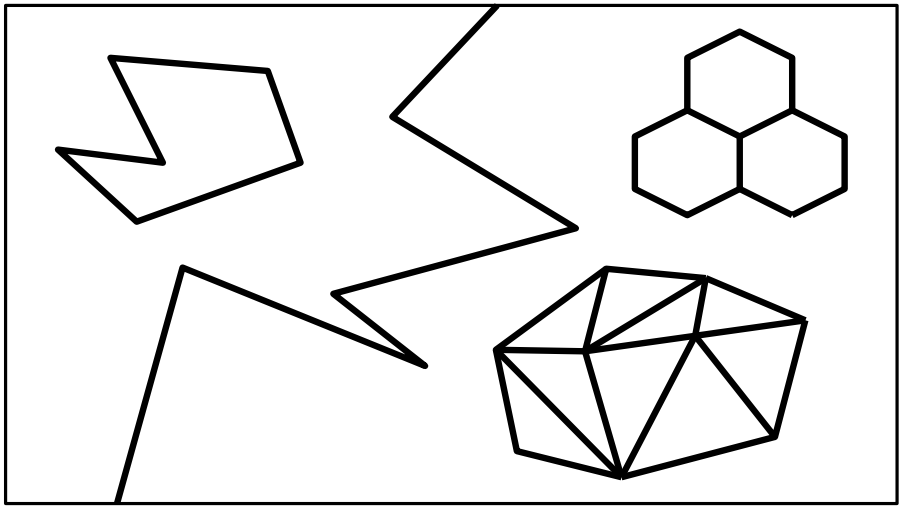


POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

General case



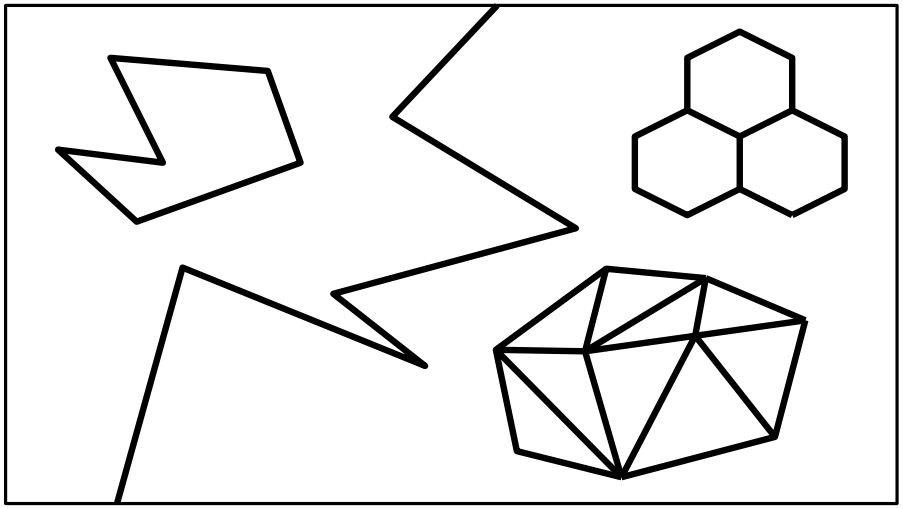
Strategy

POINT LOCATION: Introduction

The problem

Given a planar subdivision defined by a planar and rectilinear graph of size n, decide in which region of the decomposition is located a given point p.

General case



Strategy

Adequately preprocess the planar subdivision,
to be able to efficiently answer point location *queries*

Walking in a triangulation

POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Rectilinear walk

The algorithm visits all the triangles intersected by the line-segment pq.

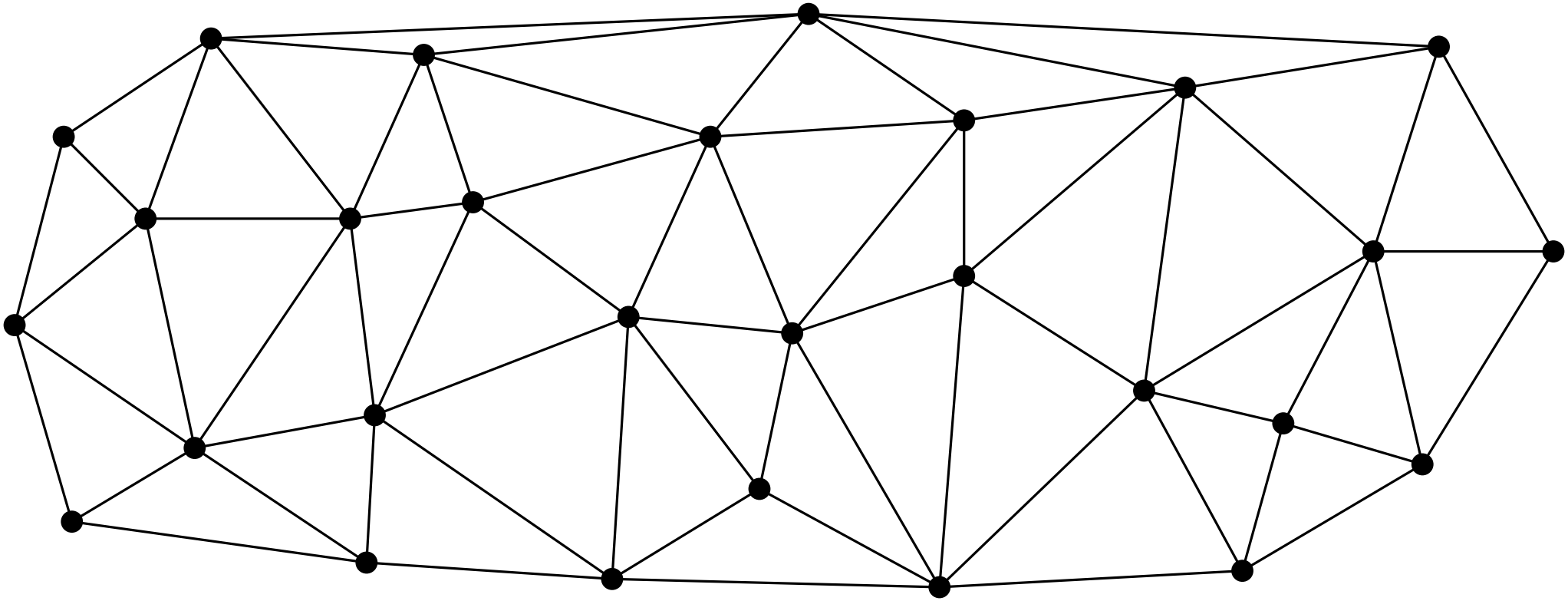
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Rectilinear walk

The algorithm visits all the triangles intersected by the line-segment pq.



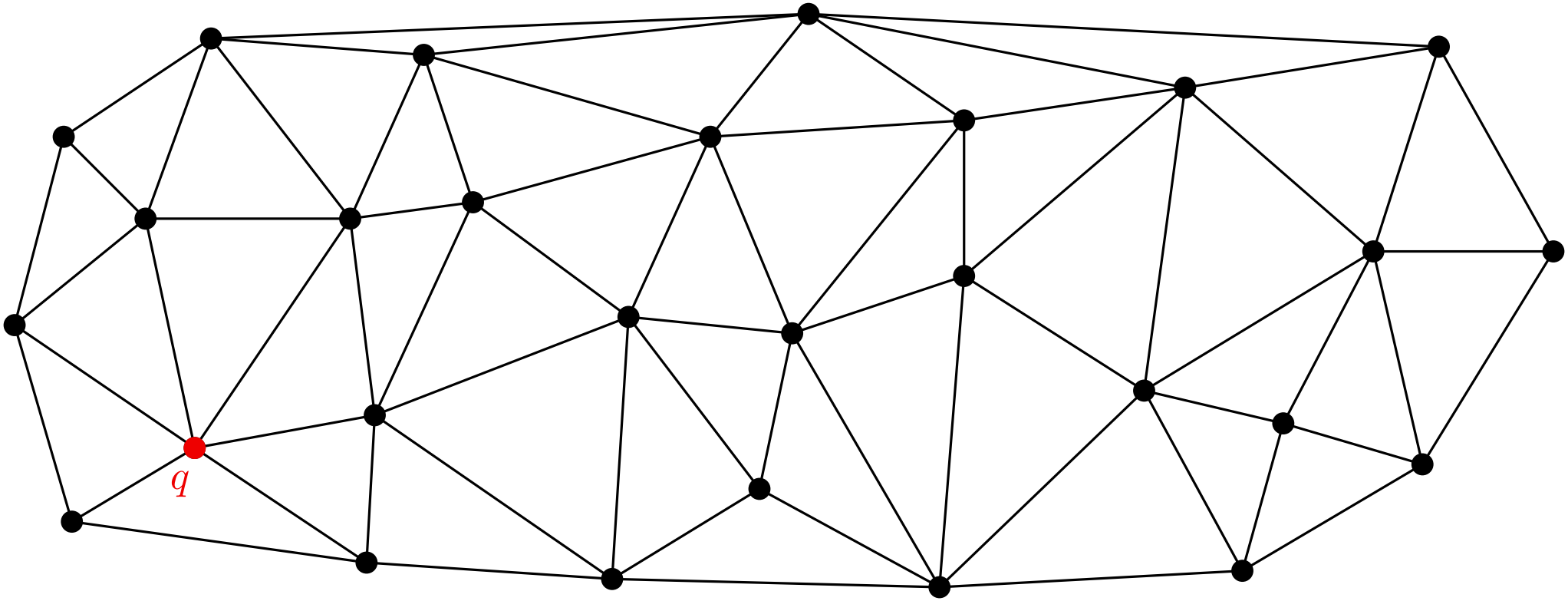
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Rectilinear walk

The algorithm visits all the triangles intersected by the line-segment pq.



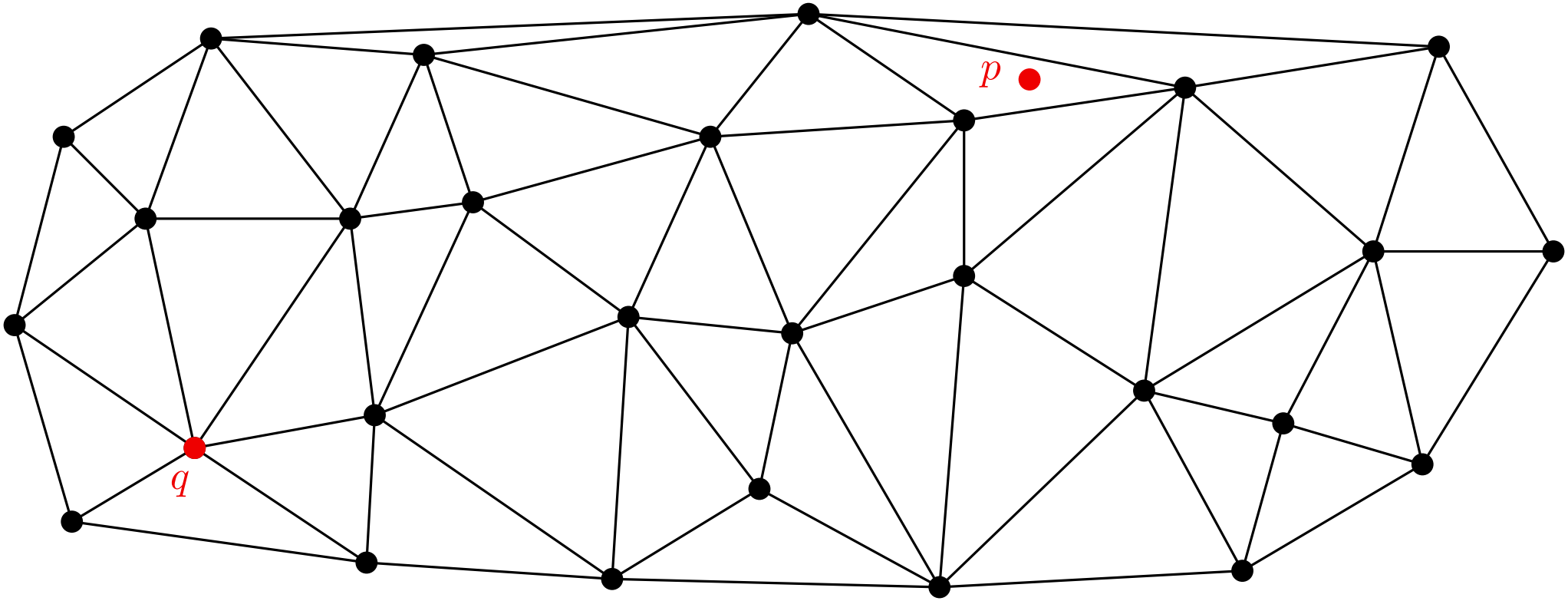
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Rectilinear walk

The algorithm visits all the triangles intersected by the line-segment pq.



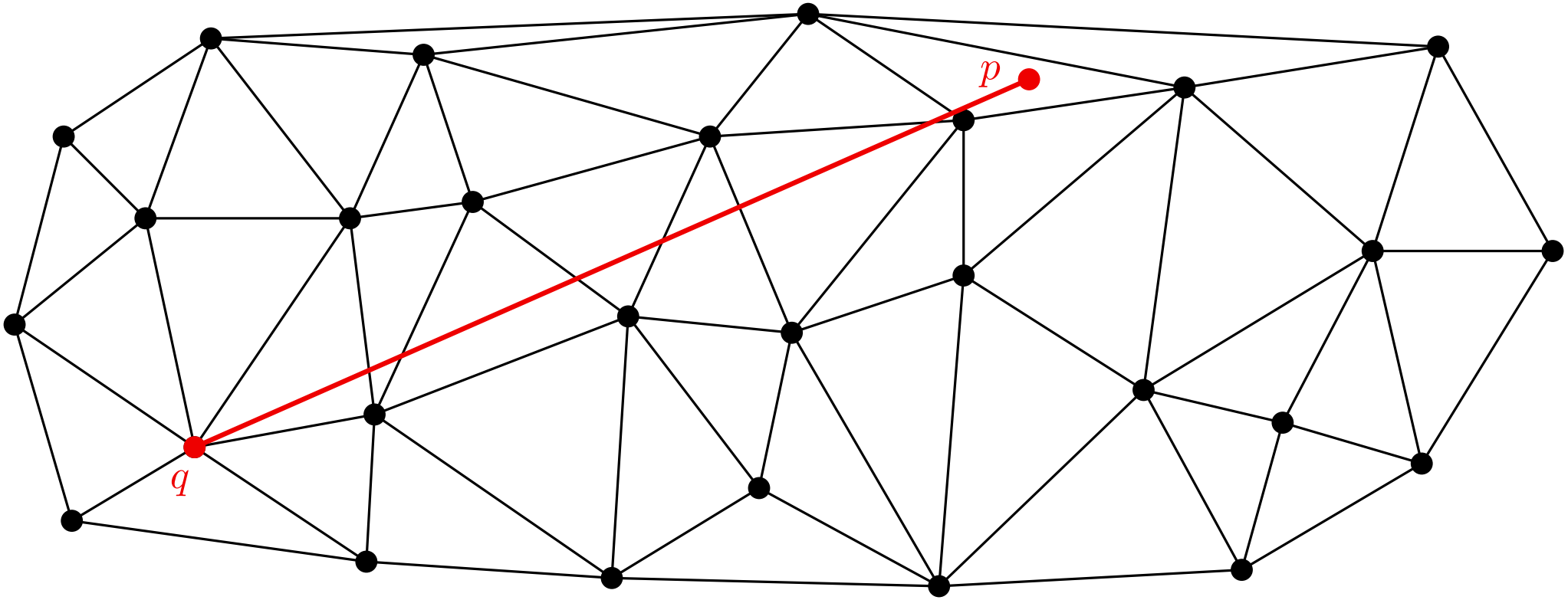
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Rectilinear walk

The algorithm visits all the triangles intersected by the line-segment pq.



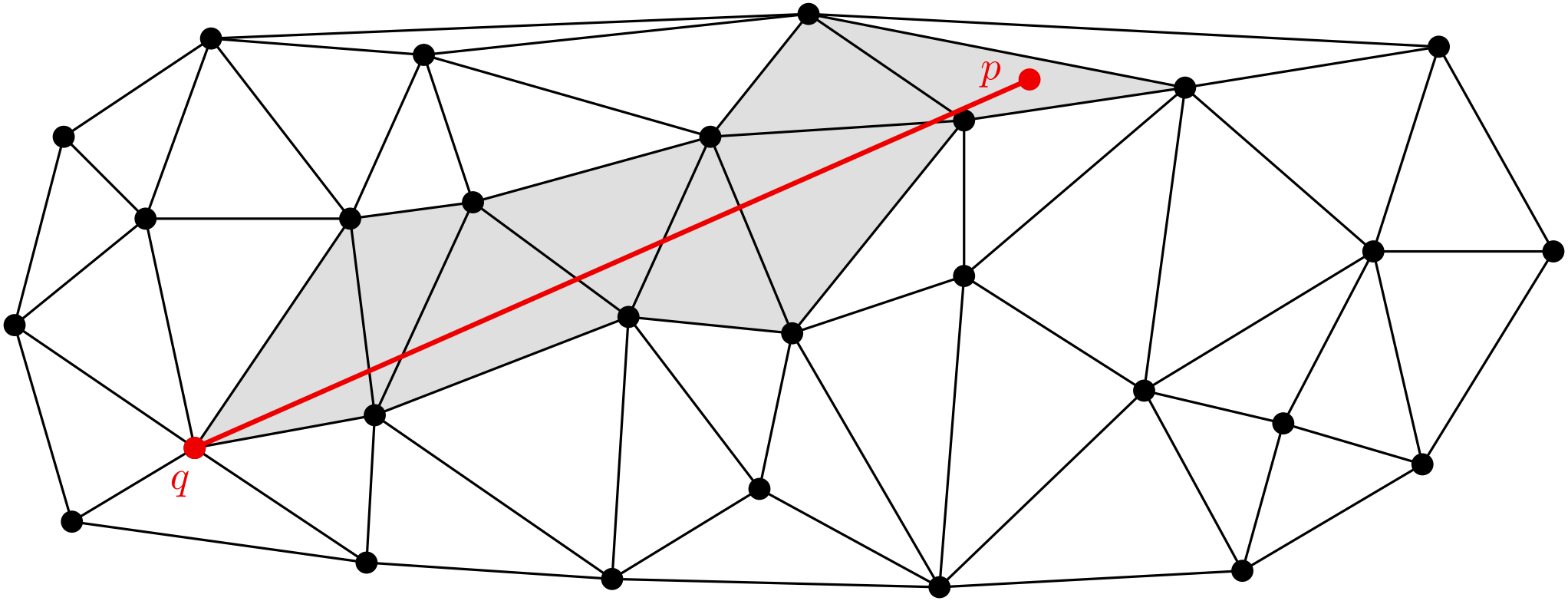
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Rectilinear walk

The algorithm visits all the triangles intersected by the line-segment pq.



POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Orthogonal walk

The algorithm visits all the triangles intersected by an isothetic path from p to q, increasing first one coordinate then the other one.

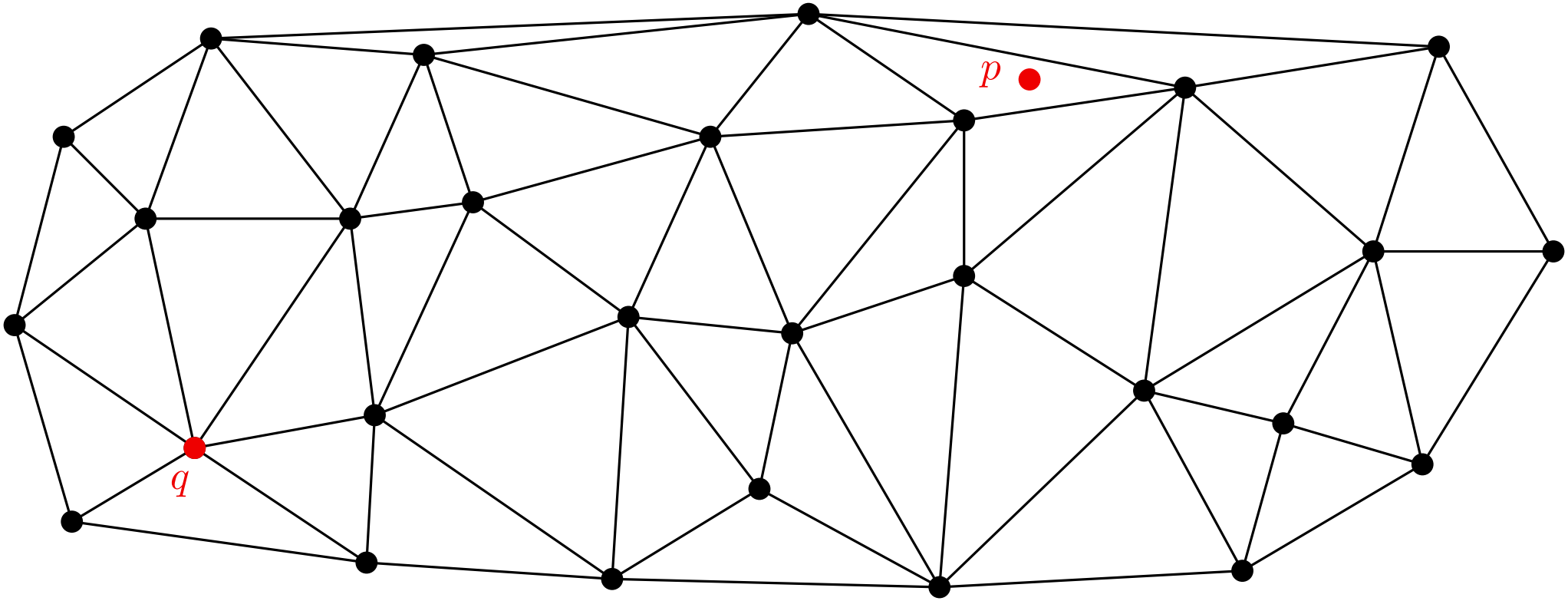
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Orthogonal walk

The algorithm visits all the triangles intersected by an isothetic path from p to q, increasing first one coordinate then the other one.



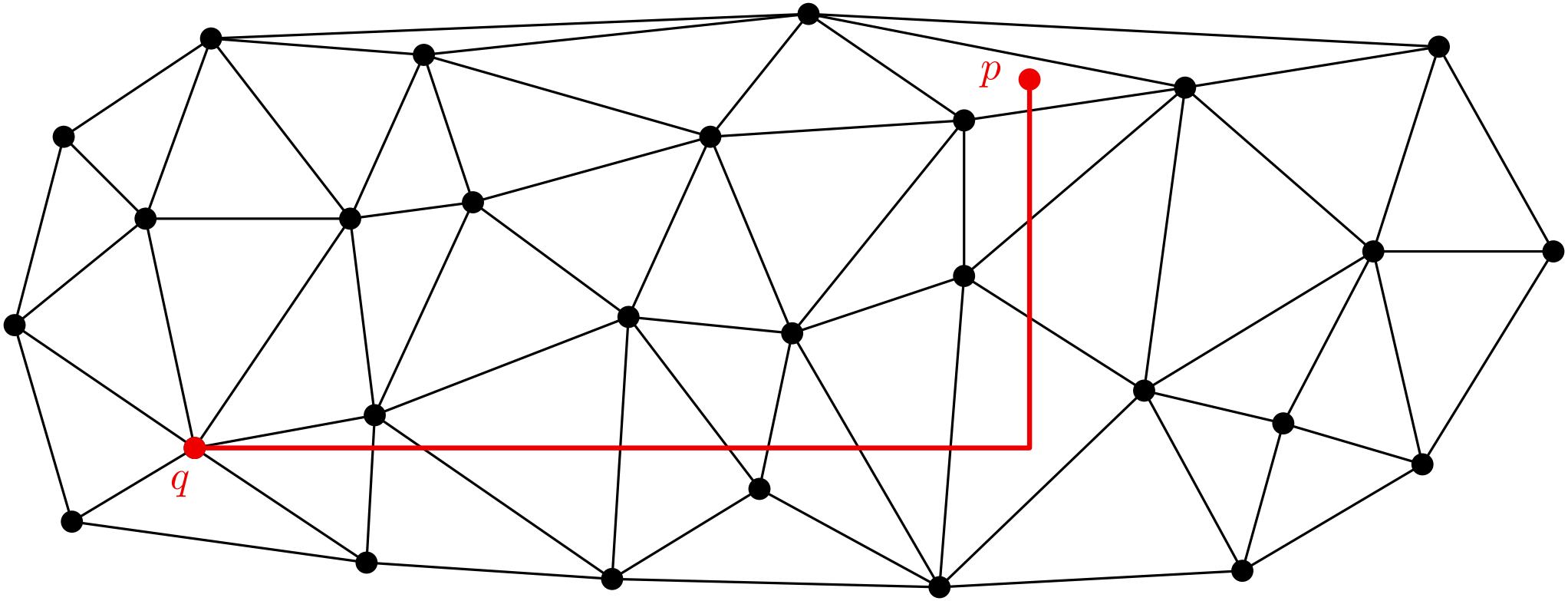
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Orthogonal walk

The algorithm visits all the triangles intersected by an isothetic path from p to q, increasing first one coordinate then the other one.



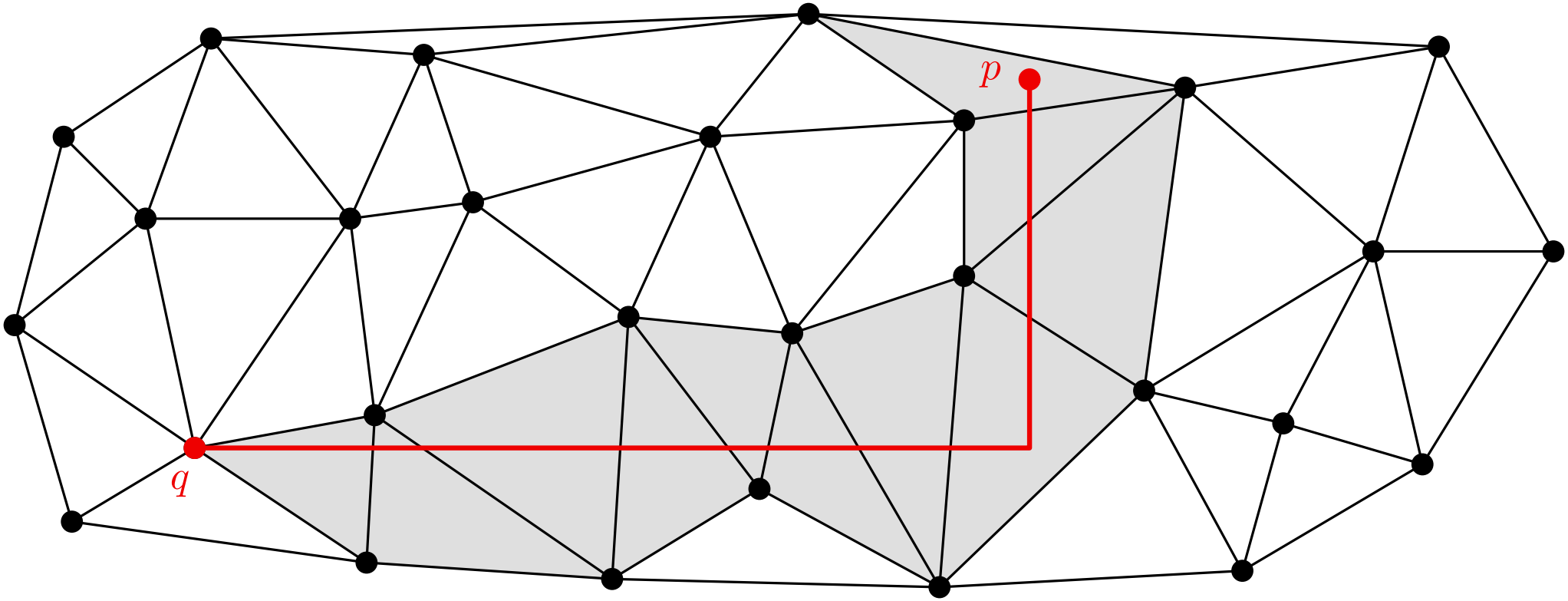
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Orthogonal walk

The algorithm visits all the triangles intersected by an isothetic path from p to q, increasing first one coordinate then the other one.



POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Visibility walk

It consists on visiting adjacent triangles, crossing edges for which p and q lie in opposite sides.

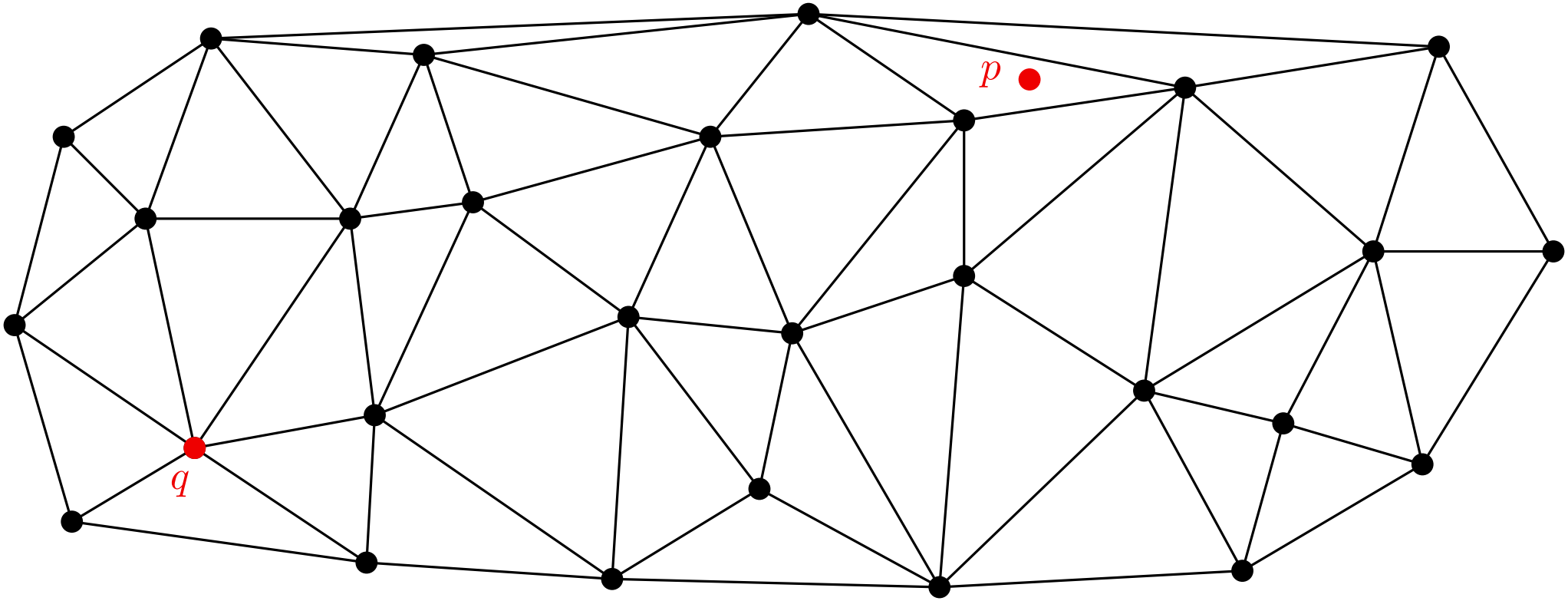
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Visibility walk

It consists on visiting adjacent triangles, crossing edges for which p and q lie in opposite sides.



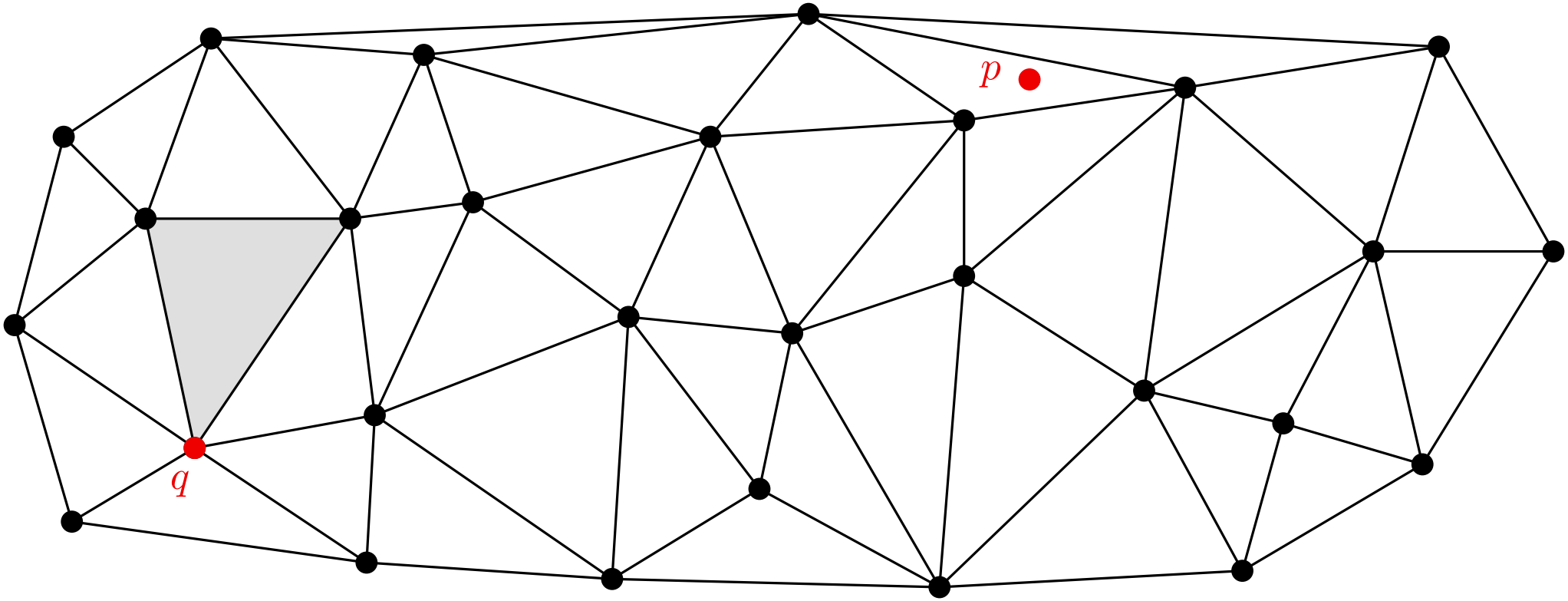
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Visibility walk

It consists on visiting adjacent triangles, crossing edges for which p and q lie in opposite sides.



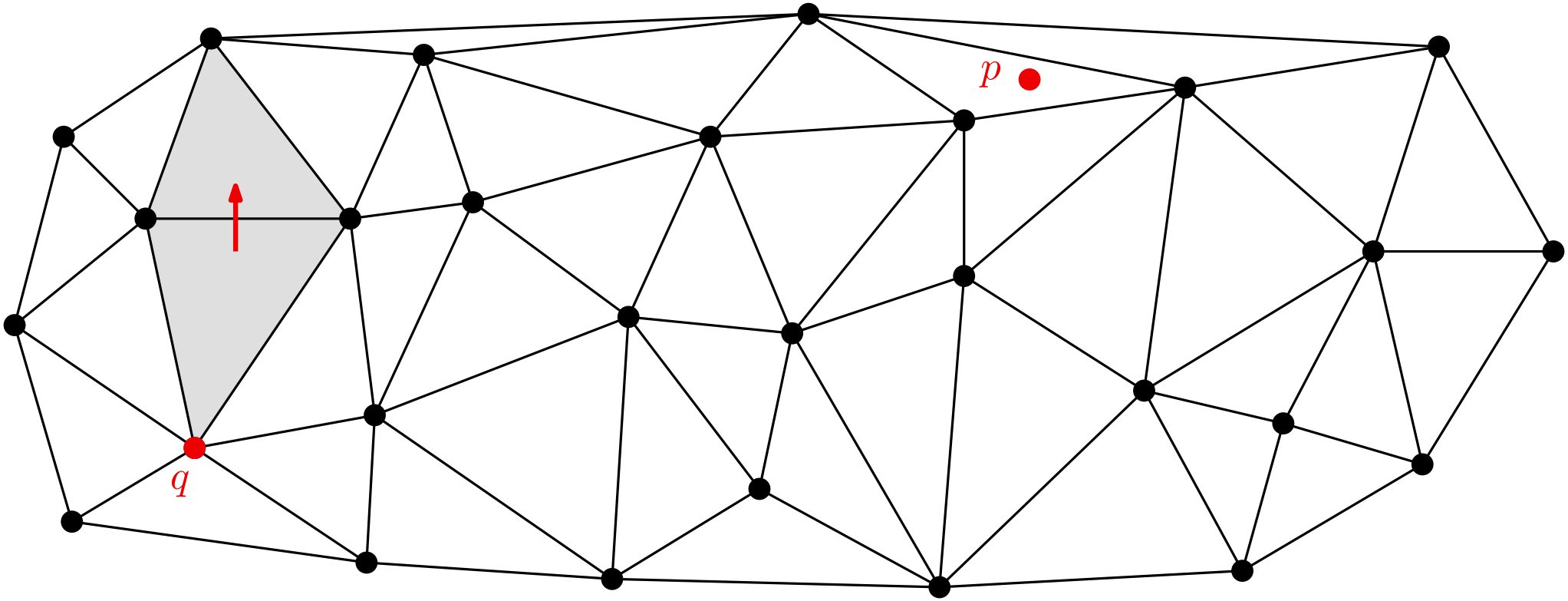
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Visibility walk

It consists on visiting adjacent triangles, crossing edges for which p and q lie in opposite sides.



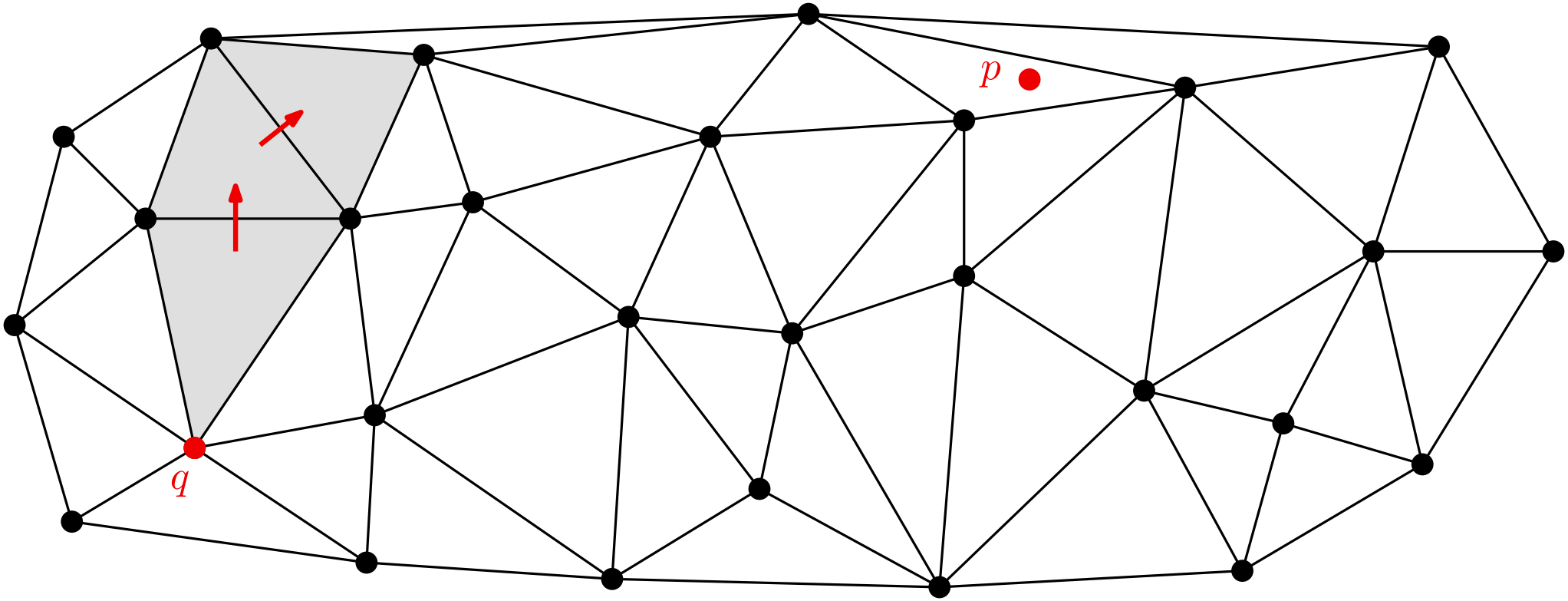
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Visibility walk

It consists on visiting adjacent triangles, crossing edges for which p and q lie in opposite sides.



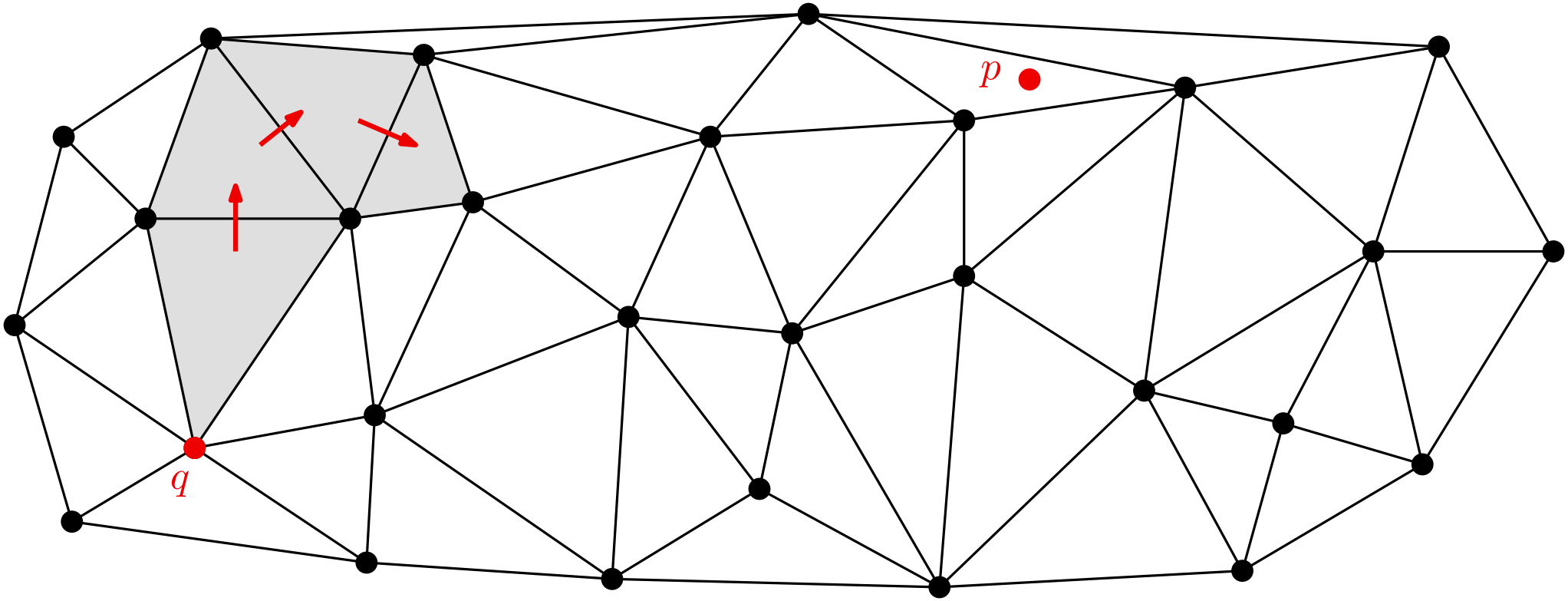
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Visibility walk

It consists on visiting adjacent triangles, crossing edges for which p and q lie in opposite sides.



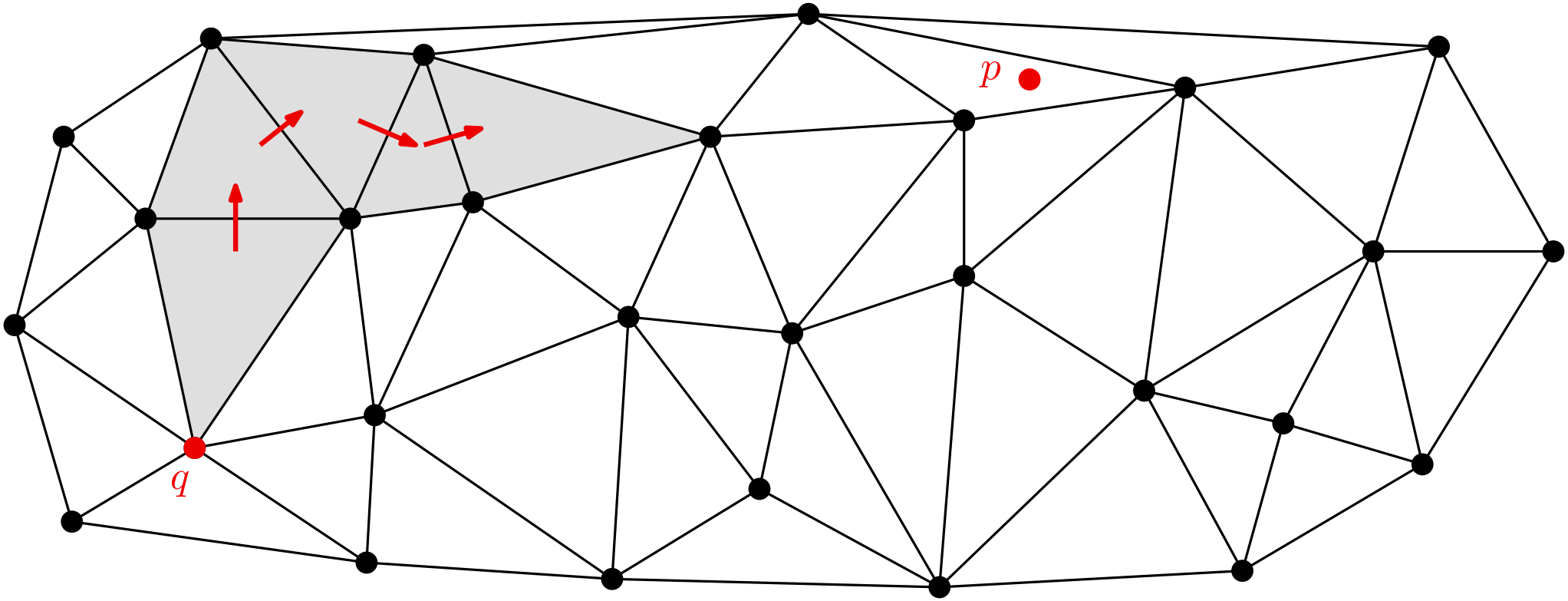
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Visibility walk

It consists on visiting adjacent triangles, crossing edges for which p and q lie in opposite sides.



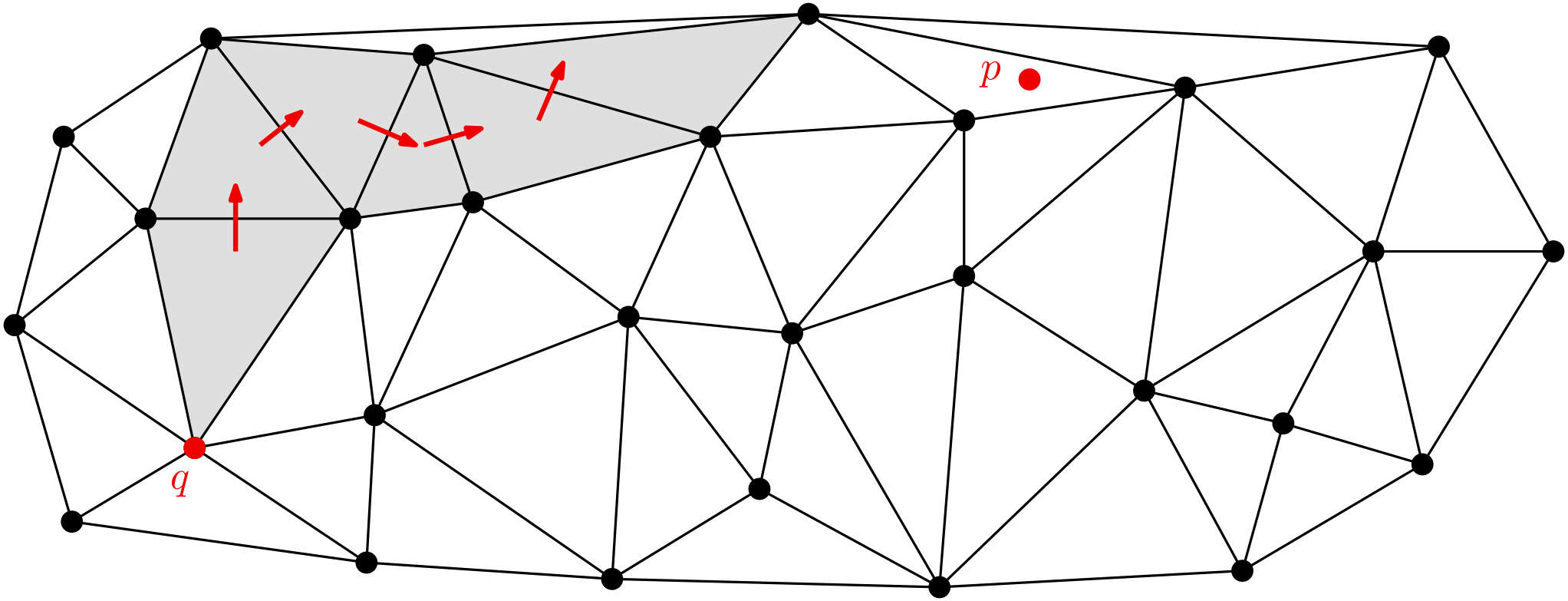
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Visibility walk

It consists on visiting adjacent triangles, crossing edges for which p and q lie in opposite sides.



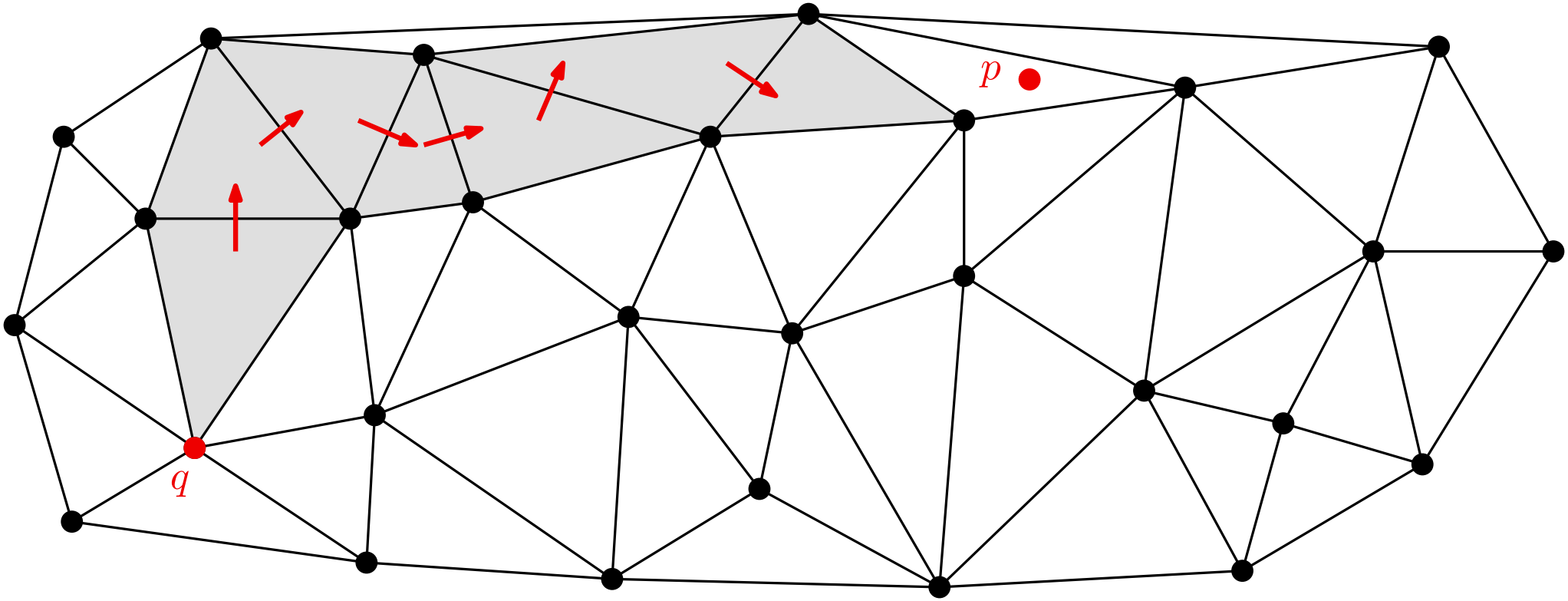
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Visibility walk

It consists on visiting adjacent triangles, crossing edges for which p and q lie in opposite sides.



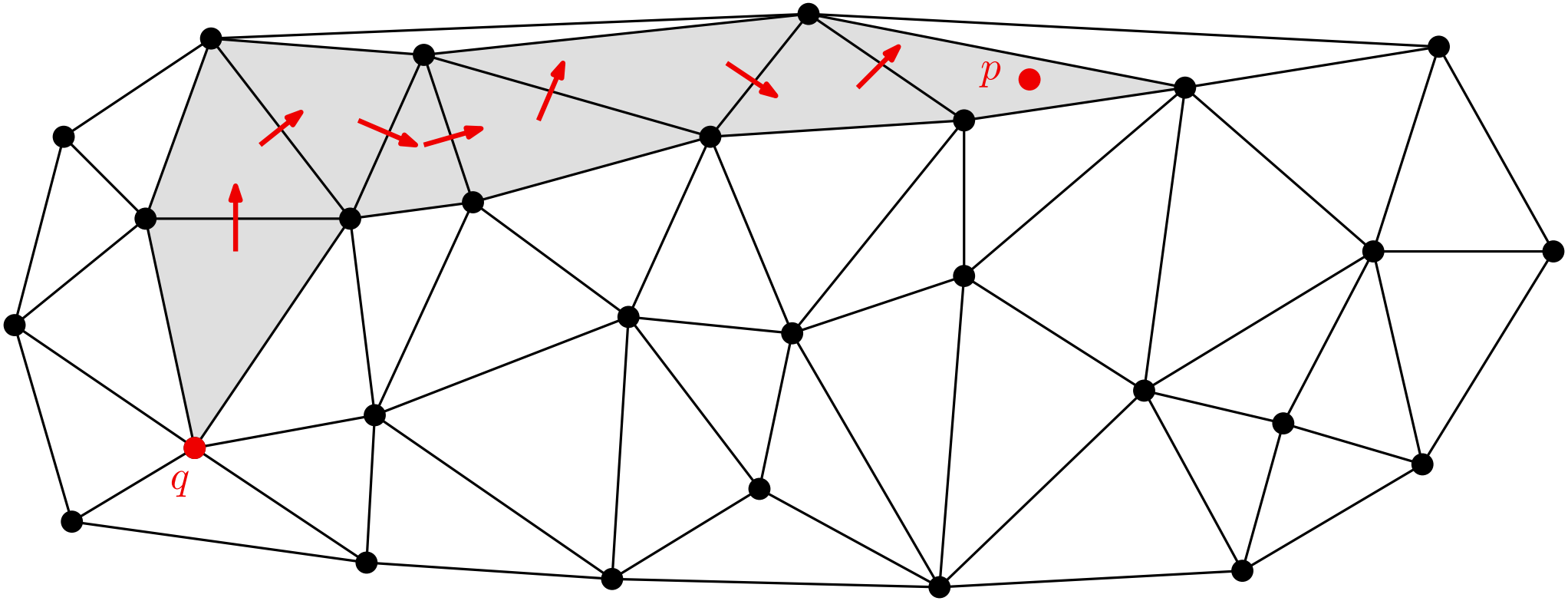
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Visibility walk

It consists on visiting adjacent triangles, crossing edges for which p and q lie in opposite sides.



POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Correction

- Rectilinear walk: obvious.
- Orthogonal walk: obvious.
- Visibility walk: works fine for Delaunay triangulations, but not for arbitrary triangulations.

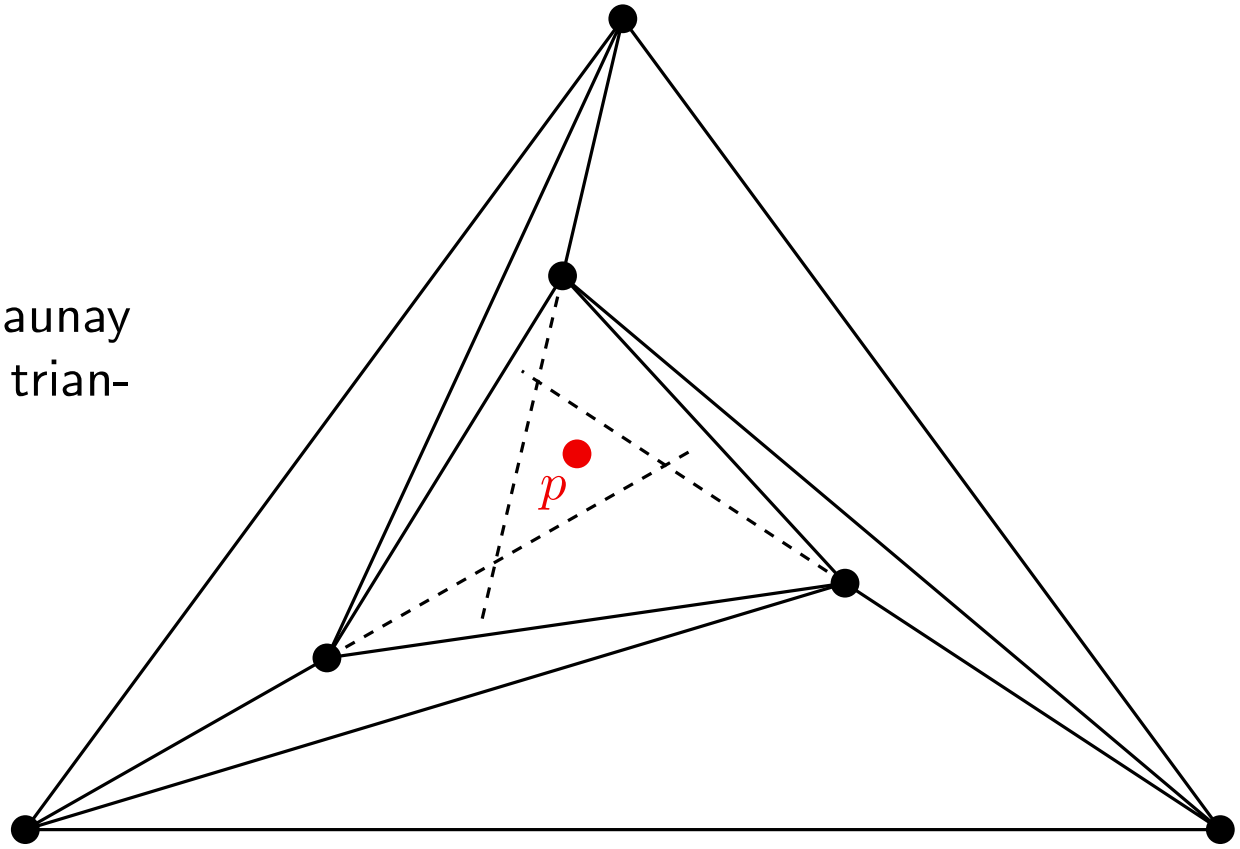
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Correction

- Rectilinear walk: obvious.
- Orthogonal walk: obvious.
- Visibility walk: works fine for Delaunay triangulations, but not for arbitrary triangulations.



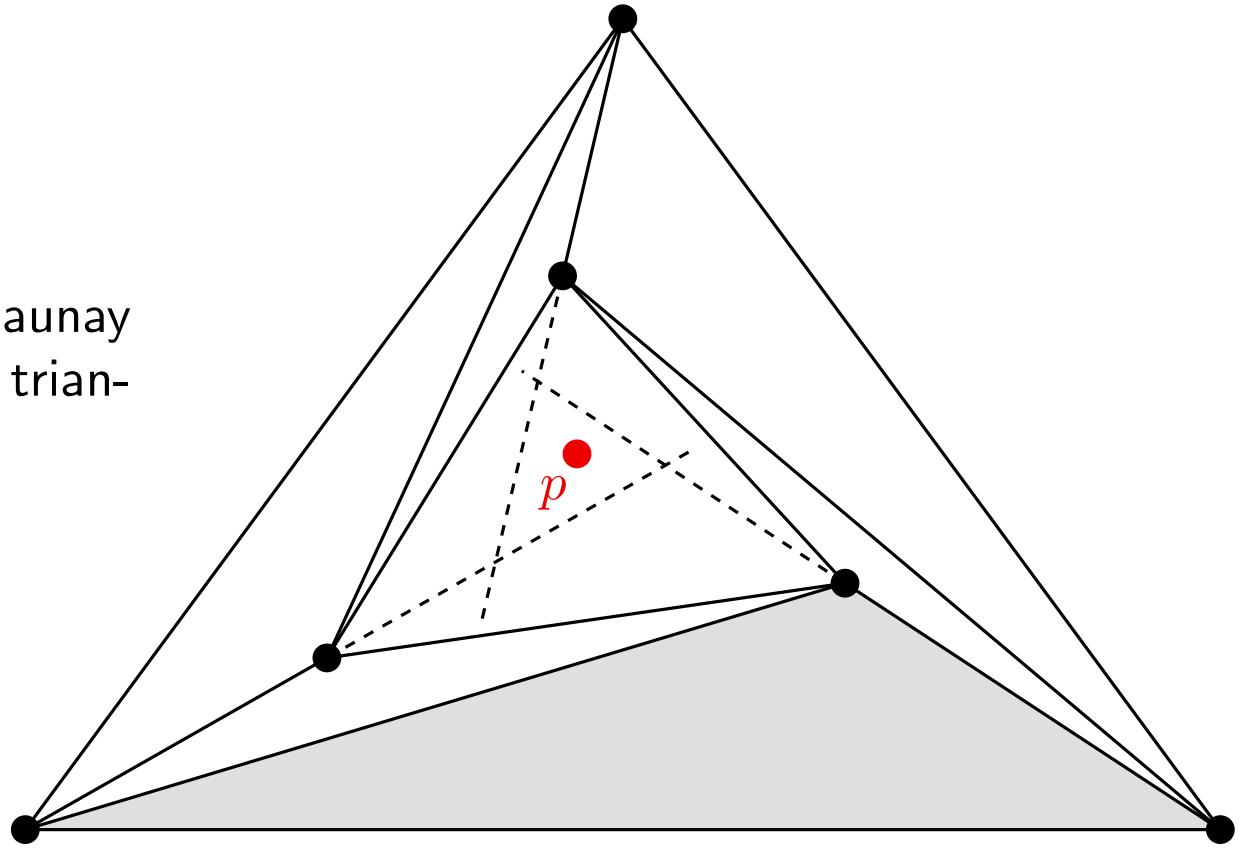
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Correction

- Rectilinear walk: obvious.
- Orthogonal walk: obvious.
- Visibility walk: works fine for Delaunay triangulations, but not for arbitrary triangulations.



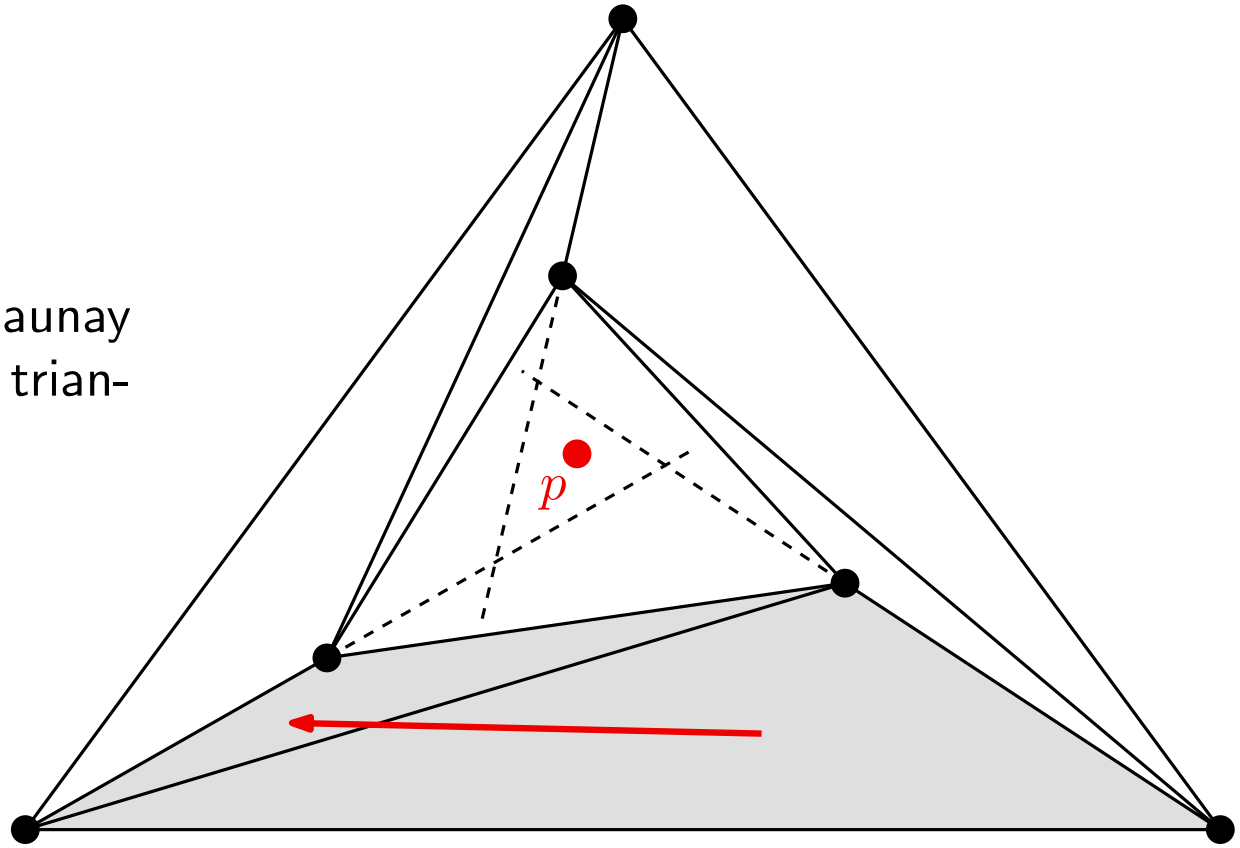
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Correction

- Rectilinear walk: obvious.
- Orthogonal walk: obvious.
- Visibility walk: works fine for Delaunay triangulations, but not for arbitrary triangulations.



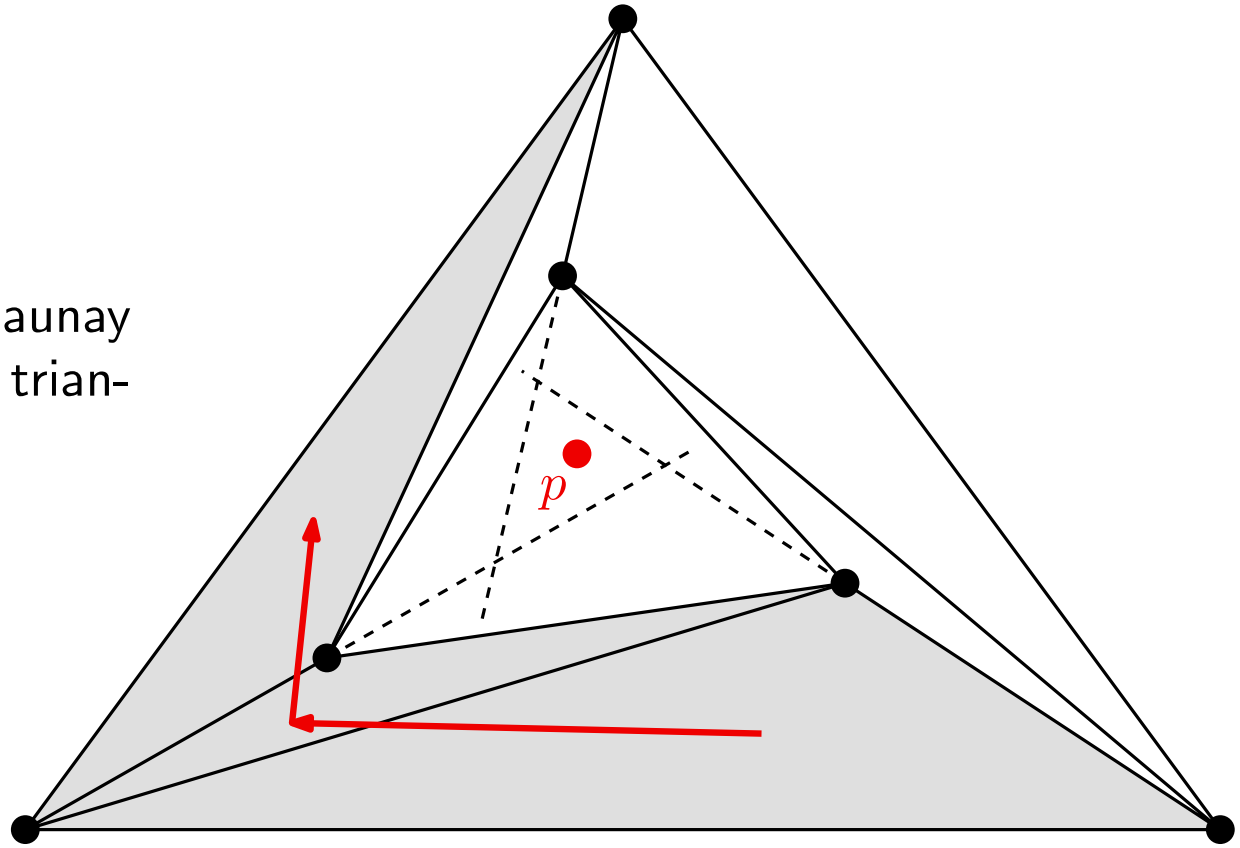
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Correction

- Rectilinear walk: obvious.
- Orthogonal walk: obvious.
- Visibility walk: works fine for Delaunay triangulations, but not for arbitrary triangulations.



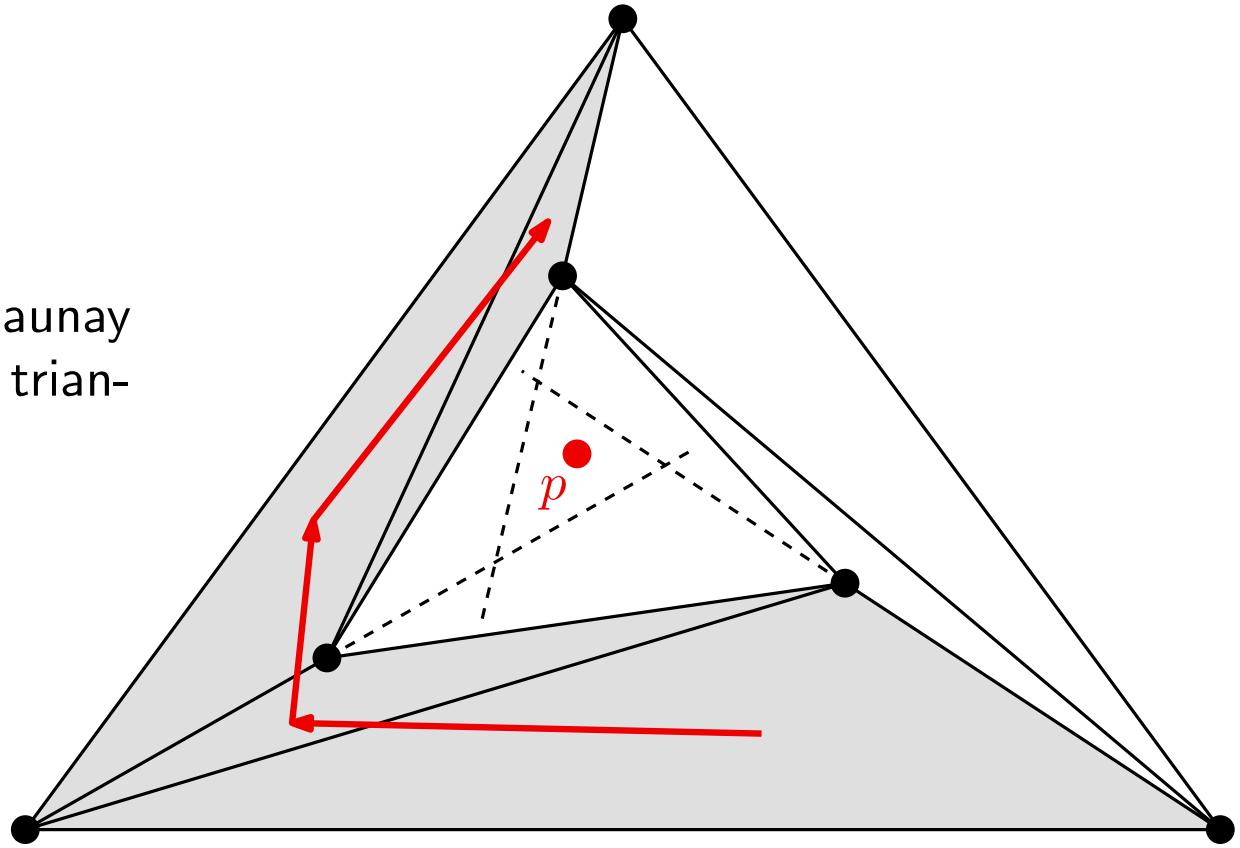
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Correction

- Rectilinear walk: obvious.
- Orthogonal walk: obvious.
- Visibility walk: works fine for Delaunay triangulations, but not for arbitrary triangulations.



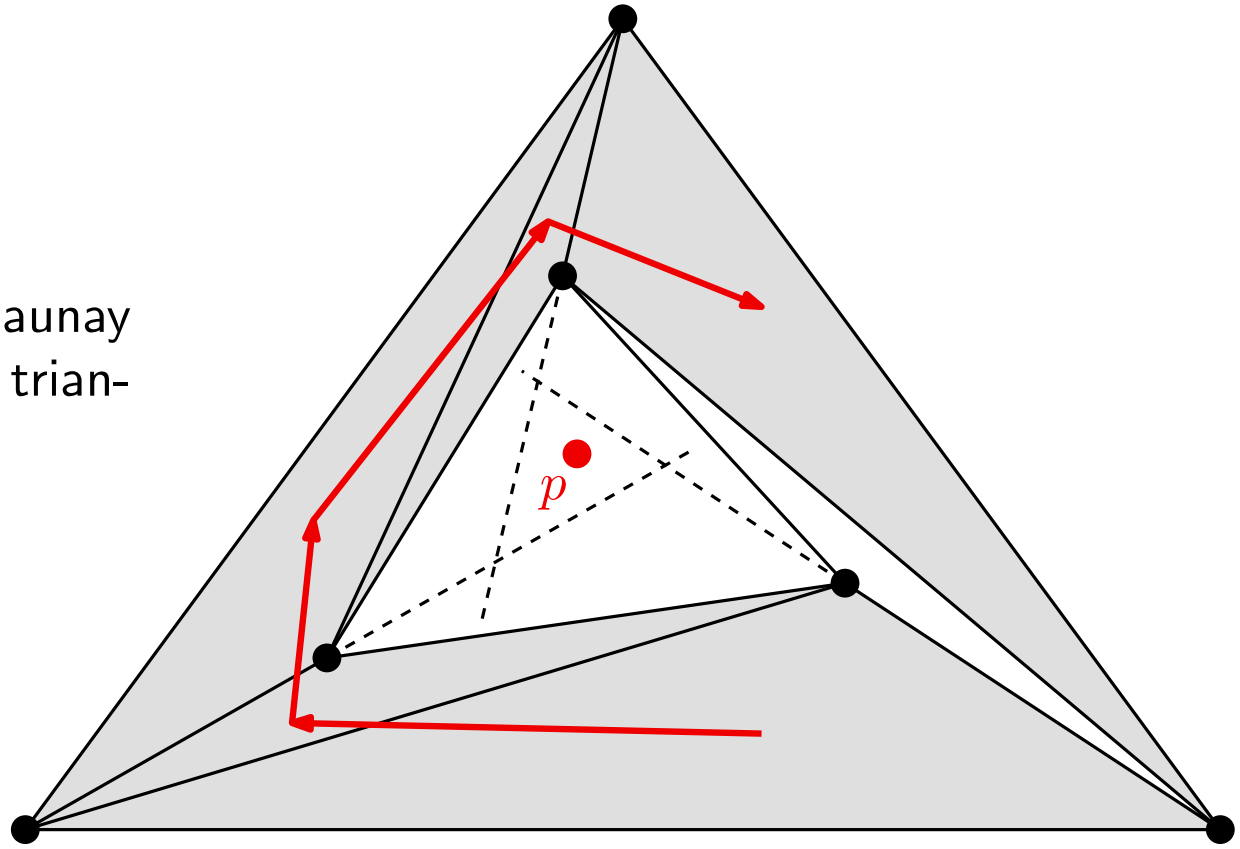
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Correction

- Rectilinear walk: obvious.
- Orthogonal walk: obvious.
- Visibility walk: works fine for Delaunay triangulations, but not for arbitrary triangulations.



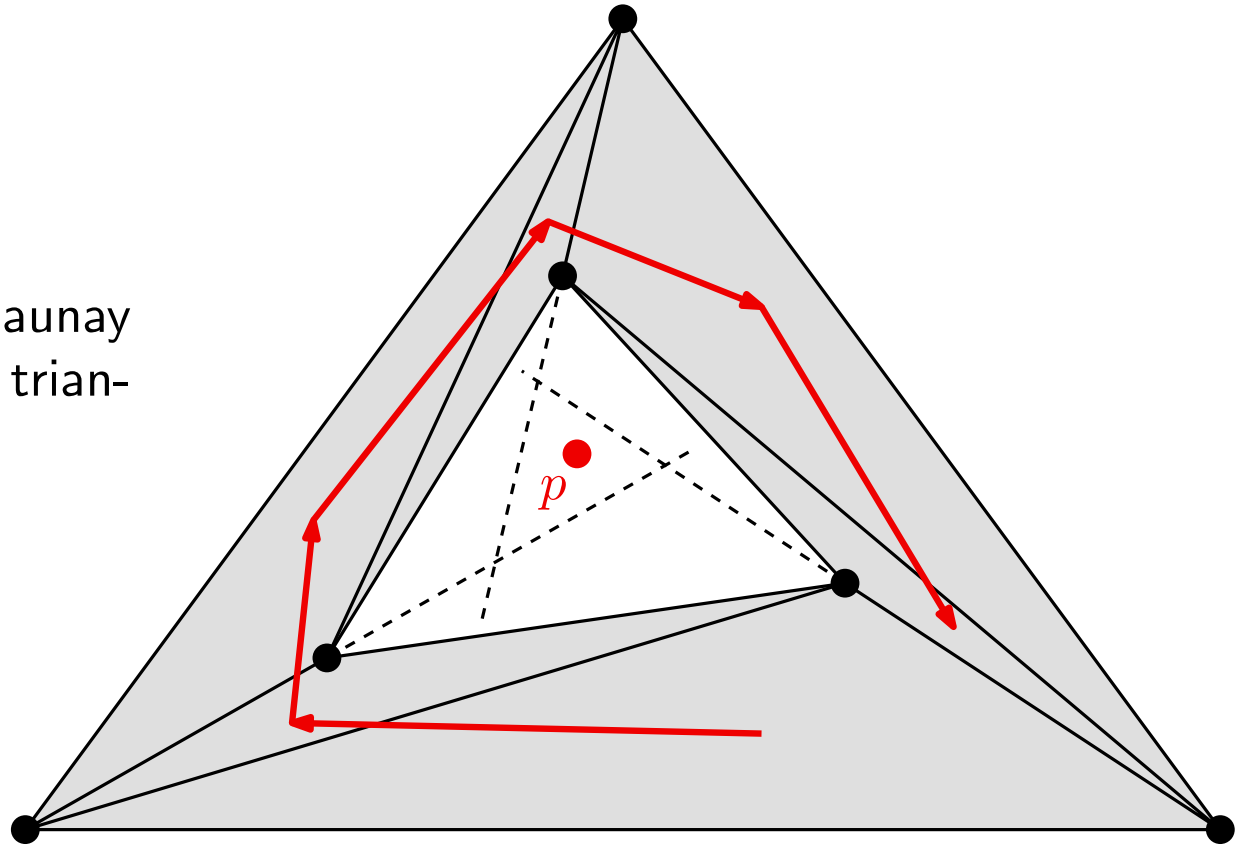
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Correction

- Rectilinear walk: obvious.
- Orthogonal walk: obvious.
- Visibility walk: works fine for Delaunay triangulations, but not for arbitrary triangulations.



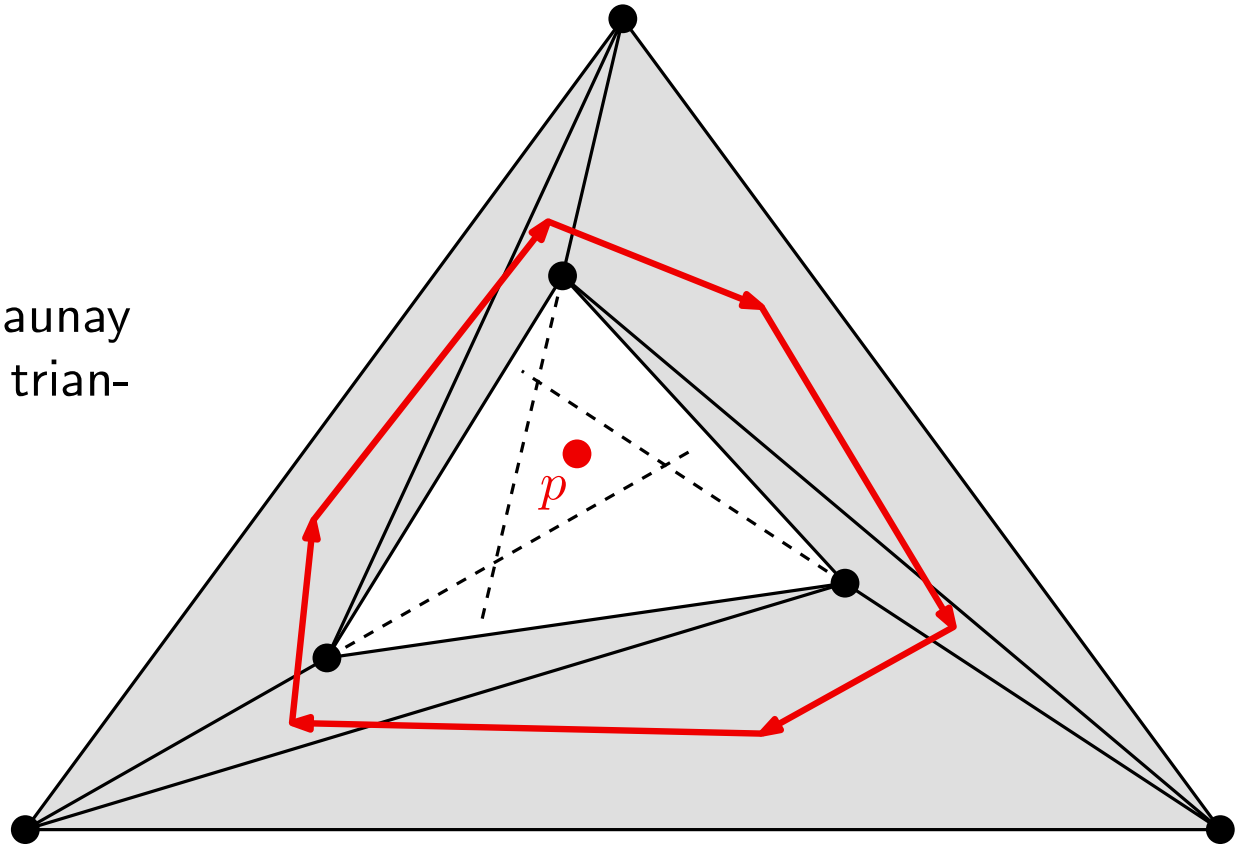
POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Correction

- Rectilinear walk: obvious.
- Orthogonal walk: obvious.
- Visibility walk: works fine for Delaunay triangulations, but not for arbitrary triangulations.



POINT LOCATION: Walking in a triangulation

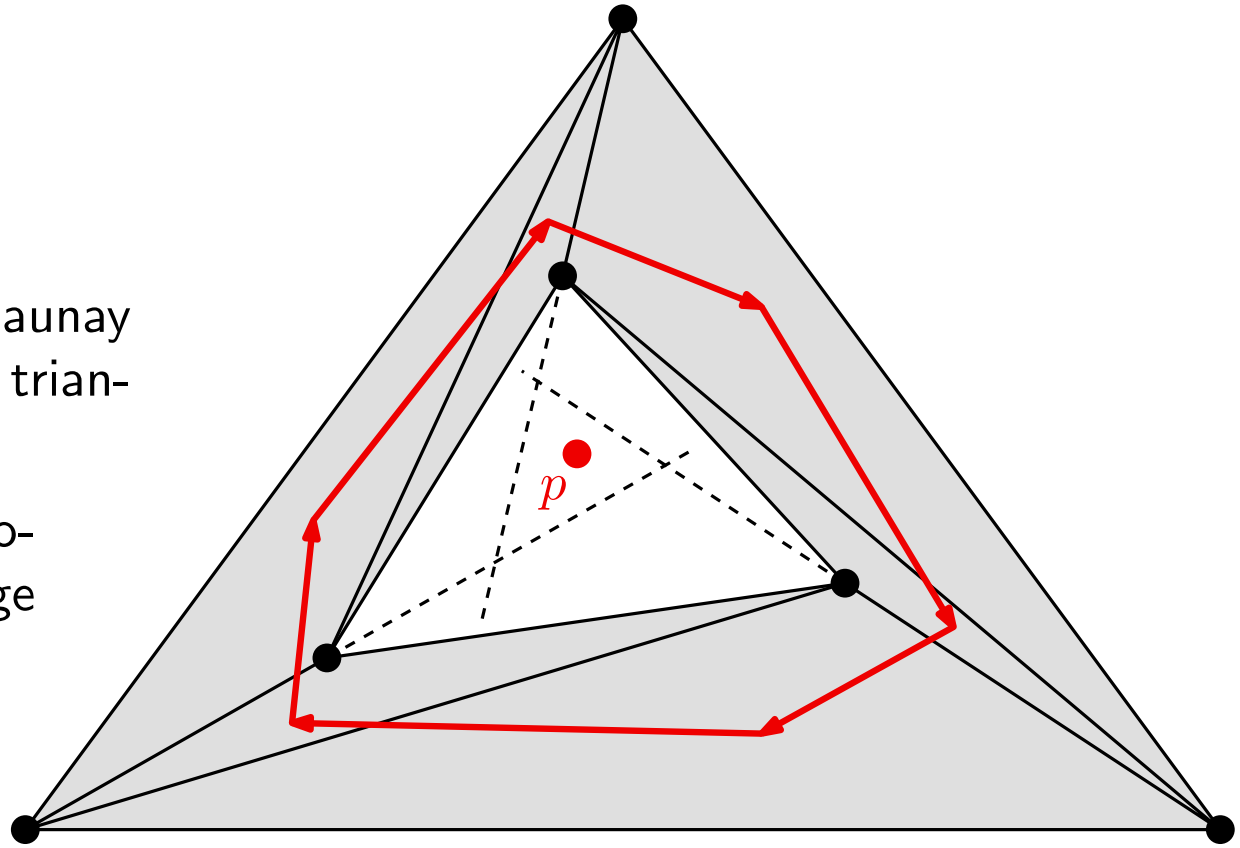
This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Correction

- Rectilinear walk: obvious.
- Orthogonal walk: obvious.
- Visibility walk: works fine for Delaunay triangulations, but not for arbitrary triangulations.

  This problem can be solved by randomizing the selection of the first edge to be tested on each triangle.



POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Advantages and disadvantages

- On average, rectilinear walks explore a smaller number of triangles.
- Orthogonal walks have the advantage that almost each test is in dimension 1. This is specially interesting when working in higher dimension.
- Visibility walks are easier to implement, because no degenerate positions need to be taken into account, as opposed to rectilinear walks, which require solving the case of the line-segment pq containing a vertex of the triangulation.
- Deterministic visibility walks cannot be applied to arbitrary triangulations if they are not Delaunay.
- All three walks can be generalized to higher dimension.

POINT LOCATION: Walking in a triangulation

This method is used for locating points in triangulations (suggested for Lab 4)

If finds out in which triangle a new point p lies, starting from a known vertex q of the triangulation.

Number of intersected triangles

For Delaunay triangulations on uniformly distributed random points, the *expected* number of visited triangles for each walk is:

- Rectilinear walk: $O(|p-q|\sqrt{n})$.
- Orthogonal walk: $O((|p|+|q|)\sqrt{n})$.
- Visibility walk: there exist triangulations for which the expected number is $> 2^{3\sqrt{n}}$.

In practice, in addition to the number of intersected triangles, the cost of each operation must be taken into account, as well as the effort of programming degenerated cases.

Finally, the choice of point q can speed up or slow down the process.

POINT LOCATION: Slab decomposition

Towards more general and efficient methods

Method 1: **Slab decomposition**

POINT LOCATION: Slab decomposition

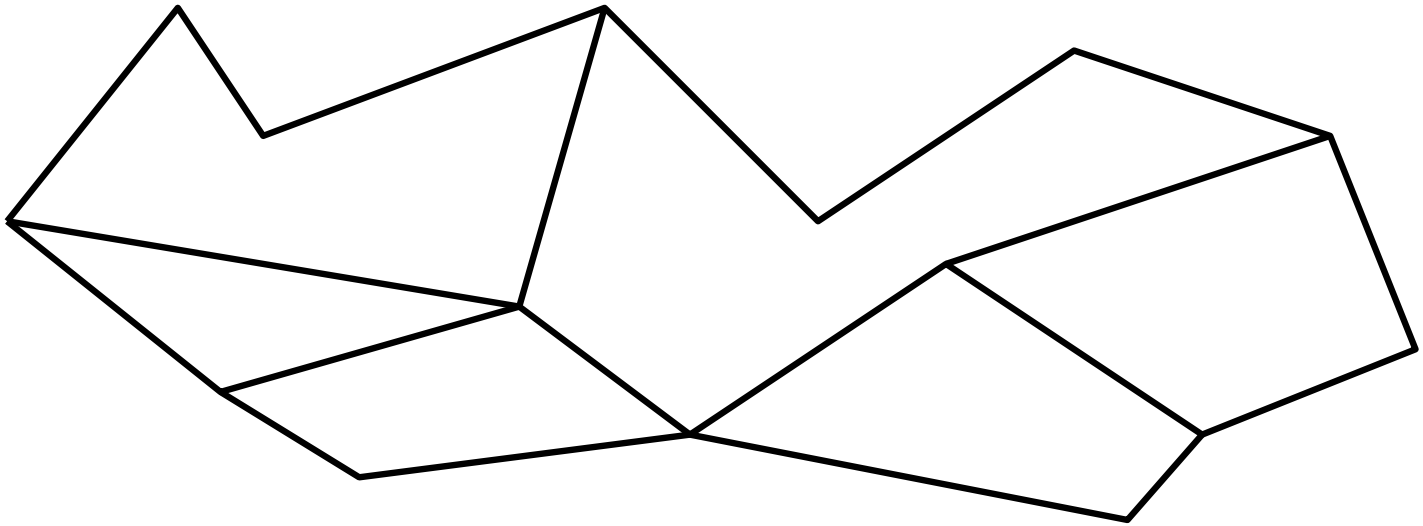
Preprocessing

Decompose the plane into the slabs determined by all vertical lines through the vertices of the original decomposition.

POINT LOCATION: Slab decomposition

Preprocessing

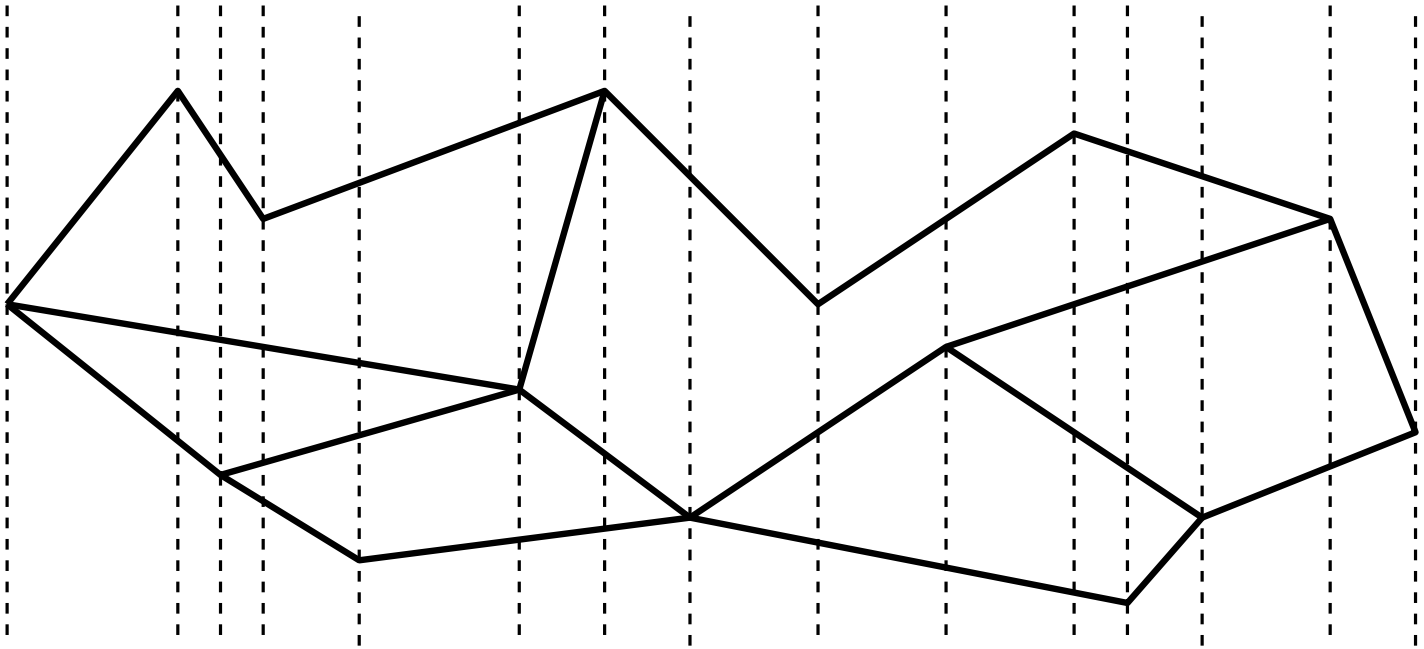
Decompose the plane into the slabs determined by all vertical lines through the vertices of the original decomposition.



POINT LOCATION: Slab decomposition

Preprocessing

Decompose the plane into the slabs determined by all vertical lines through the vertices of the original decomposition.



POINT LOCATION: Slab decomposition

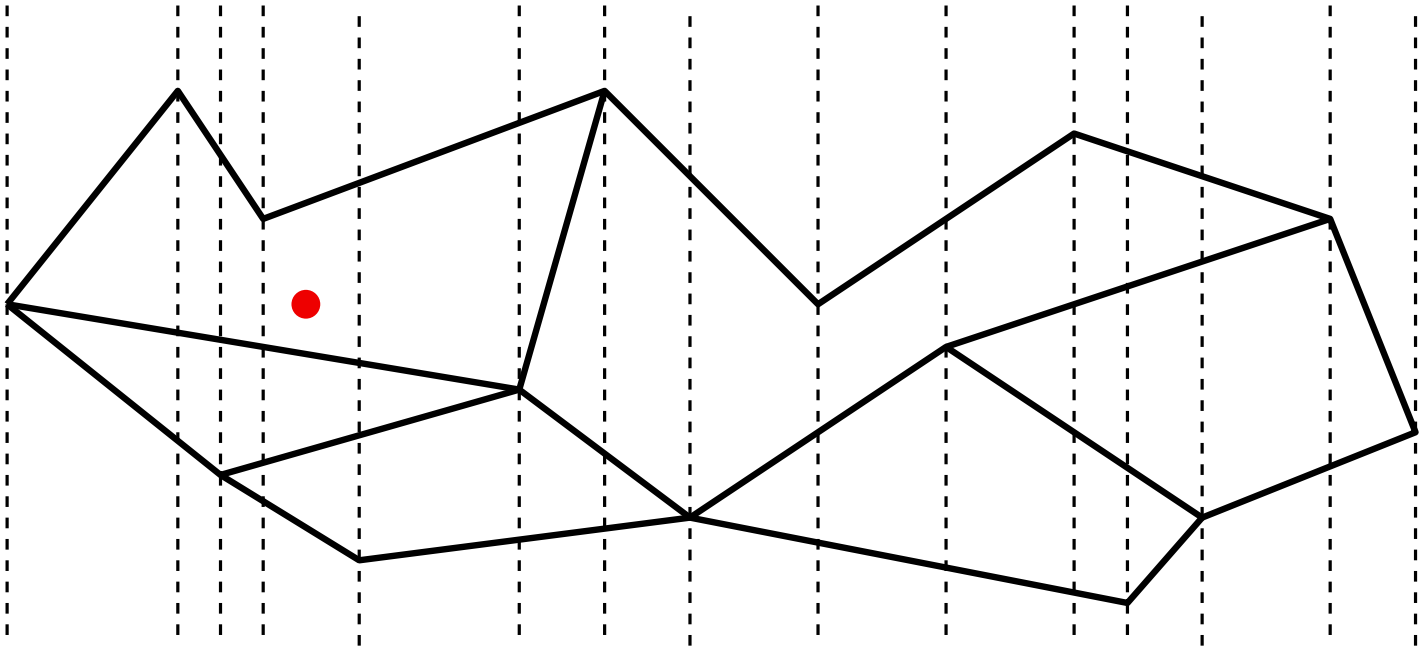
Preprocessing

Decompose the plane into the slabs determined by all vertical lines through the vertices of the original decomposition.

Location

Given a point q:

1. Locate the abscsissa of q in the corresponding slab.
2. Within the slab, locate the two segments between which q lies.



POINT LOCATION: Slab decomposition

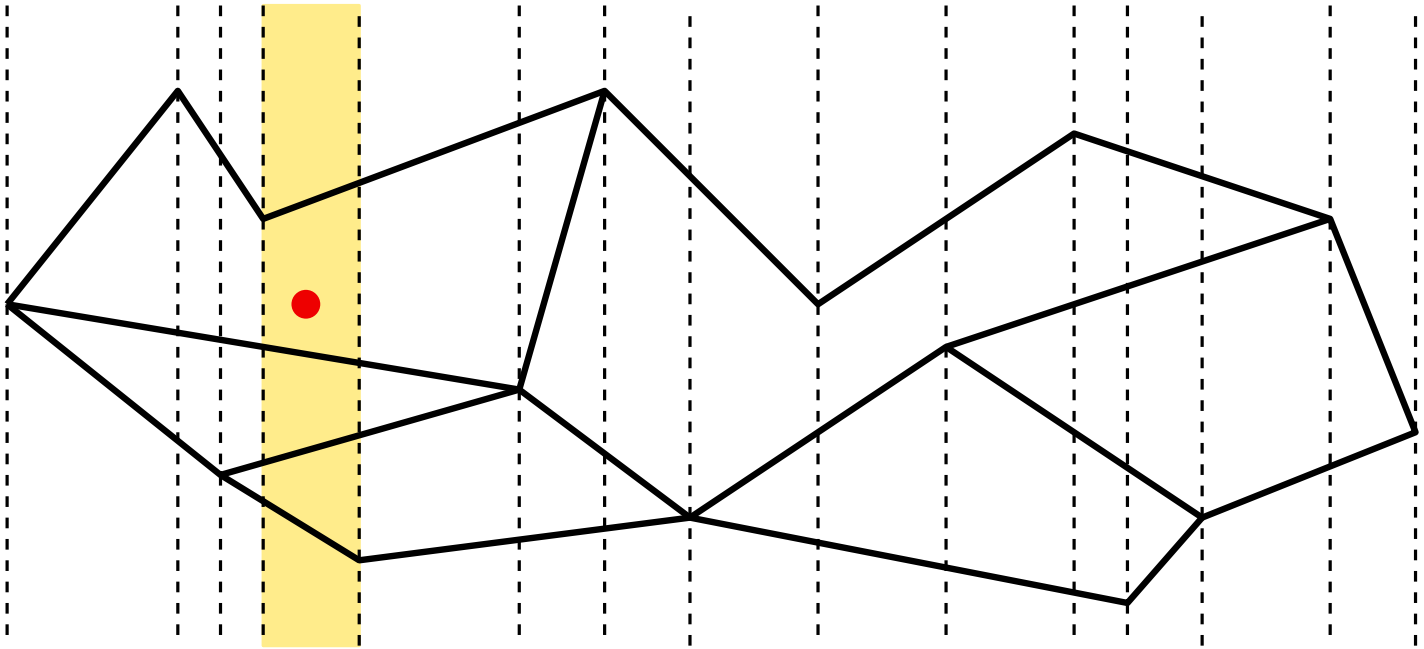
Preprocessing

Decompose the plane into the slabs determined by all vertical lines through the vertices of the original decomposition.

Location

Given a point q:

1. Locate the abscsissa of q in the corresponding slab.
2. Within the slab, locate the two segments between which q lies.



POINT LOCATION: Slab decomposition

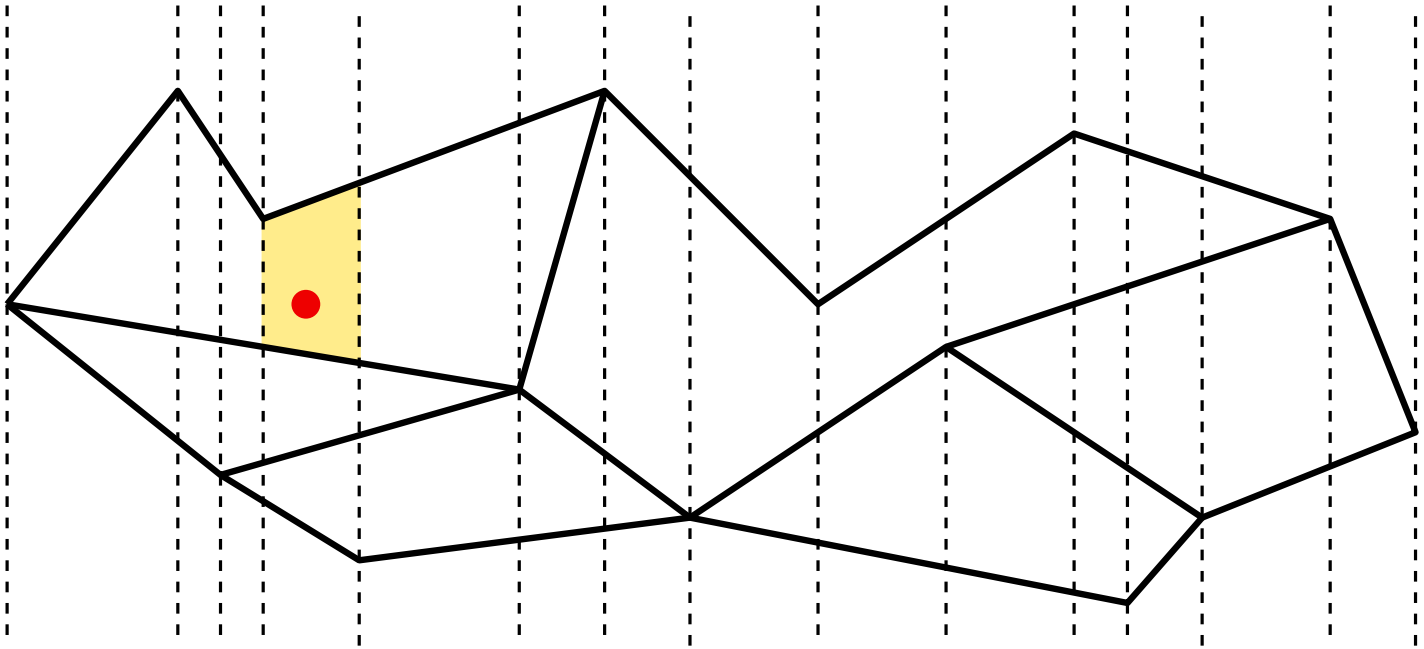
Preprocessing

Decompose the plane into the slabs determined by all vertical lines through the vertices of the original decomposition.

Location

Given a point q:

1. Locate the abscsissa of q in the corresponding slab.
2. Within the slab, locate the two segments between which q lies.



POINT LOCATION: Slab decomposition

Preprocessing

Decompose the plane into the slabs determined by all vertical lines through the vertices of the original decomposition.

Location

Given a point q:

1. Locate the abscsissa of q in the corresponding slab.
2. Within the slab, locate the two segments between which q lies.

The preprocessing should:

- Store the x-coordinate of the slabs in a structure allowing binary searching.
- Store the segments intersecting each slab in a structure allowing binary searching.
- For each segment of the initial decomposition, store a pointer to the face above (or below) it.

In this way, it will be possible to perform each point location in $O(\log n)$ time.

POINT LOCATION: Slab decomposition

Preprocessing

It can be done by sweeping the planar decomposition with a vertical line:

Events queue: the vertices of the decomposition, sorted by their x-coordinate

Sweep line status: the line segments of the decomposition stabbed by the line, in order.

Action at each event:

1. In the sweep line:
    - Insert vertex: insert the incident edges in the sweep line, in counterclockwise order.
    - Delete vertex: delete the incident edges from the sweep line.
    - Update vertex: delete the edges incident to the left and insert the edges incident to the right, in sorted order.
2. In the slab decomposition:
    - Store the line segments of the new slab in order.

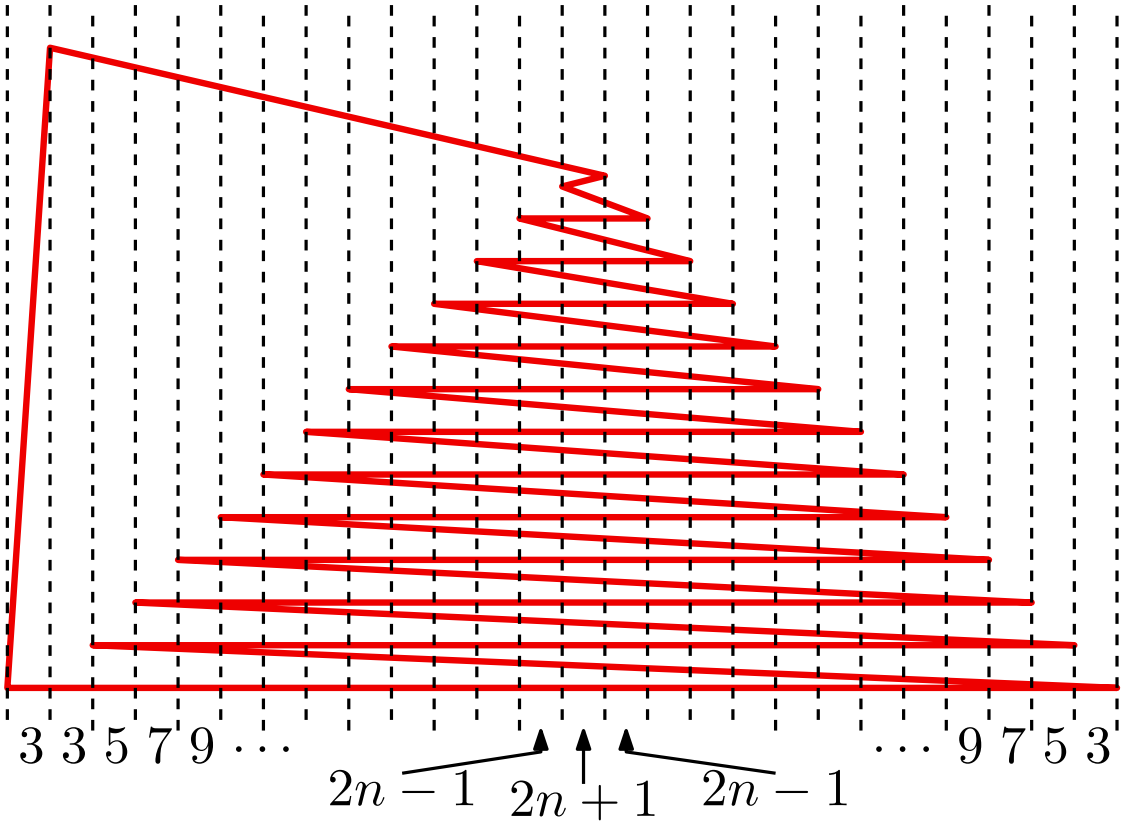
POINT LOCATION: Slab decomposition

Complexity

POINT LOCATION: Slab decomposition

Complexity

Some planar decompositions have a quadratic-size slab decomposition:



In this example, if the total number of vertices is $2n+1$, the total number of regions of the slab decomposition is

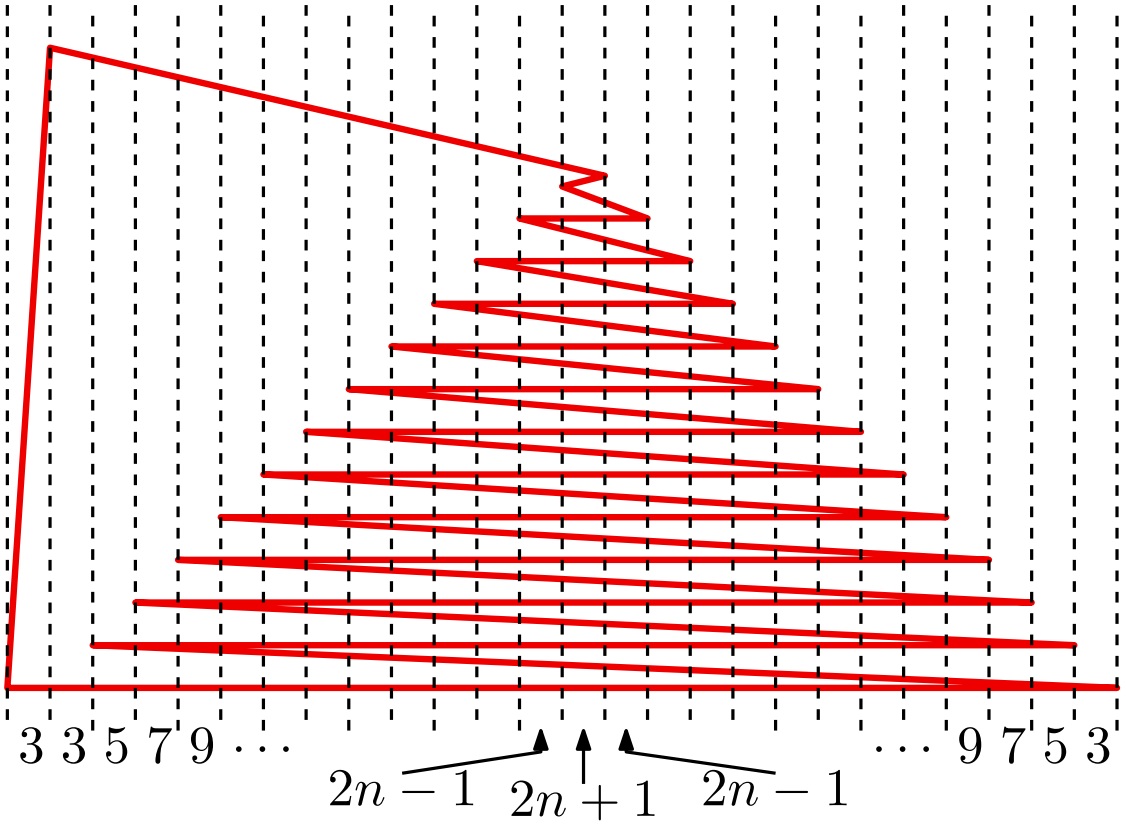
$$3 + (2n+1) + 2\sum_{i=1}^{n-1}(2i+1) \geq 2\sum_{i=1}^{n-1} 2i =$$

$$= 4\sum_{i=1}^{n-1} i = 4\frac{n(n-1)}{2} = \Omega(n^2).$$

POINT LOCATION: Slab decomposition

Complexity

Some planar decompositions have a quadratic-size slab decomposition:



In this example, if the total number of vertices is $2n+1$, the total number of regions of the slab decomposition is

$$3 + (2n+1) + 2\sum_{i=1}^{n-1}(2i+1) \geq 2\sum_{i=1}^{n-1} 2i =$$

$$= 4\sum_{i=1}^{n-1} i = 4\frac{n(n-1)}{2} = \Omega(n^2).$$

Space: the space needed to store the information is $O(n^2)$.

Preprocessing: the preprocessing is done in $O(n^2)$ time.

Location: locating a point is done in $O(\log n)$ time.

Method 2: **Monotone subdivision**

POINT LOCATION: Monotone subdivision

This method is based on the following observation: if C is an x-monotone chain with n vertices, deciding whether a point q lies above or below C can be done in $O(\log n)$ time.

1. Locate the x-coordinate q_x of q between the x-coordinates x_i and x_{i+1} of two consecutive vertices of C, by binary search.

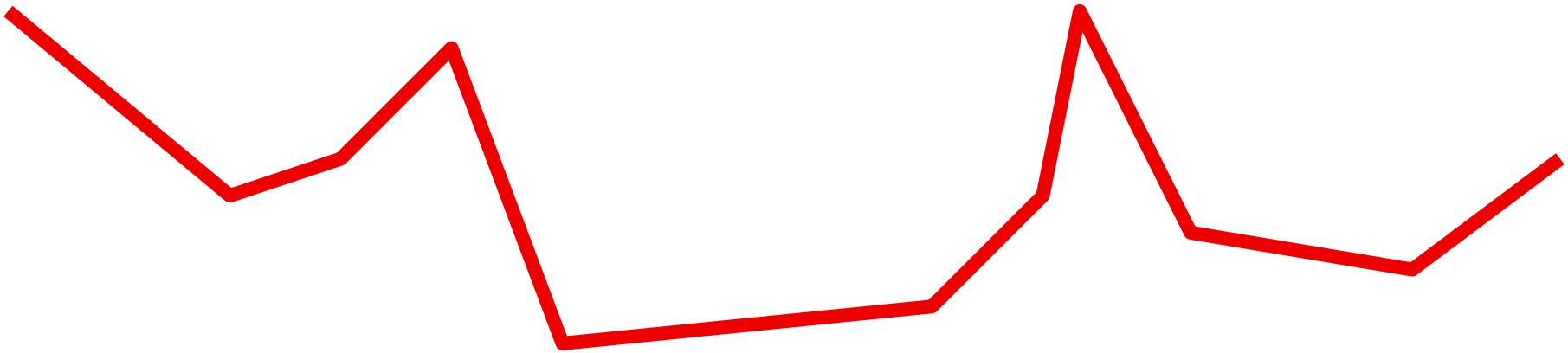
2. Decide whether q lies above or below the line-segment $p_i p_{i+1}$ of C.

POINT LOCATION: Monotone subdivision

This method is based on the following observation: if C is an x-monotone chain with n vertices, deciding whether a point q lies above or below C can be done in $O(\log n)$ time.

1. Locate the x-coordinate q_x of q between the x-coordinates x_i and x_{i+1} of two consecutive vertices of C, by binary search.

2. Decide whether q lies above or below the line-segment $p_i p_{i+1}$ of C.

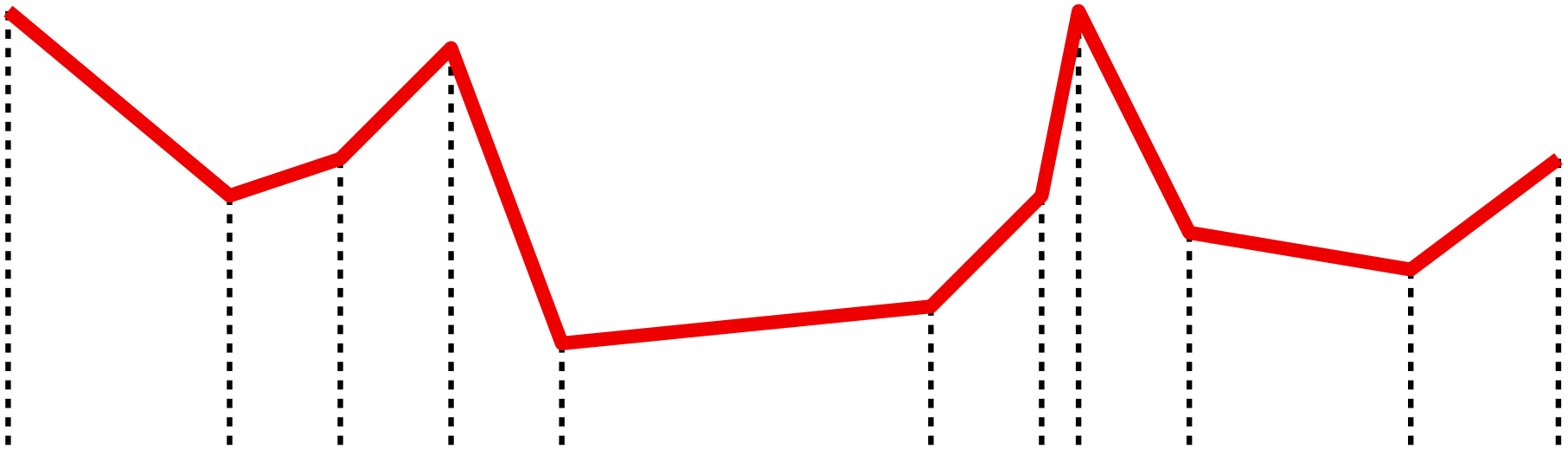


POINT LOCATION: Monotone subdivision

This method is based on the following observation: if C is an x-monotone chain with n vertices, deciding whether a point q lies above or below C can be done in $O(\log n)$ time.

1. Locate the x-coordinate q_x of q between the x-coordinates x_i and x_{i+1} of two consecutive vertices of C, by binary search.

2. Decide whether q lies above or below the line-segment $p_i p_{i+1}$ of C.

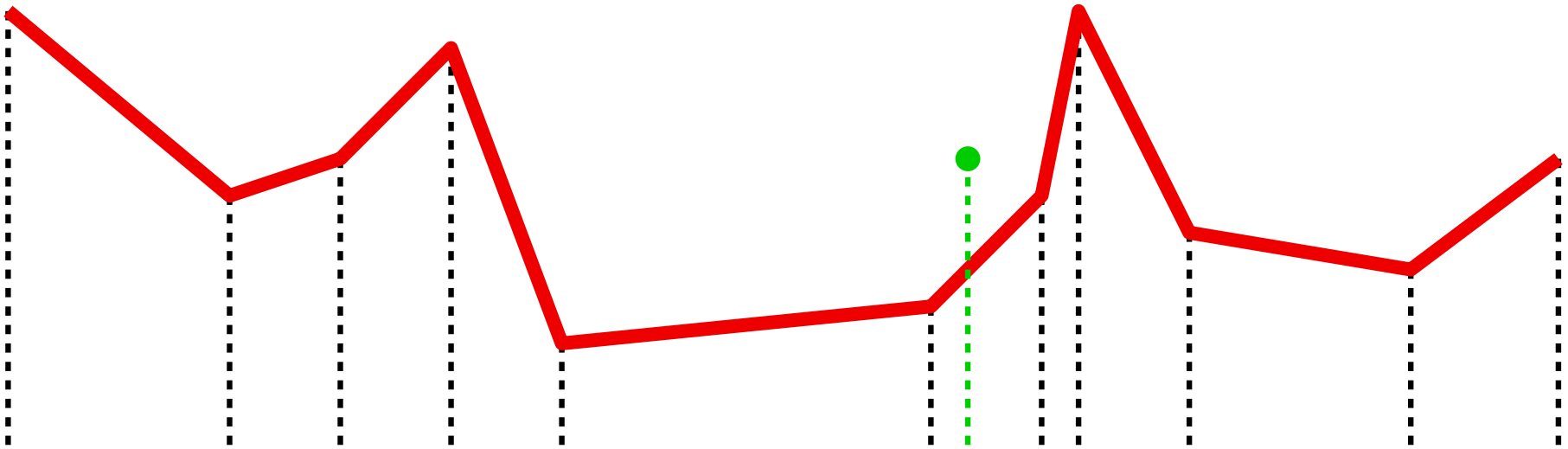


POINT LOCATION: Monotone subdivision

This method is based on the following observation: if C is an x-monotone chain with n vertices, deciding whether a point q lies above or below C can be done in $O(\log n)$ time.

1. Locate the x-coordinate q_x of q between the x-coordinates x_i and x_{i+1} of two consecutive vertices of C, by binary search.

2. Decide whether q lies above or below the line-segment $p_i p_{i+1}$ of C.

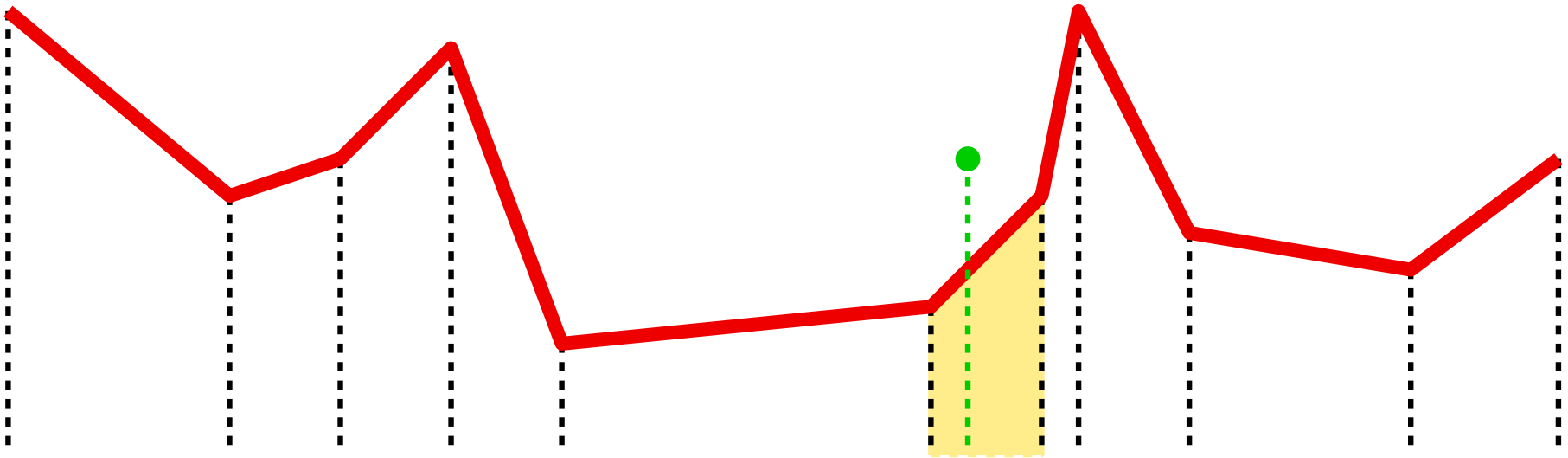


POINT LOCATION: Monotone subdivision

This method is based on the following observation: if C is an x-monotone chain with n vertices, deciding whether a point q lies above or below C can be done in $O(\log n)$ time.

1. Locate the x-coordinate q_x of q between the x-coordinates x_i and x_{i+1} of two consecutive vertices of C, by binary search.

2. Decide whether q lies above or below the line-segment $p_i p_{i+1}$ of C.



POINT LOCATION: Monotone subdivision

Preprocessing

Compute $\mathcal{C}$, a complete set of x-monotone chains for the planar decomposition:

- The union of the chains of $\mathcal{C}$ contains the 1-skeleton of the decomposition.
- If C_i and C_j are two chains of $\mathcal{C}$, all the vertices of C_i that do not belong to C_j lie on the same side of C_j. This implies that $\mathcal{C}$ is a totally ordered set.

Therefore, if $\mathcal{C}$ contains r chains, and the largest of the chains has k vertices, it is possible to locate a point between two consecutive chains in $O(\log r \log k)$ time by binary search.

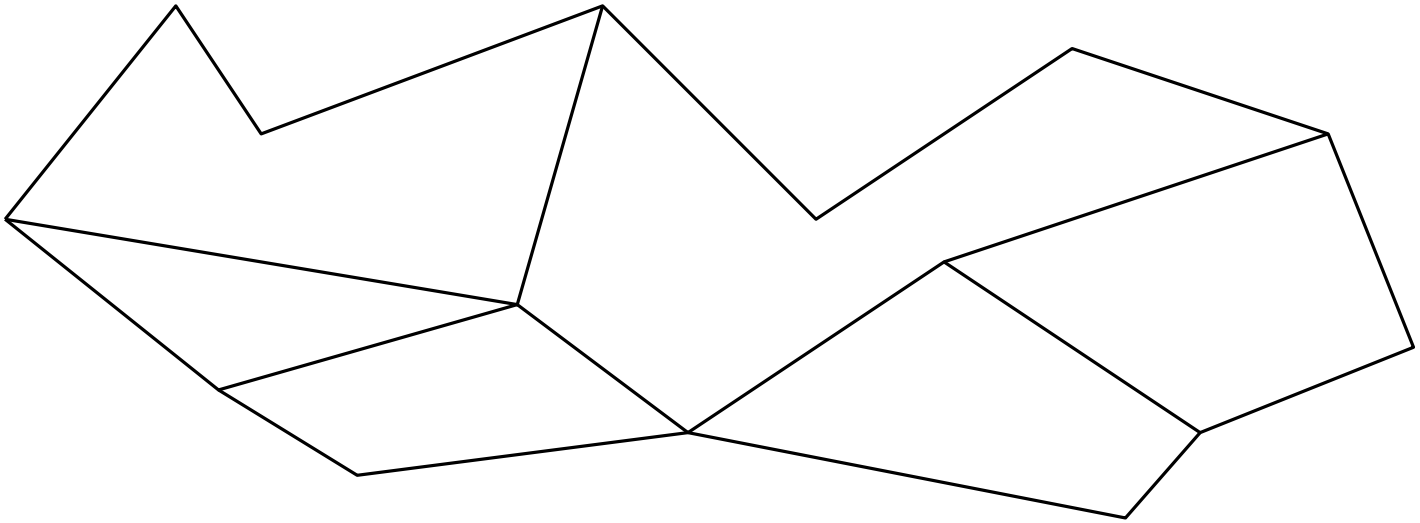
POINT LOCATION: Monotone subdivision

Preprocessing

Compute $\mathcal{C}$, a complete set of x-monotone chains for the planar decomposition:

- The union of the chains of $\mathcal{C}$ contains the 1-skeleton of the decomposition.
- If C_i and C_j are two chains of $\mathcal{C}$, all the vertices of C_i that do not belong to C_j lie on the same side of C_j. This implies that $\mathcal{C}$ is a totally ordered set.

Therefore, if $\mathcal{C}$ contains r chains, and the largest of the chains has k vertices, it is possible to locate a point between two consecutive chains in $O(\log r \log k)$ time by binary search.



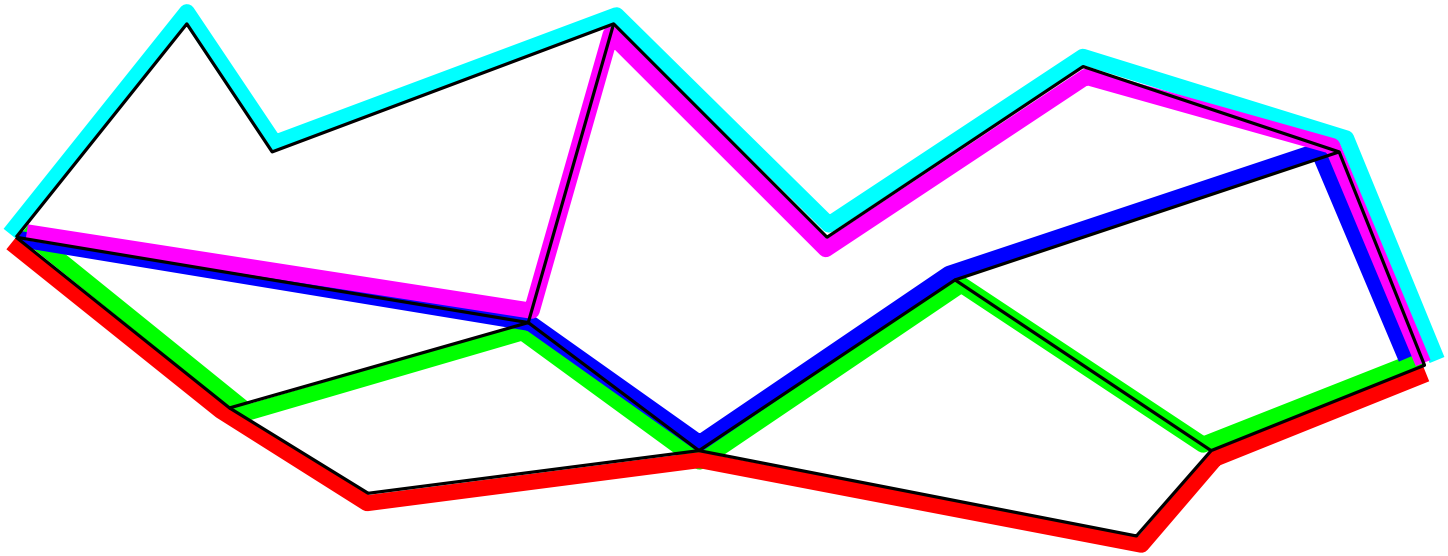
POINT LOCATION: Monotone subdivision

Preprocessing

Compute $\mathcal{C}$, a complete set of x-monotone chains for the planar decomposition:

- The union of the chains of $\mathcal{C}$ contains the 1-skeleton of the decomposition.
- If C_i and C_j are two chains of $\mathcal{C}$, all the vertices of C_i that do not belong to C_j lie on the same side of C_j. This implies that $\mathcal{C}$ is a totally ordered set.

Therefore, if $\mathcal{C}$ contains r chains, and the largest of the chains has k vertices, it is possible to locate a point between two consecutive chains in $O(\log r \log k)$ time by binary search.



POINT LOCATION: Monotone subdivision

Conditions

For a planar decomposition to admit a complete set of x-monotone chains, the graph G is required to be *regular*:

- Consider the vertices $v_1, v_2, \ldots, v_n$ of G to be lexicographically sorted:

$$i < j \iff x(v_i) < x(v_j) \text{ or } (x(v_i) = x(v_j) \text{ and } y(v_i) < y(v_j)).$$

- A vertex v_i is regular if $\exists\ j, k$ with $j < i < k$ such that $\overline{v_j v_i}$ and $\overline{v_i v_k}$ are edges of G.
- The graph G is regular if all its vertices, other than v_1 and v_n, are regular.

POINT LOCATION: Monotone subdivision

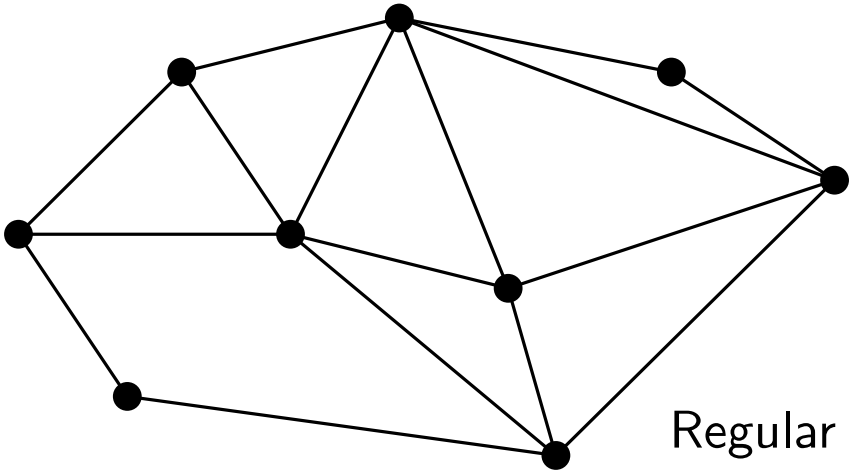
Conditions

For a planar decomposition to admit a complete set of x-monotone chains, the graph G is required to be *regular*:

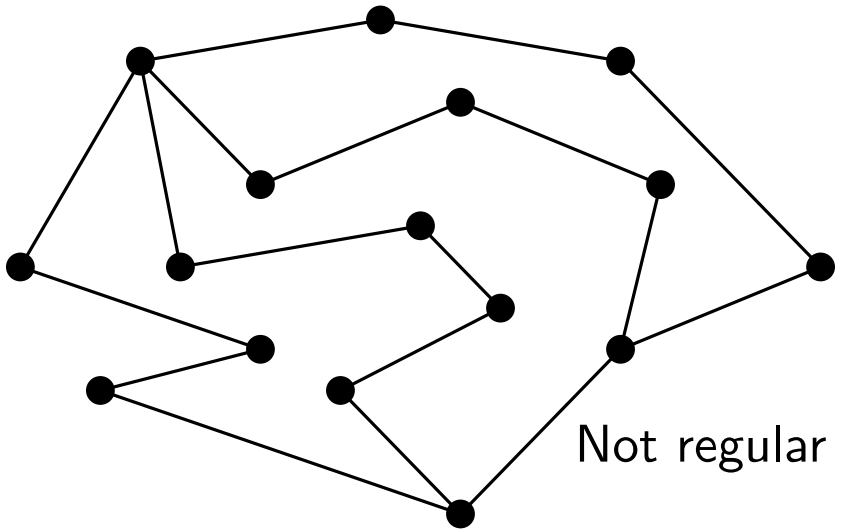
- Consider the vertices $v_1, v_2, \ldots, v_n$ of G to be lexicographically sorted:

$$i < j \iff x(v_i) < x(v_j) \text{ or } (x(v_i) = x(v_j) \text{ and } y(v_i) < y(v_j)).$$

- A vertex v_i is regular if $\exists\ j, k$ with $j < i < k$ such that $\overline{v_j v_i}$ and $\overline{v_i v_k}$ are edges of G.
- The graph G is regular if all its vertices, other than v_1 and v_n, are regular.



Regular



Not regular

POINT LOCATION: Monotone subdivision

Theorem

If G is a regular graph, then it admits a complete set of x-monotone chains.

POINT LOCATION: Monotone subdivision

Theorem

If G is a regular graph, then it admits a complete set of x-monotone chains.

Proof:

1. By induction, we will prove that it is possible to compute an x-monotone chain from v_1 to v_i for all i:
   - For $i = 2$ the result is trivial, because $\exists\ j < 2$ such that $\overline{v_j v_2}$ is an edge of G.
   - Assume the result true for all $j < i$. Due to the regularity of v_i, $\exists\ j < i$ such that $\overline{v_j v_i}$ is an edge of G. By induction hypothesis, there exists an x-monotone chain C from v_1 to v_j. The concatenation of C and $\overline{v_j v_i}$ is x-monotone. Since this can be repeated for every v_j that is a neighbor of v_i with $j < i$, there is a complete set of x-monotone chains from v_1 to v_i.
2. In the following, we will prove that it is possible to build a complete set $\mathcal{C}$ of x-monotone chains.

POINT LOCATION: Monotone subdivision

Theorem

Notation:

- $in(v_i) = \{\text{edges } \overline{v_j v_i} \text{ of } G \text{ with } j < i\}$.
- $out(v_i) = \{\text{edges } \overline{v_i v_k} \text{ of } G \text{ with } i < k\}$.
- $w(e) =$ weight of edge e, defined as number of chains of $\mathcal{C}$ where e appears.
- $w_{in}(v) =$ ingoing weight of vertex v, i.e., $= \sum_{e \in in(v)} w(e)$.
- $w_{out}(v) =$ outgoing weight of vertex v, i.e., $= \sum_{e \in out(v)} w(e)$.

POINT LOCATION: Monotone subdivision

Theorem

Notation:

- $in(v_i) = \{\text{edges } \overline{v_j v_i} \text{ of } G \text{ with } j < i\}$.
- $out(v_i) = \{\text{edges } \overline{v_i v_k} \text{ of } G \text{ with } i < k\}$.
- $w(e) =$ weight of edge e, defined as number of chains of $\mathcal{C}$ where e appears.
- $w_{in}(v) =$ ingoing weight of vertex v, i.e., $= \sum_{e \in in(v)} w(e)$.
- $w_{out}(v) =$ outgoing weight of vertex v, i.e., $= \sum_{e \in out(v)} w(e)$.

It can be proved:

- $\forall e \quad w(e) > 0$.

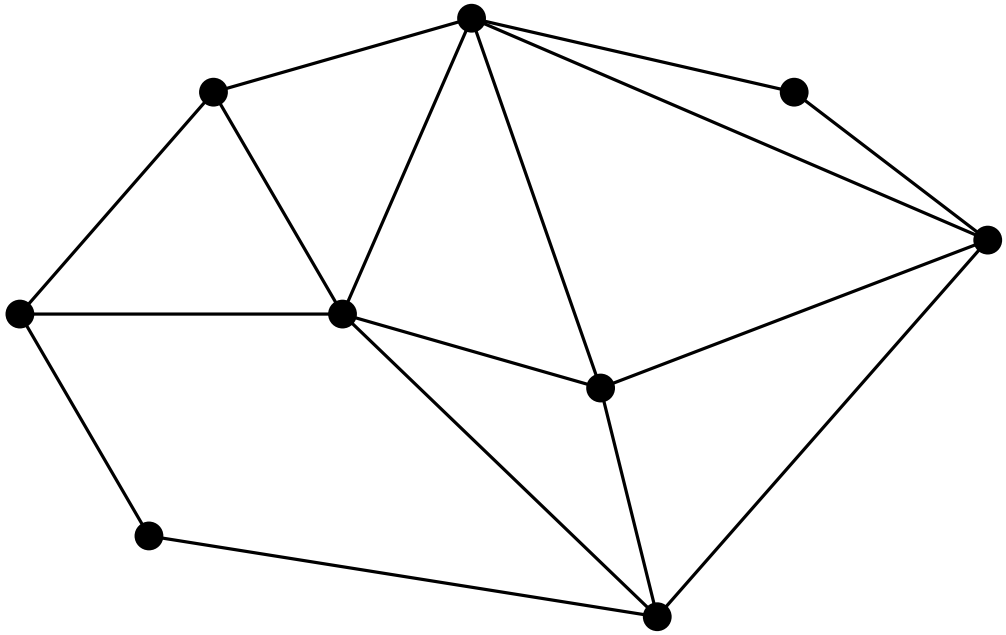
  This implies that the union of the chains of $\mathcal{C}$ contains the 1-skeleton of G.
- $\forall i \neq 1, n \quad w_{in}(v_i) = w_{out}(v_i)$.

  This implies that $\mathcal{C}$ can be constructed so that it is a totally ordered set.

POINT LOCATION: Monotone subdivision

Theorem

Weight assignment:



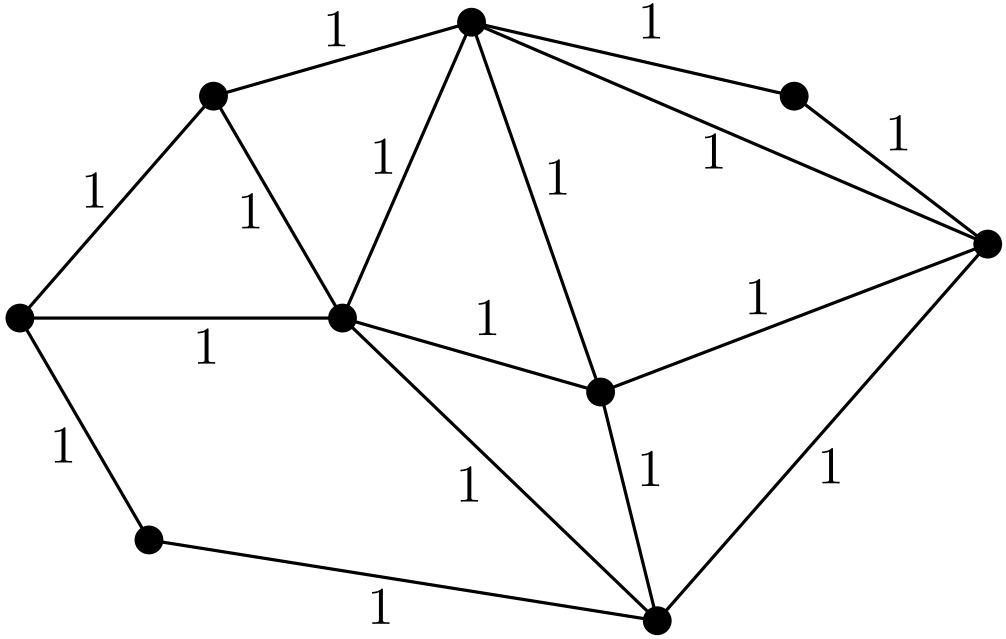
POINT LOCATION: Monotone subdivision

Theorem

Weight assignment:

1. *Initialization*

   Assign weight 1 to all edges



POINT LOCATION: Monotone subdivision

Theorem

Weight assignment:

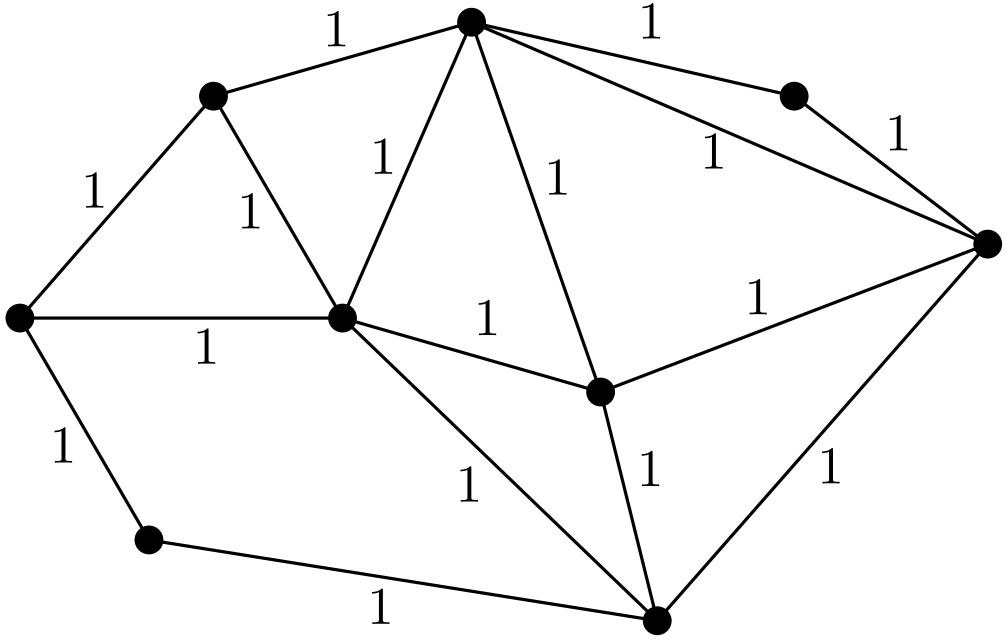
1. *Initialization*

   Assign weight 1 to all edges

2. *First sweep (forwards)*

   From $i = 2$ to $i = n - 1$ do:

   If $w_{in}(v_i) > w_{out}(v_i)$, replace the weight 1 of the (counterclockwise) first outgoing edge of v_i by the weight $w_{in}(v_i) - w_{out}(v_i) + 1$.



POINT LOCATION: Monotone subdivision

Theorem

Weight assignment:

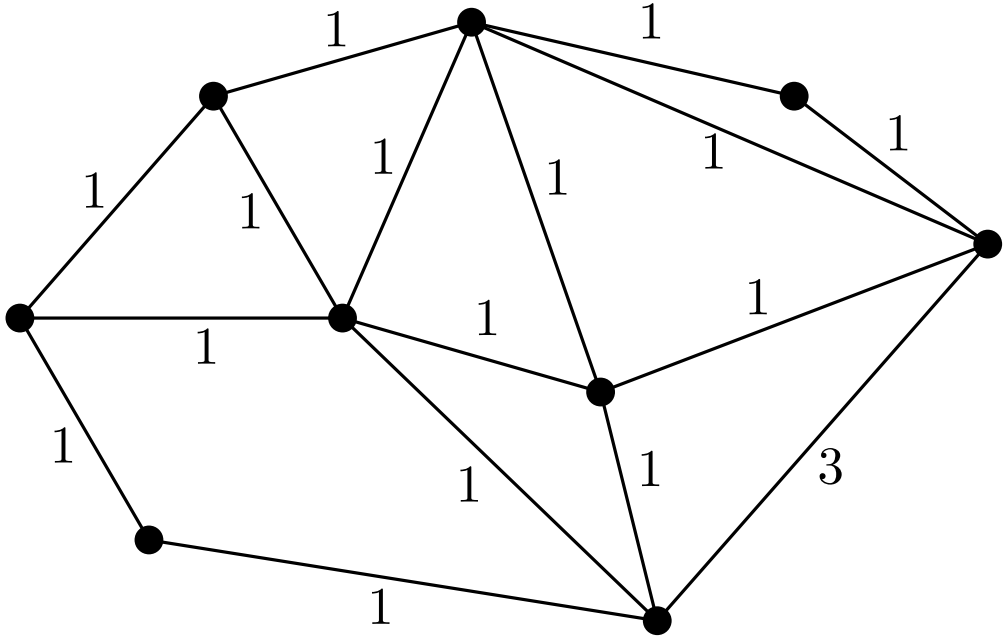
1. *Initialization*

   Assign weight 1 to all edges

2. *First sweep (forwards)*

   From $i = 2$ to $i = n - 1$ do:

   If $w_{in}(v_i) > w_{out}(v_i)$, replace the weight 1 of the (counterclockwise) first outgoing edge of v_i by the weight $w_{in}(v_i) - w_{out}(v_i) + 1$.



POINT LOCATION: Monotone subdivision

Theorem

Weight assignment:

1. *Initialization*

   Assign weight 1 to all edges

2. *First sweep (forwards)*

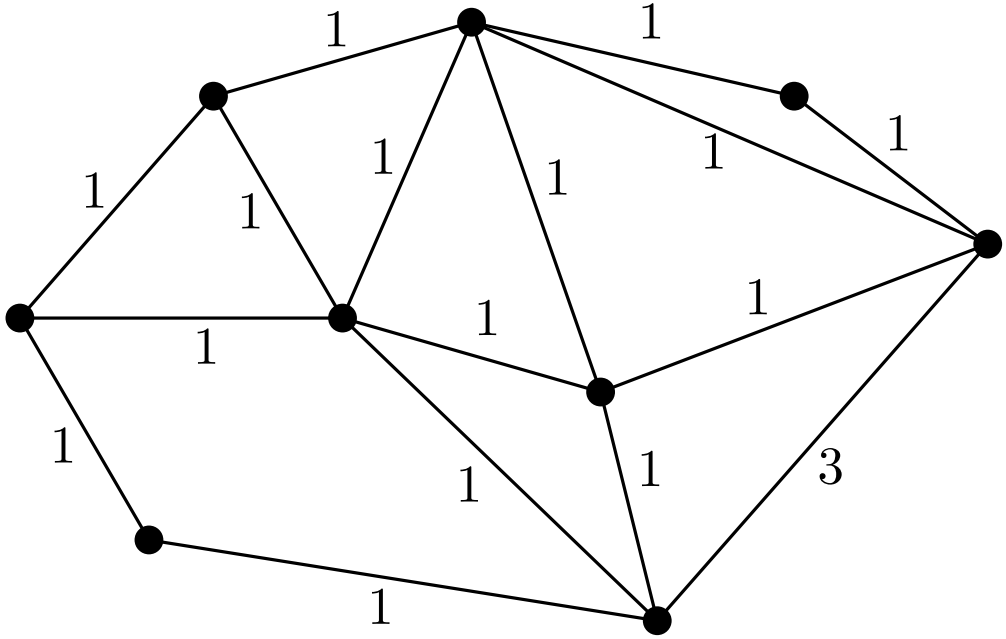
   From $i = 2$ to $i = n - 1$ do:

   If $w_{in}(v_i) > w_{out}(v_i)$, replace the weight 1 of the (counterclockwise) first outgoing edge of v_i by the weight $w_{in}(v_i) - w_{out}(v_i) + 1$.

3. *Second sweep (backwards)*

   From $i = n - 1$ to $i = 2$ do:

   If $w_{out}(v_i) > w_{in}(v_i)$, replace the weight $w(e)$ of the (counterclockwise) first ingoing edge of v_i by the weight $w_{out}(v_i) - w_{in}(v_i) + w(e)$.



POINT LOCATION: Monotone subdivision

Theorem

Weight assignment:

1. *Initialization*

   Assign weight 1 to all edges

2. *First sweep (forwards)*

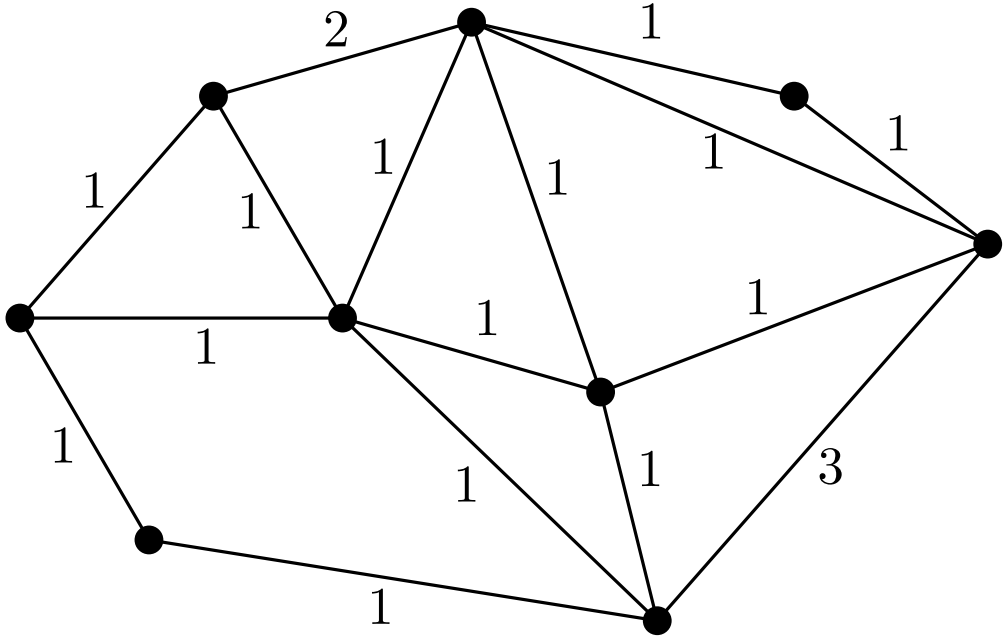
   From $i = 2$ to $i = n - 1$ do:

   If $w_{in}(v_i) > w_{out}(v_i)$, replace the weight 1 of the (counterclockwise) first outgoing edge of v_i by the weight $w_{in}(v_i) - w_{out}(v_i) + 1$.

3. *Second sweep (backwards)*

   From $i = n - 1$ to $i = 2$ do:

   If $w_{out}(v_i) > w_{in}(v_i)$, replace the weight $w(e)$ of the (counterclockwise) first ingoing edge of v_i by the weight $w_{out}(v_i) - w_{in}(v_i) + w(e)$.



POINT LOCATION: Monotone subdivision

Theorem

Weight assignment:

1. *Initialization*

   Assign weight 1 to all edges

2. *First sweep (forwards)*

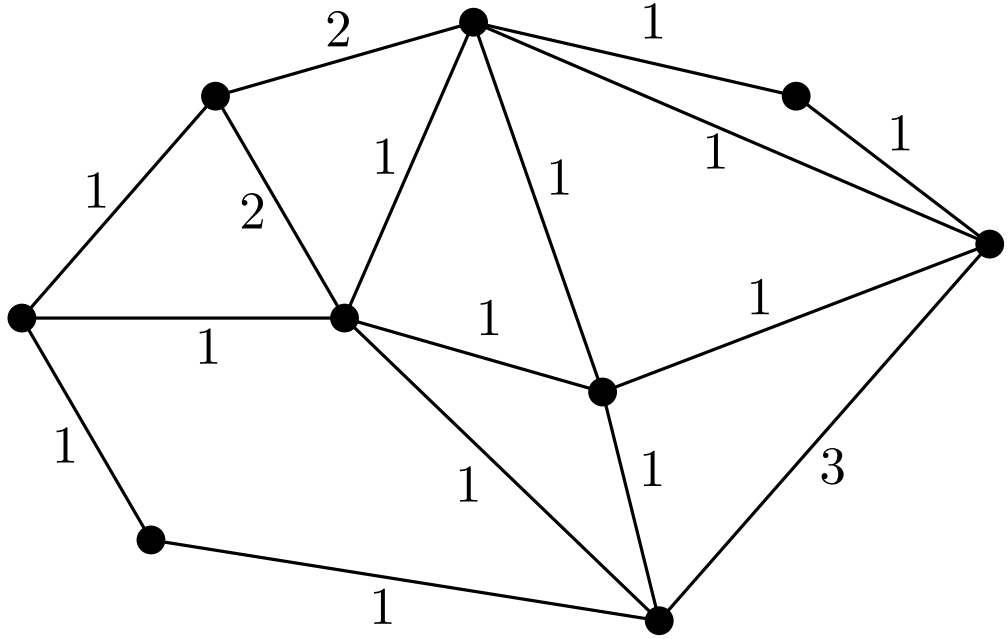
   From $i = 2$ to $i = n - 1$ do:

   If $w_{in}(v_i) > w_{out}(v_i)$, replace the weight 1 of the (counterclockwise) first outgoing edge of v_i by the weight $w_{in}(v_i) - w_{out}(v_i) + 1$.

3. *Second sweep (backwards)*

   From $i = n - 1$ to $i = 2$ do:

   If $w_{out}(v_i) > w_{in}(v_i)$, replace the weight $w(e)$ of the (counterclockwise) first ingoing edge of v_i by the weight $w_{out}(v_i) - w_{in}(v_i) + w(e)$.



POINT LOCATION: Monotone subdivision

Theorem

Weight assignment:

1. *Initialization*

   Assign weight 1 to all edges

2. *First sweep (forwards)*

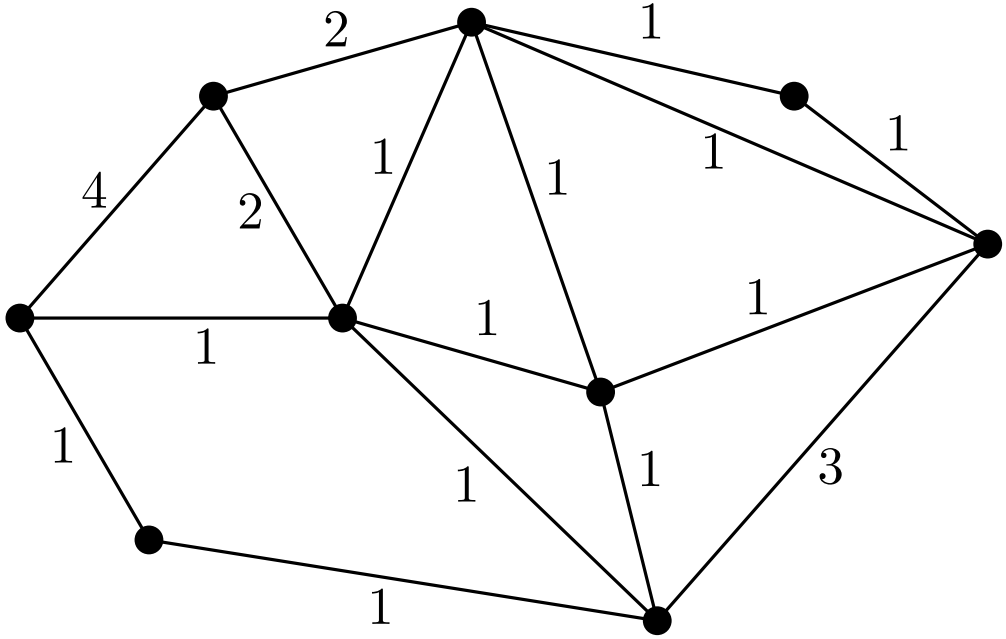
   From $i = 2$ to $i = n - 1$ do:

   If $w_{in}(v_i) > w_{out}(v_i)$, replace the weight 1 of the (counterclockwise) first outgoing edge of v_i by the weight $w_{in}(v_i) - w_{out}(v_i) + 1$.

3. *Second sweep (backwards)*

   From $i = n - 1$ to $i = 2$ do:

   If $w_{out}(v_i) > w_{in}(v_i)$, replace the weight $w(e)$ of the (counterclockwise) first ingoing edge of v_i by the weight $w_{out}(v_i) - w_{in}(v_i) + w(e)$.

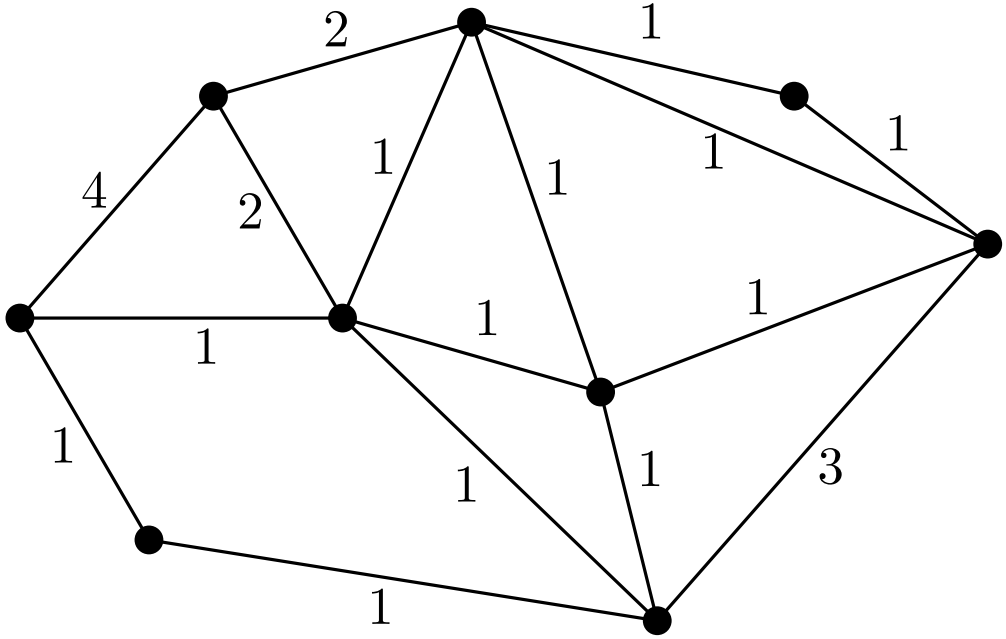


POINT LOCATION: Monotone subdivision

Theorem

Computing the chains:

- All chains start at v_1 and end at v_n.
- Each time an edge e is used to build a chain, its weight $w(e)$ is decreased by one.
- During the construction, always leave vertices through the (counterclockwise) first edge having positive weight.

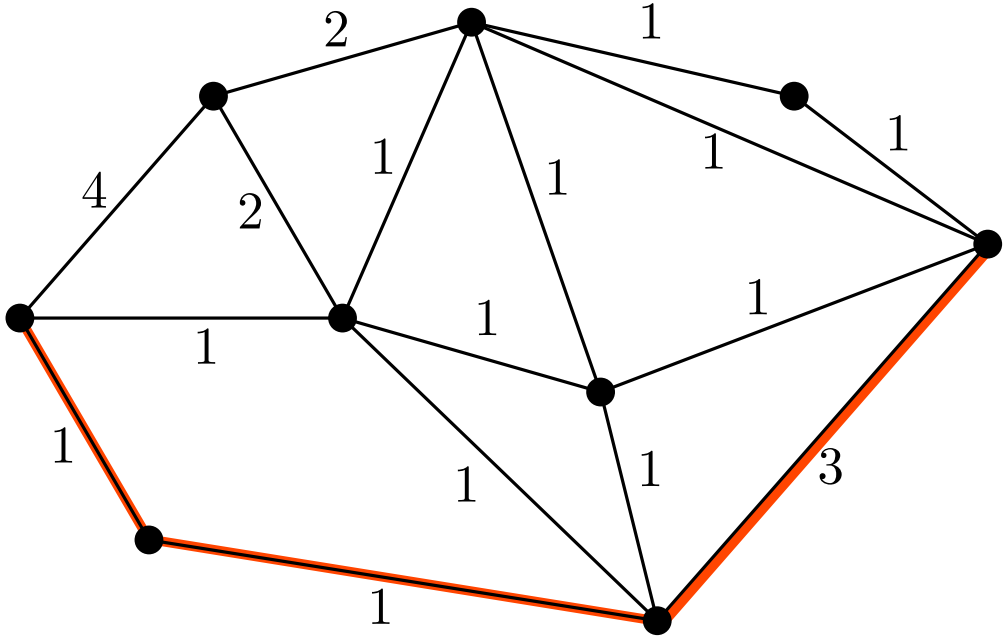


POINT LOCATION: Monotone subdivision

Theorem

Computing the chains:

- All chains start at v_1 and end at v_n.
- Each time an edge e is used to build a chain, its weight $w(e)$ is decreased by one.
- During the construction, always leave vertices through the (counterclockwise) first edge having positive weight.

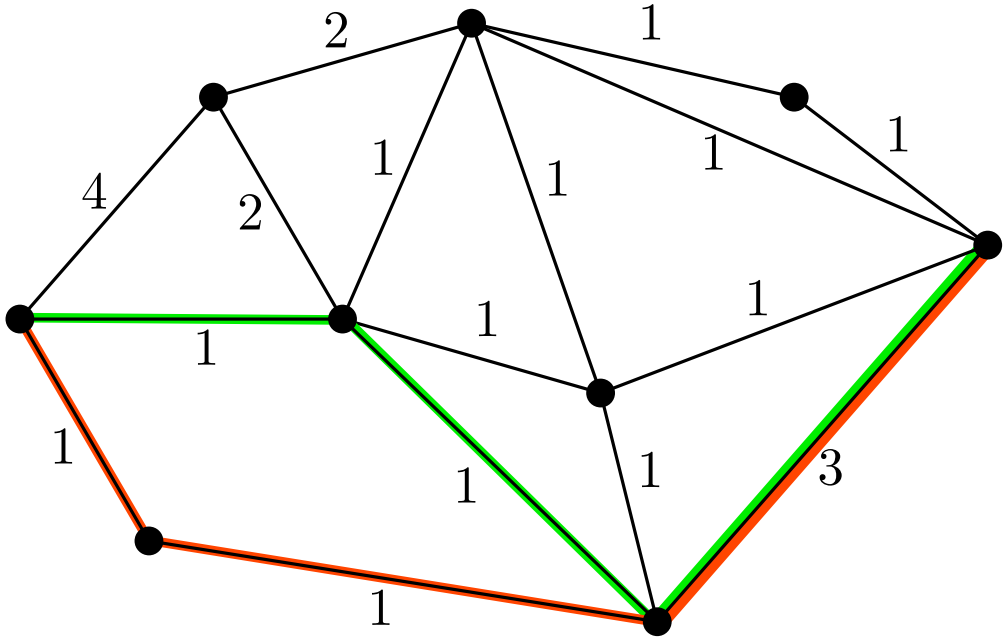


POINT LOCATION: Monotone subdivision

Theorem

Computing the chains:

- All chains start at v_1 and end at v_n.
- Each time an edge e is used to build a chain, its weight $w(e)$ is decreased by one.
- During the construction, always leave vertices through the (counterclockwise) first edge having positive weight.

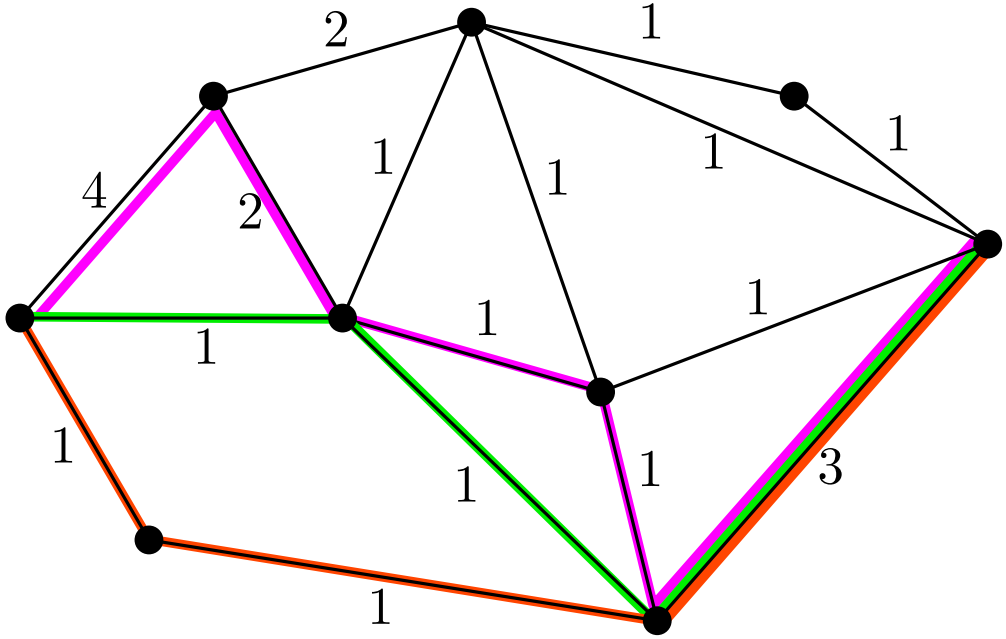


POINT LOCATION: Monotone subdivision

Theorem

Computing the chains:

- All chains start at v_1 and end at v_n.
- Each time an edge e is used to build a chain, its weight $w(e)$ is decreased by one.
- During the construction, always leave vertices through the (counterclockwise) first edge having positive weight.

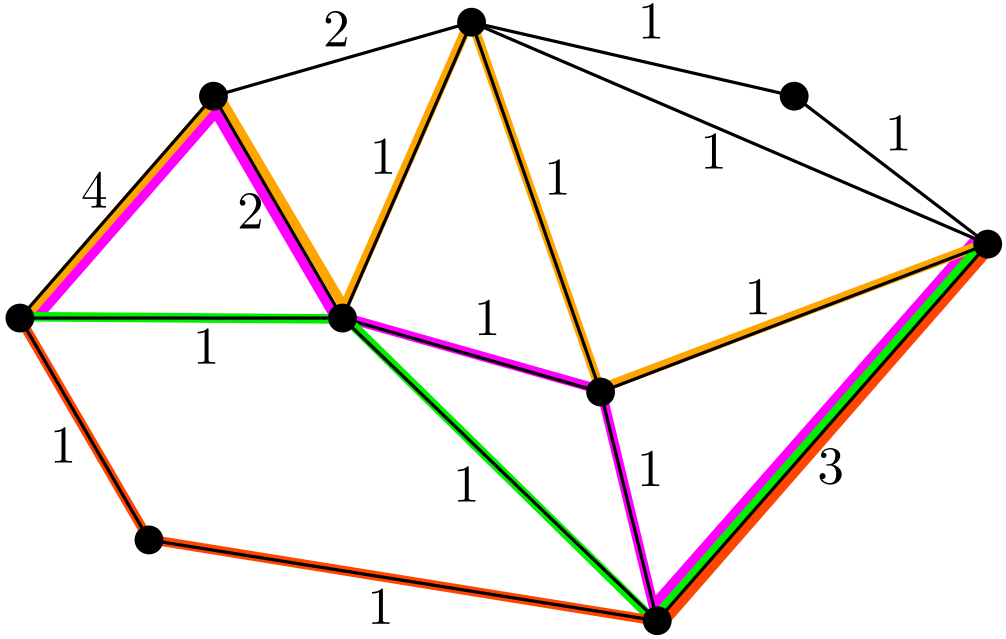


POINT LOCATION: Monotone subdivision

Theorem

Computing the chains:

- All chains start at v_1 and end at v_n.
- Each time an edge e is used to build a chain, its weight $w(e)$ is decreased by one.
- During the construction, always leave vertices through the (counterclockwise) first edge having positive weight.

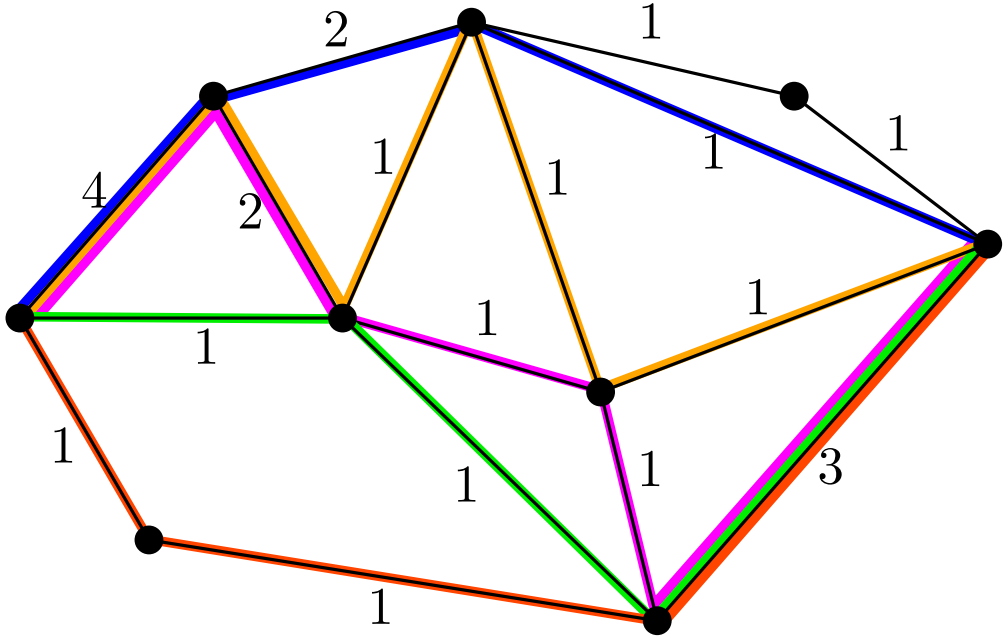


POINT LOCATION: Monotone subdivision

Theorem

Computing the chains:

- All chains start at v_1 and end at v_n.
- Each time an edge e is used to build a chain, its weight $w(e)$ is decreased by one.
- During the construction, always leave vertices through the (counterclockwise) first edge having positive weight.

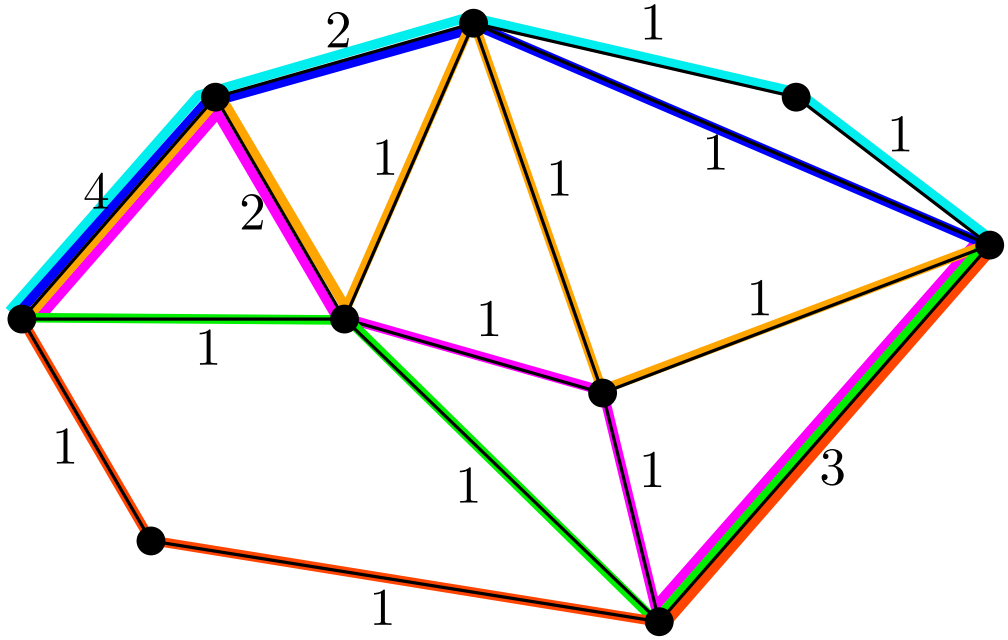


POINT LOCATION: Monotone subdivision

Theorem

Computing the chains:

- All chains start at v_1 and end at v_n.
- Each time an edge e is used to build a chain, its weight $w(e)$ is decreased by one.
- During the construction, always leave vertices through the (counterclockwise) first edge having positive weight.



POINT LOCATION: Monotone subdivision

Complexity

POINT LOCATION: Monotone subdivision

Complexity

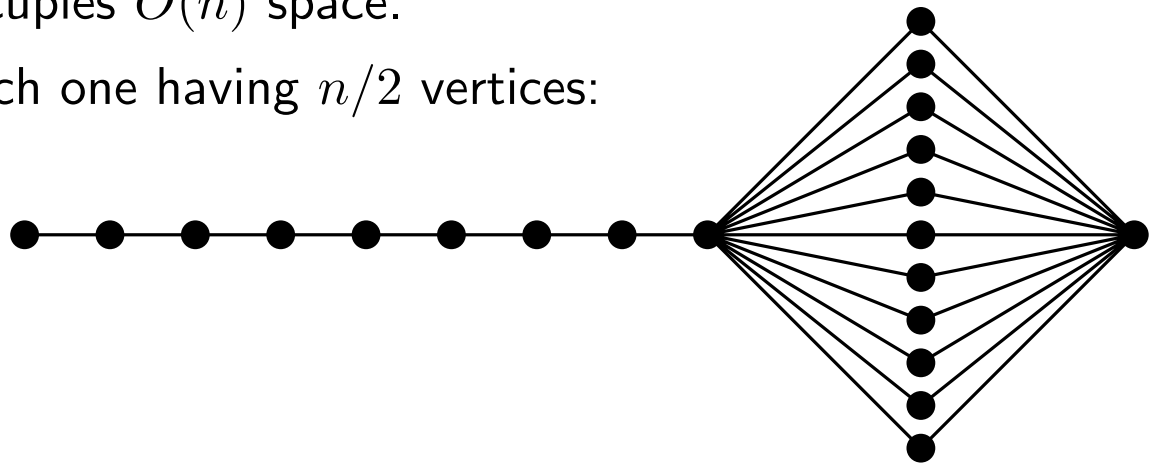
Space: The complete set of chains occupies $O(n)$ space.

POINT LOCATION: Monotone subdivision

Complexity

Space: The complete set of chains occupies $O(n)$ space.

There exist graphs with $n/2$ chains, each one having $n/2$ vertices:

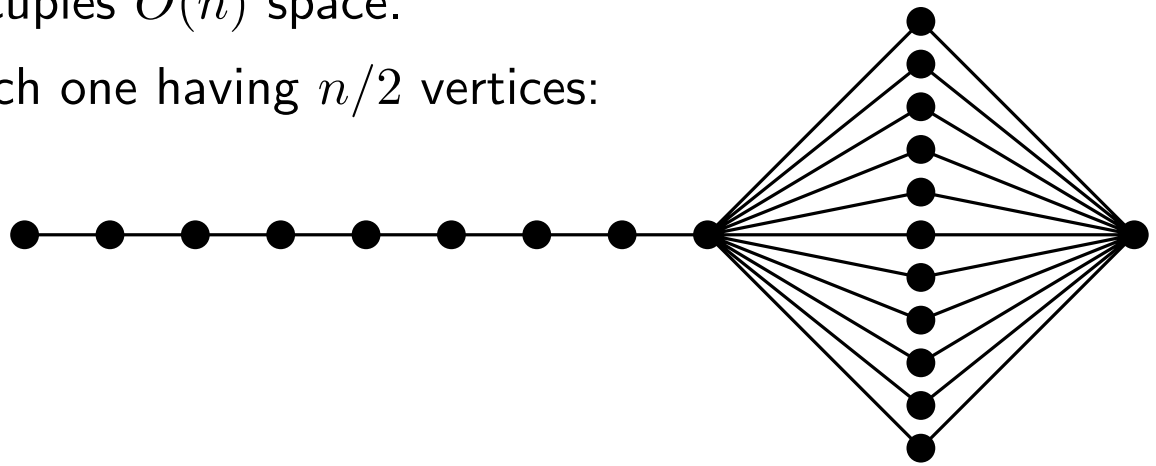


POINT LOCATION: Monotone subdivision

Complexity

Space: The complete set of chains occupies $O(n)$ space.

There exist graphs with $n/2$ chains, each one having $n/2$ vertices:



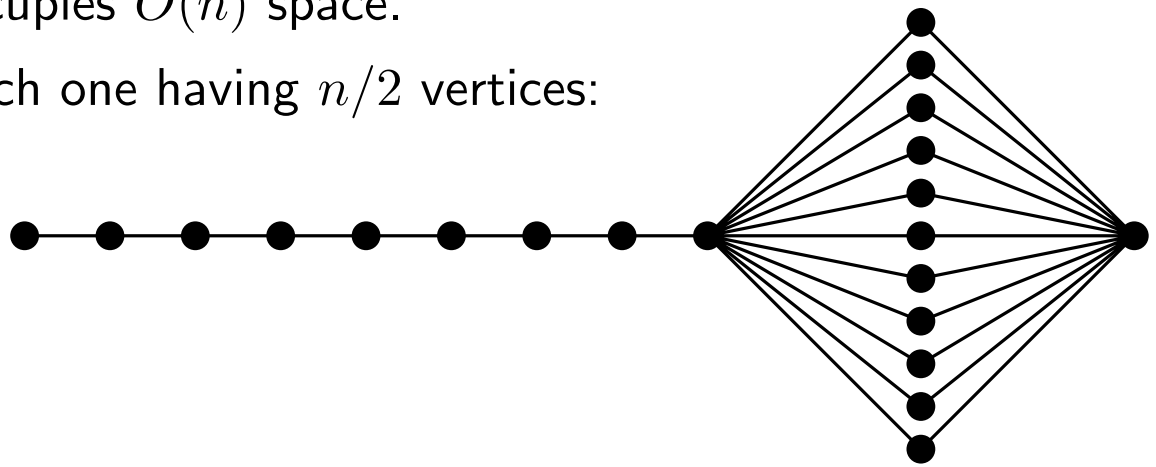
This seems to mean that information of size $O(n^2)$ should be stored, but this is redundant: the graph can be stored in $O(n)$ space, based on the following observation:

POINT LOCATION: Monotone subdivision

Complexity

Space: The complete set of chains occupies $O(n)$ space.

There exist graphs with $n/2$ chains, each one having $n/2$ vertices:



This seems to mean that information of size $O(n^2)$ should be stored, but this is redundant: the graph can be stored in $O(n)$ space, based on the following observation:

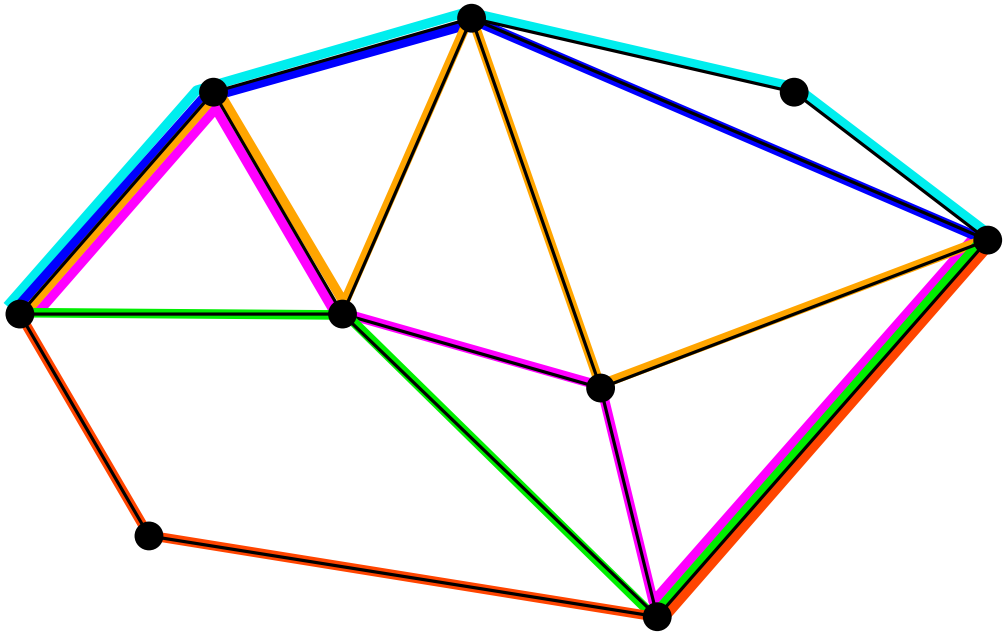
When an edge e belongs to several chains, it belongs to an interval of consecutive chains (chains are totally ordered). Store e in the ascendant of the entire interval of chains in the search structure C. It is in that chain where e will be used in the search algorithm. For each remaining chain of the interval, the edge e is replaced by a bypass pointer.

The number of pointers is not quadratic, but linear, because each pointer points to one single edge of G, and each edge gets at most one single pointer.

POINT LOCATION: Monotone subdivision

Complexity

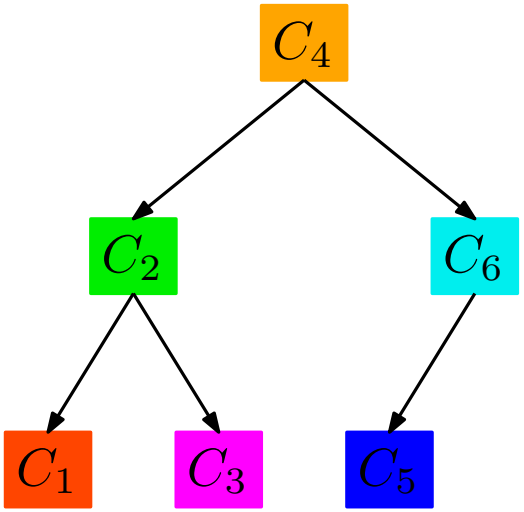
Space: The complete set of chains occupies $O(n)$ space.

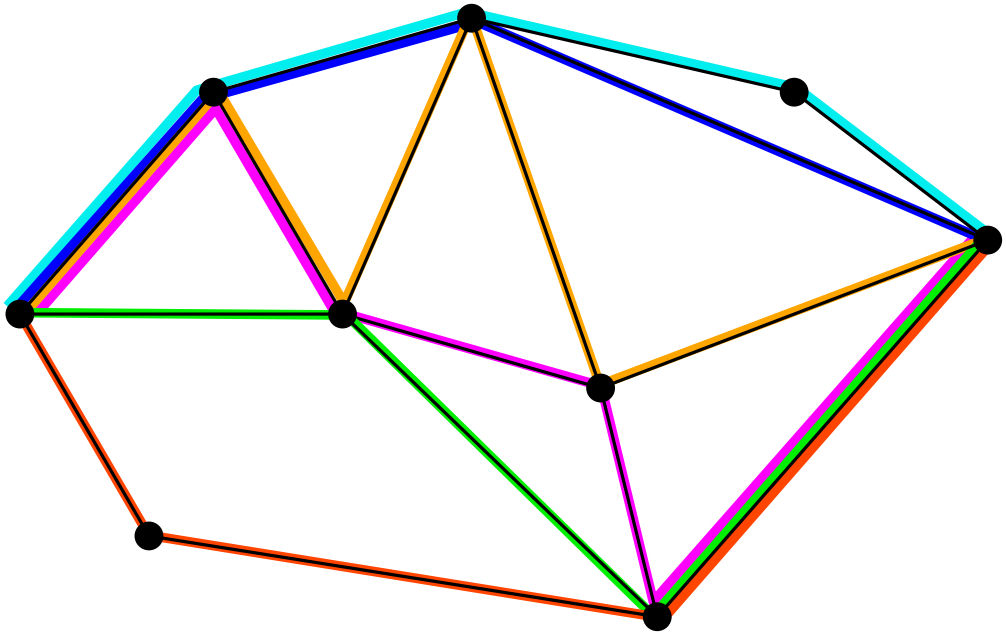


POINT LOCATION: Monotone subdivision

Complexity

Space: The complete set of chains occupies $O(n)$ space.

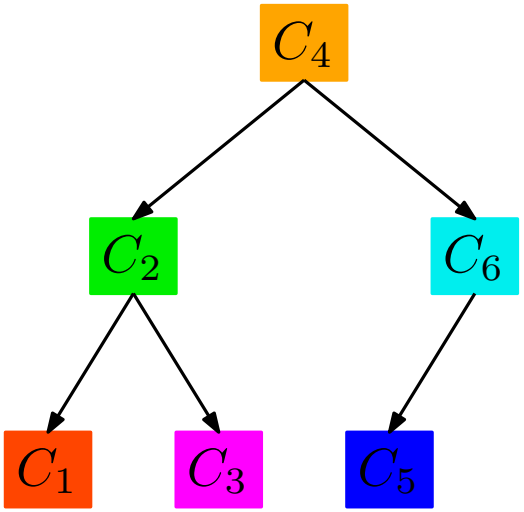


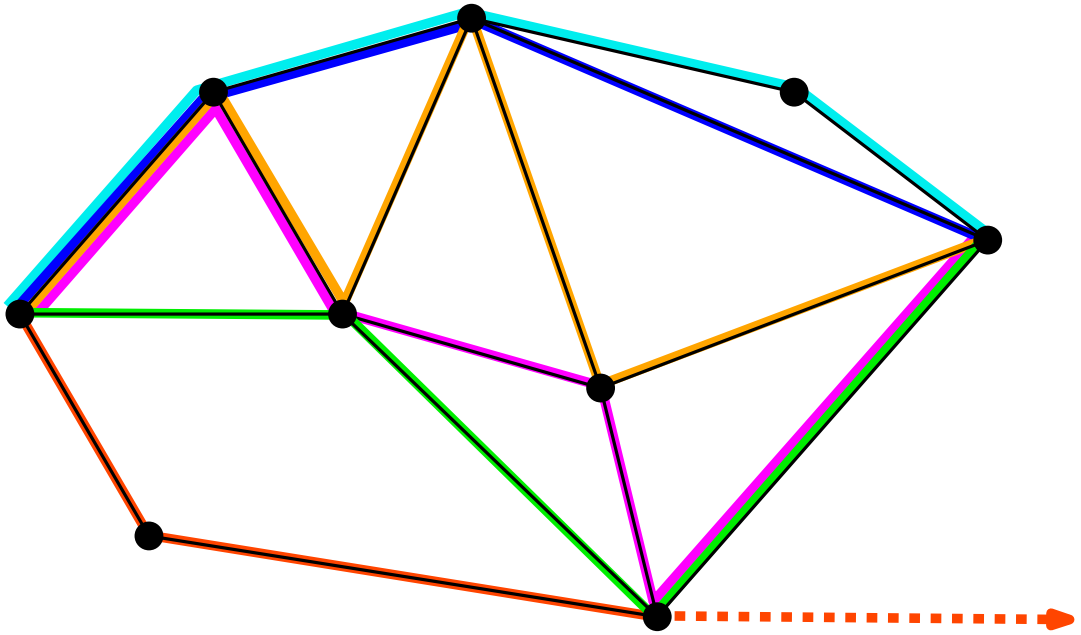


POINT LOCATION: Monotone subdivision

Complexity

Space: The complete set of chains occupies $O(n)$ space.

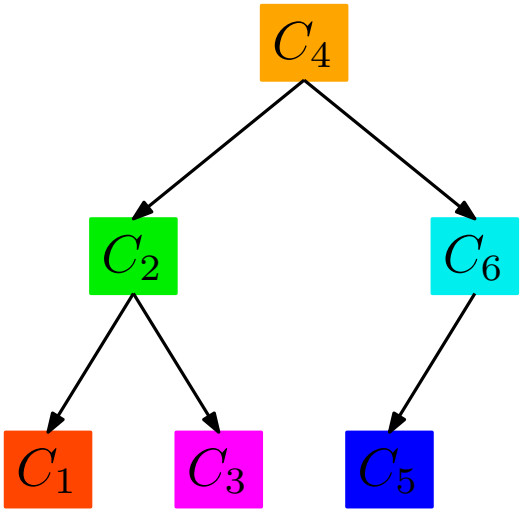


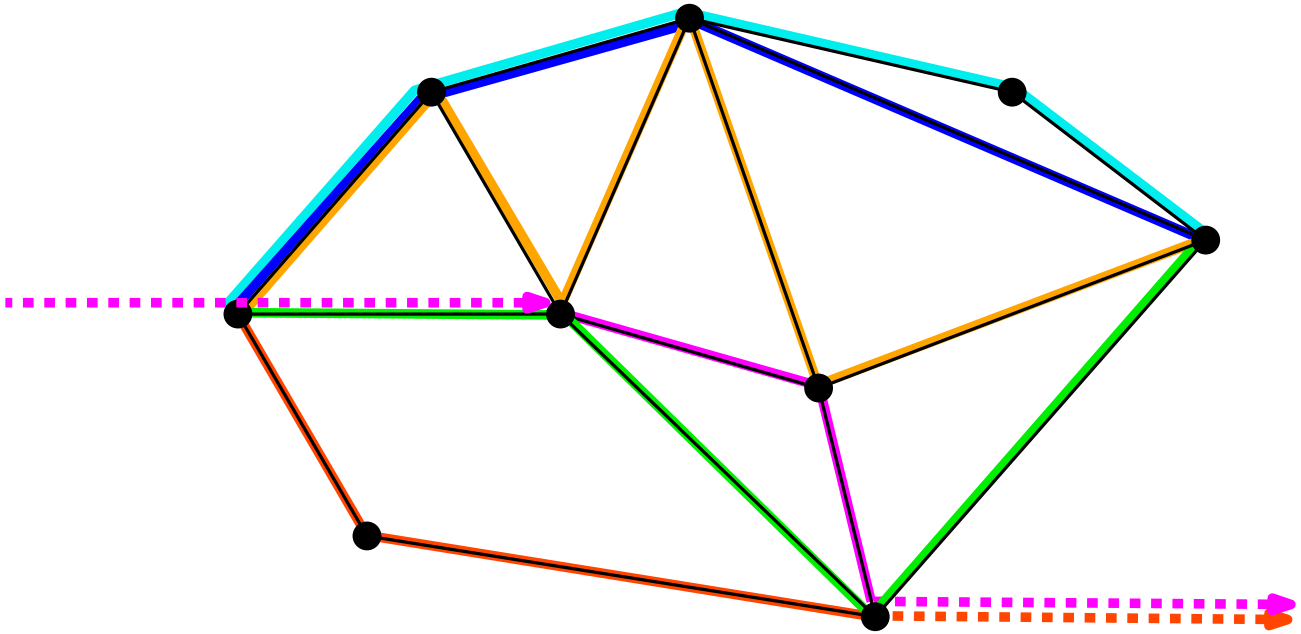


POINT LOCATION: Monotone subdivision

Complexity

Space: The complete set of chains occupies $O(n)$ space.

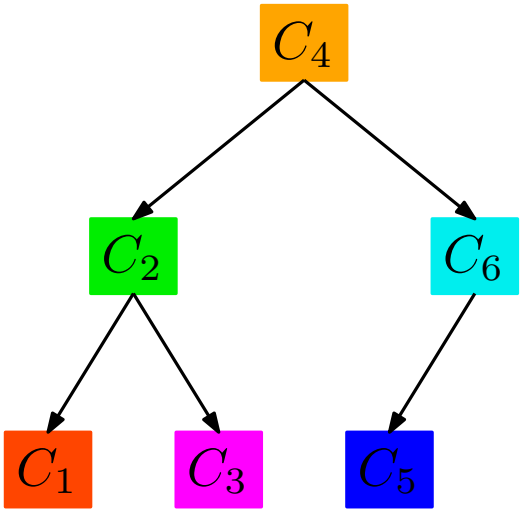


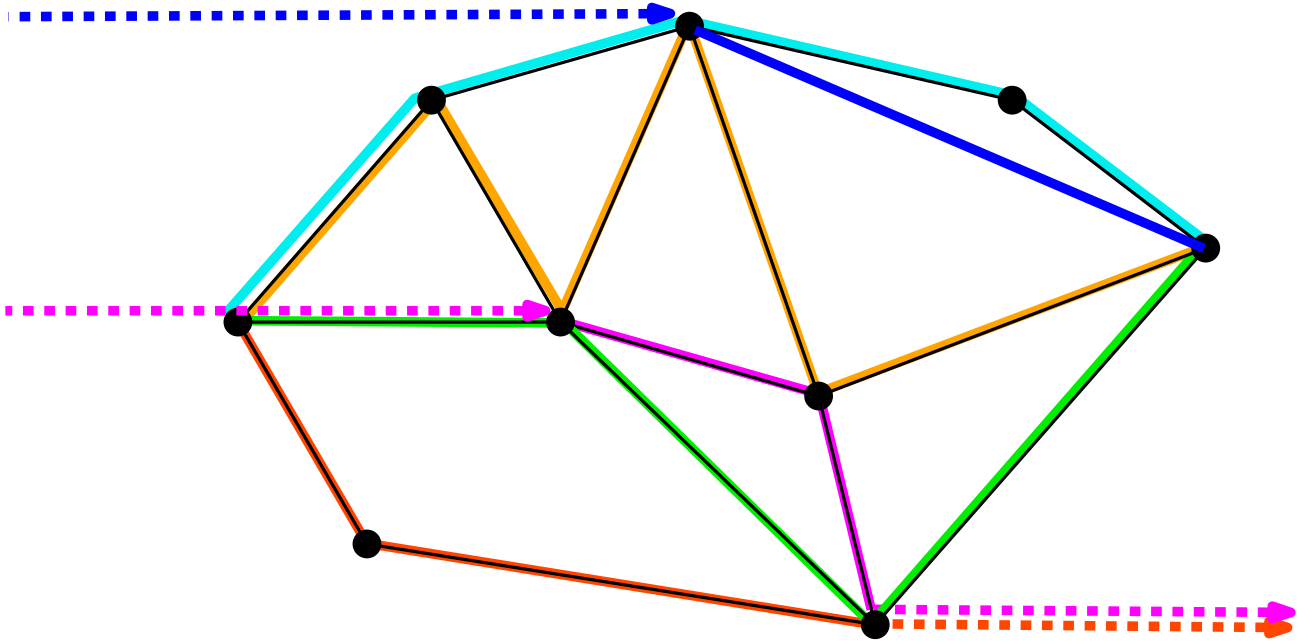


POINT LOCATION: Monotone subdivision

Complexity

Space: The complete set of chains occupies $O(n)$ space.

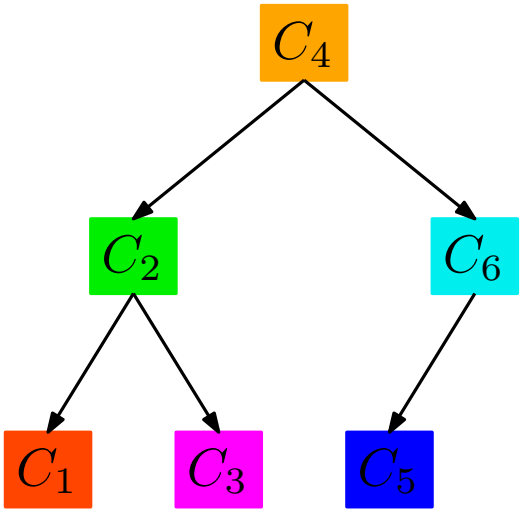


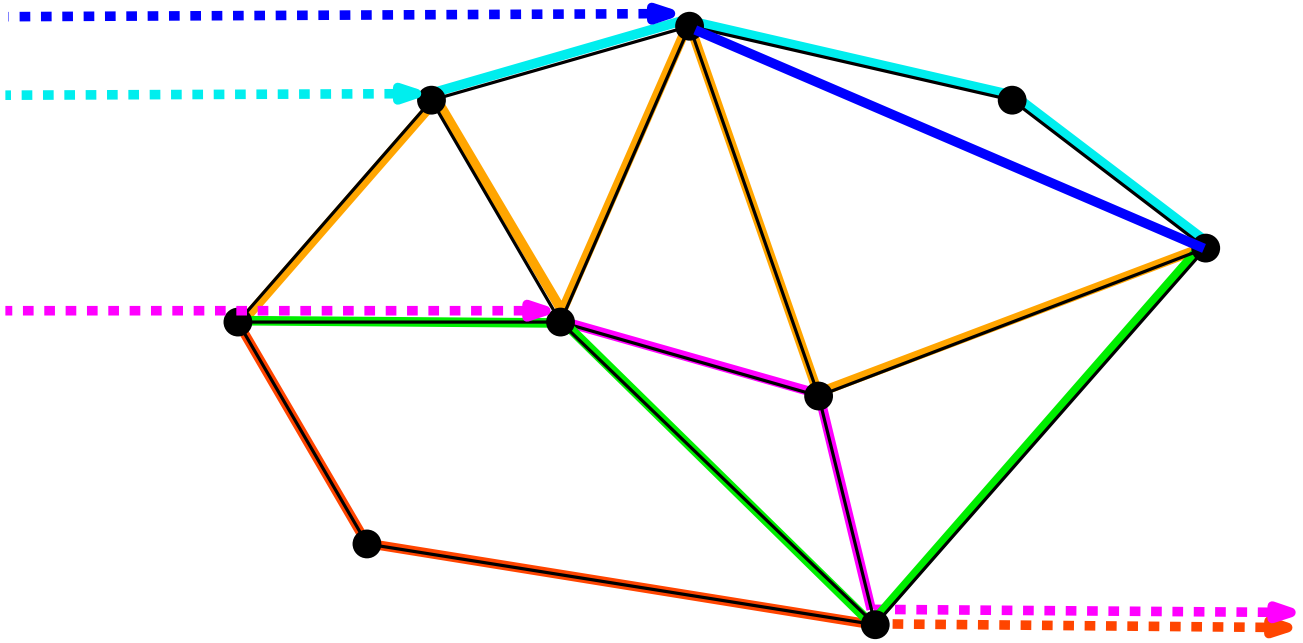


POINT LOCATION: Monotone subdivision

Complexity

Space: The complete set of chains occupies $O(n)$ space.





POINT LOCATION: Monotone subdivision

Complexity

Space: The complete set of chains occupies $O(n)$ space.

Preprocessing: If the graph is regular, the preprocessing can be performed in $O(n)$ time from the DCEL of G, assuming that the vertices of G are sorted by abscissa. Otherwise, sorting the vertices is necessary as first step, requiring $O(n \log n)$ time.

POINT LOCATION: Monotone subdivision

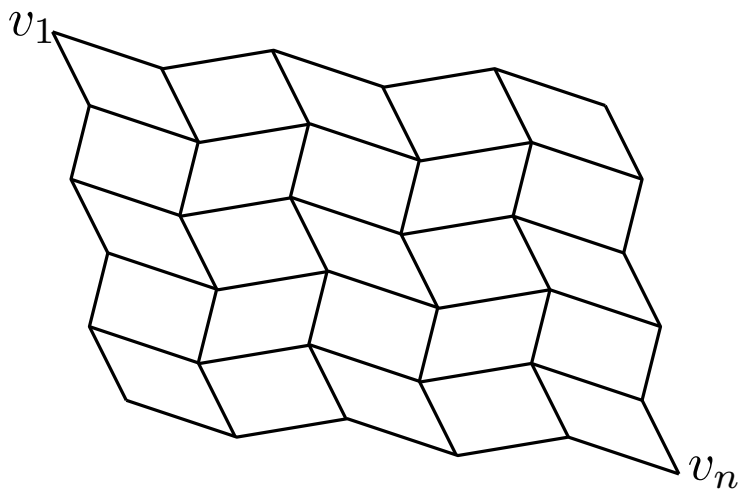
Complexity

Space: The complete set of chains occupies $O(n)$ space.

Preprocessing: If the graph is regular, the preprocessing can be performed in $O(n)$ time from the DCEL of G, assuming that the vertices of G are sorted by abscissa. Otherwise, sorting the vertices is necessary as first step, requiring $O(n \log n)$ time.

Location: Point location can be done in $O(\log^2 n)$ time.

We have already observed that the search running time is $O(\log r \log k)$, where r is the number of chains and k is the maximum number of vertices of the chains. The following example shows a case where $r, k \in \Omega(\sqrt{n})$, so that the running time is as bad as $\Omega(\log r \log k) = \Omega(\log^2 \sqrt{n}) = \Omega(\log^2 n)$:



POINT LOCATION: Monotone subdivision

Extension to non-regular graphs

Every non-regular graph can be regularized in $O(n \log n)$ time.

POINT LOCATION: Monotone subdivision

Extension to non-regular graphs

Every non-regular graph can be regularized in $O(n \log n)$ time.

Let $v \neq v_1$ be a vertex without any ingoing edge (the case of a vertex without outgoing edges is analogous).

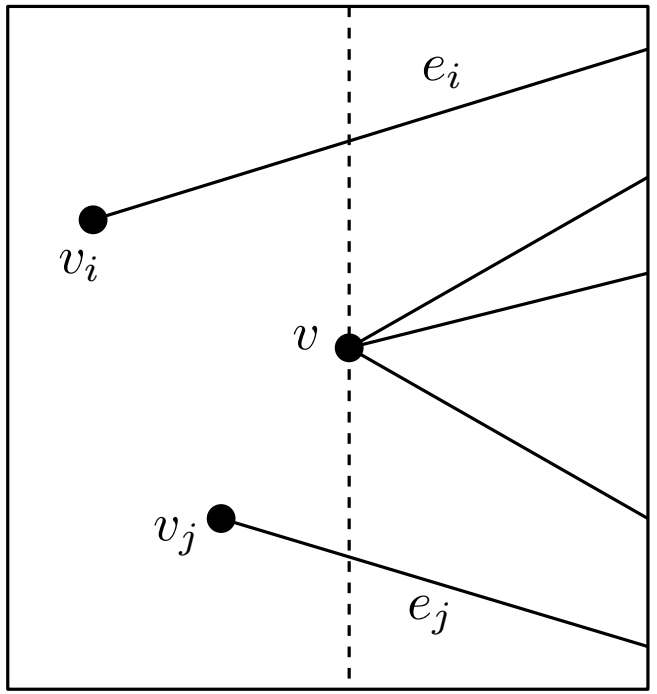
POINT LOCATION: Monotone subdivision

Extension to non-regular graphs

Every non-regular graph can be regularized in $O(n \log n)$ time.

Let $v \neq v_1$ be a vertex without any ingoing edge (the case of a vertex without outgoing edges is analogous).

Since $v \neq v_1$, the vertical line through v intersects at least one, if not two, edges of G, e_i and e_j, adjacent to v. Let v_i and v_j respectively be their left endpoints. Connect v with the rightmost vertex among those lying in the trapezoid limited by e_i, e_j, and the vertical lines through v and through the righmost vertex among v_i and v_j. The resulting line-segment does not intersect any edge of G; inserting it in G regularizes vertex v.



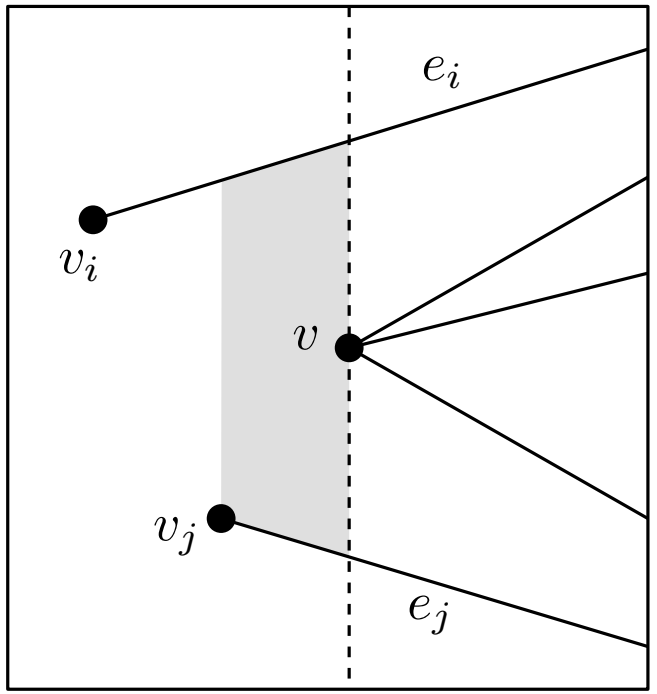
POINT LOCATION: Monotone subdivision

Extension to non-regular graphs

Every non-regular graph can be regularized in $O(n \log n)$ time.

Let $v \neq v_1$ be a vertex without any ingoing edge (the case of a vertex without outgoing edges is analogous).

Since $v \neq v_1$, the vertical line through v intersects at least one, if not two, edges of G, e_i and e_j, adjacent to v. Let v_i and v_j respectively be their left endpoints. Connect v with the rightmost vertex among those lying in the trapezoid limited by e_i, e_j, and the vertical lines through v and through the righmost vertex among v_i and v_j. The resulting line-segment does not intersect any edge of G; inserting it in G regularizes vertex v.



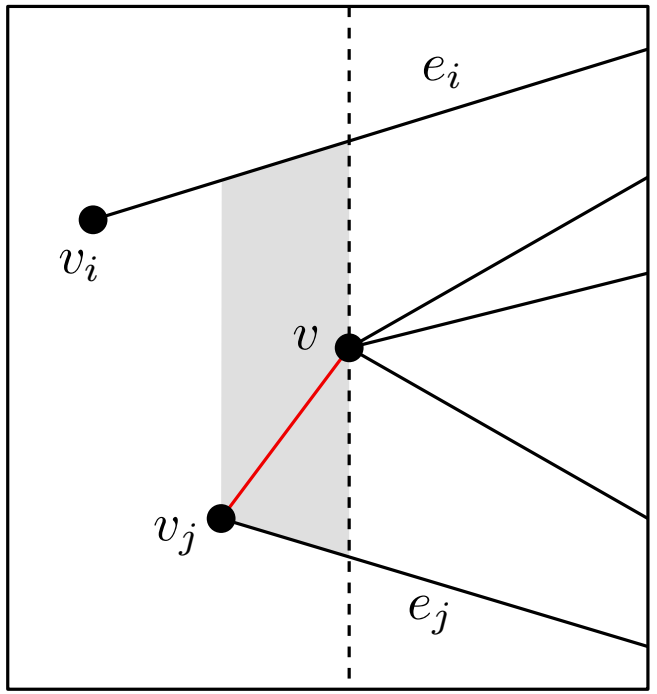
POINT LOCATION: Monotone subdivision

Extension to non-regular graphs

Every non-regular graph can be regularized in $O(n \log n)$ time.

Let $v \neq v_1$ be a vertex without any ingoing edge (the case of a vertex without outgoing edges is analogous).

Since $v \neq v_1$, the vertical line through v intersects at least one, if not two, edges of G, e_i and e_j, adjacent to v. Let v_i and v_j respectively be their left endpoints. Connect v with the rightmost vertex among those lying in the trapezoid limited by e_i, e_j, and the vertical lines through v and through the righmost vertex among v_i and v_j. The resulting line-segment does not intersect any edge of G; inserting it in G regularizes vertex v.



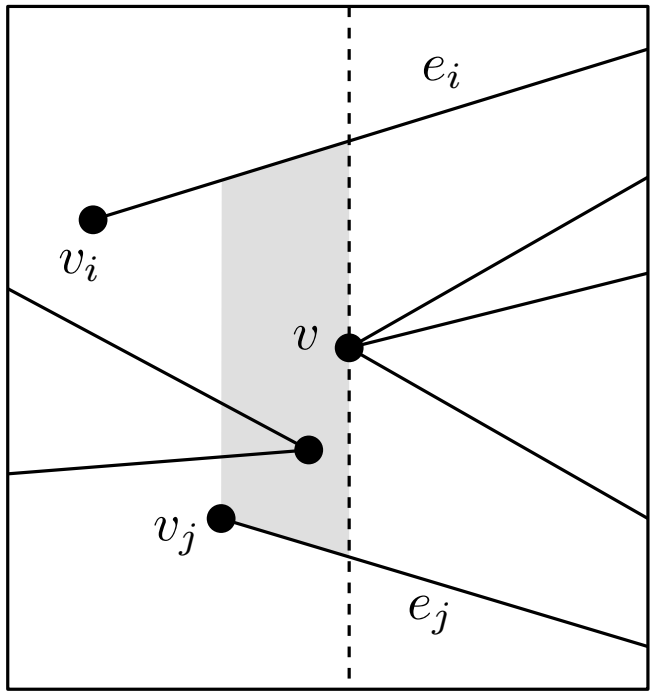
POINT LOCATION: Monotone subdivision

Extension to non-regular graphs

Every non-regular graph can be regularized in $O(n \log n)$ time.

Let $v \neq v_1$ be a vertex without any ingoing edge (the case of a vertex without outgoing edges is analogous).

Since $v \neq v_1$, the vertical line through v intersects at least one, if not two, edges of G, e_i and e_j, adjacent to v. Let v_i and v_j respectively be their left endpoints. Connect v with the rightmost vertex among those lying in the trapezoid limited by e_i, e_j, and the vertical lines through v and through the righmost vertex among v_i and v_j. The resulting line-segment does not intersect any edge of G; inserting it in G regularizes vertex v.



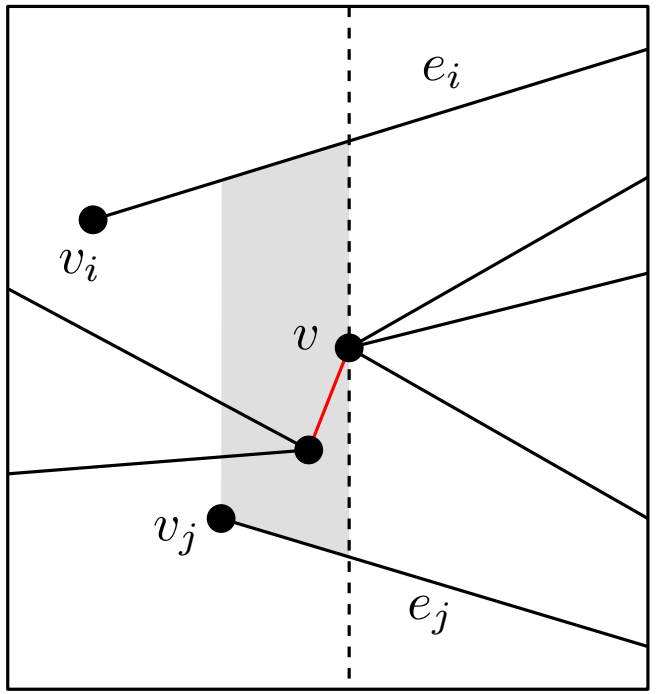
POINT LOCATION: Monotone subdivision

Extension to non-regular graphs

Every non-regular graph can be regularized in $O(n \log n)$ time.

Let $v \neq v_1$ be a vertex without any ingoing edge (the case of a vertex without outgoing edges is analogous).

Since $v \neq v_1$, the vertical line through v intersects at least one, if not two, edges of G, e_i and e_j, adjacent to v. Let v_i and v_j respectively be their left endpoints. Connect v with the rightmost vertex among those lying in the trapezoid limited by e_i, e_j, and the vertical lines through v and through the righmost vertex among v_i and v_j. The resulting line-segment does not intersect any edge of G; inserting it in G regularizes vertex v.



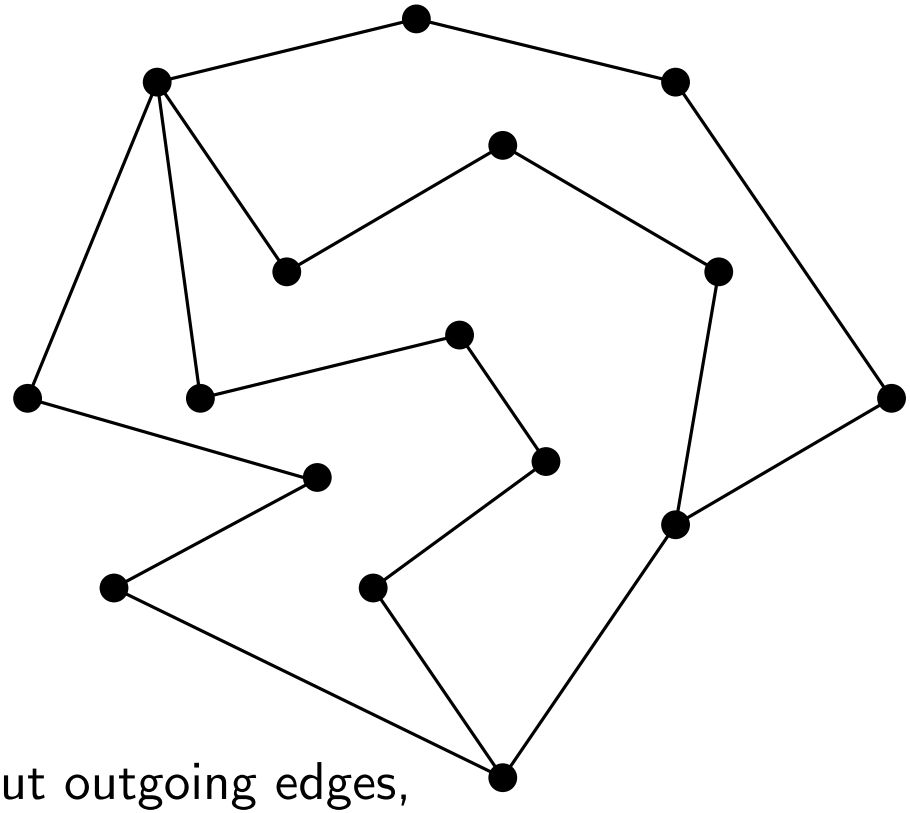
POINT LOCATION: Monotone subdivision

Extension to non-regular graphs

Every non-regular graph can be regularized in $O(n \log n)$ time.

Let $v \neq v_1$ be a vertex without any ingoing edge (the case of a vertex without outgoing edges is analogous).

Since $v \neq v_1$, the vertical line through v intersects at least one, if not two, edges of G, e_i and e_j, adjacent to v. Let v_i and v_j respectively be their left endpoints. Connect v with the rightmost vertex among those lying in the trapezoid limited by e_i, e_j, and the vertical lines through v and through the righmost vertex among v_i and v_j. The resulting line-segment does not intersect any edge of G; inserting it in G regularizes vertex v.



G can be regularized by sweeping it twice:
once from left to right to regularize all vertices without outgoing edges,
once from right to left to regularize all vertices without ingoing edges.

The resulting algorithm runs in $O(n \log n)$ time.

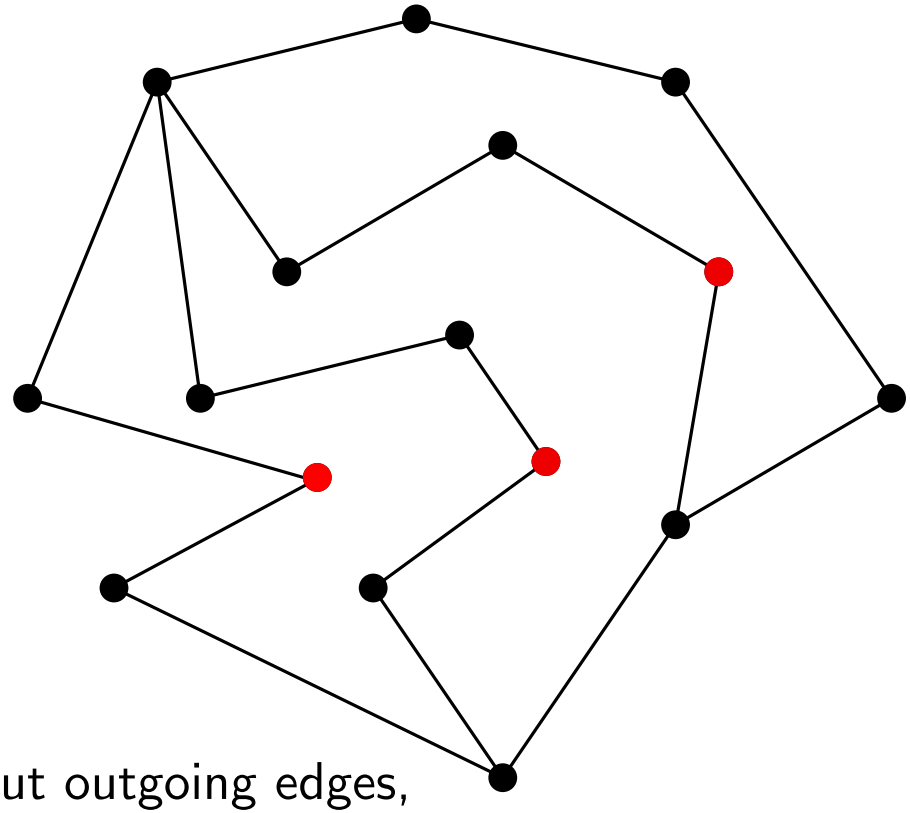
POINT LOCATION: Monotone subdivision

Extension to non-regular graphs

Every non-regular graph can be regularized in $O(n \log n)$ time.

Let $v \neq v_1$ be a vertex without any ingoing edge (the case of a vertex without outgoing edges is analogous).

Since $v \neq v_1$, the vertical line through v intersects at least one, if not two, edges of G, e_i and e_j, adjacent to v. Let v_i and v_j respectively be their left endpoints. Connect v with the rightmost vertex among those lying in the trapezoid limited by e_i, e_j, and the vertical lines through v and through the righmost vertex among v_i and v_j. The resulting line-segment does not intersect any edge of G; inserting it in G regularizes vertex v.



G can be regularized by sweeping it twice:
once from left to right to regularize all vertices without outgoing edges,
once from right to left to regularize all vertices without ingoing edges.

The resulting algorithm runs in $O(n \log n)$ time.

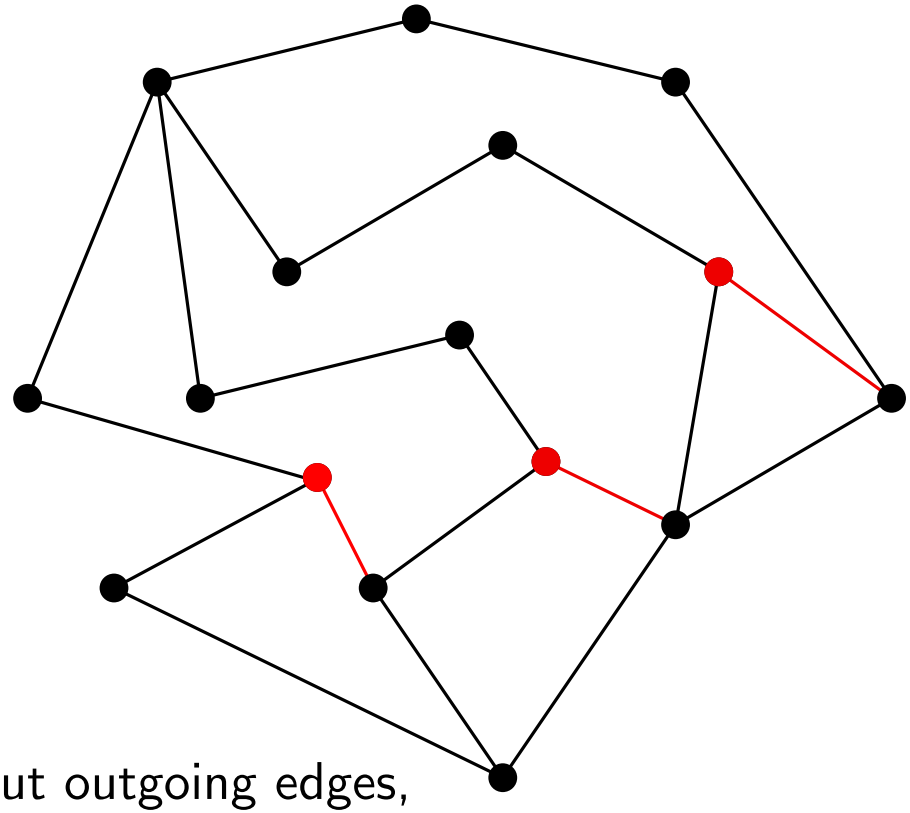
POINT LOCATION: Monotone subdivision

Extension to non-regular graphs

Every non-regular graph can be regularized in $O(n \log n)$ time.

Let $v \neq v_1$ be a vertex without any ingoing edge (the case of a vertex without outgoing edges is analogous).

Since $v \neq v_1$, the vertical line through v intersects at least one, if not two, edges of G, e_i and e_j, adjacent to v. Let v_i and v_j respectively be their left endpoints. Connect v with the rightmost vertex among those lying in the trapezoid limited by e_i, e_j, and the vertical lines through v and through the righmost vertex among v_i and v_j. The resulting line-segment does not intersect any edge of G; inserting it in G regularizes vertex v.



G can be regularized by sweeping it twice:
once from left to right to regularize all vertices without outgoing edges,
once from right to left to regularize all vertices without ingoing edges.

The resulting algorithm runs in $O(n \log n)$ time.

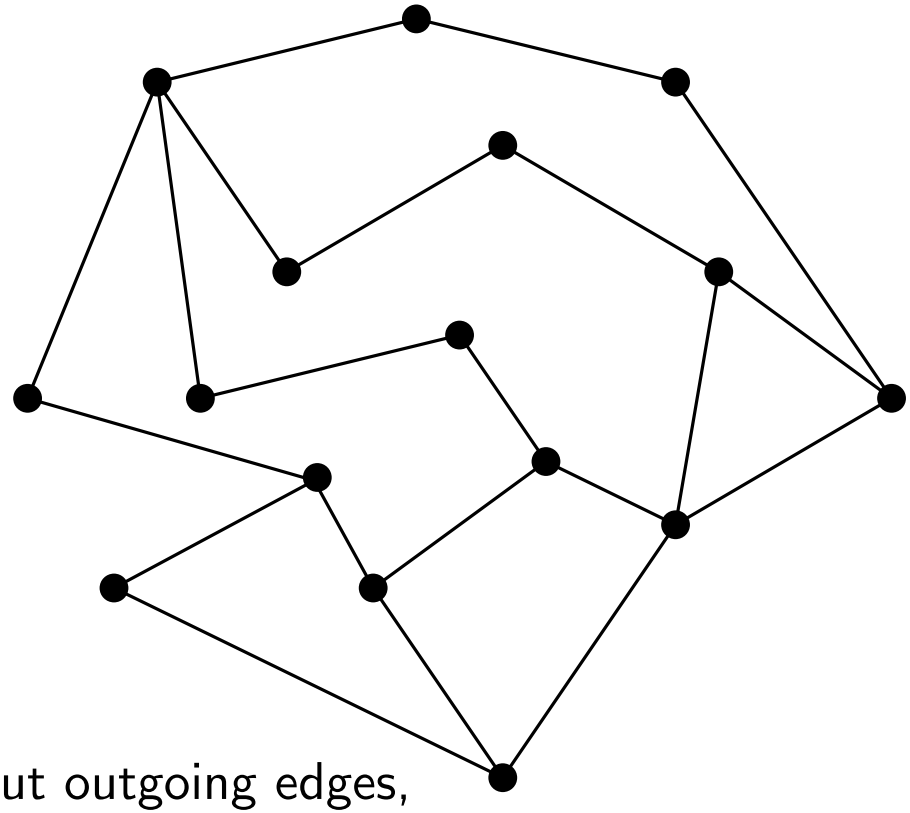
POINT LOCATION: Monotone subdivision

Extension to non-regular graphs

Every non-regular graph can be regularized in $O(n \log n)$ time.

Let $v \neq v_1$ be a vertex without any ingoing edge (the case of a vertex without outgoing edges is analogous).

Since $v \neq v_1$, the vertical line through v intersects at least one, if not two, edges of G, e_i and e_j, adjacent to v. Let v_i and v_j respectively be their left endpoints. Connect v with the rightmost vertex among those lying in the trapezoid limited by e_i, e_j, and the vertical lines through v and through the righmost vertex among v_i and v_j. The resulting line-segment does not intersect any edge of G; inserting it in G regularizes vertex v.



G can be regularized by sweeping it twice:
once from left to right to regularize all vertices without outgoing edges,
once from right to left to regularize all vertices without ingoing edges.

The resulting algorithm runs in $O(n \log n)$ time.

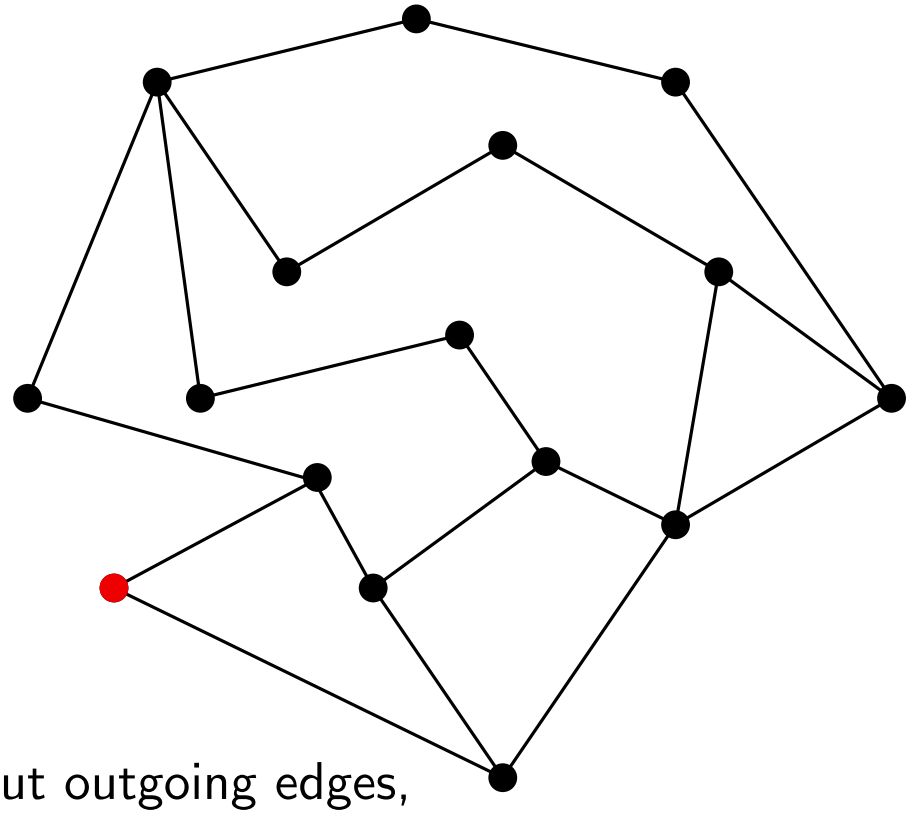
POINT LOCATION: Monotone subdivision

Extension to non-regular graphs

Every non-regular graph can be regularized in $O(n \log n)$ time.

Let $v \neq v_1$ be a vertex without any ingoing edge (the case of a vertex without outgoing edges is analogous).

Since $v \neq v_1$, the vertical line through v intersects at least one, if not two, edges of G, e_i and e_j, adjacent to v. Let v_i and v_j respectively be their left endpoints. Connect v with the rightmost vertex among those lying in the trapezoid limited by e_i, e_j, and the vertical lines through v and through the righmost vertex among v_i and v_j. The resulting line-segment does not intersect any edge of G; inserting it in G regularizes vertex v.



G can be regularized by sweeping it twice:
once from left to right to regularize all vertices without outgoing edges,
once from right to left to regularize all vertices without ingoing edges.

The resulting algorithm runs in $O(n \log n)$ time.

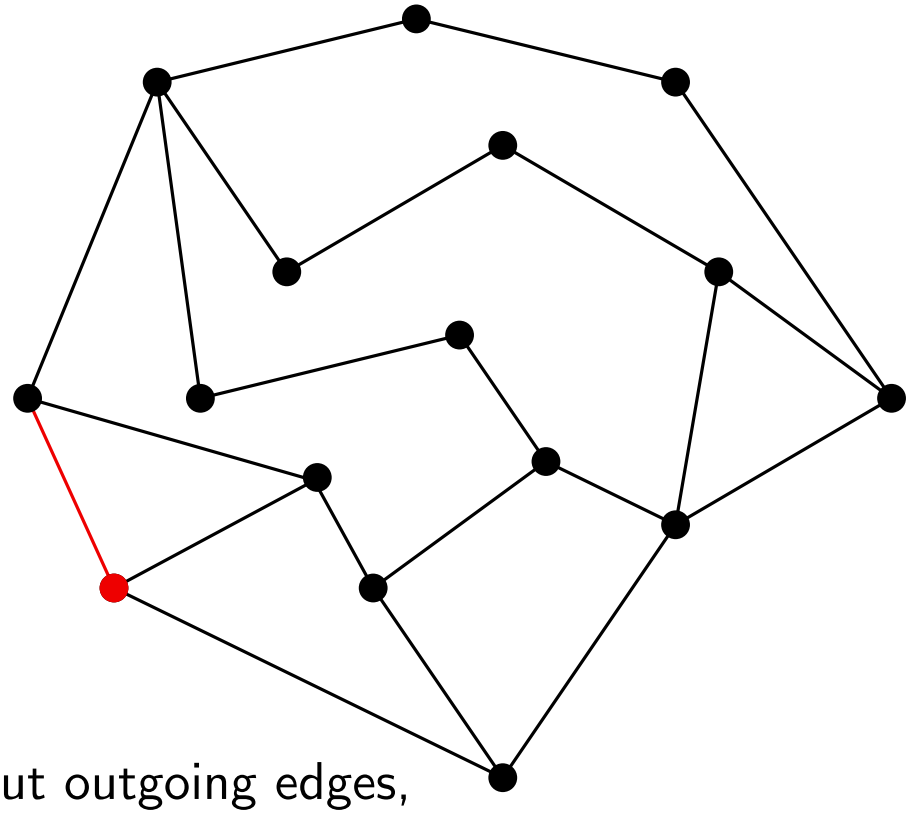
POINT LOCATION: Monotone subdivision

Extension to non-regular graphs

Every non-regular graph can be regularized in $O(n \log n)$ time.

Let $v \neq v_1$ be a vertex without any ingoing edge (the case of a vertex without outgoing edges is analogous).

Since $v \neq v_1$, the vertical line through v intersects at least one, if not two, edges of G, e_i and e_j, adjacent to v. Let v_i and v_j respectively be their left endpoints. Connect v with the rightmost vertex among those lying in the trapezoid limited by e_i, e_j, and the vertical lines through v and through the righmost vertex among v_i and v_j. The resulting line-segment does not intersect any edge of G; inserting it in G regularizes vertex v.



G can be regularized by sweeping it twice:
once from left to right to regularize all vertices without outgoing edges,
once from right to left to regularize all vertices without ingoing edges.

The resulting algorithm runs in $O(n \log n)$ time.

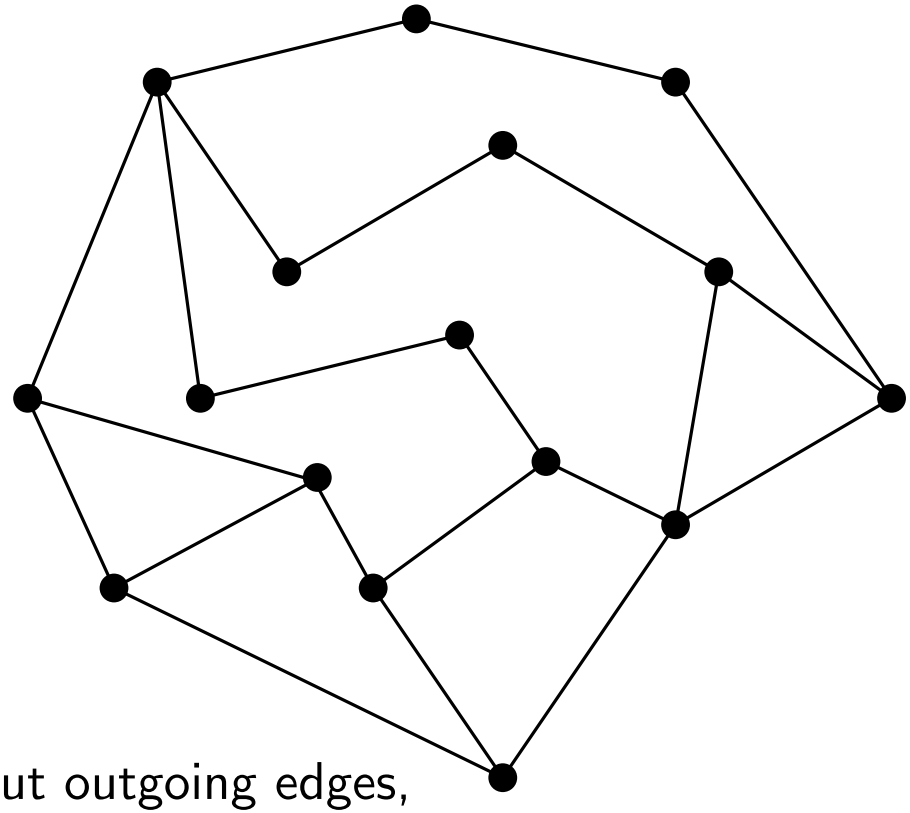
POINT LOCATION: Monotone subdivision

Extension to non-regular graphs

Every non-regular graph can be regularized in $O(n \log n)$ time.

Let $v \neq v_1$ be a vertex without any ingoing edge (the case of a vertex without outgoing edges is analogous).

Since $v \neq v_1$, the vertical line through v intersects at least one, if not two, edges of G, e_i and e_j, adjacent to v. Let v_i and v_j respectively be their left endpoints. Connect v with the rightmost vertex among those lying in the trapezoid limited by e_i, e_j, and the vertical lines through v and through the righmost vertex among v_i and v_j. The resulting line-segment does not intersect any edge of G; inserting it in G regularizes vertex v.



G can be regularized by sweeping it twice:
once from left to right to regularize all vertices without outgoing edges,
once from right to left to regularize all vertices without ingoing edges.

The resulting algorithm runs in $O(n \log n)$ time.

Method 3: **Trapezoidal refinement**

POINT LOCATION: Trapezoidal refinement

This method is an improvement of the slab decomposition we saw before.

It is a refinement method (based on a hierarchy of trapezoidal decompositions).

It can be applied to any planar decomposition

POINT LOCATION: Trapezoidal refinement

This method is an improvement of the slab decomposition we saw before.

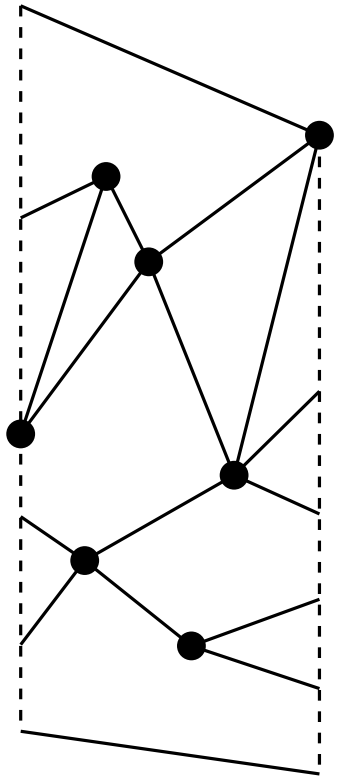
It is a refinement method (based on a hierarchy of trapezoidal decompositions).

It can be applied to any planar decomposition

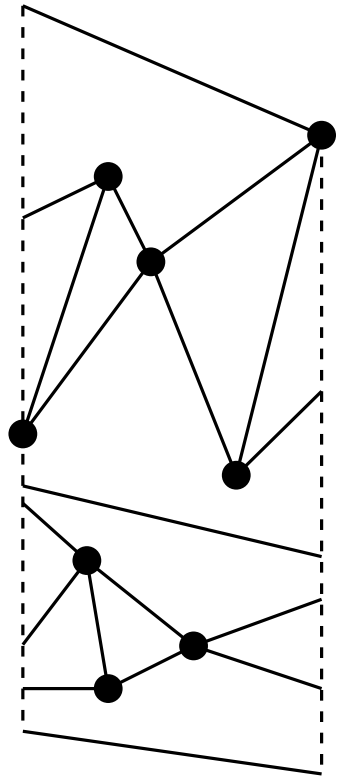
Trapezoid characteristics

The trapezoids used in this method have the following characteristics:

- Their vertical edges are line-segments or half-lines through vertices of the initial decomposition.
- The other edges of the trapezoid are edges or portions of edges of the initial graph.
- No edge of the initial graph simultaneoulsy intersects both vertical edges of a trapezoid.



Legal trapezoid

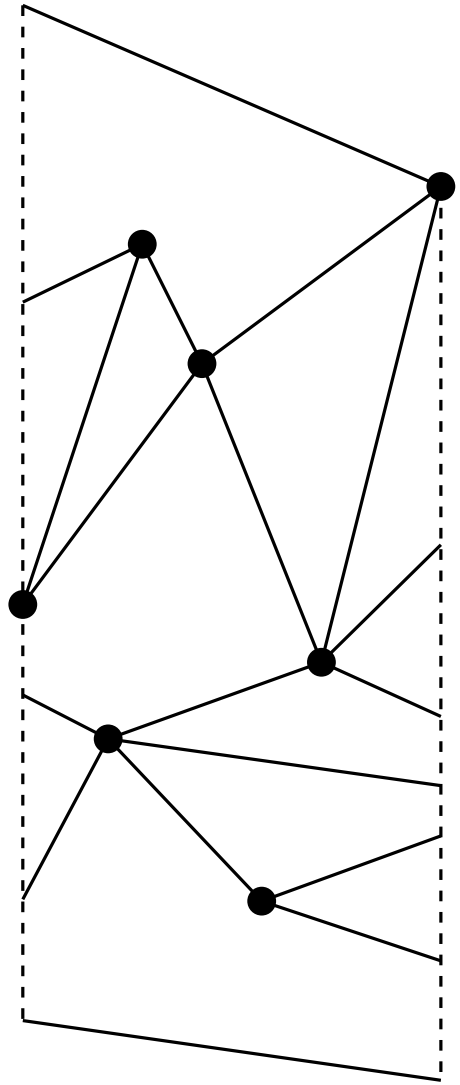


Non-legal trapezoid

POINT LOCATION: Trapezoidal refinement

Refinement

Given a trapezoid T:

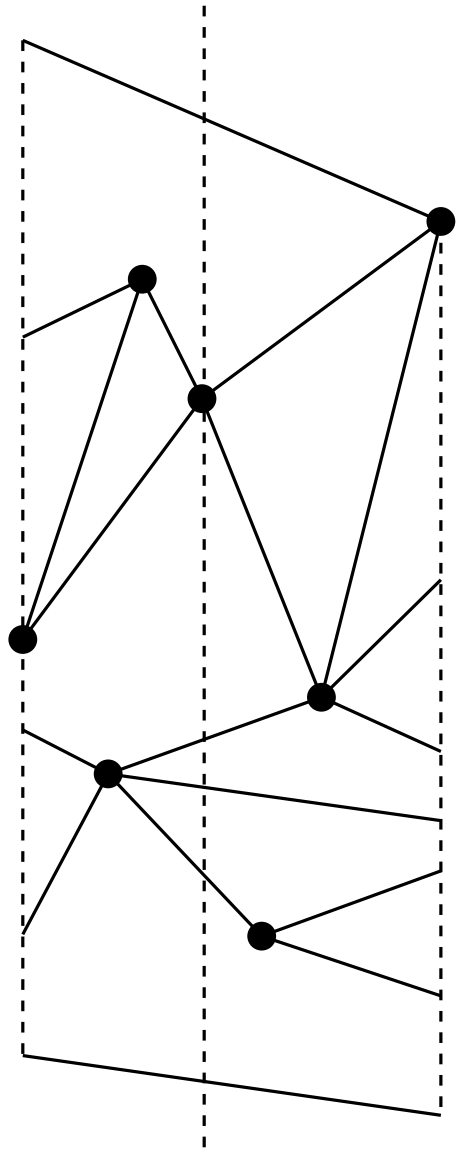


POINT LOCATION: Trapezoidal refinement

Refinement

Given a trapezoid T:

1. Consider the vertical line through the vertex of median abscissa among all vertices in T.

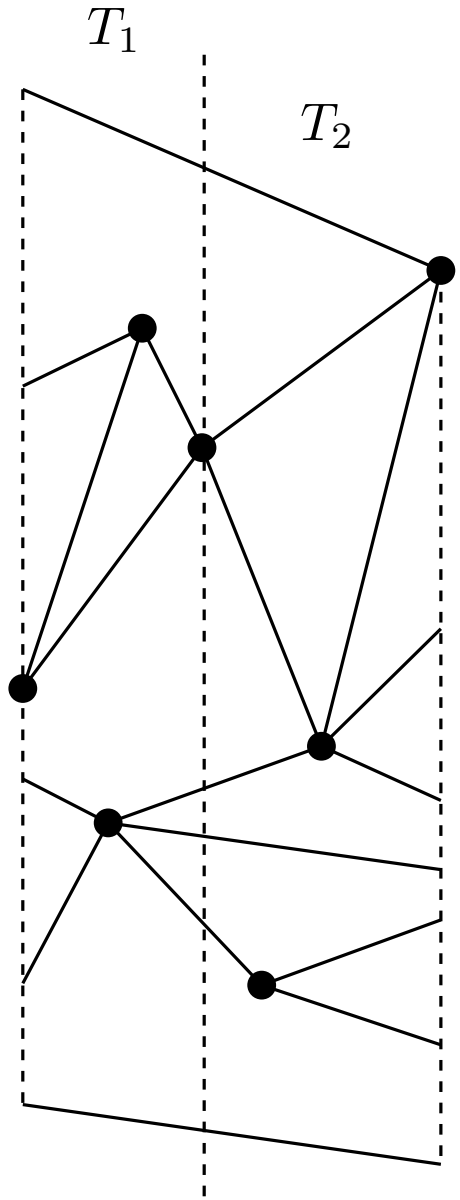


POINT LOCATION: Trapezoidal refinement

Refinement

Given a trapezoid T:

1. Consider the vertical line through the vertex of median abscissa among all vertices in T.
2. Decompose T into two strips, T_1 and T_2, left and right of the line.

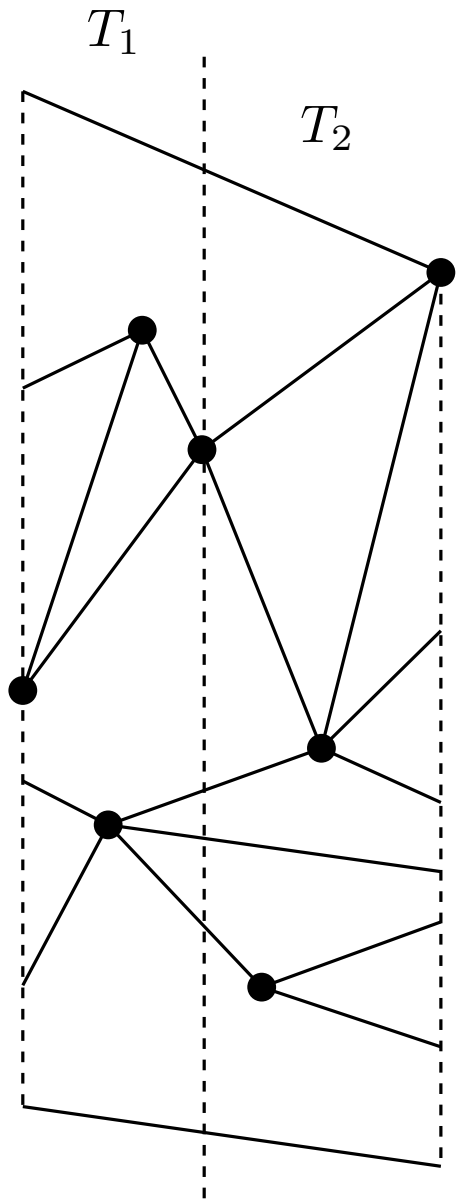


POINT LOCATION: Trapezoidal refinement

Refinement

Given a trapezoid T:

1. Consider the vertical line through the vertex of median abscissa among all vertices in T.
2. Decompose T into two strips, T_1 and T_2, left and right of the line.
3. If T_1 and T_2 are legal trapezoids, no further action is needed.

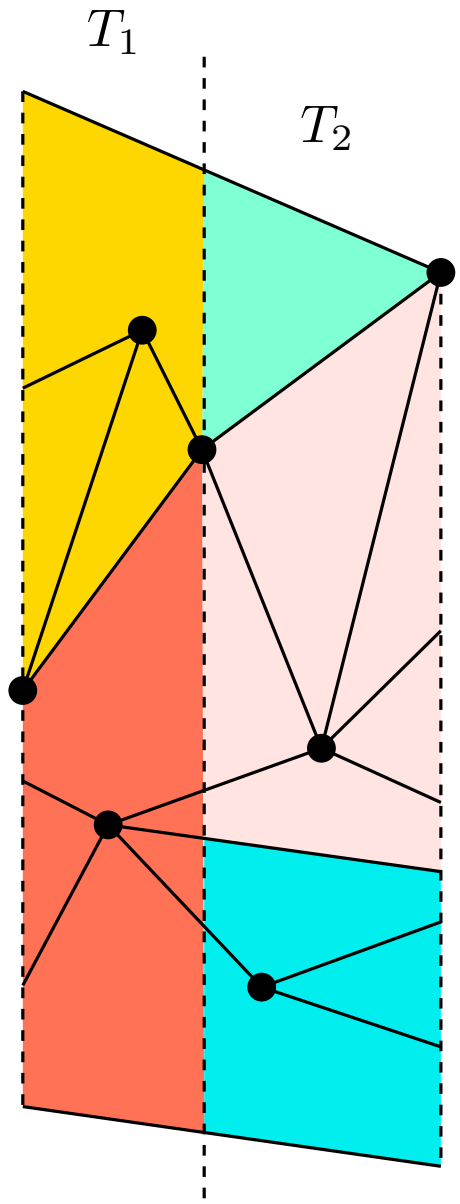


POINT LOCATION: Trapezoidal refinement

Refinement

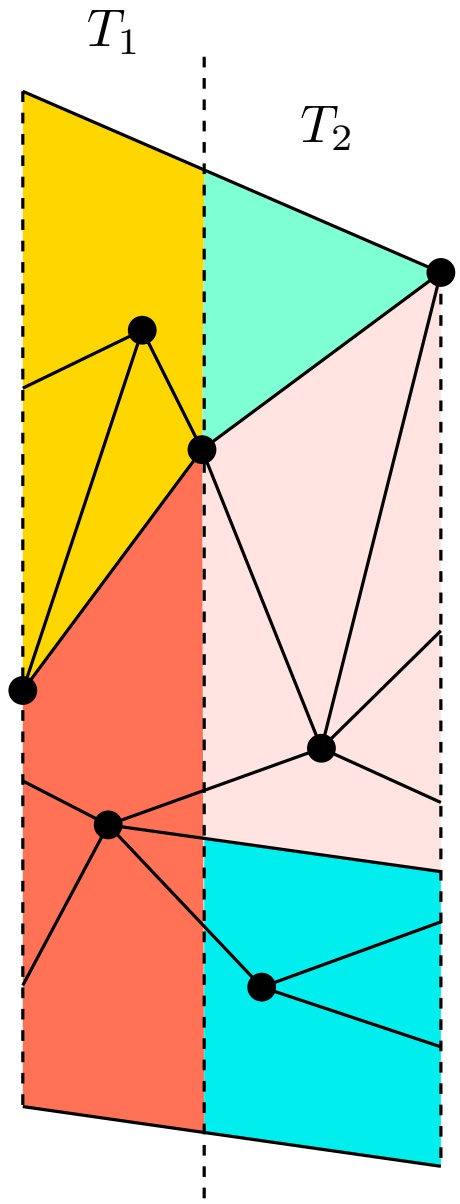
Given a trapezoid T:

1. Consider the vertical line through the vertex of median abscissa among all vertices in T.
2. Decompose T into two strips, T_1 and T_2, left and right of the line.
3. If T_1 and T_2 are legal trapezoids, no further action is needed.
4. For each edge completely traversing T_1 (respectively T_2)—recall that no edge can traverse both—, decompose T_1 (T_2) into two pieces, one above and one below the edge.



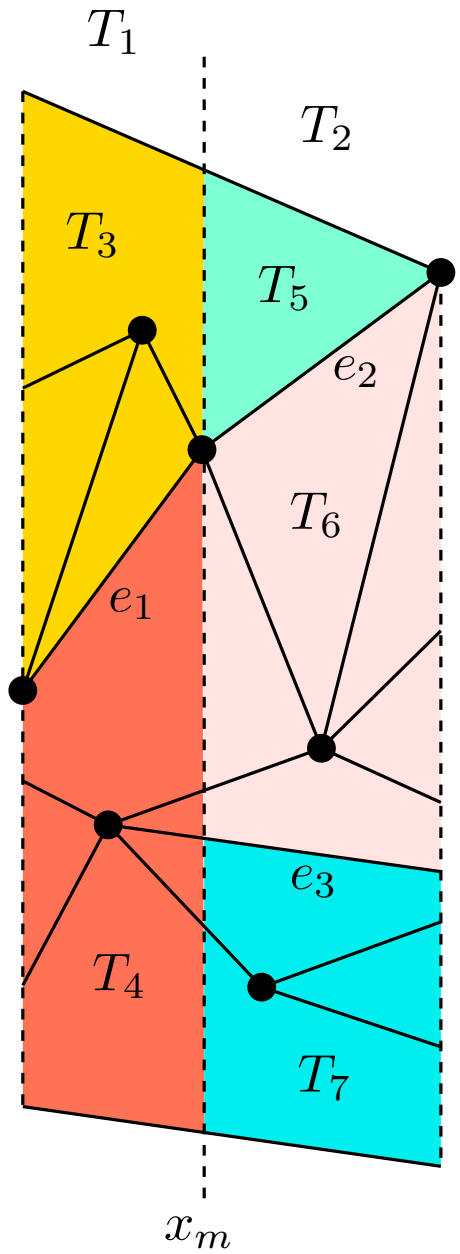
POINT LOCATION: Trapezoidal refinement

Search structure



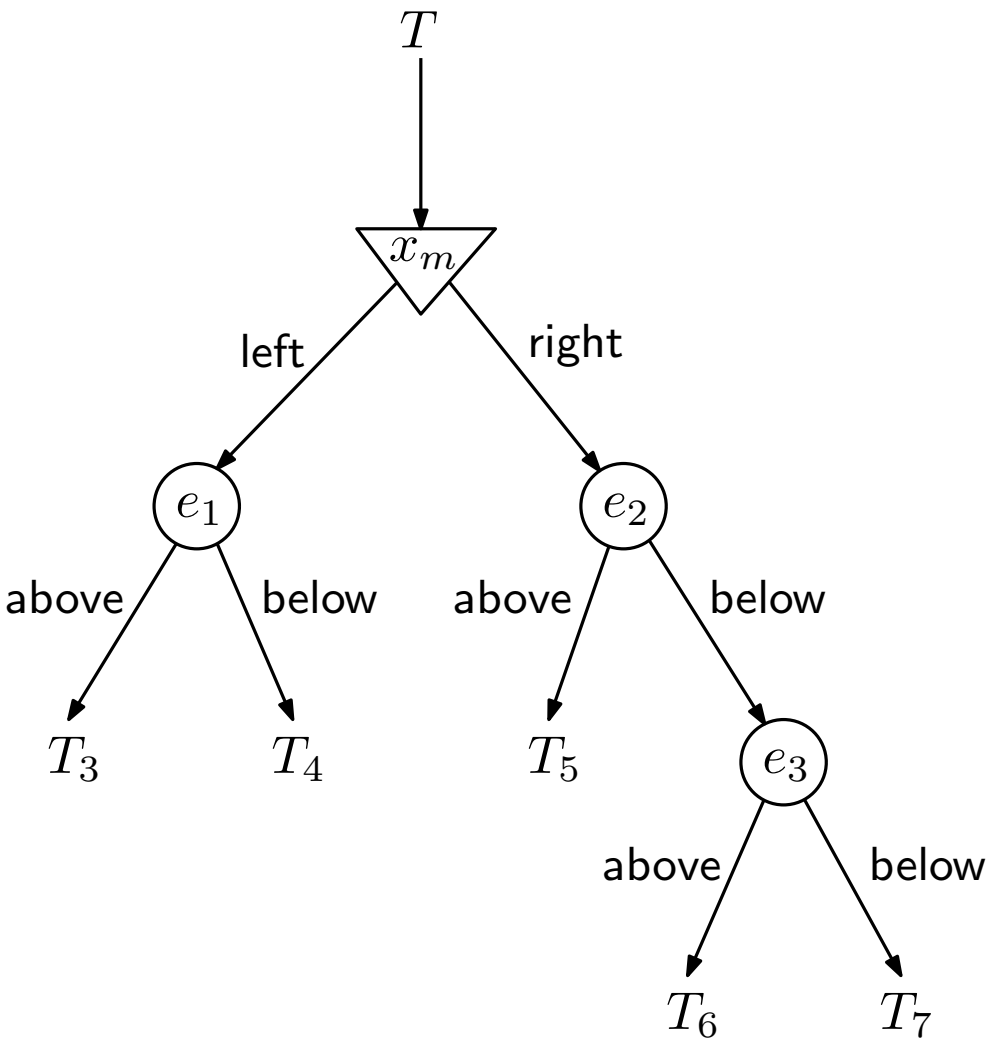
POINT LOCATION: Trapezoidal refinement

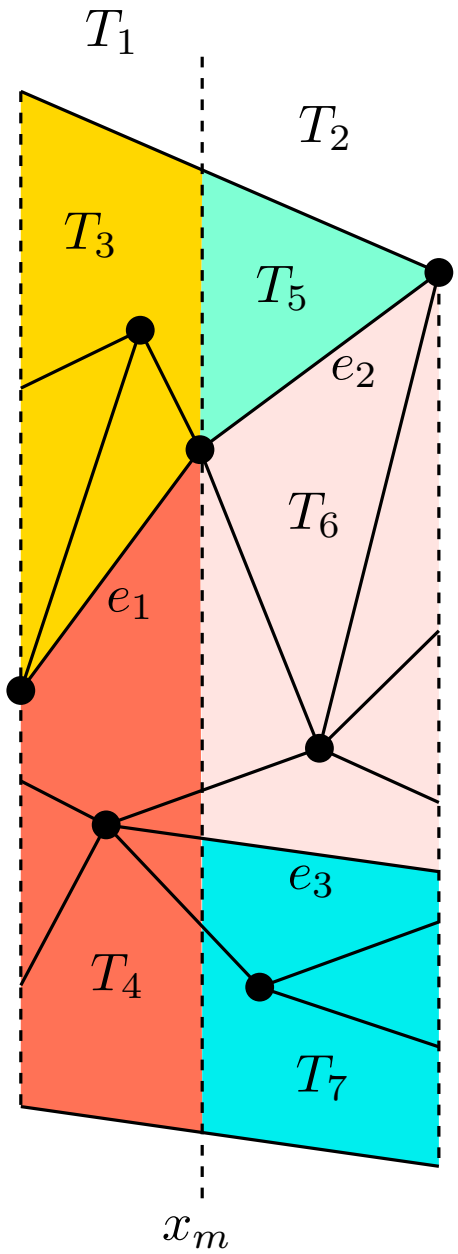
Search structure



POINT LOCATION: Trapezoidal refinement

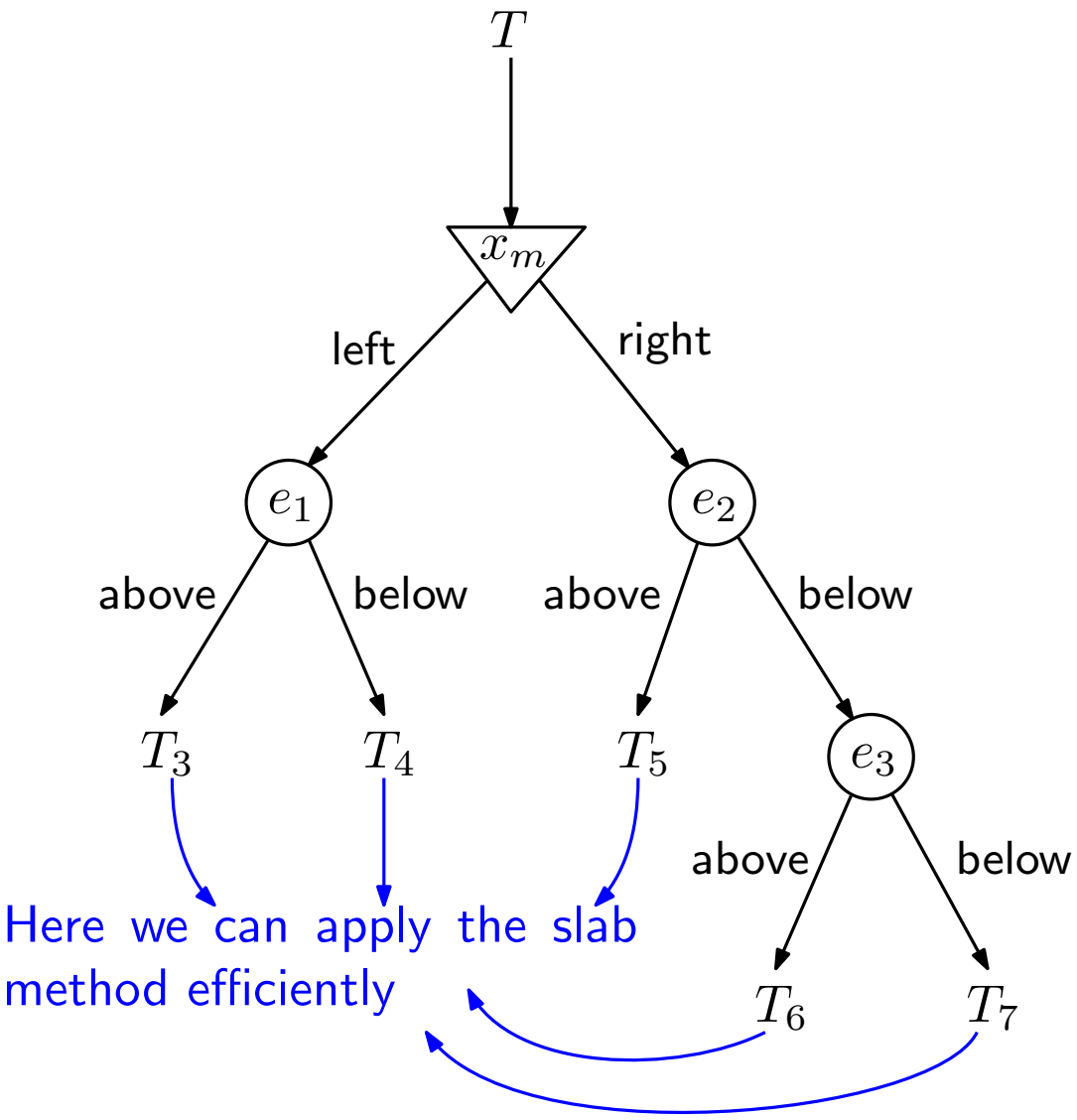
Search structure

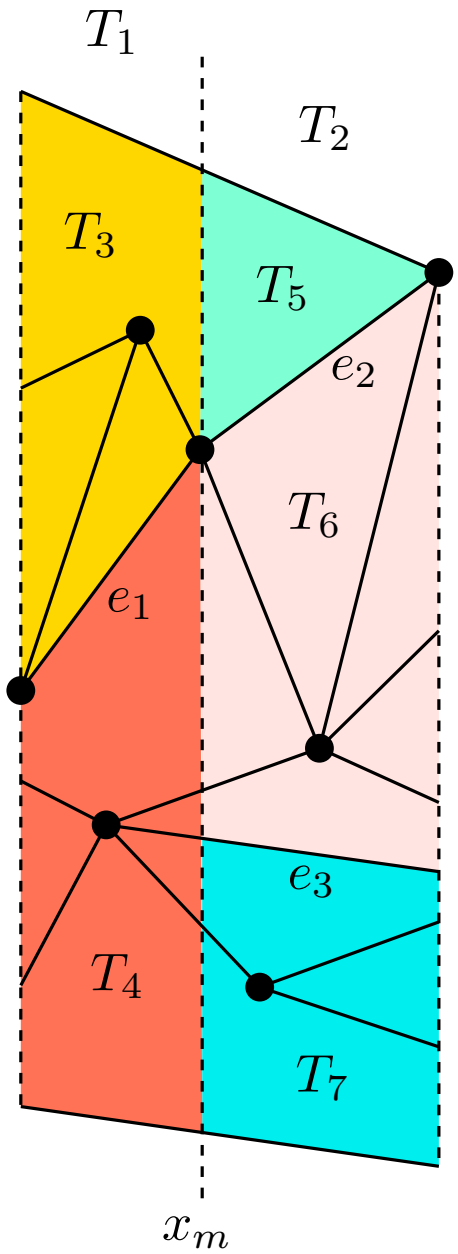




POINT LOCATION: Trapezoidal refinement

Search structure





POINT LOCATION: Trapezoidal refinement

Complexity

Space: the space used to store the hierarchy of trapezoidal decompositions is $O(n \log n)$:

- There is a triangular node for each vertex of the initial graph.
- There is a circular node for each "piece" of an edge of the initial graph.
- There is a leaf for each legal trapezoid. (Alternatively, one can also continue subdividing until all trapezoids are empty.)

It can be proved that this hierarchy cannot produce more than $O(n \log n)$ overall trapezoids.

Preprocessing: computing the refinement of the trapezoids is done in $O(n \log n)$ time.

Location: locating a point is done in $O(\log n)$ time.

Method 4: **Triangulation refinement – Kirkpatrick's algorithm**

POINT LOCATION: Triangulation refinement

This is a method for point location in triangulations, although it can be extended to more general decompositions. It is based on a refinement process of the triangulation, and it requires the exterior face of the triangulation to be triangular.

POINT LOCATION: Triangulation refinement

This is a method for point location in triangulations, although it can be extended to more general decompositions. It is based on a refinement process of the triangulation, and it requires the exterior face of the triangulation to be triangular.

Input

A triangulation T, whose exterior face is a triangle.

POINT LOCATION: Triangulation refinement

This is a method for point location in triangulations, although it can be extended to more general decompositions. It is based on a refinement process of the triangulation, and it requires the exterior face of the triangulation to be triangular.

Input

A triangulation T, whose exterior face is a triangle.

Preprocessing

Create a hierarchy of triangulations $S_0, S_1, \ldots, S_h$ such that:

- $S_0 = T$
- S_i is obtained from S_{i-1} as follows:
    1. Delete a set of independent vertices not belonging to the boundary of the convex hull, as well as all their incident edges.
    2. Retriangulate the resulting polygons.
- S_h is the enclosing triangle

POINT LOCATION: Triangulation refinement

This is a method for point location in triangulations, although it can be extended to more general decompositions. It is based on a refinement process of the triangulation, and it requires the exterior face of the triangulation to be triangular.

Input

A triangulation T, whose exterior face is a triangle.

Preprocessing

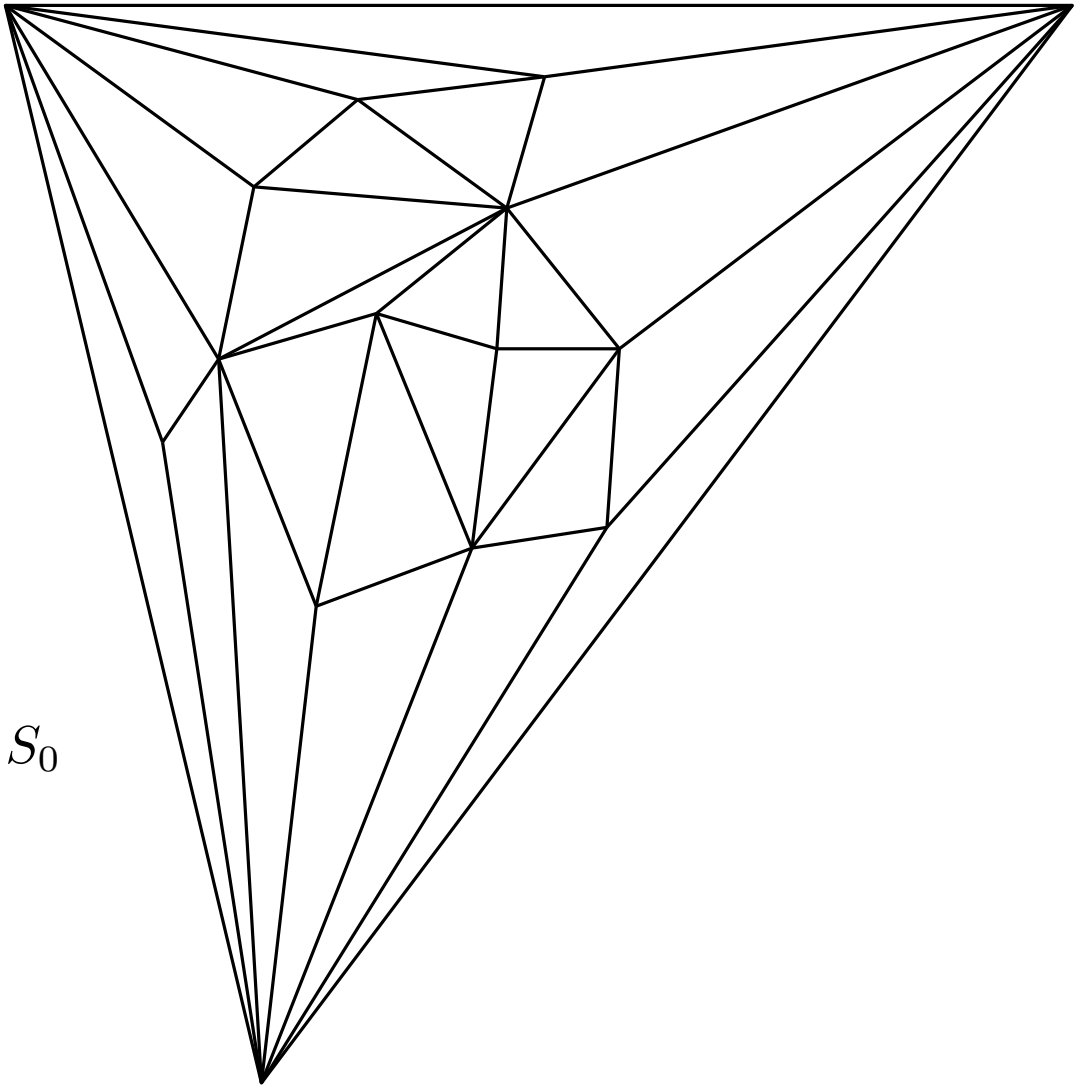
Create a hierarchy of triangulations $S_0, S_1, \ldots, S_h$ such that:

- $S_0 = T$
- S_i is obtained from S_{i-1} as follows:
  1. Delete a set of independent vertices not belonging to the boundary of the convex hull, as well as all their incident edges.
  2. Retriangulate the resulting polygons.
- S_h is the enclosing triangle

No two vertices are adjacent.

POINT LOCATION: Triangulation refinement

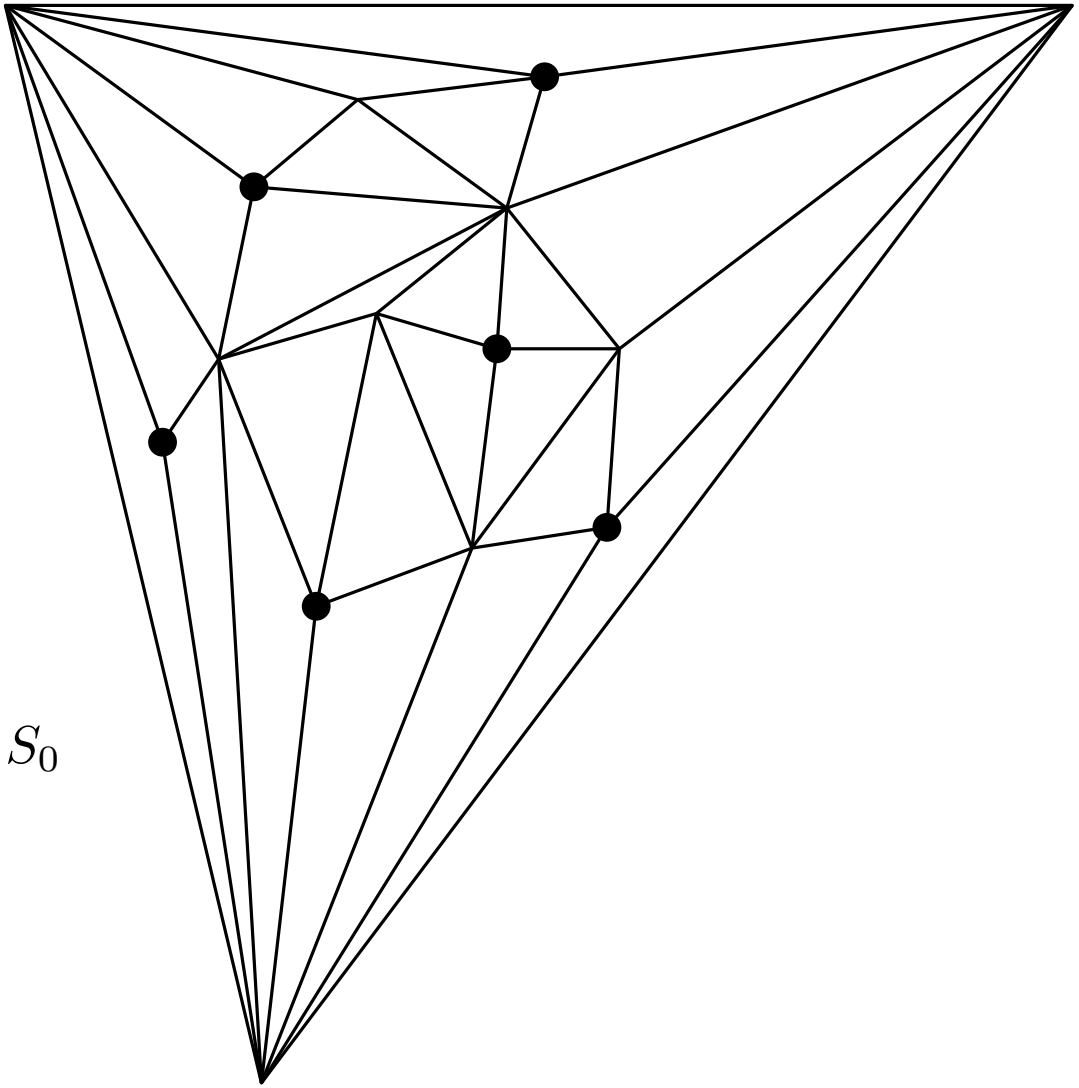
Preprocessing



S_0

POINT LOCATION: Triangulation refinement

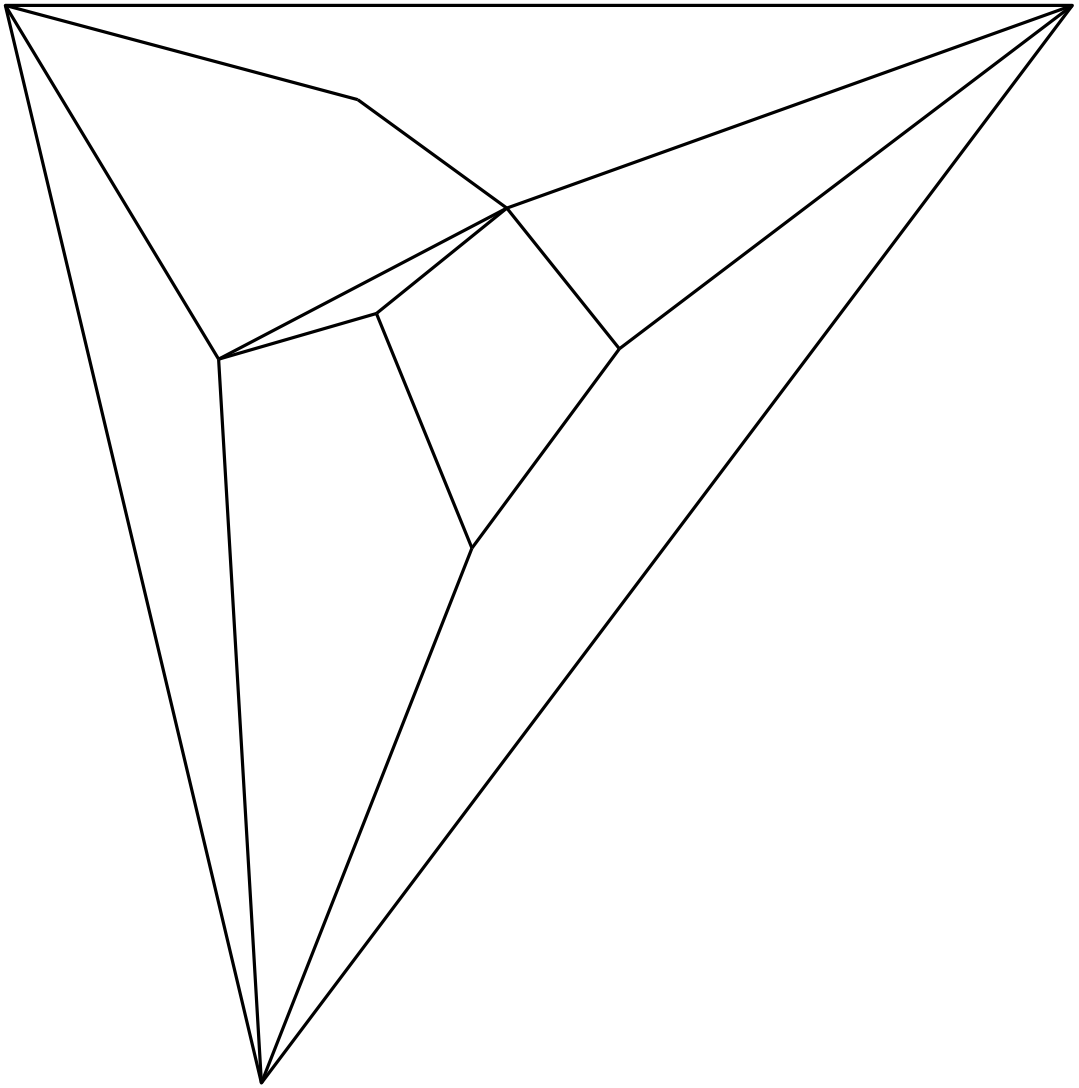
Preprocessing



S_0

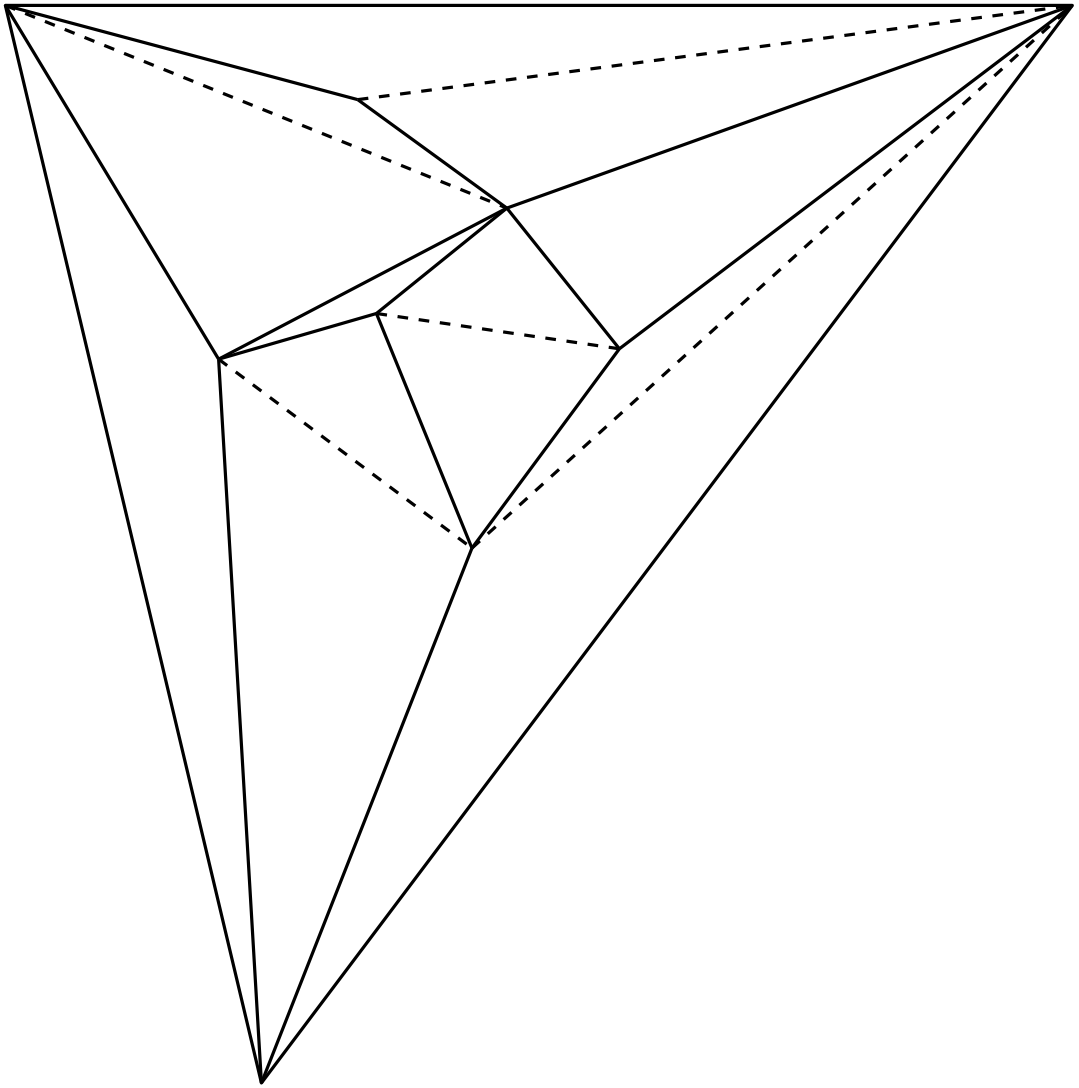
POINT LOCATION: Triangulation refinement

Preprocessing



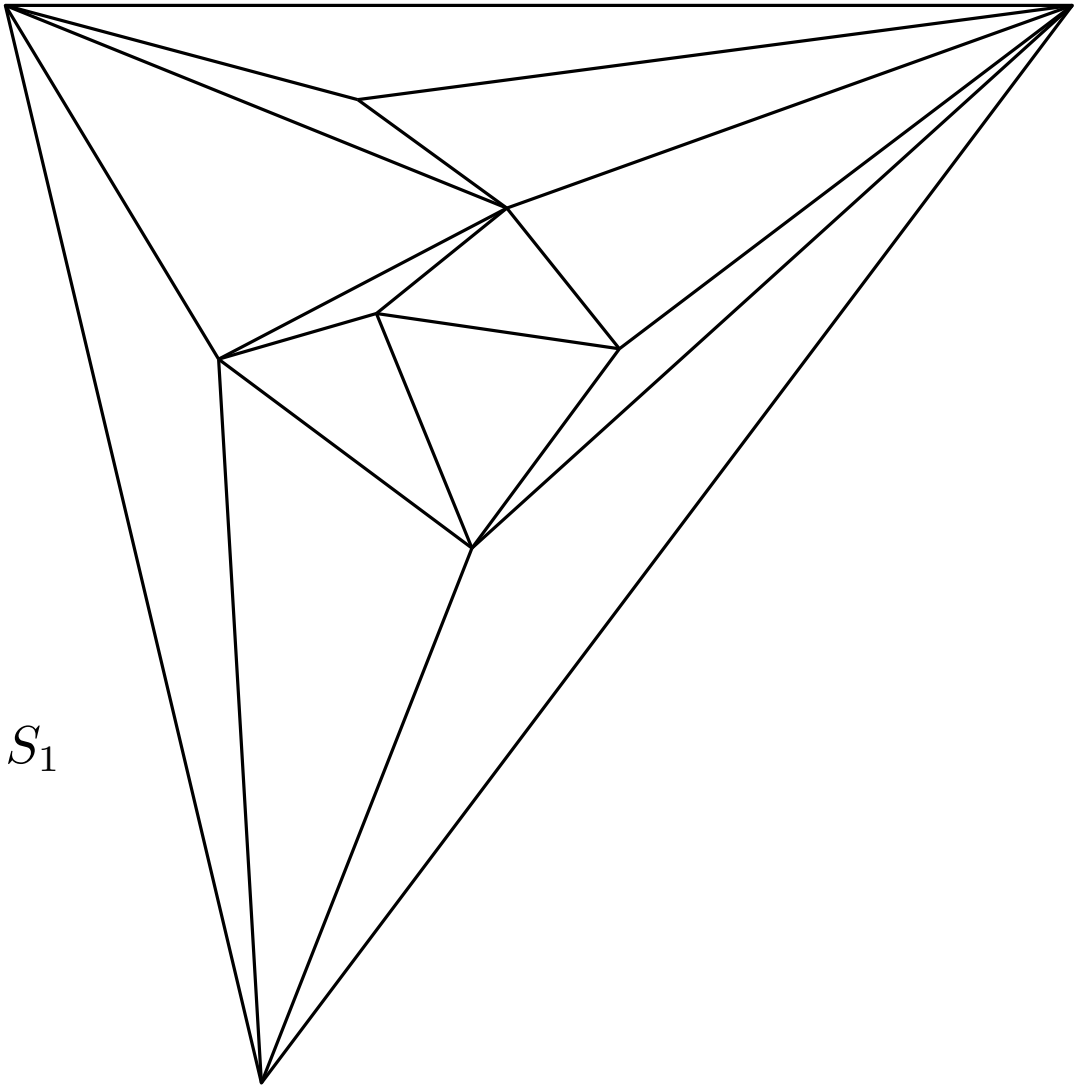
POINT LOCATION: Triangulation refinement

Preprocessing



POINT LOCATION: Triangulation refinement

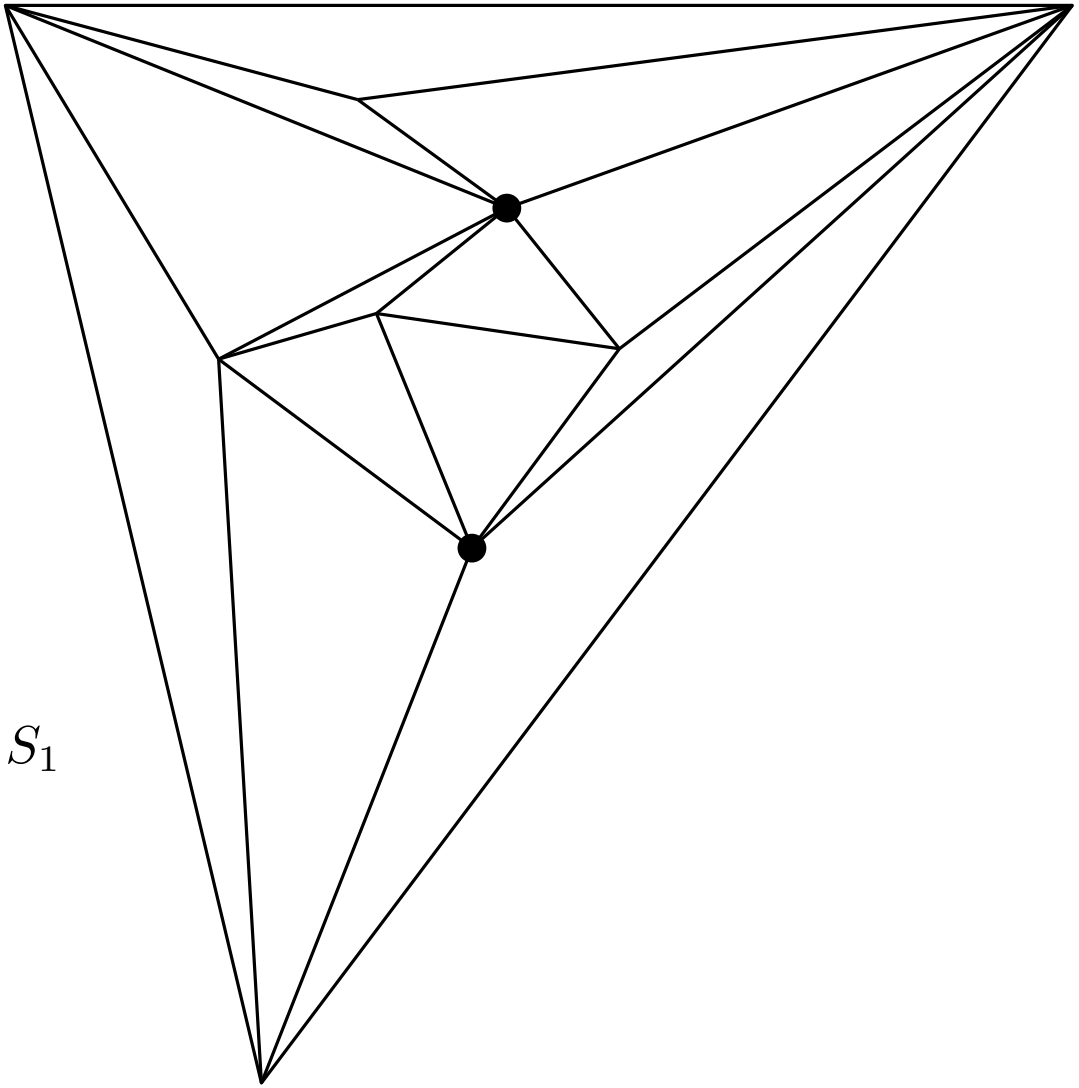
Preprocessing



S_1

POINT LOCATION: Triangulation refinement

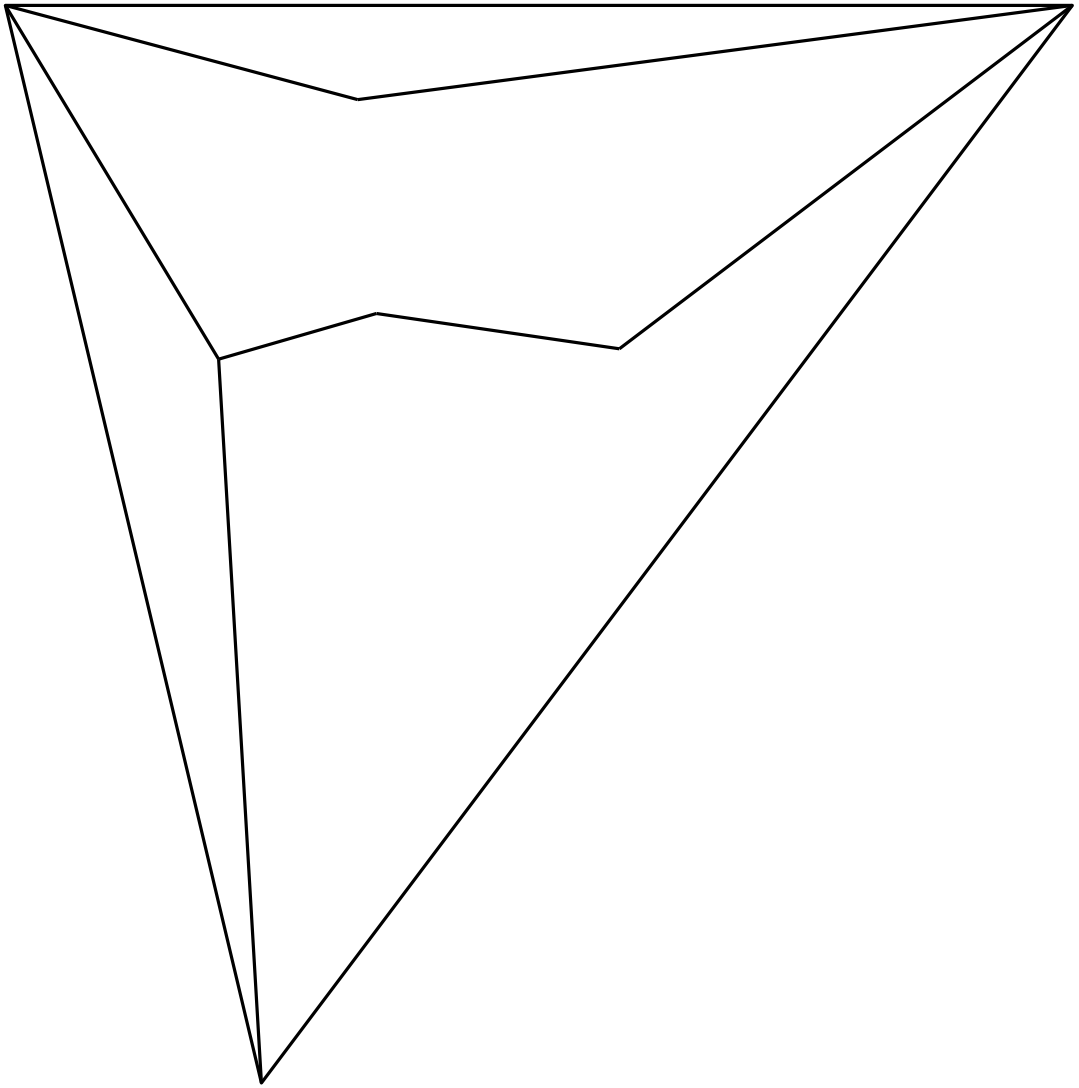
Preprocessing



S_1

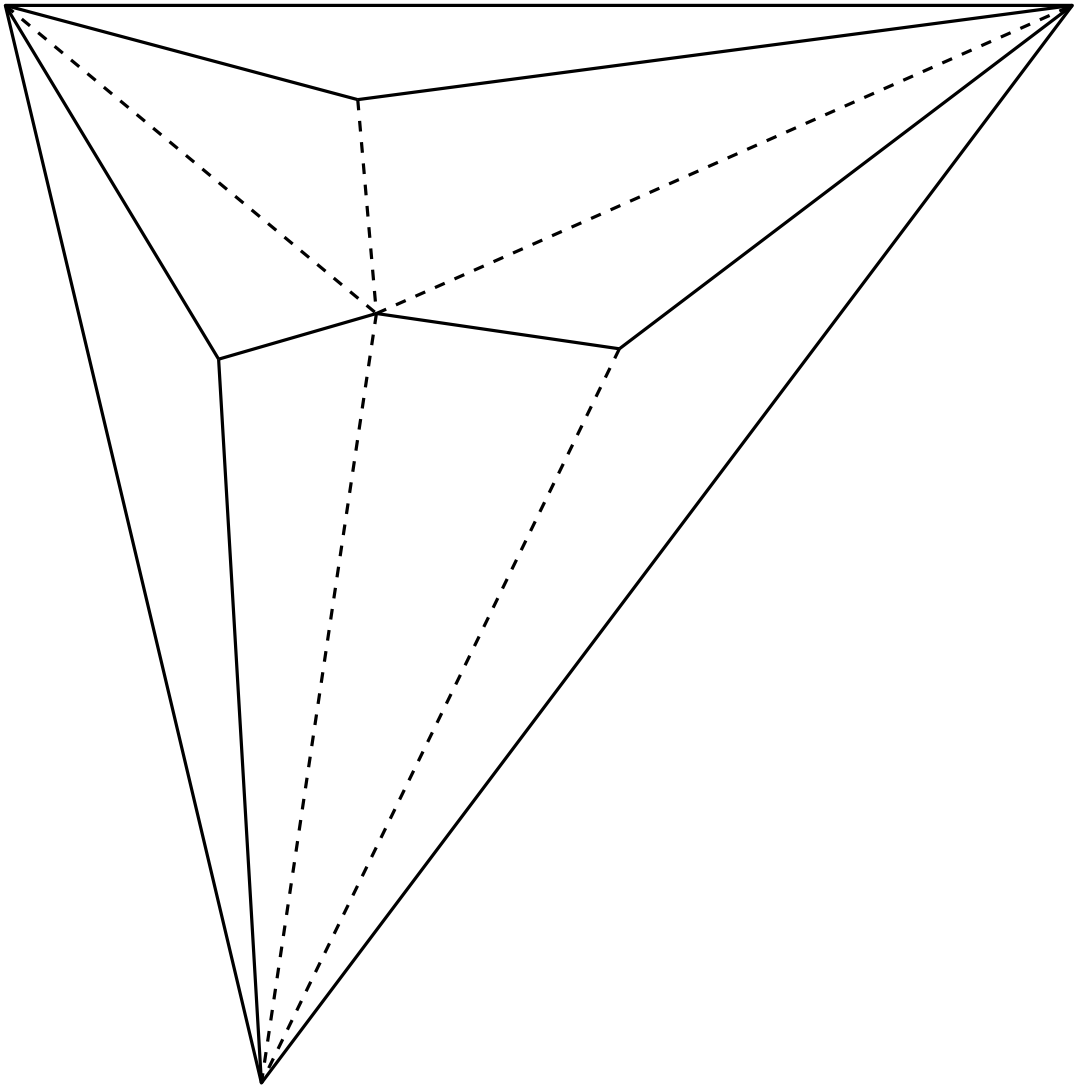
POINT LOCATION: Triangulation refinement

Preprocessing



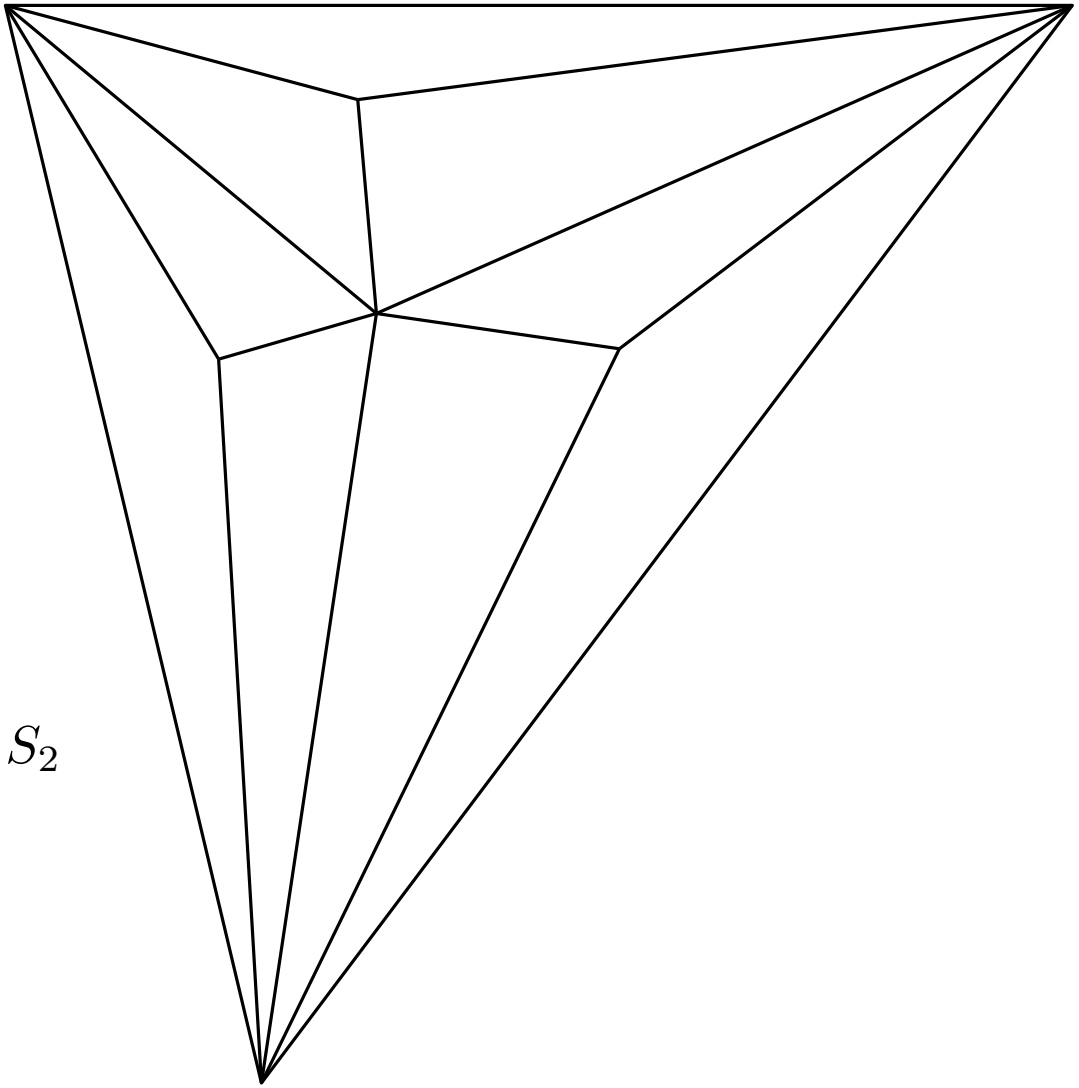
POINT LOCATION: Triangulation refinement

Preprocessing



POINT LOCATION: Triangulation refinement

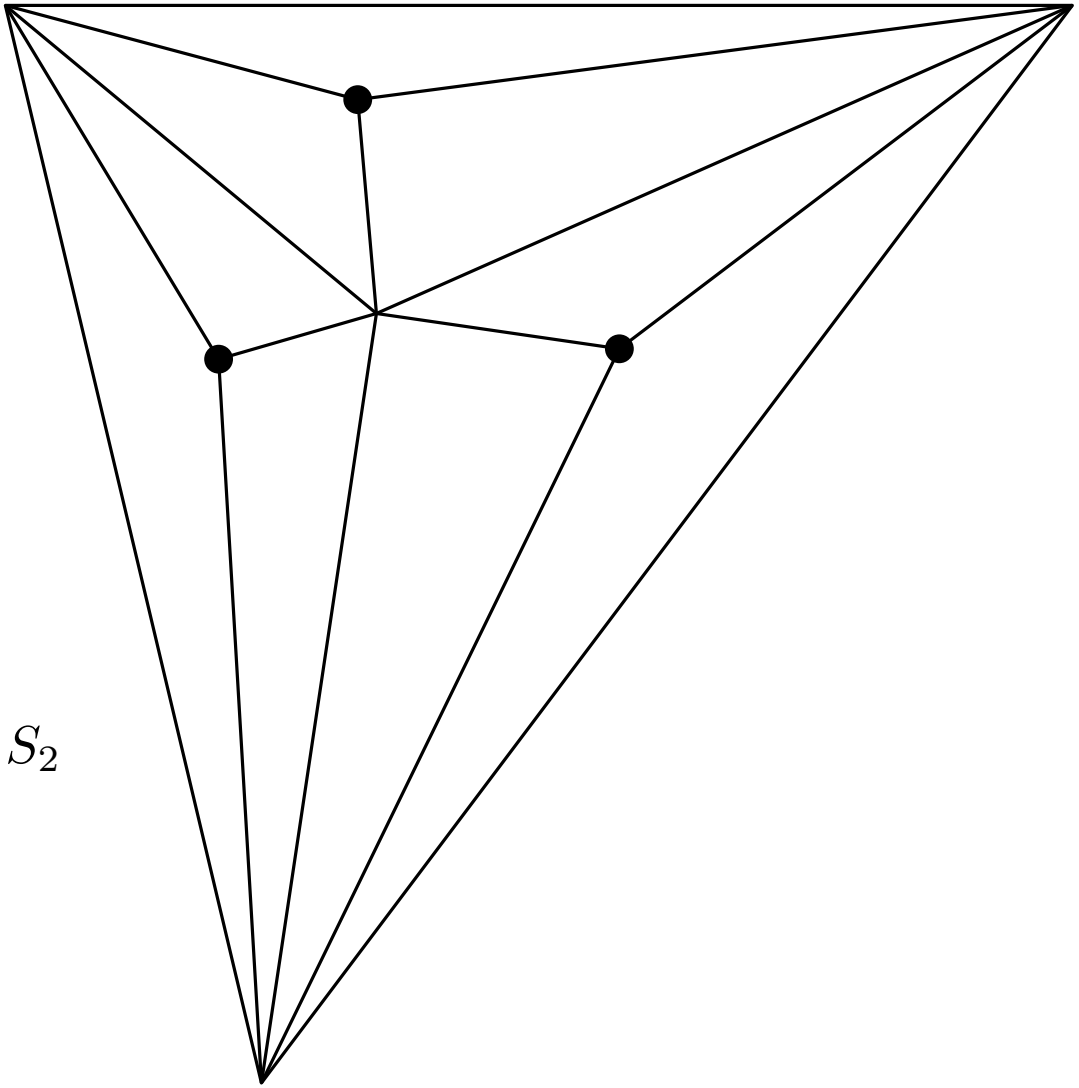
Preprocessing



S_2

POINT LOCATION: Triangulation refinement

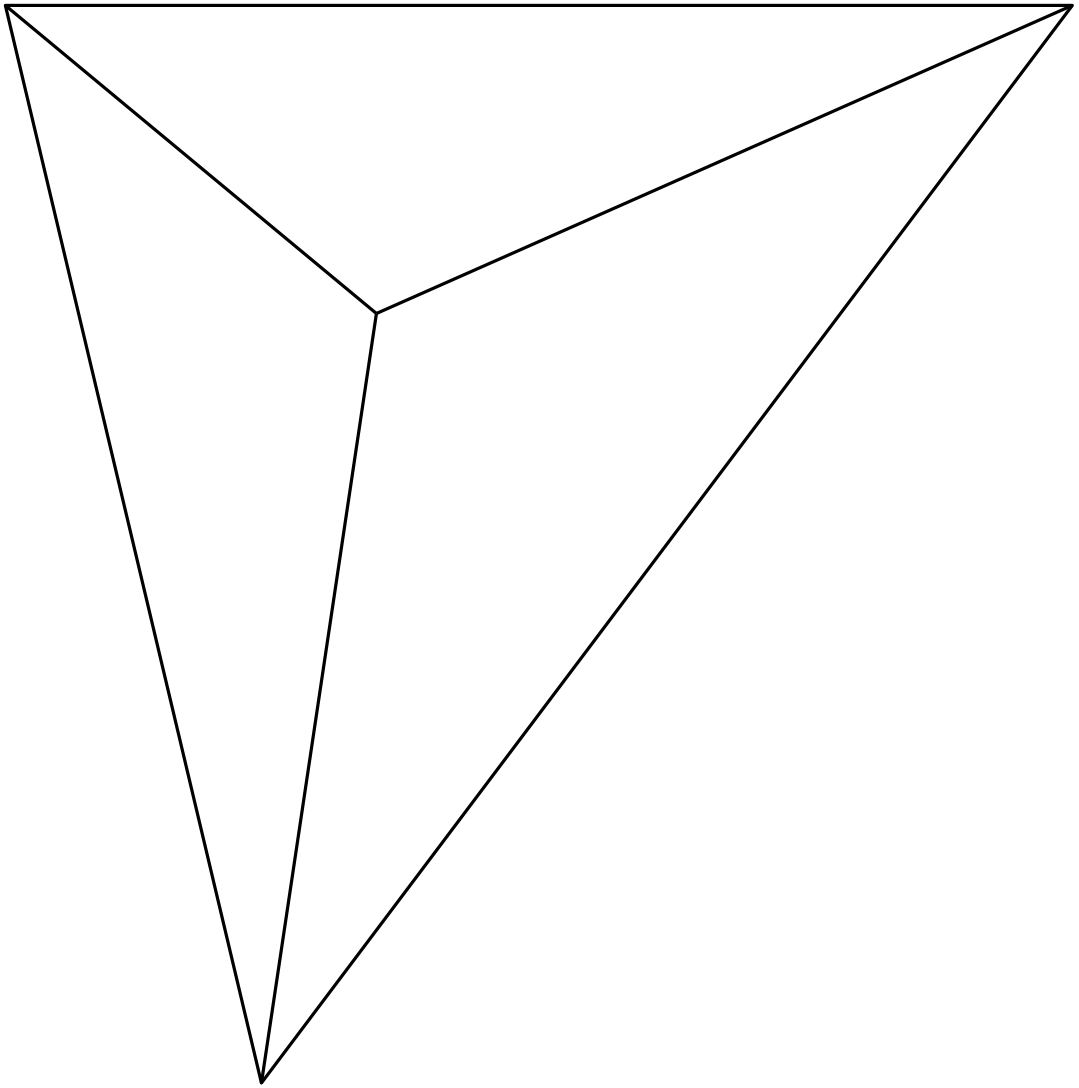
Preprocessing



S_2

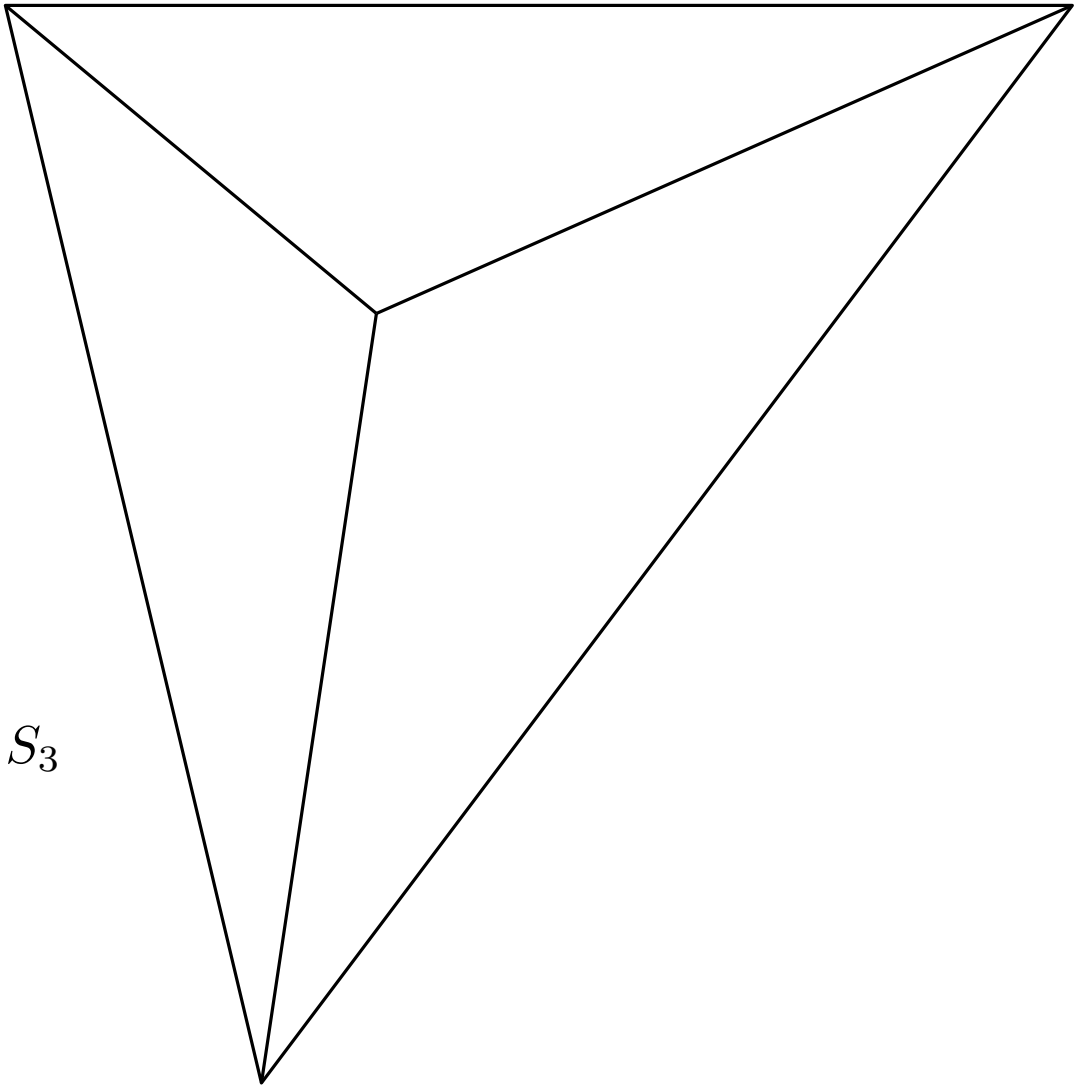
POINT LOCATION: Triangulation refinement

Preprocessing



POINT LOCATION: Triangulation refinement

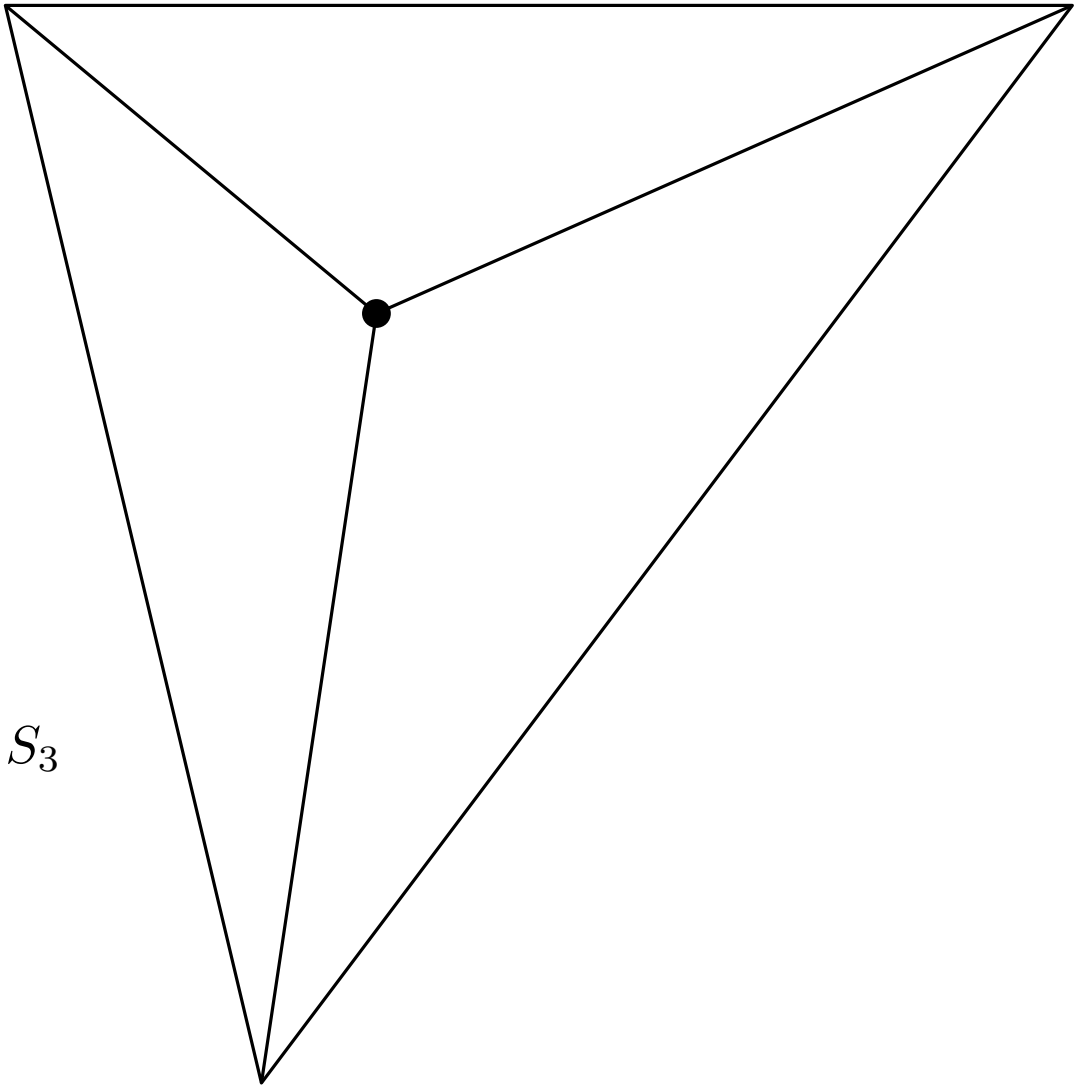
Preprocessing



S_3

POINT LOCATION: Triangulation refinement

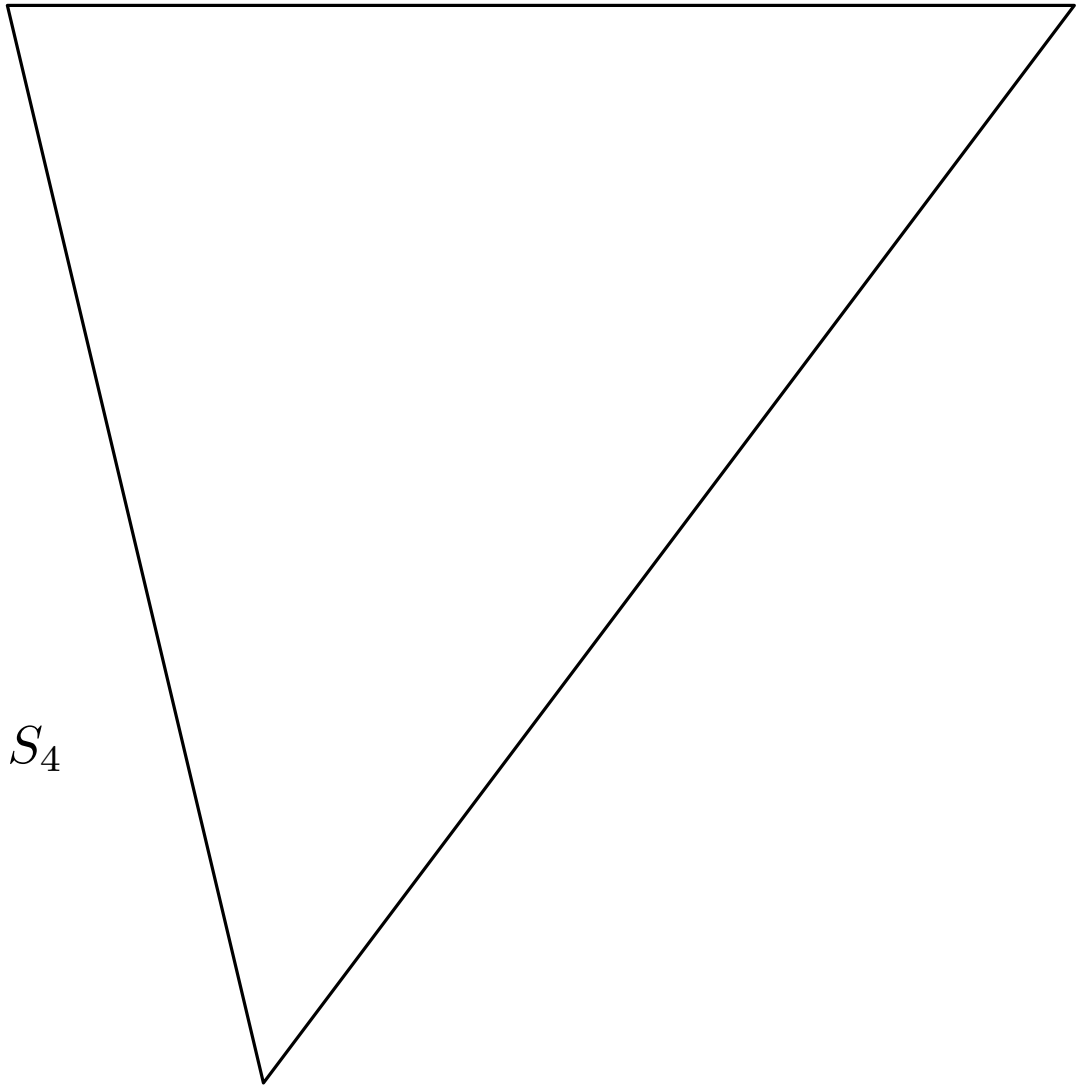
Preprocessing



S_3

POINT LOCATION: Triangulation refinement

Preprocessing



S_4

POINT LOCATION: Triangulation refinement

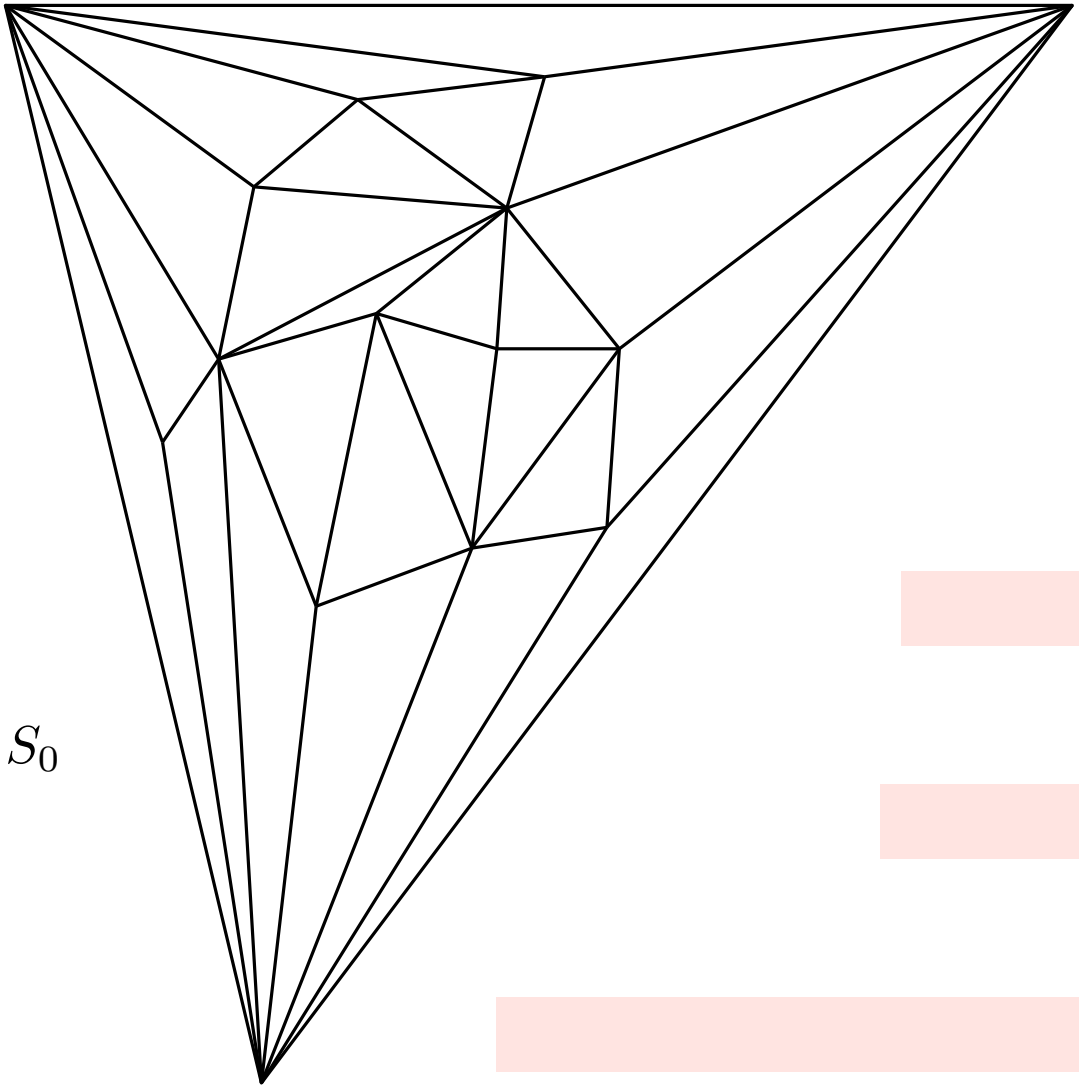
Preprocessing

The search structure to be build is a directed tree:

- The vertices of the tree are the triangles of the hierarchy of triangulations.
- There exists an edge from triangle T_k to triangle T_j if, when computing S_i from S_{i-1}:
  - T_j is deleted from S_{i-1} in step 1.
  - T_k is created in S_i in step 2.
  - $T_k \cap T_j \neq \emptyset$.

POINT LOCATION: Triangulation refinement

Preprocessing



S_4

S_3

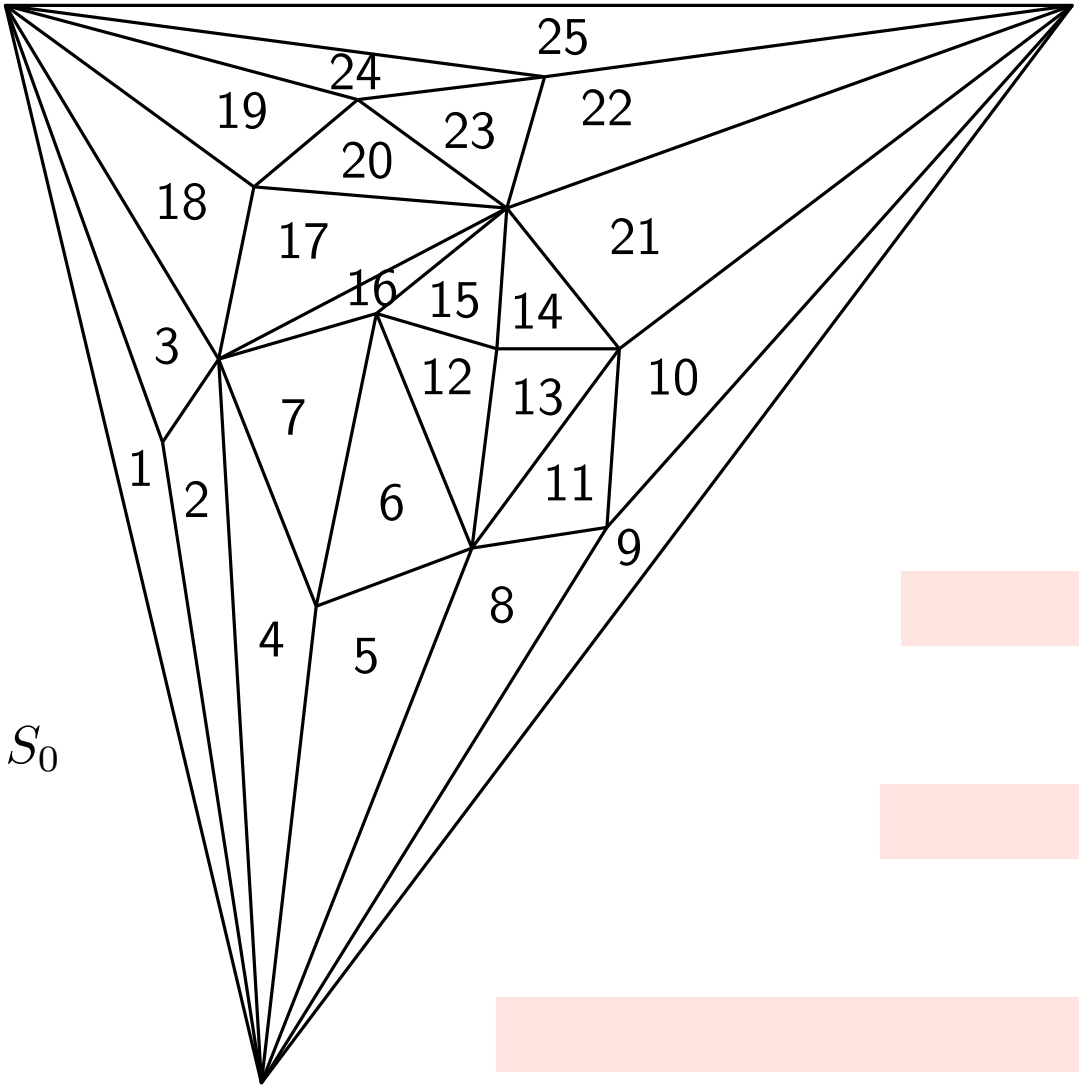
S_2

S_1

S_0

POINT LOCATION: Triangulation refinement

Preprocessing



S_4

S_3

S_2

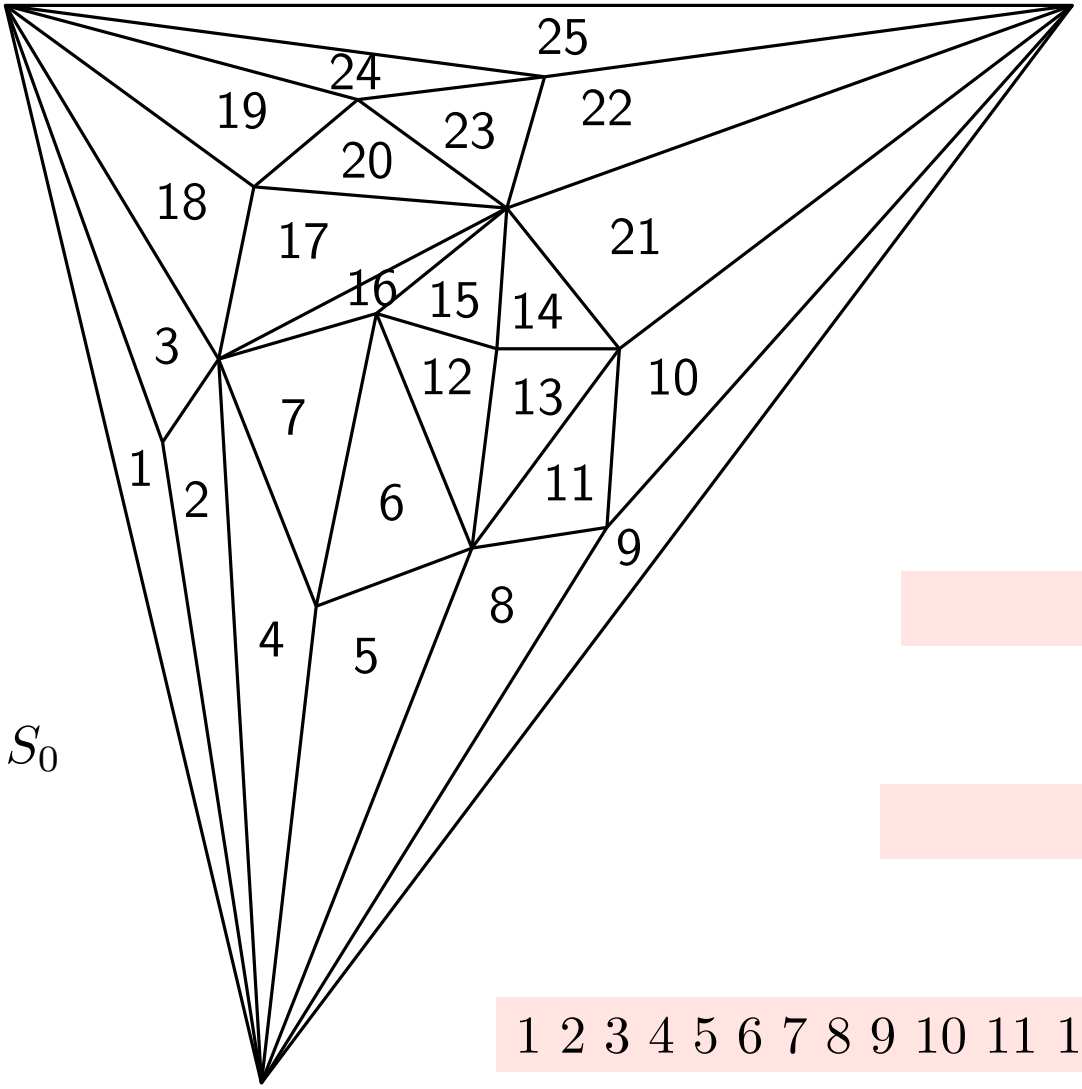
S_1

S_0

S_0

POINT LOCATION: Triangulation refinement

Preprocessing



S_0

S_4

S_3

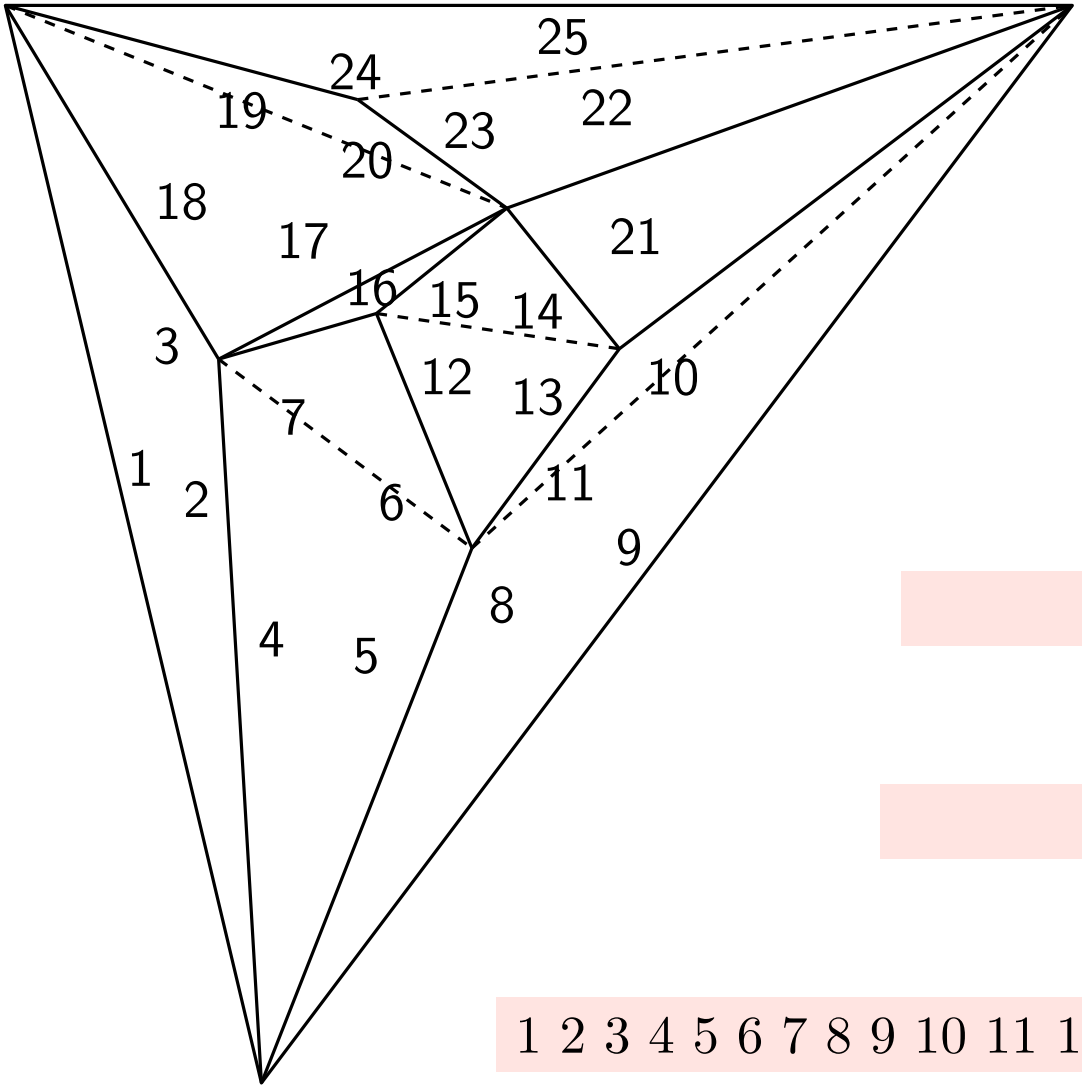
S_2

S_1

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 S_0

POINT LOCATION: Triangulation refinement

Preprocessing



S_4

S_3

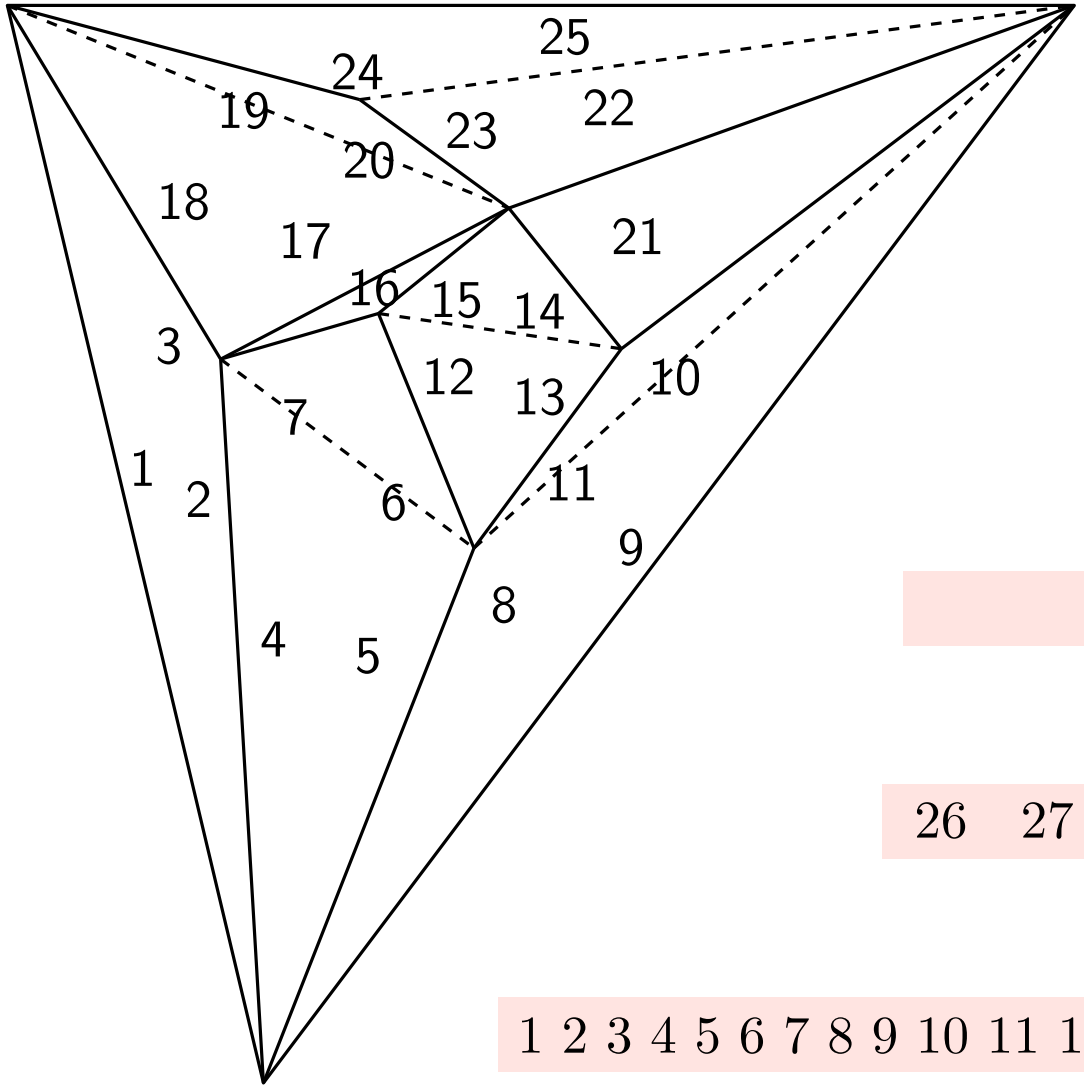
S_2

S_1

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 S_0

POINT LOCATION: Triangulation refinement

Preprocessing



S_4

S_3

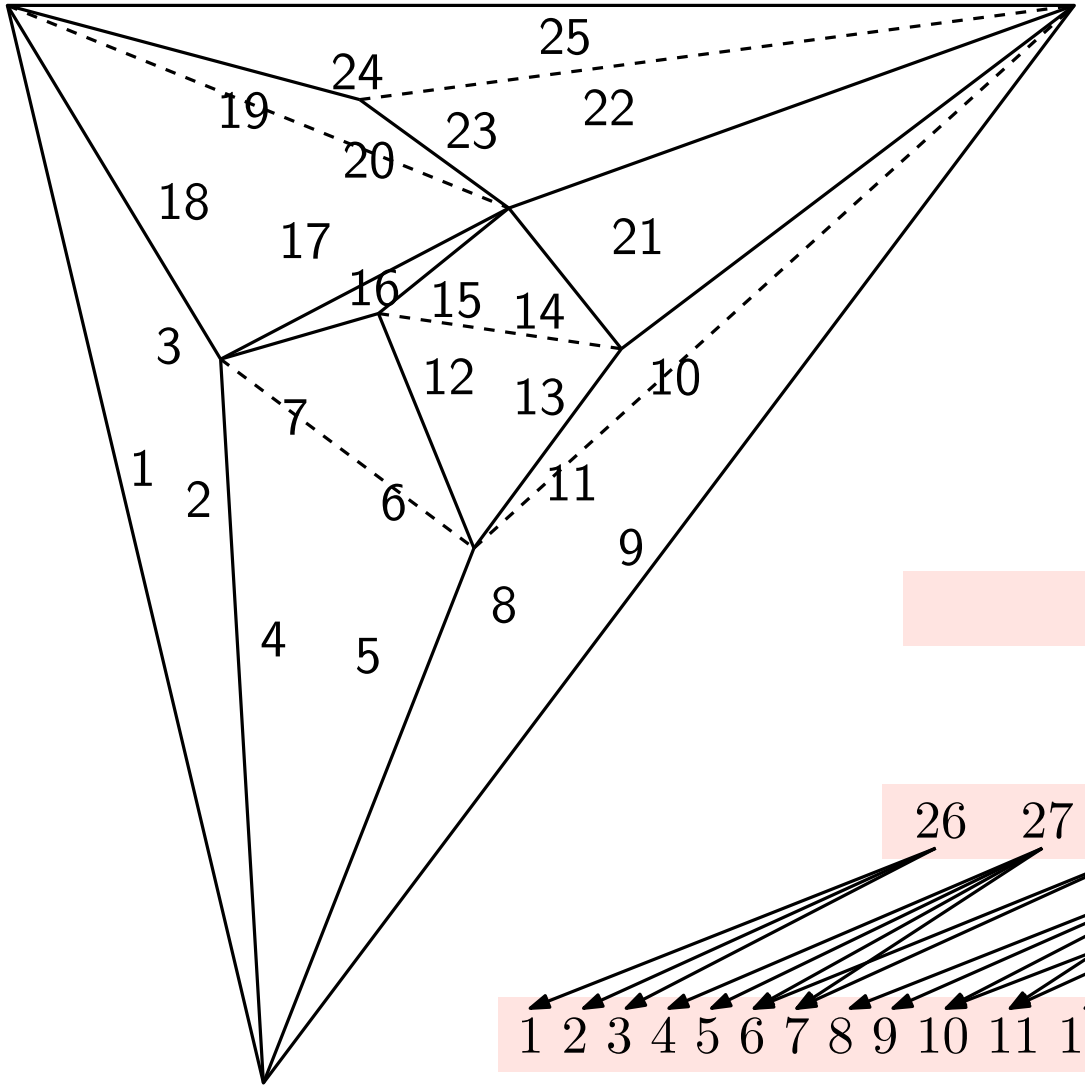
S_2

26 27 28 29 30 31 32 33 34 35 36 S_1

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 S_0

POINT LOCATION: Triangulation refinement

Preprocessing



S_4

S_3

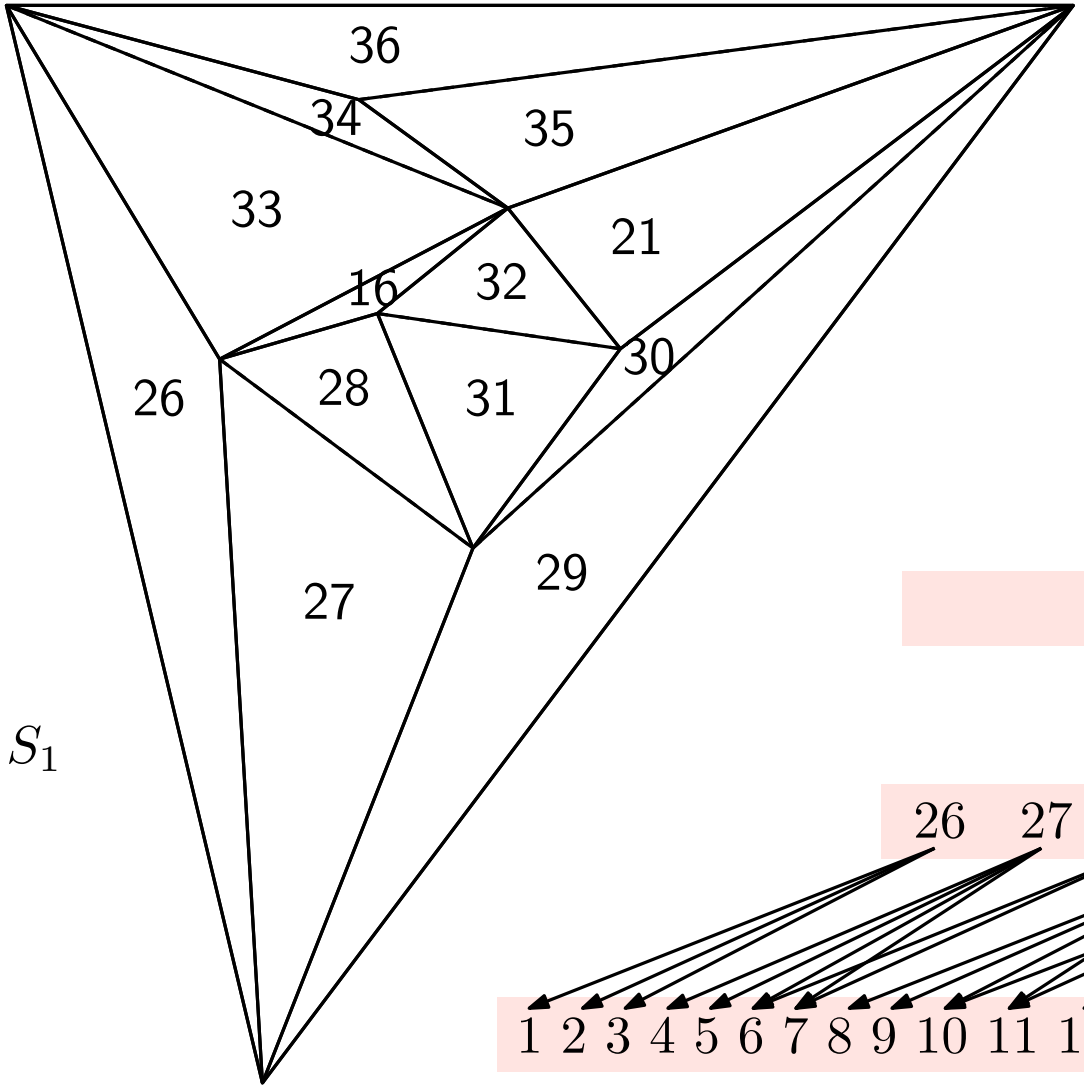
S_2

26 27 28 29 30 31 32 33 34 35 36 S_1

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 S_0

POINT LOCATION: Triangulation refinement

Preprocessing



S_4

S_3

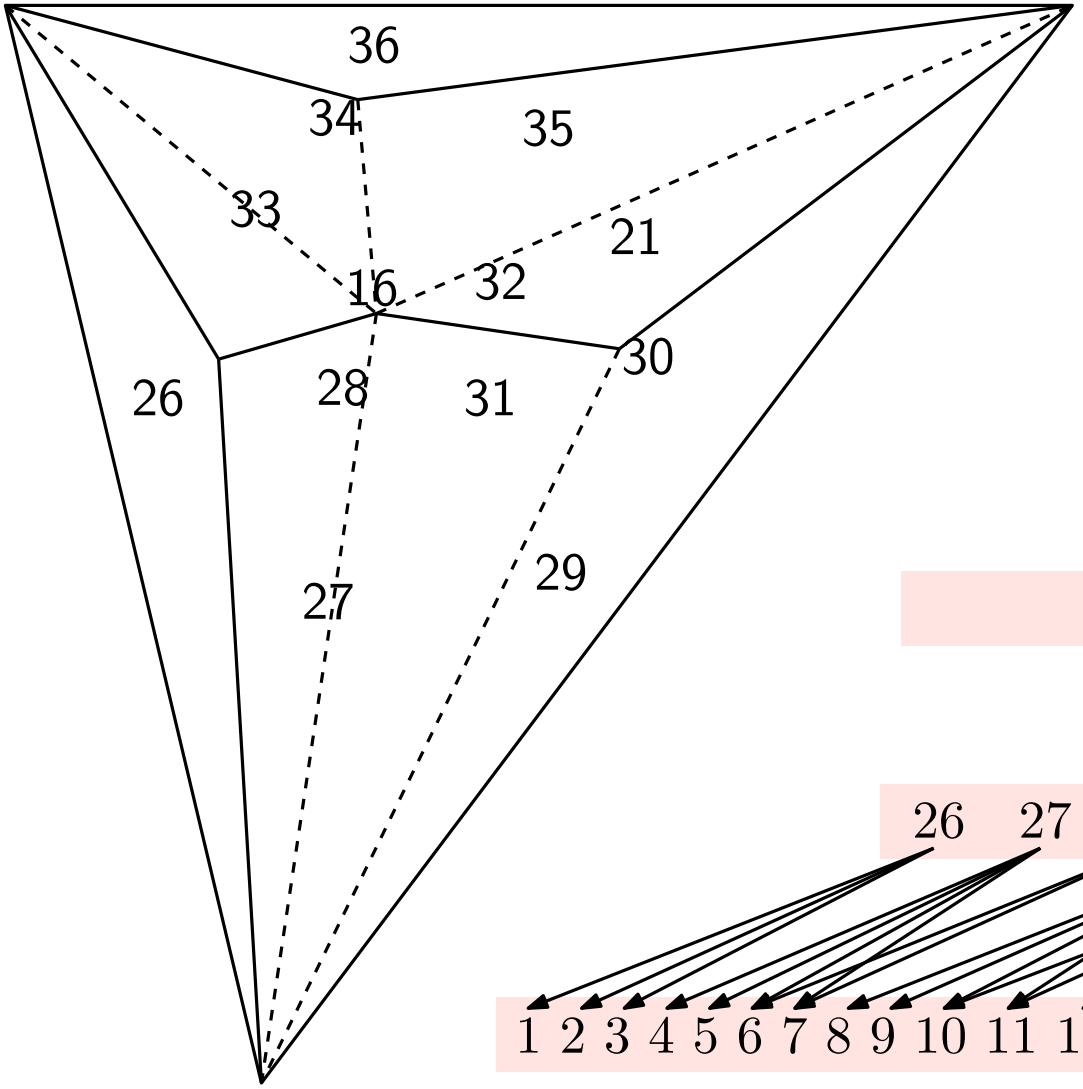
S_2

26 27 28 29 30 31 32 33 34 35 36 S_1

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 S_0

POINT LOCATION: Triangulation refinement

Preprocessing



S_4

S_3

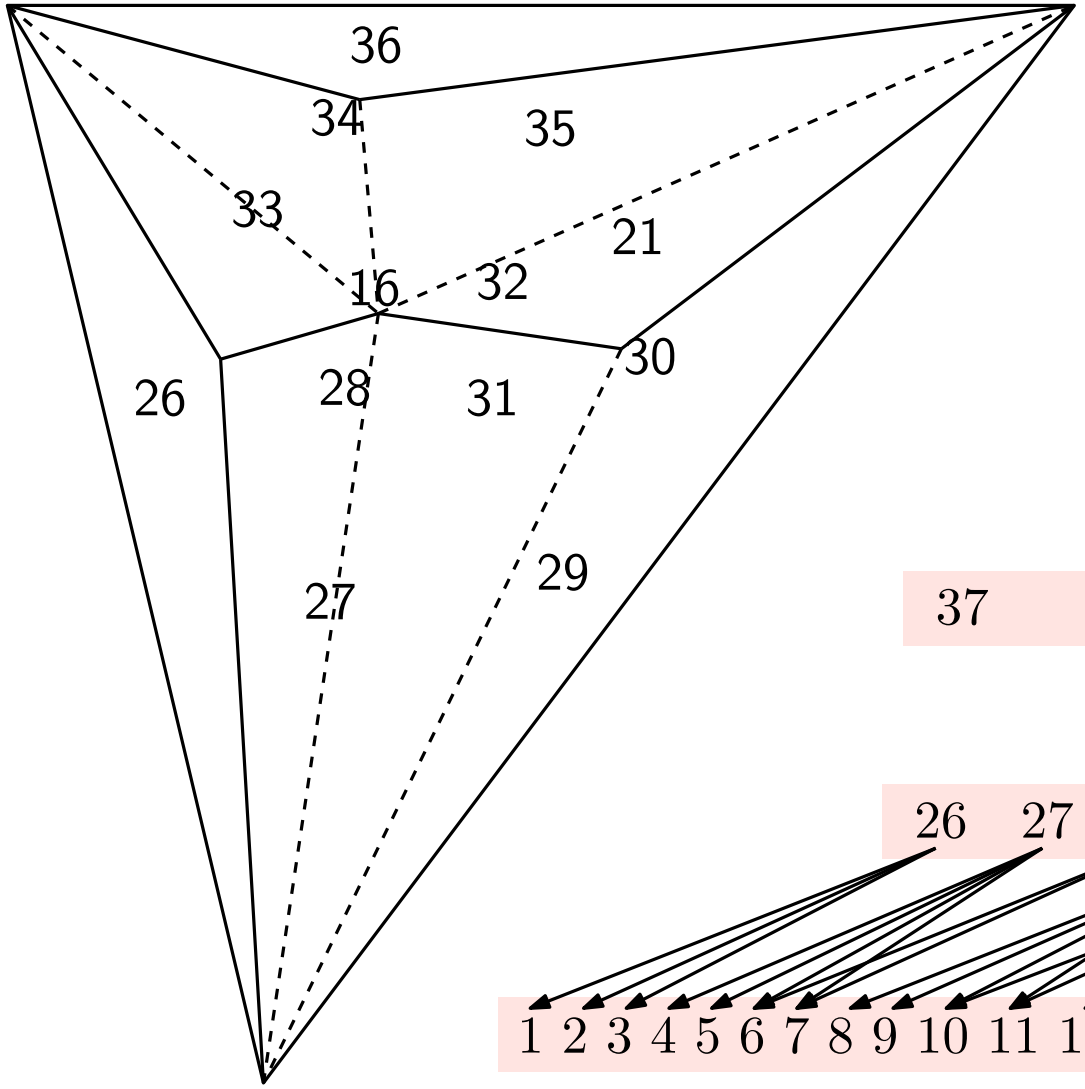
S_2

26 27 28 29 30 31 32 33 34 35 36 S_1

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 S_0

POINT LOCATION: Triangulation refinement

Preprocessing



S_4

S_3

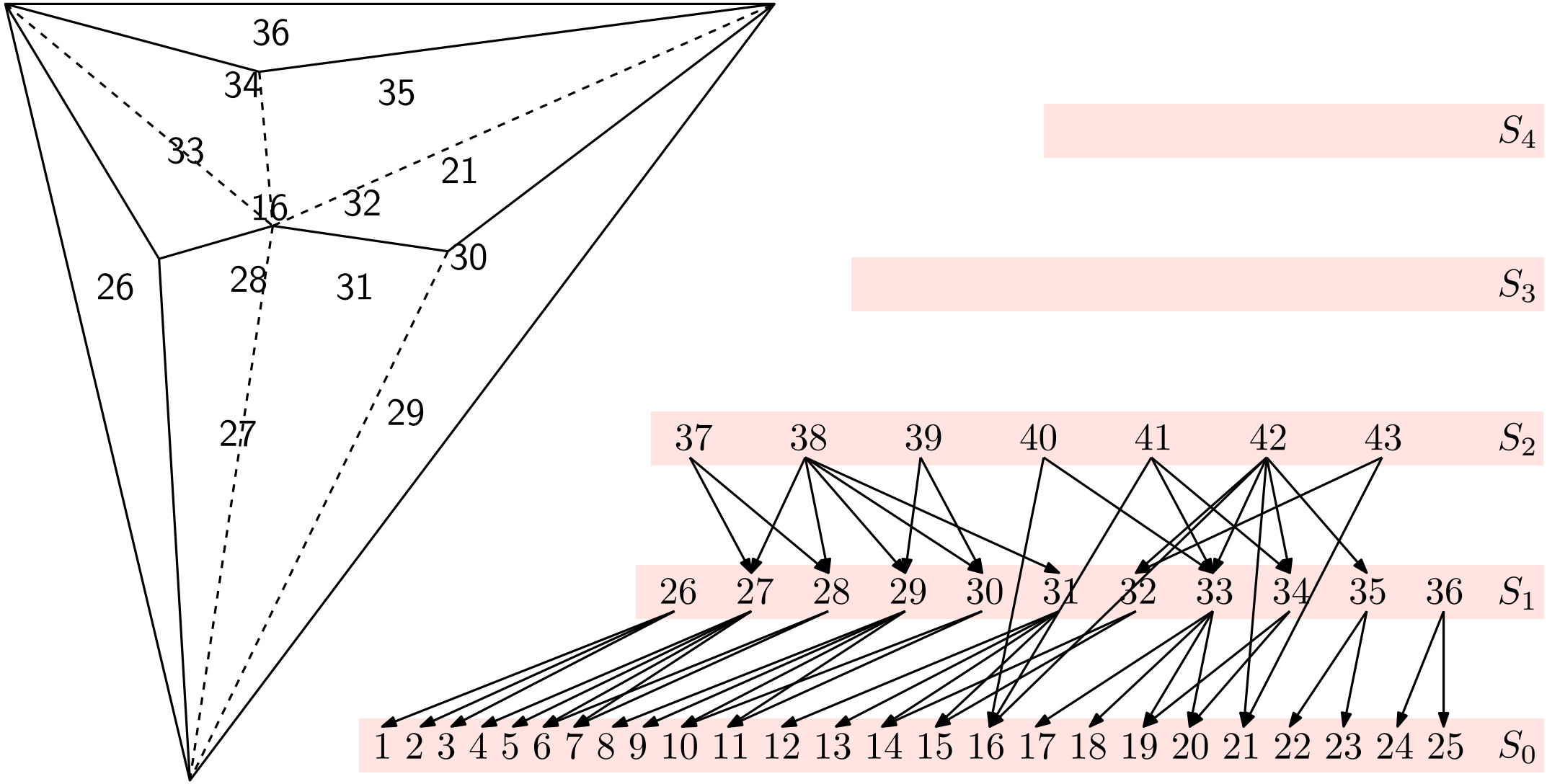
37 38 39 40 41 42 43 S_2

26 27 28 29 30 31 32 33 34 35 36 S_1

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 S_0

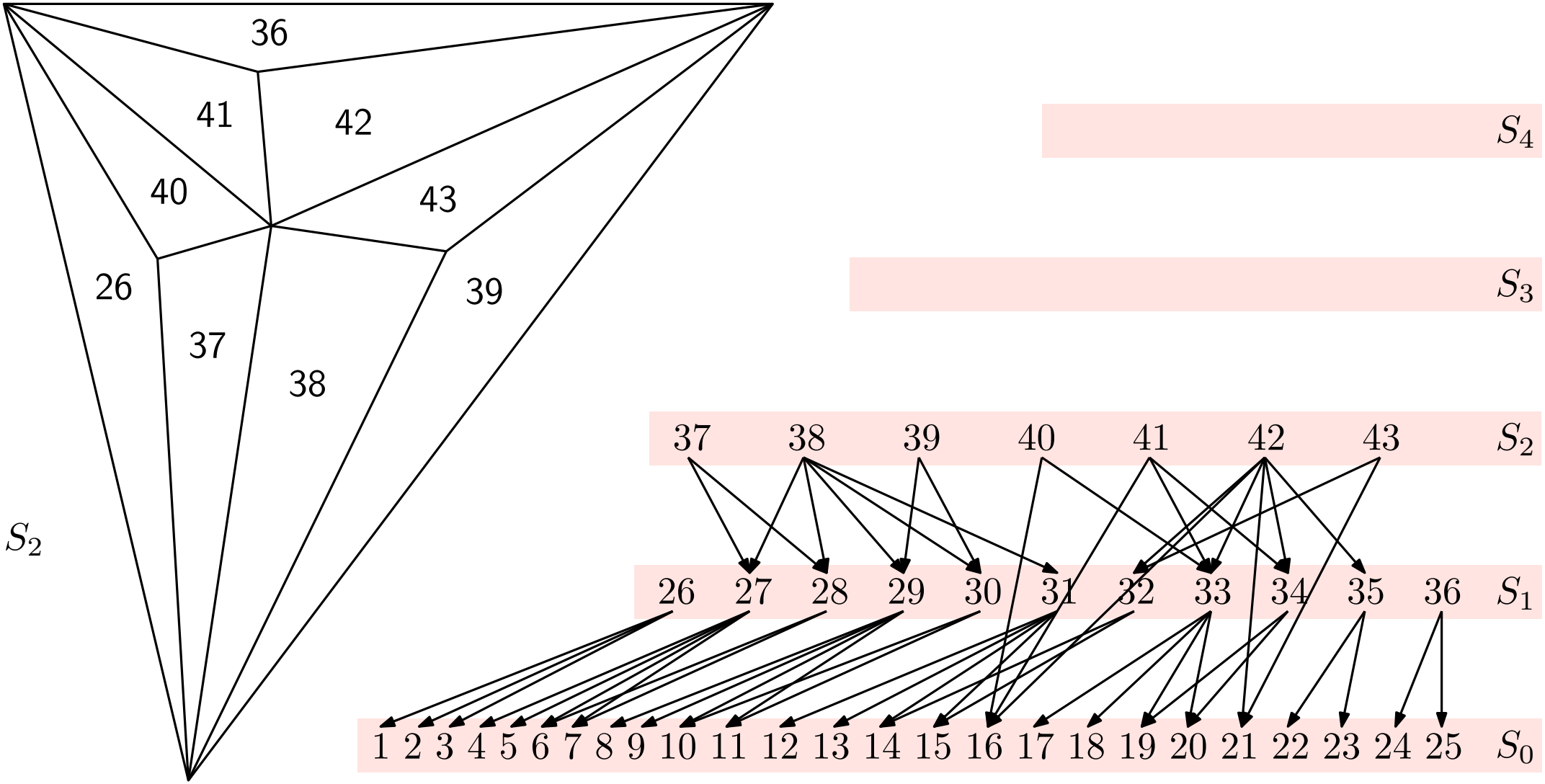
POINT LOCATION: Triangulation refinement

Preprocessing



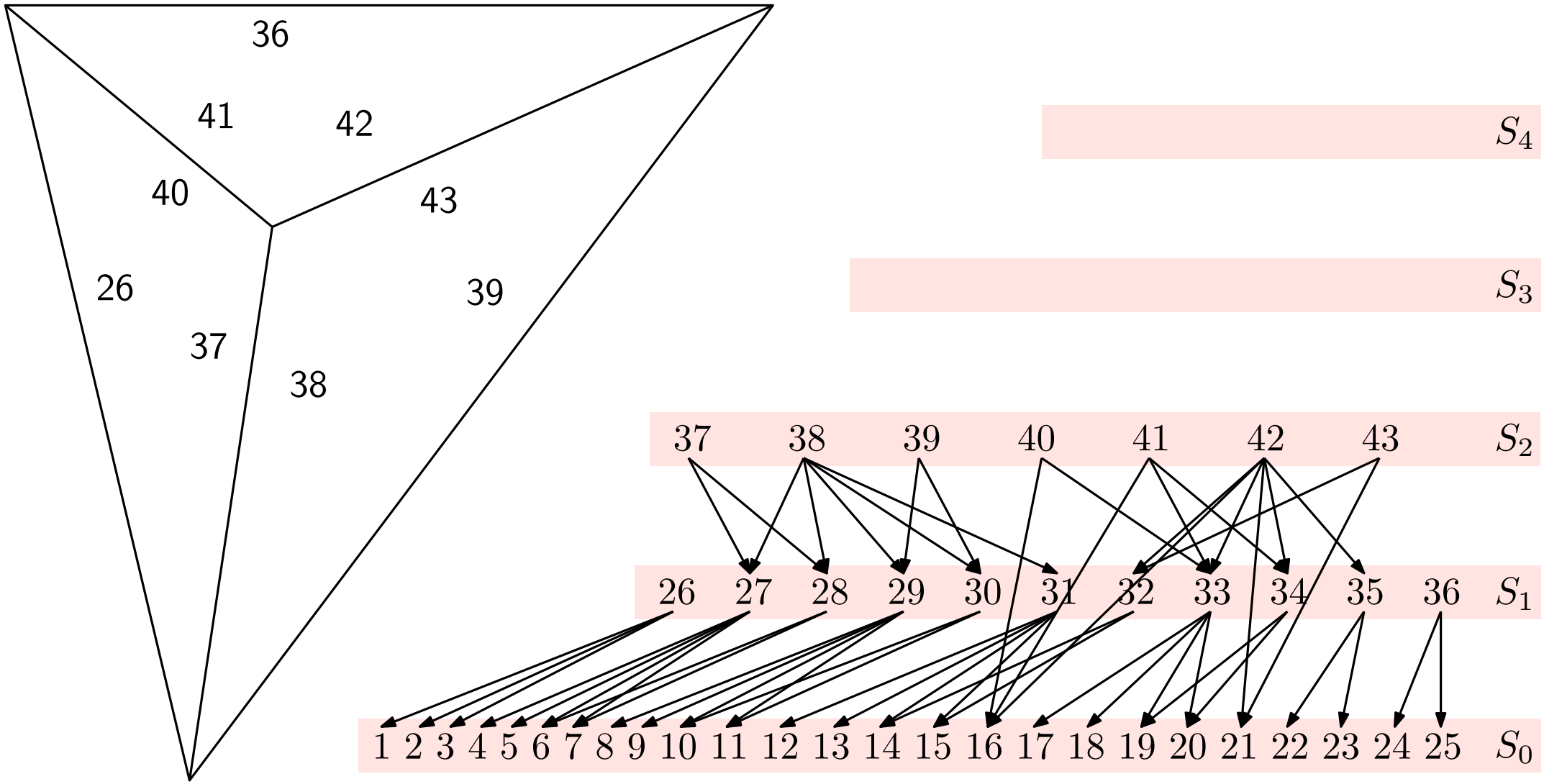
POINT LOCATION: Triangulation refinement

Preprocessing



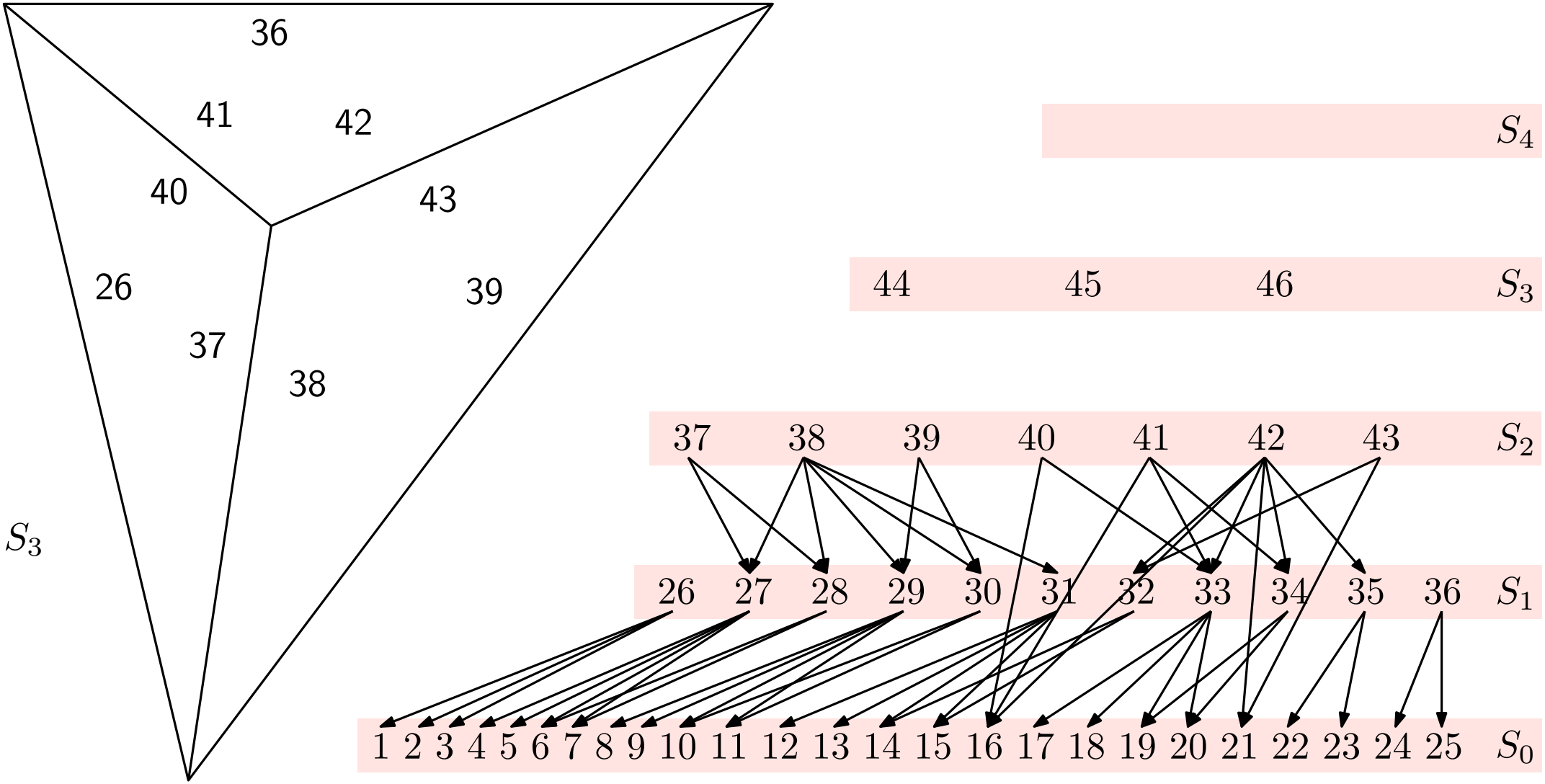
POINT LOCATION: Triangulation refinement

Preprocessing



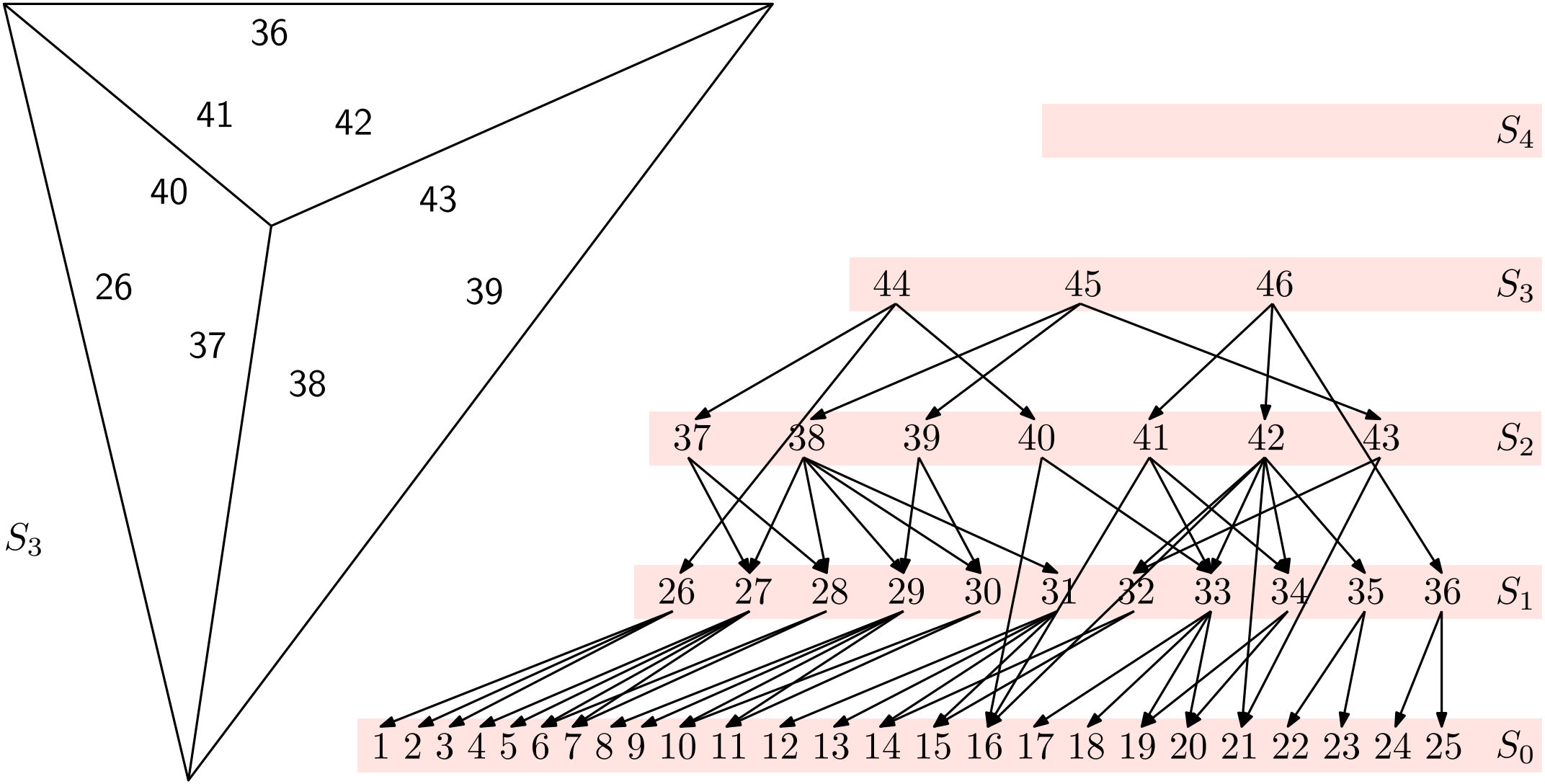
POINT LOCATION: Triangulation refinement

Preprocessing



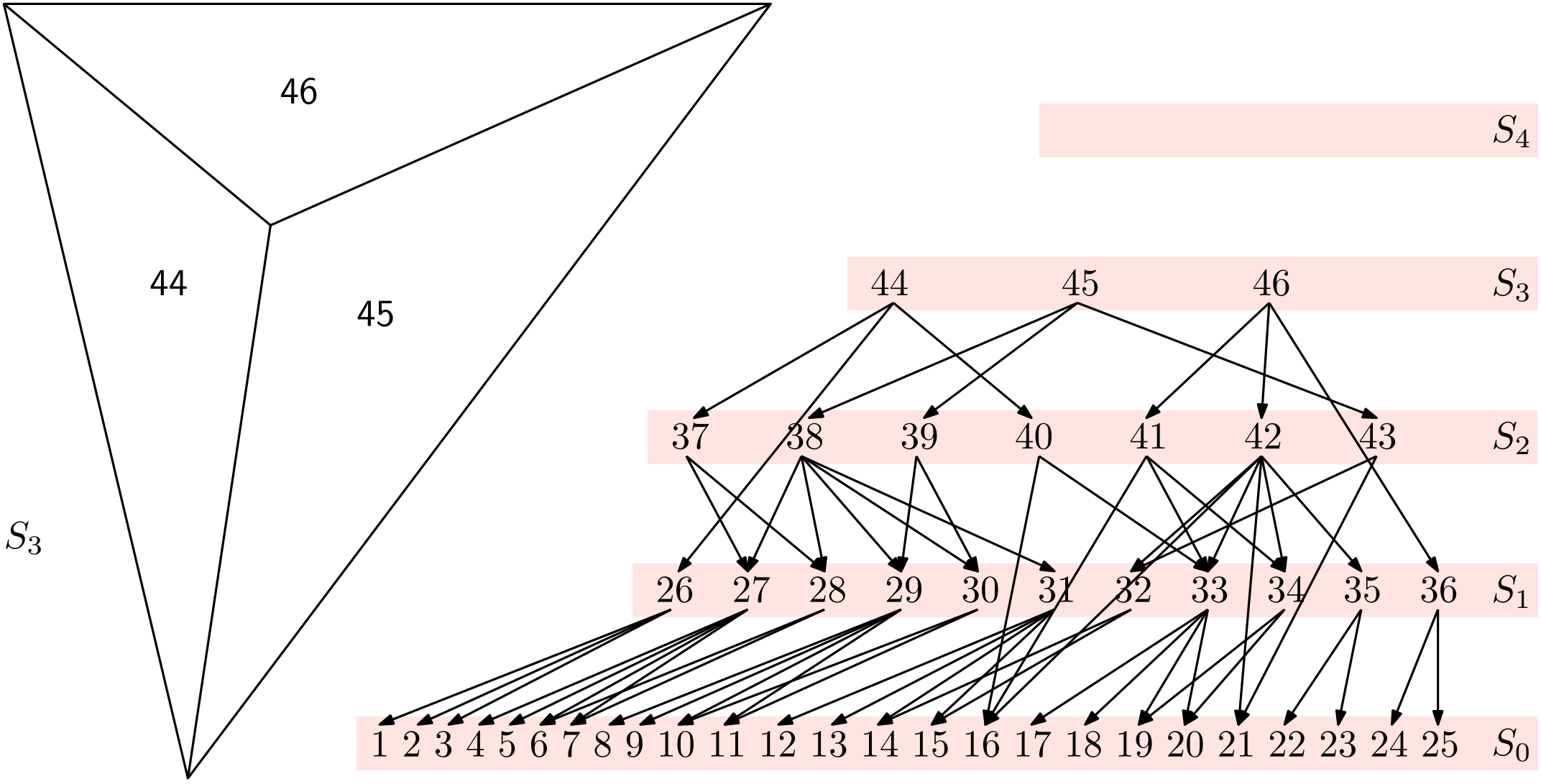
POINT LOCATION: Triangulation refinement

Preprocessing



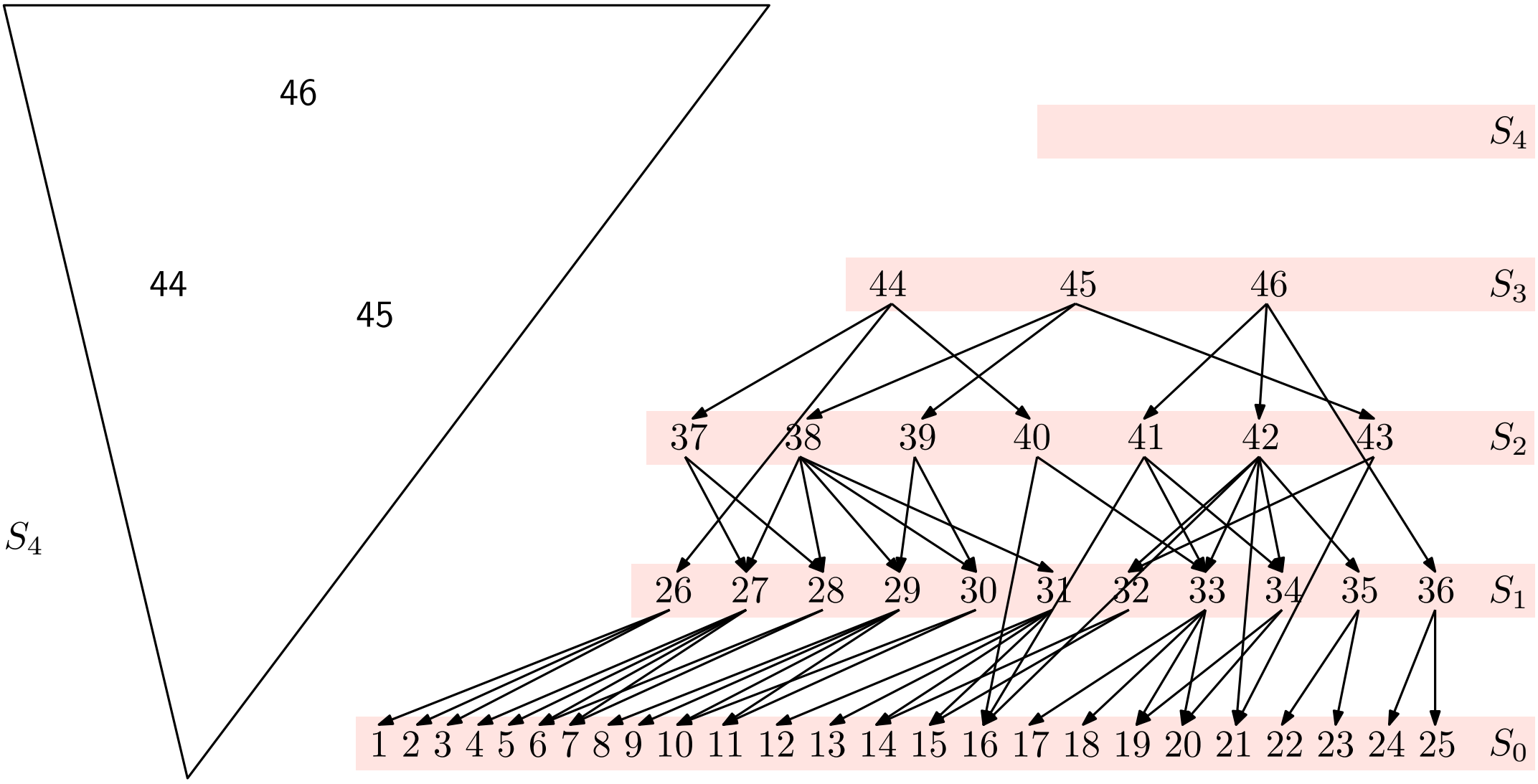
POINT LOCATION: Triangulation refinement

Preprocessing



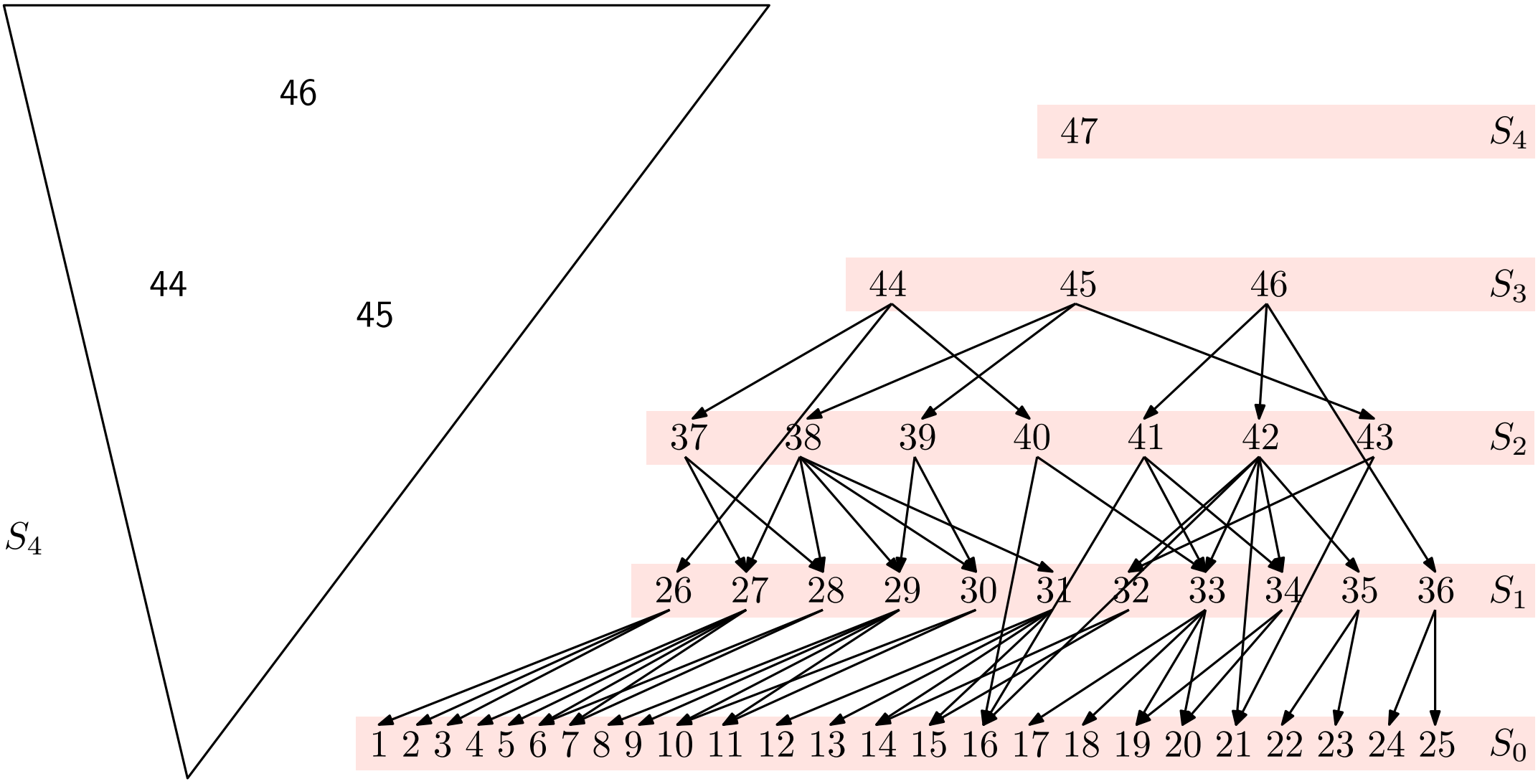
POINT LOCATION: Triangulation refinement

Preprocessing



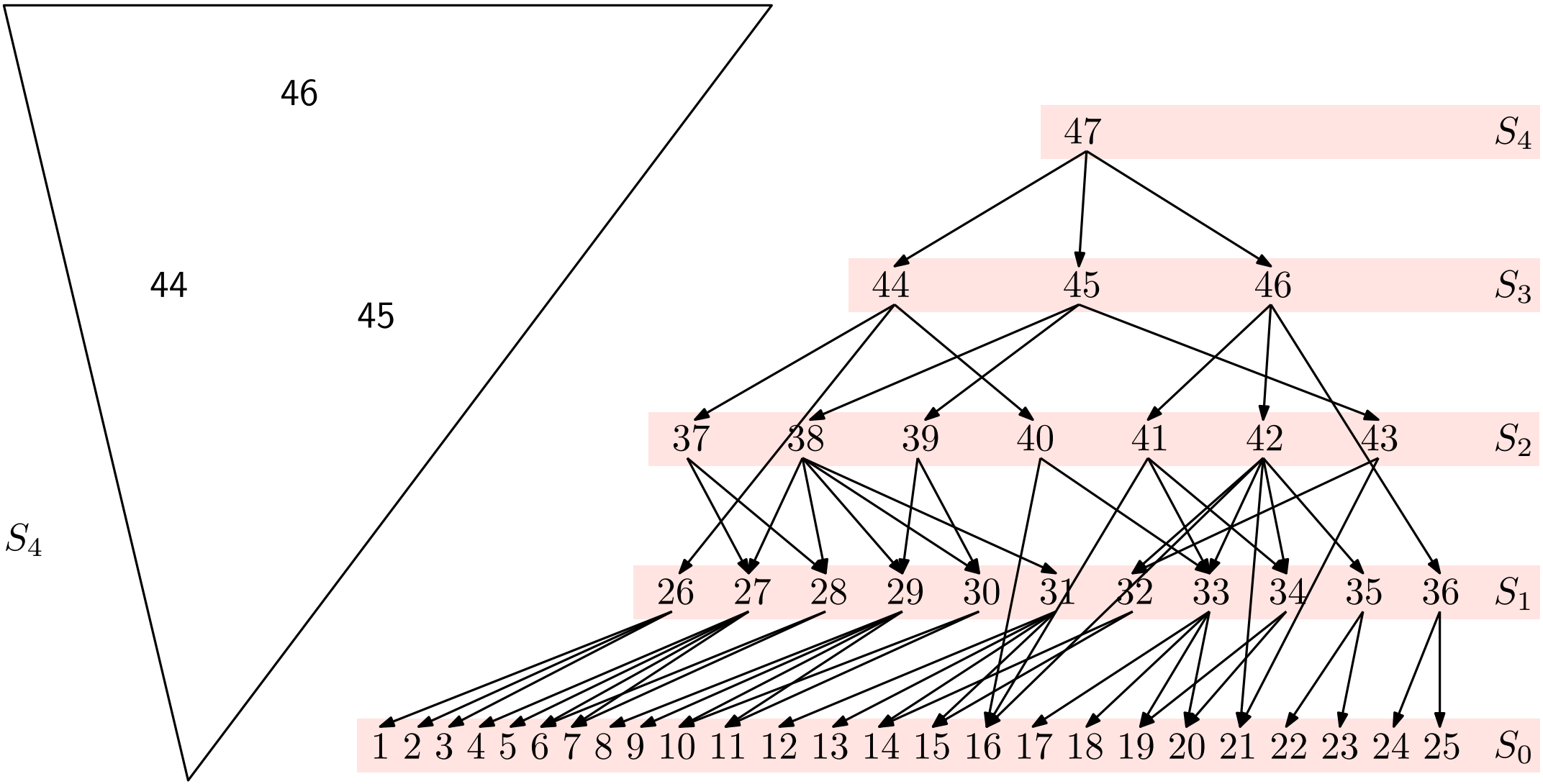
POINT LOCATION: Triangulation refinement

Preprocessing



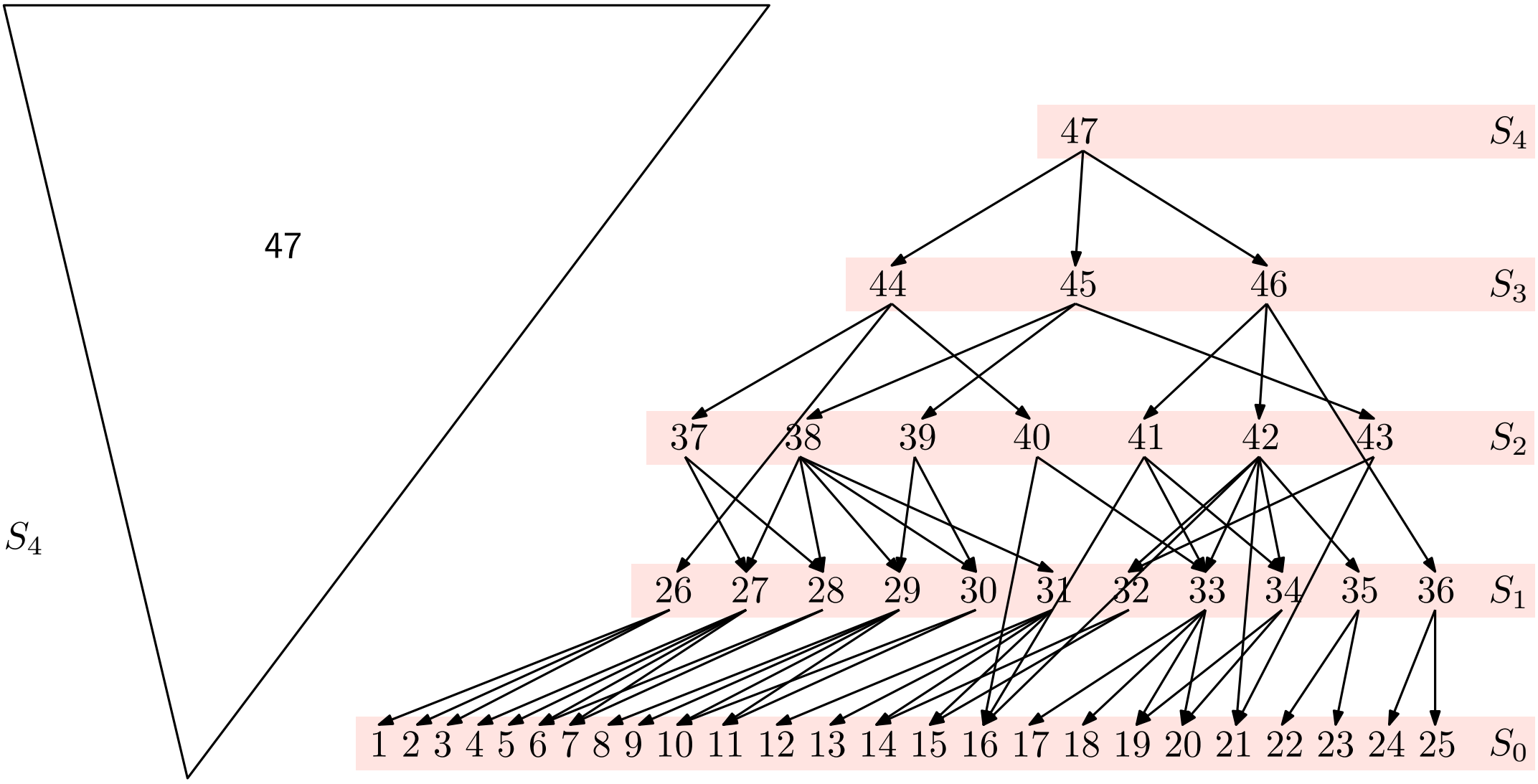
POINT LOCATION: Triangulation refinement

Preprocessing



POINT LOCATION: Triangulation refinement

Preprocessing



POINT LOCATION: Triangulation refinement

Preprocessing

How to make the height of the search tree to be $h = O(\log n)$?

POINT LOCATION: Triangulation refinement

Preprocessing

How to make the height of the search tree to be $h = O(\log n)$?

If, at each step, the number of vertices in the independent set is a constant fraction cn of the current number of vertices ($0 < c < 1$), then $h = O(\log n)$.

POINT LOCATION: Triangulation refinement

Preprocessing

How to make the height of the search tree to be $h = O(\log n)$?

If, at each step, the number of vertices in the independent set is a constant fraction cn of the current number of vertices ($0 < c < 1$), then $h = O(\log n)$.

Proof:

The final number of vertices is $3 = n(1-c)^h$, therefore $h = \frac{\log n - \log 3}{-\log(1-c)} = O(\log n)$.

POINT LOCATION: Triangulation refinement

Preprocessing

How to make the height of the search tree to be $h = O(\log n)$?

If, at each step, the number of vertices in the independent set is a constant fraction cn of the current number of vertices ($0 < c < 1$), then $h = O(\log n)$.

Proof:

The final number of vertices is $3 = n(1-c)^h$, therefore $h = \frac{\log n - \log 3}{-\log(1-c)} = O(\log n)$.

How to make the number of chosen independent vertices to always be a constant fraction?

POINT LOCATION: Triangulation refinement

Preprocessing

How to make the height of the search tree to be $h = O(\log n)$?

If, at each step, the number of vertices in the independent set is a constant fraction cn of the current number of vertices ($0 < c < 1$), then $h = O(\log n)$.

Proof:
The final number of vertices is $3 = n(1-c)^h$, therefore $h = \frac{\log n - \log 3}{-\log(1-c)} = O(\log n)$.

How to make the number of chosen independent vertices to always be a constant fraction?

Choosing in a greedy way all possible vertices of degree ≤ 8 (as long as they stay independent), allows eliminating at each step at least $n/18$ vertices.

POINT LOCATION: Triangulation refinement

Preprocessing

How to make the height of the search tree to be $h = O(\log n)$?

If, at each step, the number of vertices in the independent set is a constant fraction cn of the current number of vertices ($0 < c < 1$), then $h = O(\log n)$.

Proof:

The final number of vertices is $3 = n(1-c)^h$, therefore $h = \frac{\log n - \log 3}{-\log(1-c)} = O(\log n)$.

How to make the number of chosen independent vertices to always be a constant fraction?

Choosing in a greedy way all possible vertices of degree ≤ 8 (as long as they stay independent), allows eliminating at each step at least $n/18$ vertices.

Order does not matter: choose one, label all its neighbors as being not independent, choose a second one, ...

POINT LOCATION: Triangulation refinement

Preprocessing

How to make the height of the search tree to be $h = O(\log n)$?

If, at each step, the number of vertices in the independent set is a constant fraction cn of the current number of vertices ($0 < c < 1$), then $h = O(\log n)$.

Proof:
The final number of vertices is $3 = n(1-c)^h$, therefore $h = \frac{\log n - \log 3}{-\log(1-c)} = O(\log n)$.

How to make the number of chosen independent vertices to always be a constant fraction?

Choosing in a greedy way all possible vertices of degree ≤ 8 (as long as they stay independent), allows eliminating at each step at least $n/18$ vertices.

Proof:

1. There exist at least $n/2$ vertices of order ≤ 8. Otherwise, if more than half of the vertices had degree ≥ 9, then $\sum \text{degrees} \geq 9\frac{n}{2} + 3(\frac{n}{2} - h) + 2h = 6n - 3$, and this cannot happen since $\sum \text{degrees} = 2e = 2(3n - h - 3) = 6n - 12$.

2. Each time a vertex of degree ≤ 8 is chosen, all its neighbors (at most 8 vertices) must be discarded. In the worst case, all of them will also be of degree ≤ 8, and the process will be choosing $1/9$ of the vertices of degree ≤ 8. As there are at least $n/2$ such vertices, the minimum number of independent vertices is $n/18$.

POINT LOCATION: Triangulation refinement

Complexity

Space: the space used to store the hierarchy of triangulations is $O(n)$:

- The total number of triangles is $O(n) + O((1-c)n) + O((1-c)^2 n) + \cdots = O(n)$.
- The number of pointers leaving a triangle is less or equal to the number of triangles that appear when retriangulating a hole of constant size ≤ 8.

Preprocessing: computing the refinement of triangulations is done in $O(n)$ time, since at each step the algorithm:

- Finds the independent vertices (exploring all current vertices).
- Retriangulates a linear amount of holes, each of size $O(1)$.

Hence, the overall task is done in time $O(n) + O((1-c)n) + O((1-c)^2 n) + \cdots = O(n)$.

Location: locating a point is done in $O(\log n)$ time, since the height of the search tree is $h = O(\log n)$.

POINT LOCATION: Triangulation refinement

Extension to arbitrary graphs

When the planar decomposition is not a triangulation but an arbitrary graph:

1. Enclose the graph in a triangle.
2. Triangulate all the resulting non triangular regions.

POINT LOCATION: Triangulation refinement

Extension to arbitrary graphs

When the planar decomposition is not a triangulation but an arbitrary graph:

1. Enclose the graph in a triangle.
2. Triangulate all the resulting non triangular regions.

This adds $O(n \log n)$ running time to the preprocessing step (except if the polygons are triangulated with Chazelle's linear algorithm).

POINT LOCATION: Triangulation refinement

Extension to Voronoi diagrams

The Voronoi diagram is not a proper planar decomposition, since some of its edges are half-lines.

POINT LOCATION: Triangulation refinement

Extension to Voronoi diagrams

The Voronoi diagram is not a proper planar decomposition, since some of its edges are half-lines.

Preprocessing

1. Consider a triangle enclosing all Voronoi vertices.
2. Clip each half-line with the boundary of the enclosing triangle.
3. Triangulate all the regions.

This can be done in $O(n)$ time.

Search

When a point lies in the exterior of the enclosing triangle, the algorithm must detect its relative position with respect to the half-lines. This can be done in $O(\log n)$ time (with the appropriate data structure), since the half-lines are sorted.

POINT LOCATION: Trapezoidal map

Another well-known method you may encounter:

Trapezoidal map

POINT LOCATION: Trapezoidal map

This is a randomized method, very convenient in practice.

It consists in a variation of the slab method:

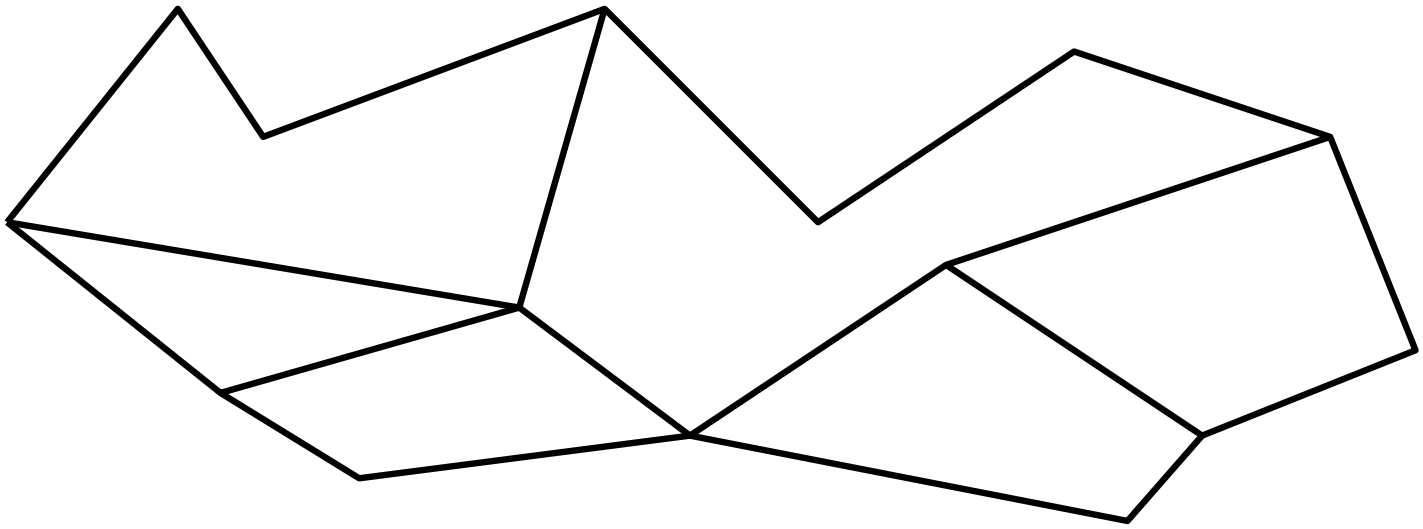
1. Compute a rectangle enclosing the 1-skeleton of the graph G.
2. From each vertex of G, shoot two vertical rays, upwards and downwards, until they reach an edge (of the graph or of the enclosing rectangle).
3. Can be built incrementally, inserting segments one by one, together with a tree-like point location data structure

POINT LOCATION: Trapezoidal map

This is a randomized method, very convenient in practice.

It consists in a variation of the slab method:

1. Compute a rectangle enclosing the 1-skeleton of the graph G.
2. From each vertex of G, shoot two vertical rays, upwards and downwards, until they reach an edge (of the graph or of the enclosing rectangle).
3. Can be built incrementally, inserting segments one by one, together with a tree-like point location data structure

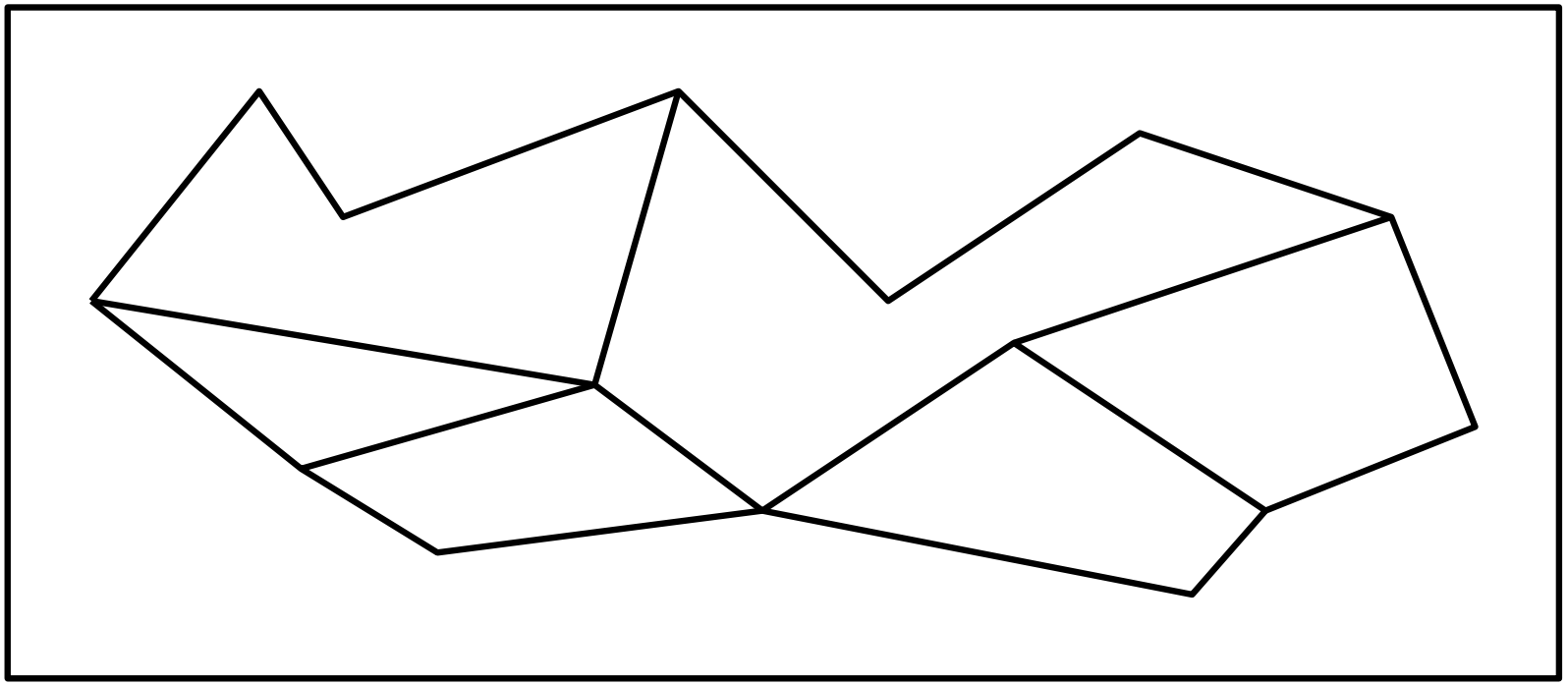


POINT LOCATION: Trapezoidal map

This is a randomized method, very convenient in practice.

It consists in a variation of the slab method:

1. Compute a rectangle enclosing the 1-skeleton of the graph G.
2. From each vertex of G, shoot two vertical rays, upwards and downwards, until they reach an edge (of the graph or of the enclosing rectangle).
3. Can be built incrementally, inserting segments one by one, together with a tree-like point location data structure

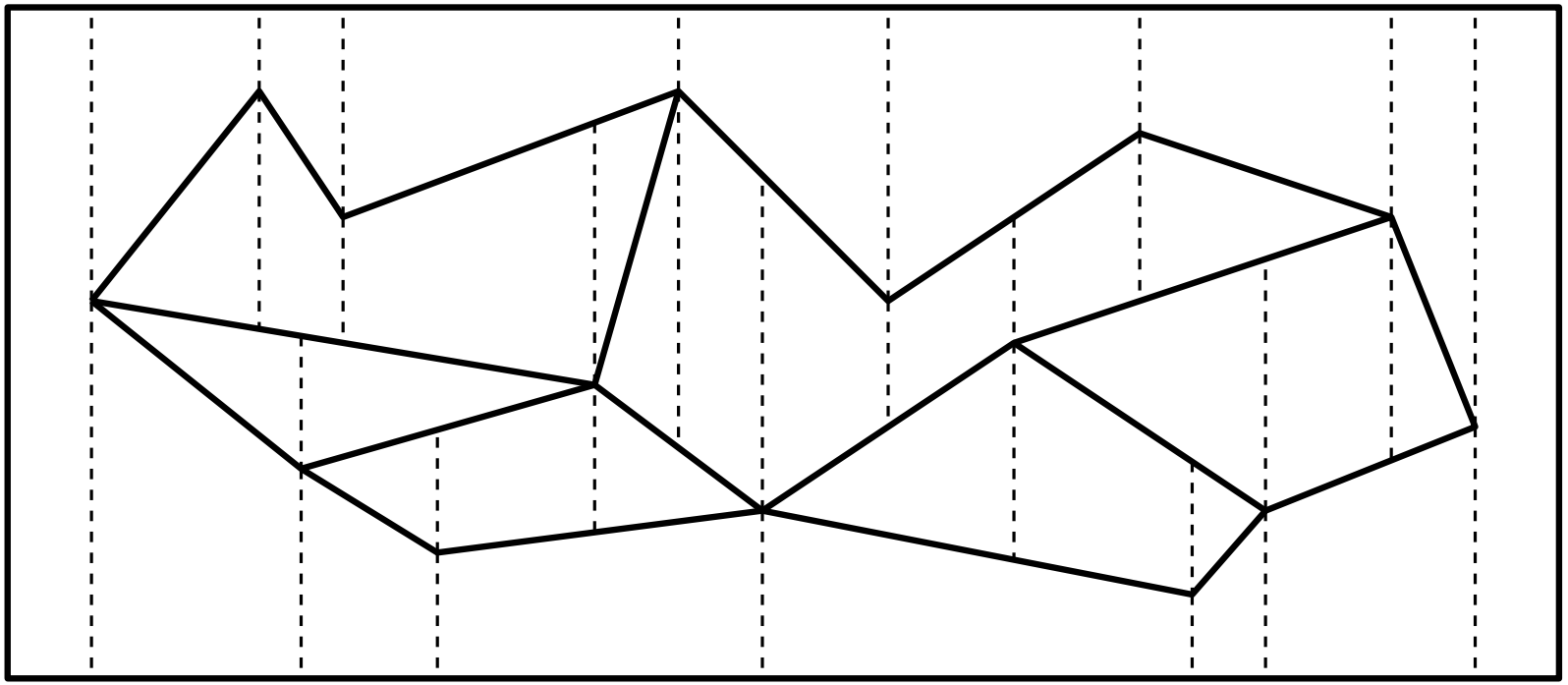


POINT LOCATION: Trapezoidal map

This is a randomized method, very convenient in practice.

It consists in a variation of the slab method:

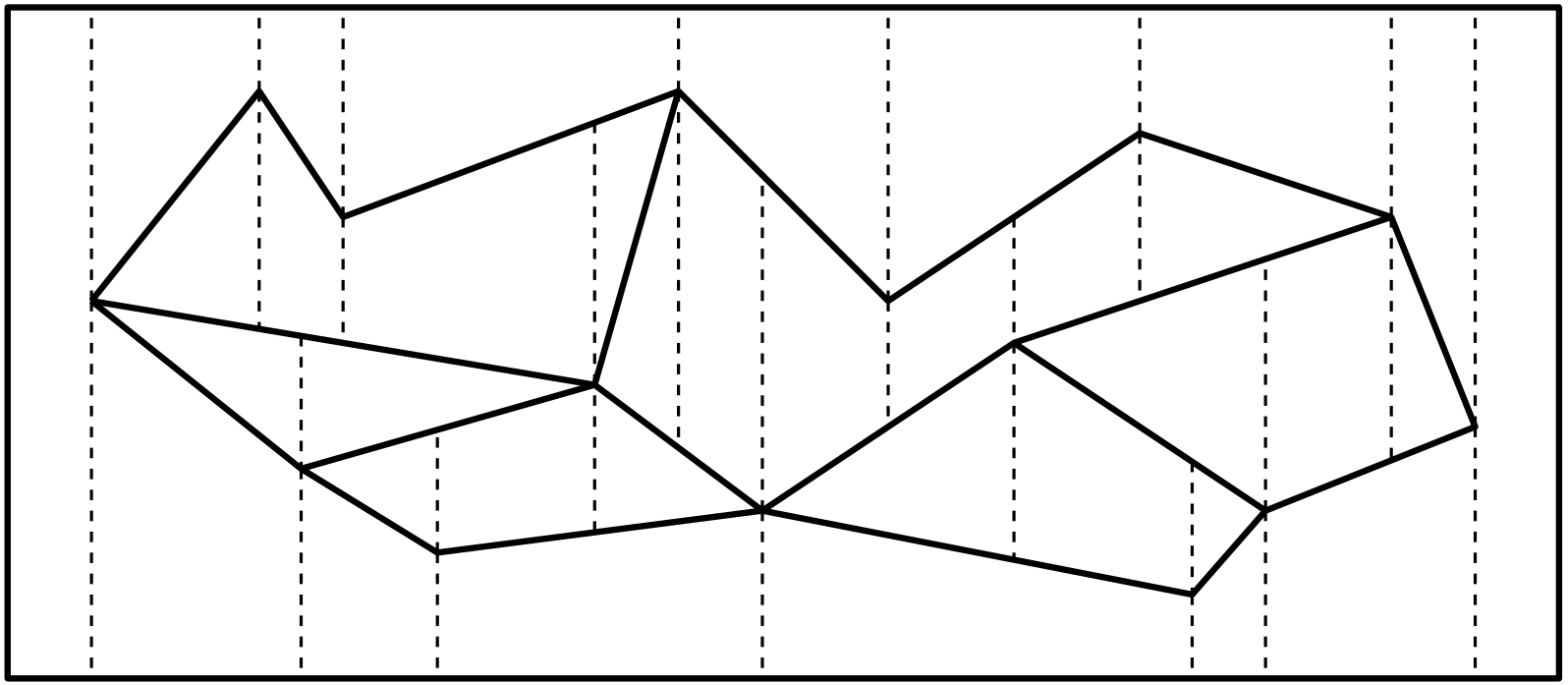
1. Compute a rectangle enclosing the 1-skeleton of the graph G.
2. From each vertex of G, shoot two vertical rays, upwards and downwards, until they reach an edge (of the graph or of the enclosing rectangle).
3. Can be built incrementally, inserting segments one by one, together with a tree-like point location data structure



POINT LOCATION: Trapezoidal map

Complexity

The total number of resulting trapezoids is linear: for each vertex of G, at most two new edges and two new vertices are added.



POINT LOCATION: Trapezoidal map

Complexity

The total number of resulting trapezoids is linear: for each vertex of G, at most two new edges and two new vertices are added.

If the line segments of the initial decomposition G are processed in random order:

Space: The appropriate search structure has expected size $O(n)$.

Preprocessing: The expected running time for computing the trapezoids and building the search structure is $O(n \log n)$.

Location: Given a point q of the plane, the region where it is located is found in $O(\log n)$ expected running time.

POINT LOCATION: To learn more

TO LEARN MORE

- F. P. Preparata, M. I. Shamos: *Computational Geometry - An Introduction*, Springer.
- M. de Berg, O. Cheong, M. van Kreveld, M. Overmars: *Computational Geometry: Algorithms and Applications*, Springer.
- O. Devillers, S. Pion, M. Teillaud: Walking in a Triangulation, *International Journal of Foundations of Computer Science*, Vol. 13, pp. 181-199, 2002.