Computational Geometry – Problem Session

Convex Hull & Line Segment Intersection

LEHRSTUHL FÜR ALGORITHMIK · INSTITUT FÜR THEORETISCHE INFORMATIK · FAKULTÄT FÜR INFORMATIK

Guido Brückner

04.05.2018

Modus Operandi

To register for the oral exam we expect you to present an original solution for at least one problem in the exercise session.

- this is about working *together*
- don't worry if your idea doesn't work!

Outline

Convex Hull

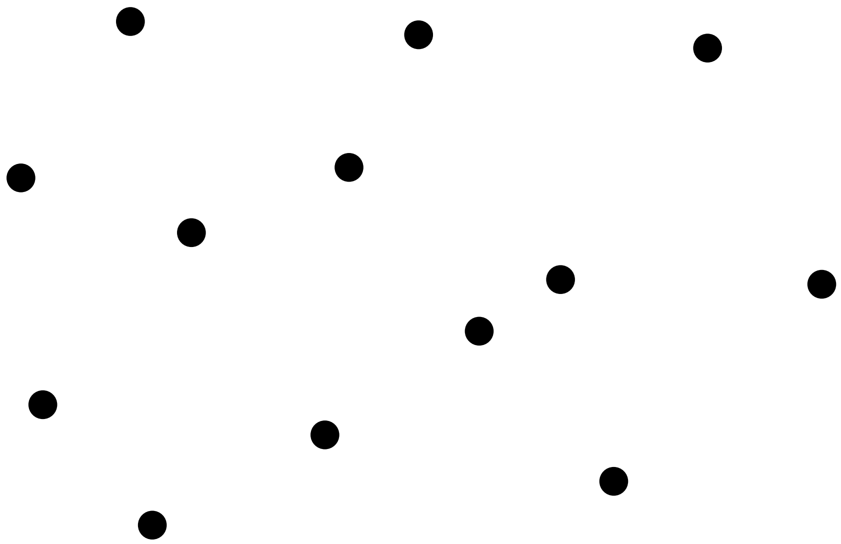
Line Segment Intersection

Definition of Convex Hull

Def: A region $S \subseteq \mathbb{R}^2$ is called **convex**, when for two points $p, q \in S$ then line $\overline{pq} \in S$.
The **convex hull** $CH(S)$ of S is the smallest convex region containing S.

Definition of Convex Hull

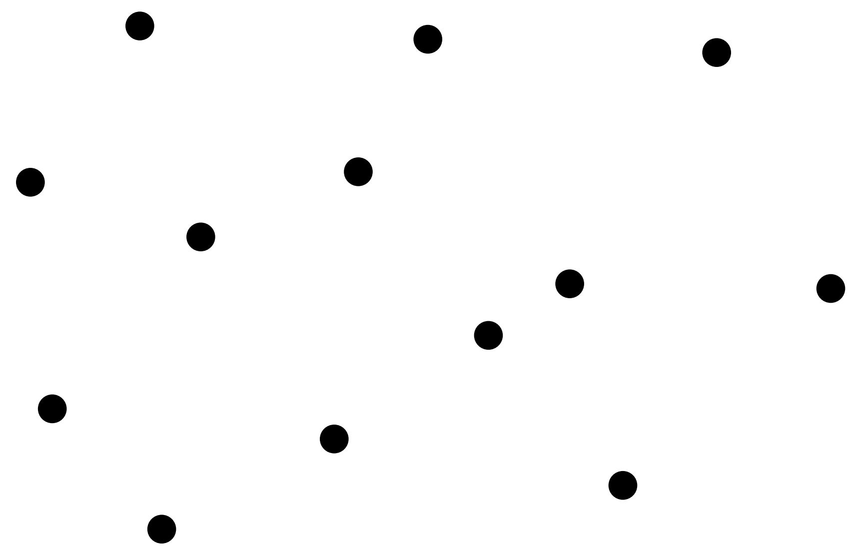
Def: A region $S \subseteq \mathbb{R}^2$ is called **convex**, when for two points $p, q \in S$ then line $\overline{pq} \in S$.
The **convex hull** $CH(S)$ of S is the smallest convex region containing S.



Definition of Convex Hull

Def: A region $S \subseteq \mathbb{R}^2$ is called **convex**, when for two points $p, q \in S$ then line $\overline{pq} \in S$.
The **convex hull** $CH(S)$ of S is the smallest convex region containing S.

In physics:

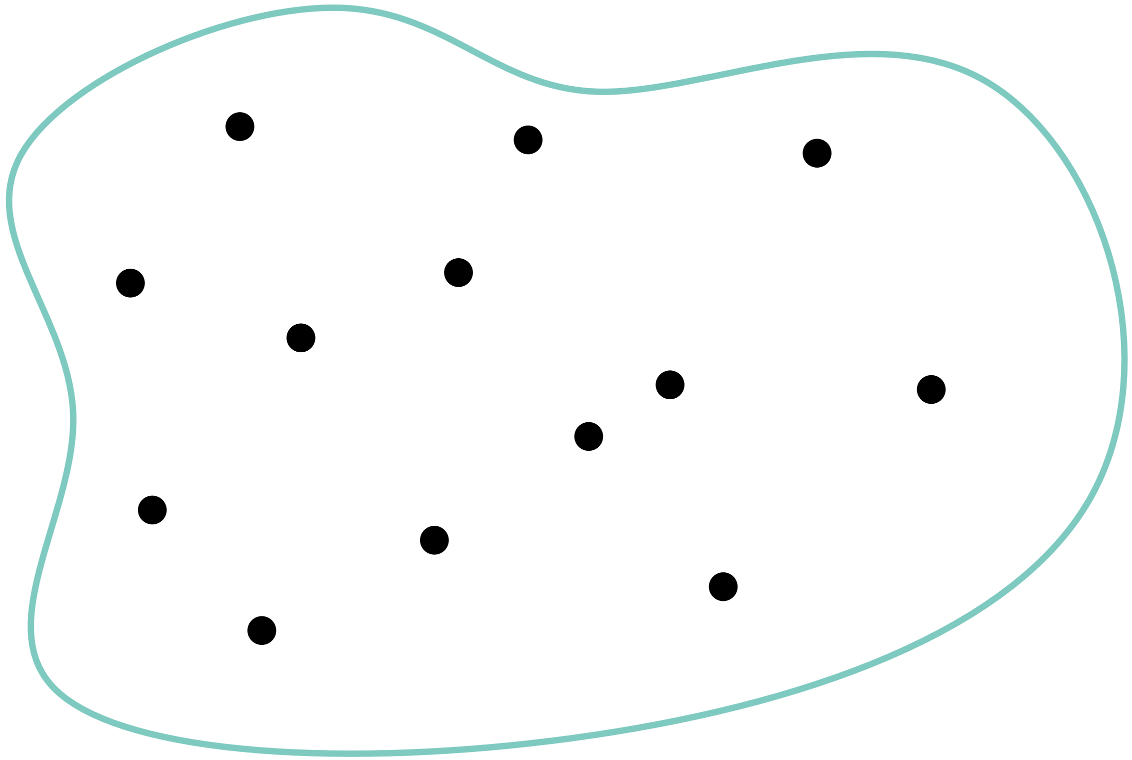


Definition of Convex Hull

Def: A region $S \subseteq \mathbb{R}^2$ is called **convex**, when for two points $p, q \in S$ then line $\overline{pq} \in S$.
The **convex hull** $CH(S)$ of S is the smallest convex region containing S.

In physics:

- put a large rubber band around all points

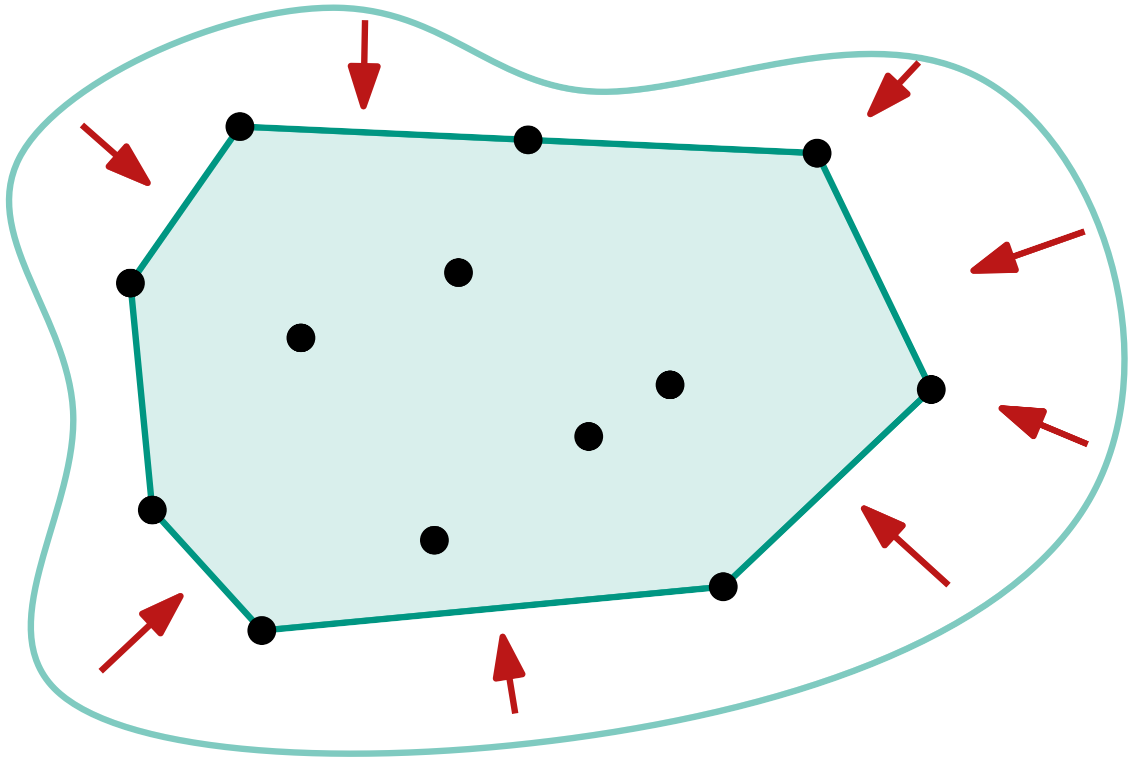


Definition of Convex Hull

Def: A region $S \subseteq \mathbb{R}^2$ is called **convex**, when for two points $p, q \in S$ then line $\overline{pq} \in S$.
The **convex hull** $CH(S)$ of S is the smallest convex region containing S.

In physics:

- put a large rubber band around all points
- and let it go!

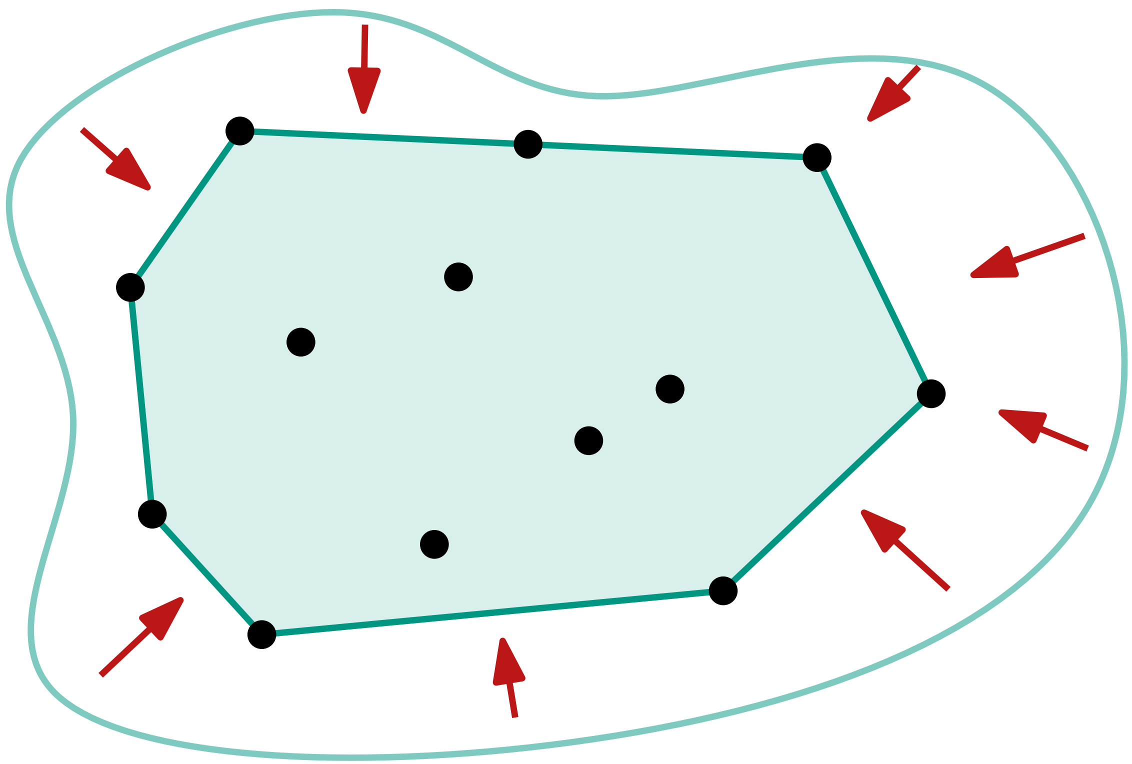


Definition of Convex Hull

Def: A region $S \subseteq \mathbb{R}^2$ is called **convex**, when for two points $p, q \in S$ then line $\overline{pq} \in S$.
The **convex hull** $CH(S)$ of S is the smallest convex region containing S.

In physics:

- put a large rubber band around all points
- and let it go!
- unfortunately, does not help algorithmically

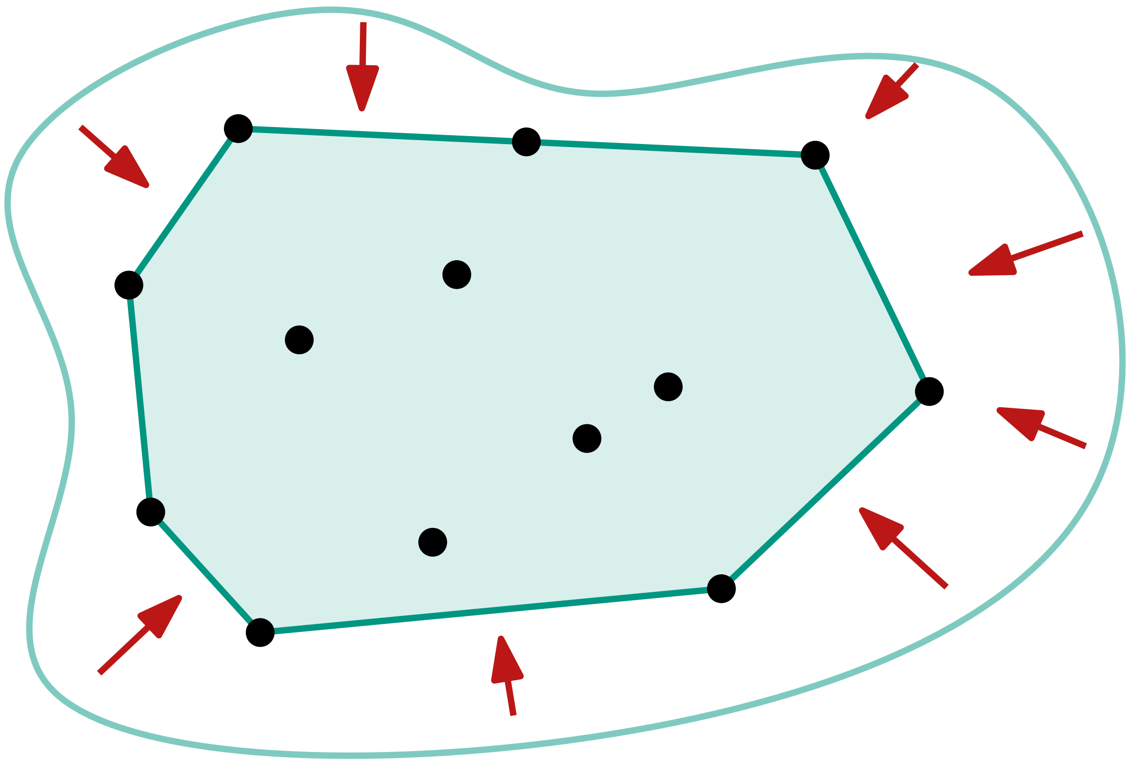


Definition of Convex Hull

Def: A region $S \subseteq \mathbb{R}^2$ is called **convex**, when for two points $p, q \in S$ then line $\overline{pq} \in S$.
The **convex hull** $CH(S)$ of S is the smallest convex region containing S.

In physics:

- put a large rubber band around all points
- and let it go!
- unfortunately, does not help algorithmically



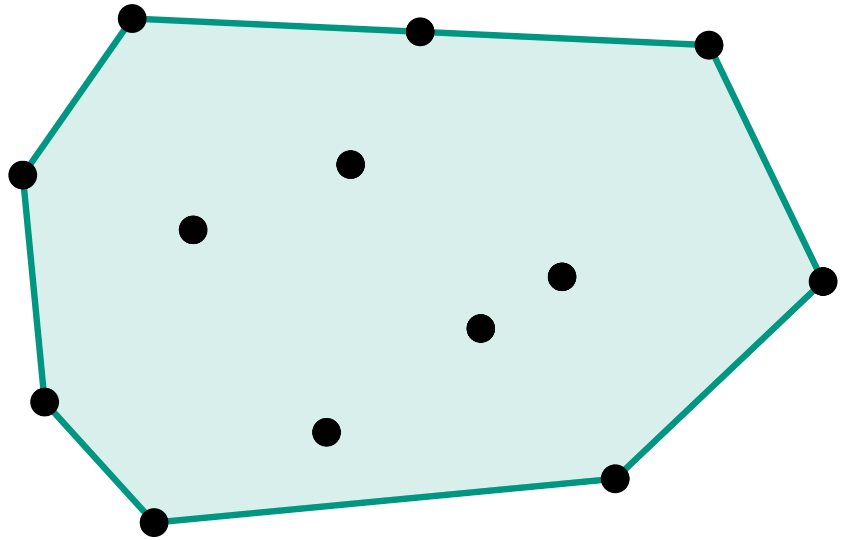
In mathematics:

- define $CH(S) = \bigcap_{C \supseteq S:\ C \text{ convex}} C$
- does not help :-(

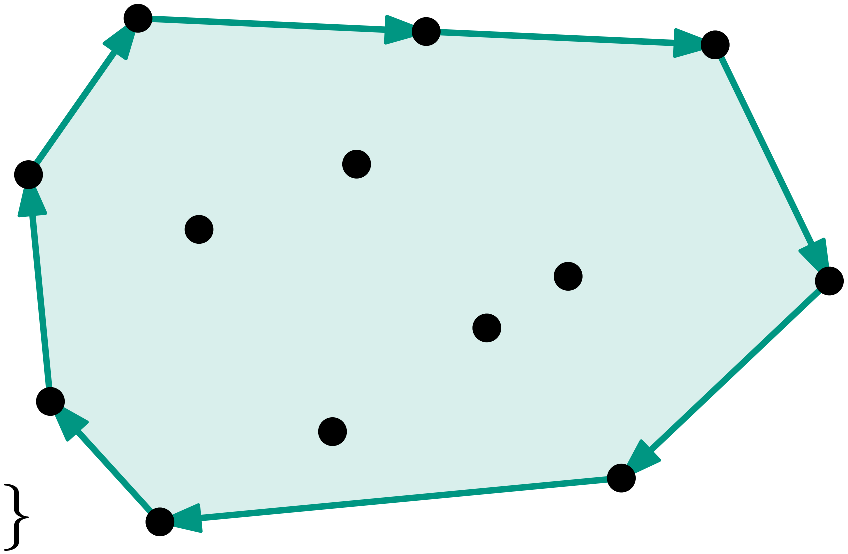
Algorithmic Approach

Lemma:

For a set of points $P \subseteq \mathbb{R}^2$, $CH(P)$ is a convex polygon that contains P and whose vertices are in P.



Algorithmic Approach



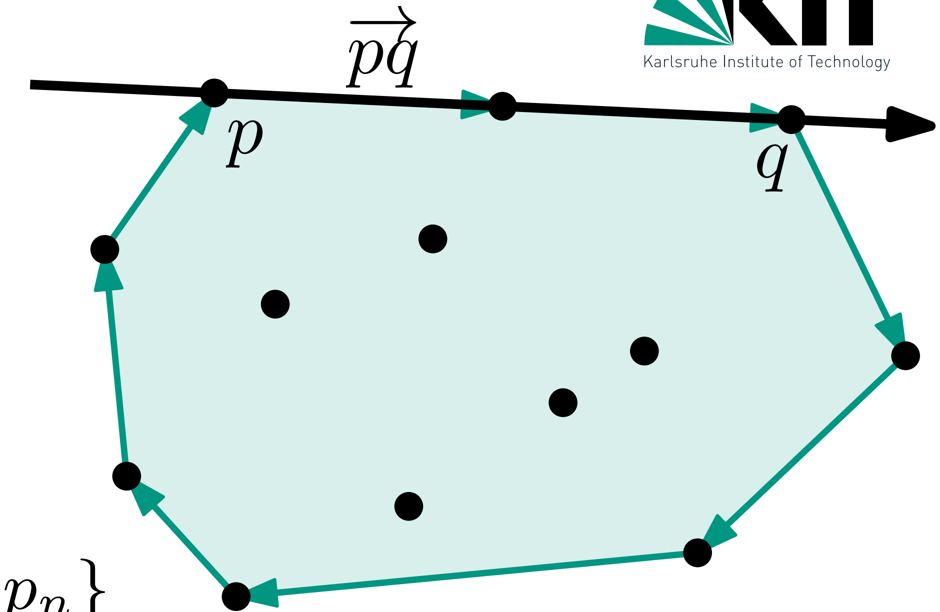
Lemma:

For a set of points $P \subseteq \mathbb{R}^2$, $CH(P)$ is a convex polygon that contains P and whose vertices are in P.

Input: A set of points $P = \{p_1, \ldots, p_n\}$

Output: List of nodes of $CH(P)$ in clockwise order

Algorithmic Approach



Lemma:

For a set of points $P \subseteq \mathbb{R}^2$, $CH(P)$ is a convex polygon that contains P and whose vertices are in P.

Input: A set of points $P = \{p_1, \ldots, p_n\}$

Output: List of nodes of $CH(P)$ in clockwise order

Observation:

(p, q) is an edge of $CH(P) \Leftrightarrow$ each point $r \in P \setminus \{p, q\}$

- strictly right of the oriented line $\overrightarrow{pq}$ or
- on the line segment $\overline{pq}$

Running Time Analysis

FirstConvexHull(P)

$E \leftarrow \emptyset$

foreach $(p, q) \in P \times P$ with $p \neq q$ **do** $(n^2 - n)\cdot$

  valid ← *true*

  foreach $r \in P$ **do**

    if not (r is strictly right of $\overrightarrow{pq}$ **or** $r \in \overline{pq}$) **then**

      valid ← *false*

  if *valid* **then**

    $E \leftarrow E \cup \{(p, q)\}$

$\Theta(1)$ $\Theta(n)$ $\Theta(n^3)$

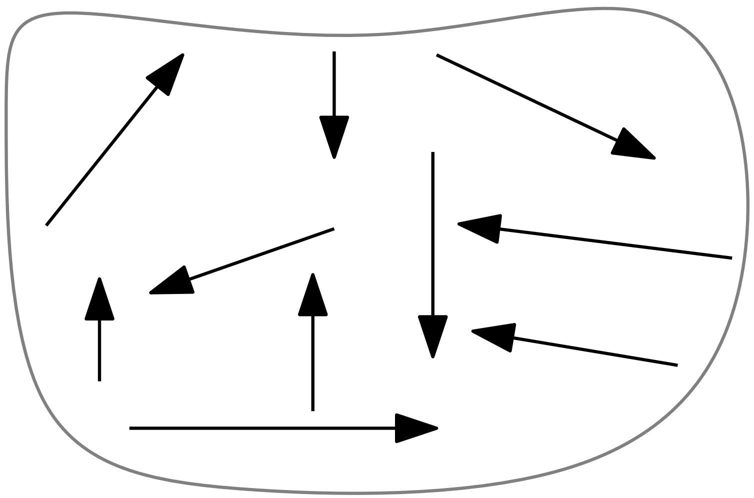
construct the sorted node list L from $CH(P)$ out of E

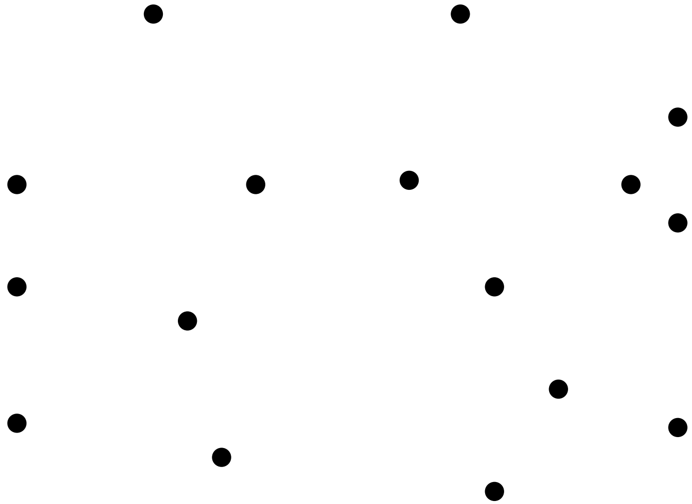
return L

Question: How do we implement this?

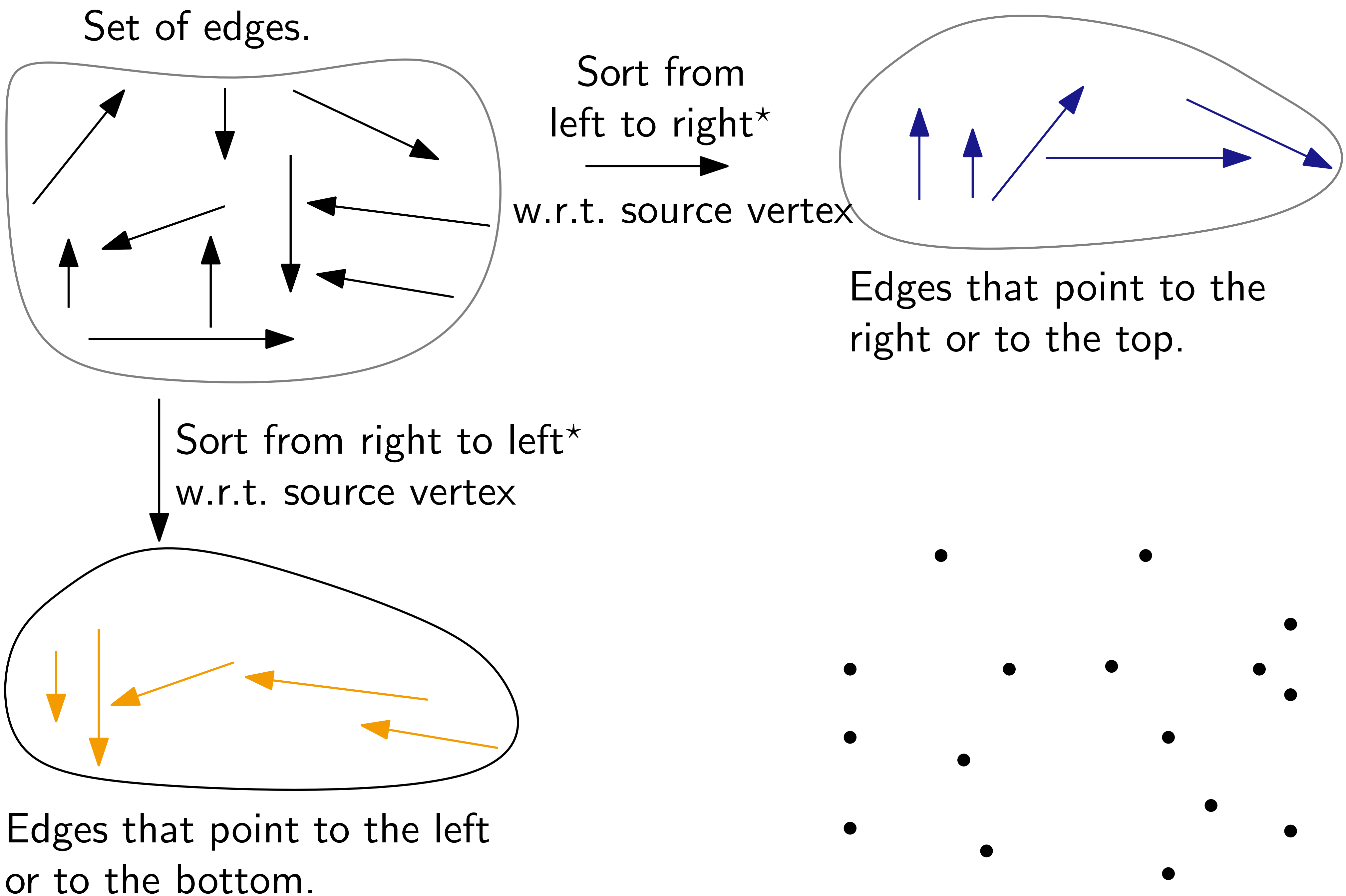
Solution

Set of edges.



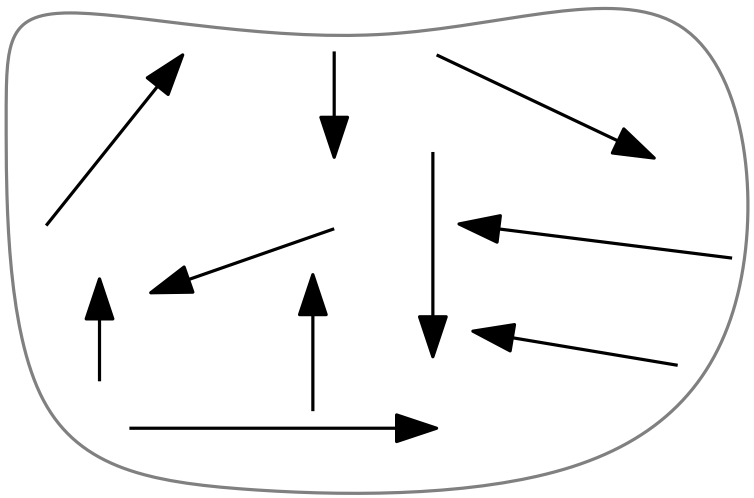


Solution

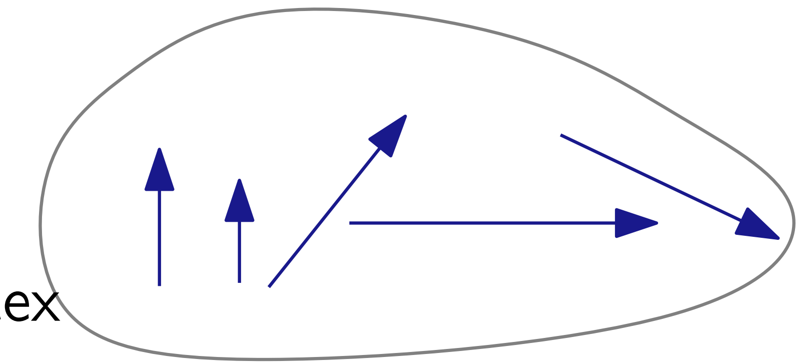

Set of edges.
Sort from
left to right*
w.r.t. source vertex
Edges that point to the
right or to the top.
Sort from right to left*
w.r.t. source vertex
Edges that point to the left
or to the bottom.

Solution

Set of edges.



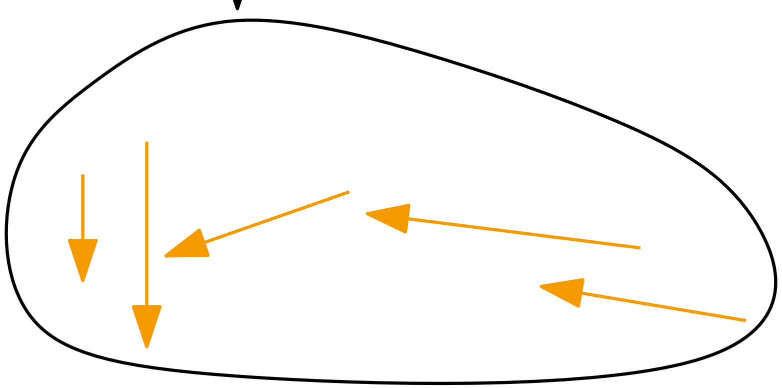
Sort from left to right* w.r.t. source vertex



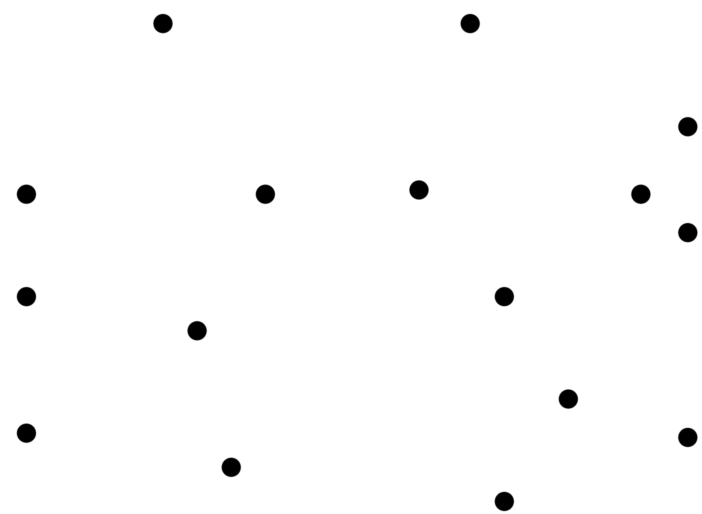
Edges that point to the right or to the top.

Sort from right to left* w.r.t. source vertex

*if not unique:
from bottom to top.
from top to bottom.

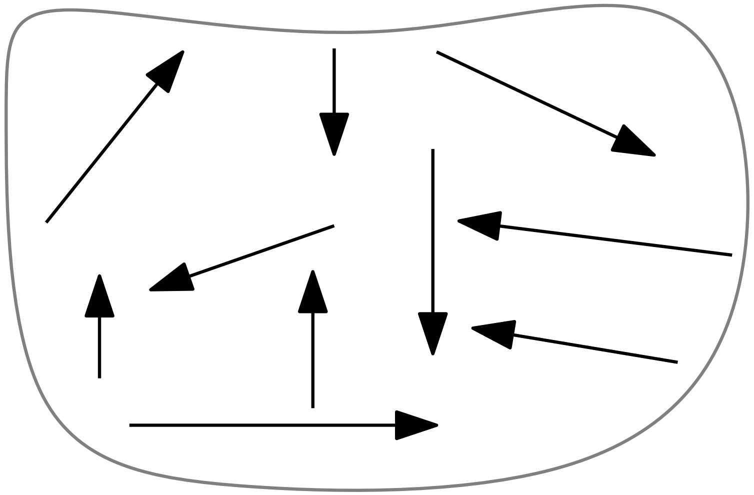


Edges that point to the left or to the bottom.

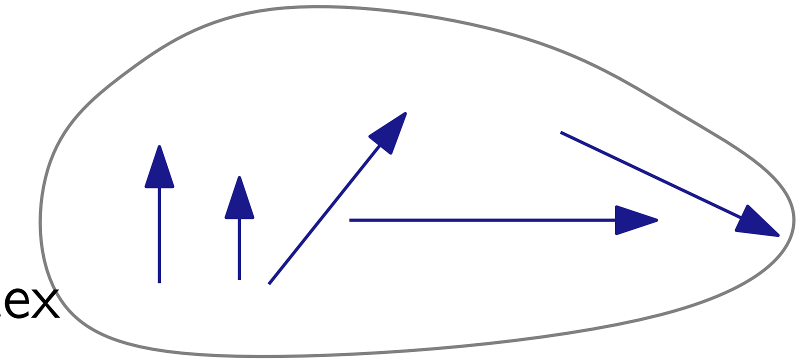


Solution

Set of edges.



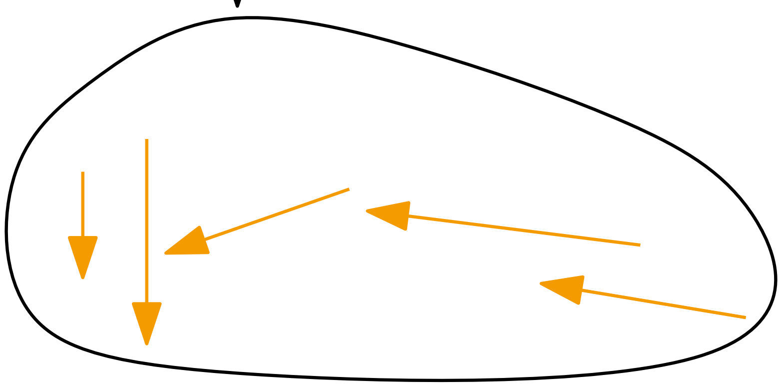
Sort from left to right* w.r.t. source vertex



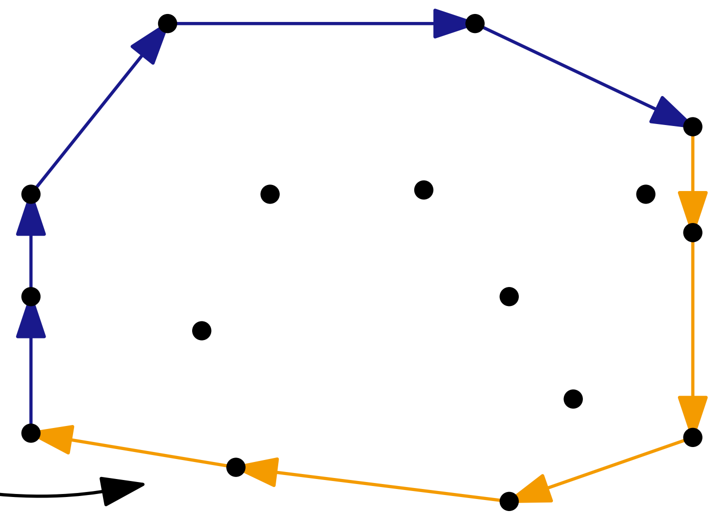
Edges that point to the right or to the top.

*if not unique:
from bottom to top.
from top to bottom.

Sort from right to left* w.r.t. source vertex



Edges that point to the left or to the bottom.



Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

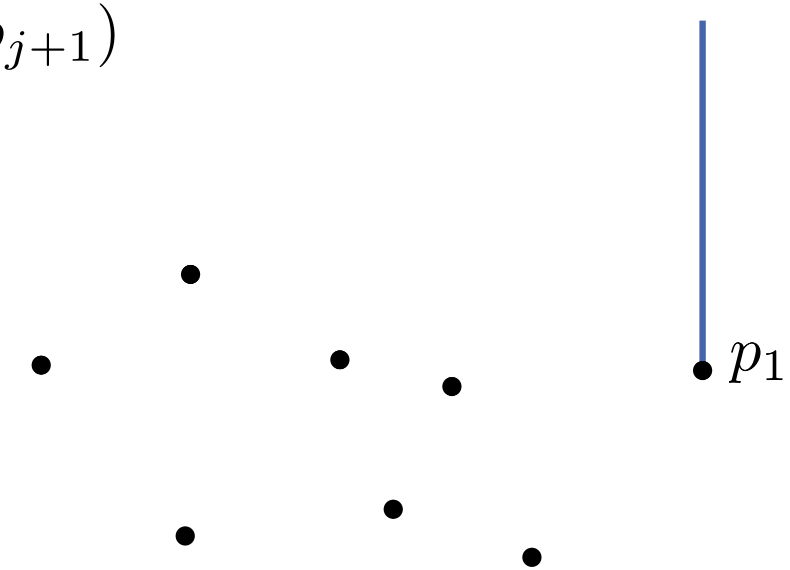
$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
return $(p_1, \ldots, p_{j+1})$

Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
    $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
    if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
return $(p_1, \ldots, p_{j+1})$

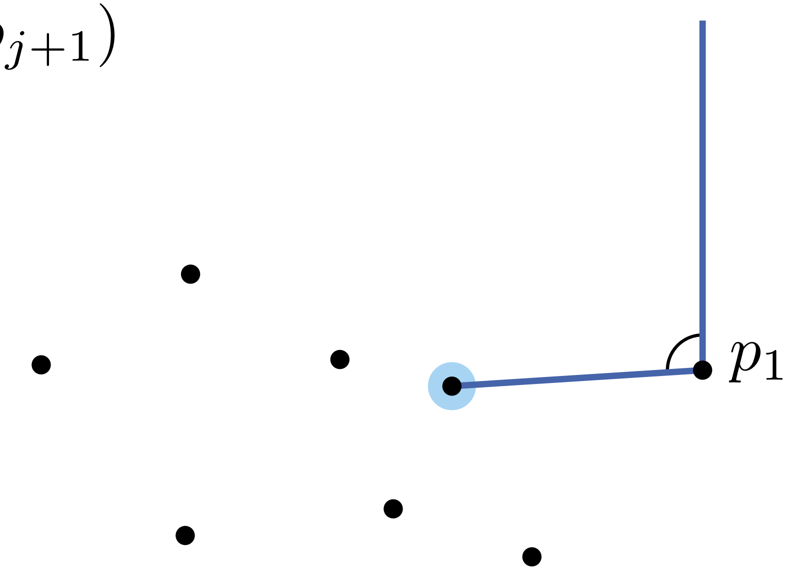


Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
return $(p_1, \ldots, p_{j+1})$

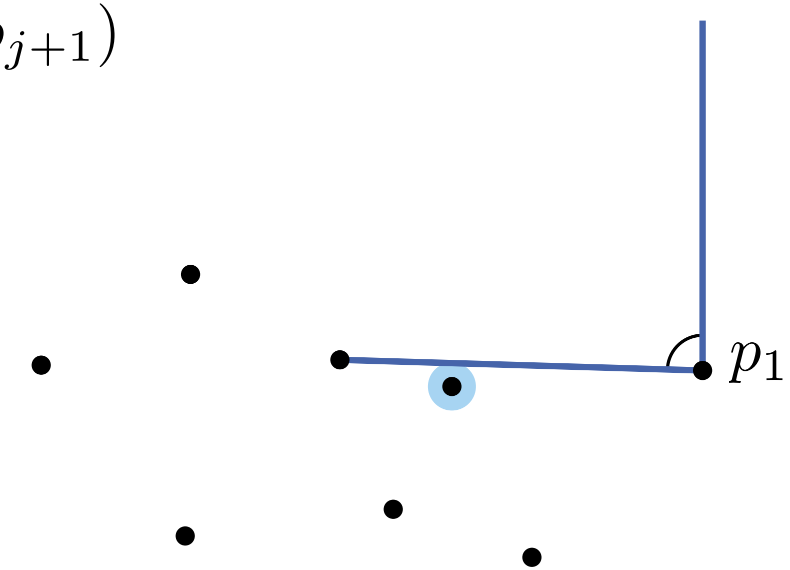


Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
return $(p_1, \ldots, p_{j+1})$

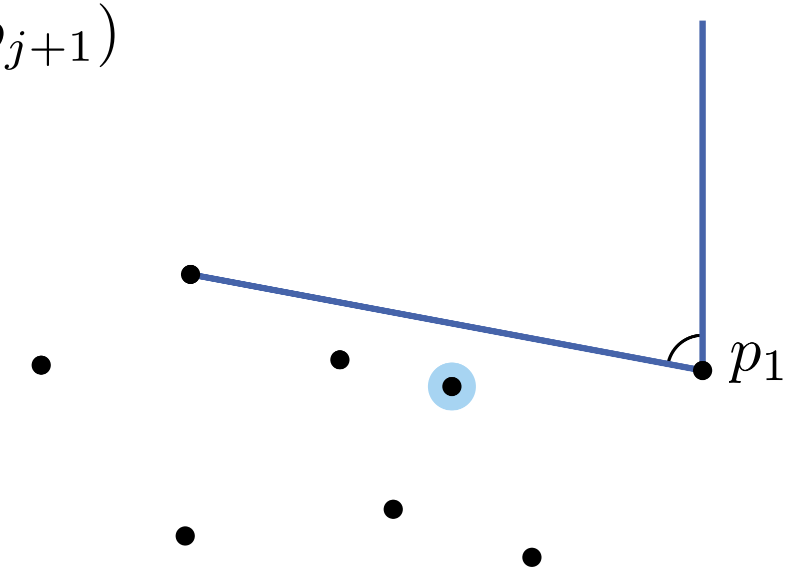


Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
return $(p_1, \ldots, p_{j+1})$



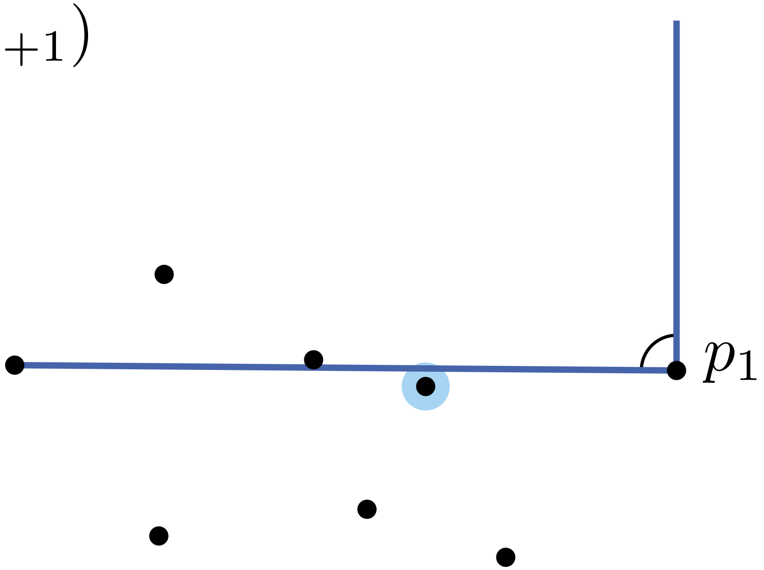
Alternative: Gift Wrapping

Karlsruhe Institute of Technology

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
return $(p_1, \ldots, p_{j+1})$

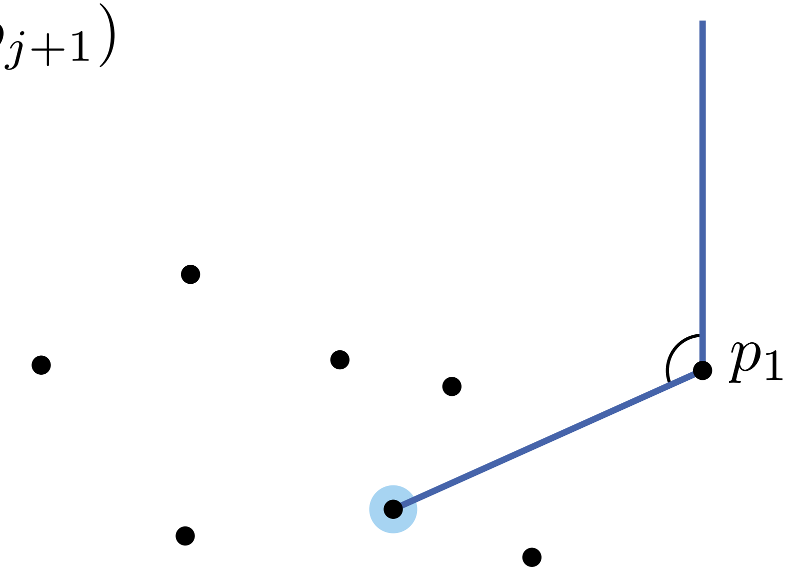


Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
return $(p_1, \ldots, p_{j+1})$

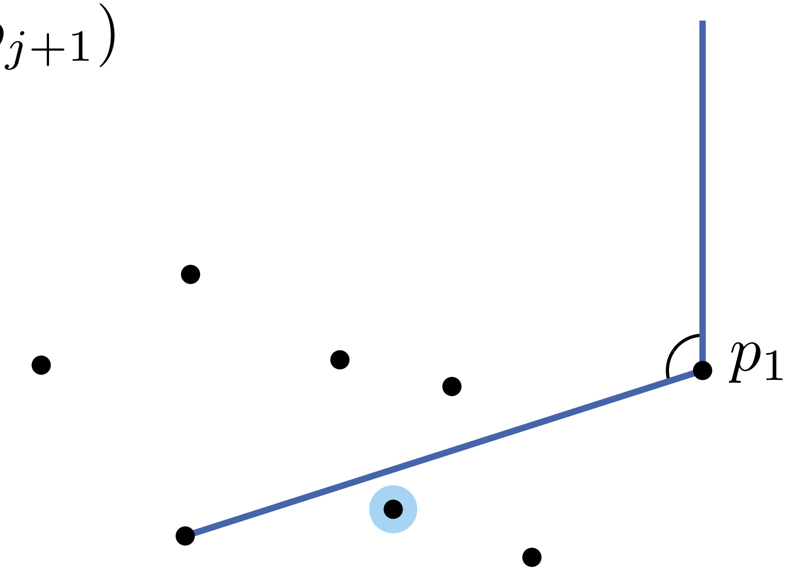


Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
return $(p_1, \ldots, p_{j+1})$

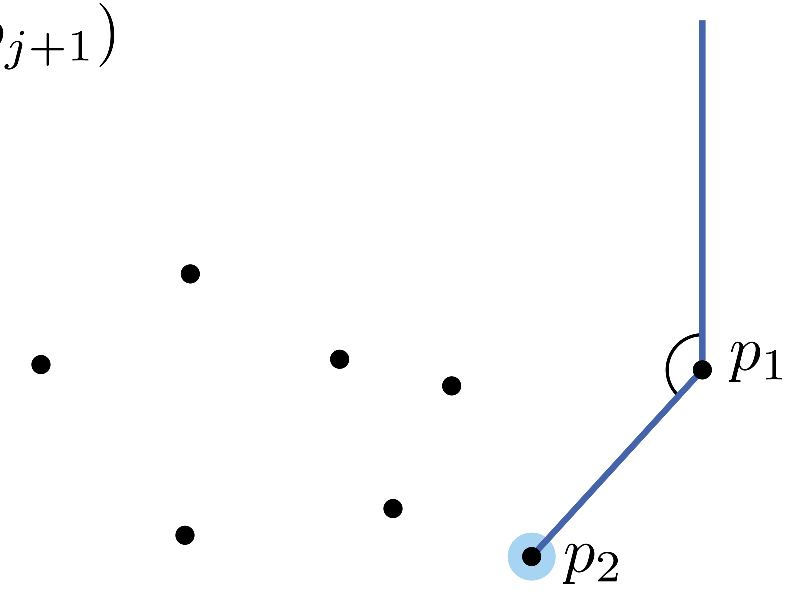


Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j+1$
return $(p_1, \ldots, p_{j+1})$

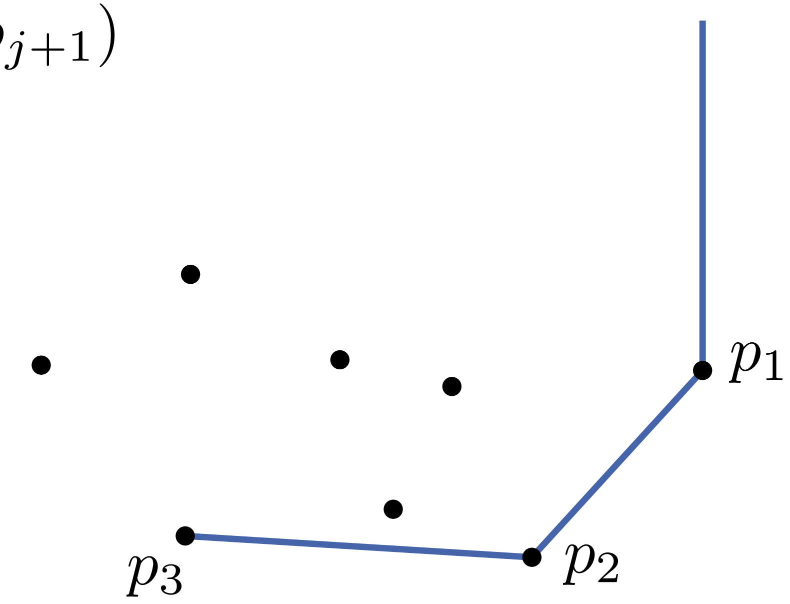


Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
return $(p_1, \ldots, p_{j+1})$

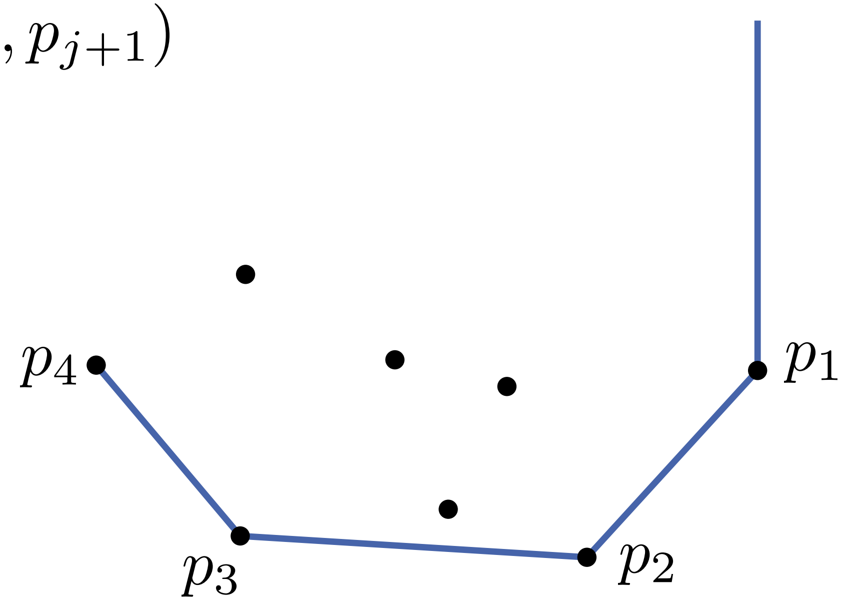


Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
return $(p_1, \ldots, p_{j+1})$

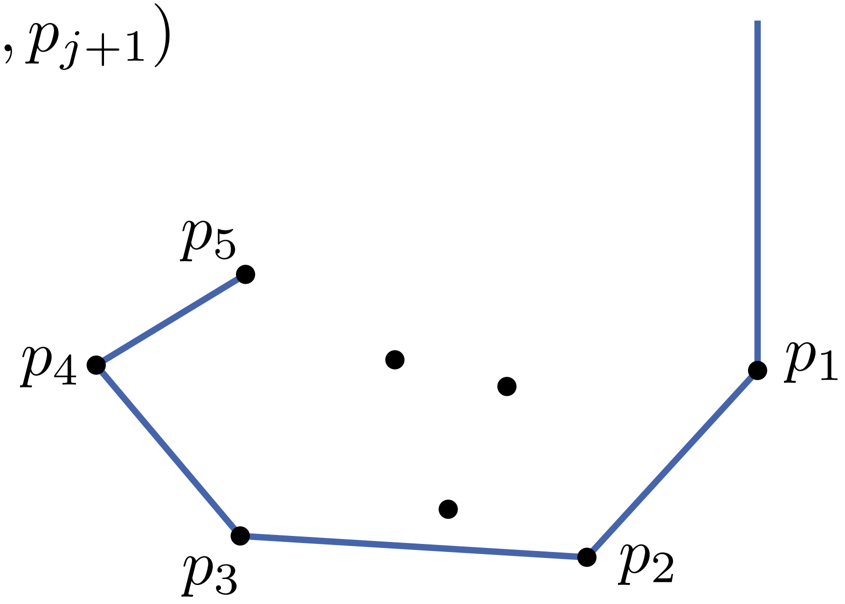


Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
return $(p_1, \ldots, p_{j+1})$

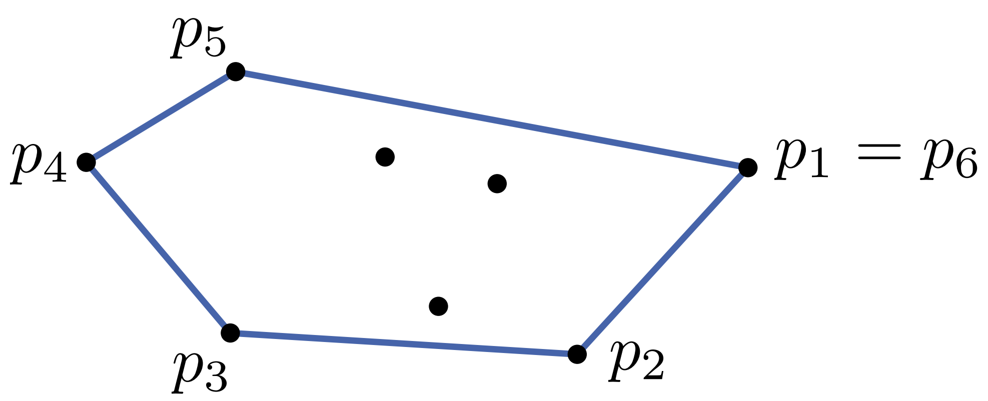


Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
return $(p_1, \ldots, p_{j+1})$



Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
    $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
    if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
return $(p_1, \ldots, p_{j+1})$

Correctness (ideas):

Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
return $(p_1, \ldots, p_{j+1})$

Correctness (ideas):

- **Base Case**: p_1 lies on convex hull.

Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
return $(p_1, \ldots, p_{j+1})$

Correctness (ideas):

- **Base Case**: p_1 lies on convex hull.
- **Assumption**: First i points belong to convex hull CH(P)

Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
return $(p_1, \ldots, p_{j+1})$

Correctness (ideas):

- **Base Case**: p_1 lies on convex hull.
- **Assumption**: First i points belong to convex hull CH(P)
- **Step**: By assump. p_{i+1} lies to the right of line $\overrightarrow{p_{i-1}p_i}$ $\Rightarrow$ 'right bend'
  By the chosen angle: all points lie to the right of line $\overrightarrow{p_i p_{i+1}}$

Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j+1$
return $(p_1, \ldots, p_{j+1})$

Degenerated cases:

Alternative: Gift Wrapping

Idea: Begin with a point p_1 of $CH(P)$, then find the next edge of $CH(P)$ in clockwise order.

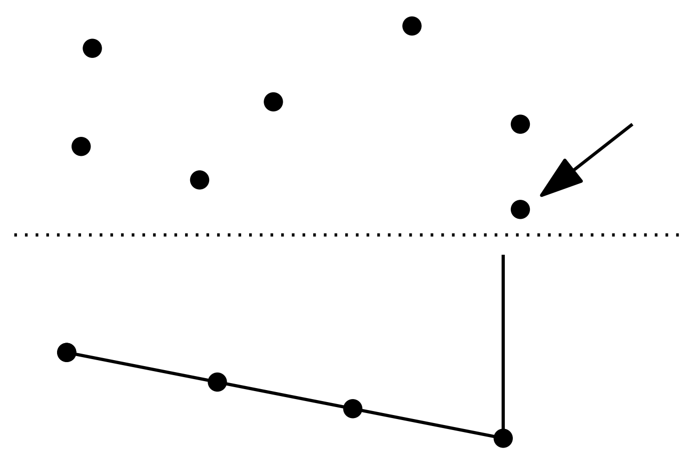
GiftWrapping(P)

$p_1 = (x_1, y_1) \leftarrow$ rightmost point in P; $p_0 \leftarrow (x_1, \infty)$; $j \leftarrow 1$
while true **do**
  $p_{j+1} \leftarrow \arg\max\{\angle p_{j-1}, p_j, q \mid q \in P \setminus \{p_{j-1}, p_j\}\}$
  if $p_{j+1} = p_1$ **then** break **else** $j \leftarrow j + 1$
return $(p_1, \ldots, p_{j+1})$

Degenerated cases:

1. Choice of p_1 is not unique.
   Choose the bottommost rightmost point.

2. Choice of p_{j+1} is not unique.
   Choose the point of largest distances.



Computation of Tangents

Given: convex polygon P (clockswise) and point p outside of P

Find: *right* tangent at P through p in $O(\log n)$ time.

Right tangent means polygon lies left to tangent.

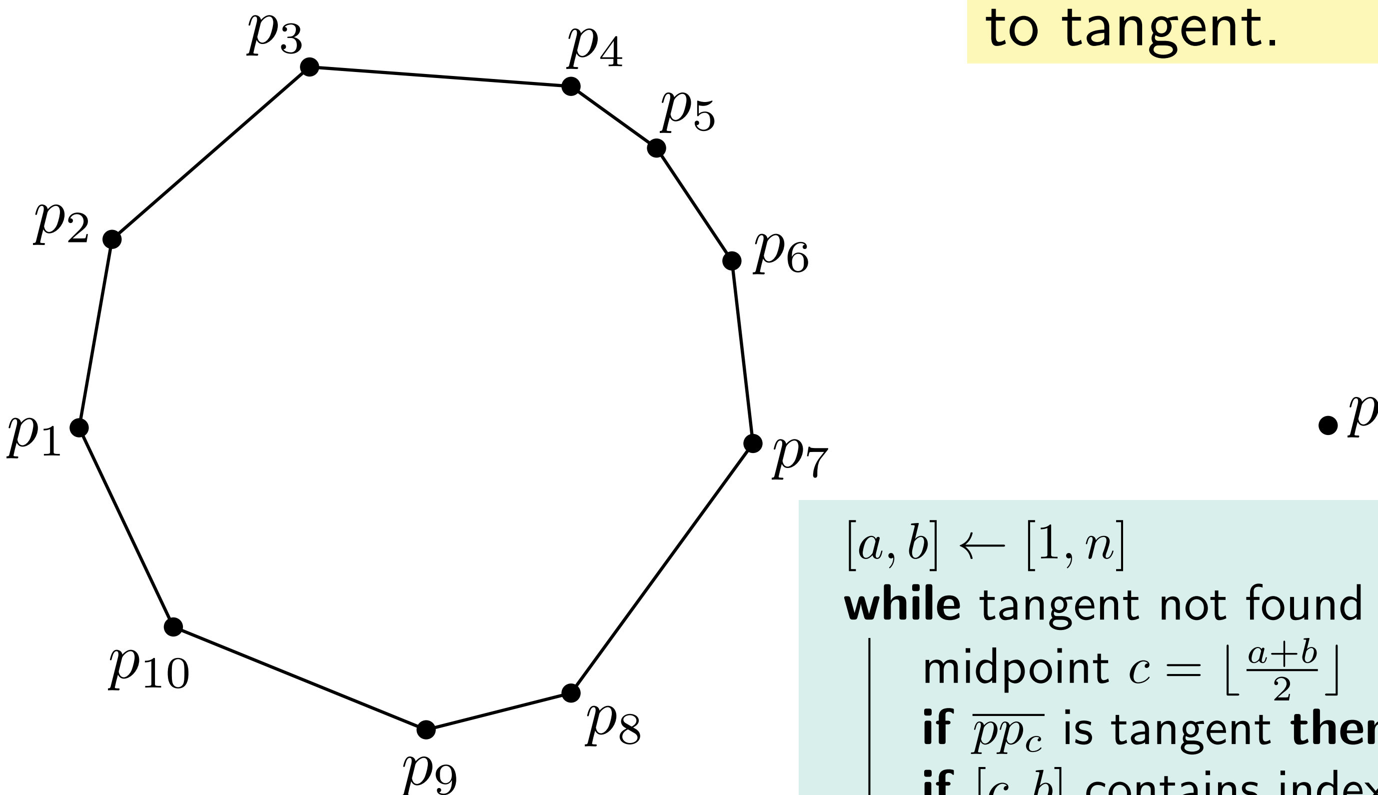
Computation of Tangents

Given: convex polygon P (clockswise) and point p outside of P

Find: *right* tangent at P through p in $O(\log n)$ time.

Idea: Use binary search.

Right tangent means polygon lies left to tangent.



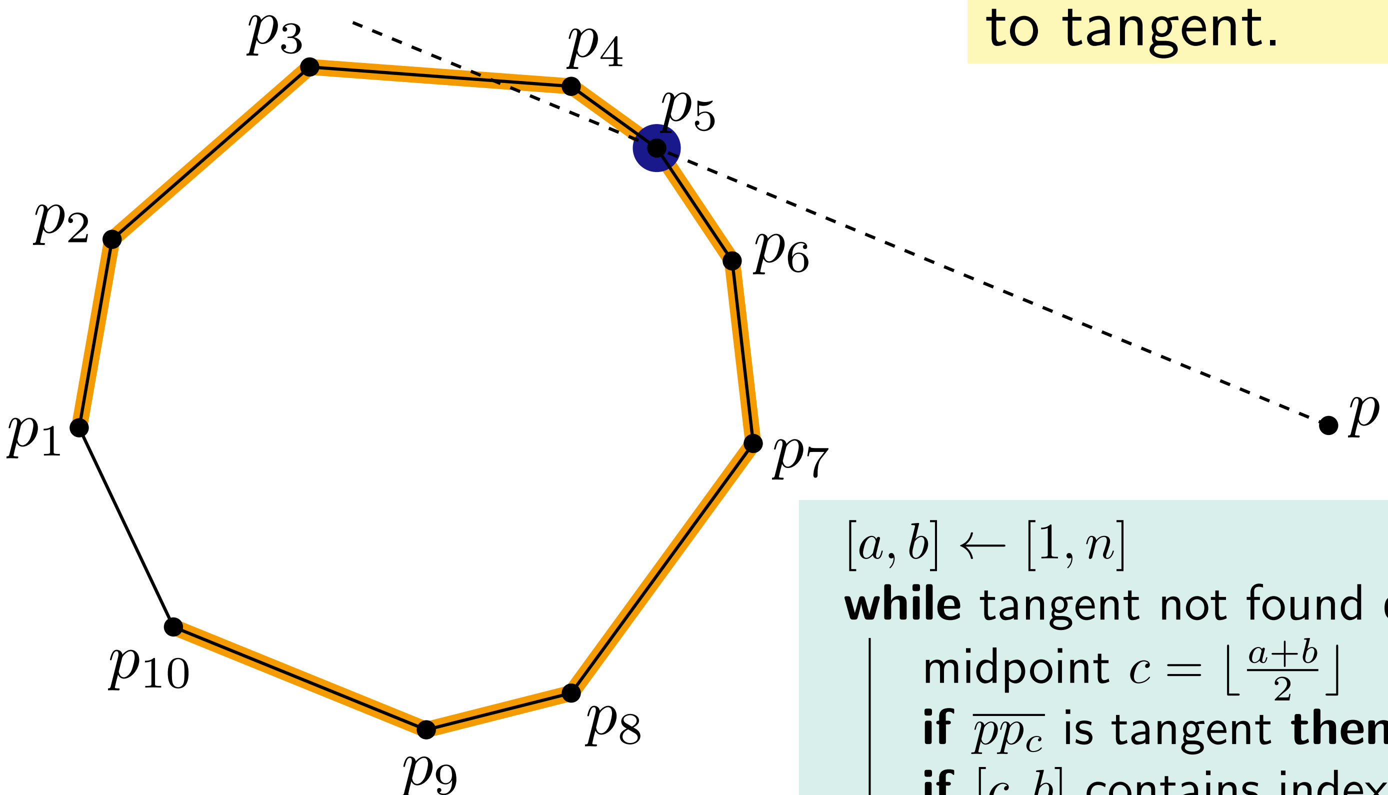
$[a, b] \leftarrow [1, n]$
while tangent not found **do**
  midpoint $c = \lfloor \frac{a+b}{2} \rfloor$
  if $\overline{pp_c}$ is tangent **then return** p_c
  if $[c, b]$ contains index of contact point **then**
    $[a, b] \leftarrow [c, b]$
  else
    $[a, b] \leftarrow [a, c]$

Computation of Tangents

Given: convex polygon P (clockswise) and point p outside of P
Find: *right* tangent at P through p in $O(\log n)$ time.

Idea: Use binary search.

Right tangent means polygon lies left to tangent.



$[a, b] \leftarrow [1, n]$
while tangent not found **do**
  midpoint $c = \lfloor \frac{a+b}{2} \rfloor$
  if $\overline{pp_c}$ is tangent **then return** p_c
  if $[c, b]$ contains index of contact point **then**
    $[a, b] \leftarrow [c, b]$
  else
    $[a, b] \leftarrow [a, c]$

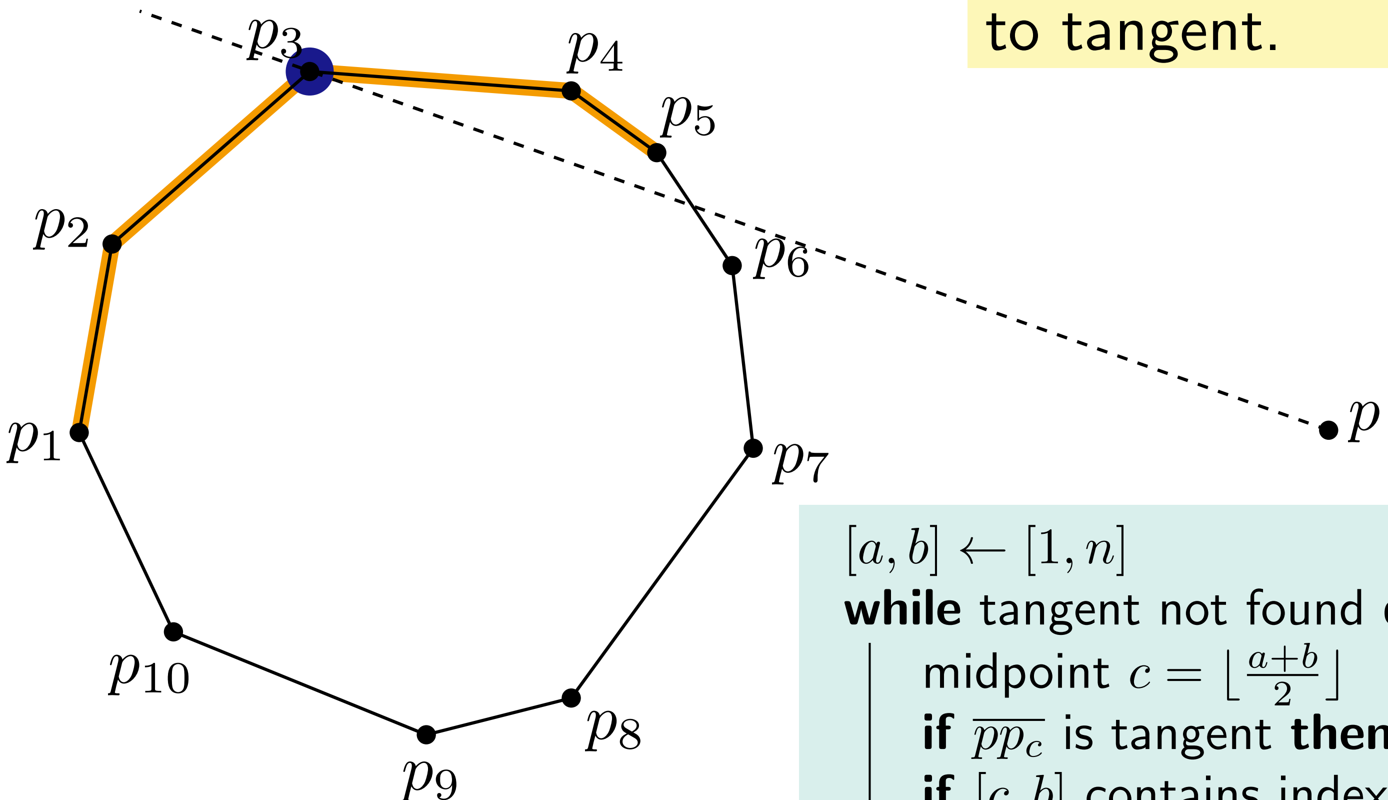
Computation of Tangents

Given: convex polygon P (clockwise) and point p outside of P

Find: *right* tangent at P through p in $O(\log n)$ time.

Idea: Use binary search.

Right tangent means polygon lies left to tangent.



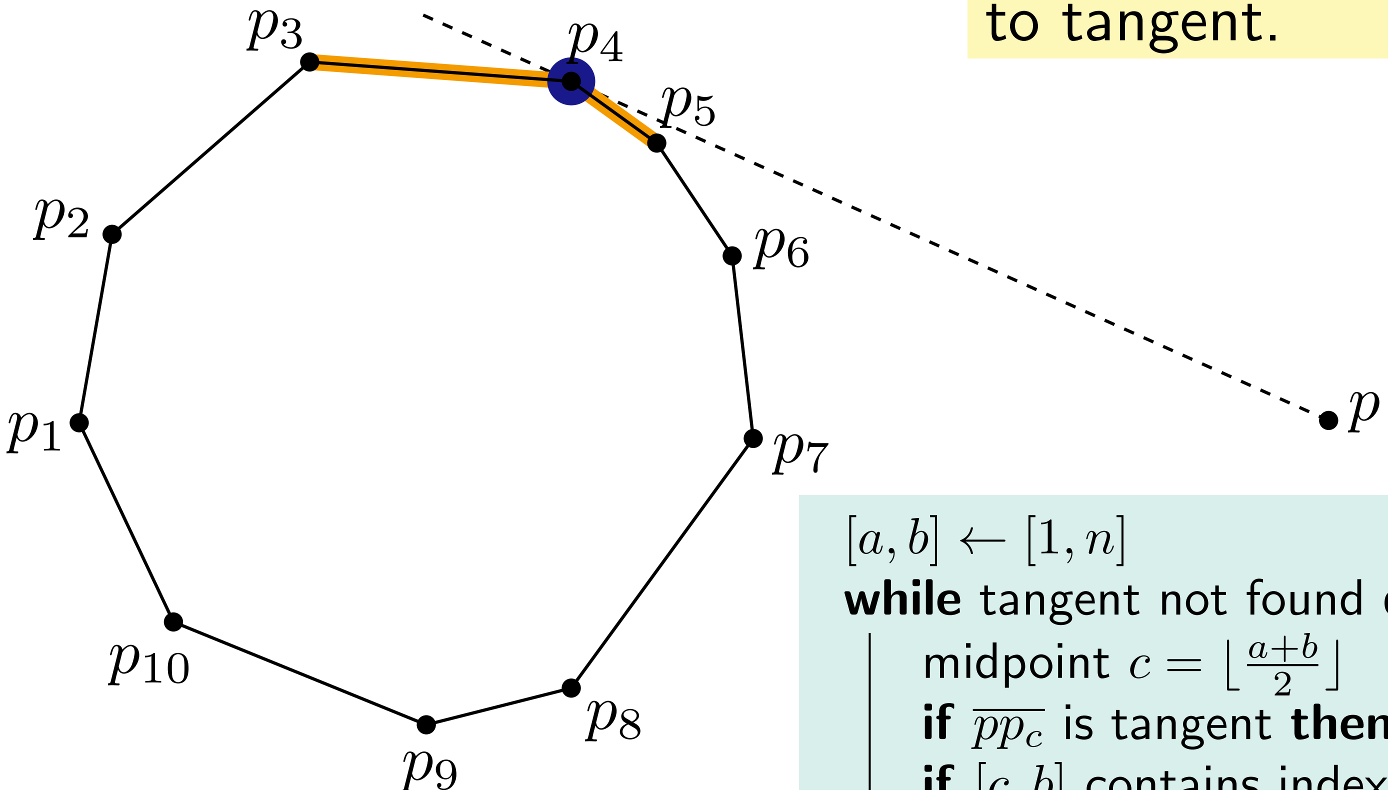
$[a, b] \leftarrow [1, n]$
while tangent not found **do**
  midpoint $c = \lfloor \frac{a+b}{2} \rfloor$
  if $\overline{pp_c}$ is tangent **then return** p_c
  if $[c, b]$ contains index of contact point **then**
    $[a, b] \leftarrow [c, b]$
  else
    $[a, b] \leftarrow [a, c]$

Computation of Tangents

Given: convex polygon P (clockswise) and point p outside of P
Find: *right* tangent at P through p in $O(\log n)$ time.

Idea: Use binary search.

Right tangent means polygon lies left to tangent.




```
[a, b] ← [1, n]
while tangent not found do
    midpoint c = ⌊(a+b)/2⌋
    if pp_c is tangent then return p_c
    if [c, b] contains index of contact point then
        [a, b] ← [c, b]
    else
        [a, b] ← [a, c]
```

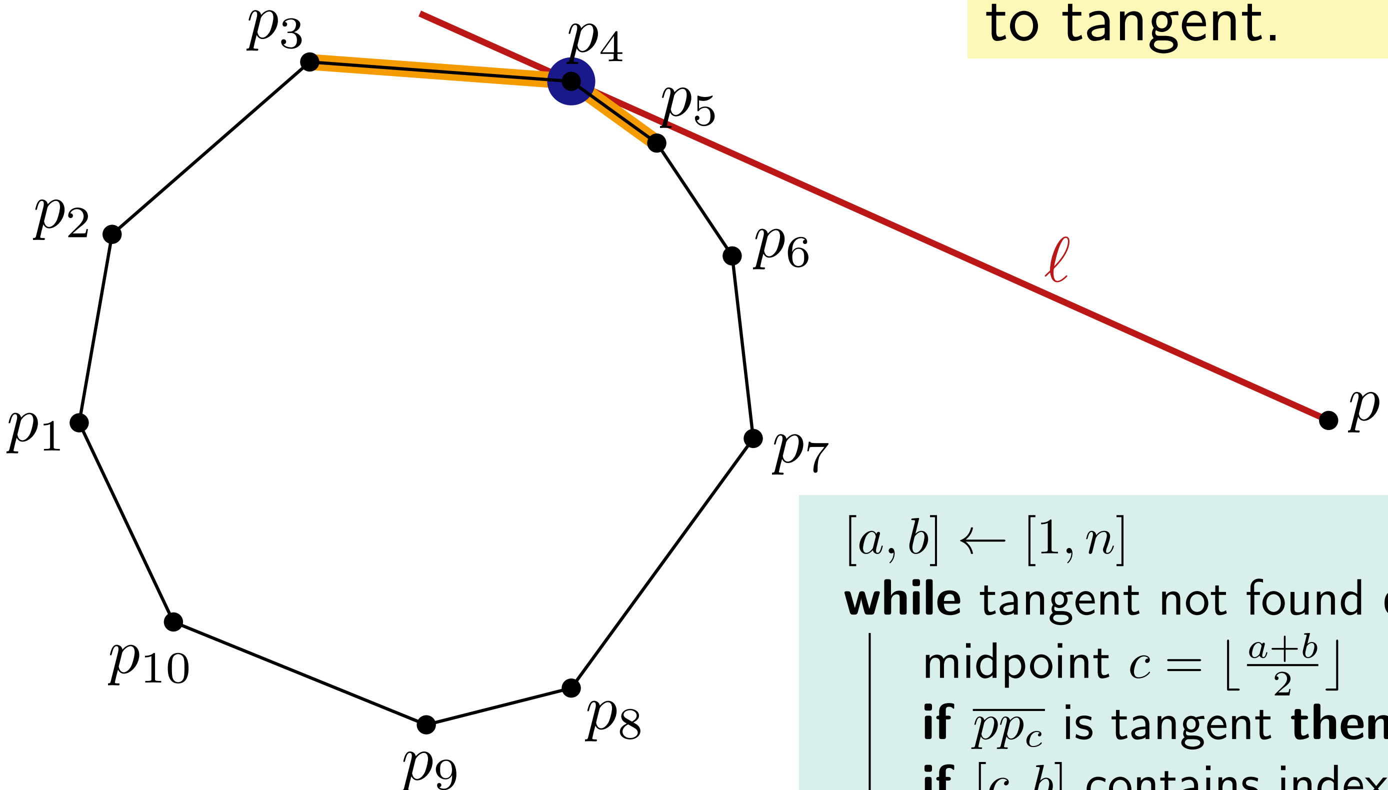
Computation of Tangents

Given: convex polygon P (clockswise) and point p outside of P

Find: *right* tangent at P through p in $O(\log n)$ time.

Idea: Use binary search.

Right tangent means polygon lies left to tangent.



$[a, b] \leftarrow [1, n]$

while tangent not found **do**

- midpoint $c = \lfloor \frac{a+b}{2} \rfloor$
- **if** $\overline{pp_c}$ is tangent **then return** p_c
- **if** $[c, b]$ contains index of contact point **then**
  - $[a, b] \leftarrow [c, b]$
- **else**
  - $[a, b] \leftarrow [a, c]$

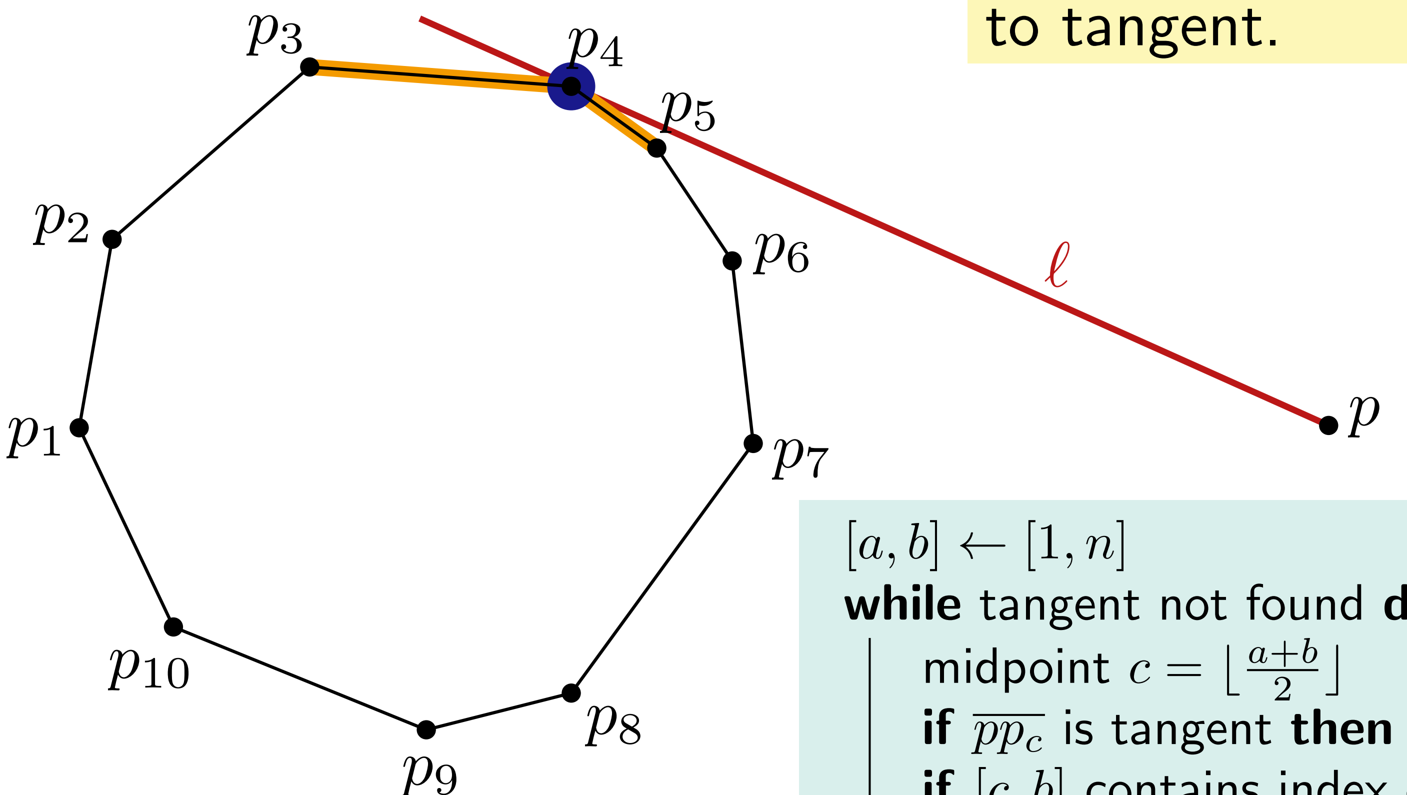
Computation of Tangents

Given: convex polygon P (clockswise) and point p outside of P

Find: *right* tangent at P through p in $O(\log n)$ time.

Idea: Use binary search.

Right tangent means polygon lies left to tangent.



$[a, b] \leftarrow [1, n]$

while tangent not found **do**

  midpoint $c = \lfloor \frac{a+b}{2} \rfloor$

  if $\overline{pp_c}$ is tangent **then return** p_c

  if $[c, b]$ contains index of contact point **then**

    $[a, b] \leftarrow [c, b]$

  else

    $[a, b] \leftarrow [a, c]$

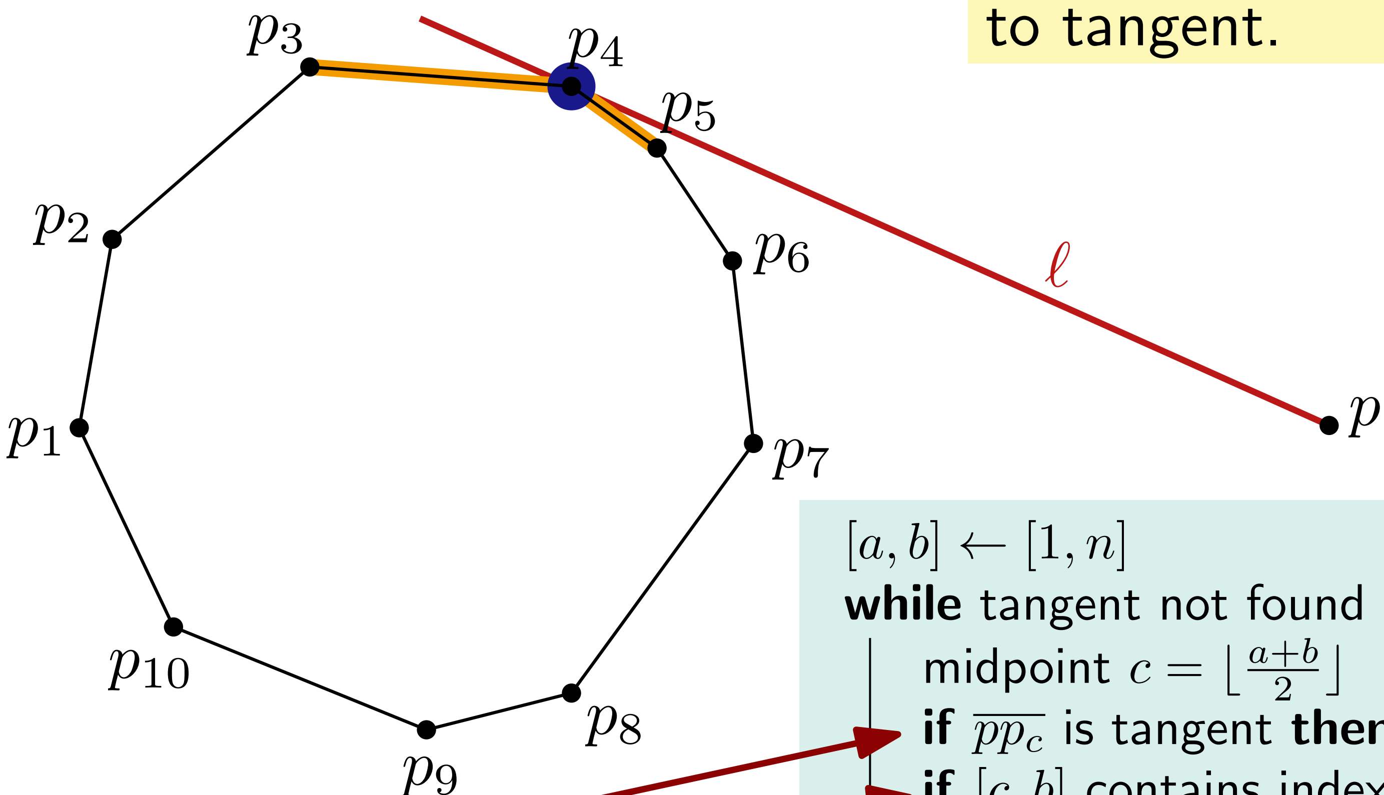
Computation of Tangents

Given: convex polygon P (clockswise) and point p outside of P

Find: *right* tangent at P through p in $O(\log n)$ time.

Idea: Use binary search.

Right tangent means polygon lies left to tangent.



$[a, b] \leftarrow [1, n]$

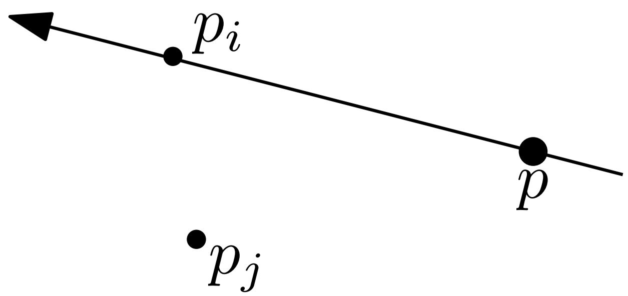
while tangent not found **do**

- midpoint $c = \lfloor \frac{a+b}{2} \rfloor$
- **if** $\overline{pp_c}$ is tangent **then return** p_c
- **if** $[c, b]$ contains index of contact point **then**
  - $[a, b] \leftarrow [c, b]$
- **else**
  - $[a, b] \leftarrow [a, c]$

How to test in constant time?

Computation of Tangents

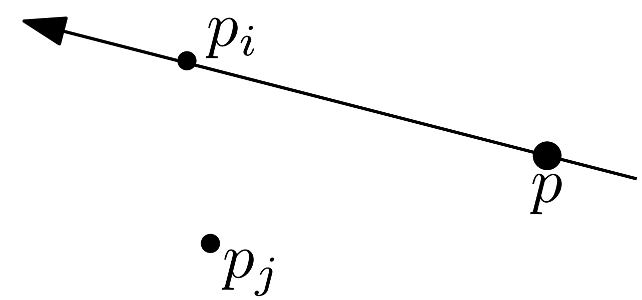
$[a, b] \leftarrow [1, n]$
while tangent not found **do**
  middle $c = \lfloor \frac{a+b}{2} \rfloor$
  if $\overline{pp_c}$ is tangent **then return** p_c
  if $[c, b]$ contains index of contact point **then**
    $[a, b] \leftarrow [c, b]$
  else
    $[a, b] \leftarrow [a, c]$



p_i lies above p_j, if p_j lies left to $\overrightarrow{pp_i}$.

Computation of Tangents

$[a, b] \leftarrow [1, n]$
while tangent not found **do**
  middle $c = \lfloor \frac{a+b}{2} \rfloor$
  if $\overline{pp_c}$ is tangent **then return** p_c
  if $[c, b]$ contains index of contact point **then**
    $[a, b] \leftarrow [c, b]$
  else
    $[a, b] \leftarrow [a, c]$

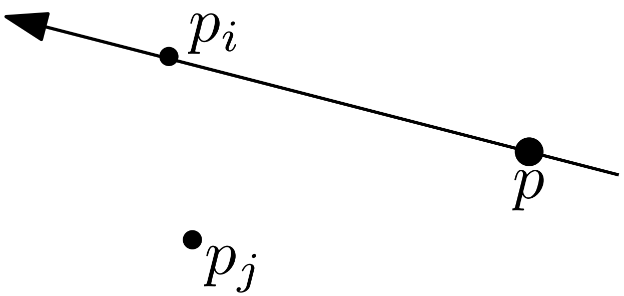


p_i lies above p_j, if p_j lies left to $\overrightarrow{pp_i}$.

Assumption: $\overrightarrow{pp_i}$ points from right to left.

Computation of Tangents

$[a, b] \leftarrow [1, n]$
while tangent not found **do**
  middle $c = \lfloor \frac{a+b}{2} \rfloor$
  if $\overline{pp_c}$ is tangent **then return** p_c
  if $[c, b]$ contains index of contact point **then**
    $[a, b] \leftarrow [c, b]$
  else
    $[a, b] \leftarrow [a, c]$

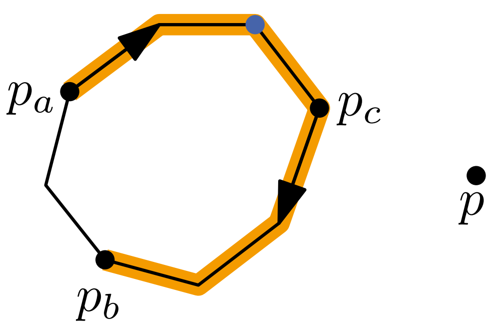


p_i lies above p_j, if p_j lies left to $\overrightarrow{pp_i}$.

Assumption: $\overrightarrow{pp_i}$ points from right to left.

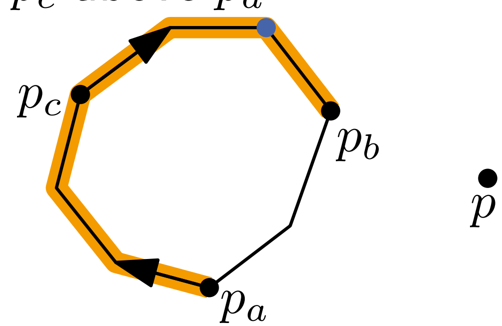
p_{a+1} above p_a:

p_c above p_{c+1}



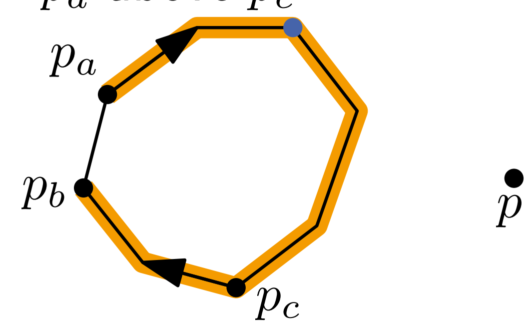
$[a, b] \leftarrow [a, c]$

p_{c+1} above p_c
p_c above p_a



$[a, b] \leftarrow [c, b]$

p_{c+1} above p_c
p_a above p_c



$[a, b] \leftarrow [a, c]$

p_a above p_{a+1}: Analogous statements.

Lower Bound

We require that any algorithm computing the convex hull of a given set of points returns the convex hull vertices as a clockwise sorted list of points.

Lower Bound

We require that any algorithm computing the convex hull of a given set of points returns the convex hull vertices as a clockwise sorted list of points.

1. Show that any algorithm for computing the convex hull of n points has a worst case running time of $\Omega(n \log n)$ and thus *Graham Scan* is worst-case optimal.

Lower Bound

We require that any algorithm computing the convex hull of a given set of points returns the convex hull vertices as a clockwise sorted list of points.

1. Show that any algorithm for computing the convex hull of n points has a worst case running time of $\Omega(n \log n)$ and thus *Graham Scan* is worst-case optimal.

2. Why is the running time of the *gift wrapping* algorithm not in contradiction to part (a)?

Outline

Convex Hull

Line Segment Intersection

Problem Formulation

Given: Set $S = \{s_1, \ldots, s_n\}$ of line segments in the plane

Output:
- all intersections of two or more line segments
- for each intersection, the line segments involved.

Problem Formulation

Given: Set $S = \{s_1, \ldots, s_n\}$ of line segments in the plane

Output:
- all intersections of two or more line segments
- for each intersection, the line segments involved.

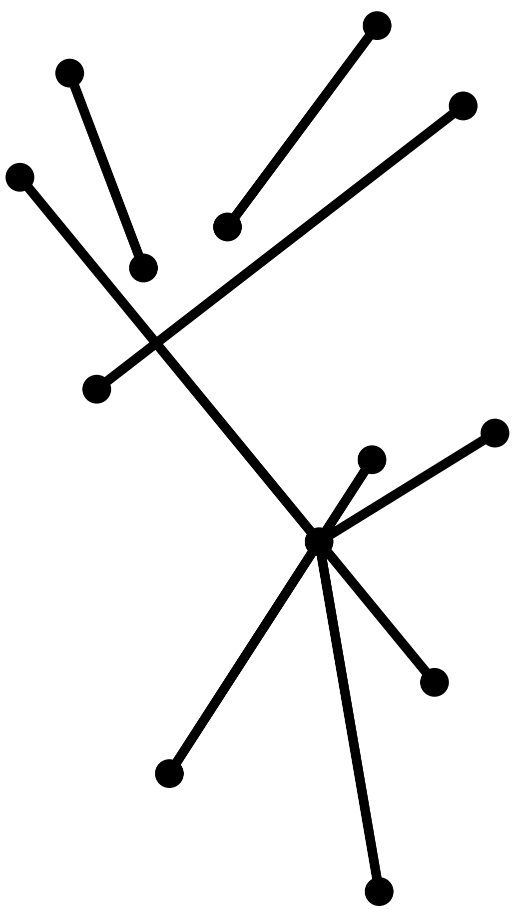
Def: Line segments are **closed**

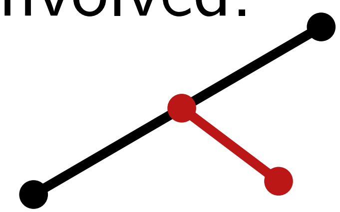
Problem Formulation

Given: Set $S = \{s_1, \ldots, s_n\}$ of line segments in the plane

Output:
- all intersections of two or more line segments
- for each intersection, the line segments involved.

Def: Line segments are **closed**

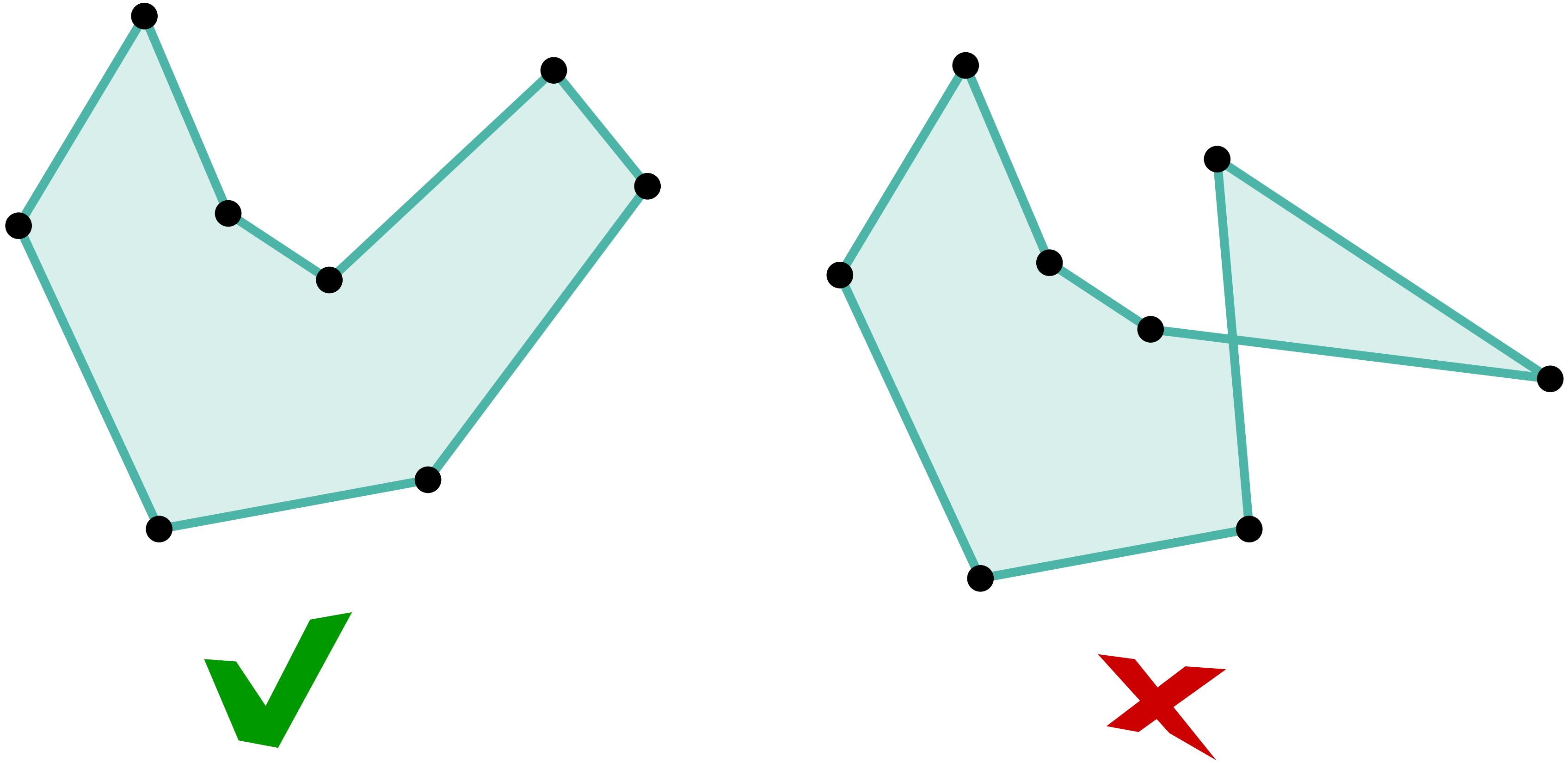




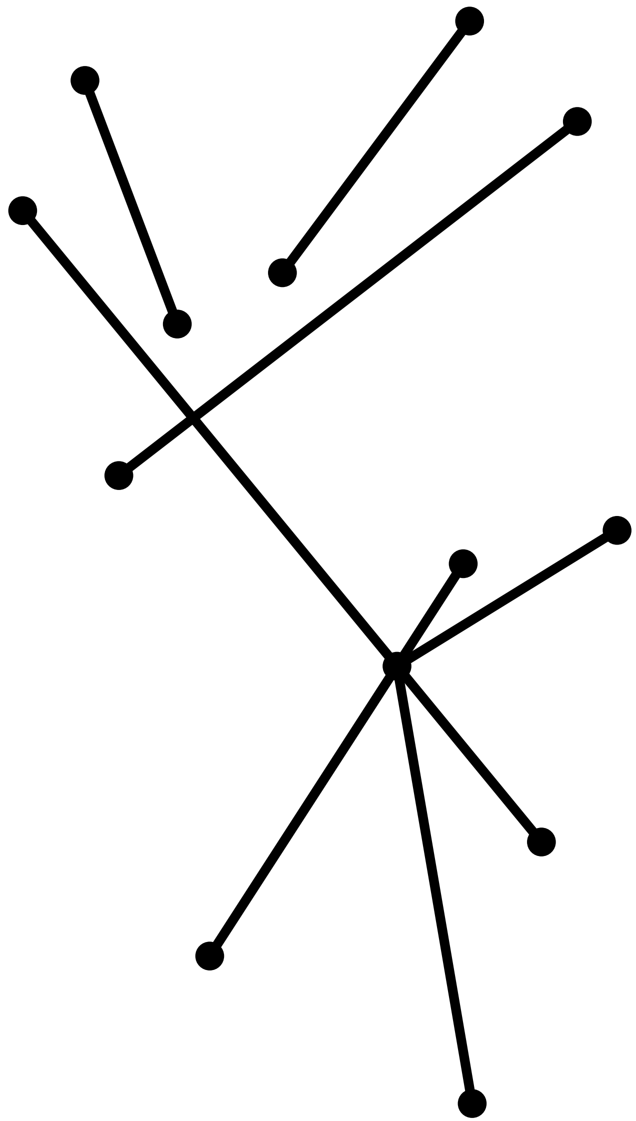
Warm Up

Find:

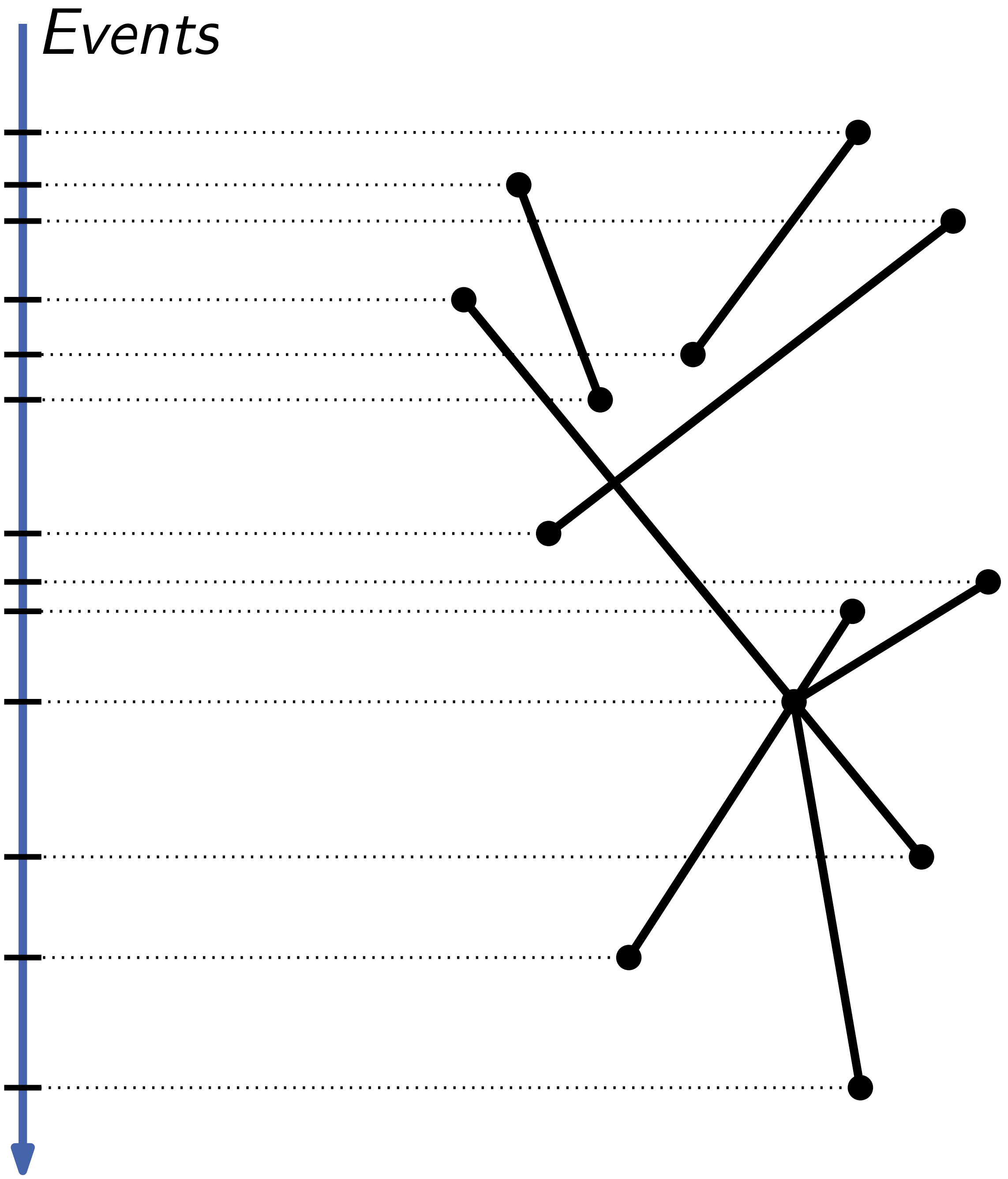
Algorithm that determines whether a polygon has no self-intersection using $\mathcal{O}(n \log n)$ running time.



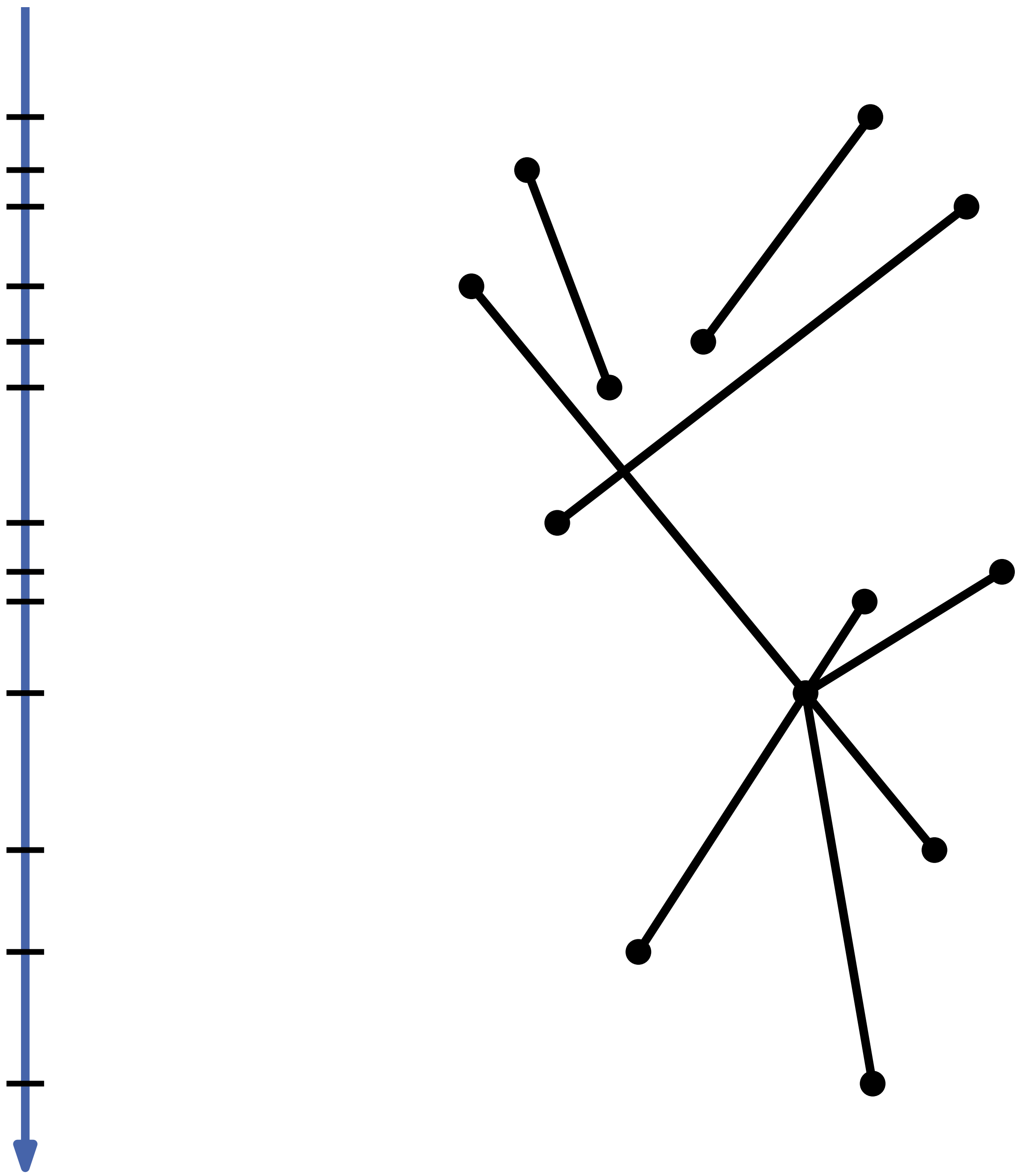
Sweep-Line: Example



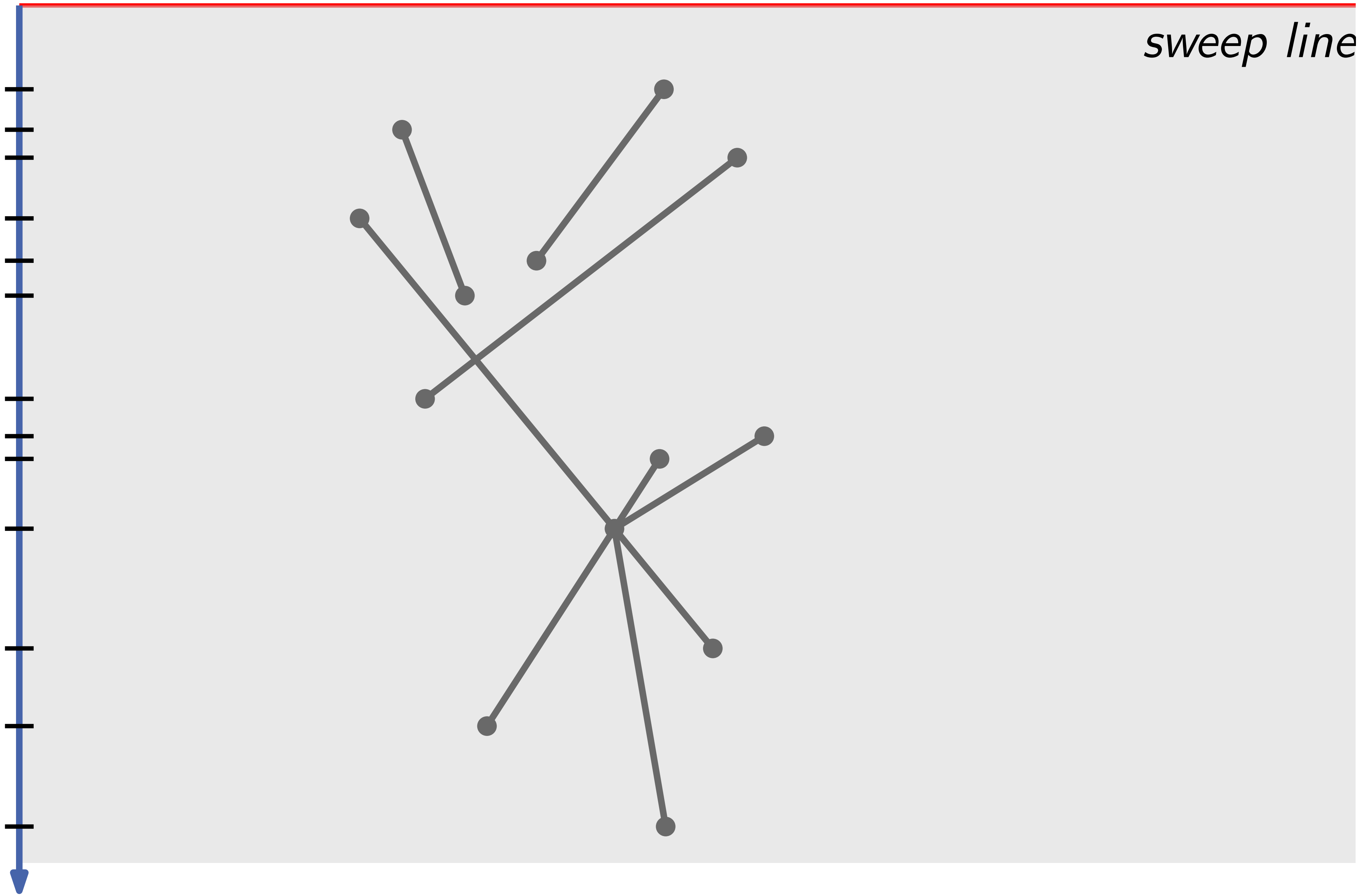
Sweep-Line: Example



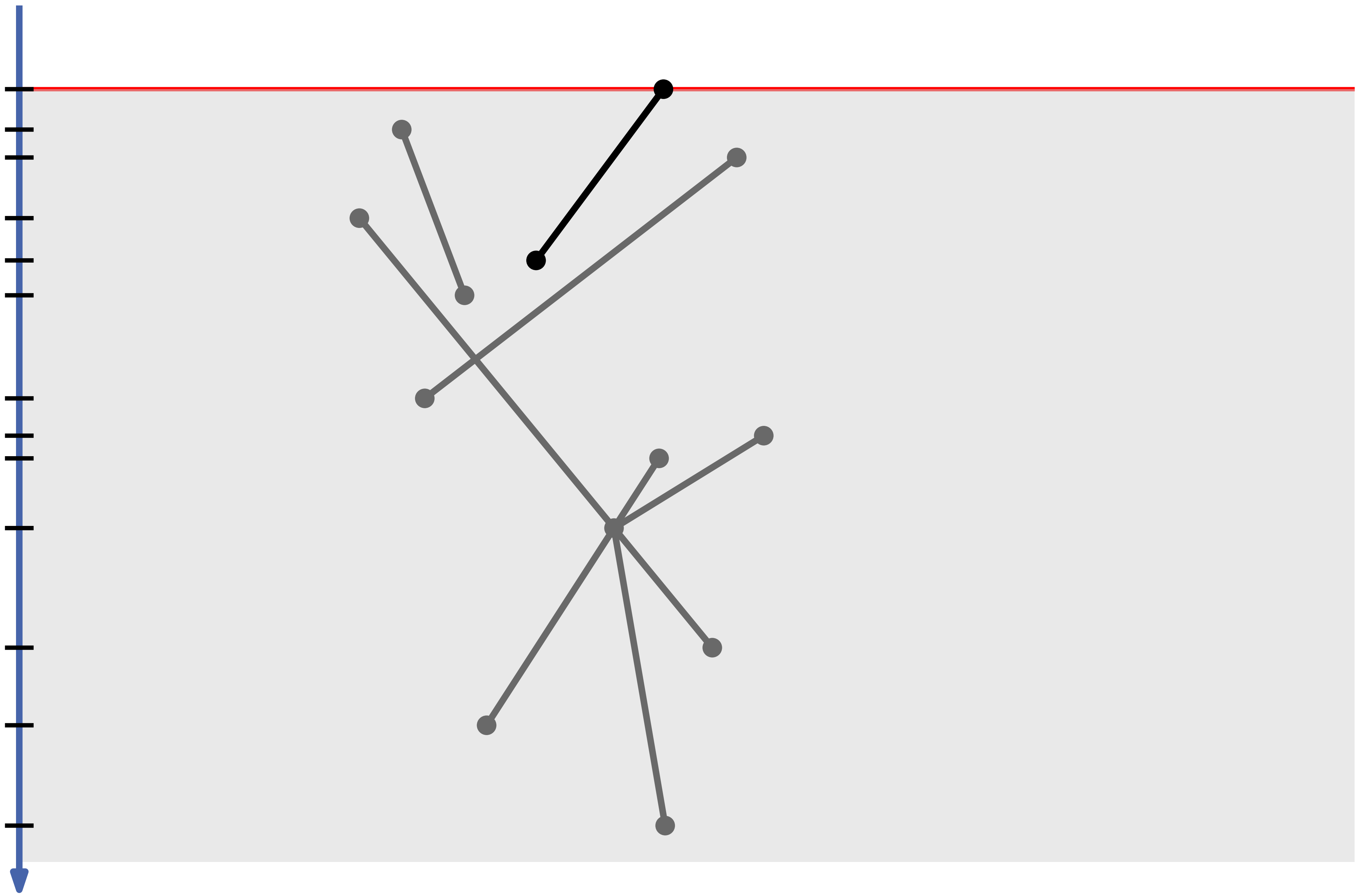
Sweep-Line: Example



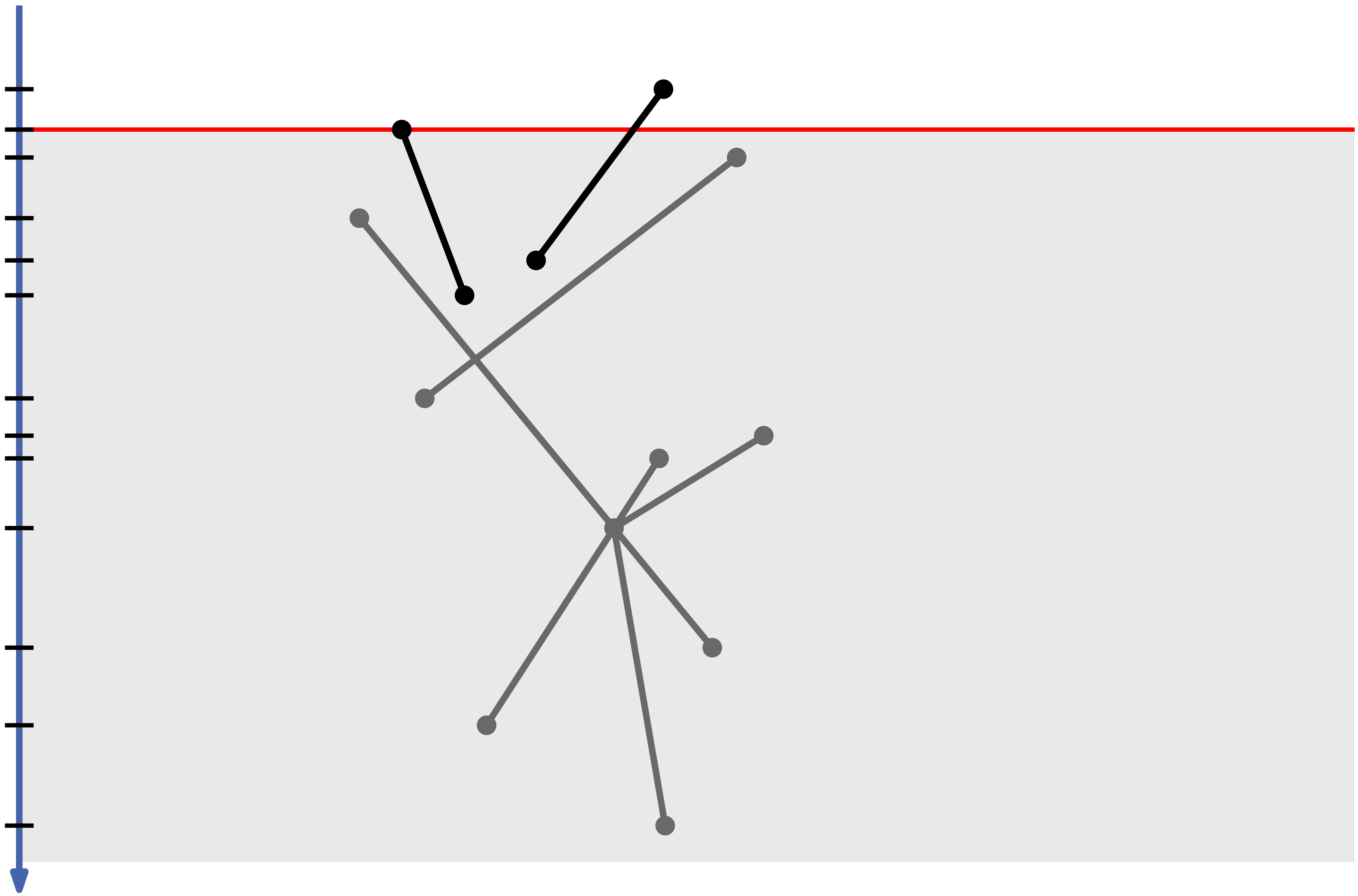
Sweep-Line: Example



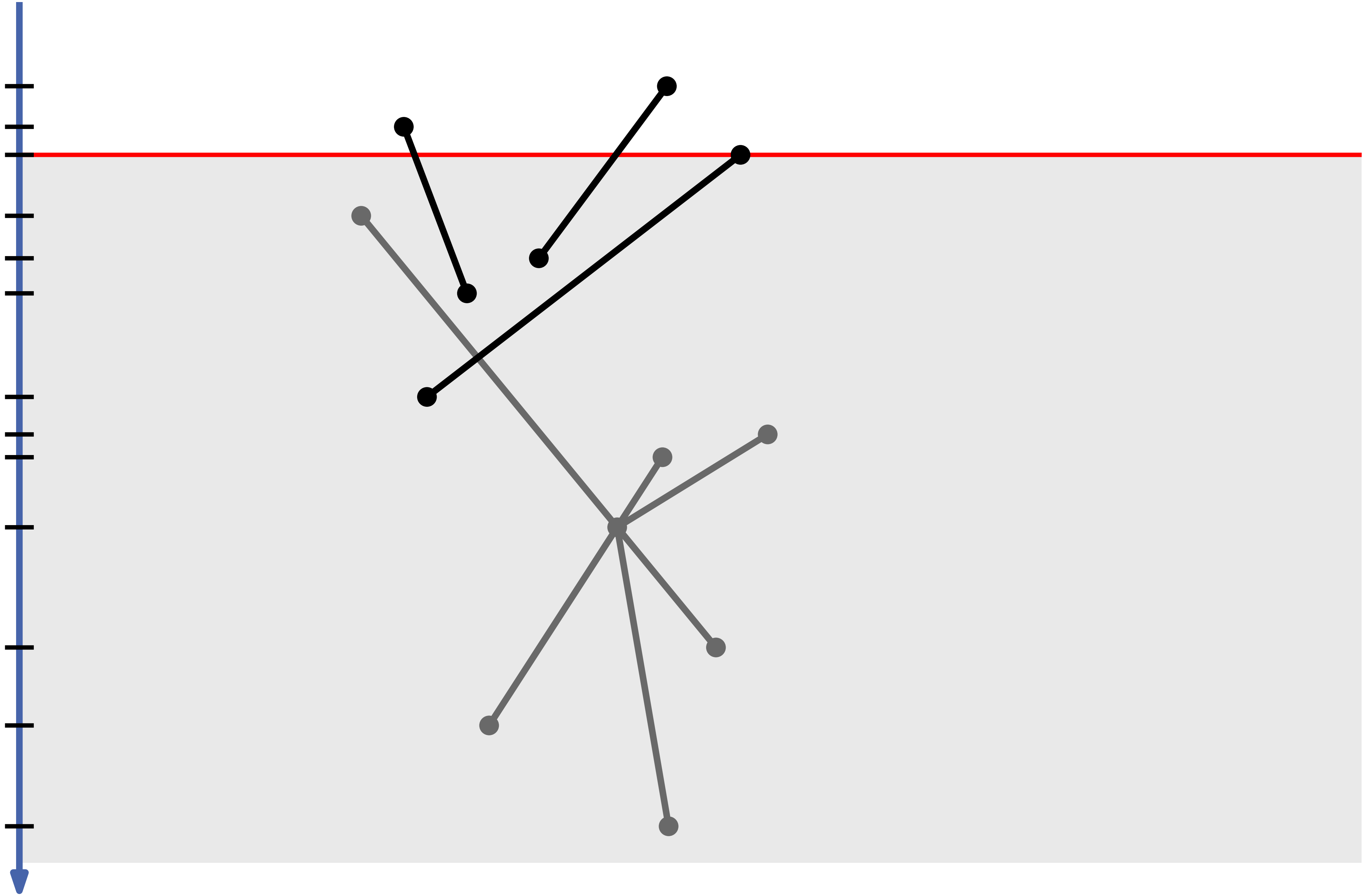
Sweep-Line: Example



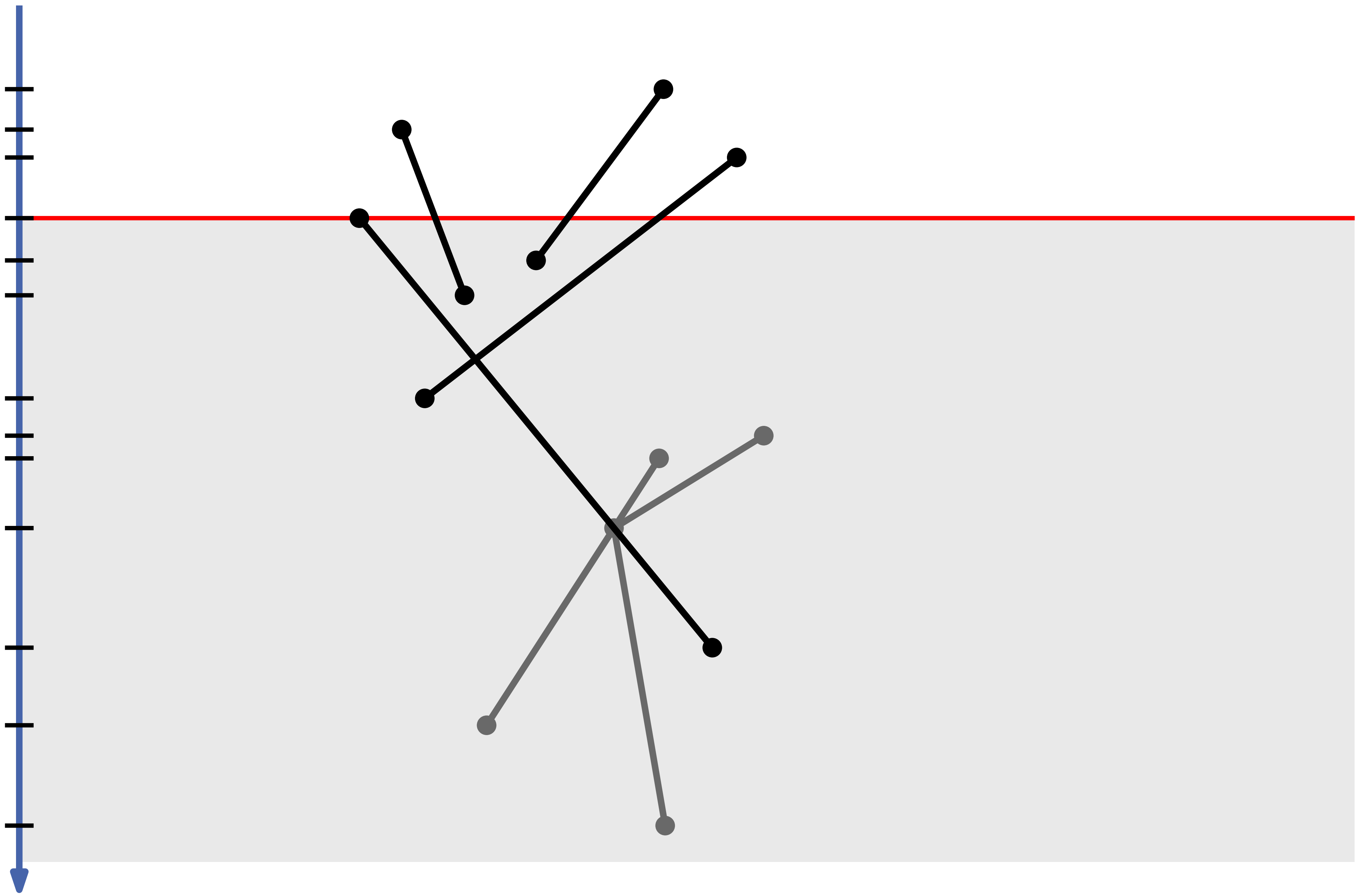
Sweep-Line: Example



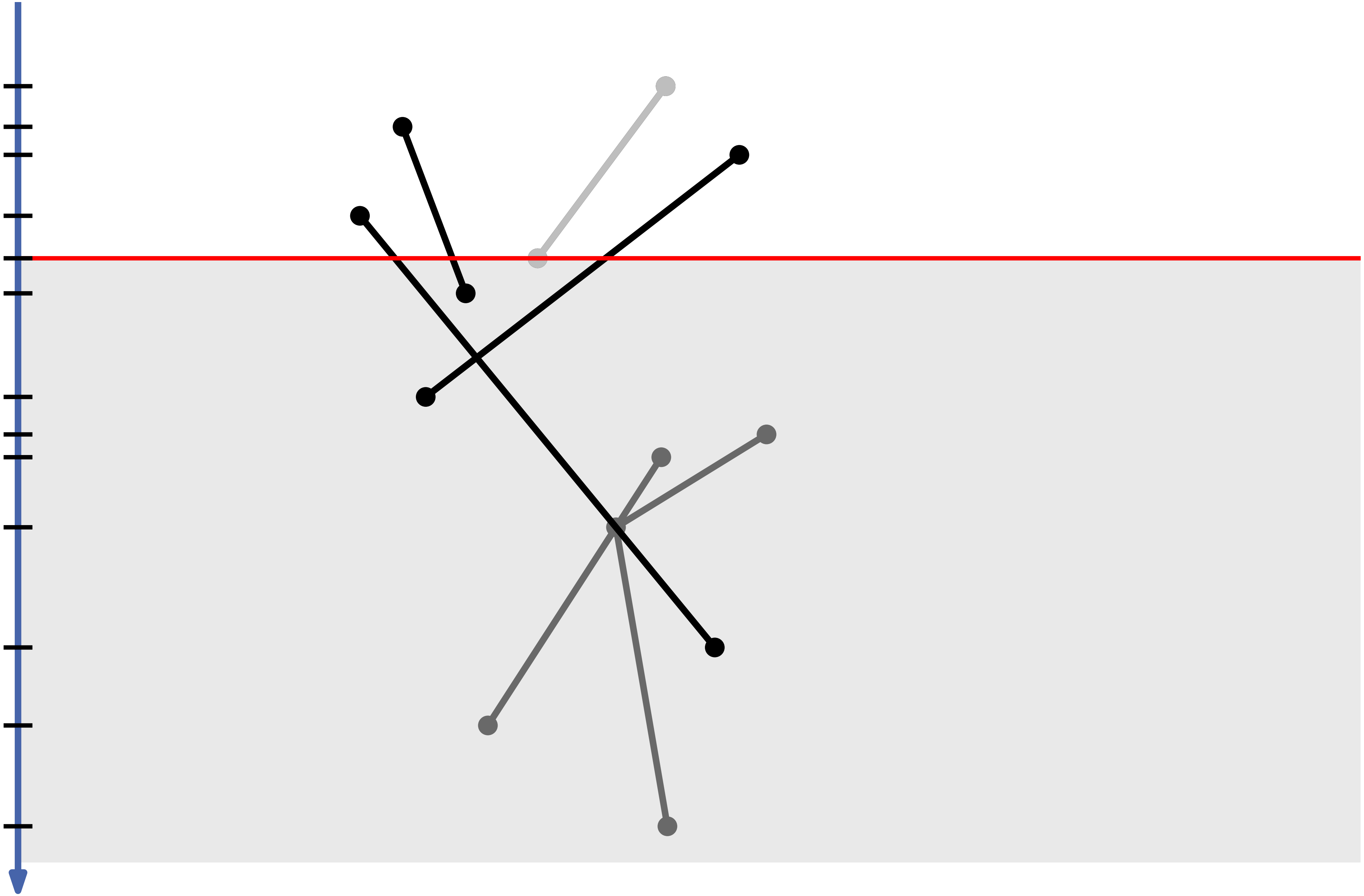
Sweep-Line: Example



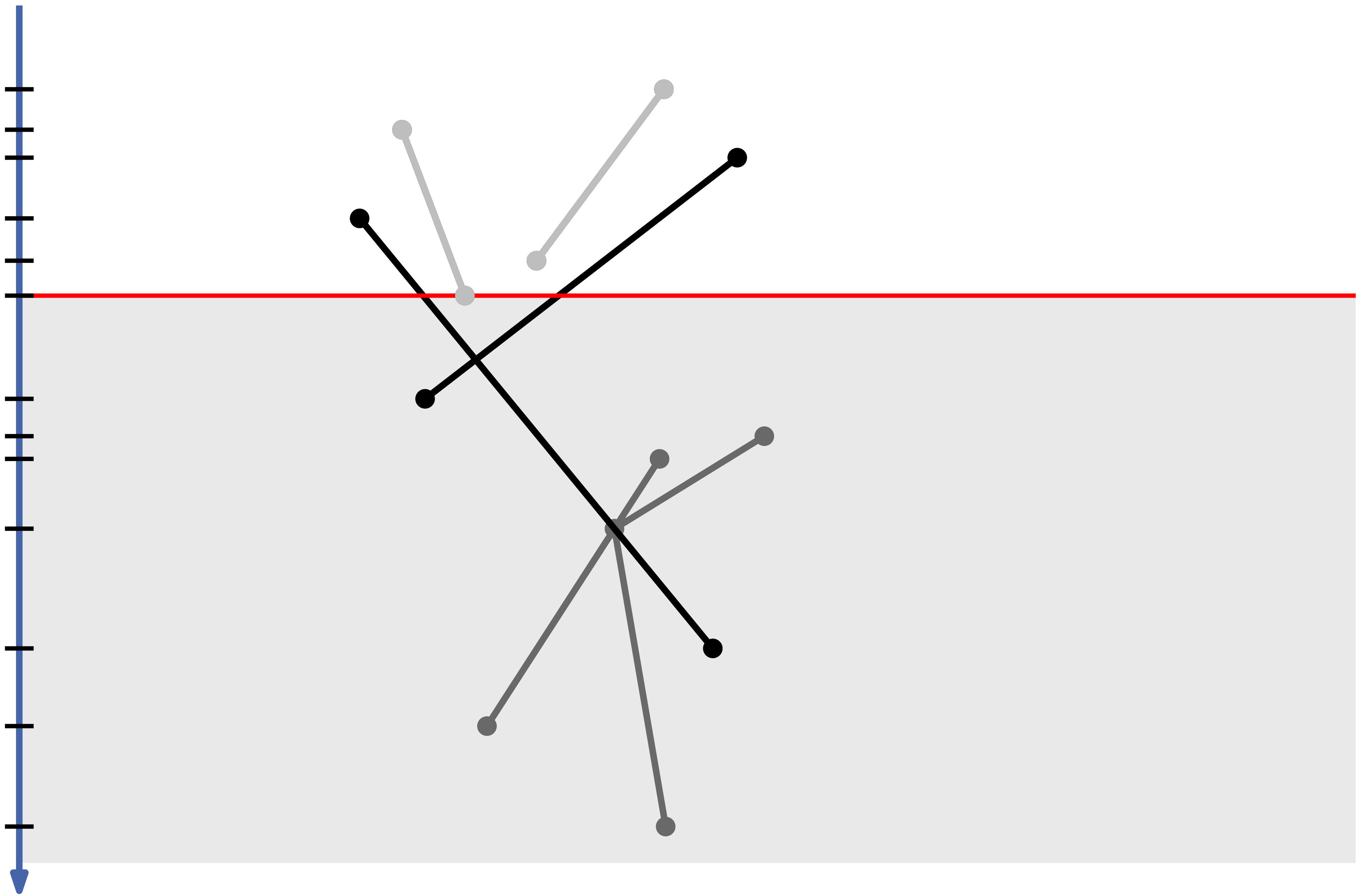
Sweep-Line: Example



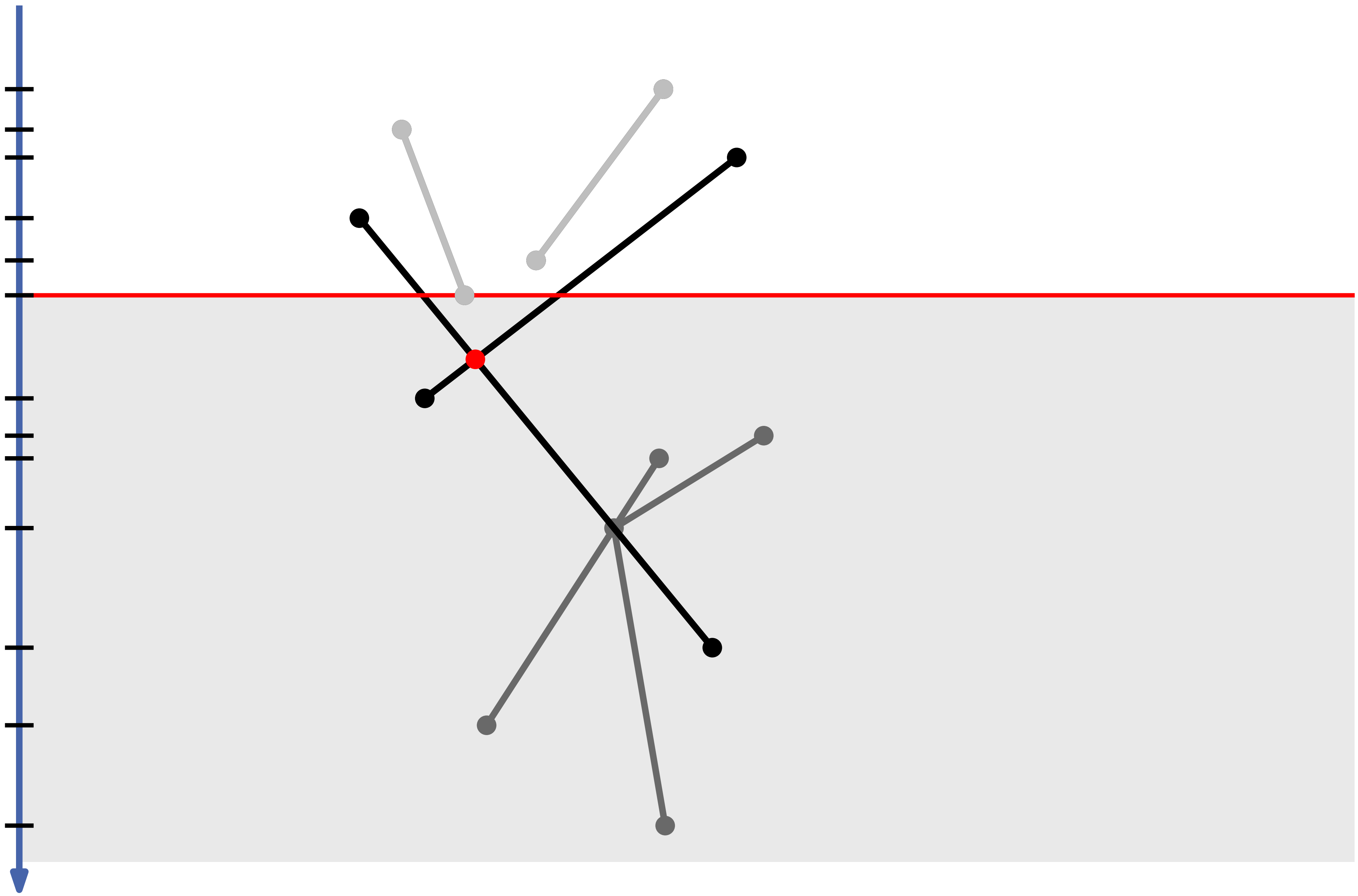
Sweep-Line: Example



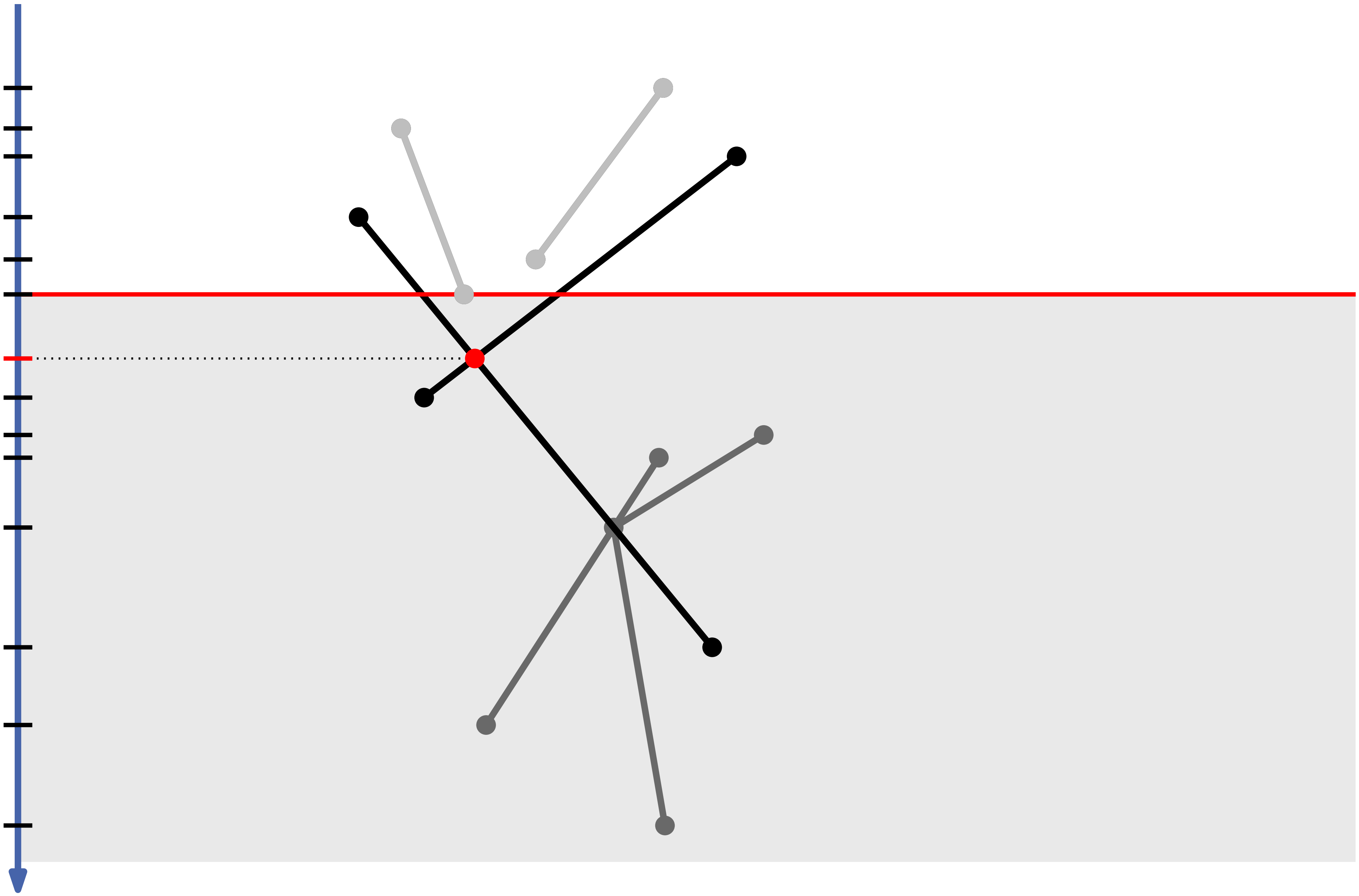
Sweep-Line: Example



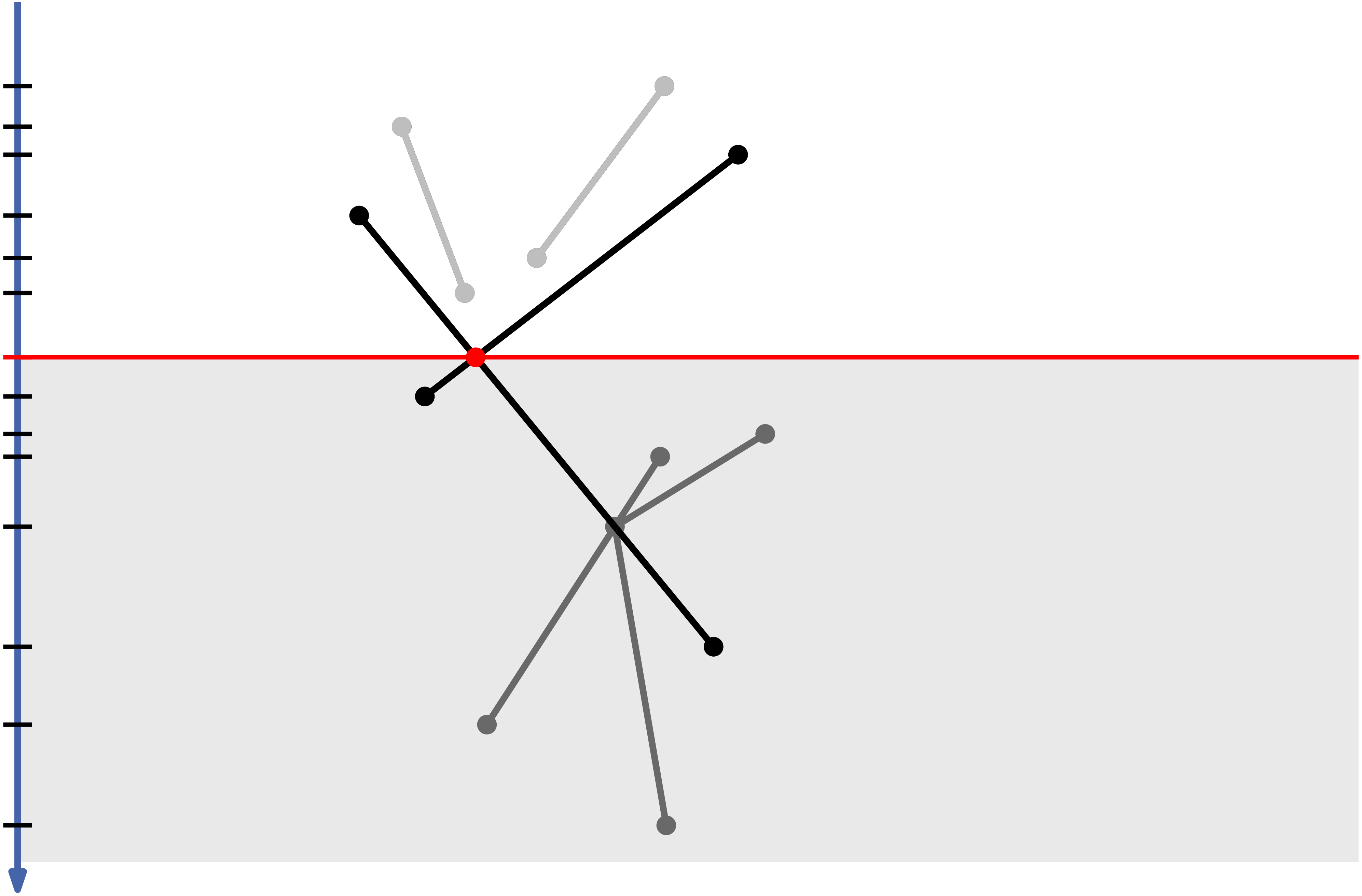
Sweep-Line: Example



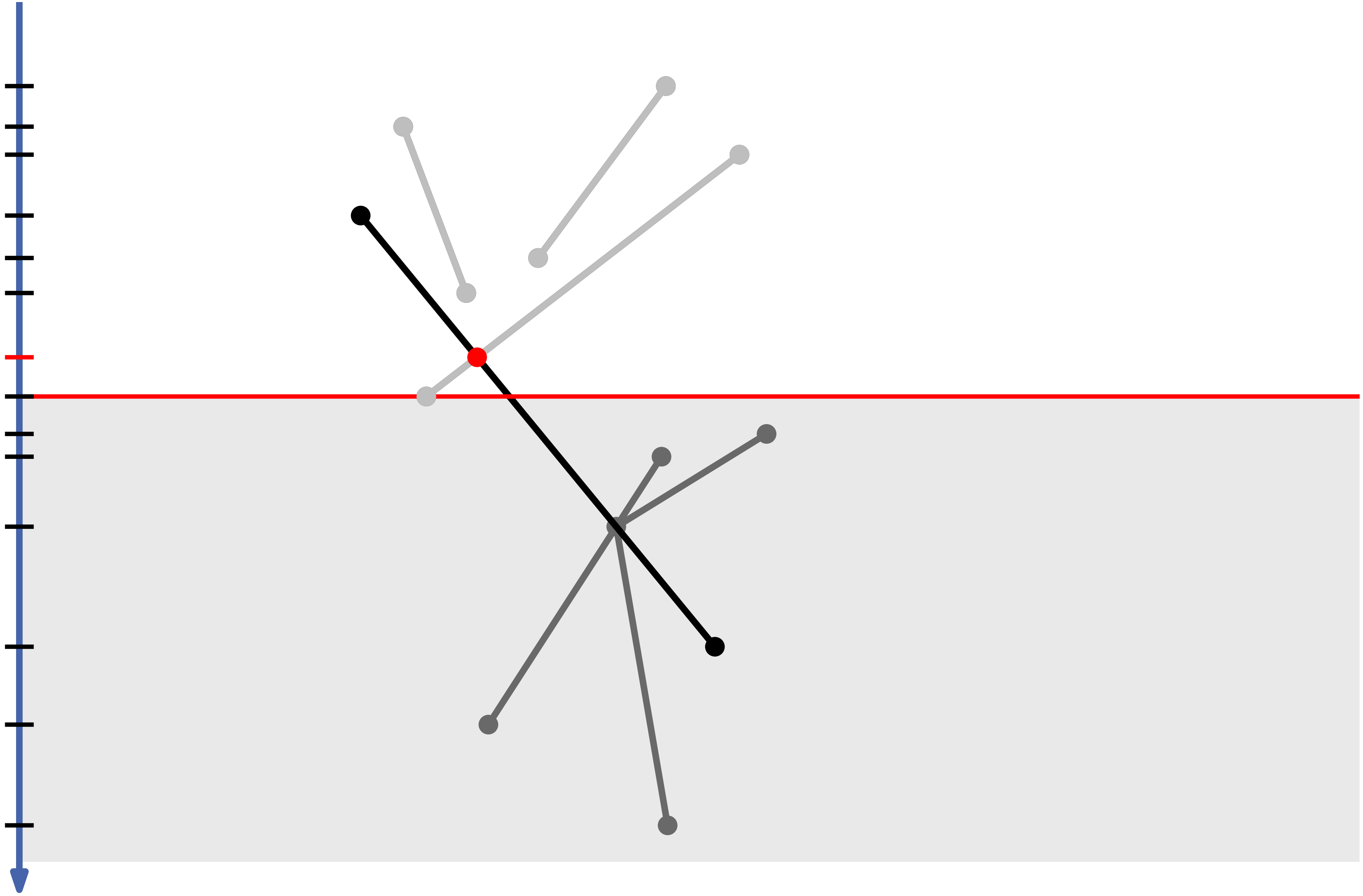
Sweep-Line: Example



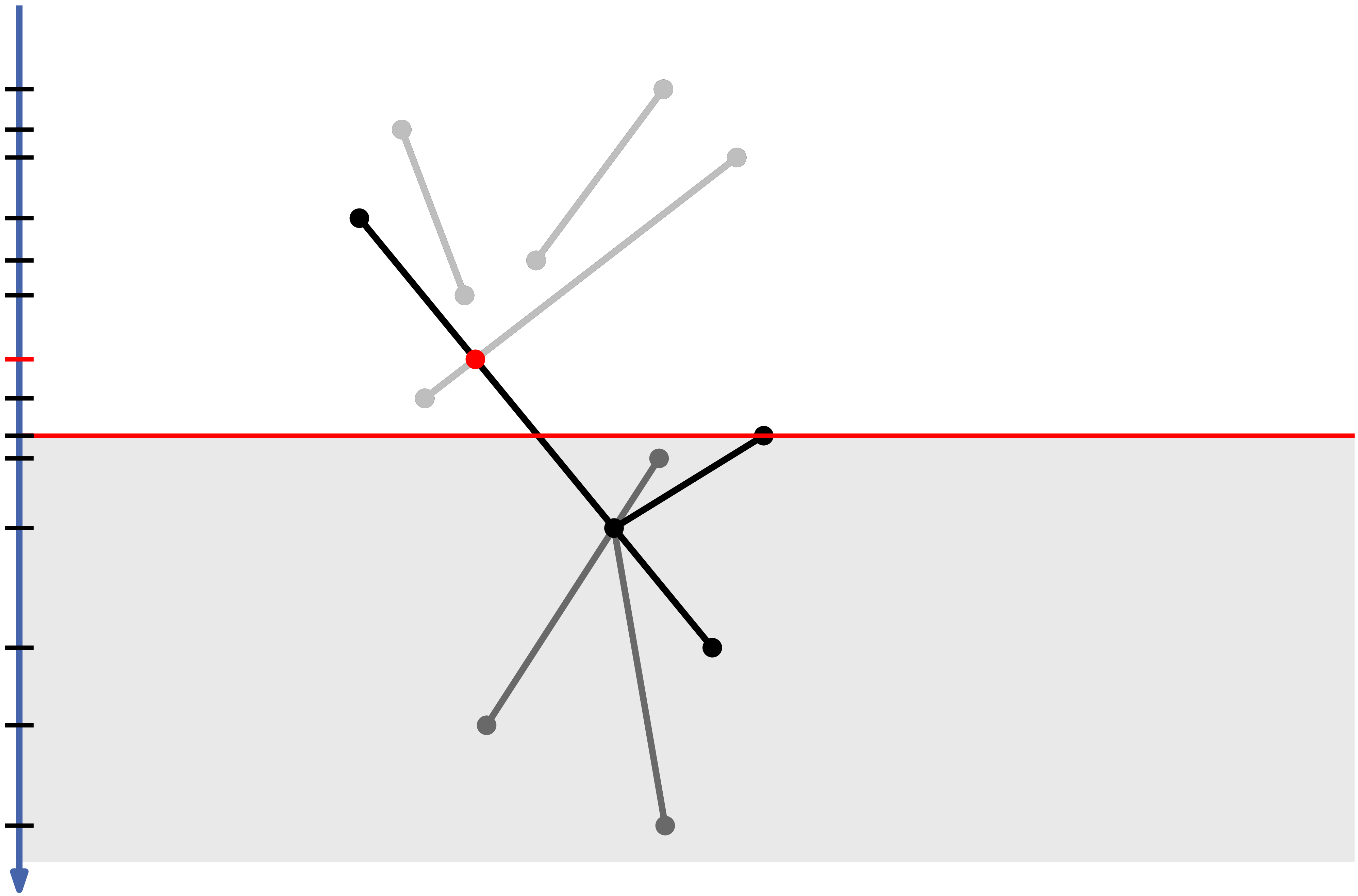
Sweep-Line: Example



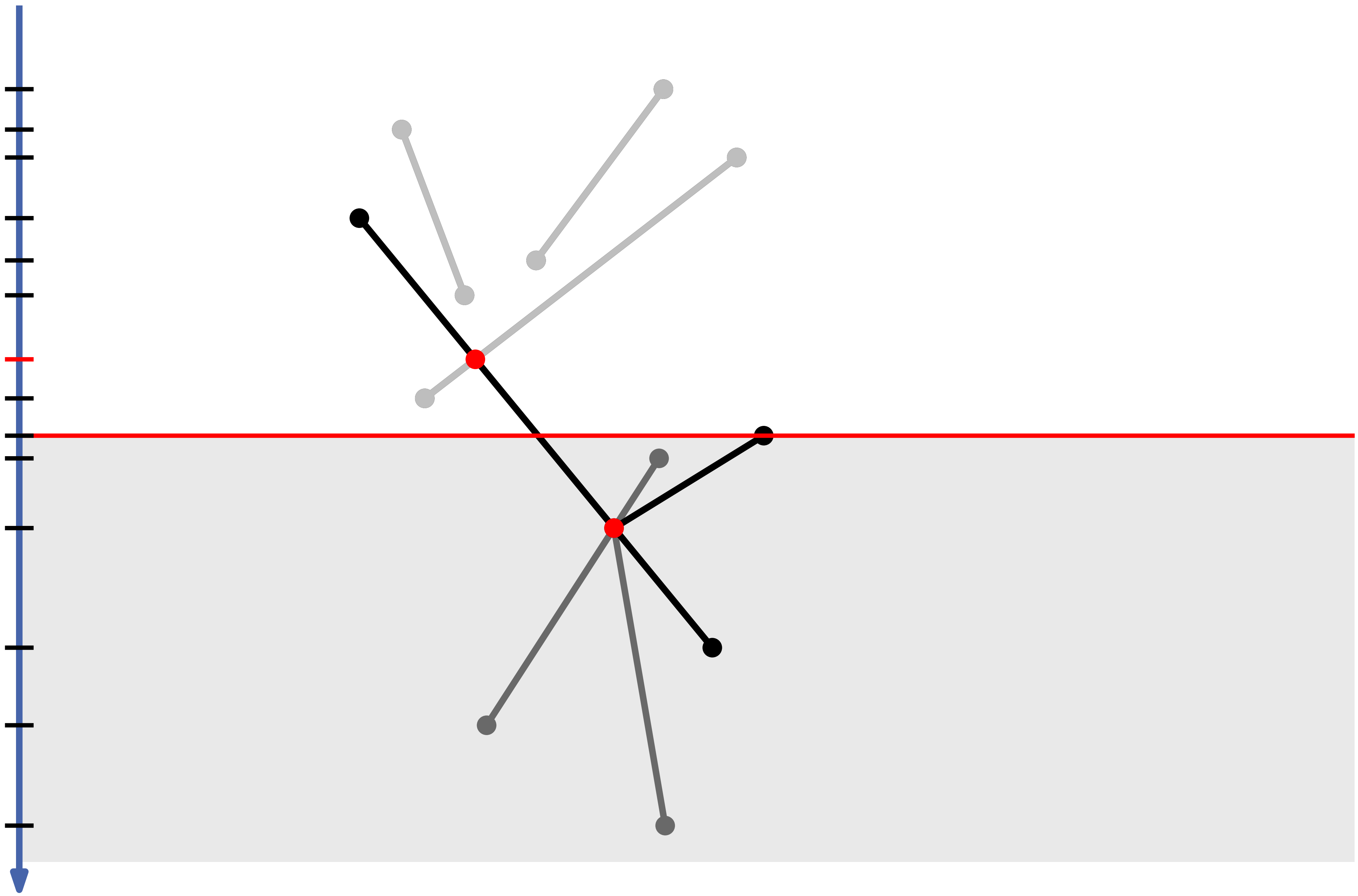
Sweep-Line: Example



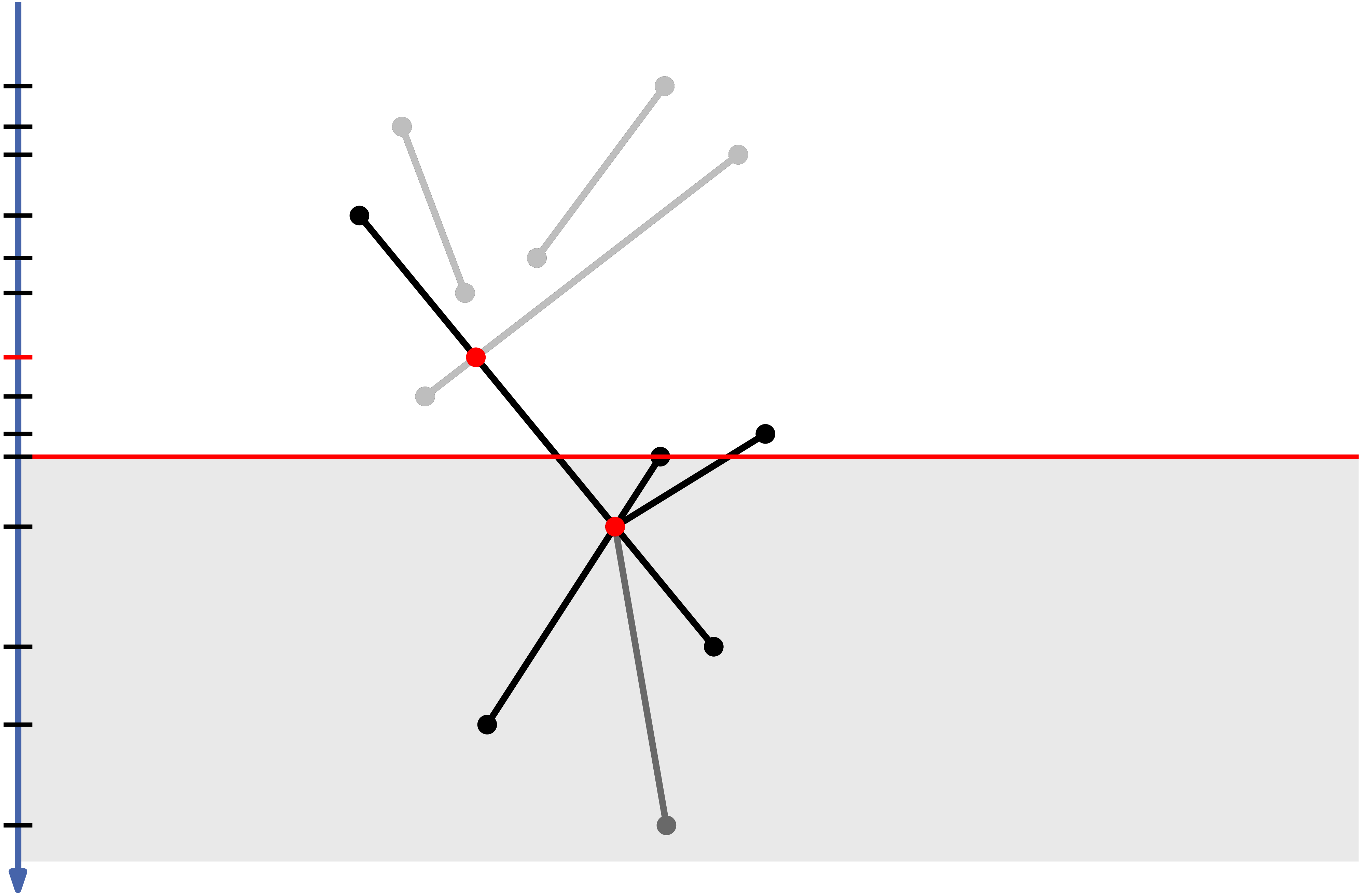
Sweep-Line: Example



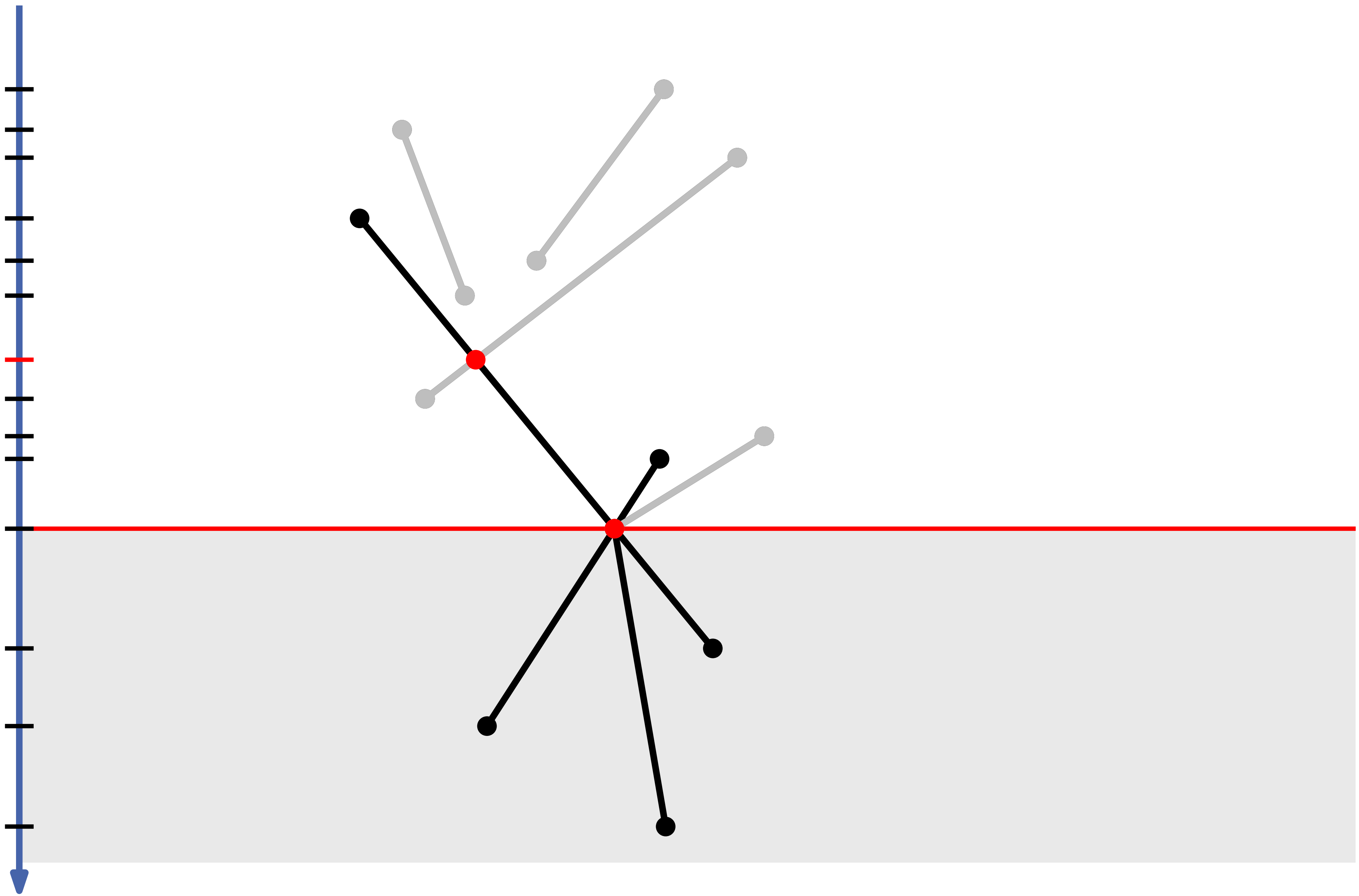
Sweep-Line: Example



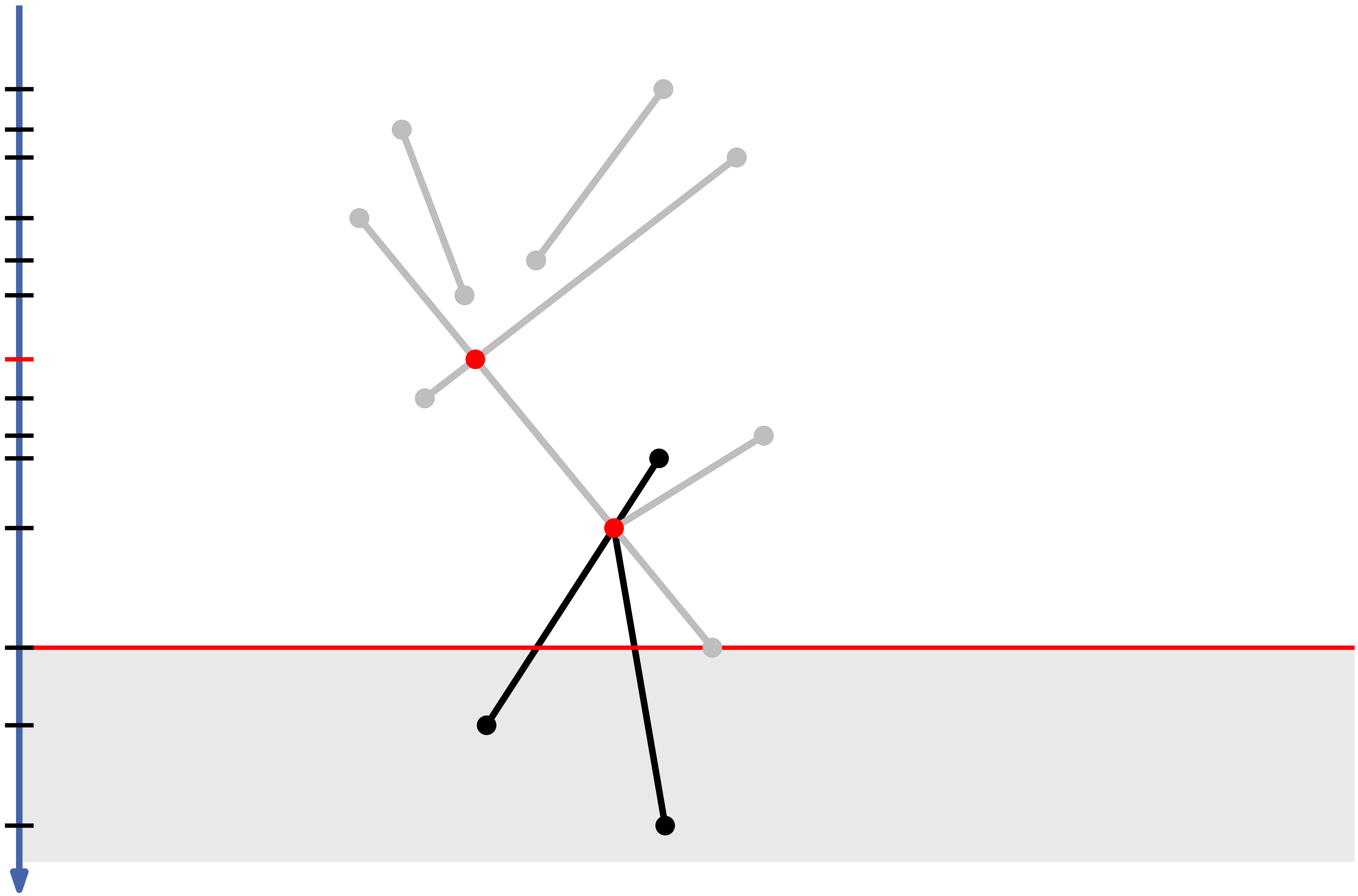
Sweep-Line: Example



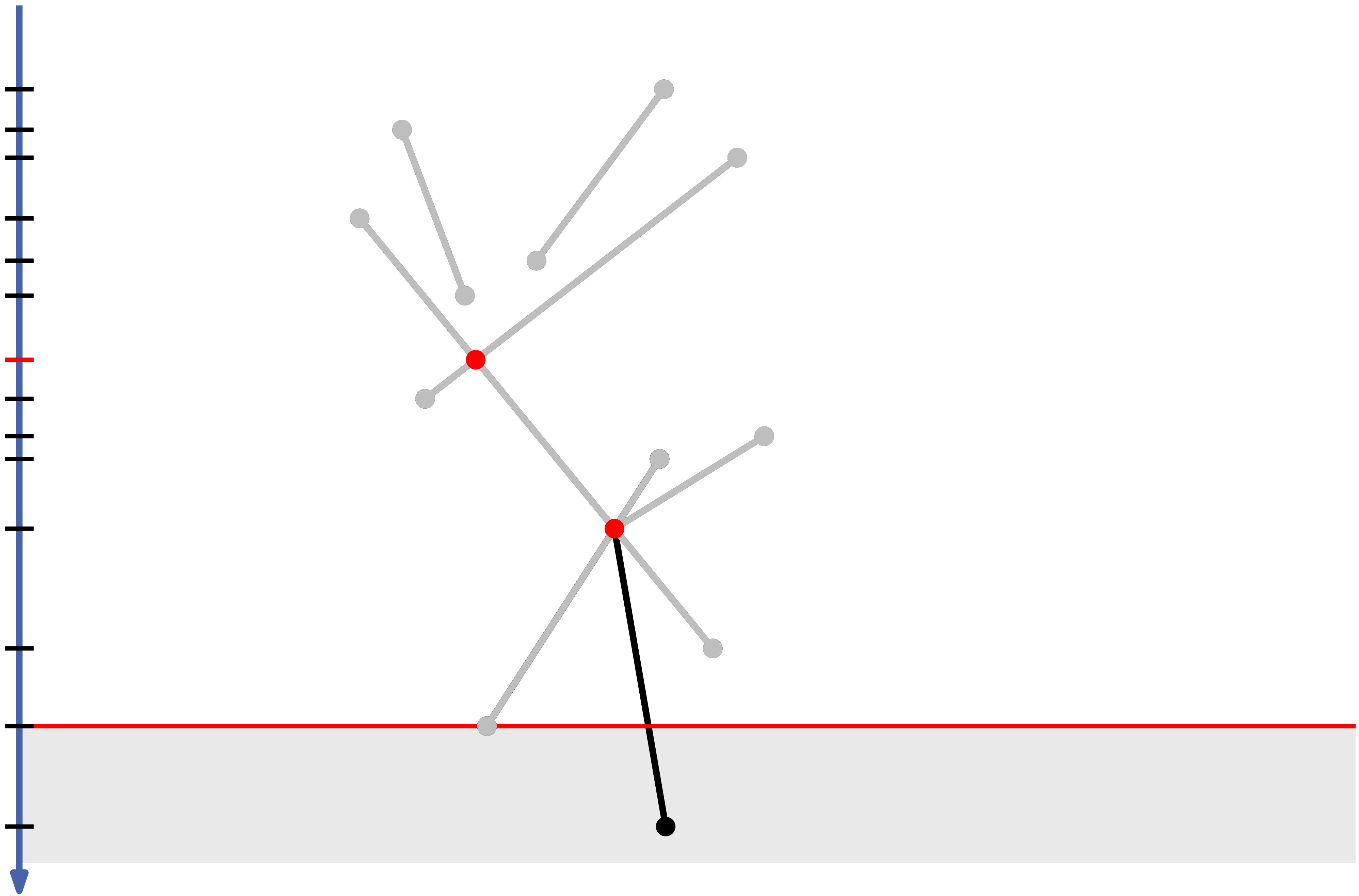
Sweep-Line: Example



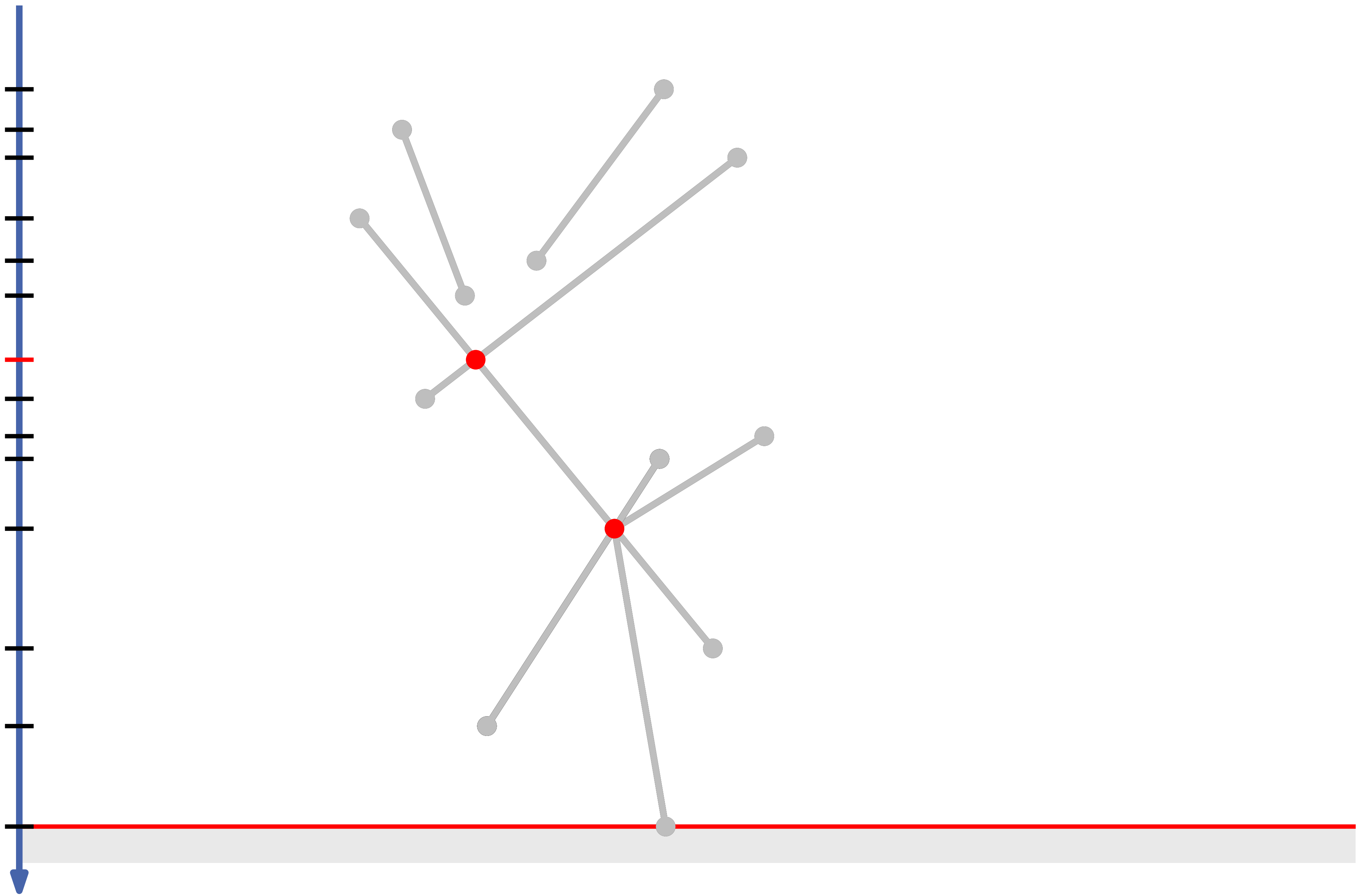
Sweep-Line: Example



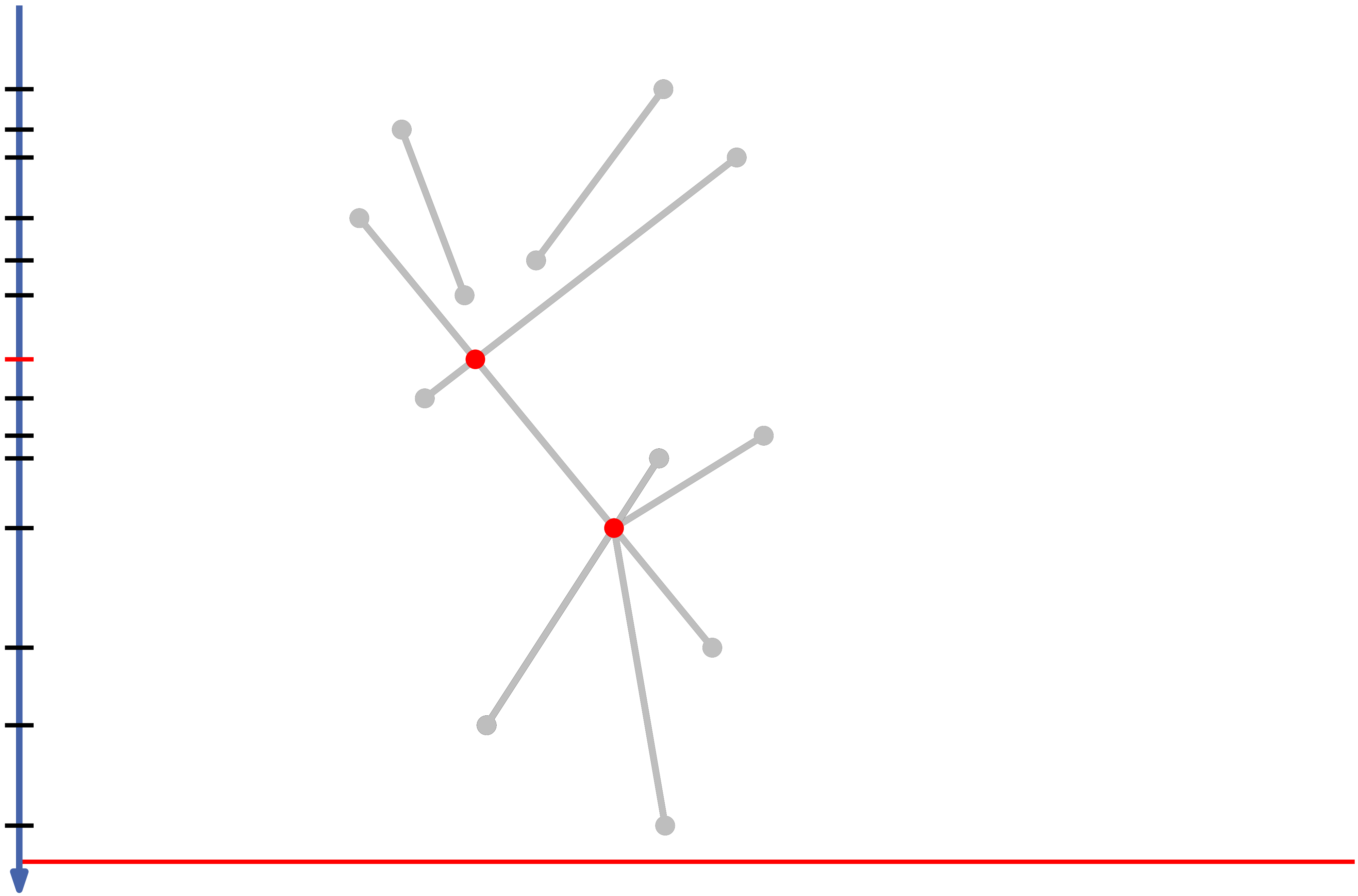
Sweep-Line: Example



Sweep-Line: Example



Sweep-Line: Example

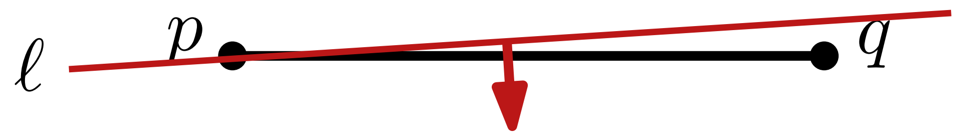


Data Structures

Karlsruhe Institute of Technology

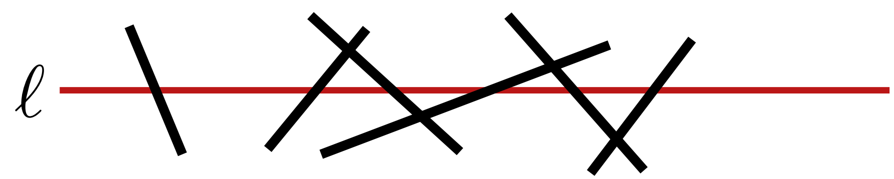
1.) Event Queue $\mathcal{Q}$

- define $p \prec q \quad \Leftrightarrow_{\text{def.}} \quad y_p > y_q \vee (y_p = y_q \wedge x_p < x_q)$



- Store events by $\prec$ in a **balanced binary search tree**
  $\rightarrow$ e.g., AVL tree, red-black tree, . . .
- Operations `insert`, `delete` and `nextEvent` in $O(\log |\mathcal{Q}|)$ time

2.) Sweep-Line Status $\mathcal{T}$



- Stores ℓ cut lines ordered from left to right
- Required operations `insert`, `delete`, `findNeighbor`
- This is also a balanced binary search tree with line segments stored in the leaves!

Algorithm

FindIntersections(S)

Input: Set S of line segments

Output: Set of all intersection points and the line segments involved

$\mathcal{Q} \leftarrow \emptyset; \quad \mathcal{T} \leftarrow \emptyset$

foreach $s \in S$ **do**

- $\mathcal{Q}$.insert(upperEndPoint(s))
- $\mathcal{Q}$.insert(lowerEndPoint(s))

while $\mathcal{Q} \neq \emptyset$ **do**

- $p \leftarrow \mathcal{Q}$.nextEvent()
- $\mathcal{Q}$.deleteEvent(p)
- handleEvent(p)

Algorithm

handleEvent(p)

$U(p) \leftarrow$ Line segments with p as upper endpoint
$L(p) \leftarrow$ Line segments with p as lower endpoint
$C(p) \leftarrow$ Line segments with p as interior point
if $|U(p) \cup L(p) \cup C(p)| > 1$ **then**
    report p and $U(p) \cup L(p) \cup C(p)$
remove $L(p) \cup C(p)$ from $\mathcal{T}$
add $U(p) \cup C(p)$ to $\mathcal{T}$
if $U(p) \cup C(p) = \emptyset$ **then** $//s_l$ and s_r, neighbors of p in $\mathcal{T}$
    $\mathcal{Q} \leftarrow$ check if s_l and s_r intersect below p
else $//s'$ and s'' left- and rightmost line segment in $U(p) \cup C(p)$
    $\mathcal{Q} \leftarrow$ check if s_l and s' intersect below p
    $\mathcal{Q} \leftarrow$ check if s_r and s'' intersect below p

Space Consumption

Karlsruhe Institute of Technology

Lecture:
Running time: $\mathcal{O}((n+I)\log n)$
Storage: $\mathcal{O}(n+I)$

Find:
Find algorithm that needs linear space.

Question:
Which data structure may use more than linear space?

Space Consumption

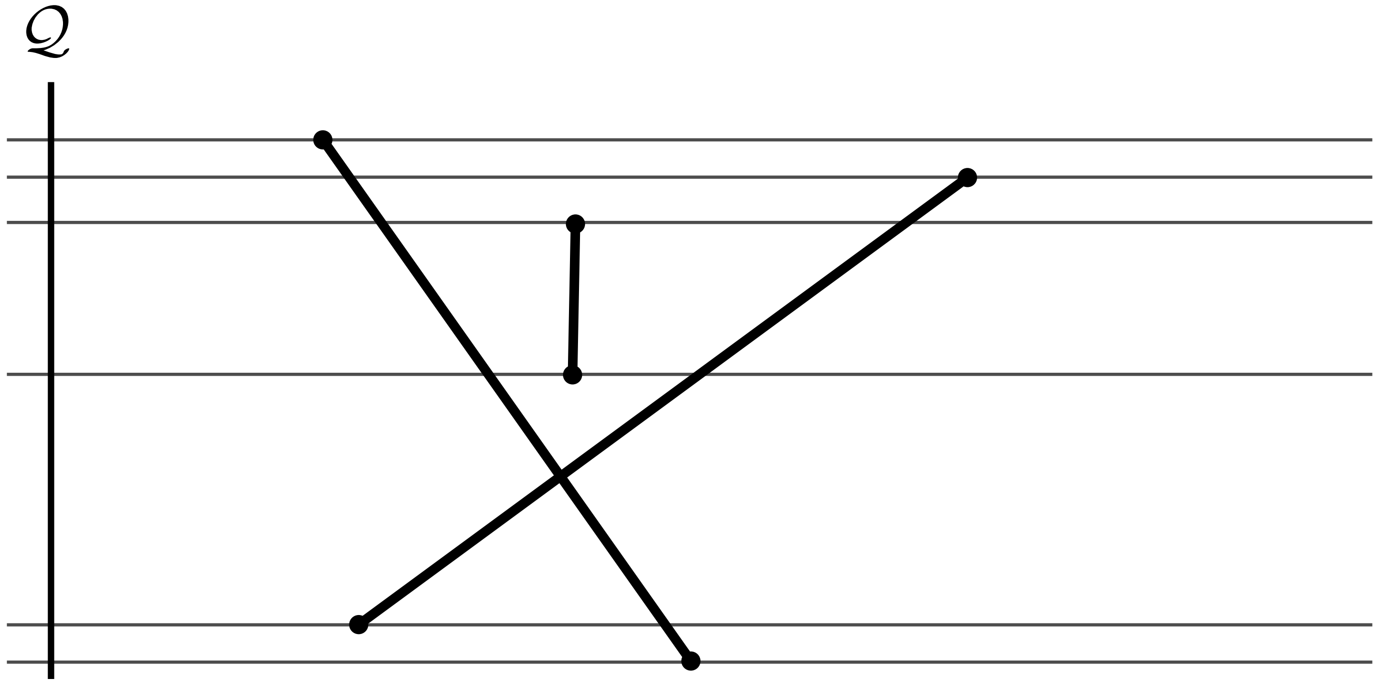
Lecture:
Running time: $\mathcal{O}((n + I) \log n)$
Storage: $\mathcal{O}(n + I)$

Find:
Find algorithm that needs linear space.

Question:
Which data structure may use more than linear space?

Event-Queue may contain $2n + I$ many events,
where $I \in \Omega(n^2)$ in the worst case.

Space Consumption

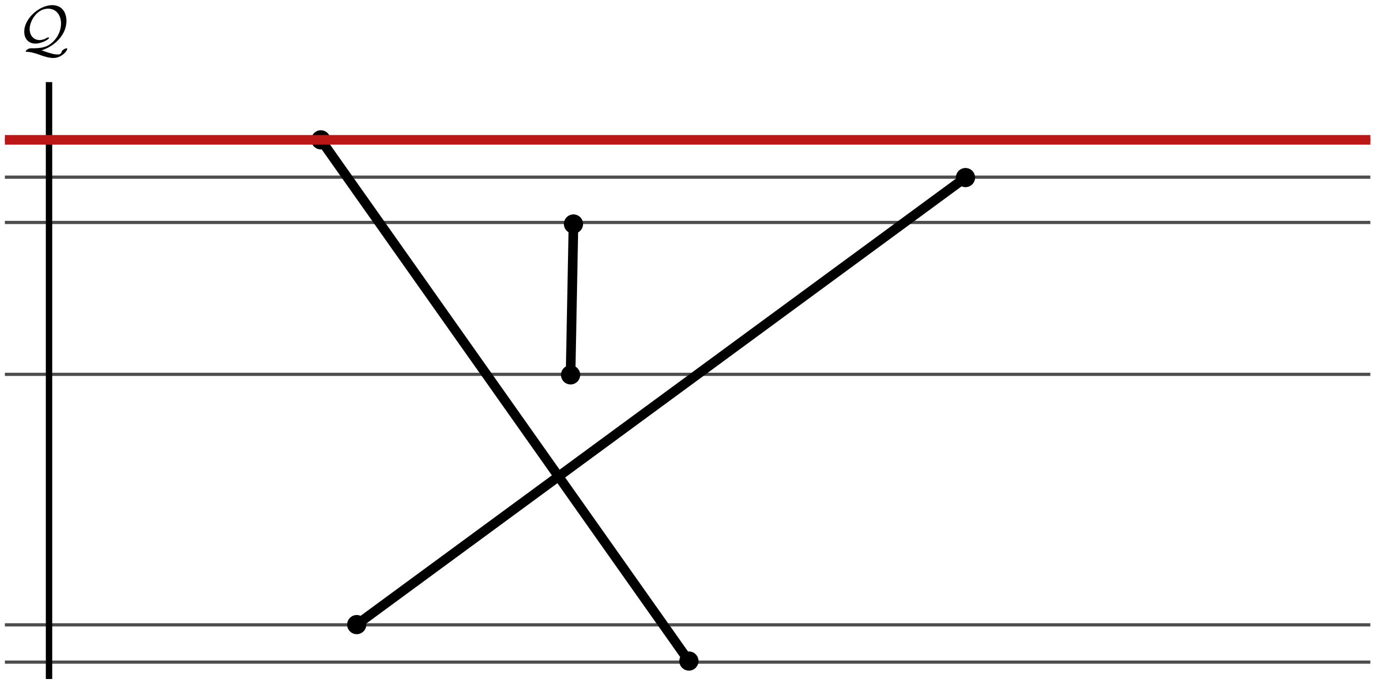


Idea: Store only intersection points that are **currently** adjacent in $\mathcal{T}$.

Obs.: At each point in time there are $O(n)$ many such intersection points.

Procedure: If line segments loose their adjacency in $\mathcal{T}$, remove corresponding intersection points in Q.

Space Consumption

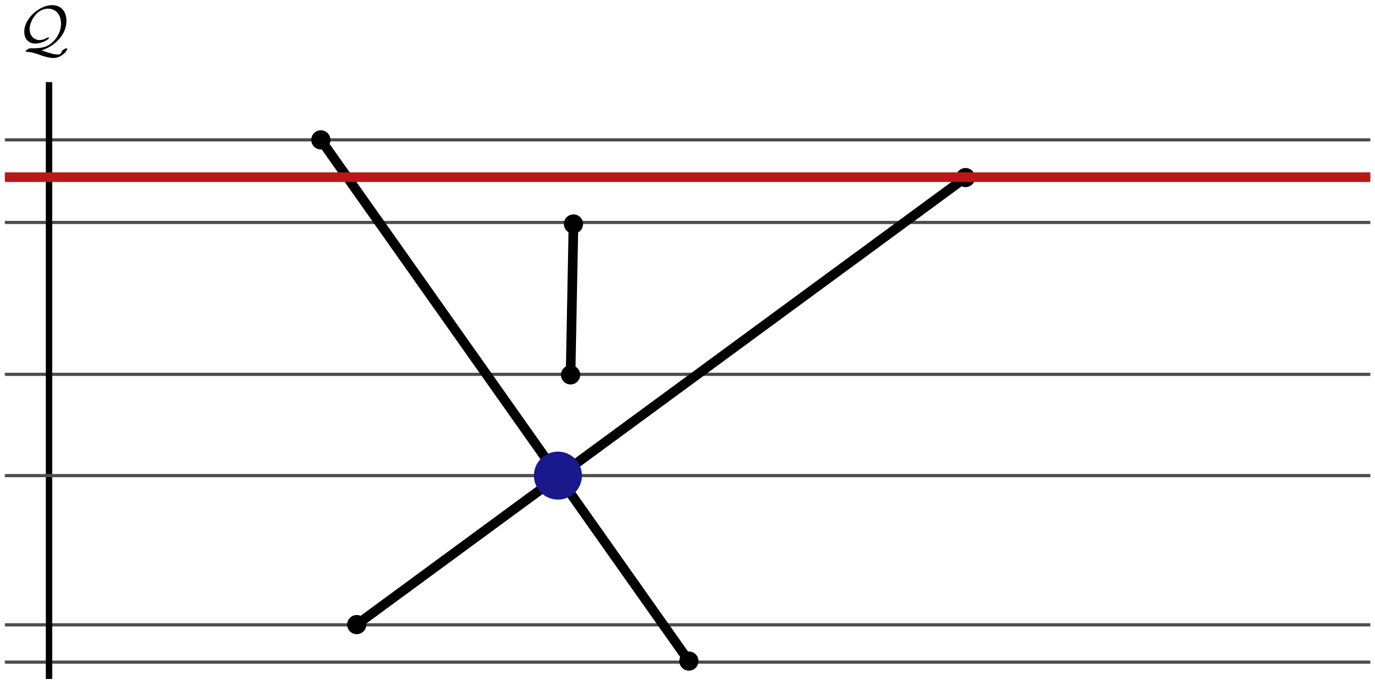


Idea: Store only intersection points that are **currently** adjacent in $\mathcal{T}$.

Obs.: At each point in time there are $O(n)$ many such intersection points.

Procedure: If line segments loose their adjacency in $\mathcal{T}$, remove corresponding intersection points in Q.

Space Consumption

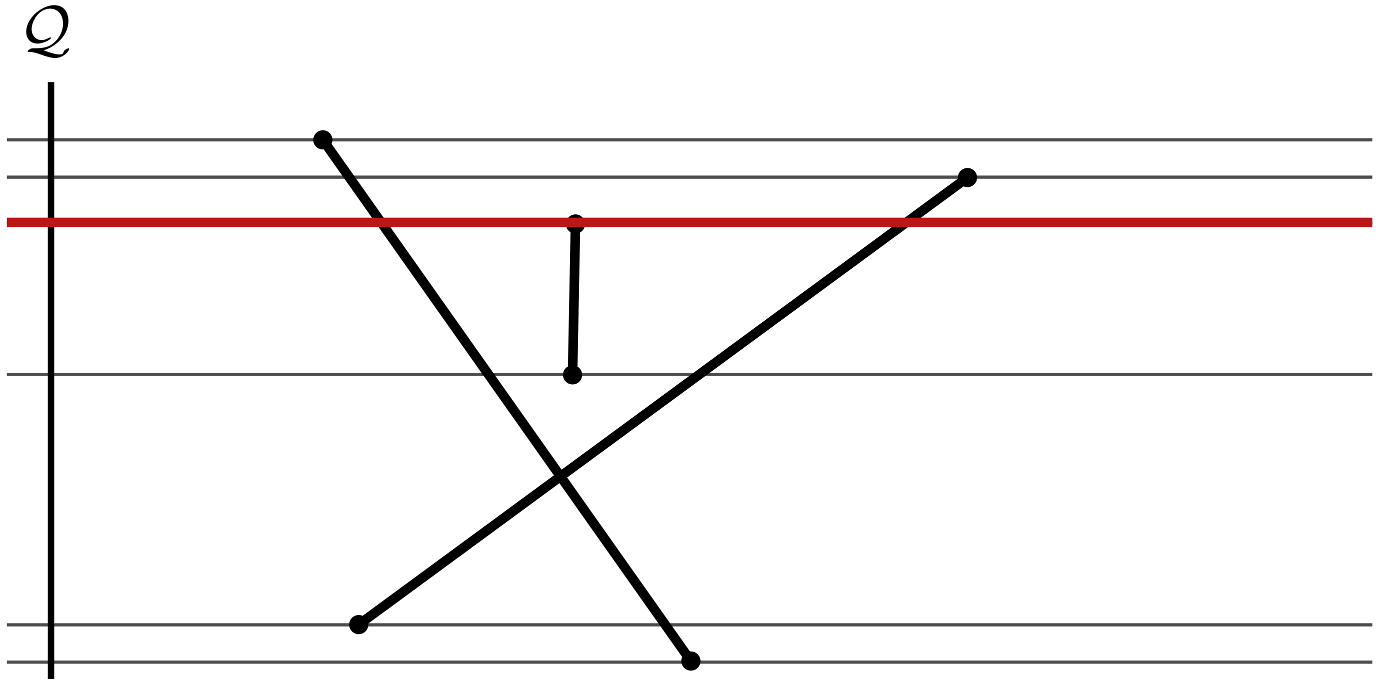


Idea: Store only intersection points that are **currently** adjacent in $\mathcal{T}$.

Obs.: At each point in time there are $O(n)$ many such intersection points.

Procedure: If line segments loose their adjacency in $\mathcal{T}$, remove corresponding intersection points in Q.

Space Consumption

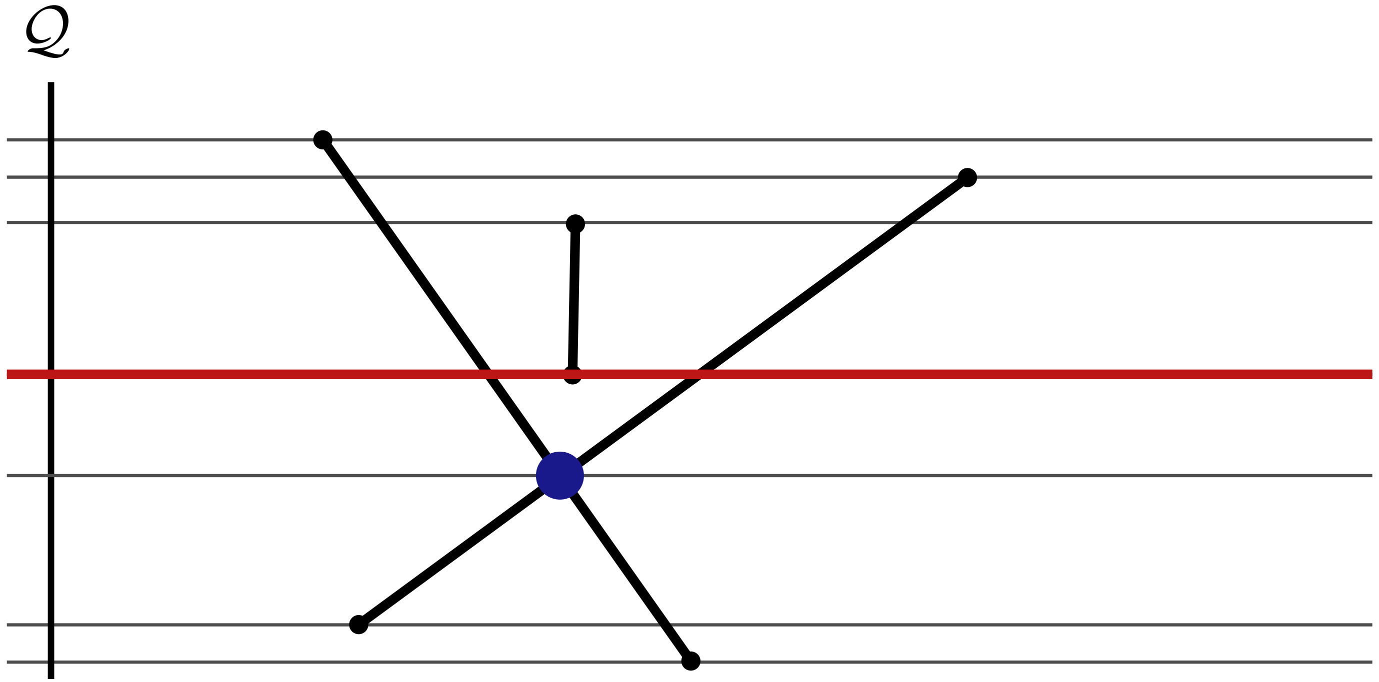


Idea: Store only intersection points that are **currently** adjacent in $\mathcal{T}$.

Obs.: At each point in time there are $O(n)$ many such intersection points.

Procedure: If line segments loose their adjacency in $\mathcal{T}$, remove corresponding intersection points in Q.

Space Consumption



Idea: Store only intersection points that are **currently** adjacent in $\mathcal{T}$.

Obs.: At each point in time there are $O(n)$ many such intersection points.

Procedure: If line segments loose their adjacency in $\mathcal{T}$, remove corresponding intersection points in Q.

Largest Top-Right Region

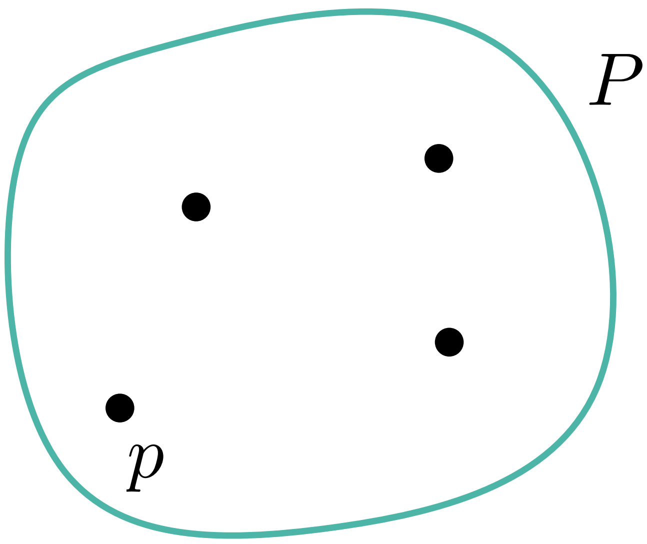
Given: Set P with n points.

Definition:
The *largest top-right region* of a point $p \in P$ is the union of all open axis-aligned squares that touch p with their bottom left corner and contain no other point of P in their interior.

Largest Top-Right Region

Given: Set P with n points.

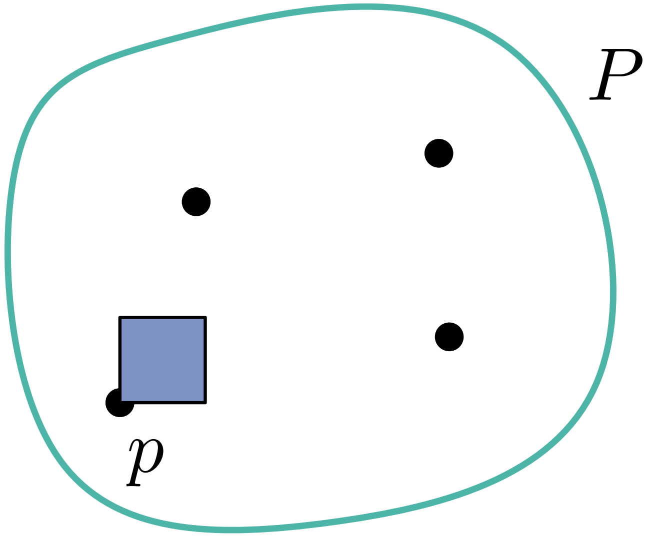
Definition:
The *largest top-right region* of a point $p \in P$ is the union of all open axis-aligned squares that touch p with their bottom left corner and contain no other point of P in their interior.



Largest Top-Right Region

Given: Set P with n points.

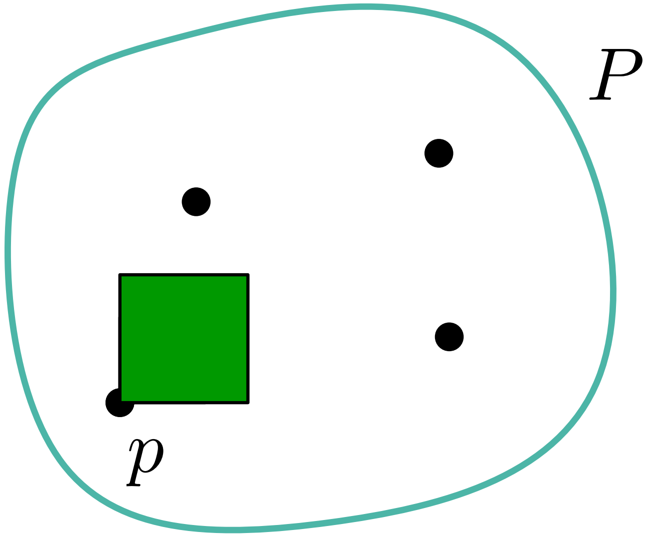
Definition:
The *largest top-right region* of a point $p \in P$ is the union of all open axis-aligned squares that touch p with their bottom left corner and contain no other point of P in their interior.



Largest Top-Right Region

Given: Set P with n points.

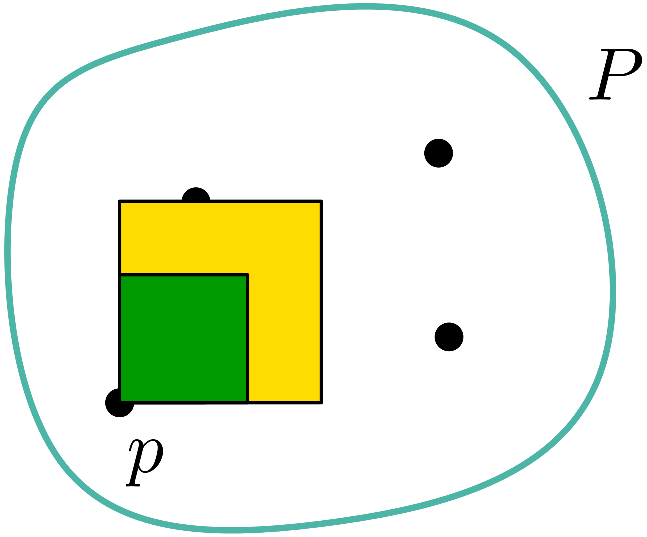
Definition:
The *largest top-right region* of a point $p \in P$ is the union of all open axis-aligned squares that touch p with their bottom left corner and contain no other point of P in their interior.



Largest Top-Right Region

Given: Set P with n points.

Definition:
The *largest top-right region* of a point $p \in P$ is the union of all open axis-aligned squares that touch p with their bottom left corner and contain no other point of P in their interior.

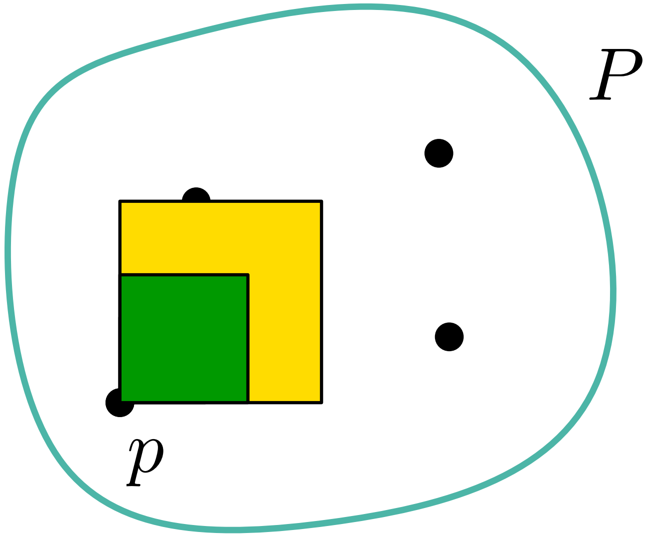


Largest Top-Right Region

Given: Set P with n points.

Definition:
The *largest top-right region* of a point $p \in P$ is the union of all open axis-aligned squares that touch p with their bottom left corner and contain no other point of P in their interior.

a) Prove that the largest top-right region of a point is either a square or the intersection of two open half-planes.



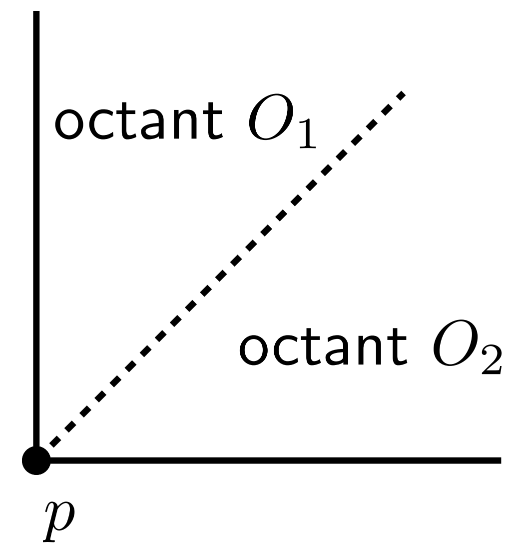
Largest Top-Right Region

Given: Set P with n points.

Definition:
The *largest top-right region* of a point $p \in P$ is the union of all open axis-aligned squares that touch p with their bottom left corner and contain no other point of P in their interior.

a) Prove that the largest top-right region of a point is either a square or the intersection of two open half-planes.

b1) Which point in $O_1 \cap P$ restricts the largest top-right region of p the most?
b2) Which point in $O_2 \cap P$ restricts the largest top-right region of p the most?



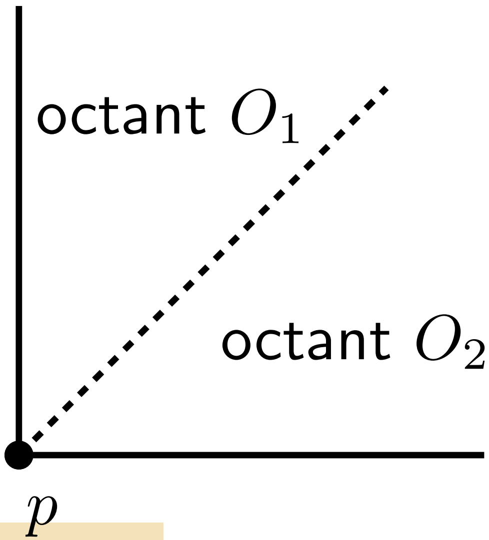
Largest Top-Right Region

Given: Set P with n points.

Definition:
The *largest top-right region* of a point $p \in P$ is the union of all open axis-aligned squares that touch p with their bottom left corner and contain no other point of P in their interior.

a) Prove that the largest top-right region of a point is either a square or the intersection of two open half-planes.

b1) Which point in $O_1 \cap P$ restricts the largest top-right region of p the most?
b2) Which point in $O_2 \cap P$ restricts the largest top-right region of p the most?



$t(p) \in P$: Point in O_1 with smallest vert. distance to p.

$r(p) \in P$: Point in O_2 with smallest horz. distance to p.

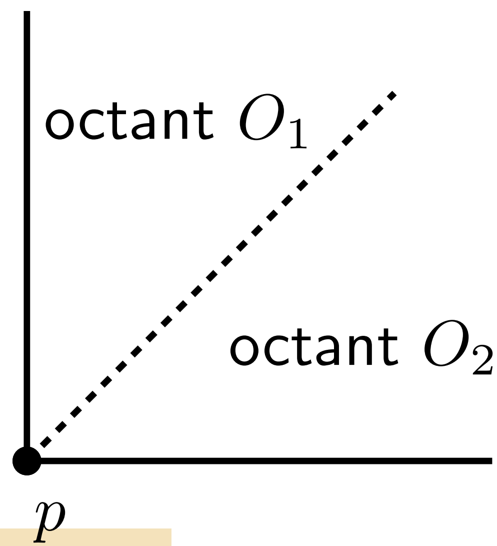
Largest Top-Right Region

Karlsruhe Institute of Technology

Given: Set P with n points.

Definition:
The *largest top-right region* of a point $p \in P$ is the union of all open axis-aligned squares that touch p with their bottom left corner and contain no other point of P in their interior.

a) Prove that the largest top-right region of a point is either a square or the intersection of two open half-planes.



b1) Which point in $O_1 \cap P$ restricts the largest top-right region of p the most?
b2) Which point in $O_2 \cap P$ restricts the largest top-right region of p the most?

$t(p) \in P$: Point in O_1 with smallest vert. distance to p.

$r(p) \in P$: Point in O_2 with smallest horz. distance to p.

c) Largest top-right region for all points in $O(n \log n)$ running time.

Largest Top-Right Region

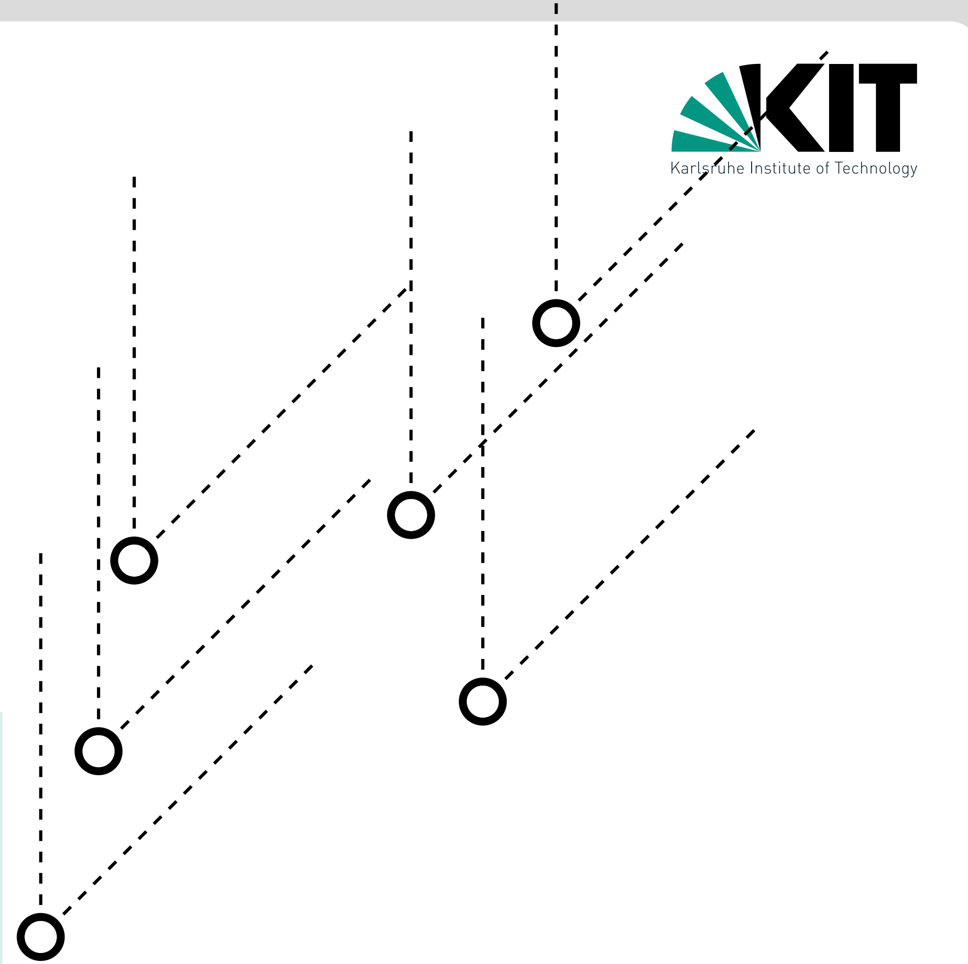
Idea: Determine for each point p the point $t(p)$ (and $r(p)$)

Sweepline: from bottom to top
Events: Points in P

Handling event p

1. Insert p into $\mathcal{T}$.
2. Find point $p' \in \mathcal{T}$ directly left to p:

If p lies in upper octant of p' :
  $t(p') \leftarrow p$, delete p' from $\mathcal{T}$
  repeat step 2



Data structure:
Binary search tree $\mathcal{T}$ over P, where point $p \in \mathcal{T}$, if

1. p lies below the sweep-line
2. $t(p)$ has not been determined yet.

Initially $\mathcal{T}$ is empty and points in $\mathcal{T}$ are sorted w.r.t. their x-coord.

Largest Top-Right Region

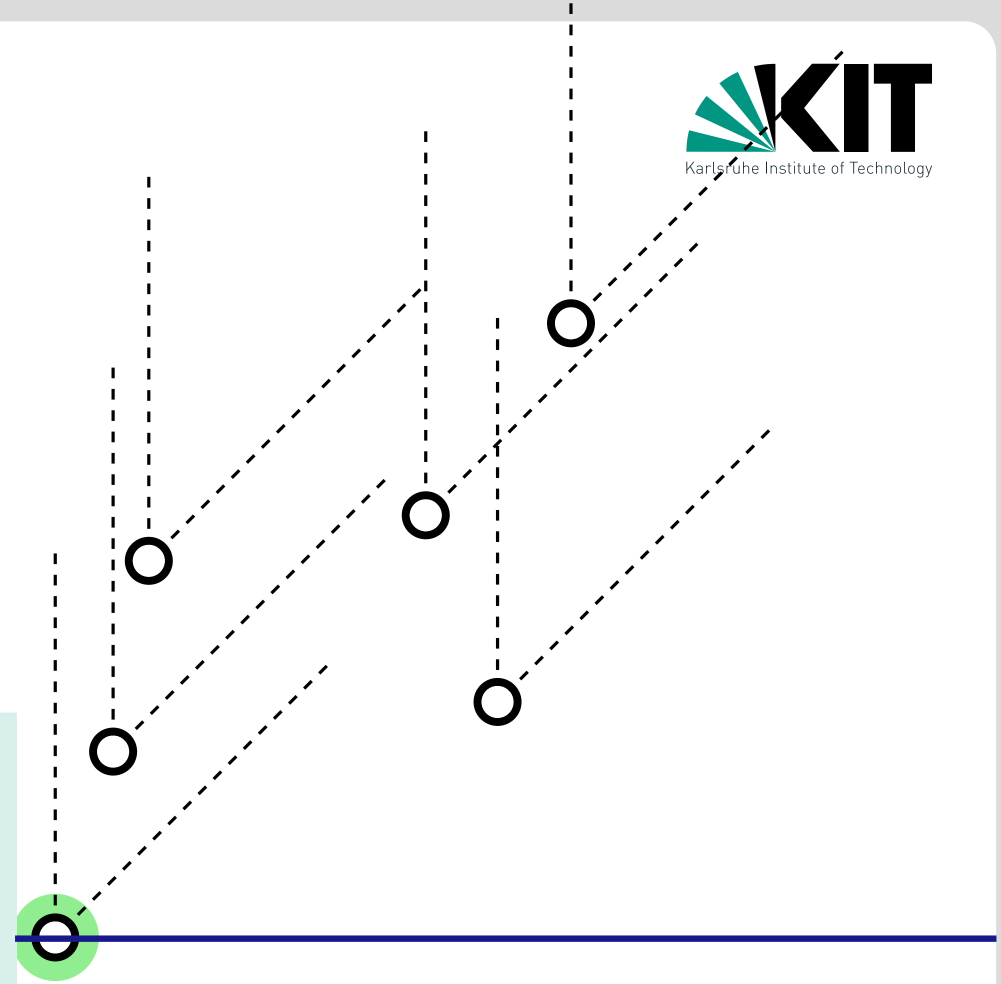
Idea: Determine for each point p the point $t(p)$ (and $r(p)$)

Sweepline: from bottom to top
Events: Points in P

Handling event p

1. Insert p into $\mathcal{T}$.
2. Find point $p' \in \mathcal{T}$ directly left to p:

If p lies in upper octant of p' :
  $t(p') \leftarrow p$, delete p' from $\mathcal{T}$
  repeat step 2



= contained in $\mathcal{T}$

Data structure:
Binary search tree $\mathcal{T}$ over P, where point $p \in \mathcal{T}$, if

1. p lies below the sweep-line
2. $t(p)$ has not been determined yet.

Initially $\mathcal{T}$ is empty and points in $\mathcal{T}$ are sorted w.r.t. their x-coord.

Largest Top-Right Region

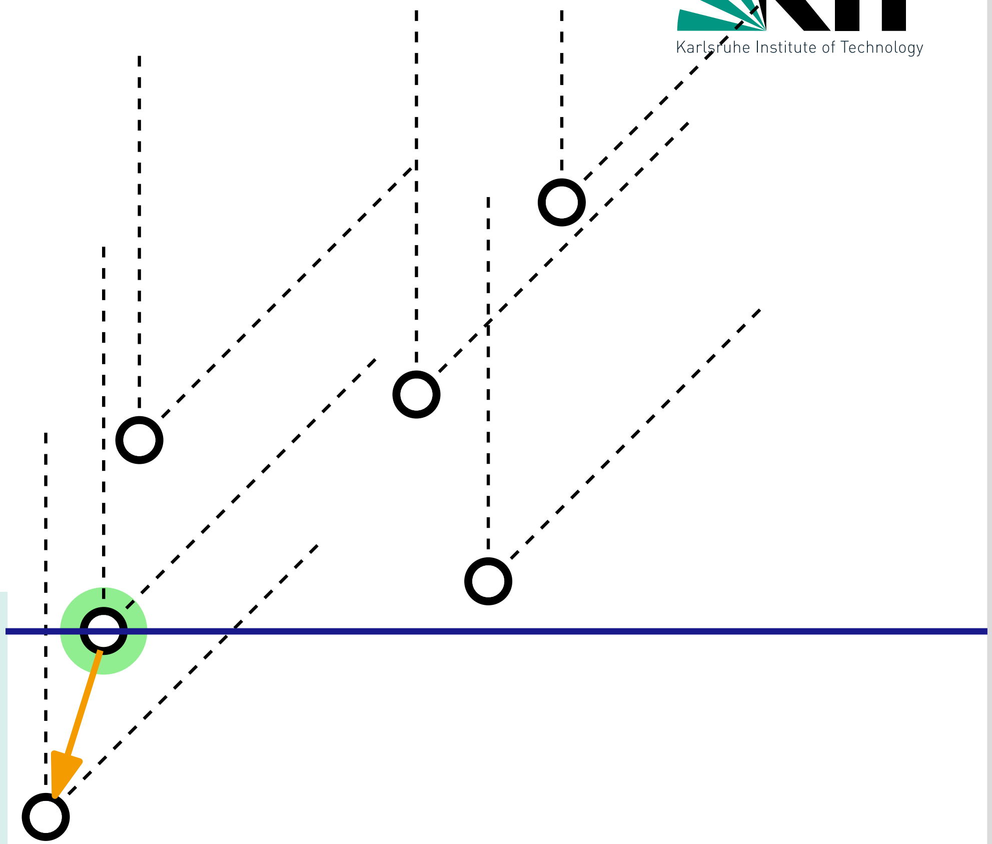
Idea: Determine for each point p the point $t(p)$ (and $r(p)$)

Sweepline: from bottom to top
Events: Points in P

Handling event p

1. Insert p into $\mathcal{T}$.
2. Find point $p' \in \mathcal{T}$ directly left to p:

If p lies in upper octant of p' :
 $t(p') \leftarrow p$, delete p' from $\mathcal{T}$
 repeat step 2



$=$ contained in $\mathcal{T}$

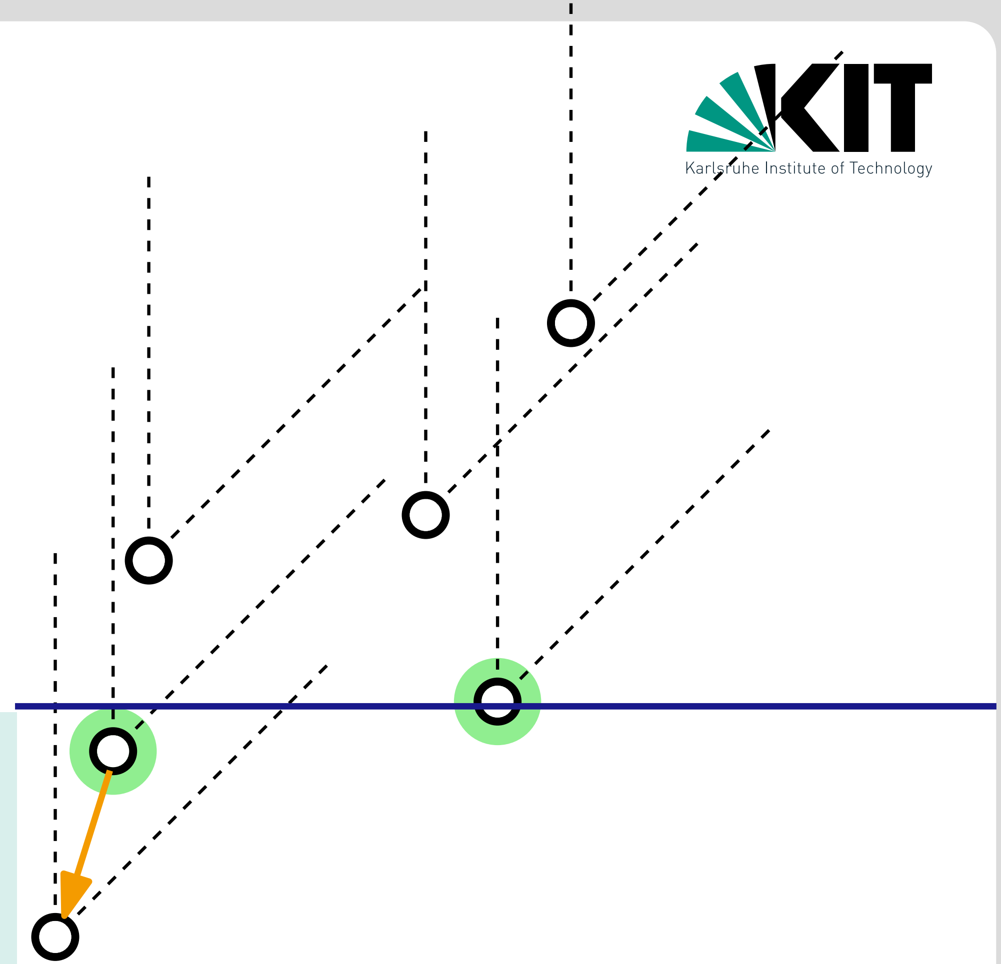
Data structure:
Binary search tree $\mathcal{T}$ over P, where point $p \in \mathcal{T}$, if

1. p lies below the sweep-line
2. $t(p)$ has not been determined yet.

Initially $\mathcal{T}$ is empty and points in $\mathcal{T}$ are sorted w.r.t. their x-coord.

Largest Top-Right Region

Idea: Determine for each point p the point $t(p)$ (and $r(p)$)



Sweepline: from bottom to top

Events: Points in P

Handling event p

1. Insert p into $\mathcal{T}$.
2. Find point $p' \in \mathcal{T}$ directly left to p:

If p lies in upper octant of p' :
- $t(p') \leftarrow p$, delete p' from $\mathcal{T}$
- repeat step 2

= contained in $\mathcal{T}$

Data structure:

Binary search tree $\mathcal{T}$ over P, where point $p \in \mathcal{T}$, if

1. p lies below the sweep-line
2. $t(p)$ has not been determined yet.

Initially $\mathcal{T}$ is empty and points in $\mathcal{T}$ are sorted w.r.t. their x-coord.

Largest Top-Right Region

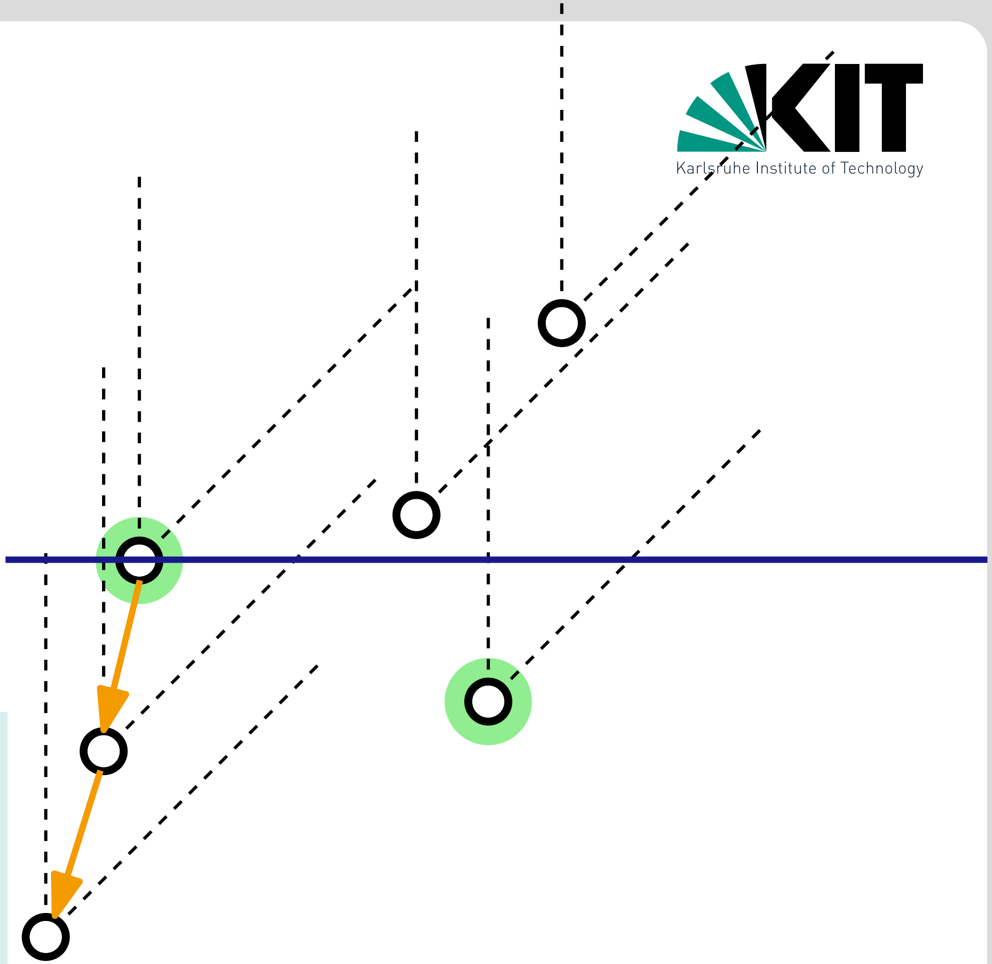
Idea: Determine for each point p the point $t(p)$ (and $r(p)$)

Sweepline: from bottom to top
Events: Points in P

Handling event p

1. Insert p into $\mathcal{T}$.
2. Find point $p' \in \mathcal{T}$ directly left to p:

If p lies in upper octant of p' :
  $t(p') \leftarrow p$, delete p' from $\mathcal{T}$
  repeat step 2



= contained in $\mathcal{T}$

Data structure:
Binary search tree $\mathcal{T}$ over P, where point $p \in \mathcal{T}$, if

1. p lies below the sweep-line
2. $t(p)$ has not been determined yet.

Initially $\mathcal{T}$ is empty and points in $\mathcal{T}$ are sorted w.r.t. their x-coord.

Largest Top-Right Region

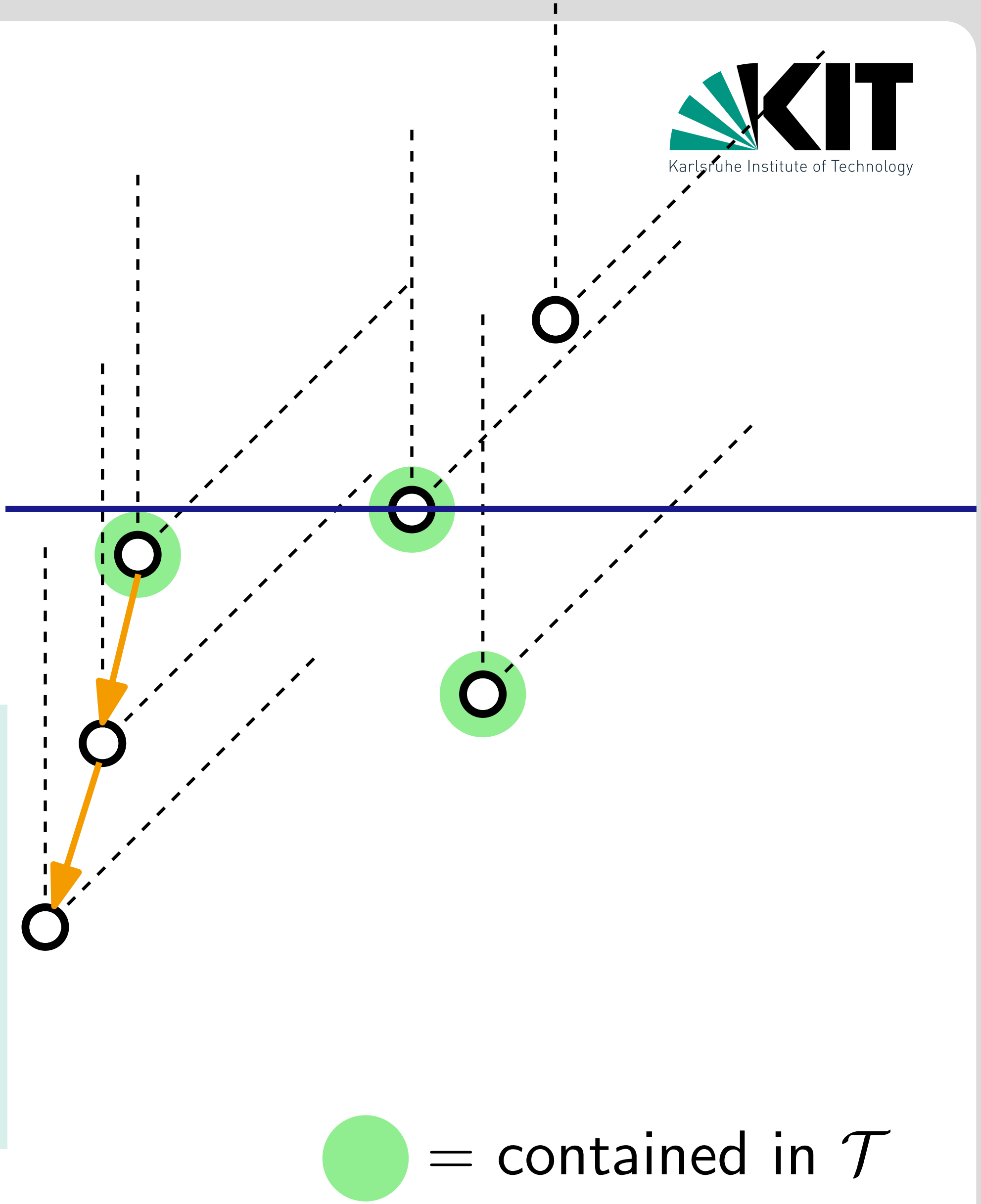
Idea: Determine for each point p the point $t(p)$ (and $r(p)$)

Sweepline: from bottom to top
Events: Points in P

Handling event p

1. Insert p into $\mathcal{T}$.
2. Find point $p' \in \mathcal{T}$ directly left to p:

If p lies in upper octant of p' :
  $t(p') \leftarrow p$, delete p' from $\mathcal{T}$
  repeat step 2



Data structure:
Binary search tree $\mathcal{T}$ over P, where point $p \in \mathcal{T}$, if

1. p lies below the sweep-line
2. $t(p)$ has not been determined yet.

Initially $\mathcal{T}$ is empty and points in $\mathcal{T}$ are sorted w.r.t. their x-coord.

Largest Top-Right Region

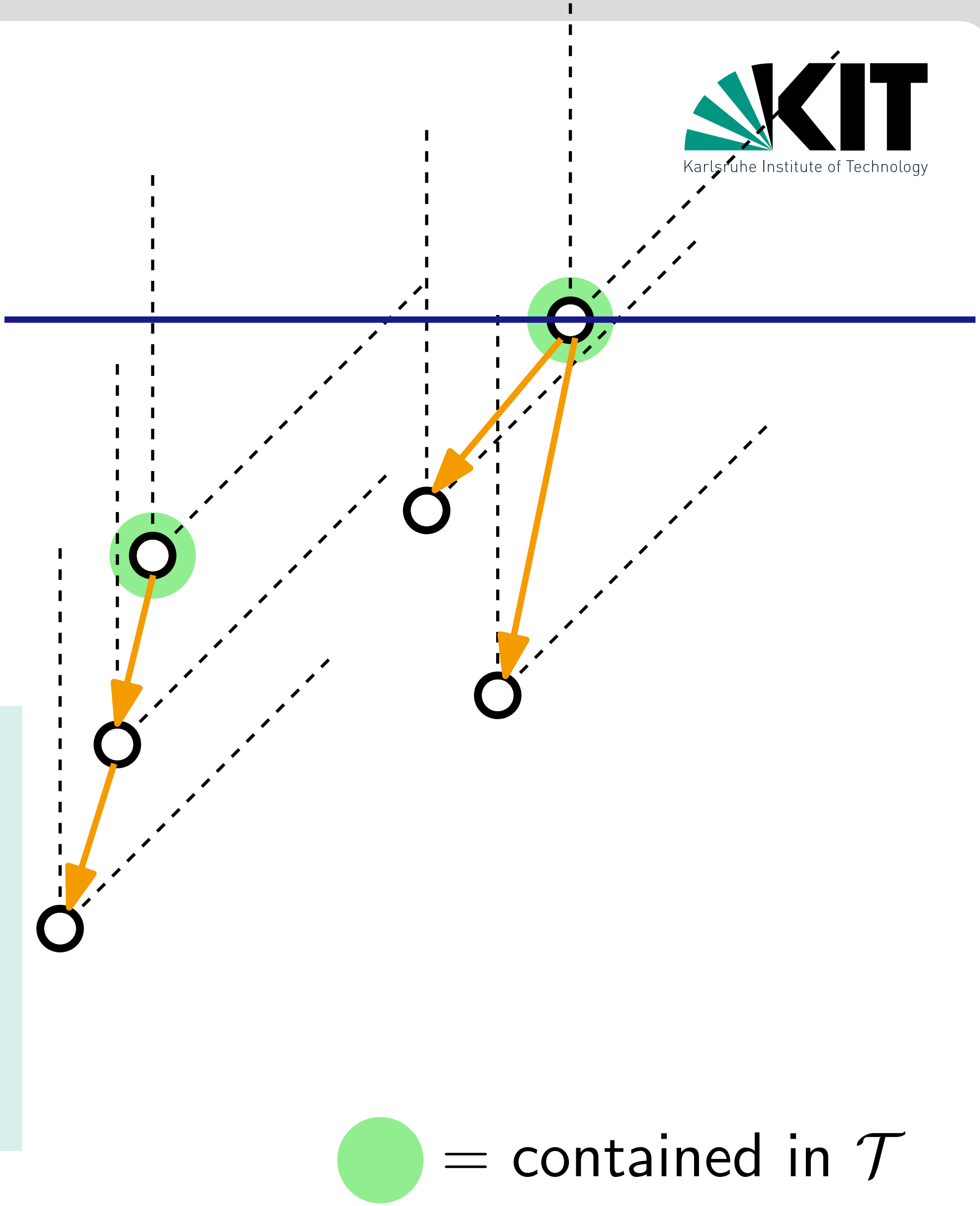
Idea: Determine for each point p the point $t(p)$ (and $r(p)$)

Sweepline: from bottom to top
Events: Points in P

Handling event p

1. Insert p into $\mathcal{T}$.
2. Find point $p' \in \mathcal{T}$ directly left to p:

If p lies in upper octant of p' :
  $t(p') \leftarrow p$, delete p' from $\mathcal{T}$
  repeat step 2



Data structure:
Binary search tree $\mathcal{T}$ over P, where point $p \in \mathcal{T}$, if

1. p lies below the sweep-line
2. $t(p)$ has not been determined yet.

Initially $\mathcal{T}$ is empty and points in $\mathcal{T}$ are sorted w.r.t. their x-coord.

Subdivision of plane.

Quelle: Google Earth

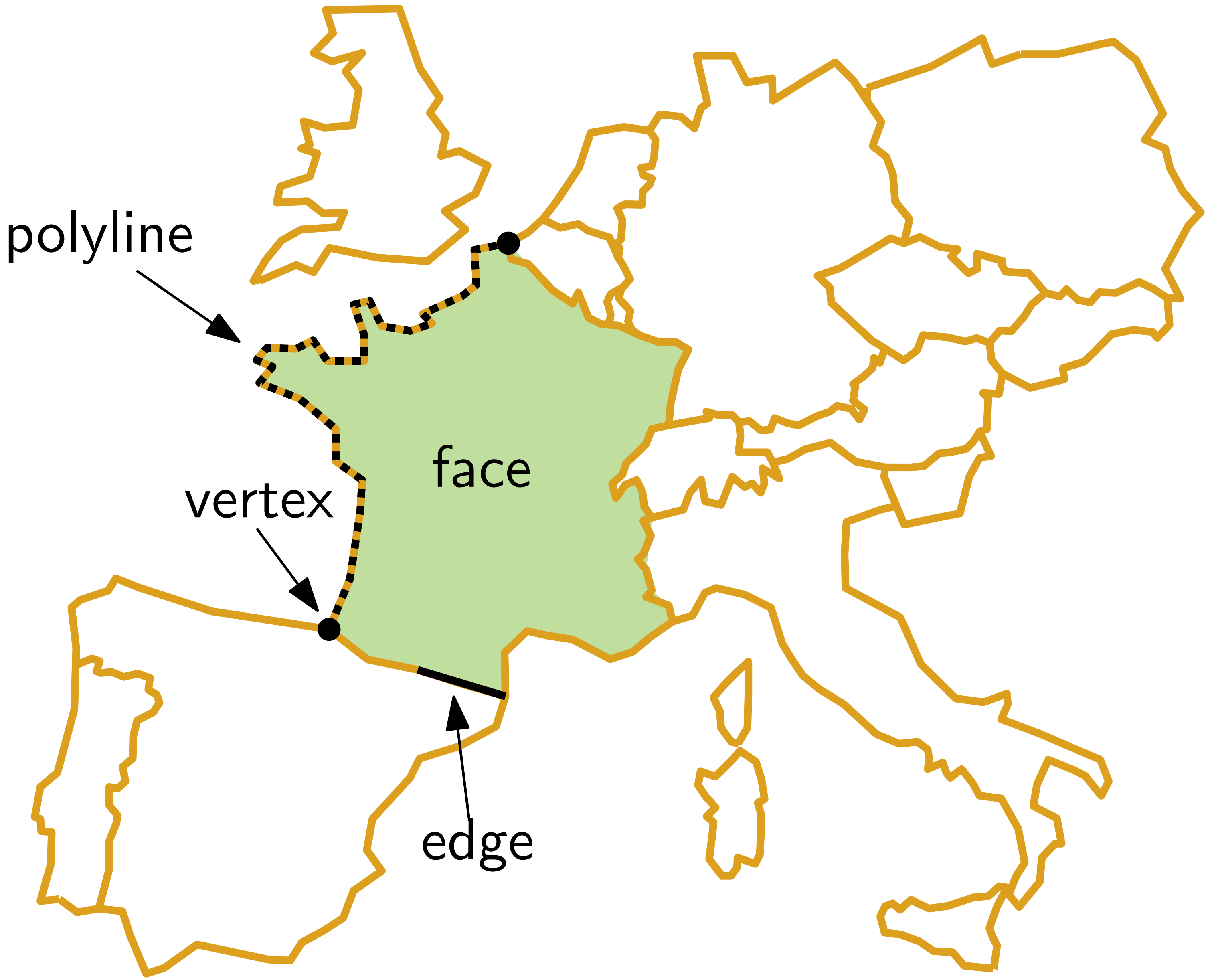
Subdivision of plane.

Quelle: Google Earth

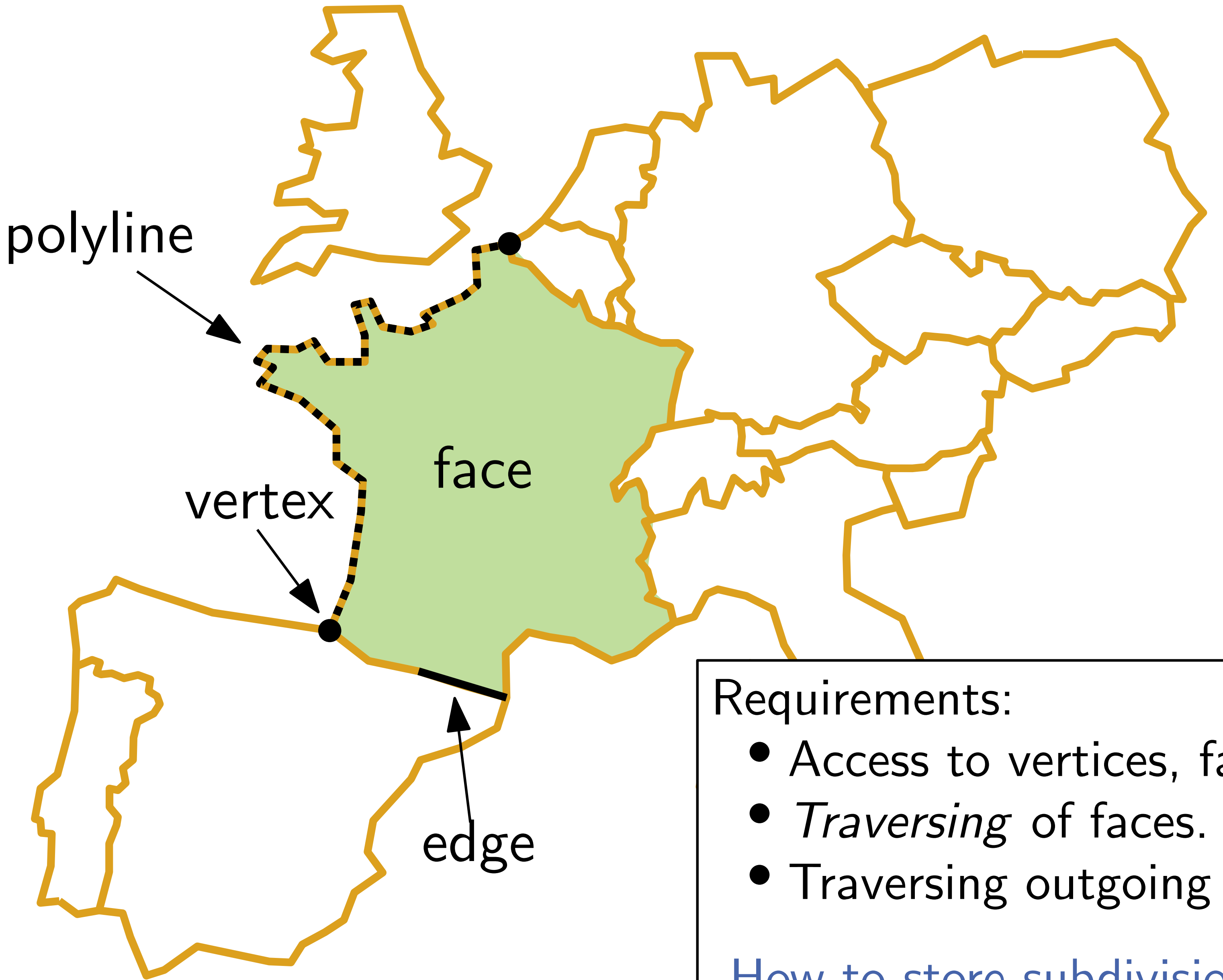
Subdivision of plane.

Subdivision of plane.

Subdivision of plane.



Subdivision of plane.

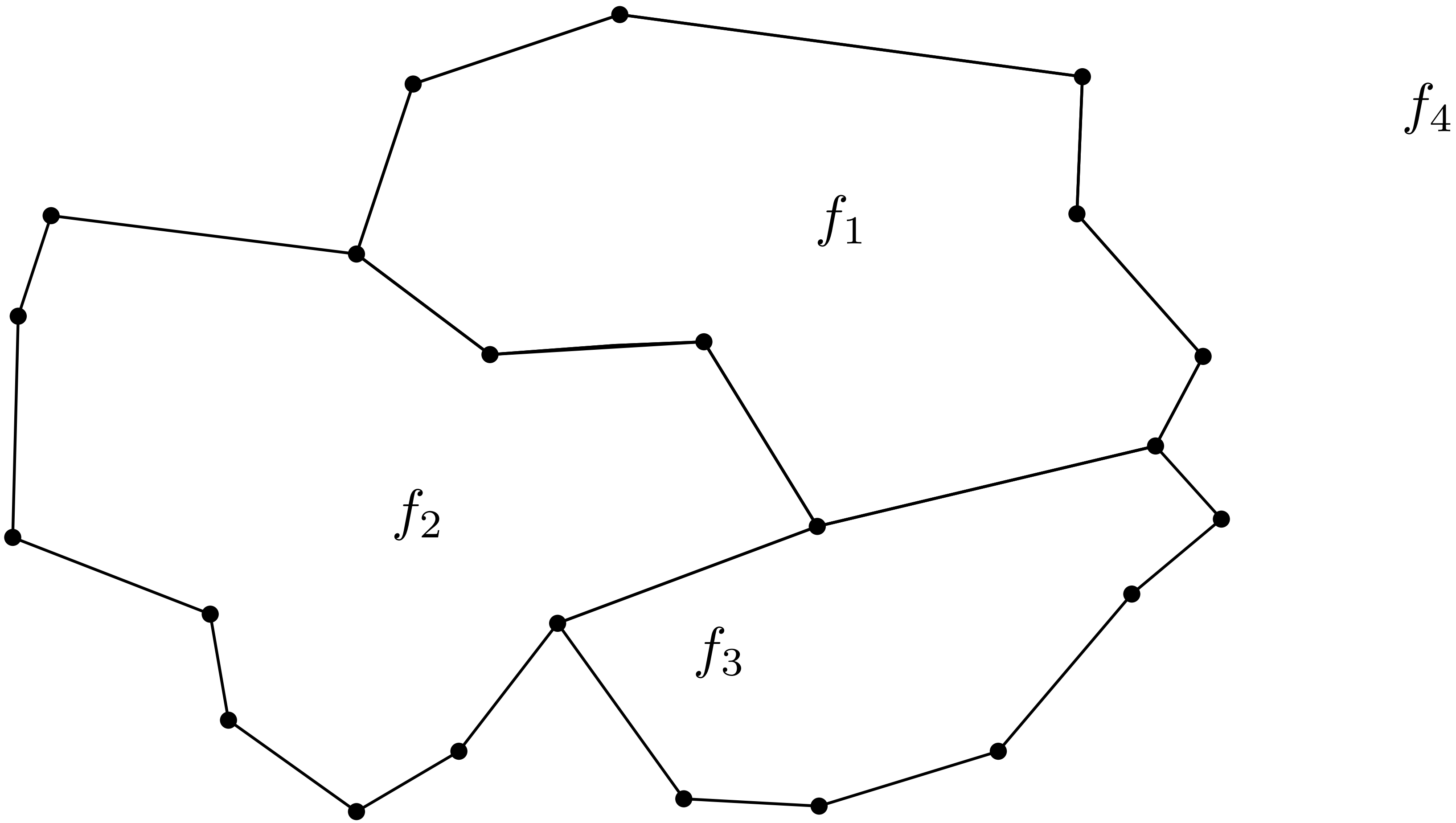


Requirements:

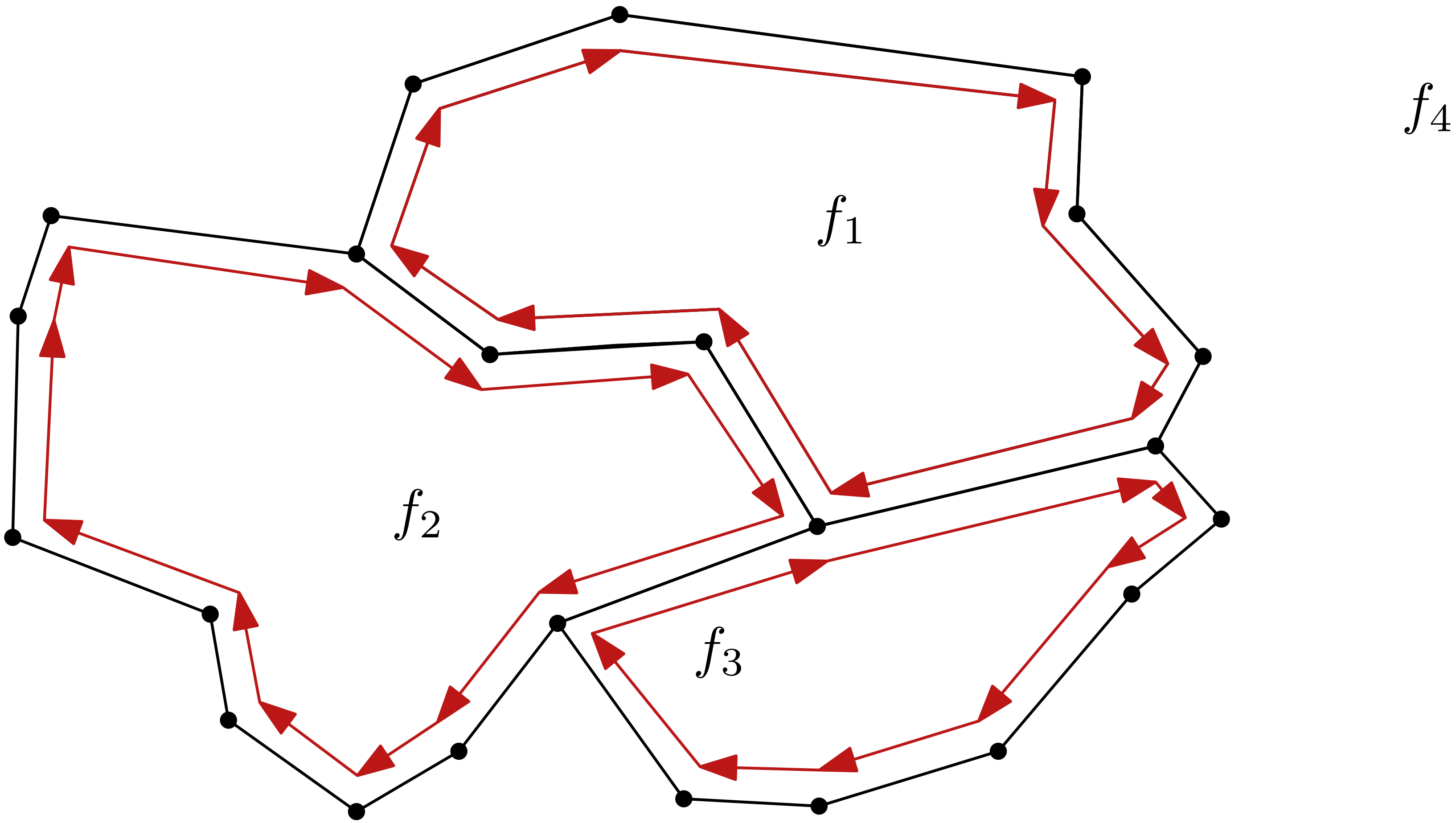
- Access to vertices, faces and edges.
- *Traversing* of faces.
- Traversing outgoing edges.

How to store subdivision efficiently?

Subdivision

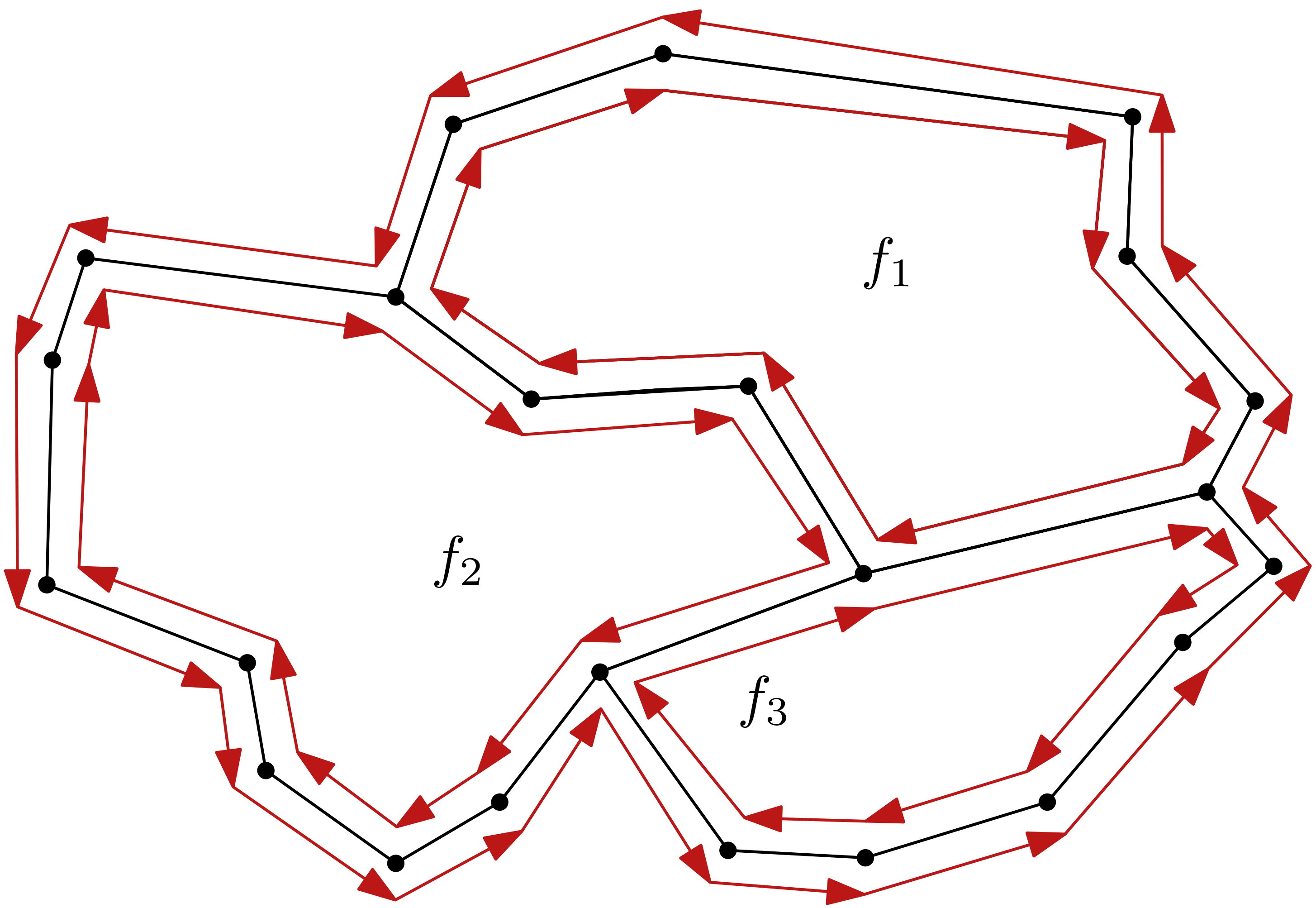


Subdivision



For each edge of internal faces introduce directed half-edge (clockwise)

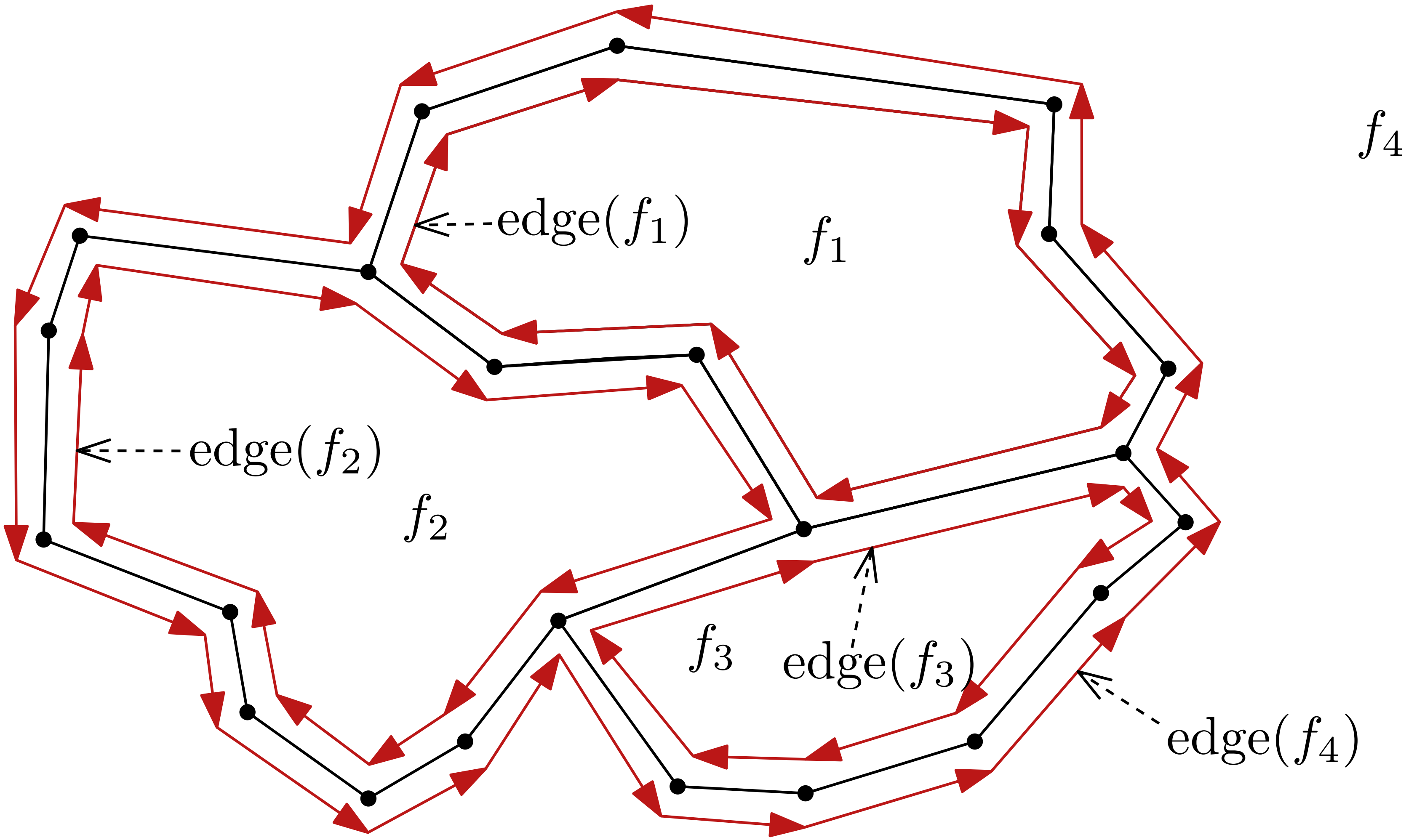
Subdivision



For each edge of internal faces introduce directed half-edge (clockwise)

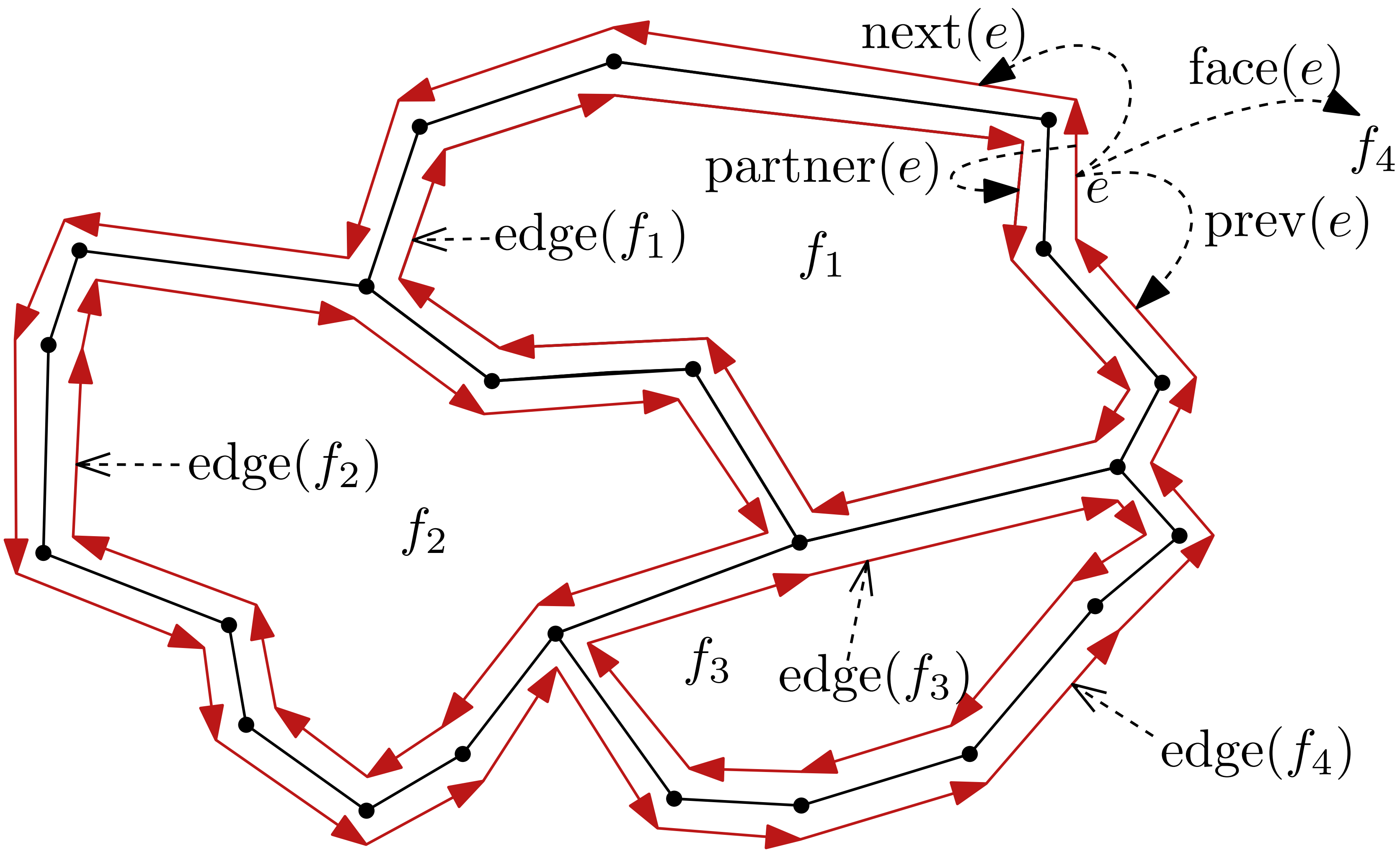
For each edge of external face introduce directed half-edge (counter-clockw.)

Subdivision



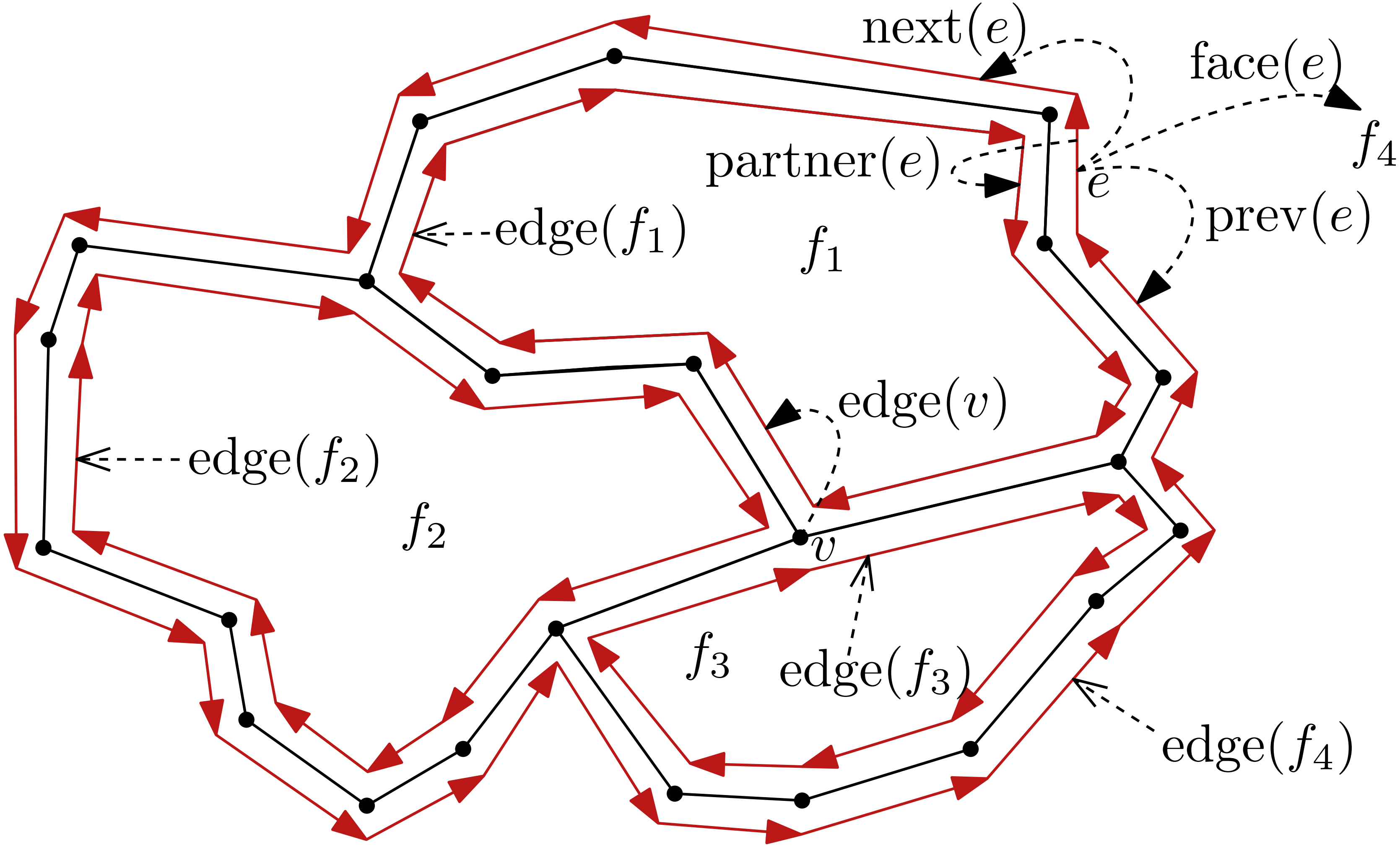
Store for each face arbitrary adjacent half-edge.

Subdivision



Store for each face arbitrary adjacent half-edge.
Store for each half-edge successor/predecessor, the half-edge on the opposite side, and the adjacent face.

Subdivision

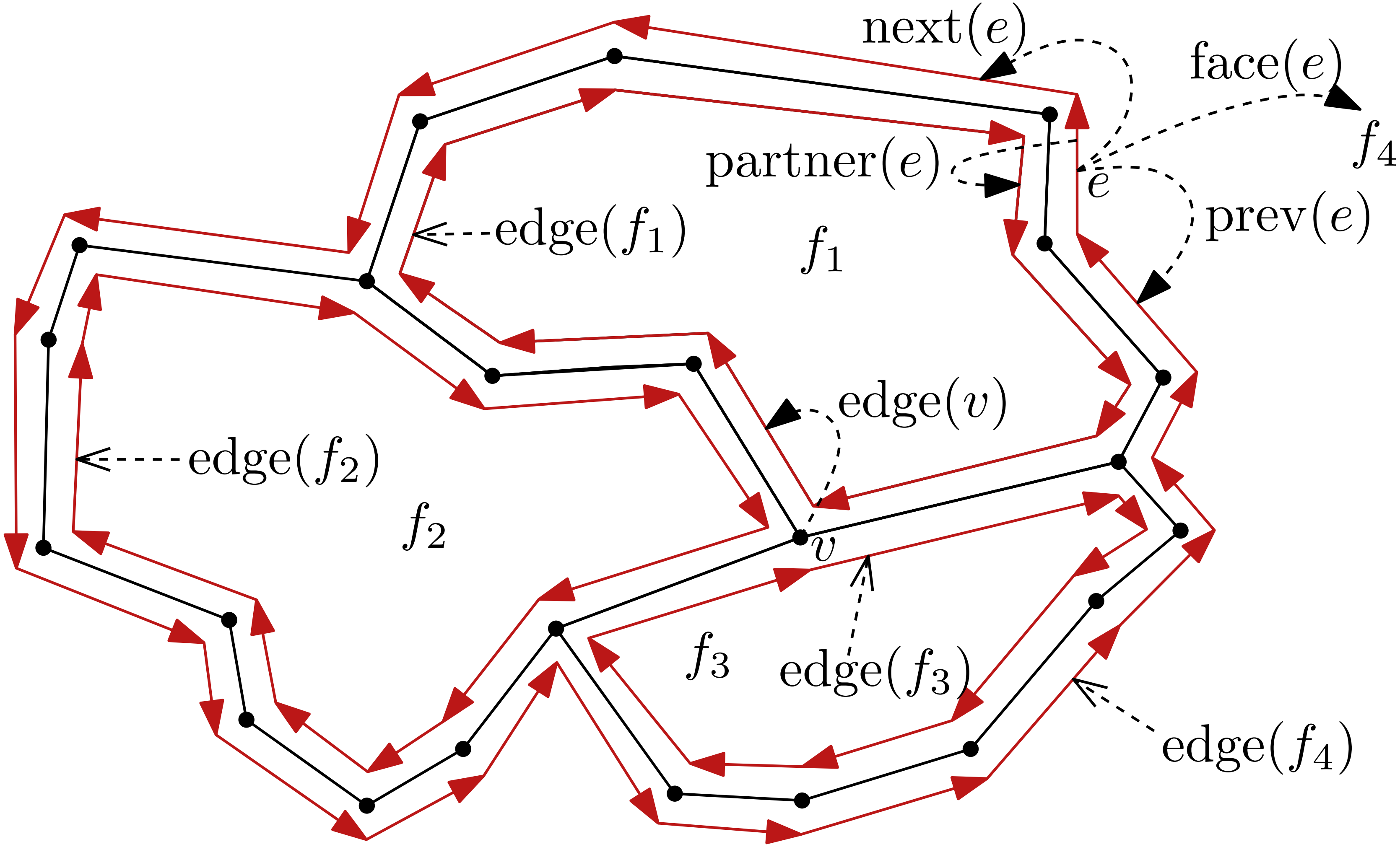


Store for each face arbitrary adjacent half-edge.

Store for each half-edge successor/predecessor, the half-edge on the opposite side, and the adjacent face.

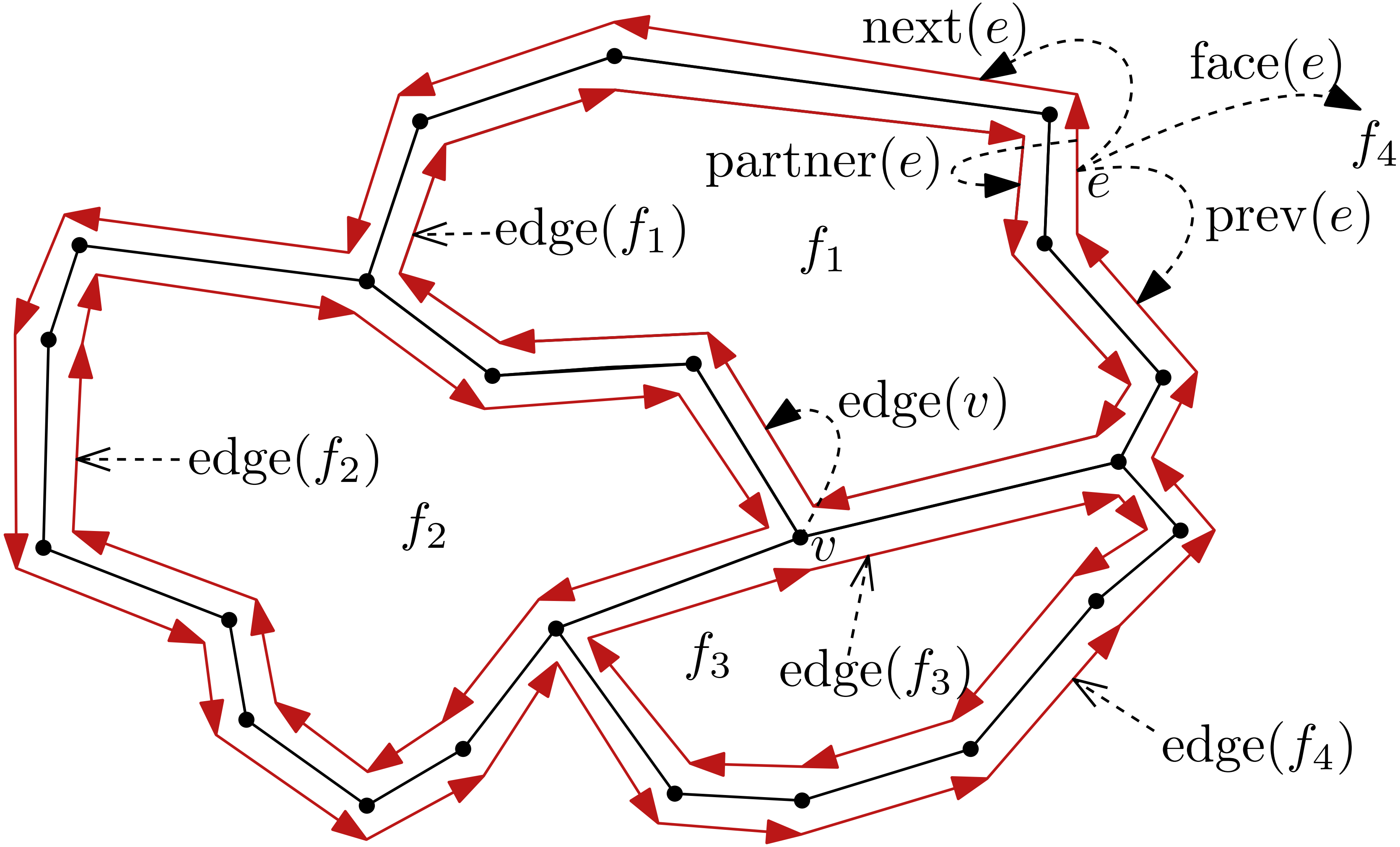
Store for each vertex an arbitrary incident outgoing half-edge.

Subdivision



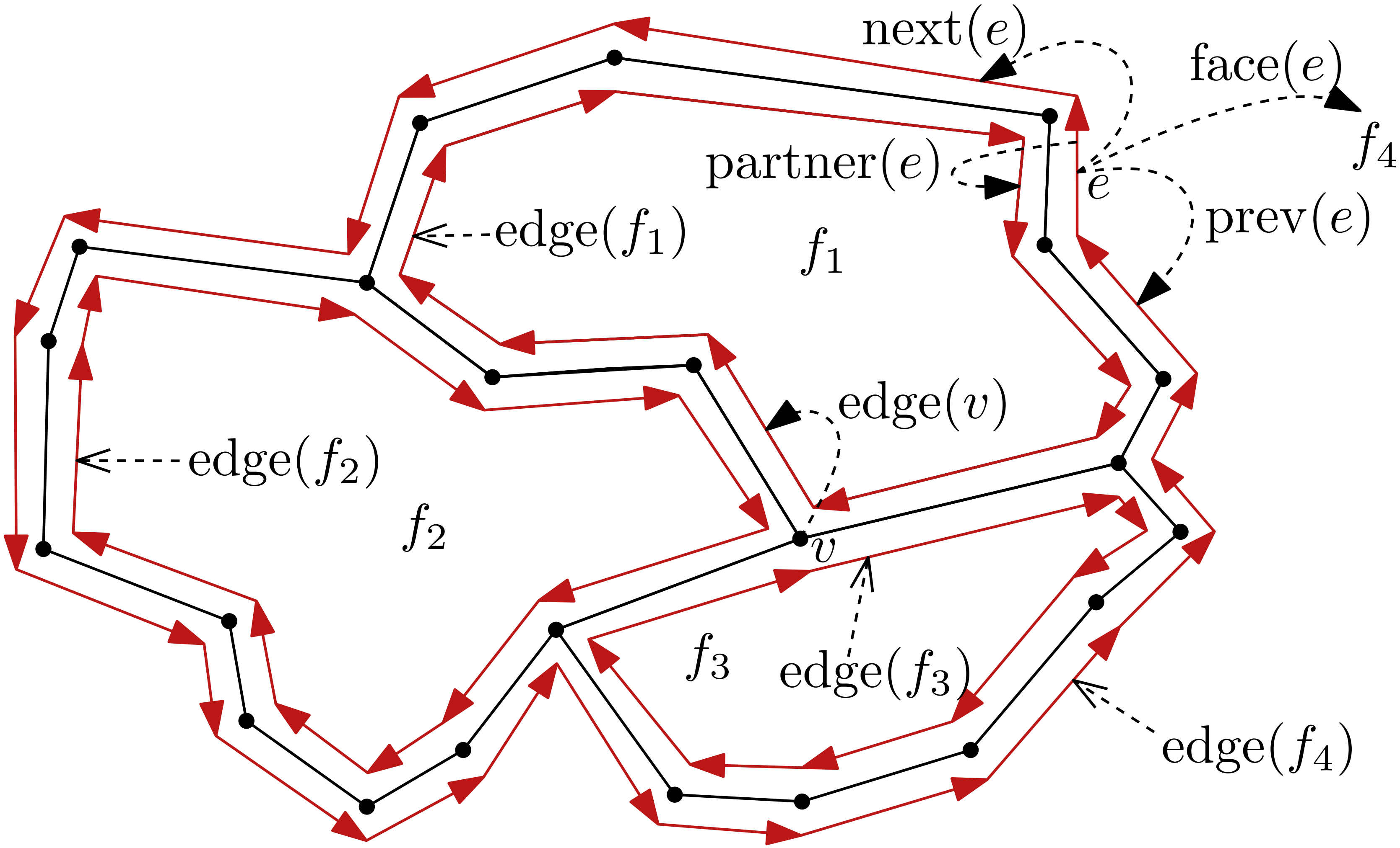
- Access vertices, faces and edges.
- *Traversing* single faces.
- Traversing outgoing edges of vertex.

Subdivision



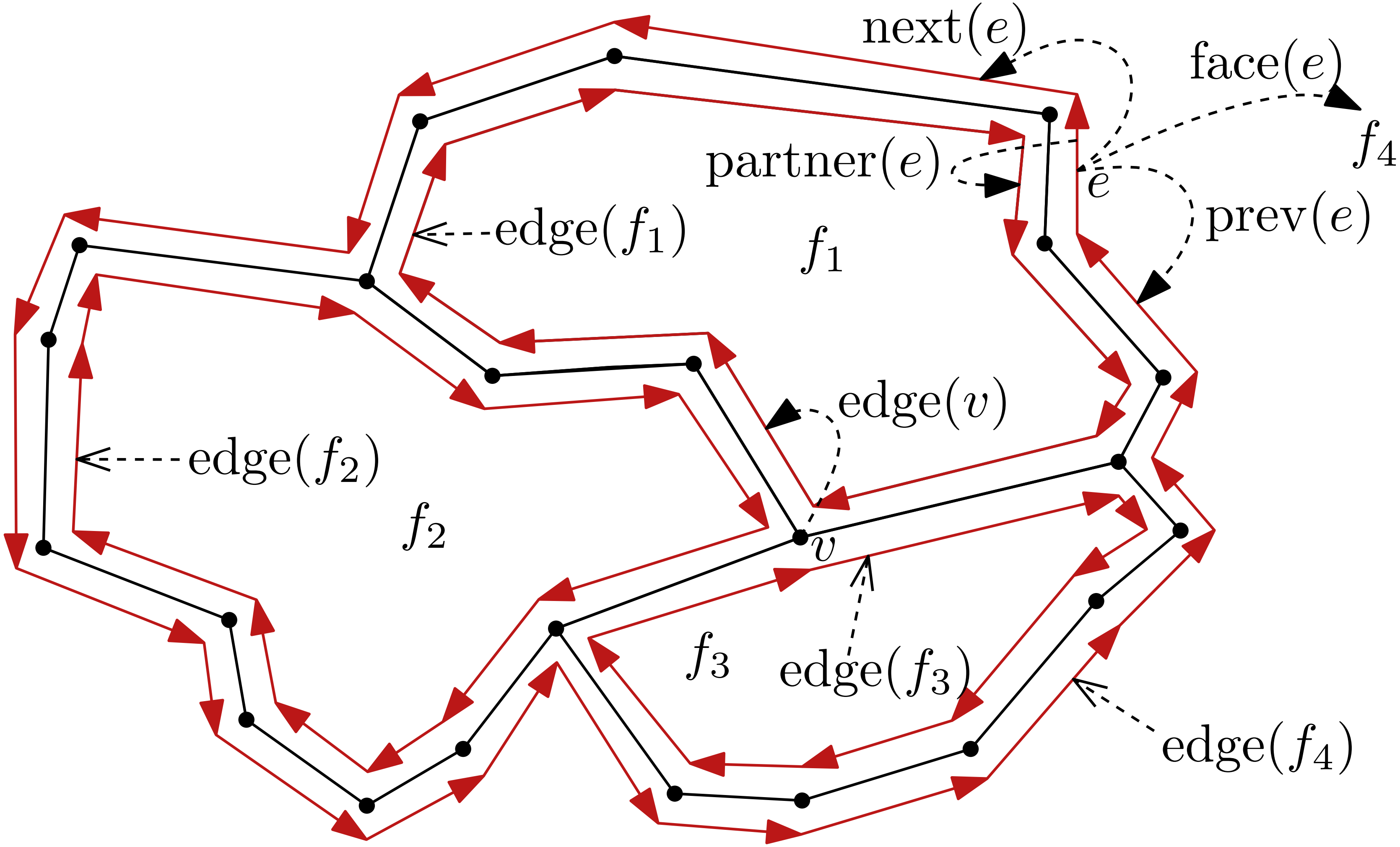
- Access vertices, faces and edges ✓
- *Traversing* single faces.
- Traversing outgoing edges of vertex.

Subdivision



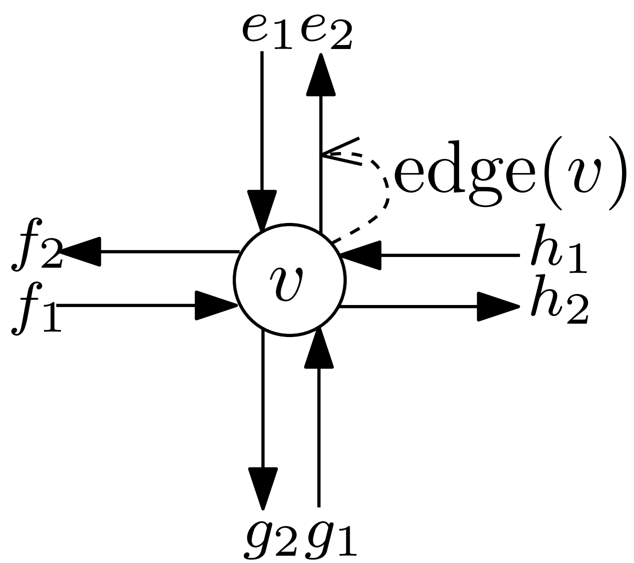
- Access vertices, faces and edges ✓
- *Traversing* single faces ✓
- Traversing outgoing edges of vertex.

Subdivision

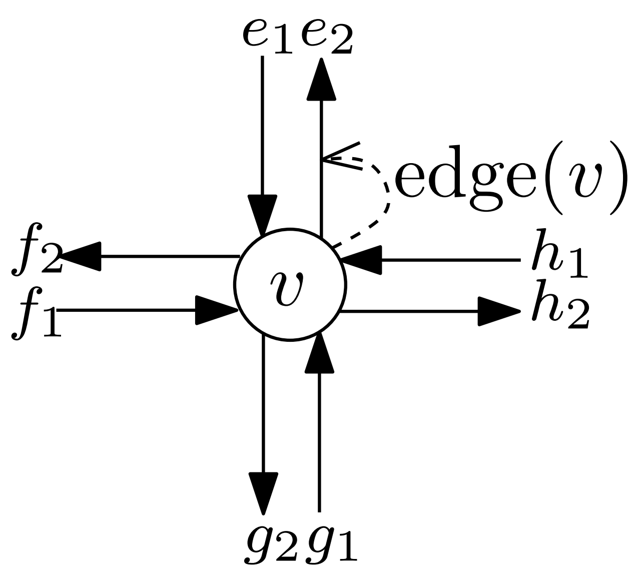


- Access vertices, faces and edges ✓
- *Traversing* single faces ✓
- Traversing outgoing edges of vertex. ?

Traversing incident edges



Traversing incident edges



Traversing in counter clockwise order.

$$f_2 = \text{next}(\text{partner}(e_2))$$
$$g_2 = \text{next}(\text{partner}(f_2))$$
$$h_2 = \text{next}(\text{partner}(g_2))$$
$$e_2 = \text{next}(\text{partner}(h_2))$$