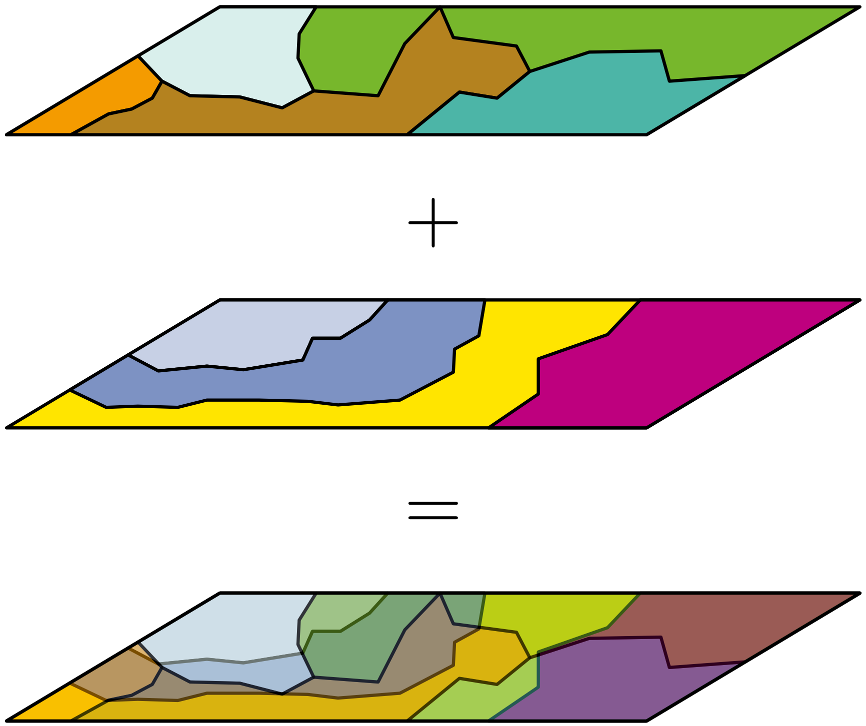
Vorlesung Algorithmische Geometrie
Streckenschnitte

INSTITUT FÜR THEORETISCHE INFORMATIK · FAKULTÄT FÜR INFORMATIK

Martin Nöllenburg
24.04.2011

Überlagern von Kartenebenen

Beispiel: Gegeben zwei verschiedene Kartenebenen, bestimme deren Schnitt.



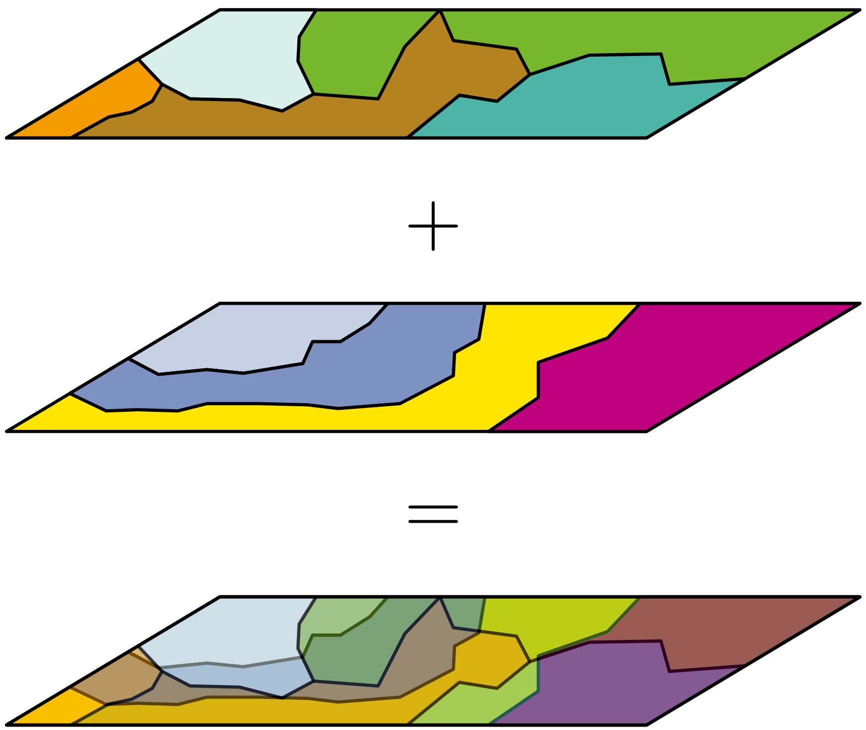
Flächennutzung

Niederschlag

Kombination beider Themen

Überlagern von Kartenebenen

Beispiel: Gegeben zwei verschiedene Kartenebenen, bestimme deren Schnitt.



Flächennutzung

Niederschlag

Kombination beider Themen

- Regionen sind Polygone
- Polygone sind Streckenmengen
- **berechne alle Streckenschnittpunkte**
- berechne induzierte Regionen

Problemstellung

Geg: Menge $S = \{s_1, \ldots, s_n\}$ von Strecken in der Ebene

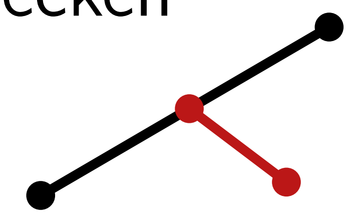
Ges:
- alle Schnittpunkte von zwei oder mehr Strecken
- für jeden Schnittpunkt die beteiligten Strecken

Problemstellung

Geg: Menge $S = \{s_1, \dots, s_n\}$ von Strecken in der Ebene

Ges:
- alle Schnittpunkte von zwei oder mehr Strecken
- für jeden Schnittpunkt die beteiligten Strecken

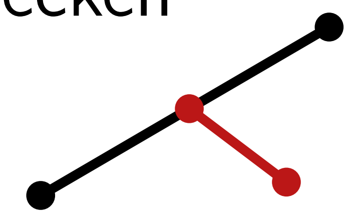
Def: Strecken sind **abgeschlossene** Mengen



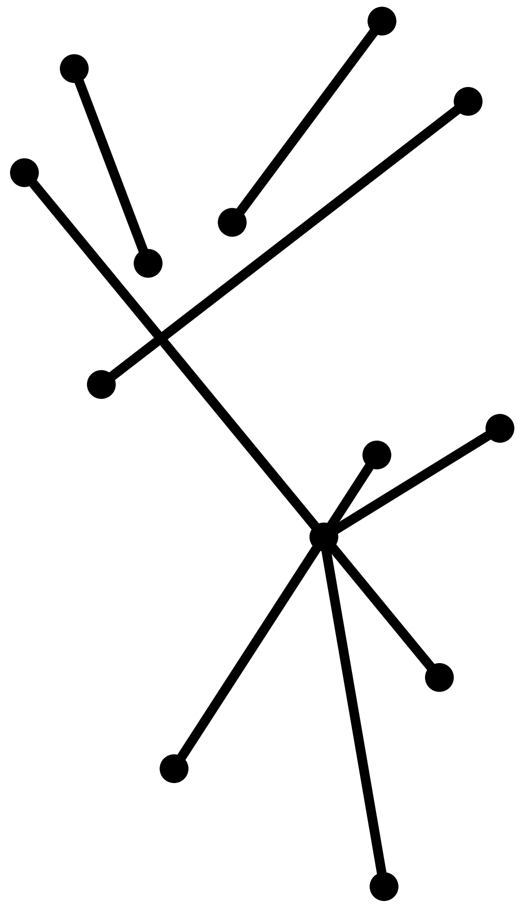
Problemstellung

Geg: Menge $S = \{s_1, \dots, s_n\}$ von Strecken in der Ebene

Ges:
- alle Schnittpunkte von zwei oder mehr Strecken
- für jeden Schnittpunkt die beteiligten Strecken



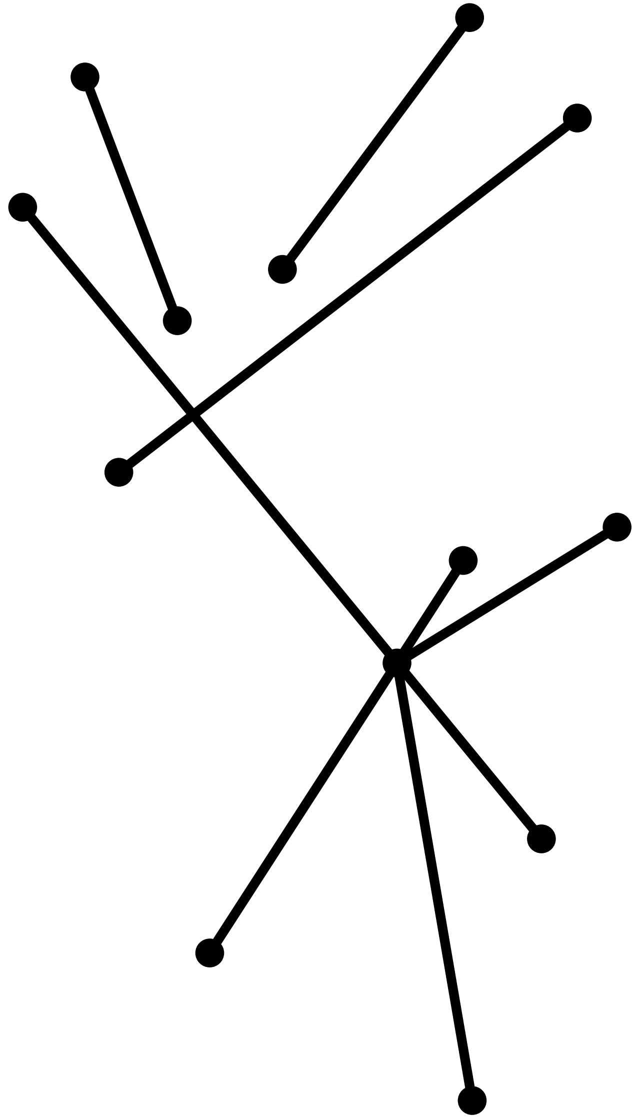
Def: Strecken sind **abgeschlossene** Mengen



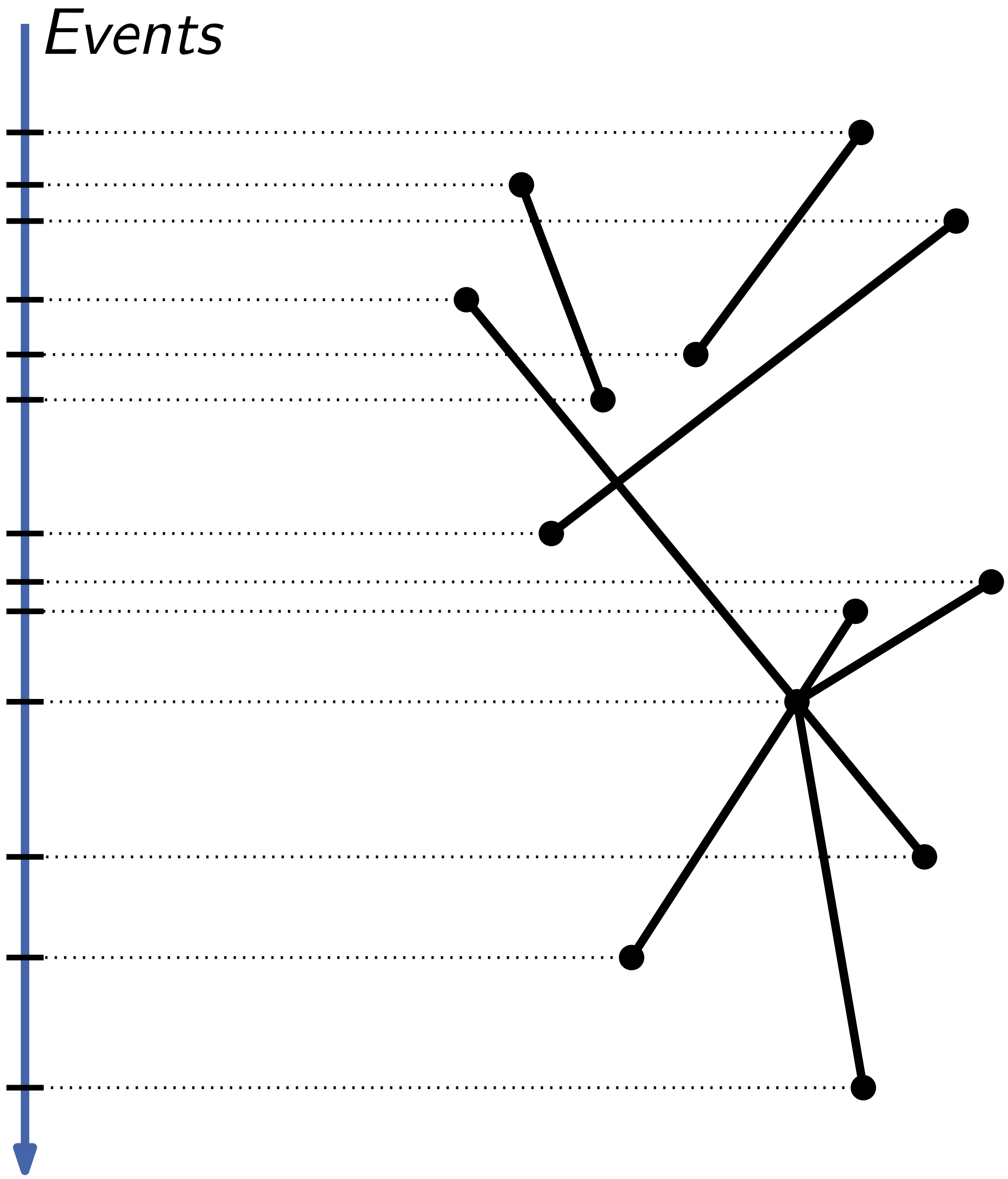
Diskussion:

– Wie kann man das Problem naiv lösen?
– Ist das evtl. schon optimal?
– Sehen Sie Verbesserungsansätze?

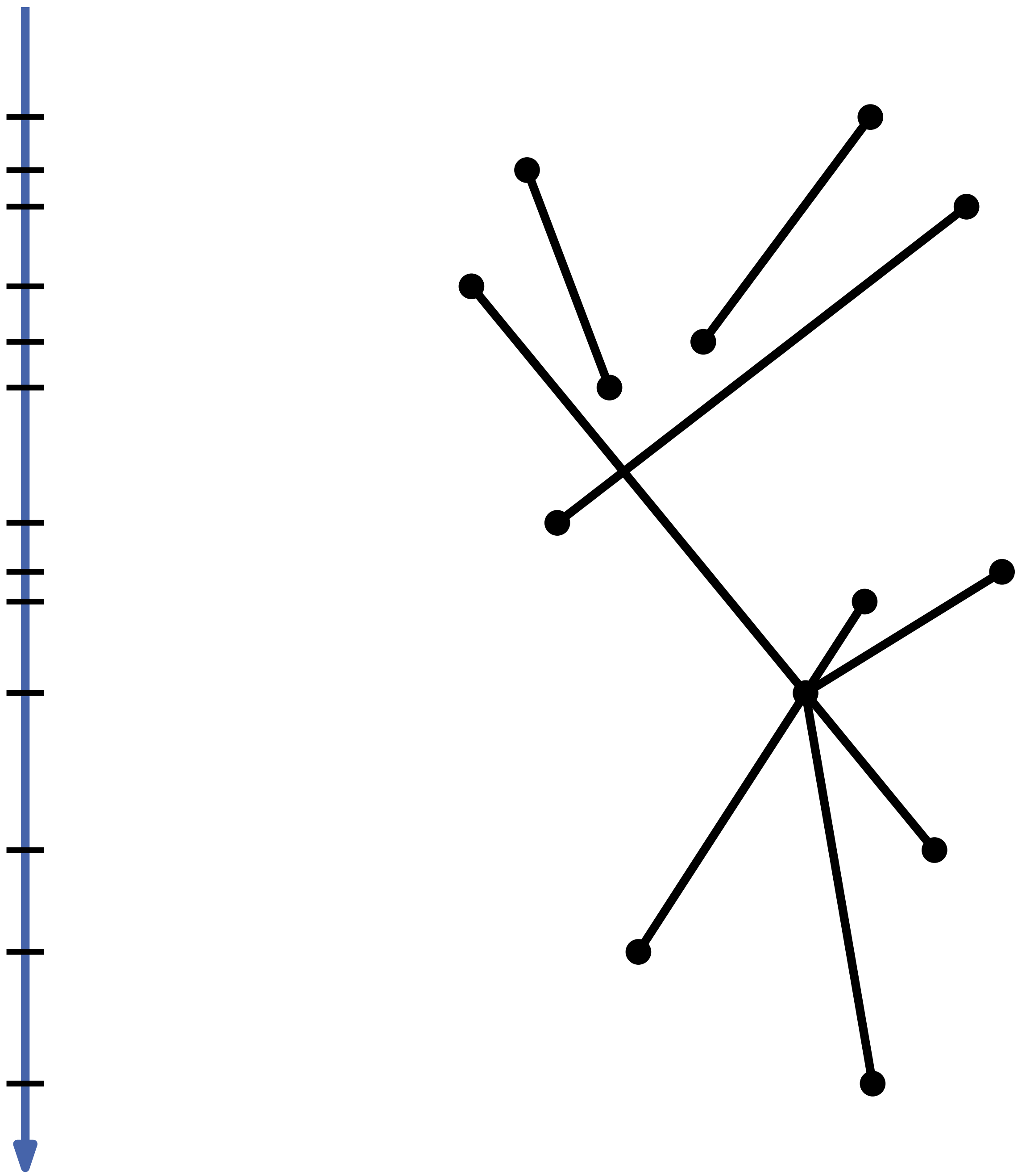
Das Sweep-Line Verfahren: Beispiel



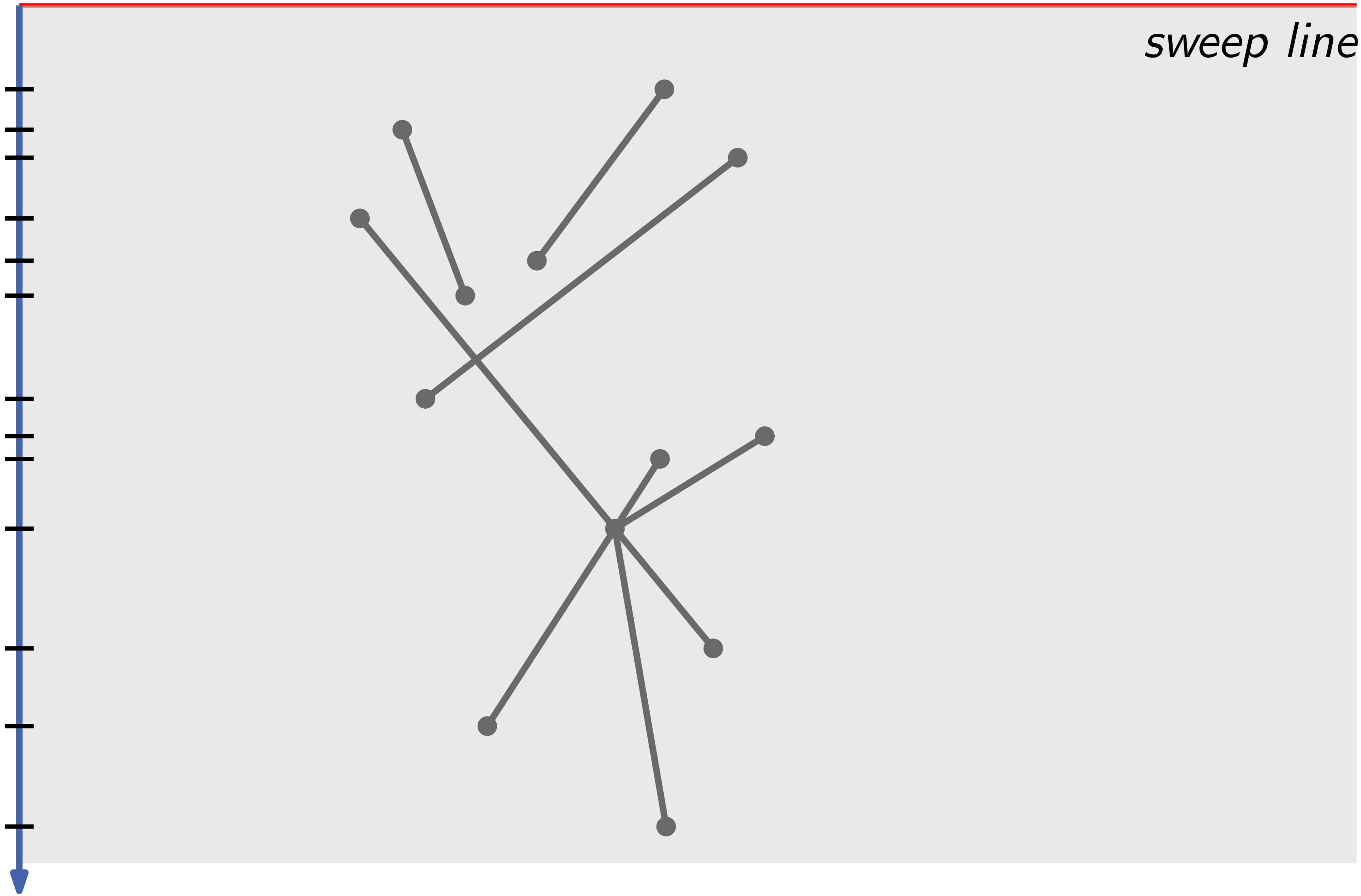
Das Sweep-Line Verfahren: Beispiel



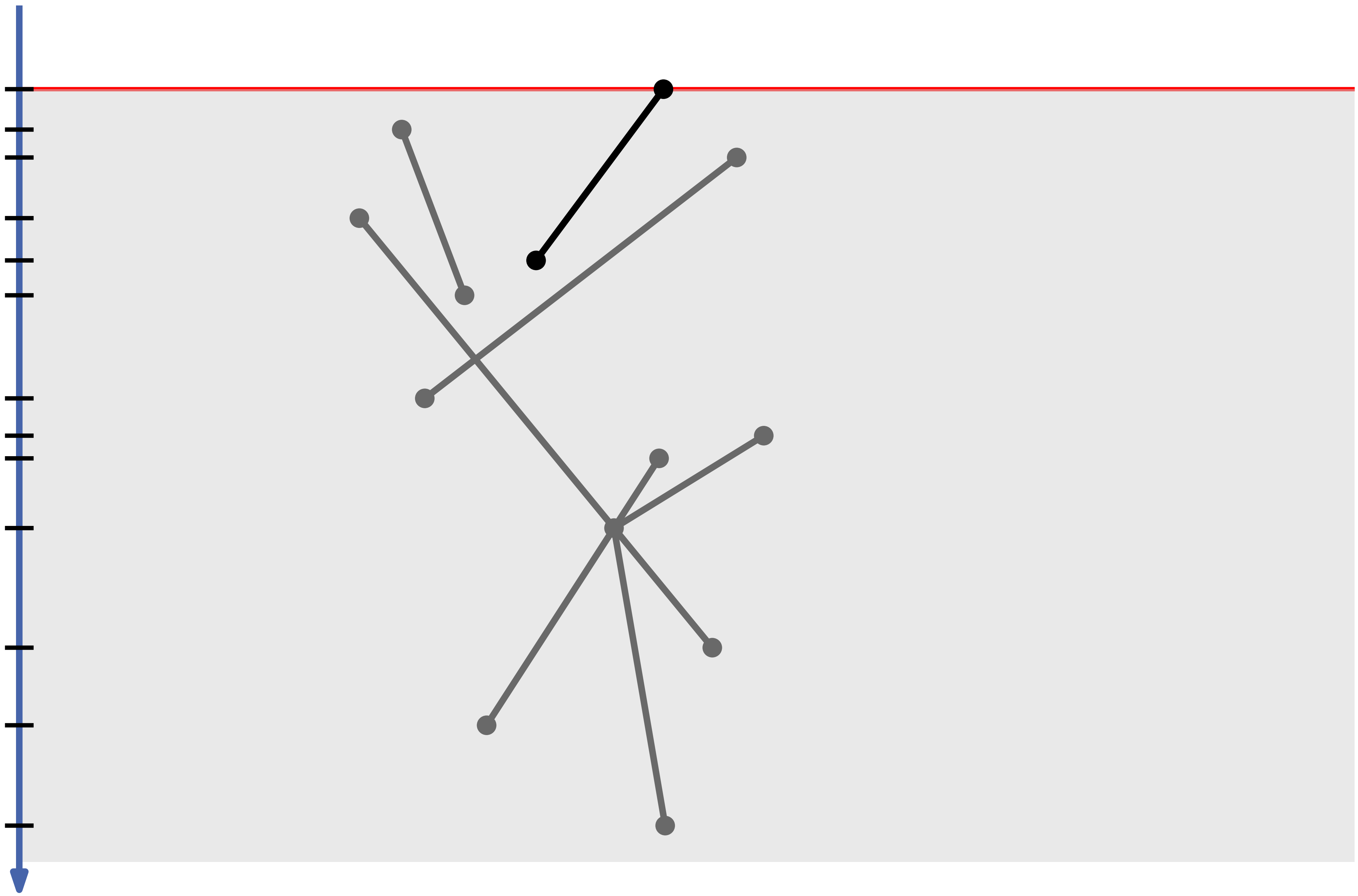
Das Sweep-Line Verfahren: Beispiel



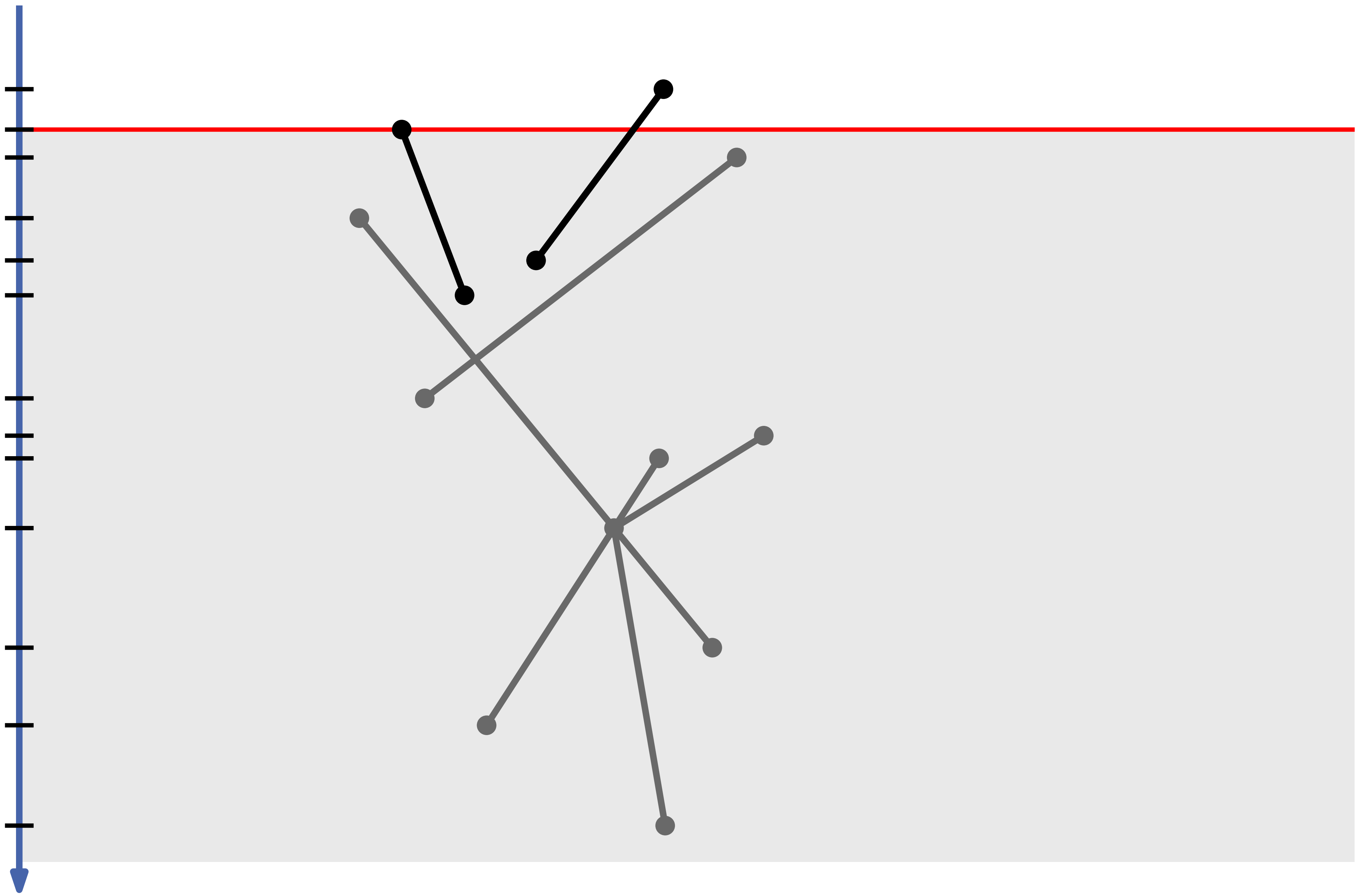
Das Sweep-Line Verfahren: Beispiel



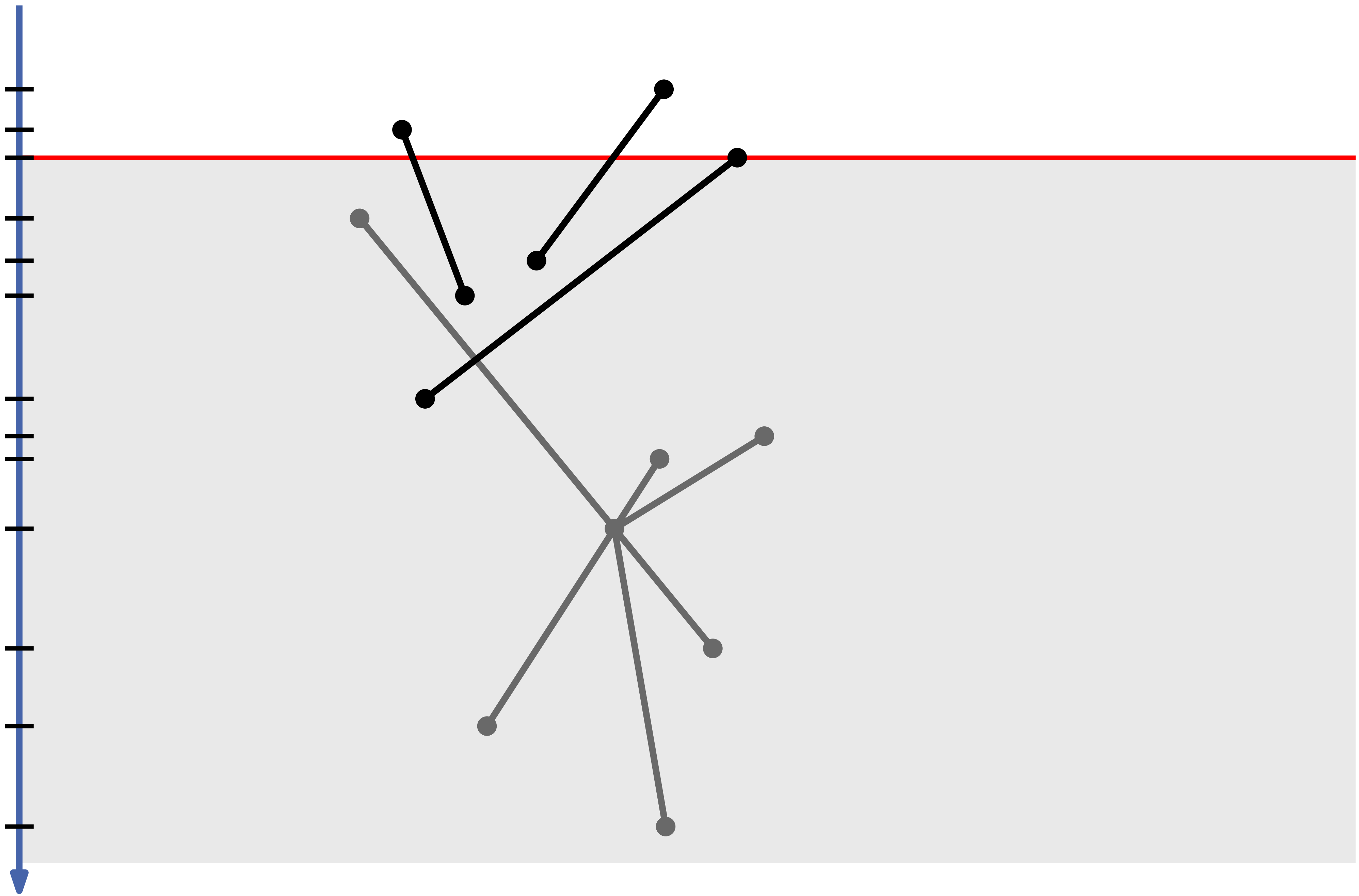
Das Sweep-Line Verfahren: Beispiel



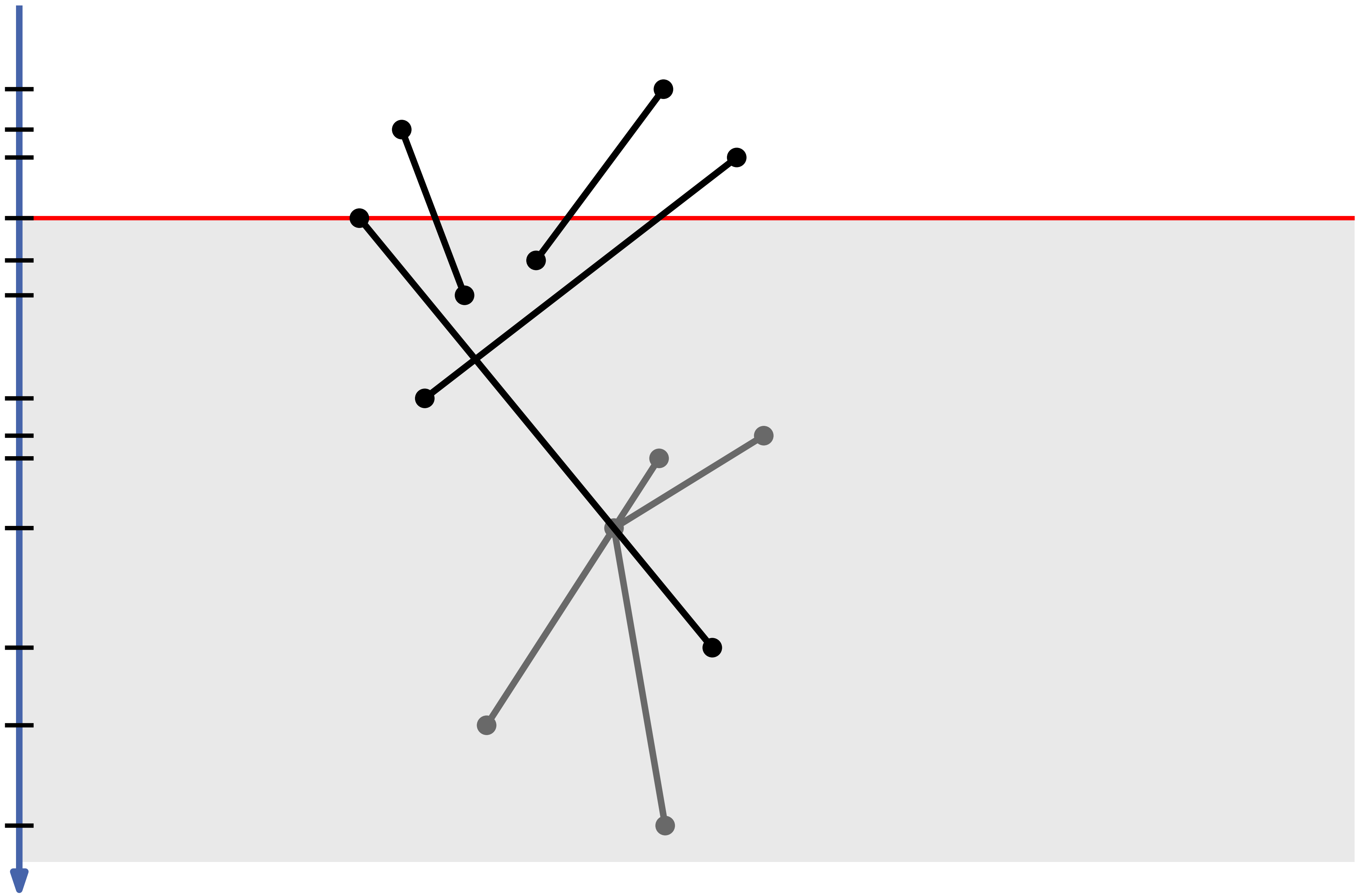
Das Sweep-Line Verfahren: Beispiel



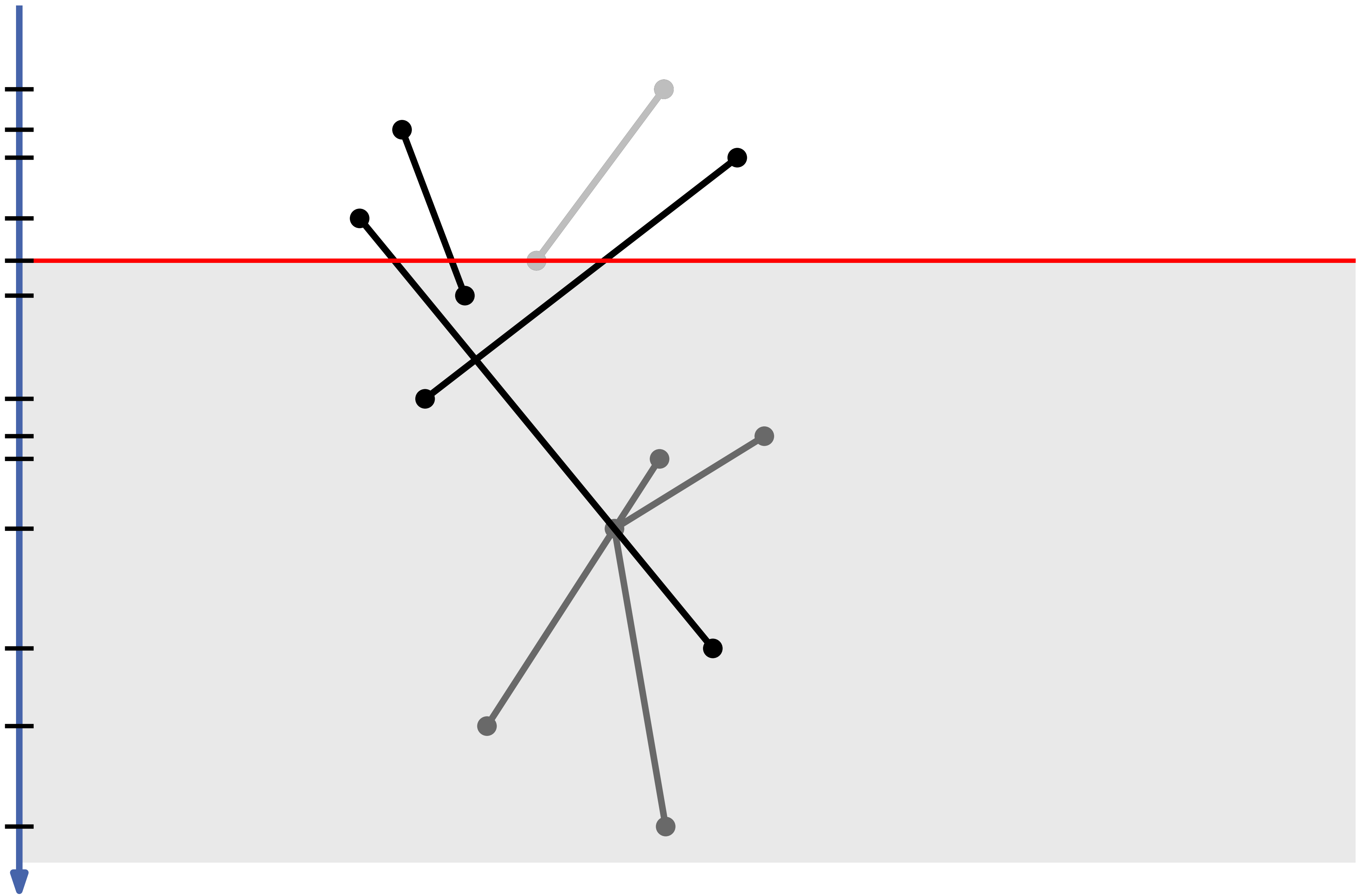
Das Sweep-Line Verfahren: Beispiel



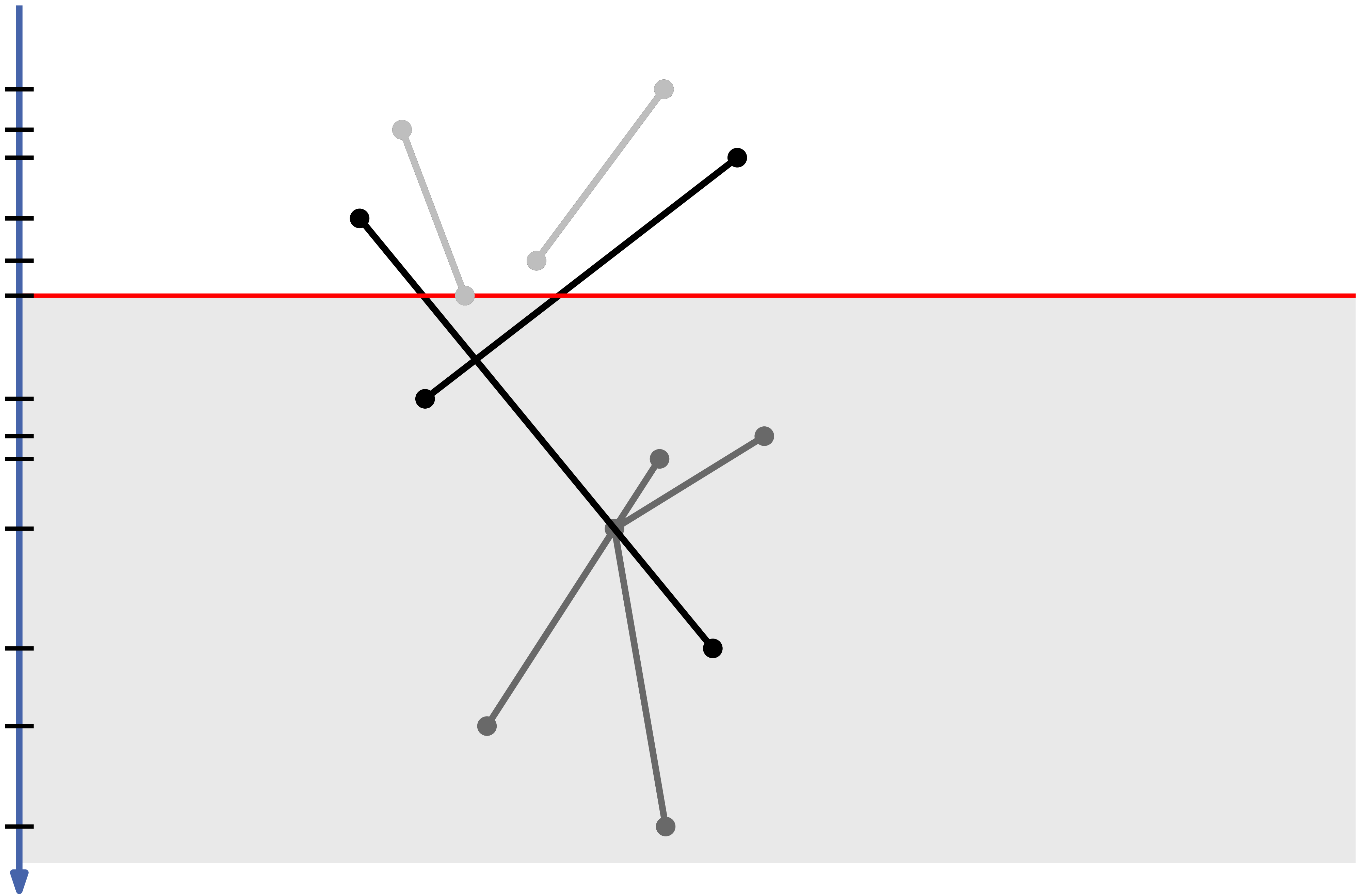
Das Sweep-Line Verfahren: Beispiel



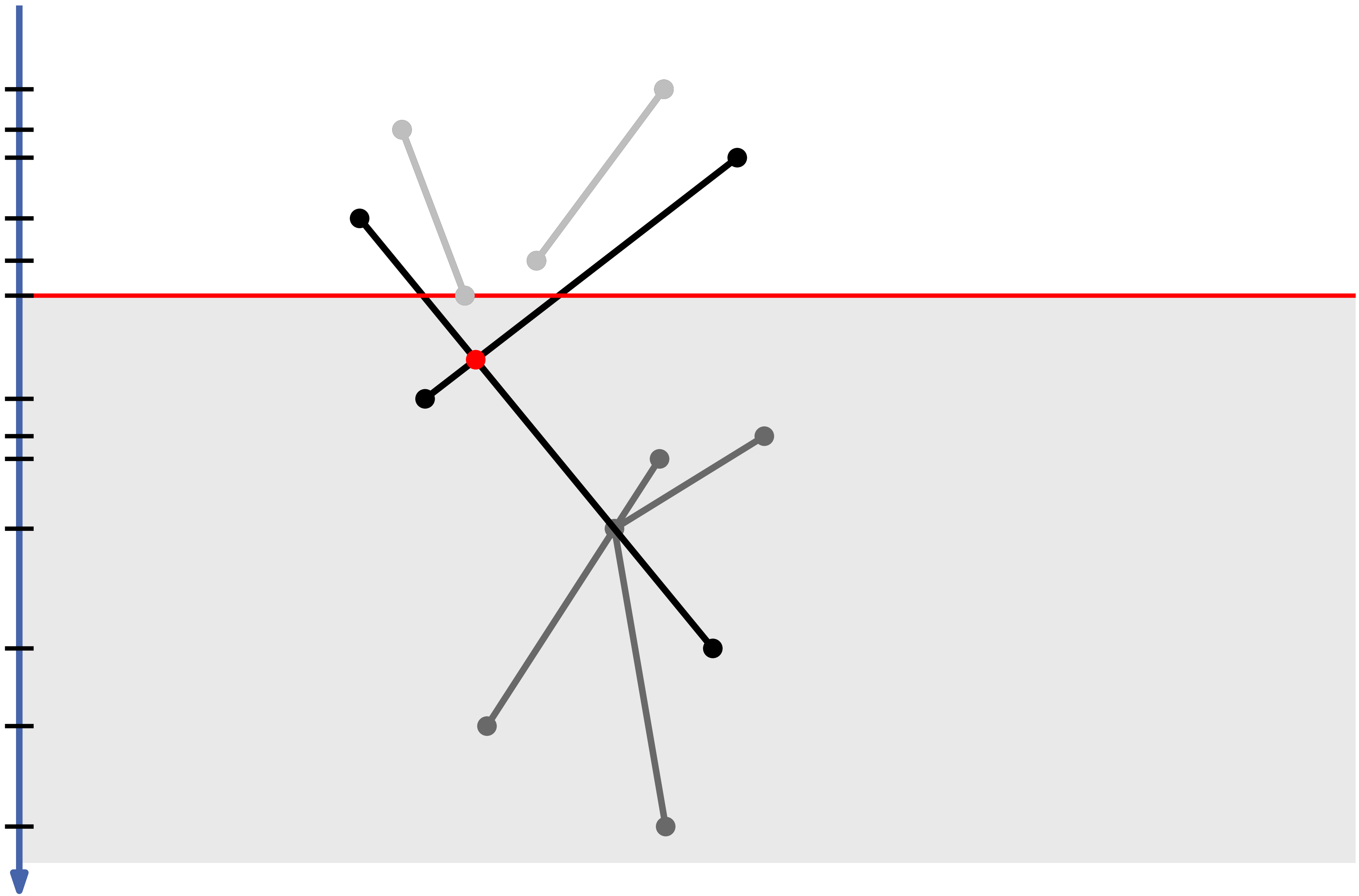
Das Sweep-Line Verfahren: Beispiel



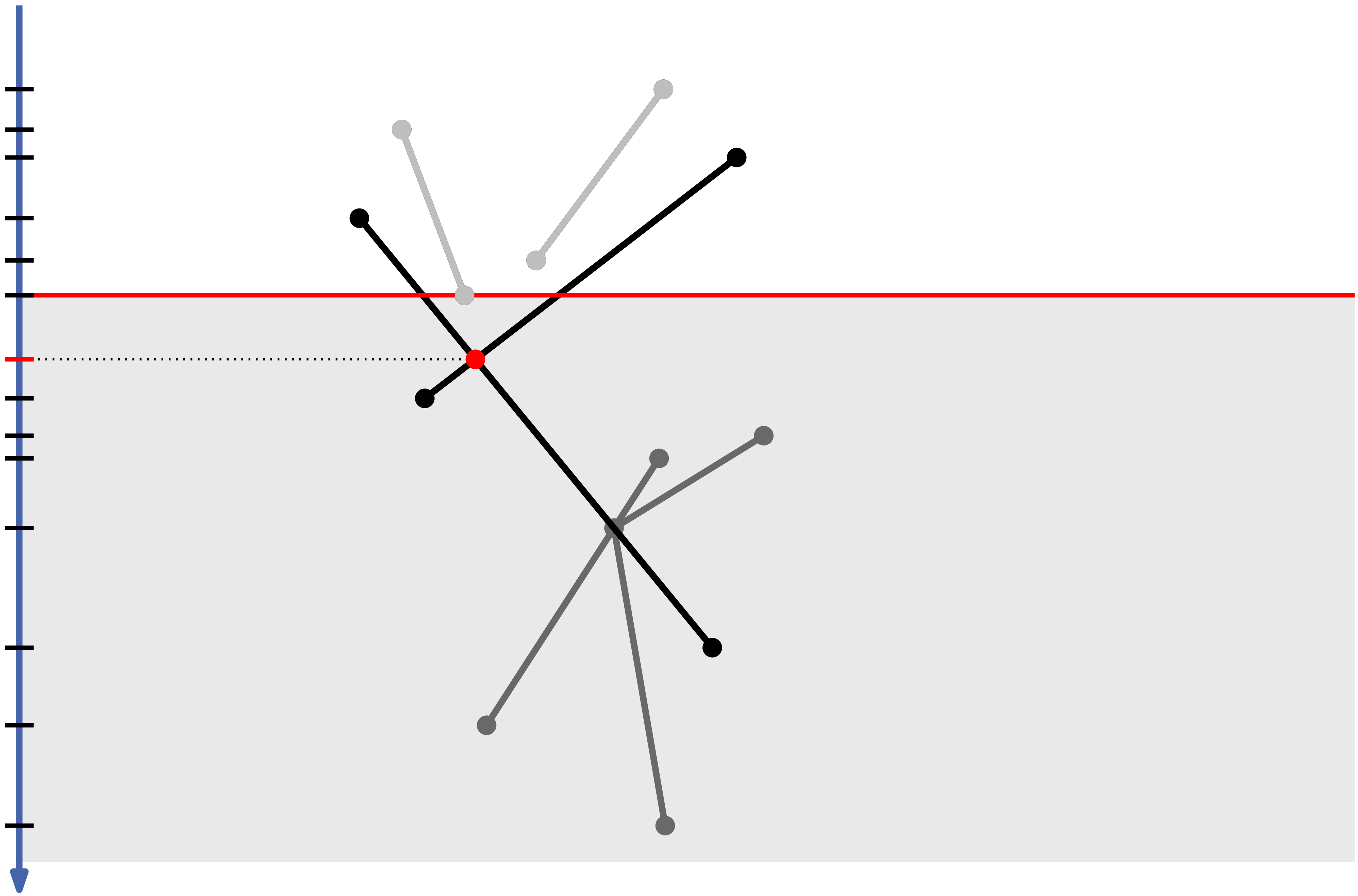
Das Sweep-Line Verfahren: Beispiel



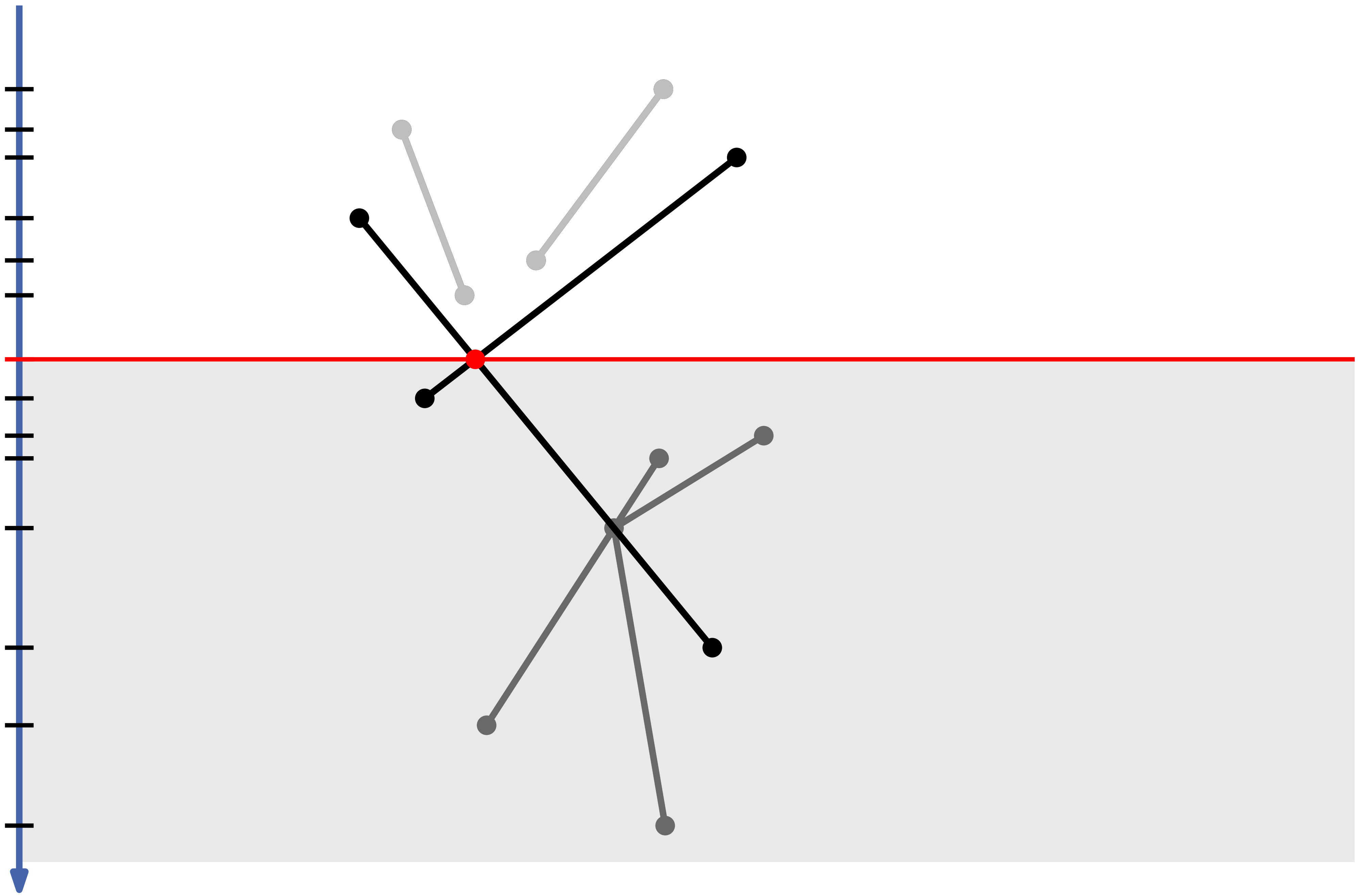
Das Sweep-Line Verfahren: Beispiel



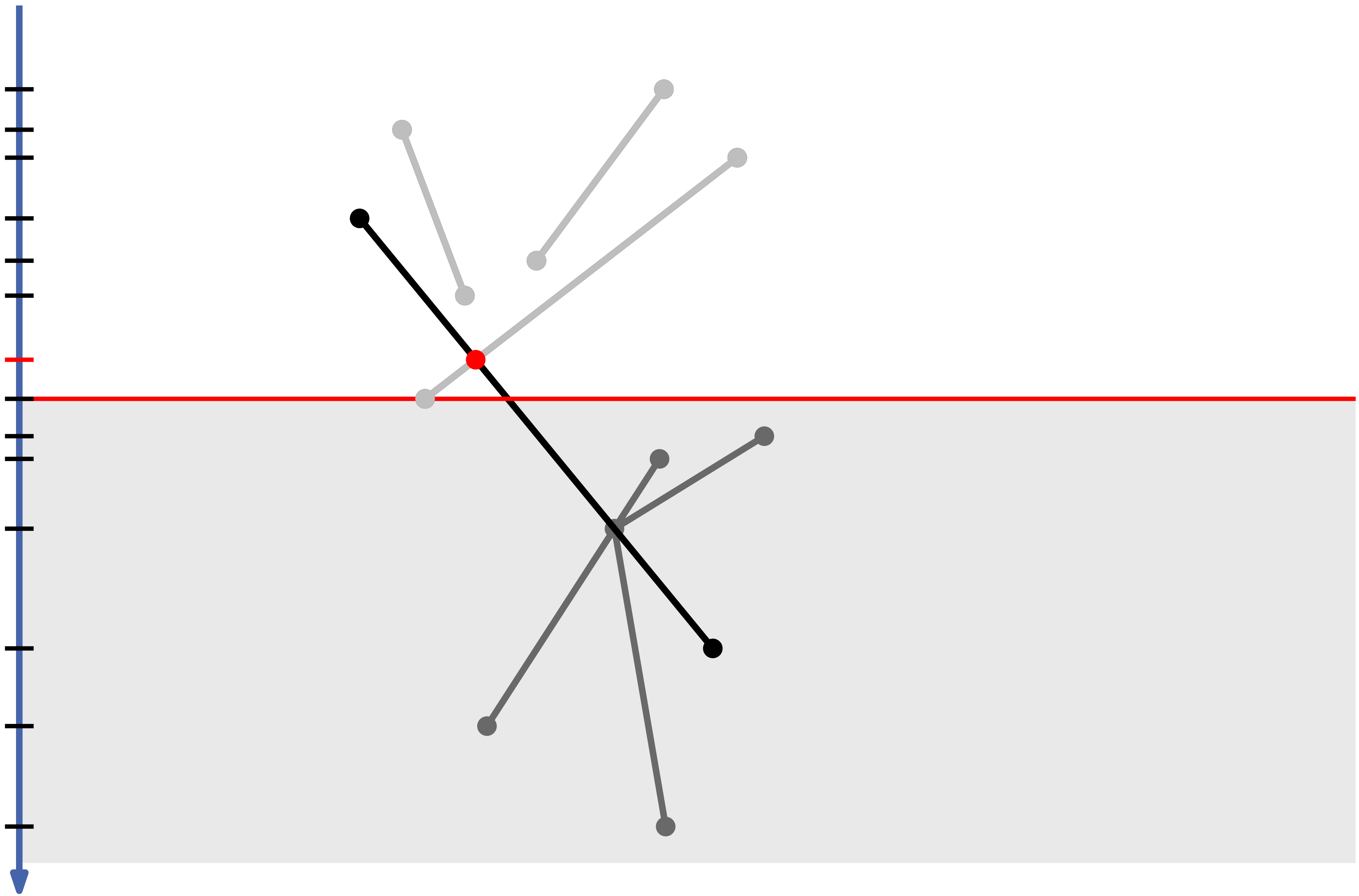
Das Sweep-Line Verfahren: Beispiel



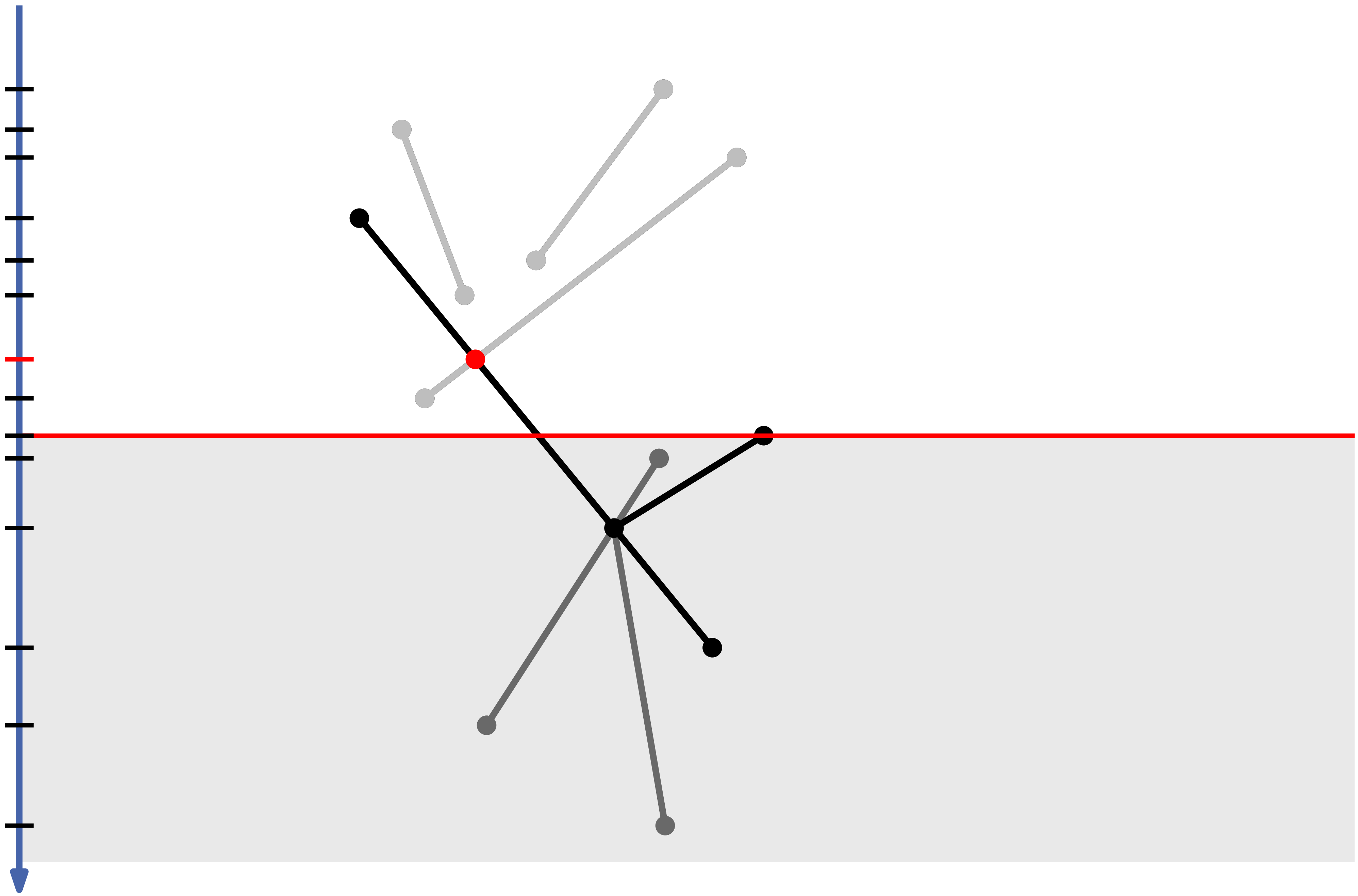
Das Sweep-Line Verfahren: Beispiel



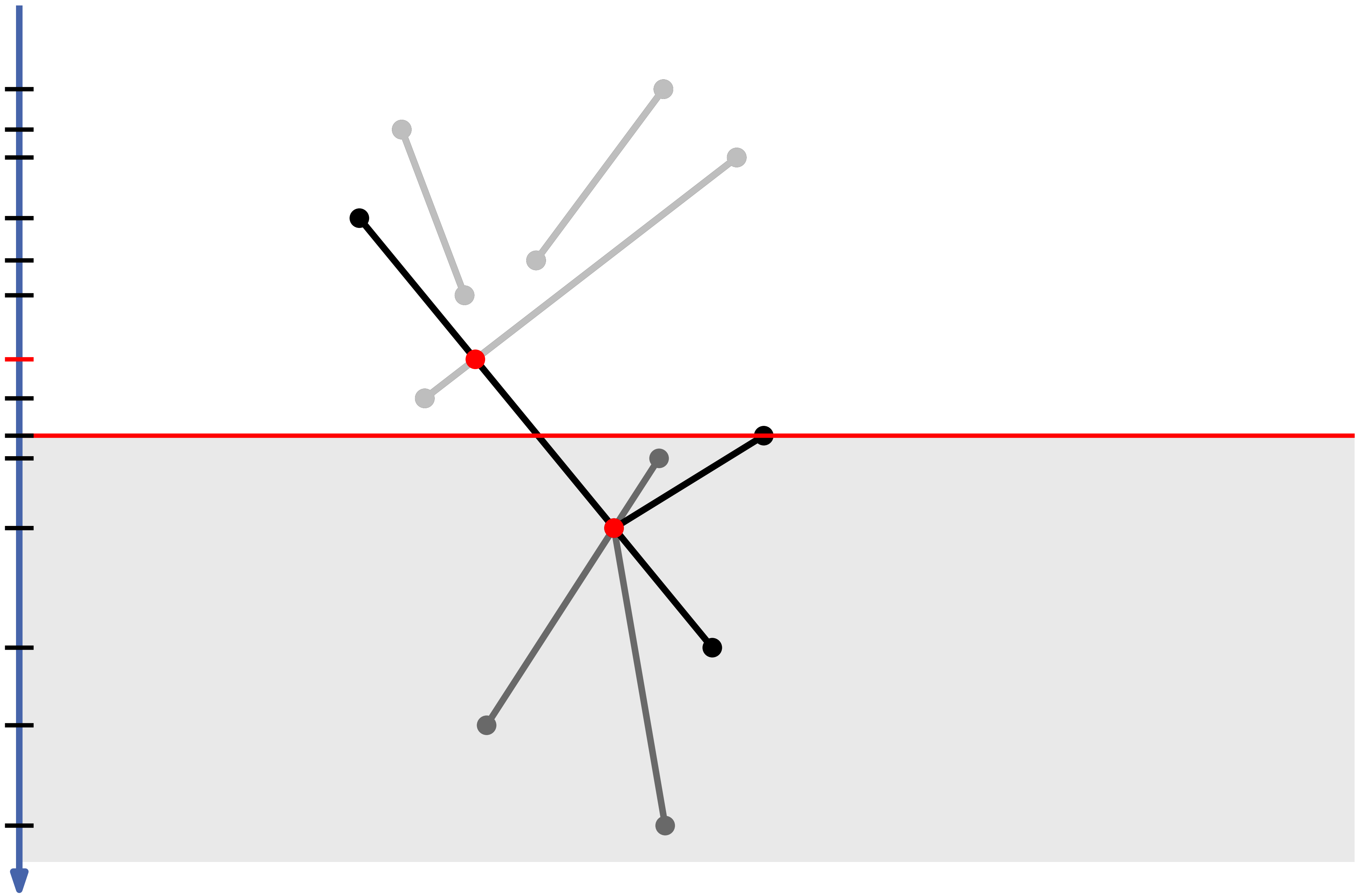
Das Sweep-Line Verfahren: Beispiel



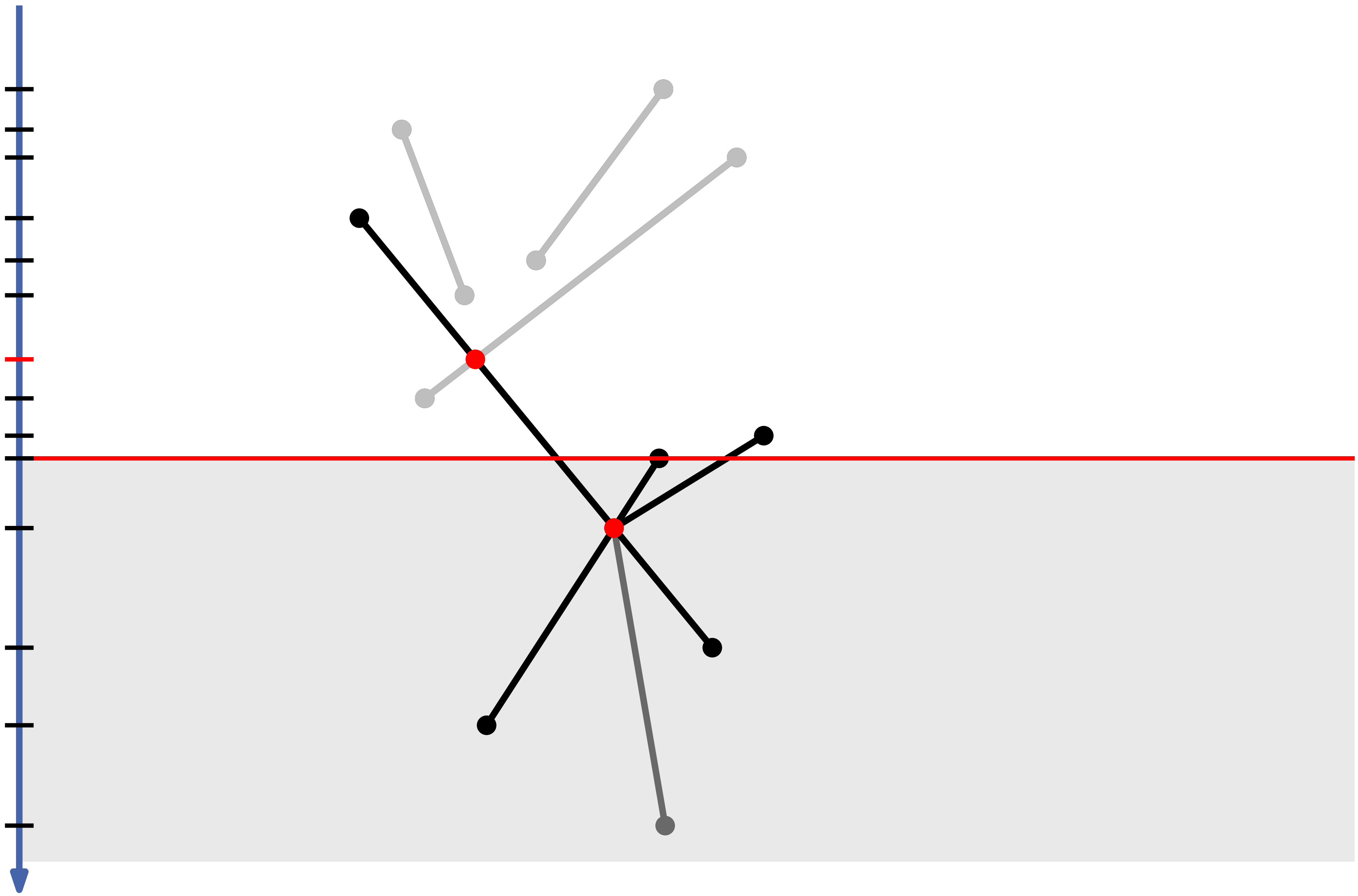
Das Sweep-Line Verfahren: Beispiel



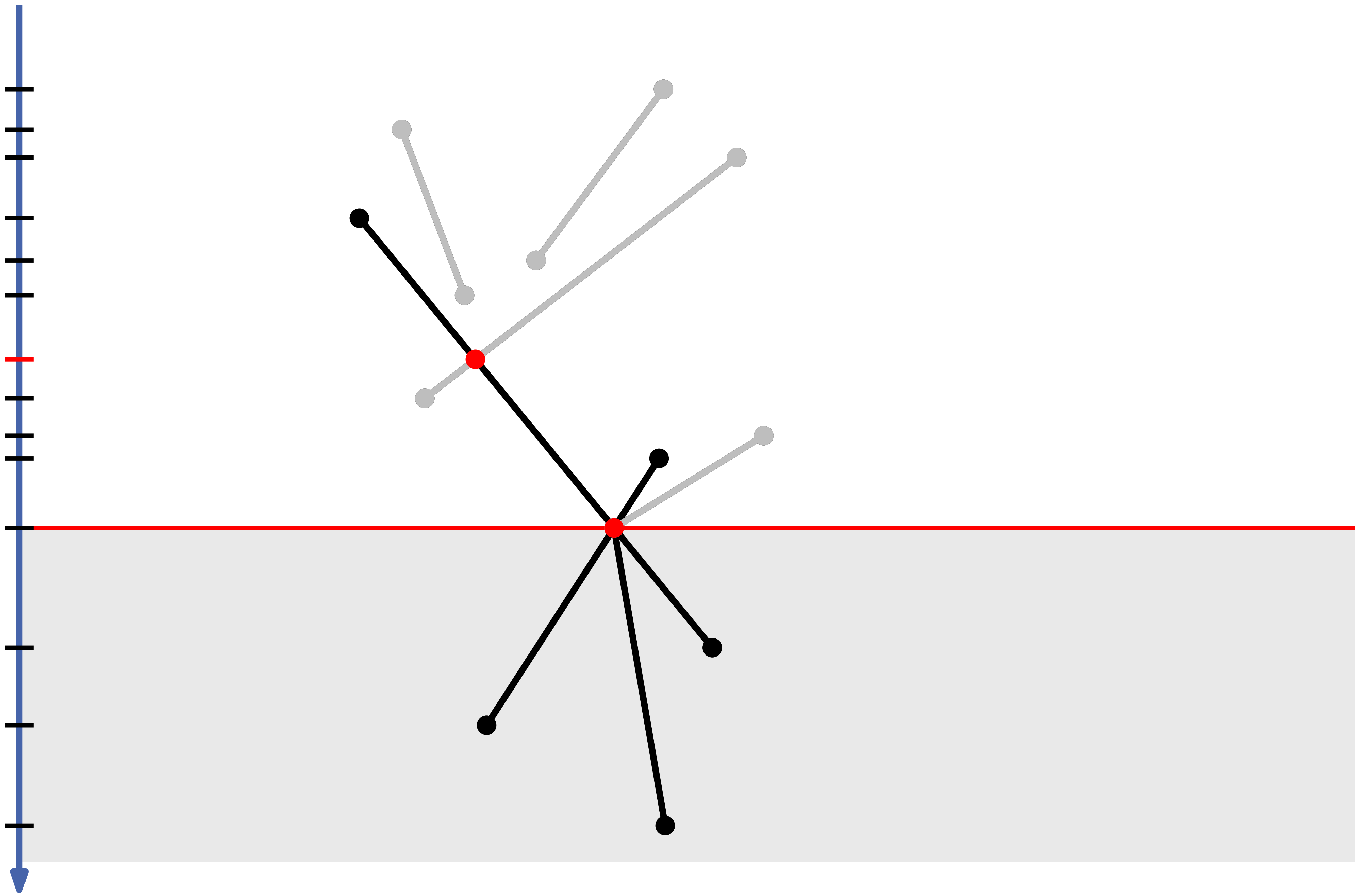
Das Sweep-Line Verfahren: Beispiel



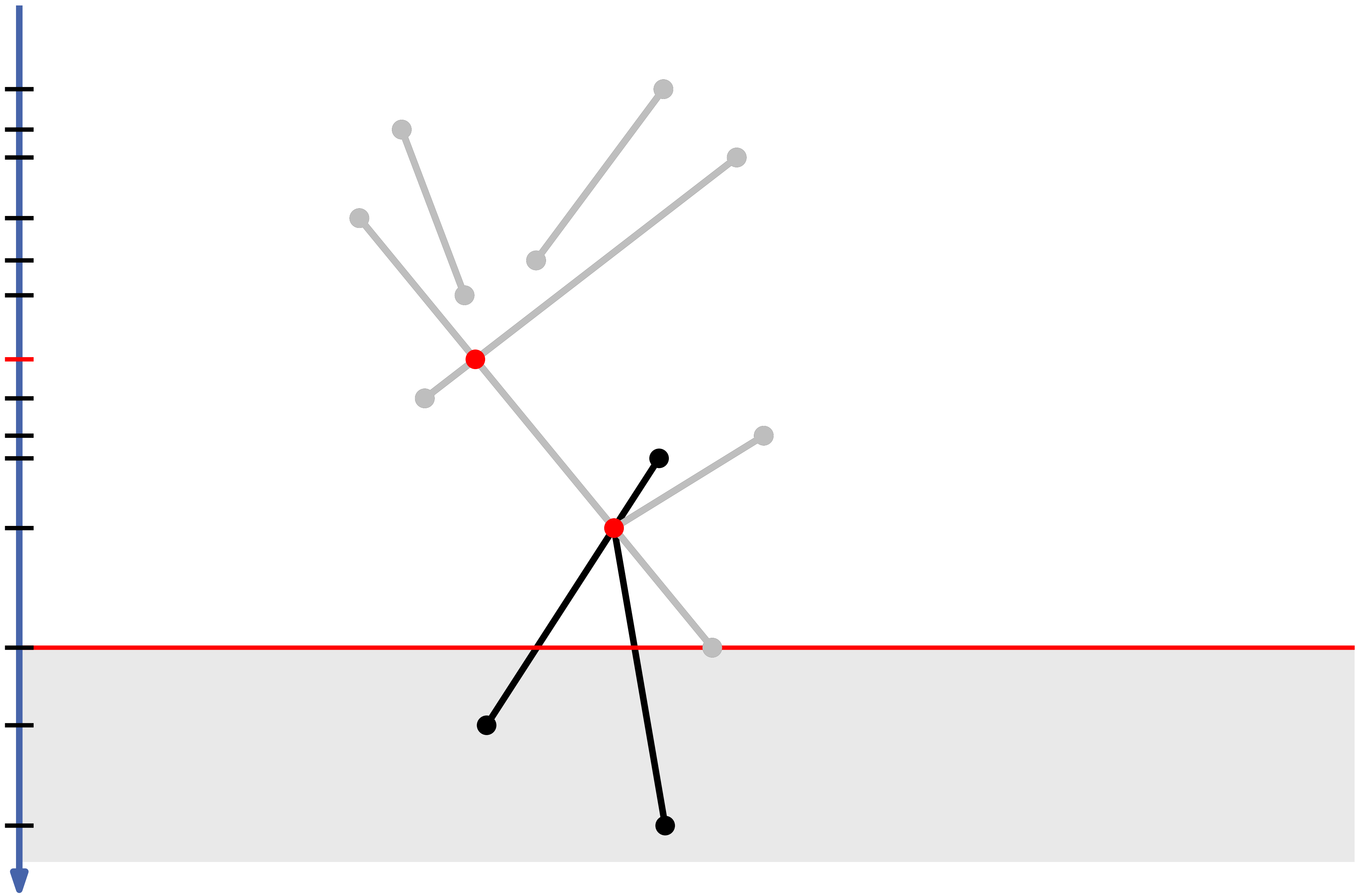
Das Sweep-Line Verfahren: Beispiel



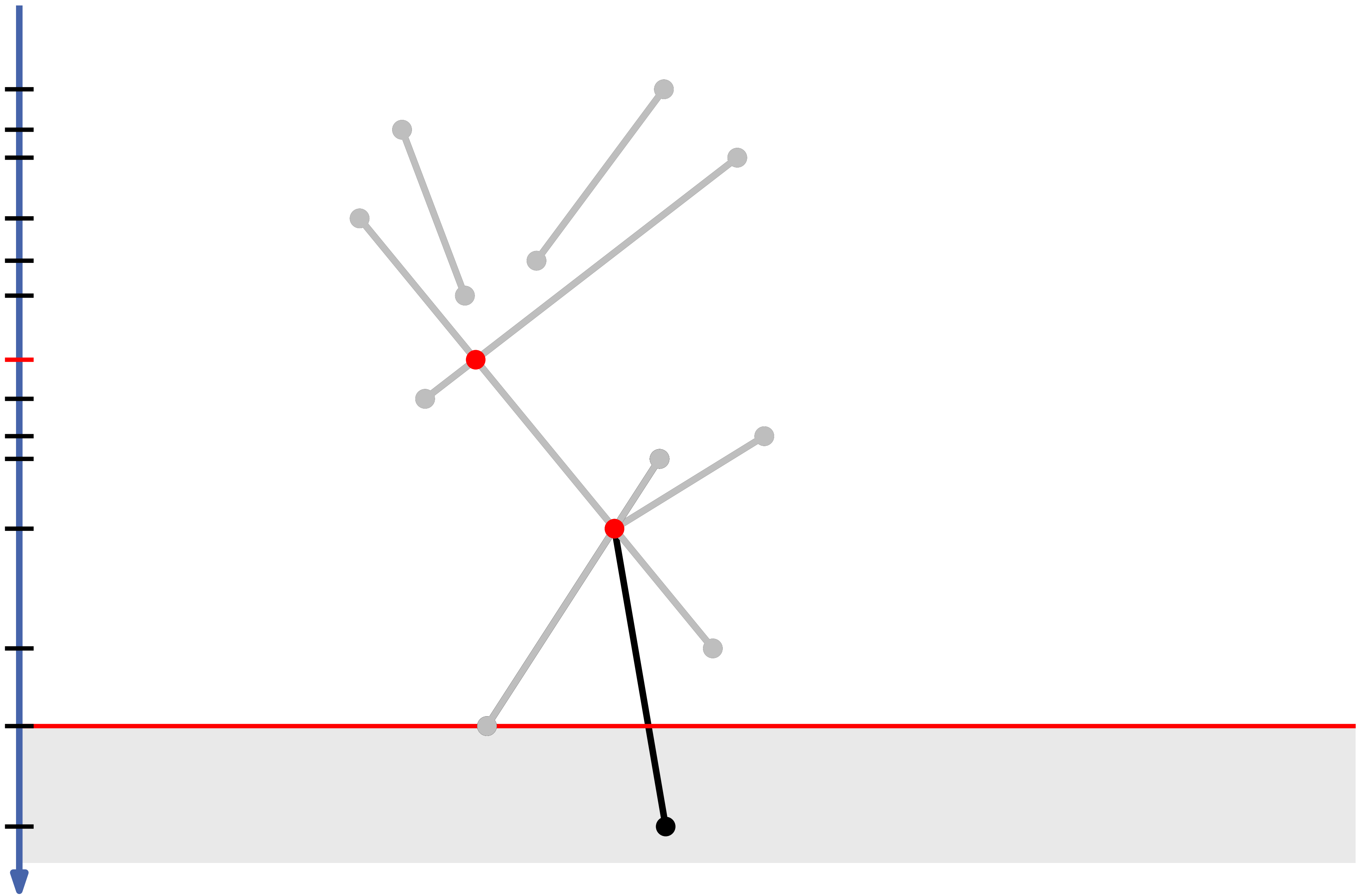
Das Sweep-Line Verfahren: Beispiel



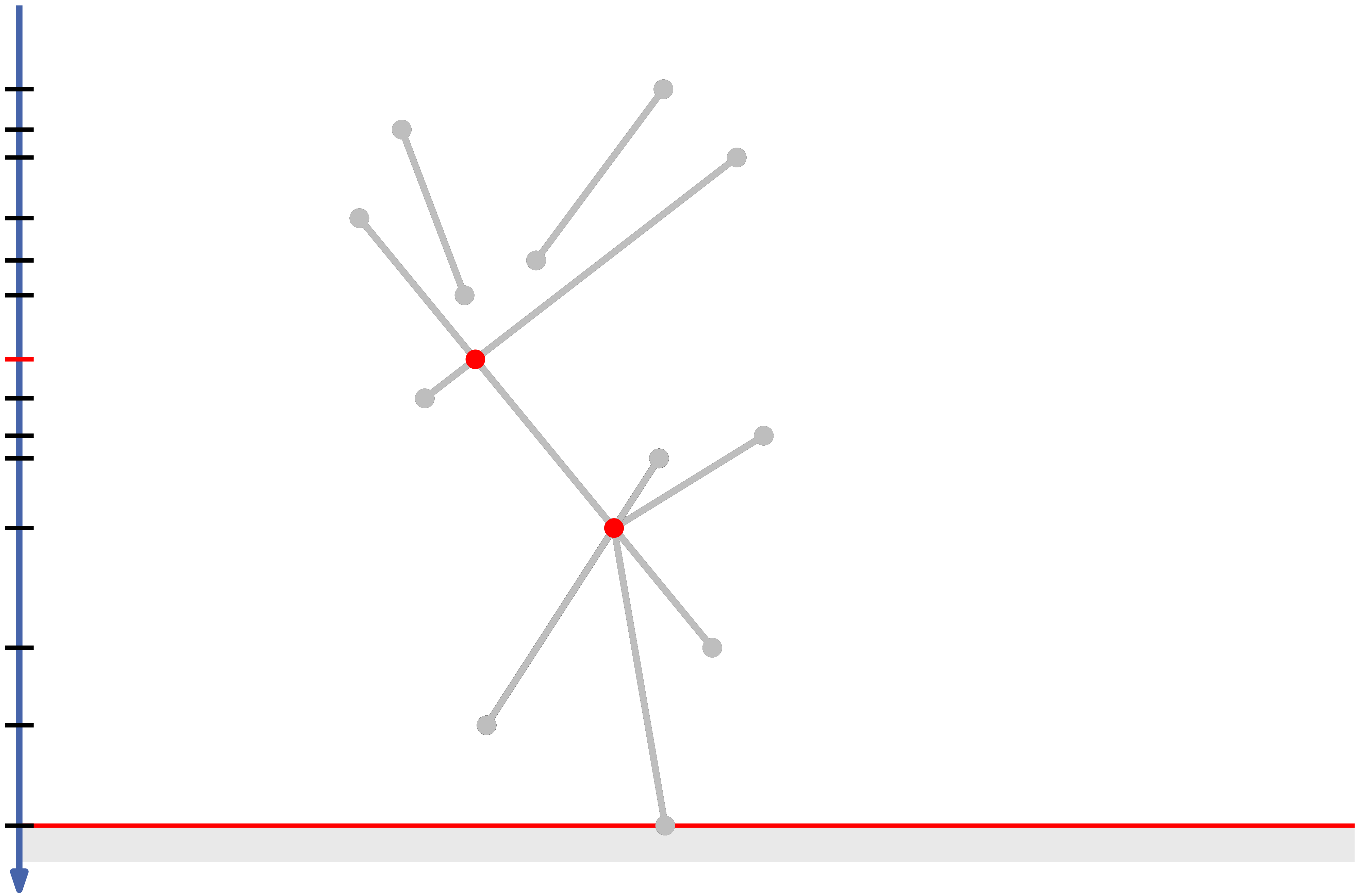
Das Sweep-Line Verfahren: Beispiel



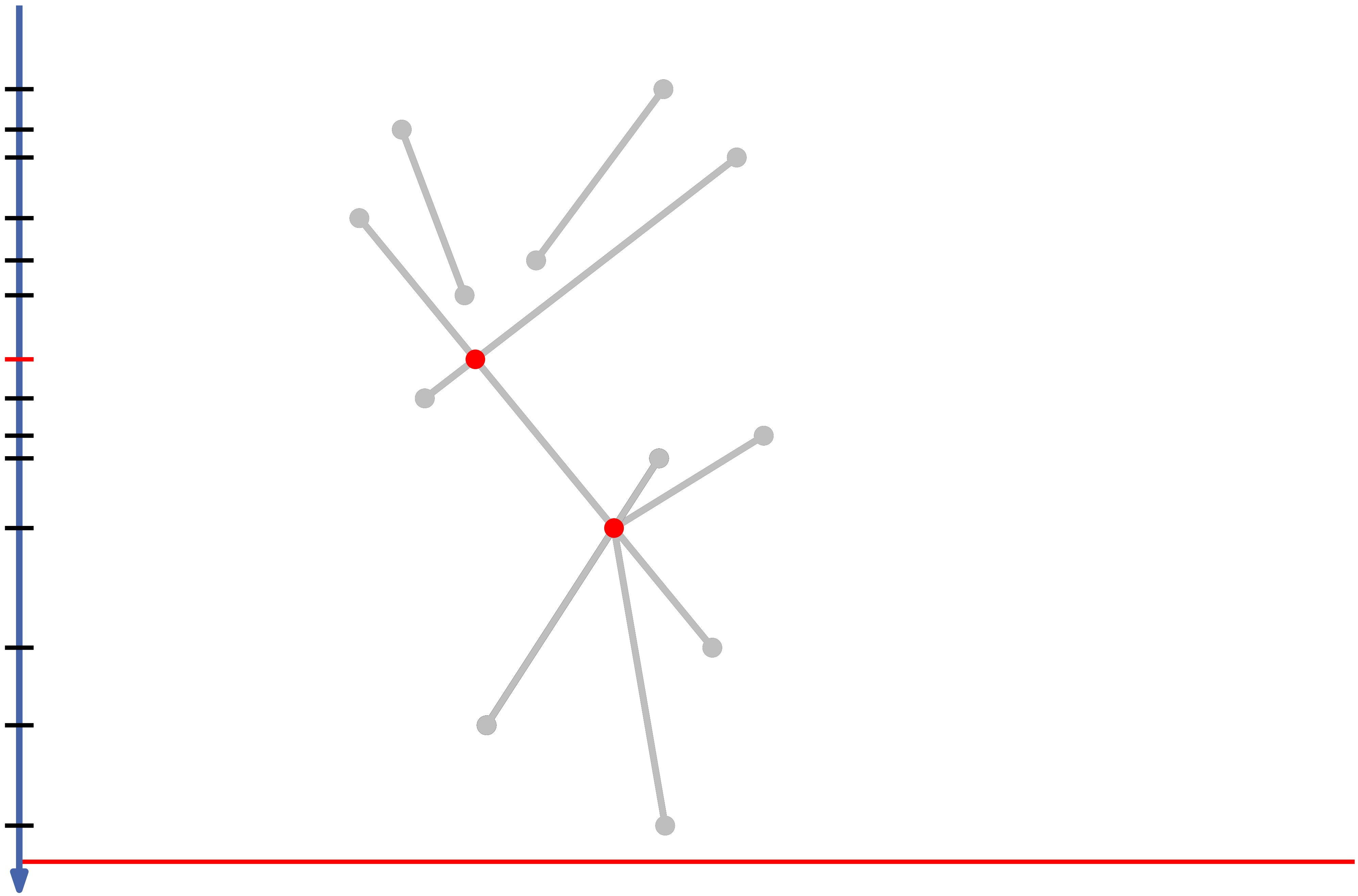
Das Sweep-Line Verfahren: Beispiel



Das Sweep-Line Verfahren: Beispiel



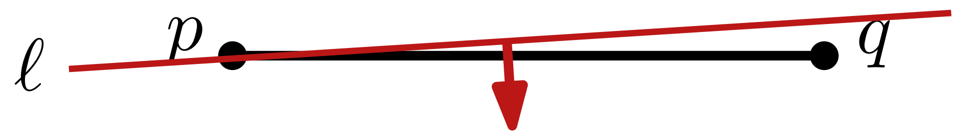
Das Sweep-Line Verfahren: Beispiel



Datenstrukturen

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

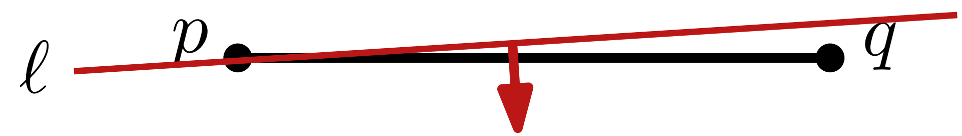
- definiere $p \prec q \quad \Leftrightarrow_{\text{def.}} \quad y_p > y_q \vee (y_p = y_q \wedge x_p < x_q)$



Datenstrukturen

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

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

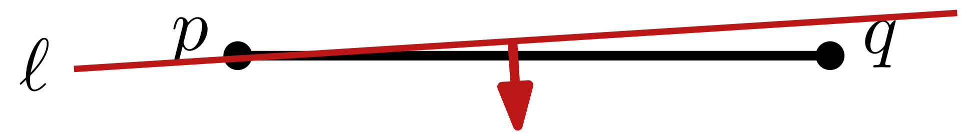


- speichere Events sortiert nach $\prec$ in **balanciertem binärem Suchbaum**
  $\rightarrow$ AVL-Baum, Rot-Schwarz-Baum, ...

Datenstrukturen

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

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

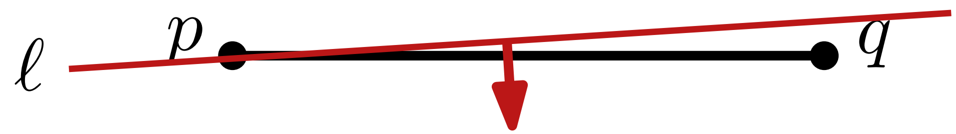


- speichere Events sortiert nach $\prec$ in **balanciertem binärem Suchbaum**
  $\rightarrow$ AVL-Baum, Rot-Schwarz-Baum, . . .
- Operationen `insert`, `delete` und `nextEvent` in $O(\log|\mathcal{Q}|)$ Zeit

Datenstrukturen

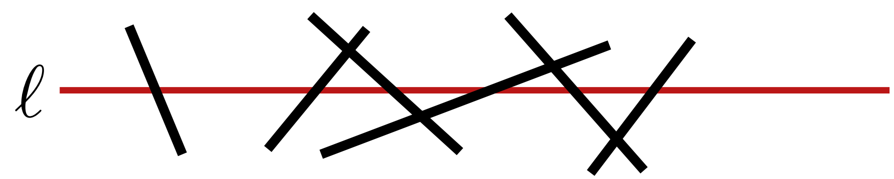
1.) Event Queue $\mathcal{Q}$

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



- speichere Events sortiert nach $\prec$ in **balanciertem binärem Suchbaum**
  $\rightarrow$ AVL-Baum, Rot-Schwarz-Baum, ...
- Operationen `insert`, `delete` und `nextEvent` in $O(\log |\mathcal{Q}|)$ Zeit

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

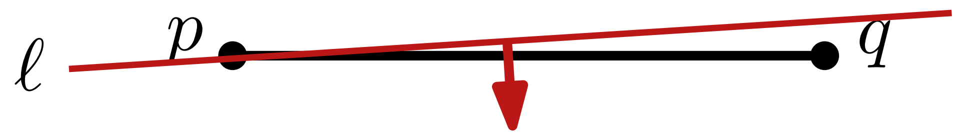


- speichere von ℓ geschnittene Strecken geordnet von links nach rechts

Datenstrukturen

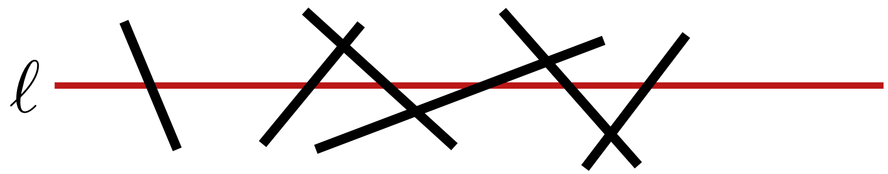
1.) Event Queue $\mathcal{Q}$

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



- speichere Events sortiert nach $\prec$ in **balanciertem binärem Suchbaum**
  $\rightarrow$ AVL-Baum, Rot-Schwarz-Baum, ...
- Operationen `insert`, `delete` und `nextEvent` in $O(\log |\mathcal{Q}|)$ Zeit

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

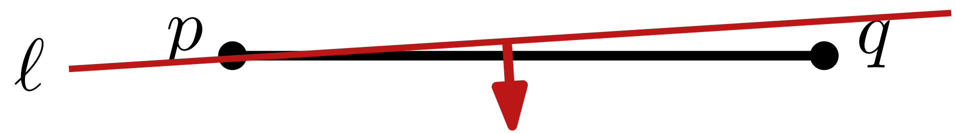


- speichere von ℓ geschnittene Strecken geordnet von links nach rechts
- benötigte Operationen `insert`, `delete`, `findNeighbor`

Datenstrukturen

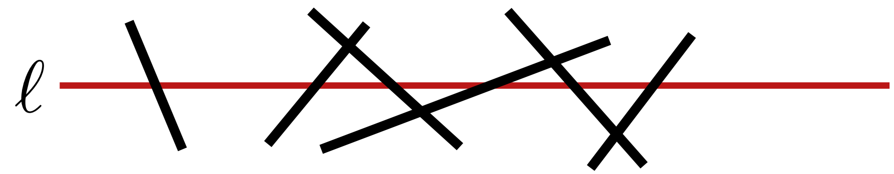
1.) Event Queue $\mathcal{Q}$

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



- speichere Events sortiert nach $\prec$ in **balanciertem binärem Suchbaum**
  $\rightarrow$ AVL-Baum, Rot-Schwarz-Baum, ...
- Operationen `insert`, `delete` und `nextEvent` in $O(\log |\mathcal{Q}|)$ Zeit

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



- speichere von ℓ geschnittene Strecken geordnet von links nach rechts
- benötigte Operationen `insert`, `delete`, `findNeighbor`
- ebenfalls balancierter binärer Suchbaum mit Strecken in den Blättern!

Algorithmus

FindIntersections(S)

Input: Menge S von Strecken

Output: Menge aller Schnittpunkte mit zugeh. Strecken

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

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)

Algorithmus

FindIntersections(S)

Input: Menge S von Strecken
Output: Menge aller Schnittpunkte mit zugeh. Strecken
$\mathcal{Q} \leftarrow \emptyset; \quad \mathcal{T} \leftarrow \emptyset$
foreach $s \in S$ **do**
    $\mathcal{Q}$.insert(upperEndPoint(s))
    $\mathcal{Q}$.insert(lowerEndPoint(s))

Was passiert mit Duplikaten?

while $\mathcal{Q} \neq \emptyset$ **do**
    $p \leftarrow \mathcal{Q}$.nextEvent()
    $\mathcal{Q}$.deleteEvent(p)
    handleEvent(p)

Algorithmus

FindIntersections(S)

Input: Menge S von Strecken
Output: Menge aller Schnittpunkte mit zugeh. Strecken

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

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)

Was passiert mit Duplikaten?

Hier versteckt sich der Kern des Algorithmus!

Algorithmus

handleEvent(p)

$U(p) \leftarrow$ Strecken mit p oberer Endpunkt
$L(p) \leftarrow$ Strecken mit p unterer Endpunkt
$C(p) \leftarrow$ Strecken mit p innerer Punkt
if $|U(p) \cup L(p) \cup C(p)| \geq 2$ **then**
  gebe p und $U(p) \cup L(p) \cup C(p)$ aus

entferne $L(p) \cup C(p)$ aus $\mathcal{T}$
füge $U(p) \cup C(p)$ in $\mathcal{T}$ ein
if $U(p) \cup C(p) = \emptyset$ **then** $//s_l$ und s_r Nachbarn von p in $\mathcal{T}$
  $\mathcal{Q} \leftarrow$ prüfe s_l und s_r auf Schnitt unterhalb p
else $//s'$ und s'' linkeste und rechteste Strecke in $U(p) \cup C(p)$
  $\mathcal{Q} \leftarrow$ prüfe s_l und s' auf Schnitt unterhalb p
  $\mathcal{Q} \leftarrow$ prüfe s_r und s'' auf Schnitt unterhalb p

Algorithmus

handleEvent(p)

$U(p) \leftarrow$ Strecken mit p oberer Endpunkt

mit p in $\mathcal{Q}$ gespeichert

$L(p) \leftarrow$ Strecken mit p unterer Endpunkt
$C(p) \leftarrow$ Strecken mit p innerer Punkt
if $|U(p) \cup L(p) \cup C(p)| \geq 2$ **then**
  gebe p und $U(p) \cup L(p) \cup C(p)$ aus

entferne $L(p) \cup C(p)$ aus $\mathcal{T}$
füge $U(p) \cup C(p)$ in $\mathcal{T}$ ein
if $U(p) \cup C(p) = \emptyset$ **then** //s_l und s_r Nachbarn von p in $\mathcal{T}$
  $\mathcal{Q} \leftarrow$ prüfe s_l und s_r auf Schnitt unterhalb p
else //s' und s'' linkeste und rechteste Strecke in $U(p) \cup C(p)$
  $\mathcal{Q} \leftarrow$ prüfe s_l und s' auf Schnitt unterhalb p
  $\mathcal{Q} \leftarrow$ prüfe s_r und s'' auf Schnitt unterhalb p

Algorithmus

handleEvent(p)

$U(p) \leftarrow$ Strecken mit p oberer Endpunkt

mit p in $\mathcal{Q}$ gespeichert

$L(p) \leftarrow$ Strecken mit p unterer Endpunkt

$C(p) \leftarrow$ Strecken mit p innerer Punkt

Nachbarn in $\mathcal{T}$

if $|U(p) \cup L(p) \cup C(p)| \geq 2$ **then**

  gebe p und $U(p) \cup L(p) \cup C(p)$ aus

entferne $L(p) \cup C(p)$ aus $\mathcal{T}$

füge $U(p) \cup C(p)$ in $\mathcal{T}$ ein

if $U(p) \cup C(p) = \emptyset$ **then** $//s_l$ und s_r Nachbarn von p in $\mathcal{T}$

  $\mathcal{Q} \leftarrow$ prüfe s_l und s_r auf Schnitt unterhalb p

else $//s'$ und s'' linkeste und rechteste Strecke in $U(p) \cup C(p)$

  $\mathcal{Q} \leftarrow$ prüfe s_l und s' auf Schnitt unterhalb p

  $\mathcal{Q} \leftarrow$ prüfe s_r und s'' auf Schnitt unterhalb p

Algorithmus

handleEvent(p)

$U(p) \leftarrow$ Strecken mit p oberer Endpunkt

mit p in $\mathcal{Q}$ gespeichert

$L(p) \leftarrow$ Strecken mit p unterer Endpunkt

$C(p) \leftarrow$ Strecken mit p innerer Punkt

Nachbarn in $\mathcal{T}$

if $|U(p) \cup L(p) \cup C(p)| \geq 2$ **then**

  gebe p und $U(p) \cup L(p) \cup C(p)$ aus

entferne $L(p) \cup C(p)$ aus $\mathcal{T}$

füge $U(p) \cup C(p)$ in $\mathcal{T}$ ein

Entfernen und Einfügen dreht $C(p)$ um

if $U(p) \cup C(p) = \emptyset$ **then** $//s_l$ und s_r Nachbarn von p in $\mathcal{T}$

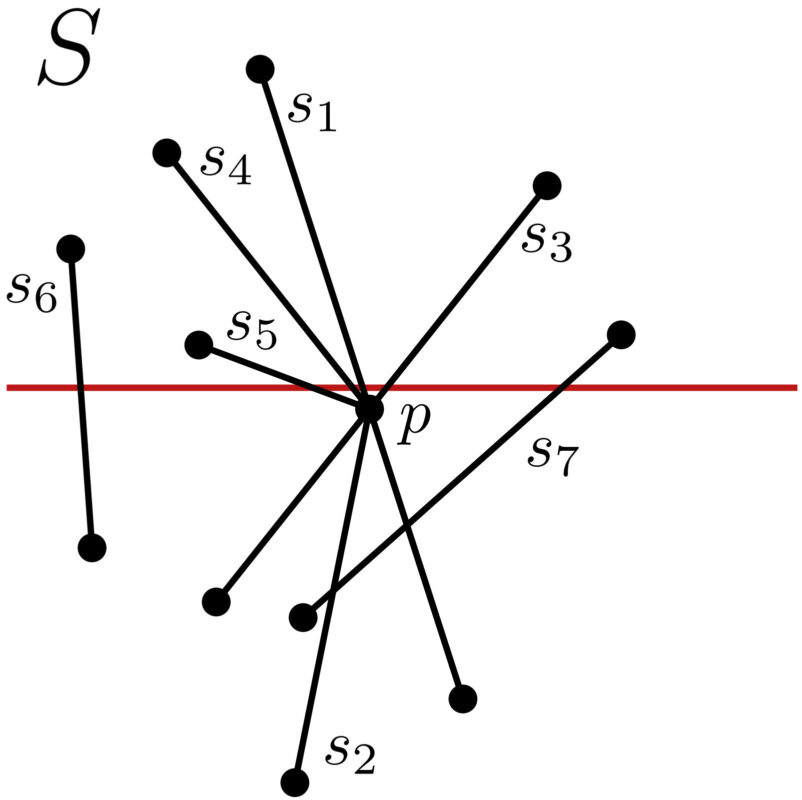
  $\mathcal{Q} \leftarrow$ prüfe s_l und s_r auf Schnitt unterhalb p

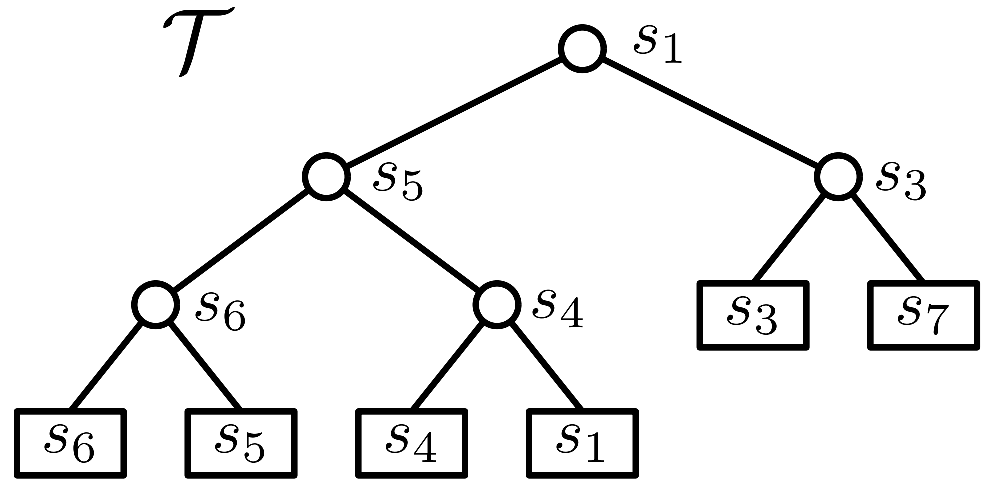
else $//s'$ und s'' linkeste und rechteste Strecke in $U(p) \cup C(p)$

  $\mathcal{Q} \leftarrow$ prüfe s_l und s' auf Schnitt unterhalb p

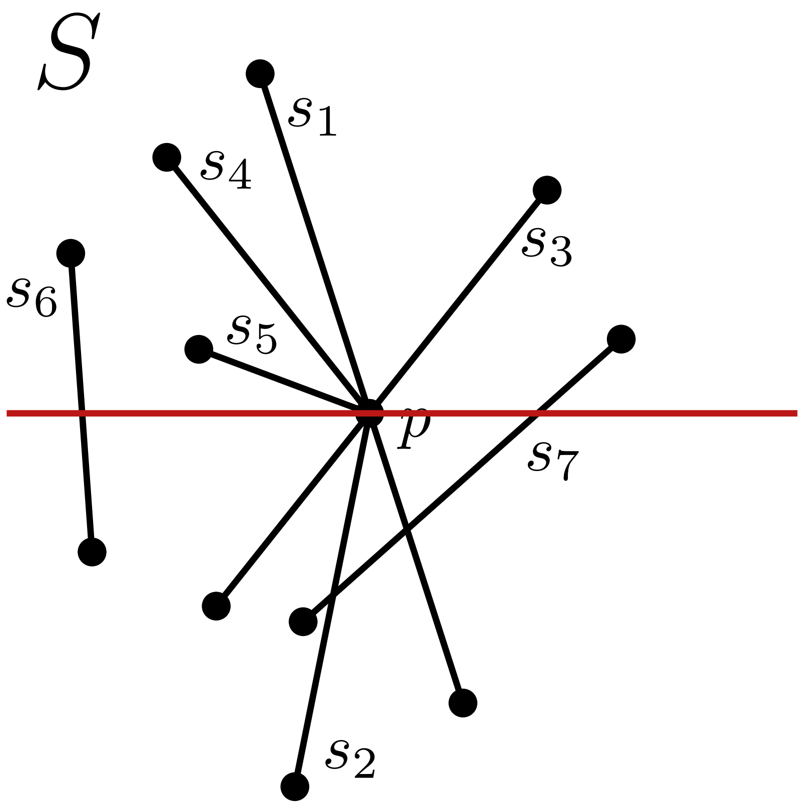
  $\mathcal{Q} \leftarrow$ prüfe s_r und s'' auf Schnitt unterhalb p

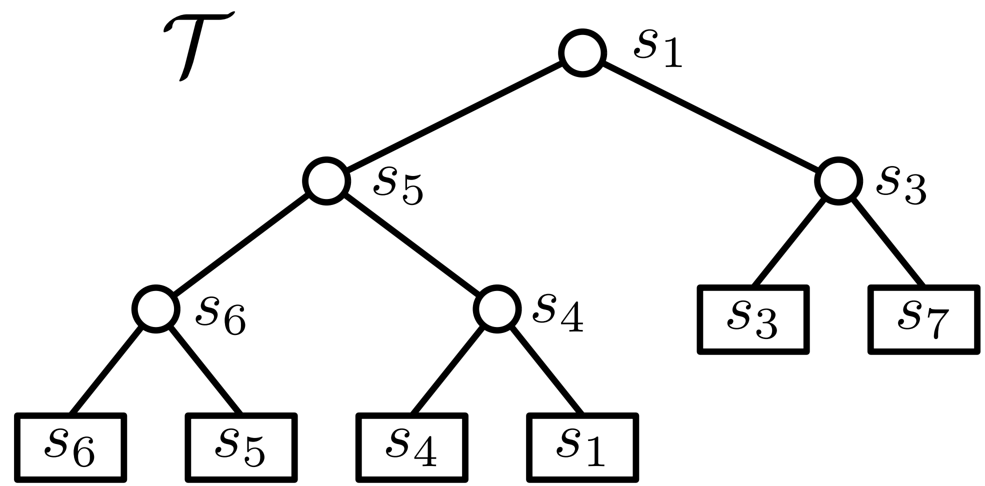
Was passiert genau?





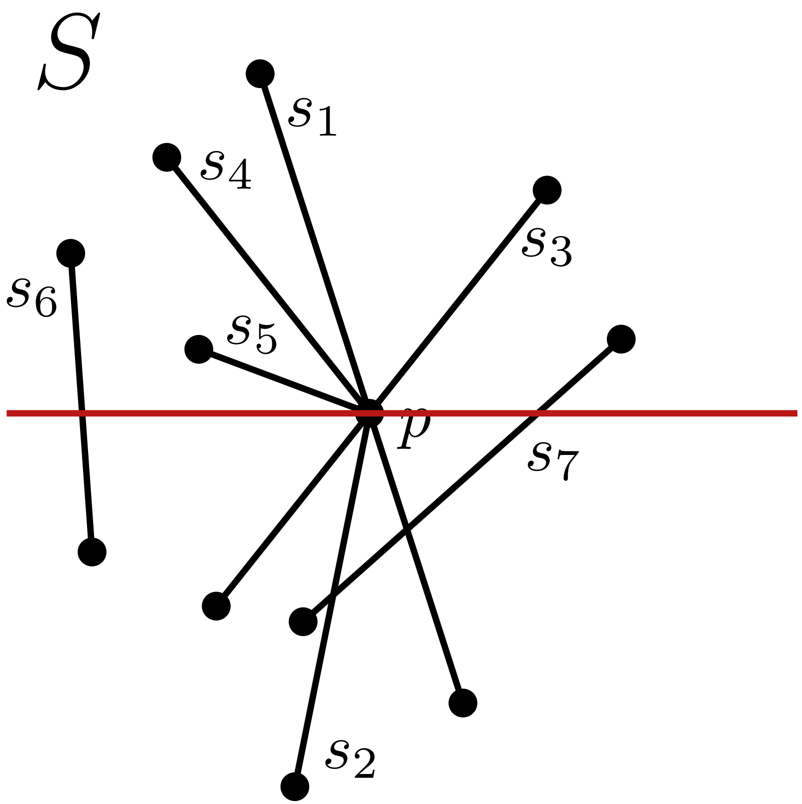
Was passiert genau?

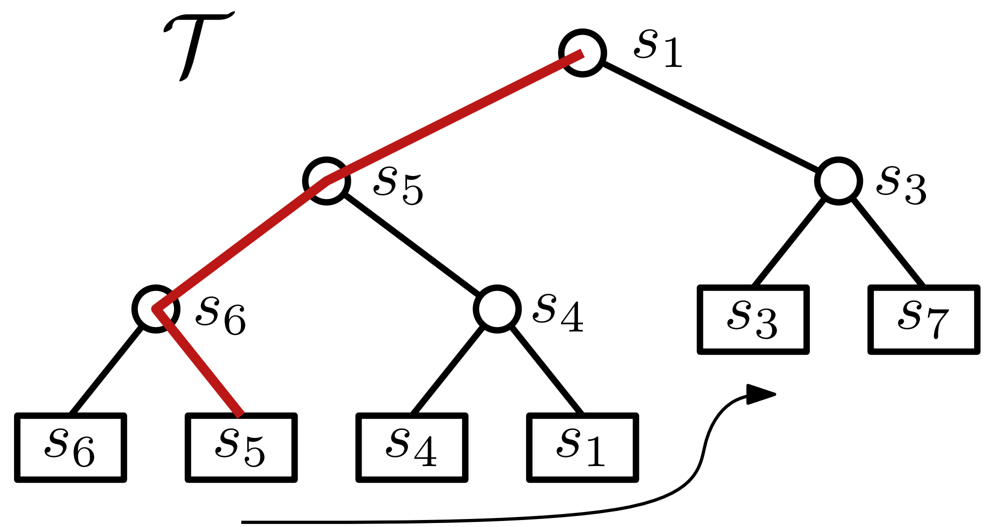




$U(p) = \{s_2\}$
$L(p) =$
$C(p) =$

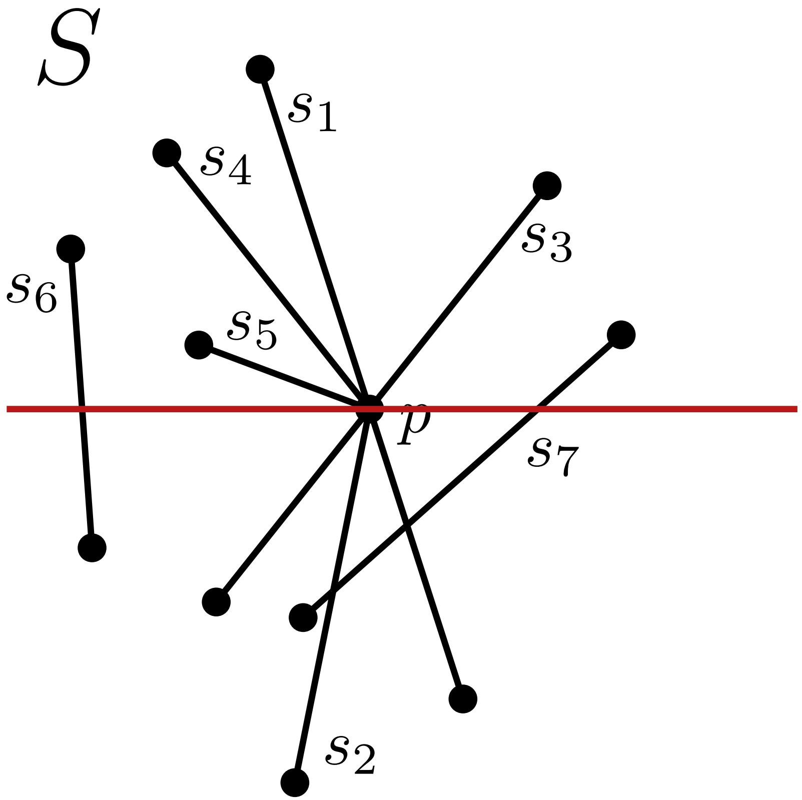
Was passiert genau?

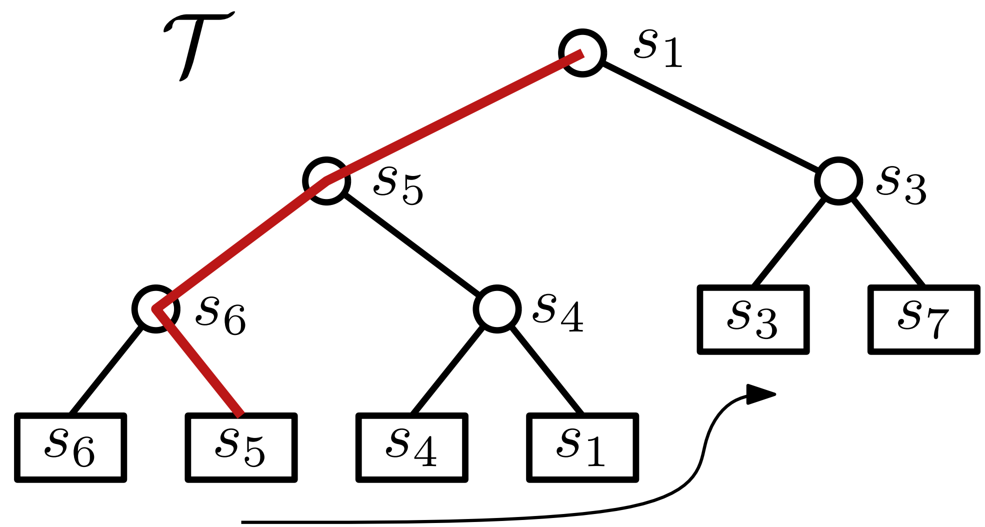




$U(p) = \{s_2\}$
$L(p) =$
$C(p) =$

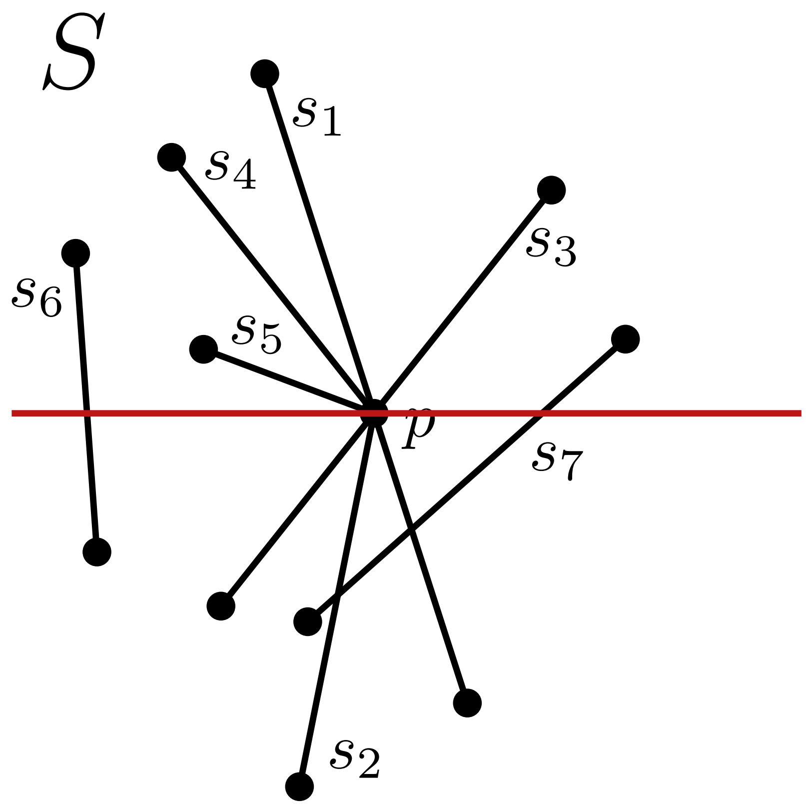
Was passiert genau?

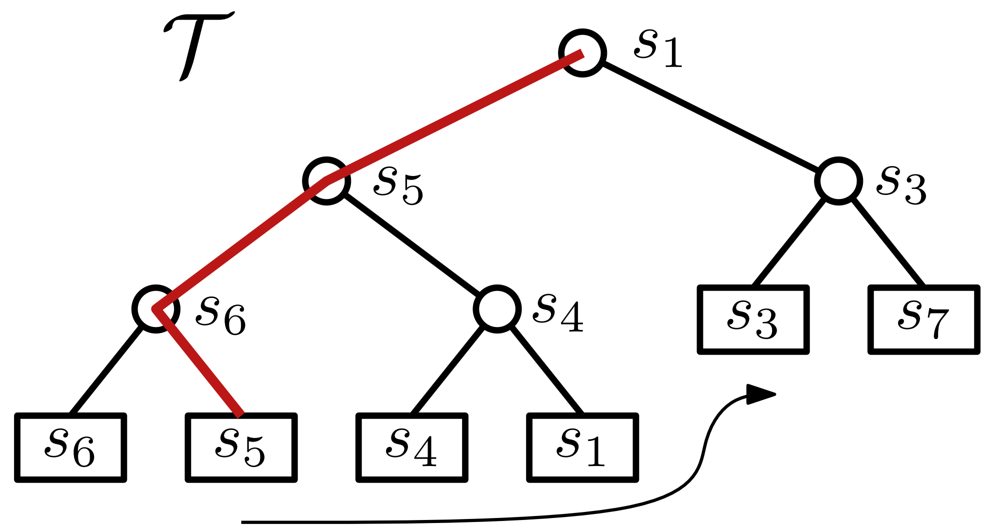




$$U(p) = \{s_2\}$$
$$L(p) = \{s_4, s_5\}$$
$$C(p) = \{s_1, s_3\}$$

Was passiert genau?

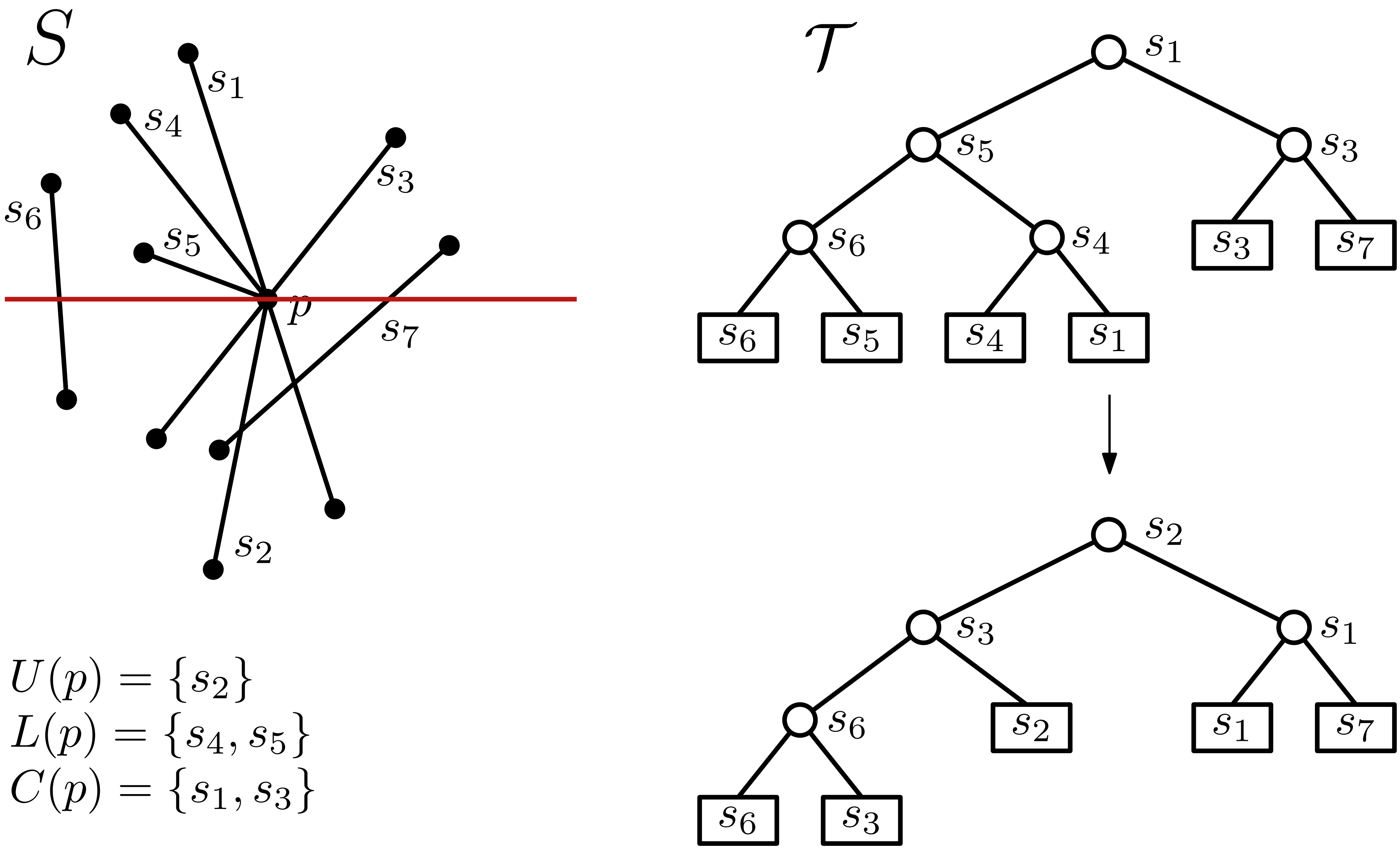




$U(p) = \{s_2\}$
$L(p) = \{s_4, s_5\}$
$C(p) = \{s_1, s_3\}$

Gib $(p, \{s_1, s_2, s_3, s_4, s_5\})$ aus

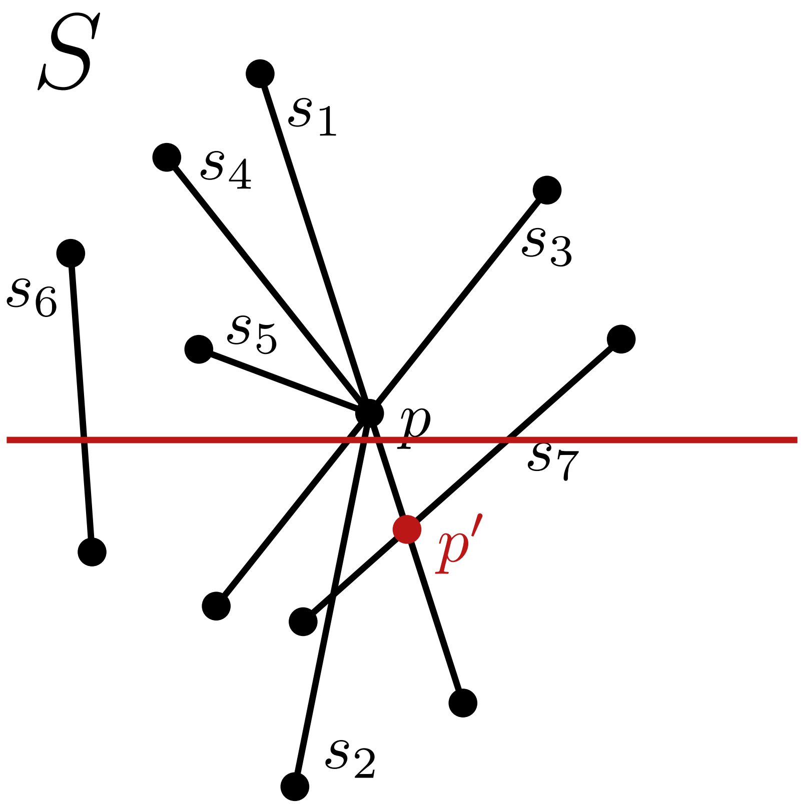
Was passiert genau?

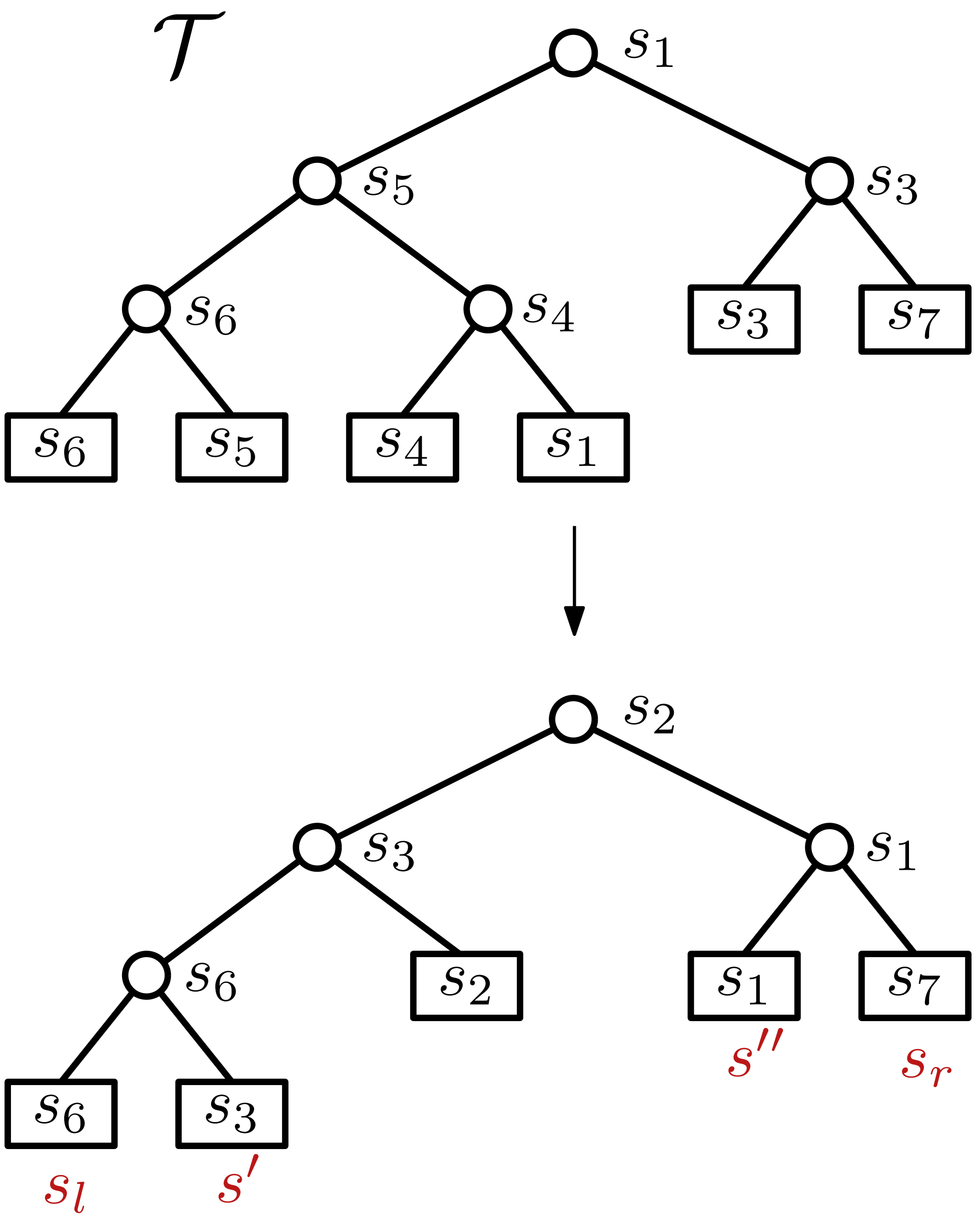


$U(p) = \{s_2\}$
$L(p) = \{s_4, s_5\}$
$C(p) = \{s_1, s_3\}$

Lösche $L(p) \cup C(p)$; füge $U(p) \cup C(p)$ ein

Was passiert genau?





$U(p) = \{s_2\}$
$L(p) = \{s_4, s_5\}$
$C(p) = \{s_1, s_3\}$

Füge Event $p' = s_1 \times s_7$ in $\mathcal{Q}$ ein

Korrektheit

Lemma 1: Algorithmus FindIntersections findet alle Schnittpunkte und die beteiligten Strecken.

Korrektheit

Lemma 1: Algorithmus FindIntersections findet alle Schnittpunkte und die beteiligten Strecken.

Beweis:

Induktion über die Reihenfolge der Events.

Sei p ein Schnittpunkt und alle Schnittpunkte $q \prec p$ seien bereits korrekt berechnet.

Fall 1: p ist Streckenendpunkt

- p wurde am Anfang in $\mathcal{Q}$ eingefügt
- $U(p)$ an p gespeichert
- $L(p)$ und $C(p)$ in $\mathcal{T}$ vorhanden

Korrektheit

Lemma 1: Algorithmus FindIntersections findet alle Schnittpunkte und die beteiligten Strecken.

Beweis:

Induktion über die Reihenfolge der Events.

Sei p ein Schnittpunkt und alle Schnittpunkte $q \prec p$ seien bereits korrekt berechnet.

Fall 1: p ist Streckenendpunkt

- p wurde am Anfang in $\mathcal{Q}$ eingefügt
- $U(p)$ an p gespeichert
- $L(p)$ und $C(p)$ in $\mathcal{T}$ vorhanden

Fall 2: p ist kein Streckenendpunkt

Überlegen Sie warum p in $\mathcal{Q}$ sein muss!

Laufzeitanalyse

FindIntersections(S)

Input: Menge S von Strecken

Output: Menge aller Schnittpunkte mit zugeh. Strecken

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

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)

Laufzeitanalyse

FindIntersections(S)

Input: Menge S von Strecken

Output: Menge aller Schnittpunkte mit zugeh. Strecken

$\mathcal{Q} \leftarrow \emptyset$; $\mathcal{T} \leftarrow \emptyset$ $O(1)$

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)

Laufzeitanalyse

FindIntersections(S)

Input: Menge S von Strecken

Output: Menge aller Schnittpunkte mit zugeh. Strecken

$\mathcal{Q} \leftarrow \emptyset$; $\mathcal{T} \leftarrow \emptyset$ $O(1)$

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

  $\mathcal{Q}$.insert(upperEndPoint(s)) $O(n \log n)$

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

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

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

  $\mathcal{Q}$.deleteEvent(p)

  handleEvent(p)

Laufzeitanalyse

FindIntersections(S)

Input: Menge S von Strecken

Output: Menge aller Schnittpunkte mit zugeh. Strecken

$\mathcal{Q} \leftarrow \emptyset;\quad \mathcal{T} \leftarrow \emptyset$ $\qquad O(1)$

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

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

  $\mathcal{Q}$.insert(lowerEndPoint(s)) $\qquad O(n \log n)$

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

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

  $\mathcal{Q}$.deleteEvent(p) $\qquad O(\log |\mathcal{Q}|)$

  handleEvent(p)

Laufzeitanalyse

FindIntersections(S)

Input: Menge S von Strecken

Output: Menge aller Schnittpunkte mit zugeh. Strecken

$\mathcal{Q} \leftarrow \emptyset; \quad \mathcal{T} \leftarrow \emptyset$ $\qquad O(1)$

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

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

$O(n \log n)$

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

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

  $O(\log |\mathcal{Q}|)$

- handleEvent(p)

?

Laufzeitanalyse

handleEvent(p)

$U(p) \leftarrow$ Strecken mit p oberer Endpunkt
$L(p) \leftarrow$ Strecken mit p unterer Endpunkt
$C(p) \leftarrow$ Strecken mit p innerer Punkt
if $|U(p) \cup L(p) \cup C(p)| \geq 2$ **then**
  gebe p und $U(p) \cup L(p) \cup C(p)$ aus

entferne $L(p) \cup C(p)$ aus $\mathcal{T}$
füge $U(p) \cup C(p)$ in $\mathcal{T}$ ein
if $U(p) \cup C(p) = \emptyset$ **then** $//s_l$ und s_r Nachbarn von p in $\mathcal{T}$
  $\mathcal{Q} \leftarrow$ prüfe s_l und s_r auf Schnitt unterhalb p
else $//s'$ und s'' linkeste und rechteste Strecke in $U(p) \cup C(p)$
  $\mathcal{Q} \leftarrow$ prüfe s_l und s' auf Schnitt unterhalb p
  $\mathcal{Q} \leftarrow$ prüfe s_r und s'' auf Schnitt unterhalb p

Lemma 2: Algorithmus FindIntersections hat eine Laufzeit von $O(n \log n + I \log n)$, wobei I die Anzahl der Schnittpunkte ist.

Zusammenfassung

Satz 1: Sei S eine Menge von n Strecken in der Ebene. Dann können alle Schnittpunkte in S zusammen mit den beteiligten Strecken in $O((n+I)\log n)$ Zeit und $O(?)$ Platz ausgegeben werden.

Zusammenfassung

Satz 1: Sei S eine Menge von n Strecken in der Ebene. Dann können alle Schnittpunkte in S zusammen mit den beteiligten Strecken in $O((n+I)\log n)$ Zeit und $O(?)$ Platz ausgegeben werden.

Beweis:

- Korrektheit ✓
- Laufzeit ✓
- Speicherplatz

Zusammenfassung

Satz 1: Sei S eine Menge von n Strecken in der Ebene. Dann können alle Schnittpunkte in S zusammen mit den beteiligten Strecken in $O((n+I)\log n)$ Zeit und $O(?)$ Platz ausgegeben werden.

Beweis:

- Korrektheit ✓
- Laufzeit ✓
- Speicherplatz

Überlegen Sie wie viel Platz die Datenstrukturen benötigen!

Zusammenfassung

Satz 1: Sei S eine Menge von n Strecken in der Ebene. Dann können alle Schnittpunkte in S zusammen mit den beteiligten Strecken in $O((n+I)\log n)$ Zeit und $O(n)$ Platz ausgegeben werden.

Beweis:

- Korrektheit ✓
- Laufzeit ✓
- Speicherplatz

Überlegen Sie wie viel Platz die Datenstrukturen benötigen!

  – $\mathcal{T}$ maximal n Elemente
  – $\mathcal{Q}$ maximal $O(n+I)$ Elemente
  – Reduktion von $\mathcal{Q}$ auf $O(n)$ Platz: s. Übung

Diskussion

Karlsruhe Institute of Technology

Ist der Sweep-Line Algorithmus immer besser als der naive?

Diskussion

Ist der Sweep-Line Algorithmus immer besser als der naive?

Nein, denn falls $I \in \Omega(n^2)$ hat der Algorithmus eine Laufzeit von $O(n^2 \log n)$.

Diskussion

Ist der Sweep-Line Algorithmus immer besser als der naive?

Nein, denn falls $I \in \Omega(n^2)$ hat der Algorithmus eine Laufzeit von $O(n^2 \log n)$.

Geht es noch besser?

Diskussion

Ist der Sweep-Line Algorithmus immer besser als der naive?

Nein, denn falls $I \in \Omega(n^2)$ hat der Algorithmus eine Laufzeit von $O(n^2 \log n)$.

Geht es noch besser?

Ja, in $\Theta(n \log n + I)$ Zeit und $\Theta(n)$ Platz [Balaban, 1995].

Diskussion

Ist der Sweep-Line Algorithmus immer besser als der naive?

Nein, denn falls $I \in \Omega(n^2)$ hat der Algorithmus eine Laufzeit von $O(n^2 \log n)$.

Geht es noch besser?

Ja, in $\Theta(n \log n + I)$ Zeit und $\Theta(n)$ Platz [Balaban, 1995].

Wie löst man damit das Kartenüberlagerungs-Problem?

Diskussion

Ist der Sweep-Line Algorithmus immer besser als der naive?

Nein, denn falls $I \in \Omega(n^2)$ hat der Algorithmus eine Laufzeit von $O(n^2 \log n)$.

Geht es noch besser?

Ja, in $\Theta(n \log n + I)$ Zeit und $\Theta(n)$ Platz [Balaban, 1995].

Wie löst man damit das Kartenüberlagerungs-Problem?

Unter Verwendung einer geeigneten Datenstruktur (**doppelt-verkettete Kantenliste**) für planare Graphen lässt sich in ebenfalls $O((n + I) \log n)$ Zeit die überlagerte Unterteilung der Karte berechnen.
(Details s. Kap. 2.3 im Buch)

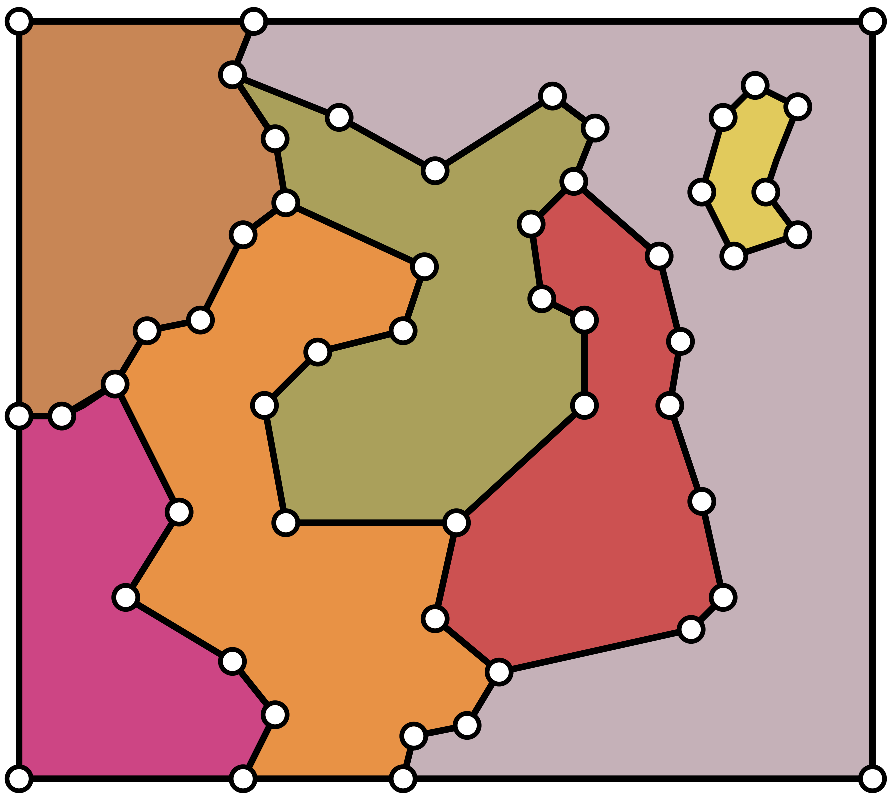
Datenstruktur für Landkarten

- (politische) Landkarte entspricht Unterteilung der Ebene in Polygone

Datenstruktur für Landkarten

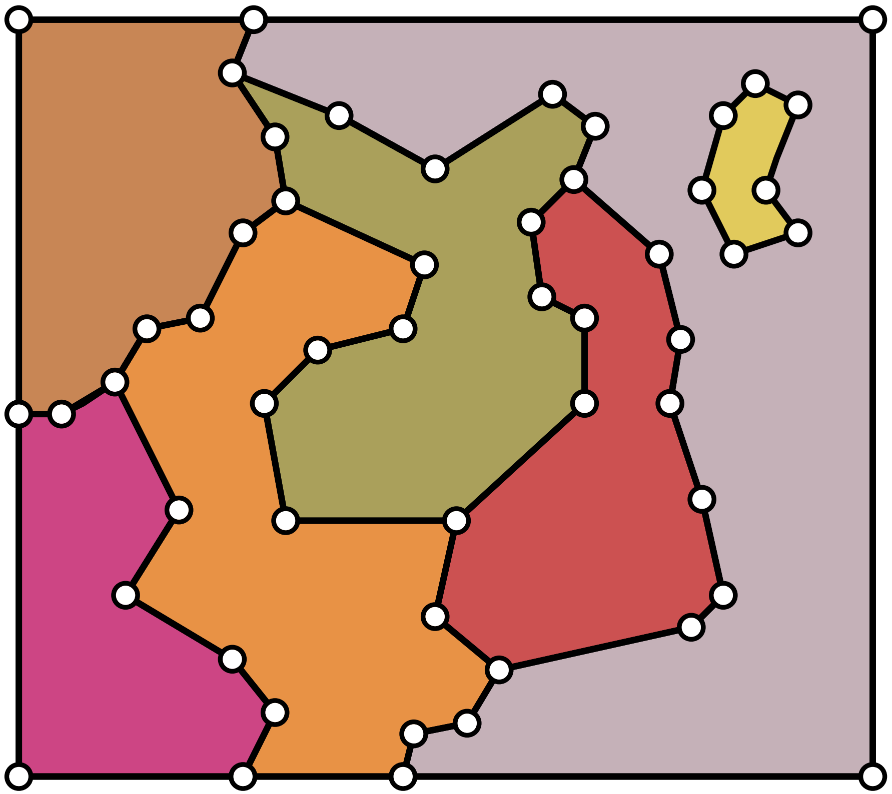
- (politische) Landkarte entspricht Unterteilung der Ebene in Polygone
- Unterteilung entspricht Einbettung eines planaren Graphen mit
  - Knoten
  - Kanten
  - Facetten

Datenstruktur für Landkarten



- (politische) Landkarte entspricht Unterteilung der Ebene in Polygone
- Unterteilung entspricht Einbettung eines planaren Graphen mit
  - Knoten
  - Kanten
  - Facetten

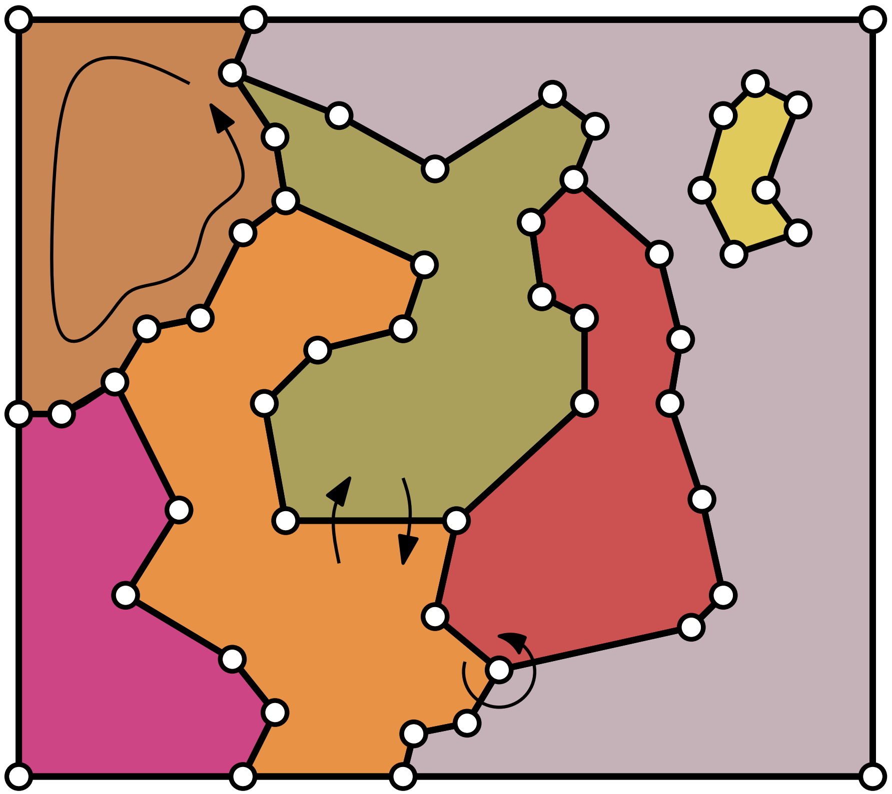
Datenstruktur für Landkarten



- (politische) Landkarte entspricht Unterteilung der Ebene in Polygone
- Unterteilung entspricht Einbettung eines planaren Graphen mit
  - Knoten
  - Kanten
  - Facetten

Welche Operationen sollte eine Datenstruktur für Unterteilungen der Ebene unterstützen?

Datenstruktur für Landkarten



- (politische) Landkarte entspricht Unterteilung der Ebene in Polygone
- Unterteilung entspricht Einbettung eines planaren Graphen mit
  - Knoten
  - Kanten
  - Facetten

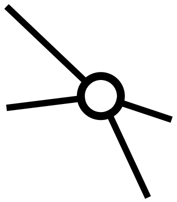
Welche Operationen sollte eine Datenstruktur für Unterteilungen der Ebene unterstützen?

- Kanten einer Facette ablaufen
- via Kante von Facette zu Facette wechseln
- Nachbarknoten in zyklischer Reihenfolge besuchen

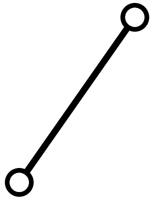
Doppelt verkettete Kantenliste (DCEL)

Zutaten:

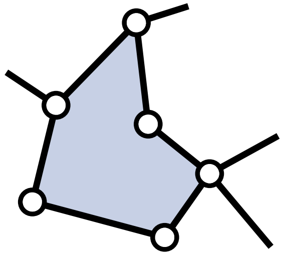
- Knoten



- Kanten



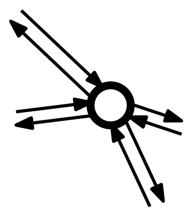
- Facetten



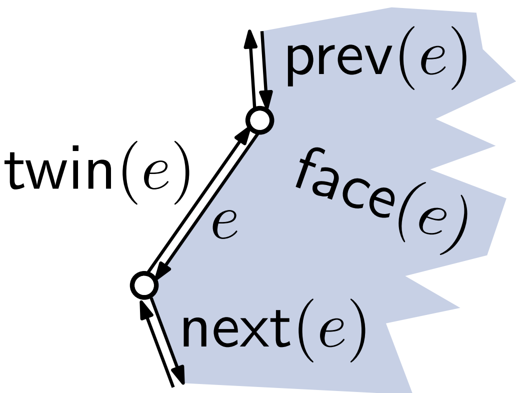
Doppelt verkettete Kantenliste (DCEL)

Zutaten:

- Knoten

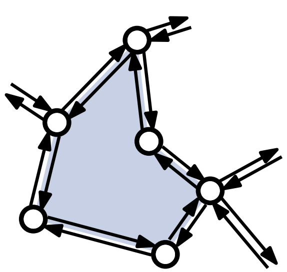


- Kanten = zwei Halbkanten



  - Knoten origin(e)
  - Gegenkante twin(e)
  - Vorgänger prev(e) & Nachfolger next(e)
  - inzidente Facette face(e)

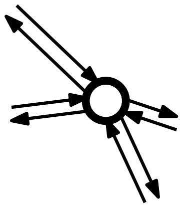
- Facetten



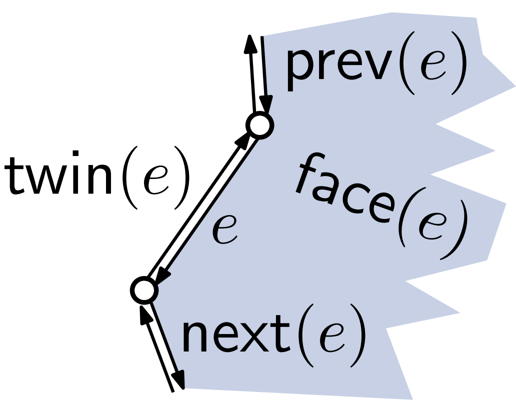
Doppelt verkettete Kantenliste (DCEL)

Zutaten:

- Knoten



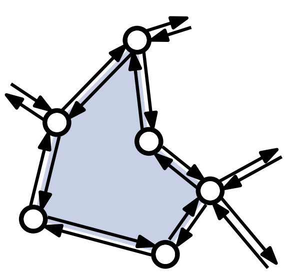
- Kanten = zwei Halbkanten



Was ist mit destination(e)?

- Knoten origin(e)
- Gegenkante twin(e)
- Vorgänger prev(e) & Nachfolger next(e)
- inzidente Facette face(e)

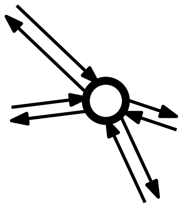
- Facetten



Doppelt verkettete Kantenliste (DCEL)

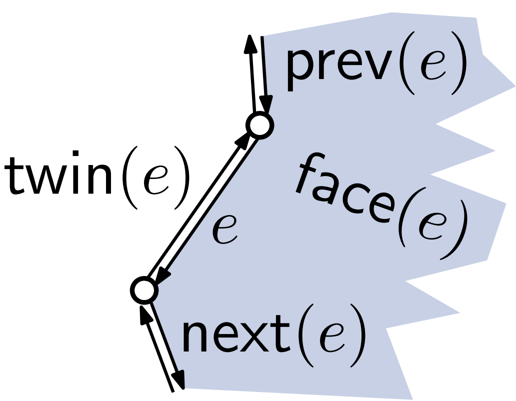
Zutaten:

- Knoten



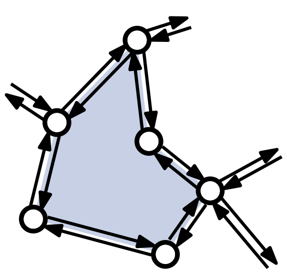
- Koordinaten $(x(v), y(v))$
- (erste) ausgehende Kante edge(v)

- Kanten $=$ zwei Halbkanten



- Knoten origin(e)
- Gegenkante twin(e)
- Vorgänger prev(e) & Nachfolger next(e)
- inzidente Facette face(e)

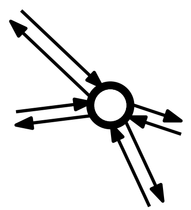
- Facetten



Doppelt verkettete Kantenliste (DCEL)

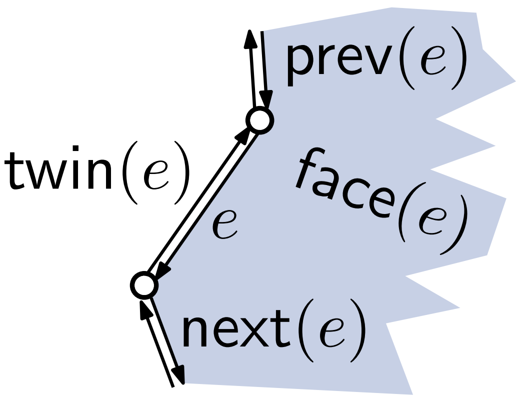
Zutaten:

- Knoten



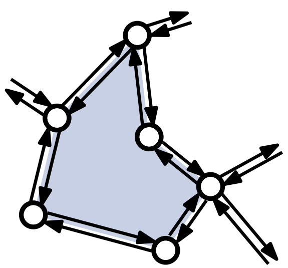
  - Koordinaten $(x(v), y(v))$
  - (erste) ausgehende Kante edge(v)

- Kanten $=$ zwei Halbkanten



  - Knoten origin(e)
  - Gegenkante twin(e)
  - Vorgänger prev(e) & Nachfolger next(e)
  - inzidente Facette face(e)

- Facetten

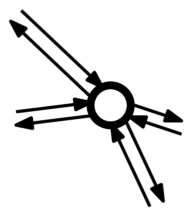


  - Randkante edge(f)

Doppelt verkettete Kantenliste (DCEL)

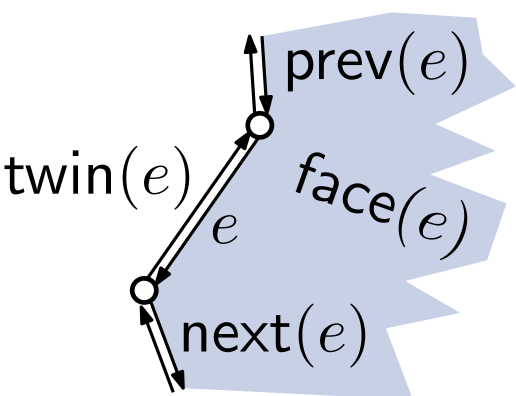
Zutaten:

- Knoten



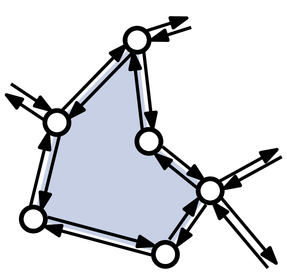
- Koordinaten $(x(v), y(v))$
- (erste) ausgehende Kante edge(v)

- Kanten $=$ zwei Halbkanten



- Knoten origin(e)
- Gegenkante twin(e)
- Vorgänger prev(e) & Nachfolger next(e)
- inzidente Facette face(e)

- Facetten



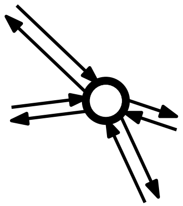
- Randkante edge(f)

Problem dabei?

Doppelt verkettete Kantenliste (DCEL)

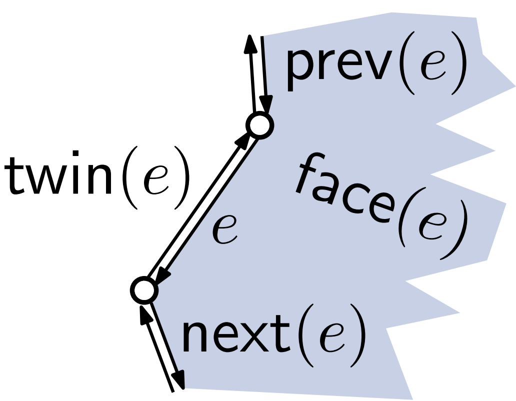
Zutaten:

- Knoten



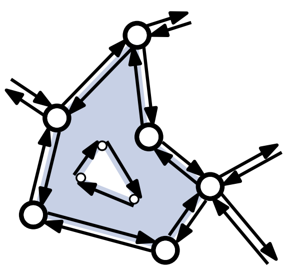
- Koordinaten $(x(v), y(v))$
- (erste) ausgehende Kante edge(v)

- Kanten $=$ zwei Halbkanten



- Knoten origin(e)
- Gegenkante twin(e)
- Vorgänger prev(e) & Nachfolger next(e)
- inzidente Facette face(e)

- Facetten



- Randkante ~~edge(f)~~ outer(f)
- Kantenliste inner(f) für evtl. Löcher

Aufgaben

Welche Gleichheiten stimmen immer?

1. $\mathsf{twin}(\mathsf{twin}(e)) = e$
2. $\mathsf{next}(\mathsf{prev}(e)) = e$
3. $\mathsf{twin}(\mathsf{prev}(\mathsf{twin}(e))) = \mathsf{next}(e)$
4. $\mathsf{face}(e) = \mathsf{face}(\mathsf{next}(e))$

Aufgaben

Welche Gleichheiten stimmen immer?

1. $\text{twin}(\text{twin}(e)) = e$
2. $\text{next}(\text{prev}(e)) = e$
3. $\text{twin}(\text{prev}(\text{twin}(e))) = \text{next}(e)$
4. $\text{face}(e) = \text{face}(\text{next}(e))$

Geben Sie einen Algorithmus an zum

1. Aufzählen aller Nachbarknoten eines Knotens v.
2. Aufzählen aller Nachbarfacetten einer Facette f.
3. Aufzählen aller Facetten mit Knoten auf der äußeren Umrandung.

Ankündigung

Nächste Woche Dienstag (1. Mai) keine Vorlesung.

Nächste Vorlesung **Donnerstag 3. Mai** um 9:45 Uhr in SR131.