Übung Algorithmische Geometrie
Lineare Programmierung

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

Benjamin Niedermann
14.05.2014

Übersicht

Karlsruhe Institute of Technology

Übungsblatt 4

Lineares Programmieren

Definition: Eine Menge linearer Nebenbedingungen H mit einer linearen Zielfunktion c in $\mathbb{R}^d$ bilden ein **lineares Programm** (LP):

$$\text{maximiere} \quad c_1x_1 + c_2x_2 + \cdots + c_dx_d$$

$$\text{unter den NB} \quad \left.\begin{array}{rcl} a_{1,1}x_1 + \cdots + a_{1,d}x_d & \leq & b_1 \\ a_{2,1}x_1 + \cdots + a_{2,d}x_d & \leq & b_2 \\ & \vdots & \\ a_{n,1}x_1 + \cdots + a_{n,d}x_d & \leq & b_n \end{array}\right\} H$$

Lineares Programmieren

Definition: Eine Menge linearer Nebenbedingungen H mit einer linearen Zielfunktion c in $\mathbb{R}^d$ bilden ein **lineares Programm** (LP):

$$\text{maximiere} \quad c_1 x_1 + c_2 x_2 + \cdots + c_d x_d$$

$$\text{unter den NB} \quad \left.\begin{array}{rcl} a_{1,1}x_1 + \cdots + a_{1,d}x_d & \leq & b_1 \\ a_{2,1}x_1 + \cdots + a_{2,d}x_d & \leq & b_2 \\ & \vdots & \\ a_{n,1}x_1 + \cdots + a_{n,d}x_d & \leq & b_n \end{array}\right\} H$$

- H entspricht einer Menge von Halbräumen in $\mathbb{R}^d$.
- Gesucht ist ein Punkt $x \in \bigcap_{h \in H} h$, der $c^T x$ maximiert, also $\max\{c^{\mathrm{T}} x \mid Ax \leq b, x \geq 0\}$.
- Lineare Programmierung ist ein zentrales Optimierungsverfahren im Operations Research.

Algorithmen für LPs

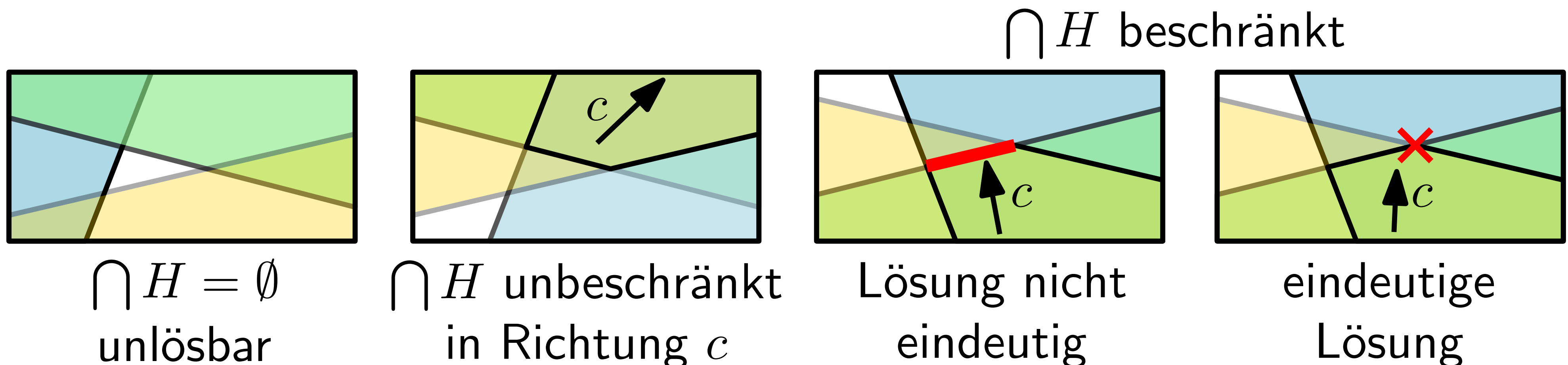
Viele Algorithmen zum Lösen von LPs in der Praxis existieren:

- Simplex-Algorithmus [Dantzig, 1947]
- Ellipsoid-Methode [Khatchiyan, 1979]
- Innere-Punkt-Methode [Karmarkar, 1979]

Funktionieren gut, besonders für große Werte von n (Anzahl Nebenbedingungen) und d (Anzahl Variablen).

Heute: Spezialfall $d = 2$

Möglichkeiten für den Lösungsraum



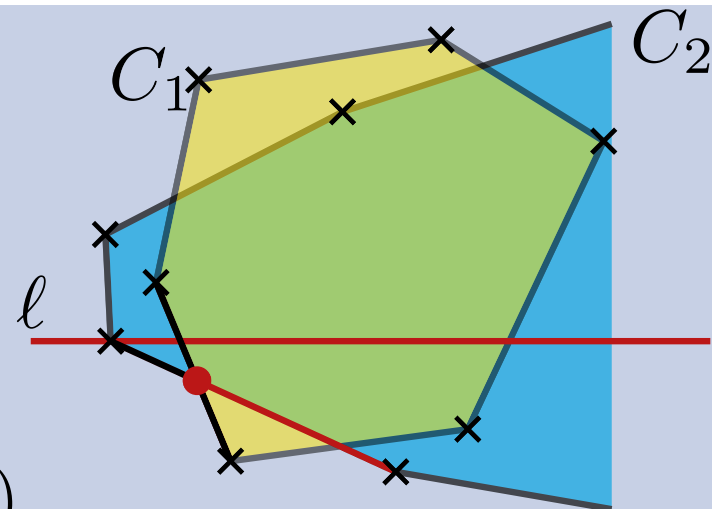
Erste Variante

Idee: Berechne den zulässigen Bereich $\bigcap H$ und suche nach der Ecke p, die $c^T p$ maximiert.

- Halbebenen sind konvex
- Versuche einfachen Divide-and-Conquer Algorithmus

IntersectHalfplanes(H)

if $|H| = 1$ **then**
  $C \leftarrow H$
else
  $(H_1, H_2) \leftarrow$ SplitInHalves(H)
  $C_1 \leftarrow$ IntersectHalfplanes(H_1)
  $C_2 \leftarrow$ IntersectHalfplanes(H_2)
  $C \leftarrow$ IntersectConvexRegions(C_1, C_2)
return C



Beschränkte LPs

Karlsruhe Institute of Technology

Idee: Statt gesamtes zulässiges Polygon zu berechnen suche inkrementell nach optimaler Ecke.

Beschränkte LPs

Karlsruhe Institute of Technology

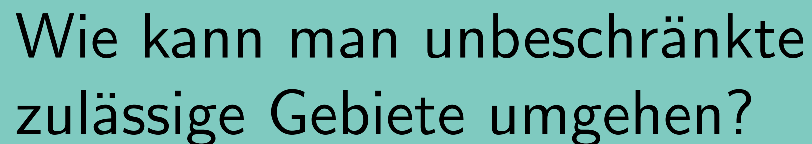
Idee: Statt gesamtes zulässiges Polygon zu berechnen suche inkrementell nach optimaler Ecke.

Invariante: aktuell beste Lösung ist eindeutige Ecke des zulässigen aktuellen Polygons

Beschränkte LPs

Idee: Statt gesamtes zulässiges Polygon zu berechnen suche inkrementell nach optimaler Ecke.

Invariante: aktuell beste Lösung ist eindeutige Ecke des zulässigen aktuellen Polygons



Beschränkte LPs

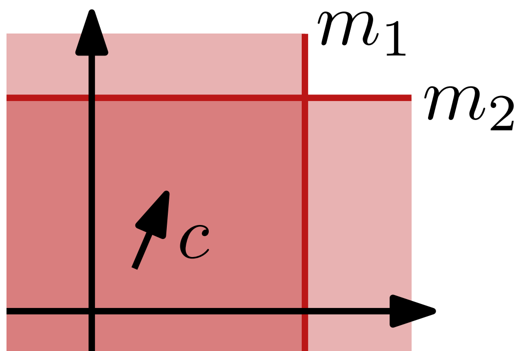
Idee: Statt gesamtes zulässiges Polygon zu berechnen suche inkrementell nach optimaler Ecke.

Invariante: aktuell beste Lösung ist eindeutige Ecke des zulässigen aktuellen Polygons

Wie kann man unbeschränkte zulässige Gebiete umgehen?

Für einen ausreichend großen Wert M definiere Halbebenen

$$m_1 = \begin{cases} x \leq M & \text{falls } c_x > 0 \\ -x \leq M & \text{sonst} \end{cases} \qquad m_2 = \begin{cases} y \leq M & \text{falls } c_y > 0 \\ -y \leq M & \text{sonst} \end{cases}$$



Beschränkte LPs

Idee: Statt gesamtes zulässiges Polygon zu berechnen suche inkrementell nach optimaler Ecke.

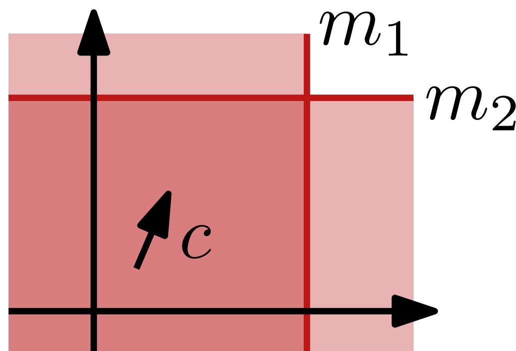
Invariante: aktuell beste Lösung ist eindeutige Ecke des zulässigen aktuellen Polygons

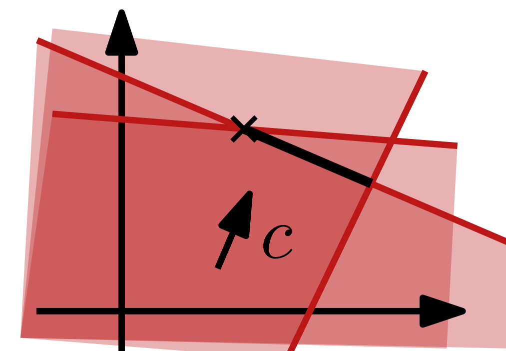
Wie kann man unbeschränkte zulässige Gebiete umgehen?

Gibt es mehrere Optima, wähle lexikographisch kleinsten Punkt!

Für einen ausreichend großen Wert M definiere Halbebenen

$$m_1 = \begin{cases} x \leq M & \text{falls } c_x > 0 \\ -x \leq M & \text{sonst} \end{cases} \qquad m_2 = \begin{cases} y \leq M & \text{falls } c_y > 0 \\ -y \leq M & \text{sonst} \end{cases}$$





Beschränkte LPs

KIT – Karlsruhe Institute of Technology

Idee: Statt gesamtes zulässiges Polygon zu berechnen suche inkrementell nach optimaler Ecke.

Invariante: aktuell beste Lösung ist eindeutige Ecke des zulässigen aktuellen Polygons

Wie kann man unbeschränkte zulässige Gebiete umgehen?

Gibt es mehrere Optima, wähle lexikographisch kleinsten Punkt!

Für einen ausreichend großen Wert M definiere Halbebenen

$$m_1 = \begin{cases} x \leq M & \text{falls } c_x > 0 \\ -x \leq M & \text{sonst} \end{cases} \qquad m_2 = \begin{cases} y \leq M & \text{falls } c_y > 0 \\ -y \leq M & \text{sonst} \end{cases}$$

Für ein LP (H, c) mit $H = \{h_1, \dots, h_n\}$, $c = (c_x, c_y)$, zulässigem Polygon C und $1 \leq i \leq n$ definiere

$$H_i = \{m_1, m_2, h_1, \dots, h_i\}, \quad C_i = m_1 \cap m_2 \cap h_1 \cap \dots \cap h_i$$

Eigenschaften

Karlsruhe Institute of Technology

- jede Region C_i hat eine eindeutige optimale Ecke v_i

Eigenschaften

Karlsruhe Institute of Technology

- jede Region C_i hat eine eindeutige optimale Ecke v_i
- es gilt die Inklusionsbeziehung $C_0 \supseteq C_1 \supseteq \cdots \supseteq C_n = C$

Eigenschaften

Karlsruhe Institute of Technology

- jede Region C_i hat eine eindeutige optimale Ecke v_i
- es gilt die Inklusionsbeziehung $C_0 \supseteq C_1 \supseteq \cdots \supseteq C_n = C$

Wie ändert sich die optimale Ecke v_{i-1} wenn man h_i hinzufügt?

Eigenschaften

Karlsruhe Institute of Technology

- jede Region C_i hat eine eindeutige optimale Ecke v_i
- es gilt die Inklusionsbeziehung $C_0 \supseteq C_1 \supseteq \cdots \supseteq C_n = C$

Wie ändert sich die optimale Ecke v_{i-1} wenn man h_i hinzufügt?

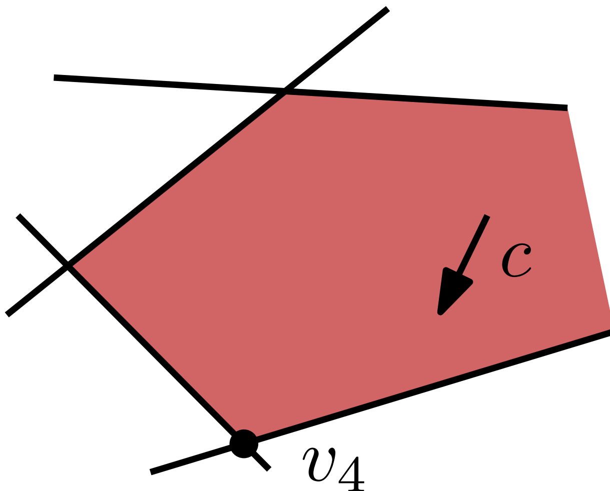
Lemma 1: Für $1 \leq i \leq n$ und Grenzgerade ℓ_i von h_i gilt:
(i) Falls $v_{i-1} \in h_i$ gilt $v_i = v_{i-1}$,
(ii) sonst ist entweder $C_i = \emptyset$ oder $v_i \in \ell_i$.

Eigenschaften

- jede Region C_i hat eine eindeutige optimale Ecke v_i
- es gilt die Inklusionsbeziehung $C_0 \supseteq C_1 \supseteq \cdots \supseteq C_n = C$

Wie ändert sich die optimale Ecke v_{i-1} wenn man h_i hinzufügt?

Lemma 1: Für $1 \leq i \leq n$ und Grenzgerade ℓ_i von h_i gilt:
(i) Falls $v_{i-1} \in h_i$ gilt $v_i = v_{i-1}$,
(ii) sonst ist entweder $C_i = \emptyset$ oder $v_i \in \ell_i$.



Eigenschaften

Karlsruhe Institute of Technology

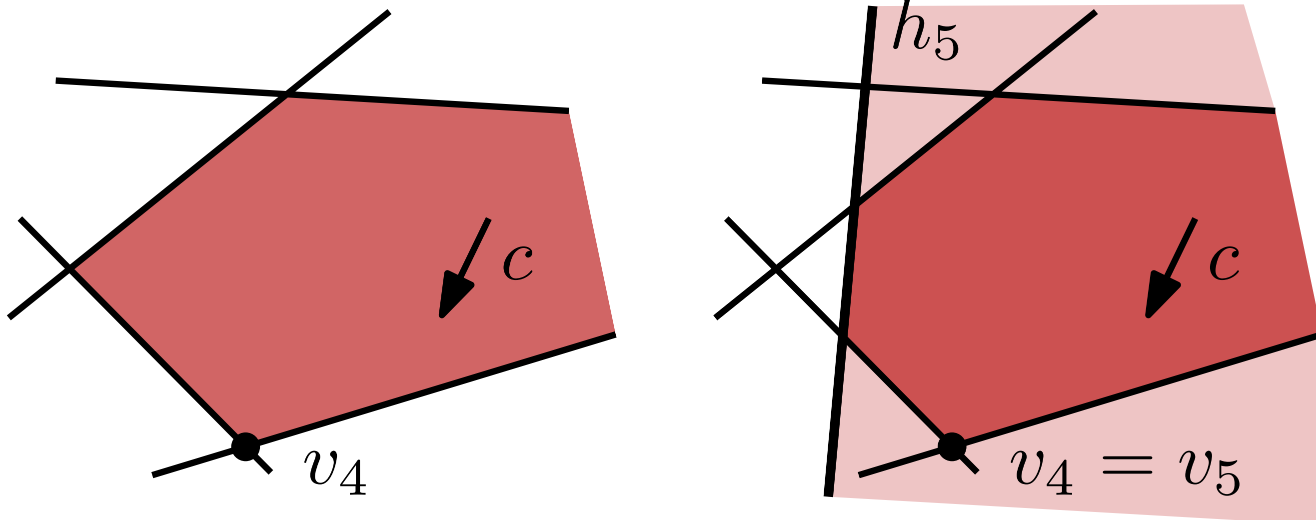
- jede Region C_i hat eine eindeutige optimale Ecke v_i
- es gilt die Inklusionsbeziehung $C_0 \supseteq C_1 \supseteq \cdots \supseteq C_n = C$

Wie ändert sich die optimale Ecke v_{i-1} wenn man h_i hinzufügt?

Lemma 1: Für $1 \leq i \leq n$ und Grenzgerade ℓ_i von h_i gilt:

(i) Falls $v_{i-1} \in h_i$ gilt $v_i = v_{i-1}$,

(ii) sonst ist entweder $C_i = \emptyset$ oder $v_i \in \ell_i$.



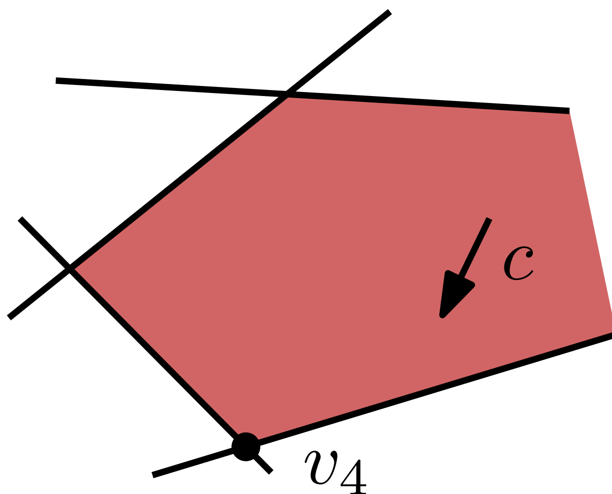
Eigenschaften

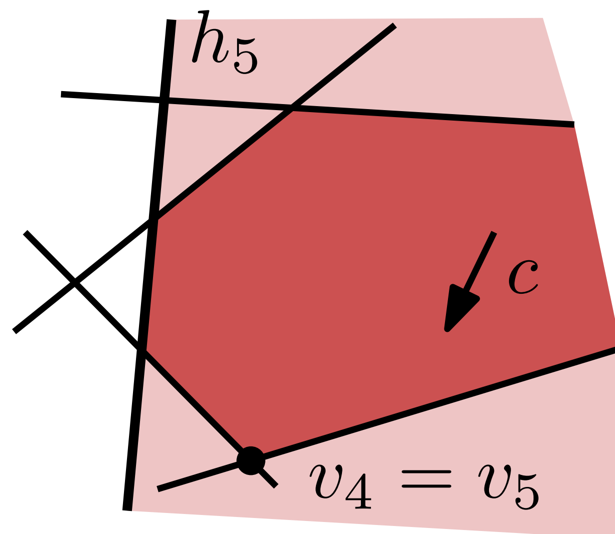
Karlsruhe Institute of Technology

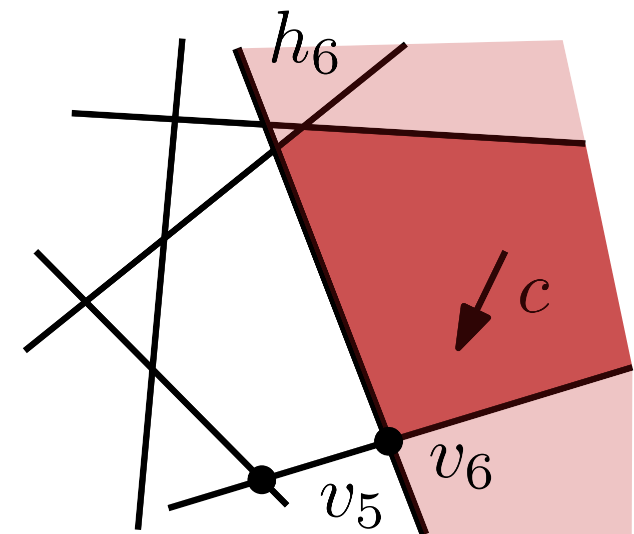
- jede Region C_i hat eine eindeutige optimale Ecke v_i
- es gilt die Inklusionsbeziehung $C_0 \supseteq C_1 \supseteq \cdots \supseteq C_n = C$

Wie ändert sich die optimale Ecke v_{i-1} wenn man h_i hinzufügt?

Lemma 1: Für $1 \leq i \leq n$ und Grenzgerade ℓ_i von h_i gilt:
(i) Falls $v_{i-1} \in h_i$ gilt $v_i = v_{i-1}$,
(ii) sonst ist entweder $C_i = \emptyset$ oder $v_i \in \ell_i$.







Inkrementeller Algorithmus

2dBoundedLP(H, c, m_1, m_2)

```
C_0 ← m_1 ∩ m_2
v_0 ← eindeutige Ecke von C_0
for i ← 1 to n do
    if v_{i-1} ∈ h_i then
        v_i ← v_{i-1}
    else
        v_i ← 1dBoundedLP(σ(H_{i-1}), f_c^i)
        if v_i = nil then
            return unlösbar
return v_n
```

Inkrementeller Algorithmus

Karlsruhe Institute of Technology

2dBoundedLP(H, c, m_1, m_2)

$C_0 \leftarrow m_1 \cap m_2$
$v_0 \leftarrow$ eindeutige Ecke von C_0
for $i \leftarrow 1$ **to** n **do**
  if $v_{i-1} \in h_i$ **then**
    $v_i \leftarrow v_{i-1}$ $O(1)$
  else
    $v_i \leftarrow$ 1dBoundedLP$(\sigma(H_{i-1}), f_c^i)$
    if $v_i =$ nil **then** $O(i)$
      return unlösbar
return v_n

Inkrementeller Algorithmus

Karlsruhe Institute of Technology

2dBoundedLP(H, c, m_1, m_2)

$C_0 \leftarrow m_1 \cap m_2$
$v_0 \leftarrow$ eindeutige Ecke von C_0
for $i \leftarrow 1$ **to** n **do**
  if $v_{i-1} \in h_i$ **then**
    $v_i \leftarrow v_{i-1}$ — $O(1)$
  else
    $v_i \leftarrow$ 1dBoundedLP$(\sigma(H_{i-1}), f_c^i)$
    if $v_i =$ nil **then** — $O(i)$
      return unlösbar
return v_n

worst-case Laufzeit:
$T(n) = \sum_{i=1}^{n} O(i) = O(n^2)$

Inkrementeller Algorithmus

Karlsruhe Institute of Technology

```
2dBoundedLP(H, c, m_1, m_2)

  C_0 ← m_1 ∩ m_2
  v_0 ← eindeutige Ecke von C_0
  for i ← 1 to n do
      if v_{i-1} ∈ h_i then
          v_i ← v_{i-1}                                          O(1)
      else
          v_i ← 1dBoundedLP(σ(H_{i-1}), f_c^i)
          if v_i = nil then                                      O(i)
              return unlösbar
  return v_n
```

worst-case Laufzeit:
$T(n) = \sum_{i=1}^{n} O(i) = O(n^2)$

Frage: Kann wirklich n-mal Fall (ii) auftreten?

Inkrementeller Algorithmus

Karlsruhe Institute of Technology

2dBoundedLP(H, c, m_1, m_2)

$C_0 \leftarrow m_1 \cap m_2$
$v_0 \leftarrow$ eindeutige Ecke von C_0
for $i \leftarrow 1$ **to** n **do**
  if $v_{i-1} \in h_i$ **then**
    $v_i \leftarrow v_{i-1}$ — $O(1)$
  else
    $v_i \leftarrow$ 1dBoundedLP$(\sigma(H_{i-1}), f_c^i)$
    if $v_i =$ nil **then** — $O(i)$
      return unlösbar
return v_n

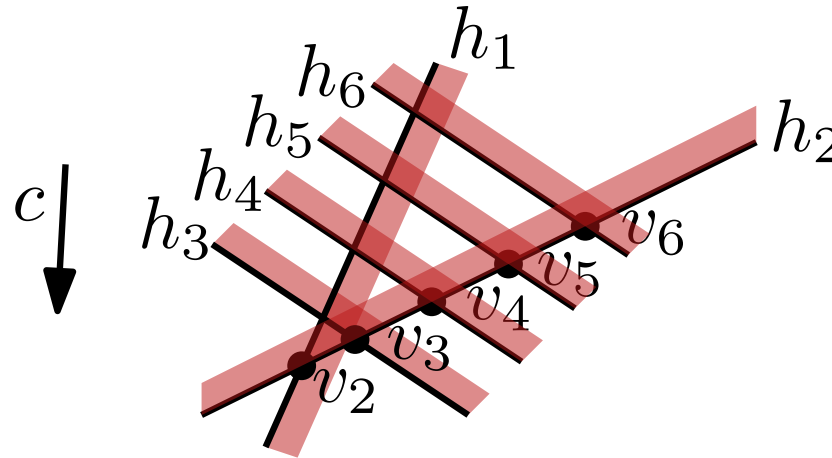
worst-case Laufzeit:
$T(n) = \sum_{i=1}^{n} O(i) = O(n^2)$

Lemma 3: Algorithmus 2dBoundedLP benötigt $\Theta(n^2)$ Laufzeit um ein LP mit n Nebenbed. und 2 Variablen zu lösen.

Gibt es einen Ausweg?

Karlsruhe Institute of Technology

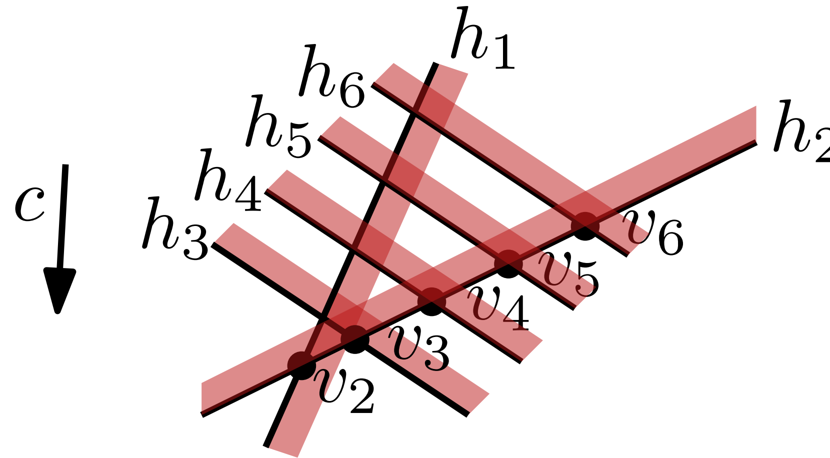
Beob.: Nicht die Halbebenen H sind problematisch für die Laufzeit, sondern die Reihenfolge der Abarbeitung.



Gibt es einen Ausweg?

Karlsruhe Institute of Technology

Beob.: Nicht die Halbebenen H sind problematisch für die Laufzeit, sondern die Reihenfolge der Abarbeitung.

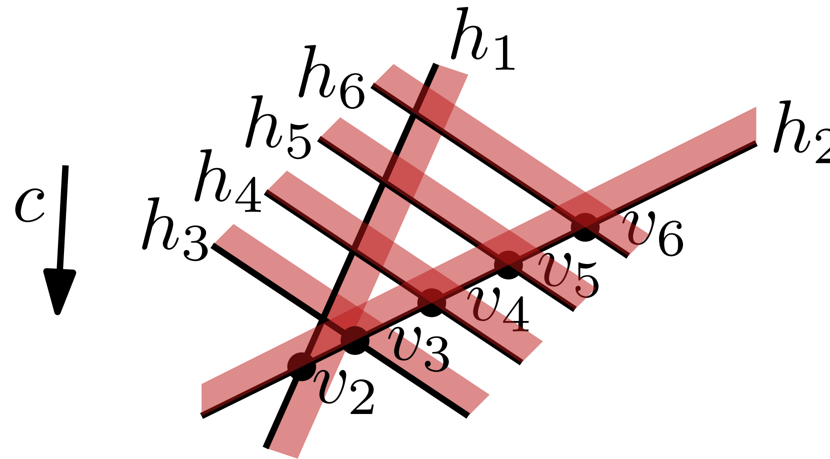


Wie findet man (schnell) eine gute Reihenfolge?

Gibt es einen Ausweg?

Karlsruhe Institute of Technology

Beob.: Nicht die Halbebenen H sind problematisch für die Laufzeit, sondern die Reihenfolge der Abarbeitung.



Wie findet man (schnell) eine gute Reihenfolge?

Randomisierter Inkrementeller Algorithmus

2dRandomizedBoundedLP(H, c, m_1, m_2)

```
C_0 ← m_1 ∩ m_2
v_0 ← eindeutige Ecke von C_0
H ← RandomPermutation(H)
for i ← 1 to n do
    if v_{i-1} ∈ h_i then
        v_i ← v_{i-1}
    else
        v_i ← 1dBoundedLP(σ(H_{i-1}), f_c^i)
        if v_i = nil then
            return unlösbar
return v_n
```

Aufgabe 1

Korrektheitsbeweis:

a) Zeige, dass jede mögliche Permutation von A gleich wahrscheinlich ist.

RandomPermutation(A)

Input: Array $A[1 \dots n]$
Output: Array A, zufällig gleichverteilt permutiert
for $k \leftarrow n$ **to** 2 **do**
  $r \leftarrow$ Random(k)
  tausche $A[r]$ und $A[k]$

Aufgabe 1

Karlsruhe Institute of Technology

Korrektheitsbeweis:

b) Zeige, dass die Aussage aus a) nicht stimmt, wenn wir in der 2. Zeile k mit n ersetzen.

RandomPermutation(A)

Input: Array $A[1 \dots n]$
Output: Array A, zufällig gleichverteilt permutiert
for $k \leftarrow n$ **to** 2 **do**
  $r \leftarrow$ Random(k)
  tausche $A[r]$ und $A[k]$

Aufgabe 1

Karlsruhe Institute of Technology

Korrektheitsbeweis:

b) Zeige, dass die Aussage aus a) nicht stimmt, wenn wir in der 2. Zeile k mit n ersetzen.

RandomPermutation(A)

Input: Array $A[1 \ldots n]$
Output: Array A, zufällig gleichverteilt permutiert ?
for $k \leftarrow n$ **to** 2 **do**
  $r \leftarrow$ Random(n)
  tausche $A[r]$ und $A[k]$

Aufgabe 2

Karlsruhe Institute of Technology

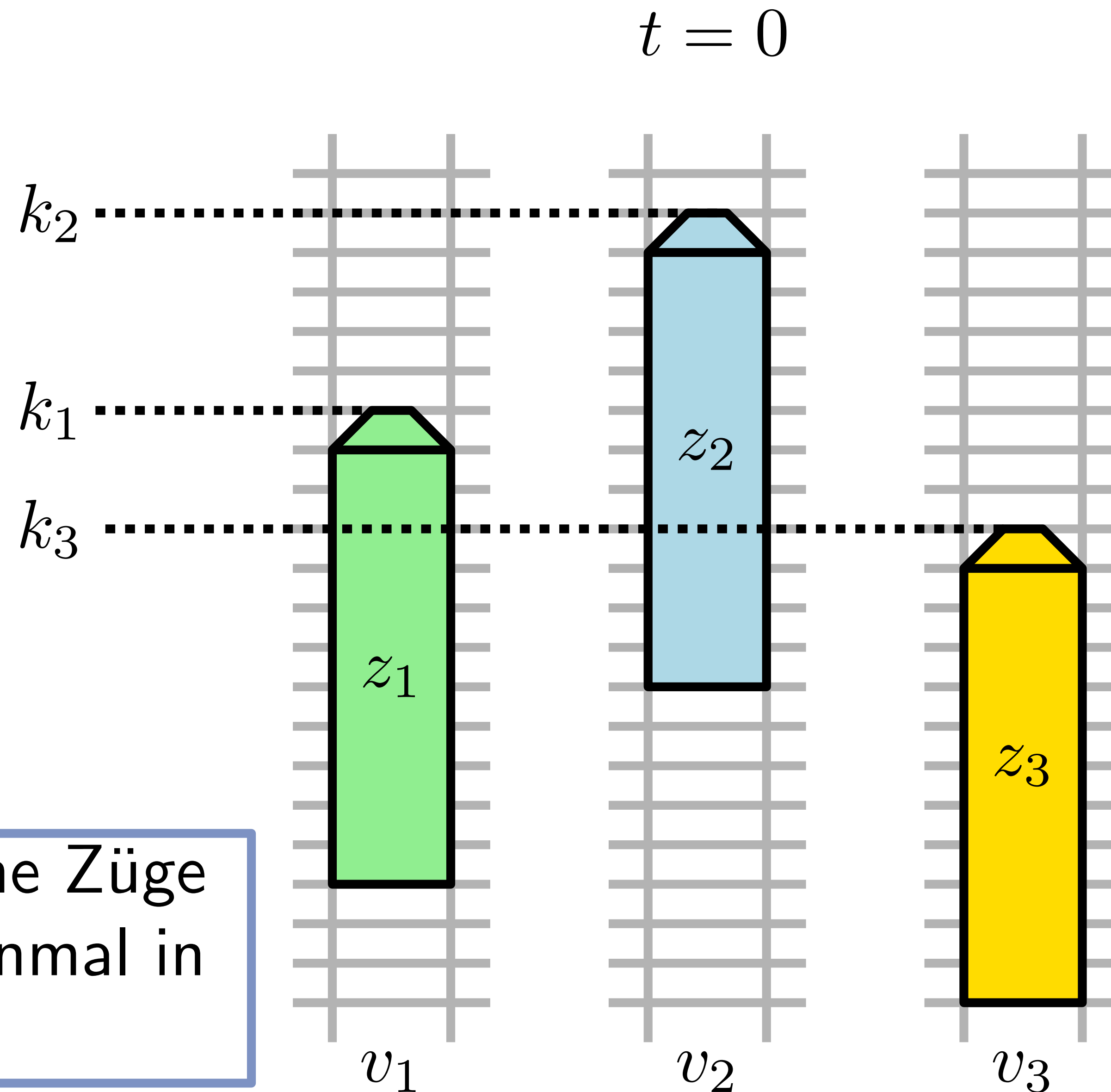
ParanoidMax

Input: Endliche Menge $A \subset \mathbb{R}$
Output: Maximum $\max_{a \in A} a$ der Menge
if $|A| = 1$ **then**
  return einziges Element $a \in A$
else
  $a =$ zufällig gewähltes Element aus A
  $b =$ ParanoidMax$(A \setminus \{a\})$
  if $b \geq a$ **then**
    return b
  else
    prüfe unnötigerweise jedes Element aus $A \setminus \{a\}$, um sicherzugehen, dass a wirklich größer ist
    return a

a) Asymptotische w-c Laufzeit?

b) erwartete Laufzeit echt besser als w-c

Aufgabe 3 : Züge



Algorithmus: welche Züge bis Zeitpunkt t_s einmal in Führung

Aufgabe 3 : Züge

Strecke

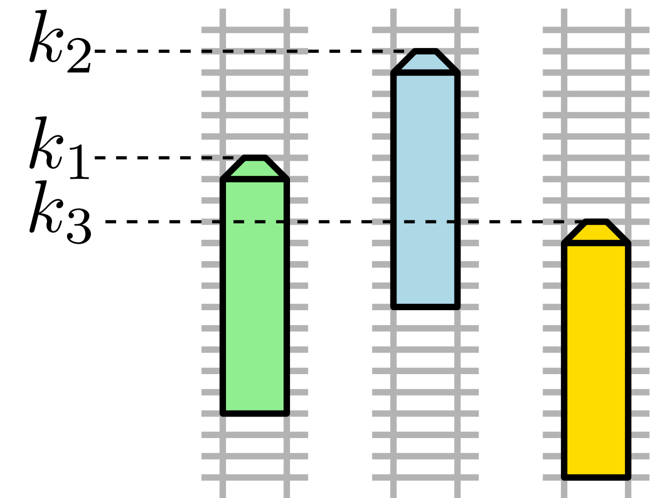
k_2

k_1

k_3

t_{stop}

Zeit



Aufgabe 3 : Züge

Strecke

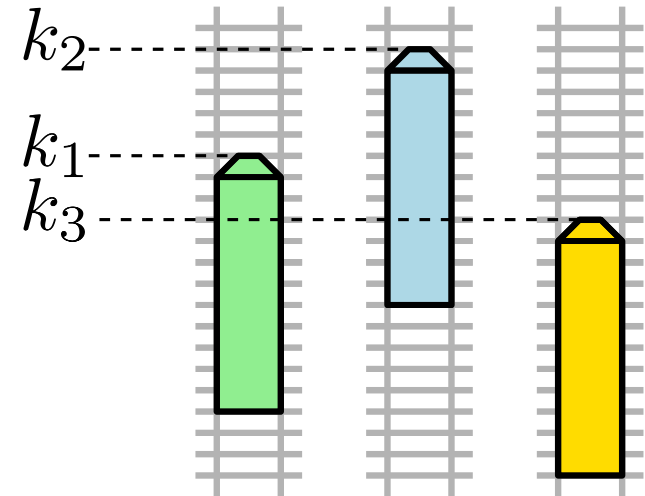
k_2

k_1

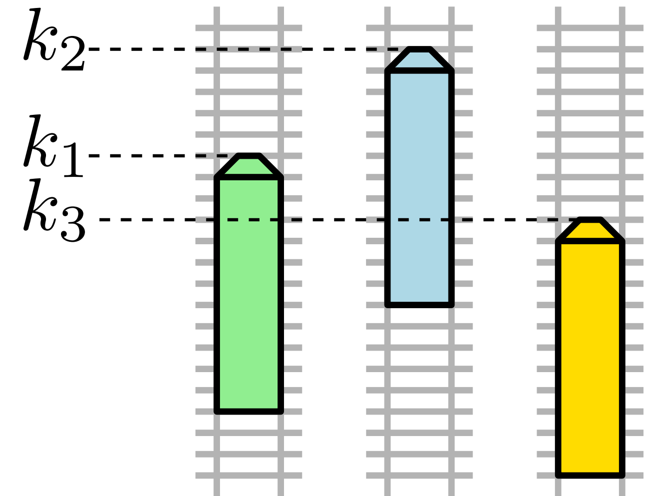
k_3

t_{stop}

Zeit



Aufgabe 3 : Züge



Aufgabe 3 : Züge

Karlsruhe Institute of Technology

Strecke

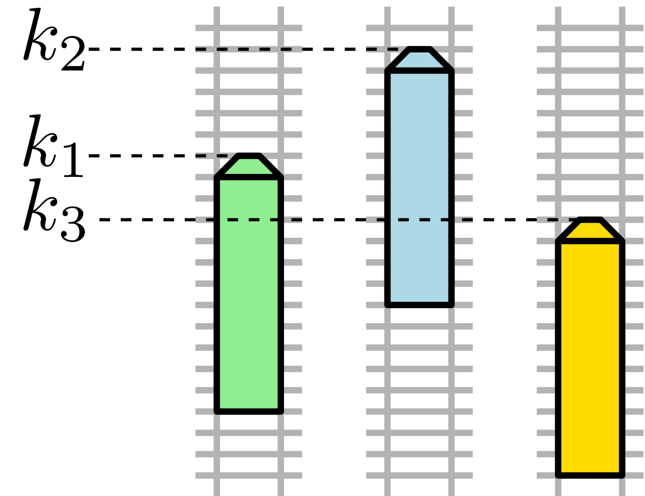
k_2

k_1

k_3

t_{stop}

Zeit



Aufgabe 3 : Züge

Strecke

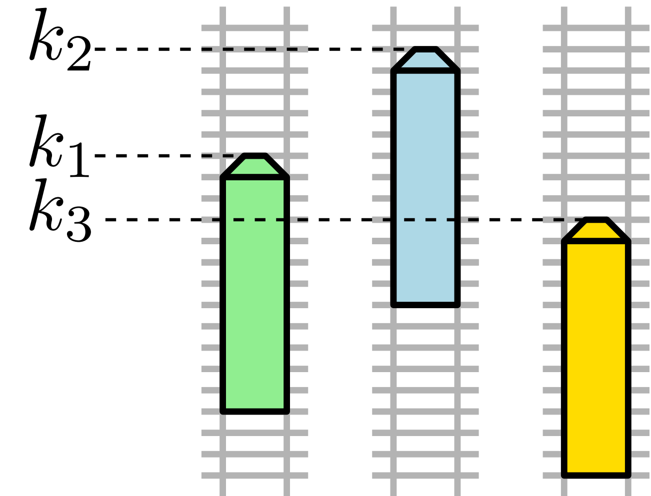
k_2

k_1

k_3

t_{stop}

Zeit



Aufgabe 3 : Züge

Karlsruhe Institute of Technology

```
IntersectHalfplanes(H)
  if |H| = 1 then
  |  C ← H
  else
  |  (H_1, H_2) ← SplitInHalves(H)
  |  C_1 ← IntersectHalfplanes(H_1)
  |  C_2 ← IntersectHalfplanes(H_2)
  |  C ← IntersectConvexRegions(C_1, C_2)
  return C
```

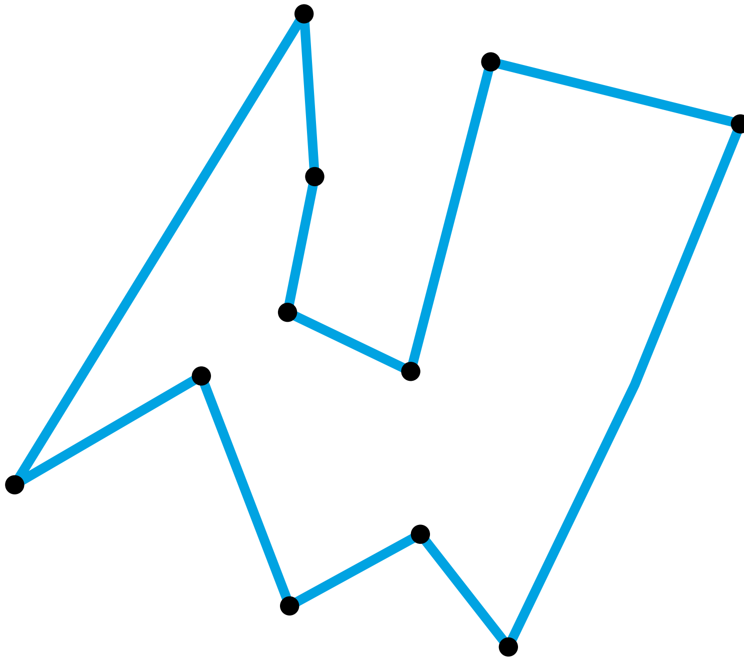
Aufgabe 4 : Polygon Partitionieren

Karlsruhe Institute of Technology

VL: ALG zur Zerlegung eines Polygons in y-mon. Teilstücke

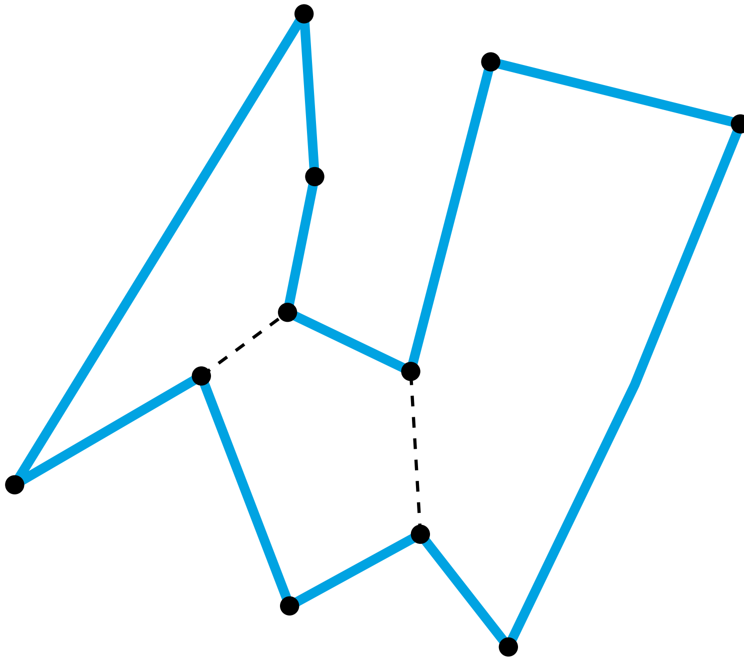
Aufgabe 4 : Polygon Partitionieren

VL: ALG zur Zerlegung eines Polygons in y-mon. Teilstücke



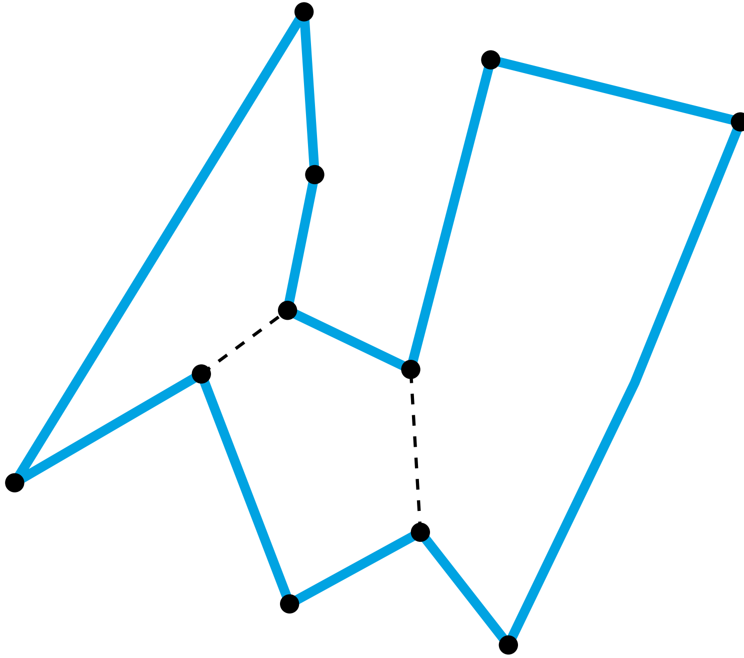
Aufgabe 4 : Polygon Partitionieren

VL: ALG zur Zerlegung eines Polygons in y-mon. Teilstücke



Aufgabe 4 : Polygon Partitionieren

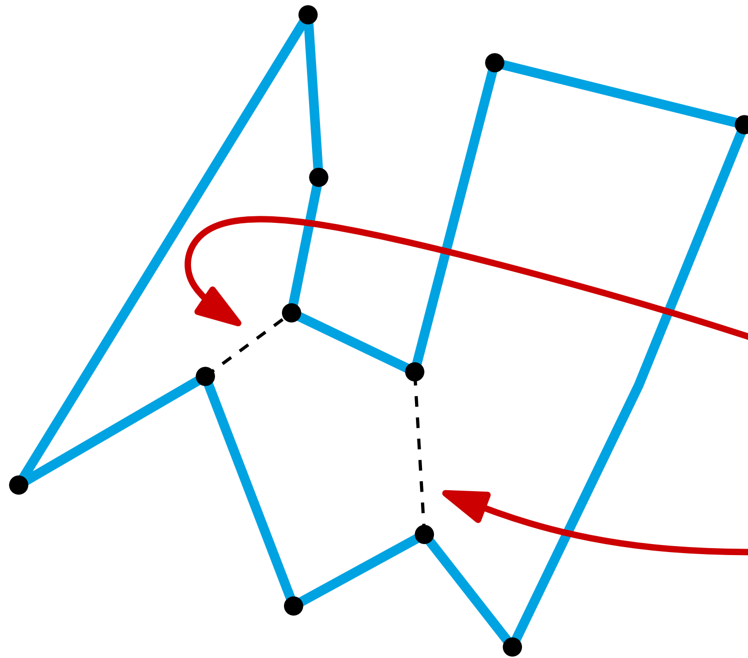
VL: ALG zur Zerlegung eines Polygons in y-mon. Teilstücke



Datenstruktur: Doppelt-verkettete Kantenliste (DCEL)

Aufgabe 4 : Polygon Partitionieren

VL: ALG zur Zerlegung eines Polygons in y-mon. Teilstücke

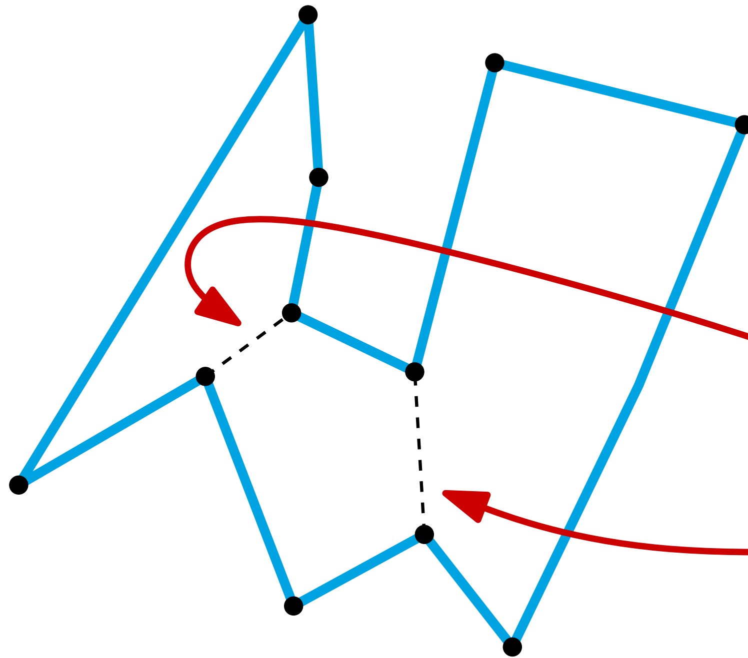


Behauptung:
Einfügen von einer Diagonalen in $O(1)$ möglich

Datenstruktur: Doppelt-verkettete Kantenliste (DCEL)

Aufgabe 4 : Polygon Partitionieren

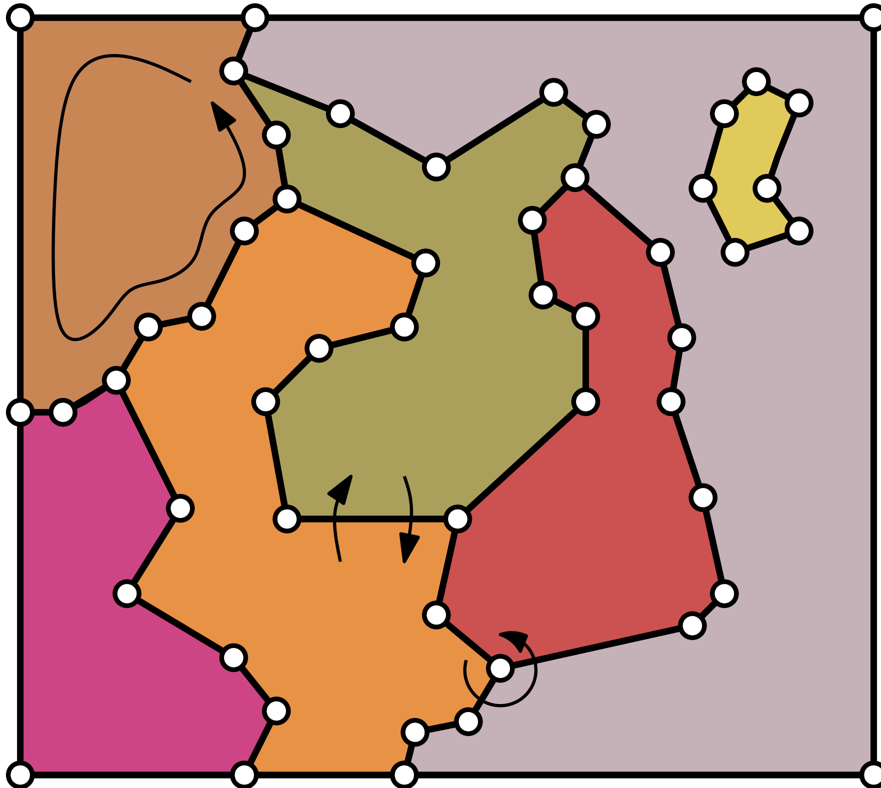
VL: ALG zur Zerlegung eines Polygons in y-mon. Teilstücke



Behauptung:
Einfügen von einer Diagonalen in $O(1)$ möglich

Datenstruktur: Doppelt-verkettete Kantenliste (DCEL)

Datenstruktur für Landkarten



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

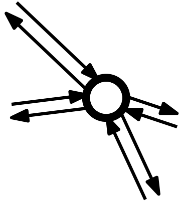
Welche Operationen sollte eine Datenstruktur für Unterteilungen der Ebene unterstützen?

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

Doppelt verkettete Kantenliste (DCEL)

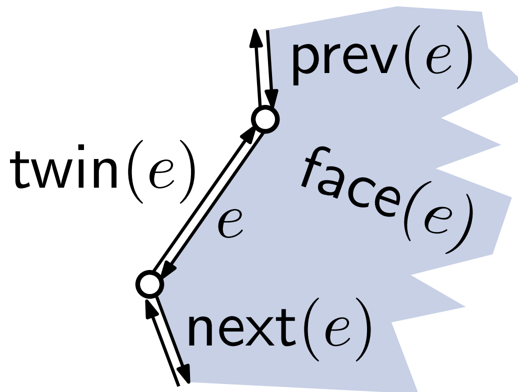
Zutaten:

- Knoten



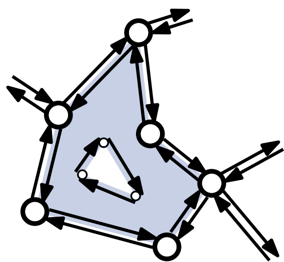
  - Koordinaten $(x(v), y(v))$
  - (erste) ausgehende Kante edge(v)

- Kanten $=$ zwei Halbkanten



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

- Facetten

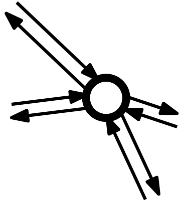


  - Randkante outer(f)
  - Kantenliste inner(f) für evtl. Löcher

Doppelt verkettete Kantenliste (DCEL)

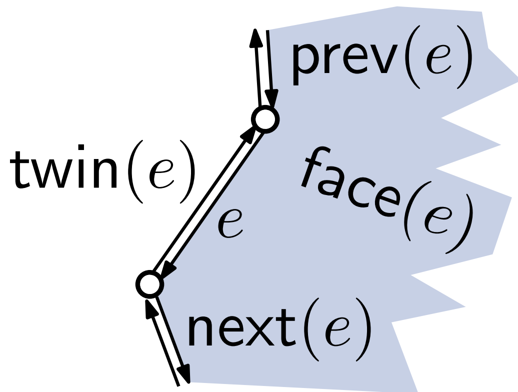
Zutaten:

- Knoten



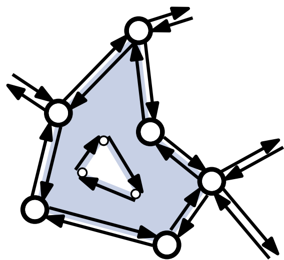
  - Koordinaten $(x(v), y(v))$
  - (erste) ausgehende Kante edge(v)

- Kanten = zwei Halbkanten



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

- Facetten



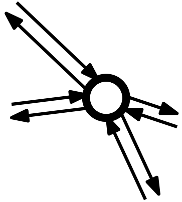
  - Randkante outer(f)
  - Kantenliste inner(f) für evtl. Löcher

Doppelt verkettete Kantenliste (DCEL)

Karlsruhe Institute of Technology

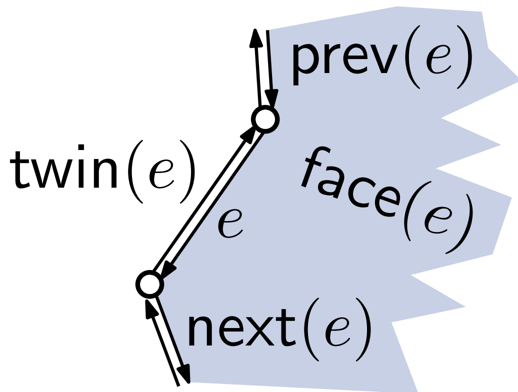
Zutaten:

- Knoten



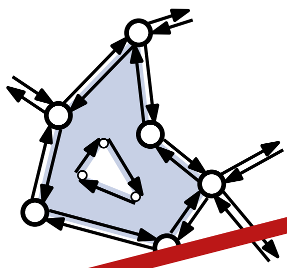
  - Koordinaten $(x(v), y(v))$
  - (erste) ausgehende Kante edge(v)

- Kanten $=$ zwei Halbkanten



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

- Facetten

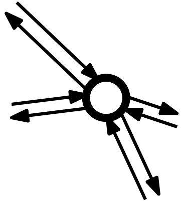


  - Randkante outer(f)
  - Kantenliste inner(f) für evtl. Löcher

Doppelt verkettete Kantenliste (DCEL)

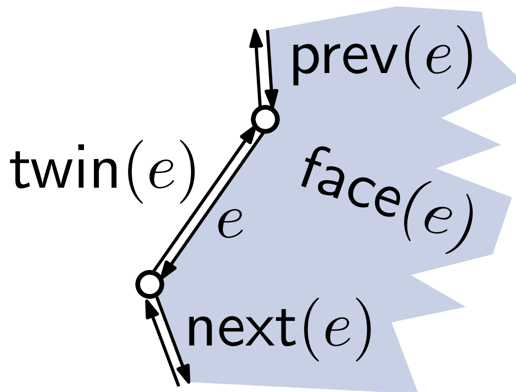
Zutaten:

- Knoten



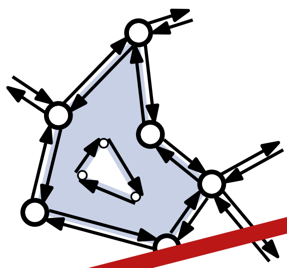
  - Koordinaten $(x(v), y(v))$
  - (erste) ausgehende Kante edge(v)

- Kanten = zwei Halbkanten



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

- ~~Facetten~~

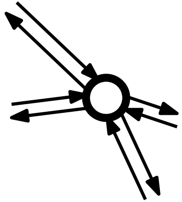


  - ~~Randkante outer(f)~~
  - ~~Kantenliste inner(f) für evtl. Löcher~~

Doppelt verkettete Kantenliste (DCEL)

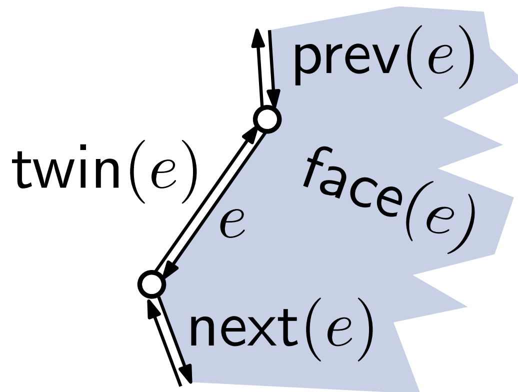
Zutaten:

- Knoten



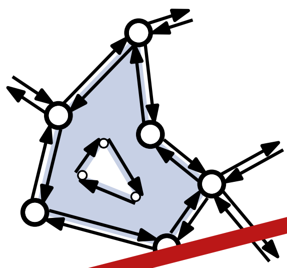
- Koordinaten $(x(v), y(v))$
- (erste) ausgehende Kante edge(v)

- Kanten = zwei Halbkanten



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

- ~~Facetten~~



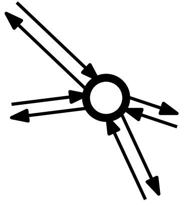
- ~~Randkante outer(f)~~
- ~~Kantenliste inner(f) für evtl. Löcher~~

Doppelt verkettete Kantenliste (DCEL)

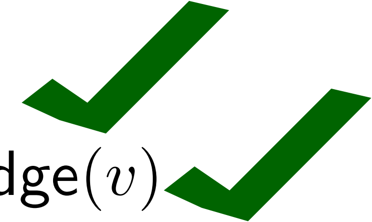
Karlsruhe Institute of Technology

Zutaten:

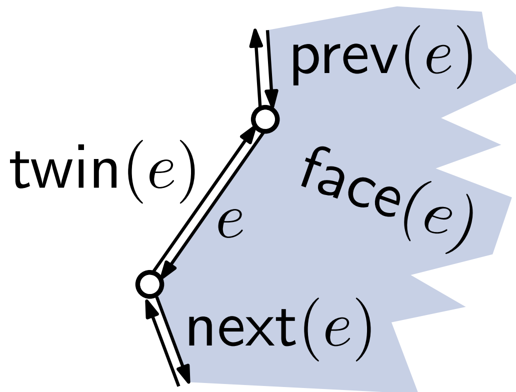
- Knoten



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

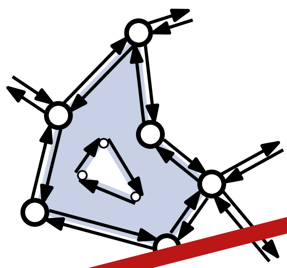


- Kanten = zwei Halbkanten



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

- ~~Facetten~~

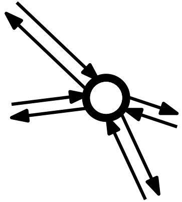


- ~~Randkante outer(f)~~
- ~~Kantenliste inner(f) für evtl. Löcher~~

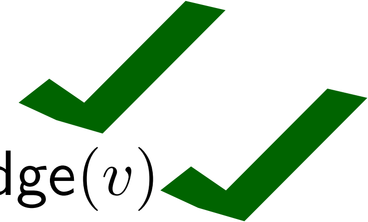
Doppelt verkettete Kantenliste (DCEL)

Zutaten:

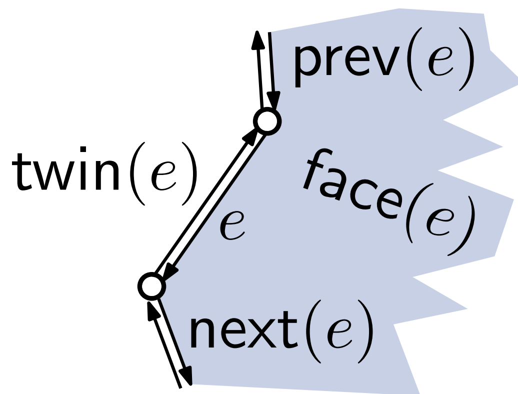
- Knoten



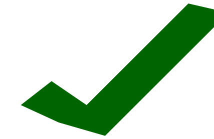
  - Koordinaten $(x(v), y(v))$
  - (erste) ausgehende Kante edge(v)



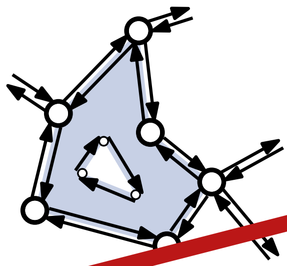
- Kanten = zwei Halbkanten



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



- ~~Facetten~~

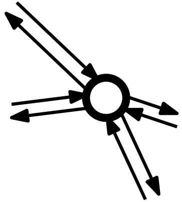


  - ~~Randkante outer(f)~~
  - ~~Kantenliste inner(f) für evtl. Löcher~~

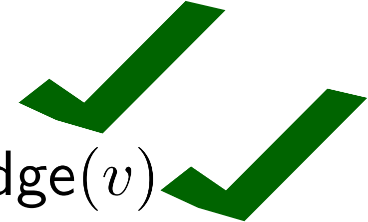
Doppelt verkettete Kantenliste (DCEL)

Zutaten:

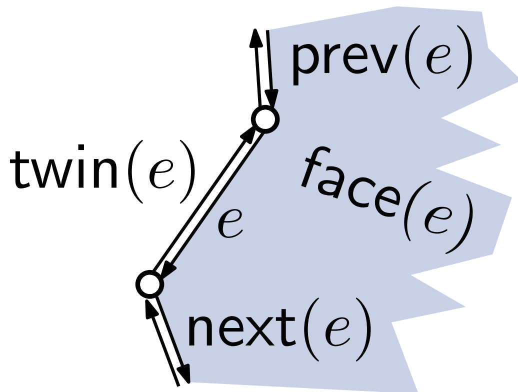
- Knoten



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

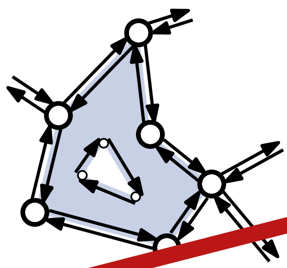


- Kanten = zwei Halbkanten



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

- ~~Facetten~~

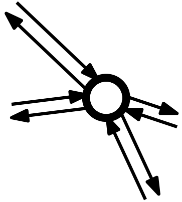


- ~~Randkante outer(f)~~
- ~~Kantenliste inner(f) für evtl. Löcher~~

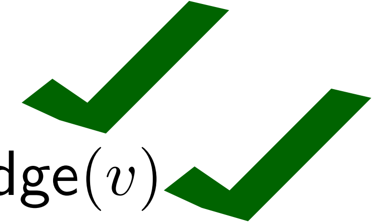
Doppelt verkettete Kantenliste (DCEL)

Zutaten:

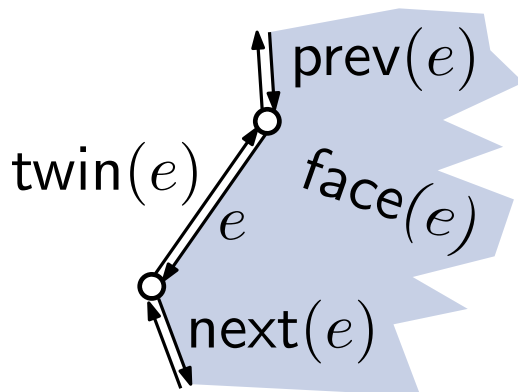
- Knoten



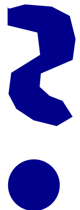
  - Koordinaten $(x(v), y(v))$
  - (erste) ausgehende Kante edge(v)



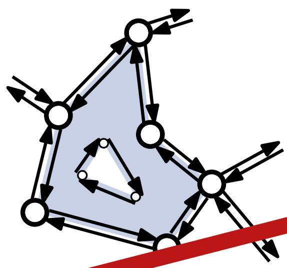
- Kanten = zwei Halbkanten



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



- ~~Facetten~~



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

Lösungsschritte

Karlsruhe Institute of Technology

1. Wie umgeht man das aktualisieren der *face*(e) Information?

Lösungsschritte

Karlsruhe Institute of Technology

1. Wie umgeht man das aktualisieren der *face*(e) Information?
   – Facetten spielen keine (große) Rolle

Lösungsschritte

Karlsruhe Institute of Technology

1. Wie umgeht man das aktualisieren der *face*(e) Information?
   – Facetten spielen keine (große) Rolle

2. Wie wählen wir in $O(1)$ die richtigen Kanten für das anpassen der *next*(e) bzw. *prev*(e) Einträge?

Lösungsschritte

Karlsruhe Institute of Technology

1. Wie umgeht man das aktualisieren der *face*(e) Information?
   – Facetten spielen keine (große) Rolle

2. Wie wählen wir in $O(1)$ die richtigen Kanten für das anpassen der *next*(e) bzw. *prev*(e) Einträge?
   a) Nur konstant viele Kanten inzident zu jedem Knoten

Lösungsschritte

Karlsruhe Institute of Technology

1. Wie umgeht man das aktualisieren der *face*(e) Information?

– Facetten spielen keine (große) Rolle

2. Wie wählen wir in $O(1)$ die richtigen Kanten für das anpassen der *next*(e) bzw. *prev*(e) Einträge?

a) Nur konstant viele Kanten inzident zu jedem Knoten

b) Durch passende Sortierung können wir die korrekten Kanten finden

Algorithmus MakeMonotone(P)

Karlsruhe Institute of Technology

MakeMonotone(Polygon P)

$\mathcal{D} \leftarrow$ doppelt-verkettete Kantenliste für $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue für $V(P)$ lexikographisch sortiert
$\mathcal{T} \leftarrow \emptyset$ (binärer Suchbaum für Sweep-Line Status)
while $\mathcal{Q} \neq \emptyset$ **do**
  $v \leftarrow \mathcal{Q}$.nextVertex()
  $\mathcal{Q}$.deleteVertex(v)
  handleVertex(v)

return $\mathcal{D}$

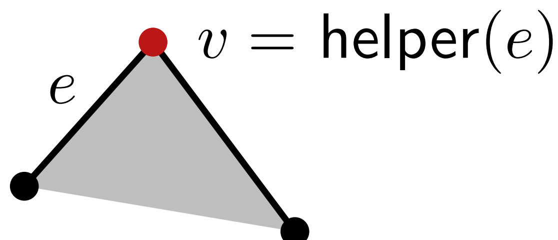
Algorithmus MakeMonotone(P)

MakeMonotone(Polygon P)

$\mathcal{D} \leftarrow$ doppelt-verkettete Kantenliste für $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue für $V(P)$ lexikographisch sortiert
$\mathcal{T} \leftarrow \emptyset$ (binärer Suchbaum für Sweep-Line Status)
while $\mathcal{Q} \neq \emptyset$ **do**
  $v \leftarrow \mathcal{Q}$.nextVertex()
  $\mathcal{Q}$.deleteVertex(v)
  handleVertex(v)
return $\mathcal{D}$

handleStartVertex(vertex v)

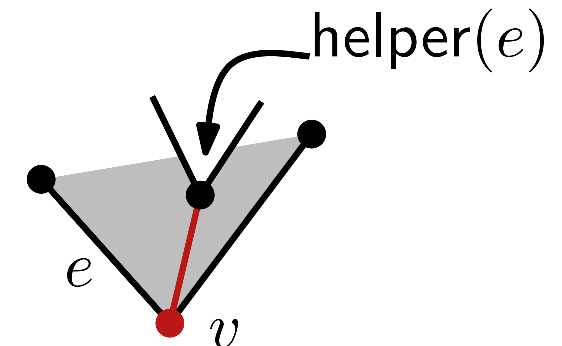
$\mathcal{T} \leftarrow$ füge linke Kante e ein
helper(e) $\leftarrow v$



Algorithmus MakeMonotone(P)

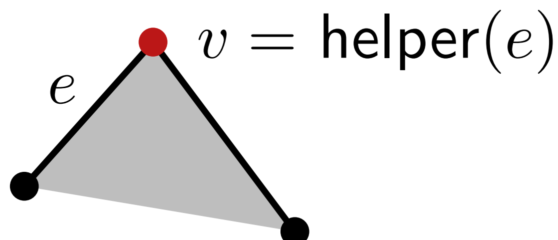
MakeMonotone(Polygon P)

$\mathcal{D} \leftarrow$ doppelt-verkettete Kantenliste für $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue für $V(P)$ lexikographisch sortiert
$\mathcal{T} \leftarrow \emptyset$ (binärer Suchbaum für Sweep-Line Status)
while $\mathcal{Q} \neq \emptyset$ **do**
  $v \leftarrow \mathcal{Q}$.nextVertex()
  $\mathcal{Q}$.deleteVertex(v)
  handleVertex(v)
return $\mathcal{D}$



handleStartVertex(vertex v)

$\mathcal{T} \leftarrow$ füge linke Kante e ein
helper(e) $\leftarrow v$



handleEndVertex(vertex v)

$e \leftarrow$ linke Kante
if isMergeVertex(helper(e)) **then**
  $\mathcal{D} \leftarrow$ füge (helper(e), v) ein
lösche e aus $\mathcal{T}$

Algorithmus MakeMonotone(P)

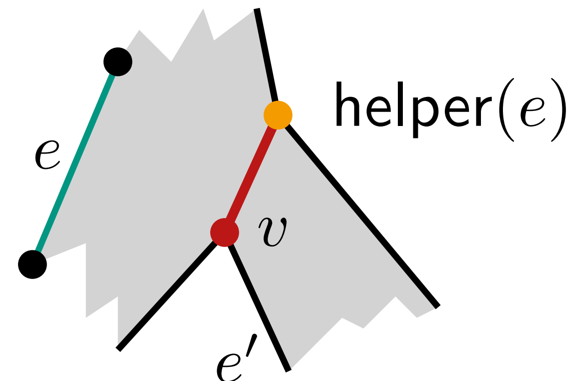
Karlsruhe Institute of Technology

MakeMonotone(Polygon P)

$\mathcal{D} \leftarrow$ doppelt-verkettete Kantenliste für $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue für $V(P)$ lexikographisch sortiert
$\mathcal{T} \leftarrow \emptyset$ (binärer Suchbaum für Sweep-Line Status)
while $\mathcal{Q} \neq \emptyset$ **do**
  $v \leftarrow \mathcal{Q}$.nextVertex()
  $\mathcal{Q}$.deleteVertex(v)
  handleVertex(v)
return $\mathcal{D}$

handleSplitVertex(vertex v)

$e \leftarrow$ Kante links von v in $\mathcal{T}$
$\mathcal{D} \leftarrow$ füge (helper(e), v) ein
helper(e) $\leftarrow v$
$\mathcal{T} \leftarrow$ füge rechte Kante e' von v ein
helper(e') $\leftarrow v$



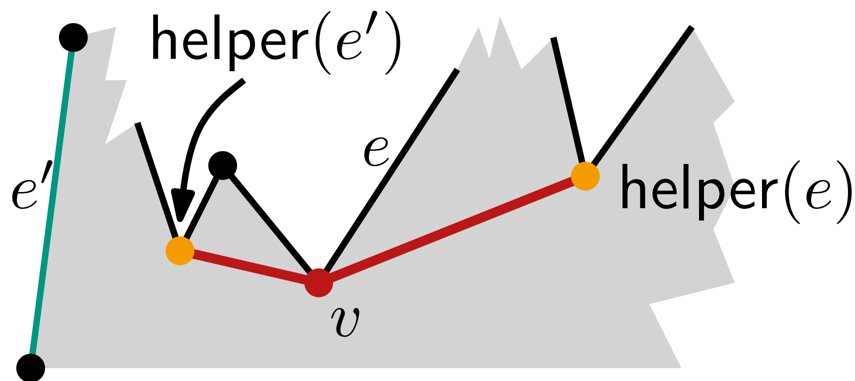
Algorithmus MakeMonotone(P)

MakeMonotone(Polygon P)

$\mathcal{D} \leftarrow$ doppelt-verkettete Kantenliste für $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue für $V(P)$ lexikographisch sortiert
$\mathcal{T} \leftarrow \emptyset$ (binärer Suchbaum für Sweep-Line Status)
while $\mathcal{Q} \neq \emptyset$ **do**
  $v \leftarrow \mathcal{Q}$.nextVertex()
  $\mathcal{Q}$.deleteVertex(v)
  handleVertex(v)
return $\mathcal{D}$

handleMergeVertex(vertex v)

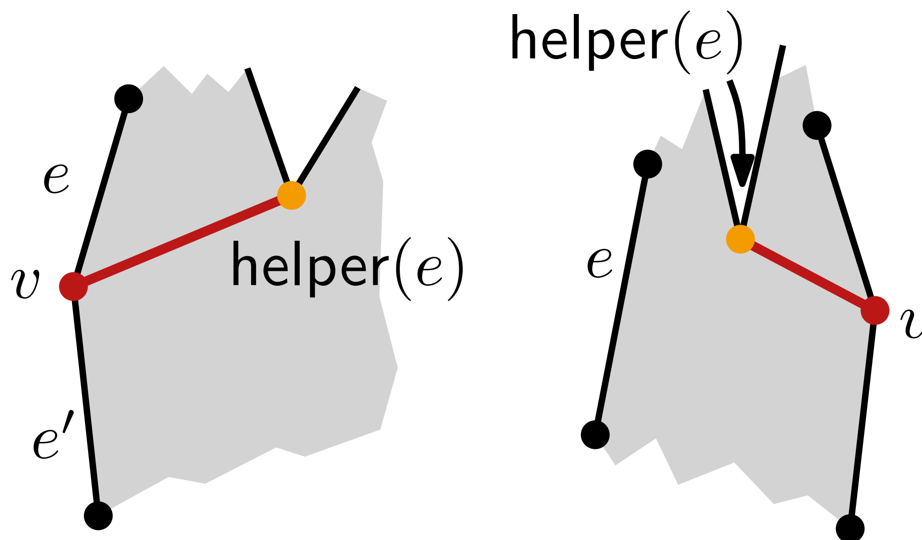
$e \leftarrow$ rechte Kante
if isMergeVertex(helper(e)) **then**
  $\mathcal{D} \leftarrow$ füge (helper(e), v) ein
lösche e aus $\mathcal{T}$
$e' \leftarrow$ Kante links von v in $\mathcal{T}$
if isMergeVertex(helper(e')) **then**
  $\mathcal{D} \leftarrow$ füge (helper(e'), v) ein
helper(e') $\leftarrow v$



Algorithmus MakeMonotone(P)

MakeMonotone(Polygon P)

$\mathcal{D} \leftarrow$ doppelt-verkettete Kantenliste für $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue für $V(P)$ lexikographisch sortiert
$\mathcal{T} \leftarrow \emptyset$ (binärer Suchbaum für Sweep-Line Status)
while $\mathcal{Q} \neq \emptyset$ **do**
    $v \leftarrow \mathcal{Q}$.nextVertex()
    $\mathcal{Q}$.deleteVertex(v)
    handleVertex(v)
return $\mathcal{D}$



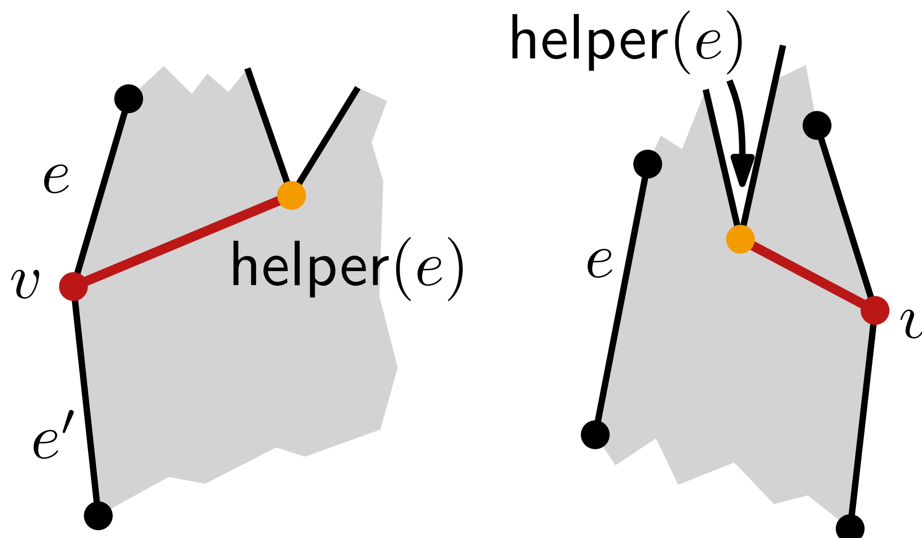
handleRegularVertex(vertex v)

if P liegt lokal rechts von v **then**
    $e, e' \leftarrow$ obere, untere Kante
    if isMergeVertex(helper(e)) **then**
        $\mathcal{D} \leftarrow$ füge (helper$(e), v$) ein
    lösche e aus $\mathcal{T}$
    $\mathcal{T} \leftarrow$ füge e' ein; helper$(e') \leftarrow v$
else
    $e \leftarrow$ Kante links von v in $\mathcal{T}$
    if isMergeVertex(helper(e)) **then**
        $\mathcal{D} \leftarrow$ füge (helper$(e), v$) ein
    helper$(e) \leftarrow v$

Algorithmus MakeMonotone(P)

MakeMonotone(Polygon P)

$\mathcal{D} \leftarrow$ doppelt-verkettete Kantenliste für $(V(P), E(P))$
$\mathcal{Q} \leftarrow$ priority queue für $V(P)$ lexikographisch sortiert
$\mathcal{T} \leftarrow \emptyset$ (binärer Suchbaum für Sweep-Line Status)
while $\mathcal{Q} \neq \emptyset$ **do**
    $v \leftarrow \mathcal{Q}$.nextVertex()
    $\mathcal{Q}$.deleteVertex(v)
    handleVertex(v)
return $\mathcal{D}$



handleRegularVertex(vertex v)

if P liegt lokal rechts von v **then**
    $e, e' \leftarrow$ obere, untere Kante
    if isMergeVertex(helper(e)) **then**
        $\mathcal{D} \leftarrow$ füge (helper(e), v) ein
    lösche e aus $\mathcal{T}$
    $\mathcal{T} \leftarrow$ füge e' ein; helper(e') $\leftarrow v$
else
    $e \leftarrow$ Kante links von v in $\mathcal{T}$
    if isMergeVertex(helper(e)) **then**
        $\mathcal{D} \leftarrow$ füge (helper(e), v) ein
    helper(e) $\leftarrow v$

Lösung

Karlsruhe Institute of Technology

2. Wie wählen wir in $O(1)$ die richtigen Kanten für das anpassen der *next*(e) bzw. *prev*(e) Einträge?

a) Nur konstant viele Kanten inzident zu jedem Knoten

- Initial hat jeder Knoten Grad 2
- Jeder Knoten ist höchstens einmal helper: $+1$
- Jeder Knoten ist höchstens einmal 'Sonderknoten': $+2$

b) Durch passende Sortierung können wir die korrekten Kanten finden