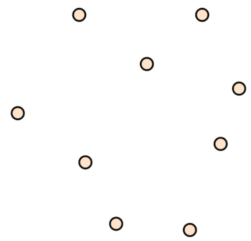
2.7 Geometrische Algorithmen

2.7 Geometrische Algorithmen

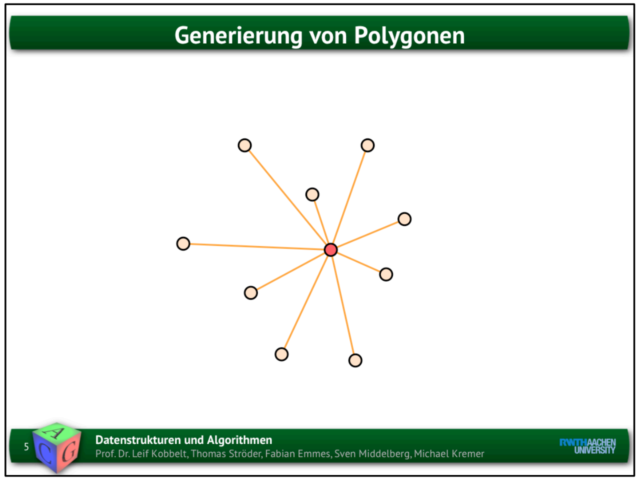
Generierung von Polygonen

- Gegeben sei eine ungeordnete Menge von Eckpunkten p_i
- Generiere ein einfaches, geschlossenes Polygon (durch “Sortieren” der p_i)

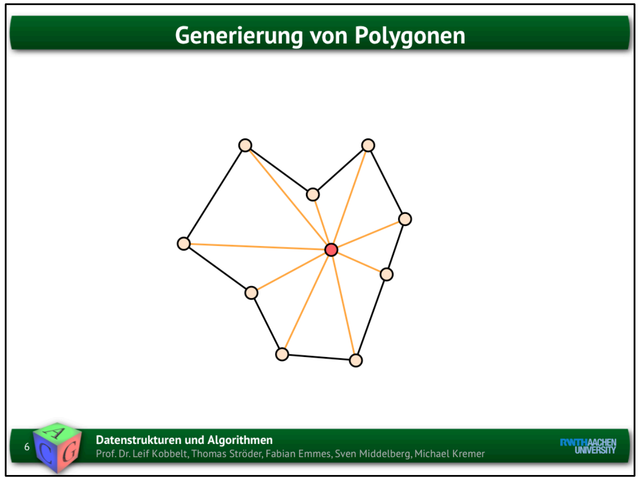
Generierung von Polygonen



Gegeben sei eine Punktwolke.



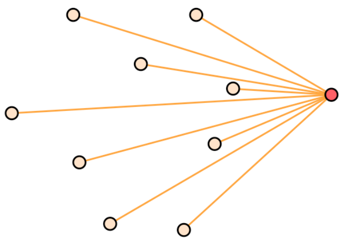
Wir suchen uns willkürlich einen Punkt heraus und verbinden alle anderen Punkte mit diesem Punkt.



Und verbinden die Punkte so, dass keine der (hier orangenen Geraden) geschnitten wird.

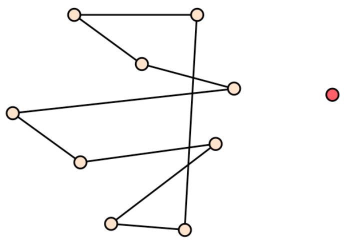
Generierung von Polygonen

aber...



Generierung von Polygonen

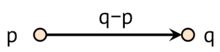
aber...



Dies führt in manchen Fällen auch zu Problemen.

Konvexität

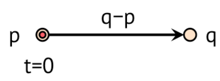
- Eine Teilmenge S des R^2 heißt konvex, wenn mit jedem Paar p,q von Punkten aus S auch alle Zwischenpunkte $(1-t)p+tq$ mit $0 \leqslant t \leqslant 1$ in S liegen.
- $(1-t)p+tq = p-tp+tq = p+t(q-p)$



Wir parametrisieren eine Gerade zwischen zwei Punkten p und q. Wenn t zwischen 0 und 1 liegt, können wir jeden Punkt auf diesem Geradensegment berechnen.

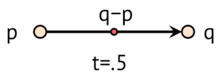
Konvexität

- Eine Teilmenge S des R^2 heißt konvex, wenn mit jedem Paar p,q von Punkten aus S auch alle Zwischenpunkte $(1-t)p$ $+tq$ mit $0 \leqslant t \leqslant 1$ in S liegen.
- $(1-t)p+tq = p-tp+tq = p+t(q-p)$



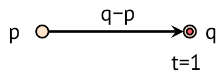
Konvexität

- Eine Teilmenge S des R^2 heißt konvex, wenn mit jedem Paar p,q von Punkten aus S auch alle Zwischenpunkte (1−t)p+tq mit $0 \leqslant t \leqslant 1$ in S liegen.
- (1−t)p+tq = p−tp+tq = p+t(q−p)

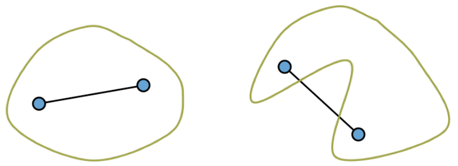


Konvexität

- Eine Teilmenge S des R^2 heißt konvex, wenn mit jedem Paar p,q von Punkten aus S auch alle Zwischenpunkte (1−t)p+tq mit $0 \leqslant t \leqslant 1$ in S liegen.
- (1−t)p+tq = p−tp+tq = p+t(q−p)



Konvexität



konvex nicht konvex

Konvex ist ein geometrisches Gebilde, wenn es keine Gerade zwischen zwei in dem Gebilde liegenden Punkten gibt, die den Rand des Gebildes schneidet.

Konvexe Hülle

- Definition
  Die Konvexe Hülle CH(S) einer Teilmenge S des R^2 ist die kleinste konvexe Menge (bzgl. ⊆), die S enthält.
- Lemma

$$CH(S) = \bigcap_{\substack{K \supseteq S \\ K \text{ ist konvex}}} K$$

Konvexe Hülle

- Lemma
  Es sei eine Menge von (Eck-)Punkten p_i gegeben. Die Konvexe Hülle $CH(p_i)$ ist die Menge aller Punkte q mit

$$q = \sum_{i=1}^{n} \alpha_i p_i \text{ mit } \sum_{i=1}^{n} \alpha_i = 1 \text{ und } \alpha_i \geq 0$$

Wenn die Koeffizienten einer Linearkombination alle positiv sind, wird dies auch „Konvexkombination" genannt.
Wenn sich die Koeffizienten zu 1 aufsummieren, ist es eine „Affinkombination".

Konvexe Hülle

- Beweis
  Es sei $C = \left\{ \sum_{i=1}^{n} \alpha_i p_i : \sum_{i=1}^{n} \alpha_i = 1, \alpha_i \geq 0 \right\}$

  Zu zeigen: $C = CH(p_i)$
- Wegen

$$\sum_{i=1}^{n} \alpha_i p_i = \alpha_1 p_1 + (1 - \alpha_1) \sum_{i=2}^{n} \frac{\alpha_i}{1 - \alpha_1} p_i$$

  gilt offensichtlich $C \subseteq CH(p_i)$
- Noch zu zeigen: C ist konvex

Konvexe Hülle

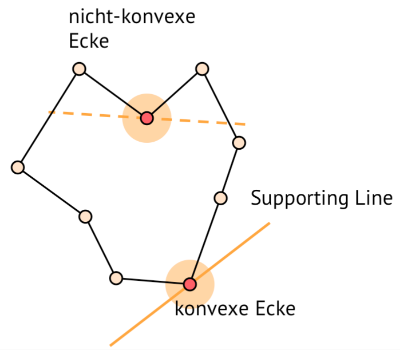
$$q = \sum_{i=1}^n \alpha_i p_i, \quad \sum_{i=1}^n \alpha_i = 1, \quad \alpha_i \geq 0$$
$$r = \sum_{i=1}^n \beta_i p_i, \quad \sum_{i=1}^n \beta_i = 1, \quad \beta_i \geq 0$$

$$\begin{aligned}(1-t)q + tr &= (1-t)\sum_{i=1}^n \alpha_i p_i + t\sum_{i=1}^n \beta_i p_i \\ &= \sum_{i=1}^n (1-t)\alpha_i p_i + t\beta_i p_i \\ &= \sum_{i=1}^n ((1-t)\alpha_i + t\beta_i)p_i\end{aligned}$$

$$\begin{aligned}\sum_{i=1}^n ((1-t)\alpha_i + t\beta_i) &= (1-t)\sum_{i=1}^n \alpha_i + t\sum_{i=1}^n \beta_i \\ &= (1-t) + t \\ &= 1\end{aligned}$$

$$(1-t)\alpha_i + t\beta_i \geq 0$$

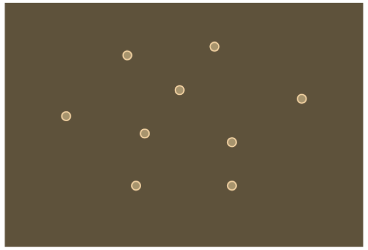
Lokale Konvexität



Konvexe Hülle

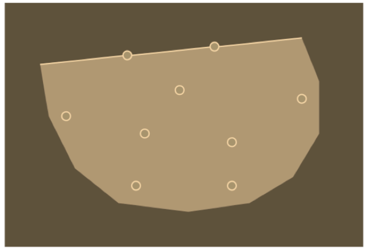
- Jede Supporting Line definiert einen Halbraum, in dem sich alle Eckpunkte befinden.
- Die konvexe Hülle ergibt sich aus dem Schnitt aller dieser Halbräume (ohne Beweis).

Konvexe Hülle

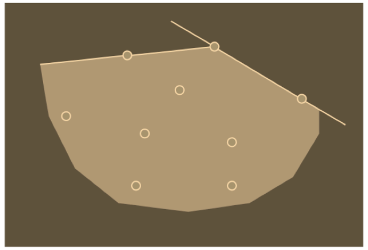


Wähle „Supporting Lines“ so knapp wie möglich. Sobald diese Gerade zwei Punkte schneiden, sind sie so knapp wie möglich und können nicht mehr verbessert werden.

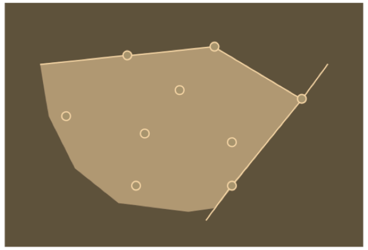
Konvexe Hülle



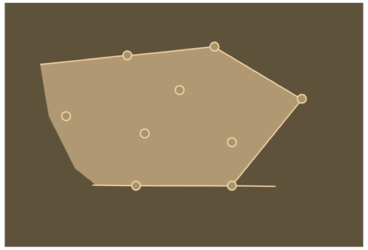
Konvexe Hülle



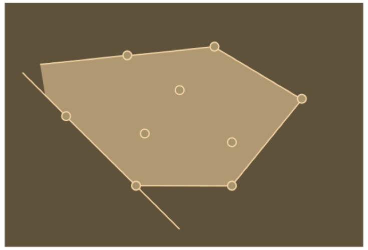
Konvexe Hülle



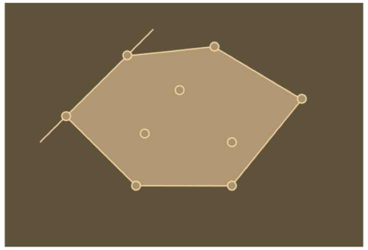
Konvexe Hülle



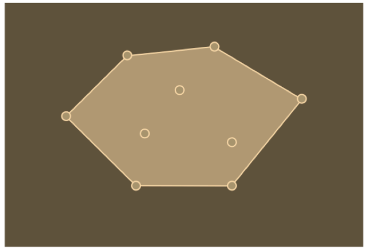
Konvexe Hülle



Konvexe Hülle



Konvexe Hülle



„Gift Wrapping“ = Man wickelt die Punkte so „platzsparend“ wie möglich ein.

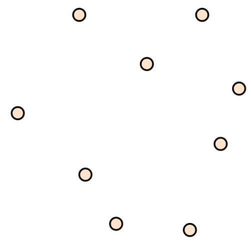
Lokale Konvexität

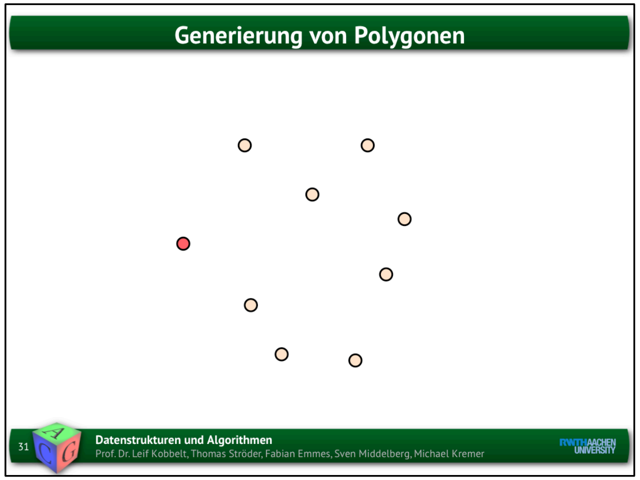
- Lemma
  Der Eckpunkt mit der kleinsten x-Koordinate (und ggfs. mit der kleinsten y-Koordinate) ist immer konvex.
- Beweis: Supporting Line parallel zur y-Achse

Generierung von Polygonen

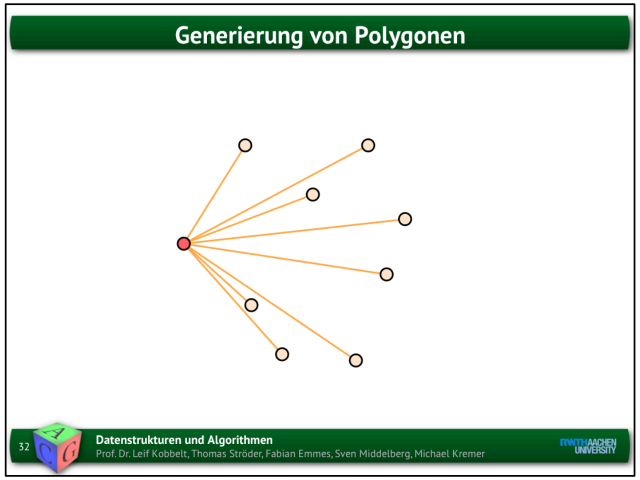
- Suche eine konvexe Ecke
- Sortiere die übrigen Ecken nach dem Winkel zur x-Achse
- Verbinde aufeinanderfolgende Ecken

Generierung von Polygonen



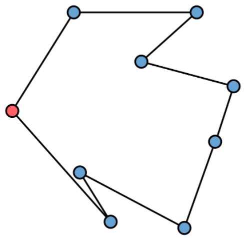


Wir nehmen uns wieder einen beliebigen Punkt heraus.



Und verbinden alle anderen Punkte mit ihm. Nun sortiert man alle Verbindungslininen nach Winkel und verbindet sie.

Generierung von Polygonen

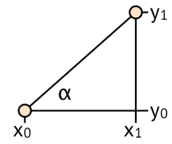


Dies ist zwar kein schönes Polygon, aber ein gültiges.

Sortierung nach Winkel

- Winkel der Kante von (x_0,y_0) nach (x_1,y_1) zur x-Achse

$$\tan \alpha = \frac{y_1-y_0}{x_1-x_0}$$

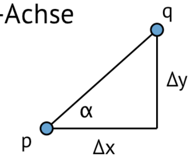


$$(\sin \alpha)^2 = \frac{(y_1-y_0)^2}{(x_1-x_0)^2 + (y_1-y_0)^2}$$

Sortierung nach Winkel

- Winkel α (p,q) der Kante von p=(x,y) nach q=(x+Δx,y+Δy) zur x-Achse

$$\tan \alpha = \frac{\Delta y}{\Delta x}$$



- Nachteile
  – Berechnung des Arcustangens ist teuer.
  – Abfangen von Division durch 0.
  – Prüfung, in welchem Quadrant der Punkt liegt, notwendig.

Sortierung nach Winkel

- Besser: Wähle einfache Funktion β mit β $(p,q_0) < \beta$ $(p,q_1) \Leftrightarrow \alpha$ $(p,q_0) < \alpha$ (p,q_1)
- Wähle z.B. im ersten Quadrant

$$\beta(p,q) = \frac{\Delta y}{\Delta x + \Delta y} \in [0,1]$$

- Beta(dx,dy)

```
beta ← dy / (abs(dx) + abs(dy))
if dx < 0 then return 2 − beta
if dy < 0 then return 4 + beta
return beta
```


Wenn man den Algorithmus implementieren möchte, benutzt man diese Vereinfachung, um die Benutzung des Arcustangens zu vermeiden. Bestimmung des Quadranten durch Fallunterscheidung.

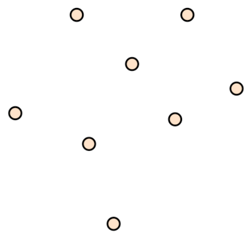
Konvexe Hülle

- Wir nehmen immer an, dass die Punkte in der konvexen Hülle einer Punktmenge so sortiert sind, dass das zugehörige Polygon gegen den Uhrzeigersinn orientiert ist.

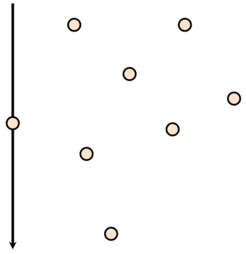
Gift Wrapping

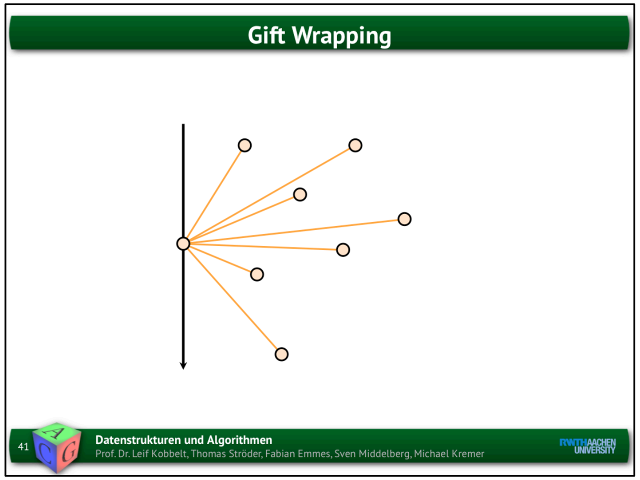
- Beginne mit einer Supporting Line.
- Ergänze in jedem Schritt den Eckpunkt, der den geringsten Winkel zur aktuellen Supporting Line aufspannt.

Gift Wrapping

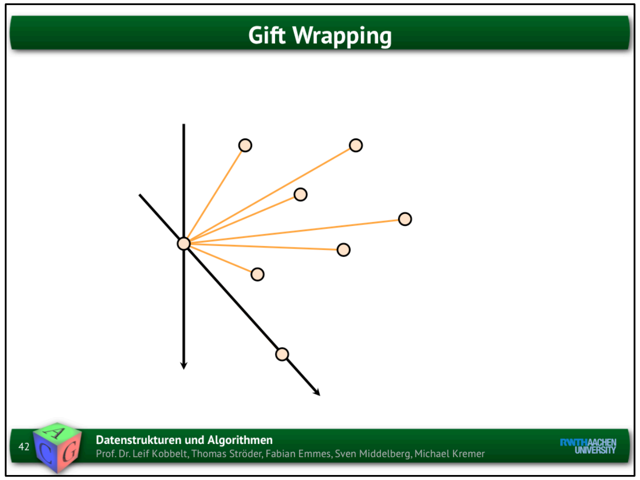


Gift Wrapping

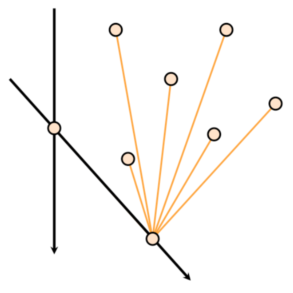




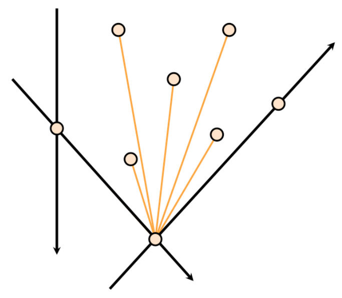
Vom Startpunkt wissen wir, dass er auf der konvexen Hülle liegt (Lemma). Nun nehmen wir die Verbindungsgerade mit dem geringsten Winkel zur Supporting Line.

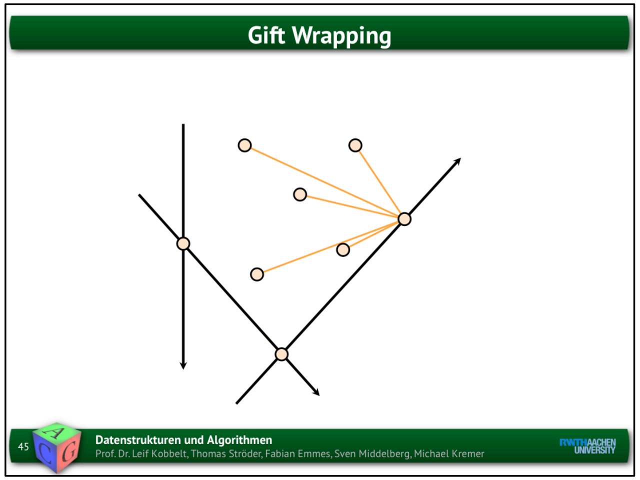


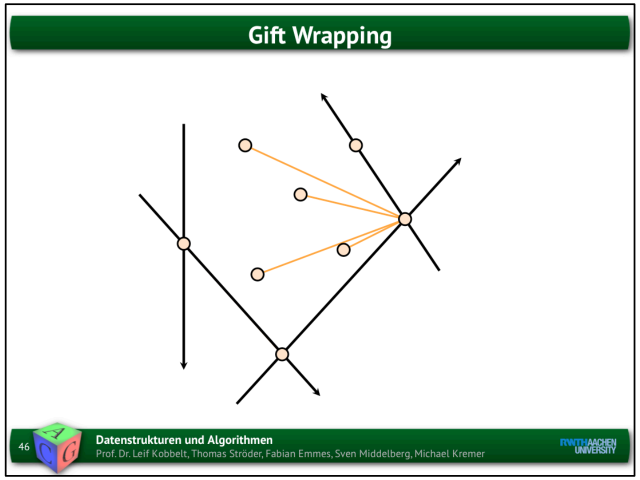
Gift Wrapping



Gift Wrapping







Und so weiter...

Gift Wrapping

```
• GiftWrap( pts[1..n] )
    p ← LeftMostPoint( pts )
    q ← Null
    CH.create()
    CH.add( p )
    L ← negative y-axis
    while q ≠ CH.top() do
        q ← select point with smallest angle
  (p,L)
        CH.add( q )
        L ← Line(p,q)
        p ← q
```


Min/Max aus einer Folge kann in O(n) bestimmt werden.

Gift Wrapping

- Die Gift Wrapping Prozedur entspricht im Wesentlichen dem Selection Sort.
- Aufwand $O(m \times n)$, wobei m die Zahl der Eckpunkte in der konvexen Hülle ist
- Best case: $m = 3$
- Worst case: $m = n$

3 <= m <= n. Best-Case: Dreieck. Worst-Case: Alle Punkte liegen auf der konvexen Hülle (z.B. Kreis).

2.7 Geometrische Algorithmen

Es gibt aber noch schlauere Algorithmen für diese Problemstellung. Z.B. Graham's Scan.

Graham's Scan

- Gift-Wrapping führt zuviele redundante Berechnungen durch.
- Bei der einfachen Generierung von Polygonen genügt ein einziger globaler Sortierungsschritt.
- Idee: Entferne die nicht-konvexen Ecken aus dem einfachen Polygon.

Idee: Erzeuge irgendein Polygon und entferne lokale Nichtkonvexitäten.

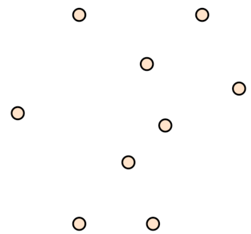
Graham's Scan

- GrahamScan(pts)

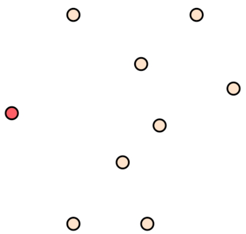
```
let p be the left-most point in pts
sort pts by angle of line (p,pts[i]) to x-axis
p ← head( pts ); q ← next(p); r ← next(q)
while r ≠ p do
   if p,q,r is convex do
      p ← next(p); q ← next(q); r ← next(r)
   else
      delete q
      q ← p; p ← prev(p)
```


Sortieren kann bekanntermaßen in O(nlog(n)) bewerkstelligt werden.

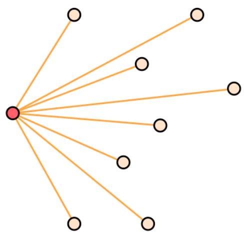
Graham's Scan



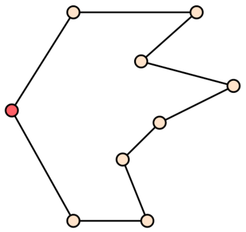
Graham's Scan



Graham's Scan

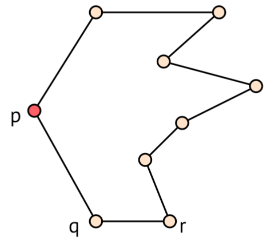


Graham's Scan



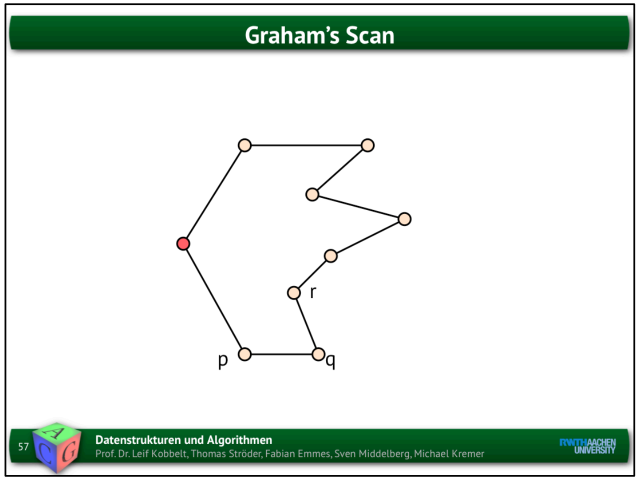
55 **Datenstrukturen und Algorithmen**
Prof. Dr. Leif Kobbelt, Thomas Ströder, Fabian Emmes, Sven Middelberg, Michael Kremer

Graham's Scan

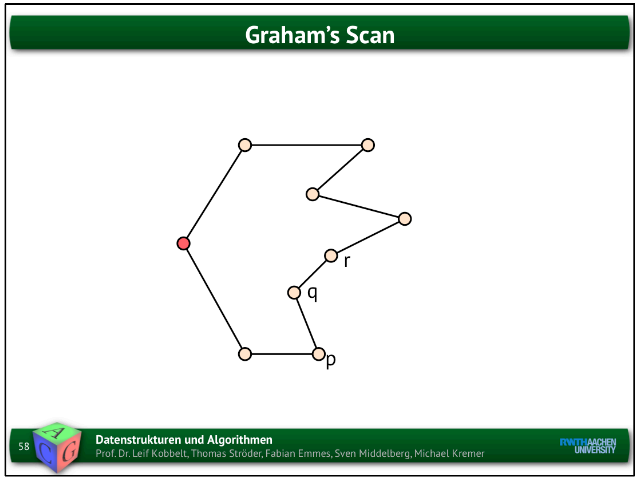


56 **Datenstrukturen und Algorithmen**
Prof. Dr. Leif Kobbelt, Thomas Ströder, Fabian Emmes, Sven Middelberg, Michael Kremer

Wir „biegen links“ ab => Konvex

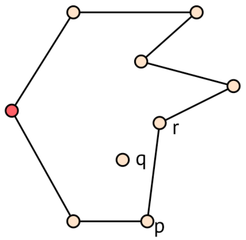


Wir biegen wieder links ab => Konvex

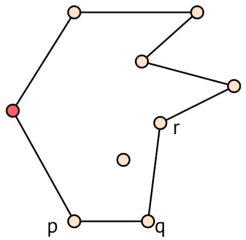


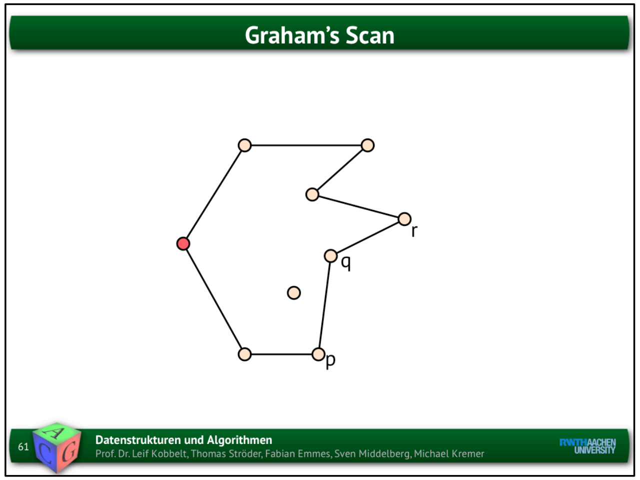
Wir biegen rechts ab => Nicht-konvex, schmeiße Punkt raus.

Graham's Scan

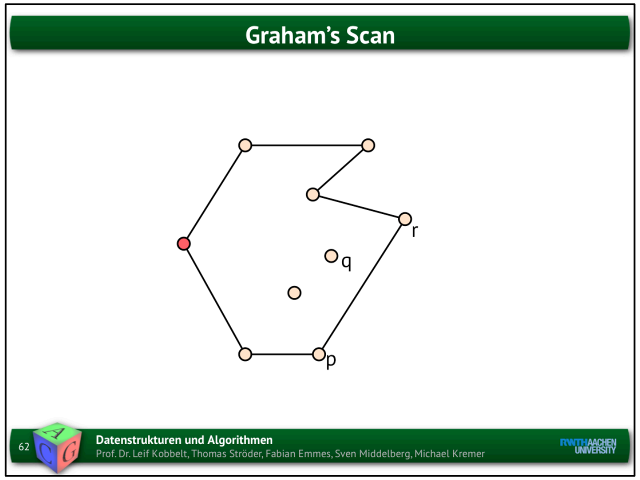


Graham's Scan



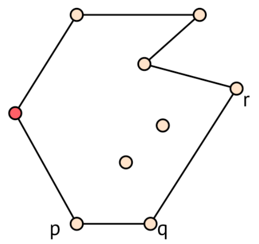


Wieder rechts abbiegen => Nicht-konvex, schmeiße Punkt raus.

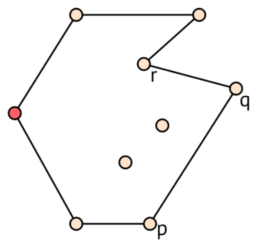


Und so weiter...

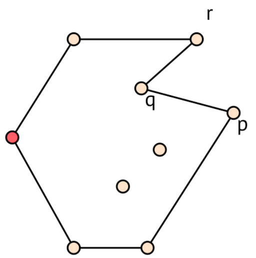
Graham's Scan



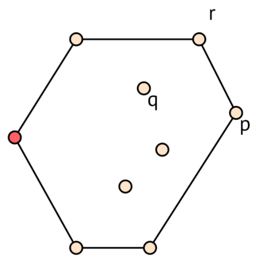
Graham's Scan



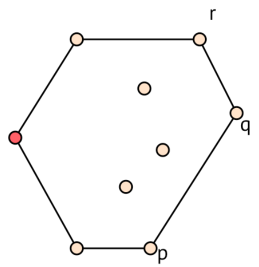
Graham's Scan



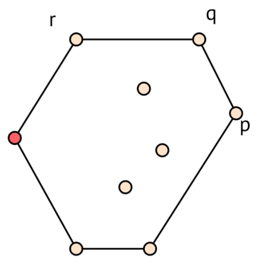
Graham's Scan



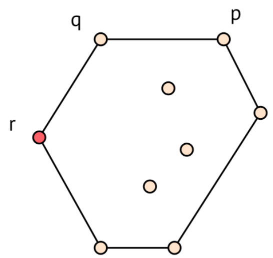
Graham's Scan



Graham's Scan

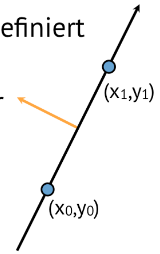


Graham's Scan



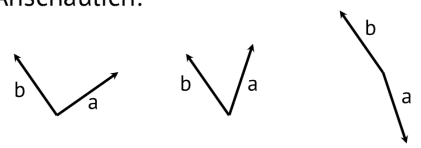
Halbraum-Test

- Es sei eine Supporting-Line durch die Punkte (x_0,y_0) und (x_1,y_1) gegeben
- Die Reihenfolge der Punkte definiert eine Orientierung
- Links ist innen, d.h. der Vektor
  (y_0-y_1, x_1-x_0)
  zeigt senkrecht nach innen



Skalarprodukt

- Erinnerung: Skalarprodukt
  $(x_0,y_0)\cdot(x_1,y_1) = x_0 \times x_1 + y_0 \times y_1$
- Anschaulich:

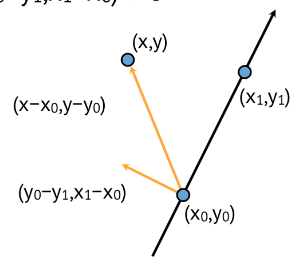


a · b = 0 (⇔ a⊥b)   a · b > 0   a · b < 0

Das Skalarprodukt gibt Auskunft über den Winkel zwischen zwei Vektoren.

Halbraum-Test

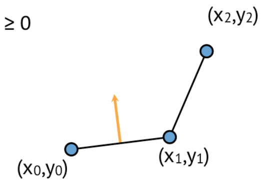
- Punkt (x,y) im Innern:
  $(x-x_0,y-y_0)\cdot(y_0-y_1,x_1-x_0) \geq 0$



Halbraum-Test

- Punkt (x_2,y_2) im Innern:
  $(x_2-x_0,y_2-y_0)\cdot(y_0-y_1,x_1-x_0) \geqslant 0$
  $(x_2-x_0)\times(y_0-y_1)+(y_2-y_0)\times(x_1-x_0) \geqslant 0$

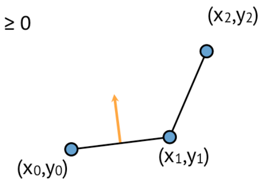
$$\det \begin{bmatrix} x_1-x_0 & x_2-x_0 \\ y_1-y_0 & y_2-y_0 \end{bmatrix} \geq 0$$



Halbraum-Test

- Punkt (x_2,y_2) im Innern:
  $(x_2-x_0,y_2-y_0)\cdot(y_0-y_1,x_1-x_0) \geqslant 0$
  $(x_2-x_0)\times(y_0-y_1)+(y_2-y_0)\times(x_1-x_0) \geqslant 0$

$$\det \begin{bmatrix} x_1-x_0 & x_2-x_1 \\ y_1-y_0 & y_2-y_1 \end{bmatrix} \geq 0$$



Das Kreuzprodukt ist dasselbe wie die Determinante der o.g. Matrix.

Analyse

- GrahamScan(pts)

```
let p be the left-most point in pts                    ] O(n)
sort pts by angle of line (p,pts[i]) to x-axis         ] O(n logn)
p ← head( pts ); q ← next(p); r ← next(q)
while r ≠ p do                                         ]
   if p,q,r is convex do                               ]
      p ← next(p); q ← next(q); r ← next(r)            ] O(2n) =
   else                                                ]   O(n)
      delete q                                         ]
      q ← p; p ← prev(p)                               ]
```


Analyse

- Grahams Scan Algorithmus benötigt zur Berechnung der konvexen Hülle von n Punkten

  $O(n \times \log n)$

2.7 Geometrische Algorithmen

Divide & Conquer

- Die konvexe Hülle von drei Punkten (in allg. Lage) ist ein Dreieck.
- **Divide:**
  Teile die Punkte in zwei Teilmengen mit disjunkter konvexer Hülle.
- **Conquer:**
  Vereinige die konvexen Hüllen der Teilmengen.

Divide & Conquer

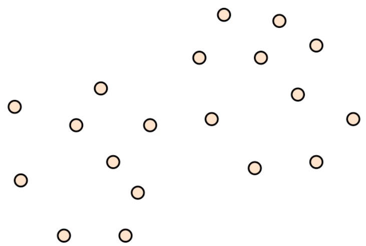
- ConvexHull(pts[1..n])
  sort pts by increasing x-coordinate
  if x ⩾ 3 pts have same x-coordinate then
    remove interior points
  ConvexHullRec(pts[1..n])

Divide & Conquer

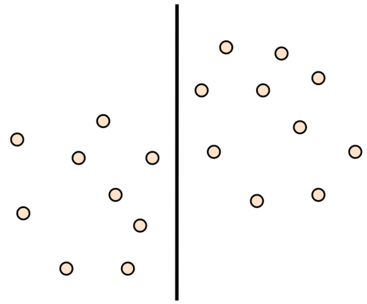
- ConvexHullRec(pts[1..n])
  p ← pts[1..n/2]
  q ← pts[n/2+1..n]
  $[p_1..p_m]$ ← ConvexHullRec(p)
  $[q_1..q_n]$ ← ConvexHullRec(q)
  find upper supporting line (p_i,q_j)
  find lower supporting line (p_k,q_l)
  return $[p_i,..,p_k,q_l,..q_j]$

Man teilt die Menge der Punkte sukkzessive in zwei Hälften.

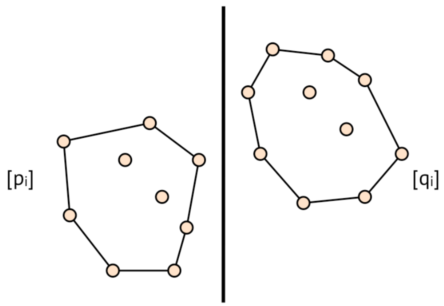
Divide & Conquer



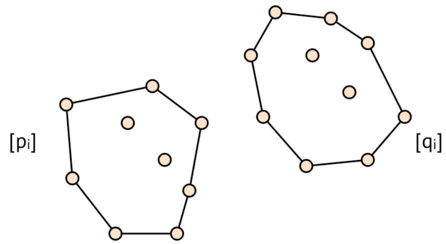
Divide & Conquer

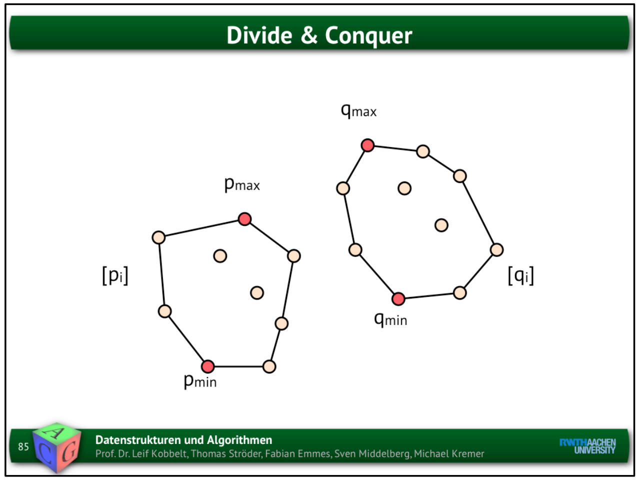


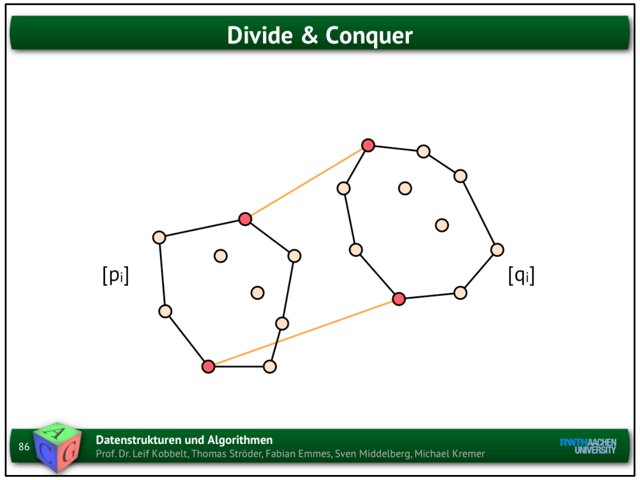
Divide & Conquer



Divide & Conquer

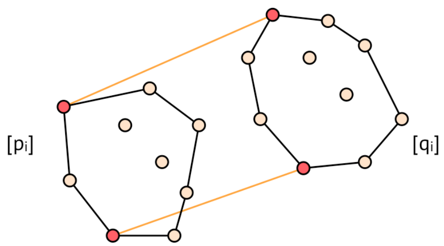




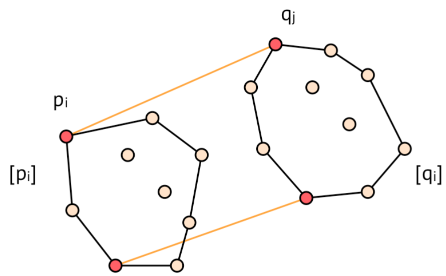


Diese Lösung ist zwar noch nicht optimal, kann aber in wenigen Schritten „repariert“ werden, weil die Initiallösung schon recht gut ist.

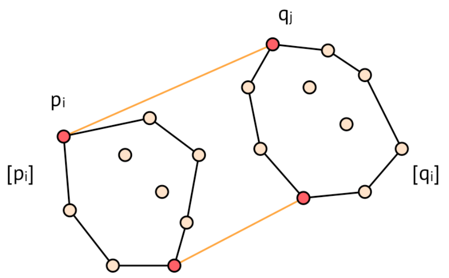
Divide & Conquer



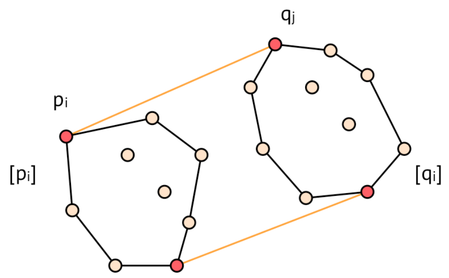
Divide & Conquer



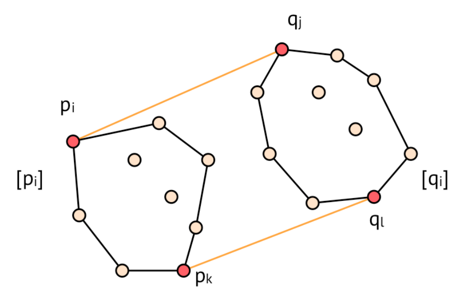
Divide & Conquer



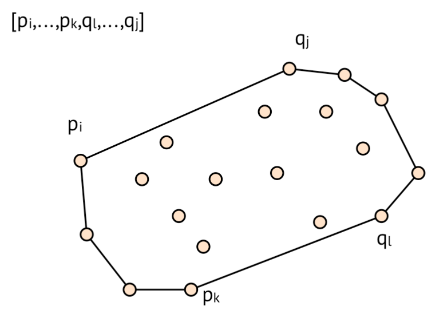
Divide & Conquer



Divide & Conquer



Divide & Conquer



Supporting Lines

- Bestimme Punkte mit maximaler und minimaler y-Koordinate.
- Verschiebe die Supporting Line solange die entsprechende Ecke nicht konvex ist oder der Nachbarpunkt ausserhalb liegt.

Analyse

- ConvexHull(pts[1..n])
  sort pts by increasing x-coordinate] O(nlogn)
  if x ≥ 3 pts have same x-coordinate then] O(n)
  remove interior points
  ConvexHullRec(pts[1..n])

Analyse

- ConvexHullRec(pts[1..n])
  p ← pts[1..n/2]
  q ← pts[n/2+1..n]
  [p_1..p_m] ← ConvexHullRec(p)
  [q_1..q_n] ← ConvexHullRec(q)
  find upper supporting line (p_i,q_j)
  find lower supporting line (p_k,q_l) O(n)
  return [p_i,..,p_k,q_l,..q_j]

Analyse

- T(n) = 2 T(n/2) + O(n)
- Master-Theorem: T(n) = O(n×log n)
- Vorverarbeitung: O(n×log n)
- Gesamtaufwand: O(n×log n)

2.7 Geometrische Algorithmen

Optimaler Algorithmus

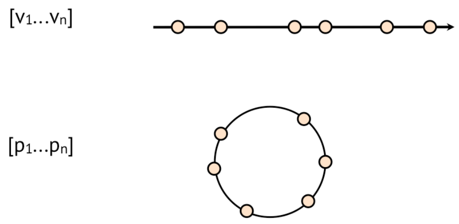
- Erinnerung: Ein optimaler Sortieralgorithmus hat beweisbar Aufwand O(n log n)
- Alle bisher vorgestellten Algorithmen zur Berechnung der konvexen Hülle benutzen einen Sortierschritt. Sie haben daher ebenfalls **mindestens** O(n logn).
- Gibt es einen besseren Algorithmus?

Optimaler Algorithmus

- Sei A ein optimaler CH-Algorithmus
- Verwende ihn zum Sortieren ...
- Eingabe: $[v_1 \ldots v_n]$, oBdA $0 \leqslant v_i < 2\pi$
- Konvertierung: $v_i \longrightarrow p_i = (\cos v_i, \sin v_i)$
- CH-Polygon ergibt Reihenfolge

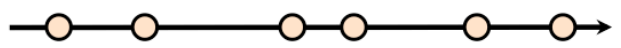
Die Konvertierung bewirkt, dass alle Sample-Punkte auf dem Einheitskreis verteilt liegen.

Optimaler Algorithmus

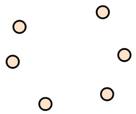


Optimaler Algorithmus

$[v_1 \ldots v_n]$

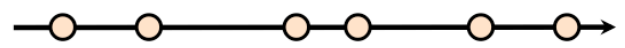


$[p_1 \ldots p_n]$

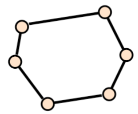


Optimaler Algorithmus

$[v_1 \ldots v_n]$



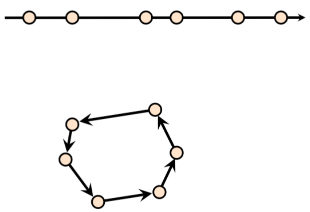
$[p_1 \ldots p_n]$



Optimaler Algorithmus

$[v_1 \ldots v_n]$

$[p_1 \ldots p_n]$



Das Polygon gibt sortierte Folge zurück. Deswegen muss das Problem des Berechnens der konvexen Hülle gleich komplex sein wie das Sortieren.

Optimaler Algorithmus

- Angenommen A ist ein CH-Algorithmus, der schneller als $O(n \log n)$ ist.
- Die Konvertierung $v_i \rightarrow p_i$ liefert einen Sortieralgorithmus, der schneller als $O(n \log n)$ ist.
- Dies ist ein Widerspruch zur Lower Bound von Sortieralgorithmen.