Algorithmische Geometrie

Gesellschaft für Informatik - Fachgruppe 0.1.2 Algorithmische Geometrie

Was ist Algorithmische Geometrie?

Bereits im Altertum haben sich Wissenschaftler wie Pythagoras und Euklid mit geometrischen Problemen beschäftigt. Ihr Interesse galt der Entdeckung geometrischer Sachverhalte und deren Beweis. Sie operierten ausschließlich mit geometrischen Figuren (Punkten, Geraden, Kreisen etc.). Erst die Einführung von Koordinaten durch Descartes machte es möglich, geometrische Objekte durch Zahlen zu beschreiben.

Heute gibt es in der Geometrie verschiedene Richtungen, deren unterschiedliche Ziele man vielleicht an folgendem Beispiel verdeutlichen kann. Denken wir uns eine Fläche im Raum, etwa das Paraboloid, das durch Rotation einer Parabel um seine Symmetrieachse entsteht. In der *Differentialgeometrie* werden mit analytischen Methoden Eigenschaften wie die Krümmung der Fläche an einem Punkt definiert und untersucht.

Die *Algebraische Geometrie* faßt das Paraboloid als *Nullstellenmenge* des Polynoms $p(X,Y,Z) = X^2 + Y^2 - Z$ auf; hier würde man zum Beispiel den Durchschnitt mit einer anderen algebraischen Menge, etwa dem senkrechten Zylinder $(X - x_0^2) + (Y - y_0^2) - r^2$ betrachten und sich fragen, durch welche Gleichungen der Durchschnitt beschrieben wird.

Aus der Mathematik sind uns solche Fragestellungen von einfachen Beispielen vertraut: In der Analysis werden Tangenten an Kurven betrachtet, und in der linearen Algebra immerhin Durchschnitte von Objekten, die sich durch *lineare Gleichungen* beschreiben lassen.

Die *Algorithmische Geometrie*, verfolgt andere Ziele. Ihre Aufgaben bestehen in

- der Entwicklung von *effizienten* und *praktikablen* Algorithmen zur Lösung geometrischer Probleme, und in
- der Bestimmung der *algorithmischen Komplexität* geometrischer Probleme.

Die untersuchten Probleme haben meistens sehr *reale Anwendungshintergründe*. Bei der *Bahnplanung für Roboter* geht es darum, eine Bewegung von einer Anfangskonfiguration in eine Endkonfiguration zu planen, die Kollisionen mit der Umgebung vermeidet und außerdem möglichst effizient ist. Wer je eine Leiter durch verwinkelte Korridore getragen hat, kann sich ein Bild von der Schwierigkeit dieser Aufgabe machen. Sie wächst noch, wenn der Roboter seine Umgebung noch gar nicht kennt, sondern sie während der Ausführung erkunden muß.

Beim *computer aided geometric design (CAGD)* kommt es unter anderem darauf an, Durchschnitt und Vereinigung von dreidimensionalen Körpern schnell zu berechnen. Oder es sollen interpolierende Flächen durch vorgegebene Stützpunkte konstruiert werden.

Bei der Arbeit mit *geographischen Daten*, die in der Regel in Datenbanken gespeichert sind, müssen

Zusammenfassung von Algorithmische Geometrie.

Kapitel 0: Vorbemerkungen:

Inhalt der Vorlesung: Algorithmen und Datenstrukturen zur Lösung geometrischer Probleme.
Beispiele: Robotik / Bewegungsplanung, Computergrafik, CAD, VLSI
Typische Probleme:
- Konvexe Hülle
- Zerlegung in einfache konvexe Teile (zB. Triangulierung).
- Schnittprobleme (zB. Segmente, Polygone)
- Suchprobleme (Memberanfrage in Pktmenge, Nearest-Neighbor, Range-Abfragen).
- Bewegungsplanung (Roboter)
- Hidden line / Surface (Elimination).

Grundlegende Ansätze bzw. Vorgehensweisen:
- Devide & Conquer
- Plane Sweep.
- Hierarchische Darstellung
- Dualität
- Randomisierte Algorithmen.
- Rundungsfehler u. degenerierte Eingaben.

Kapitel I: Konvexe Hüllen.

1.1. Konstruktion der konvexen Hülle einer Pktmenge im $\mathbb{R}^2$ – Grundlagen.

1.1.1. Def: Sei $S \subset \mathbb{R}^2$ Pktmenge.
S heißt konvex: $\Leftrightarrow \forall\, p, q \in S$ gilt: $\overline{pq} \subset S$.
konvexe Hülle einer Menge $S \subset \mathbb{R}^2$ ($CH(S)$) ist die kleinste konvexe Menge, die S enthält.

Wir betrachten nun endliche Pktmengen, dh. $|S| = n \in \mathbb{N}$.

Anm.: Der Rand von $CH(S)$ ist ein konvexes Polygon mit Ecken aus S.
Beweis: Übung.

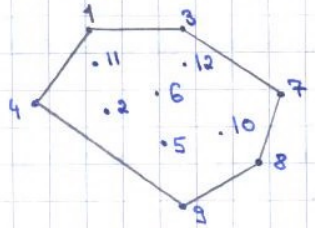
Konvexe Hülle-Problem für endl. Menge $S \subset \mathbb{R}^2$:
Berechne die Folge der Ecken von $CH(S)$ gg. den Uhrzeigersinn (pos. orientiert).
$\rightarrow q_1, \dots q_\ell \in S$. (Anordnung definiert Kanten der Fläche).
Degenerierte Fälle: $\ell = 2$ (alle Pkte sind colinear)
$\ell = 1$ (alle Pkte in S sind gleich).

Problem im $\mathbb{R}^3$ analog, $CH(S)$ ist konvexes Polyeder mit Ecken aus S.

1.1.2. Satz: Berechnung der konvexen Hülle von n Pktn. im $\mathbb{R}^2$ ist mind. so schwer wie das Sortieren von n Zahlen.
Beweis: siehe Üb. (Idee: Führe Sortieren von $x_1 .. x_n$ zurück auf $CH(\{p_1 .. p_n\})$.).

1.1.3. Korollar: Im Allgemeinen braucht die Berechnung von $CH(S)$ Zeit $\Omega(n \log n)$.

1.1.4. Bsp:



$S = \{1..12\}$, $CH(S) = 8,7,3,1,4,9$
(Jedes zyklische Shift)

1.1.5. Vereb: Lexikografische Ordnung (siehe Strings) für Pkte im $\mathbb{R}^2$:
Lexikogr. Sortierung nach kartesischen (x,y)-Koordinaten, dh. für zwei Pkte in der Ebene, also $p, q \in \mathbb{R}^2$ gilt: $p < q :\Leftrightarrow p_x < q_x \vee (p_x = q_x \wedge p_y < q_y)$
Dh. Sortierung von links nach rechts bzw. von unten nach oben (bei gleicher x-Koordinate).

Zusammenfassung von Algorithmische Geometrie.

Kapitel 0: Vorbemerkungen:

Inhalt der Vorlesung: Algorithmen und Datenstrukturen zur Lösung geometrischer Probleme.
Beispiele: Robotik / Bewegungsplanung, Computergrafik, CAD, VLSI
Typische Probleme:
- Konvexe Hülle
- Zerlegung in einfache konvexe Teile (zB. Triangulierung).
- Schnittprobleme (zB. Segmente, Polygone)
- Suchprobleme (Memberanfrage in Pktmenge, Nearest-Neighbor, Range-Abfragen).
- Bewegungsplanung (Roboter)
- Hidden line / Surface (Elimination).

Grundlegende Ansätze bzw. Vorgehensweisen:
- Devide & Conquer
- Plane Sweep.
- Hierarchische Darstellung
- Dualität
- Randomisierte Algorithmen.
- Rundungsfehler u. degenerierte Eingaben.

Kapitel I: Konvexe Hüllen.

1.1. Konstruktion der konvexen Hülle einer Pktmenge im $\mathbb{R}^2$ – Grundlagen.

1.1.1. Def: Sei $S \subset \mathbb{R}^2$ Pktmenge.
S heißt konvex: $\Leftrightarrow \forall\, p, q \in S$ gilt: $\overline{pq} \subset S$.
konvexe Hülle einer Menge $S \subset \mathbb{R}^2$ ($CH(S)$) ist die kleinste konvexe Menge, die S enthält.

Wir betrachten nun endliche Pktmengen, dh. $|S| = n \in \mathbb{N}$.

Anm.: Der Rand von $CH(S)$ ist ein konvexes Polygon mit Ecken aus S.
Beweis: Übung.

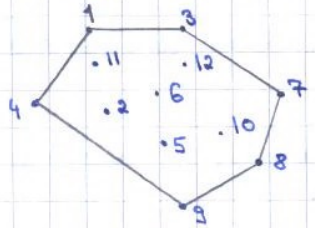
Konvexe Hülle-Problem für endl. Menge $S \subset \mathbb{R}^2$:
Berechne die Folge der Ecken von $CH(S)$ gg. den Uhrzeigersinn (pos. orientiert).
$\rightarrow q_1, \dots q_\ell \in S$. (Anordnung definiert Kanten der Fläche).
Degenerierte Fälle: $\ell = 2$ (alle Pkte sind colinear)
$\ell = 1$ (alle Pkte in S sind gleich).

Problem im $\mathbb{R}^3$ analog, $CH(S)$ ist konvexes Polyeder mit Ecken aus S.

1.1.2. Satz: Berechnung der konvexen Hülle von n Pktn. im $\mathbb{R}^2$ ist mind. so schwer wie das Sortieren von n Zahlen.
Beweis: siehe Üb. (Idee: Führe Sortieren von $x_1 .. x_n$ zurück auf $CH(\{p_1 .. p_n\})$.).

1.1.3. Korollar: Im Allgemeinen braucht die Berechnung von $CH(S)$ Zeit $\Omega(n \log n)$.

1.1.4. Bsp:



$S = \{1..12\}$, $CH(S) = 8,7,3,1,4,9$
(Jedes zyklische Shift)

1.1.5. Vereb: Lexikografische Ordnung (siehe Strings) für Pkte im $\mathbb{R}^2$:
Lexikogr. Sortierung nach kartesischen (x,y)-Koordinaten, dh. für zwei Pkte in der Ebene, also $p, q \in \mathbb{R}^2$ gilt: $p < q :\Leftrightarrow p_x < q_x \vee (p_x = q_x \wedge p_y < q_y)$
Dh. Sortierung von links nach rechts bzw. von unten nach oben (bei gleicher x-Koordinate).

1.2 Algorithmus I: Gift Wrapping.

Intuitiv: Punktmenge wird mit Geschenkpapier eingewickelt.

Zunächst: $S \subset \mathbb{R}^2$. Voraussetzung: $|S| < \infty$

Geo: $CH(S) = p_1 .. p_n$.

1.2.1. Idee:

1) Startpkt p_1 : Pkt mit der kleinsten y-Koordinate.
(Falls mehrere wähle den linkesten. DH. Minimum in p_x - lexikogr. Ordnung.
(Übung : p_1 ist Ecke der Konvexen Hülle).

2) Wie findet man p_2? (Nachfolger von p_1 auf Konv. Hülle).
Betrachte horizontalen Strahl ℓ durch p_1 (nach rechts). Drehen den Strahl gg. den Uhrzeigersinn bis wir einen weiteren Pkt von S treffen. (Ann. $|S| > 1$).
So erhalten wir p_2.
Genauer: $\forall q \in S \setminus \{p_1\}$ sei α_q der Winkel zwischen ℓ und $\overline{p_1 q}$.
Wähle p_2 so, dass α_{p_2} minimal. Falls mehrere Pkte mit minimalen Winkel, wähle den Pkt mit maximaler Entfernung zu p_1.
$\rightsquigarrow$ lineare Suche in S nach Minimum bzgl. oben definierter Ordnung.

3) Setze Algor. bei p_2 fort, dh. ℓ wird nun der Strahl, der
- in p_2 beginnt.
- in Richtung $p_1 p_2$ zeigt

$\rightsquigarrow$ Suche Minimum in $S \setminus \{p_1, p_2\}$.

4) Wiederhole bis p_1 wieder erreicht wird.

1.2.2. Korrektheit : Übung.

1.2.3. Laufzeit: $CH(S) = p_1 .. p_R$, $|S| = n$.

p_1: lineare Suche in S kostet $O(n)$ (lin. Suche nach Min. bzgl. p_y)

$p_2 .. p_R$: lin. Suche in S kostet $O(n)$ (lin. Suche nach Winkelordnung).

$\Rightarrow$ Gesamtlaufzeit: $O(R \cdot n)$.

1.2.4 Satz:

Sei S eine Menge von n Pkten im $\mathbb{R}^2$ und R die Zahl der Ecken der Konvexen Hülle $CH(S)$. Dann kann $CH(S)$ in Zeit $O(R \cdot n)$ berechnet werden. Bew. siehe oben.

1.2.5. Bem: $R \in \{1..n\}$

Worst Case: $R = n$ (zB. alle Pkte aus S liegen auf einem gemeinsamen Kreis, das Innere ist leer). $\Rightarrow$ Laufzeit: $O(n^2)$.

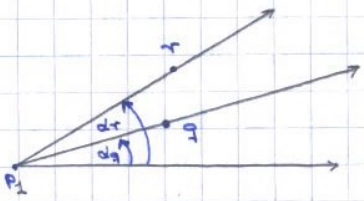
Bester Fall: $R = \text{const.} \Rightarrow$ Laufzeit: $O(R \cdot n) = O(n)$.

1.2.6. Details der Implementierung:

1) Wie findet man Pkt q so, dass α_q minimal ist?

$\rightsquigarrow$ Wir müssen Winkel vergleichen.

```
q ← undefiniert
α_q ← ∞
forall p ∈ S \ {p_1} do
    if α_p < α_q then
        q ← p
        α_q ← α_p
    fi
od
```



Im Allg. 3 Pkte p, q, r $\rightarrow$ vergleiche Winkel

Naiv: Berechne Winkel aus Koordinaten der Pkte mit trigonometrischen Fktnen.

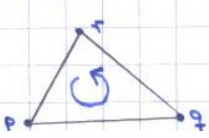
$\Rightarrow$ Nachteile:
- Langsam
- Präzisionsprobleme
- Robustheitsproblem (Definitionsbereich der trigonometrischen Fktnen ist eingeschränkt).

$\Rightarrow$ Besser: Andere Betrachtungsweise:

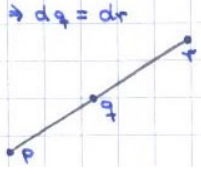
Dazu betrachte Dreieck p, q, r

Drei mögliche Orientierungen:

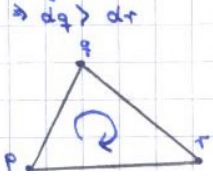
a) positiv orientiert (left-turn) $\Rightarrow \alpha_q < \alpha_r$



b) Orientierung 0 (collinear) $\Rightarrow \alpha_q = \alpha_r$



c) negativ orientiert. (right-turn) $\Rightarrow \alpha_q > \alpha_r$



Allgemein: Seien $p=(p_x,p_y)$, $q=(q_x,q_y)$ und $r=(r_x,r_y)$ drei Pkte im $\mathbb{R}^2$.
Sei $\vec{p}$ der Vektor, der vom Nullpkt ausgeht und in Richtung p zeigt. Und seien $\vec{q}$ und $\vec{r}$ Vektoren, die von p ausgehen und in Richtung q bzw. r zeigen.
Betrachte nun das Parallelogramm, welches durch die Vektoren $\vec{q}$ und $\vec{r}$ aufgespannt wird. Sei $V(P)$ = die Fläche des Parallelogramms.
Beh.: $V(P) = |\det A|$, wobei $A = \begin{pmatrix} q_x-p_x & q_y-p_y \\ r_x-p_x & r_y-p_y \end{pmatrix}$

Bem.: diese Beh. überträgt sich auch auf $\mathbb{R}^n$.
Man betrachte hierbei $u_1..u_n \in \mathbb{R}^n$ und $P = \{u_1a_1+..+u_na_n : 0\le a_i\le 1, i\in\{1..n\}\}$.
$\Rightarrow V(P) = |\det A|$, A entsprechend.

Wir betrachten Δpqr:

↳ **1.2.6.1 Satz**: $p,q,r\in\mathbb{R}^2$, $A = \begin{pmatrix} q_x-p_x & q_y-p_y \\ r_x-p_x & r_y-p_y \end{pmatrix}$.

$\Rightarrow$ (i) $|\det A|$ = 2. Fläche von Δpqr.
(ii) sign(det A) gibt die Orientierung an:
-1: neg. orientiert (right turn)
0: collinear.
$+1$: pos. orientiert (left turn).

Beweis siehe Fernrohrer.

↳ **1.2.6.2 Def**: Wir definieren das geometrische Prädikat:

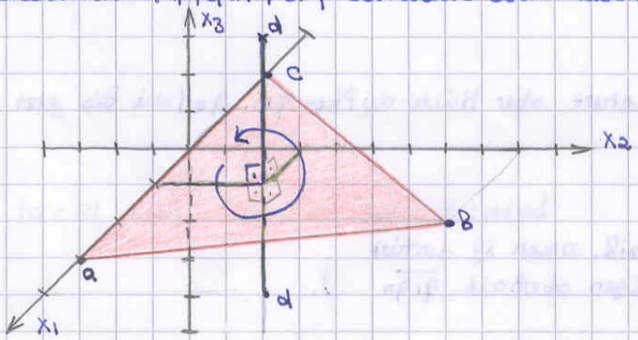
$$\text{orientation}(p,q,r) = \text{sign}\begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix} \left(= \text{sign}\big((q_x-p_x)(r_y-p_y)-(r_x-p_x)(q_y-p_y)\big)\right.$$

Dh. orientation(p,q,r) und damit der Test $d_r < d_q$ kann mit zwei Multiplikationen und fünf Subtraktionen berechnet werden.

2) Falls $d_r = d_q$ (dh. orientation$(p,q,r)=0$), müssen wir herausfinden, welcher der Pkte q oder r weiter von p entfernt liegt.
Naiv: Berechne Distanzen (mit eukl. Metrik):
$\text{dist}(p,q) = \sqrt{(p_x-q_x)^2+(p_y-q_y)^2}$
$\rightsquigarrow$ Nachteil: Wurzel!
$\Rightarrow$ Besser: Vergleiche Quadrate der Distanzen:
$(p_x-q_x)^2+(p_y-q_y)^2 \overset{?}{\lessgtr} (p_x-r_x)^2+(p_y-r_y)^2$?
($\rightsquigarrow$ 4 Subtraktionen, 4 Multiplikationen, 2 Additionen).

1.2.7. Übertragung auf höhere Dimensionen:

Das Orientierungsprädikat kann auf höhere Dimensionen verallgemeinert werden.
Insbesondere im $\mathbb{R}^3$: seien $a,b,c,d\in\mathbb{R}^3$
orientation$(a,b,c,d)\in\{-1,0,1\}$
Sei orientation$(a,b,c,d)\neq 0$, betrachte Ebene durch a,b,c.



↳ **1.2.7.1 Satz**: Für $a,b,c,d\in\mathbb{R}^3$ betrachte Ebene durch a,b,c.
Falls orientation $\neq 0$, so gilt:
(i) orientation$(a,b,c,d)=-1$, wenn von d aus $\Delta a,b,c$ negativ orientiert ist.
orientation$(a,b,c,d)=+1$, wenn von d aus $\Delta a,b,c$ positiv orientiert ist.
(ii) $$\text{orientation}(a,b,c,d) = \text{sign}\begin{vmatrix} a_{x_1} & a_{x_2} & a_{x_3} & 1 \\ b_{x_1} & b_{x_2} & b_{x_3} & 1 \\ c_{x_1} & c_{x_2} & c_{x_3} & 1 \\ d_{x_1} & d_{x_2} & d_{x_3} & 1 \end{vmatrix}$$

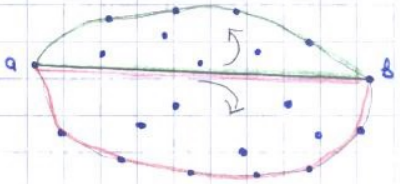
1.3. Algorithmus II: Graham's Scan.

1.3.1. Idee: Berechne "obere" und "untere" Hülle getrennt.

Sei a der linkeste und b der rechteste Pkt von S, dh. a = Minimum bzgl. x-y-Sortierung und b = Maximum bzgl. xy-Sortierung.

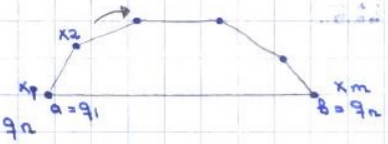
Obere Hülle = Der Teil von CH(S), der oberhalb von $\overline{ab}$ liegt (inclusive a,b). $= CH(S_1), S_1 \subset S$.

Untere Hülle = Der Teil von CH(S), der unterhalb von $\overline{ab}$ liegt (incl. a,b) $= CH(S_2), S_2 \subset S$.



Beobachtung: $S_1 = \{p \in S : \text{orientation}(a,b,p) \geq 0\}$
und $S_2 = \{p \in S : \text{orientation}(a,b,p) \leq 0\}$.

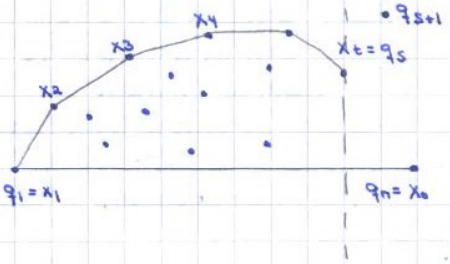
Sei S_1 lexikografisch nach xy-sortiert ($\to n \log n$)
$\rightsquigarrow$ sortierte Folge $q_1 \dots q_n = S_1$
$\Rightarrow a = q_1$ und $b = q_n$
Berechne obere Hülle als Teilfolge $x_1 .. x_m$ von $q_1 .. q_n$ (dh. im Uhrzeigersinn).



Beobachtung:
1. Jeweils drei aufeinanderfolgende Ecken x_i, x_{i+1}, x_{i+2} bilden einen Rightturn, dh. orientation $(x_i, x_{i+1}, x_{i+2}) < 0$.
2. (b,a,p) ist Rightturn $\forall p \in S_1$.
(dh. $\forall p \in S_1$: orientation $(q_n, q_1, p) < 0$).

1.3.2. Algorithmus:

verwalte stack $x_0, x_1 .. x_t$ von potentiellen Ecken der oberen Hülle.
Am Ende: genau die Ecken der oberen Hülle.
Initialisierung: stack $S = q_n, q_1, q_2$. Bem: alle drei Pkte wg. lexik. Sort. genau festgelegt und müssen nicht gesucht werden.
Schleife: Betrachte nur $q_3 .. q_{n-1}$.
Sei q_s aktueller Pkt
Invariante: $x_0, x_1 .. x_t$ ist Teilfolge von $q_n, q_1 .. q_s$



mit: 1) $t \geq 2$, $x_0 = q_n$, $x_1 = q_1$, $x_t = q_s$
besser x_1 statt x_0 2) $x_0 x_1 .. x_t$ ist konvexes Polygon
3) obere Hülle von S ist Teilfolge von $x_1 .. x_t\, q_{s+1} .. q_n$.

Schritt: $s \mapsto s+1$
while (x_{t-1}, x_t, q_{s+1}) nicht Rightturn do
pop (x_t)
od
push (q_{s+1}).

Beobachtung: stack speichert obere Hülle der Pkte $q_1 .. q_s$ (die bis jetzt gewesen).

Die Funktion:
UPPER_HULL $(q_1 .. q_n)$ besser: $q_1 .. q_m \Rightarrow b = q_m$ (weil $|S| = n$)
(Vorbed. 1). $q_1 .. q_n$ lexik. nach xy sortiert
2) $q_2 .. q_{n-1}$ liegen oberhalb $\overline{q_1 q_n}$).
Stack von Pkten. S
(stack <point> S_i)
S.push (q_n);
S.push (q_1);
S.push (q_2);
(Ann. $n \geq 2$).

→

```
for s=3 to n-1 do
    x ← S.top();          gibt das oberste Element an.
    y ← S.top_pred();     eins unter dem obersten auf dem Stack
    while orientation (y, x, q_s) ≥ 0 do
        S.pop();
        x ← y;
        y ← S.top_pred();
    od
    S.push (q_s)
od
return S.
```

While-Schleife terminiert spätestens, wenn S nur noch die Pkte q_n, q_1 enthält. (weil alle Pkte $q_2 .. q_{n-1}$ oberhalb von $\overline{q_1 q_n}$ liegen.).

1.3.3. Beispiel für die Pkt:

Anfang: Stack $q_6 q_1 q_2$.

$s = 3$

$x = q_2 \wedge y = q_1$

orientation $(q_1 q_2 q_3) = -1$

$\Rightarrow$ Stack $q_6 q_1 q_2 q_3$

$s = 4$

$x = q_3 \wedge y = q_2$.

orientation $(q_2 q_3, q_4) = -1 \Rightarrow$ Stack $q_6 q_1 q_2 q_3 q_4$

$s = 5$

$x = q_4 \wedge y = q_3$

orientation $(q_3 q_4 q_5) = +1 \Rightarrow$ stack $q_6 q_1 q_2 q_3$, $x = q_3, y = q_2$

orientation $(q_2 q_3 q_5) = -1 \Rightarrow$ stack $q_6 q_1 q_2 q_3 q_5$

fertig!

1.3.4. Korrektheit.

(1) Invariante ist stets erfüllt.

Zeige mit Induktion, dass Invariante stets erfüllt ist.

Ind.anf: $S = q_n q_1 q_2$

$x_0, x_1, x_2 = q_n q_1 q_2$ mit 1) $t = 2$, $q_n = x_0$, $q_1 = x_1$, $q_2 = x_2$

2) $x_0 x_1 x_2$ konv. Polygon

3) obere Hülle von S ist TF von $x_0 x_1 x_2 q_3 \ldots q_{n-1}$.

Ind.ann: die Invariante sei für $x_0 x_1 .. x_t$ erfüllt $(t \in \mathbb{N})$.

Indbeh: Sei q_s der nächste Pkt bzgl. der lex.Kogr. Sortierung.

Wenn orientation $(x_{t-1}, x_t, q_s) = -1$ dann ist die Beh. trivialerweise erfüllt.

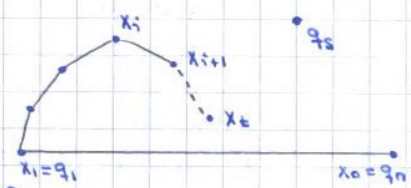
Sei also orientation $(x_{t-1}, x_t q_s) \in \{1, 0\}$

z.z. $x_0 \ldots x_t$ ist TF von $q_n q_1 .. q_s$ mit

1) $t \geq 2$, $x_0 = q_n$, $x_1 = q_1$, $x_t = q_s$

2) $x_0 .. x_t$ konv. Polygon

3) obere Hülle von S ist TF von $x_1 .. x_t q_{s+1} .. q_n$.



$\Rightarrow S = x_0 x_1 .. x_{t-1}$

... wird so lange gepopt bis orientation $(y, x, q_s) < 0$.

überall x_1.

$\Rightarrow S = x_0 x_1 .. x_i$

S.push $(q_s) \Rightarrow S = x_0 x_1 .. x_i x_t$, wobei $x_t := q_s$

$\Rightarrow x_0 x_1 .. x_i x_t$ ist TF von $q_n q_1 \ldots q_s$

zu i) $\underbrace{t \geq 2}$, $x_0 = q_n$, $x_1 = q_1$ und $x_t = q_s$ nach Def. $\Rightarrow$ ok.
weil $(x_0 x_1 q_s)$ Rightturn $\Rightarrow i \geq 1 \Rightarrow t \geq 2$.

zu 2). Folge $x_0 .. x_i$ unverändert und (x_{i-1}, x_i, q_s) bilden Rightturn

$\Rightarrow x_0 .. x_i x_t$ auch konvexes Polygon.

zu 3) Pkte $x_{i+1} .. x_{t-1}$ wurden entfernt. Sie gehören nicht zu oberen Hülle, da sie rechts von (oder auf) Strecke $\overline{x_i q_s}$ liegen. $\Rightarrow$ ok.

$\Rightarrow$ Beh.

(2). Bei Termination enthält Stack S die obere Hülle.

Voraussetzung: die Invariante ist für den letzten Pkt ($s=n$) erfüllt.

d.h. alle Pkte sind abgearbeitet.

Wissen also: $x_1 .. x_t$ ist TF von $q_1 .. q_n$ mit bsp: läuft ab 1, weil $x_0 = x_t = q_n$.

1) $t \geq 2$, $x_0 = q_n$, $x_1 = q_1$ und $x_t = q_n$

2) $x_1 .. x_t$ konv. Polygon

3). obere Hülle ist TF von $x_1 .. x_t$.

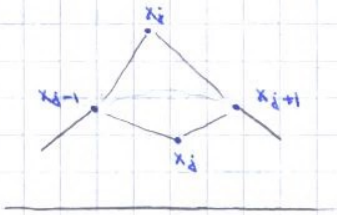
z.z. obere Hülle $= x_1 .. x_t$ also nicht echte Teilmenge.

Ann: nein, obere Hülle $\subset x_1 .. x_t$

$\Rightarrow \exists\, x_j \in \{x_1 .. x_t\}$ mit $x_j \notin$ obere Hülle

$\rightarrow$ 1.Fall: orientation $(x_{j-1}, x_j, x_{j+1}) = +1$ $\Rightarrow$ ↯ zu 2)

2.Fall: orientation $(x_{j-1}, x_j, x_{j+1}) = -1$ $\Rightarrow$ ↯ zu 2).



$\Rightarrow$ Ann. falsch

$\Rightarrow$ Beh.

1.3.5. Laufzeit:

Beobachtung: Jeder Pkt kann höchstens einmal aus S entfernt werden.

Rumpf hat Kosten $O(1)$ (orientation, konstante Zahl von stack-Operationen).

Da jeder Pkt höchstens einmal aus S entfernt werden kann $\Rightarrow$

$\Rightarrow$ while-Schleife wird höchstens n mal eingetreten } $\Rightarrow$ forschleife kostet uns $O(n)$.

In der while-Schleife konst. Operationen, außerhalb auch }

$\Rightarrow$ Gesamtkosten der UPPER-Pkt : $O(n)$.

(Analog berechnet man die untere Hülle).

Die Funktion insgesamt:

CONVEX_HULL(S)		
1. S.sort(xy);	$O(n \log n)$	$(\lvert S\rvert = n)$
a ← S.min();	$O(1)$	
b ← S.max();	$O(1)$	
2. forall $p \in S$ do eigentlich $S \setminus \{a,b\}$		
3. if (orientation(a,b,p) > 0) S_1.append(p)	$O(n)$	(S_1, S_2 Listen $\Rightarrow$ append kostet $O(1)$, n x $\Rightarrow O(n)$)
4. if (orientation(a,b,p) < 0) S_2.append(p)		
5. od		
6. S_1.push(a);	$O(n)$	(falls S_1 Liste $\Rightarrow$
S_1.append(b);	$O(1)$	einfügen am Ende kostet $O(n)$)
7. S_2.push(a);	$O(n)$	
S_2.append(b);	$O(1)$	
8. H_1 ← UPPER_HULL(S_1)	$O(m) = O(n)$	
9. H_2 ← LOWER_HULL(S_2)	$O(n-m) = O(n)$	
10. H_1 umdrehen	$O(n)$	
11. H_2 an H_1 anhängen	$O(1)$	
12 doppeltes Vorkommen von a/b löschen.	$O(n)$	

Zusammenfassung:

CONVEX_HULL(S)	Zeit:
1) Sortiere S lexik.	$O(n \log n)$
2) a ← min ∧ b ← max	$O(1)$
3) $S_1 \leftarrow \{p \in S : b,a,p$ Rechtsturn $\} \cup \{a,b\}$	$O(n)$
4) $S_2 \leftarrow \{p \in S : b,a,p$ Linksturn $\} \cup \{a,b\}$	$O(n)$
5) UPPER_HULL(S_1)	$O(n)$
6) LOWER_HULL(S_2)	$O(n)$
7) " Zusammenkleben	$O(n)$

Allgemein: Seien $p=(p_x,p_y)$, $q=(q_x,q_y)$ und $r=(r_x,r_y)$ drei Pkte im $\mathbb{R}^2$.
Sei $\vec{p}$ der Vektor, der vom Nullpkt ausgeht und in Richtung p zeigt. Und seien $\vec{q}$ und $\vec{r}$ Vektoren, die von p ausgehen und in Richtung q bzw. r zeigen.
Betrachte nun das Parallelogramm, welches durch die Vektoren $\vec{q}$ und $\vec{r}$ aufgespannt wird. Sei $V(P)$ = die Fläche des Parallelogramms.
Beh.: $V(P) = |\det A|$, wobei $A = \begin{pmatrix} q_x-p_x & q_y-p_y \\ r_x-p_x & r_y-p_y \end{pmatrix}$

Bem.: diese Beh. überträgt sich auch auf $\mathbb{R}^n$.
Man betrachte hierbei $u_1..u_n \in \mathbb{R}^n$ und $P = \{u_1a_1+..+u_na_n : 0\le a_i\le 1, i\in\{1..n\}\}$.
$\Rightarrow V(P) = |\det A|$, A entsprechend.

Wir betrachten Δpqr:

↳ **1.2.6.1 Satz**: $p,q,r\in\mathbb{R}^2$, $A = \begin{pmatrix} q_x-p_x & q_y-p_y \\ r_x-p_x & r_y-p_y \end{pmatrix}$.

$\Rightarrow$ (i) $|\det A|$ = 2. Fläche von Δpqr.
(ii) sign(det A) gibt die Orientierung an:
-1: neg. orientiert (right turn)
0: collinear.
$+1$: pos. orientiert (left turn).

Beweis siehe Fernrohrer.

↳ **1.2.6.2 Def**: Wir definieren das geometrische Prädikat:

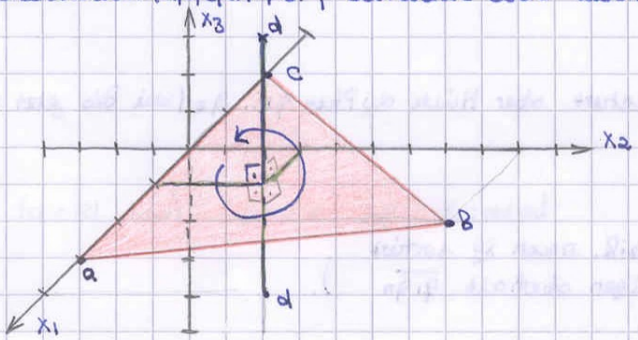
$$\text{orientation}(p,q,r) = \text{sign}\begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix} \left(= \text{sign}\big((q_x-p_x)(r_y-p_y)-(r_x-p_x)(q_y-p_y)\big)\right.$$

Dh. orientation(p,q,r) und damit der Test $d_r < d_q$ kann mit zwei Multiplikationen und fünf Subtraktionen berechnet werden.

2) Falls $d_r = d_q$ (dh. orientation$(p,q,r)=0$), müssen wir herausfinden, welcher der Pkte q oder r weiter von p entfernt liegt.
Naiv: Berechne Distanzen (mit eukl. Metrik):
$\text{dist}(p,q) = \sqrt{(p_x-q_x)^2+(p_y-q_y)^2}$
$\rightsquigarrow$ Nachteil: Wurzel!
$\Rightarrow$ Besser: Vergleiche Quadrate der Distanzen:
$(p_x-q_x)^2+(p_y-q_y)^2 \overset{?}{\lessgtr} (p_x-r_x)^2+(p_y-r_y)^2$?
($\rightsquigarrow$ 4 Subtraktionen, 4 Multiplikationen, 2 Additionen).

1.2.7. Übertragung auf höhere Dimensionen:

Das Orientierungsprädikat kann auf höhere Dimensionen verallgemeinert werden.
Insbesondere im $\mathbb{R}^3$: seien $a,b,c,d\in\mathbb{R}^3$
orientation$(a,b,c,d)\in\{-1,0,1\}$
Sei orientation$(a,b,c,d)\neq 0$, betrachte Ebene durch a,b,c.



↳ **1.2.7.1 Satz**: Für $a,b,c,d\in\mathbb{R}^3$ betrachte Ebene durch a,b,c.
Falls orientation $\neq 0$, so gilt:
(i) orientation$(a,b,c,d)=-1$, wenn von d aus $\Delta a,b,c$ negativ orientiert ist.
orientation$(a,b,c,d)=+1$, wenn von d aus $\Delta a,b,c$ positiv orientiert ist.
(ii) $$\text{orientation}(a,b,c,d) = \text{sign}\begin{vmatrix} a_{x_1} & a_{x_2} & a_{x_3} & 1 \\ b_{x_1} & b_{x_2} & b_{x_3} & 1 \\ c_{x_1} & c_{x_2} & c_{x_3} & 1 \\ d_{x_1} & d_{x_2} & d_{x_3} & 1 \end{vmatrix}$$

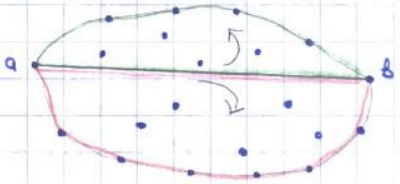
1.3. Algorithmus II: Graham's Scan.

1.3.1. Idee: Berechne "obere" und "untere" Hülle getrennt.

Sei a der linkeste und b der rechteste Pkt von S, dh. a = Minimum bzgl. x-y-Sortierung und b = Maximum bzgl. xy-Sortierung.

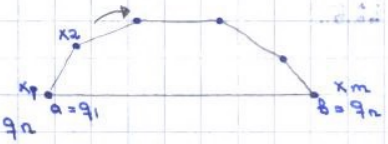
Obere Hülle = Der Teil von CH(S), der oberhalb von $\overline{ab}$ liegt (inclusive a,b). $= CH(S_1), S_1 \subset S$.

Untere Hülle = Der Teil von CH(S), der unterhalb von $\overline{ab}$ liegt (incl. a,b) $= CH(S_2), S_2 \subset S$.



Beobachtung: $S_1 = \{p \in S : \text{orientation}(a,b,p) \geq 0\}$
und $S_2 = \{p \in S : \text{orientation}(a,b,p) \leq 0\}$.

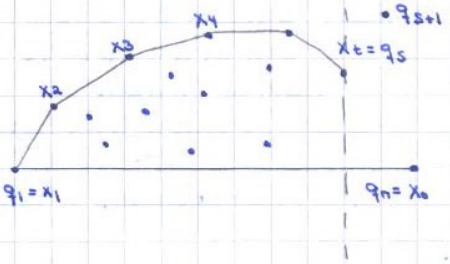
Sei S_1 lexikografisch nach xy-sortiert ($\to n \log n$)
$\rightsquigarrow$ sortierte Folge $q_1 \dots q_n = S_1$
$\Rightarrow a = q_1$ und $b = q_n$
Berechne obere Hülle als Teilfolge $x_1 .. x_m$ von $q_1 .. q_n$ (dh. im Uhrzeigersinn).



Beobachtung:
1. Jeweils drei aufeinanderfolgende Ecken x_i, x_{i+1}, x_{i+2} bilden einen Rightturn, dh. orientation $(x_i, x_{i+1}, x_{i+2}) < 0$.
2. (b,a,p) ist Rightturn $\forall p \in S_1$.
(dh. $\forall p \in S_1$: orientation $(q_n, q_1, p) < 0$).

1.3.2. Algorithmus:

verwalte stack $x_0, x_1 .. x_t$ von potentiellen Ecken der oberen Hülle.
Am Ende: genau die Ecken der oberen Hülle.
Initialisierung: stack $S = q_n, q_1, q_2$. Bem: alle drei Pkte wg. lexik. Sort. genau festgelegt und müssen nicht gesucht werden.
Schleife: Betrachte nur $q_3 .. q_{n-1}$.
Sei q_s aktueller Pkt
Invariante: $x_0, x_1 .. x_t$ ist Teilfolge von $q_n, q_1 .. q_s$



mit: 1) $t \geq 2$, $x_0 = q_n$, $x_1 = q_1$, $x_t = q_s$
besser x_1 statt x_0 2) $x_0 x_1 .. x_t$ ist konvexes Polygon
3) obere Hülle von S ist Teilfolge von $x_1 .. x_t\, q_{s+1} .. q_n$.

Schritt: $s \mapsto s+1$
while (x_{t-1}, x_t, q_{s+1}) nicht Rightturn do
pop (x_t)
od
push (q_{s+1}).

Beobachtung: stack speichert obere Hülle der Pkte $q_1 .. q_s$ (die bis jetzt gewesen).

Die Funktion:
UPPER_HULL $(q_1 .. q_n)$ besser: $q_1 .. q_m \Rightarrow b = q_m$ (weil $|S| = n$)
(Vorbed. 1). $q_1 .. q_n$ lexik. nach xy sortiert
2) $q_2 .. q_{n-1}$ liegen oberhalb $\overline{q_1 q_n}$).
Stack von Pkten. S
(stack <point> S_i)
S.push (q_n);
S.push (q_1);
S.push (q_2);
(Ann. $n \geq 2$).

→

```
for s=3 to n-1 do
    x ← S.top();          gibt das oberste Element an.
    y ← S.top_pred();     eins unter dem obersten auf dem Stack
    while orientation (y, x, q_s) ≥ 0 do
        S.pop();
        x ← y;
        y ← S.top_pred();
    od
    S.push (q_s)
od
return S.
```

While-Schleife terminiert spätestens, wenn S nur noch die Pkte q_n, q_1 enthält. (weil alle Pkte $q_2 .. q_{n-1}$ oberhalb von $\overline{q_1 q_n}$ liegen.).

1.3.3. Beispiel für die Pkt:

Anfang: Stack $q_6 q_1 q_2$.

$s = 3$

$x = q_2 \wedge y = q_1$

orientation $(q_1 q_2 q_3) = -1$

$\Rightarrow$ Stack $q_6 q_1 q_2 q_3$

$s = 4$

$x = q_3 \wedge y = q_2$.

orientation $(q_2 q_3, q_4) = -1 \Rightarrow$ Stack $q_6 q_1 q_2 q_3 q_4$

$s = 5$

$x = q_4 \wedge y = q_3$

orientation $(q_3 q_4 q_5) = +1 \Rightarrow$ stack $q_6 q_1 q_2 q_3$, $x = q_3, y = q_2$

orientation $(q_2 q_3 q_5) = -1 \Rightarrow$ stack $q_6 q_1 q_2 q_3 q_5$

fertig!

1.3.4. Korrektheit.

(1) Invariante ist stets erfüllt.

Zeige mit Induktion, dass Invariante stets erfüllt ist.

Ind.anf: $S = q_n q_1 q_2$

$x_0, x_1, x_2 = q_n q_1 q_2$ mit 1) $t = 2$, $q_n = x_0$, $q_1 = x_1$, $q_2 = x_2$

2) $x_0 x_1 x_2$ konv. Polygon

3) obere Hülle von S ist TF von $x_0 x_1 x_2 q_3 \ldots q_{n-1}$.

Ind.ann: die Invariante sei für $x_0 x_1 .. x_t$ erfüllt $(t \in \mathbb{N})$.

Indbeh: Sei q_s der nächste Pkt bzgl. der lex.Kogr. Sortierung.

Wenn orientation $(x_{t-1}, x_t, q_s) = -1$ dann ist die Beh. trivialerweise erfüllt.

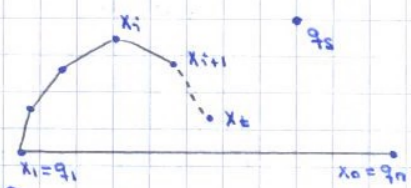
Sei also orientation $(x_{t-1}, x_t q_s) \in \{1, 0\}$

z.z. $x_0 \ldots x_t$ ist TF von $q_n q_1 .. q_s$ mit

1) $t \geq 2$, $x_0 = q_n$, $x_1 = q_1$, $x_t = q_s$

2) $x_0 .. x_t$ konv. Polygon

3) obere Hülle von S ist TF von $x_1 .. x_t q_{s+1} .. q_n$.



$\Rightarrow S = x_0 x_1 .. x_{t-1}$

... wird so lange gepopt bis orientation $(y, x, q_s) < 0$.

überall x_1.

$\Rightarrow S = x_0 x_1 .. x_i$

S.push $(q_s) \Rightarrow S = x_0 x_1 .. x_i x_t$, wobei $x_t := q_s$

$\Rightarrow x_0 x_1 .. x_i x_t$ ist TF von $q_n q_1 \ldots q_s$

zu i) $\underbrace{t \geq 2}$, $x_0 = q_n$, $x_1 = q_1$ und $x_t = q_s$ nach Def. $\Rightarrow$ ok.
weil $(x_0 x_1 q_s)$ Rightturn $\Rightarrow i \geq 1 \Rightarrow t \geq 2$.

zu 2). Folge $x_0 .. x_i$ unverändert und (x_{i-1}, x_i, q_s) bilden Rightturn

$\Rightarrow x_0 .. x_i x_t$ auch konvexes Polygon.

zu 3) Pkte $x_{i+1} .. x_{t-1}$ wurden entfernt. Sie gehören nicht zu oberen Hülle, da sie rechts von (oder auf) Strecke $\overline{x_i q_s}$ liegen. $\Rightarrow$ ok.

$\Rightarrow$ Beh.

(2). Bei Termination enthält Stack S die obere Hülle.

Voraussetzung: die Invariante ist für den letzten Pkt ($s=n$) erfüllt.

d.h. alle Pkte sind abgearbeitet.

Wissen also: $x_1 .. x_t$ ist TF von $q_1 .. q_n$ mit bsp: läuft ab 1, weil $x_0 = x_t = q_n$.

1) $t \geq 2$, $x_0 = q_n$, $x_1 = q_1$ und $x_t = q_n$

2) $x_1 .. x_t$ konv. Polygon

3). obere Hülle ist TF von $x_1 .. x_t$.

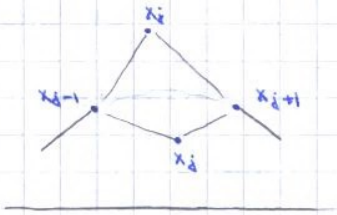
z.z. obere Hülle $= x_1 .. x_t$ also nicht echte Teilmenge.

Ann: nein, obere Hülle $\subset x_1 .. x_t$

$\Rightarrow \exists\, x_j \in \{x_1 .. x_t\}$ mit $x_j \notin$ obere Hülle

$\rightarrow$ 1.Fall: orientation $(x_{j-1}, x_j, x_{j+1}) = +1$ $\Rightarrow$ ↯ zu 2)

2.Fall: orientation $(x_{j-1}, x_j, x_{j+1}) = -1$ $\Rightarrow$ ↯ zu 2).



$\Rightarrow$ Ann. falsch

$\Rightarrow$ Beh.

1.3.5. Laufzeit:

Beobachtung: Jeder Pkt kann höchstens einmal aus S entfernt werden.

Rumpf hat Kosten $O(1)$ (orientation, konstante Zahl von stack-Operationen).

Da jeder Pkt höchstens einmal aus S entfernt werden kann $\Rightarrow$

$\Rightarrow$ while-Schleife wird höchstens n mal eingetreten } $\Rightarrow$ forschleife kostet uns $O(n)$.

In der while-Schleife konst. Operationen, außerhalb auch }

$\Rightarrow$ Gesamtkosten der UPPER-Pkt : $O(n)$.

(Analog berechnet man die untere Hülle).

Die Funktion insgesamt:

CONVEX_HULL(S)		
1. S.sort(xy);	$O(n \log n)$	$(\lvert S\rvert = n)$
a ← S.min();	$O(1)$	
b ← S.max();	$O(1)$	
2. forall $p \in S$ do eigentlich $S \setminus \{a,b\}$		
3. if (orientation(a,b,p) > 0) S_1.append(p)	$O(n)$	(S_1, S_2 Listen $\Rightarrow$ append kostet $O(1)$, n x $\Rightarrow O(n)$)
4. if (orientation(a,b,p) < 0) S_2.append(p)		
5. od		
6. S_1.push(a);	$O(n)$	(falls S_1 Liste $\Rightarrow$
S_1.append(b);	$O(1)$	einfügen am Ende kostet $O(n)$)
7. S_2.push(a);	$O(n)$	
S_2.append(b);	$O(1)$	
8. H_1 ← UPPER_HULL(S_1)	$O(m) = O(n)$	
9. H_2 ← LOWER_HULL(S_2)	$O(n-m) = O(n)$	
10. H_1 umdrehen	$O(n)$	
11. H_2 an H_1 anhängen	$O(1)$	
12 doppeltes Vorkommen von a/b löschen.	$O(n)$	

Zusammenfassung:

CONVEX_HULL(S)	Zeit:
1) Sortiere S lexik.	$O(n \log n)$
2) a ← min ∧ b ← max	$O(1)$
3) $S_1 \leftarrow \{p \in S : b,a,p$ Rechtsturn $\} \cup \{a,b\}$	$O(n)$
4) $S_2 \leftarrow \{p \in S : b,a,p$ Linksturn $\} \cup \{a,b\}$	$O(n)$
5) UPPER_HULL(S_1)	$O(n)$
6) LOWER_HULL(S_2)	$O(n)$
7) " Zusammenkleben	$O(n)$

1.3.6. Satz: Sei S Menge von n Pkten im $\mathbb{R}^2$.
a) CH(S) kann in Zeit O(n log n) berechnet werden (worst case)
b) Falls S lexik. nach xy-Koord. sortiert ist, dann kann CH(S) in Zeit O(n) berechnet werden.

Bew. siehe oben.

1.3.7. Bemerkung:
1) Alg. optimal, da CH-Problem (i.a.) genauso schwierig ist wie sortieren.
Bew. Übung.
2). obere Bzw. untere Hülle sind auch für sich alleine wichtig (für bestimmte Probleme)
3). Optimalität gilt für worst case; es gibt Situationen, in denen Alg. I. besser ist (wenn # Ecken von CH(S) < log n).

1.3.8. Varianten von Graham's Scan:
a) siehe Übung
b). untere u. obere Hülle gleichzeitig berechnen
c). Berechnung der gesamten Hülle in einer Phase.

Idee: 1. Sortiere $S = p_1 .. p_n$.
2. Allgemeiner Schritt:
Bearbeite p_i, $i = 4..n$.
Situation: $C_{i-1} := CH(\{p_1 .. p_{i-1}\})$ ist berechnet.
p_{i-1} ist maximal in $\{p_1 .. p_{i-1}\}$
$\Rightarrow p_{i-1}$ ist die rechteste Ecke von C_{i-1}.
$\Rightarrow \overline{p_{i-1} p_i} \cap C_{i-1} = \{p_{i-1}\}$.

Der eigentliche Schritt:
Berechne Berührpkte t und b der beiden Tangenten von p_i an C_{i-1}.
Genauer: $t \leftarrow p_{i-1}$ d.h. orientation$(p_i, t, succ(t)) \leq 0$
while $(p_i, t, succ(t))$ nicht Rechtsturm do
$t \leftarrow succ(t)$
od
$b \leftarrow p_{i-1}$ d.h. orientation$(p_i, b, pred(b)) \geq 0$
while $(p_i, b, pred(b))$ nicht Rechtsturm do
$b \leftarrow pred(b)$
od

Bea: Die ähnliche Argumentation:
die # unterwegs besuchten Pkte kann in $\Omega(n)$ liegen,
aber jeder von ihnen wird anschließend aus der konvexen Hülle entfernt und kann danach nie wieder als Eckpkt in Erscheinung treten
Also wird er auch nie wieder besucht
$\Rightarrow$ Es kann insgesamt höchstens n Besuche geben
$\Rightarrow$ O(n log n) Laufzeit

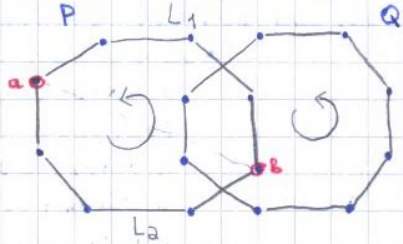
Nachdem wir nun b und t berechnet haben:
- Entferne alle Pkte zwischen b und t
- Füge p_i nach C_i ein.

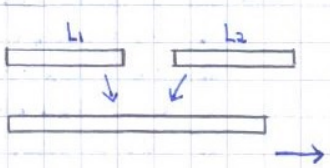
Weitere Implementierungsdetails und Laufzeitanalyse siehe 3. Übung.
Bsp. zu c) siehe Hemdordner.

1.4. Algorithmus III: Devide & Conquer.

Triangulierung schauen

1.4.1. Spezialfall: Konvexe Hülle von zwei konvexen Polygonen P und Q.
$P = p_1 .. p_m$, $Q = q_1 .. q_\ell$, Ecken gg. Uhrzeigersinn sortiert.
Aufgabe: Berechne $CH(P \cup Q)$
Triviale Lsg: Graham's Scan auf die Menge aller Ecken. O(n log n)
Bessere Lsg: Ausnutzen der Polygonstruktur. O(n),
1) Finde Sortierung aller Ecken in Zeit O(n) } O(n)
2) Berechne die Hülle in Zeit O(n) (siehe S.1.3.6.8.). }
zu 1): Betrachte P:
i) Finde extreme Ecken a und B in Zeit O(m)
L_2 := untere Polygonzug
starte bei a und laufe gg. Uhrzeigersinn über P bis B erreicht ist.
L_1 := obere Polygonzug
laufe von b nach a. und drehe um.





(ii) Mische L_1 und L_2 zu einer sortierten Gesamtliste L_P zusammen in Zeit $O(m)$ (→ siehe Mergesort)
Analog: Sortiere Folge L_Q der Ecken von Q in Zeit $O(l)$.
(iii) Mische L_P und L_Q in Zeit $O(m+l) = O(n)$ (wobei $n := m+l$) zu einer sortierten Gesamtfolge zusammen.

→ dann Graham's Scan.

1.4.2. Allgemein: Sei $S \subset \mathbb{R}^2$, $|S| = n$.

```
CONVEX_HULL(S).
if |S| = 1 then
    output S
else
    teile S in zwei möglichst gleich große Teile S1 und S2   } DEVIDE
    (zB. |S1| = ⌈|S|/2⌉ und |S2| = ⌊|S|/2⌋
    P ← CONVEX_HULL(S1)                                      } CONQUER
    Q ← CONVEX_HULL(S2)                                      }
    berechne CH(P ∪ Q) wie in letzter Vorlesung gezeigt.     } MISCHSCHRITT
fi.
```

↑ hier wird die eigentliche Rechnung durchgeführt.

1.4.3. Laufzeit:

Teilen: $O(n)$
Mischen: $O(n)$

Merge auf Listen
Graham's Scan (ohne Sortieren).

$$\Rightarrow T(n) = \begin{cases} c_0, & n=1 \\ c_1 \cdot n + 2 \cdot T(n/2), & n>1 \end{cases}$$

Das ist also dieselbe Rekursion wie für Mergesort (aber mit anderen Konstanten natürlich).
⇒ Gesamtlaufzeit des Verfahrens ist $O(n \log n)$ (siehe Analyse von Mergesort).

1.5. Eine Anwendung von CONVEX HULL.

1.5.1 Problem: Geg. sind n Halbebenen $H_1 \dots H_n$ von $\mathbb{R}^2$
Berechne $P = \bigcap_{i=1}^{n} H_i$

1.5.2 Def: Eine abgeschlossene Halbebene = { alle Pkte auf gleicher Seite einer Geraden h } ∪ h.

1.5.3 Anm: Schnitt P ist konvexes (möglicherweise unbeschränktes) Polygon, da der Schnitt konvexer Mengen wieder konvex ist.

1.5.4 Ziel und Lösungsansatz:

Ziel: Berechne die Folge der Ecken von P gg. den Uhrzeigersinn.
Lösung: Zurückführung auf konvexe Hülle einer geeigneten Pkmenge.
Dazu transformieren wir die definierenden Geraden in "duale" Pkte.

1.5.5 Definition: (Geometr. Transformation)

a) Sei $l = \{ y : \underbrace{y = ax + b}_{\text{Geradengleichung von } l}, x \text{ bel.}, a,b \text{ fest} \}$ eine nicht vertikale Gerade.
Da Punkt $D(l) := (a,b)$ heißt der duale Punkt zu l.

b) Sei $p = (a,b) \in \mathbb{R}^2$ ein Pkt. Die Gerade $D(p) = \{ y : y = -ax + b, x \text{ beliebig} \}$ heißt die duale Gerade zu p.

Abbildung D erlaubt die relative Lage von Objekten.

1.5.6 Lemma: Pkt p liegt auf (oberhalb / unterhalb) einer Geraden l
⇔ Gerade D(p) liegt auf (oberhalb/ unterhalb) Pkt D(l).

Beweis: Sei $p = (p_x, p_y)$, $l = \{ y \in \mathbb{R} : y = ax + b, x \in \mathbb{R} \}$ (a,b fest).
$\Rightarrow D(l) = (a,b)$ und $D(p) = \{ y : y = -p_x x + p_y, x \in \mathbb{R} \}$

- sei p gelegen auf l
  ⇔ $p_y = a \cdot p_x + b$
  ⇔ $-b = a p_x - p_y$
  ⇔ $b = -p_x a + p_y$
  ⇔ D(p) liegt auf D(l).

- sei p gelegen oberhalb l (unterhalb analog).
  ⇔ $ax + b < p_y \ \forall x \in \mathbb{R}$
  Bzw. sogar $a p_x + b < p_y$.
  ⇔ $-a p_x - b > -p_y$
  ⇔ $-a p_x + p_y > b$
  ⇔ D(p) liegt oberhalb D(l).

1.3.6. Satz: Sei S Menge von n Pkten im $\mathbb{R}^2$.
a) CH(S) kann in Zeit O(n log n) berechnet werden (worst case)
b) Falls S lexik. nach xy-Koord. sortiert ist, dann kann CH(S) in Zeit O(n) berechnet werden.

Bew. siehe oben.

1.3.7. Bemerkung:
1) Alg. optimal, da CH-Problem (i.a.) genauso schwierig ist wie sortieren.
Bew. Übung.
2). obere Bzw. untere Hülle sind auch für sich alleine wichtig (für bestimmte Probleme)
3). Optimalität gilt für worst case; es gibt Situationen, in denen Alg. I. besser ist (wenn # Ecken von CH(S) < log n).

1.3.8. Varianten von Graham's Scan:
a) siehe Übung
b). untere u. obere Hülle gleichzeitig berechnen
c). Berechnung der gesamten Hülle in einer Phase.

Idee: 1. Sortiere $S = p_1 .. p_n$.
2. Allgemeiner Schritt:
Bearbeite p_i, $i = 4..n$.
Situation: $C_{i-1} := CH(\{p_1 .. p_{i-1}\})$ ist berechnet.
p_{i-1} ist maximal in $\{p_1 .. p_{i-1}\}$
$\Rightarrow p_{i-1}$ ist die rechteste Ecke von C_{i-1}.
$\Rightarrow \overline{p_{i-1} p_i} \cap C_{i-1} = \{p_{i-1}\}$.

Der eigentliche Schritt:
Berechne Berührpkte t und b der beiden Tangenten von p_i an C_{i-1}.
Genauer: $t \leftarrow p_{i-1}$ d.h. orientation$(p_i, t, succ(t)) \leq 0$
while $(p_i, t, succ(t))$ nicht Rechtsturm do
$t \leftarrow succ(t)$
od
$b \leftarrow p_{i-1}$ d.h. orientation$(p_i, b, pred(b)) \geq 0$
while $(p_i, b, pred(b))$ nicht Rechtsturm do
$b \leftarrow pred(b)$
od

Bea: Die ähnliche Argumentation:
die # unterwegs besuchten Pkte kann in $\Omega(n)$ liegen,
aber jeder von ihnen wird anschließend aus der konvexen Hülle entfernt und kann danach nie wieder als Eckpkt in Erscheinung treten
Also wird er auch nie wieder besucht
$\Rightarrow$ Es kann insgesamt höchstens n Besuche geben
$\Rightarrow$ O(n log n) Laufzeit

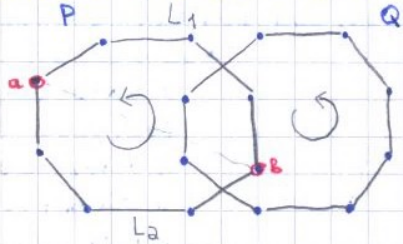
Nachdem wir nun b und t berechnet haben:
- Entferne alle Pkte zwischen b und t
- Füge p_i nach C_i ein.

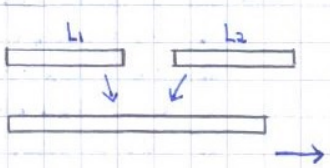
Weitere Implementierungsdetails und Laufzeitanalyse siehe 3. Übung.
Bsp. zu c) siehe Hemdordner.

1.4. Algorithmus III: Devide & Conquer.

Triangulierung schauen

1.4.1. Spezialfall: Konvexe Hülle von zwei konvexen Polygonen P und Q.
$P = p_1 .. p_m$, $Q = q_1 .. q_\ell$, Ecken gg. Uhrzeigersinn sortiert.
Aufgabe: Berechne $CH(P \cup Q)$
Triviale Lsg: Graham's Scan auf die Menge aller Ecken. O(n log n)
Bessere Lsg: Ausnutzen der Polygonstruktur. O(n),
1) Finde Sortierung aller Ecken in Zeit O(n) } O(n)
2) Berechne die Hülle in Zeit O(n) (siehe S.1.3.6.8.). }
zu 1): Betrachte P:
i) Finde extreme Ecken a und B in Zeit O(m)
L_2 := untere Polygonzug
starte bei a und laufe gg. Uhrzeigersinn über P bis B erreicht ist.
L_1 := obere Polygonzug
laufe von b nach a. und drehe um.





(ii) Mische L_1 und L_2 zu einer sortierten Gesamtliste L_P zusammen in Zeit $O(m)$ (→ siehe Mergesort)
Analog: Sortiere Folge L_Q der Ecken von Q in Zeit $O(l)$.
(iii) Mische L_P und L_Q in Zeit $O(m+l) = O(n)$ (wobei $n := m+l$) zu einer sortierten Gesamtfolge zusammen.

→ dann Graham's Scan.

1.4.2. Allgemein: Sei $S \subset \mathbb{R}^2$, $|S| = n$.

```
CONVEX_HULL(S).
if |S| = 1 then
    output S
else
    teile S in zwei möglichst gleich große Teile S1 und S2   } DEVIDE
    (zB. |S1| = ⌈|S|/2⌉ und |S2| = ⌊|S|/2⌋
    P ← CONVEX_HULL(S1)                                      } CONQUER
    Q ← CONVEX_HULL(S2)                                      }
    berechne CH(P ∪ Q) wie in letzter Vorlesung gezeigt.     } MISCHSCHRITT
fi.
```

↑ hier wird die eigentliche Rechnung durchgeführt.

1.4.3. Laufzeit:

Teilen: $O(n)$
Mischen: $O(n)$

Merge auf Listen
Graham's Scan (ohne Sortieren).

$$\Rightarrow T(n) = \begin{cases} c_0, & n=1 \\ c_1 \cdot n + 2 \cdot T(n/2), & n>1 \end{cases}$$

Das ist also dieselbe Rekursion wie für Mergesort (aber mit anderen Konstanten natürlich).
⇒ Gesamtlaufzeit des Verfahrens ist $O(n \log n)$ (siehe Analyse von Mergesort).

1.5. Eine Anwendung von CONVEX HULL.

1.5.1 Problem: Geg. sind n Halbebenen $H_1 \dots H_n$ von $\mathbb{R}^2$
Berechne $P = \bigcap_{i=1}^{n} H_i$

1.5.2 Def: Eine abgeschlossene Halbebene = { alle Pkte auf gleicher Seite einer Geraden h } ∪ h.

1.5.3 Anm: Schnitt P ist konvexes (möglicherweise unbeschränktes) Polygon, da der Schnitt konvexer Mengen wieder konvex ist.

1.5.4 Ziel und Lösungsansatz:

Ziel: Berechne die Folge der Ecken von P gg. den Uhrzeigersinn.
Lösung: Zurückführung auf konvexe Hülle einer geeigneten Pkmenge.
Dazu transformieren wir die definierenden Geraden in "duale" Pkte.

1.5.5 Definition: (Geometr. Transformation)

a) Sei $l = \{ y : \underbrace{y = ax + b}_{\text{Geradengleichung von } l}, x \text{ bel.}, a,b \text{ fest} \}$ eine nicht vertikale Gerade.
Da Punkt $D(l) := (a,b)$ heißt der duale Punkt zu l.

b) Sei $p = (a,b) \in \mathbb{R}^2$ ein Pkt. Die Gerade $D(p) = \{ y : y = -ax + b, x \text{ beliebig} \}$ heißt die duale Gerade zu p.

Abbildung D erlaubt die relative Lage von Objekten.

1.5.6 Lemma: Pkt p liegt auf (oberhalb / unterhalb) einer Geraden l
⇔ Gerade D(p) liegt auf (oberhalb/ unterhalb) Pkt D(l).

Beweis: Sei $p = (p_x, p_y)$, $l = \{ y \in \mathbb{R} : y = ax + b, x \in \mathbb{R} \}$ (a,b fest).
$\Rightarrow D(l) = (a,b)$ und $D(p) = \{ y : y = -p_x x + p_y, x \in \mathbb{R} \}$

- sei p gelegen auf l
  ⇔ $p_y = a \cdot p_x + b$
  ⇔ $-b = a p_x - p_y$
  ⇔ $b = -p_x a + p_y$
  ⇔ D(p) liegt auf D(l).

- sei p gelegen oberhalb l (unterhalb analog).
  ⇔ $ax + b < p_y \ \forall x \in \mathbb{R}$
  Bzw. sogar $a p_x + b < p_y$.
  ⇔ $-a p_x - b > -p_y$
  ⇔ $-a p_x + p_y > b$
  ⇔ D(p) liegt oberhalb D(l).

1.5.7 Folgerung: Wenn $p = \ell_1 \cap \ell_2 \Rightarrow D(\ell_1), D(\ell_2) \in D(p)$.
Bew. siehe Vorordner. eigentlich "$\Leftrightarrow$"

1.5.8. Betrachte folgendes Problem:
Seien $\ell_1 .. \ell_n$ n nicht vertikale Geraden im $\mathbb{R}^2$.
$\ell_i^+ :=$ Halbraum oberhalb von ℓ_i $(i \in \{1..n\})$
$\ell_i^- :=$ Halbraum unterhalb von ℓ_i $(i \in \{1..n\})$.
Sei weiter $m \leq n$.
Nun berechne: $S := \ell_1^+ \cap \dots \cap \ell_m^+ \cap \ell_{m+1}^- \cap \dots \cap \ell_n^-$.

⟶ 1.5.8.1 Beobachtung: Unser ursprüngliches Problem kann in dieser Weise formuliert werden.

Sei $S^+ := \bigcap_{i=1}^{m} \ell_i^+$ und $S^- := \bigcap_{j=m+1}^{n} \ell_j^-$

$\Rightarrow S = S^+ \cap S^-$
Wir berechnen S^+ und S^- getrennt.
Zunächst S^+ (S^- analog):
S^+ ist ein nach oben unbeschränktes konvexes Polygon.

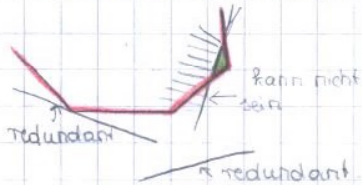
⟶ 1.5.8.2 Def: Eine Gerade ℓ_i, $1 \leq i \leq m$ heißt redundant, falls sie nicht zum Rand von S^+ beiträgt, d.h. es gibt keine Kante von S^+ auf ℓ_i.

Beobachtung: Redundante Geraden können ignoriert werden.
Frage: Wie findet man sie? → Dualität.

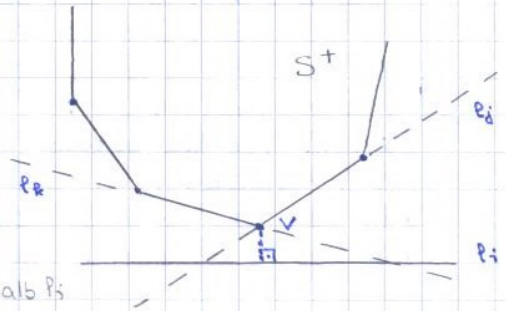
⟶ 1.5.8.3 Lemma: ℓ_i ist redundant $(i \in \{1..m\})$
$\Leftrightarrow$ Pkt $D(\ell_i)$ ist keine Ecke der oberen konvexen Hülle von $\{D(\ell_1) .. D(\ell_m)\}$.

Beweis:
Vorbemerkung:
Sei ℓ_i redundant
$\Rightarrow S^+ \cap \ell_i = \emptyset$ oder Ecke von S^+
Sei v die Ecke von S^+, die ℓ_i am nächsten liegt.

Bea: Zwei Ecken kann es im Schnitt nicht geben, denn:



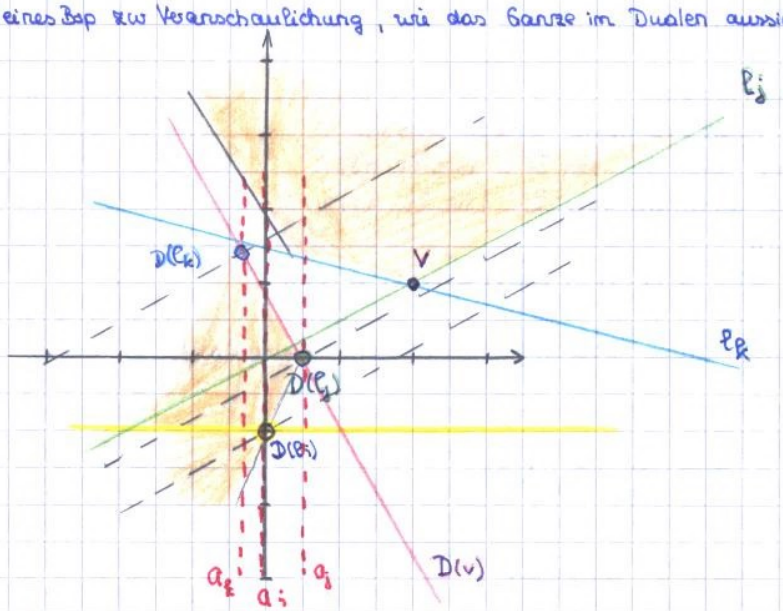
Beobachtung:
1). $\exists$ zwei nicht redundante Geraden ℓ_j und ℓ_k mit $v = \ell_j \cap \ell_k$.
Skizze:



2). Steigung von ℓ_i liegt zw. Steigungen von ℓ_j und ℓ_k, da sonst entweder ℓ_i nicht redundant oder eine andere Ecke als v näher zu ℓ_i liegt.
3). Pkt $D(\ell_i)$ liegt auf oder unterhalb der Geraden $D(v)$
Bea: v ist auf oder oberhalb ℓ_i
$\Leftrightarrow D(v)$ auf oder oberhalb $D(\ell_i)$
$\Leftrightarrow D(\ell_i)$ auf oder unterhalb $D(\ell_i)$
4). $D(\ell_j)$ und $D(\ell_k)$ liegen auf der Geraden $D(v)$
Bea: v liegt auf ℓ_j und ℓ_k
$\Leftrightarrow D(v)$ liegt auf $D(\ell_j)$ und $D(\ell_k)$

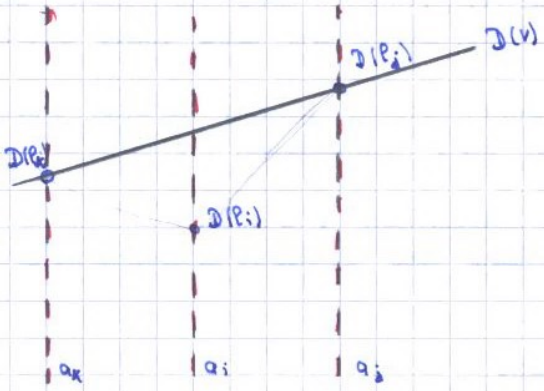
→

Kleines Bsp zur Veranschaulichung, wie das Ganze im Dualen aussieht:



$p_j = \{y : y = \frac{1}{2}x\}$
$\Rightarrow D(p_j) = (\frac{1}{2}, 0)$
$p_k = \{y : y = -\frac{1}{4}x + \frac{3}{2}\}$
$\Rightarrow D(p_k) = (-\frac{1}{4}, \frac{3}{2})$.
$v = (2,1)$
$\Rightarrow D(v) = \{y : y = -2x + 1\}$
$p_i = \{y : y = -1\}$
$\Rightarrow D(p_i) = (0, -1)$

Dh. also: Situation im Dualen:



Der eigentliche Beweis:

„⇒" sei p_i redundant

z.z. $D(p_i)$ ist keine Ecke der oberen $CH(\{D(p_1) .. D(p_m)\})$

Setze: $D(p_i) := (a_i, b_i)$
$D(p_j) := (a_j, b_j)$
$D(p_k) := (a_k, b_k)$

dh. a_i, a_j und a_k sind Steigungen von p_i, p_j und p_k

Beob. 2) $\Rightarrow a_k \leq a_i \leq a_j$ $\rightarrow D(p_j)$ ist weiter rechts als $D(p_i) \Rightarrow D(p_i)$ kann nicht mehr die rechteste Ecke der obs. Hülle sein!

Beob. 3) $\Rightarrow$ **$D(p_i)$ liegt auf oder unterhalb $D(v)$**

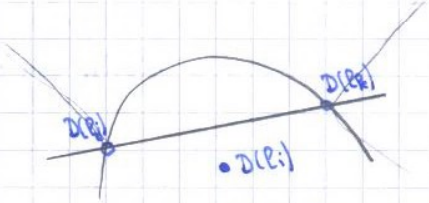
- $D(p_k)$ ist weiter links $\Rightarrow D(p_i)$ nicht die linkeste Ecke
- $D(p_i)$ auf oder unterhalb $D(v) \Rightarrow$ **$D(p_i)$ ist nicht Ecke der oberen Hülle.**

„⇐" (rückwärts lesen von „⇒")

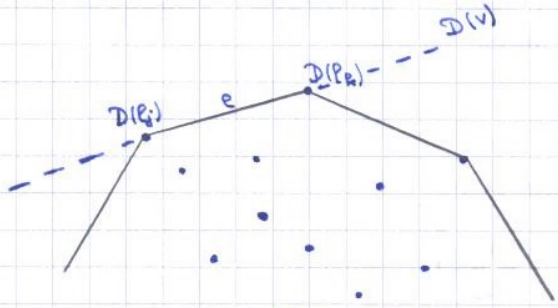
Sei $D(p_i)$ nicht Ecke der oberen Hülle.

z.z. p_i redundant

$D(p_i)$ nicht Ecke der oberen Hülle $\Rightarrow \exists$ $D(p_j)$ und $D(p_k)$ so, dass $D(p_i)$ auf oder unterhalb der Geraden durch $D(p_j)$ und $D(p_k)$ liegt.



⟶ 1.5.8.4. Lemma: $v \in \mathbb{R}^2$ ist Ecke von S^+
⇔ Die Gerade $D(v)$ enthält eine Kante der oberen Hülle von $\{D(l_1) \dots D(l_m)\}$.



Beweis:

„⇐": $D(v)$ enthält eine Kante e der oberen Hülle von $\{D(l_1) \dots D(l_m)\}$.
z.z. v ist Ecke von S^+
Sei $e = (D(l_j), D(l_k))$
Dualität: $v \in l_j \cap l_k$
obere Hülle ⇒ alle Pkte $D(l_i)$, $i \in \{1..m\}$ liegen unterhalb oder auf $D(v)$
Dualität ⇒ Gerade l_i $(1 \le i \le m)$ liegt unterhalb oder auf v
d.h. v liegt über oder auf l_i.
⇒ $v \in S^+$
v = Schnittpkt zweier Geraden ⇒ v ist Ecke von S^+
(Bea: l_j und l_k sind nicht redundant
denn wenn sie es wären ⇒ $D(l_j)$ und $D(l_k)$ wären nicht Ecken von $CH(D(l_1) \dots D(l_m))$ nach 1.5.8.3 ⇒ ↯)

„⇒" v ist Ecke von S^+
⇒ $v = l_j \cap l_k$
1.5.7 ⇒ $D(l_j), D(l_k) \in D(v)$
⇒ $D(v)$ enthält also die Kante $e := (D(l_j), D(l_k))$ } (*)
bleibt z.z.: diese Kante ist die Kante der konvexen Hülle.
Es gilt: $v \in S^+$
⇒ v liegt über oder auf l_i $\forall\, i \in \{1..m\}$
⇒ $D(l_i)$ liegt unter oder auf $D(v)$ $\forall\, i \in \{1..m\}$
⇒ $D(v)$ enthält eine Kante der oberen Hülle oder $D(v)$ ist redundant
Wegen (*) kann $D(v)$ nicht redundant sein
⇒ $D(v)$ enthält eine Kante der oberen Hülle.

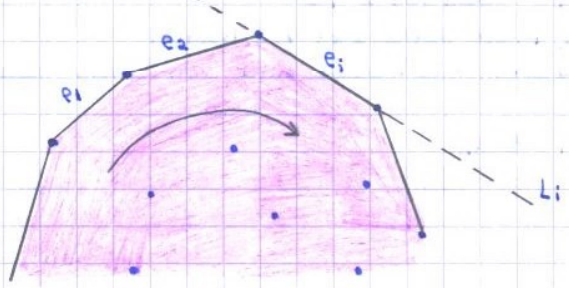
Dieses Lemma liefert einen Algorithmus zur Berechnung von $S^+ = \bigcap_{i=1}^{m} l_i^+$

⟶ 1.5.8.5. Algorithmus:

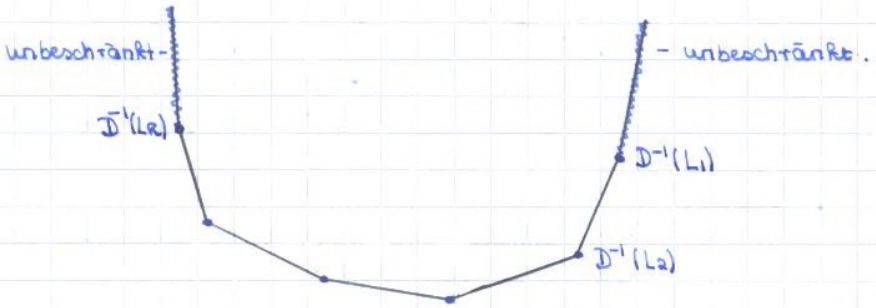
1. Berechne die dualen Pkte $p_i = D(l_i)$, $i = 1..m$ $O(m)$
2. Berechne die obere konvexe Hülle H von $\{p_1 .. p_m\}$ $O(m \log m)$
   Seien $e_1 .. e_k$ die Kanten von H von links nach rechts
3. Berechne die Folge der Geraden $L_1 .. L_k$ so, dass $e_i \subset L_i$ $O(m)$
4. Ausgabe:
   $D^{-1}(L_1) \dots D^{-1}(L_k)$ (= Eckenfolge von S^+) $O(m)$

Achtung: $D^{-1} \neq D$.

$L_i: y = ax + b$
$\Rightarrow D^{-1}(L_i) = (-a, b)$



. $e_1 \dots e_R$ Kanten von H von links nach rechts.
⇒ $L_1 \dots L_R$ nach Steigungen absteigend sortiert. (siehe Skizze ⇒ klar!)
⇒ Ecken von S^+ sind nach x-Koordinaten absteigend sortiert, d.h. von rechts nach links.



Frage: Wie finden wir die beiden unbeschränkten Kanten?
Antwort: Linke Gerade mit minimaler Steigung.
Rechte Gerade mit maximaler Steigung.

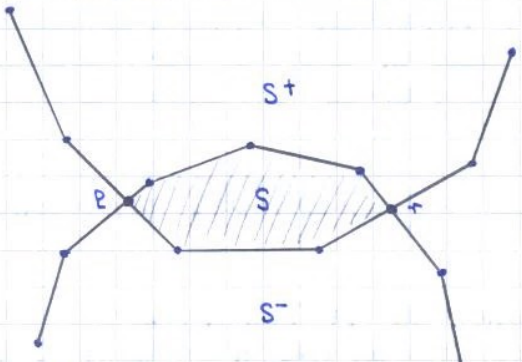
⟶ 1.5.8.6. Zwischenresultat:
$S^+ = \bigcap_{i=1}^{m} l_i^+$ kann in Zeit $O(m \log m)$ berechnet werden.

$\underbrace{O(m)}$ + $\underbrace{O(m \log m)}$
$D, D^{-1} \dots$ Graham's Scan.

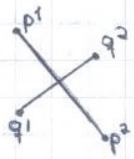
Falls $l_1 \dots l_m$ nach Steigung sortiert gegeben sind, dann Laufzeit $O(m)$ (↝ Graham's Scan)

$S^- = \bigcap_{i=m+1}^{n} l_i^-$ kann genauso (symmetrisch) berechnet werden.

Es bleibt noch $S = S^+ \cap S^-$ auszurechnen.
⟶ 1.5.8.7. Berechne S.

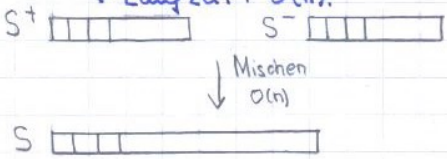


Finde Schnittpkte l und r des Ränder von S^+ und S^-
Achtung: Sonderfälle:
- ex. nicht ($S = \emptyset$)
- $l = r$ ($S = \{l\} = \{r\}$)
- l, r Ecken von S^+, S^-



Allgemein: Innere Schnittpkte von Kanten
Voraussetzung: Die Ränder sind von links nach rechts sortiert.

⇒ Laufzeit: $O(n)$.



und gleichzeitig jeweils das l_i, welches eingefügt wird, in int S^+ und S^- schreiben und vergleichen.
Sobald Änderung ⇒ Kante gefunden. ⇒ Schnittpkt berechnen
⇒ Gesamtlaufzeit: $O(n)$.

(ii) Mische L_1 und L_2 zu einer sortierten Gesamtliste L_P zusammen in Zeit $O(m)$ (→ siehe Mergesort)
Analog: Sortiere Folge L_Q der Ecken von Q in Zeit $O(l)$.
(iii) Mische L_P und L_Q in Zeit $O(m+l) = O(n)$ (wobei $n := m+l$) zu einer sortierten Gesamtfolge zusammen.

→ dann Graham's Scan.

1.4.2. Allgemein: Sei $S \subset \mathbb{R}^2$, $|S| = n$.

```
CONVEX_HULL(S).
if |S| = 1 then
    output S
else
    teile S in zwei möglichst gleich große Teile S1 und S2   } DEVIDE
    (zB. |S1| = ⌈|S|/2⌉ und |S2| = ⌊|S|/2⌋
    P ← CONVEX_HULL(S1)                                      } CONQUER
    Q ← CONVEX_HULL(S2)                                      }
    berechne CH(P ∪ Q) wie in letzter Vorlesung gezeigt.     } MISCHSCHRITT
fi.
```

↑ hier wird die eigentliche Rechnung durchgeführt.

1.4.3. Laufzeit:

Teilen: $O(n)$
Mischen: $O(n)$

Merge auf Listen
Graham's Scan (ohne Sortieren).

$$\Rightarrow T(n) = \begin{cases} c_0, & n=1 \\ c_1 \cdot n + 2 \cdot T(n/2), & n>1 \end{cases}$$

Das ist also dieselbe Rekursion wie für Mergesort (aber mit anderen Konstanten natürlich).
⇒ Gesamtlaufzeit des Verfahrens ist $O(n \log n)$ (siehe Analyse von Mergesort).

1.5. Eine Anwendung von CONVEX HULL.

1.5.1 Problem: Geg. sind n Halbebenen $H_1 \dots H_n$ von $\mathbb{R}^2$
Berechne $P = \bigcap_{i=1}^{n} H_i$

1.5.2 Def: Eine abgeschlossene Halbebene = { alle Pkte auf gleicher Seite einer Geraden h } ∪ h.

1.5.3 Anm: Schnitt P ist konvexes (möglicherweise unbeschränktes) Polygon, da der Schnitt konvexer Mengen wieder konvex ist.

1.5.4 Ziel und Lösungsansatz:

Ziel: Berechne die Folge der Ecken von P gg. den Uhrzeigersinn.
Lösung: Zurückführung auf konvexe Hülle einer geeigneten Pkmenge.
Dazu transformieren wir die definierenden Geraden in "duale" Pkte.

1.5.5 Definition: (Geometr. Transformation)

a) Sei $l = \{ y : \underbrace{y = ax + b}_{\text{Geradengleichung von } l}, x \text{ bel.}, a,b \text{ fest} \}$ eine nicht vertikale Gerade.
Da Punkt $D(l) := (a,b)$ heißt der duale Punkt zu l.

b) Sei $p = (a,b) \in \mathbb{R}^2$ ein Pkt. Die Gerade $D(p) = \{ y : y = -ax + b, x \text{ beliebig} \}$ heißt die duale Gerade zu p.

Abbildung D erlaubt die relative Lage von Objekten.

1.5.6 Lemma: Pkt p liegt auf (oberhalb / unterhalb) einer Geraden l
⇔ Gerade D(p) liegt auf (oberhalb/ unterhalb) Pkt D(l).

Beweis: Sei $p = (p_x, p_y)$, $l = \{ y \in \mathbb{R} : y = ax + b, x \in \mathbb{R} \}$ (a,b fest).
$\Rightarrow D(l) = (a,b)$ und $D(p) = \{ y : y = -p_x x + p_y, x \in \mathbb{R} \}$

- sei p gelegen auf l
  ⇔ $p_y = a \cdot p_x + b$
  ⇔ $-b = a p_x - p_y$
  ⇔ $b = -p_x a + p_y$
  ⇔ D(p) liegt auf D(l).

- sei p gelegen oberhalb l (unterhalb analog).
  ⇔ $ax + b < p_y \ \forall x \in \mathbb{R}$
  Bzw. sogar $a p_x + b < p_y$.
  ⇔ $-a p_x - b > -p_y$
  ⇔ $-a p_x + p_y > b$
  ⇔ D(p) liegt oberhalb D(l).

1.5.7 Folgerung: Wenn $p = \ell_1 \cap \ell_2 \Rightarrow D(\ell_1), D(\ell_2) \in D(p)$.
Bew. siehe Vorordner. eigentlich "$\Leftrightarrow$"

1.5.8. Betrachte folgendes Problem:
Seien $\ell_1 .. \ell_n$ n nicht vertikale Geraden im $\mathbb{R}^2$.
$\ell_i^+ :=$ Halbraum oberhalb von ℓ_i $(i \in \{1..n\})$
$\ell_i^- :=$ Halbraum unterhalb von ℓ_i $(i \in \{1..n\})$.
Sei weiter $m \leq n$.
Nun berechne: $S := \ell_1^+ \cap \dots \cap \ell_m^+ \cap \ell_{m+1}^- \cap \dots \cap \ell_n^-$.

⟶ 1.5.8.1 Beobachtung: Unser ursprüngliches Problem kann in dieser Weise formuliert werden.

Sei $S^+ := \bigcap_{i=1}^{m} \ell_i^+$ und $S^- := \bigcap_{j=m+1}^{n} \ell_j^-$

$\Rightarrow S = S^+ \cap S^-$
Wir berechnen S^+ und S^- getrennt.
Zunächst S^+ (S^- analog):
S^+ ist ein nach oben unbeschränktes konvexes Polygon.

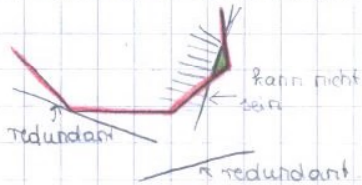
⟶ 1.5.8.2 Def: Eine Gerade ℓ_i, $1 \leq i \leq m$ heißt redundant, falls sie nicht zum Rand von S^+ beiträgt, d.h. es gibt keine Kante von S^+ auf ℓ_i.

Beobachtung: Redundante Geraden können ignoriert werden.
Frage: Wie findet man sie? → Dualität.

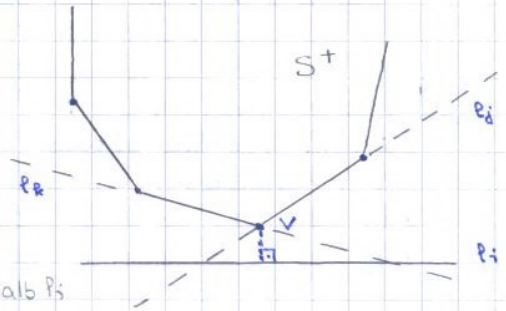
⟶ 1.5.8.3 Lemma: ℓ_i ist redundant $(i \in \{1..m\})$
$\Leftrightarrow$ Pkt $D(\ell_i)$ ist keine Ecke der oberen konvexen Hülle von $\{D(\ell_1) .. D(\ell_m)\}$.

Beweis:
Vorbemerkung:
Sei ℓ_i redundant
$\Rightarrow S^+ \cap \ell_i = \emptyset$ oder Ecke von S^+
Sei v die Ecke von S^+, die ℓ_i am nächsten liegt.

Bea: Zwei Ecken kann es im Schnitt nicht geben, denn:



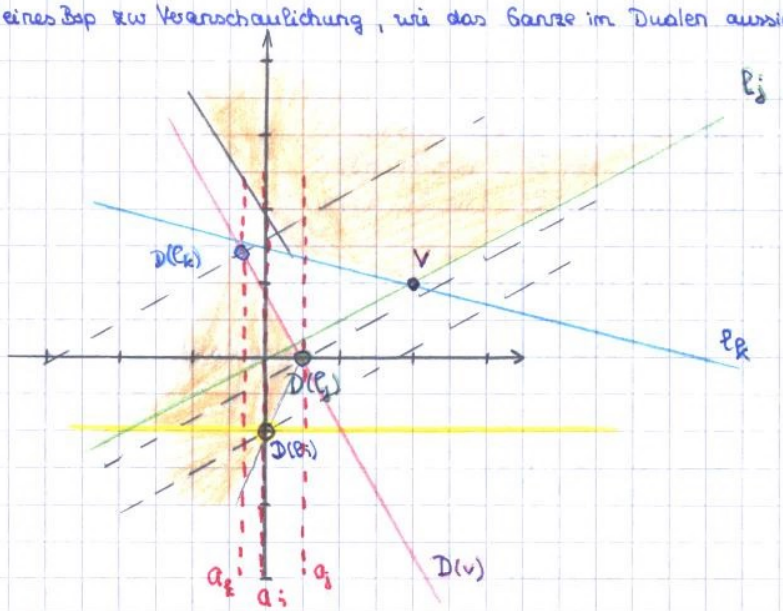
Beobachtung:
1). $\exists$ zwei nicht redundante Geraden ℓ_j und ℓ_k mit $v = \ell_j \cap \ell_k$.
Skizze:



2). Steigung von ℓ_i liegt zw. Steigungen von ℓ_j und ℓ_k, da sonst entweder ℓ_i nicht redundant oder eine andere Ecke als v näher zu ℓ_i liegt.
3). Pkt $D(\ell_i)$ liegt auf oder unterhalb der Geraden $D(v)$
Bea: v ist auf oder oberhalb ℓ_i
$\Leftrightarrow D(v)$ auf oder oberhalb $D(\ell_i)$
$\Leftrightarrow D(\ell_i)$ auf oder unterhalb $D(\ell_i)$
4). $D(\ell_j)$ und $D(\ell_k)$ liegen auf der Geraden $D(v)$
Bea: v liegt auf ℓ_j und ℓ_k
$\Leftrightarrow D(v)$ liegt auf $D(\ell_j)$ und $D(\ell_k)$

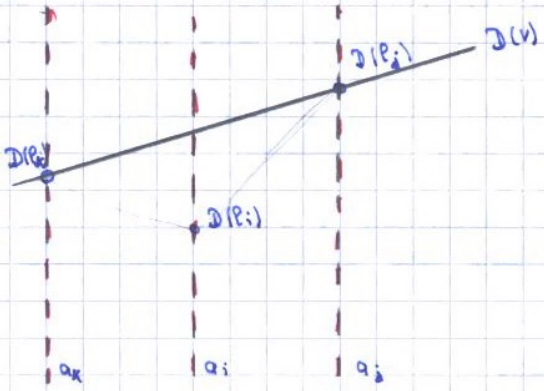
→

Kleines Bsp zur Veranschaulichung, wie das Ganze im Dualen aussieht:



$p_j = \{y : y = \frac{1}{2}x\}$
$\Rightarrow D(p_j) = (\frac{1}{2}, 0)$
$p_k = \{y : y = -\frac{1}{4}x + \frac{3}{2}\}$
$\Rightarrow D(p_k) = (-\frac{1}{4}, \frac{3}{2})$.
$v = (2,1)$
$\Rightarrow D(v) = \{y : y = -2x + 1\}$
$p_i = \{y : y = -1\}$
$\Rightarrow D(p_i) = (0, -1)$

Dh. also: Situation im Dualen:



Der eigentliche Beweis:

„⇒" sei p_i redundant

z.z. $D(p_i)$ ist keine Ecke der oberen $CH(\{D(p_1) .. D(p_m)\})$

Setze: $D(p_i) := (a_i, b_i)$
$D(p_j) := (a_j, b_j)$
$D(p_k) := (a_k, b_k)$

dh. a_i, a_j und a_k sind Steigungen von p_i, p_j und p_k

Beob. 2) $\Rightarrow a_k \leq a_i \leq a_j$ $\rightarrow D(p_j)$ ist weiter rechts als $D(p_i) \Rightarrow D(p_i)$ kann nicht mehr die rechteste Ecke der obs. Hülle sein!

Beob. 3) $\Rightarrow$ **$D(p_i)$ liegt auf oder unterhalb $D(v)$**

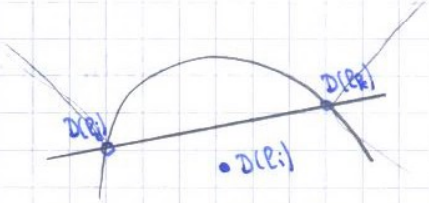
- $D(p_k)$ ist weiter links $\Rightarrow D(p_i)$ nicht die linkeste Ecke
- $D(p_i)$ auf oder unterhalb $D(v) \Rightarrow$ **$D(p_i)$ ist nicht Ecke der oberen Hülle.**

„⇐" (rückwärts lesen von „⇒")

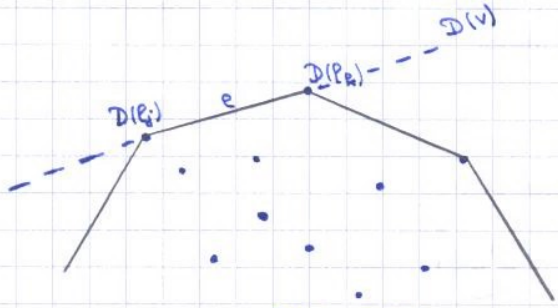
Sei $D(p_i)$ nicht Ecke der oberen Hülle.

z.z. p_i redundant

$D(p_i)$ nicht Ecke der oberen Hülle $\Rightarrow \exists$ $D(p_j)$ und $D(p_k)$ so, dass $D(p_i)$ auf oder unterhalb der Geraden durch $D(p_j)$ und $D(p_k)$ liegt.



⟶ 1.5.8.4. Lemma: $v \in \mathbb{R}^2$ ist Ecke von S^+
⇔ Die Gerade $D(v)$ enthält eine Kante der oberen Hülle von $\{D(l_1) \dots D(l_m)\}$.



Beweis:

„⇐": $D(v)$ enthält eine Kante e der oberen Hülle von $\{D(l_1) \dots D(l_m)\}$.
z.z. v ist Ecke von S^+
Sei $e = (D(l_j), D(l_k))$
Dualität: $v \in l_j \cap l_k$
obere Hülle ⇒ alle Pkte $D(l_i)$, $i \in \{1..m\}$ liegen unterhalb oder auf $D(v)$
Dualität ⇒ Gerade l_i $(1 \le i \le m)$ liegt unterhalb oder auf v
d.h. v liegt über oder auf l_i.
⇒ $v \in S^+$
v = Schnittpkt zweier Geraden ⇒ v ist Ecke von S^+
(Bea: l_j und l_k sind nicht redundant
denn wenn sie es wären ⇒ $D(l_j)$ und $D(l_k)$ wären nicht Ecken von $CH(D(l_1) \dots D(l_m))$ nach 1.5.8.3 ⇒ ↯)

„⇒" v ist Ecke von S^+
⇒ $v = l_j \cap l_k$
1.5.7 ⇒ $D(l_j), D(l_k) \in D(v)$
⇒ $D(v)$ enthält also die Kante $e := (D(l_j), D(l_k))$ } (*)
bleibt z.z.: diese Kante ist die Kante der konvexen Hülle.
Es gilt: $v \in S^+$
⇒ v liegt über oder auf l_i $\forall\, i \in \{1..m\}$
⇒ $D(l_i)$ liegt unter oder auf $D(v)$ $\forall\, i \in \{1..m\}$
⇒ $D(v)$ enthält eine Kante der oberen Hülle oder $D(v)$ ist redundant
Wegen (*) kann $D(v)$ nicht redundant sein
⇒ $D(v)$ enthält eine Kante der oberen Hülle.

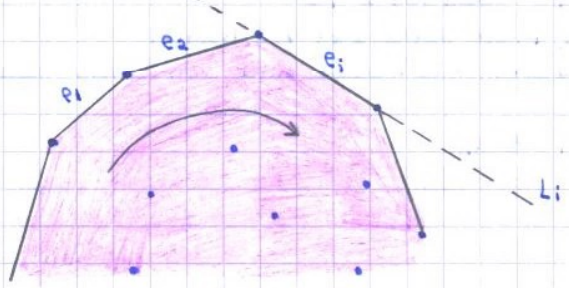
Dieses Lemma liefert einen Algorithmus zur Berechnung von $S^+ = \bigcap_{i=1}^{m} l_i^+$

⟶ 1.5.8.5. Algorithmus:

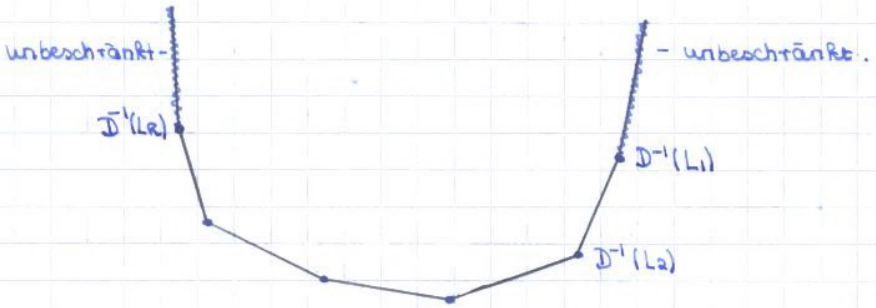
1. Berechne die dualen Pkte $p_i = D(l_i)$, $i = 1..m$ $O(m)$
2. Berechne die obere konvexe Hülle H von $\{p_1 .. p_m\}$ $O(m \log m)$
   Seien $e_1 .. e_k$ die Kanten von H von links nach rechts
3. Berechne die Folge der Geraden $L_1 .. L_k$ so, dass $e_i \subset L_i$ $O(m)$
4. Ausgabe:
   $D^{-1}(L_1) \dots D^{-1}(L_k)$ (= Eckenfolge von S^+) $O(m)$

Achtung: $D^{-1} \neq D$.

$L_i: y = ax + b$
$\Rightarrow D^{-1}(L_i) = (-a, b)$



. $e_1 \dots e_R$ Kanten von H von links nach rechts.
⇒ $L_1 \dots L_R$ nach Steigungen absteigend sortiert. (siehe Skizze ⇒ klar!)
⇒ Ecken von S^+ sind nach x-Koordinaten absteigend sortiert, d.h. von rechts nach links.



Frage: Wie finden wir die beiden unbeschränkten Kanten?
Antwort: Linke Gerade mit minimaler Steigung.
Rechte Gerade mit maximaler Steigung.

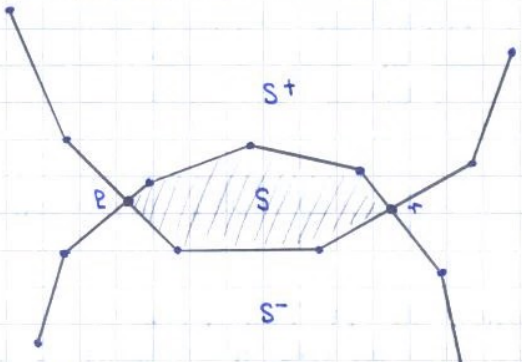
⟶ 1.5.8.6. Zwischenresultat:
$S^+ = \bigcap_{i=1}^{m} l_i^+$ kann in Zeit $O(m \log m)$ berechnet werden.

$\underbrace{O(m)}$ + $\underbrace{O(m \log m)}$
$D, D^{-1} \dots$ Graham's Scan.

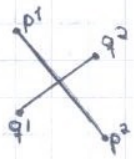
Falls $l_1 \dots l_m$ nach Steigung sortiert gegeben sind, dann Laufzeit $O(m)$ (↝ Graham's Scan)

$S^- = \bigcap_{i=m+1}^{n} l_i^-$ kann genauso (symmetrisch) berechnet werden.

Es bleibt noch $S = S^+ \cap S^-$ auszurechnen.
⟶ 1.5.8.7. Berechne S.

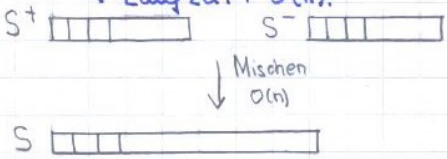


Finde Schnittpkte l und r des Ränder von S^+ und S^-
Achtung: Sonderfälle:
- ex. nicht ($S = \emptyset$)
- $l = r$ ($S = \{l\} = \{r\}$)
- l, r Ecken von S^+, S^-



Allgemein: Innere Schnittpkte von Kanten
Voraussetzung: Die Ränder sind von links nach rechts sortiert.

⇒ Laufzeit: $O(n)$.



und gleichzeitig jeweils das l_i, welches eingefügt wird, in int S^+ und S^- schreiben und vergleichen.
Sobald Änderung ⇒ Kante gefunden. ⇒ Schnittpkt berechnen
⇒ Gesamtlaufzeit: $O(n)$.

1.5.9. Satz (Zusammenfassung)

1). Der Schnitt von n Halbebenen kann in Zeit $O(n \log n)$ berechnet werden.

2). Falls die definierenden Geraden nach Steigung sortiert gegeben sind, dann ist die Laufzeit $O(n)$.

1.5.10 Bemerkungen:

1) Die geometrische Transformation der Dualität wird auch noch für andere Probleme dieser Vorlesung wichtig sein.

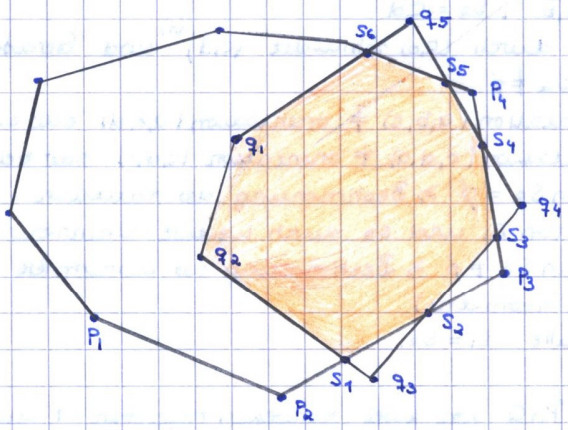
2) Der Schnitt von n Halbebenen kann auch direkt berechnet werden.
   ⇝ Divide & Conquer.

1.5.11. Schnitt von Halbebenen durch Divide and Conquer

Schnitt von zwei konvexen Polygonen.
Hier abgeschlossen (offen → Übung, voriger Abschnitt)
Geg: Seien $P = (p_1 .. p_m)$, $Q = (q_1 .. q_e)$ konvexe abgeschlossene Polygone, geg. durch Eckenfolge gg. den Uhrzeigersinn.
Ausgabe: Eckenfolge von $P \cap Q$ gegen Uhrzeigersinn.

↳ **1.5.11.1. Beispiel.**

Mögliche Ausgabe:
$q_1 q_2 s_1 s_2 s_3 s_4 s_5 s_6$

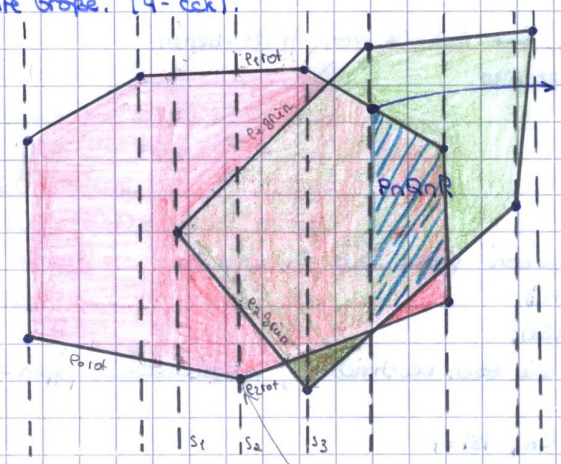


↳ **1.5.11.2. Idee:**

Zerlege Ebene in Regionen so, dass für jede Region R $(P \cap Q) \cap R$ einfach zu berechnen ist.

↳ **1.5.11.3 Regionen:** Vertikale Streifen.

Zeichne durch jede Ecke von P und Q eine senkrechte Gerade.
Jeweils zwei benachbarte Senkrechten definieren eine Region, nämlich den vertikalen Streifen dazwischen.
Dann ist für jeden Streifen R $R \cap P$ und $R \cap Q$ ein Trapezoid, d.h. haben konstante Größe. (4-Eck).



Es gilt $P \cap Q \cap R = (P \cap R) \cap (Q \cap R)$
wegen konst. Größe braucht Zeit $O(1)$

1.5.11.4. Algorithmus:

1). Berechne Streifen von links nach rechts sortiert in Zeit $O(n)$, wobei $n = m+p$. Gesamtzahl der Ecken, durch Mischen der Eckenfolgen von P und Q. (siehe D&C für CH.)

2). Berechne für jeden Streifen R Trapezoide $P \cap R$ und $Q \cap R$. Das geht auch in $O(n)$, da wir Eckenfolgen (siehe 1) von links nach rechts sortiert haben.

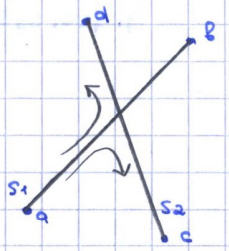
3) Berechne für jeden Streifen R $P \cap Q \cap R := (P \cap R) \cap (Q \cap R)$
Zeit $O(1)$ pro Streifen.

4). Zusammen kleben der Teile aus 3)
Insbesondere Elimination von Ecken, die nicht zur Ausgabe gehören.
Laufzeit: $O(n)$ (Durchlaufen der Senkrechten von links nach rechts).
→ Eckenfolge von $P \cap Q$ gg. Uhrzeigersinn.

1.5.11.5. Implementierungsdetails:

Teilprobleme:

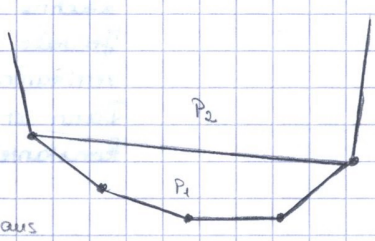
- Test ob Segmente s_1 ein anderes schneidet.
  $s_1 = \overline{a,b}$, $s_2 = \overline{c,d}$
  Gerade durch (a,b) schneidet (c,d) (1) und Gerade durch (c,d) schneidet (a,b) (2)
  $\Leftrightarrow s_1 \cap s_2 \neq \emptyset$
  (1) orientation(a,b,c) $\neq$ orientation(a,b,d) oder beide 0
  (2) orientation(c,d,a) $\neq$ orientation(c,d,b) oder beide 0.
- falls $s_1 \cap s_2 = \emptyset$ ⇒ Bestimmung der relativen Lage von s_1 und s_2 durch weitere Orientierungstests.
- falls $s_1 \cap s_2 \neq \emptyset$ ⇒ Bestimmung des Schnittpkte (anal. Geometrie)
- Sonderfälle: $s_1 = s_2$.



1.5.12. Satz: Der Schnitt $P \cap Q$ von zwei konvexen Polygonen P und Q kann in Zeit $O(n)$ berechnet werden, wobei n = Summe der Ecken von P und Q.
Mischschritt

1.5.13. Fragen:

1) Geht das schneller, wenn alle Polygone im Voraus geg. sind, für die man evtl. Schnittoperationen ausführt.
2). Existiert Algorithmus, der output-sensitiv ist?
Dh. Laufzeit hängt von Größe der Ausgabe ab.
→ siehe nächster Abschnitt über konvexe Polygone.



1.5.14. Divide & Conquer - Alg. für Schnitt von Halbebenen.

Sei S Menge von Halbebenen.

```
INTERSECT(S)
if |S| = 1 then
    return S;
else
    teile S in zwei gleich große Teile S1 und S2
    P ← INTERSECT(S1);
    Q ← INTERSECT(S2);
    return P ∩ Q   // wie oben beschrieben, möglicherweise offen → Übung.
```

Ein unbeschr. Polygon besteht aus

- einem beschr. Pol. P_1 und unbeschr. Pol. P_2 ⇒ $P = P_1 \cup P_2$
  Berechne $P_2 \cap Q_2$ mit D&C und Mischritt von oben ⇒ $O(n \log n)$
  Berechne $P_2 \cap Q_1$ und $P_1 \cap Q_2$ als Schnitt von Polygon mit drei Geraden in Zeit $3 \cdot O(n \log n) = O(n \log n)$. Anschließend berechne $P_2 \cap Q_2$ als Schnitt von 6 Geraden in $O(9) = O(1)$
  ⇒ Gesamtlaufzeit: $O(n \log n)$.

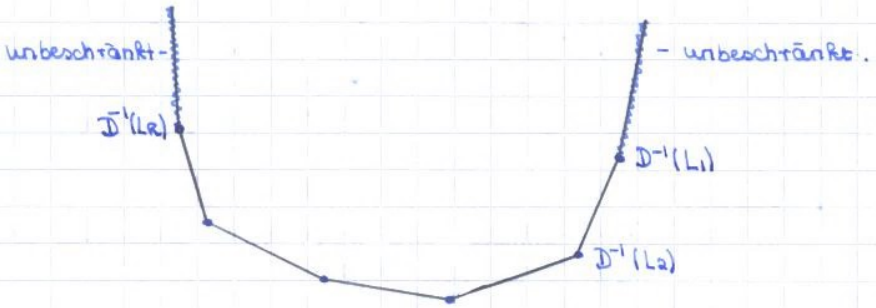
1.5.15 Laufzeit:

$$T(n) = \begin{cases} c_0, & |S| = 1 \\ c_1 n + 2 \cdot T(n/2), & |S| > 1 \end{cases}$$

⇒ $T(n) = O(n \log n)$
Alternative zu Dualitätsalgorithmus

1.5.16. Frage: Welcher Algorithmus ist in welchen Fällen schneller? Wahrscheinlich Divide & Conquer

. $e_1 \dots e_R$ Kanten von H von links nach rechts.
⇒ $L_1 \dots L_R$ nach Steigungen absteigend sortiert. (siehe Skizze ⇒ klar!)
⇒ Ecken von S^+ sind nach x-Koordinaten absteigend sortiert, d.h. von rechts nach links.



Frage: Wie finden wir die beiden unbeschränkten Kanten?
Antwort: Linke Gerade mit minimaler Steigung.
Rechte Gerade mit maximaler Steigung.

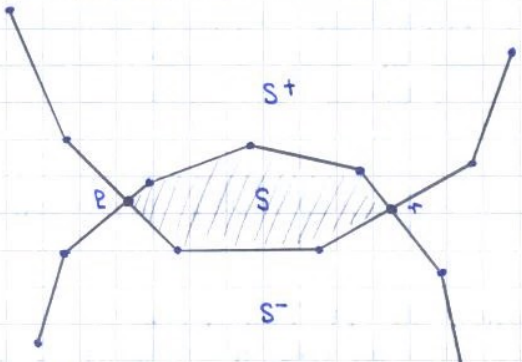
⟶ 1.5.8.6. Zwischenresultat:
$S^+ = \bigcap_{i=1}^{m} l_i^+$ kann in Zeit $O(m \log m)$ berechnet werden.

$\underbrace{O(m)}$ + $\underbrace{O(m \log m)}$
$D, D^{-1} \dots$ Graham's Scan.

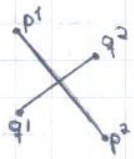
Falls $l_1 \dots l_m$ nach Steigung sortiert gegeben sind, dann Laufzeit $O(m)$ (↝ Graham's Scan)

$S^- = \bigcap_{i=m+1}^{n} l_i^-$ kann genauso (symmetrisch) berechnet werden.

Es bleibt noch $S = S^+ \cap S^-$ auszurechnen.
⟶ 1.5.8.7. Berechne S.

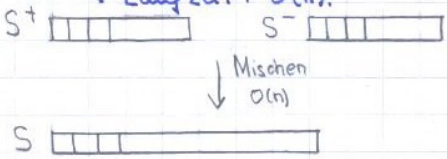


Finde Schnittpkte l und r des Ränder von S^+ und S^-
Achtung: Sonderfälle:
- ex. nicht ($S = \emptyset$)
- $l = r$ ($S = \{l\} = \{r\}$)
- l, r Ecken von S^+, S^-



Allgemein: Innere Schnittpkte von Kanten
Voraussetzung: Die Ränder sind von links nach rechts sortiert.

⇒ Laufzeit: $O(n)$.



und gleichzeitig jeweils das l_i, welches eingefügt wird, in int S^+ und S^- schreiben und vergleichen.
Sobald Änderung ⇒ Kante gefunden. ⇒ Schnittpkt berechnen
⇒ Gesamtlaufzeit: $O(n)$.

1.5.9. Satz (Zusammenfassung)

1). Der Schnitt von n Halbebenen kann in Zeit $O(n \log n)$ berechnet werden.

2). Falls die definierenden Geraden nach Steigung sortiert gegeben sind, dann ist die Laufzeit $O(n)$.

1.5.10 Bemerkungen:

1) Die geometrische Transformation der Dualität wird auch noch für andere Probleme dieser Vorlesung wichtig sein.

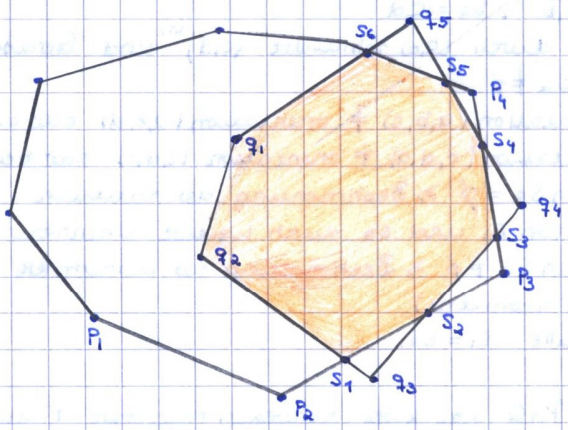
2) Der Schnitt von n Halbebenen kann auch direkt berechnet werden.
   ⇝ Divide & Conquer.

1.5.11. Schnitt von Halbebenen durch Divide and Conquer

Schnitt von zwei konvexen Polygonen.
Hier abgeschlossen (offen → Übung, voriger Abschnitt)
Geg: Seien $P = (p_1 .. p_m)$, $Q = (q_1 .. q_e)$ konvexe abgeschlossene Polygone, geg. durch Eckenfolge gg. den Uhrzeigersinn.
Ausgabe: Eckenfolge von $P \cap Q$ gegen Uhrzeigersinn.

↳ **1.5.11.1. Beispiel.**

Mögliche Ausgabe:
$q_1 q_2 s_1 s_2 s_3 s_4 s_5 s_6$

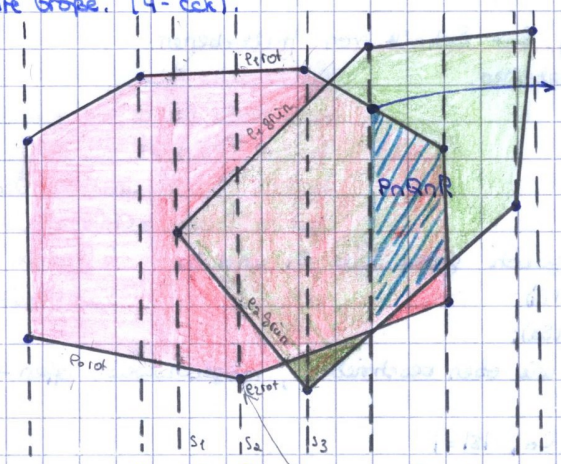


↳ **1.5.11.2. Idee:**

Zerlege Ebene in Regionen so, dass für jede Region R $(P \cap Q) \cap R$ einfach zu berechnen ist.

↳ **1.5.11.3 Regionen:** Vertikale Streifen.

Zeichne durch jede Ecke von P und Q eine senkrechte Gerade.
Jeweils zwei benachbarte Senkrechten definieren eine Region, nämlich den vertikalen Streifen dazwischen.
Dann ist für jeden Streifen R $R \cap P$ und $R \cap Q$ ein Trapezoid, d.h. haben konstante Größe. (4-Eck).



Es gilt $P \cap Q \cap R = (P \cap R) \cap (Q \cap R)$
wegen konst. Größe braucht Zeit $O(1)$

1.5.11.4. Algorithmus:

1). Berechne Streifen von links nach rechts sortiert in Zeit $O(n)$, wobei $n = m+p$. Gesamtzahl der Ecken, durch Mischen der Eckenfolgen von P und Q. (siehe D&C für CH.)

2). Berechne für jeden Streifen R Trapezoide $P \cap R$ und $Q \cap R$. Das geht auch in $O(n)$, da wir Eckenfolgen (siehe 1) von links nach rechts sortiert haben.

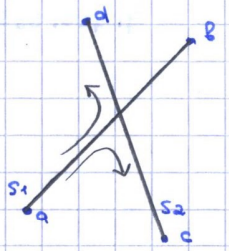
3) Berechne für jeden Streifen R $P \cap Q \cap R := (P \cap R) \cap (Q \cap R)$
Zeit $O(1)$ pro Streifen.

4). Zusammen kleben der Teile aus 3)
Insbesondere Elimination von Ecken, die nicht zur Ausgabe gehören.
Laufzeit: $O(n)$ (Durchlaufen der Senkrechten von links nach rechts).
→ Eckenfolge von $P \cap Q$ gg. Uhrzeigersinn.

1.5.11.5. Implementierungsdetails:

Teilprobleme:

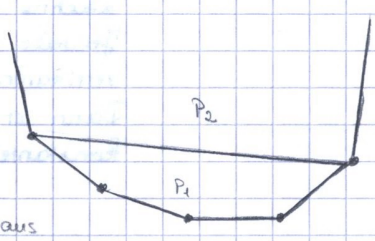
- Test ob Segmente s_1 ein anderes schneidet.
  $s_1 = \overline{a,b}$, $s_2 = \overline{c,d}$
  Gerade durch (a,b) schneidet (c,d) (1) und Gerade durch (c,d) schneidet (a,b) (2)
  $\Leftrightarrow s_1 \cap s_2 \neq \emptyset$
  (1) orientation(a,b,c) $\neq$ orientation(a,b,d) oder beide 0
  (2) orientation(c,d,a) $\neq$ orientation(c,d,b) oder beide 0.
- falls $s_1 \cap s_2 = \emptyset$ ⇒ Bestimmung der relativen Lage von s_1 und s_2 durch weitere Orientierungstests.
- falls $s_1 \cap s_2 \neq \emptyset$ ⇒ Bestimmung des Schnittpkte (anal. Geometrie)
- Sonderfälle: $s_1 = s_2$.



1.5.12. Satz: Der Schnitt $P \cap Q$ von zwei konvexen Polygonen P und Q kann in Zeit $O(n)$ berechnet werden, wobei n = Summe der Ecken von P und Q.
Mischschritt

1.5.13. Fragen:

1) Geht das schneller, wenn alle Polygone im Voraus geg. sind, für die man evtl. Schnittoperationen ausführt.
2). Existiert Algorithmus, der output-sensitiv ist?
Dh. Laufzeit hängt von Größe der Ausgabe ab.
→ siehe nächster Abschnitt über konvexe Polygone.



1.5.14. Divide & Conquer - Alg. für Schnitt von Halbebenen.

Sei S Menge von Halbebenen.

```
INTERSECT(S)
if |S| = 1 then
    return S;
else
    teile S in zwei gleich große Teile S1 und S2
    P ← INTERSECT(S1);
    Q ← INTERSECT(S2);
    return P ∩ Q   // wie oben beschrieben, möglicherweise offen → Übung.
```

Ein unbeschr. Polygon besteht aus

- einem beschr. Pol. P_1 und unbeschr. Pol. P_2 ⇒ $P = P_1 \cup P_2$
  Berechne $P_2 \cap Q_2$ mit D&C und Mischritt von oben ⇒ $O(n \log n)$
  Berechne $P_2 \cap Q_1$ und $P_1 \cap Q_2$ als Schnitt von Polygon mit drei Geraden in Zeit $3 \cdot O(n \log n) = O(n \log n)$. Anschließend berechne $P_2 \cap Q_2$ als Schnitt von 6 Geraden in $O(9) = O(1)$
  ⇒ Gesamtlaufzeit: $O(n \log n)$.

1.5.15 Laufzeit:

$$T(n) = \begin{cases} c_0, & |S| = 1 \\ c_1 n + 2 \cdot T(n/2), & |S| > 1 \end{cases}$$

⇒ $T(n) = O(n \log n)$
Alternative zu Dualitätsalgorithmus

1.5.16. Frage: Welcher Algorithmus ist in welchen Fällen schneller? Wahrscheinlich Divide & Conquer

Kapitel II: Konvexe Polygone

2.1. Einführung:

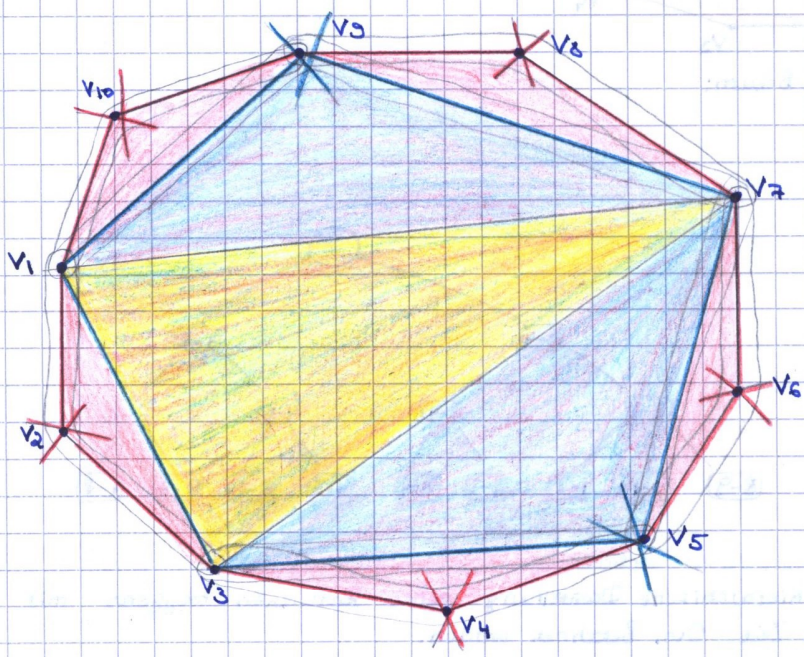
2.1.1. Ziel: Datenstruktur für konvexe Polygone, die verschiedene Operationen effizient unterstützt. → Hierarchische Darstellung.

2.1.2. Idee: Investiere Zeit und Platz $O(n)$ in Aufbau dieser Struktur, um nachfolgende Operationen billig zu machen.
Lohnt sich nur, wenn mit demselben Polygon viele Operationen durchgeführt werden.
Hierarchische Darstellung: Konstruiere Folge von Teilpolygonen, die das Polygon immer genauer beschreiben.

2.1.3. Def: Sei P konvexes Polygon mit n Ecken $v_1 .. v_n$.
Eine Folge $P_0, P_1, ..., P_k$ von konvexen Polygonen heißt (innere) hierarchische Darstellung von P, wenn gilt:

1) P_0 hat ≤ 4 Ecken
2) $P_k = P$, $P_i \neq P_{i+1}$ $\forall i \in \{0..k\}$
3) Jede Ecke von P_i ist Ecke von P_{i+1} und von vier aufeinanderfolgenden Ecken von P_{i+1} ist mindestens eine und höchstens drei von P_i. (für $0 \leq i \leq k$).

2.1.4. Beispiel:



$P = v_1 .. v_{10} = P_2$
$P_1 = v_1 v_3 v_5 v_7 v_9$
$P_0 = v_1 v_3 v_7$
$\Rightarrow \{P_i\}_{i=0}^{2}$ ist die Hierarchische Darstellung von P.

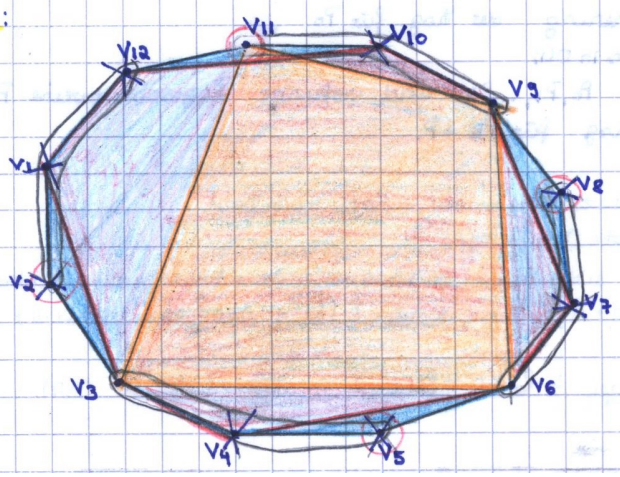
Bea: P_0 kann auch $v_1 v_3$ sein.
Dann wäre die Darstellung bzgl der Hierarchie noch tiefer.
$P_k = P_2$ $|P_2| = 10$
$|P_{k-1}| = |P_1| = 5$

2.1.5 Bemerkung: P_{i-1} entsteht also aus P_i durch entfernen einiger Ecken.
- von jeweils drei aufeinander folgenden Ecken von P_i wird mindestens eine entfernt.
- es werden nie vier aufeinanderfolgende entfernt.

2.1.6. Eigenschaften:

1) Beim Übergang von $P_{i+1} \to P_i$ verlieren wir mindestens einen konstanten Bruchteil der Ecken ($1/3$)
2) P_{i+1} ist ähnlich zu P_i, da wir nie mehr als drei aufeinanderfolgende Ecken entfernen.

2.1.7. Beispiel:



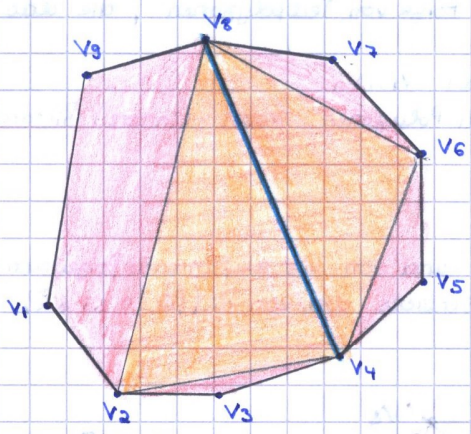
Blau: P_0, $|P_0| = 12$
rot: P_{k-1}: es wurden möglichst wenig Ecken entfernt.
$|P_{k-1}| = 8$
$P_k \to P_{k-1}$ wurden $\frac{1}{3} \cdot |P_k| = \frac{1}{3} \cdot 12 = 4$ entfernt!
orange: P_{k-1}: es wurden möglichst viele Ecken entfernt.
$|P_{k-1}| = 4$
$P_k \to P_{k-1}$ wurden 8 Ecken entfernt!

2.1.8 Alternative Darstellung beim Zeichnen:

balancierte Baum über Kanten von P_i $(0 \le i \le n)$.
(muß kein binärer Baum sein).

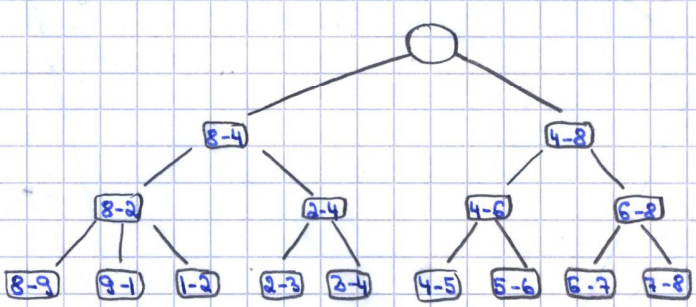
2.1.9 Beispiel:

1) wie vorhin:



$P_0 = v_4v_8$, $|P_0| = 2$
$P_1 = v_8v_9v_1v_2v_3v_4v_5v_6$, $|P_1| = 4$
$P_2 = P = v_1 \dots v_9$, $|P| = 9$.

2) durch balancierten Baum:



2.1.10 Lemma: Könnte man zB mit ähnlichem Alg wie bei Triangulierungsmethode machen.

1) Eine balancierte hierarchische Darstellung eines konvexen Polygons mit n Ecken kann in Zeit $O(n)$ berechnet werden.
2) benötigt Speicherplatz $O(n)$
3) die Tiefe k dieser Darstellung ist $O(\log n)$

Beweis: Alle Teile folgen unmittelbar aus 2.1.6. i). bzw. Baumdarstellung.

zu 2) von $P_i \to P_{i-1}$ verlieren mind $\frac{1}{3}$ der Knoten

$$\Rightarrow \#\text{Knoten} \le n + \tfrac{2}{3}n + \tfrac{2}{3}\left(\tfrac{2}{3}n\right) + \dots = n\left(1 + \tfrac{2}{3} + \tfrac{4}{9} + \dots\right) = n \cdot \sum_{\nu=0}^{\infty} \left(\tfrac{2}{3}\right)^\nu \le n \cdot \sum_{\nu=0}^{n} \left(\tfrac{2}{3}\right)^\nu =$$

$$= n \cdot \frac{\left(\frac{2}{3}\right)^{n+1} - 1}{\frac{2}{3} - 1} = n \cdot \frac{\left(\frac{2}{3}\right)^{n+1} - 1}{-\frac{1}{3}} = n \cdot \frac{1 - \left(\frac{2}{3}\right)^{n+1}}{\frac{1}{3}} = n \cdot 3 \cdot \underbrace{\left(1 - \left(\tfrac{2}{3}\right)^{n+1}\right)}_{\le 1} \le 3n = O(n)$$

2.2. Anwendungen der hierarchischen Darstellung.

verwenden immer dieselbe Strategie:

2.2.1. Strategie:

1). Wir berechnen eine Annäherung der Lsg (des Problems) für P_0
Da P_0 maximal 4 Ecken $\Rightarrow$ in $O(1)$
2) Dann durchlaufe die Folge $P_0, P_1, \dots P_k$ und verfeinere Lösungsschrittweise $P_i \to P_{i+1}$.
3) Am Ende haben wir die Lsg für $P_k = P$.

zu 2.1.10 3): $\text{Höhe}(T) \le \underbrace{\text{Höhe}(\tilde{T})}_{=O(\log n)}$, wobei $\tilde{T}$ binärer Baum mit entspr. Hierarch. Darst.
$\Rightarrow$ Höhe $(T) = O(\log n)$.

zu 2.1.10. 1) Sei $c_1 :=$ Arbeit, um einen Knoten in den Baum zu schreiben (4 Stk); $c_2 :=$ Arbeit, um einen Knoten zu streichen

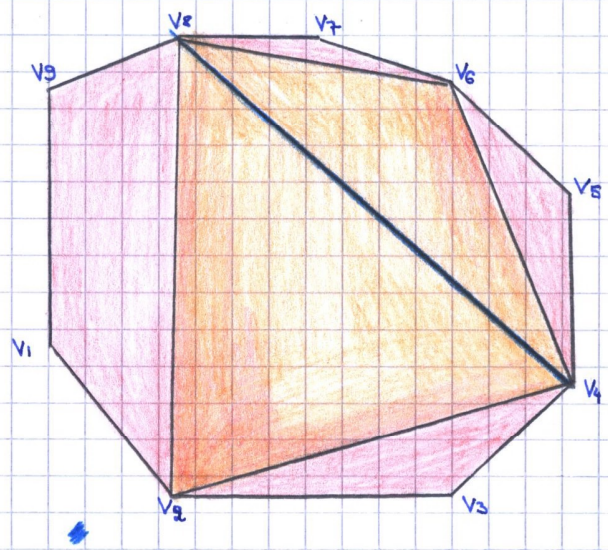
$$\Rightarrow \text{Arbeit} \le c_1 \cdot n + c_2 \cdot \tfrac{1}{3}n + c_1 \cdot \tfrac{2}{3}n + c_2 \cdot \left(\tfrac{1}{3}\right)^2 n + c_1 \left(\tfrac{2}{3}\right)^2 n + \dots \le n c_1 \cdot \sum_{0}^{\infty} \left(\tfrac{2}{3}\right)^\nu + c_2 n \sum_{0}^{n} \left(\tfrac{1}{3}\right)^\nu \le 3c_1 \cdot n + \tfrac{3}{2} \cdot c_2 n = O(n)$$

Aus Listen dann in Baum einfügen $\Rightarrow$ Kostet #Element in allen Listen $\Rightarrow O(n)$.

2.1.8. Alternative Darstellung:

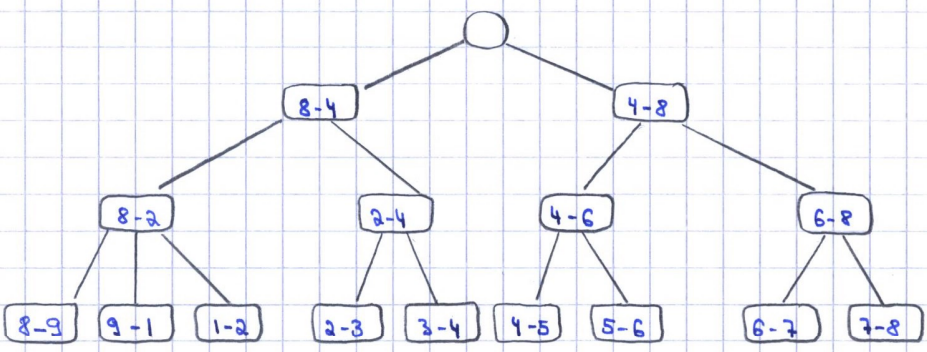
→ balancierter Baum über Kanten von P_i $(0 \le i \le n)$
(muß kein binärer Baum sein.)

2.1.9. Beispiel:



$P_0 = \{v_4, v_8\}$, $|P_0| = 2$
$P_1 = \{v_2, v_4, v_6, v_8\}$, $|P_1| = 4$
$P_2 = \{v_1 .. v_9\}$, $|P_2| = 9$

Darstellung durch einen balancierten Baum:



2.1.10 Lemma:

1. Eine balancierte hierarchische Darstellung eines konvexen Polygons mit n Ecken kann in Zeit $O(n)$ berechnet werden.
2. benötigt Speicherplatz $O(n)$
3. die Tiefe k dieser Darstellung ist $O(\log n)$.

Beweis: Alle Teile folgen unmittelbar aus 2.1.6 i) bzw. Baumdarstellung.

Beachte:

zu 1):

→ könnte man zB. mit ähnlichem Alg. wie bei der Triangulierungsmethode machen

$$c \cdot n + c \cdot \left(\tfrac{2}{3}\right) \cdot n + c \cdot \left(\tfrac{2}{3}\right)^2 \cdot n + \ldots =$$

$$= c \cdot n \cdot \sum_{\nu=0}^{\log n} \left(\tfrac{2}{3}\right)^\nu \le c \cdot n \cdot \sum_{\nu=0}^{\infty} \left(\tfrac{2}{3}\right)^\nu = c \cdot n \cdot \frac{1 - \left(\frac{2}{3}\right)^{n+1}}{1 - \frac{2}{3}} =$$

$$= c \cdot n \cdot 3 \cdot \underbrace{\left(1 - \left(\tfrac{2}{3}\right)^{n+1}\right)}_{\substack{n \to \infty \\ \to 0}} \le \underbrace{M}_{\text{const}} \cdot n = O(n).$$

zu 2):

$\#\text{Knoten} = O(n)$

zu 3): $\text{Höhe}(T) \le \underbrace{\text{Höhe}(\tilde{T})}_{= O(\log n)}$, wobei $\tilde{T}$ binärer Baum.

2.2. Anwendung der Kirkpatrickschen Darstellung:

→ verwenden immer dieselbe Strategie:

2.2.1. Strategie:

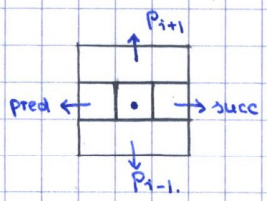
1). Wir berechnen eine Annäherung der Lösung für P_0.
Da P_0 maximal vier Ecken ⇒ in $O(1)$.

2). Dann durchlaufe die Folge $P_0, P_1 .. P_k$ und verfeinere die Lösung schrittweise $P_i \to P_{i+1}$.

3) Am Ende haben wir die Lsg für $P_k = P$.

2.2.2. Darstellung im Rechner:

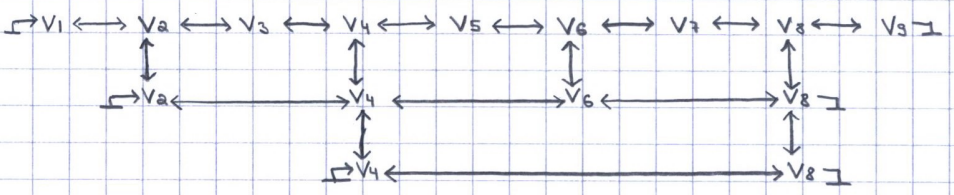
- P_i, $0 \le i \le k$ als doppelt verkettete Liste der Ecken und jede Ecke von P_i zeigt zusätzlich auf ihre Kopie in P_{i+1} und umgekehrt.



- explizite Speicherung des Kantenbaumes
→ Dynamisierung (Ecken einfügen | löschen)
(wie bei Suchbäumen).

2.2.3 Beispiel:

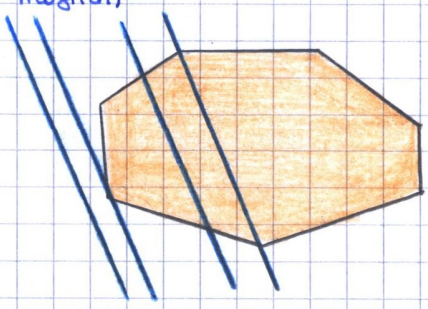
Sei P gegeben wie in 2.1.9.
⇒



2.24. Anwendung: Schnitt mit einer Geraden:

⟶ 2.2.4.1. Ziel:

Geg: Hierarchische Darstellung $P_0 \dots P_k$ eines konvexen Polygons P und eine Gerade g
Ausgabe: $P \cap g$
Konvexität ⇒ $P \cap g$ = Strecke (wenn nicht leer und Ecke)
⇒ Es genügt Schnittpkte a, b mit den Randsegmenten zu berechnen.
Fälle: 1) $a \neq b$ (allg. Fall, Ecken möglich)
2) $a = b$ (eine Ecke)
3) ex. nicht! $P \cap g = \emptyset$.



⟶ 2.24.2. Idee für Algorithmus:

1). Schnittest mit P_0:
2 Fälle: a) $P_0 \cap g = \emptyset$
b) $P_0 \cap g \neq \emptyset$
Im Fall B) → STOP
Im Fall a):
sei $g_0 \in P_0$ mit minimalem Abstand zu g.

Kapitel II: Konvexe Polygone

2.1. Einführung:

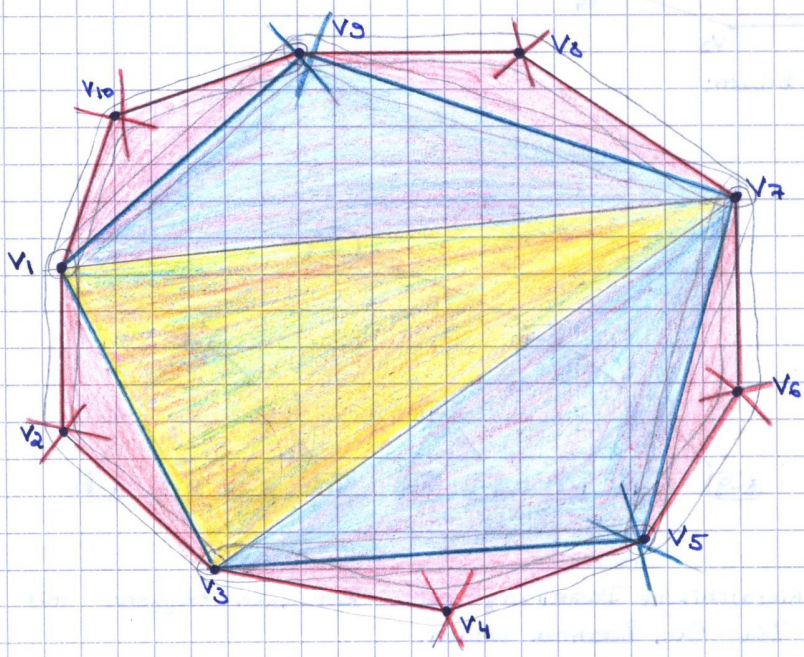
2.1.1. Ziel: Datenstruktur für konvexe Polygone, die verschiedene Operationen effizient unterstützt. → Hierarchische Darstellung.

2.1.2. Idee: Investiere Zeit und Platz $O(n)$ in Aufbau dieser Struktur, um nachfolgende Operationen billig zu machen.
Lohnt sich nur, wenn mit demselben Polygon viele Operationen durchgeführt werden.
Hierarchische Darstellung: Konstruiere Folge von Teilpolygonen, die das Polygon immer genauer beschreiben.

2.1.3. Def: Sei P konvexes Polygon mit n Ecken $v_1 .. v_n$.
Eine Folge $P_0, P_1, ..., P_k$ von konvexen Polygonen heißt (innere) hierarchische Darstellung von P, wenn gilt:

1) P_0 hat ≤ 4 Ecken
2) $P_k = P$, $P_i \neq P_{i+1}$ $\forall i \in \{0..k\}$
3) Jede Ecke von P_i ist Ecke von P_{i+1} und von vier aufeinanderfolgenden Ecken von P_{i+1} ist mindestens eine und höchstens drei von P_i. (für $0 \leq i \leq k$).

2.1.4. Beispiel:



$P = v_1 .. v_{10} = P_2$
$P_1 = v_1 v_3 v_5 v_7 v_9$
$P_0 = v_1 v_3 v_7$
$\Rightarrow \{P_i\}_{i=0}^{2}$ ist die Hierarchische Darstellung von P.

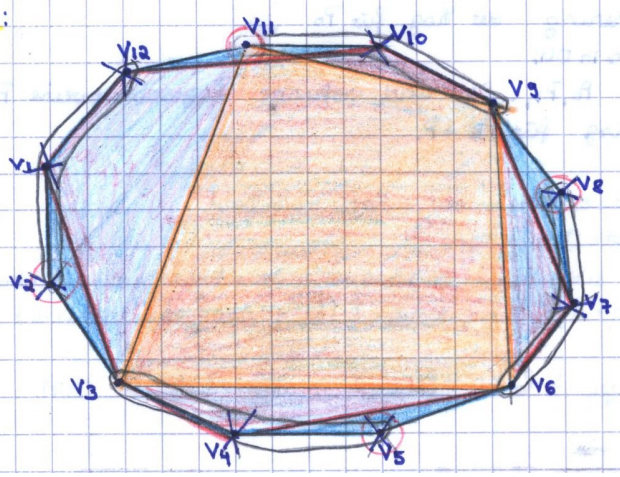
Bea: P_0 kann auch $v_1 v_3$ sein.
Dann wäre die Darstellung bzgl der Hierarchie noch tiefer.
$P_k = P_2$ $|P_2| = 10$
$|P_{k-1}| = |P_1| = 5$

2.1.5 Bemerkung: P_{i-1} entsteht also aus P_i durch entfernen einiger Ecken.
- von jeweils drei aufeinander folgenden Ecken von P_i wird mindestens eine entfernt.
- es werden nie vier aufeinanderfolgende entfernt.

2.1.6. Eigenschaften:

1) Beim Übergang von $P_{i+1} \to P_i$ verlieren wir mindestens einen konstanten Bruchteil der Ecken ($1/3$)
2) P_{i+1} ist ähnlich zu P_i, da wir nie mehr als drei aufeinanderfolgende Ecken entfernen.

2.1.7. Beispiel:



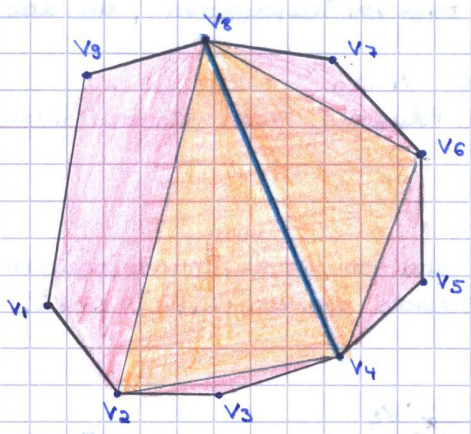
Blau: P_0, $|P_0| = 12$
rot: P_{k-1}: es wurden möglichst wenig Ecken entfernt.
$|P_{k-1}| = 8$
$P_k \to P_{k-1}$ wurden $\frac{1}{3} \cdot |P_k| = \frac{1}{3} \cdot 12 = 4$ entfernt!
orange: P_{k-1}: es wurden möglichst viele Ecken entfernt.
$|P_{k-1}| = 4$
$P_k \to P_{k-1}$ wurden 8 Ecken entfernt!

2.1.8 Alternative Darstellung beim Zeichnen:

balancierte Baum über Kanten von P_i $(0 \le i \le n)$.
(muß kein binärer Baum sein).

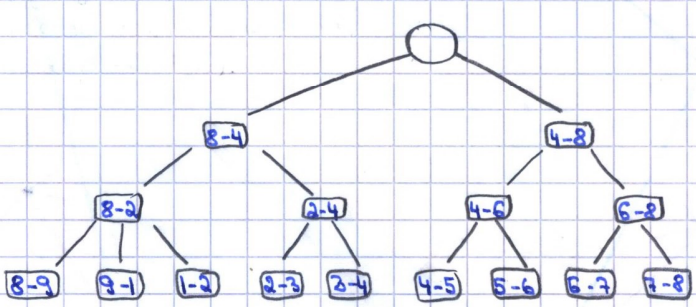
2.1.9 Beispiel:

1) wie vorhin:



$P_0 = v_4v_8$, $|P_0| = 2$
$P_1 = v_8v_9v_1v_2v_3v_4v_5v_6$, $|P_1| = 4$
$P_2 = P = v_1 \dots v_9$, $|P| = 9$.

2) durch balancierten Baum:



2.1.10 Lemma: Könnte man zB mit ähnlichem Alg wie bei Triangulierungsmethode machen.

1) Eine balancierte hierarchische Darstellung eines konvexen Polygons mit n Ecken kann in Zeit $O(n)$ berechnet werden.
2) benötigt Speicherplatz $O(n)$
3) die Tiefe k dieser Darstellung ist $O(\log n)$

Beweis: Alle Teile folgen unmittelbar aus 2.1.6. i). bzw. Baumdarstellung.

zu 2) von $P_i \to P_{i-1}$ verlieren mind $\frac{1}{3}$ der Knoten

$$\Rightarrow \#\text{Knoten} \le n + \tfrac{2}{3}n + \tfrac{2}{3}\left(\tfrac{2}{3}n\right) + \dots = n\left(1 + \tfrac{2}{3} + \tfrac{4}{9} + \dots\right) = n \cdot \sum_{\nu=0}^{\infty} \left(\tfrac{2}{3}\right)^\nu \le n \cdot \sum_{\nu=0}^{n} \left(\tfrac{2}{3}\right)^\nu =$$

$$= n \cdot \frac{\left(\frac{2}{3}\right)^{n+1} - 1}{\frac{2}{3} - 1} = n \cdot \frac{\left(\frac{2}{3}\right)^{n+1} - 1}{-\frac{1}{3}} = n \cdot \frac{1 - \left(\frac{2}{3}\right)^{n+1}}{\frac{1}{3}} = n \cdot 3 \cdot \underbrace{\left(1 - \left(\tfrac{2}{3}\right)^{n+1}\right)}_{\le 1} \le 3n = O(n)$$

2.2. Anwendungen der hierarchischen Darstellung.

verwenden immer dieselbe Strategie:

2.2.1. Strategie:

1). Wir berechnen eine Annäherung der Lsg (des Problems) für P_0
Da P_0 maximal 4 Ecken $\Rightarrow$ in $O(1)$
2) Dann durchlaufe die Folge $P_0, P_1, \dots P_k$ und verfeinere Lösungsschrittweise $P_i \to P_{i+1}$.
3) Am Ende haben wir die Lsg für $P_k = P$.

zu 2.1.10 3): $\text{Höhe}(T) \le \underbrace{\text{Höhe}(\tilde{T})}_{=O(\log n)}$, wobei $\tilde{T}$ binärer Baum mit entspr. Hierarch. Darst.
$\Rightarrow$ Höhe $(T) = O(\log n)$.

zu 2.1.10. 1) Sei $c_1 :=$ Arbeit, um einen Knoten in den Baum zu schreiben (4 Stk); $c_2 :=$ Arbeit, um einen Knoten zu streichen

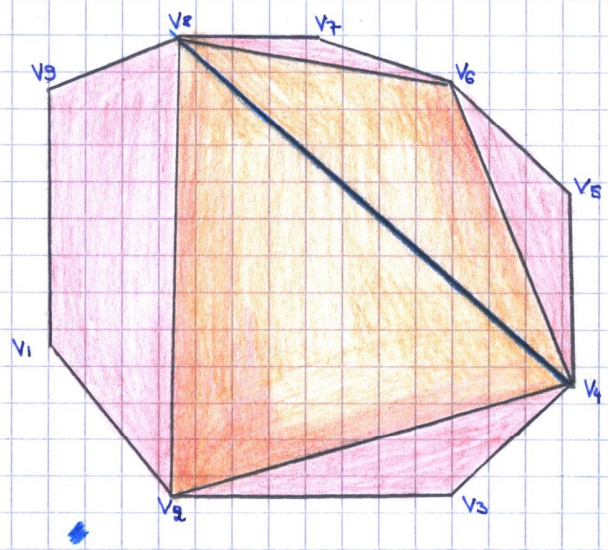
$$\Rightarrow \text{Arbeit} \le c_1 \cdot n + c_2 \cdot \tfrac{1}{3}n + c_1 \cdot \tfrac{2}{3}n + c_2 \cdot \left(\tfrac{1}{3}\right)^2 n + c_1 \left(\tfrac{2}{3}\right)^2 n + \dots \le n c_1 \cdot \sum_{0}^{\infty} \left(\tfrac{2}{3}\right)^\nu + c_2 n \sum_{0}^{n} \left(\tfrac{1}{3}\right)^\nu \le 3c_1 \cdot n + \tfrac{3}{2} \cdot c_2 n = O(n)$$

Aus Listen dann in Baum einfügen $\Rightarrow$ Kostet #Element in allen Listen $\Rightarrow O(n)$.

2.1.8. Alternative Darstellung:

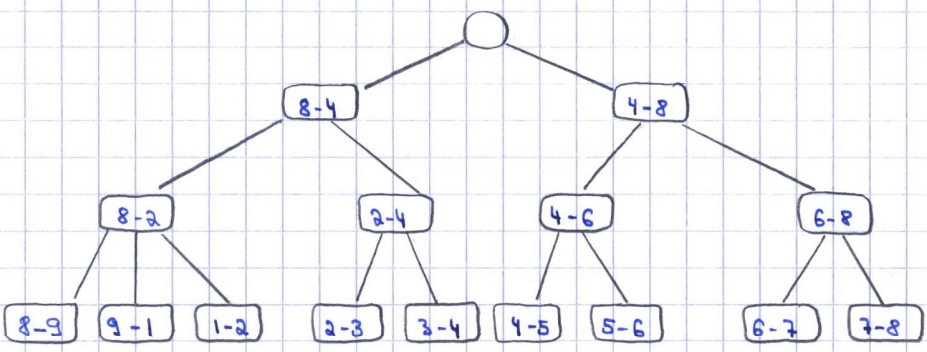
→ balancierter Baum über Kanten von P_i $(0 \le i \le n)$
(muß kein binärer Baum sein.)

2.1.9. Beispiel:



$P_0 = \{v_4, v_8\}$, $|P_0| = 2$
$P_1 = \{v_2, v_4, v_6, v_8\}$, $|P_1| = 4$
$P_2 = \{v_1 .. v_9\}$, $|P_2| = 9$

Darstellung durch einen balancierten Baum:



2.1.10 Lemma:

1. Eine balancierte hierarchische Darstellung eines konvexen Polygons mit n Ecken kann in Zeit $O(n)$ berechnet werden.
2. benötigt Speicherplatz $O(n)$
3. die Tiefe k dieser Darstellung ist $O(\log n)$.

Beweis: Alle Teile folgen unmittelbar aus 2.1.6 i) bzw. Baumdarstellung.

Beachte:

zu 1):

→ könnte man zB. mit ähnlichem Alg. wie bei der Triangulierungsmethode machen

$$c \cdot n + c \cdot \left(\tfrac{2}{3}\right) \cdot n + c \cdot \left(\tfrac{2}{3}\right)^2 \cdot n + \ldots =$$

$$= c \cdot n \cdot \sum_{\nu=0}^{\log n} \left(\tfrac{2}{3}\right)^\nu \le c \cdot n \cdot \sum_{\nu=0}^{\infty} \left(\tfrac{2}{3}\right)^\nu = c \cdot n \cdot \frac{1 - \left(\frac{2}{3}\right)^{n+1}}{1 - \frac{2}{3}} =$$

$$= c \cdot n \cdot 3 \cdot \underbrace{\left(1 - \left(\tfrac{2}{3}\right)^{n+1}\right)}_{\substack{n \to \infty \\ \to 0}} \le \underbrace{M}_{\text{const}} \cdot n = O(n).$$

zu 2):

$\#\text{Knoten} = O(n)$

zu 3): $\text{Höhe}(T) \le \underbrace{\text{Höhe}(\tilde{T})}_{= O(\log n)}$, wobei $\tilde{T}$ binärer Baum.

2.2. Anwendung der Kirkpatrickschen Darstellung:

→ verwenden immer dieselbe Strategie:

2.2.1. Strategie:

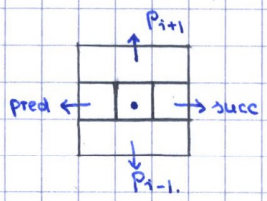
1). Wir berechnen eine Annäherung der Lösung für P_0.
Da P_0 maximal vier Ecken ⇒ in $O(1)$.

2). Dann durchlaufe die Folge $P_0, P_1 .. P_k$ und verfeinere die Lösung schrittweise $P_i \to P_{i+1}$.

3) Am Ende haben wir die Lsg für $P_k = P$.

2.2.2. Darstellung im Rechner:

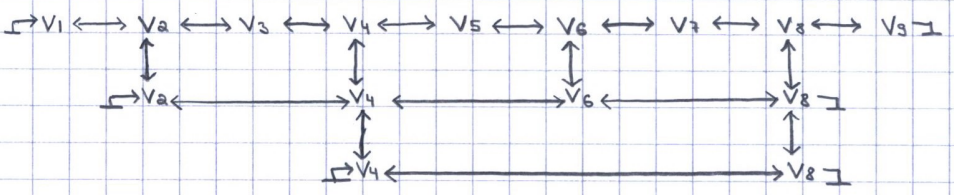
- P_i, $0 \le i \le k$ als doppelt verkettete Liste der Ecken und jede Ecke von P_i zeigt zusätzlich auf ihre Kopie in P_{i+1} und umgekehrt.



- explizite Speicherung des Kantenbaumes
→ Dynamisierung (Ecken einfügen | löschen)
(wie bei Suchbäumen).

2.2.3 Beispiel:

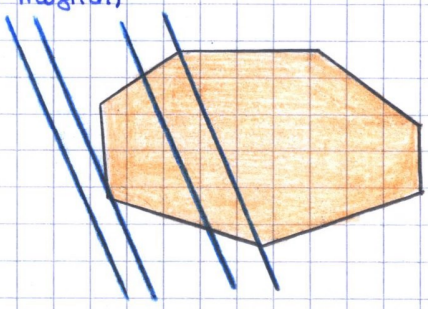
Sei P gegeben wie in 2.1.9.
⇒



2.24. Anwendung: Schnitt mit einer Geraden:

⟶ 2.2.4.1. Ziel:

Geg: Hierarchische Darstellung $P_0 \dots P_k$ eines konvexen Polygons P und eine Gerade g
Ausgabe: $P \cap g$
Konvexität ⇒ $P \cap g$ = Strecke (wenn nicht leer und Ecke)
⇒ Es genügt Schnittpkte a, b mit den Randsegmenten zu berechnen.
Fälle: 1) $a \neq b$ (allg. Fall, Ecken möglich)
2) $a = b$ (eine Ecke)
3) ex. nicht! $P \cap g = \emptyset$.

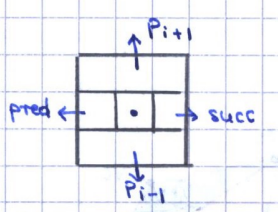


⟶ 2.24.2. Idee für Algorithmus:

1). Schnittest mit P_0:
2 Fälle: a) $P_0 \cap g = \emptyset$
b) $P_0 \cap g \neq \emptyset$
Im Fall B) → STOP
Im Fall a):
sei $g_0 \in P_0$ mit minimalem Abstand zu g.

2.2.2. Darstellung im Rechner:

- P_i , $0 \le i \le k$ als doppelt verkettete Liste der Ecken + jede Ecke von P_i zeigt zusätzlich auf ihre Kopie in P_{i+1} und umgekehrt

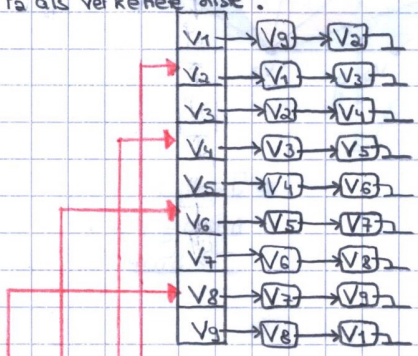


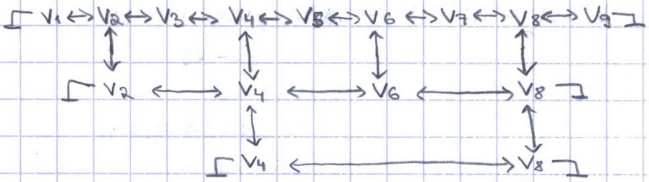
- explizite Speicherung des Kantenbaumes
  → Dynamisierung (Ecken einfügen / löschen)
  (wie bei Suchbäumen)

2.2.3 Beispiel:

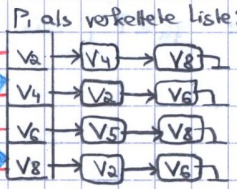
Sei P gegeben wie in 2.1.9.

P_2 als verkettete Liste:





P_1 als verkettete Liste:



P_0 als verkettete Liste:

2.2.4. Anwendung: Schnitt mit einer Geraden:

→ 2.2.4.1. Ziel:

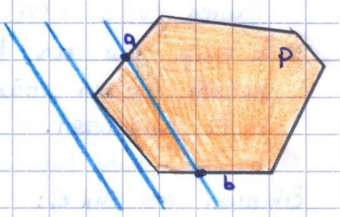
Geg: Hierarchische Darstellung $P_0 .. P_k$ eines konvexen Polygons P und eine Gerade g.

Ausgabe: $P \cap g$

Konvexität → $P \cap g$ = Strecke

→ Es genügt Schnittpkte a, b mit den Randsegmenten zu berechnen.

Fälle: 1) $a \neq b$ (allg. Fall, Ecken möglich)
2) $a = b$ (eine Ecke)
3) ex. nicht!, Schnitt = $\emptyset$



2.2.4.2. Idee für Algorithmus:

1) Schnittest mit P_0:
   2 Fälle: a) $P_0 \cap g = \emptyset$ konst Operationen, weil $|P_0| \le 4$
   b) $P_0 \cap g \neq \emptyset$
   Ann: wir haben Fall a)
   → sei $p_0 \in P_0$ mit minimalen Abstand zu g
   Im Fall b → STOP

d). Durchlaufe die hierarchische Darstellung $P_1, P_2 \ldots$ und finde entweder:

a) $q_i \in P_i$ mit minimalen Abstand, falls $P_i \cap g = \emptyset$

oder

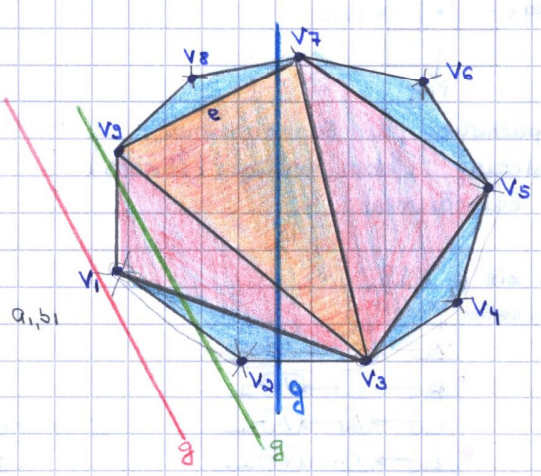
b) Schnittpkte a_i, b_i von P_i mit g.

Im Fall b) → STOP.

Bsp: $P = v_1 .. v_9 = P_2$

$P_1 = v_1 v_2 v_8 v_7 v_9$

$P_0 = v_2 v_7 v_9$



Im Alg.:

1) $P_0 \cap g = \emptyset$
   ⇒ $q_0 = v_9$
2) $P_1 \cap g \neq \emptyset$
   ⇒ Fall b) ⇝ finde a_1, b_1
   STOP

Im Alg.:

1) $P_0 \cap g = \emptyset$
   ⇒ $q_0 = v_9$
2) $P_1 \cap g \neq \emptyset$
   ⇒ Teil a) ⇒ $q_1 = v_1$

   $P_2 \cap g = \emptyset$ -
   ⇒ Teil a) ⇒ $q_2 = v_1$.

Im Alg.:

1) $P_0 \cap g \neq \emptyset$
   ⇒ STOP

Zwei mögliche Ausgänge: (wenn $i = k$)

a) $P_k \cap g = P_n \cap g = \emptyset$ und wir kennen Ecke $q_k \in P$, die g am nächsten

b) $P_i \cap g = (a_i, b_i)$ für $0 \leq i \leq k$

Im Bsp:

- ⇝ Ausgang b) ⇒ $P_1 \cap g = (a_1, b_1)$ für $i = 1$.
- ⇝ Ausgang a) ⇒ $P_n \cap g = \emptyset$, $q_2 = v_1 \in P$.
- ⇝ Ausgang b) ⇒ $P_0 \cap g = (a_0, b_0)$ für $i = 0$

Ausgang a) ⇒ fertig, $P_n \cap g = \emptyset$

Ausgang b) ⇒ Wir müssen Schnittpkte (a_i, b_i) später verfeinern bis (a_k, b_k) gefunden. } Phase 2.

Nun: Übergang von $q_i \to q_{i+1}$ (Fall a)) Bzw. Test, ob Fall b) gilt.

⟶ 2.2.4.3 Laufzeit:

Phase 1: Kandidaten für nächsten Pkt q_{i+1} sind schon genau die Ecken von P_{i+1} die Verfeinerungen der Nachbarkanten von q_i in P_i

Das folgt unmittelbar aus der Konvexität.

- Im Bsp: $i = 0$, $q_0 = v_9$, Kand. für q_1 ist Ecke von P_1 nämlich v_1
  Dies ist die Ecke der Verfeinerungen der Nachbarkanten von $q_0 = v_9$, nämlich der Kanten $(v_9, v_1) \wedge (v_1, v_3)$

⇒ Es muß nur ein Minimum in einer konstanten Zahl von Kandidaten gesucht werden, nämlich in den Verfeinerungen der Kanten, die an q_i in P_i angrenzen.

- Im Bsp: Suche Min. in (v_9, v_1), (v_1, v_3)

Erkennen von Fall b):

Teste in der Kandidatenmenge, ob ein Pkt auf der anderen Seite von g liegt. (Orientation)

⇒ (a_{i+1}, b_{i+1}) in Konst. Zeit denn Konst viele Kand

(Schnittberechnung mit Verfeinerungskanten).

- Im Bsp: $i = 0$, $q_0 = v_9$ ⇒ Kandidatenmenge: (v_9, v_1) und (v_1, v_3)

  v_1 liegt auf der anderen Seite von g, denn $\text{orient}(v_9, v_1, g) > 0$

  ⇒ Teil b) erkannt

  (bez: $\text{orient}(v_9, v_3, g) \leq 0$)

Laufzeit: • $O(1)$ pro Hierarchiestufe
• Haben nur $O(\log n)$ Stufen
$\Rightarrow O(\log n)$ für die erste Phase.

Nach Phase 1 haben wir zwei Fälle:
Fall a) $\Rightarrow$ fertig.
Fall b) $\Rightarrow$ machen weiter zu Phase 2.

Phase 2: Geg: $\{a_i, b_i\} = P \cap g$ für $i < k$ (sonst fertig) und es sind keine Ecken (sonst ebenfalls fertig).

Betrachte a_i (b_i analog):

a_{i+1} muss Schnittpkt sein mit einer Verfeinerungskante von e, wobei $a_i = e \cap g$

Im Bsp: Betrachte a_0, $i = 0 < 2$

a_1 ist Schnpkt mit Verfeinerungskante von e, wobei $a_0 = e \cap g$ und der Geraden g.

Die Verfeinerungskanten von e sind (v_9, v_8) und (v_8, v_7)

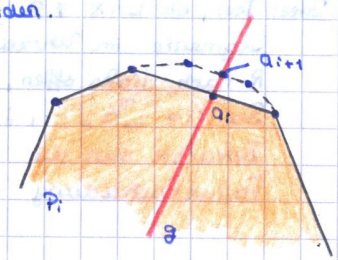
$\Rightarrow a_{i+1}$ kann in Zeit $O(1)$ aus a_i berechnet werden.

(b_{i+1} analog aus b_i)

Weil # Verf.kanten konst. Genauer ≤ 4.

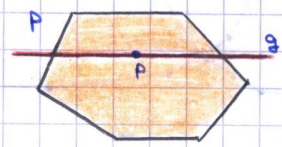
$\Rightarrow$ Laufzeit von Phase 2 auch $O(\log n)$.

$\Rightarrow$ Gesamte Laufzeit also $O(\log n)$!



2.2.5 Satz: Für ein konvexes Polygon P kann:
1) die Hierarchische Darstellung in Zeit $O(n)$ aufgebaut werden.
2) Braucht Platz $O(n)$
3) Für beliebige Gerade g kann $P \cap g$ mit Hilfe der Hierarchischen Darstellung in Zeit $O(\log n)$ Berechnet werden.

2.2.6 Bem.
1) Teil 3) $\Rightarrow$ Alternative $\log n$-Alg. zum Test, ob Pkt $p \in P$ ist.
Idee:



$p \in P \Leftrightarrow p \in P \cap g$
- Betr. Gerade g durch p parallel zu x-Achse
- Berechne hierar. Darst.
- Berechne Schnpkte
- Schaue ob $p_x \in [a_x, b_x]$

$\Rightarrow$ Zeit: $O(1) + O(n) + O(\log n) + O(1) = O(n)$!

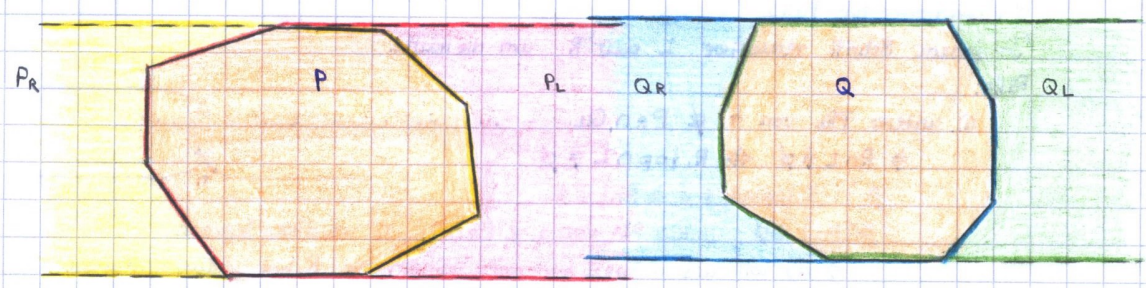
2) Hierar. Darstellung auch im $\mathbb{R}^3$ möglich, d.h. für konvexe Polyeder.
(dazu später mehr...).

2.3 Weiteres Problem auf konvexen Polygonen: Schnitt von zwei konvexen Polygonen.

2.3.1 Ziel: Geg: zwei konvexe Polygone P, Q
Test ob $P \cap Q \neq \emptyset$

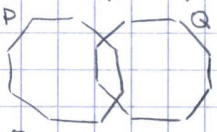
Anm. Wir wissen schon, wie man $P \cap Q$ in Zeit $O(n)$ Berechnet, wobei n = Gesamtzahl der Ecken.

2.3.2 Idee:
1) Betrachte ein einfacheres Problem: Test, ob sich zwei polygonale Ketten schneiden.
- Zerlege P in linken (P_L) & rechten (P_R) Rand durch Zerschneiden an Extremalpkten gemäß yx-Sortierung der Ecken.
- Erweitere durch entsprechende horizontale Strahlen.



Beobachtung: $P \cap Q \neq \emptyset \Leftrightarrow P_L \cap Q_R \neq \emptyset \wedge P_R \cap Q_L \neq \emptyset$

Beweis: $P \cap Q \neq \emptyset \Rightarrow P_R \cap Q_L \neq \emptyset \wedge P_L \cap Q_R \neq \emptyset$ klar.

Für 

mit $P_R \cap Q_L \neq \emptyset \Leftrightarrow P \cap Q \neq \emptyset$
(hierbei gilt auch $P_L \cap Q_R \neq \emptyset$)

Für 

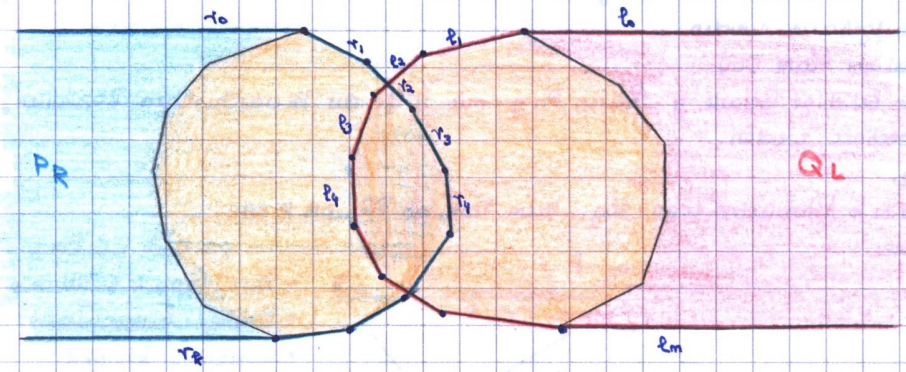
mit $Q_R \cap P_L \neq \emptyset \Leftrightarrow P \cap Q \neq \emptyset$
(hierbei gilt auch $Q_L \cap P_R \neq \emptyset$)

$\Rightarrow$ falls $P_R \cap Q_L \neq \emptyset \wedge P_L \cap Q_R \neq \emptyset \Rightarrow P \cap Q \neq \emptyset$

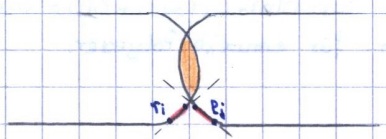
2). Lösung des einfacheren Problems:

Bea: Wir wissen nicht, ob P linkes und Q rechtes Polygon oder umgekehrt.

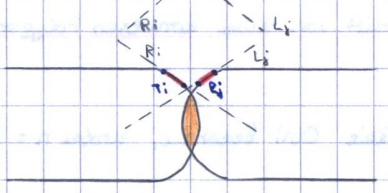
Idee: Test, ob $L \cap R \neq \emptyset$ in Zeit $O(\log n)$, wobei $R = r_0 .. r_k$ gegeben durch die Segmente im Uhrzeigersinn (r_0, r_k Strahlen) und $L = \ell_0 .. \ell_m$ analog gg. Uhrzeigersinn.
R nach links offen, L nach rechts offen.
Sei $i := \lfloor m/2 \rfloor$, $j := \lfloor k/2 \rfloor$
Betrachte nun die mittleren Segmente r_i und ℓ_j.
Fallunterscheidung gemäß der Lage von ℓ_j und r_i auf Geraden R_i und L_j.



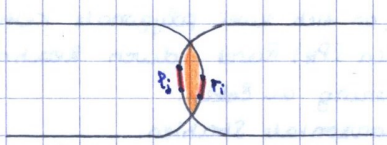
Es gibt nun mehrere Fälle:

1) 

← $P_R \cap Q_L$ oberhalb von den unteren Endpkten von r_i und ℓ_j

2) 

← $P_R \cap Q_L$ unterhalb von den oberen Endpkten von r_i und ℓ_j

3) 

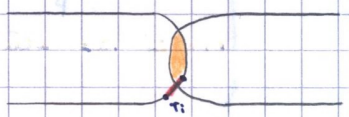
← Endpkte von r_i und $\ell_j \in P_R \cap Q_L$
(Bea: hierbei müssen sich die Polygone nicht notw. schneiden!)

Wir betrachten hier Fall 1

In jedem Schritt reduziere L oder R um die Hälfte.

Fälle:

a). unterer Pkt von $r_i \notin P_R \cap Q_L$



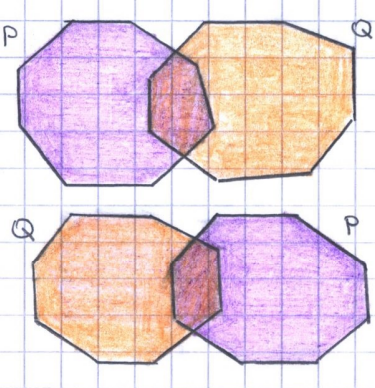
$\Rightarrow P \cap L \neq \emptyset \Leftrightarrow R.top \cap L \neq \emptyset$

"$\Leftarrow$": $R.top \cap L \neq \emptyset \Rightarrow R \cap L \neq \emptyset$

"$\Rightarrow$" $P \cap L \neq \emptyset$, unterer Pkt von $r_i \notin P_R \cap Q_L$, r_i mittleres Segment
$\Rightarrow$ da wir uns im Fall 1 befinden, können wir R.bot rausschmeissen.
$\Rightarrow R.top \cap L \neq \emptyset$.

Beobachtung: $P \cap Q \neq \emptyset \Leftrightarrow P_L \cap Q_R \neq \emptyset \wedge P_R \cap Q_L \neq \emptyset$.

Denn:



Hier: $P \cap Q \neq \emptyset \Leftrightarrow P_R \cap Q_L \neq \emptyset$

Bea: Man weiß in der Regel nicht, welches Polygon rechts und welches links liegt.

Hier: $P \cap Q \neq \emptyset \Leftrightarrow P_L \cap Q_R \neq \emptyset$

$\Rightarrow$ Insgesamt gilt also: falls $P_L \cap Q_R \neq \emptyset \wedge P_R \cap Q_L \neq \emptyset \Rightarrow P \cap Q \neq \emptyset$

„$\Leftarrow$" klar

$\Rightarrow$ die obige Beh.

2) Lösung des einfacheren Problems:

Idee: Test, ob $L \cap R \neq \emptyset$ in Zeit $O(\log n)$, wobei $R = \{r_0 \dots r_k\}$ und $L = \{\ell_0 \dots \ell_m\}$ gegeben durch die Segmente im Uhrzeigersinn (für R) und gegen Uhrzeigersinn (für L).

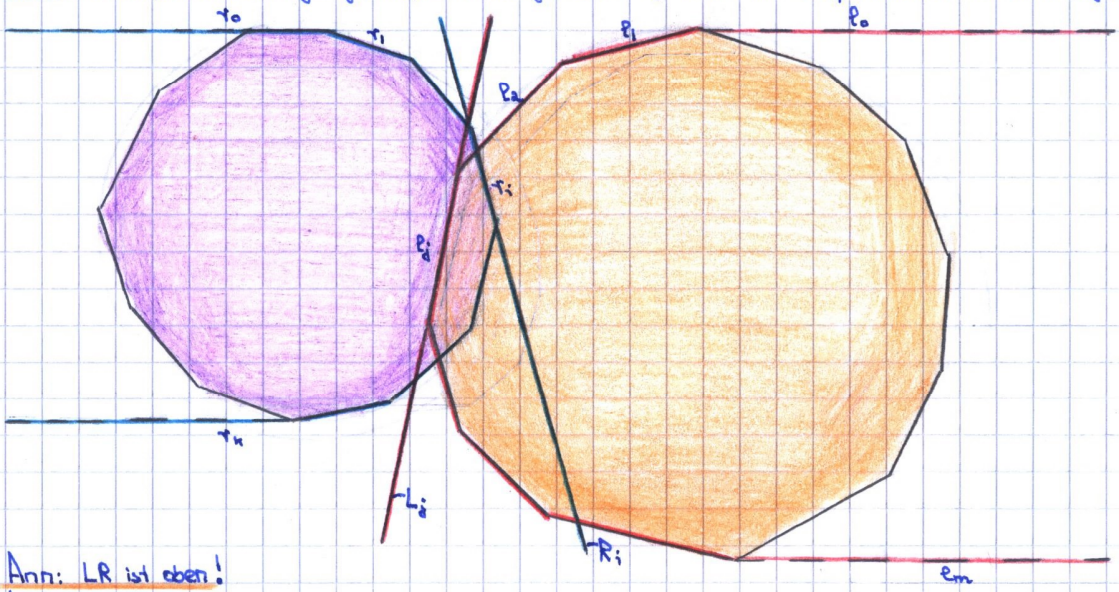
(r_0 und r_k bzw ℓ_0 und ℓ_m sind Strahlen).

R ist nach links offen und L nach rechts offen.

Sei $i := \lfloor m/2 \rfloor$ und $j := \lfloor k/2 \rfloor$

Betrachte nun die mittleren Segmente r_i und ℓ_j

Nun: Fallunterscheidung gemäß der Lage von ℓ_j und r_i auf Geraden R_i und L_j:



Ann: LR ist oben!

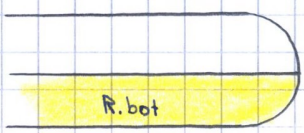
In jedem Schritt reduziere L oder R um die Hälfte.

Fälle:

a) unterer Pkt von r_i ist nicht in LR

$\Rightarrow R \cap L \neq \emptyset \Leftrightarrow R.top \cap L \neq \emptyset$

wobei:



b). unterer Pkt von ℓ_j ist nicht in LR.

$\Rightarrow R \cap L \neq \emptyset \Leftrightarrow R.L.top \neq \emptyset$

c) beide unteren Pkte sind in LR

(i) falls u_{r_i} unterhalb von u_{ℓ_j}

$\Rightarrow R \cap L \neq \emptyset \Leftrightarrow R.top \cap L \neq \emptyset$

(ii) falls u_{ℓ_j} unterhalb von u_{r_i}

$\Rightarrow R \cap L \neq \emptyset \Leftrightarrow R \cap L.top \neq \emptyset$

Hierbei: $u_{r_i} \hat{=}$ unterer Endpkt von r_i

$u_{\ell_j} \hat{=}$ unterer Endpkt von ℓ_j.

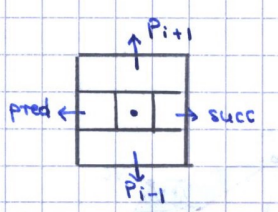
⇒ In Zeit $O(1)$ wird entweder L oder R auf die Hälfte reduziert.
Wiederhole nun bis l_j sich mit r_i schneidet oder bis nur noch konstante Größe von L und R.

2.3.3. Laufzeit: sei $k + m + 2 =: n$
⇒ der eigentliche Test, ob $L \cap R \neq \emptyset$ in Zeit $O(\log n)$.

Kleines Beispiel zur Veranschaulichung:

2.2.2. Darstellung im Rechner:

- P_i , $0 \le i \le k$ als doppelt verkettete Liste der Ecken + jede Ecke von P_i zeigt zusätzlich auf ihre Kopie in P_{i+1} und umgekehrt

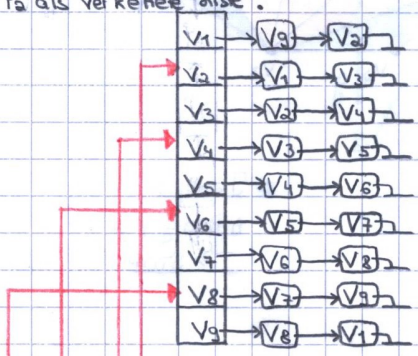


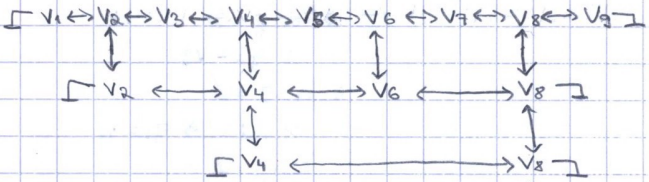
- explizite Speicherung des Kantenbaumes
  → Dynamisierung (Ecken einfügen / löschen)
  (wie bei Suchbäumen)

2.2.3 Beispiel:

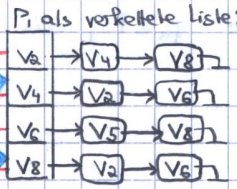
Sei P gegeben wie in 2.1.9.

P_2 als verkettete Liste:





P_1 als verkettete Liste:



P_0 als verkettete Liste:

2.2.4. Anwendung: Schnitt mit einer Geraden:

→ 2.2.4.1. Ziel:

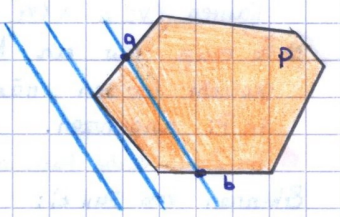
Geg: Hierarchische Darstellung $P_0 .. P_k$ eines konvexen Polygons P und eine Gerade g.

Ausgabe: $P \cap g$

Konvexität → $P \cap g$ = Strecke

→ Es genügt Schnittpkte a, b mit den Randsegmenten zu berechnen.

Fälle: 1) $a \neq b$ (allg. Fall, Ecken möglich)
2) $a = b$ (eine Ecke)
3) ex. nicht!, Schnitt = $\emptyset$



2.2.4.2. Idee für Algorithmus:

1) Schnittest mit P_0:
   2 Fälle: a) $P_0 \cap g = \emptyset$ konst Operationen, weil $|P_0| \le 4$
   b) $P_0 \cap g \neq \emptyset$
   Ann: wir haben Fall a)
   → sei $p_0 \in P_0$ mit minimalen Abstand zu g
   Im Fall b → STOP

d). Durchlaufe die hierarchische Darstellung $P_1, P_2 \ldots$ und finde entweder:

a) $q_i \in P_i$ mit minimalen Abstand, falls $P_i \cap g = \emptyset$

oder

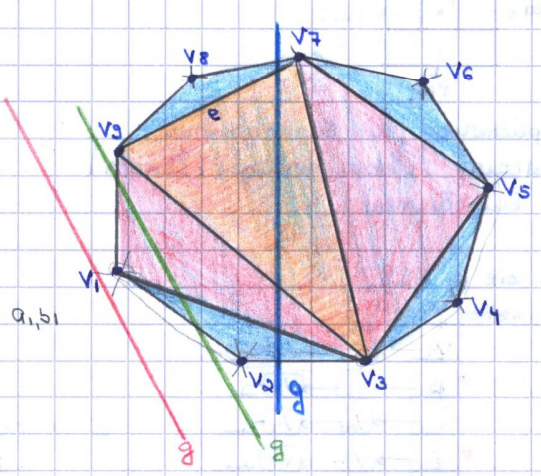
b) Schnittpkte a_i, b_i von P_i mit g.

Im Fall b) → STOP.

Bsp: $P = v_1 .. v_9 = P_2$

$P_1 = v_1 v_2 v_8 v_7 v_9$

$P_0 = v_2 v_7 v_9$



Im Alg.:

1) $P_0 \cap g = \emptyset$
   ⇒ $q_0 = v_9$
2) $P_1 \cap g \neq \emptyset$
   ⇒ Fall b) ⇝ finde a_1, b_1
   STOP

Im Alg.:

1) $P_0 \cap g = \emptyset$
   ⇒ $q_0 = v_9$
2) $P_1 \cap g \neq \emptyset$
   ⇒ Teil a) ⇒ $q_1 = v_1$

   $P_2 \cap g = \emptyset$ -
   ⇒ Teil a) ⇒ $q_2 = v_1$.

Im Alg.:

1) $P_0 \cap g \neq \emptyset$
   ⇒ STOP

Zwei mögliche Ausgänge: (wenn $i = k$)

a) $P_k \cap g = P_n \cap g = \emptyset$ und wir kennen Ecke $q_k \in P$, die g am nächsten

b) $P_i \cap g = (a_i, b_i)$ für $0 \leq i \leq k$

Im Bsp:

- ⇝ Ausgang b) ⇒ $P_1 \cap g = (a_1, b_1)$ für $i = 1$.
- ⇝ Ausgang a) ⇒ $P_n \cap g = \emptyset$, $q_2 = v_1 \in P$.
- ⇝ Ausgang b) ⇒ $P_0 \cap g = (a_0, b_0)$ für $i = 0$

Ausgang a) ⇒ fertig, $P_n \cap g = \emptyset$

Ausgang b) ⇒ Wir müssen Schnittpkte (a_i, b_i) später verfeinern bis (a_k, b_k) gefunden. } Phase 2.

Nun: Übergang von $q_i \to q_{i+1}$ (Fall a)) Bzw. Test, ob Fall b) gilt.

⟶ 2.2.4.3 Laufzeit:

Phase 1: Kandidaten für nächsten Pkt q_{i+1} sind schon genau die Ecken von P_{i+1} die Verfeinerungen der Nachbarkanten von q_i in P_i

Das folgt unmittelbar aus der Konvexität.

- Im Bsp: $i = 0$, $q_0 = v_9$, Kand. für q_1 ist Ecke von P_1 nämlich v_1
  Dies ist die Ecke der Verfeinerungen der Nachbarkanten von $q_0 = v_9$, nämlich der Kanten $(v_9, v_1) \wedge (v_1, v_3)$

⇒ Es muß nur ein Minimum in einer konstanten Zahl von Kandidaten gesucht werden, nämlich in den Verfeinerungen der Kanten, die an q_i in P_i angrenzen.

- Im Bsp: Suche Min. in (v_9, v_1), (v_1, v_3)

Erkennen von Fall b):

Teste in der Kandidatenmenge, ob ein Pkt auf der anderen Seite von g liegt. (Orientation)

⇒ (a_{i+1}, b_{i+1}) in Konst. Zeit denn Konst viele Kand

(Schnittberechnung mit Verfeinerungskanten).

- Im Bsp: $i = 0$, $q_0 = v_9$ ⇒ Kandidatenmenge: (v_9, v_1) und (v_1, v_3)

  v_1 liegt auf der anderen Seite von g, denn $\text{orient}(v_9, v_1, g) > 0$

  ⇒ Teil b) erkannt

  (bez: $\text{orient}(v_9, v_3, g) \leq 0$)

Laufzeit: • $O(1)$ pro Hierarchiestufe
• Haben nur $O(\log n)$ Stufen
$\Rightarrow O(\log n)$ für die erste Phase.

Nach Phase 1 haben wir zwei Fälle:
Fall a) $\Rightarrow$ fertig.
Fall b) $\Rightarrow$ machen weiter zu Phase 2.

Phase 2: Geg: $\{a_i, b_i\} = P \cap g$ für $i < k$ (sonst fertig) und es sind keine Ecken (sonst ebenfalls fertig).

Betrachte a_i (b_i analog):

a_{i+1} muss Schnittpkt sein mit einer Verfeinerungskante von e, wobei $a_i = e \cap g$

Im Bsp: Betrachte a_0, $i = 0 < 2$

a_1 ist Schnpkt mit Verfeinerungskante von e, wobei $a_0 = e \cap g$ und der Geraden g.

Die Verfeinerungskanten von e sind (v_9, v_8) und (v_8, v_7)

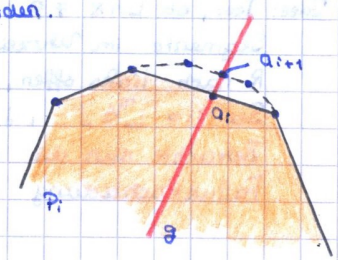
$\Rightarrow a_{i+1}$ kann in Zeit $O(1)$ aus a_i berechnet werden.

(b_{i+1} analog aus b_i)

Weil # Verf.kanten konst. Genauer ≤ 4.

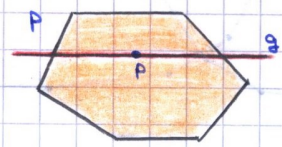
$\Rightarrow$ Laufzeit von Phase 2 auch $O(\log n)$.

$\Rightarrow$ Gesamte Laufzeit also $O(\log n)$!



2.2.5 Satz: Für ein konvexes Polygon P kann:
1) die Hierarchische Darstellung in Zeit $O(n)$ aufgebaut werden.
2) Braucht Platz $O(n)$
3) Für beliebige Gerade g kann $P \cap g$ mit Hilfe der Hierarchischen Darstellung in Zeit $O(\log n)$ Berechnet werden.

2.2.6 Bem.
1) Teil 3) $\Rightarrow$ Alternative $\log n$-Alg. zum Test, ob Pkt $p \in P$ ist.
Idee:



$p \in P \Leftrightarrow p \in P \cap g$
- Betr. Gerade g durch p parallel zu x-Achse
- Berechne hierar. Darst.
- Berechne Schnpkte
- Schaue ob $p_x \in [a_x, b_x]$

$\Rightarrow$ Zeit: $O(1) + O(n) + O(\log n) + O(1) = O(n)$!

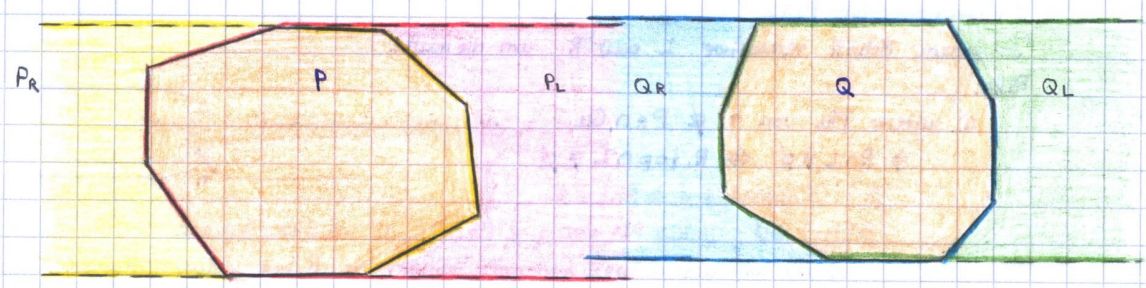
2) Hierar. Darstellung auch im $\mathbb{R}^3$ möglich, d.h. für konvexe Polyeder.
(dazu später mehr...).

2.3 Weiteres Problem auf konvexen Polygonen: Schnitt von zwei konvexen Polygonen.

2.3.1 Ziel: Geg: zwei konvexe Polygone P, Q
Test ob $P \cap Q \neq \emptyset$

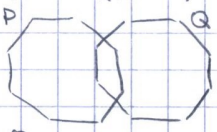
Anm. Wir wissen schon, wie man $P \cap Q$ in Zeit $O(n)$ Berechnet, wobei n = Gesamtzahl der Ecken.

2.3.2 Idee:
1) Betrachte ein einfacheres Problem: Test, ob sich zwei polygonale Ketten schneiden.
- Zerlege P in linken (P_L) & rechten (P_R) Rand durch Zerschneiden an Extremalpkten gemäß yx-Sortierung der Ecken.
- Erweitere durch entsprechende horizontale Strahlen.



Beobachtung: $P \cap Q \neq \emptyset \Leftrightarrow P_L \cap Q_R \neq \emptyset \wedge P_R \cap Q_L \neq \emptyset$

Beweis: $P \cap Q \neq \emptyset \Rightarrow P_R \cap Q_L \neq \emptyset \wedge P_L \cap Q_R \neq \emptyset$ klar.

Für 

mit $P_R \cap Q_L \neq \emptyset \Leftrightarrow P \cap Q \neq \emptyset$
(hierbei gilt auch $P_L \cap Q_R \neq \emptyset$)

Für 

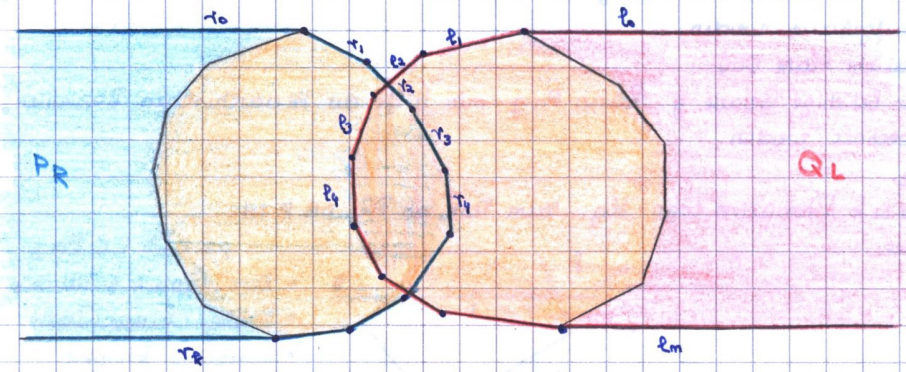
mit $Q_R \cap P_L \neq \emptyset \Leftrightarrow P \cap Q \neq \emptyset$
(hierbei gilt auch $Q_L \cap P_R \neq \emptyset$)

$\Rightarrow$ falls $P_R \cap Q_L \neq \emptyset \wedge P_L \cap Q_R \neq \emptyset \Rightarrow P \cap Q \neq \emptyset$

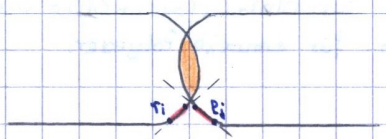
2). Lösung des einfacheren Problems:

Bea: Wir wissen nicht, ob P linkes und Q rechtes Polygon oder umgekehrt.

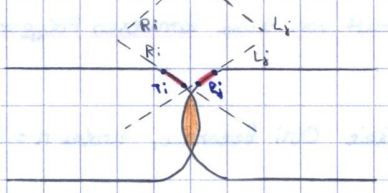
Idee: Test, ob $L \cap R \neq \emptyset$ in Zeit $O(\log n)$, wobei $R = r_0 .. r_k$ gegeben durch die Segmente im Uhrzeigersinn (r_0, r_k Strahlen) und $L = \ell_0 .. \ell_m$ analog gg. Uhrzeigersinn.
R nach links offen, L nach rechts offen.
Sei $i := \lfloor m/2 \rfloor$, $j := \lfloor k/2 \rfloor$
Betrachte nun die mittleren Segmente r_i und ℓ_j.
Fallunterscheidung gemäß der Lage von ℓ_j und r_i auf Geraden R_i und L_j.



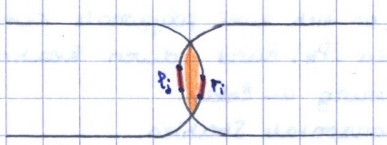
Es gibt nun mehrere Fälle:

1) 

← $P_R \cap Q_L$ oberhalb von den unteren Endpkten von r_i und ℓ_j

2) 

← $P_R \cap Q_L$ unterhalb von den oberen Endpkten von r_i und ℓ_j

3) 

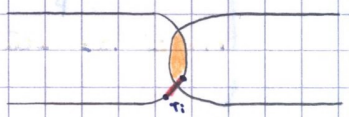
← Endpkte von r_i und $\ell_j \in P_R \cap Q_L$
(Bea: hierbei müssen sich die Polygone nicht notw. schneiden!)

Wir betrachten hier Fall 1

In jedem Schritt reduziere L oder R um die Hälfte.

Fälle:

a). unterer Pkt von $r_i \notin P_R \cap Q_L$



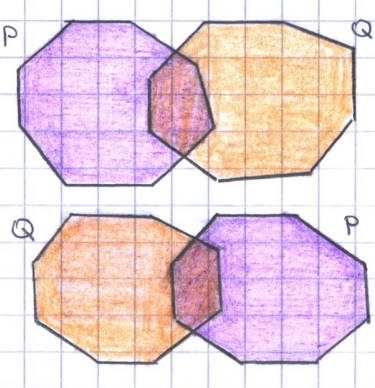
$\Rightarrow P \cap L \neq \emptyset \Leftrightarrow R.top \cap L \neq \emptyset$

"$\Leftarrow$": $R.top \cap L \neq \emptyset \Rightarrow R \cap L \neq \emptyset$

"$\Rightarrow$" $P \cap L \neq \emptyset$, unterer Pkt von $r_i \notin P_R \cap Q_L$, r_i mittleres Segment
$\Rightarrow$ da wir uns im Fall 1 befinden, können wir R.bot rausschmeissen.
$\Rightarrow R.top \cap L \neq \emptyset$.

Beobachtung: $P \cap Q \neq \emptyset \Leftrightarrow P_L \cap Q_R \neq \emptyset \wedge P_R \cap Q_L \neq \emptyset$.

Denn:



Hier: $P \cap Q \neq \emptyset \Leftrightarrow P_R \cap Q_L \neq \emptyset$

Bea: Man weiß in der Regel nicht, welches Polygon rechts und welches links liegt.

Hier: $P \cap Q \neq \emptyset \Leftrightarrow P_L \cap Q_R \neq \emptyset$

$\Rightarrow$ Insgesamt gilt also: falls $P_L \cap Q_R \neq \emptyset \wedge P_R \cap Q_L \neq \emptyset \Rightarrow P \cap Q \neq \emptyset$

„$\Leftarrow$" klar

$\Rightarrow$ die obige Beh.

2) Lösung des einfacheren Problems:

Idee: Test, ob $L \cap R \neq \emptyset$ in Zeit $O(\log n)$, wobei $R = \{r_0 \dots r_k\}$ und $L = \{\ell_0 \dots \ell_m\}$ gegeben durch die Segmente im Uhrzeigersinn (für R) und gegen Uhrzeigersinn (für L).

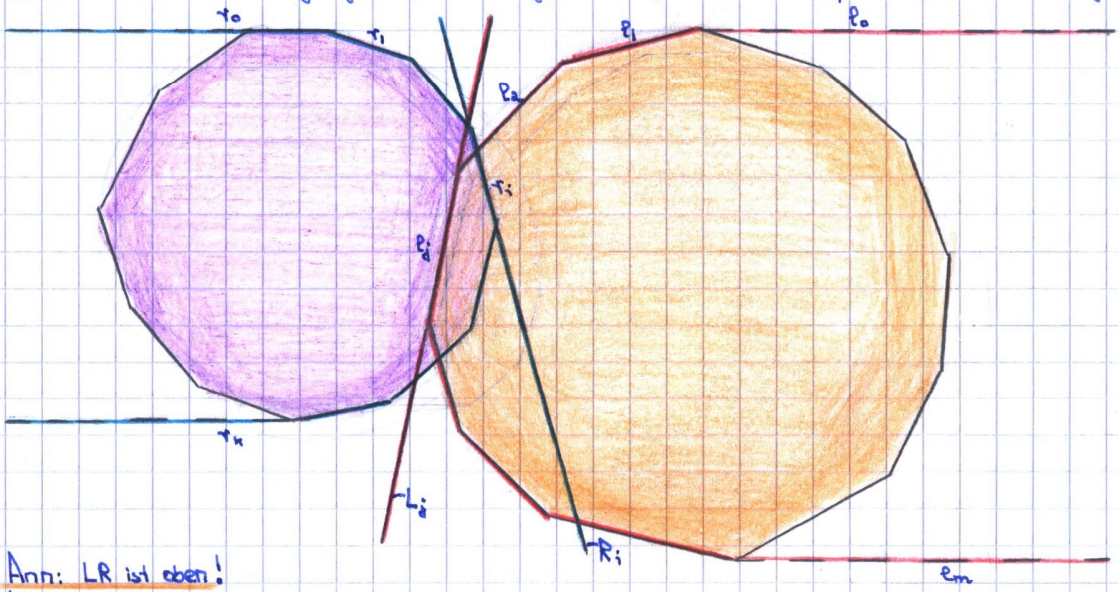
(r_0 und r_k bzw ℓ_0 und ℓ_m sind Strahlen).

R ist nach links offen und L nach rechts offen.

Sei $i := \lfloor m/2 \rfloor$ und $j := \lfloor k/2 \rfloor$

Betrachte nun die mittleren Segmente r_i und ℓ_j

Nun: Fallunterscheidung gemäß der Lage von ℓ_j und r_i auf Geraden R_i und L_j:



Ann: LR ist oben!

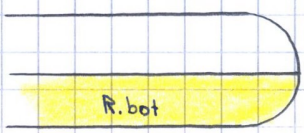
In jedem Schritt reduziere L oder R um die Hälfte.

Fälle:

a) unterer Pkt von r_i ist nicht in LR

$\Rightarrow R \cap L \neq \emptyset \Leftrightarrow R.top \cap L \neq \emptyset$

wobei:



b). unterer Pkt von ℓ_j ist nicht in LR.

$\Rightarrow R \cap L \neq \emptyset \Leftrightarrow R.L.top \neq \emptyset$

c) beide unteren Pkte sind in LR

(i) falls u_{r_i} unterhalb von u_{ℓ_j}

$\Rightarrow R \cap L \neq \emptyset \Leftrightarrow R.top \cap L \neq \emptyset$

(ii) falls u_{ℓ_j} unterhalb von u_{r_i}

$\Rightarrow R \cap L \neq \emptyset \Leftrightarrow R \cap L.top \neq \emptyset$

Hierbei: $u_{r_i} \hat{=}$ unterer Endpkt von r_i

$u_{\ell_j} \hat{=}$ unterer Endpkt von ℓ_j.

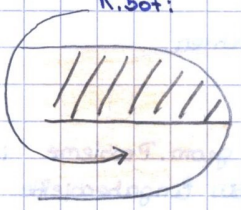
⇒ In Zeit $O(1)$ wird entweder L oder R auf die Hälfte reduziert.
Wiederhole nun bis l_j sich mit r_i schneidet oder bis nur noch konstante Größe von L und R.

2.3.3. Laufzeit: sei $k + m + 2 =: n$
⇒ der eigentliche Test, ob $L \cap R \neq \emptyset$ in Zeit $O(\log n)$.

Kleines Beispiel zur Veranschaulichung:

Bea: R.top:

R.bot:



b) unterer Pkt von l_j nicht $\in L \cap L$
$\Rightarrow R \cap L \neq 0 \Leftrightarrow R \cap L.top \neq \emptyset$
$\Rightarrow$ Man teilt also so weiter bis nur ein Segment übrig bleibt.

c) beide unteren Pkte $\in L \cap L$
$\Rightarrow$ Ketten schneiden sich.

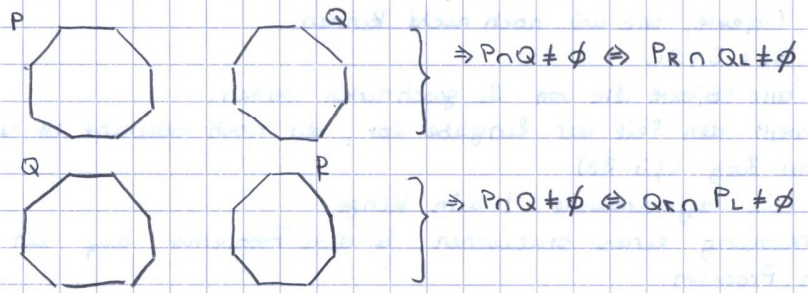
$\Rightarrow$ In Zeit $O(1)$ wird entweder L oder R auf die Hälfte reduziert.

Wiederhole bis r_i sich mit l_j schneidet oder bis nur noch konstante Größe von L und R also bis nur jeweils ein Segment übrig geblieben ist.

2.3.3. Laufzeit:
Sei $k+m =: n$. $\Rightarrow$ Test ob $L \cap R \neq \emptyset$ in Zeit $O(\log n)$.

Zusatz: Bea: $P \cap Q \neq \emptyset \Leftrightarrow P_L \cap Q_R \neq \emptyset \wedge P_R \cap Q_L \neq \emptyset$

Skizze:



$\Rightarrow P \cap Q \neq \emptyset \Leftrightarrow P_R \cap Q_L \neq \emptyset \wedge Q_R \cap P_L \neq \emptyset$

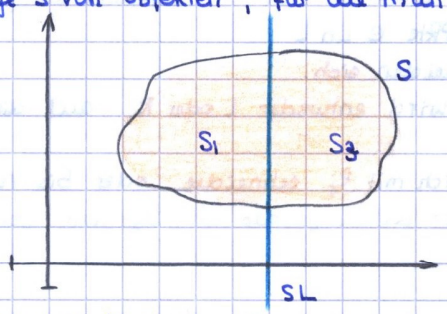
Ferner: $P \cap Q \neq \emptyset \Leftrightarrow L \cap R \neq \emptyset$
$\Rightarrow$ Haben unser Problem auf ein einfacheres zurückgeführt und es mit Zeit $O(\log n)$ gelöst.
Wg. Äquivalenz $\Rightarrow$ Test ob $P \cap Q \neq \emptyset$ in Zeit $O(\log n)$.

Kapitel III: Das Plane Sweep Verfahren.

3.1. Einführung:

3.1.1. Idee:

Allg. Ansatz zur Lsg. geom. Probleme in der Ebene, die inkrementell durch schrittweises Betrachten der Eingabeobjekte in einer bestimmten Reihenfolge gelöst werden können.
d.h. meistens von links nach rechts (xy-Ordnung).
Genauer: Man schiebt / bewegt eine sog. Sweepline, also eine senkrechte Gerade SL über die Menge S von Objekten, für die man ein Problem lösen möchte.



Im Allg. zerlegen wir S in drei Mengen:
- S_1 links von SL
  (Teilproblem für S ist gelöst).
- S_3 rechts von SL
  Objekte, die wir noch nicht kennen.
- $S_2 = S \cap SL$
  alle Objekte die von SL geschnitten werden.

3.1.2. Bem:
S_2 stellt den Teil der Eingabe dar, der noch relevant ist zur Berechnung restliches Lsg (für S_3)
S_2 ist eine dynamische 1-dim. Menge.
⇒ Reduzierung eines statischen 2-dim. Problems auf ein dynamisches 1-dim. Problem.
(die dynam. 1-dim. Probleme werden balancierte Suchbäume sein.).

3.1.3. Bem:
Wir haben spezielle Varianten des SL-Verfahrens kennen gelernt:
- Inkrementelle konv. Hülle
  13.8. Grahams Scan für beide Hüllen gleichzeitig
- Triangulierung.
  Übung.

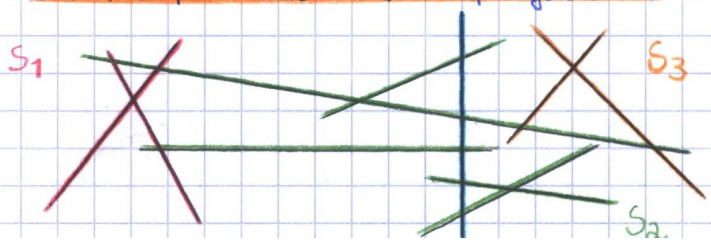
3.2. Line Segment Intersection.
Schnitt von Geradensegmenten / Strecken.

3.2.1. Problem:
Geg: Menge S von n Segmenten
Ges: alle Schnittpkte.
(Bzw. Unterteilung der Ebene in Graph)
Grundoperation: $s_i \cap s_j \quad \forall s_i, s_j \in S$
(dies geht in konst. Zeit O(1) mit orientation-Test).

3.2.2. Triviale Lsg:
Teste alle Paare nacheinander
⇒ Laufzeit $O(n^2)$.

3.2.3. Ziel:
outputsensitiver Algorithmus, d.h. Laufzeit hängt von Zahl der Schnittpkte ab.

3.2.4. Idee für einen Plane Sweep Algorithmus:



3. Das "Plane-Sweep" Verfahren

3.1 Einleitung

- "Plane Sweep":
  - Allg Ansatz zur Lsg geom Probleme in der Ebene.
  - Die Eingabe wird schrittweise in einer bestimmten Reihenfolge (meist von links nach rechts) betrachtet und dabei wird das Problem schrittweise gelöst.
  - Die Eingabemenge S wird in 3 Mengen zerlegt:
    - S_1: links von SL
      Teilproblem ist für S_1 gelöst
    - S_3: rechts von SL
      Objekte die wir noch nicht kennen
    - $S_2 := S \cap SL$
      dynamische 1-dim Menge ⇒ Lösen des Problems für $S_1 \cup (S \cap SL)$

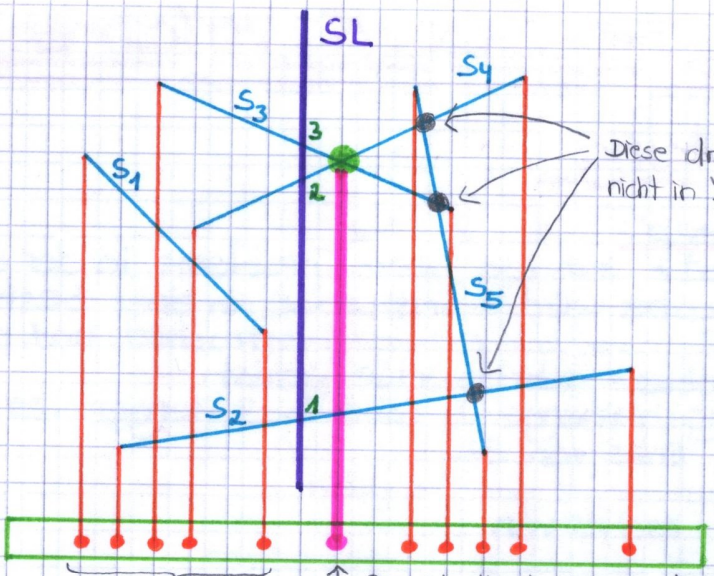
    ⇒Reduzierung eines statischen 2-dim Problems auf ein dynamisches 1-dim.

- Schon mit Plane Sweep berechnet:
  - Inkrementelle konv Hülle ~> Graham's Scan
  - Triangulierung

3.2 Anwendung I: Schnitt von Segmenten

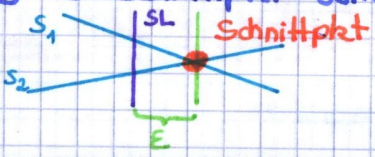
Geg.: Menge S von n Segmenten
Ges.: Alle Schnittpkte

- Trivialer Algorithmus:
  Teste alle Paare ~> $O(n^2)$ schlecht!!
- Ziel:
  Output-sensitiver Algorithmus, dh Laufzeit abhängig von # Schnittpkte
- Da man bei Plane Sweep Verfahren allgemein Events von links nach rechts verarbeitet, benötigt man Datenstrukturen für diese Events: X-Struktur und Y-Struktur
- X-Struktur:
  Verwaltung aller Positionen von SL an denen sich $S \cap SL = S_2$ verändert. (S_2 = Folge der von SL geschnittenen Segmente, sortiert nach y-Koord der Schnittpkte mit SL)
  S_2 verändert sich wenn neue Segmente anfangen, alte aufhören oder wenn man einen Schnittpkt von 2 Segmenten erreicht.
  Diese Stellen nennt man Events
  - Statische X-Struktur: (einfache Liste)
    Dh alle Events sind bekannt zB bei konvexe Hülle, Closest Pair, ...
    X-Struktur = sortierte Liste der Eingabepkte
  - Dynamische X-Struktur: (dynamische Warteschlange)
    Dh Events sind nicht alle bekannt, sie werden (zum Teil) während des Sweeps berechnet. zB Segmentschnitt



- **Y-Struktur:**
  Verwaltung der Menge (Segmente welche aktuell einen) der aktuellen Schnittpkte mit SL haben
  Diese Segmente sollen von unten nach oben (dh gem ihrem Schnittpkt mit SL) entlang der SL sortiert werden.

- Dadurch können Schnitttests auf benachbarte Segmente beschrkt werden, nur diese müssen in X-Struktur enthalten sein.
  ⇒ Jeder Schnittpkt wird E vor ihm erkannt



- **Operationen auf X- bzw Y-Struktur beim Segmentschnitt:**
  - X-Struktur:

X.insert(p)	// p = Event-Pkt	$O(\log n)$
X.delete(p)		$O(\log n)$
X.findmin()	// nächstes Event	$O(1)$

  - Y-Struktur:

Y.insert(s)		$O(\log n)$
Y.delete(s)		$O(\log n)$
Y.swap(S_1, S_2)		$O(1)$
Y.succ(s)	// Nachfolger	$O(1)$
Y.pred(s)	// Vorgänger	$O(1)$

- **Implementierung von X- und Y-Struktur:**
  - balancierter, blattorientierter binärer Baum
  - Für Y-Struktur: zusätzliche Verkettung der Blätter

- **Invariante:**
  X-Struktur enthält zu jedem Zeitpkt:
  - alle Endpkte rechts von SL
  - alle Schnittpkte von in Y benachbarten Segmenten rechts von SL

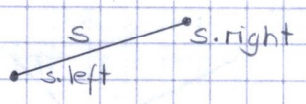
  ⇒ max n-1 Schnittpkte in X-Struktur ($\exists$ n Segmente)
  ⇒ **Platzbedarf: $O(n)$** wir speichern am Anfang alle linken & rechten Endpkte, das sind aber auch nur linear viele

  Ohne Invariante bis zu n^2 Schnittpkte ⇒ Platzbedarf quadratisch, wäre schlecht !!

- **Algorithmus:**

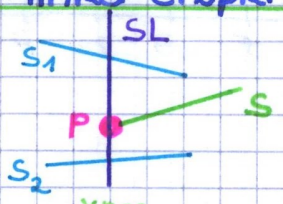
SWEEP(S)
- X-Struktur X ← ∅ — Θ(1)
- Y-Struktur Y ← ∅ — Θ(1)
- double xpos ← −∞ // xpos ist die Position der SL — Θ(1)
- forall se S do — Θ(n log n)
  - X.insert(s.left)
  - X.insert(s.right)

od



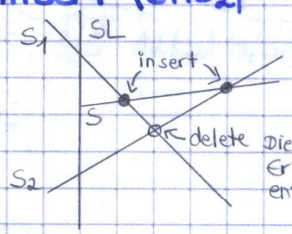
- while (X ≠ ∅) {
  - p ← X.findmin()
  - xpos ← p.xcoord() // schiebt SL zum Pkt p

Fallunterscheidungen gemäß der verschiedenen Events:
  - switch (p) {
  - case linker Endpkt von s: { 6



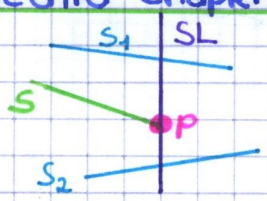
Y.insert(s)
s_1 ← Y.succ(s)
s_2 ← Y.pred(s)
X.delete($s_1 \cap s_2$)
X.insert($s_1 \cap s$)
X.insert($s \cap s_2$)

s_1 exist nicht immer
s_2 ——— " ———
$s_1 \cap s_2$ ——— " ———
$s_1 \cap s$ ——— " ———
$s \cap s_2$ ——— " ———



Dieser Schnittpkt wird jetzt gelöscht um Platz zu sparen. Er wird aber später (wenn die SL nahe genug ist) wieder entdeckt.

}
  - case rechter Endpkt von s: { 4



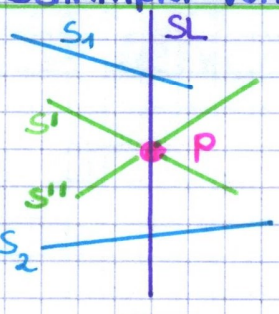
Geht nicht !! Weil man in der Y-Strukt. nur Vorgänger & Nachf. kennt. Und wenn s weg ist, weiß man nicht mehr dass es zw. s_1 und s_2 war, dass man diese beiden jetzt neu "verbinden" muss.

Y.delete(s)
s_1 ← Y.succ(s_2)
[s_2 ← Y.pred(s_1)]

s_1 ← Y.succ(s)
s_2 ← Y.pred(s)
Y.delete(s)
X.insert($s_1 \cap s_2$)

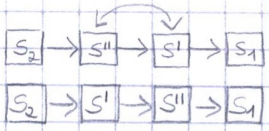
$s_1 \cap s_2$ liegt recht von SL !!

}
  - case Schnittpkt von s' und s'': { 8



BRUNNEN
Papier und viel mehr.

$S_1 \leftarrow Y.succ(s')$
$S_2 \leftarrow Y.pred(s'')$
$Y.swap(s', s'')$
$X.delete(S_1 \cap s')$
$X.delete(S_2 \cap s'')$
$X.insert(S_1 \cap s'')$
$X.insert(S_2 \cap s')$
Ausgabe: "$p = s' \cap s''$"



} end switch } X.delete(p)
end while }

- **Annahmen:**
  - Alle x-Koordinaten von Segment-Anf-, -End- und -Schnittpktn sind paarweise verschieden (d.h. in einem Pkt schneiden sich höchstens 2 Segmente)
  - $\nexists$ vertikalen Segmente

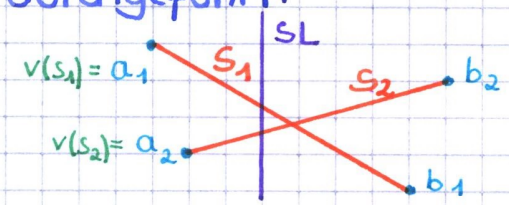
- **Laufzeit:**
  - Alle Operationen auf X und Y benötigen höchstens Zeit $O(\log n)$ (siehe: Operationen auf X bzw Y-Struktur, 2 Seiten vorher)
  - Die Initialisierung vor der while-Schleife benötigt Zeit $O(n \log n)$ Einfügen von 2n Anfangs- bzw Endpktn
  - Die while-Schleife wird $2n+s$ ($s \hat{=}$ # Schnittpkte mal ausgeführt. Jeder Schleifendurchlauf kostet $O(\log n)$ da in jedem Fall konstant viele Operationen auf X und Y und konstant viele Schnitttests ausgeführt werden

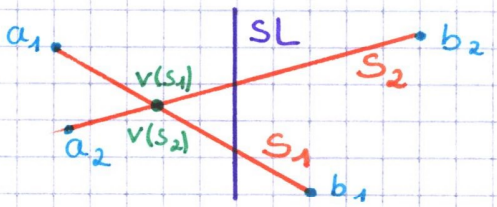
$\Rightarrow$ insgesamt: $O((n+s)\log n) = O(n \cdot \log n + s \cdot \log n)$
Dies ist wie gewünscht output-sensitiv !!

3.2.1 1. Modifikation

orientation-Tests statt "compare" Fkt

- Zur Definition der linearen Ordnung in der Y-Struktur wurde im obigen Algorithmus die "compare"-Fkt verwendet. Anstelle dieser Fkt werden jetzt nur noch orientation-Tests durchgeführt.





Für jedes Segment s in Y werden der letzte Schnittpkt bzw der linke Endpkt als v(s) gespeichert.

1. Fall: orientation $(v(s_1), b_1, v(s_2)) > 0$
$\Rightarrow s_1 < s_2$

2. Fall: orientation $(v(s_1), b_1, v(s_2)) < 0$
$\Rightarrow s_1 > s_2$

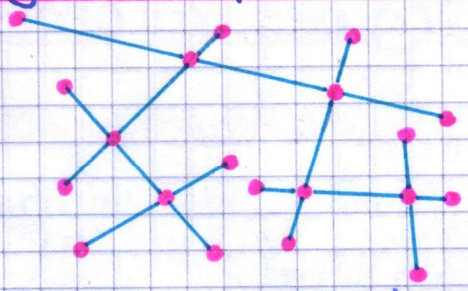
3. Fall: orientation $(v(s_1), b_1, v(s_2)) = 0$
$\Rightarrow v(s_1) = v(s_2)$

$\Rightarrow$ orientation $(v(s_2), b_1, b_2)$ $< 0 \Rightarrow s_1 > s_2$
$= 0 \Rightarrow$ collinear
$> 0 \Rightarrow s_1 < s_2$

BRUNNEN

3.2.2 2. Modifikation Ausgabe des planaren Graphen

- **Ausgabe des planaren Graphen:**



$G = (V, E)$

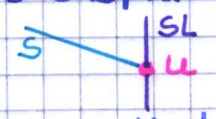
V: Anf-, End- und Schnittpkte

E: Intervalle durch Zerlegung der Segmente durch V

- **Erzeugung von Knoten:**
  - für jeden Anf- und Endpkt der Segmente
  - Für jeden Schnittpkt der Segmente
- **Erzeugung von Kanten:**
  - bei Schnittpkt:

2 neue Kanten entstehen:
$v(s_1) \leftrightarrow u$ und $v(s_2) \leftrightarrow u$ (jeweils 2 fach gerichtet)

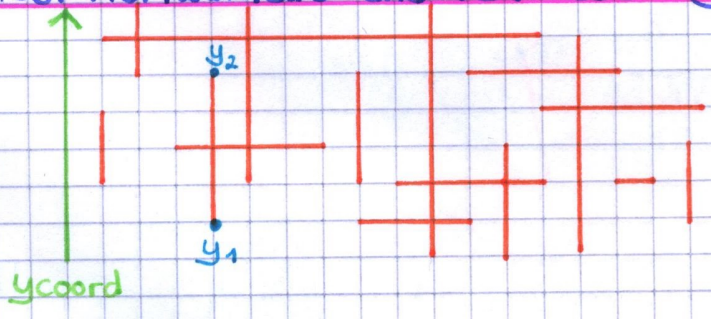
  - bei Endpkt:



1 neue Kante entsteht:
$v(s) \leftrightarrow u$ (2 fach gerichtet)

3.2.3 3. Modifikation nur horizontale und vertikale Segmente

- **Nur horizontale und vertikale Segmente:**



$n = 16$ (# Segmente)
$s = 10$ (# Schnittpkte)
$\ell = 9$ (# vertikale Segmente)

- **Events:**
  - linker Endpkt von Horizontalen
  - rechter ——"——
  - vertikales Segment

} Einfügen und löschen
} Test ob Schnittpkte exist

- **1-dim Bereichsabfrage:** hat immer Zeit $O(\log n + s)$ s = Ausgabe
  Welche horizontalen Segmente in Y drinstehen

$\sum_{i=1}^{\ell} \log n + k_i$ für ℓ vertikale Segmente

Hier: $\log n + 0 + \log n + 1 + \log n + 1 + \log n + 2 + \log n + 0 + \log n + 3 + \log n + 1 + \log n + 2 + \log n + 0 =$
$= 9 \cdot \log n + 10$
$\Rightarrow O(n \cdot \log n + s)$
↑ Hier zwar $9 = \ell \neq n$ aber es könnte ja sein, dass nur vertikale Segmente existieren !!
(dann wäre $\ell = n$)

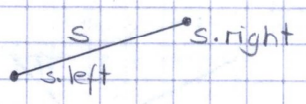
- **Laufzeit:**
  $O(n \cdot \log n + s)$

BRUNNEN Papier und viel mehr.

- **Algorithmus:**

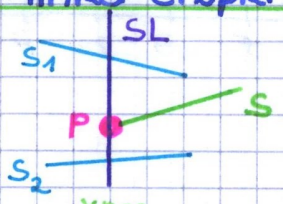
SWEEP(S)
- X-Struktur X ← ∅ — Θ(1)
- Y-Struktur Y ← ∅ — Θ(1)
- double xpos ← −∞ // xpos ist die Position der SL — Θ(1)
- forall se S do — Θ(n log n)
  - X.insert(s.left)
  - X.insert(s.right)

od



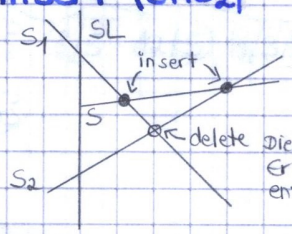
- while (X ≠ ∅) {
  - p ← X.findmin()
  - xpos ← p.xcoord() // schiebt SL zum Pkt p

Fallunterscheidungen gemäß der verschiedenen Events:
  - switch (p) {
  - case linker Endpkt von s: { 6



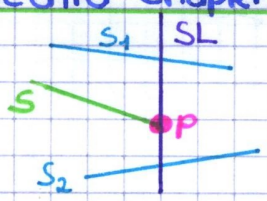
Y.insert(s)
s_1 ← Y.succ(s)
s_2 ← Y.pred(s)
X.delete($s_1 \cap s_2$)
X.insert($s_1 \cap s$)
X.insert($s \cap s_2$)

s_1 exist nicht immer
s_2 ——— " ———
$s_1 \cap s_2$ ——— " ———
$s_1 \cap s$ ——— " ———
$s \cap s_2$ ——— " ———



Dieser Schnittpkt wird jetzt gelöscht um Platz zu sparen. Er wird aber später (wenn die SL nahe genug ist) wieder entdeckt.

}
  - case rechter Endpkt von s: { 4



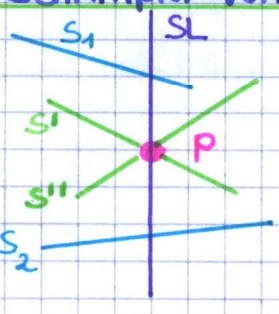
Geht nicht !! Weil man in der Y-Strukt. nur Vorgänger & Nachf. kennt. Und wenn s weg ist, weiß man nicht mehr dass es zw. s_1 und s_2 war, dass man diese beiden jetzt neu "verbinden" muss.

Y.delete(s)
s_1 ← Y.succ(s_2)
[s_2 ← Y.pred(s_1)]

s_1 ← Y.succ(s)
s_2 ← Y.pred(s)
Y.delete(s)
X.insert($s_1 \cap s_2$)

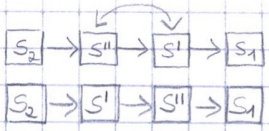
$s_1 \cap s_2$ liegt recht von SL !!

}
  - case Schnittpkt von s' und s'': { 8



BRUNNEN
Papier und viel mehr.

$S_1 \leftarrow Y.succ(s')$
$S_2 \leftarrow Y.pred(s'')$
$Y.swap(s', s'')$
$X.delete(S_1 \cap s')$
$X.delete(S_2 \cap s'')$
$X.insert(S_1 \cap s'')$
$X.insert(S_2 \cap s')$
Ausgabe: "$p = s' \cap s''$"



} end switch } X.delete(p)
end while }

- **Annahmen:**
  - Alle x-Koordinaten von Segment-Anf-, -End- und -Schnittpktn sind paarweise verschieden (d.h. in einem Pkt schneiden sich höchstens 2 Segmente)
  - $\nexists$ vertikalen Segmente

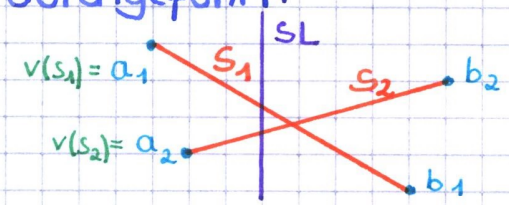
- **Laufzeit:**
  - Alle Operationen auf X und Y benötigen höchstens Zeit $O(\log n)$ (siehe: Operationen auf X bzw Y-Struktur, 2 Seiten vorher)
  - Die Initialisierung vor der while-Schleife benötigt Zeit $O(n \log n)$ Einfügen von 2n Anfangs- bzw Endpktn
  - Die while-Schleife wird $2n+s$ ($s \hat{=}$ # Schnittpkte mal ausgeführt. Jeder Schleifendurchlauf kostet $O(\log n)$ da in jedem Fall konstant viele Operationen auf X und Y und konstant viele Schnitttests ausgeführt werden

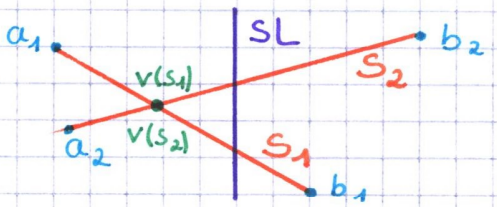
$\Rightarrow$ insgesamt: $O((n+s)\log n) = O(n \cdot \log n + s \cdot \log n)$
Dies ist wie gewünscht output-sensitiv !!

3.2.1 1. Modifikation

orientation-Tests statt "compare" Fkt

- Zur Definition der linearen Ordnung in der Y-Struktur wurde im obigen Algorithmus die "compare"-Fkt verwendet. Anstelle dieser Fkt werden jetzt nur noch orientation-Tests durchgeführt.





Für jedes Segment s in Y werden der letzte Schnittpkt bzw der linke Endpkt als v(s) gespeichert.

1. Fall: orientation $(v(s_1), b_1, v(s_2)) > 0$
$\Rightarrow s_1 < s_2$

2. Fall: orientation $(v(s_1), b_1, v(s_2)) < 0$
$\Rightarrow s_1 > s_2$

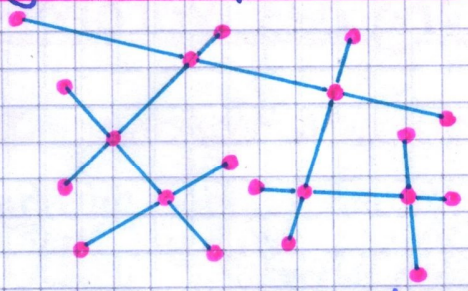
3. Fall: orientation $(v(s_1), b_1, v(s_2)) = 0$
$\Rightarrow v(s_1) = v(s_2)$

$\Rightarrow$ orientation $(v(s_2), b_1, b_2)$ $< 0 \Rightarrow s_1 > s_2$
$= 0 \Rightarrow$ collinear
$> 0 \Rightarrow s_1 < s_2$

BRUNNEN

3.2.2 2. Modifikation Ausgabe des planaren Graphen

- **Ausgabe des planaren Graphen:**



$G = (V, E)$

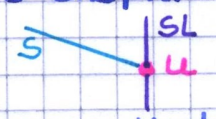
V: Anf-, End- und Schnittpkte

E: Intervalle durch Zerlegung der Segmente durch V

- **Erzeugung von Knoten:**
  - für jeden Anf- und Endpkt der Segmente
  - Für jeden Schnittpkt der Segmente
- **Erzeugung von Kanten:**
  - bei Schnittpkt:

2 neue Kanten entstehen:
$v(s_1) \leftrightarrow u$ und $v(s_2) \leftrightarrow u$ (jeweils 2 fach gerichtet)

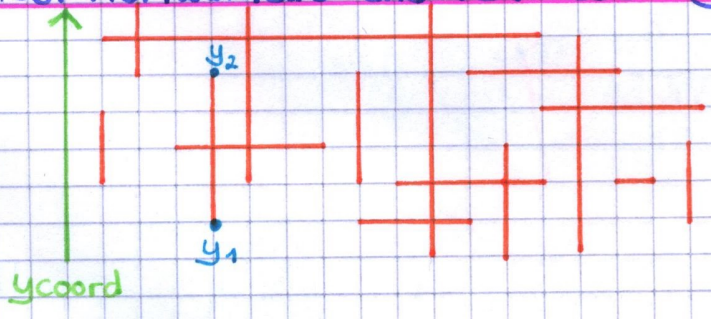
  - bei Endpkt:



1 neue Kante entsteht:
$v(s) \leftrightarrow u$ (2 fach gerichtet)

3.2.3 3. Modifikation nur horizontale und vertikale Segmente

- **Nur horizontale und vertikale Segmente:**



$n = 16$ (# Segmente)
$s = 10$ (# Schnittpkte)
$\ell = 9$ (# vertikale Segmente)

- **Events:**
  - linker Endpkt von Horizontalen
  - rechter ——"——
  - vertikales Segment

} Einfügen und löschen
} Test ob Schnittpkte exist

- **1-dim Bereichsabfrage:** hat immer Zeit $O(\log n + s)$ s = Ausgabe
  Welche horizontalen Segmente in Y drinstehen

$\sum_{i=1}^{\ell} \log n + k_i$ für ℓ vertikale Segmente

Hier: $\log n + 0 + \log n + 1 + \log n + 1 + \log n + 2 + \log n + 0 + \log n + 3 + \log n + 1 + \log n + 2 + \log n + 0 =$
$= 9 \cdot \log n + 10$
$\Rightarrow O(n \cdot \log n + s)$
↑ Hier zwar $9 = \ell \neq n$ aber es könnte ja sein, dass nur vertikale Segmente existieren !!
(dann wäre $\ell = n$)

- **Laufzeit:**
  $O(n \cdot \log n + s)$

BRUNNEN Papier und viel mehr.

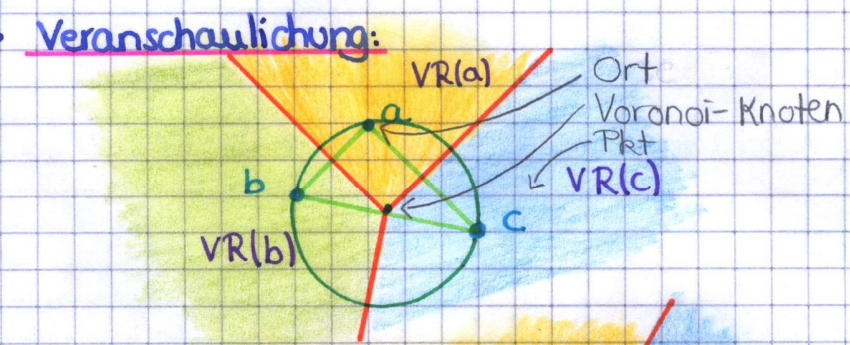
3.3 Anwendung II: Post Office Probleme

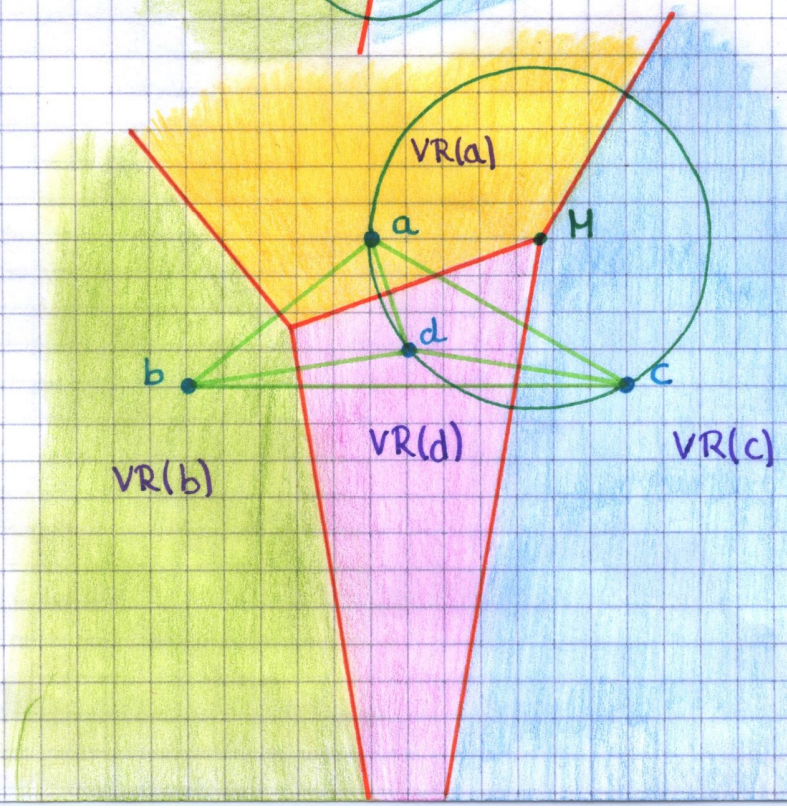
3.3.1 Definition des Voronoi-Diagramms

- Das Voronoi-Diagramm ist eine wichtige geometrische Datenstruktur zur Lsg der sog. "Post Office Probleme"
  Geg.: Menge S von n Orten (z.B. Postämter)
  Ges.: Ort $x \in S$, sodass dist(p,x) minimal für $p \in \mathbb{R}^2$ bel
  ↓
- Idee:
  1.) • Berechne für jeden Ort $x \in S$ das Gebiet aller Pkte, deren Abstand zu x kleiner (gleich) ist, als zu allen anderen Orten $\in S$.
  → planare Unterteilung: Voronoi-Diagramm von S
  2.) • Teste für Eingabepkt $p \in \mathbb{R}^2$ in welchem Gebiet des VD(S) er liegt.
  → Teilproblem: Point Location

- Definition:
  Voronoi-Region für Ort $x \in S$:
  $VR(x) := \{p \in \mathbb{R}^2 : dist(x,p) \leq dist(y,p) \text{ für } x \neq y\}$ $y \in S$, y ort
  $= \bigcap_{y \in S \setminus \{x\}} H(x,y)$
  ⇒ VR(x) ist konvexes Polygon als Schnitt von n-1 Halbebenen

- Veranschaulichung:





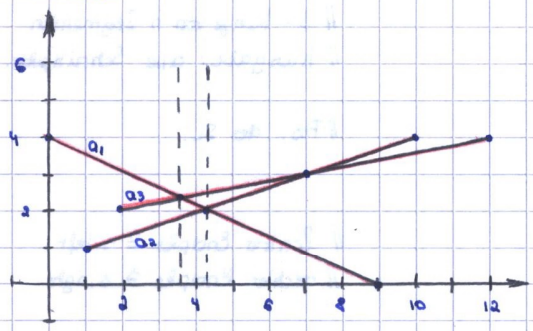
H = Mittelpkt des Umkreises durch a,d,c
= Schnittpkt der Mittelsenkrechten

dualer Graph ≈ hellgrün
(Kreiseigenschaft muss dabei erfüllt sein!!)

planarer Graph = orange
aber auch = hellgrün

BRUNNEN
Papier und viel mehr.

Beispiel zum Alg.:



$X = \{(0,4),(1,1),(2,2),(9,0),(10,4),(12,4)\}$.

$p = (0,4)$, linker Endpkt

$\Rightarrow Y = \{a_1\}$, $s_1 = s_2 = \emptyset$, $X = \{(1,1),(2,2),(9,0),(10,4),(12,4)\}$

$p = (1,1)$, linker Endpkt

$\Rightarrow Y = \{a_2, a_1\}$, $s_1 = a_1$, $s_2 = \emptyset$, $X \leftarrow s_1 \cap s = a_1 \cap a_2 \approx (4.2,2)$

$\Rightarrow X = \{(2,2),(4.2,2),(9,0),(10,4),(12,4)\}$

$p = (2,2)$, linker Endpkt.

$\Rightarrow Y = \{a_2, a_3, a_1\}$, $s_1 = a_1$, $s_2 = a_2$, $X \leftarrow X \setminus (4.2,2)$, $X \leftarrow a_1 \cap a_3 \approx (3.5, 2.5)$, $X \leftarrow a_2 \cap a_3 \approx (7,3)$

$\Rightarrow X = \{(3.5,2.5),(7,3),(9,0),(10,4),(12,4)\}$

$p = (3.5, 2.5)$ Schnittpkt von a_1 und a_3

$\Rightarrow s' = a_1$, $s'' = a_3$, $s_1 = \emptyset$, $s_2 = a_2$, $Y = \{a_2, a_1, a_3\}$

$X \leftarrow X \setminus a_2 \cap a_3$, $X \leftarrow a_1 \cap a_2 \approx (4.2,2)$

$\Rightarrow X = \{(4.2,2),(9,0),(10,4),(12,4)\}$

"Ausgabe: $p = a_1 \cap a_3$"

$p = (4.2,2)$ Schnpkt von a_1 und a_2

$\Rightarrow s' = a_1$, $s'' = a_2$, $s_1 = a_3$, $s_2 = \emptyset$, $Y = \{a_1, a_2, a_3\}$

$X \leftarrow X \setminus a_1 \cap a_3$, $X \leftarrow a_3 \cap a_2 \approx (7,3)$

$\Rightarrow X = \{(7,3),(9,0),(10,4),(12,4)\}$

"Ausgabe: $p = a_1 \cap a_2$"

$p = (7,3)$ Schnpkt von a_3 und a_2

$\Rightarrow s' = a_3$, $s'' = a_2$, $s_1 = \emptyset$, $s_2 = a_1$, $Y = \{a_1, a_3, a_2\}$

$X \leftarrow X \setminus a_1 \cap a_2$ // ist links von SL, also nicht mehr in X

$X \leftarrow a_1 \cap a_3$ // links von SL $\Rightarrow$ nicht einfügen

$\Rightarrow X = \{(9,0),(10,4),(12,4)\}$

"Ausgabe: $p = a_3 \cap a_2$".

$p = (9,0)$ rechter Endpkt

$\Rightarrow s_2 = a_3$, $s_1 = \emptyset$, $Y = \{a_3, a_2\}$

$\Rightarrow X = \{(10,4),(12,4)\}$

$p = (10,4)$ rechter Endpkt

$\Rightarrow s_2 = \emptyset$, $s_1 = a_3$, $Y = \{a_3\}$

$\Rightarrow X = \{(12,4)\}$

$p = (12,4)$ rechter Endpkt

$\Rightarrow s_1 = s_2 = \emptyset$, $Y = \emptyset$, $X = \emptyset$.

3.2.11. Geometrische Primitive (Prädikate Operatoren).

1. Schnitt von zwei Segmenten:
   - Test + Berechnung
2. Lineare Ordnung auf Y-Struktur.
   - abh. von xpos der SL
   - Datenstruktur (binärer blattorientierter Suchbaum)

     verwendet eine Fkt. compare zum Vergleich von zwei Segmenten.

     Suchbaum: lookup(s), compare(s, s')

     Mögl. Implementierung:

     compare(s1, s2) // berechne y-Koordinaten der Schnittpkte der Geradengleichungen v. s1 und s2 mit SL.

     $y_1 \leftarrow f_1(xpos)$

     $y_2 \leftarrow f_2(xpos)$

     wobei $f_1(x) = a_1x + b_1 \wedge f_2(x) = a_2x + b_2$ keine Vertikalen.

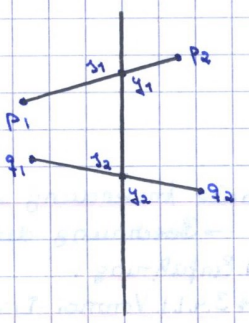
     return compare(y_1, y_2)

     Übung: verwende nur orientation-Aufrufe.

     Lsg: if(orientation(p_1, p_2, s_2) > 0) $\Rightarrow y_1 > y_2$

     if(orientation(q_1, q_2, s_1) > 0) $\Rightarrow y_2 > y_1$

     "=" gibt es nicht ...



3. Lineare Ordnung auf X-Struktur.

   = lexik. xy-Ordnung von Pkten.

   (Vergleich zweier Pkte p und q)

   → compare(p, q)

   $\Delta x \leftarrow$ (q.xcord() − p.xcord());

   if ($\Delta x \neq 0$) then

   return sign(Δx)

   else

   return sign(q.ycord() − p.ycord).

4. Test, ob Pkt p rechts von SL

   trivial

   if(SL, p, p.xcord) < 0 ⇒ rechts von SL

   > 0 links

   = 0 auf.

3.2.12. Bem:

Bisher Segmente in allgemeiner Lage

a) Keine drei Segmente schneiden sich in einem Pkt

B) Alle Eventpkte haben verschiedene x-Koordinaten.

c) Keine vertikalen Segmente.

In der Praxis unrealistisch!

→ Lsg: Modifizierter Alg., der mit degenerierten Eingaben umgehen kann.

→ siehe Übung dazu.

3.2.13. Varianten des Problems:

⟶ 3.2.13.1. Red/Black-Intersection Problem:

Haben zwei Mengen von Segmenten S_1 und S_2

Aufgabe ist: Berechne Schnittpkte sins2 $\forall s_1 \in S_1 \wedge \forall s_2 \in S_2$

⟶ 3.2.13.2. Kurvensegmente:

z.B. Kreisbögen

→ Übung

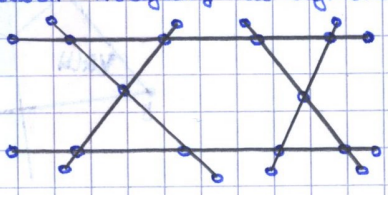
⟶ 3.2.13.3 Berechnung der planaren Unterteilung der Ebene

→ planarer Graph.

G = (V, E)

V = Endpkte und Schnittpkte

E = Intervalle durch Zerlegung der Segmente durch V



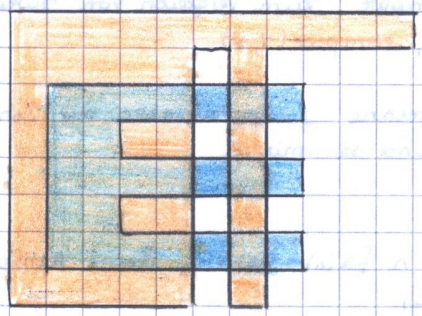
3.3. Erste Anwendung von Plane Sweep:

→ Schnitt von beliebigen Polygonen.

- setzt sich zusammen aus Flächen der planaren Unterteilung.
- Flächen können best. boolesche Operationen zugeordnet werden.

⇝ Bestimmung der planaren Unterteilung

⇝ Übung (Konstruktion des Graphen...)



3.4. Zweite Anwendung von Plane Sweep:

→ Berechnung des Voronoi-Diagramms.

3.4.1. Einführung:

⟼ 3.4.1.1. Voronoi-Diagramm:

- (wichtigste geom. Struktur)
- $\subseteq \mathbb{R}^2$
- Datenstr. zur Lsg. des sog. "Post Office"

⟼ 3.4.1.2. Problem:

Geg: Menge S von n Orten in einer Ebene (zB. Postämter)

Aufgabe: Finde $\forall p \in \mathbb{R}^2$ den Ort $x \in S$, so, dass dist(p,x) minimal.

⟼ 3.4.1.3. Mögl. Variationen:

Orte: Pkte, Segmente, Kreise, Polygone

Abstand: • Euklidischer
- Manhatten-Metrik
- L_∞
- gewichtet

...

Wir betrachten hier pktförmige Orte mit dem eukl. Abstand im $\mathbb{R}^2$.

⟼ 3.4.1.4. Idee:

1) Berechne für jeden Ort $x \in S$ das Gebiet aller Pkte, deren Abstand zu x kleiner oder gleich ist als zu allen anderen Orten.
   → man bekommt eine planare Unterteilung und diese heißt Voronoi-Diagramm von S.
2) Teste für Eingabepkt $p \in \mathbb{R}^2$, in welchem Gebiet des Voronoi-Diagramms er liegt
   → Teilproblem: Point-Location.

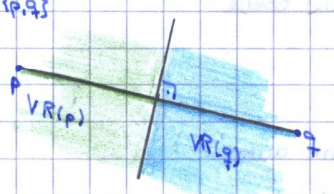
3.4.2. Voronoi-Diagramm:

⟼ 3.4.2.1. Definition: Geg $S = \{x_1 .. x_n\}$.

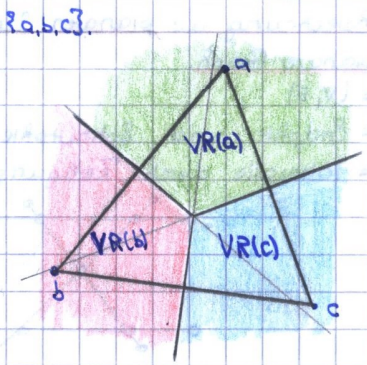
$VR(x_i) := \{p \in \mathbb{R}^2 : dist(p,x_i) \leq dist(p,x_k), \forall x_i \neq x_k\}$ heißt Voronoi-Region für x_i ($i \in \{1..n\}$)

⟼ 3.4.2.2. Beispiele:

a) $S = \{p,q\}$



b) $S = \{a,b,c\}$.

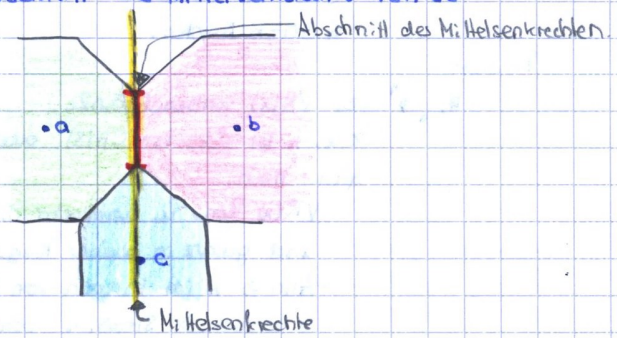


⟶ **3.4.2.3 Bem:** Allgemein für $x,y \in S$ ist
$H(x,y) := \{p \in \mathbb{R}^2 : dist(p,x) \leq dist(p,y)\}$ ein Halbraum bzw. Halbebene definiert durch Mittelsenkrechte.
Dann ist $\forall$ Orte $x \in S$: $VR(x) = \bigcap_{y \in S \setminus \{x\}} H(x,y)$ Schnitt von Halbebenen.

⇒ VR(x) ist konvexes Polygon.

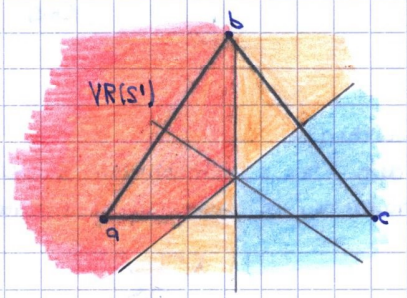
⟶ **3.4.2.4 Definition:** Das Voronoi-Diagramm für eine Menge S von n Orten VD(S) ist die planare Unterteilung der Ebene, die durch die Menge der Voronoi-Regionen VR(x), $x \in S$, definiert wird.
→ Graph dessen Flächen die Voronoi-Regionen darstellen.

⟶ **3.4.2.5 Def:** Voronoi-Knoten:
def. durch mind. drei Orte a, b, c zu denen sie gleichen Abstand haben
⇒ sie sind Mittelpkte von Kreisen durch a, b, c,
Voronoi-Kanten:
def. durch zwei Orte a, b; Abschnitt der Mittelsenkrechte von $\overline{ab}$



⟶ **3.4.2.6 Bem:**
Verallgemeinerung für Mengen $S' \subseteq S$:
$VR(S') = \{p \in \mathbb{R}^2 : dist(p,x) \leq dist(p,y)\ \forall\ x \in S' \wedge y \in S \setminus S'\} =$
$= \bigcap_{\substack{x \in S' \\ y \in S \setminus S'}} H(x,y)$

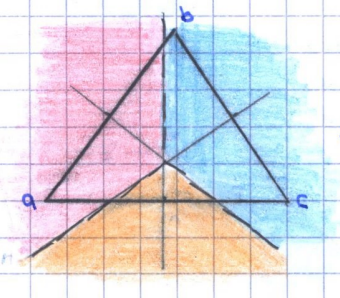
⟶ **3.4.2.7 Bsp:** $S = \{a,b,c\}$, $S' = \{a,b\}$.



$VR(S') = \{p : dist(p,x) \leq dist(p,c), x \in S'\}$

⟶ **3.4.2.8 Def:**
Voronoi-Diagramm der Ordnung $k \leq n$
$VD_k(S) := \{VR(S') : S' \subset S \text{ mit } |S'| = k\}$

⟶ **3.4.2.9 Bsp:**
$S = \{a,b,c\}$
$k = 2$

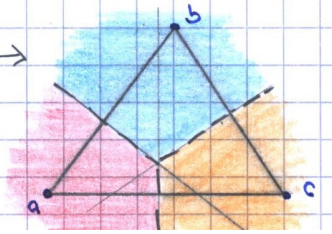


⇒ $VD_2(S) := \{VR(S') : S' \subset \{a,b,c\}, |S'| = 2\}$

← $VD_2(S)$ $VD_2(S) = \{$Mengen $VR(S')\}$

beachte den Unterschied!

Dies ist $VD_1(S)$! →



← VR'en
Jede einzelne $VR = \{$Pkte p : ... $\}$

34.2.10 Spezialfälle:

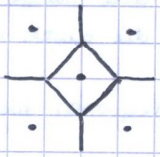
i) "normales" Voronoi-Diagramm: $VD(S) = VD_1(S)$. [closest point VD]

ii) VD_{n-1} [farthest point VD]

↳ Voronoi-Regionen sind hier Mengen von Pkten, die von einem Ort weitest entfernt sind als von allen anderen.

Ab jetzt: $VD(S) = VD_1(S)$

Typisches Bsp:



34.2.11 Lemma:

a) $\forall$ Orte $x \in S$ gilt: $VR(x)$ ist unbeschränkt $\Leftrightarrow$ x ist Ecke von $CH(S)$ oder auf dem Rand von $CH(S)$ (?)

b) Voronoi-Diagramm für n Orte hat:

$\leq 2n-4$ Knoten

$\leq 3n-6$ Kanten.

Beweis:

a) „$\Rightarrow$" sei $VR(x)$ unbeschränkt.

z.z. x ist Ecke von $CH(S)$, oder auf Rand von $CH(S)$

Ann.: nein.

$VR(x)$ konv u. unbeschr n. V. $\Rightarrow \exists$ Strahl s, der in x startet und ganz in dieser Region $VR(x)$ verläuft.

hier notwendig, dass x nicht auf Kante! ↳ Ann $\Rightarrow$ x im Inneren von $CH(S)$

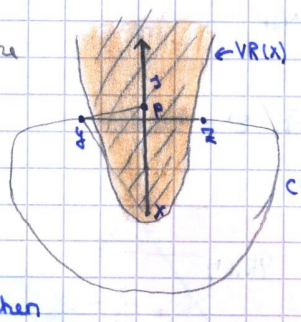
S endlich $\Rightarrow$ Strahl s schneidet den Rand von $CH(S)$ in einer Kante (y,z) oder in Pkt $z=y$. (da $CH(S)$ auch dann endlich)

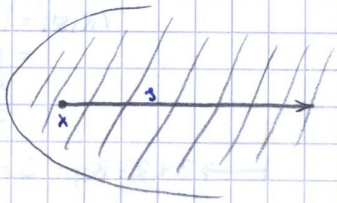
$\exists p \in s$ (genügend weit entfernt von x), der näher zu y oder z ist als zu x

$\Rightarrow \lightning$ zu $p \in VR(x)$

dh. $\lightning$ zu: das orangene ist $VR(x)$

dh. $\lightning$ zu $VR(x)$ ist unbeschränkt.





$\Rightarrow$ Beh

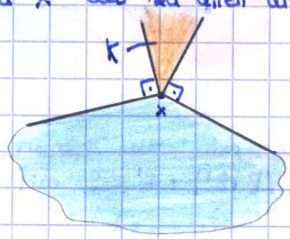
„$\Leftarrow$": Sei nun x Ecke von $CH(S)$.

Betrachte Kegel K zwischen den Senkrechten auf dem zu x betrachteten Kanten.

Alle Pkte in K liegen näher zu x als zu allen anderen Orten. ✓

$\Rightarrow K \subseteq VR(x)$

$\Rightarrow VR(x)$ unbeschränkt (da K unbeschränkt)



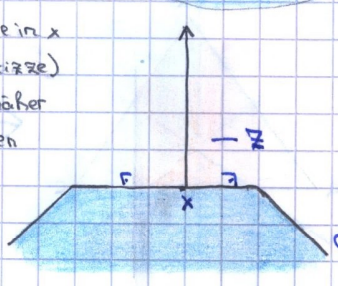
Sei nun x auf dem Rand von $CH(S)$

Betrachte senkrechte Gerade die in x startet und hoch zeigt (wie Skizze)

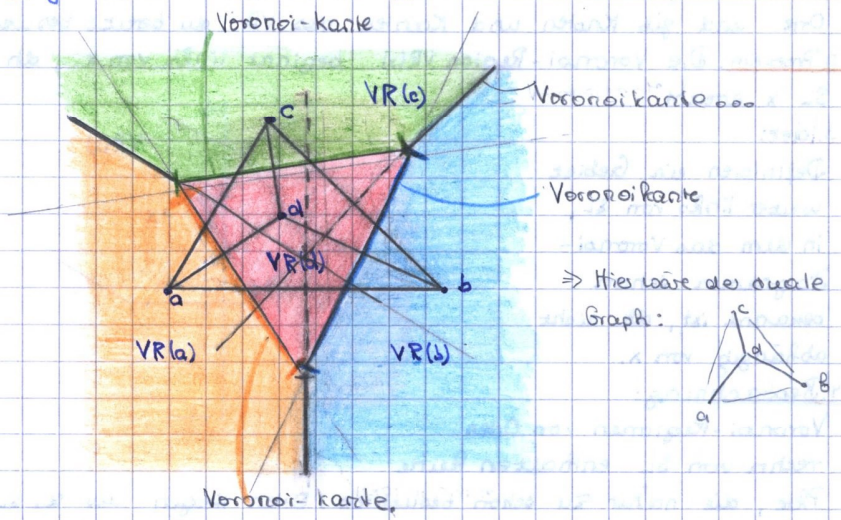
Alle Pkte auf Gerade G liegen näher zu x als zu allen anderen Orten

$\Rightarrow G \subseteq VR(x)$

$\rightarrow VR(x)$ unbeschränkt (da K unbeschränkt)



b). VD ist ein Graph, der $\mathbb{R}^2$ in Voronoi-Regionen unterteilt.
(→ Voronoi-Knoten + Voronoi-Kanten).
Sei $G=(V,E)$ der duale Graph zu VD(S), dh.
$V=S$ (allg. Flächen (Regionen)) und $(x,y)\in E \Leftrightarrow$ VR(x) und VR(y) haben gemeinsame Voronoi-Kante.



- G ist ein planarer Graph und jede Kante $e\in E$ entspricht genau einer Voronoi-Kante (siehe Konstruktion von G).
(cd) entspricht grün, (a,d) entspricht orange und (b,d) entspricht blau.
Graphentheorie ⇒ Jeder planare Graph mit n Knoten hat maximal $3n-6$ Kanten.
⇒ Voronoi-Kanten $\leq 3n-6$.
Außerdem hat jeder Voronoi-Knoten mind. Grad 3 (da er durch mind. drei Orte definiert wird, von denen er gleiche Distanz hat).

$\forall v\in V$: Grad(v) ≥ 3
$\Rightarrow \forall v\in V, |v|=1$: Grad(v) ≥ 3
$\Rightarrow \forall v\in V$: $3\cdot|v| \leq$ Grad(v)
$\Rightarrow \sum_{v\in V} 3\cdot|v| \leq \sum_{v\in V}$ Grad(v)
$\Rightarrow 3\cdot$ #Knoten $\leq \sum_{v\in V}$ Grad(v)
$\Rightarrow$ & gilt: $\sum_{v\in V}$ grad(v) $\geq 3\cdot$ # V Knoten $\wedge \sum_{v\in V}$ grad(v) $= 2\cdot|E|$

$\Rightarrow 3\cdot$ # V Knoten $\leq \sum$ grad(v) $= 2\cdot|E| \leq 2\cdot(3n-6) = 6n-12$.
$\Rightarrow$ # V Knoten $\leq 2n-4$.

→ 3.4.2.12. Bemerkungen:

1). VD für n Orte hat lineare Größe in n.
2). Sei $G=(V,E)$ dualer Graph zu VD(S), dh. $V=S$ und $(x,y)\in E \Leftrightarrow$ VR(x) und VR(y) haben gemeinsame Voronoi-Kante. ∨
G heißt Delaunay-Triangulierung.
Eigenschaften: Jeder Umkreis eines Dreiecks enthält keinen Ort in seinem Inneren.

Bew: Gemeinsame Rand von zwei VR'nen heißt Voronoikante
Und Endpkte von Voronoi-kanten sind die Voronoiknoten
$\Rightarrow$ v Vor. knoten $\Rightarrow \exists$ Orte a,b,c mit v Mittelpkt von Kreis durch a,b,c
$\not\Rightarrow$!

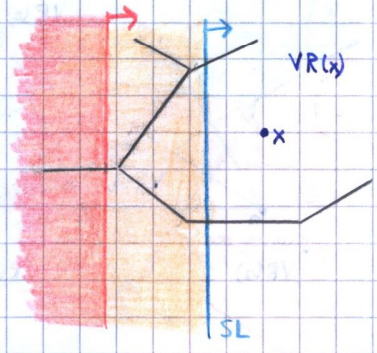
→ Besser: andere Definition als 3.4.2.5!

3.4.3. Konstruktion von Voronoi-Diagrammen durch einen "Plane Sweep Algorithmus".

Zunächst: allg. Lage der Orte
- Keine 4 Orte liegen auf einem gemeinsamen Kreis
- paarweise verschiedene x-Koordinaten aller Orte und Events.

→ 3.4.3.1. Ziel: Bewege Senkrechte SL von links nach rechts über die Menge der Orte und gib Knoten und Kanten des VD der bereits besuchten Orte aus.

→ 3.4.3.2 Problem: Die Voronoi-Region VR(x) beginnt links von x, dh. bevor SL x erreicht.



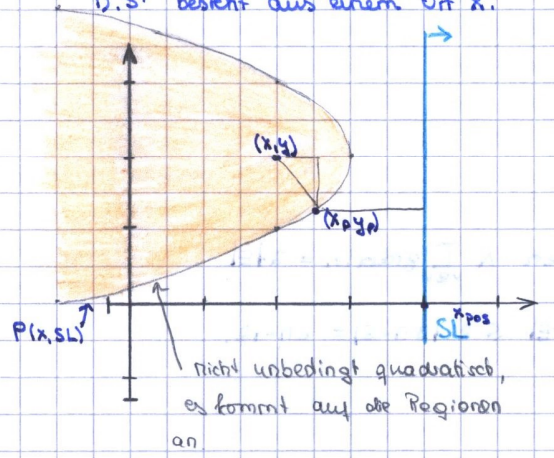
→ 3.4.3.3. Idee:
Definieren ein Gebiet weiter links von SL, in dem das Voronoi-Diagramm schon bekannt ist, dh. nicht abhängig von x.

→ 3.4.3.4. Beobachtung:
Voronoi-Regionen von Orten rechts von SL enthalten keine Pkte, die näher zu schon besuchten Orten liegen als zu SL
Denn sonst würde ein solcher Pkt (unabh. von SL) näher zu einem anderen Ort liegen.
Abstand zu SL ist nicht größer als Abstand zum nächsten Ort.

→ 3.4.3.5 Frage:
Betrachte das Gebiet $L := \{p \in \mathbb{R}^2 : dist(p,x) \leq dist(p,SL)$ für einen schon besuchten Ort $x\}$
⇒ Voronoi-Diagramm von S im Gebiet L hängt nicht von Orten rechts von SL ab.
Welche Form hat L?

→ 3.4.3.6. Beispiele: S' ← Menge der besuchten Orte.

1). S' besteht aus einem Ort x.

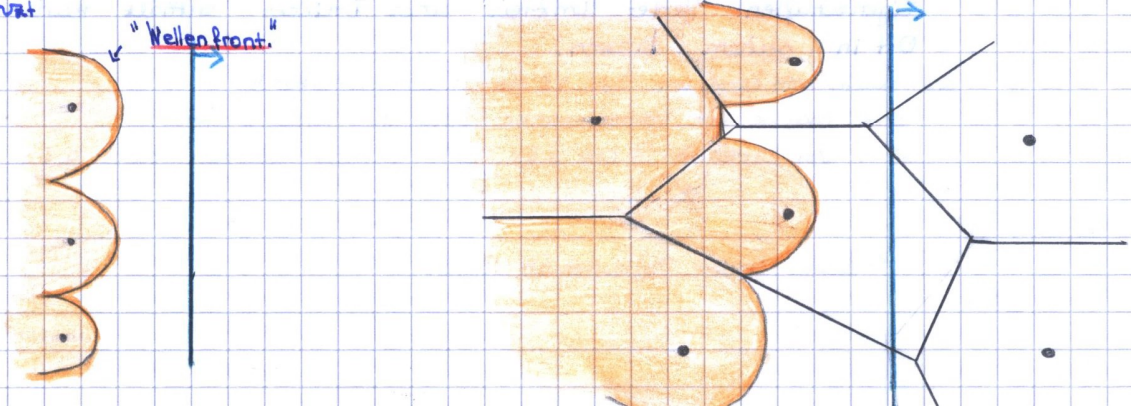


nicht unbedingt quadratisch, es kommt auf die Regionen an

(x,y) und SL bekannt
Wie sieht L aus?
→ Es muß gelten $\forall (x_p y_p) \in L$:
$(x_p - x_{pos})^2 \overset{!}{=} (x_p - x)^2 + (y_p - y)^2$
weil es muß eigentlich gelten:
$x_p - x_{pos} = \sqrt{(x_p - x)^2 + (y_p - y)^2}$.
→ (x,y) = (2,2), SL ≡ 4 bekannt
$\Rightarrow (x_p - 4)^2 = (x_p - 2)^2 + (y_p - 2)^2$
$\Leftrightarrow x_p^2 - 8x_p + 16 = x_p^2 - 4x_p + 4 + y_p^2 - 4y_p + 4$
$\Leftrightarrow -4x_p + 8 = y_p^2 - 4y_p$
$\Leftrightarrow 4x_p = -y_p^2 + 4y_p + 8$
$\Leftrightarrow x_p = -\frac{1}{4}y_p^2 + y_p + 2$
→ dies ist eine Parabelgleichung mit der Variablen y

→ Inneres der Parabel P(x,SL) ist die Menge der Pkte mit gleicher Distanz zu x und SL

2). Im Allgemeinen wird L durch eine Folge von Parabelbögen nach rechts begrenzt



34.2.10 Spezialfälle:

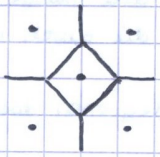
i) "normales" Voronoi-Diagramm: $VD(S) = VD_1(S)$. [closest point VD]

ii) VD_{n-1} [farthest point VD]

↳ Voronoi-Regionen sind hier Mengen von Pkten, die von einem Ort weitest entfernt sind als von allen anderen.

Ab jetzt: $VD(S) = VD_1(S)$

Typisches Bsp:



34.2.11 Lemma:

a) $\forall$ Orte $x \in S$ gilt: $VR(x)$ ist unbeschränkt $\Leftrightarrow$ x ist Ecke von $CH(S)$ oder auf dem Rand von $CH(S)$ (?)

b) Voronoi-Diagramm für n Orte hat:

$\leq 2n-4$ Knoten

$\leq 3n-6$ Kanten.

Beweis:

a) „$\Rightarrow$" sei $VR(x)$ unbeschränkt.

z.z. x ist Ecke von $CH(S)$, oder auf Rand von $CH(S)$

Ann.: nein.

$VR(x)$ konv u. unbeschr n. V. $\Rightarrow \exists$ Strahl s, der in x startet und ganz in dieser Region $VR(x)$ verläuft.

hier notwendig, dass x nicht auf Kante! ↳ Ann $\Rightarrow$ x im Inneren von $CH(S)$

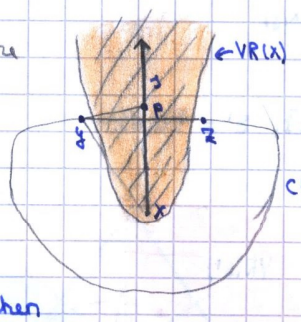
S endlich $\Rightarrow$ Strahl s schneidet den Rand von $CH(S)$ in einer Kante (y,z) oder in Pkt $z=y$. (da $CH(S)$ auch dann endlich)

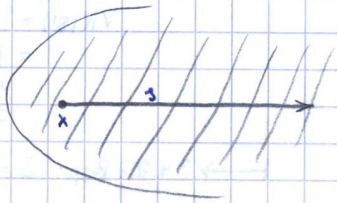
$\exists p \in s$ (genügend weit entfernt von x), der näher zu y oder z ist als zu x

$\Rightarrow \lightning$ zu $p \in VR(x)$

dh. $\lightning$ zu: das orangene ist $VR(x)$

dh. $\lightning$ zu $VR(x)$ ist unbeschränkt.





$\Rightarrow$ Beh

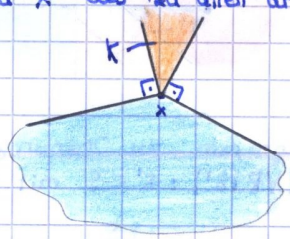
„$\Leftarrow$": Sei nun x Ecke von $CH(S)$.

Betrachte Kegel K zwischen den Senkrechten auf dem zu x betrachteten Kanten.

Alle Pkte in K liegen näher zu x als zu allen anderen Orten. ✓

$\Rightarrow K \subseteq VR(x)$

$\Rightarrow VR(x)$ unbeschränkt (da K unbeschränkt)



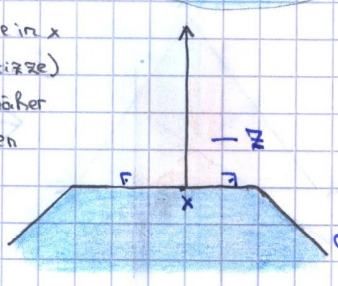
Sei nun x auf dem Rand von $CH(S)$

Betrachte senkrechte Gerade die in x startet und hoch zeigt (wie Skizze)

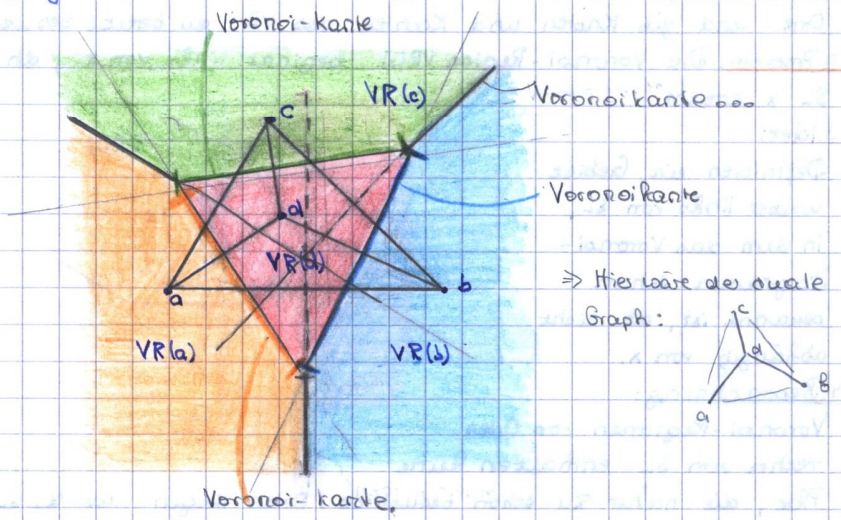
Alle Pkte auf Gerade G liegen näher zu x als zu allen anderen Orten

$\Rightarrow G \subseteq VR(x)$

$\rightarrow VR(x)$ unbeschränkt (da K unbeschränkt)



b). VD ist ein Graph, der $\mathbb{R}^2$ in Voronoi-Regionen unterteilt.
(→ Voronoi-Knoten + Voronoi-Kanten).
Sei $G=(V,E)$ der duale Graph zu VD(S), dh.
$V=S$ (allg. Flächen (Regionen)) und $(x,y)\in E \Leftrightarrow$ VR(x) und VR(y) haben gemeinsame Voronoi-Kante.



- G ist ein planarer Graph und jede Kante $e\in E$ entspricht genau einer Voronoi-Kante (siehe Konstruktion von G).
(cd) entspricht grün, (a,d) entspricht orange und (b,d) entspricht blau.
Graphentheorie ⇒ Jeder planare Graph mit n Knoten hat maximal $3n-6$ Kanten.
⇒ Voronoi-Kanten $\leq 3n-6$.
Außerdem hat jeder Voronoi-Knoten mind. Grad 3 (da er durch mind. drei Orte definiert wird, von denen er gleiche Distanz hat).

$\forall v\in V$: Grad(v) ≥ 3
$\Rightarrow \forall v\in V, |v|=1$: Grad(v) ≥ 3
$\Rightarrow \forall v\in V$: $3\cdot|v| \leq$ Grad(v)
$\Rightarrow \sum_{v\in V} 3\cdot|v| \leq \sum_{v\in V}$ Grad(v)
$\Rightarrow 3\cdot$ #Knoten $\leq \sum_{v\in V}$ Grad(v)
$\Rightarrow$ & gilt: $\sum_{v\in V}$ grad(v) $\geq 3\cdot$ # V Knoten $\wedge \sum_{v\in V}$ grad(v) $= 2\cdot|E|$

$\Rightarrow 3\cdot$ # V Knoten $\leq \sum$ grad(v) $= 2\cdot|E| \leq 2\cdot(3n-6) = 6n-12$.
$\Rightarrow$ # V Knoten $\leq 2n-4$.

→ 3.4.2.12. Bemerkungen:

1). VD für n Orte hat lineare Größe in n.
2). Sei $G=(V,E)$ dualer Graph zu VD(S), dh. $V=S$ und $(x,y)\in E \Leftrightarrow$ VR(x) und VR(y) haben gemeinsame Voronoi-Kante. ∨
G heißt Delaunay-Triangulierung.
Eigenschaften: Jeder Umkreis eines Dreiecks enthält keinen Ort in seinem Inneren.

Bew: Gemeinsame Rand von zwei VR'nen heißt Voronoikante
Und Endpkte von Voronoi-kanten sind die Voronoiknoten
$\Rightarrow$ v Vor. knoten $\Rightarrow \exists$ Orte a,b,c mit v Mittelpkt von Kreis durch a,b,c
$\not\Rightarrow$!

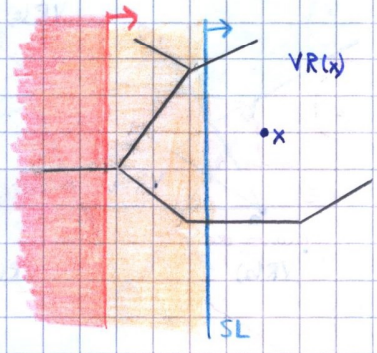
→ Besser: andere Definition als 3.4.2.5!

3.4.3. Konstruktion von Voronoi-Diagrammen durch einen "Plane Sweep Algorithmus".

Zunächst: allg. Lage der Orte
- Keine 4 Orte liegen auf einem gemeinsamen Kreis
- paarweise verschiedene x-Koordinaten aller Orte und Events.

→ 3.4.3.1. Ziel: Bewege Senkrechte SL von links nach rechts über die Menge der Orte und gib Knoten und Kanten des VD der bereits besuchten Orte aus.

→ 3.4.3.2 Problem: Die Voronoi-Region VR(x) beginnt links von x, dh. bevor SL x erreicht.



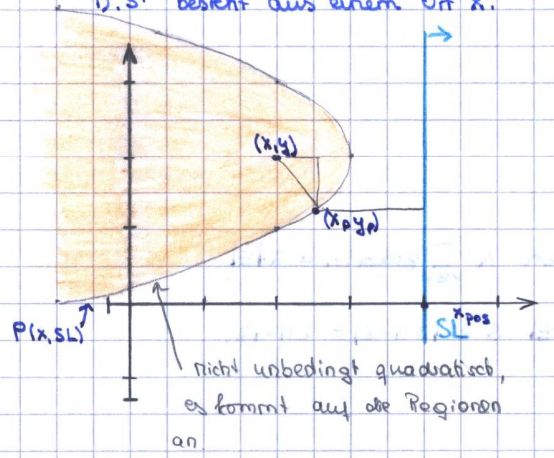
→ 3.4.3.3. Idee:
Definieren ein Gebiet weiter links von SL, in dem das Voronoi-Diagramm schon bekannt ist, dh. nicht abhängig von x.

→ 3.4.3.4. Beobachtung:
Voronoi-Regionen von Orten rechts von SL enthalten keine Pkte, die näher zu schon besuchten Orten liegen als zu SL
Denn sonst würde ein solcher Pkt (unabh. von SL) näher zu einem anderen Ort liegen.
Abstand zu SL ist nicht größer als Abstand zum nächsten Ort.

→ 3.4.3.5 Frage:
Betrachte das Gebiet $L := \{p \in \mathbb{R}^2 : dist(p,x) \leq dist(p,SL)$ für einen schon besuchten Ort $x\}$
⇒ Voronoi-Diagramm von S im Gebiet L hängt nicht von Orten rechts von SL ab.
Welche Form hat L?

→ 3.4.3.6. Beispiele: S' ← Menge der besuchten Orte.

1). S' besteht aus einem Ort x.

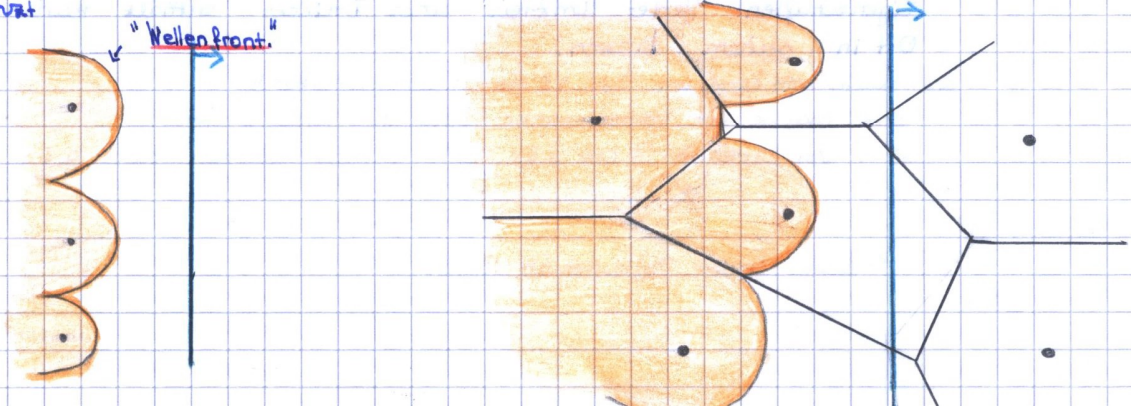


nicht unbedingt quadratisch, es kommt auf die Regionen an

(x,y) und SL bekannt
Wie sieht L aus?
→ Es muß gelten $\forall (x_p y_p) \in L$:
$(x_p - x_{pos})^2 \overset{!}{=} (x_p - x)^2 + (y_p - y)^2$
weil es muß eigentlich gelten:
$x_p - x_{pos} = \sqrt{(x_p - x)^2 + (y_p - y)^2}$.
→ (x,y) = (2,2), SL ≡ 4 bekannt
$\Rightarrow (x_p - 4)^2 = (x_p - 2)^2 + (y_p - 2)^2$
$\Leftrightarrow x_p^2 - 8x_p + 16 = x_p^2 - 4x_p + 4 + y_p^2 - 4y_p + 4$
$\Leftrightarrow -4x_p + 8 = y_p^2 - 4y_p$
$\Leftrightarrow 4x_p = -y_p^2 + 4y_p + 8$
$\Leftrightarrow x_p = -\frac{1}{4}y_p^2 + y_p + 2$
→ dies ist eine Parabelgleichung mit der Variablen y

→ Inneres der Parabel P(x,SL) ist die Menge der Pkte mit gleicher Distanz zu x und SL

2). Im Allgemeinen wird L durch eine Folge von Parabelbögen nach rechts begrenzt



⟶ 3.4.3.7 Beobachtung: Schnittpkte von benachbarten Parabeln in der Wellenfront überstreichen die Kanten von VD(S).

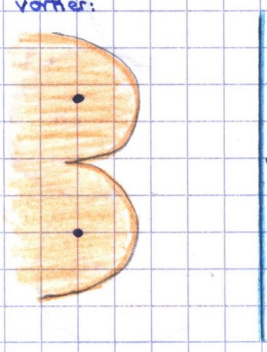
⟶ 3.4.3.8 Idee: Verwalte die Wellenfront der Parabelbögen in einer Y-Struktur.

⟶ 3.4.3.9 Fragen: An welchen Positionen (Transaktionspkten / Events) der SL ändert sich die Y-Struktur strukturell?
→ Welche Events gibt es?

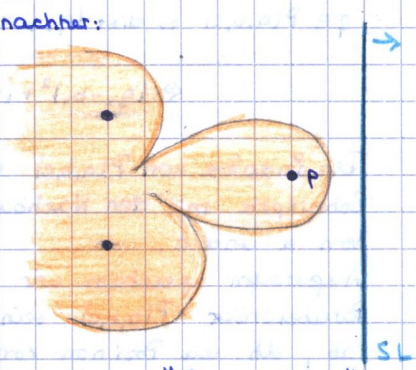
⟶ 3.4.3.10 Zwei Arten von Events:

1). SL überstreicht neuen Ort p "Site Event"

vorher:

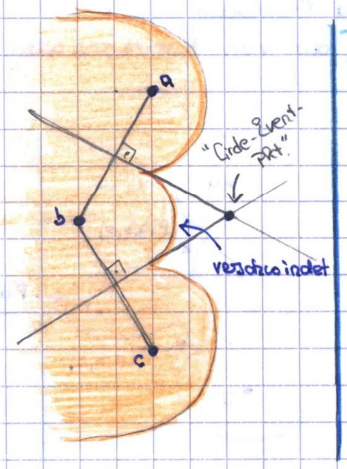


nachher:



2). Ein Parabelbogen verschwindet aus der Wellenfront. "Circle Event"

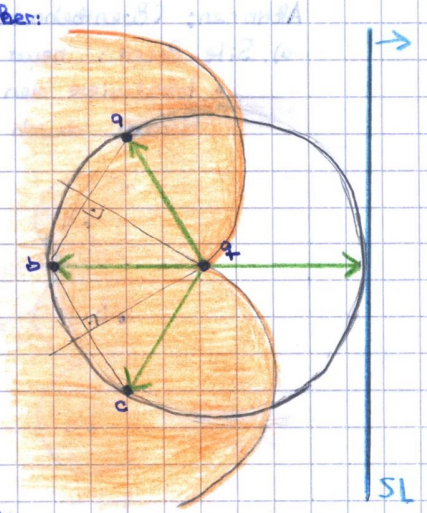
vorher:



? drei Parabeln schneiden sich in einem Pkt

bea: q gehört nicht zu S(1,a) Es ist nur ein Schnpkt von VR'en und zwar Circle-Eventpkt.

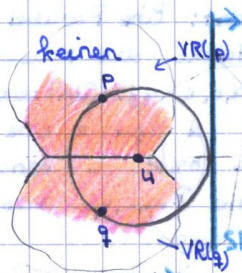
nachher:



↑ Hier entsteht ein neuer Voronoi-Knoten q = Mittelpkt des Kreises durch a,b,c
SL ist zu diesem Zeitpkt Tangente an den Kreis (durch a,b,c).

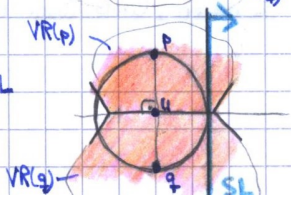
⟶ 3.4.3.11 Lemma: Jeder Pkt auf der Kante von VD(S) kommt während des Sweep als Schnittpkt zweier in der Y-Struktur benachbarter Parabelbögen vor.
(Bei kontinuierlicher Sweep)

Beweis:
Sei e beliebige Voronoi-Kante von VD(S) und u ein beliebiger Pkt auf e. e trennt zwei Voronoi-Regionen VR(p) und VR(q)
u liegt näher zu p und q (gleiche Distanz d) als zu allen anderen Orten.
Kreismittelpkt u enthält im Inneren keinen Ort.
Nun betrachte den Zeitpkt des Sweep, in dem SL rechte Tangente an diesem Kreis K ist



a) u hat den gleichen Abstand zu p und SL
⇒ u liegt auf Parabel von p

b) u hat gl. Abstand zu q und SL
⇒ u liegt auf Parabel von q



⇒ Beh.

3.4.3.12. Implementierungsdetails:

1). Darstellung der Parabelbögen:

Ein Parabelstück wird durch drei Orte a,b,c und die aktuelle Position x_{pos} der Sweepline definiert.

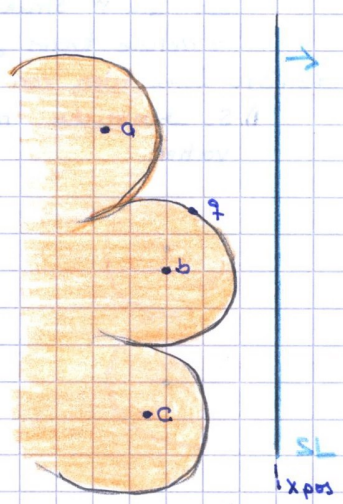
$P(a,b,c)$ bezeichnet Parabelstück von b, das an Parabeln von Ort a und von Ort c grenzt.

Gleichung der Parabel:

$q \in P(a,b,c) \Leftrightarrow dist(q,b) = dist(q,SL)$

auf Parabelstück, gemeint auf dem Rand $\Rightarrow$ "$=$"

$\Leftrightarrow (q_x - b_x)^2 + (q_y - b_y)^2 = (q_x - x_{pos})^2$



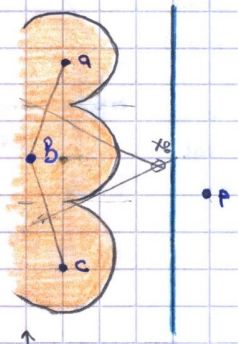
Die Endpkte von $P(a,b,c)$ ergeben sich als Schnittpkte mit den Nachbarparabeln, dh. von a und c.

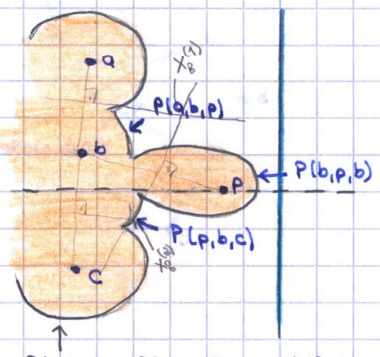
Außerdem speichern wir mit jedem Parabelstück $P(a,b,c)$ sein Circle-Event ab, dh. die Position von SL, bei der $P(a,b,c)$ aus Wellenfront verschwindet.

Aktionen: (Eventbehandlung).

a). Site-Event: neuer Ort p erreicht. i.d. Y-Struktur

- lokalisiere den Parabelbogen $P(a,b,c)$, der von horizontaler Geraden durch p geschnitten wird.





Y-Str.: In der Y-Str. $= \{P(-,a,b), P(a,b,c), P(b,c,+)\}$

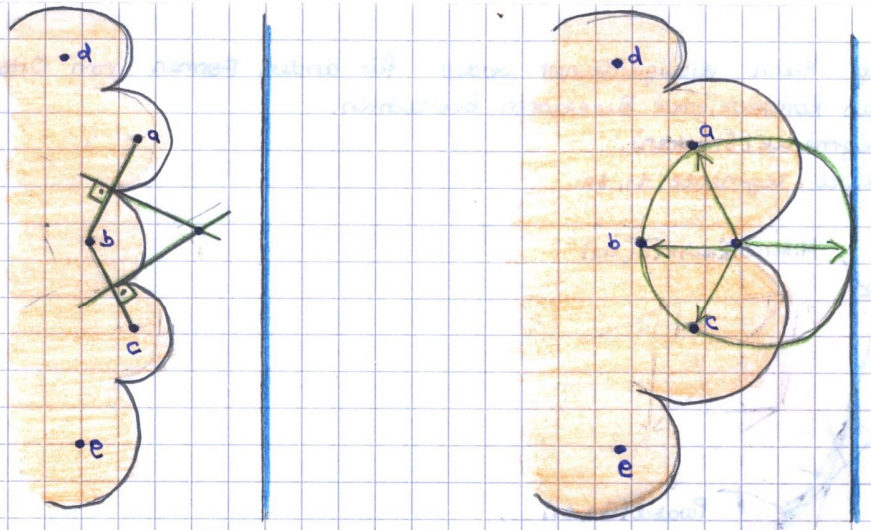
$P(a,b,c)$ löschen aus Y-Str. und drei neue Bögen in die Y-Str. einfügen

$\Rightarrow$ Y-Str. $= \{P(-,a,b), P(a,b,p), P(b,p,b), P(p,b,c), P(b,c,+)\}$

- lösche $P(a,b,c)$ aus Y-Str.
- lösche Circle-Event von $P(a,b,c)$ aus X-Str. (PRkt)
- füge 3 neue Bögen in die Y-Str. ein
- berechne die Circle-Events der neuen Parabelstücke (PRkt) und füge sie in die X-Str. ein.

b). Circle-Event: Eine Parabel $P(a,b,c)$ verschwindet.

- Gib Mittelpkte des Kreises durch a,b,c als Voronoi-Knoten aus mit a,b,c gg. Uhrzeigersinn. (berechne zugleich Mittelsenkrechte von ab und bc und gebe sie aus)
- Seien d und e Orte mit Parabelbögen, die mit Parabel von a bzw. c benachbart sind.
- Entferne $P(a,b,c)$ aus Y-Struktur. und lösche den bearbeiteten Circle-Event-Pkt aus der X-Struktur
- berechne die neuen Circle-Event-Pkte



→ $P(d,a,b) \rightsquigarrow P(d,a,c)$ und $P(b,c,e) \rightsquigarrow P(a,c,e)$.

$X = \{C(d,a,b), C(a,b,c), C(b,c,e)\}$ $X = \{C(d,a,c), C(a,c,e)\}$.

2) Initialisierung

3). Geometrische Primitive

a) Schnitt von Parabeln

b) Vergleichsoptt für Ordnung der Parabeln in Y-Struktur für beliebigen Suchbaum

Y. locate (x)

Y. insert (P(a,b,c))

Y. delete (...)

4) Konstruktion des Voronoi-Diagramms aus Folge der Circle-Events.

→ 2), 3), 4) das alles in Übungsaufgaben.

⊢→ 3.4.3.13. Ausgabe:

Für jedes Circle-Event geben wir einen Mittelpkt als Voronoi-Knoten v aus und die Orte auf dem Kreis a,b,c gegen Uhrzeigersinn.
(Können mehr als drei sein bei degenerierten Eingaben)
d.h. wir geben eigentlich die Dreiecke der Delaunay-Triangulierung aus (dualer Graph zum Voronoi-Diagramm).

⊢→ 3.4.3.14. Übung:

1). Berechne aus dieser Ausgabe eine explizite Darstellung des Voronoi-Diagramms (dh. Voronoi-Kanten)

2) Modifiziere sie den Alg. so, dass die Voronoi-Kanten direkt ausgegeben werden.

3) Degenerierte Eingaben

Circle-Event: mehr als drei Orte auf einem Kreis ⇒ mehr als ein Bogen verschwindet.

4). Alle geometrischen Primitiven:

- Schnitt von Parabeln
- Vgl. von Parabeln in Y-Struktur.

→ können alle in O(1) Berechnet werden.

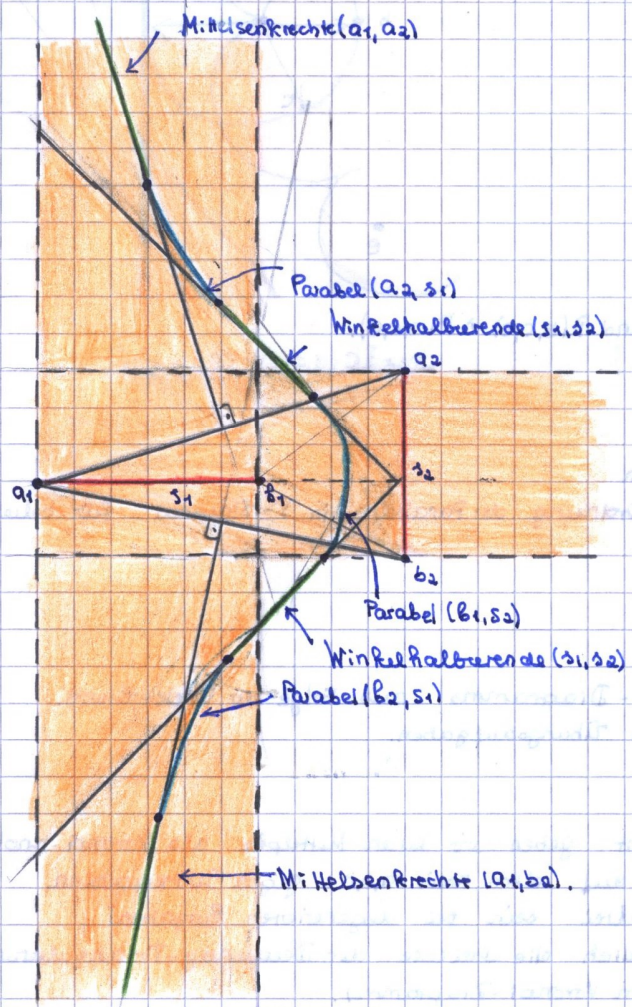
⊢→ 3.4.3.15 Satz: Das Voronoi-Diagramm von n punktförmigen Orten kann durch plane-Sweep in Zeit O(n log n) berechnet werden.

Bew: Alle Operationen auf X- bzw. Y- Str. können in Zeit O(log n) ausgeführt werden, wenn sie durch binäre Suchbäume realisiert werden.
Außerdem hat VD lineare Größe in Zahl der Orte.

⟶ 3.4.3.16 Bem: Diese Alg. kann verallgemeinert werden für andere Formen von Orten. Dazu muß man kompliziertere Bisektoren betrachten.

⟶ 3.4.3.17 Bsp: Kurvensegmente (Strecken).

Bisektor für zwei Segmente s_1, s_2:



Bisektor von zwei Strecken ist eine Kurve, die sich aus Parabelstücken, Mittelsenkrechten und Winkelhalbierenden zusammensetzt.
Je nach Lage und Länge der Strecken können Kurvenarten fehlen, z.B. Winkelhalbierende oder so nicht vorhanden.

⟶ 3.4.3.18: Bemerkung:
Ann. Segmente schneiden sich nicht (sonst Zerlegung an Schnittpkten)
Prinzipiell funktioniert der Sweep-Alg. zur Berechnung des VD von Segmenten genauso wie für Pkte.
Lediglich geometrische Primitiven sind komplizierter: Schnitt & Ordnung von Bisektoren.

3.4.4 Planar point location:

⟶ 3.4.4.1 Aufgabe: Finde für beliebigen Pkt $p \in \mathbb{R}^2$ die Voronoi-Region, die p enthält. Dann ist der Ort dieser Region, der p am nächsten liegende Ort.

⟶ 3.4.4.2 Idee der Vorverarbeitung:
Verbrauche $O(n \log n)$ für Konstruktion von VD, um danach (viele) Anfragen effizient beantworten zu können.

⟶ 3.4.4.3 Ziel: Laufzeit $O(\log n)$ pro Frage.

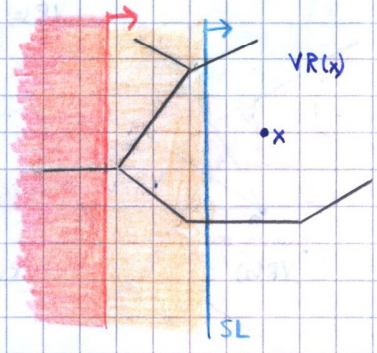
Frage: ab wieviel Abfragen lohnt es sich VD zu konstruieren?
Abfragen: $= M < n \Rightarrow$ Konstruktion v. VD + anschließend Streifenmethode kostet für M Abfragen:
$O(n \log n) + M \cdot O(\log n) = O(n \log n) + O(M \cdot \log n) = O(n \log n)$ } ⇒ Falls $M < n$, besser kein VD
Suche nach Region durch Lineare Suche (ohne VD) kostet für M Abfragen: $O(M \cdot n) = O(n)$ } aufbauen!
Für $M > n \Rightarrow$ 1 Mögl. kostet: $O(n \log n) + O(M \log n) = O(M \log n)$ } ⇒ Besser VD aufbauen! [weil $\log n < M$].
2 Mögl. kostet: $O(Mn) = O(M^2)$

3.4.3. Konstruktion von Voronoi-Diagrammen durch einen "Plane Sweep Algorithmus".

Zunächst: allg. Lage der Orte
- Keine 4 Orte liegen auf einem gemeinsamen Kreis
- paarweise verschiedene x-Koordinaten aller Orte und Events.

→ 3.4.3.1. Ziel: Bewege Senkrechte SL von links nach rechts über die Menge der Orte und gib Knoten und Kanten des VD der bereits besuchten Orte aus.

→ 3.4.3.2 Problem: Die Voronoi-Region VR(x) beginnt links von x, dh. bevor SL x erreicht.



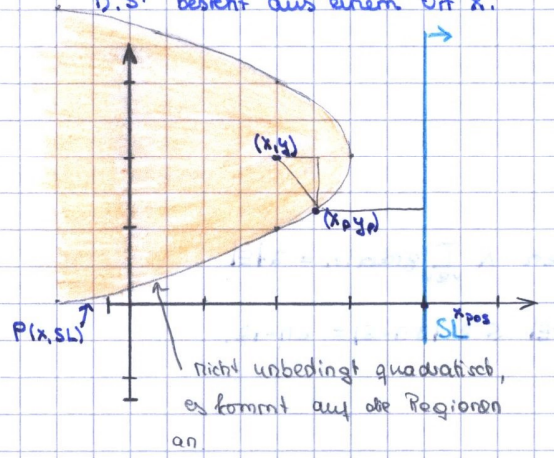
→ 3.4.3.3. Idee:
Definieren ein Gebiet weiter links von SL, in dem das Voronoi-Diagramm schon bekannt ist, dh. nicht abhängig von x.

→ 3.4.3.4. Beobachtung:
Voronoi-Regionen von Orten rechts von SL enthalten keine Pkte, die näher zu schon besuchten Orten liegen als zu SL
Denn sonst würde ein solcher Pkt (unabh. von SL) näher zu einem anderen Ort liegen.
Abstand zu SL ist nicht größer als Abstand zum nächsten Ort.

→ 3.4.3.5 Frage:
Betrachte das Gebiet $L := \{p \in \mathbb{R}^2 : dist(p,x) \leq dist(p,SL)$ für einen schon besuchten Ort $x\}$
⇒ Voronoi-Diagramm von S im Gebiet L hängt nicht von Orten rechts von SL ab.
Welche Form hat L?

→ 3.4.3.6. Beispiele: S' ← Menge der besuchten Orte.

1). S' besteht aus einem Ort x.

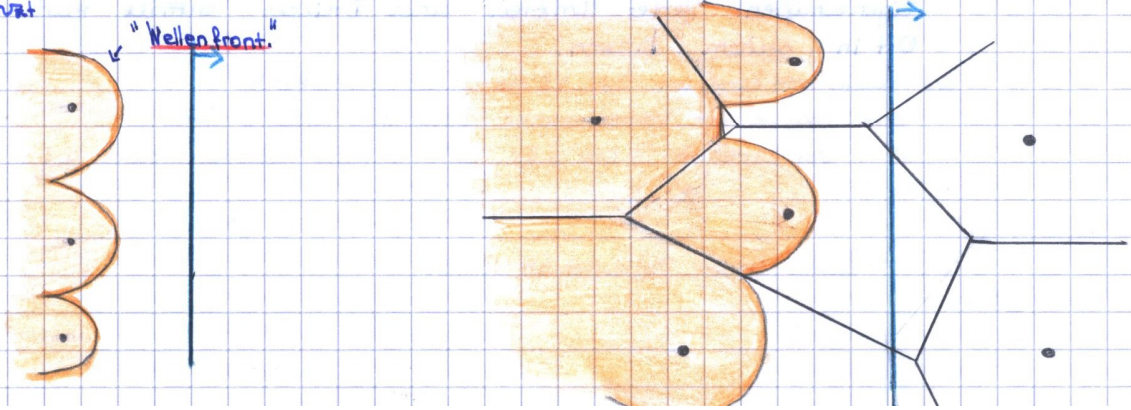


nicht unbedingt quadratisch, es kommt auf die Regionen an

(x,y) und SL bekannt
Wie sieht L aus?
→ Es muß gelten $\forall (x_p y_p) \in L$:
$(x_p - x_{pos})^2 \overset{!}{=} (x_p - x)^2 + (y_p - y)^2$
weil es muß eigentlich gelten:
$x_p - x_{pos} = \sqrt{(x_p - x)^2 + (y_p - y)^2}$.
→ (x,y) = (2,2), SL ≡ 4 bekannt
$\Rightarrow (x_p - 4)^2 = (x_p - 2)^2 + (y_p - 2)^2$
$\Leftrightarrow x_p^2 - 8x_p + 16 = x_p^2 - 4x_p + 4 + y_p^2 - 4y_p + 4$
$\Leftrightarrow -4x_p + 8 = y_p^2 - 4y_p$
$\Leftrightarrow 4x_p = -y_p^2 + 4y_p + 8$
$\Leftrightarrow x_p = -\frac{1}{4}y_p^2 + y_p + 2$
→ dies ist eine Parabelgleichung mit der Variablen y

→ Inneres der Parabel P(x,SL) ist die Menge der Pkte mit gleicher Distanz zu x und SL

2). Im Allgemeinen wird L durch eine Folge von Parabelbögen nach rechts begrenzt



⟶ 3.4.3.7 Beobachtung: Schnittpkte von benachbarten Parabeln in der Wellenfront überstreichen die Kanten von VD(S).

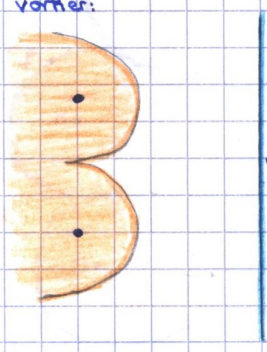
⟶ 3.4.3.8 Idee: Verwalte die Wellenfront der Parabelbögen in einer Y-Struktur.

⟶ 3.4.3.9 Fragen: An welchen Positionen (Transaktionspkten / Events) der SL ändert sich die Y-Struktur strukturell?
→ Welche Events gibt es?

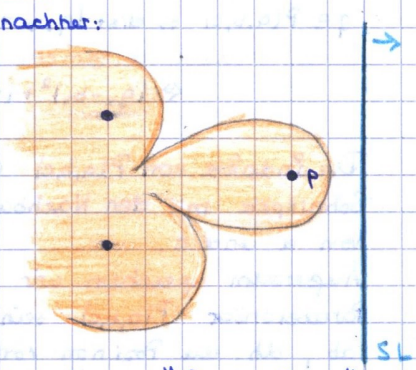
⟶ 3.4.3.10 Zwei Arten von Events:

1). SL überstreicht neuen Ort p "Site Event"

vorher:

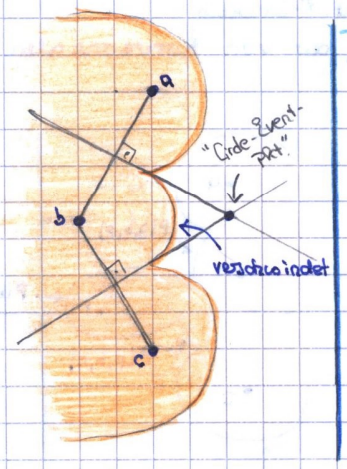


nachher:



2). Ein Parabelbogen verschwindet aus der Wellenfront. "Circle Event"

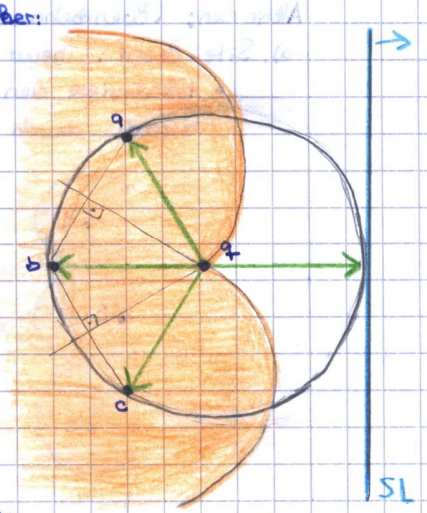
vorher:



? drei Parabeln schneiden sich in einem Pkt

bea: q gehört nicht zu S(1,a) Es ist nur ein Schnpkt von VR'en und zwar Circle-Eventpkt.

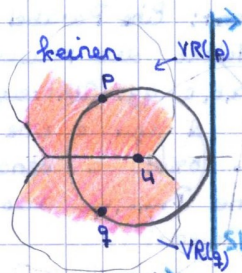
nachher:



↑ Hier entsteht ein neuer Voronoi-Knoten q = Mittelpkt des Kreises durch a,b,c
SL ist zu diesem Zeitpkt Tangente an den Kreis (durch a,b,c).

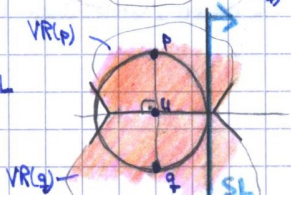
⟶ 3.4.3.11 Lemma: Jeder Pkt auf der Kante von VD(S) kommt während des Sweep als Schnittpkt zweier in der Y-Struktur benachbarter Parabelbögen vor.
(Bei kontinuierlicher Sweep)

Beweis:
Sei e beliebige Voronoi-Kante von VD(S) und u ein beliebiger Pkt auf e. e trennt zwei Voronoi-Regionen VR(p) und VR(q)
u liegt näher zu p und q (gleiche Distanz d) als zu allen anderen Orten.
Kreismittelpkt u enthält im Inneren keinen Ort.
Nun betrachte den Zeitpkt des Sweep, in dem SL rechte Tangente an diesem Kreis K ist



a) u hat den gleichen Abstand zu p und SL
⇒ u liegt auf Parabel von p

b) u hat gl. Abstand zu q und SL
⇒ u liegt auf Parabel von q



⇒ Beh.

3.4.3.12. Implementierungsdetails:

1). Darstellung der Parabelbögen:

Ein Parabelstück wird durch drei Orte a,b,c und die aktuelle Position x_{pos} der Sweepline definiert.

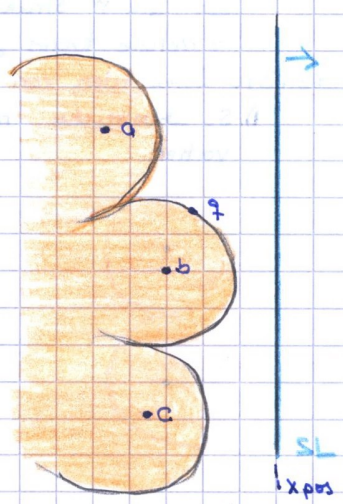
$P(a,b,c)$ bezeichnet Parabelstück von b, das an Parabeln von Ort a und von Ort c grenzt.

Gleichung der Parabel:

$q \in P(a,b,c) \Leftrightarrow dist(q,b) = dist(q,SL)$

auf Parabelstück, gemeint auf dem Rand $\Rightarrow$ "$=$"

$\Leftrightarrow (q_x - b_x)^2 + (q_y - b_y)^2 = (q_x - x_{pos})^2$



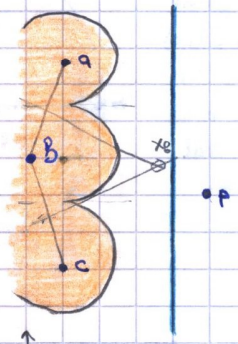
Die Endpkte von $P(a,b,c)$ ergeben sich als Schnittpkte mit den Nachbarparabeln, dh. von a und c.

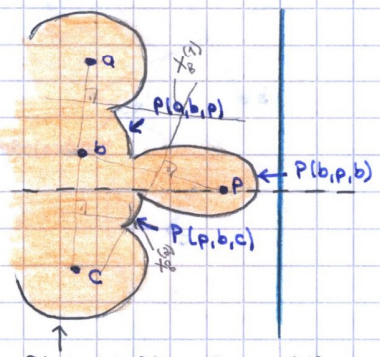
Außerdem speichern wir mit jedem Parabelstück $P(a,b,c)$ sein Circle-Event ab, dh. die Position von SL, bei der $P(a,b,c)$ aus Wellenfront verschwindet.

Aktionen: (Eventbehandlung).

a). Site-Event: neuer Ort p erreicht. i.d. Y-Struktur

- lokalisiere den Parabelbogen $P(a,b,c)$, der von horizontaler Geraden durch p geschnitten wird.





Y-Str.: In der Y-Str. $= \{P(-,a,b), P(a,b,c), P(b,c,+)\}$

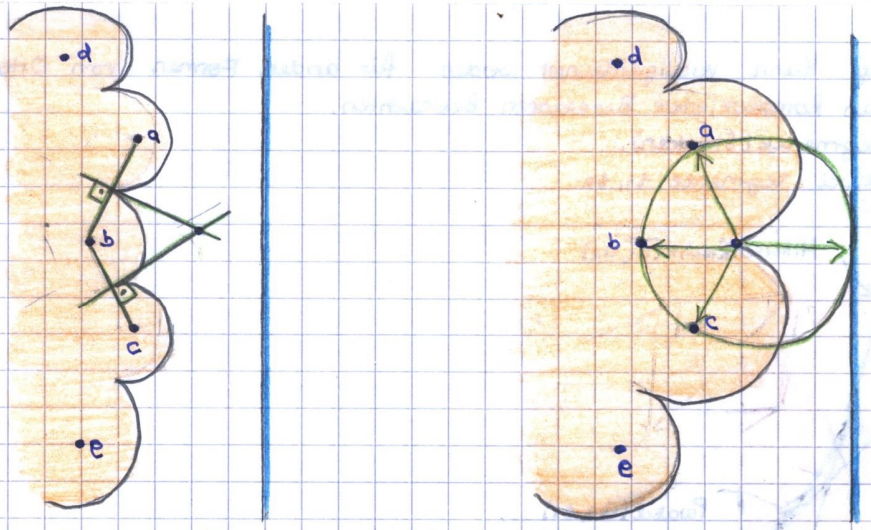
$P(a,b,c)$ löschen aus Y-Str. und drei neue Bögen in die Y-Str. einfügen

$\Rightarrow$ Y-Str. $= \{P(-,a,b), P(a,b,p), P(b,p,b), P(p,b,c), P(b,c,+)\}$

- lösche $P(a,b,c)$ aus Y-Str.
- lösche Circle-Event von $P(a,b,c)$ aus X-Str. (PRkt)
- füge 3 neue Bögen in die Y-Str. ein
- berechne die Circle-Events der neuen Parabelstücke (PRkt) und füge sie in die X-Str. ein.

b). Circle-Event: Eine Parabel $P(a,b,c)$ verschwindet.

- Gib Mittelpkte des Kreises durch a,b,c als Voronoi-Knoten aus mit a,b,c gg. Uhrzeigersinn. (berechne zugleich Mittelsenkrechte von ab und bc und gebe sie aus)
- Seien d und e Orte mit Parabelbögen, die mit Parabel von a bzw. c benachbart sind.
- Entferne $P(a,b,c)$ aus Y-Struktur. und lösche den bearbeiteten Circle-Event-Pkt aus der X-Struktur
- berechne die neuen Circle-Event-Pkte



→ $P(d,a,b) \rightsquigarrow P(d,a,c)$ und $P(b,c,e) \rightsquigarrow P(a,c,e)$.

$X = \{C(d,a,b), C(a,b,c), C(b,c,e)\}$ $X = \{C(d,a,c), C(a,c,e)\}$.

2) Initialisierung

3). Geometrische Primitive

a) Schnitt von Parabeln

b) Vergleichsoptt für Ordnung der Parabeln in Y-Struktur für beliebigen Suchbaum

Y. locate (x)

Y. insert (P(a,b,c))

Y. delete (...)

4) Konstruktion des Voronoi-Diagramms aus Folge der Circle-Events.

→ 2), 3), 4) das alles in Übungsaufgaben.

⊢→ 3.4.3.13. Ausgabe:

Für jedes Circle-Event geben wir einen Mittelpkt als Voronoi-Knoten v aus und die Orte auf dem Kreis a,b,c gegen Uhrzeigersinn.
(Können mehr als drei sein bei degenerierten Eingaben)
d.h. wir geben eigentlich die Dreiecke der Delaunay-Triangulierung aus (dualer Graph zum Voronoi-Diagramm).

⊢→ 3.4.3.14. Übung:

1). Berechne aus dieser Ausgabe eine explizite Darstellung des Voronoi-Diagramms (dh. Voronoi-Kanten)

2) Modifiziere sie den Alg. so, dass die Voronoi-Kanten direkt ausgegeben werden.

3) Degenerierte Eingaben

Circle-Event: mehr als drei Orte auf einem Kreis ⇒ mehr als ein Bogen verschwindet.

4). Alle geometrischen Primitiven:

- Schnitt von Parabeln
- Vgl. von Parabeln in Y-Struktur.

→ können alle in O(1) Berechnet werden.

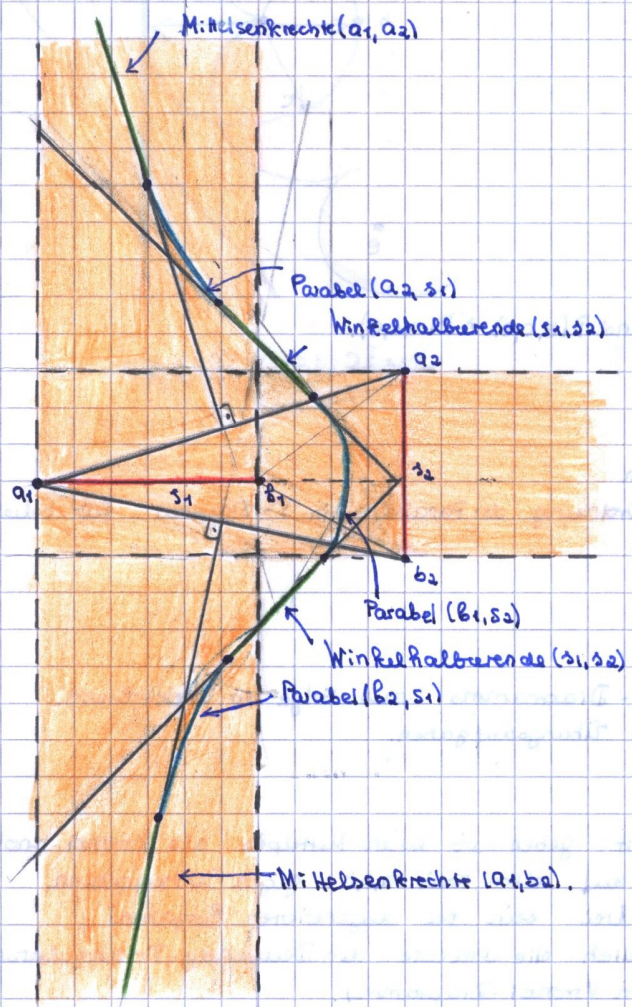
⊢→ 3.4.3.15 Satz: Das Voronoi-Diagramm von n punktförmigen Orten kann durch plane-Sweep in Zeit O(n log n) berechnet werden.

Bew: Alle Operationen auf X- bzw. Y- Str. können in Zeit O(log n) ausgeführt werden, wenn sie durch binäre Suchbäume realisiert werden.
Außerdem hat VD lineare Größe in Zahl der Orte.

⟶ 3.4.3.16 Bem: Diese Alg. kann verallgemeinert werden für andere Formen von Orten. Dazu muß man kompliziertere Bisektoren betrachten.

⟶ 3.4.3.17 Bsp: Kurvensegmente (Strecken).

Bisektor für zwei Segmente s_1, s_2:



Bisektor von zwei Strecken ist eine Kurve, die sich aus Parabelstücken, Mittelsenkrechten und Winkelhalbierenden zusammensetzt.
Je nach Lage und Länge der Strecken können Kurvenarten fehlen, z.B. Winkelhalbierende oder so nicht vorhanden.

⟶ 3.4.3.18: Bemerkung:
Ann. Segmente schneiden sich nicht (sonst Zerlegung an Schnittpkten)
Prinzipiell funktioniert der Sweep-Alg. zur Berechnung des VD von Segmenten genauso wie für Pkte.
Lediglich geometrische Primitiven sind komplizierter: Schnitt & Ordnung von Bisektoren.

3.4.4 Planar point location:

⟶ 3.4.4.1 Aufgabe: Finde für beliebigen Pkt $p \in \mathbb{R}^2$ die Voronoi-Region, die p enthält. Dann ist der Ort dieser Region, der p am nächsten liegende Ort.

⟶ 3.4.4.2 Idee der Vorverarbeitung:
Verbrauche $O(n \log n)$ für Konstruktion von VD, um danach (viele) Anfragen effizient beantworten zu können.

⟶ 3.4.4.3 Ziel: Laufzeit $O(\log n)$ pro Frage.

Frage: ab wieviel Abfragen lohnt es sich VD zu konstruieren?
Abfragen: $= M < n \Rightarrow$ Konstruktion v. VD + anschließend Streifenmethode kostet für M Abfragen:
$O(n \log n) + M \cdot O(\log n) = O(n \log n) + O(M \cdot \log n) = O(n \log n)$ } ⇒ Falls $M < n$, besser kein VD
Suche nach Region durch Lineare Suche (ohne VD) kostet für M Abfragen: $O(M \cdot n) = O(n)$ } aufbauen!
Für $M > n \Rightarrow$ 1 Mögl. kostet: $O(n \log n) + O(M \log n) = O(M \log n)$ } ⇒ Besser VD aufbauen! [weil $\log n < M$].
2 Mögl. kostet: $O(Mn) = O(M^2)$

3.4.5. Point-Location allgemein (unabh. v. Voronoi-Diagramm).

→ 3.4.5.1. Problem:

Geg: Beliebige planare Unterteilung der Ebene (Subdivision)

Ges: point location.

dh. finde für beliebigen Pkt p das Gebiet in dem p liegt.

⇔ finde die Kante der Unterteilung, die von einem vertikalen Strahl von p aus nach unten zuerst getroffen wird.

dh. die die man "sieht", wenn man von p nach unten schaut.

(→ vertikales Sichtbarkeitsproblem).

Schreibe den Namen (Nummer, ID, ...) jedes Gebiets an seine untere Folge von Kanten.

⇒ wenn untere Kante ausgeg. wird, so weiß man in welchem Gebiet p liegt.

⇒ daher sind die Aussagen äquivalent

→ 3.4.5.2. Streifenmethode: allgemein:

die Streifenmethode zerlegt dieses 2-dim. Suchproblem in 2 1-dimensionale.

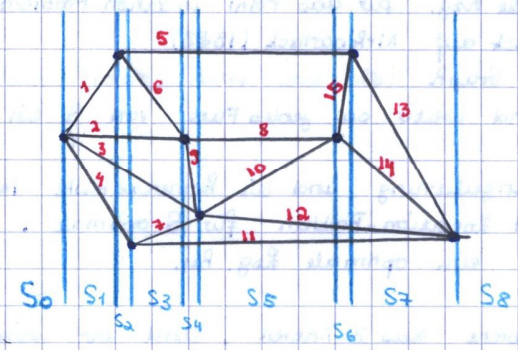
→ 3.4.5.2 Streifenmethode: Idee:

Zerlege die Ebene in vertikale Streifen $S_0, S_1, .., S_n$ durch n vertikale Geraden jeweils durch einen Knoten von G.

G = planare Unterteilung

n = # Knoten von G.

Preprocessing: $\Theta(n^2)$ im schlechtesten Fall.



Speichere die Streifen in einem Feld S[0..n].

Schritt 1: Finde den Streifen S[i], der p enthält durch binäre Suche nach p.xcoord() in S

→ kostet Zeit O(log n).

Darstellung jedes Streifens S[i]:

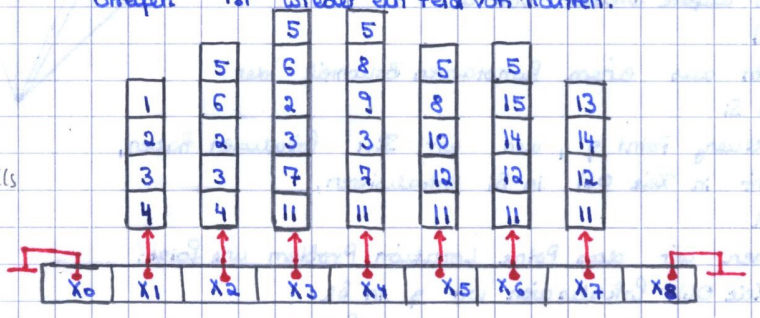
Eine Folge der Kanten von G, die S_i überqueren von unten nach oben (gemäß ihrer Lage in S_i) sortiert.

Bem: Kanten schneiden sich nicht innerhalb eines Streifens, dh. jeder Streifen ist wieder ein Feld von Kanten.

Ein Feld von 2-dim Pkten bzw. Intervallen $x_0 := [-\infty, x_0], [x_0, x_1], [x_1, x_2] \dots, [x_8, \infty]$

Zweites Feld von Zeigern:

Zeiger zeigen auf Felder von ebenfalls 2-dim Pkten.

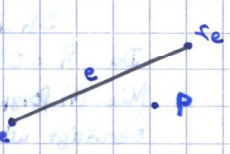


Schritt 2: Binärsuche auf Streifen S[i] selbst.

genauer: sortiere p in Folge der Kanten ein.

Liegt oberhalb, auf oder unterhalb einer Kante e

↝ orientation $(l_e, r_e, p) = ?$



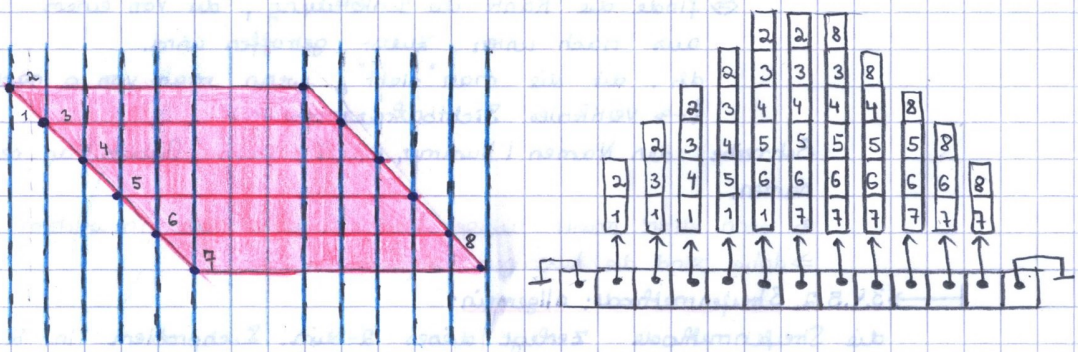
Da insgesamt O(n) Kanten } ⇒ kostet auch Zeit O(log n)
(G ist planarer Graph)

Zu Laufzeit: binäre Suche ist $T(n) = T(\frac{n}{2}) + c_1$ ⇒ $f(n) = c_1$, $\log_8 a = \log_2 1 = 0$

⇒ $f(n) = \Theta(n^{\log_8 a})$ ⇒ $T(n) = \Theta(n^{\log_8 a} \cdot \log n) = \underline{\Theta(\log n)}$

Streifenmethode: ergebnis:

Laufzeit: $O(\log n)$ (optimal)
Platzbedarf: $O(n^2)$ (unpraktisch für großes n) $O(n)+O(n^2)=O(n^2)$
Ziel: $O(\log n)$ Suchzeit und $O(n)$ Platz.
Beispiel für quadratischen Platz:



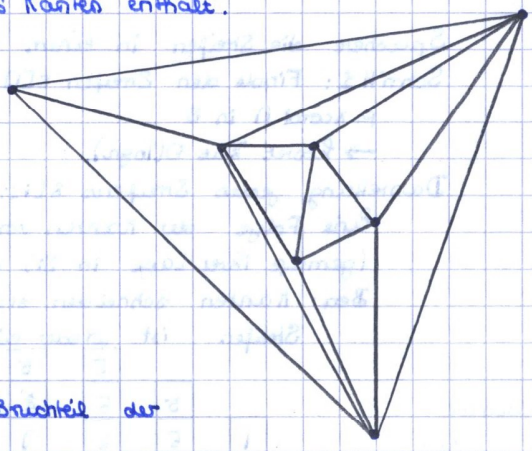
Gesamt: n Knoten (hier n=12)
Jede der $n/2$ Kanten ist in $n/2$ Streifen gespeichert.

3.4.5.4 Triangulierungsmethode: allgemein:

- Ber. eine optimale Lsg. für das Point Location Problem.
- Methode geht zurück auf Kirkpatrick (1983)

Sei G ein planarer Graph (dh. zwei versch. Kanten überschneiden sich nicht u. ungerichtet) auf n Knoten und weiter sei jedes Face von G ein Dreieck (auch das äußere Face)
$\Rightarrow$ G ist eine Triangulierung und die konvexe Hülle ist auch Dreieck. $|CH|=3$.
Wir lösen das Point Location-Problem für G optimal. Später leiten wir für jede planare Unterteilung eine optimale Lsg. her.

Die konv. Hülle besteht aus 3 Knoten. Und wir wissen weiter, dass jede Triangulierung dieser n Knoten $3n-6$ Kanten enthält.
$(3n-k-3,\ k=|CH|)$.



3.4.5.5 Triangulierungsmethode: Idee für Algorithmus:

Konstruiere eine Folge $S_1 .. S_R$ von Triangulierungen, so dass gilt:
(1) $S_1 = G$
(2) S_R ist das äußere Dreieck von G
(3) $R = O(\log n)$
(4) S_{i+1} besteht aus einem konstanten Bruchteil der Knoten von S_i
(5) Für jeden Query Point q, den wir S_{i+1} lokalisiert haben, können wir in Zeit $O(1)$ in S_i lokalisieren.

Geg: $S_1 .. S_R$
dann lösen wir das Point Location Problem wie folgt:

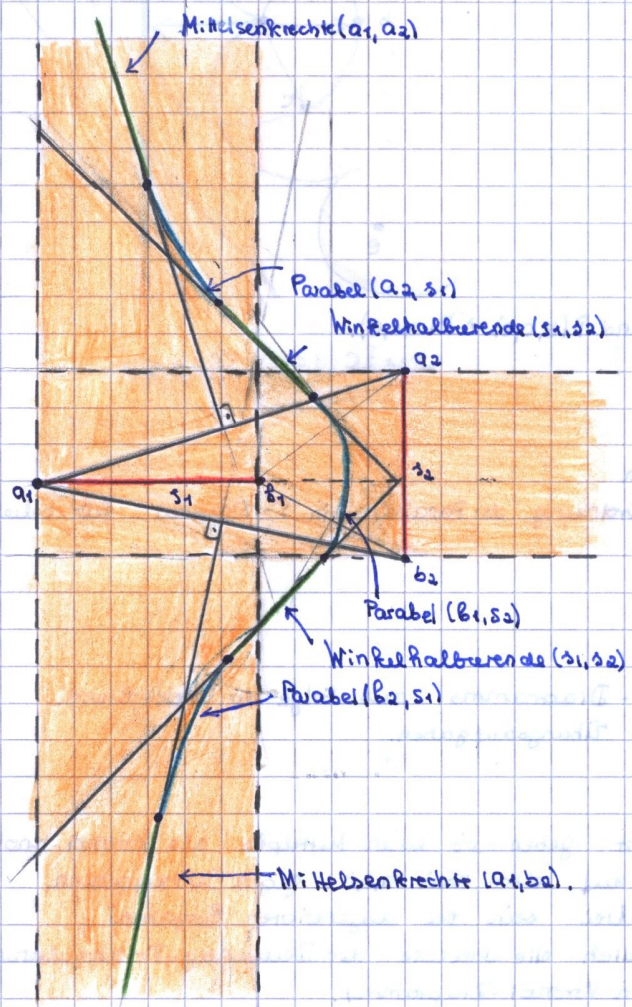
- in Zeit $O(1)$ lokalisieren wir q in S_R
- dann unter Benutzung von (5) lokalisieren wir nacheinander q in $S_{R-1}, S_{R-2}, .., S_1 = G$.

Da $R = O(\log n)$ haben wir eine Suchzeit von $O(\log n)$
Mit (4) folgt, dass für jede Folge der Triangulierung nur Platz $O(n)$ benötigt wird.

⟶ 3.4.3.16 Bem: Diese Alg. kann verallgemeinert werden für andere Formen von Orten. Dazu muß man kompliziertere Bisektoren betrachten.

⟶ 3.4.3.17 Bsp: Kurvensegmente (Strecken).

Bisektor für zwei Segmente s_1, s_2:



Bisektor von zwei Strecken ist eine Kurve, die sich aus Parabelstücken, Mittelsenkrechten und Winkelhalbierenden zusammensetzt.
Je nach Lage und Länge der Strecken können Kurvenarten fehlen, z.B. Winkelhalbierende oder so nicht vorhanden.

⟶ 3.4.3.18: Bemerkung:
Ann. Segmente schneiden sich nicht (sonst Zerlegung an Schnittpkten)
Prinzipiell funktioniert der Sweep-Alg. zur Berechnung des VD von Segmenten genauso wie für Pkte.
Lediglich geometrische Primitiven sind komplizierter: Schnitt & Ordnung von Bisektoren.

3.4.4 Planar point location:

⟶ 3.4.4.1 Aufgabe: Finde für beliebigen Pkt $p \in \mathbb{R}^2$ die Voronoi-Region, die p enthält. Dann ist der Ort dieser Region, der p am nächsten liegende Ort.

⟶ 3.4.4.2 Idee der Vorverarbeitung:
Verbrauche $O(n \log n)$ für Konstruktion von VD, um danach (viele) Anfragen effizient beantworten zu können.

⟶ 3.4.4.3 Ziel: Laufzeit $O(\log n)$ pro Frage.

Frage: ab wieviel Abfragen lohnt es sich VD zu konstruieren?
Abfragen: $= M < n \Rightarrow$ Konstruktion v. VD + anschließend Streifenmethode kostet für M Abfragen:
$O(n \log n) + M \cdot O(\log n) = O(n \log n) + O(M \cdot \log n) = O(n \log n)$ } ⇒ Falls $M < n$, besser kein VD
Suche nach Region durch Lineare Suche (ohne VD) kostet für M Abfragen: $O(M \cdot n) = O(n)$ } aufbauen!
Für $M > n \Rightarrow$ 1 Mögl. kostet: $O(n \log n) + O(M \log n) = O(M \log n)$ } ⇒ Besser VD aufbauen! [weil $\log n < M$].
2 Mögl. kostet: $O(Mn) = O(M^2)$

3.4.5. Point-Location allgemein (unabh. v. Voronoi-Diagramm).

→ 3.4.5.1. Problem:

Geg: Beliebige planare Unterteilung der Ebene (Subdivision)

Ges: point location.

dh. finde für beliebigen Pkt p das Gebiet in dem p liegt.

⇔ finde die Kante der Unterteilung, die von einem vertikalen Strahl von p aus nach unten zuerst getroffen wird.

dh. die die man "sieht", wenn man von p nach unten schaut.

(→ vertikales Sichtbarkeitsproblem).

Schreibe den Namen (Nummer, ID, ...) jedes Gebiets an seine untere Folge von Kanten.

⇒ wenn untere Kante ausgeg. wird, so weiß man in welchem Gebiet p liegt.

⇒ daher sind die Aussagen äquivalent

→ 3.4.5.2. Streifenmethode: allgemein:

die Streifenmethode zerlegt dieses 2-dim. Suchproblem in 2 1-dimensionale.

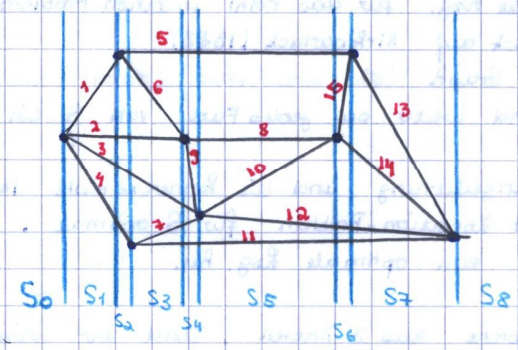
→ 3.4.5.2 Streifenmethode: Idee:

Zerlege die Ebene in vertikale Streifen $S_0, S_1, .., S_n$ durch n vertikale Geraden jeweils durch einen Knoten von G.

G = planare Unterteilung

n = # Knoten von G.

Preprocessing: $\Theta(n^2)$ im schlechtesten Fall.



Speichere die Streifen in einem Feld S[0..n].

Schritt 1: Finde den Streifen S[i], der p enthält durch binäre Suche nach p.xcoord() in S

→ kostet Zeit O(log n).

Darstellung jedes Streifens S[i]:

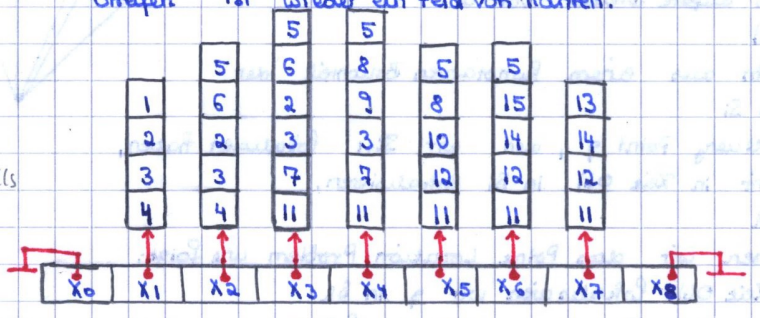
Eine Folge der Kanten von G, die S_i überqueren von unten nach oben (gemäß ihrer Lage in S_i) sortiert.

Bem: Kanten schneiden sich nicht innerhalb eines Streifens, dh. jeder Streifen ist wieder ein Feld von Kanten.

Ein Feld von 2-dim Pkten bzw. Intervallen $x_0 := [-\infty, x_0], [x_0, x_1], [x_1, x_2] \dots, [x_8, \infty]$

Zweites Feld von Zeigern:

Zeiger zeigen auf Felder von ebenfalls 2-dim Pkten.

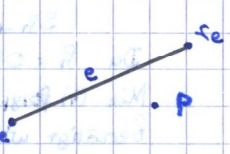


Schritt 2: Binärsuche auf Streifen S[i] selbst.

genauer: sortiere p in Folge der Kanten ein.

Liegt oberhalb, auf oder unterhalb einer Kante e

↝ orientation $(l_e, r_e, p) = ?$



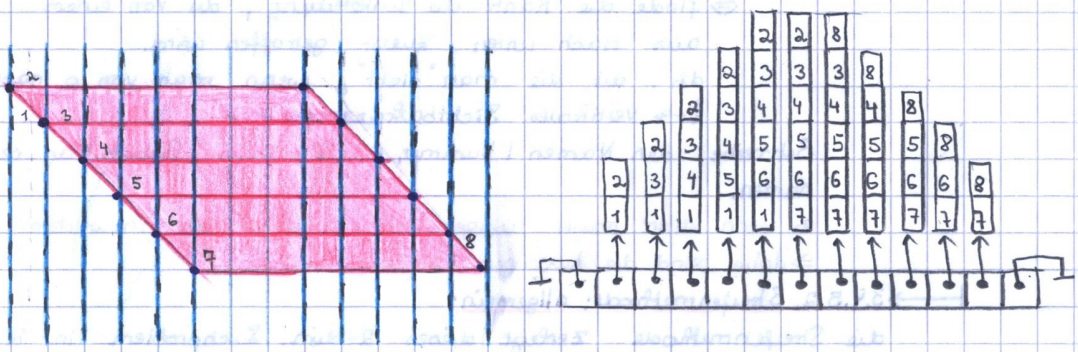
Da insgesamt O(n) Kanten } ⇒ kostet auch Zeit O(log n)
(G ist planarer Graph)

Zu Laufzeit: binäre Suche ist $T(n) = T(\frac{n}{2}) + c_1$ ⇒ $f(n) = c_1$, $\log_8 a = \log_2 1 = 0$

⇒ $f(n) = \Theta(n^{\log_8 a})$ ⇒ $T(n) = \Theta(n^{\log_8 a} \cdot \log n) = \underline{\Theta(\log n)}$

Streifenmethode: ergebnis:

Laufzeit: $O(\log n)$ (optimal)
Platzbedarf: $O(n^2)$ (unpraktisch für großes n) $O(n)+O(n^2)=O(n^2)$
Ziel: $O(\log n)$ Suchzeit und $O(n)$ Platz.
Beispiel für quadratischen Platz:



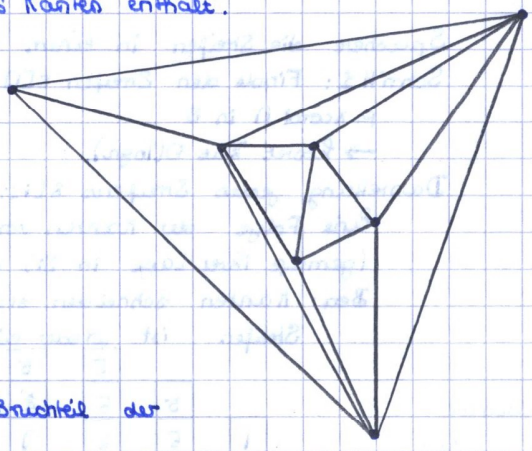
Gesamt: n Knoten (hier n=12)
Jede der $n/2$ Kanten ist in $n/2$ Streifen gespeichert.

3.4.5.4 Triangulierungsmethode: allgemein:

- Ber. eine optimale Lsg. für das Point Location Problem.
- Methode geht zurück auf Kirkpatrick (1983)

Sei G ein planarer Graph (dh. zwei versch. Kanten überschneiden sich nicht u. ungerichtet) auf n Knoten und weiter sei jedes Face von G ein Dreieck (auch das äußere Face)
$\Rightarrow$ G ist eine Triangulierung und die konvexe Hülle ist auch Dreieck. $|CH|=3$.
Wir lösen das Point Location-Problem für G optimal. Später leiten wir für jede planare Unterteilung eine optimale Lsg. her.

Die konv. Hülle besteht aus 3 Knoten. Und wir wissen weiter, dass jede Triangulierung dieser n Knoten $3n-6$ Kanten enthält.
$(3n-k-3,\ k=|CH|)$.



3.4.5.5 Triangulierungsmethode: Idee für Algorithmus:

Konstruiere eine Folge $S_1 .. S_R$ von Triangulierungen, so dass gilt:
(1) $S_1 = G$
(2) S_R ist das äußere Dreieck von G
(3) $R = O(\log n)$
(4) S_{i+1} besteht aus einem konstanten Bruchteil der Knoten von S_i
(5) Für jeden Query Point q, den wir S_{i+1} lokalisiert haben, können wir in Zeit $O(1)$ in S_i lokalisieren.

Geg: $S_1 .. S_R$
dann lösen wir das Point Location Problem wie folgt:

- in Zeit $O(1)$ lokalisieren wir q in S_R
- dann unter Benutzung von (5) lokalisieren wir nacheinander q in $S_{R-1}, S_{R-2}, .., S_1 = G$.

Da $R = O(\log n)$ haben wir eine Suchzeit von $O(\log n)$
Mit (4) folgt, dass für jede Folge der Triangulierung nur Platz $O(n)$ benötigt wird.

⟶ 3.4.5.6 Triangulierungsmethode: Fragen & Antworten.

1) Frage: Wie konstruieren wir die Folge $S_1 .. S_R$?
   → Ber. $S_1 = G_1$.
   Wollen einen konst. Bruchteil der Knoten entfernen.
   Sei v ein non-boundary Knoten von G_1 und sei d sein Grad, dh. in G_1 sind d Kanten inzident zu v.
2) Frage: Was passiert, wenn wir v aus G_1 entfernen?
   → wenn wir v entfernen, entfernen wir auch die d inzidenten Kanten.
   ⇒ wenn wir v entfernen, dann auch die d Dreiecke. Die d Dreiecke werden durch ein einfaches Polygon ersetzt.

Sei q ein beliebiger Pkt im d-gon, das wir durch Entfernen von v erhalten. Dann können wir in Zeit $O(d)$ bestimmen, welches der d Dreiecke von G_1 q enthält.

Um Eig. (5) zu erhalten, sollten wir Knoten vom kleinen Grad entfernen. Für die Eig. (4) sollten wir möglichst viele dieser Knoten (mit kleinem Grad) entfernen.

3) Frage: Ist es immer möglich viele Knoten mit kleinem Grad zu bestimmen?

⟶ 3.4.5.7 Triangulierungsmethode: Def (unabh.):

Eine Teilmenge der Knotenmenge heißt unabhängig, wenn keine zwei Knoten durch eine Kante verbunden sind.

⟶ 3.4.5.8 Lemma:

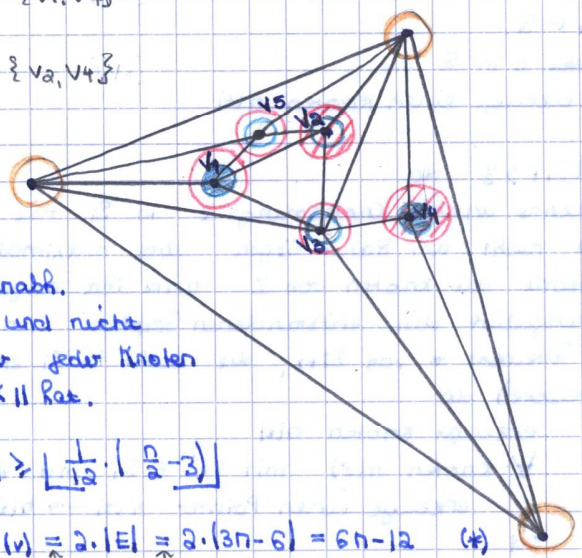
Der Graph G enthält eine unabhängige Knotenmenge I mit der Größe von mindestens $\left\lceil \frac{1}{12}\left(\frac{n}{2}-3\right)\right\rceil$, so dass jeder Knoten aus I höchstens Grad 11 hat und jeder Knoten ein nicht Hüll-Knoten ist. So eine Menge I kann in Zeit $O(n)$ gefunden werden.

Beweis: Betrachte folgenden Grady-Alg:

1. markiere alle drei Hüllknoten.
2. $I \leftarrow \emptyset$
3. repeat
4. wähle einen Knoten v mit Grad ≤ 11, der nicht markiert ist
5. $I \leftarrow I \cup \{v\}$
6. markiere v und alle seine Nachbarn.
7. until es gibt keine unmarkierten Knoten mit Grad ≤ 11.

a) $I \leftarrow \{v_1\} \cup I$, $I \leftarrow \{v_4\} \cup I$ $\Rightarrow I = \{v_1, v_4\}$

b) $I \leftarrow \{v_3\} \cup I$, $I \leftarrow \{v_8\} \cup I$ $\Rightarrow I = \{v_2, v_4\}$



Also:
- Alg. benötigt Zeit $O(n)$
- Er findet eine unabh. Knotenmenge I und nicht Hüllknoten, in der jeder Knoten höchstens Grad ≤ 11 hat.

Es bleibt z.z.: $|I| \geq \left\lfloor \frac{1}{12} \cdot \left(\frac{n}{2} - 3 \right) \right\rfloor$

Wissen: $\sum_{v \in V} \text{degree}(v) \overset{\text{allg.}}{=} 2 \cdot |E| \overset{\text{HÜ 2.1.}}{=} 2 \cdot (3n-6) = 6n-12$ (*)

→

$\Rightarrow$ G enthält mind. $\frac{n}{2}$ Knoten vom Grad ≤ 11.

Denn: Ann: nein.

$\Rightarrow$ mind. $\frac{n}{2}$ Knoten sind vom Grad > 11 bzw. dann ≥ 12.

$\Rightarrow \sum_{v \in V} \text{degree}(v) \geq \frac{n}{2} \cdot 12 = 6n$

$\Rightarrow \lightning$ zu (*)

// (*) sagt $\sum_{v \in V} \text{degree}(v) < 6n$!

Betrachte Algorithmus:

Alg. startet mit Markierung der drei begrenzten Knoten.

In diesem Moment gibt es mind. $\frac{n}{2} - 3$ unmarkierte Knoten mit Grad ≤ 11.

Dann wählt der Alg. einen dieser Knoten und markiert ihn zusammen mit seinen höchstens 11 Nachbarn.

Das wird solange wiederholt bis es keine unmarkierten Knoten mit Grad ≤ 11 mehr gibt.

Deshalb werden während jeder Iteration höchstens 12 Knoten markiert.

Es gibt also mindestens $\left\lceil \frac{1}{12} \left(\frac{n}{2} - 3 \right) \right\rceil$

Während jeder Iteration wird ein Knoten zu I hinzugefügt.

Ergebnis: $\forall$ Triang. G gilt: G enthält unabh. Knotenmenge I, so dass

1) $|I| \geq \left\lceil \frac{1}{12} \left(\frac{n}{2} - 3 \right) \right\rceil$
2) $\forall v \in I$: degree(v) ≤ 11
3) I enthält keinen der drei Randknoten.
4) I kann in Zeit $O(n)$ berechnet werden.

$\longmapsto$ 3.4.5.9 Lemma: Sei I eine unabh. Knotenmenge von G gemäß 3.4.5.8.
Sei G' der Graph, der durch Entfernen von I aus G entsteht

$\Rightarrow$ 1). Jede Fläche von G' ist ein 1-faches Polygon. mit maximal 11 Knoten

2). Die äußeren Flächen (Dreiecke) von G und G' sind gleich.

$\longmapsto$ 3.4.5.10 Alg. zu Konstruktion der Datenstruktur D zur Darstellung der Folge $S_1 .. S_k$:

Datenstruktur D: Knoten + Pointer.

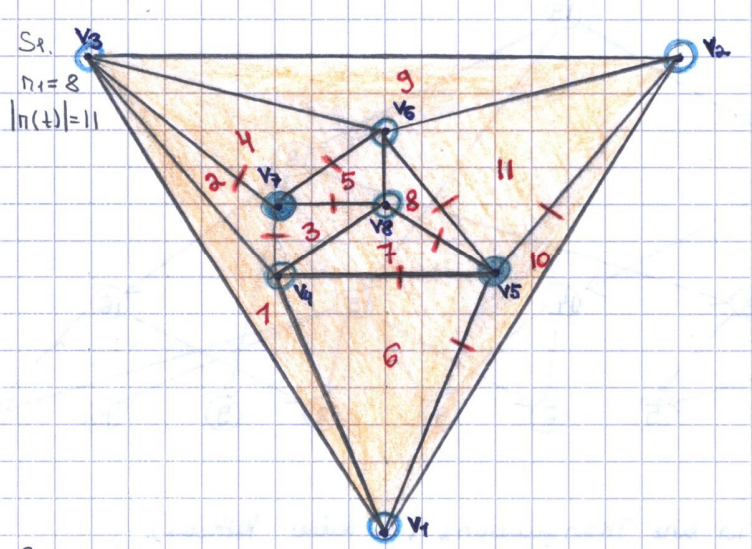
Knoten v speichert ein Dreieck t $\quad v = n(t)$.

Triangulierung S: Pkte, Kanten, Flächen

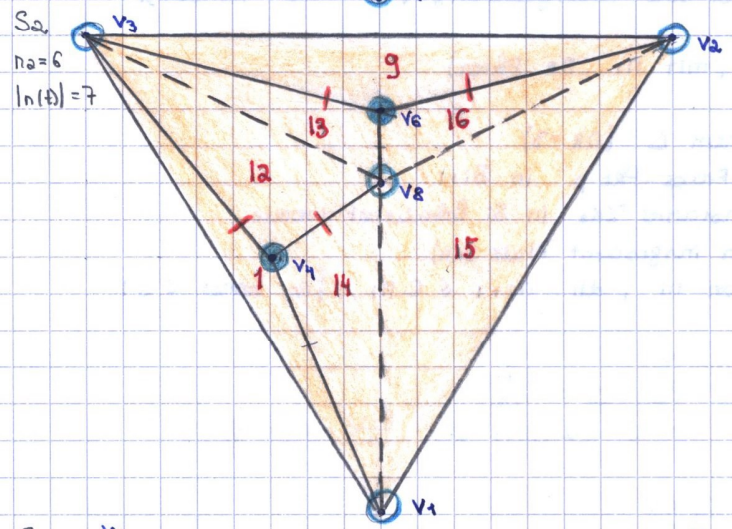
$n_i = |S_i|$ = # der Pkte.

1. $i \leftarrow 1$, $S_1 \leftarrow G$.
2. $\forall$ Dreiecke t in G do
3. erzeuge einen Knoten n(t)
4. od.
5. while $|S_i| > 3$ do
6. berechne unabh. Knotenmenge I in S_i mit mind. $\left\lceil \frac{1}{12} \left(\frac{n}{2} - 3 \right) \right\rceil$ Knoten, die nicht am Rand liegen und maximal Grad 11 haben.
7. Entferne die Knoten von I und ihre angegrenzten Kanten aus S_i
8. Triangulire den entstandenen Graphen und nenne das Resultat S_{i+1}.
9. $\forall$ Dreiecke t von S_{i+1}, die nicht in S_i enthalten sind (d.h. neue Dreiecke) do
10. erzeuge Knoten n(t)
11. $\forall$ Knoten n(t') mit $t' \in S_i$ und t' schneidet t do
12. erzeuge einen Pointer $n(t) \longrightarrow n(t')$
13. od
14. od
15. $i \leftarrow i+1$
16. od

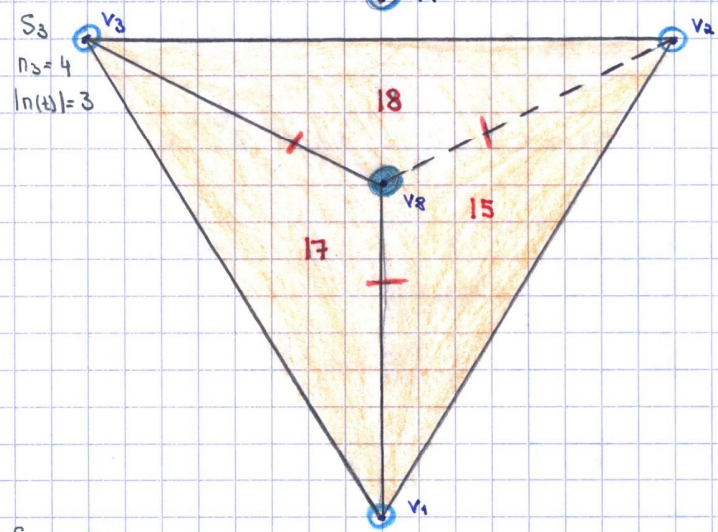
⟶ 3.4.5.11. Beispiel:



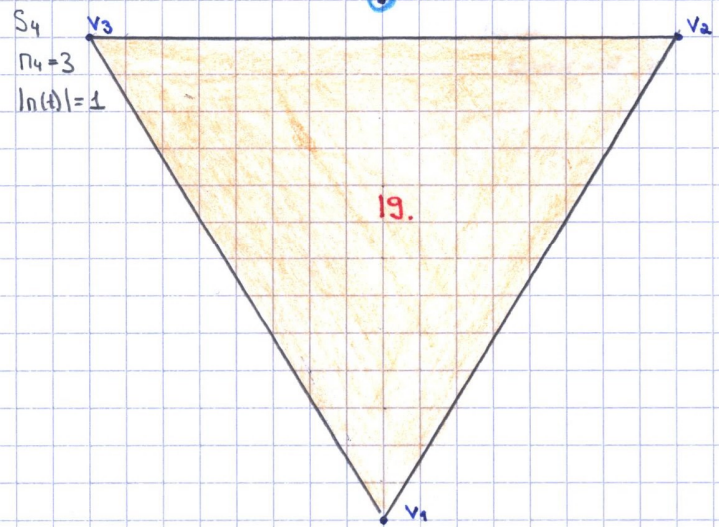
i = 1
I = { v5, v7 }
n(1) ... n(11) n(16)
neu Dreiecke: n(12), n(13), n(14), n(15)
n(12) ⟶ n(2), n(4), n(3), n(5)
n(13) ⟶ n(4), n(5)
n(14) ⟶ n(6), n(7)
n(15) ⟶ n(10), n(6), n(7), n(8), n(11)
n(16) ⟶ n(11), n(8)



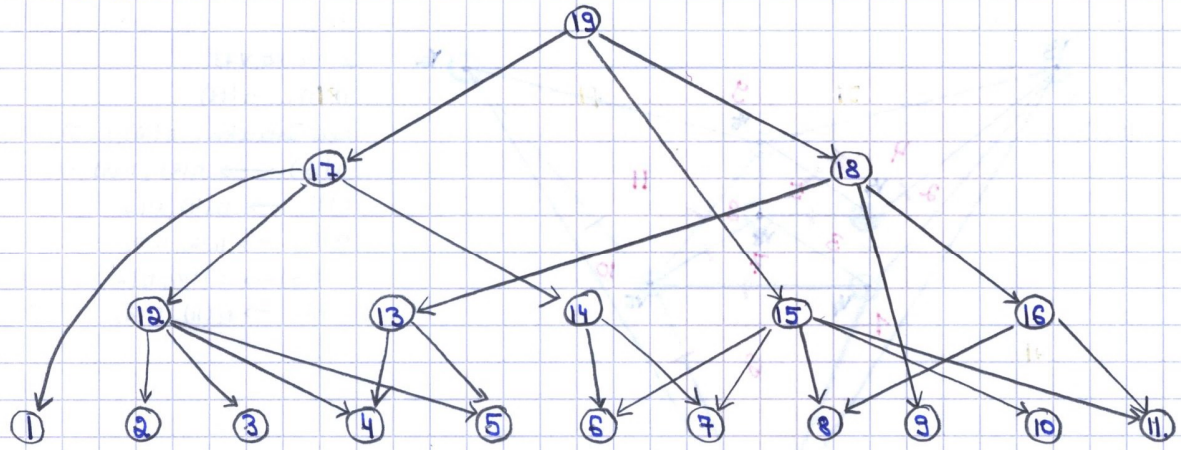
i = 2
I = { v6, v4 }
neue Dreiecke: n(16), n(17), n(18)
n(17) ⟶ n(1), n(14), n(12)
n(18) ⟶ n(13), n(9), n(16)



i = 3
I = { v8 }
neues Dreieck: n(19)
n(19) → n(15), n(17), n(18)



Datenstruktur D:



3.4.5.12. Zusammenfassung:

Planare Unterteilung G ist eine Triangulierung (bei dieser Methode).
(auch Rand ist Δ).

Ziel: Folge $S_1 .. S_R$ (mit $n_1 .. n_R$ Pkten)

✓ 1) $S_1 = G$

✓ 2) S_R besteht aus einem Δ ($n_R = 3$)

3) Lokalisierung von Frage-Pkt p in S_{i+1}
⇒ p kann in konstanter Zeit in S_i lokalisiert werden.

4) $S_1 .. S_R$ brauchen insgesamt Platz $O(n)$

5) n_{i+1} ist Bruchteil von n_i, d.h. $n_{i+1} \leq c \cdot n_i$ für konst $c \leq 1$
⇒ $R = O(\log n)$.

Es gilt: $n_{i+1} < n_i \ \forall i$

⇒ $n_{i+1} < c_i \cdot n_i$ für $c_i > \frac{n_{i+1}}{n_i}$

Ferner: $\frac{n_{i+1}}{n_i} < 1$ ⇒ c_i wählbar als $\frac{n_{i+1}}{n_i} < c_i < 1$

Gilt $\forall i$ und alle $c_i < 1$
⇒ wähle $c := c_R$ $\quad [\, n_3 < \underbrace{c_3}_{<1} \cdot n_2 < \underbrace{c_3 \cdot c_2}_{<1} \cdot n_1 < c_3 \cdot n_1 \,]$

⇒ $n_{i+1} < c \cdot n_i$

⇒ $3 = n_R < c \cdot n_{R-1} < c^2 \cdot n_{R-2} < \ldots < c^{R-1} \cdot n_1 = c^{R-1} \cdot n$

⇒ $n > \frac{3}{c^{R-1}}$

⇒ $\log_2 n > \log_2 3 + \log_2 \left(\frac{1}{c}\right)^{R-1}$

⇒ $\log_2 n > \frac{\ln 3}{\ln 2} + (R-1) \cdot (\underbrace{\log_2 1}_{=0} - \log_2 c)$

⇒ $R - 1 < \frac{\log n - \frac{\ln 3}{\ln 2}}{-\frac{\ln c}{\ln 2}} = \log n \cdot \left(-\frac{\ln c}{\ln 2}\right) + \frac{\ln 3}{\ln c}$

⇒ $R < \underbrace{-\frac{\ln c}{\ln 2}}_{\substack{>0 \text{ weil } c<1 \\ \text{const}}} \cdot \log n + \underbrace{\frac{\ln 3}{\ln c} + 1}_{\text{const}}$

⇒ $R \leq M \cdot \log n$ mit $M = \text{konst}$

⇒ $R = O(\log n)$.

3.4.5.13. Algorithmus für Point-Location:

Eingabe: Pkt q
Datenstruktur $\mathcal{D}$.

$\mathcal{D}$.search(q) liefert Dreieck von G, das q enthält.

if q außerhalb Dreieck i.d. Wurzel then
  Aussage: "äußeres Gebiet von G".
else
  $v \leftarrow$ Wurzel
  while v kein Blatt (dh. $\mathcal{D}$.outdeg(v)>0) do
    forall Knoten u mit $\exists$ Pointer $v \to u$ do
      if q innerhalb Dreieck von u then
        $v \leftarrow u$;
      fi
    od
  od
  Ausgabe: Dreieck von v
fi

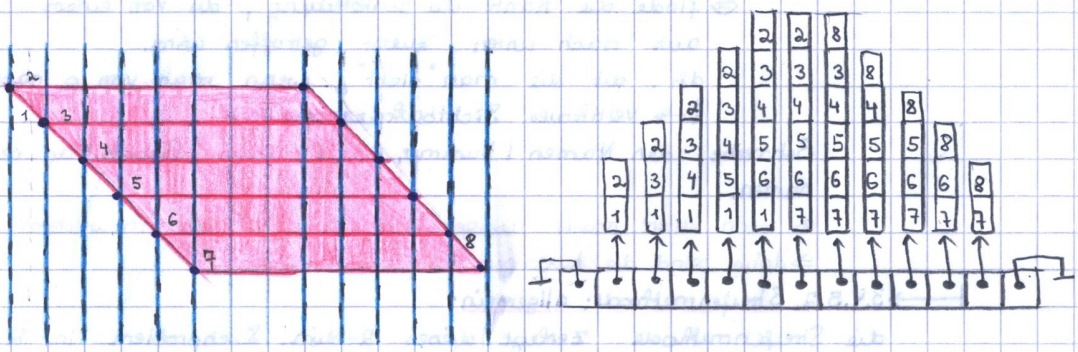
Schleifeninvariante: Während der Ausführung der while-Schleife gilt stets, dass $q \in$ Dreieck von v.

Bemerkung: Platz:

$$\#\text{Knoten} = n_1 + n_2 + \dots + n_R \leq$$
$$\leq n_1 + c \cdot n_1 + \dots + c^{R-1} \cdot n_1 =$$
$$= n_1 \sum_{\nu=0}^{R-1} c^\nu = n_1 \cdot \frac{c^R - 1}{c-1} = n \cdot \underbrace{\frac{c^R - 1}{c-1}}_{\text{const}} = \mathcal{O}(n)$$

Streifenmethode: ergebnis:

Laufzeit: $O(\log n)$ (optimal)
Platzbedarf: $O(n^2)$ (unpraktisch für großes n) $O(n)+O(n^2)=O(n^2)$
Ziel: $O(\log n)$ Suchzeit und $O(n)$ Platz.
Beispiel für quadratischen Platz:



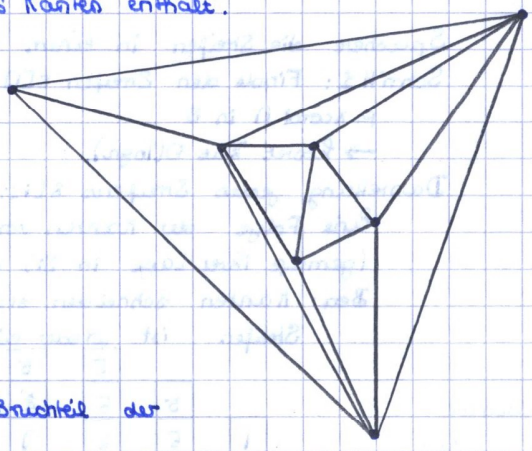
Gesamt: n Knoten (hier n=12)
Jede der $n/2$ Kanten ist in $n/2$ Streifen gespeichert.

3.4.5.4 Triangulierungsmethode: allgemein:

- Ber. eine optimale Lsg. für das Point Location Problem.
- Methode geht zurück auf Kirkpatrick (1983)

Sei G ein planarer Graph (dh. zwei versch. Kanten überschneiden sich nicht u. ungerichtet) auf n Knoten und weiter sei jedes Face von G ein Dreieck (auch das äußere Face)
$\Rightarrow$ G ist eine Triangulierung und die konvexe Hülle ist auch Dreieck. $|CH|=3$.
Wir lösen das Point Location-Problem für G optimal. Später leiten wir für jede planare Unterteilung eine optimale Lsg. her.

Die konv. Hülle besteht aus 3 Knoten. Und wir wissen weiter, dass jede Triangulierung dieser n Knoten $3n-6$ Kanten enthält.
$(3n-k-3,\ k=|CH|)$.



3.4.5.5 Triangulierungsmethode: Idee für Algorithmus:

Konstruiere eine Folge $S_1 .. S_R$ von Triangulierungen, so dass gilt:
(1) $S_1 = G$
(2) S_R ist das äußere Dreieck von G
(3) $R = O(\log n)$
(4) S_{i+1} besteht aus einem konstanten Bruchteil der Knoten von S_i
(5) Für jeden Query Point q, den wir S_{i+1} lokalisiert haben, können wir in Zeit $O(1)$ in S_i lokalisieren.

Geg: $S_1 .. S_R$
dann lösen wir das Point Location Problem wie folgt:

- in Zeit $O(1)$ lokalisieren wir q in S_R
- dann unter Benutzung von (5) lokalisieren wir nacheinander q in $S_{R-1}, S_{R-2}, .., S_1 = G$.

Da $R = O(\log n)$ haben wir eine Suchzeit von $O(\log n)$
Mit (4) folgt, dass für jede Folge der Triangulierung nur Platz $O(n)$ benötigt wird.

⟶ 3.4.5.6 Triangulierungsmethode: Fragen & Antworten.

1) Frage: Wie konstruieren wir die Folge $S_1 .. S_R$?
   → Ber. $S_1 = G_1$.
   Wollen einen konst. Bruchteil der Knoten entfernen.
   Sei v ein non-boundary Knoten von G_1 und sei d sein Grad, dh. in G_1 sind d Kanten inzident zu v.
2) Frage: Was passiert, wenn wir v aus G_1 entfernen?
   → wenn wir v entfernen, entfernen wir auch die d inzidenten Kanten.
   ⇒ wenn wir v entfernen, dann auch die d Dreiecke. Die d Dreiecke werden durch ein einfaches Polygon ersetzt.

Sei q ein beliebiger Pkt im d-gon, das wir durch Entfernen von v erhalten. Dann können wir in Zeit $O(d)$ bestimmen, welches der d Dreiecke von G_1 q enthält.

Um Eig. (5) zu erhalten, sollten wir Knoten vom kleinen Grad entfernen. Für die Eig. (4) sollten wir möglichst viele dieser Knoten (mit kleinem Grad) entfernen.

3) Frage: Ist es immer möglich viele Knoten mit kleinem Grad zu bestimmen?

⟶ 3.4.5.7 Triangulierungsmethode: Def (unabh.):

Eine Teilmenge der Knotenmenge heißt unabhängig, wenn keine zwei Knoten durch eine Kante verbunden sind.

⟶ 3.4.5.8 Lemma:

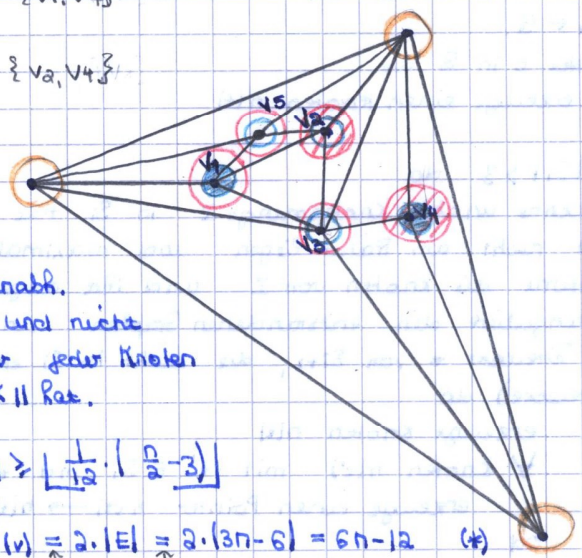
Der Graph G enthält eine unabhängige Knotenmenge I mit der Größe von mindestens $\left\lceil \frac{1}{12}\left(\frac{n}{2}-3\right)\right\rceil$, so dass jeder Knoten aus I höchstens Grad 11 hat und jeder Knoten ein nicht Hüll-Knoten ist. So eine Menge I kann in Zeit $O(n)$ gefunden werden.

Beweis: Betrachte folgenden Grady-Alg:

1. markiere alle drei Hüllknoten.
2. $I \leftarrow \emptyset$
3. repeat
4. wähle einen Knoten v mit Grad ≤ 11, der nicht markiert ist
5. $I \leftarrow I \cup \{v\}$
6. markiere v und alle seine Nachbarn.
7. until es gibt keine unmarkierten Knoten mit Grad ≤ 11.

a) $I \leftarrow \{v_1\} \cup I$, $I \leftarrow \{v_4\} \cup I$ $\Rightarrow I = \{v_1, v_4\}$

b) $I \leftarrow \{v_3\} \cup I$, $I \leftarrow \{v_8\} \cup I$ $\Rightarrow I = \{v_2, v_4\}$



Also:
- Alg. benötigt Zeit $O(n)$
- Er findet eine unabh. Knotenmenge I und nicht Hüllknoten, in der jeder Knoten höchstens Grad ≤ 11 hat.

Es bleibt z.z.: $|I| \geq \left\lfloor \frac{1}{12} \cdot \left(\frac{n}{2} - 3 \right) \right\rfloor$

Wissen: $\sum_{v \in V} \text{degree}(v) \overset{\text{allg.}}{=} 2 \cdot |E| \overset{\text{HÜ 2.1.}}{=} 2 \cdot (3n-6) = 6n-12$ (*)

→

$\Rightarrow$ G enthält mind. $\frac{n}{2}$ Knoten vom Grad ≤ 11.

Denn: Ann: nein.

$\Rightarrow$ mind. $\frac{n}{2}$ Knoten sind vom Grad > 11 bzw. dann ≥ 12.

$\Rightarrow \sum_{v \in V} \text{degree}(v) \geq \frac{n}{2} \cdot 12 = 6n$

$\Rightarrow \lightning$ zu (*)

// (*) sagt $\sum_{v \in V} \text{degree}(v) < 6n$!

Betrachte Algorithmus:

Alg. startet mit Markierung der drei begrenzten Knoten.

In diesem Moment gibt es mind. $\frac{n}{2} - 3$ unmarkierte Knoten mit Grad ≤ 11.

Dann wählt der Alg. einen dieser Knoten und markiert ihn zusammen mit seinen höchstens 11 Nachbarn.

Das wird solange wiederholt bis es keine unmarkierten Knoten mit Grad ≤ 11 mehr gibt.

Deshalb werden während jeder Iteration höchstens 12 Knoten markiert.

Es gibt also mindestens $\left\lceil \frac{1}{12} \left(\frac{n}{2} - 3 \right) \right\rceil$

Während jeder Iteration wird ein Knoten zu I hinzugefügt.

Ergebnis: $\forall$ Triang. G gilt: G enthält unabh. Knotenmenge I, so dass

1) $|I| \geq \left\lceil \frac{1}{12} \left(\frac{n}{2} - 3 \right) \right\rceil$
2) $\forall v \in I$: degree(v) ≤ 11
3) I enthält keinen der drei Randknoten.
4) I kann in Zeit $O(n)$ berechnet werden.

$\longmapsto$ 3.4.5.9 Lemma: Sei I eine unabh. Knotenmenge von G gemäß 3.4.5.8.
Sei G' der Graph, der durch Entfernen von I aus G entsteht

$\Rightarrow$ 1). Jede Fläche von G' ist ein 1-faches Polygon. mit maximal 11 Knoten

2). Die äußeren Flächen (Dreiecke) von G und G' sind gleich.

$\longmapsto$ 3.4.5.10 Alg. zu Konstruktion der Datenstruktur D zur Darstellung der Folge $S_1 .. S_k$:

Datenstruktur D: Knoten + Pointer.

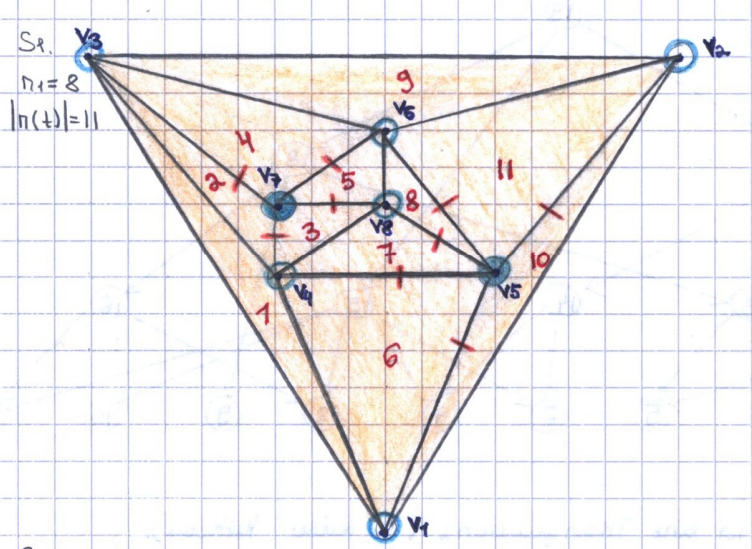
Knoten v speichert ein Dreieck t $\quad v = n(t)$.

Triangulierung S: Pkte, Kanten, Flächen

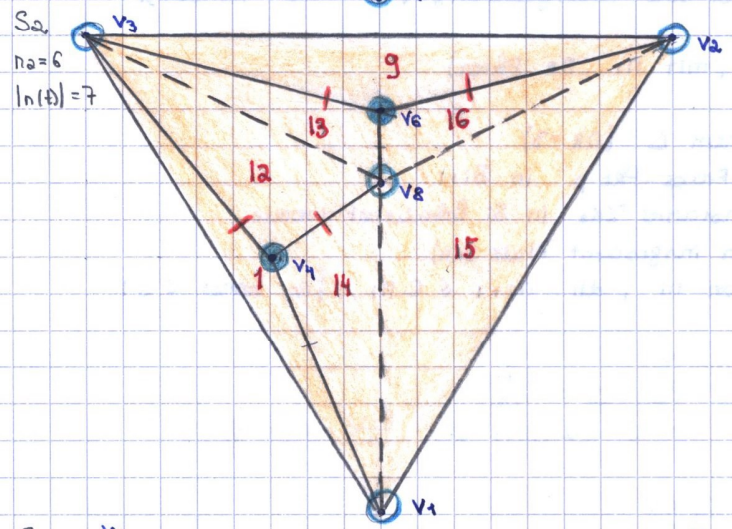
$n_i = |S_i|$ = # der Pkte.

1. $i \leftarrow 1$, $S_1 \leftarrow G$.
2. $\forall$ Dreiecke t in G do
3. erzeuge einen Knoten n(t)
4. od.
5. while $|S_i| > 3$ do
6. berechne unabh. Knotenmenge I in S_i mit mind. $\left\lceil \frac{1}{12} \left(\frac{n}{2} - 3 \right) \right\rceil$ Knoten, die nicht am Rand liegen und maximal Grad 11 haben.
7. Entferne die Knoten von I und ihre angegrenzten Kanten aus S_i
8. Triangulire den entstandenen Graphen und nenne das Resultat S_{i+1}.
9. $\forall$ Dreiecke t von S_{i+1}, die nicht in S_i enthalten sind (d.h. neue Dreiecke) do
10. erzeuge Knoten n(t)
11. $\forall$ Knoten n(t') mit $t' \in S_i$ und t' schneidet t do
12. erzeuge einen Pointer $n(t) \longrightarrow n(t')$
13. od
14. od
15. $i \leftarrow i+1$
16. od

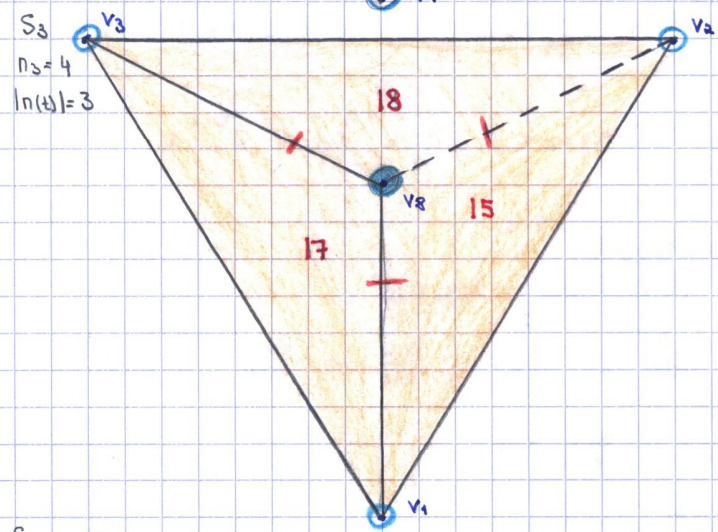
⟶ 3.4.5.11. Beispiel:



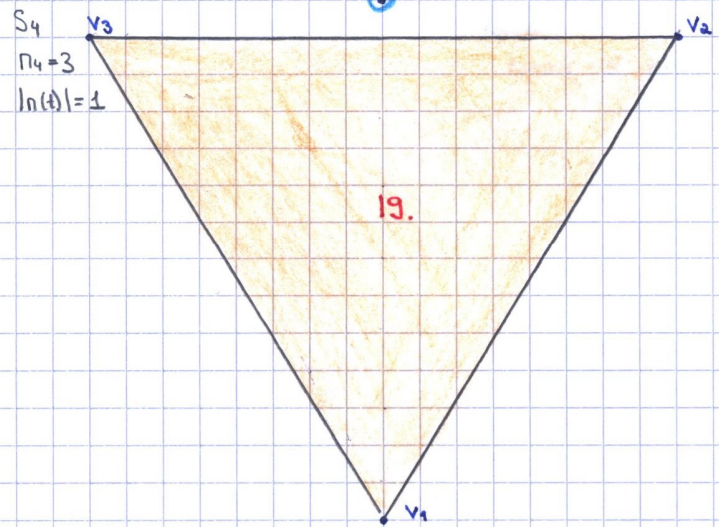
i = 1
I = { v5, v7 }
n(1) ... n(11) n(16)
neu Dreiecke: n(12), n(13), n(14), n(15)
n(12) ⟶ n(2), n(4), n(3), n(5)
n(13) ⟶ n(4), n(5)
n(14) ⟶ n(6), n(7)
n(15) ⟶ n(10), n(6), n(7), n(8), n(11)
n(16) ⟶ n(11), n(8)



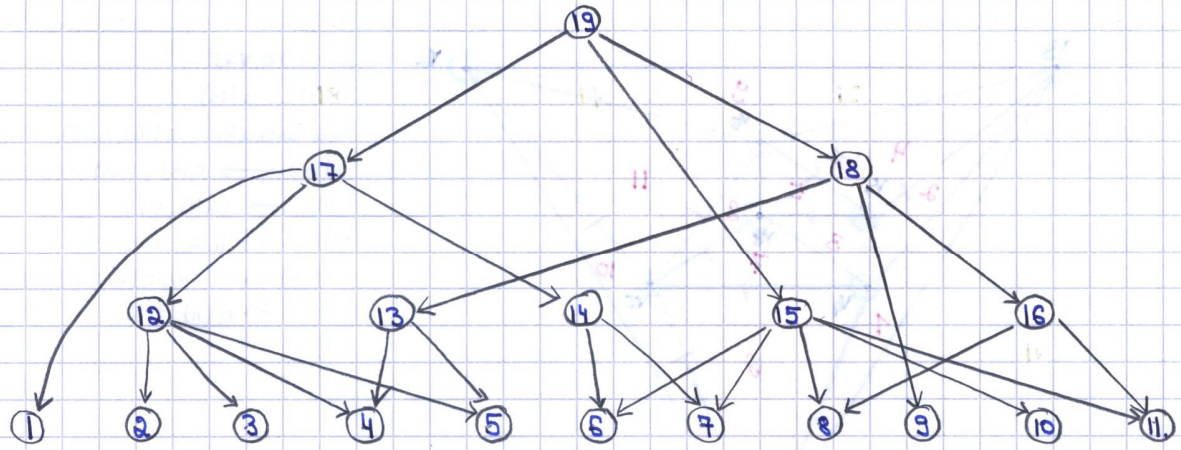
i = 2
I = { v6, v4 }
neue Dreiecke: n(16), n(17), n(18)
n(17) ⟶ n(1), n(14), n(12)
n(18) ⟶ n(13), n(9), n(16)



i = 3
I = { v8 }
neues Dreieck: n(19)
n(19) → n(15), n(17), n(18)



Datenstruktur D:



3.4.5.12. Zusammenfassung:

Planare Unterteilung G ist eine Triangulierung (bei dieser Methode).
(auch Rand ist Δ).

Ziel: Folge $S_1 .. S_R$ (mit $n_1 .. n_R$ Pkten)

✓ 1) $S_1 = G$

✓ 2) S_R besteht aus einem Δ ($n_R = 3$)

3) Lokalisierung von Frage-Pkt p in S_{i+1}
⇒ p kann in konstanter Zeit in S_i lokalisiert werden.

4) $S_1 .. S_R$ brauchen insgesamt Platz $O(n)$

5) n_{i+1} ist Bruchteil von n_i, d.h. $n_{i+1} \leq c \cdot n_i$ für konst $c \leq 1$
⇒ $R = O(\log n)$.

Es gilt: $n_{i+1} < n_i \ \forall i$

⇒ $n_{i+1} < c_i \cdot n_i$ für $c_i > \frac{n_{i+1}}{n_i}$

Ferner: $\frac{n_{i+1}}{n_i} < 1$ ⇒ c_i wählbar als $\frac{n_{i+1}}{n_i} < c_i < 1$

Gilt $\forall i$ und alle $c_i < 1$
⇒ wähle $c := c_R$ $\quad [\, n_3 < \underbrace{c_3}_{<1} \cdot n_2 < \underbrace{c_3 \cdot c_2}_{<1} \cdot n_1 < c_3 \cdot n_1 \,]$

⇒ $n_{i+1} < c \cdot n_i$

⇒ $3 = n_R < c \cdot n_{R-1} < c^2 \cdot n_{R-2} < \ldots < c^{R-1} \cdot n_1 = c^{R-1} \cdot n$

⇒ $n > \frac{3}{c^{R-1}}$

⇒ $\log_2 n > \log_2 3 + \log_2 \left(\frac{1}{c}\right)^{R-1}$

⇒ $\log_2 n > \frac{\ln 3}{\ln 2} + (R-1) \cdot (\underbrace{\log_2 1}_{=0} - \log_2 c)$

⇒ $R - 1 < \frac{\log n - \frac{\ln 3}{\ln 2}}{-\frac{\ln c}{\ln 2}} = \log n \cdot \left(-\frac{\ln c}{\ln 2}\right) + \frac{\ln 3}{\ln c}$

⇒ $R < \underbrace{-\frac{\ln c}{\ln 2}}_{\substack{>0 \text{ weil } c<1 \\ \text{const}}} \cdot \log n + \underbrace{\frac{\ln 3}{\ln c} + 1}_{\text{const}}$

⇒ $R \leq M \cdot \log n$ mit $M = \text{konst}$

⇒ $R = O(\log n)$.

3.4.5.13. Algorithmus für Point-Location:

Eingabe: Pkt q
Datenstruktur $\mathcal{D}$.

$\mathcal{D}$.search(q) liefert Dreieck von G, das q enthält.

if q außerhalb Dreieck i.d. Wurzel then
  Aussage: "äußeres Gebiet von G".
else
  $v \leftarrow$ Wurzel
  while v kein Blatt (dh. $\mathcal{D}$.outdeg(v)>0) do
    forall Knoten u mit $\exists$ Pointer $v \to u$ do
      if q innerhalb Dreieck von u then
        $v \leftarrow u$;
      fi
    od
  od
  Ausgabe: Dreieck von v
fi

Schleifeninvariante: Während der Ausführung der while-Schleife gilt stets, dass $q \in$ Dreieck von v.

Bemerkung: Platz:

$$\#\text{Knoten} = n_1 + n_2 + \dots + n_R \leq$$
$$\leq n_1 + c \cdot n_1 + \dots + c^{R-1} \cdot n_1 =$$
$$= n_1 \sum_{\nu=0}^{R-1} c^\nu = n_1 \cdot \frac{c^R - 1}{c-1} = n \cdot \underbrace{\frac{c^R - 1}{c-1}}_{\text{const}} = \mathcal{O}(n)$$

$\Rightarrow$ G enthält mind. $\frac{n}{2}$ Knoten vom Grad ≤ 11.

Denn: Ann: nein.

$\Rightarrow$ mind. $\frac{n}{2}$ Knoten sind vom Grad > 11 bzw. dann ≥ 12.

$\Rightarrow \sum_{v \in V} \text{degree}(v) \geq \frac{n}{2} \cdot 12 = 6n$

$\Rightarrow \lightning$ zu (*)

// (*) sagt $\sum_{v \in V} \text{degree}(v) < 6n$!

Betrachte Algorithmus:

Alg. startet mit Markierung der drei begrenzten Knoten.

In diesem Moment gibt es mind. $\frac{n}{2} - 3$ unmarkierte Knoten mit Grad ≤ 11.

Dann wählt der Alg. einen dieser Knoten und markiert ihn zusammen mit seinen höchstens 11 Nachbarn.

Das wird solange wiederholt bis es keine unmarkierten Knoten mit Grad ≤ 11 mehr gibt.

Deshalb werden während jeder Iteration höchstens 12 Knoten markiert.

Es gibt also mindestens $\left\lceil \frac{1}{12} \left(\frac{n}{2} - 3 \right) \right\rceil$

Während jeder Iteration wird ein Knoten zu I hinzugefügt.

Ergebnis: $\forall$ Triang. G gilt: G enthält unabh. Knotenmenge I, so dass

1) $|I| \geq \left\lceil \frac{1}{12} \left(\frac{n}{2} - 3 \right) \right\rceil$
2) $\forall v \in I$: degree(v) ≤ 11
3) I enthält keinen der drei Randknoten.
4) I kann in Zeit $O(n)$ berechnet werden.

$\longmapsto$ 3.4.5.9 Lemma: Sei I eine unabh. Knotenmenge von G gemäß 3.4.5.8.
Sei G' der Graph, der durch Entfernen von I aus G entsteht

$\Rightarrow$ 1). Jede Fläche von G' ist ein 1-faches Polygon. mit maximal 11 Knoten

2). Die äußeren Flächen (Dreiecke) von G und G' sind gleich.

$\longmapsto$ 3.4.5.10 Alg. zu Konstruktion der Datenstruktur D zur Darstellung der Folge $S_1 .. S_k$:

Datenstruktur D: Knoten + Pointer.

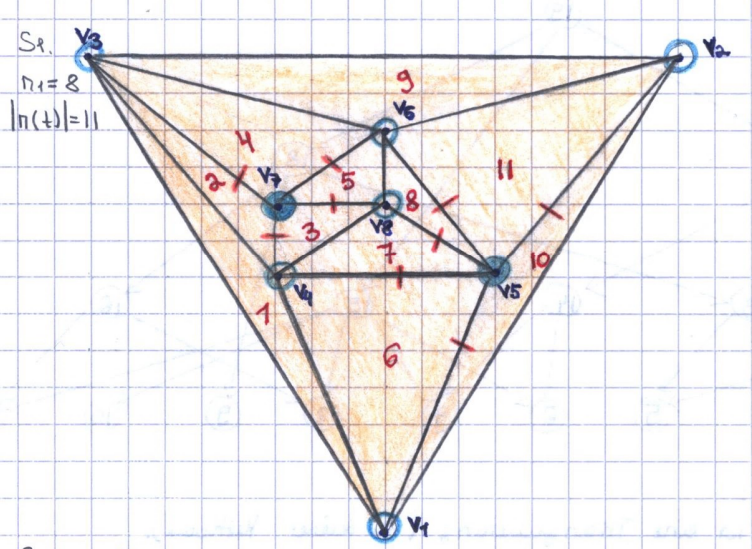
Knoten v speichert ein Dreieck t $\quad v = n(t)$.

Triangulierung S: Pkte, Kanten, Flächen

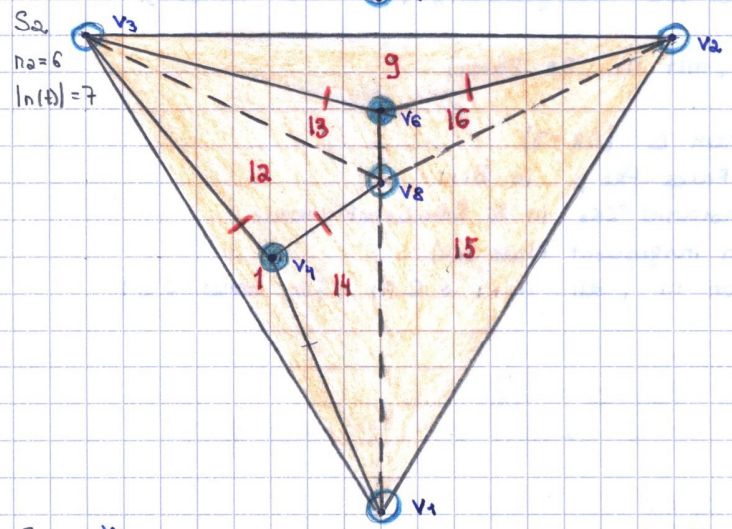
$n_i = |S_i|$ = # der Pkte.

1. $i \leftarrow 1$, $S_1 \leftarrow G$.
2. $\forall$ Dreiecke t in G do
3. erzeuge einen Knoten n(t)
4. od.
5. while $|S_i| > 3$ do
6. berechne unabh. Knotenmenge I in S_i mit mind. $\left\lceil \frac{1}{12} \left(\frac{n}{2} - 3 \right) \right\rceil$ Knoten, die nicht am Rand liegen und maximal Grad 11 haben.
7. Entferne die Knoten von I und ihre angegrenzten Kanten aus S_i
8. Triangulire den entstandenen Graphen und nenne das Resultat S_{i+1}.
9. $\forall$ Dreiecke t von S_{i+1}, die nicht in S_i enthalten sind (d.h. neue Dreiecke) do
10. erzeuge Knoten n(t)
11. $\forall$ Knoten n(t') mit $t' \in S_i$ und t' schneidet t do
12. erzeuge einen Pointer $n(t) \longrightarrow n(t')$
13. od
14. od
15. $i \leftarrow i+1$
16. od

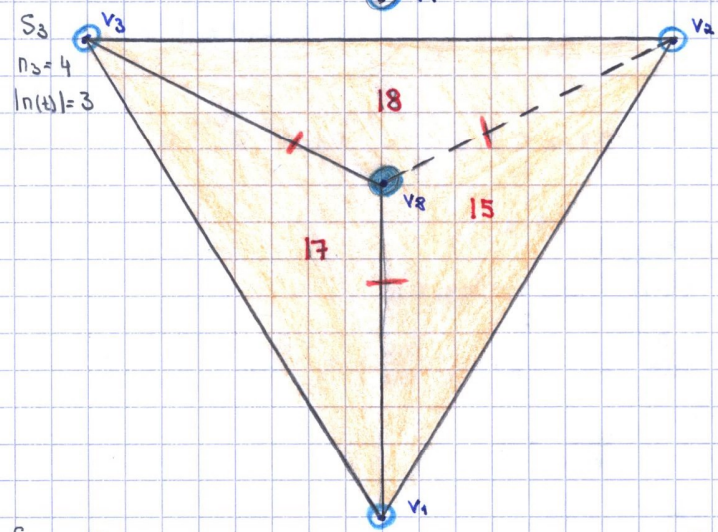
⟶ 3.4.5.11. Beispiel:



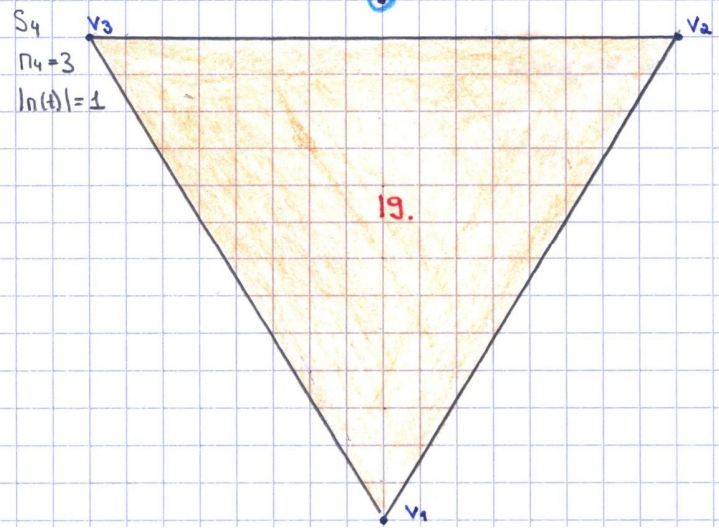
i = 1
I = { v5, v7 }
n(1) ... n(11) n(16)
neu Dreiecke: n(12), n(13), n(14), n(15)
n(12) ⟶ n(2), n(4), n(3), n(5)
n(13) ⟶ n(4), n(5)
n(14) ⟶ n(6), n(7)
n(15) ⟶ n(10), n(6), n(7), n(8), n(11)
n(16) ⟶ n(11), n(8)



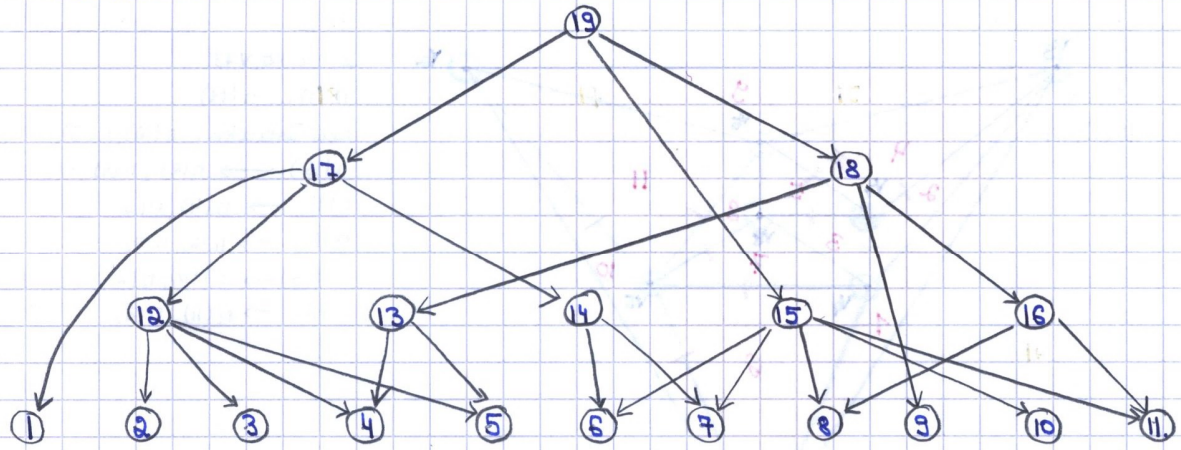
i = 2
I = { v6, v4 }
neue Dreiecke: n(16), n(17), n(18)
n(17) ⟶ n(1), n(14), n(12)
n(18) ⟶ n(13), n(9), n(16)



i = 3
I = { v8 }
neues Dreieck: n(19)
n(19) → n(15), n(17), n(18)



Datenstruktur D:



3.4.5.12. Zusammenfassung:

Planare Unterteilung G ist eine Triangulierung (bei dieser Methode).
(auch Rand ist Δ).

Ziel: Folge $S_1 .. S_R$ (mit $n_1 .. n_R$ Pkten)

✓ 1) $S_1 = G$

✓ 2) S_R besteht aus einem Δ ($n_R = 3$)

3) Lokalisierung von Frage-Pkt p in S_{i+1}
⇒ p kann in konstanter Zeit in S_i lokalisiert werden.

4) $S_1 .. S_R$ brauchen insgesamt Platz $O(n)$

5) n_{i+1} ist Bruchteil von n_i, d.h. $n_{i+1} \leq c \cdot n_i$ für konst $c \leq 1$
⇒ $R = O(\log n)$.

Es gilt: $n_{i+1} < n_i \ \forall i$

⇒ $n_{i+1} < c_i \cdot n_i$ für $c_i > \frac{n_{i+1}}{n_i}$

Ferner: $\frac{n_{i+1}}{n_i} < 1$ ⇒ c_i wählbar als $\frac{n_{i+1}}{n_i} < c_i < 1$

Gilt $\forall i$ und alle $c_i < 1$
⇒ wähle $c := c_R$ $\quad [\, n_3 < \underbrace{c_3}_{<1} \cdot n_2 < \underbrace{c_3 \cdot c_2}_{<1} \cdot n_1 < c_3 \cdot n_1 \,]$

⇒ $n_{i+1} < c \cdot n_i$

⇒ $3 = n_R < c \cdot n_{R-1} < c^2 \cdot n_{R-2} < \ldots < c^{R-1} \cdot n_1 = c^{R-1} \cdot n$

⇒ $n > \frac{3}{c^{R-1}}$

⇒ $\log_2 n > \log_2 3 + \log_2 \left(\frac{1}{c}\right)^{R-1}$

⇒ $\log_2 n > \frac{\ln 3}{\ln 2} + (R-1) \cdot (\underbrace{\log_2 1}_{=0} - \log_2 c)$

⇒ $R - 1 < \frac{\log n - \frac{\ln 3}{\ln 2}}{-\frac{\ln c}{\ln 2}} = \log n \cdot \left(-\frac{\ln c}{\ln 2}\right) + \frac{\ln 3}{\ln c}$

⇒ $R < \underbrace{-\frac{\ln c}{\ln 2}}_{\substack{>0 \text{ weil } c<1 \\ \text{const}}} \cdot \log n + \underbrace{\frac{\ln 3}{\ln c} + 1}_{\text{const}}$

⇒ $R \leq M \cdot \log n$ mit $M = \text{konst}$

⇒ $R = O(\log n)$.

3.4.5.13. Algorithmus für Point-Location:

Eingabe: Pkt q
Datenstruktur $\mathcal{D}$.

$\mathcal{D}$.search(q) liefert Dreieck von G, das q enthält.

if q außerhalb Dreieck i.d. Wurzel then
  Aussage: "äußeres Gebiet von G".
else
  $v \leftarrow$ Wurzel
  while v kein Blatt (dh. $\mathcal{D}$.outdeg(v)>0) do
    forall Knoten u mit $\exists$ Pointer $v \to u$ do
      if q innerhalb Dreieck von u then
        $v \leftarrow u$;
      fi
    od
  od
  Ausgabe: Dreieck von v
fi

Schleifeninvariante: Während der Ausführung der while-Schleife gilt stets, dass $q \in$ Dreieck von v.

Bemerkung: Platz:

$$\#\text{Knoten} = n_1 + n_2 + \dots + n_R \leq$$
$$\leq n_1 + c \cdot n_1 + \dots + c^{R-1} \cdot n_1 =$$
$$= n_1 \sum_{\nu=0}^{R-1} c^\nu = n_1 \cdot \frac{c^R - 1}{c-1} = n \cdot \underbrace{\frac{c^R - 1}{c-1}}_{\text{const}} = \mathcal{O}(n)$$

⟶ 3.4.5.13. Algorithmus für Point Location.

Eingabe: Pkt q, Datenstruktur D.
$D.search(q)$ liefert Dreieck von G, dass q enthält
if q außerhalb Dreieck in der Wurzel then
    Aussage: äußeres Gebiet von G.
else
    $v \leftarrow$ Wurzel
    while v kein Blatt (dh. $D.outdeg(v) > 0$) do
        forall Knoten u mit $\exists$ Pointer $v \rightarrow u$ do
            if q innerhalb Dreieck von u then
                $v \leftarrow u$;
            fi
        od
    od
    Ausgabe: Dreieck von v
fi.

Schleifen-Invariante: Während der Ausführung der while-Schleife gilt stets, dass $q \in$ Dreieck von v.

Knoten $\hat{=}$ Objekt einer Klasse "Dreieck".
→ mit orient. Teste ...

Kapitel IV: Bewegungsplanung i.d. Ebene.

4.1. Einführung:

4.1.1. Allgemeines Problem:

Geg: Eine Szene S von Hindernissen (Polygone, Segmente, ...), ein Roboter R (Polygon, Kreisscheibe, ...) und zwei Pkte A, B.

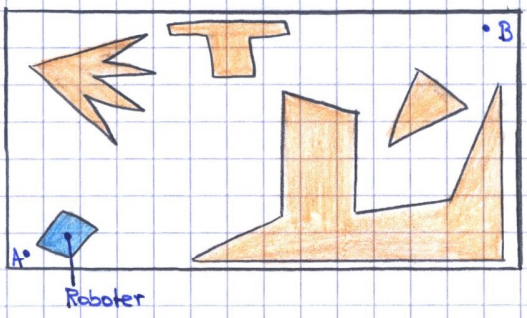
Aufgabe: Beantworte die Frage, ob R von A nach B bewegt werden kann, ohne mit Hindernissen aus S zu kollidieren bzw. gib einen kollisionsfreien Weg an.

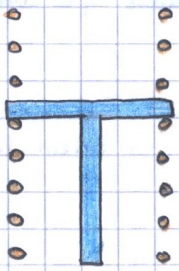
→ Piano Moves Problem.

4.1.2. Bemerkungen:

1) Wir behandeln zwei Spezialfälle:
   - S ist Menge von Kurvensegmenten u. R ist eine Kreisscheibe.
   - S ist Menge von konvexen Polygone, R ist konvexes Polygon.
2) Bewegung: nur Translationen.

Es gibt Vielzahl weiterer Varianten:
- Rotationen erlaubt
- Kürzester Weg.
- Sichester Weg.





4.2. Problem 1: R ist Kreis und S Menge von Segmenten.

4.2.1. Idee: Versuche den sichersten Weg zu finden, dh. R hält immer max. möglichen Abstand zu den Segmenten.

4.2.2 Frage: Was sind die Orte mit maximalen Abstand zu allen Hindernissen?

4.2.3 Antwort: Alle Pkte auf (Knoten & Kanten des Voronoi-Diagramms von S. VD(S)

4.2.4. Voronoi Diagramm von Segmenten i.d. Praxis:

→ annähern durch setzen von Hilfspkten (genügend dicht) und Berechnung des VD's dieser Pkte.

Anschließend Entfernung aller Voronoi-Kanten, die von Segmenten geschnitten werden.

4.2.5 Definition: $\forall p \in \mathbb{R}^2$ ist

a) Freiheit (p) := minimaler Abstand von p zu allen Hindernissen.

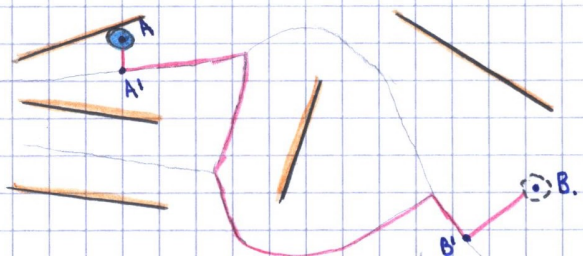
b) p heißt frei $\Leftrightarrow$ Freiheit (p) $\geq r$, wobei r = Radius von R.
(dh. Kreisscheibe R kann mit Mittelpkt auf p plaziert werden, ohne ein Segment zu schneiden).

c) $FP := \{p \in \mathbb{R}^2 : p \text{ ist frei}\}$.

4.2.6. Idee für Algorithmus:

- Bewege R von Anfangsposition A (Ann. freie Position) auf nächste Voronoi Kante (→ Position A').
- Bewege R auf Voronoi-Kanten, die mindestens Freiheit r haben, so nah wie möglich an B heran. (→ Position B')
- Bewege R von B' nach B (Ann. B ist frei).

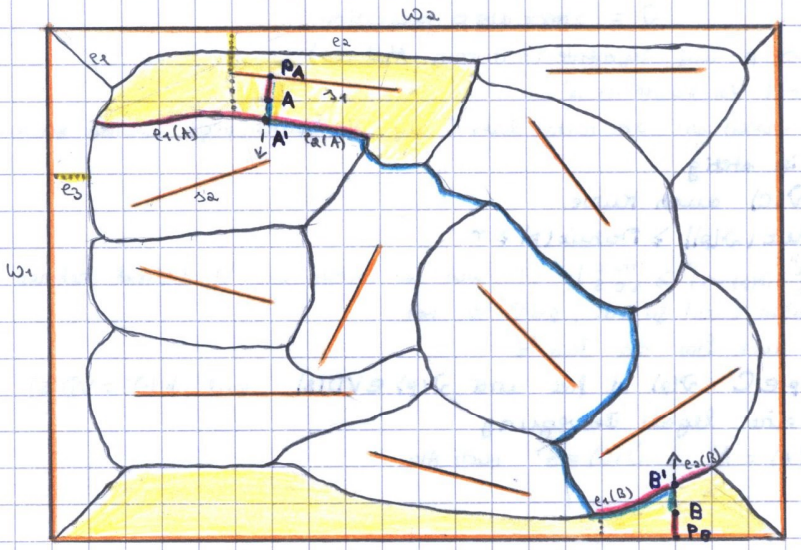
4.2.7 Beispiel:



4.2.8 Algorithmus:

1. Konstruiere VD(S). Speichere für jede Kante die beiden Orte, die e definieren. (entweder Segmentendpkte oder Segmentwand).
   Und Freiheit(e) := minimaler Abstand eines Pktes auf e zu einem Segment.
   (Freiheit(e) ≥ r ⇒ R kann über e bewegt werden).
   ↑ Preprocessing.
2. Bestimme für A und B jeweils den nächsten Pkt P_A bzw. P_B auf einem Segment durch lineare Suche auf S (→ Point Location)
   (P_A, P_B sind Schnittpkte des Lots von A bzw. B auf nächstes Segment).
   Dh: Betrachte Lots von A bzw. B auf alle Segmente in S, suche nach einem Lot, der eine Voronoi-Kante schneidet → lin. Suche.
3. A' ← erster Schnittpkt des Strahls $\overrightarrow{P_A A}$ mit VD(S)
   B' ← erster Schnittpkt des Strahls $\overrightarrow{P_B B}$ mit VD(S)
   Falls A' bzw. B' keine Voronoi-Knoten, dann mache sie zu künstlichen Knoten durch Spalten der entsprechenden Kante e in e_1, e_2.
   → Berechne Freiheit(e_1), Freiheit(e_2)!
4. Suche einen Pfad P in VD(S) von A' nach B' mit Freiheit(e) ≥ r ∀ e ∈ P.
   z.B. durch DFS oder BFS.
   Falls kein solcher Pfad existiert, melde "keine Lösung".
   → STOP.
5. Der Weg A → P → B ist eine mögliche Bewegung.

4.2.9 Bsp:



1. für e_1 speichere: (ω_1, ω_2) — Freiheit(e_1) = 0 cm
   für e_2 speichere: (s_1, ω_2) — Freiheit(e_2) = 0,5 cm
   für e_3 speichere: (Endpkt von s_2, ω_1) — Freiheit(e_3) = 0,5 cm. usw. ...
2. Berechne Lots von A und B auf alle Segmente (ind. Ränder)
   Von den berechneten suche die mit dem kleinsten Abstand aus (lin. Suche)
   Dadurch wissen wir in welcher Region sich Ort A und B befinden (point Location)
   Im Bsp. sind die jeweiligen Regionen gelb.
3. Teile e(A) in $e_1(A)$ und $e_2(A)$ und e(B) in $e_1(B)$ und $e_2(B)$
   Freiheit($e_1(A)$) = 0,6 cm, Freiheit($e_2(A)$) = 0,7 cm
   Freiheit($e_1(B)$) = 0,3 cm, Freiheit($e_2(B)$) = 1 cm.
4. ∀ e ∈ blau gilt Freiheit(e) ≥ r

4.2.10. Lemma:

1) Falls A bzw. B frei ist $\Rightarrow$ $\exists$ Bewegung von A nach A' bzw. von B nach B'.
2) $\exists$ Bewegung von A nach B $\Leftrightarrow$ $\exists$ Bewegung von A' nach B', die nur Voronoi-Kanten besitzt.

Beweis:

1). Sei $VR(s)$ die Voronoi-Region, die A enthält
$\Rightarrow p_A \in S_v \in S$
Seien A, B frei $\Rightarrow$ Freiheit(A) $\geq r \wedge$ Freiheit(B) $\geq r$
$\text{Freiheit}(A) := \min_{s \in S} |A - s|$, $\text{Freiheit}(B) := \min_{s \in S} |B - s|$

Es gibt: $r \leq \text{Freiheit}(A) \leq \text{Freiheit}(p) \quad \forall p \in \overline{AA'}$
$\Rightarrow$ Es gibt eine legale Bewegung von R von A nach A'
weil $\text{Freiheit}(p) \geq r \quad \forall p \in \overline{AA'}$.
$B \to B'$ analog.

2) „$\Leftarrow$": $\exists$ Bewegung von A' nach B', die nur Voronoi-Kanten besitzt.
Beh folgt nach Teil 1), falls A und B frei sind. (?)

„$\Rightarrow$": z.z. $\exists$ Bew von A' nach B', die nur Vor. K. besitzt.
Eine beliebige legale Bewegung von A nach B wird durch eine Kurve $C \subset \mathbb{R}^2$ definiert. (oBdA eine einfache Kurve).
$\forall p \in C$ ist p frei, dh. $\text{Freiheit}(p) \geq r$
Betr. Abb. $\vec{v}: \mathbb{R}P \to VD(S)$ bea: $C \subset \mathbb{R}P$ $(D(\vec{v}) = C)$
die durch Schritte 2 bis 3 des Algorithmus definiert ist.
$\vec{v}(p) = p'$. $\vec{v}$ = Streckung o projektion
Betr. Lots von p auf Segmente (mit min. Abstand)
Schnittpkt mit Vor.kante ist p'.
Bea: Jetzt wissen wir noch nicht, dass C auf Vor.kanten liegt, wollen genau das zeigen

$\Rightarrow$ a). $\vec{v}$ ist stetig
$\Rightarrow \vec{v}(C)$ auch Kurve.
b) $\text{Freiheit}(\vec{v}(p)) \geq \text{Freiheit}(p) \geq r$
$\min''_{s \in S} |\vec{v}(p) - s| \geq \min_{s \in S} |p - s|$ weil wir immer den kl. Abstand nehmen.
p haben zuerst gesucht, $p \in C \Rightarrow$ Beh.
$\geq r$ auch klar da Bew. $\exists$.
$\Rightarrow \forall p \in C$: $\vec{v}(p)$ ist frei und $\vec{v}(p) \in VD(S)$ weil $W(\vec{v}) = VD(S)$
$\Rightarrow \vec{v}$ ist eine legale Bewegung
Bea: $\vec{v}(A) = A' \wedge \vec{v}(B) = B'$ nach Alg.

4.2.11. Laufzeit:

Schritt 1: $O(n \log n)$ durch Plane Sweep.
Schritt 2: $O(n)$ lin. Suche in S
Schritt 3: $O(n)$ lin Such auf $VD(S)$, $\overrightarrow{p_A A}$ berechnen $\to O(1)$, A' berechnen $\to O(1)$, e_1, e_2 aufteilen und $\text{Freiheit}(e_1)$, $\text{Freiheit}(e_2)$ Berechnen $\to O(1)$
Überprüfen ob A' bzw B' Vor. Knoten $\to O(n)$, denn lin. Suche bei den gespeicherten Voronoiknoten.
Schritt 4: $O(n)$ Pfadsuche in $VD(S)$ (ZB DFS)
Schritt 5: $O(n)$ Ausgabe des Pfades.

4.2.12. Satz: Bewegungsplanungsproblem für eine Kreisscheibe in einer Szene von n Liniensegmenten in der Ebene kann in Zeit $O(n \log n)$ und Platz $O(n)$ gelöst werden.
(Platz $O(n)$: weil in einzelnen 5 Schritten immer Platz $O(n)$.)

Beweis: s.o.

Kapitel IV: Bewegungsplanung i.d. Ebene.

4.1. Einführung:

4.1.1. Allgemeines Problem:

Geg: Eine Szene S von Hindernissen (Polygone, Segmente, ...), ein Roboter R (Polygon, Kreisscheibe, ...) und zwei Pkte A, B.

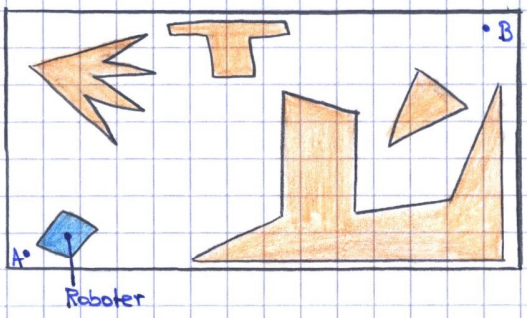
Aufgabe: Beantworte die Frage, ob R von A nach B bewegt werden kann, ohne mit Hindernissen aus S zu kollidieren bzw. gib einen kollisionsfreien Weg an.

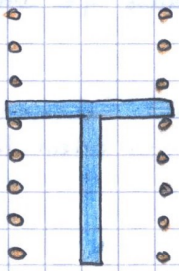
→ Piano Moves Problem.

4.1.2. Bemerkungen:

1) Wir behandeln zwei Spezialfälle:
   - S ist Menge von Kurvensegmenten u. R ist eine Kreisscheibe.
   - S ist Menge von konvexen Polygone, R ist konvexes Polygon.
2) Bewegung: nur Translationen.

Es gibt Vielzahl weiterer Varianten:
- Rotationen erlaubt
- Kürzester Weg.
- Sichester Weg.





4.2. Problem 1: R ist Kreis und S Menge von Segmenten.

4.2.1. Idee: Versuche den sichersten Weg zu finden, dh. R hält immer max. möglichen Abstand zu den Segmenten.

4.2.2 Frage: Was sind die Orte mit maximalen Abstand zu allen Hindernissen?

4.2.3 Antwort: Alle Pkte auf (Knoten & Kanten des Voronoi-Diagramms von S. VD(S)

4.2.4. Voronoi Diagramm von Segmenten i.d. Praxis:

→ annähern durch setzen von Hilfspkten (genügend dicht) und Berechnung des VD's dieser Pkte.

Anschließend Entfernung aller Voronoi-Kanten, die von Segmenten geschnitten werden.

4.2.5 Definition: $\forall p \in \mathbb{R}^2$ ist

a) Freiheit (p) := minimaler Abstand von p zu allen Hindernissen.

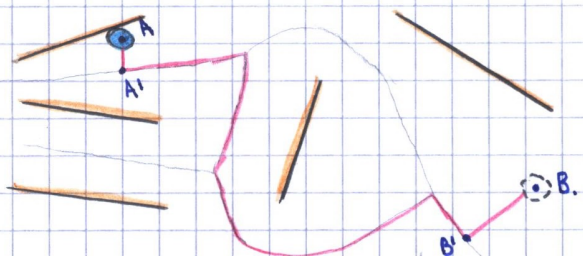
b) p heißt frei $\Leftrightarrow$ Freiheit (p) $\geq r$, wobei r = Radius von R.
(dh. Kreisscheibe R kann mit Mittelpkt auf p plaziert werden, ohne ein Segment zu schneiden).

c) $FP := \{p \in \mathbb{R}^2 : p \text{ ist frei}\}$.

4.2.6. Idee für Algorithmus:

- Bewege R von Anfangsposition A (Ann. freie Position) auf nächste Voronoi Kante (→ Position A').
- Bewege R auf Voronoi-Kanten, die mindestens Freiheit r haben, so nah wie möglich an B heran. (→ Position B')
- Bewege R von B' nach B (Ann. B ist frei).

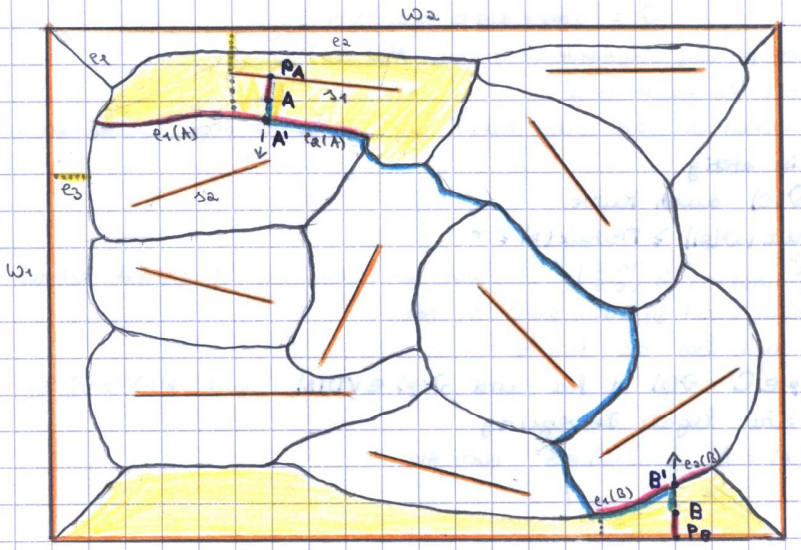
4.2.7 Beispiel:



4.2.8 Algorithmus:

1. Konstruiere VD(S). Speichere für jede Kante die beiden Orte, die e definieren. (entweder Segmentendpkte oder Segmentwand).
   Und Freiheit(e) := minimaler Abstand eines Pktes auf e zu einem Segment.
   (Freiheit(e) ≥ r ⇒ R kann über e bewegt werden).
   ↑ Preprocessing.
2. Bestimme für A und B jeweils den nächsten Pkt P_A bzw. P_B auf einem Segment durch lineare Suche auf S (→ Point Location)
   (P_A, P_B sind Schnittpkte des Lots von A bzw. B auf nächstes Segment).
   Dh: Betrachte Lots von A bzw. B auf alle Segmente in S, suche nach einem Lot, der eine Voronoi-Kante schneidet → lin. Suche.
3. A' ← erster Schnittpkt des Strahls $\overrightarrow{P_A A}$ mit VD(S)
   B' ← erster Schnittpkt des Strahls $\overrightarrow{P_B B}$ mit VD(S)
   Falls A' bzw. B' keine Voronoi-Knoten, dann mache sie zu künstlichen Knoten durch Spalten der entsprechenden Kante e in e_1, e_2.
   → Berechne Freiheit(e_1), Freiheit(e_2)!
4. Suche einen Pfad P in VD(S) von A' nach B' mit Freiheit(e) ≥ r ∀ e ∈ P.
   z.B. durch DFS oder BFS.
   Falls kein solcher Pfad existiert, melde "keine Lösung".
   → STOP.
5. Der Weg A → P → B ist eine mögliche Bewegung.

4.2.9 Bsp:



1. für e_1 speichere: (ω_1, ω_2) — Freiheit(e_1) = 0 cm
   für e_2 speichere: (s_1, ω_2) — Freiheit(e_2) = 0,5 cm
   für e_3 speichere: (Endpkt von s_2, ω_1) — Freiheit(e_3) = 0,5 cm. usw. ...
2. Berechne Lots von A und B auf alle Segmente (ind. Ränder)
   Von den berechneten suche die mit dem kleinsten Abstand aus (lin. Suche)
   Dadurch wissen wir in welcher Region sich Ort A und B befinden (point Location)
   Im Bsp. sind die jeweiligen Regionen gelb.
3. Teile e(A) in $e_1(A)$ und $e_2(A)$ und e(B) in $e_1(B)$ und $e_2(B)$
   Freiheit($e_1(A)$) = 0,6 cm, Freiheit($e_2(A)$) = 0,7 cm
   Freiheit($e_1(B)$) = 0,3 cm, Freiheit($e_2(B)$) = 1 cm.
4. ∀ e ∈ blau gilt Freiheit(e) ≥ r

4.2.10. Lemma:

1) Falls A bzw. B frei ist $\Rightarrow$ $\exists$ Bewegung von A nach A' bzw. von B nach B'.
2) $\exists$ Bewegung von A nach B $\Leftrightarrow$ $\exists$ Bewegung von A' nach B', die nur Voronoi-Kanten besitzt.

Beweis:

1). Sei $VR(s)$ die Voronoi-Region, die A enthält
$\Rightarrow p_A \in S_v \in S$
Seien A, B frei $\Rightarrow$ Freiheit(A) $\geq r \wedge$ Freiheit(B) $\geq r$
$\text{Freiheit}(A) := \min_{s \in S} |A - s|$, $\text{Freiheit}(B) := \min_{s \in S} |B - s|$

Es gibt: $r \leq \text{Freiheit}(A) \leq \text{Freiheit}(p) \quad \forall p \in \overline{AA'}$
$\Rightarrow$ Es gibt eine legale Bewegung von R von A nach A'
weil $\text{Freiheit}(p) \geq r \quad \forall p \in \overline{AA'}$.
$B \to B'$ analog.

2) „$\Leftarrow$": $\exists$ Bewegung von A' nach B', die nur Voronoi-Kanten besitzt.
Beh folgt nach Teil 1), falls A und B frei sind. (?)

„$\Rightarrow$": z.z. $\exists$ Bew von A' nach B', die nur Vor. K. besitzt.
Eine beliebige legale Bewegung von A nach B wird durch eine Kurve $C \subset \mathbb{R}^2$ definiert. (oBdA eine einfache Kurve).
$\forall p \in C$ ist p frei, dh. $\text{Freiheit}(p) \geq r$
Betr. Abb. $\vec{v}: \mathbb{R}P \to VD(S)$ bea: $C \subset \mathbb{R}P$ $(D(\vec{v}) = C)$
die durch Schritte 2 bis 3 des Algorithmus definiert ist.
$\vec{v}(p) = p'$. $\vec{v}$ = Streckung o projektion
Betr. Lots von p auf Segmente (mit min. Abstand)
Schnittpkt mit Vor.kante ist p'.
Bea: Jetzt wissen wir noch nicht, dass C auf Vor.kanten liegt, wollen genau das zeigen

$\Rightarrow$ a). $\vec{v}$ ist stetig
$\Rightarrow \vec{v}(C)$ auch Kurve.
b) $\text{Freiheit}(\vec{v}(p)) \geq \text{Freiheit}(p) \geq r$
$\min''_{s \in S} |\vec{v}(p) - s| \geq \min_{s \in S} |p - s|$ weil wir immer den kl. Abstand nehmen.
p haben zuerst gesucht, $p \in C \Rightarrow$ Beh.
$\geq r$ auch klar da Bew. $\exists$.
$\Rightarrow \forall p \in C$: $\vec{v}(p)$ ist frei und $\vec{v}(p) \in VD(S)$ weil $W(\vec{v}) = VD(S)$
$\Rightarrow \vec{v}$ ist eine legale Bewegung
Bea: $\vec{v}(A) = A' \wedge \vec{v}(B) = B'$ nach Alg.

4.2.11. Laufzeit:

Schritt 1: $O(n \log n)$ durch Plane Sweep.
Schritt 2: $O(n)$ lin. Suche in S
Schritt 3: $O(n)$ lin Such auf $VD(S)$, $\overrightarrow{p_A A}$ berechnen $\to O(1)$, A' berechnen $\to O(1)$, e_1, e_2 aufteilen und $\text{Freiheit}(e_1)$, $\text{Freiheit}(e_2)$ Berechnen $\to O(1)$
Überprüfen ob A' bzw B' Vor. Knoten $\to O(n)$, denn lin. Suche bei den gespeicherten Voronoiknoten.
Schritt 4: $O(n)$ Pfadsuche in $VD(S)$ (ZB DFS)
Schritt 5: $O(n)$ Ausgabe des Pfades.

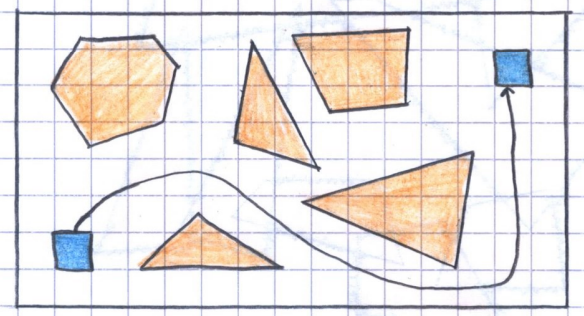
4.2.12. Satz: Bewegungsplanungsproblem für eine Kreisscheibe in einer Szene von n Liniensegmenten in der Ebene kann in Zeit $O(n \log n)$ und Platz $O(n)$ gelöst werden.
(Platz $O(n)$: weil in einzelnen 5 Schritten immer Platz $O(n)$.)

Beweis: s.o.

4.3 Problem 2: R ist konv. Polygon und S Menge von konv. Polygonen.

4.3.1. Problemschilderung: S ist eine Szene von m konvexen Polygonen $P_1 .. P_m$ mit $n := \sum_{i=1}^{m}$ # Ecken von P_i und $P_i \cap P_j = \emptyset \quad \forall i \neq j$

R (Roboter) ist konvexes Polygon mit c Ecken (c = Konstante).
Bewegungen: Nur Translationen von R (keine Rotationen).



4.3.2 Anm:

- S ist in einem Rechteck eingeschlossen, z.B. Dummi-Hindernisse.
- Sei p ein beliebiger Referenzpkt im Inneren von R, dann ist die Position von R durch die Position von p im $\mathbb{R}^2$ definiert.

4.3.3. Idee:

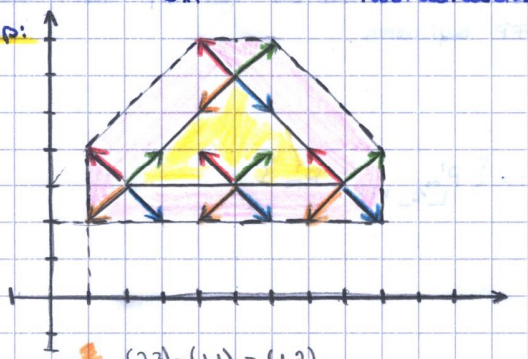
Reduziere das Problem auf das Bewegungsplanungsproblem eines pktförmigen Roboters (p) in einer komplizierten Szene S'.
Dazu blähen wir alle Hindernisse $P_1 .. P_m$ auf.
D.h. alle Hindernisse werden um das Maß vergrößert, auf das der Roboter verkleinert werden muß, um ein Pkt zu werden.

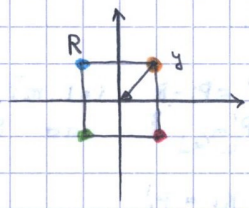
4.3.4. Konstruktion von P_i' (= aufgeblähtes Hindernis).

$P_i' = P_i - R =$ "Minkowski-Differenz".

$= \{ x-y : \underbrace{x \in P_i}_{\in \mathbb{R}^2} \wedge \underbrace{y \in R}_{\text{im Koordinatensystem des Roboters.}} \}$

Bsp:





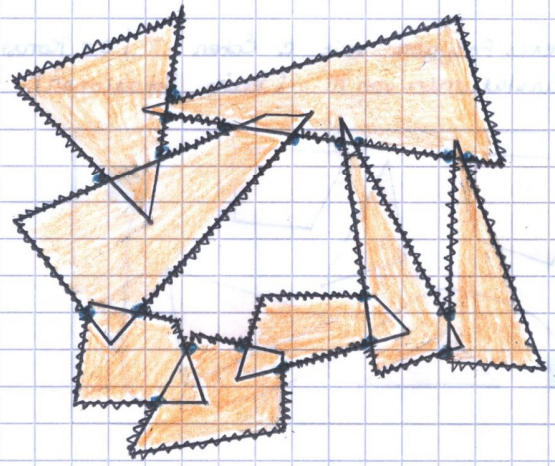
Übung ! 8.1.

- $(2,3) - (1,1) = (1,2)$
- $(2,3) - (-1,1) = (3,2)$
- $(2,3) - (1,-1) = (1,4)$
- $(2,3) - (-1,-1) = (3,4)$
- Kommt dazu

Betrachte die Szene $S' = \{P_1' \ldots P_m'\}$ mit $P_i' = P_i - R$, $i = 1..m$.
Dann ist die Menge der freien Plazierungen von R in S' (bzgl. des Referenzpktes p):
$FP = \mathbb{R}^2 \setminus \bigcup_{i=1}^{m} P_i'$

d.h. das Komplement der Vereinigung aller aufgeblähten Hindernisse.
FP ist i.a. nicht mehr zushgd.
D.h. FP ist ein u.U. nicht zushgdes Gebiet der Ebene, begrenzt durch Polygonzüge.
Es ist offensichtlich, daß eine Bewegung von A nach B genau dann möglich ist, wenn A und B in derselben Zusammenhangskomponente von FP liegen.

4.3.5. Beispiel:



n = # alle Ecken
Hier: 28 = n
Auf dem Rand dann: 19 < n.

4.3.6 Lemma: der Rand oder die Kontur von FP hat ~~Größe~~ O(n)
Vermutung: # alle Ecken auf dem Rand der Vereinigung ist O(n).

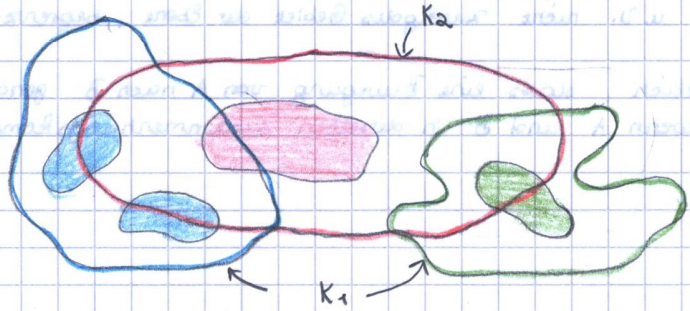
4.3.7 Satz: Seien $P_1..P_m$ paarweise disjunkte, konvexe Polygone mit insgesamt n Ecken und R ein konvexes Polygon mit konstant vielen Ecken.
Dann hat Kontur = $K = \underbrace{\bigcup_{i=1}^{m} (P_i - R)}_{\text{Polygonfläche mit Löchern}}$ O(n) Ecken.

Beweis später.

4.3.8 Algorithmus:
Um Problem 2 zu lösen, müssen wir FP und seine Zusammenhangskomponenten berechnen. Dazu berechnen wir zunächst die Kontur (K) von FP, dh. die Menge der Kanten, die den Rand von FP definieren.
⇝ Divide & Conquer.
Algorithmus:
1) Berechne $S' = \{P_i' = P_i - R : i = 1..m\}$.
2) Sei $S_1 := \{P_1' \dots P'_{\lfloor m/2 \rfloor}\}$ und $S_2 := \{P'_{\lfloor m/2 \rfloor + 1}, .. P_m'\}$

Falls $|S_1| = 1 \Rightarrow K_1 = P_1'$
Falls $|S_2| = 1 \Rightarrow K_2 = P'_{\lfloor m/2 \rfloor + 1}$
sonst:
Berechne rekursiv:
K_1 = Kontur von S_1
K_2 = Kontur von S_2
3) Berechne K = Kontur von S' durch Überlagerung von K_1 und K_2.

4.3.9 Laufzeit: Schritt 1: O(n)
Dazu nötig, dass Minkowski-Differenz von konvexen Polygonen in linearer Zeit.
⇝ Übung
Rest: $T(n) = 2 \cdot T(n/2) +$ Zeit für Mischen
↑ Vereinigung von zwei Konturen K_1 und K_2.



4.3.10. Plane Sweep Alg. zum Mischen von zwei Konturen A und B.

→ 4.3.10.1. Problemstellung:

Eingabe: Zwei Konturen A, B in den Ecken durch zwei Mengen von Liniensegmenten.

Ausgabe: Menge von Segmenten, die die Kanten von $A \cup B$ definieren.

→ 4.3.10.2 Definition:

Ein Segment heißt sichtbar, wenn es zur Ausgabe gehört, d.h. auf Rand von $A \cup B$ liegt.

→ 4.3.10.3. Idee:

Wir modifizieren den Plane Sweep Alg. zum Linienschnitt.

Y-Struktur: Folge der von SL geschnittenen Segmenten (von unten nach oben sortiert). Wir speichern für jedes Segment in der Y-Struktur, ob es zur Zeit sichtbar oder unsichtbar ist.

Außerdem speichern wir für jedes Paar (s_1, s_2) von in Y benachbarten Segmenten, ob und von wem das Gebiet zwischen s_1 und s_2 zur Zeit überdeckt wird.

$$cover(s_1,s_2) = \begin{cases} \emptyset, & \text{nicht überdeckt} \\ \{A\}, & \text{von A} \\ \{B\}, & \text{von B} \\ \{A \cup B\}, & \text{von beiden.} \end{cases}$$

→ 4.3.10.4. Aktionen:

1. Linker Endpkt $p \in A$ (B analog) d.h. linke Ecke von zwei Segmenten $s_1, s_2 \in A$.

$s' := Y.succ(s_1)$, $s'' = Y.pred(s_2)$

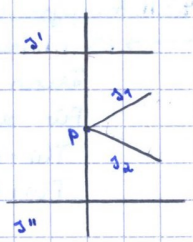
Falls $cover(s', s'') = \emptyset$ oder $cover(s', s'') = A$ dann
  markiere s_1, s_2 als sichtbar
sonst unsichtbar

Falls $A \in cover(s', s'')$ (d.h. bei p beginnt ein Loch) dann:
  $cover(s_1, s_2) \leftarrow cover(s', s'') \setminus \{A\}$
sonst $cover(s_1, s_2) \leftarrow cover(s', s'') \cup \{A\}$.

$cover(s', s_1) \leftarrow cover(s', s'')$

$cover(s_2, s'') \leftarrow cover(s', s'')$

$Y \leftarrow Y \cup \{s_1, s_2\}$.

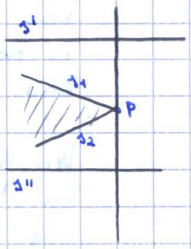


2. Rechter Endpkt $p \in A$ (B analog)

Falls s_1 bzw. s_2 sichtbar, dann Ausgabe der betreffenden Segmente.

$Y \leftarrow Y \setminus \{s_1, s_2\}$

$cover(s', s'') \leftarrow cover(s', s_1)$ $(= cover(s_2, s''))$.



3. Schnittpkt $p = s_1 \cap s_2$

o.Bd.A $s_1 \in A$, $s_2 \in B$

Falls s_1 (s_2) sichtbar ⇒ Ausgabe bis p (splitten!)

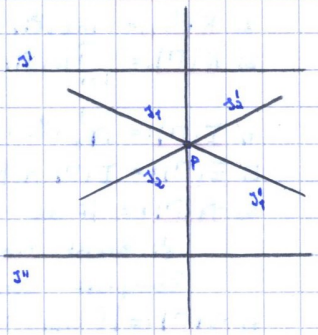
$Y \leftarrow Y \setminus \{s_1, s_2\}$, $Y \leftarrow Y \cup \{s_1', s_2'\}$ in umgekehrter Reihenfolge mit negativer Sichtbarkeitsinformation

$cover(s', s_2') \leftarrow cover(s', s_1)$

$cover(s_1', s'') \leftarrow cover(s_2, s'')$

Falls $A \in cover(s_1, s_2)$ dann
  $cover(s_2', s_1') \leftarrow cover(s_1, s_2) \setminus \{A\}$
sonst $cover(s_2', s_1') \leftarrow cover(s_1, s_2) \cup \{A\}$

Falls $B \in cover(s_1, s_2)$ dann
  $cover(s_2', s_1') \leftarrow cover(s_1, s_2) \setminus \{B\}$
sonst $cover(s_2', s_1') \leftarrow cover(s_1, s_2) \cup \{B\}$.



4. Durchgang $p \in A$ (B analog)

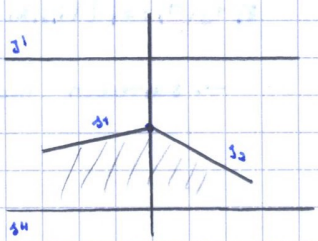
s_1 endet und s_2 beginnt

$Y \leftarrow Y \setminus \{s_1\}$, $Y \leftarrow Y \cup \{s_2\}$

Falls s_1 sichtbar ⇒ Ausgabe.

$Sichtbarkeit(s_2) \leftarrow Sichtbarkeit(s_1)$

$cover(s', s_2) \leftarrow cover(s', s_1)$; $cover(s_2, s'') \leftarrow cover(s_1, s'')$.



⟶ 4.3.10.5 Bem:

a). Es gibt nur Schnittsegmente zwischen Segmenten aus verschiedenen Kurven.

b). Segmente werden an Schnittpkten gespalten (⇝ entweder ganz sichtbar oder ganz unsichtbar.)

c) Die Sichtbarkeitsinformation ändert sich nur bei Schnittpkten (wechselt von sichtbar zu unsichtbar oder umgekehrt).

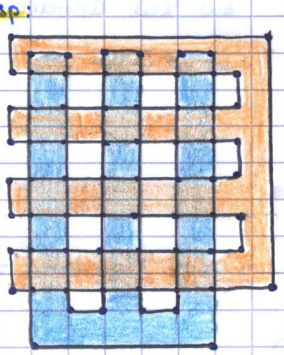
⟶ 4.3.10.6. Lemma:

Zwei Konturen A und B mit insgesamt n vielen Kanten können in Zeit $O((n+s)\log n)$ berechnet werden, wobei s = # Schnittpkten zw. Segmenten in A und B.

Beweis: siehe Segmentschnitt.

⟶ 4.3.10.7 Bem: Im Allg. ist $s = O(n^2)$ ...

⟶ 4.3.10.8 Bsp:



z.Z: $T(n) = O(n\log^2 n)$.

Ind.anf: Beh. gelte für zB $\frac{n}{2}$

Ind.schritt: dann gilt sie auch für n!

$$T(n) \le c_1 n\log n + 2T\left(\tfrac{n}{2}\right) \le c_1 n\log n + 2\cdot c_2 \tfrac{n}{2}\cdot \log^2\left(\tfrac{n}{2}\right) =$$
$$= c_1 n\log n + c_2 n(\log n - \log 2)^2 = c_1 n\log n + c_2 n(\log n - 1)^2 =$$
$$= c_1 n\log n + c_2 n\log^2 n - 2c_2 n\log n + c_2 n =$$
$$= n\log^2 n\left[c_1\cdot\tfrac{1}{\log n} + c_2 - \tfrac{2c_2}{\log n} + \tfrac{c_2}{\log^2 n}\right] =$$
$$= n\log^2 n\cdot\left[\underbrace{\frac{c_1 - 2c_2}{\log n}}_{\to 0} + c_2 + \underbrace{\frac{c_2}{\log^2 n}}_{\to 0}\right] \le$$

$$\le M\cdot n\log^2 n \quad \forall\, n \ge n_0$$

4.3.11. Analyse der Laufzeit:

⟶ 4.3.11.1. Idee: Wir zeigen nun, dass bei unserer Anwendung $s = O(n)$

Dann kostet der Hirschschritt $O(n\log n)$

Für die Gesamtlaufzeit $T(n)$:

Übung 8.

$T(1) = 1$

$T(n) = O(n\log n) + 2\cdot T\left(\frac{n}{2}\right) \quad \Rightarrow T(n) = O(n\log^2 n)$

Um die $O(n)$ Schranke für s zu zeigen, verwenden wir die Tatsache, dass sich die Ränder von jeweils zwei aufgeblähten Hindernissen Q_i und Q_j in höchstens zwei Pkten schneiden.

Bew. Übung.

Wir betrachten ein etwas allgemeineres Problem:

Sei $\Gamma = \gamma_1 .. \gamma_m$ eine Menge einfacher geschlossener Kurven (= Jordankurven) mit $|\gamma_i \cap \gamma_j| \le 2 \;\forall\, i \ne j$. d.h. zwei Kurven schneiden sich höchstens zweimal.

⟶ 4.3.11.2 Definitionen:

1) $Int(\gamma_i) :=$ Innere Regionen von γ_i

2) $K(\gamma_i) :=$ $Int(\gamma_i) \cup \gamma_i$ (also Abschluss)

3) $K(\Gamma) := \bigcup_{i=1}^{m} K(\gamma_i)$

4) $I(\Gamma) := \bigcup_{i\ne j} (\gamma_i \cap \gamma_j)$ (alle Schnittpkte)

5) $E(\Gamma) := I(\Gamma) \cap Rand(K(\Gamma))$

6) $r_1(\Gamma) :=$ # der redundanten Kurven in Γ, wobei:

γ_i heißt redundant, falls $\gamma_i \subset \bigcup_{i\ne j}^{m} K(\gamma_i)$, d.h. γ_i wird komplett von anderen Kurvenflächen überdeckt. also $\Rightarrow E(\Gamma \setminus \{\gamma_i\}) = E(\Gamma)$

7) $r_2(\Gamma) := |\{(i,j) : i \ne j \wedge \gamma_i \cap \gamma_j \ne \emptyset\}|$

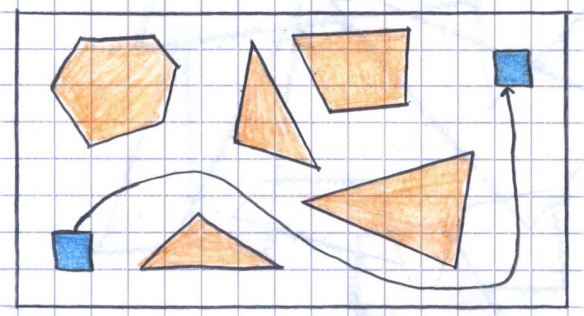
8) $r_3(\Gamma) := |\{(i,j,k) : i,j,k \text{ paarw. versch.}, K(\gamma_i)\cap K(\gamma_j)\cap K(\gamma_k) \ne \emptyset\}|$.

→ schreiben nun einfacher r_i statt $r_i(\Gamma)$ für $i = 1,2,3$.

4.3 Problem 2: R ist konv. Polygon und S Menge von konv. Polygonen.

4.3.1. Problemschilderung: S ist eine Szene von m konvexen Polygonen $P_1 .. P_m$ mit $n := \sum_{i=1}^{m}$ # Ecken von P_i und $P_i \cap P_j = \emptyset \quad \forall i \neq j$

R (Roboter) ist konvexes Polygon mit c Ecken (c = Konstante).
Bewegungen: Nur Translationen von R (keine Rotationen).



4.3.2 Anm:

- S ist in einem Rechteck eingeschlossen, z.B. Dummi-Hindernisse.
- Sei p ein beliebiger Referenzpkt im Inneren von R, dann ist die Position von R durch die Position von p im $\mathbb{R}^2$ definiert.

4.3.3. Idee:

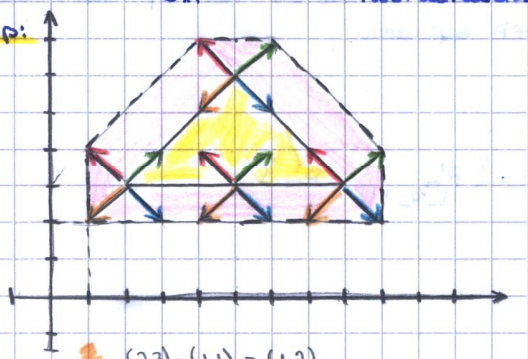
Reduziere das Problem auf das Bewegungsplanungsproblem eines pktförmigen Roboters (p) in einer komplizierten Szene S'.
Dazu blähen wir alle Hindernisse $P_1 .. P_m$ auf.
D.h. alle Hindernisse werden um das Maß vergrößert, auf das der Roboter verkleinert werden muß, um ein Pkt zu werden.

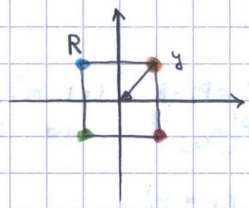
4.3.4. Konstruktion von P_i' (= aufgeblähtes Hindernis).

$P_i' = P_i - R =$ "Minkowski-Differenz".

$= \{ x-y : \underbrace{x \in P_i}_{\in \mathbb{R}^2} \wedge \underbrace{y \in R}_{\text{im Koordinatensystem des Roboters.}} \}$

Bsp:





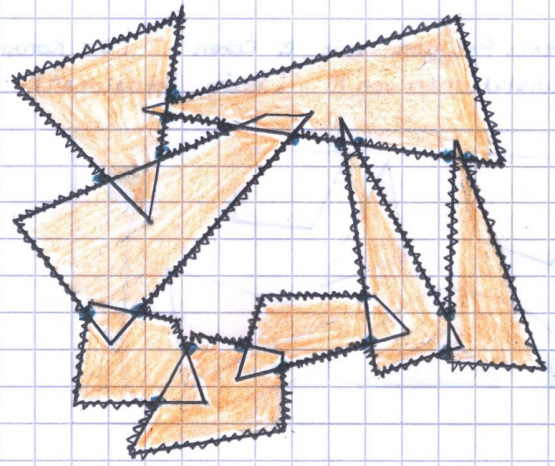
Übung ! 8.1.

- $(2,3) - (1,1) = (1,2)$
- $(2,3) - (-1,1) = (3,2)$
- $(2,3) - (1,-1) = (1,4)$
- $(2,3) - (-1,-1) = (3,4)$
- Kommt dazu

Betrachte die Szene $S' = \{P_1' \ldots P_m'\}$ mit $P_i' = P_i - R$, $i = 1..m$.
Dann ist die Menge der freien Plazierungen von R in S' (bzgl. des Referenzpktes p):
$FP = \mathbb{R}^2 \setminus \bigcup_{i=1}^{m} P_i'$

d.h. das Komplement der Vereinigung aller aufgeblähten Hindernisse.
FP ist i.a. nicht mehr zushgd.
D.h. FP ist ein u.U. nicht zushgdes Gebiet der Ebene, begrenzt durch Polygonzüge.
Es ist offensichtlich, daß eine Bewegung von A nach B genau dann möglich ist, wenn A und B in derselben Zusammenhangskomponente von FP liegen.

4.3.5. Beispiel:



n = # alle Ecken
Hier: 28 = n
Auf dem Rand dann: 19 < n.

4.3.6 Lemma: der Rand oder die Kontur von FP hat ~~Größe~~ O(n)
Vermutung: # alle Ecken auf dem Rand der Vereinigung ist O(n).

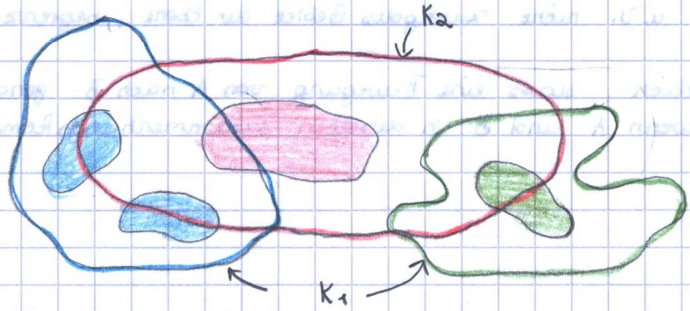
4.3.7 Satz: Seien $P_1..P_m$ paarweise disjunkte, konvexe Polygone mit insgesamt n Ecken und R ein konvexes Polygon mit konstant vielen Ecken.
Dann hat Kontur = $K = \underbrace{\bigcup_{i=1}^{m} (P_i - R)}_{\text{Polygonfläche mit Löchern}}$ O(n) Ecken.

Beweis später.

4.3.8 Algorithmus:
Um Problem 2 zu lösen, müssen wir FP und seine Zusammenhangskomponenten berechnen. Dazu berechnen wir zunächst die Kontur (K) von FP, dh. die Menge der Kanten, die den Rand von FP definieren.
⇝ Divide & Conquer.
Algorithmus:
1) Berechne $S' = \{P_i' = P_i - R : i = 1..m\}$.
2) Sei $S_1 := \{P_1' \dots P'_{\lfloor m/2 \rfloor}\}$ und $S_2 := \{P'_{\lfloor m/2 \rfloor + 1}, .. P_m'\}$

Falls $|S_1| = 1 \Rightarrow K_1 = P_1'$
Falls $|S_2| = 1 \Rightarrow K_2 = P'_{\lfloor m/2 \rfloor + 1}$
sonst:
Berechne rekursiv:
K_1 = Kontur von S_1
K_2 = Kontur von S_2
3) Berechne K = Kontur von S' durch Überlagerung von K_1 und K_2.

4.3.9 Laufzeit: Schritt 1: O(n)
Dazu nötig, dass Minkowski-Differenz von konvexen Polygonen in linearer Zeit.
⇝ Übung
Rest: $T(n) = 2 \cdot T(n/2) +$ Zeit für Mischen
↑ Vereinigung von zwei Konturen K_1 und K_2.



⟶ **4.3.11.3 Satz:** Für $n \geq 3$ gilt: $E(\Gamma) \leq 6n-12$

Beweis:

1. Vorbemerkungen:

Wir nehmen an, dass sich die Kurven in allgemeiner Lage befinden:

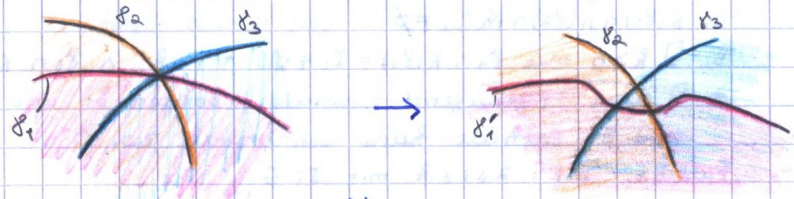
a) Keine drei Kurven schneiden sich in einem Pkt.

b) Zwei Kurven schneiden sich entweder in zwei Pkten oder gar nicht. (oh. keine Berührpkte).

Dies sind keine echten Einschränkungen, da sich jede Menge Γ durch minimale lokale Verfeinerungen in allgemeine Lage versetzen lässt ohne die Zahl der äußeren Ecken $E(\Gamma)$ zu ändern.

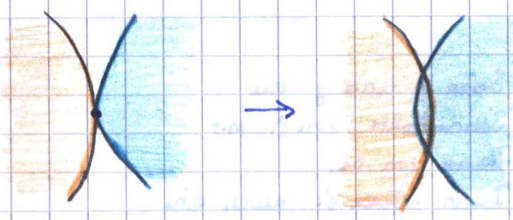
Bsple kurz:

zu a): Seien $\gamma_1, \gamma_2, \gamma_3 \in \Gamma$ mit $\gamma_1 \cap \gamma_2 \cap \gamma_3 = \{p\}$



ersetze γ_1 durch γ_1' ↑ vollkommen verdeckt.

zu b):



Berührpkt p — Zwei Schnittpkte.

2. Wir zeigen nun den Satz durch Induktion über die Tripel (r_1, r_2, r_3) in lexikographischer Ordnung. Die Idee für den Induktionsschritt besteht darin, dass eine der Zahlen r_1, r_2, r_3 durch Strecken oder Verformen einer Kurve γ_i vermindert wird, so dass $E(\Gamma)$ höchstens wächst. (sich nicht verkleinert also).

Induktanf: Wir zeigen die Beh für $r_1 = r_3 = 0, r_2 \geq 0$.

Es gibt: a) keine redundanten Kurven

b) $K(\gamma_i) \cap K(\gamma_j) \cap K(\gamma_k) = \emptyset \quad \forall\ 1 \leq i < j < k \leq n$.

c) $E(\Gamma) = I(\Gamma)$ (folgt aus b).).

$\forall\ \gamma_i$ sieht die Situation wie folgt aus:

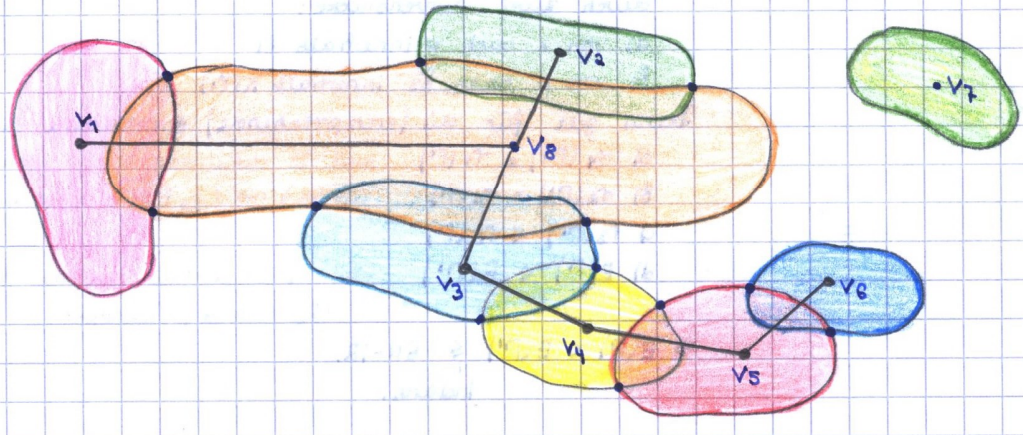
Wir konstruieren einen planaren Graphen $G = (V, E)$ mit $V = \{v_1 .. v_n\}$, wobei v_i Beliebiger Pkt in $K(\gamma_i) \setminus \bigcup_{i \neq j} \text{rel}(\gamma_j)$ und $E = \{(v_i, v_j) : \gamma_i \cap \gamma_j \neq \emptyset\}$

Planarität: Zeichne Kanten als Kurven durch Schnittflächen $K(\gamma_i) \cap K(\gamma_j)$.

⇒ a) G ist planar

b) jede Kante werden genau zwei Schnittpkte aus $E(\Gamma)$ zugeordnet.

⇒ $|E(\Gamma)| = 2 \cdot |E| \leq 2 \cdot (3n-6) = 6n-12$.

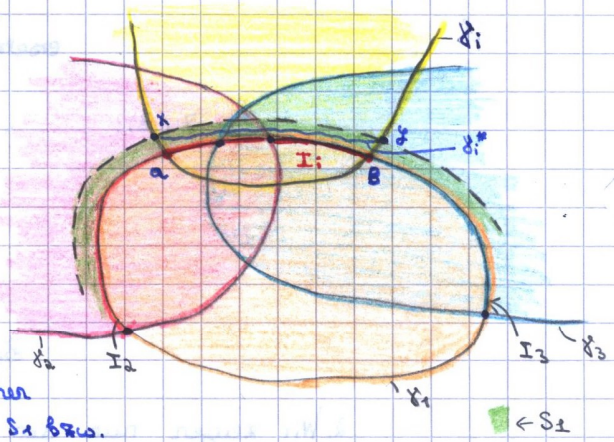


Ind.schritt: Sei Γ eine beliebige Menge von n Jordankurven, die sich paarw. nicht oder zweimal schneiden.
Sei $R(\Gamma) := (r_1, r_2, r_3)$.
Indann: Beh. gilt $\forall\, \Gamma'$ mit $R(\Gamma') \leq_L R(\Gamma)$

Fall 1: $r_1 > 0$
$\Rightarrow \exists\, \gamma_i \in \Gamma$ mit γ_i redundant (hier nur r_1 verkleinert $\Rightarrow$ "$<_L$")
Sei $\Gamma' := \Gamma \setminus \{\gamma_i\} \Rightarrow E(\Gamma') = E(\Gamma) \wedge R(\Gamma') <_L R(\Gamma)$, $|\Gamma'| = n-1$
$\Rightarrow E(\Gamma) = E(\Gamma') \overset{\text{Indann}}{\leq} 6(n-1)-12 = 6n-6-12 \leq 6n-12$.

Fall 2: $r_1 = 0 \wedge r_3 > 0$ (Bem: Fall $r_1 = r_3 = 0$ war Ind.anf.)
Wir können oBdA annehmen (durch geeignete Nummerierung der γ_i):
a) $K(\gamma_1) \cap K(\gamma_2) \cap K(\gamma_3) \neq \emptyset$ weil $r_3 > 0 \Rightarrow$ möglich
b) $\exists\, k \geq 0$ mit $\gamma_1 \cap K(\gamma_i) = I_i \neq \emptyset \;\; \forall\, i \leq k \;\wedge\; \gamma_1 \cap K(\gamma_j) = \emptyset \;\forall\, j > k$.
$\Rightarrow \exists$ einen (genügend schmalen) Streifen S_1 um die Kurve γ_1 mit $S_1 \cap \gamma_j \doteq \emptyset \;\forall\, j > k$ und S_1 enthält keine Schnittpkte. zw. γ_i's.

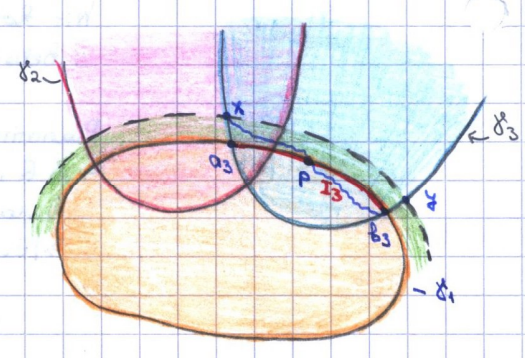
Fall 2.1: $\exists\, i : 2 \leq i \leq k$ mit $I_i \subseteq \bigcup^{*}_{\substack{j=2 \\ j \neq i}} I_j$



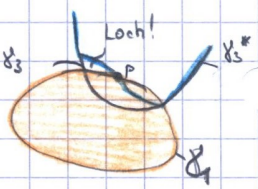
Bsp:

Seien x und y die Schnittpkte von γ_i mit Rand von S_1.
Dann ersetze γ_i durch eine Kurve γ_i^*.
Ersetze das Stück von γ_i zwischen x und y, das innerhalb von S_1 bzw.
$K(\gamma_1)$ verläuft, durch eine Kurve, die x und y im Inneren von S_1 verbindet.
Dann gilt für $\Gamma' := (\Gamma \setminus \{\gamma_i\}) \cup \{\gamma_i^*\}$:
a) $r_1(\Gamma') = r_1(\Gamma)$ (γ_i redundante $\Leftrightarrow$ γ_i^* redundante).
b) $r_2(\Gamma') < r_2(\Gamma)$ da $\gamma_i \cap \gamma_1 \neq \emptyset$ aber $\gamma_i^* \cap \gamma_1 = \emptyset$ und alle anderen Schnittpkte bleiben erhalten.
$\Rightarrow E(\Gamma) = E(\Gamma')$ da die Schnittpkte a, b von $\gamma_i \cap \gamma_1$ verdeckte waren
$\Rightarrow E(\Gamma) = E(\Gamma') \overset{\text{Indann.}}{\leq} 6n-12$

Fall 2.2: $\forall\, 2 \leq i \leq k$: $I \not\subseteq \bigcup^{*}_{\substack{j=2 \\ j \neq i}} I_j$ ←



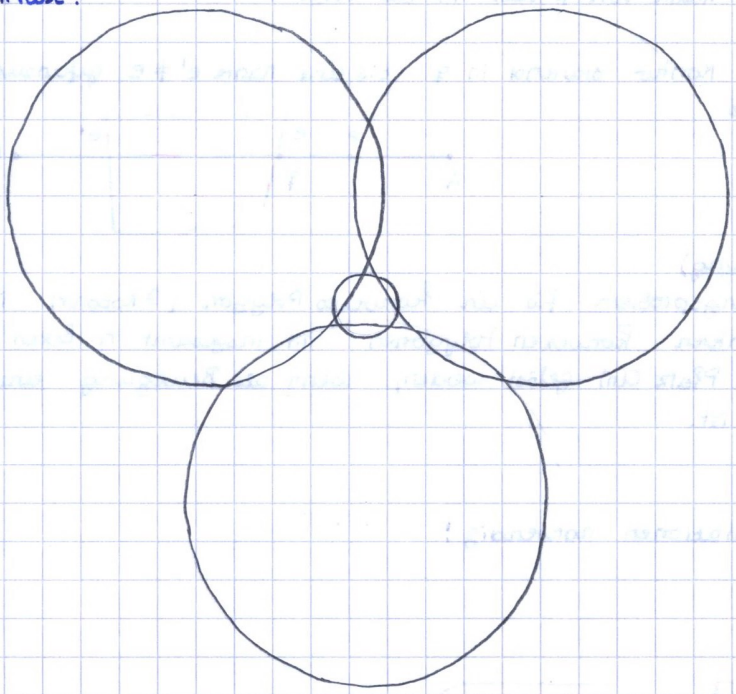
bei entspr. Nummerierung ↓
$\Rightarrow \exists\, p \in I_3$ mit $p \notin I_j$ für $j \neq 3$ wg.
Sei $I_3 = (a_3, b_3)$ und x, y Schnittpkte von γ_3 mit Rand von S_1
Verfeinere γ_3 zu γ_3^* wie folgt:
ersetze das Stück zwischen x und b_3 durch zwei Kurvenstücke:
a) von x nach p innerhalb S_1
b) von p nach b_3 innerhalb $K(\gamma_1)$
Dann gilt für die (entsprechende) entstandene Menge Γ':
a) $r_1(\Gamma') = r_1(\Gamma)$ weil γ_3 redundant $\Leftrightarrow$ γ_3^* redundant
b) $r_2(\Gamma') = r_2(\Gamma)$ $(a_3, b_3) \rightsquigarrow (p, b_3)$, sonst nix verändert $\Rightarrow$ # Schnpkte gleich.
"=" ? c) $r_3(\Gamma') < r_3(\Gamma)$
d) $E(\Gamma) < E(\Gamma')$ Weil: p ist ein neuer extremer Schnittpkt! (Wg. Loch ist p Randpkt!)



$\Rightarrow E(\Gamma) < E(\Gamma') \overset{\text{Indann.}}{\leq} 6n-12$.

⟶ 4.3.11.4 Bem:
$E(n) \leq 6n-12$ ist eine obere Schranke für Komplexität der Konvexen.
Es ex. Bsple, bei denen tatsächlich $6n-12$ äußere Schnittpkte vorkommen.

⟶ 4.3.11.5 Bsp:
n Kreise:



Die ersten drei Kreise haben 6 Schnittpkte. Jeder weitere Kreis erzeugt nochmals 6 Schnittpkte. → $6 + 6(n-3) = 6n-12$
(6: die ersten 3; $6(n-3)$: noch n-3 Kreis, pro Kreis 6 Schnittpkte $\Rightarrow 6 \cdot (n-3)$)

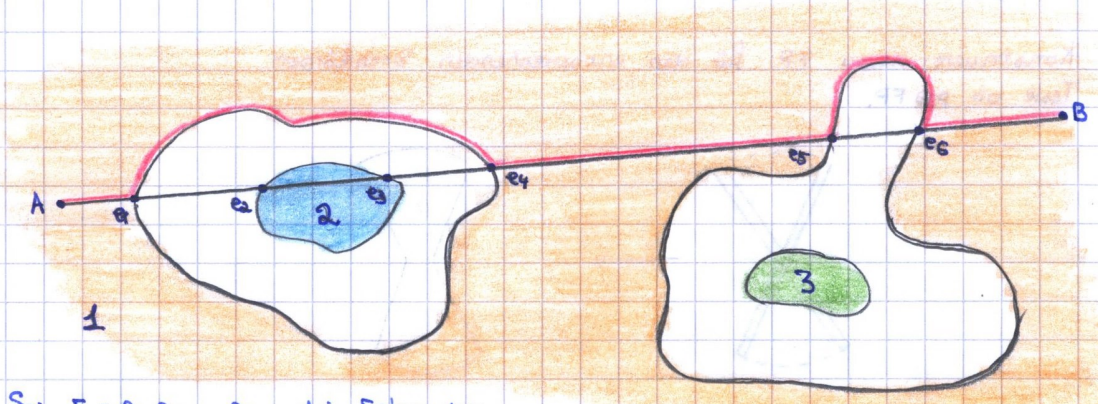
$\Rightarrow$ Schnittpkte $= O(n)$!

⟶ 4.3.11.6 Satz: Die Kontur von FP hat Größe $O(n)$ und kann in Zeit $O(n \log^2 n)$ berechnet werden.

4.3.12 Lösung des Bewegungsplanungsproblems:

1. Berechne die Zusammenhangskomponenten der Kontur in Zeit $O(n)$.
   Kontur: besteht aus geschlossenen Polygonzügen (evtl. ineinander verschachtelt), die sich nicht schneiden
   → planarer Graph.
2. Berechne die Schnittpkte der Strecke AB mit allen Konturkanten (nur echte Schnittpkte, keine Berührpkte).

$ZHK(e_1) = 1$ $ZHK(e_4) = 1$
$ZHK(e_2) = 2$ $ZHK(e_5) = 1$
$ZHK(e_3) = 2$ $ZHK(e_6) = 1$



1 und 2 kommen gerade oft vor!

Sei $F = e_1, e_2, \ldots e_\ell$ die Folge der geschnittenen Kanten entlang AB sortiert.
Zeit: $O(n \log n)$ $O(n)$ [für finden der Kanten] + $O(\ell \log \ell)$ [für sortieren der gefundenen Kanten] = $O(n \log n)$

3. Dann gilt: A und B liegen in derselben Komponente von FP
   $\Leftrightarrow$ In der Folge der ZHK's $ZHK(e_1) \ldots ZHK(e_\ell)$ kommt jeder Element gerade oft vor. ($\ell \leq n$)
   Suche e_i in 1, 2, 3 $\Rightarrow$ Zeit $O(n)$ $\Rightarrow$ Zeit $\forall i: O(\ell \cdot n) = O(n)$

4.3.13. Grober Algorithmus zur Lsg des Bewegungsproblems:

Eine mögliche Bewegung des Robotors kann nun wie folgt konstruiert werden:

$p \leftarrow A$

while $p \neq B$ do

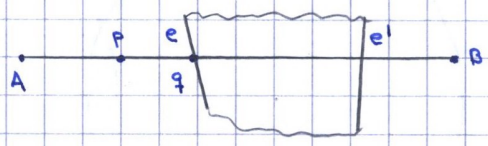
sei e nächste Kante von p aus in der Folge F

$q \leftarrow e \cap \overline{AB}$

durchlaufe die Kontur startend in q bis eine Kante $e' \neq e$ gefunden wird mit: $e' \cap \overline{AB} \neq \emptyset$

$p \leftarrow e' \cap \overline{AB}$

od



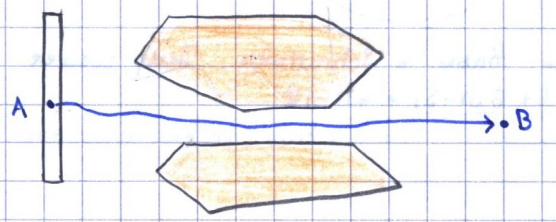
4.3.14. Satz: (Zusammenfassung)

Das Bewegungsplanungsproblem für ein konvexes Polygon (≙ Robotor) in einer Szene von paarweise disjunkten konvexen Polygonen mit insgesamt n Ecken kann in Zeit $O(n \log^2 n)$ und Platz $O(n)$ gelöst werden, wenn als Bewegung eine Folge von Translationen erlaubt ist.

Bew. s.o.

4.3.15 Bemerkung:

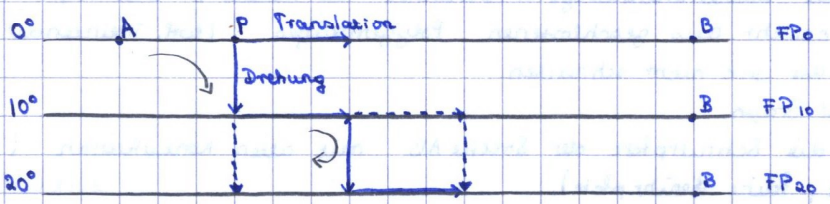
In der Praxis sind Rotationen notwendig!



Idee: Diskretisierung

d.h. erlaube nur eine Konstante Zahl von möglichen Winkeln (d.h. Orientierung des Robotors). z.B. 0°, 10°, 20°, ... 360°

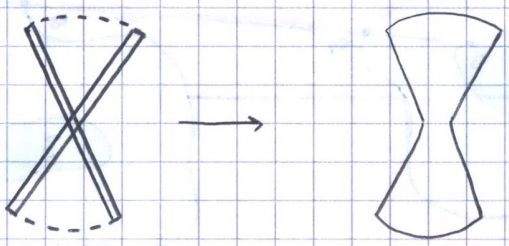
Nun konstruiere "Schichten" von FP-Konturen für jeden Winkel



Wichtig: Test, ob Drehung möglich ist!

Konstruiere dazu FP für den entsprechenden Drehkörper

Teste ob $p \in FP$.



4.3.10. Plane Sweep Alg. zum Mischen von zwei Konturen A und B.

→ 4.3.10.1. Problemstellung:

Eingabe: Zwei Konturen A, B in den Ecken durch zwei Mengen von Liniensegmenten.

Ausgabe: Menge von Segmenten, die die Kanten von $A \cup B$ definieren.

→ 4.3.10.2 Definition:

Ein Segment heißt **sichtbar**, wenn es zur Ausgabe gehört, d.h. auf Rand von $A \cup B$ liegt.

→ 4.3.10.3. Idee:

Wir modifizieren den Plane Sweep Alg. zum Linienschnitt.

Y-Struktur: Folge der von SL geschnittenen Segmenten (von unten nach oben sortiert). Wir speichern für jedes Segment in der Y-Struktur, ob es zur Zeit sichtbar oder unsichtbar ist.

Außerdem speichern wir für jedes Paar (s_1, s_2) von in Y benachbarten Segmenten, ob und von wem das Gebiet zwischen s_1 und s_2 zur Zeit überdeckt wird.

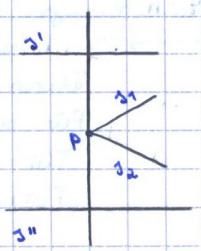
$$cover(s_1,s_2) = \begin{cases} \emptyset, & \text{nicht überdeckt} \\ \{A\}, & \text{von A} \\ \{B\}, & \text{von B} \\ \{A \cup B\}, & \text{von beiden.} \end{cases}$$

→ 4.3.10.4. Aktionen:

1. Linker Endpkt $p \in A$ (B analog) d.h. linke Ecke von zwei Segmenten $s_1, s_2 \in A$.

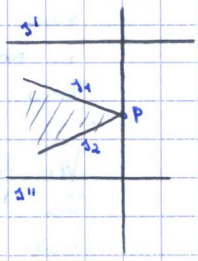
$s' := Y.succ(s_1)$, $s'' = Y.pred(s_2)$

Falls $cover(s', s'') = \emptyset$ oder $cover(s', s'') = A$ dann
  markiere s_1, s_2 als sichtbar
sonst unsichtbar
Falls $A \in cover(s', s'')$ (d.h. bei p Beginnt ein Loch) dann:
  $cover(s_1, s_2) \leftarrow cover(s', s'') \setminus \{A\}$
sonst $cover(s_1, s_2) \leftarrow cover(s', s'') \cup \{A\}$.
$cover(s', s_1) \leftarrow cover(s', s'')$
$cover(s_2, s'') \leftarrow cover(s', s'')$
$Y \leftarrow Y \cup \{s_1, s_2\}$.



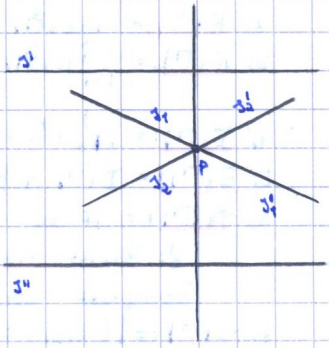
2. Rechter Endpkt $p \in A$ (B analog)

Falls s_1 bzw. s_2 sichtbar, dann Ausgabe der betreffenden Segmente.
$Y \leftarrow Y \setminus \{s_1, s_2\}$
$cover(s', s'') \leftarrow cover(s', s_1)$ $(= cover(s_2, s''))$.



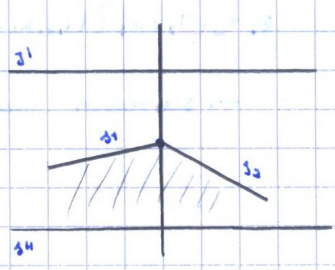
3. Schnittpkt $p = s_1 \cap s_2$

o.Bd.A. $s_1 \in A$, $s_2 \in B$
Falls s_1 (s_2) sichtbar $\Rightarrow$ Ausgabe bis p (splitten!)
$Y \leftarrow Y \setminus \{s_1, s_2\}$, $Y \leftarrow Y \cup \{s_1', s_2'\}$ in umgekehrter Reihenfolge mit negativer Sichtbarkeitsinformation
$cover(s', s_2') \leftarrow cover(s', s_1)$
$cover(s_1', s'') \leftarrow cover(s_2, s'')$
Falls $A \in cover(s_1, s_2)$ dann
  $cover(s_2', s_1') \leftarrow cover(s_1, s_2) \setminus \{A\}$
sonst $cover(s_2', s_1') \leftarrow cover(s_1, s_2) \cup \{A\}$
Falls $B \in cover(s_1, s_2)$ dann
  $cover(s_2', s_1') \leftarrow cover(s_1, s_2) \setminus \{B\}$
sonst $cover(s_2', s_1') \leftarrow cover(s_1, s_2) \cup \{B\}$.



4. Durchgang $p \in A$ (B analog)

s_1 endet und s_2 beginnt
$Y \leftarrow Y \setminus \{s_1\}$, $Y \leftarrow Y \cup \{s_2\}$
Falls s_1 sichtbar $\Rightarrow$ Ausgabe.
$Sichtbarkeit(s_2) \leftarrow Sichtbarkeit(s_1)$
$cover(s', s_2) \leftarrow cover(s', s_1)$; $cover(s_2, s'') \leftarrow cover(s_1, s'')$.



⟶ 4.3.10.5 Bem:

a). Es gibt nur Schnittsegmente zwischen Segmenten aus verschiedenen Kurven.

b). Segmente werden an Schnittpunkten gespalten (⇝ entweder ganz sichtbar oder ganz unsichtbar.)

c) Die Sichtbarkeitsinformation ändert sich nur bei Schnittpunkten (wechselt von sichtbar zu unsichtbar oder umgekehrt).

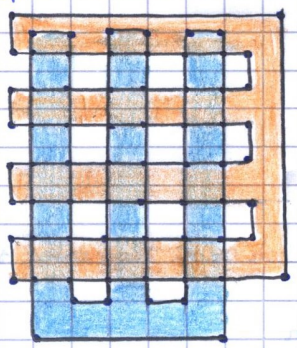
⟶ 4.3.10.6 Lemma:

Zwei Konturen A und B mit insgesamt n ebenen Kanten in Zeit $O((n+s)\log n)$ berechnet werden, wobei s = # Schnittpunkten zw. Segmenten in A und B.

Beweis: siehe Segmentschnitt.

⟶ 4.3.10.7 Bem: Im Allg. ist $s = O(n^2)$ ...

⟶ 4.3.10.8 Bsp:



z.Z: $T(n) = O(n\log^2 n)$.

Induktionsanfang: Beh. gelte für zB $\frac{n}{2}$

Induktionsschritt: dann gilt sie auch für n!

$$T(n) \leq c_1 n\log n + 2T\left(\tfrac{n}{2}\right) \leq c_1 n\log n + 2\cdot c_2 \tfrac{n}{2}\cdot \log^2\left(\tfrac{n}{2}\right) =$$
$$= c_1 n\log n + c_2 n(\log n - \log 2)^2 = c_1 n\log n + c_2 n(\log n - 1)^2 =$$
$$= c_1 n\log n + c_2 n\log^2 n - 2c_2 n\log n + c_2 n =$$
$$= n\log^2 n\left[c_1\cdot\frac{1}{\log n} + c_2 - \frac{2c_2}{\log n} + \frac{c_2}{\log^2 n}\right] =$$
$$= n\log^2 n\cdot\left[\underbrace{\frac{c_1 - 2c_2}{\log n}}_{n\to\infty \to 0} + c_2 + \underbrace{\frac{c_2}{\log^2 n}}_{n\to\infty \to 0}\right] \leq$$
$$\leq M\cdot n\log^2 n \quad \forall\, n \geq n_0$$

4.3.11. Analyse der Laufzeit:

⟶ 4.3.11.1. Idee: Wir zeigen nun, dass bei unserer Anwendung $s = O(n)$

Dann Kosten des Durchschnitt $O(n\log n)$

Für die Gesamtlaufzeit $T(n)$: Übung 8.

$T(1) = 1$

$T(n) = O(n\log n) + 2\cdot T\left(\frac{n}{2}\right) \quad \Rightarrow T(n) = O(n\log^2 n)$

Um die $O(n)$ Schranke für s zu zeigen, verwenden wir die Tatsache, dass sich die Ränder von jeweils zwei aufgeblähten Hindernissen Q_i und Q_j in höchstens zwei Punkten schneiden.

Beweis Übung.

Wir betrachten ein etwas allgemeineres Problem:

Sei $\Gamma = \gamma_1 .. \gamma_m$ eine Menge einfacher geschlossener Kurven (= Jordankurven) mit $|\gamma_i \cap \gamma_j| \leq 2 \;\forall\, i \neq j$. d.h. zwei Kurven schneiden sich höchstens zweimal.

⟶ 4.3.11.2 Definitionen:

1) $Inr(\gamma_i) :=$ Innere Regionen von γ_i

2) $K(\gamma_i) :=$ $Inr(\gamma_i) \cup \gamma_i$ (also Abschluß)

3) $K(\Gamma) := \bigcup_{i=1}^{m} K(\gamma_i)$

4) $I(\Gamma) := \bigcup_{i\neq j} (\gamma_i \cap \gamma_j)$ (alle Schnittpkte)

5) $E(\Gamma) := I(\Gamma) \cap Rand(K(\Gamma))$

6) $r_1(\Gamma) :=$ # der redundanten Kurven in Γ, wobei:

γ_i heißt redundant, falls $\gamma_i \subset \bigcup_{i\neq j} K(\gamma_i)$, d.h. γ_i wird komplett von anderen Kurvenflächen überdeckt. also $\Rightarrow E(\Gamma \setminus \{\gamma_i\}) = E(\Gamma)$

7) $r_2(\Gamma) := |\{(i,j) : i \neq j \wedge \gamma_i \cap \gamma_j \neq \emptyset\}|$

8) $r_3(\Gamma) := |\{(i,j,k) : i,j,k \text{ paarw. versch.}, K(\gamma_i) \cap K(\gamma_j) \cap K(\gamma_k) \neq \emptyset\}|$.

→ schreiben nun einfacher r_i statt $r_i(\Gamma)$ für $i = 1,2,3$.

Kapitel V: Geometrische Datenstrukturen

5.1. Segmentbaum

5.1.1. Definitionen + Bemerkungen

Sei $S = \{s_1 .. s_n\}$ Menge von n horizontalen Liniensegmenten in der Ebene.

Wir bezeichnen mit:

$x_1(s_i)$: die x-Koord. des linken Endpkts von s_i

$x_2(s_i)$: die x-Koord. des rechten Endpkts von s_i

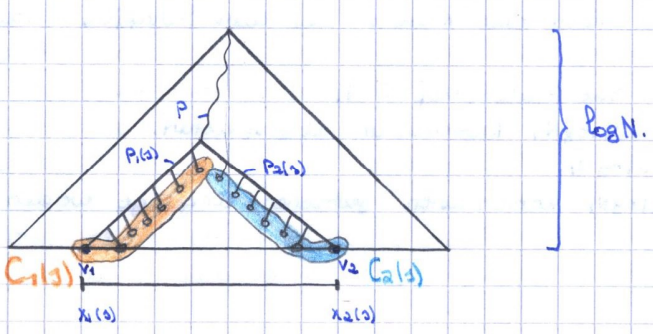
$y(s_i)$: die y-Koord von s_i.

Wir nehmen zunächst an, dass alle x-Koord. ganze Zahlen aus $\{1..N\}$ sind und die y-Koord. beliebige reelle Zahlen sind.
(→ sog. halbdynamischer Fall).

Ein Segmentbaum T zur Speicherung von S ist ein blattorientierter, binärer Suchbaum für die x-Koord. $\{1..N\}$ der Höhe $\log N$.
Jeder Knoten enthält zusätzlich eine Liste $NL(v)$ von Segmenten nach y-Koord. sortiert, die Knotenliste von v.
Bea: Beim Segmentbaum müssen die Blätter nicht verkettet sein.
Segmente von S werden wie folgt abgespeichert:
Sei $s \in S$.

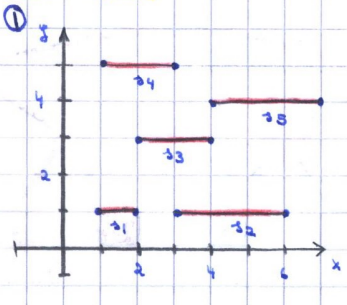
$P_1(s)$ = Suchpfad nach $x_1(s)$ \ Suchpfad nach $x_2(s)$
$P_2(s)$ = Suchpfad nach $x_2(s)$ \ Suchpfad nach $x_1(s)$
$C_1(s) = \{v : v$ rechtes Kind eines Knotens von $P_1(s)$ und $v \notin P_1(s)\} \cup \{v_1\}$
$C_2(s) = \{v : v$ linkes Kind eines Knotens von $P_2(s)$ und $v \notin P_2(s)\} \cup \{v_2\}$.
$C(s) = C_1(s) \cup C_2(s)$.

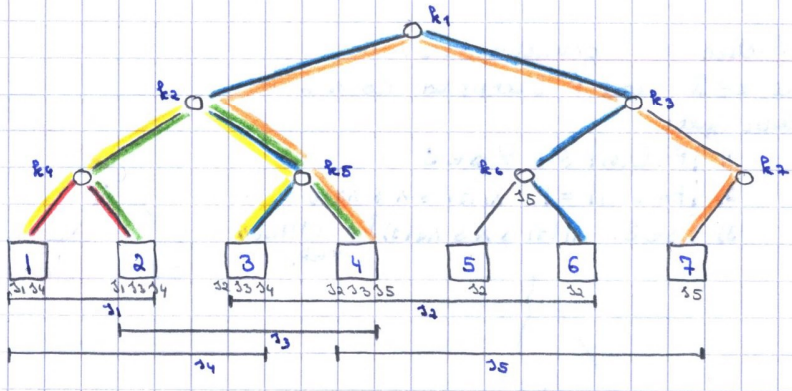
Dann wird s_i, $1 \le i \le n$ in alle Knotenlisten $NL(v)$ mit $v \in C(s)$ abgespeichert.
Skizze:



5.1.2. Beispiele:

①





$P_1(s_1)$ — $P_2(s_1)$ — $P_1(s_2)$ — $P_2(s_2)$ — $P_1(s_3)$ — $P_2(s_3)$ — $P_1(s_4)$ — $P_2(s_4)$ — $P_1(s_5)$ — $P_2(s_5)$

$C_1(s_1) = \{1\}$
$C_2(s_1) = \{2\}$
$C_1(s_2) = \{4,3\}$
$C_2(s_2) = \{5,6\}$
$C_1(s_3) = \{2\}$
$C_2(s_3) = \{3,4\}$
$C_1(s_4) = \{2,1\}$
$C_2(s_4) = \{3\}$
$C_1(s_5) = \{4\}$
$C_2(s_5) = \{k_6, 7\}$

$NL(1) = \{s_1, s_4\}$
$NL(2) = \{s_1, s_3, s_4\}$
$NL(3) = \{s_2, s_3, s_4\}$
$NL(4) = \{s_2, s_3, s_5\}$
$NL(5) = \{s_2\}$
$NL(6) = \{s_2\}$
$NL(7) = \{s_5\}$

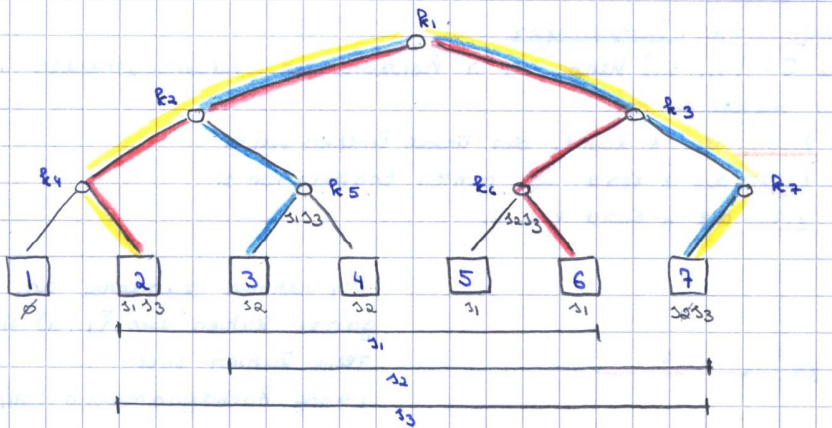
$NL(k_1 .. k_5) = \emptyset$
$NL(k_6) = \{s_5\}$
$NL(k_7) = \emptyset$

② $S = \{s_1, s_2, s_3\}$

$x_1(s_1) = 2,\ x_2(s_1) = 6$

$x_1(s_2) = 3,\ x_2(s_2) = 7$

$x_1(s_3) = 2,\ x_2(s_3) = 7.$



$P_1(s_1)$

$P_2(s_1)$ $C_1(s_1) = \{k_5, 2\}$

$P_1(s_2)$ $C_2(s_1) = \{5, 6\}$

$P_2(s_2)$ $C_1(s_2) = \{3, 4\}$

$P_1(s_3)$ $C_2(s_2) = \{k_6, 7\}$

$P_2(s_3)$ $C_1(s_3) = \{k_5, 2\}$

$C_2(s_3) = \{k_6, 7\}$

$NL(1) = \emptyset$

$NL(2) = \{s_1, s_3\}$

$NL(3) = \{s_2\}$

$NL(4) = \{s_2\}$

$NL(5) = \{s_1\}$

$NL(6) = \{s_1\}$

$NL(7) = \{s_2, s_3\}$

$NL(k_1) .. NL(k_4) = \emptyset$

$NL(k_5) = \{s_1, s_3\}$

$NL(k_6) = \{s_2, s_3\}$

$NL(k_7) = \emptyset$

5.1.3 Lemma:

✓ 1. $|C(s)| \leq 2 \cdot \log N \quad \forall s \in S.$

2) T hat Platzbedarf $O(N + n \cdot \log N)$

3) Einfügen eines Segmentes s kostet $O(\log N \cdot f(n))$, wobei $f(n)$ Kosten für Einfügen in Knotenliste der Länge n.

4) Streichen eines Segments s kostet $O(\log N \cdot g(n))$, wobei $g(n)$ Kosten für Streichen aus Knotenliste der Länge n.

Beweis:

1). Jeder Knoten $v \in C(s)$ ist Kind eines Knotens auf dem Suchpfad nach $x_1(s)$ bzw $x_2(s)$. im schlimmsten Fall: $|C_1(s)| = \log N = |C_2(s)| \Rightarrow |C(s)| \leq 2 \log N.$

2). Das Gerüst des Baums hat Platzbedarf $O(N)$ ← Großer Baum aus

1) ⇒ Jedes Segment ist in $O(\log N)$ Knotenlisten abgespeichert. denn s ist nur in den Knoten $v \in C(s)$ gespeichert $|C(s)| \leq 2 \log N \Rightarrow s$ ist in $O(\log N)$ Knoten gespeichert.

⇒ Platzbedarf: $O(N + n \cdot \log N)$. ← Ganzer Baum

3)/4). s muss aus / in $O(\log N)$ Knotenlisten gestrichen / eingefügt werden.

$f(n) \hat{=}$ Kosten für ein-einfügen im kleinen Baum } $2 \cdot \log N \cdot f(n)$ Kosten, um s in alle kleinen (erforderlichen (gesamt $\leq 2 \log N$)) Bäume einzufügen

Insgesamt in $\leq 2 \cdot \log N$ Bäume einfügen }

⇒ $O(\log N \cdot f(n))$

Für Streichen analog: $O(\log N \cdot g(n))$.

5.1.4 Suchen in Segmentbäumen:

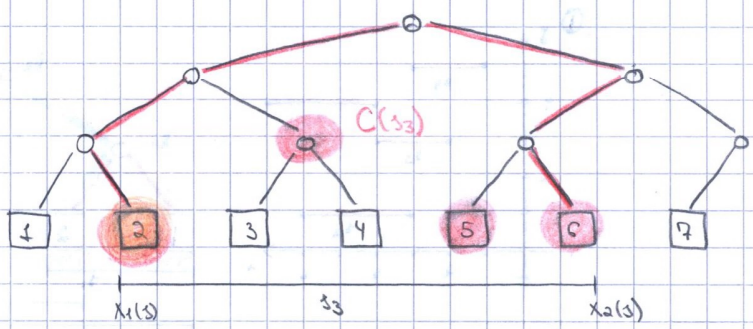
Sei $x \in \mathbb{R}$ und P Suchpfad nach x

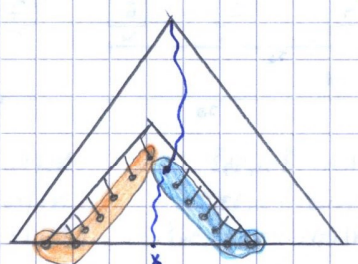
Dann gilt:

1) $|P \cap C(s)| \leq 1 \quad \forall s \in S$

2) $|P \cap C(s)| = 1 \quad x_1(s) \leq x \leq x_2(s)$

3) $\{s \in S : x_1(s) \leq x \leq x_2(s)\} = \bigcup_{v \in P} NL(v)$





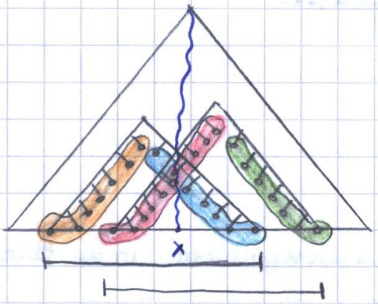
Bew: Ein Pfad ist eindeutig bestimmt

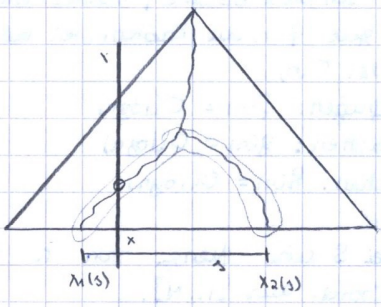
$\forall s \in S$ gilt: jedes x mit $x_1(s) \leq x \leq x_2(s)$ liegt unterhalb $C(s)$ und der Pfad von der Wurzel bis zu x geht durch genau einen Knoten von $C(s)$. ⇒ $|P \cap C(s)| = 1$

falls $x > x_2(s)$ oder $x < x_1(s)$ so schneidet der Pfad von der Wurzel bis zu x keinen Knoten aus $C(s)$

⇒ $|P \cap C(s)| = 0$

dh. die Knotenlisten auf dem Suchpfad nach x enthalten die Segmente, die von der vertikalen Geraden durch x geschnitten werden.





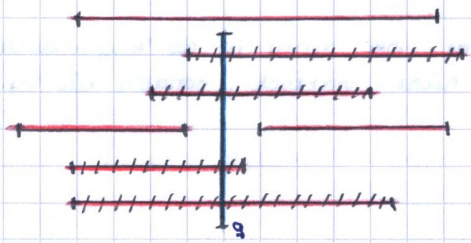
5.1.5. Laufzeit: $S(x) := \{s \in S : x_1(s) \le x \le x_2(s)\}$ kann in Zeit $O(\log N + m)$ berechnet werden, wobei $m := |S(x)|$, dh. die Größe der Ausgabe.

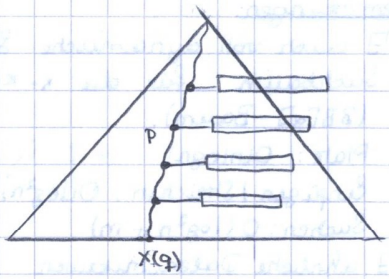
den Pfad durchlaufen ↑ in jedem Knoten in den kleineren Baum gehen und alle Knoten des kleineren Baumes (= die gesuchten Segmente) ausgeben. Für jede Ausgabe Const. Zeit Gesamt m Ausgaben ⇒ $O(\log N + m)$.

5.1.6. Problem:

Geg: das vertikale Segment q

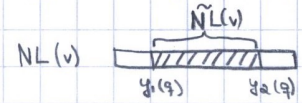
Ges: berechne $S(q) = \{s \in S : s \cap q \neq \emptyset\}$.





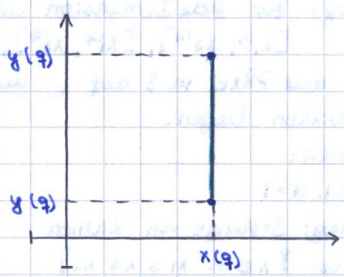
$\tilde{NL}(v) := \{s \in NL(v) : y_1(q) \le y(s) \le y_2(q)\}$

Dann ist $S(q) = \{s \in S : x_1(s) \le x(q) \le x_2(s) \text{ und } y_1(q) \le y(s) \le y_2(q)\} = \bigcup_{v \in P} \tilde{NL}(v)$



5.1.7 Algorithmus zur Berechnung des Problems:

1. P ← Suchpfad nach x(q)
2. Forall v ∈ P do
3. Lokalisiere $y_1(q)$ und $y_2(q)$ in NL(v)
4. gib alle Segmente davor aus
5. od.



5.1.8. Laufzeit:

Damit kostet die Berechnung von S(q): $O(\log N \cdot f(n) + m)$ Zeit, wobei $f(n)$ die Suchzeit in einer Knotenliste der Länge n ist.

Unterschied:

1). Man wandert den Pfad entlang und bei jedem Knoten gibt man seine Knotenliste NL(v) aus.
⇒ Kosten für Ausgabe im Knoten 1: $C \cdot |NL(v_1)|$
Kosten für Ausgabe im Knoten 2: $C \cdot |NL(v_2)|$
usw.
Gesamt m Ausgaben ⇒ $C \cdot |NL(v_1)| + C \cdot |NL(v_2)| + \ldots = C \cdot \underbrace{(|NL(v_1)| + |NL(v_2)| + \ldots)}_{= m} = C \cdot m$
weil alle ausgeg. werden!
⇒ $O(\log N + m)$
// Bem: nicht kennen!!!
// Anzahl der Elemente i.d Knotenlist i.a. verschieden
// ⇒ $\log N \cdot \underbrace{C_i \cdot |NL(v)|}_{=: f(n)}$ geht nicht!!! geht nur wenn
// Dann aber nicht mehr genau und nicht mehr output sensitiv und viel größere obere Schranke!!!

2). Man wandert den Pfad entlang und bei jedem Knoten sucht man nach entspr. Segmenten in Zeit $f(n)$ (= die größte Zeit, die es geben kann)
Höhe $\log N$, in jedem Knoten $f(n)$
⇒ $\log N \cdot f(n)$!

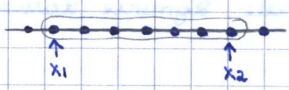
5.1.9. Realisierung der Knotenlisten.
→ durch Binäre balancierte Suchbäume. (Keine Gitterbäume!)
z.B. rot-schwarz Bäume, BB[α]-Bäume, AVL-Bäume, ...
Dann gilt für die Knotenlisten der Länge n:
- Platz: O(n)
- Einfügen: $f(n) = O(\log n)$
- Streichen: $g(n) = O(\log n)$
- Suchen: $R(n) = O(\log n)$.

5.1.10 Satz: Sei S eine Menge von n horizontalen Liniensegmenten in der Ebene mit x-Koord. aus $\{1..N\}$.
Dann gilt:
a). Ein Segmentbaum für S braucht Platz $O(N + n \log N)$.
b). Einfügen / Streichen eines Segments kostet: $O(\log n \log N)$
c) Suchen nach allen von einem vertikalen Suchsegment geschnittenen Segmenten
Kosten $O(\log n \cdot \log N + m)$
(Suchen in Knotenliste · Pfad · Ausgabe.)

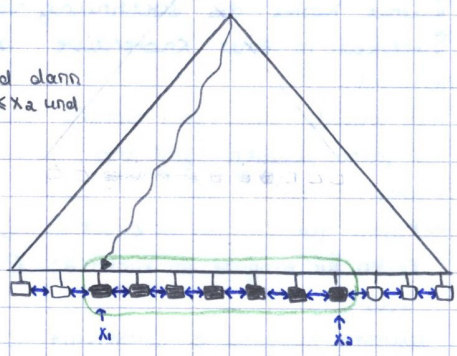
5.1.11. Bemerkungen:
1) ∃ auch voll dynamische Segmentbäume. Bei denen ist der zugrundeliegende Suchbaum für die x-Koordinaten nicht statisch, sondern ein bel. Baum. (BB[α]-Baum).
Platz: $O(n \log n)$ ← Da keine Gitterbäume mehr ⇒ Höhe: $\log n$
Einfügen / Streichen: $O(\log^2 n)$
Suchen: $O(\log^2 n + m)$
2) ∃ ähnliche Datenstrukturen für bel. (nicht notwendig horizontale) Segmente → Partition Tree.

5.2 Range-Tree (Bereichsabfragebaum).
5.2.1. Def: Range-Tree speichert Menge von Pkten im $\mathbb{R}^d$
Query: für jede Dimension ein Intervall.
$[x_1^{(1)}, x_2^{(1)}], [x_1^{(2)}, x_2^{(2)}], \ldots, [x_1^{(d)}, x_2^{(d)}]$.
Liste alle Pkte in S auf, dessen Koordinaten jeweils in den entsprechenden Intervallen liegen.

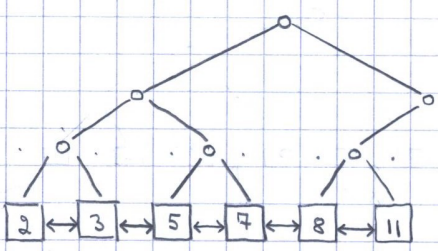
5.2.2 Beispiele:
→ 5.2.2.1. d=1
Geg: S=Menge von Zahlen
Ges: $\{x \in S : x_1 \le x \le x_2\}$



Datenstruktur: blattorientierter Suchbaum, wobei Blätter verkettet. (Hier kein Gitterbaum!)
(= 1-dim. Range-Tree).
Platz: O(n) weil bin. Baum.
Zeit Query: $O(\log n + k)$ wandere Pfad nach x_1 ($\log n$) und dann wandere über die verketteten Blätter solange $x \le x_2$ und gebe sie aus (k).
$n = |S|$
Insert / Delete: $O(\log n)$



Bsp: $S = \{2, 5, 3, 8, 11, 7\}$



5.2.2.2. d = 2.

Geg: $S \subset \mathbb{R}^2$ von n Pkten.

Ges: $\{q \in S : x_1 \le q_x \le x_2 \wedge y_1 \le q_y \le y_2\}$

Datenstruktur: blattorientierter Suchbaum, wobei Blätter verkettet.

Zunächst x-Koord. aus $\{1..N\}$. ⇒ Gitterbaum.

Ein 2 dim. Range-Tree besteht aus einem blattorientierten Suchbaum T für $\{1..N\}$.
Jeder Knoten v speichert eine nach y-Koord. sortierte Folge NL(v) (Knotenliste) von Pkten.
Die Pkte aus S werden wie folgt in einen anfangs leeren Baum T eingefügt:
Sei $p \in S$ mit $p = (p_x, p_y)$, dann wird p in jede Knotenliste NL(v) auf dem Suchpfad nach p_x gespeichert.

↑ Hier auch: blattorientierte Suchbäume, aber keine Gitterbäume!

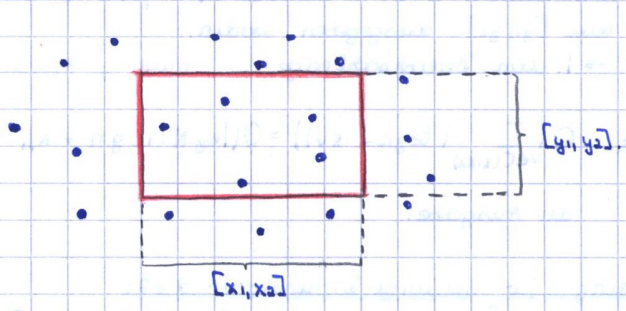
Jeder Pkt wird in log N Knoten gespeichert. $|S| = n \Rightarrow n \cdot \log N$.

Platz: $O(N + n \log N)$

Insert/Delete: $O(\log N \cdot \log n)$

↑ jeder NL(v) ist balancierter Baum (1. dim. Range Tree).

2-dim. Range Query:



gib alle Pkte $p \in S$ aus mit:
$x_1 \le p_x \le x_2 \wedge y_1 \le p_y \le y_2$

Vorgehen:

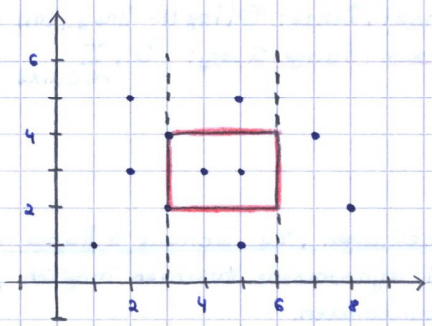
1. Schritt: Betrachte Pkte im unendlichen Streifen zw. x_1, x_2.

Beh: Die Knotenlisten NL(v) mit $v \in C(x_1, x_2)$ enthalten genau die gesuchten Pkte.

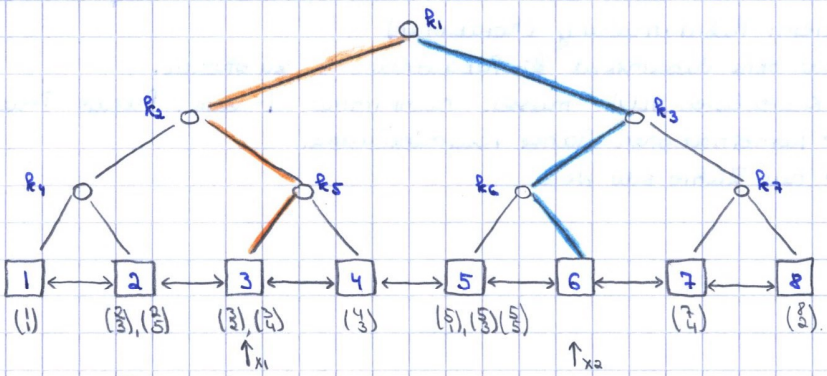
Bsp. zu Veranschaulichung:

$S = \{(1,1), (3,3), (3,5), (5,3), (5,5), (7,4), (4,3), (8,2)\}$.

$x_1 = 3$, $x_2 = 6$, $y_1 = 2$, $y_2 = 4$.



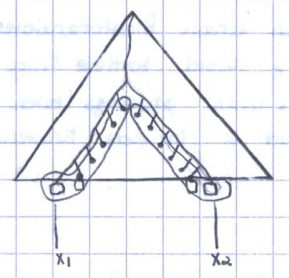
$p_1 = S$
$p_2 = \{(1,1), (3,3), (3,5), (5,3), (5,5), (7,4), (4,3)\}$
$p_3 = \{(5,1), (5,3), (5,5), (7,4), (8,2)\}$
$p_4 = \{(1,1), (3,3), (3,5)\}$
$p_5 = \{(3,3), (5,3), (4,3)\}$
$p_6 = \{(5,1), (5,3), (5,5),$
$p_7 = \{(7,4), (8,2)\}$



$C(x_1, x_2) = \{3, 4, 5, 6\}$
⇒ NL(3), NL(4), NL(5), NL(6) enthalten die gesuchten Pkte
⇒ $\{(3,3), (3,4), (4,3), (5,1), (5,3), (5,5)\}$

Begründung für die Behauptung:

Für beliebigen Knoten v enthält $NL(v)$ genau die Pkte, deren Suchpfad (nach x-Koord.) durch v geht, dh. alle Pkte im Bereich der Blätter des Unterbaumes von v.



Die Bereiche der Knoten $v \in C(x_1, x_2)$ bilden eine Partition des Intervalls $[x_1, x_2]$.

2. Schritt: Zur Beantwortung der eigentlichen Bereichsabfrage müssen wir nun die Knotenlisten $NL(v)$ $\forall v \in C(x_1, x_2)$ filtern, so dass nur Pkte mit y-Koord. aus $[y_1, y_2]$ ausgegeben werden.

→ 1. dim. Bereichsabfrage. nach y-Koord.!

⇒ Laufzeit: $O\left(\sum_{v \in C(x_1,x_2)} (\log n + K_v)\right) = O(\log N \cdot \log n + K)$, wobei $K := \sum_{v \in C(x_1,x_2)} K_v$ = Gesamtgröße der Ausgabe.

5.2.3 Verallgemeinerung für beliebige Dimensionen $d \geq 2$:

d-dim. Range-Tree besteht aus einem blattorientierten Suchbaum T für die erste Koordinate. Jeder Knoten v speichert $NL(v)$ als $(d-1)$-dim. Range-Tree. (Bzgl. Koord. 2..d).

Jeder Pkt $p \in S$ wird in allen $NL(v)$ für v auf dem Suchpfad nach 1. Koord. von p gespeichert.

Zuerst: Jede Dimension hat Koord. $\{1..N\}$ außer der letzten.

Platzbedarf: $O(\log N \cdot Platz_{d-1}(n) + N) = O(n \cdot \log^{d-1} N + N)$ ✓

Insert / Delete: $O(\log N \cdot Ins_{d-1}(n)) = O(\log^{d-1} N \cdot \log n)$

d-dim. Range-Query: $O\left(\sum_{v \in C(x_1,x_2)} (Query_{d-1}(n) + K_v)\right) = O(\log^{d-1} N \cdot \log n + K)$. ✓

5.2.4 Bemerkungen (zu Segment- & Range-Trees)

1). Voll dynamische Varianten möglich, dh. kein Gitter $\{1..N\}$ sondern beliebige, reelle Koordinaten.

Dazu: dynamischer Gerüstbaum T, dh. beim Einfügen und Löschen von Pkten (Segmenten) muss eventuell Blatt (mit entsprechender x-Koordinate) in T eingefügt oder entfernt werden. neben den Insert / Delete-Operationen auf Knotenlisten.

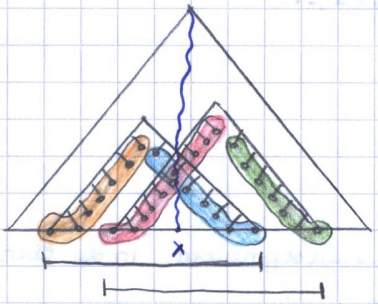
Problem: Rebalancierung (Rotationen)

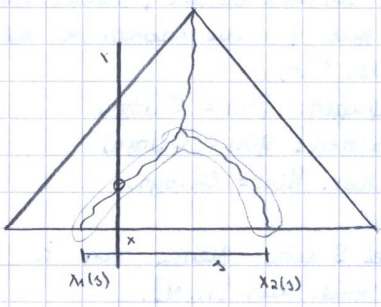
Lösbar ohne Gesamtzeit für Rebalancierung zu erhöhen.

2). Die Knotenlisten $NL(v)$ müssen nicht immer 1-dim. Range-Trees (Suchbäume) sein.

- Kombinationen Range / Segmentbäume.
- nur Zähler oder Werte

dh. die Knotenlisten auf dem Suchpfad nach x enthalten die Segmente, die von der vertikalen Geraden durch x geschnitten werden.





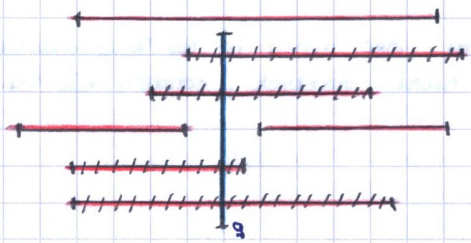
5.1.5. Laufzeit: $S(x) := \{s \in S : x_1(s) \le x \le x_2(s)\}$ kann in Zeit $O(\log N + m)$ berechnet werden, wobei $m := |S(x)|$, dh. die Größe der Ausgabe.

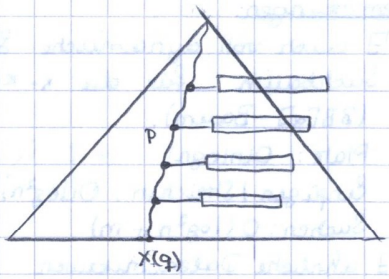
den Pfad durchlaufen ↑ in jedem Knoten in den kleineren Baum gehen und alle Knoten des kleineren Baumes (= die gesuchten Segmente) ausgeben. Für jede Ausgabe Const. Zeit Gesamt m Ausgaben ⇒ $O(\log N + m)$.

5.1.6. Problem:

Geg: das vertikale Segment q

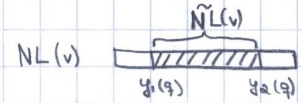
Ges: berechne $S(q) = \{s \in S : s \cap q \neq \emptyset\}$.





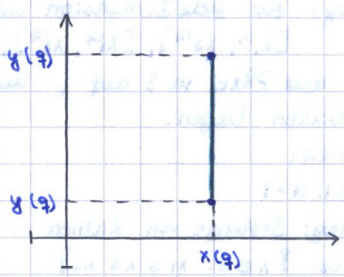
$\tilde{NL}(v) := \{s \in NL(v) : y_1(q) \le y(s) \le y_2(q)\}$

Dann ist $S(q) = \{s \in S : x_1(s) \le x(q) \le x_2(s) \text{ und } y_1(q) \le y(s) \le y_2(q)\} = \bigcup_{v \in P} \tilde{NL}(v)$



5.1.7 Algorithmus zur Berechnung des Problems:

1. P ← Suchpfad nach x(q)
2. Forall v ∈ P do
3. Lokalisiere $y_1(q)$ und $y_2(q)$ in NL(v)
4. gib alle Segmente davor aus
5. od.



5.1.8. Laufzeit:

Damit kostet die Berechnung von S(q): $O(\log N \cdot f(n) + m)$ Zeit, wobei $f(n)$ die Suchzeit in einer Knotenliste der Länge n ist.

Unterschied:

1). Man wandert den Pfad entlang und bei jedem Knoten gibt man seine Knotenliste NL(v) aus.
⇒ Kosten für Ausgabe im Knoten 1: $C \cdot |NL(v_1)|$
Kosten für Ausgabe im Knoten 2: $C \cdot |NL(v_2)|$
usw.
Gesamt m Ausgaben ⇒ $C \cdot |NL(v_1)| + C \cdot |NL(v_2)| + \ldots = C \cdot \underbrace{(|NL(v_1)| + |NL(v_2)| + \ldots)}_{= m} = C \cdot m$
weil alle ausgeg. werden!
⇒ $O(\log N + m)$
// Bem: nicht kennen!!!
// Anzahl der Elemente i.d Knotenlist i.a. verschieden
// ⇒ $\log N \cdot \underbrace{C_i \cdot |NL(v)|}_{=: f(n)}$ geht nicht!!! geht nur wenn
// Dann aber nicht mehr genau und nicht mehr output sensitiv und viel größere obere Schranke!!!

2). Man wandert den Pfad entlang und bei jedem Knoten sucht man nach entspr. Segmenten in Zeit $f(n)$ (= die größte Zeit, die es geben kann)
Höhe $\log N$, in jedem Knoten $f(n)$
⇒ $\log N \cdot f(n)$!

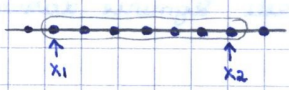
5.1.9. Realisierung der Knotenlisten.
→ durch Binäre balancierte Suchbäume. (Keine Gitterbäume!)
z.B. rot-schwarz Bäume, BB[α]-Bäume, AVL-Bäume, ...
Dann gilt für die Knotenlisten der Länge n:
- Platz: O(n)
- Einfügen: $f(n) = O(\log n)$
- Streichen: $g(n) = O(\log n)$
- Suchen: $R(n) = O(\log n)$.

5.1.10 Satz: Sei S eine Menge von n horizontalen Liniensegmenten in der Ebene mit x-Koord. aus $\{1..N\}$.
Dann gilt:
a). Ein Segmentbaum für S braucht Platz $O(N + n \log N)$.
b). Einfügen / Streichen eines Segments kostet: $O(\log n \log N)$
c) Suchen nach allen von einem vertikalen Suchsegment geschnittenen Segmenten
Kosten $O(\log n \cdot \log N + m)$
(Suchen in Knotenliste · Pfad · Ausgabe.)

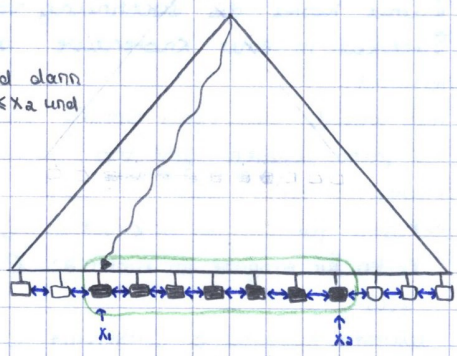
5.1.11. Bemerkungen:
1) ∃ auch voll dynamische Segmentbäume. Bei denen ist der zugrundeliegende Suchbaum für die x-Koordinaten nicht statisch, sondern ein bel. Baum. (BB[α]-Baum).
Platz: $O(n \log n)$ ← Da keine Gitterbäume mehr ⇒ Höhe: $\log n$
Einfügen / Streichen: $O(\log^2 n)$
Suchen: $O(\log^2 n + m)$
2) ∃ ähnliche Datenstrukturen für bel. (nicht notwendig horizontale) Segmente → Partition Tree.

5.2 Range-Tree (Bereichsabfragebaum).
5.2.1. Def: Range-Tree speichert Menge von Pkten im $\mathbb{R}^d$
Query: für jede Dimension ein Intervall.
$[x_1^{(1)}, x_2^{(1)}], [x_1^{(2)}, x_2^{(2)}], \ldots, [x_1^{(d)}, x_2^{(d)}]$.
Liste alle Pkte in S auf, dessen Koordinaten jeweils in den entsprechenden Intervallen liegen.

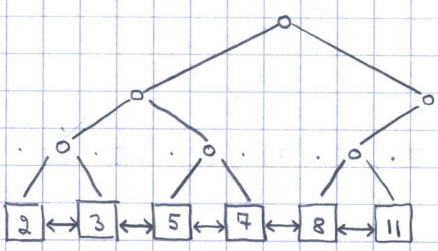
5.2.2 Beispiele:
→ 5.2.2.1. d=1
Geg: S=Menge von Zahlen
Ges: $\{x \in S : x_1 \le x \le x_2\}$



Datenstruktur: blattorientierter Suchbaum, wobei Blätter verkettet. (Hier kein Gitterbaum!)
(= 1-dim. Range-Tree).
Platz: O(n) weil bin. Baum.
Zeit Query: $O(\log n + k)$ wandere Pfad nach x_1 ($\log n$) und dann wandere über die verketteten Blätter solange $x \le x_2$ und gebe sie aus (k).
$n = |S|$
Insert / Delete: $O(\log n)$



Bsp: $S = \{2, 5, 3, 8, 11, 7\}$



5.2.2.2. d = 2.

Geg: $S \subset \mathbb{R}^2$ von n Pkten.

Ges: $\{q \in S : x_1 \le q_x \le x_2 \wedge y_1 \le q_y \le y_2\}$

Datenstruktur: blattorientierter Suchbaum, wobei Blätter verkettet.

Zunächst x-Koord. aus $\{1..N\}$. ⇒ Gitterbaum.

Ein 2 dim. Range-Tree besteht aus einem blattorientierten Suchbaum T für $\{1..N\}$.
Jeder Knoten v speichert eine nach y-Koord. sortierte Folge NL(v) (Knotenliste) von Pkten.
Die Pkte aus S werden wie folgt in einen anfangs leeren Baum T eingefügt:
Sei $p \in S$ mit $p = (p_x, p_y)$, dann wird p in jede Knotenliste NL(v) auf dem Suchpfad nach p_x gespeichert.

↑ Hier auch: blattorientierte Suchbäume, aber keine Gitterbäume!

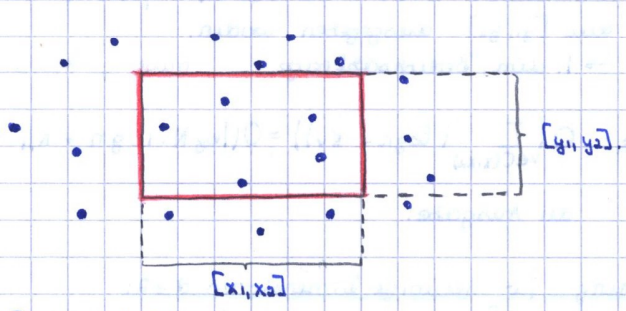
Jeder Pkt wird in log N Knoten gespeichert. $|S| = n \Rightarrow n \cdot \log N$.

Platz: $O(N + n \log N)$

Insert/Delete: $O(\log N \cdot \log n)$

↑ jeder NL(v) ist balancierter Baum (1. dim. Range Tree).

2-dim. Range Query:



gib alle Pkte $p \in S$ aus mit:
$x_1 \le p_x \le x_2 \wedge y_1 \le p_y \le y_2$

Vorgehen:

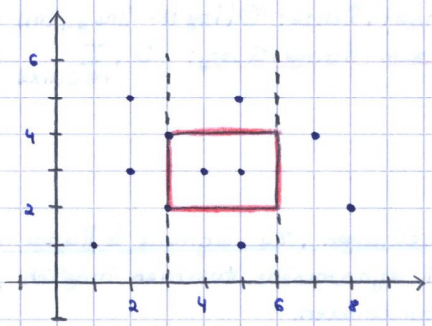
1. Schritt: Betrachte Pkte im unendlichen Streifen zw. x_1, x_2.

Beh: Die Knotenlisten NL(v) mit $v \in C(x_1, x_2)$ enthalten genau die gesuchten Pkte.

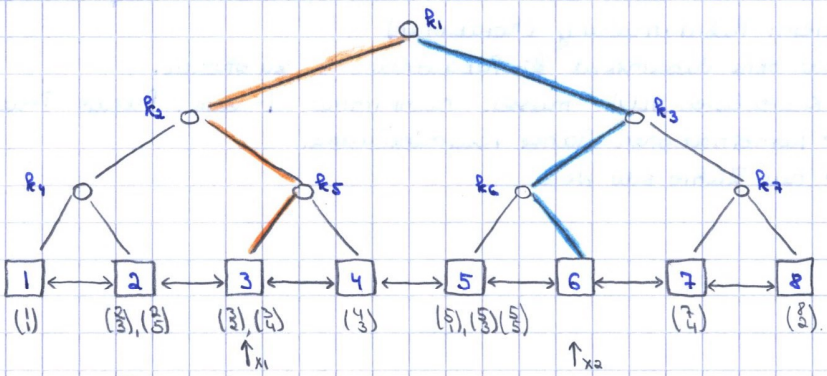
Bsp. zu Veranschaulichung:

$S = \{(1,1), (3,3), (3,5), (5,3), (5,5), (7,4), (4,3), (8,2)\}$.

$x_1 = 3$, $x_2 = 6$, $y_1 = 2$, $y_2 = 4$.



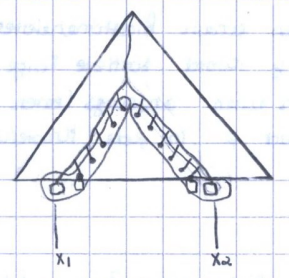
$p_1 = S$
$p_2 = \{(1,1), (3,3), (3,5), (5,3), (5,5), (7,4), (4,3)\}$
$p_3 = \{(5,1), (5,3), (5,5), (7,4), (8,2)\}$
$p_4 = \{(1,1), (3,3), (3,5)\}$
$p_5 = \{(3,3), (5,3), (4,3)\}$
$p_6 = \{(5,1), (5,3), (5,5),$
$p_7 = \{(7,4), (8,2)\}$



$C(x_1, x_2) = \{3, 4, 5, 6\}$
⇒ NL(3), NL(4), NL(5), NL(6) enthalten die gesuchten Pkte
⇒ $\{(3,3), (3,4), (4,3), (5,1), (5,3), (5,5)\}$

Begründung für die Behauptung:

Für beliebigen Knoten v enthält $NL(v)$ genau die Pkte, deren Suchpfad (nach x-Koord.) durch v geht, dh. alle Pkte im Bereich der Blätter des Unterbaumes von v.



Die Bereiche der Knoten $v \in C(x_1, x_2)$ bilden eine Partition des Intervalls $[x_1, x_2]$.

2. Schritt: Zur Beantwortung der eigentlichen Bereichsabfrage müssen wir nun die Knotenlisten $NL(v)$ $\forall v \in C(x_1, x_2)$ filtern, so dass nur Pkte mit y-Koord. aus $[y_1, y_2]$ ausgegeben werden.

→ 1. dim. Bereichsabfrage. nach y-Koord.!

⇒ Laufzeit: $O\left(\sum_{v \in C(x_1,x_2)} (\log n + K_v)\right) = O(\log N \cdot \log n + K)$, wobei $K := \sum_{v \in C(x_1,x_2)} K_v$ = Gesamtgröße der Ausgabe.

5.2.3 Verallgemeinerung für beliebige Dimensionen $d \geq 2$:

d-dim. Range-Tree besteht aus einem blattorientierten Suchbaum T für die erste Koordinate. Jeder Knoten v speichert $NL(v)$ als $(d-1)$-dim. Range-Tree. (Bzgl. Koord. 2..d).

Jeder Pkt $p \in S$ wird in allen $NL(v)$ für v auf dem Suchpfad nach 1. Koord. von p gespeichert.

Zuerst: Jede Dimension hat Koord. $\{1..N\}$ außer der letzten.

Platzbedarf: $O(\log N \cdot Platz_{d-1}(n) + N) = O(n \cdot \log^{d-1} N + N)$ ✓

Insert / Delete: $O(\log N \cdot Ins_{d-1}(n)) = O(\log^{d-1} N \cdot \log n)$

d-dim. Range-Query: $O\left(\sum_{v \in C(x_1,x_2)} (Query_{d-1}(n) + K_v)\right) = O(\log^{d-1} N \cdot \log n + K)$. ✓

5.2.4 Bemerkungen (zu Segment- & Range-Trees)

1). Voll dynamische Varianten möglich, dh. kein Gitter $\{1..N\}$ sondern beliebige, reelle Koordinaten.

Dazu: dynamischer Gerüstbaum T, dh. beim Einfügen und Löschen von Pkten (Segmenten) muss eventuell Blatt (mit entsprechender x-Koordinate) in T eingefügt oder entfernt werden. neben den Insert / Delete-Operationen auf Knotenlisten.

Problem: Rebalancierung (Rotationen)

Lösbar ohne Gesamtzeit für Rebalancierung zu erhöhen.

2). Die Knotenlisten $NL(v)$ müssen nicht immer 1-dim. Range-Trees (Suchbäume) sein.

- Kombinationen Range / Segmentbäume.
- nur Zähler oder Werte

5.3. Priority - Search - Tree:

5.3.1. Def: Sei $S \subset \mathbb{R}^2$

Ein Priority - Search - Tree (PST) T ist primärorientierter Suchbaum nach den x-Koord. der Pkte aus S.

5.3.2. Speichern der Pkte:

Ann: paarw. verschiedene x-Koordinaten.

$p \in S$ wird in einem Knoten auf Suchpfad nach p_x gespeichert und zwar so, dass T bzgl. der y-Koord. einen Maximumsheap bildet.

⇒ Auf jedem Pfad werden y-Koord. kleiner

⇒ Wurzel enthält Knoten mit max. y-Koord.

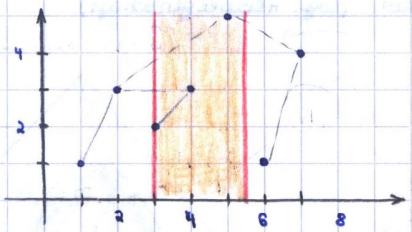
PST hat zwei Eigenschaften:
- Suchbaum nach x-Koord. der Pkte
- Heap nach y-Koord. der Pkte.

Aufbau (rekursiv):

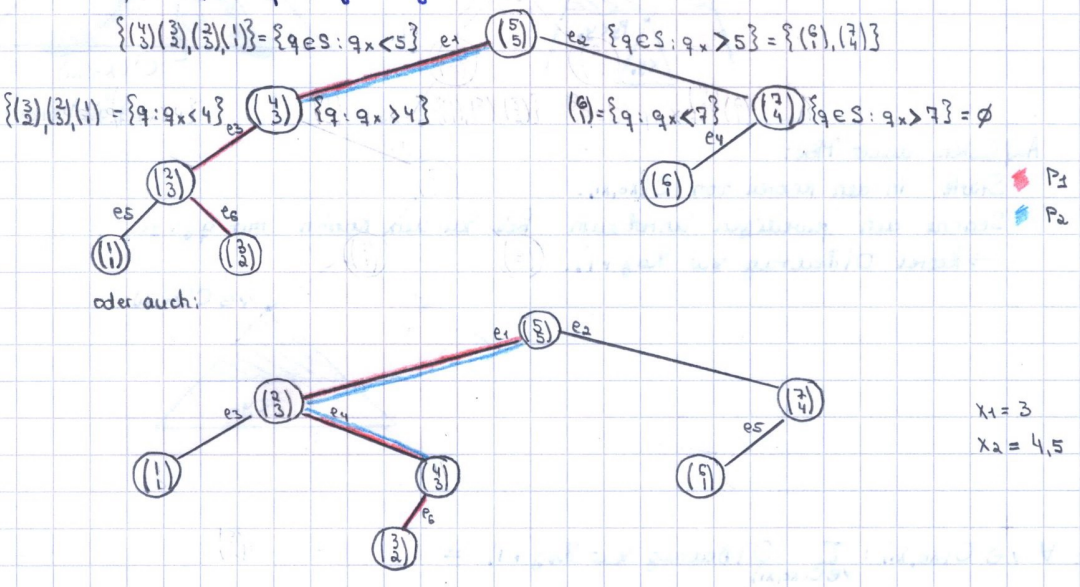
Wurzel w speichert $p \in S$ mit max. y-Koord.
- linker Unterbaum T_ℓ von w ist PST für $\{q \in S : q_x < p_x\}$
- rechter Unterbaum T_r von w ist PST für $\{q \in S : q_x > p_x\}$.

5.3.3. Beispiel:

$S = \{(1),(\frac{3}{2}),(\frac{2}{3}),(\frac{4}{3}),(\frac{6}{1}),(\frac{5}{5}),(\frac{7}{4})\}$.



Im Streifen: $(\frac{3}{2}),(\frac{4}{3}),(\frac{5}{5})$



oder auch:

$x_1 = 3$

$x_2 = 4{,}5$

5.3.4. Problem1 und Lösung:

Wo liegen alle Pkte aus einem vertikalen Streifen?

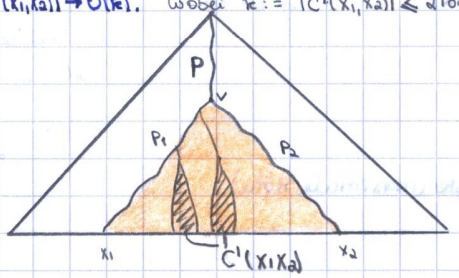
Antwort: auf P_1, P_2, P und allen Knoten zwischen P_1 und P_2!

Achtung: Auf den Pfaden P, P_1, P_2 können auch andere Pkte liegen.

Berechnung der Pkte:
- Filtere die Pkte auf Pfaden nach $[x_1, x_2]$ in Zeit $2 \cdot \log n + k$.
- Gib alle Pkte, die in Knoten zwischen P_1, P_2 gespeichert sind, aus $|C'(x_1,x_2)| \to O(\tilde{k})$. wobei $\tilde{k} := |C'(x_1,x_2)| \leq 2 \log n$

$2(\text{Höhe}(v) + 1 + O(k_1) + 1 + O(k_2) + \dots) =$
$= 2 \cdot (\text{Höhe}(v) + O(k_1) + O(k_2) + \sum 1) =$
$= 2 \cdot (\log n + \tilde{k})$

$\tilde{k}$:= # Knoten in orange.



Im Bsp: 1 Bild ⇒ $\{(\frac{5}{5}),(\frac{4}{3}),(\frac{2}{3}),(\frac{3}{2})\} \supset$ gesuchte Menge

2 Bild ⇒ $\{(\frac{5}{5}),(\frac{2}{3}),(\frac{4}{3}),(\frac{3}{2})\} \supset$ gesuchte Menge

Platz: $O(n)$!

Gesamtlaufzeit: $2\log n + k + O(\log n + \tilde{k}) =$
$= O(\log n + K)$

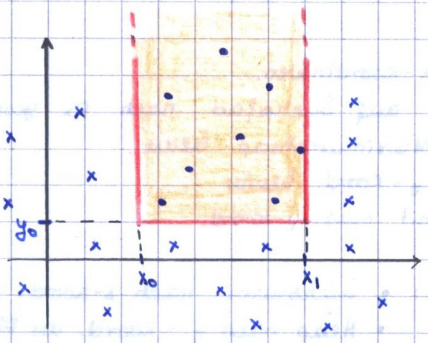
K = # Ausgaben. $(= k + \tilde{k})$

5.3.5 Problem 2 + Lösung:

Wozu Heap-Eigenschaft nach y-Koordinaten?
→ 1½-dimensionale Range Abfragen
(Halb offene oder 3 seitige).

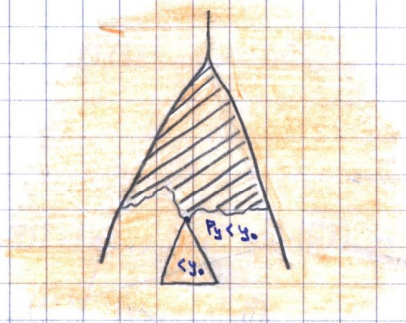
Geg: $x_0, x_1, y_0, \ldots, S \subset \mathbb{R}^2$

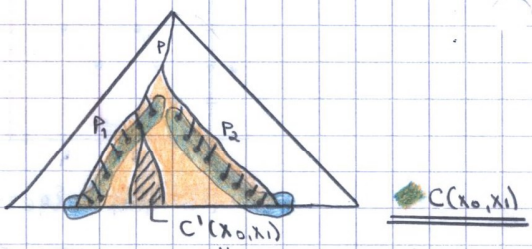
Ges: alle $p \in S$ mit
$x_0 \leq p_x \leq x_1$ und
$p_y \geq y_0$



Lösung:

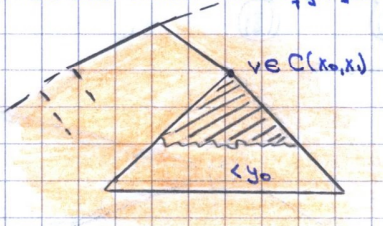
- Filtern der Pfade: $\forall$ Knoten $v \in P_1 \cup P_2 \cup P$ gib enthaltenen Pkt p aus, falls $x_0 \leq p_x \leq x_2 \wedge p_y \geq y_0$
  → Kostet Zeit $O(\log n)$. Mit Ausgabe kostet dies $O(\log n + k)$
- Rest der Ausgabe (normalerweise müssen alle $v \in P_1 \cup P_2 \cup P$ und zwischen P_1 und P_2 betrachtet werden. Oben nur $v \in P_1 \cup P_2 \cup P$, hier also die restlichen, dh. also: v zwischen P_1 und P_2) ist im oberen Bereich von $C'(x_0, x_1)$ gespeichert (wg. Heapeigenschaft)





{"alle kleinen Bäume"}

Auflisten dieser Pkte:
Starte in den Knoten von $C(x_0, x_1)$.
Scanne den jeweiligen Unterbaum bis zu den Knoten mit $p_y < y_0$
→ Kostet O(Beitrag zur Ausg + 1).



Laufzeit: $\forall v \in C(x_0, x_1)$: $\sum_{v \in C(x_0, x_1)} O$(Beitrag zur Ausg + 1) =

$= O(\log n + K')$, wobei K' = Beitrag zur Gesamtlösung von $C'(x_0 x_1)$

$\text{Höhe}(v) + 1 + O(\text{Beitr. zu Ausg}(v_1) + 1) + 1 + O(\ldots) + \ldots + \text{Höhe}(v) + 1 + O(\ldots) + \ldots = 2 \cdot \log n + \underbrace{\sum_{v \in C(x_0, x_1)} O(\text{Beitr. zu Ausg} + 1)}_{=: K'} = O(\log n + K')$

5.3.6. Satz (Zusammenfassung):

Der Priority-Search Tree verwaltet eine Menge von n Pkten im $\mathbb{R}^2$ unter folgenden Operationen und Laufzeiten:

- Insert / Delete: $O(\log n)$
- Drei-seitige Bereichsabfrage: $O(\log n + K)$, K = # Ausgaben.

und benötigt Speicherplatz: $O(n)$.

⇒ Gesamtlaufzeit: $a \log n + k + O(\log n + K') = \underline{O(\log n + \tilde{K})}$
$\tilde{K} = k + K'$, # Ausgaben.

5.3.7 Anwendung:

feste (maximale Höhe).

↑ Range-Abfragen mit PST (→ Übung).

5.3. Priority - Search - Tree:

5.3.1. Def: Sei $S \subset \mathbb{R}^2$

Ein Priority - Search - Tree (PST) T ist primärorientierter Suchbaum nach den x-Koord. der Pkte aus S.

5.3.2. Speichern der Pkte:

Ann: paarw. verschiedene x-Koordinaten.

$p \in S$ wird in einem Knoten auf Suchpfad nach p_x gespeichert und zwar so, dass T bzgl. der y-Koord. einen Maximumsheap bildet.

⇒ Auf jedem Pfad werden y-Koord. kleiner

⇒ Wurzel enthält Knoten mit max. y-Koord.

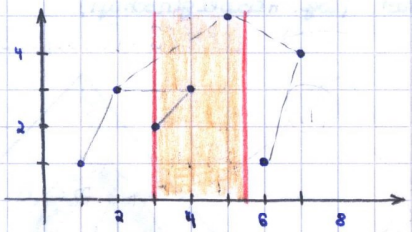
PST hat zwei Eigenschaften:
- Suchbaum nach x-Koord. der Pkte
- Heap nach y-Koord. der Pkte.

Aufbau (rekursiv):

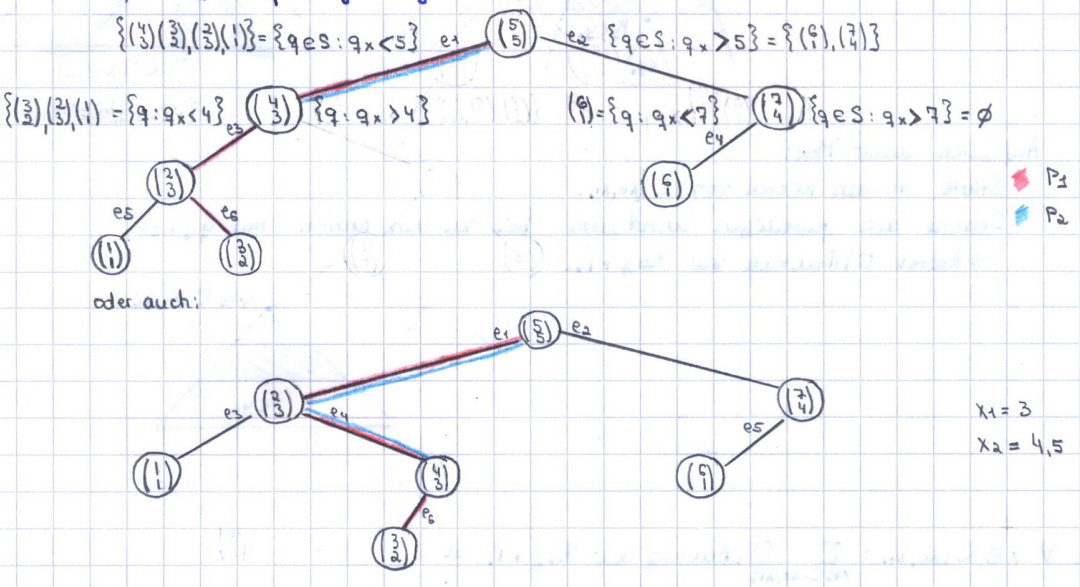
Wurzel w speichert $p \in S$ mit max. y-Koord.
- linker Unterbaum T_ℓ von w ist PST für $\{q \in S : q_x < p_x\}$
- rechter Unterbaum T_r von w ist PST für $\{q \in S : q_x > p_x\}$.

5.3.3. Beispiel:

$S = \{(1),(\frac{3}{2}),(\frac{2}{3}),(\frac{4}{3}),(\frac{6}{1}),(\frac{5}{5}),(\frac{7}{4})\}$.



Im Streifen: $(\frac{3}{2}),(\frac{4}{3}),(\frac{5}{5})$



oder auch:

$x_1 = 3$

$x_2 = 4{,}5$

5.3.4. Problem1 und Lösung:

Wo liegen alle Pkte aus einem vertikalen Streifen?

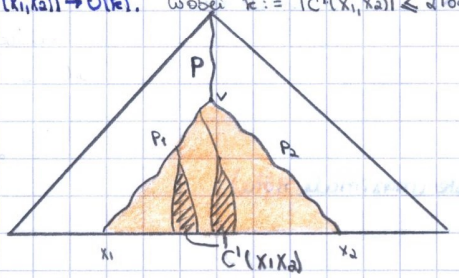
Antwort: auf P_1, P_2, P und allen Knoten zwischen P_1 und P_2!

Achtung: Auf den Pfaden P, P_1, P_2 können auch andere Pkte liegen.

Berechnung der Pkte:
- Filtere die Pkte auf Pfaden nach $[x_1, x_2]$ in Zeit $2 \cdot \log n + k$.
- Gib alle Pkte, die in Knoten zwischen P_1, P_2 gespeichert sind, aus $|C'(x_1,x_2)| \to O(\tilde{k})$. wobei $\tilde{k} := |C'(x_1,x_2)| \leq 2 \log n$

$2(\text{Höhe}(v) + 1 + O(k_1) + 1 + O(k_2) + \dots) =$
$= 2 \cdot (\text{Höhe}(v) + O(k_1) + O(k_2) + \sum 1) =$
$= 2 \cdot (\log n + \tilde{k})$

$\tilde{k}$:= # Knoten in orange.



Im Bsp: 1 Bild ⇒ $\{(\frac{5}{5}),(\frac{4}{3}),(\frac{2}{3}),(\frac{3}{2})\} \supset$ gesuchte Menge

2 Bild ⇒ $\{(\frac{5}{5}),(\frac{2}{3}),(\frac{4}{3}),(\frac{3}{2})\} \supset$ gesuchte Menge

Platz: $O(n)$!

Gesamtlaufzeit: $2\log n + k + O(\log n + \tilde{k}) =$
$= O(\log n + K)$

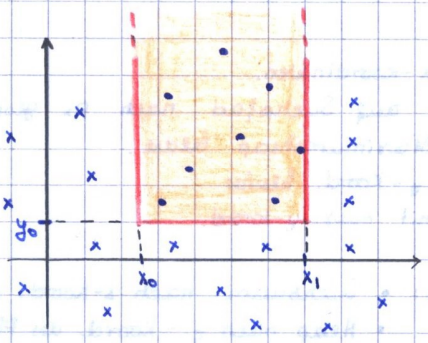
K = # Ausgaben. $(= k + \tilde{k})$

5.3.5 Problem 2 + Lösung:

Wozu Heap-Eigenschaft nach y-Koordinaten?
→ 1½-dimensionale Range Abfragen
(Halb offene oder 3 seitige).

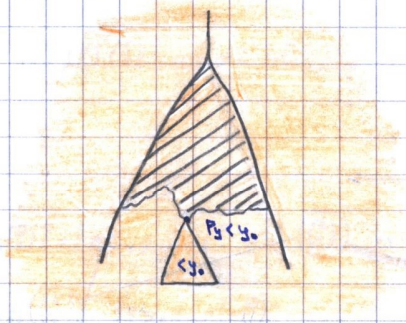
Geg: $x_0, x_1, y_0, \ldots, S \subset \mathbb{R}^2$

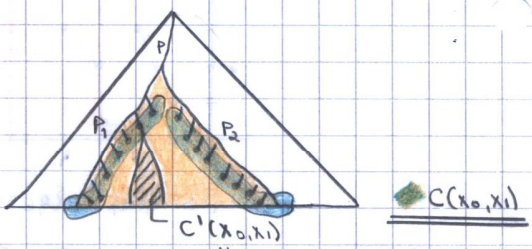
Ges: alle $p \in S$ mit
$x_0 \leq p_x \leq x_1$ und
$p_y \geq y_0$



Lösung:

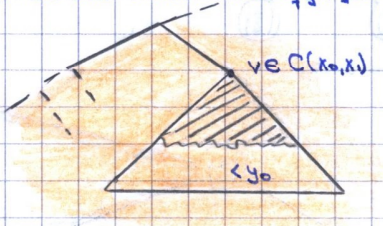
- Filtern der Pfade: $\forall$ Knoten $v \in P_1 \cup P_2 \cup P$ gib enthaltenen Pkt p aus, falls $x_0 \leq p_x \leq x_2 \wedge p_y \geq y_0$
  → Kostet Zeit $O(\log n)$. Mit Ausgabe kostet dies $O(\log n + k)$
- Rest der Ausgabe (normalerweise müssen alle $v \in P_1 \cup P_2 \cup P$ und zwischen P_1 und P_2 betrachtet werden. Oben nur $v \in P_1 \cup P_2 \cup P$, hier also die restlichen, dh. also: v zwischen P_1 und P_2) ist im oberen Bereich von $C'(x_0, x_1)$ gespeichert (wg. Heapeigenschaft)





{"alle kleinen Bäume"}

Auflisten dieser Pkte:
Starte in den Knoten von $C(x_0, x_1)$.
Scanne den jeweiligen Unterbaum bis zu den Knoten mit $p_y < y_0$
→ Kostet O(Beitrag zur Ausg + 1).



Laufzeit: $\forall v \in C(x_0, x_1)$: $\sum_{v \in C(x_0, x_1)} O$(Beitrag zur Ausg + 1) =

$= O(\log n + K')$, wobei K' = Beitrag zur Gesamtlösung von $C'(x_0 x_1)$

$\text{Höhe}(v) + 1 + O(\text{Beitr. zu Ausg}(v_1) + 1) + 1 + O(\ldots) + \ldots + \text{Höhe}(v) + 1 + O(\ldots) + \ldots = 2 \cdot \log n + \underbrace{\sum_{v \in C(x_0, x_1)} O(\text{Beitr. zu Ausg} + 1)}_{=: K'} = O(\log n + K')$

5.3.6. Satz (Zusammenfassung):

Der Priority-Search Tree verwaltet eine Menge von n Pkten im $\mathbb{R}^2$ unter folgenden Operationen und Laufzeiten:

- Insert / Delete: $O(\log n)$
- Drei-seitige Bereichsabfrage: $O(\log n + K)$, K = # Ausgaben.

und benötigt Speicherplatz: $O(n)$.

⇒ Gesamtlaufzeit: $a \log n + k + O(\log n + K') = \underline{O(\log n + \tilde{K})}$
$\tilde{K} = k + K'$, # Ausgaben.

5.3.7 Anwendung:

feste (maximale Höhe).

↑ Range-Abfragen mit PST (→ Übung).

5.4. Das Maßproblem für achsenparallele Rechtecke.

5.4.1. Problem:

Geg: n achsenparallele Rechtecke $R_1 .. R_n$.

Ges: Fläche von $\dot{\bigcup_{i=1}} R_i$

→ Anwendung von Plane Sweep und einer vereinfachten Form von Segmentbäumen.

5.4.2. Idee:

Schnitt von SL mit Vereinigung.

→ Menge von disjunkten vertikalen Segmenten.

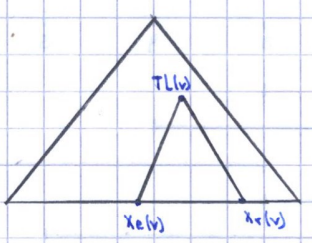
Sei V Menge dieser Segmente.

- an Event-Pkten (linker / rechter Rand von Rechtecken) kann sich V ändern.
- eigentlich brauchen wir nur L = Summe der Längen aller Segmente in V.

Dann können wir die gesuchte Fläche A durch Multiplizieren des aktuellen L-Wertes mit der zw. zwei Events zurückgelegten Distanz und Aufsummieren auf A berechnen.

Zur Verwaltung der Segmente und der Länge L verwenden wir einen Segmentbaum T. Segmente $s_i = R_i \cap SL$.

Sei $TL(v)$ die Länge der Vereinigung aller Segmente geschnitten mit $x_{range}(v)$, wobei $x_{range}(v) := [x_e(v), x_r(v)]$ mit $x_e(v)$ x-Koord. des linken Blattes im UB und $x_r(v)$ die x-Koord. des rechten Blattes im UB.



Dann gilt:

1. $L = TL(v)$
2. $$TL(v) = \begin{cases} x_r(v) - x_e(v), & NL(v) \neq \emptyset, \ v \text{ kein Blatt} \\ 0, & v \text{ Blatt} \\ TL(v_e) + TL(v_r), & NL(v) = \emptyset, \ v \text{ kein Blatt}, \ NL(u_r) \cap NL(u_e) = \emptyset \\ TL(v_e) + TL(v_r) + 1, & NL(v) = \emptyset, \ v \text{ kein Blatt}, \ NL(u_r) \cap NL(u_e) \neq \emptyset \end{cases}$$

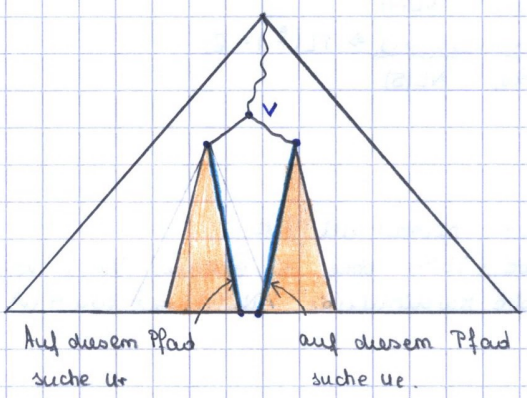
wobei $NL(u_r)$ wie folgt definiert ist:

Betrachte Pfad von v nach dem rechtesten Blatt von dem UB von dem linken Sohn von v. $\forall u \in$ diesem Pfad kontrolliere jeweils $NL(u)$ angefangen mit dem linken Sohn von v.

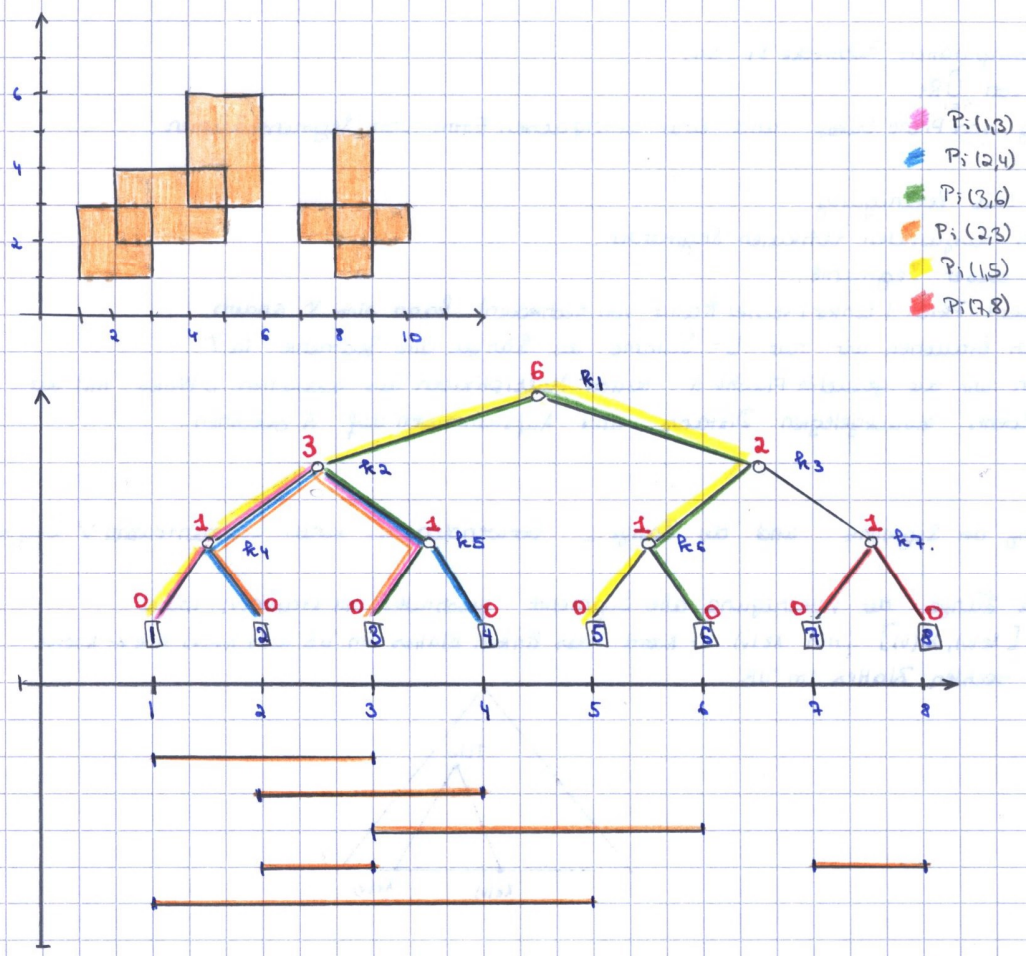
$NL(u_r) := NL(u)$, wobei u der erste Knoten auf dem Pfad, welches erfüllt: $NL(u) \neq \emptyset$

Falls $\forall u \in$ Pfad gilt: $NL(u) = \emptyset \Rightarrow NL(u_r) :=$ NL(rechtestes Blatt von dem linken UB von v) (darf dann auch leer sein).

$NL(u_e)$ analog



5.4.3. Beispiel:



$C(1,3) = \{1,2,3\}$
$C(2,4) = \{2,3,4\}$
$C(3,6) = \{3,4,5,6\}$
$C(2,3) = \{2,3\}$
$C(1,5) = \{1,2,k_5,5\}$
$C(7,8) = \{7,8\}$

$NL(1) = \{(1,3),(1,5)\}$
$NL(2) = \{(1,3),(2,4),(2,3),(1,5)\}$
$NL(3) = \{(1,3),(2,4),(3,6),(2,3)\}$
$NL(4) = \{(2,4),(3,6)\}$
$NL(5) = \{(1,5),(3,6)\}$
$NL(6) = \{(3,6)\}$

$NL(7) = \{(7,8)\}$
$NL(8) = \{(7,8)\}$
$NL(k_5) = \{(1,5)\}$
$NL(k_1 .. k_4) = \emptyset$

k_4: $(1,3) \in NL(1) \cap NL(2) \Rightarrow TL(k_4) = 0+0+1 = 1$
k_5: $(2,4) \in NL(3) \cap NL(4) \Rightarrow TL(k_5) = 1$
k_6: $(3,6) \in NL(5) \cap NL(6) \Rightarrow TL(k_6) = 1$
k_7: $(7,8) \in NL(7) \cap NL(8) \Rightarrow TL(k_7) = 1$
k_2: $(1,5) \in NL(k_5) \cap \underbrace{NL(u_1)}_{NL(2)} \Rightarrow TL(k_2) = 1+1+1 = 3$
k_3: $\emptyset = \underbrace{NL(u_1)}_{NL(6)} \cap \underbrace{NL(u_2)}_{NL(7)} \Rightarrow TL(k_3) = 2$
k_1: $(1,5) \in \underbrace{NL(u_1)}_{NL(k_6)} \cap \underbrace{NL(u_2)}_{NL(5)} \Rightarrow TL(k_1) = 6.$

5.4.4. Beobachtung:

$NL(v)$ zu speichern nimmt viel Platz
⇒ Idee: definiere $TL(v)$ ohne die genaue Menge von $NL(v)$ zu kennen und speichere nur die Kardinalität $|NL(v)| =: count(v)$.

5.4.5. Genauere Betrachtung der Aktionen:

a) linker Rand von R_i:

1) Füge $s_i = R_i$ in SL ein.
   statt einfügen von NL(v): count(v)++;
   $\forall$ Knoten $\in C(s_i)$, die schon vorher einen Zähler count(v) > 0 hatten, brauchen wir nichts zu tun.
2) Falls count(v) von 0 auf 1 erhöht wird, dann:
   - $TL(v) \leftarrow x_r(v) - x_e(v)$
   - Laufe von v nach oben und korrigiere die TL-Werte.

→ Laufzeit: $O(\log n)$, da wir Suchpfade von $C(s)$ zweimal durchlaufen.
(Laufzeit f. Insert)

b) rechter Rand von R_i:

1) entferne $s_i = R_i$ in SL aus Baum: count(v) − −; $\forall v \in C(s_i)$
2) Falls nun count(v) = 0, dann müssen TL-Werte von v und seinen Vorfahren geändert werden.

→ Laufzeit: $O(\log n)$ (s. Insert)

c) Bei jedem Event-Pkt x: $A \leftarrow A + TL(\text{wurzel}) \cdot (x - x_{old})$.

5.4.6. Satz: Das Maß-problem von n achsenparallelen Rechtecken kann in Zeit $O(n \log n)$ und Platz $O(n)$ gelöst werden.

Beweis: 1). Platzbedarf ist $O(n)$, da keine Knotenlisten gespeichert werden, sondern in jedem Knoten zwei Zahlen count(v) und TL(v).

2). Rechtecke sortieren (nach linkem / rechten Rand) nach x-Koord. also
→ x-Struktur: $O(n \log n)$
Pro Event Zeit $O(\log n)$ für Operationen im Segmentbaum.

In unserem Bsp:

Kapitel VI: Drei-dimensionale konvexe Hüllen.

6.1. Einführung:

6.1.1. Problem:

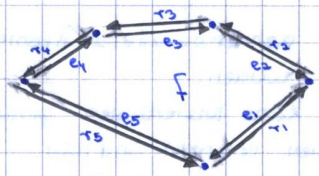
Geg: Menge S von n Pkten im $\mathbb{R}^3$, $p \in S$, $p = (x,y,z)$ (kart. Koord.)
Ges: CH(S) (= kleinste konvexe Menge, die S enthält).
Analog zum Gummiband-Modell: Gummi-Fläche
$\Rightarrow$ CH(S) ist ein konvexes Polyeder.
Ausgabe: Oberfläche des Polyeders.

- besteht aus Ecken, Kanten und Flächen (Euler-Formel)
- kann beschrieben werden durch einen planaren Graphen.

6.1.2. Darstellung des (planaren) Oberflächengraphen:

als zweifach gerichteten (bidirected) Graphen.

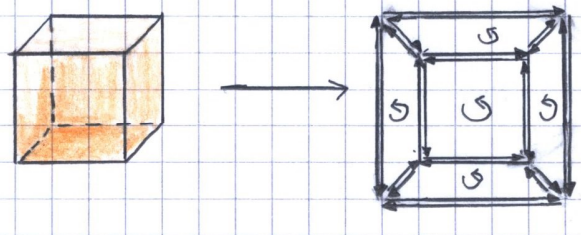
1. Für jeden Knoten (Ecke) v ordnen wir die ausgehenden Kanten gg. den Uhrzeigersinn (bei Ansicht von außen)
2. Jede Kante $e = (v,w)$ hat einen Verweis auf ihre Gegenkante $r = (w,v) =: rev(e)$
3. Die Flächen sind dann implizit definiert
   e_2 = Vorgänger (rev(e)) (in der Adjazenzliste)
   Das definiert einen Kantenzyklus für jede Fläche f.



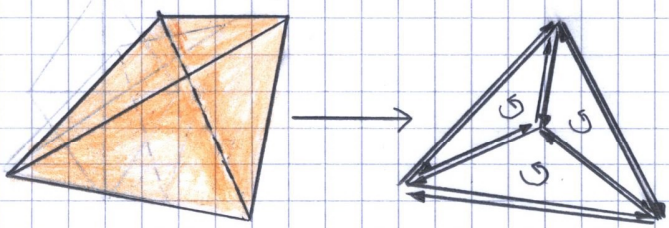
e_2 = G.face_cycle_source(ei)
G.cyclic_pred(G.reversal(ei))

4. In den CH-Algorithmen werden zunächst alle Flächen Dreiecke sein.
   Bei allg. eingebetteten planaren Graphen ist u.U. die äußere Fläche kein Dreieck.

6.1.3. Beispiel:



← nach außen gegen den Uhrzeigersinn bzw. !!



6.1.4. Geometrische Prädikate:

Für $a,b,c,d \in \mathbb{R}^3$:

$$\text{orientation}(a,b,c,d) := \text{sign det} \begin{pmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{pmatrix}$$

Diese Determinante heißt Spatprodukt.
= Vielfaches des Volumens (mit Vorzeichen) des Simplex (a,b,c,d)
d.h. orientation(a,b,c,d) sagt, auf welcher Seite der Ebene durch a,b,c (a,b,c nicht collinear) der Pkt d liegt.

→

Kapitel VI: Drei-dimensionale konvexe Hüllen.

6.1. Einführung:

6.1.1. Problem:

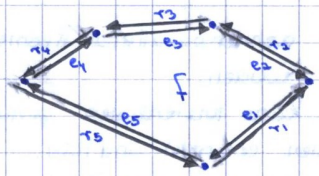
Geg: Menge S von n Pkten im $\mathbb{R}^3$, $p \in S$, $p = (x,y,z)$ (kart. Koord.)
Ges: CH(S) (= kleinste konvexe Menge, die S enthält).
Analog zum Gummiband-Modell: Gummi-Fläche
$\Rightarrow$ CH(S) ist ein konvexes Polyeder.
Ausgabe: Oberfläche des Polyeders.

- besteht aus Ecken, Kanten und Flächen (Euler-Formel)
- kann beschrieben werden durch einen planaren Graphen.

6.1.2. Darstellung des (planaren) Oberflächengraphen:

als zweifach gerichteten (bidirected) Graphen.

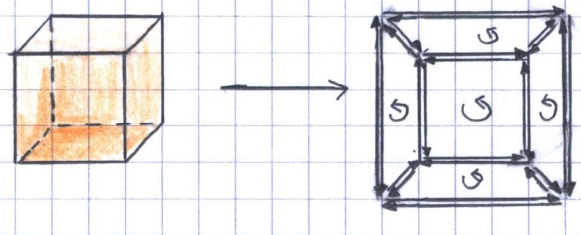
1. Für jeden Knoten (Ecke) v ordnen wir die ausgehenden Kanten gg. den Uhrzeigersinn (bei Ansicht von außen)
2. Jede Kante $e = (v,w)$ hat einen Verweis auf ihre Gegenkante $r = (w,v) =: rev(e)$
3. Die Flächen sind dann implizit definiert
   e_2 = Vorgänger (rev(e)) (in der Adjazenzliste)
   Das definiert einen Kantenzyklus für jede Fläche f.



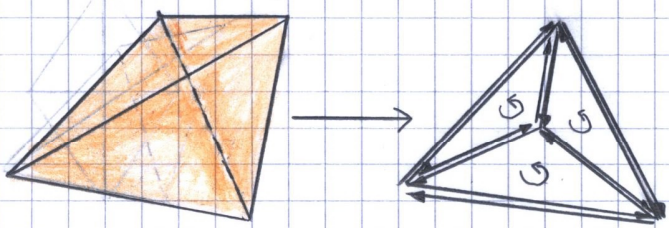
e_2 = G.face_cycle_source(ei)
G.cyclic_pred(G.reversal(ei))

4. In den CH-Algorithmen werden zunächst alle Flächen Dreiecke sein.
   Bei allg. eingebetteten planaren Graphen ist u.U. die äußere Fläche kein Dreieck.

6.1.3. Beispiel:



← nach außen gegen den Uhrzeigersinn bzw. !!



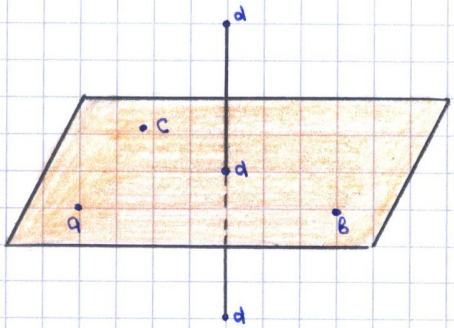
6.1.4. Geometrische Prädikate:

Für $a,b,c,d \in \mathbb{R}^3$:

$$\text{orientation}(a,b,c,d) := \text{sign det} \begin{pmatrix} a_x & a_y & a_z & 1 \\ b_x & b_y & b_z & 1 \\ c_x & c_y & c_z & 1 \\ d_x & d_y & d_z & 1 \end{pmatrix}$$

Diese Determinante heißt Spatprodukt.
= Vielfaches des Volumens (mit Vorzeichen) des Simplex (a,b,c,d)
d.h. orientation(a,b,c,d) sagt, auf welcher Seite der Ebene durch a,b,c (a,b,c nicht collinear) der Pkt d liegt.

→



> 0

$= 0$ (linabh., dh. in einer gemeinsamen Ebene).

< 0

6.2. Algorithmen:

6.2.1. Inkrementelle Alg. (Variante von Graham's Scan)

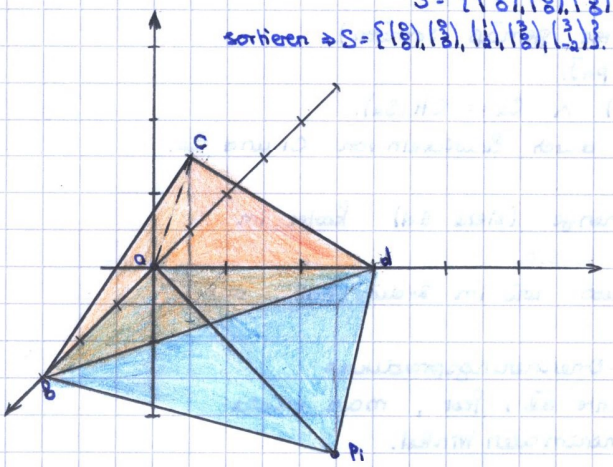
→ 6.2.1.1. Alg: (analog zu $\mathbb{R}^2$-Alg.)

1. Sortiere Pkte nach x-Koord. (bei Gleichheit nach y, z, dh. lexikografisch).
2. Baue Simplex aus ersten vier Pkten.
   (Sonderfall: coplanar) dann suche solange bis linanabh.
3. Allg. Situation:
   füge p_i hinzu, Hülle $CH(p_{i-1}, (\{p_1 .. p_{i-1}\})$ liegt als Oberflächengraph vor.
   Startend im Knoten p_{i-1} durchsuchen wir den Graphen und besuchen alle Dreiecke (Flächen) (a,b,c) mit orientation $(a,b,c,p_i) \geq 0$.
   (Alle von der Lichtquelle p_i beleuchteten Dreiecke oder alle von p_i sichtbaren Dreiecke).
   Dann entfernen wir alle Kanten von besuchten, sichtbaren (beleuchteten) Dreiecken und alle danach isolierten Knoten.
   → Am Rand entstehen Kanten ohne Gegenkanten.
   Für jede Kante (v,w) auf dem Rand füge ein neues Dreieck (v,w,p_i) hinzu.
   → Tangentialer Kegel mit Spitze p_i.

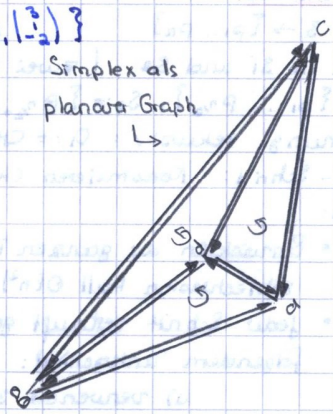
→ 6.2.1.2 Bsp:

$$S = \{\begin{pmatrix}0\\0\end{pmatrix}, \begin{pmatrix}3\\0\end{pmatrix}, \begin{pmatrix}0\\3\end{pmatrix}, \begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}3\\-2\end{pmatrix}\}$$

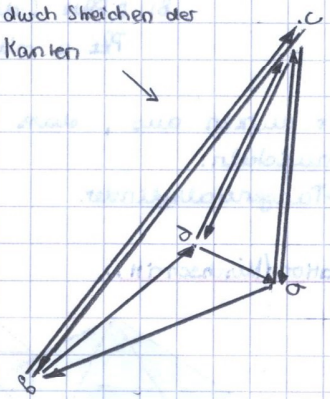
sortieren $\Rightarrow S = \{\begin{pmatrix}0\\0\end{pmatrix}, \begin{pmatrix}0\\3\end{pmatrix}, \begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}3\\0\end{pmatrix}, \begin{pmatrix}3\\-2\end{pmatrix}\}$.



Simplex als planarer Graph →



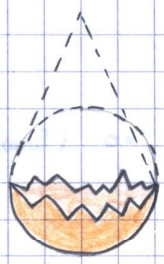
durch Streichen der Kanten ↘



$\text{orient}(d,c,b,p_i) \leq 0$
$\text{orient}(d,b,a,p_i) \geq 0$
$\text{orient}(d,c,a,p_i) \leq 0$
$\text{orient}(c,b,a,p_i) \leq 0$
⇒ entstehen neue Dreiecke:
$\Delta\, dBp_i$, $\Delta\, dap_i$, $\Delta\, abp_i$
$\Delta\, Bda$ verschwindet.

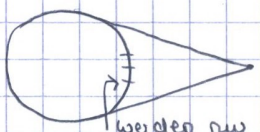
↳ 6.2.1.3 Bem:

Am Ende bleibt also:



↳ 6.2.1.4. Laufzeit.

Im $\mathbb{R}^2$:



werden nur einmal entfernt + Kosten für zwei Tangenten.

im $\mathbb{R}^3$:

Rand ist nicht konstant (⇒ mehr als zwei Tangenten)

$O(n)$ möglich!

Im schlechtesten Fall: Laufzeit: $\Theta(n^2)$.

In der Praxis wird der Alg. dennoch eingesetzt, da es für viele Eingaben doch relativ schnell ist.

↳ 6.2.1.5 Bem:

⊕ einfach, praktisch, effizient (für bes Eingaben)

⊖ im schlechtesten Fall: $O(n^2)$.

→ Randomisierung: Sweep in zufällige Richtung

⇔ Drehung der Punktmenge in zufälligen Winkel.

Resultat: Erwartete Laufzeit: $O(n \log n)$.

6.2.2. Divide & Conquer - Alg:

↳ 6.2.2.1. Algorithmus:

1. Sortiere $S \to \{p_1 .. p_n\}$
2. Zerlege in S_1 und S_2 (zwei Hälften gemäß sortiert)
   $S_1 := \{p_1 .. p_{n/2}\}$, $S_2 := \{p_{n/2+1} .. p_n\}$.
3. Berechnung rekursiv: $C_1 := CH(S_1) \wedge C_2 := CH(S_2)$.
4. Misch - Schritt: Konstruieren $CH(S)$ durch Einwickeln von C_1 und C_2.

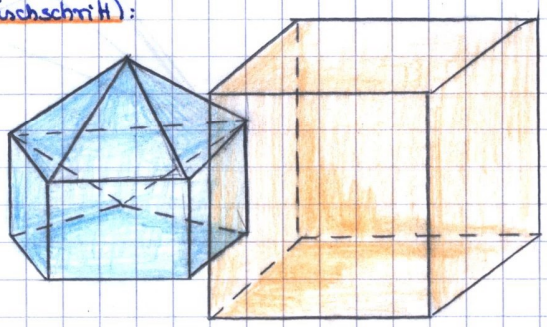
Bem:

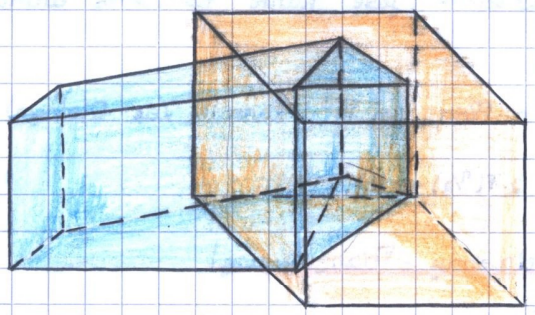
- Einwickeln der ganzen Punktmenge (siehe 2d) kostet im schlechtesten Fall $\Theta(n^2)$. vgl. S.O.
- Jeder Schritt verläuft genauso wie im 2-dim. Fall mit folgendem Unterschied:
  a) verwende 3d - Orientierungsprädikat
  b) 2 Pkte (ehem. Kante $\overline{ab}$) fest, man suche Pkt c mit minimalem Winkel.

Wir nutzen aus, dass wir zwei konvexe Polyeder C_1 und C_2 einwickeln.

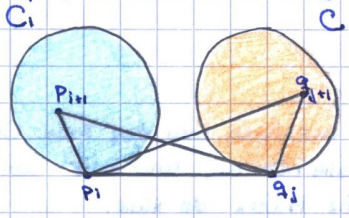
→ Tangentialzylinder.

↳ 6.2.2.2. Situation (Mischschritt):





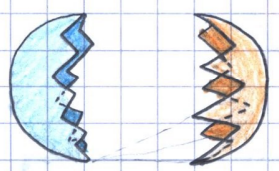
Sei $p_i \in C_1$ und $q_j \in C_2$ so, dass die Kante $\overline{p_i q_j}$ der aktuellen Rand des Einwickelzylinders darstellt.
Initialisierung: $\overline{p_1 q_1}$ durch Berechnung der Tangente in den Projektionen



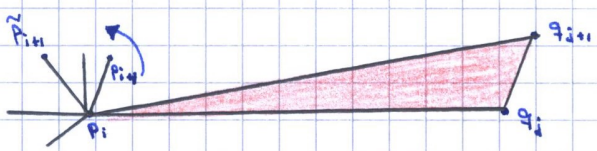
Zwei Mögl: a). Nächster Berührpkt der Einwickelebene p_{i+1} liegt in C_1.
⇒ füge $\Delta\, p_i q_j p_{i+1}$ hinzu. (zum Zylinder) und die Drehachse der Ebene geht nun durch $\overline{p_{i+1} q_j}$
b) Nächster Berührpkt q_{j+1} in C_2
(analog wie oben)

Auf diese Weise entstehen auf jeder Seite zwei Berührkreise $p_1 .. p_e$ und $q_1 .. q_k$. Nach Konstruktion des Zylinders (Gleichzeitig) müssen die abgedeckten Teile von C_1 bzw. C_2 entfernt werden.
Diese Änderungen der Datenstruktur kosten $O(n)$ Zeit ⇒ ok.

⟶ 6.2.2.3. Problem: Wie findet man die Berührkreise $p_1 .. p_e$ und $q_1 .. q_k$ in Zeit $O(n)$?



⟶ 6.2.2.4. Beobachtung:
1). Die Kandidaten für p_{i+1} und q_{j+1} müssen Nachbarn von p_i bzw q_j sein.
Dies folgt aus Konvexität von C_1 und C_2.
⇒ Es genügt in jedem Schritt die Nachbarknoten von p_i und q_j anzuschauen
Bem: Darauf folgt aber noch nicht $O(n)$ Laufzeit, denn:
a) der Grad eines Knotens kann $O(n)$ sein.
b) p_i oder q_j kann für eine große Zahl von Schritten beibehalten werden.
Dann müsste man in jedem dieser Schritte immer wieder alle $O(n)$ Nachbarn untersuchen.
2). Es liegt jedoch eine Monotonie-Eigenschaft vor:
Wenn zB. p_i in einem Schritt fest gehalten wird (d.h. Minimum der beiden Kandidaten p_{i+1} und q_{j+1} war q_{j+1}), dann folgt der neue Kandidat $\tilde{p}_{i+1}$ (bzgl. der neuen Drehachse) nach dem Knoten p_{i+1} in der Adjazenzliste von p_i oder ist gleich p_{i+1}. (ohne Beweis).



1) + 2) ⇒ Insgesamt wird im Durchschnitt jede Kante höchstens einmal betrachtet.
⇒ Laufzeit: $O(n)$

⟶ 6.2.2.5. Satz: Die Konv. Hülle von n Pkte im $\mathbb{R}^3$ kann in Zeit $O(n \log n)$ berechnet werden.
Beweis: a) sortieren $\rightarrow O(n \log n)$
b) Divide & Conquer $\rightarrow T(n) = 2 \cdot T(n/2) + O(n)$.

6.3. Anwendung von 3d-Konvexe Hülle (Delaunay-Triangulierung).

6.3.1. Einführung:

Ziel: Berechnung des Voronoi-Diagramms (Bzw. Delaunay-Triangulierung) im $\mathbb{R}^2$.
dualer Graph zum Voronoi-Diagramm.

Triangulière eine Menge $S \subset \mathbb{R}^2$ so, dass $\forall \Delta abc$ gilt:
Der Umkreis von Δabc enthält im Innern keinen Pkt aus S

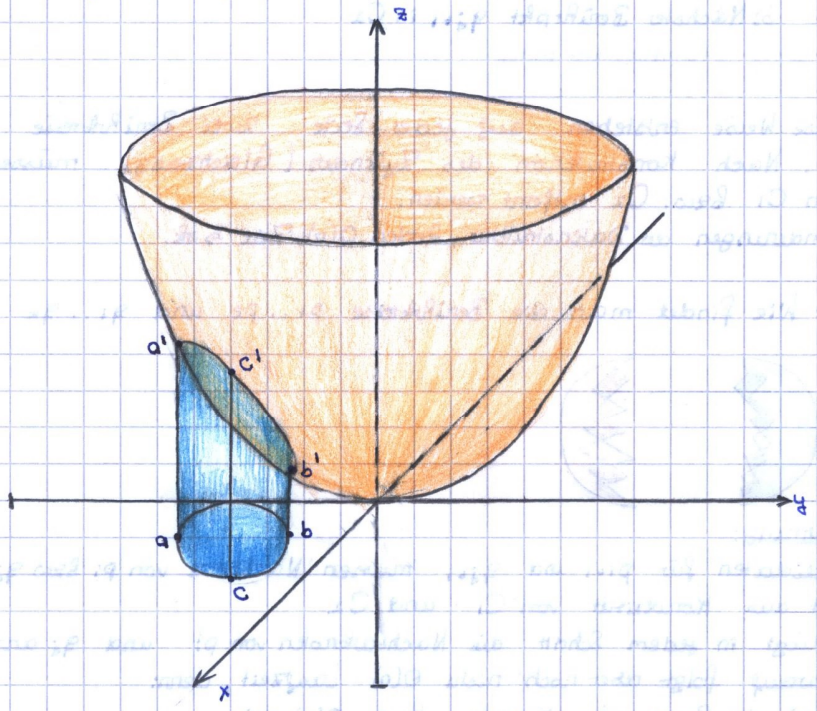
Verbindung zw Konvexen Hülle im $\mathbb{R}^3$:

6.3.2. Idee:

Übersetze die Tatsache, dass alle Pkte außerhalb oder auf Kreis $O(a,b,c)$ liegen, in die Eigenschaft der konvexen Hülle, dass alle Pkte auf einer Seite der Ebene durch drei Pkte liegen. Diese drei Pkte definieren eine Fläche der Hülle.

6.3.3. Umsetzung:

Projiziere jeden Pkt $p \in S \subset \mathbb{R}^2$, $p = (x,y)$ auf einen Paraboloiden, dh. bilde p ab auf $p' = (x,y,z)$ mit $z = x^2 + y^2$



Betrachte nun Delaunay-Eigenschaft:
Betr. beliebigen Kreis im $\mathbb{R}^2$ durch 3 Pkte a,b,c
Dann definieren die gelifteten Pkte a',b',c' eine Ebene E(abc) im $\mathbb{R}^3$, die den Paraboloiden schneidet. (in eine Elipse).
Dann gilt.
1) Alle Pkte auf dem Rand des Kreises $O(a,b,c)$ werden abgebildet auf diese Elipse.
2). Alle Pkte im Inneren werden auf die positive Seite der Ebene E(abc) abgebildet.
3) Alle Pkte außen — " — negative — " —