2.1. Konvexe Hülle in 2D

Wir wollen die konvexe Hülle einer Menge in der Ebene bestimmen.

Def. 2.1: Eine Teilmenge S der Ebene ist konvex gdw
für jedes Paar $p, q \in S$ das Liniensegment $\overline{pq}$ in S liegt.

Def. 2.2: Die konvexe Hülle *CH*(*S*) einer Teilmenge S ist die kleinste konvexe Menge, die *S* enthält, d. h. *CH*(*S*) ist der Schnitt aller S enthaltenden konvexen Mengen.

Konkret wollen wir die konvexe Hülle einer endlichen Menge $P = \{p_1, \ldots, p_n\}$ von Punkten in der Ebene $\mathbb{R}^2$ berechnen.

2.1. Konvexe Hülle in 2D

Als konvexe Hülle ergibt sich ein konvexes Polygon! Dieses wollen wir als Liste von Punkten aus P angeben, die die Eckpunkte von CH(P) im Uhrzeigersinn enthält.

input = set of points:

$p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9$

output = representation of the convex hull:

p_4, p_5, p_8, p_2, p_9

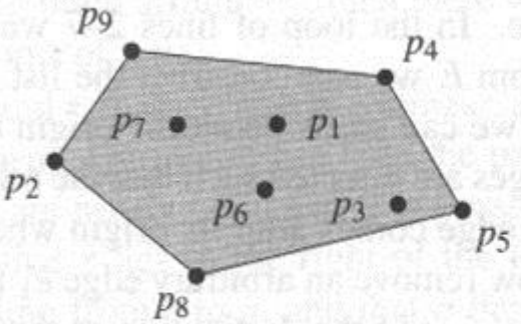
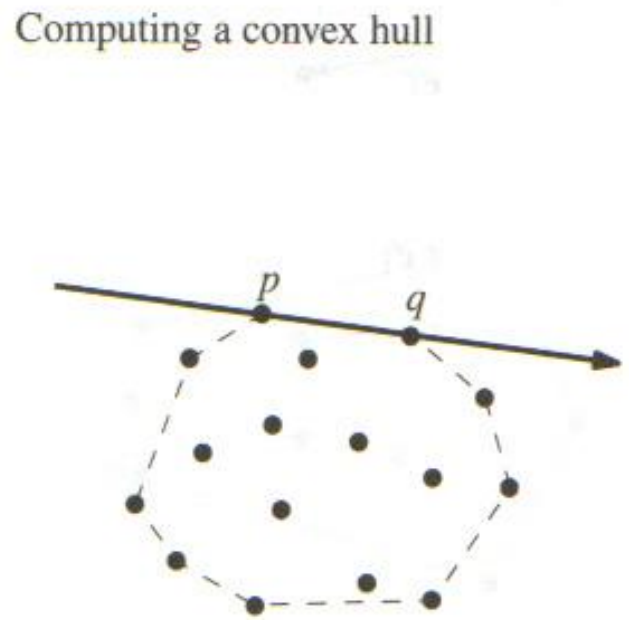


Figure 1.1
Computing a convex hull

Leider ist unsere obige Definition der konvexen Hülle nicht wirklich hilfreich. Daher nutzen wir folgende Aussage über die Eckpunkte der konvexen Hülle.

Satz 2.3: Die Kanten $\overline{pq}$ von CH(P) besitzen zwei Eckpunkte $p, q \in P$ und alle Punkte von P liegen rechts von $\overline{pq}$.

2.1. Konvexe Hülle in 2D



2.1. Konvexe Hülle in 2D

Dies liefert einen ersten Algorithmus

Algorithm SLOWCONVEXHULL(P)
Input. A set P of points in the plane.
Output. A list $\mathcal{L}$ containing the vertices of $\mathcal{CH}(P)$ in clockwise order.

1. $E \leftarrow \emptyset$.
2. **for** all ordered pairs $(p,q) \in P \times P$ with p not equal to q
3. **do** *valid* $\leftarrow$ **true**
4. **for** all points $r \in P$ not equal to p or q
5. **do if** r lies to the left of the directed line from p to q
6. **then** *valid* $\leftarrow$ **false**.
7. **if** *valid* **then** Add the directed edge $\overrightarrow{pq}$ to E.
8. From the set E of edges construct a list $\mathcal{L}$ of vertices of $\mathcal{CH}(P)$, sorted in clockwise order.

2.1. Konvexe Hülle in 2D

Bemerkungen:

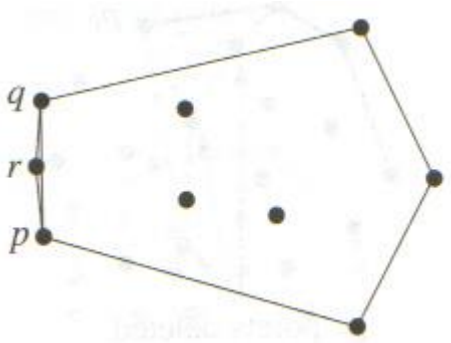
- In Zeile 5 berechnen wir, ob ein Punkt rechts oder links einer Geraden liegt. Eine Implementierung könnte etwa über das Vektorprodukt erfolgen. Abstandsberechnung zu einer Geraden ist eine weitere Variante. Die Operation gelingt in jedem Fall in konstanter Zeit. Wichtig ist, dass man bei einer Implementierung hier vorsichtig ist, da Zeit und Effizienz verloren gehen können → Bibliotheken benutzen!
- In Zeile 8 müssen die Segmente sortiert werden, so dass mit einem beliebigen Segment begonnen wird und dann das daran anschließende (gerichtete) Segment gefunden wird. Als Aufwand ergibt sich $O(k^2)$ bzw. $O(k \log k)$, wenn k Segmente/Punkte die konvexe Hülle erzeugen.
- Insgesamt ergeben sich $n(n\text{-}1)$ Paare, die gegen $n\text{-}2$ Punkte getestet werden: $O(n^3)$.
- Ferner liefert der letzte Schritt $O(k \log k), k \leq n$ also insgesamt: $O(n^3)$.

2.1. Konvexe Hülle in 2D

Das ist sicher nicht akzeptabel für mehr als 10000 Punkte! Ferner gibt es weitere versteckte Probleme:

1) Wie behandeln wir Punkte, die auf einer Linie liegen?

2) Wie berücksichtigen wir, dass bei fast kollinearen Punkten (Rundungsfehler) Punkte auf die falsche Seite liegen oder gar keiner von drei Punkten *p, q, r* rechts von den anderen beiden liegt?



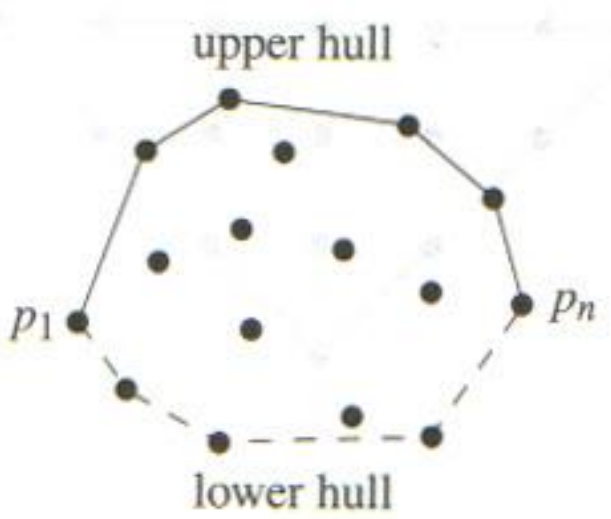
2.1. Konvexe Hülle in 2D

Zu 1): Wir müssen in Zeile 5 testen, ob *r* links von $\overline{pq}$ oder außerhalb der offenen Strecke $\overline{pq}$ liegt. (Mehrfache Punkte schließen wir hier aus.)

Zu 2): Dieses Problem ist wirklich ernst, denn wenn keine Verbindung von *p* nach *q* existiert, wird unser Algorithmus in Zeile 8 nicht enden oder das Programm aussteigen. Unserem Algorithmus fehlt die **Robustheit**.

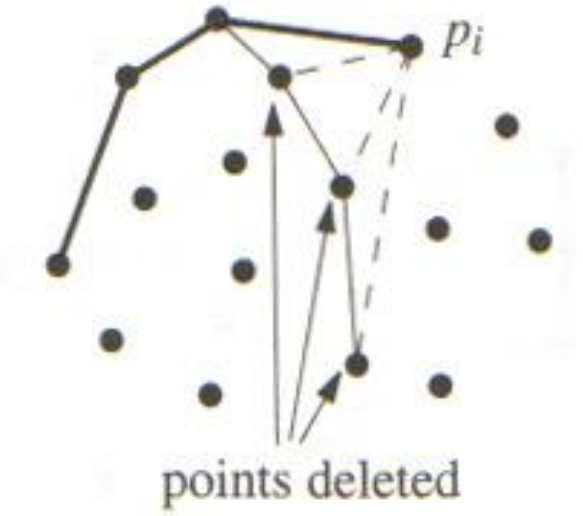
2.1. Konvexe Hülle in 2D

Um zu einem verbesserten Algorithmus zu gelangen, nutzen wir einen inkrementellen Ansatz, indem wir einen Punkt nach dem anderen zu P hinzufügen und unsere konvexe Hülle jeweils anpassen, sofern nötig.
Wir sortieren die Punkte nach aufsteigender x-Koordinate und berechnen nur die obere Hälfte der konvexen Hülle. (Die untere Hälfte erhalten wir durch Sortieren nach absteigender x-Koordinate).



2.1. Konvexe Hülle in 2D

Die Idee liegt darin, dass wir entlang der oberen Hülle in einem konvexen Polygon stets nach rechts abbiegen und somit eine Linksbiegung auf einen Fehler im letzten Teilstück der konvexen Hülle hinweist. Beachtet man, dass in diesem Fall auch vorherige Stücke zu testen sind, ist der Algorithmus fertig.



2.1. Konvexe Hülle in 2D

Algorithm CONVEXHULL(P)
Input. A set P of points in the plane.
Output. A list containing the vertices of $\mathcal{CH}(P)$ in clockwise order.

1. Sort the points by x-coordinate, resulting in a sequence $p_1, \ldots, p_n$.
2. Put the points p_1 and p_2 in a list $\mathcal{L}_{\text{upper}}$, with p_1 as the first point.
3. **for** $i \leftarrow 3$ **to** n
4. **do** Append p_i to $\mathcal{L}_{\text{upper}}$.
5. **while** $\mathcal{L}_{\text{upper}}$ contains more than two points **and** the last three points in $\mathcal{L}_{\text{upper}}$ do not make a right turn
6. **do** Delete the middle of the last three points from $\mathcal{L}_{\text{upper}}$.
7. Put the points p_n and p_{n-1} in a list $\mathcal{L}_{\text{lower}}$, with p_n as the first point.
8. **for** $i \leftarrow n-2$ **downto** 1
9. **do** Append p_i to $\mathcal{L}_{\text{lower}}$.
10. **while** $\mathcal{L}_{\text{lower}}$ contains more than 2 points **and** the last three points in $\mathcal{L}_{\text{lower}}$ do not make a right turn
11. **do** Delete the middle of the last three points from $\mathcal{L}_{\text{lower}}$.
12. Remove the first and the last point from $\mathcal{L}_{\text{lower}}$ to avoid duplication of the points where the upper and lower hull meet.
13. Append $\mathcal{L}_{\text{lower}}$ to $\mathcal{L}_{\text{upper}}$, and call the resulting list $\mathcal{L}$.
14. **return** $\mathcal{L}$

2.1. Konvexe Hülle in 2D

Bemerkungen:

- Wenn zwei Punkte die gleiche *x*-Koordinate haben, müssen wir sie anhand der *y*-Koordinate sortieren.
- Wenn drei Punkte auf einer Geraden liegen, sollte der Mittlere entfernt werden. Unser Test sollte also ermitteln, ob eine echte Rechtsdrehung erfolgt.
- Im Falle von Rundungsfehlern entfernen wir Punkte, die wir nicht entfernen sollten oder wir behalten Punkte, die entfernt werden sollten. Beides führt zu kleinen, vermutlich nicht sichtbaren Fehlern, aber unser Algorithmus arbeitet ordentlich! Das einzige ernste Problem bzgl. Robustheit sind sehr nahe Punkte, was wir durch Zusammenlegen dieser Punkte beseitigen können.

2.1. Konvexe Hülle in 2D

Als Regel zur Entwicklung geometrischer Algorithmen seien drei Phasen festgehalten:

1) Ignoriere zunächst alle degenerierten Fälle wie kollineare Punkte oder vertikale Segmente!
2) Betrachte die degenerierten Fälle und versuche sie in den Algorithmus zu integrieren! Zunächst wird man Fallunterscheidungen bevorzugen, aber oft lässt sich eine geschicktere Lösung finden, die viele Fälle vermeidet.
3) Implementiere den Algorithmus durch sorgfältiges Umsetzen der Basisoperationen! Rundungsfehlern gebührt hier große Aufmerksamkeit, da sie oft zu Frustrationen führen. Sie sind nicht einfach zu vermeiden. Neben exakter Arithmetik hilft nur die genaue Untersuchung möglicher Fälle und eine genaue Spezifikation der Ergebnisse des Algorithmus, sowie der Anforderungen der Anwendungen. (Reicht es letztlich, die richtigen Pixel der konvexen Hülle zu zeichnen?).

2.1. Konvexe Hülle in 2D

Literatur:

Der Algorithmus stammt aus [A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. Inform. Process. Letters 9:216-219,1979] und basiert auf [R.L. Graham. An efficient algorithm for determining the convex hull of a finite set. Inform. Process. Letters 1:132-133,1972].

Es ist bekannt, dass $\Omega(n \log n)$ eine untere Schranke des Problems ist [A. C. Yao. A lower bound to finding convex hulls. J. ACM 28:780-787, 1981].

Allerdings gilt dies nur, wenn viele Punkt ein der konvexen Hülle liegen. Deshalb kann man dies verbessern, wenn man annimmt, dass in der Regel die Anzahl der Punkte h in der konvexen Hülle klein ist. [D.G. Kirkpatrick, R. Seidel. The ultimate planar convex hull algorithm? SIAM J. Computing 15:287-299, 1986.] und die einfachere Variante [T.M.Y. Chan. Output-sensitive results on convex hulls, extreme points, and related problems. In Proc. 11th Annual ACM Symp. Comput. Geom., 1995, 10-19.] erreichen $O(n \log h)$!

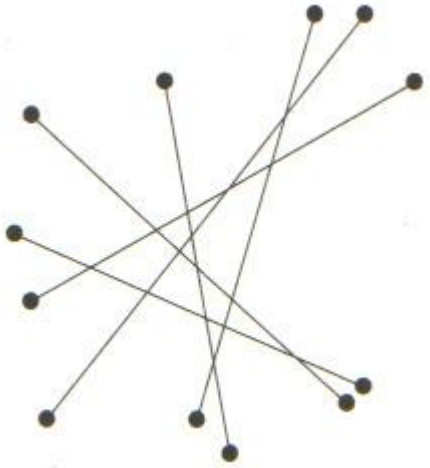
2.2. Schnitte von Liniensegmenten

Wir wenden uns nun dem Problem (2) aus §1 zu. Aus zwei Mengen S_1, S_2 von Liniensegmenten möchten wir alle Schnittpunkte der Segmente aus S_1 mit denen aus S_2 ermitteln.

- Wir legen fest, dass sich zwei Segmente auch schneiden, falls der Endpunkt des einen Segmentes auf dem anderen liegt. Wir betrachten Segmente also als abgeschlossen.
- Um es einfacher zu machen, betrachten wir Schnitte in $S = S_1 \cup S_2$.
  (Unser Problem lässt sich dann lösen, indem wir bei jedem Schnitt fragen, ob die Segmente zu verschiedenen Mengen gehören.)

2.2. Schnitte von Liniensegmenten

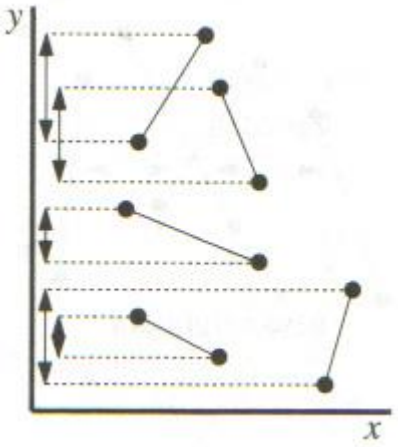
Offensichtlich ist der Schnitttest zweier Segmente in konstanter Zeit zu lösen und folglich eine brute-force Lösung in $O(n^2)$ Schritten möglich. Dies ist sogar optimal, falls jedes Segment jedes andere schneidet.



Allerdings ist dies in der Praxis meistens nicht der Fall. Wir suchen also einen output-sensitiven Algorithmus, dessen Komplexität von der Größe des Output (Anzahl der Schnittpunkte) abhängt.
Wie soll das gehen?

2.2. Schnitte von Liniensegmenten

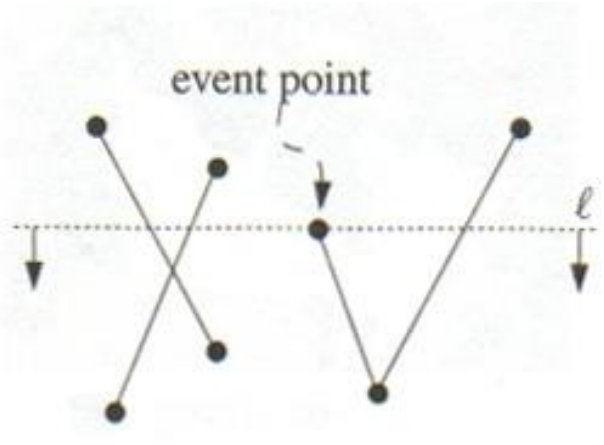
Im Wesentlichen wollen wir ausnutzen, dass sich nur "nahe" Segmente auch schneiden können. Liegen zwei Segmente S_1, S_2 etwa so, dass sich das Intervall der y-Koordinaten der Punkte in S_1 außerhalb des Intervalls der y-Koordinaten der Punkte in S_2 befindet, so gibt es keinen Schnitt.



Um nur potentielle Schnitte testen zu müssen, verfolgen wir eine horizontale Gerade (Sweep Line), die sich von oben nach unten bewegt und testen nur Segmente, welche die Gerade schneiden $\Rightarrow$ **Plane Sweep**.

2.2. Schnitte von Liniensegmenten

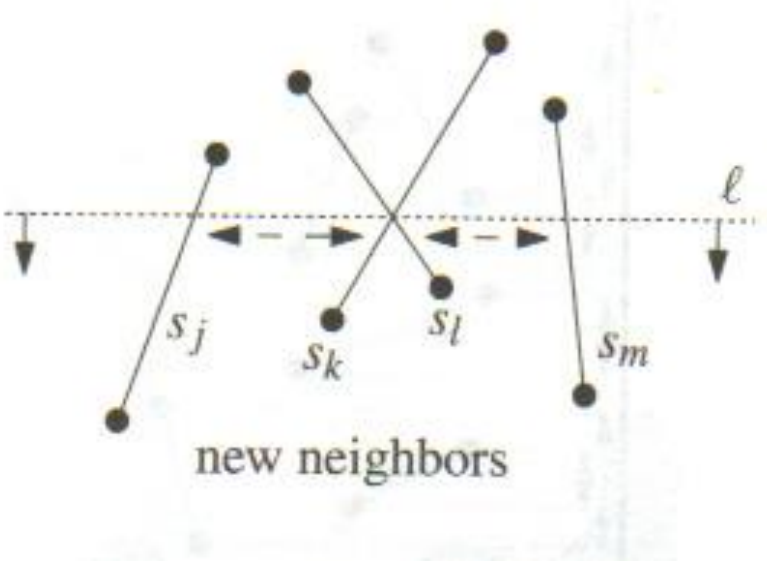
Als Zustand unserer Sweep Line definieren wir die von ihr geschnittenen Liniensegmente. Dieser Zustand ändert sich, wenn sich die Gerade nach unten bewegt, und zwar stets dann, wenn ein Segmentendpunkt erreicht wird, da genau dann ein Segment ausscheidet oder ein neues hinzukommt. Diese Punkte nennen wir Ereignispunkte (**Event Points)**.



Wenn ein neues Segment zum Zustand der Sweep Line hinzukommt, testen wir es auf Schnitte mit allen Segmenten im Zustand.

2.2. Schnitte von Liniensegmenten

Leider stellt sich das Problem etwas schwieriger dar: Wir können noch immer viel zu viele Tests durchführen, etwa bei einer Menge vertikaler Segmente, die die *x*-Achse schneiden.
Daher ordnen wir die Segmente entlang der Sweep Line und testen nur Nachbarn auf Schnitte. Ferner führen wir Schnitte als weitere Ereignisse (Events) ein, da sich dort die Reihenfolge der Segmente ändert.



2.2. Schnitte von Liniensegmenten

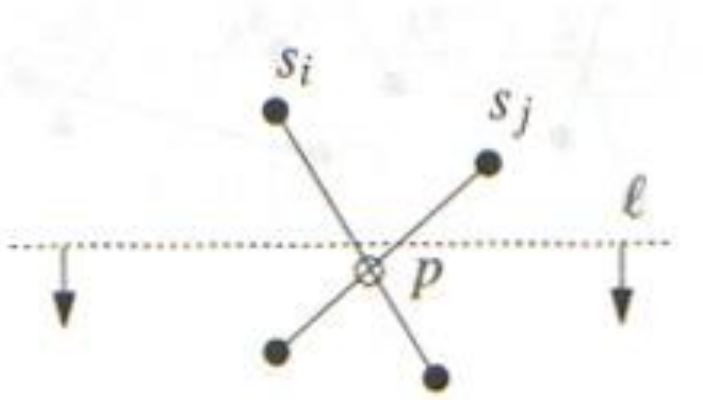
Außer einigen Spezialfällen (horizontale Segmente, keine überlappenden Segmente, kein Schnitt von mehr als zwei Segmenten in einem Punkt) ist unser Algorithmus nun korrekt:

Lemma 2.5:
Seien S_i und S_j zwei nicht-horizontale Segmente, die sich im inneren Punkt p schneiden, und es gebe kein drittes Segment durch p. Dann gibt es einen Eventpunkt oberhalb p, an dem S_i und S_j benachbart werden und somit auf Schnitt getestet werden.

Beweis:
Einerseits sind S_i und S_j für eine horizontale Gerade l ausreichend nahe oberhalb von p benachbart. (Nimm an, dass alle Events oberhalb p gefunden werden und wähle l zwischen dem niedrigsten dieser Events und p !*)*
Andererseits ist der Zustand der Sweep Line oberhalb aller Segmente leer, also wird ein Event behandelt, an dem S_i und S_j Nachbarn werden.

QED

2.2. Schnitte von Liniensegmenten



Unser Algorithmus funktioniert also bis auf Spezialfälle und wir können uns an die Datenstrukturen begeben.
Für die Event Queue Q fordern wir:

- Eine **Remove-Funktion**, die den nächsten Event Point liefert, und zwar nach y- Koordinate und bei gleichen *y*-Werten nach *x*-Koordinate geordnet.
- Eine **Insert-Funktion**, die mehrfaches Einfügen des gleichen Events unterbindet (etwa bei zwei Segmenten mit gleichem Startpunkt)
- Als Implementierung verwenden wir einen ausgeglichenen binären Suchbaum, so dass beide Operationen $O(\log m)$ Schritte für m Elemente in *Q* benötigen.

2.2. Schnitte von Liniensegmenten

Für die Zustandsstruktur *T* bestehend aus der geordneten Liste der Segmente benötigen wir:

- **Insert**
- **Remove**
- **LeftNeighbor**, **RightNeighbor** gemäß Sortierung

Wieder verwenden wir einen ausgeglichenen binären Suchbaum. (Dabei ist es zuweilen etwas einfacher die Objekte nur in den Blättern abzulegen, aber dies ist nicht nötig und kostet Speicher.)
Wieder erfordern alle Operationen $O(\log n)$ Schritte.

2.2. Schnitte von Liniensegmenten

Der Algorithmus hat folgende Gestalt.

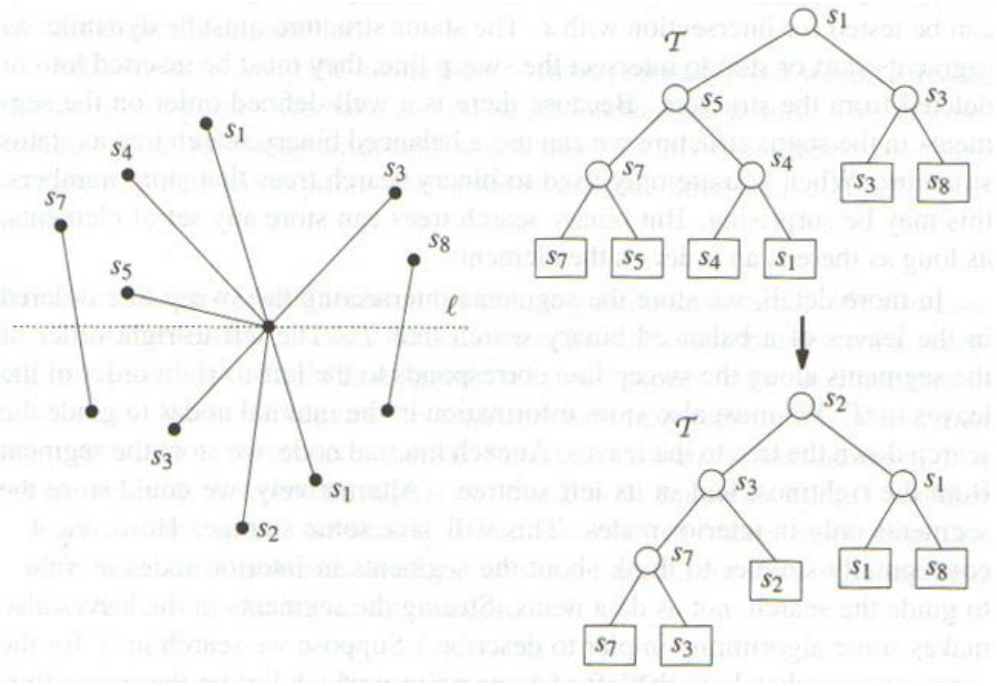
Algorithm FINDINTERSECTIONS(S)
Input. A set S of line segments in the plane.
Output. The set of intersection points among the segments in S, with for each intersection point the segments that contain it.

1. Initialize an empty event queue Q. Next, insert the segment endpoints into Q; when an upper endpoint is inserted, the corresponding segment should be stored with it.
2. Initialize an empty status structure $\mathcal{T}$.
3. **while** Q is not empty
4. **do** Determine the next event point p in Q and delete it.
5. HANDLEEVENTPOINT(p)

2.2. Schnitte von Liniensegmenten

HANDLEEVENTPOINT(p)
1. Let $U(p)$ be the set of segments whose upper endpoint is p; these segments are stored with the event point p. (For horizontal segments, the upper endpoint is by definition the left endpoint.)
2. Search in $\mathcal{T}$ for the set $S(p)$ of all segments that contain p; they are adjacent in $\mathcal{T}$. Let $L(p) \subset S(p)$ be the set of segments whose lower endpoint is p, and let $C(p) \subset S(p)$ be the set of segments that contain p in their interior.
3. **if** $L(p) \cup U(p) \cup C(p)$ contains more than one segment
4. **then** Report p as an intersection, together with $L(p)$, $U(p)$, and $C(p)$.
5. Delete the segments in $L(p) \cup C(p)$ from $\mathcal{T}$.
6. Insert the segments in $U(p) \cup C(p)$ into $\mathcal{T}$. The order of the segments in $\mathcal{T}$ should correspond to the order in which they are intersected by a sweep line just below p. If there is a horizontal segment, it comes last among all segments containing p.
7. ($*$ Deleting and re-inserting the segments of $C(p)$ reverses their order. $*$)
8. **if** $U(p) \cup C(p) = \emptyset$
9. **then** Let s_l and s_r be the left and right neighbors of p in $\mathcal{T}$.
10. FINDNEWEVENT(s_l, s_r, p)
11. **else** Let s' be the leftmost segment of $U(p) \cup C(p)$ in $\mathcal{T}$.
12. Let s_l be the left neighbor of s' in $\mathcal{T}$.
13. FINDNEWEVENT(s_l, s', p)
14. Let s'' be the rightmost segment of $U(p) \cup C(p)$ in $\mathcal{T}$.
15. Let s_r be the right neighbor of s'' in $\mathcal{T}$.
16. FINDNEWEVENT(s'', s_r, p)

2.2. Schnitte von Liniensegmenten



An event point and the changes in the status structure

Bemerkung:
Wenn es in Zeile 8–16 keinen Nachbarn gibt, werden die entsprechenden Schnittberechnungen nicht durchgeführt.

2.2. Schnitte von Liniensegmenten

Um auch mit horizontalen Segmenten umgehen zu können, setzen wir

FINDNEWEVENT(s_l, s_r, p)
1. **if** s_l and s_r intersect below the sweep line, or on it and to the right of the current event point p, and the intersection is not yet present as an event in Q
2. **then** Insert the intersection point as an event into Q.

2.2. Schnitte von Liniensegmenten

Lemma 2.6:
FINDINTERSECTIONS berechnet alle Schnittpunkte und die betroffenen Segmente korrekt.
Beweis: Induktion über die Reihenfolge der Event Points
Seien alle Event Points *q* vor *p* korrekt behandelt. Seien *U*(*p*) die Segmente mit *p* als oberen Endpunkt, *L*(*p*) die Segmente mit *p* als unterem Endpunkt und *C(p*) die Segmente mit *p* im Inneren. Wenn *p* ein Endpunkt eines Segmentes ist, wird *p* zu Beginn in *Q* eingefügt und alle Segmente in *U*(*p*) in *p* gespeichert. Wenn *p* behandelt wird, werden Segmente in *L*(*p*) und *C*(*p*) in T sein und dann in Zeile 2 von HANDLEEVENTPOINT gefunden.
Wenn *p* kein Endpunkt eines Segmentes ist, haben alle betroffenen Segmente *p* als inneren Punkt. Wir ordnen diese Segmente nach dem Winkel und betrachten zwei benachbarte Segmente S_i und S_j. Nach dem Beweis von Lemma 2.5 gibt es dann einen Eventpoint *q* vor *p,* an dem S_i und S_j Nachbarn werden und *p* entdeckt wird. (Dass wir in 2.5. horizontale Segmente ausgeschlossen haben, lässt sich schnell korrigieren.)

QED

2.2. Schnitte von Liniensegmenten

Es gilt ferner, dass der Algorithmus $O(n \log n + k \log n)$ Schritte benötigt, wobei k die Anzahl sich schneidender Segmente ist. Es gilt sogar

Lemma 2.7:
Die Laufzeit des Algorithmus FINDINTERSECTIONS für eine Menge *S* von *n* Liniensegmenten ist $O(n \log n + l \log n)$, wobei *I* die Anzahl von Schnittpunkten von Segmenten in *S* ist.

Beweis:
Der Aufbau von *Q* mit den Endpunkten erfordert $O(n \log n)$ Zeit. T ist zunächst leer, also $O(1)$. Eine Eventbehandlung erfordert drei Operationen auf *Q*, das Löschen von *p* in Zeile 4 von FINDINTERSECTIONS und ein oder zwei Aufrufe von FINDNEWEVENT mit bis zu zwei neuen Events in *Q*. Also ist das Update von *Q* $O(\log n)$. Wir führen außerdem Operationen auf T (Einfügen, Löschen, Nachbarsuche) durch und zwar linear in $m(p) = \#\big(U(p) \cup C(p) \cup L(p)\big)$. Mit $m \coloneqq \sum_p m(p)$ folgt $O(m \log n)$ für den Algorithmus.

2.2. Schnitte von Liniensegmenten

Es gilt sicher $\mathrm{m} = O(n + k)$, da wir für *m*(*p*) > 1 alle betroffenen Segmente auf einmal liefern. Um $\mathrm{m} = O(n + l)$ zu zeigen, fassen wir unser Problem als Graph mit Endpunkten und Schnittpunkten als Knoten und Segment(-teilen) als Kanten auf. Dann ist *m*(*p*) durch den Grad des Knotens beschränkt und da jede Kante im Graph zwei Knoten trifft, gilt $m < 2ne$ mit n_e Kanten im Graph. Mit der Anzahl n der Endpunkte und l der Schnitte ergibt sich $n_v \leq 2n + l$ mit n_v Knoten im Graph.
Wegen $n_e = O(n_v)$ folgt die Aussage.

QED

Nebenbemerkung:
Die Euler-Formel in der Ebene lautet: $n_v - n_e + n_f = 2$ wobei n_f die Anzahl der Flächen ist.
Es gilt $n_f < \frac{2}{3} n_e$, da jede Kante nur zwei Flächen (Facetten) beranden kann und jede Fläche von mindestens 3 Kanten berandet werden muss. Bei den Facetten ist die äußere Facette mitgezählt.

2.2. Schnitte von Liniensegmenten

Für den Speicherplatz gilt, dass *T* nie mehr als alle *n* Segmente enthält, also $O(n)$ Speicherplätze benötigt. *Q* kann größer sein, aber es reicht, nur die Schnitte gerade in *T* benachbarter Segmente zu speichern, also $O(n)$ Speicherplätze für *Q*.

Theorem 2.8.:
Sei *S* eine Menge von *n* Liniensegmenten in der Ebene. Alle Schnittpunkte in *S* mit den betroffenen Segmenten können in $O(n \log n + l \log n)$ Schritten und $O(n)$ Speicherplätzen gefunden werden, wobei *I* die Anzahl der Schnittpunkte ist.

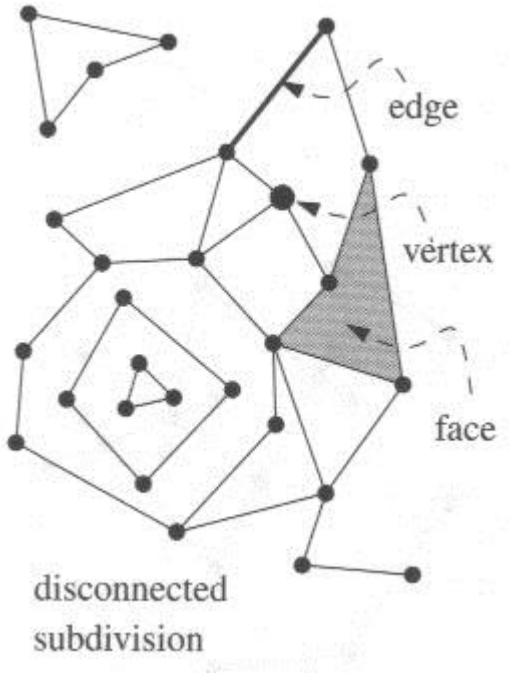
2.3. Überlagerung planarer Unterteilungen

Bisher haben wir nur Schnitte von Flüssen mit Straßen betrachtet. Viel wichtiger ist jedoch die Überlagerung von ebenen Unterteilungen, etwa Bodennutzung und Niederschlagsmenge in mm. Es handelt sich also um den Schnitt zweier planarer Unterteilungen, die als Einbettung von Graphen gesehen werden können.

Eine solche Einbettung zerlegt die Ebene in

- Eckpunkte
- Kanten (ohne die Eckpunkte, also offen)
- Facetten (maximal zusammenhängende Teilmengen der Ebene, ohne Eckpunkte und Kanten)

2.3. Überlagerung planarer Unterteilungen



Wichtig: Zu einem beliebigen Punkt in der Ebene wollen wir hier nicht die richtige Facette, Kante oder den richtigen Eckpunkt finden. Dieses (in der Visualisierung sehr bedeutsame) Problem behandeln wir später.

2.3. Überlagerung planarer Unterteilungen

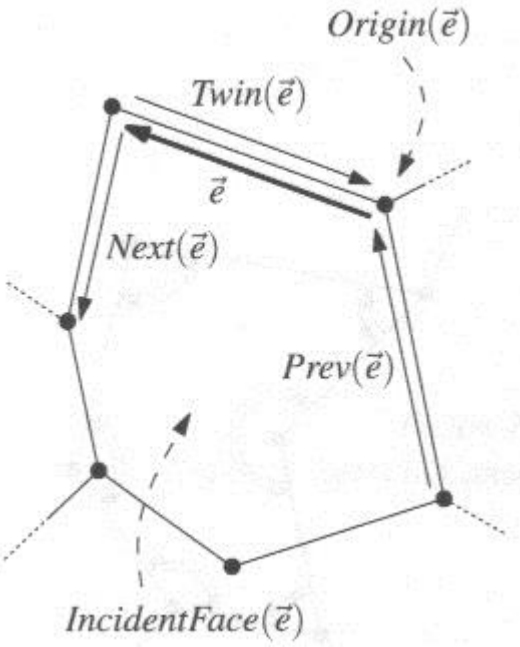
Datenstruktur

Bevor wir Überlagerungen solcher Einbettungen berechnen, brauchen wir eine geeignete Datenstruktur. Typische Operationen auf dieser Datenstruktur sind:

- Alle Kanten gegen den Uhrzeigersinn um einen Punkt angeben
- Alle äußeren Kanten gegen den Uhrzeigersinn um eine Facette angeben
- Alle Ränder von Löchern in der Facette als Kanten im Uhrzeigersinn angeben
- Zu einer Facette die Nachbarfacette entlang einer Kante angeben

2.3. Überlagerung planarer Unterteilungen

Als Lösung benutzen wir eine **verknüpfte Liste gerichteter Halbkanten** mit Anfangspunkt und inzidenter Facette.

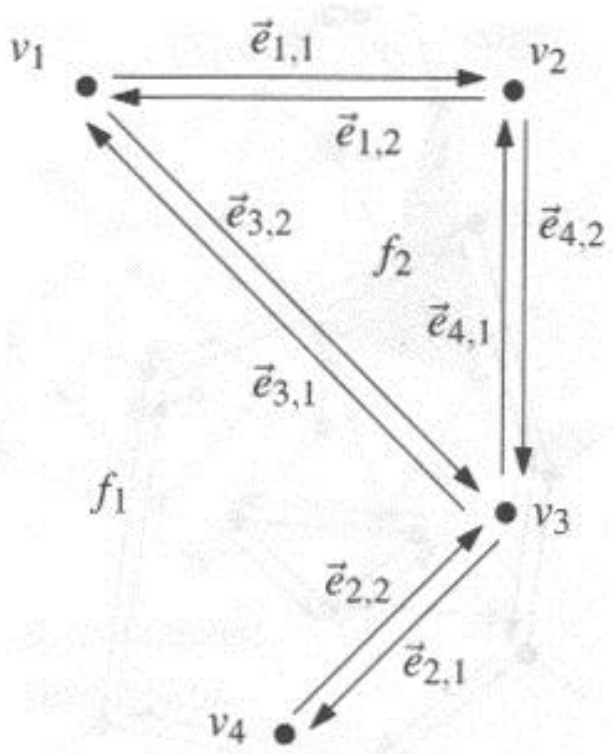


2.3. Überlagerung planarer Unterteilungen

Es gibt drei Felder (Arrays) für die Eckpunkte, die Halbkanten und die Facetten.

- Im Eckpunktfeld werden die Koordinaten *Coordinates*(*v*) jedes Eckpunktes und ein Zeiger (oder Index) einer beliebigen inzidenten Halbkante *IncidentEdge*(*v*) abgelegt.
- Im Feld der Facetten speichern wir zu jeder Facette *f* einen Zeiger *OuterComponent*(*f*) auf eine Halbkante und eine Liste *InnerComponents(f)* mit Zeigern auf genau eine Halbkante für jedes Loch in *f*.
- In das Feld der Halbkanten stellen wir für jede Halbkante *e* einen Zeiger (Index) *Origin*(*e*) des Starteckpunktes von *e*, einen Zeiger (Index) *Twin*(*e*) zur entgegengesetzten Halbkante und einen Zeiger *IncidentFace*(e) zur berandeten Facette, die stets links der Halbkante liegt. *Next*(*e*) und *Prev*(*e*) dienen zur Speicherung der nächsten und vorigen Halbkante längs *IncidentFace*(*e*).

2.3. Überlagerung planarer Unterteilungen



Vertex	Coordinates	IncidentEdge
v_1	(0,4)	$\vec{e}_{1,1}$
v_2	(2,4)	$\vec{e}_{4,2}$
v_3	(2,2)	$\vec{e}_{2,1}$
v_4	(1,1)	$\vec{e}_{2,2}$

Face	OuterComponent	InnerComponents
f_1	**nil**	$\vec{e}_{1,1}$
f_2	$\vec{e}_{4,1}$	**nil**

Half-edge	Origin	Twin	IncidentFace	Next	Prev
$\vec{e}_{1,1}$	v_1	$\vec{e}_{1,2}$	f_1	$\vec{e}_{4,2}$	$\vec{e}_{3,1}$
$\vec{e}_{1,2}$	v_2	$\vec{e}_{1,1}$	f_2	$\vec{e}_{3,2}$	$\vec{e}_{4,1}$
$\vec{e}_{2,1}$	v_3	$\vec{e}_{2,2}$	f_1	$\vec{e}_{2,2}$	$\vec{e}_{4,2}$
$\vec{e}_{2,2}$	v_4	$\vec{e}_{2,1}$	f_1	$\vec{e}_{3,1}$	$\vec{e}_{2,1}$
$\vec{e}_{3,1}$	v_3	$\vec{e}_{3,2}$	f_1	$\vec{e}_{1,1}$	$\vec{e}_{2,2}$
$\vec{e}_{3,2}$	v_1	$\vec{e}_{3,1}$	f_2	$\vec{e}_{4,1}$	$\vec{e}_{1,2}$
$\vec{e}_{4,1}$	v_3	$\vec{e}_{4,2}$	f_2	$\vec{e}_{1,2}$	$\vec{e}_{3,2}$
$\vec{e}_{4,2}$	v_2	$\vec{e}_{4,1}$	f_1	$\vec{e}_{2,1}$	$\vec{e}_{1,1}$

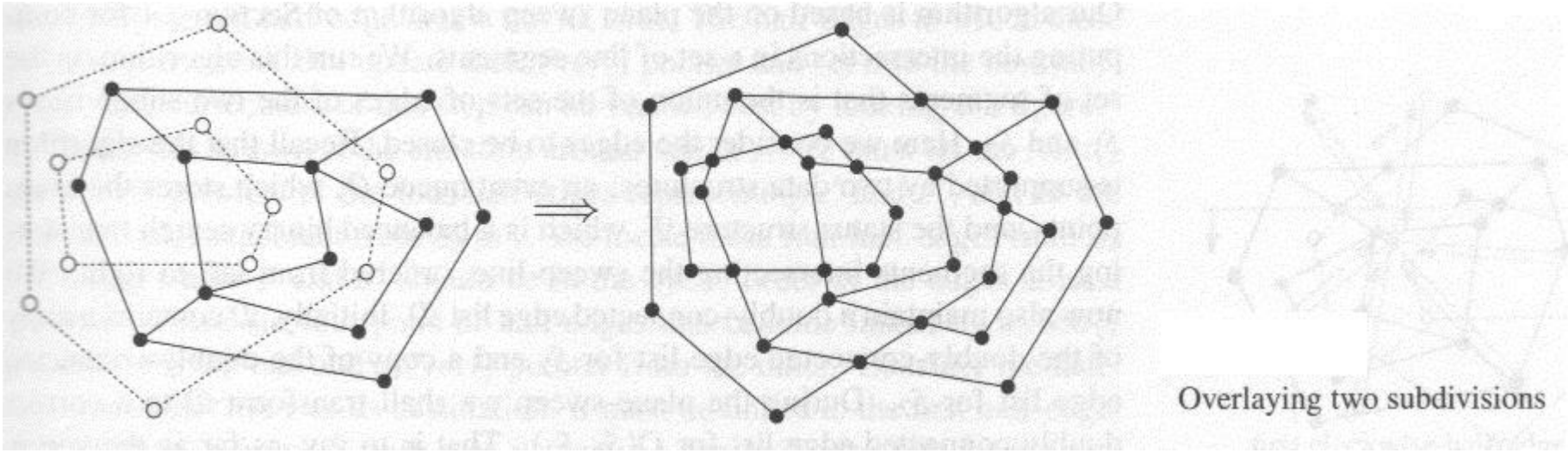
2.3. Überlagerung planarer Unterteilungen

Für Eckpunkte und Halbkanten ist der Speicheraufwand konstant. Im Falle nicht zusammenhängender Unterteilungen benötigt man jedoch Listen für die Facetten. Da aber jede Halbkante nur in einer Liste auftaucht, ist der Speicheraufwand linear bzgl. der Summe von Eckpunkten, Kanten und Facetten.

2.3. Überlagerung planarer Unterteilungen

Mit dieser Datenstruktur können wir nun die Überlagerung zweier Unterteilungen angehen.

Definition 2.9
Sind S_1 und S_2 zwei Unterteilungen (subdivisions) S_1 und S_2 der Ebene, so heiße $O(S_1, S_2)$ die Überlagerung (overlay) von S_1 und S_2.
$O(S_1, S_2)$ enthält eine Facette f gdw
es gibt eine Facette $f_1 \in S_1$ und eine Facette $f_2 \in S_2$,
so dass f eine maximale zusammenhängende Teilmenge von $f_1 \cap f_2$ ist.



Overlaying two subdivisions

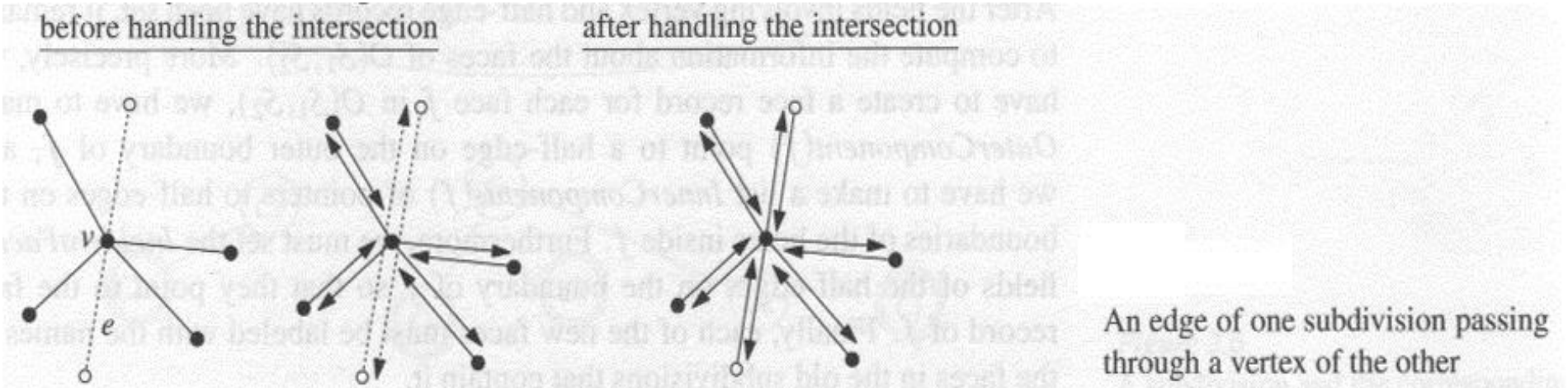
2.3. Überlagerung planarer Unterteilungen

Wir müssen also aus der verknüpften Halbkantenliste von S_1 und der verknüpften Halbkantenliste von S_2 eine verknüpfte Halbkantenliste $O(S_1, S_2)$ erstellen, wobei zu jeder Facette f in $O(S_1, S_2)$ anzugeben ist, aus welchen Facetten $f_1 \in S_1$ und $f_2 \in S_2$ sie entstanden ist. Dies erlaubt es etwa unserem geographischem Informationssystem (GIS) zu jeder Facette in $O(S_1, S_2)$ die Bodennutzung über S_1 und die Niederschlagsmenge über S_2 zu bestimmen.

2.3. Überlagerung planarer Unterteilungen

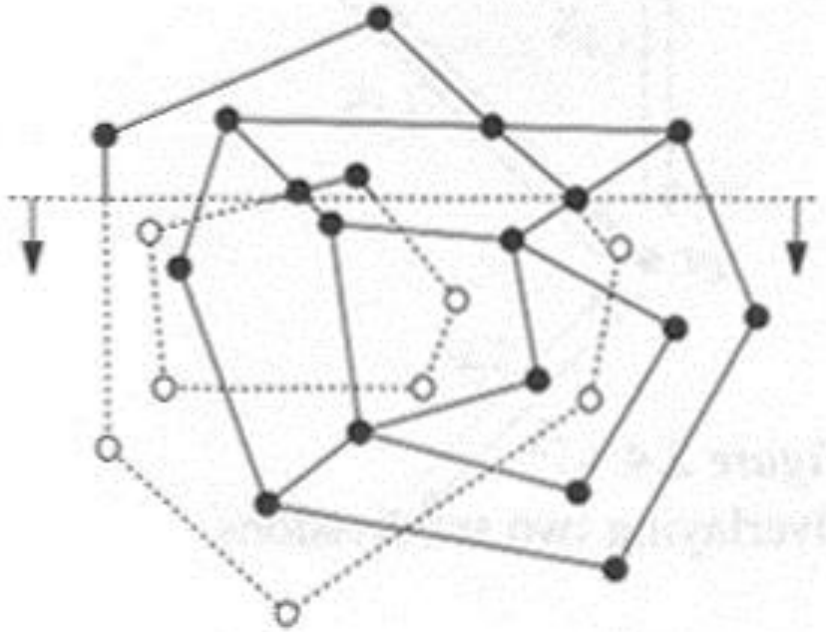
Algorithmus

Zunächst halten wir fest, dass wir alle Eckpunkte aus S_1 und S_2 auch in $O(S_1, S_2)$ haben. Ferner ergeben sich alle Eckpunkte in $O(S_1, S_2)$ durch Hinzunahme der Schnitte (des inneren Teils) der Kanten in S_1 mit denen in S_2! Bei den Halbkanten in S_1 und S_2 können wir alle behalten, die nicht geschnitten werden. Wenn ein Schnitt auftritt, müssen wir in unserer Datenstruktur nicht einmal Einträge entfernen, sondern lediglich zwei (wenn eine Kante von S_1 durch einen Eckpunkt von S_2 läuft) oder vier (wenn zwei Kanten sich schneiden) neue Halbkanten einfügen.



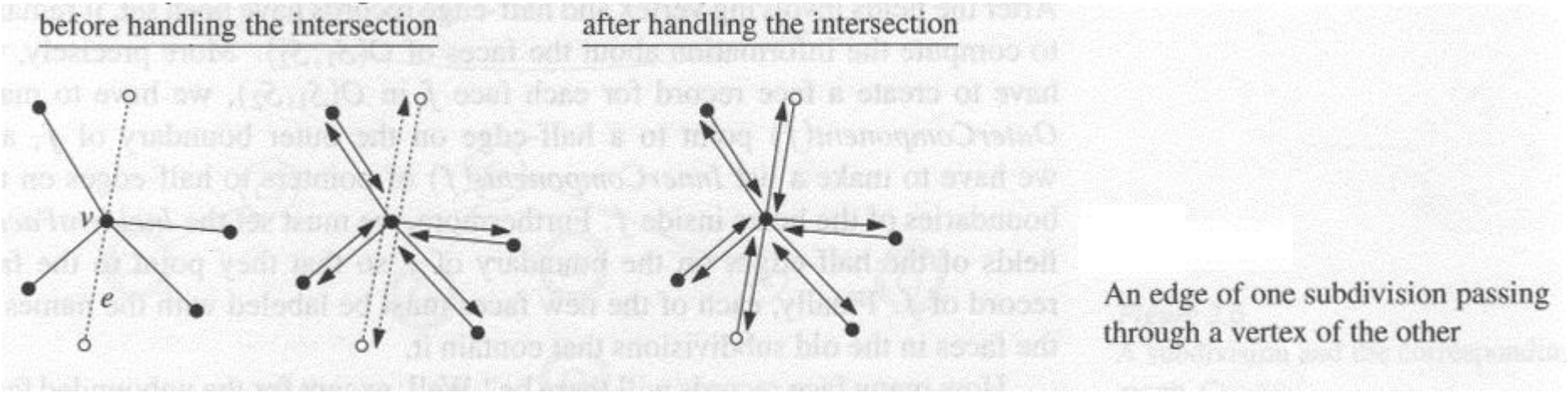
2.3. Überlagerung planarer Unterteilungen

Um die Facetten kümmern wir uns später. Unsere zentrale Idee ist wieder der Plane Sweep Algorithmus für den Schnitt einer Menge von Liniensegmenten. Die dortigen Event Points liefern uns sicher die zusätzlichen Eckpunkte und erlauben das nötige Einfügen neuer Halbkanten. Der Algorithmus nutzt als Invariante, dass oberhalb der Sweep Line Eckpunkte und Halbkanten $O(S_1, S_2)$ korrekt berechnet sind. Ferner kopieren wir alle Halbkanten und Eckpunkte von S_1 und S_2 in die Datenstruktur für $O(S_1, S_2)$.



2.3. Überlagerung planarer Unterteilungen

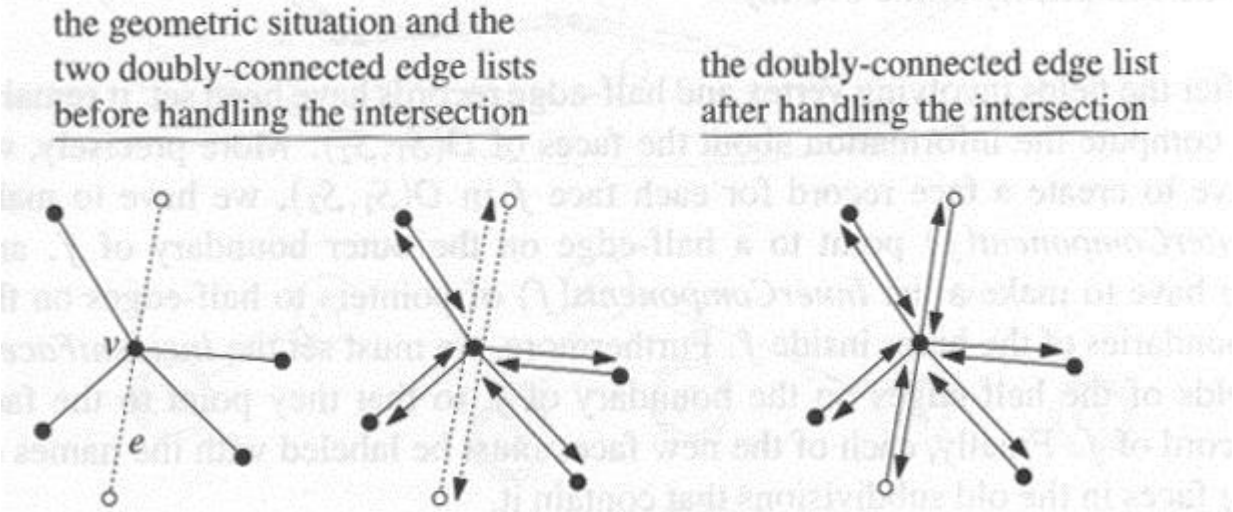
Den Schnitt einer Kante e aus S_1 mit einem Eckpunkt v aus S_2 betrachten wir näher.



An edge of one subdivision passing through a vertex of the other

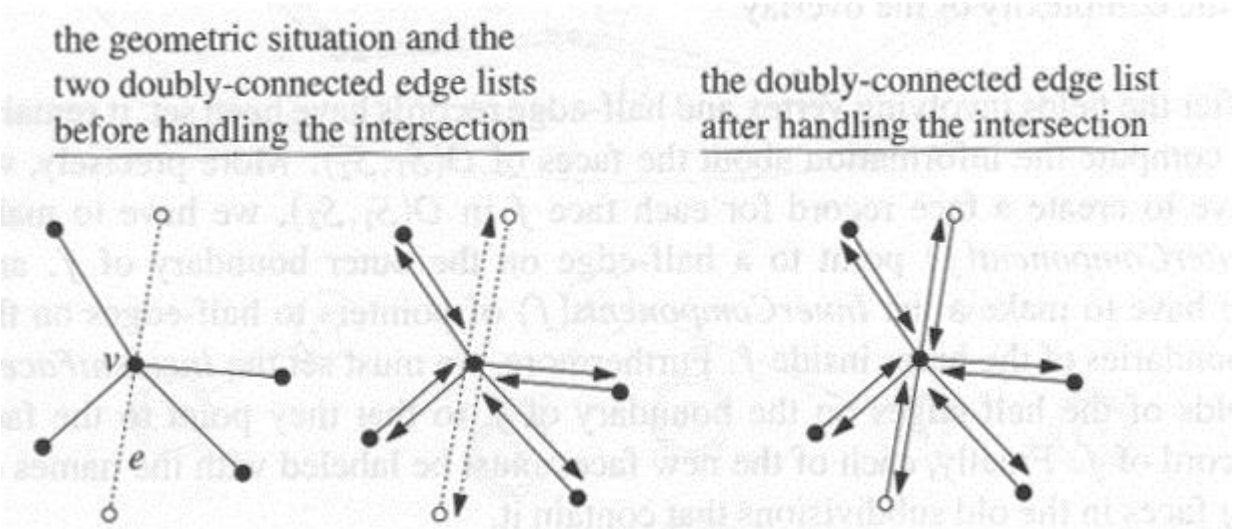
2.3. Überlagerung planarer Unterteilungen

Die Kante *e* ist durch zwei Kanten *e′* und *e″* zu ersetzen, also entstehen aus dem Halbkantenpaar für *e* zwei Paare. Wir erzeugen zwei neue Halbkanteneinträge, die beide *v* als Ursprung haben, während die beiden alten Halbkanten ihren Ursprung behalten. Dann setzen wir die *Twin*-Zeiger so, dass je eine neue und eine alte Halbkante ein Paar bilden und *e′* bzw. *e″* repräsentieren. Wir müssen nun noch *Prev* und *Next* Zeiger setzen bzw. aktualisieren. An den Endpunkten von *e* bekommen die neuen Halbkanten stets den Nachfolger der alten Halbkante, die nicht ihr Zwilling ist. Die so referenzierten Halbkanten erhalten dann die neuen Halbkanten als Vorgänger.



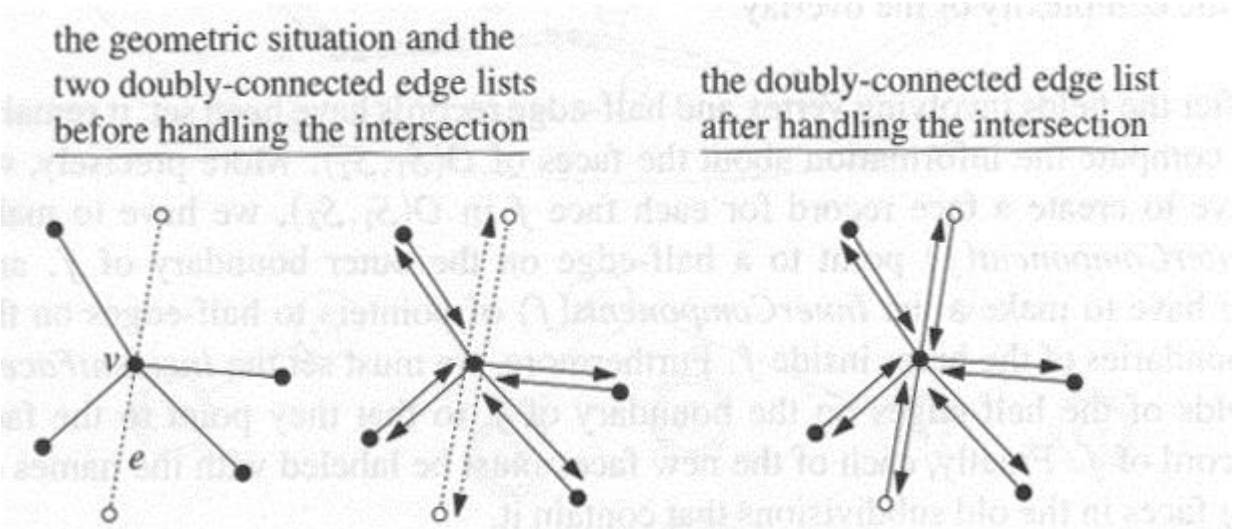
2.3. Überlagerung planarer Unterteilungen

Um die Situation bei *v* zu klären, müssen wir berechnen, an welcher Stelle in der zyklischen Ordnung um *v* die neuen Halbkantenpaare angenommen werden müssen. (Dies kann über Winkelberechnungen geschehen und gehört zu unseren kritisch zu betrachtenden Basisoperationen!)



2.3. Überlagerung planarer Unterteilungen

Insgesamt sind hier acht Zeiger anzupassen, je einer für die beiden alten Halbkanten, je einer für die beiden neuen Halbkanten und vier für die referenzierten Kanten aus S_2. (Wenn alle Kanten in S_2, die mit v inzident sind, auf einer Seite von e liegen, sind es weniger Zeiger, aber darauf muss man nicht achten, wenn man einfach die richtigen Halbkanten ermittelt, egal ob aus S_2, aus S_1 oder den beiden neuen Halbkanten)



2.3. Überlagerung planarer Unterteilungen

Die anderen Fälle Schnitt zweier Kanten und Übereinanderlegen zweier Eckpunkte sind ähnlich, aber einfacher zu bearbeiten.

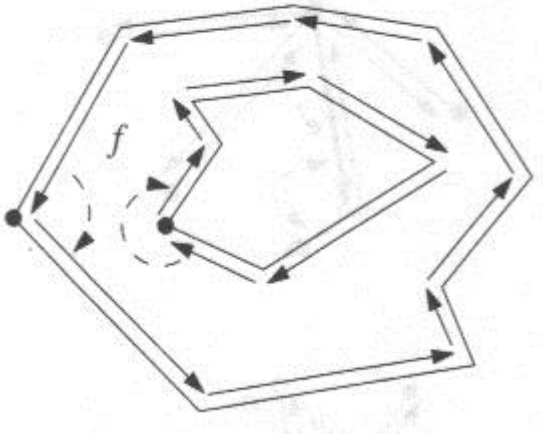
Wir betrachten noch die Komplexität der Operationen. Alle Operationen sind von konstanter Zeit außer der zyklischen Suche nach der nächsten Halbkante. Diese hängt vom Grad des Eckpunktes (also der Anzahl inzidenter Kanten) ab. In den beiden anderen Fällen gilt dies ebenfalls.

Bemerkung:
Die Berechnung der Eckpunktfelder und Kantenfelder der Überlagerung $O(S_1, S_2)$ planarer Unterteilungen S_1, S_2 hat die gleiche Komplexität wie der Schnitt von Liniensegmenten, also $O(n \log n + k \log n)$, wobei n die Summe der Eckpunkte, Kanten und Facetten in S_1 und S_2 und k die Anzahl der Schnitte von Eckpunkten und Kanten aus S_1 mit denen aus S_2 ist.

2.3. Überlagerung planarer Unterteilungen

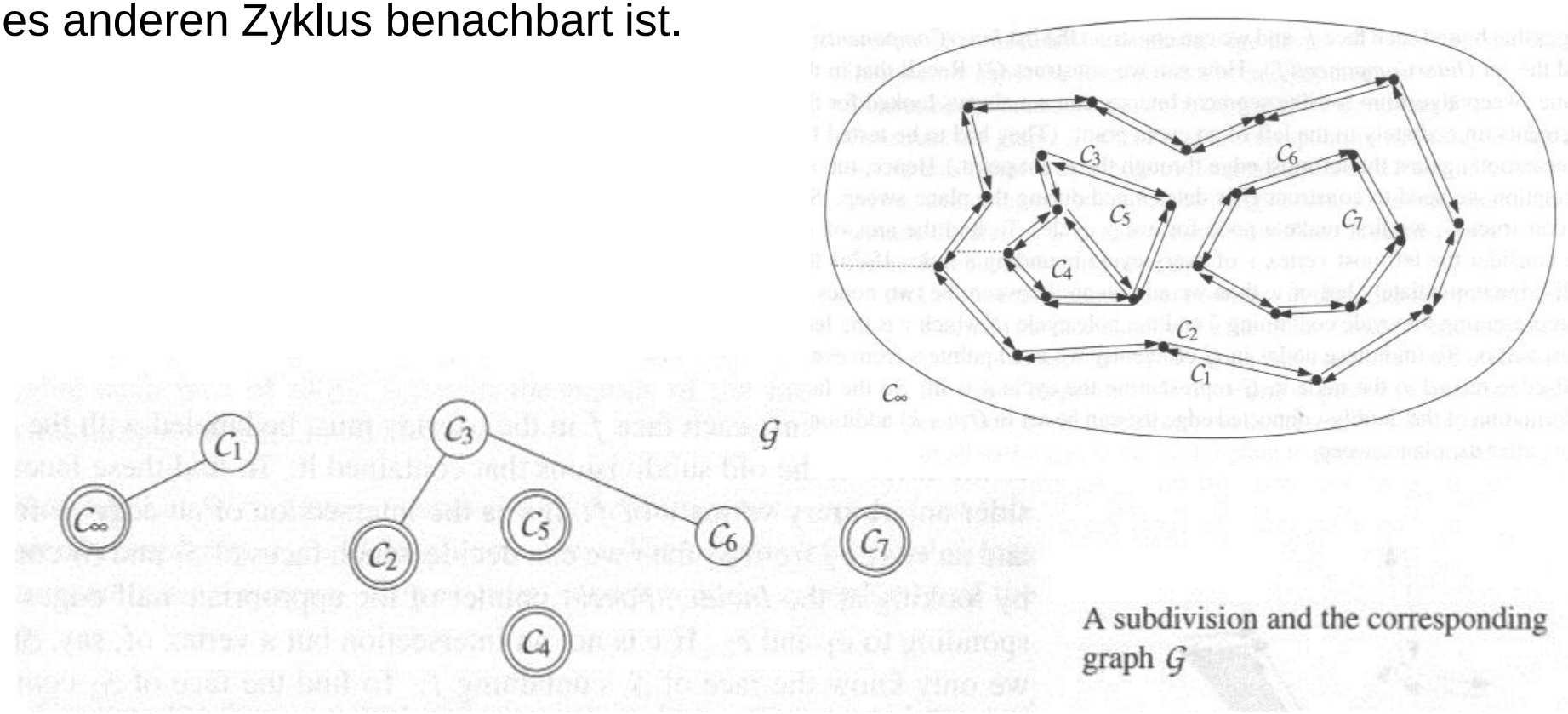
Wir müssen noch die Facetten ermitteln:

1) Wir ermitteln aus den Halbkanten alle Zyklen, welche die Facetten nach innen und außen begrenzen. Ferner definieren wir einen imaginären äußeren Zyklus um die unbegrenzte Facette.
2) In jedem Zyklus suchen wir den am weitesten links und bei gleicher x-Koordinate den am weitesten unten liegenden Eckpunkt v. Wenn die Halbkanten einen Winkel kleiner 180° aufspannen, ist es ein äußerer Zyklus, bei einem Winkel größer 180° ein innerer Zyklus.
3) Jeder äußere Zyklus definiert genau eine Facette.



2.3. Überlagerung planarer Unterteilungen

Um die inneren Zyklen den Facetten zuzuordnen, konstruieren wir einen Graphen *G*, der für jeden Zyklus (äußeren und inneren) einen Knoten enthält. In dem Graphen gibt es eine Kante zwischen zwei Kanten, wenn der am weitesten links liegende Punkt *v* des einen Zyklus entlang der *x*-Achse einer Halbkante des anderen Zyklus benachbart ist.



A subdivision and the corresponding graph *G*

2.3. Überlagerung planarer Unterteilungen

Lemma 2.10: Jede Zusammenhangskomponente von G entspricht genau der Menge der Zyklen inzident zu einer Facette.

Beweis: Sei *C* ein Zyklus um ein Loch einer Facette *f*. Dann liegt *f* teilweise links des am meisten links liegenden Eckpunktes von *C*, also muss *C* zu einem anderen Zyklus in *G* eine Kante haben. Da dieser Zyklus *f* begrenzt, gehören Zyklen in der gleichen Zusammenhangskomponente von *G* zur gleichen Facette
Annahme: Es gibt einen inneren Zyklus *G*, der nicht mit dem äußeren Rand einer Facette verbunden ist.
Sei *C*' der Zyklus in der Zusammenhangskomponente von *C* innerhalb *G*, der den am weitesten links liegenden Eckpunkt enthält. Da die zugehörige Facette *f*' teilweise links von *C*' verläuft, gibt es einen Zyklus in *G* der zur gleichen Komponente in *G* gehört und weiter links Eckpunkte hat $\Rightarrow$ Widerspruch zur Definition von *C*'.
Folgerung: In jeder Komponente von *G* gibt es einen äußeren Zyklus.

QED.

2.3. Überlagerung planarer Unterteilungen

Wir können G durch einen Plane Sweep berechnen, da wir ein Event an jedem Eckpunkt haben. Über die inzidenten Kanten können wir die Zyklen finden und in innere und äußere unterteilen. Wir können ferner die links des definierenden Eckpunktes liegende Kante ermitteln, da sie in der Zustandsstruktur *T* genau links des Event Point liegt.

2.3. Überlagerung planarer Unterteilungen

Der letzte Teil unseres Algorithmus bestimmt schließlich, zu welchen Facetten $f_1 \in S_1$ und $f_2 \in S_2$ eine Facette $f \in O(S_1, S_2)$ gehört. Wir betrachten einen beliebigen Eckpunkt $v \in O(S_1, S_2)$. Wenn er durch den Schnitt zweier Kanten $e_1 \in S_1$ und $e_2 \in S_2$ entstanden ist, wissen wir zu welchen Facetten in S_1 und S_2 er gehört. Wenn er von S_1 oder S_2 stammt, müssen wir in einem weiteren Plane Sweep die Facetten zwischen den Kanten in der Zustandsstruktur mit ablegen und an den EventPoints abändern.

2.3. Überlagerung planarer Unterteilungen

Dies liefert den Algorithmus:

Algorithm MAPOVERLAY($\mathcal{S}_1, \mathcal{S}_2$)
Input. Two planar subdivisions $\mathcal{S}_1$ and $\mathcal{S}_2$ stored in doubly-connected edge lists.
Output. The overlay of $\mathcal{S}_1$ and $\mathcal{S}_2$ stored in a doubly-connected edge list $\mathcal{D}$.

1. Copy the doubly-connected edge lists for $\mathcal{S}_1$ and $\mathcal{S}_2$ to a new doubly-connected edge list $\mathcal{D}$.
2. Compute all intersections between edges from $\mathcal{S}_1$ and $\mathcal{S}_2$ with the plane sweep algorithm of Section 2.2. In addition to the actions on $\mathcal{T}$ and Q required at the event points, do the following:
   - Update $\mathcal{D}$ as explained above if the event involves edges of both $\mathcal{S}_1$ and $\mathcal{S}_2$. (This was explained for the case where an edge of $\mathcal{S}_1$ passes through a vertex of $\mathcal{S}_2$.)
   - Store the half-edge immediately to the left of the event point at the vertex in $\mathcal{D}$ representing it.

2.3. Überlagerung planarer Unterteilungen

3. (∗ Now $\mathcal{D}$ is the doubly-connected edge list for $O(S_1, S_2)$, except that the information about the faces has not been computed yet. ∗)
4. Determine the boundary cycles in $O(S_1, S_2)$ by traversing $\mathcal{D}$.
5. Construct the graph G whose nodes correspond to boundary cycles and whose arcs connect each hole cycle to the cycle to the left of its leftmost vertex, and compute its connected components. (The information to determine the arcs of G has been computed in line 2, second item.)
6. **for** each connected component in G
7. **do** Let C be the unique outer boundary cycle in the component and let f denote the face bounded by the cycle. Create a face record for f, set *OuterComponent*(f) to some half-edge of C, and construct the list *InnerComponents*(f) consisting of pointers to one half-edge in each hole cycle in the component. Let the *IncidentFace*() pointers of all half-edges in the cycles point to the face record of f.
8. Label each face of $O(S_1, S_2)$ with the names of the faces of S_1 and S_2 containing it, as explained above.

2.3. Überlagerung planarer Unterteilungen

Theorem 2.11:
Sei S_1 eine ebene Unterteilung mit Komplexität (Summe der Eckpunkte, Kanten und Facetten) n_1, S_2 eine ebene Unterteilung mit Komplexität n_2. Setze $n = n_1 + n_2$. Die Überlagerung von S_1 und S_2 kann in $O(n \log n + k \log n)$ Zeit berechnet werden, wobei k die Komplexität (Anzahl der Schnitte) der Überlagerung ist.

Beweis:
Listenkopieren in Zeile 1 ist linear, also $O(n)$.
Der Plane Sweep kostet $O(n \log n + k \log n)$ Zeit nach Lemma 2.7.
Zeilen 4-7 benötigen lineare Zeit $O(n)$.
Schließlich kann die Facettenherkunft in $O(n \log n + k \log n)$ berechnet werden.
QED.

2.3. Überlagerung planarer Unterteilungen

Die Überlagerung zweier Unterteilungen kann auch Schnitt, Vereinigung und Differenz zweier Polygone P_1 und P_2 ermitteln.
Offensichtlich ist ein Polygon eine sehr einfache Unterteilung der Ebene (genau zwei Facetten). Aus der Überlagerung der beiden Unterteilungen P_1 und P_2 berechnen wir:

- $P_1 \cap P_2$ als die Facetten, die von der berandeten Facette von P_1 und der berandeten Facette von P_2 stammen.
- $P_1 \cup P_2$ als die Facetten, die von einer berandeten Facette von P_1 oder P_2 stammen.
- $P_1 - P_2$ als die Facetten, die von der berandeten Facette von P_1 und der unbeschränkten Facette von P_2 stammen.

2.3. Überlagerung planarer Unterteilungen

Korollar 2.12:
Sei P_1 ein Polygon mit n_1 Ecken und P_2 ein Polygon mit n_2 Ecken, $n = n_1 + n_2$. Dann können $P_1 \cap P_2$, $P_1 \cup P_2$, $P_1 - P_2$ in $O(n \log n + k \log n)$ Zeit berechnet werden, wobei k die Komplexität der Outputs (Eckpunkte + Kanten + Facetten) ist.

2.3. Überlagerung planarer Unterteilungen

Literatur:
Der Algorithmus zum Linienschnitt in $O(n \log n + k \log n)$ ist aus
[J. L. Bentley and T.A. Ottmann: Algorithms for reporting and counting geometric intersections. IEEE Tran. Comput., C-28:643-647, 1979].

Die Reduktion des Arbeitsspeichers von $O(n + k)$ zu $O(n)$ stammt aus
[C. H. Papadimitriou. An algorithm for shortest-path motion in three dimensions. Inform. Process. Lett., 20:259-263, 1985].

2.3. Überlagerung planarer Unterteilungen

Literatur:
Die untere Schranke ist $\Omega(n \log n + k)$.
Chazelle und Edelsbrunner stellten den ersten zeitoptimalen Algorithmus vor *[B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. In Prox. 29th Annu. IEEE Sympos. Found. Comput. Sci., pages 590-600, 1988, B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. J. ACM, 39:1-54, 1992],* benötigen aber $O(n + k)$ Speicher.

Clarkson und Shor haben einen (nicht deterministischen, da mit Zufall arbeitenden) Algorithmus mit $O(n \log n + k)$ Zeitbedarf und $O(n)$ Speicherbedarf angegeben.

Balaban *[I. J. Balaban. An optimal algorithm for finding segment intersections. In Prox. 11th Annu. ACM Sympos. Comput. Geom., pages 211-219, 1995]* hat einen ersten deterministischen Alg. mit $O(n \log n + k)$ Zeit und $O(n)$ Speicherkomplexität angegeben.

2.3. Überlagerung planarer Unterteilungen

Es sei erwähnt, dass Plane Sweep eines der wichtigsten Prinzipien in der Algorithmischen Geometrie ist und erstmals in [A. F. van der Stappen and M. H. Overmars. Motion planning amidst fat obstacles. In Proc. 10th Annu. ACM Sympos. Comput. Geom., pages 31-40, 1994], [J. van Leeuwen and D. Wood. Dynamization of decomosable searching problems, Inform. Process. Lett., 10:51-56, 1980], [J. L. Brown.Vertex based data dependent triangulations. Comput. Aided Geom. Design, 8:239-251, 1991] genutzt wurde.

Das Prinzip kann auch in höheren Dimensionen benutzt werden [J. E. Hopcroft, J. T. Schwartz, and M. Sharir. Planning, Geometry, and Complexity of Robot Motion. Ablex Publishing, Norwood,, NJ, 1987].

Die zweifach verkettete Liste wurde (in einer Variante) von Muller und Preparata [D.E. Muller, F.P. Preparata, Finding the Intersection of two convex polyhedra, Theoret. Cumput. Sci. 7:217-236, 1978] eingeführt.

2.4. Triangulierung von Polygonen

Als drittes Problem haben wir in Kapitel 1 die Triangulierung von Polygonen identifiziert, die etwa bei der Übersetzung eines Museums durch Kameras auftritt.

2.4. Triangulierung von Polygonen

Definition und Theorie:

Definition 2.13: Sei *P* ein einfaches Polygon (d. h., ein Polygon ohne Loch). Eine **Diagonale** ist ein offenes Liniensegment, das zwei Eckpunkte von *P* verbindet und vollständig in *P* liegt.

Definition 2.14: Sei *P* ein einfaches Polygon. Eine Zerlegung von *P* in Dreiecke durch eine maximale Menge schnittfreier Diagonalen heißt **Triangulierung von *P*.**

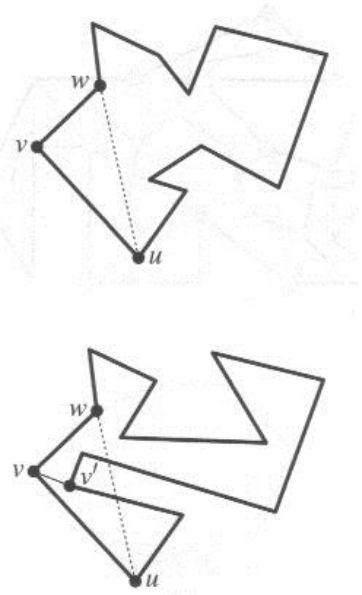
2.4. Triangulierung von Polygonen

Da wir Triangulierungen bestimmen wollen, ist es günstig ihre Existenz zu zeigen.

Theorem 2.15: Jedes einfache Polygon gestattet eine Triangulierung und jede Triangulierung eines einfachen Polygons mit *n* Ecken besteht aus genau *n-2* Dreiecken.

Beweis: Induktion über *n*
n = 3: klar
n > 3: Sei P ein Polygon mit *n* Ecken. Wir müssen die Existenz einer Diagonale zeigen. Sei *v* der am weitesten links liegende Eckpunkt (bei gleicher *x*-Koordinate, die Ecke mit kleinster *y*-Koordinate). Seien *u, w* die Nachbarn von *v* entlang des Randes von *P*. Wenn die Kante uv nicht vom Rand geschnitten wird, sind wir fertig. Sonst sei v' der am weitesten links liegende Punkt im Dreieck u*vw* bzw. auf der Diagonalen uw. Dann liegt die Kante vv‘ komplett in *P*. Da jede Diagonale *P* in zwei kleinere Polygone zerlegt, erhalten wir mit der Induktionsvoraussetzung eine Triangulierung.

2.4. Triangulierung von Polygonen



Sei nun T eine beliebige Triangulierung von P. Wir betrachten eine beliebige Diagonale aus T. Diese zerlegt P in zwei Teilpolygone, die mit m_1 und m_2 Ecken nach Induktionsvoraussetzung $m_1 - 2$ und $m_2 - 2$ Dreiecke enthalten. Da die beiden Teilpolygone jede Ecke außer denen der Diagonale von P genau einmal und die beiden Diagonalenden doppelt enthalten, gilt $m_1 + m_2 = n + 2$. Also gilt für die Anzahl der Dreiecke der Triangulierung $m_1 - 2 + m_2 - 2 = n + 2 - 2 - 2 = n - 2$.

QED.

2.4. Triangulierung von Polygonen

Da jedes Dreieck von einer Kamera überwacht werden kann, reichen $n - 2$ Kameras. Offensichtlich sollte es etwas besser gehen, wenn wir nutzen, dass Eckpunkte von Dreiecken zu mehreren Dreiecken gehören und dort platzierte Kameras alle diese Dreiecke überwachen können. Wir wollen in jedem Dreieck einen Eckpunkt abdecken und markieren daher jeden Eckpunkt eines Dreieckes weiß, grau oder schwarz.

Definition 2.16: Sei P ein einfaches Polygon und T_p eine Triangulierung. Eine 3-Färbung (3-coloring) von (P, T_p) ist eine Markierung der Eckpunkte von P mit drei Farben, so dass Kanten in der Triangulierung T_p stets verschieden farbige Endpunkte haben.

Thoerem 2.17: (Art Gallery Theorem) Für ein einfaches Polygon mit n Ecken reichen stets [n/3] Kameras zur Überwachung und so viele Kameras sind gelegentlich nötig.

2.4. Triangulierung von Polygonen

Wir suchen nach Polygonen, die sich rasch triangulieren lassen und in die sich Polygone (relativ) rasch zerlegen lassen. Allgemeine Polygone sind nämlich nicht immer leicht zu triangulieren.

Die Lösung bieten monotone Polygone.

Definition 2.18: Ein Polygon P heißt **monoton bzgl. der Geraden** l, wenn für jede zu l senkrechte Gerade l' der Schnitt $P \cap l'$ zusammenhängend ist (also aus keinem Punkt, aus einem Punkt oder aus einem Liniensegment besteht). Ein bzgl. der y-Achse monotones Polygon heißt $\boldsymbol{y}$**-monoton**.

Wir wollen P zuerst in y-monotone Teile zerlegen und dann diese triangulieren. Um dies zu tun, beginnen wir am obersten Punkt von P und laufen abwärts zum tiefsten Punkt. Ein Eckpunkt, an dem die Kanten von abwärts nach aufwärts (oder umgekehrt) wechseln, heißt Umkehrpunkt (turn vertex). Diese Eckpunkte müssen durch Zerlegungen mit Diagonalen entfernt werden.

2.4. Triangulierung von Polygonen

Wir müssen dazu fünf Typen von Eckpunkten in P unterscheiden.

Definition 2.19: Für zwei Punkte *p, q* der Ebene gelte:

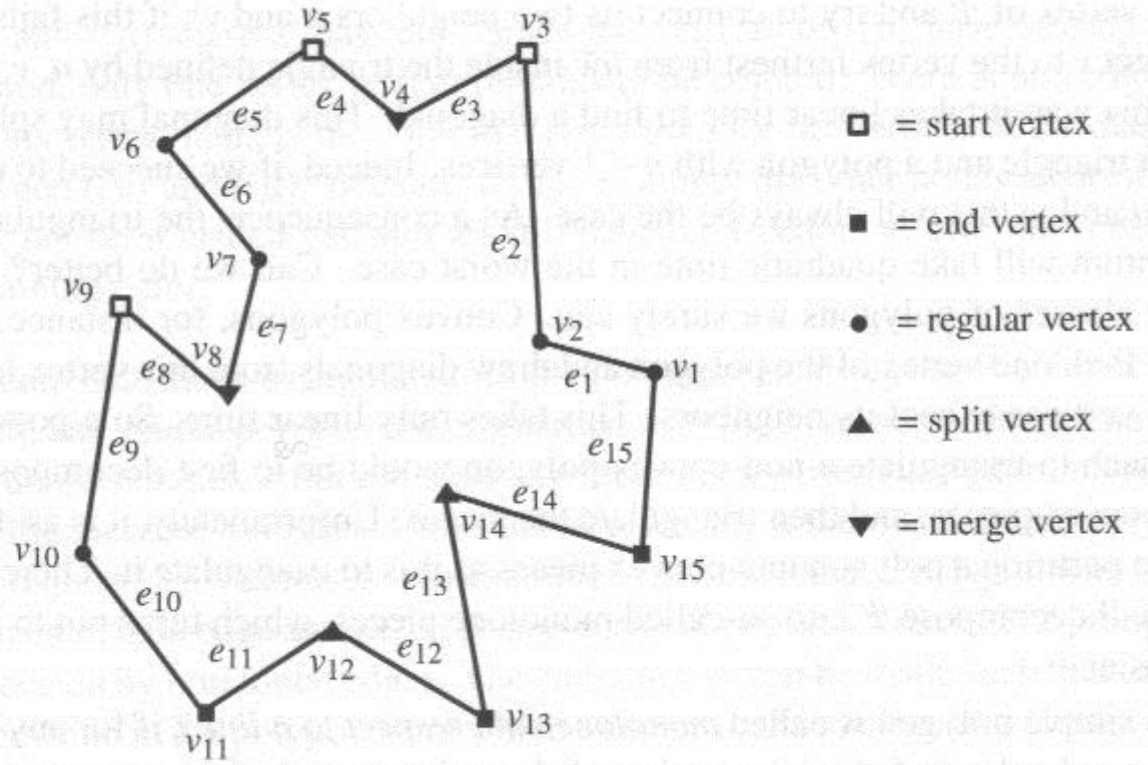
p ist **unterhalb von** q, falls $p_y < q_y$ oder $p_y = q_y$ und $p_x > q_x$

p ist **oberhalb von** q, falls $p_y > q_y$ oder $p_y = q_y$ und $p_x < q_x$

Definition 2.20: Ein Eckpunkt v eines Polygons *p* heißt

- **Startpunkt** (start vertex), wenn beide Nachbarn unterhalb von v liegen und der innere Winkel bei v kleiner als π ist.
- **Trennpunkt** (split vertex), wenn beide Nachbarn unterhalb von v liegen und der innere Winkel bei v größer als π ist.
- **Endpunkt** (end vertex), wenn beide Nachbarn oberhalb von v liegen und der innere Winkel bei v kleiner als π ist.
- **Schmelzpunkt** (merge vertex), wenn beide Nachbarn oberhalb von v liegen und der innere Winkel bei v größer als π ist.
- **Regulärer Punkt** (regular vertex) in allen übrigen Fällen.

2.4. Triangulierung von Polygonen



Five types of vertices

2.4. Triangulierung von Polygonen

Lemma 2.21: Ein Polygon ist y-monoton, wenn es keine Trennpunkte und Schmelzpunkte hat.

Beweis: Sei P nicht y-monoton. Dann müssen wir einen Trennpunkt oder Schmelzpunkt finden.
Da P nicht y-monoton ist, gibt es eine Gerade l, die P in mehreren Komponenten schneidet. l lässt sich so wählen, dass die am meisten linke Schnittkomponente ein Liniensegment (und kein Punkt) ist. Sei *p* der linke und *q* der rechte Schnitt von l mit P in diesem linken Segment. Wir laufen bei *q* nach oben los und folgen dem Rand bis wir wieder auf l treffen. Wenn dieser Punkt *r* rechts von *q* liegt, ist der höchste Punkt zwischen *q* und *r* ein Trennpunkt $\Rightarrow$ fertig.
Falls dagegen *p* = *r* gilt, laufen wir von *q* aus nach unten bis wir wieder l treffen, diesmal in *r*'. Diesmal muss *r*' != *p* gelten, da der Rand von P l mindestens dreimal trifft. Der tiefste Punkt zwischen *q* und *r*' ist nun ein Schmelzpunkt.

QED.

2.4. Triangulierung von Polygonen

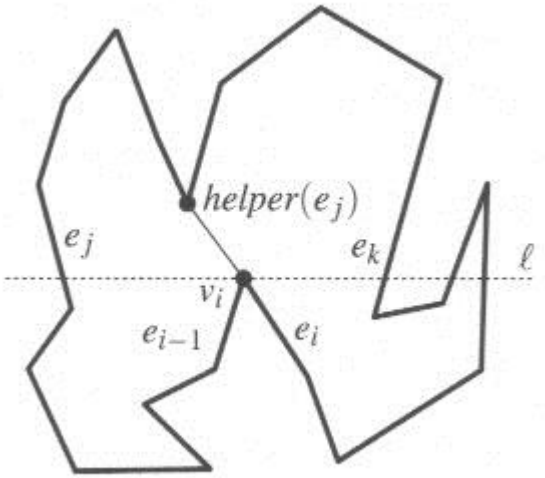
Die Unterteilung in y-monotone Teile erreicht man durch Einfügen passender Diagonalen an den Trenn- und Schmelzpunkten. Wir nummerieren die Ecken $v_1,...,v_n$ des Polygons gegen den Uhrzeigersinn und bezeichnen mit $e_i = \overline{v_i v_{i+1}}$ die Kanten.

Wieder gelangt ein Plane Sweep zum Einsatz, wobei die Kanten in der Zustandsstruktur gehalten werden und die Eckpunkte die Eventpunkte sind. Ferner brauchen wir noch die Eckpunkte für die Diagonalen.

2.4. Triangulierung von Polygonen

Dazu definieren wir:

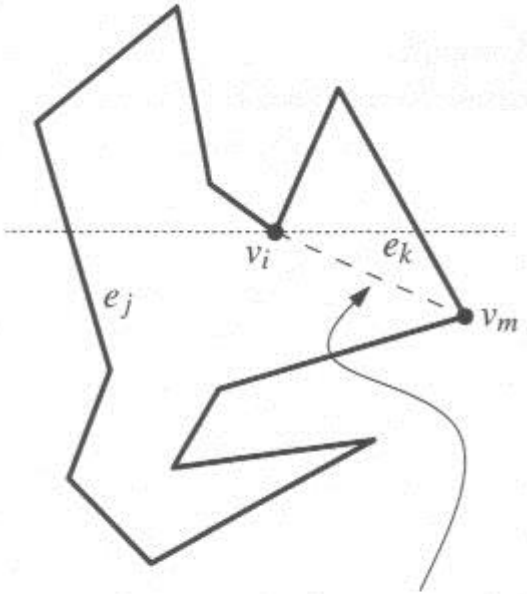
Definition 2.22: Sei v_i ein Trennpunkt und e_j die Kante links von v_i entlang der Sweep Line. Dann ist der Hilfspunkt helper(e_j) der niedrigste Eckpunkt oberhalb der Sweep Line, so dass die x-parallele Strecke e_j zu helper(e_j) ganz in P liegt.



Die Hilfpunkte helper(e_j) speichern wir stets mit den Kanten und aktualisieren an den Eventpunkten. Der Trennpunkt wird dann mit helper(e_j) verbunden und verschwindet.

2.4. Triangulierung von Polygonen

Um die Schmelzpunkte zu entfernen, drehen wir die Betrachtung einfach um. Der Schmelzpunkt wird helper(e_j) einer Kante e_j. Wir verbinden ihn mit dem nächsten Hilfspunkt der Kante e_j oder dem Endpunkt der Kante e_j, sofern kein weiterer Hilfspunkt auftaucht.



diagonal will be added when the sweep line reaches v_m

2.4. Triangulierung von Polygonen

Nun können wir den Algorithmus zur Umwandlung in y-monotone Polygone formulieren. Wir nehmen dazu an, dass das Polygon P wieder als doppelt verknüpfte Liste gegeben ist, da dies das Trennen erleichtert. Ist es nicht so gegeben, müssen wir P erst in diese Form bringen.

Algorithm MAKEMONOTONE($\mathcal{P}$)
Input. A simple polygon $\mathcal{P}$ stored in a doubly-connected edge list $\mathcal{D}$.
Output. A partitioning of $\mathcal{P}$ into monotone subpolygons, stored in $\mathcal{D}$.

1. Construct a priority queue Q on the vertices of $\mathcal{P}$, using their y-coordinates as priority. If two points have the same y-coordinate, the one with smaller x-coordinate has higher priority.
2. Initialize an empty binary search tree $\mathcal{T}$.
3. **while** Q is not empty
4. **do** Remove the vertex v_i with the highest priority from Q.
5. Call the appropriate procedure to handle the vertex, depending on its type.

2.4. Triangulierung von Polygonen

Im Algorithmus sind die einzelnen Fälle Startpunkt, Endpunkt, Trennpunkt, Schmelzpunkt und regulärer Punkt zu trennen.

HANDLESTARTVERTEX(v_i)
1. Insert e_i in $\mathcal{T}$ and set $helper(e_i)$ to v_i.

At the start vertex v_5 in the example figure, for instance, we insert e_5 into the tree $\mathcal{T}$.

HANDLEENDVERTEX(v_i)
1. **if** $helper(e_{i-1})$ is a merge vertex
2. **then** Insert the diagonal connecting v_i to $helper(e_{i-1})$ in $\mathcal{D}$.
3. Delete e_{i-1} from $\mathcal{T}$.

HANDLEMERGEVERTEX(v_i)
1. **if** $helper(e_{i-1})$ is a merge vertex
2. **then** Insert the diagonal connecting v_i to $helper(e_{i-1})$ in $\mathcal{D}$.
3. Delete e_{i-1} from $\mathcal{T}$.
4. Search in $\mathcal{T}$ to find the edge e_j directly left of v_i.
5. **if** $helper(e_j)$ is a merge vertex
6. **then** Insert the diagonal connecting v_i to $helper(e_j)$ in $\mathcal{D}$.
7. $helper(e_j) \leftarrow v_i$

2.4. Triangulierung von Polygonen

HANDLESPLITVERTEX(v_i)
1. Search in $\mathcal{T}$ to find the edge e_j directly left of v_i.
2. Insert the diagonal connecting v_i to $helper(e_j)$ in $\mathcal{D}$.
3. $helper(e_j) \leftarrow v_i$
4. Insert e_i in $\mathcal{T}$ and set $helper(e_i)$ to v_i.

HANDLEREGULARVERTEX(v_i)
1. **if** the interior of $\mathcal{P}$ lies to the right of v_i
2. **then if** $helper(e_{i-1})$ is a merge vertex
3. **then** Insert the diagonal connecting v_i to $helper(e_{i-1})$ in $\mathcal{D}$.
4. Delete e_{i-1} from $\mathcal{T}$.
5. Insert e_i in $\mathcal{T}$ and set $helper(e_i)$ to v_i.
6. **else** Search in $\mathcal{T}$ to find the edge e_j directly left of v_i.
7. **if** $helper(e_j)$ is a merge vertex
8. **then** Insert the diagonal connecting v_i to $helper(e_j)$ in $\mathcal{D}$.
9. $helper(e_j) \leftarrow v_i$

2.4. Triangulierung von Polygonen

Nun lässt sich die Korrektheit beweisen.

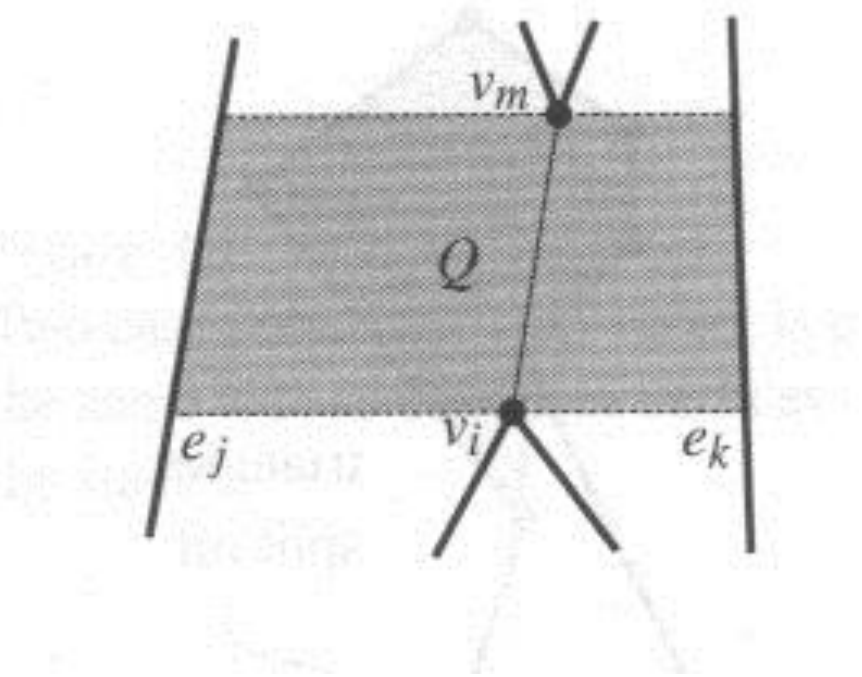
Lemma 2.23: MAKEMONOTONE zerlegt ein Polygon P in y-monotone Teilpolygone durch Hinzufügen schnittpunktfreier Diagonalen.

Beweis:
Offensichtlich enthalten die verbleibenden Stücke keine Trenn- und Schmelzpunkte mehr, da die Diagonalen am Trennpunkt nach oben und am Schmelzpunkt nach unten verlaufen. Also sind die Teile y-monoton nach Lemma 2.21.
Wir zeigen nun, dass eine neue Diagonale in HANDLESPLITVERTEX keine Linien in unserer gegenwärtigen Struktur schneidet. Die Korrektheit der übrigen Funktionen lässt sich analog beweisen. Ferner nehmen wir an, dass alle Eckpunkte verschiedene y-Koordinate haben. Sei $\overline{v_m v_i}$ ein neues Segment in HANDLESPLITVERTEX. Es sei e_j die Kante links von v_i und e_k die Kante rechts von v_i. Es ist v_m = helper(e_j). Wir betrachten den Bereich Q zwischen e_j und e_k, den wir parallel zur x-Achse bei v_i und v_m abschneiden.

2.4. Triangulierung von Polygonen

Die neue Diagonale verläuft in diesem Bereich. Ferner liegen dort keine anderen Punkte oder Ecken, da v_m sonst nicht Hilfspunkt wäre, also sind e_j und e_k dort stets benachbart. Daher kann die Diagonale keine anderen Kanten oder Ecken schneiden.

QED



2.4. Triangulierung von Polygonen

Theorem 2.24: Ein einfaches Polygon mit n Ecken kann in $O(n \log n)$ Zeit mit $O(n)$ Speicher in y-monotone Teile zerlegt werden.

Beweis:
Der Aufbau von Q kann in $O(n \log n)$ erfolgen, T wird in $O(1)$ leer initialisiert. Die Schritte in den Unterroutinen bestehen aus maximal einer Operation auf Q, einer Suche, einem Einfügen und einem Löschen in T. Dies erfolgt in $O(\log n)$. Da es insgesamt n Ereignisse gibt, folgt die Aussage.

QED

2.4. Triangulierung von Polygonen

Triangulieren eines monotonen Polygons

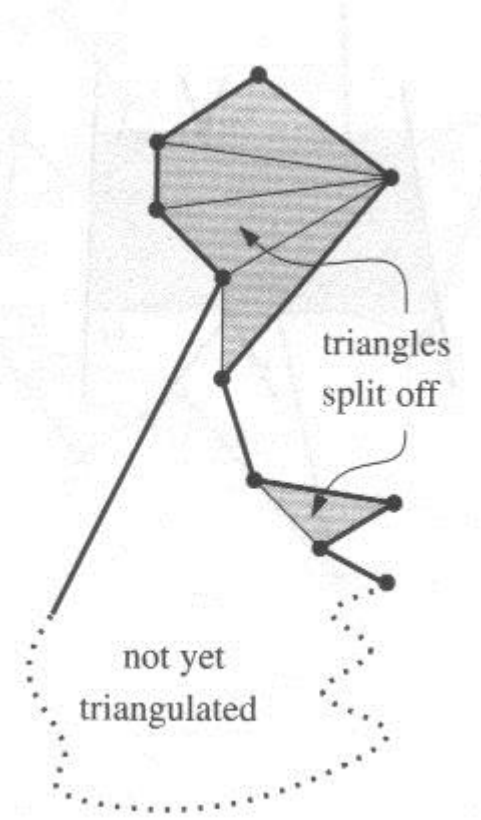
Die Idee liegt natürlich in der schönen Eigenschaft, dass wir beim Triangulieren jetzt den beiden Rändern (links und rechts) entlang laufen können und uns stets nach unten bewegen. Dadurch können wir einfach die geeigneten Eckpunkte für die Diagonalen finden.

Es ergibt sich ein schöner $O(n)$ Greedy Algorithmus.

2.4. Triangulierung von Polygonen

Die verbleibende kleine Schwierigkeit gegenüber konvexen Polygonen ist die Existenz von Eckpunkten mit einem Innenwinkel > 180°.

Dieses lösen wir durch einen Stapel (stack) S. Der Algorithmus läuft die Punkte in absteigender y-Koordinatenrichtung ab. Der noch nicht triangulierte Teil des Polygons oberhalb der y-Koordinate besteht dabei stets auf einer Seite aus einer Kante, die beim untersten Element des Stapels beginnt und auf der anderen Seite aus den Punkten im Stapel, die jeweils einen Innenwinkel > 180° haben, wobei die Reihenfolge im Stapel genau von unten nach oben ist.

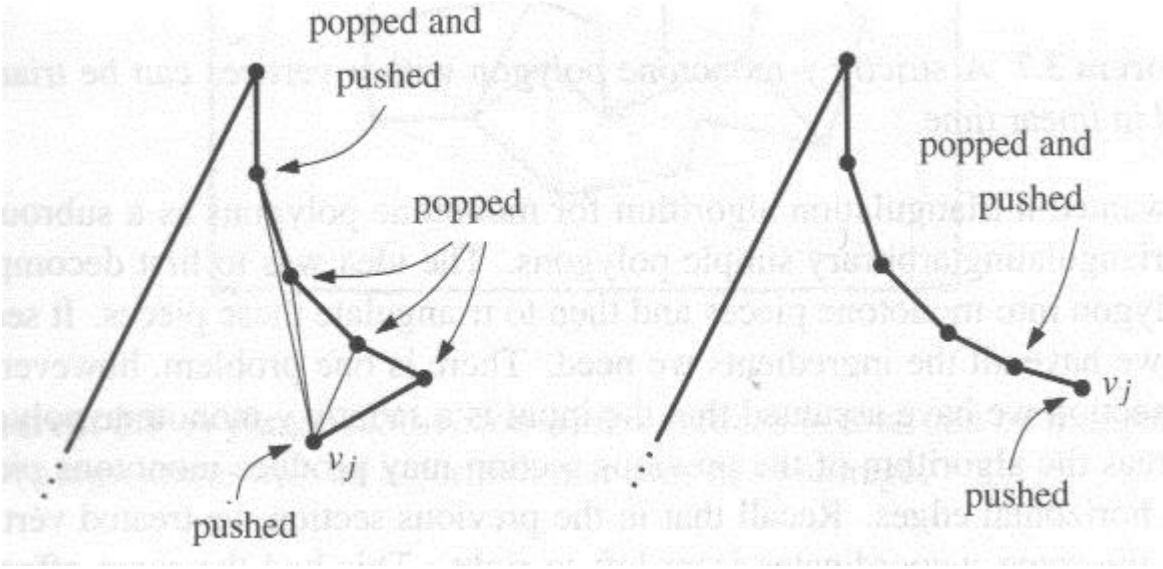


2.4. Triangulierung von Polygonen

Wenn ein neuer Punkt v_j bearbeitet wird, gibt es zwei Fälle:

1) Er liegt auf der gleichen Seite wie die Kette von Punkten im Stapel.
2) Er ist der Endpunkt der Kante zum untersten Punkt im Stapel.

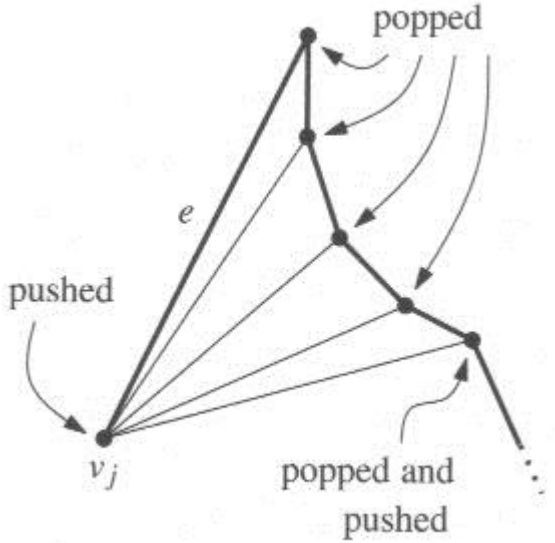
Im ersten Fall holen wir den obersten Punkt vom Stapel. Dieser ist durch eine Kante mit v_j verbunden. Also holen wir den nächsten Punkt vom Stapel und versuchen ein Dreieck zu bilden. Wenn dies gelingt, fahren wir fort. Wenn das Dreieck außerhalb des Polygons liegt, kommen erst die beiden anderen Punkte und dann v_j oben auf den Stapel.



2.4. Triangulierung von Polygonen

Der zweite Fall ist einfach, denn wir können Diagonalen zu allen Punkten im Stapel bis auf den unteren bilden, der ja bereits mit v_j verbunden ist. Wir tun dies und legen anschließend den zuvor oben auf dem Stapel gelegenen Punkt und dann v_j in den Stapel.

Das ist alles!



2.4. Triangulierung von Polygonen

Algorithm TRIANGULATEMONOTONEPOLYGON($\mathcal{P}$)
Input. A strictly y-monotone polygon $\mathcal{P}$ stored in a doubly-connected edge list $\mathcal{D}$.
Output. A triangulation of $\mathcal{P}$ stored in the doubly-connected edge list $\mathcal{D}$.

1. Merge the vertices on the left chain and the vertices on the right chain of $\mathcal{P}$ into one sequence, sorted on decreasing y-coordinate. If two vertices have the same y-coordinate, then the leftmost one comes first. Let $u_1, \ldots, u_n$ denote the sorted sequence.
2. Initialize an empty stack $\mathcal{S}$, and push u_1 and u_2 onto it.
3. **for** $j \leftarrow 3$ **to** $n-1$
4. **do if** u_j and the vertex on top of $\mathcal{S}$ are on different chains
5. **then** Pop all vertices from $\mathcal{S}$.
6. Insert into $\mathcal{D}$ a diagonal from u_j to each popped vertex, except the last one.
7. Push u_{j-1} and u_j onto $\mathcal{S}$.
8. **else** Pop one vertex from $\mathcal{S}$.
9. Pop the other vertices from $\mathcal{S}$ as long as the diagonals from u_j to them are inside $\mathcal{P}$. Insert these diagonals into $\mathcal{D}$. Push the last vertex that has been popped back onto $\mathcal{S}$.
10. Push u_j onto $\mathcal{S}$.
11. Add diagonals from u_n to all stack vertices except the first and the last one.

2.4. Triangulierung von Polygonen

Offensichtlich gilt

Theorem 2.25: Ein y-monotones Polygon mit n Ecken kann in linearer Zeit trianguliert werden.

(Dabei sind Punkte mit gleicher y-Koordinate von links nach rechts zu behandeln.)

Ferner folgt zusammen mit der Zerlegung in y-monotone Teile:

Theorem 2.26: Ein einfaches Polygon mit n Ecken kann in $O(n \log n)$ Zeit und $O(n)$ Speicher trianguliert werden.

2.4. Triangulierung von Polygonen

Wichtig ist, dass unsere Algorithmen genauso gut auf einer planaren Unterteilung oder Polygonen mit Löchern arbeiten können. Es gilt:

Theorem 2.27: Eine planare Unterteilung mit n Ecken kann in $O(n \log n)$ Zeit mit $O(n)$ Speicher trianguliert werden.

2.4. Triangulierung von Polygonen

Literatur

Mit dem Art Gallery Theorem [V. Chvátal. A combinatorial theorem in plane geometry. J. Combin. Theory Ser. B, 18:39-41, 1975] beantwortete Chvátal eine Frage von Victor Klee aus dem Jahr 1973.

Der einfache Beweis in diesem Abschnitt stammt von Fisk [S. Fisk. A short proof of Chvátal's watchman theorem. J. Combin. Theory Ser. B, 24:374, 1978].

Das Finden der minimalen Anzahl von Wächtern ist dagegen NP-hart [A. Aggarwal. The art gallery problem: Its variations, applications, and algorithmic aspects. Ph.D. thesis, Johns Hopkins Univ., Baltimore, MD, 1984].

Die Triangulierung monotoner Polygone in linearer Zeit stammt von Garex et al. [M. R. Garex, D. S. Johnson, F. P. Preparata, and R. E. Tarjan. Triangulating a simple polygon. Inform. Process. Lett., 7:175-179, 1978], während die Plane Sweep Lösung zur Zerlegung eines einfachen Polygons in monotone Teile von Lee und Preparata [D. T. Lee and F. P. Preparata. Location of a point in a planar subdivision and its applications. SIAM J. Comput., 6:594-606, 1977] erstmals beschrieben wurde.

2.4. Triangulierung von Polygonen

Ein interessantes Resultat ist, dass eine beliebige Unterteilung $O(n \log n)$ Schritte für die Triangulierung benötigt, während ein einfaches Polygon von Tarjan und van Wyk in $O(n \log \log n)$ trianguliert wurde [R. E. Tarjan and C. J. Van Wyk. An *O*(*n* log log *n*)-time algorithm for traingulating a simple poygon. SIAM J. Comput., 17:143-178, 1988. Erratum in 171988), 106].

Seidel [R. Seidel. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. Comput. Geom. Theory Appl., 1:51-64, 1991] hat sogar einen $O(n \log^* n)$ Algorithmus angegeben, wobei $\log^* n$ die beliebig o*ft* wiederholte Anwendung des Logarithmus ist.

1990 hat Chazelle [B. Chazelle. Triangulating a simple polygon in linear time. In Proc. 31st Annu. IEEE Sympos. Found. Comput. Sci., pp. 220-230, 1990, B. Chazelle. Triangulating a simple polygon in linear time. Discrete Comput. Geom., 6:485-524, 1991] schließlich einen **sehr komplizierten, deterministischen linearen Algorithmus** angegeben.

2.4. Triangulierung von Polygonen

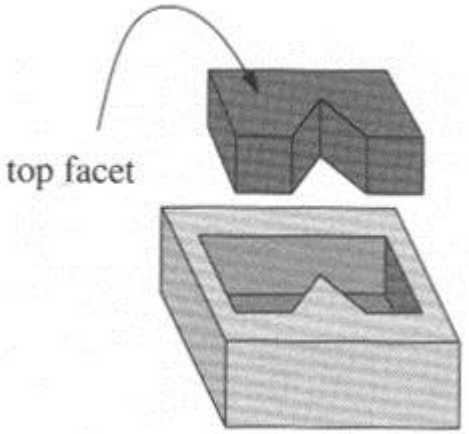
Das 3D-Problem der Zerlegung eines Polytops in nicht überlappende Tetraeder ist wesentlich schwieriger. Chazelle und Palios [B. Chazelle and L. Palios. Triangulating a non-convex polytope. Discrete Comput. Geom., 5:505-526, 1990] fügen $O(n + r^2)$ zusätzliche Punkte ein und benötigen $O(nr + r^2 \log r)$ Zeit. Dabei ist r die Anzahl der konkaven Kanten.

Die Frage, ob zusätzliche Punkte nötig sind, ist sogar NP-vollständig [J. Ruppert and R. Seidel. On the difficulty of triangulating three-dimensional non-convex polyhedra. Discrete Comput. Geom., 7:227-253, 1992].

2.5. Lineare Programme

Gussformen

Wir wollen entscheiden, ob ein polyhedrisches Objekt durch Gießen in eine Form hergestellt werden kann, wobei das Objekt ohne Zerstörung der Form aus dieser wieder entfernt werden soll. Wir gehen davon aus, dass die Oberfläche der Form eben und parallel zur xy-Ebene ausgerichtet ist. Die obere frei liegende Facette des Polyeders nennen wir **top facet** und die übrigen Facetten **gewöhnlich** (ordinary facets).



2.5. Lineare Programme

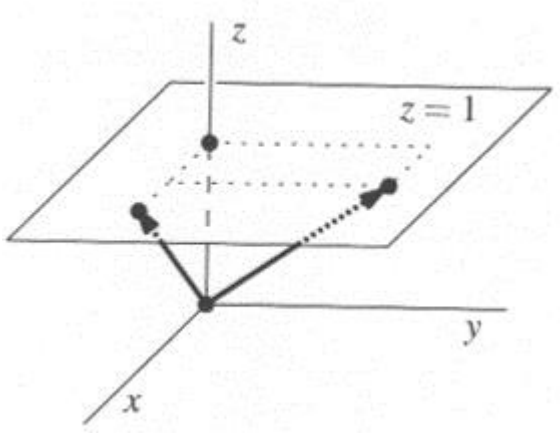
Zu den gewöhnlichen Facetten f gibt es stets eine Facette $\hat{f}$ der Form. Wir suchen eine Richtung d, in die wir das Objekt aus der Form herausziehen können. Diese Richtung d muss eine positive *z*-Komponente haben und mit jeder nach außen gerichteten Normalen $n(f)$ der Facetten des Objektes einen Winkel von mindestens 90° bilden.

Lemma 2.28: Ein Polyeder *P* kann durch Translation in Richtung d aus einer Form entfernt werden *gdw* d einen Winkel von mindestens 90° mit jeder nach außen gerichteten Normale der Facetten des Objektes bildet.

Beweis: "$A \Rightarrow B$": Wenn d mit einer Normale $n(f)$ einen Winkel kleiner als 90° bildet, gibt es einen Punkt q im Inneren der Facette f, der mit der Form kollidiert.
"$B \Rightarrow A$" = "$\neg A \Rightarrow \neg B$". *P* kollidiere mit der Form beim Herausziehen in Richtung d am Punkt p.
Sei $\hat{f}$ die betroffene Facette der Form. Dann haben d und $n(\hat{f})$ einen Winkel < 90°. QED

2.5. Lineare Programme

Um nun eine solche Richtung d zu finden, beschreiben wir alle d mit positiver z-Komponente durch Punkte $d = (d_x, d_y, d_z)^T = (x, y, 1)$ in der Ebene $z = 1$.



Für jede nach außen gerichtete Normale $n(f) = (n_x, n_y, n_z)^T$ ergibt sich nun die Bedingung

$$n_x d_x + n_y d_y + n_z \leq 0$$

2.5. Lineare Programme

Da die obere Facette (top facet) keine Bedingung liefert, müssen wir ein d finden, dass $n-1$ solche Bedingungen erfüllt. Wir werden sehen, dass dies in $O(n)$ erwarteter durchschnittlicher Laufzeit erfolgen kann. Da jede der n Facetten als obere Facette in Frage kommt, ergibt sich

Theorem 2.29: Sei P ein Polyeder mit n Facetten. In $O(n^2)$ Zeit und $O(n)$ Speicher kann entschieden werden, ob P in einer einzigen Form gegossen und aus dieser ohne Zerstörung der Form entfernt werden kann. Wenn P so gegossen werden kann, lässt sich mit gleichem Aufwand die Form und die Richtung für das Entfernen bestimmen.

2.5. Lineare Programme

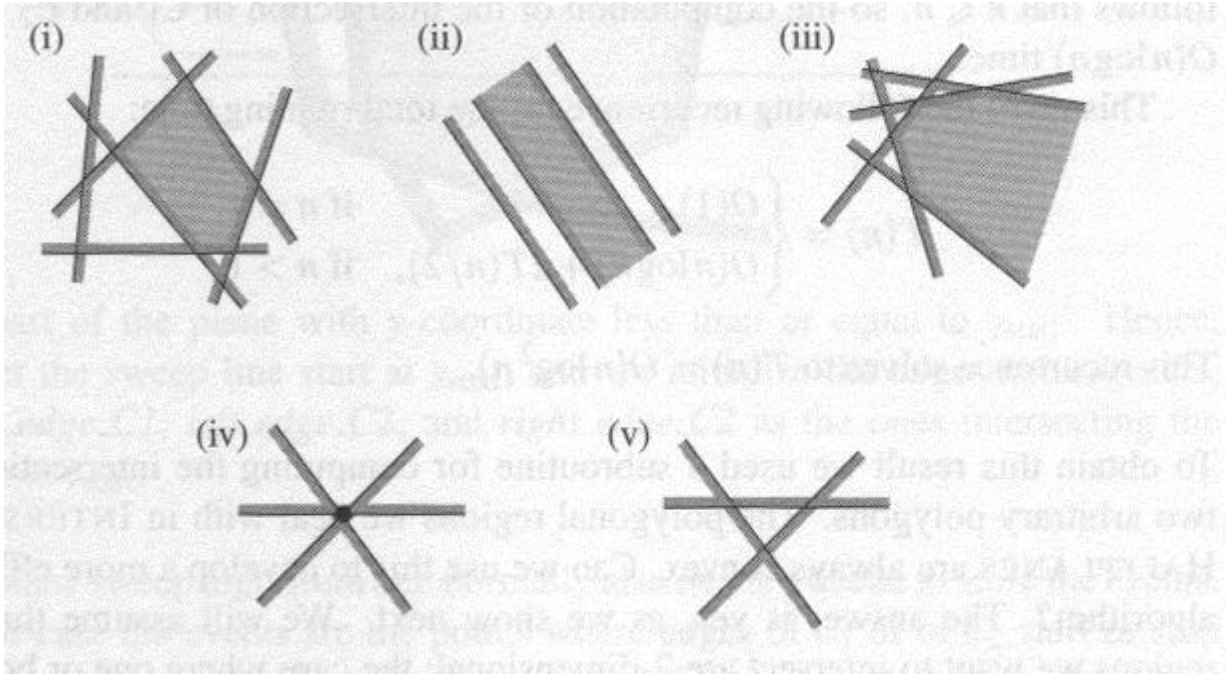
Schnitte von Halbebenen

Um das vorangegangene Problem allgemeiner angehen zu können, betrachten wir eine Menge $H = \{h_1, \ldots, h_n\}$ linearer Restriktionen der Form

$$a_i x + b_i y \leq c_i$$

für Punkte $(x, y) \in \mathbb{R}^2$. Als Ergebnis erhalten wir eine konvexe Teilmenge des $\mathbb{R}^2$, die unbeschränkt sein kann.

2.5. Lineare Programme



Examples of the intersection of half-planes

2.5. Lineare Programme

Ein divide-and-conquer Ansatz führt hier schnell zum Ziel.

Algorithm INTERSECTHALFPLANES(H)
Input. A set H of n half-planes in the plane.
Output. The convex polygonal region $C := \bigcap_{h \in H} h$.

```
1.  if card(H) = 1
2.      then C ← the unique half-plane h ∈ H
3.      else Split H into sets H1 and H2 of size ⌈n/2⌉ and ⌊n/2⌋.
4.           C1 ← IntersectHalfplanes(H1)
5.           C2 ← IntersectHalfplanes(H2)
6.           C ← IntersectConvexRegions(C1, C2)
```

Die Prozedur INTERSECTCONVEXREGIONS kennen wir schon fast. Den Schnitt zweier Polygone in $O(n \log n + k \log n)$ haben wir mit Hilfe eines Plane Sweep bereits behandelt. Ein Einbau unbeschränkter Facetten ist schnell möglich.

2.5. Lineare Programme

Die Komplexitätsanalyse liefert für C_1 und C_2 mit maximal $\frac{n}{2}+1$ Kanten:

Die Überlagerung von C_1 und C_2 kann in $O(n \log n + k \log n)$ berechnet werden, wobei k die Anzahl der Schnittpunkte ist. Da $C_1 \cap C_2$ alle Schnittpunkte von C_1 und C_2 als Eckpunkte enthält, und ferner $C_1 \cap C_2$ als Schnitt von n Halbebenen maximal n Kanten hat, folgt $k \leq n$ also $O(n \log n)$.
Die Formel

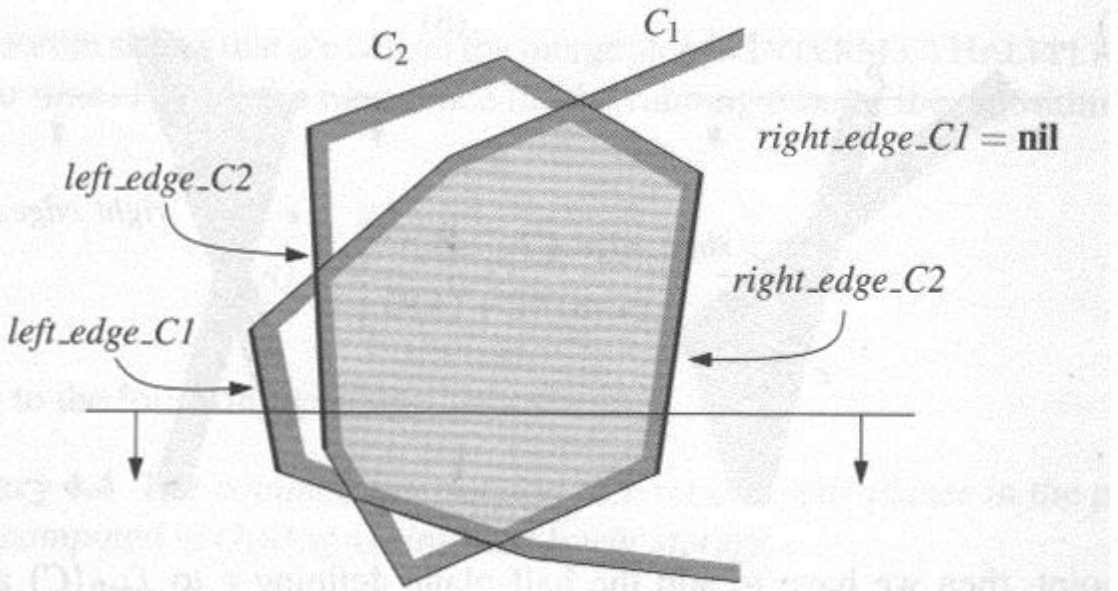
$$T(n) = \begin{cases} O(1) & n = 1 \\ O(n \log n) + 2T\left(\frac{n}{2}\right) & n > 1 \end{cases}$$

liefert $T(n) = O(n \log^2 n)$.

2.5. Lineare Programme

Der Schnitt konvexer Polygone lässt sich aber noch schneller ermitteln. Dazu führen wir direkt einen Plane Sweep durch. Als Zustandsstruktur reichen die vier Zeiger left_edge_C1, left_edge_C2, right_edge_C1, right_edge_C2 aus. Als y_1 definieren wir die größte y-Koordinate der Ecken in C_1 ggf. $y_1 = -\infty$. Analog wird y_2 für C_2 definiert. Wir beginnen bei $y_{\text{start}} = \min(y_1, y_2)$. Die Ereignisse (Events) ergeben sich nun durch Schnitte der aktiven Kanten und deren Endpunkte, so dass wir keine Ereignisschlange brauchen.

2.5. Lineare Programme



The edges maintained by the sweep line algorithm

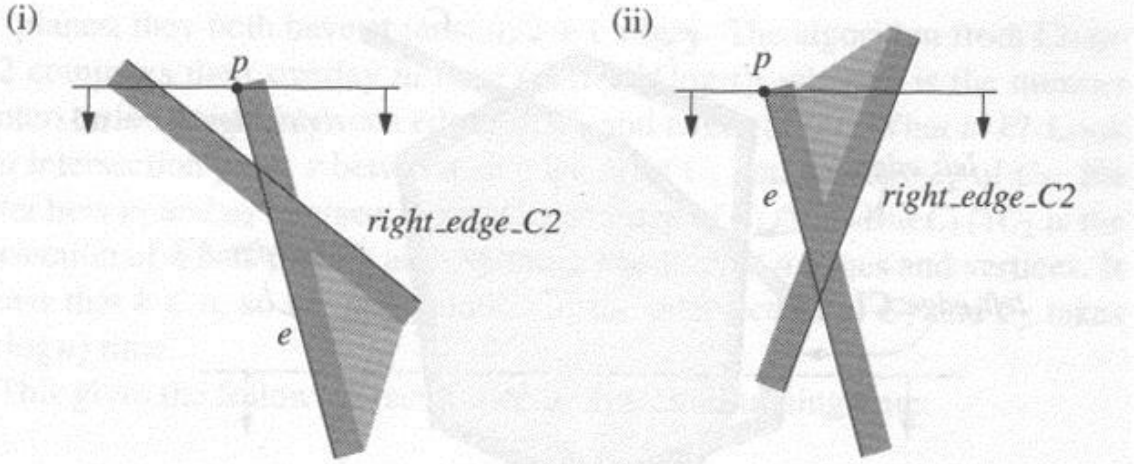
2.5. Lineare Programme

An jedem Ereignis kommt eine neue Kante *e* am Rand hinzu. Wenn *e* ein neues left_edge_C1 ist und p der obere Endpunkt von e ist, suchen wir nach drei möglichen neuen Kanten für C. Eine Kante mit p als oberem Eckpunkt, eine Kante mit $e \cap$ left_edge_C2 als oberem Eckpunkt oder eine Kante mit $e \cap$ right_edge_C2 als oberem Eckpunkt.

Der Algorithmus löst folgende Aufgaben:

2.5. Lineare Programme

- Wenn p zwischen left_edge_C2 und right_edge_C2 liegt, ergibt sich eine Kante von C aus e mit p als Startpunkt.
- Wenn e right_edge_C2 schneidet, ist der Schnittpunkt ein Eckpunkt von C. Liegt p rechts von right_edge_C2, ergeben sich zwei Kanten vom Schnittpunkt aus, nämlich eine aus e für den linken Rand von C und eine aus right_edge_C2 für den rechten Rand. Liegt p links von right_edge_C2, ergibt sich keine neue Kante.



The two possibilities when e intersects *right_edge_C2*

2.5. Lineare Programme

- Wenn e left_edge_C2 schneidet, ist der Schnittpunkt ebenfalls Eckpunkt von C. Liegt p links von left_edge_C2, ergibt sich eine neue Kante aus e für den linken Rand von C. Andernfalls ergibt sich eine neue Kante aus left_edge_C2 für den linken Rand von C.

2.5. Lineare Programme

Da diese Schritte alle konstante Zeit benötigen, folgt:

Theorem 2.30: Der Schnitt zweier konvexer polygonaler Regionen in der Ebene benötigt $O(n)$ Zeit.

Aus der erreichten Verbesserung der Rekurrenz zu

$$T(n) = \begin{cases} O(1) & n = 1 \\ O(n) + 2T\left(\frac{n}{2}\right) & n > 1 \end{cases}$$

ergibt sich

Korollar 2.31: Der gemeinsame Schnitt von n Halbebenen in der Ebene kann in $O(n \log n)$ Zeit mit $O(n)$ Speicher bestimmt werden.

2.5. Lineare Programme

Inkrementelle Lineare Programmierung

Die vollständige Bestimmung der Schnittmenge von n Halbebenen erfordert $O(n \log n)$ Schritte. Sucht man aber nur eine Lösung, so ist dieses Problem einfacher.

Diese Fragestellung ist eng verwandt mit den linearen Programmen der linearen Optimierung. Ein solches "Programm" ist eine Aufgabe des Typs

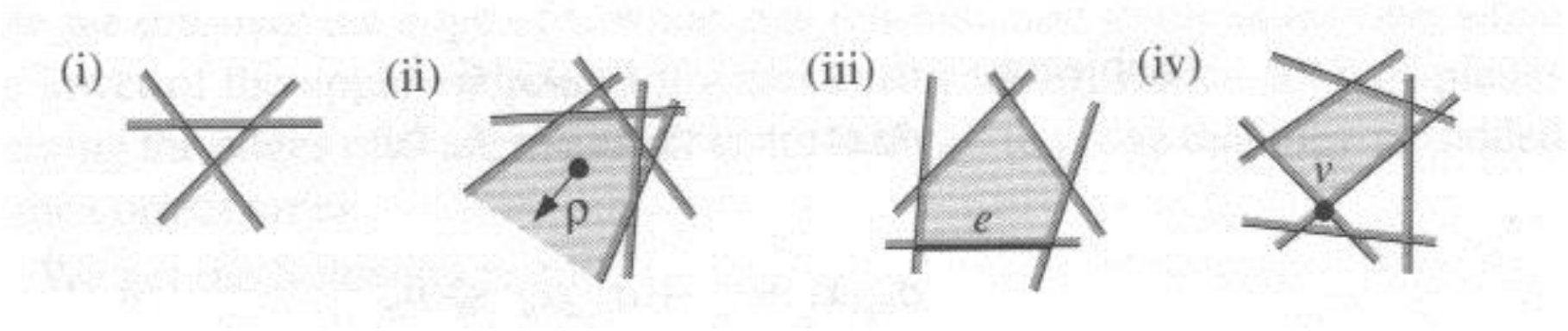
$$
\begin{array}{lccccccccc}
\text{Maximiere} & c_1x_1 & + & c_2x_2 & + & \ldots & + & c_dx_d & & \\
\text{unter} & a_{11}x_1 & + & a_{12}x_2 & + & \ldots & + & a_{1d}x_d & \leq & b_1 \\
\text{Neben}- & \vdots & & \vdots & & \ldots & & \vdots & & \vdots \\
\text{bedingungen} & a_{n1}x_1 & + & a_{n2}x_2 & + & \ldots & + & a_{nd}x_d & \leq & b_n
\end{array}
$$

2.5. Lineare Programme

$c_1x_1 + c_2x_2 + \ldots + c_dx_d$ heißt **Zielfunktion**, während die Nebenbedingungen den Raum der **zulässigen Lösungen** angeben. Eine in der Praxis effiziente Lösung stellt der **Simplexalgorithmus** dar. Allerdings sind die Algorithmen der linearen Optimierung für $d = 2$ oder $d = 3$ nur beschränkt effizient. Wir werden daher andere Ansätze kennenlernen.

2.5. Lineare Programme

Die Menge der n linearen Beschränkungen heiße wieder H.
Als Zielfunktion nutzen wir $f_c(p) = c_x p_x + c_y p_y$.
Für die Lösung $p \in H$ mit $f_c(p)$ gibt es vier Fälle:



i. Das lineare Programm ist unzulässig, d. h. es gibt keine zulässige Lösung.
ii. Das zulässige Gebiet ist unbeschränkt in Richtung c. Als Lösung ergibt sich für uns die Beschreibung eines zulässigen Strahls in Richtung c.
iii. Die zulässige Region wird durch eine zu c senkrechte Kante beschränkt. Jeder Punkt auf der Kante ist optimal.
iv. Wenn keiner der obigen Fälle eintritt, ist das lineare Programm eindeutig lösbar und der in Richtung c extreme Eckpunkt ist die Lösung.

2.5. Lineare Programme

Unser Algorithmus wird inkrementell arbeiten, also eine Beschränkung nach der anderen hinzufügen und dabei stets eine optimale Lösung der Teilprobleme finden. Dazu fügen wir zwei zusätzliche Beschränkungen m_1, m_2 mit sehr großer Konstante M ein, um die Fälle ii. und iii. zu vermeiden, ohne das Problem im Fall iv. zu verändern.

$$m_1 := \begin{cases} p_x \leq M & c_x > 0 \\ -p_x \leq M & \text{sonst} \end{cases}$$

$$m_2 := \begin{cases} p_y \leq M & c_y > 0 \\ -p_y \leq M & \text{sonst} \end{cases}$$

Im Fall iii. fordern wir die Suche nach der lexikographisch kleinsten Lösung.

2.5. Lineare Programme

Mit $H = \{h_1, \ldots, h_n\}$, m_1, m_2, $C = m_1 \cap m_2 \cap h_1 \cap \cdots \cap h_n$definieren wir

$$\begin{aligned} H_i &:= \{m_1, m_2, h_1, \ldots, h_i\} \\ C_i &:= m_1 \cap m_2 \cap h_1 \cap \cdots \cap h_i \\ v_i &:= \text{Lösung von } (H_i, C_i) \end{aligned}$$

und finden

$$C_0 \supseteq C_1 \supseteq C_2 \supseteq \ldots \supseteq C_n = C$$

Im Fall i. gilt $C_i = \emptyset$ und $C_j = \emptyset$ für $j > i$, so dass der Algorithmus bei unzulässigen Problemen einfach aussteigen kann.

2.5. Lineare Programme

Um das Hinzufügen einer Schranke h_i zu untersuchen, nutzen wir das folgende Lemma.

Lemma 2.32: Seien $1 \leq i \leq n$ und C_i, v_i definiert wie zuvor. Dann gilt

i. $v_{i-1} \in h_i$, dann $v_i = v_{i-1}$.

ii. Wenn $v_{i-1} \notin h_i$, dann ist entweder $C_i = \emptyset$ oder $v_i \in l_i$ wobei l_i die berandende Gerade ist.

Beweis:

i. Wenn $v_{i-1} \in h_i$, dann ist $v_i = v_{i-1}$, da $v_{i-1} \in C_{i-1} \supset C_i$ optimal ist.

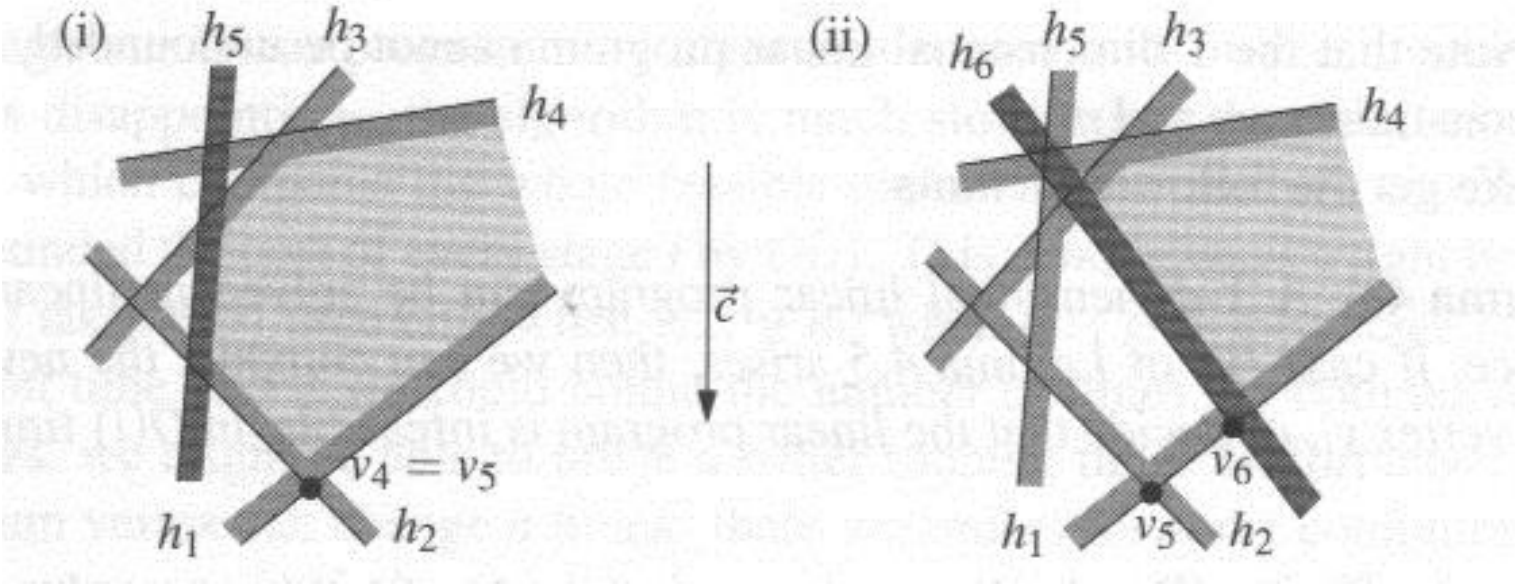
ii. Annahme: C_i nicht leer und v_i nicht auf l:

Betrachte die Stecke $\overline{v_{i-1}v_i}$ und ihren Schnittpunkt mit l:

Da $v_{i-1} \in C_{i-1}$ und $v_i \in C_i$ gibt es diesen Schnittpunkt. Da f_c linear ist, nimmt der Wert von v_{i-1} zu v_i hin ab, ist also am zulässigen Schnittpunkt mit l_i höher als bei $v_i \Rightarrow$ Widerspruch.

QED

2.5. Lineare Programme



2.5. Lineare Programme

Im ersten Fall hat unser Algorithmus keine Arbeit. Im zweiten Fall müssen wir die beiden Extremalpunkte von C_i auf l_i finden und ihre Werte vergleichen. Dies ist nichts weiter als ein 1D lineares Programm. Wenn h_i nicht parallel zur *y*-Achse ist, können wir die Punkte auf l_i über x parametrisieren und $f_c(p) = f_c(p_x)$ ist eine Funktion von x. Es ergibt sich

Maximiere $\widetilde{f_c(x)}$

mit $x \geq \sigma(h, l_i)$, $h \in H_{i-1}$,$l_i \cap h$ begrenzt nach links bei der x-Koordinate $\sigma(h, l_i)$ des Schnittes;
und mit $x \leq \sigma(h, l_i)$, $h \in H_{i-1}$,$l_i \cap h$ begrenzt nach rechts bei der x-Koordinate $\sigma(h, l_i)$ des Schnittes.

Das Intervall $[x_{\text{left}}, x_{\text{right}}]$ der zulässigen Punkte auf l_i lässt sich dann in linearer Zeit ermitteln.

2.5. Lineare Programme

Lemma 2.33: Ein eindimensionales lineares Programm kann in linearer Zeit gelöst werden, d. h. Schritt 2 des vorangegangenes Lemmas benötigt $O(i)$ Schritte.

Als Algorithmus ergibt sich

Algorithm 2DBOUNDEDLP($H, \vec{c}, m_1, m_2$)
Input. A linear program $(H \cup \{m_1, m_2\}, \vec{c})$, where H is a set of n half-planes, $\vec{c} \in \mathbb{R}^2$, and m_1, m_2 bound the solution.
Output. If $(H \cup \{m_1, m_2\}, \vec{c})$ is infeasible, then this fact is reported. Otherwise, the lexicographically smallest point p that maximizes $f_{\vec{c}}(p)$ is reported.

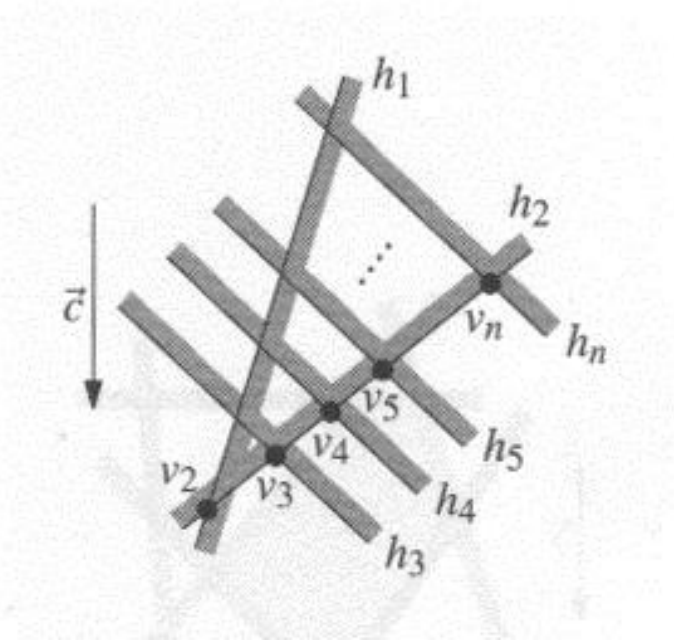
1. Let v_0 be the corner of C_0.
2. Let $h_1, \ldots, h_n$ be the half-planes of H.
3. **for** $i \leftarrow 1$ **to** n
4. **do if** $v_{i-1} \in h_i$
5. **then** $v_i \leftarrow v_{i-1}$
6. **else** $v_i \leftarrow$the point p on ℓ_i that maximizes $f_{\vec{c}}(p)$, subject to the constraints in H_{i-1}.
7. **if** p does not exist
8. **then** Report that the linear program is infeasible and quit.
9. **return** v_n

2.5. Lineare Programme

Mit der Formel aus dem Lemma für natürliche Zahlen folgt die Aussage.

Lemma 2.34: Algorithmus 2DBOUNDEDLP berechnet die Lösung eines beschränkten linearen Programmes mit n Bedingungen in $O(n^2)$ Zeit mit $O(n)$ Speicher.

Dieses unbefriedigende Ergebnis ist in einigen Fällen optimal!



2.5. Lineare Programme

Nicht-deterministisches Lineares Programmieren

Das Ergebnis des vorigen Abschnittes ist nicht gerade ermutigend. Ein Blick auf das Beispiel gibt jedoch einen Hoffnungsschimmer. Wenn wir die Halbebenen in einer anderen Reihenfolge abarbeiten, läuft unser Algorithmus schnell. Leider können wir diese Reihenfolge nur schwer zu Beginn ermitteln. Dennoch hilft eine zufällige Auswahl der Reihenfolge hier weiter! Wir werden nämlich zeigen, dass fast alle Reihenfolgen eine lineare Laufzeit erzeugen!

2.5. Lineare Programme

Algorithm 2DRANDOMIZEDBOUNDEDLP($H, \vec{c}, m_1, m_2$)
Input. A linear program $(H \cup \{m_1, m_2\}, \vec{c})$, where H is a set of n half-planes, $\vec{c} \in \mathbb{R}^2$, and m_1, m_2 bound the solution.
Output. If $(H \cup \{m_1, m_2\}, \vec{c})$ is infeasible, then this fact is reported. Otherwise, the lexicographically smallest point p that maximizes $f_{\vec{c}}(p)$ is reported.

1. Let v_0 be the corner of C_0.
2. Compute a *random* permutation $h_1, \ldots, h_n$ of the half-planes by calling RANDOMPERMUTATION($H[1 \cdots n]$).
3. **for** $i \leftarrow 1$ **to** n
4. **do if** $v_{i-1} \in h_i$
5. **then** $v_i \leftarrow v_{i-1}$
6. **else** $v_i \leftarrow$the point p on ℓ_i that maximizes $f_{\vec{c}}(p)$, subject to the constraints in H_{i-1}.
7. **if** p does not exist
8. **then** Report that the linear program is infeasible and quit.
9. **return** v_n

2.5. Lineare Programme

Mit einer Funktion RANDOM(k), die in konstanter Zeit eine zufällige Zahl zwischen 1 und k auswählt, definieren wir

Algorithm RANDOMPERMUTATION(A)
Input. An array $A[1 \cdots n]$.
Output. The array $A[1 \cdots n]$ with the same elements, but rearranged into a random permutation.
1. **for** $k \leftarrow n$ **downto** 2
2. **do** $rndindex \leftarrow$ RANDOM(k)
3. Exchange $A[k]$ and $A[rndindex]$.

Die Analyse nicht deterministischer Algorithmen erfolgt durch Analyse aller möglichen Abläufe und dem Bilden des mit der Wahrscheinlichkeit des Auftretens gewichteten Mittels.

2.5. Lineare Programme

Lemma 2.35: Das 2-dimensionale lineare Programm mit n Nebenbedingungen kann in $O(n)$ erwarteter Zeit mit maximal $O(n)$ Speicher gelöst werden.

Beweis: Der Speicherbedarf ist wie zuvor linear. RANDOMPERMUTATION benötigt lineare Zeit. Die Frage bleibt, wann $v_{i-1} \notin h_i$ gilt. Mit den Zufallsvariablen

$$X_i = \begin{cases} 1 & v_{i-1} \notin h_i \\ 0 & \text{sonst} \end{cases}$$

erhalten wir als Aufwand

$$\sum_{i=1}^{n} O(i)\, X_i$$

Für den Erwartungswert der Summe gilt wegen der Unabhängigkeit der X_i

$$E\left(\sum_{i=1}^{n} O(i) X_i\right) = \sum_{i=1}^{n} O(i) E(X_i)$$

2.5. Lineare Programme

Wir benötigen noch die Erwartungswerte $E(X_i)$.
Wir nutzen die "Rückwärtsanalyse": Wenn der Algorithmus fertig ist, hat er v_n bestimmt. v_n ist eine Ecke von $C = C_n$. Wenn $v_n \neq v_{n-1}$ ist, und somit durch mindestens 2 Halbebenen definiert ist, so muss v_n durch die Halbebene h_n definiert werden. Wegen der zufälligen Wahl der Reihenfolge ergibt sich als Wahrscheinlichkeit maximal $\frac{2}{n}$. Es folgt

$$\sum_{i=1}^{n} O(i)E(X_i) \leq \sum_{i=1}^{n} O(i)\frac{2}{i} = O(n)$$

QED.

Es sei betont, dass diese erwartete Zeit für alle Eingaben gilt, es ist also nicht die durchschnittliche Laufzeit!

2.5. Lineare Programme

Unbeschränkte Lineare Programme

Bisher haben wir unbeschränkte lineare Programme durch zusätzliche Nebenbedingungen vermieden. Wir wollen diese zusätzlichen Nebenbedingungen nun vermeiden. Dazu betrachten wir einen Strahl

$$\rho = p + \delta d, \delta > 0$$

Dabei nimmt f_c beliebig große Werte auf dem Strahl an, falls $d \cdot c > 0$.
Ferner beschränkt eine Nebenbedingung den Strahl nicht, falls für die nach innen gerichtete Normale $n(h)$ gilt: $n(h) \cdot d > 0$.
Diese beiden Bedingungen reichen, um das LP (H, c) unbeschränkt zu machen:

Lemma 2.36: Ein lineares Programm ist unbeschränkt *gdw* es einen Vektor d mit $d \cdot c > 0$ und $d \cdot n(h) \geq 0$ für alle $h \in H$ gibt, und das lineare Programm H' zulässig ist, wobei $H' = \{h \in H | n(h) \cdot d = 0\}$.

2.5. Lineare Programme

Beweis:
"⇒" bereits vor dem Lemma gezeigt.

"⇐" Sei (H, c) ein lineares Programm und d wie beschreiben.
Da (H, c) zulässig ist, existiert ein Punkt $p_0 \in \bigcap_{h \in H'} h$.
Der Strahl $\rho_0 := \{p_0 + \lambda d | \lambda > 0\}$ liegt dann wegen $d \cdot n(h) = 0$ für $h \in H'$ in allen $h \in H'$.
Ferner wird die Zielfunktion wegen $d \cdot c > 0$ beliebig groß.
Für eine Halbebene $h \in H \backslash H'$ gilt: $d \cdot n(h) > 0$, also existiert ein λ_h, so dass $p_0 + \lambda d \in h$ für $\lambda > \lambda_h$.
Mit $\lambda' := \max_{h \in H \backslash H'} \{\lambda_h\}$ und $p := p_0 + \lambda' d$ ist

$$\rho := \{p + \lambda d \,|\lambda > 0\}$$

zulässig.

QED.

2.5. Lineare Programme

Analog zu unseren Betrachtungen der Gussformen bestimmen wir nun, ob (H, c) unbeschränkt ist. Wir drehen das Koordinatensystem, so dass $c = (0,1)$. Jede Richtung $d = (d_x, d_y)$ mit $d \cdot c > 0$ kann nun als $d = (d_x, 1)$ beschrieben werden. Aus der Ungleichung

$$d \cdot n(h) = d_x n_x + n_y \geq 0$$

folgt

$$d_x n_x \geq -n_y$$

Dies ist ein eindimensionales lineares Programm $\hat{H}$ (ohne Zielfunktion), das sich in linearer Zeit lösen lässt.

2.5. Lineare Programme

Wenn wir eine Lösung $d_x{}^*$ haben, suchen wir die Halbebenen mit $d_x{}^* + n_y = 0$.
Diese zu $d = (d_x{}^*, 1)$ parallelen Geraden ergeben ein einfaches lineares Programm $\bar{H}'$.
Wenn es zulässig ist, ist das Ausgangsprogramm unbeschränkt und das Lemma gestattet das Finden des Strahls in $O(n)$.
Wenn es unzulässig ist, ist (H, c) beschränkt.
Im letzteren Fall nennen wir die beiden Nebenbedingungen h_1 (linker Rand) und h_2 (rechter Rand), die das Programm $\bar{H}'$ unzulässig machen und beim Lösen gefunden werden, **Zertifikate**.
Diese Zertifikate machen $m_1 \text{und } m_2$ überflüssig, wobei wir aber testen müssen, ob es eine lexikographische kleinste Lösung von $(\{h_1, h_2\}, c)$ gibt, etwa $n(h_1) = -c = (0, -1)$.
Dann durchsuchen wir die übrigen Halbebenen nach einer anderen Wahl für h_2 mit $n_x(h_2) > 0$.
Wenn eine solche Lösung nicht existiert, gibt es keine lexikographisch kleinste Lösung und wir geben einen Strahl in Richtung $(-1, 0)$ mit lauter optimalen Lösungen an.

2.5. Lineare Programme

Algorithm 2DRANDOMIZEDLP$(H,\vec{c})$
Input. A linear program $(H,\vec{c})$, where H is a set of n half-planes and $\vec{c} \in \mathbb{R}^2$.
Output. If $(H,\vec{c})$ is unbounded, a ray is reported. If it is infeasible, then two or three certificate half-planes are reported. Otherwise, the lexicographically smallest point p that maximizes $f_{\vec{c}}(p)$ is reported.

1. Determine whether there is a direction vector $\vec{d}$ such that $\vec{d}\cdot\vec{c} > 0$ and $\vec{d}\cdot\vec{\eta}(h) \geqslant 0$ for all $h \in H$.
2. **if** $\vec{d}$ exists
3. **then** compute H' and determine whether H' is feasible.
4. **if** H' is feasible
5. **then** Report a ray proving that $(H,\vec{c})$ is unbounded and quit.
6. **else** Report that $(H,\vec{c})$ is infeasible and quit.
7. Let $h_1,h_2 \in H$ be certificates proving that $(H,\vec{c})$ is bounded and has a unique lexicographically smallest solution.
8. Let v_2 be the intersection of ℓ_1 and ℓ_2.
9. Let $h_3,h_4,\ldots,h_n$ be a random permutation of the remaining half-planes in H.
10. **for** $i \leftarrow 3$ **to** n
11. **do if** $v_{i-1} \in h_i$
12. **then** $v_i \leftarrow v_{i-1}$
13. **else** $v_i \leftarrow$the point p on ℓ_i that maximizes $f_{\vec{c}}(p)$, subject to the constraints in H_{i-1}.
14. **if** p does not exist
15. **then** Let h_j,h_k (with $j,k<i$) be the certificates (possibly $h_j = h_k$) with $h_j \cap h_k \cap \ell_i = \emptyset$.
16. Report that the linear program is infeasible, with h_i,h_j,h_k as certificates, and quit.
17. **return** v_n

2.5. Lineare Programme

Literatur

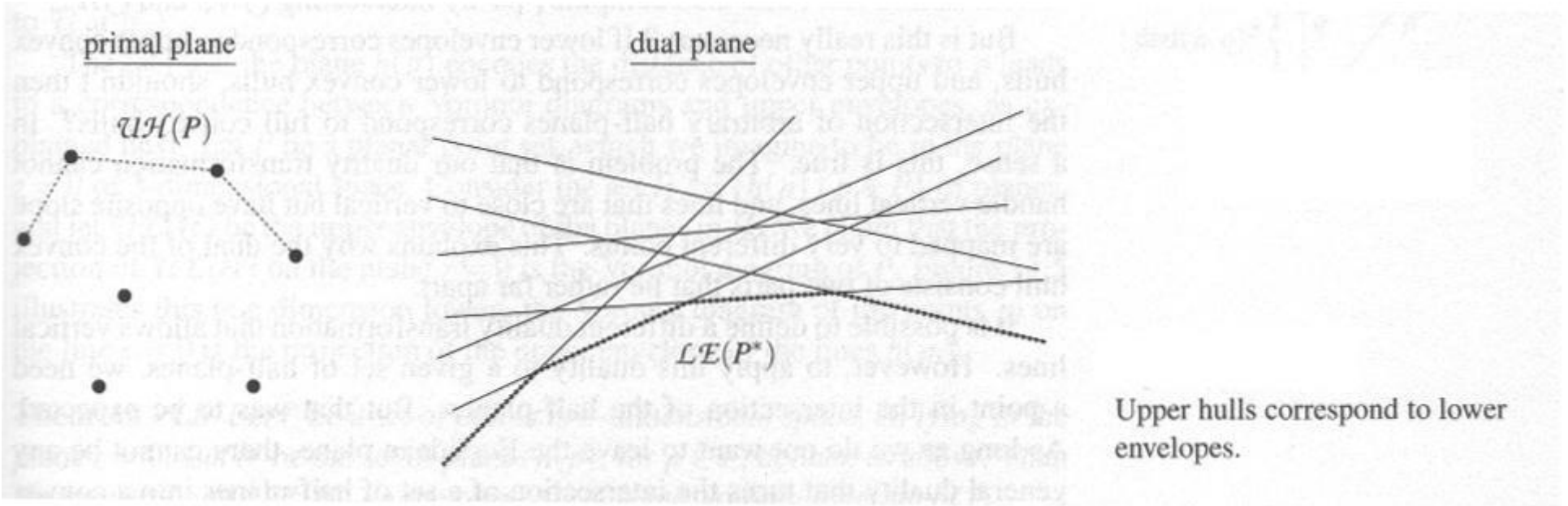
Der nicht-deterministische Algorithmus für lineare Programme stammt von R. Seidel [R. Seidel. Small-dimensional linear programming and convex hulls made easy. Discrete Comput. Geom., 6:423-434, 1991.], wobei er den Parameter symbolisch behandelt.

Es ist nützlich zu erkennen, dass die Berechnung des Schnittes von n Halbebenen dual zur Berechnung der konvexen Hülle von Punkten ist.
Um dies zu sehen, muss man zunächst Dualität verstehen. In der Ebene definiert man zu jedem Punkt $p = (p_x, p_y) \in \mathbb{R}^2$ die Gerade $p^* := \{(x, y) | y = p_x x - p_y\}$ (Dualitätstransformation). Zu einer Gerade $l := \{(x, y) | y = mx + b\}$ definiert man den Punkt $l^* := (m, -b) \in \mathbb{R}^2$. Dann gilt

$$p \in l \Leftrightarrow l^* \in p^* \text{ und } p \text{ oberhalb } l \Leftrightarrow l^* \text{ oberhalb } p^*$$

[Es gibt weitere Dualitätstransformationen.]
Betrachtet man nun die obere konvexe Hülle einer Punktmenge *P*, so entspricht sie genau dem oberen Teil des Schnittes der dualen Halbebenen.

2.5. Lineare Programme



Upper hulls correspond to lower envelopes.

Dualität wird uns später wieder begegnen.