Geometrische Algorithmen

$\longrightarrow$ Bisher: Primitive erzeugen und darstellen

$\longrightarrow$ jetzt: "Rechnen mit Primitiven"

Also beispielsweise:

- Primitive (Linien, Dreiecke) schneiden
- konvexe Hülle bilden
- Voronoi-Diagramme ausrechnen

$\longrightarrow$ insbesondere: was geschieht bei vielen Primitiven?

$\Rightarrow$ O-Kalkül :-(

Analyse graphisch-geometrischer Probleme

Zuerst: Problemspezifikation

Beispiel: Nächster Nachbar

Gegeben:

Eine feste Menge M von n Punkten im $\mathbb{R}^2$

Gesucht:

- Für viele Punkte q_i jeweils der nächste Nachbar in M.
  (Die Anfragepunkte q sind nacheinander abzuarbeiten und sind bei der Vorverarbeitung von M noch unbekannt.)
- Für einen Punkt q der nächste Nachbar in M.
  (Nur genau ein Anfragepunkt.)

- Zu jedem Punkt $p \in M$ der jeweils nächste Nachbar in M.
- Zu jedem Punkt $p \in M$ die jeweils k nächsten Nachbarn in M, mit einer ebenfalls fest vorgegebenen Konstanten k.
- Zu einer gegebenen Menge $Q \subset \mathbb{R}^2$ von Anfragepunkten $q_1, q_2, ..., q_k$ (simultan) der jeweils nächste Nachbar in M.

Oder als andere Aufgabenstellung:

Gegeben:

Eine Menge M von Punkten im $\mathbb{R}^2$

Gesucht: Eine effiziente Datenstruktur, die es erlaubt, folgende Operationen in beliebiger Reihenfolge und Anzahl nacheinander auszuführen:

1. Erweitere M um einen neuen Punkt $p \in \mathbb{R}^2$
2. Entferne aus M den Punkt $p \in M$
3. Bestimme zu $q \in \mathbb{R}^2$die k nächsten Nachbarn in M

$\rightarrow$ Dynamisierung

$\Rightarrow$ macht das Leben schwerer

Vorverarbeitung

$\longrightarrow$ wird Vorverarbeitung zugelassen, können Probleme oft effizienter gelöst werden

Gegeben:

Feste Menge M von n Punkten im $\mathbb{R}^2$

Gesucht:

Vorverarbeitung von M, so daß zu jedem Anfragepunkt q sehr schnell entschieden werden kann, welcher Punkt von M zu q den geringsten Abstand hat.

Problemklassifikation

Klassifikation über Räume

$\Rightarrow \mathbb{R}, \mathbb{R}^2, \mathbb{R}^3, \ldots, \mathbb{R}^n$).

höherdimensionale Probleme:

- statistische Probleme mit vieldimensionalen Parametervektoren
- lineares Programmieren (engl. linear programming).

Klassifikation nach Objekttypen

Dim.	Beispiele
0	Punkt
1	Strecke, Halbstrahl, Gerade, Kurvenstück, geschlossene Kurve
2	Rechteck, konvexes Polygon, einfaches Polygon, Polygon mit Löchern, Halbebene, Polygon mit Asymptoten, gekrümmtes Flächenstück
3	Quader, konvexes Polyeder, Polyeder mit Asymptoten, Körper mit gekrümmten Begrenzungsflächen
> 3	Halbraum, Polyeder, Polytop

$\rightarrow$ n-dimensionale Objekte können in jeden Raum ab Dimension $n+1$ eingebettet werden.

Klassifikation über verwendete Grundoperationen

Gegeben

M_1, M_2,... geometrische Gebilde.

Gesucht

Mengenoperation Prädikat	Bedeutung
$M := M_1 \cup M_2$	Zusammenfassung von zwei geometrischen Gebilden zu einem Gesamtgebilde.
$M := M_1 \cap M_2$	Bestimmung des Schnittgebildes zweier geometrischer Gebilde, z.B. Schnitt zweier Polygone bestimmen.
$M := \bigcap M_i$	Bestimmung des Schnittgebildes vieler Gebilde (z.B. Halbebenenschnitt).

$M := M_1 - M_2$	Entfernen von M_2 aus M_1.
$M := \bigcup_{i \neq j} M_i \cap M_j$	Alle paarweisen Schnittgebilde, z.B. alle Schnittpunke von n gegebenen Strecken in der Ebene.
$M_1 \in M_2$	Z.B. Punktlokalisation (M_1 einelementig).
$M_1 \subset M_2$	Enthaltensein bei graphischen Objekten.

Hüllenbildung über Mengenoperationen:

Gegeben

Geometrisches Gebilde M_1.

Gesucht

Konvexe Menge M_2 mit $M_1 \subset M_2$ und M_2 minimal.

Zerlegungsprobleme (Triangulation):

Gegeben

Geometrisches Gebilde M

Gesucht

Dreiecke Δ_i, so daß $M = \cup_{i=1}^{n} \Delta_i$ und $Maß(\Delta_i \cap \Delta_j) = 0$
für alle $i \neq j$.

Algorithmenentwurf und Analyse

$\longrightarrow$ neben Effizienz (O-Kalkül, Konstanten) sind im praktischen Einsatz oft weitere Eigenschaften wichtig:

- Suche geeignete Repräsentation von graphischen Objekten und passende interne Datenstrukturen.
- Suche Algorithmus mit geringstem Programmieraufwand.
- Suche Algorithmus mit asymptotisch bestem Zeitverhalten.
- Definiere typische Problemkomplexitäten in der Praxis und daraus abgeleitete Abschätzungen für konkrete Rechenzeiten.
- Bestimme Algorithmus mit bester Eignung für die Implementierung in professioneller Software.
- Bestimme bzw. verbessere die numerische Stabilität.

- Bestimme Algorithmus mit bester Parallelisierbarkeit.
- Kombiniere Algorithmen einer bestimmten Problemklasse, um insgesamt einen besseren Algorithmus zu erhalten.

Algorithmenmodell

$\longrightarrow$ früher: Verwendung von Integer-Arithmetik angestrebt, da Floating-Point Operationen vergleichsweise teuer

$\longrightarrow$ heute: etwa gleicher Aufwand

Daher folgendes Maschinenmodell sinnvoll (RAM-Modell):

- arithmetischen Operationen $+, -, /, *, <, \leq$: Rechenzeit $O(1)$
- Zeiger- und Indexoperationen ($B := A[i, j]$): $O(1)$
- Speicherreservierung, Initialisierung für Maschinenwort: $O(1)$

Algorithmenkomplexität

Definition

Die Funktion $T(P)$ beschreibt die Anzahl der Zeiteinheiten (= Elementaroperationen), die der Algorithmus A benötigt, um die Eingabedaten P zu verarbeiten

$\rightarrow$ eigentlich Interessant: was geschieht bei verschiedenen Parametersätzen

$T_{max}(|P| = n) :=$ Maximale Anzahl der Zeiteinheiten, die bei Problemen der Komplexität n nötig ist.

$$T_{max}(n) := \max_{P,\ |P|=n} T(P).$$

Interessant - vor allem aus praxisbezogener Sicht - ist die durchschnittliche Zeit, die der Algorithmus zur Verarbeitung benötigt:

$$T_{mittel}(n) := MittelwertP,\ |P| = nT(P).$$

minimale Zeit:

$$T_{min}(n) := \min_{P,\ |P|=n} T(P).$$

es gilt (natürlich):

$$T_{min}(n) \leq T_{mittel}(n) \leq T_{max}(n).$$

Speicherbedarf: statt T schreibe S, Rest analog

Asymptotisches Wachstum, O-Kalkül

$O(f(n)) :=$ $\{g(n) | \exists\ n_0$ und $c > 0 : \forall n \geq n_0$ gilt
$g(n) \leq c \cdot f(n)\}$,
d.h. $g(n)$ wächst höchstens so stark wie $f(n)$.

$\Omega(f(n)) :=$ $\{g(n) | \exists\ n_0$ und $c > 0 : \forall n \geq n_0$ gilt
$g(n) \geq c \cdot f(n)\}$,
d.h. $g(n)$ wächst mindestens so stark wie $f(n)$.

$\Theta(f(n)) :=$ $\{g(n) | \exists\ c_1, c_2, n_0 : \forall n \geq n_0$ gilt
$c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}$,
d.h. $g(n)$ wächst genau so stark wie $f(n)$.
Also: $g(n) = O(f(n))$ und $g(n) = \Omega(f(n))$.

Definition Untere Schranke

Für ein bestimmtes algorithmisches Problem bezeichnen wir eine Wachstumsfunktion $f(n)$ als untere Zeitschranke, wenn für jeden Algorithmus zur Lösung des Problems gilt:

$$T_{max}(n) = \Omega(f(n)).$$

$\rightarrow$ Eine untere Schranke muß nicht erreichbar sein.

$\rightarrow$ $\Omega(1)$ ist untere Schranke für alle Algorithmen.

Definition Optimaler Algorithmus

Ein Algorithmus heißt optimal (bzgl. des T_{max}-Zeitverhaltens), wenn gilt:

$$T_{max}(n) = O(f(n)),$$

mit: $f(n)$ ist untere Schranke des algorithmischen Problems.

$\rightarrow$ $f(n)$ untere Schranke $\Rightarrow T_{max}(n) = \Omega(f(n))$ und damit

$$T_{max}(n) = \Theta(f(n)).$$

Beweis für Optimalität von Algorithmen

$\longrightarrow$ nutze den zu testenden Algorithmus zur Lösung eines Standardproblems (sortieren, suchen etc.) dessen untere Zeitschranke man kennt (Problemreduktion).

$\longrightarrow$ geschieht Transformation auf eine Weise, daß Gesamtkomplexität von Algorithmus + Transformation untere Schranke des Standardproblems nicht überschreitet, so gilt:

Ein Algorithmus, der das konkrete Problem mit geringerer Zeitkomplexität lösen könnte, würde auch den notwendigen Zeitaufwand zur Lösung des Standardproblems senken.

$\Rightarrow$ Widerspruch

Untere Schranken für elementare Algorithmen

- **Sortieren**
  Das Problem, n Zahlen zu sortieren, hat eine untere Schranke von:

$$T_{max}(n) = \Omega(n \log n).$$

- **Elementeindeutigkeit**
  Das Problem, für n gegebene Zahlen zu entscheiden, ob mindestens zwei der Zahlen gleich sind, hat eine untere Schranke von:

$$T_{max}(n) = \Omega(n \log n).$$

- **Lokalisierung von Intervallen** Das Problem, bei erlaubter Vorverarbeitung für n gegebene sortierte Zahlen x_i und eine beliebige

Zahl $x \in [x_1, x_n)$ den Index i mit $x \in [x_i, x_{i+1})$ zu finden, hat eine untere Schranke von:

$$T_{max}(n) = \Omega(\log n).$$

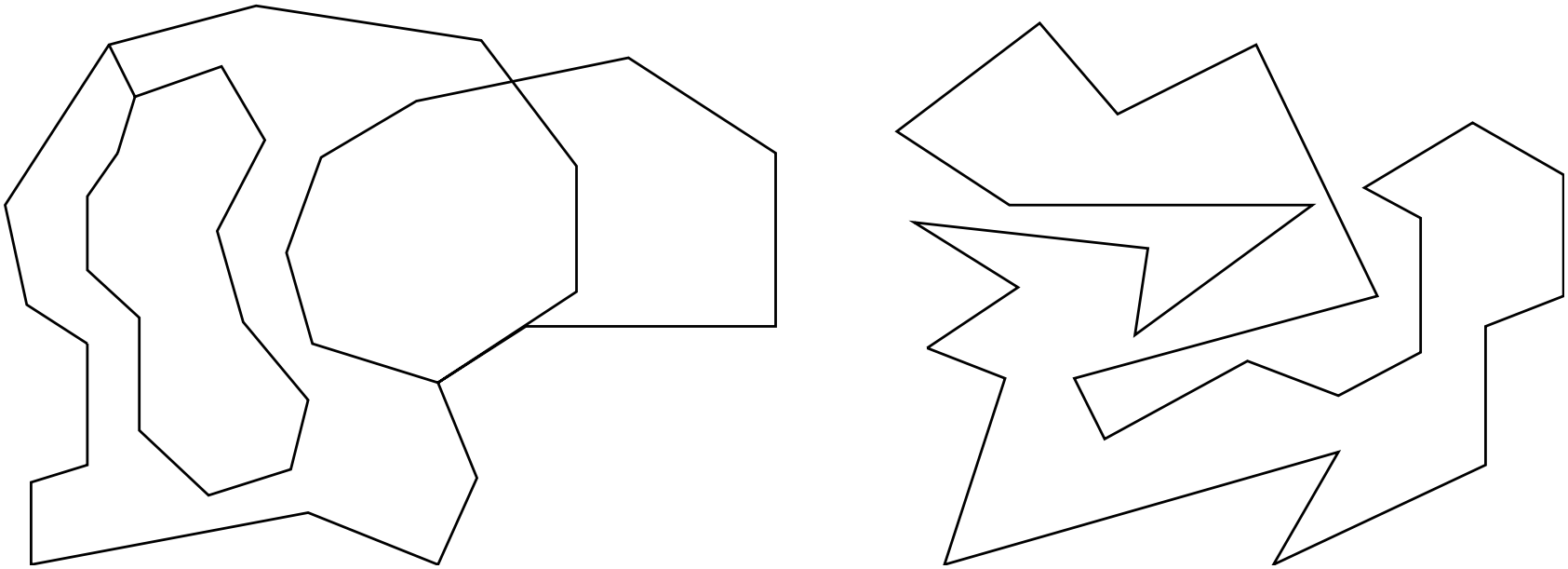
- **Minimum-Maximum-Suche**
  Das Problem, für n gegebene Zahlen das Minimum bzw. Maximum zu bestimmen, hat eine untere Schranke von:

$$T_{max}(n) = \Omega(n).$$

Definition geometrischer Objekte

$\longrightarrow$ Gerade, Halbgerade, Ebene, Halbebene siehe führe Kapitel

Polygone, Polyeder (Bezeichnung im $\mathbb{R}^n$, $n > 2$)



(a) Allgemeines Polygon; (b) einfaches Polygon.

Definition Polyeder

Ein Polyeder im $\mathbb{R}^k$ ist eine durch eine endliche Menge von Seitenpolyedern der Dimension $\mathbb{R}^{k-1}$ begrenzte Punktmenge. Die Ränder der Seitenpolyeder sind ihrerseits Polyeder der Dimension $\mathbb{R}^{k-2}$ usw. Eckpunkte werden als nulldimensionale Polyeder definiert.

Definition Polygon

Ein Polyeder der Dimension zwei heißt Polygon. Es wird dargestellt durch eine geschlossene Streckenfolge $p_1 \longrightarrow p_2 \longrightarrow ... \longrightarrow p_n \longrightarrow p_1$ von Punkten in einer Ebene. Die Randstrecken $p_i \longrightarrow p_{i+1}$ heißen Kanten oder Segmente; die von ihnen eingeschlossenen Punkte bilden das Innere des Polygons. Die Endpunkte der Kanten werden als Polygoneckpunkte bezeichnet.

Definition Einfaches Polygon

Ein Polygon heißt einfach, wenn es keine Löcher besitzt und weder selbstüberlappend noch selbstberührend ist. In diesem Fall haben je zwei nicht aneinandergrenzende Kanten keinen gemeinsamen Punkt.

$\longrightarrow$ Einfaches Polygon teilt die Ebene in zwei disjunkte Gebiete: Innere und das Äußere des Polygons.

Definition Konvexe Menge

Eine Menge P ist konvex, wenn mit $p_i, p_j \in P$ stets auch die Strecke $\overline{p_i p_j}$ ganz in P enthalten ist.

Definition Konvexe Hülle

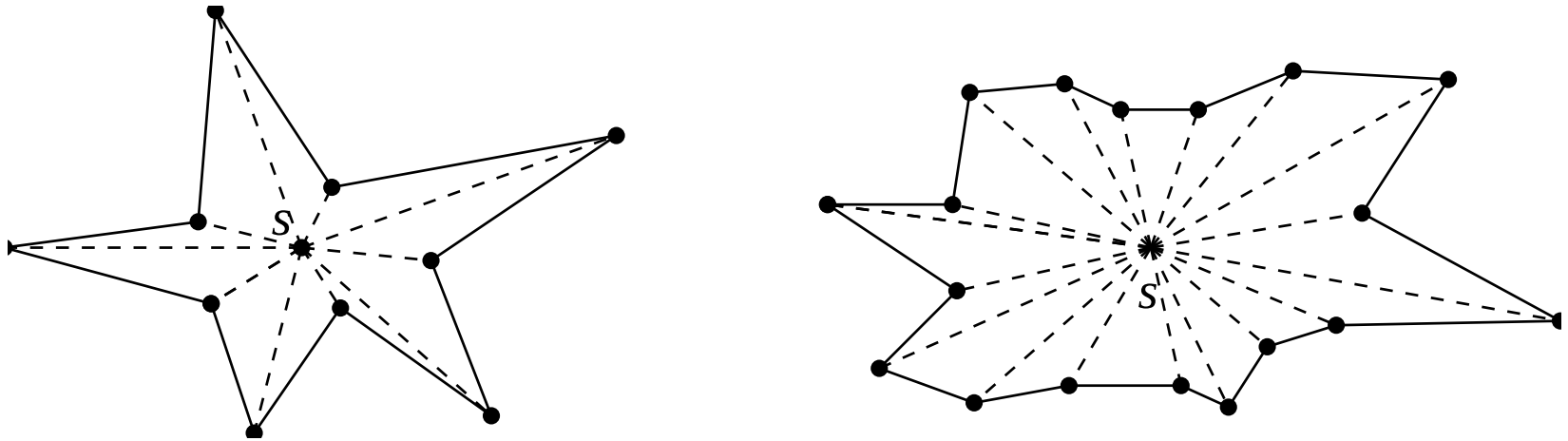
Die konvexe Hülle einer Punktmenge P im $\mathbb{R}^k$ ist der Rand der kleinsten konvexen Teilmenge von $\mathbb{R}^k$, die P enthält.

Definition Konvexes Polygon

Ein einfaches Polygon P ist konvex, wenn sein Inneres eine konvexe Menge ist.

Definition Sternpolygon

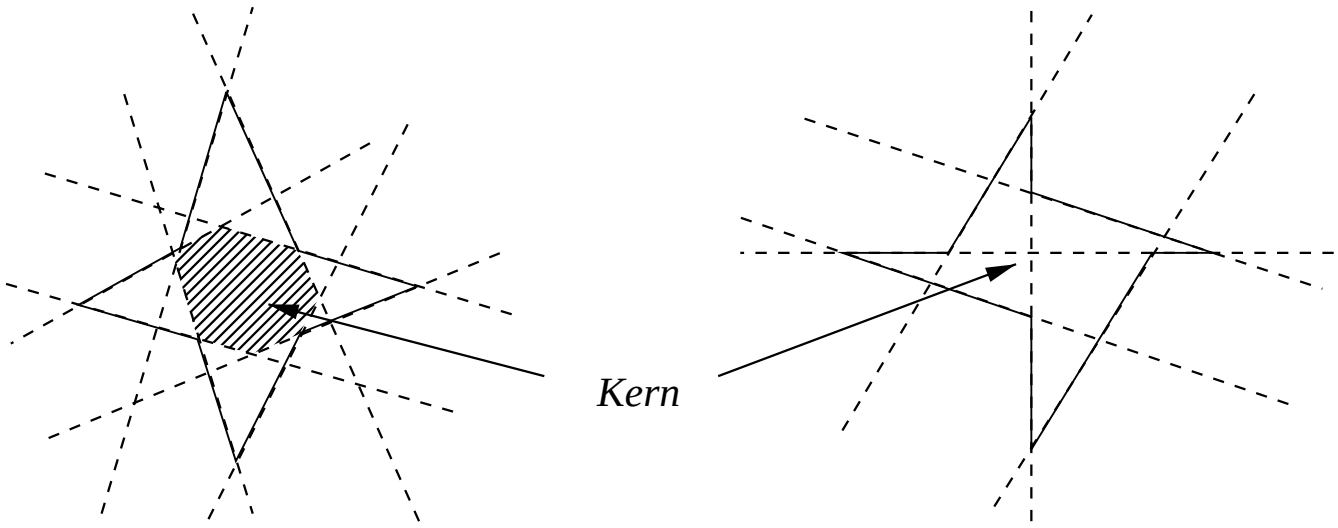
Ein einfaches Polygon P heißt Sternpolygon, wenn es einen Punkt s im Inneren von P gibt, so daß die Halbgeraden von s durch alle Eckpunkte p_i von keiner Polygonkante geschnitten werden. Der Punkt s heißt Sternpunkt.



Sternpolygone

Definition Kern eines Sternpolygons

Der Kern eines Sternpolygons wird aus der Menge aller Sternpunkte s gebildet.



Kerne von Sternpolygonen.

Allgemein gilt die Beziehung:

konvexe Polygone $\subset$ Sternpolygone $\subset$

$\subset$ einfache Polygone $\subset$ allgemeine Polygone

Orientierung von Polygonen

Definition Normaldarstellung Ein Polygon ist in Normaldarstellung gegeben, falls es als Folge von Eckpunkten $(p_1, ..., p_n)$, im Uhrzeigersinn (negativ) orientiert, vorliegt.

Gegeben

Die Eckpunktfolge $(p_1, ..., p_n)$ eines einfachen Polygons im $\mathbb{R}^2$.

Gesucht

Die Orientierung des Polygons, d.h. Antwort auf die Frage, ob das Polygon im oder gegen den Uhrzeigersinn orientiert ist.

Orientierungsbestimmung eines Polygons

begin

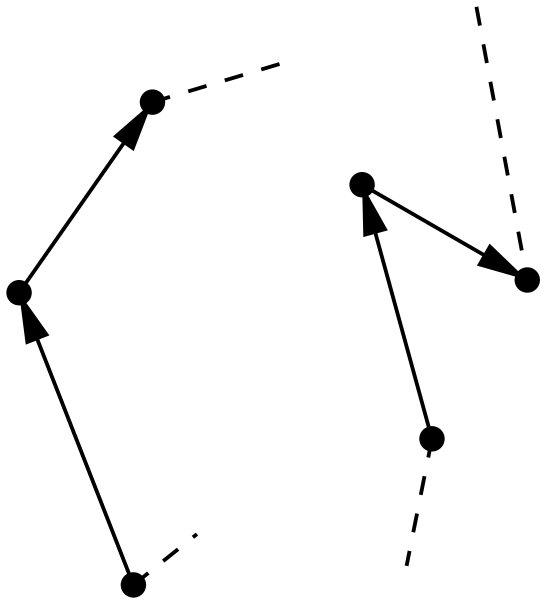
Bestimme den Punkt p_i mit minimaler $x-$Koordinate,
der zusätzlich eine maximale $y-$Koordinate hat;
Untersuche die Kanten $\overline{p_i p_{i+1}}$ und $\overline{p_i p_{i-1}}$;
if Kante $\overline{p_i p_{i+1}}$ liegt über Kante $\overline{p_i p_{i-1}}$
then Orientierung im Uhrzeigersinn;
else Orientierung im Gegenuhrzeigersinn;

end;

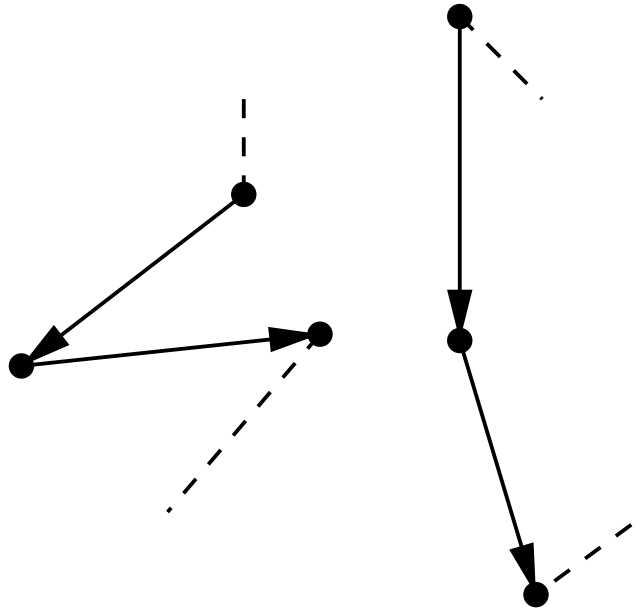
Kante $\overline{p_i p_{i+1}}$ "über" Kante $\overline{p_i p_{i-1}}$, wenn gilt:

$$\frac{p_{i+1}.y - p_i.y}{p_{i+1}.x - p_i.x} > \frac{p_i.y - p_{i-1}.y}{p_i.x - p_{i-1}.x}.$$

$\rightarrow T_{max}(n) = O(n)$.



Uhrzeigersinn



Gegenuhrzeigersinn

Steigungsvergleich in x-minimalem Punkt.

Schnittbestimmung

→ zentrale geometrische Aufgabe

→ viele Lösungsmöglichkeiten, viele unterschiedliche Randbedingungen

→ daher zuerst: Schnitt isoorientierter Strecken
später allgemeine Strecken

→ dient als Beispiel für Grundkonzepte geometrischer Algorithmen (Sweep-Line, Teile & Herrsche, Inkrementell, Randomisiert)

Schnitt isoorientierter Strecken

Gegeben

n vertikal bzw. horizontal liegende Strecken im $\mathbb{R}^2$mit $n = n_v + n_h$. ($V = \{v_1, v_2, \ ... \ v_{n_v}\}$ vertikalen Strecken, $H = \{h_1, h_2, \ ... \ h_{n_h}\}$ horizontale Strecken

Gesucht

Menge S sich schneidender Streckenpaare ($S \subseteq V \times H$).

Einfacher Algoritmus (Brute Force):

$\rightarrow$ teste jede vertikale mit jeder horizontalen Strecke

$T_{max}(n_v, n_h) = O(n_v \cdot n_h) \leq O(n^2)$.

Lösung mit Sweep-Verfahren

→ gängiges Verfahren zur Bearbeitung geometrischer Daten

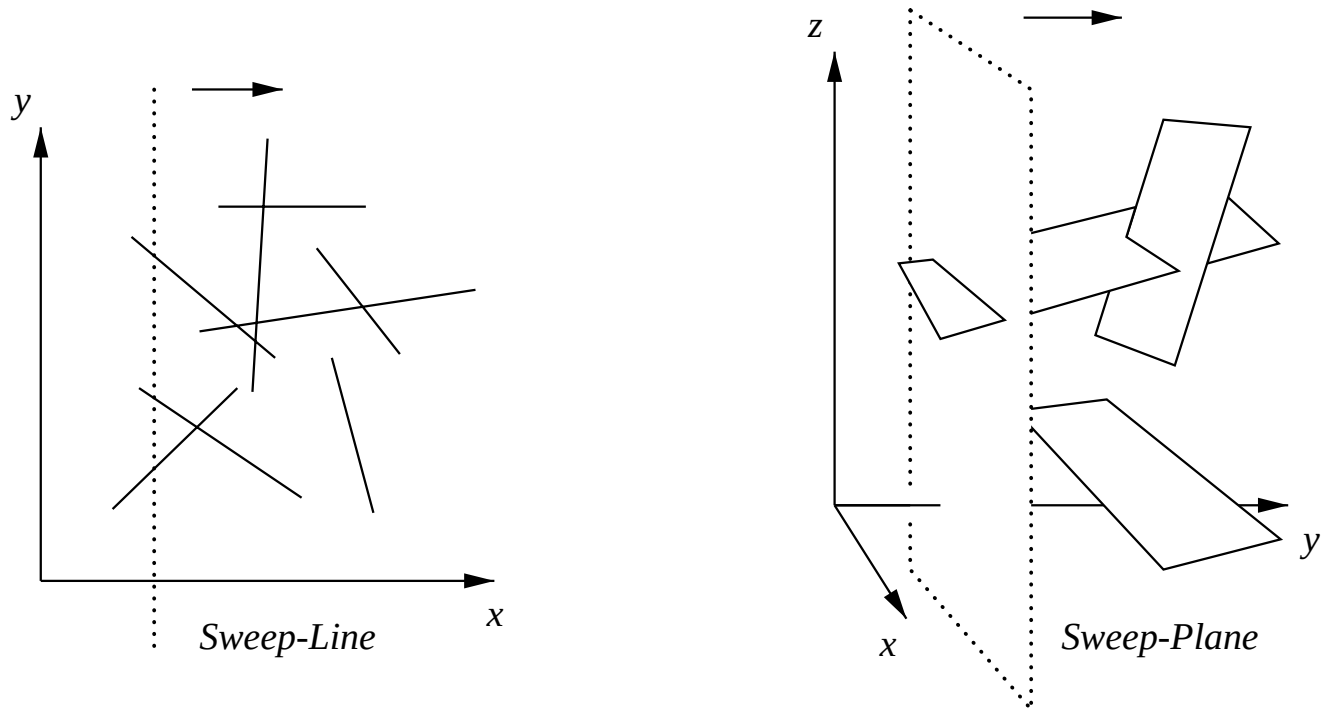
→ Gerade (Sweep-Line) wird über die Daten geschwenkt
(im höherdimensioanlen: Ebene bzw. Hyperebenen)

→ Manipulation wird an den Objekten vorgenommen, die sich
in der lokalen Umgebung der Sweep-Line befinden
(meistens: die von ihr geschnitten werden)

Damit: Aufteilung der Daten in drei Teilmengen

1. schon bearbeitete Objekte
2. aktuell zu bearbeitende Objekte
3. noch zu bearbeitende Objekte

→ bei genügender Lokalität der Objekte: sinnvolle Aufteilung



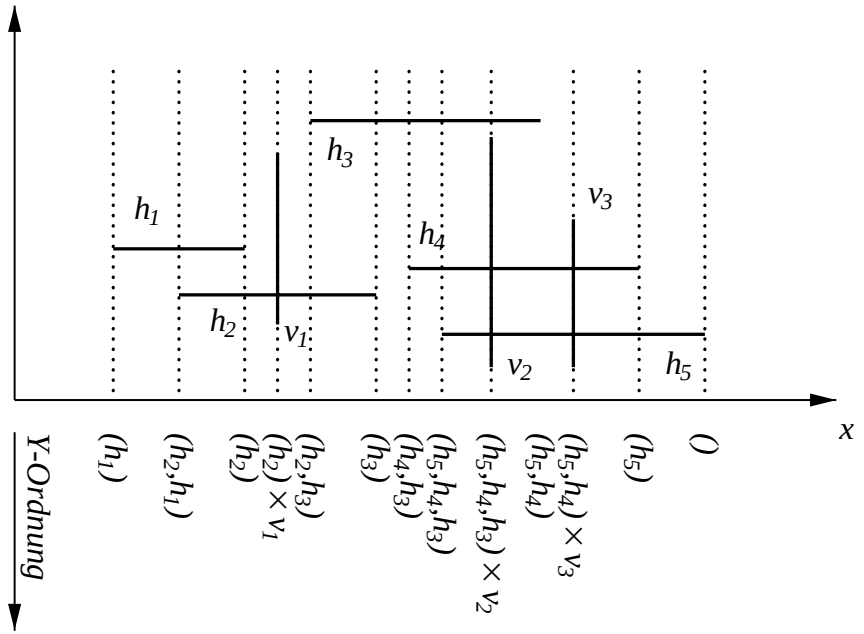
Sweep-Verfahren

Der Algorithmus verwendet folgende Datenstrukturen:

1. **X-Ordnung:**
   Enthält alle x-Ereignisse in aufsteigend sortierter Folge (z.B. Liste).
   x-Ereignisse sind:
   - x-Werte aller linken Endpunkte horizontaler Strecken $x_l(h_i)$
   - x-Werte aller rechten Endpunkte horizontaler Strecken $x_r(h_i)$
   - x-Werte vertikaler Strecken $x(v_j)$

2. **Y-Ordnung** (Sweep-Line-Status)**:**
   Enthält alle Strecken aus H, die von der Sweep-Line geschnitten oder berührt werden, sortiert nach $y(h_i)$ (z.B. als AVL-Baum mit doppeltverketteten Blättern, ein sog. Bereichssuchbaum)

$\longrightarrow$ Algorithmus: lasse Sweep-Line in pos. x-Richtung laufen, halte bei jedem x-Ereignis an



Schnitt von isoorientierten Strecken

Ereignistypen:

- Linker Endpunkt einer horizontalen Strecke erreicht
  $\Rightarrow$ füge diese in Y-Ordnung ein
- Rechter Endpunkt einer horizontalen Strecke erreicht
  $\Rightarrow$ lösche diese aus Y-Ordnung
- x-Koordinate einer vertikalen Strecke
  $\Rightarrow$ schneide mit allen in der Y-Ordnung enthaltenen Strecken

zusätzlich:

- keine Überlappungen von horizontalen Strecken und von vertikalen Strecken
- keine reinen Berührungen von Strecken

Algorithmus: Schnittbestimmung isoorientierter Strecken mit Sweep-Line

begin

X-Ordnung = sortierte Folge der x-Ereignisse;
Y-Ordnung = $\emptyset$;
while X-Ordnung nicht leer **do begin**
(* 1 *)
x_{akt} =kleinster x-Wert aus X-Ordnung;
Lösche kleinsten x-Wert aus X-Ordnung;
case x_{akt}
(* 1a *)
ist linker Endpunkt einer horizontalen Strecke h_i :
Füge h_i in Y-Ordnung ein;
(* 1b *)
ist rechter Endpunkt einer horizontalen Strecke h_i :
Lösche h_i aus Y-Ordnung;
(* 1c *)
ist x-Wert einer verikalen Strecke v_j:

Bestimme alle Schnittpunkte von v_j mit den h_i in der Y-Ordnung und füge die korrespondierenden sich schneidenden Streckenpaare in S ein;

end;

end;

(* 2 *)

end;

Zeitkomplexität:

Vorverarbeitung	
Elemente einlesen:	$O(n)$
Y-Ordnung initialisieren:	$O(1)$
X-Ordnung aufbauen und sortieren:	$O(n \log n)$
Arbeitsschleife	
jeweils kleinsten x-Wert auslesen:	$O(n)$
n Strecken in Y-Ordnung einordnen und herausnehmen (z.B. mit AVL-Baum)	$O(n \log n)$
n Anfragen an Y-Ordnung und Ausgabe aller k Schnittpaare: (Hierbei ist $k = \sum_i k_i$ die Gesamtanzahl der gefundenen Schnittpunkte, pro v_i ensteht bei der Intervallanfrage an die Y-Ordnung $T_{max}(n) = O(k_i + \log n)$)	$O(k + n \log n)$
Insgesamt:	$O(k + n \log n)$

$\rightarrow$ Parameter k nimmt Werte zwischen 0 und $O(n^2)$ an

$\Rightarrow$ kann nicht vom Term $O(n \log n)$ geschluckt werden

Satz

Die Bestimmung aller Schnittpunkte isoorientierter Strecken im $\mathbb{R}^2$benötigt einen Zeitaufwand von $T_{max}(n) = O(k + n \log n)$, und das ist optimal.

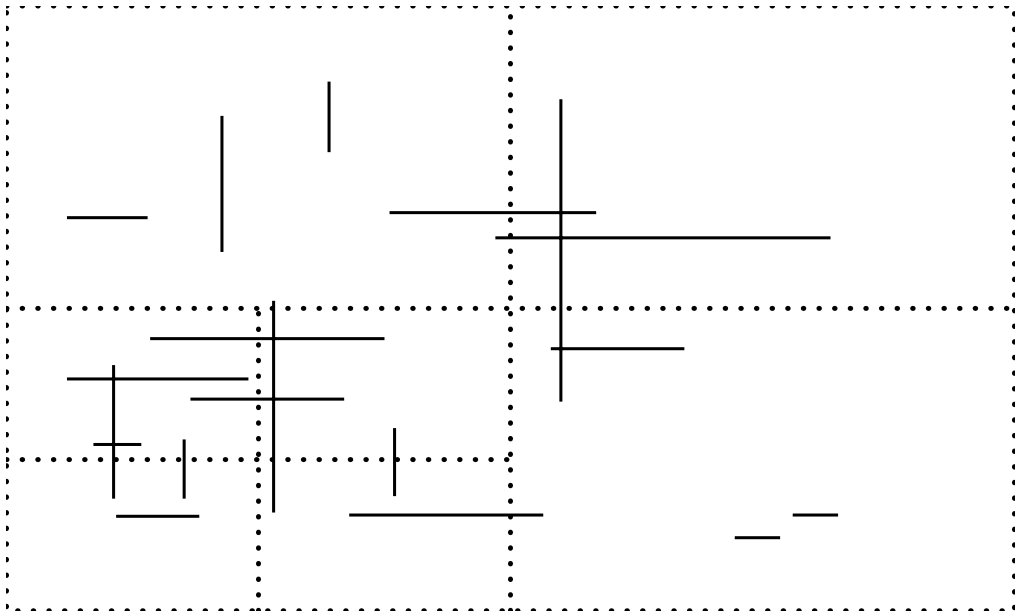
Beweis

Wird der Algorithmus so implementiert, daß er auch Berührungen von degenerierten, auf einen Punkt reduzierten Linien korrekt meldet, so kann man mit ihm das Problem der Elementeindeutigkeit von n-Werten $\{x_1, ..., x_n\}$ lösen.

Dazu wird das Problem durch Umwandeln der x_i mit linearem Zeitaufwand in degenerierte Strecken $(x_i, 0)$ umgeformt. Der Algorithmus kann diese Strecken auf Gleichheit testen und damit die Elementeindeutigkeit feststellen. Könnte er es mit geringerem Aufwand als $T_{max}(n) = O(n \log n)$, so wäre auch die Elementeindeutigkeit schneller gelöst. Der Teilterm k ist optimal, weil er genau den Reporting-Aufwand des Algorithmus angibt, also auf keinen Fall unterboten werden kann.

Lösung mit Teilen & Herrschen

$\longrightarrow$ zerlege Szene rekursiv, löse Problem in Teilbereichen



Algorithmenschema

1. zerlege Datenmenge rekursiv jeweils in zwei bzw. vier Teilszenen
   $\rightarrow$ hierbei: teile an Koordinatenmitte oder am Median-Element
   (Einzelne Strecken evtl. in kürzere Stücke zerlegen)
2. Teilszene wird dann nicht weiter zerlegt, wenn sich kein Rechenzeitvorteil mehr ergibt

   in Teilszene: einfacher Schnittalgorithmus ($T(n) = O(n^2)$)

$\rightarrow$ je nach Implementierung des Schnittalgorithmus:
unterschiedliche Zellgröße

Schnitt isoorientierter Strecken mit Teilen & Herrschen

begin

Zerlege Szene S in Unterszenen S_1, S_2, S_3, S_4 mit n_1, n_2, n_3, n_4 Strecken;

(* $n \leq n_1 + n_2 + n_3 + n_4 \leq 2n$ *)

if $(c_0 \cdot n^2 \leq c_o(n_1^2 + n_2^2 + n_3^2 + n_4^2) + c_1(n_1 + n_2 + n_3 + n_4))$

then Berechne Schnittpunkte in S direkt;

else

begin

Divide (S_1);

Divide (S_2);

Divide (S_3);

Divide (S_4);

end;

end;

Komplexitätsanalyse

$\rightarrow$ teile Zellen an Medianelement v_i bzgl. x-Koordinate bzw. h_i bzgl. y-Koordinte

$\rightarrow$ Suche Medianelement: $T_{max}(n_i) = O(n_i)$

$\rightarrow$ Abbruchkriterium: $n_i \leq 8 \Rightarrow$ Schnittalgorithmus: $O(1)$

Rekkurenzrelation:

günstigster Fall:	$T(n) = 4 \cdot T(\frac{n}{4}) + c \cdot n$	$T_{max}(n) = O(n \log n)$
ungünstigster Fall:	$T(n) = 4 \cdot T(\frac{n}{2}) + c \cdot n$	$T_{max}(n) = O(n^2)$

$\rightarrow$ Rechenzeit szenenabhängig

$\rightarrow$ Term k fällt weg, weil Reporting-Aufwand in Rekkurenz-relation eingeht.

Lösung mit Zellraster

für n Objekte O_i in der Ebene mit Koordinaten $(x_{i1}, y_{i1}), ..., (x_{ik}, y_{ik})$

Algorithmenschema: Zellrasterinitialisierung

(1) Bestimme die Werte $x_{min}, x_{max}, y_{min}, y_{max}$ über alle beteiligten Punkte.

(2) Definiere ein äquidistantes Zellraster der Größe $\sqrt{n} \times \sqrt{n}$, als Array $[0 \ldots m, 0 \ldots m]\ of\ Zelle$, $(m = \lfloor \sqrt{n} \rfloor)$.
Ein Punkt (x, y) liegt in der Zelle mit dem Index $[I(x), I(y)]$, wobei für $I(x)$ gilt: $I(x) := \left\lfloor \frac{x - x_{min}}{x_{max} - x_{min}} \cdot m \right\rfloor$, $I(y)$ analog.

(3) Lege in der Datenstruktur für jede Zelle eine Liste derjenigen Objekte O_i an, die die Zelle „geometrisch“ berühren oder schneiden.

Komplexitätsanalyse

$\longrightarrow$ Bildung Bounding-Box (1): $O(n)$

$\longrightarrow$ Initialisierung Zellraster-Datenstruktur (2):
$T_{max}(n) = O(\sqrt{n} \times \sqrt{n}) = O(n)$.

$\longrightarrow$ Rechenzeitaufwand für (3) ist szenenabhängig

Objekte gross

$\Rightarrow$ einzelne Objekte schneiden in einer Richtung
bis zu $O(\sqrt{n})$ Zellen

$\Rightarrow T_{max}(n) = O(n \cdot \sqrt{n})$

$\longrightarrow$ überlineares Wachstum in Vorverarbeitung unerwünscht

$\longrightarrow$ in gewissen Grenzen: Linearisierung durch Vergröberung der Zellen möglich

$\longrightarrow$ Bestimme Breite der Objekte (x-Spanne):

$$x_{spanne} := \frac{1}{n} \left(\sum_i |x_{i_{max}} - x_{i_{min}}| \right) .$$

$\longrightarrow$ rastere so, daß Zellen etwa x_{spanne} breit sind:

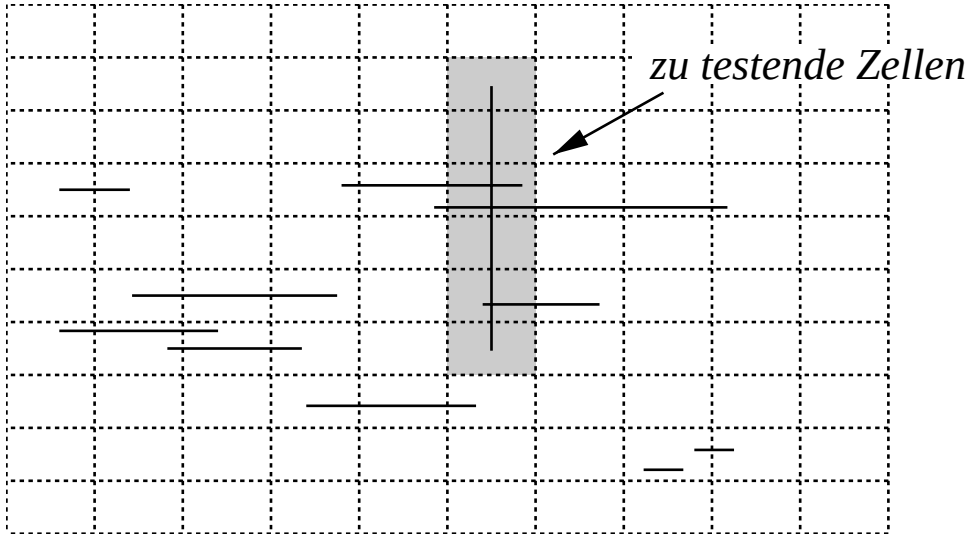
$$m_x \cong \min \left\{ \sqrt{n}, \frac{(x_{max} - x_{min})}{x_{spanne}} \right\} .$$

$\longrightarrow$ m_y analog

Lösung isoorientierter Streckenschnitt mit Zellraster

$\rightarrow$ bisher $T_{max}(n) = O(k + n \log n)$ mit $k \in [O(n), O(n^2)]$

$\rightarrow$ jetzt: bei Gleichverteilung $O(n)$



Algorithmus zur Schnittbestimmung ($n = n_v + n_h$):

(1) (* Vorverarbeitung *)

Lege Zellraster Z der Größe $\lfloor\sqrt{n}\rfloor \times \lfloor\sqrt{n}\rfloor$ über die Daten;

Trage horizontale Strecken $h_1, h_2, \ldots h_{n_h}$ in diejenigen

Zellen ein, die sie berühren;

$M = v_1, v_2, \ldots, v_{n_v}$;

(2) **while** $M \neq \emptyset$ **do begin**

Entferne ein Element v aus M und bestimme alle Z_i,

die von v berührt werden;

Schneide v mit allen h_j, die in diesen Zellen vermerkt sind;

end;

Komplexitätsanalyse

Vorverarbeitung: $O(n)$

bei Gleichverteilung der Endpunkte der Strecken $[0, 1]$:

mittlere Länge:	l	$=$	$\frac{1}{2}$
Zellen pro Strecke:	n_s	$=$	$\frac{\sqrt{n}}{2}$
Strecken pro Zelle:	n_z	$=$	$\frac{\sqrt{n}}{2}$

$\rightarrow$ Anzahl Schnittoperationen pro Strecke v

$$n_z \cdot n_s = \frac{n}{4} = O(n)$$

$\rightarrow$ insgesamt $O(n^2)$ Schnittoperationen

Rastergünstigen Szenen

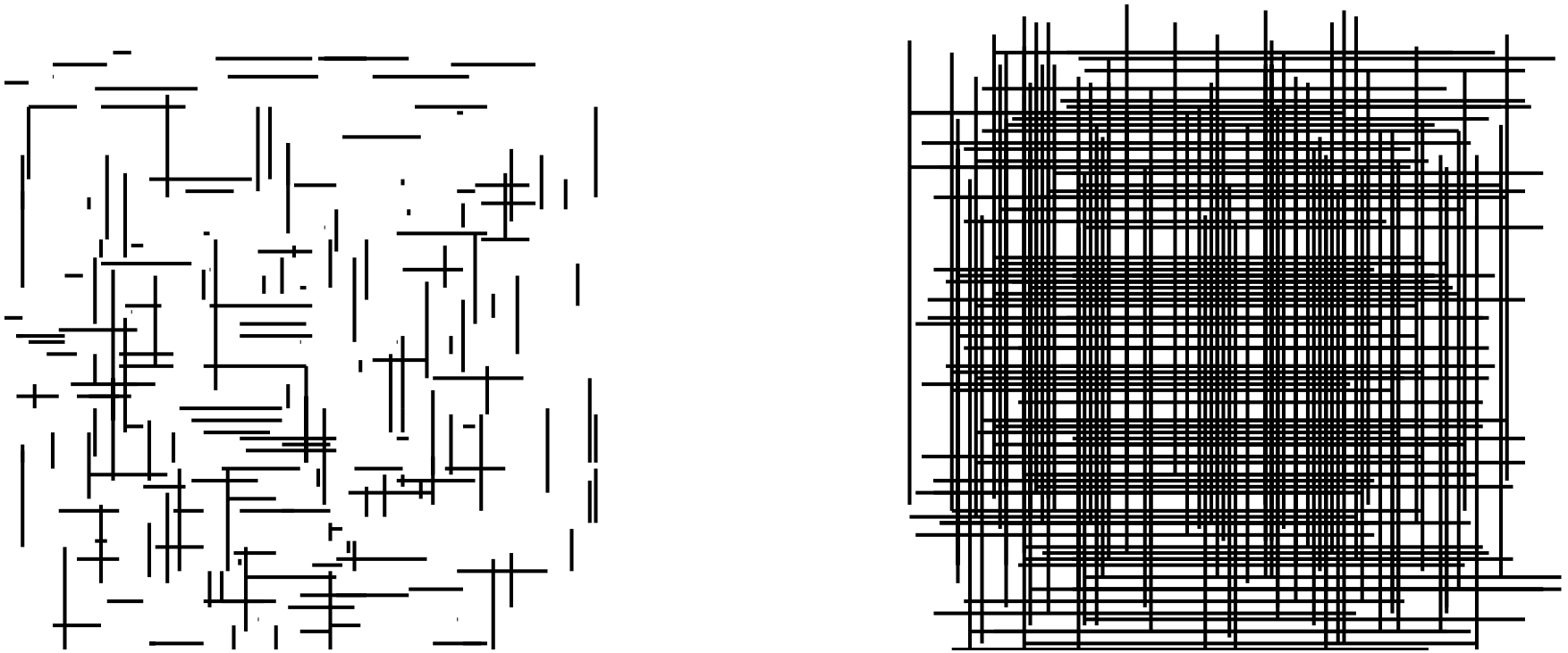
Definition: Rasterkomplexität einer Szene

Eine Szene S hat eine Rasterkomplexität $K \in \mathbb{N}$, wenn beim Eintragen in ein $\lfloor\sqrt{n}\rfloor \times \lfloor\sqrt{n}\rfloor$-Raster höchstens K Teilobjekte von S in eine Zelle des Rasters einzutragen sind.

Definition: K_0-Rastergünstige Szene

Eine Szene gehört zur Klasse der K_0-rastergünstigen Szenen, wenn für ihre Rasterkomplexität K gilt: $K \leq K_0$.

$\rightarrow$ Schnitt für rastergünstige Szenen: $T_{max}(n) = O(n)$.



Szene: (a) rastergünstig; (b) rasterungünstig.

Komplexität Schnittbestimmung mit Zellraster

	Normale Szene	Rastergünstig
Vorverarbeitung:	$O(n) + O(n_v \cdot \sqrt{n})$	$O(n) + O(K_0 \cdot n_v)$
Arbeitsschleife:	$O(n^2)$	$O(K_0^2 \cdot n)$
Insgesamt:	$O(n^2)$	$O(n)$

$\longrightarrow$ stärkere Abbhängigkeit des Algorithmus von den Eingabedaten als andere Verfahren

$\longrightarrow$ weiterer Vorteil: einfach zu implementieren

Schnitt beliebiger Strecken

Gegeben

n Strecken bzw. Segmente $S = \{s_1, s_2, .., s_n\}$ in der Ebene, jedes s_i ist durch seine Eckpunkte gegeben.

Gesucht

a) Feststellung, ob sich zwei der Segmente schneiden
b) Alle schneidenden Paare (s_i, s_j)

$\rightarrow$ Brute-Force: Aufgabenstellung a): $T_{max}(n) = O(n^2)$

$\rightarrow$ Shamos und Hoey (1976), Sweep-Line: $T_{max}(n) = O(n \log n)$

$\rightarrow$ Bentley und Ottmann (1979) für b): $T_{max}(n) = O((n + k) \log n)$

Der Bentley-Ottmann Algorithmus

$\longrightarrow$ nun Y-Ordnung nicht mehr statisch:

Gilt für die Strecken S_1, S_2 in der y-Ordnung:
S_1 oberhalb S_2, so gilt nach einem gemeinsamen Schnitt:S_2 oberhalb S_1

$\longrightarrow$ Haltepunkte Sweep-Line nun zusätzlich an Schnittpunkten

Benötigte Datenstrukturen sind ($n \in \mathbb{N}$: Anzahl Strecken):

X-Ordnung (Q) als Prioritäts-Warteschlange oder AVL-Baum

$insert(s, Q)$	$O(\log n)$
$min_x(Q)$	$O(1)$
$min_x remove(Q)$	$O(\log n)$

Y-Ordnung (L) als balancierter AVL-Baum

$insert(s, L)$	$O(\log n)$
$delete(s, L)$	$O(\log n)$
$pred_y(s, L)$	$O(\log n)$
$succ_y(s, L)$	$O(\log n)$
$swap(s_1, s_2, L)$	$O(\log n)$

Schnittbestimmung allgemeiner Strecken mit Sweep-Line

begin

$Q :=$ Alle Anfangs- und Endpunkte, geordnet nach x-Koordinate;

$L := \emptyset$;

while Q nicht leer **do begin**

$p := min_x(Q)$;

$min_x remove(Q)$;

if p linker Endpunkt eines Segments s

then begin

$insert(s, L)$;

$\hat{s} := succ_y(s, L)$;

$\tilde{s} := pred_y(s, L)$;

if $s \cap \hat{s} \neq \emptyset$ **then** $insert(s \cap \hat{s}, Q)$;

if $s \cap \tilde{s} \neq \emptyset$ **then** $insert(s \cap \tilde{s}, Q)$;

end else begin

if p rechter Endpunkt eines Segments s

then begin

$\hat{s} := succ_y(s, L)$;

$\tilde{s} := pred_y(s, L)$;

if $\hat{s} \cap \tilde{s} \neq \emptyset$

then $insert(\hat{s} \cap \tilde{s}, Q)$;

$delete(s, L)$;

end;

else (* p ist Schnittpunkt von $\hat{s}$ und $\tilde{s}$ *)

begin

Ausgeben von p;

$swap(\hat{s}, \tilde{s}, L)$; (* Vertausche $\hat{s}$ und $\tilde{s}$ in L*)

$\tilde{t} := pred_y(\tilde{s}, L)$;

if $\tilde{s} \cap \tilde{t} \neq \emptyset$

then $insert(\tilde{s} \cap \tilde{t}, Q)$;

$\hat{t} := succ_y(\hat{s}, L)$;

if $\hat{s} \cap \hat{t} \neq \emptyset$

then $insert(\hat{s} \cap \hat{t}, Q)$;

end;

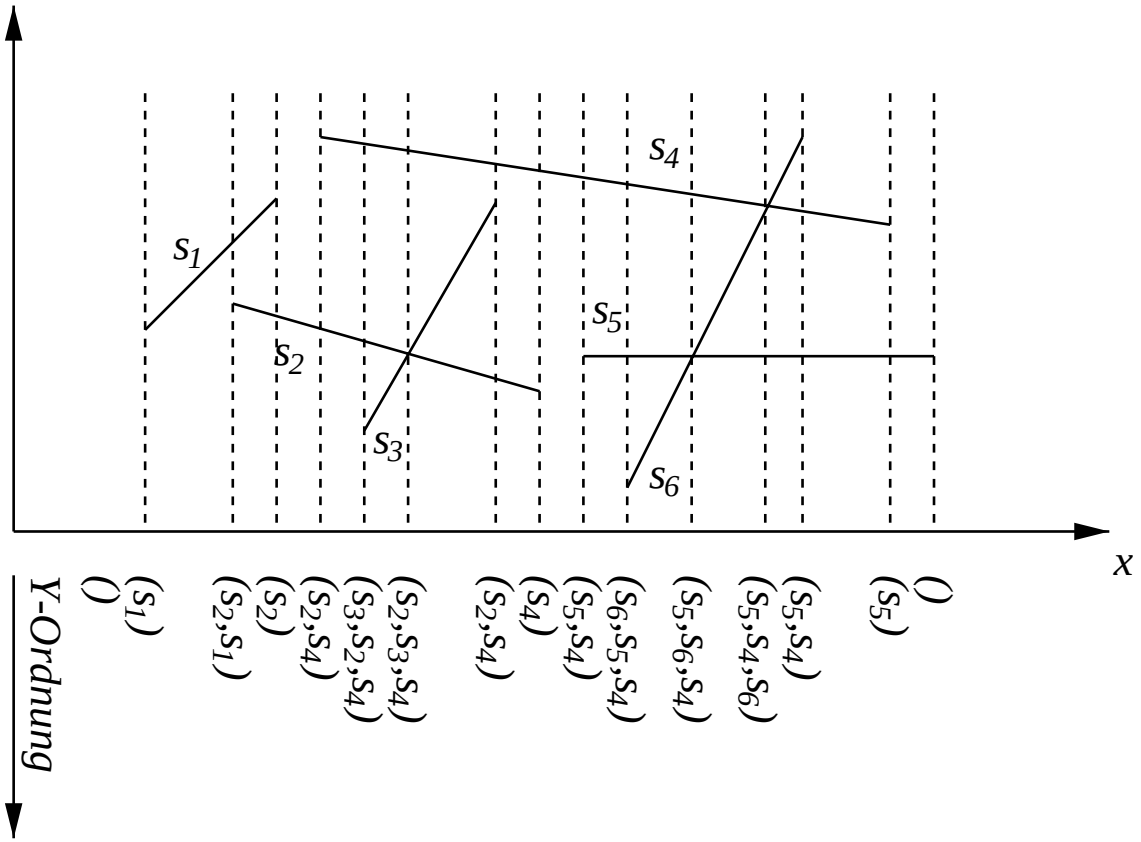
end;

end; (* while *)

end

$\rightarrow$ alle Operationen: $O(\log n)$

$\rightarrow$ y-Koordinaten der Strecken können nicht mehr als Schlüssel verwendet werden, daher Geradengleichungen $(a \cdot x + b)$ als Schlüssel



Bentley-Ottmann Algorithmus (unten Y-Ordnung)

Komplexitätsanalyse

Arbeitsschleife	
jeweils kleinsten x-Wert auslesen	
und löschen:	$O((n+k)\log(n+k))$
n Strecken in Y-Ordnung einordnen	
und herausnehmen (z.B. mit AVL-Baum)	$O(n\log(n))$
k Strecken in Y-Ordnung vertauschen	$O(k\log(n))$
k Schnittpunkte in X-Ordnung einfügen	$O(k\log(n+k))$
Insgesamt:	$O((n+k)\log(n+k))$

$\rightarrow k = O(n^2)$ daher $\log(n+k) = O(\log n)$

Schnitt Gerade mit Polygon

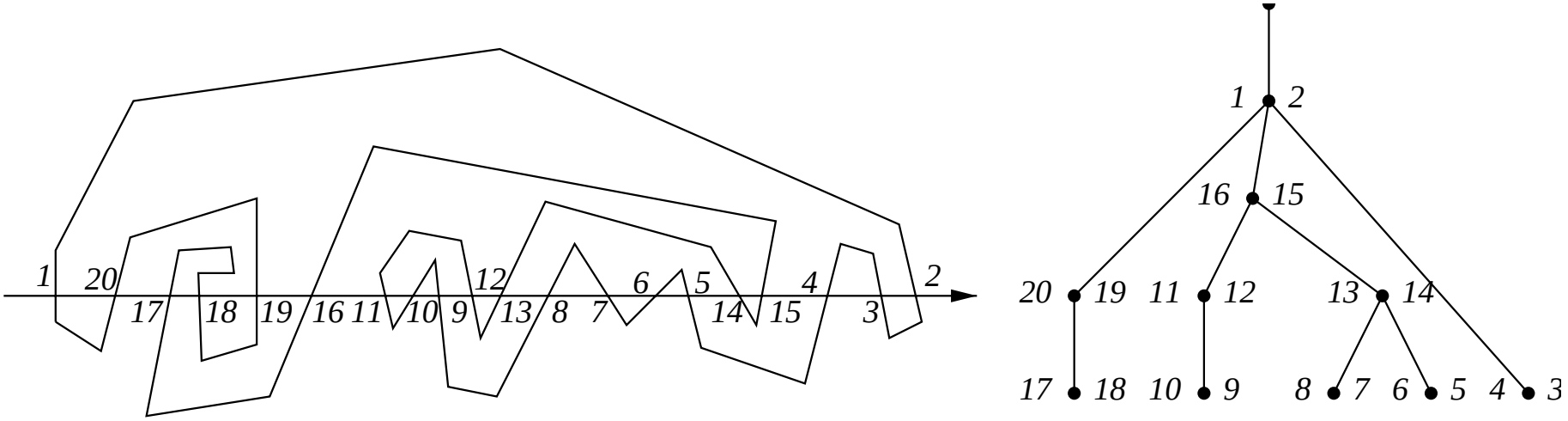
$\rightarrow$ über Bentley-Ottmann Algorithmus: $T_{max}(n) = O(n \log n)$

$\rightarrow$ nichtlinearer Aufwand durch Sortierung der Kanten

$\rightarrow$ in einem einfachen Polygon: implizite Sortierung, da sich Polygonkanten nicht schneiden dürfen (geschlossene Jordankurve)

$\Rightarrow$ Spezialfall des Schnittproblems, kann effizienter gelöst werden

$\rightarrow$ Hoffman, Mehlhorn et al. (1986): $T_{max}(n) = O(n)$ (bei sortierter Schnittpunktausgabe)

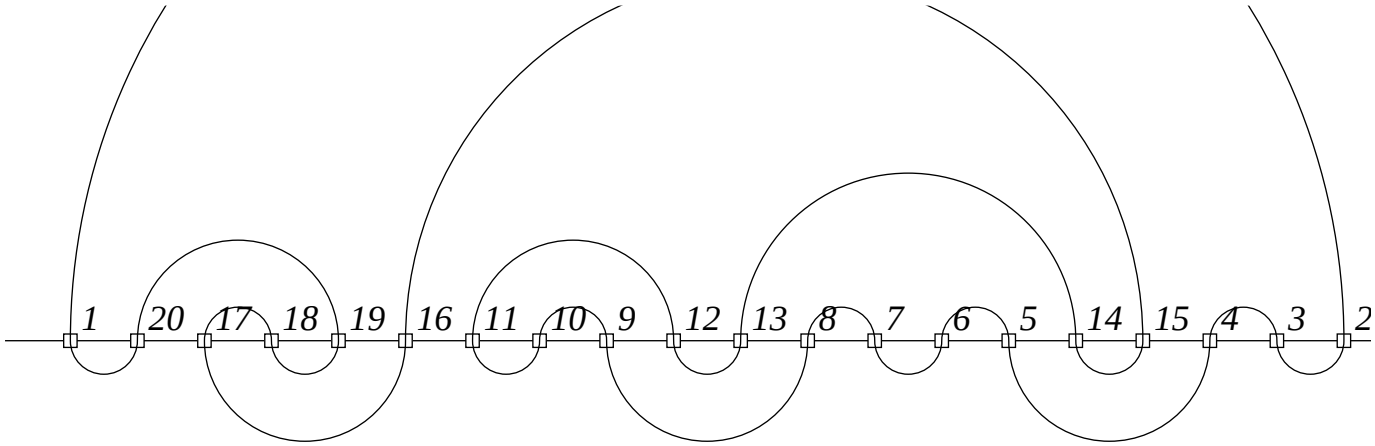


Schnitt von Gerade und Polygon; (b) zugehörige Baumstruktur

$\longrightarrow$ erstens: Polygonkanten durchlaufen und Schnittpunkte $S = \{s_1, s_2, .., s_k\}(k \leq n)$ in Durchlauf-Reihenfolge ermittelt $\Rightarrow T_{max}(n) = O(n)$.

$\longrightarrow$ zwischen Schnittpunkten verläuft Polygon ganz über oder ganz unter x-Achse

$\longrightarrow$ topologisch äquivalent zu Halbkreisen ober- und unterhalb der x-Achse



$\longrightarrow$ baue Baum auf: Knoten Schnittpunktpaare, die durch über der Schnittgerade liegende Polygonteile direkt verbunden sind.

$\longrightarrow$ Vaterknoten: alle Kinder bezeichnen Halbkriese, die innerhalb des Halbkreises des Vaters liegen

$\longrightarrow$ durchlaufen des Baumes in Tiefensuche (Ausgabe linker Punkt, Ausgabe Kinder, Ausgabe rechter Punkt) liefert Sortierreihenfolge