Notizen zu Algorithmen II

Jens Ochsenmeier

10. Februar 2018

Inhaltsverzeichnis

1

Geometrische Algorithmen

Inhalt dieses Kapitels:

- Plane-Sweep-Algorithmus
- Konvexe Hülle
- Kleinste einschließende Kugel
- Range Search

1.1 Grundlegende Definitionen

Wir nennen $p \in \mathbb{R}^d$ einen **Punkt**. $p.i$ stelle die i-te Komponente von p dar. Für $d \in \{2, 3\}$ schreiben wir $p.x$, $p.y$, $p.z$ statt $p.1$, $p.2$, $p.3$.
Für zwei Punkte a, b definieren wir

$$\overline{ab} := \{\alpha \cdot a + (1 - \alpha) \cdot b : \alpha \in [0, 1]\}$$

als das **Segment** zwischen a und b.
Ein **Polygon** ist eine Menge an Segmenten, gegeben als Punktemenge $P = p_1, \ldots, p_n$ mit $p_i \in \mathbb{R}^d$, $p_n = p_1$. $\overline{p_i, p_{i+1}}$ für $i = 1, \ldots, n - 1$ ist der **Umriss** des Polygons.
Ist für alle $a, b \in P$ auch $\overline{ab} \in P$, so nennen wir P **konvex**.

1.2 Streckenschnitte

Bei diesem Problem sind n Strecken $S = \{s_1, \dots, s_n\}$ gegeben und wir wollen alle Schnittpunkte dieser, also $\bigcup_{s,t \in S} s \cap t$ berechnen.
Naiv lassen sich diese Streckenschnitte in $O(n^2)$ berechnen:

```
foreach {s, t} ⊆ S do
  if s ∩ t ≠ ∅ then output {s, t}
```

Dieser Algorithmus ist für große Datenmengen offensichtlich zu langsam.
Idee ist nun, dass eine (waagerechte) **Sweep-Line** von oben nach unten läuft. Dabei speichern wir Segmente, die l schneiden, und finden deren Schnittpunkte. Invariante ist, dass Schnittpunkte oberhalb von l korrekt ausgegeben wurden.

Orthogonale Streckenschnitte

Zuerst betrachten wir die Vereinfachung, dass nur orthogonale Segmente (also parallel zur x- oder y-Achse existieren).

```
T := ⟨⟩ SortedSequence of Segment
invariant T stores vertical segments intersecting l
Q := sort(⟨(y, s) : ∃ hor-seg s at y ∨ ∃ ver-seg s starting/ending at y⟩)
foreach (y, s) ∈ Q in descending order do
  if   s is ver-seg and starts at y then T.insert(s)
  elif s is ver-seg and ends at y then T.remove(s)
  else   // horizontal segment s = (x1, y)(x2, y)
    foreach t = (x, y1)(x, y2) ∈ T with x ∈ [x1, x2] do output {s, t}
```

Hier sind T und Q die einzigen komplexen Datenstrukturen, die wir benötigen, also sortierte Listen an Segmenten (T geordnet nach x-Wert, Q nach y).
`insert` und `remove` gehen in $O(\log n)$, die `rangeQuery` für ein Segment in $O(\log n + k_s)$ (bei k_s Schritten mit horizontalem Segment s). Insgesamt haben wir also

$$O(n \log n + \sum_s k_s) = O(n \log n + k).$$

Verallgemeinerung

Wir verallgemeinern jetzt den Spezialfall von oben, verwenden allerdings folgende Vereinfachungen: Es gebe keine horizontalen Segmente und Überschneidungen sind

immer nur zwischen zwei Segmenten, nicht mehr. Außerdem soll es keine Überlappungen geben, die Anzahl an Schnitten zwischen zwei Segmenten ist also immer entweder 0 oder 1.

Wir verwenden wieder T als nach x geordnete Liste der Strecken, die l schneidet. Außerdem verwenden wir *Ereignisse* — diese sind Änderungen von T, also das Starten und Enden von Segmenten sowie Schnittpunkte.
Einen Schnittest müssen wir nur dann durchführen, wenn zwei Segmente an einem Ereignispunkt in T benachbart sind.

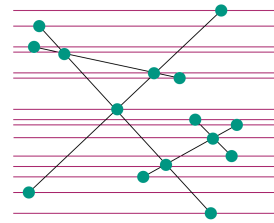

Abbildung 1.1. Die grünen Punkte stellen die Ereignisse dar. Außerdem ist l zum Zeitpunkt der Ereignisse dargestellt.

Zur Implementierung brauchen wir nun einige Zusatzmethoden:

findNewEvent ermittelt, ob es einen Schnitt zwischen zwei Segmenten s und t gibt.

```
FINDNEWEVENT(s, t)
if s and t cross at y' < y then
  Q.insert((y', intersection, (s, t)))
```

Die Event-Handler werden kümmern sich um die Handhabung der drei möglichen Event-Types.

```
HANDLEEVENT(y, intersection, (a, b), T, Q)
output(s ∩ t)
T.swap(a, b)
prev := pred(b)
next := succ(a)
findNewEvent(prev, b)
findNewEvent(a, next)
```

```
HANDLEEVENT(y, start, s, T, Q)
h := T.insert(s)
prev := pred(h)
next := succ(h)
findNewEvent(prev, h)
findNewEvent(h, next)
```

```
HANDLEEVENT(y, finish, s, T, Q)
h := T.locate(s)
prev := pred(h)
next := succ(h)
T.remove(s)
findNewEvent(prev, next)
```

Nun können wir den Algorithmus implementieren.

```
T := ⟨⟩ SortedSequence of Segment
invariant T stores relative order of segments intersecting l
Q := MaxPriorityQueue
Q := Q ∪ {(max{y, y'}, start, s) : s = (x, y)(x', y') ∈ S}
Q := Q ∪ {(min{y, y'}, finish, s) : s = (x, y)(x', y') ∈ S}
while Q ≠ ∅ do
  (y, type, s) := Q.deleteMax
  handleEvent(y, type, s, T, Q)
```

Dieser Algorithmus benötigt $O(n \log n)$ zur Initialisierung und $O((n + k) \log n)$ für die Event-Schleife, insgesamt also $O((n + k) \log n)$.

1.3 Konvexe Hülle

Wir werden uns in diesem Abschnitt mit dem folgenden Problem beschäftigen: Gegeben sei eine Punktmenge $P = \{p_1, \ldots, p_n\} \subset R^2$. Gesucht ist ein konvexes Polygon C mit Eckpunkten $\in P$, sodass alle Punkte von P in C liegen.
Zuerst sortieren wir P lexikographisch. Das bedeutet, dass

$$p > q \iff p.x > q.x \vee (p.x = q.x \wedge p.y > q.y)\,.$$

Wir berechnen ohne Einschränkung nur die obere Hülle, also die Hülle um die Punkte oberhalb von $\overline{p_1 p_n}$.

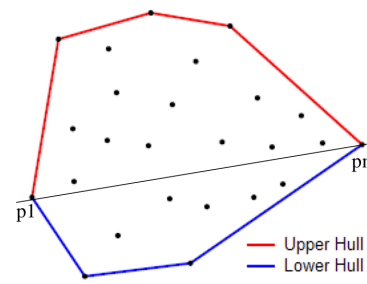

Abbildung 1.2. Obere Hülle.

Wir können beobachten, dass die obere Hülle ausschließlich Abbiegungen nach rechts macht (und die untere nur Abbiegungen nach links). Um damit arbeiten zu können müssen wir Abbiegungen definieren:

Definition 1.3.1 (Abbiegung). Für eine Punktemenge $P = \{p_1, \ldots, p_n\}$ ist eine **Abbiegung nach rechts** an Stelle i vorhanden, falls p_{i+1} rechts von $\overline{p_{i-1} p_i}$ liegt.

Das konstruieren der oberen Hülle nennt sich auch **Graham's Scan**.[1]

```
UPPERHULL(p_1, ..., p_n)
L ≔ ⟨p_n, p_1, p_2⟩: Stack of Point
invariant L is upper hull of ⟨p_n, p_1, ..., p_i⟩
for i ≔ 3 to n do
  while ¬rightTurn(L.secondButlast, L.last, p_i) do L.pop
  L ≔ L ∘ ⟨p_i⟩
return L
```

Der Algorithmus selbst läuft in $O(n)$, weswegen das Sortieren dominiert und das ganze in $O(n \log n)$ liegt.

[1] Graham 1972, Andrew 1979