Übungsblatt 3

Julius Auer, Alexa Schlegel

Aufgabe 1 (Graham-Scan):

Im Folgenden wird der implementierte Algorithmus mit Hilfe von Pseudocode beschrieben. Dieser orientiert sich an der in *Introduction to Algorithms*[1] vorgestellten Lösung. Der Stack S wird verwendet, um mögliche Kandidaten der konvexen Hülle zu verwalten. Nach terminieren des Algorithmus enthält S die Knoten der konvexen Hülle entgegen dem Uhrzeigersinn.

Algorithm 1 Graham-Scan(Q)

1: sei $p_0 \in Q$ mit minimaler x-Koordinate:
bei gleicher x-Koordinate wird minimale y-Koordinate verwendet
2: sei $p_1, p_2, \ldots, p_n$ die restlichen Punkte aus Q:
von p_0 aus gesehen in Polarkorrdinaten gegen den Urzeigersinn sortiert
bei gleichem Winkel, nach Abstand von p_0 sortiert
3: PUSH(p_0, S)
4: PUSH(p_1, S)
5: PUSH(p_2, S)
6: **for** $i = 3$ **to** n **do**
7: **while** Winkel (NEXT-TO-TOP(S), TOP(S), p_i) ist keine Linkskurve **do**
8: POP(S)
9: **end while**
10: PUSH(p_i, S)
11: **end for**
12: **return** S

Die Methode *grahamScan(points)* erhält als Eingabe eine Punktwolke und liefert ein Polygon zurück, welches der konvexen Hülle dieser Punktwolke entspricht.

Wir verwenden als Startpunkt des Algorithmus den Punkt der am weitesten links liegt und die kleinste y-Koordinate besitzt (in der Vorlesung wird ein beliebiger Punkt in der Mitte der Punktwolke ermittelt). Die Sortierung der Punkte erfolgt nach ihrer Steigung.

Um Links- und Rechtskurven zu prüfen, wird die Determinante einer 2×2-Matrix bestimmt, welche sich aus den zwei Vektoren der folgenden drei Punkte zusammensetzt: $p2$ ist der letzte Punkt auf dem Stack, $p3$ der vorletzte Punkt und $p1$ der Punkt der aktuell betrachtet wird. Es wird getestet ob $\overline{p2\ p1}$ links oder rechts von $\overline{p2\ p3}$ liegt.

```
1   /**
2    * Compute the convex hull for a set of points using the
3    * graham-scan-algorithm.
4    *
5    * @param frame a Frame for drawing, or null if no debug drawings are needed
6    * @param pointcloud a set of points
7    * @return the polygonial chain forming a convex hull for the set of points
8    */
9   public static Polygon grahamScan(Frame frame, Point... pointcloud) {
10      // copy points to list and find minimum
11      Point m = pointcloud[0];
12      LinkedList<Point> sortedPoints = new LinkedList<Point>();
```

[1] Introduction to Algorithms, Second Edition, S. 949

```
13
14          for(Point p : pointcloud) {
15              sortedPoints.add(p);
16
17              if(p.compareTo(m) < 0)
18                  m = p;
19          }
20
21          final Point min = m;
22          sortedPoints.remove(min);
23
24          // sort by ascent first and distance second
25          Collections.sort(sortedPoints, new Comparator<Point>() {
26
27              @Override
28              public int compare(Point p1, Point p2) {
29                  Vector pos1 = p1.toPosition().substract(min.toPosition());
30                  Vector pos2 = p2.toPosition().substract(min.toPosition());
31                  double ascent1 = pos1.getAscent();
32                  double ascent2 = pos2.getAscent();
33
34                  if(Math.abs(ascent1 - ascent2) < C.E) {
35                      double l = pos1.length() - pos2.length();
36                      return (int) (l < 0 ? Math.floor(l) : Math.ceil(l));
37                  }
38
39                  double d = ascent1 - ascent2;
40                  return (int) (d < 0 ? Math.floor(d) : Math.ceil(d));
41              }
42          });
43
44          // put first 3 points on stack
45          Stack<Point> stack = new Stack<Point>();
46          stack.push(min);
47          stack.push(sortedPoints.removeFirst());
48          stack.push(sortedPoints.removeFirst());
49
50          for(Point p1 : sortedPoints) {
51              Point p2 = stack.pop();
52              Point p3 = stack.peek();
53              stack.push(p2);
54
55              // use determinant for testing left/right turns
56              Vector v1 = p3.toPosition().substract(p2.toPosition());
57              Vector v2 = p1.toPosition().substract(p2.toPosition());
58              Mat2x2 matrix = new Mat2x2(v1.get(0), v1.get(1), v2.get(0), v2.get(1));
59
60              // testing for not left-turns
61              while(matrix.getDeterminant() > 0) {
62                  // remove last point
63                  stack.pop();
64
65                  // have a look at the next point
66                  p2 = stack.pop();
67                  p3 = stack.peek();
68                  stack.push(p2);
69                  v1 = p3.toPosition().substract(p2.toPosition());
70                  v2 = p1.toPosition().substract(p2.toPosition());
71                  matrix = new Mat2x2(v1.get(0), v1.get(1), v2.get(0), v2.get(1));
72              }
73
74              // put current point on stack
75              stack.push(p1);
76          }
77
78          // generate polygon from points on stack
79          Point[] points = new Point[stack.size()];
80          stack.toArray(points);
81          return new Polygon(points);
82      }
```

Zur Veranschaulichung des Algorithmus gibt es einen Testklasse, wo die einzelnen Schritte visualisiert werden. Hier ein Beispiel mit 10 zufälligen Punkten:

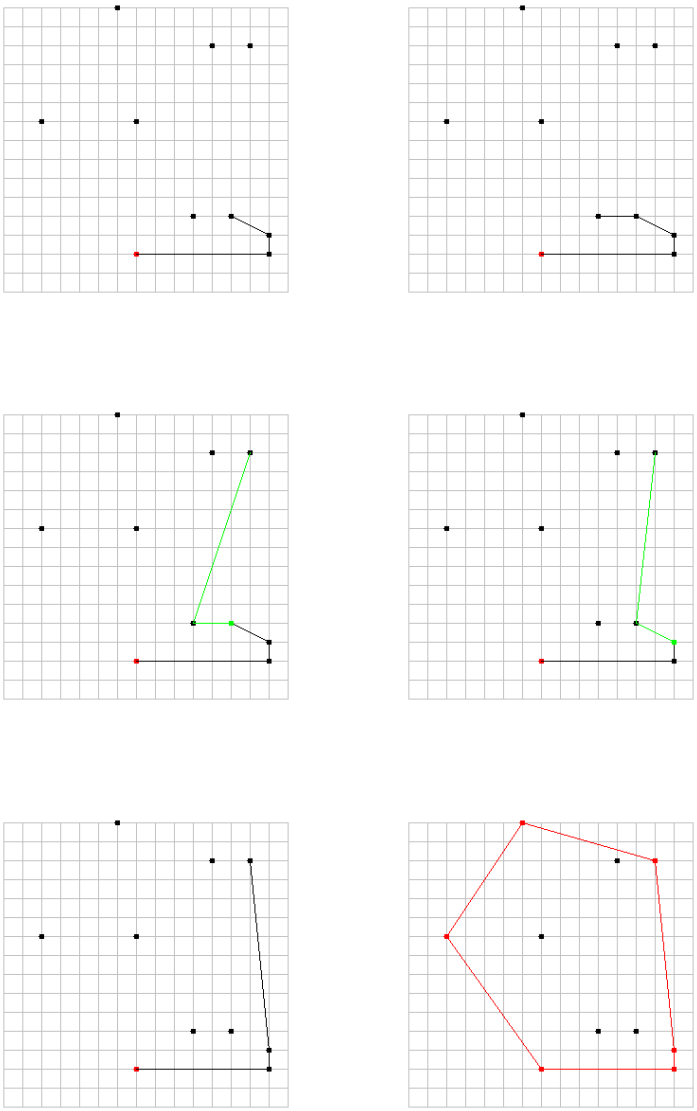

Abbildung 1: Die ersten 5 Schritte und die resultierende konvexe Hülle.

Aufgabe 2 (inkrementelle Konstruktion der konvexen Hülle):

Die ersten drei Punkte $S = p_1, p_2, p_3$ werden zunächst nach X-Koordinate aufsteigend und dann nach Y-Koordinate absteigend sortiert. Die konvexe Hülle $CH(S) = S$ ist somit ein Polygonzug der im Uhrzeigersinn verläuft. Ferner wird das Center of Mass $com = ((x_1 + x_2 + x_3)/3, (y_1 + y_2 + y_3)/3)$ bestimmt. Außerdem wird als Datenstruktur ein binärer Suchbaum T angelegt, der $search(p)$, $insert(p)$ und $delete(p)$ in $O(\log n)$ garantiert (z.B. ein AVL-Baum) - in diesen werden p_1, p_2, p_3 eingefügt, wobei die totale Ordnung gegeben ist durch

$$p_i < p_j \text{ g.d.w. } \frac{y_i - y_{com}}{x_i - x_{com}} < \frac{y_j - y_{com}}{x_j - x_{com}}$$

wenn also die Steigung zwischen com und p_i kleiner ist, als die von com und p_j. Diese Ordnung

gilt auch für zukünftige Operationen auf dem Baum. $search(p)$ soll hierbei den Punkt aus $CH(S)$ liefern, der die nächst kleinere Steigung im Vergleich zu p hat. Diese Initialisierung benötigt offensichtlich nur $O(1)$ Zeit.

Für jeden weiteren Punkt p_n um den die konvexe Hülle $S = p_1, ..., p_{n-1}$ erweitert werden soll wird nun $CH(S, p_n, T)$ aufgerufen, wobei $left(p, \overrightarrow{q})$ die Funktion sei, die prüft ob p auf der linken Seite des Vektors $\overrightarrow{q}$ liegt und $S.insert(p_i, j)$ den Punkt p_i hinter dem Punkt p_j zum Polygonzug S der konvexen Hülle hinzufügt:

Algorithm 2 $CH(S, p_n, T)$

```
1:  p_i = T.search(p_n)
2:  if left(p_n, p_i p_{i+1}→) then
3:     return
4:  end if
5:  T.insert(p_n)
6:  S.insert(p_n, i)
7:  j = i
8:  while left(p_n, p_{j-1} p_j→) do
9:     T.remove(p_j)
10:    S.remove(p_j)
11:    j− = 1
12: end while
13: while left(p_n, p_i p_{i+1}→) do
14:    T.remove(p_i)
15:    S.remove(p_i)
16:    i+ = 1
17: end while
```

Speicherplatz:
$O(n)$ für einen AVL-Baum.

Laufzeit:
Es bietet sich hier an, den Polygonzug in einer verketteten Liste zu speichern. Die Elemente der Liste werden stets über den Baum adressiert - was effizient ist - während in konstanter Zeit Zeiger umgelegt werden können. Für das Einfügen von n Punkten muss $n\times$ *CH(...)* ausgeführt werden. Dessen Zeitaufwand ergibt sich zu:

- $O(\log n)$ für das Suchen im Baum (1)
- $O(1)$ für Listenoperationen (2-4, 6)
- $O(\log n)$ für das Einfügen in den Baum (5)
- $O(f(n)) \cdot O(\log n)$ wobei die Größe von $f(n)$ im Folgenden zu untersuchen ist (8-17)

Anstatt über planare Graphen zu argumentieren ist hier vlt. eine sehr einfache Anwendung der Buchhalter-Methode möglich!?:

Jeder Punkt wird nur höchstens einmal zu der konvexen Hülle hinzugefügt und beim Hinzufügen mit einem Guthaben von '1' ausgestattet. Wird ein Punkt gar nicht zur konvexen Hülle hinzugefügt verbleibt er mit einem Guthaben von '0'. Beim Hinzufügen eines Punktes p_i werden in den beiden Schleifen einmal der linke und einmal der rechte Nachbar inspiziert, was beim Hinzufügen von n Punkten $O(n)$ Zeit benötigt. Nur wenn p_{i-1} oder p_{i+1} entfernt werden müssen, besteht die Notwendigkeit einen weiteren Punkt zu untersuchen. Für diesen zusätzlichen Schritt wird nun das Guthaben von p_{i-1} bzw. p_{i+1} aufgebraucht. p_{i-1} bzw. p_{i+1} werden in diesem Fall allerdings

entfernt und verbleiben so mit einem Guthaben von '0'.
D.h.: neben konstant 2 Vergleichen pro Aufruf des Algorithmus wird jede Kante des Polygons bei der Suche nach zu Entfernenden Kanten höchstens 1 mal passiert (nach dem ersten Mal fliegt die Kante raus). Bei n-maligem Aufruf fallen somit nur Kosten von $O(n)$ an.

Die Gesamtlaufzeit ergibt sich somit zu $O(n \cdot \log n)$.

Aufgabe 3 (untere Schranke):

Angenommen die konvexe Hülle der Punkte $S = \{(a_1, a_1^2), ..., (a_n, a_n^2)\}$ ließe sich in weniger als $\Omega(n \cdot \log n)$ bestimmen. Die konvexe Hülle hat hier stets die Form $CH(S) = p_1, .., p_n$ mit $x_i \geq x_{i+1}$ und $y_i \geq y_{i+1}$ wobei kein Punkt aus S im Inneren des entstehenden Polygons liegt (vgl. auch Abb. 2).

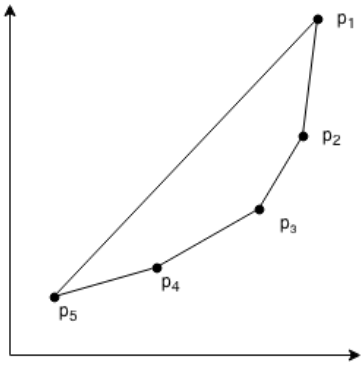

Abbildung 2: Konvexe Hülle (keine lineare Skala!)

Beweis (Induktion):
I.A.: Der Polygonzug p_1, p_2, p_3 aus 3 Punkten beschreibt offensichtlich immer eine konvexe Hülle.

I.V.: Der Polygonzug $p_1, ..., p_n$ mit $x_i \geq x_{i+1}$ beschreibt die konvexe Hülle von $\{p_1, ..., p_n\}$

$n \to n+1$: Alle Punkte $p_1, ..., p_n$ liegen rechts der Geraden $\overrightarrow{p_{n+1}p_1}$, da stets

$$\frac{y_1 - y_{n+1}}{x_1 - x_{n+1}} > \frac{y_1 - y_n}{x_1 - x_n}$$
$$\Leftrightarrow \frac{x_1^2 - x_{n+1}^2}{x_1 - x_{n+1}} > \frac{x_1^2 - x_n^2}{x_1 - x_n}$$

(aufgrund der Tatsache, dass nach Voraussetzung $x_n \geq x_{n+1}$) und nach I.V. alle p_i mit $i < n$ schon vorher rechts der Geraden $\overrightarrow{p_n p_1}$ lagen. p_n (und somit auch alle p_i mit $i < n$) kann jedoch auf keinen Fall aus der konvexen Hülle "gestrichen" werden, da (nach derselben Rechnung) die Steigung von $\overrightarrow{p_n p_{n-1}}$ steiler sein muss, als die von $\overrightarrow{p_{n+1}p_n}$. Damit muss die konvexe Hülle $CH(p1, ..., p_{n+1}) = p1, ..., p_{n+1}$ sein.

Somit könnte man den Polygonzug in linearer Zeit ablaufen um die sortierte Folge $a_n, .., a_1$ zu erhalten (da stets $x_i \geq x_{i+1}$). Das Sortieren der Folge hätte somit nur soviel Zeit benötigt wie die Konstruktion der konvexen Hülle zzgl. $O(n)$ für das Berechnen der Quadrate und noch einmal $O(n)$ für das Ablaufen des Polygonzugs. Dies ist nach Annahme jedoch nicht möglich: die Konstruktion der konvexen Hülle muss folglich $\Omega(n \cdot \log n)$ Zeit benötigen.