Übungsblatt 3

Julius Auer, Alexa Schlegel

Aufgabe 1 (Graham-Scan):

Implementieren Sie den Graham-Scan. Benutzen Sie dabei aus Gründen besserer Laufzeit und zur Vermeidung von Rundungsfehlern möglichst nur die Grundrechenarten $+, -, \times, \div$*. Eine Umrechnung in Polarkoordinaten, wie in der Vorlesung beschrieben, ist nicht nötig, um die Strahlen nach Steigung zu sortieren.*

Im Folgenden wird der implementierte Algorithmus mit Hilfe von Pseudocode beschreiben, 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 terminierung des Algorithmus enthält S die Knoten der konvexen Hülle entgegen dem Urzeigersinn.

Algorithm 1 Graham-Scan(Q)

sei $p_0 \in Q$ mit minimaler y-Koordinate:
bei gleicher y-Koordinate wird minimale x-Koordinate verwendet
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, nur Punkt mit größtem Abstand von p_0 behalten
PUSH(p_0, S)
PUSH(p_1, S)
PUSH(p_2, S)
for $i = 3$ **to** n **do**
  while Winkel (NEXT-TO-TOP(S), TOP(S), p_i) ist keine Linkskurve **do**
    POP(S)
  end while
  PUSH(p_i, S)
end for
return S

Die Methode *grahamscan()* erhält als Eingabe eine Punktwolke und liefert ein Polygon zurück, welches der Konvexenhülle dieser Punktwolke entspricht.

Wir verwenden als Startpunkt des Algorithmus den Punkt der am weitestens Links liegt und die kleinsten y-Koordinte besitzt (*getMinY()*). (In der Vorlesung wird ein beliebiebiger Punkt in der Mitte der Punktwolke ermittelt.)

Die Sortierung der Punkte erfolgt nach ihrer Steigung. Damit die Punkte gegen den Urzeigersinn sortiert sind werden sie vorher an der Funktion $f(x) = |x|$ gespiegelt und erst dann der Anstiegt berechnet (*getRelevantPoints()*).

Um Links- und Rechskurven zu prüfen, wird die Determinante einer $2x2$-Matrix bestimmt, welche sich aus den zwei Vektoren der folgenden drei Punkte zusammensetzt. t ist der letzte Punkt auf dem Stack, *ntt* der vorletzte Punkt und *ca* der Punkt der aktuelle betrachtet wird. Es wird getestet ob $\overline{t\ cp}$ links oder rechts von $\overline{t\ ntt}$ liegt.

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

```
public static Polygon grahamscan(Points pointcloud) {

    // get minimum
    Point min = pointcloud.getMinY();

    // select only relevant points from point cloud
    Map<MyAnstieg, Point> relevantPoints = pointcloud.getRelevantPoints();

    // sort by ascent
    Map<MyAnstieg, Point> sortedPoints = new TreeMap<MyAnstieg, Point>(Collections.
         reverseOrder());
    for (MyAnstieg a : relevantPoints.keySet()) {
        sortedPoints.put(a, relevantPoints.get(a));
    }

    // put first 3 points on stack
    Stack<Point> stack = new Stack<Point>();
    stack.push(min);

    MyAnstieg ascent = ((TreeMap<MyAnstieg, Point>) sortedPoints).firstKey();
    Point p = sortedPoints.get(ascent);
    sortedPoints.remove(ascent);
    stack.push(p);

    ascent = ((TreeMap<MyAnstieg, Point>) sortedPoints).firstKey();
    p = sortedPoints.get(ascent);
    sortedPoints.remove(ascent);
    stack.push(p);

    // have a look at each point
    for (MyAnstieg ca : sortedPoints.keySet()) { // ca ... current ascent

        Point t = stack.pop(); // t ... top
        Point ntt = stack.peek(); // ntt ... next-to-top
        stack.push(t); // put top back on stack

        // use determinant for testing left/right turns
        Vector v1 = new Vector(ntt.getX()-t.getX(), ntt.getY() - t.getY());
        Vector v2 = new Vector(sortedPoints.get(ca).getX() - t.getX(), sortedPoints.get
             (ca).getY() - t.getY());

        Mat2x2 matrix = new Mat2x2(v1.get(0), v1.get(1), v2.get(0), v2.get(1));

        // testing for not left-turns
        while (! (matrix.getDeterminant() <= 0) ) {

            // remove last point
            stack.pop();

            // have a look at the next point
            t = stack.pop();
            ntt = stack.peek();
            stack.push(t);

            v1 = new Vector(ntt.getX()-t.getX(), ntt.getY() - t.getY());
            v2 = new Vector(sortedPoints.get(ca).getX() - t.getX(), sortedPoints.get(ca
                 ).getY() - t.getY());

            matrix = new Mat2x2(v1.get(0), v1.get(1), v2.get(0), v2.get(1));
        }

        // put current point on stack
        stack.push(sortedPoints.get(ca));
    }

    // generate polygon from points on stack
    Point[] points = new Point[stack.size()];
    stack.toArray(points);

    return new Polygon(points);
```

```
    }
```

```
    public Point getMinY() {
        Point min = points[0];
        for(Point p : points) {
            if(min.getY() <= p.getY()) {
                if(min.getY() == p.getY()) {
                    // Compare x-values
                    if(min.getX() > p.getX()) {
                        min = p;
                    }
                }
            }
            else {
                min = p;
            }
        }
        return min;
    }
```

```
    public Map<MyAnstieg, Point> getRelevantPoints() {
        Point min = getMinY();
        Map<MyAnstieg, Point> relevantPoints = new HashMap<MyAnstieg, Point>();

        for(Point p : points) {

            if(p != min) {

                Double d = p.toPosition().substract(min.toPosition()).getMirroredAscent();
                MyAnstieg m = new MyAnstieg(d);

                if (Double.isInfinite(d)) {
                    m = new MyAnstieg(Double.MAX_VALUE);
                }

                if(!relevantPoints.containsKey(m)) {
                    relevantPoints.put(m, p);
                } else {
                    Point cp = relevantPoints.get(m);
                    double distanceold = cp.getX()*cp.getX()+cp.getY()*cp.getY();
                    double distancenew = p.getX()*p.getX()+p.getY()*p.getY();

                    if(distancenew > distanceold) {
                        relevantPoints.remove(m);
                        relevantPoints.put(m, p);
                    }
                }
            }
        }
        return relevantPoints;
    }
```

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

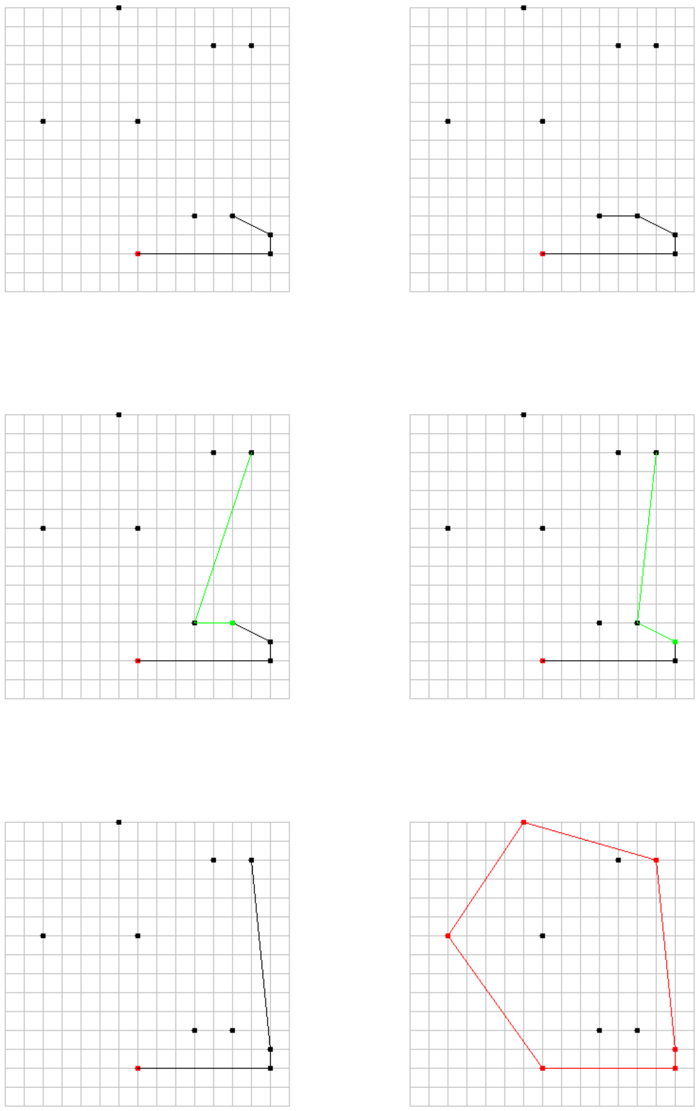

Abbildung 1: Die ersten 5 Schritte und die resultierende Konvexehülle.

Aufgabe 2 (inkrementelle Konstruktion der konvexen Hülle):

TODO

Aufgabe 3 (untere Schranke):

TODO