Übungsblatt 2

Julius Auer, Alexa Schlegel

Aufgabe 1 (rotating calipers):

Das Berechnen des Durchmessers entspricht dem Finden der antipodalen Punkte mit dem größten Abstand (zur Erinnerung: *toPosition()* wandelt lediglich einen *Point* in einen *Vector* um. Nur Letztere haben eine Länge):

```
1  public double diameter() {
2         double max = 0.0d;
3
4         for(PodalPoints p : antipodalPoints()) {
5             double d = Math.abs(p.point1.toPosition().substract(p.point2.toPosition()).
                   length());
6
7             if(d > max)
8                 max = d;
9         }
10
11        return max;
12    }
```

Instanzen der Klasse *PodalPoint* speichern hierbei die Ergebnisse einer Iteration der *RotatingCalipers*. Relevant sind an dieser Stelle nur die antipodalen Punkte *point1* und *point2*, darüber hinaus werden allerdings auch noch die gefundenen Tangenten (zur Visualisierung) sowie die Indizes der Punkte und die Brücken-Eigenschaft der CSL (zur Berechnung der Konvexen Hülle aus ko-podalen Punkten (wie heißen die im Deutschen?)) gespeichert.

Interessanter ist der Aufruf der *RotatingCalipers* (zur Erinnerung: alle Punkte des Polygons liegen in einem Array *points*):

```
1     public LinkedList<PodalPoints> antipodalPoints() {
2         LinkedList<PodalPoints> result = new LinkedList<PodalPoints>();
3         int p1 = 0;
4         int p2 = 0;
5
6         // the first antipodal points are given by the points with the min/max x
7         // coordinate
8         for(int i = 1; i < points.length; i++) {
9             if(points[i].compareTo(points[p1]) < 0)
10                p1 = i;
11
12            if(points[i].compareTo(points[p2]) > 0)
13                p2 = i;
14        }
15
16        // the left/right caliper is constructed 'upwards'/'downwards'
17        Line caliper1 = new Line(points[p1], new Point(points[p1].getX(), points[p1].getY
              () + 10.0d));
18        Line caliper2 = new Line(points[p2], new Point(points[p2].getX(), points[p2].getY
              () - 10.0d));
19        double rotatedAngle = 0.0d;
20
21
22
23
24
25
```

```
26          // do until the calipers rotated a total of 180 degrees
27          while(rotatedAngle < Math.PI) {
28              // for both calipers: compute the angle between the caliper at point
29              // p_i and the linesegment p_i->p_(i+1)
30              double angle1 = caliper1.angleTo(lines[p1]);
31              double angle2 = caliper2.angleTo(lines[p2]);
32              angle1 = Math.abs(angle1 > 0 ? 2.0d * Math.PI - angle1 : angle1);
33              angle2 = Math.abs(angle2 > 0 ? 2.0d * Math.PI - angle2 : angle2);
34
35              // choose the smaller angle and rotate both calipers by that angle
36              double minAngle = Math.min(angle1, angle2);
37              caliper1 = new Line(points[p1], caliper1.rotate(points[p1], minAngle).u);
38              caliper2 = new Line(points[p2], caliper2.rotate(points[p2], minAngle).u);
39              rotatedAngle += minAngle;
40
41              // move the point that lies on the choosen caliper
42              if(angle1 < angle2)
43                  p1 = (p1 + 1) % points.length;
44              else
45                  p2 = (p2 + 1) % points.length;
46
47              result.add(new PodalPoints(p1, p2, points[p1], points[p2], caliper1, caliper2))
                    ;
48          }
49
50          // just in case we rotated slightly more then 180 degrees, check if the
51          // first point is a duplicate of the last one
52          if(result.getLast().point1 == result.getFirst().point2 && result.getLast().point2
                == result.getFirst().point1)
53              result.removeFirst();
54
55          return result;
56      }
```

Neu Implementiert wurden hierfür die Methoden *angleTo()* für *Line*s und *rotate(point, angle)* ebenfalls für *Line*s:

```
1       /**
2        * Compute the angle between this and another line. Result is positive or
3        * negative just as like an atan2 were used.
4        *
5        * @param line a line with the direction this line would have if rotated
6        * anticlockwise by the resulting angle
7        * @return angle in radians
8        */
9       public double angleTo(Line line) {
10          return Math.atan2(line.u.get(1), line.u.get(0)) - Math.atan2(u.get(1), u.get(0));
11      }
12
13      /**
14       * Rotate this line around a given point
15       *
16       * @param point centre of rotation
17       * @param angle angle in radians
18       * @return the rotated line
19       */
20      public Line rotate(Point point, double angle) {
21          Vector pos = point.toPosition();
22          double cosa = Math.cos(angle);
23          double sina = Math.sin(angle);
24          Mat2x2 mat = new Mat2x2(cosa, -sina, sina, cosa);
25
26          // shift the line so that 'point' lies at the coordinates center, rotate
27          // it around the euler-angle, shift it back
28          return new Line(p1.add(pos.multiply(-1)).compose(mat).add(pos),
29                  p2.add(pos.multiply(-1)).compose(mat).add(pos));
30      }
```

Die implementierten Testfälle sind visualisiert und animiert. Es lohnt sich trotzdem, das Jar über die Kommandozeile zu starten um ein paar zusätzliche Informationen zu bekommen. Es wird auch ein Test für das finden der Konvexen Hülle zweier Polygone animiert - da hier auch Calipers rotieren gehe ich davon aus, das dies nicht allzu sehr stört - einfach wegklicken.

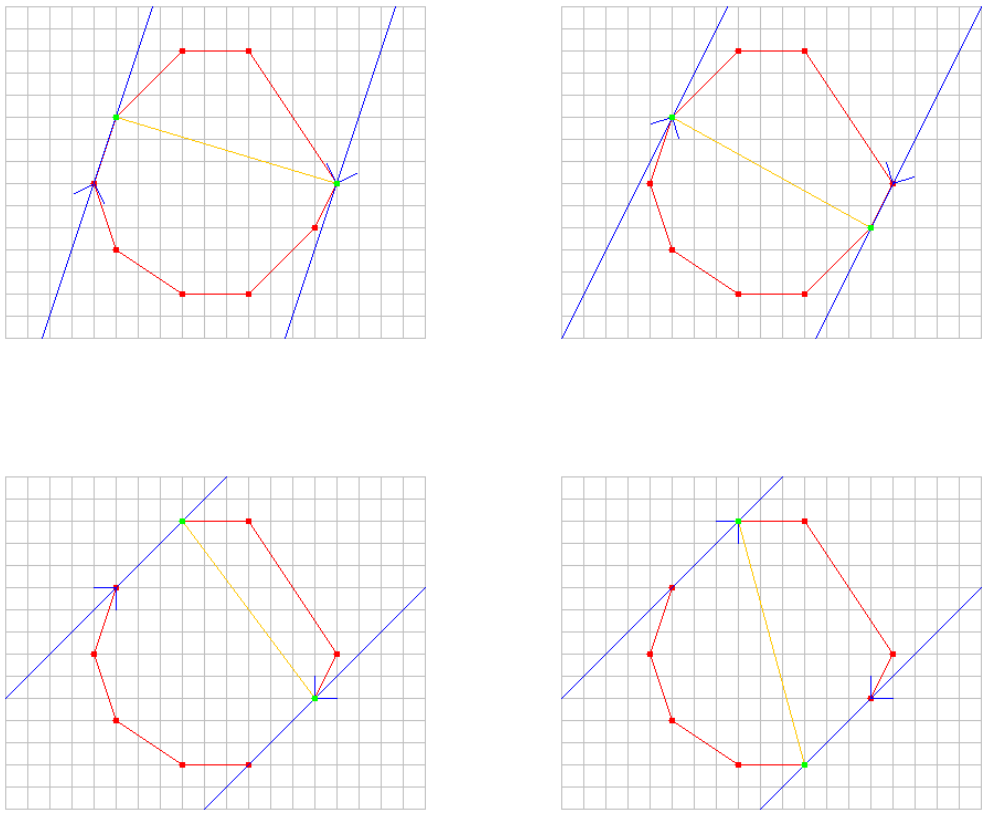

Abbildung 1: 4 Schleifen-Durchläufe der Rotating Calipers

Aufgabe 2 (konvexe Hülle):

a) Bei Erreichen des Rekursionsankers des divide-and-conquer-Algorithmus wird aus 3 Punkten in $O(1)$ ein konvexes Polygon S konstruiert. Offensichtlich braucht es hier nicht mehr als noch einmal $O(1)$ Zeit, um für dieses Polygon die Punkte mit der kleinsten/größten y-Koordinate sy_{min}/sy_{max} zu bestimmen und für dieses Polygon zu speichern.

Werden zwei Polygone S_1, S_2 gemerged wird folglich ebenfalls nur konstante Zeit benötigt um $y_{min} = min_y(s_1 y_{min}, s_2 y_{min})$ und $y_{max} = max_y(s_1 y_{miax}, s_2 y_{max})$ zu bestimmen. Hierdurch stehen zwei Ecken von $CH(S_1 \cup S_2)$ fest.

Da S_1, S_2 disjunkt sind (aufgrund der Sortiertheit) gibt es zwischen S_1 und S_2 genau 2 Brücken. Es muss nun also lediglich für y_{min}/y_{max} die obere/untere CLS zum anderen Polygon gefunden werden. Es wird im Folgenden geschildert wie eine solche Brücke gefunden wird, falls S_1 das linke Polygon ist mit $p_i = y_{max} \in S_1$. Die übrigen Fälle können analog berechnet werden.

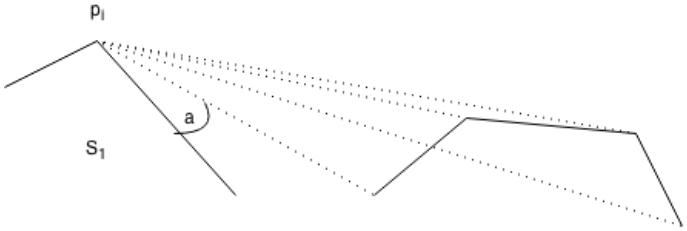

Abbildung 2: Brücke Punkt-Polygon

Das von Toussaint vorgeschlagene Prinzip ist eine Binärsuche (die bekanntlich in $O(\log n)$ läuft): für einen Punkt $q_i \in S_2$ wird der Winkel $\alpha = angle(\overline{p_i p_{i+1}}, \overline{p_i, q_i})$ betrachtet. Gesucht ist der Punkt mit maximalem Winkel. Für einen zufälligen Startpunkt q_i wird nun einfach eine Binärsuche in Richtung q_{i-1} und eine weitere in Richtung q_{i+1} durchgeführt - es ist zu Beginn nicht klar, in welcher Richtung der Punkt gefunden werden kann: Während sich in der einen Richtung der Winkel vergrößert wird er sich in der anderen zunächst verkleinern um sich ggf. später zu vergrößern.
In beiden Fällen lässt sich mit binärer Suche der Punkt mit maximalem α finden, der somit zum Brückenpunkt wird.

b) TODO