Übungsblatt 12

Julius Auer, Alexa Schlegel

Aufgabe 1 (Sichtbarkeit und kürzeste Wege):

Eine Anmerkung vorab: Rundungsfehler sind hier (insbesondere für die Konsistenz des Suchbaums) kritisch. Ohne einen signifikanten Mehraufwand zu betreiben, konnten wir nicht alle auf Rundungsfehlern zurückzuführende Sonderfälle berücksichtigen. Wenn Du das Teil ausführst ist deshalb damit zu rechnen das ggf. zufällige Punkte erzeugt werden, bei denen es nicht funktioniert. Das passiert ungefähr in 1/20 Fällen (also "selten") vor allem dann, wenn Punkte aus unterschiedlichen Polygonen in guter Näherung kollinear sind. Einfach noch mal versuchen :)

a) Sei p_l der Punkt, um den der Strahl rotieren soll.

In der Vorverarbeitung werden die Punkte aus den Polygonen in eine Liste kopiert und mit entsprechenden Zeigern versehen um Punkt-In-Liste-Indizes, Punkt-In-Polygon-Indizes, Polygon-Adressen und Punkt-Adressen zusammenzubringen. Hierfür ist auch die kleine Klasse "PP" ("Point-Pointer") erforderlich, die bei der Gelegenheit gleich noch eine Vergleichs-Methode spendiert bekommen hat, um die Punkte zu sortieren - die Sortierung erfolgt für einen Punkt $p$anhand des Winkels zwischen $\overline{p_l p}$ und der zur x-Achse parallelen Geraden durch p_l.
Außerdem wird für jede Kante des Polygons geprüft, ob sie initial in den Baum eingefügt werden muss. Das Iterieren über Ecken und Kanten benötigt $O(n)$ Zeit, zusätzlich benötigt die Vorverarbeitung jedoch noch $O(n \cdot \log n)$ um die Punkte mit *TimSort* zu sortieren.

Die Hauptschleife des Algos läuft erneut über alle Punkte , wobei in jedem Schritt höchstens zwei Kanten aus dem Baum entfernt oder zum Baum hinzugefügt werden. Auch hier wird folglich $O(n \cdot \log n)$ Zeit benötigt.

Einige Probleme machte die Ordnung des Baums, die ja für unterschiedliche Strahlen eine konsistente Sortierung der Kanten gewährleisten muss. Für kollineare Punkte klappt das nicht immer. So wie alle anderen Details auch, findet sich dazu etwas mehr Information in den Kommentaren:

```java
public class RotationalSweep {
    // the rotating beam is an instance field to provide accesse from inner
    // anonymous classes
    protected Beam beam;

    /**
     * Find all vertices of given polygons that are visible from a given point.
     * Result is an array containing the distance from the point to all visible
     * vertices.
     */
    private double[] _visiblePoints(Frame frame, Point location, Polygon... obstacles) {
        // A wrapper-class for points that provides access to the vertex of a
        // polygon by either:
        //
        // 1) knowing the polygon and the index of the point inside the polygon
        // ('PP' means 'Point-Pointer' ;) )
        //
        // 2) or knowing the index of the point in the list of all points
        // (needed for mapping purposes)
        class PP implements Comparable<PP> {
            Polygon polygon;
```

```
        int index_poly;
        int index_global;

        PP(Polygon polygon, int index_poly, int index_global) {
            this.polygon = polygon;
            this.index_poly = index_poly;
            this.index_global = index_global;
        }

        // the natural order of points here is determined by the angle
        // between 'location' and a point
        @Override
        public int compareTo(PP p) {
            Line axis = new Line(location, new Point(location.getX() + 1, location.getY()));
            double angle1 = new Line(location, point()).angleTo(axis);
            double angle2 = new Line(location, p.point()).angleTo(axis);

            if(angle1 < 0)
                angle1 += 2 * Math.PI;

            if(angle2 < 0)
                angle2 += 2 * Math.PI;

            return angle1 == angle2 ? 0 : angle1 < angle2 ? -1 : 1;
        }

        Point point() {
            return polygon.points[index_poly];
        }
    }

    // a simple list containing all points (decoupled from the original
    // polygons)
    LinkedList<PP> points = new LinkedList<PP>();

    // initialize the beam (it points straight to the right)
    beam = new Beam(location, new Point(location.getX() + 1, location.getY()));

    // a red-black tree storing linesegments. The order of two linesegments
    // is given by the distance to the corresponding intersections with the
    // beam
    TreeMap<LineSegment, Object> tree = new TreeMap<LineSegment, Object>(new Comparator<LineSegment>() {

        @Override
        public int compare(LineSegment l1, LineSegment l2) {
            if(l1.p1 == l2.p1 && l1.p2 == l2.p2)
                return 0;

            double dist1 = Math.abs(beam.intersectionWith(l1).toPosition().substract(location.toPosition()).length());
            double dist2 = Math.abs(beam.intersectionWith(l2).toPosition().substract(location.toPosition()).length());

            // if the intersections are very close, it is highly likely the
            // two lines have a common point. In this case the two
            // linesegments are ordered by the index they have in their
            // polygon
            if(Math.abs(dist1 - dist2) < C.E) {
                if(l1.p2 == l2.p1)
                    return -1;

                if(l2.p2 == l1.p1)
                    return 1;
            }

            return dist1 < dist2 ? -1 : 1;
        }
    });
```

```java

        int index = 0;

        // fill the point-list with all vertices of the polygons. Check for each
        // edge e of the polygons if e intersects with the initial beam - if so,
        // insert e into the tree
        for(Polygon polygon : obstacles)
            for(int i = 0; i < polygon.points.length; ++i) {
                points.add(new PP(polygon, i, index++));
                LineSegment l = polygon.edge(i);

                if(beam.intersectionWith(l) != null)
                    tree.put(l, null);
            }

        // sort the points (by angle to 'location' - see comparator above)
        Collections.sort(points);
        double[] result = new double[points.size()];
        Arrays.fill(result, Double.MAX_VALUE);

        // run the actual algorithm
        for(PP p : points) {
            // rotate the beam to the next point
            beam = new Beam(location, p.point());
            LineSegment edge1 = p.polygon.edge(p.index_poly - 1);
            LineSegment edge2 = p.polygon.edge(p.index_poly);

            // check for the two vertices at p wether they lie left or right of
            // the beam
            boolean e1LeftOfBeam = p.polygon.point(p.index_poly - 1).distanceTo(beam) >= 0;
            boolean e2LeftOfBeam = p.polygon.point(p.index_poly + 1).distanceTo(beam) >= 0;
            boolean unableToRemove = false;

            // remove those edges from the tree which lie left of the beam
            if(e1LeftOfBeam)
                unableToRemove = tree.remove(edge1) == null;

            if(e2LeftOfBeam)
                tree.remove(edge2);

            // who looked at the comparator closely knows, there might be a
            // singularity issue if the beam hits a vertex and both edges have
            // to be removed from the tree. In this case it may be edge1 can not
            // be found in the tree - after removing edge2 the singularity is
            // gone and edge1 can be removed as well
            if(e1LeftOfBeam && unableToRemove)
                tree.remove(edge1);

            // IF (the beam does not intersect the very first edge in the tree
            // OR if the intersection of beam and edge is farther away then p)
            // THEN p is visible
            LineSegment candidateForBlocking = tree.isEmpty() ? null : tree.firstKey();
            Point intersection = candidateForBlocking == null ? null : beam.intersectionWith(candidateForBlocking);

            if(intersection != null
                    && Math.abs(location.toPosition().substract(p.point().toPosition()).length())
                    < Math.abs(location.toPosition().substract(intersection.toPosition()).length()))
                intersection = null;

            if(intersection == null)
                result[p.index_global] = Math.abs(location.toPosition().substract(p.point().toPosition()).length());

            // insert those edges to the tree which lie right of the beam
            if(!e1LeftOfBeam)
                tree.put(edge1, null);

```

```java
            if(!e2LeftOfBeam)
                tree.put(edge2, null);

            // draw debug
            if(frame != null) {
                Scene scene = new Scene(500);
                scene.add(location, Color.BLUE);

                for(LineSegment l : tree.keySet())
                    scene.add(l, Color.CYAN);

                if(intersection != null) {
                    scene.add(p.point(), Color.RED);
                    scene.add(candidateForBlocking, Color.RED);
                }

                scene.add(beam, Color.YELLOW);

                for(int i = 0; i < result.length; ++i)
                    if(result[i] < Double.MAX_VALUE) {
                        int j = i;
                        int polygon = 0;

                        while(j >= obstacles[polygon].points.length) {
                            j -= obstacles[polygon].points.length;
                            ++polygon;
                        }

                        scene.add(obstacles[polygon].point(j), Color.GREEN);
                    }

                frame.addScene(scene);
            }
        }

        return result;
    }

    public static double[] visiblePoints(Frame frame, Point location, Polygon... obstacles) {
        return new RotationalSweep()._visiblePoints(frame, location, obstacles);
    }
}
```

Abbildung 2 zeigt zwei aufeinanderfolgende Schritte im Algorithmus:

- Gelb: der rotierende Strahl
- Cyan: Kanten die aktuell im Baum sind
- Rot: Kante die aktuell die Sicht auf rote Ecke versperrt
- Grün: bisher gefundene sichtbare Ecken

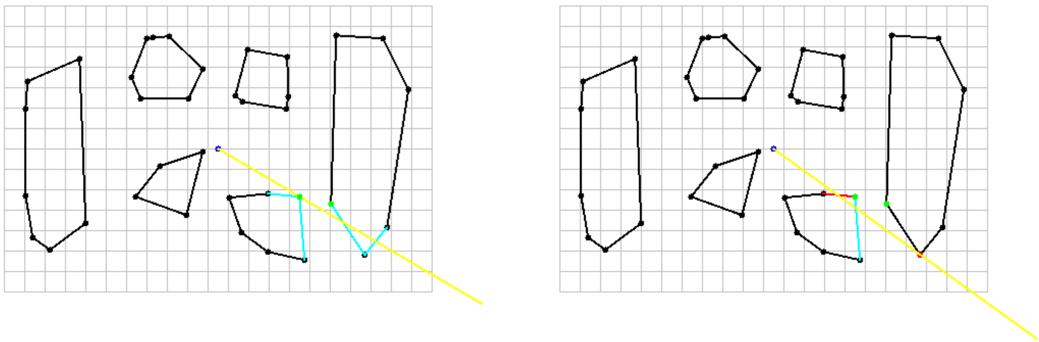

Abbildung 1: Rotational Sweep

b) Der Rest ist leicht: Der Sichtbarkeitsgraph entsteht aus der Anwendung des Sweep für jede Ecke aller Polygone. Bei insgesamt n Ecken benötigt das folglich $O(n^2 \cdot \log n)$ Zeit und $O(n^2)$ Platz für das Speichern des Graphen.

Interessant ist hier lediglich, welcher Punkt als Ausgangspunkt für den Strahl gewählt wird - der Sweep funktioniert in unserer Implementierung nicht für Punkte innerhalb eines Polygons (auch nicht auf dem Rand!). Gerade unter Berücksichtigung von Rundungsfehlern wären hier einige Spezialfälle zu behandeln wann inzidente Kanten die Sicht blockieren oder eben nicht.

Wir haben es uns hier ein wenig leicht gemacht: da wir ohnehin nur eine gewisse Präzison gewährleisten können (aufgrund der oben erwähnten Probleme mit Ungenauigkeiten bei der Sortierung des Baums) können wir auch hier eine kleine Ungenauigkeit erlauben und p_l einfach ein kleines Stück von seinem Polygon "wegschieben". Hiermit sind - bei sehr kleinem Aufwand - alle Rundungs- und sonstigen Ungenauigkeiten erledigt und die möglichen Abweichungen in der Ergebnismenge fallen ohnehin in einen Bereich der schon aufgrund der Ungenauigkeit des Sweeps hingenommen werden muss.

Auf die (kürzesten) Wege hat diese Heuristik keine Auswirkung, da hierfür wieder die Original-Punkte (statt der verschobenen) genutzt werden.

Im Fall einer Ungenauigkeit kann ein Punkt der (annähernd) kollinear zu einer nicht inzidenten Kante ist als sichtbar erkannt werden, obwohl er es knapp nicht ist.

```
// the visibility graph. For values equal to Double.MAX_VALUE two vertices
// do not see each other
double[][] graph;

// all vertices of the polygons as list. The indices are corresponding to
// those used for the arrays above
ArrayList<Point> points;

private VisibilityGraph(Polygon... obstacles) {
    // nothing special here. Put all points into a list, and allocate memory
    // for the arrays
    int n = 0;

    for(Polygon poly : obstacles)
        n += poly.points.length;

    points = new ArrayList<Point>(n);
    graph = new double[n][];

    for(Polygon poly : obstacles)
        for(Point p : poly.points)
```

```java
                points.add(p);
    }

    public static VisibilityGraph create(Frame frame, Polygon... obstacles) {
        VisibilityGraph graph = new VisibilityGraph(obstacles);
        int i = 0;

        // for each point, run the rotational sweep
        for(Polygon poly : obstacles)
            for(int j = 0; j < poly.points.length; ++j) {
                // the beams origin for the rotational sweep must not be within
                // any polygon. Pull it a little outside ...
                Point p = Point.fromPosition(poly.point(j).toPosition().add(
                        new Line(poly.point(j - 1), poly.point(j + 1)).n0.multiply(0.1)));

                // output of the rotational sweep is one line of the result
                graph.graph[i] = RotationalSweep.visiblePoints(null, p, obstacles);

                // the weight of edge i->i is 0
                graph.graph[i][i] = 0.0d;
                ++i;

                // draw debug
                if(frame != null) {
                    Scene s = new Scene(500);
                    Color c = Color.RED;

                    for(int k = 0; k < i; ++k) {
                        if(k == i - 1)
                            c = Color.GREEN;

                        for(int l = 0; l < graph.points.size(); ++l)
                            if(graph.graph[k][l] != 0 && graph.graph[k][l] < Double.
                                MAX_VALUE)
                                s.add(new LineSegment(graph.points.get(k), graph.points.get(l
                                    )), c);
                    }

                    s.add(p, Color.BLUE);
                    frame.addScene(s);
                }
            }

        return graph;
    }
```

Abbildung 2 zeigt zwei aufeinanderfolgende Schritte im Algorithmus:

- Grün: Ergebnis des Sweeps für eine Ecke v_i
- Rot: Ergebnisse der bisherigen Sweeps aller Ecken $v_1, ..., v_{i-1}$

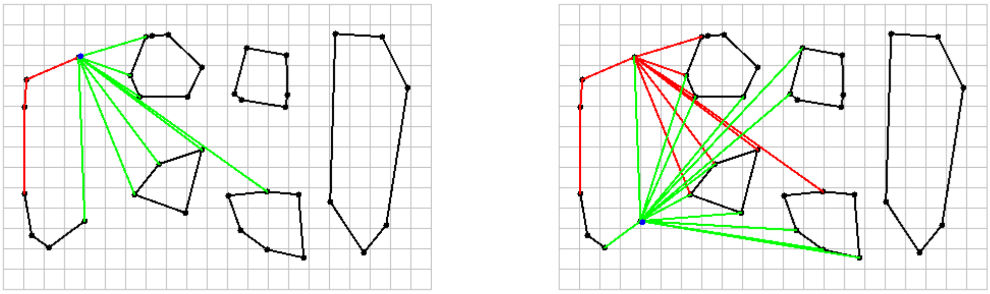

Abbildung 2: Sichtbarkeitsgraph

c) Warum einen berechnen, wenn man alle berechnen kann? Da der Anwendungsfall nicht näher spezifiziert ist (und ich sowieso noch eine alte Implementierung griffbereit habe ;)) machen wir keinen Dijkstra (wie ggf. vom Autor der Aufgabe angedacht) sondern einen Floyd-Warshall um *all-pairs-shortest-paths* zu berechnen. Im schlimmsten Fall (der Sichtbarkeitsgraph ist vollständig) benötigt man hierfür $O(n^3)$ Zeit und erneut $O(n^2)$ Platz. Abbildung 3 zeigt exemplarisch zwei dieser Wege.

```
// all pairs shortest paths. At dimension 3 there are two values: [0] stores
// the weight of the path, [1] stores the index of the next vertex on the
// path
double[][][] paths = null;
```

```
/**
 * It's the very popular algorithm of Floyd and Warshall ...
 */
private void floydWarshall() {
    double[][] adjacencyMatrix = new double[graph.length][];

    for(int i = 0; i < graph.length; ++i) {
        adjacencyMatrix[i] = new double[graph.length];
        System.arraycopy(graph[i], 0, adjacencyMatrix[i], 0, graph.length);
    }

    paths = new double[graph.length][graph.length][2];

    for(int i = 0; i < graph.length; i++)
        for(int j = 0; j < graph.length; j++) {
            paths[i][j][0] = graph[i][j];

            if(i == j || adjacencyMatrix[i][j] == Double.MAX_VALUE)
                paths[i][j][1] = -1;
            else
                paths[i][j][1] = i;
        }

    // above: initialization
    // ======================
    // below: actual algorithm

    for(int k = 0; k < graph.length; k++)
        for(int i = 0; i < graph.length; i++)
            for(int j = 0; j < graph.length; j++)
                if(paths[i][k][0] != Double.MAX_VALUE && paths[k][j][0] != Double.MAX_VALUE) {
                    double newDist = paths[i][k][0] + paths[k][j][0];

```

```java
                        // the only 'special' thing on this implementation is
                        // that the weight and the next vertex on the path are
                        // stored in the same array (as explained at top of this
                        // file)
                        if(newDist < paths[i][j][0]) {
                            paths[i][j][0] = newDist;
                            paths[i][j][1] = paths[k][j][1];
                        }
                    }
    }

    /**
     * List the vertices on the shortest path from 'from' to 'to'
     */
    public LinkedList<Point> shortestPath(Point from, Point to) {
        if(paths == null)
            floydWarshall();

        LinkedList<Point> result = new LinkedList<Point>();
        int f = points.indexOf(from);
        int t = points.indexOf(to);

        while(f != t) {
            f = (int) paths[t][f][1];
            result.add(points.get(f));
        }

        return result;
    }

    /**
     * Get the length of the shortest path from 'from' to 'to'
     */
    public double shortestPathLength(Point from, Point to) {
        if(paths == null)
            floydWarshall();

        return paths[points.indexOf(to)][points.indexOf(from)][0];
    }
```

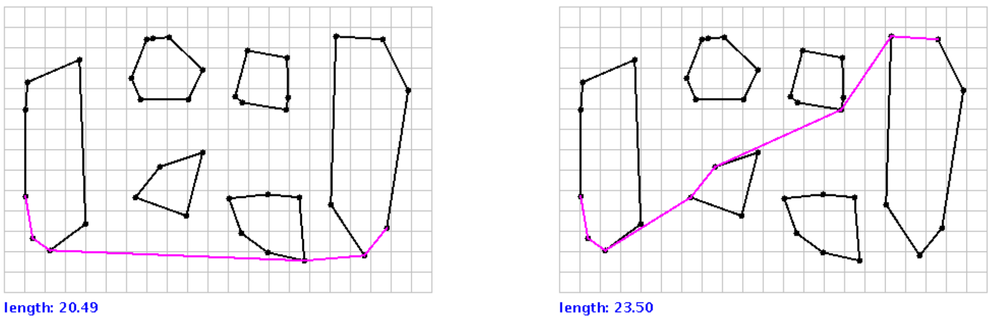

Abbildung 3: Kürzester Weg