Computational Geometry - Abgabe 2

1st Bartolovic Eduard
Hochschule München
München, Deutschland
eduard.bartolovic0@hm.edu

I. Berechnung des Flächeninhalts von einem Dreieck

Für die Berechnung der Fläche eines Dreiecks wird die Formel verwendet:

$$A = \frac{1}{2} ccw(p_1, p_2, p_3)$$

Der CCW ist so definiert:

$$ccw(p, q, r) := \begin{vmatrix} p_1 & p_2 & 1 \\ q_1 & q_2 & 1 \\ r_1 & r_2 & 1 \end{vmatrix}$$

II. Berechnung der Fläche eines Polygons

Für die Berechnung der Fläche eines Polygons berechnet man die Summe aller Flächen der Dreiecke die sich mit den Eckpunkten des Polygons und dem Nullpunkt bilden lassen können.

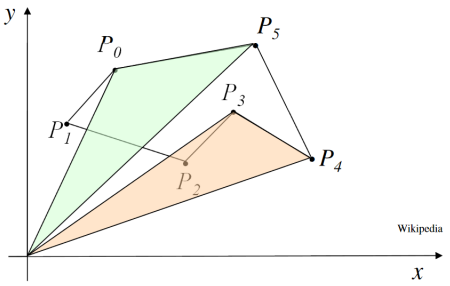

Abbildung 1. Berechnung der Fläche eines Polygons mit mehreren Dreiecken

$$A = \sum_{i=1}^{n} \frac{ccw(0, p_n, p_{n+1})}{2}$$

Da in die Formel des ccw der Nullpunkt eingesetzt wird lässt sich das Problem vereinfachen:

$$ccw(p, q, r) = p_x * q_y - p_y * q_x + q_x * r_y - q_y * r_x + p_y * r_x - p_x * r_y$$

$$ccw(0, q, r) = 0 * q_y - p_y * q_x + q_x * r_y - q_y * r_x + 0 * r_x - 0 * r_y$$

$$ccw(0, q, r) = 0 * q_y - 0 * q_x + q_x * r_y - q_y * r_x + 0 * r_x - 0 * r_y$$

$$ccw(0, q, r) = q_x * r_y - q_y * r_x$$

Zurück in die Formel eingesetzt:

$$A = \sum_{i=1}^{n} \frac{(x_i * y_{i+1} - y_i * x_{i+1})}{2}$$

Um die Anzahl der Divisionen zu verringern wird das $\frac{1}{2}$ aus der Summe herausgezogen:

$$A = \frac{1}{2} \sum_{i=1}^{n} (x_i * y_{i+1} - y_i * x_{i+1})$$

Zusätzlich lässt sich auch noch die Anzahl der Multiplikationen verringern [1]:

$$\begin{aligned} A &= \frac{1}{2} \sum_{i=1}^{n} (x_i * y_{i+1} - y_i * x_{i+1}) \\ &= \frac{1}{2} \sum_{i=1}^{n} (x_i * y_{i+1}) - \frac{1}{2} \sum_{i=1}^{n} (y_i * x_{i+1}) \\ &= \frac{1}{2} \sum_{i=2}^{n+1} (x_{i-1} * y_i) - \frac{1}{2} \sum_{i=1}^{n} (y_i * x_{i+1}) \\ &= \frac{1}{2} \underbrace{\sum_{i=2}^{n+1} (x_{i-1} * y_i)}_{= \sum_{i=1}^{n} (x_{i-1} * y_i), \text{ da } y_{n+1} = y_i \text{ und } x_0 = x_n} - \frac{1}{2} \sum_{i=1}^{n} (y_i * x_{i+1}) \\ &= \frac{1}{2} \sum_{i=1}^{n} (x_{i-1} * y_i) - \frac{1}{2} \sum_{i=1}^{n} (y_i * x_{i+1}) \\ &= \frac{1}{2} \sum_{i=1}^{n} (x_{i-1} * y_i - y_i * x_{i+1}) \\ A &= \frac{1}{2} \sum_{i=1}^{n} y_i (x_{i-1} - x_{i+1}) \end{aligned}$$

Wenn die Eckpunkte gegen den Uhrzeigersinn definiert sind dann ist die Fläche positiv. Im Uhrzeigersinn ist die Fläche negativ.

III. Berechnung der Fläche eines Bundeslandes

Ein Bundesland besteht mindestens aus einem Polygon. Manche Bundesländer sind aber über mehrere Polygone definiert. So zum Beispiel besitzt Niedersachsen Inseln die einzeln definiert sind. Auch ist die Stadt Bremen als Loch definiert. So muss die Fläche der Inseln und der Hauptfläche summiert werden. Die Fläche der Stadt Bremen aber abgezogen werden. Idealerweise hätte das Loch ein negatives Vorzeichen und die Inseln alle ein positives Vorzeichen sodass einfach eine Summe über alle Polygone gebildet werden kann.
Das Problem mit dem vorliegenden Datensatz ist das dieser sich nicht an die Konvention bezüglich des Vorzeichens der

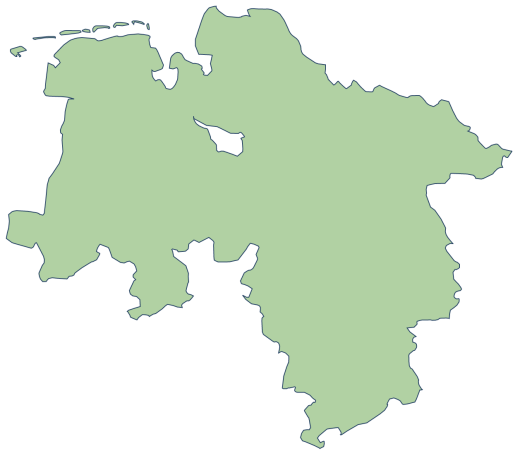

Abbildung 2. Niedersachsen aus dem Datensatz

Fläche und der Definierung der Eckpunkte im Uhrzeigersinn hält. So gibt es Inseln die eine negative Fläche besitzen oder Löcher mit postivem Vorzeichen.
Aus diesem Grund werden erst einmal alle positiv Flächen behandelt. Zusätzlich muss eine neue Überprüfung stattfinden die überprüft ob ein Polygon ein Loch ist. Hierbei wird getestet ob ein Polygon in einem anderen Polygon vollständig enthalten ist.

A. *Sonderfall: Polygon in einem anderen Polygon*

Wie testet man ob ein Polygon P_{klein} in einem anderen Polygon $P_{groß}$ vollständig enthalten ist? Ein einfacher Ansatz ist es zu überprüfen ob sich alle Punkte des Polygons P_{klein} in $P_{groß}$ befinden.
Dies würde für die Aufgabe bei der Flächenberechnung der Bundesländer schon funktionieren.
Es gebe nur einen Sonderfall wie den in der Abbildung 3. Mit dem oben beschriebenen Ansatz würde P_{klein} trotzdem als Loch bezeichnet werden da alle Eckpunkte von P_{klein} in $P_{groß}$ befinden.

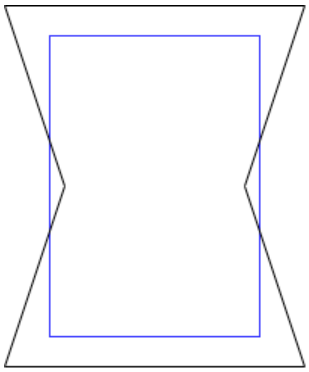

Abbildung 3. Ein Randfall in dem das blaue Polygon nicht im schwarzen liegt

Jetzt könnte man zusätzlich überprüfen ob sich alle Punkte von $P_{groß}$ außerhalb von P_{klein} befinden. Doch dies würde auch noch nicht alle Probleme wie man in Abbildung 4 sehen kann lösen.

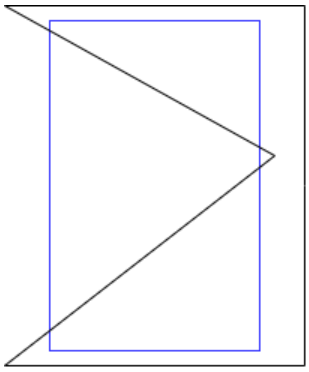

Abbildung 4. Ein Randfall in dem das blaue Polygon nicht im schwarzen liegt

Deshalb müsste man alle Kanten $P_{groß}$ und P_{klein} nach Schnittpunkten untersuchen.

IV. Überprüfung ob ein Punkt in einem Polygon liegt

Für die Überprüfung ob ein Punkt $\tilde{p}$ in einem Polygon P liegt wird zuerst Strecke aus dem Punkt $\tilde{p}$ zu einem festgelegtem Punkt $\tilde{q}$ definiert. Der Punkt $\tilde{q}$ muss außerhalb des Polygons P liegen. Dabei legt man den Punkt rechts vom Polygon P und auf der selben Höhe des Punktes $\tilde{p}$.
Um sicher zu gehen das sich ein Punkt rechts von einem Polygon P befindet, iteriert man über alle Eckpunkte des Polygons und sucht dabei den höchsten X Wert und addiert dann einen Faktor darauf.
Jetzt zählt man die Schnittpunkte nur mit echten Seitenwechseln zwischen der Strecke $[\tilde{p}\tilde{q}]$ und allen Kanten des Polygons P. Dabei startet man bei einem Eckpunkt p_{Start} der nicht auf $[\tilde{p}\tilde{q}]$ liegt. Ist die Anzahl ungerade liegt der Punkt p im Polygon P. Ist die Anzahl gerade liegt p außerhalb von P.
Um echte Seitenwechsel zu identifizieren wird muss folgende Bedingung wahr sein:

$$ccw(p_{i-1}, p_i, \tilde{p}) * ccw(p_{i-1}, p_i, \tilde{q}) \leq 0$$

Zusätzlich muss der Fall abgedeckt werden das der Punkt p auf der Kante des Polygons liegt. Hierbei iteriert man einfach über alle Kanten $[p_n, q_n]$.
Die Abbildung 5 zeigt die wichtigsten Fälle die abgedeckt werden müssen.
In folgender Tabelle werden die einzelnen Punkte/Fälle kurz beschrieben. So wird die Anzahl der Schnittpunkte, die Anzahl der echten Seitenwechsel und der Output des Algorithmus aufgelistet.

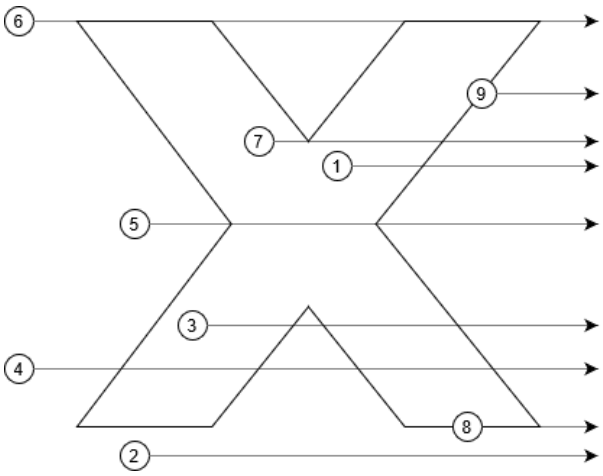

Abbildung 5. Verschiedene Fälle wie ein Punkt in einem Polygon liegen kann.

Punkt	Schnittpunkte	Seitenwechsel	Kante	Output
1	1	1	Nein	Inside
2	0	0	Nein	Outside
3	3	3	Nein	Inside
4	4	4	Nein	Outside
5	2	2	Nein	Outside
6	∞	0	Nein	Outside
7	2	1	Nein	Inside
8	∞	0	Ja	Inside
9	1	0	Ja	Inside

V. Überprüfung ob eine Stadt in einem Bundesland liegt

Um zu überprüfen ob eine Stadt in einem Bundesland liegt wird erst einmal nach allen Polygonen gesucht in denen der Punkt liegt. Ist die Anzahl gerade dann liegt der Punkt nicht im Bundesland und liegt damit in einem Loch. Ist die Zahl ungerade dann liegt das Polygon in keinem Loch. Für dieses Verfahren dürfen die Polygone sich nicht schneiden. So sollte aber dieser Algorithmus auch mit komplizierten Grenzverläufen wie zum Beispiel mit der Belgischen Exklave Baarle-Hertog zurechtkommen.

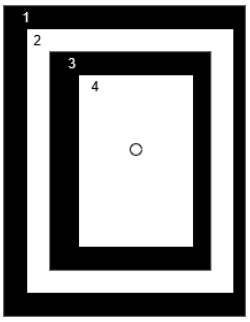

Abbildung 6. Loch in einem Polygon in einem Loch von einem Polygon

VI. Ergebnisse

Die Flächen der einzelnen Bundesländer:

Bundesland	Fläche
Thüringen	13725
Schleswig-Holstein	13456
Sachsen-Anhalt	17451
Sachsen	15668
Saarland	2180
Rheinland-Pfalz	16914
Nordrhein-Westfalen	28966
Niedersachsen	40634
Mecklenburg-Vorpommern	19659
Hessen	17977
Hamburg	633
Bremen	341
Brandenburg	25276
Berlin	766
Bayern	60026
Baden-Württemberg	30522
Deutschland	304194

Die Ergebnisse für die Städte die einzelnen Bundesländern zugewiesen werden:

Bundesland	Landeshauptstadt
Thüringen	Erfurt
Schleswig-Holstein	Kiel
Sachsen-Anhalt	Magdeburg
Sachsen	Dresden
Saarland	Saarbrücken
Rheinland-Pfalz	Mainz
Nordrhein-Westfalen	Düsseldorf
Niedersachsen	Hannover
Mecklenburg-Vorpommern	Schwerin
Hessen	Wiesbaden
Hamburg	Hamburg
Bremen	Bremen
Brandenburg	Potsdam
Berlin	Berlin
Bayern	München
Baden-Württemberg	Stuttgart

VII. Testen des Algorithmus

Eine Möglichkeit ist es die Fläche mit der echten Daten aus Wikipedia zu vergleichen.

Bundesland	Algorithmus	Wikipedia	Faktor
Thüringen	13725	16202	0.8471
Schleswig-Holstein	13456	15800	0.8516
Sachsen-Anhalt	17451	20456	0.8530
Sachsen	15668	18449	0.8492
Saarland	2180	2571	0.8479
Rheinland-Pfalz	16914	19858	0.8517
Nordrhein-Westfalen	28966	34112	0.8491
Niedersachsen	40634	47709	0.8517
Mecklenburg-Vorpommern	19659	23295	0.8439
Hessen	17977	21115	0.8513
Hamburg	633	755	0.8384
Bremen	341	419	0.8138
Brandenburg	25273	29654	0.8522
Berlin	768	891	0.8619
Bayern	60026	70541	0.8509
Baden-Württemberg	30522	35747	0.8538
Deutschland	304194	357580	0.8507

Im Grunde sich alle Verhältnisse sehr ähnlich. Ein grober Fehler ist erst einmal unwahrscheinlich. Diese Methode ist aber etwas zu ungenau um kleine Fehler zu entdecken. Dafür ist der Datensatz zu Fehlerhaft. So fehlen zum Beispiel einige innerdeutsche En- und Exklaven.
Deshalb wurden über 50 Junit Tests geschrieben um die Funktionalität des Codes zu überprüfen. Diese Tests decken vorallem die bisher angesprochenen Randfälle ab. Die Tests wurden an Kommilitonen weitergegeben und diese bestätigten deren Qualität.
Auch wurde jedes Bundesland genau untersucht. So wurde überprüft ob Inseln und Exklaven addiert werden und Löcher subtrahiert werden.
Die korrekte Zuordnung der Landeshauptsädte

VIII. Komplexität

- Die Komplexitätsklasse für das Einlesen der Daten liegt ist $\mathcal{O}(n)$, wobei n die Anzahl der Datenpunkte beschreibt.
- Die Komplexitätsklasse für das Berechnen der Fläche eines Polygon P ist $\mathcal{O}(n)$, wobei n die Anzahl der Eckpunkte des Polygons P beschreibt. Die oben durchgeführten Optimierungen verändern nichts an der Komplexitätsklasse sondern beschleunigen nur die Operationen selbst.
- Die Komplexitätsklasse für das Berechnen ob ein Punkt $\tilde{p}$ in einem Polygon P liegt ist $\mathcal{O}(n)$, wobei n die Anzahl der Eckpunkte des Polygons P beschreibt.
- Die Komplexitätsklasse für das Berechnen ob ein Polygon P_1 in einem Polygon P_2 liegt ist $\mathcal{O}(n^2)$. Dies entsteht da alle Kanten von P_1 mit allen Kanten von P_2 verglichen werden müssen. Dieser Vergleich alleine hat im Worstcase die Komplexität $\mathcal{O}(m * n)$ wobei m und n die Anzahl der Eckpunkte der Polygone P_1 und P_2 beschreibt. Zusätzlich muss noch getestet werden ob ein Punkte von P_1 in P_2 liegt was die Komplexität $\mathcal{O}(n)$ besitzt. Es muss nur ein Punkt getestet werden da schon überprüft wird ob sich die Kanten schneiden. Somit ist die Komplexität $\mathcal{O}(n^2+n)$ was trotzdem in der Komplexitätsklasse $\mathcal{O}(n^2)$ liegt.
- Die Komplexitätsklasse für das Berechnen wie viele Polygone aus einer Menge $\{P_1, P_2, ..., P_n\}$ in einander liegen ist $\mathcal{O}(n^2)$, wobei n die Anzahl der Polygone beschreibt. Dies würde sich theoretisch optimieren lassen. Da aber die Menge der Polygone in der Aufgabe nie wirklich groß ist wäre der Aufwand für einer effizienteren Implementierung unverhältnismäßig groß und würde keine oder nur kleine Verbesserungen bringen.

Zusammengefasst kann man sagen das die je nach Aufgaben die Brechungen in der linearen oder in der quadratischen Komplexitätsklasse liegen.

Literatur

[1] https://www.wikiwand.com/de/Gaußsche_Trapezformel
[2] http://www.dcs.gla.ac.uk/ pat/52233/slides/Geometry1x1.pdf

IX. Anhang

Berechnung der Fläche eines Polygons:

```
  /**
 * calculate the Area of this Polygon
 * @return double
 */
 public double calculateArea(){
   if(!isClosed())
     throw new IllegalArgumentException("Polygon is not Closed");

   double area = 0;
   for(int counter = 1 ; counter < cords.size()-1; counter++)
     area += gausstriangle( cords.get(counter), cords.get(counter-1),
           cords.get(counter+1));

   area += gausstriangle( cords.get(0), cords.get(cords.size()-2), cords.get(1));
   return area/2;
 }


 private double gausstriangle(Point a, Point b, Point c){
   return a.getY()*(b.getX()-c.getX());
 }
```

Überprüfung ob Polygon P_{this} in Polygon P_{that} liegt:

```
/**
* Check if This is inside That
* @param that
* @return boolean
*/
public boolean isPolygonInside(Polygon that){
  if( !isClosed() || !that.isClosed() )
    throw new IllegalArgumentException("Polygon is not Closed");

  for(int thisCounter = 1; thisCounter < this.cords.size(); thisCounter++){
    final Line2Points thisLine = new Line2Points(this.cords.get(thisCounter-1),
          this.cords.get(thisCounter));
    for(int thatCounter = 1; thatCounter < that.cords.size(); thatCounter++){
      final Line2Points thatLine = new Line2Points(that.cords.get(thatCounter-1),
            that.cords.get(thatCounter));
      if(thisLine.isIntersecting(thatLine)){
        return false; // Is intersecting so this can not be in that
      }
    }
  }

  return that.isPointInside(this.getCords().get(0)); //check if one Point of this
        is in that
}
```

Berechnung der Fläche eines Bundeslandes:

```
  public double calculateArea(){
    double sum = 0;
    for(Polygon p : areas){
      boolean isInside = false;
      for(Polygon p2 : areas){
        //Check if Hole
        if(!p.equals(p2) && p.isPolygonInside(p2) ){
          isInside = true;
          break;
        }
      }
      if(isInside)
        sum -= Math.abs(p.calculateArea());
      else
        sum += Math.abs(p.calculateArea());
    }
    return sum;
  }
```

Berechnung ob ein Punkt/Stadt in einem Bundeslandes liegt:

```java
/**
* Polygone muessen vollstaendige Loecher voneinander sein. Sie duerfen sich nicht
      schneiden.
* @param p Punkt
* @return Ob Punkt in Polygon oder im Loch ist.
*/
public boolean isPointInside(Point p){
  final List<Polygon> relevantPolygons = areas.stream()
  .filter(a -> a.isPointInside(p))
  .collect(Collectors.toList()); // all Polygons containing P


  return relevantPolygons.size()%2==1;
}
```

Berechnung ob ein Punkt in einem Polygon liegt:

```java
public boolean isPointInside(Point p){
  if(!isClosed())
    throw new IllegalArgumentException("Polygon is not Closed");
  //Check if Point lies on an edge
  final Line2Points pointLine = new Line2Points(p,p); // generate Dummy Line to
        check if it does intersect with Edges
  for(int counter = 1; counter < cords.size() ; counter++){
    if(pointLine.isIntersecting(new
          Line2Points(cords.get(counter-1),cords.get(counter))))
    return true;
  }

  final double maxXValue = cords.stream()
    .mapToDouble(c -> c.getX())
    .reduce(Double::max)
    .orElseThrow(IllegalStateException::new);

  if(p.getX() > maxXValue)
    return false;

  final Point extreme = new Point(maxXValue+1, p.getY());

  int startCounter = 0;
  while(ccw(extreme, p, cords.get(startCounter)) == 0)
    startCounter++;

  int intersections = 0;
  double lr = orientation(extreme, p, cords.get(startCounter));
  for(int counter = startCounter+1; counter < cords.size() ; counter++){
    final double lrNew = orientation(extreme, p, cords.get(counter));
    if( Math.abs( lrNew - lr ) == 2){
      lr = lrNew;
      double o1 = ccw(cords.get(counter-1), cords.get(counter), extreme);
      double o2 = ccw(cords.get(counter-1), cords.get(counter), p);
      if(o1*o2<=0){
        intersections++;
      }
    }
  }
  for(int counter = 1; counter <= startCounter ; counter++){
    final double lrNew = orientation(extreme, p, cords.get(counter));
    if( Math.abs( lrNew - lr ) == 2){
      lr = lrNew;
      double o1 = ccw(cords.get(counter-1), cords.get(counter), extreme);
      double o2 = ccw(cords.get(counter-1), cords.get(counter), p);
      if(o1*o2<=0)
        intersections++;
    }
  }
  return intersections % 2 == 1; // return true if count is odd, false otherwise
}
```