Computational Geometry - Abgabe 3

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

***Zusammenfassung*—**

I. Basis Bentley-Ottmann Algorithmus

PSEUDOCODE

```
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;
}
```

Zu Beginn werden alle Start und Endpunkte aller Strecken $L_{InputStrecken}$ in eine EventQueue Q_{Event} eingefügt. Diese EventQueue ist eine PriorityQueue die intern die Events auf der X-Achse und Eventtyp sortiert. *START* Events haben bei gleichem X Wert Vorrang. *END* Events sind immer zuletzt dran. Die PriorityQueue ist als ein Heap implementiert. Die Operationen besitzen deshalb auch die Komplexität [1]:

- add: $\mathcal{O}(log(n))$
- poll: $\mathcal{O}(1)$

Relevante Strecken liegen in der Sweepline T_{Sweep}. Für die Sweepline T_{Sweep} wird als Datenstruktur eine Baum basierend auf einer Rot-Schwarz-Baum Implementierung verwendet. Dieser hat die Komplexität [2]:

- add: $\mathcal{O}(log(n))$
- remove: $\mathcal{O}(log(n))$
- contains: $\mathcal{O}(log(n))$
- get: $\mathcal{O}(log(n))$

Es gibt im regulären Bentley-Ottmann Algorithmus 3 Events (*START*, *END*, *INTERSECTION*). Zu beginn liegen nur *START* und *END* Events in Q_{Event}.
Nun wird nacheinander ein Event aus Q_{Event} genommen. Dieses Event wird je nach Typ behandelt.
Das Event *START* fügt das neue Segment S_{new} in die Sweepline T_{Sweep} ein. Es wird überprüft ob ein Segment über oder unter S_{new} liegt. Sollte dies der Fall sein wird überprüft ob diese sich mit S_{new} schneiden. Bei einem gefundenen Schnittpunkten wird ein neues *INTERSECTION* Event in T_{Sweep} eingefügt.

```
{}
```

Bei einem *END* Event endet ein Segment S_{old}. Es wird überprüft ob ein Segment über und unter S_{old} liegt. Sollten diese existieren dann wird überprüft ob diese sich schneiden. Bei einem gefundenen Schnittpunkten wird ein neues *INTERSECTION* Event in Q_{Event} eingefügt. S_{old} wird aus der Sweepline T_{Sweep} entfernt.

```
{}
```

Bei einem *INTERSECTION* Event.....

```
{}
```

II. Schnittpunkt zwischen 2 Geraden

Da es nicht reicht nur zu Wissen das zwei geraden einen Schnittpunkt haben sondern es auch nötig ist auch dessen genaue Position zu kennen musste ein neuer ALgortithmus entwickelt werden.
Hierfür wurde erstmal überprüft ob der Start- oder Endpunkt p1,q1 der Strecke s1 identisch zu einem der Punkt p2 oder q2 der Strecke s2 ist. Sollte dies der Fall sein wird direkt der gemeinsame Punkt zurückgegeben.
Nun werden die beiden Strecken als Geraden behandelt. So kann man mit der Geradengleichung den Schnittpunkt der zwei Geraden berechnen.
Es besteht die Möglichkeit das der Schnittpunkt S nicht existiert da beide Geraden parallel zu einander sind. In diesem Fall wären wären x und y von S unendlich. In diesem Fall wird ein Optional.empty() zurückgegeben. Da der Schnittpunkt S über die Geradengleichung berechnet worden ist kann es sein das dieser sich auf den Geraden von s1 und s2 befindet aber nicht auf der Strecke s1 oder s2. Deshalb muss überprüft werden ob sich der Punkt in der Bounding Box der beiden Strecken befindet. Sollte dies bei beiden der Fall sein dann ist dies der korrekte Schnittpunkt.
Um numerische Fehler zu reduzieren wird bei Horizontalen und Verticalen Strecken ....

III. Probleme für den Algoritmus

Der Algorithmus hat Probleme wenn folgende Voraussetzungen nicht erfüllt sind:

1) x-Koordinaten der Schnitt- und Endpunkte sind paarweise verschieden
2) Länge der Segmente > 0
3) nur echte Schnittpunkte
4) keine Linien parallel zur y-Achse
5) keine Mehrfachschnittpunkte
6) keine überlappenden Segment

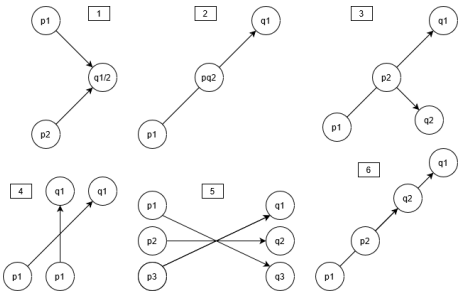

Abbildung 1. Problemfälle für den Standard Bentley Ottman algorithmus

IV. Behandlung der Sonderfälle

In meiner Implementierung werden alle Sonderfälle behandelt.

A. Nur echte Schnittpunkte

Meine Implementierung unterstützt auch unechte Schnittpunkte. Hierfür wurde vor allem die Sweepline angepasst. So wird die gewöhnliche Sweepline Implementierung die nur aus einem Baum T_{Sweep} mit Strecken besteht mit einer Funktionalität ergänzt die ähnlich wie bei Buckets bei einem Hashset funktioniert. So wird bei START Events überprüft ob die Position des Startpunktes in T_{Sweep} bereits existiert. Sollte dies der Fall sein wird das Segment einfach zusätzlich in den Knoten eingefügt. Der Schlüssel der Knoten ist der aktuelle Y-Wert.

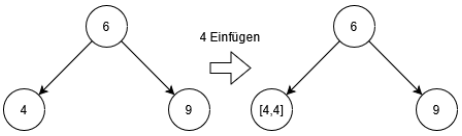

Abbildung 2. Einfügen in die Sweepline mit Kollision

Leider müssen werden jedes mal wenn die Sweepline bewegt wird die Elemente in T_{Sweep} zu neu sortiert werden. Operationen auf T_{Sweep} laufen in $\mathcal{O}(n)$.

B. X-Koordinaten der Schnitt- und Endpunkte sind paarweise identisch

Dieses Problem ist ein Teilproblem des vorherigen Problems und damit schon gelöst. So wird beim einfügen neuer Strecken überprüft ob bereits andere Segmente an diesem Punkt liegen. Sollte dies der Fall sein werden wird dieser Punkt entsprechend der Anzahl der Segmente als Schnittpunkte in die Output Liste eingefügt. Außerdem immer wenn ein Schnittpunkt gefunden wird welcher direkt auf der Sweepline T_{Sweep} liegt wird dieser ohne ein *INTERSECTION* Event zu generieren der Outputliste hinzugefügt.

C. Linien parallel zur Y-Achse

Für Linien die zur Y-Achse gehören wurden in ein neues *VERTICALLINE* Event erschaffen. So werden keine *Start* und *END* Event in die Eventqueue Q_{Event} eingefügt sondern nur das *VERTICALLINE* Event. Vertikale Strecken schneiden sich mit allen Strecken, die aktuell die Sweepline zwischen dem Start- und Endpunkt der vertikalen Strecke schneiden. So muss nicht die gesamte Sweepline T_{Sweep} untersucht werden sondern nur ein Teil des Baumes. Das gilt selbstverständlich auch

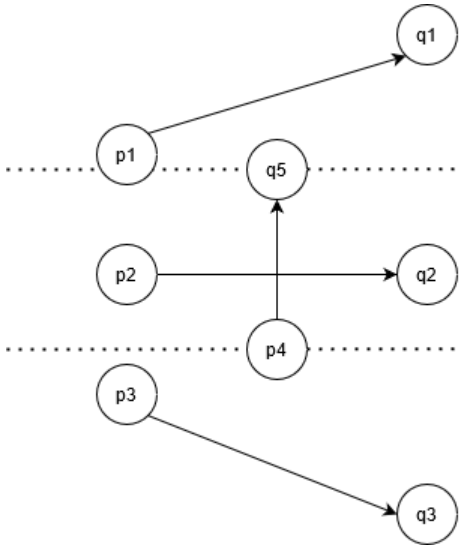

Abbildung 3. Suche nach Schnittpunkten mit Vertikalen Strecken in einem eingeschränkten Bereich

für anderen Vertikalen Strecken an aktueller X-Stelle. Deshalb wird eine vertikale Strecke auch an einem *VERTICALLINE* Event in die Sweepline T_{Sweep} eingefügt. Sie werden aber nicht in den Baum eingefügt sondern in eine separate Liste für vertikale Segmente. Sobald die Sweepline verschoben wird werden alle vertikalen Strecken aus der Liste entfernt.

D. Länge der Segmente gleich 0

Elemente der Länge 0 werden einfach als vertikale Segmente behandelt.

E. Mehrfachschnittpunkte

Mehrfachschnittpunkte wurden behandelt. Hierfür wird die oben beschriebene Baumstruktur der Sweepline T_{Sweep} genutzt. So wird an einem *INTERSECTION* Event in der SweepLine gesucht ob an der Stelle weitere Segmente durchlaufen. Sollte dies der Fall sein wird werden direkt die korrekte menge an Schnittpunkten der Outputliste hinzugefügt. Wichtig ist aber das der Fall in der Abbildung 4 betrachtet wird.

F. Überlappende Segmente

Auch dieses Problem ist auch ein Teilproblem der nur echten Schnittpunkte. Dies kann aber noch mehr Probleme bereiten als das nur Überlappungen in einzelnen Punkten. Wichtig ist das die Funktion die Schnittpunkte zwischen 2 Strecken findet in der Lage ist trotz Überlappung einen Schnittpunkt findet. Da eigentlich überlappende Segmente unendlich viele Schnittpunkte hat wird einfach der untere linke

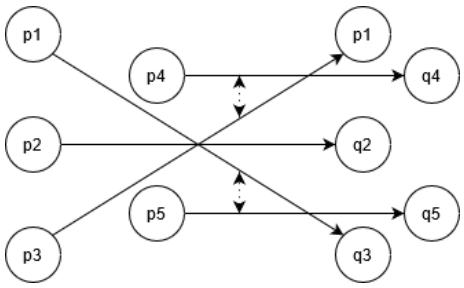

Abbildung 4. Problemfälle mit überlappenden Elementen

Punkt gewählt.
Die wichtigsten Fälle in Abbildung 2: Schwierig ist vor allem

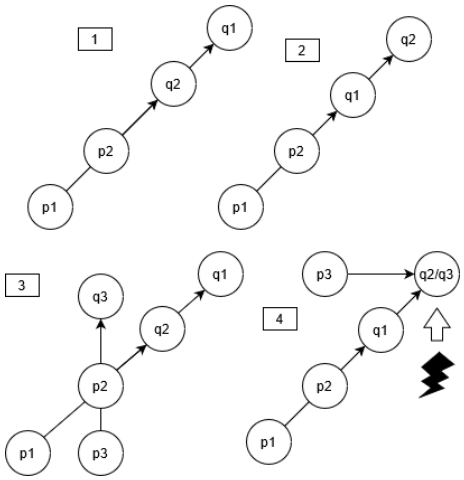

Abbildung 5. Problemfälle mit überlappenden Elementen

der Fall 4. Hier muss am Endpunkt $q1$ der Schnittpunkt $q2/q3$ gefunden werden.

V. Komplexität

Die Komplexität ist abhängig vom Input. So gibt es normalerweise $2n + k$ Events, wobei n die Anzahl der Segmente und k die Anzahl der Schnittpunkten entspricht. Wobei bei Sonderfälle die Anzahl geringer Ausfallen kann. Da bei vertikalen Strecken nur ein Punkt in die Eventqueue eingefügt wird. Das selbe gilt auch für Schnittpunkte die in einem Startpunkt liegen und keinen neuen Eintrag in die EventQueue bekommen. So gilt für m nicht vertikale Strecken, s Schnittpunkte ohne Überschneidungen mit anderen Events und v vertikalen Strecken:

$$2n + k \geq 2m + s + v$$

Die Komplexität des Sortierens der Eventqueue beträgt $\mathcal{O}((2m + v) * log(2m + v))$ was trotzdem $\mathcal{O}(n * log(n))$ entspricht.
Operationen in die EventQueue und der Sweepline haben die Komplexität $\mathcal{O}(log(m))$. Allerdings wird bei jedem bewegen der Sweepline die Sweepline neu berechnet. Dies resultiert leider in $\mathcal{O}(m * log(m))$.

Datei	BF S C	BF S L	BF P C	BF P L	BO
s_1000_1	18	20	37	57	57
s_1000_10	13	13	1	2	14
s_10000_1	1263	1423	136	128	43
s_100000_1	111624	121718	8712	8825	45893

Zusammengefasst kann man den Aufwand wie folgt beschreiben:

$$\mathcal{O}(m * log(m)) + m * \mathcal{O}(m * log(m))$$

VI. Performance

1) **BF S C:** BruteForce, SingleThread, nur Anzahl Schnittpunkte
2) **BF S L:** BruteForce, SingleThread, Liste von Schnittpunkten
3) **BF P C:** BruteForce, MultiThread, nur Anzahl Schnittpunkte
4) **BF P L:** BruteForce, MultiThread, Liste von Schnittpunkten
5) **BO:** Bentley-Ottmann, Liste von Schnittpunkten

Auch noch test mit 4Kerner?

VII. Durchschnittliche Sweeplinegrösse

Die durchschnittliche Füllgrad und die Anzahl der Verschiebungen der Sweepline beeinflusst die Performance maßgeblich. Die Anzahl der Schnittpunkte beeinflusst meistens auch die Anzahl der Verschiebungen der Sweepline. Dies kann man gut sehen wenn man die Zeiten in der oberen Tabelle mit der Abbildung 6 vergleicht.

Literatur

[1] https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html
[2] https://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html

VIII. Anhang

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;
}
```

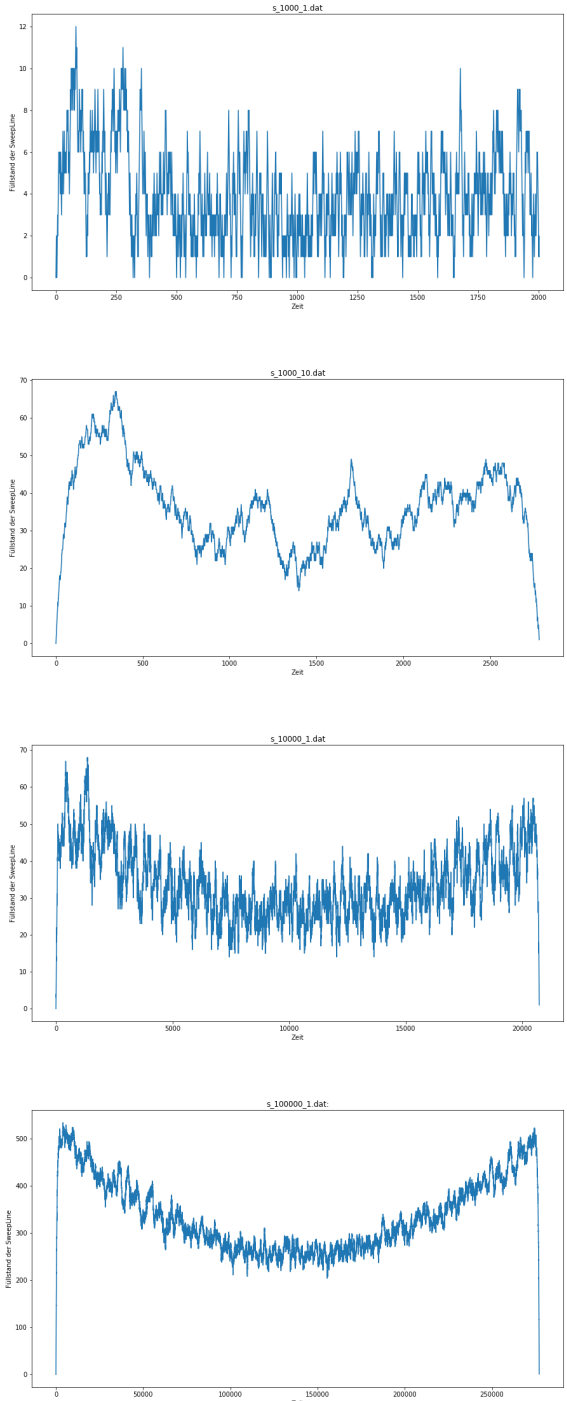

Abbildung 6. Füllstand der Sweepline über die Zeit