Computational Geometry - Abgabe 1

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

I. Berechnung ob zwei Strecken sich schneiden

Es gibt verschiedene Möglichkeiten zu überprüfen ob sich zwei Geraden schneiden. Bei Strecken ist dies etwas komplexer aber auch einfach möglich.
Dazu betrachtet man die Orientierung der Start- und Endpunkte der beiden Strecken auf einer Ebene.

II. Orientierung von drei Punkten in einer Ebene

Drei Punkte in einer Ebene können in 3 verschiedenen Arten zueinander stehen:

- Im **Uhrzeigersinn**
- **Gegen den Uhrzeigersinn**
- **Kollinear**

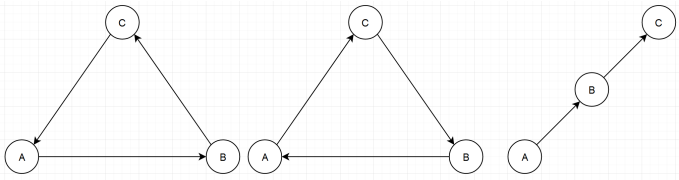

Abbildung 1. Die drei verschiedenen Orientierungen

Die Abbildung 1 zeigt die Verschiedenen Orientierungen. Für die Berechnung der Orientierung wird der CCW verwendet:

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

$$ccw = p_1 * q_2 - p_2 * q_1 + q_1 * r_2 - q_2 * r_1 + p_2 * r_1 - p_1 * r_2$$

$$ccw(p,q,r) \begin{cases} < 0 & \text{r liegt rechts von [p,q]} \\ = 0 & \text{r liegt auf Strahl [p,q]} \\ > 0 & \text{r liegt links von [p,q]} \end{cases}$$

III. Orientierung der zwei Strecken

Die zwei Strecken mit den Punkten (p_1, q_1) und (p_2, q_2) schneiden sich wenn der normale Fall oder der Spezialfall bei Kollinearität zutrifft. Sollte keiner der beiden Fälle zutreffen dann schneiden sich beide Strecken nicht.

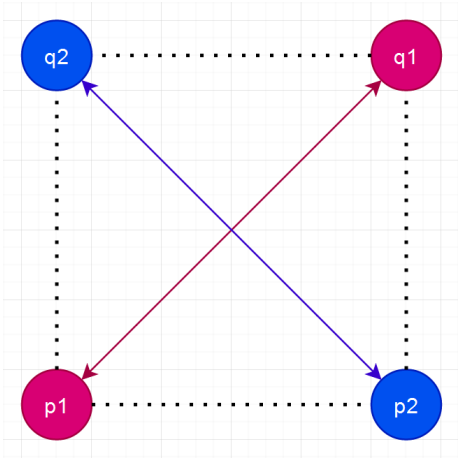

Abbildung 2. Zwei Strecken die sich schneiden

A. Genereller Fall

Wenn sich die Orientierungen von (p_1, q_1, p_2) und (p_1, q_1, q_2) unterscheiden und dann die Orientierungen von (p_2, q_2, p_1) und (p_2, q_2, q_1) sich auch unterscheiden, dann müssen sich die beiden Strecken schneiden.

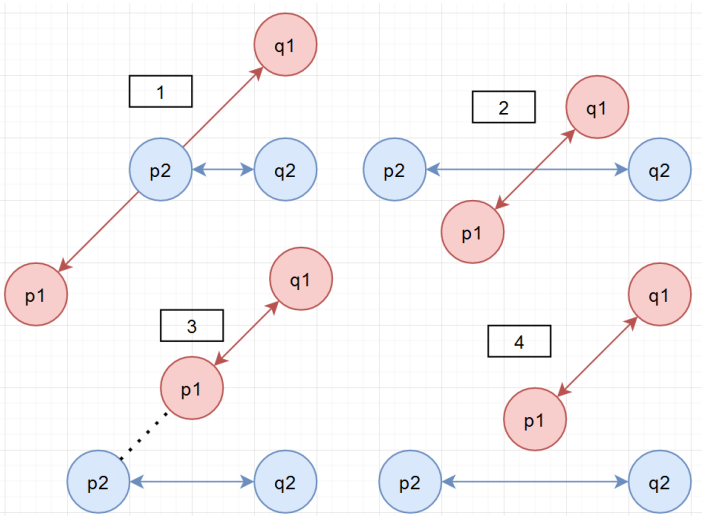

Abbildung 3. Alle generellen Fälle

1) Für diesen Fall unterscheiden sich (p_1, q_1, p_2), (p_1, q_1, q_2) und (p_2, q_2, p_1), (p_2, q_2, q_1) => Sie schneiden sich.
2) Für diesen Fall unterscheiden sich (p_1, q_1, p_2), (p_1, q_1, q_2) und (p_2, q_2, p_1), (p_2, q_2, q_1) => Sie schneiden sich.
3) Für diesen Fall unterscheiden sich (p_1, q_1, p_2), (p_1, q_1, q_2) aber (p_2, q_2, p_1), (p_2, q_2, q_1) sind gleich => Sie schneiden sich nicht.
4) Für diesen Fall unterscheiden sich (p_1, q_1, p_2), (p_1, q_1, q_2) aber (p_2, q_2, p_1), (p_2, q_2, q_1) sind gleich => Sie schneiden sich nicht.

B. Spezial Fall bei Kollinearität

Wenn (p_1, q_1, p_2), (p_1, q_1, q_2), (p_2, q_2, p_1) und (p_2, q_2, q_1) alle gleich 0 sind, dann sind beide Strecken kollinear. Beide Strecken liegen auf einem Strahl. Jetzt muss noch überprüft werden ob sich die Beiden mindestens in einem Punkt überschneiden.
Hierfür werden 4 Prüfungen durchgeführt. Es wurde überprüft ob p_2 auf der Strecke $p_1, q_1)$, q_2 auf (p_1, q_1), p_1 auf (p_2, q_2) und q_1 auf (p_2, q_2) liegt. Sollte einer der Punkte auf einer der Strecken liegen gibt es einen Schnittpunkt.

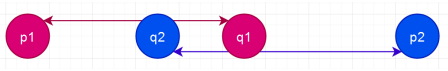

Abbildung 4. Spezialfall bei Kolinearität und Überlappung

Abbildung 5. Spezialfall bei Kolinearität ohne sich zu schneiden

C. Berechnung ob Strecken sich Überschneiden

Auch hier werden vier Test durchgeführt. Es wird getestet:

1) ob p2 zwischen p1 und q1 liegt,
2) ob q2 zwischen p1 und q1 liegt,
3) ob p1 zwischen p2 und q2 liegt,
4) ob q1 zwischen p2 und q2 liegt.

Sollte einer dieser Fälle zutreffen dann Überlappen sich die beiden Strecken.

IV. Ergebnisse

Die Ergebnisse für die einzelnen Dateien sind:

Datei	Schneidende Strecken
s_1000_1.dat	11
s_10000_1.dat	732
s_100000_1.dat	77126

V. Testen des Algorithmus

Für die Korrektheit des Program's entwickelte ich 19 Testfälle die weitestgehend alle Fälle abdecken sollten. Es wurde nicht systematisch gegen numerischer Stabilität getestet.

Zusätzlich wurde mit einer Vielzahl von Kommilitonen die Zahl der schneidenden Strecken verglichen. Hierbei war die Anzahl identisch. Es ist nicht auszuschließen das alle den selben Fehler gemacht haben.

VI. Ausgleich von Ungenauigkeiten bei Berechnungen mit dem Datentyp Double

Um Fehler bei Berechnungen mit Double Werten auszugleichen wurde ein kleiner Threshold der Größe $0.0000000001d$ eingebaut. Da in diesem Programm alle Überprüfungen nahe 0 sind ist auch die Genauigkeit der Double Werten sehr hoch. Es wurde auch getestet ob dieser Threshold selbst eine Fehlerquelle ist.

VII. Laufzeitverbesserung des Algorithmus

Um die Laufzeit des Programm's zu verbessern wurden zwei Wege untersucht.

A. Optimierung der Komplexität des Algorithmus

Ein naiver Ansatz würde alle Strecken jeweils zweimal vergleichen. So würde mit zwei *for-Schleifen* einmal zwei Strecken AB verglichen werden und später nochmal BA. Ein Vergleich einer Strecke mit sich selbst wird nicht durchgeführt da diese sich ohnehin schneiden.
Hier gibt es Verbesserungspotential. So kann man die zweite *for-Schleife* mit dem aktuellen Index der äußeren Schleife um Eins erhöht beginnen lassen. Den letzten äußeren Schleifendurchgang kann man sich ebenfalls sparen da hier bereits alle Segmente verglichen wurden.

```
int counter = 0;
for(int m = 0 ; m < size-1 ; m++){
  for(int n = m+1 ; n < size; n++){
    if(lines.get(n).isIntersecting(lines.get(m))){
      counter++;
    }
  }
}
```

Bei jedem äußeren Schleifendurchgang spart man sich einen weiteren inneren Durchlauf. Man kann die Anzahl der Durchläufe mit der Gaußschen Summenformel beschreiben:

$$\sum_{k=1}^{n} k = 1 + 2 + 3 + 4 + 5 + ... + n = (n^2 + n)/2$$

Der letzte Durchlauf kann weggelassen werden:

$$\sum_{k=1}^{n-1} k = 1 + 2 + 3 + 4 + 5 + ... + (n-1) = (n^2 - n)/2$$

Es sollte gelten:

$$\mathcal{O}(n^2) \geq \mathcal{O}(\frac{(n^2-n)}{2})$$

Selbst nach dieser Optimierung liegt das Problem noch immer in der gleichen Komplexitätsklasse $\mathcal{O}(n^2)$.

B. Parallelisierung

Um die Laufzeit noch weiter zu verbessern wurde eine Parallelisierung untersucht.
Hierbei konnte man ganz einfach den Code parallelisieren indem man Vorteile der Funktionalen Programmierung nutzt. Es werden einfach die gesamte Menge der zu vergleichenden Strecken auf m Kerne verteilt. Die Ergebnisse der einzelnen Threads werden zusammenaddiert. Der Anteil des sinkt bei größeren Problemen zunehmend. Bei kleinen Problemen lohnt sich die Parallelisierung wegen des großen Overheads nicht. Getestet wurde auf auf einem 6 Kern Prozessor mit Hyperthreading. Erwartungsgemäß erhaltet man einen Speedup von etwa 8 bei größeren Problemen wo der Overhead keine große Rolle mehr spielt.

Datei	Single	Parallelisiert	Speedup
s_1000_10.dat	64	153	0.42
s_1000_1.dat	22	29	0.76
s_10000_1.dat	1509	188	8.02
s_100000_1.dat	127037	15713	8.01

LITERATUR

[1] http://www.dcs.gla.ac.uk/ pat/52233/slides/Geometry1x1.pdf

VIII. ANHANG

Berechnung ob zwei Strecken sich schneiden:

```
public boolean isIntersecting(
    Line2Points that){
  final Point start1 = this.start;
  final Point end1 = this.end;
  final Point start2 = that.start;
  final Point end2 = that.end;

  final int o1 = orientation(start1,
     end1, start2);
  final int o2 = orientation(start1,
     end1, end2);
  final int o3 = orientation(start2,
     end2, start1);
  final int o4 = orientation(start2,
     end2, end1);

  if (o1 != o2 && o3 != o4)// General
     Interesction case
  return true;

  if (o1 == 0 && o2 == 0 && o3 == 0 &&
     o3 == 0){// If the segments are
     colinear -> check for overlap
    return onSegment(start1, start2,
       end1) || onSegment(start1, end2,
       end1) || onSegment(start2,
       start1, end2) ||
       onSegment(start2, end1, end2);
  }

  return false; // Doesn't fall in any
     in general or special cases -> No
     Intersection
}
```

Berechnung der Orientierung von drei Punkten:

```
/**
* Find orientation of p, q, r.
* @param p Point
* @param q Point
* @param r Point
* @return 0 -> p, q and r are colinear,
   1 -> Clockwise, 2 -> Counterclockwise
*/
private int orientation(Point p, Point
   q, Point r){
  final double ccw = p.getX()*q.getY() -
     p.getY()*q.getX() +
     q.getX()*r.getY() -
     q.getY()*r.getX() +
     p.getY()*r.getX() -
     p.getX()*r.getY();

  if(Tool.compareDouble(ccw, 0))
    return 0; // colinear
  else if(ccw > 0)
    return 1; //clockwise
  else
    return 2; //counterclock
}
```

Berechnung ob ein Punkt auf einer Strecke liegt:

```
// Given three colinear points p, r, q,
   the function checks if point r lies
   on line segment 'pq'
private boolean onSegment(Point p, Point
   r, Point q){
  return r.getX() <= Math.max(p.getX(),
     q.getX()) &&
  r.getX() >= Math.min(p.getX(),
     q.getX()) &&
  r.getY() <= Math.max(p.getY(),
     q.getY()) &&
  r.getY() >= Math.min(p.getY(),
     q.getY());
}
```