Aufgabe 3: Dreiecke zählen

Team: “throw new Exception();”

Einsendenummer: ???

26. November 2017

**Inhaltsverzeichnis**

[Lösungsidee 1](#_Toc492592180)

[Umsetzung 1](#_Toc492592181)

[Beispiele 3](#_Toc492592182)

[Quellcode 3](#_Toc492592183)

# Lösungsidee

Zur Lösung dieser Aufgabe gibt es eine sehr einfache Idee:

Es werden alle Kombinationen von genau drei eingegebenen Strecken getestet und für jedes Tripel überprüft, ob alle möglichen Paare dieser drei Strecken einen gemeinsamen Punkt haben. (Dabei ist zu beachten, dass trotzdem dies von der Aufgabenstellung ausgeschlossen wurde, Schnittpunkte von mehr als zwei Strecken existieren. Dreiecke, bei denen Eckpunkte identisch sind, werden nicht gewertet.) Sollte das der Fall sein, kann daraus direkt geschlussfolgert werden, dass die aktuell ausgewählte Kombination von Strecken ein Dreieck umschließt. Die Eckpunkte dieses Dreiecks sind die Schnittpunkte der Strecken miteinander.

# Umsetzung

Die Umsetzung dieser Idee erforderte zunächst einige grundlegende Implementationen.

Eine Strecke wurde durch die Parameter einer Geradengleichung (a, b, c) und die Maximal-/Minimalwerte für x und y definiert. Ich habe mich für die Parameterdarstellung entschieden, weil dadurch Strecken, welche parallel zur y-Achse verlaufen, ebenfalls, ohne Ausnahmeregelungen, darstellen lassen. Dies wäre in der Normalform nicht möglich gewesen. Die einzelnen Parameter wurden folgendermaßen berechnet:

Außerdem wurden die x und y-Werte der Koordinaten jeweils in ein eindimensionales, zweistelliges Array geschrieben, wobei an Stelle 0 immer der kleinere der Werte steht. Somit ist klar, dass alle jene Punkte zur Strecke zählen, die einerseits die o.g. Gleichung erfüllen und anderseits innerhalb der soeben definierten Bereiche liegen.

Des Weiteren musste überprüft werden, ob sich zwei Strecken schneiden (wenn ja, wird ebenfalls der Schnittpunkt benötigt).

Da die Lösung eines Gleichungssystems mit Geradengleichungen in Koordinatenform nur funktioniert, wenn keiner der beiden b-Paramerter 0 ist, mussten diese Fälle direkt ausgeschlossen werden. Sollten beide dieser Parameter den Wert 0 annehmen, so wird ein ungültiger Punkt (ein Array mit nur einer Stelle) zurückgegeben. Sollte das für nur einen der beiden Parameter zutreffen, besteht immernoch die Möglichkeit, dass ein gemeinsamer Punkt existiert. Somit wird der y-Wert der Funktion (Strecke mit ) für den x-Wert der senkrechten Strecke () berechnet. Wenn dieser Wert nicht existiert, wird absichtlich ein Wert außerhalb des Wertebereiches der Funktion zurückgegeben. Anschließend wird überprüft, ob der erhaltene Wert innerhalb des genanneten Wertebereiches liegt. Sollte das so sein, so werden die beiden Werte x und y in einem Array als Schnittpunkt zurückgegeben. Wenn das nicht so ist, ist der Output ein ungültiger Punkt.

Eine weitere Problemquelle sind parallele Strecken, da sich diese ebenfalls nicht schneiden. Also wird für alle Strecken, bei denen sowohl a, als auch b identisch sind, ebenfalls ein ungültiger Punkt zurückgegeben. Der Fall der identischen Strecken / Streckenabschnitte wurde bereits in der Aufgabenstellung ausgeschlossen und wird somit auch nicht weiter behandelt.

Der wichtigste Fall ist jedoch, wenn es sich weder um Parallelen, noch um senkrechte Strecken handelt. Sollte das der Fall sein, muss zunächst der Schnittpunkt der beiden Geraden ermittelt werden und anschließend eine Überprüfung stattfinden, ob dieser Punkt auch Teil beider Strecken ist.

Die Koordinaten des potentiellen Schnittpunktes können über einfaches Umstellen folgender Gleichungen ermittelt werden:

Nach der Berechnung des x-Wertes, kann für eine der beiden Funktionen deren Wert an dieser Stelle ermtittelt werden. Es folgt eine Überprüfung, ob das ermittelte Wertepaar innerhalb aller Grenzen der beiden Strecken liegt. Das bedeutet der x-Wert muss zwischen (bzw. auf) den Minimal- und Maximalwerten der beiden Funktionen liegen. Gleiches gilt für y. Sollte all das gegeben sein, so kann der Punkt letztendlich zurückgegeben werden. Wenn nicht, wird (wie bereits zuvor) ein unzulässiger Punkt ausgegeben.

Die mittlerweile bereits mehrfach erwähnte Methode zum Ermitteln der y-Koordinate einer Strecke für einen beliebigen x-Wert prüft zunächst, ob der x-Wert innerhalb des Definitionsbereiches der Strecke liegt und ob die Strecke nicht parallel zur y-Achse ist. Sollte beides zutreffen wird der y-Wert durch Umstellen in die Normalform für lineare Funktionen berechnet.

In der Programm-Klasse gibt es drei Methoden, auf deren Funktion ich nicht näher eingehen möchte:

1. eine Methode zum Konvertieren einer Datei in eine Liste von Strecken
2. eine Methode zum Ausgeben einer Liste von Dreiecken in die Konsole
3. die gleiche Methode für die zusätzliche Ausgabe in eine Textdatei

Sobald das Programm ausgeführt wird, wird mit Hilfe der o.g. Methode eine Liste mit allen zu betrachtenden Strecken erstellt. Außerdem wird eine weitere Liste deklariert, die im Folgenden alle gefundenen Dreiecke speichern soll, damit sie am Ende ausgegeben werden können. Dreiecke sind als Array von drei Punkten definiert, wobei Punkte Arrays mit zwei Dezimalbrüchen sind.

Nun werden alle möglichen Kombinationen von jeweils drei der eingelesenen Strecken getestet. Um nicht mehr zu prüfen als nötig, werden drei Zählschleifen ineinander geöffnet. Jede dieser Schleifen steht für eine Strecke. Dabei läuft die Äußerste (i) von null bis zur Anzahl der vorhandenen Strecken -3, die zweite (j) durchläuft dann alle Werte, die größer sind als i bis zur Anzahl der vorhandenen Strecken -2 und die letzte (k) verwendet alle Werte größer j, die kleiner sind als die Anzahl der vorhandenen Strecken (nullbasierte Inizes...).

Für alle Kombinationen werden die möglichen drei Schnittpunkte berechnet. Auf Grund guter Vorarbeit kann sehr schnell überprüft werden, ob die Punkte gültig sind, indem die Anzahl der enthaltenen Koordinaten gezählt werden. Wenn jeder Punkt zwei Koordinaten enthält, wird der eingangs erwähnten Liste der gefundenenen Dreiecke ein neues hinzügefügt, welches die gefundenen Schnittpunkte als Ecken hat.

Zum Schluss werden die gesammelten Dreiecke ausgegeben.

# Beispiele

Die vom BwInf vorgesehenen Beispiele wurden alle gelöst. Im Anhang befinden sich sowohl die Input-Dateien, als auch alle unbearbeiteten Outputs zu allen hier genannten Dateien.

Folgendes Ergebnis wurde erreicht:

* dreiecke1.txt: 6 Strecken / 9 Dreiecke
* dreiecke2.txt: 7 Strecken / 0 Dreiecke
* dreiecke3.txt: 9 Strecken / 3 Dreiecke
* dreiecke4.txt: 13 Strecken / 4 Dreiecke
* dreiecke5.txt: 18 Strecken / 1 Dreieck
* dreiecke6.txt: 20 Strecken / 19 Dreiecke

Es ist sofort erkennbar, dass KEIN direkter Zusammenhang zwischen der Anzahl der Strecken und der Anzahl der gefundenen Dreiecke besteht. Zum Beispiel werden in Datei 1 eineinhalb mal so viele Dreiecke wie Strecken gefunden, während in Datei 2 (trotz einer Strecke mehr) kein einziges vorhanden ist.

Zusätzlich habe ich eine Testdatei (dreiecke0.txt) erstellt, die genau drei Dreiecke enthält, die man aus den Werten ablesen kann. Diese Datei diente zum Testen / Debuggen des Programmes.

Drei der Strecken schneiden den Ursprung und die vierte schneidet diese drei Strecken in anderen Punkten. Somit ergeben sich drei Dreiecke.

# Quellcode

Das beschriebene Programm wurde in C# geschrieben. Die Projektdatei befindet sich im Ordner bearbeiteten Aufgabe.

In diesem Abschnitt gibt es drei wichtige Methoden bzw. Abschnitte, die betrachtet werden sollten. Alles andere wurde im Abschnitt „Umsetzung“ bereits ausführlich beschrieben. Dennoch kann es sich bei den Ausführungen in diesem Abschnitt um Wiederholungen des bereits Erwähnten handeln.

Als erstes gibt es die Fallunterscheidung für die Kombinationen von Strecken:

public class Line

{

public Line(double x1, double y1, double x2, double y2) {…}

public double A { get; set; }

public double B { get; set; }

public double C { get; set; }

public double[] X = new double[2]; //min, max

public double[] Y = new double[2];

public double[] SingleSharedPoint(Line line2)

{

Line line1 = this;

if (line1.B == 0)   
 { return SingleShardPointWithVerticalLine(line2, line1); }

else if (line2.B == 0)

{ return SingleShardPointWithVerticalLine(line1, line2); }

else if (AreParallel(line1, line2)) { return new double[1]; }

else

{

double x = (line1.C - (line1.B / line2.B) \* line2.C)

/ (line1.A - (line1.B / line2.B) \* line2.A);

double y = line1.ValueAt(x);

if (PointIsInAreaOfBothLines(line1, line2, new double[2] { x, y }))

{ return new double[2] { x, y }; }

else { return new double[1]; }

}

}

private double[] SingleShardPointWithVerticalLine  
 (Line normalLine, Line verticalLine)

{

if (normalLine.B == 0)

{

return new double[1];

}

else

{

double x = verticalLine.X[0];

double y = normalLine.ValueAt(verticalLine.X[0]);

if (PointIsInAreaOfBothLines(verticalLine, normalLine,   
 new double[2] { x, y }))

{ return new double[2] { x, y }; }

else { return new double[1]; }

}

}

public double ValueAt(double x)

{

if (x >= this.X[0] && x <= this.X[1] && this.B != 0)

{ return -(this.A / this.B) \* x + (this.C / this.B); }

else { return this.X[1] + 1 }

}

}

private static List<double[][]> GetTriangles (List<Line> lines)

{

List<double[][]> triangles = new List<double[][]>();

for (int i = 0; i < lines.Count() - 2; i++)

{

for (int j = i + 1; j < lines.Count() - 1; j++)

{

for (int k = j + 1; k < lines.Count(); k++)

{

Line[] currentLines = new Line[3]

{ lines[i], lines[j], lines[k] };

double[][] currentSharedPoints = new double[3][]

{

currentLines[0].SingleSharedPoint(currentLines[1]),

currentLines[0].SingleSharedPoint(currentLines[2]),

currentLines[1].SingleSharedPoint(currentLines[2])

};

bool validTriangle = true;

foreach (double[] d in currentSharedPoints)

{

if (d.Count() != 2) { validTriangle = false; }

}

if (validTriangle && TriangleHasCorners  
 (currentSharedPoints))

{

double[][] triangle = new double[3][]

{

new double[2] { currentSharedPoints[0][0],   
 currentSharedPoints[0][1] },

new double[2] { currentSharedPoints[1][0],   
 currentSharedPoints[1][1] },

new double[2] { currentSharedPoints[2][0],   
 currentSharedPoints[2][1] }

};

triangles.Add(triangle);

}

}

}

}

return triangles;

}

Die Methode TriangleHasCorners gibt genau dann wahr zurück, wenn keine zwei der drei Eckpunkte identisch sind.

Die Methode AreParallel gibt genau dann wahr zurück, wenn sowohl a, als auch b einer Line gleich sind (schließt identische Strecken nicht aus).

Die Methode PointIsInAreaOfBothLines ist selbsterklärend.

Alle weiteren Methoden dienen der Interaktion mit dem Nutzer, nicht der eigentlichen Problemlösung.