3. Aufgabe

„Dreiecke zählen“- Dokumentation

36. Bundeswettbewerb Informatik 2017/18 - 1.Runde

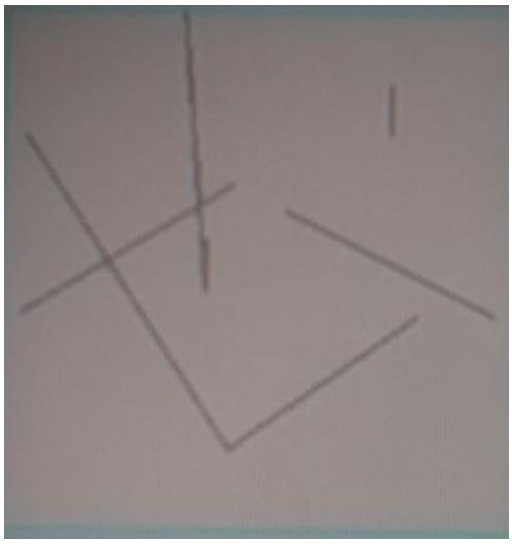
Sebastian Baron, Simon Fiebich, Lukas Rost und Max Woithe

Team-ID: 00020

27. November 2017

Lösungsidee:

Offensichtlicherweise besteht ein Dreieck aus drei Seiten. Demnach kann man alle Dreiecke finden, wenn man alle Kombinationen von drei Seiten bzw. Strecken betrachtet. Bei *n* Strecken gibt es also $\binom{n}{3}$ theoretisch mögliche Dreiecke.



Die Lösungsidee ist es also, für jede Kombination dreier Strecken die Schnittpunkte (falls diese vorhanden, die Strecken also nicht parallel sind) der aus diesen hypothetisch entstehenden (Funktions-) Geraden auszurechnen und zu überprüfen, ob sie tatsächlich auf den Strecken liegen. Weiterhin muss überprüft werden, ob alle Schnittpunkte voneinander verschieden sind. Sonst kann sich logischerweise kein Dreieck ergeben.

So sind zum Beispiel auf diesem Bild, das die Strecken aus der Beispieldatei ‚*dreiecke2.txt*' darstellt, keine Dreiecke zu erkennen. Jedoch muss man beim Berechnen der Schnittpunkte mit Funktionsgeraden arbeiten. Bei diesen jedoch ist nicht sicher, ob ein Schnittpunkt tatsächlich auf der Strecke liegt oder nur auf der dazugehörigen Funktionsgeraden. Zum Beispiel würden die ersten drei Geraden von links auf dem Bild ein Dreieck bilden, wenn die rechte nicht enden würde, bevor sie die untere schneidet.

Umsetzung:

Die Lösungsidee wird konsolenbasiert in Python 3 implementiert. Zuerst kann der Name der Eingabedatei in die Konsole eingegeben werden. Daraufhin wird die ausgewählte Textdatei in einzelne Zeilen (mit je einer Strecke) aufgespalten und diese werden in einer Liste gesammelt.

Dann wird mithilfe von drei geschachtelten *for*-Schleifen die Betrachtung der drei Seiten eines möglichen Dreiecks realisiert. Dies geschieht für alle $\binom{n}{3}$ mögliche Kombinationen von drei Strecken, die ein Dreieck bilden könnten. Da die *for*-Schleifen jeweils einen um 1 versetzten Index haben, kann es nicht zu mehrfachen Betrachtungen von Streckenkombinationen kommen. In der innersten *for*-Schleife wird mit der Funktion SCHNITTPUNKT() überprüft, ob sich je zwei Strecken des möglichen Dreiecks schneiden. Dies wird mit allen drei Schnittpunkten durchgeführt.

Wenn alle drei Schnittpunkte existieren, wird überprüft, ob zwei oder gar alle drei Schnittpunkte identisch sind. Dies kann auftreten, wenn sich Strecken überlappen oder sich mehr als zwei Strecken in einem Punkt schneiden. In diesem Fall existiert trotz der Existenz aller drei Schnittpunkte kein Dreieck, da sie nicht alle voneinander verschieden sind. Wenn die drei Strecken auch dieser Prüfung standgehalten haben, bilden sie tatsächlich ein Dreieck. Falls dies der Fall ist, werden die Schnittpunkte dem String ‚*loesung*' zusammen mit einem Zeilenumbruch hinzugefügt.

Als letzter Schritt vor der Ausgabe wird dem String eine Zeile am Beginn hinzugefügt, die angibt, wie viele Dreiecke gefunden wurden. Dies wird anhand der Zahl der Zeilenumbrüche berechnet.

Zuletzt wird der String in die Konsole ausgegeben sowie als *Name der Datei,* gefolgt von „-loes.txt" (z.B. *„dreiecke3-loes.txt")* in das Verzeichnis, in dem auch das Programm liegt, gespeichert.

Funktionen:

1. *VAR(STRECKE)* : erstellt aus den einzelnen Zeilen, also Strecken, der Eingabedatei (String-Format) einzelne Koordinaten (Float), die in einer Liste der Form [x1, y1, x2, y2] ausgegeben werden. Die Funktion sucht nach der Textstelle *'.'* . Anhand dieser werden die Zeichen seit dem Ende der letzten Koordinate als Vorkommaanteil und die 6 folgenden Zeichen als Nachkommaanteil der Float-Zahl verwendet. Das Programm geht vom Vorhandensein dieses Punktes sowie von sechs Nachkommastellen aus.

2. *FUNKTION(STRECKE):* berechnet die Funktionsparameter (m; n) der Funktion in der Form $y = m * x + n$, die die angegebene Strecke (bzw. die aus dieser durch Verlängerung über die Endpunkte hinaus entstehende Gerade) beschreibt. Die Funktion benutzt *VAR()*, um numerische Werte der einzelnen Koordinaten zu erhalten. Es gelten die Gleichungen $m = \frac{y_2 - y_1}{x_2 - x_1}$ sowie $n = y - m * x$.

3. *LIMIT(STRECKE1, STRECKE2, SPX, SPY)*: *spx* und *spy* sind die x- und y-Koordinaten des Schnittpunkts von *strecke1* und *strecke2*. *LIMIT()* kontrolliert, ob der Schnittpunkt auf den beiden Strecken liegt. Dazu wird überprüft, ob der Punkt *(spx|spy)* zwischen den x- und y-Koordinaten der Endpunkte der beiden Strecken liegt. Die Funktion benutzt ebenfalls *VAR()*, um numerische Werte der einzelnen Koordinaten zu erhalten.

4. *SCHNITTPUNKT(STRECKEN, A, B)*: *strecken* steht hier für die Liste mit den einzelnen Punkten der Strecken. *a* und *b* sind Indexe dieser Liste und stehen somit für jeweils eine Strecke. Die Funktion berechnet die Koordinaten des Schnittpunkts der Funktionsgeraden der einen Strecke mit der Funktionsgeraden der anderen Strecke. Hier wird die Funktion *FUNKTION()* benutzt, um die Gleichungen der Geraden zu erhalten. Anschließend wird mit *LIMIT()* überprüft, ob der Schnittpunkt tatsächlich existiert. Zuletzt wird, falls vorhanden, der Schnittpunkt in der Form *[vorhanden?, x-Koordinate, y-Koordinate]* zurückgegeben. Der x-Wert eines Schnittpunktes lässt sich mit folgender Gleichung bestimmen: $x_S = \frac{n_1 - n_2}{m_2 - m_1}$. Anschließend kann daraus mithilfe der gegebenen Funktionsgleichung einer der Funktionen der y-Wert bestimmen. Da diese Zusammenhänge nur für nicht parallele Funktionen gelten, muss $m_1 \neq m_2$ sein. Falls eine der Strecken parallel zur y-Achse ist, hat die korrespondierende Gerade eine Gleichung der Form $x = a$ (mit a als positive reelle Zahl). Da alle Punkte dieser Strecke (bzw. Gerade) also den x-Wert a haben, muss auch der Schnittpunkt diesen x-Wert haben. Diesen kann man nun in die Gleichung der anderen Funktion einsetzen und erhalt den y-Wert des Schnittpunkts. Auch hier dürfen nicht beide Strecken parallel zur y-Achse sein.

Beispiele:

Beispiel 1:

```
Es gibt 9 Dreiecke:
(0.0 | 0.0) (0.0 | 200.0) (120.0 | 0.0)
(0.0 | 200.0) (0.0 | 20.0) (55.1 | 108.17)
(120.0 | 0.0) (20.0 | 0.0) (41.74 | 130.43)
(120.0 | 0.0) (100.0 | 0.0) (82.76 | 62.07)
```

```
(20.0 | 0.0) (100.0 | 0.0) (50.0 | 180.0)
(55.1 | 108.17) (41.74 | 130.43) (31.82 | 70.91)
(55.1 | 108.17) (82.76 | 62.07) (65.38 | 124.61)
(41.74 | 130.43) (82.76 | 62.07) (50.0 | 180.0)
(31.82 | 70.91) (65.38 | 124.61) (50.0 | 180.0)
```

Beispiel 2: Es gibt kein Dreieck

Beispiel 3:

```
Es gibt 3 Dreiecke:
(104.58 | 80.48) (140.0 | 40.0) (118.35 | 91.96)
(75.64 | 56.37) (89.57 | 67.97) (77.34 | 73.4)
(132.76 | 103.97) (118.35 | 91.96) (109.75 | 112.59)
```

Beispiel 4:

```
Es gibt 5 Dreiecke:
(60.0 | 110.0) (52.34 | 129.15) (38.82 | 141.77)
(60.0 | 110.0) (70.0 | 85.0) (80.0 | 80.0)
(117.59 | 68.25) (130.0 | 0.0) (126.47 | 59.96)
(124.0 | 102.0) (110.0 | 130.0) (122.46 | 128.18)
(161.69 | 122.61) (163.96 | 132.82) (180.0 | 120.0)
```

Beispiel 5:

```
Es gibt ein Dreieck:
(61.74 | 103.48) (73.33 | 93.34) (80.0 | 140.0)
```

Beispiel 6:

```
Es gibt 20 Dreiecke:
(91.76 | 70.59) (128.28 | 82.76) (111.82 | 54.54)
(128.28 | 82.76) (104.12 | 74.71) (118.93 | 66.74)
(128.28 | 82.76) (49.57 | 56.52) (80.0 | 0.0)
(128.28 | 82.76) (150.0 | 90.0) (141.92 | 106.15)
(128.28 | 82.76) (75.0 | 65.0) (102.11 | 37.9)
(104.12 | 74.71) (75.0 | 65.0) (20.0 | 120.0)
(49.57 | 56.52) (75.0 | 65.0) (10.0 | 130.0)
(49.57 | 56.52) (60.0 | 60.0) (60.0 | 37.14)
(111.82 | 54.54) (124.44 | 44.45) (80.0 | 0.0)
(80.0 | 0.0) (118.93 | 66.74) (137.0 | 57.0)
(80.0 | 0.0) (141.92 | 106.15) (156.67 | 76.67)
(80.0 | 0.0) (102.11 | 37.9) (110.0 | 30.0)
(118.93 | 66.74) (102.11 | 37.9) (20.0 | 120.0)
(80.0 | 0.0) (102.11 | 37.9) (10.0 | 130.0)
(137.0 | 57.0) (110.0 | 30.0) (20.0 | 120.0)
(80.0 | 0.0) (110.0 | 30.0) (10.0 | 130.0)
(71.0 | 92.54) (20.0 | 120.0) (55.65 | 84.35)
(39.28 | 75.62) (10.0 | 130.0) (55.65 | 84.35)
(160.0 | 170.0) (140.0 | 180.0) (50.0 | 170.0)
(50.0 | 170.0) (120.0 | 170.0) (98.46 | 175.38)
```

Die Laufzeiten lagen jeweils im Bereich weniger Sekunden.

Quellcode:

```
##############################################Funktionen####################
##############################################
```

```
#berechnet den Schnittpunkt der a-ten und b-ten Gerade der Liste Strecken;
gibt Liste des Formats [Schnittpunkt existiert, x-Koordinate, y-Koordinate]
aus
def Schnittpunkt(strecken, a, b):
    x = 0
#   Überprüfung ob eine der beiden Geraden parallel zur y-Achse ist (es
gibt keine Funktionsgleichungen in diesem Fall, deswegen muss anders
gerechnet werden)
    if Var(strecken[a])[0] == Var(strecken[a])[2]:
#       Überprüfung ob beide Strecken parallel zur y-Achse sind (parallele
Geraden können keinen einzelnen Schnittpunkt haben)
        if Var(strecken[b])[0] == Var(strecken[b])[2]:
            return [False]
#       Berechnung des Schnittpunkts und Überprüfung auf dessen Existenz
        SPabx = round(Var(strecken[a])[0],2)
        SPaby = round((Funktion(strecken[b])[0] * SPabx) +
Funktion(strecken[b])[1],2)
        if limit(strecken[a], strecken[b], SPabx, SPaby):
            return [True, str(SPabx), str(SPaby)]
        else:
            return [False]
    elif Var(strecken[b])[0] == Var(strecken[b])[2]:
        SPabx = round(Var(strecken[b])[0],2)
        SPaby = round((Funktion(strecken[a])[0] * SPabx) +
Funktion(strecken[a])[1],2)
        if limit(strecken[a], strecken[b], SPabx, SPaby):
            return [True, str(SPabx), str(SPaby)]
        else:
            return [False]
    else:       # Überprüfung ob beide Strecken parallel zueinander sind
(parallele Geraden können keinen einzelnen Schnittpunkt haben)
        if Funktion(strecken[b])[0] == Funktion(strecken[a])[0]:
            return [False]
#       normale Berechnung des Schnittpunkts sowie Überprüfung auf dessen
Existenz
        else:
            SPabx = round((Funktion(strecken[a])[1] -
Funktion(strecken[b])[1]) / (Funktion(strecken[b])[0] -
Funktion(strecken[a])[0]),2)
            SPaby = round((Funktion(strecken[a])[0] * SPabx) +
Funktion(strecken[a])[1],2)
            if limit(strecken[a], strecken[b], SPabx, SPaby):
                return [True, str(SPabx), str(SPaby)]
            else:
                return [False]

#aus den Punkten einer Strecke Funktionsgleichung berechnen; Ausgabe in
Form [m, n] (y=mx+n)
def Funktion(strecke):
    ret = [(Var(strecke)[3] - Var(strecke)[1])/(Var(strecke)[2] -
Var(strecke)[0])]
    ret = ret + [Var(strecke)[1] - (Var(strecke)[0] * ret[0])]
    return ret

#aus den in einem String vorliegenden Punkten eine Liste mit 4 Koordinaten
(Float) erstellen; Ausgabe in der Form [x1, y1, x2, y2]
def Var(strecke):
    var = []
    x = 0
#   sucht nach dem Punkt in den Koordinaten und wandelt die Zahl mit 6
Nachkommastellen in einen Float um
    for a in range(0, len(strecke)):
        if strecke[a] == '.':
```

```
            var = var + [float(strecke[x:(a+7)])]
            x = a + 7
    return var

#Test ob der Schnittpunkt zweier Geraden auf den zugehörigen Strecken
liegt; Ausgabe erfolgt in Form eines Booleschen Wertes
def limit(strecke1, strecke2, spx, spy):
    x = 0
    if Var(strecke1)[0] <= spx <= Var(strecke1)[2] or Var(strecke1)[0] >=
spx >= Var(strecke1)[2]:
        x = x + 1
    if Var(strecke1)[1] <= spy <= Var(strecke1)[3] or Var(strecke1)[1] >=
spy >= Var(strecke1)[3]:
        x = x + 1
    if Var(strecke2)[0] <= spx <= Var(strecke2)[2] or Var(strecke2)[0] >=
spx >= Var(strecke2)[2]:
        x = x + 1
    if Var(strecke2)[1] <= spy <= Var(strecke2)[3] or Var(strecke2)[1] >=
spy >= Var(strecke2)[3]:
        x = x + 1
    if x == 4:
        return True
    else:
        return False


##################################################Programm##################
#############################################

datei = input('Geben sie den Namen der zu bearbeitenden Datei ein: ')

streckentxt = open(datei)
streckentext = streckentxt.read()
streckentxt.close()

streckenp = []

#String in einzelne Zeilen (Strecken) zerlegen und sammeln dieser in der
Liste streckenp
x = 0
for a in range(0,len(streckentext)):
    if streckentext[a:(a+1)] == '\n':
        streckenp = streckenp + [streckentext[x:a]]
        x = a + 1

streckenp = streckenp[1:]


loesung = ''

#die jeweiligen Schnittpunkte dreier Geraden berechnen und Überprüfung auf
deren Existenz auf den Strecken (wenn es sie gibt, gibt es ein Dreieck)
for a in range(0,(len(streckenp)-2)):       #durch versetzte Parameter kann
es nicht zu mehrfachen Betrachtungen kommen
    for b in range(a+1,(len(streckenp)-1)):
        for c in range(b+1,len(streckenp)):
            x = 0
            if Schnittpunkt(streckenp, a, b)[0]:
                x = x + 1
            if Schnittpunkt(streckenp, a, c)[0]:
                x = x + 1
            if Schnittpunkt(streckenp, b, c)[0]:
                x = x + 1
```

```
            if x == 3:      #Überprüfung ob es identische Punkte gibt
                if Schnittpunkt(streckenp, a, b) == Schnittpunkt(streckenp,
a, c) or Schnittpunkt(streckenp, a, b) == Schnittpunkt(streckenp, b, c) or
Schnittpunkt(streckenp, a, c) == Schnittpunkt(streckenp, b, c):
#Verhindern von Dreiecken aus nur einem Punkt, sowie doppelt auftretenden
Dreiecken aufgrund von überlappenden Strecken
                    x= x - 3
                else:
                    loesung = loesung + "(" + Schnittpunkt(streckenp, a,
b)[1] + ' | ' + Schnittpunkt(streckenp, a, b)[2] + ') (' +
Schnittpunkt(streckenp, a, c)[1] + ' | ' + Schnittpunkt(streckenp, a, c)[2]
+ ') (' + Schnittpunkt(streckenp, b, c)[1] + ' | ' +
Schnittpunkt(streckenp, b, c)[2] + ') \n'

#Zählen der Dreiecke
zaehler = 0
for a in range(0, len(loesung)):
    if loesung[a:a+1] == '\n':
        zaehler = zaehler + 1

#Benutzerfreundliche Angabe der Anzahl der Dreiecke
if zaehler == 0:
    loesung = 'Es gibt kein Dreieck'
elif zaehler == 1:
    loesung = 'Es gibt ein Dreieck:\n' + loesung[:-1]
else:
    loesung = ('Es gibt %s Dreiecke:\n' %zaehler) + loesung[:-1]

#Ausgeben der Lösung(en) in der Konsole, sowie einer separaten Textdatei
print (loesung)

outdatei = datei.split(".")[0] + "-loes.txt"
loesungtxt = open(outdatei, mode='w')
loesungtxt.write(loesung)
loesungtxt.close()
```