# Aufgabe 2: Dreiecksbeziehungen

Teilnahme-Id: 01189

Bearbeiter/-in dieser Aufgabe:
Georgijs Vilums

29. April 2019

## Inhaltsverzeichnis

**1 Lösungsidee** 1

**2 Umsetzung** 3

**3 Beispiele** 5

3.1 dreiecke1.txt . . . 5
3.2 dreiecke2.txt . . . 5
3.3 dreiecke3.txt . . . 6
3.4 dreiecke4.txt . . . 7
3.5 dreiecke5.txt . . . 8
3.6 Diskussion . . . 9

**4 Quellcode** 9

## 1 Lösungsidee

Die Perfekte Anordnung von Dreiecken entlang einer Achse zu finden ist ein extrem schweres, möglicherweise sogar ein NP-Schweres Problem. Daher ist es eher sinnvoll, sich mit möglichen Annäherungen an eine ideale Konstellation auseinanderzusetzen.
Der hier gewählte Ansatz versucht, das Problem durch eine Näherung zu lösen. Die grundlegende Idee ist dabei, die gegebenen Dreiecke nacheinander entlang einer Achse anzureihen, wobei versucht wird, bei der individuellen Platzierung der Dreiecke möglichst platzsparend zu arbeiten.

Bevor man die Aneinanderreihung mehrerer Dreiecke betrachtet, sollte man in Betracht ziehen, auf welche Weise man ein einzelnes Dreieck möglichst gut in eine Gegebene Lücke einfügen kann. Natürlich kann man ein Dreieck, welches mit einer Ecke die x-Achse berührt, auf unendlich viele Winkel rotieren. Es gibt aber zwei Positionen, die besonders sinnvoll sind, um Platz zu sparen.

In Abbildung 1 erkennt man, dass das rote Dreieck so weit wie möglich nach Links gelehnt wurde. Außerdem wurde sein kleinster Winkel an der x-Achse platziert, weil der Platz dort am wichtigsten ist. Für den Großteil der Dreiecke ist diese *linksseitige* Platzierung am sinnvollsten, da nur sehr wenig Platz beansprucht wird.

In Abbildung 2 zeigt sich ein Fall, in dem eine linksseitige Platzierung nicht ganz Vorteilhaft wäre. Würde man das rote Dreieck möglichst weit nach Links drehen, wäre der restliche Winkel an $x \approx 100$ nicht groß genug, um ein weiteres Dreieck zu plazieren. Gleichzeitig würde die Platzierung weiterer Dreiecke an $x \approx 400$ erschwert werden. Das rot markierte Dreieck sollte also stattdessen *rechtsseitig* plaziert werden, damit die Plazierung nachfolgender Dreiecke erleichtert wird. Rechtsseitig bedeutet in diesem Zusammenhang, dass das Dreieck so weit es geht nach rechts gedreht und somit an die x-Achse angeschmiegt wird.

Abbildung 1: Linksseitige Platzierung

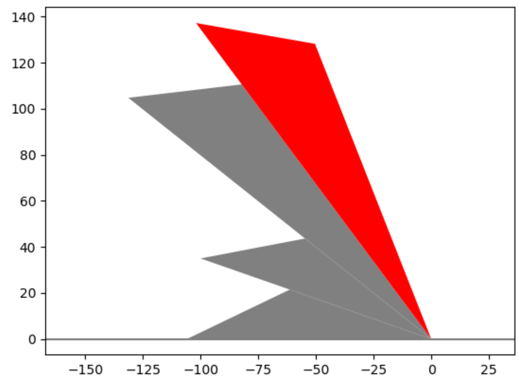
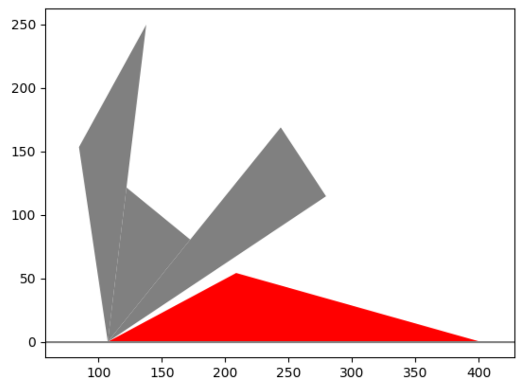



Abbildung 2: Rechtsseitige Platzierung




Sowohl bei der Links- als auch bei der rechtsseitigen Platzierung könnten Fälle auftreten, wo es zu einer Kollision von Dreiecken kommt, wenn der Winkel einfach so weit es geht nach links bzw. rechts gesetzt werden würde. Wie in Abbildung 3 dargestellt ist, sollte die Platzierung auch vorher platzierte Dreiecke beachten, um Kollisionen zu verhindern. Im Fall, dass man eine linksseitige Kollision mit einer Kante vermeiden möchte, lässt sich ein korrigierter Winkel $\alpha$ mit der Formel

$$\alpha = \pi - \beta - \arcsin \frac{\Delta x \cdot \sin \beta}{b} \tag{1}$$

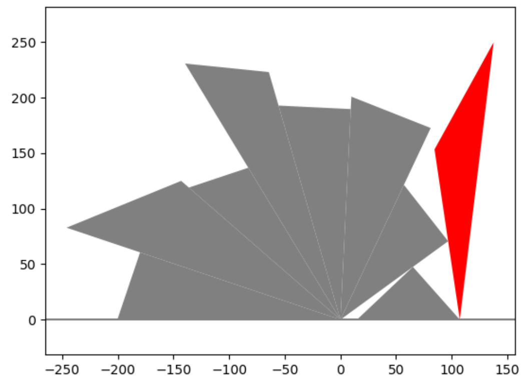
berechnen. Dabei ist $\beta$ der Winkel der Kante des vorherigen Dreiecks zur x-Achse, $\Delta x$ die Distanz der Dreiecke an der x-Achse und $b$ die Länge der linksseitigen Kante des zu korrigierenden Dreiecks. Bei einer rechtsseitigen Platzierung kann der Fall auftreten, dass der gegebene Winkel zu klein für das zu platzierende Dreieck ist. In diesem Fall kann die angepasste x-Koordinate des Dreiecks mit der Formel

$$x_2 = x_1 - \frac{y_1}{\tan(\gamma_2)} - \frac{y_1 - y_2}{\tan(\pi - \alpha_1 - \gamma_1)} + \frac{y_1 - y_2}{\tan(\gamma_2)} \tag{2}$$

berechnet werden, wobei $x_1$ und $y_1$ der obere rechte Eckpunkt, $\alpha_1$ der Neigungswinkel an der x-Achse und $\gamma_1$ der Basiswinkel an der x-Achse, des vorherigen Dreiecks sind. Entsprechendes gilt beim korrigierten Dreieck für die Parameter mit Subskript 2. Wenn $y_2 > y_1$ ist, fallen die letzten beiden Brüche bei der Berechnung weg.

Mithilfe der genannten Platzierungsverfahren lässt sich eine Anordnung nun aufbauen, indem für jedes Dreieck in Folge entschieden wird, ob es rechts- oder linksseitig plaziert werden sollte. Man betrachtet während der Platzierung immer den aktuellen Winkel, der noch offen ist. Ist der Winkel klein und erfüllt

Abbildung 3: Verhindern von Kollisionen




ein Dreieck diesen gut, wird dieses rechtsseitig plaziert. Ist dies nicht der Fall, wird es linksseitig plaziert, außer es ist nicht mehr genug Platz vorhanden, in welchem Fall wieder auf die Rechtsseitige Platzierung mit den oben genannten Korrekturverfahren zurückgegriffen wird.
Auf diese Art und Weise können beliebig viele Dreiecke relativ kompakt entlang einer Achse platziert werden.

## 2 Umsetzung

Das Programm wurde in Python implementiert. Es wurden zwei Python-Libraries verwendet: numpy zur Erleichterung mathematischer Berechnungen und matplotlib zur Visualisierung.

Ein erstes grundlegendes Problem ist die Frage der Speicherung der Dreiecke. Diese werden zwar in einer Datei als Listen von Eckpunkten gegeben, aber konkrete Eckpunkte sind für die Berechnungen uninteressant, da diese erst bei einer Platzierung festgelegt werden. Wichtig sind nur die Seitenlängen, und die den Seiten gegenüberliegenden Winkel.
Wenn ein Dreieck plaziert wird, müssen außerdem einige weitere Informationen festgelegt werden:

- Der Neigungswinkel $\alpha$, der den Winkel zwischen der x-Achse und der linken Kante des Dreiecks angibt
- Der Basiswinkel $\gamma$, der angibt, welcher Winkel an der x-Achse anliegt
- Die $x$-Koordinate, an der $\gamma$ anliegt
- Die Länge $b$ der vom Basiswinkel aus linken Kante
- Die Länge der vom Basiswinkel aus rechten Kante

Diese Werte werden als Felder von Objekten der Klasse `Triangle` verwaltet und sind nötig, um die Position des Dreiecks in der Ebene festzulegen.
Außerdem implementiert die Klasse `Triangle` die Methoden, mit denen ein Dreieck plaziert werden kann, nämlich `place_on_fitting_side` für linksseitige und `place_on_fitting_angle` für rechtsseitige Platzierung. Diese Methoden sollen im Folgenden erläutert werden, vorher sollen aber noch einige Grundlagen von beiden Methoden thematisiert werden, die auch für den restlichen Programmablauf relevant sind.

Während der Ausführung des Programms gibt es zwei Variablen, die den aktuellen Status mitverfolgen. Dies sind `current_x` und `current_angle`. Diese Variablen sind Parameter für beide gerade genannten Methoden und dienen als eine Art Koordinatenpaar entlang der x-Achse. Sie beschreiben eine der Kanten des zuletzt platzierten Dreiecks, und zwar die, an die im nächsten Schritt ein Dreieck Linksseitig platziert werden könnte. Ein Beispiel: Es ist bisher kein Dreieck platziert, sowohl `current_x` als auch `current_angle` sind 0. Nun wird ein Dreieck linksseitig platziert, mit $\gamma = 10°$. `current_x` bleibt 0, während `current_angle` den Wert 10° annimmt. Ein weiteres Beispiel: Ein Dreieck mit der längeren Kante

10 und einem Winkel von 20° in der oberen rechten Ecke wird rechtsseitig platziert. `current_x` beträgt nun 10, während `current_angle` den Wert 20° annimmt. Im Folgenden wird `current_angle` als belegter Winkel bezeichnet.

Die Methode `place_on_fitting_side` akzeptiert als Parameter die aktuelle $x$-Koordinate, den aktuell durch andere Dreiecke belegten Winkel und eine Liste aller bisher platzierten Dreiecke. Die Methode beginnt, indem der kleinste Winkel zum Basiswinkel gemacht wird. Dies wird, wie bereits gesagt, getan, da der Platz an der x-Achse am wertvollsten ist und ein kleiner Winkel wenig Platz verschwendet. Außerdem werden die $x$-Koordinate und die beiden Seiten des Dreiecks festgelegt. Der Neigungswinkel $\alpha$ des Dreiecks wird vorerst gleich dem bereits belegten Winkel gesetzt. Im Idealfall, wie bspw. in Abbildung 1, ist dies bereits die ideale Platzierung.
Es kann aber auch sein, dass, wie es in Abbildung 3 der Fall sein würde, ein Neigungswinkel gleich dem belegten Winkel zu einer Kollision führen würde. Die Methode prüft daher nach der vorläufigen Festlegung von $\alpha$ für jedes bereits platzierte Dreieck, ob es zu Kollisionen zwischen seinen Kanten und den Kanten des neuen Dreiecks kommt. Wenn es zu einer Kollision zwischen Zwei Kanten kommt, wird diese mithilfe von Formel (1) beseitigt. Dieses Verfahren wird für verschiedene Kanten durchgeführt. Ein Mögliches Resultat ist die Position des roten Dreiecks in Abbildung 3.
Abschließend gibt die Methode den neuen belegten Winkel und die $x$-Koordinate wieder, sodass diese als Parameter für die nächste Platzierung dienen Können.

Ähnlich arbeitet die Methode `place_on_fitting_angle`. Die wesentlichen Unterschiede liegen im Festlegen des Basiswinkels $\gamma$ und dem Verfahren zur Beseitigung von Kollisionen.
Die Methode beginnt, indem eine Liste erzeugt wird, wobei jedes Listenelement die Differenz vom jeweiligen der Dreieckswinkel zum restlichen Winkel repräsentiert. Ist bspw. ein Winkel von 90° gesucht, bei einem Dreieck mit den Winkeln 30°, 60° und 90°, dann wird die Liste die Werte 60°, 30° und 0° enthalten. Der Winkel, bei dem die geringste nichtnegative Differenz vorliegt ist der, der am besten in die aktuelle Lücke passt. Dieser Winkel wird zum Basiswinkel $\gamma$ gemacht. Die Seitenlängen und der Neigungswinkel $\alpha$ werden so festgelegt, dass die längere Seite an der x-Achse anliegt.
Zur Verhinderung von Kollisionen wird ähnlich wie bei der Methode `place_on_fitting_side` vorgegangen, außer dass in diesem Fall Formel (2) zur Lösung von Kollisionen verwendet wird.

Mithilfe dieser Beiden Methoden besteht nun die Möglichkeit, ein Dreieck `triangle` einer Anordnung hinzuzufügen, einfach, indem man `triangle.place_on_fitting_side(*args)` oder `triangle.place_on_fitting_angle(*args)` aufruft. Es ist die Aufgabe der Funktion `arrange` herauszufinden, in welcher Reihenfolge die Dreiecke angeordnet werden sollen und auf welche Art sie plaziert werden. Die Funktion akzeptiert als Parameter eine Liste von Dreiecken und die Parameter `delta_a1`, `delta_a2` und `delta_b`.
Die Funktion ist so aufgebaut, dass sie eine Schleife abarbeitet, solange noch unplatzierte Dreiecke vorhanden sind. In jedem Schleifendurchlauf wird ein Dreieck platziert. Die innere Logik der Schleife ist wie folgt aufgebaut:

1. Es wird eine Liste aller Winkel der verbleibenden Dreiecke erstellt, zusammen mit der Referenz ihres zugehörigen Dreiecks.
2. Für jeden Winkel in dieser Liste wird geprüft, ob seine Differenz mit dem restlichen Winkel den Parameter `delta_a1` und der Winkel selbst den Parameter `delta_a2` nicht überschreitet. Wenn dies dies der Fall ist, wird die weitere Überprüfung abgebrochen und das dem Winkel zugehörige Dreieck rechtsseitig Platziert.
3. Wird kein solches passendes Dreieck gefunden, wird stattdessen eine Liste der kleinsten Winkel jedes Dreiecks erstellt. Für jeden dieser Winkel wird geprüft, ob die Summe des letzteren und dem Parameter `delta_b` kleiner als der gesuchte Winkel ist. Trifft dies zu, wird, wie auch im anderen Fall, die Suche abgebrochen und das dem Winkel zugehörige Dreieck linksseitig platziert.
4. Wird auch hier kein passendes Dreieck gefunden, so wird stattdessen das Dreieck, das den Winkel mit der kleinsten absoluten Differenz zum gesuchten Winkel besitzt, rechtsseitig plaziert, unter Umständen mit einer Verschiebung, um Kollisionen zu vermeiden.

Für eine gegebene Liste von Dreiecken gibt diese Funktion also eine weitere Liste von Dreiecken zurück, wobei alle Dreiecke in dieser Liste an einem konkreten Punkt nach den o.g. Regeln platziert wurden.

Das letzte Glied der Kette ist die Funktion `optimize`, welche versucht, für eine gegebene Menge von Dreiecke möglichst gute Parameter `delta_a1`, `delta_a2` und `delta_b` zu finden. Die Methode wird mit erprobten Werten für diese Parameter initialisiert. Dann wird eine bestimmte Anzahl von Durchläufen durchlaufen, wobei in jedem Durchlauf die genannten Parameter ein wenig justiert werden und geguckt wird, welchen Einfluss diese Justierung auf das Ergebnis der Methode `arrange` hat. Wird das Ergebnis besser, wird die Veränderung beibehalten, verschlechtert es sich, werden die Parameter in die andere Richtung verändert. Auf diese Art und Weise werden gezielt verschiedene Optionen für die Parameter erprobt, sodass eine Anpassung an die Daten stattfinden kann.

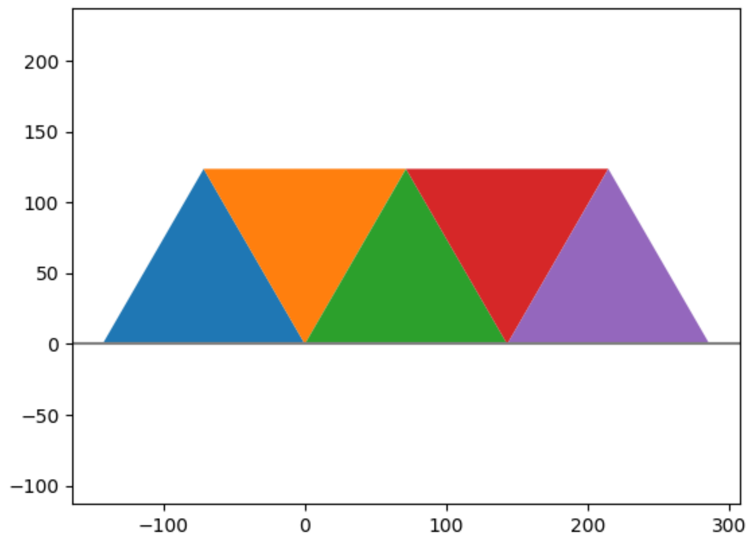
Nachdem im Programm die Parameter für die Funktion `arrange` mithilfe der Funktion `optimize` optimiert wurden, wird das Ergebnis von `arrange` an die Konsole bzw. graphisch ausgegeben.

# 3 Beispiele

## 3.1 dreiecke1.txt

Eingabe: `python 2_2.py dreiecke1.txt`
Ausgabe:

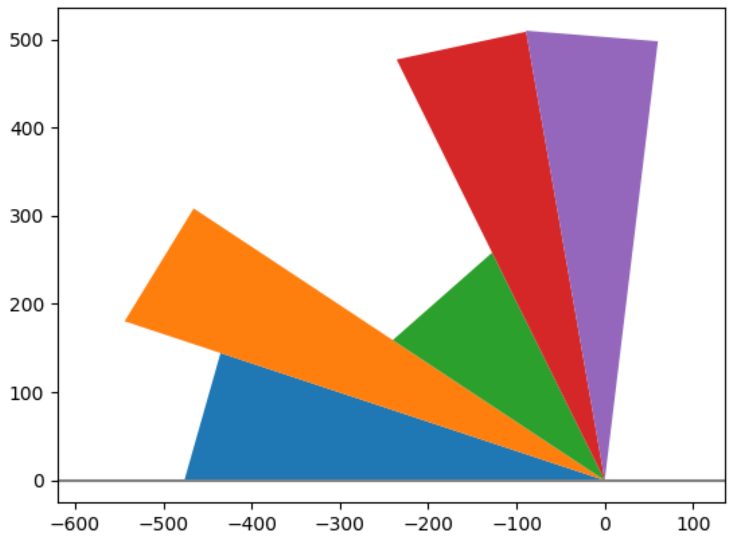
```
D1 :    P(0|0)              Q(-142.874|0.0)     R(-71.476|123.71)
D2 :    P(0|0)              Q(-71.476|123.71)   R(71.36|123.777)
D3 :    P(0|0)              Q(71.399|123.71)    R(142.874|0.0)
D4 :    P(142.874|0.0)      Q(71.399|123.71)    R(214.234|123.777)
D5 :    P(142.874|0.0)      Q(214.273|123.71)   R(285.748|0.0)
Gesamtdistanz: 142.874
```




## 3.2 dreiecke2.txt

Eingabe: `python 2_2.py dreiecke2.txt`
Ausgabe:

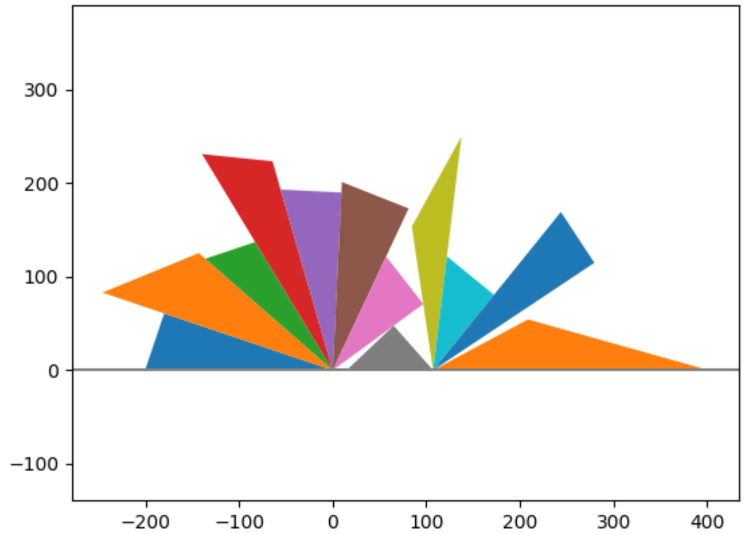
```
D1 :    P(0|0)              Q(-476.092|0.0)     R(-435.134|144.3)
D2 :    P(0|0)              Q(-543.82|180.342)  R(-465.532|308.292)
D3 :    P(0|0)              Q(-240.07|158.983)  R(-127.533|258.156)
D4 :    P(0|0)              Q(-235.686|477.083) R(-89.045|508.652)
D5 :    P(0|0)              Q(-89.253|509.838)  R(60.26|497.765)
Gesamtdistanz: 0
```




## 3.3 dreiecke3.txt

Eingabe: `python 2_2.py dreiecke3.txt`
Ausgabe:

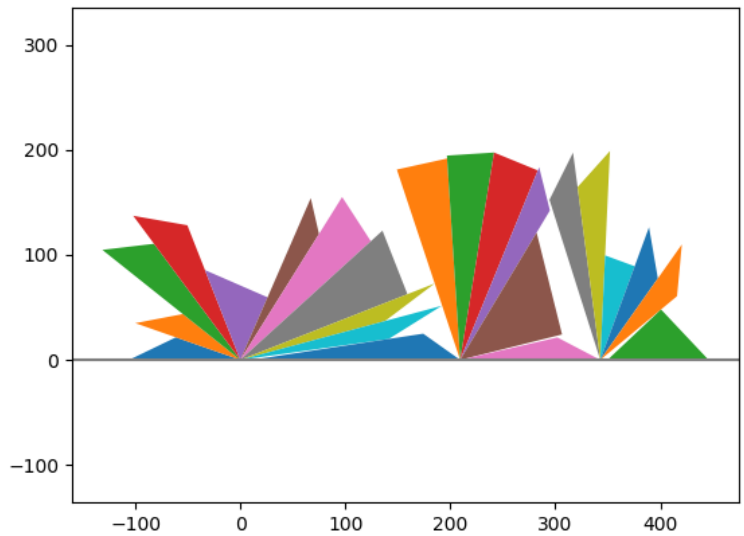
```
D1 :    P(0|0)              Q(-200.489|0.0)     R(-180.059|60.652)
D2 :    P(0|0)              Q(-246.113|82.901)  R(-143.108|125.08)
D3 :    P(0|0)              Q(-136.197|119.04)  R(-82.813|136.905)
D4 :    P(0|0)              Q(-139.63|230.834)  R(-64.332|223.14)
D5 :    P(0|0)              Q(-55.629|192.952)  R(9.293|189.773)
D6 :    P(0|0)              Q(9.842|200.98)     R(81.107|172.727)
D7 :    P(0|0)              Q(57.057|121.509)   R(96.806|70.715)
D8 :    P(14.864|0.0)       Q(65.146|47.588)    R(107.281|0.0)
D9 :    P(107.281|0.0)      Q(84.551|153.442)   R(137.648|249.908)
D10:    P(107.281|0.0)      Q(122.057|121.6)    R(172.383|80.559)
D11:    P(107.281|0.0)      Q(243.842|168.986)  R(279.598|114.704)
D12:    P(107.281|0.0)      Q(208.649|54.236)   R(401.664|0.0)
Gesamtdistanz: 107.281
```




## 3.4 dreiecke4.txt

Eingabe: `python 2_2.py dreiecke4.txt`
Ausgabe:

```
D1 :  P(0|0)            Q(-105.948|0.0)     R(-61.351|21.473)
D2 :  P(0|0)            Q(-100.0|35.0)      R(-54.722|43.653)
D3 :  P(0|0)            Q(-131.334|104.768) R(-82.181|110.595)
D4 :  P(0|0)            Q(-101.991|137.255) R(-50.431|128.225)
D5 :  P(0|0)            Q(-33.645|85.545)   R(25.95|59.595)
D6 :  P(0|0)            Q(67.073|154.033)   R(74.945|119.93)
D7 :  P(0|0)            Q(96.946|155.137)   R(124.295|113.029)
D8 :  P(0|0)            Q(135.344|123.077)  R(159.135|62.587)
D9 :  P(0|0)            Q(185.124|72.809)   R(137.835|37.021)
D11:  P(0|0)            Q(191.797|51.515)   R(141.32|19.939)
D14:  P(11.055|0.0)     Q(174.155|24.964)   R(209.337|0.0)
D10:  P(209.337|0.0)    Q(149.048|181.288)  R(196.908|191.798)
D12:  P(209.337|0.0)    Q(196.727|194.592)  R(241.632|197.502)
D13:  P(209.337|0.0)    Q(241.632|197.502)  R(283.272|180.44)
D15:  P(209.337|0.0)    Q(284.517|183.478)  R(294.793|142.314)
D16:  P(209.337|0.0)    Q(282.14|121.243)   R(306.36|24.22)
D19:  P(209.337|0.0)    Q(301.905|21.362)   R(342.754|0.0)
D17:  P(342.754|0.0)    Q(294.349|152.502)  R(316.817|197.404)
D18:  P(342.754|0.0)    Q(321.129|164.585)  R(351.923|198.889)
D20:  P(342.754|0.0)    Q(347.342|99.519)   R(375.503|89.177)
D21:  P(342.754|0.0)    Q(389.164|126.377)  R(397.561|77.597)
D22:  P(342.754|0.0)    Q(420.423|109.966)  R(415.849|60.68)
D23:  P(349.669|0.0)    Q(400.6|48.021)     R(445.877|0.0)
Gesamtdistanz: 349.669
```

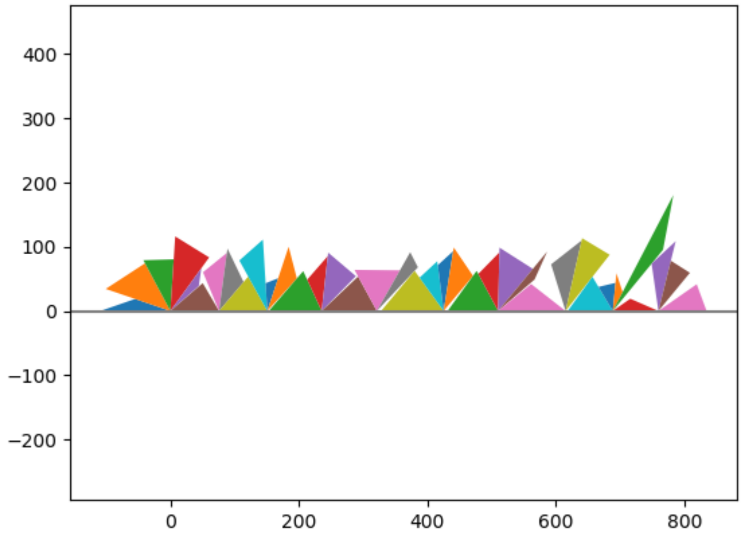



## 3.5 dreiecke5.txt

Eingabe: `python 2_2.py dreiecke5.txt`
Ausgabe:

```
D1 :   P(0|0)            Q(-109.037|0.0)    R(-54.945|18.737)
D2 :   P(0|0)            Q(-100.71|34.343)  R(-39.503|74.239)
D3 :   P(0|0)            Q(-42.06|79.044)   R(5.185|80.437)
D4 :   P(0|0)            Q(7.485|116.121)   R(60.321|83.435)
D25:   P(0|0)            Q(48.161|66.615)   R(44.302|39.081)
D35:   P(0|0)            Q(50.354|43.41)    R(75.743|0.0)
D5 :   P(75.743|0.0)     Q(50.762|60.332)   R(87.951|90.449)
D6 :   P(75.743|0.0)     Q(88.863|97.205)   R(113.952|45.056)
D9 :   P(75.888|0.0)     Q(121.177|53.609)  R(150.894|0.0)
D7 :   P(150.894|0.0)    Q(107.314|78.618)  R(144.074|111.155)
D8 :   P(150.894|0.0)    Q(148.244|43.197)  R(167.835|51.02)
D10:   P(150.894|0.0)    Q(184.103|100.016) R(196.962|50.664)
D22:   P(153.216|0.0)    Q(206.83|62.093)   R(235.544|0.0)
D11:   P(235.544|0.0)    Q(212.98|48.794)   R(245.088|85.592)
D12:   P(235.544|0.0)    Q(245.678|90.881)  R(288.873|54.231)
D13:   P(235.544|0.0)    Q(292.257|53.709)  R(321.451|0.0)
D14:   P(321.451|0.0)    Q(286.684|63.963)  R(357.391|63.311)
D16:   P(321.451|0.0)    Q(373.397|91.507)  R(385.403|67.994)
D26:   P(328.113|0.0)    Q(380.256|62.523)  R(425.422|0.0)
D15:   P(425.422|0.0)    Q(387.972|51.841)  R(415.534|77.48)
D17:   P(425.422|0.0)    Q(416.324|71.29)   R(440.297|93.16)
D18:   P(425.422|0.0)    Q(441.21|98.877)   R(471.198|54.677)
D32:   P(431.637|0.0)    Q(476.865|62.509)  R(510.281|0.0)
D19:   P(510.281|0.0)    Q(479.431|57.709)  R(512.202|91.221)
D20:   P(510.281|0.0)    Q(512.359|98.695)  R(564.38|65.592)
D37:   P(510.281|0.0)    Q(586.638|92.578)  R(567.034|48.981)
D23:   P(510.281|0.0)    Q(561.72|41.989)   R(615.358|0.0)
D21:   P(615.358|0.0)    Q(592.832|72.613)  R(640.29|109.811)
D24:   P(615.358|0.0)    Q(641.096|113.36)  R(684.001|87.511)
D33:   P(618.647|0.0)    Q(657.149|53.279)  R(688.825|0.0)
D27:   P(688.825|0.0)    Q(666.544|37.477)  R(693.01|43.766)
D28:   P(688.825|0.0)    Q(694.448|58.808)  R(704.146|29.347)
D34:   P(688.825|0.0)    Q(783.14|180.659)  R(767.048|95.348)
```

```
D30:    P(688.825|0.0)         Q(716.055|19.067)     R(759.889|0.0)
D29:    P(759.889|0.0)         Q(749.294|73.707)     R(786.809|108.717)
D31:    P(759.889|0.0)         Q(779.267|78.259)     R(808.929|59.44)
D36:    P(759.889|0.0)         Q(819.483|41.623)     R(834.895|0.0)
Gesamtdistanz: 759.889
```




### 3.6 Diskussion

Die ersten beiden Beispiele stellen überhaupt kein Problem dar, und die Ergebnisse werden innerhalb von Bruchteilen von Sekunden geliefert. Die Anordnungen entsprechen in etwa dem, was man von einem Menschen erwarten würde.
Auch bei dem etwas komplexerem Beispiel 3 ist die Leistung des Programms immer noch hervorragend. Die Lücken in der Anordnung sind ziemlich minimal.
Beispiel 4 zeigt nochmals eine Erhöhung der Komplexität. Die Rechenzeit wird merklich länger und erreicht einige Sekunden. Die gefundene Anordnung ist aber trotzdem zufriedenstellen, wobei aber eine große Lücke nahe $x = 350$ zu beobachten ist.
In Beispiel 5 ist ein Fall zu sehen, bei dem die Komplexität ein hohes Maß erreicht. Innerhalb von etwa 12 Sekunden findet das Programm aber eine insgesamt sehr gute Lösung. Vor allem bei höheren $x$-Werten ist aber zu beobachten, dass mit einer geringeren Auswahl an Dreiecken mehr Lücken zu beobachten sind, die nicht optimal gefüllt sind. Die Berechnungszeit wird größtenteils durch das Optimierungsverfahren beeinflusst, welches verschiedene Parameter für die Platzierung ausprobiert. Ohne ein solches Verfahren würde das Programm zwar auch große Mengen an Dreiecken innerhalb von Sekundenbruchteilen bewältigen können, worunter die Effizienz bei der Platzierung natürlich leiden würde.

## 4 Quellcode

```python
#!/usr/bin/env python3
# Python 3.7

import sys
import os
import numpy as np
import math
import matplotlib.pyplot as plt
import copy
import itertools
```

```python


def main():
    # Festlegen der Standarddatei
    file_name = os.path.join(os.path.dirname(__file__), "dreiecke5.txt")

    if len(sys.argv) == 2:
        file_name = os.path.join(os.path.dirname(__file__), sys.argv[1])

    # Einlesen der Datei
    input_file = open(file_name, 'r')
    str_data = input_file.read().splitlines()
    input_file.close()
    str_data_split = [list(map(int, string.split(' '))) for string in str_data]

    # Erzeugen der Dreiecke
    triangles = [Triangle([np.array([element[i], element[i + 1]])
                 for i in range(1, 6, 2)], 'D' + str(index + 1))
                 for index, element in enumerate(str_data_split[1:])]

    # Optimieren der Parameter a, b und c
    epochs = 30 if len(triangles) < 50 else round(30 * 50 / (len(triangles)**1.4))
    a, b, c = optimize(triangles, epochs)

    # Anordnung der Dreiecke
    arranged_triangles = arrange(triangles, delta_a1=a, delta_a2=b, delta_b=c)

    # Ausgabe von Text und Grafik
    for triangle in arranged_triangles:  # TODO remove splice
        triangle.print()
        triangle.plot()
    print('Gesamtdistanz: ' + str(round(arranged_triangles[-1].x, 3)))
    plt.axis('equal')
    plt.axhline(y=0, color='grey')
    plt.show()


# Findet eine moegliche Anordnung der Dreiecke
def arrange(p_triangles, delta_a1=10, delta_a2=25, delta_b=30):
    triangles = copy.deepcopy(p_triangles)
    current_x = 0
    current_angle = 0
    placed_triangles = []

    while triangles:
        remaining_angle = 180 - current_angle
        # Erstellt eine Liste aller Winkel und der Indizes ihres zugehoerigen Dreiecks
        angles = list(itertools.chain(*[[(index, angle) for angle in triangle.angles]
                                        for index, triangle in enumerate(triangles)]))

        # Sucht ein Dreieck, das gut in den Winkel passt, der aktuell offen ist
        triangle = None
        for index, angle in angles:
            if abs(remaining_angle - angle) < delta_a1 and angle < delta_a2:
                triangle = triangles[index]
                break

        # Wird ausgefuehrt, wenn ein potentielles Dreieck gefunden wurde
        if triangle is not None:
            current_x, current_angle = triangle.place_on_fitting_angle(current_x,
                                                                       current_angle, placed_triangles)
```

```
        else:
            # Falls kein Dreieck die Luecke gut fuellt,
            # wird stattdessen ein Dreieck im aktuellen Punkt plaziert
            min_angles = [(index, min(triangle.angles)) for index, triangle in enumerate(triangles)]
            triangle = None
            for index, angle in min_angles:
                if angle + delta_b < remaining_angle:
                    triangle = triangles[index]
                    break

            if triangle is not None:
                current_x, current_angle = triangle.place_on_fitting_side(current_x,
                                                                          current_angle, placed_triangles)
            # Passt kein Dreieck in den aktuellen Punkt,
            # wird das am besten passende Dreieck entlang der x-Achse plaziert
            else:
                angle_diffs = [(index, abs(remaining_angle - angle)) for index, angle in angles]
                triangle = triangles[min(angle_diffs, key=lambda n: n[1])[0]]
                current_x, current_angle = triangle.place_on_fitting_angle(current_x,
                                                                           current_angle, placed_triangles)

        placed_triangles.append(triangle)
        remove_by_id(triangles, triangle)

    return placed_triangles


# Optimiert die Parameter a, b, c fuer die gegebenen Daten
def optimize(data, epochs):
    a, b, c = 10, 25, 30

    arrangement = arrange(data, delta_a1=a, delta_a2=b, delta_b=c)
    baseline = arrangement[-1].x
    parameters = {baseline: (a, b, c)}
    # repetitions = 0

    for i in range(epochs):
        print('\r' + '#' * (i + 1) + '-' * (epochs - (i + 1)) + ' '
              + str(i + 1) + '/' + str(epochs), flush=True, end='')
        delta = 10 / (1.02 ** i)

        a += 0.1 * delta
        arrangement = arrange(data, delta_a1=a, delta_a2=b, delta_b=c)
        a_x = arrangement[-1].x
        if a_x > baseline:
            a -= 2 * 0.1 * delta
        elif math.isclose(a_x, baseline):
            a -= 0.1 * delta
        parameters[a_x] = (a, b, c)

        b += delta
        arrangement = arrange(data, delta_a1=a, delta_a2=b, delta_b=c)
        b_x = arrangement[-1].x
        if b_x > baseline:
            b -= 2 * delta
        elif math.isclose(b_x, baseline):
            b -= delta
        parameters[b_x] = (a, b, c)

        c += delta
        arrangement = arrange(data, delta_a1=a, delta_a2=b, delta_b=c)
```

```
        c_x = arrangement[-1].x
        if c_x > baseline:
            c -= 2 * delta
        elif math.isclose(c_x, baseline):
            c -= delta
        parameters[c_x] = (a, b, c)
        baseline = c_x

    print('\r ')
    return parameters[min(parameters)]


class Triangle:
    def __init__(self, vertices, identification=''):
        self.angles = [calculate_angle(vertices[i - 1], vertices[i], vertices[(i + 1) % 3])
                       for i in range(3)]
        self.lengths = [np.linalg.norm(vertices[i - 1] - vertices[(i + 1) % 3])
                        for i in range(3)]
        self.x = None
        self.rotation_angle = None
        self.base_angle = None
        self.left_side = None
        self.right_side = None
        self.placement_type = None
        self.ID = identification

    # Plaziert das Dreieck so, dass der Kleinste Winkel an der x-Achse liegt
    def place_on_fitting_side(self, x, angle, placed_triangles):
        self.placement_type = 'side'
        self.x = x
        self.rotation_angle = angle

        # Der kleinste Winkel soll als Basis dienen
        smallest_angle_index = self.angles.index(min(self.angles))
        self.base_angle = self.angles[smallest_angle_index]

        # Wenn der aktuelle Platzierungswinkel kleiner als 90 Grad ist, soll die groessere Kante rechts
        # liegen, wenn er groesser als 90 Grad ist, links, ausser, die x-Position ist 0
        l1 = self.lengths[smallest_angle_index - 1]
        l2 = self.lengths[(smallest_angle_index + 1) % 3]
        longer = l1 if l1 > l2 else l2
        shorter = l2 if l1 > l2 else l1
        self.left_side = longer if angle > 90 or x == 0 else shorter
        self.right_side = shorter if angle > 90 or x == 0 else longer

        # Justieren des Winkels, damit keine Kollision mit vorher platzierten Dreiecken stattfindet
        for triangle in placed_triangles:
            if triangle.placement_type == 'angle':
                p1 = triangle.get_right_top_vertex()
                p2 = triangle.get_left_top_vertex()
                p3 = triangle.get_base_vertex()
            else:
                p1 = triangle.get_base_vertex()
                p2 = triangle.get_right_top_vertex()
                p3 = triangle.get_left_top_vertex()

            qb, ql, qr = self.get_vertices()
            if intersects(p1, p2, qb, ql):
                self.adjust_angle(p1, p2, True)

            qb, ql, qr = self.get_vertices()
```

```
            if intersects(p3, p1, qb, ql) or intersects(p3, p1, ql, qr):
                self.adjust_angle(p2, p3, True)

            qb, ql, qr = self.get_vertices()
            if intersects(p1, p2, qr, qb):
                self.adjust_angle(p1, p2, False)

            qb, ql, qr = self.get_vertices()
            if intersects(p1, p2, qb, ql) or intersects(p2, p3, qb, ql):
                self.adjust_angle(p2, p3, True)

            qb, ql, qr = self.get_vertices()
            if intersects(p2, p3, qr, qb):
                self.adjust_angle(p2, p3, False)

            qb, ql, qr = self.get_vertices()
            if intersects(p2, p3, ql, qr):
                self.adjust_angle(p2, p3, True, choose_max=True)

        new_x = x
        new_angle = self.rotation_angle + self.base_angle
        return new_x, new_angle

    # Justiert den Winkel so, dass keine Kollision mit der Kante p1p2 vorliegt
    def adjust_angle(self, p1, p2, left, choose_max=False):
        beta_rad = np.arctan((p1[1] - p2[1]) / (p1[0] - p2[0]))
        c = self.x - p1[0]
        b = self.left_side if left else self.right_side
        max_angle = np.rad2deg(np.arctan(p2[1] / (self.x - p2[0])))
        max_angle = max_angle if max_angle >= 0 else max_angle + 180
        if not choose_max:
            gamma_rad = np.arcsin(c * np.sin(beta_rad) / b)
            alpha = np.rad2deg(np.pi - beta_rad - gamma_rad)
            self.rotation_angle = max_angle if alpha > max_angle else alpha
        else:
            self.rotation_angle = max_angle

    # Plaziert das Dreieck so, dass die am besten passende Ecke den restlichen Winkel fuellt,
    # und eine Kante an der x-Achse liegt
    def place_on_fitting_angle(self, x, angle, placed_triangles):
        self.placement_type = 'angle'
        remaining_angle = 180 - angle
        # Findet den Winkel mit der kleinsten positiven Differenz zum gesuchten Wert
        best_angle_index = min(enumerate([remaining_angle - current_angle
                               for current_angle in self.angles]),
                               key=lambda n: n[1] if n[1] >= 0 else (abs(n[1] - 180)))[0]

        self.base_angle = self.angles[best_angle_index]
        self.x = x

        l1 = self.lengths[best_angle_index - 1]
        l2 = self.lengths[(best_angle_index + 1) % 3]
        self.left_side = l2 if l1 > l2 else l1
        self.right_side = l1 if l1 > l2 else l2
        self.rotation_angle = 180 - self.base_angle  # Das Dreieck soll flach auf dem Boden liegen

        # Verschieben des Dreiecks nach Rechts, falls es nicht in die aktuelle Luecke passt
        if len(placed_triangles) != 0 and self.base_angle > remaining_angle:
            self.adjust_x(placed_triangles[-1])

        for triangle in placed_triangles:
```

```python
            p1, p3, p2 = triangle.get_vertices()
            qb, ql, qr = self.get_vertices()
            if (intersects(p1, p2, qb, ql) or intersects(p2, p3, qb, ql)
                or intersects(p3, p1, qb, ql) or intersects(p1, p2, ql, qr)
                or intersects(p2, p3, ql, qr) or intersects(p3, p1, ql, qr)):
                self.adjust_x(triangle)

        new_x = self.x + self.right_side
        new_angle = calculate_angle(np.array([self.x, 0]), self.get_right_top_vertex(),
                                    self.get_left_top_vertex())
        return new_x, new_angle

    # Passt den x-Wert des Dreiecks so an, dass es in die Luecke passt
    def adjust_x(self, triangle):
        if triangle.get_right_top_vertex()[1] <= self.get_left_top_vertex()[1]:
            self.x = triangle.get_right_top_vertex()[0] \
                     - triangle.get_right_top_vertex()[1] / np.tan(np.deg2rad(self.base_angle))
        else:
            y_diff = triangle.get_right_top_vertex()[1] - self.get_left_top_vertex()[1]
            m_triangle = np.tan(np.deg2rad(180 - triangle.rotation_angle - triangle.base_angle))
            m_self = np.tan(np.deg2rad(self.base_angle))
            x_diff = y_diff / m_triangle - y_diff / m_self
            self.x = (triangle.get_right_top_vertex()[0]
                      - triangle.get_right_top_vertex()[1] / np.tan(np.deg2rad(self.base_angle))
                      - x_diff)

    def get_right_top_vertex(self):
        return np.array([self.x + self.right_side * np.cos(np.deg2rad(180 -
                         self.base_angle - self.rotation_angle)),
                         self.right_side * np.sin(np.deg2rad(180 -
                         self.base_angle - self.rotation_angle))])

    def get_left_top_vertex(self):
        return np.array([self.x + self.left_side * np.cos(np.deg2rad(180 - self.rotation_angle)),
                         self.left_side * np.sin(np.deg2rad(180 - self.rotation_angle))])

    def get_base_vertex(self):
        return np.array([self.x, 0])

    def get_vertices(self):
        return self.get_base_vertex(), self.get_left_top_vertex(), self.get_right_top_vertex()

    def intersects(self, p1, p2):
        return intersects(np.array([self.x, 0]), self.get_right_top_vertex(), p1, p2)

    def plot(self):
        points = [np.array([self.x, 0]), self.get_right_top_vertex(), self.get_left_top_vertex()]
        x = [point[0] for point in points]
        y = [point[1] for point in points]
        plt.fill(x, y)

    def print(self):
        p1, p2, p3 = self.get_vertices()
        p1, p2, p3 = np.around(p1, 3), np.around(p2, 3), np.around(p3, 3)
        p = 'P({}|{})'.format(p1[0], p1[1])
        q = 'Q({}|{})'.format(p2[0], p2[1])
        r = 'R({}|{})'.format(p3[0], p3[1])
        output = '{:3}:    {:20} {:20} {:20}'.format(self.ID, p, q, r)
        print(output)
```

```python
# Berechnet den Winkel an b, in grad
def calculate_angle(a, b, c):
    return np.rad2deg(np.arccos(np.dot(a - b, c - b) / (np.linalg.norm(a - b) * np.linalg.norm(c - b))))


# Bestimmt, ob die Geradensegmente p1p2 und q1q2 sich auf eine problematische Weise schneiden
def intersects(p1, p2, q1, q2, recall=True):
    w = np.subtract(p1, q1)
    rotation_matrix = np.array([[0, 1],
                                [-1, 0]])
    v_perp = np.subtract(q2, q1).dot(rotation_matrix)
    numerator = (v_perp * -1).dot(w)
    denominator = v_perp.dot(np.subtract(p2, p1))

    if denominator == 0:
        return False
    else:
        result = numerator / denominator
        if math.isclose(result, 0, abs_tol=1e-6):
            result = 0
        elif math.isclose(result, 1, abs_tol=1e-6):
            result = 1
        if recall:
            cond2 = intersects(q1, q2, p1, p2, recall=False)
            return 0 < result < 1 and cond2
        else:
            return 0 < result < 1


def remove_by_id(triangles, triangle):
    for t in triangles:
        if t.ID == triangle.ID:
            triangles.remove(t)
            break


if __name__ == '__main__':
    main()
```