Aufgabe 2: Dreiecksbeziehungen

Teilnahme-Id: 48825

Bearbeiter dieser Aufgabe:
Sammy Sawischa

29. April 2019

Inhaltsverzeichnis

1 Lösungsidee

Bei der vorliegenden Aufgabe handelt es sich um ein Optimierungsproblem, bei dem eine Liste von gegebenen Dreiecken mit kleinstmöglichem Gesamtabstand angeordnet werden sollen. Der Gesamtabstand ist dabei die kleinste Distanz der Dreiecke, die am weitesten voneinander entfernt sind. Dieses Optimierungsproblem lässt sich auf den ersten Blick nicht effizient lösen, sofern die gefundene Lösung optimal sein soll, da zu viele Anordnungen in Betracht gezogen werden müssen. Stattdessen wurde bei dieser Aufgabe auf einen Approximationsalgorithmus gesetzt. Damit der Gesamtabstand möglichst gering ist, müssen die Dreiecke so nah aneinander wie möglich angeordnet werden. Zudem sollen dabei die Verluste beim Gesamtabstand möglichst gering sein.

Der entwickelte Approximationsalgorithmus setzt sich hierbei aus folgenden Schritten zusammen:

1. Teile jedem Dreieck die Heuristik H zu (setzt sich zusammen aus dem kleinsten Winkel des Dreiecks und der längsten Seite im Vergleich zu den anderen Dreiecken) und füge es zur Liste l hinzu.
2. Sortiere l absteigend nach H
3. Beginne eine neue Phase (Innerhalb einer Phase werden Dreiecke kreisförmig aneinander angeordnet, wobei der kleinste Winkel des anzuordnenden Dreiecks nach innen zeigt. Die Gesamtkapazität einer Phase beträgt somit 180° bzw. π)
4. Beginne nun die Dreiecke zu legen (bis keine mehr in l vorhanden sind), angefangen mit dem ersten Element in l. Lege dabei das aktuelle Dreieck, sofern es nicht das erste ist, mit der längsten Seite c an die rechte Kante a des vorherigen Dreiecks. Ansonsten lege es an die Koordinatenachse mit Seite c. Entferne das aktuelle Dreieck von l. Wenn ein Dreieck die Phase 90° bzw. $\frac{\pi}{2}$ überschreitet, beginne damit die Dreiecke mit dem letzten Element von l zu legen und entferne es anschließend von der Liste.
5. Wiederhole Schritt 4 solange, bis der kleinste Winkel des zu setzenden Dreieck den übrigen Winkel der aktuellen Phase überschreiten würde.

6. Lege das erste Dreieck von l mit der Seite c auf der Koordinatenachse so nah wie möglich an das zuletzt gelegte Dreieck und entferne es anschließend von l.
7. Lege das neue erste Dreieck von l so nah wie möglich an das vorherige Dreieck, ohne irgendwelche anderen Dreiecksseiten zu schneiden.
8. Gehe zurück zu Schritt 3, um eine neue Phase zu starten.

Die zuvor definierte Phase bestehend aus π kann mithilfe eines Halbkreises dargestellt werden, auf dem die Dreiecke mit ihren kleinsten Winkeln nacheinander angeordnet werden (s. Abbildung 1).

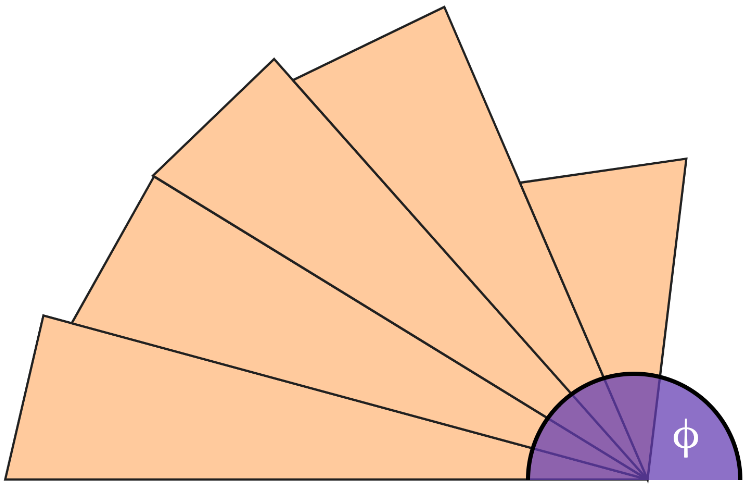


Abbildung 1: Beispiel einer Phase mit verbleibendem Winkel ϕ

1.1 Auswahl der Heuristik

Die ausgewählte Heuristik soll die Approximation so gut wie möglich machen. Damit der Gesamtabstand möglichst gering ist, sollten diejenigen Dreiecke möglichst weit links angeordnet werden, die eine lange Seite haben und einen möglichst kleinen Winkel, da eine lange Seite zwischen zwei Phasen große Einbußen im Gesamtabstand zufolge hat. Des Weiteren nehmen Dreiecke mit kleinen Winkel wenig Platz innerhalb einer Phase ein, was nicht auf die bereits beschriebene Platzierung zwischen zwei Phasen zutrifft. Aus diesen beiden Eigenschaften wurde für die Heuristik H folgende Formel gewählt:

$$H = \frac{\text{Länge der längsten Seite des Dreiecks}}{\text{Gesamtlänge aller längsten Seiten}} - \frac{\text{Kleinster Winkel des Dreiecks}}{\text{Summe aller kleinsten Winkel}}$$

1.2 Das Anordnen im Detail

Ein Dreieck wird in l so definiert, dass es über die längste Seite c verfügt sowie die zwei benachbarten Winkel α und β. Durch die Zuweisung dieser drei Eigenschaften lassen sich gemäß des Kongruenzsatzes WSW kongruente Dreiecke konstruieren, sodass die Möglichkeit des Spiegelns bereits eingeschlossen ist, um ideale Anordnungen zu finden. Wurde bereits ein Dreieck mit der Seite c auf die Koordinatenachse gelegt, wobei sich der Punkt A links und der Punkt B dieses Dreiecks rechts befindet, so muss die zu setzende Kante $\mathrm{c}_{\mathrm{neu}}$ des nächsten Dreiecks auf der Seite a ($\overline{BC}$) des vorherigen Dreiecks liegen. Um nun das nächste Dreieck zu setzen, kann man den Punkt $\mathrm{B}_{\mathrm{neu}}$ dieses Dreiecks an den Punkt B des vorherigen Dreiecks setzen. Für den zu setzenden Punkt $\mathrm{A}_{\mathrm{neu}}$ gilt:

$$\overrightarrow{O\mathrm{A}_{\mathrm{neu}}} = \overrightarrow{OB} + \overrightarrow{B\mathrm{A}_{\mathrm{neu}}}$$

$$<=> \overrightarrow{O\mathrm{A}_{\mathrm{neu}}} = \overrightarrow{OB} + \frac{\mathrm{c}_{\mathrm{neu}}}{|\overrightarrow{BC}|} * \overrightarrow{BC}$$

Mit anderen Worten: Durch Skalieren der vorherigen Kante $\overline{BC}$ auf die Länge von c_{neu} erhält man den Punkt A_{neu}. Zum Konstruieren des Punktes C_{neu} lässt man einfach zwei Geraden sich schneiden, die von den Punkten A und B ausgehen und den Steigungswinkel α bzw. β besitzen. Der Schnittpunkt dieser Geraden ist dann der Punkt C_{neu}.

Sobald der kleinste Winkel eines zu legenden Dreiecks größer ist als der verbleibende Winkel einer Phase, wird dieses Dreieck gemäß dem soeben beschriebenen Algorithmus mit der Seite c auf die Koordinatenachse gelegt (mit dem Punkt A links und dem Punkt B rechts). Dabei soll die Seite b_{neu} den Punkt C des vorherigen Dreiecks gerade so berühren, damit das Dreieck der neuen Phase möglichst nah am vorherigen liegt, ohne dass sie sich überschneiden. Hierbei werden α und β getauscht, sodass α der kleinere Winkel ist. Zur Ermittlung des Punktes A_{neu} stellt man eine Funktionsgleichung y auf mit α_{neu} als Steigungswinkel und C als Punkt dieser Funktion (s. Abbildung 2).

Daraus folgt:

$$y = m * x + b$$
$$<=> y = tan(\alpha_{\text{neu}}) * x + b.$$

Die Nullstelle dieser Geradengleichung ist demzufolge die x-Koordinate des Punktes (A_{neu}). Durch Einsetzen des Punktes $C(c_x|c_y)$ erhält man für b und somit für die Nullstelle:

$$c_y = tan(\beta_{\text{neu}}) * c_x + b \quad | - tan(\beta_{\text{neu}}) * c_x$$
$$<=> b = c_y - tan(\beta_{\text{neu}}) * c_x.$$
$$0 = tan(\beta_{\text{neu}}) * x + c_y - tan(\beta_{\text{neu}}) * c_x$$
$$<=> x_n = \frac{-c_y + tan(\beta_{\text{neu}}) * c_x}{tan(\beta_{\text{neu}})}.$$

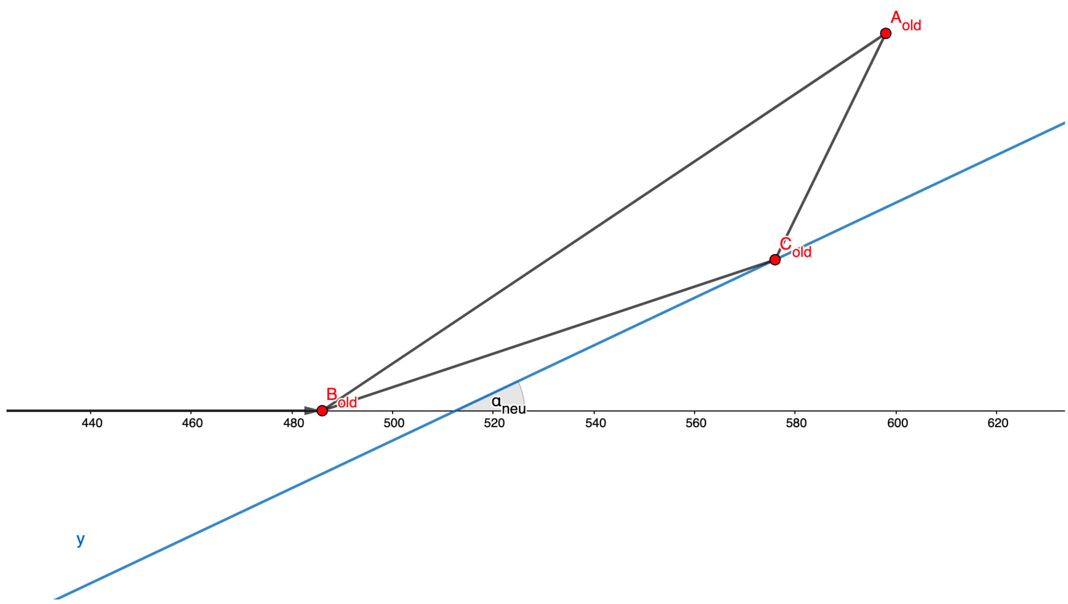


Abbildung 2: Der Beginn einer neuen Phase. Die Gerade der Funktion y repräsentiert die fortgeführte Kante b_{neu} des nächsten Dreiecks.

Nachdem eine neue Phase begonnen wurde, muss das nächste Dreieck wieder mit c_{neu} an die Kante b gelegt werden. Im Gegensatz zur ersten Phase besteht hier die Möglichkeit, dass es zu Überschneidungen mit einem zuvor gesetzten Dreieck kommt. Aus diesem Grund muss das Dreieck eventuell justiert werden. Wenn c_{neu} ein Dreieck der vorherigen Phase an dessen Kante a schneidet, wird c_{neu} in Richtung des Punktes C des vorherigen Dreiecks rotiert. Anschließend muss überprüft werden, ob die neu konstruierte Kante c_{neu} das vorherige Dreieck an der Kante b schneidet. Ist dies der Fall, so wird c_{neu} erneut rotiert, diesmal in Richtung Punkt A des vorherigen Dreiecks.

1.3 Laufzeitanalyse und Alternativen

Das Anordnen an sich erfolgt in $O(n)$. Das Sortieren der Dreiecke nach ihren Heuristiken hingegen hat mit $O(n * log(n))$ die höchste Zeitkomplexität. In der Praxis jedoch ist diese nicht bemerkbar, zumal das Sortieren nur einmal zu Beginn erfolgen muss.

Die Schnelligkeit ist ein besonderer Vorteil des entwickelten Approximationsalgorithmus. Andererseits kann keine optimale Lösung garantiert werden. Wenn man diese dennoch garantieren möchte, muss auf ein anderes Verfahren wie beispielsweise das Backtracking [1] zurückgegriffen werden. Der Nachteil hierbei besteht in der exponentiellen Laufzeit im Worstcase, auch wenn diese in der Praxis nicht erreicht wird, da bei der Tiefensuche Teillösungen, die nicht zu einem besseren Gesamtergebnis führen (hier ein kürzerer Gesamtabstand) ignoriert werden.

2 Umsetzung

Die Lösungsidee wurde in Python implementiert. Nach Starten des Programms kann man im Interface eine Umgebung auswählen. Anschließend wird die Anordnung der Dreiecke berechnet.

`class Point`

(repräsentiert einen Punkt mit seinen x- und y-Koordinaten)

def euclidean_distance(self, other_point)	Berechnet den euklidischen Abstand zweier Punkte mithilfe des Satz des Pythagoras.

`class Edge`

(repräsentiert eine Strecke mit zwei definierten Punkten)

def does_intersect_with (self, other_edge)	Gibt zurück, ob sich zwei Kanten schneiden, die als Geraden fortgeführt werden. Um den Schnittpunkt der fortgeführten Kanten zu erhalten, müssen dessen Geradengleichungen gleichgesetzt werden: $$m_1 * x + b_1 = m_2 * x + b_2 \quad \vert - b_1$$ $$<=> m_1 * x = m_2 * x + b_2 - b_1 \quad \vert - (m_2 * x)$$ $$<=> (m_1 - m_2) * x = b_2 - b_1 \quad \vert : (m_1 - m_2)$$ $$<=> x = \frac{b_2 - b_1}{m_1 - m_2}$$ Durch Umformen erhält man den x-Wert des Schnittpunkts bei $(b2 - b1)/(m1 - m2)$. Zudem wird in der Methode der Fall beachtet, wenn die eine Kante parallel zur y-Achse ist, da dann die Steigung unendlich wäre. Liegt der Schnittpunkt auf beiden Kanten, dann schneiden sie sich.
def get_ratio (self, point_a, point _b)	Gibt den Faktor zurück, mit dem die Kante erweitert werden muss, um auf einen Punkt X (der auf der fortgeführten Kante als Gerade liegt) aufzutreffen. Dazu wird das Teilverhältnis des Vektors $\overrightarrow{AB}$ zu $\overrightarrow{AX}$ berechnet.

[1] `https://de.m.wikipedia.org/wiki/Backtracking`

`class TrianglePlacement`

(enthält Kernmethoden zur Lösung der Aufgabe)

def arrange_triangles (sorted_triangles)	Erhält als Parameter eine bereits nach der Heuristik sortierte Liste an Dreiecken und gibt die Liste der angeordneten Dreiecke zurück. Dazu wird zunächst das erste Dreieck gesetzt. Anschließend kommt es zum rekursiven Methoden-Aufruf von place_triangle().
def place_triangle (previous, to_place, output)	Erhält als Paramter die zu setzenden Dreiecke, das zuvor gesetzte Dreieck und die aktuelle Anordnung. Gemäß dem Approximationsalgorithmus wird geschaut, ob das zu setzende Dreieck einen verbleibenden Winkel der Phase von weniger als 90° zur Folge hätte. Falls ja, wird to_place von hinten abgearbeitet, ansonsten von vorne. Wenn ansonsten kein Sonderfall vorliegt (Überschreitung der Phase oder Beginn einer neuen Phase), wird einfach die c Kante des zu setzenden Dreiecks gerichtet und mithilfe der Methode construct_point_c() der Klasse Triangle wird der Punkt C konstruiert. Wurde ein Dreieck gültig gesetzt, kommt es zum rekursiven Methodenaufurf, sofern die Liste output nicht leer ist (Rekursionsanker).

`class Triangle`

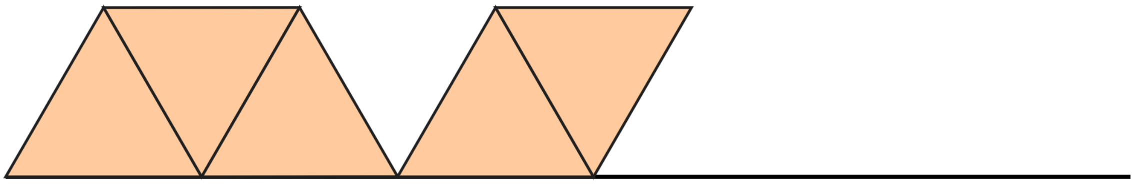
(repräsentiert ein Dreieck mit seiner längsten Seite c und den zwei anliegenden Winkeln)

def construct_point _c (self)	Berechnet den Punkt C eines Dreiecks ausgehend von der Länge der Seite c und den zwei anliegenden Winkeln. Dies geschieht mithilfe des Sinus-Satzes und der Rotationsmatrix (s. Aufgabe 1).

3 Beispiele

Im Folgenden werden alle Programmausgaben sowie die erzeugten SVG-Bilder der gegebenen Beispiele dargestellt.

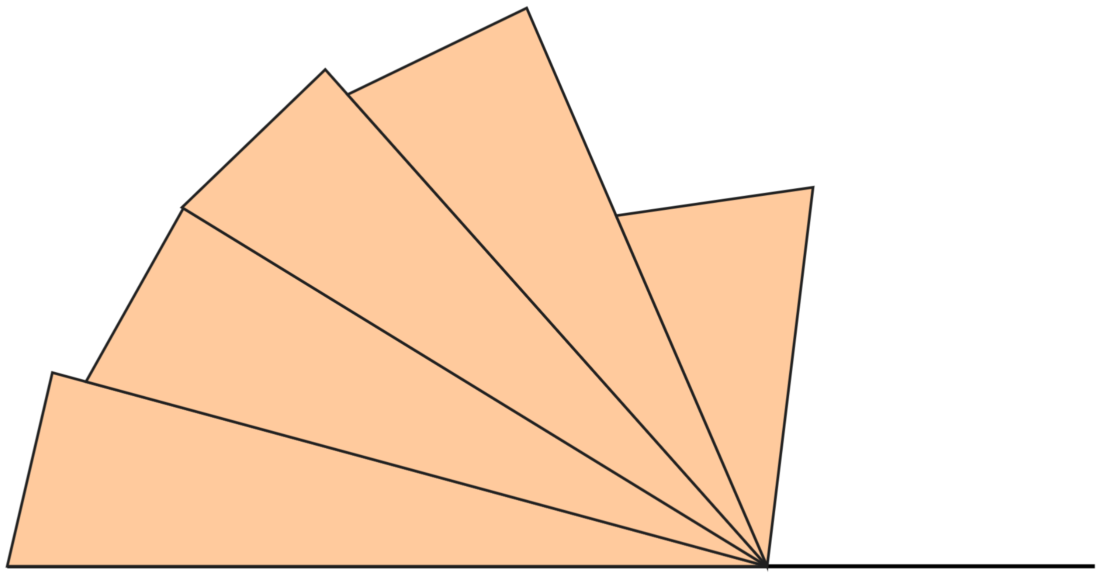
dreiecke1.txt



```
Gesamtsbtand 285.67
Anordnung der Dreiecke:
D1: A (0.0 | 0.0) , B (142.87 | 0.0) , C (71.4 | 123.71)
D2: A (71.4 | 123.71) , B (142.87 | 0.0) , C (214.23 | 123.78)
D3: A (214.23 | 123.78) , B (142.87 | 0.0) , C (285.75 | 0.13)
D4: A (285.67 | 0.0) , B (428.55 | 0.0) , C (357.15 | 123.71)
D5: A (357.13 | 123.74) , B (428.55 | 0.0) , C (499.96 | 123.74)
```

Zwar ist keine Anordnung mit kürzerem Gesamtabstand möglich, dennoch sieht man, dass der Algorithmus hier bei weiteren Dreiecken keine optimale Lösung liefern würde. Man kann hierbei klar sehen, wie zwei Phasen gebildet worden sind.

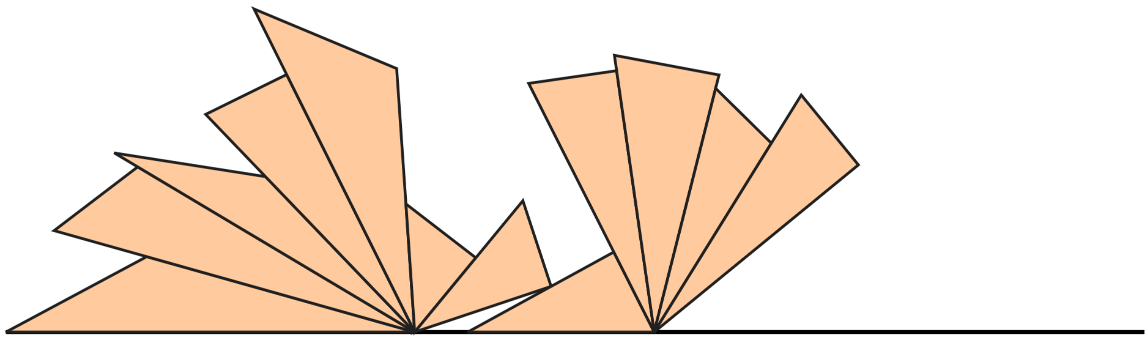
dreiecke2.txt



```
Gesamtsbtand 0.0
Anordnung der Dreiecke:
D1: A (0.0 | 0.0) , B (572.94 | 0.0) , C (34.03 | 146.09)
D2: A (59.36 | 139.22) , B (572.94 | 0.0) , C (132.78 | 270.02)
D3: A (131.75 | 270.65) , B (572.94 | 0.0) , C (239.79 | 374.71)
D4: A (256.6 | 355.8) , B (572.94 | 0.0) , C (391.66 | 421.07)
D5: A (459.08 | 264.47) , B (572.94 | 0.0) , C (607.55 | 285.85)
```

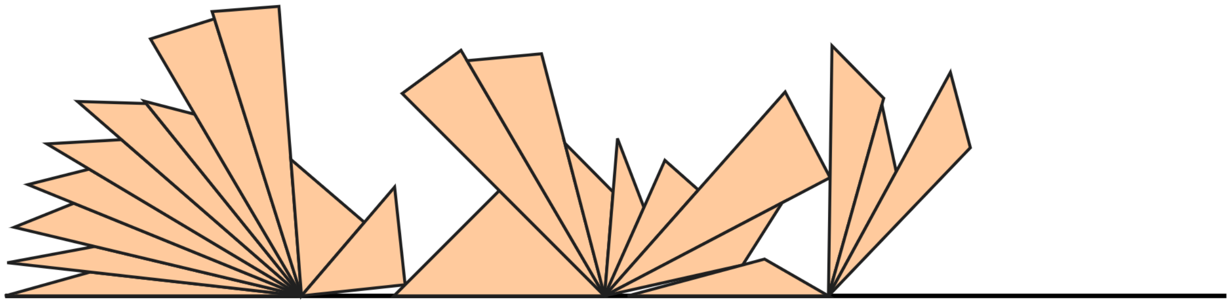
Der Algorithmus hat erfolgreich eine optimale Lösung mit einem Gesamtabstand von 0 gefunden.

dreiecke3.txt



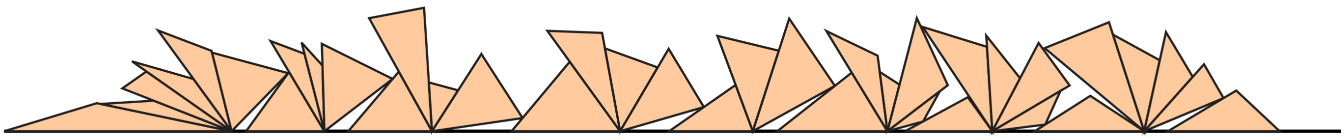
```
Gesamtsbtand 172.5
Anordnung der Dreiecke:
D1: A (0.0 | 0.0) , B (294.38 | 0.0) , C (101.37 | 54.24)
D2: A (34.66 | 72.98) , B (294.38 | 0.0) , C (94.87 | 118.85)
D3: A (78.1 | 128.84) , B (294.38 | 0.0) , C (186.9 | 111.84)
D4: A (143.83 | 156.65) , B (294.38 | 0.0) , C (202.16 | 185.32)
D5: A (178.68 | 232.5) , B (294.38 | 0.0) , C (281.4 | 189.62)
D6: A (288.07 | 92.2) , B (294.38 | 0.0) , C (338.42 | 53.42)
D7: A (372.31 | 94.51) , B (294.38 | 0.0) , C (392.62 | 32.83)
D8: A (332.65 | 0.0) , B (466.89 | 0.0) , C (437.8 | 57.57)
D9: A (376.48 | 178.95) , B (466.89 | 0.0) , C (439.83 | 188.06)
D10: A (438.23 | 199.17) , B (466.89 | 0.0) , C (513.58 | 185.02)
D11: A (511.15 | 175.39) , B (466.89 | 0.0) , C (551.3 | 135.93)
D12: A (572.82 | 170.59) , B (466.89 | 0.0) , C (613.97 | 120.28)
```

dreiecke4.txt



```
Gesamtsbtand 353.85
Anordnung der Dreiecke:
D1: A (0.0 | 0.0) , B (198.93 | 0.0) , C (57.11 | 16.0)
D2: A (1.58 | 22.26) , B (198.93 | 0.0) , C (60.11 | 33.17)
D3: A (6.07 | 46.08) , B (198.93 | 0.0) , C (46.09 | 62.19)
D4: A (15.27 | 74.73) , B (198.93 | 0.0) , C (56.4 | 85.11)
D5: A (27.98 | 102.08) , B (198.93 | 0.0) , C (78.11 | 104.9)
D6: A (48.59 | 130.54) , B (198.93 | 0.0) , C (94.67 | 129.17)
D7: A (93.41 | 130.73) , B (198.93 | 0.0) , C (127.29 | 121.93)
D8: A (97.55 | 172.55) , B (198.93 | 0.0) , C (140.48 | 186.03)
D9: A (138.94 | 190.92) , B (198.93 | 0.0) , C (183.81 | 194.41)
D10: A (191.8 | 91.65) , B (198.93 | 0.0) , C (241.19 | 49.39)
D11: A (261.48 | 73.1) , B (198.93 | 0.0) , C (268.53 | 7.48)
D12: A (261.05 | 0.0) , B (402.47 | 0.0) , C (331.76 | 70.71)
D13: A (266.56 | 135.91) , B (402.47 | 0.0) , C (306.04 | 164.93)
D14: A (310.14 | 157.93) , B (402.47 | 0.0) , C (360.13 | 162.58)
D15: A (375.77 | 102.53) , B (402.47 | 0.0) , C (408.2 | 69.77)
D16: A (411.14 | 105.59) , B (402.47 | 0.0) , C (428.89 | 59.39)
D17: A (442.97 | 91.02) , B (402.47 | 0.0) , C (465.43 | 71.14)
D18: A (523.71 | 136.99) , B (402.47 | 0.0) , C (553.9 | 79.43)
D19: A (521.69 | 62.54) , B (402.47 | 0.0) , C (495.18 | 20.74)
D20: A (418.15 | 0.0) , B (552.78 | 0.0) , C (509.88 | 24.7)
D21: A (555.16 | 167.99) , B (552.78 | 0.0) , C (589.87 | 132.7)
D22: A (588.69 | 128.49) , B (552.78 | 0.0) , C (598.27 | 83.4)
D23: A (634.66 | 150.12) , B (552.78 | 0.0) , C (648.07 | 99.52)
```

dreiecke5.txt



```
Gesamtsbtand 839.2
Anordnung der Dreiecke:
D1: A (0.0 | 0.0) , B (203.8 | 0.0) , C (83.07 | 25.22)
D2: A (86.33 | 24.54) , B (203.8 | 0.0) , C (134.04 | 27.46)
D3: A (105.89 | 38.54) , B (203.8 | 0.0) , C (127.47 | 53.74)
D4: A (114.64 | 62.77) , B (203.8 | 0.0) , C (169.66 | 46.95)
D5: A (137.93 | 90.59) , B (203.8 | 0.0) , C (185.87 | 72.27)
D6: A (186.27 | 70.66) , B (203.8 | 0.0) , C (254.53 | 52.21)
D7: A (216.8 | 0.0) , B (286.97 | 0.0) , C (255.3 | 53.28)
D8: A (238.76 | 81.09) , B (286.97 | 0.0) , C (269.12 | 69.61)
D9: A (266.55 | 79.62) , B (286.97 | 0.0) , C (285.25 | 59.05)
D10: A (284.68 | 78.61) , B (286.97 | 0.0) , C (348.14 | 47.03)
D11: A (308.4 | 0.0) , B (383.41 | 0.0) , C (353.69 | 53.61)
D12: A (327.05 | 101.67) , B (383.41 | 0.0) , C (376.27 | 110.99)
D13: A (380.59 | 43.88) , B (383.41 | 0.0) , C (406.84 | 36.77)
D14: A (427.66 | 69.43) , B (383.41 | 0.0) , C (464.59 | 11.84)
D15: A (454.71 | 0.0) , B (552.02 | 0.0) , C (506.86 | 62.52)
D16: A (486.81 | 90.27) , B (552.02 | 0.0) , C (535.87 | 88.43)
D17: A (538.41 | 74.51) , B (552.02 | 0.0) , C (585.68 | 57.34)
D18: A (595.51 | 74.09) , B (552.02 | 0.0) , C (627.05 | 21.72)
D19: A (595.95 | 0.0) , B (670.95 | 0.0) , C (655.54 | 41.62)
D20: A (639.2 | 85.75) , B (670.95 | 0.0) , C (694.25 | 72.4)
D21: A (703.55 | 101.29) , B (670.95 | 0.0) , C (744.5 | 40.78)
D22: A (692.88 | 0.0) , B (791.59 | 0.0) , C (759.59 | 52.71)
D23: A (736.9 | 90.08) , B (791.59 | 0.0) , C (782.83 | 67.91)
D24: A (784.71 | 53.32) , B (791.59 | 0.0) , C (802.45 | 41.89)
D25: A (817.95 | 101.72) , B (791.59 | 0.0) , C (845.69 | 41.39)
D26: A (838.51 | 35.9) , B (791.59 | 0.0) , C (823.52 | 8.74)
D27: A (807.47 | 0.0) , B (885.58 | 0.0) , C (863.63 | 30.59)
D28: A (823.23 | 93.77) , B (885.58 | 0.0) , C (880.82 | 75.88)
D29: A (880.18 | 85.95) , B (885.58 | 0.0) , C (910.48 | 47.65)
D30: A (927.05 | 79.36) , B (885.58 | 0.0) , C (954.88 | 41.16)
D31: A (946.68 | 36.29) , B (885.58 | 0.0) , C (933.0 | 5.99)
D32: A (923.64 | 0.0) , B (1021.73 | 0.0) , C (973.41 | 31.84)
D33: A (932.55 | 74.74) , B (1021.73 | 0.0) , C (990.16 | 98.0)
D34: A (993.75 | 86.87) , B (1021.73 | 0.0) , C (1035.67 | 63.79)
D35: A (1041.2 | 89.14) , B (1021.73 | 0.0) , C (1065.23 | 48.89)
D36: A (1075.32 | 60.24) , B (1021.73 | 0.0) , C (1092.81 | 29.77)
D37: A (1043.0 | 0.0) , B (1143.13 | 0.0) , C (1104.21 | 36.58)
```

Wie man in den Beispielen 3 bis 5 sieht liefert der Algorithmus eine gute Approximation zu einer optimalen Lösung.

4 Quellcode

```
import os
from math import sin, cos, pi, sqrt, acos, tan

# Repraesentiert einen Punkt in Form einer Koordinate oder einen Vektor mit dessen x- und y-Wert
class Point:
    def __init__(self, x, y):
        self.x = x      # x-Koordinate
        self.y = y      # y-Koordinate

    # Punkt als Zeichenkette
    def __repr__(self):
        return " (" + str(round(self.x, 2)) + " | " + str(round(self.y, 2)) + ") "

    # Vergleich-Funktion
    def __eq__(self, other):
        return (self.x, self.y) == (other.x, other.y)

    # Euklidische Distanz zweier Punkte berechnen (Anwenden des Satzes des Pythagoras)
    def euclidean_distance(self, other_point):
        cathetus_a = other_point.x - self.x
        cathetus_b = other_point.y - self.y
        hypotenuse = sqrt(cathetus_a ** 2 + cathetus_b ** 2)
        return hypotenuse

    # Multipliziert einen Ortsvektor mit einem bestimmten Faktor
    def scale(self, factor):
        return Point(self.x*factor, self.y*factor)

# Repraesentiert ein Dreieck mit einer Seite und dessen zwei anliegenden Winkel
class Triangle:
    def __init__(self, c, alpha, beta, id):
        self.c = c
        self.alpha = alpha
        self.beta = beta
        self.id = id

        self.point_a = None
        self.point_b = None
        self.point_c = None
        self.heuristic = 0

    def gamma(self):
        return pi - self.alpha - self.beta

    def construct_point_c(self):
        # 1. Laenge von BC mit Sinussatz berechnen
        length_bc = abs((self.c * sin(self.alpha)) / sin(self.gamma()))
        # 2. Zu rotierenden Vektor mit Laenge BC in Richtung c
        ratio = length_bc/self.c
        vector_c = Point(self.point_a.x - self.point_b.x, self.point_a.y - self.point_b.y)
        to_rotate = Point(ratio * vector_c.x, ratio * vector_c.y)
        angle = 2*pi - self.beta
        # 3. Ortsvektor OC mithilfe der Rotationsmatrix berechnen
        rotated_x = to_rotate.x * cos(angle) - to_rotate.y * sin(angle)
        rotated_y = to_rotate.x * sin(angle) + to_rotate.y * cos(angle)
        self.point_c = Point(rotated_x + self.point_b.x, rotated_y + self.point_b.y)
```

```
    def construct_point_a(self, target_point):
        # Erforderlich beim Beginn einer neuen Phase
        # Konstruktion des Punktes A, ausgehend vom Punkt B zum Zielpunkt
        ratio = self.c / self.point_b.euclidean_distance(target_point)
        prev_vector_bc = Point(target_point.x - self.point_b.x,
                               target_point.y - self.point_b.y)
        vector_ba = prev_vector_bc.scale(ratio)
        self.point_a = Point(self.point_b.x + vector_ba.x,
                             self.point_b.y + vector_ba.y)

    def __repr__(self):
        return str([self.point_a, self.point_b, self.point_c])

    # Vergleichsoperationen zum Sortieren nach den Heuristiken
    def __eq__(self, other):
        return self.heuristic == other.heuristic

    def __lt__(self, other):
        return self.heuristic > other.heuristic


# Repraesentiert eine Kante, bzw. eine Strecke mit zwei definierten Punkten A und B
class Edge:
    def __init__(self, point_a, point_b):
        self.point_a = point_a
        self.point_b = point_b

    # Prueft, ob zwei Kanten sich schneiden
    def does_intersect_with(self, other_edge):
        # Als allgm. Geradengleichung gilt: y=mx+b.
        # Hier: Zwei Geraden mit jeweils m1 bzwn. m2 und b1 bzw. b2
        # Es gilt: m1 = dy1/dx1 und m2 = dy2/dx2
        dy1 = self.point_b.y - self.point_a.y
        dy2 = other_edge.point_b.y - other_edge.point_a.y
        dx1 = float(self.point_b.x - self.point_a.x)
        dx2 = float(other_edge.point_b.x - other_edge.point_a.x)

        # Wenn Kante a parallel zur y-Achse ist
        if dx1 == 0.0:
            intersection_x = self.point_a.x
            m2 = dy2 / dx2
            b2 = other_edge.point_a.y - m2 * other_edge.point_a.x
            intersection_y = m2 * intersection_x + b2
            ratio_edge_1 = self.get_ratio(Point(intersection_x, intersection_y))
            ratio_edge_2 = other_edge.get_ratio(Point(intersection_x, intersection_y))
            return 0 < ratio_edge_1 < 1 and 0 < ratio_edge_2 < 1

        # Steigung m1 kann berechnet werden, da dx1 ungleich 0 ist
        m1 = dy1 / dx1
        b1 = self.point_a.y - m1 * self.point_a.x

        # Wenn Kante b parallel zur y-Achse ist
        if dx2 == 0.0:
            intersection_x = other_edge.point_a.x
            intersection_y = m1 * intersection_x + b1
            ratio_edge_1 = self.get_ratio(Point(intersection_x, intersection_y))
            ratio_edge_2 = other_edge.get_ratio(Point(intersection_x, intersection_y))
```

```
            return 0 < ratio_edge_1 < 1 and 0 < ratio_edge_2 < 1

        # Steigung m2 kann berechnet werden, da dx2 ungleich 0 ist
        m2 = dy2 / dx2
        b2 = other_edge.point_a.y - m2 * other_edge.point_a.x
        intersection_x = (b2 - b1) / float(m1 - m2)
        intersection_y = m1 * intersection_x + b1
        ratio_edge_1 = self.get_ratio(Point(intersection_x, intersection_y))
        ratio_edge_2 = other_edge.get_ratio(Point(intersection_x, intersection_y))
        return 0 < ratio_edge_1 < 1 and 0 < ratio_edge_2 < 1

    # Erhalte den Faktor mit dem die Kante erweitert werden muss,
    # um auf einen bestimmten Punkt zu treffen
    def get_ratio(self, point):
        # Zu pruefende Kante ist parallel zur Y-Achse --> Verhaeltnis ueber y-Werte
        if self.point_b.x == self.point_a.x:
            ratio = (point.y - self.point_a.y) / float(self.point_b.y - self.point_a.y)
        # Sonst: Verhaeltnis ueber x-Werte
        else:
            ratio = (point.x - self.point_a.x) / float(self.point_b.x - self.point_a.x)

        return ratio


# Klasse, die alle wichtigen Methoden zum Anordnen von Dreiecken enthaelt
class TrianglePlacement:
    remaining_angle = 0     # Verbleibende Winkel einer Phase
    new_phase = False       # True, wenn erstes Dreieck einer neuen

    @staticmethod
    def arrange_triangles(sorted_triangles):
        output = []

        # Anlegen des ersten Dreiecks
        current = sorted_triangles[0]
        sorted_triangles.pop(0)
        current.point_a = Point(0, 0)
        current.point_b = Point(current.c, 0)
        current.id = 0
        current.construct_point_c()
        TrianglePlacement.remaining_angle = pi - current.beta
        output.append(current)

        # Methode zum Setzen aller folgenden Dreiecke
        return TrianglePlacement.place_triangle(current, sorted_triangles, output)

    # Setzen eines Dreiecks an die Kante des vorherigen
    @staticmethod
    def place_triangle(previous, to_place, output):
        # Wenn Dreieck 90 Grad der Phase ueberschreitet...
        if TrianglePlacement.remaining_angle - to_place[0].beta <= pi/2:
            # ... Liste von hinten abarbeiten
            current = to_place[len(to_place)-1]
            to_place.pop(len(to_place)-1)

        else:
            # Ansonsten von vorne abarbeiten
            current = to_place[0]
```

```
            to_place.pop(0)

        current.id = previous.id + 1    # Setzen der ID

        if current.beta > TrianglePlacement.remaining_angle:
            # Eine neue Phase muss gestartet werden
            # Das Dreieck wird mit c so nah wie moeglich ans vorherige geschoben
            to_place.append(current)
            current = to_place[len(to_place)-1]
            to_place.pop(len(to_place)-1)

            # Tausche Beta mit alpha
            save_alpha = current.alpha
            current.alpha = current.beta
            current.beta = save_alpha

            # Zu setzende Kante als Gerade: y(x) = m * x + b
            # y(x) = tan^-1(beta) * x + (c_y - m * c_x)
            m = tan(current.alpha)
            b = previous.point_c.y - m * previous.point_c.x

            # x-Koordinate von Punkt A ist Nullstelle von y(x)
            # 0 = m*x +b <=> x_n = -b/m
            point_a_x = -b / m

            current.point_a = Point(point_a_x, 0)
            current.point_b = Point(current.point_a.x + current.c, 0)
            current.construct_point_c()
            output.append(current)
            TrianglePlacement.remaining_angle = pi - current.beta
            # Naechstes Dreieck kann andere Dreiecke schneiden
            TrianglePlacement.new_phase = True

        else:
            current.point_b = previous.point_b
            if TrianglePlacement.new_phase:
                # Beim ersten Dreieck einer neuen Phase (sofern es nicht die erste ist)
                # muss das Dreieck eventuell justiert werden, damit es nicht
                # zu Ueberschneidungen kommt
                TrianglePlacement.new_phase = False
                current.construct_point_a(previous.point_c)
                additional_angle = 0
                # Iteriere durch jedes gesetzte Dreieck, bis auf das letzte
                for triangle in output[:len(output)-1]:
                    # Aktuelle Kante c
                    edge_c_new = Edge(current.point_b, current.point_a)
                    prev_edge_a = Edge(triangle.point_b, triangle.point_c)  # Vorherige Kante a
                    prev_edge_b = Edge(triangle.point_c, triangle.point_a)  # Vorherige Kante b

                    if prev_edge_a.does_intersect_with(edge_c_new):
                        # Neue Kante c schneidet vorherige Kante a --> Justierung erforderlich
                        current.construct_point_a(triangle.point_c)

                        # Winkel zwischen BC und BAnew berechnen
                        vector_bc = Point(previous.point_c.x - previous.point_b.x,
                                          previous.point_c.y - previous.point_b.y)
                        vector_ba_new = Point(current.point_a.x - previous.point_b.x,
                                          current.point_a.y - previous.point_b.y)
```

```
                            scalar = (vector_ba_new.x * vector_bc.x + vector_ba_new.y * vector_bc.y)
                            length_product = sqrt(vector_ba_new.y**2 + vector_ba_new.x**2) * \
                                             sqrt(vector_bc.y**2 + vector_bc.x**2)
                            additional_angle = acos(scalar/length_product)

                        edge_c_new = Edge(current.point_b, current.point_a)

                        if prev_edge_b.does_intersect_with(edge_c_new):
                            # Neue Kante c schneidet vorherige Kante b --> Justierung erforderlich
                            current.construct_point_a(triangle.point_a)

                            # Winkel zwischen BC und BAnew berechnen
                            vector_bc = Point(previous.point_c.x - previous.point_b.x,
                                              previous.point_c.y - previous.point_b.y)
                            vector_ba_new = Point(current.point_a.x - previous.point_b.x,
                                                  current.point_a.y - previous.point_b.y)
                            scalar = (vector_ba_new.x * vector_bc.x + vector_ba_new.y * vector_bc.y)
                            length_product = sqrt(vector_ba_new.y ** 2 + vector_ba_new.x ** 2) * \
                                             sqrt(vector_bc.y ** 2 + vector_bc.x ** 2)

                            additional_angle = acos(scalar / length_product)

                    # Verbleibenden Winkel aktualisieren
                    TrianglePlacement.remaining_angle -= current.beta + additional_angle
                else:
                    TrianglePlacement.remaining_angle -= current.beta
                    # Berechne Vektor von B zu A (Kante c) durch
                    # Skalieren der Seite BC des vorherigen Dreiecks
                    current.construct_point_a(previous.point_c)

                # Konstruktion des neuen Punktes C
                current.construct_point_c()
                output.append(current)

            if len(to_place) == 0:
                return output

            return TrianglePlacement.place_triangle(current, to_place, output)
```