Aufgabe 2: Dreiecksbeziehungen

Teilnahme-Id: 48933

Bearbeiter/-in dieser Aufgabe:
Oliver Schirmer

26. April 2019

Inhaltsverzeichnis

Lösungsidee

Zu Beginn wird die Eingabedatei eingelesen und die Dreiecke entsprechend den Formatdetails auf der Materialwebsite eingelesen. Da nur Koordinaten vorgegeben sind, müssen die Seitenlängen, mit Hilfe des Satz des Pythagoras, und die Winkel, mit Hilfe des Kosinussatzes, berechnet werden. Zudem wird die Fläche der Dreiecke, mit Hilfe der Heronschen Formel berechnet.

Um die Dreiecke so nah wie möglich aneinander anzuordnen, wird versucht Rechtecke zu bilden, d.h. Paare bei dem ein Dreieck mit einer Seite und das andere mit einer Ecke an der Straße liegt. Um den Abstand der Paare zueinander zu minimieren, wird die kürzeste Seite des Dreiecks, das mit der Seite an der Straße liegt, auf diese platziert. Wichtig ist nun, dass die Paare möglichst einen rechten Winkel in der rechten unteren Ecke des „Rechtecks“ aufweisen, um den Abstand zum nächsten Paar zu minimieren. Da die Dreiecke an ihren längsten Seiten aneinandergelegt werden, müssen die beiden, für die rechte untere Ecke relevanten, Winkel an dieser Seite liegen. Also werden zwei Listen mit den Winkeln an der längsten Seite aller Dreiecke erstellt und eine weitere die die beiden Winkel jeweils als Winkelpaar betrachtet. Letztere ist dafür wichtig, dass für jedes Dreieck der Winkel mit dem besten Paar, d.h. am nächsten zu 90 Grad mit einem Winkel eines anderen Dreiecks, verwendet wird, da nur einer der beiden Winkel genutzt werden kann. Die beiden Listen, mit den einzelnen Winkeln, werden nun aufsteigend nach ihrer Größe sortiert.
Es wird über die Liste mit den Winkelpaaren iteriert und für beide Winkel der best mögliche Partner gefunden. Dafür wird über die beiden Listen mit den einzelnen Winkeln iteriert und geprüft, welcher der beiden Winkel ein näher an 90 Grad liegendes Paar bildet und ein Vergleich zum vorherigen gefundenen Paar durchgeführt. Dann wird geprüft, welches der beiden gebildeten Paare für das momentane Winkelpaar das bessere Paar ist (auch hier: näher an 90 Grad). Dieses wird gespeichert und die 4 Winkel, der beiden in diesem Paar „verbrauchten“ Dreiecke, werden aus den beiden Einzel-Winkellisten entfernt.
Da es auch relevant ist den Überstand in der oberen linken Ecke eines Paares zu minimieren, um den Abstand zum vorherigen Paar zu minimieren, werden die Dreiecke zu Beginn nach ihrer Fläche aufsteigend sortiert.

Nachdem die Paare gefunden wurden, werden diese nacheinander entlang der Straße platziert. Um den Abstand zu minimieren, wird wie bereits gesagt, die kürzeste Seite an der Straße platziert. Dafür wird das Dreieck auch gespiegelt, falls nötig. Das wird dadurch umgesetzt, dass bei den Winkelpaaren, nur der Winkelname gespeichert wird (bspw. Alpha, Beta, Gamma). Da die Seitennamen (a, b, c) durch den Kosinussatz direkt mit den Winkeln verbunden sind, lässt sich dadurch mit Hilfe der zu Beginn errechneten Seitenlängen das Dreieck mit nur einem Winkel vollständig wieder aufbauen.

Ab dem zweiten Paar ist es wichtig zu überprüfen, dass sich die beiden Dreiecke nicht mit dem zweiten Dreieck des letzten Paares überschneiden. Falls doch, wird das neue Paar nach rechts verschoben. Dafür wird beim ersten Dreieck des aktuellen Paares die kurze, nicht an der Straße befindliche Seite, und beim zweiten die lange Seite betrachtet, da diese am weitesten links und damit am nächsten zum vorherigen Dreieck liegen. Schneiden diese beiden Seiten sich nicht mit dem vorherigen Dreieck, tun es die anderen auch nicht. Beim vorherigen Dreieck, ist nur die kurze, am weitesten rechts gelegene Seite relevant.
Die Seite des vorherigen Dreiecks wird im folgenden als f bezeichnet, die Seite des ersten Dreiecks des neuen Paares als g1 und die Seite des zweiten Dreiecks des neuen Paares als g2.
Aus der Strecke f wird eine Funktion f(x) gebildet, indem mit Hilfe der Standardgleichung einer linearen Funktion $y=m*x+n$ das n, unter Einsetzen einer Nullstelle und des Anstiegs berechnet wird. Das Ziel ist nun jeweils für g1 und g2 eine Funktion zu finden, die sich jeweils entweder beim höchsten y-Wert des neuen Dreiecks oder beim höchsten y-Wert des alten Dreiecks schneidet, je nachdem welcher der beiden Werte niedriger ist. Damit wird sichergestellt, dass das neue Paar nicht zu weit nach rechts verschoben wird, da Funktionen Geraden mit unendlicher Länge repräsentieren, die Seiten der Dreiecke jedoch Strecken und somit begrenzt sind. Dieser gewünschte y-Wert des Schnittpunkts von g1 mit f bzw. g2 mit f wird als yCut bezeichnet. Nachdem yCut ermittelt wurde, wird für den Schnittpunkt der x-Wert ermittelt, mit Hilfe von $f(xCut)=yCut$. Nun können wie bereits bei der Berechnung von f(x) g1(x) und g2(x) gebildet werden, durch $yCut=m_{1,2}*xCut+n$, umstellen zu n und Einsetzen des Anstiegs. Von g1(x) und g2(x) werden letztlich die Nullstellen gebildet und geprüft, ob diese weiter rechts liegen als ohne Verschiebung. Ist das der Fall, wird das Paar entsprechend der Nullstellen weiter nach rechts verschoben.

Umsetzung

Für das Einlesen der Informationen, wird ein BufferedReader genutzt und über jede einzelne Zeile der Datei iteriert. Relevant sind nur alle Zeilen, folgend auf die 1. Zeile. Die Koordinaten werden als Point-Objekte, welche 2 Integer-Werte speichern können in einer ArrayList gespeichert. Diese wird, zusammen mit dem Namen, entsprechend der Reihenfolge in der Liste, an ein neu erzeugtes Triangle-Objekt übergeben. Dieses berechnet mit Hilfe der Koordinaten bei Instanziierung die 3 Winkel, Seiten, sowie die Fläche des Dreiecks. Zudem enthält die Klasse Funktionen, um die längste Seite, die beiden Winkel an der längsten Seite, welche zwangsläufig die beiden kleinsten Winkel sein müssen, die benachbarten Seiten eines Winkels und die kurze benachbarte Seite eines Winkels, der an der langen Seite des Dreiecks liegt, zu bestimmen und eine Subklasse, welche die Winkelpaare abspeichert. Vor allem die Funktionen sind für den späteren Neuaufbau des Dreiecks wichtig.
Erwähnenswert für die Winkel und Seiten ist, dass diese eigene Klassen besitzen, die nicht nur Länge bzw Gradzahl, sondern auch Typ, also a, b & c bei den Seiten und Alpha, Beta & Gamma bei den Winkeln speichern. Bei den Winkeln wird zudem noch ein Verweis auf das Dreieck gespeichert, welcher später noch wichtig wird.

Für die Sortierung der Dreiecke nach Fläche und die Sortierung der Winkel nach Gradzahl wird der Java-interne Sortierungsalgorithmus der Klasse List genutzt, welcher einen adaptiven MergeSort-Algorithmus nutzt und somit bei einer bereits teilweise sortierten Liste um einiges weniger Vergleiche, als bei der herkömmlichen MergeSort-Laufzeit $O(n)=n*\log(n)$ benötigt.

Bei der Iteration über die Winkelpaare wird jedoch nur über die Hälfte der Anzahl der Winkelpaare, d.h. über die Hälfte aller Dreiecke iteriert. Das liegt daran, dass theoretisch für jedes Dreieck ein Partner gefunden wird, für den dann logischerweise nicht auch noch ein Partner gesucht werden muss. Ausnahme bildet die Ausgangssituation, dass in der Eingabedatei eine ungerade Anzahl an Dreiecken eingegeben wird. Hier wird das Dreieck mit der größten Fläche einfach direkt anliegend an das letzte Paar mit der langen Seite auf der Straße platziert. Da sowieso der innerste x-Wert eines Dreiecks zur Messung des Gesamtabstands gewählt wird, hat dieses Dreieck keine Auswirkung auf den Gesamtabstand. Das Dreieck mit der größten Fläche wird aus dem Grund gewählt, dass dieses statistisch für die größte Abstandsvergrößerung sorgt und somit „entschärft“ wird.

Bei der Speicherung der Winkelpaare wird nur der Dreiecksname und der Winkeltyp gespeichert, da diese beiden Informationen, wie bereits bei der Lösungsidee erläutert, genügen, um das Dreieck wieder aufzubauen.

Nach der Bildung der Paare werden mit Hilfe der SVG-Library Batik von Apache die Straße und die Dreiecke gezeichnet. In den Beispielen auf der BwInf-Webseite ist die ViewBox auf der x-Achse auf 0 bis 820 begrenzt. Diesen Bereich habe ich auf -200 bis 1500 geändert, um einerseits das erste Dreieck, welches möglicherweise links übersteht voll sichtbar zu machen und bei Beispiel 5 alle Dreiecke sichtbar zu machen.
Zudem war es nötig auf die Font der Schriften einen AffineTransform anzuwenden, da diese sonst, aufgrund der allgemeinen Invertierung der y-Achse, horizontal gespiegelt wäre. Zudem wurde die Schrift etwas nach unten links verschoben, da die Koordinaten, bei denen geschrieben werden soll, nicht als Zentrum des Textes sondern als untere linke Ecke interpretiert werden.

Für die Ausgabe der x- und y-Koordinaten und die Berechnung des Gesamtabstands, wird das zur reinen Darstellung, der neu platzierten Dreiecke, genutzte Path2D-Objekt an ein neu instanziiertes TrianglePath-Objekt übergeben. Dieses liest die Point2D-Objekte, also die einzelnen Koordinaten aus dem Path2D-Objekt aus und berechnet ebenfalls, wie ein normales Triangle-Objekt, die Seiten. Der Hauptunterschied ist hier, dass die Punkte mit Fließkommazahlen als Koordinaten angegeben werden können und dadurch die Genauigkeit steigt. Beim Einlesen der vorgegebenen Dreiecke ist dies noch irrelevant, da diese sowieso nur ganzzahlige Koordinaten enthalten.

Beispiel

Eigenes Beispiel 1:

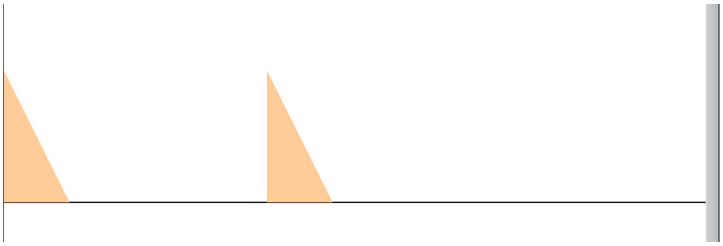
Zur Vereinfachung wurden in diesem Beispiel nur 2 gleichschenklige Dreiecke gewählt. Man erkennt, dass die kurze Seite des kleinen Dreiecks an die Straße gesetzt wurde, woran sich auch die Sortierung nach Fläche direkt erkennen lässt. Auch der Winkel des großen Dreiecks, der als Partner-Winkel gewählt wurde ist der kleinste der 3 Winkel. Die Distanz in diesem Beispiel ist hier nicht relevant, da diese bei 2 Dreiecken im Idealfall immer 0 ist.
Zu beachten ist, dass bei den Ausgangswerten und den Endwerten die Seiten und Winkel gleich sind, jedoch bspw. bei D2 a & b vertauscht wurden, woran sich der Neuaufbau nochmals erkennen lässt.

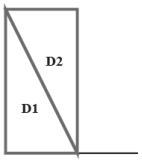
```
---INFOS---
D1: Ecke1: 0.0|0.0 Ecke2: 100.0|0.0 Ecke3: 50.0|100.0 a=100.0 b=111.8 c=111.8 Alpha=53.13
Beta=63.43 Gamma=63.43
D2: Ecke1: 200.0|0.0 Ecke2: 400.0|0.0 Ecke3: 300.0|200.0 a=200.0 b=223.6 c=223.6 Alpha=53.13
Beta=63.43 Gamma=63.43
---ALGORITHMUS---
Algorithmus-Paare:
D1-BETA + D2-ALPHA
---ERGEBNIS---
D1: Ecke1: 0.0|0.0 Ecke2: 100.0|0.0 Ecke3: 49.99|100.0 a=100.0 b=111.8 c=111.8 Alpha=53.13
Beta=63.43 Gamma=63.43
D2: Ecke1: 100.0|0.0 Ecke2: 0.0|200.0 Ecke3: 200.0|200.0 a=223.6 b=200.0 c=223.6 Alpha=63.43
Beta=53.13 Gamma=63.43
Distanz: 0.0
```

Eigenes Beispiel 2:

In diesem Beispiel wurden 2 kongruente rechtwinklige Dreiecke gewählt, um zu veranschaulichen, dass im Optimalfall tatsächlich Rechtecke gebildet werden.

---INFOS---
D1: Ecke1: 0.0|0.0 Ecke2: 50.0|0.0 Ecke3: 0.0|100.0 a=50.0 b=111.8 c=100.0 Alpha=26.56 Beta=90.0 Gamma=63.43
D2: Ecke1: 200.0|0.0 Ecke2: 250.0|0.0 Ecke3: 200.0|100.0 a=50.0 b=111.8 c=100.0 Alpha=26.56 Beta=90.0 Gamma=63.43
---ALGORITHMUS---
Algorithmus-Paare:
D1-GAMMA + D2-ALPHA
---ERGEBNIS---
D1: Ecke1: 0.0|0.0 Ecke2: 50.0|0.0 Ecke3: 0.0|100.0 a=50.0 b=111.8 c=100.0 Alpha=26.56 Beta=90.0 Gamma=63.43
D2: Ecke1: 50.0|0.0 Ecke2: 0.0|100.0 Ecke3: 50.0|100.0 a=111.8 b=50.0 c=100.0 Alpha=89.99 Beta=26.56 Gamma=63.43
Distanz: 0.0

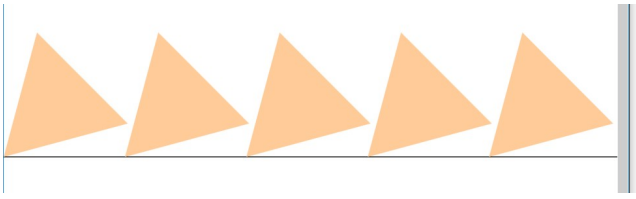


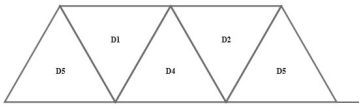


Beispiel 1:

Dieses und alle folgenden Beispiele sind der BwInf-Website entnommen. Es wird auf die Ausgangsinfos, die Seitenlängen und Winkelgrößen verzichtet, um den Output nicht unnötig zu vergrößern. Die Kongruenz der Ausgangs- und Enddreiecke wurde bereits in den eigenen Beispielen 1 und 2 nachgewiesen.

---ALGORITHMUS---
Algorithmus-Paare:
D5-BETA + D1-BETA
D4-BETA + D2-BETA
D5-ALPHA + D5-ALPHA
---ERGEBNIS---
D5: Ecke1: 0.0|0.0 Ecke2: 142.87|0.0 Ecke3: 71.39|123.71
D1: Ecke1: 142.87|0.0 Ecke2: 71.39|123.71 Ecke3: 214.23|123.77
D4: Ecke1: 142.87|0.0 Ecke2: 285.74|0.0 Ecke3: 214.27|123.71
D2: Ecke1: 285.74|0.0 Ecke2: 214.27|123.71 Ecke3: 357.1|123.77
D5: Ecke1: 285.74|0.0 Ecke2: 428.58|0.0 Ecke3: 357.16|123.74
Distanz: 142.87

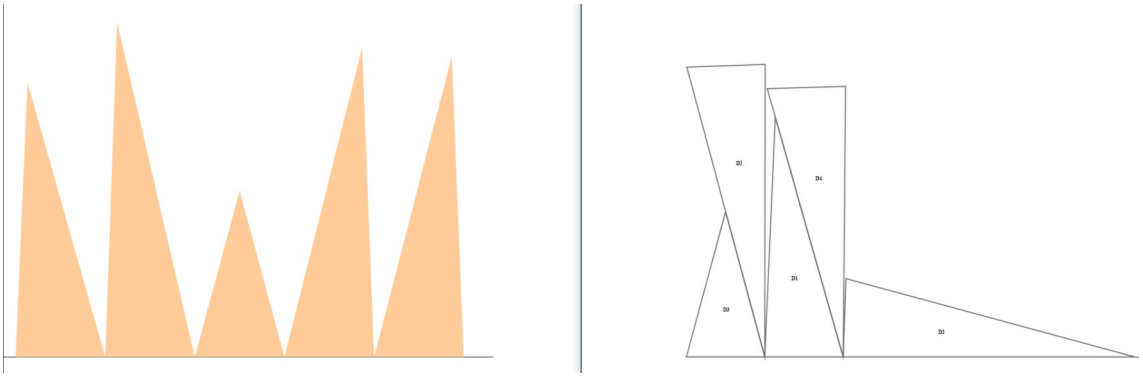




Beispiel 2:

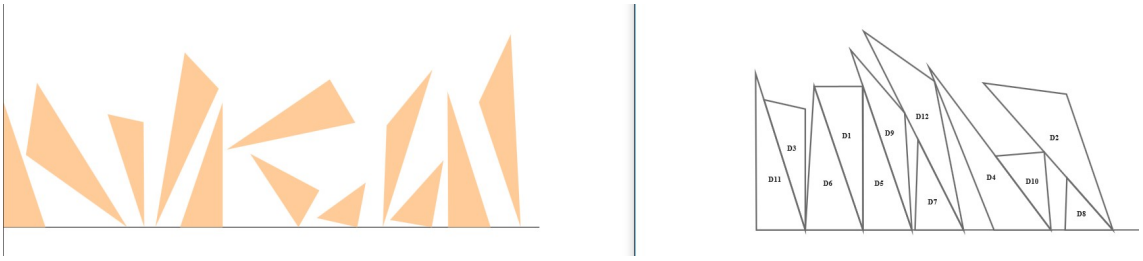
---ALGORITHMUS---
Algorithmus-Paare:
D3-ALPHA + D2-BETA
D1-GAMMA + D4-BETA
D2-BETA + D2-BETA
---ERGEBNIS---
D3: Ecke1: 0.0|0.0 Ecke2: 150.0|0.0 Ecke3: 75.0|278.0
D2: Ecke1: 150.0|0.0 Ecke2: 0.76|553.16 Ecke3: 150.67|558.35
D1: Ecke1: 150.0|0.0 Ecke2: 300.0|0.0 Ecke3: 170.0|458.0

D4: Ecke1: 300.0|0.0 Ecke2: 154.7|511.9 Ecke3: 304.63|516.36
D2: Ecke1: 300.0|0.0 Ecke2: 858.35|0.0 Ecke3: 305.37|149.9
Distanz: 150.0



Beispiel 3:

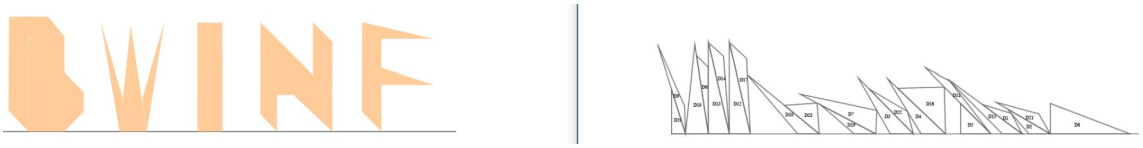
---ALGORITHMUS---
Algorithmus-Paare:
D11-GAMMA + D3-BETA
D6-GAMMA + D1-ALPHA
D5-BETA + D9-BETA
D7-BETA + D12-GAMMA
D4-BETA + D10-BETA
D8-BETA + D2-GAMMA
---ERGEBNIS---
D11: Ecke1: 0.0|0.0 Ecke2: 65.0|0.0 Ecke3: -1.0|207.0
D3: Ecke1: 65.0|0.0 Ecke2: 10.05|172.33 Ecke3: 64.97|160.0
D6: Ecke1: 65.0|0.0 Ecke2: 141.66|0.0 Ecke3: 76.73|190.46
D1: Ecke1: 141.66|0.0 Ecke2: 76.97|189.76 Ecke3: 140.97|189.99
D5: Ecke1: 141.66|0.0 Ecke2: 206.66|0.0 Ecke3: 141.66|190.0
D9: Ecke1: 206.66|0.0 Ecke2: 125.17|238.19 Ecke3: 197.1|154.82
D7: Ecke1: 210.82|0.0 Ecke2: 275.31|0.0 Ecke3: 214.78|119.81
D12: Ecke1: 275.31|0.0 Ecke2: 142.57|262.75 Ecke3: 236.69|196.73
D4: Ecke1: 315.85|0.0 Ecke2: 391.54|0.0 Ecke3: 229.16|215.44
D10: Ecke1: 391.54|0.0 Ecke2: 317.81|97.82 Ecke3: 382.53|103.18
D8: Ecke1: 410.32|0.0 Ecke2: 473.89|0.0 Ecke3: 412.62|69.19
D2: Ecke1: 473.89|0.0 Ecke2: 301.73|194.43 Ecke3: 412.06|179.72
Distanz: 408.89



Beispiel 4:

---ALGORITHMUS---
Algorithmus-Paare:
D20-BETA + D9-GAMMA
D10-ALPHA + D6-ALPHA
D13-ALPHA + D14-BETA
D12-ALPHA + D17-BETA
D18-BETA + D23-GAMMA
D19-GAMMA + D7-BETA
D3-ALPHA + D22-BETA
D4-GAMMA + D16-BETA
D5-BETA + D11-GAMMA

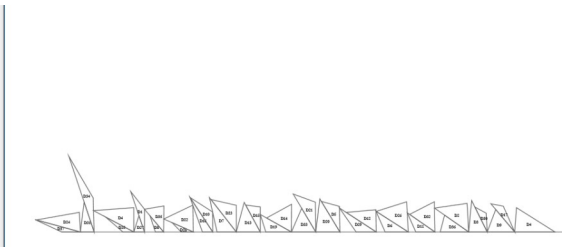
D15-ALPHA + D1-BETA
D2-BETA + D21-ALPHA
D8-ALPHA + D8-ALPHA
---ERGEBNIS---
D20: Ecke1: 0.0|0.0 Ecke2: 30.0|0.0 Ecke3: 0.0|95.0
D9: Ecke1: 30.0|0.0 Ecke2: -29.9|189.69 Ecke3: 28.05|59.27
D10: Ecke1: 30.0|0.0 Ecke2: 79.0|0.0 Ecke3: 49.99|190.0
D6: Ecke1: 79.0|0.0 Ecke2: 53.65|166.07 Ecke3: 78.48|141.42
D13: Ecke1: 79.0|0.0 Ecke2: 124.0|0.0 Ecke3: 78.99|195.0
D14: Ecke1: 124.0|0.0 Ecke2: 79.41|193.2 Ecke3: 111.65|164.53
D12: Ecke1: 124.0|0.0 Ecke2: 169.0|0.0 Ecke3: 123.99|195.0
D17: Ecke1: 169.0|0.0 Ecke2: 124.23|194.0 Ecke3: 160.98|159.79
D18: Ecke1: 270.71|0.0 Ecke2: 316.81|0.0 Ecke3: 162.68|126.03
D23: Ecke1: 316.81|0.0 Ecke2: 242.33|60.9 Ecke3: 312.15|65.83
D19: Ecke1: 391.95|0.0 Ecke2: 438.05|0.0 Ecke3: 319.82|61.82
D7: Ecke1: 438.05|0.0 Ecke2: 275.94|84.77 Ecke3: 440.34|50.15
D3: Ecke1: 469.13|0.0 Ecke2: 518.63|0.0 Ecke3: 400.54|119.5
D22: Ecke1: 518.63|0.0 Ecke2: 424.0|95.76 Ecke3: 506.04|47.87
D4: Ecke1: 535.27|0.0 Ecke2: 587.61|0.0 Ecke3: 463.47|117.6
D16: Ecke1: 587.61|0.0 Ecke2: 484.95|97.26 Ecke3: 584.91|99.96
D5: Ecke1: 619.87|0.0 Ecke2: 684.87|0.0 Ecke3: 619.87|65.0
D11: Ecke1: 684.87|0.0 Ecke2: 544.44|140.42 Ecke3: 597.01|112.47
D15: Ecke1: 709.48|0.0 Ecke2: 751.91|0.0 Ecke3: 592.1|117.37
D1: Ecke1: 751.91|0.0 Ecke2: 666.52|62.71 Ecke3: 728.68|43.7
D2: Ecke1: 766.14|0.0 Ecke2: 812.24|0.0 Ecke3: 720.59|53.14
D21: Ecke1: 812.24|0.0 Ecke2: 695.77|67.53 Ecke3: 787.52|42.88
D8: Ecke1: 812.24|0.0 Ecke2: 983.24|0.0 Ecke3: 812.24|65.0
Distanz: 782.24



Beispiel 5:

---ALGORITHMUS---
Algorithmus-Paare:
D37-GAMMA + D24-GAMMA
D31-GAMMA + D34-GAMMA
D25-BETA + D4-BETA
D27-ALPHA + D1-GAMMA
D8-GAMMA + D35-ALPHA
D28-BETA + D22-BETA
D16-ALPHA + D10-GAMMA
D7-BETA + D23-BETA
D13-ALPHA + D15-ALPHA
D19-ALPHA + D14-ALPHA
D33-BETA + D21-ALPHA
D20-ALPHA + D5-ALPHA
D29-ALPHA + D12-ALPHA
D6-GAMMA + D26-GAMMA
D11-BETA + D32-ALPHA
D36-ALPHA + D2-GAMMA
D3-BETA + D30-GAMMA
D9-ALPHA + D17-BETA
D4-ALPHA + D4-ALPHA
---ERGEBNIS---
D37: Ecke1: 0.0|0.0 Ecke2: 47.8|0.0 Ecke3: -67.94|31.67
D24: Ecke1: 47.8|0.0 Ecke2: -64.32|30.68 Ecke3: 45.2|50.02
D31: Ecke1: 47.8|0.0 Ecke2: 82.93|0.0 Ecke3: 57.36|76.46
D34: Ecke1: 82.93|0.0 Ecke2: 18.31|193.28 Ecke3: 80.5|86.78

D25: Ecke1: 156.45|0.0 Ecke2: 184.25|0.0 Ecke3: 111.6|38.44
D4: Ecke1: 184.25|0.0 Ecke2: 81.4|54.42 Ecke3: 184.07|62.12
D27: Ecke1: 184.25|0.0 Ecke2: 211.45|0.0 Ecke3: 197.26|41.61
D1: Ecke1: 211.45|0.0 Ecke2: 176.26|103.2 Ecke3: 211.46|58.05
D8: Ecke1: 239.18|0.0 Ecke2: 260.28|0.0 Ecke3: 225.63|41.09
D35: Ecke1: 260.28|0.0 Ecke2: 211.46|57.9 Ecke3: 261.01|66.47
D28: Ecke1: 305.02|0.0 Ecke2: 336.04|0.0 Ecke3: 281.94|23.72
D22: Ecke1: 336.04|0.0 Ecke2: 260.64|33.06 Ecke3: 334.68|68.39
D16: Ecke1: 359.18|0.0 Ecke2: 385.58|0.0 Ecke3: 327.71|87.87
D10: Ecke1: 385.58|0.0 Ecke2: 327.62|88.01 Ecke3: 385.23|50.99
D7: Ecke1: 396.55|0.0 Ecke2: 445.64|0.0 Ecke3: 377.07|87.75
D23: Ecke1: 445.64|0.0 Ecke2: 380.94|82.79 Ecke3: 445.7|68.11
D13: Ecke1: 445.64|0.0 Ecke2: 506.77|0.0 Ecke3: 465.74|75.47
D15: Ecke1: 506.77|0.0 Ecke2: 469.47|68.62 Ecke3: 506.82|63.95
D19: Ecke1: 520.91|0.0 Ecke2: 586.34|0.0 Ecke3: 506.8|44.69
D14: Ecke1: 586.34|0.0 Ecke2: 522.88|35.66 Ecke3: 586.69|70.7
D33: Ecke1: 586.34|0.0 Ecke2: 648.33|0.0 Ecke3: 612.46|60.32
D21: Ecke1: 648.33|0.0 Ecke2: 590.78|96.79 Ecke3: 647.39|76.02
D20: Ecke1: 648.33|0.0 Ecke2: 709.99|0.0 Ecke3: 658.76|84.38
D5: Ecke1: 709.99|0.0 Ecke2: 662.62|78.01 Ecke3: 708.75|65.28
D29: Ecke1: 751.37|0.0 Ecke2: 802.69|0.0 Ecke3: 708.83|61.11
D12: Ecke1: 802.69|0.0 Ecke2: 726.06|49.89 Ecke3: 801.82|56.64
D6: Ecke1: 825.42|0.0 Ecke2: 884.5|0.0 Ecke3: 801.88|52.86
D26: Ecke1: 884.5|0.0 Ecke2: 802.53|52.44 Ecke3: 880.15|77.0
D11: Ecke1: 903.66|0.0 Ecke2: 952.49|0.0 Ecke3: 881.73|49.08
D32: Ecke1: 952.49|0.0 Ecke2: 887.87|44.82 Ecke3: 950.95|77.14
D36: Ecke1: 965.82|0.0 Ecke2: 1038.51|0.0 Ecke3: 977.02|42.94
D2: Ecke1: 1038.51|0.0 Ecke2: 951.28|60.92 Ecke3: 1034.51|72.95
D3: Ecke1: 1038.51|0.0 Ecke2: 1085.78|0.0 Ecke3: 1046.06|80.24
D30: Ecke1: 1085.78|0.0 Ecke2: 1054.26|63.69 Ecke3: 1083.42|47.74
D9: Ecke1: 1085.78|0.0 Ecke2: 1155.95|0.0 Ecke3: 1107.55|57.29
D17: Ecke1: 1155.95|0.0 Ecke2: 1095.07|72.06 Ecke3: 1126.9|65.73
D4: Ecke1: 1155.95|0.0 Ecke2: 1258.91|0.0 Ecke3: 1160.42|61.96
Distanz: 1108.15



Quellcode

```
public class Dreiecke {
    public static List<Triangle> triangles = new ArrayList<>();
    public static List<Triangle.AnglePair> algorithmPairs = new
ArrayList<>();
    public static List<TrianglePath> finalTriangles = new
ArrayList<>();

    public static void main(String[] args) {
        System.out.println("---INFOS---");
        // read information from file
        readFileData(coordsFileName);
        for (Triangle triangle : triangles) {
            System.out.println(triangle.toString());
        }
        // execute algorithm
```

```
        System.out.println("---ALGORITHMUS---");
        executeAlgorithm();
        System.out.println("Algorithmus-Paare:");
        for (Triangle.AnglePair algorithmPair : algorithmPairs) {
            System.out.println(
                    algorithmPair.getAngle1().toString() + " + " +
algorithmPair.getAngle2().toString());
        }
        SVGGenerator.generateSVG(coordsFileName.substring(0,
coordsFileName.length() - 4) + ".svg");
        // print out result
        System.out.println("---ERGEBNIS---");
        for (TrianglePath finalTriangle : finalTriangles) {
            System.out.println(finalTriangle.toString());
        }
        double firstX = 0;
        double lastX = Double.POSITIVE_INFINITY;
        for (Point2D corner : finalTriangles.get(0).getCorners())
{
            if (corner.getY() == 0 && corner.getX() > firstX) {
                firstX = corner.getX();
            }
        }
        for (Point2D corner :
finalTriangles.get(finalTriangles.size() - 1).getCorners()) {
            if (corner.getY() == 0 && corner.getX() < lastX) {
                lastX = corner.getX();
            }
        }
        double distance = lastX - firstX;
        System.out.println("Distanz: " + Utils.round(distance,
2));
    }

    private static void executeAlgorithm() {
        // sort by area asc
        triangles.sort(Comparator.comparing(Triangle::getArea));

        // create pairs of almost 90 degree (only angles at
longest side of a triangle) to form rectangles
        List<Triangle.AnglePair> anglePairs = new ArrayList<>();
        for (Triangle triangle : triangles) {
            anglePairs.add(triangle.getTwoSmallestAngles());
        }

        List<Angle> angles1 = new ArrayList<>();
        List<Angle> angles2 = new ArrayList<>();
        for (Triangle.AnglePair anglePair : anglePairs) {
            angles1.add(anglePair.getAngle1());
            angles2.add(anglePair.getAngle2());
        }


angles1.sort(Comparator.comparingDouble(Angle::getDegrees));

angles2.sort(Comparator.comparingDouble(Angle::getDegrees));

        int n = triangles.size();
        for (int i = 0; i < n / 2; i++) {
            Angle start1 = angles1.get(0);
            Angle start2 = getPairAngle(start1, angles2);
```

```
            // get best angle to create a sum near to 90° for
first angle of current AnglePair
            Angle partner1 = bestPair(start1, angles1, angles2);
            // get best angle to create a sum near to 90° for
second angle of current AnglePair
            Angle partner2 = bestPair(start2, angles1, angles2);
            // get best angle to create sum near to 90° from both
angles of current AnglePair
            Angle partner = nearerTo90(start1, partner1, start2,
partner2);
            boolean second = partner == partner2 ? true : false;
            Angle start = second ? start2 : start1;

            // save AnglePair - triangle with shorter angle
adjacent side first
            if
(start.getTriangle().getSideFromType(Triangle.getShortAdjacentSide
(start)) <

partner.getTriangle().getSideFromType(Triangle.getShortAdjacentSid
e(partner))) {
                algorithmPairs.add(new
Triangle.AnglePair(start.getTriangle(), partner.getTriangle(),
                        start.getType(), partner.getType()));
            } else {
                algorithmPairs.add(new
Triangle.AnglePair(partner.getTriangle(), start.getTriangle(),
                        partner.getType(), start.getType()));
            }

            // remove angles in this pair from available angles
            for (int p = 0; p < angles1.size(); p++) {
                if (sameTriangle(angles1.get(p), start)) {
                    angles1.remove(p);
                } else if (sameTriangle(angles1.get(p), partner))
{
                    angles1.remove(p);
                }
                if (sameTriangle(angles2.get(p), start)) {
                    angles2.remove(p);
                } else if (sameTriangle(angles2.get(p), partner))
{
                    angles2.remove(p);
                }
            }
        }

        // check if there's an odd number of triangles -> put
biggest at the end
        if (n % 2 != 0) {
            Triangle spareTriangle =
triangles.get(triangles.size() - 1);
            algorithmPairs.add(new
Triangle.AnglePair(spareTriangle, spareTriangle,

spareTriangle.getTwoSmallestAngles().getAngle1().getType(),

spareTriangle.getTwoSmallestAngles().getAngle1().getType()));
        }
    }
```

```
    private static Angle getPairAngle(Angle first, List<Angle>
angles) {
        for (Angle angle : angles) {
            if (sameTriangle(first, angle)) {
                return angle;
            }
        }
        return null;
    }

    private static boolean sameTriangle(Angle angle1, Angle
angle2) {
        return
angle1.getTriangle().getName().equals(angle2.getTriangle().getName
());
    }

    private static Angle bestPair(Angle angleStart, List<Angle>
angles1, List<Angle> angles2) {
        Angle partner1 = null;
        Angle partner2 = null;
        // create sums with all AnglePairs following in the array
        for (int j = 0; j < angles1.size(); j++) {
            Angle check1 = angles1.get(j);
            Angle check2 = angles2.get(j);

            if (check1 != angleStart && !sameTriangle(check1,
angleStart)) {
                // create sum of first angle of the AnglePair
                partner1 = partner1 == null ? check1 :
nearerTo90(angleStart, partner1, check1);
            }

            if (check2 != angleStart && !sameTriangle(check2,
angleStart)) {
                // create sum of second angle of the AnglePair
                partner2 = partner2 == null ? check2 :
nearerTo90(angleStart, partner2, check2);
            }
        }
        return nearerTo90(angleStart, partner1, partner2);
    }

    private static Angle nearerTo90(Angle start, Angle check1,
Angle check2) {
        double diff1 = Math.abs(90 - start.getDegrees() -
check1.getDegrees());
        double diff2 = Math.abs(90 - start.getDegrees() -
check2.getDegrees());
        if (diff1 < diff2) {
            return check1;
        } else {
            return check2;
        }
    }

    private static Angle nearerTo90(Angle start1, Angle partner1,
Angle start2, Angle partner2) {
        double diff1 = Math.abs(90 - start1.getDegrees() -
partner1.getDegrees());
```

```
        double diff2 = Math.abs(90 - start2.getDegrees() -
partner2.getDegrees());
        if (diff1 < diff2) {
            return partner1;
        } else {
            return partner2;
        }
    }
}

public class SVGGenerator {
    private static void paint(SVGGraphics2D graphic) {
        // draw TrianglePairs
        double xShift = 0;
        Point2D safePoint = null;
        for (Triangle.AnglePair pair : Dreiecke.algorithmPairs) {
            Triangle triangle1 = pair.getAngle1().getTriangle();
            Triangle triangle2 = pair.getAngle2().getTriangle();
            Angle angle1 = pair.getAngle1();
            Angle angle2 = pair.getAngle2();

            // calculate side values for first triangle
            double shortSideLength1 =
triangle1.getSideFromType(Triangle.getShortAdjacentSide(angle1));
            double longSideLength1 =
triangle1.getSideFromType(triangle1.getLongestSide());
            double angleValue1 = angle1.getDegrees();
            double xOff1 = Math.cos(Math.toRadians(angleValue1)) *
longSideLength1;
            double yOff1 = Math.sin(Math.toRadians(angleValue1)) *
longSideLength1;

            // calculate side values for second triangle
            double longSideLength2 =
triangle2.getSideFromType(triangle2.getLongestSide());
            double xOff2 = Math.cos(Math.toRadians(angleValue1)) *
longSideLength2;
            double yOff2 = Math.sin(Math.toRadians(angleValue1)) *
longSideLength2;
            double shortSideLength2 =
triangle2.getSideFromType(Triangle.getShortAdjacentSide(angle2));
            double angleValue2 = 180 - angleValue1 -
angle2.getDegrees();
            double xOff2_1 = Math.cos(Math.toRadians(angleValue2))
* shortSideLength2;
            double yOff2_1 = Math.sin(Math.toRadians(angleValue2))
* shortSideLength2;

            if (safePoint != null) {
                double newStart = -1;
                // move the pair a bit further right if the first
triangle would intersect with the previous pair
                Point2D absoluteSafePoint;
                if (safePoint.getY() > yOff1) {
                    absoluteSafePoint = new Point2D.Double(xShift
+ safePoint.getX(), yOff1);
                } else {
                    absoluteSafePoint = new Point2D.Double(xShift
+ safePoint.getX(), safePoint.getY());
                }
```

```
                double safeStart =
    calcXWantCutY(absoluteSafePoint.getY(), xShift, safePoint.getY() /
    safePoint.getX(),
                        yOff1 / (shortSideLength1 - xOff1));
                if (safeStart > xShift) {
                    newStart = safeStart;
                }
                // move the pair a bit further right if the second
    triangle would intersect with the previous pair
                if (safePoint.getY() > yOff2) {
                    absoluteSafePoint = new Point2D.Double(xShift
    + safePoint.getX(), yOff2);
                } else {
                    absoluteSafePoint = new Point2D.Double(xShift
    + safePoint.getX(), safePoint.getY());
                }
                safeStart =
    calcXWantCutY(absoluteSafePoint.getY(), xShift, safePoint.getY() /
    safePoint.getX(),
                        yOff2 / -xOff2) - shortSideLength1;
                if (safeStart > xShift && safeStart > newStart) {
                    newStart = safeStart;
                }

                if (newStart > 0) {
                    xShift = newStart;
                }
            }

            Path2D path1 = new Path2D.Double();
            // first triangles starting point
            path1.moveTo(xShift, 0);
            // first triangles shortest side
            path1.lineTo(xShift + shortSideLength1, 0);
            xShift += shortSideLength1;
            // first triangles last point (longest side)
            path1.lineTo(xShift - xOff1, yOff1);
            path1.closePath();
            graphic.draw(path1);
            TrianglePath trianglePath1 = new
    TrianglePath(triangle1.getName(), path1);
            Dreiecke.finalTriangles.add(trianglePath1);
            // draw triangle1 name
            graphic.drawString(trianglePath1.getName(), (int)
    trianglePath1.getCenter().getX(),
                    (int) trianglePath1.getCenter().getY());

            // check whether last pair is the spare triangle
            if (triangle2 == triangle1) {
                break;
            }

            Path2D path2 = new Path2D.Double();
            // second triangles bottom point
            path2.moveTo(xShift, 0);
            // second triangles side, that's aligning with the
    hypotenuse of the first one
            path2.lineTo(xShift - xOff2, yOff2);
            // second triangles short angle side
            path2.lineTo(xShift + xOff2_1, yOff2_1);
            path2.closePath();
```

```
            graphic.draw(path2);
            TrianglePath trianglePath2 = new
TrianglePath(triangle2.getName(), path2);
            Dreiecke.finalTriangles.add(trianglePath2);

            safePoint = new Point2D.Double(xOff2_1, yOff2_1);
            // draw triangle2 name
            graphic.drawString(trianglePath2.getName(), (int)
trianglePath2.getCenter().getX(),
                    (int) trianglePath2.getCenter().getY());
        }
    }

    private static double calcXWantCutY(double cutY, double
xZero1, double m1, double m2) {
        // safe function: f(x)=m1*x+n1
        // calculate n1: 0=m1*xZero1+n1 -> n1=-m1*xZero1
        double n1 = -m1 * xZero1;
        // calc cutX: f(x)=cutY -> cutY=m1*cutX+n1 -> cutX=(cutY-
n1)/m1
        double cutX = (cutY - n1) / m1;

        // new function: g(x)=m2*x+n2
        // calc n2 for which func runs through the cut point:
cutY=m2*cutX+n2 -> n2=cutY-m2*cutX
        double n2 = cutY - m2 * cutX;

        // calculate at which point on the x axis the line (g(x))
has to start -> calculating zero point g(x)=0
        // 0=m2*xZero+n2 -> x=-n2/m2
        double xZero = -n2 / m2;
        return xZero;
    }
}
```