Aufgabe 2: Dreiecksbeziehungen

Teilnahme-ID: 48302

Bearbeiter dieser Aufgabe:
Florian Rädiker (15 Jahre)

14. April 2019

Inhalt

1. Lösungsidee

Die Grundidee des Algorithmus ist, Dreiecke in Gruppen zusammenzufassen und diese nebeneinander zu platzieren. Die Dreiecke in einer Gruppe berühren sich alle in einem Punkt, dem *Ursprung* der Gruppe. Deshalb darf die Summe der ursprungsberührenden Winkel der Dreiecke einer Gruppe 180° nicht überschreiten. Es ist also offensichtlich sinnvoll, dass die Dreiecke mit ihrem kleinsten Winkel den Ursprung berühren, sodass möglichst viele Dreiecke in eine Gruppe passen.
Ist die Summe der kleinsten Winkel aller Dreiecke, die angeordnet werden sollen, kleiner oder gleich 180°, dann können sie immer in einer Gruppe mit einem Gesamtabstand von 0 Metern angeordnet werden. Für mehr als 180° sind zwingend zwei oder mehr Gruppen erforderlich (Abb. 1). Der Abstand zwischen dem Ursprung der Gruppe ganz links und dem Ursprung der Gruppe ganz rechts ist der Gesamtabstand. In den Abbildungen dieser Dokumentation haben die Dreiecke einer Gruppe immer den gleichen Farbton.

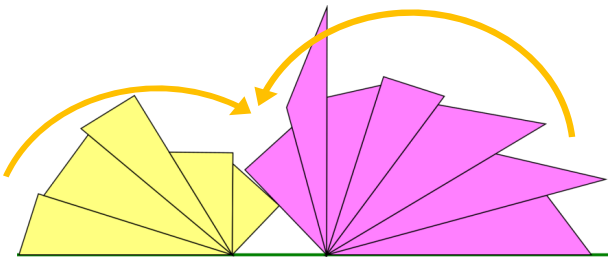


Abb. 1: Beispielhafte Anordnung von Dreiecken mit einer Summe der kleinsten Winkel über 180° in zwei Gruppen

Um einen möglichst geringen Gesamtabstand zu erhalten, muss es nicht immer sinnvoll sein, dass alle Dreiecke nur mit ihrem kleinsten Winkel die Ursprünge der Gruppen berühren. Daher kann pro Gruppe maximal ein Dreieck, dessen kleinste Seite besonders klein ist, oder das sich aufgrund anderer Beschaffenheiten besonders gut dazu eignet, mit dessen kürzester Seite an der x-Achse platziert werden. Da sich der kleinste Winkel eines Dreiecks immer der kleinsten Seite gegenüber befindet, berührt dann nicht der kleinste, sondern einer der zwei anderen Winkel den Ursprung (Abb. 2). Solche Dreiecke, die in einer Gruppe absichtlich mit einer Kante die x-Achse berühren, heißen *Basisdreiecke*. Sie sind in allen Abbildungen durch stärkere Farben hervorgehoben.

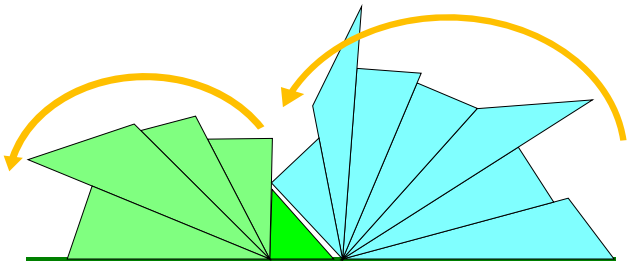


Abb. 2: Wie in Abb. 1 gibt es zwei Gruppen, die linke Gruppe hat ein Basisdreieck

Gruppen können entweder gegen oder im Uhrzeigersinn aufgebaut werden. Die Pfeile in Abb. 1 und Abb. 2 geben die Richtung für die jeweiligen Gruppen an. Wenn eine Gruppe im Uhrzeigersinn aufgebaut wird, liegt ihr erstes Dreieck links vom Ursprung auf der x-Achse, ansonsten auf der rechten Seite. „Erstes Dreieck“ steht bei einer Gruppe mit Basisdreieck immer für das Basisdreieck selbst (z.B. die grüne Gruppe in Abb. 2 ist gegen den Uhrzeigersinn aufgebaut). Der *Winkel einer Gruppe* ist definiert durch die Summe aller Winkel, die den Ursprung berühren; dieser beträgt also maximal 180°.

Es gilt zu beachten, dass laut Aufgabenstellung nur die „Größe und Winkel“ der Dreiecke entscheidend sind, also können sie auch beliebig gespiegelt werden.

1.1 Der Algorithmus

Hier wird die Struktur des Algorithmus erläutert. Die Angaben in Klammern beziehen sich jeweils auf den Abschnitt, in dem dieser Schritt näher erläutert wird.

1. Prüfe, ob die Summe aller kleinsten Winkel der Dreiecke kleiner als 360° ist.
   Wenn ja, ordne die Dreiecke so wie in Abb. 1 in zwei Gruppen an und speichere die resultierende Gesamtlänge und die entstandenen Gruppen als bisher bestes Ergebnis (1.2).
2. Suche die Auswahl an Basisdreiecken, bei der die Summe der kleinsten Seiten der Basisdreiecke möglichst klein ist und gleichzeitig der durch die Basisdreiecke zur Verfügung stehende Platz (Winkel) für die übrigen Dreiecke höchstwahrscheinlich ausreicht (z.B. bei einer Gesamtsumme der kleinsten Winkel von 1000° wird ein Basisdreieck nicht ausreichen) (1.3).
3. Ordne die Auswahl an Basisdreiecken mit den übrigen Dreiecken in Gruppen an. Es gibt eine Gruppe mehr als es Basisdreiecke gibt (siehe auch Abb. 2) (1.4).
4. Sortiere die Dreiecke innerhalb der Gruppen und platziere die Gruppen grafisch, um eine SVG-Grafik und den Gesamtabstand zu erhalten (1.5).
5. Wenn der Gesamtabstand kleiner als das bisher beste Ergebnis ist, speichere den neuen Gesamtabstand und die neuen Gruppen als bisher bestes Ergebnis.
6. Suche die nächstlängere Auswahl an Basisdreiecken (1.3, entspricht Schritt 2).
   Sollte die neue Summe der kleinsten Seiten der Basisdreiecke allein schon größer als der bisher beste Gesamtabstand sein, ist das bisher beste Ergebnis nicht weiter optimierbar, ansonsten kehre zu Schritt 3 zurück.

1.2 Platzierung der Dreiecke für Summe der kleinsten Winkel bis 360°

Dreiecke, bei denen die Summe der kleinsten Winkel kleiner oder gleich 360° ist, können in zwei Gruppen mit möglichst gleich großem Winkel aufgeteilt werden. Das zugehörige Problem, das sich damit beschäftigt, eine Untermenge zu finden, deren Summe möglichst nah an einem vorgegebenen Wert liegt ohne diesen zu überschreiten, nennt sich Teilsummenproblem (engl. „subset sum problem“) und ist NP-vollständig. Für eine akzeptable Lösung für dieses Problem siehe Abschnitt 1.6. Die linke Gruppe wird dann im Uhrzeigersinn und die rechte gegen den Uhrzeigersinn platziert (Abb. 3).

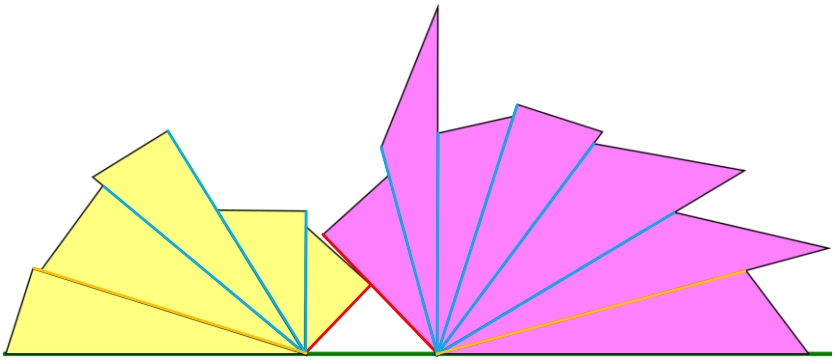


Abb. 3: Platzierung der Dreiecke in zwei Gruppen, die mittellangen Seiten sind jeweils eingefärbt

Die Sortierung der Dreiecke innerhalb der Gruppen wird durch die in Abschnitt 1.5 beschriebene Methode vorgenommen, die auch für Dreiecke mit einer Winkelsumme größer als 360° benutzt wird und daher entsprechend komplexer ist. Folgendes ist für diesen Teil relevant:
Die Dreiecke werden innerhalb der Gruppen nach der Länge der mittellangen Seite absteigend sortiert, sodass das Dreieck mit der längsten mittellangen Seite (in Abb. 3 orange) in beiden Gruppen zuerst an der x-Achse platziert wird und das Dreieck mit der kürzesten Seite (in Abb. 3 rot) jeweils zuletzt innerhalb der Gruppe platziert wird. Außerdem zeigt die mittellange Seite der Dreiecke jeweils in die Richtung der anderen Gruppe, sodass sich möglichst die kleineren Seiten berühren.

1.3 Suchen guter Auswahlen für die Basisdreiecke

Es werden verschiedene Auswahlen für die Basisdreiecke gesucht, die dann an den nächsten Schritt (Abschnitt 1.4) weitergegeben werden. Für die Suche nach Dreiecken mit besonders guten Eigenschaften werden drei Kriterien berücksichtigt:

1. möglichst kleine Seite an der x-Achse (die kleinste Seite soll möglichst klein sein),
2. möglichst kleine Dreiecke insgesamt (die längste Seite des Basisdreiecks soll möglichst klein sein) und
3. möglichst viel Platz für die übrigen Dreiecke (also die beiden Winkel an der x-Achse sollen möglichst klein sein, sodass links und rechts viel Platz für die übrigen Dreiecke entsteht; gleichbedeutend mit einem möglichst großen kleinsten Winkel)

Die von den Auswahlen benötigte Länge auf der x-Achse werden immer größer. Die Länge einer Auswahl auf der x-Achse ist die Summe aller kleinsten Seiten der Dreiecke in der Auswahl. Mit dieser Länge gibt es ein gutes Abbruchkriterium in Schritt 6 des Algorithmus, weil diese Länge das Minimum für einen Gesamtabstand angibt (die Summe der kleinsten Seiten der Basisdreiecke wird auf der x-Achse mindestens benötigt).

Die Suche nach Auswahlen läuft folgendermaßen ab:

1. Ordne jedem Kriterium das Dreieck zu, was es am besten erfüllt.
2. Wähle das Kriterium, dessen bisher zugeordnete Dreiecke die kleinste Summe aller kleinsten Seiten haben.
3. Prüfe, ob die Dreiecke des ausgewählten Kriteriums überhaupt genug Platz für die übrigen Dreiecke bieten würden (bei einer sehr großen Summe der kleinsten Winkel wird ein Basisdreieck z.B. wahrscheinlich nicht ausreichen).
4. Wenn die Dreiecke genug Platz bieten, gib sie als Basisdreiecke zum nächsten Schritt weiter.
5. Ordne dem in Schritt 2 ausgewählten Kriterium zusätzlich das für dieses Kriterium nächstbeste Dreieck zu.
6. Kehre zurück zu Schritt 2 (außer natürlich wenn die Suche vom Algorithmus abgebrochen wurde, weil die Abbruchbedingung erfüllt ist).

Die Suche nach guten Auswahlen für die Basisdreiecke wird an folgendem Beispiel verdeutlicht; es handelt sich um die Dreiecke der Beispieldaten Nummer 3:

ID	kürzeste Seite	längste Seite	kleinster Winkel	Kriterien		
				kürzeste Seite (1)	längste Seite (2)	kleinster Winkel (3)
1	64,0	200,5	18,6°	3.		
2	111,3	259,7	22,5°			
3	56,3	180,9	17,7°	1.		
4	75,7	269,8	15,1°			
5	65,0	200,8	18,9°			
6	76,7	201,2	22,3°			
7	64,5	134,2	28,7°		3.	3.
8	63,6	92,4	43,4°	2.	1.	1.
9	110,1	251,7	15,4°			
10	64,9	122,5	32,0°		2.	2.
11	65,0	217,3	17,4°			
12	115,0	294,4	15,7°			
			Σ = 267,8°			

Die Zahlen in den drei letzten Spalten stehen jeweils für die ersten drei Plätze, die die Dreiecke durch die Bewertung des Kriteriums einnimmt. „1.“ steht für das für dieses Kriterium beste Dreieck, „2.“ für das zweitbeste Dreieck und so weiter.

Die Schritte des Algorithmus werden folgendermaßen durchlaufen:

1. Den Kriterien das jeweils beste Dreieck zuordnen: Kriterium 1 → D3, Kriterium 2 → D8, Kriterium 3 → D8 (siehe Tabelle)
2. Da Kriterium 1 immer nach Länge der kürzesten Seite beurteilt, wird es im ersten Durchlauf immer ausgewählt
3. Zu Kriterium 1 gehört derzeit Dreieck 3 und für die anderen Dreiecke verbleibt damit eine Summe der kleinsten Winkel von 267,8° – 17,7° = 250,1°. Die beiden Winkel von D3, die später an der x-Achse anliegen, sind zusammen 180° – 17,7° = 162,3° groß. Für die übrigen Dreiecke entsteht also ein Platz von 360° – 162,3° = 197,7° (180° links und rechts vom Dreieck ergeben 360°). Dieser Platz reicht nicht aus, um die übrigen Dreiecke zu einzufügen (197,7° < 250,1°). Es gibt aber Fälle, in denen der Platz nur knapp nicht ausreicht und es deshalb sinnvoll sein kann, die verbleibenden Dreiecke in die bereits bestehenden Gruppen zu ergänzen, sofern diese Gruppen noch genug Platz haben. Die erste Gruppe ist meist schon bis zur x-Achse gefüllt, aber die anderen Gruppen haben oft noch genug Platz für mehr Dreiecke. Ein solcher Fall ist in Abb. 4 dargestellt.

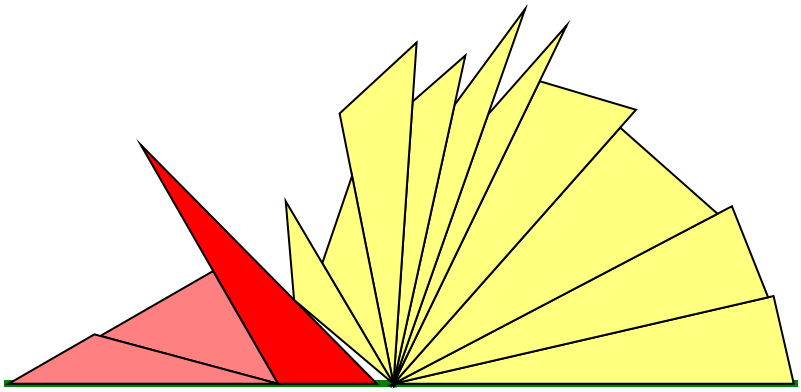


Abb. 4: Die zweite Gruppe nimmt ein Dreieck mehr auf als vorgesehen, denn der Winkel bis zur Kante des roten Basisdreiecks wurde überschritten

Wie zu sehen ist, wurden in der zweiten Gruppe schon mehr Dreiecke als vorgesehen untergebracht; in der ersten Gruppe ist dies nicht möglich. Das Hinzufügen von Dreiecken ist unter Umständen sogar sehr gut für den Gesamtabstand. Wenn im Beispiel eine neue Gruppe rechts angefügt werden würde, um das einzelne Dreieck noch unterzubringen, wäre der Gesamtabstand schon deutlich größer. Um solche Fälle zu berücksichtigen, wird der zur Verfügung stehende Platz vor dem Vergleichen immer mit einem Toleranzfaktor multipliziert. Dieser Faktor wurde willkürlich auf 1,2 festgelegt. In diesem Beispiel ergeben sich also 197,7° × 1,2 = 237,24°, was aber immer noch nicht mehr als 250,1° ist.

4. Die Auswahl bestehend aus D3 hat nicht genug Platz geboten und wird deshalb nicht weitergegeben.
5. Kriterium 1 wird nun ein zusätzliches Dreieck, nämlich D8, zugeordnet und es ergibt sich als Länge der kürzesten Seiten 56,3 + 63,6 = 119,9.

2. Sowohl Kriterium 2 als auch Kriterium 3 ist D8 zugeordnet, das mit einer Länge von 63,6 kürzer als die Dreiecke von Kriterium 1 ist. Immer, wenn zwei Kriterien dieselbe Länge haben, wird das erste ausgewählt, also Kriterium 2.
3. Es verbleiben 267,8° – 43,4° = 224,4° und durch die Auswahl von D8 kann ein Winkel von 360° – (180° – 43,4°) = 223,4° zur Verfügung gestellt werden (also Wert zum Vergleich 223,4° × 1,2 = 268,08°).
4. Da die verbleibende Summe der kleinsten Winkel mit 224,4° kleiner ist als der zur Verfügung gestellte Winkel, wird die Auswahl für die Basisdreiecke {D8} an den nächsten Schritt weitergegeben.
5. Kriterium 2 sind nun die Dreiecke {D8, D10} zugeordnet (Länge 63,6 + 64,9 = 101,5).

2. Es würde nochmals D8 mit Kriterium 3 ausgewählt werden. Um dies aber zu verhindern, wird geprüft, ob die Länge der neuen Auswahl tatsächlich größer und nicht gleich der letzten Auswahl ist. Hier ist sie gleich, also ist auch die Auswahl der Basisdreiecke dieselbe. Die nächsten zwei Schritte werden übersprungen.
5. Kriterium 3 wird zusätzlich das nächstbeste Dreieck D10 zugeordnet.

2. Länge für Kriterium 1: 119,9, Länge für Kriterium 2 und 3: 101,5. Es wird Kriterium 2 ausgewählt.
3. …

Wie dieser Algorithmus zeigt, werden nicht alle möglichen Auswahlen für Basisdreiecke berücksichtigt. Wenn es nur ein Kriterium gäbe, dann könnte z.B. das drittbeste Dreieck immer nur mit dem erst- und zweitbesten Dreieck zusammen in einer Auswahl sein. Um dennoch immer mehrere Möglichkeiten für 1, 2, 3, … Basisdreiecke zu bieten, gibt es die drei Kriterien.

Es gibt noch einen Zusatz: Es kann manchmal vorkommen, dass die letzte Gruppe (ganz rechts) ohne Basisdreieck weggelassen wird, weil alle übrigen Dreiecke schon in die Gruppen davor hineingepasst haben (siehe die folgenden Abschnitte). Deshalb ist es zur Berechnung der Länge eigentlich falsch, die kürzesten Seiten aller Dreiecke für ein Kriterium zu summieren. Die Reihenfolge der Basisdreiecke wird von diesem Teilalgorithmus aber noch nicht bestimmt und deshalb ist auch noch nicht bekannt, welches Basisdreieck als letztes platziert werden wird. Aus diesem Grund wird immer das zuletzt zum Kriterium hinzugekommene Dreieck zur Berechnung der Länge nicht hinzugezählt. Dies gibt einen ungefähren Anhaltspunkt und es werden mehr Auswahlen für die Basisdreiecke ausprobiert, weil das Abbruchkriterium länger unerfüllt bleibt. Man sollte aber beachten, dass dieser Fall sehr selten ist (hier nützt er nur bei Beispieldaten 5 etwas), und vor allem bei einer großen Anzahl von Dreiecken am besten nicht berücksichtigt wird. Im obigen Beispiel beträgt die Länge also immer 0, wenn nur ein Dreieck ausgewählt wurde, und sobald für die Kriterien 1 und 2 ein weiteres Dreieck hinzukommt, betragen die Längen nicht 56,3 + 63,6 = 119,9 und 63,6 + 64,9 = 101,5, sondern 56,3 und 63,6.

1.4 Anordnung der Dreiecke in Gruppen

Für jede Auswahl von Basisdreiecken wird dieser Schritt vorgenommen, um mit den vorgegebenen Basisdreiecken Gruppen zu erstellen. Alle Gruppen sind dabei gegen den Uhrzeigersinn aufgebaut und jede außer der letzten Gruppe hat ein Basisdreieck (Abb. 5).

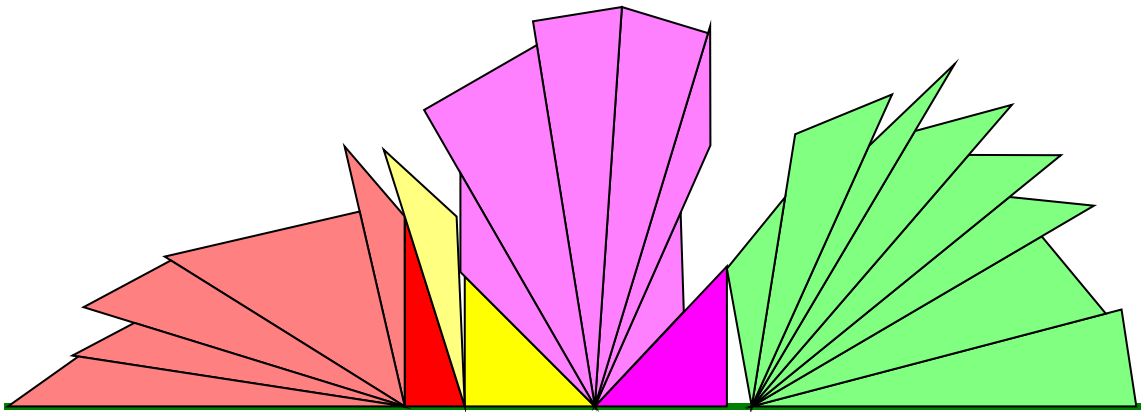


Abb. 5: Die hellen Dreiecke wurden zu den Basisdreiecken hinzugefügt

Von links nach rechts werden die Gruppen erstellt. Immer, wenn eine neue Gruppe an der rechten Seite angefügt werden soll, werden alle noch verbleibenden Basisdreiecke durchgegangen. Für ein Basisdreieck gibt es immer zwei Möglichkeiten für die Platzierung: Es kann mit dem größten bzw. mittelgroßen Winkel entweder links oder rechts platziert werden. Deshalb wird jedes Basisdreieck mit der einen und mit der anderen Platzierungsmethode ausprobiert. Für jedes dieser Basisdreiecke mit Platzierungsmethode wird versucht, Dreiecke mithilfe der Lösung zum Teilsummenproblem (Abschnitt 1.6) zu finden, die den Platz zum zuletzt platzierten Basisdreieck möglichst gut auffüllen. Diese gefundenen Dreiecke lassen unterschiedlich viel Platz übrig und es wird das Basisdreieck mit Platzierungsmethode und platzauffüllenden Dreiecken als neue Gruppe angefügt, bei dem die auffüllenden Dreiecke am wenigsten Platz freilassen. Es kommt häufig vor, dass sich der übriggelassene Platz nur um weniger als 1° unterscheidet. Würde man aber mehrere Möglichkeiten

für die neue Gruppe ausprobieren, dann würde dies auch entsprechend länger dauern, denn es würde eine Art Baumdiagramm entstehen, bei dem für jede neue Gruppe mehrere Möglichkeiten hinzukommen.
Um trotzdem verschiedene Anordnungen zu erhalten, werden für die erste Gruppe – denn diese hat ja noch keine vorherige Gruppe mit Basisdreieck – alle Basisdreiecke mit beiden Platzierungsmethoden ausprobiert. Am Ende gibt es also doppelt so viele Möglichkeiten wie es Basisdreiecke gibt, aus denen dann diejenige mit dem besten Gesamtabstand gewählt wird.

Sobald es keine Basisdreiecke mehr gibt, sind höchstwahrscheinlich noch „normale“ Dreiecke übrig. Deshalb wird eine Gruppe ganz rechts zum Schluss angefügt, die auch wieder mit der Lösung des Teilsummenproblems möglichst passende Dreiecke enthält (in Abb. 5 die grüne Gruppe). Wenn nun aber immer noch Dreiecke übrig sind, müssen diese zu bereits bestehenden Gruppen hinzugefügt werden, wie dies z.B. in Abb. 5 in der grünen Gruppe der Fall ist (dass sich mehr Dreiecke als vorgesehen in einer Gruppe befinden, kommt im vorherigen Schritt nie vor, denn die Lösung des Teilsummenproblems liefert nie Dreiecke, die den zur Verfügung stehenden Platz überschreiten).

Es gibt verschiedene Methoden, um die Gruppen auszuwählen, in die die übrigen Dreiecke eingefügt werden. Die Dreiecke können beispielsweise zu den Gruppen, die noch am meisten Platz übrighaben, hinzugefügt werden. Das ist aber nicht besonders effektiv, da die Gruppen meistens nur einen Platz von weniger als 1° zur Verfügung haben. Dreiecke, bei denen der kleinste Winkel kleiner als 1° ist, sind eher unwahrscheinlich, denn das hieße, dass das Grundstück recht schmal und kaum bewohnbar wäre. Obwohl man auch mit dieser Methode gute Ergebnisse erzielen kann, ist der verfügbare Platz also kein ausschlaggebendes Kriterium.
Eine bessere Möglichkeit besteht darin, Dreiecke in die Gruppen einzufügen, deren Gruppenwinkel möglichst nah an 90° liegt (und in die das Dreieck noch passt, sodass der maximale Winkel von 180° nicht überschritten wird). Wenn eine Gruppe einen Winkel von 90° hat, ist davon auszugehen, dass das Basisdreieck der Gruppe davor (links) auf der rechten Seite auch ungefähr einen Winkel von 90° hat. Wenn eine Gruppe stattdessen einen Winkel von 160° hätte, hat das Basisdreieck der vorherigen Gruppe auf der rechten Seite vermutlich einen Winkel von etwa 20°. Für ein einzufügendes Dreieck ist nun entscheidend, dass es in einer Gruppe mit einem sehr großen oder sehr kleinen Winkel den Abstand zur vorherigen Gruppe wahrscheinlich mehr verschiebt als bei einer Gruppe mit 90°.

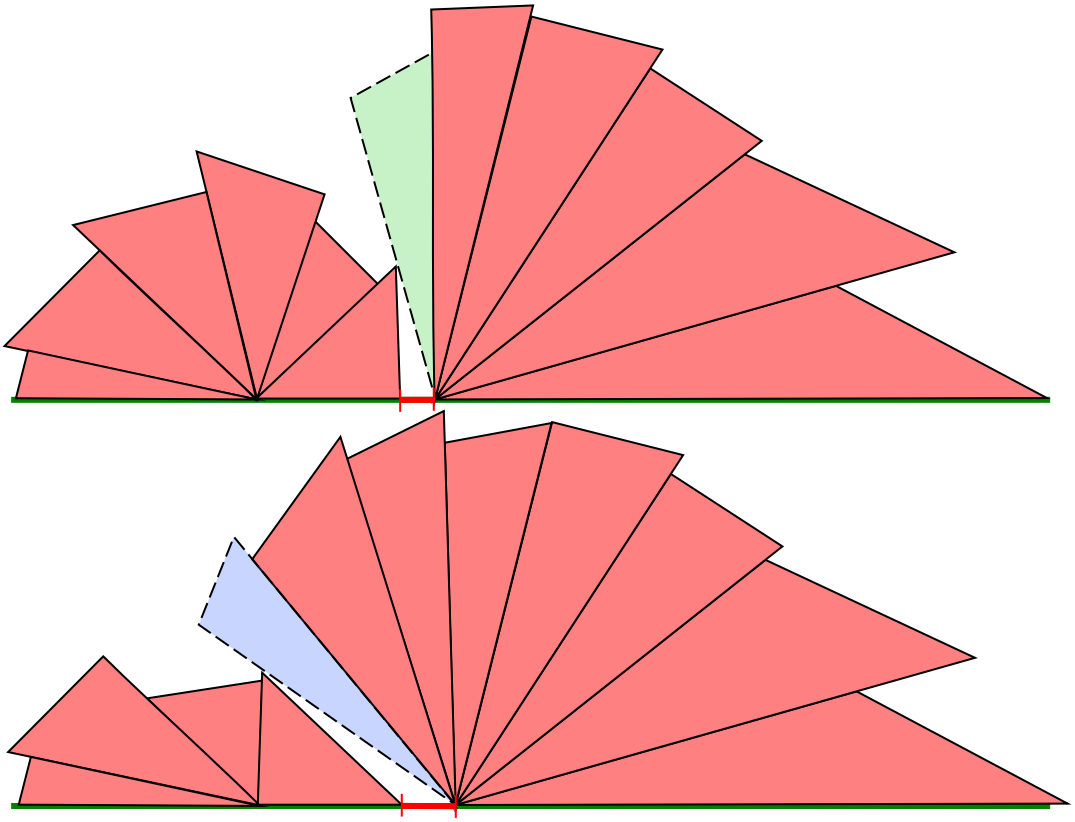


Abb. 6: Beispiele für eingefügte Dreiecke: nahe an 90° (grün), mehr als 90° (blau); der Abstand in rot

Abb. 6 stellt die zwei Situationen grafisch dar. Beim grünen Dreieck, das an eine Gruppe mit etwa 90° angefügt wurde, verschiebt sich die zweite Gruppe weniger als wenn die Gruppe bereits einen sehr großen Winkel hat.

Es ist noch zu beachten, dass das zusätzlich eingefügte Dreieck nicht unbedingt das Dreieck wird, das tatsächlich in der Gruppe ganz links platziert werden wird, denn die Dreiecke der Gruppen werden später noch möglichst platzsparend sortiert. Das macht aber nichts, denn einen Winkel von etwas mehr als 90° hat die Gruppe letztendlich sowieso und dadurch wird der Abstand zur vorherigen Gruppe gering gehalten.

1.5 Sortierung der Dreiecke und grafische Platzierung der Gruppen

Jede Gruppe hat nun einige Dreiecke zugewiesen bekommen, die Reihenfolge der Gruppen ist festgelegt und allen Gruppen außer der letzten wurde ein Basisdreieck zugeordnet.

1.5.1 Sortierung der Dreiecke innerhalb der Gruppe

Die Dreiecke der Gruppe ganz links werden nach der Länge der mittellangen Seite sortiert (die kleinste Seite berührt den Ursprung nicht, die längste und die mittellange Seite schon), sodass das Dreieck mit der kürzesten mittellangen Seite am Basisdreieck anliegt. Die mittellange Seite eines Dreiecks befindet sich dabei stets rechts, während die längste Seite des Dreiecks links ist. Die gleiche Sortierung gilt für die letzte Gruppe, nur spiegelverkehrt (außerdem gibt es kein Basisdreieck).

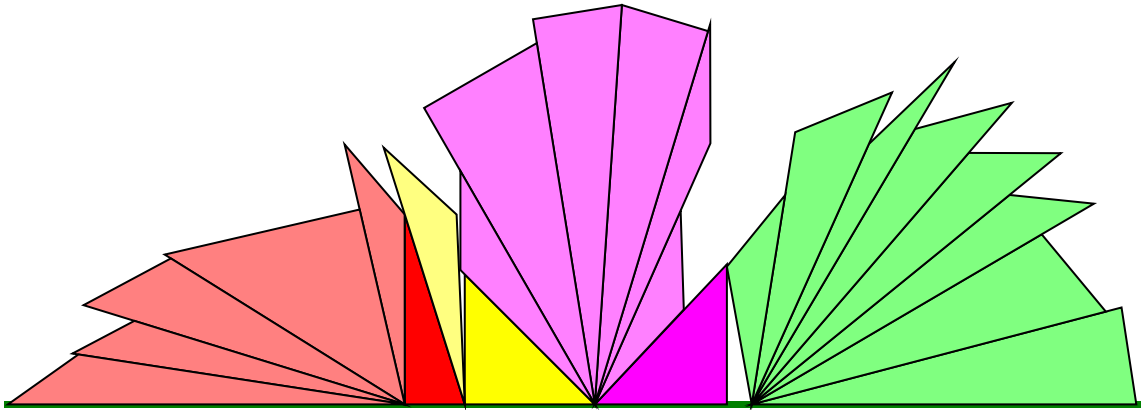


Abb. 7 (Wiederholung von Abb. 5): Sortierung der Dreiecke in ihren Gruppen

Für die Sortierung der Dreiecke innerhalb der anderen Gruppen ist relevant, ob ein Punkt der vorherigen Gruppe überlappt. Das ist dann der Fall, wenn ein Punkt einer Gruppe die (gedacht verlängerte) Kante auf der rechten Seite des Basisdreieck von dieser Gruppe überschreitet (Abb. 8).

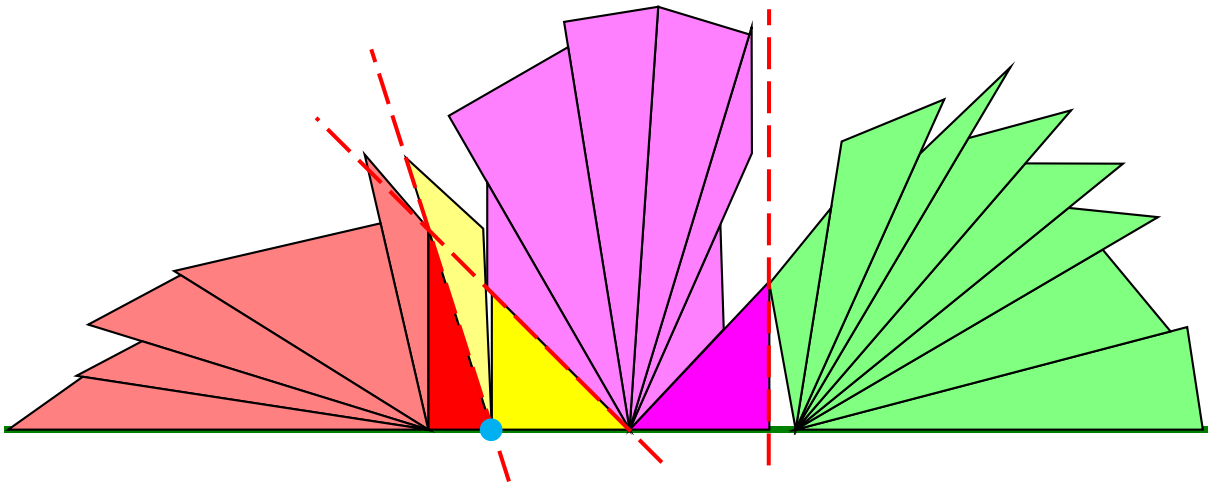


Abb. 8: Die rot gestrichelten Linien sind jeweils entscheidend, um zu bestimmen, ob es in einer Gruppe überlappende Punkte gibt

Von der ersten, roten Gruppe überlappt kein Punkt. Deshalb werden die Dreiecke der zweiten Gruppe wie schon in der ersten Gruppe angeordnet, nämlich links die großen und rechts die kleinen Dreiecke (allerdings gibt es in dieser Gruppe nur ein Dreieck, das auch mit der längsten Seite nach links platziert wird).
Es fällt auf, dass das einzige Dreieck der zweiten Gruppe eigentlich direkt am gelben Basisdreieck platziert werden müsste. Hier tritt aber der Fall ein, dass noch sehr viel Platz ist, und deshalb werden die Dreiecke der Gruppe (hier nur eines), nach links gedreht. Um zu prüfen, wie viel Platz es noch gibt, wird übrigens nicht wie beim Auswählen der Dreiecke für die neue Gruppe der rechts liegende Winkel am Basisdreiecks verwendet (hier der Winkel am hellblauen Punkt), denn es könnte ja ein Punkt überlappen. Stattdessen wird der Punkt benutzt, der zum blau markierten Punkt den größten Winkel hat. Das wäre bei einer Überlappung dann einer der überlappenden Punkte. In Abb. 8 ist dieser Punkt aber identisch mit dem oberen Punkt des Basisdreiecks, weil es keine überlappenden Punkte gibt, also spielt es hier keine Rolle. Wenn der ausgewählte Punkt mit dem größten Winkel sehr hoch liegt, kann es natürlich passieren, dass man die Dreiecke der nächsten Gruppe eigentlich noch viel weiter nach links drehen könnte, weil sie mit der Kante überhaupt nicht kollidieren würden. Das würde aber auch in der Berechnung länger dauern und kommt in der Praxis wohl eher seltener

vor.
In der zweiten Gruppe gibt es sogar zwei Punkte, die überlappen (die Punkte der vorherigen Gruppen zählen auch), also werden die Dreiecke ein bisschen anders sortiert, um sowohl der vorherigen als auch der nächsten Gruppe möglichst viel Platz zu bieten. Die Dreiecke werden wie gehabt nach der Größe der mittellangen Seite sortiert und das kleinste Dreieck kommt nach ganz rechts. Das zweitkleinste kommt aber nach ganz links, das drittkleinste wieder nach rechts und so weiter. So befinden sich die großen Dreiecke in der Mitte und stören links und rechts nicht, und es gibt kleine Dreiecke an den Seiten. Die Dreiecke sind auch entsprechend gedreht, sie zeigen also jeweils mit der mittellangen Seite nach links bzw. rechts.

1.5.2 Grafische Platzierung der Gruppen

Beginnend mit der Gruppe ganz links werden alle Gruppen nebeneinander grafisch platziert. Die erste Gruppe wird so platziert, dass ihr Ursprung der gleiche wie der Ursprung des Koordinatensystems ist. Immer, wenn eine Gruppe platziert wurde, werden alle relevanten Punkte und Kanten gesucht, die für die Kollision mit der nächsten Gruppe wichtig sein könnten. So wird auch die Zahl der Punkte und Kanten, die für die Platzierung der nächsten Gruppe überprüft werden müssen, gering gehalten. Da es auch vorkommen kann, dass Dreiecke von Gruppen sehr weit nach rechts ragen und so für die Kollision mit einer Gruppe weiter rechts relevant sind, werden bei der Suche der relevanten Punkte und Kanten für eine Gruppe auch alle relevanten Punkte und Kanten der Gruppe weiter links berücksichtigt und überprüft, ob diese auch für die nächste Gruppe relevant sind.

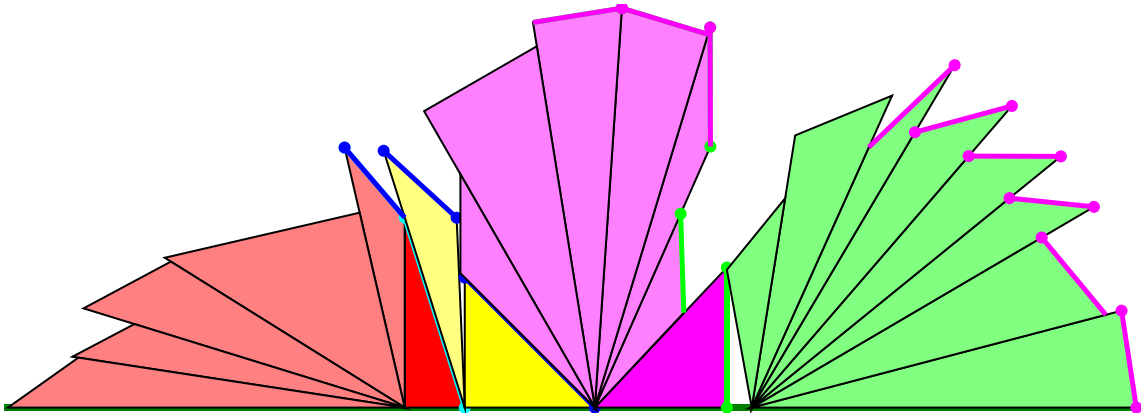


Abb. 9: Die für die Kollision mit der nächsten Gruppe relevanten Punkte und Kanten; zur ersten Gruppe gehört hellblau, zur zweiten blau, zur dritten grün und zur vierten lila (immer vertauschte RGB Farben, aus rot (#FF0000) wird hellblau (#00FFFF), aus gelb (#FFFF00) wird blau (#0000FF) usw.). Wenn ein Punkt/eine Kante für mehrere Gruppen relevant ist, wird er in der Farbe der Gruppe, die sich am weitesten rechts befindet, dargestellt (die blaue Kante in der roten Gruppe ist auch für die rote Gruppe relevant, die lila Kanten der vorletzten Gruppe sind auch für die lilafarbene Gruppe relevant).

Ein Punkt ist relevant, wenn er nicht durch eine Kante verdeckt wird, die sich weiter rechts befindet. Von rechts nach links muss die y-Koordinate der Punkte also immer weiter zunehmen, ansonsten ist ein Punkt nicht sichtbar. Deshalb werden alle Außenkanten (Kanten, die den Ursprung nicht berühren) von rechts nach links durchgegangen. Am Anfang beträgt die minimale y-Koordinate -1, sodass der Punkt ganz rechts auf jeden Fall relevant ist. Von einer Außenlinie wird zuerst geprüft, ob der Punkt mit der größeren y-Koordinate überhaupt sichtbar ist. Ist er nicht sichtbar, ist auch der Punkt mit der kleineren y-Koordinate nicht sichtbar. Ist er aber sichtbar, wird dieser Punkt zu den relevanten Punkten und die Außenkante zu den relevanten Kanten hinzugefügt. Anschließend wird

geprüft, ob auch der Punkt mit der kleineren y-Koordinate sichtbar ist. Ist das der Fall, wird auch dieser zu den relevanten Punkten hinzugefügt. Immer, wenn der höherliegende Punkt relevant ist, wird auch die Außenkante als relevant gezählt. Würde das nicht passieren, könnte es sein, dass z.B. die zweite lilafarbene Kante ganz rechts in Abb. 9 nicht als relevant gewertet wird, weil nur ein Eckpunkt der Kante relevant ist. Dadurch werden aber auch Kanten hinzugefügt, die eigentlich überhaupt nicht relevant sind (z.B. die lilafarbene Kante links auf der lilafarbenen Gruppe). Dies könnte verhindert werden, wenn eine Außenkante nur als relevant gezählt wird, wenn der zweite Eckpunkt der Kante (also der, der gemäß der Richtung der Gruppe als zweites kommt) sichtbar ist. Dazu wären im Programm aber auch mehr Berechnungen nötig.
Die zwei Kanten, die den Ursprung berühren, werden nicht hinzugefügt, denn bei der späteren Berechnung der Position werden alle Punkte und auch automatisch alle Kanten von diesen Punkten aus zum Ursprung berücksichtigt.
Auch die Punkte und Kanten von Gruppen weiter rechts könnten relevant sein, wie oben schon erläutert. Daher werden zusätzlich zu den eigenen Außenkanten auch alle relevanten Kanten der vorherigen Gruppe durchgegangen.

Wenn eine neue Gruppe an eine bestehende (an der rechten Seite) angefügt wird, dann wird sie zuerst so platziert als wäre sie die erste Gruppe, also ist der Ursprung der Gruppe der Punkt $(0|0)$. Dann werden alle Punkt-Kante-Kombinationen zwischen den beiden Gruppen gesucht, also immer ein Punkt von der einen Gruppe und eine Kante von der anderen. Dabei sind nur die relevanten Punkte und Kanten der vorherigen Gruppe(n) wichtig, während bei der neuen Gruppe alle Punkte und Kanten berücksichtigt werden. Da die Kanten von einem Punkt zum Ursprung seiner Gruppe nicht gespeichert werden, zählen natürlich auch diese Kanten zu den Kanten dazu. Wenn sich die beiden theoretisch berühren könnten (die y-Koordinate des Punktes befindet sich also zwischen den y-Koordinaten der beiden Punkte von der Kante), dann wird berechnet, welche Position die neue Gruppe hätte, wenn diese Punkt-Kante-Kombination sich tatsächlich berühren würde. Die größte Position, die durch eine der Kombinationen entsteht, ist dann die neue Position der Gruppe, weil sich in der größtmöglichen Position immer ein Punkt und eine Kante berühren und sich damit keine Kanten schneiden. Der Punkt, an dem der Punkt die Kante berührt, wird mithilfe der Steigung der Kante berechnet, so wie man auch die Werte einer linearen Funktion ausrechnet (nur dass hier die x-Koordinate und nicht die y-Koordinate gefragt ist, also wird die Steigung genau andersherum berechnet, nämlich Seitendifferenz geteilt durch Höhendifferenz). Zur exakten Berechnung siehe die Implementierung bzw. den Quellcode. Sollte es zwischen den beiden Gruppen eine Punkt-Punkt-Kombination statt einer Punkt-Kante-Kombination geben, ist das nicht weiter schlimm, denn eine Punkt-Punkt-Kombination kann immer als eine Punkt-Kante-Kombination betrachtet werden, bei der der Punkt einen Eckpunkt der Kante berührt.

1.6 Das Teilsummenproblem für Dreiecke

Zur Lösung des Teilsummenproblems werden alle Kombinationsmöglichkeiten der Dreiecke solange betrachtet, bis ein zufriedenstellendes Ergebnis gefunden wurde, um Dreiecke mit ihren kleinsten Winkeln in einen vorgegebenen Platz einzufügen. Dabei wird zur Minderung der Anzahl der Kombinationsmöglichkeiten erst bei einer gewissen Anzahl von Dreiecken begonnen und abgebrochen. Die minimale Anzahl kann mit dem Quotient aus dem benötigten Winkel und dem größten Winkel aller kleinsten Winkel der Dreiecke berechnet werden, für die maximale Anzahl verwendet man den kleinsten Winkel aller kleinsten Winkel der Dreiecke.
Die Summe der kleinsten Winkel für eine Kombination muss nicht exakt passen; stattdessen gibt es

eine Toleranz, die hier auf 0,2° festgelegt ist. Für eine Suche nach 180° zählt also auch schon 179,8° als ein ausreichend gutes Ergebnis.

1.7 Variante: Jede Gruppe soll ein kleines Dreieck enthalten

Die meisten Gruppen haben bereits ein kleines Dreieck, das direkt am Basisdreieck platziert werden kann, um den Abstand zur nächsten Gruppe zu verringern. Da das aber nicht immer der Fall ist, ist die Idee dieser Variante, den Teilalgorithmus für das Einfügen der übrigen Dreiecke (Abschnitt 1.4) leicht zu modifizieren. Zuerst werden so viele kleine Dreiecke gesucht, wie es später Gruppen geben wird (also die Anzahl der Basisdreiecke plus 1 für die letzte Gruppe ohne Basisdreieck). Ob ein Dreieck groß oder klein ist, wird wieder wie bei der Sortierung der Dreiecke innerhalb der Gruppe mit der Länge der mittellangen Seite bestimmt. Wenn Dreiecke für einen bestimmten Platz gefragt sind, wird zuerst das größtmögliche (nach kleinstem Winkel) der kleinsten Dreiecke ausgewählt, was diesen Platz nicht überschreitet. Sollte es keines geben, wird stattdessen das größte der kleinsten Dreiecke wieder den „normalen“ Dreiecken zugeordnet, denn ansonsten würde ein kleines Dreieck am Ende übrigbleiben. Gibt es aber ein passendes kleines Dreieck, wird nach Dreiecken für den noch übrigbleibenden Platz wie früher gesucht. Sobald alle Basisdreiecke platziert wurden, bleibt noch ein kleinstes Dreieck übrig, das für die letzte Gruppe „aufgehoben“ wurde. Dieses wird dann auch als „normales“ Dreieck betrachtet.

Diese Variante ist deutlich schneller als die normale, weil der Suchalgorithmus für passende Kombinationen von Dreiecken für einen bestimmten Platz nicht mehr so viele Dreiecke zur Auswahl hat. Auch in den Beispielen (Abschnitt 3.1) ist dies erkennbar.

2. Umsetzung

Das Programm wurde in Python 3 geschrieben und unter Windows 10 und macOS X mit Python 3.6 und unter Linux Mint mit Python 3.5 getestet. Bei korrekter Ausführung treten erwartungsgemäß keine Fehler auf. Zum Teil wurden recht neue Sprachelemente verwendet, weshalb ältere Versionen unter Umständen nicht funktionieren.
Um die SVG-Grafiken zu erstellen, benutze ich das Drittanbietermodul `svgwrite` unter der MIT-Lizenz. Zur Verwendung siehe Abschnitt 2.4.

Hier werden die genauere Umsetzung und die Struktur des Programms erläutert. Um nicht alles doppelt zu erklären, finden sich detailreiche Erklärungen zur Funktionsweise der einzelnen Funktionen/Methoden bzw. Klassen zusammen mit dem Quellcode in Abschnitt 4.

Das Programm besteht hauptsächlich aus zwei Modulen, nämlich `trianglearranger` (`Aufgabe2/trianglearranger.py`) und `geometry` (`Aufgabe2/geometry.py`). Das `trianglearranger`-Modul enthält den eigentlichen Algorithmus und das `geometry`-Modul stellt Funktionalitäten wie die Darstellung eines platzierten und eines nichtplatzierten Dreiecks und die Gruppen bereit.

2.1 Einlesen und Darstellung der Daten

Die einzelnen, unplatzierten Dreiecke werden in der Klasse `geometry.Triangle` gespeichert. Sie enthält die längen der Seiten sowie die Größen der Winkel. Die Seitenlängen werden (in aufsteigender Reihenfolge, von der kürzesten zur längsten Seite) in den Attributen `shortest_line`, `middle_line` und `longest_line` gespeichert, den jeweils gegenüberliegenden Winkel wird zusätzlich

ein `_angle` angehängt. Für das Einlesen ist die Funktion `geometry.parse_file` zuständig, der der Pfad zur Datei übergeben wird. Für jede Zeile mit einem Dreieck wird die statische Methode `Triangle.from_str` aufgerufen, die die Längen zwischen den einzelnen Punkten berechnet und diese an den eigentlichen Konstruktor weitergibt.
Weil zur Definition eines Dreiecks eigentlich nur dessen drei Seitenlängen benötigt werden, gibt es noch ein von mir entworfenes Format. Dateien beginnen (als Kennzeichnung) mit `type=lengths` und jede Zeile enthält die drei Längen, durch Leerzeichen getrennt, entweder als `float` (mit Punkt) oder `int`. Das Format kann ebenfalls von `geometry.parse_file` eingelesen werden, wobei stattdessen `Triangle.from_lengths_str` benutzt wird, um eine Zeile mit einem Dreieck einzulesen.

2.2 Die TriangleGroup-Klasse

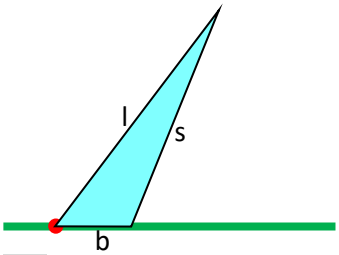
Die `TriangleGroup`-Klasse befindet sich im Modul `geometry` und stellt eine Gruppe dar. Zur Speicherung der Objekte der unplatzierten Dreiecke wird das Attribut `triangles`, eine Liste, verwendet. Ein eventuelles Basisdreieck wird von `TriangleGroup.base_triangle` gespeichert, ansonsten ist `base_triangle` `None`. Die Richtung wird in `direction` als String – entweder `"counter-clockwise"` oder `"clockwise"` – gespeichert. Die aktuelle Summe aller kleinsten Winkel (bzw. vom Basisdreieck einer der anderen Winkel) befindet sich im Attribut `angle`. Beim Platzieren der Gruppe werden alle Punkte bzw. Außenkanten, die nicht den Ursprung `TriangleGroup.origin` berühren, in `points` und `outer_lines` gespeichert. Nach dem Platzieren werden die relevanten Punkte/Außenkanten gesucht und zu den Attributen `relevant_points` und `relevant_lines` hinzugefügt. Außerdem gibt es die zwei Attribute `base_triangle_outer_angle` und `outer_angle`, wobei ersteres den Winkel des Basisdreiecks an der x-Achse speichert, der nicht den Ursprung berührt, und letzteres den Winkel an derselben Stelle, aber zum Punkt der Gruppe mit dem größten Winkel (letzterer wird verwendet, um zu wissen, ob die Dreiecke der nächsten Gruppe weiter nach links gedreht werden können).

Sobald die Gruppe alle Dreiecke mit der Methode `append()` erhalten hat, kann sie mit `place()` platziert werden. Dafür bekommt `place` das Objekt der bereits platzierten linken Gruppe (`pre_group`) und der (noch unplatzierten) rechten Gruppe (`next_group`) übergeben. Beide werden in entsprechenden Attributen der Klasse gespeichert. Für die Sortierung der Dreiecke ist die Methode `get_sorted_triangles` zuständig. Diese gibt zurück, wo und wie (zu den Platzierungsmethoden siehe Abschnitt 2.2.1) die Dreiecke platziert werden sollen.
Die statische Methode `PlacedTriangle.place_triangle` platziert dann jedes Dreieck an einer bestimmten Stelle in einem bestimmten Winkel. Außerdem werden alle Punkte und die Außenkante in den entsprechenden Attributen `points` und `outer_lines` gespeichert. Das Suchen der Position und das Suchen der für die nächste Gruppe relevanten Punkte und Außenkanten funktioniert genau so wie in der Lösungsidee beschrieben, für Details siehe den Quellcode. Nachdem die Position gefunden wurde, werden alle Dreiecke zu dieser Position mit der Methode `PlacedTriangle.set_x()`, die die x-Koordinate des Ursprungs `PlacedTriangle.origin` ändert, verschoben. Die `PlacedTriangle`-Klasse wird weiter unten besprochen.

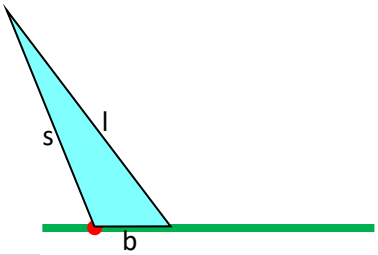
2.2.1 Platzierungsmethoden

Ein Dreieck kann in einer Gruppe auf sechs unterschiedliche Arten platziert werden. Ein Basisdreieck kann auf genau zwei Arten platziert werden (siehe Lösungsidee) und ein „normales“ Dreieck kann ebenfalls auf zwei Arten platziert werden. Die Art der Platzierung wird durch einen String mit drei Buchstaben angegeben. Für das Basisdreieck wird die Platzierung im Attribut

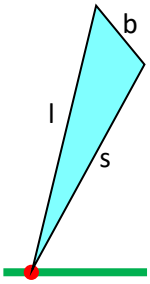
`placement_base_triangle` angegeben. Ein String für eine Platzierungsmethode enthält immer die drei Zeichen `l`, `s` und `b` in unterschiedlicher Reihenfolge. Die Zeichen stehen dafür respektive für die längste Seite („longest arm“), die mittellange Seite („shortest arm“, also der kleinste Schenkel) und kürzeste Seite („base line“, die eigentliche kürzeste Seite). Der erste Buchstabe gibt für das Basisdreieck die Seite an, die an der x-Achse platziert wird (beim Basisdreieck also immer `b`). Für alle anderen Dreiecke gibt er die Seite des Dreiecks an, die gemäß der Richtung der Gruppe zuerst platziert wird. Der zweite Buchstabe gibt die andere Seite an, die wie die erste am Ursprung liegt, allerdings als zweites platziert wird. Die letzte Seite müsste man jetzt eigentlich nicht mehr angeben, aber für eine bessere Verständlichkeit steht der letzte Buchstabe für die Außenkante. Für die Basisdreiecke gibt es also zwei mögliche Strings: `bsl` und `bls`. Für alle anderen Dreiecke (die ja immer mit dem kleinsten Winkel an der x-Achse platziert werden) gibt es die Strings `slb` und `lsb`. Diese Strings werden zum Beispiel für jedes Dreieck von der `TriangleGroup.get_sorted_triangles`-Methode zurückgegeben (siehe Abschnitt 2.2). Abb. 10 zeigt für jede Platzierungsmethode ein Beispiel.



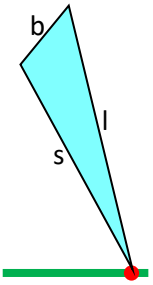
`bls` – die mittellange Seite berührt den Ursprung nicht



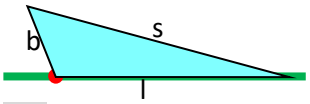
`bsl` – die längste Seite berührt den Ursprung nicht



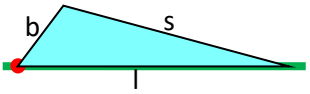
`slb` – die kürzeste Seite berührt den Ursprung nicht



`lsb` – die kürzeste Seite berührt den Ursprung nicht



`sbl` – die längste Seite berührt den Ursprung nicht



`lbs` – die mittellange Seite berührt den Ursprung nicht

Abb. 10: Platzierungsmethoden der Dreiecke mit Beispielen. Die erste Zeile ist für Basisdreiecke, die zweite für andere Dreiecke und die Platzierungen der letzten Zeile kommen nicht vor. Der rote Punkt steht jeweils für den Ursprung der Gruppe; alle Gruppen verlaufen gegen den Uhrzeigersinn.

2.2.2 Platzierte Dreiecke - PlacedTriangle

Die eben erwähnte `PlacedTriangle`-Klasse speichert ein platziertes Dreieck mithilfe von dessen drei Eckpunkten. Das `Triangle`-Objekt, dessen Größen für dieses platzierte Dreieck benutzt wurden, befindet sich im Attribut `original_triangle`, im Attribut `origin` wird der Ursprung der Gruppe, zu der das Dreieck gehört, gespeichert. In einer Gruppe wird zuerst das Basisdreieck platziert. Danach werden alle anderen Dreiecke entweder im Uhrzeigersinn oder gegen den Uhrzeigersinn gemäß der

Richtung der Gruppe platziert. Es ergibt also Sinn, auch die Kanten der Dreiecke in dieser Reihenfolge zu speichern. Die Länge der zuerst kommenden Kante wird in `lower_arm_length`, die der danach folgenden Kante in `upper_arm_length` gespeichert. Die jeweiligen Ecken dieser Kanten, die nicht der Ursprung sind, befinden sich in `lower_arm_point` bzw. `upper_arm_point` und der Winkel, der jeweils zwischen der Außenkante und der Kante entsteht in `lower_arm_angle` bzw. `upper_arm_angle`. `PlacedTriangle.origin_angle` enthält den Winkel am Ursprung, also den kleinsten Winkel, sofern es sich nicht um ein Basisdreieck handelt.

2.3 Der Algorithmus

Der eigentliche Algorithmus befindet sich im `trianglearranger`-Modul. Zum Finden einer möglichst guten Anordnung wird die Funktion `search_arrangement()` benutzt, welcher die Liste von `Triangle`-Objekten übergeben wird. Für jede Basisdreieck-Kombination, die der Generator `base_triangles_generator` abwirft (`yield`), ruft sie die Funktionen `arrange_triangles` und `arrange_triangles_with_smallelst` auf. Erstere enthält den Algorithmus in seiner ursprünglichen Form, letztere ist die Implementierung der Variante, bei der möglichst jede Gruppe ein kleines Dreieck erhalten soll.

2.3.1 Der Basisdreieck-Generator base_triangles_generator

Diese Funktion implementiert den in Abschnitt 1.3 beschriebenen Teilalgorithmus. Es werden nacheinander die verschiedenen Auswahlen für die Basisdreiecke mit `yield` „abgeworfen“.

2.3.2 Anordnung der Dreiecke – arrange_triangles und arrange_triangles_with_smallest

Die Funktion `arrange_triangles` ordnet die als Parameter übergebene Liste von Basisdreiecken mit den ebenfalls übergebenen übrigen Dreiecken wie beschrieben an.
`arrange_triangles_with_smallest` implementiert die in Abschnitt 1.7 beschriebene Variante, bei der versucht wird, für jede Gruppe ein kleines Dreieck aufzuheben.

2.4 Grafische Ausgabe als SVG

Die Ausgabe erfolgt wie schon erwähnt mit dem `svgwrite`-Modul. Wie der Name impliziert, ist das Modul nur dazu geeignet, SVG-Grafiken zu erstellen, nicht aber zu lesen. Jedes SVG-Element hat eine eigene Klasse (z.B. `svgwrite.container.Group`, `svgwrite.shapes.Line`, `svgwrite.shapes.Circle`, `svgwrite.shapes.Polygon`) und mit der Klasse `svgwrite.Drawing` lässt sich eine Datei mit `.save()` speichern, nachdem mit der Methode `.add()` entsprechende Elemente hinzugefügt wurden. `.add()` funktioniert auch, um Elemente zu Gruppen hinzuzufügen.

3. Beispiele

3.1 Übersicht

Name	**Anzahl Dreiecke**	**Summe der kleinsten Winkel**	**Gesamt-abstand**	**Methode, mit der der beste Gesamtabstand gefunden wurde**	**Anzahl Basis-dreiecke**	**Zeit `normal`**	**Zeit `smallest`**
dreiecke1.txt	5	299,91°	142,84 m	`normal` oder `smallest`	1	0,0006 s	0,0006 s
dreiecke2.txt	5	96,90°	0,00 m	`lt360`	0	0,0000 s	0,0000 s
dreiecke3.txt	12	267,75°	72,46 m	`smallest`	1	0,0139 s	0,0355 s
dreiecke4.txt	23	399,27°	171,71 m	beim Ausprobieren verschiedener Möglichkeiten für den Algorithmus	3	2,2053 s	0,4286 s
dreiecke5.txt	37	1178,15°	656,78 m	`normal`	15	67,2897 s	10,437 s

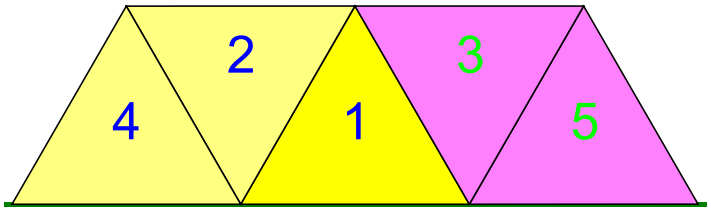
Die Methode `normal` steht für den normalen Algorithmus wie er in der Lösungsidee beschrieben wurde. `smallest` bezeichnet die Variante, in der möglichst jede Gruppe ein kleines Dreieck enthalten soll (siehe Abschnitt 1.7). `lt360` heißt „less than 360°“ und bedeutet, dass die Methode für eine Summe der kleinsten Winkel, die weniger als 360° beträgt, das beste Ergebnis geliefert hat (Abschnitt 1.2). Das beste Ergebnis für dreiecke4.txt mit `normal` oder `smallest` ist eigentlich etwas schlechter, diese Anordnung ist beim Ausprobieren verschiedener Möglichkeiten für den Algorithmus entstanden (siehe die Erläuterungen zu dreiecke4.txt).

Schon hier ist deutlich zu erkennen, dass die Zeit für den normalen Algorithmus deutlich länger ist als die Zeit für die `smallest`-Variante. Das fällt besonders bei dreiecke5.txt auf. Hier liefert der normale Algorithmus ein besseres Ergebnis von 657 Metern als die `smallelst`-Variante, bei der das beste Ergebnis bei nur 669 Metern mit 10 Basisdreiecken liegt. Dafür benötigt die `smallelst`-Variante aber auch weniger Zeit.

Eine Analyse mit dem Profiler zeigt, dass die meiste Zeit von der Funktion benötigt wird, die das Teilsummenproblem benutzt, um Dreiecke in einen vorgegebenen Platz zu füllen. Das war zu erwarten, zumal diese Funktion alle Möglichkeiten so lange durchprobiert bis eine akzeptable Lösung gefunden wurde. Um auch eine größere Anzahl von Dreiecken in angemessener Zeit anzuordnen, ist es durchaus sinnvoll, die Toleranz, ab der ein Ergebnis als akzeptabel zählt, für mehr Dreiecke höher einzustellen. Dadurch können einerseits auch viele Dreiecke in einer guten Zeit angeordnet werden, andererseits hat man die Möglichkeit zu bestimmen, wie gut das Ergebnis werden soll. Zu dieser Erhöhung der Geschwindigkeit siehe auch die Erläuterungen zu dreiecke5.txt, wo das Ergebnis um etwa 5 Meter schlechter wird, dafür aber auch eine zehnfache Geschwindigkeitserhöhung möglich ist.

3.2 Erläuterungen zu den Beispielen

dreiecke1.txt



Alle Dreiecke sind identische gleichseitige Dreiecke. Die Kantenlänge beträgt etwa 143m, was auch die Gesamtlänge ist. Das Programm erstellt für diese Datei während den Berechnungen folgende Ausgabe:

```
ALL TRIANGLES:
  ID 1 shortest line:142.8 longest line:142.9 smallest angle:60.0
  […]
  ID 5 shortest line:142.8 longest line:142.9 smallest angle:60.0
Smallest angle sum = 299.91
Angle sum <= 360°
  time:  0.0003
  BEST 142.95
BASE len: 1 distance:0.000
  time normal:   0.0006
  time smallest: 0.0006
  distance normal:   142.84
  distance smallest: 142.84
  BEST 142.84
BASE len: 2 distance:142.836
  new base distance too long: 142.836 >= 142.836
whole time: 0.0017
```

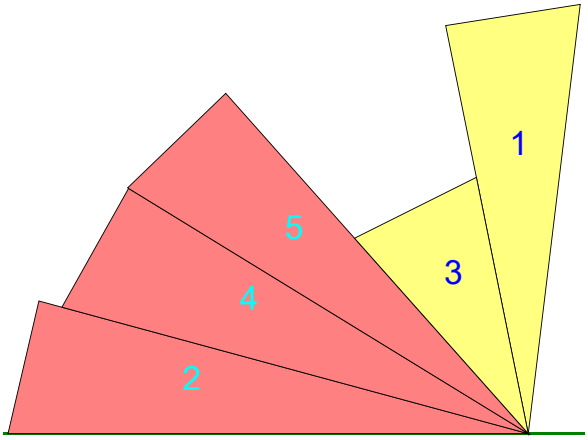
Die Summe der kleinsten Winkel beträgt hier etwa 299,91°. Eigentlich wäre eine Summe von 300° (=60°*5) zu erwarten, aber die Punkte in den Beispieldaten sind leider auf Ganzzahlen gerundet, wodurch die Winkel nicht genau 60° groß sind, sondern etwas kleiner bzw. etwas größer.
In der Ausgabe befindet sich zuerst eine Auflistung aller Dreiecke sowie ihre wichtigsten Größen. Das ist hier nicht weiter interessant, da die Angaben immer die gleichen sind. Da die Summe der kleinsten Winkel kleiner als 360° ist, können die Dreiecke in zwei Gruppen angeordnet werden. Davon hat eine Gruppe 3 und die andere 2 Dreiecke, was schon zu einem sehr guten Ergebnis führt. Mit einem Basisdreieck wird der Gesamtabstand nur geringfügig verbessert, denn jetzt liegt ein Basisdreieck in der Mitte (siehe Grafik oben, das gelbe Dreieck in der Mitte ist hervorgehoben), und dieses Basisdreieck liegt natürlich mit seiner kleinsten Seite an der x-Achse.
Die Distanz mit einem Basisdreieck ist 0, weil die Distanz wie in Abschnitt 1.3 beschrieben immer ohne das zum Kriterium zuletzt hinzugekommene Dreieck berechnet wird, denn es könnte ein Basisdreieck ganz rechts liegen, welches dann nicht hinzuzählen würde (siehe auch Beispiel 5). Für zwei Basisdreiecke ist die Distanz identisch mit dem bisher besten Gesamtabstand, und daher lautet das Ergebnis 143 Meter.

Hier noch die erforderliche Ausgabe für die Lage der Dreiecke:

```
dreiecke1.txt
```

```
Gesamtabstand: 143m
Dreiecke:
D 4 (   0|   0) (-143|   0) ( -71| 124)
D 2 (   0|   0) ( -71| 124) (  71| 124)
D 1 (   0|   0) (  71| 124) ( 143|   0)
D 3 ( 143|   0) (  71| 124) ( 214| 124)
D 5 ( 143|   0) ( 214| 124) ( 286|   0)
```

dreiecke2.txt



Hier ist die Summe aller kleinsten Winkel kleiner als 180°, weshalb ein Gesamtabstand von 0 Metern entsteht. In der Grafik ist auch zu sehen, dass die Dreiecke in zwei Gruppen aufgeteilt werden und die zweite Gruppe nach links gedreht wird und damit nicht mehr auf der x-Achse liegt, weil zur vorherigen Gruppe noch Platz ist.

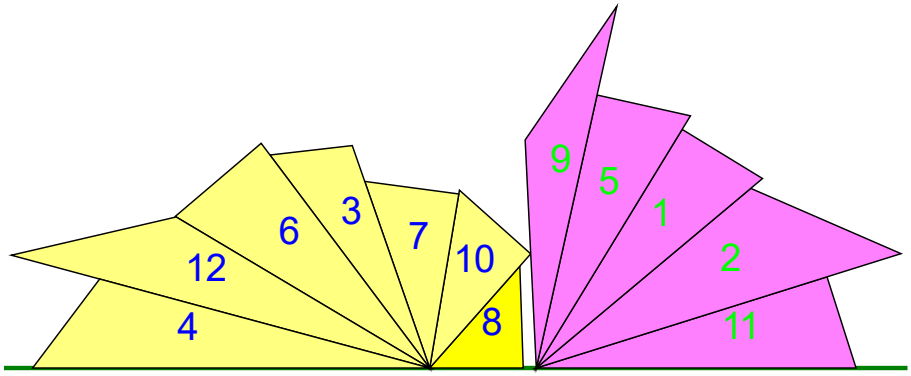
Hier die Ausgabe während der Berechnung:

```
ALL TRIANGLES:
  ID 1 shortest line:150.0 longest line:476.1 smallest angle:18.3
  ID 2 shortest line:150.0 longest line:572.9 smallest angle:15.2
  ID 3 shortest line:150.0 longest line:287.9 smallest angle:30.2
  ID 4 shortest line:150.0 longest line:532.1 smallest angle:16.4
  ID 5 shortest line:150.0 longest line:517.6 smallest angle:16.8
Smallest angle sum = 96.90
Angle sum <= 360°
  time: 0.0006
  BEST 0.00
BASE len: 1 distance:0.000
  new base distance too long: 0.000 >= 0.000
whole time: 0.0007
```

Nach der Anordnung mit dem Verfahren für Winkel bis 360° entsteht mit einem Basisdreieck eine Länge von 0 (denn das zuletzt hinzugekommene Basisdreieck wird ja nicht mitgezählt), was aber die bisher beste Distanz nicht übertrifft.
Die Ausgabe für die Anordnung der Dreiecke lautet wie folgt:

```
dreiecke2.txt
Gesamtabstand: 0m
Dreiecke:
D 2 (   0|   0) (-539| 146) (-573|   0)
D 4 (   0|   0) (-440| 270) (-514| 139)
D 5 (   0|   0) (-333| 375) (-441| 271)
D 3 (   0|   0) (-191| 215) ( -57| 282)
D 1 (   0|   0) ( -91| 449) (  57| 473)
```

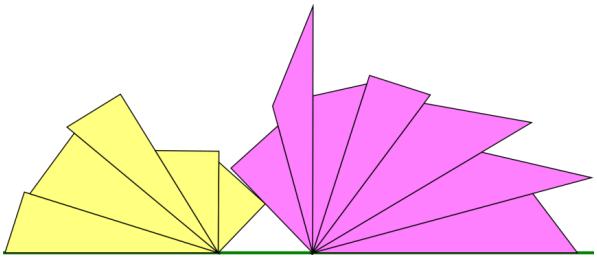
dreiecke3.txt



Eine Anordnung in zwei Gruppen ist aufgrund der kleinen Summe der kleinsten Winkel von weniger als 360° möglich, aber es resultiert ein schlechterer Gesamtabstand als mit einem Basisdreieck, wie folgender Ausgabe entnommen werden kann:

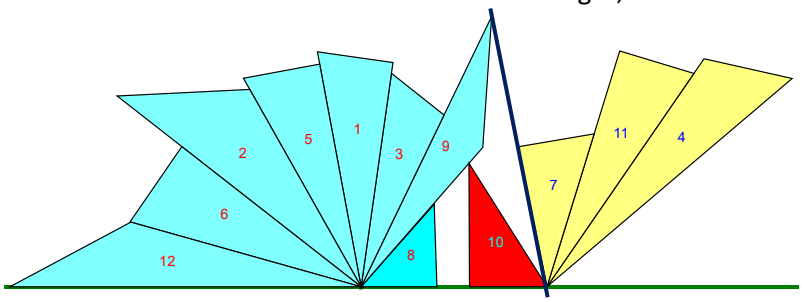
```
ALL TRIANGLES:
  ID 1 shortest line: 64.0 longest line:200.5 smallest angle:18.6
  ID 2 shortest line:111.3 longest line:259.7 smallest angle:22.5
  ID 3 shortest line: 56.3 longest line:180.9 smallest angle:17.7
  ID 4 shortest line: 75.7 longest line:269.8 smallest angle:15.1
  ID 5 shortest line: 65.0 longest line:200.8 smallest angle:18.9
  ID 6 shortest line: 76.7 longest line:201.2 smallest angle:22.3
  ID 7 shortest line: 64.5 longest line:134.2 smallest angle:28.7
  ID 8 shortest line: 63.6 longest line: 92.4 smallest angle:43.4
  ID 9 shortest line:110.1 longest line:251.7 smallest angle:15.4
  ID10 shortest line: 64.9 longest line:122.5 smallest angle:32.0
  ID11 shortest line: 65.0 longest line:217.3 smallest angle:17.4
  ID12 shortest line:115.0 longest line:294.4 smallest angle:15.7
Smallest angle sum = 267.75
Angle sum <= 360°
  time: 0.0042
  BEST 96.07
BASE len: 1 distance:0.000
  time normal:   0.0026
  time smallest: 0.0052
  distance normal:   72.46
  distance smallest: 72.50
  BEST 72.46
BASE len: 2 distance:56.294
  time normal:   0.0078
  time smallest: 0.0031
  distance normal:   119.85
  distance smallest: 119.85
BASE len: 2 distance:63.561
  time normal:   0.0082
  time smallest: 0.0167
  distance normal:   128.50
  distance smallest: 128.50
BASE len: 3 distance:119.855
  new base distance too long: 119.855 >= 72.456
whole time: 0.0495
```

Das Ergebnis mit zwei Gruppen ist mit 96,07 schlechter als das beste Ergebnis (72,46 mit einem Basisdreieck) und sieht so aus:



Wie in Abschnitt 1.3 beschrieben, wird das zuletzt hinzugekommene Basisdreieck nicht zur Berechnung der Länge der Basisdreiecke hinzugezählt. Hier wäre dies gar nicht nötig und das Abbruchkriterium wäre sogar früher erfüllt.

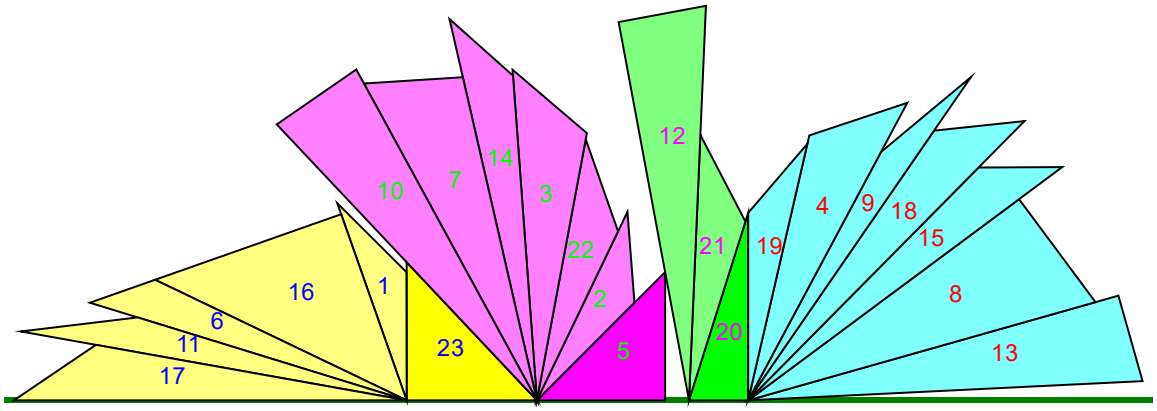
Mit zwei Basisdreiecken funktioniert es nicht so gut, denn hier würden die Basisdreiecke allein schon mehr Platz einnehmen. Links die Anordnung für die oben blau markierte Stelle mit zwei Basisdreiecken und einem Gesamtabstand von 128,50. Die gelbe Gruppe wird leicht nach links gedreht, weil die Gruppe zuvor noch mehr Platz bietet (blaue Linie, vom rechten Punkt des Basisdreiecks auf der x-Achse zum Punkt mit dem größten Winkel für die rote Gruppe). Hier tritt auch der seltene Fall ein, dass eine Gruppe tatsächlich noch weiter nach links gedreht werden könnte (siehe Abschnitt 1.5.2).



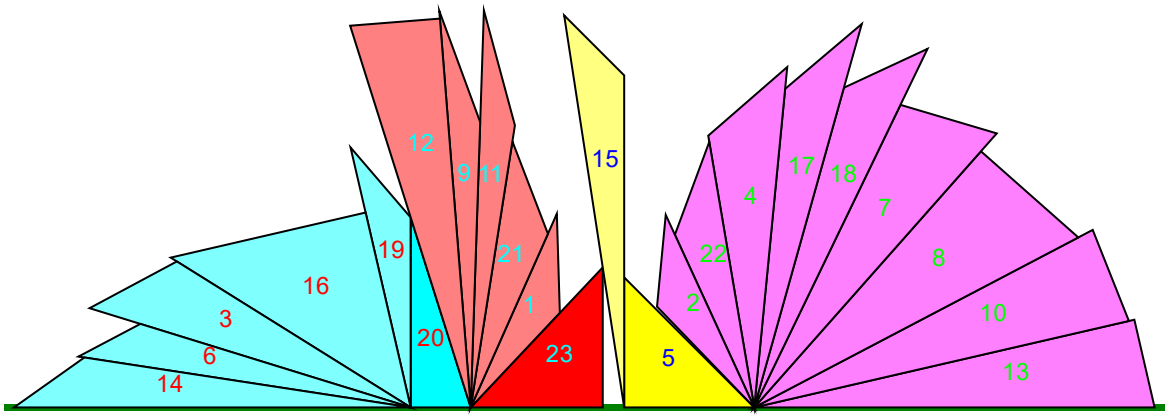
Hier die Platzierung der Dreiecke:

```
dreiecke3.txt
Gesamtabstand: 72m
Dreiecke:
D 4 (   0|   0) (-270|   0) (-224|  60)
D12 (   0|   0) (-284|  77) (-172| 103)
D 6 (   0|   0) (-173| 103) (-114| 153)
D 3 (   0|   0) (-109| 145) ( -53| 151)
D 7 (   0|   0) ( -44| 127) (  20| 118)
D10 (   0|   0) (  20| 121) (  69|  78)
D 8 (   0|   0) (  61|  69) (  64|   0)
D 9 (  72|   0) (  65| 155) ( 127| 246)
D 5 (  72|   0) ( 114| 185) ( 177| 171)
D 1 (  72|   0) ( 172| 162) ( 226| 129)
D 2 (  72|   0) ( 218| 122) ( 320|  78)
D11 (  72|   0) ( 270|  62) ( 290|   0)
```

dreiecke4.txt



Diese Dreiecke können mit einem Gesamtabstand von fast 173,09 Metern angeordnet werden. Während des Ausprobierens verschiedener Möglichkeiten für den Algorithmus bin ich allerdings auf eine etwas bessere Anordnung gestoßen, diese benötigt nur 171,71 Meter:



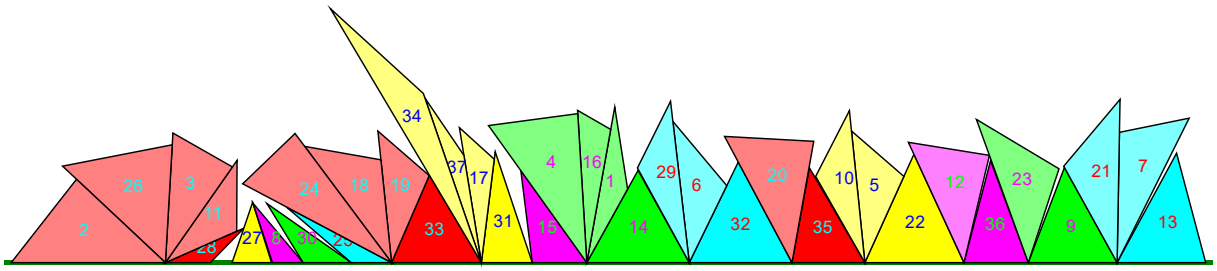
Diese Begebenheit zeigt, dass schon kleine Veränderungen am Quellcode leichte Veränderungen am Ergebnis hervorrufen können. Durch eine Veränderung werden zufällig andere Dreiecke ausgewählt, die manchmal in etwa besseren und manchmal in etwas schlechteren Ergebnissen resultieren. Auffällig ist auch, dass in der besseren Anordnung dieselben Basisdreiecke D23, D5 und D20 verwendet werden.

Die Ausgabe für diese Dreiecke gleicht sich mit den anderen und ist nicht weiter interessant. Es werden Anordnungen mit 2, 3, 4 und dann wieder 3 Basisdreiecke getestet, 5 Basisdreiecke sind dann zu lang. Das beste Ergebnis wird mit der ersten Auswahl von 3 Basisdreiecken erzielt, die zweite Auswahl mit 3 Basisdreiecken ist mit längeren Basisdreiecken deutlich schlechter, nur 231 statt 173 Meter.
Die Dreiecke und ihre Positionen lauten wie folgt:

```
dreiecke4.txt
Gesamtabstand: 173m
Dreiecke:
D17 (   0|   0) (-199|   0) (-158|  28)
D11 (   0|   0) (-196|  35) (-136|  42)
D 6 (   0|   0) (-161|  49) (-128|  61)
D16 (   0|   0) (-128|  61) ( -33|  94)
D 1 (   0|   0) ( -35| 100) (  -0|  65)
```

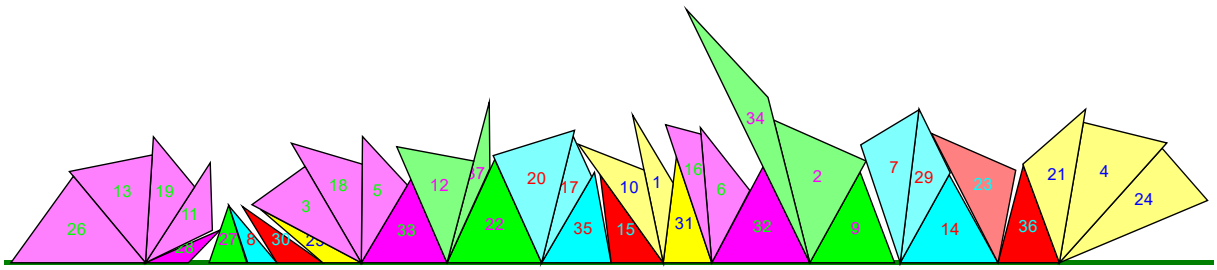
```
D23 (   0|   0) (   0|  70) (  66|   0)
D10 (  66|   0) ( -66| 140) ( -26| 168)
D 7 (  66|   0) ( -22| 161) (  28| 164)
D14 (  66|   0) (  22| 193) (  54| 165)
D 3 (  66|   0) (  54| 168) (  91| 135)
D22 (  66|   0) (  91| 132) ( 107|  86)
D 2 (  66|   0) ( 112|  95) ( 115|  50)
D 5 (  66|   0) ( 131|  65) ( 131|   0)
D12 ( 143|   0) ( 107| 192) ( 152| 200)
D21 ( 143|   0) ( 149| 135) ( 172|  91)
D20 ( 143|   0) ( 173|  95) ( 173|   0)
D19 ( 173|   0) ( 173|  95) ( 203| 130)
D 4 ( 173|   0) ( 204| 134) ( 254| 151)
D 9 ( 173|   0) ( 240| 126) ( 286| 164)
D18 ( 173|   0) ( 267| 137) ( 313| 142)
D15 ( 173|   0) ( 290| 118) ( 332| 118)
D 8 ( 173|   0) ( 310| 102) ( 349|  50)
D13 ( 173|   0) ( 361|  53) ( 373|  10)
```

dreiecke5.txt



Diese Anordnung mit einem Gesamtabstand von 657 Metern zu finden, dauert deutlich länger als bei den anderen Beispieldaten (siehe auch die Tabelle in Abschnitt 3.1). Wie in Abschnitt 3.1 bereits beschrieben, muss hier der Algorithmus in seiner ursprünglichen Form (`normal`) verwendet werden, um zu einem guten Ergebnis zu gelangen, ansonsten wäre es deutlich schlechter. Besonders interessant sind hier die Dreiecke am Anfang, denn offensichtlich wurden sehr viele kleine Dreiecke nebeneinander angeordnet, sodass sie sehr wenig Platz benötigen. Zwischen Dreieck D28 und D27 ist der noch zur Verfügung stehende Platz sogar negativ. Das letzte Dreieck ist im Gegensatz zu den anderen Beispieldaten ein Basisdreieck. Nur durch das Ignorieren der Länge des zuletzt zu einem Kriterium hinzugekommenen Basisdreiecks (siehe Abschnitt 1.3) ist es möglich, diese Anordnung zu finden.

Weil die Berechnung so lange dauert, wurde versucht, die Toleranz für die Funktion, die passende Dreiecke für eine Lücke sucht, auf 3° (statt 0,2°) zu erhöhen. Das Ergebnis ist im Folgenden dargestellt, es hat einen Gesamtabstand von 661 Metern.



Dieses Ergebnis lässt sich mit einer Zeit von knappen 7 Sekunden mit dem normalen Algorithmus berechnen und ist damit etwa zehnmal so schnell wie vorher. Große verbleibende Lücken durch die höhere Toleranz sind nicht wirklich erkennbar.

Da die Ausgabe während der Berechnung entsprechend lang ist und keine weiter wichtigen Informationen enthält, wird sie hier nicht weiter besprochen.

Hier die Ausgabe für die Anordnung der Dreiecke:

```
dreiecke5.txt
Gesamtabstand: 657m
Dreiecke:
D 2 (   0|   0) (-106|   0) ( -61|  58)
D26 (   0|   0) ( -71|  67) (   5|  81)
D 3 (   0|   0) (   5|  89) (  46|  66)
D11 (   0|   0) (  49|  71) (  49|  22)
D28 (   0|   0) (  54|  24) (  31|   0)
D27 (  46|   0) (  60|  42) (  73|   0)
D 8 (  74|   0) (  60|  41) (  95|   0)
D30 (  95|   0) (  70|  41) ( 128|   0)
D25 ( 128|   0) (  83|  38) ( 156|   0)
D24 ( 156|   0) (  53|  54) (  89|  89)
D18 ( 156|   0) (  96|  80) ( 149|  71)
D19 ( 156|   0) ( 147|  91) ( 182|  60)
D33 ( 156|   0) ( 182|  60) ( 218|   0)
D34 ( 218|   0) ( 114| 175) ( 178| 117)
D37 ( 218|   0) ( 179| 114) ( 206|  74)
D17 ( 218|   0) ( 203|  93) ( 227|  71)
D31 ( 218|   0) ( 228|  76) ( 253|   0)
D15 ( 253|   0) ( 245|  63) ( 291|   0)
D 4 ( 291|   0) ( 223|  95) ( 285| 103)
D16 ( 291|   0) ( 284| 105) ( 307|  92)
D 1 ( 291|   0) ( 310| 107) ( 319|  51)
D14 ( 291|   0) ( 326|  64) ( 362|   0)
D29 ( 362|   0) ( 326|  65) ( 349| 111)
D 6 ( 362|   0) ( 350|  97) ( 387|  53)
D32 ( 362|   0) ( 395|  69) ( 432|   0)
D20 ( 432|   0) ( 386|  87) ( 447|  84)
D35 ( 432|   0) ( 444|  65) ( 483|   0)
D10 ( 483|   0) ( 449|  59) ( 472| 105)
D 5 ( 483|   0) ( 473|  91) ( 510|  59)
D22 ( 483|   0) ( 517|  75) ( 551|   0)
D12 ( 551|   0) ( 513|  83) ( 569|  74)
D36 ( 551|   0) ( 569|  70) ( 595|   0)
D23 ( 595|   0) ( 560|  99) ( 617|  65)
D 9 ( 595|   0) ( 620|  66) ( 657|   0)
D21 ( 657|   0) ( 620|  66) ( 659| 113)
D 7 ( 657|   0) ( 658|  90) ( 706| 100)
D13 ( 657|   0) ( 698|  75) ( 718|   0)
```

4. Quellcode

Hier finden sich Informationen zu den einzelnen Skripten. Abschnitt 4.1 gibt einen Überblick über alle Python-Dateien, im Folgenden finden sich Informationen zu den einzelnen Klassen und Funktionen bzw. Methoden. Alle Funktionen bzw. Methoden, die nicht direkt zur Lösung beitragen, werden nicht genannt. Das betrifft vor allem das Einlesen der Daten. Außerdem werden kleinere Funktionen bzw. Methoden ausgelassen, aber ihre Funktionsweise wird durch einen Kommentar in den Quellcode-Auszügen jeweils erläutert. Auch die Konstruktoren der Klassen werden meistens nur kurz beschrieben.
Die Kommentare zum Quellcode in dieser Dokumentation sind ausführlicher als in den Python-Skripten und zusätzlich auf Deutsch statt auf Englisch. Auch in Quellcode-Auszügen werden unwichtige Teile (Debug-Ausgabe, …) ausgelassen.

Anmerkung: In Python ist es möglich, den Typ der Parameter und des Rückgabewertes von Funktionen durch sogenannte Annotationen anzugeben. Dies geschieht mit Doppelpunkten hinter den Parametern und Pfeilen (`->`) vor dem abschließenden Doppelpunkt. Vom Interpreter werden sie jedoch ignoriert und haben keinen Einfluss auf das Programm. Annotationen werden im Programm verwendet, damit z.B. IDEs besser mit Funktionsparametern umgehen können, hier werden sie weggelassen.

4.1 Bedeutung der Ordner und Skripte

```
python3 Aufgabe2/aufgabe2-beispieldaten.py
```

Nutzt das Modul `trianglearranger`, um für die fünf Beispieldaten möglichst gute Gesamtabstände zu berechnen. Der Quellcode ist nicht weiter interessant und wird hier nicht besprochen. Zu beachten ist, dass beide Algorithmen (unverändert und `smallest`-Variante) verwendet werden, wodurch die Laufzeit steigt.

```
Aufgabe2/trianglearranger.py
```

Modul, das alle Funktionen enthält, um die Dreiecke anzuordnen.

```
Aufgabe2/geometry.py
```

Stellt alle geometrischen Funktionalitäten (Klassen für die Dreiecke, eine Klasse für die Gruppe, speichern als SVG-Grafik, …) bereit.

```
Aufgabe2/site-packages/
```

Ordner enthält die erforderlichen Drittanbieter-Module: `svgwrite` und `pyparsing` (`pyparsing` wird von `svgwrite` benötigt).

4.2 Modul geometry

```
def save_svg(path, groups, border_width=5, labels=True,
relevant_lines=False)
```

Diese Funktion ist für die grafische Speicherung der Gruppen `groups` im Pfad `path` im SVG-Format zuständig. Verwendet wird das Modul `svgwrite` wie in Abschnitt 2.4 beschrieben. Die Parameter `border_width`, `labels` und `relevant_lines` bestimmen einen Rand, der an den Seiten der SVG-

Grafik eingefügt wird, ob die Dreiecke mit ihren IDs beschriftet werden sollen und ob die relevanten Kanten/Punkte einer Gruppe hinzugefügt werden sollen.

4.2.1 Klasse geometry.Point

Die Klasse bekommt einen Punkt (x, y) übergeben, der in entsprechenden Attributen gespeichert wird.

`__sub__(other)`

Diese sogenannte Magic-Method sorgt dafür, dass ein `Point`-Objekt `other` von einem anderen (`self`) abgezogen werden kann (Addition wird nicht benötigt, deshalb gibt es die `__add__`-Methode nicht). Dazu werden die jeweiligen x- und y-Koordinaten subtrahiert.
Subtraktion wird von der `Point.rotate_around`-Methode benötigt.

`get_distance(other)`

Berechnet die Distanz zwischen diesem Punkt (`self`) und einem anderen (`other`) mit dem Satz des Pythagoras. Wird benötigt, um die Seitenlängen der durch Punkte definierten Dreiecke zu berechnen.

`get_angle(other)`

Berechnet den Winkel in Grad zwischen diesem Punkt (`self`) und einem anderen (`other`). Ein Winkel von 0° steht für gleiche y-Koordinate, aber größere x-Koordinate des anderen Punkts. Bei 90° ist der andere Punkt in positiver y-Richtung verschoben. Zur Berechnung wird die `math.atan2`-Funktion benutzt, die nicht wie eine normale Arkustangens-Funktion einen, sondern zwei Parameter (x- und y-Differenz) bekommt. Da die Funktion nun zwei Vorzeichen zur Verfügung hat, kann bestimmt werden, in welche Richtung genau der Punkt verschoben ist. Bei nur einem Parameter und positivem Vorzeichen wäre es z.B. nicht möglich zu bestimmen, ob beide Differenzen (deren Quotient übergeben werden würde) positiv oder negativ waren.

`rotate_around(origin, angle)`

Rotiert diesen Punkt (`self`) im Winkel `angle` (in Grad) um den Ursprung `origin`. Durch die Subtraktion `self-origin` wird dieser Punkt zuerst so verschoben, dass der Ursprung des Koordinatensystems nun gleichbedeutend mit `origin` ist, dann wird mit der Sinus- und Kosinus-Funktion die neue Koordinate ausgerechnet und schließlich die Verschiebung vom Anfang wieder addiert:

```
angle = math.radians(angle)
p = self-origin
self.x = p.x*math.cos(angle) - p.y*math.sin(angle) + origin.x
self.y = p.y*math.cos(angle) + p.x*math.sin(angle) + origin.y
```

4.2.2 Klasse geometry.Triangle

Diese Klasse speichert die Größen eines Dreiecks und seine ID. Zu den verschiedenen Attributen siehe Abschnitt 2.1 „Einlesen und Darstellung der Daten“. Für die Berechnung der Winkel wird für die ersten zwei der Kosinussatz verwendet und der letzte Winkel ist so groß wie die Differenz von 180° und den zwei anderen Winkeln. Der Kosinussatz lässt sich folgendermaßen darstellen; es wird von einem Dreieck ausgegangen, dessen Seiten nach dem jeweils gegenüberliegenden Winkel benannt sind:

$$\alpha = \cos^{-1}\frac{b^2 + c^2 - a^2}{2bc}$$

`get_origin_angle(placement)`

Berechnet den Winkel, der mit einer bestimmten Platzierungsmethode `placement` am Ursprung der Gruppe anliegen würde. `get_origin_angle` ist für die Basisdreiecke relevant, da alle anderen Dreiecke mit ihrem kleinsten Winkel am Ursprung platziert werden.

4.2.3 Klasse geometry.PlacedTriangle

Diese Klasse stellt ein platziertes Dreieck dar. Für die Attribute siehe Abschnitt 2.2.2.

`@staticmethod`
`place_triangle(triangle, x_pos, angle, direction="counter-clockwise", placement="lsb")`

Diese Methode erstellt ein `PlacedTriangle`-Objekt mit den übergebenen Parametern und gibt es zurück. Als `triangle` sollte ein `Triangle`-Objekt übergeben werden und als `x_pos` der Punkt, an dem das neue Dreieck den Ursprung berühren soll. Die x-Position ist hier im Programm immer 0, denn die Gruppen werden ja erst am Punkt $(0|0)$ aufgebaut und später zur richtigen Position verschoben, aber um die Methode möglichst offen für Anwendungsmöglichkeiten zu halten, wurde dieser Weg gewählt. `angle` gibt den Winkel an, in dem die Unterkante (später `PlacedTriangle.lower_arm_length` bzw. `.lower_arm_point`) später zur x-Achse steht. Die Richtung `direction` von der Gruppe des Dreiecks bestimmt dabei, ob dieser Winkel von der linken Seite des Ursprungs (`clockwise`) oder von der rechten Seite des Ursprungs (`counter-clockwise`) definiert ist.

`set_x(new_x_pos)`

Verschiebt das Dreieck zur Position `new_x_pos` auf der x-Achse, indem zu allen x-Koordinaten der drei Punkte die Different auf `new_x_pos` und der aktuellen x-Koordinate des Ursprungs (`PlacedTriangle.origin.x`) addiert wird.

4.2.4 Klasse geometry.TriangleGroup

Wie in der Umsetzung bereits ausführlich erläutert, speichert diese Klasse eine Gruppe.

`has_overlapping_point()`

Diese Methode gibt entweder `True` oder `False` zurück – je nachdem, ob ein relevanter Punkt der Gruppe die verlängerte rechte Kante des Basisdreiecks überschreitet oder nicht (siehe Abschnitt 1.5.1). Dazu wird für jeden relevanten Punkt (außer den beiden Eckpunkten der rechten Kante des Basisdreiecks) das Kreuzprodukt berechnet. Ist dieses kleiner als 0, überschreitet der Punkt die Linie nicht. Ist es gleich 0, liegt der Punkt auf der Linie. Nur wenn das Kreuzprodukt größer als 0 ist, überschreitet der Punkt die Linie. Die folgende Zeile wird benutzt, um das Kreuzprodukt zu berechnen. `line_point1` ist der Punkt der Kante, der die x-Achse berührt, und `line_point2` der andere Punkt der Kante. `point` ist der relevante Punkt der Gruppe, der gerade überprüft wird.

```
d = (point.x - line_point1.x) * (line_point2.y - line_point1.y) - \
    (point.y - line_point1.y) * (line_point2.x - line_point1.x)
```

Es ist zu beachten, dass bei Vertauschung von `line_point1` und `line_point2` statt einem positiven Kreuzprodukt ein negatives Kreuzprodukt bedeuten würde, dass ein Punkt überlappt.

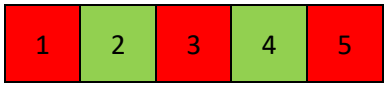
`get_sorted_triangles()`

Hier werden die exakte Reihenfolge und die jeweilige Platzierungsmethode der Dreiecke für die Gruppe berechnet. Mit verschiedenen `if`-Bedingungen wird bestimmt, wie die Dreiecke sortiert und platziert werden müssen. Zuerst gilt es, herauszufinden, ob eine Gruppe die erste oder letzte Gruppe ist. Wenn es die erste Gruppe ist, dann werden die Dreiecke von klein (am Basisdreieck) zu groß sortiert. Wenn es die letzte Gruppe ist, dann werden die Dreiecke von groß nach klein sortiert (siehe auch Abschnitt 1.5.1). Außerdem werden die Dreiecke von groß nach klein sortiert, wenn die Richtung der Gruppe `clockwise` ist (das kommt nur vor, wenn die Summe der kleinsten Winkel kleiner als 360° ist, siehe Abschnitt 1.2), obwohl es sich eigentlich um die erste Gruppe handelt. Hier ist aber wichtig, dass die Gruppe ja im Uhrzeigersinn läuft und somit links an der x-Achse die größten Dreiecke platziert werden.

Wenn die Gruppe nicht die erste oder letzte ist, gibt es zwei Möglichkeiten: Entweder, die Dreiecke werden ganz gewöhnlich von klein nach groß sortiert, oder das kleinste Dreieck kommt zuerst, aber als zweites das drittkleinste, weil das zweitkleinste zum Schluss platziert wird (siehe wieder Abschnitt 1.5.1). Letzterer Fall tritt nur ein, wenn ein Punkt der Gruppe zuvor überlappt oder der Winkel rechts am Basisdreieck der vorherigen Gruppe addiert mit dem Winkel dieser Gruppe größer als 180° ist. Diese Bedingung lässt sich folgendermaßen formulieren:

```
self.pre_group.base_triangle_outer_angle+self.angle > 180 or
self.pre_group.has_overlapping_point()
```

Sollte die Bedingung zutreffen, werden die Dreiecke zuerst ganz „normal“ nach der Länge der mittellangen Seite sortiert. Hier stellt die folgende Grafik eine Liste der sortierten Dreiecke dar. Das Dreieck an Position 1 hat die kürzeste mittellange Seite und das Dreieck an Position 5 die längste. In der Gruppe werden dann zuerst die Dreiecke mit den Positionen 1, 3 und 5 platziert, also die rot gefärbten. Die Dreiecke an den Positionen 2 und 4 kommen in umgekehrter Reihenfolge hinzu, sodass sich eine Reihenfolge von {1, 3, 5, 4, 2} ergibt, also kleinstes, drittkleinstes, fünftkleinstes, viertkleinstes und zweitkleinstes Dreieck.



In Python können alle roten Dreiecke mit `[::2]` hinter einer Liste (jedes zweite Element) ausgewählt werden. Bei der Auswahl der grün gefärbten Dreiecke kommt es darauf an, ob es insgesamt eine gerade oder ungerade Anzahl von Dreiecken ist. Hier ist die Zahl ungerade, also muss beim vorletzten Element begonnen werden: `[-2::-2]`. Wäre die Anzahl gerade, dann käme zum Schluss ein grün gefärbtes Dreieck und der Zugriff müsste `[-1::-2]` lauten.

Die Methode gibt eine Liste zurück, die für jedes Dreieck ein Tupel mit dem Dreieck und seiner Platzierungsmethode enthält.

`place(pre_group, next_group)`

Diese Methode ist für die Gruppe am interessantesten, denn hier werden die Dreiecke grafisch platziert. Die übergebenen Gruppen stehen für die bereits platzierte, vorherige Gruppe und die noch nicht platzierte, nachfolgende Gruppe, die beide in entsprechenden Attributen gespeichert werden. Sie sind `None`, wenn es keine vorherige bzw. nachfolgende Gruppe gibt. Falls es ein Basisdreieck gibt, wird dies zuerst folgendermaßen platziert:

```
angle = self._place_triangle(self.base_triangle, self.color_base_triangle, 0,
self.placement_base_triangle)
```

Die Methode `TriangleGroup._place_triangle` macht nichts anderes als ein übergebenes Dreieck in einer bestimmten Farbe und in einem bestimmten Winkel mit einer bestimmten Platzierungsmethode zu platzieren. `self.placement_base_triangle` wurde als Parameter dem Konstruktor übergeben. Außerdem werden in der Methode die neuen Punkte und Außenkanten (die Kanten, die nicht den Ursprung berühren) den Listen `self.points` und `self.outer_lines` hinzugefügt und die Liste `self.placed_triangle` erhält ein zusätzliches Element mit dem soeben platzierten Dreieck. Die Liste `self.outer_lines` enthält immer Tupel, die eine Kante aus zwei Punkte definieren. Diese Punkte sind so sortiert, dass sich der Punkt mit der kleineren y-Koordinate zuerst im Tupel befindet. Warum das so ist, wird später erklärt. Die Methode gibt den Winkel zurück, den das neue Dreieck in der Gruppe benötigt. Die Variable `angle` zählt den Winkel mit, der bis jetzt platziert wurde. Immer, wenn ein neues Dreieck platziert wird, bekommt `_place_triangle` also `angle` als Winkel übergeben und der zurückgegebene Winkel wird zu `angle` addiert.
Die Rotation der Dreiecke, wenn noch etwas Platz zur vorherigen Gruppe vorhanden sein sollte, erfolgt auch mit der `angle`-Variablen. Dieser wird nach der Platzierung des Basisdreiecks einfach ein bestimmter Winkel hinzugefügt:

```
if self.pre_group:
    pre_group_outer_angle = self.pre_group.outer_angle
else:
    # es gibt keine vorherige Gruppe, also ist der Winkel der vorherigen Gruppe 0
    pre_group_outer_angle = 0
# Platz, der der Gruppe noch zur Verfügung steht
space = 180 - pre_group_outer_angle - self.angle
if space > 0:
    # wenn der Winkel größer als 0 ist, ist noch Platz vorhanden
    angle += space
```

Die Variable `self.angle`, die den Winkel der Gruppe speichert, ist dabei nicht mit der lokalen Variablen `angle` zu verwechseln.

Im nächsten Schritt werden alle Dreiecke mit der `TriangleGroup.get_sorted_triangles`-Methode sortiert und dann mit der `TriangleGroup._place_triangle`-Methode nacheinander platziert. Danach folgt der schwierigste Teil der Methode: Die neue x-Position für die Gruppe wird ausgerechnet. Dafür existiert die lokale Funktion (in Python können Funktionen auch innerhalb einer Funktion/Methode definiert werden, sie sind dann jeweils nur innerhalb der Funktion erreichbar) `get_x`, die eine Punkt/Kante-Kombination bestehend aus `vertex` (Punkt) und `edge_bottom` und `edge_top` (Kante) übergeben bekommt. Hier wird auch deutlich, warum die Punkte der Außenkanten sortiert werden: Die Funktion benötigt den tieferliegenden Punkt (`edge_bottom`) mit der kleineren y-Koordinate und den größeren (`edge_top`). Der Parameter `own_vertex` gibt an, ob der übergebene Punkt zur eigenen Gruppe oder zur vorherigen Gruppe gehört.

```
def get_x(vertex, edge_bottom, edge_top, own_vertex=True):
    # Prüfe, ob der Punkt die Kante überhaupt berühren kann
    if edge_bottom.y <= vertex.y <= edge_top.y:
        if math.isclose(edge_bottom.y, edge_top.y, abs_tol=0.1):
            # die Kante liegt (fast) horizontal, also kollidiert der Punkt mit einem Ende
            # der Kante
            if not own_vertex:
                # es ist die eigene Kante, also kollidiert der linke Punkt mit kleinerer
                # x-Koordinate
```

```
                return min(edge_bottom.x, edge_top.x) - vertex.x
            else:
                # es ist die andere Kante, also kollidiert der linke Punkt mit größerer
                # x-Koordinate
                return max(edge_bottom.x, edge_top.x) - vertex.x
        else:
            # intersection_x ist die x-Koordinate der Kollision
            intersection_x = (
                        (vertex.y - edge_bottom.y) *
                        ((edge_top.x - edge_bottom.x) / (edge_top.y - edge_bottom.y)) +
                        edge_bottom.x
                        )
            if own_vertex:
                return intersection_x - vertex.x
            else:
                return vertex.x - intersection_x
    return None
```

Die x-Koordinate der Kollision wird berechnet, indem die Höhe des Punktes relativ zur kleinsten y-Koordinate der Kante (`vertex.y – edge_bottom.y`) mit der Steigung der Kante multipliziert wird. Da der x- und nicht wie gewöhnlich der y-Wert berechnet werden soll, wird die Steigung nicht wie gewöhnlich mit dem Quotienten aus Höhen- und Seitendifferenz, sondern aus Seiten- und Höhendifferenz berechnet. Da das Produkt aus Steigung und Höhe des Punktes relativ zur Kantenunterseite nur relativ zur x-Koordinate des unteren Punktes der Kante berechnet wird, wird `edge_bottom.x` addiert. Außerdem sollte beachtet werden, dass die negative x-Koordinate des Punktes (`-vertex.x`) den Abstand vom Punkt zum Gruppenursprung angibt, weil die Gruppe mit dem Ursprung am Punkt $(0|0)$ platziert wurde und somit alle Punkte, die links vom Ursprung platziert wurden, eine negative x-Koordinate haben. Der Rückgabewert ist abhängig davon, ob der Punkt zur eigenen Gruppe oder zur Gruppe davor gehört. Gehört er zur eigenen Gruppe, ist die neue Position des Ursprungs dieser Gruppe (für dieses Punkt/Kante-Paar) die Summe aus der x-Position der Kollision und des Abstands zum Gruppenursprung vom Punkt, also `-vertex.x`. Andernfalls ist es genau umgekehrt und die x-Position der Kollision muss von der x-Koordinate des Punktes abgezogen werden. Das liegt daran, dass `intersection_x` immer relativ zur Kante berechnet wird. Wenn die Kante zur neuen Gruppe gehört, dann liegt die x-Position der Kollision auch irgendwo auf dieser Kante, aber die neue Gruppe ist ja im Moment noch bei $(0|0)$ platziert.

Zurück zur Berechnung der neuen x-Koordinate für den Ursprung: Alle Punkt/Kante-Kombinationen werden nacheinander durchgegangen. Dafür gibt es drei Schleifen-Konstrukte:

```
max_x = -math.inf
# Gehe alle eigenen Punkte mit den relevanten Punkten der vorherigen Gruppe durch
# Kombinationen zwischen einem Punkt und einer Kante, die den Ursprung berührt
for own in self.points:
    for other, other_origin in self.pre_group.relevant_points:
        # eigener Punkt - andere kante
        new_x = get_x(own, other_origin, other, True)
        if new_x is not None and new_x > max_x:
            max_x = new_x
        # anderer Punkt und eigene Kante (eigener Ursprung ist noch (0|0))
        new_x = get_x(other, Point(0, 0), own, False)
        if new_x is not None and new_x > max_x:
            max_x = new_x
# Kombinationen zwischen eigenem Punkt und anderer Kante
for own in self.points:
    for other_line, _ in self.pre_group.relevant_lines:
        new_x = get_x(own, other_line[0], other_line[1], True)
        if new_x is not None and new_x > max_x:
```

```
            max_x = new_x
# Kombinationen zwischen eigener Kante und anderem Punkt
for own_line in self.outer_lines:
    for other, _ in self.pre_group.relevant_points:
        new_x = get_x(other, own_line[0], own_line[1], False)
        if new_x is not None and new_x > max_x:
            max_x = new_x
self.set_x(max_x)
```

Es ist jeweils zu beachten, dass die relevanten Punkte und Kanten immer in Tupeln gespeichert werden, von denen das erste Element den eigentlichen Punkt bzw. die eigentliche Kante enthält und das zweite den Ursprung der Gruppe, zu der sie gehören. So ist es möglich, den Ursprung auch zu kennen, wenn die Punkte oder Kanten aus einer Gruppe stammen, die weiter links liegt. Wenn von der `get_x`-Funktion eine größere x-Koordinate für den Ursprung gefunden wurde, dann wird `max_x` dementsprechend verändert. Anschließend wird mit der Methode `set_x` die x-Koordinate verändert. Die Methode passt die x-Koordinate des Ursprungs und die Punkte aller platzierten Dreiecke an die neue x-Position an.

4.3 Modul trianglearranger

`TRIANGLE_SUBSET_FINDER_TOLERANCE = 0.2`
Legt die Toleranz für das Teilsummenproblem (`trianglearranger.get_triangle_sum_subset`) fest (siehe dort).

`BASE_TRIANGLES_ANGLE_TOLERANCE_FACTOR = 1.2`
Legt einen Faktor fest, mit dem bei der Auswahl neuer Basisdreiecke (`trianglearranger.base_triangles_generator`) der zur Verfügung stehende Winkel multipliziert wird, bevor er mit dem Winkel der übrigen Dreiecke verglichen wird (siehe dort).

`def search_arrangement(triangles)`
Diese Funktion sucht nach passenden Anordnungen für die Dreiecke, die im Parameter `triangles` als Liste übergeben wurden. Dafür werden verschiedene Auswahlen für die Basisdreiecke (`base_triangles_generator`) so lange mit `arrange_triangles` und `arrange_triangles_smallest` angeordnet, bis das bisher beste Ergebnis kleiner ist als die Länge der neuen Basisdreiecke.
Da diese Funktion für den eigentlichen Algorithmus irrelevant ist, ist hier nur kurz die genauere Funktionsweise beschrieben: Sollte die Summe der kleinsten Winkel aller Dreiecke kleiner gleich 360° sein, wird die Funktion `arrange_triangles_lt360` (siehe dort) aufgerufen, um die Dreiecke in zwei Gruppen anzuordnen (siehe Abschnitt 1.2 „Platzierung der Dreiecke für Summe der kleinsten Winkel bis 360°“). Die resultierende Gesamtlänge sowie die Gruppen werden als bisher bestes Ergebnis gespeichert. Sollte die Summe größer sein, wird die bisher beste Gesamtlänge auf unendlich (`math.inf`) gesetzt, sodass auf jeden Fall ein Ergebnis gefunden wird, was einen niedrigeren Gesamtabstand besitzt.

`def arrange_triangles_lt360(triangles, angle_sum)`
Diese Funktion ist dafür zuständig, die Dreiecke anzuordnen, wenn die Summe der kleinsten Winkel nicht mehr als 360° beträgt. Dafür werden mit `get_triangle_sum_subset` Dreiecke gesucht, die genau in die Hälfte von `angel_sum` passen. Diese Dreiecke werden in der einen Gruppe, die Dreiecke, die nicht in dem Ergebnis vorhanden sind, in der anderen Gruppe untergebracht. Die beiden Gruppen werden gewöhnlich platziert.

`def get_triangle_sum_subset(triangles, required_sum)`

Diese Funktion stellt eine Lösung für das Teilsummenproblem bereit. Aus den Dreiecken `triangles` werden Dreiecke herausgesucht, deren Summe der kleinsten Winkel möglichst nah an `required_sum` liegt, diese aber nicht überschreitet. Es wird ein Tupel zurückgegeben, das die gefundene Summe und die gefundenen Dreiecke enthält. Für weitere Details siehe Abschnitt 1.6. Die Funktion sieht folgendermaßen aus:

```
if required_sum == 0 or not triangles:
    # es gibt keine Dreiecke oder die Summe ist sowieso 0, also ist das Ergebnis leer
    return 0, ()
s = sum(t.shortest_line_angle for t in triangles)  # die Summe aller kleinsten Winkel
if s < required_sum:
    # Alle Dreiecke zusammen können überhaupt nicht größer als required_sum werden
    return s, triangles

smallest_angle = min(triangles, key=lambda t: t.shortest_line_angle).shortest_line_angle
greatest_angle = max(triangles, key=lambda t: t.shortest_line_angle).shortest_line_angle
if smallest_angle > required_sum:
    # jedes Dreieck ist größer als die benötigte Summe
    return required_sum, ()
best_sum = -math.inf  # beste bis jetzt gefundene Summe
best_values = None  # zugehörige Dreiecke

# minimale Anzahl für die Dreiecke
min_triangle_count = math.floor(required_sum / greatest_angle)
if min_triangle_count < 1:
    # durch das Abrunden könnte die minimale Anzahl 0 werden
    min_triangle_count = 1

# maximale Anzahl für die Dreiecke
max_triangle_count = min(len(triangles), math.ceil(required_sum / smallest_angle))
if max_triangle_count < min_triangle_count:
    # die maximale Anzahl der Dreiecke darf nicht kleiner als die minimale Anzahl sein
    max_triangle_count = min_triangle_count

# Gehe Dreiecke für jede Anzahl durch
# 'i' steht für die Anzahl an Dreiecken
for i in range(min_triangle_count, max_triangle_count+1):
    for values in itertools.combinations(triangles, i):
        s = sum(t.shortest_line_angle for t in values)
        if best_sum <= s <= required_sum:
            # Summe 's' ist besser
            best_sum = s
            best_values = values
            if math.isclose(s, required_sum, abs_tol=TRIANGLE_SUBSET_FINDER_TOLERANCE):
                # (Toleranz ist normalerweise 0.2)
                return best_sum, values
return best_sum, best_values
```

`def base_triangles_generator(triangles, angle_sum)`

Diese Funktion generiert nach und nach die verschiedenen Auswahlen von Basisdreiecken aus den Dreiecken `triangles` mit der Summe der kleinsten Winkel `angle_sum`. Jedes Kriterium wird durch die lokal definierte Klasse `SortMethod` repräsentiert, deren Konstruktor die Liste von Dreiecken sowie eine Funktion übergeben bekommt, die die Dreiecke bewertet. Mit dieser Funktion wird die Dreiecksliste sortiert und als Attribut gespeichert. Das Attribut `count` speichert die derzeitige Anzahl an Basisdreiecken, die diesem Kriterium zurzeit zugeordnet sind. Mit jeder Auswahl des Kriteriums wird das `count`-Attribut um 1 erhöht (siehe den folgenden Quellcode). Außerdem hat jede Klasse die Attribute `available_angle` und `remaining_angle`, die jeweils den zur Verfügung stehenden Winkel

und den Winkel der übrigen Dreiecke speichern. Mit der Methode `get_next_combination` kann man eine Liste von Basisdreiecken, den übrigen Dreiecken und der Länge erhalten. Die derzeitige Länge wird im Attribut `next_distance` gespeichert. Damit aus den Kriterien das Kriterium mit der kleinsten Länge einfach herausgesucht werden kann, implementiert die Klasse zusätzlich die Magic-Methods `__lt__` und `__gt__` zum Vergleichen. Beide Funktionen vergleichen jeweils den Wert von `next_distance`. Hier ist der Quellcode der Funktion:

```
# definiere Funktionen für die Kriterien
sort_methods = (lambda t: t.shortest_line, lambda t: t.longest_line, lambda t: -
t.shortest_line_angle)
methods = tuple(SortMethod(triangles, sort_key) for sort_key in sort_methods)
last_distance = -1  # die Länge des zuletzt verwendeten Kriteriums
while True:
    method = min(methods)  # Methode mit der kleinsten Länge
    if method.next_distance == math.inf:
        # sobald ein Kriterium alle Dreiecke abgearbeitet hat, wird die Länge auf inf
        # gesetzt; wenn sogar das Kriterium mit der kleinsten Länge inf als Länge hat,
        # haben alle Kriterien die Länge inf und es gibt für kein Kriterium neue Dreiecke
        break
    if method.remaining_angle < method.available_angle * \
         BASE_TRIANGLES_ANGLE_TOLERANCE_FACTOR and method.next_distance > last_distance:
        # die Dreiecke, die dem Kriterium zugeordnet sind, bieten genug Platz und die
        # Länge ist nicht mit der letzten identisch, d.h. es ist wirklich eine neue
        # Auswahl von Basisdreiecken
        last_distance = method.next_distance
        yield method.get_next_combination()
    try:
        # Füge das nächste Dreieck (method.triangles[method.count]) hinzu
        # das zuletzt hinzugefügte Dreieck wird nicht mitgezählt, d.h. -1
        method.next_distance += method.triangles[method.count - 1].shortest_line
        # neues Dreieck verbraucht keinen Platz mehr, ...
        method.remaining_angle -= method.triangles[method.count].shortest_line_angle
        # ... stellt aber neuen Platz zur Verfügung
        method.available_angle += 180 - \
                                  method.triangles[method.count].middle_line_angle - \
                                  method.triangles[method.count].longest_line_angle
        method.count += 1
    except IndexError:
        # IndexError weil method.count zu groß ist, es sind keine neuen Dreiecke mehr
        # vorhanden
        method.next_distance = math.inf
```

`def arrange_triangles(base_triangles, triangles)`

Alle übergebenen Dreiecke mit Basisdreiecken werden so, wie es in der Lösungsidee beschrieben wurde, angeordnet. Dies ist die wichtigste Funktion des Algorithmus.

Wie in Abschnitt 1.4 beschrieben, wird jedes der Basisdreiecke einmal als Basisdreieck der ersten Gruppe ausprobiert. Außerdem gibt es für das erste Basisdreieck die zwei Platzierungsmethoden `bsl` und `bls`. Die Auswahl des ersten Basisdreiecks sowie dessen Platzierung wird mit zwei `for`-Schleifen implementiert. Hier ist die Funktion:

```
best_distance = math.inf  # der beste bis jetzt gefundene Gesamtabstand
best_groups = None  # die zum besten Gesamtabstand gehöhrenden Gruppen als Liste
for first_base_triangle in base_triangles:  # jedes Basisdreieck einmal als Basisdreieck
                                            # der ersten Gruppe ausprobieren
    for placement_base_triangle in ("bsl", "bls"):  # mit beiden Platzierungsmethoden
        remaining_triangles = triangles.copy()  # alle noch nicht platzierte Dreiecke
        remaining_base_triangles = base_triangles.copy()  # alle noch nicht platzierten
```

```
                                                    # Basisdreiecke
        remaining_base_triangles.remove(first_base_triangle)

        # die erste Gruppe (ganz links)
        first_group = TriangleGroup(first_base_triangle,
                                    placement_base_triangle=placement_base_triangle)
        # Suche Dreiecke, die in die erste Gruppe links neben das Basisdreieck passen und
        # füge sie hinzu
        _, new_triangles = get_triangle_sum_subset(remaining_triangles, 180 -
                                                                        first_group.angle)
        first_group.extend(new_triangles)
        # Entferne die verwendeten Dreiecke
        for t in list(new_triangles):
            remaining_triangles.remove(t)

        groups = [first_group]  # alle Gruppen, bis jetzt nur die erste
        last_group_angle = first_group.base_triangle_outer_angle  # der Winkel an der
                                    # rechten Seite des Basisdreiecks der letzten Gruppe
        while remaining_base_triangles:  # solange bis es keine Basisdreiecke mehr gibt
            # Suche das beste nächste Basisdreieck
            diffs = []  # enthält die Ergebnisse für alle Basisdreiecke
            for base_triangle in remaining_base_triangles:
                # Für Platzierungsmethode 'bls'
                # 'space': Platz auf der linken Seite des Basisdreiecks
                space = 180 - last_group_angle - base_triangle.middle_line_angle
                # Suche Dreiecke für 'space'
                s, group_triangles = get_triangle_sum_subset(remaining_triangles, space)
                # ein Element in 'diffs' enthält:
                #   den Platz, der übriggelassen wird;
                #   die Platzierungsmethode für das Basisdreieck;
                #   das Basisdreieck;
                #   alle anderen Dreiecke, die den Platz auffüllen
                diffs.append((space - s, "bls", base_triangle, group_triangles))

                # Für Platzierungsmethode 'bsl'
                space = 180 - last_group_angle - base_triangle.longest_line_angle
                # Suche passende Dreiecke für 'space'
                s, group_triangles = get_triangle_sum_subset(remaining_triangles, space)
                diffs.append((space - s, "bsl", base_triangle, group_triangles))
            # Suche Basisdreieck, das den wenigsten Platz verschwendet (Element 0 des
            # Tupels möglichst klein)
            _, placement, best_base_triangle, best_group_triangles = min(diffs,
                                                                key=lambda x: x[0])
            # die neue Gruppe
            new_group = TriangleGroup(best_base_triangle,
                                                placement_base_triangle=placement)
            new_group.extend(best_group_triangles)
            groups.append(new_group)
            last_group_angle = new_group.base_triangle_outer_angle
            # Entferne verwendete Dreiecke
            for t in list(best_group_triangles):
                remaining_triangles.remove(t)
            remaining_base_triangles.remove(best_base_triangle)
        # Alle Basisdreiecke wurden benutzt, ...
        if remaining_triangles:
            # ... aber es sind normale Dreiecke übrig
            # Füge übrige Dreiecke in letzte Gruppe ohne Basisdreieck ein
            space = 180 - last_group_angle  # Platz, den die letzte Gruppe hat
            s, group_triangles = get_triangle_sum_subset(remaining_triangles, space)
            group = TriangleGroup()
            group.extend(group_triangles)
            groups.append(group)
            for t in list(group_triangles):
```

```
                remaining_triangles.remove(t)
        if remaining_triangles:
            # Es sind immer noch Dreiecke übrig
            try:
                # Füge die übrigen Dreiecke in die bereits erstellten Gruppen ein.
                # die Funktion fügt die Dreiecke Gruppen mit einem Winkel möglichst nah an
                # 90° hinzu (Abschnitt 1.4) und wird nicht weiter beschrieben
                insert_remaining_triangles(groups, remaining_triangles)
            except ValueError:
                # insert_remaining_triangles wirft einen ValueError, wenn nicht alle
                # Dreiecke in die Gruppen passen (die Gruppen haben zu wenig Platz frei).
                # Mit diesem Basisdreieck funktioniert es also nicht, also springe zur
                # nächsten Möglichkeit für das erste Basisdreieck
                continue
        # Platziere die Gruppen grafisch (Funktion wird nicht weiter erläutert, ruft für
        # jede Gruppe die place-Methode auf)
        place_groups(groups)
        # der Gesamtabstand ist die x-Koordinate des Ursprungs der letzten Gruppe, weil
        # die erste Gruppe ihren Ursprung bei (0|0) hat
        distance = groups[-1].origin.x
        if distance < best_distance:
            # der Gesamtabstand übertrifft das bisher beste Ergebnis,
            # also speichere es als bisher bestes Ergebnis
            best_distance = distance
            best_groups = groups
return best_groups, best_distance
```

`def arrange_triangles_smallest(base_triangles, triangles)`

Diese Funktion ist fast identisch mit `arrange_triangles`, es wurden lediglich einzelne Code-Segmente hinzugefügt, um für jede Gruppe möglichst ein kleines Dreieck aufzuheben. Dazu werden am Anfang so viele kleine Dreiecke (gemäß der Länge der mittellangen Seite) in einer Liste gespeichert wie es später Gruppen gibt (siehe Abschnitt 1.7). Die kleinsten Dreiecke werden mit der `sorted`-Funktion bestimmt:

```
sorted(triangles, key=lambda t: t.middle_line)[:len(base_triangles)+1]
```

Dann werden die jetzt ausgewählten Dreiecke ein weiteres Mal sortiert, und zwar nach dem kleinsten Winkel von groß nach klein und dieses Ergebnis wird in `smallest_triangles` gespeichert. Das ist nötig, damit später schnell das Dreieck mit dem größten Winkel gefunden werden kann, was in eine bestimmte Lücke passt. Aus der normalen `triangles`-Liste werden die kleinsten Dreiecke entfernt. Immer, wenn eine Lücke gefüllt werden soll, wird zuerst geprüft, ob es ein kleines Dreieck gibt, das noch in die Lücke passt. Ist das der Fall, wird dieses Dreieck auf jeden Fall hinzugefügt und es wird mit der Lösung des Teilsummenproblems ganz normal der restliche Platz aufgefüllt. Immer, wenn kein Dreieck gefunden wurde, wird das Dreieck aus `smallelst_triangles` mit dem größten kleinsten Winkel entfernt und stattdessen zu `triangles` hinzugefügt, denn ein Dreieck wird nun nicht mehr benötigt. Da für die letzte Gruppe ein Dreieck aufgehoben wurde, befindet sich in `smallelst_triangles` am Ende noch ein Dreieck, das nun zu `triangles` hinzugefügt wird. Aus `triangles` wird dann ganz gewöhnlich die letzte Gruppe gebildet.