Aufgabe 2 „Dreiecksbeziehungen“- Dokumentation

37. Bundeswettbewerb Informatik 2018/19 - 2. Runde

Lukas Rost

Teilnahme-ID: 48125

29. April 2019

Inhaltsverzeichnis

1 Lösungsidee

1.1 Mathematische Präzisierung der Aufgabenstellung

Bei der Eingabe handelt es sich um eine Menge $D = \{d_1, ..., d_n\}$ von Dreiecken d_i. Jedes Dreieck ist dabei durch seine drei Eckpunkte vollständig definiert ($d_i = \{p_1, p_2, p_3\}$). Ein Eckpunkt ist dabei wiederum ein Punkt $p_i = (x_i, y_i)$ des $\mathbb{R}^2$.

Die Aufgabenstellung fordert nun, dass eine Abbildung $D' = f(D)$ gefunden werden soll. Diese ordnet der Menge D eine Bildmenge D' zu. Für diese müssen bestimmte Bedingungen gelten:

- Für jedes $d \in D'$ gilt:
$$\forall\ (x, y) \in d : y \geq 0 \land x \geq 0 \tag{1}$$
Alle Punkte müssen also über oder auf der x-Achse sowie rechts oder auf der y-Achse liegen.

- Für jedes $d \in D'$ gilt:
$$\exists\ (x, y) \in d : y = 0 \tag{2}$$
Es muss also in jedem Dreieck mindestens einen Punkt geben, der auf der x-Achse liegt. Die Menge aller solchen Punkte eines Dreiecks sei N_i (anschaulich die Menge der Straßenecken).

- Jedes $d'_i \in D'$ muss kongruent zum entsprechenden $d_i \in D$ sein. Genauer gesagt muss d'_i aus d_i durch eine Abfolge von Kongruenzabbildungen, d.h. Translationen, Rotationen und senkrechten Achsenspiegelungen[1] hervorgehen.

- Für jedes $d \in D'$ und jedes $e \in D'$ gilt:
$$d \cap e = \emptyset \tag{3}$$
$d \cap e$ stellt dabei die Schnittfläche der beiden Dreiecke dar. Es dürfen sich also keine zwei Dreiecke überlappen.

  Eine Dreiecksanordnung wird als **erlaubt** bezeichnet, wenn sie diese Bedingungen erfüllt. Die Menge der erlaubten Dreiecksanordnungen sei dabei E.

Nun ist eine Dreiecksanordnung D' gesucht, die **optimal** ist. Eine optimale Dreiecksanordnung sei dabei folgendermaßen definiert:

- D' minimiert den folgenden Wert über alle erlaubten Dreiecksanordnungen E:
$$\max_{d_i \in D'} \min_{d_j \in D'} \min_{n \in N_i} \min_{m \in N_j} |n.x - m.x| \tag{4}$$
Der Minimums-Term bildet dabei den Abstand zwischen zwei Dreiecken als minimalen Abstand der Straßenecken, während der Maximums-Term den maximalen solchen Abstand berechnet.

Die optimale Dreiecksanordnung D' bildet die Ausgabe des Algorithmus, der $f(D)$ möglichst effizient berechnen soll.

[1] und Spiegelungen an einem Punkt, wobei man diese jedoch auch durch Rotationen um 180° erreichen kann. Demzufolge müssen sie nicht betrachtet werden.

1.2 Wahl eines geeigneten Algorithmus

Die Aufgabe ähnelt einem Packproblem aus der algorithmischen Geometrie. Bei diesen muss man Objekte (z.B. Flächen wie Dreiecke) möglichst dicht in gegebene Container (z.B. ebenfalls Flächen) packen, ohne dass sich die Objekte überlappen.[3] In der hier gegebenen Aufgabe hat man jedoch zusätzliche Nebenbedingungen, die im vorherigen Abschnitt schon erläutert worden sind. Außerdem muss nicht die eingenommene Gesamtfläche minimiert werden, sondern ein Abstand auf der x-Achse.

Leider sind jedoch fast alle Packprobleme NP-vollständig, sodass auch hier die Annahme nahe liegt, dass dies der Fall ist. Demzufolge stellt sich die Frage, wie man ein solches Problem möglichst so lösen kann, dass man ein Gleichgewicht zwischen Effizienz (d.h. Laufzeit) des Algorithmus und Optimalität der Lösung einstellt.

Dafür gibt es verschiedene Herangehensweisen:

- **Brute Force und Backtracking:** Bei Brute Force werden einfach alle möglichen Lösungen durchprobiert, während man bei Backtracking eine Lösung schrittweise aufbaut und Schritte wieder zurücknimmt, wenn sie zu keiner zulässigen Gesamtlösung mehr führen können. Beide Ansätze sind in diesem Fall nicht geeignet, da der Lösungsraum extrem groß ist, d.h. es gibt sehr viele mögliche Lösungen. Wenn man Laufzeiten wie $\mathcal{O}(n! \cdot 6^n)$ vermeiden will, die sich durch Beachtung aller Permutationen und Rotationen ergeben, sollte man diese Lösungsansätze also nicht verwenden.

- **Metaheuristiken:** Zu diesen zählt beispielsweise Simulated Annealing, bei dem man die möglichen Lösungen nach einem globalen Maximum bzw. Minimum einer Bewertungsfunktion absucht. Die Bewertungsfunktion wäre in diesem Fall der Gesamtabstand. Außerdem braucht man für Simulated Annealing eine Möglichkeit, aus einer Lösung eine Nachbarlösung zu generieren, was man in diesem Fall durch z.B. Rotationen der Dreiecke erreichen könnte. Da dies jedoch schwierig zu implementieren ist und man schlimmstenfalls genauso viele Lösungen wie bei Brute Force betrachtet, sind solche Heuristiken ebenfalls nicht geeignet. Auch kann man nicht verhindern, mögliche Lösungen doppelt zu betrachten, was für die Laufzeit ebenfalls nicht so gut ist.

- **Dynamic Programming oder Greedy-Ansätze:** DP- und Greedy-Algorithmen sind zwar meistens laufzeiteffizient, jedoch nicht immer optimal. Aus diesem Grund sind sie für eine optimale Lösung dieses Problems nicht geeignet. Beispielsweise könnte die von einem Greedy-Algorithmus getroffene Entscheidung für den besten Folgezustand, also z.B. die Platzierung eines Dreiecks, zu einem nicht optimalen Gesamtergebnis führen. Es könnte dann beispielsweise nicht mehr möglich sein, andere Dreiecke dicht an das aktuelle anzulegen, wodurch der Gesamtabstand erhöht würde.

- **Heuristiken und Approximationsalgorithmen:** Bei Heuristiken versucht man durch intelligentes Raten und zusätzliche Annahmen über die optimale Lösung zu einer guten Lösung zu gelangen. Eine speziell an das Problem angepasste Heuristik ist für dieses Problem das Mittel der Wahl. Dadurch kann man sowohl eine gute (also polynomielle oder pseudopolynomielle) Laufzeit als auch eine Lösung, die relativ nah am Optimum liegt, erreichen. Die heuristische Herangehensweise an dieses Problem wird in den folgenden Abschnitten näher beschrieben.

1.3 Beschreibung der Lösungsidee

1.3.1 Beobachtungen bezüglich einer guten Lösung

Da ein Dreieck sowohl durch seine Seitenlängen als auch durch seine Innenwinkel charakterisiert wird, scheint es sinnvoll zu sein, diese erst einmal zu berechnen. Sei nun φ_i der kleinste Innenwinkel des Dreiecks d_i. Ist die Summe $\sum_{i=1}^{n} \varphi_i < 180°$, dann ist eine optimale Lösung des Problems sehr leicht ersichtlich:

Beobachtung 1 *In diesem Fall genügt es, alle Dreiecke so zu platzieren, dass alle einen festen Punkt gemeinsam haben und sie um diesen herum halbkreisförmig angeordnet sind (siehe Abbildung). Dann ist der Gesamtabstand* 0*, da sich alle Dreiecke einen Punkt teilen. Dieser Abstand muss optimal sein, da kein geringerer Abstand als* 0 *möglich ist.*

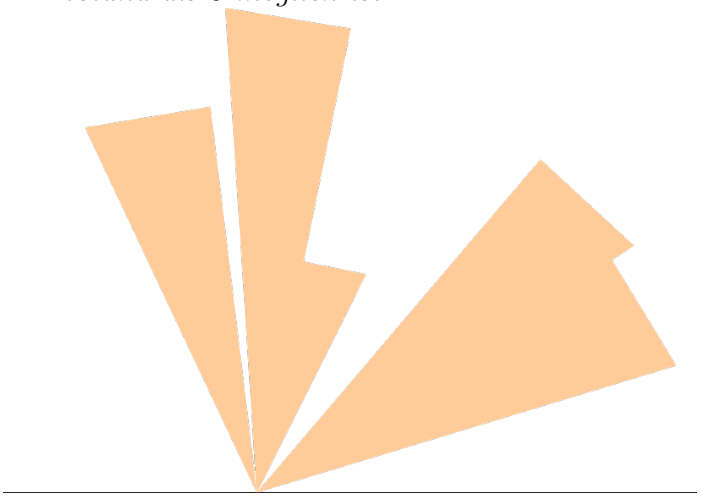

Abbildung 1: Optimale Anordnung für eine Summe kleiner 180°

Sollte die Summe jedoch größer sein, ist es nicht mehr so einfach, eine optimale Lösung zu finden. Genauer gesagt kann man ab diesem Punkt nur noch eine Heuristik einsetzen, die ein möglichst gutes Ergebnis liefert. Dabei stellt sich heraus:

Beobachtung 2 *Es scheint sinnvoll zu sein, eine Teilmenge der Dreiecke zu finden, für die* $\sum \varphi_i < 180°$ *gilt. Für diese kann die in der vorherigen Beobachtung beschriebene Strategie angewandt werden.*

Nun müssen aber noch die übriggebliebenen Dreiecke angeordnet werden.

1.3.2 Subset Sum und ein DP-Algorithmus

Um anhand der Winkel der Dreiecke die jeweils (anfangs z.B. in einem Halbkreis) zu platzierenden Dreiecke zu ermitteln, muss man das Subset-Sum-Problem lösen. Dabei ist eine Menge von ganzen Zahlen $I = \{w_1, ..., w_n\}(w_i \in \mathbb{Z})$ gegeben. Nun wird eine Teilmenge S mit maximaler Summe gesucht, die aber nicht größer als eine obere Schranke c (in diesem Fall z.B. 180° bzw. π) ist. Formal erfüllt S also folgende Eigenschaften:

$$S = \arg\max_{S \in 2^I} \sum_{w_j \in S} w_j \tag{5}$$

$$\sum_{w_j \in S} \leq c \tag{6}$$

Leider ist das Subset-Sum-Problem aber NP-vollständig und somit grundsätzlich nicht effizient lösbar. Ist c jedoch klein genug, existiert ein Dynamic-Programming-Algorithmus zur Lösung des Problems in pseudopolynomieller Zeit.[1] Dazu lässt sich eine DP-Funktion definieren, die mittels einer DP-Tabelle effizient berechnet werden kann:

$$dp(i, sum) = \begin{cases} false & sum > 0 \land i = 0 \\ true & sum = 0 \\ dp(i-1, sum) \lor dp(i-1, sum - w_i) & \text{sonst} \end{cases} \tag{7}$$

Diese Funktion gibt an, ob sich eine Summe von sum mit den ersten i Elementen der Menge erreichen lässt. Die Funktion baut darauf auf, dass es an jeder Stelle genau zwei Möglichkeiten gibt: Entweder das aktuelle Element wird nicht in das Subset aufgenommen (dann muss man die Summe mit den anderen $i-1$ Elementen erreichen) oder das Element wird in das Subset aufgenommen (dann muss man mit $i-1$ Elementen nur noch eine um w_i verringerte Summe erreichen).

Nun gibt $dp(n, c)$ an, ob es möglich ist, mit allen Elementen die Summe c zu erreichen. Doch da diese Summe oft nicht exakt erreicht werden kann, muss man c entsprechend oft dekrementieren, bis $dp(n, c) = true$ ist und es möglich ist, diese Summe zu erreichen.

Um aus der DP-Tabelle nun das eigentliche Subset zu erhalten, muss man die Lösung backtracen.[2] Dabei betrachtet man für jedes Element w_i einerseits die Möglichkeit, dass das Element enthalten ist, und andererseits, dass das Element nicht enthalten ist. Wenn eine dieser Möglichkeiten laut DP-Tabelle möglich ist, kann man rekursiv eine Lösung für das entsprechende Feld für $i-1$ generieren und dann das aktuelle Element anhängen oder nicht (je nachdem). Am Ende erhält man dann ein mögliches Subset.

Da man eine DP-Tabelle mit $n \cdot c$ Elementen ausfüllt, ergibt sich für diesen Algorithmus somit auch eine Laufzeit in $\mathcal{O}(n \cdot c)$, also in pseudopolynomieller Zeit.

Mittels dieses Subset-Sum-Algorithmus ist es möglich, Dreiecke so auszuwählen, dass ihre kleinsten Winkel φ_i einen gegebenen freien Winkel (z.B. den Halbkreis oberhalb der x-Achse) möglichst gut ausnutzen. Dadurch können die Dreiecke relativ dicht gepackt und der Gesamtabstand verkleinert werden. Entsprechend sind für diese Aufgabe die $w_i = \varphi_i$.

Da es sich bei den Winkeln in der Realität jedoch um Gleitkommazahlen handelt, müssen diese zunächst in Festkommazahlen mit wenigen Nachkommastellen umgewandelt und anschließend mit einem festen Faktor (eine entsprechende Zehnerpotenz) multipliziert werden, damit man natürliche Zahlen erhält, die vom Algorithmus verarbeitet werden können.

1.3.3 Der Algorithmus zur Platzierung der Dreiecke

1.3.4 Implementierte Verbesserungen

1.4 Optimalität des Algorithmus und Verbesserungsmöglichkeiten

1.5 Laufzeitbetrachtung und NP-Vollständigkeit

Eine Frage, die sich hierbei auch stellt, ist diejenige, ob es für dieses Problem einen in Polynomialzeit terminierenden Algorithmus geben kann, der eine optimale Lösung liefert. Dies entspricht der Frage, ob das Problem in der Klasse NPC[2] (NP-vollständig bzw. NP-complete) liegt. Ich vermute, dass dies der Fall ist, kann es jedoch nicht beweisen.

Zum Beweis, dass ein Problem in NPC liegt, werden zwei Voraussetzungen benötigt:

1. Eine deterministisch arbeitende Turingmaschine benötigt nur Polynomialzeit, um zu entscheiden, ob eine z.B. von einer Orakel-Turingmaschine vorgeschlagene Lösung tatsächlich eine Lösung des Problems ist. Dies ist hier der Fall, denn wenn eine Lösung vorgeschlagen wird, kann man in Polynomialzeit überprüfen, ob es sich dabei um eine erlaubte Dreiecksanordnung handelt.

   Dazu überprüft man alle vier Bedingungen dafür. Die ersten beiden Bedingungen lassen sich einfach für jedes Dreieck in konstanter Zeit, insgesamt also in $\mathcal{O}(n)$, überprüfen. Für die dritte Bedingung (Kongruenz) ist dies mithilfe von Kongruenzsätzen ebenfalls in linearer Zeit möglich. Bei der vierte Bedingung (keine Überlappung) muss man alle Dreieckspaare, insgesamt also $\mathcal{O}(n^2)$, auf Überlappung überprüfen. Insgesamt erhält man mit $\mathcal{O}(n^2)$ also Polynomialzeit.

2. Das Problem ist NP-schwer. Das bedeutet, dass alle anderen NP-schweren Probleme auf dieses Problem in Polynomialzeit zurückgeführt werden können. Es ist also eine Polynomialzeitreduktion notwendig. Dabei ist ein Problem aus NPC als Ausgangsproblem nötig, wie z.B. 3-Satisfiability. Eine solche Reduktion zu vollziehen, ist mir jedoch nicht möglich.[3]

Literatur

[1] GeeksforGeeks-Artikel zur DP-Lösung von Subset Sum, `https://www.geeksforgeeks.org/subset-sum-problem-dp-25/`

[2] GeeksforGeeks-Artikel zum Backtracen bei der DP-Lösung, `https://www.geeksforgeeks.org/perfect-sum-problem-print-subsets-given-sum/`

[3] Wikipedia-Artikel zu Packproblemen, `https://en.wikipedia.org/wiki/Packing_problems`

[4] Wikipedia-Artikel zu Drehmatrizen, `https://de.wikipedia.org/wiki/Drehmatrix` und Stack-Overflow-Antwort zur Umsetzung in C++, `https://stackoverflow.com/questions/2259476/rotating-a-point-about-another-point-2d`

[2] Genaugenommen ist diese Klasse nur für Entscheidungsprobleme definiert, daher handelt es sich bei diesem Suchproblem um NP-Äquivalenz.

[3] Auch wenn eine Beziehung zwischen *3*-SAT und *Drei*ecken natürlich naheliegt.

2 Umsetzung

2.1 Allgemeine Hinweise zur Benutzung

Das Programm wurde in C++ implementiert und benötigt bis auf die *Standard Library* (STL) und die beigelegte `argparse`-Library[4],die für die Verarbeitung der Konsolenargumente zuständig ist, keine weiteren Bibliotheken. Es wurde unter Linux kompiliert und getestet; auf anderen Betriebssystemen müsste mit G++ erneut kompiliert werden.

Die Eingabe und Ausgabe des Programms erfolgt in Dateien, die mithilfe der Konsolenparameter frei gewählt werden können. Dafür gibt es folgende Parameter:

```
Usage: ./main --input INPUT --svg SVG --output OUTPUT
```

2.2 Struktur des Programms und Implementierung der Algorithmen

2.2.1 Die Datei `main.cpp`

`main.cpp` enthält ausschließlich Funktionen, die für Eingabe und Ausgabe des Programms zuständig sind. Aus diesem Grund wird der Quellcode dieser Datei auch nicht mit abgedruckt.

2.2.2 Die Datei `triangles.cpp`

2.2.3 Die Datei `triangleAlgorithm.cpp`

3 Beispiele

3.1 Beispiel 1

Abbildung 2: Die Dreiecksanordnung für das Beispiel 1

Ausgabe für Beispiel 1

```
1 Gesamtabstand: 0.000 Meter
2 Platzierung der Dreiecke:
3 D1 300.000 0.000 228.640 123.777 157.126 0.133
4 D2 300.000 0.000 371.476 123.710 228.640 123.777
5 D3 300.000 0.000 442.874 0.000 371.476 123.710
6 D4 405.000 0.000 543.000 37.000 442.000 138.000
7 D5 540.000 0.000 678.000 37.000 577.000 138.000
```

[4] https://github.com/hbristow/argparse

3.2 Beispiel 2

Abbildung 3: Die Dreiecksanordnung für das Beispiel 2

Ausgabe für Beispiel 2

```
Gesamtabstand: 0.000 Meter
Platzierung der Dreiecke:
D1 300.000 0.000 242.781 472.641 390.957 449.323
D2 300.000 0.000 413.676 561.553 550.107 499.210
D3 300.000 0.000 428.977 257.437 540.957 157.634
D4 300.000 0.000 732.130 282.700 809.326 154.089
D5 300.000 0.000 779.917 145.192 817.592 0.000
```

3.3 Beispiel 3

Abbildung 4: Die Dreiecksanordnung für das Beispiel 3

Ausgabe für Beispiel 3

```
Gesamtabstand: 0.000 Meter
Platzierung der Dreiecke:
D1 119.997 60.819 99.511 0.187 300.000 0.000
D2 34.000 110.000 189.000 0.000 51.000 220.000
D3 215.000 0.000 214.000 160.000 159.000 172.000
D4 44.415 86.357 300.000 0.000 106.923 129.040
D5 270.000 0.000 335.000 190.000 335.000 0.000
D6 300.000 0.000 187.787 167.028 141.349 106.032
D7 451.000 0.000 377.000 112.000 483.000 56.000
D8 315.299 91.142 261.392 57.467 300.000 0.000
D9 300.000 0.000 325.679 152.976 405.928 228.375
D10 300.000 0.000 402.613 66.899 343.582 93.961
D11 499.284 55.997 482.003 118.658 300.000 0.000
D12 594.382 -0.000 300.000 0.000 493.014 54.236
```

3.4 Beispiel 4

Abbildung 5: Die Dreiecksanordnung für das Beispiel 4

Ausgabe für Beispiel 4

```
Gesamtabstand: 0.000 Meter
Platzierung der Dreiecke:
D1 200.015 35.042 235.000 0.027 300.000 0.000
D2 111.000 171.000 81.000 206.000 11.000 206.000
D3 10.000 38.000 46.000 171.000 11.000 206.000
D4 10.000 38.000 46.000 0.000 46.000 171.000
D5 213.250 30.404 271.827 58.577 300.000 0.000
D6 250.279 160.477 238.703 127.447 300.000 0.000
D7 111.000 0.000 146.000 36.000 46.000 171.000
D8 46.000 0.000 111.000 0.000 46.000 171.000
D9 177.000 196.000 211.000 0.000 217.000 59.000
D10 291.575 190.864 300.000 0.000 243.118 183.590
D11 291.243 198.402 300.000 0.000 311.139 142.285
D12 315.219 194.405 300.000 0.000 360.082 190.893
D13 397.000 195.000 397.000 0.000 352.000 0.000
D14 300.000 0.000 359.529 189.136 372.779 148.082
D15 394.918 136.186 300.000 0.000 387.459 177.952
D16 380.864 116.022 398.443 17.579 300.000 0.000
D17 300.000 0.000 496.000 35.000 460.000 -0.000
D18 650.000 196.000 685.000 166.000 685.000 0.000
D19 780.000 166.000 650.000 196.000 685.000 166.000
D20 685.000 114.000 780.000 84.000 685.000 84.000
```

```
D21 561.000 196.000 561.000 101.000 596.000 66.000
D22 561.000 196.000 596.000 161.000 596.000 66.000
D23 596.000 66.000 596.000 0.000 526.000 66.000
```

3.5 Beispiel 5

Abbildung 6: Die Dreiecksanordnung für das Beispiel 5

Ausgabe für Beispiel 5

```
Gesamtabstand: 0.000 Meter
Platzierung der Dreiecke:
D1 190.963 0.046 300.000 0.000 245.063 18.760
D2 0.000 160.000 39.000 61.000 76.000 124.000
D3 122.000 155.000 153.000 71.000 169.000 150.000
D4 226.822 90.471 202.568 33.272 300.000 0.000
D5 364.000 106.000 411.000 97.000 441.000 155.000
D6 605.000 172.000 654.000 139.000 615.000 229.000
D7 274.044 108.297 243.470 69.888 300.000 0.000
D8 306.323 42.814 287.470 52.278 300.000 0.000
D9 6.000 221.000 60.000 250.000 7.000 296.000
D10 374.000 217.000 398.000 172.000 407.000 277.000
D11 676.000 289.000 657.000 373.000 640.000 322.000
D12 300.000 0.000 355.224 52.300 313.360 90.463
D13 356.713 53.709 300.000 0.000 385.907 0.000
D14 100.000 0.000 170.000 20.000 120.000 70.000
D15 337.000 151.000 373.000 140.000 348.000 214.000
D16 96.000 61.000 104.000 154.000 80.000 165.000
D17 271.000 142.000 297.000 209.000 279.000 236.000
D18 48.000 307.000 33.000 406.000 6.000 340.000
D19 504.000 115.000 530.000 154.000 489.000 205.000
D20 668.000 46.000 666.000 131.000 605.000 122.000
D21 276.000 259.000 336.000 253.000 370.000 321.000
D22 224.000 295.000 158.000 313.000 171.000 232.000
D23 699.000 271.000 743.000 323.000 703.000 376.000
D24 93.000 223.000 136.000 331.000 86.000 334.000
```

```
D25 753.000 251.000 770.000 273.000 767.000 332.000
D26 298.000 114.000 335.000 24.000 260.000 42.000
D27 792.000 178.000 766.000 213.000 750.000 191.000
D28 243.000 244.000 254.000 273.000 240.000 303.000
D29 439.000 273.000 467.000 204.000 467.000 316.000
D30 68.000 201.000 97.000 163.000 80.000 232.000
D31 549.000 148.000 584.000 151.000 552.000 225.000
D32 564.000 131.000 544.000 63.000 621.000 79.000
D33 464.000 366.000 403.000 377.000 418.000 313.000
D34 240.000 212.000 213.000 10.000 249.000 89.000
D35 729.000 192.000 714.000 240.000 663.000 184.000
D36 171.000 172.000 188.000 213.000 116.000 223.000
D37 57.000 334.000 70.000 288.000 69.000 408.000
```

---

4 Quellcode

```cpp
#include<bits/stdc++.h>

using namespace std;

class Point{
    public:

    double x;
    double y;

    Point(double _x, double _y){
        x = _x;
        y = _y;
    }

    Point(){
        x = 0;
        y = 0;
    }
};

class Vektor{
    public:

    double x;
    double y;

    Vektor(double _x, double _y){
        x = _x;
        y = _y;
    }

    Vektor(Point a, Point b){
        x = b.x - a.x;
        y = b.y - a.y;
    }
```

```
37
38      Vektor(){
39          x = 0;
40          y = 0;
41      }
42
43      double betrag(){
44          return sqrt(x * x + y * y);
45      }
46  };
47
48  class Triangle{
49      public:
50
51      vector<Point> points;
52      vector<Vektor> vektoren;
53      vector<double> lengths;
54      int id;
55
56      Triangle(Point p1, Point p2, Point p3, int idd){
57          points = {p1,p2,p3};
58          id = idd;
59          reGenVectors();
60          for(int i=0;i<=2;i++){
61              lengths.push_back(vektoren[i].betrag());
62          }
63      }
64
65      void reGenVectors(){
66          Vektor p1p2 = Vektor(points[0],points[1]);
67          Vektor p2p3 = Vektor(points[1],points[2]);
68          Vektor p3p1 = Vektor(points[2],points[0]);
69          vektoren = {p1p2,p2p3,p3p1};
70      }
71
72      double shortestLength(int bestPoint){
73          switch(bestPoint) {
74          case 0: if(lengths[0] < lengths[2]){
75                      return 0;
76                  } else {
77                      return 2;
78                  }
79                  break;
80          case 1: if(lengths[0] < lengths[1]){
81                      return 0;
82                  } else {
83                      return 1;
84                  }
85                  break;
86          case 2: if(lengths[1] < lengths[2]){
87                      return 1;
88                  } else {
89                      return 2;
```

```
                }
                break;
        default: return 0;
                 break;
        }
    }
};

Point addVektor(Point &p, Vektor &v){
    p.x += v.x;
    p.y += v.y;
    return p;
}

double dotProduct(Vektor &v1, Vektor &v2){
    return v1.x * v2.x + v1.y * v2.y;
}

double angle(Vektor &v1, Vektor &v2){
    double cosvalue = dotProduct(v1,v2)/(v1.betrag()*v2.betrag());
    return acos(abs(cosvalue));
}

pair<int,double> locateSmallestAngle(Triangle t){
    double bestangle = M_PI;
    int pointindex;
    for(size_t i=0;i<=2;i++){
        double thisangle = angle(t.vektoren[i],t.vektoren[(i+1)%3]);
        if(thisangle < bestangle){
            bestangle = thisangle;
            pointindex = (i+1)%3;
        }
    }
    return {pointindex,bestangle};
}

//based on https://stackoverflow.com/questions/2259476/rotating-a-point-about-another-point-2d
void rotate_tri(Point center, Point &p, double angle){
    double sinus = sin(angle);
    double cosinus = cos(angle);

    p.x -= center.x;
    p.y -= center.y;

    double xnew = p.x * cosinus - p.y * sinus;
    double ynew = p.x * sinus + p.y * cosinus;

    p.x = xnew + center.x;
    p.y = ynew + center.y;
}

double atan_angle(Point center, Point p){
```

```
    double dx = p.x - center.x;
    double dy = p.y - center.y;

    double angle = atan2(dy,dx);
    if(dy < 0){
        angle += 2 * M_PI;
    }

    return 2* M_PI - angle;
}

double ccw(Point a, Point b, Point c){
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}
```

Quellcode 1: Die Klassen `Triangle`, `Vektor` und `Point`

```
#include<bits/stdc++.h>
#include"triangles.cpp"

using namespace std;

vector<int> bestPointIndex;
vector<int> bestAngle;
vector<double> bestAngleDouble;
vector<int> sol;

bool getSubsetsRec(vector<int> arr, int i, int sum, vector<int>& p,
↪   vector<vector<bool>> &dp)
{
    // If we reached end and sum is non-zero. We print
    // p[] only if arr[0] is equal to sun OR dp[0][sum]
    // is true.
    if (i == 0 && sum != 0 && dp[0][sum])
    {
        p.push_back(i);
        sol = p;
        return true;
    }

    // If sum becomes 0
    if (i == 0 && sum == 0)
    {
        sol = p;
        return true;
    }

    // If given sum can be achieved after ignoring
    // current element.
    if (dp[i-1][sum])
    {
        // Create a new vector to store path
        vector<int> b = p;
```

```
        if(getSubsetsRec(arr, i-1, sum, b,dp)){
            return true;
        }
    }

    // If given sum can be achieved after considering
    // current element.
    if (sum >= arr[i] && dp[i-1][sum-arr[i]])
    {
        p.push_back(i);
        if(getSubsetsRec(arr, i-1, sum-arr[i], p,dp)){
            return true;
        }
    }
    return false;
}

void subsetSum(vector<int> set,int sum){
    int n = set.size();
    vector<vector<bool>> dp(n,vector<bool>(sum+1));
    for (int i=0; i<n; ++i) {
        dp[i][0] = true;
    }

    // Sum arr[0] can be achieved with single element
    if (set[0] <= sum)
       dp[0][set[0]] = true;

    // Fill rest of the entries in dp[][]
    for (int i = 1; i < n; ++i)
        for (int j = 0; j < sum + 1; ++j)
            dp[i][j] = (set[i] <= j) ? dp[i-1][j] ||
                                       dp[i-1][j-set[i]]
                                     : dp[i - 1][j];

    int best = sum;
    for(;best>=0;best--){
        if(dp[n-1][best]) break;
    }
    cout << best << endl;
    vector<int> p;
    getSubsetsRec(set, n-1, best, p,dp);
    for(auto x: sol) cout << x << " ";
    cout << "\n";
}

int findAngleCalcPoint(Triangle &t, int bestPoint){
    //just return the point that is counterclockwise from line between other
    ↪ points
    switch(bestPoint) {
        case 0: if(ccw(t.points[0],t.points[1],t.points[2]) > 0){
                    //point 2 is clockwise
                    return 1;
```

```
                } else {
                    //point 2 is counterclockwise
                    return 2;
                }
                break;
        case 1: if(ccw(t.points[1],t.points[0],t.points[2]) > 0){
                    //point 2 is clockwise
                    return 0;
                } else {
                    //point 2 is counterclockwise
                    return 2;
                }
                break;
        case 2: if(ccw(t.points[2],t.points[1],t.points[0]) > 0){
                    //point 0 is clockwise
                    return 1;
                } else {
                    //point 0 is counterclockwise
                    return 0;
                }
                break;
        default: return 0;
                 break;
    }
}

bool triangleSortFunc(Triangle &t1, Triangle &t2){
    return t1.shortestLength(bestPointIndex[t1.id-1]) <
    ↪  t2.shortestLength(bestPointIndex[t2.id-1]);
}

pair<vector<Triangle>,double> doAlgorithm(vector<Triangle> triangles, bool
↪  debug){
    Point centerPoint = Point(300,0);
    for(size_t i = 0; i<triangles.size(); i++){
        auto &t = triangles[i];
        int ind;
        double angle;
        tie(ind,angle) = locateSmallestAngle(t);
        bestPointIndex.push_back(ind);
        bestAngle.push_back((int) ceil(10000*angle));
        bestAngleDouble.push_back(angle);
        cout << t.id << " " << t.points[ind].x << " " << t.points[ind].y << "
        ↪  " << angle << "\n";
    }

    subsetSum(bestAngle,(int) floor(10000*M_PI));

    for(auto index : sol){
        auto &t = triangles[index];
        Vektor translation =
        ↪  Vektor(t.points[bestPointIndex[index]],centerPoint);
        for(int i=0;i<=2;i++){
```

```
137             t.points[i] = addVektor(t.points[i],translation);
138         }
139         int rotatePoint = findAngleCalcPoint(t,bestPointIndex[index]);
140         double rotateAngle = atan_angle(centerPoint,t.points[rotatePoint]);
141         for(int i=0;i<=2;i++){
142             rotate_tri(centerPoint, t.points[i], rotateAngle);
143         }
144         t.reGenVectors();
145     }
146 
147     double triRotateAngle = bestAngleDouble[sol[0]];
148     for(size_t i=1;i<sol.size();i++){
149         auto &t = triangles[sol[i]];
150         for(int j=0;j<=2;j++){
151             rotate_tri(centerPoint, t.points[j], triRotateAngle);
152         }
153         triRotateAngle += bestAngleDouble[sol[i]];
154     }
155 
156     //TODO order the triangles -> lengths
157     //TODO what to do with triangles outside the subset
158     //TODO Gesamtabstand berechnen
159 
160     return {triangles,0};
161 }
```

Quellcode 2: Die Datei `triangleAlgorithm`, die alle wesentlichen Bestandteile des Algorithmus enthält