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.*

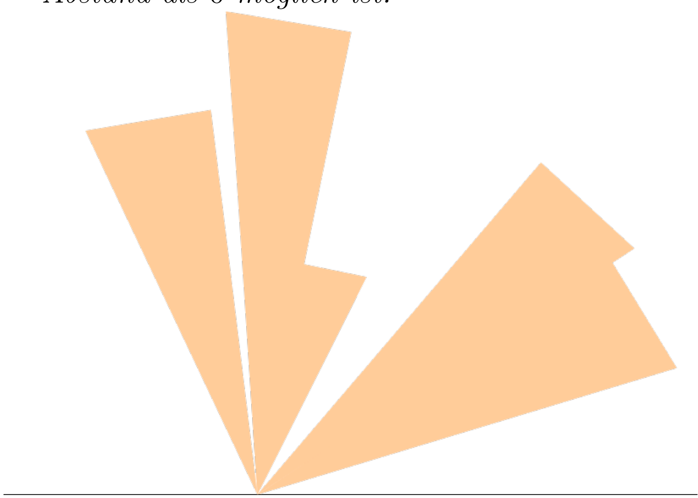

Abbildung 1: Optimale Anordnung für eine Summe kleiner 180°

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

```
Gesamtabstand: 142.874 Meter
Platzierung der Dreiecke:
D1 300.000 0.000 228.640 123.777 157.126 0.133
D2 300.000 0.000 371.476 123.710 228.640 123.777
D3 300.000 0.000 442.874 0.000 371.476 123.710
D4 442.874 0.000 514.292 123.744 371.456 123.744
D5 442.874 0.000 585.748 0.067 514.292 123.744
```

[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:
D2 300.000 0.000 231.141 568.790 380.260 552.560
D4 300.000 0.000 374.227 511.025 521.723 483.730
D3 300.000 0.000 548.868 144.822 587.939 0.000
D5 300.000 0.000 508.921 455.799 642.677 387.909
D1 300.000 0.000 615.202 356.807 696.231 230.576
```

3.3 Beispiel 3

Abbildung 4: Die Dreiecksanordnung für das Beispiel 3

Ausgabe für Beispiel 3

```
Gesamtabstand: 93.000 Meter
Platzierung der Dreiecke:
D4 30.220 0.000 300.000 0.000 75.776 60.446
D11 125.405 111.206 90.222 56.551 300.000 0.000
D12 103.745 219.419 300.000 0.000 130.899 107.707
D6 300.000 0.000 232.961 189.726 172.785 142.230
D1 297.198 189.979 233.205 189.036 300.000 0.000
D9 300.000 0.000 297.713 155.099 363.073 243.717
D10 300.000 0.000 388.889 84.283 325.950 100.273
D8 392.417 0.086 350.238 47.635 300.000 0.000
D2 383.741 189.840 393.000 0.000 480.741 244.429
D5 518.371 156.867 393.000 0.000 457.193 178.828
D7 526.729 11.682 393.000 0.000 492.747 66.502
D3 393.000 0.000 526.129 88.757 505.932 141.303
```

3.4 Beispiel 4

Abbildung 5: Die Dreiecksanordnung für das Beispiel 4

Ausgabe für Beispiel 4

```
Gesamtabstand: 162.000 Meter
Platzierung der Dreiecke:
D12 105.000 0.082 300.000 0.000 105.019 45.082
D10 130.855 88.826 300.000 0.000 112.740 43.297
```

```
D15 166.402 98.527 300.000 0.000 124.452 92.189
D14 300.000 0.000 140.420 117.688 183.553 116.897
D17 300.000 0.000 159.486 141.056 208.766 131.440
D11 186.759 163.146 300.000 0.000 233.523 126.292
D6 244.500 158.571 234.128 125.143 300.000 0.000
D16 253.281 133.482 343.382 90.100 300.000 0.000
D1 374.917 74.917 328.198 58.565 300.000 0.000
D5 365.000 65.000 365.000 0.000 300.000 0.000
D9 365.000 0.000 413.042 193.039 383.728 141.486
D4 432.055 120.369 406.298 165.938 365.000 0.000
D20 437.440 68.392 365.000 0.000 411.233 82.991
D23 415.899 48.055 461.208 0.064 365.000 0.000
D13 462.000 0.000 396.095 183.525 438.447 198.734
D8 503.525 165.882 440.470 181.666 462.000 0.000
D7 552.832 158.794 502.797 162.974 462.000 0.000
D21 462.000 0.000 539.784 54.540 548.348 103.291
D2 567.941 1.254 527.792 23.905 462.000 0.000
D22 572.232 77.291 551.289 32.443 462.000 0.000
D18 560.857 172.824 568.469 127.359 462.000 0.000
D19 462.000 0.000 594.354 16.808 551.138 32.853
D3 462.000 0.000 591.285 47.650 599.558 96.451
```

3.5 Beispiel 5

Abbildung 6: Die Dreiecksanordnung für das Beispiel 5

Ausgabe für Beispiel 5

```
Gesamtabstand: 951.154 Meter
Platzierung der Dreiecke:
D21 84.395 0.039 131.164 38.021 197.000 0.000
D29 123.253 10.312 197.000 0.000 100.051 56.080
D19 140.995 72.031 136.676 25.358 197.000 0.000
D25 184.805 81.291 175.590 55.060 197.000 0.000
D4 201.538 62.010 197.044 0.044 300.000 0.000
D7 240.479 94.124 223.938 47.902 300.000 0.000
D13 258.253 66.017 300.000 0.000 316.589 84.290
D12 300.000 0.000 357.655 49.607 317.658 89.723
D8 343.278 0.000 349.887 20.033 300.000 0.000
D1 382.653 71.116 300.000 0.000 353.870 21.633
D16 344.000 0.000 406.686 69.163 394.249 92.450
D37 402.975 46.281 424.590 88.917 344.000 0.000
D23 344.000 0.000 412.117 0.082 426.661 64.870
D31 459.920 61.127 429.614 78.892 413.000 0.000
D10 473.133 32.757 477.167 83.598 413.000 0.000
D15 476.953 0.009 481.592 37.365 413.000 0.000
D3 487.921 88.869 477.000 0.000 527.410 62.896
D36 548.971 21.123 522.461 56.721 477.000 0.000
D17 477.000 0.000 548.868 0.015 567.522 26.568
D2 549.000 0.000 655.404 0.311 610.182 57.695
```

```
D34 549.000 0.000 697.270 139.818 619.531 101.170
D35 711.889 36.006 677.442 72.645 656.000 0.000
D6 705.663 31.994 656.000 0.000 754.086 0.306
D32 760.356 76.970 694.638 50.413 755.000 0.000
D9 815.519 44.311 759.871 70.009 755.000 0.000
D27 755.000 0.000 798.600 0.013 790.474 25.973
D20 799.000 0.000 871.691 44.103 833.482 92.499
D11 872.630 44.673 799.000 0.000 852.759 0.062
D18 893.281 91.670 853.000 0.000 911.116 41.322
D5 943.888 8.328 906.218 37.840 853.000 0.000
D24 943.888 0.000 993.600 105.080 943.888 111.221
D14 943.888 0.000 1016.211 8.333 975.021 65.808
D33 1073.091 41.105 1016.211 65.734 1016.211 0.000
D30 1054.955 27.998 1016.211 0.000 1085.631 15.193
D26 1085.631 0.000 1085.631 97.309 1148.154 52.143
D28 1141.904 17.982 1111.055 21.204 1085.631 0.000
D22 1210.467 53.805 1148.154 82.037 1148.154 0.000
```

---

4 Quellcode

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

using namespace std;

// Klasse für einen Punkt (x/y-Koordinate)
class Point{
    public:

    double x;
    double y;

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

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

// KLasse für einen Vektor im R^2
class Vektor{
    public:

    double x;
    double y;

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

```
    }

    // Vektor zwischen zwei Punkten
    Vektor(Point a, Point b){
        x = b.x - a.x;
        y = b.y - a.y;
    }

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

    // Betrag/Laenge des Vektors
    double betrag(){
        return sqrt(x * x + y * y);
    }
};

// Klasse für ein Dreieck
class Triangle{
    public:

    // Punkte + Vektoren der Seiten + Längen dieser
    vector<Point> points;
    vector<Vektor> vektoren;
    vector<double> lengths;
    int id;

    Triangle(Point p1, Point p2, Point p3, int idd){
        points = {p1,p2,p3};
        id = idd;
        reGenVectors();
        for(int i=0;i<=2;i++){
            lengths.push_back(vektoren[i].betrag());
        }
    }

    // Vektoren nach Drehung etc. neu erstellen
    void reGenVectors(){
        Vektor p1p2 = Vektor(points[0],points[1]);
        Vektor p2p3 = Vektor(points[1],points[2]);
        Vektor p3p1 = Vektor(points[2],points[0]);
        vektoren = {p1p2,p2p3,p3p1};
    }

    // Berechnung kürzeste an einem Punkt anliegende Seite
    double shortestLength(int bestPoint){
        switch(bestPoint) {
        case 0: if(lengths[0] < lengths[2]){
                    return lengths[0];
                } else {
                    return lengths[2];
```

```
                }
                break;
        case 1: if(lengths[0] < lengths[1]){
                    return lengths[0];
                } else {
                    return lengths[1];
                }
                break;
        case 2: if(lengths[1] < lengths[2]){
                    return lengths[1];
                } else {
                    return lengths[2];
                }
                break;
        default: return 0;
                 break;
        }
    }
};

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

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

// Winkel zwischen zwei Vektoren
// allgemein bekannte Cosinus-Formel
double angle(Vektor &v1, Vektor &v2){
    double cosvalue = dotProduct(v1,v2)/(v1.betrag()*v2.betrag());
    return acos(abs(cosvalue));
}

// kleinsten Winkel und anliegenden Punkt eines Dreiecks bestimmen
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};
}
```

```
// Punkt mithilfe einer Drehmatrix rotieren um ein Zentrum rotieren
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;
}

// Winkel zur positiven x-Achse mit atan2
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 angle;
}

// 360 Grad minus diesen Winkel
// (zum anfänglichen Drehen, so dass Dreieck auf x-Achse liegt)
double atan360(Point center, Point p){
    return 2 * M_PI - atan_angle(center,p);
}

// ähnlich (Berechnen des freien Winkels links)
double atan180(Point center, Point p){
    return M_PI - atan_angle(center,p);
}

// Lage Punkt c von ab aus
// CCW: < 0, CW: >0
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);
}

// Punkt finden, von dem aus Drehwinkel bestimmt wird
// (beim Drehen auf die x-Achse)
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;
    }
}
```

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

```
#include<bits/stdc++.h>
#include"triangles.cpp"
#define EPSILON 0.5

using namespace std;

vector<int> bestPointIndex;
vector<int> bestAngle;
vector<double> bestAngleDouble;
vector<int> sol;
vector<Triangle> triangles;
vector<Triangle> placedTriangles;

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";
}

bool lengthSortFunc(const int t1, const int t2){
    return triangles[t1].shortestLength(bestPointIndex[t1]) < triangles[t2].shortestLength(bestPointIndex[t2]);
}

// Sortieren der Dreiecke anhand ihres Mittelpunkts (x-Koordinate)
bool triangleSortFunc(const Triangle &t1, const Triangle &t2){
    double mid1 = (t1.points[0].x + t1.points[1].x + t1.points[2].x) / 3;
    double mid2 = (t2.points[0].x + t2.points[1].x + t2.points[2].x) / 3;
    return mid1 < mid2;
}

void deleteUsedTriangles(){
    sort(sol.begin(),sol.end(),greater<int>());
    for(auto index: sol){
        placedTriangles.push_back(triangles[index]);
        triangles.erase(triangles.begin()+index);
        bestAngle.erase(bestAngle.begin()+index);
        bestAngleDouble.erase(bestAngleDouble.begin()+index);
        bestPointIndex.erase(bestPointIndex.begin()+index);
    }
}

void translateAndRotateToAxis(Point centerPoint){
    for(auto index : sol){
        auto &t = triangles[index];
        Vektor translation = Vektor(t.points[bestPointIndex[index]],centerPoint);
        for(int i=0;i<=2;i++){
            t.points[i] = addVektor(t.points[i],translation);
        }
        int rotatePoint = findAngleCalcPoint(t,bestPointIndex[index]);
        double rotateAngle = atan360(centerPoint,t.points[rotatePoint]);
        for(int i=0;i<=2;i++){
            rotate_tri(centerPoint, t.points[i], rotateAngle);
        }
    }
}

void rotateToPositionRight(Point centerPoint, double free_angle, bool rotateLeft){
    double triRotateAngle = 0;
    for(size_t i=0;i<sol.size();i++){
        auto &t = triangles[sol[i]];
        for(int j=0;j<=2;j++){
```

```
            rotate_tri(centerPoint, t.points[j], triRotateAngle);
        }
        triRotateAngle += bestAngleDouble[sol[i]];
    }
    // "Randrehen"
    if(rotateLeft){
        for(size_t i=0;i<sol.size();i++){
            auto &t = triangles[sol[i]];
            for(int j=0;j<=2;j++){
                rotate_tri(centerPoint, t.points[j], free_angle-triRotateAngle);
            }
        }
    }
}

void rotateToPositionLeft(Point centerPoint, double free_angle){
    double triRotateAngle = 0;
    for(size_t i=0;i<sol.size();i++){
        auto &t = triangles[sol[i]];
        for(int j=0;j<=2;j++){
            rotate_tri(centerPoint, t.points[j], M_PI - triRotateAngle);
        }
        triRotateAngle += bestAngleDouble[sol[i]];
    }
    // "Randrehen"
    for(size_t i=0;i<sol.size();i++){
        auto &t = triangles[sol[i]];
        for(int j=0;j<=2;j++){
            rotate_tri(centerPoint, t.points[j], -(free_angle-triRotateAngle));
        }
    }
}

pair<double,double> calculateDistance(){
    double leftmost=300,rightmost=300;
    for(auto tri: placedTriangles){
        double left_triangle = 100000, right_triangle = 0;
        for(auto p: tri.points){
            if(p.y == 0){
                if(p.x < left_triangle) left_triangle = p.x;
                if(p.x > right_triangle) right_triangle = p.x;
            }
        }
        if(left_triangle < 300 && right_triangle < leftmost) leftmost = right_triangle;
        if(right_triangle > 300 && left_triangle > rightmost) rightmost = left_triangle;
    }
    return {leftmost, rightmost};
}
```

```
void moveToRightOfY(){
    double minx = 100000;
    for(auto t: placedTriangles){
        for(auto pnt: t.points){
            if(pnt.x < minx) minx = pnt.x;
        }
    }
    if(minx < 0){
        minx = abs(minx);
        for(auto t: placedTriangles){
            for(auto pnt: t.points){
                pnt.x += minx;
            }
        }
    }
}

pair<vector<Triangle>,double> doAlgorithm(vector<Triangle> i_triangles){
    triangles = i_triangles;
    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);
    }

    subsetSum(bestAngle,(int) floor(10000*M_PI));
    sort(sol.begin(),sol.end(),lengthSortFunc);

    translateAndRotateToAxis(centerPoint);
    rotateToPositionRight(centerPoint,M_PI,false);
    deleteUsedTriangles();

    bool move = false;
    double leftmost, rightmost;
    while(!triangles.empty()){
        tie(leftmost,rightmost) = calculateDistance();

        double minx_above = 100000, miny_above = -1, maxx_above = 0,
         ↪  maxy_above = -1;
        double minx_axis = 100000, maxx_axis = 0;
        for(auto t: placedTriangles){
            for(auto pnt: t.points){
                if(pnt.y <= EPSILON){
                    if(pnt.x < minx_axis) {minx_axis = (pnt.y > 0) ?
                     ↪  floor(pnt.x) : pnt.x;}
                    if(pnt.x > maxx_axis) {maxx_axis = (pnt.y > 0) ?
                     ↪  ceil(pnt.x) : pnt.x;}
                } else if(pnt.y > EPSILON) {
```

```
                    if(pnt.x < minx_above) { minx_above = pnt.x; miny_above = pnt.y; }
                    if(pnt.x > maxx_above) { maxx_above = pnt.x; maxy_above = pnt.y; }
                }
            }
        }

        Point newCenterRight = (move) ? Point(maxx_above,0) : Point(maxx_axis,0);
        Point newCenterLeft = (move) ? Point(minx_above,0) : Point(minx_axis,0);

        bool left = false;
        double free_angle = 0;
        if(leftmost - newCenterLeft.x < newCenterRight.x - rightmost){
            //place left
            free_angle = atan180(newCenterLeft,Point(minx_above,miny_above));
            left = true;
        } else {
            //place right
            free_angle = atan_angle(newCenterRight,Point(maxx_above,maxy_above));
        }
        if(move) move = false;

        subsetSum(bestAngle,(int) floor(10000*free_angle));
        if(sol.size() == 0){
            move = true;
            continue;
        }

        sort(sol.begin(),sol.end(),lengthSortFunc);

        if(left){
            reverse(sol.begin(),sol.end());
            translateAndRotateToAxis(newCenterLeft);
            rotateToPositionLeft(newCenterLeft,free_angle);
        } else {
            translateAndRotateToAxis(newCenterRight);
            rotateToPositionRight(newCenterRight,free_angle,true);
        }

        deleteUsedTriangles();
    }

    moveToRightOfY();
    sort(placedTriangles.begin(),placedTriangles.end(),triangleSortFunc);

    //TODO sortieren von beiden seiten wenn sinnvoll -> nach oben hin
    //TODO ggf. spiegeln an seitenhalbierender
    //TODO sometimes triangles are overlapping (sweepline?)
    //TODO randrehen ist bei Beispiel 5 schlecht
```

```

    auto dpair = calculateDistance();
    double dist = dpair.second - dpair.first;
    return {placedTriangles,dist};
}
```

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