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:
$$\max_{d_i \in D'} \max_{d_j \in D'} \min_{n \in N_i} \min_{m \in N_j} |n.x - m.x| \tag{4}$$
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.*

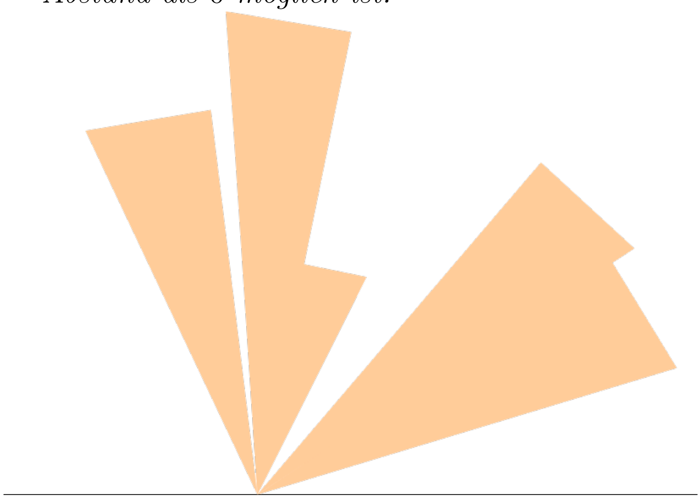

Abbildung 1: Optimale Anordnung für eine Summe kleiner 180°

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 0.000 514.350 123.710 371.514 123.777
D5 442.874 0.000 585.748 0.000 514.350 123.710
```

[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:
D12 16.668 79.899 300.000 0.000 99.511 0.187
D11 207.951 185.410 149.592 156.788 300.000 0.000
D9 300.000 0.000 166.811 79.508 125.724 181.669
D6 300.000 0.000 285.779 200.718 215.146 170.918
D4 40.347 73.222 300.000 0.000 100.599 119.034
D1 347.774 183.896 285.830 199.988 300.000 0.000
D10 300.000 0.000 388.968 84.201 326.043 100.248
D8 392.418 0.000 350.283 47.588 300.000 0.000
D2 400.296 189.926 393.000 0.000 501.679 235.866
D7 527.239 0.000 393.000 0.000 498.156 57.569
D5 531.546 145.362 393.000 0.000 472.511 172.563
D3 393.000 0.000 533.347 76.835 517.799 130.939
```

3.4 Beispiel 4

Abbildung 5: Die Dreiecksanordnung für das Beispiel 4

Ausgabe für Beispiel 4

```
Gesamtabstand: 267.948 Meter
Platzierung der Dreiecke:
D15 187.962 122.489 300.000 0.000 145.578 124.378
D14 300.000 0.000 128.537 99.582 171.498 103.500
D12 105.000 0.082 300.000 0.000 105.019 45.082
```

```
D11 117.387 78.055 300.000 0.000 176.585 71.677
D10 211.011 169.059 300.000 0.000 170.279 141.822
D6 244.500 158.571 234.128 125.143 300.000 0.000
D17 300.000 0.000 106.017 44.852 152.876 62.886
D16 253.281 133.482 343.382 90.100 300.000 0.000
D5 365.000 65.000 365.000 0.000 300.000 0.000
D1 374.917 74.917 328.198 58.565 300.000 0.000
D9 365.000 0.000 413.170 193.007 383.822 141.473
D4 432.134 120.324 406.407 165.911 365.000 0.000
D20 437.486 68.344 365.000 0.000 411.287 82.961
D23 415.931 48.021 461.208 0.000 365.000 0.000
D13 462.000 0.000 398.272 184.293 440.801 198.999
D18 440.910 197.980 473.707 165.587 462.000 0.000
D7 523.849 172.165 473.848 167.585 462.000 0.000
D21 462.000 0.000 540.425 53.615 549.565 102.261
D2 567.948 0.000 528.070 23.125 462.000 0.000
D22 573.139 75.981 551.667 31.383 462.000 0.000
D8 573.221 129.888 523.849 172.165 462.000 0.000
D19 567.948 0.000 701.365 0.000 660.515 21.362
D3 567.948 0.000 702.206 30.983 716.561 78.352
```

3.5 Beispiel 5

Abbildung 6: Die Dreiecksanordnung für das Beispiel 5

Ausgabe für Beispiel 5

```
Gesamtabstand: 880.767 Meter
Platzierung der Dreiecke:
D7 240.479 94.124 223.938 47.902 300.000 0.000
D4 201.538 62.010 197.044 0.044 300.000 0.000
D1 241.723 92.156 300.000 0.000 286.470 56.453
D12 300.000 0.000 331.757 69.112 278.687 88.926
D8 343.278 0.000 349.887 20.033 300.000 0.000
D13 332.613 70.975 300.000 0.000 379.719 32.013
D21 456.606 0.000 409.823 38.044 344.000 0.000
D34 344.000 0.000 520.445 101.982 435.902 82.243
D14 457.000 0.000 529.801 0.000 495.461 61.812
D37 535.627 74.740 518.860 119.504 529.801 0.000
D18 537.583 99.827 529.801 0.000 571.030 58.182
D17 529.801 0.000 586.175 44.575 584.346 76.973
D25 603.922 35.539 576.141 36.641 529.801 0.000
D31 606.860 0.000 602.499 34.857 529.801 0.000
D11 599.828 85.834 606.860 0.000 630.844 48.112
D15 659.290 36.620 641.708 69.904 606.860 0.000
D36 681.866 0.000 666.454 41.623 606.860 0.000
D29 656.015 69.833 681.866 0.000 685.332 111.946
D23 681.866 0.000 725.494 52.312 685.117 105.026
D26 681.866 0.000 744.191 74.731 763.279 0.000
D16 763.279 0.000 762.311 93.338 737.366 101.983
D20 763.279 0.000 815.291 67.259 762.255 98.711
```

```
D28 822.355 0.000 793.951 12.458 763.279 0.000
D5 847.838 34.346 803.225 51.656 763.279 0.000
D10 882.051 33.547 885.415 84.436 822.355 0.000
D19 913.596 0.000 879.401 32.058 822.355 0.000
D32 958.788 62.536 888.957 74.685 913.596 0.000
D33 983.775 0.000 952.098 53.279 913.596 0.000
D24 983.775 0.000 991.878 115.963 943.336 103.608
D27 983.775 0.000 1027.375 0.000 1019.256 25.963
D9 1044.306 44.293 988.667 70.008 983.775 0.000
D30 1071.209 19.067 1027.375 0.000 1098.439 0.000
D2 1027.375 0.000 1124.948 42.444 1060.691 77.214
D22 1180.767 0.000 1152.053 62.093 1098.439 0.000
D6 1165.395 57.041 1180.767 0.000 1210.311 93.531
D3 1207.736 85.380 1180.767 0.000 1241.817 52.630
D35 1247.250 0.000 1238.135 49.456 1180.767 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]);
}

bool triangleSortFunc(Triangle t1, Triangle t2){
    double right1 = 0, right2 = 0;
    for(auto p: t1.points){
        if(p.x > right1) right1 = p.x;
    }
    for(auto p: t2.points){
        if(p.x > right2) right2 = p.x;
    }
    return right1 < right2;
}

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 rotateToPosition(Point centerPoint){
    double triRotateAngle = bestAngleDouble[sol[0]];
    for(size_t i=1;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]];
    }
}

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 rightmost - leftmost;
}

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);
    rotateToPosition(centerPoint);
    deleteUsedTriangles();

    while(!triangles.empty()){
        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 newCenter = Point(maxx_axis,0);
        double free_angle =
        ↪  atan_angle(newCenter,Point(maxx_above,maxy_above));

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

        translateAndRotateToAxis(newCenter);
        rotateToPosition(newCenter);
        deleteUsedTriangles();
    }

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

    //TODO order the triangles -> lengths (von beiden seiten wenn sinnvoll)
    //TODO triangles auch links anfügen, wenn besser + spiegeln, wenn besser
    //TODO sometimes triangles are overlapping
    //TODO an die bisherige Konstruktion "randrehen"
    //TODO alles kommentieren

    return {placedTriangles,calculateDistance()};
}
```

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