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:
$$\forall\ (x, y) \in d : y \geq 0 \wedge x \geq 0 \tag{1}$$
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:
$$\max_{d_i \in D'\ d_j \in D'}\ \min_{n \in N_i\ m \in N_j} |n.x - m.x| \tag{4}$$
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 (möglichst) 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[2] 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. Eine heuristische Herangehensweise an dieses Problem wird in den folgenden Abschnitten näher beschrieben.

[2] Auf Englisch übrigens auch als *ansatz* bekannt.

1.3 Beschreibung der Heuristik

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. Der Punkt des Dreiecks, an dem der Winkel anliegt, werde als **Optimalpunkt** bezeichnet. Ist die Summe $\sum_{i=1}^{n} \varphi_i \leq 180°$, 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 (dieser ist mit dem jeweiligen Optimalpunkt identisch) 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.*

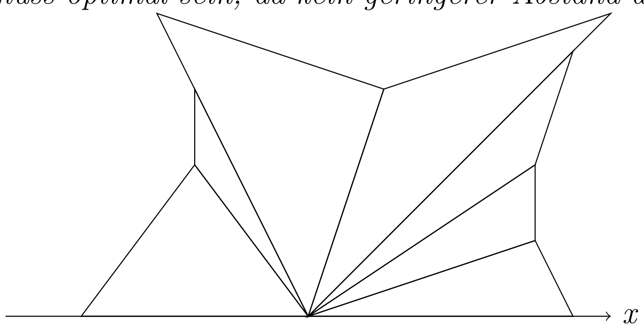

Abbildung 1: Optimale Anordnung für eine Summe kleiner gleich 180°

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 $\sum \varphi_i \leq 180°$ gilt und $\sum \varphi_i$ maximal ist. Für diese kann die in der vorherigen Beobachtung beschriebene Strategie angewandt werden.*

Nun müssen aber noch alle weiteren Dreiecke angeordnet werden.

Auch dafür gibt es eine sinnvoll erscheinende Strategie:

Beobachtung 3. *Für das Anfügen der Dreiecke sollte man folgendermaßen vorgehen:*

1. *Finde zwei Punkte, die auf der x-Achse liegen[a], zu einem platzierten Dreieck gehören und möglichst weit links (Punkt L mit minimaler x-Koordinate) bzw. rechts (Punkt R mit maximaler x-Koordinate) liegen.*
2. *Wähle von diesen beiden Punkten den aus, der die Gesamtdistanz weniger erhöht.*
3. *Berechne für diesen Punkt den von der x-Achse aus **freien Winkel** α, in dem weitere Dreiecke platziert werden können.*

4. *Finde eine Teilmenge der noch nicht platzierten Dreiecke, für die* $\sum \varphi_i \leq \alpha$ *gilt und* $\sum \varphi_i$ *maximal ist.*
5. *Platziere diese Dreiecke mittels Verschiebungen und Rotationen im freien Winkel, sodass der Optimalpunkt jeweils auf L bzw. R (je nachdem, welcher gewählt wurde) liegt.*
6. *Wiederhole diese Schritte, bis alle Dreiecke platziert sind.*

[a] Oder nur wenig über der x-Achse. In diesem Fall muss man den Punkt aber ein wenig nach links/rechts und auf die x-Achse verschieben.

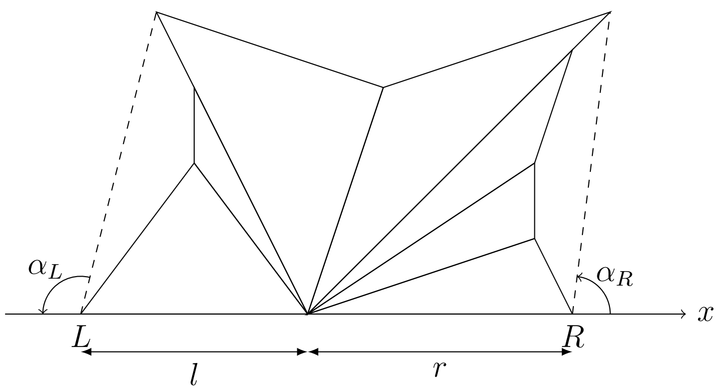

Abbildung 2: Veranschaulichung der Strategie

Die ersten beiden Schritte dienen dabei dazu, die Gesamtdistanz möglichst zu minimieren, während die restlichen Schritte Dreiecke finden, die optimal im freien Winkel platziert werden können. Die Funktionsweise der Strategie ist in obiger Abbildung veranschaulicht.

Die Gesamtdistanz ist dort 0 und kann entweder links um l oder rechts um r erweitert werden. Da in diesem Fall $l < r$, würden die Dreiecke auf der linken Seite im freien Winkel α_L angefügt werden.

1.3.2 Implementierte Verbesserungen

Diese Strategie lässt sich natürlich noch durch einige weitere Heuristiken verbessern. Als besonders erfolgreich haben sich dabei folgende herausgestellt, die deshalb auch implementiert wurden.

Sortieren der Dreiecke. Es ist sinnvoll, die Dreiecke des gewählten Subsets nach der längsten am Optimalpunkt anliegenden Seite zu sortieren. Dadurch befinden sich „längere“ bzw. größere Dreiecke weiter links und kleinere Dreiecke weiter rechts. Dadurch wird der Gesamtabstand beim Anlegen auf der rechten Seite nur wenig erhöht und es steht dort bestenfalls ein größerer freier Winkel zur Verfügung. Insgesamt führte diese Strategie in Experimenten zu einer Verringerung der Gesamtdistanz.

Nicht zum gewünschten Erfolg führte es dagegen, die Dreiecke so zu sortieren, dass Dreiecke mit langen anliegenden Seiten nach oben (d.h eher in y-Richtung) zeigen, während Dreiecke mit kurzen anliegenden Seiten nach links bzw. rechts (d.h eher in x-Richtung) zeigen. Deshalb ist diese Sortierung nicht in der endgültigen Implementierung enthalten. □

„Randrehen“ der Dreiecke. Es erwies sich auch als sinnvoll, die Dreiecke des ausgewählten Subsets so nah wie möglich an die bisherige Konstruktion heranzudrehen. Dadurch wird der freie Winkel nicht von unten, sondern von oben her mit Dreiecken aufgefüllt. Falls die Dreiecke den freien Winkel nicht komplett ausfüllen, bleibt unten (an der x-Achse) mehr Platz, um neue Dreiecke anzulegen, was ebenfalls zu einer Verringerung der Gesamtdistanz führte. □

Die erstgenannte Sortierheuristik führt jedoch dazu, dass links fast keine Dreiecke mehr angefügt werden. Dies ist dadurch begründet, dass durch die langen Dreiecke links die Gesamtdistanz im Vergleich zur rechten Seite unverhältnismäßig erhöht würde. Trotzdem erzeugt der Algorithmus mit Sortierheuristik nahezu immer bessere Ergebnisse als ohne.

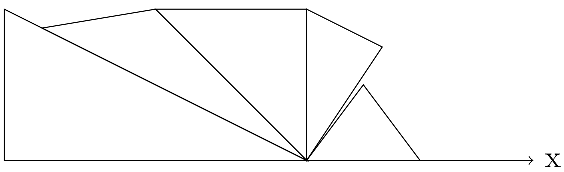

Abbildung 3: Sortieren von rechts nach links

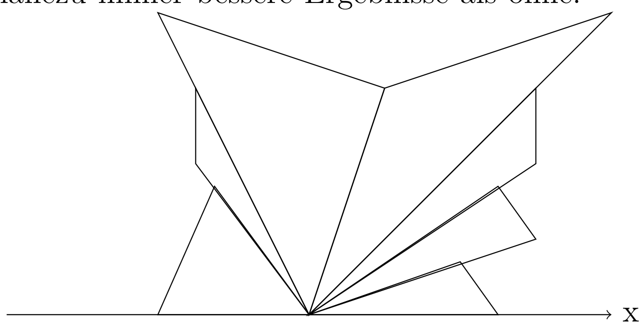

Abbildung 4: Sortieren von unten nach oben

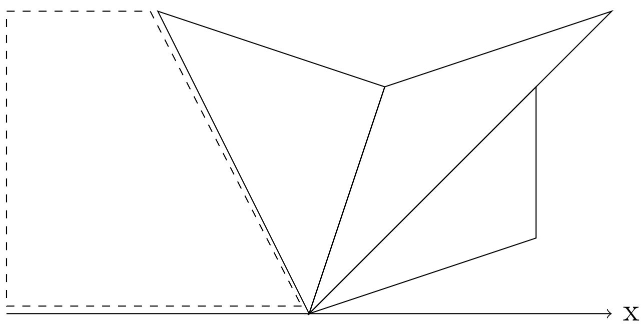

Abbildung 5: Herandrehen der Dreiecke
(der gepunktete Bereich stellt die bisherige Dreiecksanordnung dar)

1.3.3 Geometrische Berechnungen

1.3.4 Der DP-Algorithmus für Subset Sum

Um anhand der Winkel der Dreiecke die jeweils im freien Winkel (anfangs z.B. in einem Halbkreis) zu platzierenden Dreiecke zu ermitteln, muss man das Subset-Sum-Problem (Teilsummenproblem) 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:

$$dp(i, sum) = \begin{cases} true & sum = 0 \lor (i = 0 \land sum = w_1) \\ false & sum > 0 \land i = 0 \\ dp(i-1, sum) \lor dp(i-1, sum - w_i) & \text{sonst} \end{cases} \tag{7}$$

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 $dp(n-1, c)$ 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 $dp(n-1, c) = true$ 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.[3][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.

1.3.5 Der Algorithmus zur Platzierung der Dreiecke

Die in den vorherigen Abschnitten gewonnenen Erkenntnisse können in folgendem Algorithmus zusammengefasst werden:

Algorithmus 1 Algorithmus zur Dreiecksplatzierung

```
function doAlgorithm(T)
    return 42
end function
```

[3] Hiermit ist das Rückverfolgen einer Lösung bei DP-Algorithmen gemeint und nicht Backtracking.

1.4 Optimalität des Algorithmus und Verbesserungsmöglichkeiten

Bezüglich der Qualität des Verfahrens lässt sich feststellen, dass es viele typische Nachteile einer Heuristik besitzt. So muss die von diesem Verfahren erzeugte Dreiecksanordnung nicht unbedingt optimal sein. Um das zu erreichen, müsste man jedoch alle möglichen Lösungen mittels Backtracking durchprobieren.

Dies ist aber angesichts der zu erwartenden hohen Laufzeiten kein guter Ansatz. Für Beispiel 5 (mit $n = 37$) würde sich beispielsweise ergeben:

$$37! \cdot 6^{37} \approx 8,5 \cdot 10^{71} \tag{8}$$

Geht man davon aus, dass ein Computer in einer Sekunde ca. 1 Million Schritte ausführen kann, ergibt sich eine Laufzeit von ca. 10^{65} Sekunden oder 10^{58} Jahren. Selbst wenn man diese Laufzeit durch Backtracking verbessert, dürfte sie immer noch im Bereich mehrerer hundert Jahre liegen. Wenn man also nicht Deep Thought aus *Per Anhalter durch die Galaxis* nacheifern will, ist es sinnvoll, von der Nutzung eines solchen Algorithmus abzusehen.

In jedem Fall würden, wenn ein solcher Algorithmus terminiert, wohl weder die Trianguläre noch die Küstenstraße[4] noch existieren. Bei kleineren Beispielen mag dieser Ansatz zwar noch eine Überlegung wert sein. Da der hier vorgestellte Algorithmus solche Beispiele jedoch oft (nahezu) optimal löst, sehe ich es nicht als notwendig an, Backtracking zu implementieren.

Außerdem lässt sich feststellen, dass der Algorithmus Beispiele mit $\sum \varphi_i \leq 180°$ immer optimal löst und somit für diese die optimale Strategie ist. Auch sonst werden meist relativ gute Ergebnisse (die subjektiv meist nah am Optimum zu liegen scheinen) in einer sehr geringen Laufzeit erreicht.

Bezüglich der Beispiele des BwInf ist sichtbar, dass für die Beispiele 1 bis 3 eine vermutlich optimale Lösung errechnet wird, während dies bei den größeren Beispielen nicht mehr der Fall ist. Insbesondere sehe ich Beispiel 5 als verbesserungswürdig an, da dort noch einige kleinere ungenutzte freie Flächen zwischen den Dreiecken sichtbar sind.

Es sind noch einige Verbesserungsmöglichkeiten denkbar, um bessere Ergebnisse zu erreichen:

- Theoretisch könnte es nicht nur ein Subset mit der maximalen Summe, sondern auch mehrere geben. Da sich die einzelnen ausgewählten Dreiecke in den Seitenlängen unterscheiden können, kann bei einem anderen Subset möglicherweise auch ein anderer Gesamtabstand herauskommen. Für eine bessere Lösung könnte man also alle möglichen maximalen Subsets ausprobieren. Da dadurch für das Backtracen jedoch im Extremfall eine exponentielle Laufzeit in $\mathcal{O}(2^n)$ entstehen könnte, habe ich dies nicht implementiert.
- Im Algorithmus werden bisher keine Achsenspiegelungen betrachtet. Dadurch lässt sich jedoch möglicherweise ein besseres Ergebnis erzeugen. Insbesondere kann es nötig sein, ein Dreieck an der Seitenhalbierenden, welche durch den Optimalpunkt verläuft, zu spiegeln.

  Dadurch lässt sich in dem unten dargestellten Fall die längere Seite des schraffierten

[4] Dem Klimawandel sei Dank.

Dreiecks nach oben spiegeln und die kürzere nach unten. Gegebenenfalls lässt sich dadurch ein größerer freier Winkel für den nächsten Anlegeschritt erreichen. Die so erreichten Vorteile haben jedoch nur einen sehr geringen Einfluss, weshalb auch dieser Schritt nicht implementiert wurde.

- Teilweise füllt der Algorithmus nicht alle Lücken zwischen Dreiecken auf. Dies ist insbesondere im unten dargestellten Beispiel zu sehen. Die gepunktete Lücke enthält kein Dreieck, obwohl dort Platz für eines wäre. Dies ist durch den verwendeten Algorithmus bedingt, welcher im Subset-Sum-Schritt die Seitenlängen der Dreiecke nicht berücksichtigt. So kann es vorkommen, dass ein Dreieck mit dem gleichen kleinsten Winkel wie das hier platzierte so lange Seiten hat, dass es überhaupt nicht in diese Lücke passen würde. Aus diesem Grund werden solche Lücken nicht aufgefüllt, was verbesserungswürdig ist.

- Es wäre denkbar, andere Arten von Heuristiken verwenden, insbesondere *evolutionäre Algorithmen* oder *selbstlernende neuronale Netzwerke*. Damit ließen sich möglicherweise noch bessere Ergebnisse erzielen. Diese beiden Arten von Heuristiken sind jedoch in etwa das Äquivalent von Magie in der Informatik. Man kann weder Aussagen darüber treffen, ob und warum die Heuristik vermutlich gute Ergebnisse produziert noch darüber, was während der Ausführung des Algorithmus genau passiert oder welche Laufzeit er (in Landau-Notation) hat.

  Zudem variieren die Ergebnisse solcher Heuristiken zufällig abhängig von den verwendeten Startwerten. Da ich mit dieser Einsendung nicht am trimagischen Turnier, sondern am Bundeswettbewerb Informatik teilnehmen will, habe ich mich gegen die Implementierung einer solchen Heuristik entschieden.

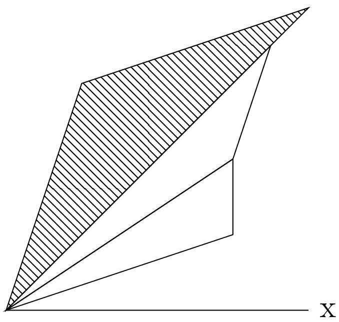

Abbildung 6: Beispiel für die Sinnhaftigkeit von Achsenspiegelungen

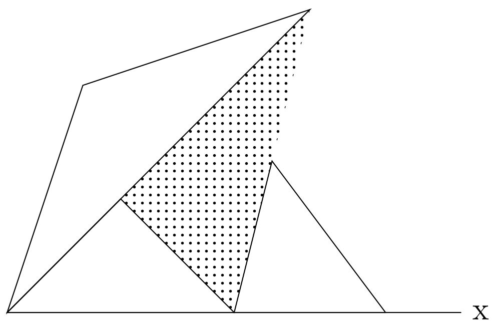

Abbildung 7: Beispiel für nicht aufgefüllte Lücken

1.5 Laufzeitbetrachtung und NP-Vollständigkeit

Die Anzahl der Dreiecke sei n. In der Methode `doAlgorithm()` wird, wie beschrieben, zunächst für alle Dreiecke der kleinste Winkel berechnet, was $\mathcal{O}(n)$ Laufzeit benötigt. Anschließend wird der Subset-Sum-Algorithmus auf der Dreiecksmenge ausgeführt[5], die Teilmenge sortiert und weitere Anweisungen zur Platzierung der Dreiecke in linearer Laufzeit ausgeführt. Dies wird so lange wiederholt, bis keine nicht platzierten Dreiecke mehr übrig sind. Für die Laufzeit eines Schleifendurchlaufs ergibt sich also folgende Gleichung:

[5] Dabei wird als Summe jeweils der aktuelle freie Winkel genommen.

$$T(n) = T_{sum}(n) + \mathcal{O}(k_i \cdot \log k_i) + \mathcal{O}(k_i) \tag{9}$$

Dabei ist n die Anzahl der aktuell noch nicht platzierten Dreiecke, k_i die Anzahl der von Subset Sum ausgewählten Dreiecke und T_{sum} die Laufzeit für Subset Sum.

Die k_i können nach oben durch $\mathcal{O}(n)$ abgeschätzt werden. Die Laufzeit für Subset Sum ist, wie bereits beschrieben:

$$T_{sum}(n) = \mathcal{O}(n * c) = \mathcal{O}(n * \alpha_i) = \mathcal{O}(n) \tag{10}$$

da c durch den freien Winkel α_i bestimmt wird und dieser immer $\leq 180°$ ist, also als Konstante angenommen werden kann.

Für die Gesamtlaufzeit einer Iteration der Schleife erhält man:

$$T(n) = \mathcal{O}(n) + \mathcal{O}(n \cdot \log n) + \mathcal{O}(n) = \mathcal{O}(n \cdot \log n) \tag{11}$$

Nehmen wir an, dass es m Schleifendurchläufe gibt, erhalten wir $\mathcal{O}(m \cdot n \cdot \log n)$. Da die Anzahl der Schleifendurchläufe von der jeweiligen Rückgabe des Subset-Sum-Algorithmus abhängt, kann man nur feststellen, dass $m = \mathcal{O}(n)$ und $m = \Omega(1)$ ist.[6]

Insgesamt erhält man für die Laufzeit also:[7]

$$T_{gesamt}(n) = \mathcal{O}(n^2 \cdot \log n) \text{ und } T_{gesamt}(n) = \Omega(n \cdot \log n) \tag{12}$$

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[8] (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.

[6] Die geringste Anzahl tritt auf, wenn die Summe aller kleinsten Winkel $\leq 180°$ ist.

[7] Da das n nicht für jede Iteration der Schleife gleich ist, sondern kleiner wird, handelt es sich bei der Laufzeit in O-Notation nur um eine grobe Abschätzung nach oben. Die Laufzeit wird also in den meisten Fällen eher $\mathcal{O}(n \cdot \log n)$ sein.

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

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.[9]

Abschließend möchte ich bemerken, dass das Problem viel einfacher zu lösen wäre, wenn sich die Trianguläre einfach für rechteckige Grundstücksformen entscheiden würden. Bei diesen ließe sich einfach die kürzere Seite an die Küstenstraße anlegen. Die Beschreibung der Trianguläre als „seltsam“ ist also offensichtlich gerechtfertigt.

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`

[5] Wikipedia-Artikel zum Subset-Sum-Problem, `https://en.wikipedia.org/wiki/Subset_sum_problem`

[6] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford: *Introduction to Algorithms*, Third Edition, MIT Press, 2009 (insbesondere Kapitel 34 zur NP-Vollständigkeit und Kapitel 34 und 35 zum Subset-Sum-Problem)

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[10],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
```

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

[10] `https://github.com/hbristow/argparse`

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`

`triangles.cpp` enhält drei im Algorithmus oft benötigte Klassen:

- Die `Point`-Klasse, die einen Punkt im gegebenen zweidimensionalen Koordinatensystem repräsentiert.
- Die `Vektor`-Klasse, die einen Vektor in ebendiesem Koordinatensystem repräsentiert.[11] Außerdem gibt es eine Funktion, die den Betrag des Vektors berechnet.
- Die `Triangle`-Klasse, die ein Dreieck repräsentiert. Dabei werden eine ID für die Ausgabe, die drei Eckpunkte sowie die Vektoren zwischen ihnen gespeichert. Weiterhin kann mittels eine Funktion die längste an einem Punkt des Dreieck anliegende Seite bestimmt werden, was für das Sortieren der Dreiecke vor der Platzierung notwendig ist.

Weiterhin gibt es folgende Methoden:

Funktion	Beschreibung
`addVektor()`	Addiert einen Vektor zu einem Punkt.
`dotProduct()`	Berechnet das Skalarprodukt zweier Vektoren. ($a \cdot b = a.x \cdot b.x + a.y \cdot b.y$)
`angle()`	Berechnet den Winkel zwischen zwei Vektoren wie in der Lösungsidee beschrieben.
`locateSmallestAngle()`	Findet den kleinsten Winkel eines Dreiecks und den dazugehörigen Punkt.
`rotate_tri()`	Rotiert einen Punkt p um einen Winkel α um das Drehzentrum c. (siehe Lösungsidee)
`atan_angle()`	Berechnet den Winkel, den ein Vektor zwischen zwei Punkten zur positiven x-Achse besitzt. `atan180()` und `atan360()` sind Shortcuts für das Subtrahieren des Winkels von 180 bzw. 360 Grad.
`ccw()`	Berechnet, ob ein Punkt gegen den Uhrzeigersinn bezüglich des Vektors zwischen zwei anderen Punkten liegt (siehe Lösungsidee).
`findAngleCalcPoint()`	Findet den Punkt, der für die Bestimmung des Winkels zur x-Achse beim Drehen maßgeblich ist. Dies ist immer der Punkt, der gegen Uhrzeigersinn bezüglich des Vektors aus dem Optimalpunkt und dem dritten Punkt des Dreiecks liegt.

[11] Hier wurde nicht die englische Bezeichnung verwendet, um Verwechslungen mit der STL-Klasse `vector` auszuschließen.

2.2.3 Die Datei `triangleAlgorithm.cpp`

`triangleAlgorithm.cpp` enthält den eigentlichen Algorithmus, der aus folgenden Methoden besteht:

Funktion	Beschreibung
`getSubsetsRec()`	Ist für das Backtracen des maximalen Subsets aus dem DP-Array verantwortlich.
`subsetSum()`	Berechnet die Werte des DP-Arrays nach der in der Lösungsidee angegebenen Rekursionsgleichung.
`lengthSortFunc()`	Wird als Vergleichsfunktion beim Sortieren der Dreiecke eines Subsets verwendet. Dabei wird die Länge der längsten am Optimalpunkt anliegenden Seite verglichen.
`triangleSortFunc()`	Sortiert die platzierten Dreiecke nach dem x-Wert ihres Mittelpunkts ($\frac{1}{3} \cdot \sum x_i$). Dadurch können die Dreiecke in einer Reihenfolge von links nach rechts ausgegeben werden.
`deleteUsedTriangles()`	Löscht das aktuell platzierte Subset aus der Liste der noch nicht platzierten Dreiecke und allen damit verbundenen Listen (kleinster Winkel usw.). Fügt außerdem das gesamte Subset der Liste der platzierten Dreiecke hinzu.
`translateAndRotateToAxis()`	Verschiebt die Dreiecke eines Subsets so, dass ihre Optimalpunkte auf den gleichen Punkt auf der x-Achse abgebildet werden. Rotiert die Dreiecke dann so, dass alle Dreiecke mit einer Seite auf der x-Achse liegen und sich oberhalb dieser befinden.
`rotateToPosition..()`	Rotiert die Dreiecke eines Subsets so von der x-Achse weg, dass sie sich nicht mehr überlappen. Anschließend werden die Dreiecke an die bisherige Anordnung „herangedreht“ , um Platz zu sparen und eine bessere Anordnung zu erzeugen. Die Version `rotateToPositionRight()` fügt auf der rechten Seite an und `rotateToPositionLeft()` auf der linken Seite, wobei auf der linken Seite aufgrund der Heuristik normalerweise nicht angefügt wird.
`calculateDistance()`	Berechnet den Gesamtabstand der Dreiecksanordnung. Dabei werden der linkeste und der rechteste Punkt auf der x-Achse, die für die Berechnung zählen, bestimmt.
`moveToRightOfY()`	Verschiebt die Anordnung so, dass der linkeste Punkt genau auf der y-Achse liegt.
`doAlgorithm()`	Setzt den in der Lösungsidee vorgeschlagenen Algorithmus um, wobei die Dreiecksanordnung vom willkürlich gewählten Punkt (300|0) aufgebaut wird.

Außerdem werden in dieser Datei mehrere globale Listen (`vector`) benutzt. Dies sind:

- `bestPointIndex`: Speichert für jedes Dreieck den Index des Optimalpunktes in der internen Punkteliste des Dreiecks.
- `bestAngle` und `bestAngleDouble`: Speichern für jedes Dreieck den kleinsten Winkel

in Radians. `bestAngle` speichert dabei den Winkel als Integer, der dem Winkel als Double multipliziert mit 10000 entspricht.

- `sol`: Speichert die Indexe der durch den Subset-Sum-Algorithmus ermittelten Teilmenge.
- `triangles` und `placedTriangles` speichern die (nicht) platzierten Dreiecke.

3 Beispiele

Laufzeiten

Beispiel	Dreiecksanzahl	Laufzeit (ca.)
dreiecke1.txt	5	8 Millisekunden
dreiecke2.txt	5	8 Millisekunden
dreiecke3.txt	12	11 Millisekunden
dreiecke4.txt	23	10 Millisekunden
dreiecke5.txt	37	27 Millisekunden

Die Laufzeiten wurden mit dem Linux-Befehl `time` bestimmt. Dazu wurde ein PC mit einem Intel Core i7 und 8 GB RAM verwendet und das Programm mit der Option `-O3` kompiliert. Sie sind demzufolge nur grobe Orientierungswerte, die von der verwendeten Hardware abhängen. Es lässt sich aber erkennen, dass der Algorithmus sehr effizient ist und selbst größere Beispiele extrem schnell lösen kann, auch wenn die Ergebnisse nicht immer optimal sind.

3.1 Beispiel 1

Abbildung 8: Die Dreiecksanordnung für das Beispiel 1

Ausgabe für Beispiel 1

```
Gesamtabstand: 142.874 Meter
Platzierung der Dreiecke:
D1 142.874 0.000 71.514 123.777 0.000 0.133
D2 142.874 0.000 214.350 123.710 71.514 123.777
D3 142.874 0.000 285.748 0.000 214.350 123.710
D4 285.748 0.000 357.166 123.744 214.330 123.744
D5 285.748 0.000 428.622 0.067 357.166 123.744
```

3.2 Beispiel 2

Abbildung 9: Die Dreiecksanordnung für das Beispiel 2

Ausgabe für Beispiel 2

```
Gesamtabstand: 0.000 Meter
Platzierung der Dreiecke:
D2 68.859 0.000 0.000 568.790 149.119 552.560
D4 68.859 0.000 143.086 511.025 290.582 483.730
D3 68.859 0.000 317.727 144.822 356.798 0.000
D5 68.859 0.000 277.780 455.799 411.536 387.909
D1 68.859 0.000 384.061 356.807 465.090 230.576
```

3.3 Beispiel 3

Abbildung 10: Die Dreiecksanordnung für das Beispiel 3

Ausgabe für Beispiel 3

```
Gesamtabstand: 92.418 Meter
Platzierung der Dreiecke:
D12 0.000 79.899 283.332 0.000 82.843 0.187
D4 23.679 73.222 283.332 0.000 83.931 119.034
D9 283.332 0.000 150.143 79.508 109.056 181.669
D11 191.283 185.410 132.924 156.788 283.332 0.000
D6 283.332 0.000 269.111 200.718 198.478 170.918
D1 331.106 183.896 269.162 199.988 283.332 0.000
D10 283.332 0.000 372.300 84.201 309.376 100.248
D8 375.750 0.000 333.615 47.588 283.332 0.000
D2 375.750 190.066 375.750 0.000 475.294 239.864
D5 508.613 150.573 375.750 0.000 448.578 175.488
D7 509.889 5.153 375.750 0.000 478.618 61.563
D3 375.750 0.000 513.044 82.166 495.431 135.634
```

3.4 Beispiel 4

Abbildung 11: Die Dreiecksanordnung für das Beispiel 4

Ausgabe für Beispiel 4

```
Gesamtabstand: 268.280 Meter
Platzierung der Dreiecke:
D12 0.000 0.082 195.000 0.000 0.019 45.082
D17 195.000 0.000 1.017 44.852 47.876 62.886
D11 12.387 78.055 195.000 0.000 71.585 71.677
```

```
D14 195.000 0.000 23.537 99.582 66.498 103.500
D15 82.962 122.489 195.000 0.000 40.578 124.378
D10 106.011 169.059 195.000 0.000 65.279 141.822
D6 139.500 158.571 129.128 125.143 195.000 0.000
D16 148.281 133.482 238.382 90.100 195.000 0.000
D1 269.917 74.917 223.198 58.565 195.000 0.000
D5 260.000 65.000 260.000 0.000 195.000 0.000
D9 260.000 0.000 308.042 193.039 278.728 141.486
D4 327.055 120.369 301.298 165.938 260.000 0.000
D20 332.440 68.392 260.000 0.000 306.233 82.991
D23 310.899 48.055 356.208 0.064 260.000 0.000
D13 356.272 0.000 310.275 189.498 354.005 200.112
D18 354.017 199.088 383.600 163.735 356.272 0.000
D7 434.139 165.537 383.930 165.711 356.272 0.000
D3 356.272 0.000 444.833 105.555 427.783 152.024
D21 356.272 0.000 434.153 54.401 442.804 103.137
D2 462.215 1.066 422.106 23.788 356.272 0.000
D22 466.641 77.095 445.618 32.284 356.272 0.000
D8 530.943 157.044 471.248 182.764 463.280 0.000
D19 463.280 0.000 542.271 107.519 500.871 87.247
```

3.5 Beispiel 5

Abbildung 12: Die Dreiecksanordnung für das Beispiel 5

Ausgabe für Beispiel 5

```
Gesamtabstand: 769.468 Meter
Platzierung der Dreiecke:
D4 4.494 62.010 0.000 0.044 102.956 0.000
D7 43.435 94.124 26.895 47.902 102.956 0.000
D1 44.680 92.156 102.956 0.000 89.426 56.453
D12 102.956 0.000 134.713 69.112 81.643 88.926
D8 146.234 0.000 152.843 20.033 102.956 0.000
D13 135.569 70.975 102.956 0.000 182.676 32.013
D25 218.418 39.326 190.617 38.990 146.234 0.000
D15 210.188 0.012 214.824 37.369 146.234 0.000
D27 210.200 0.000 253.800 0.019 245.670 25.979
D5 283.831 53.930 236.344 59.837 210.200 0.000
D34 210.200 0.000 291.793 186.750 235.426 120.721
D21 350.466 57.789 290.789 66.432 253.820 0.000
D18 339.758 51.387 253.820 0.000 325.129 0.016
D37 366.358 62.622 373.306 109.916 325.145 0.000
D10 382.743 37.032 383.080 88.031 325.145 0.000
D6 374.836 31.949 325.145 0.000 423.231 0.216
D29 392.542 67.749 423.447 0.000 418.690 111.899
D20 423.447 0.000 473.272 68.895 419.254 98.628
D32 500.603 0.018 469.534 63.726 423.447 0.000
```

```
D16 500.620 0.000 487.649 92.438 461.798 97.800
D17 500.620 0.000 511.065 71.105 487.510 93.424
D14 500.620 0.000 567.366 29.069 511.201 72.028
D30 544.446 19.087 500.620 0.000 571.684 0.032
D24 571.715 0.000 605.331 111.279 555.257 109.996
D23 571.715 0.000 627.421 39.203 602.101 100.587
D36 646.722 0.268 631.161 41.836 571.715 0.000
D19 660.964 90.165 624.047 61.283 646.990 0.000
D11 660.180 85.106 646.990 0.000 681.538 41.187
D9 721.996 0.263 692.090 53.767 646.990 0.000
D2 722.259 0.000 747.741 103.309 681.109 73.339
D26 722.259 0.000 745.562 94.477 795.450 35.653
D31 799.316 0.479 794.739 35.307 722.259 0.000
D35 836.794 55.236 790.632 75.187 799.795 0.000
D33 869.929 2.496 836.378 54.614 799.795 0.000
D3 823.889 75.242 872.424 0.000 870.851 80.589
D22 933.676 55.011 870.823 82.021 872.424 0.000
D28 928.000 20.032 897.055 22.121 872.424 0.000
```

---

4 Quellcode

```cpp
#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 längste an einem Punkt anliegende Dreiecksseite
    double longestLength(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 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 Datei `triangles.cpp`, die die Klassen `Triangle`, `Vektor` und `Point` und nützliche Hilfsfunktionen für den eigentlichen Algorithmus enthält

```
#include<bits/stdc++.h>
#include"triangles.cpp"
#define EPSILON 10

using namespace std;

vector<int> bestPointIndex; // Punkt mit dem kleinsten Winkel (0,1 oder 2)
vector<int> bestAngle; // kleinster Winkel als int (für Subset Sum)
vector<double> bestAngleDouble; // kleinster Winkel als double
vector<int> sol; // Ausgabe des Subset Sum Algorithmus
vector<Triangle> triangles; // noch nicht platzierte Dreiecke
vector<Triangle> placedTriangles; // platzierte Dreiecke

// Backtracen der Subset Sum Lösung
// gibt nur ein mögliches Subset zurück
bool getSubsetsRec(vector<int> arr, int i, int sum, vector<int>& p,
↪   vector<vector<bool>> &dp) {
    // erstes Element erreicht, aber Summe nicht 0
    // -> dp[0][sum] muss true sein, damit Subset möglich ist
    if (i == 0 && sum != 0 && dp[0][sum]) {
        p.push_back(i);
        sol = p;
```

```
        return true;
    }

    // erstes Element erreicht und Summe 0 -> möglich
    if (i == 0 && sum == 0) {
        sol = p;
        return true;
    }

    // wenn Summe ohne aktuelles Element ereicht werden kann
    if (dp[i-1][sum]) {
        vector<int> b = p;
        if(getSubsetsRec(arr, i-1, sum, b,dp)){
            return true;
        }
    }

    // wenn Summe mit aktuellem Element erreicht werden kann
    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));

    // Summe 0 immer erreichbar
    for (int i=0; i<n; ++i) {
        dp[i][0] = true;
    }

    // Summe set[0] kann mit erstem Element erreicht werden
    if (set[0] <= sum)
       dp[0][set[0]] = true;

    // DP-Array nach Rekursionsgleichung füllen
    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];
        }
    }

    // größte Summe, die erreicht werden kann
    int best = sum;
    for(;best>=0;best--){
        if(dp[n-1][best]) break;
    }
```

```

    // Backtracen
    vector<int> p;
    getSubsetsRec(set, n-1, best, p,dp);
}

// Sortieren einer Teilmenge von Dreiecken
// nach der längsten Seite, die am auf der x-Achse liegenden
// Punkt des Dreiecks (mit dem kleinsten Winkel) anliegt.
// "Kleine" Dreiecke sind dadurch weiter rechts.
bool lengthSortFunc(const int t1, const int t2){
    return triangles[t1].longestLength(bestPointIndex[t1]) <
     ↪  triangles[t2].longestLength(bestPointIndex[t2]);
}

// Sortieren der Dreiecke anhand ihres Mittelpunkts (x-Koordinate)
// zur geordneten Ausgabe
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;
}

// aktuelles Subset aus den noch nicht platzierten Dreiecken löschen
// und zu den platzierten hinzufügen
void deleteUsedTriangles(){
    // in absteigender Reihenfolge der Indexe -> sonst: Segfault
    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);
    }
}

// Dreiecke eines Subsets so verschieben, dass sie
// mit dem "besten" Punkt auf der x-Achse liegen
// und so rotieren, dass sie direkt auf der Achse liegen
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);
        }
```

```
    }
}

// für Anfügen auf der rechten Seite
// Dreiecke so rotieren, dass sie sich nicht überlappen
// anschließend an die bisherige Anordnung "randrehen"
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);
            }
        }
    }
}

// für Anfügen auf der linken Seite
// wird bedingt durch Sortierung der Dreiecke normalerweise nie benutzt
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 -
            ↪ bestAngleDouble[sol[i]]);
        }
        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));
        }
    }
}

// linkesten und rechtesten Punkt finden,
// die zur Berechnung des Gesamtabstands herangezogen werden
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};
}

// Anordnung so verschieben, dass der "linkeste" Punkt
// auf der y-Achse liegt
void moveToRightOfY(){
    double minx = 100000;
    for(auto t: placedTriangles){
        for(auto pnt: t.points){
            if(pnt.x < minx) minx = pnt.x;
        }
    }
    for(auto &t: placedTriangles){
        for(auto &pnt: t.points){
            pnt.x -= minx;
        }
    }
}

// Hauptmethode des Algorithmus
pair<vector<Triangle>,double> doAlgorithm(vector<Triangle> i_triangles){
    triangles = i_triangles;
    Point centerPoint = Point(300,0); // erster Anlegepunkt (willkürlich
    ↪  gewählt)

    // kleinsten Winkel und entsprechenden Punkt der Dreiecke berechnen
    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);
    }

    // Subset-Sum ausführen und nach Dreiecke nach Seitenlänge ordnen
    subsetSum(bestAngle,(int) floor(10000*M_PI));
    sort(sol.begin(),sol.end(),lengthSortFunc);
```

```

    // Dreiecke passend verschieben und rotieren
    translateAndRotateToAxis(centerPoint);
    rotateToPositionRight(centerPoint,M_PI,false);
    deleteUsedTriangles();

    // solange nicht alle Dreiecke platziert
    double leftmost, rightmost;
    while(!triangles.empty()){
        // Punkte für die Berechnung der Gesamtdistanz
        tie(leftmost,rightmost) = calculateDistance();

        // min./max. Punkt finden, an dem angelegt werden kann
        // ist ein Punkt eines platzierten Dreiecks
        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) ? pnt.x - pnt.y : pnt.x;}
                    if(pnt.x > maxx_axis) {maxx_axis = (pnt.y > 0) ? pnt.x + pnt.y : pnt.x;}
                }
            }
        }

        Point newCenterRight = Point(maxx_axis,0);
        Point newCenterLeft = Point(minx_axis,0);

        // freien Winkel (links/rechts) finden
        // wird durch den Punkt bestimmt, der den geringsten Winkel/atan vom Anlegepunkt hat
        Point minpnt_above, maxpnt_above;
        double angle_left = M_PI, angle_right = M_PI;
        for(auto t: placedTriangles){
            for(auto pnt: t.points){
                if(pnt.y == 0) continue;
                if(atan_angle(newCenterRight,pnt) < angle_right) { angle_right = atan_angle(newCenterRight,pnt); maxpnt_above = pnt;}
                if(atan180(newCenterLeft,pnt) < angle_left) {angle_left = atan_angle(newCenterLeft,pnt); minpnt_above = pnt;}
            }
        }

        // links oder rechts platzieren - was ist besser?
        // normalerweise rechts
        bool left = false;
        double free_angle = 0;
        if(leftmost - newCenterLeft.x < newCenterRight.x - rightmost){
            //place left
            free_angle = atan180(newCenterLeft,minpnt_above);
            left = true;
        } else {
```

```
            //place right
            free_angle = atan_angle(newCenterRight,maxpnt_above);
        }

        // Subset Sum + Sortieren (wie oben)
        subsetSum(bestAngle,(int) floor(10000*free_angle));
        sort(sol.begin(),sol.end(),lengthSortFunc);

        // Links oder rechts platzieren
        // Links zuerst Reihenfolge umdrehen
        if(left){
            reverse(sol.begin(),sol.end());
            translateAndRotateToAxis(newCenterLeft);
            rotateToPositionLeft(newCenterLeft,free_angle);
        } else {
            translateAndRotateToAxis(newCenterRight);
            rotateToPositionRight(newCenterRight,free_angle,true);
        }
        deleteUsedTriangles();
    }

    // Distanz und zur Anzeige verschieben/sortieren
    auto dpair = calculateDistance();
    double dist = abs(dpair.second - dpair.first);
    moveToRightOfY();
    sort(placedTriangles.begin(),placedTriangles.end(),triangleSortFunc);

    return {placedTriangles,dist};
}
```

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