

Aufgabe 2

„Dreiecksbeziehungen“- Dokumentation

37. Bundeswettbewerb Informatik 2018/19 - 2. Runde

Lukas Rost

Teilnahme-ID: 48125

29. April 2019

Inhaltsverzeichnis

1	Lösungsidee	1
1.1	Mathematische Präzisierung der Aufgabenstellung	1
1.2	Wahl eines geeigneten Algorithmus	2
1.3	Beschreibung der Lösungsidee	3
1.3.1	Beobachtungen bezüglich einer guten Lösung	3
1.3.2	Subset Sum und ein DP-Algorithmus	3
1.3.3	Der Algorithmus zur Platzierung der Dreiecke	5
1.3.4	Implementierte Verbesserungen	5
1.4	Optimalität des Algorithmus und Verbesserungsmöglichkeiten	5
1.5	Laufzeitbetrachtung und NP-Vollständigkeit	5
2	Umsetzung	6
2.1	Allgemeine Hinweise zur Benutzung	6
2.2	Struktur des Programms und Implementierung der Algorithmen	6
2.2.1	Die Datei <code>main.cpp</code>	6
2.2.2	Die Datei <code>triangles.cpp</code>	6
2.2.3	Die Datei <code>triangleAlgorithm.cpp</code>	6
3	Beispiele	6
3.1	Beispiel 1	7
3.2	Beispiel 2	7
3.3	Beispiel 3	8
3.4	Beispiel 4	9
3.5	Beispiel 5	9
4	Quellcode	10

1 Lösungsidee

1.1 Mathematische Präzisierung der Aufgabenstellung

Bei der Eingabe handelt es sich um eine Menge $D = \{d_1, \dots, d_n\}$ 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:

$$\forall (x, y) \in d : y \geq 0 \wedge x \geq 0 \quad (1)$$

Alle Punkte müssen also über oder auf der x-Achse sowie rechts oder auf der y-Achse liegen.

- Für jedes $d \in D'$ gilt:

$$\exists (x, y) \in d : y = 0 \quad (2)$$

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 Achsenspiegelungen¹ hervorgehen.
- Für jedes $d \in D'$ und jedes $e \in D'$ gilt:

$$d \cap e = \emptyset \quad (3)$$

$d \cap e$ 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 :

$$\max_{d_i \in D'} \min_{d_j \in D'} \min_{n \in N_i} \min_{m \in N_j} |n.x - m.x| \quad (4)$$

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.

¹und 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 $\sum_{i=1}^n \varphi_i < 180^\circ$, 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.*

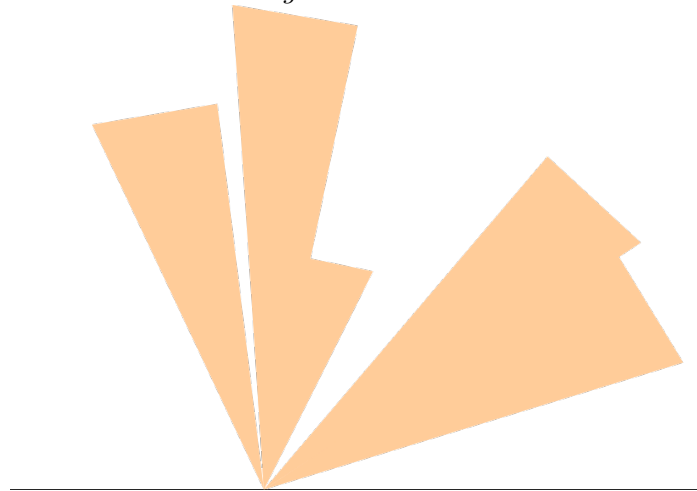


Abbildung 1: Optimale Anordnung für eine Summe kleiner 180°

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 $\sum \varphi_i < 180^\circ$ 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 $I = \{w_1, \dots, w_n\} (w_i \in \mathbb{Z})$ 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:

$$S = \arg \max_{S \subseteq 2^I} \sum_{w_j \in S} w_j \quad (5)$$

$$\sum_{w_j \in S} \leq c \quad (6)$$

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:

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

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 vierten 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>

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

³Auch wenn eine Beziehung zwischen 3-SAT und *Dreiecken* 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`-Library⁴, 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

Laufzeiten

Beispiel	Dreiecksanzahl	Laufzeit (ca.)
<code>dreiecke1.txt</code>	5	8 Millisekunden
<code>dreiecke2.txt</code>	5	8 Millisekunden
<code>dreiecke3.txt</code>	12	11 Millisekunden
<code>dreiecke4.txt</code>	23	10 Millisekunden
<code>dreiecke5.txt</code>	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.

⁴<https://github.com/hbristow/argparse>

3.1 Beispiel 1

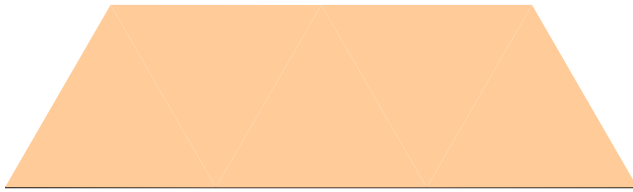


Abbildung 2: Die Dreiecksanordnung für das Beispiel 1

Ausgabe für Beispiel 1

```
1 Gesamtabstand: 142.874 Meter
2 Platzierung der Dreiecke:
3 D1 142.874 0.000 71.514 123.777 0.000 0.133
4 D2 142.874 0.000 214.350 123.710 71.514 123.777
5 D3 142.874 0.000 285.748 0.000 214.350 123.710
6 D4 285.748 0.000 357.166 123.744 214.330 123.744
7 D5 285.748 0.000 428.622 0.067 357.166 123.744
```

3.2 Beispiel 2

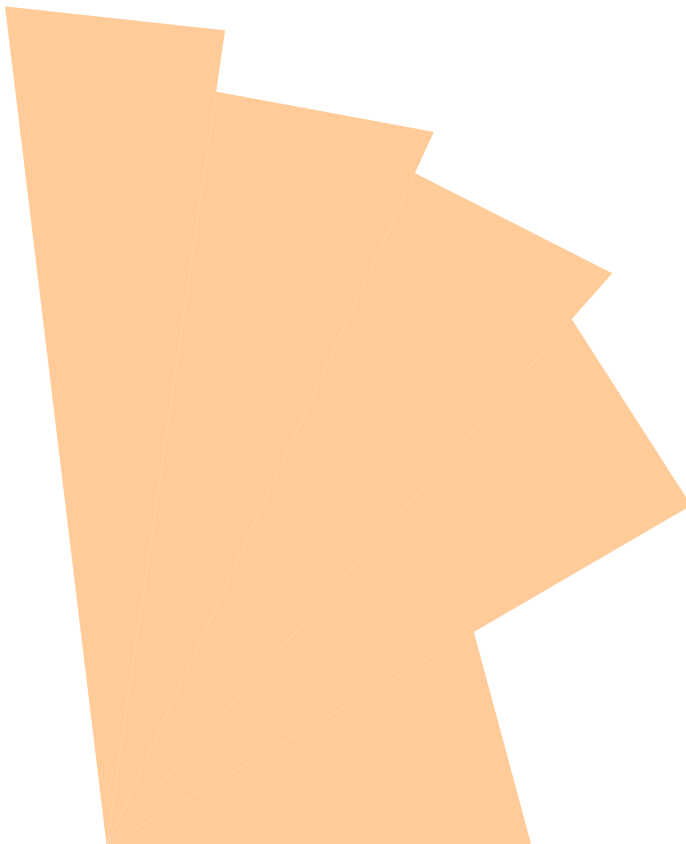


Abbildung 3: Die Dreiecksanordnung für das Beispiel 2

Ausgabe für Beispiel 2

```

1 Gesamtabstand: 0.000 Meter
2 Platzierung der Dreiecke:
3 D2 68.859 0.000 0.000 568.790 149.119 552.560
4 D4 68.859 0.000 143.086 511.025 290.582 483.730
5 D3 68.859 0.000 317.727 144.822 356.798 0.000
6 D5 68.859 0.000 277.780 455.799 411.536 387.909
7 D1 68.859 0.000 384.061 356.807 465.090 230.576

```

3.3 Beispiel 3



Abbildung 4: Die Dreiecksanordnung für das Beispiel 3

Ausgabe für Beispiel 3

```

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

```

3.4 Beispiel 4

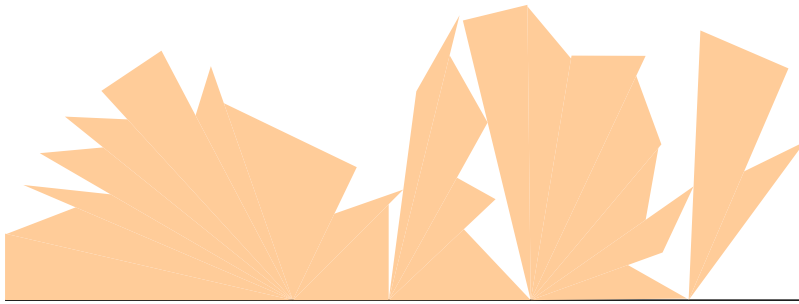


Abbildung 5: Die Dreiecksanordnung für das Beispiel 4

Ausgabe für Beispiel 4

```
1 Gesamtabstand: 268.280 Meter
2 Platzierung der Dreiecke:
3 D12 0.000 0.082 195.000 0.000 0.019 45.082
4 D17 195.000 0.000 1.017 44.852 47.876 62.886
5 D11 12.387 78.055 195.000 0.000 71.585 71.677
6 D14 195.000 0.000 23.537 99.582 66.498 103.500
7 D15 82.962 122.489 195.000 0.000 40.578 124.378
8 D10 106.011 169.059 195.000 0.000 65.279 141.822
9 D6 139.500 158.571 129.128 125.143 195.000 0.000
10 D16 148.281 133.482 238.382 90.100 195.000 0.000
11 D1 269.917 74.917 223.198 58.565 195.000 0.000
12 D5 260.000 65.000 260.000 0.000 195.000 0.000
13 D9 260.000 0.000 308.042 193.039 278.728 141.486
14 D4 327.055 120.369 301.298 165.938 260.000 0.000
15 D20 332.440 68.392 260.000 0.000 306.233 82.991
16 D23 310.899 48.055 356.208 0.064 260.000 0.000
17 D13 356.272 0.000 310.275 189.498 354.005 200.112
18 D18 354.017 199.088 383.600 163.735 356.272 0.000
19 D7 434.139 165.537 383.930 165.711 356.272 0.000
20 D3 356.272 0.000 444.833 105.555 427.783 152.024
21 D21 356.272 0.000 434.153 54.401 442.804 103.137
22 D2 462.215 1.066 422.106 23.788 356.272 0.000
23 D22 466.641 77.095 445.618 32.284 356.272 0.000
24 D8 530.943 157.044 471.248 182.764 463.280 0.000
25 D19 463.280 0.000 542.271 107.519 500.871 87.247
```

3.5 Beispiel 5



Abbildung 6: Die Dreiecksanordnung für das Beispiel 5

Ausgabe für Beispiel 5

```
1 Gesamtabstand: 769.468 Meter
2 Platzierung der Dreiecke:
3 D4 4.494 62.010 0.000 0.044 102.956 0.000
4 D7 43.435 94.124 26.895 47.902 102.956 0.000
5 D1 44.680 92.156 102.956 0.000 89.426 56.453
6 D12 102.956 0.000 134.713 69.112 81.643 88.926
7 D8 146.234 0.000 152.843 20.033 102.956 0.000
8 D13 135.569 70.975 102.956 0.000 182.676 32.013
9 D25 218.418 39.326 190.617 38.990 146.234 0.000
10 D15 210.188 0.012 214.824 37.369 146.234 0.000
11 D27 210.200 0.000 253.800 0.019 245.670 25.979
12 D5 283.831 53.930 236.344 59.837 210.200 0.000
13 D34 210.200 0.000 291.793 186.750 235.426 120.721
14 D21 350.466 57.789 290.789 66.432 253.820 0.000
15 D18 339.758 51.387 253.820 0.000 325.129 0.016
16 D37 366.358 62.622 373.306 109.916 325.145 0.000
17 D10 382.743 37.032 383.080 88.031 325.145 0.000
18 D6 374.836 31.949 325.145 0.000 423.231 0.216
19 D29 392.542 67.749 423.447 0.000 418.690 111.899
20 D20 423.447 0.000 473.272 68.895 419.254 98.628
21 D32 500.603 0.018 469.534 63.726 423.447 0.000
22 D16 500.620 0.000 487.649 92.438 461.798 97.800
23 D17 500.620 0.000 511.065 71.105 487.510 93.424
24 D14 500.620 0.000 567.366 29.069 511.201 72.028
25 D30 544.446 19.087 500.620 0.000 571.684 0.032
26 D24 571.715 0.000 605.331 111.279 555.257 109.996
27 D23 571.715 0.000 627.421 39.203 602.101 100.587
28 D36 646.722 0.268 631.161 41.836 571.715 0.000
29 D19 660.964 90.165 624.047 61.283 646.990 0.000
30 D11 660.180 85.106 646.990 0.000 681.538 41.187
31 D9 721.996 0.263 692.090 53.767 646.990 0.000
32 D2 722.259 0.000 747.741 103.309 681.109 73.339
33 D26 722.259 0.000 745.562 94.477 795.450 35.653
34 D31 799.316 0.479 794.739 35.307 722.259 0.000
35 D35 836.794 55.236 790.632 75.187 799.795 0.000
36 D33 869.929 2.496 836.378 54.614 799.795 0.000
37 D3 823.889 75.242 872.424 0.000 870.851 80.589
38 D22 933.676 55.011 870.823 82.021 872.424 0.000
39 D28 928.000 20.032 897.055 22.121 872.424 0.000
```

4 Quellcode

```
1 #include<bits/stdc++.h>
2
3 using namespace std;
4
5 // Klasse für einen Punkt (x/y-Koordinate)
6 class Point{
7     public:
8
9     double x;
10    double y;
11
```

```
12     Point(double _x, double _y){
13         x = _x;
14         y = _y;
15     }
16
17     Point(){
18         x = 0;
19         y = 0;
20     }
21 };
22
23 // Klasse für einen Vektor im  $\mathbb{R}^2$ 
24 class Vektor{
25     public:
26
27     double x;
28     double y;
29
30     Vektor(double _x, double _y){
31         x = _x;
32         y = _y;
33     }
34
35     // Vektor zwischen zwei Punkten
36     Vektor(Point a, Point b){
37         x = b.x - a.x;
38         y = b.y - a.y;
39     }
40
41     Vektor(){
42         x = 0;
43         y = 0;
44     }
45
46     // Betrag/Laenge des Vektors
47     double betrag(){
48         return sqrt(x * x + y * y);
49     }
50 };
51
52 // Klasse für ein Dreieck
53 class Triangle{
54     public:
55
56     // Punkte + Vektoren der Seiten + Längen dieser
57     vector<Point> points;
58     vector<Vektor> vektoren;
59     vector<double> lengths;
60     int id;
61
62     Triangle(Point p1, Point p2, Point p3, int idd){
63         points = {p1,p2,p3};
64         id = idd;
```

```
65     reGenVectors();
66     for(int i=0;i<=2;i++){
67         lengths.push_back(vektoren[i].betrag());
68     }
69 }
70
71 // Vektoren nach Drehung etc. neu erstellen
72 void reGenVectors(){
73     Vektor p1p2 = Vektor(points[0],points[1]);
74     Vektor p2p3 = Vektor(points[1],points[2]);
75     Vektor p3p1 = Vektor(points[2],points[0]);
76     vektoren = {p1p2,p2p3,p3p1};
77 }
78
79 // Berechnung längste an einem Punkt anliegende Dreiecksseite
80 double longestLength(int bestPoint){
81     switch(bestPoint) {
82     case 0: if(lengths[0] > lengths[2]){
83         return lengths[0];
84     } else {
85         return lengths[2];
86     }
87     break;
88     case 1: if(lengths[0] > lengths[1]){
89         return lengths[0];
90     } else {
91         return lengths[1];
92     }
93     break;
94     case 2: if(lengths[1] > lengths[2]){
95         return lengths[1];
96     } else {
97         return lengths[2];
98     }
99     break;
100    default: return 0;
101           break;
102    }
103 }
104 };
105
106 // Vektor zu Punkt addieren
107 Point addVektor(Point &p, Vektor &v){
108     p.x += v.x;
109     p.y += v.y;
110     return p;
111 }
112
113 // Skalarprodukt zweier Vektoren
114 double dotProduct(Vektor &v1, Vektor &v2){
115     return v1.x * v2.x + v1.y * v2.y;
116 }
117
```

```
118 // Winkel zwischen zwei Vektoren
119 // allgemein bekannte Cosinus-Formel
120 double angle(Vektor &v1, Vektor &v2){
121     double cosvalue = dotProduct(v1,v2)/(v1.betrag()*v2.betrag());
122     return acos(abs(cosvalue));
123 }
124
125 // kleinsten Winkel und anliegenden Punkt eines Dreiecks bestimmen
126 pair<int,double> locateSmallestAngle(Triangle t){
127     double bestangle = M_PI;
128     int pointindex;
129     for(size_t i=0;i<=2;i++){
130         double thisangle = angle(t.vektoren[i],t.vektoren[(i+1)%3]);
131         if(thisangle < bestangle){
132             bestangle = thisangle;
133             pointindex = (i+1)%3;
134         }
135     }
136     return {pointindex,bestangle};
137 }
138
139 // Punkt mithilfe einer Drehmatrix um ein Zentrum rotieren
140 void rotate_tri(Point center, Point &p, double angle){
141     double sinus = sin(angle);
142     double cosinus = cos(angle);
143
144     p.x -= center.x;
145     p.y -= center.y;
146
147     double xnew = p.x * cosinus - p.y * sinus;
148     double ynew = p.x * sinus + p.y * cosinus;
149
150     p.x = xnew + center.x;
151     p.y = ynew + center.y;
152 }
153
154 // Winkel zur positiven x-Achse mit atan2
155 double atan_angle(Point center, Point p){
156     double dx = p.x - center.x;
157     double dy = p.y - center.y;
158
159     double angle = atan2(dy,dx);
160     if(dy < 0){
161         angle += 2 * M_PI;
162     }
163
164     return angle;
165 }
166
167 // 360 Grad minus diesen Winkel
168 // (zum anfänglichen Drehen, so dass Dreieck auf x-Achse liegt)
169 double atan360(Point center, Point p){
170     return 2 * M_PI - atan_angle(center,p);
171 }
```

```

171 }
172
173 // ähnlich (Berechnen des freien Winkels links)
174 double atan180(Point center, Point p){
175     return M_PI - atan_angle(center,p);
176 }
177
178 // Lage Punkt c von ab aus
179 // CCW: < 0, CW: >0
180 double ccw(Point a, Point b, Point c){
181     return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
182 }
183
184 // Punkt finden, von dem aus Drehwinkel bestimmt wird
185 // (beim Drehen auf die x-Achse)
186 int findAngleCalcPoint(Triangle &t, int bestPoint){
187     //just return the point that is counterclockwise from line between other
188     //↪ points
189     switch(bestPoint) {
190         case 0: if(ccw(t.points[0],t.points[1],t.points[2]) > 0){
191             //point 2 is clockwise
192             return 1;
193         } else {
194             //point 2 is counterclockwise
195             return 2;
196         }
197         case 1: if(ccw(t.points[1],t.points[0],t.points[2]) > 0){
198             //point 2 is clockwise
199             return 0;
200         } else {
201             //point 2 is counterclockwise
202             return 2;
203         }
204         case 2: if(ccw(t.points[2],t.points[1],t.points[0]) > 0){
205             //point 0 is clockwise
206             return 1;
207         } else {
208             //point 0 is counterclockwise
209             return 0;
210         }
211         break;
212     default: return 0;
213         break;
214     }
215 }
216 }

```

Quellcode 1: Die Datei `triangles.cpp`, die die Klassen `Triangle`, `Vektor` und `Point` und nützliche Hilfsfunktionen für den eigentlichen Algorithmus enthält

```

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

```

```
3  #define EPSILON 10
4
5  using namespace std;
6
7  vector<int> bestPointIndex; // Punkt mit dem kleinsten Winkel (0,1 oder 2)
8  vector<int> bestAngle; // kleinster Winkel als int (für Subset Sum)
9  vector<double> bestAngleDouble; // kleinster Winkel als double
10 vector<int> sol; // Ausgabe des Subset Sum Algorithmus
11 vector<Triangle> triangles; // noch nicht platzierte Dreiecke
12 vector<Triangle> placedTriangles; // platzierte Dreiecke
13
14 // Backtracen der Subset Sum Lösung
15 // gibt nur ein mögliches Subset zurück
16 bool getSubsetsRec(vector<int> arr, int i, int sum, vector<int>& p,
17     ↪ vector<vector<bool>> &dp) {
18     // erstes Element erreicht, aber Summe nicht 0
19     // -> dp[0][sum] muss true sein, damit Subset möglich ist
20     if (i == 0 && sum != 0 && dp[0][sum]) {
21         p.push_back(i);
22         sol = p;
23         return true;
24     }
25
26     // erstes Element erreicht und Summe 0 -> möglich
27     if (i == 0 && sum == 0) {
28         sol = p;
29         return true;
30     }
31
32     // wenn Summe ohne aktuelles Element erreicht werden kann
33     if (dp[i-1][sum]) {
34         vector<int> b = p;
35         if(getSubsetsRec(arr, i-1, sum, b,dp)){
36             return true;
37         }
38     }
39
40     // wenn Summe mit aktuellem Element erreicht werden kann
41     if (sum >= arr[i] && dp[i-1][sum-arr[i]]) {
42         p.push_back(i);
43         if(getSubsetsRec(arr, i-1, sum-arr[i], p,dp)){
44             return true;
45         }
46     }
47     return false;
48 }
49
50 void subsetSum(vector<int> set,int sum){
51     int n = set.size();
52     vector<vector<bool>> dp(n,vector<bool>(sum+1));
53
54     // Summe 0 immer erreichbar
55     for (int i=0; i<n; ++i) {
```



```

55     dp[i][0] = true;
56 }
57
58 // Summe set[0] kann mit erstem Element erreicht werden
59 if (set[0] <= sum)
60     dp[0][set[0]] = true;
61
62 // DP-Array nach Rekursionsgleichung füllen
63 for (int i = 1; i < n; ++i) {
64     for (int j = 0; j < sum + 1; ++j) {
65         dp[i][j] = (set[i] <= j) ? (dp[i-1][j] || dp[i-1][j-set[i]]) :
66             ↪ dp[i - 1][j];
67     }
68 }
69
70 // größte Summe, die erreicht werden kann
71 int best = sum;
72 for(;best>=0;best--){
73     if(dp[n-1][best]) break;
74 }
75
76 // Backtracen
77 vector<int> p;
78 getSubsetsRec(set, n-1, best, p,dp);
79 }
80
81 // Sortieren einer Teilmenge von Dreiecken
82 // nach der längsten Seite, die am auf der x-Achse liegenden
83 // Punkt des Dreiecks (mit dem kleinsten Winkel) anliegt.
84 // "Kleine" Dreiecke sind dadurch weiter rechts.
85 bool lengthSortFunc(const int t1, const int t2){
86     return triangles[t1].longestLength(bestPointIndex[t1]) <
87         ↪ triangles[t2].longestLength(bestPointIndex[t2]);
88 }
89
90 // Sortieren der Dreiecke anhand ihres Mittelpunkts (x-Koordinate)
91 // zur geordneten Ausgabe
92 bool triangleSortFunc(const Triangle &t1, const Triangle &t2){
93     double mid1 = (t1.points[0].x + t1.points[1].x + t1.points[2].x) / 3;
94     double mid2 = (t2.points[0].x + t2.points[1].x + t2.points[2].x) / 3;
95     return mid1 < mid2;
96 }
97
98 // aktuelles Subset aus den noch nicht platzierten Dreiecken löschen
99 // und zu den platzierten hinzufügen
100 void deleteUsedTriangles(){
101     // in absteigender Reihenfolge der Indexe -> sonst: Segfault
102     sort(sol.begin(),sol.end(),greater<int>());
103     for(auto index: sol){
104         placedTriangles.push_back(triangles[index]);
105         triangles.erase(triangles.begin()+index);
106         bestAngle.erase(bestAngle.begin()+index);
107         bestAngleDouble.erase(bestAngleDouble.begin()+index);

```

```

106         bestPointIndex.erase(bestPointIndex.begin()+index);
107     }
108 }
109
110 // Dreiecke eines Subsets so verschieben, dass sie
111 // mit dem "besten" Punkt auf der x-Achse liegen
112 // und so rotieren, dass sie direkt auf der Achse liegen
113 void translateAndRotateToAxis(Point centerPoint){
114     for(auto index : sol){
115         auto &t = triangles[index];
116         Vektor translation =
117             ↪ Vektor(t.points[bestPointIndex[index]],centerPoint);
118         for(int i=0;i<=2;i++){
119             t.points[i] = addVektor(t.points[i],translation);
120         }
121         int rotatePoint = findAngleCalcPoint(t,bestPointIndex[index]);
122         double rotateAngle = atan360(centerPoint,t.points[rotatePoint]);
123         for(int i=0;i<=2;i++){
124             rotate_tri(centerPoint, t.points[i], rotateAngle);
125         }
126     }
127 }
128
129 // für Anfügen auf der rechten Seite
130 // Dreiecke so rotieren, dass sie sich nicht überlappen
131 // anschließend an die bisherige Anordnung "randrehen"
132 void rotateToPositionRight(Point centerPoint, double free_angle, bool
133     ↪ rotateLeft){
134     double triRotateAngle = 0;
135     for(size_t i=0;i<sol.size();i++){
136         auto &t = triangles[sol[i]];
137         for(int j=0;j<=2;j++){
138             rotate_tri(centerPoint, t.points[j], triRotateAngle);
139         }
140         triRotateAngle += bestAngleDouble[sol[i]];
141     }
142     // "Randrehen"
143     if(rotateLeft){
144         for(size_t i=0;i<sol.size();i++){
145             auto &t = triangles[sol[i]];
146             for(int j=0;j<=2;j++){
147                 rotate_tri(centerPoint, t.points[j],
148                     ↪ free_angle-triRotateAngle);
149             }
150         }
151     }
152 }
153
154 // für Anfügen auf der linken Seite
155 // wird bedingt durch Sortierung der Dreiecke normalerweise nie benutzt
156 void rotateToPositionLeft(Point centerPoint, double free_angle){
157     double triRotateAngle = 0;
158     for(size_t i=0;i<sol.size();i++){

```

```

156     auto &t = triangles[sol[i]];
157     for(int j=0;j<=2;j++){
158         rotate_tri(centerPoint, t.points[j], M_PI - triRotateAngle -
            ↪ bestAngleDouble[sol[i]]);
159     }
160     triRotateAngle += bestAngleDouble[sol[i]];
161 }
162 // "Randrehen"
163 for(size_t i=0;i<sol.size();i++){
164     auto &t = triangles[sol[i]];
165     for(int j=0;j<=2;j++){
166         rotate_tri(centerPoint, t.points[j],
            ↪ -(free_angle-triRotateAngle));
167     }
168 }
169 }
170
171 // linkensten und rechtensten Punkt finden,
172 // die zur Berechnung des Gesamtabstands herangezogen werden
173 pair<double,double> calculateDistance(){
174     double leftmost=300,rightmost=300;
175     for(auto tri: placedTriangles){
176         double left_triangle = 100000, right_triangle = 0;
177         for(auto p: tri.points){
178             if(p.y == 0){
179                 if(p.x < left_triangle) left_triangle = p.x;
180                 if(p.x > right_triangle) right_triangle = p.x;
181             }
182         }
183         if(left_triangle < 300 && right_triangle < leftmost) leftmost =
            ↪ right_triangle;
184         if(right_triangle > 300 && left_triangle > rightmost) rightmost =
            ↪ left_triangle;
185     }
186     return {leftmost, rightmost};
187 }
188
189 // Anordnung so verschieben, dass der "linkeste" Punkt
190 // auf der y-Achse liegt
191 void moveToRightOfY(){
192     double minx = 100000;
193     for(auto t: placedTriangles){
194         for(auto pnt: t.points){
195             if(pnt.x < minx) minx = pnt.x;
196         }
197     }
198     for(auto &t: placedTriangles){
199         for(auto &pnt: t.points){
200             pnt.x -= minx;
201         }
202     }
203 }
204

```

```
205 // Hauptmethode des Algorithmus
206 pair<vector<Triangle>,double> doAlgorithm(vector<Triangle> i_triangles){
207     triangles = i_triangles;
208     Point centerPoint = Point(300,0); // erster Anlegepunkt (willkürlich
        ↪ gewählt)
209
210     // kleinsten Winkel und entsprechenden Punkt der Dreiecke berechnen
211     for(size_t i = 0; i<triangles.size(); i++){
212         auto &t = triangles[i];
213         int ind;
214         double angle;
215         tie(ind,angle) = locateSmallestAngle(t);
216         bestPointIndex.push_back(ind);
217         bestAngle.push_back((int) ceil(10000*angle));
218         bestAngleDouble.push_back(angle);
219     }
220
221     // Subset-Sum ausführen und nach Dreiecke nach Seitenlänge ordnen
222     subsetSum(bestAngle,(int) floor(10000*M_PI));
223     sort(sol.begin(),sol.end(),lengthSortFunc);
224
225     // Dreiecke passend verschieben und rotieren
226     translateAndRotateToAxis(centerPoint);
227     rotateToPositionRight(centerPoint,M_PI,false);
228     deleteUsedTriangles();
229
230     // solange nicht alle Dreiecke platziert
231     double leftmost, rightmost;
232     while(!triangles.empty()){
233         // Punkte für die Berechnung der Gesamtdistanz
234         tie(leftmost,rightmost) = calculateDistance();
235
236         // min./max. Punkt finden, an dem angelegt werden kann
237         // ist ein Punkt eines platzierten Dreiecks
238         double minx_axis = 100000, maxx_axis = 0;
239         for(auto t: placedTriangles){
240             for(auto pnt: t.points){
241                 if(pnt.y <= EPSILON){
242                     if(pnt.x < minx_axis) {minx_axis = (pnt.y > 0) ? pnt.x -
                        ↪ pnt.y : pnt.x;}
243                     if(pnt.x > maxx_axis) {maxx_axis = (pnt.y > 0) ? pnt.x +
                        ↪ pnt.y : pnt.x;}
244                 }
245             }
246         }
247
248         Point newCenterRight = Point(maxx_axis,0);
249         Point newCenterLeft = Point(minx_axis,0);
250
251         // freien Winkel (links/rechts) finden
252         // wird durch den Punkt bestimmt, der den geringsten Winkel/atan vom
        ↪ Anlegepunkt hat
253         Point minpnt_above, maxpnt_above;
```

```

254     double angle_left = M_PI, angle_right = M_PI;
255     for(auto t: placedTriangles){
256         for(auto pnt: t.points){
257             if(pnt.y == 0) continue;
258             if(atan_angle(newCenterRight,pnt) < angle_right) { angle_right
↳ = atan_angle(newCenterRight,pnt); maxpnt_above = pnt;}
259             if(atan180(newCenterLeft,pnt) < angle_left) {angle_left =
↳ atan_angle(newCenterLeft,pnt); minpnt_above = pnt;}
260         }
261     }
262
263     // links oder rechts platzieren - was ist besser?
264     // normalerweise rechts
265     bool left = false;
266     double free_angle = 0;
267     if(leftmost - newCenterLeft.x < newCenterRight.x - rightmost){
268         //place left
269         free_angle = atan180(newCenterLeft,minpnt_above);
270         left = true;
271     } else {
272         //place right
273         free_angle = atan_angle(newCenterRight,maxpnt_above);
274     }
275
276     // Subset Sum + Sortieren (wie oben)
277     subsetSum(bestAngle,(int) floor(10000*free_angle));
278     sort(sol.begin(),sol.end(),lengthSortFunc);
279
280     // Links oder rechts platzieren
281     // Links zuerst Reihenfolge umdrehen
282     if(left){
283         reverse(sol.begin(),sol.end());
284         translateAndRotateToAxis(newCenterLeft);
285         rotateToPositionLeft(newCenterLeft,free_angle);
286     } else {
287         translateAndRotateToAxis(newCenterRight);
288         rotateToPositionRight(newCenterRight,free_angle,true);
289     }
290     deleteUsedTriangles();
291 }
292
293 // Distanz und zur Anzeige verschieben/sortieren
294 auto dpair = calculateDistance();
295 double dist = abs(dpair.second - dpair.first);
296 moveToRightOfY();
297 sort(placedTriangles.begin(),placedTriangles.end(),triangleSortFunc);
298
299 return {placedTriangles,dist};
300 }

```

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