

# Dreiecksbeziehungen

Aufgabe 2, Runde 2, 37. Bundeswettbewerb Informatik

Erik Klein

Teilnahme-ID: 52207

## Inhaltsverzeichnis

<b>1 Lösungsidee</b>	<b>1</b>
1.1 Grundlegende Überlegungen . . . . .	1
1.1.1 Aneinanderliegen der Kantendreiecke . . . . .	1
1.1.2 Linksgeneigte Platzierungen . . . . .	2
1.2 Simulated Annealing . . . . .	3
1.2.1 Veränderung einer LGP . . . . .	4
1.3 Laufzeitanalyse . . . . .	9
<b>2 Umsetzung und Quellcode</b>	<b>10</b>
<b>3 Beispiele</b>	<b>13</b>

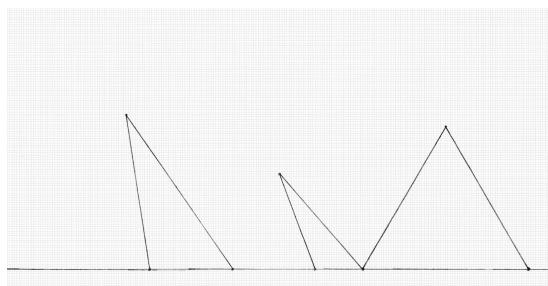
# 1 Lösungsidee

## 1.1 Grundlegende Überlegungen

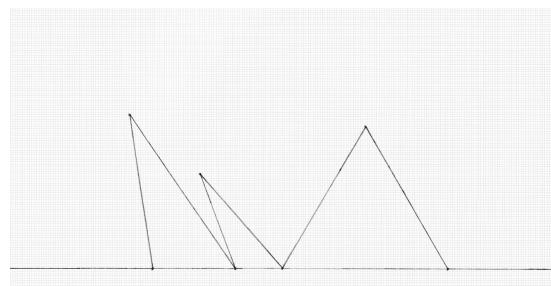
### 1.1.1 Aneinanderliegen der Kantendreiecke

Jedes Dreieck ist entweder ein *Eckendreieck* und berührt die x-Achse mit einer Ecke, der *Berührecke*, oder ein *Kantendreieck* und berührt sie mit einer Kante, der *Berührkante*.

Zunächst habe ich bedacht, dass wenn zwei nebeneinander liegende Kantendreiecke nicht aneinander liegen, das rechte und die rechts davon liegenden Dreiecke so weit nach links verschoben werden können, bis sie aneinander liegen.

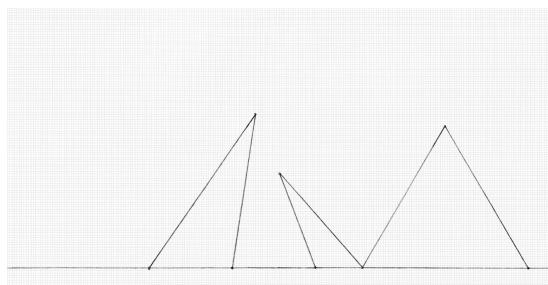


vorher

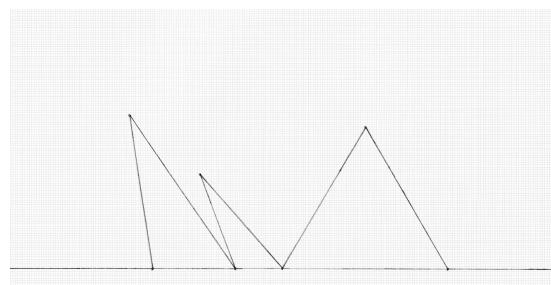


nachher

Möglicherweise muss dabei ein Dreieck „umgedreht“ werden(eine Art Spiegelung), da sich die Dreiecke nicht überschneiden können.

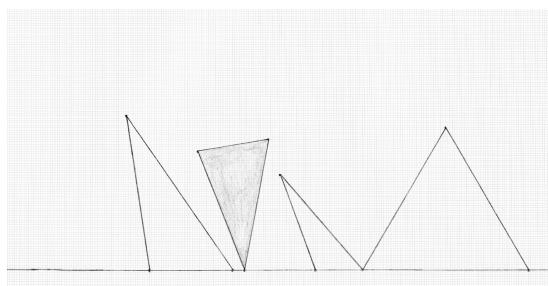


vorher

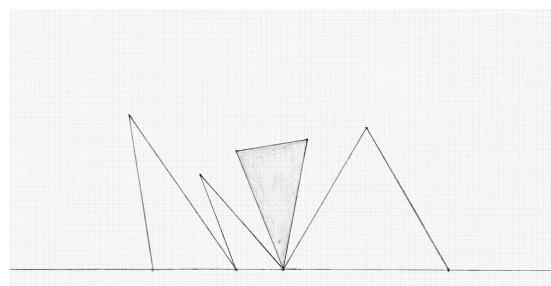


nachher

Möglicherweise liegen zwischen den beiden Kantendreiecken Eckendreiecke, die dann anders platziert werden müssen.

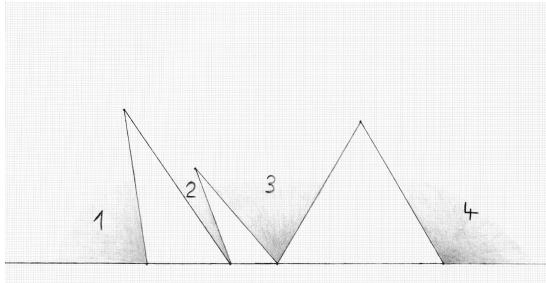


vorher

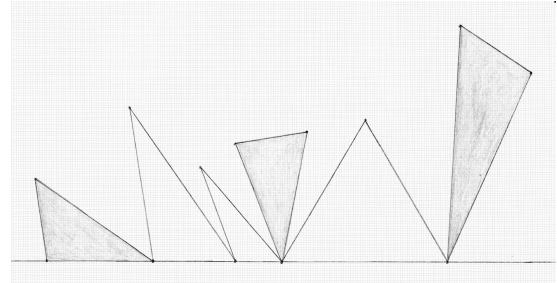


nachher

Fest steht, dass es immer möglich ist und den Gesamtabstand verringert. Damit der Gesamtabstand minimal ist, müssen also alle Kantendreiecke aneinander liegen. Die Eckendreiecke können somit nur in den Bereichen zwischen zwei Kantendreiecken und links des linksten und rechts des rechtesten, den *Nischen*, liegen.



Nischen(grau, nummeriert)



Eckendreiecke(grau) in Nischen

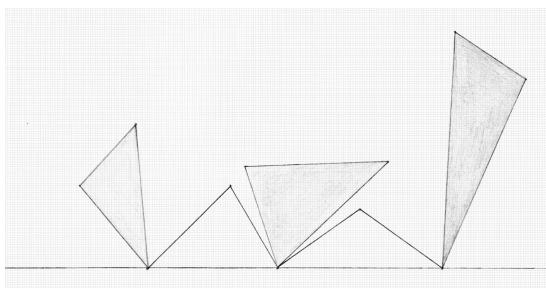
Die Berührecken müssen Endpunkte der Berührkanten berühren.

Eine *Platzierung* ist bestimmt durch eine Reihenfolge der Dreiecke von links nach rechts, eine Aufteilung in Ecken- und Kantendreiecke, die jeweiligen Berührecken/kanten, welche Dreiecke umgedreht sind und welche nicht und welche Neigungen die Eckendreiecke haben. Der Gesamtabstand ist die Summe der Längen der Berührkanten. Eine *optimale Platzierung*, eine Platzierung mit minimalem Gesamtabstand, soll gefunden werden.

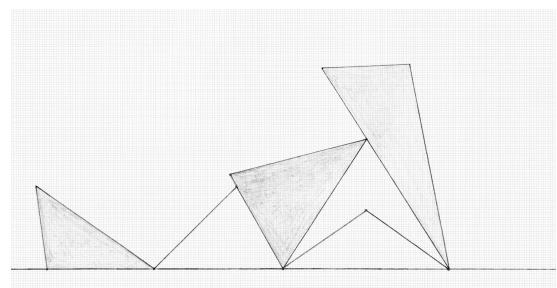
Zunächst stehen wir vor der Schwierigkeit, dass es für jedes Eckendreieck unendlich viele mögliche Neigungen und somit unendlich viele Platzierungen gibt. Um überhaupt irgendwie mit den Platzierungen arbeiten zu können, müssen wir sie auf eine endliche Anzahl begrenzen. Dafür führe ich ein neues Konzept ein:

### 1.1.2 Linksgeneigte Platzierungen

Neigt man die Eckendreiecke einer Platzierung der Reihenfolge nach so weit wie möglich nach links, erhält man die entsprechende *linksgeneigte Platzierung*, kurz LGP.



Platzierung



entsprechende LGP

Allen Platzierungen, die sich nur in den Neigungen unterscheiden, entspricht die gleiche LGP. Die Anzahl der LGPs ist somit endlich. Mehr noch: Eine LGP hat den gleichen Gesamtabstand wie alle Platzierungen, denen sie entspricht. Finden wir eine LGP mit minimalem Gesamtabstand, ist dies deshalb eine optimale Platzierung!

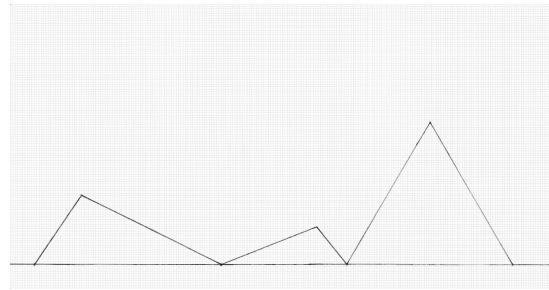
## 1.2 Simulated Annealing

Zwar ist die Anzahl der LGPs endlich, jedoch astronomisch hoch, da allein die Anzahl möglicher Reihenfolgen superexpontiell und die Anzahl möglicher Aufteilungen exponentiell mit der Anzahl  $n$  der Dreiecke ansteigen.

Deshalb ist das Problem höchstwahrscheinlich nicht in der Komplexitätsklasse  $P^1$  und kein Algorithmus mit polynomieller Laufzeit existiert, der garantiert immer exakt die optimale Platzierung findet.

Wir sollten es besser heuristisch angehen, um dennoch in annehmbarer Zeit gute Platzierungen zu finden. Dafür habe ich Simulated Annealing gewählt, da es sich bestens eignet:

- Mit dem Gesamtabstand gibt es eine Kostenfunktion, die die Güte einer LGP angibt.
- Es ist einfach, eine anfängliche LGP zu erstellen: Die Dreiecke sind alle mit ihrer längsten Kante als Berührkante Kantendreiecke und haben eine zufällige Reihenfolge, z.B. so:



Da durch ersteres die Innenwinkel an den Berührkanten nicht größer als  $90^\circ$  sind, überschneiden sich die Dreiecke nicht.

- Man kann eine LGP zu einer anderen verändern, mehr dazu im nächsten Abschnitt.

Simulated Annealing arbeitet wie folgt: Nach dem Vorbild des Verfahrens, der Abkühlung von Metall, gibt es eine Temperatur  $t$ .

Zuerst wird  $t$  initialisiert und die anfängliche LGP erstellt. Dann werden mehrere Durchläufe durchgeführt, in jedem welcher alle möglichen Veränderungen aufgelistet und eine zufällige von ihnen durchgeführt wird. Wenn sie den Gesamtabstand verringert, wird sie beibehalten, sonst nur mit einer Wahrscheinlichkeit  $p$ , die mit  $t$  abnimmt. Ansonsten wird sie zurückgenommen.

Alle paar Durchläufe wird  $t$  verringert, sodass in früheren Durchläufen Veränderungen, die den Gesamtabstand kurzzeitig erhöhen, wahrscheinlich beibehalten werden (was hilft, im Vergleich zu Alternativen wie local search nicht bei einer lokal optimalen LGP stecken zu bleiben) und in späteren Durchläufen nur noch Veränderungen, die den Gesamtabstand verringern.

Es hat sich bewährt[ADM],  $t$  mit 1 zu initialisieren und (wie bei der Abkühlung von Metall) exponentiell zu verringern, d.h. jeweils mit einem Wachstumsfaktor  $0.8 < f < 1$  zu multiplizieren. Auch sollte  $p$  mit  $e^{\frac{\Delta_{i+1} - \Delta_i}{k \cdot t}}$  berechnet werden, wobei  $k$  die Boltzmann-Konstante,  $\Delta_i$  der Gesamtabstand und  $\Delta_{i+1}$  der Gesamtabstand nach der Veränderung ist. Zusammengefasst:

<sup>1</sup>und nicht in NP, da um zu verifizieren, dass der Gesamtabstand minimal ist, wahrscheinlich die Gesamtabstände aller anderen möglichen LGPs (zumindest teilweise) berechnet werden müssen.

- 1 Erstelle anfängliche LGP
- 2  $t = 1$
- 3 Wiederhole mehrmals
- 4     Wiederhole mehrmals
- 5         Liste alle möglichen Veränderungen auf
- 6         Führe zufällige von ihnen durch
- 7         Wenn sie den Gesamtabstand erhöht
- 8             Nehme sie mit Wahrscheinlichkeit  $1 - e^{\frac{\Delta_i - \Delta_{i+1}}{k \cdot t}}$  zurück
- 9          $t = t \cdot f$

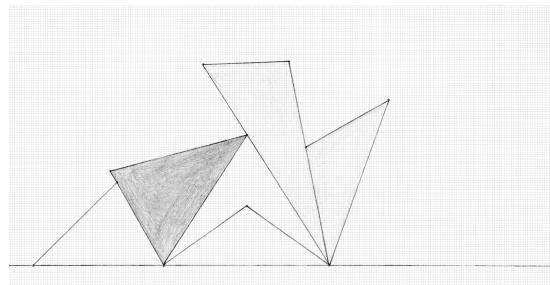
### 1.2.1 Veränderung einer LGP

Wie kann man eine Veränderung einer LGP zu einer anderen durchführen? In drei Schritten:

1. Entfernen eines Dreiecks
2. Wiedereinfügen
3. Aktualisieren der Neigungen der Eckendreiecke

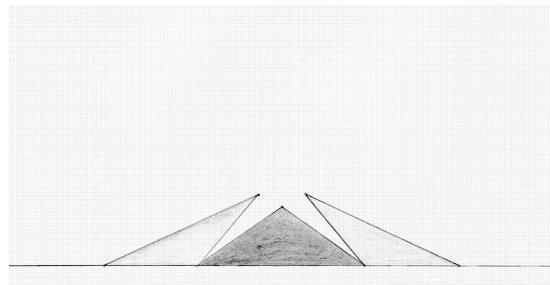
Die Veränderung bestimmt dabei, welches Dreieck entfernt wird und an welcher Position, ob umgedreht oder nicht, ob als Ecken- oder Kantendreieck und mit welcher Ecke/Kante als Berührcke/kante es wieder eingefügt wird.<sup>2</sup> Schritte 1 und 2 sind klar, wofür benötige ich aber Schritt 3?

- Damit die Platzierung linksgeneigt bleibt. Wenn z.B. in dieser LGP



das dunkelgraue Dreieck entfernt wird, sind die beiden hellgrauen Dreiecke nicht mehr so weit wie möglich nach links geneigt.

- Um zu prüfen, ob die Veränderung möglich ist, d.h. zu einer anderen LGP führt. Wenn z.B. in dieser LGP

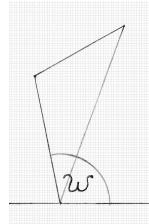



---

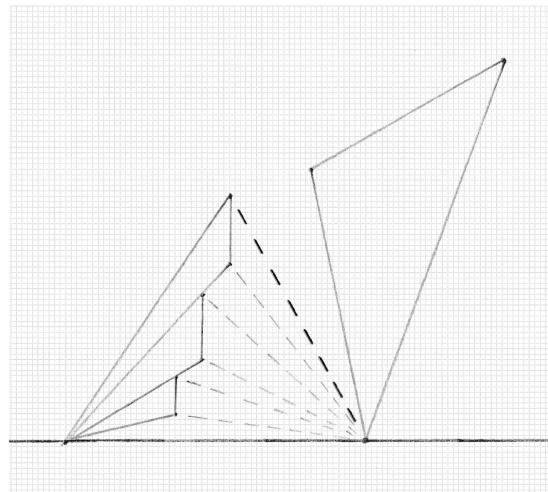
<sup>2</sup>Mit welcher Neigung ist unwichtig, da diese danach aktualisiert wird.

das dunkelgraue Dreieck entfernt wird, passen die hellgrauen Dreiecke nicht zusammen in die dabei entstehende Nische.

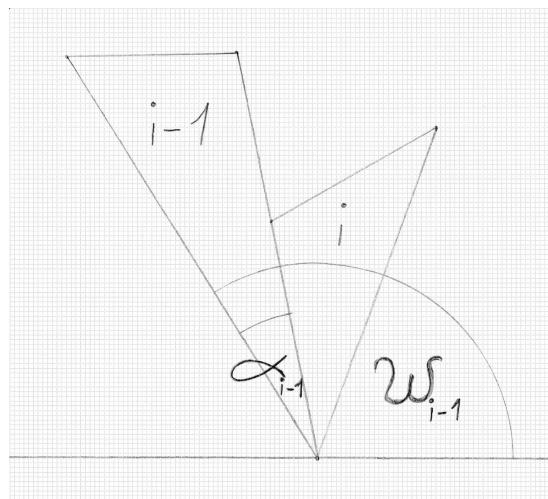
Zuerst eine Definition: Die Neigung  $\omega$  eines Kantendreiecks ist immer  $180^\circ$ , die eines Eckendreiecks ist der Winkel im Uhrzeigersinn von der linken an die Berührecke anliegenden Kante zur x-Achse.



Wie läuft Schritt 3 ab? Die Eckendreiecke werden der Reihenfolge nach so weit wie möglich nach links geneigt. So weit wie möglich ist, bis sie ein in der Reihenfolge voriges Dreieck berühren.  $\omega_i$  wird also auf die erste/geringste Neigung aktualisiert, bei der das  $i$ -te Dreieck ein  $j$ -tes berührt, wobei  $j < i$ .



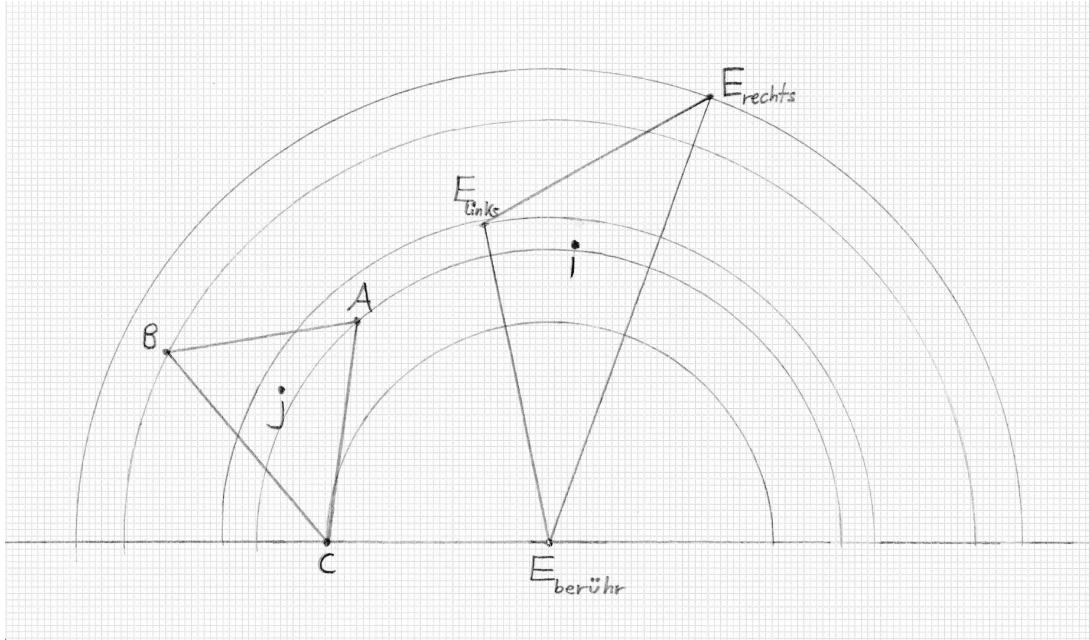
Eine von ihnen ist immer  $\omega_{i-1} - \alpha_{i-1}$  (wobei  $\alpha_{i-1}$  der Innenwinkel des  $(i-1)$ -ten Dreiecks an der Berührecke ist), also wenn das  $i$ -te die Kante des  $(i-1)$ -ten berührt.<sup>3</sup>




---

<sup>3</sup>Ich definiere  $\omega_0 - \alpha_0 = 180^\circ$ .

Um die anderen zu finden, wird jedes  $j$ -te Dreieck durchgegangen, wobei Bezeichnungen sind:

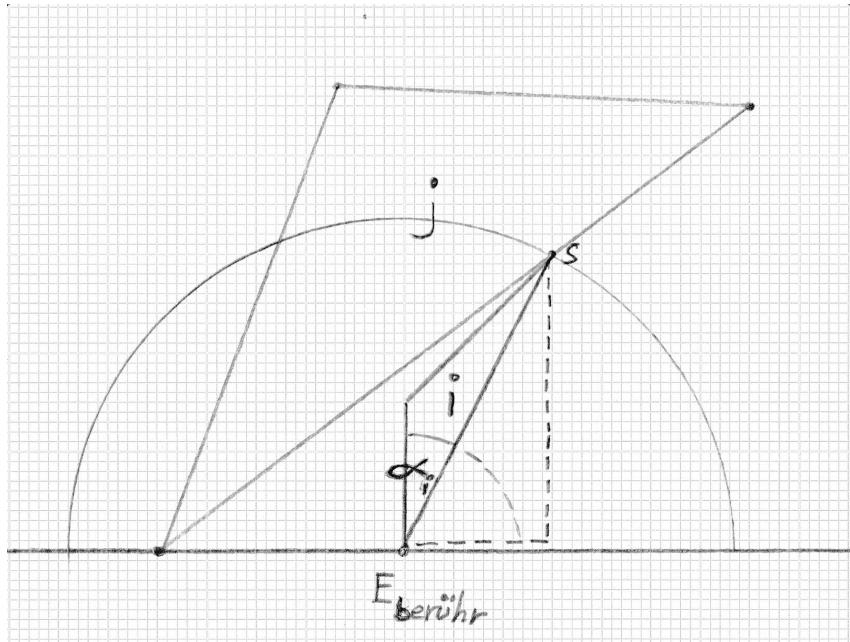


Die Kreise  $k_{\text{links}}$ ,  $k_{\text{rechts}}$ ,  $k_A$ ,  $k_B$  und  $k_C$  durch die entsprechenden Ecken und mit Mittelpunkt  $E_{\text{berühr}}$  sind eingezeichnet. Die entscheidende Beobachtung ist:

- an allen Schnittpunkten  $S$  von  $k_{\text{links}}$  oder  $k_{\text{rechts}}$  und einer Kante des  $j$ -ten Dreiecks kann das  $i$ -te das  $j$ -te mit einer Ecke berühren. Die Neigung, die es dafür haben muss, ist der Winkel von  $\overline{E_{\text{berühr}}S}$  zur x-Achse,

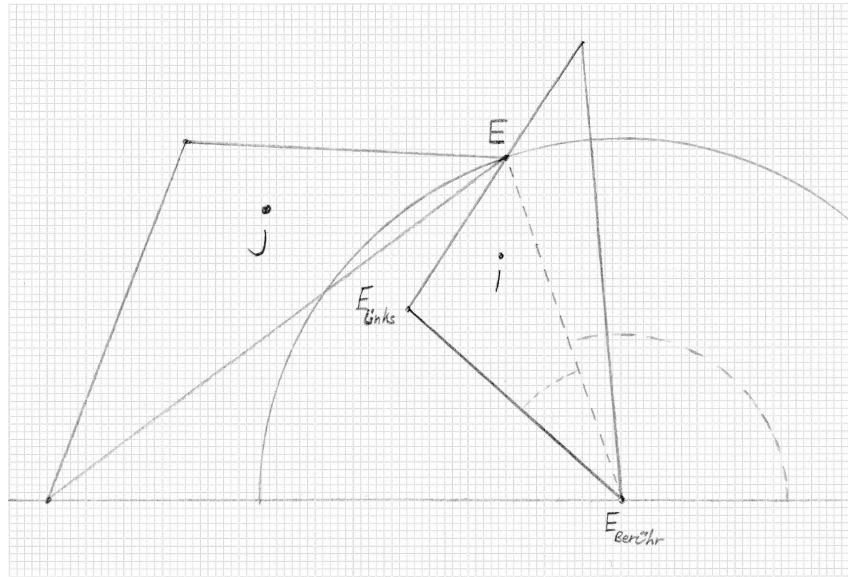
$$\begin{cases} \arctan \frac{y_S - y_{\text{berühr}}}{x_S - x_{\text{berühr}}}, & \text{wenn } x_S \geq x_{\text{berühr}} \\ 180^\circ - \arctan \frac{y_S - y_{\text{berühr}}}{x_S - x_{\text{berühr}}}, & \text{ansonsten} \end{cases} \quad (1)$$

plus den Innenwinkel an  $E_{\text{berühr}}$ ,  $\alpha_i$ , wenn  $S$  auf  $k_{\text{rechts}}$  liegt.



- mit allen Schnittpunkten  $S$  von  $k_A$ ,  $k_B$  oder  $k_C$  und einer Kante des  $i$ -ten Dreiecks kann dieses eine der Ecken des  $j$ -ten,  $A$ ,  $B$  oder  $C$ , berühren. Diese Ecke sei  $E$ . Die Neigung, die es dafür haben muss, ist der Winkel von  $\overline{E_{\text{berühr}} E}$  zur x-Achse (er wird genauso berechnet, einfach in (1)  $S$  durch  $E$  ersetzen) plus der Winkel von  $\overline{E_{\text{berühr}} S}$  zu  $\overline{E_{\text{berühr}} E_{\text{links}}}$ ,

$$\begin{cases} 180^\circ - (\arctan \frac{y_{\text{links}} - y_{\text{berühr}}}{x_{\text{links}} - x_{\text{berühr}}} - \arctan \frac{y_S - y_{\text{berühr}}}{x_S - x_{\text{berühr}}}), & \text{wenn } x_S \geq x_{\text{berühr}} \geq x_{\text{links}} \\ \arctan \frac{y_{\text{links}} - y_{\text{berühr}}}{x_{\text{links}} - x_{\text{berühr}}} - \arctan \frac{y_S - y_{\text{berühr}}}{x_S - x_{\text{berühr}}}, & \text{ansonsten} \end{cases}$$



Wie werden die Schnittpunkte gefunden? Ich demonstriere es an denen, die auf  $k_{\text{links}}$  und der Kante  $\overline{AB}$  des  $j$ -ten Dreiecks liegen: Da  $S$  auf  $\overline{AB}$  liegt, gilt

$$S = (x_A + \lambda(x_B - x_A) | y_A + \lambda(y_B - y_A))$$

für ein  $0 \leq \lambda \leq 1$ . Da  $S$  auf  $k_{\text{links}}$  liegt, gilt

$$\sqrt{(x_s - x_{\text{berühr}})^2 + (y_s - y_{\text{berühr}})^2} = \sqrt{(x_{\text{links}} - x_{\text{berühr}})^2 + (y_{\text{links}} - y_{\text{berühr}})^2}$$

Mit WolframAlpha habe ich die Gleichung nach  $\lambda$  aufgelöst:

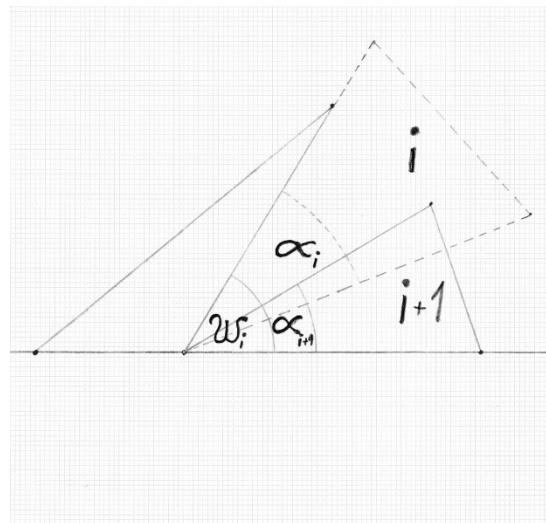
$$\begin{aligned} \lambda = & (\pm((x_A^2 - x_B x_A - x_{\text{berühr}} x_A + x_B x_{\text{berühr}} + y_A^2 - y_A y_B) \\ & - (-x_A^2 + 2x_B x_A - x_B^2 - y_A^2 - y_B^2 + 2y_A y_B)(-x_A^2 + 2x_{\text{berühr}} x_A + x_{\text{links}}^2 - 2x_{\text{berühr}} x_{\text{links}} - y_A^2 + y_{\text{links}}^2))^{1/2} \\ & + x_A^2 - x_B x_A - x_{\text{berühr}} x_A + x_B x_{\text{berühr}} + y_A^2 - y_A y_B) / (x_A^2 - 2x_B x_A + x_B^2 + y_A^2 + y_B^2 - 2y_A y_B) \end{aligned}$$

Damit  $\lambda$  existiert, muss zum einen der Term innerhalb  $(\cdot)^{1/2}$  nichtnegativ und zum anderen der Term hinter dem  $/$  nicht 0 sein. Wenn ja wird geprüft, ob  $0 \leq \lambda \leq 1$  und wenn ja,  $S$  ausgerechnet. Das funktioniert für alle anderen Kreise und Kanten genauso.

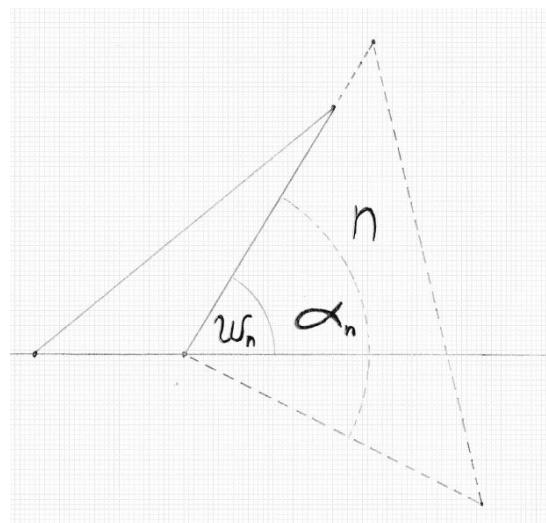
### Zusammenfassung des Ablaufs von Schritt 3

- 1 Für alle  $1 \leq i \leq n$
- 2 Wenn  $i$ -tes Dreieck Eckendreieck
- 3 Finde Neigung  $\omega_{i-1} - \alpha_{i-1}$
- 4 Für alle  $1 \leq j < i$ 
  - 5 Finde alle Neigungen, bei denen  $i$ -tes Dreieck  $j$ -tes berührt
  - 6 aktualisiere  $\omega_i$  auf geringste gefundene Neigung

Schließlich wird geprüft, ob die Veränderung möglich ist, indem jedes  $i$ -te Dreieck, auf das in der Reihenfolge ein  $(i+1)$ -tes Kantendreieck folgt, durchgegangen und geprüft wird, ob  $\omega_i - \alpha_{i+1} < \alpha_i$ . Wenn nämlich  $\omega_i - \alpha_{i+1} < \alpha_i$ , müsste ein Teil des  $i$ -ten Dreiecks mit dem  $(i+1)$ -ten überlappen, was nicht möglich ist.



Aus dem gleichen Grund wird, wenn das  $n$ -te Dreieck ein Eckendreieck ist, geprüft, ob  $\omega_n < \alpha_n$ , da dann ein Teil unter der x-Achse liegen müsste.



*Anmerkung.* Für Zeile 5 von Simulated Annealing, „Liste alle möglichen Veränderungen auf“, muss geprüft werden, ob Veränderungen möglich sind, ohne sie auf der echten Platzierung durchzuführen. Dafür werden sie auf Kopien durchgeführt.

### 1.3 Laufzeitanalyse

Wie viel Zeit benötigt Schritt 3?

- Zeile 3 benötigt Zeit  $\mathcal{O}(1)$ . Zeile 6 auch, da die geringste gefundene Neigung gespeichert und immer wenn eine weitere Neigung gefunden wird aktualisiert werden kann.
- Zeile 5 benötigt Zeit  $\mathcal{O}(1)$ , da für eine konstante Anzahl an Kreisen(5) für eine konstante Anzahl an Kanten(3) jeweils eine maximal konstante Anzahl an Rechnungen durchgeführt wird.

Am häufigsten davon wird Zeile 5 ausgeführt, nämlich im schlimmsten Fall, wenn alle Dreiecke Eckendreiecke sind,  $\sum_{i=1}^n i - 1 = \frac{n(n-1)}{2}$  Mal. Deshalb wird für Schritt 3 insgesamt Zeit  $\mathcal{O}(n^2)$  benötigt.

Wie viel Zeit benötigt es, eine Veränderung durchzuführen? Wie viel Zeit benötigt es, zu prüfen, ob eine Veränderung möglich ist? Wie viel Zeit benötigen alle drei Schritte zusammen?

Ich will die Dreiecke in einem Array speichern. Dann wird zum Entfernen und zum Einfügen eines Dreiecks Zeit  $\mathcal{O}(n)$  benötigt, da im schlimmsten Fall  $n - 1$  Dreiecke um einen Index verschoben werden müssen. Diese Zeit wird von der Zeit für Schritt 3 dominiert, sodass die Schritte zusammen Zeit  $\mathcal{O}(n^2)$  benötigen.

Wie viel Zeit benötigt Zeile 5 von Simulated Annealing, „Liste alle möglichen Veränderungen auf“?

Jedes der  $n$  Dreiecke kann entfernt und an  $n$  Positionen wiedereingefügt werden, entweder umgedreht oder nicht, entweder als Ecken- oder als Kantendreieck und mit einer der 3 Ecken/Kanten. Für  $n \cdot n \cdot 2 \cdot 3 = 12n^2$  Veränderungen muss also geprüft werden, ob sie möglich sind. Deshalb benötigt Zeile 5 von Simulated Annealing Zeit  $\mathcal{O}(n^4)$ .

Wie viel Zeit benötigt ein Durchlauf von Simulated Annealing?

- 5    Liste alle möglichen Veränderungen auf
- 6    Führe zufällige von ihnen durch
- 7    Wenn sie den Gesamtabstand erhöht
- 8       Nehme sie mit Wahrscheinlichkeit  $1 - e^{\frac{\Delta_i - \Delta_{i+1}}{k \cdot t}}$  zurück

Für Zeilen 6-8 muss die LGP kopiert, die zufällige Veränderung auf der Kopie durchgeführt und nur wenn sie nicht zurückgenommen werden soll, in die LGP zurückkopiert werden. Das Kopieren und Zurückkopieren benötigt jeweils Zeit  $\mathcal{O}(n)$ . Diese wird von der Zeit zum Durchführen der Veränderung dominiert, sodass Zeilen 6-8 insgesamt Zeit  $\mathcal{O}(n^2)$  benötigen.

Diese Zeiten werden alle von der Zeit für Zeile 5 dominiert, sodass ein Durchlauf von Simulated Annealing insgesamt Zeit  $\mathcal{O}(n^4)$  benötigt.

Wie viel Zeit benötigt das gesamte Lösungsverfahren?

Das Erstellen der anfänglichen LGP benötigt Zeit  $\mathcal{O}(n)$ , das Initialisieren von  $t$  Zeit  $\mathcal{O}(1)$ . Diese Zeiten werden von der Zeit für die Durchläufe dominiert, sodass insgesamt Zeit  $\mathcal{O}(cn^4)$  benötigt wird.

Ich habe mir überlegt, dass die Anzahl  $c$  der Durchläufe per Hand eingestellt werden kann. Dadurch kann die Laufzeit und die Qualität der Lösung gesteuert und in jedem Anwendungsfall der optimale Trade-off zwischen Laufzeit und Qualität erreicht werden.

Schon wenn man ein kleines  $c$  wählt, wird man eine Platzierung mit geringem Gesamtabstand erhalten, da in früheren Durchläufen die größten und meisten Verbesserungen stattfinden. Hat man viel Zeit zur Verfügung, kann man ein großes  $c$  wählen und wird eine Platzierung mit nahezu optimalem Gesamtabstand erhalten.

*Wie viel Speicher wird benötigt?*

Am speicheraufwändigsten ist es, die in einem Durchlauf möglichen Veränderungen zu speichern. Im schlimmsten Fall können dies nämlich alle  $12n^2$  sein, und pro Veränderung wird Speicher  $\mathcal{O}(1)$  benötigt. Deshalb benötigt das Lösungsverfahren Speicher  $\mathcal{O}(n^2)$ .

## 2 Umsetzung und Quellcode

Das Lösungsverfahren wurde in C++11 implementiert, der Ablauf ist:

```
int main() {
    eingabe();
    in_den_sinn();
    erstelle_LGP();
    simulated_annealing();
    ausgabe();
    return 0;
}
```

Dreiecke und die zugehörigen Angaben werden mit folgendem `struct` gespeichert:

```
typedef long double ld;
struct dreieck {
    int beruehr_n;
    ld x[3], y[3], inwinkel[3], laenge[3], neigung;
    bool eckendreieck, umgedreht;
};
```

`beruehr_n` gibt dabei die Nummer der Berührecke/kante an. Zuerst werden die Dreiecke in den `std::vector LGP` gelesen, wobei die Kantenlängen und nach dem Kosinussatz die Innenwinkel berechnet werden.

```
void eingabe() {
    cin >> N;
    for(int I = 0; I < N; I++) {
        for(int J = 0, temp; J < 3; J++)
            cin >> temp >> LGP[I].x[J] >> LGP[I].y[J];
        for(int J = 0; J < 3; J++)
            LGP[I].laenge[J] = sqrt(pow(LGP[I].x[(J+1)%3]-LGP[I].x[J],2)
                                    +pow(LGP[I].y[(J+1)%3]-LGP[I].y[J],2));
        for(int J = 0; J < 3; J++) // Kosinussatz
            LGP[I].inwinkel[J] = acos((pow(LGP[I].laenge[J],2)
                                         +pow(LGP[I].laenge[(J+2)%3],2)
                                         -pow(LGP[I].laenge[(J+1)%3],2))
                                         /(2 *LGP[I].laenge[(J+2)%3]
                                         *LGP[I].laenge[J]));
    }
}
```

Für die späteren Berechnungen muss sichergestellt werden, dass die Ecken aller Dreiecke im Uhrzeigersinn angegeben sind. Dies tut die Methode `in_den_sinn()`, indem sie von jedem Dreieck, dessen Ecken gegen den Uhrzeigersinn angegeben wurden, zwei Ecken vertauscht. Ob die Ecken gegen den Uhrzeigersinn angegeben wurden, wird dabei mit der gaußschen Trapezformel geprüft:

```
void in_den_sinn() {
    for(int I = 0; I < N; I++) {
        if ((LGP[I].x[1]-LGP[I].x[0])*(LGP[I].y[1]+LGP[I].y[0])
            +(LGP[I].x[2]-LGP[I].x[1])*(LGP[I].y[2]+LGP[I].y[1])
            +(LGP[I].x[0]-LGP[I].x[2])*(LGP[I].y[0]+LGP[I].y[2]) < 0) {
            swap(LGP[I].x[0], LGP[I].x[1]);
            swap(LGP[I].y[0], LGP[I].y[1]);
            swap(LGP[I].inwinkel[0], LGP[I].inwinkel[1]);
            swap(LGP[I].laenge[0], LGP[I].laenge[1]);
        }
    }
}
```

Dann wird wie beschrieben die anfängliche LGP erstellt:

```
void erstelle_LGP() {
    random_shuffle(LGP.begin(), LGP.end()); // zufällige Reihenfolge
    for(int I = 0; I < N; I++) {
        LGP[I].eckendreieck = false; // Kantendreieck
        LGP[I].neigung = M_PI; // 180
        LGP[I].umgedreht = (bool) round(rand_ld()); // zufällig
        // längste Kante Beruehrkante
        if(LGP[I].laenge[0] >= LGP[I].laenge[1]
            && LGP[I].laenge[0] >= LGP[I].laenge[2])
            LGP[I].beruehr_n = 0;
        if(LGP[I].laenge[1] >= LGP[I].laenge[0]
            && LGP[I].laenge[1] >= LGP[I].laenge[2])
            LGP[I].beruehr_n = 1;
        if(LGP[I].laenge[2] >= LGP[I].laenge[0]
            && LGP[I].laenge[2] >= LGP[I].laenge[1])
            LGP[I].beruehr_n = 2;
    }
    ld x_beruehr = 0;
    for(int I = 0; I < N; I++)
        aktualisiere_position(I, x_beruehr, LGP);
}
```

Dabei wird auch eine zentrale Methode, `aktualisiere_position()`, aufgerufen. Diese dient dazu, wenn ein Dreieck umplatziert wurde, anhand der Position des vorigen Dreiecks, den Innenwinkeln und den Seitenlängen, mit ein bisschen Trigonometrie die Koordinaten seiner Ecken neu zu berechnen:

```
void aktualisiere_position(int I, ld &x_beruehr, vector<dreieck> &auf) {
    int xb = auf[I].beruehr_n;
    if (auf[I].eckendreieck) {
        auf[I].x[xb] = x_beruehr;
        auf[I].y[xb] = 0;
        if (auf[I].umgedreht) {
            // ähnlich
        } else {
            auf[I].x[(xb+1)%3] = x_beruehr + cos(auf[I].neigung)
                * auf[I].laenge[xb];
            auf[I].y[(xb+1)%3] =
                sin(auf[I].neigung)
```

```

        * auf[I].laenge[xb];
    auf[I].x[(xb+2)\%3] = x_beruehr + cos(auf[I].neigung
        - auf[I].inwinkel[xb])
        * auf[I].laenge[(xb+2)\%3];
        sin(auf[I].neigung
        - auf[I].inwinkel[xb])
        * auf[I].laenge[(xb+2)\%3];
    }
} else {
    if (auf[I].umgedreht) {
        // aehnlich
    } else {
        auf[I].y[xb] = auf[I].y[(xb+1)\%3] = 0;
        auf[I].x[ xb ] = x_beruehr + auf[I].laenge[xb];
        auf[I].x[(xb+1)\%3] = x_beruehr;
        auf[I].x[(xb+2)\%3] = x_beruehr + cos(auf[I].inwinkel[(xb+1)\%3])
        * auf[I].laenge[(xb+1)\%3];
        sin(auf[I].inwinkel[(xb+1)\%3])
        * auf[I].laenge[(xb+1)\%3];
    }
    x_beruehr += auf[I].laenge[xb];
}
}

```

Darauf folgt das Simulated Annealing:

```

vector<veraenderung> moeglich;
void simulated_annealing() {
    for(int epoche = 0; epoche < epochen; epoche++) {
        for(int I = 0; I < c / epochen; I++) {
            liste_auf();
            veraenderung v = moeglich[rand_int(moeglich.size())]; // zufaellige
            if(gesabstands_aenderung(v) <= 0)
                || (rand_ld() < exp(-gesabstands_aenderung(v)/(k*t)))
            moeglich.clear();
        }
        t = t*f;
    }
}

```

Für die Methode `gesabstands_aenderung()` habe ich dabei bedacht, dass der Gesamtabstand sich durch Entfernen und/oder Einfügen eines Eckendreiecks nicht ändert. Wird ein Kantendreieck entfernt, verringert dies den Gesamtabstand um die Länge der Berührkante, wird eines eingefügt, erhöht es diesen. Deshalb:

```

ld gesabstands_aenderung(veraenderung v) {
    ld kommt_dazu, geht_weg;
    if(v.eckendreieck)
        kommt_dazu = 0;
    else
        kommt_dazu = LGP[v.entfernpos].laenge[v.beruehr_n];
    if(LGP[v.entfernpos].eckendreieck)
        geht_weg = 0;
    else
        geht_weg = LGP[v.entfernpos].laenge[LGP[v.entfernpos].beruehr_n];
    return kommt_dazu - geht_weg;
}

```

Eine Veränderung wird dabei durch folgendes `struct` repräsentiert:

```

struct veraenderung {
    int entfernpos, einfuegpos, beruehr_n;
    bool eckendreieck, umgedreht;
};

```

Die Methode `liste_auf()` prüft für jede Veränderung, ob sie möglich ist, wenn ja fügt sie sie in den `vector moeglich` ein:

```

void liste_auf() {
    for(int entfernpos = 0; entfernpos < N; entfernpos++) {
        for(int einfuegpos = 0; einfuegpos < N; einfuegpos++) {
            for(int beruehr_n = 0; beruehr_n < 3; beruehr_n++) {
                for(int eckendreieck = 0; eckendreieck <= 1; eckendreieck++) {
                    for(int umgedreht = 0; umgedreht <= 1; umgedreht++) {
                        veraenderung v = { entfernpos, einfuegpos,
                                            beruehr_n, eckendreieck, umgedreht };
                        vector<dreieck> kopie = LGP;
                        if(fuehre_durch(v, kopie)) { // zum Pruefen ob moeglich
                            moeglich.push_back(v);
                    }
                }
            }
        }
    }
}

```

Auch das Durchführen einer Veränderung geschieht wie beschrieben:

```

bool fuehre_durch(veraenderung v, vector<dreieck> &auf) {
    // Dreieck speichern
    dreieck d = auf[v.entfernpos];
    // Veraenderungen vornehmen
    d.beruehr_n      = v.beruehr_n;
    d.eckendreieck = v.eckendreieck;
    d.umgedreht      = v.umgedreht;

    auf.erase (auf.begin() + v.entfernpos);
    auf.insert(auf.begin() + v.einfuegpos, d);
    return aktualisiere_neigungen(auf);
}

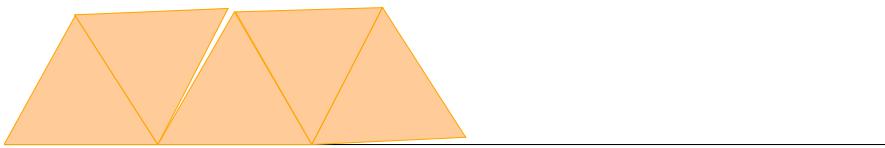
```

Schritt 3 wird dabei in `aktualisiere_neigungen()` implementiert. Sein Quellcode ist nach den Beispielen platziert. Unglücklicherweise habe ich einen Implementierungsfehler unterlassen oder es kommt zu Gleitkommapräzisionsfehlern, sodass viele Schnittpunkte nicht gefunden werden und Dreiecke überlappen können.

### 3 Beispiele

Ich habe jeweils 50 Durchläufe eingestellt, durch eine größere Anzahl und durch Tuning der Parameter ließe sich wahrscheinlich eine Verbesserung erzielen. E steht für Eckendreieck, K für Kantendreieck.

Beispiel 1 - Gesamtabstand 143



D2-E (143| 0) ( 66|119) ( 0| 0)

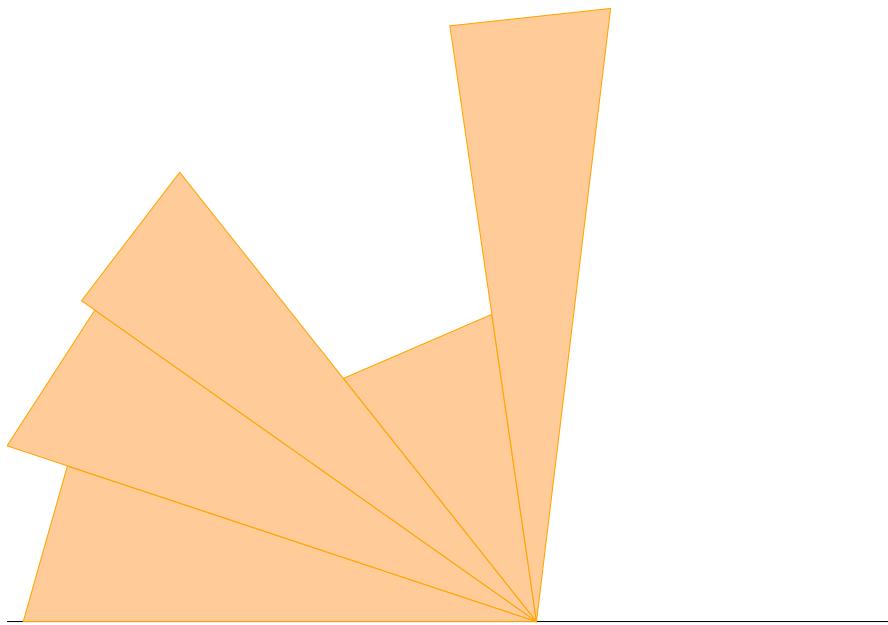
D4-E (208|126) (143| 0) ( 66|120)

D1-K (214|123) (143| 0) (286| 0)

D5-E (286| 0) (215|123) (351|127)

D3-E (351|127) (428| 7) (286| 0)

### Beispiel 2 - Gesamtabstand 0



D1-E (491| 0) ( 15| 0) ( 56|144)

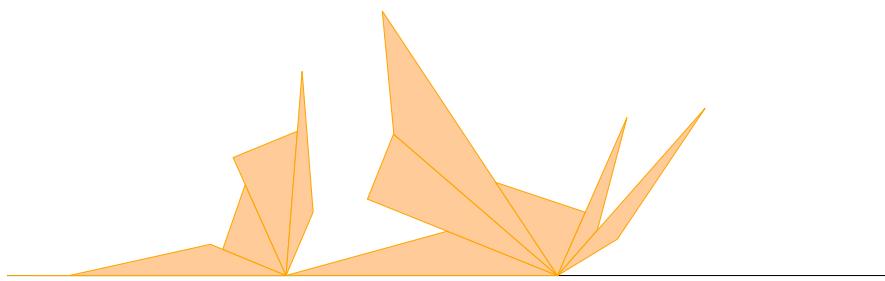
D5-E (491| 0) ( 81|289) ( 0|163)

D4-E (491| 0) ( 69|298) (160|417)

D3-E (491| 0) (450|285) (312|225)

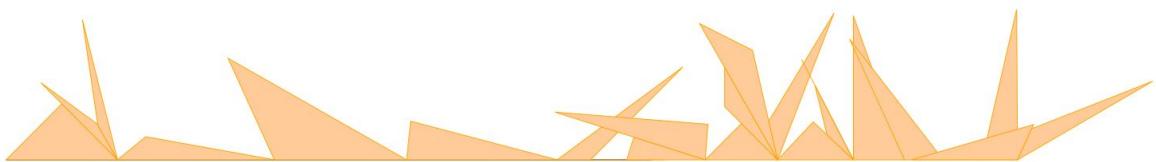
D2-E (491| 0) (560|569) (411|553)

Beispiel 3 - Gesamtabstand 252



D2-E	(259  0)	( 0  0)	(148  0)
D4-E	(259  0)	( 27  0)	(183  0)
D6-E	(189  29)	(259  0)	( 58  0)
D8-E	(221  84)	(200  24)	(259  0)
D7-E	(269 134)	(259  0)	(210 109)
D1-E	(274 189)	(284  59)	(259  0)
D9-K	(259  0)	(409  41)	(511  0)
D5-E	(359 131)	(511  0)	(334  71)
D12-E	(348 245)	(511  0)	(359 131)
D10-E	(511  0)	(537  59)	(454  86)
D3-E	(548  42)	(511  0)	(575 146)
D11-E	(648 155)	(566  34)	(511  0)

Beispiel 4 - Gesamtabstand 1143, Dreiecke überlappen



D16-E	( 0  0)	( 71  71)	(141  0)
D4-E	( 44  97)	(115  45)	(141  0)
D7-E	( 97 177)	(116  44)	(141  0)
D18-K	(141  0)	(177  29)	(341  0)
D6-K	(341  0)	(282 128)	(509  0)
D10-K	(509  0)	(700  0)	(514  49)
D15-K	(742  0)	(859 117)	(700  0)
D19-K	(742  0)	(788  0)	(837  0)
D20-K	(788  0)	(888  0)	(797  29)
D13-E	(697  60)	(891  45)	(888  0)
D5-K	(888  0)	(934  46)	(980  0)
D22-E	(912 117)	(912  67)	(980  0)
D11-E	(948 138)	(980  0)	(880 172)
D17-E	(1051 186)	(980  0)	(968  49)
D23-K	(1025  48)	(980  0)	(1076  0)
D2-E	(1027  94)	(1042  32)	(1076  0)
D9-E	(1076  0)	(1049  52)	(1010 127)
D8-K	(1076 182)	(1076  0)	(1141  0)
D3-K	(1141  0)	(1190  0)	(1071 153)
D1-K	(1247  28)	(1190  0)	(1239  0)
D12-K	(1284  0)	(1283 190)	(1239  0)
D21-E	(1284  0)	(1305  45)	(1150  0)
D14-E	(1456  99)	(1284  0)	(1302  39)

## Literatur

[ADM] Steven S. Skiena:

The Algorithm Design Manual, Second Edition, ISBN 978-1-84800-069-8,  
Kapitel 7.5.3 beschäftigt sich mit Simulated Annealing

[AdW] Peter Rossmannith:

„Algorithmus der Woche“-Artikel über Simulated Annealing,  
<https://www-i1.informatik.rwth-aachen.de/~algorithmus/algo41.php>

[WA] Webseite, um mathematische Ausdrücke zu vereinfachen

<https://www.wolframalpha.com/>

```

bool aktualisiere_neigungen(vector<dreieck>
                            &auf) {
    ld x_beruehr = 0, x_links, y_links, x_rechts,
        y_rechts, x_A, y_A, x_B, y_B, x_E, y_E;
    for(int I = 0; I < N; I++) {
        if(auf[I].eckendreieck) {
            // finde w_{i-1} - alpha_{i-1}
            if(I == 0)
                auf[I].neigung = M_PI;
            else if(auf[I-1].eckendreieck
                    || !auf[I-1].umgedreht)
                auf[I].neigung = auf[I-1].neigung - auf[I-1].inwinkel[auf[I-1].beruehr_n];
            else

```

```

        auf[I].neigung = auf[I-1].neigung - auf[I
                                -1].inwinkel[(auf[I-1].beruehr_n+1)\%3];
    aktualisiere_position(I, x_beruehr, auf);

    // Bezeichnungen setzen
    if(auf[I].umgedreht) {
        // aehnlich
    } else {
        x_links = auf[I].x[(auf[I].beruehr_n+1)\%3];
        y_links = auf[I].y[(auf[I].beruehr_n+1)\%3];
        x_rechts = auf[I].x[(auf[I].beruehr_n+2)\%3];
        y_rechts = auf[I].y[(auf[I].beruehr_n+2)\%3];
    }
    // finde andere Neigungen
    // auf kleinste gefundene aktualisieren
    for(int J = 0; J < I; J++) {
        for(int K = 0; K < 3; K++) { // alle drei Kanten
            x_A = auf[J].x[K];
            y_A = auf[J].y[K];
            x_B = auf[J].x[(K+1)\%3];
            y_B = auf[J].y[(K+1)\%3];
            // pruefen ob lambda existiert fuer k_links
            if(((x_A*x_A - x_B*x_A - x_beruehr*x_A +
                x_B*x_beruehr + y_A*y_A - y_A*y_B) -
                (-x_A*x_A + 2*x_B*x_A - x_B*x_B - y_A*y_A -
                y_B*y_B + 2*y_A*y_B) * (-x_A*x_A + 2*x_beruehr*x_A
                + x_links*x_links - 2*x_beruehr*x_links - y_A*y_A
                + y_links*y_links)) >= 0
                && (x_A*x_A - 2*x_B*x_A + x_B*x_B + y_A*y_A +
                y_B*y_B - 2*y_A*y_B) != 0) {
                // lambda ausrechnen mit Formel
                if(0 <= lambda1 && lambda1 <= 1) {
                    // Schnittpunkt ausrechnen
                    ld x_S = x_A + lambda1*(x_B - x_A);
                    ld y_S = y_A + lambda1*(y_B - y_A);
                    // Neigung ausrechnen
                    if(x_S >= x_beruehr)
                        auf[I].neigung = min(auf[I].neigung,
                                              atan(y_S/(x_S-x_beruehr)));
                    else
                        auf[I].neigung = min(auf[I].neigung,
                                              M_PI - atan(y_S/(x_S-x_beruehr)));
                }
                // das gleiche fuer lambda2
            }
            // das gleiche fuer k_rechts
        }
        for(int E = 0; E < 3; E++) { // alle drei Ecken j-tes Dreieck
            x_E = auf[J].x[E];
            y_E = auf[J].y[E];
            for(int K = 0; K < 3; K++) { // alle drei Kanten i-tes Dreieck
                // aehnlich
            }
        }
    }
    aktualisiere_position(I, x_beruehr, auf);
} else {
    auf[I].neigung = M_PI;
}

```

```

        aktualisiere_position(I, x_beruehr, auf);
        x_beruehr += auf[I].laenge[auf[I].beruehr_n];
    }
}

for(int I = 1; I < N; I++) {
    if(auf[I].eckendreieck)
        continue;
    if(auf[I].umgedreht) {
        if(auf[I-1].eckendreieck) {
            if(auf[I-1].neigung - auf[I].inwinkel[auf[I].beruehr_n]
                < auf[I-1].inwinkel[auf[I-1].beruehr_n])
                return false;
        } else if(auf[I-1].umgedreht) {
            if(auf[I-1].inwinkel[(auf[I-1].beruehr_n+1)\%3]
                + auf[I].inwinkel[auf[I].beruehr_n] > M_PI)
                return false;
        } else {
            if(auf[I-1].inwinkel[auf[I-1].beruehr_n]
                + auf[I].inwinkel[auf[I].beruehr_n] > M_PI)
                return false;
        }
    } else {
        if(auf[I-1].eckendreieck) {
            if(auf[I-1].neigung - auf[I].inwinkel[(auf[I].beruehr_n+1)\%3]
                < auf[I-1].inwinkel[auf[I-1].beruehr_n])
                return false;
        } else if(auf[I-1].umgedreht) {
            if(auf[I-1].inwinkel[(auf[I-1].beruehr_n+1)\%3]
                + auf[I].inwinkel[(auf[I].beruehr_n+1)\%3] > M_PI)
                return false;
        } else {
            if(auf[I-1].inwinkel[auf[I-1].beruehr_n]
                + auf[I].inwinkel[(auf[I].beruehr_n+1)\%3] > M_PI)
                return false;
        }
    }
}

if(auf[N-1].eckendreieck && auf[N-1].neigung
    < auf[N-1].inwinkel[auf[N-1].beruehr_n])
    return false;
return true;
}

```