Aufgabe 2: Dreiecksbeziehungen

Team-ID: 49111

Team-Name: HarperCreekFürHenning187

Bearbeiter/-innen dieser Aufgabe:
Julius Carl Ide

29. April 2019

Inhaltsverzeichnis

1 Lösungsidee

1.1 Exposition

Halbkreis Die Aufgabe besteht darin, die Dreiecke entlang der x-Achse möglichst platzsparend anzuordnen. Die folgende Herangehensweise besteht darin, die Dreiecke halbkreisförmig um einen gemeinsamen Punkt anzuorden. Für diesen Zweck wird nur der kleinste Winkel für jedes Dreieck zur Plazierung in Betracht genommen, da so die Anzahl an Dreiecken, die um einen gemeinsamen Punkt angeordnet sind, signifikant erhöht werden kann, was den Gesamtabstand verrignert.

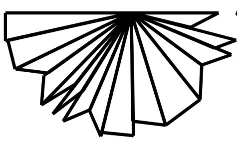


Abbildung 1

Der folgende Algorithmus besteht aus drei Schichten: Als erstes werden die zu plazierenden Dreiecke als Menge T an geordneten Paaren repräsentiert, wobei ein Paar aus dem kleinsten Winkel w, der längsten Strecke l, und den beiden längsten Strecken a, b besteht.

$$t = (w, l, a, b) \tag{1}$$

Die kürzeste Strecke kann vernachlässigt werden, da sie dem kleinsten Winkel immer gegenüberliegt und dementsprechend die Größe eines Halbkreises nicht beinflusst.

Sollte die Summe der kleinsten Winkel aller Dreiecke kleiner als 180° sein, können alle dreieckigen Grundstücke denselben Zugang zur Straße haben und der Gesamtabstand wäre 0. Dies ist in Beispiel 2 der Fall.

$$\sum_{i=1}^{|T|} w_i \leq 180 \tag{2}$$

Leider ist dies nicht immer der Fall, wenn die Anzahl an Grundstücken größer wird.

1.2 Knapsack

Maximierung der Außenseiten Im zweiten Schritt werden, damit der meiste Platz ausgenutzt werden kann, die Dreiecke mit der längsten Strecke, am Anfang und am Ende der Straße gebaut. Das bedeutet, dass möglichst viele lange Dreiecke am Anfang und am Ende der Straße plaziert werden sollen, während die Summe der Winkel immer noch 180° bleibt. Für jedes Dreieck sind dabei 2 Informationen entscheident: Der kleinste Winkel w und die längste Kante v. Die dazugehörige Definition lautet:

$$\max \sum_{i=1}^{n} v_i \tag{3}$$

In diesem Fall ist v_i die längste Strecke, die hier als Maß für Größe genommen wird, des i-ten Dreieckes. Damit der Halbkreis effizient ist, muss aber noch folgendes gelten:

$$\sum_{i=1}^{n} w_i \leq 180 \tag{4}$$

Hierbei ist w_i der kleinste Winkel des i-ten Dreieckes. So kann der meiste Platz ausgenutzt werden. Die Frage lautet nun, wie kann die Größe der Dreiecke am Anfang und am Ende maximiert werden, während die Summe der kleinsten Winkel immer noch 180° beträgt?

Knapsack Zur Lösung dieser Aufgabe eignet sich das Knapsackproblem. Knapsack ist Englisch und bedeutet Rucksack. Das Problem liefert eine Antwort auf die Frage: Wie kann ich meinen Rucksack mit möglichst vielen Wertvollen Gegenständen bepacken, sodass der Gesamtwert so groß wie möglich ist, das Gewichtlimit aber nicht ausgereizt wird? Es gibt verschieden Implementierungen, eine davon ist als 0-1 Knapsack bekannt, da entweder einmal das Objekt im Rucksack ist oder keinmal. Dies ist für dieses Problem auch nützlich, da kein Grundstücksbesitzer zwei Grundstücke möchte, und wird daher verwendet.

Im folgenden Absatz wird zu dem Knapsackproblem im eigenlichen Sinne referiert. Das bedeutet, dass instatt von Halbkreisen, Winkelsummen und der Summe der Längen von einem Knapsack und Elementen mit einem Gewicht und einem Wert gesprochen werden, um das ganze etwas anschaulicher zu erklären.

Um diese Optimierung umzusetzen, muss erstmal folgende Prämisse festgehalten werden: Ein Knapsack mit einer maximalen Kapazität von W hat einen höheren oder gleichen maximalen Profit (Summe der Kosten aller Elemente), als ein kleinerer Knapsack. Das ergibt Sinn, da man in einen Rucksack mit einem höheren Maximalgewicht mindestens alles, was in dem kleineren Rucksack war, hineinstecken kann. Dementsprechend ist es relativ einfach den maximalen Profit eines Rucksacks zu betsimmen, wenn der maximale Profit für alle kleineren Rucksäcke bekannt ist.

Repräsentation als Matrix Dazu wird eine $N \times W$ Matrix M erstellt, wobei N die Anzahl an Elementen ist und W die Kapazität des Knapsacks repräsentiert. Zuerst wird diese Matrix mit 0 für jeden Wert initialisert. Nachdem dies geschehen ist, wird iterativ für jedes Element i, das in den Knapsack gesteckt werden könnte (die Dreiecke), jedes Gewicht j durchlaufen. Dies geschieht durch zwei ineinander geschachtelte for-Loops. Sollte das momentane Gewicht w_i größer als das Gewicht sein, dass gerade durchlaufen wird ($w_i > j$), wird folgendes getan:

$$M_{i,j} = M_{i-1,j} \tag{5}$$

Das maximale Gewicht für den Knapsack in Gleichung 4 ist einfach das des kleineren Knapsack und das i-te Element wird nicht hinzugefügt. Sollte w_i noch in dem Knapsack passen, wird folgende Entscheidung getroffen:

Wenn das Hinzufügen des Elementes lukrativer als das Überspringen ist, wird es hinzugefügt. Dies lässt sich folgendermaßen mathematisch darstellen: Das Gewicht, das bereits in dem Knapsack ist, ist $M_{i-1,j}$, das Gewicht, des Elementes das möglicherweise hinzugefügt wird, beträgt v_i und der maximale Pofit, der für einen Knapsack der Größe j nach dem Hinzufügen des i-ten Elementes $(j - w_i)$ noch erzielt werden kann, ist $M_{i-1,j-w_i}$. Die 1 wird von i subtrahiert, da das i-te Element noch nicht hinzugefügt worden ist. Das Element wird hinzugefügt, wenn die Summe des Wertes des Elementes v_i (die Länge der längsten Kante im i-ten Dreieck) und des maximalen Profits des übrigbleibenden Knapsacks $M_{i-1,j-w_i}$ größer als das Überspringen des Elementes ist $(M_{i-1,j})$. Das ganze lässt sich in folgendes Gleichung zusammenfassen:

$$M_{i,j} = \max(M_{i-1,j}, v_i + M_{i-1,j-w_i}) \tag{6}$$

So wird der maximale Profit für alle Größen eines Knapsacks berechnet. Um nun den maximalen Profit des gesamten Knapsacks zu finden, muss nur das Element $e = M_{N,W}$ aufgerufen werden. e ist aber nur eine Zahl. Die wertvollere Information ist aber, welche Dreiecke ausgesucht wurden.

Um dies festzustellen, muss für jedes Dreieck i, das möglicherweise im Knapsack ist, überprüft werden, ob folgendes gilt:

$$M_{i,j} \neq M_{i-1,j} \tag{7}$$

Sollte die obere Gleichung wahr sein, wird i zu der Menge H der Dreiecke, die am Anfang der Straße sind, hinzugefügt. Diese Gleichung funcktioniert, da, wenn das Element entweder zu schwer (siehe Gleichung 4) oder es wurde aus Effizienzgründen übersprungen (siehe Gleichung 5) wurde, $M_{i,j}$ und $M_{i-1,j}$ denselben Wert enthalten.

Verwendung Die oberen Berechnungen dienen dazu, die Dreiecke, die am Anfang der Straße zu finden sind, zu berechnen. Das Ergebnis wird durch die Menge H repräsentiert. Um mit dem Ergebnis weiterzuarbeiten, muss das relative Komplement der Mengen T, was alle Dreiecke enthält, in H berechnet werden:

$$T_{rest} = \frac{T}{H} = \{x \in T | x \notin H\} \tag{8}$$

Nun wird der gleiche Algorithmus (Knapsack) benutzt, um die Dreiecke, die am besten am Ende der Straße gebaut werden, zu berechnen. Dazu wird die Menge T_{rest} als Eingabe benutzt. Die Menge der Dreiecke, die am Ende der Straße stehen werden, sei E.

$$M = \frac{T_{rest}}{E} \{x \in T_{rest} |\ x \notin E\} \tag{9}$$

Schließlich wurden die Menge T aller Dreiecke in 3 Teilmenge unterteilt.

- (H) die Dreiecke, die am Anfang stehen,
- (M) die Dreiecke, die in der Mitte stehen
- (E) die Dreiecke, die am Ende stehen

$$H \cup M \cup E = T \tag{10}$$

H und E werden der Größe (längste Kante) nach sortiert, so dass das größte Grundstück am weitesten von allen anderen entfernt ist. So kann der Gesamtabstand maximiert werden.

$$\forall i \in \{1, ...|H|\}, H_{i,l} > H_{i+1,l} \tag{11}$$

Nun gibt es zwei Möglichekeiten für M. Entweder ist die Summe kleinsten Winkel der Dreiecke in der Mitte kleiner als 180° oder nicht.

$$\sum_{i=1}^{|M|} w_i \leq 180° \tag{12}$$

Sollte dies der Fall sein, kann die Mitte einfach unverändert bleiben und es müssen keine weiteren Kombinationen mehr vorgenommen werden.

1.3 Der genetische Algorithmus

Unterteilung in Teilmengen Nun kann es jedoch vorkommen das die Summe größer als 180° ist. In diesem Fall muss M in weitere Teilmengen unterteilt werden. Dies ist nur für Beispiel 5 der Fall notwendig. Hierzu kann der Knapsack-Algorithmus nicht verwendet werden, da die Auswahl zu gering ist. Die Halbkreise würden unvollständig bleiben, weil die Summe der Kanten eine höhere Priorität als Vollständig hätte.

Das oberste Gebot bei dem Unterteilen in Teilmengen ist es, Kombinationen aus den Dreiecken zu finden, die summiert $1801/circ$ ergeben, damit die Anzahl an 'Halbkreisen' möglichst klein gehalten wird und kein Platz verschendet wird. Dazu wurde ein rekursiver Algorithmus benutzt, der Elemente eines Arrays schrittweise summiert, bis der Schwellenwert 180 erreicht worden ist.

Die Menge M setzt sich aus allen Dreieicken zusammen.

$$M = \{m_1, m_2, m_{3,...}\} \tag{13}$$

Nun muss eine Teilmenge $Q \subset M, Q = \{q_1, q_{2,q_3,...}\}$ gefunden werden, für die folgendes gilt:

$$\sum_{i=1}^{|Q|} q_i = 180° \tag{14}$$

Dieses Problem ist auch als subset-problem bekannt. Es kann in pseudo-linear Zeit gelöst werden mit einer Komplexität von $O(sN)$, wobei s die Summe und N die Anzahl an Nummern represäntieren. Nachdem eine Teilmenge gefunden worden ist, muss die Prozedur, die für das Knapsackproblem in Gleichung (8), wiederverwendet werden, damit kein Dreiecke zweimal verteilt ist. Dies wird solange wiederholt, bis die verbeleibende Restmenge eine Summe von weniger als 180° enthält. Die Implementierung dieser Funktion wird in Umsetzung diskutiert. Nun wurden alle Dreiecke, die in M enthalten sind, in Teilmengen $(\{Q_1, Q_2, ...Q_n\})$ unterteilt.

$$M = \bigcup_{i=1}^{n} Q_i \tag{15}$$

Der genetische Algorithmus Die Qualität der gefundenen Teilmengen hängt stark von der Reihenfolge der Elemente in M ab. Nun ist die Qualität des Teilmengen zu maxmimieren. Da es zu aufwändig ist alle Möglichkeiten auszuprobieren, muss ein effizienterer Algorithmus gefunden werden.

Ich habe mich für einen genetischen Algorithmus entschieden, da diese gut dafür geeignet sind, eine optimierte Lösung zu finden, wenn das simple Ausprobieren aller Möglichkeiten viel zu lange dauern würde. Während Neuronale Netze das Gehirn imitieren, ahmt ein genetischer Algorithmus die Evolution nach. Das bedeutet, dass die am besten angepassten Individuen jeder Generation überleben und Nachwuchs produzieren. Daher sind drei entscheidende Komponenten zu implementieren:

1. Vererbung
2. Variation
3. Mutation

In der Natur werden die Kriteria auf folgende Weise abgedeckt: Wenn sich zwei Indivduen derselben Spezies paaren, werden in der Meiose die Chromosomen ausgewählt. Dabei kommt es in der Metaphase II zu Variationen. Gelegenlich kann es in der DNA auch zu Mutationen kommen, beispielsweise während der Meitose.

Abstrakt Für diese Aufgabe wurde die DNA folgendermaßen definiert: Aufgrund der Tatsache, dass nur der Gesamtabstand zählt, wird nur der kleinste Winkel und die längste Strecke jedes Dreieckes in Betracht gezogen, da ein kleinerer Winkel mehr Dreiecke im Halbkreis bedeutet, was wiederum bedeutet, dass weniger Halbkreise von Nöten sind, was den Gesamtabstand entscheidend verringert. Zudem gibt die längste Seite genug Auskunft über die Größe eines jeden Dreieckes. Die DNA jedes Individuums setzte sich aus den Teilmengen zusammen, die kombiniert M ergeben.

Algorithm 1 Genetic Algorithm Pseudocode

```
generate random first population
repeat
    breed all individuals
    mutate some individuals
    calculate fitness of everyone
until sufficient individual has been found
```

Im ersten Schritt muss eine Anfangspopulation erzeugt werden. Dazu wird der obere Algorithmus zur Einteilung in Teilmengen benutzt, um alle Dreiecke zu verteilen. Dies wird solange wiederholt, bis die gewünschte Anzahl an Individuen gefunden worden ist. Dabei ist es wichtig, dass die Menge M vor dem nächsten Benutzen zufällig gemischt wird. Andernfalls würde jedes Individuum identisch sein.

Fitness Der entscheidende Punkt ist die Fitnessfunktion, um alle Individuen zu bewerten und den Optimierungsprozess zu beschleunigen. Dazu werden für jede Teilmenge die entsprechenden Koordinaten für die Ecken der Dreiecke berechnet. Anschließend werden die Extremwerte (kleinster und größter y-Wert) für jeden Halbkreis berechnet, um alle Ergebnisse aufzusummieren.

$$\theta_i = \sum_{j=1}^{i} w_j \tag{16}$$

Zuerst wird der Winkel zur Straße für jedes Dreiecke berechent. Die Dreiecke werden übereinander gestapelt, daher wird der Gesamtwinkel auch immer größer.

$$x_i = \cos\theta_i \cdot a, \tag{17}$$

Nun werden die x-Koordinaten für jedes Dreieck berechnet.

$$y_i = \sin\theta_i \cdot a \tag{18}$$

Nun werden die y-Koordinaten für jedes Dreieck berechnet.

$$D(Q) = \sum_{i=1}^{|Q|} |\max(Q_{i,x}) - \min(Q_{i,x})| \tag{19}$$

Hier werden die Länge jedes Halbkreises (größter x-Wert minus kleinster x-Wert) berechnet, um anschließend alle Längen zu summieren, was dem Gesamtabstand enstpricht.

Nachdem der Gesamtabstand für alle Individuen gefunden worden ist, müssen die Besten ausgewählt werden. Die oberste Maxime in dieser Aufgabe ist es den Gesamtabstand zu verringern. Deshalb muss eine Funktion gefunden werden, um einem kleinen Wert von $D(Q)$ eine höhe Fitness zuzuweisen. Dazu wurde einfach $f(D) = \frac{10000}{D^{10}}$ verwendet.

Natural Selcetion Dieser Algorithmus verwendet ein stochastische Asuwahlverahren, um die Individuen für die nächste Generation zu bestimmen. Dazu wird jeder Fitnes eine Wahrscheinlichkeit zu gewiesen: (die Fitness eines Individuums geteilt durch die Fitness der ganzen Population)

$$p_i = \frac{f(D_i)}{\sum_{j=1}^{|D|} f(D_j)} \tag{20}$$

Nun wird aus allen Individuen der Wahrschlichkeit ensprechend neue Individuen für die nächste Generation gezogen.

Paarung Aus allen neuen Individuen, die im oberen Paragraphen bestimmen worden sind, werden zufällig zwei ausgewählt. Um die Chromosomen des Kindes zu bestimmen werden die besten n Halbkreise bestimmt, wobei $0 \leq n \leq |Q|$ gilt ($|Q|$ ist die Anzahl aller Halbkreise eines Individuums). Die Halbkreise werden nach folgenden Kriterien bewert: Wo gut sind die Dreiecke verteilt? Wie klein sind die Grundstücke, die direkt an der Straße liegen?

Um den Grad der Verteilung zu bestimmen, wird die Menge der Dreiecke in zwei gleichgroße Teilmengen unterteilt. Anschließend wird das Verhältnis der längsten Kanten der benachbarten Dreiecke bestimmt.

$$z(Q) = \sum_{j=0}^{\frac{|Q|}{2}} \frac{l_i}{l_{(i+1)}} - \sum_{j=\frac{|Q|}{2}}^{|Q|} \frac{l_i}{l_{(i+1)}} \tag{21}$$

Anschließend wird dazu zu diesem Wert z noch die Summe s der beiden Kanten, der Dreiecke, die den Halbkreis begrenzen und direkt an der Srtaße liegen, hinzuaddiert.

$$f(Q) = z(Q) + s \tag{22}$$

In diesem Fall ist ein kleinerer Wert besser, da das bedeuten würde, dass die Dreiecke besser verteilt sind und der Halbkreis vergleichsweise kurz ist.

Dieser Wert $f(Q)$ wird für alle Halbkreise der beiden Individuen berechnet, um darauffolgend die besten Halbkreise auszuwählen.

Die Dreiecke, die nicht Teil dieser Halbkreise sind, werden durch den *subset-problem algorithm* zufällig neu verteilt.

Verwendung Dieser Algorithmus wird benutzt, um jede neue Generation zu berechnen. Damit keine Zeit verschwendet wird, werden die besten 10% jeder Generation behalten. So kann garantiert werden, dass die Optimierungsrate maximiert wird. Es werden solange neue Generationen produziert, bis einer der folgenden drei Fälle eintritt:

- die Zeit t (in s), die für Berechnungen eingeplant waren, werdne erreicht
- die Grenze g (ein Wert in m) wird unterbunden
- die Generationen konvergieren (das beste Individuum ändert sich nicht über den Verlauf von n Generationen)

Nachdem einer dieser Fälle eintritt, wird das beste Individuum zurückgegen.

1.4 Visualisierung

Das Ergebnis der oberen Berechnung ist eine Menge T mit Teilmengen, die allen Dreiecken einen Halbkreis zuordnet. Nun werden die Kanten für jedes Dreieck berechnet. Es kann vorkommen, dass es immer noch Platz zwischen zwei Dreiecken gibt. Deswegen werden alle Halbkreise solange zusammengeschoben, bis sich zwei Kanten von Dreiecken, die sich nicht im selben Halbkreis befinden, schneiden. So kann die optimale Sequence für jede Menge von dreieckeigen Grunstücken gefunden werden.

2 Umsetzung

Diese Prgamme wurden auf einem Macbookpro mit den Betriebssystem macOs mojave in Python 3 geschrieben.

Für das Programm müssen dies Pfade (*sys.append()*) am Anfang der Dateien geändert werden, damit die Pakete gefunden werden können. Das gesamte Projekt besteht aus zwei Paketen:

HelperMethods In diesem Paket befinden sich alle Dateien, die helfen Dateien zu verwalten und zu Berechnungen dienen, beispielswiese um aus den Koordinaten Winkel und Kantenlängen zu bestimmen.

- **init.py** Diese Klasse dient dazu, die anderen Klassen zu steuern und dient als Schnittstelle.
- **draw.py** Hier wird aus den Koordinaten der Kanten ein visuelles output im .svg Format erzeugt. Dazu wurde das Framwork *svgwrite* benutzt-
- **findIntersection.py** Diese Klasse wird bei dem zusammenschieben benutzt, damit keine zwei Halbkreise überlappen, ohne dass dies festgestellt wird.
- **moveTogether.py** In einem rekursiven *Dynamic Programming* Verfahren werden alle Halbkreise solange zusammengeschoben, bis sich zwei überlappen (siehe *findIntersection*)

- **polygonesToAngles.py** Diese Klasse berechent aus den Koordinaten die Längen aller Kanten und Größe alle Winkel aller Dreiecke.
- **sequenceToEdges.py** Hier werden die Mengen mit Dreiecken, die in dem oberen Algorithmus gefunden worden sind, in ein Format umgewandelt, in dem für jedes Dreieck die Koordinaten (der Kanten) gefunden werden, um dieses Ergebnis visuell auszugeben.
- **textToPolygones** Dieses Programm erwartet eine txt-Datei als Input, um die Koordinaten von allen Dreiecken als Array zu extrahieren.

GeneticAlgorithm Dieses Paket dient dazu für die Kanten, die noch unverteilt waren, Teilmengen unter der Benutzung eines genetischen Algorithmuses zu finden.

- **init.py** Diese Klasse dient dazu, die anderen Klassen zu steuern und dient als Schnittstelle.
- **findBestSequence.py** Hier werden alle Dreiecke in Teilmengen unterteil, die möglichst oft 180° als Summe haben. (siehe *subset-problem*). Dazu wurde *Dynamic Programming* benutzt.
- **new-breeding.py** Hier werden zwei beliebige Individuen für die nächste Generation gepaart.
- **population.py** Dieses Programm sucht die optimale Sequence und simuliert/steuert alle Generationen, bis ein gewünschtes Individuum gefunden worden ist.

Die Knapsackimplementierung[1] befindet sich in der Datei *knapsack.py*.

Main.py Alle diese Klassen werden von der Datei *main.py* gesteuert.

```
beispiel5 = Main("Aufgabe2/examples/dreiecke5.txt")
```

Abbildung 2

Zuerst werden alle Koordinaten aus der txt-Datei extrahiert (siehe *textToPolygones.py*). Anschließend werden diese in Winkel und Streckenlängen umgewandelt, sodass ein Array mit allen Dreiecken in Format wie in Gleichung 1 ensteht.

Nun wird für die Summe der kleinsten Winkel evaluiert, welcher Fall eintritt, sodass die Teilmengen für alle Dreiecke gefunden werden können, was in Variable *sequences* gespeichert wird. Sollte diese kleiner gleich 180° sein, werden alle Dreiecke zusammen als Sequenz abgespeichert. Anschließend wird der Knapsack-Algrithmus zweimal benutzt, sodass ein Anfang und ein Ende zum *sequence*-Array hinzugefügt wird. Für den Fall das die verbleibenden Dreiecke zu viele sind, wird der genetische Algorithmus benutzt. Dazu können die Parameter Größe jeder Generation *pop*, die Dauer *sec* und das Limit verstellt werden. Optional kann noch gestgelegt werden, dass die Ergebnisse als Graph visuell ausgegeben werden (siehe Abbildung 3). Anschließend werden in *drawSequences()* aus den Teilmengen Koordinaten berechent. Ein weiterer Zwischenschritt ist das Ausrichten der Dreiecke für jeden Halbkreise, sodass die das Dreieck mit der längsten Kante in dier Mitte ist und alle anderen der Größe nach darum herum verteilt werden, um den Platz zu minimieren. Schließlich wird noch *moveTogether* benutzt, um alle Halbkreise rekrusiv zusammenzuschieben.

3 Angaben zur Laufzeitanalyse

Der aufwändigste Teil des Programmes passiert erst nachdem die Koordinaten in Dreiecke umgerechnet worden sind. Dabei sind drei verschiedene Fälle zu betrachten:

1. Die Summe ist kleiner als 180 Grad In diesem Fall beträgt die Komplexität $O(N)$, wobei N der Anzahl an Dreiecken enspricht, da nur alle Werte summiert werden müssen.

[1] “0-1 Knapsack Problem in Python - Mike’s Coderama.” Google Sites, sites.google.com/site/mikescoderama/Home/0-1-knapsack-problem-in-p. Boldyreva, Maria. “Dijkstra’s Algorithm in Python: Algorithms for Beginners.” The Practical Dev, dev.to/mxl/dijkstras-algorithm-in-python-algorithms-for-beginners-dkc.

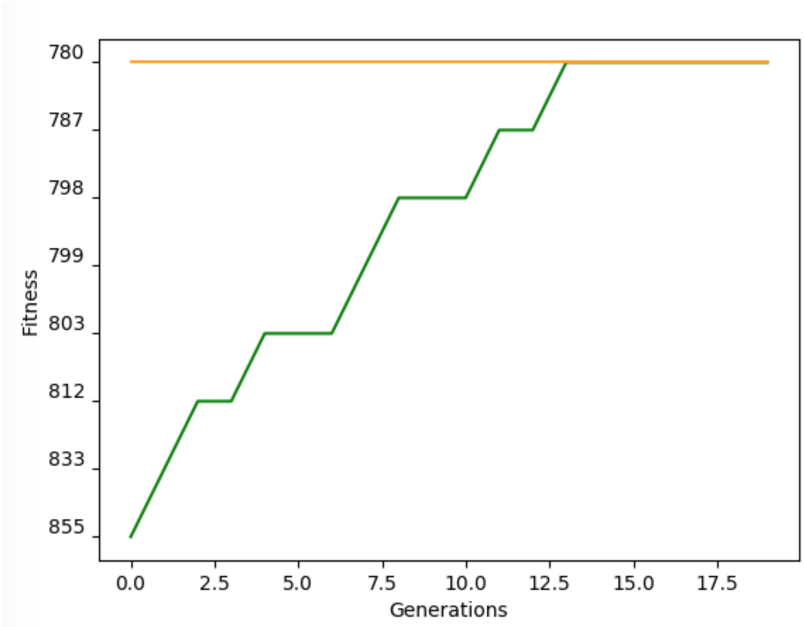


Abbildung 3

2. Knapsack Sollte für die Summe s aller kleinsten Winkel aller Dreiecke $180° < s < 360°$ gelten, wird der Knapsack-Algorithmus zeimal benutzt. Dieser Algorithmus hat eine Komplexität von $O(sN)$, wobei N für die Anzahl an Dreiecken steht und s das maximal Gewicht repräsentiert. Dementsprechend hat dieser Teil eine Komplexität von $O(360N)$.

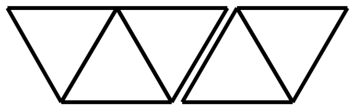
3. Genetischer Algorithmus Der *subset-problem-Algorithmus* hat wie Knapsack eine Komplexität von $O(sN)$. Um alle Dreiecke in Halbkreise zu veteilen müssen in der Regel etwa $\frac{w}{180}$ Teilmengen gefunden werden, wobei w für die Summe aller kleinsten Winkel aller Dreiecke steht. Darausfolgend ist die Komplexität des Finden eines Indiviuums einer Generation etwa $O(w \cdot N)$. Schließlich ist die gesamte Komplexität etwa $O(w \cdot N \cdot g)$, wobei g für die Anzahl an Generationen steht. Für Beispiel 5 lag die Konvergenzrate bei etwa 7 Generationen.

4 Beispiele

```
[Running] python -u "/Users/juliuside/Desktop/bwinfRound2/Aufgabe2/main.py"
input filename:  Aufgabe2/examples/dreiecke1.txt
Total distance:  154
Coordinates:  [{(242, 0), (171, 122), (100, 0)}, {(242, 0), (171, 122), (312, 122)}, {(242, 0), (384, 0), (312, 122)}, {(395.8, 0), (537.8, 0), (466.8, 122)}, {(395.8, 0), (325.8, 122), (466.8, 122)}]
output filename:  output.svg

[Done] exited with code=0 in 0.69 seconds
```

(a)



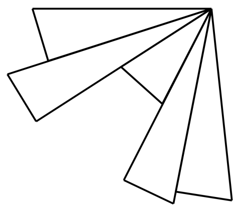
(b)

Abbildung 4: Beispiel 1

```
[Running] python -u "/Users/juliuside/Desktop/bwinfRound2/Aufgabe2/main.py"
input filename:  Aufgabe2/examples/dreiecke2.txt
Total distance:  0
Coordinates:  [{(209, 141), (168, 0), (644, 0)}, {(644, 0), (177, 303), (100, 176)}, {(514, 255), (404, 156), (644, 0)}, {(410, 459), (543, 522), (644, 0)}, {(644, 0), (698, 514), (549, 491)}]
output filename:  output.svg

[Done] exited with code=0 in 0.447 seconds
```

(a)



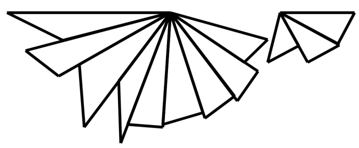
(b)

Abbildung 5: Beispiel 2

```
[Running] python -u "/Users/juliuside/Desktop/bwinfRound2/Aufgabe2/main.py"
input filename:  Aufgabe2/examples/dreiecke3.txt
Total distance:  195
Coordinates:  [{(394, 0), (201, 51), (100, 0)}, {(394, 0), (135, 69), (194, 115)}, {(394, 0), (239, 206), (230, 94)}, {(305, 234), (394, 0), (301, 123)}, {(394, 0), (317, 202), (380, 206)}, {(394, 0), (452, 180), (380, 200)}, {(394, 0), (455, 190), (508, 151)}, {(514, 159), (394, 0), (551, 106)}, {(394, 0), (526, 89), (567, 49)}, {(694.0, 57), (590.0, 0), (724.0, 0)}, {(696.0, 59), (590.0, 0), (639.0, 90)}, {(568.0, 89), (623.0, 60), (590.0, 0)}]
output filename:  output.svg

[Done] exited with code=0 in 0.487 seconds
```

(a)



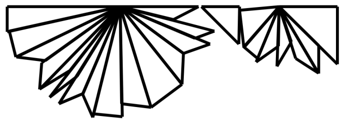
(b)

Abbildung 6: Beispiel 3

```
[Running] python -u "/Users/juliuside/Desktop/bwinfRound2/Aufgabe2/main.py"
input filename:  Aufgabe2/examples/dreiecke4.txt
Total distance:  212
Coordinates:  [{(101, 44), (295, 0), (100, 0)}, {(116, 87), (105, 43), (295, 0)}, {(117, 87), (295, 0), (166, 94)}, {(157, 143), (295, 0), (161, 97)}, {
(295, 0), (208, 111), (158, 142)}, {(174, 156), (295, 0), (220, 120)}, {(231, 151), (295, 0), (191, 167)}, {(295, 0), (248, 192), (231, 152)}, {(298,
191), (295, 0), (249, 185)}, {(295, 0), (348, 174), (297, 167)}, {(400, 134), (295, 0), (348, 174)}, {(400, 134), (295, 0), (404, 82)}, {(404, 82), (295,
 0), (449, 65)}, {(457, 43), (295, 0), (424, 55)}, {(386, 24), (295, 0), (429, 0)}, {(499.8, 0), (434.8, 0), (497.8, 65)}, {(568.8, 0), (667.8, 99),
(668.8, 0)}, {(568.8, 0), (635.8, 116), (635.8, 67)}, {(595.8, 90), (568.8, 0), (634.8, 115)}, {(563.8, 104), (568.8, 0), (587.8, 62)}, {(568.8, 0),
(527.8, 96), (565.8, 69)}, {(506.8, 71), (568.8, 0), (530.8, 91)}, {(499.8, 7), (568.8, 0), (506.8, 72)}]
output filename:  output.svg

[Done] exited with code=0 in 0.567 seconds
```

(a)



(b)

Abbildung 7: Beispiel 4

```
input filename:  Aufgabe2/examples/dreiecke5.txt
Total distance:  743
Coordinates:  [{(303, 0), (183, 23), (100, 0)}, {(233, 21), (186, 22), (303, 0)}, {(227, 87), (303, 0), (206, 29)}, {(273, 106), (227, 87), (303, 0)}, {
(314, 111), (283, 71), (303, 0)}, {(312, 88), (303, 0), (360, 95)}, {(359, 93), (346, 38), (303, 0)}, {(395, 49), (303, 0), (373, 61)}, {(365, 33), (303,
 0), (394, 19)}, {(385, 0), (303, 0), (360, 12)}, {(545, 0), (511, 61), (473, 0)}, {(473, 0), (518, 72), (463, 97)}, {(410, 55), (462, 105), (473, 0)}, {
(473, 0), (407, 0), (417, 49)}, {(670, 57), (622, 0), (692, 0)}, {(604, 74), (672, 59), (622, 0)}, {(550, 35), (599, 94), (622, 0)}, {(551, 35), (545, 0)
, (622, 0)}, {(832, 0), (760, 0), (821, 43)}, {(786, 87), (760, 0), (822, 43)}, {(783, 78), (760, 0), (717, 69)}, {(715, 72), (714, 24), (760, 0)}, {
(697, 0), (692, 36), (760, 0)}, {(834, 0), (873, 16), (887, 0)}, {(834, 0), (886, 90), (897, 25)}, {(834, 0), (878, 77), (832, 79)}, {(985, 56), (944, 0)
, (1009, 0)}, {(959, 69), (971, 38), (944, 0)}, {(961, 76), (944, 0), (901, 72)}, {(897, 35), (944, 0), (928, 28)}, {(910, 25), (944, 0), (901, 0)}, {
(1099, 0), (1195, 55), (1175, 0)}, {(1138, 55), (1189, 52), (1099, 0)}, {(1099, 0), (1105, 70), (1157, 80)}, {(1099, 0), (1104, 58), (1055, 87)}, {(1099,
 0), (1070, 57), (1024, 50)}, {(1099, 0), (1046, 36), (1009, 7)}]
output filename:  output.svg
```

(a)

(b)

Abbildung 8: Beispiel 5

5 Quellcode

```
import sys
sys.path.append("/Users/juliuside/Desktop/bwinf_alles/bwinfRound2/Aufgabe2/helper_methods/")

import matplotlib
matplotlib.use('TkAgg')
import matplotlib.pyplot as plt
import numpy as np
import logging
from collections import namedtuple
import itertools
import random

import GeneticAlgorithm as ga
from findBestSequence import FindBestSequence
import HelperMethods as hm
from knapsack import Knapsack

class Main:
    def __init__(self, filename):
        self.filename = filename

        LOG_FORMAT = "%(levelname)s␣%(asctime)s␣-␣%(message)s"
        logging.basicConfig(filename = "henning.log", level = logging.DEBUG, format = LOG_FORMAT, filem
        logger = logging.getLogger()
        logger.info("beginning")

        triangles = hm.textToTriangles(self.filename)
        self.triangles = list(triangles)
        logger.info("calculating␣the␣triangles")
        sequences = None

        if sum([t[0] for t in triangles]) <= 180:
            sequences = []
            sequences.append(list(triangles))
        else:
            result = self.head(triangles)
            head = result[0]
            triangles = result[1]
            result = self.head(triangles)
            tail = result[0]
            triangles = result[1]

            if sum([t[0] for t in triangles]) <=  180:
                if triangles != []:
                    sequences = [head, triangles , tail]
                else:
                    sequences = [head, tail]
            else:
                pop_size = 20
                sec = 5
                sequences = self.rest(list(triangles))
                sequences.insert(0, head)
                sequences.append(tail)
                sequences = self.run(triangles, pop_size, sec, head, tail)
        self.plotFile()
        self.drawSequences(sequences)

        print("input␣filename:␣", self.filename)
        print("Total␣distance:␣", self.distance)
        print("Coordinates:␣", self.coordinates)
        print("output␣filename:␣", self.filename.replace("txt", "svg"))


    def head(self, triangles):
        LIMIT = 180
        knapsack = Knapsack(triangles, LIMIT)
        head = knapsack.main()
        for h in head:
            triangles.remove(h)
        head = self.sortHead(head)
        return [head, triangles]
```

```
    def sortHead(self, triangles, reverse=True):
        lengths = [t[1] for t in triangles]
        lengths.sort(reverse=reverse)

        sorted = []
        for l in lengths:
            for t in triangles:
                if l == t[1]:
                    sorted.append(t)
                    triangles.remove(t)
                    break

        return sorted

    def rest(self, triangles):
        remainig = []
        while True:
            ks = Knapsack(triangles, 180)
            seq = ks.main()
            for s in seq:
                triangles.remove(s)
            remainig.append(seq)
            if triangles == []:
                break

        return remainig


    def run(self, triangles, pop_size, sec, head, tail):
        add = hm.completeFitness([head, tail]) #hm.sequenceToEdges([self.head, self.tail])[1]
        print(add, "here")
        sequences = ga.optimization(triangles, pop_size, 700.0, sec, add)

        for s in sequences:
            s = self.sortSequence(s)

        sequences.insert(0, head)
        sequences.append(tail)

        return sequences


    def drawSequences(self, sequences):
        fitness = hm.completeFitness(sequences)

        #finding out the ids of the triangles
        _raw = [s for seq in sequences for s in seq]
        tris = list(self.triangles)
        ids = [self.triangles.index(r) + 1 for r in _raw]
        self.ids = ids

        peak = hm.sequenceToEdges(sequences)
        edges = []
        #moved together
        _edges = hm.moveTogether(peak[0])

        if _edges[1] < 0:
            fitness += _edges[1]
            edges = _edges[0]
        else:
            edges = []
            for a in peak[0]:
                for b in a:
                    edges.append(b)

        tris = [edges[i*3:i*3+3] for i in range(int(len(edges) / 3))]
        self.coordinates = [set([tuple(point) for edge in t for point in edge]) for t in tris]

        self.distance = fitness
        hm.save(edges, fitness, "peak.txt")

        hm.draw(edges, 0.9, self.filename.replace("txt", "svg"))
```

```
        #self.plotFile()

    def plotFile(self, file="averages.txt"):
        file = open(file, "r")

        y = [line for line in file]
        x = [i for i in range(len(y))]

        plt.plot(x, y, color='g')
        plt.plot(x, [y[len(y) - 1]] * len(y), color='orange')
        plt.xlabel('Generations')
        plt.ylabel('Fitness')
        plt.show()


    #+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    #the following function will straighten up all the edges, so that the longest edges stick to the to

    def sortSequence(self, sequence):
        """input: sequence as Triangles """
        sorted_triangles = self.indexTriangles(sequence)
        transformed = [None] * len(sorted_triangles)
        peak = int((len(sequence) / 2))

        iterator = True
        leftside = 1
        rightside = 1
        transformed[peak] = sorted_triangles[0]
        sorted_triangles.pop(len(sorted_triangles) - 1)

        for triangle in sorted_triangles:
            if iterator == True:
                transformed[peak - leftside] = triangle
                leftside += 1
                iterator = False
            else:
                transformed[peak + rightside] = triangle
                rightside += 1
                iterator = True

        return transformed

    def indexTriangles(self, triangles):
        largest_lengths = [t[1] for t in triangles]
        largest_lengths.sort(reverse = True)
        sorted_triangles = []

        for a in largest_lengths:
            for b in triangles:
                if a == b[1]:
                    sorted_triangles.append(b)
                    break

        return sorted_triangles



beispiel5 = Main("Aufgabe2/examples/dreiecke5.txt")
```

../bwinfRound2/Aufgabe2/main.py

```
from collections import namedtuple


class Knapsack:
    def __init__(self, triangles, capacity):
        self.triangles = list(triangles)
        self.w = [t[0] for t in triangles]
        self.v =  [t[1] for t in triangles]
        self.capacity = capacity

    def main(self):
        w = self.w
```

```
        v = self.v
        maxCost = self.capacity
        answer = self.zeroOneKnapsack(v,w,maxCost)
        result = []
        for i in range(len(answer[1])):
            if (answer[1][i] != 0):
                result.append(self.triangles[i])
        return result

    def zeros(self, rows, cols):
        row = []
        data = []
        for i in range(cols):
            row.append(0)
        for i in range(rows):
            data.append(row[:])
        return data

    def getItemsUsed(self, w,c):
        # item count
        i = len(c)-1
        # weight
        currentW =  len(c[0])-1

        # set everything to not marked
        marked = []
        for i in range(i+1):
            marked.append(0)

        while (i >= 0 and currentW >=0):
            # if this weight is different than
            # the same weight for the last item
            # then we used this item to get this profit
            #
            # if the number is the same we could not add
            # this item because it was too heavy
            if (i==0 and c[i][currentW] >0 )or c[i][currentW] != c[i-1][currentW]:
                marked[i] =1
                currentW = currentW-w[i]
            i = i-1
        return marked

    # v = list of item values or profit
    # w = list of item weight or cost
    # W = max weight or max cost for the knapsack
    def zeroOneKnapsack(self, v, w, W):
        # c is the cost matrix
        c = []
        n = len(v)
        #  set inital values to zero
        c = self.zeros(n,W+1)
        #the rows of the matrix are weights
        #and the columns are items
        #cell c[i,j] is the optimal profit
        #for i items of cost j

        #for every item
        for i in range(0,n):
            #for ever possible weight
            for j in range(0,W+1):
                #if this weight can be added to this cell
                #then add it if it is better than what we aready have

                if (w[i] > j):

                    # this item is to large or heavy to add
                    # so we just keep what we aready have

                    c[i][j] = c[i-1][j]
                else:
                    # we can add this item if it gives us more value
                    # than skipping it
```

```
                    # c[i-1][j-w[i]] is the max profit for the remaining
                    # weight after we add this item.

                    # if we add the profit of this item to the max profit
                    # of the remaining weight and it is more than
                    # adding nothing , then it't the new max profit
                    # if not just add nothing.

                    c[i][j] = max(c[i-1][j],v[i] + c[i-1][j-w[i]])

        return [c[n-1][W],  self.getItemsUsed(w,c)]
```

../bwinfRound2/Aufgabe2/knapsack.py

```
class FindBestSequence:
    def __init__(self, triangles):
        """
            input: - triangles = list of [smallest_angle, lengths]
        """
        self.vertex = []
        self.triangles = list(triangles)
        self.len = len(triangles)

    @property
    def finalSequence(self):
        LIMIT = 180
        self.findVertices([], self.triangles, LIMIT)
        result = self.result
        return result

    def findVertices(self, vertices, numbers, limit=180):
        """
            finds all sequences
            vetices = points where the road and the triangles meet
        """

        self.findSequence(numbers, limit)
        vertices.append(self.vertex)

        for v in self.vertex:
            numbers.remove(v)

        if numbers == []:
            self.result = vertices
        else:
            self.findVertices(vertices, numbers, limit)


    def findSequence(self, numbers, limit):
        """
            This function finds a sequence (list of numbers that add up to a specific limit).
            In case this is not excatly possible the limit is graduallly decreased.

            input: - numbers: list of
                   - limit: int
        """

        try:
            next(self.subset_sum(numbers, limit))
        except StopIteration:
            self.findSequence(numbers, (limit - 1))
        else:
            self.vertex = next(self.subset_sum(numbers, limit))

    def subset_sum(self, numbers, target, partial=[], partial_sum=0):
        """
            This recursive function adds elements of a list of numbers up to fit a specified limit (tar
        """

        if partial_sum == target:
            yield partial
        if partial_sum >= target:
            return

```

```
        for i, n in enumerate(numbers):
            remaining = numbers[i + 1:]
            yield from self.subset_sum(remaining, target, partial + [n], partial_sum + n[0])
```

../bwinfRound2/Aufgabe2/GeneticAlgorithm/findBestSequence.py

```
from new_breeding import Breeding
from findBestSequence import FindBestSequence


import random
import numpy as np
import time
import logging

import sys
sys.path.append("/Users/juliuside/Desktop/bwinf_alles/bwinfRound2/Aufgabe2")
import HelperMethods as hm

logging
LOG_FORMAT = "%(levelname)s␣%(asctime)s␣-␣%(message)s"
logging.basicConfig(filename = "Aufgabe2/GeneticAlgorithm/population.log", level = logging.DEBUG, forma
logger = logging.getLogger()


class Population:
    def __init__(self, triangles, population_size, treshold, time=None, add=0):
        """
            input: - population_size = integer representing the size of the populations
                   - triangles = list of the smallest angle and largest edge of each triangle
        """
        #making sure that the input has the right type and format
        if (type(population_size) is int) == False or population_size <= 0:
            raise ValueError("population_size␣has␣to␣be␣a␣positive␣integer:␣{}".format(population_size)
        invalid_elements = [i for i in triangles if len(i) != 3 or (type(i[0]) is int) == False or (typ
        if invalid_elements:
            raise ValueError("Wrong␣input,␣every␣elements␣has␣to␣have␣the␣value␣of␣the␣smallest␣angle␣a
        if (type(population_size) is int) == False or population_size <= 0:
            raise ValueError("Wrong␣input␣for␣population_size:␣{},␣has␣to␣be␣an␣int␣value".format(popul
        if (type(treshold) is float) == False or treshold <= 0:
            raise ValueError("treshold␣has␣to␣be␣a␣positive␣float␣value:␣{}".format(treshold))

        self.first_population = list(self.generateSequences(population_size, triangles))
        self.check(triangles, self.first_population)
        self.triangles = triangles
        self.population_size = population_size
        self.treshold = treshold
        self.time = time
        self.add = add

    @property
    def optimizedIndividual(self):
        current_time = time.time()
        if self.time == None:
            timeStamp = None
        else:
            timeStamp = [current_time, self.time]

        self.next_generation(self.first_population, self.treshold, timeStamp)

        return self.peak

    def next_generation(self, population, treshold, time_treshold, averages=[], bestOne=None, i=None):
        """
            input: - treshold = desired minimum fitness
                   - time_treshold = [start_time, amount of time you want to let it evolve]
                   - crossovers = rate in percent (0 ... 1.0)
                   - mutation = rate in percent (0 ... 1.0)
        """
        if time_treshold != None:
            if (time_treshold[0] + time_treshold[1]) <= time.time():
                print("******TERMINATED******")
                _max = max([self.fitness(i) for i in population])
                peak = [a for a in population if self.fitness(a) == _max][0]
```

```
                self.peak = peak
                self.writeFile(averages)

                return None



        logger.info("start")

        bests = 0.5 # for keeping the best
        toBeKept = self.keepBest(list(population), int(len(population) * bests))
        logger.info("keep␣best")

        selected = self.selectIndividuals(population)
        logger.info("select")

        #breed the new population
        next_pop = self.breed(selected, 1)
        logger.info("breed")

        if averages != []:
            print(averages[len(averages) - 1])

        l = ([self.fitness(i) for i in population])
        peak = max(l)
        _best = [a for a in population if self.fitness(a) == peak][0]
        if self.reversedFitness(peak) <= int(treshold):
            self.peak = _best
            self.writeFile(averages)
            return None

        averages.append(self.reversedFitness(peak))

        #exchange individual if it is better than its ancestor
        for i in range(int(bests * self.population_size)):
            j = random.randint(0, self.population_size - 1)
            if self.fitness(next_pop[j]) <= self.fitness(toBeKept[i]):
                next_pop[j] = toBeKept[i]

        self.next_generation(next_pop, treshold, time_treshold, averages)

    def check(self, triangles, pop):
        for p in pop:
            try:
                raw = []
                for seq in p:
                    for s in seq:
                        raw.append([s[0], s[1], s[2]])

                for t in triangles:
                    v = [t[0], t[1], t[2]]
                    raw.remove(v)
            except:
                print("Wrong␣triangles")
                print(p)
                exit()

    def fitness(self, basepairs):
        bs = basepairs
        bs = [a for a in bs if a != []]
        f = hm.fitness(bs)
        f += self.add
        for seq in bs:
            a_sum = sum([s[0] for s in seq])
            if a_sum > 180:
                return 0.00001

        return (100000/f) ** 10

    def reversedFitness(self, number):
        root = number ** (10**-1)
        return int(100000 / root)

```

```
    def generateSequences(self, size, triangles):
        for _ in range(size):
            random.shuffle(triangles)
            yield FindBestSequence(triangles).finalSequence

    def switchSequences(self, pop, activation):
        for p in pop:
            r = random.randint(0, 100)
            if r <= int(activation * 100):
                random.shuffle(p)
        return pop

    #+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    def keepBest(self, population, number):
        """
            number: number of individuals to keep
        """
        toBeKept = []
        for _ in range(number):
            _max = max([self.fitness(i) for i in population])
            peak = [a for a in population if self.fitness(a) == _max][0]

            toBeKept.append(peak)
            population.remove(peak)

        return toBeKept


    def writeFile(self, averages):
        f = open("averages.txt", "w")
        for a in averages:
            f.write(str(a) + "\n")
        f.close()

    def breed(self, population, crossovers):
        new_pop = [] # the next generation

        for _ in range(int(len(population))): #because each parent will get one child
            a = random.choice(population)
            b = random.choice(population)
            child = None
            if self.fitness(a) > self.fitness(b):
                child = Breeding(a, b, random.randint(0, len(a) - 1)).child()
            else:
                child = Breeding(b, a, random.randint(0, len(a) - 1)).child()
            new_pop.append(child)

        return new_pop



    def selectIndividuals(self, population):
        total_fitness = sum([self.fitness(ind) for ind in population])
        weights = [self.fitness(p)/total_fitness for p in population]
        choices = list(np.random.choice(len(population), len(population), p=weights))

        new_pop = []

        for c in choices:
            new_pop.append(population[c])

        return new_pop
```

../bwinfRound2/Aufgabe2/GeneticAlgorithm/population.py