Aufgabe 1: Lisa rennt

Teilnahme-Id: 48825

Bearbeiter dieser Aufgabe:
Sammy Sawischa

29. April 2019

Inhaltsverzeichnis

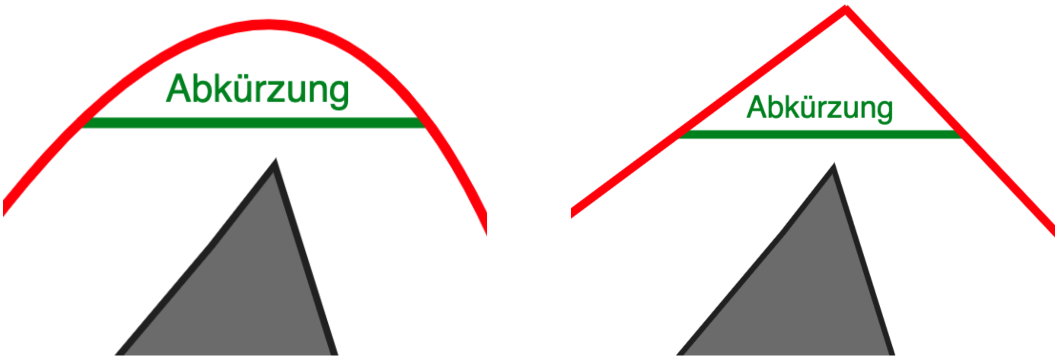
1 Lösungsidee

Der Aufgabe liegt das Problem der optimalen Wegfindung (engl. Pathfinding) zugrunde. Dabei soll der optimale Weg von einem Punkt L (Lisas Haus) zu einem anderen Punkt B (ihr Auftreffen mit dem Bus) ermittelt werden, wobei Polygone Hindernisse darstellen. Zusätzlich dazu besteht eine Schwierigkeit der Aufgabe darin, den optimalen Zielpunkt auf der Y-Achse zu bestimmen. Dieser hängt nämlich von Lisas Geschwindigkeit, der Busgeschwindigkeit, den Koordinaten von Lisas Haus und von der Länge des gefundenen Wegs an den Polygonen vorbei ab.

1.1 Erstellen eines Sichtbarkeitsgraphen

Um den optimalen Weg zwischen Lisas Haus und einem beliebigen Punkt auf der Y-Achse zu finden, kann im ersten Schritt die vorliegende Umgebung mit den Polygonen als Graph interpretiert werden, der die Eckpunkte eines Polygons als Knoten enthält. Genauer gesagt handelt es sich bei diesem Graph um einen Sichtbarkeitsgraphen (engl. Visibility Graph), bei dem die Kanten jene Knoten (hier: Eckpunkte) verbinden, die sich sehen können. Anschließend müssen noch Start – und Endknoten dem Sichtbarkeitsgraphen hinzugefügt werden.

Es gilt: Der kürzeste Weg s von Punkt L zu B liegt stets auf dem Sichtbarkeitsgraphen, d.h., dass s ein Polygonzug ist, der verschiedene Eckpunkte der gegebenen Polygone direkt verbindet (sofern Polygone den Weg von L zu B verdecken). Der Beweis wird durch die Abbildungn 1.1 und 1.2 deutlich. So gibt es keine polynomische Linie p, die nicht lokal abgekürzt werden kann. Zum Beispiel kann in Abbildung 1.1 p (in Rot) durch die grüne Linie abgekürzt werden. Da jeder kürzeste Weg auch lokal am kürzesten sein muss, kann s nicht polynomisch sein. Ebenfalls kann ein Polygonzug, der einen Punkt nicht auf einem Eckpunkt eines Polygons hat, durch eine gerade Linie abgekürzt werden (s. Abbildung 1.2). Folglich muss s ein Polygonzug sein, der auf den Eckpunkten der Polygone liegt, da dieser nicht abgekürzt werden kann. Somit liegt s auf dem Sichtbarkeitsgraphen.



(a) Abkürzung einer polynomischen Linie (b) Abkürzung von zwei geraden Linien

Abbildung 1: Zwei Beispiele lokaler Abkürzungen

Des Weiteren hat s als Punkte immer nur konvexe Ecken eines Polygons als Punkte, da s die Polygone umgehen soll. Eine gerichtete Kante zu einem Knoten mit konkavem Eckpunkt wäre dabei kontraproduktiv. Um die konvexen Punkte eines Sichtbarkeitsgraphen zu bestimmen geht man wie folgt vor: Man schaut sich für jede Polygonkante k an, ob sie nach rechts oder links gerichtet ist im Vergleich zur vorherigen Kante k-1. Dazu setzt man einfach die entsprechenden x- und y-Werte der relevanten Punkte in die Orientierungsmatrix[1]:

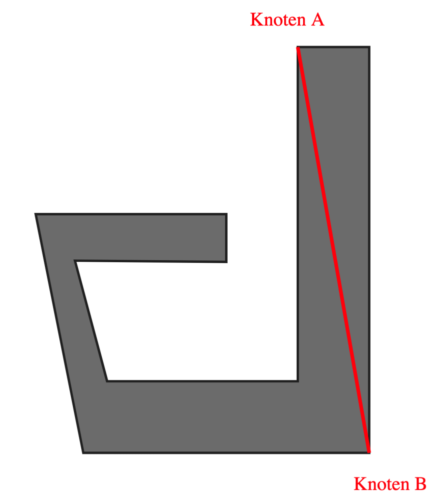
$$O = \begin{pmatrix} 1 & x_a & y_a \\ 1 & x_b & y_b \\ 1 & x_c & y_c \end{pmatrix}$$

A ist der Start-Punkt auf der vorherigen Kante. B ist der Punkt der von k und k-1 geteilt wird, während C der Endpunkt von k ist. Ist die Determinante dieser Matrix negativ, so ist k im Verhältnis zu k-1 rechtsgerichtet. Ist sie positiv, so ist k linksgerichtet. Für den Fall $|O| = 0$ gilt, dass A, B und C auf einer Linie liegen, was aber hier ausgeschlossen werden kann, da zwei Polygonkanten nicht im 0 Grad Winkel aneinander liegen. In einem Polygon haben die Kanten aller konkaven Eckpunkte stets dieselbe Orientierung und alle konvexen die gegenteilige. Es sind also entweder alle Kanten der konvexen Eckpunkte linksgerichtet und alle konkaven rechtsrum oder andersrum. Ist ein konvexer Punkt bekannt, weiß man, dass alle anderen Eckpunkte deren Kanten die gleiche Orientierung haben ebenfalls konvex sind. Um einen konvexen Eckpunkt zu bestimmen, kann man einfach den obersten linken Eckpunkt nehmen, der ohnehin konvex sein muss, da er am Äußersten des Polygons liegt.

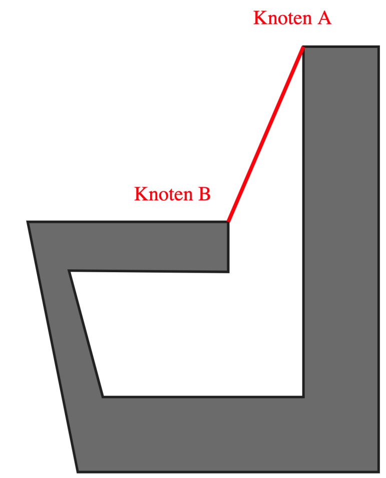
Um nun den Sichtbarkeitsgraphen erstellen zu können, muss geprüft werden, ob zwei Knoten A und B sich sehen. Dies ist genau dann der Fall, wenn die Verbindung v der beiden Knoten weder von Kanten eines Polygons verdeckt wird noch durch einen Eckpunkt eines Polygons geht, denn in dem Fall werden die beiden zugehörigen Kanten des Eckpunkts nicht richtig geschnitten (Berührungen zählen nicht, müssen aber in Betracht gezogen werden). Dabei muss zuvor die Fallentscheidung gemacht werden, ob die Knoten A und B zum selben Polygon gehören. Ist dies der Fall, so besteht die Möglichkeit, dass zwei Knoten eines Polygons sich nicht sehen, obwohl keine Kante die Verbindung v verdeckt und es keine Schnittpunkte mit Eckpunkten gibt (auch hier: Berührungen zählen nicht). Dies wird in Abbildung 2.1 deutlich. Zum Unterscheiden dieser Situation mit der in Abbildung 2.2, kann überprüft werden, ob der Punkt in der Mitte von v innerhalb des Polygons liegt oder nicht. Tut er dies, so können sich die Knoten A und B nicht sehen, andernfalls schon.

Ob ein Punkt P sich in einem Polygon befindet kann mithilfe des Punkt-in-Polygon-Tests nach Jordan[2] ermittelt werden. Dabei kann man einen virtuellen Strahl schicken, der horizontal durch das Polygon und den Punkt P geht. Anschließend werden die Schnittpunkte des Strahls mit den Kanten jeweils links und rechts von P gezählt. Ist die Anzahl jeweils ungerade, liegt P im Polygon, ist sie stattdessen jeweils gerade liegt P außerhalb des Polygons. Es gibt den Sonderfall, dass der Strahl durch einen Eckpunkt geht, sodass theoretisch zwei Kanten an diesem Punkt einen Schnittpunkt haben, obwohl nur eine gezählt

[1] https://en.wikipedia.org/wiki/Curve_orientation
[2] https://de.wikipedia.org/wiki/Punkt-in-Polygon-Test_nach_Jordan



(a) Verbindung liegt im Polygon

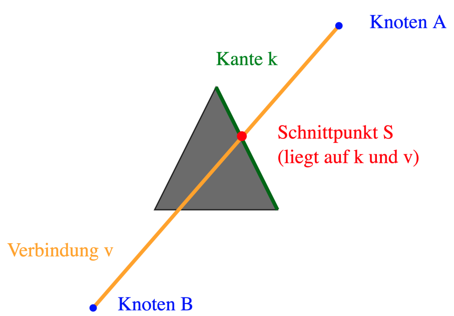


(b) Verbindung liegt außerhalb des Polygons

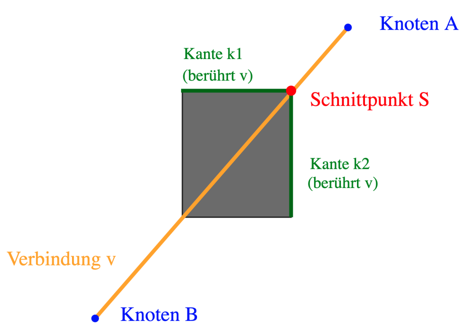
Abbildung 2: Prüfen, ob zwei Knoten des selben Polygons sich sehen können

werden sollte. Dies kann vermieden werden, indem man vorher definiert, dass der Punkt auf dem Strahl als Punkt über dem Strahl gezählt wird, sodass eine Kante über dem Strahl liegt und keinen Schnittpunkt hat und eine andere wiederum einen Punkt oberhalb des Strahls hat und einen unterhalb, wodurch diese dann einen Schnittpunkt mit dem Strahl hat.

Für den Fall, dass zwei Knoten A und B nicht in einem Polygon liegen, muss also lediglich geprüft werden, ob dessen Verbindung v von einer Kante aller Polygone verdeckt wird oder durch einen Eckpunkt durchgeht. Eine Kante k verdeckt dann die Sichtbarkeit von A und B, wenn sie einen Schnittpunkt mit v hat (s. Abbildung 3.1). Zur Überprüfung, ob also k die Verbindung v schneidet, kann man beide Strecken als Geraden fortführen und anschließend deren Schnittpunkt S berechnen. Liegt dieser sowohl auf k als auch v, so ist Knoten A nicht für Knoten B sichtbar, und umgekehrt. Um zu prüfen, ob S auf v liegt kann man einfach das Teilverhältnis des Vektors $\overrightarrow{AS}$ zu $\vec{v}$ (nichts Anderes als $\overrightarrow{AB}$) ermitteln und man erhält den Faktor, mit dem v multipliziert werden muss, um auf S zu treffen. Sofern dieser Faktor größer als 0 und kleiner als 1 ist, liegt S auf v. Dasselbe Prozedere gilt für die Kante bzw. Strecke k. Sind nun beide Faktoren größer als 0 und kleiner als 1, dann verdeckt k eindeutig v, wodurch A und B nicht sichtbar wären. Weiterhin müssen die Kanten in Betracht gezogen werden, die v nur berühren, wenn also bei k der Faktor 0 oder 1 beträgt um auf S zu treffen. Gibt es zwei dieser Kanten k1 und k2, die einen Punkt gemeinsam haben, so handelt es sich hierbei um einen Eckpunkt durch den die Verbindung v geht (s. Abbildung 3.2). Damit Kanten, die mit Sicherheit außerhalb des von Knoten A und B aufgespannten Rechtecks liegen, nicht unnötig auf ihren Schnittpunkt überprüft werden, werden sie nicht beachtet, da diese die Verbindung v nicht schneiden können.



(a) Verbindung schneidet Kante k



(b) Verbindung geht durch einen Eckpunkt

Abbildung 3: Prüfen, ob zwei Knoten verschiedener Polygone sich sehen können

Laufzeitanalyse bei der Erstellung des Sichtbarkeitsgraphen:

Gegeben seien n Polygoneckpunkte. Daraus folgt, dass es auch insgesamt n Polygonkanten gibt, da ein Polygon genauso viele Ecken wie Kanten hat. Im Grunde genommen muss bei jedem Eckpunkt bzw. Knoten geprüft werden, ob er für jeden anderen Eckpunkt sichtbar ist. Dieser Vorgang hat die Zeitkomplexität $O(n^2)$. Dabei wird dieser Vorgang optimiert, indem bereits geprüfte Paare von Eckpunkten ausgelassen werden können. Zum Testen, ob zwei Knotenpunkte sich sehen können muss jede Polygonkante in Betracht gezogen werden, dies geschieht in $O(n)$. Damit erhält man insgesamt für den Algorithmus eine Laufzeit von $O(n^3)$, was für den gestellten Aufgabenbereich auf jeden Fall passabel ist, zumal der Sichtbarkeitsgraph nur einmal erstellt werden muss.

Optimierung des Sichtbarkeitsgraphen:

Der Sichtbarkeitsgraph wie er beschrieben worden ist, weist jetzt noch einige Redundanzen auf, die noch zu entfernen sind. So hat zum Beispiel jeder konvexe Eckpunkt einen Bereich B (s. Abbildung 4), der durch eine Fortführung der Kanten dieses Eckpunkts entsteht. Alle gerichteten Kanten von einem Knoten, der sich in B befindet, zu einem anderen des zugehörigen Bereichs sind redundant[3]. Durch Ausnutzen dieser Eigenschaft lässt sich der Sichtbarkeitsgraph etwas optimieren. In Beispiel 5 konnten dadurch 406 gerichtete Kanten auf 336 reduziert werden und in Beispiel 4 wurden 308 Kanten auf 248 reduziert, was in beiden Fällen einer durchschnittlichen Reduzierung von rund 22 % entspricht.

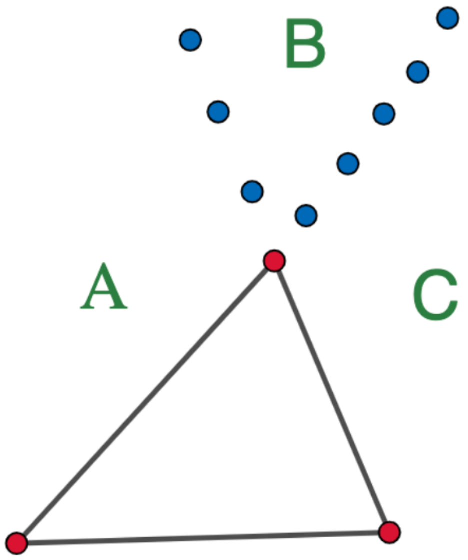


Abbildung 4: Jeder Eckpunkt hat einen zugehörigen B-Bereich

1.2 Anwenden des A*-Algorithmus

Zur Bestimmung des kürzesten Wegs zwischen Lisas Haus und einem Punkt auf der Y-Achse kann auf den zuvor erstellten Sichtbarkeitsgraphen der A*-Algorithmus[4] angewandt werden. Hierbei handelt es sich um einen informierten Suchalgorithmus, der eine Heuristik (Schätzfunktion) bei der Auswahl des zunächst besuchenden Knoten einbezieht. Beim uninformierten Dijkstra-Algorithmus wäre der Wert dieser Heuristik 0, weshalb der A*-Algorithmus im Grunde genommen eine Erweiterung des Dijkstra-Algorithmus ist. Durch diese Heuristik ist die Suche bei A* zielgerichtet, da jeder Knoten die euklidische Distanz zum Endknoten speichert (h_{cost}), sodass abgeschätzt werden kann, ob der zu besuchende Knoten wahrscheinlich schnell zum Ziel führt. Genauer gesagt werden die Gesamtkosten f_{cost} eines Knotens X berechnet, die sich folgendermaßen zusammensetzen: $f_{cost} = g_{cost} + h_{cost}$

Die Werte für g_{cost} sind dabei die Kosten für den Weg von X zum Startknoten. In dem Fall ist es also die Summe der Kanten des Sichtbarkeitsgraphen, die zu X führen. Die Länge einer solchen Kante

[3] Anders Strand-Holm Vinther & Magnus Strand-Holm Vinther. 2015. Pathfinding in Two-dimensional Worlds, S.33.
[4] `https://de.wikipedia.org/wiki/A*-Algorithmus`

kann ebenfalls mit der euklidischen Distanz berechnet werden, was praktisch die Anwendung des Satzes des Pythagoras ist, um den Abstand zweier Koordinaten zu berechnen. Der A*-Algorithmus ist bei der Bearbeitung der Aufgabe äußerst hilfreich und sinnvoll, da er drei Eigenschaften mit sich bringt:

1. Vollständigkeit (Gibt es eine Lösung, so wird diese zurückgegeben)
2. Optimalität (Es wird immer der optimale Weg zurückgegeben)
3. Optimale Effizienz

Im Detail funktioniert A* so, dass es drei verschiedene Kategorien gibt, mit denen Knoten markiert werden können:

1. Unbekannte Knoten: Knoten, die noch nicht besucht worden sind.
2. Offene Knoten: Knoten, deren Gesamtkosten bekannt sind. Ihre nachfolgenden Knoten wurden möglicherweise noch nicht untersucht.
3. Abgeschlossene Knoten: Knoten, bei denen die Gesamtkosten der nachfolgenden Knoten bekannt sind.

Der Suchalgorithmus beginnt nun beim Startknoten und berechnet für alle folgenden Knoten die Gesamtkosten $\mathrm{f_{cost}}$. Diese werden alle als offene Knoten markiert und zeigen auf den Startknoten (mithilfe der Zeiger wird später der Pfad rekonstuiert). Anschließend werden die nachfolgenden Knoten des offenen Knoten mit den niedrigsten Gesamtkosten weiter untersucht. Dieser offene Knoten wird anschließend als abgeschlossen markiert. Die neu untersuchten Knoten zeigen nun auf den gerade geschlossenen Knoten. Dieser Prozess wird solange wiederholt, bis es keine offenen Knoten mehr gibt (dann wurde keine Lösung gefunden), oder es sich bei dem offenen Knoten mit den niedrigsten Gesamtkosten um den Endknoten handelt. In diesem Prozess kann es passieren, dass der Wert für $\mathrm{g_{cost}}$ bei einem Knoten verschieden sein kann, je nach dem von welchen vorherigen Knoten man ihn besucht. In diesem Fall nimmt man einfach den niedrigeren Wert für $\mathrm{g_{cost}}$ und ändert den Zeiger dieses Knotens so, dass er zu dem Knoten zeigt, der den geringeren $\mathrm{g_{cost}}$ hat.

1.3 Ermitteln des optimalsten Wegs für Lisa

Zunächst kann der optimalste Punkt auf dem y-Achsenabschnitt bestimmt werden, bei dem Lisa auf den Busfahrer trifft. Optimal heißt dabei, dass Lisa ihr Haus so spät wie möglich verlassen muss. Man stelle sich also vor, es gäbe keine Hindernisse. Da Lisa und der Bus sich mit einer gleichförmigen Bewegung fortbewegen (Geschwindigkeit v ist konstant), gilt:

$$\mathrm{s_{Bus}} = \mathrm{v_{Bus}} * \mathrm{t_{Bus}} <=> \mathrm{t_{Bus}} = \frac{\mathrm{s_{Bus}} * 3s}{25m}$$
$$\mathrm{s_{Lisa}} = \mathrm{v_{Lisa}} * \mathrm{t_{Lisa}} <=> \mathrm{t_{Lisa}} = \frac{\mathrm{s_{Lisa}} * 6s}{25m}$$

Damit Lisa ihr Haus so spät wie möglich verlassen muss, muss die Differenz $t = \mathrm{t_{Bus}} - \mathrm{t_{Lisa}}$ möglichst groß sein. Der Wert für t gibt dabei die Zeit (orientiert an der Abfahrt des Busses um 7:30) in Sekunden an, an der Lisa ihr Haus verlassen muss. Für t(x) gilt (wobei x die Entfernung in Metern vom Startpunkt des Busses ist, bei der Lisa und der Bus sich treffen):

$$t(x) = \frac{x * 3}{25} - \frac{\mathrm{s_{Lisa}} * 6}{25}$$

Wie man sieht ist t(x) immer von Lisas Strecke bis zum Auftreffpunkt abhängig. Im speziellen Fall ohne Hindernisse gilt für die Strecke Lisas lediglich die euklidische Distanz vom Standort Lisas Haus $(s_x|s_y)$ und dem Auftreffpunkt A(0|x). Dann gilt:

$$t(x) = \frac{x * 3}{25} - \frac{\sqrt{s_x^2 + (s_y - x)^2} * 6}{25}$$

Der optimalste Punkt ist im Grunde genommen der x-Wert des Maximums dieser Funktion. Da im Falle von Hindernissen die Strecke Lisas zum Punkt A stets abweichen kann, müssen mehrere Möglichkeiten für x in Betracht gezogen werden. Genauer gesagt kann man für x immer ganzzahlige Zahlen einsetzen und zwar jene, die zwischen den y-Werten des Knotens, der am weitesten unten liegt und des Knotens, der am weitesten oben liegt. Liegt der optimalste Punkt nicht in diesem Bereich, so muss dieser Bereich entsprechend nach oben oder unten justiert werden. Jeder andere Punkt außerhalb des Bereichs muss nicht in Betracht gezogen werden, da die Zeit t(x) nur noch kleiner werden könnte.

2 Umsetzung

Die Lösungsidee wurde in Python implementiert. Nach Starten des Programms kann man im Interface eine Umgebung auswählen. Anschließend wird die optimalste Route berechnet und ausgegeben mit allen weiteren Details. Zusätzlich befindet sich im output-Ordner eine generierte SVG-Datei mit einer Visualisierung des optimalsten Wegs und dem dazugehörigen Sichtbarkeitsgraphen.

Zur Umsetzung der Lösungsidee wurden mehrere Klassen implementiert. Die Klassen Point, Edge, Node und Polygon dienen dabei als Baupläne für Objekte, die instanziiert werden können. Des Weiteren beinhalten diese Klassen objektspezifische Hilfsmethoden, auf die im Folgenden genauer eingegangen wird (triviale Methoden werden ausgelassen). Die anderen Klassen VisibilityGraph, AStar, LisaRennt enthalten lediglich statische Methoden und dienen zur besseren Strukturierung.

`class Point`

(repräsentiert einen Punkt mit seinen x- und y-Koordinaten)

def euclidean_distance(self, other_point)	Berechnet die euklidische Distanz zweier Punkte mithilfe des Satz des Pythagoras.

`class Edge`

(repräsentiert eine Strecke mit zwei definierten Punkten)

def get_intersection_with (self, other_edge)	Gibt den Schnittpunkt zweier Strecken zurück, die als Geraden fortgeführt werden. Um den Schnittpunkt der fortgeführten Kanten zu erhalten, müssen dessen Geradengleichungen gleichgesetzt werden: $m_1 * x + b_1 = m_2 * x + b_2 \quad \mid - b_1$ $<=> m_1 * x = m_2 * x + b_2 - b_1 \quad \mid -(m_2 * x)$ $<=> (m_1 - m_2) * x = b_2 - b_1 \quad \mid : (m_1 - m_2)$ $<=> x = \frac{b_2 - b_1}{m_1 - m_2}$ Durch Umformen erhält man den x-Wert des Schnittpunkts bei $(b2 - b1)/(m1 - m2)$. Zudem wird in der Methode der Fall beachtet, wenn die eine Kante parallel zur y-Achse ist, da dann die Steigung unendlich wäre.
def is_out _of_area (self, point a, point b)	Prüft, ob eine Kante sich ausserhalb des von zwei Punkten A und B aufgespannten Rechtecks befindet. Dies geschieht lediglich durch mehrere if-Verzweigungen, weshalb diese Methode nicht im Quellcode der Dokumentation aufgeführt wird.
def get_ratio (self, point_a, point _b)	Gibt den Faktor zurück, mit dem die Kante erweitert werden muss, um auf einen Punkt X (der auf der fortgeführten Kante als Gerade liegt) aufzutreffen. Dazu wird das Teilverhältnis des Vektors $\overrightarrow{AB}$ zu $\overrightarrow{AX}$ berechnet.

`class Node`

(repräsentiert einen Knoten im Sichtbarkeitsgraphen)

def can_see_(self, node_b)	Prüft, ob der aktuelle Knoten A einen anderen Knoten B sehen kann. Wenn beide zu einem Polygon gehören und sich eine Kante teilen, dann wird direkt True zurückgegeben. Andernfalls wird durch jede Kante aller Polygone iteriert und auf Schnittpunkte überprüft. Sollte eine Kante die Verbindung von A zu B lediglich berühren, so wird diese der Liste potential_obstacles hinzugefügt (Verbindung könnte durch einen Eckpunkt gehen). Abschließend wird geprüft, ob Kanten in dieser Liste einen gemeinsamen Punkt haben. Ist dies der Fall und Knoten A und B sind nicht auf dieser Kante enthalten, wird False zurückgegeben. Kommt es weder zu einer Verdeckung einer Kante noch zu einem Schnittpunkt durch einen Eckpunkt, wird True zurückgegeben.
def is_redundant _to (self, node_2)	Prüft, ob eine Kante sich ausserhalb des von zwei Punkten A und B aufgespannten Rechtecks befindet. Dies geschieht lediglich durch mehrere if-Verzweigungen, weshalb diese Methode nicht im Quellcode der Dokumentation aufgeführt wird.

`class Polygon`

(repräsentiert ein Polygon mit seinen Eckpunkt und einer ID)

def contains (self, point)	Prüft, ob sich ein Punkt im Polygon befindet. Es handelt sich dabei um die Implementierung des Punkt-in-Polygon-Tests nach Jordan.
def convex_vertices _only (self)	Gibt alle konvexen Punkte eines Polygons zurück. Dazu durchläuft eine for-Schleife jeden Eckpunkt und berechnet die zugehörige Orientierungs-Matrix. Anschließend wird determiniert, ob konvexe Eckpunkte jene sind, die einen Links-Turn machen oder jene die einen Rechts-Turn machen, indem der äußerste Punkt links oben ermittelt wird und als konvex vorausgesetzt wird. Zur Ermittlung dieses konvexen Punkts wird zu Beginn durch jeden Punkt iteriert und verglichen.

`class VisibilityGraph`

(enthält alle essentiellen Methoden zur Erstellung eines Sichtbarkeitsgraphen)

@staticmethod def get_graph(to_visit, current_graph)	Diese Methode gibt einen Sichtbarkeitsgraphen als Hash-Table zurück. Es handelt sich hierbei um eine rekursive Methode, die zu Beginn eine Liste der zu prüfenden Knoten enthält und einen Startgraphen mit den zu prüfenden Knoten als Schlüssel in einer Hash-Table/Dictionary. In der Methode wird dann durch jeden zu besuchenden Knoten iteriert und auf Sichtbarkeit mit den noch zu besuchenden Knoten geprüft. Falls diese vorhanden sein sollte und nicht redundant ist, wird bei beiden Knoten in der Hash-Table jeweils der andere Knoten ergänzt (Die Werte der Hash-Table entsprechen Listen mit den jeweils sichtbaren Knoten). Durch den Algorithmus müssen Knotenpaare nur einmal überprüft werden.
@staticmethod def add_node(to_add, visibility_graph)	Methode zum Hinzufügen eines Start- oder Endknotens in einen bereits vorhandenen Sichtbarkeitsgraphen. Dies erfolgt nach dem selben Prinzip wie in get_graph, nur dass hier nur ein Knoten mit jedem anderen auf Sichtbarkeit überprüft werden muss.

`class AStar`

(enthält Methoden zur Implementation des A*-Algorithmus)

@staticmethod def get_shortest_path(visibility_graph)	Gibt den kürzesten Pfad und dessen Länge in einem Sichtbarkeitsgraphen in Form einer Liste von Knoten dieses Pfades zurück. Hierbei handelt es sich um eine direkte Implementierung des A*-Algorithmus, wie er in der Lösungsidee beschrieben wurde. Die Knoten werden dabei markiert, indem sie zu der jeweiligen Liste hinzugefügt werden (Offene Knoten gelangen in open_list, geschlossene in closed_list und noch nicht bekannte Knoten in keine der beiden). Durch die while-Schleife werden solange alle sichtbaren Knoten des aktuell offenen Knotens (der mit den geringsten Gesamtkosten) geprüft, bis es keine mehr in open_list gibt. In dem Fall gäbe es keinen Pfad. Durch eine for-Schleife gegen Ende der while-Schleife wird durch alle offenen Knoten iteriert und derjenige mit den geringsten Gesamtkosten ermittelt. Handelt es sich hierbei um den Endknoten, so kann der kürzeste Pfad rekonstruiert werden.
@staticmethod def reconstruct_path(end_node)	Ermittelt den kürzesten Pfad und dessen Länge, indem so lange alle vorherigen Knoten des Endknotens zur Liste shortest_path hinzugefügt werden, bis es sich bei einem vorherigen Knoten um den Startknoten handelt, da dann der Pfad vollständig rekonstruiert worden ist.

`class LisaRennt`

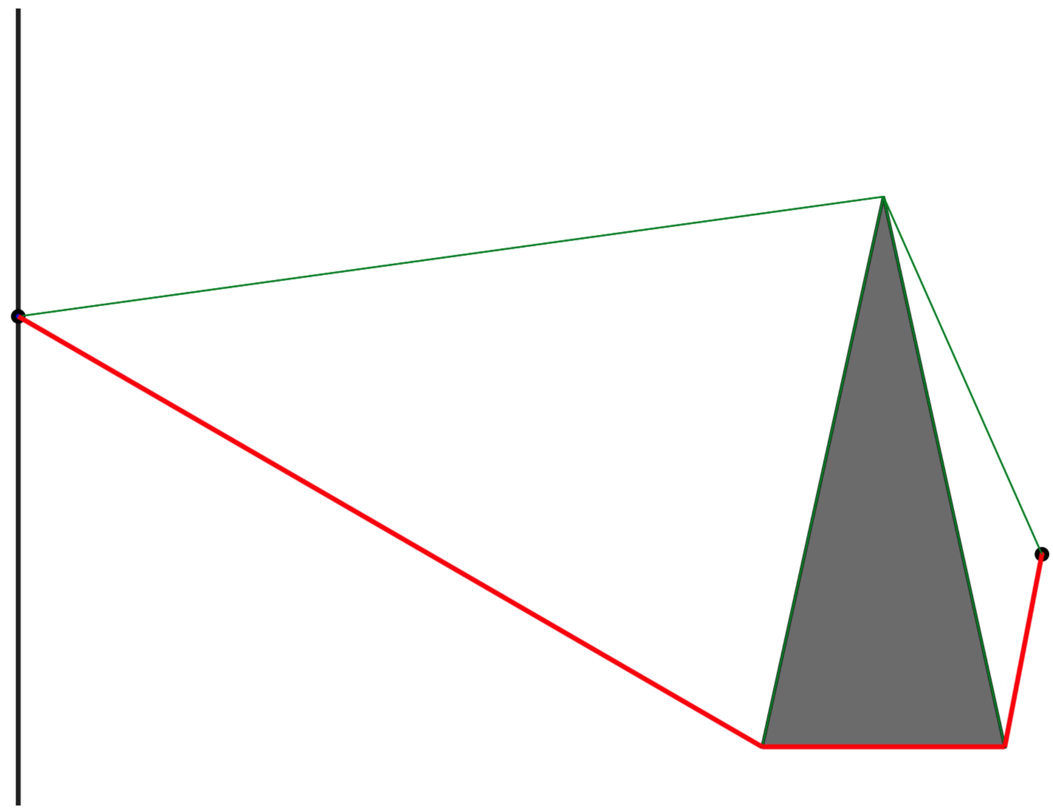
(enthält die Methode zum finalen Lösen der Aufgabe)

@staticmethod def get_optimal_result()	In dieser Methode wird der Sichtbarkeitsgraph initialisiert und um den Startknoten erweitert. Anschließend wird der zu prüfende Bereich ermittelt, in dem der Endknoten des optimalsten Pfads für Lisa auf der y-Achse liegt. Anschließend werden in einer for-Schleife alle ganzzahligen Zahlen dieses Bereichs ausprobiert. Die beste Lösung (der Pfad, bei dem Lisa am spätesten aufstehen muss) wird als best_path gespeichert und zusammen mit dessen Sichtbarkeitsgraph zurückgegeben.

3 Beispiele

Im Folgenden werden alle Programmausgaben sowie die erzeugten SVG-Bilder der gegebenen Beispiele dargestellt. Zusätzlich dazu wurde ein eigenes Beispiel kreiert, um einen Spezialfall zu zeigen.

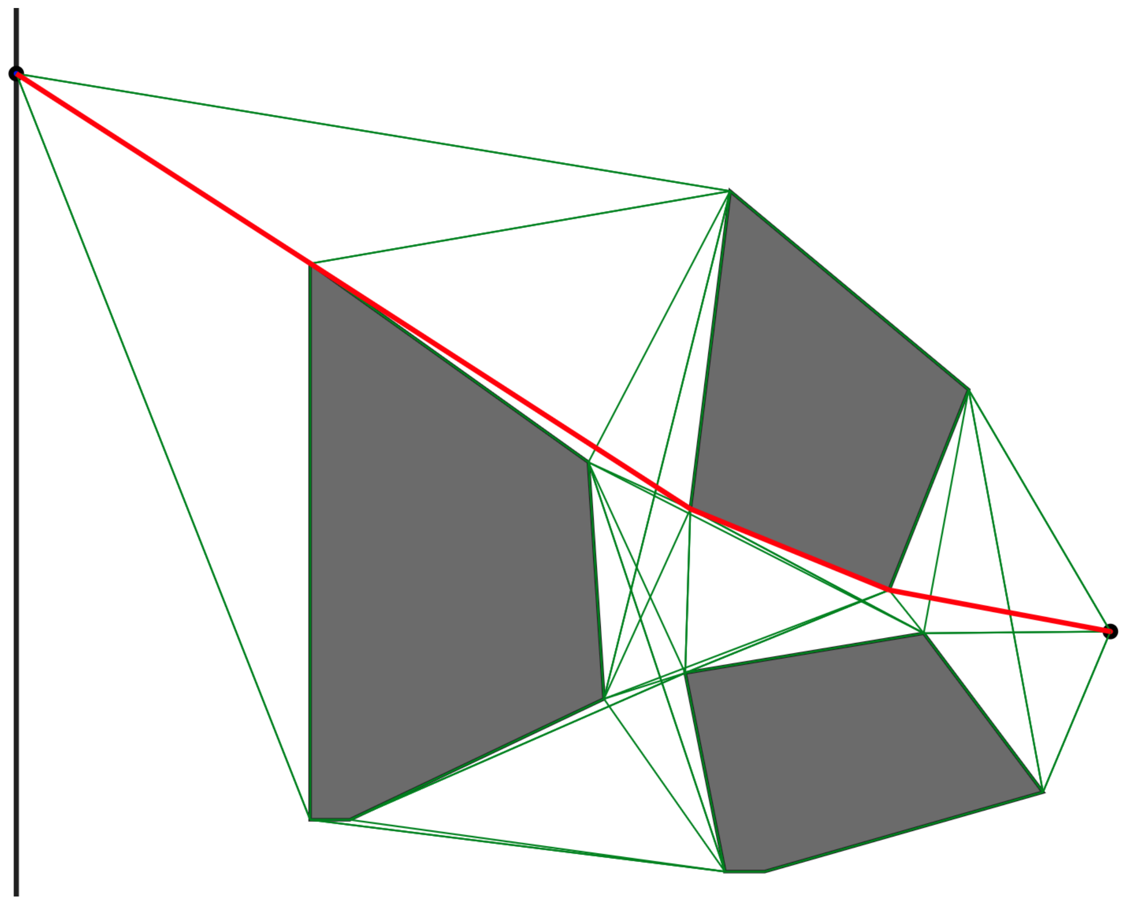
lisarennt1.txt



```
Startzeit: 7:26:59
Zielzeit: 7:30:40
y-Koordinate des Auftreffens: 336 Meter
Laenge von Lisas Route: 924 Meter
Dauer von Lisas Route: 3.7 Minuten

Lisas Route:
[L, (633 | 189)] --> [P1, (610 | 70)] --> [P1, (460 | 70)] --> [Y, (0 | 336)]
```

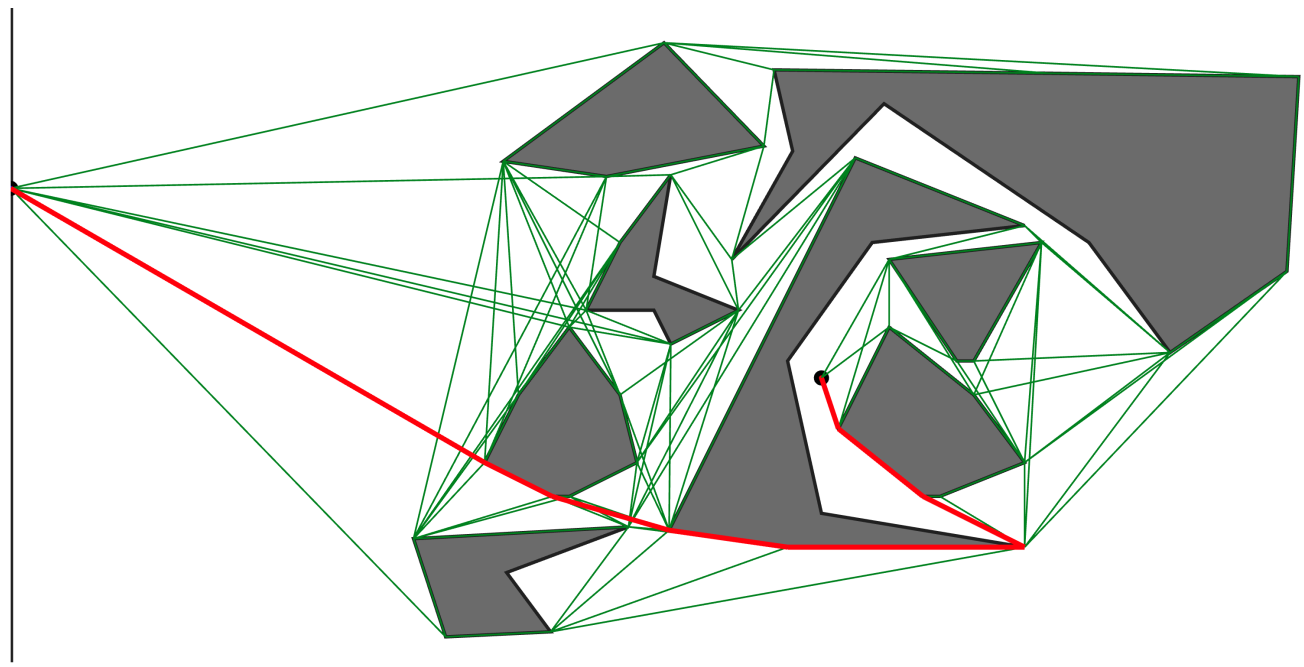
lisarennt2.txt



```
Startzeit: 7:27:38
Zielzeit: 7:31:01
y-Koordinate des Auftreffens: 512 Meter
Laenge von Lisas Route: 849 Meter
Dauer von Lisas Route: 3.4 Minuten

Lisas Route:
[L, (633 | 189)] --> [P1, (505 | 213)] --> [P1, (390 | 260)] --> [Y, (0 | 512)]
```

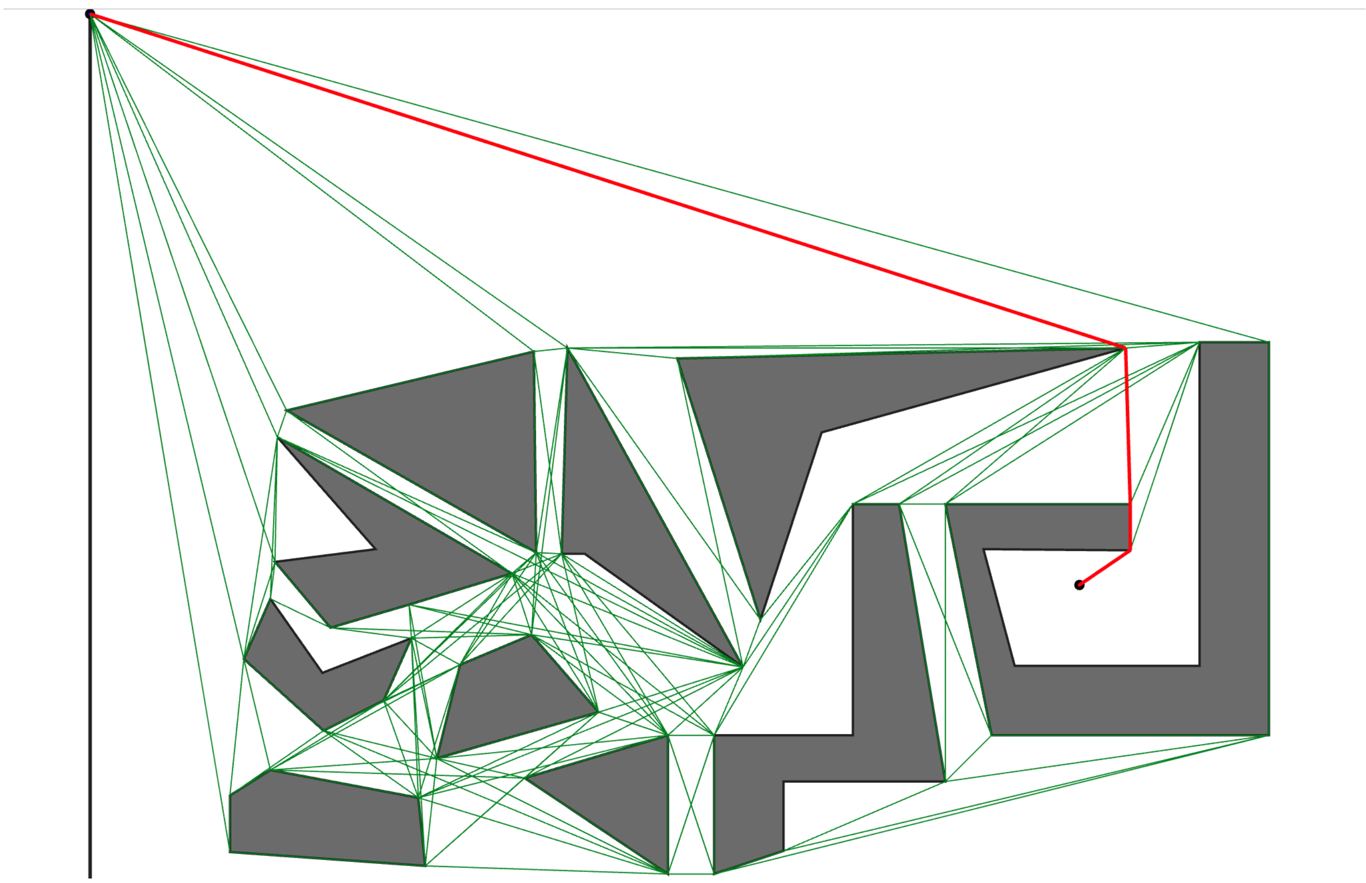
lisarennt3.txt



```
Startzeit: 7:27:11
Zielzeit: 7:30:33
y-Koordinate des Auftreffens: 280
Laenge von Lisas Route: 845 Meter
Dauer von Lisas Route: 3.38 Minuten

Lisas Route:
[L, (479 | 168)] --> [P1, (489 | 138)] --> [P1, (539 | 98)] --> [P3, (599 | 68)]
--> [P3, (459 | 68)] --> [P3, (389 | 78)] --> [P4, (320 | 98)] --> [P4, (280 | 118)]
--> [Y, (0 | 280)]
```

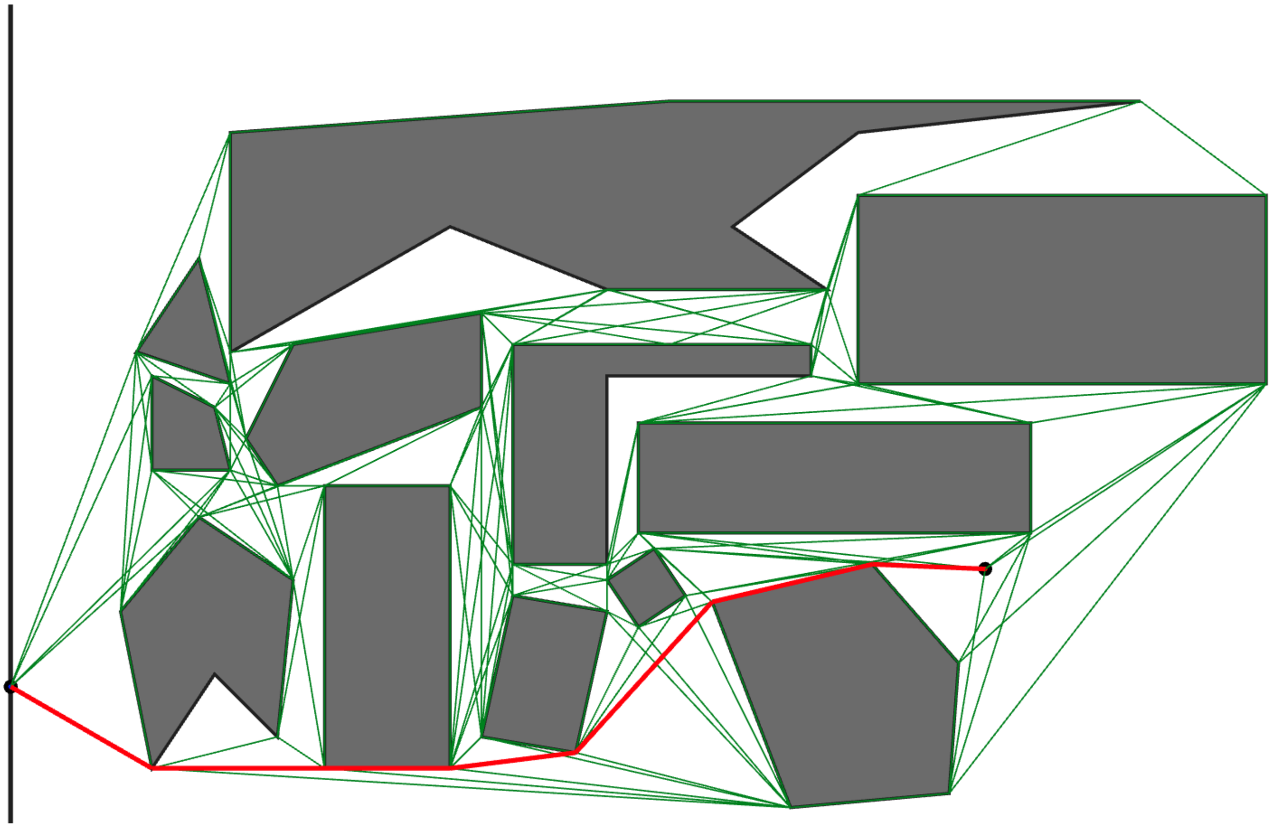
<u>lisarennt4.txt</u>



```
Startzeit: 7:26:38
Zielzeit: 7:31:31
y-Koordinate des Auftreffens: 764 Meter
Laenge von Lisas Route: 1223 Meter
Dauer von Lisas Route: 4.89 Minuten

Lisas Route: [L, (856 | 270)] --> [P10, (900 | 300)] --> [P10, (900 | 340)]
--> [P9, (896 | 475)] --> [Y, (0 | 764)]
```

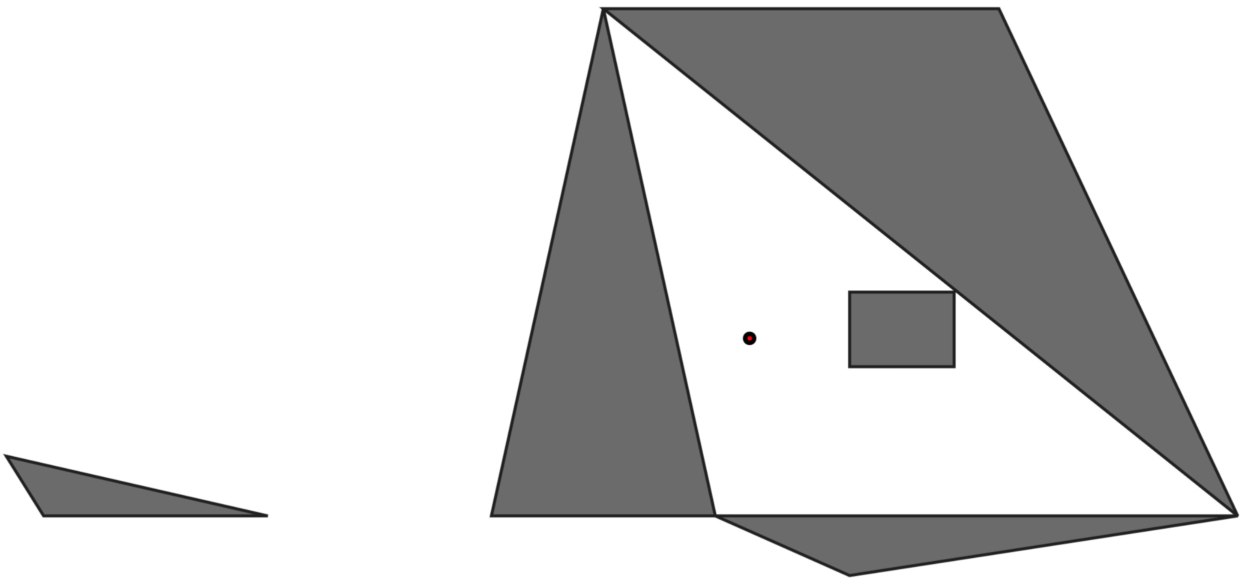
lisarennt5.txt



```
Startzeit: 7:27:10
Zielzeit: 7:30:10
y-Koordinate des Auftreffens: 87 Meter
Laenge von Lisas Route: 753 Meter
Dauer von Lisas Route: 3.01 Minuten

Lisas Route: [L, (621 | 162)] --> [P2, (549 | 165)] --> [P2, (447 | 141)]
--> [P4, (360 | 45)] --> [P5, (280 | 35)] --> [P7, (90 | 35)] --> [Y, (0 | 87)]
```

eigenesbeispiel.txt



```
Ausgabe
Es gibt keinen Pfad zum Ziel
```

Wie man sieht findet das Konstrukt verschiedenster Algorithmen den optimalen Weg für Lisa in den Beispielen 1-5. Ebenfalls kann herausgefunden werden, ob ein Pfad existiert, oder nicht, wie das Programm im eigenen Beispiel erkannt hat, bei dem zudem noch Polygone aneinander liegen. Die optimalen Pfade wurden dabei innerhalb weniger Sekunden berechnet, eine Skalierung mit mehreren Polygonen ist daher kein Problem.

4 Quellcode

```
# Klasse, die allgemeine Methoden zur Loesung der Aufgabe enthaelt
class LisaRennt:

    # Gibt den optimalsten Pfad mit Sichtbarkeitsgraphen zurueck,
    # bei dem Lisa ihr Haus so spaet wie moeglich verlassen muss
    @staticmethod
    def get_optimal_result():
        to_visit = []       # Liste aller Knoten, die auf
                            # Sichtbarkeit ueberprueft werden muessen
        start_graph = {}    # Anfangsgraph mit allen Polygonecken als Knoten

        # Konvexe Ecken aller Polygone als Ausgangsknoten hinzufuegen
        for polygon in Input.polygon_list:
            for convex_vertice in polygon.convex_vertices_only():
                # Erstellung eines Knotens aus einem konvexen Eckpunkt
                converted_node = Node(convex_vertice, polygon.id)
```

```python

                # Heuristik fuer Knoten berechnen und hinzufuegen
                converted_node = converted_node.assign_h_cost(Input.end_node)

                # Knoten jeweils initialisieren
                to_visit.append(converted_node)
                start_graph[converted_node] = []

        # Liste aller Kanten generieren
        # (Zur Erstellung des Sichtbarkeitsgraphen erforderlich)
        for polygon in Input.polygon_list:
            Input.list_of_all_edges.extend(polygon.convert_to_edges())

        # Sichtbarkeitsgraph mit Start- aber ohne Endknoten
        visibility_graph = VisibilityGraph.get_graph(to_visit, start_graph)
        visibility_graph = VisibilityGraph.add_node(Input.start_node, visibility_graph)

        # Untersten und obersten Knoten ermitteln (um Bereich zu definieren),
        # Startknoten als Referenzwert
        smallest_y = Input.start_node.point.y
        biggest_y = Input.start_node.point.y
        for node in visibility_graph.keys():
            if node.point.y < smallest_y:
                smallest_y = node.point.y
            if node.point.y > biggest_y:
                biggest_y = node.point.y

        # Maximum von t(x), s. Dokumentation
        optimum_y = fmin(lambda x: -LisaRennt.t(x), 0, disp=False)

        # Bereich justieren, indem optimaler Weg gesucht wird,
        # da das Optimum ein Teil dieses Bereichs sein muss
        if optimum_y < smallest_y:
            smallest_y = int(floor(optimum_y))
        if optimum_y > biggest_y:
            biggest_y = int(ceil(optimum_y))

        # Ermitteln des optimalsten Wegs durch Ausprobieren
        # aller moeglichen Endknoten im vordefinierten Bereich
        full_visibility_graph = VisibilityGraph.add_node(Node(Point(0,smallest_y), -2),
                                                         deepcopy(visibility_graph))
        best_visibility_graph = deepcopy(full_visibility_graph)
        calc_path = AStar.get_shortest_path(full_visibility_graph)
        if calc_path is None:
            return  # Kein Pfad
        best_path = calc_path
        Output.highest_time_after_departure = LisaRennt.time(smallest_y, calc_path[1])
        for i in range(smallest_y+1, biggest_y):
            full_visibility_graph = VisibilityGraph.add_node(Node(Point(0, i), -2),
                                    deepcopy(visibility_graph))
            calc_path = AStar.get_shortest_path(full_visibility_graph)
            time_after_departure = LisaRennt.time(i, calc_path[1])

            if time_after_departure > Output.highest_time_after_departure:
                Output.highest_time_after_departure = time_after_departure

                best_path = calc_path
                best_visibility_graph = deepcopy(full_visibility_graph)
```

```python

        # Rueckgabe des optimalsten Wegs (inkl. Laenge) und Sichtbarkeitsgraph
        return best_path, best_visibility_graph

    # Zeitdifferenz ohne Hindernisse
    @staticmethod
    def t(x):
        # Optimaler Punkt (0|x) auf der y-Achse, wenn es keine Hindernisse gibt
        s_x = Input.start_node.point.x  # Standort Lisa, x-Wert
        s_y = Input.start_node.point.y  # Standort Lisa, y-Wert
        return (3*x/25.0) - (6 * sqrt(s_x**2 + (s_y-x)**2))/25.0

    # Zeitdifferenz mit Hindernissen
    @staticmethod
    def time(x, path_length):
        return (3*x/25.0) - (6*path_length)/25.0


class VisibilityGraph:

    # Gibt einen Sichtbarkeitsgraph als Hash-Table zurueck (Rekursive Methode)
    @staticmethod
    def get_graph(to_visit, current_graph):

        # Es muessen keine Knoten mehr besucht werden
        if len(to_visit) == 1:
            return current_graph

        # Knoten wird abgearbeitet
        current_node = to_visit[0]
        to_visit.pop(0)

        for node in to_visit:
            if node.can_see(current_node):
                if not node.is_redundant_to(current_node):
                    current_graph[current_node].append(node)
                if not current_node.is_redundant_to(node):
                    current_graph[node].append(current_node)

        # Naechsten zu besuchenden Knoten pruefen,
        # Erweiterung des aktuellen Sichtbarkeitsgraphen
        return VisibilityGraph.get_graph(to_visit, current_graph)

    # Methode zum Hinzufuegen eines Start- oder Endknotens
    @staticmethod
    def add_node(to_add, visibility_graph):
        all_key_nodes = visibility_graph.keys()
        visibility_graph[to_add] = []

        # Jeden Knoten mit zu ergaenzendem Knoten auf Sichtbarkeit und Redundanz pruefen
        for node in all_key_nodes:
            if node.can_see(to_add):
                    if node.id >= 0:
                        if not node.is_redundant_to(to_add):
                            visibility_graph[node].append(to_add)
                            visibility_graph[to_add].append(node)
                    else:
                        visibility_graph[node].append(to_add)
```

```python
                        visibility_graph[to_add].append(node)

        return visibility_graph

# Klasse, die alle wichtigen Methoden zum A* Pathfinding-Algorithmus enthaelt
class AStar:

    # Findet den kuerzesten Pfad in einem Sichtbarkeitsgraphen
    @staticmethod
    def get_shortest_path(visibility_graph):
        start_node = Input.start_node
        open_list = [start_node]
        current_node = start_node
        closed_list = []

        # Solange die Open-Liste nicht leer ist
        while len(open_list) != 0:

            # Alle sichtbaren Knoten des aktuellen Knotens
            visible_nodes = visibility_graph[current_node]

            # Knoten wird als abgeschlossen markiert
            closed_list.append(current_node)
            open_list.remove(current_node)

            # Iteriere durch sichtbare Knoten des aktuellen Knotens
            for i in range(len(visible_nodes)):
                visible_node = visible_nodes[i]
                # Berechne gCost (Abstand zum Startknoten)
                new_g_cost = current_node.g_cost
                new_g_cost += current_node.point.euclidean_distance(visible_node.point)

                # Wenn sichtbarer Knoten als abgeschlossen markiert
                # und der neue Pfad zu ihm nicht guenstiger ist
                if visible_node in closed_list and \
                   new_g_cost >= visibility_graph[current_node][i].g_cost:
                    continue

                # Falls Knoten noch nicht entdeckt oder ein guenstigerer Pfad gefunden wurde
                if visible_node not in open_list or \
                   new_g_cost < visibility_graph[current_node][i].g_cost:
                    # Aktualisieren der gCost und Gesamtkosten fuer diesen Knoten
                    visibility_graph[current_node][i].g_cost = new_g_cost
                    visibility_graph[current_node][i].f_cost = new_g_cost + visible_node.h_cost

                    # Setzen des Zeigers auf den vorherigen Knoten
                    visibility_graph[current_node][i].previous_node = current_node

                    # Falls Knoten noch nicht entdeckt
                    if visible_node not in open_list:
                        # Knoten wird als offen markiert
                        open_list.append(visible_node)

            # Erhalte Knoten mit geringsten Gesamtkosten
            if len(open_list) > 0:
                best_next_node = open_list[0]
                for node in open_list:
                    if node.f_cost < best_next_node.f_cost:
```

```
                        best_next_node = node
                current_node = best_next_node

            else:
                return None     # Kein Pfad konnte gefunden werden

            # Wenn End-Knoten Knoten mit geringsten Gesamtkosten ist
            if current_node.id == -2:
                # Optimaler Pfad wurde gefunden, muss rekonstruiert werden
                return AStar.reconstruct_path(current_node)

        # Es konnte kein Pfad gefunden werden
        return None

    # Methode zur Rekonstruierung des optimalsten Wegs
    @staticmethod
    def reconstruct_path(end_node):
        shortest_path = [end_node]
        previous_node = end_node.previous_node
        total_length = end_node.point.euclidean_distance(previous_node.point)

        # Pfad erweitern mithilfe des Vorgaenger-Knotens, Erhoehung der Gesamtlaenge
        while previous_node.id != -1:           # Solange der Startknoten nicht erreicht ist
            shortest_path.append(previous_node)
            pre_pre_point = previous_node.previous_node.point
            total_length += previous_node.point.euclidean_distance(pre_pre_point)
            if previous_node.previous_node.id == -1:     # Vorheriger Knoten ist Startknoten
                total_length += previous_node.point.euclidean_distance(pre_pre_point)
            previous_node = previous_node.previous_node

        shortest_path.append(previous_node)
        return shortest_path, total_length

# Repraesentiert einen Punkt in Form einer Koordinate oder einen Vektor
class Point:
    def __init__(self, x, y):
        self.x = x      # x-Koordinate
        self.y = y      # y-Koordinate

    # Euklidische Distanz zweier Punkte berechnen (Anwendung des Satz des Pythagoras)
    def euclidean_distance(self, other_point):
        cathetus_a = other_point.x - self.x
        cathetus_b = other_point.y - self.y
        hypotenuse = sqrt(cathetus_a**2 + cathetus_b**2)
        return hypotenuse


# Repraesentiert eine Kante, bzw. eine Strecke
# mit zwei definierten Punkten A und B
class Edge:
    def __init__(self, point_a, point_b):
        self.point_a = point_a
        self.point_b = point_b

    # Erhalte Schnittpunkt mit einer anderen Kante, wobei
    # beide Kanten als Geraden fortgefuehrt werden
    def get_intersection_with(self, other_edge):
        # Als allgm. Geradengleichung gilt: y=mx+b.
```

```
        # Hier: Zwei Geraden mit jeweils m1 bzwn. m2 und b1 bzw. b2
        # Es gilt: m1 = dy1/dx1 und m2 = dy2/dx2
        dy1 = self.point_b.y - self.point_a.y
        dy2 = other_edge.point_b.y - other_edge.point_a.y
        dx1 = float(self.point_b.x - self.point_a.x)
        dx2 = float(other_edge.point_b.x - other_edge.point_a.x)

        # Wenn Kante a parallel zur y-Achse ist
        if dx1 == 0.0:
            intersection_x = self.point_a.x
            m2 = dy2 / dx2
            b2 = other_edge.point_a.y - m2 * other_edge.point_a.x
            intersection_y = m2 * intersection_x + b2
            return Point(intersection_x, intersection_y)

        # Steigung m1 kann berechnet werden, da dx1 ungleich 0 ist
        m1 = dy1 / dx1
        b1 = self.point_a.y - m1 * self.point_a.x

        # Wenn Kante b parallel zur y-Achse ist
        if dx2 == 0.0:
            intersection_x = other_edge.point_a.x
            intersection_y = m1 * intersection_x + b1
            return Point(intersection_x, intersection_y)

        # Steigung m2 kann berechnet werden, da dx2 ungleich 0 ist
        m2 = dy2 / dx2
        b2 = other_edge.point_a.y - m2 * other_edge.point_a.x
        intersection_x = (b2-b1)/float(m1-m2)
        intersection_y = m1 * intersection_x + b1

        return Point(intersection_x, intersection_y)

    # Erhalte den Faktor mit dem die Kante erweitert werden muss,
    # um auf einen bestimmten Punkt zu treffen
    def get_ratio(self, point):
        # Zu pruefende Kante ist parallel zur Y-Achse --> Verhaeltnis ueber y-Werte
        if self.point_b.x == self.point_a.x:
            ratio = (point.y - self.point_a.y) / float(self.point_b.y - self.point_a.y)
        # Sonst: Verhaeltnis ueber x-Werte
        else:
            ratio = (point.x - self.point_a.x) / float(self.point_b.x - self.point_a.x)

        return ratio


# Repraesentiert einen Knoten im Sichtbarkeitsgraphen
class Node:
    def __init__(self, point, id):
        self.point = point  # Jeder Knoten hat einen Punkt
        self.id = id        # Und eine ID, fuer Polygone gilt: ID > 0,
                            # Startknoten hat die ID -1, der Endknoten -2

        # Attribute, die fuer den A-Star-Algorithmus relevant sind
        self.g_cost = 0
        self.h_cost = 0
        self.f_cost = 0
        self.previous_node = 0
```

```python

    # Methode zum Zuweisen der Heuristik (hier: euklidische Distanz)
    def assign_h_cost(self, end_node):
        self.h_cost = self.point.euclidean_distance(end_node.point)
        return self

    # Prueft, ob aktueller Knoten A einen anderen Knoten B sehen kann
    def can_see(self, node_b):
        if node_b.id != -2:
            # Zugehoeriges Polygon des ersten Knoten
            polygon_of_first_node = Input.polygon_list[self.id-1]
        # Pruefen ob beide Knoten zu einem Polygon gehoeren
        if self.id == node_b.id:
            for edge in polygon_of_first_node.convert_to_edges():
                # Wenn Knoten sich eine Kante teilen, ...
                current_points = [edge.point_a, edge.point_b]
                if self.point in current_points and node_b.point in current_points:
                    # ... dann sehen sie sich in jedem Fall
                    return True

        # Richtungsvektor der Verbindung zwischen Knoten a und b
        vector_connection = Point(node_b.point.x - self.point.x,
                                  node_b.point.y - self.point.y)

        # Kanten, die die Verbindung a zu b lediglich beruehren
        potential_obstacles = []

        # Pruefen, ob eine aller Kanten den Weg von node_a zu node_b verdeckt
        for edge in Input.list_of_all_edges:
            # Richtungsvektor der zu pruefenden Kante
            vector_edge = Point(edge.point_b.x - edge.point_a.x,
                                edge.point_b.y - edge.point_a.y)
            # Pruefen, ob Verbindung a zu b und aktuelle Kante parallel sind
            if vector_connection.x*vector_edge.y == vector_connection.y * vector_edge.x:
                continue    # Kante kann Verbindung nicht schneiden

            # Falls Kante ausserhalb des von a und b aufgespannten Rechtecks ist
            if edge.is_out_of_area(self.point, node_b.point):
                continue    # Kante kann Verbindung nicht schneiden

            intersection_point = edge.get_intersection_with(Edge(self.point, node_b.point))
            ratio_edge = round(edge.get_ratio(intersection_point), 6)
            connection_edge = Edge(self.point, node_b.point)
            ratio_connection = round(connection_edge.get_ratio(intersection_point), 6)

            # Kante verdeckt beide Knoten
            if 0 < ratio_edge < 1 and 0 < ratio_connection < 1:
                return False

            # Verbindung koennte durch Eckpunkt eines Polygons gehen,
            # Kanten beruehren die Verbindung a zu b
            elif (ratio_edge == 1 or ratio_edge == 0) and 0 < ratio_connection < 1:
                potential_obstacles.append(edge)

        # Knoten teilt Punkt mit anderen Knoten
        if node_b.point in Input.double_corners or \
                self.point in Input.double_corners:
            return False    # Doppelte Eckpunkte werden ausgelassen
```

```python

        # Pruefen, ob Verbindung im Polygon liegt bei Knoten gleicher Polygone
        if self.id == node_b.id:
            # OA + AB/2 = OP, OP ist Ortsvektor des zu pruefenden Punktes
            local_vector_checkpoint = Point(self.point.x + vector_connection.x/2,
                                            self.point.y + vector_connection.y/2)
            # Beide Knoten koennen sich nicht sehen, da die Verbindung im Polygon liegt
            if polygon_of_first_node.contains(local_vector_checkpoint):
                return False
            else:
                return True

        # Pruefen, ob Verbindung durch eine Ecke geht, ohne Kanten zu schneiden
        # Dabei muss es benachbarte Kanten geben, die weder Knoten A noch B enthalten
        if len(potential_obstacles) > 1:
            for i in range(len(potential_obstacles)):
                for j in range(i+1, len(potential_obstacles)):
                    points_in_edges = [potential_obstacles[i].point_a,
                                       potential_obstacles[i].point_b,
                                       potential_obstacles[j].point_a,
                                       potential_obstacles[j].point_b]
                    # Kanten haben einen gemeinsamen Punkt, wenn es Duplikate in der
                    # Liste (bestehend aus jeweils beiden Punkten beider Kanten) gibt
                    if len(points_in_edges) != len(set(points_in_edges)):
                        # Wenn Knoten A und B nicht auf den Kanten liegen
                        if self.point not in points_in_edges \
                           and node_b.point not in points_in_edges:
                            return False
        return True

    # Prueft, ob Eckpunkt 2 im B-Bereich von Eckpunkt 1 liegt,
    # wodurch die Verbindung von E2 zu E1 redundant waere
    def is_redundant_to(self, node_2):

        # Polygon des Eckpunkts 1
        polygon = Input.polygon_list[self.id-1]

        # Benachbarte Ecken von Eckpunkt 1
        adjacent_points = polygon.get_adjacent_points(self.point)

        # Richtungsvektoren der beiden fortgefuehrten Kanten an Eckpunkt 1
        r1 = Point(2*self.point.x-adjacent_points[0].x, 2*self.point.y-adjacent_points[0].y)
        r2 = Point(2*self.point.x-adjacent_points[1].x, 2*self.point.y-adjacent_points[1].y)

        # Richtungsvektor vom Eckpunkt 1 zum Eckpunkt 2
        vector_connection = Point(node_2.point.x - self.point.x, node_2.point.y - self.point.y)

        # Determinante der Orientierungs-Matrix berechnen
        def det(p, q, r):
            return (q.x * r.y + p.x * q.y + p.y * r.x) - (p.y * q.x + q.y * r.x + p.x * r.y)

        r1_point = Point(r1.x + vector_connection.x, r1.y + vector_connection.y)
        r2_point = Point(r2.x + vector_connection.x, r2.y + vector_connection.y)
        determinant_r1 = det(self.point, r1_point , node_2.point)
        determinant_r2 = det(self.point, r2_point, node_2.point)

        # Falls Eckpunkt 2 zwischen den beiden fortgefuehrten Kanten von Eckpunkt 1 liegt
        # Wenn also die Determinante der Orientierungsmatrix
```

```
        # einmal negativ und einmal positiv ist
        if determinant_r1 < 0 < determinant_r2 or determinant_r2 < 0 < determinant_r1:
            return True

        return False


# Repraesentiert ein Polygon mit seiner ID und seinen Eckpunkten
class Polygon:
    def __init__(self, points, id):
        self.points = points
        self.id = id

    # Prueft, ob ein Punkt im Polygon befindet mithilfe der Strahl-Methode
    def contains(self, point):
        polygon_edges = self.convert_to_edges()
        count_to_left_of_point = 0
        for edge in polygon_edges:
            if edge.point_a.y <= point.y <= edge.point_b.y \
            or edge.point_b.y <= point.y <= edge.point_a.y:
                # Punkt ist auf einer Y-Ebene mit der aktuellen Kante des Polygons
                if edge.point_a.y == edge.point_b.y == point.y:
                    # Kante liegt auf Strahl --> Wird aber als darueber gezaehlt
                    continue
                ray_as_edge = Edge(Point(0, point.y), Point(1, point.y))
                intersection_point = edge.get_intersection_with(ray_as_edge)
                if intersection_point in [edge.point_a, edge.point_b]:
                    # Strahl geht durch Eckpunkt der Kante,
                    # Punkte auf dem Strahl werden als Punkte darueber gezaehlt
                    if edge.point_a.y == point.y:
                        # Punkt A der Kante liegt auf dem Strahl
                        if edge.point_b.y > point.y:
                            # Kante schneidet nicht Strahl,
                            # da Punkt B und Punkt A "ueber" dem Strahl
                            continue
                    else:
                        # Punkt B liegt auf dem Strahl
                        if edge.point_a.y > point.y:
                            # Kante schneidet nicht Strahl,
                            # da Punkt B und Punkt A "ueber" dem Strahl
                            continue

                if intersection_point.x < point.x:
                    # Schnittpunkt der Kante links vom zu pruefenden Punkt
                    count_to_left_of_point += 1

        # Wenn Anzahl der Schnittpunkte links vom Punkt ungerade ist
        # --> Punkt liegt im Polygon
        if count_to_left_of_point % 2 != 0:
            return True
        else:
            return False

    # Gibt nur konvexe Punkte eines Polygons zurueck
    def convex_vertices_only(self):
        points = self.points
        # Finde oberste linke Ecke, da diese in jedem Fall konvex ist
        top_left_corner_index = 0
```

```
        for i in range(1, len(points)):
            if points[i].x <= points[top_left_corner_index].x:
                if points[i].y >= points[top_left_corner_index].y:
                    top_left_corner_index = i

        # neue Liste der Eckpunkte mit der obersten linken Ecke als Startwert
        new_points_list = points[top_left_corner_index:]
        new_points_list.extend(points[0:top_left_corner_index])
        points = new_points_list

        # Alle Ecken, bei denen die Kante rechts von der
        # vorherigen Kante angelegt wird, sind in list_right
        list_right = []
        # Die restlichen Ecken kommen in list_left
        list_left = []

        # Determinante der Orientierungs-Matrix berechnen
        def det(p, q, r):
            return (q.x * r.y + p.x * q.y + p.y * r.x) - (p.y * q.x + q.y * r.x + p.x * r.y)

        # Richtungen fuer jede Ecke bestimmen mithilfe von drei Punkten,
        # der jeweils anliegenden zwei Kanten
        for i in range(len(points)):
            if i == 0:
                point_p = points[len(points)-1]
            else:
                point_p = points[i-1]

            point_q = points[i]

            if i == len(points)-1:
                point_r = points[0]
            else:
                point_r = points[i+1]

            determinant = det(point_p, point_q, point_r)

            # Orientierung ist im Uhrzeigersinn --> Rechtsrum
            if determinant < 0:
                list_right.append(point_q)
            # Orientierung gegen den Uhrzeigersinn --> Linksrum
            else:
                list_left.append(point_q)

        if points[0] in list_right:
            # Ecken, deren Kanten einen Rechts-Turn machen sind konvex

            return list_right
        else:
            # Ecken, deren Kanten einen Links-Turn machen sind konvex
            return list_left
```