Aufgabe 1
„Lisa rennt"- Dokumentation

37. Bundeswettbewerb Informatik 2018/19 - 2. Runde

Lukas Rost

Teilnahme-ID: 48125

29. April 2019

Inhaltsverzeichnis

1 Lösungsidee

1.1 Das Geometric-Shortest-Path-Problem

Das der Aufgabe zugrundeliegende Problem nennt sich *Geometric Shortest Path* (GSP) oder auch *Euclidean Shortest Path* und stammt aus dem Bereich des *Motion Planning*. Bei diesem Problem ist ein punktförmiger Roboter (oder auch eine Schülerin namens Lisa) gegeben, der/die sich an einer Startposition p_{start} in einem kartesischen Koordinatensystem befindet. In diesem Koordinatensystem befinden sich mehrere als Polygone modellierte Hindernisse, wobei jedes einzelne Polygon durch seine Eckpunkte gegeben ist.[1] Weiterhin ist eine Position p_{ziel} gegeben. Nun soll ein möglichst kurzer Weg von p_{start} zu p_{ziel} gefunden werden, wobei dieser nicht durch Hindernisse führen soll.[2][2]

Das hier gegebene Problem unterscheidet sich von GSP dadurch, dass keine Position p_{ziel}, sondern ein Strahl s_{ziel} (in Form eines beliebigen Punktes auf dem Strahl) erreicht werden soll. In diesem Fall handelt es sich dabei um die y-Achse ($x = 0$ mit $y \geq 0$). Zusätzlich soll nicht unbedingt der Weg optimiert werden, sondern die Startzeit, die abhängig von der Länge des Weges und der für jeden Punkt des Strahls unterschiedlichen Zielzeit ist. Diese Zielzeit kann mithilfe des Abstands des Punktes vom Ursprung berechnet werden.

Das Geometric-Shortest-Path-Problem wird im Allgemeinen in zwei Schritten gelöst: Zunächst wird ein Sichtbarkeitsgraph erstellt, auf welchem dann Dijkstras Algorithmus ausgeführt wird.[3] Zur Lösung des hier gegebenen Problems wird jedoch ein zusätzlicher Schritt benötigt, der auf der Lösung des Problems ohne Hindernisse basiert. Diese drei Algorithmen sollen in den folgenden Abschnitten vorgestellt und näher erläutert werden.

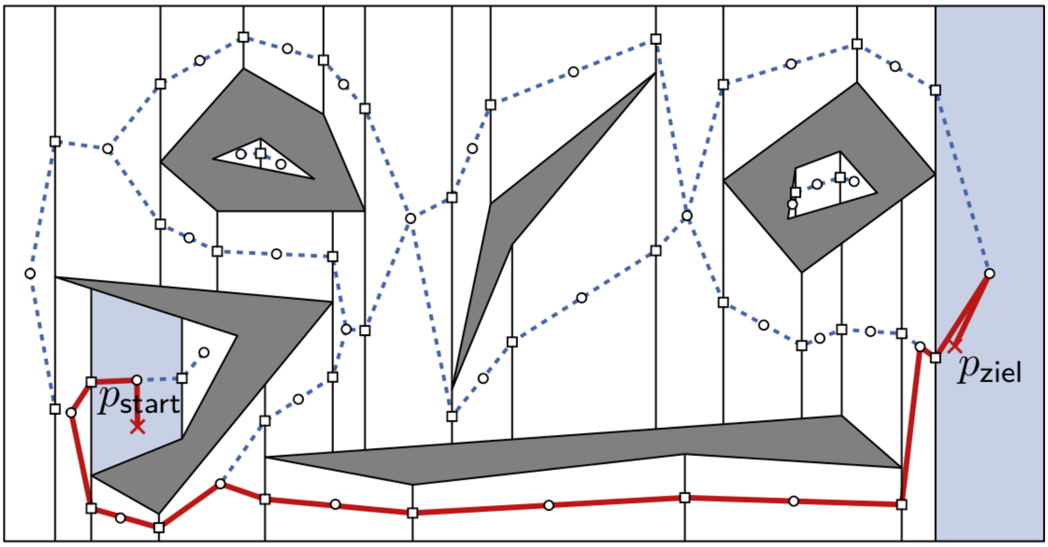


Abbildung 1: Illustration des GSP-Problems (aus [2])

[1] In dieser Dokumentation wird angenommen, dass die Punkte entgegen dem Uhrzeigersinn sortiert sind, sonst muss dies vorher geschehen.

[2] Zumindest nicht, wenn Lisa als Zielposition nicht das Krankenhaus erreichen will.

[3] Es existieren natürlich auch andere Herangehensweisen wie eine Grid- oder Intervall-basierte Suche, die das Problem jedoch nicht exakt lösen können und deshalb hier nicht geeignet sind.

1.2 Erzeugung eines Sichtbarkeitsgraphen

Es lässt sich beobachten, dass der kürzeste s-t-Weg (dabei sei s die Startposition und t die Zielposition) in einem solchen Koordinatensystem ein Polygonzug sein muss, dessen innere Knoten Knoten (bzw. Ecken) der Hindernisse sind. Andernfalls wäre es immer möglich, einen kürzeren Weg zu konstruieren, der über einen Hindernisknoten führt.[2] Ausgehend davon lässt sich ein sogenannter Sichtbarkeitsgraph erzeugen, bei dem es sich um einen gewichteten, ungerichteten Graphen handelt.

Für eine Polygonmenge S mit Knotenmenge $V(S)$ sei dieser definiert als $G_{vis}(S) = (V(S), E_{vis}(S))$. Dabei ist $E_{vis}(S)$ die Menge der Knotenpaare von S, sodass die dazwischenliegende Strecke kein Polygon (bzw. das Innere eines Polygons) schneidet. Mathematisch ausgedrückt bedeutet das $E_{vis}(S) = \{uv | u, v \in V(S) \text{ und } \overline{uv} \subset C_{free} = \mathbb{R}^2 \setminus \bigcup S\}$. Das Gewicht einer Kante sei dabei als euklidischer Abstand der beiden Endpunkte definiert.

Wenn man S^* als $S \cup \{s\}$ definiert (bei normalen Sichtbarkeitsgraphen wird auch t hinzugefügt) und den Sichtbarkeitsgraphen dafür analog, kann man damit das GSP-Problem lösen. Nun entspricht der kürzeste s-t-Weg, der Hindernisse vermeidet, einem kürzesten Weg in $G_{vis}(S^*)$.

1.2.1 Naiver Ansatz

$G_{vis}(S^*)$ lässt sich nun naiv in $\mathcal{O}(n^3)$ berechnen, wobei $n = |V(S^*)|$. Dazu bestimmt man für jeden Knoten $u \in V(S^*)$ die von ihm sichtbaren Knoten v. Dabei muss man für jede Strecke $\overline{uv}$ prüfen, ob sie eine der $|E_{vis}(S^*)| = \mathcal{O}(n)$ in Frage kommenden Hinderniskanten schneidet. Die Bestimmung der von u sichtbaren Knoten ist somit in $\mathcal{O}(n^2)$ durchführbar und insgesamt ergibt sich eine Laufzeit von $O(n^3)$.

Die Funktionsweise des naiven Algorithmus wird auch in folgendem Pseudocode deutlich:

Algorithmus 1 Naiver Algorithmus

```
function VISIBILITYGRAPH(S)
    G = (V(S), E) ← leerer Sichtbarkeitsgraph
    for all Knoten v ∈ V(S) do                          ▷ O(n)
        for all Knoten w ∈ V(S) \ {v} do                ▷ O(n)
            for all Kante e ∈ E_vis(S) do               ▷ O(n)
                if vw schneidet keine der Kanten e then
                    E ← E ∪ {vw}
                end if
            end for
        end for
    end for
    return G
end function
```

1.2.2 Der Lee-Algorithmus

Es ist jedoch auch möglich, die Bestimmung der von u sichtbaren Knoten in $\mathcal{O}(n \cdot \log n)$ durchzuführen, wodurch sich insgesamt eine Laufzeit von $\mathcal{O}(n^2 \cdot \log n)$ ergibt. Der dazu notwendige Algorithmus ist der Algorithmus von D. T. Lee. Durch dessen geringere Laufzeit lässt sich insbesondere bei großen Eingaben eine deutliche Verbesserung erreichen.

Es existieren zwar noch schnellere Algorithmen wie der nach Overmars und Welzl in $\mathcal{O}(n^2)$ oder der nach Ghosh und Mount in $\mathcal{O}(n \cdot \log n + E)$[5]. Doch die damit erreichten Verbesserungen sind nur in speziellen Fällen wirklich bemerkbar, während die Implementierung deutlich schwieriger ist.

Lees Algorithmus ist grundlegend ähnlich aufgebaut, muss jedoch für jede Strecke $\overline{uv}$ nur noch eine Hinderniskante prüfen, welche in $\mathcal{O}(\log n)$ bestimmt werden kann. Dies wird auch in folgendem Pseudocode deutlich:

Algorithmus 2 Der Algorithmus von Lee

```
function VISIBILITYGRAPH(S)
    G = (V(S), E) ← leerer Sichtbarkeitsgraph
    for all Knoten v ∈ V(S) do                           ▷ O(n)
        W ← VISIBLE_VERTICES(v,S)                        ▷ O(n · log n)
        E ← E ∪ {vw | w ∈ W}
    end for
end function
```

Der Algorithmus in der Methode `visible_vertices` benutzt dabei eine sogenannte rotierende *Sweep line* beziehungsweise *Scan line*. Diese Sweep line fegt („sweept“) den zweidimensionalen Raum aus, wobei sie durch den gesamten Raum bewegt wird, bis alle Objekte (in diesem Fall die Knoten) besucht und verarbeitet wurden. Im Falle von Lees Algorithmus rotiert die Sweep line (mathematisch gesehen ein Strahl) um den Startpunkt v gegen den Uhrzeigersinn.

Am Anfang zeigt die Sweep line nach rechts (parallel zur x-Achse) und rotiert so lange gegen den Uhrzeigersinn, bis sie einen Punkt trifft, der auf Sichtbarkeit überprüft werden muss. Dazu wird eine Liste der offenen Kanten geführt, mit denen sich die Sweep line aktuell überhaupt schneiden kann und die damit relevant sind.[4] Anfangs werden dabei einmal alle Kanten daraufhin überprüft, ob sie sich mit der Sweep line (in der Anfangsstellung) schneiden. Entsprechend wird dann diese Liste initialisiert.

Trifft die Sweep line auf einen Punkt a, werden deshalb alle Kanten überprüft, deren Endpunkt a ist. Liegt eine Kante bezüglich der Sweep line im Uhrzeigersinn (*clockwise side*), kann sie, sofern sie enthalten ist, aus der Liste der offenen Kanten entfernt werden, da die Sweep line bei Bewegung gegen den Uhrzeigersinn nie wieder mit ihr in Berührung kommt. Liegt die Kante dagegen entgegen des Uhrzeigersinns (*counter-clockwise side*), so muss sie zu den offenen Kanten hinzugefügt werden, da sich die Sweep line im Folgenden mit ihr schneiden kann.

So werden in diesem Beispiel (siehe nächste Seite) am Punkt a die Kanten $\overline{ab}$ und $\overline{ac}$ hinzugefügt, da sie bezüglich der Sweep line gegen den Uhrzeigersinn (auf ihrer linken Seite) liegen. In Punkt b kann $\overline{ab}$ dagegen wieder entfernt werden, da sie rechts der Sweep line liegt. Hier wird dann jedoch $\overline{bc}$ hinzugefügt.

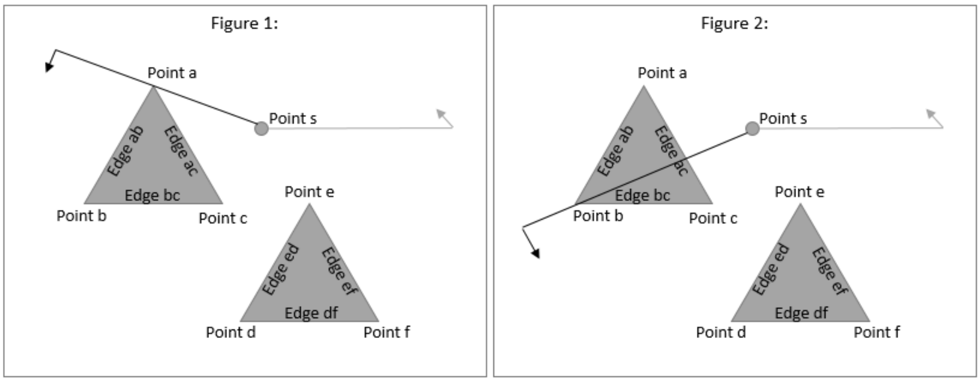


Abbildung 2: Illustration des Konzepts der *Sweep line* (aus [4])

Es lässt sich zeigen, dass sogar nur diejenige offene Kante geprüft werden muss, welche am nächsten an v liegt, d.h. diejenige, für die der Schnittpunkt mit der Sweep line am nächsten an v liegt. Um diese Kante schnell zu bestimmen, muss man die Kanten nach der Distanz zum Schnittpunkt ordnen. Dies lässt sich zum Beispiel mit einem balancierten binären Suchbaum wie einem *AVL-Baum* erreichen.[4][4] Die `visible_vertices`-Funktion sieht nun also bei einer Formulierung als Pseudocode ungefähr so aus:

Algorithmus 3 Bestimmung der sichtbaren Knoten bzw. Punkte im Lee-Algorithmus

function VISIBLE_VERTICES(v, S)
  $w \leftarrow sort(V(S) \setminus \{v\})$ ▷ Sortieren der Knoten (siehe unten)
  $s \leftarrow v + \lambda \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ mit $\lambda \in \mathbb{R}^+$ ▷ Sweep line initialisieren
  $T \leftarrow binarySearchTree()$
  $W \leftarrow \emptyset$
  for all Kante $e \in E(S)$ **do**
    if $e \cap s \neq \emptyset$ **then** ▷ alle Kanten, die die Sweep line schneiden
      $T \leftarrow T \cup e$
    end if
  end for
  for all Knoten $w_i \in w$ **do** ▷ in sortierter Reihenfolge
    Rotiere s so, dass er w_i schneidet
    for all Kante e mit Endpunkt w_i **do**
      if e rechts von s **then**
        $T \leftarrow T \setminus \{e\}$ ▷ Kante entfernen
      end if
    end for
    if VISIBLE(w_i) **then** ▷ ist sichtbar
      $W \leftarrow W \cup w_i$
    end if
    for all Kante e mit Endpunkt w_i **do**
      if e links von s **then**
        $T \leftarrow T \cup \{e\}$ ▷ Kante hinzufügen
      end if
    end for
  end for
  return W
end function

[4]Die Funktionsweise eines AVL-Baums sei hier als bekannt vorausgesetzt.

Das Sortieren der Knoten erfolgt dabei anhand ihrer Polarkoordinaten relativ zu v. Dabei wird zuerst nach kleinerem Winkel zu s und dann, falls dieser gleich sein sollte, nach kleinerem Abstand zu v sortiert. Die Formulierung *links* im Pseudocode bedeutet entgegengesetzt dem Uhrzeigersinn und *rechts* bedeutet im Uhrzeigersinn.

Algorithmus 4 Sichtbarkeitsprüfung im Lee-Algorithmus

```
function VISIBLE(w_i)
    if vw_i schneidet das Innere des Polygons, dessen Knoten w_i ist then
        return false
    end if
    if w_{i−1} nicht auf der Geraden durch vw_i then
        if vw_i ∩ smallest(T) = ∅ then
            return true
        else
            return false
        end if
    else if not VISIBLE(w_{i−1}) then
        return false
    else
        Überprüfe alle Kanten in T auf Schnitt mit vw_i und ...
        return false
        .. falls Schnitt mit einer der Kanten
    end if
    return true
end function
```

Diese Funktion `visible` prüft dabei ganz einfach die Sichtbarkeit eines Knotens. $smallest(T)$ gibt die kleinste Kante im Suchbaum an, sofern diese existiert.

1.3 Der Dijkstra-Algorithmus

Um nun im Sichtbarkeitsgraphen einen kürzesten Pfad vom Startknoten s zu allen anderen Knoten zu bestimmen, was für die Lösung des Problems notwendig ist, kann man Dijkstras Algorithmus verwenden. Dieser arbeitet nach dem Greedy-Prinzip, bei dem in jedem Schritt ein optimaler Folgezustand gewählt wird, der das aktuell beste Ergebnis verspricht. Er arbeitet grundlegend wie folgt:

1. Weise allen Knoten die beiden Eigenschaften „Distanz“ und „Vorgänger“ zu. Initialisiere die Distanz im Startknoten s mit 0 und in allen anderen Knoten mit ∞.
2. Solange es noch unbesuchte Knoten gibt, wähle darunter denjenigen mit minimaler Distanz aus und
   a) speichere, dass dieser Knoten schon besucht wurde.
   b) Berechne für alle noch unbesuchten Nachbarknoten die Summe des jeweiligen Kantengewichtes und der Distanz im aktuellen Knoten.
   c) Ist dieser Wert für einen Knoten kleiner als die dort gespeicherte Distanz, aktualisiere sie und setze den aktuellen Knoten als Vorgänger.

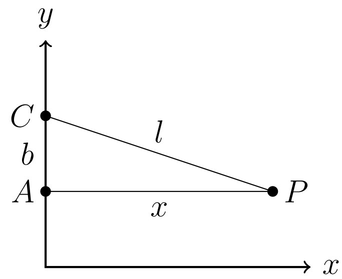
Der Dijkstra-Algorithmus hat eine Laufzeit von $\mathcal{O}((|V| + |E|) \cdot \log |V|)$ (bei Implementierung mit einer Vorrangwarteschlange). Da der Algorithmus den entsprechenden Weg durch das Setzen eines Vorgängers ebenfalls bestimmt, kann auch dieser selbst ausgegeben werden.

1.4 Lösung des Problems ohne Hindernisse

Damit man nun diese spezielle Abwandlung des Problems lösen kann, sollte man zunächst einmal versuchen, das Problem ohne Hindernisse zu lösen, wie dies die Aufgabenstellung auch *sehr* unauffällig nahelegt.

Zunächst lässt sich feststellen, dass es ausschließlich sinnvoll ist, Punkte mit einer y-Koordinate $\geq$ der y-Koordinate von Lisas Haus als Zielpunkte zu wählen. Sonst würde der Abstand zwischen Start- und Zielpunkt größer, während die Abfahrtszeit des Busses früher läge, was insgesamt für Lisa ein früheres Aufstehen bedeuten würde.

Betrachten wir nun die nebenstehende Situation, in der $P(x_p|y_p)$ Lisas Haus darstellt und $C(x_c|y_c)$ den optimalen Zielpunkt mit $y_c \geq y_p$. Die Zeit, die der Bus für die Strecke b benötigt, berechnet sich mit v_b als Geschwindigkeit des Busses zu



$$t_B = \frac{b}{v_B} \tag{1}$$

Die Zeit, welche Lisa für die Strecke l benötigt, berechnet sich äquivalent zu

$$t_L = \frac{l}{v_L} = \frac{\sqrt{x^2+b^2}}{v_L} \tag{2}$$

da das Dreieck rechtwinklig ist. Um nun den optimalen Punkt C zu finden, ist es nötig, den Term $t_L - t_B$ abhängig von der Strecke b zu minimieren. Dies lässt sich einfach mit dem aus der Kurvendiskussion bekannten Ansatz über die erste Ableitung erreichen. Dann ergibt sich:

$$\begin{aligned}\frac{d(t_L - t_B)}{db} &= \frac{d}{db}(\frac{\sqrt{x^2+b^2}}{v_L} - \frac{b}{v_B}) \\ &= \frac{1}{v_L} \cdot \frac{2 \cdot b}{2 \cdot \sqrt{x^2+b^2}} - \frac{1}{v_B} \\ &= \frac{1}{v_L} \cdot \frac{b}{\sqrt{x^2+b^2}} - \frac{1}{v_B}\end{aligned} \tag{3}$$

Setzt man diese Ableitung nun gleich 0, erhält man:

$$\frac{1}{v_L} \cdot \frac{b}{\sqrt{x^2+b^2}} - \frac{1}{v_B} = 0 \tag{4}$$

$$\frac{1}{v_L} \cdot \frac{b}{\sqrt{x^2+b^2}} = \frac{1}{v_B} \tag{5}$$

$$b = \frac{v_L}{v_B} \cdot \sqrt{x^2+b^2} \tag{6}$$

$$b^2 = \frac{v_L^2}{v_B^2} \cdot (x^2+b^2) \tag{7}$$

$$(1 - \frac{v_L^2}{v_B^2}) \cdot b^2 = \frac{v_L^2}{v_B^2} \cdot x^2 \tag{8}$$

$$b^2 = \frac{v_L^2}{v_B^2} \cdot \frac{1}{(1 - \frac{v_L^2}{v_B^2})} \cdot x^2 \tag{9}$$

$$b = \frac{v_L}{v_B} \cdot \frac{1}{\sqrt{1 - \frac{v_L^2}{v_B^2}}} \cdot x \tag{10}$$

Für den optimalen Zielpunkt C gilt somit $x_c = 0$ und $y_c = y_p + \frac{v_L}{v_B} \cdot \frac{1}{\sqrt{1-\frac{v_L^2}{v_B^2}}} \cdot x_p$.[5] Setzt man die in der Aufgabenstellung gegebenen Geschwindigkeiten ein, ergibt sich: $y_c = y_p + \frac{\sqrt{3}}{3} \cdot x_p$

1.5 Kombination der Ansätze

Um das Problem nun lösen zu können, muss man beide Ansätze (mit und ohne Hindernisse) kombinieren. Dazu bestimmt man zu jedem Knoten/Punkt p in der Eingabe einen (von mir so bezeichneten) *Companion*-Punkt c, indem man die im vorigen Abschnitt berechnete Gleichung auf diesen Punkt anwendet. Dieser Punkt c ist dann der optimale Punkt, um von p aus die y-Achse zu erreichen und dabei möglichst spät starten zu müssen.

Da es jedoch sein kann, dass c von p nicht sichtbar ist (ein Hindernis liegt dazwischen), muss man eine Sichtbarkeitsprüfung durchführen. Dazu betrachtet man c einfach bei der Erstellung des Sichtbarkeitsgraphen, genauer bei der Ermittlung der sichtbaren Punkte von p aus (und nur dabei, denn für alle anderen Punkte hat c keine Relevanz).

Nun kann man mithilfe des Dijkstra-Algorithmus' die kürzesten Wege zu allen Companion-Knoten berechnen. Aus diesen und der Lage der Companion-Punkte (aus dieser bestimmt sich die Zeit, an der der Bus dort vorbeifährt) lässt sich dann für jeden Companion-Punkt die spätestmögliche Startzeit am Startpunkt s bestimmen, wenn man an diesem Companion-Punkt auf den Bus treffen will.

Der Companion-Punkt mit der spätesten spätestmöglichen Startzeit ist der Punkt, an dem Lisa auf den Bus treffen sollte, und der Weg zu ihm der optimale Weg.

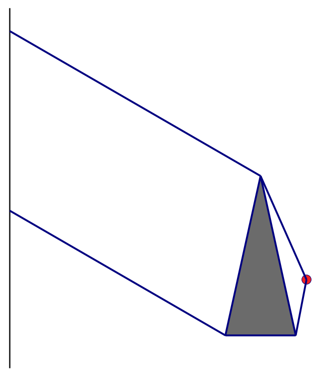


Abbildung 3: Veranschaulichung eines kompletten Sichtbarkeitsgraphen mit Companion-Punkten für Beispiel 1 des BwInf (blaue Linien sind Kanten des Graphen)

Hierbei soll noch kurz informell bewiesen werden, warum von einem Punkt p aus nur der Punkt c optimal sein kann und, wenn keine Sichtbarkeit zwischen diesen beiden Punkten besteht, c einfach ignoriert werden kann. Nehmen wir an, von p aus sei c nicht sichtbar, d.h. der Strahl von p in Richtung c führt durch ein Polygon. Nun können wir den Strahl so nach unten (gegen den Uhrzeigersinn) rotieren, dass er gerade kein Polygon mehr schneidet.

Nun schneidet er jedoch in jedem Fall einen Eckpunkt eines Polygons, hier d genannt. Die Strecke zwischen p und d muss im Sichtbarkeitsgraphen enthalten sein, muss also

[5] Ich habe keine Ahnung, warum das wie der Lorentzfaktor aus der Relativitätstheorie aussieht.

hier nicht erneut betrachtet werden. Von d aus gesehen gibt es einen eigenen Companion-Punkt c_2, der optimal ist. Also muss auch die Teilstrecke zwischen d und dem neuen Schnittpunkt des Strahls mit der y-Achse nicht beachtet werden.

1.6 Laufzeitbetrachtung

In den einzelnen Abschnitten wurde teilweise schon auf die Laufzeiten der Teilalgorithmen eingegangen. Hier soll dies noch einmal zusammengefasst werden.

Algorithmus von Lee. Die Funktion `visible_vertices` im Algorithmus von Lee läuft offensichtlich in $\mathcal{O}(n \cdot \log n)$. Der Grund dafür ist, dass die einzelnen Schritte folgende Laufzeiten haben:

- **Sortieren der Knoten**: kann in $\mathcal{O}(n \cdot \log n)$ erfolgen (mit z.B. Mergesort oder Quicksort), wobei $\Omega(n \cdot \log n)$ wie allgemein bekannt auch die Untergrenze für die Worst-Case-Laufzeit eines Sortieralgorithmus ist.
- **Schnittprüfung von Sweep line und Kanten**: Diese muss anfangs für n Polygonkanten durchgeführt werden, hier ergibt sich also $\mathcal{O}(n)$.
- **Überprüfen aller Knoten**: Dies wird für alle $\mathcal{O}(n)$ Knoten durchgeführt. Dabei benötigt die Sichtbarkeitsprüfung $\mathcal{O}(\log n)$ (durch die Verwendung eines Suchbaumes). Danach müssen noch konstant viele Kanten dem Baum hinzugefügt bzw. entfernt werden.[6] Das Hinzufügen bzw. Entfernen benötigt wieder $\mathcal{O}(\log n)$. Insgesamt ergibt sich also auch hier wieder $\mathcal{O}(n \cdot \log n)$.

Wie eindeutig sichtbar ist, dominiert hier $\mathcal{O}(n \cdot \log n)$. Da die Funktion für alle n Knoten aufgerufen wird, ergibt sich für den Gesamtalgorithmus eine Laufzeit in $\mathcal{O}(n^2 \cdot \log n)$ □

Algorithmus von Dijkstra. Der Dijkstra-Algorithmus hat bei der allgemein übliche Implementierung mit einer Priority Queue eine Laufzeit von $\mathcal{O}((|V|+|E|) \cdot \log |V|)$, wobei V die Knotenmenge und E die Kantenmenge ist. Da in diesem Fall $|V| \in \mathcal{O}(n)$ und $|E| \in \mathcal{O}(n^2)$ gilt, ergibt sich hier eine Laufzeit in $\mathcal{O}(n^2 \cdot \log n)$. □

Kombination der Ansätze. Da bei der Berechnung des Companion-Punktes nur einfache Rechenoperationen durchgeführt werden, kann dieser in konstanter Zeit, also $\mathcal{O}(1)$ berechnet werden. Für die n Punkte bzw. Knoten ergibt sich dann $\mathcal{O}(n)$. Alle weiteren Schritte aus dem Abschnitt *Kombination der Ansätze* sind in die Algorithmen von Dijkstra bzw. Lee integriert, tragen also nicht weiter zur Laufzeit bei. Nur die Berechnung der Startzeit muss für jeden Punkt noch separat erfolgen, hier ergibt sich wieder $\mathcal{O}(n)$. □

Insgesamt ergibt sich für die Laufzeit aller drei Algorithmen also $\mathcal{O}(n^2 \cdot \log n + n^2 \cdot \log n + n) = \mathcal{O}(n^2 \cdot \log n)$.

1.7 Mögliche und implementierte Erweiterungen

1.7.1 Anpassbare Geschwindigkeiten

Es ist relativ einfach möglich, die Geschwindigkeiten, mit denen sich Lisa und der Bus bewegen, zu verändern. Dazu muss einfach in allen Gleichungen, die aus Weg und Geschwindigkeit die benötigte Zeit berechnen, die Geschwindigkeit entsprechend gewählt werden. Da bei der *Lösung des Problems ohne Hindernisse* bereits die Geschwindigkeiten als Parameter verwendet wurden, ist dies auch dort ohne Probleme möglich. Diese Erweiterung läuft in $\mathcal{O}(1)$ und wurde **implementiert** (siehe Umsetzung).

[6] Normalerweise höchstens 2, in Sonderfällen können es aber auch bis zu 4 sein. Diese Sonderfälle treten dann auf, wenn sich zwei Polygone eine Kante oder einen Knoten teilen.

1.7.2 Polygonale Lisa

Nun modellieren wir Lisa nicht mehr als Punkt, sondern als Polygon. Wieso? Möglicherweise handelt es sich bei einem der Hindernispolygone um das örtliche Fastfoodrestaurant und Lisa besucht dieses sehr oft, sodass sie extrem zugenommen hat.

Möglicherweise ist der Grund aber auch unspektakulärer. Eventuell ist Lisa gerade 18 geworden und möchte nun mit ihrem Auto zur Bushaltestelle fahren, auch wenn man über die ökologische Sinnhaftigkeit dieses Vorhabens streiten kann.[7]

Dies lässt sich mathematisch beschreiben, indem man für jedes Hindernispolygon dessen Minkowski-Summe mit einer am Ursprung gespiegelten Kopie des Lisa-Polygons berechnet und auf den so entstandenen Polygonen den ursprünglichen Algorithmus ausführt. Die Minkowski-Summe ist dabei für zwei Teilmengen A und B (in diesem Fall Polygone) eines Vektorraums (des $\mathbb{R}^2$) definiert als:

$$A + B := \{a + b \,|\, a \in A, b \in B\} \tag{11}$$

Sie ist also die Menge aller Elemente, die Summen von je einem Element aus A und einem aus B sind.[9] Dies entspricht im Prinzip dem Entlangfahren des Lisa-Polygons am Rand des Hindernispolygons. Dabei wird das Hindernis so verbreitert, dass Lisa nun wieder als punktförmig angenommen und der ursprüngliche Algorithmus ausgeführt werden kann.

Seien nun das Hindernispolygon $P = (v_1, ..., v_k)$ und die Kopie des Lisa-Polygons $L = (w_1, ..., w_m)$ mit v_i und w_i als Eckpunkten. Für diese zwei Polygone im $\mathbb{R}^2$ kann die Minkowski-Summe (bzw. deren konvexe Hülle) wie folgt berechnet werden:

Zunächst berechnet man die Minkowski-Summe $S = P + L$ nach der oben angegebenen Definition. Anschließend kann man deren konvexe Hülle mit einem Algorithmus wie dem Graham Scan berechnen:

Algorithmus 5 Bestimmung der konvexen Hülle der Minkowski-Summe

```
function grahamScan(S)
    stack = ∅                                                         ▷ leerer Stack
    finde den Punkt p0, der die geringste y-Koordinate hat und am weitesten links ist.
    H = {p0}                                          ▷ leere konvexe Hülle nur mit p0
    Sortiere Punkte in S nach dem Winkel von p0pi zur positiven x-Achse
    for all p in S do
        while |stack| > 1 and ccw(next_to_top(stack),top(stack),point) < 0 do
            stack.pop()
        end while
        stack.push(p)
    end for
end function
```

Dabei gibt die Funktion $ccw(a, b, c)$ an, ob c links von $\overrightarrow{AB}$ ist (Rückgabe ist positiv) oder rechts von $\overrightarrow{AB}$ (Rückgabe ist negativ). Der Graham-Scan-Algorithmus läuft für n Punkte in $\mathcal{O}(n \cdot \log n)$. Da die Minkowski-Summe nach Definition $\mathcal{O}(k \cdot m)$ Punkte hat, ergibt sich in diesem Fall $\mathcal{O}(km \cdot \log km)$. Für alle Polygone zusammen ergibt sich eine Laufzeit von $\mathcal{O}(k_{max}m \cdot \log k_{max}m)$ mit k_{max} als maximaler Eckpunktanzahl. Da $k_{max} \in \mathcal{O}(n)$ mit n als Anzahl aller Eckpunkte, hat man eine Laufzeit von $\mathcal{O}(nm \cdot \log nm)$. Diese Erweiterung wurde **implementiert**.

[7] Hier Fridays-for-Future-Demonstration einfügen.

1.7.3 Nicht-polygonale Hindernisse

Es wäre auch möglich, die Art der Hindernisse auf Flächen, die keine Polygone sind, zu erweitern. Dies könnten beispielsweise Ovale oder Kreise sein. Eine Möglichkeit, den Sichtbarkeitsgraph darauf zu erweitern, ist in [8] beschrieben. Dabei muss man anstelle des klassischen Sichtbarkeitsgraphen einen Tangenten-Sichtbarkeitsgraphen verwenden. Da dies jedoch relativ kompliziert ist, wird an dieser Stelle nicht genauer darauf eingegangen. Außerdem lässt sich eine gute Näherung dafür erreichen, wenn man anstelle eines Kreises ein Polygon mit sehr vielen Ecken verwendet (siehe Beispiele).

Diese Erweiterung wurde **nicht implementiert**.

1.7.4 Das Problem im $\mathbb{R}^3$

Nehmen wir nun an, dass Lisa fliegen kann. Wie sie diese Fähigkeit erlangt hat, ist unwichtig. Möglicherweise ist sie mit Quax aus der letztjährigen 3. Aufgabe der 2. Runde befreundet und kann seinen Quadrocopter benutzen. Außerdem nehmen wir an, dass der Bus unendlich hoch ist. Inwiefern das sinnvoll ist, soll hier ebenfalls nicht betrachtet werden.

Dazu kann man das Koordinatensystem von einem $\mathbb{R}^2$ auf einen $\mathbb{R}^3$ erweitern, sodass ein dreidimensionales Koordinatensystem entsteht. In diesem sind die Hindernisse dann Körper und die Zielgerade eine Zielebene, genauer die yz-Ebene. Ein Ansatz dazu ist in [6] beschrieben.

Ein solcher Sichtbarkeitsgraph ist dann jedoch ein Hypergraph, bei dem jede Kante 3 Knoten verbindet. In jedem Fall ist dieses Problem jedoch NP-schwer, kann also vermutlich nicht effizient gelöst werden.[10] Aus diesem Grund wurde die Erweiterung **nicht implementiert**.

Literatur

[1] Bittel, O. (HTWG Konstanz): Autonome Roboter - Wegekartenverfahren, SS 2016 (Präsentation), `http://www-home.htwg-konstanz.de/~bittel/msi_robo/Vorlesung/06_Planung_Wegekarten.pdf`

[2] Nöllenburg, Martin (KIT): Vorlesung Algorithmische Geometrie - Sichtbarkeitsgraphen, 2011 (Präsentation), `https://i11www.iti.kit.edu/_media/teaching/sommer2011/compgeom/algogeom-ss11-vl14-printer.pdf`

[3] Reksten-Monsen, Christian: Distance Tables Part 1 - Defining the Problem, `https://taipanrex.github.io/2016/09/17/Distance-Tables-Part-1-Defining-the-Problem.html`

[4] Reksten-Monsen, Christian: Distance Tables Part 2 - Lee's Visibility Graph Algorithm, `https://taipanrex.github.io/2016/10/19/Distance-Tables-Part-2-Lees-Visibility-Graph-Algorithm.html`

[5] Kitzinger, John (University of New Mexico): The Visibility Graph Among Polygonal Obstacles: A Comparison of Algorithms, 2003, `https://www.cs.unm.edu/~moore/tr/03-05/Kitzingerthesis.pdf`

[6] Bygi, Mojtaba Nouri; Ghodsi, Mohammad (Sharif University of Technology): 3D Visibility Graph, `https://pdfs.semanticscholar.org/aba1/5853197c24ed6f164e4fb5e2f134462c7ebf.pdf`

[7] Coleman, Dave (University of Colorado): Lee's Visibility Graph Algorithm - Implementation and Analysis, 2012, `https://github.com/davetcoleman/visibility_graph/blob/master/Visibility_Graph_Algorithm.pdf`

[8] Hutchinson, Joan P. (Macalester College): Arc- and circle-visibility graphs, `https://pdfs.semanticscholar.org/d257/d8f5ea2f9bb32c555b4d5723fdcf1e97dc4f.pdf`

[9] Skript der Uni Freiburg zur Minkowski-Summe, `http://algo.informatik.uni-freiburg.de/bibliothek/books/ad-buch/k7/slides/08.pdf`

[10] Wikipedia-Artikel zur NP-Schwere des dreidimensionalen GSP, `https://en.wikipedia.org/wiki/Euclidean_shortest_path`

2 Umsetzung

2.1 Allgemeine Hinweise zur Benutzung

Das Programm wurde in der Programmiersprache Python 3.7 implementiert und unter Linux getestet. Zur Verwaltung und Erstellung des Sichtbarkeitsgraphen wurde die Bibliothek `pyvisgraph` von Christian Reksten-Monsen benutzt[8]. Dabei kommt jedoch eine von mir erheblich veränderte Version zum Einsatz, die an das hier zu lösende Problem angepasst ist. Diese Version ist die einzige mit dem Programm kompatible und ist in die Einsendung integriert.

Diese Bibliothek benötigt wiederum die Bibliothek tqdm (`sudo pip3 install tqdm`). Alle weiteren verwendeten Bibliotheken sind üblicherweise vorinstalliert.

Die Eingabe und Ausgabe des Programms erfolgt in Dateien, die mithilfe der Konsolenparameter frei gewählt werden können. Die weitere Bedienung sollte selbsterklärend sein. Es gibt folgende Konsolenparameter:

```
usage: main.py [-h] [-i INPUT] [-o OUTPUT] [-so SVG] [-d]
[-vlisa VELOCITY_LISA] [-vbus VELOCITY_BUS]
[-minkowski MINKOWSKI]

Lösung zu Lisa rennt, Aufgabe 1, Runde 2, 37. BwInf von Lukas Rost

optional arguments:
-h, --help            show this help message and exit
-i INPUT              Eingabedatei
-o OUTPUT             Ausgabedatei
-so SVG               SVG-Ausgabedatei
-d                    Debug-Ausgaben aktivieren
-vlisa VELOCITY_LISA  Erweiterung Geschwindigkeiten: Lisa in km/h
-vbus VELOCITY_BUS    Erweiterung Geschwindigkeiten: Bus in km/h
-minkowski MINKOWSKI  Erweiterung Minkowski-Summe: Eingabedatei (1 Polygon
im gleichen Format wie in der normalen Eingabe)
```

[8] `https://github.com/TaipanRex/pyvisgraph/`

Die Eingabedatei für die Minkowski-Erweiterung ist im Format der anderen Eingaben gestaltet. Durch Leerzeichen getrennt wird zunächst die Anzahl der Eckpunkte und anschließend für jeden Eckpunkt x- und y-Koordinate angegeben. Dabei stehen alle Eingaben in einer Zeile (Beispiel: `4 0 0 0 1 1 1 1 0`).

2.2 Struktur des Programms und Implementierung der Algorithmen

Von den im folgenden vorgestellten Dateien waren `visible_vertices.py`, `vis_graph.py`, `shortest_path.py` und `graph.py` ursprünglich Teil der `pyvisgraph`-Library. Diese wurden jedoch von mir für den Einsatz in diesem Programm abgeändert.

In der Dokumentation sind nur Teile des Quellcodes aus `vis_graph.py`, `shortest_path.py`, `visible_vertices.py`, `minkowski.py` und `main.py` abgedruckt. Alle weiteren Dateien sind algorithmisch uninteressant und daher nur in der Implementierung enthalten.

2.2.1 Die Datei `main.py`

Die Datei `main.py` bildet den Startpunkt des Programms und wird beim Start aufgerufen. Sie liest die übergebenen Konsolenargumente mithilfe der `argparse`-Library ein. Daraufhin werden die entsprechenden Eingabedaten in der angegebenen Eingabedatei eingelesen. Dann wird der Algorithmus ausgeführt (dazu wird ein Sichtbarkeitsgraph aus der Datei `vis_graph.py` erstellt und in ihm der kürzeste Pfad berechnet) und seine Ausgaben werden in den entsprechenden Ausgabedateien gespeichert.

2.2.2 Die Datei `svggen.py`

In dieser Datei existieren nur die zwei Funktionen `gen_vis_svg` und `gen_output_svg`, die jeweils eine SVG-Datei für die graphische Ausgabe des Sichtbarkeitsgraphs bzw. der endgültigen Route erzeugen.

2.2.3 Die Datei `graph.py`

Diese Datei enthält drei im weiteren Verlauf des Algorithmus häufig benötigte Klassen. Dies sind:

- Die `Point`-Klasse, die einen Punkt anhand seiner x- und y-Koordinaten sowie seines Polygons (dessen ID gespeichert wird) darstellt.
- Die `Edge`-Klasse, die eine Kante (sowohl eines Polygons als auch im Sichtbarkeitsgraphen) mithilfe ihrer beiden Endpunkte darstellt.
- Die `Graph`-Klasse. Diese modelliert einen Graphen als Dictionary bzw. Map, bei dem die Schlüssel die Punkte sind und die Werte die zu ihnen inzidenten Kanten. Außerdem werden alle Kanten in einem Set gespeichert sowie in einem weiteren Dictionary für jedes Polygon die ihm zugehörigen Kanten.

2.2.4 Die Datei `vis_graph.py`

Diese Datei enthält die Klasse `VisGraph`, die einen Sichtbarkeitsgraphen aus einer Liste von Polygonen, die wiederum eine Liste von Punkten sind, generiert. Dazu werden in der Methode `build` die Punkte in mehrere Batches unterteilt, für die einzeln der Sichtbarkeitsgraph berechnet wird. Dabei wird der in Abschnitt 1.2.2 vorgestellte Algorithmus für `vis_graph()` genutzt, wobei die Zielpunkte bzw. Companion-Punkte mit dem in Abschnitt 1.4/1.5 vorgestellten Algorithmus berechnet werden.

2.2.5 Die Datei `shortest_path.py`

Diese Datei implementiert den Dijkstra-Algorithmus und berechnet mit dessen Hilfe den besten Companion-Punkt, wobei der Ansatz aus Abschnitt 1.5 genutzt wird. Für diesen Companion-Punkt wird anschließend der Pfad vom Startpunkt aus rekonstruiert. Zur Implementierung des Dijkstra-Algorithmus wird ein ähnlich einer Priority-Queue arbeitendes Dictionary benutzt, sodass die Punkte als Schlüssel benutzt werden können.

2.2.6 Die Datei `visible_vertices.py`

In dieser Datei wird die Funktion `visible_vertices()` aus Abschnitt 1.2.2 implementiert. Um diese Funktion implementieren zu können, sind mehrere Hilfsfunktionen notwendig. Im einzelnen sind dies (neben `visible_vertices()`):

Funktion	Beschreibung
`polygon_crossing()`	Überprüft, ob ein Punkt in einem gegebenen Polygon liegt. Dazu wird der Punkt-in-Polygon-Test nach Jordan verwendet. Bei diesem sendet man vom zu untersuchenden Punkt in eine beliebige Richtung einen Strahl aus. Dann zählt man die Anzahl der Schnittpunkte mit den Kanten des Polygons. Ist diese gerade, liegt der Punkt außerhalb des Polygons, sonst innerhalb.
`edge_in_polygon()`	Überprüft, ob eine Kante zwischen 2 Punkten innerhalb eines Polygons liegt. Dies ist nur der Fall, wenn beide Endpunkte Punkte des gleichen Polygons sind *und* der Mittelpunkt der Kante in diesem Polygon liegt.
`point_in_polygon()`	Überprüft, ob ein Punkt innerhalb irgendeines Polygons liegt und verwendet dazu `polygon_crossing()`.
`edge_distance()`	Berechnet die euklidische Distanz der Endpunkte einer Kante.
`intersect_point()`	Berechnet den Schnittpunkt zweier Kanten, indem der Schnittpunkt der durch sie gegebenen linearen Funktionen berechnet wird. Dies ist in diesem Fall $x_p = \frac{n_2-n_1}{m_1-m_2}$ und $y_p = m_1 \cdot x_p + n_1$. Dabei müssen zur y-Achse parallele Geraden gesondert behandelt werden. Lineare Funktionen werden in diesem Fall genutzt, da die Berechnung bei ihnen deutlich einfacher als die Nutzung von Vektorrechnung ist.
`point_edge_distance()`	Bekommt als Eingabe eine Kante $\overline{p_1p_2}$ und eine Kante e. Berechnet die Distanz von p_1 bis zum Schnittpunkt beider Kanten.
`angle()`	Wird für das Sortieren der Kanten am Anfang des Algorithmus genutzt. Berechnet den Winkel, den eine Kante $\overline{cp}$ zur positiven x-Achse besitzt.
`angle2()`	Berechnet mithilfe des Cosinussatzes einen gesuchten Winkel in einem Dreieck.
`ccw()`	Überprüft, ob ein Punkt c gegen den Uhrzeigersinn bezüglich eines Strahls ab liegt. Dazu wird eine Version der Gaußschen Trapezformel (Shoelace formula) verwendet.
`on_segment()`	Überprüft, ob ein Punkt q auf einer Strecke pr liegt. Das ist genau dann der Fall, wenn sowohl der x-Wert als auch der y-Wert von q zwischen den x- bzw. y-Werten der beiden anderen Punkte liegen.
`edge_intersect()`	Überprüft, ob zwei Strecken sich schneiden.

Funktion	Beschreibung
`insort()` und `bisect()`	Diese beiden Funktionen sind für das Einfügen in den bzw. Suchen im binären Suchbaum, der die offenen Kanten beinhaltet, zuständig. Dafür nutzen sie **surprise** binäre Suche.

2.2.7 Die Datei `minkowski.py`

Hier wird die Erweiterung, in der Lisa als polygonal angenommen wird, implementiert. Dabei wird einfach der in Abschnitt 1.7.2 angegebene Algorithmus ausgeführt und anschließend der Hauptalgorithmus aufgerufen.

3 Beispiele

Laufzeiten

Beispiel	Knotenanzahl	Laufzeit (ca.)
lisarennt1.txt	4	210 Millisekunden
lisarennt2.txt	15	193 Millisekunden
lisarennt3.txt	51	205 Millisekunden
lisarennt4.txt	57	256 Millisekunden
lisarennt5.txt	63	273 Millisekunden
reg1000.txt (eigenes Beispiel)	1001	296 Sekunden

Die Laufzeiten wurden mit dem Linux-Befehl `time` bestimmt. Dazu wurde ein PC mit einem Intel Core i7 und 8 GB RAM verwendet. Die Laufzeiten sind demzufolge nur grobe Orientierungswerte, die von der verwendeten Hardware abhängen. Bei den Knoten wurden die Companion-Punkte nicht mitgezählt. Da dadurch jedoch die Knotenanzahl nur mit einem konstanten Faktor (≈ 2) multipliziert wird, ist dies vernachlässigbar.

Eine Laufzeitverbesserung für größere Beispiele ließe sich erreichen, indem man eine kompilierte Programmiersprache wie C++ verwendet.[9] Da die gegebenen Beispiele jedoch in weniger als einer Sekunde bearbeitet werden können und es beim BwInf eher um die Algorithmen an sich geht, erachte ich dies nicht für notwendig.

Hinweis: Die Angabe für die Dauer in den Ausgaben ist als Dezimalzahl in Minuten angegeben. Der Nachkommaanteil entspricht also nicht der Sekundenanzahl.

[9] siehe z.B. `https://stackoverflow.com/questions/801657/is-python-faster-and-lighter-than-c`

3.1 Beispiel 1

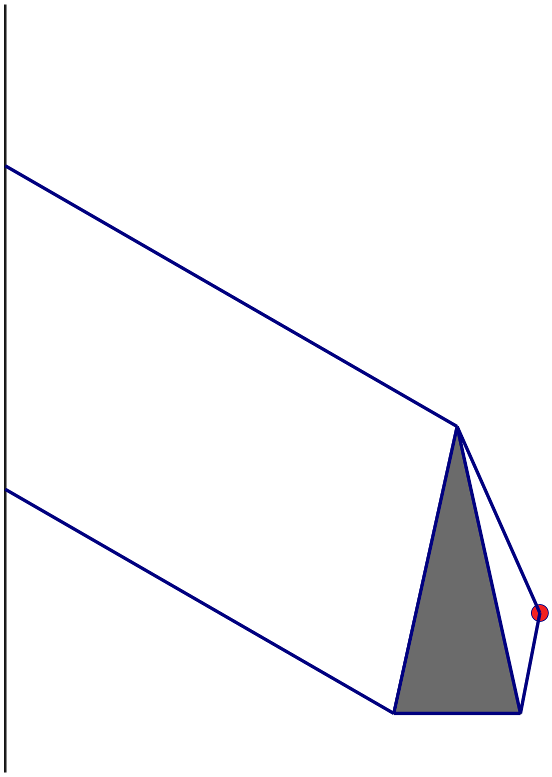


Abbildung 4: Der Sichtbarkeitsgraph für das Beispiel 1

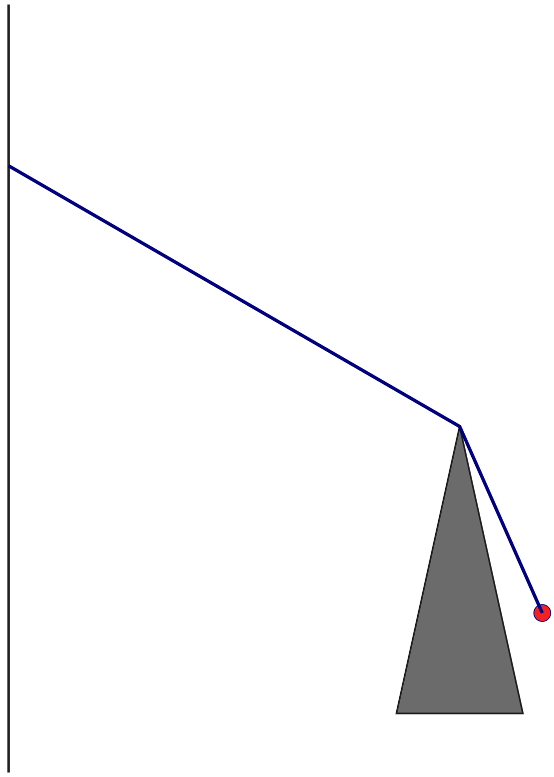


Abbildung 5: Die optimale Route für das Beispiel 1

Ausgabe für Beispiel 1

```
Lisa startet um 07:28:00 und erreicht den Bus um 07:31:26.
Sie trifft bei der y-Koordinate 718.93 auf den Bus.
Die Route dauert 3.44 Minuten und ist 859.54 Meter lang.
Die Route besteht aus folgenden Punkten:
633.0 189.0 L
535.0 410.0 P1
0.0 718.93 Y
```

3.2 Beispiel 2

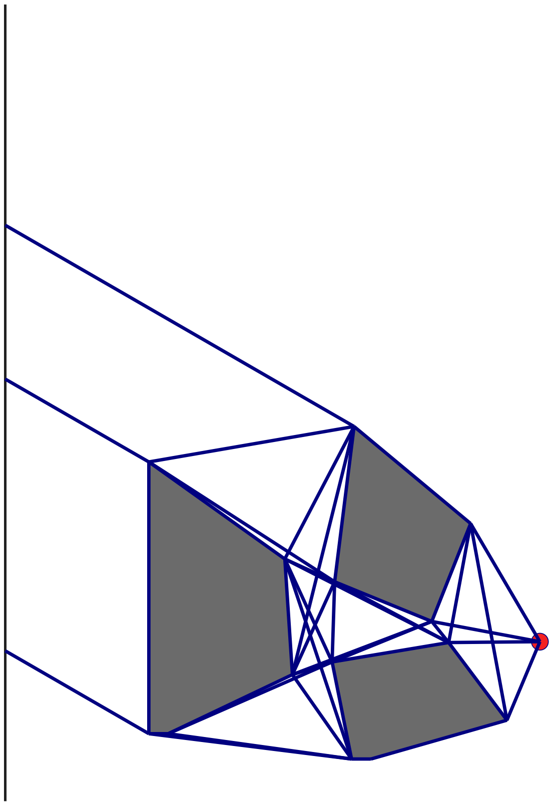


Abbildung 6: Der Sichtbarkeitsgraph für das Beispiel 2

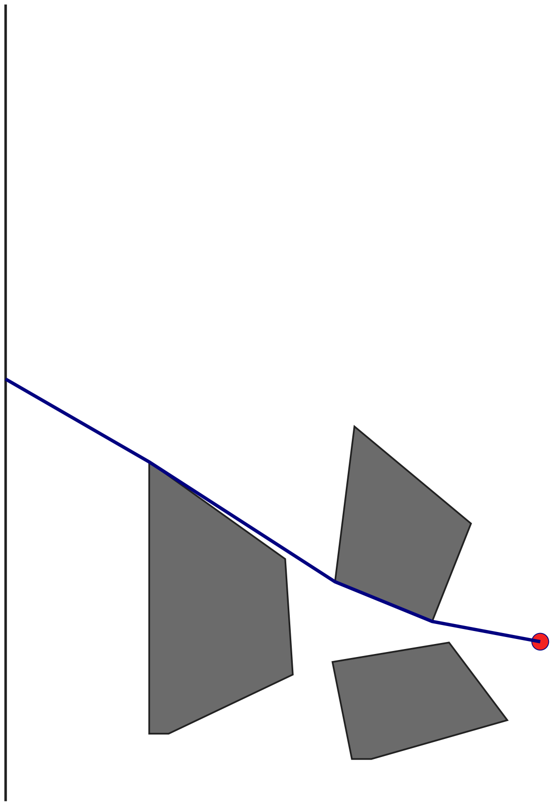


Abbildung 7: Die optimale Route für das Beispiel 2

Ausgabe für Beispiel 2

```
Lisa startet um 07:28:09 und erreicht den Bus um 07:31:00.
Sie trifft bei der y-Koordinate 500.17 auf den Bus.
Die Route dauert 2.85 Minuten und ist 712.62 Meter lang.
Die Route besteht aus folgenden Punkten:
633.0 189.0 L
505.0 213.0 P1
390.0 260.0 P1
170.0 402.0 P3
0.0 500.17 Y
```

3.3 Beispiel 3

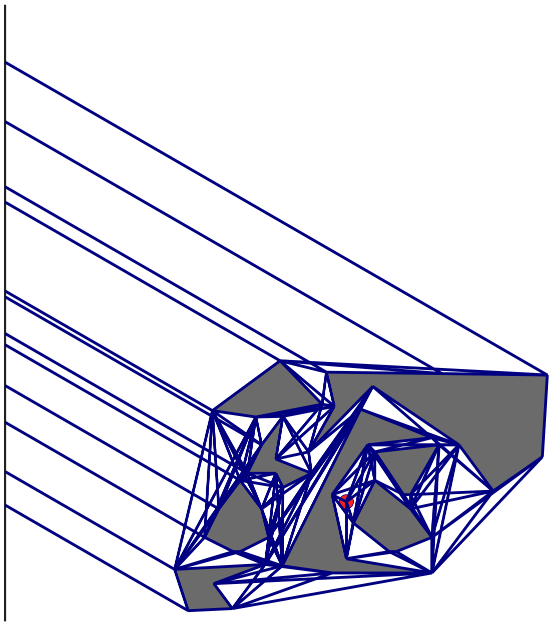


Abbildung 8: Der Sichtbarkeitsgraph für das Beispiel 3

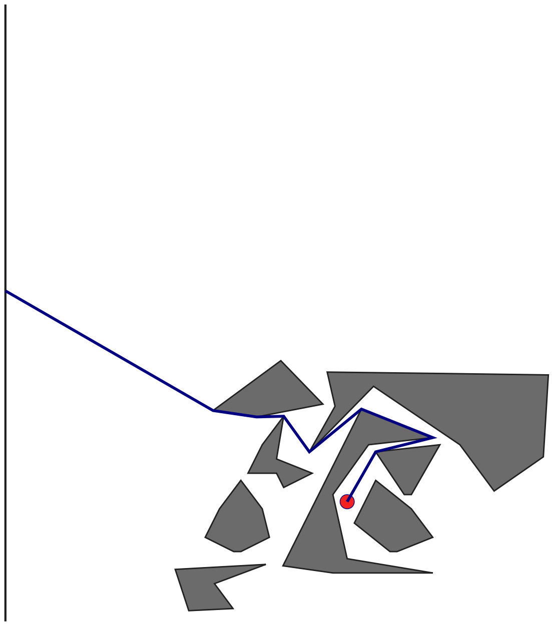


Abbildung 9: Die optimale Route für das Beispiel 3

Ausgabe für Beispiel 3

```
Lisa startet um 07:27:29 und erreicht den Bus um 07:30:56.
Sie trifft bei der y-Koordinate 464.04 auf den Bus.
Die Route dauert 3.45 Minuten und ist 862.60 Meter lang.
Die Route besteht aus folgenden Punkten:
479.0 168.0 L
519.0 238.0 P2
599.0 258.0 P3
499.0 298.0 P3
426.0 238.0 P8
390.0 288.0 P5
352.0 287.0 P6
291.0 296.0 P6
0.0 464.04 Y
```

3.4 Beispiel 4

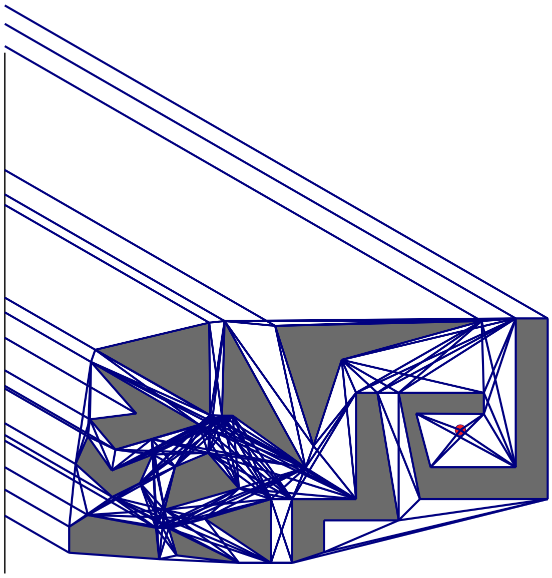


Abbildung 10: Der Sichtbarkeitsgraph für das Beispiel 4

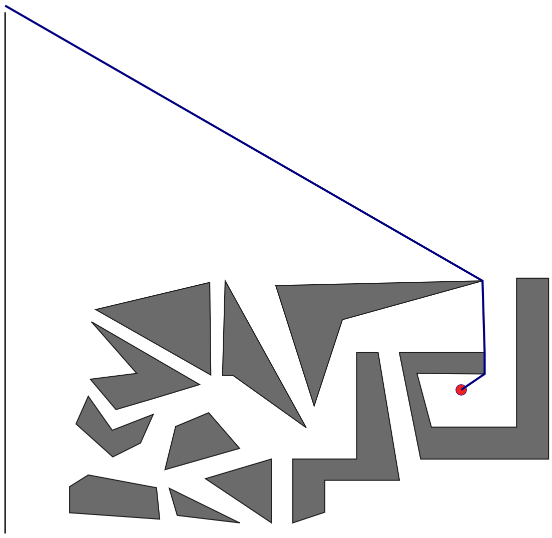


Abbildung 11: Die optimale Route für das Beispiel 4

Ausgabe für Beispiel 4

```
Lisa startet um 07:26:56 und erreicht den Bus um 07:31:59.
Sie trifft bei der y-Koordinate 992.39 auf den Bus.
Die Route dauert 5.05 Minuten und ist 1262.97 Meter lang.
Die Route besteht aus folgenden Punkten:
856.0 270.0 L
900.0 300.0 P11
900.0 340.0 P11
896.0 475.0 P10
0.0 992.39 Y
```

3.5 Beispiel 5

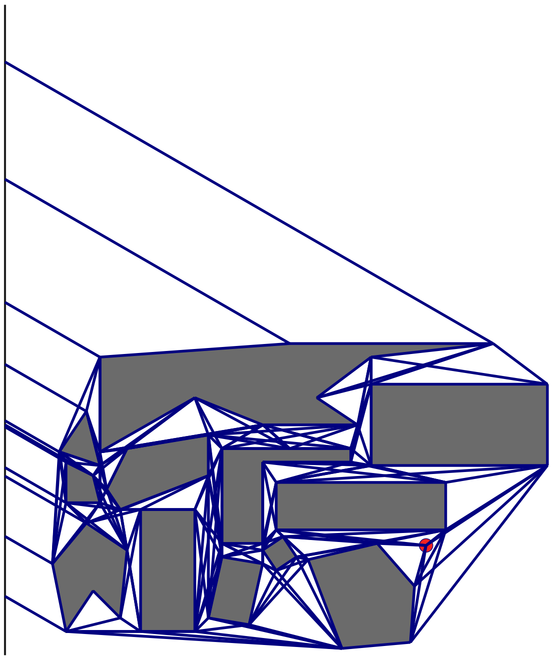


Abbildung 12: Der Sichtbarkeitsgraph für das Beispiel 5

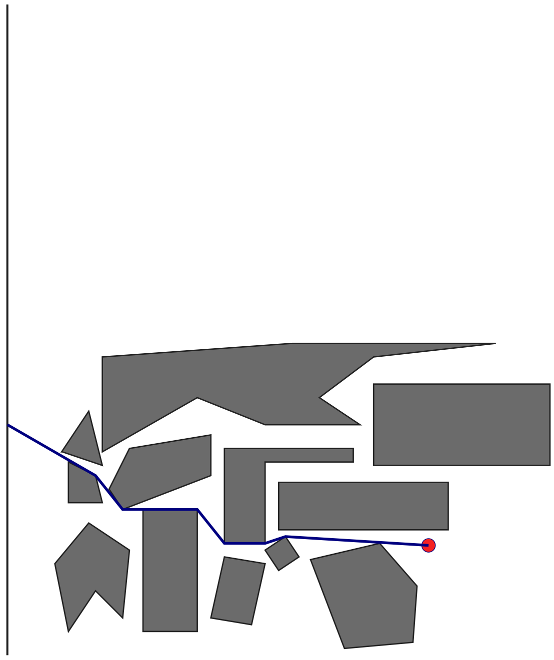


Abbildung 13: Die optimale Route für das Beispiel 5

Ausgabe für Beispiel 5

```
Lisa startet um 07:27:55 und erreicht den Bus um 07:30:41.
Sie trifft bei der y-Koordinate 340.07 auf den Bus.
Die Route dauert 2.76 Minuten und ist 691.20 Meter lang.
Die Route besteht aus folgenden Punkten:
621.0 162.0 L
410.0 175.0 P8
380.0 165.0 P3
320.0 165.0 P3
280.0 215.0 P5
170.0 215.0 P6
130.0 265.0 P9
0.0 340.07 Y
```

Eigenes Beispiel

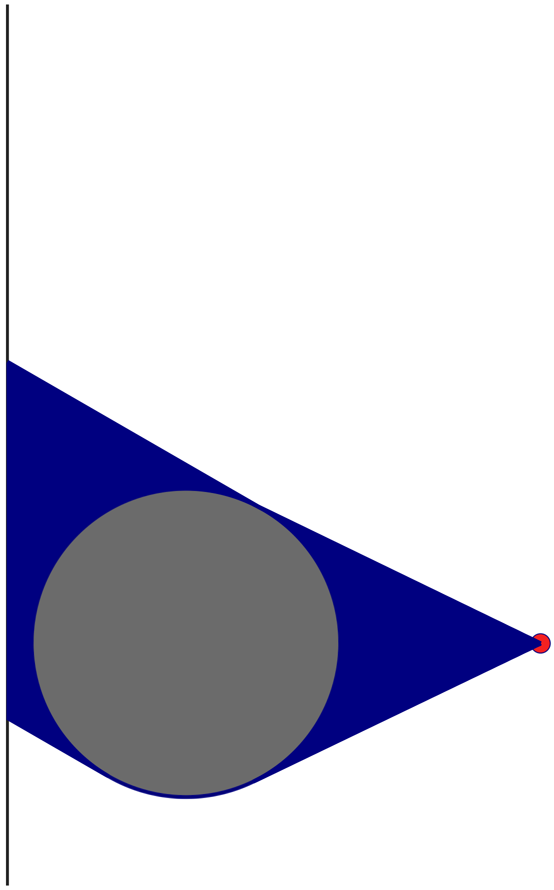


Abbildung 14: Der Sichtbarkeitsgraph für eigenes Beispiel

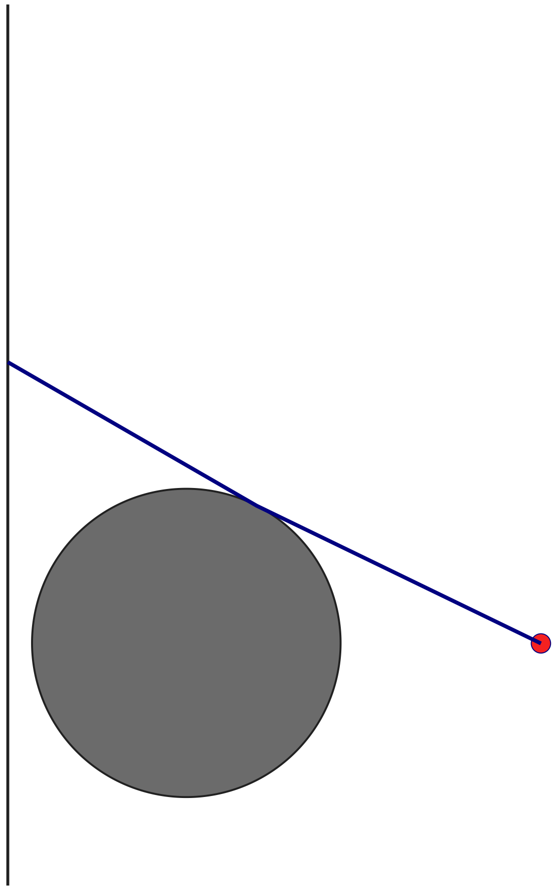


Abbildung 15: Die optimale Route für eigenes Beispiel

Ausgabe für eigenes Beispiel

```
Lisa startet um 07:28:36 und erreicht den Bus um 07:31:05.
Sie trifft bei der y-Koordinate 540.4 auf den Bus.
Die Route dauert 2.49 Minuten und ist 622.36 Meter lang.
Die Route besteht aus folgenden Punkten:
550.0 250.0 L
256.855 392.081 P1
0.0 540.4 Y
```

Bei diesen Beispiel handelt es sich um ein einziges Polygon mit 1000 Ecken. Es ist der Implementierung als `reg1000.txt` beigelegt.

3.6 Beispiele für die Erweiterungen

Geschwindigkeiten

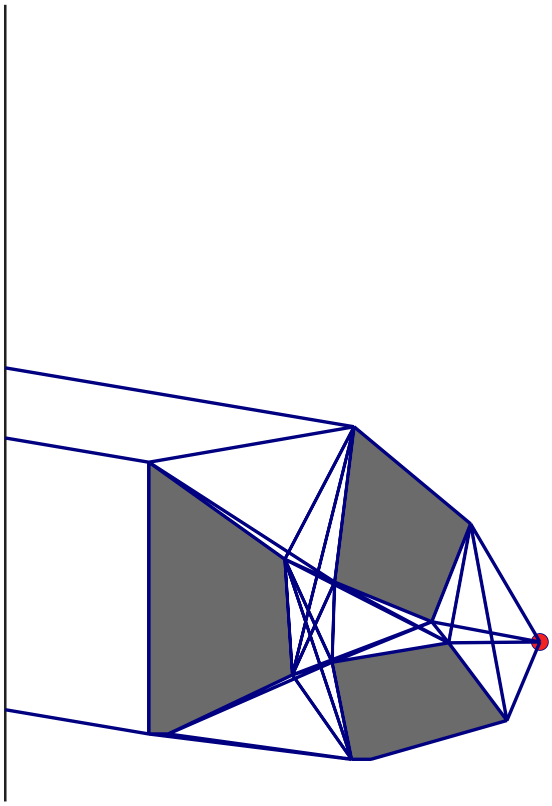


Abbildung 16: Der Sichtbarkeitsgraph

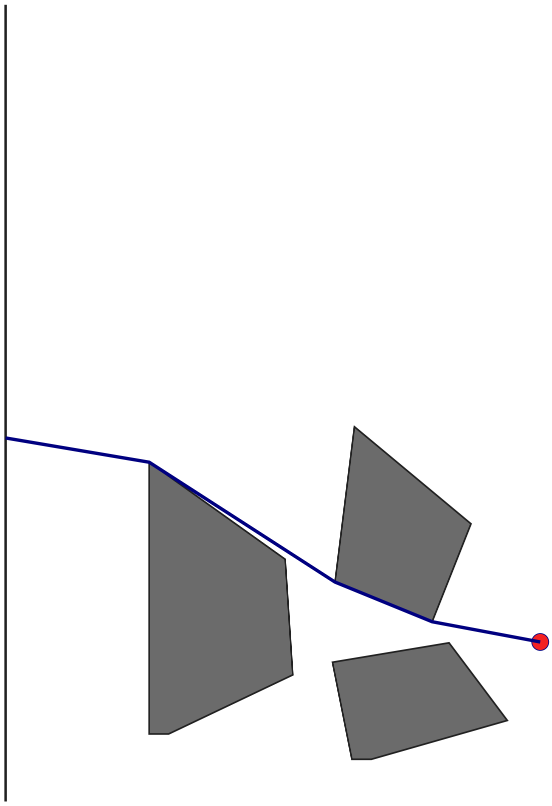


Abbildung 17: Die optimale Route

Ausgabe für das Beispiel

```
Lisa startet um 07:22:36 und erreicht den Bus um 07:30:52.
Sie trifft bei der y-Koordinate 430.74 auf den Bus.
Die Route dauert 8.26 Minuten und ist 688.72 Meter lang.
Die Route besteht aus folgenden Punkten:
633.0 189.0 L
505.0 213.0 P1
390.0 260.0 P1
170.0 402.0 P3
0.0 430.74 Y
```

Für dieses Beispiel wurde die Geschwindigkeit des Busses bei 30 km/h belassen, die von Lisa aber auf 5 km/h gesetzt, da sie sehr erschöpft ist. Es ist im Vergleich zum normalen Beispiel 2 eine deutlich geringere Steigung des letzten Streckenabschnitts zu erkennen. Außerdem dauert die Route natürlich deutlich länger.

Minkowski-Summe

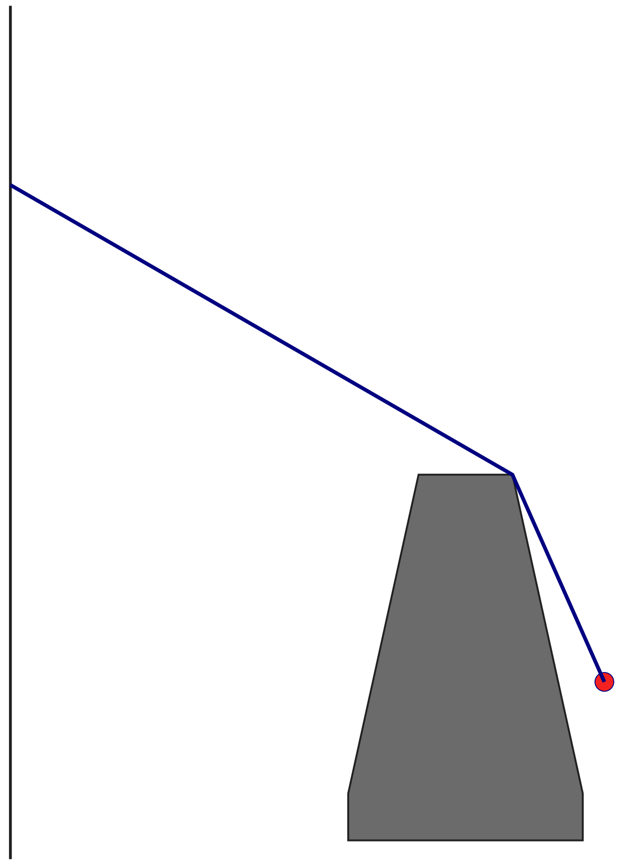


Abbildung 18: Die optimale Route

Ausgabe für das Beispiel

```
Lisa startet um 07:28:00 und erreicht den Bus um 07:31:26.
Sie trifft bei der y-Koordinate 718.93 auf den Bus.
Die Route dauert 3.44 Minuten und ist 859.54 Meter lang.
Die Route besteht aus folgenden Punkten:
633.0 189.0 L
535.0 410.0 P1
0.0 718.93 Y
```

Hier wurde als Lisa-Polygon ein Rechteck der Maße `100 x 50` benutzt, dass der Implementierung als `lisas_auto.txt` beigelegt ist. Es lässt sich eindeutig erkennen, dass das Dreieck aus Beispiel 1 durch die Berechnung der Minkowski-Summe verändert wurde und dadurch einige zusätzliche Kanten besitzt.

4 Quellcode

```python
#!/usr/bin/python3
#Imports
import argparse
import os

import pyvisgraph as vg
import svggen
import minkowski

def ensure_dir(file_path):
    directory = os.path.dirname(file_path)
    if not os.path.exists(directory):
        os.makedirs(directory)

def numtotime(num):
    num = round(abs(num))
    hours = num // 3600
    minutes = (num % 3600) // 60
    seconds = (num % 3600) % 60
    return hours, minutes, seconds

#Commandline-Argumente parsen
parser = argparse.ArgumentParser(description="Lösung zu Lisa rennt, Aufgabe 1, Runde 2, 37. BwInf von Lukas Rost")

parser.add_argument('-i', action="store",dest="input",default="lisarennt1.txt",help="Eingabedatei")
parser.add_argument('-o',action="store",dest="output", default="lisarennt1_output.txt",help="Ausgabedatei")
parser.add_argument('-so', action="store",dest="svg", default="lisarennt1_svg.svg",help="SVG-Ausgabedatei")
parser.add_argument('-d',action="store_true",default=False,dest="debug", help="Debug-Ausgaben aktivieren")
parser.add_argument('-vlisa',action="store",dest="velocity_lisa",default=15, type=float,help="Erweiterung Geschwindigkeiten: Lisa in km/h")
parser.add_argument('-vbus',action="store",dest="velocity_bus",default=30, type=float,help="Erweiterung Geschwindigkeiten: Bus in km/h")
parser.add_argument('-minkowski',action="store",default=None,help="Erweiterung Minkowski-Summe: Eingabedatei (1 Polygon im gleichen Format wie in der normalen Eingabe)")

args = parser.parse_args()

#Geschwindigkeiten in m/s umrechnen
real_v_lisa = round(args.velocity_lisa / 3.6 ,3)
real_v_bus = round(args.velocity_bus / 3.6 ,3)

#Maximale x und y für Darstellung
maxx = 0
```

```
maxy = 0

# Polygone einlesen
infile = open(args.input,'r')
numpoly = int(infile.readline())
polylist = []

for i in range(numpoly):
    pointlist = []
    line = infile.readline().split(" ")
    line = [float(x) for x in line]
    index = 1
    for j in range(int(line[0])):
        maxx = max(maxx,line[index])
        maxy = max(maxy,line[index+1])
        pointlist.append(vg.Point(line[index],line[index+1],polygon_id=("P" +
        ↪  str(i+1))))
        index += 2
    polylist.append(pointlist)

#Lisas Position einlesen
pos = infile.readline().split(" ")
pos = [float(x) for x in pos]
lisa = vg.Point(pos[0],pos[1],polygon_id="L")
infile.close()

maxx = max(maxx,pos[0])
maxy = max(maxy,pos[1])

#Erweiterung Minkowski-Summe
if args.minkowski is not None:
    minfile = open(args.minkowski,'r')
    lisa_poly = []
    line = minfile.readline().split(" ")
    minfile.close()
    line = [float(x) for x in line]
    index = 1
    for j in range(int(line[0])):
        lisa_poly.append(vg.Point(-line[index],-line[index+1]))
        index += 2
    polylist = minkowski.minkowski_sum_list(polylist,lisa_poly)

#Graph erstellen und Algorithmus ausführen
graph = vg.VisGraph(real_v_lisa,real_v_bus,lisa)
graph.build(polylist)
path,mintime,min_bus_time, minpoint,dist_minpoint = graph.shortest_path()

#Debug-Ausgaben
if args.debug:
    outpath = os.path.dirname(args.input) + "/out/debug/" +
    ↪  os.path.basename(args.input).split(".")[0]
    ensure_dir(outpath)
    svgfile = open(outpath + "-visgraph.svg","w")
    svgfile.write(svggen.gen_vis_svg(graph.get_visgraph(),polylist,lisa,
    ↪  maxx+200,maxy+500))
```

```python
    svgfile.close()


#Ausgabe SVG
svgfile = open(args.svg,"w")
svgfile.write(svggen.gen_output_svg(path,polylist,lisa,maxx+200,maxy+500))
svgfile.close()

#Ausgabe Text
outtext = ""
hours, minutes, seconds = numtotime(mintime)
# Normalfall: Startzeit vor 7.30
if mintime < 0:
    hours = 7 - hours
    minutes = 30 - minutes
    if seconds != 0:
        minutes -= 1
        seconds = 60 - seconds
# Wenn Startzeit nach 7.30
else:
    hours = 7 + hours
    minutes = 30 + minutes
bhours, bminutes, bseconds = numtotime(min_bus_time)
bhours = 7 + bhours
bminutes = 30 + bminutes
outtext += "Lisa startet um {:02d}:{:02d}:{:02d} und erreicht den Bus um {:02d}:{:02d}:{:02d}.\n".format(int(round(hours)), int(round(minutes)), int(round(seconds)),int(round(bhours)), int(round(bminutes)), int(round(bseconds)))
outtext += "Sie trifft bei der y-Koordinate {} auf den Bus.\n".format(minpoint.y)
outtext += "Die Route dauert {:0.2f} Minuten und ist {:0.2f} Meter lang.\n".format(dist_minpoint/(real_v_lisa*60),dist_minpoint)
outtext += "Die Route besteht aus folgenden Punkten:\n"
for point in path:
    outtext += "{} {} {}\n".format(point.x,point.y,point.real_polygon_id)

outfile = open(args.output,"w")
outfile.write(outtext)
outfile.close()
```

Quellcode 1: Das Hauptprogramm (*main.py*)

```python
def _vis_graph(self, points):
    """Sichtbarkeitsgraph für points berechnen. Dabei wird für jeden Punkt der Companion-Punkt berechnet. Anschließend wird dieser Punkt bei der Berechnung der visible vertices mit betrachtet. """
    visible_edges = []
    for p1 in points:
        dest_point_y = p1.y + (self.vlisa / self.vbus) * (1 / math.sqrt(1 - ((self.vlisa * self.vlisa) / (self.vbus * self.vbus)))) * p1.x
        dest_point = Point(0,round(dest_point_y,2), polygon_id="Y")
        for p2 in visible_vertices(p1, self.graph, origin=self.lisa_point,destination=dest_point):
```

```
            visible_edges.append(Edge(p1, p2))
    return visible_edges
```

Quellcode 2: Berechnungsfunktion für den Sichtbarkeitsgraph in *vis_graph.py*

```python
from __future__ import division
from math import pi, sqrt, atan, acos
from pyvisgraph.graph import Point

INF = 10000
# Toleranz für Gleitkommazahlen
COLIN_TOLERANCE = 10
T = 10**COLIN_TOLERANCE
T2 = 10.0**COLIN_TOLERANCE

def visible_vertices(point, graph, origin=None, destination=None, scan='full'):
    """Gibt Liste der von point sichtbaren Punkte zurück. origin und destination werden dabei ebenfalls mit einbezogen. scan='full' überprüft die Sichtbarkeit aller Punkte und wird hier ausschließlich verwendet.
    """
    edges = graph.get_edges()
    points = graph.get_points()
    if origin: points.append(origin)
    if destination: points.append(destination)
    points.sort(key=lambda p: (angle(point, p), edge_distance(point, p)))

    # open_edges initialisieren (alle Kanten, die die anfaengliche Sweep line schneiden)
    open_edges = []
    point_inf = Point(INF, point.y)
    for e in edges:
        if point in e: continue
        if edge_intersect(point, point_inf, e):
            if on_segment(point, e.p1, point_inf): continue
            if on_segment(point, e.p2, point_inf): continue
            k = EdgeKey(point, point_inf, e)
            insort(open_edges, k)

    visible = []
    prev = None
    prev_visible = None
    for p in points:
        if p == point: continue
        if scan == 'half' and angle(point, p) > pi: break

        #Inzidente Kanten, die in Uhrzeigerrichtung liegen, entfernen
        if open_edges:
            for edge in graph[p]:
                if ccw(point, p, edge.get_adjacent(p)) == -1:
                    k = EdgeKey(point, p, edge)
                    index = bisect(open_edges, k) - 1
                    if len(open_edges) > 0 and open_edges[index] == k:
```

```
                        del open_edges[index]

        # Sichtbarkeit von p von point aus überprüfen
        is_visible = False
        # vorheriger Punkt (prev) nicht auf der Geraden point-p
        if prev is None or ccw(point, prev, p) != 0 or not on_segment(point, prev, p):
            #keine offenen Kanten -> sichtbar
            if len(open_edges) == 0:
                is_visible = True
            #Strecke schneidet erste Kante nicht -> sichtbar
            elif not edge_intersect(point, p, open_edges[0].edge):
                is_visible = True
        # ...liegen auf einer Geraden *und* vorheriger Punkt war nicht sichtbar -> nicht sichtbar
        elif not prev_visible:
            is_visible = False
        # ...liegen auf einer Geraden *und* vorheriger Punkt war sichtbar -> alle offenen Kanten überprüfen
        else:
            is_visible = True
            for e in open_edges:
                if prev not in e.edge and edge_intersect(prev, p, e.edge):
                    is_visible = False
                    break
            if is_visible and edge_in_polygon(prev, p, graph):
                    is_visible = False

        # Strecke liegt komplett in einem Polygon -> nicht sichtbar
        if is_visible and p not in graph.get_adjacent_points(point):
            is_visible = not edge_in_polygon(point, p, graph)

        if is_visible: visible.append(p)

        # Inzidente Kanten, die gegen Uhrzeigerrichtung liegen, hinzufügen
        for edge in graph[p]:
            if (point not in edge) and ccw(point, p, edge.get_adjacent(p)) == 1:
                k = EdgeKey(point, p, edge)
                insort(open_edges, k)

        prev = p
        prev_visible = is_visible
    return visible


def polygon_crossing(p1, poly_edges):
    """Überprüft, ob Punkt in Polygon liegt und verwendet dazu die Strahl-Methode (https://de.wikipedia.org/wiki/Punkt-in-Polygon-Test_nach_Jordan). """
    p2 = Point(INF, p1.y)
    intersect_count = 0
    co_flag = False
    co_dir = 0
```

```python
    for edge in poly_edges:
        if p1.y < edge.p1.y and p1.y < edge.p2.y: continue
        if p1.y > edge.p1.y and p1.y > edge.p2.y: continue
        co0 = (ccw(p1, edge.p1, p2) == 0) and (edge.p1.x > p1.x)
        co1 = (ccw(p1, edge.p2, p2) == 0) and (edge.p2.x > p1.x)
        if co0 and co1: continue
        co_point = edge.p1 if co0 else edge.p2
        if co0 or co1:
            if edge.get_adjacent(co_point).y > p1.y:
                co_dir += 1
            else:
                co_dir -= 1
            if co_flag:
                if co_dir == 0:
                    intersect_count += 1
                co_flag = False
                co_dir = 0
            else:
                co_flag = True
        elif edge_intersect(p1, p2, edge):
            intersect_count += 1
    if intersect_count % 2 == 0:
        return False
    return True


def edge_in_polygon(p1, p2, graph):
    """Überprüft, ob Kante zwischen zwei Polygon-Punkten innerhalb eines Polygons liegt. """
    if p1.polygon_id != p2.polygon_id:
        return False
    if p1.polygon_id == -1 or p2.polygon_id == -1:
        return False
    mid_point = Point((p1.x + p2.x) / 2, (p1.y + p2.y) / 2)
    return polygon_crossing(mid_point, graph.polygons[p1.polygon_id])


def point_in_polygon(p, graph):
    """Überprüft, ob Punkt innerhalb irgendeines Polygons liegt. """
    for polygon in graph.polygons:
        if polygon_crossing(p, graph.polygons[polygon]):
            return polygon
    return -1

def edge_distance(p1, p2):
    """Euklidische Distanz zwischen zwei Punkten."""
    return sqrt((p2.x - p1.x)**2 + (p2.y - p1.y)**2)


def intersect_point(p1, p2, edge):
    """Schnittpunkt von p1p2 und edge."""
    if p1 in edge: return p1
    if p2 in edge: return p2
    if edge.p1.x == edge.p2.x:
```

```python
        if p1.x == p2.x:
            return None
        pslope = (p1.y - p2.y) / (p1.x - p2.x)
        intersect_x = edge.p1.x
        intersect_y = pslope * (intersect_x - p1.x) + p1.y
        return Point(intersect_x, intersect_y)

    if p1.x == p2.x:
        eslope = (edge.p1.y - edge.p2.y) / (edge.p1.x - edge.p2.x)
        intersect_x = p1.x
        intersect_y = eslope * (intersect_x - edge.p1.x) + edge.p1.y
        return Point(intersect_x, intersect_y)

    pslope = (p1.y - p2.y) / (p1.x - p2.x)
    eslope = (edge.p1.y - edge.p2.y) / (edge.p1.x - edge.p2.x)
    if eslope == pslope:
        return None
    intersect_x = (eslope * edge.p1.x - pslope * p1.x + p1.y - edge.p1.y) /
    ↪  (eslope - pslope)
    intersect_y = eslope * (intersect_x - edge.p1.x) + edge.p1.y
    return Point(intersect_x, intersect_y)


def point_edge_distance(p1, p2, edge):
    """Distanz von p1 bis zum Schnittpunkt von p1p2 mit edge."""
    ip = intersect_point(p1, p2, edge)
    if ip is not None:
        return edge_distance(p1, ip)
    return 0


def angle(center, point):
    """Winkel zwischen Gerade cp und positiver x-Achse.
      ------p
      |   /
      |  /
     c|a/
    """
    dx = point.x - center.x
    dy = point.y - center.y
    if dx == 0:
        if dy < 0:
            return pi * 3 / 2
        return pi / 2
    if dy == 0:
        if dx < 0:
            return pi
        return 0
    if dx < 0:
        return pi + atan(dy / dx)
    if dy < 0:
        return 2 * pi + atan(dy / dx)
    return atan(dy / dx)
```

```

def angle2(point_a, point_b, point_c):
    """Implementierung des Cosinussatzes.
            c
          /   \
        /    B\
      a-------b
    """
    a = (point_c.x - point_b.x)**2 + (point_c.y - point_b.y)**2
    b = (point_c.x - point_a.x)**2 + (point_c.y - point_a.y)**2
    c = (point_b.x - point_a.x)**2 + (point_b.y - point_a.y)**2
    cos_value = (a + c - b) / (2 * sqrt(a) * sqrt(c))
    return acos(int(cos_value*T)/T2)


def ccw(A, B, C):
    """Ueberprueft, ob C gegen den Uhrzeigersinn bezueglich des Strahls AB liegt (1 wenn ja, -1 wenn im Uhrzeigersinn, 0 wenn auf dem Strahl) """
    #  Rounding this way is faster than calling round()
    area = int(((B.x - A.x) * (C.y - A.y) - (B.y - A.y) * (C.x - A.x))*T)/T2
    if area > 0: return 1
    if area < 0: return -1
    return 0


def on_segment(p, q, r):
    """Ueberprüft, ob q auf pr liegt."""
    if (q.x <= max(p.x, r.x)) and (q.x >= min(p.x, r.x)):
        if (q.y <= max(p.y, r.y)) and (q.y >= min(p.y, r.y)):
            return True
    return False


def edge_intersect(p1, q1, edge):
    """Ueberprueft ob p1q1 und edge sich schneiden.
    http://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/"""
    p2 = edge.p1
    q2 = edge.p2
    o1 = ccw(p1, q1, p2)
    o2 = ccw(p1, q1, q2)
    o3 = ccw(p2, q2, p1)
    o4 = ccw(p2, q2, q1)

    # General case
    if (o1 != o2 and o3 != o4):
        return True
    # p1, q1 and p2 are colinear and p2 lies on segment p1q1
    if o1 == 0 and on_segment(p1, p2, q1):
        return True
    # p1, q1 and p2 are colinear and q2 lies on segment p1q1
    if o2 == 0 and on_segment(p1, q2, q1):
        return True
```

```
    # p2, q2 and p1 are colinear and p1 lies on segment p2q2
    if o3 == 0 and on_segment(p2, p1, q2):
        return True
    # p2, q2 and q1 are colinear and q1 lies on segment p2q2
    if o4 == 0 and on_segment(p2, q1, q2):
        return True
    return False


def insort(a, x):
    """Einfügen in den binaeren Suchbaum. """
    lo = 0
    hi = len(a)
    while lo < hi:
        mid = (lo+hi)//2
        if x < a[mid]: hi = mid
        else: lo = mid+1
    a.insert(lo, x)


def bisect(a, x):
    """Binaere Suche für den Suchbaum."""
    lo = 0
    hi = len(a)
    while lo < hi:
        mid = (lo+hi)//2
        if x < a[mid]: hi = mid
        else: lo = mid+1
    return lo


class EdgeKey(object):
    """Element des Suchbaums. """
    def __init__(self, p1, p2, edge):
        self.p1 = p1
        self.p2 = p2
        self.edge = edge

    def __eq__(self, other):
        if self.edge == other.edge:
            return True

    def __lt__(self, other):
        if self.edge == other.edge:
            return False
        if not edge_intersect(self.p1, self.p2, other.edge):
            return True
        self_dist = point_edge_distance(self.p1, self.p2, self.edge)
        other_dist = point_edge_distance(self.p1, self.p2, other.edge)
        if self_dist > other_dist:
            return False
        if self_dist < other_dist:
            return True
        # If the distance is equal, we need to compare on the edge angles.
```

```
        if self_dist == other_dist:
            if self.edge.p1 in other.edge:
                same_point = self.edge.p1
            elif self.edge.p2 in other.edge:
                same_point = self.edge.p2
            aslf = angle2(self.p1, self.p2, self.edge.get_adjacent(same_point))
            aot = angle2(self.p1, self.p2, other.edge.get_adjacent(same_point))
            if aslf < aot:
                return True
            return False

    def __repr__(self):
        reprstring = (self.__class__.__name__, self.edge, self.p1, self.p2)
        return "{}(Edge={!r}, p1={!r}, p2={!r})".format(*reprstring)
```

Quellcode 3: Die Datei *visible_vertices.py*, in der alle geometrischen Algorithmen gesammelt sind. Die Funktion `visible_vertices` wird bei der Erstellung des Sichtbarkeitsgraphs benutzt.

```
#Implementierung des Dijkstra-Algorithmus
#mit einem Priority-Dict (aehnlich Priority-Queue) umgesetzt
def dijkstra(graph, origin):
    D = {} #Distanz
    P = {} #Elternknoten
    Q = priority_dict()
    Q[origin] = 0

    for v in Q:
        D[v] = Q[v]

        edges = graph[v]
        for e in edges:
            w = e.get_adjacent(v)
            elength = D[v] + edge_distance(v, w)
            if w in D:
                if elength < D[w]:
                    raise ValueError
            elif w not in Q or elength < Q[w]:
                Q[w] = elength
                P[w] = v
    return (D, P)

#Kuerzesten Weg vom Startpunkt zur y-Achse bestimmen
#Idee ist in Kapitel 1.5 beschrieben
def shortest_path(graph, origin, vlisa, vbus):
    #Dijkstra für alle Knoten ausführen
    D, P = dijkstra(graph, origin)
    mintime = -math.inf
    minpoint = Point(0,0)
    min_bus_time = 0.0
    #Besten Companion-Punkt bestimmen...
    for point in graph.get_points():
```

```
        if point.x == 0:
            bus_time = point.y / vbus
            lisa_time = D[point] / vlisa
            #... mithilfe der spätesten Startzeit
            #Zeitangabe: negativ vor 7.30, positiv danach
            total_time = bus_time - lisa_time
            if total_time > mintime:
                mintime = round(total_time,2)
                minpoint = point
                min_bus_time = round(bus_time,2)
    #Pfad zum Punkt finden und zurückgeben
    path = []
    destination = minpoint
    while 1:
        path.append(destination)
        if destination == origin: break
        destination = P[destination]
    path.reverse()
    return path,mintime,min_bus_time, minpoint,round(D[minpoint],2)
```

Quellcode 4: Implementierung des Dijkstra-Algorithmus und der Berechnung des kürzesten Pfades in *shortest_path.py*

```
import math
from pyvisgraph.graph import Point

#Minkowski-Summe auf alle Polygone anwenden
def minkowski_sum_list(polygons, lisa_poly):
    newpolygons = []
    for poly in polygons:
        newpolygons.append(minkowski_sum(poly,lisa_poly))
    return newpolygons

#Algorithmus für die Minkowski-Summe wie in der Dokumentation beschrieben
#zuerst Summe laut Definition und dann Convex Hull mit Graham Scan
def minkowski_sum(v,w):
    sump = []
    for p1 in v:
        for p2 in w:
            sump.append(Point(p1.x + p2.x, p1.y + p2.y, polygon_id=p1.real_polygon_id))
    p0 = sump[findsmallest(sump)]
    chull = [p0]
    sump.remove(p0)
    sump.sort(key=lambda point: angle(p0,point))
    for point in sump:
        while len(chull) > 1 and ccw(chull[len(chull)-2],chull[len(chull)-1],point) < 0:
            chull.pop()
        chull.append(point)
    return chull

# Richtung der drei Punkte feststellen (Counter/Clockwise)
# mittels z-Koordinate des Kreuzprodukts aus ab und ac
```

```python
def ccw(a,b,c):
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x)

#Winkel zwischen Vektor ab und positiver x-Achse
#Ansatz siehe https://math.stackexchange.com/questions/1910825/how-do-i-find-the-angle-a-vector-makes-to-the-x-axis
def angle(a, b):
    vx = b.x - a.x
    vy = b.y - a.y
    angle = math.atan2(vy,vx)
    if vy < 0:
        angle += 2 * math.pi
    return angle

#finde den Punkt mit der kleinsten y-Koordinate und setze ihn als ersten Punkt im Polygon
def findsmallest(polygon):
    index = 0
    miny = math.inf
    minx = math.inf
    for i,point in enumerate(polygon):
        if point.y < miny or (point.y == miny and point.x < minx):
            minx = point.x
            miny = point.y
            index = i
    return index
```

Quellcode 5: Quellcode der Erweiterung (*minkowski.py*)