

Aufgabe 1

„Lisa rennt“- Dokumentation

37. Bundeswettbewerb Informatik 2018/19 - 2. Runde

Lukas Rost

Teilnahme-ID: 48125

29. April 2019

Inhaltsverzeichnis

1	Lösungsidee	1
1.1	Das Geometric-Shortest-Path-Problem	1
1.2	Erzeugung eines Sichtbarkeitsgraphen	2
1.2.1	Naiver Ansatz	2
1.2.2	Der Lee-Algorithmus	3
1.3	Der Dijkstra-Algorithmus	5
1.4	Lösung des Problems ohne Hindernisse	6
1.5	Kombination der Ansätze	7
1.6	Laufzeitbetrachtung	8
1.7	Mögliche und implementierte Erweiterungen	8
1.7.1	Anpassbare Geschwindigkeiten	8
1.7.2	Polygonale Lisa	9
1.7.3	Nicht-polygonale Hindernisse	10
1.7.4	Das Problem im \mathbb{R}^3	10
2	Umsetzung	11
2.1	Allgemeine Hinweise zur Benutzung	11
2.2	Struktur des Programms und Implementierung der Algorithmen	12
2.2.1	Die Datei <code>main.py</code>	12
2.2.2	Die Datei <code>svggen.py</code>	12
2.2.3	Die Datei <code>graph.py</code>	12
2.2.4	Die Datei <code>vis_graph.py</code>	12
2.2.5	Die Datei <code>shortest_path.py</code>	13
2.2.6	Die Datei <code>visible_vertices.py</code>	13
2.2.7	Die Datei <code>minkowski.py</code>	14
3	Beispiele	14
3.1	Beispiel 1	15
3.2	Beispiel 2	16
3.3	Beispiel 3	17
3.4	Beispiel 4	18

3.5	Beispiel 5	19
3.6	Beispiele für die Erweiterungen	21
4	Quellcode	22

1 Lösungsidee

1.1 Das Geometric-Shortest-Path-Problem

Das der Aufgabe zugrundeliegende Problem nennt sich *Geometric Shortest Path* (GSP), auf Deutsch auch bekannt als Problem des geometrischen kürzesten Weges. 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. 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.

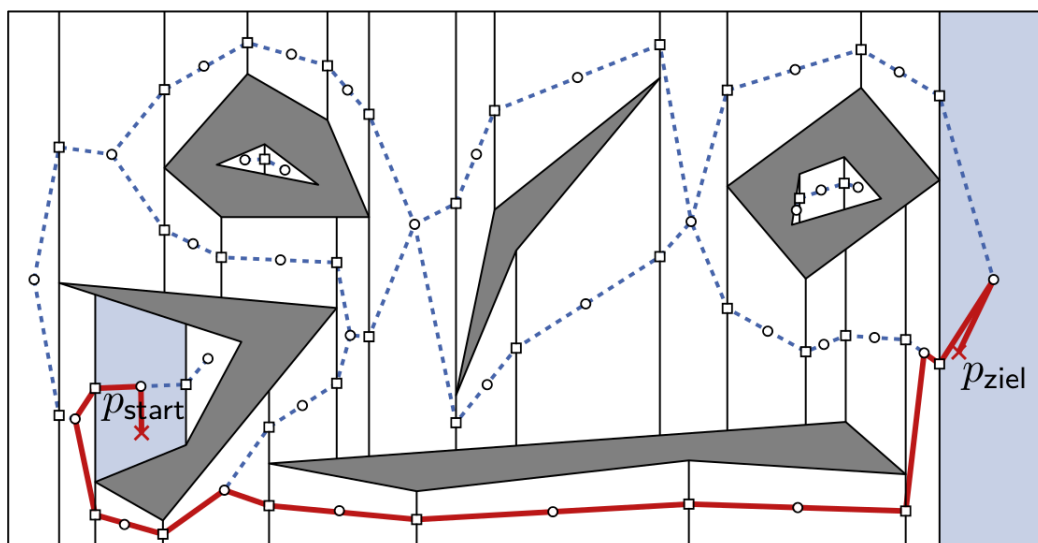


Abbildung 1: Illustration des GSP-Problems (aus [2])

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

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

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 $\mathcal{O}(n^3)$.

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

Algorithmus 1 Naiver Algorithmus

```

function VISIBILITYGRAPH( $S$ )
   $G = (V(S), E) \leftarrow$  leerer Sichtbarkeitsgraph
  for all Knoten  $v \in V(S)$  do  $\triangleright \mathcal{O}(n)$ 
    for all Knoten  $w \in V(S) \setminus \{v\}$  do  $\triangleright \mathcal{O}(n)$ 
      for all Kante  $e \in E_{vis}(S)$  do  $\triangleright \mathcal{O}(n)$ 
        if  $\overline{vw}$  schneidet keine der Kanten  $e$  then
           $E \leftarrow E \cup \{vw\}$ 
        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( $S$ )
   $G = (V(S), E) \leftarrow$  leerer Sichtbarkeitsgraph
  for all Knoten  $v \in V(S)$  do                                      $\triangleright \mathcal{O}(n)$ 
     $W \leftarrow \text{VISIBLE\_VERTICES}(v, S)$                           $\triangleright \mathcal{O}(n \cdot \log n)$ 
     $E \leftarrow E \cup \{vw \mid w \in W\}$ 
  end for
end function

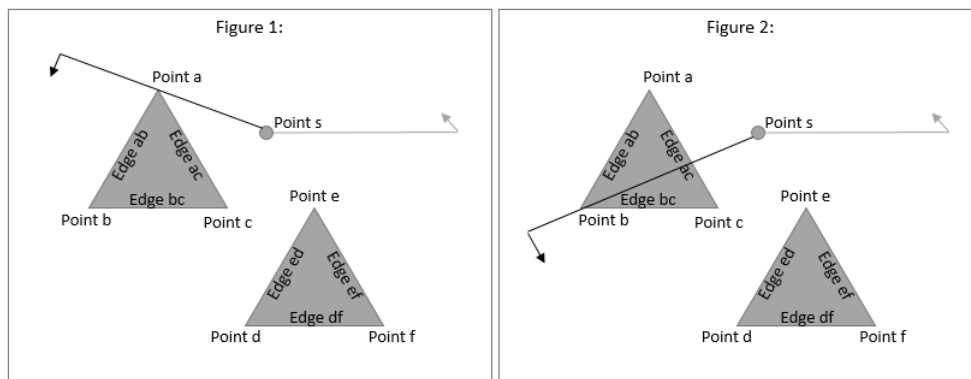
```

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 dem 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.³[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( $v, S$ )
   $w \leftarrow \text{sort}(V(S) \setminus \{v\})$                                 ▷ Sortieren der Knoten (siehe unten)
   $s \leftarrow v + \lambda \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$  mit  $\lambda \in \mathbb{R}^+$       ▷ Sweep line initialisieren
   $T \leftarrow \text{binarySearchTree}()$ 
   $W \leftarrow \emptyset$ 
  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
  
```

³Die 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  $w_{i-1}$  nicht auf der Geraden durch  $\overline{vw_i}$  then
    if  $T = \emptyset$  then
      return true
    end if
    if  $\overline{vw_i} \cap \text{smallest}(T) = \emptyset$  then
      return true
    end if
  else if not VISIBLE( $w_{i-1}$ ) then
    return false
  else
    Überprüfe alle Kanten in  $T$  auf Schnitt mit  $\overline{vw_i}$  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. Sonderfälle wie, dass eine Strecke komplett in einem Polygon liegt, wurden hier der Einfachheit halber nicht berücksichtigt. Dies muss jedoch in der Implementierung geschehen.

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.

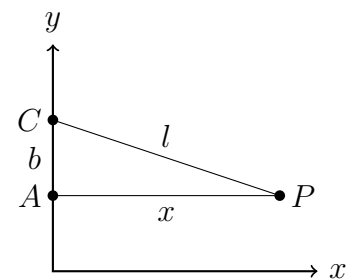
Zunächst lässt sich feststellen, dass es ausschließlich sinnvoll ist, Punkte mit einer y-Koordinate \geq der y-Koordinate von Lisas Haus als Zielpunkte zu wählen. Sonst würde der Abstand zwischen Start- und Zielpunkt größer, während die Abfahrtszeit des Busses früher läge, was insgesamt für Lisa ein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

$$t_B = \frac{b}{v_B} \quad (1)$$

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

$$t_L = \frac{l}{v_L} = \frac{\sqrt{x^2 + b^2}}{v_L} \quad (2)$$



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:

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

Setzt man diese Ableitung nun gleich 0, erhält man:

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

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

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

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

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

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

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

Für den optimalen Zielpunkt C gilt somit $x_c = 0$ und $y_c = y_p + \frac{v_L}{v_B} \cdot \frac{1}{\sqrt{1 - \frac{v_L^2}{v_B^2}}} \cdot x_p$.⁴ 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-Algorithmus⁴ 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.

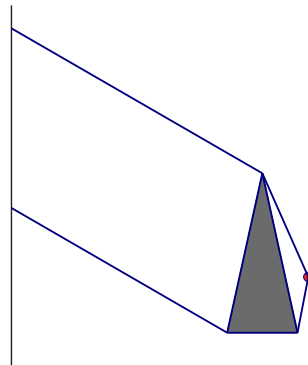


Abbildung 3: Veranschaulichung eines kompletten Sichtbarkeitsgraphen mit Companion-Punkten für Beispiel 1 des BwInf (blaue Linien sind Kanten des Graphen)

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

⁴Ich 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. 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 werden⁵ 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 $\mathcal{O}(n^2 \cdot \log n)$ \square

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 $\mathcal{O}(n^2 \cdot \log n)$. \square

Kombination der Ansätze. Da bei der Berechnung des Companion-Punktes nur einfache Grundrechenoperationen 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 $\mathcal{O}(n)$. \square

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).

⁵Normalerweise 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.⁶

Dies lässt sich mathematisch beschreiben, indem man für jedes Hindernispolygon dessen Minkowski-Summe mit dem Lisa-Polygon 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:

$$A + B := \{a + b \mid a \in A, b \in B\} \quad (11)$$

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 $P = (v_1, \dots, v_k)$ und das Lisa-Polygon $L = (w_1, \dots, w_m)$ mit v_i und w_i als Eckpunkten⁷, wobei v_1 und w_1 die Punkte mit der minimalen y-Koordinate⁸ sind. Für diese zwei konvexen Polygone (im weiteren werden für diese Erweiterung nur solche betrachtet) im \mathbb{R}^2 kann die Minkowski-Summe (bzw. deren konvexe Hülle) wie folgt berechnet werden:

Algorithmus 5 Bestimmung der Minkowski-Summe

```

function MINKOWSKI SUM( $P, L$ )
   $i \leftarrow 1; j \leftarrow 1$ 
   $v_{k+1} \leftarrow v_1; v_{k+2} \leftarrow v_2; w_{m+1} \leftarrow w_1; w_{m+2} \leftarrow w_2$ 
   $S = \emptyset$  ▷ leeres Polygon
  repeat
     $S = S \cup \{v_i + w_j\}$ 
    if WINKEL( $v_i, v_{i+1}$ ) < WINKEL( $w_j, w_{j+1}$ ) then
       $i \leftarrow i + 1$ 
    else if WINKEL( $v_i, v_{i+1}$ ) > WINKEL( $w_j, w_{j+1}$ ) then
       $j \leftarrow j + 1$ 
    else
       $i \leftarrow i + 1; j \leftarrow j + 1$ 
    end if
  until  $i = k + 1$  and  $j = m + 1$ 
end function

```

Dabei gibt die Funktion $winkel(p, q)$ den Winkel zwischen der positiven x-Achse und dem Vektor pq zurück, was mithilfe des $atan2(y, x)$ (Arkustangens, der die Richtung berücksichtigt) implementiert werden kann. Dieser Algorithmus arbeitet ähnlich wie der *Merge*-Schritt des Mergesort-Verfahrens.

⁶Hier Fridays-for-Future-Demonstration einfügen.

⁷gegen den Uhrzeigersinn geordnet

⁸Falls dies mehrere sind, derjenige mit der minimalen x-Koordinate.

Dieser Algorithmus läuft offensichtlich in $\mathcal{O}(k+m)$, denn i wird höchstens k Mal inkrementiert und j m Mal. Da er $n_{Polygon}$ Mal⁹ ausgeführt wird, erhält man $\mathcal{O}(\sum_1^{n_{Polygon}} (k_i + m_i)) = \mathcal{O}(n + n_{Polygon} \cdot m)$, wobei n die Anzahl aller Eckpunkte aller Polygone ist. Diese Erweiterung wurde **implementiert**.

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. 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/alggeom-ss11-v114-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>

⁹Anzahl der Polygone

- [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>

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. 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 kompatibel 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:

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

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
<code>polygon_crossing()</code>	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.
<code>edge_in_polygon()</code>	Überprüft, ob eine Kante zwischen 2 Punkten innerhalb eines Polygons liegt. Dies ist nur der Fall, wenn beide Endpunkte Punkte des gleichen Polygons sind <i>und</i> der Mittelpunkt der Kante in diesem Polygon liegt.
<code>point_in_polygon()</code>	Überprüft, ob ein Punkt innerhalb irgendeines Polygons liegt und verwendet dazu <code>polygon_crossing()</code> .
<code>edge_distance()</code>	Berechnet die euklidische Distanz der Endpunkte einer Kante.
<code>intersect_point()</code>	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.
<code>point_edge_distance()</code>	Bekommt als Eingabe eine Kante $\overline{p_1 p_2}$ und eine Kante e . Berechnet die Distanz von p_1 bis zum Schnittpunkt beider Kanten.
<code>angle()</code>	Wird für das Sortieren der Kanten am Anfang des Algorithmus genutzt. Berechnet den Winkel, den eine Kante \overline{cp} zur positiven x-Achse besitzt.
<code>angle2()</code>	Berechnet mithilfe des Cosinussatzes einen gesuchten Winkel in einem Dreieck.
<code>ccw()</code>	Überprüft, ob ein Punkt c gegen den Uhrzeigersinn bezüglich eines Strahls ab liegt. Dazu wird eine Version der Gaußschen Trapezformel (Shoelace formula) verwendet.
<code>on_segment()</code>	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.
<code>edge_intersect()</code>	Überprüft, ob zwei Strecken sich schneiden.

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

2.2.7 Die Datei `minkowski.py`

Hier wird die Erweiterung, in der Lisa als polygonal angenommen wird, implementiert. Nachdem die Vorbedingungen für alle Polygone hergestellt wurden (counter-clockwise und beginnend mit der kleinsten y-Koordinate), kann der in Abschnitt 1.7.2 angegebene Algorithmus darauf ausgeführt werden. Es war mir jedoch aufgrund total bescheuerter und nicht auffindbarer Implementierungsfehler nicht möglich, eine lauffähige Version der Erweiterung zu implementieren. Der Versuch, dies zu tun, ist im Quellcode-Teil abgedruckt.

3 Beispiele

Laufzeiten

Beispiel	Knotenanzahl	Laufzeit (ca.)
<code>lisarennt1.txt</code>	4	210 Millisekunden
<code>lisarennt2.txt</code>	15	193 Millisekunden
<code>lisarennt3.txt</code>	51	205 Millisekunden
<code>lisarennt4.txt</code>	57	256 Millisekunden
<code>lisarennt5.txt</code>	63	273 Millisekunden
<code>reg1000.txt</code> (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.¹⁰ 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.

¹⁰siehe z.B. <https://stackoverflow.com/questions/801657/is-python-faster-and-lighter-than-c>

3.1 Beispiel 1

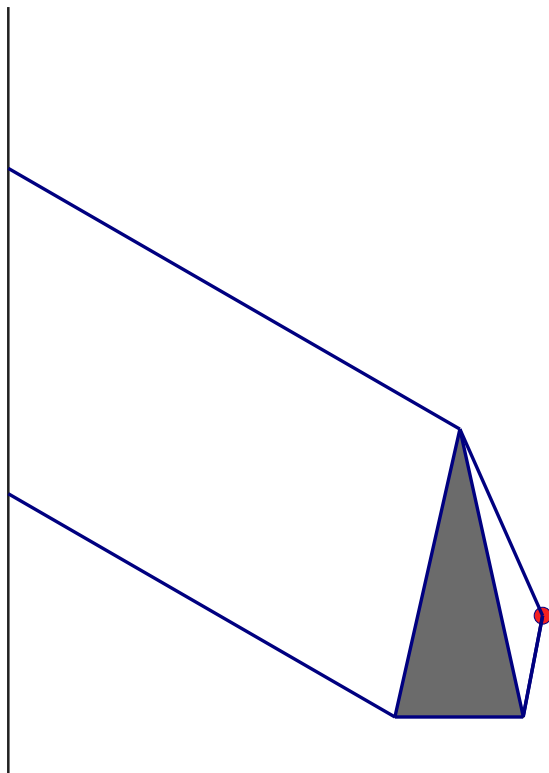


Abbildung 4: Der Sichtbarkeitsgraph für das Beispiel 1

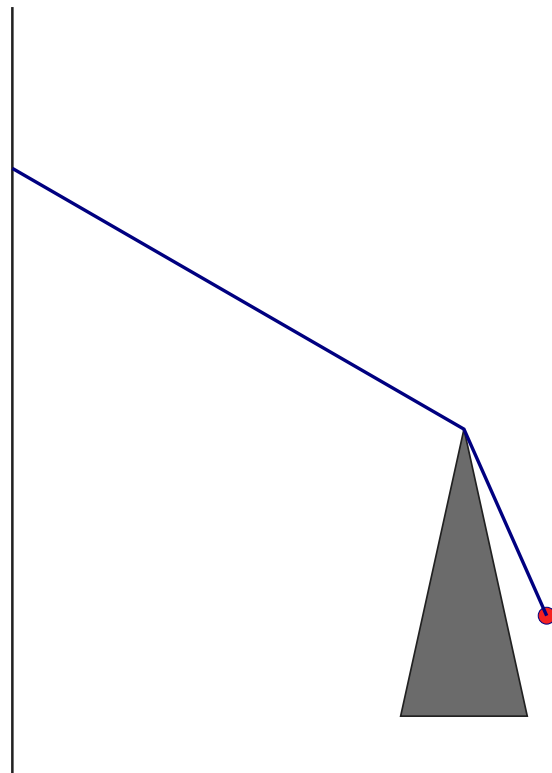


Abbildung 5: Die optimale Route für das Beispiel 1

Ausgabe für Beispiel 1

```

1 Lisa startet um 07:28:00 und erreicht den Bus um 07:31:26.
2 Sie trifft bei der y-Koordinate 718.93 auf den Bus.
3 Die Route dauert 3.44 Minuten und ist 859.54 Meter lang.
4 Die Route besteht aus folgenden Punkten:
5 633.0 189.0 L
6 535.0 410.0 P1
7 0.0 718.93 Y

```

3.2 Beispiel 2

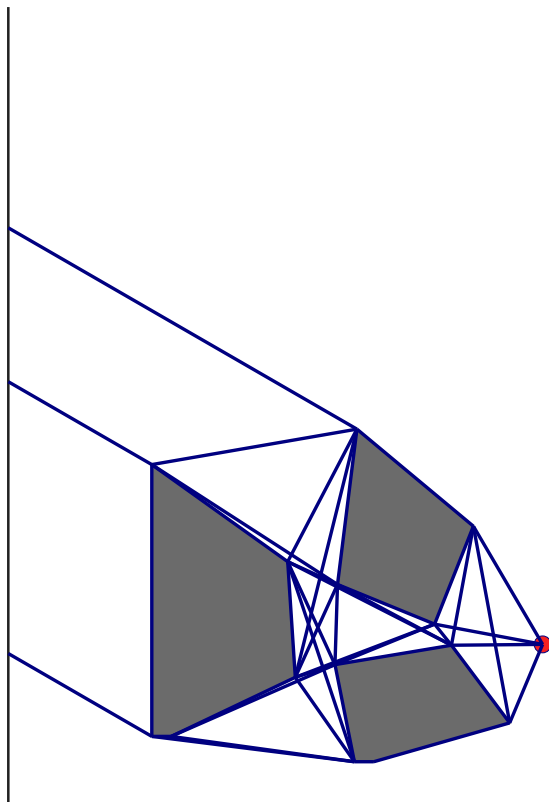


Abbildung 6: Der Sichtbarkeitsgraph für das Beispiel 2

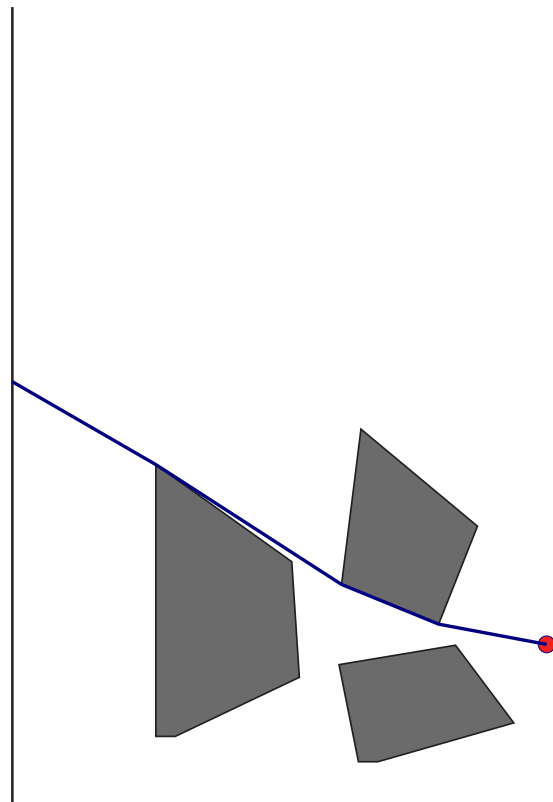


Abbildung 7: Die optimale Route für das Beispiel 2

Ausgabe für Beispiel 2

```

1 Lisa startet um 07:28:09 und erreicht den Bus um 07:31:00.
2 Sie trifft bei der y-Koordinate 500.17 auf den Bus.
3 Die Route dauert 2.85 Minuten und ist 712.62 Meter lang.
4 Die Route besteht aus folgenden Punkten:
5 633.0 189.0 L
6 505.0 213.0 P1
7 390.0 260.0 P1
8 170.0 402.0 P3
9 0.0 500.17 Y

```

3.3 Beispiel 3

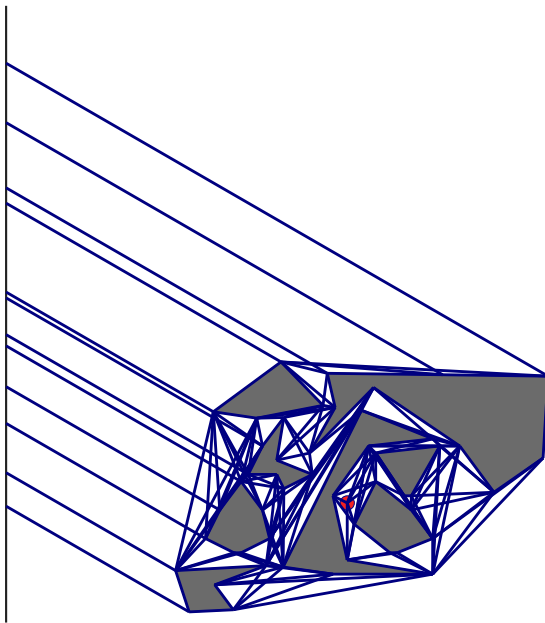


Abbildung 8: Der Sichtbarkeitsgraph für das Beispiel 3

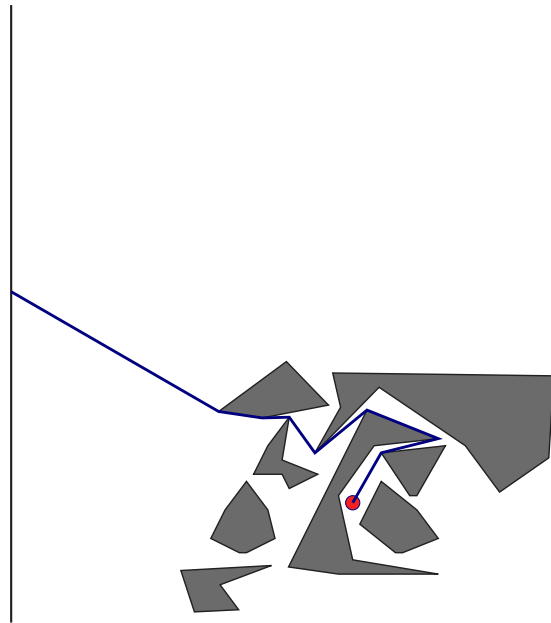


Abbildung 9: Die optimale Route für das Beispiel 3

Ausgabe für Beispiel 3

```

1 Lisa startet um 07:27:29 und erreicht den Bus um 07:30:56.
2 Sie trifft bei der y-Koordinate 464.04 auf den Bus.
3 Die Route dauert 3.45 Minuten und ist 862.60 Meter lang.
4 Die Route besteht aus folgenden Punkten:
5 479.0 168.0 L
6 519.0 238.0 P2
7 599.0 258.0 P3
8 499.0 298.0 P3
9 426.0 238.0 P8
10 390.0 288.0 P5
11 352.0 287.0 P6
12 291.0 296.0 P6
13 0.0 464.04 Y

```

3.4 Beispiel 4

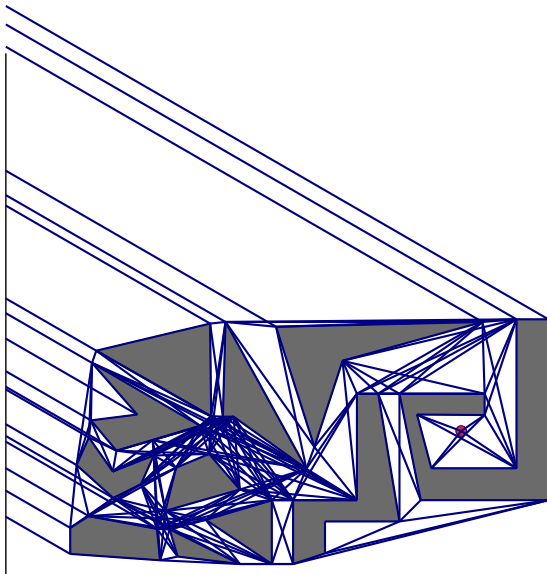


Abbildung 10: Der Sichtbarkeitsgraph für das Beispiel 4

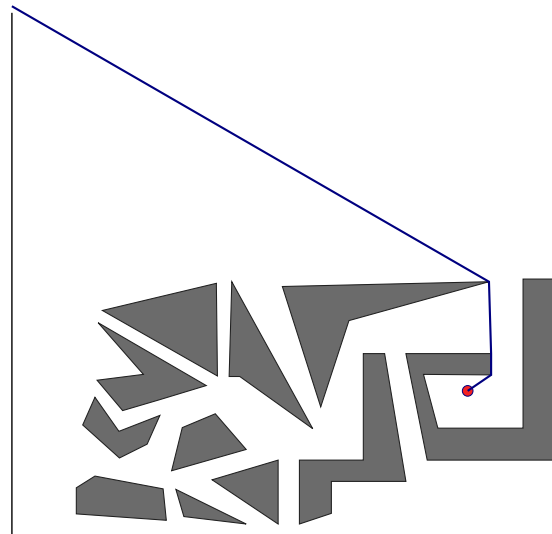


Abbildung 11: Die optimale Route für das Beispiel 4

Ausgabe für Beispiel 4

```

1 Lisa startet um 07:26:56 und erreicht den Bus um 07:31:59.
2 Sie trifft bei der y-Koordinate 992.39 auf den Bus.
3 Die Route dauert 5.05 Minuten und ist 1262.97 Meter lang.
4 Die Route besteht aus folgenden Punkten:
5 856.0 270.0 L
6 900.0 300.0 P11
7 900.0 340.0 P11
8 896.0 475.0 P10
9 0.0 992.39 Y

```

3.5 Beispiel 5

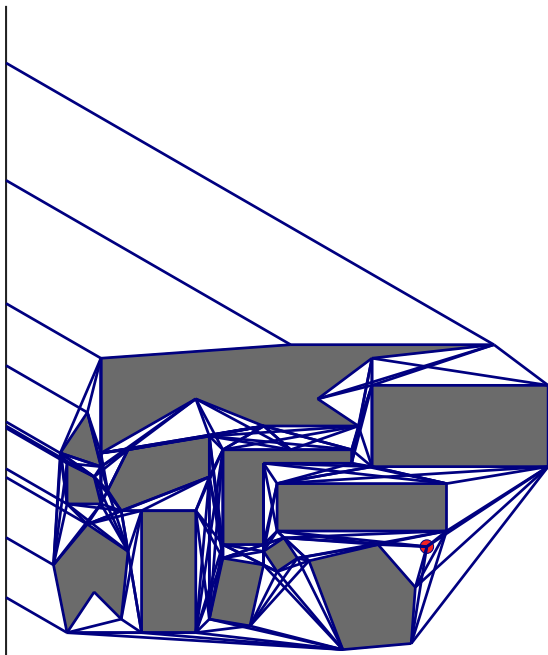


Abbildung 12: Der Sichtbarkeitsgraph für das Beispiel 5

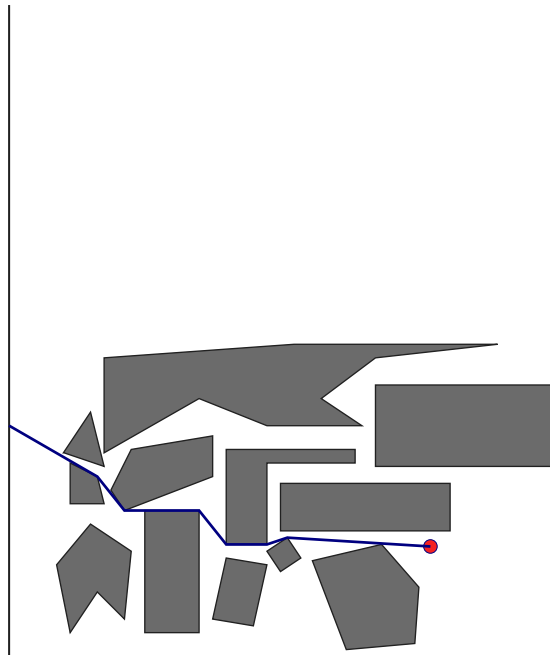


Abbildung 13: Die optimale Route für das Beispiel 5

Ausgabe für Beispiel 5

```

1 Lisa startet um 07:27:55 und erreicht den Bus um 07:30:41.
2 Sie trifft bei der y-Koordinate 340.07 auf den Bus.
3 Die Route dauert 2.76 Minuten und ist 691.20 Meter lang.
4 Die Route besteht aus folgenden Punkten:
5 621.0 162.0 L
6 410.0 175.0 P8
7 380.0 165.0 P3
8 320.0 165.0 P3
9 280.0 215.0 P5
10 170.0 215.0 P6
11 130.0 265.0 P9
12 0.0 340.07 Y

```

Eigenes Beispiel

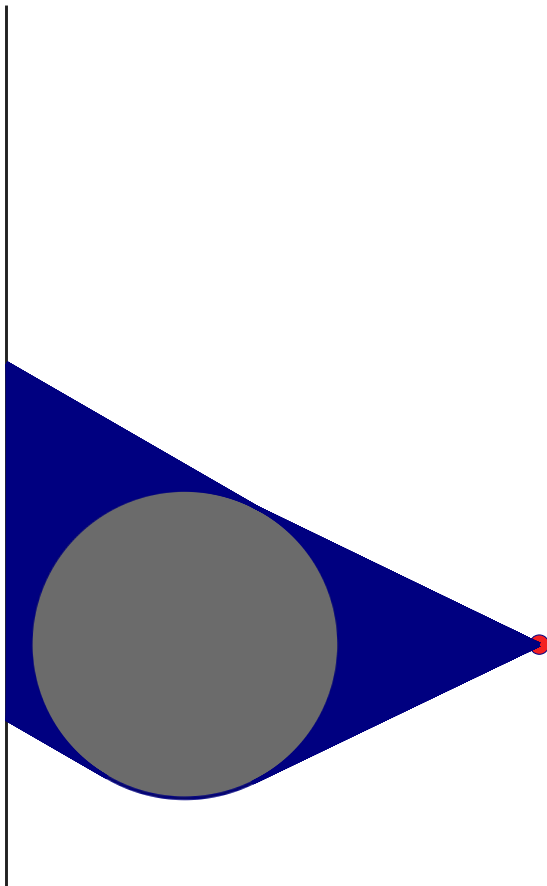


Abbildung 14: Der Sichtbarkeitsgraph für eigenes Beispiel

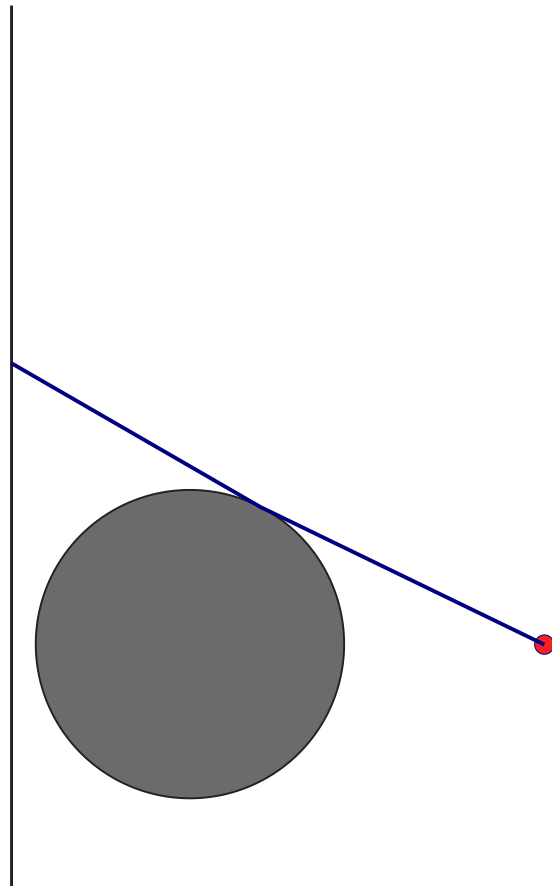


Abbildung 15: Die optimale Route für eigenes Beispiel

Ausgabe für eigenes Beispiel

```

1 Lisa startet um 07:28:36 und erreicht den Bus um 07:31:05.
2 Sie trifft bei der y-Koordinate 540.4 auf den Bus.
3 Die Route dauert 2.49 Minuten und ist 622.36 Meter lang.
4 Die Route besteht aus folgenden Punkten:
5 550.0 250.0 L
6 256.855 392.081 P1
7 0.0 540.4 Y

```

Bei diesen Beispielen 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

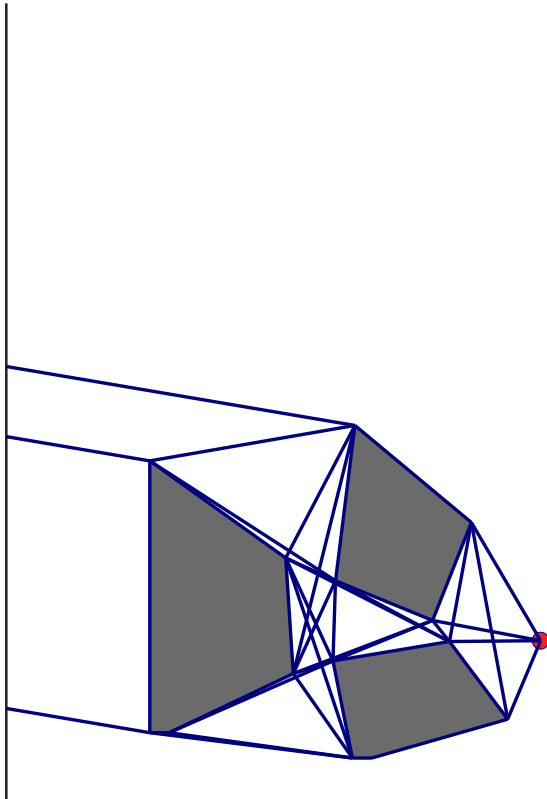


Abbildung 16: Der Sichtbarkeitsgraph

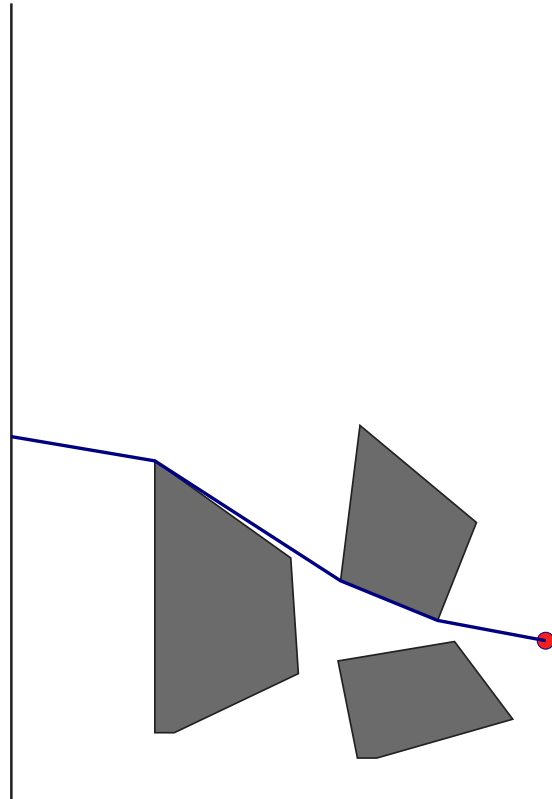


Abbildung 17: Die optimale Route

Ausgabe für das Beispiel

```

1 Lisa startet um 07:22:36 und erreicht den Bus um 07:30:52.
2 Sie trifft bei der y-Koordinate 430.74 auf den Bus.
3 Die Route dauert 8.26 Minuten und ist 688.72 Meter lang.
4 Die Route besteht aus folgenden Punkten:
5 633.0 189.0 L
6 505.0 213.0 P1
7 390.0 260.0 P1
8 170.0 402.0 P3
9 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.

4 Quellcode

```
1  #Imports
2  import argparse
3  import os
4
5  import pyvisgraph as vg
6  import svgggen
7  import minkowski
8
9  def ensure_dir(file_path):
10     directory = os.path.dirname(file_path)
11     if not os.path.exists(directory):
12         os.makedirs(directory)
13
14  def numtotime(num):
15     num = round(num)
16     hours = num // 3600
17     minutes = (num % 3600) // 60
18     seconds = (num % 3600) % 60
19     return hours, minutes, seconds
20
21  #Commandline-Argumente parsen
22  parser = argparse.ArgumentParser(description="Lösung zu Lisa rennt, Aufgabe 1,
23     ↳ Runde 2, 37. BwInf von Lukas Rost")
24
25  parser.add_argument('-i',
26     ↳ action="store",dest="input",default="lisarennt1.txt",help="Eingabedatei")
27  parser.add_argument('-o',action="store",dest="output",
28     ↳ default="lisarennt1_output.txt",help="Ausgabedatei")
29  parser.add_argument('-so', action="store",dest="svg",
30     ↳ default="lisarennt1_svg.svg",help="SVG-Ausgabedatei")
31  parser.add_argument('-d',action="store_true",default=False,dest="debug",
32     ↳ help="Debug-Ausgaben
33     ↳ aktivieren")
34  parser.add_argument('-vlisa',action="store",dest="velocity_lisa",default=15,
35     ↳ type=float,help="Erweiterung Geschwindigkeiten: Lisa in
36     ↳ km/h")
37  parser.add_argument('-vbus',action="store",dest="velocity_bus",default=30,
38     ↳ type=float,help="Erweiterung Geschwindigkeiten: Bus in
39     ↳ km/h")
40  parser.add_argument('-minkowski',action="store",default=None,help="Erweiterung
41     ↳ Minkowski-Summe: Eingabedatei (1 Polygon im gleichen Format wie in der
42     ↳ normalen Eingabe)")
43
44  args = parser.parse_args()
45
46  #Geschwindigkeiten in m/s umrechnen
47  real_v_lisa = round(args.velocity_lisa / 3.6 ,3)
48  real_v_bus = round(args.velocity_bus / 3.6 ,3)
49
50  #Maximale x und y für Darstellung
51  maxx = 0
52  maxy = 0
```

```
41
42 # Polygone einlesen
43 infile = open(args.input, 'r')
44 numpoly = int(infile.readline())
45 polylist = []
46
47 for i in range(numpoly):
48     pointlist = []
49     line = infile.readline().split(" ")
50     line = [float(x) for x in line]
51     index = 1
52     for j in range(int(line[0])):
53         maxx = max(maxx, line[index])
54         maxy = max(maxy, line[index+1])
55         pointlist.append(vg.Point(line[index], line[index+1], polygon_id=("P" +
56             ↪ str(i+1))))
57         index += 2
58     polylist.append(pointlist)
59
60 #Lisas Position einlesen
61 pos = infile.readline().split(" ")
62 pos = [float(x) for x in pos]
63 lisa = vg.Point(pos[0], pos[1], polygon_id="L")
64 infile.close()
65
66 maxx = max(maxx, pos[0])
67 maxy = max(maxy, pos[1])
68
69 #Erweiterung Minkowski-Summe
70 if args.minkowski is not None:
71     minfile = open(args.minkowski, 'r')
72     lisa_poly = []
73     line = minfile.readline().split(" ")
74     minfile.close()
75     line = [float(x) for x in line]
76     index = 1
77     for j in range(int(line[0])):
78         lisa_poly.append(vg.Point(-line[index], -line[index+1]))
79         index += 2
80     polylist = minkowski.minkowski_sum_list(polylist, lisa_poly)
81
82 #Graph erstellen und Algorithmus ausführen
83 graph = vg.VisGraph(real_v_lisa, real_v_bus, lisa)
84 graph.build(polylist)
85 path, mintime, min_bus_time, minpoint, dist_minpoint = graph.shortest_path()
86 mintime = abs(mintime)
87
88 #Debug-Ausgaben
89 if args.debug:
90     outpath = os.path.dirname(args.input) + "/out/debug/" +
91         ↪ os.path.basename(args.input).split(".")[0]
92     ensure_dir(outpath)
93     graph.save(outpath + "-pickle.pk1")
94     svgfile = open(outpath + "-visgraph.svg", "w")
```



```

93     svgfile.write(svggen.gen_vis_svg(graph.get_visgraph(),polylist,lisa,
    ↪     maxx+200,maxy+500))
94     svgfile.close()
95
96
97     #Ausgabe SVG
98     svgfile = open(args.svg,"w")
99     svgfile.write(svggen.gen_output_svg(path,polylist,lisa,maxx+200,maxy+500))
100    svgfile.close()
101
102    #Ausgabe Text
103    outtext = ""
104    hours, minutes, seconds = numtotime(mintime)
105    hours = 7 - hours
106    minutes = 30 - minutes
107    if seconds != 0:
108        minutes -= 1
109        seconds = 60 - seconds
110    bhours, bminutes, bseconds = numtotime(min_bus_time)
111    bhours = 7 + bhours
112    bminutes = 30 + bminutes
113    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)))
114    outtext += "Sie trifft bei der y-Koordinate {} auf den
    ↪     Bus.\n".format(minpoint.y)
115    outtext += "Die Route dauert {:.02f} Minuten und ist {:.02f} Meter
    ↪     lang.\n".format(dist_minpoint/(real_v_lisa*60),dist_minpoint)
116    outtext += "Die Route besteht aus folgenden Punkten:\n"
117    for point in path:
118        outtext += "{} {} {} \n".format(point.x,point.y,point.real_polygon_id)
119
120    outfile = open(args.output,"w")
121    outfile.write(outtext)
122    outfile.close()

```

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

```

1  def _vis_graph(self, points):
2      """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. """
3      visible_edges = []
4      for p1 in points:
5          dest_point_y = p1.y + (self.vlisa / self.vbus) * (1 / math.sqrt(1 -
    ↪     ((self.vlisa * self.vlisa) / (self.vbus * self.vbus)))) * p1.x
6          dest_point = Point(0,round(dest_point_y,2), polygon_id="Y")
7          for p2 in visible_vertices(p1, self.graph,
    ↪     origin=self.lisa_point,destination=dest_point):
8              visible_edges.append(Edge(p1, p2))
9      return visible_edges

```

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

```

1  from __future__ import division
2  from math import pi, sqrt, atan, acos
3  from pyvisgraph.graph import Point
4
5  INF = 10000
6  # Toleranz für Gleitkommazahlen
7  COLIN_TOLERANCE = 10
8  T = 10**COLIN_TOLERANCE
9  T2 = 10.0**COLIN_TOLERANCE
10
11 def visible_vertices(point, graph, origin=None, destination=None,
12     ↪ scan='full'):
13     """Gibt Liste der von point sichtbaren Punkte zurück. origin und
14     ↪ destination werden dabei ebenfalls mit einbezogen. scan='full'
15     ↪ überprüft die Sichtbarkeit aller Punkte und wird hier ausschließlich
16     ↪ verwendet.
17     """
18
19     edges = graph.get_edges()
20     points = graph.get_points()
21     if origin: points.append(origin)
22     if destination: points.append(destination)
23     points.sort(key=lambda p: (angle(point, p), edge_distance(point, p)))
24
25     # open_edges initialisieren (alle Kanten, die die anfaengliche Sweep line
26     ↪ schneiden)
27     open_edges = []
28     point_inf = Point(INF, point.y)
29     for e in edges:
30         if point in e: continue
31         if edge_intersect(point, point_inf, e):
32             if on_segment(point, e.p1, point_inf): continue
33             if on_segment(point, e.p2, point_inf): continue
34             k = EdgeKey(point, point_inf, e)
35             insert(open_edges, k)
36
37     visible = []
38     prev = None
39     prev_visible = None
40     for p in points:
41         if p == point: continue
42         if scan == 'half' and angle(point, p) > pi: break
43
44         #Inzidente Kanten, die in Uhrzeigerrichtung liegen, entfernen
45         if open_edges:
46             for edge in graph[p]:
47                 if ccw(point, p, edge.get_adjacent(p)) == -1:
48                     k = EdgeKey(point, p, edge)
49                     index = bisect(open_edges, k) - 1
50                     if len(open_edges) > 0 and open_edges[index] == k:
51                         del open_edges[index]
52
53         # Sichtbarkeit von p von point aus überprüfen
54         is_visible = False

```

```

49     # vorheriger Punkt (prev) nicht auf der Geraden point-p
50     if prev is None or ccw(point, prev, p) != 0 or not on_segment(point,
    ↪ prev, p):
51         #keine offenen Kanten -> sichtbar
52         if len(open_edges) == 0:
53             is_visible = True
54         #Strecke schneidet erste Kante nicht -> sichtbar
55         elif not edge_intersect(point, p, open_edges[0].edge):
56             is_visible = True
57     # ...liegen auf einer Geraden *und* vorheriger Punkt war nicht
    ↪ sichtbar -> nicht sichtbar
58     elif not prev_visible:
59         is_visible = False
60     # ...liegen auf einer Geraden *und* vorheriger Punkt war sichtbar ->
    ↪ alle offenen Kanten überprüfen
61     else:
62         is_visible = True
63         for e in open_edges:
64             if prev not in e.edge and edge_intersect(prev, p, e.edge):
65                 is_visible = False
66                 break
67         if is_visible and edge_in_polygon(prev, p, graph):
68             is_visible = False
69
70     # Strecke liegt komplett in einem Polygon -> nicht sichtbar
71     if is_visible and p not in graph.get_adjacent_points(point):
72         is_visible = not edge_in_polygon(point, p, graph)
73
74     if is_visible: visible.append(p)
75
76     # Inzidente Kanten, die gegen Uhrzeigerrichtung liegen, hinzufügen
77     for edge in graph[p]:
78         if (point not in edge) and ccw(point, p, edge.get_adjacent(p)) ==
    ↪ 1:
79         k = EdgeKey(point, p, edge)
80         insort(open_edges, k)
81
82     prev = p
83     prev_visible = is_visible
84     return visible
85
86
87 def polygon_crossing(p1, poly_edges):
88     """Überprüft, ob Punkt in Polygon liegt und verwendet dazu die
    ↪ Strahl-Methode
    ↪ (https://de.wikipedia.org/wiki/Punkt-in-Polygon-Test_nach_Jordan). """
89     p2 = Point(INF, p1.y)
90     intersect_count = 0
91     co_flag = False
92     co_dir = 0
93     for edge in poly_edges:
94         if p1.y < edge.p1.y and p1.y < edge.p2.y: continue
95         if p1.y > edge.p1.y and p1.y > edge.p2.y: continue
96         co0 = (ccw(p1, edge.p1, p2) == 0) and (edge.p1.x > p1.x)

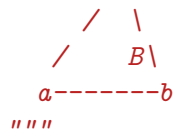
```

```
97     co1 = (ccw(p1, edge.p2, p2) == 0) and (edge.p2.x > p1.x)
98     if co0 and co1: continue
99     co_point = edge.p1 if co0 else edge.p2
100    if co0 or co1:
101        if edge.get_adjacent(co_point).y > p1.y:
102            co_dir += 1
103        else:
104            co_dir -= 1
105        if co_flag:
106            if co_dir == 0:
107                intersect_count += 1
108            co_flag = False
109            co_dir = 0
110        else:
111            co_flag = True
112    elif edge.intersect(p1, p2, edge):
113        intersect_count += 1
114    if intersect_count % 2 == 0:
115        return False
116    return True
117
118
119    def edge_in_polygon(p1, p2, graph):
120        """Überprüft, ob Kante zwischen zwei Polygon-Punkten innerhalb eines
121        ↪ Polygons liegt. """
122        if p1.polygon_id != p2.polygon_id:
123            return False
124        if p1.polygon_id == -1 or p2.polygon_id == -1:
125            return False
126        mid_point = Point((p1.x + p2.x) / 2, (p1.y + p2.y) / 2)
127        return polygon_crossing(mid_point, graph.polygons[p1.polygon_id])
128
129    def point_in_polygon(p, graph):
130        """Überprüft, ob Punkt innerhalb irgendeines Polygons liegt. """
131        for polygon in graph.polygons:
132            if polygon_crossing(p, graph.polygons[polygon]):
133                return polygon
134        return -1
135
136    def edge_distance(p1, p2):
137        """Euklidische Distanz zwischen zwei Punkten."""
138        return sqrt((p2.x - p1.x)**2 + (p2.y - p1.y)**2)
139
140
141    def intersect_point(p1, p2, edge):
142        """Schnittpunkt von p1p2 und edge."""
143        if p1 in edge: return p1
144        if p2 in edge: return p2
145        if edge.p1.x == edge.p2.x:
146            if p1.x == p2.x:
147                return None
148            pslope = (p1.y - p2.y) / (p1.x - p2.x)
149            intersect_x = edge.p1.x
```

```

150         intersect_y = pslope * (intersect_x - p1.x) + p1.y
151         return Point(intersect_x, intersect_y)
152
153     if p1.x == p2.x:
154         eslope = (edge.p1.y - edge.p2.y) / (edge.p1.x - edge.p2.x)
155         intersect_x = p1.x
156         intersect_y = eslope * (intersect_x - edge.p1.x) + edge.p1.y
157         return Point(intersect_x, intersect_y)
158
159     pslope = (p1.y - p2.y) / (p1.x - p2.x)
160     eslope = (edge.p1.y - edge.p2.y) / (edge.p1.x - edge.p2.x)
161     if eslope == pslope:
162         return None
163     intersect_x = (eslope * edge.p1.x - pslope * p1.x + p1.y - edge.p1.y) /
164     ↪ (eslope - pslope)
165     intersect_y = eslope * (intersect_x - edge.p1.x) + edge.p1.y
166     return Point(intersect_x, intersect_y)
167
168 def point_edge_distance(p1, p2, edge):
169     """Distanz von p1 bis zum Schnittpunkt von p1p2 mit edge."""
170     ip = intersect_point(p1, p2, edge)
171     if ip is not None:
172         return edge_distance(p1, ip)
173     return 0
174
175
176 def angle(center, point):
177     """Winkel zwischen Gerade cp und positiver x-Achse.
178     -----p
179     | /
180     | /
181     c/a/
182     """
183     dx = point.x - center.x
184     dy = point.y - center.y
185     if dx == 0:
186         if dy < 0:
187             return pi * 3 / 2
188         return pi / 2
189     if dy == 0:
190         if dx < 0:
191             return pi
192         return 0
193     if dx < 0:
194         return pi + atan(dy / dx)
195     if dy < 0:
196         return 2 * pi + atan(dy / dx)
197     return atan(dy / dx)
198
199
200 def angle2(point_a, point_b, point_c):
201     """Implementierung des Cosinussatzes.
202     c

```



```

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):
    """Ueberprueft, 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

```

```
254     if o4 == 0 and on_segment(p2, q1, q2):
255         return True
256     return False
257
258
259 def insort(a, x):
260     """Einfügen in den binären Suchbaum. """
261     lo = 0
262     hi = len(a)
263     while lo < hi:
264         mid = (lo+hi)//2
265         if x < a[mid]: hi = mid
266         else: lo = mid+1
267     a.insert(lo, x)
268
269
270 def bisect(a, x):
271     """Binäre Suche für den Suchbaum. """
272     lo = 0
273     hi = len(a)
274     while lo < hi:
275         mid = (lo+hi)//2
276         if x < a[mid]: hi = mid
277         else: lo = mid+1
278     return lo
279
280
281 class EdgeKey(object):
282     """Element des Suchbaums. """
283     def __init__(self, p1, p2, edge):
284         self.p1 = p1
285         self.p2 = p2
286         self.edge = edge
287
288     def __eq__(self, other):
289         if self.edge == other.edge:
290             return True
291
292     def __lt__(self, other):
293         if self.edge == other.edge:
294             return False
295         if not edge_intersect(self.p1, self.p2, other.edge):
296             return True
297         self_dist = point_edge_distance(self.p1, self.p2, self.edge)
298         other_dist = point_edge_distance(self.p1, self.p2, other.edge)
299         if self_dist > other_dist:
300             return False
301         if self_dist < other_dist:
302             return True
303         # If the distance is equal, we need to compare on the edge angles.
304         if self_dist == other_dist:
305             if self.edge.p1 in other.edge:
306                 same_point = self.edge.p1
307             elif self.edge.p2 in other.edge:
```

```

308         same_point = self.edge.p2
309         aslf = angle2(self.p1, self.p2,
310             ↪ self.edge.get_adjacent(same_point))
311         aot = angle2(self.p1, self.p2,
312             ↪ other.edge.get_adjacent(same_point))
313         if aslf < aot:
314             return True
315         return False
316
317     def __repr__(self):
318         reprstring = (self.__class__.__name__, self.edge, self.p1, self.p2)
319         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.

```

1  #Implementierung des Dijkstra-Algorithmus
2  #mit einem Priority-Dict (aehnlich Priority-Queue) umgesetzt
3  def dijkstra(graph, origin):
4      D = {} #Distanz
5      P = {} #Elternknoten
6      Q = priority_dict()
7      Q[origin] = 0
8
9      for v in Q:
10         D[v] = Q[v]
11
12         edges = graph[v]
13         for e in edges:
14             w = e.get_adjacent(v)
15             elength = D[v] + edge_distance(v, w)
16             if w in D:
17                 if elength < D[w]:
18                     raise ValueError
19             elif w not in Q or elength < Q[w]:
20                 Q[w] = elength
21                 P[w] = v
22         return (D, P)
23
24  #Kuerzesten Weg vom Startpunkt zur y-Achse bestimmen
25  #Idee ist in Kapitel 1.5 beschrieben
26  def shortest_path(graph, origin, vlisa, vbus):
27      #Dijkstra für alle Knoten ausführen
28      D, P = dijkstra(graph, origin)
29      mintime = -math.inf
30      minpoint = Point(0,0)
31      min_bus_time = 0.0
32      #Besten Companion-Punkt bestimmen...
33      for point in graph.get_points():
34          if point.x == 0:
35              bus_time = point.y / vbus
36              lisa_time = D[point] / vlisa
37              #... mithilfe der spätesten Startzeit

```



```

38         #Zeitangabe: negativ vor 7.30, positiv danach
39         total_time = bus_time - lisa_time
40         if total_time > mintime:
41             mintime = round(total_time,2)
42             minpoint = point
43             min_bus_time = round(bus_time,2)
44         #Pfad zum Punkt finden und zurückgeben
45         path = []
46         destination = minpoint
47         while 1:
48             path.append(destination)
49             if destination == origin: break
50             destination = P[destination]
51         path.reverse()
52         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*

```

1  import math
2  from pyvisgraph.graph import Point
3
4  #Minkowski-Summe auf alle Polygone anwenden
5  #vorher Vorverarbeitung, damit die Eingaben korrekt sind
6  def minkowski_sum_list(polygons, lisa_poly):
7      lisa_poly = findsmallest(makeccw(lisa_poly))
8      polygons = [findsmallest(makeccw(poly)) for poly in polygons]
9      polygons = [minkowski_sum(poly,lisa_poly) for poly in polygons]
10     return polygons
11
12     #Algorithmus für die Minkowski-Summe wie in der Dokumentation beschrieben
13     #Das ist die gleiche scheiß Implementierung wie in "Computational Geometry"
14     #und das ist das Standardwerk dazu und es funktioniert trotzdem nicht
15     def minkowski_sum(v,w):
16         i = 0
17         j = 0
18         n = len(v)
19         m = len(w)
20         v.append(v[0])
21         v.append(v[1])
22         w.append(w[0])
23         w.append(w[1])
24         newpoly = []
25         while True:
26             newpoly.append(Point(v[i].x + w[j].x, v[i].y + w[j].y,
27                                 ↪ polygon_id=v[i].polygon_id))
28             if angle(v[i],v[i+1]) < angle(w[j],w[j+1]):
29                 i = i+1
30             elif angle(v[i],v[i+1]) > angle(w[j],w[j+1]):
31                 j = j+1
32             else:
33                 i = i+1
34                 j = j+1
35         if i == n and j == m:

```

```

35         break
36     return newpoly
37
38     #Winkel zwischen Vektor ab und positiver x-Achse
39     #Ansatz siehe https://math.stackexchange.
40     ↪ com/questions/1910825/how-do-i-find-the-angle-a-vector-makes-to-the-x-axis
41     def angle(a, b):
42         vx = b.x - a.x
43         vy = b.y - a.y
44         angle = math.atan2(vy, vx)
45         if vy < 0:
46             angle += 2 * math.pi
47         return angle
48
49     #Polygon-Punkte in Richtung gegen den Uhrzeigersinn bringen
50     def makeccw(polygon):
51         if not isccw(polygon):
52             polygon.reverse()
53         return polygon
54
55     #Berechnung, ob ein Polygon gegen den Uhrzeigersinn (CCW) orientiert ist
56     #Ansatz: https://de.wikipedia.org/wiki/Gau%C3%9Fsche_Trapezformel
57     def isccw(polygon):
58         sum = 0
59         length = len(polygon)
60         for i in range(len(polygon)):
61             sum += (polygon[(i+1) % length].x - polygon[i].x) * (polygon[(i+1) %
62                 ↪ length].y + polygon[i].y)
63         return sum < 0
64
65     #finde den Punkt mit der kleinsten y-Koordinate und setze ihn als ersten Punkt
66     ↪ im Polygon
67     def findsmallest(polygon):
68         index = 0
69         miny = math.inf
70         minx = math.inf
71         for i, point in enumerate(polygon):
72             if point.y < miny or (point.y == miny and point.x < minx):
73                 minx = point.x
74                 miny = point.y
75                 index = i
76
77     #schiebe Punkte davor ans Ende
78     for i in range(index):
79         polygon.append(polygon[i])
80     polygon = polygon[index:]
81     return polygon

```

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