Aufgabe 1: Lisa Rennt

Team-ID: 49111

Team-Name: HarperCreekFürHenning187

Bearbeiter/-innen dieser Aufgabe:
Julius Carl Ide

28. April 2019

Inhaltsverzeichnis

1 Lösungsidee

Graph der Polygone Lisa's Ziel ist es, den Bus möglichst schnell zu erreichen. Dazu eignet es sich, einen Graphen aufzustellen, um anschließend den kürzesten Pfad zu finden. Nun ist die Frage, welche Punkte als Knoten für diesen Graphen geeignet sind. Die optimalen Knoten sind die Ecken der Polygone. Angenommen, dass eine direkte Strecke zwischen den Punkten A und C aufgrund des Hindernisses $BFGH$ nicht möglich ist (Abbildung 1). In diesem Fall ist der Pfad $C - B - A$ der kürzeste, um von B A zu erreichen. da es keinen Punkt D gibt, der einen kürzeren Pfad $C - D - A$ darstellen würde. Dies ist der Fall, da der Winkel $\angle ABC$ größer als 180 Grad ist.

$$\overline{BC} + \overline{BA} \leq \overline{DC} + \overline{DA} \tag{1}$$

Dies kann bewiesen werden, indem ein Kreis mit dem Radius $\overline{AB}$ um den Punkt A gezogen wird. Sollte es einen Punkt D geben, der einen kürzeren Pfad darstellen würde, wäre die Summe der Strecken $\overline{ED} + \overline{DC}$ kürzer als die Strecke $\overline{CB}$, da die Strecke $\overline{AE}$ aufgrund des Kreises die gleiche Länge wie $\overline{AB}$ aufweist. Dies ist nicht der Fall, da die Strecke $\overline{EC}$ kleiner als die Summe der Strecken $\overline{ED} + \overline{DC}$ ist, weil für jede Kante a eines Dreieckes $a \leq b + c$ gelten muss, wobei b und c die beiden anderen Kanten sind. Außerdem ist die Strecke $\overline{CB}$ kürzer als die Strecke $\overline{EC}$, wie sich aus Abbildung 1 entnehmen lässt.

$$\overline{BC} \leq \overline{EC} \leq \overline{ED} + \overline{DC} \tag{2}$$

Daraus lässt sich nun ein kantengewichteter gerichteter Graph $G = (V, E)$ mit den Knoten V und den Kanten E erstellen. V setzt sich folgender Maßen zusammen:

$$V = \bigcup_{i=1}^{n} P_i \tag{3}$$

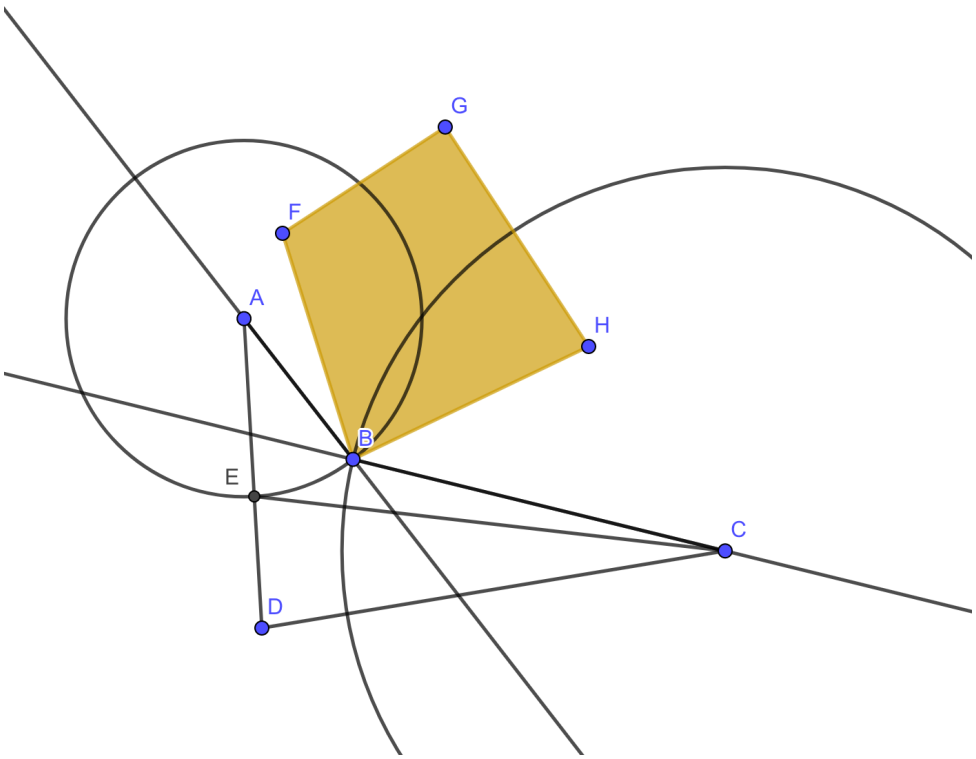


Abbildung 1

Hierbei ist n die Anzahl an Häusern in Lisas Weg und $P_i = \{e_1, e_2, e_3, ...\}$ entspricht dem i-ten Haus (Polygon), welches sich aus seinen Ecken $\{e_1, e_2, e_3, ...\}$ zusammensetzt. Jede Ecke $e \in P_i$ ist ein geordnetes Paar (x, y), wobei x und y Koordinaten sind. Anschließend müssen noch alle Kanten E gefunden werden. Dazu werden als erstes alle möglichen Kanten summiert. Eine Kante sei als georndetes Paar $e = (u, v)$ definiert, wobei diese bei dem Knoten u beginnt und mit dem Knoten v aufhört.

$$E = \sum_{i=1}^{n} \sum_{j=1}^{n} e(V_i, V_j) \tag{4}$$

Gleichung 4 summiert alle möglichen Kanten. n repräsentiert die Anzahl an Knoten. Nun muss für jede Kante $e_i \in E$ überprüft werden, ob sie irgendwelche Polygone (Häuser) schneidet. Dazu werden die Kanten aller Polygone aufsummiert, wobei P die Menge aller Polygone ist und das Polygon P_i aus den Ecken $\{e_1, e_2, ...e_{P_i}\}$ zusammengesetzt ist.

$$E_{polygone} = \sum_{i=1}^{|P|} \sum_{j=1}^{|P_i|} (e_j, e_{j+1}) \tag{5}$$

Eine Kante $e = (u, v)$ schneidet eine andere Kante $f = (s, t)$, wenn folgende Bedingungen gelten: Zuerst werden die Funktionsvorschriften für beide Kanten ermittelt ($y = m \cdot x + c$):

$$e(x) = m_e \cdot x + \frac{v_y}{m_e \cdot v_x} \tag{6}$$

$$f(x) = m_f \cdot x + \frac{t_y}{m_f \cdot t_x} \tag{7}$$

Darauffolgend muss der Schnittpunkt ermittelt werden:

$$x = \frac{\frac{v_y}{m_f \cdot v_x} - \frac{t_y}{m_e \cdot t_x}}{m_e - m_f} \tag{8}$$

Nun kann es jedoch vorkommen, dass sich die beiden Kanten schneiden würden, da sie nur teilweise definiert sind. Deshalb muss zuletzt noch überprüft werden, ob sich der Schnittpunkt im richtigen Intervall befindet:

$$u_x \leq x \leq v_x \cap s_x \leq x \leq t_x \tag{9}$$

Daraus ergibt sich nun ein Graph wie in Abbildung 2:

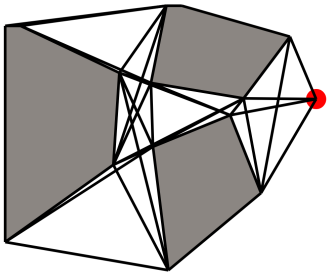


Abbildung 2

Schnittpunkte mit der y-Achse Anschließend ist zu überlegen, in welchem Winkel die Straße angelaufen wird. Lisa's oberste Maxime ist es, das Haus möglichst spät zu verlassen. Dementsprechend ist es weniger wichtig, wie lange sie laufen muss.

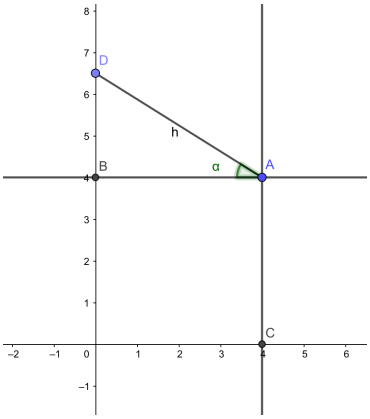


Abbildung 3

Angenommen, dass Punkt A (Abbildung 3) mit dem Punkt D eine direkte Strecke mit der y-Achse bildet (keine Hindernisse zwischen den Punkten). Nun stelt sich die Frage, welchen Wert der Winkel α annehmen sollte. Dementsprechend gilt folgende Gleichung:

$$T_{Lisa} = \frac{h + AC}{15\frac{km}{h}} \tag{10}$$

Dies kann noch vereinfacht werden, indem h durch AB dargestellt wird:

$$h = \frac{BA}{\cos\alpha} \tag{11}$$

Nun kann dies Auch für den Bus aufgestellt werden:

$$T_{Bus} = \frac{\frac{BA}{\cos\alpha} \cdot \sin\alpha + AC}{30\frac{km}{h}} \tag{12}$$

Der optimale Winkel kann gefunden werden, indem die beiden Gleichungen gleichgesetzt werden ($T_{Lisa} = T_{Bus}$):

$$\frac{\frac{BA}{\cos\alpha} + AC}{15\frac{km}{h}} = \frac{\frac{BA}{\cos\alpha} \cdot \sin\alpha + AC}{30\frac{km}{h}} \tag{13}$$

Vereinfacht entspricht das:

$$\tan\alpha - 2 \cdot \sec\alpha = \frac{AC}{BC} \tag{14}$$

So kann der optimale Winkel für jeden Punkt berechnet werden. Dies wird dementensprechend auch für jeden Punkt getan. Sollte so eine Kante $e \in E, e = (p, q)$, wobei q der Schnitpunkt mit der y-Achse ist und p den Anfangspunkt repräsentiert, ohne das Schneiden anderer Polygone möglich sein, wird diese zu der Menge E der Kanten hinzugefügt. Nun kann ein vollständiger Graph visuell dargestellt werden:

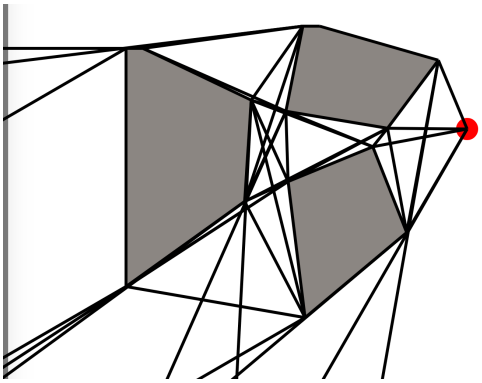


Abbildung 4: Vollständiger Graph

Um den kürzesten Pfad zu finden, eignet es sich Dijkstra's-Algorithmus zu verwenden, da es ein positiver kantengewichteter Graph ist. Dijkstras-Algorithmus funktioniert folgendermaßen: Zuerst werden allen Knoten V die Eigenschaften Distanz (in diesem Fall die Distanz in m) und Vorgänger zugewiesen. Anschließend wird die Distanz zum Startknoten (Lisas Haus) mit 0 initialisiert, während die Distanz zu allen anderen Knoten als ∞ abgespeichert wird. Solange es noch unbesuchte Knoten gibt, wählt der Algorithmus den Knoten k mit der kürzesten Distanz aus und speichert, dass dieser besucht wurde. Darauffolgend werden die Distanzen zu allen Nachbarknoten $\{n_1, n_2, ..., n_i\}$ berechnet, indem die Distanz vom Startknoten zum unbesuchten Knoten k und das Kantengewicht der Kante $(k, n_i$ addiert werden. Sollte diese Dsitanz kürzer als die bereits gespeichert sein, wird der Distanz Wert aktualiesiert und k als Vorgänger gespeichert. So kann der kürzeste Pfad zwischen zwei Knoten gefunden werden. In einigen Spezialfällen kann es jedoch vorkommen, dass der kürzeste Weg (in m) nicht unbedingt den spätesten Zeitpunkt für das Verlassen des Hauses bedeutet, was Lisas oberste Priorität ist. Deswegen muss für jeden möglichen Punkt, an dem Lisa den Bus betreten kann, der Zeitpunkt des Verlassen des Hauses berechnet werden.

$$T_{Lisa} = \frac{D_{gelaufeneDistanz}}{15 \cdot \frac{km}{h}} \tag{15}$$

Die Zeit, die der Bus braucht, um zu Lisas selbst ausgesuchter Haltestelle zu gelangen, setzt sich einfach nur aus der y-Koordinate von Lisas Haltestelle zusammen.

$$T_{Bus} = \frac{D_y}{30 \cdot \frac{km}{h}} \tag{16}$$

Um den Zeitpunkt des Verlassen des Hauses zu berechnen, muss nun nur noch der Zeitpunkt, an dem Lisa den Bus erreicht, von dem Zeitpunkt, an dem der Bus die Haltestelle erreicht, subtrahiert werden. Nun kann es jedoch vorkommen, dass der kürzeste Weg (in m) nicht unbedingt den spätesten Zeitpunkt des Verlassen des Hause garantiert. Aus der Aufgabenstellung wird jedoch klar, dass dies Lisas oberste Priorät ist. Dementsprechend muss iterativ für jeden Knoten $v \in V, v = (x, y)$, dessen x-Koordinate 0 ist, der Zeitpunkt der Verlassen des Hauses für Lisa berechnet werden. Nachdem so die effizientesete Route gefunden wurde, kann mit Gewissheit gesagt werden, dass dies der späteste mögliche Zeitpunkt für Lisa ist, an dem sie das Haus verlassen kann.

$$T_{departure} = 7:30 + T_{Bus} - T_{Lisa} \tag{17}$$

Dieser gesamte Ablauf kann in den folgendes Pseudocode zusammengefasst werden:

Algorithm 1 Pseudocode

```
edges = findAllPossibleEdges()      ▷ zuerst werden alle möglichen Kanten für den Graphen berechnet
validEdges = reduceToValidEdge(edges)      ▷ für jede Kante wird überprüft, ob sie irgendwelche Polygone schneidet
finalEdges = intersectionsWithYaxis()      ▷ für jeden Punkt wird ein möglicher Schnittpunkt mit der y-Achse gesucht
graph = Graph(validEdges, finalEdges)
for all f ∈ finalEdges do
    time = timeToLeaveTheHouse(f)
end for
                                    ▷ aus allen Endpunkten, wird die günstigste Route gefunden
```

2 Angaben zur Laufzeitanalyse

Eine Zeitabschätzung darf natürlich nicht vernachlässigt werden. Zuerst müssen alle möglichen Kanten gefunden werden. Dazu werden alle möglichen Kombinationen aus 2 Punkten aus der Menge V aller Knoten/Ecken der Polygone. Reduziert entspricht das einer Laufzeit von $O(V^2)$ in Big-O Notation. Nun muss für jede Kante (V^2) überprüft werden, ob sie keine Polygone (V) schneidet. Da ein n-eckiges Polygon n Kanten hat, gilt $O(V^3)$. Nun müssen noch alle Schnittpunkte mit der y-Achse gefunden werden, dies ist polymial mit $O(V^3)$, da für jeden Knoten der optimale Winkel gefunden werden muss, um anschließend sicher zu stellen, dass die neue Kante keine anderen Polygone schneidet. Der aufwändigste und dominierende Teil des ganzen Algorithmus ist Dijkstra mit $0(|E| + |V| \cdot log(|V|))$.

3 Umsetzung

Programmstruktur Wichtige Klassen, die zur Lösung der Aufgaben dienen:

- draw.py Diese Datei liest die fertigen Koordinaten für Lisas Weg zum Bus ein und generiert ein visuelles Ouput mit Hilfe von .svg.
- finalPath.py Der Graph, Lisas Haus und alle Knoten des Graphen, die auf der y-Achse liegen, um für jeden Knoten den Zeitpunkt zu berechnen, an dem Lisa ihr Haus verlassen muss, damit der Pfad mit dem spätesten Zeitpunkt gefunden werden kann.
- findIntersection.py Der Kern dieser Klasse besteht darin zu berechnen, ob sich 2 Kanten schneiden. Sie wird dazu benutzt, um die Kanten des Graphen zu finden.
- graph.py Diese Datei erstellt einen Graphen und erwartet die Kanten als Input. Darauffolgend kann dann die kürzeste Strecke zwischen zwei beliebigen Punkten mithilfe von Dijkstra-Algorithmus gefunden werden.
- textToPolygones.py Dieses Programm erwartet eine txt-Datei als Input, um die Koordinaten von Lisas Haus und von allen Hindernissen als Array zu extrahieren.

Diese ganzen Dateien werden von der Datei *lisasRoute.py* gesteuert.

3.1 Die Klasse LisasRoute

Einlesen der Dreiecke Zur Lösung dieser Aufgabe wurde Python verwendent. Zuerst wird die Beispiel-Textdatei eingelesen. Die Datei *textToPolygones.py* extrahiert die wichtigen Werte in gibt einen Array mit allen Hindernissen, dabei ist jedes Element ein Array im Format (x, y) und jede Ecke eines Hindernisses wird durch so ein geordnetes Paar repräsentiert, und den Koordinaten von Lisas Haus. (jjiji)

Alle möglichen Kanten Nachdem alle Koordinaten bekannt sind, ist es an der Reihe den Graphen, der in Abbildung 2 zu sehen ist, zu berechnen. Dazu werden als erstes alle möglichen Kanten gefunden, indem aus der Menge m aller Koordinaten der Ecken der Hindernisse und Lisas Haus alle zwei möglichen Endpunkte gefunden werden $\binom{m}{2}$.

Schnittpunkte mit der y-Achse In der Funktion **addIntersectionWithBus(self, points)** wird für jeden Punkt p (die Eckpunkte der Hindernisse und Lisas Haus) und der y-Achse der optimale Winkel w zum Anlaufen der Straße, der in **getAlpha(self, point)** mithilfe von Gleichung 14 berechnet wird, benutzt. Da die Gleichung nicht direkt lösbar ist, wird in einer for loop jeder Winkel in Grad zwischen 0 und 90 eingesetzt und der Winkel, der die geringste Differenz aufweist, zurückgegeben. Anschließend wird eine neue Strecke zwischen Punkt p und dem Punkt, an dem eine Strecke, die durch p geht und einen Winkel von w zur x-Achse hat, berechnet und gespeichert. Im weiteren Verlauf wird die Funktion **findIntersection(self, polygones, road)** benutzt, um auf die Klasse **FindIntersection** zuzugreifen, die überprüft, ob die oben berechnete Strecke (road) keine Kanten der Hindernisse schneidet. Sollte dies nicht der Fall sein, wird die neue Kante in einem Array mit allen anderen Kanten, die Lisas letzten Meter zum Bus repräsentieren, gespeichert. Sollte es jedoch vorkommen, dass das Anlaufen im optimalen Winkel nicht möglich ist, wird die Funktion **findNearestSpot(self, origin)** benutzt, die iterativ für jeden Winkel von 0° bis 90° überprüft, ob keine Hindernisse geschnitten werden. Alle Ergebnisse werden in einem Array gespeichert. Nun wird dieser Array durchlaufen, um die Strecke mit den Winkel, der am wenigsten von dem optimalen Winkel abweicht zu bestimen. Diese Strecke wird dann zurückgegen. Nachdem dies für alle Eckpunkte alle Hindernisse und Lisas Haus geschehen ist, wird ein Array mit allen neuen Kanten, die auf der y-Achse enden zurückgegeben.

Gültige Kanten Nun ist an der Zeit aus dem Array aller möglichen Kanten die Kanten, die für Lisa begehbar sind (keine Hindernisse schneiden), herauszufiltern. Dies geschieht in der Funktion **getPossibleLines(self, points, polygones, possibleEdges)**. Damit überprüft werden kann, ob ein Hindernis geschnitten wird, werden aus den Koordinaten der Polygone (*self.polygones*) alle Kanten eines Hindernisses in der Funktion **polyponToLines(self, polygon)** berechnet und in einem Array im Format $[(x_1, y_1), (x_2, y_2)]$ zurückgegeben. Die Klasse **FindIntersection** dient nun dazu für jede mögliche Kante (*possibleEdges*) zu überprüfen, ob kein Hinderniss geschnitten wird. Sollte dies der Fall sein, wird diese Kante im array *trueEdges* gespeichert, der nach dem Durchlaufen aller Kanten zurückgegeben wird.

Eliminierung der überflüßiger Kanten Alle Kanten, die von einer Ecke zu einer anderen Ecke des selben Hindernisses gehen werden im folgenden Verfahren in Betracht gezogen: Es gibt Kanten, die Diagonalen eines Hindernissen darstellen. Diese Kanten sind besonders, da sie in keinem Fall ein anderes Hinderniss kreuzen und deshalb nicht von der Funktion **getPossibleLines** erkannt werden. Es wird nun iterativ jede Kante durchlaufen, um das Hinderniss und die Position im Hindernis selbst (n-te Ecke) für die beiden Punkte, die die Kante definieren zu bestimmen. Zur Speicherung der Informationen wird der Array *used* benutzt. Nun werden die Informationend der beiden Punkte verglichen. Sollte es sich um Diagonalen handeln, werden sie nicht im finalen Array der gültigen Kanten enthalten sein. Bei einer Diagonalen würden beide Punkte im gleichen Hindernis enthalten sein, die Positionen wären aber nicht aufeinanderfolgend. Schließlich wird der Array mit allen gültigen Kanten zurückgegeben.

Der Graph Im abschließendem Schritt wird aus allen Kanten, die im oberen Teil berechnent worden waren, ein Graph erzeugt. Dazu wird folgendes Tupel kreiert: *Edge(start, end, cost)*. Nun wird in der Funktion **getEdges** für jede Kante die Länge berechnet und in einem Array gespeichert. Da dies ein kantengewichteter gerichteter Graph ist, muss für jede Kante $k = (a, b)$ die neue Kante $Edge(a, b, cost)$ und $Edge(b, a, cost)$ hinzugefügt werden. Nachdem dieser Graph erstellt worden ist, wird die Klasse **FinalPath** dazu verwendent, um für jeden Knoten, der auf der y-Achse liegt, den kürzesten Pfad zu finden. Darauffolgend wird der Zeitpunkt des Verlassen des Hauses bestimmt und der Pfad, der den spätesten Zeitpunkt garantiert, zurückgegeben.

Output Um den besten Weg anzuzeigen, eignet sich ein visuelles Ouput. Dazu wurde das Framework *svgwrite* benutzt, das es einfach macht ein .svg-file in Python zu schreiben. Dies geschiet in der Klasse **Draw**, wobei der rote Punkt Lisas Haus ist. Die weiteren Informationen, wie z.B. der Zeitpunkt, an dem sie spätestens das Haus verlassen muss, (*Departure*) werden in der Konsole ausgegeben. *path* ist ein Array mit allen Koordinaten der Punkte, an denen Lisas die Richtung ändert (Knoten des finalen Pfades). *Polygones* ist ein Array mit allen Koordinaten der Hindernisse.

3.2 Weitere Klassen

Graph.py In dieser Klasse wird der kürzeste Weg zwischen zwei beliebigen Knoten eines Graphen gefunden. Dazu wurde Dijkstras-Algorithmus[1] implementiert. Zuerst wird die Klasse mit einem Array von Kanten initialisiert. Das Format dieser Kanten sollte *Edge(start, end, cost)* sein. Anschließend erwartet die Funktion *dijkstra* die Parameter *source* (Lisas Haus) und *dest* (ein Knoten auf der y-Achse). Als erstes wird überprüft, ob die beiden Parameter überhaupt Knoten des Graphen sind. Sollte dies der Fall sein werden die Distanzen zu allen Knoten bis auf den Ursprung mit ∞ initialisiert. Solange es noch unbesuchte Knoten gibt, wählt der Algorithmus den Knoten k mit der kürzesten Distanz aus und speichert, dass dieser besucht wurde. Darauffolgend werden die Distanzen zu allen Nachbarknoten $\{n_1, n_2, ..., n_i\}$ berechnet, indem die Distanz vom Startknoten zum unbesuchten Knoten k und das Kantengewicht der Kante $(k, n_i$ addiert werden. Sollte diese Dsitanz kürzer als die bereits gespeichert sein, wird der Distanz Wert aktualiesiert und k als Vorgänger gespeichert. So kann der kürzeste Pfad zwischen zwei Knoten gefunden werden.

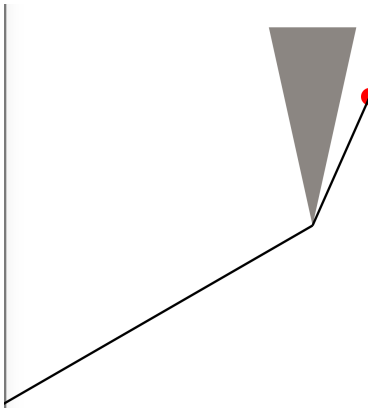
[1] Boldyreva, Maria. "Dijkstra's Algorithm in Python: Algorithms for Beginners." The Practical Dev, dev.to/mxl/dijkstras-algorithm-in-python-algorithms-for-beginners-dkc.

4 Beispiele

```
[Running] python -u "/Users/juliuside/Desktop/bwinfRound2/Aufgabe1/main.py"
Distance to the bus (in m):  859.0780712109558
Required time for Lisa to reach the bus (in s):  206.1787370906294
Departure:  07:28:00
Arrival at the bus : 07:31:26
Distance the bus has already driven when Lisa enters the bus (in m): 718.8823940164497
Path: [(633, 189), (535, 410), (0, 718.8823940164498)]
Polygones: [[[535, 410], [610, 70], [460, 70]]]

[Done] exited with code=0 in 0.345 seconds
```

(a)



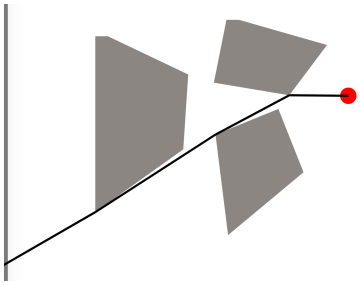
(b)

Abbildung 5: Beispiel 1

```
[Running] python -u "/Users/juliuside/Desktop/bwinfRound2/Aufgabe1/main.py"
Distance to the bus (in m):  719.0762746491316
Required time for Lisa to reach the bus (in s):  172.5783059157916
Departure:  07:28:07
Arrival at the bus : 07:31:00
Distance the bus has already driven when Lisa enters the bus (in m): 500.1495457622363
Path: [(633, 189), (525, 188), (390, 260), (170, 402), (0, 500.14954576223636)]
Polygones: [[[390, 260], [505, 213], [551, 329], [413, 444]], [[410, 50], [433, 50], [594, 96], [525, 188], [387, 165]], [[170, 80], [193, 80], [340, 150]
, [331, 287], [170, 402]]]

[Done] exited with code=0 in 0.311 seconds
```

(a)



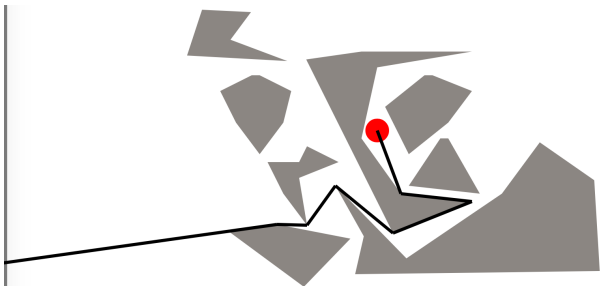
(b)

Abbildung 6: Beispiel 2

```
[Running] python -u "/Users/juliuside/Desktop/bwinfRound2/Aufgabe1/main.py"
Distance to the bus (in m):  833.2095612009809
Required time for Lisa to reach the bus (in s):  199.9702946882354
Departure:  07:27:20
Arrival at the bus : 07:30:40
Distance the bus has already driven when Lisa enters the bus (in m): 336.4703738152417
Path: [(479, 168), (509, 248), (599, 258), (499, 298), (426, 238), (390, 288), (352, 287), (0, 336.47037381524177)]
Polygones: [[[539, 98], [549, 98], [599, 118], [569, 158], [519, 198], [489, 138]], [[559, 178], [569, 178], [609, 248], [519, 238]], [[389, 78], [459,
68], [599, 68], [479, 88], [459, 178], [509, 248], [599, 258], [499, 298]], [[320, 98], [330, 98], [370, 118], [360, 158], [330, 198], [300, 158], [280,
118]], [[380, 208], [390, 188], [430, 208], [380, 228], [390, 288], [360, 248], [340, 208]], [[352, 287], [445, 305], [386, 366], [291, 296]], [[319, 18],
 [293, 53], [365, 80], [238, 73], [257, 15]], [[637, 248], [516, 330], [426, 238], [462, 302], [451, 350], [613, 348], [761, 346], [754, 231], [685, 183]]
]

[Done] exited with code=0 in 1.875 seconds
```

(a)



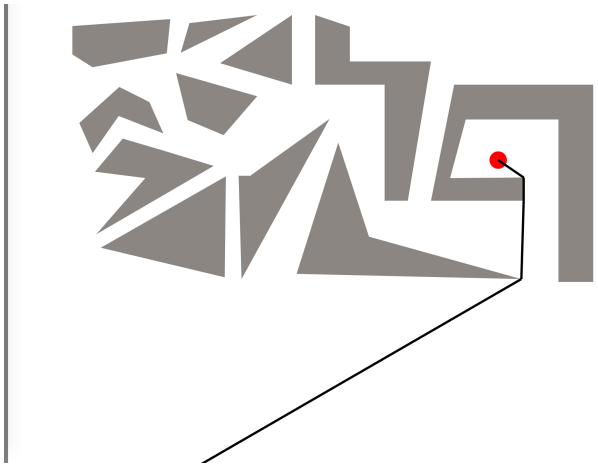
(b)

Abbildung 7: Beispiel 3

```
[Running] python -u "/Users/juliuside/Desktop/bwinfRound2/Aufgabe1/main.py"
Distance to the bus (in m):  1262.7721497810662
Required time for Lisa to reach the bus (in s):  303.0653159474559
Departure:  07:26:56
Arrival at the bus : 07:31:59
Distance the bus has already driven when Lisa enters the bus (in m): 992.3058411939044
Path: [(856, 270), (900, 300), (900, 340), (896, 475), (0, 992.3058411939045)]
Polygones: [[[121, 39], [290, 27], [284, 86], [156, 110], [121, 88]], [[133, 206], [202, 144], [254, 170], [278, 224], [201, 194], [156, 258]], [[160,
290], [247, 301], [162, 398], [365, 280], [276, 253], [208, 233]], [[170, 421], [386, 298], [384, 472]], [[408, 297], [428, 297], [565, 199], [413, 475]],
 [[300, 120], [440, 160], [382, 227], [320, 201]], [[323, 34], [440, 20], [308, 85]], [[500, 20], [500, 140], [376, 103]], [[540, 20], [600, 40], [600,
100], [740, 100], [700, 340], [660, 340], [660, 140], [540, 140]], [[580, 240], [633, 402], [896, 475], [508, 466]], [[780, 140], [1020, 140], [1020, 480]
, [960, 480], [960, 200], [800, 200], [773, 301], [900, 300], [900, 340], [740, 340]]]

[Done] exited with code=0 in 4.342 seconds
```

(a)



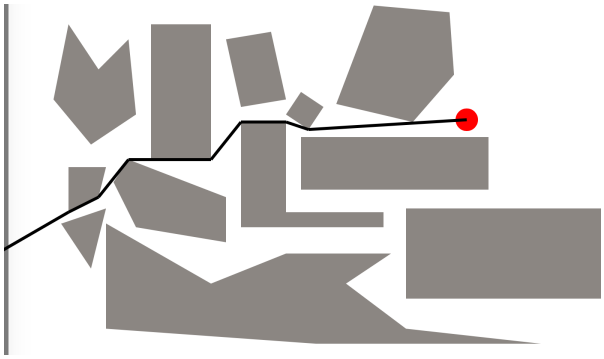
(b)

Abbildung 8: Beispiel 4

```
[Running] python -u "/Users/juliuside/Desktop/bwinfRound2/Aufgabe1/main.py"
Distance to the bus (in m):  689.2523534050948
Required time for Lisa to reach the bus (in s):  165.42056481722275
Departure:  07:27:55
Arrival at the bus : 07:30:40
Distance the bus has already driven when Lisa enters the bus (in m): 336.9615242270663
Path: [(621, 162), (410, 175), (380, 165), (320, 165), (280, 215), (170, 215), (130, 265), (90, 285), (0, 336.9615242270663)]
Polygones: [[[400, 185], [650, 185], [650, 255], [400, 255]], [[497, 10], [598, 19], [604, 102], [549, 165], [447, 141]], [[320, 165], [380, 165], [380,
285], [510, 285], [510, 305], [420, 305], [320, 305]], [[300, 55], [360, 45], [380, 135], [320, 145]], [[200, 35], [280, 35], [280, 215], [200, 215]], [
[170, 215], [300, 265], [300, 325], [180, 305], [150, 245]], [[90, 35], [130, 95], [170, 55], [180, 155], [120, 195], [70, 135]], [[400, 125], [430, 145],
 [410, 175], [380, 155]], [[90, 225], [140, 225], [130, 265], [90, 285]], [[540, 280], [541, 280], [800, 280], [800, 400], [540, 400]], [[380, 340], [381,
 340], [520, 340], [460, 380], [540, 440], [720, 460], [420, 460], [140, 440], [140, 300], [280, 380]], [[80, 300], [81, 300], [140, 280], [120, 360]]]

[Done] exited with code=0 in 2.987 seconds
```

(a)



(b)

Abbildung 9: Beispiel 5

5 Quellcode

```
from textToPolygones import TextToPolygones
from findIntersection import FindIntersection
from draw import Draw
from graph import Edge, Graph
from finalPath import FinalPath

from itertools import combinations
import numpy as np

import sys
import logging

class LisasRoute:
    def __init__(self, filename=None):
        if filename == None:
            filename = input("Please enter the filename:")
        self.filename = filename
        self.polygones = TextToPolygones(filename).getData()["polygones"]
        self.origin = TextToPolygones(filename).getData()["origin"]
        self.main()


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

        points = self.getRawPoints(self.polygones, self.origin)
        self.possibleEdges = list(combinations((points), 2))
        logger.info("possible edges")

        finalEdges = self.addIntersectionWithTheBus(points)
        logger.info("intersections with the bus")

        edges = self.getPossibleLines(points, self.polygones, self.possibleEdges)
        logger.info("final edges")

        edges = self.eliminateInnerEdges(self.polygones, edges)
        logger.info("eliminate inner edges")

        allEdges = finalEdges + edges
        self.graph = Graph((self.getEdges(allEdges)))
        logger.info("graph is created")

        path = FinalPath(finalEdges, self.graph, self.origin)
        logger.info("get final path")

        self.draw(path.edges)

        print("Distance to the bus (in m): ", path.distance)
        print("Required time for Lisa to reach the bus (in s): ", path.time)
        print("Departure: ", path.departure)
        print("Arrival at the bus :", path.arrival)
        print("Distance the bus has already driven when Lisa enters the bus (in m):", path.arrivalinM)
        print("Path:", path.path)
        self.path = path
        print("Polygones:", self.polygones)

    def draw(self, edges):
        # this function adds the edges to the graphical output
        drawing = Draw(self.polygones, self.origin,
                       self.filename.replace("txt", "svg"))
        drawing.edges(edges)

    # ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    # MARK :- the head functions

    def getEdges(self, allEdges):
        # this function returns the edges in a Edge(start, end, cost) format
        edges = []
```

```
        for e in allEdges:
            x = (abs(int(e[1][0]) - int(e[0][0]))) ^ 2
            y = (abs(int(e[1][1]) - int(e[0][1]))) ^ 2
            length = np.sqrt(x + y)

            edges.append(self.__makeEdge__(tuple(e[0]), tuple(e[1]), length))
            edges.append(self.__makeEdge__(tuple(e[1]), tuple(e[0]), length))
        return edges

    def __makeEdge__(self, start, end, cost):
        return Edge(start, end, cost)

    def getPossibleLines(self, points, polygones, possibleEdges):
        # this function returns all the possible edges where Lisa could effeciently walk
        lines = []

        for p in polygones:
            lines.append(self.polygonToLines(p))

        lines = self.reduceToOneList(lines)
        find = FindIntersection(lines)
        # a big list with all edges of polygones if created

        trueEdges = []
        # this section checks for every possible edge if it intersects with any other edge of any polyg
        for edge in possibleEdges:
            if find.isItPossible(list(edge)) == False:
                trueEdges.append(edge)

        # all the free paths where Lisa could walk from polygon to polygon are returned
        return trueEdges

    def eliminateInnerEdges(self, polygones, edges):
        # this function gets rid of lines within a polygon
        # they arent detetected by the algorithm because it does not intersect with any other polygones
        # input: polygones, edges as lists

        trueEdges = []
        used = [None] * 4  # this keeps track to what polygones the edges belong
        # for the first edge:    used[0] = number of polygones
        #                        used[1] = number of point
        # for the second edge:   used[2] = number of polygones
        #                        used[3] = number of point

        for edge in edges:
            if edge[0] == self.origin or edge[1] == self.origin:
                trueEdges.append(edge)
            else:
                for i in range(len(polygones)):
                    # iterates through every polygon
                    for j in range(len(polygones[i])):
                        # iterates through every point of a certain polygon
                        if edge[0] == polygones[i][j]:
                            used[0] = i
                            used[1] = j
                        if edge[1] == polygones[i][j]:
                            used[2] = i
                            used[3] = j

                if used[0] == used[2]:
                    polygon = polygones[used[0]]
                    priorNode = polygon[(used[3] - 1) % len(polygon)]
                    nextNode = polygon[(used[3] + 1) % len(polygon)]

                    if edge[0] == priorNode or edge[0] == nextNode:
                        # means it is not a diagonal
                        trueEdges.append(edge)
                else:
                    trueEdges.append(edge)

        return trueEdges

    def unique(self, edges):
```

```
145         # this function makes sure that the edges are unique
            # e.g. [3, 5], [6, 8] and [6, 8], [3, 5] are the same two lines
147
            true_edges = []
149         skip = False

151         for e in edges:
                for t in true_edges:
153                 if e[0] == t[1] and e[1] == t[0]:
                        skip = True
155             if skip == False:
                    true_edges.append(e)
157             skip = False
            return true_edges
159
    # ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
161 # MARK :- Helper methods

163 def getRawPoints(self, polygones, origin):
        new = []
165
        for a in polygones:
167         for b in a:
                new.append(b)
169     new.append(origin)
        return new
171
    def findIntersection(self, polygones, road):
173     lines = []

175     for p in polygones:
            lines.append(self.polygonToLines(p))
177
        lines = self.reduceToOneList(lines)
179     find = FindIntersection(lines)

181     return find.isItPossible(road)

183 def reduceToOneList(self, polygones):
        # this function breaks the huge list of lines of different polygones down into one list of lines
185     # input: polygones as a multidimensional list of lines
        lines = []
187     for p in polygones:
            for i in p:
189             lines.append(i)

191     return lines

193 def polygonToLines(self, polygon):
        # this function returns all the lines of one polygon
195
        lines = []
197     for i in range(len(polygon)):
            lines.append([polygon[i], polygon[(i + 1) % len(polygon)]])
199     return lines

201 # ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    # MARK:- this section looks for a possible direct way to the road for every point
203
    def addIntersectionWithTheBus(self, points):
205     newEdges = []
        for p in points:
207         alpha = self.__getAlpha__(p)
            line = self.__findLine__(alpha, p)
209
            if self.findIntersection(self.polygones, line) == False:
211             newEdges.append(line)
            else:
213             alternativeLine = self.__findNearestSpot__(p)
                if alternativeLine != None:
215                 newEdges.append(alternativeLine)
        return newEdges
217
```

```
    def __findNearestSpot__(self, origin):
        optimum = self.__getAlpha__(origin)
        validValues = []
        alphas = []

        for a in range(90):
            line = self.__findLine__(a, origin)
            if self.findIntersection(self.polygones, line) == False:
                validValues.append(abs(optimum - a))
                alphas.append(a)

        if (validValues != []):
            minimum = min(validValues)
            index = validValues.index(minimum)
            bestFit = alphas[index]

            return self.__findLine__(bestFit, origin)
        else:
            return None

    def __getAlpha__(self, point):
        # returns the best angle of starting running
        # bus: 30 km/h
        # lisa: 15 km/h
        # convert from meters to kilometers
        x = point[0]/1000 #form point
        y = point[1]/1000

        def left(x, y, a): return (y + np.tan(np.deg2rad(a)) * x) / 30
        def right(x, a): return x/np.cos(np.deg2rad(a)) / 15


        max = [None, None]

        for a in range(90):
            #value = left(x, y, a) - right(x, a)
            value = np.tan(np.deg2rad(a)) - 2 / np.cos(np.deg2rad(a)) - y/x

            if (max[0] == None or value > max[0]):
                max[0] = value
                max[1] = a


        return max[1]

    def __findLine__(self, angle, origin):
        # this function creates a lines between an input origin and the y-axis at a given angle
        # the format is [x1, y1], [x2, y2]
        line = []
        # first point origin
        line.append(origin)
        # second point intersection with y-axis
        line.append([0, origin[0] / np.cos(np.deg2rad(angle)) * np.sin(np.deg2rad(angle)) + origin[1]])

        return line
```

../bwinfRound2/Aufgabe1/lisasRoute.py

```
from collections import namedtuple, deque

inf = float('inf')
Edge = namedtuple('Edge', 'start, end, cost')

#just works with directed graphs

class Graph:
    def __init__(self, edges):
        if len(edges) > 1:
            for e in edges:
                if (type(e) is Edge) == False:
                    raise ValueError('Invalid edges data: {}'.format(edges))
            self.edges = edges
        else:
            raise AttributeError("Not enough edges: {}".format(edges))
```

```

    @property
    def vertices(self):
        return set(
            sum(
                ([edge.start, edge.end] for edge in self.edges), []
            )
        )

    @property
    def neighbours(self):
        neighbours = {vertex: set() for vertex in self.vertices}
        for edge in self.edges:
            neighbours[edge.start].add((edge.end, edge.cost))

        return neighbours

    def dijkstra(self, source, dest):
        source = tuple(source)
        dest = tuple(dest)

        assert source in self.vertices, 'Such source node doesn\'t exist'
        distances = {vertex: inf for vertex in self.vertices}
        previous_vertices = {
            vertex: None for vertex in self.vertices
        }
        distances[source] = 0
        vertices = self.vertices.copy()

        while vertices:
            current_vertex = min(
                vertices, key=lambda vertex: distances[vertex])
            vertices.remove(current_vertex)
            if distances[current_vertex] == inf:
                break
            for neighbour, cost in self.neighbours[current_vertex]:
                alternative_route = distances[current_vertex] + cost
                if alternative_route < distances[neighbour]:
                    distances[neighbour] = alternative_route
                    previous_vertices[neighbour] = current_vertex

        path, current_vertex = deque(), dest
        while previous_vertices[current_vertex] is not None:
            path.appendleft(current_vertex)
            current_vertex = previous_vertices[current_vertex]
        if path:
            path.appendleft(current_vertex)
        return path
```

../bwinfRound2/Aufgabe1/graph.py

```

#findIntersection.py class FindIntersection
#   -this class is meant to check if a possible path to the bus intersects with any obstacles

#   -it can check that if it receives a list of lines in [[x1, y1], [x2, y2]] format as an input

#   -these lines should be the borders of Lisa's obstacles (the polygones)


class FindIntersection:
    def __init__(self, lines):
        #because the lines / obstacles do not change we can store it like this
        self.lines = lines

    def isItPossible(self, road):
        #lines = all the lines
        #road = the two points where Lisa would like to walk
        #this function checks if a road does not cross any line

        for line in self.lines:
            if (line != road and line[0] != road[1] and line[0] != road[0]):
                if self.findIntersection(line, road):
                    return True
```

```
25          return False

27      def findIntersection(self, a, b):
            #this function will determine if two lines cross
29          #the format should be (x, y)
            #it returns true if two lines intersect
31
            functionA = self.__functionExpression__(a)
33          functionB = self.__functionExpression__(b)

35          if functionA == None or functionB == None:
                return None
37
            if functionA[0] == functionB[0]:
39              return False

41          #if two lines have a common point, they do not intersect
            if a[0] == b[0] or a[0] == b[1] or a[1] == b[0] or a[1] == b[1]:
43              return False

45          if len(functionA) == 2:
                if len(functionB) == 2:
47                  x = (functionB[1] - functionA[1]) / (functionA[0] - functionB[0])
                    if x >= a[0][0] and x <= a[1][0] and x >= b[0][0] and x <= b[1][0]:
49                      return True
                    else:
51                      return False
                else:
53                  return self.__infinitySlope__(functionB[0], b, functionA, a)
            else:
55              if len(functionB) == 2:
                    return self.__infinitySlope__(functionA[0], a, functionB, b)
57              else:
                    return False
59

61      def __infinitySlope__(self, x, a, function, b):
            #this function finds out if two lines interdect when one line goes vertically
63          #a has infinity slope (x)
            #b can be represented by a normal function
65          #input:
            #       x as a constant,
67          #       a as a line (x1, y1) and (x2, y2),
            #       function as a normal function (mx +c)(__functionExpression)
69
            # x = c for function a
71          y = function[0] * x + function[1]

73          if y > a[0][1] and y < a[1][1] or y > a[1][1] and y < a[0][1]:
                if x > b[0][0] and x < b[1][0] or x > b[1][0] and x < b[0][0]:
75                  return True
            else:
77              return False

79
        def __functionExpression__(self, line):
81          #this functions returns a function expression
            #input = (x1, y1) and (x2, y2)
83          #output = mx + c as a list [m, c]

85          if line[1][0] < line[0][0]:
                l = line[1]
87              line[1] = line[0]
                line[0] = l
89
            if line[1][0] == line[0][0]:
91              return [line[0][0]]

93          m = (line[1][1] - line[0][1]) / (line[1][0] - line[0][0])
            c = line[1][1] - m * line[1][0]
95
```

```
        return [m, c]
```

../bwinfRound2/Aufgabe1/findIntersection.py

```
import numpy as np
class FinalPath():
    """
        This class finds the best path for Lisa.
        It takes a graph, the origin and the edges that intersect with the y-axis
        and finds the path that lets her leave her house as late as possible.
    """

    def __init__(self, finalEdges, graph, origin):
        self.graph = graph
        self.origin = origin
        self.finalEdges = finalEdges
        self.shortestPath = self.__findShortestPath__()

    def __findShortestPath__(self):
        minimum = [None, None]

        for path in self.__pathToFinalEdges__(self.finalEdges):
            time = self.timeForAPath(path)

            if (minimum[0] == None or time >= minimum[0]):
                minimum[0] = time
                minimum[1] = path
        minimum[0] = self.__lengthOfAPath__(minimum[1])
        return (minimum)

    #++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    @property
    def path(self):
        return list(self.shortestPath[1])

    @property
    def edges(self):
        return self.pointsToEdges(self.shortestPath[1])

    @property
    def distance(self):
        return self.shortestPath[0]
    #++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

    @property
    def time(self):
        #returns the required time for Lisa to reach the bus if she constantly runs 15 km/h
        lisasVelocity = 15 #km/h
        mPS = lisasVelocity / 3.6 #meter per second
        requiredTime = self.shortestPath[0] / mPS

        return requiredTime

    def timeForAPath(self, path):
        self.shortestPath = [self.__lengthOfAPath__(path), path]
        return self.departure

    @property
    def arrivalinM(self):
        return self.__whenLisaEntersTheBus__(self.shortestPath[1]) * 30/3.6

    @property
    def arrival(self):
        time = self.__whenLisaEntersTheBus__(self.shortestPath[1])
        return self.__secToTime__(7*3600 + 30*60 + time)

    @property
    def departure(self):
        #returns the latest possible departure as a string, e.g '07:15:58'
        time = self.__whenLisaEntersTheBus__(self.shortestPath[1])
        busDeparture = 7*3600 + 30*60 + time #the bus' departure in seconds
        difference = busDeparture - self.time

        return self.__secToTime__(difference)
```

```
72      def __whenLisaEntersTheBus__(self, path):
            shortestPath = list(path)
74          whenLisaEntersTheBus = shortestPath[len(shortestPath) - 1][1]

76          velocityOfTheBus = 30 / 3.6 #in meters per seconds
            time = whenLisaEntersTheBus/velocityOfTheBus
78
            return time
80
        def __secToTime__(self, seconds):
82          hours = int(seconds / 3600)
            seconds -= hours * 3600
84          if hours < 10:
                hours = "0" + str(hours)
86
            min = int(seconds / 60)
88          seconds -= min * 60
            if min < 10:
90              min = "0" + str(min)

92          seconds = int(seconds)
            if seconds < 10:
94              seconds = "0" + str(seconds)

96          time = (str(hours) + ":" + str(min) + ":" + str(seconds))

98          return time

100     def __pathToFinalEdges__(self, finalEdges):
            paths = []
102
            for edge in finalEdges:
104             if edge[0][0] == 0:
                    paths.append(self.graph.dijkstra(self.origin, edge[0]))
106             elif edge[1][0] == 0:
                    paths.append(self.graph.dijkstra(self.origin, edge[1]))
108             else:
                    raise ValueError("Final␣edges␣have␣to␣intersect␣with␣the␣y-axis:␣{}".format(edge))
110
            return paths
112
        def __lengthOfAPath__(self, path):
114         distance = 0

116         for i in range(len(path) - 1):
                #(y2 - y1)^2 + (x2 - x1)^2 = edge
118             length = lambda a: (path[(i + 1) % len(path)][a] - path[i][a])
                # length(0) = x, length(1) = y
120             distance += np.sqrt(np.power(int(length(0)), 2) + np.power(int(length(1)), 2))

122         return distance

124     def pointsToEdges(self, path):
            edges = []
126         for i in range(len(path)):
                if i != (len(path) - 1):
128                 edges.append([path[i], path[(i + 1) % len(path)]])

130         return edges
```

../bwinfRound2/Aufgabe1/finalPath.py