Aufgabe 1: Lisa rennt

37. Bundeswettbewerb Informatik 2018/19 - 2. Runde

Agnes Totschnig

Teilnahme-Id: 48804

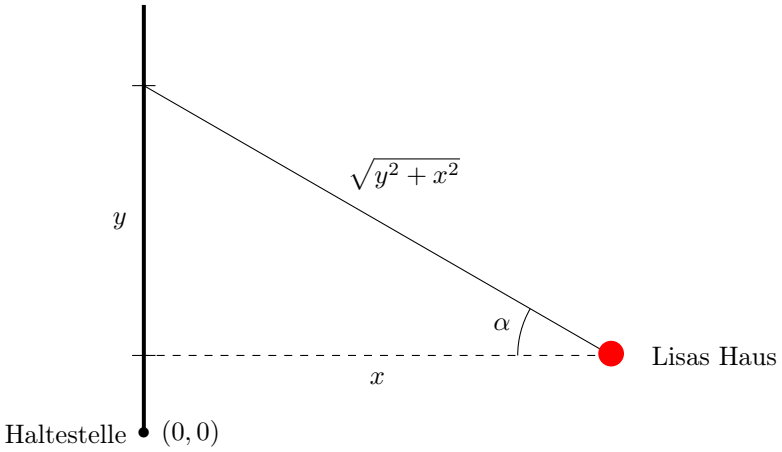
29. April 2019

Inhaltsverzeichnis

1 Lösungsidee

1.1 Route zur Landstraße ohne Hindernisse

Der Bus fährt um 7.30 Uhr von der Haltestelle am Punkt $(0, 0)$ ab. Da der Bus überall wo er vorbei kommt Lisa mitnehmen kann, stellt sich folgende Frage : Wo sollte Lisa auf der Landtstraße auf den Bus treffen, wenn es keine Hindernisse gibt, damit sie ihr Haus so spät wie möglich verlassen kann ?



Wir können zuerst ausschließen, dass Lisa in Richtung Süden läuft. Denn in diesem Fall könnte sie den gleichen Weg in Richtung Norden laufen, womit sie die selbe Strecke zurücklegt und erst später auf den Bus trifft. Nun stellt sich also die Frage ob Lisa horizontal oder schräg nach Norden laufen sollte, und in letzterem Fall welcher Winkel optimal wäre. Wenn Lisa weiter nördlich auf die Landtstraße trifft, wird der Anteil den sie laufen muss zwar größer, doch sie trifft erst später auf den Bus.

Um den optimalen Winkel α genau zu bestimmen, berechnen wir wieviel früher sie das Haus verlassen muss, wenn sie y Meter nördlich von ihrem Haus auf der Straße den Bus nehmen möchte. Wir ziehen die eingesparte Buszeit (im Vergleich zur horizontalen Route) von der Zeit, die Lisa braucht um zur Landstraße zu rennen, ab.

$$\begin{aligned} t(y) &= t_{\text{Lisa}} - t_{\text{Bus}} = \frac{d_{\text{Lisa}}}{15} - \frac{d_{\text{Bus}}}{30} \\ &= \frac{\sqrt{y^2+x^2}}{15} - \frac{y}{30} \\ &= \frac{1}{15} \cdot \left(\sqrt{y^2+x^2} - \frac{y}{2} \right) \end{aligned}$$

Wir differenzieren diese Funktion.

$$t'(y) = \frac{1}{15} \cdot \left(\frac{2y}{2\sqrt{y^2+x^2}} - \frac{1}{2} \right) = \frac{1}{15} \cdot \frac{2y - \sqrt{y^2+x^2}}{2\sqrt{y^2+x^2}}$$

Und suchen schließlich ihr Minimum.

$$\begin{aligned} t'(y) = 0 &\Leftrightarrow 2y - \sqrt{y^2+x^2} = 0 \\ &\Leftrightarrow 4y^2 = y^2 + x^2 \\ &\Leftrightarrow y = \frac{x}{\sqrt{3}} \end{aligned}$$

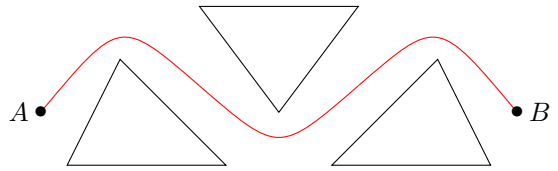
Somit beträgt der optimale Winkel α für Lisas Route zur Landstraße ohne Hindernisse :

$$\tan\alpha = \frac{\frac{x}{\sqrt{3}}}{x} = \frac{1}{\sqrt{3}} \qquad \text{also} \qquad \alpha = \tan^{-1}\frac{1}{\sqrt{3}} = \frac{\pi}{6} = 30°$$

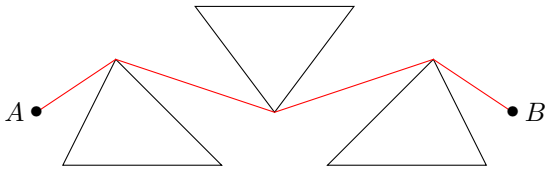
Diesen Winkel können wir nun auch in einem Feld mit Hindernissen anwenden, wenn die Landstraße von einem Eckpunkt eines Polygons aus direkt erreichbar ist.

1.2 Graph aller möglichen Verbindungen zwischen Eckpunkten

Bei den vielen Hindernissen wie Baustellen und Tümpel, muss Lisa viele Eckpunkte umlaufen. Dabei ist es optimal wenn Lisa geradlinig von Eckpunkt zu Eckpunkt läuft, wie es rechts im unten stehenden Beispiel zu sehen ist, denn der kürzeste Weg zwischen zwei Punkten ist immer die Gerade.



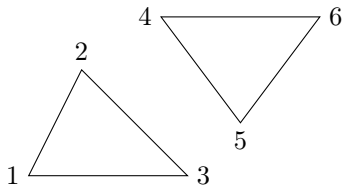
langer Weg von A nach B



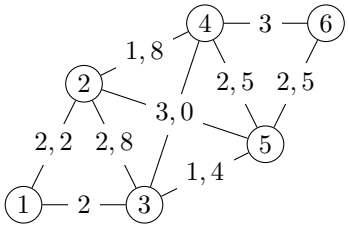
kürzester Weg von A nach B

Nun kann man aber von einem Eckpunkt nicht zu jedem anderen geradlinig laufen, denn es stehen Hindernisse im Weg. Wir stellen das Feld in der Form eines Graphen dar : die Knoten sind die Eckpunkte, die miteinander verbunden sind falls man geradlinig vom einem zum anderen laufen kann ohne auf ein Hindernis zu stoßen. Außerdem handelt es sich um einen gewichteten Graphen : jede Kante bekommt als Gewicht die Distanz zwischen den zwei Punkten.

In der Informatik stellt man einen solchen Graphen gewöhnlich mit einer Adjazenzmatrix dar. Der Eintrag in der i-ten Zeile und j-ten Spalte beträgt 0 falls die zwei i-te und j-te Knoten nicht verbunden sind und andernfalls gibt er das Gewicht der Kante zwischen den Punkten an.



Feld mit Hindernissen



Graph der möglichen Routen

$$\begin{bmatrix} 0 & 2,2 & 2 & 0 & 0 & 0 \\ 2,2 & 0 & 2,8 & 1,8 & 3,2 & 0 \\ 2 & 2,8 & 0 & 3,0 & 1,4 & 0 \\ 0 & 1,8 & 3,0 & 0 & 2,5 & 3 \\ 0 & 3,2 & 1,4 & 2,5 & 0 & 2,5 \\ 0 & 0 & 0 & 3 & 2,5 & 0 \end{bmatrix}$$

Adjazenzmatrix

In diesem Beispiel sind die Knoten 1 und 4 nicht miteinander verbunden, also kann man nicht direkt vom Punkt 1 zum Punkt 4 laufen. Man muss über den Punkt 2 oder 3 gehen.

1.2.1 Erstellung durch Rotierung

Um diese Matrix zu erstellen, wird für jedes Paar von zwei Eckpunkten überprüft ob eine Kante im Weg liegt. Sei n die Anzahl an Eckpunkten. Dann gibt es genau $n(n-1)$ solche Paare für die jeweils n Kanten überprüft werden müssen, womit sich eine Laufzeit von $\mathcal{O}(n^3)$ ergibt.

Dieser Algorithmus kann noch verbessert werden indem man die Anzahl an Kanten die jeweils überprüft werden müssen verringert. Von einem bestimmten Eckpunkt kann man genau die Punkte erreichen, die man sieht, ohne dass ein Hindernis davor steht. Um alle solche Punkte zu finden würde man als Person sich einmal rundum drehen um in alle Richtungen zu schauen. Auf dieser Idee basiert unser Algorithmus, der für einen bestimmten Punkt alle erreichbare Eckpunkte sucht.

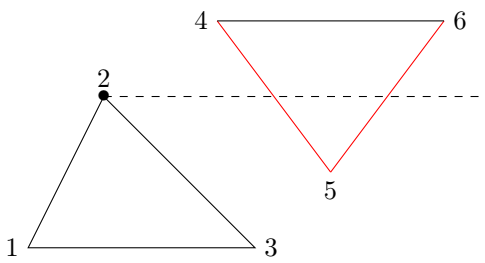
Algorithm 1 Rotierung

```
1:  procedure ROTIERUNG(Punkt P, Menge aller Punkte M)
2:      K ← Menge der Kanten die horizontal schneiden
3:      sortiere M nach aufsteigendem Winkel
4:
5:      for P_i in M do
6:          if sichtbar(P, P_i, K) then
7:              Graph[P, P_i] ← distanz(P, P_i)
8:          for K_i adjazente Kante von P_i do
9:              if K_i nicht in S then hinzufügen
10:             if K_i in S then entfernen
```

Wir schauen in eine bestimmte Anfangsrichtung, die wir als Osten festlegen. Nun wird erstmal überprüft auf welche Kanten wir hier horizontal stoßen würden. Anschließend werden die $n-1$ Punkte linksherum der Reihenfolge nach, so als würden wir uns nach links drehen, abgegangen. Dafür müssen also am Anfang alle Punkte nach aufsteigendem Winkel sortiert werden.

Bei jedem Punkt wird überprüft ob er von unserem Anfangspunkt aus sichtbar ist. Falls keine Kanten auf dem Weg liegen, wird die Distanz zwischen den zwei Eckpunkten im Graphen gespeichert. Außerdem wird geschaut ob wir auf neue Kanten stoßen, oder ob andere Kanten nicht mehr von Interesse sind. Dafür werden die adjazenten Kanten, d.h. die von diesem Punkt ausgehen, hinzugefügt falls sie noch nicht in der Menge sind, und entfernt falls sie schon enthalten sind.

Um schließlich die ganze Adjazenzmatrix zu erstellen, müssen wir diesen Algorithmus für jeden Eckpunkt aufrufen.



Wir schauen uns das am Beispiel des Punktes 2 an.

Am Anfang enthält die Menge K die zwei roten Kanten $(4,5)$ und $(5,6)$, denn sie schneiden die gestrichelte Halbgerade, die vom Punkt 2 aus nach Osten verläuft.

Die anderen Eckpunkte werden drehweise in dieser Reihenfolge abgegangen : 6, 4, 1, 3 und 5.

Die Iterationen des Algorithmus werden in folgender Tabelle zusammen gefasst.

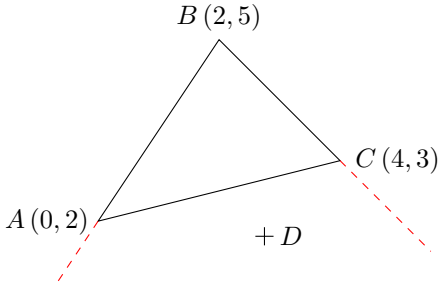
Eckpunkt	Kanten	entfernt	hinzugefügt	Graph
Anfange	$(4,5)$, $(5,6)$			
6	$(4,5)$, $(4,6)$	$(5,6)$	$(4,6)$	6 nicht sichtbar
4	$\emptyset$	$(4,5)$, $(4,6)$		Graph[2,4] = 1,8
1	$(1,3)$		$(1,3)$	Graph[2,1] = 2,2
3	$\emptyset$	$(1,3)$		Graph[2,3] = 2,8
5	$(4,5)$, $(5,6)$		$(4,5)$, $(5,6)$	Graph[2,5] = 3,2

Die Kanten, die vom Punkt 2 selbst ausgehen, werden nicht in die Menge der relevanten Kanten aufgenommen, denn sie stehen uns niemals im Weg. Doch alles was zwischen diesen zwei Kanten liegt kann natürlich nicht erreicht werden, da sich hier ein Hindernis befindet.

1.2.2 Überprüfung zwischen den eigenen Kanten

Wir müssen alles zwischen den zwei eigenen Kanten eines Eckpunktes ausschließen, denn da befindet sich ein Polygon. Nun liegt die Schwierigkeit aber dabei zu wissen in welcher Reihenfolge die Kanten angegeben sind. Deshalb müssen wir am Anfang überprüfen ob die Eckpunkte aller Polygone im Uhrzeigersinn angegeben werden. Dafür benutzen wir die Formel für die Fläche eines Polygons, die genau dann positif ist wenn die Eckpunkte gegen den Uhrzeigersinn angegeben werden.

$$\begin{aligned} A = {} & x_1y_2 - x_2y_1 \\ & + x_2y_3 - x_3y_2 \\ & \quad \dots \\ & + x_{n-1}y_n - x_ny_{n-1} \\ & + x_ny_1 - x_1y_n \end{aligned}$$



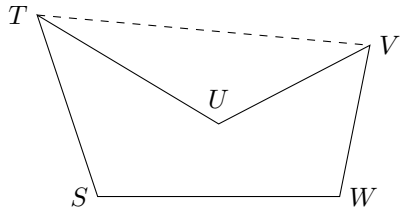
$$\begin{aligned} A(ABC) = {} & 0 \cdot 5 - 2 \cdot 2 \\ & + 2 \cdot 3 - 4 \cdot 5 \\ & + 4 \cdot 2 - 0 \cdot 3 = -10 \end{aligned}$$

$$\begin{aligned} A(CBA) = {} & 4 \cdot 5 - 2 \cdot 3 \\ & + 2 \cdot 2 - 0 \cdot 5 \\ & + 0 \cdot 3 - 4 \cdot 2 = 10 \end{aligned}$$

Wenn diese Summe positif ist, drehen wir die Reihenfolge der Eckpunkte um. So können wir also sicherstellen, dass alle Polygone im Uhrzeigersinn angegeben sind.

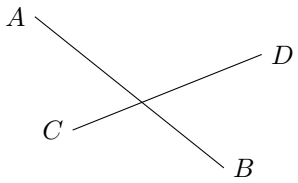
Nun müssen wir schließlich alles zwischen den Eckpunkten davor und danach ausschließen. So kann man vom Punkt B zum Beispiel nicht den Punkt D erreichen, denn er liegt zwischen A und C.

Man hätte sich hierfür auch überlegen können, dass nur die Punkte auf dem eigenen Polygon, mit Ausnahme für die zwei Nachbarn, einfach ausgeschlossen werden können. Denn die anderen Punkte, wie der Punkt D im oben stehendem Beispiel, werden von den übrigen Kanten, wie (A, C) blockiert. Doch bei konkaven Polygonen können auch andere Eckpunkte des selben Polygons erreicht werden, wie hier rechts zu sehen ist : T und V sind nicht benachbart und doch erreichbar.

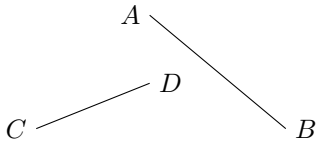


1.2.3 Überprüfen ob zwei Strecken sich kreuzen

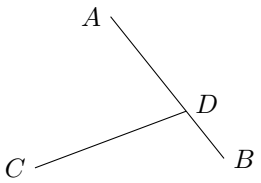
Wir wollen uns jetzt genauer anschauen wie die Methode `sichtbar` aussieht. Eine Kante $[CD]$ liegt genau dann im Weg zwischen A und B, wenn sich die zwei Strecken $[AB]$ und $[CD]$ kreuzen.



Fall 1: kreuzen sich



Fall 2: kreuzen sich nicht



Fall 3: berühren sich

Damit sich die zwei Strecken kreuzen, müssen A und B auf unterschiedlichen Seiten von $[CD]$ sein (siehe Fall 1). Doch wie hier im obigen Fall 2 gezeigt, reicht dies nicht aus : C und D müssen auch auf unterschiedlichen Seiten von $[AB]$ sein. Um zu bestimmen auf welcher Seite einer Strecke sich ein Punkt befindet benutzen wir das Zeichen des Kreuzproduktes.

$$\texttt{kreuzprodukt}(a, b, c, d) = (x_b - x_a) \cdot (y_d - y_c) - (x_d - x_c) \cdot (y_b - y_a)$$

Es müssen also $\texttt{kreuzprodukt}(C, D, C, A)$ und $\texttt{kreuzprodukt}(C, D, C, B)$, sowie $\texttt{kreuzprodukt}(A, B, A, C)$ und $\texttt{kreuzprodukt}(A, B, A, D)$, unterschiedliche Zeichen haben.

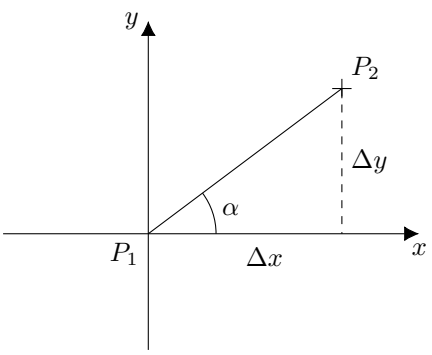
Wie sieht es aus wenn sich die zwei Strecken wie im Fall 3 berühren? Nun könnten wir das als nicht kreuzend passieren lassen, denn schließlich kann man ja an einem Eckpunkt vorbeilaufen. Das wäre jedoch problematisch, denn unser Progamm könnte es dann in manchen Fällen auch erlauben, dass man in ein Polygon hineinläuft. Daher schließen wir solche Fälle, wo das Kreuzprodukt Null beträgt, als kreuzend aus. Man kann immer noch an diesem Eckpunkt vorbeilaufen, nur halt nicht direkt.

Um letztendlich zu wissen ob ein Punkt B von einem Punkt A erreichbar ist, müssen wir diese Überprüfung für jede relevante Kante aufrufen. Nur wenn keine einzige im Weg liegt ist B für A sichtbar.

Diese Methode können wir auch für unsere Initialisierung der horizontal schneidenden Kanten benutzen. Sollte sich jedoch ein Punkt auf dem horizontalen Strahl befinden, dürfen wir diese Kante noch nicht als relevant markieren, da diese Kante sonst gleich wieder deaktiviert wird, wenn wir uns diesen Punkt anschauen. Deshalb müssen wir zusätzlich alle Kanten mit einem Punkt mit der selben y-Kordinate wie unser Ausgangspunkt, d.h. auf dem horizontalem Strahl, ausschließen.

1.2.4 Winkel berechnen

Um am Anfange die Punkte zu sortieren, müssen wir ihren Winkel zum horizontalem Strahl berechnen.



Wir berechnen den Winkel mit $\alpha = \tan^{-1} \frac{\Delta y}{\Delta x}$. Dies ist jedoch nicht möglich wenn $\Delta x = 0$, wofür wir $\alpha = \pm\frac{\pi}{2}$ setzen, je nachdem ob Δy positif oder negatif ist.

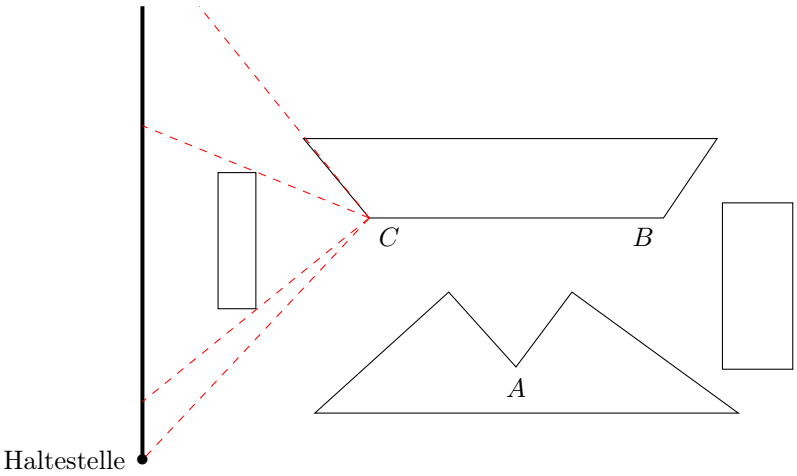
Des weiteren hat die Funktion tan die Periode π. Daher müssen wir bei allen Punkten auf der linken Seite der y-Axe, d.h. mit $\Delta x < 0$, den berechneten Winkel um π erhöhen.

Die Funktion $\tan^{-1}$ gibt Werte zwischen $-\frac{\pi}{2}$ und $\frac{\pi}{2}$ aus, doch wir wollen nur positive Werte zwischen 0 und 2π. Deshalb müssen wir schließlich noch bei den Punkten im vierten Quadranten den berechneten Winkel um 2π erhöhen.

1.2.5 Verbindung zur Landstraße

Letztendlich muss noch die letzte Spalte der Adjazenzmatrix ausgefüllt werden, nämlich der eventuelle Weg von jedem Eckpunkt zur Landstraße. Es gibt genau dann einen Ausblick aus dem Feld an Hindernissen wenn die Menge der relevanten Kanten leer ist. Damit dieser Ausblick auch zur Landstraße führt, muss er zwischen $\frac{\pi}{2}$ und dem Winkel w_H zur Haltestelle liegen, den wir für jeden Eckpunkt mit der Methode aus 1.2.4 berechnen können.

In dem unten stehendem Beispiel, gibt es vom Punkt A keinen Ausblick aus dem Feld an Hindernissen. Vom Punkt B gibt es einen Ausblick, jedoch führt er nicht zur Landstraße, denn der Winkel ist kleiner als $\frac{\pi}{2}$, d.h. er zeigt nur nach Osten. Vom Punkt C gibt es sogar mehrere Ausblicke zur Landstraße, von denen der nördlichere für Lisa besser geeignet ist, wie es in 1.1 erklärt wurde.



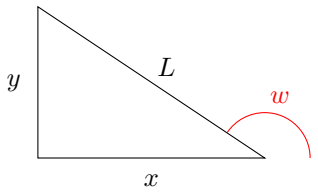
Ein Ausblick besteht aus einem Interval zwischen zwei Winkeln. So stellen wir uns die Frage : Welchen Winkel soll Lisa laufen? Wir haben in 1.1 gezeigt, dass es optimal ist mit einem Winkel von $\frac{\pi}{6}$ zur horizontalen Geraden zu laufen, was in unseren Polarkoodrinaten $\pi - \frac{\pi}{6} = \frac{5\pi}{6}$ entspricht. Dies ist leider nicht immer möglich. Wir müssen in solchen Fällen den nächst möglichen Winkel wählen, wie es folgender Algorithmus macht.

Algorithm 2 Optimaler Winkel

```
1: procedure OPTIMAL(Winkel w_1, Winkel w_2)
2:     if w_2 ⩽ π/2 oder w_1 > w_H then kein Zugang zur Landstraße
3:     else if w_1 > 5π/6 then optimal ← w_1        ▷ größer als optimaler Winkel
4:     else if w_2 < 5π/6 then optimal ← w_2        ▷ kleiner als optimaler Winkel
5:     else optimal ← 5π/6                          ▷ optimaler Winkel enthalten
```

Zu allererst stellen wir sicher, dass der Ausblick zwischen den zwei Winkeln überhaupt zur Landstraße führt. Falls der kleinere Winkel schon zu groß ist für den optimalen Weg, nehmen wir diesen, und falls der größere Winkel noch zu klein ist, nehmen wir letzteren. Sollte keiner der beiden Fälle zutreffen, bedeutet es, dass der optimale Weg in unserem Ausblick liegt und wir können diesen nehmen.

Schließlich müssen wir noch mit einer einfachen trigonometrischen Formel die Länge L vom Weg zur Straße, die vom gewählten Winkel w und der horizontalen Distanz x zur Straße abhängt, berechnen.



$$L = \frac{x}{\cos(\pi - w)} = -\frac{x}{\cos w}$$

Wenn wir einfach diese Distanz, die Lisa vom Eckpunkt zur Landtstraße laufen muss, in der Adjazenzmatrix speichern, vergessen wir, dass es vorteilhaft ist weiter nördlich auf die Straße zu kommen, da der Bus dort erst später vorbei fährt. Deshalb müssen wir die y-Kordinate von Lisas Treffen auf den Bus berechnen, nämlich $y = -x \cdot \tan w$, und die damit eingesparte Zeit berücksichtigen. Wir können zum Beispiel die in 1.1 beschriebene Differenz $t_{\text{Lisa}} - t_{\text{Bus}}$ für die letzte Spalte benutzen. Diese kan auch negativ sein, wenn viel Zeit eingespart wird oder wenn der Eckpunkt nahe zur Landstraße liegt.

1.3 Schnellste Route mit Dijkstra-Algorithmus

Um den schnellsten Weg zu finden wenden wir den Dijkstra-Algorithmus an. Dieser gibt uns den kürzesten Weg von Lisas Haus, unser Ursprungsknoten U, zu allen anderen Knoten im Graphen.

Der Algorithmus beruht auf folgender Tatsache : wenn wir den kürzesten Weg zwischen zwei Punkten suchen, ist der Teilweg zu allen vorherigen Punkten auch optimal, denn sonst könnte man diesen Teilweg noch optimieren. So können wir also vom Ursprung aus Stück für Stück längere optimale Wege bauen die auf einem kürzeren optimalen Weg beruhen.

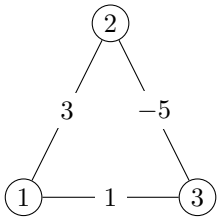
Algorithm 3 Dijkstra-Algorithmus

```
1:  procedure DIJKSTRA(Graph G, Knoten U)
2:      besucht ← []
3:      ursprung ← [0]× anzahlKnoten                    ▷ speichert Vorgänger
4:      for i = 1 to n do distanz[i] ← infinity
5:      distanz[U] ← 0
6:      while besucht ist nicht voll do
7:          K ← unbesuchter Knoten mit kleinster Distanz
8:          for unbesuchter Nachbar N von K in G do
9:              distanz[N] = min(distanz[N], distanz[K] + G[K, N])
10:             ursprung[N] = K
11:         besucht ← besucht ∪ {K}
```

Wir initialiseren die Distanzen zu allen Punkten als unendlich, außer für unseren Ursprung, der selbstverständlich die Distanz Null zu sich selbst hat. Bei jeder Iteration besuchen wir einen neuen Punkt, der mit der bisher kürzesten berechneten Distanz, die somit schon optimal ist. Von diesem Punkt aus schauen wir welche noch nicht optimalen Nachbarknoten wir besuchen können und ob sich ein kürzerer Weg über unseren aktuellen Knoten ergibt. Falls ja speichern wir diese neue Distanz zum Knoten. Wenn wir hiermit fertig sind, markieren wir diesen Knoten als besucht und gehen zum Nächsten weiter, bis alle Knoten besucht wurden.

In unserer Aufgabe wollen wir natürlich nicht nur wissen wie lang der kürzeste Weg ist, sondern auch wie Lisa laufen muss. Dafür speichern wir bei jedem Knoten den direkten Vorgänger, wenn wir einen neuen kürzeren Weg finden, in einer Liste. Anschließend können wir den kürzesten Weg zu jedem Punkt rekursiv wiederherstellen.

Schließlich stellt sich uns noch das Problem, dass dieser Algorithmus für negative Gewichte nicht funktioniert, wie es folgendes Beispiel erklärt.



Wir setzen 1 als Ursprung. Beim ersten Schritt werden die Punkte 2 und 3 erstmals jeweils mit den Distanzen 3 und 1 beschriftet. Nun widmet sich der Algorithmus dem nächsten unbesuchten Knoten zu. Hier ist es der Punkt 3, der somit als optimal abgeschlossen wird. Wir sehen jedoch, dass der Weg über den Punkt 2, nämlich $3 - 5 = -2$, kürzer als der direkte Weg zum Punkt 3 ist.

Dijkstras Algorithmus benutzt die Tatsache, dass Wege nur länger werden können wenn man Kanten hinzufügt. Mit negativen Gewichten ist dies jedoch nicht mehr der Fall.

Es können nur in der letzten Spalte unserer Adjazenzmatrix negative Werte vorkommen. Daher können wir mit Dijkstra schonmal den kürzesten Weg zu allen Eckpunkten finden. Anschließend müssen wir die von Lisa benötigte Zeit um diesen Weg zu laufen berechnen und die jeweilige Zeit (mit eingesparter Buszeit) zur Landstraße hinzufügen. Von dieser Liste können wir dann das kleinste Element, d.h. den schnellsten Weg, nehmen.

1.4 Laufzeitanalyse und Verbesserungen

Sei n die Anazhl an Eckpunkten. Die Laufzeitkomplexität von diesem Algorithmus hängt ab von der ausgewählten Methode um die Liste der Eckpunkte nach aufsteigendem Winkel zu sortieren. Die Standardfunktion in Python hat eine Laufzeit von $\mathcal{O}(n \log n)$. Zur Initialisierung der horizontal schneidenden Kanten müssen n Kanten überprüft werden. Dafür beträgt die Laufzeit also $\mathcal{O}(n)$, was die gesamte Komplexität nicht erhöht. Anschließend müssen noch alle Punkte durchgegangen werden und in konstanter Laufzeit untersucht. Da es $n-1$ solche Eckpunkte gibt, beträgt die Laufzeit hierfür auch $\mathcal{O}(n)$. Dieses Verfahren hat also eine Laufzeit von $\mathcal{O}(n \log n)$. Da es für jeden Eckpunkt wiederholt wird, beträgt die gesamte Laufzeit des Programms $\mathcal{O}(n^2 \log n)$. Diese Laufzeit ist besser als die benötigte $\mathcal{O}(n^3)$, wenn man für jedes Paar an Eckpunkten jede Kante überprüfen würde.

Zur weiteren Verbesserung des Algorithmus könnte man zuerst denken, dass es vielleicht reicht nur unbesuchte Eckpunkte zu überprüfen, denn wenn man die Distanz von A nach B hat, kennt man auch die Distanz von B nach A. Das ist aber leider nicht möglich, denn dann würde die Liste an relevanten Kanten beim Rotieren nicht richtig aktualisiert werden.

Literatur

[1] Steven S. Skiena, *The Algorithm Design Manual, Second Edition*, Kapitel 6 über gewichtete Graphen

[2] Vikas B. Patel, *Visualization of a plane sweep algorithm for construction of the visibility graph for robot path planning*, 1990, University of Nevada, Las Vegas

[3] Wikipedia-Artikel zur Gaußschen Trapezformel, `https://de.wikipedia.org/wiki/Gau\T1\sssche_Trapezformel`

[4] Wikipedia-Artikel zum Kreuzprodukt, `https://de.wikipedia.org/wiki/Kreuzprodukt`

2 Umsetzung

Das Programm ist in Python implementiert. Wir lesen zuerst die Koordinaten der Polygone ein, wobei wir wie in 1.2.2 beschrieben mit der Funktion `rechtsherum` sicherstellen, dass alle Eckpunkte im Uhrzeigersinn angegeben werden. Sollte das nicht der Fall sein, wird die Reihenfolge der Punkte mit `reverse` umgekehrt. In der Liste `knoten` werden alle Kordinaten der Eckpunkte, d.h. Knoten des Graphen, mit der ID des zugehörigen Polygons gespeichert. In der Liste `kanten` werden die Kanten als Tuple von zwei IDs von Eckpunkten gespeichert. Analog werden die Polygone in `polygone` gespeichert. Die IDs für die Eckpunkte werden, so wie für die Polygone, nach der Reihenfolge in der Eingabedatei zugewiesen. Diese ID entspricht auch der Position in der Liste `knoten`.

Im nächsten Abschnitt wir die Adjazentmatrix, die mit Nullen initialisiert wird, erstellt. Dafür werden folgende Funktionen aufgerufen :

— `winkel` berechnet den Winkel zu einem Punkt k_2 in dem Polarkoordinatensystem mit Ursprungspunkt k_1 wie in 1.2.4 beschrieben;

— `sichtbar` besagt ob ein Punkt k_2 von einem Punkt k_1 aus sichtbar ist oder ob eine Kante aus der relevanten Menge `kanten` den Weg versperrt. Hierfür wird anhand der Funktion `schneidet` für jede Kante überprüft ob sie die Streke zwischen den zwei Punkten kreuzt. Diese letztere ruft die mathematische Funktion `kreuzprodukt` auf;

— `horizontal` gibt alle hoizontal schneidende Kanten in einer Liste zurück. Hierfür geht sie ähnlich wie die Funktion `sichtbar` vor, jedoch mit dem Unterschied das anlehnende Kanten ausgeschlossen werden. Es handelt sich theoretisch um einen horizontalen Strahl, d.h. der in eine Direktion unendlich lang ist, da wir aber konkrete Punkte brauchen, lassen wir den *Strahl* etwas weiter als die größte x-Koordinate der Eckpunkte gehen;

— `distanz` berechnet die Distanz zwischen zwei Pukten anhand der Koordinaten, damit diese in der Matrix gespeichert werden kann;

— `verbunden` gibt die direkten Nachbarn eines Eckpunktes im Uhrzeigersinn zurück. So kann der unsichtbare Teil zwischen den eigen Kanten ausgeschlossen und die Liste der relevanten Kanten aktualisiert werden;

— `überprüfung` stellt sicher, dass sich der untersuchte Knoten nicht im unsichtbaren Teil zwischen den eigenen Kanten befindet. Hierbei muss auf den Fall geachtet werden wo der erste *kleinere* Winkel größer ist weil er nach Süden zeigt. In einem solchen Fall muss der Knoten nicht zwischen beiden Werten sein, sondern es reicht wenn er größer als der erste oder kleiner als der zweite Wert is um im unsichtbaren Bereich zu liegen;

— `optimalerWinkel` findet den optimalen Winkel in einem Ausblick zwischen zwei Werten, falls dieser überhaupt zur Landstraße führt, wie es in 2 beschrieben wurde. Es gibt die Differenz zwischen der benötigten und ersparten Zeit sowie die y-Kordinate des Treffens auf den Bus zurück.

Anschließend wird in der Matrix mit dem Dijkstra Algorithmus der kürzeste Weg zu allen Punkten berechnet. Dabei werden folgende Funktionen benutzt :

— `naechste` gibt den noch unbesuchten Knoten mit der kürzesten Distanz an;

— `weg` baut den Weg zu einem Punkt rekursiv anhand der Liste der direkten Ursprungsknoten wieder auf. Der globale Ursprungsknoten hat den Wert -1 wo diese Funktion stehen bleibt.

Für jeden Eckpunkt mit einem Ausblick zur Landstraße wird die gesamte Zeit mit Einsparnis anhand einer `lambda`-Funktion summiert. Von diesen Zeiten wird die kürzeste ausgesucht.

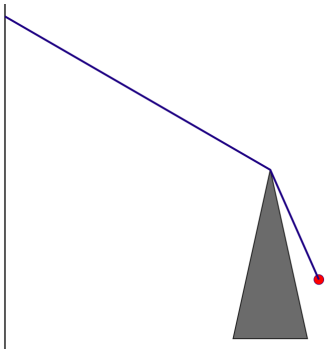
Schließlich werden noch die gefordeten Angaben zum optimalen Weg ausgegeben und eine svg-Datei zur Veranschaulichung des Weges erstellt. Die Zeit wird mit der Methode `zeitangabe` von Sekunden in Stunden, Minuten und Sekunden umgerechnet. Für die Startzeit und Zielzeit werden schließlich noch 7 Stunden und 30 Minuten hizugefügt.

3 Beispiele

Das erste Beispiel besteht aus einem einzelnen Dreieck. Da hat Lisa also zwei Möglichkeiten : im Norden oder im Süden daran vorbei laufen. Die folgende Programm Ausgabe zeigt, dass es besser ist über den nördlichen Eckpunkt zu laufen, da Lisa da erst später in den Bus steigen kann. Schließlich muss sie um 7 Uhr 27 Minuten und 60 Sekunden losgehen.

```
Dateiname : beispieldaten/lisarennt1
Startzeit : 7 : 27 : 60
Zielzeit : 7 : 31 : 26
y-Koordinate von Lisas Treffen auf den Bus : 719
Dauer :  3.43333333333 Minuten
Länge :  860 Meter
Koordinaten aller Eckpunkte der berechneten Route :
633 , 189 Lisas Haus
535 , 410 von Polygon P1
0 , 719 Landstraße
```

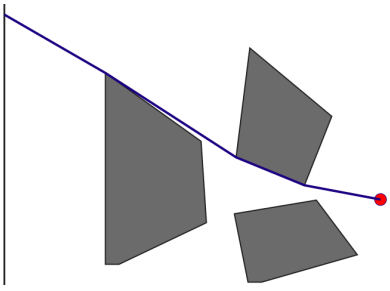
lisarennt1.txt



Im zweiten Beispiel geht der optimale Weg ähnlich in Richtung Norden. Der aller nördlichste Weg jedoch würde zwar erst später auf den Bus treffen aber Lisa müsste zu weit laufen.

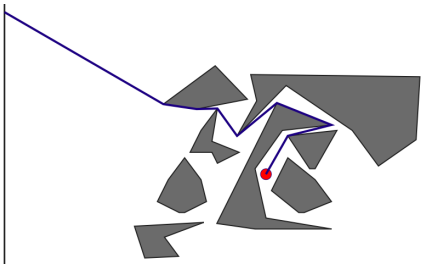
```
Dateiname : beispieldaten/lisarennt2
Startzeit : 7 : 28 : 9
Zielzeit : 7 : 31 : 0
y-Koordinate von Lisas Treffen auf den Bus : 500
Dauer :  2.85 Minuten
Länge :  713 Meter
Koordinaten aller Eckpunkte der berechneten Route :
633 , 189 Lisas Haus
505 , 213 von Polygon P1
390 , 260 von Polygon P1
170 , 402 von Polygon P3
0 , 500 Landstraße
```

lisarennt2.txt



Im dritten Beispiel muss Lisa erstmal nach Osten laufen um ein großes Hindernis zu umgehen.

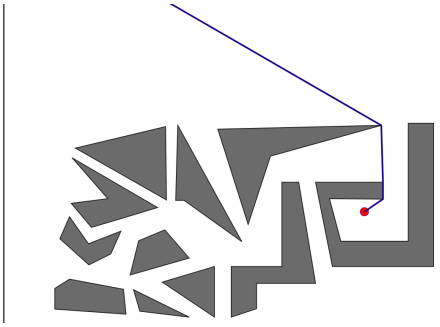
```
Dateiname : beispieldaten/lisarennt3
Startzeit : 7 : 27 : 29
Zielzeit : 7 : 30 : 56
y-Koordinate von Lisas Treffen auf den Bus : 464
Dauer :  3.45 Minuten
Länge :  863 Meter
Koordinaten aller Eckpunkte der berechneten Route :
479 , 168 Lisas Haus
519 , 238 von Polygon P2
599 , 258 von Polygon P3
499 , 298 von Polygon P3
426 , 238 von Polygon P8
390 , 288 von Polygon P5
352 , 287 von Polygon P6
291 , 296 von Polygon P6
0 , 464 Landstraße
```



lisarennt3.txt

Im vierten Beispiel lohnt es sich für Lisa gleich aus dem Feld an Hindernissen zu entkommen, da sie sonst zu weit nach Süden laufen müsste.

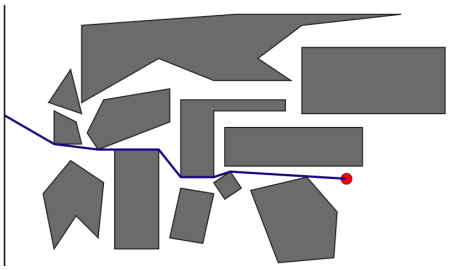
```
Dateiname : beispieldaten/lisarennt4
Startzeit : 7 : 26 : 56
Zielzeit : 7 : 31 : 59
y-Koordinate von Lisas Treffen auf den Bus : 992
Dauer :  5.05 Minuten
Länge :  1263 Meter
Koordinaten aller Eckpunkte der berechneten Route :
856 , 270 Lisas Haus
900 , 300 von Polygon P11
900 , 340 von Polygon P11
896 , 475 von Polygon P10
0 , 992 Landstraße
```



lisarennt4.txt

Auch im fünften Beispiel findet das Programm im Bruchteil einer Sekunde den optimalen Weg.

```
Dateiname : beispieldaten/lisarennt5
Startzeit : 7 : 27 : 54
Zielzeit : 7 : 30 : 33
y-Koordinate von Lisas Treffen auf den Bus : 277
Dauer :  2.65 Minuten
Länge :  662 Meter
Koordinaten aller Eckpunkte der berechneten Route :
621 , 162 Lisas Haus
410 , 175 von Polygon P8
380 , 165 von Polygon P3
320 , 165 von Polygon P3
280 , 215 von Polygon P5
200 , 215 von Polygon P5
170 , 215 von Polygon P6
90 , 225 von Polygon P9
0 , 277 Landstraße
```



lisarennt5.txt

4 Quellcode

```
import numpy as np
import math

dateiname = input("Dateiname : ")
datei = open(dateiname+".txt").read().split('\n')

# Überprüfung ob Eckpunkte rechtsherum angegeben werden
def rechtsherum(polygon) :
    summe = 0
    seiten = len(polygon)
    for i in range(seiten) :
```

```
        p1, p2 = knoten[polygon[i]], knoten[polygon[(i+1)%seiten]]
        summe += p1[0]*p2[1] - p2[0]*p1[1]
    if summe > 0 : # linksherum
        polygon.reverse()
    return polygon

# Einlesen der Polygone und der Koordinaten des Haus
anzahlPolygone = int(datei[0])
polygone = [0] * anzahlPolygone
knoten, kanten = [], []
bereits = 0
for i in range(anzahlPolygone) :
    eingabe = list(map(int, datei[i+1].split("␣")))
    seiten = eingabe[0]
    for j in range(seiten) :
        knoten.append([eingabe[2*j+1],eingabe[2*j+2],i])
        p1 = bereits + j
        p2 = bereits + (j+1) % seiten
        kanten.append(sorted([p1,p2]))
    polygone[i] = rechtsherum(list(range(bereits, bereits+seiten)))
    bereits += seiten
haus = list(map(int, datei[-1].split("␣")))
knoten.append(haus + [-1])
anzahlKnoten = len(knoten)
maxx = max(knoten, key=lambda x: x[0])[0]


# gibt zwei Nachbar-Eckpunkte auf dem selben Polygon zurück
def verbunden(k) :
    nachbar = []
    if k == anzahlKnoten-1 : # Haus hat keinen Nachbarn
        return nachbar
    polygon = polygone[knoten[k][2]]
    for i in range(len(polygon)) :
        if polygon[i] == k :
            nachbar = [polygon[i-1],polygon[(i+1)%len(polygon)]]
    return nachbar

# berechnet Winkel zwischen zwei Punkten zur horizontalen Gerade
def winkel(k1, k2) :
    if k1[0] == k2[0] :
        w = np.sign(k2[1]-k1[1]) * math.pi / 2
    else :
        w = math.atan((k2[1]-k1[1])/(k2[0]-k1[0]))
    if k1[0] > k2[0] :
        w += math.pi
    elif k1[1] > k2[1] :
        w += 2*math.pi
    return w

# brechnet mathematisches Kreuzprodukt
def kreuzprodukt(a,b,c,d) :
    return (b[0]-a[0])*(d[1]-c[1])-(d[0]-c[0])*(b[1]-a[1])

# überprüft ob zwei Strecken sich schneiden
def schneidet(a,b,c,d) :
    if np.sign(kreuzprodukt(c,a,c,d)) != np.sign(kreuzprodukt(c,b,c,d)) :
        if np.sign(kreuzprodukt(a,c,a,b)) != np.sign(kreuzprodukt(a,d,a,b)) :
            return True
    return False

# überprüft ob zwei Punkte für einander sichtbar sind
def sichtbar(k1,k2,kanten) :
    for kante in kanten :
        if not(k2 in kante) and schneidet(knoten[k1],knoten[k2],knoten[kante[0]],knoten[kante[1]]) :
            return False
    return True

# initialisiert die horizontal schneidenden Kanten
def horizontal(k) :
    horizontaleKanten = []
    for kante in kanten :
        if knoten[kante[0]][1] != knoten[k][1] and knoten[kante[1]][1] != knoten[k][1] :
```

```
            if schneidet(knoten[k],[maxx+1,knoten[k][1]],knoten[kante[0]],knoten[kante[1]]) :
                horizontaleKanten.append(kante)
    return horizontaleKanten

# berechnet die Distanz zwische zwei Punkten
def distanz(k1,k2) :
    d = math.sqrt((k2[0]-k1[0])**2 + (k2[1]-k1[1])**2)
    return d

# Überprüfung zwischen den eigenen kanten
def überprüfung(w,hindernis) :
    if len(hindernis) == 0 : # Haus hat keine Kanten
        return True
    if hindernis[0] > hindernis[1] :
        if hindernis[0] < w or w < hindernis[1] :
            return False
    if hindernis[0] < w and w < hindernis[1] :
        return False
    return True

# sucht optimalen Winkel zur Landstrasse
def optimalerWinkel(k,w1,w2) :
    haltestelle = winkel(k,[0,0])
    if w2 <= math.pi/2 or w1 > haltestelle : # kein Blick zur Landstrasse
        return [False, 0, 0]
    if w1 > 5*math.pi/6 :
        optimal = w1
    elif w2 < 5*math.pi/6 :
        optimal = w2
    else :
        optimal = 5*math.pi/6
    weg = -math.tan(optimal) * k[0]
    d = -k[0] / math.cos(optimal)
    zeit = 3.6 * (d/15 - (k[1]+weg)/30)
    return [True, zeit, k[1]+weg] # falls Weg zur Strasse, kürzeste Distanz zur Strasse

### Graph aller möglichen Verbindungen zwischen Eckpunkten ######################

graph = np.zeros((anzahlKnoten+1,anzahlKnoten+1))
strasse = [0] * anzahlKnoten
for k in range(anzahlKnoten) :
    andereKnoten = [[i,winkel(knoten[k],knoten[i])] for i in range(anzahlKnoten) if i != k]
    andereKnoten.sort(key=lambda x: x[1])
    relevanteKanten = horizontal(k)
    # überprüft ob Blick raus aus dem Feld an Hindernissen
    if len(relevanteKanten) == 0 :
        moeglich, zeit, weg = optimalerWinkel(knoten[k],0,andereKnoten[0][1])
        if moeglich :
            graph[k,anzahlKnoten] = zeit
            strasse[k] = weg
    hindernis = [winkel(knoten[k],knoten[i]) for i in verbunden(k)]
    for j in range(anzahlKnoten-1) :
        ki, wi = andereKnoten[j]
        if überprüfung(wi,hindernis) and sichtbar(k,ki,relevanteKanten) :
            graph[k,ki] = distanz(knoten[k],knoten[ki])
        # relevante Kanten aktualisieren
        eigeneKanten = [sorted([i,ki]) for i in verbunden(ki) if i!=k]
        for kante in eigeneKanten :
            if kante in relevanteKanten :
                relevanteKanten.remove(kante)
            else :
                relevanteKanten.append(kante)

        # überprüft ob Blick raus aus dem Feld an Hindernissen
        if len(relevanteKanten) == 0 :
            if j+1 < anzahlKnoten-1 :
                wmax = andereKnoten[j+1][1]
            else :
                wmax = 2* math.pi
            moeglich, zeit, weg = optimalerWinkel(knoten[k],wi,wmax)
            if moeglich and (graph[k,anzahlKnoten]>zeit or graph[k,anzahlKnoten]==0) :
                graph[k,anzahlKnoten] = zeit
```

```
158                 strasse[k] = weg

160
### Schnellster Weg mit Dijkstra ################################################
162
# wählt unbesuchten Knoten mit kleinster Distanz
164 def naechste(distanzen, besucht) :
    minimum = math.inf
166     for i in range(anzahlKnoten+1) :
        if distanzen[i] < minimum and besucht[i] == 0 :
168             minimum = distanzen[i]
            index = i
170     return index

172 # baut den optimalen Weg zu einem Punkt rekursiv wieder auf
def weg(i) :
174     if ursprung[i] == -1 :
        return [i]
176     return weg(ursprung[i])+[i]

178 besucht = [0] * (anzahlKnoten+1)
distanzen = [math.inf] * (anzahlKnoten+1)
180 distanzen[anzahlKnoten-1] = 0
ursprung = [-1] * (anzahlKnoten+1)
182 for i in range(anzahlKnoten+1) :
    punkt = naechste(distanzen, besucht)
184     besucht[punkt] = 1
    for j in range(anzahlKnoten+1) :
186         if graph[punkt,j]>0 and besucht[j]==0 and distanzen[j]>distanzen[punkt]+graph[punkt,j] :
            distanzen[j] = distanzen[punkt]+graph[punkt,j]
188             ursprung[j] = punkt

190 index = min([i for i in range(anzahlKnoten-1) if graph[i,-1] != 0], key=lambda x: 3.6*distanzen[x]/15 +
optimalerWeg = weg(index) + [anzahlKnoten]
192

194 ### Ausgabe und Schreiben der svg Datei ########################################
with open(dateiname+".svg", 'r') as datei :
196     graphik = datei.readlines()

198 polyline = '<polyline␣id␣=␣"R"␣points="'
for i in optimalerWeg[:-1] :
200     polyline += str(knoten[i][0]) + '␣' + str(knoten[i][1]) + '␣'
polyline += '0␣'+ str(strasse[optimalerWeg[-2]]) + '"␣fill="none"␣stroke="#000080"␣stroke-width="4"/>' +
202 graphik[-4] = polyline

204 with open(dateiname+".svg", 'w') as datei :
    datei.writelines(graphik)
206
def zeitangabe(name, zeit) :
208     stunden = int((30*60 + zeit) // 3600)
    minuten = int((zeit - stunden*3600) // 60)
210     sekunden = round(zeit - stunden*3600 - minuten*60)
    print(name, ":", 7+stunden, ":", 30+minuten, ":", int(sekunden))
212
letzter = optimalerWeg[-2]
214 laenge = distanzen[letzter] + distanz(knoten[letzter],[0,strasse[letzter]])
dauer = 3.6/15 * laenge
216 treffpunkt = strasse[letzter]
zielzeit = 3.6 * treffpunkt / 30
218 startzeit = zielzeit - dauer
zeitangabe("Startzeit", startzeit)
220 zeitangabe("Zielzeit", zielzeit)
print("y-Koordinate␣von␣Lisas␣Treffen␣auf␣den␣Bus␣:", round(treffpunkt))
222 print("Dauer␣:␣", round(dauer)/60, "Minuten")
print("Länge␣:␣", str(int(round(laenge))), "Meter")
224 print("Koordinaten␣aller␣Eckpunkte␣der␣berechneten␣Route␣:")
print(haus[0], ",", haus[1], "Lisas␣Haus")
226 for punkt in optimalerWeg[1:-1] :
    print(knoten[punkt][0], ",", knoten[punkt][1], "von␣Polygon␣P" + str(knoten[punkt][2]+1))
228 print("0␣,", round(treffpunkt), "Landstraße")
```

aufgabe1.py